深入C++系列

C++Primer

第三版

Stanley B Lippman
Josée Lajoie 著

中文版

潘爱民　张丽　译

Addison-Wesley

中国电力出版社
www.infopower.com.cn

译序

这是我心仪已久的一本书。我相信很多读者也有同样的感受。

在所有的编程语言中，C++可以说是最为复杂的。它既是一门传统的编程语言，也是一门新的编程语言。说它是一门传统语言，是因为 C++诞生已将近 20 年的历史了，特别是最近 10 年来 C++得到了快速的发展。C++是计算机软件领域中覆盖面最为广阔的编程语言。并且，与 C++相关的智力投入也是其他任何一门语言所无法比拟的。人们对于 C++的研究已经远远超出了对于一门编程语言所应有的关注。所以，现在的 C++已经非常成熟，有大量的资源（文档、书籍、源代码等等）可供我们使用。说 C++是一门新的编程语言，是因为在 1998 年 C++由 ISO（International Standards Organization）完成了标准化，从此 C++领域有了统一的标准，所有的编译器都将向标准靠拢（或者说，与标准兼容），这有利于我们写出可移植的 C++代码来。同时 C++标准也统一了 C++标准库，为 C++用户提供了最为基本的基础设施。C++经历了多年的发展，终于有了一个相对稳定的版本，所以，我们应该用一种新的眼光来看待 C++，而不再简单地把 C++认为是 C 语言的超集。本书正是新版本 C++的写照，通过本书，你可以重新审视 C++语言，这是我翻译过程中最为真切的体会，它纠正了我过去对于 C++语言的一些误解。虽然我从 1993 年开始就一直在使用 C++，但是直到阅读了这本书之后，我才从真正意义上全面地认识了 C++语言。

本书的权威性无需我多说，看看本书原著的前言，了解了两位作者的背景之后，你就可以知道，这本书是经验和标准的完美结合。Stanley Lippman 从 1984 年开始一直从事 C++方面的工作，在 C++的实现与应用方面有着丰富的经验。本书前两个版本的成功也证明了他在阐释 C++语言方面的独到之处。Josée Lajoie 从 1990 年开始成为 C++标准委员会的一名成员，并且承担了很重要的职务。由于她的参与，毫无疑问，本书一定是与标准兼容的。

讲述 C++的书非常多，并且不乏优秀和经典之作。在如此众多的 C++书籍中，本书仍具有不可替代的地位，我想主要的原因在于本书具有以下几个特色：

（1）内容广阔。从本书的规模（厚度）就可以看出这一点，C++语言融入了大量优秀的特性，其内容的丰富程度已经远非 C 语言所能及。在所有的 C++书籍中，本书的覆盖面是最为广阔的，从最基本的 C++程序设计，到面向对象程序设计，以及基于模板的程序设计，面面俱到，而且讲解细致入微，值得仔细品味。

（2）许多实际的范例程序。纯粹的技术讲解总是非常枯燥的，但是阅读本书并不感觉枯燥，因为作者在介绍每一部分内容的时候都结合一个实际的例子，读者通过这些例子能够很容易地掌握相应的技术要点，并且看到每一种技术的实际用法，这是本书之所以引人入胜的重要原因之一。

（3）叙述内容的安排。C++是一门多风格的程序设计语言（multi-paradigm Programming language），不仅支持面向对象程序设计，也支持其他的程序设计思想。本书的叙述结构正体现了 C++的这种特点，作者从程序设计思想的角度分别讲述了 C++的各种语言要素，便读者比较

容易抓住 C++语言的本质特征。

（4）与编译器无关，遵从 C++标准。本书的内容并不特定于某一个 C++编译器实现，而是适用于所有与 C++标准兼容的编译器。作者在讲解过程中也指出了编译器的一些内部考虑，例如，编译器如何在各种上下文环境中解析重载函数，如何处理除式类型转换，等等，这些内容有利于加深读者对 C++的理解。

（5）配套的练习。在每一节讲解之后，作者给出了一些练习，这些练习反映了这一节的中心内容，读者通过这些练习可以巩固所学的知识。所以，本书也可以被用作教材，用于系统全面地学习 C++语言。

虽然本书书名《C++ Primer》的中文含义是“C++初级读本”，但是它绝对不是一本很轻松的入门教材，特别是关于名字空间、函数重载解析过程、模板机制和泛型算法（generic algorithms）等内容并不是一个 C++初学者能够很快掌握的。如果你以前没有看过其他的 C++书籍，那么可能需要反复阅读多遍才能掌握本书讲述的内容：如果你已经有了 C++的基础（比如，已经看过其他的 C++入门书籍），那么阅读本书可以让你快速掌握 C++的要点：如果你是一名有多年 C++实践经验的程序员，那么阅读本书可以让你重新理解 C++。总之，这是一本很好的学习和参考书籍，值得你反复阅读。但是，正如书名所指示的，它不是一本高级书籍。按照我个人理解，它的技术水准应该在中等偏深一点的层次上。

本书的翻译工作由我和张丽共同完成，张丽完成了初稿的翻译工作，我做了第二遍翻译检查工作，书中每一句话我都认真检查过，个别地方还修改了原著的一些错误。C++中有些术语还没有统一的中文说法，对于这些术语的处理，我们尽可能地做到符合中文的语言习惯，读者可以参考本书最后所附的英汉对照索引。这份索引是由中国电力出版社的诸位编辑手工制作完成的。他们是刘江、朱恩从、陈维宁、程璐、关敏、刘君、夏平、宋宏、姚贵胜、常虹、乔晶、阎宏。感谢他（她）们的辛勤劳动。

在翻译过程中，不断收到读者来信或者来电询问这本书的出版情况。我理解读者对于一本好书的迫切心情，我的想法是，有关 C++的书籍和资料如此之多，所以，学习 C++不一定非要阅读这本书，但是它可以加快你学习的步伐，并且帮助你深入而全面地理解 C++。既然你已经看到了这本书，那就不要错过吧。

这本书不会让你失望的，我坚信这一点。

潘爱民

北京大学燕北园

前言

本书第二版和第三版之间的变化非常大。其中最值得注意的是，C++已经通过了国际标准化，这不但为语言增加了新的特性，比如异常处理、运行时刻类型识别（RTTI）、名字空间、内置布尔数据类型、新的强制转换方式，而且还大量修改并扩展了现有的特性，比如模板（template）、支持面向对象（object-oriented）和基于对象（object-based）程序设计所需要的类（class）机制。嵌套类型以及重载函数的解析机制。也许更重要的是，一个覆盖面非常广阔的库现在成了标准 C++的一部分，其中包括以前称为 STL（标准模板库）的内容。新的 string 类型、一组顺序和关联容器类型（比如 vector、list、map 和 set），以及在这些类型上进行操作的一组可扩展的泛型算法（generic algorithm），都是这个新标准库的特性。本书不但包括了许多新的资料，而且还阐述了怎样在 C++中进行程序设计的新的思考方法。简而言之，实际上。不但 C++已经被重新创造。本书第三版也是如此。

在第三版中，不但对语言的处理方式发生了根本的变化，而且作者本身也发生了变化：首先，我们的人数已经加倍。而且，我们的写作过程也已经国际化了（尽管我们还牢牢扎根于北美大陆）：Stan Lippman 是美国人，Josée Lajoie 是加拿大人。最后，这种双作者关系也反映了 C++团体的两类主要活动：Stan 现在正在迪斯尼动画公司（Walt Disney Feature Animation）*致力于以 C++为基础的 3D 计算机图形和动画应用，而 Josée 正专心于 C++的定义与实现，同时她也是 C++标准的核心语言小组的主席**，以及 IBM 加拿大实验室的 C++编译器组的成员。

Stan 是 Bell 实验室中与 Bjarne Stroustrup（C++的发明者）一起工作的早期成员之一。从 1984 年开始一直从事 C++方面的工作。Stan 曾经致力于原始 C++编译器 cfront 的各种实现，从 1986 年的版本 1.1 到版本 3.0，并领导了 2.1 和 3.0 版本的开发组。之后，他参与了 Stroustrup 领导的、Foundation Research Project 项目中关于程序设计环境的对象模型部分。

Josée 作为 IBM 加拿大实验室 C++编译器组的成员已经有八年时间了。从 1990 年开始她成为 C++标准委员会的成员。她曾经担任委员会的副主席三年，日前担任核心语言小组委员会的主席已经达四年之久。

本书第三版是一个大幅修订的版本，不仅反映了语言的变化和扩展，也反映了作者洞察力和经验的变化。

---

* Stan Lippman 现已受雇于 Microsoft，成为 Visual C++ .Net 的架构设计师。

** Josée Lajoie 现正在滑铁卢大学攻读硕士学位.已不再担任该委员会的主席。现任主席为 Sun 公司的 Steve Clamage。

本书的结构

本书为 C++国际标准进行了全面的介绍。在此意义上，它是一个初级读本（primer），它提供了一种指导性的方法来描述 C++语言。（但是，它也为 C++语言提供了一种简单而温和的描述，从这个角度来看，它不是一本初级读物。）C++语言的程序设计要素，比如异常处理、容器类型、面向对象的程序设计等等，都在解决特定问题或程序设计任务的上下文环境中展示出来。C++语言的规则，比如重载函数调用的解析过程以及在面向对象程序设计下支持的类型转换，本书都有广泛的论述，这似乎超出了一本初级读本的范畴。我们相信，为了加强读者对于 C++语言的理解，覆盖这些内容是必要的。对于这些材料，读者应该不时地回头翻阅，而不是一次消化了事。如果开始的时候你发现这些内容比较难以接受或者过于枯燥，请把它们放到一边，以后再回头来看——我们为这样的章节加上了特殊的记号：※。

阅读本书不需要具备 C 语言的知识，但是，熟悉某些现代的结构化语言会使学习进展更快一些。本书的意图是作为学习 C++的第一本书；而不是学习程序设计的第一本书！为了确保这一点，我们会以一个公共的词汇表作为开始；然而，开始的章节涵盖了一些基本的概念，比如循环语句和变量等，有些读者可能会觉得这些概念太浅显了。不必担心：深层的内容很快就会看到。

C++的许多威力来自于它对程序设计新方法的支持，以及对程序设计问题的思考方式。因此，要想有效地学习使用 C++，不要只想简单地学会一组新的语法和语义。为了使这种学习更加容易，本书将围绕一系列可扩展的例子来组织内容。这些例子被用来介绍各种语言特性的细节，同时也说明了这些语言特性的动机所在。当我们在一个完整例子的上下文环境中学习语言特性时，对这些特性为什么会有用处也就变得很清楚了，它会使我们对于“何时以及怎样在实际的问题解决过程中使用这些特性”有一些感觉。另外，把焦点放在例子上，可使读者能够尽早地使用一些概念，随着读者的知识基础被建立起来之后，这些概念会进一步完整地解释清楚。本书前面的例子含有 C++基本概念的简单用法，读者可以先领略一下 C++中程序设计的概貌，而不要求完全理解 C++程序设计和实现的细节。

第 1 章和第 2 章形成了一个独立完整的 C++介绍和概述。第一篇的目的是使我们快速地理解 C++支持的概念和语言设施，以及编写和执行一个程序所需要的基础知识。读完这部分内容之后，你应该对 C++语言有了一些认识，但是还谈不上真正理解 C++。这就够了：那是本书余下部分的目的。

第 1 章向我们介绍了语言的基本元素：内置数据类型、变量、表达式、语句以及函数。它将介绍一个最小的、合法的 C++程序，简要讨论编译程序的过程，介绍所谓的预处理器（preprocessor），以及对输入和输出的支持。它给出了多个简单但却完整的 C++程序，鼓励读者亲自编译并执行这些程序。第 2 章介绍了 C++是如何通过类机制，为基于对象和面向对象的程序设计提供支持的，同时通过数组抽象的演化过程来说明这些设计思想。另外，它简要介绍了模板、名字空间、异常处理，以及标准库为一般容器类型和泛型程序设计提供的支持。这一章的进度比较快，有些读者可能会觉得难以接受。如果是这样，我们建议你跳过这一章，以后再回过头来看它。

C++的基础是各种设施，它们使用户能够通过定义新的数据类型来扩展语言本身，这些

新类型可以具有与内置类型一样的灵活性和简单性。掌握这些设施的第一步是理解基本语言本身。第 3 章到第 6 章（第二篇）在这个层次上介绍了 C++语言。

第 3 章介绍了 C++语言预定义的内置和复合数据类型，以及 C++标准库提供的 string、complex、vector 类数据类型。这些类型构成了所有程序的基石。第 4 章详细讨论了 C++语言支持的表达式，比如算术、关系、赋值表达式。语句是 C++程序中最小的独立单元，它是第 5 章的主题。C++标准库提供的容器类型是第 6 章的焦点。我们不是简单地列出所有可用的操作，而是通过一个文本查询系统的实现，来说明这些容器类型的设计和用法。

第 7 章到第 12 章（第三篇）集中在 C++为基于过程化的程序设计所提供的支持上。第 7 章介绍 C++函数机制。函数封装了一组操作，它们通常形成一项单一的任务，如 print()。（名字后面的括号表明它是一个函数。）关于程序域和变量生命期的概念、以及名字空间设施的讨论是第 8 章的主题。第 9 章扩展了第 7 章中引入的关于函数的讨论，介绍了函数的重载。函数重载允许多个函数实例（它们提供一个公共的操作）共享一个公共的名字（但是，要求不同的实现代码）。例如，我们可以定义一组 print()函数来输出不同类型的数据。第 10 章介绍和说明函数模板的用法。函数模板为自动生成多个函数实例（可能是无限多个）提供了一种规范描述（prescription），这些函数实例的类型不同，但实现方式保持不变。

C++支持异常处理设施。异常表示的是一个没有预料到的程序行为，比如所有可用的程序内存耗尽。出现异常情况的程序部分会抛出一个异常——即程序的其他部分都可以访问到。程序中的某个函数必须捕获这个异常并做一些必要的动作。对于异常处理的讨论跨越了两章。第 11 章用一个简单的例子介绍了异常处理的基本语法和用法，该例子捕获和抛出一个类类型（class type）的异常。因为在我们的程序中，实际被处理的异常通常是一个面向对象类层次结构的类对象，所以，关于怎样抛出和处理异常的讨论一直继续到第 19 章，也就是在介绍面向对象程序设计之后。

第 12 章介绍标准库提供的泛型算法集合，看一看它们怎样和第 6 章的容器类型以及内置数组类型互相作用。这一章以一个使用泛型算法的程序设计作为开始。第 6 章介绍的 iterator（迭代器）在第 12 章将进一步讨论，因为它们为泛型算法与实际容器的绑定提供了粘合剂。这一章也介绍并解释了函数对象的概念。函数对象使我们能够为泛型算法中用到的操作符（比如等于或小于操作符）提供另一种可替换的语义。关于泛型算法在附录中有详细说明，并带有用法的示例。

第 13 章到第 16 章（第四篇）的焦点集中在基于对象的程序设计上——即创建独立的抽象数据类型的那些类设施的定义和用法。通过创建新的类型来描述问题域，C++允许程序员在写应用程序时可以不用关心各种乏味的簿记工作。应用程序的基本类型可以只被实现一次，而多次被重用，这使程序员能够将注意力集中在问题本身，而不是实现细节上。这些封装数据的设施可以极大地简化应用程序的后续维护和改进工作。

第 13 章集中在一般的类机制上：怎样定义一个类，信息隐藏的概念（即，把类的公有接口同私有实现分离），以及怎样定义并封装一个类的对象实例。这一章还有关于类域、嵌套类、类作为名字空间成员的讨论。

第 14 章详细讨论 C++为类对象的初始化、析构以及赋值而提供的特殊支持。为了支持这些特殊的行为，需要使用一些特殊的成员函数，分别是构造函数、析构函数和拷贝赋值操作符。这一章我们还将看一看按成员初始化和拷贝的主题（即指一个类对象被初始化为或者

赋值为该类的另一个对象），以及为了有效地支持按成员初始化和拷贝而提出的**命名返回值**（named return value）扩展。

第 15 章将介绍类特有的操作符重载，首先给出一般的概念和设计考虑，然后介绍一些特殊的操作符，如赋值、下标、调用以及类特有的 new 和 delete 操作符。这一章还介绍了类的友元（它对一个类具有特殊的访问特权）及其必要性。然后讨论用户定义的转换，包括底层的概念和用法的扩展实例。这一章还详细讨论了函数重载解析的规则，并带有代码示例说明。

类模板是第 16 章的主题。类模板是用来创建类的规范描述，其中的类包含一个或多个参数化的类型或值、例如，一个 vector 类可以对内含的元素类型进行参数化。一个 buffer 类，可以对内含的元素类型以及缓冲区的大小进行参数化。更复杂的用法，比如在分布式计算中，IPC 接口、寻址接口、同步接口等，都可以被参数化。这一章讨论了怎样定义类模板，怎样创建一个类模板特定类型的实例，怎样定义类模板的成员（成员函数、静态成员和嵌套类型），以及怎样用类模板来组织我们的程序。最后以一个扩展的类模板的例子作为结束。

面向对象的程序设计和 C++的支持机制是第 17、18、19 和 20 章（第五篇）的主题。第 17 章介绍了 C++对于面向对象程序设计主要要素的支持：继承和动态绑定。在面向对象的程序设计中。用父/子关系（也称类型/子类型关系）来定义“有共同行为的各个类”。类不用重新实现共享特性，它可以继承了父类的数据和操作。子类或者子类型只针对它与父类不同的地方进行设计。例如，我们可以定义一个父类 Employee，以及两个子类型：TemporaryEmpl 和 Manager。这些子类型继承了 Employee 的全部行为。它们只实现自己特有的行为。

继承的第二个方面，称为多态性，是指父类型具有“引用由它派生的任何子类型”的能力。例如，一个 Employee 可以指向自己的类型，也可以指向 TemporaryEmpl 或者 Manager。动态绑定是指“在运行时刻根据多态对象的实际类型来确定应该执行哪个操作”的解析能力，在 C++中，这是通过虚拟函数机制来处理的。

第 17 章介绍了面向对象程序设计的基本特性。这一章说明了如何设计和实现一个 Query 类层次结构，用来支持第 6 章实现的文本查询系统。

第 18 章介绍更为复杂的继承层次结构，多继承和虚拟继承机制使得这样的层次结构成为可能。这一章利用多继承和虚拟继承，把第 16 章的模板类例子扩展成一个三层的类模板层次结构。

第 19 章介绍 RTTI（运行时刻类型识别）设施。使用 RTTI 我们的程序在执行过程中可以查询一个多态类对象的类型。例如，我们可以询问一个 Employee 对象，它是否实际指向一个 Manager 类型。另外，第 19 章回顾了异常处理机制，讨论了标准库的异常类层次机构，并说明了如何定义和处理我们自己的异常类层次结构。这一章也深入讨论了在继承机制下重载函数的解析过程。

第 20 章详细说明了如何使用 C++的 iostream 输入/输出库。它通过例子说明了一般的数据输入和输出，说明了如何定义类特有的输入输出操作符实例、如何辨别和设置条件状态、如何对数据进行格式化。iostream 库是一个用虚拟继承和多继承实现的类层次结构。

本书以一个附录作为结束，附录给出了每个泛型算法的简短讨论和程序例子。这些算法按字母排序，以便参考。

最后，我们要说的是，无论谁写了一本书，他所省略掉的，往往与他所讲述的内容一样

重要。C++语言的某些方面，比如构造函数的工作细节、在什么条件下编译器会创建内部临时对象、或者对于效率的一般性考虑，虽然这些方面对于编写实际的应用程序非常重要，但是不适合于一本入门级的语言书籍。在开始写作本书第三版之前，Stan Lippman 写的《Inside the C++ Object Model》（参见本前言最后所附的参考文献中的［LIPPMAN96a］）包含了许多这方面的内容。当读者希望获得更详细的说明（特别是讨论基于对象和面向对象的程序设计）时，本书常常会引用该书中的讨论。

本书故意省略了 C++标准库中的某些部分，比如对本地化和算术运算库的支持。C++标准库非常广泛，要想介绍它的所有方面，则远远超出了本书的范围。在后面所附的参考文献中，某些书更详细地讨论了该库（见［MUSSER96］和［STROUSTRUP97］）。我们相信，在这本书出版之后，一定还会有更多的关于 C++标准库各个方面的书面世。

第三版的变化

本书第三版的变化主要是以下四个方面：

1.涵盖了语言所增加的新特性：异常处理、运行时刻类型识别、名字空间、内置 bool 类型、新风格的类型强制转换。

2.涵盖了新的 C++标准库。包括 complex 和 string 类型、auto_ptr 和 pair 类型、顺序容器和关联容器类型（主要是 list、vector、map、set 容器），以及泛型算法。

3.对原来的文字作了调整，以反映出标准 C++对原有语言特性的精炼、变化以及扩展。语言精炼的一个例子是，现在能够前向声明一个嵌套类型，这在以前是不允许的。语言变化的一个例子是，一个虚拟函数的派生类实例能够返回一个“基类实例的返回类型”的派生类。这种变化支持一个被称为 clone 或 factory 的方法［关于 clone()虚拟函数，见 17.4.7 节说明］。对原有语言特性进行扩展的一个例子是，现在可以显式地指定一个函数模板的一个或多个模板实参。（实际上，模板已经被大大地扩展了，差不多已经成为一个新特性！）

4.加强了对 C++高级特性的处理和组织方式，尤其是对于模板、类以及面向对象程序设计。这几年 Stan 从一个相对较小的 C++提供者团体转到了一般的 C++用户团体，这种影响使他相信，越是深入地了解问题，则程序员越是能够高明地使用 C++语言。因此，在第三版中，许多情况下，我们已经把焦点转移到如何更好地说明底层特性的概念，以及怎样最好地使用它们，并在适当的时候指出应该避免的潜在陷阱。

C++的未来

在出版这本书的时候，ISO/ANSI++标准委员会已经完成了 C++第一个国际标准的技术工作。该标准已于 1998 年的夏天由 ISO 公布。

C++标准公布之后，支持标准 C++的 C++编译器实现出将很快会推出。随着标准的公布，C++语言的进化将会稳定下来。这种稳定性使得以标准 C++编写的复杂的程序库，可以被用来解决工业界特有的问题。因此，在 C++世界中，我们将会看到越来越多的 C++程序库。

一旦标准被公布，标准委员会仍然会继续工作（当然步伐会慢下来），以解决 C++标准的用户所提出的解释请求。这会导致对 C++标准作一些细微的澄清和修正。如果需要，国际

标准将会每五年修订一次，以考虑技术的变化和工业界的需要。

C++标准公布五年之后将会发生什么事情我们还不知道。有可能是，一个被工业界广泛使用的新库组件将被加入到 C++标准库的组件集中。但是，到现在为止，面对 C++标准委员会已经完成的工作，C++的命运就全掌握在用户的手中了。

致谢

特别的感谢总是给予 Bjarne Stroustrup，感谢他给予我们如此美妙的编程语言，以及这些年他对我们的关心。特别感谢 C++标准委员会成员的奉献和艰苦工作（常常是自愿的），以及他们对 C++所作的贡献。

下面这些人为本书稿的各个草稿提供了许多有益的建议：Paul Abrahams、Michael Ball、Stephen Edwards、Cay Horstmann、Brian Kernighan、Tom Lyons、Robert Murray、Ed Scheibel、Roy Turner 和 Jon Wada。尤其要感谢 Michael Ball 的建议和鼓励。特别感谢 Clovis Tondo 和 Bruce Leung 为本书所作的深刻评论。

Stan 想把特别的感谢给予 Shyh-Chyuan Huang 和 Jinko Gotoh，感谢他们对 Firebird 给与的帮助和支持，当然还有 Jon Wada 和 Josée。

Josée 要感谢 Gabby Silberman、Karen Bennet 以及 IBM 高级研究中心（the Center for Advanced Studies）的其他小组成员，感谢他们为写作这本书所提供的支持。最大的感谢要给予 Stan，感谢他带着她经历了这项伟大的冒险活动。

最后，我们两个都要感谢编辑们的辛苦工作以及巨大的耐心：Debbie Lafferty，他从一开始，就为本书操劳；Mike Hendrickson 以及 John Fuller、Big Purple 公司在排版上做了精彩的工作。6.1 节的插图是 Elena Driskill 做的。非常感谢 Elena 使我们能够再版它。

第二版的致谢

这本书是无数看不见的手在帮助作者的结果。最由衷地感谢 Barbara Moo。她的鼓励、建议，以及对原手稿无数遍地仔细阅读已经无法评价。特别感谢 Bjarne Stroustrup 持续不断的帮助和鼓励，以及他给予我们如此美妙的编程语言；还有 Stephen Dewhurst，他在我第一次学习 C++时给了许多支持；以及 Nancy Wilkinson，另一位 cfront 编写者和 Gummi Bears 的提供者。

Dag Brück、Martin Carroll、William Hopkins、Brian Kernighan、Andrew Koenig、Alexis Layton 以及 Barbara Moo 提供了特别详细和敏锐的建议。他们的评论大大完善了这本书。Andy Baily、Phil Brown、James Coplien、Elizabeth Flanagan、David Jordan、Don Kretsch、Craig Rubin、Jonathan Shopiro、Judy Ward、Nancy Wilkinson 以及 Clay Wilson 检查了书稿的各种草稿，提出了许多有益的建议。David Prosser 澄清了 ANSI C 的许多问题。Jerry Schwarz，他实现了 iostream 包，提供了附录 A 所依据的原始文档（在第三版中变成了第 20 章）。非常感谢他对附录的细致检查。感谢 3.0 版本开发小组的其他成员：Laura Eaves、George Logothetis、Judy Ward、和 Nancy Wilkinson。

以下的人对 Addison-Wesley 的手稿作了评论：James Adcock、Steven Bellovin、Jon Forrest、

Maurice Herlihy、Norman Kerth、Darrell Long、Victor Milenkovic 以及 Justin Smith。

以下的人指出了第一版的各种印刷错误：David Beckedorff、Dag Bruck。John Eldridge、Jim Humelsine、Dave Jordan、Ami Kleinman、Andrew Koenig、Tim O'Konski、Clovis Tondo 和 Steve Vinoski。

我非常感谢 Brian Kernighan 和 Andrew Koenig 提供了大量可用的排版工具。

参考文献

[BOOCH94] Booch, Grady, *Object-Oriented Analysis and Design,* Benjamin/Cummings, RedwoodCity,CA (1994) ISBNO-8053-5340-2.

[GAMMA95] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns,* Addison Wesley Longman, Inc., Reading, MA (1995) ISBN 0- 201-63361-2.

[GHEZZI97] Ghezzi, Carlo, and Mehdi Jazayeri, *Programming Language Concepts, 3rd Edition,* John Wiley and Sons, New York, NY (1997) ISBN 0-471-10426-4.

[HARBISON88] Samuel P. Harbison and Guy L. Steele, Jr, *C: A Reference Manual, 3rd Edition,* Prentice-Hall, Englewood Cliffs, NJ (l988) ISBN 0-13-110933-2.

[ISO-C++97] Draft Proposed International Standard for Information Systems — *Programming Language C++ - Final Draft (FDIS) 14882*.

[KERNIGHAN88] Kernighan, Brian W., and Dennis M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, NJ (1988) ISBN 0-13-110362-8.

[KOENIG97] Koenig, Andrew, and Barbara Moo, *Ruminations on C++*, Addison Wesley Longman, Inc., Reading, MA (1997) lSBN 0-201-42339-l.

[LIPPMAN91] Lippman, Stanley, *C++ Primer, 2nd Edition,* Addison Wesley Longman, Inc., Reading, MA (1991) ISBN 0-201-54848-8.

[LIPPMAN96a] Lippman, Stanley, *Inside the C++ Object Model,* Addison Wesley Longman, Inc., Reading, MA (1996) ISBN 0-201-83454-5.

[LIPPMAN96b] Lippman, Stanley, Editor, *C++ Gems,* a SIGS Books imprint, Cambridge University Press, Cambridge,England (1996) ISBN0-13570581-9.

[MEYERS98] Meyers, Scott, *Effective C++, 2nd Edition,* Addison Wesley Longman, Inc., Reading, MA (1998) ISBN 0-201-92488-9.

[MEYERS96] Meyers, Scott, *More Effective* C++, Addison Wesley Longman, Inc., Reading, MA (1996) ISBN 0-201-63371-X.

[MURRAY93] Murray, Robert B., *C++ Strategies and Tactics,* Addison Wesley Longman, Inc., Reading, MA (1993) ISBN 0-20l-56382-7.

[MUSSER96J Musser, David R., and Atul Saini, *STL Tutorial and Reference Guide,* Addison Wesley Longman, Inc., Reading, MA (1996) ISBN 0-201-63398-l.

第一篇

C++概述

我们编写的程序由两个主要方面组成：

1．**算法**的集合。（就是将指令组织成程序来解决某个特定的问题。）

2．**数据**的集合。（算法在这些数据上操作，以提供问题的解决方案。）

纵观短暂的计算机发展史，这两个主要方面（算法和数据）一直保持不变。发展演化的是它们之间的关系，就是所谓的**程序设计方法**（programming paradigm）。

在**过程化程序设计方法**（procedural programming）中，一个问题可直接由一组算法来建立模型。例如，公共图书馆的资料借阅/登记（check out/check in）系统是由一系列过程表现出来的。其中两个主要的过程是资料的借阅和登记。这些数据被独立存储起来，我们既可以在某个全局位置上访问这些数据，或者把数据传递给过程以便它能够访问这些数据。Fortran、C 和 Pascal 是三种著名的过程语言。C++也支持过程化程序设计。单独的过程，如 check_in()。check_out()、over_due()、fine()等等，都被称为函数、第三篇将集中讨论 C++对过程化程序设计方法的支持，尤其将重点讨论函数、函数模板和通用算法。

在 20 世纪 70 年代，程序设计的焦点从过程化程序设计方法转移到了**抽象数据类型**（abstract data type，简写为 ADT）的程序设计上，现在通常称之为**基于对象**(object based）的程序设计。在基于对象的程序设计方法中，我们通过一组数据抽象来建立问题的模型，在C++中我们把这些抽象称为**类**（class）。例如，在这种方法下，图书馆资料借阅 / 登记系统就由类的对象实例（比如书、借阅者、还书时间、罚款等）之间的相互作用表现出来，以此表示出图书馆的抽象概念。与每个类相关的算法被称为该类的**公有接口**（public interface）。数据以私有形式被存储在每个对象中，对数据的访问应与一般的程序代码隔离开来。CLU、Ada 和 Modula-2 是三种支持抽象数据类型的程序设计语言。第四篇将说明和讨论 C++对抽象数据类型程序设计方法的支持。

面向对象的程序设计方法通过**继承**（inheritance）机制和**动态绑定**（dynamic binding）机制扩展了抽象数据类型；继承机制是对现有实现代码的重用，动态绑定是指对现有的公有接口的重用。以前独立的类型现在有了类型/子类型的特定关系，一本书、一盒录像带、一段录音，甚至孩子的宠物，尽管它们有各自的借阅/登记方式，但都可以成为图书馆的收藏资料。共享的公有接口和私有的数据都放在一个抽象类（图书馆资料，LibraryMaterial）中，每个特殊的图书馆资料类都从 LibraryMaterial 抽象类继承共享的行为，它们只需要提供与自身行为相关的算法和数据。Simula、Smalltalk 和 Java 是三种支持面向对象程序设计方法的著名语言。第五篇将集中讨论 C++对面向对象程序设计方法的支持。

C++是一种支持多种程序设计方法的语言。虽然我们主要把它当作面向对象的语言，但实际上它也提供对过程化的和基于对象的程序设计方法的支持。这样做的好处是对每个问题都能够提供最合适的解决方案。事实上。没有一种程序设计方法能够对所有的问题都提供最好的解决方案。这样做带来的缺点是使得语言过于庞大、复杂。

第一篇将对整个 C++进行快速浏览。这样做的一个原因是，它可以提供对语言特性的介绍，以便我们在完全面对这些特性之前，可以自由地引用语言的各个部分。例如，直到第 13 章我们才会详细介绍类，但如果到那时候才提起类，那么在此之前我们将不得不使用很多非典型的、不恰当的程序例子。

提供快速浏览的第二个原因是从美学的角度出发。除非首先让你领略到贝多芬交响曲的美丽与博大，否则，无关联的升半音、降半音、八度音符、和弦……等一定会让你厌烦。但是，只有掌握了这些细节，才有可能创作音乐。程序设计也一样，精通 C++程序设计的基础是首先要穿过操作符优先级或标准算术转换规则的迷宫，这样做既是必要的，也是非常枯燥的。

第 1 章将首先介绍 C++语言的基本元素，包括内置数据类型、变量、表达式。语句、函数。它将通过一个最小的，并且是合法的 C++程序，来讨论程序的编译过程、预处理、以及 C++对输入／输出的支持。这一章将给出多个简单但完整的 C++程序，鼓励读者亲自编译并执行这些程序。

第 2 章我们将浏览一个过程化程序、一个基于对象的程序和一个面向对象的程序，它们都实现了一个数组（一个由相同类型的元素组成的有限元素的集合）。然后，我们将这些程序中的数组抽象与 C++标准库中的向量（vector）类进行比较，同时也将首次介绍标准库中的通用算法。沿着这条路线，我们还将介绍 C++对异常处理、模板、名字空间的支持。实际上，这一章对整个 C++语言作了大致的介绍，细节部分将在以后各章节中详细介绍。

部分读者可能会感觉第 2 章很难理解，给出的许多资料没有初学者所期望的完整说明（这些细节在以后的章节中讨论）。如果你对某些细节部分感到吃力或失去耐心，建议略读或跳过它们，等到熟悉这些资料以后，再回头重读这些内容。第 3 章将以传统的叙述方式进行，建议对第 2 章不够适应的读者，从第 3 章开始。

1

开　始

本章介绍 C++语言的基本元素；包括内置数据类型、对象的定义、表达式、语句、函数的定义和使用。本章将给出一个最小的合法 C++程序，主要用它来讨论程序的编译过程、预处理，并将首次介绍 C++对输入／输出的支持。我们还将给出一些简单但完整的 C++程序。

1.1 问题的解决

程序常常是针对某些要解决的问题和任务而编写的。我们来看一个例子，某个书店将每本售出图书的书名和出版社，输入到一个文件中，这些信息以书售出的时间顺序输入，每两周店主将手工计算每本书的销售量，以及每个出版社的销售量。报表以出版社名称的字母顺序排列，以使下订单。现在，我们希望写一个程序来完成这项工作。

解决大问题的一种方法，是把它分解成许多小问题。理想情况下，这些小问题可以很容易地被解决。然后，再把它们合在一起，就可以解决大问题了。如果新分割的小问题解决起来还是太大，就把它分割得再小一些，重复整个过程，直到能够解决每个小问题。这个策略就是**分而治之**（divide and conquer）和**逐步求精**（stepwise refinement）。书店问题可以分解成四个子问题（或任务）：

1．读销售文件；

2．根据书名和出版社计算销售量；

3．以出版社名称对书名进行排序；

4．输出结果。

我们知道怎样解决第 1、2 和 4 个子问题，因此它们不需要进一步分解。但是，第 3 个子问题解决起来还是有些大，所以对这个子问题重复我们的做法，继续分解：

3a．按出版社排序；

3b．对每个出版社的书，按书名排序；

3c．在每个出版社的组中，比较相邻的书名，如果两者匹配，增加第一个的数量，删除第二个。

3a、3b 和 3c 所代表的问题。现在都已经能够解决了。由于我们能够解决这些子问题，

因此也就能够有效地解决原始的大问题了。而且，我们也知道任务的原始顺序是不正确的，正确的动作序列应该是：

1. 读销售文件；
2. 对文件排序——先按出版社，然后在出版社内部按书名排序；
3. 压缩重复的书名；
4. 将结果写入文件。

这个动作序列就是**算法**（algorithm）。下一步我们把算法转换成一种特定的程序设计语言——在这里是 C++语言。

1.2 C++程序

在 C++中，动作被称为**表达式**（expression）。以分号结尾的表达式被称作**语句**（statement）。C++中最小的程序单元是语句，在自然语言中，与此类似的结构就是句子。例如，下面是一组 C++语句：

```
int book_count = 0;
book_count = books_on_shelf + books_on_order;
cout << "The value of book_count: " << book_count;
```

第一条语句是一个**声明**（declaration）语句，book_count 被称为**标识符**（identifier）或**符号变量**（symbolic variable，简称变量），或者**对象**（object），它定义了计算机内存的一块区域，并且与名字 book_count 相关联，被用来存储整数值。0 是一个**文字常量**（literal constant），book_count 被初始化为 0。

第二条语句是一个**赋值**（assignment）语句，它把 books_on_shelf 和 books_on_order 的值
相加，并把结果放入与 book_count 相关联的计算机内存区域中。这里假定 books_on_shelf 和 books_on_order 已经在前面的代码中被声明为整型，并赋了初值。

第三条是**输出**（output）语句，cout 是与用户终端相关联的输出目标。<<是输出操作符，该语句向 cout（即用户终端）先输出用引号括起来的字符串文字，然后输出存储在与名字 books_count 相关联的内存区域中的值。假设此时 books_count 中的值为 11273，那么输出结果为：

```
the value of book_count: 11273
```

把语句按逻辑语义分组，就形成了一些有名字的单元，这些单元被称为**函数**（function）。例如，把所有需要读取销售文件的语句组织到一个被称为 readln()的函数中。类似地，我们可以构成 sort()、compact()和 print()函数。

在 C++中，每个程序必须包含一个被称作 main()的函数，它是由程序员提供的，并且只有这样的程序才能运行。下面是按前述算法定义的一种 main()函数：

```
int main()
{
    readIn();
    sort();
    compact();
```

```
    print();
    return 0;
}
```

C++程序从 main()函数的第一条语句开始执行，在本例中，程序从函数 readln()开始。并且程序按顺序执行 main()函数中的语句。在执行完 main()函数的最后一条语句之后，程序正常结束。

函数由四部分组成：返回类型、函数名、参数表，以及函数体。前三部分合起来称为**函数原型**（function prototype）。

参数表由小括号括起来，包含一个或多个由逗号分开的参数。函数体由一对花括号括起来，由程序语句序列构成。

在本例中，main()函数的函数体调用（invoke）函数 readIn()、sort()、compact()和 print(). 当这些函数调用都完成时，下面的语句：

```
return 0;
```

将被执行。return 是 C++预定义的语句，它提供了终止函数执行的一种方法。当 return 语句提供了一个值时，例如 0，这个值就成为函数的返回值（return value）。本例中，返回值为 0 表示 main()函数成功执行完毕。（标准 C++中，如果 main()函数没有显式地提供返回语句，则它缺省返回 0。）

现在我们来看一下，如果想让我们的程序能够执行起来，我们还需要做哪些准备工作。首先，必须提供函数 readln()、sort()。compact()以及 print()的定义。下面的哑函数实例已经足够满足这个要求了。

```
void readIn() { cout << "readIn()\n"; }
void sort() { cout << "sort()\n"; }
void compact() { cout << "compact()\n"; }
void print() { cout << "print()\n"; }
```

void 用来指定一个没有返回值的函数。正如上面的定义所示，每个函数在被 main()函数调用时只会简单地在用户终端上显示它的存在，以后，我们会用真正的实现函数来代替这些哑函数。

这种渐进式生成程序的方法，为控制程序设计中不可避免的错误，提供了一种有效的控制手段。试图一下子就能写出一个完全成功的程序，几乎是不可能的。

程序源文件的名字，一般包括两部分：文件名（例如 bookstore）以及文件后缀。文件后缀一般用来标识文件的内容。比如文件：

```
bookstore.h
```

在 C 或 C++中习惯上被称为头（header）文件。（标准 C++头文件没有后缀，这是个例外。）而以下文件：

```
bookstore.c
```

习惯上被当作 C 程序文本文件。但在 UNIX 操作系统中，以下文件：

```
bookstore.c
```

习惯上被当作 C++程序的文本文件。C++程序文件的后缀在不同的 C++实现产品中是不

同的，特别在 DOS 中大写的字母 C 与小写的字母 c 是不能区分的。其他常用来识别 C++程序文本文件的后缀还包括：

```
bookstore.cxx
bookstore.cpp
```

类似地，头文件的后缀在 C++的不同实现中也不相同（这也是标准 C++没有指定头文件后缀的一个原因）。请查阅你的编译器的用户指南，以确定在当前平台上使用什么后缀。

使用某个文本编辑器，将下面这段完整的程序输入到一个 C++文本文件中。

```
#include <iostream>
using namespace std;

void read() { cout << "read()\n"; }
void sort() { cout << "sort()\n"; }
void compact() { cout << "compact()\n"; }
void write() { cout << "write()\n"; }

int main()
{
    read();
    sort();
    compact();
    write();
    return 0;
}
```

iostream 是输入 / 输出流库标准文件（注意它没有后缀），它包含 cout 的信息，这对我们的程序是必需的。#include 是**预处理器指示符**（preprocessor directive），它把 iostream 的内容
读入我们的文本文件中（1.3 节将讨论预处理器指示符）。

在 C++标准库中定义的名字，如 cout，不能在程序中直接使用，除非在预处理器指示符：

```
#include <iostream>
```

后面加上语句：

```
using namespace std;
```

这条语句被称作 using **指示符**（using directive）。C++标准库中的名字都是在一个称作 std 的名字空间中声明的，这些名字在我们的程序文本文件中是不可见的，除非我们显式地使它们可见。using 指示符告诉编译器要使用在名字空间 std 中声明的名字。（在 2.7 和 2.8 节中将更进一步讨论名字空间与 using 指示符。）[1]

程序已经被输入到文件（如 prog1.c）中之后，接下来就应将其编译。在 Unix 系统中，按以下步骤进行（$表示系统提示符）。

```
$ CC prog1.C
```

用来调用 C++编译器的命令的名字，在不同的实现中也不相同（在 Windows 中，通常通

[1]在本书写作时（大约指 1997 年前后——译注），并不是所有的 C++实现都支持名字空间。如果你使用的 C++实现不支持名字空间，那么 using 指示符必须要忽略掉。因为本书的许多例子都是用不支持名字空间的 C++实现来编译的，所以绝大多数的代码例子都省略了 using 指示符。

过选择菜单项来调用命令)。CC 是 Unix 工作站上使用 C++编译器的命令名。可以通过参考手册或系统管理员获得系统的 C++命令名。

编译器的一部分工作是分析程序代码的正确性。编译器不能判断程序的语义是否正确，但能够判断出程序形式（form）上的错误，下面是两个常见的程序形式错误。

1．语法错误。程序员犯了 C++语言的语法错误。例如：

```
int main ( { // 错误：缺少)
     readIn(): // 错误：非法字符
     sort();
     compact();
     print();
     return 0 // 错误：缺少 ';'
}
```

2．类型错误。在 C++中，每个数据项都有一个相对应的数据类型，例如，10 是一个整型数值。由双引号括起来的词“hello”是一个字符串。如果为一个需要整型参数的函数提供了一个字符串，编译器就会报告类型错误。

错误消息包含一个行号以及编译器对错误的简要描述。按报告的顺序逐一修正错误，是个好的习惯。一个简单的错误常常有很多关联影响，会使编译器报告的错误比实际要多得多，因此，一旦错误被改正后，应当马上重新编译。这个循环过程通常被称为编辑一编译一调试（edit－compile－debug）。

编译器的第二部分工作是转换正确的程序代码。这种转换被称为代码生成（code generation）。典型情况下，它生成目标代码或汇编指令代码。这些代码能够被运行该程序的计算机所理解。

成功编译的结果是一个可执行文件。前面的程序执行时，其输出结果如下：

```
readln()
sort()
compact()
print()
```

C++定义了一组内置的基本数据类型：整数类型（int）、浮点数类型（float）、字符类型（char）、以及只有 false 和 true 两个值的布尔类型（boolean）。每种类型都与 C++语言中某一个关键字（keyword）相关联，程序中的每个对象都与一个特定的类型相关联。例如，下面的代码：

```
int age = 10;
double price = 19.99;
char delimiter = ' ';
bool found = false;
```

定义了四个对象：age、price、delimiter 和 found，分别是整数类型、双精度浮点数类型、字符类型和布尔类型。每个类型都赋予了一个文字常量初始值：整数 10、浮点数 19.99、空格字符、布尔值 false。

在内置类型之间经常发生隐式的类型转换（Conversion）。例如，将一个 double 双精度型的常量赋给一个 int 型的 age：

```
age = 33.333;
```

实际上，赋给 age 的是被截断后的整数值 33。[标准转换（standard conversion）以及一般类型的转换将在 4.14 节中详细讨论。]

C++标准库还提供了一组扩展的基本数据类型。其中包括字符串（string）、复数（complex number）、向量（vector）和列表（list）。例如：

```
// 为了使用 string 对象，下面的头文件是必需的
#include <string>
string current_chapter = "Getting Started";

// 为了使用 vector 对象，下面的头文件是必需的
#include <vector>
vector<string> chapter_titles( 20);
```

current_chapter 是一个字符串对象，被初始化为字符串文字“Getting Started”。chapter_title 是一个向量，包含有 20 个字符串类型的元素。以下这种特殊语法：

```
vector <string>
```

指示编译器创建一个能够存放字符串元素的向量类型。要定义一个能够存放 20 个整数的向量对象，我们可以这样写：

```
vector<int> ivec(20);
```

（本书对向量还将进行更多的描述。）

无论是一种语言还是它的标准库，都不可能提供实际程序设计环境要求的所有数据类型，因此，现代语言都提供了类型定义工具设施，使我们能够在语言中引入新的类型，这些类型的用法与内置类型的用法一样方便。在 C++中，这种设施就是类机制。在 C++中。像 string、complex、vector、list 这样的标准库类型都被设计成类。实际上，输入 / 输出库也是这样的。

类设施可能是 C++中最重要的组成部分，第 2 章对整个类机制作了基本的概述性介绍。

1.2.1 程序流程控制

缺省情况下，语句将按顺序执行。例如，在前面的程序中（重新列在下面），read()总是先被执行，然后是 sort()、compact()、write()。

```
int main()
{
    read();
    sort();
    compact();
    write();
    return 0;
}
```

然而，如果销售进展得很慢，例如，只有 0 或 1 项，那么就没有必要排序和压缩了，但是我们仍然需要输出这一项销售记录，或者指出没有销售记录产生。通过条件语句 if 我们可以完成这项工作。（假设已经重写了 readln()函数，使其能够返回读入的项数。）

```
// read()返回读入的项数
```

```
// 返回值的类型为 int
int read() { ... }
// ...
int main()
{
     int count = read();

     // 如果读入的项数大于 1
     // 就调用 sort()和 compact()

     if ( count > 1 ) {
          sort();
          compact();
     }
     if ( count == 0 )
          cout << "no sales for this month\n";
     else write();

     return 0;
}
```

第一个 if 语句给出了在括号中的条件表达式为真的情况下应该执行的动作。在这个被修改过的程序中，只有在 count 大于 1 的时候 sort()，compact()函数才会被调用。在第一个 if 语句中，为两个执行分支。如果条件为真（在这里，即如果 count 等于 0 ）则简单地输出没行销售产量，否则，只要 count 不等于 0，就调用 write()。（我们将在 5. 3 节中详细讨论 if 语句。）

第二种非顺序执行的语句是**迭代**（iterate）或称**循环**（loop）语句。当条件保持为真的时候，循环重复执行一条或多条语句。例如：

```
int main()
{
     int iterations = 0;
     bool continue_loop = true;
     while ( continue_loop != false )
     {
          iterations++;

          cout << "the while loop has executed "
               << iterations << " times\n";

          if ( iterations == 5 )
               continue_loop = false;
     }
     return 0;
}
```

在这个看似人为构造的例子中，while 循环执行 5 次，再到 iterations 等于 5 并且 continue_loop 被赋值为 false。如下语句：

```
iterations++;
```

将使 iterations 加 1。在 1.5 节中将有更实际的 while 循环的例子。第 15 章将详细讲解循环语句。

1.3 预处理器指示符

头文件通过 include **预处理器指示符**（preprocessor include directive）而成为我们程序的一部分。预处理器指示符用“#”号标识，这个符号将放在程序中该行的最起始一列上。处理这些指示符的程序被称做预处理器（preprocessor）（通常捆绑在编译器中）。

#include 指示符读入指定文件的内容，它有两种格式：

```
#include <some_file.h>
#include "my_file.h"
```

如果文件名用尖括号“<”和“>”括起来，表明这个文件是一个工程或标准头文件，查找过程会检查预定义的目录。我们可以通过设置搜索路径环境变量或命令行选项来修改这些目录。（在不同的平台上这些方法大不相同，建议你请教同事或查阅编译器手册以获得更进一步的信息。）如果文件名用一对引号括起来，则表明该文件是用户提供的头文件，查找该文件时将从当前文件目录开始。

被包含的文件还可以含有#include 指示符。由于嵌套包含文件的原因，一个头文件可能会被多次包含在一个源文件中。条件指示符可防止这种头文件的重复处理。例如：

```
#ifndef BOOKSTORE_H
#define BOOKSTORE_H
/* Bookstore.h 的内容 */
#endif
```

条件指示符#ifndef 检查 BOOKSTORE_H 在前面是否已经被定义。这里，BOOKSTORE_H 是一个预编译器常量。（习惯上预编译器常量往往被写成大写字母。）如果 BOOKSTORE_H 在前面没有被定义，则条件指示符的值为真，于是从#ifndef 到#endif 之间的所有语句都被包含进来进行处理。相反，如果#ifndef 指示符的值为假，则它与#endif 指示符之间的行将被忽略。

为了保证头文件只被处理一次，把如下#define 指示符：

```
#define BOOKSTORE_H
```

放在#ifndef 后面，这样在头文件的内容第一次被处理时，BOOKSTORE_H 将被定义，从而防止了在程序文本文件中以后#ifndef 指示符的值为真。

只要不存在“两个必须包含的头文件要检查一个同名的预处理器常量”这样的情形，这个策略就能够很好地运作。

#ifdef 指示符常被用来判断一个预处理器常量是否已被定义，以便有条件地包含程序代码。例如：

```
int main()
{
#ifdef DEBUG
cout << "Beginning execution of main()\n";
#endif
```

```
        string word;
        vector< string > text;
        while ( cin >> word )
        {
            #ifdef DEBUG
            cout << "word read: " << word << "\n";
            #endif
            text.push_back( word );
        }

        // ...
}
```

本例中，如果没有定义 DEBUG，实际被编译的程序代码如下：

```
int main()
{
        string word;
        vector< string > text;
        while ( cin >> word )
        {
            text.push_back( word );
        }
        // ...
}
```

反之，如果定义了 DEBUG，则传给编译器的程序代码是：

```
int main()
{
        cout << "Beginning execution of main()\n";
        string word;
        vector< string > text;
        while ( cin >> word )
        {
            cout << "word read: " << word << "\n";
            text.push_back( word );
        }
        // ...
}
```

我们在编译程序时可以使用-D 选项，并且在后面写上预处理器常量的名字，这样就能在命令行中定义预处理器常量：”[2]

```
$ CC -DDEBUG main.C
```

也可以在程序中用#define 指示符定义预处理器常量。

[2]对于 UNIX 系统，确实是这样的。Windows 程序员应该检查一下编译器的用户指南。

编译 C++程序时，编译器自动定义了一个预处理器名字__cplusplus（注意前面有两个下划线）。因此，我们可以根据它来判断该程序是否是 C++程序，以便有条件地包含一些代码。例如：

```
#ifdef __cplusplus
     // 不错，我们要编译 C++
     // extern "C" 到第 7 章再解释!
     extern "C"
#endif
int min( int, int );
```

在编译标准 C 时，编译器将自动定义名字__STDC__。当然，__cplusplus 与__STDC__不会同时被定义。

另外两个比较有用的预定义名字是：__LINE__和__FILE__。__LINE__记录文件已经被编译的行数，__FILE__包含正在被编译的文件的名字。可以这样使用它们：

```
if ( element_count == 0 )
cerr << "Error: " << __FILE__
     << " : line " << __LINE__
     << "element_count must be non-zero.\n";
```

另外两个预定义名字分别包含当前被编译文件的编译时间（__TIME__）和日期（__DATE__）。时间格式为 hh:mm:ss，因此如果在上午 8 点 17 分编译一个文件，则时间表示为 08:17:05。如果这一天是 1996 年 10 月 31 日，星期四，则日期表示为：

```
Oct 31 1996
```

若当前处理的行或文件发生变化，则__LINE__和__FILE__的值将分别被改变，其他四个预定义名字在编译期间保持不变。它们的值也不能被修改。

assert()是 C 语台标准库中提供的一个通用预处理器宏。在代码中常利用 assert()来判断一个必需的前提条件，以便程序能够正确执行。例如，假定我们要读入一个文本文件，并对其中的词进行排序，必需的前提条件是文件名已经提供给我们了，这样我们才能打开这个文件。为了使用 assert()，必须包含与之相关联的头文件：

```
#include <assert.h>
```

下面是一个简单的使用示例：

```
assert( filename != 0 );
```

assert()将测试 filename 不等于 0 的条件是否满足。这表示。为了后面的程序能够正确执行，我们必须断言一个必需的前提条件。如果这个条件为假（即：filename 等于 0），断言失败，则程序将输出诊断消息，然后终止。

assert.h 是 C 库头文件的 C 名字，C++程序可以通过 C 库的 C 名字或 C++名字来使用它。这个头文件的 C++名字是 cassert。C 库头文件的 C++名字总是以字母 C 开头，后面是去掉后缀.h 的 C 名字。（正如前面所解释的，由于在各种 C++实现中，头文件的后缀各不相同，因此标准 C++头文件没有指定后缀。）

使用头文件的 C 名字，或者 C++名字，两种情况下头文件的#include 预处理器指示符的效果也会不同。下面的#include 指示符：

```
#include <cassert>
```

将 cassert 的内容被读入到我们的文本文件中。但是由于所有的 C++库名字是在名字空间 std 中被定义的，因而在我们的程序文本文件中，它们是不可见的，除非用下面的 using 指示符显式地使其可见：

```
using namespace std;
```

使用 C 头文件的#include 指示符：

```
#include <assert.h>
```

就可以直接在程序文本文件中使用名字 assert()，而无需使用 using 指示符。[3]［库文件厂商用名字空间来控制全局名字空间污染（即名字冲突）问题，以避免它们的库“污染”了用户程序的名字空间。8.5 节将讨论这些细节。］

1.4 注释

注释是用来帮助程序员读程序的语言结构，它是一种程序礼仪，可以用来概括程序的算法、标识变量的意义、或者阐明一段比较难懂的程序代码。注释不会增加程序的可执行代码的长度。在代码生成以前，编译器会将注释从程序中剔除掉。

C++中有两种注释符号，一种是注释对（/*，*/），与 C 语言中的一样。注释的开始用/*标记，编译器会把/*与*/之间的代码当作注释。注释可以放在程序的任意位置，可以含有制表符（tab）、空格或换行，还可以跨越多行程序。例如：

```
/*
* 这是我们第一次看到（++的类定义
* 类可用于基于对象和
* 面向对象编程中，screen 类的
* 实现代码在第 13 章中
*/
class Screen {
      /* 此部分称为类体 */
      public:
      void home(); /* 将光标移到 0，0 */
      void refresh(); /* 重绘 Screen */

private:
      /* 类支持"信息隐藏" */
      /* 信息隐藏限制了程序 */
      /* 对类的内部表示（其数据）的 */
      /* 访问，这是用"private" */
      /* 来表示的。 */
      int height, width;
};
```

在代码中混杂过多的注释会使程序更难于理解。例如，注释几乎淹没了 width 和 height

[3] 在本书写作时，并不是所有的 C++实现都支持 C 库头文件的 C++名字，因为本书的许多例子是在不支持 C++头文件名的实现中编译的，所以有时候例子代码会用 C 名字引用 C 库头文件，而有时候用 C++ 名字引用 C 库头文件。

的声明。通常，把注释放在要描述的文本之上比较合适。与其他软件文档一样。考虑到有效性问题，注释必须随着软件的发展而升级。但是，注释与所描述的代码随时间推移而渐行渐远的情况却是经常发生的。

注释对不能嵌套，即一个注释对不能出现在另外一个注释对之中。请尝试在系统中编译下面的程序，它会使大多数编译器无法正常处理：

```
#include <iostream>
/*
* 注释对 /* */ 不能嵌套
* 这里“不能嵌套”几个字将被认为是代码
* 还包括接下来的这几行
*/
int main() {
     cout << "hello, world\n";
}
```

解决这种嵌套注释的一个办法是在星号和斜线之间加一个空格。

```
/ * * /
```

对于星号和斜线序列，只有当这两个字符之间没有被空格分割时，它们才被看作是注释符。

第二种注释符是双斜线（//），它可用来注释一个单行，程序行中注释符右边的内容都将被当作注释而被编译器忽略。例如，下面的 Screen 类使用了两种注释：

```
/*
* 这是我们第一次看到 C++的类定义
* 类可用于基于对象和
* 面向对象编程中，Screen 类的
* 实现代码在第 13 章中
*/
class Screen {

// 这部分被称为类体
public:
     void home(); // 将光标移至 0,0
     void refresh(); // 重绘 Screen

private:
     /* 类支持"信息隐藏" */
     /* 信息隐藏限制了程序 */
     /* 对类的内部表示（其数据）的 */
     /* 访问，这是用"private" */
     /* 来表示的 */
     // private 数据省略. . .
};
```

大多数程序往往包含两种格式的注释。多行的说明通常被放在注释对中，半行或单行的注释则由双斜线指出。

1.5 输入／输出初步

C++的输入/输出功能由输入/输出流（iostream）库提供。输入/输出流库是 C++中一个面向对象的类层次结构，也是标准库的一部分。

终端输入，也被称为标准输入（standard input），与预定义的 iostream 对象 cin（发音为 see-in）绑定在一起。直接向终端输出，也被称为标准输出（standard output），与预定义的 iostream 对象 cout（发音为 see-out）绑定在一起。第三个预定义 iostream 对象 cerr（发音为 see-err）称为标准错误（standard error），也与终端绑定。cerr 通常用来产生给程序用户的警告或错误信息。

任何要想使用 iostream 库的程序必须包含相关的系统头文件：

```
#include <iostream>
```

输出操作符<<用来将一个值导向到标准输出（cout）或标准错误（cerr）上。例如：

```
int v1, v2;
// ...
cout << "The sum of v1 + v2 = ";
cout << v1 + v2;
cout << '\n';
```

双字符序列“\n”表示换行符（newline）。输出换行符时，它结束当前的行，并将随后的输出导向到下一行。除了显式地使用换行符外，我们还可以使用预定义的 iostream 操纵符（manipulator）endl。

操纵符在 iostream 上执行的是一个操作，而不只是简单地提供数据。例如，endl 在输出流中插入一个换行符，然后刷新输出缓冲区。我们一般不写：

```
cout << '\n';
```

而是写成：

```
cout << endl;
```

（预定义 iostream 操纵符将在第 20 章中讨论。）

连续出现的输出操作符可以连接在一起，例如：

```
cout << "The sum of v1 + v2 = " << v1 + v2 << endl;
```

连续的输出操作符按顺序应用在 cout 上。为了便于阅读，连接在一起的输出操作符，可以分写在几行。下面的三行组成一条输出语句：

```
cout << "The sum of "
<< v1 << " + "
<< v2 << " = " << v1 + v2 << endl;
```

类似地，输入操作符（>>）用来从标准输入读入一个值。例如：

```
string file_name;
// ...
cout << "Please enter input and output file names: ";
cin >> file_name;
```

连续出现的输入操作符，也可以连接起来。例如：

```
string ifile, ofile;

// ...
cout << "Please enter input and output file names: ";
cin >> ifile >> ofile;
```

怎样读入未知个数的输入值呢？在 1.2 节结束的时候，我们已经做过。请看下面的代码序列：

```
string word;
while ( cin >> word )
// ...
```

在 while 循环中，每次迭代都从标准输入读入一个字符串，直到所有的串都读进来。当到达文件结束处（end-of-file）时，条件：

```
( cin >> word )
```

为假（第 20 章将解释这是如何发生的）。下面是使用这段代码序列的一个例子：

```
#include <iostream>
#include <string>

int main()
{
    string word;
    while ( cin >> word )
        cout >> "word read is: " >> word >> '\n';
    cout >> "ok: no more words to read: bye!\n";
    return 0;
}
```

下面是 James Joyce 的小说《Finnegans Wake》的前五个词：

```
riverrun, past Eve and Adam's
```

从键盘上输入这些词，程序的输出是：

```
word read is: riverrun,
word read is: past
word read is: Eve
word read is: and
word read is: Adam's
word read is: ok: no more words to read: bye!
```

（在第 6 章，我们将会看到怎样从各种输入字符串中删除标点符号。）

1.5.1 文件输入和输出

iostream 库也支持文件的输入和输出。所有能应用在标准输入和输出上的操作符，也都可以应用到已经被打开的输入或输出（或两者兼有）文件上。为了打开一个文件供输入或输出，除了 iostream 头文件外，还必须包含头文件：

```
#include <fstream>
```

为了打开一个输出文件，我们必须声明一个 ofstream 类型的对象：

```
ofstream outfile( "name-of-file" );
```

为了测试是否已经成功地打开了一个文件，我们可以写出这样的代码：

```
// 如文件不能打开值为 false
if ( ! outfile )
     cerr << "Sorry! We were unable to open the file!\n";
```

类似地，为了打开一个文件供输入，我们必须声明一个 ifstream 类型的对象：

```
ifstream infile( "name of file" );
if ( ! infile )
     cerr << "Sorry! We were unable to open the file!\n";
```

下面是一个简单的程序。它从一个名为 in_file 的文本文件中读取单词，然后把每个词写到一个名为 out_file 的输出文件中，并且每个词之间用空格分开。

```
#include <iostream>
#include <fstream>
#include <string>

int main()
{
     ofstream outfile( "out_file" );
     ifstream infile( "in_file" );
     if ( ! infile ) {
          cerr << "error: unable to open input file!\n";
          return -1;
     }
     if ( ! outfile ) {
          cerr << "error: unable to open output file!\n";
          return -2;
     }
     string word;
     while ( infile >> word )
          outfile << word << ' ';

     return 0;
}
```

第 20 章将对 iostream 库进行全面的讨论，包括文件输入和输出。现在，我们对 C++提供的内容有了大致的了解，下一步，我们将通过使用类和模板设施把新的类型引入到语言中。

2

C++浏览

本章将首先讲述 C++对数组类型的支持。数组是相同类型元素的集合，例如整型数组，可能代表考试的分数；或者字符串数组；可能代表在文本文件中包含的单词。然后，我们会看一看内置数组类型的缺点，以及怎样通过提供一个基于对象的 Array 类型的类来改善这些缺点。在这之后再将其扩展成一个含有特化的 Array 子类型的面向对象层次结构。最后，我们还要比较一下 Array 类型与 C++标准库的 vector 类，并第一次了解泛型算法。沿着这条路，我们将进一步了解 C++对异常处理、模板和名字空间的支持。

2.1 内置数组数据类型

正如第 1 章所介绍的那样，C++为基本算术数据类型（如整数类型）提供了内置的支持。如：

```
// 声明一个整型对象，ival
// 初始化为初始值 1024
int ival = 1024;
```

同时，它也支持双精度和单精度浮点数据类型：

```
// 声明一个双精度浮点对象，dval
// 初始化为初始值 3.14159
double dval = 3.14159;
// 声明一个单精度浮点对象，fval
// 初始化为初始值 3.14159
float fval = 3.14159;
```

此外，C++还支持布尔类型以及用来存放字符集中单个元素的字符类型。

C++为算术数据类型提供了赋值、一般算术运算以及关系运算的内置支持。算术运算如加、减、乘、除，关系运算如等于、不等于、小于和大于。例如：

```
int ival2 = ival + 4096; // addition (加)
int ival3 = ival2 - ival; // subtraction (减)
```

```
dval = fval * ival;    // multiplication (乘)
ival = ival3 / 2; // division (除)
bool result = ival2 == ival3;   // equality (等于)
result = ival2 + ival != ival3; // inequality (不等于)
result = fval + ival2 < dval;   // less-than (小于)
result = ival > ival2; // greater-than (大于)
```

另外，标准库还支持基本类抽象的组合，例如字符串、复数（在 2.7 节之前我们暂时不考虑标准库提供的 vector 类）

在内置数据类型与标准库类的类型之间是复合类型（compound type），特别是指针和数组类型（我们将在 2.2 节中介绍指针类型）

数组（array）是一种顺序容器，它包含单一类型的元素。例如，序列：

```
0 1 1 2 3 5 8 13 21
```

代表菲波那契数列的前 9 个数。（只要给出最前面两个元素，后面的元素依次可以由前面两个元素相加得出。）

为了定义和初始化一个数组以便存放这些数，我们可以这样写：

```
int fibon[ 9 ] = { 0, 1, 1, 2, 3, 5, 8, 13, 21 };
```

数组对象的名字是 fibon，这是一个包含 9 个元素的整型一维（dimension）数组。第一个元素为 0，最后一个为 21，通过数组下标（subscript）操作符，我们可以以索引方式访问数组的元素。例如，为了读取数组的第一个元素，我们可能会这样写；

```
int first_elem = fibon[ 1 ]; // 不正确
```

不幸的是，这是不正确的，虽然它本身并没有语言错误。

在 C++中，数组下标从 0 开始，而不是 1。在位置 1 上的元素实际上是数组的第一个元素。类似地，位置 0 上的元素才是第一个元素，为了访问数组的最后一个元素，我们总是要索引到数组长度-1 的位置处的元素。

```
fibon[ 0 ]; // 第一个元素
fibon[ 1 ]; // 第二个元素
fibon[ 8 ]; // 最后一个元素
fibon[ 9 ]; // 喔!
```

用 9 索引的元素不是数组的元素，fibon 的 9 个元素由下标 0－8 索引。初学者常见的错误就是用位置 1－9 来索引。事实上，这非常普遍，以至于这个错误有了自己专用的名字：一位偏移（off-by-one）错误。

通常，我们用循环来遍历数组中的元素。例如，下面的程序初始化了一个包含 10 个元素的数组，其值分别从 0 到 9。然后再在标准输出上以降序输出：

```
int main()
{
    int ia[ 10 ];
    int index;

    for ( index = 0; index < 10; ++index )
    // ia[0] = 0, ia[1] = 1, 等等
```

```
        ia[ index ] = index;
        for ( index = 9; index >= 0; --index )
        cout << ia[ index ] << " ";
        cout << endl;
}
```

两个循环各迭代 10 次，关键字 for 后面的三条语句控制循环，第一条语句向 index 赋值 0：

```
index = 0;
```

它只在循环开始真正工作之前被执行一次。第二条语句：

```
index < 10;
```

表示循环的结束条件（stopping condition），它开始真正的循环序列。如果它的值为真，则与 for 循环相关联的语句（或一组语句）将执行。如果它的值为假，则循环终止。在本例中，每次当 index 值小于 10 时，会执行如下语句：

```
ia[ index ] = index;
```

第三条语句：

```
++index
```

是对一个算术对象加一的简短写法，它等价于：

```
index = index + 1;
```

它在与 for 循环相关联的语句执行之后才执行，这里与 for 循环相关联的语句是“把 index 的值赋给以 index 为下标的元素”。这第三条语句的执行完成了 for 循环的一次迭代。序列的每次重复都重新测试条件，条件为假时，循环终止。（第 5 章将详细介绍 for 循环。）第二个循环则以相反的顺序输出这些值。

虽然 C++对数组类型提供了内置支持，但是这种支持仅限于“用来读写单个元素”的机制。C++不支持数组的抽象（abstraction），也不支持对整个数组的操作，我们有时会希望对整个数组进行操作，例如，把一个数组赋值给另外一个数组。对两个数组进行相等比较，或者想知道数组的大小（size），例如，给出两个数组，我们不能用赋值操作符把一个数组拷贝到另一个中去：

```
int array0[ 10 ], array1[ 10 ];

// 错误：不能直接把一个数组赋值给另一个数组
array0 = array1;
```

如果我们希望把一个数组赋值给另外一个，则必须自己写程序，按顺序拷贝每个元素：

```
for ( int index = 0; index < 10; ++index )
        array0[ index ] = array1[ index ];
```

而且，数组类型本身没有自我意识，它不知道自己的长度，我们必须另外记录数组本身的这些信息。当我们希望把数组作为一个参数传递给一个函数的时候，问题就出现了。在 C++中，数组不同于整数类型和浮点数类型，它不是 C++语言的一等（first-class）公民。数组是

从 C 语言中继承来的，它反映了数据与对其进行操作的算法的分离，而这正是过程化程序设计的特征。在本章后面部分，我们将会了解一些不同的策略，通过这些策略可以使数组具有一些额外的公民特权。

练习 2.1

为什么内置数组类型不支持数组之间的赋值，支持这种操作需要什么信息？

练习 2.2

你认为作为一等公民的数组应该支持什么操作？

2.2 动态内存分配和指针

在开始基于对象的设计之前，我们需要暂时偏离一下主题，先来介绍一下 C++程序内存分配的问题。原因是，我们必须首先介绍在程序执行期间怎样申请和访问内存。否则，没有办法真正实现我们的设计（并且展示理想的 C++代码），这是本小节的目的。

在 C++中，对象可以静态分配——即编译器在处理程序源代码时分配，也可以动态分配——即程序执行时调用运行时刻库函数来分配。这两种内存分配方法的主要区别是效率与灵活性之间的平衡准则不同。出于静态内存分配是在程序执行之前进行的，因而效率比较高。但是，它缺少灵活性，它要求在程序执行之前就知道所需内存的类型和数量。例如，利用静态分配的字符串数组，我们无法很容易地处理和存储任意的文本文件。一般来说，存储未知数目的元素需要动态内存分配的灵活性。

到目前为止，所有的内存分配都是静态的。例如，以下定义：

```
int ival = 1024;
```

指示编译器分配足够的存储区以存放一个整型值，该存储区与名字 ival 相关联。然后，用数值 1024 初始化该存储区。这些工作都在程序执行之前完成。

有两个值与对象 ival 相关联：一个是它包含的值——本例中为 1024，另一个是存放这个值的存储区的地址。在 C++中，这两个值都可以被访问。当我们写出下面的代码时：

```
int ival2 = ival + 1
```

我们访问 ival 所包含的值，并把它加 1，然后再用该新值初始化 ival2。在本例中，ival2 初始值为 1025。怎样访问和存储内存地址呢？

C++支持用指针类型来存放对象的内存地址值。例如，为了声明一个能存放 ival 内存地址的指针类型。我们可以这样写：

```
// 一个指向 int 类型的指针
int *pint;
```

C++预定义了一个专门的取地址（address-of）操作符（&）。当我们把它应用在一个对象上时，返回的是对象的地址值。因此，为了将 ival 内存地址值赋给 pint，我们可以这样写：

```
int *ping;
pint = &ival; // 把 ival 的地址 pint
```

为了访问 pint 所指向的实际对象，我们必须先用解引用（dereference）操作符（*）来解除 pint 的引用（dereference pint）。例如，下面我们通过 pint 间接地给 ival 加 1：

```
// 通过 pint 间接地给 ival 加 1
*pint = *pint + 1;
```

它等价于下面直接对 ival 操作的语句：

```
// 直接给 ival 加 1
ival = ival + 1;
```

在本例中，使用指针间接地操作 ival 没有什么实际的好处，这样做比直接操作 ival 的效率要低，而且又容易出错。我们只是用它来简单地介绍一下指针。在 C++中，指针的主要用处是管理和操纵动态分配的内存。

静态与动态内存分配的两个主要区别是：

1.静态对象是有名字的变量，我们直接对其进行操作。而动态对象是没有名字的变量，我们通过指针间接地对它进行操作。稍后我们会看到一个例子。

2.静态对象的分配与释放由编译器自动处理。程序员需要理解这一点，但不需要做任何事情。相反，动态对象的分配与释放，必须由程序员显式地管理，相对来说比较容易出错，它通过 new 和 delete 两个表达式来完成。

对象的动态分配可通过 new 表达式的两个版本之一来完成。第一个版本用于分配特定类型的单个对象。例如：

```
int *pint = new int( 1024 );
```

分配了一个没有名字的 int 类型的对象，对象初始值为 1024。然后，表达式返回对象在内存中的地址。接着，这个地址被用来初始化指针对象 pint。对于动态分配的内存，惟一的访问方式是通过指针间接地访问。

new 表达式的第二个版本，用于分配特定类型和维数的数组。例如：

```
int *pia = new int[ 4 ];
```

分配了一个含有四个整数元素的数组。不幸的是，我们没有办法给动态分配的数组的每个元素显式地指定一个初始值。

分配动态数组时一个常令人迷惑的问题是，返回值只是一个指针，与分配单一动态对象的返回类型相同。例如，pint 与 pia 的不同之处在于，pia 拥有四元素数组的第一个元素的地址，而 pint 只是简单地包含单一对象的地址。当用完了动态分配的对象或对象的数组时，我们必须显式地释放这些内存。我们可以通过使用 delete 表达式的两个版本之一来完成这件事情，而释放之后的内存则可以被程序重新使用。单一对象的 delete 表达式形式如下：

```
// 删除单个对象
delete pint;
```

数组形式的 delete 表达式如下：

```
// 删除一个对象数组
delete [] pia;
```

如果忘了删除动态分配的内存，又会怎么样呢？如果真的如此，程序就会在结束时出现内存泄漏（memory leak）的问题。内存泄漏是指一块动态分配的内存，我们不再拥有指向这

块内存的指针，因此我们没有办法将它返还给程序供以后重新使用。（现在大多数系统提供识别内存泄漏的工具，可以向系统管理员咨询。）

对指针类型和动态内存分配讲得这么快，可能会留下很多应该回答的问题。但是，动态内存分配和指针操作是 C++实际编程基础的一个方面，我们不想推迟到后面再介绍它。在本章接下去的介绍中，我们将在基于对象的与面向对象的 Array 类的实现中，看到它的用法。8.4 节将详细介绍动态内存分配以及 new 与 delete 表达式的用法。

练习 2.3

说出下面定义的四个对象之间的区别：

```
(a) int ival = 1024;  (c) int *pi2 = new int( 1024 );
(b) int *pi = &ival;  (d) int *pi3 = new int[ 1024 ];
```

练习 2.4

下面的代码段是做什么的，有什么严重错误？（注意，指针 pia 的下标操作符的用法是正确的，在 3.9.2 节中我们会解释其理由。）

```
int *pi = new int( 10 );
int *pia = new int[ 10 ];
while ( *pi < 10 ) {
     pia[ *pi ] = *pi;
     *pi = *pi + 1;
}
delete pi;
delete [] pia;
```

2.3 基于对象的设计

在本节中，我们将使用 C++的类机制来设计和实现一个数组抽象。我们最初的实现只支持一个整型数组。以后，我们将用模板机制对这种抽象进行扩展，使其能够支持无限数目的数据类型。

第一步，我们需要决定数组应该提供哪些操作。尽管我们希望能提供所有的操作，但是我们却不能一次提供所有的功能。下面是第一步所支持操作的集合：

1.数组类的实现中有内置的自我意识。首先，它知道自己的大小。

2.数组类支持数组之间的赋值、以及两个数组之间的相等和不相等的比较操作。

3.数组类应该支持对其所含的值进行下列查询操作：数组中最小值是什么？最大值是什么？某个特殊的值是否在数组中？如果存在，它占的第一个位置的索引是什么？

4.数组类支持自排序。为了便于讨论，假定存在一群用户，他们认为数组支持排序的功能很重要。而另外一些人对此却不以为然。

除了支持数组操作，还必须支持数组本身的机制，包括：

5.能够指定长度，以此来创建数组（这个值无需在编译时刻知道）。

6.能够用一组值初始化数组。

7.能够通过一个索引来访问数组中的单个元素。为便于讨论，假设用户强烈要求用数组下标操作符来实现这项功能。

8.能够截获并指出错误的索引值。假设我们认为这很有必要，所以没有询问用户的想法。我们认为这是一个设计良好的数组所必须实现的。

我们与潜在用户的讨论已经引起了极大的热情，现在我们要真正实现它。但是怎样把这个设计转换成 C++代码呢？支持基于对象设计的类的一般形式如下：

```
class classname {
public:
// 公有操作集合
private:
// 私有实现代码
};
```

这里，class、public 和 private 是 C++语言的关键字，classname 是用户定义的标识符，它用来命名这个类，以便在后面引用该类。我们将前面设计的类命名为 IntArray，等到我们使它支持的数据类型更广泛时，再将它改名为 Array。

类名代表的是一个新的数据类型，我们可以用它来定义这种类类型的对象，就像用内置的类型定义对象一样。例如：

```
// 单个 IntArray 类对象
IntArray myArray;

// 指向 IntArray 类对象的指针
IntArray *pArray = new IntArray;
```

类定义包括两个部分：类头（class head），由关键字 class 与相关联的类名构成。类体（class body），由花括号括起来，以分号结束。类头本身也用作类的声明。例如：

```
// 在程序中声明 IntArray 类，但是不提供定义
class IntArray;
```

类体包含成员定义，以及访问标签，如 public 和 private。类的成员包括“该类能执行的操作”和“代表类抽象所必需的数据”。这些操作称为成员函数（member function）或方法（method）。对于 IntArray 类来说，它由以下的内容构成：

```
class IntArray {
public:
    // 相等与不相等操作:    #2b
    bool operator==( const IntArray& ) const;
    bool operator!=( const IntArray& ) const;

    // 赋值操作符: #2a
    IntArray& operator=( const IntArray& );

    int size() const; // #1
    void sort(); // #4

    int min() const;  // #3a
    int max() const;  // #3b
```

```
        // 如值在数组中找到
        // 返回第一次出现的索引
        // 否则返回-1
        int find( int value ) const;  // #3c
    private:
        // 私有实现代码
    };
```

成员函数右边的数字对应着我们前面定义的规范表中的条目，我们现在不打算解释参数表中的 const 修饰符或参数表后面的 const 修饰符，现在还没必要详细解释。但是在实际的程序中，它们还是必需的。

通过使用两个成员访问操作符（member access operator）中的一个，我们可以调用一个有名字的成员函数，如 min()。这两个操作符为：用于类对象的点操作符（.），以及用于类对象指针的箭头操作打（->）。例如，为了在数组 myArray 类对象中找到最小值，我们可以这样写：

```
// 用 myArray 数组中的最小元素来初始化 min_val
int min_val = myArray.min();
```

为了在动态分配的 IntArray 对象中查找最大值，我们可以这样写：

```
int max_val = pArray->max();
```

［是的，我们还没介绍怎样用一个长度与一组值来初始化 IntArray 类对象。有个特殊的称为构造函数（constructor）的成员函数可以完成这些工作，我们将会简要地介绍它。］

把操作符应用在这些类对象上的方式与应用在内置数据类型上的对象一样。下面，给出两个 IntArray 对象：

```
IntArray myArray0, myArray1;
```

赋值操作符可以这样应用：

```
// 调用拷贝赋值成员函数:
// myArraya.operator=( myArray1 )
myArray0 = myArray1;
```

等于操作符的调用如下所示：

```
// 调用等于成员函数:
// myArray0.operator==( myArray1 )
if ( myArray0 == myArray1 )
    cout << "!!our assignment operator works!\n";
```

关键字 private 和 public 控制对类成员的访问。出现在类体中公有（public）部分的成员，在一般程序的任何地方都可以访问它们。出现在私有（private）部分的成员只能在该类的成员函数或友元（friend）中被访问。（我们要到 15.2 节才会解释友元。）

一般来说，公有成员提供了该类的公有接口（public interface）——即实现了这个类的行为的操作集合。它包括该类的所有成员函数，或者只包括其中一个子集。私有成员提供私有实现代码（private implementation）——即存储信息的数据。

这种“类的公共接口与私有实现代码的分离”，被称为信息隐藏（information hiding）。信息隐藏是软件工程中一个非常重要的概念，在后面的章节中将为详细的介绍。简要说来，

它为程序提供了两个主要好处：

1．如果类的私有实现代码需要修改或扩展，那么，只有相对很小一部分“要求访问这些实现代码的成员函数”需要修改。而许多使用该类的用户程序无需修改，但是要求重新编译。（6.18 节将演示这个过程。）

2．如果类的私有实现代码有错误，那么通常需要检查的代码数量只局限在相对较少的需要访问这些实现代码的成员函数上，而无需检查整个程序。

哪些数据成员是 IntArray 必需的呢？当声明一个 IntArray 对象时，用户会指定数组大小，我们需要存储它。因此，我们将定义一个数据成员来做到这一点。另外，我们需要实际分配并存储底层的数组，这将通过 new 表达式来变现，我们将定义一个指针数据成员来存储 new 表达式返回的地址值：

```
class IntArray {
public:
    // ...
    int size() const { return _size; }
private:
    // 内部数据
    int _size;
    int *ia;
};
```

由于我们把_size 放在类的私有区内，因此我们有必要声明一个公有成员函数，以便允许用户访问它的值。由于 C++不允许成员函数与数据成员共享同一个名字，所以在这样的情况下，一般的习惯是在数据成员名字前面加一个下划线（_）。因此，我们有了公有访问函数 size()和私有数据成员_size。在本书以前的版本中，我们在访问函数前加上 get 或 set。实践证明这样做有些累赘。

尽管这种公有访问函数的用法允许用户读取相应的值，但是这种实现似乎还有些根本的错误，至少第一眼看上去是这样的。你看出来了吗？考虑下面的语句：

```
IntArray array;
int array_size = array.size();
```

还有：

```
// 假设_size 是 public 的
int array_size = array._size;
```

将 array_size 初始化为数组的维数。但是很显然，第一个例子需要一个函数调用，而第二个只需直接访问内存就行了。一般来说，函数调用比直接访问内存的开销要大得多。因而信息隐藏是否给程序的执行效率增加了严重的额外负担，或许是阻碍性的负担呢？幸运的是，在一般情况下，回答是：不。

C++提供的解决方案是内联函数（inline function）机制。内联函数在它的调用点上被展开。一般来说。内联函数不会引入任何函数调用。[4]例如，在 for 循环的条件子句中的 size()调用：

[4] 然而，实际并不总是这样的，对于编译器来说，内联函数是一种请求，而不是一种保证。参见 7.6 节的讨论。

```
for ( int index = 0; index < array.size(); ++index )
    // ...
```

并没有真的被调用_size 次，而是在编译期间被内联扩展为下面的一般形式：

```
// array.size()的内联扩展
for ( int index = 0; index < array._size; ++index)
    // ...
```

在类定义中被定义的成员函数［如 size()］会被自动当作是内联函数。此外，我们也可以用 inline 关键字显式地要求一个函数被视为内联函数。（7.6 节中有更多关于内联函数的说明。）

到目前为止，我们已经提供了 IntArray 类所要求的操作（前面的 1－4 项），但是还没有提供初始化机制和数组中单个元素的访问方式（5－8 项）

程序设计中的一个常见错误是使用事先并没向被正确初始化的对象。实际上，这是一个极为常见的错误，所以 C++为用户定义的类提供了一种自动初始化机制：类构造函数（class constructor）

构造函数是一种特殊的类成员函数，专门用于初始化对象。如果构造函数被定义了，那么在类的每个对象第一次被使用之前，这构造函数就被自动应用在对象上。构造函数由谁来定义呢？类的提供者——也就是我们来定义构造函数。为一个类确定必要的构造函数是程序设计不可缺少的一部分。

为了定义一个构造函数，我们只要给它与类相同的名字即可。另外，我们不能给构造函数指定返回值。但是可以给类定义多个构造函数，尽管它们都具有相同的名字，但只要编译器能够根据参数表区分它们就行。

更一般地，C++支持被称为函数重载（function overloading）的机制。函数重载允许两个或更多个函数使用同一个名字，限制条件是它们的参数表必须不同：参数类型不同，或参数的数目不同。根据不同的参数表，编译器就能够判断出，对某个特定的调用应该选择哪一个版本的重载函数。下面是一组合法的 min()重载函数（这些函数也可以是类成员函数）：

```
// 一组 min()重载函数
// 每个函数都有一个特有的参数表
#include <string>;
int min( const int *pia, int size );
int min( int, int );
int min( const char *str );
char min( string );
string min( string, string );
```

重载函数在运行时刻的行为与非重载函数完全一样，主要的负担是在编译时刻用来决定中该调用哪个实例所需要的时间。如果 C++不提供函数重载支持，那么我们就必须为程序中每个函数都要提供一个独一无二的名字。（第 9 章将详细讨论函数重载的内容。）

我们为 IntArray 类指定了下面三个构造函数：

```
class IntArray {
public:
    explicit IntArray( int sz = DefaultArraySize );
    IntArray( int *array, int array_size );
```

```
        IntArray( const IntArray &rhs );
        // ...
    private:
        static const int DefaultArraySize = 12;
        // ...
    };
```

构造函数：

```
    IntArray( int sz = DefaultArraySize );
```

被称为缺省构造函数（default constructor），用为它不需要用户提供任何参数。（我们现在不打算解释这个缺省构造函数声明中出现的关键字 explicit，之所以显示它仅仅是为了完整性。）如果程序员提供参数，则该值将被传递给构造函数。例如：

```
    IntArray array1( 1024 );
```

将参数 1024 传递给构造函数。另一方面，如果用户不指定长度，那么构造函数将使用 DefaultArraySize 的值。例如：

```
    IntArray array2;
```

将导致用 DefaultArraySize 的值来调用构造函数。（被声明为 static 的数据成员是一类特殊的共享数据成员，无论这个类的对象被定义了多少个，静态数据成员在程序中也只有一份。这是在类的所有对象之间共享数据的一种方式。13.5 节有全面的讨论。）

下面是缺省构造函数的一个简化实现版本，简化到没有考虑出错的可能性。（可能出现什么错误呢？本例中有两个：首先，提供给程序的动态内存不是无限的，因此，new 表达式有可能失败，2.6 节中将介绍如何处理这种情况。第二，传递给参数 sz 的值可能是无效的，例如，负数或 0，或者一个很大的值，以至于无法存储在 int 类型的变量中。）

```
    IntArray::
    IntArray( int sz )
    {

        // 设置数据成员
        size = sz;
        ia = new int[ _size ];

        // 初始化内存
        for ( int ix=0; ix < _size; ++ix )
            ia[ ix ] = 0;
    }
```

这是我们第一次在类体的外面定义类的成员函数。惟一的语法区别是要指出成员函数属于哪个类，这可以通过类域操作符（class scope operator）来实现：

```
    IntArray::
```

双冒号（::）操作符被称为域操作符（scope operator）。当与一个类名相连的时候，像上面例子中那样，它就成为一个类域操作符。我们可以非正式地把域看作是一个可视窗口。全局域的对象在它被定义的整个文件里（一直到文件末尾）都是可见的。在一个函数内被定义的对象是局域的（local scope），它只在定义其的函数体内可见。每个类维持一个域，在这个域之外，它的成员是不可见的。类域操作符告诉编译器，后面的标识符可在该类的范围内

被找到。本例中：

```
IntArray::
IntArray( int sz )
```

告诉编译器 IntArray()函数被定义为 IntArray 类的成员。（尽管程序域不是我们现在应该关注的事情，但是最终我们还是要理解域的概念。在第 8 章中我们将详细讲解程序域，而类域将在 13.9 节中特别讨论。）

IntArray 类的第二个构造函数用内置的整数数组初始化一个新的 IntArray 类对象。它需要两个参数：一个是实际的数组，另一个参数指明数组的大小。例如：

```
int ia[10] = {0,1,2,3,4,5,6,7,8,9};
IntArray iA3(ia,10);
```

这个构造函数的实现几乎与第一个构造函数相同。（这里我们又一次没有保护自己的代码。）

```
IntArray::
IntArray( int *array, int sz )
{
    // 设置数据成员
    size = sz;
    ia = new int[ _size ];
    // 拷贝数据成员
    for ( int ix=0; ix < _size; ++ix )
        iz[ix ] = array[ ix ];
}
```

最后一个 IntArray 构造函数用另外一个 IntArray 对象来初始化当前的 IntArray 对象，对于下面两种形式，无论是哪一种，它都将被自动调用：

```
IntArray array;

// 等价的初始化方式
IntArray ia1 = array;
IntArray ia2( array );
```

这种构造函数被称为类的拷贝构造函数（copy constructor）。在后面的章节中我们将会看到更多的示例。下面的实现再次忽略了可能出现的运行时刻程序异常：

```
IntArray::
IntArray( const IntArray &rhs )
{
    // 拷贝构造函数
    _size = rhs._size;
    ia = new int[ _size ];
    for (int ix = 0; ix < _size; ix++ )
        iz[ ix ] = rhs.ia[ ix ];
}
```

本例引入了一种新的复合类型：引用（reference），即 IntArray &rhs。引用是一种没有指针语法的指针。（因此，我们写成 rhs._size，而不是 rhs->_size。）与指针一样，引用提供对对象的间接访问。（3.6 节中我们将对指针和引用作更多的介绍。）

注意，这三个构造函数都是以相似的方式来实现的。一般来说，当两个或多个函数重复相同的代码时，就会将这部分代码抽取出来，形成独立的函数，以便共享。以后，如果需要改变这些实现，则只需改变一次。而且，这种实现的共享本质更容易为大家所理解。

怎么样把构造函数中的代码抽取出来形成独立的共享函数呢？下面是一种可能的实现：

```
class IntArray {
public:
    // ...
private:
    void init( int sz, int *array );
    // ...
};
void
IntArray::
init( int sz, int *array )
{
    _size = sz;
    ia = new int[ _size ];
    for ( int ix=0; ix < _size; ++ix )
        if ( ! array )
            ia[ ix ] = 0;
        else ia[ ix ] = array[ ix ];
}
```

三个构造函数可重写为；

```
IntArray::IntArray( int sz ){ init( sz, 0 ); }
IntArray::IntArray( int *array, int sz )
{ init( sz, array ); }
IntArray::IntArray( const IntArray &rhs )
{ init( rhs.size, rhs.ia ); }
```

类机制还支持特殊的析构成员函数（destructor member function）。每个类对象在被程序最后一次使用之后，它的析构函数就会被自动调用。我们通过在类的名字前面加一个波浪线（~）来标识析构函数。一般地，析构函数会释放在类对象使用和构造过程中所获得的资源。例如，在 IntArray 的析构函数中，它会删除构造时分配的内存。（我们将在第 14 章详细讨论构造函数和析构函数。）下面是我们的实现：

```
class IntArray {
public:
    // 构造函数
    explicit IntArray( int size = DefaultArraySize );
    IntArray( int *array, int array_size );
    IntArray( const IntArray &rhs );

    // 析构函数
    ~IntArray() { delete [] ia; }
    // ...
private:
```

```
        // ...
};
```

除非用户能够很容易地通过索引访问单个元素，否则数组类就没有更实际的用处。例如，我们的类需要支持下面的一般用法：

```
IntArray array;
int last_pos = array.size()-1;
int temp = array[ 0 ];
array[ 0 ] = array[ last_pos ];
array[ last_pos ] = temp;
```

我们通过提供“专用于一个类的下标操作符实例”，来支持索引 IntArray 类对象。下面是支持这种用法的一个实现：

```
#include <cassert>
int&
IntArray::
operator[]( int index )
{
    assert( index >= 0 && index < size );
    return ia[ index ];
}
```

一般而言，C++语言支持操作符重载（operator overloading），这样就可以为特定的类（类型）定义新的操作符实例。典型地，类提供一个或多个赋值操作符、等于操作符，可能还有一个或多个关系操作符，以及 iostream 输入和输出操作符。（3.15 节有关于操作符重载的更进一步的说明。在第 15 章我们将详细讨论操作符重载。）

类定义以及相关的常数值或 typedef 名通常都存储在头文件中，并且头文件以类名来命名。因此，假如我们创建一对头文件：IntArray.h 和 Matrix.h，则所有打算使用 IntArray 类或 Matrix 类的程序就都必须包含相关的头文件。

类似地，不在类定义内部定义的类成员函数都存储在与类名同名的程序文本文件中。例如，我们将创建一对程序文本文件：IntArray.C 和 Matrix.C，用来存储相关类的成员函数。（记住，程序文本文件的后缀因编译系统而不同，你应该检查自己的系统所使用的命名习惯。）这些函数不用随每个使用相关类的程序而重新编译，这些成员函数经过预编译之后被保存在类库中，iostream 库就是这样一个例子。

练习 2.5

C++类的关键特征是接口与实现的分离。接口是一些“用户可以应用到类对象上的操作”的集合。它由三部分构成：这些操作的名字、它们的返回值，以及它们的参数表。一般地，这些就是该类用户所需要知道的全部内容。私有实现包括为支持公有接口所必需的算法和数据。理想情况下，即使类的接口增长了，它也不用变得与以前的版本不相兼容。另一方面，在类的生命周期内其实现可以自由演化。从下面选择一个抽象（指类），并为该类编写一个公共接口。

```
(a) Matrix    (c) Person  (e) Pointer
(b) Boolean   (d) Date    (f) Point
```

练习 2.6

构造函数和析构函数是程序员提供的函数，它们既不构造也不销毁类的对象（编译器自动把它们作用到这些对象上）。因此构造函数（constructor）和析构函数（destructor）这两个词多少有些误导，当我们写：

```
int main() {
    IntArray myArray( 1024 );
    // ...
    return 0;
}
```

在构造函数被应用之前，用于维护 myArray 中数据成员的内存已经被分配了。实际上，编译器在内部把程序转换成如下的代码（注意这不是合法的 C++代码[5]）：

```
int main() {
    IntArray myArray;
    // 伪 C++代码--应用构造函数
    myArray.IntArray::IntArray( 1024 );
    // ...
    // 伪 C++代码--应用析构函数
    myArray.IntArray::~IntArray();
    return 0;
}
```

类的构造函数主要用来初始化类对象的数据成员。析构函数主要负责释放类对象在生命期内申请到的所有资源。请定义在练习 2.5 中选择的类所需要的构造函数集，你的类需要析构函数吗？

练习 2.7

在练习 2.5 和练习 2.6 中，你差不多已经定义了使用该类的完整公有接口。（我们还需要定义一个拷贝赋值操作符，但是现在我们忽略这个事实——C++为“从一个类对象向另一个类对象赋值”提供了缺省支持。问题在于，缺省的行为常常是不够的。这将在 14.6 节中讨论。）写一个程序来实践在前面两个练习中定义的公有接口。用起来容易还是麻烦？你希望重写这些定义吗？你能在重写的同时保持兼容性吗？

2.4 面向对象的设计

max()与 min()函数的实现没有对数组元素的存储做特殊的假设，因此，我们需要检查数组的每个元素。如果我们要求所有的元素已经排序，则这两个操作就变得非常简单，只要索引第一个元素和最后一个元素即可。而且，如果已知元素已经排序，那么查找一个元素的存在就会更加高效。但是，对数组进行排序增加了 Array 类实现的复杂性。我们的设计出错了吗？

[5] 对于感兴趣的读者，可以参见本书的姐妹篇《Inside the C++ Object Model》，该书讨论了这方面的内容。

实际上，我们现在是否犯了错误与我们做出的选择息息相关。排序的数组是一种特殊的实现：需要时，它完全必要；否则，支持排序数组的额外开销就是一项负担。我们的实现是比较通用的，在大多数情况下是足够的。它支持更广阔范围内的用户。不幸的是，如果用户绝对需要排序数组的行为，那么我们的实现就不能提供支持。对用户来说，他没有办法对 min()、max()以及 find()这些函数比较通用的实现进行改写。实际上，我们选择的通用实现，并不适合特殊的环境。

另一方面，我们的实现对另外一类用户而言又针对性太强了：对索引的范围检查为每次访问元素增加了额外的负担。在设计中，我们没有考虑这样的开销（2.3 节中第 8 条），而是假设：如果结果不正确，那么速度再快也没有价值。但是，这种设计至少对于某一类主要用户：实时虚拟和虚拟现实提供商，就不成立。在这种情况下，数组代表复杂 3D 几何图形的顶点。场景飞快地变化，以至于一些偶然的错误不会被看到。但如果访问速度太慢了，那么实时效果就会被打破。我们实现的数组类虽然比没有范围检查的数组类会更安全，但是在这样的实时应用领域却不够实际。

我们怎样才能支持这三种用户的需要呢？解决的方案已经多多少少体现在代码中了。例如，范围检查局限在下标操作符中。去掉 check.range()的调用，重新命名该数组。现在我们就有了两种实现：一个有范围检查，一个没有范围检查。进一步拷贝一份代码，并把它修改成针对已排序的数组。现在我们就有了对已排序数组的支持：

```
// 未排序，也没有边界检查
class IntArray{ ... };
// 未排序，但支持边界检查
class IntArrayRC{ ... };
// 已排序，但没有边界检查
class IntSortedArray{ ... };
```

这种方案的缺点是什么呢？

1.我们必须维护三个包含大量重复代码的数组实现。我们更希望把这些公共代码只保留一份，然后由“这三个数组类（以及其他一些我们以后会选择支持的数组类）”共享。（比如，可能会是一个带有边界检查的排序数组。）

2.由于三个数组实现是完全独立的类型，所以我们必须编写独立的函数来操作它们，尽管函数内的一般性操作都是相同的。例如：

```
void process_array( IntArray& );
void process_array( IntArrayRC& );
void process_array( IntSortedArray& );
```

我们希望只编写一个函数，它不但能接受现有的数组类，而且，还能够接受任意将来的数组类，只要同样的操作集合也能够应用到这些类上。

面向对象的程序设计方法正是为我们提供了这样一种能力。上面第 1 项可由继承（inheritance）机制提供。当一个 IntArrayRC 类（也就是一个带有范围检查的 IntArray 类）继承了 IntArray 类时，它就可以访问 IntArray 的数据成员和成员函数，而不要求我们维护两份代码拷贝。新的类只需提供实现其额外语义所必需的数据成员和成员函数。

在 C++中，被继承的类，如本例中的 IntArray，被称作基类（base class）。新类被称作从基类派生（derived）而来。我们把它叫做基类的派生类（derived class）或子类型（subtype）。

我们说 IntArrayRC 是一种有特殊行为的 IntArray，它支持对索引值的范围检查。子类型与基类共享公共的接口（common interface）——公有操作的公共集。由于共享公共接口，允许了子类和基类在程序内部可互换使用，而无需考虑对象的实际类型。从某种意义上来说，公共接口封装了单个子类型中与类型相关的细节。类之间的类型/子类型关系形成了继承或派生层次关系（inheritance or derivation hierarchy）。例如，下面的非成员函数 swap()把指向基类 IntArray 对象的引用作为第一个参数，该函数交换索引 i 和 j 处的元素：

```
#include <IntArray.h>
void swap( IntArray &ia, int i, int j )
{
    int tmp = ia[ i ];
    ia[ i ] = ia[ j ];
    ia[ j ] = tmp;
}
```

下面是 swap()函数的三个合法调用：

```
IntArray ia;
IntArrayRC iarc;
IntSortedArray ias;

// ok: ia 是一个 IntArray
swap( ia, 0, 10 );

// ok: iarc 是 IntArray 的子类型
swap( iarc, 0, 10 );

// ok: ias 也是 IntArray 的子类型
swap( ias, 0, 10 );

// error: string 不是 IntArray 的子类型
string str( "not an IntArray!" );
swap( str, 0, 10 );
```

三个数组类都提供了自己的下标操作符实现。当然，我们的要求是，当调用如下函数时：

```
swap(iarc, 0, 10);
```

IntArrayRC 的下标操作符被调用。当调用如下函数时：

```
swap( ias, 0, 10 );
```

IntSortedArray 下标操作符被调用等等。swap()调用的下标操作符必须潜在地随着每次调用而改变，它必须由被交换元素的数组的实际类型来决定。在 C++中，这可以由虚拟函数（virtual function）机制来自动完成。

为使 IntArray 类能够被继承，我们需要在语法上做一点小小的改变：必须（可选择的）减少封装的层次，以便允许派生类访问非公有的实现，而且我们也必须显式地指明哪些函数应该是虚拟的。最重要的变化在于我们如何把一个类设计成为基类。

在基于对象的程序设计中，通常类的提供者只有一个，但是类的用户有许多个。提供者设计并且通常也会实现类。用户使用提供者提供的公有接口，行为的分离可通过将类分成公

有与私有访问级别而反映出来。

在继承机制下有多个类的提供者：一个提供基类实现（可能还有一些派生类），另外一个或多个提供者在继承层次的生命周期内提供派生类，这种行为也是一种实现行为。于类的提供者经常（但并不总是）需要访问基类的实现。为了提供这种能力，同时还要防止对基类实现的一般性访问，C++提供了另外一个访问级别：保护（protected）级别。在类的保护区域内的数据成员和成员函数，不提供给一般的程序，只提供给派生类。（放在基类的私有区域内的成员只能供该类自己使用，派生类不能使用。）下面是修改过的 IntArray 类：

```
class IntArray {
public:
    // 构造函数
    explicit IntArray( int size = DefaultArraySize );
    IntArray( int *array, int array_size );
    IntArray( const IntArray &rhs );

    // 虚拟析构函数!
    virtual ~IntArray() { delete [] ia; }

    // 等于和不等于操作:
    bool operator==( const IntArray& ) const;
    bool operator!=( const IntArray& ) const;
    IntArray& operator=( const IntArray& );
    int size() const { return _size; }

    // 去掉了索引检查功能 . . .
    virtual int& operator[](int index) { return ia[index]; }
    virtual void sort();
    virtual int min() const;
    virtual int max() const;
    virtual int find( int value ) const;
protected:
    // 参见 13.5 节的说明
    static const int DefaultArraySize = 12;
    void init( int sz, int *array );
    int _size;
    int *ia;
};
```

在面向对象与基于对象的设计中，指明一个类的成员是 public 的准则没有变化。重新设计的 IntArray 类将被用作基类，它仍然把构造函数、析构函数、下标操作符、min()和 max()等等声明为公有成员。这些成员继续提供公有接口，但现在接口不只为 IntArray 类服务，同时也为从它派生的整个继承层次服务。

非公有的成员到底该声明为 protected 还是 private 类成员是新的设计准则。如果希望防止派生类直接访问某个成员，我们就把该成员声明为基类的 private 成员。如果确信某个成员提供了派生类需要直接访问的操作或数据存储，而且通过这个成员，派生类的实现会更有效，则我们把该成员声明为 protected。对于 IntArray 类，我们已经将全部数据成员设置成 protected，

也就是实际上允许后续派生的类访问 IntArray 的实现细节。

为了把一个类设计成基类，要做的第二个设计考虑是找出类型相关的成员函数，并把这些成员函数标记为 virtual（虚拟的）。

对于类型相关的成员函数，它的算法由特定的基类或派生类的行为或实现来决定。例如，对每种数组类型，下标操作符的实现是不同的，所以，我们将它声明为 Virtual。

等于、不等于操作符和 size()成员函数的实现对于其应用的数组类型来说是独立的，因此，不把它声明成 Virtual。

对于一个非虚拟函数的调用，编译器在编译时刻选择被调用的函数。而虚拟函数调用的决定则要等到运行时刻。在执行程序内部的每个调用点上，系统根据被调用对象的实际基类或派生类的类型来决定选择哪一个虚拟函数实例。例如，考虑下面的代码：

```
void init( IntArray &ia )
{
    for ( int ix = 0; ix < ia.size(); ++ix )
        ia[ ix ] = ix;
}
```

形式参数 ia 可以引用 IntSortedArray、IntArrayRC 或 IntArray 类的对象（我们将简要介绍这里的派生类）。函数 size()作为非虚拟函数，由编译器处理并内联展开。但是，下标操作符要直到执行循环的每次迭代时才能被处理，因为在编译期间编译器不知道数组 ia 指向的实际类型。

（第 17 章将详细讨论虚拟函数，包括虚拟析构函数的主题，以及使用虚拟函数设计带来的效率问题。[LIPPMAN96a] 对虚拟函数的实现与效率有更深入的讨论。）

一旦我们定好了设计方案，C++的实现就很容易了。例如，下面这个完整的 IntArrayRC 派生类定义，被放在一个独立的头文件 IntArrayRC.h 中，该文件包含头文件 IntArray.h，而 IntArray.h 包含有 IntArray 类的定义：

```
#ifndef IntArrayRC_H
#define IntArrayRC_H

#include "IntArray.h"

class IntArrayRC : public IntArray {
public:
    IntArrayRC( int sz = DefaultArraySize );
    IntArrayRC( int *array, int array_size );
    IntArrayRC( const IntArrayRC &rhs );
    virtual int& operator[]( int );

private:
    void check_range( int );
};
#endif
```

IntArrayRC 只需定义不同于 IntArray 实现的那些方面，或者加上对 IntArray 扩展的实现。

1．它必须提供自己的下标操作符实例，以支持范围检查。

2．它必须提供一个操作来做实际的检查工作（由于它不是公有接口的一部分，所以我们把它声明为 private）。

3．它必须提供一组自动初始化函数，即自己的构造函数集。

IntArray 的成员函数与数据成员对于 IntArrayRC 来说都是可用的，就如同 IntArrayRC 已经显式地定义了它们一样。这正是下面这句话的含义：

```
class IntArrayRC : public IntArray
```

冒号定义了 IntArrayRC 是从 IntArray 派生而来的。[关键字 public 表明派生类共享基类的公有接口。IntArrayRC 类型的对象可以用在任何“可以使用基类类型对象”的位置上，比如在 swap()例子中。第 18 章会详细解释这一点。] IntArrayRC 可以看作是 IntArray 的扩展，它增加了下标范围检查的额外特性。下面是下标操作符的一个实现：

```
inline int&
IntArrayRC::operator[]( int index )
{
    check_range( index );
    return ia[ index ];
}
```

这里，check_range()被实现为一个内联成员函数。它调用 assert()宏 [关于 assert()宏的讨论见 1.3 节。]：

```
#include <cassert>
inline void
IntArrayRC::check_range( int index )
{
    assert( index >= 0 && index < size );
}
```

[我们把 check_range()函数作为一个独立的函数，以便说明私有成员函数并且将范围检查的处理封装起来，方便我们以后改变边界错误的处理方式，或是用异常处理代替 assert()。]

派生类对象实际上由几部分构成：每个基类是一个类的子对象（subobject），它在新定义的派生类中有独立的一部分。派生类对象的初始化过程是这样的，首先自动调用每个基类的构造函数来初始化相关的基类子对象，然后再执行派生类的构造函数。从设计的角度来看，派生类的构造函数应该只初始化那些在派生类中被定义的数据成员，而不是某类中的数据成员。

虽然我们引入了与类相关的下标操作符版本，以及一个私有的 check_range()辅助函数，但是我们并没有引入需要初始化的额外数据成员。因此，我们可以合理地假设.继承基类的构造函数已经足够了，我们不需要再提供 IntArrayRC 的构造函数——因为不需要它们做任何事情。

但是，实际上，我们还是需要提供 IntArrayRC 的构造函数，因为基类的构造函数并没有被派生类继承（析构函数和拷贝赋值操作符同样也没有），还因为我们需要某个接口，以便通过这个接口把必要的参数传递给某类 IntArray 的构造函数。

例如，假设我们定义了一个 IntArrayRC 对象：

```
int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13 };
```

```
IntArrayRC iarc( ia, 8 );
```

怎样才能把 ia 和 8 传递给基类的构造函数呢？（不可否认，如果 IntArray 构造函数被继承了，那么就没有这个问题。实际上，那样的话我们会有其他更严重的问题，但现在没有足够的篇幅向你证明这一点。）无论如何，派生类构造函数的语法提供了向基类构造函数传递参数的接口。例如，下面是两个必需的 IntArrayRC 构造函数。（第 14 章与第 17 章将对构造函数作更多的讲解，其中包括关于“为什么我们不需要提供 IntArrayRC 拷贝构造函数”的解释）：

```
inline IntArrayRC::IntArrayRC( int sz)
: IntArray( sz ) {}
inline IntArrayRC::IntArrayRC( const int *iar, int sz )
: IntArray( iar, sz ) {}
```

由冒号分割出来的部分称作成员初始化列表（member initialization list），它提供了一种机制，通过这种机制，我们可以向 IntArray 的构造函数传递参数。两个 IntArrayRC 构造函数的函数体都是空的，因为它们的工作就是把参数传递给相关的 IntArray 构造函数。我们无需提供显式的 IntArrayRC 析构函数，因为派生类没有引入任何需要析构的数据成员。继承过来的、需要析构的 IntArray 成员都由 IntArray 的析构函数来处理。

头文件 IntArrayRC 上中含有 IntArrayRC 的定义，以及在类定义之外定义的全部内联成员函数的定义。如果我们定义了非内联函数，则把它们放在 IntArrayRC.C——这个相关联的程序文本文件中。

下面这个小程序实现了 IntArray 与 IntArrayRC 两个类的层次结构。

```
#include <iostream>
#include <IntArray.h>
#include <IntArrayRC.h>

extern void swap(IntArray&,int,int);

int main()
{
    int array[ 4 ] = { 0, 1, 2, 3 };
    IntArray ia1( array, 4 );
    IntArrayRC ia2( array, 4 );

    // 错误：一位偏移（off-by-one），应该是 size-1
    // IntArray 对象捕捉不到这个错误
    cout << "swap() with IntArray ia1\n";
    swap( ia1, 1, ia1.size() );

    // ok: IntArrayRC 对象可以捕捉到这样的错误
    cout << "swap() with IntArrayRC ia2\n";
    swap( ia2, 1, ia2.size() );
    return 0;
}
```

编译并执行这个程序，产生如下结果：

```
swap() with IntArray ia1
swap() with IntArrayRC ia2
```

```
Assertion failed: index >= 0 && index < size
```

C++支持另外两种形式的继承：多继承（multiple inheritance，也译多重继承），也就是一个类可以从两个或多个基类派生而来。以及虚拟继承（virtual inheritance），在这种继承方式下基类的单个实例在多个派生类之间共享。第 18 章将讨论这些内容。面向对象程序设计的另一个较为深入的方面是，在程序执行过程中任意一个点上，我们都能够查询某类的引用或指针所指向的实际类型。这是由 RTTI（运行时刻类型识别）设施提供的，我们将在 19.1 节中讨论它。

练习 2.8

一般来说，类型/子类型继承关系反映了一种“is-a（是一种）”的关系：具有范围检查功能的 ArrayRC 是一种 Array。一本书（Book）是一种图书外借资源（LibraryRentalMaterial）。有声书（AudioBook）是一种书（Book）等等。下面哪些反映出这种“is-a”关系？

(a) 成员函数是一种(isA_kindOf)函数
(b) 成员函数是一种类
(c) 构造函数是一种成员函数
(d) 飞机是一种交通工具
(e) 摩托车是一种卡车
(f) 圆形是一种几何图形
(g) 正方形是一种矩形
(h) 汽车是一种飞机
(i) 借阅者是一种图书馆

练习 2.9

判断以下操作哪些可能是类型相关的，因此可把它们定义为虚拟函数？哪些可以在所有类之间共享？对单个基类或派生类来说哪些是惟一的？

```
(a) rotate(); (b) print();
(c) size();   (d) dateBorrowed();
(e) rewind(); (f) borrower();
(g) is_late();    (h) is_on_loan();
```

练习 2.10

对于保护（protected）访问级别的使用已经有了一些争论。有人认为，使用保护访问级别允许派生类直接访问基类的成员，这破坏了封装的概念，因此，所有的基类实现细节都应该是私有的（private）。另外一些人认为，如果派生类不能直接访问基类的成员，那么派生类的实现将无法有足够的效率供用户使用；如果没有关键字 protected，类的设计者将被迫把基类成员设置为 public。你怎样认为？

练习 2.11

第二个争论是关于将成员函数显式地声明为 virtual 的必要性。一些人认为，这意味着如

果类的设计者没有意识到一个函数需要被声明为 virtual，则派生类的设计者就没有办法改写这个关键函数。因此，他们建议把所有成员函数都设置为 virtual 的。另一方面，虚拟函数比非虚拟函数的效率要低一些。[6]因为它们不能被内联（内联发生在编译时刻，而虚拟函数是在运行时刻被处理的），所以它们可能是运行时刻效率低下的原因之一，尤其是小巧而又被频繁调用的、与类型无关的函数，比如 Array（数组）的 size 函数。你又怎样认为呢？

练习 2.12

下面的每个抽象类型都隐式地包含一族抽象子类型。例如，图书馆藏资料（LibraryRentalMaterial）抽象隐式地包含书（Book）、音像（Puppet）、视盘（Video）等。选择其中一个，找出该抽象的子类型层次，并为这个层次指定一个小的公有接口，且其中包括构造函数。如果存在的话，指出哪些函数是虚拟的，并且写一小段伪代码程序来练习使用这个公有接口。

```
(a) Points     (b) Employees
(c) Shapes     (d) TelephoneNumbers
(e) BankAccounts   (f) CourseOfferings
```

2.5 泛型设计

IntArray 类为预定义的整型数组类型提供了一个有用的替代类型。如果用户希望使用一个 double 或 string 类型的数组，那该怎么办呢？实现一个 double 类型的数组与 IntArray 类的区别只是在其所包含的元素的类型不同，而代码本身无需改变。

C++的模板设施提供了一种机制，它能够将类成函数定义内部的类型和值参数化（parameterizing）（我们要到 10.1 节才会讨论值参数）。这些参数在其他方面不变的代码中用作占位符，以后，这些参数会被绑定到实际类型上，可能是内置的类型，也可能是用户定义的类型。例如，在 Array 类模板中，我们把数组所包含的元素的类型参数化。以后，当我们实例化（instantiate）一个特定类型的实例时，如 int、double 或 string 类型的 Array（数组），就可以在程序中直接使用这三个实例，就好像我们已经显式地为它们编写过代码一样。现在来看一下，怎样把 IntArray 类转换成 Array 类模板。下面是定义：

```
template < class elemType >
class Array {
public:
    // 把元素类型参数化
    explicit Array( int size = DefaultArraySize );
    Array( elemType *array, int array_size );

    Array( const Array &rhs );

    virtual ~Array() { delete [] ia; }
    bool operator==( const Array& ) const;
    bool operator!=( const Array& ) const;
```

[6]参见［LIPPMAN96a］，其中讨论了虚拟函数性能的问题。

```
        Array& operator=( const Array& );
        int size() const { return _size; }
        virtual elemType& operator[](int index){ return ia[index]; }
        virtual void sort();
        virtual elemType min() const;
        virtual elemType max() const;
        virtual int find( const elemType &value ) const;
    protected:
        static const int DefaultArraySize = 12;
        int _size;
        elemType *ia;
    };
```

关键字 template 引入模板，参数由一对尖括号（<，>）括起来——本例中，有一个参数 elemType。关键字 class 表明这个参数代表一个类型。标识符 elemType 代表实际的参数名，它在 Array 类定义中出现了七次，都是作为实际类型的占位符。

在 Array 类的每次实例化中，不论是实例化为 int、double 或 string 等等，实例化的实际类型都将代替 elemType 参数。下面的例子演示了怎样使用 Array 类模板；

```
#include <iostream>
#include "Array.h"

int main()
{
    const int array_size = 4;

    // elemType 变成了 int
    Array<int> ia(array_size);

    // elemType 变成了 double
    Array<double> da(array_size);

    // elemType 变成了 char
    Array<char> ca(array_size);
    int ix;

    for ( ix = 0; ix < array_size; ++ix ) {
        ia[ix] = ix;
        da[ix] = ix * 1.75;
        ca[ix] = ix + 'a';
    }

    for ( ix = 0; ix < array_size; ++ix )
    cout << "[ " << ix << " ] ia: " << ia[ix]
       << "\tca: " << ca[ix]
       << "\tda: " << da[ix] << endl;
    return 0;
}
```

本例中，我们定义了三个独立的 Array 类模板的实例：

```
Array<int> ia(array_size);
Array<double> da(array_size);
Array<char> ca(array_size);
```

这些实例声明就是在类模板名的后面加上一对尖括号，然后在里面写上数组的实际类型。当我们定义类模板对象，如 ia、da 或 ca 时，会发生什么事情呢？编译器必须为相关的对象分配内存。为了做到这一点，形式模板参数被绑定到指定的实际参数类型上。对 ia 来说，Array 类模板通过将 elemType 绑定到类型 int 上，产生如下的类数据成员：

```
// Array<int> ia(array_size);
int _size;
int *ia;
```

结果是一个类，它与我们前面手工编码实现的 IntArray 类等价。对 da 来说，通过将 elemType 绑定到类型 double 上，成员变为：

```
// Array<do uble> da(array_size);
int _size;
double *ia;
```

类似地，对 ca 来说，通过将 elemType 绑定到类型 char 上，成员变为：

```
// Array<char> ca(array_size);
int _size;
char *ia;
```

类模板的成员函数会怎么样呢？不是所有的成员函数都能自动地随类模板的实例化而被实例化，只有真正被程序使用到的成员函数才会被实例化，这一般发生在程序生成过程中的一个独立阶段。（16.8 节将详细讨论这个过程。）

编译并运行程序，会产主如下结果：

```
[ 0 ] ia: 0   ca: a    da: 0
[ 1 ] ia: 1   ca: b    da: 1.75
[ 2 ] ia: 2   ca: c    da: 3.5
[ 3 ] ia: 3   ca: d    da: 5.25
```

模板机制也支持面向对象的程序设计，类模板可以作为基类或派生类。下面是一个带有范围检查的 Array 类模板的定义：

```
#include <cassert>
#include "Array.h"
template <class elemType>
class ArrayRC : public Array<elemType> {
public:
    ArrayRC( int sz = Array<elemType>::DefaultArraySize )
    : Array< elemType >( sz ){};

    ArrayRC( elemType *ia, int sz )
    : Array< elemType >( ia, sz ) {}

    ArrayRC( const ArrayRC &rhs )
    : Array< elemType >( rhs ) {}

    virtual elemType&
```

```
        operator[]( int index )
        {
            assert( index >= 0 && index < Array<elemType>::size() );
            return ia[ index ];
        }
    private:
        // ...
    };
```

每个 ArrayRC 类的实例化过程都会实例化相应的 Array 类模板的实例。例如，下面的定义：

```
    ArrayRC<int> ia_rc( 10 );
```

引起 Array 类和 ArrayRC 类的一个 int 实例被实例化。ia_rc 同上一节的非模板实例相同。为了说明这一点，我们重写前面的程序来练习 Array 和 ArrayRC 类模板类型。首先，为了支持语句：

```
    // 现在 swap()必须也是一个模板
    swap( ia1, 1, ia1.size() );
```

我们必须将 swap()定义成一个函数模板。例如：

```
    #include "Array.h"
    template <class elemType>
    void swap( Array<elemType> &array, int i, int j )
    {
        elemType tmp = array[ i ];
        array[ i ] = array[ j ];
        array[ j ] = tmp;
    }
```

每个 swap()调用都会根据数组的类型产生适当的实例。下面是重新改写之后的 main()函数。它使用了 Array 和 ArrayRC 类模板：

```
    #include <iostream>

    #include "Array.h"
    #include "ArrayRC.h"

    template <class elemType>
    inline void
    swap( Array<elemType> &array, int i, int j )
    {
        elemType tmp = array[ i ];
        array[ i ] = array[ j ];
        array[ j ] = tmp;
    }

    int main()
    {
        Array<int> ia1;
        ArrayRC<int> ia2;
        cout << "swap() with Array<int> ia1\n";
        int size = ia1.size();
```

```
        swap( ia1, 1, size );
        cout << "swap() with ArrayRC<int> ia2\n";
        size = ia2.size();
        swap( ia2, 1, size );
        return 0;
    }
```

程序的输出结果与非模板的 IntArray 类实现相同。

练习 2.13

给出下列类型声明：

```
template <class elemType> class Array;
enum Status { ... };
typedef string *Pstring;
```

如果存在的话，下面哪些对象的定义是错误的？

```
(a) Array< int*& > pri( 1024 );
(b) Array< Array<int> > aai( 1024 );
(c) Array< complex< double > > acd( 1024 );
(d) Array< Status > as( 1024 );
(e) Array< Pstring > aps( 1024 );
```

练习 2.14

重写下面的类定义，使它成为一个类模板：

```
class example1 {
public:
    example1( double min, double max );
    example1( const double *array, int size );

    double& operator[]( int index );
    bool operator==( const example1& ) const;

    bool insert( const double*, int );
    bool insert( double );

    double min() const { return _min; };
    double max() const { return _max; };

    void min( double );
    void max( double );
    int count( double value ) const;

private:
    int size;
    double *parray;
    double _min;
    double _max;
};
```

练习 2.15

给出如下的类模板：

```
template <class elemType>
class Example2 {
public:
     explicit Example2( elemType val = 0 )
     : _val( val ){}

     bool min( elemType value ) { return _val < value; }
     void value( elemType new_val ) { _val = new_val; }
     void print( ostream &os ) { os << _val; }
private:
     elemType _val;
};

template<class elemType>
ostream& operator<< ( ostream &os, const Example2<elemType> &ex )
{ ex.print( os ); return os; }
```

如下这样写会发生什么事情？

```
(a) Example2< Array<int>* > ex1;
(b) ex1.min( &ex1 );
(c) Example2< int > sa( 1024 ), sb;
(d) sa = sb;
(e) Example2< string > exs( "Walden" );
(f) cout << "exs: " << exs << endl;
```

练习 2.16

在 Example2 的定义中，我们写：

```
explicit Example2( elemType val = 0 )
: _val( val ){}
```

其意图是指定一个缺省值，以便用户可以写：

```
Example2< Type > ex1( value );
Example2< Type > ex2;
```

但是，我们的实现把 Type 限制在一个“不能用 0 进行初始化的类型”的子集中（例如，用 0 初始化一个 string 类型，就是一个错误）。[7]类似的情况是，如果 Type 不支持输出操作符，那么 print()调用就会失败（因此，Example2 的输出操作符也会失败）。如果 Type 不支持小于操作符，那么 min()调用就会失败。

C++语言本身并没有提供可以指示在实例化模板时 Type 有哪些隐含限制的方法。在最坏的情况下，当程序编译失败时程序员才发现这些限制。你认为 C++语言应该支持限制 Type 的语法吗？如果你认为应该的话，请说明语法，并用它重写 Example2 的定义。如果认为不需要，请说明理由。

[7] 通常解决这个问题的做法是：Example2(elemType nval = elemType()): _val(nval) {}。

练习 2.17

在上一个练习中，我们说如果 Type 不支持输出操作符和小于操作符，那么对 print()和 min()的调用就会出错。在标准 C++中，错误的产生不是发生在类模板被创建的时候，而是在 print()与 min()被调用的时候。你认为这样的语义正确吗？是否应该在模板定义中标记这个错误？为什么？

2.6 基于异常的设计

异常（exception）是指在运行时刻程序出现的反情形，例如数组下标越界、打开文件失败以及可用动态内存耗尽等等。程序员一般有自己的处理异常的风格，这导致了不同的编码习惯，因而很难整合到一个单一的应用程序中。

异常处理（exception handling）为“响应运行时刻的程序异常”提供了一个标准的语言级的设施，它支持统一的语法和风格，也允许每个程序员进行微调。异常处理使得我们个需要在程序中处处显式地测试异常状态，从而可以将测试异常状态的代码抽取出来，放入指定的、显式标记的代码块中，因此异常处理设施大人地减少了程序代码的长度和复杂度。

异常处理机制的主要构成如下：

1．程序中异常出现的点。一旦识别出程序异常，就会导致抛出（raise 或 throw）异常。与异常被抛出时，正常的程序就被挂起，直到异常被处理完毕。在 C++中，异常的抛出由 throw 表达式来执行。例如，在下面的程序段中，一个 string 类型的异常被抛出来以便响应打开文件失败异常。

```
if ( ! infile ) {
     string errMsg( "unable to open file: " );
     errMsg += fileName;
     throw errMsg;
}
```

2．程序中异常被处理的点。典型地，程序异常的抛出与处理位于独立的函数或成员函数调用中。找到处理代码通常要涉及到展开程序调用栈（Program call stack）。一旦异常被处理完毕，就恢复正常的程序执行。但不是在发生异常的地方恢复执行过程，而是在处理异常的地方恢复执行过程。C++中，异常的处理由 catch 子句来执行。例如，下面的 catch 子句处理在第 1 项中被抛出的异常：

```
catch( string exceptionMsg ) {
     log_message( exceptionMsg );
     return false;
}
```

catch 子句与 try 块相关联。一个 try 块用一个或多个 catch 子句将一条或多条语句组织起来。例如，下面是函数 stats()：

```
int*
stats( const int *ia, int size )
{
     int *pstats = new int[ 4 ];
```

```
        try {
            pstats[ 0 ] = sum_it( ia, size );
            pstats[ 1 ] = min_val( ia, size );
            pstats[ 2 ] = max_val( ia, size );
        }
        catch( string exceptionMsg )
            {/* 处理异常的代码 */}
        catch( const statsException &statsExcp )
            {/* 处理异常的代码 */}

        pstats[ 3 ] = pstats[ 0 ]/size;
        do_something( pstats );

        return pstats;
    }
```

在 stats()内部有 4 条语句在 try 块之外，在下面两条语句完成之前，可能会有异常被抛出：

```
(1) int *pstats = new int[ 4 ];
(2) do_something( pstats );
```

在语句(1)中，new 表达式可能会失败。如果发生了这样的情况，标准库将产生 bad_alloc 标准异常。由于 bad_alloc 是在 try 块之外被抛出的，所以在 stats()中并没有试图要处理它，于是函数将终止：pstats 没有被初始化，stats()中后面的语句也不会被执行。异常机制承接了控制权开一直保持直到异常处理完毕。

在语句(2)中，在 do_something()中的语句，以及在 do_something()中被调用的语句，或 do_something()函数中被调用的函数所调用的语句等等，都可能会抛出异常。在从 do_something()调用开始的函数调用链返回之前，这个异常可能（也可能不）会被捕捉到。如果异常被处理了，那么 stats()继续执行，就像什么也没有发生过一样。如果在 do_something()结束之前，异常没有被处理，那么 stats()也会被终止，因为异常发生在 try 块之外。

（注意：如果 size 等于 0，那么：

```
pstats[ 3 ] = pstats[ 0 ]/size;
```

将导致一个除以 0 的除法。尽管这将导致向 pstats[3]赋一个未定义的数据值，但是对于除以 0 并没有标准异常被抛出。）

try 块内的三条语句会怎么样呢？不同的行为区别如下：如果在 stats()里面 sum_it()、min_val()及 max_val()终止之后，被抛出的异常是活动的（有效的），那么系统不是简单地终止 stats()，而是顺序地检查与 try 块相关联的 catch 子句，试图处理被抛出来的异常。假设 sum_it()抛出如下异常：

```
throw string( "internal error: adump27832" );
```

则 pstats[0]不会被初始化，在 try 块中接下来的两条语句也不会被执行，异常机制意识到 sum_it()是在 try 块中被调用的，因而它将检查相关的两条 catch()子句。

系统根据被抛出来的异常与 catch 子句中异常类型的匹配情况来选择 catch 子句。在本例中，异常是 string 类型，与下面的 catch 子句相匹配：

```
catch( string exceptionMsg )
    {/* 处理异常的代码 */}
```

系统把控制传递给被选中的 catch 子句体，其中的语句将顺序执行。完成之后，除非在处理该异常的子句中又抛出异常，否则控制将被传回到程序的当前点上。例如，如果我们已经这样写：

```
catch( string exceptionMsg )
{
    cerr << "stats(): exception occurred: "
    << exceptionMsg << endl;
    pstats[0] = pstats[1] = pstats[2] = 0;
}
```

那么，在 catch 子句完成时，控制将被传递给 catch 子句集后面的可执行语句。本例中，语句：

```
pstats[ 3 ] = pstats[ 0 ]/size;
```

被执行，然后是 do_something()调用，以及返回 pstats。而调用 stats()的函数根本不知道曾经有异常被抛出。

一段更为合理的异常处理代码可能如下所示：

```
c atch( string exceptionMsg )
{
    cerr << "stats(): exception occurred: "
    << exceptionMsg
    << " unable to stat array "
    << endl;
    delete [] pstats;
    return 0;
}
```

在上面的代码中，catch 子句直接把控制返回给外面的调用函数。我们希望外面的函数在把返回值用作索引数组之前，先测试它是否为 0。

如果 try 块内抛出的异常不能被相关联的 catch 子句处理，那么函数将被终止。然后，异常机制再在调用 stats()的函数中查找处理代码。

如果异常机制按照函数被调用的顺序回查每个函数直到 main()函数，仍然没有找到处理代码，那么它将调用标准库函数 terminate()。缺省情况下，terminate()函数结束程序。

一种特殊的、能够处理全部异常的 catch 子句如下：

```
catch( ... )
{
    // 处理所有异常，虽然它无法
    // 访问异常对象
}
```

我们可以把它看作是一种捕捉所有异常（catch-all）的 catch 子句。

异常处理机制为统一地处理程序异常提供了语言一级的设施。第 11 章与 19 章将进一步详细讨论。另一本配套的书《Inside the C++ Object Model》（[LIPPMAN96a]）中讨论了实现与性能的话题。Jos é e Lajoie 在[LIPPMAN96b]中的文章“Exception Handling: Behind the Scenses”对此也有讨论。[LIPPMAN96b] 中 Tom Cargill 的文章“Exception Handling: A False Sense of

Security”则对使用异常处理过程中易犯的错误做了很好的讨论。

练习 2.18

下面的函数对可能的非法数据以及可能的操作失败完全没有提供检查。找出程序中所有可能出错的地方。（本练习中；我们不关心可能会抛出的异常。）

```
int *alloc_and_init( string file_name )
{
    ifstream infile( file_name );
    int elem_cnt;
    infile >> elem_cnt;
    int *pi = allocate_array( elem_cnt );

    int elem;
    int index = 0;
    while ( cin >> elem )
        pi[ index++ ] = elem;

    sort_array( pi, elem_cnt );
    register_data( pi );

    return pi;
}
```

练习 2.19

alloc_and_init()函数会调用到下面的函数，如果这些函数调用失败了，它们将抛出相应类型的异常：

```
allocate_array() noMem
sort_array() int
register_data() string
```

请在合适的地方插入一个或多个 try 块以及相应的 catch 子句来处理这些异常。在 catch 子句中只需简单地输出错误的出现情况。

练习 2.20

检查在练习 2.18 中的函数 alloc_and_init()中所有可能出现的错误，指出哪些错误会抛出异常。修改该函数（或用练习 2.18 的版本，或用练习 2.19 的版本）来抛出对被识别的异常（抛出文字串就可以了）。

2.7 用其他名字来命名数组

把代码分发给其他部门的诸多困难中，有一个是我们不知道全局名字会有什么样的影响。例如，在 Intel 公司，有人写了：

```
class Array { ... };
```

那么他就不能在相同的程序中既使用上面的 Array 类，又使用我们实现的那个 Array 类

名字的可视性使这两份实现代码相互排斥。

在 C++标准化之前，解决这个问题的传统做法是在全局可见的名字前加上一个唯一的字符串前缀。例如，我们可以这样发行数组 Array 类：

```
class Cplusplus_Primer_Third_Edition_Array { ... };
```

虽然这个名字可能是惟一的（我们不能保证这一点），但是写起来并不方便。标准 C++的名字空间机制是 C++语言针对这个问题提供的语言一级的解决方案。

名字空间机制允许我们封装名字，否则这些名字就有可能会污染（影响）全局名字空间（pollute the global namespace）。一般来说，只有当我们希望自己的代码被外部软件开发部门使用时，才使用名字空间。例如，我们可以这样封装 Array 类：

```
namespace Cplusplus_Primer_3E {
template <class elemType>
    class Array { ... };
    // ...
}
```

关键字 namespace 后面的名字标识了一个名字空间，它独立于全局名字空间，我们可以在里面放一些希望声明在函数或类之外的实体。名字空间并不改变其中的声明的意义，只是改变了它们的可视性。在继续讨论之前，先扩展我们的可用名字空间集：

```
namespace IBM_Canada_Laboratory {
    template <class elemType>
        class Array { ... };
    class Matrix { ... };
    // ...
}

namespace Disney_Feature_Animation {
    class Point { ... };
    template <class elemType, int size>
        class Array { ... };
    // ...
}
```

如果名字空间内的声明对程序而言不是立即可见的，那么我们怎样访问它们呢？我们可以使用限定修饰名字符（qualified name notation），格式如下：

```
namespace_identifier::entity_name;
```

如在：

```
Cplusplus_Primer_3E::Array<string> text;

IBM_Canada_Laboratory::Matrix mat;

Disney_Feature_Animation::Point origin( 5000, 5000 );
```

虽然 Disney_Feature_Animation、IBM_Canada_Laboratory 以及 Cplusplus_Primer_3E 都能够唯一地标识相应的名字空间，但是，如果在程序中经常这样使用。则多少会有些麻烦。使用名字空间标识符如 P3E、DFA 或 IBM_CL 会更方便一些，但是它们表达的信息相对比较少，同时也增加了名字冲突的可能性。为了提供有意义的名字空间标识符，同时程序员又能很方

便地访问在名字空间内定义的实体，C++提供了别名设施。

名字空间别名（namespace alias）允许用一个可替代的、短的或更一般的名字与一个现有的名字空间关联起来。例如：

```
// 提供一个更一般化的别名
namespace LIB = IBM_Canada_Laboratory;
// 提供一个更短的别名
namespace DFA = Disney_Feature_Animation;
```

然后这个别名就可以用作原始名字空间的同义词。例如：

```
#include "IBM_Canada.h"

namespace LIB = IBM_Canada_Laboratory;
int main()
{
    LIB::Array<int> ia(1024);
    // ...
}
```

别名也可以用来封装正在使用的实际名字空间。例如，在此情形下，我们可以通过改变分配给别名的名字空间，来改变所使用的声明集，而无需改变“通过别名访问这些声明”的实际代码。例如：

```
namespace LIB = Cplusplus_Primer_3E;

int main()
{
    // 在这种情况下，下面的声明无须改变
    LIB::Array<int> ia(1024);
    // ...
}
```

但是，如果要让这项技术在实际工作中发挥作用，那么两个名字空间中的声明必须提供同样的接口。例如，下面的代码就不能工作，因为 Disney 的 Array 类需要一个类型参数和一个数组长度参数：

```
namespace LIB = Disney_Feature_Animation;

int main()
{
    // 不再是一个有效的声明
    LIB::Array<int> ia(1024);
    // ...
}
```

程序员常常希望在访问名字空间内声明的名字时不加限定修饰符。即使我们已经为名字空间标识符提供了较短的别名，在每次访问该名字空间内声明的名字时也都要进行限定，还是太麻烦！using 指示符（using directive）使名字空间内的所有声明都可见，这样这些声明能够不加限定地使用。例如：

```
#include "IBM_Canada_Laboratory.h"
```

```
// 使所有的名字都可见
using namespace IBM_Canada_Laboratory;
int main()
{
    // ok: IBM_Canada_Laboratory::Matrix
    Matrix mat( 4,4 );

    // ok: IBM_Canada_Laboratory::Array
    Array<int> ia( 1024 );
    // ...
}
```

using 与 namespace 都是关键字。被引用的名字空间必须已经被声明了，否则会引起编译错误。

using 声明（using declaration）提供了选择更为精细的名字可视性机制。它允许使名字中间中的单个声明可见。例如：

```
#include "IBM_Canada_Laboratory.h"

// 只让 Matrix 可见
using IBM_Canada_Laboratory::Matrix;
int main()
{
    // ok: IBM_Canada_Laboratory::Matrix
    Matrix mat(4,4);

    // error: IBM_Canada_Laboratory::Array not visible
    Array<int> ia( 1024 );
    // ...
}
```

为了防止标准 C++库的组件污染用户程序的全局名字空间，所有标准 C++库的组件都声明在一个被称为 std 的名字空间内。正如在第 1 章中提到的，即使我们在程序文本文件中包含了 C++库头文件，头文件中声明的组件在我们的文本文件中也不是自动可见的。例如，在标准 C++中，下面的代码实例就不能正常编译：

```
#include <string>

// 错误: string 不是可见的
string current_chapter = "A Tour of C++";
```

在<string>头文件中的所有声明都包含在名字空间 std 中。正如第 1 章所提到的，我们可以用“在#include 预处理器指示符后面加上 using 指示符”的办法，使 C++头文件<string>中的、在名字空间 std 中声明的组件对于我们的程序都是可见的：

```
#include <string>

using namespace std;
// ok: string 是可见的
string current_chapter = "A Tour of C++";
```

为了使在名字空间 std 中声明的名字在我们的程序中可见，指示符 using 通常被看作是一种比较差的选择方案。在上面的例子中，指示符 using 使头文件<string>中声明的、并且位于

名字空间 std 中的所有组件在程序文本文件中都是可见的，这又将全局名字空间污染问题带回来了。而这个问题正是 std 名字空间首先要努力避免的，它增加了“C++标准库组件的名字”与“我们程序中声明的全局名字”冲突的机会。

现在，我们对名字空间机制已经有了一些了解，知道有另外两种机制可代替指示符 using，使我们能够引用到隐藏在名字空间 std 中的名字 string。我们可以使用限定的名字，例如：

```
#include <string>
// ok: 使用限定的名字
std::string current_chapter = "A Tour of C++";
```

或如下使用 using 声明：

```
#include <string>
using std::string;
// ok: 上面的 using 声明使 string 可见
string current_chapter = "A Tour of C++";
```

为了使用名字空间中声明的名字，建议使用带有精细选择功能的 using 声明代替 using 指示符。这也正是本书的代码示例中没有出现 using 指示符的另一个原因。理想情况下，每一个代码示例对它所用到的每个库组件都应该有一个 using 声明。为了限制例子代码的长度，也因为本书的许多例子是在不支持名字空间的情况下被编译的，所以 using 声明就没有显示出来。8.6 节将进一步讨论怎样对标准 C++库的组件使用 using 声明。

在接下来的四章中，我们将讲述另外四个类的设计。第 3 章讲的是 String（字符串）类的设计与实现，第 4 章介绍整数 Stack（栈）类的设计，第 5 章是 List（列表）类，第 6 章对第 4 章定义的 Stack（栈）类进行重新设计。名字空间机制允许我们把每个类放在单独的头文件中，但是仍然能把它们的名字封装到单个 Cplusplus_Primer_3E 名字空间中。在第 8 章中，我们将讨论这项技术，并对名字空间作更多的介绍。

练习 2.21

给出如下名字空间定义：

```
namespace Exercise {
    template <class elemType>
        class Array { ... };

    template <class Etype>
        void print( Array< Etype > );

    class String { ... };
    template <class listType>
        class List { ... };
}
```

以及下面的程序；

```
int main() {
    const int size = 1024;
    Array< String > as( size )
```

```
        List< int > il( size );
        // ...

        Array< String > *pas = new Array<String>(as);
        List <int> *pil = new List<int>(il);

        print( *pas );
    }
```

同为类型名被封装在名字空间中，所以当前程序编译失败。把程序修改为：

1．用限定名字修饰符来访问名字空间 Exercise 中的类型定义。

2．使用 using 声明来访问类型定义。

3．用名字空间别名机制。

4．用 using 指示符。

2.8 标准数组——向量

正如我们已经看到的，尽管 C++内置的数组支持容器的机制，但是它不支持容器抽象的语义。为了在这样的层次上编写程序，在标准 C++之前，我们要么从某个途径获得这样的类。要么自己实现这样的类。在标准 C++中，数组类是 C++标准库的一部分，现在它不叫数组，而叫向量（vector）了。

当然，向量是一个类模板，所以我们这样写：

```
vector<int> ivec( 10 );
vector<string> svec( 10 );
```

上面的代码分别定义了一个包含 10 个整型对象的向量和一个包含 10 个字符串对象的向量。

在我们实现的 Array 类模板与 vector 类模板的实现之间有两个主要区别。第一个区别是 vector 类模板支持“向现有的数组元素赋值”的概念以及“插入附加元素”的概念——即 vector 数组可以在运行时刻动态增长（如果程序员希望使用这个特性的话）。第二个区别是更加广泛，代表了设计方法的重要转变。vector 类不是提供一个巨大的“可以适用于向量”的成员操作集，如 sort()、min()、max()及 find()等等，而是只提供了一个最小集：如等于、小于操作符、size()、empty()等操作。而一些通用的操作如 sort()、min()、max()和 find()等等，则是作为独立的泛型算法（generic algorithm）被提供的。

要定义一个向量，我们必须包含相关的头文件：

```
#include < vector >
```

下面都是 vector 对象的合法定义：

```
#include < vector >

// 创建 vector 对象的各种方法
vector<int> veco; // 空的 vector
```

```
const int size = 8;
const int value = 1024;

// size 为 8 的 vector
// 每个元素都被初始化为 0
vector<int> vec1( size );

// size 为 8 的 vector
// 每个元素都被动始化为 1024
vector<int> vec2( size, value );

// vtc3 的 size 为 4
// 被初始化为 ia 的 4 个值
int ia[4] = { 0, 1, 1, 2 };
vector<int> vec3( ia, ia+4 );

// vec4 是 vec2 的拷贝
vector<int> vec4( vec2 );
```

既然定义了向量，我们就需要遍历里边的元素。与 Array 类模板一样，标准 vector 类模板也支持使用下标操作符，例如：

```
#include <vector>

extern int getSize();
void mumble()
{
    int size = getSize();
    vector< int > vec( size );

    for ( int ix = 0; ix < size; ++ix )
        vec[ ix ] = ix;
    // ...
}
```

另外一种遍历方法是使用迭代器对（iterator pair）来标记向量的起始处和结束处。迭代器是一个支持指针类型抽象的类对象。vector 类模板提供了一对操作 begin()和。end()，它们分别返回指向“向量开始处”和“结束处后 1 个”的迭代器。这一对迭代器合起来可以标记出待遍历元素的范围。例如，下面的代码是前面代码段的一个等价实现：

```
#include < vector >

extern int getSize();
void mumble()
{
    int size = getSize();
    vector< int > vec( size );
    vector< int >::iterator iter = vec.begin();

    for ( int ix = 0; iter != vec.end(); ++iter, ++ix )
        *iter = ix;
```

```
        // ...
    }
```

iter 的定义：

```
    vector< int >::iterator iter = vec.begin();
```

将其初始值指向 vec 的第一个元素。iterator 是 vector 类模板中用 typedef 定义的类型，而这里的 vector 类实例包含 int 类型的元素。下面的代码使迭代器指向 vector 的下一个元素：

```
    ++iter
```

下面的代码解除迭代器的引用，以便访问实际的元素：

```
    *iter
```

能够应用到向量上的操作惊人地多，但是它们并不是作为 vector 类模板的成员函数提供的。它们是以一个独立的泛型算法集的形式，由标准库提供。下面是一组可供使用的泛型算法的示例：

- **搜索**（search）算法：find()、find_if()、search()、binary_search()、count()和 count_if()。
- **分类排序**（sorting）与通用排序（ordering）算法：sort()、partial_sort()、merge()、partition()、rotate()、reverse()和 random_shuffle()。
- **删除**（deletion）算法：unique()和 remove()。
- **算术**（numeric）算法：accumulate()、partial_sum()、inner_product()和 adjacent_difference()。
- **生成**（generation）和变异（mutation）算法：generate()、fill()、transformation()、copy()和 for_each()。
- **关系**（Relational）算法：equal()、min()和 max()。

泛型算法接受一对迭代器，它们标记了要遍历元素的范围。例如，ivec 是一个包含某种类型元素的、某个长度的向量，要用 sort()对它的全部元素进行排序。我们只需简单地这样写；

```
    sort( ivec.begin(), ivec.end() );
```

只想对 ivec 向量的前面一半进行排序，可以这样写：

```
    sort( ivec.begin(), ivec.begin()+ivec.size()/2 );
```

泛型算法还能接受指向内置数组的指针对，例如，已知数组：

```
    int ia[7] = { 10, 7, 9, 5, 3, 7, 1 };
```

我们可以如下对整个数组排序：

```
    sort( ia, ia+7 );
```

我们还可以只对前四个元素排序：

```
    sort( ia, ia+4 );
```

要使用这些算法，我们必须包含与它们相关的头文件：

```
    #include <algorithm>
```

下面的代码显示了怎样把各种各样的泛型算法应用到 vector 类对象上：

```
#include <vector>
#include <algorithm>
#include <iostream>
int ia[ 10 ] = {
     51, 23, 7, 88, 41, 98, 12, 103, 37, 6 };

int main()
{
     vector< int > vec( ia, ia+10 );

     // 排序数组
     sort( vec.begin(), vec.end() );

     // 获取值
     int search_value;
     cin >> search_value;

     // 搜索元素
     vector<int>::iterator found;
     found = find( vec.begin(), vec.end(), search_value );
     if ( found != vec.end() )
          cout << "search_value found!\n";
     else cout << "search_value not found!\n";

     // 反转数组
     reverse( vec.begin(), vec.end() );

     // ...
}
```

标准库还提供了对 map 关联数组的支持，即数组元素可以被整数值之外的其他东西索引。例如，我们可以这样来支持一个电话目录：这个电话目录是电话号码的数组，但是它的元素可以由该号码所属人的姓名来索引：

```
#include <map>
#include <string>
#include "TelephoneNumber.h"

map< string, telephoneNum > telephone_directory;
```

在第 6 章我们将看到 vector、map 以及标准 C++库支持的其他容器类型，本书将通过一个文本查询系统的实现来说明这些类型的用法。而第 12 章会讲解泛型算法。附录按字母顺序提供了每个算法的解释及其用法。

本章大致地讲述了 C++为数据抽象（基于对象的程序设计）、面向对象的程序设计、泛型程序设计（模板、容器类型以及泛型算法）、大型程序设计（异常处理与名字空间）提供的支持、而本书余下的部分将更详细地介绍这些内容，逐步讲解 C++中基本、但又非常先进的特性。

练习 2.22

解释每个 vector 定义的结果：

```
string pals[] = {
     "pooh", "tigger", "piglet", "eeyore", "kanga" };

(a) vector<string> svec1( pals, pals+5 );
(b) vector<int> ivec1( 10 );
(c) vector<int> ivec2( 10, 10 );
(d) vector<string> svec2( svec1 );
(e) vector<double> dvec;
```

练习 2.23

已知下列函数声明，请实现 min()的函数体，它查找并返回 vec 的最小元素。要求首先使用“索引 vec 中元素的 for 循环”来实现 min()，然后，再使用“通过迭代器遍历 vec 的 for 循环”来实现 min()：

```
template <class elemType>
elemType
min( const vector<elemType> &vec );
```

第二篇

基本语言

我们编写的程序以及所保存的程序数据在计算机的内存中是以二进制位序列的方式存放的。位（bit ）是含有 0 或 1 值的一个单元。在物理上它的值是个负或正电荷。典型的计算机内存段如下所示：

```
00011011011100010110010000111011 ...
```

在这个层次上，位的集合没有结构，很难以某种意义来解释这些位序列。但是，偶然情况下（尤其是当我们访问实际的机器硬件时），我们会因为需要或者为了方便在单独的位或者位集合的层次上编写程序。C++语言提供了一套位操作符以支持位操作，以及一个位集合（bitset）容器类型，可以用来声明含有位集合的对象（第 4 章将讨论这些操作符以及位集合容器类型）。

为了能够从整体上考虑这些位，我们给位序列强加上结构的概念，这样的结构被称作字节（byte）和字（word）、通常，一个字节由 8 位构成，而一个字由 32 位构成，或者说是 4 个字节（但是，工作站操作系统现在正在朝 64 位系统的方向转换）。不同计算机中的字的大小也不尽相同。我们说这个值是依赖于机器的（machine-dependent，或者说机器相关的）。下面的这张图说明了位流怎样被组织成四个对寻址的字节行。

1024	0	0	0	1	1	0	1	1
1032	0	1	1	1	0	0	0	1
1040	0	1	1	0	0	1	0	0
1048	0	0	1	1	1	0	1	1

通过对内存进行组织，我们可以引用特定的位集合。因此，我们可以说“在地址 1024 上的字”或者“在地址 1040 上的字节”。例如，我们可以这样说，在地址 1032 上的字节不等于在地址 1048 上的字节。

但是，我们仍然不能讲出在地址 1032 处的内容的意义。为什么呢？因为不知道怎样解释这些位序列。为了说明在地址 1032 上的字节的意义，我们必须知道这些值代表的类型。

类型抽象使我们能够对一个定长的位序列进行有意义的解释。C++提供了一组预定义的数据类型，如字符型、整型、浮点型，以及一组基本的数据抽象，如 string、vector 和复数。它还提供了一组操作符（或称运算符），如加、减、等于。小于操作符等来操纵这些类型。C++还为程序流控制提供了为数不多的一组语句，如 while 循环和 if 语句。这些要素构成了一个符号系统，人们已经用它写出了许多大型的，复杂的实用系统。掌握 C++的第一步就是要理解这些基本的组件，这是本书第二篇的主题。

第 3 章将概括说明预定义的和扩展的数据类型，并讲解构造新数据类型的机制，主要是在 2.3 节中介绍的类机制。第 4 章将集中讨论对表达式的支持，并讲解预定义操作符、类型转换、操作符优先级以及相关的问题。程序的最小独立单元是语句，这是第 5 章的主题。第 6 章将介绍标准库容器类型，比如 vector 和 map，我们将通过一个文本查询系统的实现说明它们的用法。

3

C++数据类型

本章将概括介绍C++中预定义的内置的（built in）、或称基本的（primitive）数据类型。本章将以文字常量（literal constant）开始，如3.14159和“pi”，然后介绍符号变量（symbolic variable）和对象（object）的概念。C++程序中的对象必须被定义为某种特定的类型，本章的余下部分将介绍可以用来声明对象的各种类型。另外，我们还将把C++内置的对字符串与数组的支持与C++标准库提供的类抽象进行对比。虽然标准库中的抽象类型不是基本类型，但是它们也是使用C++程序的基础。我们希望尽早地介绍它们，以此来鼓励和说明它们的使用。我们把这些类型看作是基本内置类型和基本类抽象类型的扩展基础语言.

3.1 文字常量

C++预定义了一组数值数据类型，可以用来表示整数、浮点数和单个字符。此外，还预定义了用来表示字符串的字符数组。

- 字符型char，通常用来表示单个字符和小整数。它可以用一个机器字节来表示。
- 整型int、短整型short、长整型long，它们分别代表不同长度的整数值，典型情况下，short以半个字表示，int以一个机器字表示，而long为一个或两个机器字。（在32位机器中，int和long通常长度相同）。
- 浮点型float、双精度double和长双精度long double，分别表示单精度浮点数、双精度浮点数和扩展精度的浮点数值。典型情况下，float为一个字，double是两个字，long double为三个或四个字。

char，short，int和long称为整值类型（integral type）。整值类型可以有符号，也可以无符号。在有符号类型中，最左边的位是符号位，余下的位代表数值。在无符号类型中，所有的位都表示数值。如果符号位被置为1，数值被解释成负数；如果是0，则为正数。一个8位有符号的char可以代表从-128到127的数值，而一个无符号的char则表示0到255范围内的数值。

当一个数值，例如1，出现在程序中时，它被称为文字常量（literal constant）：称之为“文字”是因为我们只能以它的值的形式指代它，称之为“常量”是因为它的值不能被改变。

每个文字都有相应的类型。例如，0 是 int 型，而 3.14159 是 double 型的文字常量。文字常量是不可寻址的（nonaddressable），尽管它的值也存储在机器内存的某个地方，但是我们没有办法访问它们的地址。

整数文字常量可以被写成十进制、八进制或者十六进制的形式（这不会改变该整数值的位序列）。例如，20 可以写成下面三种形式中的任意一种：

```
20    // 十进制
024   // 八进制
0x14  // 十六进制
```

在整型文字常量前面加一个 0，该值将被解释成一个八进制数。而在前面加一个 0x 或 0X，则会使一个整型文字常量被解释成十六进制数。（第 20 章“输入/输出流库”将讨论八进制或十六进制形式的输出值。）

在缺省情况下，整型文字常量被当作是一个 int 型的有符号值。我们可以在文字常量后面加一个“ L”或“l”（字母 L 的大写形式或者小写形式），将其指定为 long 类型。一般情况下，我们应该避免使用小写字母，因为它很容易被误当作数字 1。类似地，我们可以在整型文字常量的后面加上“u”或“U”，将其指定为一个无符号数。此外，我们还可以指定无符号 long 型的文字常量。例如：

```
128u   1024UL    1L    8Lu
```

浮点型文字常量可以被写成科学计数法形式或普通的十进制形式。使用科学计数法，指数可写作“e”或“E”。浮点型文字常量在缺省情况下被认为是 double 型，单精度文字常量由值后面的“f”或“F”来标示。类似地，扩展精度中值后面跟的“l”或“L”来指示。（注意，“f”、“F”、“l”、“L”后缀只能用在十进制形式中。）例如：

```
3.14159F  0.1f      12.345L  0.0
3e1       1.0E-3    2.       1.0L
```

单词 true 和 false 是 bool 型的文字常量。例如，可以这样写：

```
true false
```

可打印的文字字符常量可以写成用单引号括起来的形式。例如：

```
'a'       '2'       ','       ' ' (空格)
```

一部分不可打印的字符、单引号、双引号以及反斜杠可以用如下的转义序列来表示（转义序列以反斜杠开头）：

```
newline(换行符)     \n
horizontal tab(水平制表键) \t
vertical tab(垂直制表键)\v
backspace(退格键)   \b
carriage return (回车键)   \r
formfeed (进纸键)   \f
alert (beel) (响铃符)    \a
backslash (反斜杠键)     \\
question mark (问号)     \?
single quote (单引号)    \'
double quote (双引号)    \"
```

一般的转义序列采用如下格式：

```
\ooo
```

这里的 ooo 代表三个八进制数字组成的序列。八进制序列的值代表该字符在机器字符集里的数字值。下面的示例使用 ASCII 码字符集表示文字常量：

```
\7 (bell) \14 (newline)
\0 (null) \062 ('2')
```

另外，字符文字前面可以加“L”，例如：

```
L'a'
```

这称为宽字符文字，类型为 wchar_t。宽字符常量用来支持某些语言的字符集合，如汉语、日语，这些语言中的某些字符不能用单个字符来表示。

字符串文字常量由零个或多个用双引号括起来的字符组成。不可打印字符可以由相应的转义序列来表示，而一个字符串文字可以扩展到多行。在一行的最后加上一个反斜杠，表明字符串文字在下一行继续。例如：

```
"" (空字符串)
"a"
"\nCC\toptions\tfile.[cC]\n"
"a multi-line \
string literal signals its \
continuation with a backslash"
```

字符串文字的类型是常量字符数组。它由字符串文字本身以及编译器加上的表示结束的空（null）字符构成。例如：

```
'A'
```

代表单个字符‘A’，下面则表示单个字符 A 后面跟一个空字符。

```
"A "
```

空字符是 C 和 C++用来标记字符串结束的符号。

正如存在宽字符文字，比如：

```
L'a'
```

同样地，也有宽字符串文字，它仍然以“L”开头，如：

```
L"a wide string literal"
```

宽字符串文字的类型是常量宽字符的数组。它也有一个等价的宽空字符作为结束标志。

如果两个字符串或宽字符串在程序中相邻，C++就会把它们连接在一起，并在最后加上一个空字符。例如；

```
"two"     "some"
```

它的输出结果是“twosome”。如果将一个字符串常量与一个宽字符串常量连接起来，会发生什么后果？例如：

```
// 不建议这样使用
"two"     L"some"
```

结果是未定义的（undefined）——即，没有为这两种不同类型的连接定义标准行为。使用未定义行为的程序被称作是不可移植的。虽然程序可能在当前编译器下能正确执行，但是不能保证相同的程序在不同的编译器、或当前编译器的以后版本下编译后，仍然能够正确执行。在本来能够运行的程序中跟踪这类问题是一件很令人不快的任务。因此，建议不要使用未定义的程序特性。我们会在合适的时候指出这样的特性。

练习 3.1

说明下列文字常量的区别：

```
(a) 'a', L'a', "a", L"a"
(b) 10, 10u, 10L, 10uL, 012, 0xC
(c) 3.14, 3.14f, 3.14L
```

练习 3.2

下列语句哪些是非法的？

```
(a) "Who goes with F\144rgus?\014"
(b) 3.14e1L
(c) "two" L"some"
(d) 1024f
(e) 3.14UL
(f) "multiple line
    comment"
```

3.2 变量

假设有这样一个问题：计算 2 的 10 次方。我们首先想到的可能是：

```
#i nclude <iostream>
int main() {
    // 第一个解决方案
    cout << "2 raised to the power of 10: ";
    cout << 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2;
    cout << endl;
    return 0;
}
```

这样确实能够解决问题，但是，可能需要检查两到三遍，以确保正好有 10 个常数 2 参与乘法。这个程序产生正确的答案 1024。

接着，我们被要求算出 2 的 17 次方和 2 的 23 次方。每次都要修改程序确实很麻烦。但更糟糕的是，这样做经常会出错。修改后的程序常常会多乘或少乘了一个 2。最后我们又被要求生成 2 的从 0 到 15 次方的数值的表。使用文字常量需要与 32 行类似下面的格式：

```
cout << "2 raised to the power of X\t";
cout << 2 * ... * 2;
```

这里 X 随每对语句递增 1。

从某种角度来看，这样确实完成了任务。我们的老板不可能去看我们具体的做法，只要我们的结果正确并且及时就可以啦。实际上，在许多实际开发环境中，成功的主要评价标准是最后的结果，至于对处理过程的讨论则很可能被视为学究气、不切实际，总是得不到重视。

虽然这种蛮力型的方案也能解决问题，但是它总让人感到不快，而且有些危机感。这种方案吸引人的地方就是简单：我们明白需要做什么，虽然它常常很乏味。复杂的技术方案一般在开始阶段需要很多时间，这时常常会感觉什么都没有做。而且因为处理过程是自动的，所以就更有可能出错。

事情不可避免会出错。但好处在于，这些错误过程中，不但事情很快能完成，而且拓展了想像的空间。有时候，这个过程也挺有趣的。

在本例中，用来取代这种蛮力型的方案包括两部分内容：使用有名字的对象来读写每步的计算：引入一个控制流结构，以便在某个条件为真时，可以重复执行一系列语句。下面是一种“技术先进的”计算 2 的 10 次幂的程序：

```
#include <iostream.h>
int main()
{
    // int 类型的对象
    int value = 2;
    int pow = 10;
    cout << value << " raised to the power of "
        << pow << ": \t";

    int res = 1; // 保存结果

    // 循环控制语句：反复计算 res
    // 直至 cnt 大于 pow
    for ( int cnt=1; cnt <= pow; ++cnt)
        res = res * value;

    cout << res << endl;
}
```

value、pow、res 以及 cnt 是变量，它们允许对数值进行存储、修改和查询。for 循环使计算过程重复执行 pow 次。

虽然这种层次的通用化能使程序更加灵活，但是这样的程序仍然是不可重用的。我们必须进一步通用化：把计算指数值的那部分程序代码抽取出来，定义成一个独立的函数，以使其他函数能够调用它。例如：

```
int
pow( int val, int exp )
{
    for ( int res = 1; exp > u; --exp )
        res = res * val;

    return res;
}
```

现在，每个需要计算指数值的程序，都可以使用 pow()的实例，而不是重新实现它。我

们可以用如下的代码来生成 2 的幂的表：

```
#include <iostream>
extern int pow(int, int);

int main()
{
    int val = 2;
    int exp = 15;
    cout << "The Powers of 2\n";

    for ( int cnt=0; cnt <= exp; ++cnt )
    cout << cnt << ": "
        << pow(val, cnt) << endl;

    return 0;
}
```

实际上，这个 pow()的实现既不够健壮也不够通用。例如，如果指数是负数，该怎么办？如果是 1 000 000 呢？对于负数指数，我们的程序总是返回 1。对于一个非常大的指数，变量 int res 又小得不能够容纳这个结果。因此，对于一个大的指数将返回一个任意的、不正确的值。（在这种情况下，最好的解决方案是将返回值的类型修改为 double 类型）。从通用的角度来说，我们的程序应该能够处理整数和浮点数类型的底数和指数，甚至其他的类型。正如你所看到的，为一个未知的用户组写一个健壮的通用函数，比“实现一个特定的算法来解决眼前的问题”要复杂得多。pow()的实际实现代码见［PLAUGER92］。

3.2.1 什么是变量？

变量为我们提供了一个有名字的内存存储区，可以通过程序对其进行读、写和处理。C++中的每个符号变量都与一个特定的数据类型相关联，这个类型决定了相关内存的大小、布局、能够存储在该内存区的值的范围以及可以应用其上的操作集。我们也可以把变量说成对象（object）。下面是 5 个不同类型的变量定义（在后面我们会介绍变量定义的细节情况）：

```
int student_count;
double salary;
bool on_loan;
string street_address;
char delimiter;
```

变量和文字常量都有存储区，并且有相关的类型。区别在于变量是可寻址的（addressable）。对于每一个变量，都有两个值与其相关联：

1.它的数据值，存储在某个内存地址中。有时这个值也被称为对象的右值（rvalue，读做 are-value）。我们也可认为右值的意思是被读取的值（read value）。文字常量和变量都可被用作右值。

2.它的地址值——即，存储数据值的那块内存的地址。它有时被称为变量的左值（lvalue，读作 ell-value）。我们也可认为左值的意思是位置值（location value）。文字常量不能被用作左值。

在下面的表达式中：

```
ch = ch - 'O';
```

变量 ch 同时出现在赋值操作符的左边和右边。右边的实例被读取，与其相关联的内存中的数据值被读出。左边的 ch 用作写入。减操作的结果被存储在 ch 的位置值所指向的内存区中，原来的数据值会被覆盖。在表达式的右边，ch 和文字字符常量用作右值。在左边，ch 用作左值。

一般地，赋值操作符的左边总是要求一个左值。例如，下列的写法将产生编译错误：

```
// 编译错误：等号左边不是一个左值

// 错误：文字常量不是一个左值
0 = 1;

// 错误：算术表达式不是一个左值
salary + salary * 0.10 = new_salary;
```

在本书中，我们将会看到许多“左值和右值的用法会影响程序的语义行为和性能”的情况——尤其在“向函数传递值”或者“从函数返回值”的时候。

变量的定义会引起相关内存的分配。因为一个对象只能有一个位置，所以程序中的每个对象只能被定义一次。如果在一个文件中定义的对象需要在另一个文件中被访问，就可能会出现问题。例如：

```
// file module0.C
// 定义 fileName 对象
string fileName;

// ... 为 fileName 赋一个值

// file module1.C
// 需要使用 fileName 对象

// 喔：编译失败：
// 在 module1.C 中，fileName 未定义
ifstream input_file( fileName );
```

在 C++中，程序在使用对象之前必须先知道该对象。这对“编译器保证对象在使用时的类型正确性”是必需的。引用一个未知的对象将引起编译错误。在本例中，由于在 model1.C 中没有定义 fileName，所以该文件编译失败。

要编译 model1.C，必须让程序知道 fileName，但又不能引入第二个定义。我们可以通过声明（declaring）该变量来做到这一点：

```
// file module1.C
// 需经使用 fileName 对象
// 声明 fileName，也即，让程序知道它，
// 但又不引入第二个定义
extern string fileName;

ifstream input_file( fileName );
```

对象声明（declaration）的作用是使程序知道该对象的类型和名字。它由关键字 extern 以及跟在后面的对象类型以及对象的名字构成。（关于 extern 的全面介绍见 8.2 节。）声明

不是定义，不会引起内存分配。实际上，它只是说明了在程序之外的某处有这个变量的定义。

虽然一个程序只能包含一个对象的一个定义，但它可以包含任意数目的对象声明。比较好的做法，不是在每个使用对象的文件中都提供一个单独的声明，而是在一个头文件中声明这个对象，然后再在需要声明该对象的时候包含这个头文件。按照这种做法；如果需要修改对象的声明，则只需要修改一次，就能维持多个使用该对象的文件中声明的一致性。（8.2 节将对头文件有更多的说明。）

3.2.2 变量名

变量名：即变量的标识符（identifier），可以由字母、数字以及下划线字符组成。它必须以字母或下划线开头，并且区分大写字母和小写字母。语言本身对变量名的长度没有限制，但是为用户着想，它不应该过长。下面这个变量名虽然合法。但是太长了：

```
gosh_this_is_an_impossibly_long_name_to_type
```

C++保留了一些词用作关键字。关键字标识符不能再作为程序的标识符使用。我们已经见到过 C++语言的许多关键字。表 3.1 列出了 C++关键字全集。

表 3.1 C++关键字

asm	auto	bool	break	case
catch	char	class	const	const_cast
Continue	default	delete	do	double
dynamic_cast	else	enum	explicit	export
extern	false	float	for	friend
goto	if	inline	int	long
mutable	namespace	new	operator	private
protected	public	register	reinterpret_cast	return
short	signed	sizeof	static	static_cast
struct	switch	template	this	throw
true	try	typedef	typeid	typename
union	unsigned	using	virtual	void
volatile	wchar_t	while		

对于命名对象有许多已普遍接受的习惯，主要考虑因素是程序的可读性。

- 对象名一般用小写字母。例如，我们往往写成 index，而不写 INDEX。（一般把 Index 当作类型名，而 INDEX 则一般被看作常量值，通常用预处理器指示符#define 定义。）
- 标识符一般使用助记的名字——即，能够对程序中的用法提供提示的名字，如 on_loan 或 salary。至于是应写成 table 还是 tbl，这纯粹是风格问题，不是正确性的问题。

- 对于多个词构成的标识符，习惯上，一般在每个词之间加一个下划线，或内嵌的每个词第一个字母大写。例如，一般会写成 student_loan 或 studentLoan，而不是 studentloan（我在这里已经列出了所有三种形式）。一般有面向对象背景的人（ObjectOrientedBackground）喜欢用大写字母，而有 C 或过程化背景的人（C_or_procedural_background）则喜欢下划线。（再次说明，使用 isa、isA 或 is_a 只是个风格问题，与正确与否无关。）

3.2.3 对象的定义

一个简单的对象定义由一个类型指示符后面跟一个名字构成，以分号结束。例如：

```
double salary;
double wage;
int month;
int day;
int year;
unsigned long distance;
```

当同类型的多个标识符被定义的时候，我们可以在类型指示符后面跟一个由逗号分开的标识符列表。这个列表可跨越多行，最后以分号结束。例如，上面的定义可写成：

```
double salary, wage;
int month,
day, year;
unsigned long distance;
```

一个简单的定义指定了变量的类型和标识符，它并不提供初始值。如果一个变量是在全局域（global scope）内定义的，那么系统会保证给它提供初始值 0。在本例中，salary、wage、month、day、year 以及 distance 都被初始化为 0，因为它们都是在全局域内定义的。如果变量是在局部域（local scope）内定义的，或是通过 new 表达式动态分配的，则系统不会向它提供初始值 0。这些对象被称为是未初始化的（uninitialized）。未初始化的对象不是没有值，而是它的值是未定义的（undefined）。（与它相关联的内存区中含有一个随机的位串，可能是以前使用的结果。）

因为使用未初始化对象是个常见错误，而且很难发现，所以，一般建议为每个被定义的对象提供一个初始值。（在有些情况下，这不是必须的。然而，在你能够识别这些情况之前，为每个对象提供初始值是个安全的作法。）类机制通过所谓的缺省构造函数（2.3 节已经介绍过）提供了类对象的自动初始化。我们将在本章后面部分关于标准库 string 和复数类型（3.11 节和 3.15 节）的讨论中看到这一点。现在，请注意以下代码：

```
int main() {
    // 未初始化的局部对象
    int ival;

    // 通过 string 的缺省构造函数进行初始化
    string project;

    // ...
}
```

ival 是一个未初始化的局部变量，但 project 是一个已经初始化的类对象——被缺省的 string 类构造函数自动初始化。

初始的第一个值可以在对象的定义中指定。一个被声明了初始值的对象也被称为已经初始化的（initialized）。C++支持两种形式的初始化。第一种形式是使用赋值操作符的显式语法形式：

```
int ival = 1024;
string project = "Fantasia 2000";
```

在隐式形式中，初始值被放在括号中：

```
int ival( 1024 );
string project( "Fantasia 2001" );
```

在这两种情况中，ival 都被初始化为 1024，而 project 的初始值为“Fantasia 2000”。

逗号分隔的标识符列表同样也能为每个对象提供显式的初始值。语法形式如下：

```
double salary = 9999.99, wage = salary + 0.01;
int month = 08;

day = 07, year = 1955;
```

在对象的定义中，当对象的标识符在定义中出现后，对象名马上就是可见的，因此用对象初始化它自己是合法的，只是这样做不太明智。例如：

```
// 合法，但不明智
int bizarre = bizarre;
```

另外，每种内置数据类型都支持一种特殊的构造函数语法，可将对象初始化为 0。例如：

```
// 设置 ival 为 0，dval 为 0.0
int ival = int();

double dval = double();
```

下列定义中：

```
// int() applied to each of the 10 elements
vector< int > ivec( 10 );
```

函数 int()被自动应用在 ivec 包含的 10 个元素上。（2.8 节介绍了 vector。3.6 节与第 6 章将有详细讨论。）

对象可以用任意复杂的表达式来初始化，包括函数的返回值。例如：

```
#include <cmath>
#include <string>

double price = 109.99, discount = 0.16;
double sale_price( price * discount );
string pet( "wrinkles" );

extern int get_value();

int val =  get_value();
unsigned abs_val = abs( val );
```

abs()是标准 C 数学库中预定义的函数，它返回其参数的绝对值。get_value()是一个用户

定义的函数，它返回一个随机整数值。

练习 3.3

下列定义哪些是非法的？请改正之。

```
(a) int car = 1024, auto = 2048;
(b) int ival = ival;
(c) int ival( int() );
(d) double salary = wage = 9999.99;
(e) cin >> int input_value;
```

练习 3.4

区分左值与右值，并给出它们的例子。

练习 3.5

说明下列 student 和 name 两个实例的区别。

```
(a)  extern string name;
     string name( "exercise 3.5a" );
(b)  extern vector<string> students;
     vector<string> students;
```

练习 3.6

下列名字哪些是非法的？请改正之。

```
(a) int double = 3.14159;    (b) vector< int > _;
(c) string namespace;  (d) string catch-22;
(e) char 1_or_2 = '1'; (f) float Float = 3.14f;
```

练习 3.7

下面的全局对象定义和局部对象定义有什么区别(如果你认为有区别的话)？

```
string global_class;
int global_int;

int main() {
     int local_int;
     string local_class;
     // ...
}
```

3.3 指针类型

在 2.2 节中，我们简要地介绍了指针和动态内存分配。指针持有另一个对象的地址，使我们能够间接地操作这个对象。指针的典型用法是构建一个链接的数据结构，例如树（tree）

和链表（list），并管理在程序执行过程中动态分配的对象，以及作为函数参数类型，主要用来传递数组或大型的类对象。

每个指针都有一个相关的类型。不同数据类型的指针之间的区别不是在指针的表示上，也不在指针所持有的值（地址）上——对所有类型的指针这两方面都是相同的。[8]不同之处在于指针所指的对象的类型上。指针的类型可以指示编译器怎样解释特定地址上内存的内容，以及该内存区域应该跨越多少内存单元。

- 如果一个 int 型的指针寻址到 1000 内存处，那么在 32 位机器上，跨越的地址空间是 1000~1003。
- 如果一个 double 型的指针寻址到 1000 内存处，那么在 32 位机器上，跨越的地址空间是 1000~1007。

下面是指针定义的例子：

```
int *ip1, *ip2;
complex<double> *cp;
string *pstring;
vector<int> *pvec;
double *dp;
```

我们通过在标识符前加一个解引用操作符（*）来定义指针。在逗号分隔的标识符列表中，每个将被用作指针的标识符前都必须加上解引用操作符。在下面的例子中，lp 是一个指向 long 类型对象的指针，而 lp2 则是一个 long 型的数据对象，不是指针：

```
long *lp, lp2;
```

在下面的例子中，fp 是一个 float 型的数据对象，而 fp2 是一个指向 float 型对象的指针：

```
float fp, *fp2;
```

为清楚起见，最好写成：

```
string *ps;
```

而不是：

```
string* ps;
```

有可能发生的情况是，当程序员后来想定义第二个字符串指针时，他会错误地修改定义如下：

```
// 喔：ps2 不是一个字符串指针
string* ps, ps2;
```

当指针持有 0 值时，表明它没有指向任何对象，或持有一个同类型的数据对象的地址。已知 ival 的定义：

```
int ival = 1024;
```

下面的定义以及对两个指针 pi 和 pi2 的赋值都是合法的。

```
// pi 被初始化为 "没有指向任何对象"
```

[8] 这对函数指针并不成立，函数指针指向程序的代码段。函数指针和数据指针是不同的，函数指针将在 7.9 节说明。

```
int *pi = 0;

// pi2 被初始化为 ival 的地址
int *pi2 = &ival;

// ok: pi 和 pi2 现在都指向 ival
pi = pi2;

// 现在 pi2 没有指向任何对象
pi2 = 0;
```

指针不能持有非地址值。例如，下面的赋值将导致编译错误：

```
// 错误: pi 被赋以 int 值 ival
pi = ival;
```

指针不能被初始化或赋值为其他类型对象的地址值。例如，已知如下定义：

```
double dval;
double *pd = &dval;
```

那么，下列两条语句都会引起编译时刻错误：

```
// 都是编译时刻错误
// 无效的类型赋值: int* <== double*
pi = pd;
pi = &dval;
```

不是说 pi 在物理上不能持有与 dval 相关联内存的地址：它能够。但是不允许，因为，虽然 pi 和 pd 能够持有同样的地址值，但对那块内存的存储布局和内容的解释却完全不同。

当然，如果我们要做的仅仅是持有地址值（可能是把一个地址同另一个地址作比较），那么指针的实际类型就不重要了。C++提供了一种特殊的指针类型来支持这种需求：空（void*）类型指针，它可以被任何数据指针类型的地址值赋值（函数指针不能赋值给它）。

```
// ok: void* 可以持有任何指针类型的地址值
void *pv = pi;
pv = pd;
```

void*表明相关的值是个地址，但该地址的对象类型不知道。我们不能够操作空类型指针所指向的对象，只能传送该地址值或将它与其他地址值作比较。（在 4.14 节我们将会看到更多关于 void*类型的细节。）

已知一个 int 型指针对象 pi，当我们写下 pi 时：

```
// 计算包含在 pi 内部的地址值
// 类型: int*
pi;
```

这将计算 pi 当前持有的地址值。当我们写下&pi 时：

```
// 计算 pi 的实际地址
// 类型: int**
&pi;
```

这将计算指针对象 pi 被存储的位置的地址。那么，怎样访问 pi 指向的对象呢？

在缺省情况下，我们没有办法访问 pi 指向的对象，以对这个对象进行读或者写的操作。

为了访问指针所指向的对象，我们必须解除指针的引用。C++提供了解引用操作符（*）（dereference operator）来间接地读和写指针所指向的对象。例如，已知下列定义：

```
int ival = 1024, ival2 = 2048;
int *pi = &ival;
```

下面给出了怎样解引用 pi 以便间接访问 ival：

```
// 解除 pi 的引用，为它所指向的对象 ival
// 赋以 ival2 的值
*pi = ival2;

// 对于右边的实例，读取 pi 所指对象的值
// 对于左边的实例，则把右边的表达式赋给对象
*pi = abs( *pi ); // ival = abs(ival);
*pi = *pi + 1;    // ival = ival + 1;
```

我们知道，当取一个 int 型对象的地址时，

```
int *pi = &ival;
```

结果是 int*——即指向 int 的指针。当我们取指向 int 型的指针的地址时：

```
int **ppi = &pi;
```

结果是 int**——即指向 int 指针的指针。当我们解引用 ppi 时：

```
int *pi2 = *ppi;
```

我们获得指针 ppi 持有的地址值——在本例中，即 pi 持有的值，而 pi 又是 ival 的地址。为了实际地访问到 ival，我们需要两次解引用 ppi。例如：

```
cout << "The value of ival\n"
     << "direct value: " << ival << "\n"
     << "indirect value: " << *pi << "\n"
     << "doubly indirect value: " << **ppi
     << endl;
```

下面两条赋值语句的行为截然不同，但它们都是合法的。第一条语句增加了 pi 指向的数据对象的值，而第二条语句增加了 pi 包含的地址的值。

```
int i, j, k;
int *pi = &i;

// i 加 2 (i = i + 2)
*pi = *pi + 2;

// 加到 pi 包含的地址上
pi = pi + 2;
```

指针可以让它的地址值增加或减少一个整数值。这类指针操作，被称为指针的算术运算（pointer arithmetic）。这种操作初看上去并不直观，我们总认为是数据对象的加法，而不是离散的十进制数值的加法。指针加 2 意味着给指针持有的地址值增加了该类型两个对象的长度。例如，假设一个 char 是一个字节，一个 int 是 4 个字节，double 是 8 个字节，那么指针加 2 是给其持有的地址值增加 2、8、还是 16，完全取决于指针的类型是 char、int 还是 double。

实际上，只有指针指向数组元素时，我们才能保证较好地运用指针的算术运算。在前面的例子中，我们不能保证三个整数变量连续存储在内存中。因此，lp+2 可能，也可能不产生一个有效的地址，这取决于在该位置上实际存储的是什么。指针算术运算的典型用法是遍历一个数组。例如：

```
int ia[ 10 ];
int *iter = &ia[0];
int *iter_end = &ia[10];

while ( iter != iter_end ) {
    do_something_with_value( *iter );
    ++iter;    // 现在 iter 指向下一个元素
}
```

练习 3.8

已知下列定义

```
int ival = 1024, ival2 = 2048;
int *pi1 = &ival, *pi2 = &ival2, **pi3 = 0;
```

说明下列赋值将产生什么后果？哪些是错误的？

```
(a) ival = *pi3;   (e) pi1 = *pi3;
(b) *pi2 = *pi3;   (f) ival = *pi1;
(c) ival = pi2;    (g) pi1 = ival;
(d) pi2 = *pi1;    (h) pi3 = &pi2;
```

练习 3.9

指针是 C 和 C++程序设计一个很重要的方面，也是程序错误的常见起源。例如：

```
pi = &ival2;
pi = pi + 1024;
```

几乎可以保证，pi 会指向内存的一个随机区域。这个赋值在做什么？什么时候它不是一个错误？

练习 3.10

类似地，下面的小程序的行为是未定义的，可能在运行时失败：

```
int foobar( int *pi ) {
    *pi = 1024;
    return *pi;
}

int main() {
    int *pi2 = 0;
    int ival = foobar( pi2 );
    return 0;
}
```

问题出在哪里？怎样改正它？

练习 3. 11

在前面两个练习中，出现错误是因为缺少在运行时刻对指针使用的检查。如果指针在 C++程序设计中起重要作用，你认为为什么没有为指针的使用增加更多的安全性？你能想到哪些指导规则能使指针的使用更加安全？

3.4 字符串类型

C++提供了两种字符串的表示：C 风格的字符串和标准 C++引入的 string 类类型。一般我们建议使用 string 类，但实际上在许多程序的情形中，我们有必要理解和使用老式的 C 风格字符串。在第 7 章我们会看到一个例子。它处理命令行选项，而这些选项被作为 C 风格的字符串数组传递给 main()函数。

3.4.1 C 风格字符串

C 风格的字符串起源于 C 语言，并在 C++中继续得到支持。（实际上，在标准 C++之前，除了第三方字符串库类之外，它是惟一一种被支持的字符串。）

字符串被存储在一个字符数组中，一般通过一个 char*类型的指针来操纵它。标准 C 库为操纵 C 风格的字符串提供了一组函数。例如：

```
// 返回字符串的长度
int strlen( const char* );

// 比较两个字符串是否相等
int strcmp( const char*, const char* );

// 把第二个字符串拷贝到第一个字符串中
char* strcpy(char*, const char* );
```

（标准 C 库作为标准的 C++的一部分被包含在其中。）为使用这些函数。我们必须包含相关的 C 头文件，

```
#include <cstring>
```

指向 C 风格字符串的字符指针总是指向一个相关联的字符数组。即使当我们写一个字符串常量时，如：

```
const char *st = "The expense of spirit\n";
```

系统在内部也把字符串常量存储在一个字符串数组中。然后，st 指向该数组的第一个元素。那么，我们怎样以字符串的形式来操纵 st 呢？

一般地，我们用指针的算术运算来遍历 C 风格的字符串，每次指针增加 1，直到到达终止空字符为止。例如：

```
while ( *st++ ) { ... }
```

char*类型的指针被解除引用，并且测试指向的字符是 true 还是 false。true 值是除了空字

符外的任意字符。++是增加运算符，它使指针对指向数组中的下一个字符。

一般来说，当我们使用一个指针时，在解除指针的引用之前，测试它是否指向某个对象是必要的。否则，程序很可能会失败。例如：

```
int
string_length( const char *st )
{
int cnt = 0;

if ( st )
    while ( *st++ )
          ++cnt; return cnt;
}
```

C 风格字符串的长度可以为 0（因而被视为空串），有两种方式：字符指针被置为 0，因而它不指向任何对象。或者，指针已经被设置，但是它指向的数组只包含一个空字符。如：

```
// pc1 不指向任何一个数组对象
char *pc1 = 0;

// pc2 指向空字符
const char *pc2 = "";
```

由于 C 风格字符串的底层（low-level）特性，C 或 C++的初学者很容易在这上面出错。在下面的一系列程序中，我们罗列了一些初学者易犯的错误。程序的任务很简单：计算 st 的长度。不幸的是，第一个尝试就是错误的。你能看到问题所在吗？

```
#include <iostream>
const char *st = "The expense of spirit\n";
int main() {
int len = 0;

while ( st++ ) ++len;
        cout << len << "; " << st;
return 0;
}
```

程序失败是因为 st 没有被解除引用，即：

```
st++
```

测试的是 st 中的地址是否为零，而不是它指向的字符是否为空。这个条件将一直为真，因为循环的每次迭代都给 st 中的地址加 1。程序将永远执行下去或者由系统终止它。这样的循环被称作**无限循环**（infinite loop）。

我们的第二个版本改正了这个错误。它能执行到结束。不幸的是：输出的结果是错误的。你能发现我们这次犯的错误吗？

```
#include <iostream>
const char *st = "The expense of spirit\n";
int main()
{
   int len = 0;
   while ( *st++ ) ++len;
```

```
    cout << len << ": " << st << endl;

    return 0;
}
```

这次的错误是 st 已经不再指向字符串文字常量。st 已经前进到终止空字符之后的字符上去了。(程序的输出结果取决于 st 所指向的内存单元的内容。)下面是一种可能的解决办法:

```
st = st - len;
cout << len << ": " << st;
```

编译并执行程序。但是，输出仍然是不正确的。它产生如下结果:

```
22: he expense of spirit
```

这反映了程序设计某些本质的方面。你能看到这次我们犯的错误吗?

在计算字符串的长度的时候，空字符并没有被考虑在内。st 必须被重新定位到字符串长度加 1 的位置。下列代码是正确的:

```
st = st - len - 1;
```

编译并执行，程序最终产生正确的结果如下:

```
22: The expense of spirit
```

现在程序是正确的了，但是，从程序风格的角度来说，它还有些不太雅致。语句:

```
st = st - len - 1;
```

被加进来，以便改正由直接递增 st 引起的错误。st 的赋值不符合程序的原始逻辑，而且，现在的程序有些难以理解。

像这样的程序修正通常被称作补丁（patch）——把某些东西伸展开以便补上现有程序中的洞。我们通过补偿原始设计中的逻辑错误来补救我们的程序。较好的解决方案是修正原始设计中的漏洞。一种方案是定义第二个指针，用 st 对它初始化。例如:

```
const char *p = st;
```

现在可以用 p 来计算 st 的长度，而 st 不变:

```
while ( *p++ )
```

3.4.2 字符串类型

正如我们前面所看到的，因为字符指针的底层特性，用它表示字符串很容易出错。为了将程序员从许多“与使用 C 风格字符串相关的错误”中解脱出来，每个项目、部门或公司都提供了自己的字符串类——实际上，本书的前两个版本就是这样做的。问题是，如果每个人都提供自己的字符串实现，那么程序的可移植性和兼容性就变得非常困难。C++标准库提供了字符串类抽象的一个公共实现。

你希望字符串类有哪些操作呢？最小的基本行为集合出什么构成呢？

1．支持用字符序列或第二个字符串对象来初始化一个字符串对象。C 风格的字符串不支持用另外一个字符串初始化一个字符串。

2．支持字符串之间的拷贝。C 风格字符串通过使用库的数 strcpy()来实现。

3．支持读写访问单个字符。对于 C 风格字符串，单个字符访问由下标操作符或直接解除指针引用来实现。

4．支持两个字符串的相等比较。对于 C 风格字符串，字符串比较通过库函数 strcmp()来实现。

5．支持两个字符串的连接：把一个字符串接到另一个字符串上，或将两个字符串组合起来形成第三个字符串。对于 C 风格的字符串，连接由库函数 strcat()来实现。把两个字符串连接起来形成第三个字符串的实现是，用 strcpy()把一个字符串拷贝到一个新实例中，然后用 strcat()把另一个字符串连接到新的实例上。

6．支持对字符串长度的查询。对于 C 风格字符串，字符串长度由库函数 strlen()返回。

7．支持字符串是否为空的判断。对于 C 风格字符串，通过下面两步条件测试来完成。

```
char *str = 0;
// ...
if ( ! str || ! *str )
     return;
```

标准 C++提供了支持这些操作的 string 类（在第 6 章我们会看到更多的操作）。在本小节中，让我们来看 string 类型怎样支持这些操作。

要使用 string 类型，必须先包含相关的头文件：

```
#include <string>
```

例如，下面是上一小节定义的字符数组：

```
#include <string>

string st( "The expense of spirit\n" );
```

st 的长度由 size()操作返回（不包含终止空字符）：

```
cout << "The size of "
    << st
    << " is " << st.size()
    << " characters, including the newline\n";
```

string 构造函数的第二种形式定义了一个空字符串。例如：

```
string st2; // 空字符串
```

我们怎样能保证它是空的？当然，一种办法是测试 size()是否为 0：

```
if ( ! st.size() )
     // ok: 空
```

更直接的办法是使用 empty()操作：

```
if ( st.empty() )
     // ok: 空
```

如果字符串中不含有字符，则 empty()返回布尔常量 true；否则，返回 false。

第三种形式的构造函数，用一个 string 对象来初始化另一个 string 对象。例如：

```
string st3( st );
```

将 st3 初始化成 st 的一个拷贝。怎样验证呢？等于操作符比较两个 string 对象，如果相

等则返回 true：

```
if ( st == st3 )
    // 初始化成功
```

怎样拷贝一个字符串呢？最简单的办法是使用赋值操作符。例如：

```
st2 = st3;    // 把 st3 拷贝到 st2 中
```

首先将与 st2 相关联的字符存储区释放掉，然后再分配足够存储与 st3 相关联的字符的存储区，最后将与 st3 相关联的字符拷贝到该存储区中。

我们可以使用加操作符“+”或看起来有点怪异的复合赋值操作符“+=”，将两个或多个字符串连接起来。例如，给出两个字符串：

```
string s1( "hello, " );
string s2( "world\n" );
```

我们可以按如下方式将两个字符串连接起来形成第三个字符串：

```
string s3 = s1 + s2;
```

如果希望直接将 s2 附加在 s1 后面，那么可使用“+=”操作符：

```
s1 += s2;
```

s1 和 s2 的初始化包含了一个空格、一个逗号以及一个换行，这多少有些不方便。它们的存在限制了对这些 string 对象的重用，尽管它满足了眼前的需要。一种替代做法就是混合使用 C 风格的字符串与 string 对象，如下所示：

```
const char *pc = ", ";
string s1( "hello" );
string s2( "world" );

string s3 = s1 + pc + s2 + "\n";
```

这种连接策略比较受欢迎，因为它使 s1 和 s2 处于一种更容易被重用的形式。这种方法能够生效是由于 string 类型能够自动将 C 风格的字符串转换成 string 对象。例如，这使我们可以将一个 C 风格的字符串赋给一个 string 对象：

```
string s1;

const char *pc = "a character array";
s1 = pc; // ok
```

但是，反向的转换不能自动执行。对隐式地将 string 对象转换成 C 风格的字符串，string 类型没有提供支持。例如，下面试图用 s1 初始化 str，就会在编译时刻失败：

```
char *str = s1;    // 编译时刻类型错误
```

为实现这种转换，必须显式地调用名为 c_str()的操作：

```
char *str = s1.c_str();  // 几乎是正确的，但是还差一点
```

名字 c_str()代表了 string 类型与 C 风格字符串两种表示法之间的关系。字面意思是：给我一个 C 风格的字符串表示——即，指向字符数组起始处的字符指针。

但是，这个初始化还是失败了。这次是由于另外一个不同的原因：为了防止字符数组被

程序直接处理，c_str()返回了一个指向常量数组的指针（下一节将解释常量修饰符 const）

```
const char*
```

str 被定义为非常量指针，所以这个赋值被标记为类型违例。正确的初始化如下：

```
const char *str = s1.c_str();   // ok
```

string 类型支持通过下标操作符访问单个字符。例如，在下面的代码段中，字符串中的所有句号被下划线代替：

```
string str( "fa.disney.com" );
int size = str.size();

for ( int ix = 0; ix < size; ++ix )
     if ( str[ ix ] == '.' )
          str[ ix ] = '_';
```

对 string 类型的介绍现在就讲这些，尽管我们还有许多内容要说。例如，上面代码段的实现可用如下语句替代：

```
replace( str.begin(), str.end(), '.', '_' );
```

replace()是 2.8 节中简要介绍的泛型算法中的一个（第 12 章将详细介绍泛型算法，本书附录按字母顺序给出了泛型算法及其用法的例子）。

begin()和 end()操作返回指向 string 开始和结束处的迭代器(iterator)。迭代器是指针的类抽象，由标准库提供（在 2.8 节中我们简要地介绍了迭代器，在第 6 章和第 12 章将详细介绍）。

replace()扫描 begin()和 end()之间的字符。每个等于句号的字符，都被替换成下划线。

练习 3.12

下列语句哪些是错误的？

```
(a) char ch = "The long, winding road";
(b) int ival = &ch;
(c) char *pc = &ival;
(d) string st( &ch );
(e) pc = 0;    (i) pc = '0';
(f) st = pc;   (j) st = &ival;
(g) ch = pc[0];    (k) ch = *pc;
(h) pc = st;   (l) *pc = ival;
```

练习 3.13

解释下面两个 while 循环的区别。

```
while ( st++ )
      ++cnt;

while ( *st++ )
      ++cnt;
```

练习 3.14

考虑下面两个语义上等价的程序，一个使用 C 风格字符串，另一个使用 string 类型：

```
// ***** C-style character string implementation *****
#include <iostream>
#include <cstring>

int main()
{
    int errors = 0;
    const char *pc = "a very long literal string";
    for ( int ix = 0; ix < 1000000; ++ix )
    {
        int len = strlen( pc );
        char *pc2 = new char[ len + 1 ];
        strcpy( pc2, pc );
        if ( strcmp( pc2, pc ))
          ++errors;
        delete [] pc2;
    }
    cout << "C-style character strings: "
         << errors << " errors occurred.\n";
}

// ***** string implementation *****
#include <iostream>
#include <string>

int main() {
    int errors = 0;
    string str( "a very long literal string" );

    for ( int ix = 0; ix < 1000000; ++ix )
    {
        int len = str.size();
        string str2 = str;
        if ( str != str2 )
            ++errors;
    }

    cout << "string class: "
         << errors << " errors occurred.\n";
}
```

a) 说明程序完成了什么功能；

b) 平均来说，string 类型实现的执行速度是 C 风格字符串的两倍，在 UNIX 的 timex 命令下显示的执行时间如下：

```
user       0.96   # string class user
```

```
user       1.98   # C-style character string
```

你是这样预想的吗？说明原因。

练习 3.15

C++的 string 类型是基于对象的类抽象的一个例子。对于本节中所介绍的关于它的用法及操作集，你有什么希望改变的吗？你认为还有哪些其他操作是必需的？有用的？请说明。

3.5 const 限定修饰符

下面的循环有两个问题，都是由于使用 512 作为循环上限引起的：

```
for ( int index = 0; index < 512; ++index )
```

第一个问题是可读性。用 512 来测试 index 是什么意思呢？循环在做什么呢——即 512，是什么意思？［在本例中，512 被称作魔数（magic number），它的重要性在上下文中没有体现出来，就好像这个数是凭空出现的。］

第二个问题是可维护性。想像程序有 10000 行，512 在 4%的代码中出现。在这 400 个出现中，80%必须要被改成 1024。为了做到这一点，我们必须明白哪些 512 是要被转换的，而哪些不是。即使只有一个地方弄错了，也会中断程序，要我们回头全部重新检查一遍。

这两个问题的解决方案就是使用一个被初始化为 512 的对象。通过选择一个助记名，可能是 bufSize，使程序更具可读性。现在，条件测试变成与对象作比较，而不是与一个文字常量作比较：

```
index < bufSize
```

我们不需要再把 320 个出现 512 的地方——找出来，只需改变 bufSize 的值就行了。我们只需改变 bufSize 被初始化的那一行。这种方法不仅只需要很少的工作量，而且大大减少了出错的可能性。这种方案的代价是一个额外的变量。现在 512 被称为是局部化的（localized）。

```
int bufSize = 512;// 缓冲区大小

// ...
for ( int index = 0; index < bufSize; ++index )
     // ...
```

这种方案的问题是，bufSize 是一个左值。在程序中 bufSize 有可能被偶然修改。例如，下面是一个常见的程序错误：

```
// 偶然地改变了 bufSize 的值
if ( bufSize = 1 )
     // ...
```

在 C++中，“=”是赋值操作符，而“==”是等于操作符。程序员不小心将 bufSize 的值改成 1，将导致了一个很难跟踪的错误。（这种错误很难被发现，因为程序员一般不会认为这行代码是错的，这就是为什么许多编译器会对此类的赋值表达式生成警告的原因。）

const 类型限定修饰符提供了一个解决方案。它把一个对象转换成一个常量（constant）。

例如：

```
const int bufSize = 512   // 缓冲区大小
```

定义 bufSize 是一个常量，并将其初始化为 512。在程序中任何改变这个值的企图都将导致编译错误。因此，它被称为是只读的（read- only）。例如：

```
// 错误：企图写入 const 对象
if ( bufsize = 0 ) ...
```

因为常量在定义后就不能被修改，所以它必须被初始化。未初始化的常量定义将导致编译错误。

```
const double pi;   // 错误：未初始化的常量
```

一旦一个常量被定义了，我们就不能改变与 const 对象相关联的值。另一方面，我们能把它的地址赋值给一个指针吗？例如，下面代码是否可行？

```
const double minWage = 9.60;

// ok? error?
double *ptr = &minWage;
```

这是否可行呢？minWage 是一个常量对象，因此它不能被改写为一个新的值。但是 ptr 是一个普通指针，没有什么能阻止我们写出这样的代码：

```
*ptr += 1.40; // 修改了 minwage!
```

一般编译器不能跟踪指针在程序中任意一点指向的对象。[这种内部工作需要进行数据流分析（data flow analysis），通常由单独的优化器（optimizer）组件来完成。] 允许非 const 对象的指针指向一个常量对象，把“试图通过该指针间接地改变对象值”的动作标记为非法的，这对编译器来说是不可行的。因而任何“试图将一个非 const 对象的指针指向一个常量对象”的动作都将引起编译错误。

这并不意味着我们不能间接地指向一个 const 对象，只意味着我们必须声明一个指向常量的指针来做这件事。例如：

```
const double *cptr;
```

cptr 是一个指向 double 类型的 const 对象的指针。（我们可以从右往左把这个定义读为“cptr 是一个指向 double 类型的、被定义成 const 的对象的指针”。）此中微妙在于 cptr 本身不是常量。我们可以重新赋值 cptr，使其指向不同的对象，但不能修改 cptr 指向的对象。例如：

```
const double *pc = 0;
const double minWage = 9.60;

// ok: 不能通过 pc 修改 minWage
pc = &minWage;

double dval = 3.14;

// ok: 不能通过 pc 修改 dval
// 虽然 dval 本身不是一个常量
pc = &dval; // ok
```

```
dval = 3.14159;    // ok
*pc = 3.14159;     // 错误
```

const 对象的地址只能赋值给指向 const 对象的指针，例如 pc。但是，指向 const 对象的指针可以被赋以一个非 const 对象的地址，例如：

```
pc = &dval;
```

虽然 dval 不是常量，但试图通过 pc 修改它的值，仍会导致编译错误（因为在运行程序的任意一点上，编译器不能确定指针所指的实际对象）。

在实际的程序中，指向 const 的指针常被用作函数的形式参数。它作为一个约定来保证：被传递给函数的实际对象在函数中不会被修改。例如：

```
// 在实际的程序中，指向常量的指针
// 往往被用作函数参数
int strcmp( const char *str1, const char *str2 );
```

（在第 7 章关于函数的讨论中我们会更多地讨论指向 const 对象的指针。）

我们可以定义一个 const 指针指向一个 const 或一个非 const 对象。例如：

```
int errNumb = 0;
int *const curErr = &errNumb;
```

curErr 是指向一个非 const 对象的 const 指针。（我们可以从右往左把定义读作“curErr 是一个指向 int 类型对象的 const 指针”。）这意味着不能赋给 curErr 其他的地址值，但可以修改 curErr 指向的值。

下面的代码说明我们可以怎样使用 curErr：

```
do_something();

if ( *curErr ) {
    errorHandler();
    *curErr = 0;    // ok: 重置指针所指的对象
}
```

试图给 const 指针赋值会在编译时刻被标记为错误：

```
curErr = &myErrNumb; // 错误
```

指向 const 对象的 const 指针的定义就是将前面两种定义结合起来。例如：

```
const double pi = 3.14159;
const double *const pi_ptr = &pi;
```

在这种情况下，pi_ptr 指向的对象的值以及它的地址本身都不能被改变。（我们可以从右往左将定义读作“pi_ptr 是指向被定义为 const 的 double 类型对象的 const 指针”。）

练习 3.16

解释下列五个定义的意思，并指出其中非法的定义。

```
(a) int i;     (d) int *const cpi;
(b) const int ic;  (e) const int *const cpic;
(c) const int *pic;
```

练习 3.17

下列哪些初始化是合法的？为什么？

```
(a) int i = -1;
(b) const int ic = i;
(c) const int *pic = &ic;
(d) int *const cpi = &ic;
(e) const int *const cpic = &ic;
```

练习 3.18

根据上个练习的定义，下列哪些赋值是合法的？为什么？

```
(a) i = ic;    (d) pic = cpic;
(b) pic = &ic;     (e) cpic = &ic;
(c) cpi = pic;     (f) ic = *cpic;
```

3.6 引用类型

引用（reference）有时候又称为别名（alias），它可以用作对象的另一个名字。通过引用我们可以间接地操纵对象，使用方式类似于指针。但是不需要指针的语法。在实际的程序中，引用主要被用作函数的形式参数——通常将类对象传递给一个函数。但是现在我们用独立的对象来介绍并示范引用的用法。

引用类型由类型标识符和一个取地址操作符来定义，引用必须被初始化。例如：

```
int ival = 1024;

// ok: refVal 是一个指向 ival 的引用
int &refVal = ival;

// 错误: 引用必须被初始化为指向一个对象
int &refVal2;
```

虽然引用也可以被用作一种指针，但是像对指针那样用一个对象的地址初始化引用，却是错误的。然而，我们可以定义一个指针引用，例如：

```
int ival = 1024;

// 错误: refVal 是 int 类型，不是 int*
int &refVal = &ival;

int *pi = &ival;

// ok: refPtr 是一个指向指针的引用
int *&ptrVal2 = pi;
```

一旦引用已经定义，它就不能再指向其他的对象（这是它为什么必须要被初始化的原因）。例如，下列的赋值不会使 refVal 指向 min_val，而是会使 refVal 指向的对象 ival 的值被

设置为 min_val 的值。

```
int min_val = 0;

// ival 被设置为 min_val 的值
// refVal 并没有引用到 min_val 上
refVal = min_val;
```

引用的所有操作实际上都被应用在它所指的对象身上，包括取地址操作符。例如：

```
refVal += 2;
```

将 refVal 指向的对象 ival 加 2。类似地，如下语句：

```
int ii = refVal;
```

把与 ival 相关联的值赋给 ii，而：

```
int *pi = &refVal;
```

用 ival 的地址初始化 pi。

每个引用的定义必须以取地址操作符开始。（这与前面我们对指针的讨论是同样的问题。）例如：

```
// 定义两个 int 类型的对象
int ival = 1024, ival2 = 2048;

// 定义一个引用和一个对象
int &rval = ival, rval2 = ival2

// 定义一个对象、一个指针和一个引用
int ival3 = 1024, *pi = &ival3, &ri = ival3;

// 定义两个引用
int &rval3 = ival3, &rval4 = ival2;
```

const 引用可以用不同类型的对象初始化（只要能从一种类型转换到另一种类型即可），也可以是不可寻址的值，如文字常量。例如：

```
double dval = 3.14159;

// 仅对于 const 引用才是合法的
const int &ir = 1024;
const int &ir2 = dval;
const double &dr = dval + 1.0;
```

同样的初始化对于非 const 引用是不合法的，将导致编译错误。原因有些微妙，需要适当作些解释。

引用在内部存放的是一个对象的地址，它是该对象的别名。对于不可寻址的值，如文字常量，以及不同类型的对象，编译器为了实现引用，必须生成一个临时对象，引用实际上指向该对象，但用户不能访问它。例如，当我们写：

```
double dval = 1024;
const int &ri = dval;
```

编译器将其转换成：

```
int temp = dval;
const int &ri = temp;
```

如果我们给 ri 赋一个新值，则这样做不会改变 dval，而是改变 temp。对用户来说，就好像修改动作没有生效（这对于用户来说，这并不总是好事情）。

const 引用不会暴露这个问题，因为它们是只读的。不允许非 const 引用指向需要临时对象的对象或值，一般来说，这比“允许定义这样的引用，但实际上不会生效”的方案要好得多。

下面给出的例子很难在第一次就能正确声明。我们希望用一个 const 对象的地址来初始化一个引用。非 const 引用定义是非法的，将导致编译时刻错误：

```
const int ival = 1024;

// 错误：要求一个 const 引用
int *&pi_ref = &ival;
```

下面是我们在打算修正 pi_ref 定义时首先想到的做法，但是它不能生效——你能看出来这是为什么吗？

```
const int ival = 1024;

// 仍然错误
const int *&pi_ref = &ival;
```

如果我们从右向左读这个定义，会发现 pi_ref 是一个指向定义为 const 的 int 型对象的指针。我们的引用不是指向一个常量，而是指向一个非常量指针，指针指向一个 const 对象。正确的定义如下：

```
const int ival = 1024;

// ok：这是可以被编译器接受的
const int *const &pi_ref = &ival;
```

指针和引用有两个主要区别：引用必须总是指向一个对象。如果用一个引用给另一个引用赋值，那么改变的是被引用的对象而不是引用本身。让我们来看几个例子。当我们这样写：

```
int *pi = 0;
```

用 0 初始化 pi——即，pi 当前不指向任何对象。但当我们写如下语句时：

```
const int &ri = 0;
```

在内部，发生了以下转换：

```
int temp = 0;
const int &ri = temp;
```

引用之间的赋值是第二个不同。当给定了以下代码后：

```
int ival = 1024, ival2 = 2048;
int *pi = &ival, *pi2 = &ival2;
```

我们再写如下语句：

```
pi = pi2;
```

pi 指向的对象 ival 并没有被改变。实际上 pi 被赋值为指向 pi2 所指的对象——在本例中即 ival2。重要的是，现在 pi 和 pi2 都指向同一对象。（这是一个重要的错误源：如果我们把一个类对象拷贝给另一个类对象，而该类有一个或多个成员是指针。我们将在第 14 章详细讨论这个问题。）

但是，假定有下列代码：

```
int &ri = ival, &ri2 = ival2;
```

当我们写出这样的赋值语句时：

```
ri = ri2;
```

改变的是 ival，而不是引用本身。赋值之后，两个引用仍然指向原来的对象。

实际的 C++程序很少使用指向独立对象的引用类型。引用类型主要被用作函数的形式参数，例如：

```
// 在实际的例子中，引用是如何被使用的
// 返回访问状态，将值放入参数
bool get_next_value( int &next_value );

// 重载加法操作符
Matrix operator+( const Matrix&, const Matrix& );
```

这些引用的用法和我们讨论的指向独立对象的引用类型有什么联系呢？在下面这样的调用中：

```
int ival;
while ( get_next_value( ival )) ...
```

实际参数（本例中为 ival）同形式参数 next_value 的绑定，等价于下面的独立对象定义：

```
int &next_value = ival;
```

（引用作为函数参数的用法将在第 7 章中详细讨论。）

练习 3.19

下列定义中，哪些是无效的？为什么？怎样改正？

```
(a) int ival = 1.01;   (b) int &rval1 = 1.01;
(c) int &rval2 = ival; (d) int &rval3 = &ival;
(e) int *pi = &ival;   (f) int &rval4 = pi;
(g) int &rval5 = *pi;  (h) int &*prval1 = pi;
(i) const int &ival2 = 1;    (j) const int &*prval2 = &ival;
```

练习 3.20

已知下面的定义，下列赋值哪些是无效的？

```
(a) rval1 = 3.14159;
(b) prval1 = prval2;
(c) prval2 = rval1;
(d) *prval2 = ival2;
```

练习 3.21

(a)中的定义有什么区别？(b)中的赋值又有什么区别？哪些是非法的？

```
(a) int ival = 0;
    const int *pi = 0;
    const int &ri = 0;
(b) pi = &ival;
    ri = &ival;
    pi = &rval;
```

3.7 布尔类型

布尔型对象可以被赋以文字值 true 或 false。例如：

```
// 初始化一个 string 对象，用来存放搜索的结果
string search_word = get_word();
// 把一个 bool 变量初始化为 false
bool found = false;
string next_word;
while ( cin >> next_word )
    if ( next_word == search_word )
         found = true;
// ...

// 缩写，相当于：if ( found == true )
if ( found )
     cout << "ok, we found the word\n";
else cout << "nope, the word was not present.\n";
```

虽然布尔类型的对象也被看作是一种整数类型的对象，但是它不能被声明为 signed、unsigned、short 或 long。例如，下列代码是非法的：

```
// 错误：不能指定 bool 为 short
short bool found = false;
```

当表达式需要一个算术值时，布尔对象(如 found)和布尔文字都被隐式地提升成 int(正如下面的例子)：false 变成 0，而 true 变成 1。例如：

```
bool found = false;
int occurrence_count = 0;

while ( /* 条件省略 */ )
{
   found = look_for( /* 内容省略 */ );

// found 的值被提升为 0 或者 1
   occurrence_count += found;
}
```

正如文字 false 和 true 能自动转换成整数值 0 和 1 一样，如果有必要，算术值和指针值也

能隐式地被转换成布尔类型的值。0 或空指针被转换成 false，所有其他的值都被转换成 true。例如：

```
// 返回出现次数
extern int find( const string& );
bool found = false;
if ( found = find( "rosebud" ))
    // ok: found == true


// 如找到返回该项的指针
extern int* find( int value );
if ( found = find( 1024 ))
    // ok: found == true
```

3.8 枚举类型

我们在写程序的时候，常常需要定义一组与对象相关的属性。例如，一个文件可能会以三种状态（输入、输出和追加）之一被打开。

典型情况下，我们通过把每个属性和一个唯一的 const 值相关联，来记录这些状态值。因此，我们可能会这样写：

```
const int input = 1;
const int output = 2;
const int append = 3;
```

并按如下方式使用这些常量：

```
bool open_file( string file_name, int open_mode);

// ...
open_file( "Phoenix_and_the_Crane", append );
```

尽管这样做也能奏效，但是它有许多缺点。一个主要的缺点是，我们没有办法限制传递给函数的值只能是 input、output 和 append 之一。

枚举（enumeration）提供了一种替代的方法，它不但定义了整数常量，而且还把它们组成一个集合。例如：

```
enum open_modes{ input = 1, output, append };
```

open_modes 是一个枚举类型。每个被命名的枚举定义了一个唯一的类型，它可以被用作类型标识符，例如：

```
void open_file( string file_name, open_modes om );
```

input、output 和 append 是**枚举成员**（enumerator）。它们代表了能用来初始化和赋值 open_modes 类型变量的值的全集。例如：

```
open_file( "Phoenix and the Crane", append );
```

如果我们试图向 open_file()传递一个 input、output、append 之外的值，就会产生编译错误。而且，如果像下面这样传递一个相等的整数值，编译器仍然会将其标记为错误。

```
// 错误：1 不是 open_modes 的枚举成员 ...
open_file( "Jonah", 1 );
```

此外，我们还可以声明枚举类型对象，如：

```
open_modes om = input;

// ...
om = append;
```

并用 om 代替一个枚举成员：

```
open_file( "TailTell", om );
```

我们不能做到的是打印枚举成员的实际枚举名。当我们这样写的时候：

```
cout << input << " " << om << endl;
```

输出为：

```
13
```

一种解决方案是定义一个由枚举成员的值索引的字符串数组。因此，我们可以这样写：

```
cout << open_modes_table[ input ] << " "
     << open_modes_table[ om ]    << endl;
```

产生输出：

```
input append
```

第二件不能做的事情是，我们不能使用枚举成员进行迭代，如：

```
// 不支持
for ( open_modes iter = input; iter != append; ++iter )
      // ...
```

C++不支持在枚举成员之间的前后移动。

枚举类型用关键字 enum，加上一个自选的枚举类型名来定义，类型名后面跟一个用花括号括起来的枚举成员列表，枚举成员之间用逗号分开。 在缺省情况下，第一个枚举成员被赋以值 0，后面的每个枚举成员依次比前面的大 1。在前面的例子中，赋给 input 值 1，output 值 2，append 值 3。下面的枚举成员 shape 与 0 相关，sphere 是 1，cylinder 是 2，polygon 是 3。

```
// shape == 0, sphere == 1, cylinder == 2, polygon == 3
enum Forms{ shape, sphere, cylinder, polygon };
```

我们也可以显式地把一个值赋给一个枚举成员。这个值不必是唯一的。下面的例子中，point2d 被赋值为 2，在缺省情况下，point2w 等于 point2d 加 1 为 3，point3d 被显式地赋值为 3，point3w 在缺省情况下是 4。

```
// point2d == 2, point2w == 3, point3d == 3, point3w == 4
enum Points { point2d = 2, point2w, point3d = 3, point3w };
```

我们可以定义枚举类型的对象，它可以参与表达式运算，也叫以被作为参数传递给函数。枚举类型的对象能够被初始化，但是只能被一个相同枚举类型的对象或枚举成员集中的某个值初始化或赋值。例如，虽然 3 是一个与 Points 相关联的合法值，但是它不能被显式地赋给 Points 类型的对象：

```
void mumble() {
    Points pt3d = point3d; // ok: pt3d == 3

    // 错误: pt2w 被初始化为一个 int 整数
    Points pt2w = 3;

    // 错误: polygon 不是 Points 的枚举成员
    pt2w = polygon;

    // ok: pt2w 和 pt3d 都是 Points 枚举类型
    pt2w = pt3d;
}
```

但是，在必要时，枚举类型会自动被提升成算术类型。例如：

```
const int array_size = 1024;

// ok: pt2w 被提升成 int 类型
int chunk_size = array_size * pt2w;
```

3.9 数组类型

正如我们在 2.1 节中所看到的，数组是一个单一数据类型对象的集合。其中单个对象并没有被命名，但是我们可以通过它在数组中的位置对它进行访问。这种访问形式被称作**索引访问**（indexing）或**下标访问**（subscripting）。例如：

```
int ival;
```

声明了一个 int 型对象。而如下形式：

```
int ia[10];
```

声明了一个包含 10 个 int 对象的数组。每个对象被称作是 ia 的一个元素。因此：

```
ival = ia[ 2 ];
```

将 ia 中由 2 索引的元素的值赋给 ival。类似地：

```
ia[ 7 ] = ival;
```

把 ival 的值赋给 ia 的由 7 索引的元素。

数组定义由类型名、标识符和维数组成。维数指定数组中包含的元素的数目，它被写在一对方括号里边。我们必须为数组指定一个大于等于 1 的维数。维数值必须是常量表达式——即，必须能在编译时刻计算出它的值。这意味着非 const 的变量不能被用来指定数组的维数。下面的例子包含合法的和非法的数组定义：

```
extern int get_size();
// buf_size 和 max_files 都是 const
const int buf_size = 512, max_files = 20;
int staff_size = 27;

// ok: const 变量
```

```
char input_buffer[ buf_size ];

// ok: 常量表达式: 20 - 3
char *fileTable[ max_files - 3 ];

// 错误: 非 const 变量
double salaries[ staff_size ];

// 错误: 非 const 表达式
int test_scores[ get_size() ];
```

虽然 staff_size 被一个文字常量初始化，但是 staff_size 本身是一个非 const 对象，系统只能在运行时刻访问它的值。因此，它作为数组维数是非法的。另一方面，表达式：

```
max_files - 3
```

是常量表达式，因为 max_files 是用 20 作初始值的 const 变量，所以这个表达式在编译时刻被计算成 17。

正如我们在 2.1 节所看到的，数组元素是从 0 开始计数的。对一个包含 10 个元素的数组，正确的索引值是从 0 到 9，而不是从 1 到 10。在下面的例子中，for 循环遍历数组的 10 个元素，并用它们的索引值作初始值：

```
int main()
{
    const int array_size = 10;
    int ia[ array_size ];

    for ( int ix = 0; ix < array_size; ++ix )
        ia[ ix ] = ix;
}
```

数组可以被显式地用一组数来初始化，这组数用逗号分开，放在大括号中。例如：

```
const int array_size = 3;
int ia[ array_size ] = { 0, 1, 2 };
```

被显式初始化的数组不需要指定维数值。编译器会根据列出来的元素的个数来确定数组的维数：

```
// 维数为 3 的数组
int ia[] = { 0, 1, 2 };
```

如果指定了维数，那么初始化列表提供的元素的个数不能超过这个值。否则，将导致编译错误。如果指定的维数大于给出的元素的个数，那么没有被显式初始化的元素将被置为 0。

```
// ia ==> { 0, 1, 2, 0, 0 }
const int array_size = 5;
int ia[ array_size ] = { 0, 1, 2 };
```

字符数组可以用一个由逗号分开的字符文字列表初始化，文字列表用花括号括起来，或者用一个字符串文字初始化。但是，注意这两种形式不是等价的，字符串常量包含一个额外的终止空字符。例如：

```
const char ca1[] = { 'C', '+', '+' };
const char ca2[] = "C++";
```

cal 的维数是 3，ca2 的维数是 4。下面的声明将被标记为错误：

```
// 错误："Daniel"是 7 个元素
const char ch3[ 6 ] = "Daniel";
```

一个数组不能被另外一个数组初始化，也不能被赋值给另外一个数组。而且，C++不允许声明一个引用数组(即由引用组成的数组)。

```
const int array_size = 3;
int ix, jx, kx;

// ok：类型为 int*的指针的数组
int *iap [] = { &ix, &jx, &kx };

// 错误：不允许引用数组
int &iar[] = { ix, jx, kx };

// 错误：不能用另一个数组来初始化一个数组
int ia2[] = ia; // 错误
int main()
{
    int ia3[ array_size ]; // ok
    // 错误：不能把一个数组赋给另一个数组
    ia3 = ia;
    return 0;
}
```

要把一个数组拷贝到另一个中去，必须按顺序拷贝每个元素。例如：

```
const int array_size = 7;
int ia1[] = { 0, 1, 2, 3, 4, 5, 6 };

int main()
{
    int ia2[ array_size ];
    for ( int ix = 0; ix < array_size; ++ix )
        ia2[ ix ] = ia1[ ix ];

    return 0;
}
```

任意结果为整数值的表达式都可以用来索引数组。例如：

```
int someVal, get_index();
ia2[ get_index() ] = someVal;
```

但是用户必须清楚，C++没有提供编译时刻或运行时刻对数组下标的范围检查。除了程序员自己注意细节，并彻底地测试自己的程序之外，没有别的办法可防止数组越界。能够通过编译并执行的程序仍然存在致命的错误，这不是不可能的。

练习 3.22

下面哪些数组定义是非法的？为什么？

```
int get_size();
int buf_size = 1024;

(a) int ia[ buf_size ];  (d) int ia[ 2 * 7 - 14 ];
(b) int ia[ get_size() ];   (e) char st[ 11 ] = "fundamental";
(c) int ia[ 4 * 7 - 14 ];
```

练习 3.23

下面代码试图用数组中每个元素的索引值来初始化该元素。其中包含了一些索引错误，请把它们指出来：

```
int main() {
const int array_size = 10;

int ia[ array_size ];
for ( int ix = 1; ix <= array_size; ++ix )
    ia[ ix ] = ix;

// ...
}
```

3.9.1 多维数组

我们也可以定义多维数组。每一维用一个方括号对来指定，例如：

```
int ia[ 4 ] [ 3 ];
```

定义了一个二维数组。第一维被称作行（row）维，第二维称作列（column）维。ia 是一个二维数组，它有 4 行，每行 3 个元素。多维数组也可以被初始化：

```
int ia[ 4 ][ 3 ] = {
    { 0, 1, 2 },
    { 3, 4, 5 },
    { 6, 7, 8 },
    { 9, 10, 11 }
}
```

用来表示行的花括号，即被内嵌在里边的花括号是可选的。下面的初始化与前面的是等价的，只是有点不清楚：

```
int ia[4][3] = { 0,1,2,3,4,5,6,7,8,9,10,11 };
```

下面的定义只初始化了每行的第一个元素。其余的元素被初始化为 0：

```
int ia[ 4 ][ 3 ] = { {0}, {3}, {6}, {9} };
```

如果省略了花括号，结果会完全不同。下面的定义：

```
int ia[ 4 ][ 3 ] = { 0, 3, 6, 9 };
```

初始化了第一行的 3 个元素和第二行的第一个元素，其余元素都被初始化为 0。为了索引到一个多维数组中，每一维都需要一个方括号对。例如，下面的一对嵌套 for 循环初始化了一个二维数组：

```
int main()
```

```
{
    const int rowSize = 4;
    const int colSize = 3;
    int ia[ rowSize ][ colSize ];

    for ( int i = 0; i < rowSize; ++i )
        for ( int j = 0; j < colSize; ++j )
            ia[ i ][ j ] = i + j;
}
```

虽然表达式：

```
ia[ 1, 2 ]
```

在 C++中是合法的结构，但它的意思可能不是程序员所希望的：ia[1,2]等价于 ia[2]，因为“1,2”是一个逗号表达式，它的结果是一个单值 2（逗号表达式将在 4.10 节中讨论）。这将访问 ia 的第三行的第一个元素，而程序员希望的可能是 ia[1][2]。

在 C++中，多维数组的索引访问要求对程序员希望访问的每个索引都有一对方括号。

3.9.2 数组与指针类型的关系

已知下面的数组定义

```
int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21 };
```

那么，只简单写

```
ia;
```

意味着什么呢？

数组标识符代表数组中第一个元素的地址，它的类型是数组元素类型的指针。在 ia 这个例子中，它的类型是 int*。因此，下面两种形式是等价的，它们都返回数组的第一个元素的地址：

```
ia;
&ia[0];
```

类似地，为了访问相应的值，我们可以取下列两种方式之一：

```
// 两者都得到第一个元素的值
*ia;
ia[0];
```

我们知道怎样用下标操作符来访问第二个元素的地址：

```
&ia[1];
```

同样，下面这个表达式：

```
ia+1;
```

也能得到第二个元素的地址等等。类似地，下面两个表达式都可以访问第二个元素的值：

```
*(ia+1);
ia[1];
```

但是，如下的表达式：

```
*ia + 1;
```

与下面的表达式完全不同：

```
*(ia + 1);
```

解引用操作符比加法运算符的优先级高（我们将在 4.13 节中讨论优先级），所以它先被计算。解引用 ia 将返回数组的第一个元素的值。然后对其加 1。如果在表达式里加上括号，那么 ia 将先被加 1，然后解引用新的地址值。对 ia 加 1 将使 ia 增加其元素类型的大小，ia+1 指向数组中的下一个元素。

数组元素遍历则可以通过下标操作符来实现，到目前为止我们一直这样做，或者我们也可以通过直接操作指针来实现数组元素遍历。例如：

```
#include <iostream>
int main()
{
    int ia[9] = { 0, 1, 1, 2, 3, 5, 8, 13, 21 };
    int *pbegin = ia;
    int *pend = ia + 9;

    while ( pbegin != pend ) {
          cout << *pbegin << ' ';
          ++pbegin;
    }
}
```

pbegin 被初始化指向数组的第一个元素。在 while 循环的每次迭代中它都被递增以指向数组的下一个元素，最难的是判断何时停止。在本例中，我们将 pend 初始化指向数组最末元素的下一个地址。当 pbegin 等于 pend 时，表示已经迭代了整个数组。

如果我们把这一对指向数组头和最末元素下一位置的指针，抽取到一个独立的函数中，那么，就有了一个能够迭代整个数组的工具，却无须知道数组的实际大小（当然，调用函数的程序员必须知道）。例如：

```
#include <iostream>

void ia_print( int *pbegin, int *pend )
{
    while ( pbegin != pend ) {
       cout << *pbegin << ' ';
       ++pbegin;
    }
}

int main()
{
    int ia[9] = { 0, 1, 1, 2, 3, 5, 8, 13, 21 };
    ia_print( ia, ia + 9 );
}
```

当然，这是有限制的：它只支持指向整型数组的指针。我们可以通过把 ia_print()转换成模板函数来消除这个限制（在 2.5 节我们已经简要地介绍了模板）。例如：

```
#include <iostream>

template <class elemType>
void print( elemType *pbegin, elemType *pend )
{
    while ( pbegin != pend ) {
        cout << *pbegin << ' ';
        ++pbegin;
    }
}
```

现在我们可以给通用的函数 print()传递一对指向任意类型数组的指针，只要该类型的输出操作符已经被定义即可，例如：

```
int main()
{
    int ia[9] = { 0, 1, 1, 2, 3, 5, 8, 13, 21 };
    double da[4] = { 3.14, 6.28, 12.56, 25.12 };
    string sa[3] = { "piglet", "eeyore", "pooh" };
    print( ia, ia+9 );
    print( da, da+4 );
    print( sa, sa+3 );
}
```

这种程序设计形式被称为**泛型程序设计**（generic programming），标准库提供了一组泛型算法（我们在 2.8 节和 3.4 节结束的时候简要地介绍了这些算法），它们通过一对标记元素范围的开始/结束指针来遍历其中的元素。例如，我们可以如下调用泛型算法 sort()：

```
#include <algorithm>

int main()
{
    int ia[6] = { 107, 28, 3, 47, 104, 76 };
    string sa[3] = { "piglet", "eeyore", "pooh" };
    sort( ia, ia+6 );
    sort( sa, sa+3 );
}
```

我们将在第 12 章详细讨论泛型算法。本书附录以字母顺序给出这些算法以及用法示例。

更一般化的是，标准库提供了一组类，它们封装了容器和指针的抽象。在 2.8 节我们已经对其进行了简要的介绍。在下一节中，我们将讨论 vector 容器类型，它为内置数组提供了一个基于对象的替代品。

3.10 vector 容器类型

vector 类为内置数组提供了一种替代表示（在 2.8 节中我们简要介绍了（vector），通常我们建议使用 vector。（但是仍然有许多程序环境必须使用内置数组，例如处理命令行选项——我们将在 7.8 节中可以看到。）与 string 类一样，vector 类是随标准 C++引入的标准库的一部分。

为了使用 vector，我们必须包含相关的头文件：

```
#include <vector>
```

使用 vector 有两种不同的形式，即所谓的数组习惯和 STL 习惯。在数组习惯用法中，我们模仿内置数组的用法：定义一个已知长度的 vector：

```
vector< int > ivec( 10 );
```

这与如下定义一个包含十个元素的内置数组相似：

```
int ia[ 10 ];
```

我们可以用下标操作符访问 vector 的元素，与访问内置数组的元素的方式一样。例如：

```
void simple_example()
{
    const int elem_size = 10;
    vector< int > ivec( elem_size );

    int ia[ elem_size ];

    for ( int ix = 0; ix < elem_size; ++ix )
         ia[ ix ] = ivec[ ix ];
    // ...
}
```

我们可以用 size()查询 vector 的大小，也可以用 empty()测试它是否为空。例如：

```
void print_vector( vector<int> ivec )
{
    if ( ivec.empty() )
        return;

    for ( int ix = 0; ix < ivec.size(); ++ix )
        cout << ivec[ ix ] << ' ';
}
```

vector 的元素被初始化为与其类型相关的缺省值，算术和指针类型的缺省值是 0，对于 class 类型，缺省值可通过调用这类的缺省构造函数获得（关于缺省构造函数的介绍见 2.3 节）。我们还可以为每个元素提供一个显式的初始值来完成初始化。例如：

vector< int > ivec(10, -1);

定义了 ivec，它包含十个 int 型的元素，每个元素都被初始化为-1。

对于内置数组，我们可以显式地把数组的元素初始化为一组常量值。例如：

```
int ia[ 6 ] = { -2, -1, 0, 1, 2, 1024 };
```

我们不能用同样的方法显式地初始化 vector。但是，可以将 vector 初始化为一个已有数组的全部或一部分，只需指定希望被用来初始化 vector 的数组的开始地址以及数组最末元素的下一位置来实现。例如：

```
// 把 ia 的 6 个元素拷贝到 ivec 中
vector< int > ivec( ia, ia+6 );
```

被传递给 ivec 的两个指针标记了用来初始化对象的值的范围。第二个指针总是指向要被拷贝的末元素的下一位置，标记出来的元素范围也可以是数组的一个子集。例如：

```
// 拷贝 3 个元素: ia[2], ia[3], ia[4]
vector< int > ivec( &ia[ 2 ], &ia[ 5 ] );
```

与内置数组不同，vector 可以被另一个 vector 初始化，或被赋给另一个 vector。例如：

```
vector< string > svec;

void init_and_assign()
{
    // 用另一个 vector 初始化一个 vector
    vector< string > user_names( svec );
    // ...

    // 把一个 vector 拷贝给另一个 vector
    svec = user_names;
}
```

在 STL[9]中对 vector 的习惯用法完全不同。我们不是定义一个已知大小的 vector，而是定义一个空 vector：

```
vector< string > text;
```

我们向 vector 中插入元素，而不再是索引元素，以及向元素赋值。例如，push_back()操作，就是在 vector 的后面插入一个元素。下面的 while 循环从标准输入读入一个字符串序列，并每次将一个字符串插入到 vector 中：

```
string word;
while ( cin >> word ) {
     text.push_back( word );
     // ...
}
```

虽然我们仍可以用下标操作符来迭代访问元素：

```
cout << "words read are: \n";

for ( int ix = 0; ix < text.size(); ++ix )
     cout << text[ ix ] << ' ';

cout << endl;
```

但是，更典型的做法是使用 vector 操作集中的 begin()和 end()所返回的**迭代器**（iterator）对：

```
cout << "words read are: \n";

for ( vector<string>::iterator it = text.begin();
   it != text.end(); ++it )
         cout << *it << ' ';

cout << endl
```

iterator 是标准库中的类，它具有指针的功能。

```
*it;
```

对迭代器解引用，并访问其指向的实际对象。

```
++it;
```

[9] STL 表示标准模板库（Standard Template Library）。在被纳入到标准 C++中之前，vector 与泛型算法是独立库 STL 的一部分（见 [MUSSER96]）。

向前移动迭代器 it，使其指向下一个元素（在第 6 章，我们将非常详细地讨论 iterator、vector 和一般的 STL 习惯用法）

注意，不要混用这两种习惯用法。例如，下面的定义：

```
vector< int > ivec;
```

定义了一个空 vector，再写这样的语句：

```
ivec[ 0 ] = 1024;
```

就是错误的，因为 ivec 还没有第一个元素。我们只能索引 vector 中已经存在的元素。size()操作返回 vector 包含的元素的个数。

类似地，当我们用一个给定的大小定义一个 vector 时，例如：

```
vector<int> ia( 10 );
```

任何一个插入操作都将增加 vector 的大小，而不是覆盖掉某个现有的元素。这看起来好像是很显然的，但是，下面的错误在初学者中并不少见：

```
const int size = 7;
int ia[ size ] = { 0, 1, 1, 2, 3, 5, 8 };
vector< int > ivec( size );

for ( int ix = 0; ix < size; ++ix )
   ivec.push_back( ia[ ix ]);
```

程序结束时 ivec 包含 14 个元素，ia 的元素从第八个元素开始插入。

另外，在 STL 习惯用法下，vector 的一个或多个元素可以被删除。（我们将在第 6 章讨论。）

练习 3.24

下列 vector 定义中，哪些是错误的？

```
int ia[ 7 ] = { 0, 1, 1, 2, 3, 5, 8 };
(a) vector< vector< int > > ivec;
(b) vector< int > ivec = { 0, 1, 1, 2, 3, 5, 8 };
(c) vector< int > ivec( ia, ia+7 );
(d) vector< string > svec = ivec;
(e) vector< string > svec( 10, string( "null" ));
```

练习 3.25

已知下面的函数声明：

```
bool is_equal( const int*ia, int ia_size,
               const vector<int> &ivec );
```

请实现下列行为：如果两个容器大小不同，则比较相同大小部分的元素。一旦某个元素不相等，则返回 false。如果所有元素都相等，则返回 true。请用 iterator 迭代访问 vector，可以以本节中的例子为模型，并且写一个 main()函数来测试 is_equal()函数。

3.11 复数类型

复数（complex number）类是标准库的一部分。为了能够使用它，我们必须包含其相关的头文件：

```
#include <complex>
```

每个复数都有两部分：实数部分和虚数部分。虚数代表负数的平方根，这个术语是由笛卡儿首创的。复数的一般表示法如下：

```
2 + 3i
```

这里 2 代表实数部分，而 3i 表示虚数部分。这两部分合起来表示单个复数。

复数对象的定义一般可以使用以下形式：

```
// 纯虚数: 0 + 7i
complex< double > purei( 0, 7 );

// 虚数部分缺省为 0: 3 + 0i
complex< float > real_num( 3 );

// 实部和虚部均缺省为 0: 0 + 0i
complex< long double > zero;

// 用另一个复数对象来初始化一个复数对象
complex< double > purei2( purei );
```

这里，复数对象有 float、double 或 long double 几种表示。我们也可以声明复数对象的数组：

```
complex< double > conjugate[ 2 ] = {
          complex< double >( 2, 3 ),
          complex< double >( 2, -3 )
};
```

我们也可以声明指针或引用：

```
complex< double > *ptr = &conjugate[0];
complex< double > &ref = *ptr;
```

复数支持加、减、乘、除和相等比较。另外，它也支持对实部和虚部的访问。这些操作将在 4.6 节中详细介绍。

3.12 typedef 名字

typedef 机制为我们提供了一种通用的类型定义设施，可以用来为内置的或用户定义的数据类型引入助记符号。例如：

```
typedef double wages;
typedef vector<int>  vec_int;
typedef vec_int      test_scores;
typedef bool         in_attendance;
```

```
typedef int *Pint;
```

这些 typedef 名字在程序中可被用作类型标识符：

```
// double hourly, weekly;
wages hourly, weekly;

// vector<int> vec1( 10 );
vec_int vec1( 10 );

// vector<int> test0( class_size );
const int class_size = 34;
test_scores test0( class_size );

// vector< bool > attendance;
vector< in_attendance > attendance( class_size );

// int *table[ 10 ];
Pint table[ 10 ];
```

typedef 定义以关键字 typedef 开始，后面是数据类型和标识符。这里的标识符即 typedef 名字，它并没有引入一种新的类型，而只是为现有类型引入了一个助记符号。typedef 名字对以出现在任何类型名能够出现的地方。

typedef 名字可以被用作程序文档的辅助说明，它也能够降低声明的复杂度。例如，在典型情况下，typedef 名字可以用来增强“复杂模板声明的定义”的可读性（见 3.14 节中的例子），增强“指向函数的指针”（将在 7.9 节中讨论）以及“指向类的成员函数的指针”（将在 13.6 节中讨论）的可读性。

下面是一个几乎所有人刚开始时都会答错的问题，错误在于将 typedef 当作宏扩展。已知下面的 typedef：

```
typedef char *cstring;
```

在以下声明中，cstr 的类型是什么？

```
extern const cstring cstr;
```

第一个回答差不多都是：

```
const char *cstr
```

即指向 const 字符的指针。但是，这是不正确的。const 修饰 cstr 的类型。cstr 是一个指针，因此，这个定义声明了 cstr 是一个指向字符的 const 指针（见 3.5 节关于 const 指针类型的讨论）

```
char *const cstr;
```

3.13 volatile 限定修饰符

当一个对象的值可能会在编译器的控制或监测之外被改变时，例如一个被系统时钟更新的变量，那么该对象应该声明成 volatile。因此，编译器执行的某些例行优化行为不能应用在已指定为 volatile 的对象上。

volatile 限定修饰符的用法同 const 非常相似——都是作为类型的附加修饰符。例如：

```
volatile int display_register;
volatile Task *curr_task;
volatile int ixa[ max_size ];
volatile Screen bitmap_buf;
```

display_register 是一个 int 型的 volatile 对象。curr_task 是一个指向 volatile 的 Task 类对象的指针。ixa 是一个 volatile 的整型数组。数组的每个元素都被认为是 volatile 的。bitmap_buf 是一个 volatile 的 Screen 类对象，它的每个数据成员都被视为 volatile 的。

volatile 修饰符的主要目的是提示编译器，该对象的值可能在编译器未监测到的情况下被改变。因此编译器不能武断地对引用这些对象的代码作优化处理。

3.14 pair 类型

pair 类也是标准库的一部分，它使得我们可以在单个对象内部把相同类型或不同类型的两个值关联起来。为了使用 pair 类，我们必须包含下面的头文件：

```
#include <utility>
```

例如：

```
pair< string, string > author( "James", "Joyce" );
```

创建了一个 pair 对象 author，它包含两个字符串，分别被初始化为“James”和“Joyce”。

我们可以用**成员访问符号**（member access notation）访问 pair 中的单个元素，它们的名字为 first 和 second。例如：

```
string firstBook;

if ( author.first == "James" &&
     author.second == "Joyce" )
      firstBook = "Stephen Hero";
```

如果我们希望定义大量相同 pair 类型的对象，那么最方便的做法就是用 typedef，如下所示：

```
typedef pair< string, string > Authors;

Authors proust( "marcel", "proust" );
Authors joyce( "james", "joyce" );
Authors musil( "robert", "musil" );
```

下面是第二个 pair。一个元素持有对象的名字，另一个元素持有指向其符号表入口的指针：

```
// 前向声明(forward declaration)
class EntrySlot;
extern EntrySlot* look_up( string );
typedef pair< string, EntrySlot* > SymbolEntry;
SymbolEntry current_entry( "author", look_up( "author" ));
```

```
// ...

if ( EntrySlot *it = look_up( "editor" ))
{
     current_entry.first = "editor";
     current_entry.second = it;
}
```

我们将在第 6 章讨论标准库容器类型、以及第 12 章讨论标准库泛型算法的时候，再次看到 pair 类型。

3.15 类类型

类机制支持新类型的设计，如本章讨论的基于对象的 string、vector、complex、pair 类型，以及第 1 章介绍的面向对象的 iostream 类层次结构。在第 2 章中，我们通过一个 Array 类抽象的实现和进化过程，将支持面向对象的与基于对象的类设计的某本概念和机制快速浏览了一遍。在本节中，我们将简要地介绍一个基于对象的 String 类抽象的设计与实现。它将得益于我们前面给出的、对 C 风格字符串以及标准库 string 类型的讨论。这个实现将着重说明 C++对**操作符重载**（operator overloading）的支持，2.3 节曾简单介绍过这方面的知识。（从第 13 章到第 15 章将详细介绍类。我们在本书的开始部分先介绍类的某些方面，是为了使我们能够在本书 13 章之前就可以提供一些更有意义的、并且用到了类的例子。初次阅读本书的读者可跳过本节，在对后面章节有了更多的了解后，再回头来看。）

现在我们对 String 类应该做些什么已经很清楚：我们需要支持 String 对象的初始化和赋值，包括用字符串文字、C 风格字符串。以及另外一个 String 对象进行初始化或者赋值，我们将通过特定的构造函数以及类特定的”赋值操作符实例来实现这样的功能。

我们需要支持用索引访问 String 中的单个字符，以便与 C 风格字符串和标准库 string 类型具有相同的方式。我们将提供一个类特定的下标操作符实例来做到这一点。

另外，我们还想支持这样一些操作，如确定 String 长度的 size()、两个 String 对象的相等比较，或者 String 同 C 风格字符串的比较、读写一个 String 对象等等。我们将提供等于、iostream 输入、iostream 输出操作符的实例，以实现后两个操作。最后，我们还需要访问底层的 C 风格字符串。

类的定义由关键字 class 开始，后面是一个标识符，该标识符也被用作类的类型指示符，如 complex、vector 及 Array 等等。一般地，一个类包括公有的（public）操作部分和私有的（private）数据部分。这些操作被称为该类的**成员函数**（member function）或**方法**（method），它们定义了类的**公有接口**（public interface）——即，用户可以在该类对象上执行的操作的集合。我们的 String 类的私有数据包括：_string，一个指向动态分配的字符数组的 char*类型的指针，和_size，记录 String 中字符串长度的 int 型变量。下面是我们的定义：

```
#include <iostream>
class String;
```

* 这里的“类特定的”，即 class-specific，是指相应的操作符属于 String 这个类，也就是说与 String 相关联，而不是系统全局缺省的操作符实例。

```
istream& operator>>( istream&, String& );
ostream& operator<<( ostream&, const String& );

class String {
public:
// 一组重载的构造函数
// 提供自动初始化功能
// String str1;            // String()
// String str2( "literal" ); // String( const  char* );
// String str3( str2 );     // String( const String& );
String();
String( const char* );
String( const String& );

// 析构函数: 自动析构
~String();

// 一组重载的赋值操作符
// str1 = str2
// str3 = "a string literal"
String& operator=( const String& );
String& operator=( const char* );

// 一组重载的等于操作符
// str1 == str2;
// str3 == “a string literal”;
bool operator==( const String& );
bool operator==( const char* );

// 重载的下标操作符
// str1[ 0 ] = str2[ 0 ];
char& operator[]( int );

// 成员访问函数
int   size()  { return _size;   }
char* c_str() { return _string; }

private:
int   _size;
char *_string;
};
```

String 类定义了三个构造函数。正如在 2.3 节中简要讨论的那样。重载函数机制允许同函数名或操作符引用到多个实例，只要通过参数表能区分开每个实例就行。我们的三个构造函数形成了一个有效的重载函数集合，首先由参数个数，然后由参数类型来区分它们。第一个构造函数：

```
String();
```

被称做缺省构造函数，因为它不需要任何显式的初始值。当我们写如下语句时：

```
String str1;
```

缺省构造函数将被应用到 str1 上。

另外两个 String 构造函数都有一个参数。当我们写如下语句时：

```
String str2( "a string literal" );
```

根据参数类型，构造函数：

```
String( const char* );
```

被应用在 str2 上。类似地，当我们写如下语句时：

```
String str3( str2 );
```

构造函数：

```
String( const String& );
```

被应用在 str3 上——这是根据被传递给构造函数的参数类型来判断的。这种构造函数被称为**拷贝构造函数**（copy constructor），因为它用另一个对象的拷贝来初始化一个对象。当我们写如下语句时：

```
String str4( 1024 );
```

实参的类型与构造函数集期望的参数类型都不匹配，因此，str4 的定义导致一个编译错误。

被重载的操作符采用下面的一般形式：

```
return_type operator op ( parameter_list );
```

这里的 operator 是关键字，op 是一个预定义的操作符，如“+”、“=”、“==”、“[]”等等（第 15 章有精确的规则）。下面的声明：

```
char& operator[]( int );
```

声明了一个下标操作符的重载实例，它带有一个 int 型的参数，返回指向 char 的引用。重载的操作符还可以被重载，只要每个实例的参数表能够被区分开即可。例如，我们为 String 类提供了两个不同的赋值与等于操作符的实例。

有名字的成员函数可以通过成员访问符号来调用。例如，已知下列 String 定义：

```
String object( "Danny" );
String *ptr = new String( "Anna" );
String array[2];
```

我们可以如下调用成员函数 size()，它们分别返回长度值 5、4 和 0（一会儿我们会看到 String 类的实现）：

```
vector<int> sizes( 3 );

// 针对对象的点成员访问符号 .
// object 的长度为 5
sizes[ 0 ] = object.size();

// 针对指针的箭头成员访问符号->
// ptr 的长度为 4
```

```
sizes[ 1 ] = ptr->size();

// 再次使用点成员访问符号
// array[0] 的长度为 0
sizes[ 2 ] = array[0].size();
```

被重载的操作符也可以直接应用在类对象上。例如：

```
String name1( "Yadie" );
String name2( "Yodie" );

// 应用: bool operator==(const String&)
if ( name1 == name2 )
     return;
else
// 应用: String& operator=( const String& )
     name1 = name2;
```

类的成员函数可以被定义在类的定义中，也可以定义在外面。[例如，size()和 c_str()，都是在 String 类的定义中被定义的。]在类定义之外定义的成员函数不但要告诉编译器它们的名字、返回类型、参数表，而且还要说明它们所属的类。我们应该把成员函数的定义放到一个程序文本文件中（例如，String.C），并且把含有该类定义的头文件（本例中为 String.h）包含进来。例如：

```
// 放在程序文本文件中: String.C
// 包含 String 类的定义
#include "String.h"

// 包含 strcmp()函数的声明
// cstring 是标准 C 库的头文件

#include <cstring>
bool // 返回类型
String:: // 说明这是 String 类的一个成员
operator==    // 函数的名字: 等于操作符
(const String &rhs)   // 参数列表
{
    if ( _size != rhs._size )
         return false;
    return strcmp( _string, rhs._string ) ? false : true;
}
```

strcmp()是 C 标准库函数。它比较两个 C 风格的字符串。如果相等则退回 0，否则返回非 0。条件操作符（?:）测试问号前面的条件，如果为 true，选择问号与冒号之间的表达式，如果为 false，则选择冒号后面的表达式。在本例中，如果 strcmp()返回非 0 值，条件操作符返回 false，否则返回 true（4.7 节将详细讨论条件操作符）。

因为等于操作符是个可能要频繁调用的小函数，因此把它声明成内联（inline）函数是个好办法。内联函数在每个调用点上被展开，因此，这样做可以消除函数调用相关的额外消耗。只要该函数被调用足够多次（7.6 节将详细介绍内联函数），内联函数就能够显著地提高性能。在类定义内部定义的成员函数。如 size()，在缺省情况下被设置为 inline。在类外而定义的成员函数必须显式地声明为 inline：

```
inline bool
String::operator==(const String &rhs)
{
    // 如前
}
```

在类体外定义的内联成员函数，应该被包含在含有该类定义的头文件中。我们在重新定义了等于操作符之后，应当把它的定义从 String.C 移到 String.h 中。

下面是比较 String 对象和 C 风格字符串的等于操作符（它也被定义成内联函数，因而被放在 String.h 头文件中）：

```
inline bool
String::operator==(const char *s)
{
     return strcmp( _string, s ) ? false : true;
}
```

构造函数的名字与类名相同。我们不能在它的声明或构造函数体中指定返回值。它的一个或多个实例都可以被声明成 inline：

```
#include <cstring>
// 缺省构造函数
inline String::String()
{
     _size = 0;
     _string = 0;
}
inline String::String( const char *str )
{
     if ( ! str ) {
          _size = 0; _string = 0;
     }
     else {
          _size = strlen( str );
          _string = new char[ _size + 1 ];
          strcpy( _string, str );
     }
}
// 拷贝构造函数
inline String::String( const String &rhs )
{
      _size = rhs._size;
      if ( ! rhs._string )
          _string = 0;
      else {
          _string = new char[ _size + 1 ];
          strcpy( _string, rhs._string );
      }
}
```

因为我们用 new 表达式动态地分配内存来保留字符串。所以当不再需要该字符串对象的时候，必须用 delete 表达式释放该内存区。这可以通过定义类的析构函数自动实现，把 delete

表达式放在构析函数中。如果类的析构函数存在，那么在每个类的生命期结束时它会被自动调用（第 8 章将解释一个对象的三种可能的生命期）。析构函数由类名前面加一个波浪号（~）来标识。下面是 String 类的析构函数的定义：

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

两个被重载的赋值操作符引用了一个特殊的关键字 this。当我们写如下代码时：

```
String name1( "orville" ), name2( "wilbur" );
name1 = "Orville Wright";
```

在赋值操作符中，this 指向 name1。

更一般的情况下，在类成员函数中 this 指针被自动设置为指向左侧的类对象（我们通过这对象调用这个成员函数）。当我们写如下代码时：

```
ptr->size();
obj[ 1024 ];
```

在 size()中，this 指针指向 ptr；在下标操作符中，this 指针指向 obj。当我们写*this 时，访问的是 this 所指的实际对象（13.4 节将详细讨论 this 指针）。

```
inline String&
String::operator=( const char *s )
{
    if ( ! s ) {
        _size = 0;
        delete [] _string;
        _string = 0;
    }
    else {
        _size = strlen( s );
        delete [] _string;
        _string = new char[ _size + 1 ];
        strcpy( _string, s );
    }
    return *this;
}
```

当我们把一个类对象拷贝给另一个时，最常犯的错误是忘了先测试这两个类对象是否确实是同一个对象。当一个或两个对象都是通过解除一个指针的引用而来时，这个错误最经常发生。此时，this 指针将再次发挥作用，以支持这种测试。例如：

```
inline String&
String::operator=( const String &rhs )
{
    // 在表达式 name1 = *pointer_to_string 中,
    // this 指向 name1,
    // rhs 代表*pointer_to_string. if ( this != &rhs ) {
```

下面是完整的实现代码：

```
inline String&
String::operator=( const String &rhs )
{
    if ( this != &rhs )
```

```
        {
            delete [] _string;
            _size = rhs._size;
            if ( ! rhs._string )
                _string = 0;
            else {
                _string = new char[ _size + 1 ];
                strcpy( _string, rhs._string );
            }
        }
        return *this;
    }
```

下标操作符几乎与 2.3 节中 Array 类的实现相同：

```
#include <cassert>

inline char&
String::operator[]( int elem )
{
    assert( elem >= 0 && elem < _size );
    return _string[ elem ];
}
```

输入操作符和输出操作符是作为非成员函数实现的。（原因将在 15.2 节中讨论。20.4 节与 20.5 节将对重载 iostream 输入和输出操作符进行详细讨论。）我们的输入操作符最多读入 4095 个字符。setw()是一个预定义的 iostream 操纵符。它读入的字符数最多为传递给它的参数减 1。因此，我们可以保证不会溢出 inBuf 字符数组。为了使用它，我们必须包含 iomanip 头文件。[第 20 章将详细讨论 setw()。]

```
#include <iomanip>
inline istream&
operator>>( istream &io, String &s )
{
    // 人工限制最多 4096 个字符
    const int limit_string_size = 4096;
    char inBuf[ limit_string_size ];

    // setw()是 iostream 库的一部分
    // 限制被读取的字符个数为 limit_string_size-1
    io >> setw( limit_string_size ) >> inBuf;
    s = inBuf; // String::operator=( const char* );

    return io;
}
```

为了显示 String，输出操作符需要访问内部的 char“表示。但是，因为它不是类的成员函数，所以它没有访问_string 的权限。有两种可能的解决方案：一种是给输出操作符赋予一个特殊的访问许可[把它声明成类的友元（friend）——我们将在 15.2 节中看到]。第二种方法是提供一个内联的访问函数——在本例中为 c_str()，这是以标准库 string 类提供的解决方案为模型的。下面是实现：

```
inline ostream&
operator<<( ostream& os, String &s )
{
    return os << s.c_str();
}
```

下面的小程序练习了 String 类的实现。它从标准输入读入一个 String 序列，然后再顺序访问 String，并记录出现的元音字母。

```
#include <iostream>
#include "String.h"
int main()
{
    int aCnt = 0, eCnt = 0, iCnt = 0, oCnt = 0, uCnt = 0,
        theCnt = 0, itCnt = 0, wdCnt = 0, notVowel = 0;
    // 为了使用 operator==( const char* )
    // 我们并不定义 The( "The" )和 It( "It" )
    String buf, the( "the" ), it( "it" );

    // 调用 operator>>( istream&, String& )
    while ( cin >> buf ) {
        ++wdCnt;

        // 调用 operator<<( ostream&, const String& )
        cout << buf << ' ';
        if ( wdCnt % 12 == 0 )
            cout << endl;
        // 调用 String::operator==(const String&) and
        //       String::operator==( const char* );
        if ( buf == the || buf == "The" )
            ++theCnt;
        else
            if ( buf == it || buf == "It" )
                ++itCnt;

        // 调用 String::size()
        for ( int ix = 0; ix < buf.size(); ++ix )
        {
            // 调用 String::operator[](int)
            switch( buf[ ix ] )
            {
                case 'a': case 'A': ++aCnt; break;
                case 'e': case 'E': ++eCnt; break;
                case 'i': case 'I': ++iCnt; break;
                case 'o': case 'O': ++oCnt; break;
                case 'u': case 'U': ++uCnt; break;
                default: ++notVowel; break;
            }
        }
    }
    // 调用 operator<<( ostream&, const String& )
    cout << "\n\n"
```

```
            << "Words read: " << wdCnt  << "\n\n"
            << "the/The: "    << theCnt << '\n'
            << "it/It: "      << itCnt  << "\n\n"
            << "non-vowels read: " << notVowel << "\n\n"
            << "a: " << aCnt << '\n'
            << "e: " << eCnt << '\n'
            << "i: " << iCnt << '\n'
            << "o: " << oCnt << '\n'
            << "u: " << uCnt << endl;
}
```

程序的输入是 Stan 写的儿童故事中的一段话（在第 6 章我们会再次看到）。编译并执行程序，它产生如下输出：

```
Alice Emma has long flowing red hair. Her Daddy says when the
wind blows through her hair, it looks almost alive, like a fiery
bird in flight. A beautiful fiery bird, he tells her, magical but
untamed. "Daddy, shush, there is no such thing," she tells him, at
the same time wanting him to tell her more. Shyly, she asks,
"I mean, Daddy, is there?"
Words read: 65

the/The: 2
it/It: 1
non-vowels read: 190

a: 22
e: 30
i: 24
o: 10
u: 7
```

练习 3.26

在 String 类的构造函数和赋值操作符的实现中，有大量的重复代码。请使用 2.3 节中展示的模型，把这些公共代码抽取成独立的私有成员函数。用它们重新实现构造函数和赋值操作符，并重新运行程序以确保其正确。

练习 3.27

修改程序，使其也能够记下辅音字母 b、d、f、s 和 t 的个数。

练习 3.28

实现能够返回 String 中某个字符出现次数的成员函数。声明如下：

```
class String {
public:
    // ...
    int count( char ch ) const;
    // ...
};
```

练习 3.29

实现一个成员操作符函数，它能把一个 String 与另一个连接起来，并返回一个新的 String。声明如下：

```
class String {
public:
    // ...
    String operator+( const 2String &rhs ) const;
    // ...
};
```

4

表达式

在第 3 章中，我们已经介绍了内置类型以及标准库支持的类型、在本章中，我们将了解预定义的操作符，如加、减、赋值、相等测试等等，我们利用这些操作符来操纵数据。然后，再讨论一下操作符的优先级问题。例如：给出表达式 3+4*5，结果总是 23，而不是 35，这是因为乘法运算符的优先级比较高。最后，我们还将讨论对象类型之间的显式和隐式转换。例如，在表达式 3+0.7 中，整数 3 总是在加法执行前先被转换成浮点数。

4.1 什么是表达式?

表达式由一个或多个**操作数**（operand）构成。最简单的表达式由一个文字常量或一个对象构成。一般地，表达式的结果是操作数的右值。例如，下面是三个表达式：

```
void mumble()
{
    3.14159;
    "melancholia";
    upperBound;
}
```

3.14159 的结果是 3.14159，它的类型是 double。“melancholia”的结果是字符串第一个元素的内存地址，它的类型是 const char*。upperBound 的结果是与其相关联的值，类型由它的定义来决定。

在更一般的情况下，表达式由一个或多个操作数、以及应用在这些操作数上的操作构成。例如：下面都是表达式（我们省略了对象的定义：根据操作数的类型，自然会有适当的操作被应用在这些操作数上）：

```
salary + raise
ivec[ size/2 ] * delta
first_name + " " + last_name
```

应用在操作数上的操作由**操作符**（operator）表示。例如，在第一个表达式中，浮点加法操作符被施加在操作数 salary 和 raise 上。在第二个表达式中，操作数 size 被 2 除，其结果被

用来索引整型数组 ivec，然后这个值再被乘以操作数 delta。在第三个表达式中，两个 string 操作数与一个字符串文字连接起来形成一个新的 string，这个动作是通过标准库 string 类定义的加法操作符实例来实现。

作用在一个操作数上的操作符被称为**一元操作符**（unary operator），比如**取地址操作符**（&）和**解引用操作符**（*）。作用在两个操作数上的操作符，比如加法操作符、减法操作符，被称为**二元操作符**（binary operator）。有些操作符既能表示一元操作也能表示二元操作（确切地说，是相同的符号用来表示两个不同的操作）。例如：

```
*ptr
```

表示一元解引用操作符，它返回 ptr 指向的存储区存储的值。而：

```
var1 *var2
```

则表示二元乘法操作符。它计算两个操作数 var1 和 var2 相乘的结果。

表达式的计算是指执行一个或多个操作，最后产生一个结果。除非特别声明，一般来说表达式的结果是个右值。算术表达式结果的类型由操作数的类型来决定。当存在多种数据类型时，编译器将根据一套预定义的类型转换规则集进行类型转换（4.14 节将详细介绍类型转换）。

当两个或两个以上的操作符被组合起来的时候，这样的表达式被称为**复合表达式**（compound expression）。例如，下面表达式的目的是判断指针 ptr 是否指向一个对象（如果它的值不是 0，则它指向一个对象）以及指向的对象是否有一个非零值[10]。

```
ptr != 0 && *ptr != 0
```

整个表达式由三个子表达式构成：ptr 是否为 0 的不等于测试、ptr 的解引用，以及解引用的结果是否为 0 的不等于测试。如果定义 ptr 如下：

```
int ival = 1024;

int *ptr = &ival;
```

解引用子表达式的结果为 1024，两个不等于测试子表达式的结果都是 true。整个表达式的结果也是 true：ptr 没有被设置为 0，并且它指向的对象也没有被设置为 0。（&&操作符被称为逻辑与操作符，或者逻辑 AND 操作符：如果它左右两边的子表达式都为 true，则它的值为 true，否则为 false。）

如果我们进一步仔细地看一下该复合表达式，则会注意到，是否能够成功地计算表达式要取决于子表达式的计算顺序。例如，如果表达式的第二部分先被计算（即，如果在确信指针 ptr 不为 0 之前解引用这个指针）那么，当 ptr 被置为 0 时，程序就可能在运行时刻失败。对于逻辑与操作符，C++严格定义了子表达式的计算顺序：如果左边的子表达式的值为 false，则不计算右边的子表达式，因此上面的错误就不会发生。

在实际情况下，子表达式的计算顺序常常是 C 或 C++初学者出错的根源。这样的错误很难查找，因为这种错误不是凭直觉就能看出来的，除非我们了解子表达式的计算规则。一般

[10] 显式地与 0 测试是可选的，不是必需的。所以下面的表达式与它是等价的：

```
ptr &&*ptr
```

而且这样的表达式更符合实际 C++程序的习惯。

来说，子表达式的计算顺序由操作符的**优先级**（precedence）和**结合性**（associativity）来决定。我们将在了解了 C++支持哪些操作符之后，在 4.13 节中详细地讨论这个问题。以下部分我们将按照一般习惯上的顺序来讨论 C++预定义的操作符。

表格 4-1 算术操作符

操作符	功能	用法
*	乘	expr * expr
/	除	expr / expr
%	求余	expr % expr
+	加	expr + expr
-	减	expr - expr

4.2 算术操作符

两个整数相除的结果是整数。如果商含有小数部分，将被截掉，例如：

```
int ival1 = 21 / 6;
int ival2 = 21 / 7;
```

结果是，ival1 和 ival2 都被 3 初始化。

%操作符计算两个数相除的余数，第一个数被第二个数除。该操作符只能被应用在整值类型（char、short、int 和 long）的操作数上。当两个操作数都是正数时，结果为正。但是，如果有一个（或两个）操作数为负，余数的符号则取决于机器。因此，移植性无法保证。%操作符被称作**取模**（modulus）或**求余**（remainder）操作符：

```
3.14 % 3; // 编译时刻错误：浮点操作数
21 % 6; // ok: 结果是 3
21 % 7; // ok: 结果是 0
21 % -5; // 机器相关：结果为 -1 或 1

int ival = 1024;
double dval = 3.14159;

ival % 12;    // ok: 返回值在 0 和 11 之间
ival % dval;  // 编译时刻错误：浮点操作数
```

在某些实例中，算术表达式的计算会导致不正确或未定义的值，这些情况被称为**算术异常**（arithmetic exception）（但是不会导致抛出实际的异常）。算术异常要归咎于算术的自然本质（比如除以 0）或归咎于计算机的自然本质——比如**溢出**（overflow）（指结果值超出了被赋值对象的类型长度）。例如 8 位的 char。根据它有符号还是无符号，它可以包含最大数 127 或 255。下面的乘法向一个 char 赋值 256，因而导致了溢出：

```
#include <iostream>
```

```
int main() {
    char byte_value = 32;

    int ival = 8;

    // overflow of byte_value's available memory
    byte_value = ival * byte_value;
    cout << "byte_value: " << static_cast<int>(byte_value) << endl;
}
```

表示 256 需要 9 位，因而向 byte_value 赋值 256 导致了与其相关联的内存的溢出。byte_value 包含的实际值是未定义的，所以在执行时就可能会引起问题。例如，在 SGI 工作站上，byte_value 被设置为 0。当用表达式：

```
cout << "byte_value: " << byte_value << endl;
```

试图输出它时，程序输出结果如下：

```
byte_value:
```

经过几分钟的迷惑之后，我们意识到，在 ASCII 码集中，0 代表空（null）字符，所以什么也不输出。如下特殊表达式：

```
static_cast<int> ( byte_value )
```

称为**显式类型转换**（explicit type conversion）或**强制类型转换**（cast）。强制转换使编译器把一个对象（或表达式）从它当前的类型转换成程序员指定的类型。在这种情况下，我们把 byte_value 转换成一个 int 型的对象。现在程序输出：

```
byte_value: 0
```

在本例中，我们改变的不是与 byte_value 相关的值，而是它被输出操作符解释的方式。当它被当作 char 型时，它的值被映射到相关联的 ASCII 表示上（例如，12 表示换行，97 表示小写 a，0 代表空字符等），输出的是它所代表的字符，而不是它的值。当它被看作 int 型时，它的值被直接输出。（我们将在 4.14 节讨论类型转换。）

我们的叙述需要中断一下，转而讨论类型转换以及 byte_value 的输出失败，这有点类似于我们常遇到的情况——即，当程序不能如我们期望的那样运行时，就需要把程序设计任务先放下，去查看一下出了什么问题。那些看起来比较晦涩、并不有趣的语言要素，比如数据类型的长度等，在实践中有时候会影响我们所写的程序。例如，发生在 byte_value 上的溢出错误，就不会被语言捕捉到，因为它涉及到每个计算的运行时刻检查，从性能的角度来看，这是不切合实际的。但是我们必须知道，它是有发生的可能性的。

标准 C++头文件 limits 提供了与内置类型表示有关的信息，例如一个类型能表示的最大值和最小值。另外，C++编译系统也提供了标准 C 头文件 climits 和 cfloat，它们定义了提供类似信息的预处理器宏。怎样使用这些头文件来防止**溢出**（overflow）和**下溢**（underflow），请参见［PLAUGER92］的第 4 章和第 5 章。

浮点数的算术运算还有一个精度问题：在计算机中，当它表示一个数值时，只有固定的数位可以使用。当一个数值被修改，以便适合“用来表示该数的 float、double 或 long double 类型”时，就会发生**浮点舍入**（roundoff）。浮点数加法、乘法和减法的结果精度受到底层

数据类型的固有精度的影响。（关于算术运算舍入错误的详细讨论，请参见［SHAMPINE97］。）

练习 4.1

下列两个除法表达式的主要区别是什么？

```
double dval1 = 10.0, dval2 = 3.0;
int ival1 = 10, ival2 = 3;

dval1 / dval2;
ival1 / ival2;
```

练习 4.2

给出一个有序对象，可用什么操作符来判定它是奇数还是偶数？写出表达式。

练习 4.3

在你的系统中找到并检查 C++头文件 limits 和标准 C 头文件 climits 以及 cfloat。

4.3 等于、关系和逻辑操作符

表格 4.2 等于、关系以及逻辑操作符

操作符	功能	用法
!	逻辑非	!expr
<	小于	expr < expr
<=	小于等于	expr <= expr
>	大于	expr > expr
>=	大于等于	expr >= expr
==	等于	expr == expr
!=	不等于	expr != expr
&&	逻辑与	expr && expr
||	逻辑或	expr || expr

注：这些操作符的结果是 bool 类型。

等于、关系和逻辑操作符的计算结果是布尔常量 true 或 false。如果这些操作符用在要求整数值的上下文环境中，它们的结果将被提升成 1（true）或 0（false）。例如，在下面的代码段中，我们准备统计 vector 中小于某个给定值的元素的个数。为了完成它，我们把小于操

作符的结果加到一个记录元素个数的计数器上。（+=操作符是一个简化记号，它表示把右边的表达式加到左边的操作数的当前值上。我们将在 4.4 节讨论复合赋值操作符。）

```
int elem_cnt = 0;

vector<int>::iterator iter = ivec.begin();
while ( iter != ivec.end() )
{
    // 同下: elem_cnt = elem_cnt + (*iter < some_value)
    // *iter < some _value 的布尔值
    // 将提升为 1 或 0
    elem_cnt += *iter < some_value;
    ++iter;
}
```

只有当逻辑与（&&）操作符的两个操作数都为 true 时，结果值才会是 true。对于逻辑或（||）操作符，只要两个操作数之一为 true，它的值就为 true。这些操作数被保证按从左至右的顺序计算。只要能够得到表达式的值（true 或 false），运算就会结束。给定以下形式：

```
expr1 && expr2
expr1 || expr2
```

如果下列条件有一个满足：

- 在逻辑与表达式中，expr1 的计算结果为 false；
- 在逻辑或表达式中，expr1 的计算结果为 true；

则保证不会计算 expr2。

对于逻辑与操作符，一个很有价值的用法是：在某些使 expr2 的计算变得危险的边界条件出现前，先使 expr1 计算为 false。例如：

```
while ( ptr != 0 &&
    ptr->value < upperBound &&
    ptr->value >= 0 &&
    notFound( ia[ ptr->value ] ))
{ ... }
```

值为 0 的指针不指向任何对象。把成员访问操作符应用在 0 值指针上总会引起麻烦。第一个逻辑与操作符保证不会发生这种事情。数组下标越界是同样麻烦的事情。第二个和第三个操作数保证不会发生这种可能。只有当前三个操作数计算的结果全为 true 的，最后一个操作数才会安全地被计算。

对于逻辑非操作符（!）来说，当它的操作数为 false 或 0 时其值为 true，否则为 false。

```
bool found = false;

// 当未找到所需项
// 且 ptr 仍要寻址某对象时
while ( ! found && ptr ) {
    found = lookup( *ptr );
    ++ptr;
}
```

如下的子表达式：

```
! found
```

只要 found 等于 false，其值就为 true。它是如下显式测试的简短表示：

```
// 含义等同于!found
found == false
```

类似地，下面的测试：

```
if ( found )
```

是如下显式测试表达式的简短表示：

```
if ( found == true )
```

虽然二元关系操作符（小于或等于操作符）的用法十分简单，但是我们必须知道其潜在的缺点，左右操作数的计算顺序在标准 C 和 C++中都是未定义的，因此计算过程必须是与顺序无关的。例如，下列表达式：

```
// 喔！ C++语言本身并没有定义计算顺序
if ( ia[ index++ ] < ia[ index ] )
    // 交换元素
```

程序员假设左边的操作数先计算，因此，比较 ia[0]是否小于 ia[1]，但是，C 或 C++语言并不保证从左到右的计算顺序，实际上的实现可能是先计算右边的操作数，在这种情况下，ia[0]与它自己作比较，而 ia[1]从来没有被计算。安全且可移植的实现如下：

```
if ( ia[ index ] < ia[ index+1 ] )
    // 交换元素
++index;
```

第二个潜在的缺点如下所述。我们的目的是判断 ival、jval 和 kval 是否各不相同。你能看出有什么不对吗？

```
// 喔！ 这样做并不能判断 3 个值是否不相等
if ( ival != jval != kval )
// 省略其他代码
```

正如我们所实现的，相关联的三个值 0、1 和 0 可使我们的表达式值为 true。原因是，第一个不等于表达式的左操作数为 true 或 false，它是第一个表达式的结果——即，kval 被测试是否与转换来的 0 或 1 不相等。要实现我们的测试目的，我们必须重写表达式如下：

```
if ( ival != jval && ival != kval && jval != kval )
    // 省略其他代码
```

练习 4.4

下列哪个表达式不正确或不可移植？为什么？怎样改正？（注意，在这些例子中对象的类型并不重要。）

```
(a) ptr->ival != 0 (b) ival != jval < kval
(c) ptr != 0 && *ptr++ (d) ival++ && ival
(e) vec[ ival++ ] <= vec[ ival ];
```

练习 4.5

二元操作符的计算顺序未定义，因而允许编译器自由地提供可选的实现。这是在“有效的实现”和“程序员使用的语言存在潜在缺点”之间的一种折衷。你认为这是一种可接受的折衷吗？为什么？

4.4 赋值操作符

初始化过程为对象提供了初值。例如：

```
int ival = 1024;
int *pi = 0;
```

而赋值则是用一个新值覆盖对象的当前值。例如：

```
ival = 2048;
pi = &ival;
```

赋值和初始化有时候会被混淆，因为它们都使用同一个操作符（=）。一个对象只能被初始化一次，也就是在它被定义的时候，但是在程序中可以被赋值多次。

当我们把不同类型的表达式赋值给一个对象时，会发生什么事情呢？例如：

```
ival = 3.1415926; // ok?
```

规则是，右边表达式的类型必须与左边被赋值对象的类型完全匹配。在本例中，ival 的类型是 int，而文字常量 3.14159 是 double 类型。这个赋值是错误的吗？不，编译器会试着隐式地将右操作数的类型转换成被赋值对象的类型。如果这种类型转换是有可能的。则编译器会悄悄进行（如果涉及到精度损失，如 double 转换成 int，通常会给出一个警告），在本例中，3.1415926 被转换成 int 型文字常量 3，这正是被赋给 ival 的值。

如果不可能进行隐式的类型转换，那么赋值操作被标记为编译时刻错误。例如，下面的赋值将导致编译错误，因为从 int 型的值到 int* 型没有隐式的类型转换：

```
pi = ival;    // error
```

（C++语言支持的隐式类型转换集合将在 4.14 节讨论。）

赋值操作符的左操作数必须是左值——即，它必须有一个相关联的、可写的地址值。下面是一个明显的非左值赋值的例子：

```
1024 = ival;  // 错误
```

下面是一种可能的解决方案：

```
int value = 1024;
value = ival; // ok
```

然而，在某些情况下，只有左值还不够。例如，已知下列对象定义：

```
const int array_size = 8;

int ia[ array_size ] = { 0, 1, 2, 2, 3, 5, 8, 13 };
int *pia = ia;
```

以下赋值操作：

```
array_size = 512; // 错误
```

是非法的，尽管 array_size 是一个左值，但是，array_size 的 const 定义使它的地址值个可写。类似地，以下赋值操作：

```
ia = pia;// 错误
```

是非法的。尽管 ia 是个左值，但是数组对象本身不能被赋值，只有它包含的元素才能被赋值。

而赋值操作：

```
pia + 2 = 1;  // 错误
```

也是非法的。尽管 pia + 2 计算出 ia[2]的地址，但是结果不是一个可写的地址值。然而，如果把解引用操作符应用在地址值上，如：

```
*(pia + 2) = 1;    // ok
```

则赋值就是正确的：解引用操作符表示赋值是对 pia+2 指向的对象的。

赋值的结果是实际上被放在左操作数相关内存中的值。例如如下赋值的结果是 0：

```
ival = 0;
```

而如下赋值的结果是 3：

```
ival = 3.14159;
```

两者都是 int 型。因此，赋值表达式可以被当作一个子表达式。例如，下面的 while 循环

```
extern char next_char();

int main()
{
    char ch = next_char();

    while ( ch != '\n' ) {
        // 省略代码
        ch = next_char();
    }
    // ...
}
```

可以被改写成；

```
extern char next_char();

int main()
{
    char ch;

    while (( ch = next_char() ) != '\n' ) {
        // do something ...
    }
    // ...
}
```

外加的小括号是必需的，因为赋值操作符的优先级低于不等于操作符。优先级决定了表

达式中计算的顺序，优先级高的先计算。没有小括号，那么不等于测试：

```
next_char() != '\n'
```

将先被计算，然后 ch 才被赋值为 false 或 true，即测试 next_char()是否不等于换行符的结果——显然，这不是我们想要的。（我们将在 4.13 节详细了解优先级。）

类似地，赋值操作符也可以被连接在一起，只要每个被赋值的操作数都是相同的数据类型。例如，在下列代码中：

```
int main()
{
    int ival, jval;
    ival = jval = 0;  // ok: 两个都被赋为 0
    // ...
}
```

ival 和 jval 都被赋为 0。但是，以下代码是非法的，因为 ival 和 pval 是不同类型的对象，尽管 0 可以被赋给它们中的任意一个：

```
int main()
{
    int ival; int *pval;
    ival = pval = 0; // error: not the same types
    // ...
}
```

下列赋值有可能合法，也可能非法，但它不能用作 ival 和 jval 的定义：

```
int main()
{
    // ...
    int ival = jval = 0; // 可能合法，也可能不合法
    // ...
}
```

只有当 jval 在前面已经被定义，而且是可被赋值为 0 的某些类型时，这个例子才是合法的。在这种情况下，ival 被初始化为向 jval 赋值 0 的结果，也是 0。为了让它定义两个对象，我们必须重写代码：

```
int main()
{
    // ok: 定义 ival 和 jval…
    int ival = 0, jval = 0;
    // ...
}
```

我们经常把某个操作符应用在一个对象上，然后再把结果赋给这个对象，如：

```
int arraySum( int ia[], int sz )
{
    int sum = 0;
    for ( int i = 0; i < sz; ++i )
        sum = sum + ia[ i ];
    return sum;
    a op= b;
}
```

为此，C++提供了一套复合赋值操作符。例如，前面的函数可以用“**复合赋值加操作符**（compound assignment-plus operator）”重写为：

```
int arraySum( int ia[], int sz )
{
    int sum = 0;

    for ( int i = 0; i < sz; ++i )
        // equivalent of: sum = sum + ia[ i ];
        sum += ia[ i ];
    return sum;
}
```

复合赋值操作符的一般语法格式是：

```
a op= b;
```

这里的 op=可以是下列十个操作符之一：

```
+=   -=   *=   /=   %=
<<=  >>=  &=   ^=   |=
```

每个复合赋值操作符都等价于以下“普通写法”的赋值：

```
a = a op b;
```

例如，数组 ia 求和的普通写法为：

```
sum = sum + ia [ i ];
```

练习 4.6

下列代码合法吗？为什么？怎样改正？

```
int main() {
    float fval;

    int ival;
    int *pi;
    fval = ival = pi = 0;
}
```

练习 4.7

虽然下列表达式不是非法的，但是它们的行为并不像程序员期望的那样。为什么？怎样修改以使其能反映程序员的可能意图？

```
(a) if ( ptr = retrieve_pointer() != 0 )
(b) if ( ival = 1024 )
(c) ival += ival + 1;
```

4.5 递增和递减操作符

递增（++）和递减（--）操作符为对象加 1 或减 1 操作提供了方便简短的表示。它们最一般的用法是对索引、迭代器或指向一个集合内部的指针加 1 或减 1。例如：

```
#include <vector>
#include <cassert>

int main()
{
    int ia[10] = {0,1,2,3,4,5,6,7,8,9};
    vector< int > ivec( 10 );
    int ix_vec = 0, ix_ia = 9;
    while ( ix_vec < 10 )
        ivec[ ix_vec++ ] = ia[ ix_ia-- ];
    int *pia = &ia[9];
    vector<int>::iterator iter = ivec.begin();
    while ( iter != ivec.end() )
        assert( *iter++ == *pia-- );
}
```

表达式：

```
ix_vec++
```

是递增操作符的后置（postfix）形式。它在用 ix_vec 的当前值索引 ivec 之后，将 ix_vec 递增 1。例如。while 循环的第一次迭代，ix_vec 的值为 0，这个值被用来索引 ivec。然后，ix_vec 被递增为 1，但这个新值直到下一次迭代才能被实际使用。递减操作符的后置形式用法相同。ix_ia 的当前值被用来索引 ia，然后 ix_ia 被递减 1。

C++也支持这两个操作符的前置（prefix）版本。在前置形式中，当前值先被递增或递减 1，然后再使用它的值。因此，如果写：

```
// 错误：两端都差一
int ix_vec = 0, ix_ia = 9;

while ( ix_vec < 10 )
     ivec[ ++ix_vec ] = ia[ --ix_ia ];
```

则在 ix_vec 的值被用来索引 ivec 之前，它先被递增变成 1。类似地，ix_ia 在被用来索引 ia 之前先被递减变成 8。为了使循环能正确执行，我们必须将两个索引的初始值设为一个比实际访问的值小 1，另一个比实际访问的值大 1。

```
// ok: 两端都是正确的
int ix_vec = -1, ix_ia = 10;

while ( ix_vec < 10 )
     ivec[ ++ix_vec ] = ia[ --ix_ia ];
```

作为最后一个例子，我们考虑栈（stack）的设计。栈是一个基本的计算机科学的数据抽象，它允许以后进先出（LIFO）的顺序放入或取出数值。栈的两个基本操作是“向栈中压入（push）一个新的值”以及“从栈中弹出（pop）最后的值”。为讨论方便，假设我们用 vector 来实现栈。

我们的栈维护了一个对象 top，它表示下一个可用来压入数据值的槽。要实现压入（push）语义，我们必须把这个值赋给由 top 表示的槽，然后再将 top 增加 1。这种情况需要哪种形式的递增操作符呢？我们希望先使用当前的值，然后再把它加 1。这正好是后置形式的行为：

```
stack[ top++ ] = value;
```

要实现弹出（pop）语义，则必须先将 top 减 1，然后再返回减 1 后的 top 值所指的槽内的内容，这正是前置形式的行为：

```
int value = stack[ --top ];
```

（在本章最后将提供栈类的实现。标准库的栈类将在 6.16 节讨论。）

练习 4.8

你认为为什么 C++不叫++C？

4.6 复数操作

标准库提供的**复数**（complex）类是基于对象的类抽象的完美模型。通过使用操作符重载。我们几乎可以像使用简单内置类型一样容易地使用复数类型的对象。正如本节我们将要看到的，C++不但支持一般的算术操作符，如加、减、乘、除，而且还支持复数类型与内置类型的混合运算。在程序员看来，尽管复数的实现是在标准库中，但它也是基本语言的一部分。（注意本节只说明复数类的用法，有关复数的数学知识，请参见 [PERSON68] 或任意一本关于初等数学的书籍。）例如可以写：

```
#include <complex>

complex< double > a;
complex< double > b;

// ... assign to a and b ...
complex< double > c = a * b + a / b;
```

在表达式中，我们可以混合复数类型和算术数据类型，例如：

```
complex< double > complex_obj = a + 3.14159;
```

类似地，我们也可以用一个算术数据类型的值对复数初始化或赋值，如；

```
double dval = 3.14159;

complex_obj = dval;
```

或

```
int ival = 3;
```

但是，相反的情形并不被自动支持，也就是说，算术数据类型不能直接被一个复数类对象初始化或赋值。例如，下列代码将导致编译错误：

```
// 错误：从复数到内置算术数据类型之间
// 并没有隐式转换支持
double dval = complex_obj;
```

如果我们真想这样做，则必须显式地指明我们要用复数对象的哪部分来赋值。复数类支

持一对操作，可用来读取一部或者虚部。例如，我们可以用成员访问语法（member access syntax）：

```
double re = complex_obj.real();
double im = complex_obj.imag();
```

或者用等价的非成员语法：

```
// 等价于上面的成员语法
double re = real( complex_obj );

double im = imag( complex_obj );
```

复数类支持四种复合赋值操作符：分别是加赋值（+=）、减赋值（-=）、乘赋值（*=）以及除赋值（/=）。因此，我们可以写：

```
complex_obj += second_complex_obj;
```

C++支持复数的输入和输出。复数的输出是一个由逗号分隔的序列对，它们由括号括起第一个值是实部，第二个值是虚部。例如：

```
complex< double > complex0( 3.14159, -2.171 );
complex< double > complex1( complex0.real() );

cout << complex0  << " "  << complex1 << endl;
```

产生下列输出：

```
( 3.14159, -2.171 ) ( 3.14159, 0.0 )
```

下列任意一种数值表示格式都可以被读作复数：

```
// 复数的有效输入格式
// 3.14159 ==> complex( 3.14159 );
// ( 3.14159 ) ==> complex( 3.14159 );
// ( 3.14, -1.0 ) ==> complex( 3.14, -1.0 );

// 可以被读入复数对象中
// cin >> a >> b >> c
// 这里 a、b 和 c 为复数对象
3.14159 ( 3.14159 ) ( 3.14, -1.0 )
```

复数类支持的其他操作包括 sqrt()、abs()、polar()、sin()。cos()、tan()、exp()、log()、log10()以及 pow()。

练习 4.9

当我写这本书的时候，在标准库的 Rogue Wave 实现版本中，对于复合赋值操作符，只有当右操作数是复数的时候它才是合法的。例如，下面这样的写法：

```
complex_obj += 1;
```

将导致编译错误，尽管标准 C++认为这里的赋值是合法的（跟不上 C++标准的实现是很普遍的）。我们可以通过“提供自己的复合赋值操作符实例”来修正这个错误。例如，下面

是一个 complex<double>的非模板的复合加法赋值操作符的实例：

```
#include <complex>

inline complex<double>&
operator+=( complex<double> &cval, double dval )
{
     return cval += complex<double>( dval );
}
```

当我们在程序中包含这个实例时，上面 1 的复合赋值就能正确执行。（这是为某一个类型提供重载操作符的例子。操作符重载将在第 15 章讨论。）

用前面的定义作模型，我们可以为 complex<double>提供另外三个复合操作符的实现。把它们加到下面的小程序中并运行它们。

```
#include <iostream>
#include <complex>

// 把复合操作符的定义放在这里
int main()
{
     complex< double > cval( 4.0, 1.0 );
     cout << cval << endl;
     cval += 1;
     cout << cval << endl;
     cval -= 1;
     cout << cval << endl;
     cval *= 2;
     cout << cval << endl;
     cout /= 2;
     cout << cval << endl;
}
```

练习 4.10

标准 C++不提供对复数类型递增操作符的支持，尽管这不是由于复数自身的原因——毕竟，如下语句：

```
cal += 1;
```

实际上是把 cval 的实部加 1。请提供递增操作符的定义，并把它加到下列程序中，然后编译并运行它：

```
#include <iostream>

#include <complex>

// 递增操作符的定义在这里
```

```
int main()
{
    complex< double > cval( 4.0, 1.0 );
    cout << cval << endl;
    ++cval;

    cout << cval << endl;
}
```

4.7 条件操作符

条件操作符为简单的 if-else 语句提供了一种便利的替代表示法。例如我们不必这样写：

```
bool is_equal = false;
if ( !strcmp( str1, str2 ))
    is_equal = true;
```

而可以写成：

```
bool is_equal = !strcmp( str1, str2 ) ? true : false;
```

条件操作符的语法格式如下：

```
expr1 ? expr2 : expr3;
```

expr1 的计算结果不是 true 就是 false。如果它是 true，则 expr2 被计算，否则 expr3 被计算。例如：

```
int min( int ia, int ib )
{
    return ( ia < ib ) ? ia : ib; }
}
```

是如下代码的简写形式：

```
int min( int ia, int ib ) {
    if ( ia < ib )
        return ia;

    return ib;
}
```

下面的程序说明了怎样使用条件操作符：

```
#include <iostream>

int main()
{
    int i = 10, j = 20, k = 30;

    cout << "The larger value of "
        << i << " and " << j << " is "
        << ( i > j << i : j ) << endl;

    cout << "The value of " << i << " is"
```

```
            << ( i % 2 << " odd." : " even." )
            << endl;

        /* 条件操作符可以被嵌套，
        /* 但是深度的嵌套比较难读
        /* 在本例中，
        /* max 被设置为 3 个变量中的最大值
        */
        int max = ( (i > j)
            ? (( i > k) ? i : k)
            : ( j > k ) ? j : k);

        cout << "The larger value of "
            << i << ", " << j << " and " << k
            << " is " << max << endl;
    }
```

编译并运行这个程序，产生下列输出：

```
The larger value of 10 and 20 is 20
The value of 10 is even
The larger value of 10, 20 and 30 is 30
```

4.6 sizeof 操作符

siseof 操作符的作用是返回一个对象或类型名的字节长度。它有以下三种形式：

```
sizeof (type name );
sizeof ( object );
sizeof object;
```

返回值的类型是 size_t，这是一种与机器相关的 typedef 定义，我们可以在 cstddef 头文件中找到它的定义。下面的例子使用了 sizeof 的两种格式：

```
#include <cstddef>
int ia[] = { 0, 1, 2 };

// sizeof 返回整个数组的大小
size_t array_size = sizeof ia;

// sizeof 返回 int 类型的大小
size_t element_size = array_size / sizeof( int );
```

当 sizeof 操作符应用在数组上时，例如上面例子中的 ia，它返回整个数组的字节长度，而不是第一个元素的长度，也不是 ia 包含的元素的个数。例如，在一台 int 类型是 4 个字节长的机器上，sizeof 指示 ia 的长度是 12 字节。类似地，当我们写如下代码时：

```
int *pi = new int[ 3 ];

size_t pointer_size = sizeof ( pi );
```

sizeof(pi)返回的值是指向 int 型的指针的字节长度，而不是 pi 指向的数组的长度

下面的小函数可以用来练习 sizeof()操作符：

```
#include <string>
#include <iostream>
#include <cstddef>

int main()
{
    size_t ia;
    ia = sizeof( ia ); // ok
    ia = sizeof ia; // ok

    // ia = sizeof int; // 错误
    ia = sizeof( int ); // ok
    int *pi = new int[ 12 ];
    cout << "pi: " << sizeof( pi )
         << " *pi: " << sizeof( *pi )
         << endl;

    // 一个 string 的大小与它所指的字符串的长度无关
    string st1( "foobar" );
    string st2( "a mighty oak" );
    string *ps = &st1;
    cout << "st1: "   << sizeof( st1 )
         << " st2: "  << sizeof( st2 )
         << " ps: "   << sizeof( ps )
         << " *ps: "  << sizeof( *ps )
         << endl;
    cout << "short :\t" << sizeof(short) << endl;
    cout << "short* :\t"<< sizeof(short*)   << endl;
    cout << "short& :\t"<< sizeof(short&)   << endl;
    cout << "short[3] :\t" << sizeof(short[3]) << endl;
}
```

编译并运行它，产生如下结果：

```
pi: 4 *pi: 4
st1: 12 st2: 12 ps: 4 *ps: 12
short : 2
short* : 4
short& : 2
```

```
short[3] : 6
```

正如上面的例子程序所显示的那样，应用在指针类型上的 sizeof 操作符返回的是包含该类型地址所需的内存长度。但是，应用在引用类型上的 sizeof 操作符返回的是包含被引用对象所需的内存长度。

sizeof 操作符应用在 char 类型上时，在所有的 C++实现中结果都是 1。

```
// 在所有的实现中，保证为 1
size_t char_size = sizeof( char );
```

sizeof 操作符在编译时刻计算，因此被看作是常量表达式。它可以用在任何需要常量表达式的地方。如数组的维数或模板的非类型参数。例如：

```
// ok: 编译时刻常量表达式
int array[ sizeof( some_type_T )];
```

4.9 new 和 delete 表达式

系统为每个程序都提供了一个在程序执行时可用的内存池。这个可用内存池被称为程序的**空闲存储区**（free store）或**堆**（heap）。运行时刻的内存分配被称为**动态内存分配**（dynamic memory allocation）。正如我们在第 1 章中所看到的，动态内存分配由 new 表达式应用在一个**类型指示符**（specifier）上来完成，类型指示符可以是内置类型或用户定义类型。new 表达式返回指向新分配的对象的指针。

例如：

```
int *pi = new int;
```

从空闲存储区中分配了一个 int 型的对象，并用它的地址初始化 pi。在空闲存储区内实际分配的对象并没有被初始化。我们可以如下指定一个初始值：

```
int *pi = new int( 1024 );
```

它不但分配了这个对象而且用 1024 将其初始化。

要动态分配一个对象数组，我们可以写成：

```
int *pia = new int[ 10 ];
```

它从空闲存储区中分配了一个数组，其中含有 10 个 int 型对象，并用它的地址初始化 pin，而数组的元素没有被初始化。没有语法能为动态分配的数组的元素指定一个显式的初始值集合。（在类对象数组的情况下，如果我们定义了缺省构造函数，那么它将被顺次应用在数组的每一个元素上。）例如：

```
string *ps = new string;
```

从空闲存储区分配了一个 string 类对象，并用它的地址初始化 ps，然后再在该对象上调用 string 类缺省构造函数。类似地，如下语句：

```
string *psa = new string[ 10 ];
```

从空闲存储区分配了一个含有 10 个 string 类对象的数组，用它的地址初始化 psa，然后

依次在每个元素上调用 string 类的缺省构造函数。

所有从空闲存储区分配的对象都是未命名的，这是它的另一个特点。new 表达式并不返回实际被分配的对象，而且返回这个对象的地址。对象的所有操作都通过这个地址间接来完成。

当对象完成了使命时，我们必须显式地把对象的内存返还给空闲存储区。我们通过把 delete 表达式应用在“指向我们用 new 表达式分配的对象指针”上来做到这一点（delete 表达式不应该被应用在“不是通过 new 表达式分配的指针”上）。例如：

```
delete pi;
```

释放了 pi 指向的 int 对象，将其返还给空闲存储区。类似地：

```
delete ps;
```

在 ps 指向的 string 类对象上应用 string 的析构函数后，释放其存储区，并将其返还给空闲存储区。最后：

```
delete [] pia;
```

释放了 pia 指向的 10 个 int 对象的数组，并把相关的内存区返还给空闲存储区。在关键字 delete 与指针之间的空方括号表示 delete 的一种特殊语法，它释放由 new 表达式分配的数组的存储区。

第 8 章将详细介绍动态内存分配以及 new 表达式与 delete 表达式的用法。

练习 4.11

下列语句哪些是非法的或错误的？

```
(a) vector<string> svec( 10 );
(b) vector<string> *pvec1 = new vector<string>(10);
(c) vector<string> **pvec2 = new vector<string>[10];
(d) vector<string> *pv1 = &svec;
(e) vector<string> *pv2 = pvec1;
(f) delete svec;
(g) delete pvec1;
(h) delete [] pvec2;
(i) delete pv1;
(j) delete pv2;
```

4.10 逗号操作符

逗号表达式是一系列由逗号分开的表达式。这些表达式从左向右计算。逗号表达式的结果是最右边表达式的值。在下面的例子中，条件操作符的每边都是逗号表达式。第一个逗号表达式的值是 ix，而第二个表达式的值是 0。

```
int main()
{
```

```
        // examples of a comma expression
        // ia, sz, and index are defined elsewhere ...
        int ival = (ia != 0)
            ? ix=get_value(), ia[index]=ix
            : ia=new int[sz], ia[index ]=0;
        // ...
    }
```

4.11 位操作符

表格 4.3 位操作符

操作符	功能	用法
~	按位非	~expr
<<	左移	expr1 << expr2
>>	右移	expr1 >> expr2
&	按位与	expr1 & expr2
^	按位异或	expr1 ^ expr2
|	按位或	expr1 | expr2
&=	按位与赋值	expr1 &= expr2
^=	按位异或赋值	expr1 ^= expr2
|=	按位或赋值	expr1 |= expr2

位操作符把操作数解释成有序的位集合，这些位可能是独立的，也可能组成**域**（field）。每个位可以含有 0（off）或 1（on）。位操作符允许程序员设置或测试独立的位或位域。如果一个对象被用作一组位或位域的离散集合，那么这样的对象称为**位向量**（bitvector）。位向量是一种用来记录一组项目或条件的是/否信息［有时也称为**标志**（flag）］的紧缩方法。例如，在编译器中，类型声明的**限定修饰符**（qualifier），如 const 和 volatile，有时就被存储在位向量中。iostream 库用位向量表示格式状态，例如输出的整数是以十进制、十六进制，还是八进制显示。

正如标准 C++有两种方式支持字符串（string 类类型和 C 风格字符串）以及元素的有序集合（模板 vector 类和内置数组类型）一样，它也有两种方式支持位向量。在 C 语言和标准 C++之前，它用内置整值类型来表示位向量，典型的情况是用 unsigned int。对象提供位的容器，程序员用本节讨论的位操作符来管理语义。标准库提供了一个 bitset 类，它支持位向量的类抽象。bitset 对象封装了位向量的语义，它回答了诸如以下问题的询问：有设置为 1 的位吗？设置了多少位？它提供了一组用于管理位的设置、复位和测试的操作。

通常情况下，我们建议使用标准库的类抽象——在这种情况下，我们使用 bitset 类，而不是直接按位操作整值数据类型。但是，我们认为，知道这两种表示法仍然是有必要的，因为我们可能要阅读或修改已有的一些代码。考虑到本书的完整性，我们将对这两种方法做必要的解释。在本节余下部分，我们将了解内置整值类型作为位向量的用法以及位操作符的用法。下一节将介绍 bitset 类。

用整值数据类型作为位向量时，类型可以是有符号的，也可以是无符号的，强烈建议使用无符号类型。因为在大多数的位操作中，符号位的处理是未定义的，因此在不同的实现中符号位的处理可能会不同。在一个实现下可行的程序，在另一个实现下有可能会失败。

按位非操作符（~）翻转操作数的每一位，每个 1 被设置为 0，而每个 0 被设置为 1。

移位操作符（<<，>>）将其左边操作数的位向左或右移动某些位。操作数中移到外面的位被丢弃。左移操作符（<<）从右边开始用 0 补空位。如果操作数是无符号数，则右移操作符（>>）从左边开始插入 0，否则的话，它或者插入符号位的拷贝，或者插入 0，这由具体实现定义。

按位与操作符（&）需要两个整值操作数。在每个位所在处，如果两个操作数都含有 1，则结果该位为 1，否则为 0。[请不要把该操作符与逻辑与（&&）操作符相混淆。不幸的是，好像每个人都会混淆一两次。]

按位异或操作符（^）需要两个整值操作数。在每个位所在处，如果两个操作数只有一个（注意不是两个）含有 1，则结果该位为 1，否则为 0。

按位或操作符（|）需要两个整值操作数。在每个位所在处，如果两个操作数有一个或者两个含有 1，则结果该位为 1，否则为 0。（请不要把该操作符与逻辑或（||）操作符混淆。）

让我们来看一个简单的例子。一个老师教一个有 30 名学生的班级。每周这个班都有一个通过/不通过的测试。我们用一个位向量来记录每次测试的结果。（注意，每个位的位置都是从 0 开始计数的，因此位置 1 实际上代表了第二位。在本例中，我们为了把位置 1 做成第一位，位置 2 做成第二位等等，浪费了第一位。毕竟我们的老师不是计算机科学的学生。）

```
unsigned int quiz 1 = 0;
```

这个老师必须能够翻转和测试独立的位。例如，27 号学生补考并通过了，那么，老师必须将第 27 位设置为 1。第一步是将一个整数的第 27 位设为 1，而其他位保持为 0。这可以用整形常数和左移操作符来实现：

```
1 << 27;
```

如果这个值与 quiz1 按位或，则除了第 27 位，其他位都没有改变。第 27 位被设置为 1：

```
quiz1 |= 1 << 27;
```

假设这个老师重新检查了测验结果，发现 27 号学生实际上并没有通过补考。那么现在这个老师必须将第 27 位再改为 0。注意，这次是将前面的整数翻转。将按位非操作符应用在前面的整数上，会把除了第 27 位外的每一位都设置为 1：

```
~( 1<<27 );
```

如果该值与 quiz1 按位与，则除了第 27 位外，其他位都保持不变，而第 27 位被设置为 0：

```
quiz1 &= ~(1<<27);
```

下面给出这个老师怎样判断某位是 0 还是 1，还是考虑 27 号学生（实际上，她的名字叫 Anna）。第一步是将一个整数的第 27 位设置为 1。然后再把该值与 quiz1 按位与，如果 quiz1 的第 27 位也是 1，则结果为 true，否则结果为 false。

```
bool hasPassed = quiz1 & (1<< 27);
```

由于位操作符在较低的层次上操纵位，所以它比较容易出错，因此在典型情况下，它们被封装在预处理器宏或内联函数中。例如：

```
inline bool bit_on( unsigned int ui, int pos )
{
    return ui & (1 << pos );
}
```

可以如下调用它们：

```
enum students { Danny = 1, Jeffrey, Ethan, Zev, Ebie, // ...
    AnnaP = 26, AnnaL = 27 };
const int student_size = 27;

//deliberately starts at 1
bool has_passed_quiz[ student_size+1 ];

for ( int index = 1; index <= student_size; ++index )
    has_passed_quiz[ index ] = bit_on( quiz1, index );
```

当然，一旦我们封装了位操作符的直接用法，下一个逻辑步骤就是要提供整个位向量的封装——在标准库的情况下，也就是 bitset 类抽象。这正是下一节的主题。

练习 4.12

假设有下面两个定义：

```
unsigned int ui1 = 3, ui2 = 7;
```

下列表达式的结果是什么？

```
(a) ui1 & ui2 (c) ui1 | ui2
(b) ui1 && ui2    (d)ui1 || ui2
```

练习 4.13

请以内联函数 bit_on()为模型，提供一组内联函数，它们的操作作用在由 unsigned int 表示的位向量上。这组函数包括 bit_turn_on()（将指定位设置为 1），bitoff()（测试指定的位是否为 0），bit_turn_off()（将指定位设置为 0）和 flip_bit()（将指定位翻转）。然后写一个小程序练习这些函数。

练习 4.14

在练习 4.13 中，显式地编写函数来操作 unsigned int 对象的缺点是什么？一种替代方法是使用 typedef，另一种替代方法是使用 2.5 节介绍的模板机制。分别用 typedef 和模板机制

重写上面的内联函数 bit_on()。

4.12 bitset 操作

表格 4.4 bitset 操作

操作	功能	用法
test(pos)	pos 位是否为 1?	a.test(4)
any()	任意位是否为 1?	a.any()
none()	是否没有位为 1?	a.none()
count()	值是 1 的位的个数	a.count()
size()	位元素的个数	a.size()
[pos]	访问 pos 位	a[4]
flip()	翻转所有的位	a.flip()
flip(pos)	翻转 pos 位	a.flip(4)
set()	将所有位置 1	a.set()
set(pos)	将 pos 位置 1	a.set(4)
reset()	将所有位置 0	a.reset()
reset(pos)	将 pos 位置 0	a.reset(4)

用整值类型表示位向量的问题在于，使用位操作符来设置、复位和测试单独的位，层次比较低，也比较复杂。例如，用整值类型将第 27 位设置为 1，我们这样写：

```
quiz1 |= 1<<27;
```

而用 bitset 来做，我们可以写：

```
quizl[ 27 ] = 1;
```

或：

```
quiz1.set( 27 );
```

（正如上节所提到的，位的计数从 0 开始。实际上，27 位指第 28 位。在本例中，我们浪费了第一位，所以我们的位从 1 开始。）

要使用 bitset 类，我们必须包含相关的头文件：

```
#include <bitset>
```

bitset 有三种声明方式。在缺省定义中，我们只需简单地指明位向量的长度。例如：

```
bitset< 32 > bitvec;
```

声明了一个含有 32 个位的 bitset 对象，位的顺序从 0 到 31。缺省情况下，所有的位都被初始化为 0。为了测试 bitset 对象是否含有被设置为 1 的位，我们可以使用 any()操作：当 bitset 对象的一位或多个位被设置为 1 时，any()返回 true。对于 bitvec，如下测试：

```
bool is_set = bitvec.any();
```

它的结果当然是 false。相反，如果 bitset 对象的所有位都被设置为 0，则 none()操作返回 true。对于 bitvec，测试：

```
bool is_not_set = bitvec.none();
```

结果为 true。count()操作返回被设置为 1 的位的个数：

```
int bits_set = bitvec.count();
```

我们可以用 set()操作或者下标操作符来设置某个单独的位。例如，下面的 for 循环把偶数位设置为 1：

```
for ( int index = 0; index < 32; ++ index )
     if ( index % 2 == 0 )
          bitvec[ index ] = 1;
```

类似地，测试某个单独的位是否为 1 也有两种方式。test()操作用位置做参数，返回 true 或 false。例如：

```
if ( bitvec.test( 0 ))
     // 我们的 bitve[0] 可以工作了!
```

同样地，我们也可以用下标操作符：

```
cout << "bitvec: positions turned on:\n\t";

for ( int index = 0; index < 32; ++index )
     if ( bitvec[ index ] )
          cout << index << " ";

cout << endl;
```

要将某个单独的位设置为 0，我们可以用 reset()或下标操作符。下列两个操作都将 bitvec 的第一位设为 0：

```
// 两者等价，都把第一位设置为 0
bitvec.reset( 0 );
bitvec[ 0 ] = 0;
```

我们也可以用 set()和 reset()操作将整个 bitset 对象的所有位设为 1 或 0。只要调用相应的操作，而不必传递位置参数，我们就可以做到这一点。例如：

```
// 把所有的位设置为 0
bitvec.reset();

if ( bitvec.none() != true )
     // 喔! 错了
     // 把所有的位设置为 1
     bitvec.set();
```

```
if ( bitvec.any() != true )
    // 喔！又错了
```

flip()操作翻转整个 bitset 对象或一个独立的位：

```
bitvec.flip( 0 ); // 翻转第一位
bitvec[0].flip(); // 也是翻转第一位
bitvec.flip();    // 翻转所有的位的值
```

还有两种方法可以构造 bitset 对象，它们都提供了将某位初始化为 1 的方式。一种方法是，为构造函数显式地提供一个无符号参数。bitset 对象的前 N 位被初始化为参数的相应位值。例如：

```
bitset< 32 > bitvec2( 0xffff );
```

将 bitvec2 的低 16 位设为 1：

```
00000000000000001111111111111111
```

下面的 bitvec3 的定义：

```
bitset< 32 > bitvec3( 012 );
```

将第 1 和 3 位的值设置为 1（假设位置从 0 开始计数）：

```
00000000000000000000000000001010
```

我们还可以传递一个代表 0 和 1 的集合的字符串参数来构造 bitset 对象，如下所示：

```
// 与 bitvec3 的初始化等价
string bitval( "1010" );
bitset< 32 > bitvec4( bitval );
```

bitvec4 和 bitvec3 的第 1 和 3 位都被设置为 1，而其他位保持为 0。

我们还可以标记用来初始化 bitset 的字符串的范围。例如，在下面的语句中：

```
// 从位置 6 开始，长度为 4：1010
string bitval( "11111101011000011010101" );
bitset< 32 > bitvec5( bitval, 6, 4 );
```

bitvec5 的第 1 和第 3 位被初始化为 1，其他位为 0，同 bitvec3 和 bitvec4 一样。如果去掉用来指示字符串范围长度的第 3 个参数，那么，初始化字符的范围由指定的位置开始一直到字符串的末尾。例如：

```
// 从位置 6 开始，直到字符串结束：1010101
string bitval( "11111101011000011010101" );
bitset< 32 > bitvec6( bitval, 6 );
```

bitset 类支持两个成员函数，它们能将 bitset 对象转换成其他类型。一种情况是用 to_string()操作，将任意 bitset 对象转换成 string 表示：

```
string bitval( bitvec3.to_string() );
```

另一种情况是用 to_ulong()操作，将任意 bitset 对象转换成 unsigned long 型的整数表示，只要该 bitset 对象的底层表示能用一个 unsigned long 来表示。在需要把 bitset 对象传递给 C

或标准 C++之前的程序时，这尤其有用。

bitset 类支持位操作符。例如：

```
bitset<32> bitvec7 = bitvec2 & bitvec3;
```

把 bitvec7 初始化为两个位向量按位与的结果，而：

```
bitset<32> bitvec8 = bitvec2 | bitvec3;
```

把 bitvec8 初始化为按位或的结果。它也支持按位复合赋值操作符和按位移位操作符。

练习 4.15

下列 bitset 对象的声明哪些是错误的？

```
(a) bitset<64> bitvec(32);
(b) bitset<32> bv( 1010101 );
(c) string bstr; cin >> bstr; bitset<8>bv( bstr );
(d) bitset<32> bv; bitset<16> bv16( bv );
```

练习 4.16

下列 bitset 对象的用法哪些是错误的？

```
extern void bitstring(const char*);
bool bit_on(unsigned long, int );
bitset<32> bitvec;

(a) bitstring( bitvec.to_string().c_str() );
(b) if ( bit_on( bitvec.to_long(), 64 )) ...
(c) bitvec.flip( bitvec.count() );
```

练习 4.17

已知考虑序列 1，2，3，5，8，13，21。怎样初始化一个 bitset<32>来表示这个序列？已知一个空 bitset，写一个小程序来设置每一个合适的位。

4.13 优先级

操作符优先级是指复合表达式中操作符计算的顺序。例如，在下面的定义中，最终被赋给 ival 的是什么？

```
int ival = 6 + 3 * 4 / 2 + 2;
```

纯粹从左到右的计算结果为 20，其他可能的结果包括 9、14 和 36。哪一个是实际被赋给 ival 的值呢？14。

在 C++中，乘法和除法的优先级比加法高。这意味着它们先被计算。但乘法和除法的优先级相同，所以它们将按从左至右的顺序被计算。同此，表达式的计算顺序如下：

```
1. 3 * 4 => 12
```

```
2. 12 / 2 => 6
3. 6 + 6 => 12
4. 12 + 2 => 14
```

下面的 while 循环条件测试的行为与程序员的意图完全不同，因为与不等于操作符相比赋位操作符的优先级比较低：

```
while ( ch = nextChar() != '\n' )
```

编程者的意图是将下一个字符赋给的 ch，然后测试该字符是否为‘\n’。然而表达式的行为却是测试下一个字符是否为‘\n’，然后再把测试的结果 true 或 false 赋给 ch，而不会把下一个字符赋给 ch。

用括号把一些子表达式括起来，可以改变优先级。在复合表达式计算中，第一个动作是计算所有括号中的子表达式，再用计算的结果代替每个子表达式，然后继续计算。里边的括号比外面的括号先计算。例如：

```
4 * 5 + 7 * 2 ==> 34
4 * ( 5 + 7 * 2 ) ==> 76
4 * ( (5 + 7) * 2 ) ==> 96
```

下面的 while 循环用括号正确地把赋值子表达式括起来，正好反映了编程者的意图：

```
while ( (ch = nextChar()) != '\n' )
```

操作符具有优先级和结合性。例如，赋值操作符是右结合的。被连接起来的赋值表达式：

```
ival = jval = kval = lval //  右结合的
```

先把 lval 赋值给 kval，然后再把结果赋值给 jval，最后把结果赋值给 ival。另一方面，算术操作符是左结合的。表达式：

```
ival + jval + kval + lval // 左结合的
```

先把 ival 和 jval 相加，然后再加上 kval，最后加上 lval。

表 4.5 按照优先级顺序给出了 C++操作符的全集。表中每一段的操作符的优先级都相同。每一段中的操作符的优先级高于下一段中的操作符。例如，乘、除操作符的优先级相同，它们都比关系操作符的优先级高。

练习 4.18

参照表 4.5，指出下列复合表达式的计算顺序。

```
(a) ! ptr == ptr->next
(b) ~ uc ^ 0377 & ui ? 4
(c) ch = buf[ bp++ ] != '\n'
```

练习 4.19

练习 4.18 中的三个表达式的计算顺序与程序员的意图相反，把它加上括号使其符合你想像中的程序员意图。

练习 4.20

由于操作符优先级的问题，下面两个表达式编译失败。请参照表 4.5 解释原因，应该怎样改正呢？

```
(a) int i = doSomething(), 0;
(b) cout ? ival % 2 ? "odd" : "even";
```

表格 4.5 操作符优先级

操作符	功能	用法
::	全局域	::name
::	类域	::name
::	名字空间域	namespace::name
.	成员选择	object.member
->	成员选择	pointer->member
[]	下标	variable[expr]
()	函数调用	name(expr_list)
()	类型构造	type(expr_list)
++	后置递增	lvalue++
--	后置递减	lvalue--
typeid	类型 ID	typeid(type)
typeid	运行时刻类型 ID	typeid(expr)
const_cast	类型转换	const_cast<type>(expr)
dynamic_cast	类型转换	dynamic_cast<type>(expr)
reinterpret_cast	类型转换	reinterpret_cast<type>(expr)
static_cast	类型转换	static_cast<type>(expr)
sizeof	对象的大小	sizeof object
sizeof	类型的大小	sizeof(type)
++	前置递增	++lvalue
--	前置递减	--lvalue
~	按位非	~expr
!	逻辑非	!expr
-	一元减	-expr
+	一元加	+expr

续表

操作符	功能	用法
*	解引用	&expr
&	取地址	&expr
()	类型转换	(type)expr
new	分配对象	new type
new	分配/初始化对象	new type(expr_list)
new	分配/替换对象	new(expr_list)type(expr_list)
new	分配数组	所有的形式
delete	释放对象	所有的形式
delete	释放数组	所有的形式
->*	指向成员选择	pointer->*pointer_to_member
.*	指向成员选择	object.*pointer_to_member
*	乘	expr * expr
/	除	expr / expr
%	取模（求余）	expr % expr
+	加	expr + expr
-	减	expr - expr
<<	按位左移	expr << expr
>>	按位右移	expr >> expr
<	小于	expr < expr
<=	小于等于	expr <= expr
>	大于	expr > expr
>=	大于等于	expr >= expr
=	等于	expr == expr
!=	不等于	expr != expr
&	按位与	expr & expr
^	按位异或	expr ^ expr
|	按位或	expr | expr
&&	逻辑与	expr && expr
||	逻辑或	expr || expr

续表

操作符	功能	用法
?:	条件表达式	expr ? expr : expr
=	赋值	lvalue = expr
=,*=,/=,%=,+=,-=,<<=,>>=,&=,|=,^=	复合赋值	lvalue += expr 等
throw	抛出异常	throw expr
,	逗号	expr, expr

4.14 类型转换

考虑下列赋值：

```
int ival = 0;

// 编译器往往会给出警告
ival = 3.541 + 3;
```

最终结果是，ival 的值为 6。完成赋值的实际步骤如下面所述。我们首先要把两个不同类型的值相加，这里 3.541 是 double 型的文字常量，3 是 int 型的文字常量。C++并不是把两个不同类型的值加在一起，而是提供了一组算术转换（arithmetic conversions），以便在执行算术运算前，将两个操作数转换成共同的类型。转换规则是，小类型总是被提升成大类型，以防止精度损失。本例中，在执行加法前，整数 3 被提升为 double 型。这些转换由编译器自动完成，无需程序员介入。因此，它们也被称为隐式类型转换（implicit type conversion）。

加法以及结果都是 double 型的，结果值为 6.541。下一步是把结果赋给 ival。如果赋位操作符的左右两边的类型不同，那么，有可能的话，右边操作数会被转换成左边的类型。在本例中，ival 的类型是 int，double 向 int 的转换自动按截取而不是舍入进行，小数部分被直接地抛弃。6.541 变成了 6，这就是赋给 ival 的值。因为从 double 到 int 的转换会引起精度损失，因此大多数编译器会给出一个警告。

因为从 double 到 int 的类型转换不支持舍入，所以我们需要自己写程序来实现。例如：

```
double dval = 8.6;
int ival = 5;
ival += dval + 0.5;    // 保证舍入
```

如果原意的话，我们可以通过指定显式类型转换（explicit type conversion）来禁止标准算术转换：

```
// 指示编译器把 double 转换成 int
```

```
ival = static_cast< int >( 3.541 ) + 3;
```

在本例中，我们显式地指示编译器将 double 型的值转换成 int 型，而不是遵循标准 C++ 算术转换。

在这一节中，我们将详细讨论隐式类型转换（如上面第一个例子，由编译器自动完成，无需编程者介入）以及显式类型转换（如上面第二个例子中，程序员通过应用强制类型转换，指示编译器把一个现有的类型转换成指定的第二种类型）。

4.14.1 隐式类型转换

C++定义了一组内置类型对象之间的标准转换，在必要时它们被编译器隐式地应用到对象上。隐式类型转换发生在下列这些典型的情况下：

- 在混合类型的算术表达式中。在这种情况下，最宽的数据类型成为目标转换类型。这也被称为算术转换（arithmetic conversion）。例如：

```
int ival = 3;
double dval = 3.14159;

// ival 被提升为 double 类型: 3.0
ival + dval;
```

- 用一种类型的表达式赋值给另一种类型的对象。在这种情况下，目标转换类型是被赋值对象的类型。例如，在下面第一个赋值中，文字常量 0 的类型是 int，它被转换成 int*型的指针，表示空地址。在第二个赋值中，double 型的值被截取成 int 型的值。

```
// 0 被转换成 int*类型的空指针值
int *pi = 0;

// dval 被截取为 int 值 3
ival = dval;
```

- 把一个表达式传递给一个函数调用，表达式的类型与形式参数的类型不相同。在这种情况下，目标转换类型是形式参数的类型。例如：

```
extern double sqrt( double );

// 2 被提升为 double 类型: 2.0
cout << "The square root of 2 is "
     << sqrt( 2 ) << endl;
```

- 从一个函数返回一个表达式，表达式的类型与返回类型不相同。在这种情况下，目标转换类型是函数的返回类型。例如：

```
double difference( int ival1, int ival2 )
{
     // 返回值被提升为 double 类型
     return ival1 - ival2;
}
```

4.14.2 算术转换

算术转换保证了二元操作符（如加法或乘法）的两个操作数被提升为共同的类型，然后再用它表示结果的类型。两个通用的指导原则如下：

1．为防止精度损失，如果必要的话，类型总是被提升为较宽的类型。

2．所有含有小于整型的有序类型的算术表达式。在计算之前，其类型都会被转换成整型。

规则的定义如下面所述，这些规则定义了一个类型转换层次结构。（我们从最宽的类型 long double 开始。）

如果一个操作数的类型是 long double，那么另一个操作数无论是什么类型，都将被转换成 long double。例如，在下面的表达式中，字符常量小写字母‘a’将被提升为 long double（它的 ASC 码值为 97），然后再被加到 long double 型的文字常量上：

```
3.14159L + 'a';
```

如果两个操作数都不是 long double 型，那么当其中一个操作数的类型是 double 型，则另一个就将被转换成 double 型。例如：

```
int ival;
float fval;
double dval;

// 在计算加法前，fval 和 ival 都被转换成 double
dval + fval + ival;
```

类似地，如果两个操作数都不是 double 型，而其中一个操作数是 float 型，则另一个被转换成 float 型。例如：

```
char cval;
int ival;
float fval;

// 在计算加法前，ival 和 cval 都被转换成 double
cval + fval + ival;
```

否则，因为两个操作数都不是三种浮点类型之一，它们一定是某种整值类型。在确定共同的目标提升类型之前，编译器将在所有小于 int 的整值类型上施加一个被称为整值提升（integral promotion）的过程。

在进行整值提升时，类型 char、signed char、unsigned char 和 short int 都被提升为类型 int。

如果机器上的 m 型足够表示所有 unsinned shoft 型的值（这通常发生在 short 用半个了表 i，而 int 用一个字表示的情况下），则 unsigned short int 也被转换成 int。否则，它会被提升为 unsigned int。

wchar_t 和枚举类型被提升为能够表示其底层类型（underlying type）所有值的最小整数类型。例如，已知如下枚举类型：

```
enum status { bad, ok };
```

相关联的值是 0 和 1。这两个值可以（但不是必须）存放在 char 类型的表示中。当这些值实际上被作为 char 类型来存储时，char 代表了枚举的底层类型。然后。status 的整值提升将它的底层类型转换为 int。

在下列表达式中：

```
char cval;
bool found;
enum mumble { m1, m2, m3 } mval;

unsigned long ulong;

cval + ulong; ulong + found; mval + ulong;
```

在确定两个操作数被提升的公共类型之前，cval、found 和 mval 都被提升为 int 类型。

一旦整值提升执行完毕，类型比较就又一次开始。如果一个操作数是 unsigned long 型，则第二个也被转换成 unsigned long 型。在上面的例子中，所有被加到 ulong 上的三个对象都被提升为 unsigned long 型。

如果两个操作数的类型都不是 unsigned long，而其中一个操作数是 long 型，则另一个也被转换成 long 型。例如：

```
char cval;
long lval;

// 在计算加法前，cval 和 1024 都被提升为 long 型
cval + 1024 + lval;
```

long 类型的一般转换有一个例外：如果一个操作数是 long 型。而另一个是 unsigned int 型，那么，只有机器上的 long 型足够长以便能够存放 unsigned int 的所有值时（一般来说，在 32 位操作系统中，long 型和 int 型都用一个字长来表示，所以不满足这里的假设条件），unsigned int 才会被转换为 long 型；否则两个操作数都被提升为 unsigned long 型。

若两个操作数都不是 long 型，而其中一个是 unsigned int 型，则另一个也被转换成 unsigned int 型。否则，两个操作数一定都是 int 型。

尽管算术转换的这些规则可能给你的困惑多于启发，但是，一般的思想是，尽可能地保留多类型表达式中涉及到的值的精度。这正是通过“把不同的类型提升到当前出现的最宽的类型”来实现的。

4.14.3 显式转换

显式转换也被称为强制类型转换（cast），包括下列命名的强制类型转换操作符：static_cast、dynamic_cast、const_cast 和 reinterpret_cast。虽然有时候确实需要强制类型转换，但是它们也是程序错误的源泉。通过使用它们，程序员关闭了 C++语言的类型检查设施。在了解怎样把一个值从一种类型强制转换成另一种类型之前，我们先来看一下何时需要这么做。

任何非 const 数据类型的指针都可以被赋值给 void*型的指针。void*型指针被用于“对象

的确切类型未知”或者“在特定环境下对象的类型会发生变化”的情况下。有时 void*型的指针被称为泛型（generic）指针，因为它可以指向任意数据类型的指针。例如：

```
int ival;
int *pi = 0;
char *pc = 0;
void *pv;
pv = pi; // ok: 隐式转换
pv = pc; // ok: 隐式转换

const int *pci = &ival;
pv = pci; // 错误: pv 不是一个 const void*.
const void *pcv = pci; // ok
```

但是，void*型指针不能直接被解除引用，因为没有类型信息可用来指导编译器怎样解释底层的位模式。相反，void*的指针必须先被转换成某种特定类型的指针。但是，在 C++中，不存在从 void*型指针到特殊类型的指针之间的自动转换。例如，考虑下列代码：

```
#include <cstring>

int ival = 1024;

void *pv;
int *pi = &ival;
const char *pc = "a casting call";

void mumble()
{
    pv = pi; // ok: pv 指向 ival
    pc = pv; // 错误: 没有标准的转换

    char *pstr = new char[ strlen( pc )+1 ];
    strcpy( pstr, pc );
}
```

在这种情况下，程序员在把 pv 赋给 pc 时显然犯了错误，因为 pv 指向一个整数而不是一个字符数组。随后，当 pc 被传递给函数 strlen()时，由于函数需要一个以空字符结尾的字符串，因而导致程序出现了严重错误。而在执行 strcpy()时，对于这个程序，我们希望最好的结果也就是程序异常终止了。很容易看出是什么使这个错误难以修正，这就是为什么在“把 void*型的指针赋值给任意显式类型”时，C++要求显式强制转换的原因：

```
void mumble()
{
    // ok: 仍然是错误的，但是现在可以通过编译!
    // 因为在赋值前用了显式强制转换
    // 当程序失败时，应该首先检查强制转换…
    pc = static_cast< char* >( pv );
```

```
        // 仍然是一个灾难
        char *pstr = new char[ strlen( pc )+1 ];
        strcpy( pstr, pc );
    }
```

执行显式强制转换的第二个原因是希望改变通常的标准转换。例如，下列复合赋值，首先将 ival 提升成 double 型，然后再把它加到 dval 上，最后把结果截取成 int 型来执行赋值：

```
double dval;
int ival;

ival += dval;
```

我们通过显式地将 dval 强制转换成 int 型，消除了把 ival 从 int 型到 double 型的不必要提升：

```
ival += static_cast< int >( dval );
```

进行显式强制转换的第三个原因是要避免出现多种转换可能的歧义情况。我们将在第 9 章对重载函数名的讨论中更仔细地了解这种情况。

显式转换符号的一般形式如下：

```
cast-name< type >( expression );
```

这里的 cast-name 是 static_cast、const_cast。dynamic_cast 和 reinterpret_cast 之一。const_cast，正如其名字所暗示的，将转换掉表达式的常量性（以及 volatile 对象的 volatile 性）。例如：

```
extern char *string_copy( char* );
const char *pc_str;

char *pc = string_copy( const_cast< char* >( pc_str ));
```

试图用其他三种形式来转换掉常量性会引起编译错误。类似地，用 const_cast 来执行一般的类型转换，也会引起编译错误。

编译器隐式执行的任何类型转换都可以由 static_cast 显式完成：

```
double d = 97.0;
char ch = static_cast< char >( d );
```

为什么要这样做呢？因为从一个较大类型到一个较小类型的赋值，会导致编译器产生一个警告以提醒我们潜在的精度损失。当我们提供显式强制转换时，警告消息被关闭。强制转换告诉编译器和程序的读者：我们不关心潜在的精度损失。

行为不佳的静态转换（即那些有潜在危险的类型转换）包括将 void*型的指针强制转换成某种显式指针类型、把一个算术值强制转换成枚举型，或把一个基类强制转换成其派生类或者这种类的指针或引用）。（基类与派生类的转换将在 19 章讨论。）

这些转换有潜在的危险是因为它们的正确性取决于在转换发生时该对象碰巧包含的值。例如，已知下列声明：

```
enum mumble { first = 1, second, third };

extern int ival;
mumble mums_the_word = static_cast< mumble >( ival );
```

只有当 ival 含有的值是 1、2 或 3 的时候，它到 mumble 型的转换才是正确的。

reinterpre_cast 通常对于操作数的位模式执行一个比较低层次的重新解释，它的正确性很大程度上依赖于程序员的主动管理。例如，下列转换：

```
complex<double> *pcom;
char *pc = reinterpret_cast< char* >( pcom );
```

程序员必须永远也不会失去对 pc 实际指向对象的监视。例如，把它传递给一个 string 对象，如：

```
string str( pc );
```

可能会引起 str 运行时的古怪行为。

这个例子很好地说明了“显式强制转换是多么危险”。由于显式的 reinterpret_cast，用复数对象的地址初始化 pc。，没有引起任何来自编译器的错误或警告信息。后面对 pc 的使用都把它当作 char*型对象，因此用 pc 初始化 str 是完全正确的。然而，当我们写如下语句时：

```
string str( pc );
```

程序的运行时行为就是未定义的。

寻找此类问题的原因非常困难，特别是，当 pcom 向 pc 的强制转换发生在与调用 strsize() 不同的文件中的时候［调用 str.size()时，期望的空字符结尾没有出现］。

在某种程度上，这说明了语言的矛盾性。强类型检查正是为了防止此类错误。然而，显式强制转换又允许我们暂时挂起强类型检查。当我们用 pc 初始化 str 时，类型检查机制又重新开始工作，并且 pc 确实是正确的类型：char*。但是，它其实并不正确。由于显式强制转换，编译器并不知道这些。

显然，禁止显式转换本身实际上是不可行的，而标准 C++引入的这些强制转换操作符又突出了这个矛盾。在引入这些强制转换操作符之前，显式强制转换由一对括号来完成（标准 C++仍然支持这种旧式的强制转换）：

```
char *pc = (char*) pcom;
```

效果与使用 reinterpret_cast 符号相同，但强制转换的可视性非常差。这使跟踪错误的转换更加困难。

C++提供了各种显式强制转换符号，而不是惟一一种符号，例如：

```
// 不是 C++
char *pc = explicit_cast< char* >( pcom );
```

结果是，程序员（以及读者和操作程序的工具）能够很清楚地知道代码中每个显式强制转换的潜在危险等级。

无论何时，当我们面对令人费解的运行时刻程序行为时，可能的罪魁祸首首先会是具有功能障碍的指针。一种原因就是无效的显式强制转换，因此，用 reinterpret_cast 操作符

来执行并标识出所有的显式指针强制转换是很有用的。(指针错误的第二个原因是它指向的内存变成了无效的。这可能会发生在“我们偶尔删除了一个仍然在被使用的指针”，或者“返回一个局部对象的地址”时。我们将在 8.4 节讨论动态内存分配时详细解释这个问题。)

dynamic_cast 支持在运行时刻识别由指针或引用指向的类对象。对 dynamic_cast 的讨论将在 19.1 节介绍运行时刻类型识别时再进行。

4.14.4 旧式强制类型转换

前面给出的强制转换符号语法，有时被称为新式强制转换符号，它是由标准 C++引入的。在它之前，显式强制转换由非常通用的强制转换语法（现在被称为旧式强制转换符号）来实现。虽然标准 C++仍然支持旧式强制转换符号，但是我们建议，只有当我们为 C 语言或标准 C++之前的编译器编写代码时才使用这种语法。

旧式强制转换符号有下列两种形式：

```
// C++强制转换符号
type (expr);

// C 语言强制转换符号
(type) expr;
```

旧式强制转换可以用来代替标准 C++中的 static_cast、cons_cast 或 reinterpret_cast。在标准 C++之前，我们只能使用旧式强制转换。如果我们希望自己的代码在 C++和 C 语言中都能够编译的话，那么只能使用 C 语言的强制转换符号。

以下是一些使用了旧式强制转换符号的例子：

```
const char *pc = (const char*) pcom;
int ival = (int) 3.14159;

extern char *rewrite_str( char* );
char *pc2 = rewrite_str( (char*) pc );
int addr_value = int( &ival );
```

对旧式强制转换符号的支持是为了对“在标准 C++之前写的程序”保持向后兼容性，以及提供与 C 语言兼容的符号。

练习 4.21

已知下列定义：

```
char cval; int ival;
float fval; double dval;
unsigned int ui;
```

指出可能发生的隐式类型转换：

```
(a) cval = 'a' + 3;
```

```
(b) fval = ui - ival * 1.0;
(c) dval = ui * fval;
(d) cval = ival + fval + dval;
```

练习 4.22

已知下列的定义：

```
void *pv; int ival;
char *pc; double dval;
const string *ps;
```

用强制转换符号重写下列每个语句：

```
(a) pv = (void*)ps;
(b) ival = int( *pc );
(c) pv = &dval;
(d) pc = (char*) pv;
```

4.15 栈类实例

在关于递增递减操作符的讨论结束时，我们介绍了栈（stack）的抽象来说明这些操作符的前置和后置格式。我们将用一个 iStack 类（即只支持 int 型元素的栈）的设计与实现的简要过程来结束本章。

栈是计算机科学的一个基本数据抽象，它允许以后进先出（LIFO）的顺序嵌入和获取其中的值。栈的两个基本操作是：向栈中压入（push）一个新值，以及弹出（pop）或获取最后压入的那个值。其他一些操作包括：查询栈是否满 full()或空 empty()，以及判断栈的长度 size()——即包含多少个元素。我们的初始实现只支持 int 型的元素。下面是其公有接口的声明：

```
#include <vector>

class iStack {
public:
     iStack( int capacity )
          : _stack( capacity ), _top( 0 ) {}

     bool pop( int &value );
     bool push( int value );

     bool full();
     bool empty();
     void display();

     int size();
```

```
private:
    int _top;
    vector< int > _stack;
};
```

为了演示递增递减操作符的前置和后置形式的用法，我们为 iStack 栈选择了固定长度的实现（我们将在第 6 章结尾时将其修改为可动态增长的）。我们把元素存储在一个 int 型的 vector 中，它的名字为_stack。_top 含有下一个可用槽的值，push()操作会向该槽压入一个值。_top 的当前值反映了栈中元素的个数。同此，size()只需简单地返回_top：

```
inline int iStack::size() { return _top; };
```

如果_top 等于 0，则 empty()返回 true；如果_top 等于_stack.size()，则 full()返回 true：

```
inline bool iStack::empty() {
    return _top ? false : true; }

inline bool iStack::full() {
    return _top < _stack.size()-1 ? false : true;
}
```

下面是如 pop()和 push()的实现代码。我们加入了输出函数来跟踪它们的执行情况：

```
bool iStack::pop( int &top_value )
{
    if ( empty() )
        return false;

    top_value = _stack[ --_top ];
    cout << "iStack::pop(): " << top_value << endl;

    return true;
}
bool iStack::push( int value )
{
    cout << "iStack::push( " << value << " )\n";

    if ( full() )
        return false;

    _stack[ _top++ ] = value;
    return true;
}
```

在练习 iStack 类之前，我们先来加入一个 display()函数，它允许我们查看栈的内容。给定一个空栈，它的输出如下：

```
( 0 )
```

对一个有 4 个元素 0、1、2 和 3 的栈，它产生如下输出：

```
( 4 ) ( bot: 0 1 2 3 :top )
```

下面是我们的实现：

```
void iStack::display()
{
    if ( !size() )
        { cout << "( 0 )\n"; return; }
    cout << "( " << size() << " )( bot: ";
    for ( int ix = 0; ix < _top; ++ix )
        cout << _stack[ ix ] << " ";
    cout << " :top )\n";
}
```

下面的小程序使用了我们的类。for 循环迭代 50 次。它把每个偶数值 2、4、6、8 等等压入栈中。当这些值是 5 的倍数时；比如 5、10、15 等等，就显示栈中的内容。当值是 10 的倍数时，如 10、20、30 等等，它从栈中弹出最后两项，然后再次显示栈中的内容：

```
#include <iostream>
#include "iStack.h"

int main() {
    iStack stack( 32 );

    stack.display();
    for ( int ix = 1; ix < 51; ++ix )
    {
        if ( ix%2 == 0 )
            stack.push( ix );

        if ( ix%5 == 0 )
            stack.display();

        if ( ix%10 == 0) {
            int dummy;
            stack.pop( dummy ); stack.pop( dummy );
            stack.display();
        }
    }
}
```

编译并运行程序，产生下面的输出：

```
( 0 )( bot: :top )
iStack::push( 2 )
iStack::push( 4 )
( 2 )( bot: 2 4 :top )
iStack::push( 6 )
```

```
iStack::push( 8 )
iStack::push( 10 )
( 5 )( bot: 2 4 6 8 10 :top )
iStack::pop(): 10
iStack::pop(): 8
( 3 )( bot: 2 4 6 :top )
iStack::push( 12 )
iStack::push( 14 )
( 5 )( bot: 2 4 6 12 14 :top )
iStack::push( 16 )
iStack::push( 18 )
iStack::push( 20 )
( 8 )( bot: 2 4 6 12 14 16 18 20 :top )
iStack::pop(): 20
iStack::pop(): 18
( 6 )( bot: 2 4 6 12 14 16 :top )
iStack::push( 22 )
iStack::push( 24 )
( 8 )( bot: 2 4 6 12 14 16 22 24 :top )
iStack::push( 26 )
iStack::push( 28 )
iStack::push( 30 )
( 11 )( bot: 2 4 6 12 14 16 22 24 26 28 30 :top )
iStack::pop(): 30
iStack::pop(): 28
( 9 )( bot: 2 4 6 12 14 16 22 24 26 :top )
iStack::push( 32 )
iStack::push( 34 )
( 11 )( bot: 2 4 6 12 14 16 22 24 26 32 34 :top )
iStack::push( 36 )
iStack::push( 38 )
iStack::push( 40 )
( 14 )( bot: 2 4 6 12 14 16 22 24 26 32 34 36 38 40 :top )
iStack::pop(): 40
iStack::pop(): 38
( 12 )( bot: 2 4 6 12 14 16 22 24 26 32 34 36 :top )
iStack::push( 42 )
iStack::push( 44 )
( 14 )( bot: 2 4 6 12 14 16 22 24 26 32 34 36 42 44 :top )
iStack::push( 46 )
iStack::push( 48 )
iStack::push( 50 )
( 17 )( bot: 2 4 6 12 14 16 22 24 26 32 34 36 42 44 46 48 50 :top )
iStack::pop(): 50
iStack::pop(): 48
( 15 )( bot: 2 4 6 12 14 16 22 24 26 32 34 36 42 44 46 :top )
```

练习 4.23

某些用户需要 peek()操作，以读出栈顶值，但并不把它从栈中移出，当然，条件是栈不为空。请提供 peek()的实现。然后再在前面给出的 main()程序中加入语句，来练习 Peek()操作。

练习 4.24

我们的 iStack 设计的两个主要弱点是什么？怎样修正？

5

语　　句

程序最小的独立单元是语句（statement）。在自然语言中，类似的结构是句子。与句子由句号结束一样，语句一般由分号结束。因此，一个表达式如 ival+5，给它加个分号，就变成了一个简单语句（simple statement）。复合语句（compound statement）是由一对花括号包围起来的一系列简单语句。在缺省情况下，语句以其出现的顺序执行。但是，除了最简单的程序外，顺序的程序执行过程对于我们必须要解决的问题来说是不够的。根据一个表达式的计算结果是 true 还是 false，特殊的控制流（flow-of-control）程序语句允许有条件地或重复地执行一个简单语句或复合语句。条件执行过程由 if、if-else 和 switch 语句支持，而重复执行过程由 while、do-while 和 for 语句支持。后三种语句通常也被称为循环（loop ）语句、本章将详细讨论 C++支持的程序语句类型。

5.1 简单语句和复合语句

程序语句最简单的形式是空语句，形式如下（仅一个分号）：

```
;     // 空语句
```

空语句被用在“程序的语法上要求一条语句，而逻辑上却不需要”的时候。例如，在下面的 while 循环中，把一个 C 风格字符串拷贝到另一个字符串中去所需的全部处理过程，在这个语句的被称为**条件**（condition）的部分就已经完成了。但是，while 循环的格式要求条件后面跟一条语句。因为不需要再做其他的工作，所以我们用一条空语句来满足这个语法要求：

```
while ( *string++ = *inBuf++ )
      ;    // 空语句
```

意外出现的多余空语句不会产生编译错误。例如，下面的语句：

```
ival = dval +sval;;    // ok: 多余的空语句
```

由两条语句构成：向 ival 赋值的表达式语句以及空语句。

简单语句由单个语句构成。例如：

```
//  简单语句
```

```
int ival = 1024;          // 声明语句
ival;                     // 表达式语句
ival + 5;                 // 另一个表达式语句
ival = ival + 5;          // 赋值语句
```

条件和循环语句在语法上只允许执行一条指定的相关语句。然而在实践中，这是远远不够的。在逻辑上，程序经常需要执行两条或多条语句构成的序列。在这样的情况下，我们用一个**复合语句**（compound）来代替单个语句。例如：

```
if ( ival0 > ival1 )
{
     // 由一条声明和两条赋值语句构成的复合语句
     int temp = ival0;
     ival0 = ival1;
     ival1 = temp;
}
```

复合语句是由一对花括号括起来的语句序列。复合语句被视为一个独立的单元，它可以出现在程序中任何单个语句可以出现的地方。复合语句不需要用分号作为结束，这是一种附加的语法上的便利。

空复合语句与空语句等价，它为空语句提供了一种替代语法，例如：

```
while ( *string++ = *inBuf++ )
     { } // 等价于空语句
```

包含一条或多条声明语句的复合语句，如前面的例子，也称为块（block）或语句块（statement block）。块引入了程序中的局部域，在块中声明的标识符，如前面例子中的 temp，只在该块中可见。块、域以及对象的生命期将在第 8 章中详细讨论。

5.2 声明语句

在 C++中，对象的定义，如：

```
int ival;
```

被视为 C++语言的一条语句［称作**声明语句**（declaration statement），尽管在这种情况下称为定义（definition）语句更准确］，一般它可以被放在程序中任何允许语句出现的地方。例如，考虑下面的程序（声明语句用// #n 编号，这里 n 从 1 开始计数）：

```
#include <stream>
#include <string>
#include <vector>

int main()
{
     string fileName; // #1

     cout << "Please enter name of file to open: ";
     cin >> fileName;
```

```
        if ( fileName.empty() ) {
             // 很好，但有一点要说明
             cerr << "fileName is empty(). bailing out. bye!\n";
             return -1;
        }
        ifstream inFile( fileName.c_str() ); // #2
        if ( ! inFile ) {
             cerr << "unable to open file. bailing out. bye!\n";
             return -2;
        }
        string inBuf; // #3
        vector< string > text;  // #4
        while ( inFile >> inBuf ) {
             for ( int ix = 0; ix < inBuf.size(); ++ix )    // #5
                  // 这里 ch 并不必需，
                  // 但有利于说明问题
                  if (( char ch = inBuf[ix] ) == '.' ) {    // #6
                       ch = '_';
                       inBuf[ix] = ch;
                  }
             text.push_back( inBuf );
        }
        if ( text.empty() )
             return 0;

        // 一条声明语句，有两个定义
        vector<string>::iterator iter = text.begin(),  // #7
        iend = text.end();
        while ( iter != iend ) {
             cout << *iter << '\n';
             ++iter;
        }
        return 0;
   }
```

程序包含七条声明语句和八个对象定义。声明语句展示了声明的**局部性**（locality of declaration）——即，声明语句出现在被定义对象首次被使用的局部域内。

在 70 年代，计算机程序语言设计哲学强调这样一种美德：在程序、函数或语句块的开始处、并且在其他程序语句之前定义全部对象。（例如，在 C 中，对象的定义并不被视为 C 语言的语句，块中的所有对象定义必须出现在任何程序语句之前。出于这种需要，C 程序员使自己习惯于在每个当前块的顶部定义全部对象。）在某种程度上，这是 FORTRAN 支持的动态对象定义容易出错的一种回应措施。

由于对象的定义是 C++语言的一条语句，所以可以将对象定义放在任何其他语句能够出现的地方。从语法上讲，这也是使声明的局部件成为可能的原因。

这是必需的吗？对于内置类型，如整型和浮点型，声明的局部性主要是个人的喜好问题。C++允许在 if、else-if、switch、while 和 for 循环的条件部分出现声明（前面给出的程序中有两个这样的例子），以此来鼓励使用局部声明。喜欢声明局部性的人相信它使程序更易于理

解。

对于类对象的定义来说，由于类对象与构造函数和析构函数相关联，所以声明的局部性就变成必需的了。当把这些对象放在函数或语句块的开始时，发生下面两件事情：

1．在做函数或语句块中的任何事情之前，所有类对象的构造函数均被调用。声明的局部性使我们能够把初始化的开销分摊到函数或语句块中。

2．或许更重要的是，通常情况下，在函数或语句块内部的所有程序语句都被执行之前，该函数或者语句块就结束了。例如，前面的程序显示了两个非正常终止点：获取文件名失败和打开用户指定的文件失败。在成功地经过这些终止点之前定义类对象（如 inBuf 和 text）会导致执行不必要的构造函数——析构函数对。如果给出足够多的类对象或者需要大量计算的构造函数和析构函数，这将对程序的运行效率产生不必要的影响。虽然结果仍然是正确的，但是有时候性能可能会变得不可接受。（专业的 C 程序员习惯把对象定义放在函数或语句块开始的地方。有时候他们会发现自己的 C++程序比等价的 C 程序性能低得多，这就是原因所在。）

一条声明语句可以由一个或多个对象定义构成。例如，在我们的程序中，在同一条声明语句中定义了两个向量迭代器：

```
// 一条声明语句，两个对象定义
vector<string>::iterator iter = text.begin(),
                         iend = text.end();
```

它与下面一对声明是等价的：

```
// 等价于两条声明语句
vector<string>::iterator iter = text.begin();
vector<string>::iterator iend = text.end();
```

虽然说，选择哪一种声明方案是个人喜好问题，但是，当我们把对象、指针以及引用混合在一起时，在一条声明语句中放置多个对象定义更容易出错。例如，下列声明语句就很不清楚，用户是想定义一个指针和一个对象，还是把第二个指针错写成了对象（标识符名暗示第二个定义是错误的）：

```
// 符合程序员的意图吗?
string *ptr1, ptr2;
```

在这种情况下，独立的声明语句几乎没有为错误留下空间：

```
string *ptr1;
string *ptr2;
```

在我们自己的代码中，我们倾向于根据对象的用法将其分组。例如，在下面两个声明语句中：

```
int aCnt=0, eCnt=0, iCnt=0, oCnt=0, uCnt=0;

int charCnt=0, wordCnt=0;
```

那些被用作记录五个英语元音字母个数的对象被放在一条声明语句中，记录全部字符个数和单词个数的对象则被放在第二条声明语句中。虽然这种方法在我们看起来非常明智，但是逻辑上我们不能认为它更正确或者更合适。

练习 5.1

假设你在领导一个小的程序项目组，你希望所有代码都遵循统一的声明规则。请明确定义你希望项目组遵循的声明规则，并说明理由。

练习 5.2

假设你被分到了练习 5.1 的项目组。你对于已经宣布的声明策略并不赞同，而且你也不同意任何一条声明策略。请明确说明并证明你的理由。

5.3 if 语句

C++语言提供 if 语句的动机是：根据指定的表达式是否为 true，有条件地执行一条语句或语句块。if 语句的语法形式如下：

```
if ( condition )
     statement
```

condition（条件）必须被放在括号内。它可以是表达式，如：

```
if ( a + b > c ) { ... }
```

或是一条具有初始化功能的声明语句，如：

```
if ( int ival = compute_value() ) { ... }
```

在 condition 中定义的对象，只在与 if 相关的语句或语句块中可见。例如，试图在 if 语句后面访问 ival 会导致编译错误：

```
if ( int ival = compute_value() )
{
     // ival 只在这个 if 语句块中可见
}
// 错误: ival 不可见
if ( ! ival ) ...
```

为了说明 if 语句的用法，让我们来实现一个函数 min()，它返回一个 int 型 vector 中的最小值。另外，它还记录了该最小值在 vector 中出现的次数。对于 vector 中的每个元素，我们需分做以下几件事情：

- 把元素同当前最小值作比较；
- 如果它小于最小值，则把它赋给最小值，将计数器复位为 1；
- 如果它等于最小值，把计数器加 1；
- 否则，什么都不做；
- 检查完所有元素后，将最小值及其计数器返回给用户。

这需要两条 if 语句：

```
if ( minVal > ivec[ i ] ) ... // 新的 minVal
if ( minVal == ivec[ i ] ) ... // 又一次出现
```

在使用 if 语句时，一个比较普遍的错误是：当条件为 true、并且必须执行多条语句时，

我们往往忘记了提供复合语句。找到这样的错误比较困难，因为程序代码看起来是正确的。例如：

```
if ( minVal > ivec[ i ] )
     minVal = ivec[ i ];
     occurs = 1; // 不是 if 语句的组成部分
```

它的意思与程序员的意图以及程序的缩进相反：

```
occurs = 1;
```

没有被当作 if 语句的一部分。因此，最小值的出现计数总是 1。下面是按程序员意图写的 if 语句（左花括号的确切位置是一个无休止的争论话题）：

```
if ( minVal > ivec[ i ] )
{
     minVal = ivec[ i ];
     occurs = 1;
}
```

第二个 if 语句如下：

```
if ( minVal == ivec[ i ] )
     ++occurs;
```

注意，if 语句的顺序很重要。如果按如下顺序放置语句，则函数的结果总是差 1：

```
if ( minVal > ivec[ i ] ) {
     minVal = ivec[ i ];
     occurs = 1;
}

// 如果 minVal 正好已经被设置为 ivec[i],
// 则存在潜在的错误
if ( minVal == ivec[ i ] )
     ++occurs;
```

针对同一个值的两个 if 语句执行起来不但危险，而且也没有必要。同一个元素不能既等于 minVal，同时又小于 minVal。如果一个条件为 true，则另一个就可以被安全地忽略了。if 语句通过提供 else 子句以便允许这种“如果…否则…”条件的情况。

if-else 的语法形式如下：

```
if ( condition )
     statement1
else
     statement2
```

如果 condition 为 true，则 statement1（语句 1）被执行；否则执行 statement2（语句 2）。例如：

```
if ( minVal == ivec[ i ] )
     ++occurs;
else
     if ( minVal > ivec[ i ] ) {
          minVal = ivec[ i ];
          occurs = 1;
     }
```

在本例中，statement2 本身又是一个 if 语句。如果 minVal 小于这个元素，则什么也不做。在下面的例子中，将执行三个语句之一：

```
if ( minVal < ivec[ i ] )
     {} // 空语句
else
     if ( minVal > ivec[ i ] ) {
          minVal = ivec[ i ];
          occurs = 1;
     }
else // minVal == ivec[ i ]
     ++occurs;
```

if-else 语句引入了一种二义性问题，称为**空悬 else**（dangling-else）问题。这种问题出现在当 if 子句多于 else 子句时。问题是：这些 else 子句分别和哪一个 if 子句匹配？例如：

```
if ( minVal <= ivec[ i ] )
     if ( minVal == ivec[ i ] )
          ++occurs;
else {
     minVal = ivec[ i ];
     occurs = 1;
}
```

程序的缩进形式表明程序员相信 else 应该与最外面的 if 子句匹配。然而在 C++中，空悬 else 二义性由以下规定来解决：else 子句与“最后出现的未被匹配的 if 子句”相匹配。在本例中，if-else 语句实际的计算过程如下：

```
if ( minVal <= ivec[ i ] ) {
     // 空悬 else 的解释结果
     if ( minVal == ivec[ i ] )
          ++occurs;
     else { minVal = ivec[ i ]; occurs = 1; }
```

要想改变这种缺省的空悬 else 匹配效果，一种方法是把后来出现的 if 放在复合语句中：

```
if ( minVal <= ivec[ i ] ) {
     if ( minVal == ivec[ i ] )
          ++occurs;
}
else { minVal = ivec[ i ]; occurs = 1; }
```

有些编码风格建议总是使用复合语句括号，以避免在以后修改代码时可能出现的混淆或错误。

下面是我们的 min()函数的第一次迭代。第二个参数 occurs 将记录最小值出现的次数。我们将在函数中设置它，同时确定并返回实际的最小值。我们用 for 循环来迭代全部元素。不幸的是，我们的实现中包含了一个逻辑错误，你能看出来吗？）

```
#include <vector>
int min( const vector<int> &ivec, int &occurs )
{
     int minVal = 0;
     occurs = 0;
```

```
    int size = ivec.size();

    for ( int ix = 0; ix < size; ++ix ) {
        if ( minVal == ivec[ ix ] )
            ++occurs;
        else
            if ( minVal > ivec[ ix ] ) {
                minVal = ivec[ ix ];
                occurs = 1;
            }
    }

    return minVal;
}
```

一般地，函数只返回一个值。但是我们需要返回 vector 中包含的最小值以及它在 vector 中出现的次数。在我们的实现中，增加了一个引用参数，通过它来传送第二个值（见 7.3 节关于引用参数的讨论）。在 min()中，任何针对 occurs 的赋值都作用在作为参数传递进来的实际对象上。例如：

```
int main()
{
    int occur_cnt = 0;
    vector< int > ivec;

    // ... 填充 ivec
    // occur_cnt 存有在 min()中设置的出现次数值
    int minval = min( ivec, occur_cnt );

    // ...
}
```

一种替代解决方案是用一个 pair 对象（见 3.14 节关于 pair 类型的讨论）。它拥有两个整型对象：最小值和出现次数。然后函数返回这种 pair 对象的一个实例。例如：

```
// 另外一种实现代码
// 返回一个 pair 对象…
#include <utility>
#include <vector>

typedef pair<int,int> min_val_pair;

min_val_pair
min( const vector<int> &ivec )
{
    int minVal = 0;
    int occurs = 0;

    // 同上，直到 return
    return make_pair( minVal, occurs );
}
```

不幸的是，在这两种方案中，我们的 min()函数实现都是不正确的。你能看出问题来吗？对，因为我们把 minVal 初始化为 0，如果数组的最小值大于 0，则我们的实现就不能找到它，只能返回 0，同时 occurs 被设置为 0。

minVal 的最好初值是数组的第一个元素：

```
int minVal = ivec[0];
```

它保证总是返回数组中的最小值。虽然这样做修正了程序中的 bug，但是，它也引入了一点小小的性能损失。下面是代码中不合适的部分，你能看出这点小小的性能代价是什么吗？

```
// min()函数被修改后的开始部分
// 不幸的是，它引入了一点小小的性能损失
int minVal = ivec[0];
occurs = 0;
int size = ivec.size();

for ( int ix = 0; ix < size; ++ix )
{
     if ( minVal == ivec[ ix ] )
          ++occurs;
     // ...
```

因为 ix 被初始化为 0，所以循环的第一次迭代总是发现 minVal 等于 ivec[0]，因为我们正是用它来初始化 minVal 的。通过将 ix 初始化为 1，可以避免不必要的比较操作以及对 minVal 的再次赋值。这显然是个改善，但不幸的是，这又向程序引入了另外一个错误 （或许早就应该让事情恢复它本来的面貌）。你能看出修改后的程序有什么问题吗？

```
// 修改后的 min()的开始部分
// 不幸的是，它引入了一个错误
int minVal = ivec[0];
occurs = 0;
int size = ivec.size();

for ( int ix = 1; ix < size; ++ix )
{
     if ( minVal == ivec[ ix ] )
          ++occurs;
     // ...
```

如果 ivec[0]是最小值，则 occurs 永远不会被设为 1。当然，这很容易改正，而且，以后我们会看到这样做完全有必要：

```
int minVal = ivec[0];

occurs = 1;
```

不幸的是，这还不是完全正确的。如果用户偶然传递了一个空的 vector，又会产生什么后果？试图访问空 vector 的第一个元素是不正确的，有可能导致运行时刻程序错误，我们必须对这种可能性提供保护。一种方案如下（另外一种可能的替代方案是“返回布尔值表明函

数是否成功”）：

```
int min( const vector<int> &ivec, int &occurs )
{
    int size = ivec.size();

    // 处理空 vector 异常
    // occurs 被设置为 0 表示空 vector
    if ( ! size ) { occurs = 0; return 0; }

    // ok: vector 至少含有一个元素
    int minVal = ivec[ 0 ]; occurs = 1;
    for ( int ix = 1; ix < size; ++ix )
    {
        if ( minVal == ivec[ ix ] )
            ++occurs;
        else
            if ( minVal > ivec[ ix ] ){
                minVal = ivec[ ix ];
                occurs = 1;
            }
    }
    return minVal;
}
```

空 vector 问题的另一种替代方案是，让 min()函数在通过引用返回最小值的同时，返回一个布尔值来表明失败还是成功：

```
// 空 vector 问题的另一种方案
bool min( const vector< int > &ivec, int &minVal, int &occurs );
```

另一种设计选择是让 min()函数在接收到空 vector 时，抛出一个异常。（见第 11 章关于异常处理的讨论。）

不幸的是，类似这样的错误非常普遍。作为程序员，我们将会在某个地方犯错误——有时甚至是很愚蠢的错误。如果错误发生了，最重要的是接受发生错误的事实，并保持警惕性，在条件允许的情况下尽可能严格地测试和检查代码。

对于简单的 if-else 语句，条件操作符可以提供一个便捷的简写形式。例如，下列 min()函数模板：

```
template <class valueType>
inline const valueType&

min( valueType &val1, valueType &val2 )
{
    if ( val1 < val2 )
        return val1;
    return val2;
}
```

可以写成：

```
template <class valueType>
inline const valueType&
```

```
min( valueType &val1, valueType &val2 )
{
     return ( val1 < val2 ) << val1 : val2;
}
```

连接成一长串的 if-else 语句，比如下列语句。修改的时候我们很难读懂而且容易出错：

```
if ( ch == 'a' ||
     ch == 'A' )
     ++aCnt;
else
if ( ch == 'e' ||
     ch == 'E' )
     ++eCnt;
else
if ( ch == 'i' ||
     ch == 'I' )
     ++iCnt;
else
if ( ch == 'o' ||
     ch == 'O' )
     ++oCnt;
else
if ( ch == 'u' ||
     ch == 'U' )
     ++uCnt;
```

这种 if-else 语句串的替代结构是 switch 语句。只要与测试量作比较的值是常量表达式，比如上面代码中被测试的字符常量，我们就可以用 switch 语句来替代。switch 语句是下一节的话题。

练习 5.3

改正下列代码：

```
(a) if ( ival1 != ival2 )
     ival1 = ival2
   else ival1 = ival2 = 0;
(b) if ( ival < minval )
     minval = ival;
   occurs = 1;
(c) if ( int ival = get_value())
     cout << "ival = "
          << ival << endl;
   if ( ! ival )
     cout << "ival = 0\n";
(d) if ( ival = 0 )
     ival = get_value();
(e) if ( ival == 0 )
   else ival = 0;
```

练习 5.4

把 min()函数参数表中的 occurs 声明改成非引用参数类型，并重新运行程序。程序的行为如何变化？

5.4 switch 语句

深层嵌套的 if-else 语句常常在语法上是正确的，但逻辑上却没有正确地表达程序员的意图。例如，意料之外的 else-if 更可能不会注意到而被溜过去，修改这些语句非常困难。C++提供了 switch 语句，作为一种“在一组互斥的项目中做选择”的替代方法。

为了解释 switch 语句的用法，我们考虑下面的问题：要求记录每个元音字母在随机的文本段中出现的次数（习惯上认为英语中元音 e 出现的次数最多）。程序逻辑如下：

- 按顺序读取每个字符直到没有字符为止；
- 把每个字符同元音字母集合作比较；
- 如果字符同某个元音字母匹配，则该元音计数加 1；
- 显示结果。

用此程序分析本书英文版的第一节。下面是输出，它证实了习惯的看法，元音 e 的出现频率最高：

```
aCnt: 394
eCnt: 721
iCnt: 461
oCnt: 349
uCnt: 186
```

switch 语句由以下部分构成：

1．关键字 switch，后面是一个要被计算的表达式，表达式被放在括号中。在本例中，表达式是被读取的字符。例如：

```
char ch;
while ( cin >> ch )
    switch( ch )
```

2．一组 case 标签（label），它由关键字 case 后接一个常量表达式及其冒号构成。此常量表达式将被用来与 switch 表达式的结果做比较。在本例中，每个 case 标签代表一个元音字母。例如：

```
case 'a':
case 'e':
case 'i':
case 'o':
case 'u':
```

3．与一个或一组 case 标签相关联的语句序列。例如，为了累加每个元音字母的出现计数，我们提供了一个将元音字母计数递增 1 的赋值表达式。

4．可选的 default 标签。default 标签也被看作是一种 else 子句。如果 switch 表达式与任意一个 case 标签都不匹配，则 default 标签后面的语句被计算。例如，如果我们希望计算非

元音字母的个数，则可以增加如下 default 标签和语句：

```
default: // 任何非元音字母
    ++non_vowel_cnt;
```

关键字 case 后面的值必须是一种整数类型的常量表达式。例如，下列语句将导致编译错误：

```
// 非法的 case 标签值
case 3.14:    // 非整数
case ival:    // 非常量
```

另外，任意两个 case 标签不能有同样的值：如果有，则导致编译错误。

switch 表达式可以是任意复杂的表达式，包括函数调用的返回值。它的值与每个 case 标签相关联的值作比较，直到某个匹配成功或全部标签比较完毕。如果匹配到了某个标签，则程序从其后的语句继续执行。如果所有标签都不匹配，那么若有 default 标签的话，则从 default 后面的语句继续执行，否则程序从 switch 语句后面的第一条语句继续执行。

普遍的误解是：只有与被匹配的 case 标签相关联的语句才被执行。实际上，程序从该点开始执行并继续越过 case 边界直到 switch 语句结束。如果这一点弄错了，那么可以肯定，我们的程序也是错误的。例如，下面记录元音字母个数的 switch 程序的实现就是不正确的：

```
#include<iostream>
int main()
{
    char ch;
    int aCnt=0, eCnt=0, iCnt=0, oCnt=0, uCnt=0;
    while ( cin >> ch )
        // 警告：这是不正确的
        switch ( ch ) {
        case 'a':
            ++aCnt;
        case 'e':
            ++eCnt;
        case 'i':
            ++iCnt;
        case 'o':
            ++oCnt;
        case 'u':
            ++uCnt;
        }
    cout << "Number of vowel a: \t" << aCnt << '\n'
        << "Number of vowel e: \t" << eCnt << '\n'
        << "Number of vowel i: \t" << iCnt << '\n'
        << "Number of vowel o: \t" << oCnt << '\n'
        << "Number of vowel u: \t" << uCnt << '\n';
}
```

如果 ch 被设置为 i，则程序从 case ‘i’ 后面的语句开始执行，iCnt 递增。但是，程序的执行并没有在那里停止，而是越过 case 边界继续执行，直到 switch 语句的结束花括号。oCnt 和 uCnt 也都被递增。如果下一次 ch 被设置为 e，则 eCnt、iCnt、oCnt 和 uCnt 也都将被递增。

程序员必须显式地告诉编译器停止执行 switch 中的语句。这可以通过在 switch 语句内的每个执行单元后指定一个 break 语句来完成。在大多数条件下，一个 case 标签的最后一条语句是 break。

当遇到 break 语句时，switch 语句被终止。控制权被转移到紧跟在 switch 结束花括号后面的语句上。在我们的程序中，控制权被传递给输出语句。正确的 switch 语句如下：

```
switch ( ch ) {
case 'a':
     ++aCnt;
     break;
case 'e':
     ++eCnt;
     break;
case 'i':
     ++iCnt;
     break;
case 'o':
     ++oCnt;
     break;
case 'u':
     ++uCnt;
     break;
}
```

在大多数情况下，故意省略 break 语句的 case 标签应该提供一条注释，以指明这种省略是故意的。我们的程序不但应该能够编译执行，而且对于以后负责修改和扩展的程序员来说，也应该是可读的。与预期用法相反的代码尤其难以理解，因为我们常常不能确定这种不合常理的行为是故意的、正确的，还是疏忽的、错误的。在这种情况下，说明程序员意图的注释就增强了程序的可维护性。

程序员什么时候希望省略 break 语句，允许程序执行多个 case 标签？一种情况是，两个或多个值由相同的动作序列来处理。这是必要的，因为一个 case 标签只能与一个值相关联。因此，为了指示出一个范围，典型的做法是，我们把 case 标签一个接一个堆起来。例如，如果我们只希望记录元音字母的总数，而不是每一个元音字母的个数，那么我们可以这样写：

```
int vowelCnt = 0;
// ...
switch ( ch )
{
     // a, e, i, o, u 的任何出现都使 vowelCnt 递增
     case 'a':
     case 'e':
     case 'i':
     case 'o':
     case 'u':
          ++vowelCnt;
          break;
}
```

有些程序员喜欢把 case 标签捆绑在一起，以强调这种情形代表的是一个要被匹配的范围：

```
switch ( ch )
{
     // 另外一种合法的语法
     case 'a': case 'e':
     case 'i: case 'o': case 'u':
          ++vowelCnt;
          break;
}
```

当前实现的元音字母计数程序有个问题。例如，程序怎样处理下列输入？

```
UNIX
```

大写的 U 和 I 不能被识别为元音字母。我们的程序不能记录大写元音字母出现的次数。下面是修正后的 switch 语句，也使用了省略 break 的用法。

```
switch ( ch ) {
case 'a': case 'A':
     ++aCnt;
     break;
case 'e': case 'E':
     ++eCnt;
     break;
case 'i': case 'I':
     ++iCnt;
     break;
case 'o': case 'O':
     ++oCnt;
     break;
case 'u': case 'U':
     ++uCnt;
     break;
}
```

default 标签给出了无条件 else 子句的等价体。如果所有的 case 标签与 switch 表达式的值都不匹配，并且 default 标签也存在，则执行 default 标签后面的语句。例如，给我们的程序增加 default 的情形，使它能够记录辅音字母的个数：

```
#include <iostream>
#include <ctype.h>
int main()
{
     char ch;
     int aCnt=0, eCnt=0, iCnt=0, oCnt=0, uCnt=0,
     consonantCnt = 0;
     while ( cin >> ch )
          switch ( ch )
          {
          case 'a': case 'A':
               ++aCnt;
               break;
          case 'e': case 'E':
               ++eCnt;
               break;
          case 'i': case 'I':
```

```
                ++iCnt;
                break;
            case 'o': case 'O':
                ++oCnt;
                break;
            case 'u': case 'U':
                ++uCnt;
                break;
            default:
                if ( isalpha( ch ))
                    ++consonantCnt;
                break;
            }
        cout << "Number of vowel a: \t" << aCnt << '\n'
            << "Number of vowel e: \t" << eCnt << '\n'
            << "Number of vowel i: \t" << iCnt << '\n'
            << "Number of vowel o: \t" << oCnt << '\n'
            << "Number of vowel u: \t" << uCnt << '\n'
            << "Number of consonants: \t" << consonantCnt << '\n';
}
```

isalpha()是标准 C 库的一个例程：如果它的参数是一个英文字母，则返回值为 true。为了使用它，程序员必须包含系统头文件 ctype.h（我们将在第 6 章详细介绍 ctype.h 例程。）

尽管在 switch 语句的最后一个标签中指定 break 语句不是严格必需的，但是为安全起见，我们最好总是提供一个 break 语句。如果后来在 switch 语句的末尾又加了另外一个 case 标签，那么原来最后一个标签现在已不是最后一个了，那么如果它缺少 break 的话，也将导致执行两个 case 标签中的所有语句。

声明语句也可以被放在 switch 语句的条件中，如下所示：

```
switch( int ival = get_response() )
```

ival 被初始化，并且读初始化值成为与每个 case 标签作比较的值。ival 在整个 switch 语句中是可见的，但在其外面并不可见。

把一条声明语句放在与 case 或 default 相关联的语句中是非法的，除非它被放在一个语句块中。例如，下列代码将导致编译时刻错误：

```
case illegal_definition:
    // 错误：声明语句必须被放在语句块中
    string file_name = get_file_name();

    // ...
    break;
```

如果一个定义没有被包围在一个语句块中，那么它在 case 标签之间是可见的，但是只有当定义它的 case 标签被执行时，它才能被初始化。因此需要一个语句块来保证名字是可见的，并且也只有这个语句块才能够使用它（而且可以保证，它只在这个语句块中才能被初始化）。为了使我们的程序能通过编译，必须引入语句块，重新实现 case 标签如下：

```
case ok:
    {
        // ok：声明语句被放在语句块中
```

```
            string file_name = get_file_name();
            // ...
            break;
        }
```

练习 5.5

请修改元音字母计数程序，使它能够记录被读取到的空格、TAB 键以及换行符的个数。

练习 5.6

请修改元音字母计数程序，使它能够记录下列双字符序列出现的次数：ff、fl 和 fi。

练习 5.7

下面每段代码都暴露了使用 switch 语句的一个普遍错误。请指出并修改这些错误。

(a)

```
switch ( ival ) {
case 'a': aCnt++;
case 'e': eCnt++;
default: iouCnt++;
}
```

(b)

```
switch (ival ) {
case 1:
    int ix = get_value();
    ivec[ ix ] = ival;
    break;
default:
    ix = ivec.size()-1;
    ivec[ ix ] = ival;
}
```

(c)

```
switch (ival ) {
case 1, 3, 5, 7, 9:
    oddcnt++;
    break;
case 2, 4, 6, 8, 10:
    evencnt++;
    break;
}
```

(d)

```
int ival=512 jval=1024, kval=4096;
int bufsize;
// ...
switch( swt ) {
case ival:
    bufsize = ival * sizeof( int );
    break;
case jval:
```

```
            bufsize = jval * sizeof( int );
            break;
        case kval:
            bufsize = kval * sizeof( int );
            break;
        }
(e)
        enum { illustrator = 1, photoshop, photostyler = 2 };
        switch (ival) {
        case illustrator:
            --illus_license;
            break;
        case photoshop:
            --pshop_license;
            break;
        case photostyler:
            --pstyler_license;
            break;
        }
```

5.5 for 循环语句

我们已经看到，大量程序都会涉及到在某个条件保持为真时重复执行一组语句。例如。当没有到达文件尾时，读入并处理文件的下一个元素；对于每个不等于 vector 末元素下一位置的索引，获取并处理 vector 的元素；等等。C++提供了三种循环控制语句，以支持当某个特定的条件保持为真时，重复执行一个语句或语句块。我们已经看到了 for 和 while 循环的大量例子。

for 和 while 循环首先进行条件的真值测试。这意味着这两个循环都可以在相关语句或语句块还没有被执行的情况下就终止了。第三种循环结构，do-while 循环，保证语句或语句块至少被执行一次——在这些语句被执行之后进行条件测试。在本节中，我们将详细了解 for 循环。while 循环是 5.6 节的话题，而 do-while 循环将在 5.7 节中介绍。

for 循环最普遍的用法是遍历一个定长的数据结构，如数组或 vector（向量）。例如：

```
#include<vector>

int main()
{
    int ia[10];
    for ( int ix = 0; ix< 10; ++ix )
        ia[ ix ] = ix;

    vector<int> ivec( ia, ia+10 );
    vector<int>::iterator iter = ivec.begin();

    for ( ; iter != ivec.end(); ++iter )
        *iter *= 2;
```

```
        return 0;
    }
```

for 循环的语法形式如下：

```
    for ( init-statement; condition; expression )
        statement
```

init-statement（初始化语句）可以是声明语句或表达式。一般地，它被用来对一个在循环过程中被递增的变量进行初始化，或者赋给一个起始值。如果不需要初始化或者它已经在别处出现，则可以省略 init-statement，比如前面例子中的第二个 for 循环。但是，必须要有一个分号表明缺少该语句（或给出空语句）。下面都是合法的 init-statement：

```
// 假设 index 和 iter 已经在别处定义
for ( index = 0; ...
for ( ; /* 空的初始化语句 */ ...
for ( iter = ivec.begin(); ...
for ( int lo = 0, hi = max; ...
for ( char *ptr = getStr(); ...
```

condition（条件语句）用作循环控制。condition 计算结果为 true 多少次，则 statement（语句）就执行多少次。statement 可以是单个语句，也可以是复合语句。如果 condition 的第一次计算结果为 false，则 statement 从不会被执行。下面都是合法的 condition 实例：

```
(...; index < arraySize; ... )
(...; iter != ivec.end(); ... )

(...; *st1++ = *st2++; ... )
(...; char ch = getNextChar(); ... )
```

expression（表达式）在循环每次迭代后被计算，一般用它来修改在 init-statement 中被初始化的、在 condition 中被测试的变量。如果 condition 的第一次计算结果为 false，则 expression 从不会被计算。下面都是 expression 的合法实例：

```
( ...; ...; ++index )
( ...; ...; ptr = ptr->next )

( ...; ...; ++i, --j, ++cnt )
( ...; ...; ) // null instance
```

已知下列 for 循环：

```
const int sz = 24;
int ia[ sz ];

vector<int> ivec( sz );

for ( int ix = 0; ix< sz; ++ix ) {
    ivec[ ix ] = ix;
    ia[ ix ] = ix;
}
```

它的计算顺序如下；

1．在循环开始，执行一次 init-statement。在本例中，ix 被初始化为 0。

2．执行 condition。如果它的计算结果为 true，则执行复合语句。本例中，只要 ix 小于 sz，ix 就被赋给 ivec[ix]和 ia[ix]，而 false 条件将终止循环。初始的 false 条件将导致复合语句从不被执行。

3．执行 expression。典型情况下，它修改在 init-statement 中被初始化、在 condition 中被测试的变量。在本例中，ix 被递增 1。

这三步代表了一个完整的 for 循环迭代。现在重复第 2 步，然后是第 3 步，直到 condition 为 false：即 ix 不再小于 sz 为止。

在 init-statement 中可以定义多个对象，但只能出现一个声明语句，因此，所有对象都必须是相同的类型。例如：

```
for ( int ival = 0, *pi = &ia, &ri = val;
          ival< size;
          ++ival, ++pi, ++ri )
      // ...
```

在 for 循环的 condition（条件部分）中定义的对象很难管理：它的最终计算结果必须为 false，否则循环将水远不会终止。下面是个例子，它看起来有点不太自然：

```
#include <iostream>

int main()
{
      for ( int ix = 0;
                bool done = ix == 10;
                ++ix )
          cout << "ix: " << ix << endl;
}
```

在 for 循环 condition 中定义的对象的可视性局限在 for 循环体内。例如，for 循环后面对 iter 的测试是个编译时刻错误[11]：

```
int main()
{
      string word;
      vector< string > text;

      // ...

      for ( vector< string >::iterator
                iter = text.begin(),
                iter_end = text.end();
                iter != text.end(); ++iter )
```

[11] 在标准 C++之前，定义在 init-statement 中的对象的可视性可以扩展到 for 循环包含的语句块之外。例如，给定下面两个 for 循环，它们位于同一个语句块之中：

```
{
    // 在标准 C++中合法；但是在标准 C++之前，ival 被定义两次，所以不合法
    for ( int ival = 0; ival < size; ++ival ) // ...
    for ( int ival = size -1; ival >= 0; --ival ) // ...
}
```

在标准 C++之前，ival 被标记为编译时刻错误“重复定义”。然而在标准 C++中，jval 的两个实例都是局部的，仅限于它们各自的 for 循环．所以下述程序逻辑是合法的。

```
        {
            if ( *iter == word )
                break;
            // ...
        }
        // 错误: iter 和 iter_end 不可见
        if ( iter != iter_end )
            // ...
    }
```

练习 5.8

下列哪些 for 循环是错误的?

```
(a) for ( int *ptr = &ia, ix = 0;
          ix< size && ptr != ia+size;
          ++ix, ++ptr )
        // ...
(b) for ( ; ; ) {
        if ( some_condition )
            break;
        // ...
    }
(c) for ( int ix = 0; ix< sz; ++ix )
        // ...

    if ( ix != sz )
        // ...
(d) int ix;

    for ( ix< sz; ++ix )
        // ...
(e) for ( int ix = 0; ix< sz; ++ix, ++ sz )
        // ...
```

练习 5.9

假设你被邀请写一个 for 循环的风格指导，它将被用在项目组范围内。解释并举例说明你的用法规则。如果有三部分，则分别加以说明。如果你强烈反对用法规则（至少对于 for 循环来说如此）则解释并举例说明原因。

练习 5.10

已知函数声明:

```
bool is_equal( const vector<int> &v1,
    const vector<int> &v2 );
```

写出函数体，用它来确定两个 vector 是否相等。对长度不同的两个 vector，只比较较小

的 vector 的元素数目。例如，给出向量 (0,1,1,2) 和 (0,1,1,2,3,5,8)，is_equal()返回 true，而 v1.size()和 v2.size()返回 vector 的长度。

5.6 while 语句

while 循环的语法形式如下：

```
while ( condition )
    statement
```

condition（条件）计算结果为 true 多少次，则循环就迭代多少次，语句或语句块也被执行多少次。执行序列如下：

1．计算 condition；

2．如果 condition 为 true，则执行 statement（语句）；

如果 condition 的第一次计算结果为 false，则 statement 永远不会被执行。

while 循环的 condition 可以是表达式，如：

```
bool quit = false;

// ...
while ( ! quit ) {
    // ...
    quit = do_something();
}
string word;
while ( cin >> word ) { ... }
```

或是初始化定义，如：

```
while ( symbol *ptr = search( name )) {
    // do something
}
```

在后一个例子中，ptr 只在与 while 循环相关联的语句块内可见，如同在 if、switch 及 for 语句中一样。

下面是 while 循环的一个例子，它迭代了一个由一对指针指向的元素集合：

```
int sumit( int *parray_begin, int *parray_end )
{
    int sum = 0;
    if ( ! parray_begin || ! parray_end )
        return sum;
    while ( parray_begin != parray_end )
        // 把当前元素的值加到 sum 上
        // 然后增加指针，使其指向下一个元素
        sum += *parray_begin++;

    return sum;
}
int ia[6] = { 0, 1, 2, 3, 4, 5 };
```

```
int main()
{
    int sum = sumit( &ia[0], &ia[ 6 ] );
    // ...
}
```

为了使 sumit()能够正确地执行，两个指针必须指向同一个数组的元素（parray_end 可以安全地指向数组最末元素的下一个位置）。如果不是这样，我们就说 sumit()的行为是未定义的。（最好的情况下它会返回一个没有意义的结果。）不幸的是，在 C++中我们没有办法保证两个指针都指向同一个数组。正如在第 12 章中将要看到的，C++标准库在实现泛型算法时，这些算法接收两个指针，分别指向一个容器的首尾元素。

练习 5.11

下列 while 循环声明哪些是错误的？

```
(a)  string bufString, word;
     while ( cin >> bufString >> word )
         // ...

(b)  while ( vector<int>::iterator iter != ivec.end() )
         // ...

(c)  while ( ptr = 0 )
         ptr = find_a_value();

(d)  while ( bool status = find( word )) {
         word = get_next_word();
         if ( word.empty() )
             break;
         // ...
     }
     if ( ! status )
         cout << "Did not find any words\n";
```

练习 5.12

while 循环尤其擅长在某个条件保持为真时不停地执行。例如，当没有到达文件尾时，有读取下一个值。一般认为 for 循环是一种按步循环：用一个索引按步遍历集合中一定范围内的元素。按每个循环的习惯用法编写程序，然后再用另外一种结构重新编写。如果你只能用一种循环编写程序；你会选择哪种结构？为什么？

练习 5.13

写一个小程序，从标准输入读入字符串序列，直到出现相同的词或者所有的词都已经被输入。用 while 循环每次读入一个单词。如果连续出现相同的词，则用 break 语句终止循环。如果出现重复，则输出重复的词，否则输出消息说明没有重复的词。

5.7 do while 语句

假设我们被要求写一个交互程序，实现将英里转换成公里。程序大致如下：

```
int val;
bool more = true; // 为启动循环而设置的哑元
while ( more )
{
    val = getValue();
    val = convertValue(val);
    printValue(val);
    more = doMore();
}
```

现在的问题是，循环控制是在循环体内被设置的。但是，对于 for 和 while 循环来说，除非循环条件计算结果为真，否则循环体将永不被执行。这意味着我们必须给出第一个值来启动循环。另一种更好的方案是，我们可以使用 do while 循环。do while 循环保证至少执行一次 statement。do while 循环的语法形式如下：

```
do
    statement
while ( condition );
```

statement 在 condition 被计算之前执行。如果 condition 的计算结果为 false，则循环终止。前面的程序现在看起来如下：

```
do
{
    val = getValue();
    val = convertValue(val);
    printValue(val);
} while ( doMore() );
```

不像其他循环语句，do while 循环的条件（即 condition 部分）不支持对象定义——即，不能写：

```
// 错误：在 do 循环的 condition 部分不支持声明语句
do {
    // ...
    mumble( foo );
} while ( int foo = get_foo() ) // 错误
```

因为只有在语句或语句块首先被执行之后，条件部分才被计算。

练习 5.14

下列 do while 循环，哪些是错误的？

```
(a)  do
         string rsp;
         int val1, val2;
```

```
            cout << "please enter two values: ";
            cin >> val1 >> val2;
            cout << "The sum of " << val1
                 << " and " << val2
                 << " = " << val1 + val2 << "\n\n"
                 << "More<< [yes][no] ";
            cin >> rsp;
        while ( rsp[0] != 'n' );
(b)     do {
            // ...
        } while ( int ival = get_response() );
(c)     do {
            int ival = get_response();
            if ( ival == some_value() )
                break;
        } while ( ival );
        if ( !ival )
            // ...
```

练习 5.15

写一个小程序，实现从用户处得到两个字符串，然后输出两者的比较结果（按字典顺序，即字母顺序输出哪个字符串小于另一个字符串）。继续要求用户输入，直到用户请求退出为止。请使用 string 类型、string 的小于操作符以及 do while 循环大成。

5.8 break 语句

break 语句终止最近的 while、do while、for 或 switch 语句。程序的执行权被传递给紧接着被终止语句之后的语句。例如，下面的函数在一个整数数组中查找一个特殊值的首次出现，如果找到，则返回它的索引值，否则返回-1。实现代码如下：

```
// value 是否在 ia 中？若是，则返回索引位置；否则返回-1
int search( int *ia, int size, int value )
{
    // 确保 ia != 0 以及 size > 0 ...
    int loc = -1;

    for ( int ix = 0; ix< size; ++ix ) {
        if ( value == ia[ ix ] ) {
            // ok: 找到了!
            // 设置好位置，然后离开循环
            loc = ix;
            break;
        }
    } // for 循环结束处

    // break 语句跳转到这里
```

```
        return loc;
}
```

在本例中，break 终止了 for 循环，执行权被转交给紧随其后的 return 语句。在这个例子中，break 终止了包含它的 for 循环，而不是它外边的 if 语句。如果一个 break 语句出现在 if 语句的内
部，但是并不被包含在 switch 或循环语句中，那么这样的 break 语句将导致编译错误。例如。

```
// 错误：break 语句的非法出现
if ( ptr ) {
    if ( *ptr == "quit" )
        break;
    // ...
}
```

一般来说，break 语句只能出现在循环或 switch 语句中。

当 break 出现在嵌套的 switch 或循环语句中时，里层的 switch 或循环语句被终止并不影响外层的 switch 或循环。例如：

```
while ( cin >> inBuf )
{
    switch( inBuf[ 0 ] ) {
    case '-':
        for ( int ix = 1; ix< inBuf.size(); ++ix ) {
            if ( inBuf[ ix ] == ' ' )
                break; // #1
            // ...
            // ...
        }
        break; // #2

    case '+':
        // ...
    }
}
```

由//#1 标记的 break 终止了 case 标签连字符（‘-’）内的 for 循环，但没有终止 switch 语句。类似地，由//#2 标记的 break 终止了 inBuf 第一个字符的 switch 语句，但没有终止外层的 while 循环，这个循环每次从标准输入读入一个字符串。

5.9 continue 语句

continue 语句导致最近的循环语句的当前迭代结束，执行权被传递给条件计算部分。不像 break 语句终止的是整个循环，continue 语句只终止当前的迭代。例如，下面的程序段读取一个程序文本文件，每次读入一个词。它处理每个以下划线开始的词，否则，当前的循环迭代被终止：

```
while ( cin >> inBuf ) {
    if ( inBuf[0] != '_' )
        continue; // 终止迭代劳
    // 仍然在这里？处理字符串
}
```

continue 语句只有出现在循环语句中才是合法的。

5.10 goto 语句

goto 语句提供了函数内部的无条件分支，它从 goto 语句跳转到同一函数内部某个位置的一个标号语句。在当前关于良好的程序设计实践的看法中，它的用法被认为是应该抛弃的。

goto 语句的语法形式如下：

```
goto label;
```

这里的 label 是用户定义的标识符。**标号**（labe）语句只能用作 goto 的目标，必须由冒号结束，且标号语句不能紧接在结束右花括号的前面。在标号语句后面加一个空语句，是处理这种限制的典型方法。例如：

```
        end: ;   // 空语句
}
```

goto 语句不能向前跳过没有被语句块包围的声明语句。例如，下面的代码将导致编译时刻错误：

```
int oops_in_error()
{
    // mumble
    goto end;

    // 错误：跳过声明语句
    int ix = 10;
    // ... code using ix

    end: ;
}
```

正确的做法是把声明语句以及使用该声明的语句放在一个语句块中。例如：

```
int ok_its_fixed()
{
    // mumble
    goto end;
    {
        // ok: 在语句块中声明语句
        int ix = 10;

        // ... code using ix
    }
    end: ;
}
```

这里的原因与 switch 语句的 case 标签中的声明语句相同，是编译器为类对象插入构造函数/析构函数调用的需要。语句块保证构造函数和析构函数都被执行或忽略，保证对象只在它被初始化的地方才可见。

然而，向后跳过一个已被初始化的对象定义不是非法的。为什么？跳过对象的初始化操作是个程序设计错误。多次初始化一个对象尽管低效，但仍然是正确的。例如：

```
// 向后跳过声明语句没问题
```

```
void
mumble( int max_size )
{
    begin:
    int sz = get_size();
    if ( sz<= 0 ) {
        // 发出警告 ...
        goto end;
    }
    else
    if ( sz > max_size )
        // 得到一个新的 sz 值
        goto begin;
    { // ok: 整个语句块被跳过
        int *ia = new int[ sz ];
        doit( ia, sz ); // whatever that is ...
        delete [] ia;
    }
    end:
    ;
}
```

goto 语句是现代程序设计语言中最过时的特性。使用 goto 语句常常使程序控制流难于理解和修改，goto 语句的大多数用法可以用条件或循环语句来代替。如果你发现自己正在使用 goto 语句，那么建议你不要让它跨越过多的程序序列。

5.11 链表示例

在第 3 章和第 4 章中，我们用一个类的设计和实现来练习 C++的类机制，并以此作为每一章的结束。类似地，本章我们将用一个单向链表类的实现作为本章的结束。（在第 6 章，我们将看到标准库提供的双向链表容器类。）第一次阅读本书的读者可以略过本节，在看过第 13 章之后再回头阅读本节内容。（如果你希望继续阅读本节，那么假定你至少已经阅读了 2.3 节或 3.15 节，即至少熟悉了类机制的语法和术语，比如什么是构造函数等等。如果不是这样，建议你先阅读那两节，或其中一节。）

链表是一个数据项序列，每个数据项都包含一个某种类型的值和链表下一项的地址，地址可以为空（null），链表也可以为空，即链表中可以没有数据项。链表不可能为满，但是当程序的空闲存储区被耗尽时，试图创建一个新链表项会失败。

链表类必须支持的操作是什么？用户必须能够插入（insert）或删除（remove），以及查找（find）一个项。用户必须能够查询链表的长度（size）、显示（display）链表，以及比较两个链表是否相等（equality）。另外，还要支持翻转（reverse）链表以及连接（concatenate）两个链表。

size()最简单的实现是迭代链表，返回所遍历的元素的个数。复杂一点的实现是将长度作为一个数据成员存储起来。size()的第二个实现效率很高，它只是简单地返回相关联的成员。额外的复杂性是每次插入和删除一个元素时，都需要更新这个成员。

我们选择第二种方案，把元素个数保存在数据成员 size 中，并且在必要时更新这个数据成员。我们的假设是用户可能会频繁查询 size()，因此，它必须足够快。（把公有接口和私有实现分离的好处之一是，如果我们的假设被证明是错误的，那么可以给出一个新的实现，而不必改变使用 size()的程序，只要保持同样的返回值类型和参数表。）

我们的 insert()操作有两个参数，指向一个已存在的链表元素的指针，以及一个新值。新值被插入到这个已存在的元素之后。例如，给出链表：

```
1    1    2    3    8
```

如下调用：

```
mylist.insert( pointer_to_3, 5 );
```

将把链表修改为：

```
1    1    2    3    5    8
```

为了做到这一点，我们需要向用户提供一种方法来访问某个特定的数据项的地址，例如前面例子中的元素 3。一种方法是通过 find()操作。例如：

```
pointer_to_3 = mylist.find( 3 );
```

find()把待搜索的值作为它的参数。如果这个元素存在，则 find()返回指向该元素的指针。否则返回 0。

我们还希望支持两种特殊情况的 insert()：在链表头和链表尾插入。这两种情况只需要用户指定将要被插入的位：

```
insert_front( value );

insert_end( value );
```

删除操作包括以下这些：删除单个值、删除最前面的元素、删除所有的元素。

```
remove( value );

remove_front();

remove_all();
```

display()操作为链表提供一个格式化的输出，包括链表的长度以及每个元素。空链表显示如下：

```
(0)  ( )
```

有 7 个元素的链表显示为：

```
(7)( 0 1 1 2 3 5 8 )
```

reverse()只是翻转元素的顺序。例如，在调用：

```
mylist.reverse();
```

之后，前面的链表显示为；

```
(7)( 8 5 3 2 1 1 0 )
```

连接操作将第二个链表附加到第一个的末尾。例如，已知链表：

```
(4)( 0 1 1 2 )     // 链表 1
```

```
(4)( 2 3 5 8 ) // 链表 2
```

如下操作：

```
list1.concat( list2 );
```

将 list1 修改为：

```
(8)( 0 1 1 2 2 3 5 8 )
```

为了使 list1 成为真正的斐波那契数列，可以应用 remove()：

```
list1.remove( 2 );
```

一旦定义了链表类的行为，我们下一步的工作就是实现它。我们选择把 list（链表）和 list_item（链表项）作为独立的类抽象。（现在，我们把链表实现局限在 int 型的元素上。因此，我们的类名为 ilist 和 ilist_item。）

我们的链表包含一个成员_at_front（它指向链表头）、一个成员_at_end（它指向链表的尾），以及成员_size（记录链表的当前长度）。当一个链表对象刚刚被定义时，这三个成员必须被初始化为 0。我们用缺省构造函数来保证这一点：

```
class ilist_item;
class ilist {
public:
    // 缺省构造函数
    ilist() : _at_front( 0 ),

    _at_end( 0 ), _size( 0 ) {}

    // ...

private:
    ilist_item *_at_front;
    ilist_item *_at_end;

    int _size;
};
```

这使我们能够如下定义 ilist 对象：

```
ilist mylist;
```

但是其他事情都还没有做。现在我们来增加对链表长度查询的支持。在类定义的公有部分声明成员函数 size()后，可以如下定义这个成员函数：

```
inline int ilist::size() { return _size; }
```

现在我们可以这样写：

```
int size = mylist.size();
```

我们不希望用户用一个链表初始化或赋值给另一个链表（以后我们会修改——这种修改不会要求改动用户程序）。为了防止初始化和赋值，我们把 ilist 的拷贝构造函数和拷贝赋值操作符声明为私有成员，而且不提供它们的定义。下面是修改后的 ilist 类定义：

```
class ilist {
public:
    // 不再显示成员函数的定义
    ilist();
```

```
        int size();
        // ...

    private:
        // 禁止用一个链表对象初始化或赋值给另一个
        ilist( const ilist& );
        ilist& operator=( const ilist& );
        // 数据成员同前
    };
```

由于这个定义，下面两条语句都将导致编译时刻错误，因为 main()不能访问 ilist 类的私有成员：

```
    int main()
    {
        ilist yourlist( mylist );  // 错误
        mylist = mylist;  // 错误
    }
```

下一步工作是支持插入数据项。我们已经选择把数据项表示成一个独立的类抽象，它包含成员_value 和_next。其中，_value 表示该项的值，_next 记录链表中下一项的地址：

```
    class ilist_item {
    public:
        // ...

    private:
        int _value;
        ilist_item *_next;
    };
```

ilist_item 的构造函数要求指定一个值，以及指定一个指向 ilist_item 的指针，后者是可选项，不是必需的。如果出现第二个参数（即指向 ilist_item 的指针）的话，则新的项被插入在该指针表示的 ilist_item 的后面。例如，给出链表：

```
    0    1    1    2    5
```

如下构造函数调用：

```
    ilist ( 3, pointer_to_2 );
```

将链表修改为：

```
    0    1    1    2    3    5
```

下面是我们的实现代码。（再次说明，第二个参数 item 是可选的，如果用户不提供值的话，编译器将传递缺省值 0。缺省值是在函数声明中指定的，而不是在定义中指定。第 7 章将对此给出完整的说明。）

```
    class ilist_item {
    public:
        ilist_item( int value, ilist_item *item_to_link_to = 0 );

        // ...
    };

    inline
    ilist_item::
```

```
ilist_item( int value, ilist_item *item )
: _value( value )
{
     if ( !item )
          _next = 0;
     else {
          _next = item->_next;
          item->_next = this;
     }
}
```

ilist 中普通版本的 insert()操作以一个待插入的值以及一个 ilist_item 指针作为参数，新的项被插在这个指针所指的项的后边。下向是我们的第一个作品（有两个问题——你能找出来吗？）：

```
inline void
ilist::
insert( ilist_item *ptr, int value )
{
     new ilist_item( value, ptr );
     ++_size;
}
```

第一个问题是，没有核实指针是否含有非 0 的地址。我们必须识别和处理这种情况。因为它的出现可能会使我们的程序在运行时刻崩溃。应该如何处理呢？一种方案是通过调用 C 标准库 abort()函数来终止程序（该函数在 C 头文件 cstdlib 中）：

```
#include<cstdlib>

// ...
if ( ! ptr )
     abort();
```

另一种方案是使用 assert()宏来终止程序，但首先要声明引起断言的条件：

```
#include<cassert>

// ...
assert( ptr != 0 );
```

第三种方案是抛出一个异常。例如：

```
if ( ! ptr )
     throw "Panic: ilist::insert(): ptr == 0";
```

一般地，我们应该尽可能地避免终止程序。终止程序实际上是对用户落井下石，而我们或者支持部门可以分离并解决这些问题。

如果我们在错误点上不能继续处理，那么一般来说，抛出异常比终止程序更好一些。异常会把控制权传送到程序先前被执行的部分，而它们有可能能够解决这个问题。

我们的实现能够识别空指针，并将其视为“把 value 插入链表头”的请求：

```
if ( ! ptr )
     insert_front( value );
```

在我们的实现中，第二个缺点是过于哲理性。_size 和 size()实现对是一个尝试性设计。虽然我们相信链表长度的存储和 inline 获取方式极好地满足了用户的要求，但是，实际上我

们可以用一个策略来代替它。该策略只有在必要的时候它才计算链表的长度（即按需计算）。在按需引算长度的实现中，成员_size 被去掉了。当我们写下下面的代码时：

```
++_size;
```

我们就把 insert()的实现与当前链表类的实现紧密地耦合在一起。如果链表类的实现改变了，则 insert()也不再正确，也必须随之改动。同样 insert_front()、insert_end()以及删除操作的实例也必须改动。在新的设计方案中，我们不再把“对链表类的实现细节的依赖性”扩散到这多个插入和删除操作中，而是选择了将依赖性封装到一对函数中：

```
inline void ilist::bump_up_size() { ++_size; }
inline void ilist::bump_down_size() { --_size; }
```

因为已经将函数声明为 inline，所以我们的设计不会影响实现的效率。下面是修改后的实现：

```
inline void
ilist::
insert( ilist_item *ptr, int value )
{
     if ( !ptr )
          insert_front( value );
     else {
          bump_up_size();
          new ilist_item( value, ptr );
     }
}
```

显然 insert_front()和 insert_end()的实现很容易，它们都必须处理空链表的这种特例。下面是它们的实现：

```
inline void
ilist::
insert_front( int value )
{
     ilist_item *ptr = new ilist_item( value );
     if ( !_at_front )
          _at_front = _at_end = ptr;
     else {
          ptr->next( _at_front );
          _at_front = ptr;
     }
     bump_up_size();
}

inline void
ilist::
insert_end( int value )
{
     if ( !_at_end )
          _at_end = _at_front = new ilist_item( value );
     else _at_end = new ilist_item( value, _at_end );
          bump_up_size();
```

```
}
```

find()在链表中查找一个值。如果这个值存在，则 find()返回指向该值的指针，否则返回 0。下面是它的实现：

```
ilist_item*
ilist::
find( int value )
{
    ilist_item *ptr = _at_front;

    while ( ptr )
    {
        if ( ptr->value() == value )
            break;
        ptr = ptr->next();
    }

    return ptr;
}
```

我们可以按如下方式使用 find()：

```
ilist_item *ptr = mylist.find( 8 );
mylist.insert( ptr, some_value );
```

或用更紧凑的用法：

```
mylist.insert( mylist.find( 8 ), some_value );
```

在练习插入操作之前，我们需要 display()函数，以便能够看到我们把程序修改得是否合适。display()的算法十分简单：从第一个元素开始，按顺序输出每个元素直到全部输出为止。你知道为什么下面的 for 循环设计失败了吗？

```
// 喔！不正确
// 目的：显示链表中除了最后一个元素之外的所有元素
for ( ilist_item *iter = _at_front; // 从链表的最前面开始
        iter != _at_end; // 在链表尾终止
        ++iter ) // 往前移动一项
    cout << iter->value() << ' ';

// 现在显示最后一个元素
cout << iter->value();
```

失败的原因是，链表中的元素并不是被连续存储在内存中的。指针的算术运算：

```
++iter;
```

并没有把 iter 向前移动到指向 ilist 下一个元素的位置上，而是把 iter 加上了一个 ilist_item 对象的字节长度。我们不知道递增后的 iter 指向什么对象，也不知道循环是否会结束。为了指向链表的下一个元素，iter 必须在每次迭代后，显式地被重置为由 ilist_item 的数据成员_next 指向的下一项：

```
iter = iter->_next;
```

我们用一组 inline 访问函数封装对成员_value 和_next 的访问。下面是修订过的 ilist_item

类的定义：

```
class ilist_item {
public:
     ilist_item( int value, ilist_item *item_to_link_to = 0 );
     int value() { return _value; }
     ilist_item* next() { return _next; }
     void next( ilist_item *link ) { _ne xt = link; }
     void value( int new_value ) { _value = new_value; }

private:
     int _value;
     ilist_item *_next;
};
```

下面是 display()的实现，它使用了前面的 ilist_item 类定义：

```
#include <iostream>
class ilist {
public:
     void display( ostream &os = cout );
     // ...
};

void
ilist::
display( ostream &os )
{
     os << "\n( " << _size << " )( ";
     ilist_item *ptr = _at_front;

     while ( ptr ) {
          os << ptr->value() << " ";
          ptr = ptr->next();
     }

     os << ")\n";
}
```

下面是一个小程序，它练习了目前我们定义的 ilist 类：

```
#include <iostream>
#include "ilist.h"

int main()
{
     ilist mylist;

     for ( int ix = 0; ix< 10; ++ix ) {
          mylist.insert_front( ix );
          mylist.insert_end( ix );
     }

     cout << "Ok: after insert_front() and insert_end()\n";
```

```
    mylist.display();

    ilist_item *it = mylist.find( 8 );

    cout << "\n"
         << "Searching for the value 8: found it<<"
         << ( it << " yes!\n" : " no!\n" );
    mylist.insert( it, 1024 );

    cout << "\n"
         << "Inserting element 1024 following the value 8\n";
    mylist.display();

    int elem_cnt = mylist.remove( 8 );

    cout << "\n"
         << "Removed " << elem_cnt << " of the value 8\n";
    mylist.display();

    cout << "\n" << "Removed front element\n";

    mylist.remove_front(); mylist.display();

    cout << "\n" << "Removed all elements\n";

    mylist.remove_all(); mylist.display();
}
```

编译并运行程序，产生如下输出：

```
Ok: after insert_front() and insert_end()
( 20 )( 9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9 )
Searching for the value 8: found it<< yes!
Inserting element 1024 following the value 8
( 21 )( 9 8 1024 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9 )
Removed 2 of the value 8
( 19 )( 9 1024 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 9 )
Removed front element
( 18 )( 1024 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 9 )
Removed all elements
( 0 )( )
```

如同向链表中插入数据项一样，用户也需要从链表中删除数据项。我们支持三种删除元素的操作：

```
void remove_front();
void remove_all();

int remove( int value );
```

下面是 remove_front()的实现代码：

```
inline void
ilist::
remove_front()
{
    if ( _at_front ) {
        ilist_item *ptr = _at_front;
        _at_front = _at_front ->next();
        bump_down_size();

        delete ptr;
    }
}
```

remove_all()重复调用 remove_front()，直到链表为空：

```
void
ilist::
remove_all()
{
    while ( _at_front )
        remove_front();

    _size = 0;
    _at_front = _at_end = 0;
}
```

实现 remove()的一般做法也要利用 remove_front()来处理特例，即当一个或多个要被删除的项位于链表头的时候。否则，我们用两个指针来迭代链表，一个指向前一个元素，另一个指向当前元素。在迭代过程中，找到要被删除的项并删除它，然后再重新连接链表。下面是实现代码：

```
int
ilist::
remove( int value )
{
    ilist_item *plist = _at_front;
    int elem_cnt = 0;

    while ( plist && plist->value() == value )
    {
        plist = plist->next();
        remove_front();
        ++elem_cnt;
    }

    if ( ! plist )
        return elem_cnt;
    ilist_item *prev = plist;
    plist = plist->next();

    while ( plist )
```

```
        {
            if ( plist->value() == value )
            {
                prev->next( plist->next() );
                delete plist;
                ++elem_cnt;
                bump_down_size();
                plist = prev->next();
                if ( ! plist )
                {
                    _at_end = prev;
                    return elem_cnt;
                }
            }
            else
            {
                prev = plist;
                plist = plist->next();
            }
        }
        return elem_cnt;
    }
```

下面的程序练习了这些删除操作，测试了下列情况：（1）所有被删除的项位于链表尾；（2）删除链表中的所有项；（3）不存在要被删除的项；（4）有的项位于链表头，同时有的项位于链表的后部。

```
    #include<iostream>
    #include "ilist.h"
    int main()
    {
        ilist mylist;

        cout << "\n--------------------------------------------------\n"
            << "test #1: items at end\n"
            << "--------------------------------------------------\n";
        mylist.insert_front( 1 ); mylist.insert_front( 1 );
        mylist.insert_front( 1 );
        mylist.insert_front( 2 ); mylist.insert_front( 3 );
        mylist.insert_front( 4 );
        mylist.display();
        int elem_cnt = mylist.remove( 1 );

        cout << "\n" << "Removed " << elem_cnt << " of the value 1\n";

        mylist.display();
        mylist.remove_all();

        cout << "\n--------------------------------------------------\n"
            << "test #2: items at front \n"
            << "--------------------------------------------------\n";
```

```
        mylist.insert_front( 1 ); mylist.insert_front( 1 );
        mylist.insert_front( 1 );
        mylist.display();

        elem_cnt = mylist.remove( 1 );
        cout << "\n" << "Removed " << elem_cnt << " of the value 1\n";
        mylist.display();

        mylist.remove_all();

        cout << "\n-------------------------------------------------\n"
             << "test #3: no items present\n"
             << "-------------------------------------------------\n";

        mylist.insert_front( 0 ); mylist.insert_front( 2 );
        mylist.insert_front( 4 );
        mylist.display();

        elem_cnt = mylist.remove( 1 );
        cout << "\n" << "Removed " << elem_cnt << " of the value 1\n";
        mylist.display();

        mylist.remove_all();

        cout << "\n-------------------------------------------------\n"
             << "test #4: items at front and end\n"
             << "-------------------------------------------------\n";
        mylist.insert_front( 1 ); mylist.insert_front( 1 );
        mylist.insert_front( 1 );

        mylist.insert_front( 0 ); mylist.insert_front( 2 );
        mylist.insert_front( 4 );

        mylist.insert_front( 1 ); mylist.insert_front( 1 );
        mylist.insert_front( 1 );

        mylist.display();

        elem_cnt = mylist.remove( 1 );
        cout << "\n" << "Removed " << elem_cnt << " of the value 1\n";
        mylist.display();
    }
```

编译并运行该程序，产生如下结果：

```
-------------------------------------------------
test #1: items at end
-------------------------------------------------
( 6 )( 4 3 2 1 1 1 )
Removed 3 of the value 1
( 3 )( 4 3 2 )
```

```
-----------------------------------------------------
test #2: items at front
-----------------------------------------------------
( 3 )( 1 1 1 )

Removed 3 of the value 1

( 0 )( )
-----------------------------------------------------
test #3: no items present
-----------------------------------------------------
( 3 )( 4 2 0 )

Removed 0 of the value 1

( 3 )( 4 2 0 )
-----------------------------------------------------
test #4: items at front and end
-----------------------------------------------------
( 9 )( 1 1 1 4 2 0 1 1 1 )

Removed 6 of the value 1

( 3 )( 4 2 0 )
```

我们希望提供的最后两个操作是连接操作（把一个链表附加在另一个后面）和翻转操作（翻转元素的顺序）。下面是 concat()的第一个实现，它是不正确的。你能看出问题来吗？

```
void ilist::concat( const ilist &il )
{
     if ( ! _at_end )
          _at_front = il._at_front;
     else _at_end->next( il._at_front );
          _at_end = il._at_end;
}
```

问题是，现在两个 ilist 对象指向同一个序列。如果改变一个 ilist，如 insert()或 remove()，则第二个链表也将受到影响，最简单的解决方案就是拷贝每个项。修订后的 concat()使用 insert_end()：

```
void
ilist::
concat( const ilist &il )
{
     ilist_item *ptr = il._at_front;

     while ( ptr ) {
          insert_end( ptr->value() );
          ptr = ptr->next();
     }
}
```

下面是 reverse()的实现代码：

```
void
ilist::
reverse()
{
    ilist_item *ptr = _at_front;
    ilist_item *prev = 0;

    _at_front = _at_end;
    _at_end = ptr;

    while ( ptr != _at_front )
    {
        ilist_item *tmp = ptr->next();
        ptr->next( prev );
        prev = ptr;
        ptr = tmp;
    }
    _at_front->next( prev );
}
```

下面的小程序运用了我们上面的实现：

```
#include<iostream>
#include "ilist.h"
int main()
{
    ilist mylist;

    for ( int ix = 0; ix< 10; ++ix )
        { mylist.insert_front( ix ); }

    mylist.display();

    cout << "\n" << "reverse the list\n";
    mylist.reverse(); mylist.display();
    ilist mylist_too;
    mylist_too.insert_end( 0 ); mylist_too.insert_end( 1 );
    mylist_too.insert_end( 1 ); mylist_too.insert_end( 2 );
    mylist_too.insert_end( 3 ); mylist_too.insert_end( 5 );

    cout << "\n" << "mylist_too:\n";
    mylist_too.display();
    mylist.concat( mylist_too );

    cout << "\n" << "mylist after concat with mylist_too:\n";
    mylist.display();
}
```

编译并运行该程序，产生如下输出：

```
( 10 )( 9 8 7 6 5 4 3 2 1 0 )
```

```
reverse the list

( 10 )( 0 1 2 3 4 5 6 7 8 9 )

mylist_too:

( 6 )( 0 1 1 2 3 5 )

mylist after concat with mylist_too:
( 16 )( 0 1 2 3 4 5 6 7 8 9 0 1 1 2 3 5 )
```

至此我们完成了设计与实现，不但实现了我们定义中所必需的操作，而且还对它们进行了测试以保证其正确性。缺点并不在于我们已经提供了什么，而在于我们还没有提供什么。

我们的 ilist 类最严重的缺陷是用户不能迭代链表的元素。我们实现的类太简单了，所以不支持这样的操作。而且因为我们封装了实现细节，所以用户也没有办法直接提供这样的迭代操作。

第二个不足是，这个类不支持用一个 ilist 类对象初始化或赋值给另外一个 ilist 类对象。虽然我们是有意做出这个决定的，但是用户并会不因此而减少麻烦。让我们来依次修正这些不足。首先从拷贝和初始化开始。

为了用一个对象初始化另外一个对象，我们必须定义一个 ilist 拷贝构造函数。刚开始时不允许这样做的原因是，对于我们的链表类来说缺省行为全是错误的（一般来说，对于任何包含指针成员的类，缺省行为都是错误的），“不给用户提供一个功能”比“提供一个不正确的功能导致用户程序死掉”要好得多。（缺省行为不正确的原因将在 14.5 节中解释。）拷贝构造函数使用 insert_end()：

```
ilist::ilist( const ilist &rhs )
{
    ilist_item *pt = rhs._at_front;

    while ( pt ) {
        insert_end( pt->value() );
        pt = pt->next();
    }
}
```

拷贝赋值操作符必须 remove_all()现有的项，然后按第二个 ilist 对象中值的顺序将这些值 insert_end()。因为在这两种情况下，插入部分的代码相同，所以我们可以把它抽取到成员 insert_all()中：

```
void ilist::insert_all( const ilist &rhs )
{
    ilist_item *pt = rhs._at_front;

    while ( pt ) {
        insert_end( pt->value() );
        pt = pt->next();
    }
}
```

然后如下实现拷贝构造函数和拷贝赋值操作符：

```
inline ilist::ilist( const ilist &rhs )
: _at_front( 0 ), _at_end( 0 )
{ insert_all( rhs ); }

inline ilist& ilist::operator=( const ilist &rhs ) {
    if ( this != &rhs ) {
        remove_all();
        insert_all( rhs );
    }
    return *this;
}
```

最后，用户必须能够迭代 ilist 链表中的单独元素。支持这种功能的一种策略是简单地提供对_at_front 的访问：

```
ilist_item *ilist::front() { return _at_front(); }
```

然后，用户就能够实现一般的循环迭代，就如同我们前面所做的那样：

```
ilist_item *pt = mylist.front();
while ( pt ) {
    do_something( pt->value() );
    pt = pt->next();
}
```

虽然这样做解决了问题，但是它并不是最好的方案。我们更希望支持一般化的关于容器元素迭代的概念。在这一节中，我们将提供这种形式的最小支持：

```
for ( ilist_item *iter = mylist.init_iter(); iter;
        iter = mylist.next_iter() )
    do_something( iter->value() );
```

［在第 6 章和第 12 章中，我们将会看到，标准库在支持容器类型和泛型算法的时侯定义了迭代器（iterator）类型。我们在 2.8 节中已经粗略地看到了迭代器。］

我们的迭代器不仅仅是一个指针，因为它不但能够记忆当前的迭代项，能够返回下一项，而且还能够识别出迭代的完成情况。init_iter()缺省地将 iterator 初始化指向_at_front。用户可以有选择地传递一个 ilist_item 指针，于是后面的迭代工作将从该处开始。next_iter()返回链表中的下一现 如果迭代已经完成则返问 0。为了支持迭代器，我们的 ilist 实现包含了一个附加的 ilist_iem 指针：

```
class ilist {
public:
    // ...
    init_iter( ilist_item *it = 0 );

private:
    // ...
    ilist_item *_current;
};
init_iter() looks like this:
inline ilist_item*
ilist::init_iter( ilist_item *it )
```

```
{
    return _current = it << it : _at_front;
}
```

next_item()将_current 向前移动到下一项，并将其返回，除非迭代已经完成。如果迭代已经完成，则 next_item()返回 0，直到 init_iter()重置_current 为止。下面是具体实现：

```
inline ilist_item*
ilist::
next_iter()
{
    ilist_item *next = _current
        << _current = _current->next()
        : _current;

    return next;
}
```

如果_current 指向的项已经被删除，那么我们的迭代支持就会有问题。我们的解决方案是修改 remove_front()和 remove()，使它们测试_current 是否指向被删除的项，如果是，则将_current 向前移动指向下一项（如果被删除的项是最后一项，则指向空）。如果所有的项都被删除，则_current 指向空。下面是修改后的 remove_front()：

```
inline void
ilist::remove_front()
{
    if ( _at_front ) {
        ilist_item *ptr = _at_front;
        _at_front = _at_front ->next();

        // 不希望 _current 指向被删除的项
        if ( _current == ptr )
            _current = _at_front;

        bump_down_size();
        delete ptr;
    }
}
```

下面是 remove()的部分修改代码：

```
while ( plist )
{
    if ( plist->value() == value )
    {
        prev->next( plist->next() );

        if ( _current == plist )
            _current = prev->next();
```

如果有一个项被插入到_current 指向的项的前面，又会怎么样呢？在这种情况下，我们不修改_current。为了重新同步迭代过程，用户需要调用 init_iter()。另一方面，当我们用一个 ilist 类对象初始化或拷贝给另一个对象时，_current 不会被拷贝，而是被重置为空。

下面这个小程序练习了拷贝构造函数和拷贝赋值操作符以及对迭代器的支持：

```
#include<iostream>
#include "ilist.h"

int main()
{
    ilist mylist;
    for ( int ix = 0; ix< 10; ++ix ) {
        mylist.insert_front( ix );
        mylist.insert_end( ix );
    }

    cout << "\n" << "Use of init_iter() and next_iter() "
        << "to iterate across each list item:\n";
    ilist_item *iter;

    for ( iter = mylist.init_iter();
            iter; iter = mylist.next_iter() )
        cout << iter->value() << " ";

    cout << "\n" << "Use of copy constructor\n";
    ilist mylist2( mylist );
    mylist.remove_all();

    for ( iter = mylist2.init_iter();
            iter; iter = mylist2.next_iter() )
        cout << iter->value() << " ";
    cout << "\n" << "Use of copy assignment operator\n";
    mylist = mylist2;

    for ( iter = mylist.init_iter();
            iter; iter = mylist.next_iter() )
        cout << iter->value() << " ";
    cout << "\n";
}
```

编译并运行程序，产生如下输出：

```
Use of init_iter() and next_iter() to iterate across each list item:
9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9
Use of copy constructor
9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9
Use of copy assignment operator
9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9
```

5.11.1 给出一个通用链表类

我们的 ilist 类局限在“只能拥有 int 型的元素”上，更有用的链表类应该对内置数据类型和类（class）类型都提供支持。如何改变 ilist 类，才能使其支持更广泛的元素类型，而无需重新编程或复制代码？类模板机制提供了一个解决方案（将在第 16 章详细讨论）。

通过参数化的手段，类模板抽取出我们的类设计中与类型相关的部分——在本例中，链表中元素的底层类型。以后，当用户希望使用某种特定类型的链表时，他可以为模板参数提供实际的类型。例如：

```
list< string > slist;
```

创建了一个链表模板类（list）实例，它能够包含 string 对象，而：

```
list< int > ilist;
```

创建了一个实例，它与我们原先手工编码实现的 ilist 类等价。使用类模板定义，我们可以用一个类模板实现，支持无限数目的链表元素类型。现在让我们来看一下怎样一步一步地实现它，重点放在 list_item 类上。

类模板的定义以关键字 template 开始，后面是用尖括号括起来的参数表。类型参数由 typename 或 class 加上一个标识符构成。例如：

```
template<class elemType>
class list_item;
```

它将 list_item 类声明为只有一个类型参数的类模板。elemType 是一个任意的标识符，我们用它来命名类型参数。下面是一个等价的 list_item 类声明：

```
template<typename elemType>
class list_item;
```

关键字 typename 与 class 可以互换，typename 是标准 C++中新引入的。这种写法更利于记忆，但是，在本书写作时，对 typename 的支持没有 class 广泛。由于这个原因，我们使用关键字 class，当然这也是因为老的习惯很难一下子改变过来。下面是 list_item 类模板的定义：

```
template<class elemType>
class list_item {
public:
     list_item( elemType value, list_item *item = 0 )
     : _value( value ) {
          if ( !item )
               _next = 0;
          else {
               _next = item->_next;
               item->_next = this;
          }
     }

     elemType value() { return _value; }
     list_item* next() { return _next; }
     void next( list_item *link ) { _next = link; }
     void value( elemType new_value ) { _value = new_value; }

private:
     elemType _value;
     list_item *_next;
};
```

先前 list_item 类定义中出现的 int，在我们的 list_item 类模板中全部被替换成 elemType。当我们写下这样的代码时：

```
list_item<double> *ptr = new list_item<double>( 3.14 );
```

编译器自动将 elemType 绑定到实际类型 double 上，并创建一个能够支持 double 型元素的 list_item 类。

ilist 到 list 模板类的转换以类似的方式进行。下面是类模板定义：

```
template<class elemType>
class list {
public:
    list()
    : _at_front( 0 ), _at_end( 0 ), _current( 0 ),
    _size( 0 ) {}

    list( const list& );
    list& operator=( const list& );
    ~list() { remove_all(); }

    void insert( list_item<elemType> *ptr, elemType value );
    void insert_end( elemType value );
    void insert_front( elemType value );
    void insert_all( const list &rhs );
    int remove( elemType value );
    void remove_front();
    void remove_all();

    list_item<elemType> *find( elemType value );
    list_item<elemType> *next_iter();
    list_item<elemType>* init_iter( list_item<elemType> *it );

    void display( ostream &os = cout );
    void concat( const list& );
    void reverse();
    int size() { return _size; }
private:
    void bump_up_size() { ++_size; }
    void bump_down_size() { --_size; }

    list_item<elemType> *_at_front;
    list_item<elemType> *_at_end;
    list_item<elemType> *_current;

    int _size;
};
```

模板类对象的使用方式与显式编码的 ilist 类对象完全相同。主要好处是我们能够用一个类模板定义支持无限数目的链表类型。

模板构成了标准 C++程序设计的基本组件。在第 6 章我们将了解标准库提供的模板容器类类型的集合。毫不奇怪的是，它包含了我们在第 2 章和第 3 章已经了解过的类模板 list 以及类模板 vector。

标准库 list 类的出现带来了一个矛盾。我们已经选择把我们的类称作“list”，而不是“ilist”。不幸的是，这与标准库链表类“list”冲突。现在我们不能在同一个程序中使用这两个类。当然，一种方案是重新命名我们的 list 类以便消除冲突。在目前这种情况下，这个方案是可行的，因为代码毕竟是我们自己写的。但是。在大多数情况下，这个方案并不适用。

更通用的解决方案是使用 C++名字空间机制。名字空间使得库厂商可以封装全局名字，以防止名字冲突。另外，名字空间也提供了访问符号，从而允许在我们的程序中使用这些名字。例如，C++标准库被包装在名字空间 std 中。本书第二版的代码也被放到一个唯一的名字空间中：

```
namespace Primer_Third_Edition
{
    template<typename elemType>
    class list_item{ ... };

    template<typename elemType>
    class list{ ... };
    // ...
}
```

如果用户希望练习我们的 list 类，那么他可以这样写：

```
// 我们的 list 类头文件
#include "list.h"

// 使定义对程序可见
using namespace Primer_Third_Edition;

// ok: 访问我们的 list
list< int > ilist;

// ...
```

（我们将在 8.5 节和 8.6 节详细讨论名字空间。）

练习 5.16

虽然 ilist_item 类包含了一个指针成员，但是我们并没有为它定义析构函数。原因是我们没有分配_next 指向的对象，因此也没有责任释放它。初学者容易犯的一个错误是给出一个如下定义的析构函数：

```
// a bad design choice
ilist_item::~ilist_item()
{
    delete _next;
}
```

参照 remove_all()或 remove_front()，解释为什么该析构函数的存在是个不好的设计。

练习 5.17

我们的 ilist 类不支持下列语句：

```
void ilist::remove_end();

void ilist::remove( ilist_item* );
```

你认为我们为什么没有把它们包括进去？概括出一个算法来支持这两个操作。

练习 5.18

修改 find()，使其带有第二个参数 ilist_item*。如果设置了该参数，则它指明了搜索的起始处。如果没有设置该参数，则搜索过程应当和以前一样，从链表头开始。（通过提供这个新参数，并且指定一个缺省的参数值 0，我们保留了原先的公有接口。使用前一版本 find()定义的代码不需要修改。）

```
class ilist {
public:
     // ...
     ilist_item* find( int value, ilist_item *start_at = 0 );
     // ...
};
```

练习 5.19

用新版的 find()实现 count()，它返回链表中某个值出现的次数。写一个小程序测试你的实现。

练习 5.20

修改 insert（int value）函数，返回它刚刚插入的 ilist_item 指针。

练习 5.21

利用修改过的 insert()函数，实现

```
void ilist::
insert(ilist_item *begin, int *array_of_value, int elem_cnt );
```

其中 array_of_value 指向要被插入到 ilist 中的值的数组，elem_cnt 是数组中元素的个数，begin 表明从哪里开始插入元素。例如，给出一个有下列值的 ilist：

```
(3)( 0 1 21 )
```

以及如下的数组：

```
int ia[] = { 1, 2, 3, 5, 8, 13 };
```

新的插入操作可以被调用如下：

```
ilist_item *it = mylist.find( 1 );
mylist.insert(it, ia, 6 );
```

它会把 mylist 修改成如下：

```
(9)( 0 1 1 2 3 5 8 13 21 )
```

练习 5.22

concat()和 reverse()的一个问题是它们都修改了原始的链表，但这并不总是合乎要求。请提供一对替代操作，使它们返回一个新的 ilist 对象：

```
ilist ilist::reverse_copy();
ilist ilist::concat_copy( const ilist &rhs );
```

6

抽象容器类型

本章将对第 3 章和第 4 章的内容进行扩充和完善。我们将继续在第 3 章已经开始的类型讨论，并给出关于 string 和 vector 类型更多的信息，以及 C++标准库的其他容器类型。另外，我们还将展示容器类型对象所支持的操作，以继续进行在第 4 章已经开始的、关于操作和表达式的讨论。

顺序容器（sequence container）拥有由单一类型元素组成的一个有序集合。两个主要的顺序容器是 list 和 vector。[第三个顺序容器为双端队列 deque，发音为“deck”，它提供了与 vector 相同的行为，但是对于首元素的有效插入和删除提供了特殊的支持。例如，在实现队列时（队列是一种抽象，用户每次总是获取第一个元素），deque 比 vector 更为合适。在本书的剩余部分，每当我们描述 vector 所支持的操作时，deque 也同样支持这些操作。]

关联容器（associative container）支持查询一个元素是否存在，并且可以有效地获取元素。两个基本的关联容器类型是 map（映射）和 set（集合）。map 是一个键/值（key/value）对：**键**（key）用于查询，而**值**（value）包含我们希望使用的数据。例如，map 可以很好地支持电话目录，键是人名，值是相关联的电话号码。

set 包含一个单一键值，有效支持关于元素是否存在的查询。例如，当我们要创建一个单词数据库，且它包含在某个文本中出现的单词时，文本查询系统对能会生成一个单词集合以排除 the、and 以及 but 等等。程序将顺次读取文本中的每个单词，检查它是否属于被排除单词的集合，并根据查询的结果将其丢弃或者放入数据库中。

map 和 set 都只包含每个键的惟一出现，即每个键只允许出现一次。multimap（多映射）和 multiset（多集合）支持同一个键的多次出现。例如，我们的电话目录可能需要为单个用户支持多个列表，一种实现方法是使用 multimap。

在接下去的几节中，我们将详细描述容器类型，并通过一个小的文本查询程序的循序渐进实现，对这些容器类型进行讨论。

6.1 我们的文本查询系统

文本查询系统应该由什么构成呢？

1．用户指定的任意文本文件；

2．一个逻辑查询机制，用户可以通过它查询文本中的单词或相邻的单词序列。

如果一个单词或相邻的单词序列被找到，则程序显示出该单词或单词序列的出现次数。如果用户希望的话，则包含单词或单词序列的句子也会被显示出来。例如，如果用户希望找到所有对 Civil War 或 Civil Rights 的引用，则查询可能如下[12]：

```
Civil && ( War || Rights )
```

查询结果如下：

```
Civil: 12 occurrences
War: 48 occurrences
Rights: 1 occurrence

Civil && War: 1 occurrence
Civil && Rights: 1 occurrence

(8) Civility, of course, is not to be confused with
Civil Rights, nor should it lead to Civil War.
```

这里(8)表示文本中句子的序号。我们的系统应该不会将同一个词显示多次，而且多个句子应该以升序显示（即，句子 7 应该在句子 9 之前显示）

我们的程序需要支持哪些任务呢？

1．它必须允许用户指明要打开的文本文件的名字。

2．它必须在内部组织文本文件，以便能够识别出每个单词在句子中出现的次数，以及在该句子中的位置。

3．它必须支持某种形式的布尔查询语言。在我们的例子中，它将支持：

- &&：在一行中，两个单词不仅存在，而且相邻；
- ||：在一行中，两个单词至少有一个存在；
- ！：在一行中该单词不存在；
- ()：把子查询组合起来的方式。

因此，我们可以写：

```
Lincoln
```

来找到所有出现单词 Lincoln 的句子，或者写：

```
! Lincoln
```

来找到所有没有出现单词 Lincoln 的句子，或者写：

[12] 注意：为了简化实现，我们要求用空格分割每个单词，包括括号和布尔操作符。所以，我们的程序不能理解下面的查询：

(War||Rights)

和

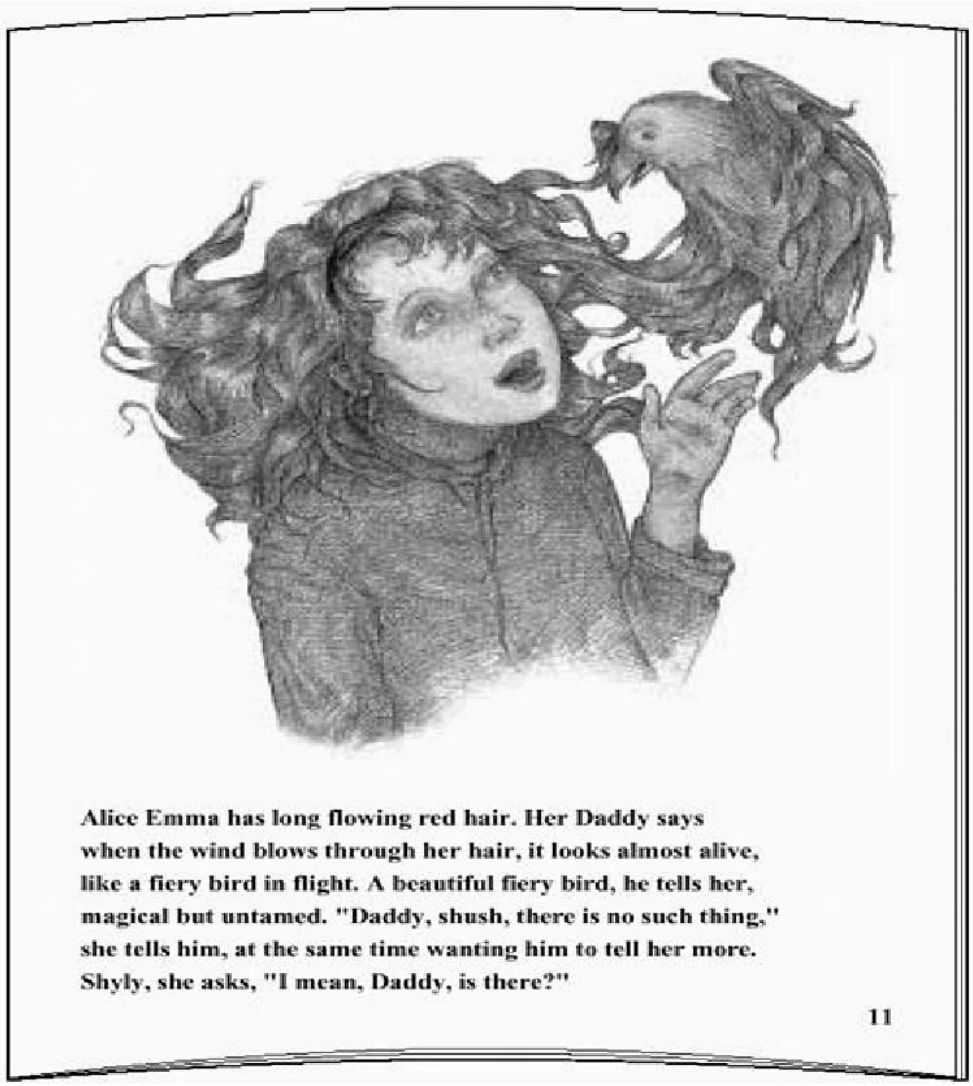
Civil&&(War||Rights)

虽然，在实际系统中，这是不合理的，因为在现实世界中，用户的便利性总是要优先于实现上的便利性。但是，在本书中，我们的目标是介绍 C++容器类型，所以这样的假设是可以被接受的。

```
( Abe || Abraham ) && Lincoln
```

将挑选出那些显式地引用 Abe Lillcoln 和 Abraham Lincoln 的句子。

我们将给出系统的两种实现。在本章中，我们给出一个实现，它把单词项及其相关联的行列位置当作一个 map，来解决文本文件的检索和存储问题。为了运用这个方案，我们提供了一个单个词的查询系统。在第 17 章中，我们将给出一个完整的咨询系统实现，它将支持如上一段讨论的那些关系操作符。之所以要到第 17 章才说明完整的实现，是因为这个方案涉及到面向对象 Query 类层次的用法。



我们使用下列六行作为文本示例，它们摘自于 Stan 写的还没有出版的儿童故事[13]：

经过我们的处理之后（这包括读入文本的每一行，把它们分成独立的单词，去掉标点符号，把大写字母变成小写，提供关于后缀的最小支持，以及去掉无语义的词比如 and、a 和 the），支持单词查询的文本的内部存储形式看起来如下所示：

```
alice ((0,0))
alive ((1,10))
almost ((1,9))
ask ((5,2))
beautiful ((2,7))
bird ((2,3),(2,9))
blow ((1,3))
```

[13] 由 Elena Driskill 插图，已获得使用许可。

```
daddy ((0,8),(3,3),(5,5))
emma ((0,1))
fiery ((2,2),(2,8))
flight ((2,5))
flowing ((0,4))
hair ((0,6),(1,6))
has ((0,2))
like ((2,0))
long ((0,3))
look ((1,8))
magical ((3,0))
mean ((5,4))
more ((4,12))
red ((0,5))
same ((4,5))
say ((0,9))
she ((4,0),(5,1))
shush ((3,4))
shyly ((5,0))
such ((3,8))
tell ((2,11),(4,1),(4,10))
there ((3,5),(5,7))
thing ((3,9))
through ((1,4))
time ((4,6))
untamed ((3,2))
wanting ((4,7))
wind ((1,2))
```

下面的简单查询示例使用了本章实现的程序（斜体为用户输入）：

```
please enter file name: alice_emma

enter a word against which to search the text.
to quit, enter a single character ==> alice

alice occurs 1 time:
     ( line 1 ) Alice Emma has long flowing red hair. Her Daddy says

enter a word against which to search the text.
to quit, enter a single character ==> daddy

daddy occurs 3 times:

     ( line 1 ) Alice Emma has long flowing red hair. Her Daddy says
     ( line 4 ) magical but untamed. "Daddy, shush, there is no such thing,"
     ( line 6 ) Shyly, she asks, "I mean, Daddy, is there?"

enter a word against which to search the text.
to quit, enter a single character ==> phoenix

Sorry. There are no entries for phoenix.
```

```
enter a word against which to search the text.
to quit, enter a single character ==> .

Ok, bye!
```

为了更容易地实现这个程序，我们需要详细地了解标准库的容器类型，同时重新回顾第3章介绍的string类。

6.2 vector还是list?

我们的程序必须要做的第一件事情是存储来自文本文件的未知数目的单词。这些单词将被依次存储为string对象。我们的第一个问题是，应该把单词存储在顺序容器还是关联容器中？

从某种意义上讲，我们需要支持查询单词是否存在，如果存在的话，还要获取到它在文本中相关的出现位置。因为我们需要查找并获取一个值，所以关联容器map是最合适的容器类型。

但是，在此之前，我们只需要简单地把输入文本存储起来以供后续处理（即去掉标点符号、处理后缀等等）。对于这个前处理过程，我们只需要顺序容器，而不是关联容器。问题是，它应该是vector还是list？

如果曾经在C语言中或在C++标准化之前编写过程序，那么你的第一个选择可能是：如果在编译时元素的个数已知，则使用数组。如果元素的个数未知或者可能变化范围很大，则使用list，为每个对象动态分配存储区，然后再把它们按顺序链接在list中。

但是，这样的选择规则对于顺序容器类型并不适用：vector、deque以及list都是动态增长的。在这三者之中选择的准则主要是关注插入特性以及对元素的后续访问要求。

vector表示一段连续的内存区域，每个元素被顺序存储在这段内存中。对vector的随机访问（比如先访问元素5，然后访问15，然后再访问7等等）效率很高，因为每次访问离vector起始处的位移都是固定的。但是，在任意位置，而不是在vector末尾插人元素，则效率很低，因为它需要把待插入元素右边的每个元素都拷贝一遍。类似地，删除任意一个，而不是vector的最后一个元素，效率同样很低，因为待删除元素右边的每个元素都必须被复制一遍。这种代价对于大型的、复杂的类对象来说尤其大。（一个deque也表示一段连续的内存区域，但是，与vector不同的是，它支持高效地在其首部插入和删除元素。它通过两级数组结构来实现，一级表示实际的容器，第二级指向容器的首和尾。）

list表示非连续的内存区域，并通过一对指向首尾元素的指针双向链接起来，从而允许向前和向后两个方向进行遍历。在list的任意位置插入和删除元素的效率都很高：指针必须被重新赋值，但是，不需要用拷贝元素来实现移动。另一方面，它对随机访问的支持并不好：访问一个元素需要遍历中间的元素。另外，每个元素还有两个指针的额外空间开销。

下面是选择顺序容器类型的一些准则：

- 如果我们需要随机访问一个容器，则vector要比list好得多。
- 如果我们已知要存储元素的个数，则vector又是一个比list好的选择。
- 如果我们需要的不只是在容器两端插入和删除元素，则list显然要比vector好。

- 除非我们需要在容器首部插入和删除元素，否则 vector 要比 deque 好。

如果我们既需要随机访问元素，又需要随机插入和删除元素，那么又该怎么办呢？我们需要在“随机访问的代价”和“拷贝右边或左边相邻元素的代价”之间进行折衷。一般来说，应该是由应用程序的主要操作（查找或插入）来决定容器类型的选择。（为了做这个决定，我们可能需要知晓两种容器类型的性能。）如果两种容器类型的性能都不能够使我们满意，则需要自己设计更复杂的数据结构。

当我们不知道需要存储的元素的个数（即容器需要动态增长），并且不需要随机访问元素，以及在首部或者中间插入元素时，我们该如何决定选择哪一个容器类型呢？在这种情况下，list 和 vector 哪一个更有效？（我们将把这个问题的答案推延到下一节。）

list 以简单方式增长：每当一个新对象被插入到 list 中时，插入处的两个元素的前指针和后指针被重新赋值为指向新对象。新对象的前后指针被初始化为指向这两个元素。list 只占有其包含的元素所必需的存储区。额外的开销有两个方面：与每个值相关联的两个附加指针，以及通过指针进行的间接访问。

动态 vector 的表示和额外开销更加复杂。我们将在下一节介绍它。

练习 6.1

对于以下程序任务，vector、deque 和 list，哪一个最合适？或者都不合适？

(a) 为了生成随机的英文句子，从一个文件读入未知数目的单词；
(b) 读入固定数目的单词，在输入时把它们按字母顺序插入到容器中；
(c) 读入未知数目的单词。总是在后面插入一个新单词。从头删除下一个元素；
(d) 从文件读入未知数目的整数。对这些整数排序，然后把它们输出到标准输出。

6.3 vector 怎样自己增长

一个需要动态增长的 vector 必须分配一定的内存以便保存新的序列、按顺序拷贝旧序列的元素以及释放旧的内存。而且，如果它的元素是类对象，那么拷贝和释放内存可能需要对每个元素依次调用拷贝构造函数和析构函数。冈为 list 的每次增长，只是简单地链接新元素，所以看起来好像没有问题。在动态增长的支持方面，这两个容器类型之中 list 更为有效。但实际上并不是这样。让我们来看看为什么。

为了提高效率，实际上 vector 并不是随每一个元素的插入而增长自己，而是当 vector 需要增长自身时，它实际分配的空间比当前所需的空间要多一些，也就是说，它分配了一些额外的内存容量，或者说它预留了这些存储区（分配的额外容量的确切数目由具体实现定义）。这个策略使容器的增长效率更高——因此，实际上，对于小的对象，vector 在实践中比 list 效率更高。让我们来看一看在 C++标准库的 Rogue Wave 实现版本下的一些例子。但是首先我们要弄清楚容器的容量和长度（size）之间的区别。

容量是指在容器下一次需要增长自己之前能够被加入到容器中的元素的总数（容量只与连续存储的容器相关：例如 vector、deque 或 string。list 不要求容量）。为了知道一个容器的容量，我们调用它的 capacity()操作。而长度（size）是指容器当前拥有元素的个数。为了获得容器的当前长度，我们调用它的 size()操作。例如：

```
#include <vector>
#include <iostream>

int main()
{
    vector< int > ivec;
    cout << "ivec: size: " << ivec.size()
         << " capacity: " << ivec.capacity() << endl;

    for ( int ix = 0; ix < 24; ++ix ) {
        ivec.push_back( ix );
        cout << "ivec: size: " << ivec.size()
             << " capacity: " << ivec.capacity() << endl;
    }
}
```

在 Rogue Wave 实现版本下，在 ivec 的定义之后，它的长度和容量都是 0。但是在插入第一个元素之后，ivec 的容量是 256，长度为 1、这意味着在 ivec 下一次需要增长之前。我们可以向它加入 256 个元素。当我们插入第 256 个元素时，vector 以下列方式重新自我增长：它分配双倍于当前容量的存储区，把当前的值拷贝到新分配的内存中，并释放原来的内存。正如稍后我们将要看到的，同 list 相比，数据类型越大越复杂，则 vector 的效率也就越低。表 6.1 显示了各种数据类型。它们的长度，及其相关 vector 的初始容量：

表格 6.1 长度、容量、以及各种数据类型

数据类型	长度（字节）	初始插入后的容量
int	4	256
double	8	128
简单类（simple class） #1	12	85
String	12	85
大型简单类（large simple class）	8000	1
大型复杂类（large complex class）	8000	1

正如你所看到的，在标准库的 Rogue Wave 实现版本下，在第一次插入时分配的元素缺省容量接近或等于 1024 字节，然后它随每次重分配而加倍。对于大型的数据类型，容量较小，所以元素的重分配和拷贝操作成为使用 vector 的主要开销（我们这里说的复杂类是指这类提供了一个拷贝构造函数和一个拷贝赋值操作符）。表 6.2 显示了在 list 和 vector 中插入 1 千万个上述类型所花的时间（以秒为单位）。表 6.3 显示了插入 1 万个元素的时间（当元素长度较大时就太慢了。）

表格 6.2 插入 1 千万个元素所需的时间

数据类型	list(s)	vector
int	1038	3.76
double	10.72	3.95
简单的类（simpleclass）	12.31	5.89
string	14.42	11.80

表格 6.3 插入 1 万个元素的时间

数据类型	list(s)	vector
大型简单类（large simple class）	0.36	2.23
大型复杂类（large complex class）	2.37	6.70

正如你所看到的，对于小的数据类型，vector 的性能要比 list 好得多，而对于大型的数据类型则相反，list 的性能要好得多。区别是由于 vector 需要重新增长以及拷贝元素。但是，数据类型的长度不是影响容器性能的惟一标准。数据类型的复杂性也会影响到元素插入的性能。为什么？

无论是 list 还是 vector，对于已定义拷贝构造函数的类来说，插入这样的类的元素都需要调用拷贝构造函数。（拷贝构造函数用该类型的一个对象初始化该类型的另一个对象——最初的讨论见 2.2 节，详细讨论见 14.5 节。）这正说明了在简单类和 string 的链表之间插入代价的区别。简单类对象和大型简单类对象通过按位拷贝插入（一个对象的所有位被拷贝到第二个对象的位中），而 string 类对象和大型复杂类对象通过调用拷贝构造函数来插入。

另外，随着每次重新分配内存，vector 必须为每个元素调用拷贝构造函数。而且，在释放原来的内存时，它要为每个元素调用其相关类型的析构函数（关于析构函数的最初讨论见 2.2 节）。vector 的动态自我增长越频繁，元素插入的开销就越大。

当然，一种解决方案是当 vector 的开销变得非常大时，把 vector 转换成 list。另一种经常使用的方案是，通过指针间接存储复杂的类对象。例如，当我们用指针存储复杂类对象时，在 vector 中插入 10000 个元素的开销从 6.70s 明显地降到 0.82s。为什么？首先，容量从 1 增加到 256，因此重新分配的次数大大减少。其次，指向类对象的指针的拷贝和释放不需要调用该类的拷贝构造函数和析构函数。

reserve()操作允许程序员将容器的容量设置成一个显式指定的值，例如：

```
int main() {
    vector< string > svec;
    svec.reserve( 32 ); // 把容量设置为 32
    // ...
}
```

使 svec 的长度为 0（即 0 个元素），而容量为 32。但是，根据经验发现，用一个非 1 的

缺省值来调整 vector 的容量看起来总会引起性能退化。例如，对于 string 和 double 型的 vector，通过 reserve()增加容量导致很差的性能。另一方面，增加大型复杂类的容量，会大大改善了性能，如表 6.4 所示。

表格 6.4 调整容量时插入 10000 元素的时间

容量	时间（s）
缺省值 1	6.70
4096	5.55
8192	4.44
10000	2.22

注：非简单类：8000 字节大小，并且带有构造函数和析构函数。

对于我们的文本查询系统，将使用一个 vector 来包含 string 对象，并且使用与它相关联的缺省容量。虽然当我们插入未知数目的 string 时，vector 会动态增长，但是正如前面显示的计时情况来看，它的性能仍然要比 list 好一些。在开始真正的实现之前，我们先来回顾一下怎样定义一个容器对象。

练习 6.2

解释容器的容量与长度（size）之间的区别。为什么在连续存储的容器中需要支持容量的概念，而非连续的容器（比如 list）则不需要？

练习 6.3

为什么用指针存储大型复杂类对象的集合效率更高，而用指针存储整型对象的集合效率却比较低？

练习 6.4

在下列情况下，list 和 vector 中哪一个是比较合适的容器类型？在每一种情况下，插入元素的数目都是未知的。请解释你的答案。

(a) 整型值；
(b) 指向大型、复杂类对象的指针；
(c) 大型、复杂类对象。

6.4 定义一个顺序容器

为了定义一个容器对象，我们必须先包含相关联的头文件，应该是下列头文件之一：

```
#include <vector>
#include <list>
#include <deque>
#include <map>
#include <set>
```

容器对象的定义以容器类型的名字开始，后面是所包含的元素的实际类型[14]，例如：

```
vector< string > svec;
list< int > ilist;
```

定义了 svec 是一个内含 string 对象的主 vector，以及 ilist 是一个 int 型对象的空 list。svec 和 ilist 都是空的。为了确认这一点，我们可以调用 empty()操作符。例如：

```
if ( svec.empty() != true )
     ;    // 喔，有错误了
```

插入元素最简单的方法是 push_back()，它将元素插入在容器的尾部。例如：

```
string text_word;

while ( cin >> text_word )
     svec.push_back( text_word );
```

每次从标准输入读取一个字符串放到 text_word 中。然后 push_back()再将 text_word 字符
串的拷贝插入到 svec 中。list 和 deque 容器也支持 push_front()，它把新元素插入在链表的前端。例如，假设我们有一个 int 型的内置数组如下：

```
int ia[ 4 ] = { 0, 1, 2, 3 };
```

用 push_back()：

```
for ( int ix = 0; ix < 4; ++ix )
     ilist.push_back( ia[ ix ] );
```

创建序列 0，1，2，3，然而，如果使用 push_front()：

```
for ( int ix = 0; ix < 4; ++ix )
     ilist.push_front( ia[ ix ] );
```

则在 ilist 中创建序列 3，2，1，0[15]。

另外，我们或许希望为容器指定一个显式的长度。长度可以是常量，也可以是非常量表达式：

```
#include <list>
#include <vector>
#include <string>

extern int get_word_count( string file_name );
const int list_size = 64;

list< int > ilist( list_size );
```

[14] 如果一个 C++编译器不支持缺省模板参数，那么它要求第二个实参指定分配器（allocator）。在这样的编译器实现下，上述两个定义被声明如下：

```
list< int > ilist( list_size, - 1 );
vector< string > svec( 24, "pooh" );
```

allocator 类封装了分配和删除动态内存的抽象过程。它也是标准库预定义的类，实际上它使用了 new 和 delete 操作符。使用这样的分配器类有两个目的：通过把容器与内存分配策略的细节分开，这可以简化容器类的实现。其次，程序员有可能实现或者指定其他的内存分配策略，比如使用共享内存。

[15] 如果容器的主要行为是在前端插入元素，则 deque 比 vector 的效率高，所以我们应该优先选择 deque。

```
vector< string > svec(get_word_count(string("Chimera")));
```

容器中的每个元素都被初始化为“与该类型相关联的缺省值”。对于整数，将用缺省值 0 初值化所有元素。对于 string 类，每个元素都将用 string 的缺省构造函数初始化。

除了用相关联的初始值来初始化每个元素外，我们还可以指定一个值，并用它来初始化每个元素。例如：

```
list< int > ilist( list_size, - 1 );
vector< string > svec( 24, "pooh" );
```

除了给出初始长度外，我们还可以通过 resize()操作重新设置容器的长度。例如，当我们写下面的代码时：

```
svec.resize( 2 * svec.size() );
```

我们将 svec 的长度加了一倍。每个新元素都被初始化为“与元素底层类型相关联的缺省值”。如果我们希望把每个新元素初始化为某个其他值，则可以把该值指定为第二个参数：

```
// 将新元素初始化为“piglet”
svec.resize( 2 * svec.size(), "piglet" );
```

那么，svec 的原始定义的容量是多少？它的初始长度是 24 个元素。它的初始容量可能是多少？对——svec 的容量也是 24。一般地，vector 的最小容量是它的当前长度，当 vector 的长度加倍时，容量一般也加倍。

我们也可以用一个现有的容器对象初始化一个新的容器对象。例如：

```
vector< string > svec2( svec );
list< int > ilist2( ilist );
```

每个容器支持一组关系操作符，我们可以用来比较两个容器，这些关系操作符分别是：等于、不等于、小于、大于、小于等于，以及大于等于。容器的比较是指两个容器的元素之间成对进行比较。如果所有元素相等而且两个容器含有相同数目的元素，则两个容器相等。否则，它们不相等。第一个不相等元素的比较决定了两个容器的小于或大于关系。例如，下面是一个程序的输出，它比较了五个 vector：

```
ivec1: 1 3 5 7 9 12
ivec2: 0 1 1 2 3 5 8 13
ivec3: 1 3 9
ivec4: 1 3 5 7
ivec5: 2 4

// 第一个不相等元素：1，0
// ivec1 大于 ivec2
ivec1 < ivec2 // flase
ivec2 < ivec1 // true

// 第一个不相等元素：5，9
ivec1 < ivec3 // true

// 所有元素相等，但是，ivec4 的元素少
// 所以 ivec4 小于 ivec1
ivec1 < ivec4 // false
```

```
// 第一个不相等元素：1，2
ivec1 < ivec5 // true
ivec1 == ivec1    // true
ivec1 == ivec4    // false
ivec1 != ivec4    // true
ivec1 > ivec2 // true
ivec3 > ivec1 // true
ivec5 > ivec2 // true
```

我们能够定义的容器的类型有三个限制（实际上，它们只适用于用户定义的类类型）：

- 元素类型必须支持等于操作符；
- 元素类型必须支持小于操作符（前面讨论的所有关系操作符都用这两个操作符来实现）；
- 元素类型必须支持一个缺省值（对于类类型，即指缺省构造函数）。

所有预定义数据类型，包括指针，都满足这些限制，C++标准库给出的所有类类型也一样。

练习 6.5

说明下列代码所做的事情：

```
#include <string>
#include <vector>
#include <iostream>

int main()
{
    vector<string> svec;
    svec.reserve( 1024 );
    string text_word;

    while ( cin >> text_word )
        svec.push_back( text_word );
    svec.resize( svec.size()+svec.size()/2 );
    // ...
}
```

练习 6.6

容器的容量可以小于它的长度吗？容量能等于它期望的长度吗？初始化时相等吗？在一个元素被插入之后呢？为什么？

练习 6.7

在练习 6.5 中，如果程序读入 256 个单词，在它被重新设置长度后可能的容量是多少？如果读入 512 个呢？1000 个？1048 个单词呢？

练习 6.8

已知如下类定义，请指出哪些类不能用来定义 vector?

```
(a) class cl1 {
    public:
        cl1( int=0 );
        bool operator==();
        bool operator!=();
        bool operator<=();
        bool operator<();
        // ...
    };
(c) class cl3 {
    public:
        int ival;
    };
```

```
(b) class cl2 {
    public:
        cl2( int=0 );
        bool operator!=();
        bool operator<=();
        // ...
    };

(d) class cl4 {
    public:
        cl4( int, int=0 );
        bool operator==();
        bool operator==();
        // ...
    };
```

6.5 迭代器

迭代器（iterator）提供了一种一般化的方法，对顺序或关联容器类型中的每个元素进行连续访问。例如，假设 iter 为任意容器类型的一个 iterator，则：

```
++iter;
```

向前移动迭代器，使其指向容器的下一个元素，而：

```
*iter;
```

返回 iterator 指向元素的值。

每种容器类型都提供一个 begin()和一个 end()成员函数。

- begin()返回一个 iterator，它指向容器的第一个元素。
- end()返回一个 iterator，它指向容器的末元素的下一个位置。

为了迭代任意容器类型的元素，我们可以这样写：

```
for ( iter = container.begin();
          iter != container.end(); ++iter )
     do_something_with_element( *iter );
```

由于模板和嵌套类语法的原因，iterator 的定义看起来有点吓人。例如，下面是一对 iterator 的定义，它们指向一个内含 string 元素的 vector：

```
// vector<string> vec;
vector<string>::iterator iter = vec.begin();
vector<string>::iterator iter_end = vec.end();
```

iterator 是 vector 类中定义的 typedef。以下语法：

```
vector<string>::iterator
```

引用了 vector 类中内嵌的 iterator typedef，并且该 vector 类包含 string 类型的元素。

为了把每个 string 元素打印到标准输出上，我们可以这样写：

```
for( ; iter != iter_end; ++iter )
    cout << *iter << '\n';
```

当然，这里*iter 的运算结果就是实际的 string 对象。

除了 iterator 类型，每个容器还定义了一个 const iterator 类型，后者对于遍历 const 容器是必需的。const iterator 允许以只读方式访问容器的底层元素。例如：

```
#include <vector >
void even_odd( const vector<int> *pvec,
               vector<int> *pvec_even,
               vector<int> *pvec_odd )
{

    // 必须声明一个 const_iterator，才能够遍历 pvec
    vector<int>::const_iterator c_iter = pvec->begin();
    vector<int>::const_iterator c_iter_end = pvec->end();

    for ( ; c_iter != c_iter_end; ++c_iter )
        if ( *c_iter % 2 )
            pvec_odd->push_back( *c_iter );
    else pvec_even->push_back( *c_iter );
}
```

最后，如果我们希望查看这些元素的某个子集，该怎么办呢？如每隔一个元素或每隔三个元素，或者从中间开始逐个访问元素。我们可以用标量算术运算使 iterator 从当前位置偏移到某个位置上。例如：

```
vector<int>::iterator iter = vec.begin()+vec.size()/2;
```

将 iter 指向 vec 的中间元素，而：

```
iter += 2;
```

将 iter 向前移动两个元素。

iterator 算术论算只适用于 vector 或 deque，而不适用于 list，因为 list 的元素在内存中不是连续存储的。例如：

```
ilist.begin() + 2;
```

是不正确的，因为在 list 中向前移动两个元素需要沿着内部 next 指针走两次。对于 vector 或 deque，前进两个元素需要把当前的地址值加上两个元素的长度（3.3 节给出了关于指针算术运算的讨论）。

容器对象也可以用“由一对 iterator 标记的起始元素和未元素后一位置之间的拷贝”来初始化。例如，假设我们有：

```
#include <vector>
#include <string>
#include <iostream>

int main()
{
    vector<string> svec;
```

```
        string intext;
        while ( cin >> intext )
            svec.push_back( intext );

        // process svec ...
    }
```

我们可以定义一个新的 vector 来拷贝 svec 的全部或部分元素：

```
int main()
{
    vector<string> svec;
    // ...
    // 用 svec 的全部元素初始化 svec2
    vector<string> svec2( svec.begin(), svec.end() );

    // 用 svec 的前半部分初始化 svec3
    vector<string>::iterator it =
    svec.begin() + svec.size()/2;
    vector<string> svec3( svec.begin(), it );
    // 处理 vectors ...
}
```

用特定的 istream_iterator 类型（12.4.3 节详细讨论），我们可以更直接地向 svec 插入文本元素：

```
#include <vector>
#include <string>
#include < iterator >

int main()
{
    // 将输入流 iterator 绑定到标准输入上
    istream_iterator<string> infile( cin );

    // 标记流结束位置的输入流 iterator
    istream_iterator<string> eos;

    // 利用通过 cin 输入的值初始化 svec
    vector<string> svec( infile, eos );

    // 处理 svec
}
```

除了一对 iterator 之外，两个指向内置数组的指针也可以被用作元素范围标记器（range marker）。例如，假设我们有下列 string 对象的数组：

```
#include <string>
string words[4] = {
    "stately", "plump", "buck", "mulligan"
};
```

我们可以通过传递数组 words 的首元素指针和末元素后一位置的指针来初始化 string

vector：

```
vector< string > vwords( words, words+4 );
```

第二个指针被用作终止条件：它指向的对象（通常指向容器或者数组中最后一个元素后面的位置上）不包含在要被拷贝或遍历的元素之中。

类似地，我们可以按如下方式初始化一个内含 int 型元素的 list：

```
int ia[6] = { 0, 1, 2, 3, 4, 5 };
list< int > ilist( ia, ia+6 );
```

在 12.4 节中，我们将进一步介绍 iterator。现在，我们所介绍的已经足够我们在文本查询系统中使用它们了。但是，在回到文本查询系统的实现之前，我们需要复习一下容器类型支持的一些其他操作。

练习 6.9

下列 iterator 的用法哪些是错误的？

```
const vector< int > ivec;
vector< string > svec;
list< int > ilist;

(a)  vector<int>::iterator it = ivec.begin();
(b)  list<int>::iterator it = ilist.begin()+2;
(c)  vector<string>::iterator it = &svec[0];
(d)  for ( vector<string>::iterator
               it = svec.begin(); it != 0; ++it )
         // ...
```

练习 6.10

下列 iterator 的用法哪些是错误的？

```
int ia[7] = { 0, 1, 1, 2, 3, 5, 8 };
string sa[6] = {
     "Fort Sumter", "Manassas", "Perryville", "Vicksburg",
     "Meridian", "Chancellorsville" };

(a) vector<string> svec( sa, &sa[6] );
(b) list<int> ilist( ia+4, ia+6 );
(c) list<int> ilist2( ilist.begin(), ilist.begin()+2 );
(d) vector<int> ivec( &ia[0], ia+8 );
(e) list<string> slist( sa+6, sa );
(f) vector<string> svec2( sa, sa+6 );
```

6.6 顺序容器操作

push_back()方法给出了“在顺序容器尾部插入单个元素”的简短表示。但是，如果我们希望在容器的其他位置插入元素，该怎么办呢？或者，如果我们希望在容器的尾部或某个其他位置插入一个元素序列，该怎么办呢？在这些情况下，我们将使用一组更通用的插

入方法。

例如，为了在容器的头部插入元素，我们将这样做：

```
vector< string > svec;
list< string > slist;

string spouse( "Beth" );
slist.insert( slist.begin(), spouse );
svec.insert( svec.begin(), spouse );
```

这里，insert()的第一个参数是一个位置（指向容器中某个位置的 iterator），第二个参数是将要被插入的值，这个值被插入到由 iterator 指向的位置的前面。更为随机的插入操作可以如下实现：

```
string son( "Danny" );
list<string>::iterator iter;

iter = find( slist.begin(), slist.end(), son );
slist.insert( iter, spouse );
```

这里，find()返回被查找元素在容器中的位置，或者返回容器的 end() iterator，表明这次查询失败（我们将在下一节结束时回头介绍 find()）。正如你所猜到的，push_back()是下列调用的简短表示：

```
// 等价于: slist.push_back( value );
slist.insert( slist.end(), value );
```

insert()方法的第二种形式支持“在某个位置插入指定数量的元素”。例如，如果希望在 vector 的开始处插入 10 个 Anna，我们可以这样做：

```
vector<string> svec;
...
string anna( "Anna" );
svec.insert( svec.begin(), 10, anna );
```

insert()方法的最后一种形式支持“向容器插入一段范围内的元素”。例如，给出下列 string 数组：

```
string sarray[4] = { "quasi", "simba", "frollo", "scar" };
```

我们可以向字符串 vector 中插入数组中的全部或部分元素：

```
svec.insert( svec.begin(), sarray, sarray+4 );

svec.insert( svec.begin() + svec.size()/2,
     sarray+2, sarray+4);
```

另外，我们还可以通过一对 iterator 来标记出待插入值的范围，可以是另一个 string 元素的 vector：

```
// 插入 svec 中含有的元素，
// 从 svec_two 的中间开始
svec_two.insert( svec_two.begin() + svec_two.size()/2,
     svec.begin(), svec.end() );
```

或者，更一般的情况下，也可以是任意一种 string 对象的容器[16]：

```
list< string > slist;

// ...
// 把 svec 中含有的元素插入到
// slist 中 stringVal 的前面
list< string >::iterator iter =
     find( slist.begin(), slist.end(), stringVal );
slist.insert( iter, svec.begin(), svec.end() );
```

6.6.1 删除操作

删除容器内元素的一般形式是一对 erase()方法：一个删除单个元素，另一个删除由一对 iterator 标记的一段范围内的元素。删除容器末元素的简短方法由 pop_back()方法支持。

例如，为了删除容器中一个指定的元素，我们可以简单地调用 erase()，用一个 iterator 表示它的位置。在下列代码段中，我们用 find()泛型算法获得待删除元素的 iterator，如果 list 中存在这样的元素，则把它的位置传递给 erase()：

```
string searchValue( "Quasimodo" );

list< string >::iterator iter =
     find( slist.begin(), slist.end(), searchValue );

if ( iter != slist.end() )
     slist.erase( iter );
```

要删除容器中的所有元素，或者由 iterator 对标记的子集，我们可以这样做：

```
// 删除容器中的所有元素
slist.erase( slist.begin(), slist.end() );

// 删除由 iterator 标记的一段范围内的元素
list< string >::iterator first, last;
first = find( slist.begin(), slist.end(), val1 );
last = find( slist.begin(), slist.end(), val2 );

// ... 检查 first 和 last 的有效性
slist.erase( first, last );
```

最后，如同在容器尾部插入一个元素的 push_back()方法相仿，pop_back()方法删除容器的末元素——它不返回元素，只是简单地删除它。例如：

```
vector< string >::iterator iter = buffer.begin();

for ( ; iter != buffer.end(); iter++ )
{
     slist.push_back( *iter );
     if ( ! do_something( slist ))
          slist.pop_back();
}
```

[16] 最后一种形式要求编译器支持模板成员函数。如果你的编译器还不支持标准 C++的这种特性，那么两种容器的类型必须相同，比如内含相同类型元素的两个 vector 或 list。

6.6.2 赋值和对换

当我们把一个容器类型赋值给另一个时，会发生什么事情？赋值操作符使用针对容器元素类型的赋值操作符，把右边容器对象中的元素依次拷贝到左边的容器对象中。如果两个容器的长度不相等，又会怎么样呢？例如：

```
// slist1 含有 10 个元素
// slist2 含有 24 个元素
// 赋值之后都含有 24 个元素
slist1 = slist2;
```

赋值的目标，在本例中为 slist1，它现在拥有与被拷贝容器（本例中为 slist2）相同的元素数目。slist1 中的前 10 个元素被删除（在 slist1 的情况下，string 的析构函数被应用在这 10 个 string 元素上）。

swap()可以被看作是赋值操作符的互补操作。当我们写：

```
slist1.swap( slist2 );
```

时，slist1 现在含有 24 个 string 元素，是用 string 赋值操作符拷贝的，就如同我们写：

```
slist1 = slist2;
```

一样。

两者的区别在于，slist2 现在含有 slist1 中原来含有的 10 个元素的拷贝。如果两个容器的长度不同，则容器的长度就被重新设置，且等于被拷贝容器的长度。

6.6.3 泛型算法

上节描述的操作都是 vector 和 deque 容器提供的基本操作。毫无疑问，那只是一个相当小的接口，它省略了像 find()、sort()和 merge()等基本操作。从概念上讲，我们的思想是把所有容器类型的公共操作抽取出来，形成一个通用算法集合，它能够被应用到全部容器类型以及内置数组类型上。这组通用算法被称作泛型算法（泛型算法将在 12 章和附录中详细讨论）。泛型算法通过一个 iterator 对，被绑定到一个特殊的容器上。例如，下面的代码显示了我们怎样在一个 list、vector，以及不同类型的数组上调用 find()泛型算法：

```
#include <list>
#include <vector>

int ia[ 6 ] = { 0, 1, 2, 3, 4, 5 };
vector<string> svec;
list<double> dlist;

// the associated header file
#include <algorithm>

vector<string>::iterator viter;
list<double>::iterator liter;
int *pia;

// 如果找到，find()返回指向该元素的 iterator
// 对于数组，返回指针
```

```
pia = find( &ia[0], &ia[6], some_int_value );
liter = find( dlist.begin(), dlist.end(), some_double_value );
viter = find( svec.begin(), svec.end(), some_string_value );
```

因为 list 容器类型不支持随机访问其元素，所以它提供了额外的操作，如 merge()和 sort()。这些都将在 12.6 节中讨论。现在，我们回头来看我们的文本查询系统。

练习 6.11

请写一个程序，使它接受下列定义：

```
int ia[] = { 1, 5, 34 };
int ia2[] = { 1, 2, 3 };
int ia3[] = { 6, 13, 21, 29, 38, 55, 67, 89 };
vector<int> ivec;
```

用各种插入操作，以及 ia2 和 ia3 适当的值修改 ivec，使它拥有序列：

```
{ 0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 89 }
```

练习 6.12

请写一个程序，使它接受下列定义：

```
int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 89 };
list<int> ilist( ia, ia+11 );
```

用单个 iterator 形式的 erase()删除 ilist 中所有奇数位置的元素。

6.7 存储文本行

我们的第一个任务是读入用户需要查询的文本文件。需要获得下列信息：每个单词，当然，还有每个单词的位置——即，它在哪一行，以及在该行的位置。而且，为了显示出与一个查询相匹配的文本行，我们必须按行号保留文本。

怎样获得文本的每一行呢？标准库支持 getline()函数，声明如下：

```
istream&
    getline( istream &is, string str, char delimiter );
```

getline()读取 istream 对象，向 string 对象插入字符，包括空格，直到遇到分割符、文件结束，或者被读入的字符序列等于 string 对象的 max_size()值，在该点处读入操作失败。

在每次调用 getline()之后，我们都会将 str 插入到代表文本的字符串 vector 中。下面是一般化的实现[17]。我们已经将它提取到一个函数中，命名为 retrieve_text()。为了增加被收集到的信息，我们定义了一对值来存储最长行的行数和长度（完整的程序列表在 6.14 节）：

---

[17] 它是在不支持缺省模板参数值的编译器下被编译的，所以我们必须显式提供一个分配器（allocator）：

```
vector< string, allocator > *lines_of_text;
```

在一个完全支持标准 C++的编译器下面。我们只需要指定元素的类型：

```
vector< string > *lines_of_text;
```

```
// 返回值是指向 string vector 的指针
vector<string,allocator>*
retrieve_text()
{
    string file_name;
    cout << "please enter file name: ";
    cin >> file_name;

    // 打开文本文件以便输入 ...
    ifstream infile( file_name.c_str(), ios::in );
    if ( ! infile ) {
        cerr << "oops! unable to open file "
             << file_name << " -- bailing out!\n";
        exit( -1 );
    }
    else cout << '\n';
    vector<string, allocator> *lines_of_text =
        new vector<string,allocator>;
    string textline;
    typedef pair<string::size_type, int> stats;
    stats maxline;
    int linenum = 0;

    while ( getline( infile, textline, '\n' )) {
        cout << "line read: " << textline << '\n';
        if ( maxline.first << textline.size() ) {
            maxline.first = textline.size();
            maxline.second = linenum;
        }

        lines_of_text->push_back( textline );
        linenum++;
    }
    return lines_of_text;
}
```

程序的输出如下（不幸的是，由于页面尺寸限制的原因，我们已经手工做了整理以便于阅读。）：

```
please enter file name: alice_emma

line read: Alice Emma has long flowing red hair. Her Daddy says
line read: when the wind blows through her hair, it looks almost
        alive,
line read: like a fiery bird in flight. A beautiful fiery bird, he
        tells her,
line read: magical but untamed. "Daddy, shush, there is no such
        thing,"
line read: she tells him, at the same time wanting him to tell her
        more.
line read: Shyly, she asks, "I mean, Daddy, is there?"
```

```
number of lines: 6
maximum length: 66
longest line: like a fiery bird in flight. A beautiful fiery bird,
         he tells her,
```

由于每个文本行都是作为 string 被存储的，所以我们需要把每行拆成独立的单词。对于每个单词，我们将首先去掉标点符号。例如，考虑下面来自 Finnegans Wake 的 Anna Livia Plurrabelle 片断的行。

```
"For every tale there's a telling,
and that's the he and she of it."
```

它产生下列单独的 string，这些 string 可能带有内嵌的标点符号：

```
"For
there's
telling,
that's
it."
```

这些 string 需要被变成：

```
For
there
telling
that
it
```

有人可能会说：

```
there's
```

应该变成：

```
there is
```

但是实际上我们打算走向另一个方向：我们将丢弃没有语义的单词，如 is、that、and、it 以及 the 等等。因此，对摘自《Finnegans Wake》的行，我们用来查询的活动单词只有以下两个会被输入：

```
tale
telling
```

（我们将用一个专门排除单词的 set 来实现这一点，这将在后面关于 set 容器类型的章节中详细讨论。）

除了去掉标点符号外，我们还需要去掉大写字母，并提供对后缀的最小处理。大写字母在下列文本行中成为一个问题：

```
Home is where the heart is.
A home is where they have to let you in.
```

显然，关于 home 的查询需要找到两次。

对于后缀的处理是要解决一个更为复杂的识别问题，例如，识别 dog 和 dogs 表示相同的名词，love、loves 以及 loving 表示同一动词。

下一节的目的是重新回顾标准库 string 类，练习 string 处理操作的扩展集合。沿着这条路，我们将进一步开发我们的文本查询系统。

6.8 找到一个子串

我们的第一个任务是将表示文本行的字符串分解成独立的单词。我们将通过找到内嵌的空格来达到这个目的。例如，给出：

```
Alice Emma has long flowing red hair.
```

通过标记出其中的六个空格，我们能识别出七个子字符串，它们表示了文本行中实际的单词。为了做到这一点，我们使用 string 类支持的多个 find()函数之一。

string 类提供了一套查找函数，都以 find 的各种变化形式命名。find()是最简单的实例：给出一个字符串，它返回匹配子串的第一个字符的索引位置。或者返回一个特定的值：

```
string::npos
```

表明没有匹配。例如：

```
#include <string>
#include <iostream>

int main() {
    string name( "AnnaBelle" );
    int pos = name.find( "Anna" );

    if ( pos == string::npos )
        cout << "Anna not found!\n";
    else cout << "Anna found at pos: " << pos << endl;
}
```

虽然返回的索引类型差不多总是 int 类型，但是，更严格的、可移植的正确声明应该使用以下形式：

```
string::size_type
```

来保存从 find()返回的索引值。例如：

```
string::size_type pos = name.find( "Anna" );
```

find()并没有提供我们所需要的确切功能。然而，find_first_of()提供了这样的功能。find_first_of()查找与被搜索字符串中任意一个字符相匹配的第一次出现，并返回它的索引位置。例如，下列代码找到字符串中的第一个数字：

```
#include <string>
#include <iostream>

int main() {
    string numerics( "0123456789" );
    string name( "r2d2" );
    string::size_type pos = name.find_first_of( numerics );

    cout << "found numeric at index: "
         << pos << "\telement is "
```

```
            << name[pos] << endl;
    }
```

在这个例子中，pos 被设置为 1（记住，字符串的元素从 0 开始索引）。

但是，这仍然不是我们所需要的。我们需要顺序找到所有的出现，而不是第一个出现。我们可以通过给出第二个参数来实现，这个参数指明了字符串中起始查找位置的索引。下面重写了对“rad2”的查找，但它还不是完全正确的。你能看出有什么问题吗？

```
#include <string>
#include <iostream>

int main() {
    string numerics( "0123456789" );
    string name( "r2d2" );
    string::size_type pos = 0;

    // 这里存在错误!
    while (( pos = name.find_first_of( numerics, pos ))
                != string::npos )
        cout << "found numeric at index: "
             << pos << "\telement is "
             << name[pos] << endl;
}
```

循环开始时，pos 被初始化为 0，即从 0 开始查找字符串。在索引位置 1 上出现一次匹配。pos 被赋值为该值。因为它不等于 npos，所以继续执行循环体。第二个 find_first_of() 执行时，pos 是 1。喔！索引位置 1 被匹配第二次、第三次。第四次……，我们已经陷入了无限循环。需要在循环的下一次迭代开始之前，将 pos 递增 1，使其指向被找到元素的后一位置。

```
// ok: 被改正之后的循环迭代
while (( pos = name.find_first_of( numerics, pos ))
             != string::npos )
{
    cout << "found numeric at index: "
         << pos << "\telement is "
         << name[pos] << endl;

    // 移到被找到元素的后一位置
    ++pos;
}
```

为了在文本行中找到内嵌的空格，我们只需把 numerics 换成一个含有可能遇到的空格字符的字符串。但是，如果我们确定只有一个空格字符被用到，那么就可以显式地提供这个字符。例如：

```
// 程序片断
while (( pos = textline.find_first_of( ' ', pos ))
                                != string::npos )
    // ...
```

为了标记出每个单词的长度，我们引入了第二个位置对象，如下所示：

```
// 程序片断
// pos: 单词后一位置的索引
// prev_pos: 单词开始的索引
string::size_type pos = 0, prev_pos = 0;

while (( pos = textline.find_first_of( ' ', pos ))
                                    != string::npos )
{
     // 对 string 进行一些操作
     // 调整位置标记器
     prev_pos = ++pos;
}
```

对于循环的每次迭代，prev_pos 索引单词的开始，pos 持有单词末尾的下一个位置（空格的位置）。然后，每个被标识的单词的长度为：

```
pos - prev_pos;    // 标识单词长度
```

现在我们已经标识出单词，接着就需要拷贝它了，把它放在一个字符串 vector 中。拷贝单词的一种策略是从 prev_pos 到 pos 减 1 循环遍历 textline，顺次拷贝每个字符，抽取出由这两个索引标记的子字符串。但是，我们不需要自己去做，由 substr()字符串操作来完成就可以了：

```
// 程序片断
vector<string> words;
while (( pos = textline.find_first_of( ' ', pos ))
                    != string::npos )
{
     words.push_back( textline.substr(
                    prev_pos, pos-prev_pos));
     prev_pos = ++pos;
}
```

substr()操作生成现有 string 对象的子串的一个拷贝。它的第一个参数指明开始的位置。第二个可选的参数指明子串的长度（如果省略第二个参数，将拷贝字符串的余下部分）。

我们的实现有个错误：在插入每一行的最后一个单词时，它就会失败。你知道为什么吗？考虑下面的文本行：

```
seaspawn and seawrack
```

前两个单词由空格标记。这两个空格的位置都由 find_first_of()返回。但是，第三次调用并没有找到空格符：它将 pos 置为 string::npos，结束了循环。那么，最后一个单词的处理必须跟在循环结束之后。

下面是完整的实现，我们已将其放入一个被称为 separate_words()的函数中。除了把每个单词存储在字符串 vector 中之外，我们还计算了每个单词的行列位置（在以后对基于位置的文本查询的支持中我们将需要这些信息）。

```
typedef pair<short,short> location;
```

```
typedef vector<location> loc;
typedef vector<string> text;
typedef pair<text*,loc*> text_loc;
text_loc*
separate_words( const vector<string> *text_file )
{
    // words: 包含独立单词的集合
    // locations: 包含相关的行/列信息
    vector<string> *words = new vector<string>;
    vector<location> *locations = new vector<location>;
    short line_pos = 0; // current line number

    // iterate through each line of text
    for ( ; line_pos < text_file->size(); ++line_pos )
    {
        // textline: 待处理的当前文本行
        // word_pos: 文本行中的当前列位置
        short word_pos = 0;
        string textline = (*text_file)[ line_pos ];
        string::size_type pos = 0, prev_pos = 0;

        while (( pos = textline.find_first_of( ' ', pos ))
                    != string::npos )
        {
            // 存储当前单词子串的拷贝
            words->push_back(
                textline.substr( prev_pos, pos - prev_pos ));
            // 将行/列信息存储为 pair
            locations ->push_back(
                make_pair( line_pos, word_pos ));
            // 为下一次迭代修改位置信息
            ++word_pos; prev_pos = ++pos;
        }

        // 现在处理最后一个单词
        words->push_back(
            textline.substr( prev_pos, pos - prev_pos ));
        locations->push_back(
            make_pair( line_pos, word_pos ));
    }
    return new text_loc( words, locations );
}
```

到现在为止，我们的程序的控制流如下：

```
int main()
{
    vector<string> *text_fo;e = retroeve_text);
```

```
        text_loc *text_locations = separate_words( text_file );
        // ...
    }
```

separate_words()在 text_file 上的部分执行情况如下：

```
eol: 52 pos: 5 line: 0 word: 0 substring: Alice
eol: 52 pos: 10 line: 0 word: 1 substring: Emma
eol: 52 pos: 14 line: 0 word: 2 substring: has
eol: 52 pos: 19 line: 0 word: 3 substring: long
eol: 52 pos: 27 line: 0 word: 4 substring: flowing
eol: 52 pos: 31 line: 0 word: 5 substring: red
eol: 52 pos: 37 line: 0 word: 6 substring: hair.
eol: 52 pos: 41 line: 0 word: 7 substring: Her
eol: 52 pos: 47 line: 0 word: 8 substring: Daddy
last word on line substring: says

...

textline: magical but untamed. "Daddy, shush, there is no such thing,"
eol: 60 pos: 7 line: 3 word: 0 substring: magical
eol: 60 pos: 11 line: 3 word: 1 substring: but
eol: 60 pos: 20 line: 3 word: 2 substring: untamed.
eol: 60 pos: 28 line: 3 word: 3 substring: "Daddy,
eol: 60 pos: 35 line: 3 word: 4 substring: shush,
eol: 60 pos: 41 line: 3 word: 5 substring: there
eol: 60 pos: 44 line: 3 word: 6 substring: is
eol: 60 pos: 47 line: 3 word: 7 substring: no
eol: 60 pos: 52 line: 3 word: 8 substring: such
last word on line substring: thing,"

...

textline: Shyly, she asks, "I mean, Daddy, is there?"
eol: 43 pos: 6 line: 5 word: 0 substring: Shyly,
eol: 43 pos: 10 line: 5 word: 1 substring: she
eol: 43 pos: 16 line: 5 word: 2 substring: asks,
eol: 43 pos: 19 line: 5 word: 3 substring: "I
eol: 43 pos: 25 line: 5 word: 4 substring: mean,
eol: 43 pos: 32 line: 5 word: 5 substring: Daddy,
eol: 43 pos: 35 line: 5 word: 6 substring: is
last word on line substring: there?"
```

在加入我们的文本查询函数集合之前，先简要地概括一下 string 类支持的其他查找函数。除了 find()和 find_first_of()外，string 类还支持其他几个查找操作：rfind()，查找最后（即最右）的指定子串的出现。例如：

```
string river( "Mississippi" );
string::size_type first_pos = river.find( "is" );
string::size_type last_pos = river.rfind( "is" );
```

find()返回索引值 1，表明第一个“is”的开始，而 rfind()返回索引值 4，表明“is”的最后一个出现的开始。

find_first_not_of()查找第一个不与要搜索字符串的任意字符相匹配的字符。例如，为找到第一个非数字字符，可以写：

```
string elems( "0123456789" );
string dept_code( "03714p3" );

// returns index to the character 'p'
string::size_type pos = dept_code.find_first_not_of(elems);
```

find_last_of()查找字符串中的“与搜索字符串任意元素相匹配” 的最后一个字符。find_last_not_of()查找字符串中的“与搜索字符串任意字符全不匹配”的最后一个字符。这些操作都有一个可选的第二参数来指明起始的查找位置。

练习 6.13

写一个程序，已知下列字符串：

```
"ab2c3d7R4E6"
```

先用 find_first_of()，然后再用 find_first_not_of()查找每个数字字符，最后查找每个英文字母。

练习 6.14

写一个程序，已知字符串：

```
string line1 = "We were her pride of 10 she named us --";
string line2 = "Benjamin, Phoenix, the Prodigal"
string line3 = "and perspicacious pacific Suzanne";

string sentence = line1 + ' ' + line2 + ' ' + line3;
```

统计句子中单词的个数并找出最大的和最小的单词。如果不只有一个最大或最小单词，则把它们全部列出来（指定位置）

6.9 处理标点符号

我们已经把每个文本行分解成独立的单词，现在需要把单词中可能附加的标点符号去掉。例如，下列文本行：

```
magical but untamed. "Daddy, shush, there is no such thing,"
```

被分解成：

```
magical
but
untamed.
"Daddy,
shush,
there
is
no
```

```
such
thing,"
```

怎样去掉不想要的标点呢？首先，我们将定义一个字符串，它包含我们希望去掉的所有标点：

```
string filt_elems( "\",.;:!?)(\\/" );
```

（\"和\\序列表示：第一个序列中的引号和第二个序列中的第二个反斜杠被视为该字符串中的文字元素，而不是字符串的结尾或下一行的续行符号。）

接下来，我们将用 find_first_of()操作在我们的字符串里找到每个匹配元素：

```
while (( pos = word.find_first_of( filt_elems, pos ))
                                    != string::npos )
```

最后，我们需要用 erase()去掉字符串中的标点：

```
word.erase(pos, 1);
```

这个版本的 erase()操作的第一个参数表示字符串中要被删除的字符的开始位置。第二个参数是可选的，表示要被删除的字符的个数。在我们的例子中，我们正在删除位置 pos 上的字符。如果我们省略第二个参数，则 erase()将删除从 pos 到字符串结束的所有字符。

下面是 filter_text()的完整实现。它有两个参数，指向包含文本的 string vector 的指针，以及含有要过滤的元素的 string 对象：

```
void
filter_text( vector<string> *words, string filter )
{
    vector<string>::iterator iter = words ->begin();
    vector<string>::iterator iter_end = words ->end();

    // 如果用户没有提供 filter，则缺省使用最小集
    if ( ! filter.size() )
        filter.insert( 0, "\"." );

    while ( iter != iter_end ) {
        string::size_type pos = 0;

        // 对于找到的每个元素，将其删除
        while (( pos = (*iter).find_first_of( filter, pos ))
                        != string::npos )
            (*iter).erase(pos,1);
        iter++;
    }
}
```

你知道为什么不随循环的每一次迭代递增 pos 吗？也就是下列代码为什么是不正确的码？

```
while (( pos = (*iter).find_first_of( filter, pos ))
                    != string::npos )
{
    (*iter).erase(pos,1);
    ++pos;    // 下正确 ...
}
```

pos 表示 string 中的位置。例如，已知字符串：

```
thing,"
```

循环的第一次迭代向 pos 赋值 5，即逗号的位置。在去掉逗号之后，字符串变成：

```
thing"
```

位置 5 现在是双引号。如果我们已经递增 pos，那么将不能找到并去掉这个标点符号。

下面给出怎样在主程序中调用 filter_text()：

```
string filt_elems( "\",.;:!?)(\\/" );
filter_text( text_locations ->first, filt_elems );
```

最后，下面是我们的文本中的一些字符串的跟踪实例，在该文本中找到了一个或多个过滤元素：

```
filter_text: untamed.
found! : pos: 7.
after: untamed

filter_text: "Daddy,
found! : pos: 0"
after: Daddy,
found! : pos: 5,
after: Daddy

filter_text: thing,"
found! : pos: 5,
after: thing"
found! : pos: 5"
after: thing

filter_text: "I
found! : pos: 0"
after: I

filter_text: Daddy,
found! : pos: 5,
after: Daddy

filter_text: there?"
found! : pos: 5?
after: there"
found! : pos: 5"
after: there
```

练习 6.15

写一个程序，已知下列字符串：

```
"/.+(STL).*$1/"
```

先用 erase(pos.count)，然后再用 erase(iter,iter)去掉除了“STL”外的所有字符。

练习 6.16

写一个程序，它能够接受下列定义：

```
string sentence( "kind of" );
string s1( "whistle" );
string s2( "pixie" );
```

各种 string 插入函数，得出值为：

```
"A whistling-dixie kind of walk."
```

的句子。

6.10 任意其他格式的字符串

文本查询系统的一件麻烦事情就是需要识别不同时态的同一个词，如 cry、cries 和 cried；不同数目的同一个词，如 baby、babies；以及大小写不同的同一个词，如 home 和 Home。前两种情况属于单词后缀问题。虽然后缀问题超出了本书的范围，但是，下面的小示例方案给出了 string 类操作的一个很好的练习。

在进入后缀问题之前，我们先来解决大小写字母的简单情形。我们不想处理各种特殊的情况，而只是简单地用小写形式来代替所有的大写字母。我们的实现看起来是这样的：

```
void
strip_caps( vector<string,allocator> *words )
{
    vector<string,allocator>::iterator iter = words ->begin();
    vector<string,allocator>::iterator iter_end = words ->end();

    string caps( "ABCDEFGHIJKLMNOPQRSTUVWXYZ" );

    while ( iter != iter_end ) {
        string::size_type pos = 0;
        while (( pos = (*iter).find_first_of( caps, pos ))
                    != string::npos )
            (*iter)[ pos ] = tolower( (*iter)[pos] );
        ++iter;
    }
}
```

以下函数：

```
tolower( (*iter)[pos] );
```

是标准 C 库函数，它接受一个大写字符，并返回与之等价的小写字母。为了使用它，我们必须包含头文件：

```
tolower( (*iter)[pos] );
```

［这个文件也包含其他函数的声明，如 isalpha()、isdigit()、ispunct()、isspace()、toupper()，以及其他一些函数。完整的列表和讨论请参见［PLAUGER92］。标准 C++库定义了一个 ctype 类，它封装了标准 C 库函数的功能以及一组非成员函数，如 toupper()、tolower()等等。为了

使用它们，我们必须包含标准 C++头文件：

```
#include <locale>
```

然而，当本书正在写作时，我们还无法得到支持这种实现的 C++编译器，所以我们使用标准 C 的实现方式。]

后缀问题很难完全解决，但是，完美的实现会显著改善我们查询单词集合的质量和大小。

我们的实现只处理以“s”结尾的单词：

```
void
suffix_text( vector<string,allocator> *words )
{
    vector<string,allocator>::iterator
                iter = words->begin(),
                iter_end = words->end();

    while ( iter != iter_end ) {
        // 如果只有 3 个字符或者更少，则不加处理
        if ( (*iter).size() <= 3 )
            { ++iter; continue; }

        if ( (*iter)[ (*iter).size ()-1 ] == 's' )
            suffix_s( *iter );

        // 其他后缀的处理，比如 ed、ing、ly 等

        ++iter;
    }
}
```

一种简单的做法是不要理会少于四个字符的单词。这使我们免于处理 has、its、is 等等，但是，却不能匹配像 tv 和 tvs 这样的词。

如果单词以“ies”结尾，如 babies 和 cries，则我们需要用“y”代替“ies”：

```
string::size_type pos3 = word.size()-3;
string ies( "ies" );
if ( ! word.compare( pos3, 3, ies )) {
    word.replace( pos3, 3, 1, 'y' );

    return;
}
```

如果两个字符串的比较结果相等，则 compare()返回 0。pos3 表示 word 中开始比较的位置。第二个参数，本例中为 3，表示从 pos3 开始的子字符串的长度。第三个参数是实际与之比较的字符串。[compare()实际上有六个版本。我们将在下节简要介绍其他版本。]

replace()用一个或多个替换字符来替换字符串中的一个或多个字符。在本例中，我们用单个字符“y”代替三个字符的子串“ies”[replace()有十个重载的实例。我们将在下节简要回顾它们。]

类似地，如果单词以“ses”结尾，比如 promises 和 purposes 中的“ses”，则我们只要

去掉尾部“es”即可[18]：

```
string ses( "ses" );
if ( ! word.compare( pos3, 3, ses )) {
     word.erase( pos3+1, 2 );
     return;
}
```

如果单词以“ous”结尾，如 oblivious、fulvous 和 cretaceous，则我们什么都不做。类似地，如果单词以“is”结尾，如 genesis、mimesis 和 hepatitis，我们也是什么都不做。（然而，这个系统并不是很完美的。例如，对于 Kimis，则需要我们去掉最后的“s”。）此外，如果单词以“ius”结尾，如 genius，或者以“ss”结尾，如 hiss、lateness 或 less，则我们什么都不做。为了决定是否什么都不做，我们使用第一种形式的 compare()：

```
string::size_type spos = 0;
string::size_type pos3 = word.size()-3;

// "ous", "ss", "is", "ius"
string suffixes( "oussisius" );

if ( ! word.compare( pos3, 3, suffixes, spos, 3 ) ||          // ous
          ! word.compare( pos3, 3, suffixes, spos+6, 3 ) ||   // ius
          ! word.compare( pos3+1, 2, suffixes, spos+2, 2 ) || // ss
          ! word.compare( pos3+1, 2, suffixes, spos+4, 2 ))   // is
     return;
```

否则，我们只是简单地去掉“s”：

```
// erase ending 's'
word.erase( pos3+2 );
```

有一些名字，比如 Pythagoras、Brahms 和前拉斐尔派画家 Burne Jones，都在此通用规则之外。当引入 set 关联容器类型时，我们将处理它们——实际上是把它留给读者作练习。

在转向 map 和 set 关联容器类型之前，我们将在下节中简要介绍 string 的其他一些操作。

练习 6.17

我们的程序不处理以 ed 结尾的后缀，如 surprised。ly 结尾，如 surprisingly。以及 ing 结尾。如 surprising。把下列后缀处理程序之一加到程序中：(a) suffix_ed(); (b) suffix_ly; (c) suffix_ing()。

6.11 其他 string 操作

erase()的第二种形式用一对迭代器（iterator）作参数，标记出要被删除的字符的范围。例如，已知 string：

```
string name( "AnnaLiviaPlurabelle" );
```

[18] 当然，也会有例外。例如，crises，按我们的做法．就变成了 cris。喔！

我们来生成字符串“Annabelle”：

```
typedef string::size_type size_type;
size_type startPos = name.find( 'L' );
size_type endPos = name.find_last_of( 'a' );

name.erase( name.begin()+startPos,
            name.begin()+endPos );
```

由第二个 iterator 指向的字符不属于要被删除的字符范围。这意味着我们将产生

```
Annaabelle，而不是 Annabelle。
```

最后，第三种形式只带一个 iterator 作参数，它标记出要被删除的字符。我们给它传递 endPos 来删除第二个重复的“a”：

```
name.erase( endPos );
```

这使 name 的值为 Annabelle。

insert()操作支持将另外的字符插入到指定的位置。它的一般格式是：

```
string_object.insert( position, new_string );
```

这里，position 表示在 string_object 中插入 new_string 的位置。new_string 可以是 string、C 风格字符串或者单个字符。例如：

```
string string_object( "Missisippi" );
string::size_type pos = string_object.find( "isi" );
string_object.insert( pos+1, "s" );
```

insert()操作也支持插入 new_string 的一个子部分。例如：

```
string new_string ( "AnnaBelle Lee" );
string_object += ' '; // 追加一个空格

// 找到 new_string 中开始和结束处的位置
pos = new_string.find( 'B' );
string::size_type posEnd = new_string.find( ' ' );
string_object.insert(
     string_object.size(),  // string_object 中的位置
     new_string, pos,       // new_string 的开始位置
     posEnd-pos             // 要拷贝字符的数目
)
```

现在，string_object 含有字符串“Mississippi Belle”。如果我们希望插入从 pos 开始的整个 new_string，则可以省略 posEnd 值。

已知下列两个字符串：

```
string s1( "Mississippi" );
string s2( "Annabelle" );
```

我们希望利用它们来创建第三个字符串，其值为“MissAnna”。应该怎样做呢？

一种方法是用 assign()和 append()字符串操作。它们允许我们顺次地把一个 string 对象的部分拷贝或连接到另一个 string 对象上。例如：

```
string s3;
```

```
// 拷贝 s1 的前 4 个字符
s3.assign( s1, 0, 4 );
```

s3 现在的值为“Miss”。

```
// 连接一个空格
s3 += ' ';
```

s3 现在的值为“Miss ”。

```
// 连接 s2 的前 4 个字符
s3.append( s2, 0, 4 );
```

s3 现在的值为“Miss Anna”。另外，我们也可以把它写成：

```
s3.assign( s1, 0, 4 ).append( ' ' ).append( s2, 0, 4 );
```

如果我们希望抽出字符串的一部分，但不是从头开始，那么可以使用另外一种形式，它用两个整型值作参数：开始位置和长度。位置从 0 开始计数。例如，为了从“Annabelle”中抽取“belle”，我们指定开始位置 4 和长度 5：

```
string beauty;

// 给 beauty 赋值"belle"
beauty.assign( s2, 4, 5 );
```

使用另外一种形式则不用提供位置和长度，而是提供一个 iterator 对。例如：

```
// 给 beauty 赋值"belle"
beauty.assign( s2, s2.begin()+4, s2.end() );
```

在下面的例子中，两个字符串分别表示当前的任务和待处理的任务。我们需要定期随项目的更换交换两者。例如；

```
string current_project( "C++ Primer, 3rd Edition" );
string pending_project( "Fantasia 2000, Firebird segment" );
```

swap()操作交换两个 string 的值。每次调用：

```
current_project.swap( pending_project );
```

都会交换两个 string 对象的值。

已知字符串：

```
string first_novel( "V" );
```

则下标：

```
char ch = first_novel[ 1 ];
```

返回一个未定义的字符串，因为索引值超出了范围：first_novel 长度为 1，由值 0 索引。下标操作符不提供范围检查。对于高质量的代码我们也不希望它这样做，如：

```
int
elem_count( const string &word, char elem )
{
    int occurs = 0;
```

```
        // 很好：不需要检查边界
        for ( int ix=0; ix < word.size(); ++ix )
            if ( word[ ix ] == elem )
                ++occurs;

        return occurs;
    }
```

但是，对可能含有错误定义的代码，如：

```
    void
    mumble( const string &st, int index )
    {
        // 潜在的范围错误
        char ch = st[ index ];

        // ...
    }
```

另一个可替代的 at()操作提供了运行时刻对索引值的范围检查。如果索引是有效的，则 at()返回相关的字符元素：与下标操作符的方式相同。但是，如果索引无效，则 at()抛出 out_of_range 异常：

```
    void
    mumble( const string &st, int index )
    {
        try {
            char ch = st.at(index);
            // ...
        }
        catch( std::out_of_range ) { ... }

        // ...
    }
```

任意两个不相等的字符串都有一个字典顺序。例如，已知下列两个字符串：

```
    string cobol_program_crash( "abend" );
    string cplus_program_crash( "abort" );
```

cobol_program_crash 字符串对象小于 cplus_program_crash 字符串对象，这是通过比较第
一个不相等的字符得到的。在英文字母表中，e 出现在 o 前面。

compare()字符串操作提供了两个字符串的字典序比较。给定：

```
    s1.compare( s2 );
```

则 compare()返回三个可能值之一：

1. 如果 s1 大于 s2，则 compare()返回一个正值；
2. 如果 s1 小于 s2，则 compare()返回一个负值；
3. 如果 s1 等于 s2，则 compare()返回 0。

例如：

```
    cobol_program_crash.compare( cplus_program_crash );
```

返回一个负值，而：

```
cplus_program_crash.compare( cobol_program_crash );
```

返回一个正值。string 关系操作符（<、>、!=、==、<=和>=）给出了 compare()操作的一种替代简短表示。

compare()操作有六个重载版本，利用这些比较函数，我们可以标记出其中一个或者两个字符串的子串以进行比较。在上一节关于后缀的讨论中我们已经看到过一些例子。

replace()提供了十种方式，使我们可以用一个或多个字符替换字符串中的一个或多个现有的字符（现有字符与替换字符的数目可以不等）。replace()操作有两种基本格式，各种变化形式主要在于如何标记出要被替换的字符集合。在第一种格式中，前两个参数给出了指向字符集开始的索引以及要被替换的字符的个数。在第二种格式中，传递了一对 iterator，分别标记出字符集的开始位置以及要被替换的最后一个字符的下一位置。下面是第一种格式的例子：

```
string sentence(
"An ADT provides both interface and implementation." );
string::size_type position = sentence.find_last_of( 'A' );
string::size_type length = 3;

// 用 Abstract Data Type 代替 ADT
sentence.replace( position, length, "Abstract Data Type" );
```

第一个参数代表开始位置，第二个参数代表从 position 开始的字符串的长度；因此，长度是 3，而不是 2，表示字符串“ADT”。第三个参数代表新的字符串。有许多变化形式可以用来指定这个新字符串。例如，下面这个变种版本用一个 string 对象而不是 C 风格字符串作参数：

```
string new_str( "Abstract Data Type" );
sentence.replace( position, length, new_str );
```

下面的变种版本插入由位置和长度标记的新字符串的子串：

```
#include <string>
typedef string::size_type size_type;

// 得到 3 个单词的位置
size_type posA = new_str.find( 'A' );
size_type posD = new_str.find( 'D' );
size_type posT = new_str.find( 'T' );

// ok: 用"Type"代替 T
sentence.replace( position+2, 1, new_str, posT, 4 );

// ok: 用"Date" 代替 D
sentence.replace( position+1, 1, new_str, posD, 5 );

// ok: 用"Abstract"代替 A
sentence.replace( position, 1, new_str, posA, 9 );
```

另外一种版本是专门为“用一个指定重复次数的单个字符替换一个子串”而提供的。例如：

```
string hmm( "Some celebrate Java as the successor to C++." );
string::size_type position = hmm.find( 'J' );
// ok: 用 xxxx 代替 Java
hmm.replace( position, 4, 4, 'x');
```

我们要说明的最后一种版本是用一个字符数组的指针和长度来标记新串。例如：

```
const char *lang = "EiffelAda95JavaModula3";
int index[] = { 0, 6, 11, 15, 22 };
string ahhem(
     "C++ is the language for today's power programmers." );
ahhem.replace(0, 3, lang+index[1], index[2]-index[1]);
```

下面是第二种格式的例子，它用一对 iterator 来标记要被替换的目标子串：

```
string sentence(
"An ADT provides both interface and implementation." );

// 指向 ADT 的'A'
string::iterator start = sentence.begin()+3;

// 用 Abstract Data Type 代替 ADT
sentence.replace( start, start+3, "Abstract Data Type" );
```

另外四种变种形式允许替换串是 string 对象、一个字符重复 N 次、一对 iterator、或 C 风格字符串中的 N 个字符被用作替换字符集。

关于 string 操作我们就介绍到这里。更详细完整的信息，请参见 C++标准定义［ISO_C++97］。（在写这本书的时候，还没有比较好一点的关于标准 C++库的参考资料。）

练习 6.18

写一个程序，使它能接受下列两个字符串：

```
string quote1( "When lilacs last in the dooryard bloom'd" );
string quote2( "The child is father of the man" );
```

用 assign()和 append()操作构造字符串：

```
string sentence( "The child is in the dooryard" );
```

练习 6.19

写一个程序，已知字符串：

```
string generic1( "Dear Ms Daisy:" );
string generic2( "MrsMsMissPeople" );
```

实现下面的函数：

```
string generate_salutation( string generic1,
                            string lastname,
                            string generic2,
                            string::size_type pos,
```

```
                        int length );
```

请使用 replace()操作，这里用 lastname 代替 Daisy，用 pos 索引 generic2 并且长度为 length 的字符替换 Ms。例如：

```
string lastName( "AnnaP" );
string greetings =
    generate_salutation( generic1, lastName, generic2, 5, 4 );
```

返回字符串：

```
Dear Miss AnnaP:
```

6.12 生成文本位置 map

本节中我们将为文本中每个单词建立一个行列位置集合，以此引入并探讨关联容器类型 map。（在下节中，我们将建立一个单词排除集，以此引入并探讨关联容器 set。）一般地，当我们只想知道一个值是否存在时，set 最有用处。希望存储（也可能修改）一个相关的值时，map 最为有用。在这两种情况下，元素都是以有序关系存储的，以此支持高效率的存储和检索。

在 map（也叫关联数组，associative array）中，我们提供一个“键/值”对：键用来索引 map，而值用作被存储和检索的数据。在我们的程序示例中，每个 string 对象用作键，行列位置的 vector 用作值。为访问位置 vector，我们用下标操作符索引 map。例如：

```
string query( "pickle" );
vector< location > *locat;

// 返回与"pickle"相关的 vector<location>*
locat = text_map[ query ];
```

map 的键类型——本例中为 string——用作索引。相关的 location<vector>*值被返回。为了使用 map，我们必须包含相关的头文件：

```
#include <map>
```

在使用 map（和 set）时，两个最主要的动作是向里面放入元素，以及查询元素是否存在。在下一小节中，我们将看到怎样定义和插入键/值对。在其后的小节中，我们将了解怎样发现一个元素是否存在，若存在，又怎样获取它的值。

6.12.1 定义并生成 map

为定义 map 对象，我们至少要指明键和值的类型。例如：

```
map<string, int> word_count;
```

定义了 map 对象 word_count，它由 string 作为索引，并拥有一个相关的 int 值。类似地：

```
class employee;
map<int, employee*> personnel;
```

定义了 map 对象 personnel，它由一个 int 作为索引（代表一个惟一的雇员号）并拥有相

关联的指向雇员类实例的指针。

对于我们的文本查询系统，map 声明如下：

```
typedef pair<short,short> location;
typedef vector<location> loc;
map<string,loc*> text_map;
```

因为在写作本书时，我们能使用的编译器都不支持模板参数的缺省参数，所以，在实际中，我们必须提供下列扩展的定义：

```
map<string, loc*,       // 键、值对
less<string>,           // 用作排序的关系操作符
allocator>              // 缺省的内存分配操作符
text_map;
```

缺省情况下，关联容器类型用小于操作符排序。然而，我们总是可以改变它，只需提供一个其他可替换的关系操作符（见 12.3 节函数对象）

定义了 map 以后，下一步工作就是加入键/值元素对。直观上，我们希望这样写代码

```
#include <map>
#include <string>

map<string,int> word_count;
word_count[ string("Anna") ] = 1;
word_count[ string("Danny") ] = 1;
word_count[ string("Beth") ] = 1;

// 等等
```

当我们写如下语句时：

```
word_count[ string("Anna") ] = 1;
```

将发生以下事情：

1．一个未命名的临时 string 对象被构造并传递给与 map 类相关联的下标操作符，这个对象用“Anna”初始化。

2、在 word_count 中查找“Anna”项，没有找到。

3．一个新的键/值对被插入到 word_count 中。当然，键是一个 string 对象，持有“Anna”。但是，值不是 1，而是 0。

4．插入完成，接着值被赋为 1。

通过下标操作符把一个键插入到 map 中时，而相关联的值被初始化为底层元素类型的缺省值。内置数值类型的缺省值为 0。

实际上，用下标操作符把 map 初始化至一组元素集合，会使每个值都被初始化为缺省值，然后再被赋值为显式的值。如果元素是类对象。而且它的缺省初始化和赋值的运算量都很大，就会影响程序的性能，尽管不会影响程序的正确性。

一种比较好的插入单个元素的方法如下所示：

```
// the preferred single element insertion method
word_count.insert(
     map<string,int>::
```

```
            value_type( string("Anna"), 1 )
    );
```

map 定义了一个类型 value_type，表示相关联的键值对。下面一行代码：

```
map< string,int >::
    value_type( string("Anna"), 1 )
```

其作用是创建一个 pair 对象，接着将其直接插入 map。为了便于阅读，我们使用 typedef：

```
typedef map<string,int>::value_type valType;
```

使用它，插入操作看起来就不那么复杂了：

```
word_count.insert( valType( string("Anna"), 1 ));
```

为插入一定范围内的键/值元素，我们可以用另一个版本的 insert()方法，它用一对 iterator 作为参数。例如：

```
map< string, int > word_count;

// ... fill it up
map< string, int > word_count_two;

// 插入所有键/值对
word_count_two.insert(word_count.begin(),word_count.end());
```

在本例中，我们也可以通过把第二个 map 对象初始化成第一个，以获得同样的效果：

```
// 用所有键/值对的拷贝初始化
map< string, int > word_count_two( word_count );
```

现在我们来浏览一下怎样建立起我们的文本 map。6.8 节讨论的 separate_words()创建了两个 vector：文本中所有词的字符串 vector，以及相应的行列对的位置 vector。对于字符串 vector 中的每个单词元素，位置 vector 中的等价元素都分别给出了该词的行列信息。字符串 vector 为文本 map 提供了键的集合，而位置 vector 提供相关联的值集合。

separate_words()返回一个 pair 对象，它拥有指向这两个 vector 的指针。这个 pair 对象是我们的函数 build_word_map()的参数。返回值是文本位置 map——或指向它的指针：

```
// typedefs to make declarations easier
typedef pair< short,short > location;
typedef vector< location > loc;
typedef vector< string > text;
typedef pair< text*,loc* > text_loc;

extern map< string, loc* >*
    build_word_map( const text_loc *text_locations );
```

第一个准备工作是从空闲存储区中分配一个空 map，以及从作为参数的 pair 对象中分离出字符串和位置 vector：

```
map<string,loc*> *word_map = new map< string, loc* >;

vector<string> *text_words = text_locations ->first;
vector<location> *text_locs = text_locations ->second;
```

接下来，我们需要并行迭代两个 vector。有两种情况需要考虑：

1. map 中还没有单词。在这种情况下，需要插入键/值对。
2. 单词已经被插入。在这种情况下，需要用另外的行列信息修改该项的位置 vector。

下面是我们的实现代码：

```
register int elem_cnt = text_words ->size();
for ( int ix = 0; ix < elem_cnt; ++ix )
{
     string textword = ( *text_words )[ ix ];

     // 排除策略：如果少于 3 个字符，
     // 或在排除集合中存在，
     // 则不输入到 map 中.
     if ( textword.size() < 3 ||
                  exclusion_set.count( textword ))
          continue;

     // 判断单词是否存在
     // 如果 count()返回 0，则不存在—加入它
     if ( ! word_map->count((*text_words)[ix] ))
     {
          loc *ploc = new vector<location>;
          ploc->push_back( (*text_locs)[ix] );
          word_map->insert( value_type( (*text_words)[ix], ploc ));
     }
     else
          // 修改该项的位置向量
          (*word_map)[(*text_words)[ix]]->push_back((*text_locs)[ix]);
}
```

对于这个语法复杂的表达式：

```
(*word_map)[(*text_words)[ix]]->push_back((*text_locs)[ix]);
```

如果我们把它分解成独立的部分可能更容易理解一些：

```
// 得到要修改的单词
string word = (*text_word)[ix];

// 得到位置向量
vector<location> *ploc = (*word_map)[ word ];

// 得到行列对
location loc = (*text_locs)[ix];

// 插入新的行列对
ploc->push_back(loc);
```

其余的语法复杂性是因为操纵指针而导致的，并不是因为 vector 本身。为直接应用下标操作符我们不能写：

```
string word = text_words[ix]; // 错误
```

相反，我们必须先解除指针的引用。

```
string word = (*text_words)[ix]; // ok
```

最后，build_word_map 返回内部建好的 map：

```
return word_map;
```

下面给出怎样在 main()函数中调用它：

```
int main()
{
    // 读入并分离文本
    vector<string,allocator> *text_file = retrieve_text();
    text_loc *text_locations = separate_words( text_file);

    // 处理单词
    // ...

    // 生成单词/位置对并提示查询
    map<string,loc*,less<string>,allocator>
        *text_map = build_word_map( text_locations );

    // ...
}
```

6.12.2 查找并获取 map 中的元素

下标操作符给出了获取一个值的最简单方法。例如：

```
// map<string,int> word_count;
int count = word_count[ "wrinkles" ];
```

但是，只有当 map 中存在这样一个键的实例时，该代码才会表现正常。如果不存在这样的实例，使用下标操作符会引起插入一个实例。在本例中，键/值对：

```
string( "wrinkles" ), 0
```

被插入到 word_count 中，count 被初始化为 0。

有两个 map 操作能够发现一个键元素是否存在，而且在键元素不存在时也不会引起插入实例：

1．Count(keyValue)：count()返回 map 中 keyValue 出现的次数。（当然，对于 map 而言，返回值只能是 0 或 1。）如果返回值非 0，我们就可以安全地使用下标操作符。例如：

```
int count = 0;
if ( word_count.count( "wrinkles" ))
    count = word_count[ "wrinkles" ];
```

2．Find(keyValue)：如果实例存在，则 find()返回指向该实例的 iterator。如果不存在，则返回等于 end()的 iterator。例如：

```
int count = 0;
map<string,int>::iterator it = word_count.find( "wrinkles" );

if ( it != word_count.end() )
    count = (*it).second;
```

指向 map 中元素的 iterator 指向一个 pair 对象，其中 first 拥有键，second 拥有值（我们将在下一小节再次看到这一点）。

6.12.3 对 map 进行迭代

现在我们已经建立了自己的 map，接下去想输出它的内容。我们可以通过对“由 begin()和 end()两个迭代器标记的所有元素”进行迭代，来做到这一点。下面是完成了此任务的函数

```
display_map_text():
display_map_text( map<string,loc*> *text_map )
{
     typedef map<string,loc*> tmap;
     tmap::iterator iter = text_map->begin(),
          iter_end = text_map->end();

     while ( iter != iter_end )
     {
          cout << "word: " << (*iter).first << " (";
          int loc_cnt = 0;
          loc *text_locs = (*iter).second;
          loc::iterator liter = text_locs->begin(),
               liter_end = text_locs->end();

          while ( liter != liter_end )
          {
               if ( loc_cnt )
                    cout << ',';
               else ++loc_cnt;
               cout << '(' << (*liter).first
                    << ',' << (*liter).second << ')';
               ++liter;
          }
          cout << ")\n";
          ++iter;
     }

     cout << endl;
}
```

如果 map 中没有任何元素，调用我们的显示函数也不会有任何麻烦。判断 map 是否为空的一种办法是调用 size()函数：

```
if ( text_map->size() )
     display_map_text( text_map );
```

但是，没有必要对所有的元素进行计数，我们可以更直接地调用 empty()：

```
if ( ! text_map->empty() )
     display_map_text( text_map );
```

6.12.4 单词转换 map

下面的小程序说明了怎样创建、查找和迭代一个 map。该程序使用了两个 map，而单词转换 map 拥有两个 string 类型的元素。键（key）代表要求特殊处理的单词，值（value）表示我们遇到该词时应该采用怎样的转换。为简单起见，我们把 map 的所有项都固定写在代码中（作为练习，你可以泛化该程序，使其从标准输入或指定的文件读入“单词/转换”对）我们的统计 map 保存了被执行的转换的使用统计信息。程序如下：

```
#include <map>
#include <vector>
#include <iostream>
#include <string>
int main()
{
    map< string, string > trans_map;
    typedef map< string, string >::value_type valType;

    // 第一个权宜之计：将转换对固定写在代码中
    trans_map.insert( valType( "gratz", "grateful" ));
    trans_map.insert( valType( "'em", "them" ));
    trans_map.insert( valType( "cuz", "because" ));
    trans_map.insert( valType( "nah", "no" ));
    trans_map.insert( valType( "sez", "says" ));
    trans_map.insert( valType( "tanx", "thanks" ));
    trans_map.insert( valType( "wuz", "was" ));
    trans_map.insert( valType( "pos", "suppose" ));

    // ok: 显示 trans_map
    map< string,string >::iterator it;
    cout << "Here is our transformation map: \n\n";
    for ( it = trans_map.begin();
          it != trans_map.end(); ++it )
        cout << "key: " << (*it).first << "\t"
             << "value: " << (*it).second << "\n";
    cout << "\n\n";

    // 第二个权宜之计：固定写入文字.
    string textarray[14]={ "nah", "I", "sez", "tanx", "cuz", "I",
        "wuz", "pos", "to", "not", "cuz", "I", "wuz", "gratz" };
    vector< string > text( textarray, textarray+14 );
    vector< string >::iterator iter;

    // ok: 显示 text
    cout << "Here is our original string vector: \n\n";
    int cnt = 1;
    for ( iter = text.begin(); iter != text.end(); ++iter, ++cnt )
        cout << *iter << ( cnt % 8 ? " " : "\n" );

    cout << "\n\n\n";
```

```
    // 包含统计信息的 map—动态生成
    map< string,int > stats;
    typedef map< string,int >::value_type statsValType;

    // ok: 真正的 map 工作—程序的核心
    for ( iter = text.begin(); iter != text.end(); ++iter )
        if (( it = trans_map.find( *iter )) != trans_map.end() )
        {
            if ( stats.count( *iter ))
                stats[ *iter ] += 1;
            else stats.insert( statsValType( *iter, 1 ));
                *iter = (*it).second;
        }

    // ok: 显示被转换后的 vector
    cout << "Here is our transformed string vector: \n\n";
    cnt = 1;
    for ( iter = text.begin(); iter != text.end(); ++iter, ++cnt )
        cout << *iter << ( cnt % 8 << " " : "\n" );
    cout << "\n\n\n";
    // ok: 现在对统计 map 进行迭代
    cout << "Finally, here are our statistics:\n\n";
    map<string,int,less<string>,allocator>::iterator siter;
    for ( siter = stats.begin(); siter != stats.end(); ++siter )
        cout << (*siter).first << " "
            << "was transformed "
            << (*siter).second
            << ((*siter).second == 1
            << " time\n" : " times\n" );
}
```

程序执行时产生如下输出：

```
Here is our transformation map:

key: 'em value: them
key: cuz value: because
key: gratz    value: grateful
key: nah value: no
key: pos value: suppose
key: sez value: says
key: tanxvalue: thanks
key: wuz value: was

Here is our original string vector:

nah I sez tanx cuz I wuz pos
to not cuz I wuz gratz

Here is our transformed string vector:
no I says thanks because I was suppose
to not because I was grateful
```

```
Finally, here are our statistics:

cuz was transformed 2 times
gratz was transformed 1 time
nah was transformed 1 time
pos was transformed 1 time
sez was transformed 1 time
tanx was transformed 1 time
wuz was transformed 2 times
```

6.12.5 从 map 中删除元素

从 map 中删除元素的 erase()操作有三种变化形式。为了删除一个独立的元素，我们传递给 erase()一个键值或 iterator。为了删除一列元素，我们传递给 erase()一对 lieator。例如，如果我们打算让用户能从 text_map 删除元素，可以这样做：

```
string removal_word;
cout << "type in word to remove: ";
cin >> removal_word;
if ( text_map->erase( removal_word ))
     cout << "ok: " << removal_word << " removed\n";
else cout << "oops: " << removal_word << " not found!\n";
```

另外，我们可以在删除单词之前，检查它是否存在：

```
map<string,loc*>::iterator where;
where = text_map.find( removal_word );
if ( where == text_map->end() )
     cout << "oops: " << removal_word << " not found!\n";
else {
     text_map->erase( where );
     cout << "ok: " << removal_word << " removed!\n";
}
```

在 text_map 的实现中，我们存储了与每个单词相关联的多个位置。这种管理使实际位置值的存储和查询复杂化。一种替代的做法是为每个位置插入一个单词项。但是，一个 map 只能拥有一个键值的惟一实例。为了给同一个键提供多个项，我们必须使用 multimap。6.15 节将讲解关联容器类型 multimap。

练习 6.20

定义一个 map，它以家族姓氏为索引，以各家孩子名的 vector 作为值。向 map 中放入至少六项。通过下列动作来测试它：基于家族姓氏的用户查询，把孩子加入到一个家庭中，为另一个家庭加入三个孩子以及输出全部 map 项。

练习 6.21

扩展练习 6.20 中的 map，使其具有存储字符串对的 vector：孩子的名字和生日。改写练习 6.20 的实现，使其支持新的字符串对 vector。修改你的测试程序，并验证其正确性。

练习 6.22

列出可能的三种应用，它们都会用到 map。写出每个 map 的定义，说明怎样插入元素和获取元素。

6.13 创建单词排除集

map 中键/值对构成，好比一个地址和电话号码以人名为键值。相反地，set 只是键的集合。例如，一个公司可能定义一个集合 bad_checks，由在过去两年中有过不良账单的人名构成。当只想知道一个值是会存在时，set 是最合适的。例如，在接受我们的账单之前，公司可能想查询 bad_checks 看看是否存在我们的名字。

我们为自己的文本查询系统创建了一个单词排除 set，它包括没有语义的词，如 the、and、into、with 和 but 等等。（虽然这大大改善了单词索引的质量，但是它使我们无法定位到哈姆雷特的著名讲演的第一行“To be or not to be”。）我们在向 map 中输入元素之前，首先检查它是否出现在排除集中。如果是，则不把它放到 map 中。

6.13.1 定义 set 并放入元素

为了定义和使用关联容器类型 set，我们必须包含其相关的头文件：

```
#include <set>
```

下面是单词排除集合对象的定义：

```
set<string> exclusion_set;
```

用 insert 操作将单个元素插入到 set 中。例如：

```
exclusion_set.insert( "the" );
exclusion_set.insert( "and" );
```

另外，我们可以通过向 insert()提供一对 iterator 以便插入一个元素序列。例如，我们的文件查询系统允许用户指定一个单词文件，文件中的所有单词都将排除在 map 之外。如果用户不提供这样的文件，我们就用缺省的单词集填充单词排除 set：

```
typedef set< string >::difference_type diff_type;
set< string > exclusion_set;
ifstream infile( "exclusion_set" );

if ( ! infile )
{
    static string default_excluded_words[25] = {
        "the","and","but","that","then","are","been",
        "can","can't","cannot","could","did","for",
        "had","have","him","his","her","its","into",
        "were","which","when","with","would"
    };

    cerr << "warning! unable to open word exclusion file! -- "
        << "using default set\n";
```

```
        copy( default_excluded_words, default_excluded_words+25,
            inserter( exclusion_set, exclusion_set.begin() ));
    }
    else {
        istream_iterator\string,diff_type> input_set(infile),eos;
        copy( input_set, eos, inserter( exclusion_set,
            exclusion_set.begin() ));
    }
```

这段代码引入了两个我们没有见过的元素。difference_type 和 inserter 类。difference_type 是字符串 set 中两个 iterator 相减的结果类型。istream_iterator 用它作参数。

copy()是一个泛型算法（在第 12 章和附录中详细讨论）。它的前两个参数或者是 iterator，或者是指针，它们标记了要拷贝元素的范围。第二个参数或是 iterator 或是指针，它们指向目标容器中这些元素要被放置的起始处。

问题是，copy()期望容器的长度大于或等于要拷贝元素的个数。这是因为 copy()顺次赋值每个元素．它并没有插入元素。但是，关联容器不支持预分配长度。为了把元素拷贝到排除集中，必须使 copy()能插入而不是为每个元素赋值。inserter 类完成的正是这项工作（12.4 节将详细讨论）。

6.13.2 搜索一个元素

查询 set 对象中是否存在一个值的两个操作是 find()和 count()。如果元素存在，则 find()返回指向这个元素的 iterator，否则返回一个等于 end()的 iterator，表示该元素不存在。如果找到元素，count()返回 1，如果元素不存在，则返回 0。在 build_word_map()函数中，我们在向 map 中输入单词之前，增加了对 exclusion_set 的测试：

```
if ( exclusion_set.count( textword ))
    continue;
// ok: 把单词加入到 map 中
```

6.13.3 迭代一个 set 对象

为了练习我们的“单词/位置”map，我们实现了一个小函数，它允许查询单个单词（第 17 章将提议对完整查询语言的支持）。如果找到了单词，我们希望显示该词出现的行。但是，一个词可能在一行中出现多次，比如：

```
tomorrow and tomorrow and tomorrow
```

我们希望只显示该行一次。

实现每行只保留一个实例的一种策略就是使用 set，如下列代码段：

```
// 获得指向位置向量的指针
loc *ploc = (*text_map)[ query_text ];

// 对 "位置项对" 进行迭代
// 把每行插入到 set 中
set< short > occurrence_lines;
loc::iterator liter = ploc->begin(),
    liter_end = ploc->end();
```

```
while ( liter != liter_end ) {
    occurrence_lines.insert( occurrence_lines.end(),
        (*liter).first );
    ++liter;
}
```

set 只能含有每个键值的惟一实例。所以，occurrence_line 保证只包含单词出现行的单实例。为了显示这些文本行，只需迭代 set：

```
register int size = occurrence_lines.size();

cout << "\n" << query_text
    << " occurs " << size
    << (size == 1 << " time:" : " times:")
    << "\n\n";
set< short >::iterator it=occurrence_lines.begin();

for ( ; it != occurrence_lines.end(); ++it ) {
    int line = *it;
    cout << "\t( line "
        << line + 1 << " ) "
        << (*text_file)[line] << endl;
}
```

（query_text()的完整实现将在下节给出。）

set 支持操作 size()、empty()和 erase()，同上节描述的 map 类型相同。另外，泛型算法提供了一组 set 特有的函数，如 set_union()和 set_difference()（我们将在第 17 章利用它们来支持查询语言）。

练习 6.23

增加一个排除集，用来识别以“s”结尾、但结尾不应去掉、又没有一般规则可循的单词，例如，放在该集合中的三个词可能是名字 Pythagoras、Brahms 和 Burne_Jones。把这个排除集的用法放入 6.10 节的 suffix_s()中。

练习 6.24

建立一个 vector，里面是你下六个月里想看的书，以及一个 set，里面是你已经看过的书的题目。写一个程序，它从 vector 中为你选择一本没有读过的书。当它选择了一本书之后，应该把该书的题目放到 set 中。如果实际上你把该书放在一边没有看，它应该支持从已读书目的 set 中去掉这本书。六个月后，输出已读的书和还没有读的书。

6.14 完整的程序

本节将给出在这一章开发的、完整有效的程序，其中有两个修改：我们没有按照过程化程序设计的方式把数据结构和函数分开，而是引入了一个类 TextQuery 来封装它们（我们将在后面章节中更详细地了解类的使用）；文本的表示也做了修改，以便能够在当前可用的编

译器下通过编译。例如 iostream 库反映了标准 C++之前的实现版本（指编译器）。模板不支持模板参数的缺省值。为使程序能在你当前的系统上运行，或许需要修改某些声明：

```
// 标准库头文件
#include <algorithm>
#include <string>
#include <vector>
#include <utility>
#include <map>
#include <set>

// 标准 C++之前的 iostream 头文件
#include <fstream.h>

// 标准 C 头文件
#include <stddef.h>
#include <ctype.h>

// typedefs 使声明更简单
typedef pair<short,short> location;
typedef vector<location,allocator> loc;
typedef vector<string,allocator> text;
typedef pair<text*,loc*> text_loc;

class TextQuery {
public:
     TextQuery() { memset( this, 0, sizeof( TextQuery )); }
     static void
     filter_elements( string felems ) { filt_elems = felems; }
     void query_text();
     void display_map_text();
     void display_text_locations();
     void doit() {
          retrieve_text();
          separate_words();
          filter_text();
          suffix_text();
          strip_caps();
          build_word_map();
     }
private:
     void retrieve_text();
     void separate_words();
     void filter_text();
     void strip_caps();
     void suffix_text();
     void suffix_s( string& );
     void build_word_map();
private:
     vector<string,allocator> *lines_of_text;
```

```
    text_loc *text_locations;
    map< string,loc*,
    less<string>,allocator> *word_map;
    static string filt_elems;
};
string TextQuery::filt_elems( "\",.;:!<<)(\\/" );

int main()
{
    TextQuery tq;
    tq.doit();
    tq.query_text();
    tq.display_map_text();
}

void
TextQuery::
retrieve_text()
{
    string file_name;
    cout << "please enter file name: ";
    cin >> file_name;
    ifstream infile( file_name.c_str(), ios::in );
    if ( !infile ) {
        cerr << "oops! unable to open file "
             << file_name << " -- bailing out!\n";
        exit( - 1 );
    }
    else cout << "\n";
    lines_of_text = new vector<string,allocator>;
    string textline;
    while ( getline( infile, textline, '\n' ))
        lines_of_text->push_back( textline );
}

void
TextQuery::
separate_words()
{
    vector<string,allocator> *words = new vector<string,allocator>;
    vector<location,allocator> *locations =
        new vector<location,allocator>;

    for ( short line_pos = 0; line_pos < lines_of_text->size();
                        line_pos++ )
    {
        short word_pos = 0;
        string textline = (*lines_of_text)[ line_pos ];
        string::size_type eol = textline.length();
        string::size_type pos = 0, prev_pos = 0;
```

```
            while (( pos = textline.find_first_of( ' ', pos ))
                            != string::npos )
            {
                words->push_back(
                    textline.substr( prev_pos, pos - prev_pos ));
                locations->push_back(
                    make_pair( line_pos, word_pos ));
                word_pos++; pos++; prev_pos = pos;
            }
            words->push_back(
                textline.substr( prev_pos, pos - prev_pos ));
            locations ->push_back(make_pair(line_pos,word_pos));
        }
        text_locations = new text_loc( words, locations );
}

void
TextQuery::
filter_text()
{
    if ( filt_elems.empty() )
        return;
    vector<string,allocator> *words = text_locations ->first;

    vector<string,allocator>::iterator iter = words ->begin();
    vector<string,allocator>::iterator iter_end = words ->end();

    while ( iter != iter_end )
    {
        string::size_type pos = 0;
        while (( pos = (*iter).find_first_of( filt_elems, pos ))
                            != string::npos )
            (*iter).erase(pos,1);
        ++iter;
    }
}

void
TextQuery::
suffix_text()
{
    vector<string,allocator> *words = text_locations ->first;

    vector<string,allocator>::iterator iter = words ->begin();
    vector<string,allocator>::iterator iter_end = words ->end();

    while ( iter != iter_end )
    {
        if ( (*iter).size() <= 3 )
            { iter++; continue; }
```

```
            if ( (*iter)[ (*iter).size()- 1 ] == 's' )
                suffix_s( *iter );

            // 其他的后缀处理放在这里
            iter++;
        }
}
void
TextQuery::
suffix_s( string &word )
{
    string::size_type spos = 0;
    string::size_type pos3 = word.size()- 3;
    // "ous", "ss", "is", "ius"
    string suffixes( "oussisius" );
    if ( ! word.compare( pos3, 3, suffixes, spos, 3 ) ||
         ! word.compare( pos3, 3, suffixes, spos+6, 3 ) ||
         ! word.compare( pos3+1, 2, suffixes, spos+2, 2 ) ||
         ! word.compare( pos3+1, 2, suffixes, spos+4, 2 ))
         return;
    string ies( "ies" );
    if ( ! word.compare( pos3, 3, ies ))
    {
         word.replace( pos3, 3, 1, 'y' );
         return;
    }
    string ses( "ses" );
    if ( ! word.compare( pos3, 3, ses ))
    {
         word.erase( pos3+1, 2 );
         return;
    }

    // 去掉尾部的 's'
    word.erase( pos3+2 );

    // watch out for "'s"
    if ( word[ pos3+1 ] == '\'' )
         word.erase( pos3+1 );
}

void
TextQuery::
strip_caps()
{
    vector<string,allocator> *words = text_locations ->first;

    vector<string,allocator>::iterator iter = words ->begin();
    vector<string,allocator>::iterator iter_end = words ->end();
```

```
        string caps( "ABCDEFGHIJKLMNOPQRSTUVWXYZ" );
        while ( iter != iter_end ) {
            string::size_type pos = 0;
            while (( pos = (*iter).find_first_of( caps, pos ))
                                            != string::npos )
                (*iter)[ pos ] = tolower( (*iter)[pos] );
            ++iter;
        }
}

void
TextQuery::
build_word_map()
{
        word_map = new map< string, loc*, less<string>, allocator >;

        typedef map<string,loc*,less<string>,allocator>::value_type
            value_type;

        typedef set<string,less<string>,allocator>::difference_type
            diff_type;

        set<string,less<string>,allocator> exclusion_set;
        ifstream infile( "exclusion_set" );

        if ( !infile )
        {
            static string default_excluded_words[25] = {
                "the","and","but","that","then","are","been",
                "can","can't","cannot","could","did","for",
                "had","have","him","his","her","its","into",
                "were","which","when","with","would"
            };
            cerr << "warning! unable to open word exclusion file! -- "
                << "using default set\n";
            copy( default_excluded_words, default_excluded_words+25,
                inserter( exclusion_set, exclusion_set.begin() ));
        }
        else {
            istream_iterator< string, diff_type >
                input_set( infile ), eos;
            copy( input_set, eos,
                inserter( exclusion_set, exclusion_set.begin() ));
        }

        // 遍历单词，输入键/值对
        vector<string,allocator> *text_words = text_locations ->first;
        vector<location,allocator> *text_locs = text_locations ->second;
```

```
        register int elem_cnt = text_words ->size();
        for ( int ix = 0; ix < elem_cnt; ++ix )
        {
            string textword = ( *text_words )[ ix ];
            if ( textword.size() < 3 ||
                    exclusion_set.count( textword ))
                continue;
            if ( ! word_map->count((*text_words)[ix] ))
            { // 没有，添加：
                loc *ploc = new vector<location,allocator>;
                ploc->push_back( (*text_locs)[ix] );
                word_map->insert( value_type( (*text_words)[ix], ploc ));
            }
            else (*word_map)[(*text_words)[ix]]->
                push_back( (*text_locs)[ix] );
        }
}

void
TextQuery::
query_text()
{
        string query_text;
        do {
            cout << "enter a word against which to search the text.\n"
                << "to quit, enter a single character ==> ";
            cin >> query_text;
            if ( query_text.size() < 2 ) break;
            string caps( "ABCDEFGHIJKLMNOPQRSTUVWXYZ" );
            string::size_type pos = 0;
            while (( pos = query_text.find_first_of( caps, pos ))
                        != string::npos )
            query_text[ pos ] = tolower( query_text[pos] );

            // 如果对 map 索引，输入 query_text，如无
            // 说明没有要找的词
            if ( !word_map->count( query_text )) {
                cout << "\nSorry. There are no entries for "
                    << query_text << ".\n\n";
                continue;
            }
            loc *ploc = (*word_map)[ query_text ];
            set<short,less<short>,allocator> occurrence_lines;
            loc::iterator liter = ploc->begin(),
                liter_end = ploc->end();
            while ( liter != liter_end ) {
                occurrence_lines.insert(
```

```
                    occurrence_lines.end(), (*liter).first);
                ++liter;
            }
            register int size = occurrence_lines.size();
            cout << "\n" << query_text
                << " occurs " << size
                << (size == 1 << " time:" : " times:")
                << "\n\n";
            set<short,less<short>,allocator>::iterator
                    it=occurrence_lines.begin();
            for ( ; it != occurrence_lines.end(); ++it ) {
                int line = *it;
                cout << "\t( line "
                    // 不要用从 0 开始有
                    // 文本行把用户弄迷糊了
                    << line + 1 << " ) "
                    << (*lines_of_text)[line] << endl;
            }
            cout << endl;
        }
        while ( ! query_text.empty() );
        cout << "Ok, bye!\n";
}

void
TextQuery::
display_map_text()
{
        typedef map<string,loc*,less<string>,allocator> map_text;
        map_text::iterator iter = word_map->begin(),
            iter_end = word_map->end();
        while ( iter != iter_end ) {
            cout << "word: " << (*iter).first << " (";
            int loc_cnt = 0;
            loc *text_locs = (*iter).second;

            loc::iterator liter = text_locs->begin(),
                liter_end = text_locs->end();

            while ( liter != liter_end )
            {
                if ( loc_cnt )
                    cout << ",";
                else ++loc_cnt;

                cout << "(" << (*liter).first
                    << "," << (*liter).second << ")";
                ++liter;
            }
```

```
            cout << ")\n";
            ++iter;
        }

        cout << endl;
    }

    void
    TextQuery::
    display_text_locations()
    {
        vector<string,allocator> *text_words = text_locations ->first;
        vector<location,allocator> *text_locs = text_locations ->second;

        register int elem_cnt = text_words ->size();

        if ( elem_cnt != text_locs->size() )
        {
            cerr << "oops! internal error: word and position vectors "
                 << "are of unequal size \n"
                 << "words: " << elem_cnt << " "
                 << "locs: " << text_locs->size()
                 << " -- bailing out!\n";
            exit( - 2 );
        }

        for ( int ix = 0; ix < elem_cnt; ix++ )
        {
            cout << "word: " << (*text_words)[ ix ] << "\t"
                 << "location: ("
                 << (*text_locs)[ix].first << ","
                 << (*text_locs)[ix].second << ")"
                 << "\n";
        }
        cout << endl;
    }
```

练习 6.25

说明为什么要用专门的 inserter 迭代器来向单词排除 set 中加入元素。（这在 6.13.1 节有简要解释，将在 12.4.1 节详细讨论。）

```
set<string> exclusion_set;
ifstream infile( "exclusion_set" );
// ...
copy( default_excluded_words, default_excluded_words+25,
inserter( exclusion_set, exclusion_set.begin() ));
```

练习 6.26

我们原先的实现反映了过程化的解决方案——即一组全局函数，它们在一组未经封装的

独立数据结构上进行操作。最终的程序反映了另外一种解决方案，它把函数和数据结构封装在 TextQuery 类中。比较两种方式。它们的缺点和长处各是什么？

练习 6.27

在程序的这个版本中，用户被提示输入待处理的文本文件。更方便的实现会允许用户在程序命令行中指定文件——我们将在第 7 章中看到怎样支持程序的命令行参数。我们的程序应该支持其他哪些命令行选项？

6.15 multimap 和 multiset

map 和 set 只能包含每个键的单个实例，而 multiset 和 multimap 允许要被存储的键出现多次。例如，在电话目录中，有人可能希望为每个人相关联的每个电话号码提供单独的列表。按作者给出的文本列表可能要为每个题目提供单独的资料，或为文本中每个单词的每次出现给出单独的位置对。要使用 multimap 和 mulitset，我们必须包含相关的 map 和 set 头文件。

```
#include <map>
multimap< key_type, value_type > multimapName;

// 按 string 索引，存有 list<string>
multimap< string, list< string > > synonyms;
#include <set>
multiset< type > multisetName;
```

对于 multimap 或 multiset，一种迭代策略是联合使用由 find()返回的 iterator（指向第一个实例）和由 count()返回的值。（这样做能奏效，因为我们可以保证实例在容器中是连续出现的。）例如：

```
#include <map>
#include <string>

void code_fragment()
{
     multimap< string, string > authors;
     string search_item( "Alain de Botton" );

     // ...
     int number = authors.count( search_item );
     multimap< string,string >::iterator iter;
     iter = authors.find( search_item );
     for ( int cnt = 0; cnt < number; ++cnt, ++iter )
          do_something( *iter );

     // ...
}
```

更精彩的策略是使用由 multiset 和 multimap 的特殊操作 equal_range()返回的 iterator 对值。

如果这个值存在，则第一个 iterator 指向该值的第一个实例，且第二个 iterator 指向这个值的

最后实例的下一位置。如果最后的实例是 multiset 的末元素，则第二个 iterator 等于 end()。例如：

```
#include <map>
#include <string>
#include <utility>
void code_fragment()
{
     multimap< string, string > authors;
     // ...
     string search_item( "Haruki Murakami" );
     while ( cin && cin >> search_item )
          switch ( authors.count( search_item ))
          {
          // 不存在，继续往下走
          case 0:
               break;

          // 只有一项，使用普通的 find()操作
          case 1: {
               multimap< string,string >::iterator iter;
               iter = authors.find( search_item );
               // do something with element
               break;
               }

          // 出现多项 ...
          default:
          {
               typedef multimap< string,string >::iterator iterator;
               pair< iterator, iterator > pos;

               // pos.first 指向第一个出现
               // pos.second 指向值不再出现的位置
               pos = authors.equal_range( search_item );
               for ( ; pos.first != pos.second; pos.first++ )
                    // 对每个元素进行操作
          }
     }
}
```

插入和删除操作与关联容器 set 和 map 相同。equal_range()可以用来提供 iterator 对，以便标记出要被删除的多个元素的范围。例如：

```
#include <multimap>
#include <string>
typedef multimap< string, string >::iterator iterator;
pair< iterator, iterator > pos;
string search_item( "Kazuo Ishiguro" );
// authors 是一个 multimap<string,string>
// 这等价于 authors.erase( search_item );
pos = authors.equal_range( search_item );
```

```
authors.erase( pos.first, pos.second );
```

每次插入增加一个元素。例如：

```
typedef multimap<string,string>::value_type valType;
multimap<string,string> authors;

// 引入 Barth 下的第一个键
authors.insert( valType(
     string( "Barth, John" ),
     string( "Sot-Weed Factor" )));

// 引入 Barth 下的第二个键
authors.insert( valType(
     string( "Barth, John" ),
     string( "Lost in the Funhouse" )));
```

不支持下标操作是访问 multimap 元素的一个限制。例如：

```
authors[ "Barth, John" ];    // 错误: multimap
```

将导致编译错误。

练习 6.28

用 multimap 重新实现 6.14 节的文本查询程序，multiset 中的每个位置都是独立输入的。两个方案的性能和设计特性各是什么？你认为哪一个更好？为什么？

6.16 栈

在 4.5 节中，我们通过栈抽象的实现说明了递增和递减操作符的用法。一般地，在程序执行过程中可能动态出现多个嵌套状态时，为了维护当前的状态，栈（stack）机制提供了一种有力的解决方案。因为栈是一种重要的数据抽象，所以标准 C++库提供了相应的类实现。为了使用它，我们必须包含相关的头文件：

```
#include <stack>
```

标准库提供的栈与我们的实现有点不同，在我们的实现中，它把对栈顶元素的访问和删除分别独立放到 top()和 pop()操作中。栈容器（stack container）支持的全部操作集如下所示：

表格 6.5 栈容器支持的操作

操作	功能
empty()	如果栈为空，则返回 true，否则返回 false。
size()	返回栈中元素的个数
pop()	删除，但不返回栈顶元素
top()	返回，但不删除栈顶元素
push(item)	放入新的栈顶元素

下面的程序练习了这五个栈操作：

```
#include <stack>
#include <iostream>

int main()
{
    const int ia_size = 10;
    int ia[ia_size ] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    // 填充 stack
    int ix = 0;
    stack< int > intStack;
    for ( ; ix < ia_size; ++ix )
        intStack.push( ia[ ix ] );
    int error_cnt = 0;
    if ( intStack.size() != ia_size ) {
        cerr << "oops! invalid intStack size: "
             << intStack.size()
             << "\t expected: " << ia_size << endl;
        ++error_cnt;
    }
    int value;
    while ( intStack.empty() == false )
    {
        // 读取栈顶元素
        value = intStack.top();
        if ( value != --ix ) {
            cerr << "oops! expected " << ix
                 << " received " << value
                 << endl;
            ++error_cnt;
        }

        // 弹出栈顶元素，并重复
        intStack.pop();
    }
    cout << "Our program ran with "
        << error_cnt << " errors!" << endl;
}
```

如下声明：

```
stack< int > intStack;
```

将声明 intStack 为一个整型元素的空栈。栈类型被称为**容器适配器**（container adapter），因为它把栈抽象施加在底层容器集上、缺省情况下，栈用容器类型 deque 实现，因为 deque 为容器前端的插入和删除提供了有效支持，而 vector 则不。如果我们希望改写这种缺省的实现，则可以定义一个栈对象，以提供显式的顺序容器类型作为第二个参数。例如：

```
stack< int, list<int> > intStack;
```

栈的元素被按值输入：每个对象被拷贝到底层的容器中。对大型或复杂类对象，这种方法可能过于昂贵，尤其是在我们只是读取元素的情况下。一种取代的存储策略是定义一个指针栈。例如：

```
#include <stack>
class NurbSurface { /* mumble */ };
stack< NurbSurface* > surf_Stack;
```

同一类型的两个栈可以比较相等、不相等、小于、大于、小于等于以及大于等于关系，只要底层元素类型支持等于和小于操作符即可。对于这些操作，栈中元素被依次比较、第一对不相等的元素决定了小于或大于关系。

我们将在 17.7 节支持复杂的用户文本查询如：

```
Civil && ( War || Rights )
```

时说明栈的用法。

6.17 队列和优先级队列

队列（queue）抽象体现了先进先出（FIFO，即“first in，first out”）的存储和检索策略。进入队列的对象被放在尾部，下一个被取出的元素取自队列的首部。标准库提供了两种风格的队列：FIFO 队列（就称作 queue），以及 priority_queue（优先级队列）。

priority_queue 允许用户为队列中包含的元素项建立优先级。它没有把新元素放在队列尾部，而是放在比它优先级低的元素前面。定义优先级队列的用户决定怎样确定优先级。在实践中，优先级队列的一个实例是机场行李检查队列。在 15 分钟后即将离港航班的乘客通常会被移到队列前面，以便他们能在飞机起飞前完成检查过程。优先级队列的大程序上的例子是操作系统的调度表，它可以决定在大量等待进程中下一个要执行的进程。

要使用这两种队列，必须包含相关的头文件：

```
#include <queue>
```

队列和 priority_queue 支持的全部操作见表 6.6：

表 6.6 队列和优先级队列支持的操作

操 作	功 能
empty()	如果队列为空，则返回 true，否则返回 false
size()	返回队列中元素的个数
pop()	删除，但不返回队首元素。在 priority_queue 中，队首元素代表优先级最高的元素
front()	返回，但不删除队首元素。它只能应用在一般队列上
back()	返回，但不删除队尾元素。它只能应用在一般队列上
top()	返回，但不删除 priority_queue 的优先级最高的元素。只能应用在 priority_queue 上
push(item)	在队尾放入一个新元素：对于 priority_queue，将根据优先级排序

priority_queue 的元素被强加了顺序关系，以便元素可以从大到小管理。这里所谓最大即等价于拥有最高优先级。缺省情况下，元素的优先级由底层元素类型相关的小于操作符执行。如果希望改写缺省的小于操作符，我们可以显式地提供一个函数或函数对象来排序优先级队列的元素（12.3 节将进一步解释和说明这种用法）

6.18 回顾 iStack 类

4.15 节给出的 iStack 类有两个方面的限制：

1．它只支持一种类型：int 型。我们希望支持所有的类型。可以通过对它做些转换，使它变成一个通用的 Stack 模板类。

2．它的长度固定。这在两个方面存在问题：栈可能变满，因而不能使用。为避免栈满，我们为它分配了过大的平均存储空间。解决方案是支持栈的动态增长。我们可以通过直接使用由底层 vector 对象提供的动态支持来实现它。

在开始之前，先给出 Stack 的原始定义；

```
#include <vector>
class iStack {
public:
     iStack( int capacity )
     : _stack( capacity ), _top( 0 ) {}

     bool pop( int &value );
     bool push( int value );

     bool full();
     bool empty();
     void display();

     int size();
private:
     int _top;
     vector< int > _stack;
};
```

让我们先把它转换成支持动态分配的类。这意味着，我们必须插入和删除元素，而不是索引固定长度的 vector，即不再需要数据成员_top。通过 push_back()和 pop_back()就可以自动管理栈顶元素。下面是修改后的 pop()和 push()实现。

```
bool iStack::pop( int &top_value )
{
     if ( empty() )
          return false;
     top_value = _stack.back(); _stack.pop_back();
     return true;
}

bool iStack::push( int value )
{
```

```
        if ( full() )
            return false;
        _stack.push_back( value );
        return true;
    }
```

empty()、size()以及 full()也必须重新实现——在这个版本中，它们与底层 vector 的耦合更加紧密：

```
    inline bool iStack::empty(){ return _stack.empty(); }
    inline int iStack::size() { return _stack.size(); }
    inline bool iStack::full() {
            return _stack.max_size() == _stack.size(); }
```

display()需要稍作修改，去掉_top 作为 for 循环结束条件的用法：

```
    void iStack::display()
    {
        cout << "( " << size() << " )( bot: ";
        for ( int ix = 0; ix < size(); ++ix )
            cout << _stack[ ix ] << " ";
        cout << " :top )\n";
    }
```

惟一比较重要的变化就是必须修改 iStack 的构造函数。严格来说，我们的构造函数不再需要做任何事情，对于重新实现的 iStack 类，下面的空构造函数已经足够了：

```
    inline iStack::iStack() {}
```

但是，对用户来说这是不够的。因此，我们完全保留了原来的接口，这样，现有的用户代码就不需要重写。必须维持一个单参数的构造函数，尽管我们不再像原来的版本那样需要这个参数。修改后的接口接收一个 int 参数，但是实际上并不需要这个参数：

```
    class iStack {
    public:
        iStack( int capacity = 0 );
        // ...
    };
```

如果出现了这个参数，该怎么办呢？我们会用它设置 vector 的容量：

```
    inline iStack::iStack( int capacity )
    {
        if ( capacity )
            _stack.reserve( capacity );
    }
```

从非模板向模板类的转换相当简单，部分原因是由于底层 vector 对象已经属于类模板。下面是修改后的类定义：

```
    #include <vector>

    template <class elemType>
    class Stack {
    public:
```

```
        Stack( int capacity=0 );
        bool push( elemType value );
        bool full();
        bool empty();
        void display();
        int size();

    private:
        vector< elemType > _stack;
    };
```

为了保持与使用早期 iStack 类实现的已有程序的兼容性，我们提供下列 typedef：

```
typedef Stack<int> iStack;
```

成员操作的修改留作练习。

练习 6.29

为动态 Stack 类模板重新实现 peek()函数（4.15 节的练习 4.23）。

练习 6.30

为 Stack 类模板提供修改后的成员操作，运行 4.15 节的测试程序测试新的实现。

练习 6.31

利用 5.11.1 节的 List 类的模型，把我们的 Stack 类模板封装到 Primer_Third_Edtion 名字空间中。

第三篇

基于过程的程序设计

第二篇介绍了 C++程序设计语言的基本要素：内置数据类型（如 int 和 double），类抽象类型（如 string 和 vector），以及在这些类型上执行的操作。在第三篇中，我们将看到这些基本程序要素怎样组成函数定义。函数主要用来实现各种算法，由这些算法执行程序中特定的任务。

对于每个 C++程序，我们都必须定义一个称作 main()的函数，它是 C++程序开始执行时第一个调用的函数。main()函数再调用其他函数来完成程序所要求的任务。程序中的函数通信（或称信息交换）都是通过函数接收的值（称为**参数**，parameter）以及函数返回的值来完成的。第 7 章将介绍 C++的函数机制。

函数可用来把程序组织成小的、独立的单元。每个函数封装一个或者一组算法，这些算法又分别应用在特定的数据集上。我们可以声明对象和类型，以让它们在整个程序中使用。但是，如果这些对象或类型只在程序的一个子集中被使用，那么，比较好的做法是将其限制在被访问的范围内，并把它们的声明与用到它们的函数相关联。**域**（scope）是一种机制，它可以由程序员用来限制程序中声明的可视性。在第 8 章中，我们将介绍 C++支持的不同的域，我们还将了解域怎样影响声明的可视性，以及 C++对象的生命期和运行时刻属性。

C++为简化程序中函数的用法提供了许多设施。在本篇中，我们将依次回顾这些设施。第一个设施是重载函数（overloaded function），它提供了公共的、但应用于不同数据类型上的操作，这样实现不同的函数可以共享同一个名字。例如，输出值类型不同的函数，比如整型。字符串等等，都可以被命名为 print()。这简化了函数的用法，因为程序员不必为相同的操作而记住不同的函数名。编译器根据函数参数的类型选择要调用的相应函数。第 9 章将讨论怎样声明和使用重载函数，以及对于给定的函数调用，编译器将怎样在一组重载函数中选择。

C++支持的简化函数用法的第二种设施是函数模板（function template）。函数模板是一个通用的（generic）函数定义，用来自动生成一组无限多的函数定义，它们类型不同，但具体实现维持不变数。第 10 章将描述怎样定义函数模板，以及怎样用它生成或实例化函数定义。

程序的函数通过接收值（又称参数）以及返回值来通信。但是，在程序执行时遇到特殊情况或程序出现异常状态时，这种机制就不够用了。这样的情况称作**异常**（exception），需要立即给予注意，并要求函数马上与调用函数通信，告知出现了异常。C++提供了异常处理

设施，允许在这些非正常情况下函数之间的通信。异常处理将在第 11 章中讲述。

最后，C++标准库还提供了一个称为泛型算法（generic algrithm）的常用函数扩展集。第 12 章将描述 C++标准库提供的泛型算法，并探讨它们是怎样与第 6 章中的容器类型以及内置数组类型进行交互的。

7

函　数

现在我们已经知道怎样声明变量（第 3 章），怎样写表达式（第 4 章）和语句（第 5 章），本章我们将了解怎样把它们组织到函数定义中，以便在程序中能够重用这些语言要素。本章将描述怎样声明和定义函数，以及怎样在程序中调用它们。本章将给出函数可以接受的不同类型的参数，以及各种类型参数的属性，此外，还将给出不同类型的返回值。然后我们再分析四种特殊的函数类型：内联（inline）函数、递归函数、用链接指示符（linkage directive）声明的非 C++函数。以及 main()函数。最后，用一个更高级的话题——函数指针来结束本章。

7.1 概述

函数可以被看作是一个由用户定义的操作。一般来说，函数由一个名字来表示。函数的操作数，称为**参数**（parameter），由一个位于括号中、并且用逗号分隔的**参数表**（parameter list）指定。函数的结果被称为**返回值**（return value），返问值的类型被称为函数**返回类型**（return type）。不产生值的函数，返回类型是 void，意思是什么都不返回。函数执行的动作在函数体（body）中指定。函数体包含在花括号中，有时也称为**函数块**（function block）。函数返回类型、以及其后的函数名、参数表和函数体构成了函数定义。下面是函数定义的一些例子：

```
inline int abs( int iobj )
{
    // 返回 iobj 的绝对值
    return( iobj < 0 ? -iobj : iobj );
}
inline int min( int p1, int p2 )
{
    // 返回较小值
    return( p1 < p2 ? p1 : p2 );
}
int gcd( int v1, int v2 )
{
    // 返回最大公约数
    while ( v2 )
```

```
    {
        int temp = v2;
        v2 = v1 % v2;
        v1 = temp;
    }
    return v1;
}
```

当函数名后面紧跟着调用操作符“()”时，这个函数就被执行了。如果函数被定义为应该接收参数，则在调用这个函数时，就需要为这些参数提供**实参**（argument）。且这些实参被放在调用操作符中，而两个相邻的实参用逗号分隔。这种安排称为“**向函数传递参数**（passing argument）”。在下面的例子中，main()调用了 abs()两次、min()和 gcd()各一次。main()被定义在文件 main.C 中。

```
#include <iostream>
int main()
{
    // get values from standard input
    cout << "Enter first value: ";
    int i;
    cin >> i;
    if ( !cin ) {
        cerr << "!<< Oops: input error - Bailing out!\n";
        return -1;
    }
    cout << "Enter second value: ";
    int j;
    cin >> j;
    if ( !cin ) {
        cerr << "!<< Oops: input error - Bailing out!\n";
        return -2;
    }
    cout << "\nmin: " << min( i, j ) << endl;
    i = abs( i );
    j = abs( j );
    cout << "gcd: " << gcd( i, j ) << endl;
    return 0;
}
```

函数调用会导致两件事情发生。如果函数已经被声明为 inline（内联），则函数体可能已经在编译期间它的调用点上就被展开。如果没有被声明为 inline，则函数在运行时才被调用。函数调用会使程序控制权被传送给正在被调用的函数，而当前活动函数的执行被挂起。当被调用的函数完成时，主调函数在调用语句之后的语句上恢复执行。函数在执行完函数体的最后一条语句或遇到**返回语句**（return statement）后完成。

我们必须在调用函数之前就声明该函数，否则会引起编译错误。当然，函数定义也可以被用作声明。但是，函数在程序中只能被定义一次。典型情况下，函数定义被放在单独的程序文本文件中，或者与其他相关的函数定义放在同一个文本文件中。要想在其他文件而不是包含函数定义的文件中使用该函数，我们必须要用到另外一种函数声明机制。

函数声明由函数返回类型、函数名和参数表构成。这三个元素被称为**函数声明**（function declaration）或**函数原型**（function prototype）。一个函数可在一个文件中被声明多次。

在我们的 main.C 例子中，如果在 main()之前没有定义函数 abs()、min()和 gcd()，那么在 main()中对它们的调用将导致编译错误。然而，要使 main.C 无编译错误，并不要求我们一定在 main()之前定义它们，只需如下声明（函数声明不需指定参数的名字，只需要每个参数的类型）：

```
int abs( int );
int min( int, int );
int gcd( int, int );
```

函数声明（以及 inline 函数的定义）最好放在头文件中。这些头文件可以被包含（include）在每个调用该函数的文件中。通过这种方式，所有文件共享一个公共的声明，如果需要修改此声明，则只有这一个实例需要被改变。程序的头文件可如下定义，我们把它称做 localMath.h：

```
// gcd.C 中的定义
int gcd( int, int );

inline int abs(int i) {
     return( i<0 ? -i : i );
}

inline int min(int v1,int v2) {
     return( v1<v2 ? v1 : v2 );
}
```

函数声明描述了函数的**接口**（interface）。它描述了函数必须接收的信息类型（参数表），以及它返回的信息类型（返回类型），如果存在返回值的话。作为函数的一个用户，我们对它的接口进行编程。只要函数的接口不变，无论函数修改多么频繁，也无需改变我们的代码。把函数接口传递给用户的机制就是把函数的声明放在头文件中，如头文件 localMath.h。

编译并运行 main.C 时，已知用户给出下列输入值：

```
Enter first value: 15
Enter second value: 123
```

程序产生下列结果：

```
min: 15
gcd: 3
```

7.2 函数原型

函数原型由函数返回类型、函数名以及参数表构成。函数原型描述的是函数的接口，它详细描述了调用函数时需要提供的参数的类型和个数，以及函数返回值的类型。本节我们将更详细地讨论函数原型的特性。

7.2.1 函数返回类型

函数返回类型可以是预定义类型（如 int 或 double）、复合类型（如 int&或 double*）

用户定义类型（如枚举、类或 void，后者意指函数不返回值）。下面的例子给出了一些函数可能的返回值：

```
#include <string>
#include <vector>

class Date { /* 定义 */ };

bool look_up( int *, int );
double calc( double );
int count( const string &, char );
Date& calendar( const char* );
void sum( vector<int>&, int );
```

函数类型和内置数组类型不能用作返回类型。例如，下面列举了一个这样的错误：

```
// 非法：数组不能作返回类型
int[10] foo_bar();
```

我们必须返回一个指向数组中元素类型的指针：

```
// ok：指向数组的第一个元素的指针
int *foo_bar();
```

指向数组第一个元素的指针被返回。（如何得到数组的长度，那是处理返回值的用户要负责的事了。）

但是，类类型和容器类型可以被直接返回。例如：

```
// ok：返回类型是 char 的 list
list<char> foo_bar();
```

（但是，这种方式效率比较低。按值返回的讨论见 7.4 节。）

函数必须指定一个返回值，没有显式返回值的声明或者定义将引起编译错误。例如：

```
// 错误：没有返回类型
const is_equal( vector<int> v1, vector<int> v2 );
```

在 C++标准化之前，如果缺少显式返回类型的话，返回值会被假定为 int 类型。在标准 C++中，返回类型不能被省略。is_equal()的正确声明是：

```
// ok：返回类型已被指定
const bool is_equal( vector<int> v1, vector<int> v2 );
```

7.2.2 函数参数表

函数的参数表不能省略。没有任何参数的函数可以用空参数表或含有单个关键字 void 的参数表来表示。例如，下面有关 fork()的两个声明是等价的：

```
int fork();                    // 隐式的 void 参数表
int fork( void );              // 等价声明
```

参数表由逗号分隔的参数类型列表构成。每个参数类型之后可以跟一个名字，它是可选的。参数表中使用逗号分隔的简写方式（即在声明中使用的方式）是错误的。例如：

```
int manip( int v1, v2 );       // 错误
int manip( int v1, int v2 );   // ok
```

参数表中不能出现同名的参数。函数定义的参数表中的参数名允许在函数体中访问这个参数。函数声明中的参数名不是必需的，如果名字存在的话，它应该被用作辅助文档。例如：

```
void print( int *array, int size );
```

为同一函数的声明和定义中的参数指定不同的名字，在语言上没有错误。但是，程序的读者可能会被弄糊涂。

在 C++中，两个函数可能同名但参数表不同：这种函数被称为**重载函数**（overloaded function）。参数表称为函数的**符号特征**（signature），因为它被用来区分函数的不同实例。有了名字和符号特征就可以惟一地标识函数了。第 9 章将更完整地讨论重载函数。

7.2.3 参数类型检查

函数 gcd()的声明如下：

```
int gcd( int, int );
```

这个声明指出函数有两个 int 型的参数。函数的参数表为编译器提供了必需的信息，使它能够在函数调用时对给出的实参进行类型检查。例如，如果实参是 const char*，会发生什么情况呢？比如，下面调用的结果是什么？

```
gcd( "hello", "world" );
```

或者，如果向函数 gcd()传递了一个或多于两个的实参，又会发生什么情况？如果不小心将 24 和 312 连接在一起，又会怎么样？

```
gcd( 24312 );
```

在编译 gcd()的后两个调用时，惟一期望的结果就是编译错误。任何希望执行该调用的企图都会导致灾难性的后果。在 C++中，这两个调用将导致如下形式的编译错误信息：

```
// gcd( "hello", "world" )
error: invalid argument types ( const char*, const char* ) --
     expecting ( int, int )

// gcd( 24312 )
error: missing value for second argument
```

如果两个参数都是 double 型，又会怎么样呢？该调用应该被标记为错误吗？

```
gcd( 3.14, 6.29 );
```

正如 4.14 节中所示，double 型的值可以被转换成 int 型的值。因此，把该调用标记为错误有些过于严格。参数被隐式地转换成 int（通过截取）就能满足参数表的类型要求。但是，因为这是可能带有精度损失的窄化转换，编译器一般都会产生一个警告。调用变为：

```
gcd( 3, 6 );
```

返回值是 3。

C++是一种**强类型**（strong typed）语言。每个函数调用的实参在编译期间都要经过**类型检查**（type-checked）。若实参类型与相应的参数类型不匹配，如果有可能，就会应用一个隐式的类型转换，如上个例子中从 double 到 int 的转换。如果不可能进行隐式转换或者实参的

个数不正确，就会产生一个编译错误。这就是函数必须先被声明才能被使用的原因。编译器必须根据函数参数表，对函数调用的实参执行类型检查，就此而言，声明是必不可少的。

省略实参或者传递类型错误的实参都是 C 语言标准化之前严重运行时刻错误的根源。由于 C++引入了强类型检查，这些接口错误都将在编译时被捕捉到。

练习 7.1

下列哪些函数原型是无效的？为什么？

```
(a) set( int *, int );
(b) void func();
(c) string error( int );
(d) arr[10] sum( int *, int );
```

练习 7.2

请为下列函数写出函数原型：

a）函数名为 compare，有两个参数，它们是名为 matrix 的类的引用，返回类型为 bool；
b）函数名为 extract，没有参数，返回值为整型 set（这里的 set 是 6.13 节定义的容器类型）。

练习 7.3

已知下列声明，哪些函数调用是错误的，为什么？

```
double calc( double );
int count( const string &, char );
void sum( vector<int> &, int );
vector<int> vec( 10 );
(a) calc( 23.4, 55.1 );
(b) count( "abcda", 'a' );
(c) sum( vec, 43.8 );
(d) calc( 66 );
```

7.3 参数传递

所有的函数都使用在程序**运行栈**（run-time stack）中分配的存储区。该存储区一直保持与该函数相关联，直到函数结束为止。那时，存储区将自动释放以便重新使用。该函数的整个存储区被称为**活动记录**（activation record）。

系统在函数的活动记录中为函数的每个参数都提供了存储区。参数的存储长度由它的类型来决定。参数传递是指用函数调用的实参值来初始化函数参数存储区的过程。

C++中参数传递的缺省初始化方法是把实参的值拷贝到参数的存储区中。这被称为**按值传递**（pass-by-value）。

按值传递时，函数不会访问当前调用的实参。函数处理的值是它本地的拷贝，这些拷贝被存储在运行栈中，因此改变这些值不会影响实参的值。一旦函数结束了，函数的活动记录

将从栈中弹出，这些局部值也就消失了。

在按值传递的情况下，实参的内容没有被改变。这意味着程序员在函数调用时无需保存和恢复实参的值。如果没有按值传递机制，那么每个没有被声明为 const 的参数就可能会随每次函数调用而被改变。按值传递的危害最小，需要用户做的工作也最少。毫无疑问，按值传递是参数传递合理的缺省机制。

但是，按值传递并不是在所有的情况下都适合。不适合的情况包括：

- 当大型的类对象必须作为参数传递时。对实际的应用程序而言，分配对象并拷贝到栈中的时间和空间开销往往过大。
- 当实参的值必须被修改时。例如，在函数 swap()中，用户想改变实参的值，但是在按值传递的情况下无法做到。

```
// swap() 没有交换两个实参的值!
void swap( int v1, int v2 ) {
     int tmp = v2;
     v2 = v1;
     v1 = tmp;
}
```

swap()交换实参的本地拷贝，代表 swap()实参的变量并没有被改变。这将在下面调用 swap()的程序中可以看出来；

```
#include <iostream>
void swap( int, int );
int main() {
     int i = 10;
     int j = 20;
     cout << "Before swap():\ti: "
          << i << "\tj: " << j << endl;
     swap( i, j );
     cout << "After swap():\ti: "
          << i << "\tj: " << j << endl;
     return 0;
}
```

编译并执行程序产生如下结果：

```
Before swap():     i: 10  j: 20
After swap():      i: 10  j: 20
```

为了获得期望的行为，程序员可以使用两种方法。一种方法是，参数被声明成指针。例如，swap()可重写如下：

```
// pswap()交换 v1 和 v2 指向的值
void pswap( int *v1, int *v2 ) {
     int tmp = *v2;
     *v2 = *v1;
     *v1 = tmp;
}
```

我们必须修改 main()来调用 pswap()。现在程序员必须传递两个对象的地址而不是对象本身：

```
pswap( &i, &j );
```

修改后的程序编译运行后的结果显示了它的正确性：

```
// 使用指针使程序员能够访问当前调用的实参
Before swap():    i: 10  j: 20
After swap(): i: 20    j: 10
```

第二种方法是把参数声明成引用。例如，swap()可重写如下：

```
// rswap() 交换 v1 和 v2 引用的值
void rswap( int &v1, int &v2 ) {
    int tmp = v2;
    v2 = v1;
    v1 = tmp;
}
```

main()中 rswap()的调用看起来像原来的 swap()调用：

```
rswap( i, j );
```

编译并运行这程序会显示 i 和 j 的值已经被正确交换了。

7.3.1 引用参数

把参数声明成引用，实际上改变了缺省的按值传递参数的传递机制。在按值传递时，函数操纵的是实参的本地拷贝。当参数是引用时，函数接收的是实参的左值而不是值的拷贝。这意味着函数知道实参在内存中的位置，因而能够改变它的值或取它的地址。

什么时候将一个参数指定为引用比较合适呢？像 swap()的情况，它必须将一个参数改变成指针来允许改变实参的值时就比较合适。引用参数的第二种普遍用法是向主调函数返回额外的结果。第三种用法是向函数传递大型类对象。我们将更详细地查看后两种情况。

作为“通过引用参数向主调函数返回额外结果”的函数的一个例子，我们来定义一个被称为 look_up()的函数，它在整型 vector 中查找一个特定的值。如果找到了该值，则 look_up()返回一个指向含有该值的 vector 元素的 iterator（迭代器）。否则，返回一个指向 vector 最后一个元素下一位置的 iterator，表明该值不存在。在多次出现的情况下，指向第一次出现的 iterator 被返回。此外，look_up()用引用参数 occurs 返回该值出现的次数。

```
#include <vector>

// 引用参数 'occurs' 可以含有第二个返回值
vector<int>::const_iterator look_up(
    const vector<int> &vec,
    int value,          // 值在 vector 中吗?
    int &occurs )       // 多少次?
{

    // res_iter 被初始化为最后一个元素的下一位置
    vector<int>::const_iterator res_iter = vec.end();
    occurs = 0;
```

```
    for ( vector<int>::const_iterator iter = vec.begin();
                                      iter != vec.end();
                                      ++iter )
        if ( *iter == value )
        {
            if ( res_iter == vec.end() )
                res_iter = iter;
            ++occurs;
        }
    return res_iter;
}
```

把一个参数声明成引用的第三种情况是在向函数传递一个大型类对象时。在按值传递情况下，整个对象将随每次调用而被拷贝。尽管按值传递对内置数据类型的对象和小型类对象比较满意，但是对于大型类对象，它的效率就太低了。使用引用参数，函数可以访问被指定为实参的类对象，而不必在函数的活动记录中拷贝它。例如：

```
class Huge { public: double stuff[1000]; };
extern int calc( const Huge & );
int main() {
    Huge table[ 1000 ];

    // ... 初始化 table
    int sum = 0;
    for ( int ix=0; ix < 1000; ++ix )
        // 函数 calc() 将指向 Huge 类型的数组元素指定为实参
        sum += calc( table[ix] );

    // ...
}
```

有人可能希望用引用参数以避免拷贝用作实参的大型类对象，同时，又希望防止函数修改实参的值。如果引用参数不希望在被调用的函数内部被修改，那么把参数声明为 const 型的引用是个不错的办法。这种方式能够使编译器防止无意的改变。例如，下列程序段违反了 foo()的参数 xx 的常量性，因为 foo_bar()的参数不是 const 型的引用，所以我们不能保证 foo_bar()不会改变参数 xx 的值。这违反了 foo()的参数 xx 的常量性，程序被编译器标记为错误：

```
class X;
extern int foo_bar( X& );

int foo( const X& xx ) {
    // 错误: const 传递给非 const
    return foo_bar( xx );
}
```

为使该程序通过编译，我们改变 foo_bar()的参数的类型。以下两种声明都是可以接受的：

```
extern int foo_bar( const X& );
extern int foo_bar( X ); // 按值传递
```

或者可以传递一个 xx 的拷贝做实参，允许 foo_bar()改变它：

```
int foo( const X &xx ) {
    // ...
    X x2 = xx; // 拷贝值

    // 当 foo_bar() 改变它的引用参数时，x2 被改变，xx 保持不变
    return foo_bar( x2 ); // ok
}
```

我们可以声明任意内置数据类型的引用参数。例如，如果程序员想修改指针本身，而不是指针引用的对象，那么他可以声明一个参数，该参数是一个指针的引用。例如，下面是交换两个指针的函数：

```
void ptrswap( int *&v1, int *&v2 ) {
    int *tmp = v2;
    v2 = v1;
    v1 = tmp;
}
```

如下声明：

```
int *&v1;
```

应该从右向左读：v1 是一个引用，它引用一个指针，指针指向 int 型的对象。用函数 main()操纵函数 rswap()，我们可以如下修改代码以便交换两个指针值：

```
#include <iostream>

void ptrswap( int *&v1, int *&v2 );
int main() {
    int i = 10;
    int j = 20;

    int *pi = &i;
    int *pj = &j;

    cout << "Before ptrswap():\tpi: "
         << *pi << "\tpj: " << *pj << endl;
    ptrswap( pi, pj );

    cout << "After ptrswap():\tpi: "
         << *pi << "\tpj: " << *pj << endl;
    return 0;
}
```

编译并运行程序，产生下列输出：

```
Before ptrswap(): pi: 10 pj: 20
After ptrswap():  pi: 20 pj: 10
```

7.3.2 引用和指针参数的关系

现在你或许想知道，应该将一个函数声明成引用还是指针呢。确实，这两种参数都允许函数修改实参指向的对象。两种类型的参数都允许有效地向函数传递大型类对象。所以，怎

样决定把函数参数声明成引用还是指针呢？

正如 3.6 节所提到的，引用必须被初始化为指向一个对象，一旦初始化了，它就不能再指向其他对象。指针可以指向一系列不同的对象也可以什么都不指向。

因为指针可能指向一个对象或没有任何对象，所以函数在确定指针实际指向一个有效的对象之前不能安全地解引用（dereference）一个指针。例如：

```
class X;
void manip( X *px )
{
    // 在解引用指针之前确信它非 0
    if ( px != 0 )

    // 解引用指针
}
```

另一方面，对于引用参数，函数不需要保证它指向一个对象。引用必须指向一个对象，甚至在我们不希望这样时也是如此。例如：

```
class Type { };
void operate( const Type& p1, const Type& p2 );
int main() {
    Type obj1;
    // 设置 obj1 为某个值
    // 错误：引用参数的实参不能为 0
    Type obj2 = operate( obj1, 0 );
}
```

如果一个参数可能在函数中指向不同的对象，或者这个参数可能不指向任何对象，则必须使用指针参数。

引用参数的一个重要用法是，它允许我们在有效地实现重载操作符的同时，还能保证用法的直观性。重载操作符的完整讨论见第 15 章。让我们从下面的例子开始，它使用了 Matrix 类类型。我们想支持两个 Matrix 类对象的加法和赋值操作符，使它们的用法同内置类型一样“自然”：

```
Matrix a, b, c;
c = a + b;
```

Matrix 类对象的加法和赋值操作符用重载操作符来实现。被重载的操作符是一个带有特殊名字的函数。在加法操作符的例子中，函数名是 operator+。让我们为这个重载操作符提供一个定义：

```
Matrix              // 加法返回一个 Matrix 对象
operator+(          // 重载操作符的名字
    Matrix m1,      // 操作符左操作数的类型
    Matrix m2       // 操作符右操作数的类型
)
{
    Matrix result;
    // do the computation in result
    return result;
}
```

该实现支持两个 Matrix 对象的加法，如：

```
a + b
```

但不幸的是，它的效率低得让人难以接受。注意，operator+()的参数不是引用，这意味着：operator+()的实参是按值传递的。两个 Matrix 对象 a 和 b 的内容被拷贝到 opertor+()函数的参数区中。因为 Matrix 类对象非常大，分配这样一个对象并把它拷贝到函数参数区中的时间和空间开销高得让人难以接受。

为了提高我们的操作符函数的效率，假定我们决定把参数声明为指针。下面是对 operator+()新的实现代码。

```
// 使用指针参数重新实现
Matrix operator+( Matrix *m1, Matrix *m2 )
{
    Matrix result;

    // 在 result 中计算
    return result;
}
```

但是，这个实现代码有这样的问题；虽然我们获得了效率，但是，它是以放弃加法操作符用法的直观性为代价的。现在指针参数要求我们传递地址作为实参，它们指向我们希望做加法操作的 Matrix 对象。现在，我们的加法操作必须如下编程：

```
&a + &b; // 不太好，但也不是不可能
```

但是，这比较难看，而且可能引起一些程序员抱怨用户接口不友好，在一个复合表达式中加三个对象变得很困难：

```
// 喔！这无法工作
// &a + &b 的返回类型是 Matrix 对象
&a + &b + &c;
```

为了使在指针方案下三个对象的加法能够很好地实现，程序必须这样写：

```
// ok: 这样能行，但是
&( &a + &b ) + &c;
```

当然，没有人希望那样写。引用参数提供了我们需要的方案。当参数是引用时，函数接收到的是实参的左值而不是值的拷贝。因为函数知道实参在内存的什么位置，所以实参值没有被拷贝到函数的参数区。引用参数的实参是 Matrix 对象本身，这允许我们像对内置数据类型的对象使用加法操作符一样自然地使用加法操作符。

下面是 Matrix 类的重载加法操作符的修订版本：

```
// 使用引用参数的新实现
Matrix operator+( const Matrix &m1, const Matrix &m2 )
{
    Matrix result;
    // 在 result 中进行计算
    return result;
}
```

该实现支持如下形式的 Matrix 对象的加法：

```
a + b + c
```

为了支持类（class）类型——尤其是支持有效直观地实现重载操作符机制，C++特别引入了引用机制。

7.3.3 数组参数

在 C++中，数组永远不会按值传递。它是传递第一个元素（准确地说是第 0 个）的指针。例如，如下声明：

```
void putValues( int[ 10 ] );
```

被编译器视为；

```
void putValues( int* );
```

数组的长度与参数声明无关。因此，下列三个声明是等价的：

```
// 三个等价的 putValues()声明
void putValues( int* );
void putValues( int[] );
void putValues( int[ 10 ] );
```

因为数组被传递为指针，所以这对程序员有两个含义：

- 在被调函数内对参数数组的改变将被应用到数组实参上而不是本地拷贝上。当用作实参的数组必须保持不变时，程序员需要保留原始数组的拷贝。函数可以通过把参数类型声明为 const 来表明不希望改变数组元素：

```
void putValues( const int[ 10 ] );
```

- 数组长度不是参数类型的一部分。函数不知道传递给它的数组的实际长度，编译器也不知道。当编译器对实参类型进行参数类型检查时，并不检查数组的长度。例如：

```
void putValues( int[ 10 ] ); // 视为 int*

int main() {
    int i, j[ 2 ];
    putValues( &i );  // ok: &i 是 int*; 潜在的运行错误
    putValues( j ); // ok: j 被转换成第 0 个元素的指针
    // 实参类型为 int*: 潜在的运行错误
    return 0;
}
```

参数的类型检查只能保证 putValues()的两次调用都提供了 int*型的实参。类型检查不能检验实参是一个 10 元素的数组。

习惯上，C 风格字符串是字符的数组，它用一个空字符编码作为结尾。但是所有其他类型，包括希望处理内含空字符的字符数组，必须以某种方式在向函数传递实参时使其知道它的长度。一种常见的机制是提供一个含有数组长度的额外参数。例如：

```
void putValues( int[], int size );
int main() {
    int i, j[ 2 ];
    putValues( &i, 1 );
    putValues( j, 2 );
    return 0;
}
```

putValues()以下列格式输出数组的值：

```
( 10 ) < 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 >
```

这里 10 代表数组的长度。下面的实现代码用一个额外的参数表示数组的长度：

```
#include <iostream>
const lineLength = 12; // 一行中的元素数
void putValues( int *ia, int sz )
{
    cout << "( " << sz << " )< ";
    for ( int i = 0; i < sz; ++i )
    {
        if ( i % lineLength == 0 && i )
            cout << "\n\t"; // 一行满了
        cout << ia[ i ];

        // 用逗号分隔元素
        if ( i % lineLength != lineLength-1 &&
                    i != sz-1 )
            cout << ", ";
    }
    cout << " >\n";
}
```

另外一种机制是将参数声明为数组的引用。当参数是一个数组类型的引用时，数组长度成为参数和实参类型的一部分，编译器检查数组实参的长度与在函数参数类型中指定的长度是否匹配。

```
// 参数为 10 个 int 的数组
// parameter is a reference to an array of 10 ints
void putValues( int (&arr)[10] );
int main() {
    int i, j[ 2 ];
    putValues( i ); // 错误：实参不是 10 个 int 的数组
    putValues( j ); // 错误：实参不是 10 个 int 的数组
    return 0;
}
```

因为数组的长度现在是参数类型的一部分，所以 putValues()的这个版本只接受 10 个 int 的数组。这限制了可以作为实参被传递给 putValues()的数组的种类。但是，它也使函数的实现更加简单：

```
#include <iostream>
void putValues( int (&ia)[10] )
{
    cout << "( 10 )< ";
    for ( int i = 0; i < 10; ++i ) {
        cout << ia[ i ];
        // 用逗号分隔元素
        if ( i != 9 )
            cout << ", ";
    }
```

```
        cout << " >\n";
}
```

还有另外一种机制是使用抽象容器类型（抽象容器类型由第 6 章引入）。这种机制将在下一小节进一步介绍。

虽然前两个 putValues()的实现版本也能生效，但是它们有许多严重的限制。第一个实现只在 int 型数组时奏效。我们需要有第二个函数处理 double 型数组、第三个处理 long 型，等等。第二个实现只在数组是 10 个 int 型元素的数组时才能工作。处理不同类型的数组需要另外的函数。putValues()的一个较好的实现是把它定义为函数模板。函数模板是一种“其代码在广泛的不同参数类型上保持不变”的机制。下面给出怎样重写 putValues()的第一个实现，使其作为函数模板处理不同类型和长度的数组：

```
template <class Type>
void putValues( Type *ia, int sz )
{
    // 同前
}
```

模板参数被放在一对尖括号中。在本例中，惟一的一个模板参数是 Type。关键字 class 表示模板参数代表一个类型。标识符 Type 用作参数名，在 putValues()的参数表中出现的 Type 用作实例化函数模板的实际类型的占位符。在每次实例化中，实例化的实际类型——int、double、string 等等——替代 Type 参数。我们将在第 10 章进一步介绍函数模板。

参数也可以是多维数组，这样的参数必须指明第一维以外的所有维的长度。例如

```
void putValues( int matrix[][1a], int rowSize );
```

把 matrix 声明成一个二维数组，每行由 10 个列元素构成。matrix 可以被等价地声明为

```
int (*matrix)[10]
```

多雏数组被传递为指向其第 0 个元素的指针。在我们的例子中，matrix 的类型是指向 10 个 int 的数组的指针。如同只有一维的数组参数一样。多维数组的第一维与参数类型无关。多维数组的参数类型检查只检验多维数组实参中除了第一维之外的所有维的长度与参数的是否相同。

注意，*matrix 周围的括号是必需的，因为下标操作符的优先级较高。下列声明：

```
int *matrix[ 10 ];
```

将 matrix 声明成一个含有 10 个指向 int 的指针的数组。

7.3.4 抽象容器类型参数

第 6 章介绍的抽象容器类型也可以被用来声明函数参数。例如，我们可以用 vector<int>型的参数代替内置数组类型定义 putValues()。

容器类型实际上是类类型，它比内置数组数据类型提供了更多的功能。例如，vector<int>型的参数知道它包含的元素的个数。在上一小节，我们已经知道，如果函数有一个数组参数，且函数并不知道数组的第一维，我们有必要定义另外一个参数来告诉函数数组的长度。而使用 vector<int>型的参数使我们避开了这个约束。例如，我们可以按如下方式修改函数 putValues()的定义：

```
#include <iostream>
#include <vector>

const lineLength = 12; // 一行中的元素数
void putValues( vector<int> vec )
{
     cout << "( " << vec.size() << " )< ";
     for ( int i = 0; i < vec.size(); ++i ) {
          if ( i % lineLength == 0 && i )
               cout << "\n\t"; // 一行满了
          cout << vec[ i ];

          // 用逗号分隔元素
          if ( i % lineLength != lineLength-1 &&
                       i != vec.size()-1 )
               cout << ", ";
     }
     cout << " >\n";
}
```

main()函数调用新函数 putValues()如下：

```
void putValues( vector<int> );
int main() {
     int i, j[ 2 ];

     // 给 i 和 j 赋值
     vector<int> vec1(1);      // 创建一个单元素的 vector
     vec1[0] = i;
     putValues( vec1 );
     vector<int> vec2;         // 创建一个空的 vector

     // 在 vec2 中添加元素
     for ( int ix = 0;
                  ix < sizeof( j ) / sizeof( j[0] );
                  ++ix )
          // vec2[ix] == j[ix]
          vec2.push_back( j[ix] );
     putValues( vec2 );
     return 0;
}
```

我们注意到，putValues()的参数是值参（即按值传递的参数）。当容器类型的参数按值传递时，容器以及全部元素都被拷贝到被调函数的本地拷贝中。因为拷贝的效率非常低，所以把容器类型的参数声明为引用参数比较好。你会怎样改变 putValues()的参数声明呢？

记住，当一个函数不会修改参数的值时，我们把参数声明为 const 类型的引用更为合适。因而 putValues()的引用参数应该被声明如下：

```
void putValues( const vector<int> & ) { ...
```

7.3.5 缺省实参

缺省实参是一种虽然并不普遍、但在多数情况下仍然适用的实参值。缺省实参使程序员从函数接口的每个小细节中解脱出来。

函数可以用参数表中的初始化语法为一个或多个参数指定缺省实参。例如，假设一个函数创建并初始化一个二维字符数组以便模拟终端显示器，则我们可以为屏幕的高、宽和背景字符提供缺省实参：

```
char *screenInit( int height = 24, int width = 80,
                                   char background = ' ' );
```

调用包含缺省实参的函数时，我们可以（也可以不）为该参数提供实参。如果提供了实参，则它将覆盖缺省的实参值。否则，函数将使用缺省实参值。下面的 screenInit()调用都是正确的：

```
char *cursor;

// 等价于 screenInit(24,80, ' ')
cursor = screenInit();

// 等价于 screenInit(66, 88, ' ')
cursor = screenInit(66);

// 等价于 screenInit(66, 256, ' ')
cursor = screenInit(66, 256);
cursor = screenInit(66, 256, '#');
```

函数调用的实参按位置解析，缺省实参只能用来替换函数调用缺少的**尾部**（tailing）实参。例如，我们不可能为 background 提供字符值作为实参而不为 height 和 width 提供实参。

```
// 等价于 screenInit('<<', 80, ' ')
cursor = screenInit('<<");

// 错误，不等价于 screenInit(24,80,'<<')
cursor = screenInit( , , '<<');
```

设计带有缺省实参函数的部分工作就是排列参数表中的参数，使最可能取用户指定值的参数先出现，而最可能使用缺省实参的参数出现在后面。在 screenInit()的设计假设中（可能是通过经验得出），height 最可能由用户来提供。

函数声明可以为全部或部分参数指定缺省实参。在左边参数的任何缺省实参被提供之前，最右边未初始化参数必须被提供缺省实参。这是由于函数调用的实参是按照位置来解析的。

```
// 错误：在指定 height 之前，width 必须有一个缺省实参
char *screenInit( int height = 24, int width,
                              char background = ' ' );
```

一个参数只能在一个文件中被指定一次缺省实参。例如，下列语句是错误的：

```
// ff.h
```

```
int ff( int = 0 );

// ff.C
#include "ff.h"
int ff( int i = 0 ) { ... } // error
```

习惯上，缺省实参在公共头文件包含的函数声明中指定，而不是在函数定义中。如果缺省实参在函数定义的参数表中提供，则缺省实参只能用在包含该函数定义的文本文件的函数调用中。

函数后继的声明中可以指定其他缺省实参——一种对特定应用定制通用函数的有用方法。UNIX 系统函数 chmod()改变文件的保护级别，它的函数声明在系统头文件<cstdlib>中。声明如下：

```
int chmod( char *filePath, int protMode );
```

protMode 表示文件保护模式，filePath 表示文件名字和路径位置。如果一个特殊的应用总是将文件的保护模式改变成只读模式（read-only），那么我们不用每次都指明它，可以重新声明 chmod()缺省地提供该值：

```
#include <cstdlib>
int chmod( char *filepath, int protMode=0444 );
```

已知下列在头文件 ff.h 中声明的函数声明：

```
int ff( int a, int b, int c = 0 ); // ff.h
```

怎样重新声明 ff()，来把缺省实参提供给 b？下列语句是错误的，因为它重新指定了 c 的缺省实参：

```
#include "ff.h"
int ff( int a, int b = 0, int c = 0 ); // 错误
```

下列看起来错误的重新声明实际上是正确的：

```
#include "ff.h"
int ff( int a, int b = 0, int c ); // ok
```

在 ff()的重新声明中，b 是没有缺省实参的最右边参数。因此，缺省实参必须从最右边位置开始赋值的规则没有被打破。实际上，我们可以再次声明 ff()为：

```
#include "ff.h"

int ff( int a, int b = 0, int c ); // ok
int ff( int a = 0, int b, int c ); // ok
```

缺省实参不一定必须是常量表达式，可以使用任意表达式。例如：

```
int aDefault();
int bDefault( int );
int cDefault( double = 7.8 );

int glob;

int ff( int a = aDefault(),
        int b = bDefault( glob ),
```

```
                int c = cDefault() );
```

当缺省实参是一个表达式时，在函数被调用时该表达式被求值。例如，每次不带第三个实参调用 ff()时。编译器都会调用 cDefault()为 c 获取一个值。

7.3.6 省略号（ellipsis）

有时候我们无法列出传递给函数的所有实参的类型和数目。在这种情况下，我们可以用省略号（...）指定函数参数表。

省略号挂起类型检查机制。它们的出现告知编译器，当函数被调用时，可以有 0 个或多个实参，而实参的类型未知。省略号有下列两种形式：

```
void foo( parm_list, ... );
void foo( ... );
```

第一种形式为特定数目的函数参数提供了声明。在这种情况下，当函数被调用时，对于与显式声明的参数相对应的实参进行类型检查，而对于与省略号对应的实参则挂起类型检查。在第一种形式中．参数声明后面的逗号是可选的。

标准 C 库输出函数 printf()就是一个必须使用省略号的例子，printf()的第一个参数总是 C 风格字符串。

```
int printf( const char* ... );
```

这要求 printf()的每次调用都必须传递第一个 const char*型的实参。在 printf()的调用中，字符串后面是否有其他实参由第一个被称作格式字符串的实参所决定。在格式字符串中，由%开头的元字符表示其他实参的存在。例如，如下调用：

```
printf( "hello, world\n" );
```

有一个字符串实参。但是，如下调用：

```
printf( "hello, %s\n", userName );
```

有两个实参。%表明需要第二个实参，s 表明该实参的类型是一个字符串。

大多数带有省略号的函数都利用显式声明的参数中的一些信息，来获取函数调用中提供的其他可选实参的类型和数目。同此带有省略号的第一种形式的函数声明最常使用。

注意下列两个声明并不等价：

```
void f();

void f( ... );
```

在第一个实例中，f()被声明为不接受任何参数的函数。在第二个中，f()被声明为一个要求 0 个或多个实参的函数。如下的调用：

```
f( someValue );

f( cnt, a, b, c );
```

只对第二个声明是合法的。而如下调用：

```
f();
```

可用来调用第一个或第二个函数。

练习 7.4

下列声明哪些是错误的？为什么？

```
(a) void print( int arr[][], int size );
(b) int ff( int a, int b = 0, int c = 0 );
(c) void operate( int *matrix[] );
(d) char *screenInit( int height = 24, int width,
                      char background );
(e) void putValues( int (&ia)[] );
```

练习 7.5

下列这些函数的重新声明都是错误的，为什么？

```
(a)  char *screenInit( int height, int width,
                       char background = ' ' );
     char *screenInit( int height = 24, int width,
                       char background );
(b)  void print( int (*arr)[6], int size );
     void print( int (*arr)[5], int size );
(c)  void manip( int *pi, int first, int end = 0 );
     void manip( int *pi, int first = 0, int end = 0 );
```

练习 7.6

已知下列函数声明，下列哪些函数调用是错误的？为什么？

```
// 声明
void print( int arr[][5], int size );
void operate(int *matrix[7]);
char *screenInit( int height = 24, int width = 80,
                  char background = ' ' );

(a)   screenInit();    // 函数调用
(b)   int *matrix[5];
      operator( matrix ); // 函数调用
(c)   int arr[5][5];
      print( arr, 5 );  // 函数调用
```

练习 7.7

对于 7.3.4 小节中针对 vector<int>给出的 putValues()函数，请重写此函数，使它能处理 list<string>。每行输出一个串，使得包含两个串的 list 输出如下：

```
( 2 )
<
"first string"
"second string"
>
```

写一个 main()函数，它调用新的 putValues()函数，使用包含下列值的字符串列表：

```
"put function declarations in header files"
"use abstract container types instead of built-in arrays"
"declare class parameters as references"
"use reference to const types for invariant parameters"
"use less than eight parameters"
```

练习 7.8

什么时候该用指针参数？什么时候该用引用参数？解释它们各自的长处与缺点。

7.4 返回一个值

return 语句被放在函数体内，这条语句结束当前正在执行的函数。在程序执行期间，遇到 return 语句时，程序控制权被返回给调用此函数的函数。return 语句有两种形式：

return;

```
return expression;
```

第一种形式用在返回类型为 void 的函数中。在返回类型为 void 的函数中，返回语句不是必需的。它主要的作用是引起函数的强制结束。（return 语句的这种用法与 loop 循环中的 break 语句类似。break 语句在 5.8 节中介绍。）隐式的 return 发生在函数的最终语句完成时。例如：

```
void d_copy( double *src, double *dst, int sz )
{
    /* 将 "src" 复制到 "dst"
    * 简化假设：数组大小相同
    */

    // 如果指针为 0，返回
    if ( !src || !dst )
        return;

    // 如果两个参数引用同一个数组，返回
    if ( src == dst )
        return;

    // 没有东西要拷贝
    if ( sz == 0 )
        return;

    // 还在这儿？那么该做一些工作了
    for ( int ix = 0; ix < sz; ++ix )
        dst[ix] = src[ix];

    // 无需显式的 return，自动返回到调用函数
}
```

return 语句的第二种形式提供了函数的结果。结果可以是任意复杂的表达式，同时也叫

以包含函数调用。例如，factorial()的实现包含下列 return 语句［我们将在下一节中看到 factorial()的实现］：

```
return val * factorial(val-1);
```

一个**具有返回值**（value_returning）的函数（即，函数返回类型没有被声明为 void）必须返回一个值，缺少返回值将引起编译错误。虽然 C++不能保证一个结果的正确性，但它至少可以保证为每个具有返回值的函数提供一个结果。例如，下列函数编译将失败，因为它的两个出口点（exit point）都没有返回值。

```
// definition of the Matrix class interface
#include "Matrix.h"
bool is_equal( const Matrix &m1, const Matrix &m2 )
{
    /* 如果两个 Matrix 对象相同
    * 则返回 true;
    * 否则返回 false
    */
    // 比较列数
    if ( m1.colSize() != m2.colSize() )
        // 程序错误： 没有返回值
        return;

    // 比较行数
    if ( m1.rowSize() != m2.rowSize() )
        // 程序错误： 没有返回值
        return;

    // 遍历每个 matrix 直到不相等
    // 或所有元素都检查完毕
    for ( int row = 0; row < m1.rowSize(); ++row )
        for ( int col = 0; col < m1.colSize(); ++col )
            if ( m1[row][col] != m2[row][col] )
                return false;

    // 程序错误： 没有返回值
    // 此时 m1 == m2
}
```

如果被返回的值的类型与函数返回类型不匹配，那么如果可能的话，将应用隐式类型转换。如果无法隐式转换，则产生一个编译错误（类型转换在 4.14 节讨论）。

缺省情况下，函数的返回值是**按值传递的**（passed by value），这意味着得到控制权的函数将接收返回语句中指定的表达式的拷贝。例如：

```
Matrix grow( Matrix* p ) {
    Matrix val;

    // ...
    return val;
}
```

grow()把存储在 val 中的值的拷贝返回到调用函数，但调用函数不能用任何方式修改 val。

该缺省行为可以被改变。一个函数可以被声明为返回一个指针或一个引用。当函数返回一个引用时，调用函数接收 val 的左值，即，调用函数可以修改 val 或取它的地址。grow() 可以如下声明返回一个引用：

```
Matrix& grow( Matrix* p ) {
    Matrix *res;

    // 在动态存储中分配一个更大的 Matrix
    // res 是指向新 Matrix 的指针
    // 将*p 内容复制到*res
    return *res;
}
```

如果返回值是一个大型类对象，用引用（或指针）返回类型比按值返回类对象效率要高得多。在某些情况下，编译器自动将按值返回转换到按引用返回。该优化被称为**命名返回值优化**（named return value optimization），将在 14.8 节中描述。

当声明一个返回引用的函数时，程序员应当知道下面两个易犯的错误：

1．返回一个指向局部对象的引用。局部对象的生命期随函数的结束而结束（局部对象的生命期将在 8.3 节讨论）。在函数结束后，该引用变成未定义内存的别名。例如：

```
// 问题：返回一个指向局部对象的引用
Matrix& add( Matrix &m1, Matrix &m2 )
{
    Matrix result;

    if ( m1.isZero() )
        return m2;
    if ( m2.isZero() )
        return m1;

    // 将两个 Matrix 对象的内容相加
    // 喔！返回之后，结果指向一个有问题的位置
    return result;
}
```

在这种情况下，返回类型应该被声明为非引用类型。然后再在局部对象的生命期结束之前，拷贝局部变量：

```
Matrix add( ...
```

2．函数返回一个左值。对返回值的任何修改都将改变被返回的实际对象。例如：

```
#include <vector>

int &get_val( vector<int> &vi, int ix ) {
    return vi[ix];
}

int ai[4] = { 0, 1, 2, 3 };

vector<int> vec( ai, ai+4 ); // 将 ai 的 4 个元素复制到 vec
```

```
int main() {
    // 将 vec[0] 增加到 1
    get_val( vec,0 )++;

    // ...
}
```

为防止对引用返回值的无意修改，返回值应该被声明为 const：

```
const int &get_val( ...
```

在 2.3 节里，对于类 IntArray 重载下标操作符的讨论中，我们曾经给出过一个为了修改被返回的实际对象而返回左值的例子。

7.4.1 参数和返回值与全局对象

一个程序中的各种函数可以通过两种机制进行通信。[这里的**通信**（communicate）指的是值的交换。]一种方法是使用全局对象，第一种方法是使用函数参数表和返回值。

全局对象（global object）被定义在函数定义之外。例如：

```
int glob;
int main() {
    // 函数体任意
}
```

对象 glob 是一个全局对象（在第 8 章将进一步讨论全局域和全局对象）。在程序的任意地方都可以访问全局对象是它的主要优势，也是它最大的负担。全局对象的可视性使其成为程序各部分之间进行通信的一种方便的机制，但函数之间依靠全局对象的通信有下列缺点：

- 使用全局对象的函数依赖于全局对象的存在和类型，这使得在不同上下文环境中重用该函数更加困难。
- 如果程序必须被修改，则全局依赖增加了引入错误的可能性。而且，既使只对局部做修改也要求程序员必须理解整个程序。
- 如果全局对象得到一个不正确的值，则必须查找整个程序以判断错误发生的位置，也就是没有实现局部化。
- 当一个函数使用全局对象时，递归更加难以正确完成。（递归在程序调用自身时才发生。我们将在 7.5 节介绍递归。）
- 在线程存在的情况下，我们必须做特殊的编码，以便同步各个线程对于全局对象的读和写操作。当我们在使用线程时，缺少同步是程序错误的常见根源。（关于 C++ 中的线程程序设计，见 Steve Vinoski 和 Doug Sohmidt 在［LIPPMAN96b］中的文章“C++中的分布式对象计算”。）

因此，建议程序中的函数使用参数表和返回值进行通信。

向一个函数传递参数发生错误的可能性随参数表的长度的增加而提高。作为一个通用规则。8 个参数应该是最大值了。为了替换一个大型的参数表，程序员可以将参数声明为类。数组或某一种容器类型。这样的参数可以用来包含一组参数值。

类似的情况，一个函数只能返回一个值。如果程序的逻辑要求返回多个值，那么程序员可以将某些函数参数声明为引用，在这种情况下，函数可以直接修改相应的实参，因而程序

员可以设置这些实参含有某些额外的“返回”值；或者程序员可以声明一个函数，它的返回类型是一个可以包含一组返回值的类或某一种容器类型。

练习 7.9

return 语句的两种形式是什么？使用每一种形式的条件是什么？

练习 7.10

你知道下列函数定义有什么潜在的运行问题吗？

```
vector<string> &readText( ) {
    vector<string> text;
    string word;

    while ( cin >> word ) {
        text.push_back( word );
        // ...
    }

    // ....
    return text;
}
```

练习 7.11

怎样从一个函数返回一个以上的值？描述你所选方法的优缺点。

7.5 递归

直接或间接调用自己的函数被称为**递归函数**（recursive function）。下面的函数 rgcd()就是一个速归函数：

```
int rgcd( int v1, int v2 )
{
    if ( v2 != 0 )
        return rgcd( v2, v1%v2 );

    return v1;
}
```

递归函数必须定义一个**停止条件**（stopping condition）。否则，函数会“永远”递归下去。有时候，这被称作**无限递归**（infinit recursion）错误。在 rgcd()的情况下，停止条件是余数为 0。

如下调用：

```
rgcd( 15, 123 );
```

计算结果为 3。表 7.1 跟踪它的执行情况。

表格 7.1 rgcd(15, 123) 的跟踪结果

v1	v2	返回值
15	123	rgcd(123,15)
123	15	rgcd(15,3)
15	3	rgcd(3,0)
3	0	3

最后一个调用：

```
rgcd(3, 0);
```

满足了停止条件，它返回最大公约数 3。该值依次成为前面每个调用的返回值，这个值被称为回渗（percolate），直到执行返回到第一次调用 rgcd()的函数。

由于与函数调用相关的额外开销，递归函数可能比非递归（或称迭代）函数执行得慢一些。但是，递归函数可能更小且更易于理解。N 的阶乘（factorial）是将数从 1 乘到 n 的结果。例如，5 的阶乘是 120。

```
1 × 2 × 3 × 4 × 5 = 120
```

阶乘的计算可以用递归函数实现：

```
unsigned long
factorial( int val ) {
    if ( val > 1 )
        return val * factorial( val-1 );

    return 1;
}
```

本例的结束条件发生在 val 的值为 1 时。

练习 7.12

将 factorial()重写为送代函数。

练习 7.13

如果 factorial()的结束条件如下所示，将会发生什么？

```
if ( val != 0 )
```

7.6 inline 函数

考虑下列 min()函数：

```
int min( int v1, int v2 )
{
```

```
        return( v1 < v2 << v1 : v2 );
    }
```

为这样的小操作定义一个函数的好处是：

- 如果一段代码包含 min()的调用，那么阅读这样的代码并解释它的含义比读一个条件操作符的实例以及理解代码在做什么，尤其是复杂表达式时要容易得多。
- 改变一个局部化的实现比更改一个应用中的 300 个出现要容易得多。例如，如果决定测试条件应该是：

```
( v1 == v2 || v1 < v2 )
```

  那么，找到该代码的每一个出现将非常乏味而且容易出错。
- 语义是统一的，每个测试都保证以相同的方式实现。
- 函数可以被重用，不必为其他的应用重写代码。

但是，将 min()写成函数有一个严重的缺点：调用函数比直接计算条件操作符要慢得多。不但必须拷贝两个实参，保存机器的寄存器。程序还必须转向一个新位置。同此，手写的条件操作符能快得多。

inline（内联）函数给出了一种解决方案。若一个函数被指定为 inline 函数，则它将在程序中每个调用点上被“内联地”展开。例如：

```
int minVal2 = min( i, j );
```

在编译时被展开为：

```
int minVal2 = i < j << i : j;
```

把 min()写成函数的额外执行开销从而被消除了。

在函数声明或定义中的函数返回类型前加上关键字 inline，即把 min()指定成为 inline：

```
inline int min( int v1, int v2 ) { /* ... */ }
```

但是，注意 inline 指示对编译器来说只是一个建议。编译器可以选择忽略该建议，因为把一个函数声明为 inline 函数，并不见得真的适合在调用点上展开。例如，一个递归函数，如 rgcd()并不能在调用点完全展开（虽然它的第一个调用可以）。一个 1200 行的函数也不太可能在调用点展开。一般地，inline 机制用来优化小的、只有几行的、经常被调用的函数。在抽象数据类的设计中，它对支持信息隐藏起着主要作用，比如在 2.3 节中介绍的 IntArray 类的 size()inline 成员函数。

inline 函数对编译器而言必须是可见的，以便它能够在调用点内联展开该函数。与非 inline 函数不同的是，inline 函数必须在调用该函数的每个文本文件中定义。当然，对于同一程序的不同文件，如果 inline 函数出现的话，其定义必须相同。对于由两个文件 compute.C 和 draw.C 构成的程序来说，程序员不能定义这样的 min()函数，它在 compute.C 中指一件事情，而在 draw.C 中指另外一件事情。如果两个定义不相同，程序将会有未定义的行为：编译器最终会使用这些不同定义中的哪一个作为非 inline 函数调用的定义是不确定的，因而程序的行为可能并不像你所期望的。

为保证不会发生这样的事情，建议把 inline 函数的定义放到头文件中。在每个调用该 inline 函数的文件中包含该头文件。这种方法保证对每个 inline 函数只有一个定义，且程序员无需

复制代码，并且不可能在程序的生命期中引起无意的不匹配的事情。

因为 min()是一个常见操作，所以 C++标准库提供了 min()的一个实现。min()操作是第 12 章介绍的泛型算法的一部分，它的用法将在附录中说明。标准库把 min()定义为模板，允许它应用在非 int 型的算术型操作数上。函数模板将在第 10 章中讨论。

7.7 链接指示符：extern “C”

如果程序员希望调用其他程序设计语言（尤其是 C）写的函数，那么。调用函数时必须告诉编译器使用不同的要求。例如，当这样的函数被调用时，函数名或参数排列的顺序可能不同，无论是 C++函数调用它，还是用其他语言写的函数调用它。

程序员用**链接指示符**（linkage directive）告诉编译器，该函数是用其他的程序设计语言编写的。链接指示符有两种形式。既可以是**单一语句**（single statement）形式，也可以是**复合语句**（compound statement）形式：

```
// 单一语句形式的链接指示符
extern "C" void exit(int);

// 复合语句形式的链接指示符
extern "C" {
     int printf( const char* ... );
     int scanf( const char* ... );
}
// 复合语句形式的链接指示符
extern "C" {
#include <cmath>
}
```

链接指示符的第一种形式由关键字 extern 后跟一个字符串常量以及一个“普通”的函数声明构成。虽然函数是用另外一种语言编写的，但调用它仍然需要类型检查。例如，编译器会检查传递给函数 exit()的实参的类型是否是 int，或者能够隐式地转换成 int 型。

多个函数声明可以用花括号包含在链接指示符复合语句中，这是链接指示符的第二种形式。花招号被用作分割符，表示链接指示符应用在哪些声明上。在其他意义上该花括号被忽略，所以在花括号中声明的函数名对外是可见的，就好像函数是在复合语句外声明的一样。例如，在前面的例子中，复合语句 extern "C"表示函数 printf()和 scanf()是在 C 语言中写的函数。因此，这个声明的意义就如同 printf()和 scanf()是在 extern "C"复合语句外面声明的一样。

当复合语句链接指示符的括号中含有#include 时，在头文件中的函数声明都被假定是用链接指示符的程序设计语言所写的。在前面的例子中，在头文件<cmath>中声明的函数都是 C 函数。

链接指示符不能出现在函数体中。下列代码段将会导致编译错误：

```
int main()
{
     // 错误：链接指示符不能出现在函数内
     extern "C" double sqrt( double );
```

```
        double getValue();  //ok
        double result = sqrt ( getValue() );
        //...

        return 0;
    }
```

如果把链接指示符移到函数体外，程序编译将无错误：

```
extern "C" double sqrt( double );
int main()
{
    double getValue();  //ok
    double result = sqrt ( getValue() );
    //...
    return 0;
}
```

但是，把链接指示符放在头文件中更合适。在那里，函数声明描述了函数的接口所属。

如果我们希望 C++函数能够为 C 程序所用，又该怎么办呢？我们也可以使用 extern "C" 链接指示符来使 C++函数为 C 程序可用。例如：

```
// 函数 calc() 可以被 C 程序调用
extern "C" double calc( double dparm ) { /* ... */ }
```

如果一个函数在同一文件中不只被声明一次，则链接指示符可以出现在每个声明中。它也可以只出现在函数的第一次声明中，在这种情况下，第二个及以后的声明都接受第一个声明中链接指示符指定的链接规则。例如：

```
// ---- myMath.h ----
extern "C" double calc( double );

// ---- myMath.C ----
// 在 Math.h 中的 calc() 的声明
#include "myMath.h"

// 定义了 extern "C" calc() 函数
// calc() 可以从 C 程序中被调用
double calc( double dparm ) { // ...
```

在本节中，我们只看到为 C 语言提供的链接指示：extern "C"。extern "C"是惟一被保证由所有 C++实现都支持的。每个编译器实现都可以为其环境下常用的语言提供其他链接指示。例如 extern "Ada"可以用来声明是用 Ada 语言写的函数，extern "FORTRAN"用来声明是用 FORTRAN 语言写的函数，等等。因为其他的链接指示随着具体实现的不同而不同，所以建议读者查看编译器的用户指南，以获得其他链接指示符的进一步信息。

本节介绍了 C++中的关键字 extern 的第一种用法。在 8.2 节中，我们将看到 extern 的其他有关对象和函数声明的用法。

练习 7.14

exit()、printf()、malloc()、strcpy()以及 strlen()都是 C 语言库例程，修改下列 C 程序使其

能在 C++下编译链接。

```
const char *str = "hello";

void *malloc( int );
char *strcpy( char *, const char * );
int printf( const char *, ... );
int exit( int );
int strlen( const char * );

int main()
{ /* C 语言程序 */
     char* s = malloc( strlen(str)+1 );
     strcpy( s, str );
     printf( "%s, world\n", s );
     exit( 0 );
}
```

7.8 main()：处理命令行选项

通常，在执行程序时，我们会传递命令行选项。例如，我们可能写如下命令行：

```
prog -d -o ofile data0
```

实际上，命令行选项是 main()的实参。在 main()函数中，我们可以通过一个名为 argv 的 C 风格字符串数组访问它。在本节中，我们将说明怎样支持命令行选项。

在本节之前，所有的 main()函数都声明了一个空的参数表：

```
int main() { ... }
```

如果用户已在命令行中指定了选项的话，那么我们可以通过 main()函数的一种扩展原型特征来访问这些选项：

```
int main( int argc, char *argv[] ) { ... }
```

argc 包含命令行选项的个数，argv 包含 aygc 个 C 风格字符串，代表了由空格分隔的命令选项。例如，对于如下命令行：

```
prog -d -o ofile data0
```

argc 被设置为 5，且 argv 被设置为下列 C 风格字符串：

```
argv[ 0 ] = "prog";
argv[ 1 ] = "-d";
argv[ 2 ] = "-o";
argv[ 3 ] = "ofile";
argv[ 4 ] = "data0";
```

argv[0]总是被设置为当前正被调用的命令。从索引 1 到 argc-1 表示被传递给命令的实际选项。

让我们来看一下如何取出在 argv 中的命令行选项。在我们的例子中，将支持下列用法：

```
program_name [-d] [-h] [-v]
```

```
        [-o output_file] [-l limit_value]
        file_name
        [ file_name [file_name [ ... ]]]
```

方括号中的内容是可选的。例如，最小的命令行只给出要处理的文件：

```
prog chap1.doc
```

其他可能的调用方式如下：

```
prog -l 1024 -o chap1-2.out chap1.doc chap2.doc
prog -d chap3.doc
prog -l 512 -d chap4.doc
```

处理命令行选项的基本步骤如下：

1．按顺序从 argv 中取出每个选项。我们将用 for 循环来完成，从索引 1 开始迭代（所以跳过程序名）：

```
for ( int ix = 1; ix < argc; ++ix ) {
    char *pchar = argv[ ix ];
    // ...
}
```

2．确定选项的类型。如果它以“-”开始的，我们就能知道它是{h,d,v,l,o}之一。否则，它或许是与-l 相关的实际限制量，或者是与-o 相关的输出文件名，或者是程序要处理的文件名。我们将用 switch 语句确定是否存在一个“-”：

```
switch ( pchar[ 0 ] ) {
    case '-': {
        // 识别 -h, -d, -v, -l, -o
    }
    default: {
        // 处理 -l 后的限制值
        // -o 后面的输出文件
        // 文件名 ...
    }
}
```

3．填写代码，处理第 2 项中的两个 case。

如果存在“-”，则我们只是简单地切换到下一个字符，确定用户指定的选项。

下面是实现代码的一般轮廓：

```
case '-': {
    switch( pchar[ 1 ] )
    {
    case 'd':
        // 处理调试
        break;

    case 'v':
        // 处理版本请求
        break;

    case 'h':
        // 处理帮助
```

```
            break;

        case 'o':
            // 准备处理输出文件
            break;

        case 'l':
            // 准备处理限制量
            break;

        default:
            // 无法辨识的选项
            // 报告并退出
        }
    }
```

选项-d 打开调试。为处理它，我们把一个对象：

```
bool debug_on = false;
```

设置为 true：

```
case 'd':
    debug_on = true;
    break;
```

我们的程序可能包含如下代码：

```
if ( debug_on )
    display_state_elements( obj );
```

选项-v 显示程序的版本号，然后结束：

```
case 'v':
    cout << program_name << "::"
        << program_version << endl;
    return 0;
```

选项-h 生成程序的 usage()消息。然后结束［结束在 usage()中完成］：

```
case 'h':
    // 无需 break: usage() 会退出
    usage();
```

选项-o 表明用户指定的输出文件名紧随其后。类似地，选项-l 表明后面是限制值（limit value）。该怎样处理它呢？

如果不存在“-”，我们知道应该有一个限制值，或者用户指定的输出文件，或者要被处理的文件的名字。为区分这三种可能，我们将记录内部状态的对象设置为 true：

```
// 如果为 true，则下一个实参是输出文件
bool ofile_on = false;

// 如果为 true，则下一个实参是限制值
bool limit_on = false;
```

在实现代码的选项处理部分里：

```
case 'l':
```

```
            limit_on = true;
            break;
        case 'o':
            ofile_on = true;
            break;
```

当我们遇到一个不以“-”开头的实参时，我们测试状态对象以便确定选项所表达的内容：

```
        // 三种可能: limit_value, output_file 或者 file_name
        default: {
            // 如果看到 -o, 则设置 ofile_on
            if ( ofile_on ) {
                // 处理 output_file
                // 关闭 ofile_on
            }
            else
                if ( limit_on ) { // 如果见到 -l
                    // 处理 limit_value
                    // 关闭 limit_on
                }
                else {
                    // 处理 file_name
                }
        }
```

如果选项是一个输出文件，我们将 ofile_on 复位为 false，并取出文件名：

```
        if ( ofile_on ) {
            ofile_on = false;
            ofile = pchar;
        }
```

如果实参是个限制值，我们需要把 C 风格的字符串转换成数值表示。我们用标准库函数 atoi()。为了使用 atoi()，我们将包含 ctype.h 头文件。此外，我们还必须保证限制值是个非负值，并必须将 limit_on 复位为 false：

```
        // int limit;
        else
            if ( limit_on ) {
                limit_on = false;
                limit = atoi( pchar );
                if ( limit < 0 ) {
                    cerr << program_name << "::"
                         << program_version << " : error: "
                         << "negative value for limit.\n\n";
                    usage( -2 );
                }
            }
```

否则，如果状态对象都不是 true，则认为是一个文件，将来可以打开来处理。我们将它的名字存储在一个字符串 vector 中：

```
        else
            file_names.push_back( string( pchar ));
```

当我们处理命令行时，可能最重要的设计部分在于选择处理无效选项的方式。例如，我

们认为指定一个负的限制值为一个严重的错误。这或许合适也或许不合适。或许我们可以认为它越界了，警告用户，并将其值设置为 0 或其他有意义的缺省值。

当用户弄乱了命令行的空格分隔符，我们的程序就会出现了两个明显的缺点。例如，下列两个命令行都不能处理：

```
prog - d data01
prog -oout_file data01
```

（我们把它们都留作本节最后的练习。）

下面是程序的完整实现（我们已经加入了输出语句来说明它的处理过程。）

```
#include <iostream>
#include <string>
#include <vector>
#include <ctype.h>

const char *const program_name = "comline";
const char *const program_version = "version 0.01 (08/07/97)";

inline void usage( int exit_value = 0 )
{
    // 输出一个格式化的用法信息
    // 并用 exit_value 退出...
    cerr << "usage:\n"
        << program_name << " "
        << "[-d] [-h] [-v] \n\t"
        << "[-o output_file] [-l limit] \n\t"
        << "file_name\n\t[file_name [file_name [ ... ]]]\n\n"
        << "where [] indicates optional option: \n\n\t"
        << "-h: help.\n\t\t"
        << "generates this message and exits\n\n\t"
        << "-v: version.\n\t\t"
        << "prints version information and exits\n\n\t"
        << "-d: debug.\n\t\tturns debugging on\n\n\t"
        << "-l limit\n\t\t"
        << "limit must be a non-negative integer\n\n\t"
        << "-o ofile\n\t\t"
        << "file within which to write out results\n\t\t"
        << "by default, results written to standard output \n\n"
        << "file_name\n\t\t"
        << "the name of the actual file to process\n\t\t"
        << "at least one file_name is required --\n\t\t"
        << "any number may be specified\n\n"
        << "examples:\n\t\t"
        << "$command chapter7.doc\n\t\t"
        << "$command -d -l 1024 -o test_7_8 "
        << "chapter7.doc chapter8.doc\n\n";
    exit( exit_value );
}

int main( int argc, char* argv[] )
```

```
{
    bool debug_on = false;
    bool ofile_on = false;
    bool limit_on = false;
    int limit = -1;
    string ofile;
    vector<string, allocator> file_names;
    cout << "illustration of handling command line arguments:\n"
        << "argc: " << argc << endl;
    for ( int ix = 1; ix < argc; ++ix )
    {
        cout << "argv[ " << ix << " ]: "
            << argv[ ix ] << endl;
        char *pchar = argv[ ix ];
        switch ( pchar[ 0 ] )
        {
        case '-':
            {
                cout << "case \'-\' found\n";
                switch( pchar[ 1 ] )
                {
                case 'd':
                    cout << "-d found: "
                      << "debugging turned on\n";
                    debug_on = true;
                    break;
                case 'v':
                    cout << "-v found: "
                      << "version info displayed\n";
                    cout << program_name
                      << " :: "
                      << program_version
                      << endl;
                      return 0;
                case 'h':
                    cout << "-h found: "
                      << "help information\n";

                    // 这里没必要用 break 了, usage() 可以退出
                    usage();
                case 'o':
                    cout << "-o found: output file\n";
                    ofile_on = true;
                    break;
                case 'l':
                    cout << "-l found: "
                      << "resource limit\n";
```

```
                    limit_on = true;
                    break;
                default:
                    cerr << program_name
                      << " : error : "
                      << "unrecognized option: - "
                      << pchar << "\n\n";

                    // 这里没必要用 break 了, usage() 可以退出
                    usage( -1 );
                }
                break;
            }
        default: // 或文件名
            cout << "default nonhyphen argument: "
                << pchar << endl;
            if ( ofile_on ) {
                ofile_on = false;
                ofile = pchar;
            }
            else
                if ( limit_on ) {
                    limit_on = false;
                    limit = atoi( pchar );
                    if ( limit < 0 ) {
                      cerr << program_name
                          << " : error : "
                          << "negative value for limit.\n\n";
                      usage( -2 );
                    }
                }
                else file_names.push_back( string( pchar ));
            break;
        }
    }
    if ( file_names.empty() ) {
        cerr << program_name
            << " : error : "
            << "no file specified for processing.\n\n";
        usage( -3 );
    }
    if ( limit != -1 )
        cout << "User-specifed limit: "
            << limit << endl;
    if ( ! ofile.empty() )
        cout << "User-specified output file: "
            << ofile << endl;
    cout << (file_names.size() == 1 ? "File " : "Files ")
```

```
            << "to be processed are the following:\n";
        for ( int inx = 0; inx < file_names.size(); ++inx )
            cout << "\t" << file_names[ inx ] << endl;
    }
```

下面是程序的执行练习：

```
a.out -d -l 1024 -o test_7_8 chapter7.doc chapter8.doc
```

下面是命令行选项处理的跟踪结果：

```
illustration of handling command line arguments:
argc: 8
argv[ 1 ]: -d
case '-' found
-d found: debugging turned on
argv[ 2 ]: -l
case '-' found
-l found: resource limit
argv[ 3 ]: 1024
default nonhyphen argument: 1024
argv[ 4 ]: -o
case '-' found
-o found: output file
argv[ 5 ]: test_7_8
default nonhyphen argument: test_7_8
argv[ 6 ]: chapter7.doc
default nonhyphen argument: chapter7.doc
argv[ 7 ]: chapter8.doc
default nonhyphen argument: chapter8.doc
User-specifed limit: 1024
User-specified output file: test_7_8
Files to be processed are the following:
    chapter7.doc
    chapter8.doc
```

7.8.1 一个处理命令行的类

我们最好是把处理命令行选项的细节封装起来，使得它不会扰乱 main()。一种封装策略当然是提供一个函数。例如：

```
extern int parse_options( int arg_count,
                          char **arg_vector );
int main( int argc, char *argv[] ) {
    // ...
    int option_status;
    option_status = parse_options( argc, argv );

    // ...
}
```

这个设计的问题在于，怎样返回由用户传递的值的集合。典型情况下，这些值被定义成全局对象，不需要在函数之间传来传去。另外一种方法是，我们可以把处理过程封装到一个类（class）中。

类的数据成员是代表用户可能设置的值的对象。一组公有 inline 函数提供了对这些值的访问途径。构造函数将这些值初始化为其缺省设置。一个成员函数取 argc 和 argv 做实参，并提供对于选项的处理：

```
#include <vector>
#include <string>

class CommandOpt {
public:
    CommandOpt() : _limit( -1 ), _debug_on( false ) {}
    int parse_options( int argc, char *argv[] );
    string out_file() { return _out_file; }
    bool debug_on() { return _debug_on; }
    int files() { return _file_names.size(); }

    // 访问 _file_names
    string& operator[]( int ix );
private:
    inline int usage( int exit_value = 0 );

    bool _debug_on;
    int _limit;
    string _out_file;
    vector<string, allocator> _file_names;

    static const char *const program_name;
    static const char *const program_version;
};
```

下面是修改后的 main()[19]：

```
#include "CommandOpt.h"

int main( int argc, char *argv[] ) {
    // ...
    CommandOpt com_opt;
    int option_status;
    option_status = com_opt.parse_options(argc,argv);
    // ...
}
```

练习 7.15

增加选项-t（打开计时器），以及选项-b（提供 bufsize 实参）的处理。来确保同时更改 usage()，例如：

```
prog -t -b 512 data0
```

[19] CommandOpt 类的完整实现可以在 Addison-Wesley Web 网站上找到。

练习 7.16

我们的实现不能处理在选项与其相关值之间没有空格的情况。理想情况下，我们可以接受有或没有空格的选项。请修改实现代码来做到这一点。

练习 7.17

我们的实现现在不能处理在“-”和选项之间增加空格的情况，如：

```
prog - d data0
```

修改我们的实现识别并显式地标记这个错误。

练习 7.18

我们的实现不能识别选项-l 和-o 的多个实例。修改实现来完成它。策略是什么？

练习 7.19

如果用户指定一个未知选项，我们的实现将产生一个致命错误。你认为这合理吗？我们还可以怎么做？

练习 7.20

为以加号（+）开头的选项增加支持，为选项+s、+pt、+sp 以及+ps 提供处理。假定+s 打开严格处理，而+p 支持现在已经过时的前构造过程（previous construct）。例如：

```
prog +s +p -d -b 1024 data0
```

7.9 指向函数的指针※

假定我们被要求提供一个如下形式的排序函数：

```
sort( start, end, compare );
```

start 和 end 是指向字符串数组中元素的指针。函数 sort()对于 start 和 end 之间的数组元素进行排序。compare 定义了比较数组中两个字符串的比较操作。

该怎样实现 compare 呢？我们或许想按字典顺序排序数组内的字符串——即，与字典中相同的方式排序单词：或许想按长度排序它们，以便将最短的字符串放在前面，而长的放在后面。指定可替换的比较操作需要某种设施。

（第 12 章将描述 sort()函数以及 C++标准库提供的其他泛型算法。在本节中，为说明函数指针的用法，我们自己写 sort()函数，它是 C++标准库提供的函数的简化版本。）

解决这些需求的一种策略是将第三个参数 compare 设为函数指针，并由它指定要使用的比较函数。

为简化 sort()的用法而又不限制它的灵活性，我们可能希望指定一个缺省的比较函数，以用于大多数的情况。让我们假设最常见的以字典序排列字符串的情况，缺省实参将指定一个

比较操作，它用到了字符串的 compare()函数（这个函数在 6.10 节首次介绍）。

本节我们将考虑怎样用函数指针来实现我们的 sort()函数。

7.9.1 指向函数的指针的类型

怎样声明指向函数的指针呢？用函数指针作为实参的参数会是什么样呢？下面是函数 lexicoCompare()的定义，它按字典序比较两个字符串：

```
#include <string>

int lexicoCompare( const string &s1, const string &s2 ) {
     return s1.compare(s2);
}
```

如果字符串 s1 和 s2 中的所有字符都相等，则 lexicoCompare()返回 0；否则，如果第一个参数表示的字符串小于第二个参数表示的字符串，则返回一个负数；如果大于，则返回一个正数。

函数名不是其类型的一部分，函数的类型只由它的返回值和参数表决定。指向 lexicoCompare()的指针必须指向与 lexicoCompare()相同类型的函数（带有相同的返回类型和相同的参数表）。让我们试一下：

```
int *pf( const string &, const string & );   // 喔！差一点
```

这几乎是正确的。问题是编译器把该语句解释成名为 pf 的函数的声明，它有两个参数，并且返回一个 int*型的指针。参数表是正确的，但是返回值不是我们所希望的。解引用操作符（*）应与返回类型关联，所以在这种情况下，是与类型名 int 关联，而不是 pf。要想让解引用操作符与 pf 关联，括号是必需的：

```
int (*pf)( const string &, const string & ); // ok: 正确
```

这个语句声明了 pf 是一个指向函数的指针，该函数有两个参数和 int 型的返回值。即指向函数的指针，它与 lexicoCompare()的类型相同。

下列函数与 lexicoCompare()类型相同，都可以用 pf 来指向：

```
int sizeCompare( const string &, const string & );
```

但是，calc()和 gcd()与前面两个函数的类型不同，不能用 Pf 来指：

```
int calc( int , int );
int gcd( int , int );
```

可以如下定义 pfi，它能够指向这两个函数：

```
int (*pfi)( int, int );
```

省略号是函数类型的一部分。如果两个函数具有相同的参数表，但是一个函数在参数表末尾有省略号，则它们被视为不同的函数类型。指向这两个函数的指针类型也不同：

```
int printf( const char*, ... );
int strlen( const char* );

int (*pfce)( const char*, ... );    // 可以指向 printf()
int (*pfc)( const char* );          // 可以指向 strlen()
```

函数返回类型和参数表的不同组合，代表了各不相同的函数类型。

7.9.2 初始化和赋值

我们知道，不带下标操作符的数组名会被解释成指向首元素的指针。当一个函数名没有被调用操作符修饰时，会被解释成指向该类型函数的指针。例如，表达式

```
lexicoCompare;
```

被解释成类型

```
int (*)( const string &, const string & );
```

的指针。

将取地址操作符作用在函数名上也能产生指向该函数类型的指针。因此，lexicoCompare和&lexioCompare 类型相同。指向函数的指针可如下被初始化：

```
int (*pfi)( const string &, const string & ) = lexicoCompare;
int (*pfi2)( const string &, const string & ) = &lexicoCompare;
```

指向函数的指针可以如下被赋值：

```
pfi = lexicoCompare;
pfi2 = pfi;
```

只有当赋值操作符左边指针的参数表和返回类型与右边函数或指针的参数表和返回类型完全匹配时，初始化和赋值才是正确的。如果不匹配，则将产生编译错误消息。在指向函数类型的指针之间不存在隐式类型转换。例如：

```
int calc( int, int );
int (*pfi2s)( const string &, const string & ) = 0;
int (*pfi2i)( int, int ) = 0;

int main() {
    pfi2i = calc;     // ok
    pfi2s = calc;     // 错误：类型不匹配
    pfi2s = pfi2i;    // 错误：类型不匹配
    return 0;
}
```

函数指针可以用 0 来初始化或赋值，以表示该指针不指向任何函数。

7.9.3 调用

指向函数的指针可以被用来调用它所指向的函数。调用函数时，不需要解引用操作符。无论是用函数名直接调用函数，还是用指针间接调用函数，两者的写法是一样的。例如：

```
#include <iostream>


int min( int*, int );
int (*pf)( int*, int ) = min;

const int iaSize = 5;
int ia[ iaSize ] = { 7, 4, 9, 2, 5 };

int main() {
    cout << "Direct call: min: "
```

```
            << min( ia, iaSize ) << endl;

        cout << "Indirect call: min: "
            << pf( ia, iaSize ) << endl;
        return 0;
    }
    int min( int* ia, int sz ) {
        int minVal = ia[ 0 ];

        for ( int ix = 1; ix < sz; ++ix )
            if ( minVal > ia[ ix ] )
                minVal = ia[ ix ];
        return minVal;
    }
```

调用

```
pf( ia, iaSize );
```

也可以用显式的指针符号写出：

```
(*pf)( ia, iaSize );
```

这两种形式产生相同的结果，但是第二种形式让读者更清楚该调用是通过函数指针执行的。

当然，如果函数指针的值为 0，则两个调用都将导致运行时刻错误。只有已经被初始化或赋值的指针（引用到一个函数）才可以被安全地用来调用一个函数。

7.9.4 函数指针的数组

我们可以声明一个函数指针的数组。例如：

```
int (*testCases[10])();
```

将 testCases 声明为一个拥有 10 个元素的数组。每个元素都是一个指向函数的函数指针该函数没有参数，返回类型为 int。

像数组 testCases 这样的声明非常难读，因为很难分析出函数类型与声明的哪部分相关。在这种情况下，使用 typedef 名字可以使声明更为易读。例如：

```
// typedefs 使声明更易读
typedef int (*PFV)();  // 定义函数类型指针的 typedef

PFV testCases[10];
```

testCases 的这个声明与前面的等价。

由 testCases 的一个元素引用的函数调用如下：

```
const int size = 10;
PFV testCases[size];
int testResults[size];

void runtests() {
    for ( int i = 0; i < size; ++i )
        // 调用一个数组元素
```

```
            testResults[ i ] = testCases[ i ]();
    }
```

函数指针的数组可以用一个初始化列表来初始化，该表中每个初始值都代表了一个与数组元素类型相同的函数。例如：

```
int lexicoCompare( const string &, const string & );
int sizeCompare( const string &, const string & );

typedef int ( *PFI2S )( const string &, const string & );
PFI2S compareFuncs[2] =
{
    lexicoCompare,
    sizeCompare
};
```

我们也可以声明指向 compareFuncs 的指针。这种指针的类型是“指向函数指针数组的指针。”声明如下：

```
PFI2S (*pfCompare)[2] = &compareFuncs;
```

声明可以分解为；

```
(*pfCompare)
```

解引用操作符（*）把 pfCompare 声明为指针，后面的[2]表示 pfCompare 是指向两个元素数组的指针：

```
(*pfCompare)[2]
```

typedef PFI2S 表示数组元素的类型，它是指向函数的指针，该函数返回 int，有两个 const string&型的参数。数组元素的类型与表达式&lexicoCompare 的类型相同，也与 compareFuncs 的第一个元素的类型相同。此外，它还可以通过下列语句之一获得：

```
compareFuncs[ 0 ];
(*pfCompare)[ 0 ];
```

要通过 pfCompare 调用 lexicoCompare，程序员可用下列语句之一：

```
// 两个等价的调用
pfCompare[ 0 ]( string1, string2 ); // 编写
((*pfCompare)[ 0 ])( string1, string2 ); // 显式
```

7.9.5 参数和返回类型

现在我们回头看一下本节开始提出的问题，在那里给出的任务要求我们写一个排序函数，怎样用函数指针写这个函数呢？因为函数参数可以是函数指针，所以我们把表示所用比较操作的函数指针作为参数传递给排序函数：

```
int sort( string*, string*,
          int (*)( const string &, const string & ) );
```

我们再次用 typedef 名字使 sort()的声明更易读：

```
// typedef 使 sort() 的声明更易读
typedef int ( *PFI2S )( const string &, const string & );
```

```
int sort( string*, string*, PFI2S );
```

因为在多数情况下使用的函数是 lexicoCompare()，所以我们让它成为缺省的函数指针参数：

```
// 提供缺省参数作为第三个参数
int lexicoCompare( const string &, const string & );
int sort( string*, string*, PFI2S = lexicoCompare );
```

sort()函数的定义可能像这样：

```
void sort( string *s1, string *s2,
           PFI2S compare = lexicoCompare )
{
    // 递归的停止条件
    if ( s1 < s2 ) {
        string elem = *s1;
        string *low = s1;
        string *high = s2 + 1;

        for (;;) {
            while ( compare( *++low, elem ) < 0 && low < s2) ;
            while ( compare( elem, *--high ) < 0 && high > s1) ;
            if ( low < high )
                low->swap(*high);
            else break;
        } // end, for(;;)

        s1->swap(*high);
        sort( s1, high - 1, compare );
        sort( high + 1, s2, compare );
    } // end, if ( s1 < s2 )
}
```

sort()是 C.A.R.Hoare 的**快速排序**（quicksort）算法的一个实现。让我们详细查看该函数的定义。该函数对 s1 和 s2 之间的数组元素进行排序。sort()是一个递归函数，它将自己逐步地应用在较小的子数组上。停止条件是当 s1 指向与 s2 相同的元素时或指向 s2 所指元素之后的元素（第 5 行）。

elem（第 6 行）被称作**分割元素**（partition element）。所有按字典序小于 elem 的元素部会被移到 elem 的左边，而所有大于的都被移到右边。现在，数组被分成若干个子数组。sort()被递归地应用在它们之上（第 20-21 行）。

for(;;)循环的目的是完成分割（第 10-17 行）。在循环的每次迭代中，low 首先被向前移动到第一个大于等于 elem 的数组元素的索引上（第 11 行）。类似地，high 一直被递减，直到移动到小于等于 elem 的数组最右元素的索引上（第 12 行）。如果 low 不再小于 high，则表示元素已经分隔完毕，循环结束。否则，这两个元素被交换，下一次迭代开始（第 14-16 行）。虽然数组已经被分隔，但 elem 仍然是数组的第一个元素。在 sort()被应用到两个子数组之前，第 19 行的 swap()把 elem 放到它在数组中最终正确的位置上。

数组元素的比较通过调用 compare 指向的函数来完成（第 11-12 行）。swap()字符串操作被调用，以便交换数组元素所指的字符串（swap()字符串操作在 6.11 节介绍）。

下面 main()的实现用到了我们的排序函数：

```
#include <iostream>
#include <string>

// 这些通常应该在头文件中
int lexicoCompare( const string &, const string & );
int sizeCompare( const string &, const string & );
typedef int (*PFI)( const string &, const string & );
void sort( string *, string *, PFI=lexicoCompare );

string as[10] = { "a", "light", "drizzle", "was", "falling",
          "when", "they", "left", "the", "museum" };

int main() {
     // 调用 sort()，使用缺省实参作比较操作
     sort( as, as + sizeof(as)/sizeof(as[0]) - 1 );

     // 显示排序之后的数组的结果
     for ( int i = 0; i < sizeof(as)/sizeof(as[0]); ++i )
     cout << as[ i ].c_str() << "\n\t";
}
```

编译并执行程序，生成下列输出：

```
"a"
"drizzle"
"falling"
"left"
"light"
"museum"
"the"
"they"
"was"
"when"
```

函数参数的类型不能是函数类型，函数类型的参数将被自动转换成该函数类型的指针。例如：

```
// typedef 表示一个函数类型
typedef int functype( const string &, const string & );
void sort( string *, string *, functype );
```

编译器把 sort()当作已经声明为：

```
void sort( string *, string *,
               int (*)( const string &, const string & ) );
```

上面这两个 sort()的声明是等价的。

注意，除了用作参数类型之外，函数指针也可以被用作函数返回值的类型。例如：

```
int (*ff( int ))( int*, int );
```

该声明将 ff()声明为一个函数，它有一个 int 型的参数，返回一个指向函数的指针，类型为：

```
int (*) ( int*, int );
```

同样，使用 typedef 名字可以使声明更容易读懂。例如，下面的 typedef PF 使得我们能更容易地分解出 ff()的返回类型是函数指针：

```
// typedef 使声明更易读
typedef int (*PF)( int*, int );

PF ff( int );
```

函数不能声明返回一个函数类型。如果是，则产生编译错误。例如，函数 ff()不能如下声明：

```
// typedef 表示一个函数类型
typedef int func( int*, int );

func ff( int ); // 错误：ff()的返回类型为函数类型
```

7.9.6 指向 extern "C"函数的指针

我们可以声明一个函数指针，它指向用其他程序设计语言编写的函数，我们可通过使用链接指示符来做到这一点。例如，指针 pf 指向一个 C 函数：

```
extern "C" void (*pf)(int);
```

当用 pf 调用一个函数时，被调用的函数是一个 C 函数。

```
extern "C" void exit(int);

// pf 指向 C 函数 exit()
extern "C" void (*pf)(int) = exit;
int main() {
    // ...

    // 调用名为 exit() 的 C 函数
    (*pf)(99);
}
```

指向 C 函数的指针与指向 C++函数的指针类型不同。记住，对于函数指针的初始化或者赋值，只有当被赋值的指针类型与赋值操作符右边的指针或函数完全匹配时，初始化或者赋值才是合法的。因此，指向 C 函数的指针不能用指向 C++函数的指针初始化或赋值，反之亦然。如果没有这样做，就会产生编译错误。例如：

```
void (*pf1)(int);
extern "C" void (*pf2)(int);

int main() {
    pf1 = pf2; // 错误：pf1 和 pf2 类型不同
    // ...
}
```

注意，对于某些 C++的实现，指向 C 函数指针的特性与指向 C++的相同。有些编译器可能接受上面的赋值作为语言的一种扩展。

当链接指示符应用在一个声明上时，所有被它声明的函数都将受到链接指示符的影响。在下面的例子中，参数 pfParm 也是一个 C 函数指针。链接指示符应用在该参数指向的函数

上。

```
// pfParm 是一个指向 C 函数的指针
extern "C" void f1( void(*pfParm)(int) );
```

因此，f1()是一个 C 函数，它有一个指向 C 函数指针的参数。因为指向 C 函数的指针与指向 C++函数的指针类型不同，所以传递给 f1()的实参必须是 C 函数名或指向 C 函数的指针（再次说明，在 C 函数指针与 C++函数指针有相同特性的编译器实现中，编译器可能会支持一种语言扩展，允许向 f1()传递一个 C++函数指针作为实参。）

由于链接指示符作用在声明中所有的函数上，那么我们应该怎样声明一个含有 C 函数指针的 C++函数的参数呢？解决方案是用 typedef。例如：

```
// FC 表示一个 C 函数类型
// 有一个 int 参数和 void 返回值
extern "C" typedef void FC( int );

// f2() 是一个带有一个参数的 C++函数
// 参数是一个 C 函数指针
void f2( FC *pfParm );
```

练习 7.21

7.5 节定义了函数 factorial()。定义一个函数指针，使它能够指向 factorial()，并通过该指针生成 11 的阶乘。

练习 7.22

下列声明的类型是什么？

```
(a) int (*mpf)(vector<int>&);
(b) void (*apf[20])(double);
(c) void (*(*papf)[2])(int);
```

怎样用 typedef 名字来使这些声明更容易阅读？

练习 7. 23

下列函数是在头文件<cmath>中定义的 C 库函数。

```
double abs(double);
double sin(double);
double cos(double);
double sqrt(double);
```

怎样声明一个 C 函数指针的数组，并初始化该数组使它包含这四个函数？写一个 main()函数，使它通过该数组的元素，用实参 97.9 调用 sqrt()。

练习 7.24

让我们回到 sort()的例子，已知函数定义：

```
int sizeCompare( const string &, const string & );
```

如果指向字符串的两个参数长度相同，则 sizeCompare()返回 0；否则，如果第一个参数表示的字符串长度比第二个参数的字符串长度短，则返回一个负数；如果大于，则返回一个正数。记住，字符串操作 size()返回字符串的长度。改变 main()函数，让它用指向 sizeCompare()的指针作为第三个参数来调用 sort()。

8

域和生命期

本章将回答关于在 C++中声明（declaration）的两个重要问题：声明引入的名字可以被用在什么地方？对一个程序来说，何时使用一个对象或调用一个函数比较安全：即由声明引入的运行时刻实体的生命期是什么？为回答第一个问题，我们将给出域的概念，并且介绍它们是怎样界定一个名字在程序文本文件中的可用范围。本章将介绍各种 C++域：全局域、局部域。而在本章结尾还将介绍一个更为高级的话题：名字空间域。为回答第二个问题，我们将讲述声明是怎样引入全局对象和函数（在整个程序生存期间一直有效的实体）、局部对象（在程序生存期间的子集上有效的对象），以及动态分配的对象（生命期由程序员控制的对象）的。此外我们还将查看与这些对象和函数相关的运行时刻特性。

8.1 域

C++程序中的每个名字都必须指向惟一的一个实体（对象、函数、类型或模板）。这并不意味着在一个 C++程序中，一个名字只能被使用一次。一个名字可以被重新使用以指向不同的实体，只要编译器能够根据**上下文**（context）区分出该名字的不同含义即可。用来区分名字含义的一般上下文就是**域**（scope）。C++支持三种形式的域：**局部域**（local scope）、**名字空间域**（namespace scope）以及**类域**（class scope）。

局部域是包含在函数定义（或者函数块）中的程序文本部分。每一个函数都有一个独立的局部域，在函数中的每个复合语句（或块）也有一个独立的局部域。

名字空间域是不包含在函数声明、函数定义或者类定义内的程序文本部分。程序的最外层的名字空间域被称作**全局域**（global scope），或**全局名字空间域**（global namespace scope）。对象、函数、类型以及模板都可以在全局域中定义。程序员也可以利用**名字空间定义**（namespace definition）来定义**用户声明的**（user-declared）的名字空间，它们被嵌套在全局域内。每个用户声明的名字空间都是一个不同的域，它们都与全局域不同。与全局域相同的是，用户声明的名字空间可以包含对象、函数、类型和模板的声明与定义，以及被嵌套其内的用户声明的名字空间。用户声明的名字空间见 8.5 节和 8.6 节。

每个类定义都引入了一个独立的类域。类定义和类域见第 13 章。

同一个名字在不同的域中可以引用不同的实体。例如，在下面的程序段中，有四个实体被命名为 s1：

```
#include <iostream>
#include <string>

// 按字典序比较 s1 和 s2
int lexicoCompare( const string &s1, const string &s2 )
{ ... }

// 比较 s1 和 s2 的长度
int sizeCompare( const string &s1, const string &s2 )
{ ... }
typedef int (*PFI)( const string &, const string & );

// 排序字符串数组
void sort( string *s1, string *s2, PFI compare =lexicoCompare )
{ ... }
string s1[10] = { "a", "light", "drizzle", "was", "falling",
     "when", "they", "left", "the", "school" };

int main()
{
     // 调用 sort() -用比较的缺省实参
     // 调用全局数组 s1
     sort( s1, s1 + sizeof(s1)/sizeof(s1[0]) - 1 );
     // display the sorted array
     for ( int i = 0; i < sizeof(s1) / sizeof(s1[0]); ++i )
     cout << s1[ i ].c_str() << "\n\t";
}
```

因为函数 lexicoCompare()、sizeCompare()和 sort()定义的域不同，以及这些域都不是全局域，所以这些域都可以定义一个名为 s1 的变量。

由声明引入的名字从声明点直到声明它的域结束为止都是可见的（包含其中的嵌套域）。因此，lexicoCompare()的参数名 s1 直到其域的结尾都是可见的，即，到 lexicoCompare()定义的结束。全局数组 s1 的名字从它的声明点直到文件的结尾都是可见的，包括里面的嵌套域，比如在 main()的定义中。

一般来说，在给定的域中，一个名字必须被声明为引用某个实体。例如，如果把下面的声明加入到前面的例子中，跟在全局域中数组 s1 的声明之后，则会产生一个编译错误：

```
void s1();     // 错误：重复声明名字 s1
```

重载函数是此规则的一个例外。在同一个域中我们可以定义不止一个同名的函数，只要每个函数的参数表不同就可以，第 9 章将讨论重载函数。

在 C++中，如果一个名字被用在表达式中，则在使用之前必须先声明它。如果在 main()函数使用 s1 之前编译器并没有找到它的声明，就会产生一个编译错误。**名字解析**（name resolution）是把表达式中的一个名字与某一个声明相关联的过程，也是给出这个名字的意义的过程。这个过程依赖于该名字是如何被使用的，以及使用该名字的域。我们对不同上下文

环境中的名字解析的讨论将贯穿全书。局部域的名字解析将在下一小节中讨论，函数模板定义中的名字解析将在 10.9 节讨论，类域中的名字解析将在 13 章结束时讨论，类模板定义中的名字解析将在 16.12 节中讨论。

域和名字解析是编译时刻的概念，它们应用在程序文本的某一部分上。这些概念给出了源文件中的程序文本的意义。编译器根据域规则和名字解析规则解释它所读入的程序文本。

8.1.1 局部域

局部域是包含在函数定义（或函数块）中的程序文本区。每一个函数都有一个独立的局部域。在函数中，每个复合语句（或块）也有它自己的局部域。局部域可以被嵌套。例如，下面的函数定义了两层局部域，它对一个有序的整型 vector 进行二分查找：

```
const int notFound = - 1; // 全局域
int binSearch( const vector<int> &vec, int val )
{ // 局部域：层次 #1
     int low = 0;
     int high = vec.size() - 1;
     while ( low <= high )
     { // 局部域：层次 #2
          int mid = ( low + high ) / 2;
          if ( val == vec[ mid ] ) return mid;
          if ( val < vec[ mid ] )
               high = mid - 1;
          else low = mid + 1;
     }
     return notFound; // 局部域：层次 #1
}
```

第一个局部域是 binSearch()函数体的域。它声明了函数参数 vec 和 val，以及变量 low 和 high。在 binSearch()中的 while 循环定义了一个嵌套的局部域，该嵌套的局部域声明了一个变量：整型 mid。此外，该嵌套局部域还使用了函数参数 vec 和 val 以及局部变量 high 和 low。全局域包括这两个局部域，它声明了一个整型常量：notFound。

为 vec 和 val 的函数参数名属于函数体的第一个局部域，这些名字不能在第一个局部域中被再次声明。例如：

```
int binSearch( const vector<int> &vec, int val )
{ // 局部域：层次 #1
     int val; // 错误：名字 val 的重复声明无效
     // ...
```

函数参数名可以在 binSearch()的函数体内以及嵌套的 while 循环域中使用，但是函数参数 vec 和 val 不能用在 binSearch()函数体之外。

局部域内的名字解析是这样进行的：首先查找使用该名字的域，如果找到一个声明，则该名字被解析。如果没有找到，则查找包含该域的域。这个过程会一直继续下去，直到找到一个声明或已经查找完整个全局域。如果后一种情况发生　即没有找到该名字的声明，则这个名字的用法将被标记为错误。

因为在名字解析期间查找域的顺序由内向外，所以在外围域中的声明被嵌套域中的同名

声明所隐藏。在前面的例子中，如果在全局域中 binSearch()的定义之前声明了变量 low，那么在 while 循环的嵌套局部域中使用的 low 仍然指向局部的 low 的声明，全局声明会被局部声明隐藏起来。例如：

```
int low;
int binSearch( const vector<int> &vec, int val )
{
    // low 的局部声明
    // 隐藏了全局域中的声明
    int low = 0;
    // ...
    // low 是局部变量
    while ( low <= high )
    { // ...
    }
    // ...
}
```

有一些语句允许在它的控制结构中定义变量。例如，for 循环允许在它的初始化语句中定义一个变量：

```
for ( int index = 0; index < vecSize; ++index )
{
    // index 只在这里可见
    if ( vec[ index ] == someValue )
        break;
}

// 错误：index 在这里不可见
if ( index != vecSize ) // 找到的元素
```

在 for 循环的初始化语句中定义的变量，如 index，只在 for 循环本身的局部域中及其中的嵌套局部域中可见（这是标准 C++中的情况，在标准 C++之前并非如此），就好像 for 语句是这样的：

```
// 编译器转换后的表示
{ // 不可见的复合语句
    int index = 0;
    for ( ; index < vecSize; ++index )
    {
        // ...
    }
}
```

这可以防止程序员在循环的局部域之外再次访问控制变量。如果程序员希望通过测试 index 来判断是否找到了这个值，则代码段必须重写如下：

```
int index = 0;
for ( ; index < vecSize; ++index )
{
    // ...
}
// ok: index 在这里可见
```

```
if ( index != vecSize ) // 找到元素
```

因为在 for 循环的初始化语句中声明的变量是局部于该循环的，所以该变量可以在同局部域内的其他 for 循环的控制结构中被再次使用。例如：

```
void fooBar( int *ia, int sz )
{
    for (int i=0; i<sz; ++i) ... // ok
    for (int i=0; i<sz; ++i) ... // ok: 不同的 i
    for (int i=0; i<sz; ++i) ... // ok: 不同的 i
}
```

类似的情形，我们也可以在一个 if 或 switch 语句的条件中声明变量，以及在 while 或 for 循环的条件中声明变量。例如：

```
if ( int *pi = getValue() )
{
    // pi != 0 --这里可以使用*pi
    int result = calc(*pi);
    // ...
}
else
{
    // pi 在这里也可见
    // pi == 0
    cout << "error: getValue() failed" << endl;
}
```

在 if 语句的条件中定义的变量，比如 pi，只在该 if 语句和相关的 else 语句，以及这些语句内部的嵌套域中可见。条件的值是变量初始化的值。如果 pi 被初始化为 0，即空指针值，则该条件值为 false，执行 if 语句的 else 部分。如果 pi 被任意其他非空指针值初始化，则该条件为 true，if 部分被执行。关于 if 语句、switch 语句、for 语句和 while 循环语句，见第 5 章中的讨论。

练习 8.1

在下面的代码例子中，请指出不同的域。ix 的哪些声明是错误的？为什么？

```
int ix = 1024;
int ix();
void func( int ix, int iy ) {
    int ix = 255;
    if ( int ix = 0 ) {
        int ix = 79;
        {
            int ix = 89;
        }
    }
    else {
        int ix = 99;
    }
}
```

练习 8.2

在下面的代码例子中，用到 ix 和 iy 的地方分别引用了哪些声明？

```
int ix = 1024;
void func( int ix, int iy ) {
     ix = 100;
     for( int iy = 0 ; iy < 400; iy += 100 ) {
          iy += 100;
          ix = 300;
     }
     iy = 400;
}
```

8.2 全局对象和函数

全局域内的函数声明将引入**全局函数**（global function），而在全局域内的变量声明将引入**全局对象**（global object）。全局对象是一个运行时刻实体，它在程序的整个执行过程中都存在。全局对象占据的存储区的**生命期**（lifetime）从程序启动开始，在程序终止时结束。

被调用的或者被取地址的全局函数必须有一个定义。同样地，程序中用到的全局对象也必须有一个定义。全局对象和非 inline 全局函数在一个程序内只能被定义一次，而只要给出的定义完全相同即可，inline 函数可以在一个程序中被定义多次。这要求全局对象和函数或者只有一个定义，或者在一个程序中有多个完全相同的定义，这样的要求被称为“**一次定义法则**（ODR，one definition rule）”。在本节中，我们将了解怎样按照 ODR 定义和声明全局对象和函数。

8.2.1 声明和定义

正如第 7 章所述，**函数声明**（declaration）指定了该函数的名字以及函数的返回类型和参数表。除了这些信息，**函数定义**（definition）还为函数提供了函数体，它是包含在花括号中的一个语句序列。在函数被调用之前，函数体必须先被声明。例如：

```
// 函数 calc() 的声明
// 其定义由其他文件提供
void calc(int);
int main()
{
     int locl = get(); // 错误: get() 尚未声明
     calc(locl);   // ok: 找到 calc() 的声明
     // ...
}
```

对象定义有下列两种形式：

```
type_specifier object_name;
type_specifier object_name = initializer;
```

例如，下面是 obj1 的定义。在该定义中，obj1 被初始化为 97：

```
int obj1 = 97;
```

下面是 obj2 的定义，它没有指定初始值：

```
int obj2;
```

在全局域中定义的对象，如果没有指定显式的初始值，则该存储区被初始化为 0。因此，下面两个定义中，var1 和 var2 有相同的初始值 0：

```
int var1 = 0;
int var2;
```

在一个程序中，一个全局对象只能有一个定义。因为在使用文件中的对象之前必须先要声明这个对象，所以对于一个由多个文件构成的程序来说，它应该能够只声明一个对象而不定义它。我们该怎样声明一个对象呢？

关键字 extern 为声明但不定义一个对象提供了一种方法。实际上，它类似于函数声明，承诺了该对象会在其他地方被定义：或者在此文本文件中的其他地方，或者在程序的其他文本文件中。例如：

```
extern int i;
```

对程序来说是一个“保证”，表示在其他某个地方存在一个如下所示的定义：

```
int i;
```

extern 声明不会引起内存被分配，它可以在同一文件中或同一程序的不同文件中出现多次。典型情况下，全局对象的声明只在公共的头文件中出现一次，当一个程序文件需要引用这个全局对象时，它可以包含这个头文件。

```
// 头文件
extern int obj1;
extern int obj2;

// 文本文件
int obj1 = 97;
int obj2;
```

既指定了关键字 extern，又指定了一个显式初始值的全局对象声明将被视为该对象的定义。编译器将会为其分配存储区，而且该对象后续的定义都被标记为错误。例如：

```
extern const double pi = 3.1416; // 定义

const double pi; // 错误：重复定义 pi
```

关键字 extern 也可以在函数声明中指定，惟一的影响是将该声明的隐式属性“在其他地方定义”变为显式的。这样的声明有下列形式：

```
extern void putValues( int*, int );
```

8.2.2 不同文件之间声明的匹配

在多个文件中声明对象或函数的一个可能问题是：在不同文件中的声明可能会随时间而不同或改变。C++为检查不同文件中函数声明的差异提供了一些支持。

例如，在文件 token.C 中，函数 addToken()被定义为带有一个 unsigned char 型的参数。

而在文件 lex.C 中，addToken()被调用，它被声明为带有一个 char 型的参数。

```
// ---- token.C 中 ----
int addToken( unsigned char tok ) { /* ... */ }

// ---- lex.C 中 ----
extern int addToken( char );
```

在 lex.C 中调用 addToken()会导致链接阶段失败。如果上面的程序链接成功，则会出现这样的情形，编译后的程序在 Sun Spare 工作站上测试，执行正常。然后它被送到 IBM390 机器上，该程序再次编译通过，不幸的是，它在第一次执行时就失败了，甚至连最简单的测试也无法通过。究竟发生了什么事情？

下面是部分 token 的声明：

```
const unsigned char INLINE = 128;
const unsigned char VIRTUAL = 129;
```

addToken()的调用如下：

```
curTok = INLINE;
// ...
addToken( curTok );
```

字符在一台机器上被实现为有符号类型，而在另一台机器上却是无符号类型。addToken()的错误声明使大于 127 的 token 在 char 为有符号类型的机器上溢出。如果代码例子通过编译和链接，则可能执行错误。

在 C++中，有一种机制，通过它可以把函数参数的类型和数目编码在函数名中，该机制叫做**类型安全链接**（type-safe-linkage）。类型安全链接可用来帮助捕捉不同文件中函数声明不匹配的情况。在前面的例子中，unsigned char 型的参数和 char 型参数的类型不同。由于类型安全链接，在 lex.C 中声明的 addToken()将会被标记为未定义的函数，而 token.C 中的定义则被视为定义了另外一个函数。

类型安全链接机制为文件之间的函数调用提供了类型检查手段。它对支持重载函数也是必需的。我们将在第 9 章关于重载函数的陈述中进一步讨论类型安全链接。

不同文件中出现的同一对象或函数声明的其他类型不匹配情况，在编译或链接时可能不会被捕捉到。因为编译器一次只能处理一个文件，它不能很容易地检查到文件之间的类型违例。这些类型违例可能是程序严重错误的根源。例如，文件之间错误的对象声明或函数返问类型就不能被检测出来。这样的错误只能在运行时刻异常或程序的错误输出中才能被揭示出来。

```
// token.C 中
unsigned char lastTok = 0;
unsigned char peekTok() { /* ... */ }

// in lex.C
extern char lastTok;          // 最后一个 token
extern char peekTok();        // 查看 token
```

使用头文件是防止此类错误的基本法则。这是下一小节的话题。

8.2.3 谈谈头文件

头文件为所向 extern 对象声明、函数声明以及 inline 函数定义提供了一个集中的位置：这被称作声明的**局部化**（localization）。如果一个文件要使用或定义一个对象或函数时，它必须**包含**（include）相应的头文件。

头文件提供了两个安全保证。第一，保证所有文件都包含同一个全局对象或函数的同一份声明。第二，如果需要修改声明，则只需改变一个头文件。从而不至于再发生只修改了某一个特殊的文件中的声明。addToken()例子给出了如下的 token.h 头文件：

```
// ---- token.h ----
typedef unsigned char uchar;
const uchar INLINE = 128;
// ...
const uchar LT = ...;
const uchar GT = ...;
extern uchar lastTok;
extern int addToken( uchar );
inline bool is_relational( uchar tok )
{ return (tok >= LT && tok <= GT); }

// ----- lex.C -----
#include "token.h"
// ...

// ----- token.C -----
#include "token.h"
// ...
```

设计头文件有一些要注意的地方。头文件提供的声明逻辑上应该属于一个组。编译头文件也需要时间。如果头文件过大，或分散的元素太多，程序员可能会不愿意因为包含它而增加编译时间开销。为降低编译时间开销，有些 C++实现提供了预编译头文件支持。请查询系统的 C++实现参考手册；了解怎样从一个普通的 C++头文件创建预编译头文件。如果应用程序有很大的头文件，则使用预编译头文件而不是普通头文件可以大大降低应用程序的编译时间。

第二个考虑是，头文件不应该含有非 inline 函数或对象的定义。例如，下面的代码表示的正是这样的定义，因此不应该出现在头文件中：

```
extern int ival = 10;
double fica_rate;
extern void dummy() {}
```

虽然 ival 是用 extern 声明的，但是它的显式初始化使得它实际上是个定义。类似的情况，虽然 dummy()显式地声明为 extern，但是空花括号代表该函数的定义。尽管 fica_rate 没有被显式地初始化，但是因为缺少 extern，因而也被视为 C++中实际的定义。这些定义如果在同一程序的两个或多个文件中被包含，就会产生重复定义的编译错误。

在前面给出的 token.h 头文件中，常量 INLINE 和 inline 函数 is_relational()好像都违反了这条规则。但是，其实并非如此，虽然它们全是定义，但是符号常量定义以及 inline 函数定

义是特殊的定义。符号常量和 inline 函数可以被定义多次。

在程序编译期间，在可能的情况下，符号常量的值会代替该名字的出现。这个替代过程被称为**常量折叠**（constant folding）。例如，当 INLINE 被用在一个文件中时，编译器用 128 代替名字 INLINE。为了使编译器能够用一个常量值替换它的名字，该常量的定义（它的初始值）必须在它被使用的文件中可见。因为这个原因，符号常量可以在同一程序的不同文件中被定义多次。尽管理想情况下，一个具有初始值的常量可以被包含在多个不同的文件中，但是常量折叠使其变得并不必需，甚至在可执行文件只要出现一次就行。

但是，在某些情况下不可能做到符号常量的常量折叠过程。在这样的情况下，最好把常量的初始化移到某一个程序文本文件中，这可以由显式地声明常量为 extern 来实现。例如：

```
// ----- 头文件 -----
const int buf_chunk = 1024;
extern char *const bufp;

// ----- 程序文本文件 -----
char *const bufp = new char[buf_chunk];
```

虽然 bufp 被声明为 const，但是它的值却无法在编译时刻被计算出来（它的初始化值是一个要求调用库函数的 new 表达式）。如果 bufp 在头文件中被初始化，那么它将在每个包含它的文件中被定义。这不但浪费了空间，而且可能与程序员的意图不符。

符号常量是任何 const 型的对象。当下面的声明被放到一个头文件中，并且由程序的两个独立的文件包含它时，就会导致链接错误，你知道这是为什么吗？

```
// 喔！不应该被放在一个头文件中
const char* msg = "?? oops: error: ";
```

问题出在 msg 不是常量，它是一个指向常量值的非常量指针。常量指针的声明如下（指针声明的完整讨论见第 3 章）：

```
const char *const msg = "?? oops: error: ";
```

该常量指针的定义可以出现在多个文件中。

与符号常量类似的情形也适用于 inline 函数。为使编译器能够在函数被调用的地方‘内联地”展开函数体，它必须能够看到 inline 函数的定义（inline 函数在 7.6 节介绍）。因此：如果一个 inline 函数将在多个文件中被用到，那么它必须被定义在头文件中。但是，指定一个函数为 inline 只是暗示该函数应该被内联。编译器实际上是否内联该函数——程序中的一般或某些特殊调用——会随编译器的实现而不同。如果编译器在调用点上没有内联该函数，则编译器会为该函数生成一个定义，放到可执行文件中。如果在多个文件中生成同一函数的定义，则会产生一个不必要的、过大的可执行文件。

如果出现下列情况，多数编译器都会产生警告（一般情况下，这要求打开编译器的警告模式。）：

1．函数的定义使其根本不可能做成 inline 函数。例如，编译器可能抱怨函数过于复杂而无法内联。在这种情况下，如果可能，就应重写该函数。否则，去掉 inline 指示符，把函数定义放到程序文本文件中。

2．函数的特殊调用不能被内联。例如，在 C++的最初实现（AT&T(cfront)）中，同

—表达式中的一个 inline 函数的第二次调用就无法被内联。在这种情况下，我们可以把表达式重新改写成两个独立的 inline 函数调用。

在把一个函数声明为 inline 之前，我们必须分析它的运行时刻行为，以确信该函数被内联对于这部分代码来说确实是必要的。建议把那些天生无法内联的函数不声明为 inline，并且不放在头文件中。

练习 8.3

指出下列语句哪些是声明，哪些是定义，为什么？

```
(a) extern int ix = 1024;
(b) int iy;
(c) extern void reset( void *p ) { /* ... */ }
(d) extern const int *pi;
(e) void print( const matrix & );
```

练习 8.4

在下列声明和定义中，哪些应被放到头文件中？哪些应被放到程序文本文件中？为什么？

```
(a) int var;
(b) inline bool is_equal( const SmallInt &, const SmallInt & ) { }
(c) void putValues( int *arr, int size );
(d) const double pi = 3.1416;
(e) extern int total = 255;
```

8.3 局部对象

在局部域中的变量声明引入了**局部对象**（local object）。有三种局部对象。**自动对象**（automatic object）、**寄存器对象**（register object）以及**局部静态对象**（local static object）。区分这些对象的是对象所在存储区的属性和生命期。自动对象所在存储区从声明它的函数被调用时开始，一直到该函数结束为止。寄存器对象是一种自动对象，它支持对其值的快速存取。局部静态对象的存储区在该程序的整个执行期间一直存在。本节我们将讨论这三种局部变量的属性。

8.3.1 自动对象

自动对象的存储分配发生在定义它的函数被调用时。分配给自动变量的存储区来自于程序的运行栈，它是函数的活动记录的一部分。自动对象也被称为具有**自动存储持续时间**（automatic storage duration），或**自动范围**（automatic extent）。未初始化的自动对象包含一个随机的位模式，是该存储区上次被使用的结果。它的值被称为**未指定的**（unspecified）。

在函数结束时，它的活动记录被从运行栈中弹出。与该自动对象相关联的存储区被真正释放。对象的生命期在函数结束时结束，它包含的任何值都被抛弃。

因为与自动对象相关联的存储区在函数结束时被释放，所以应该小心使用自动对象的地

址。自动对象的地址不应该被用作函数的返回值，因为函数一旦结束了，该地址就指向一个无效的存储区。例如：

```
#include "Matrix.h"

Matrix* trouble( Matrix *pm )
{
{
     Matrix res;
     // 用 pm 做一些事情
     // 把结果赋值给 res
     return &res; // 糟糕!
}

int main()
{
     Matrix m1;
     // ...
     Matrix *mainResult = trouble( &m1 );
     // ...
}
```

mainResult 被设置为自动 Matrix 对象 res 的地址。不幸的是，res 的存储区在 trouble()完成时被释放。在返回到 main()时，mainResult 指向一个未分配的内存。（在本例中，该地址可能仍然有效，因为我们还没有调用其他函数覆盖掉 trouble()函数的活动记录的部分或全部，所以这样的错误很难检测。）在 main()中的后续代码部分使用 mainResult 会产生意想不到的结果。

但是，把 main()的自动变量 m1 的地址传递给函数 trouble()则是安全的。我们可以保证，在 trouble()调用期间，main()的存储区在栈中一直是有效的，因此，ml 的内存区在 trouble()调用期间都是可被访问的。

当一个自动变量的地址被存储在一个生命期长于它的指针时，该指针被称为**空悬指针**（dangling pointer）。这是一个严重的程序员错误，因为它所指的内容是不可预测的。如果该地址的值正好合适（因此程序就不会产生段错误），该程序可能一直执行到完成，但是给出的是一个无效的结果。

8.3.2 寄存器自动对象

在函数中频繁被使用的自动变量可以用 register 声明。如果可能的话，编译器会把该对象装载到机器的寄存器中。如果不能够的话，则对象仍位于内存中。出现在循环语句中的数组索引和指针是寄存器对象的很好例子。

```
for ( register int ix = 0; ix < sz; ++ix ) // ...
for (register int *p = array ; p < arraySize; ++p ) // ...
```

函数参数也可以被声明为寄存器变量；

```
bool find( register int *pm, int val ) {
     while ( *pm )
          if ( *pm++ == val ) return true;
```

```
        return false;
    }
```

如果所选择的变量被频繁使用，则寄存器变量可以提高函数的执行速度。

关键字 register 对编译器来说只是一个建议。有些编译器可能忽略该建议，而是使用寄存器分配算法找出最合适的候选放到机器可用的寄存器中。因为编译器知道运行该程序的机器的结构，所以它选择寄存器的内容时常常会做出更有意义的决定。

8.3.3 静态局部对象

我们也能够在函数定义或者函数定义的复合语句中，声明可在整个程序运行期间一直存在的局部对象。当一个局部变量的值必须在多个函数调用之间保持有效时，我们不能使用普通的自动对象，自动对象的值在函数结束时被丢弃。

这种情形的一种解决方案是把局部对象声明为 static。静态局部对象具有**静态存储持续期间**（static storage duration），或**静态范围**（static extent）。虽然它的值在函数调用之间保持有效，但是其名字的可视性仍限制在其局部域内。静态局部对象在程序执行到该对象的声明处时被首次初始化。例如，下面是 gcd()的一个版本，它占用一个静态局部对象来跟踪递归的深度：

```
#include <iostream>

int traceGcd( int v1, int v2 )
{
    static int depth = 1;
    cout << "depth #" << depth++ << endl;
    if ( v2 == 0 ) {
        depth = 1;
        return v1;
    }
    return traceGcd( v2, v1%v2 );
}
```

与静态局部对象 depth 相关联的值在 traceGcd()的调用之间保持有效。初始化只在 traceGcd()首次被调用时执行一次。下面的小程序使用了 traceGcd()：

```
#include <iostream>

extern int traceGcd(int, int);
int main() {
    int rslt = traceGcd( 15, 123 );
    cout << "gcd of (15,123): " << rslt << endl;
    return 0;
}
```

编译并运行该程序，产生下列输出：

```
depth #1
depth #2
depth #3
depth #4
gcd of (15,123): 3
```

未经初始化的静态局部对象会被程序自动初始化为 0。相反，自动对象的值会是任意的，除非它被显式初始化。下面的程序说明了自动和静态局部变量的缺省初始化以及不初始化自动对象的危险。

```
#include <iostream>

const int iterations = 2;

void func() {
     int value1, value2; // 未初始化
     static int depth; // 隐式初始化为 0
     if ( depth < iterations )
          { ++depth; func(); }
     else depth = 0;
     cout << "\nvalue1:\t" << value1;
     cout << "\tvalue2:\t" << value2;
     cout << "\tsum:\t" << value1 + value2;
}

int main() {
     for ( int ix = 0; ix < iterations; ++ix ) func();
     return 0;
}
```

执行后结果如下：

```
value1: 0          value2: 74924     sum: 74924
value1: 0          value2: 68748     sum: 68748
value1: 0          value2: 68756     sum: 68756
value1: 148620     value2: 2350      sum: 150970
value1: 2147479844 value2: 671088640 sum: - 1476398812
value1: 0          value2: 68756     sum: 68756
```

注意，value1 和 value2 是未经初始化的自动对象。它们的初始值如程序输出所示，完全是个随机值，因此求和的结果也是不能预测的。但是，即使 depth 没有被初始化，它的值也会被保证是 0，保证 func()递归地调用它自己两次。

8.4 动态分配的对象

全局对象和局部对象的生命期是严格定义的，程序员不能以任何方式改变它们的生命期。但是，有时候需要创建一些生命期能被程序员控制的对象，它们的分配和释放可以根据程序运行中的操作来决定。例如，有人可能希望只在程序运行中遇到错误时，才分配一个字符串来包含错误消息的文本。如果程序不只产生一种错误消息，那么分配的字符串的长度会随着遇到的错误文本的长度而变化。我们无法预先知道应该分配多长的字符串，因为字符串的长度取决于在程序执行期间遇到的错误种类。

第三种对象允许程序员完全控制它的分配与释放。这样的对象被称为**动态分配的对象**（dynamically allocated object）。动态分配的对象被分配在程序的**空闲存储区**（free store）的可用内存池中。程序员用 new 表达式创建动态分配的对象，用 delete 表达式结束此类对象的

生命期。动态分配的对象可以是单个对象，也可以是对象的数组。在空闲存储区中分配的数组的长度可以在运行时刻计算。

在本节中，关于动态分配的对象，我们将会了解到三种形式的 new 表达式：一种支持单个对象的动态分配，另一种支持数组的动态分配，第三种形式被称为定位 new 表达式（placement new expression）。当空闲存储区被耗尽时，new 表达式会抛出异常，我们将在第 11 章进一步讨论异常。在第 15 章中，我们将详细讨论 new 表达式和 delete 表达式的用法。

8.4.1 单个对象的动态分配与释放

new 表达式是由关键字 new 及其后面的类型指示符构成的，该类型指示符可以是内置类型或 class 类型。例如：

```
new int;
```

从空闲存储区分配了一个 int 型的对象。类似地，

```
new iStack;
```

分配了一个 iStack 类对象。

new 表达式本身并不是十分有用。我们如何使用被分配的对象呢？空闲存储区的一个特点是，其中分配的对象没有名字。new 表达式没有返回实际分配的对象，而是返回指向该对象的指针。对该对象的全部操作都要通过这个指针间接完成。例如：

```
int *pi = new int;
```

该 new 表达式创建了一个 int 型的对象，由 pi 指向它。

在运行时刻从空闲存储区中分配内存，比如通过上面的 new 表达式，我们称之为动态内存分配（dynamic memory allocation）。我们说 pi 指向的内存是被动态分配的。

空闲存储区的第二个特点是分配的内存是未初始化的。空闲存储区的内存包含随机的位模式。它是程序运行前该内存上次被使用留下的结果。测试

```
it ( *pi == 0 )
```

总是会失败，因为由 pi 指向的对象含有随机的位。因此我们建议对用 new 表达式创建的对象进行初始化。程序员可以按如下方式初始化上个例子中的 int 型对象：

```
int *pi = new int( 0 );
```

括号内的常量给出了一个初始值，它被用来初始化 new 表达式创建的对象。因此，pi 指向一个 int 型的对象，该对象的值为 0。括号中的表达式被称作**初始化式**（initializer）。初始化式的值不一定是常量，任意的能够被转换成 int 型结果的表达式都是有效的初始化式。

new 表达式的操作序列如下：从空闲存储区分配对象，然后用括号内的值初始化该对象。为从空闲存储区分配对象，new 表达式调用库操作符 new()。前面的 new 表达式与下列代码序列大体上等价：

```
int ival = 0;       // 创建一个用 0 初始化的 int 对象
int *pi = &ival;    // 现在指针指向这个对象
```

当然，不同的是，pi 指向的对象是由库操作符 new()分配的，位于程序的自由存储区中。类似地，如下语句：

```
iStack *ps = new iStack( 512 );
```

创建了一个内含 512 个元素的 iStack 型的对象。在类对象的情况下，括号中的值被传递给该类相关的构造函数，它在该对象被成功分配之后才被调用（类对象的动态分配将在 15.8 节详细讨论。本节余下部分将集中在内置类型上。）

到目前为止我们所讨论的 new 表达式有一个问题。很不幸，空闲存储区代表的是有限的资源。在程序执行的某一个点上，空闲存储区可能会被耗尽，从而导致 new 表达式失败。如果 new 表达式调用的 new()操作符不能得到要求的内存，通常会抛出一个 bad_alloc 异常（异常处理将在第 11 章讨论。）。

当指针 pi 所指对象的内存被释放时，它的生命期也随之结束。当 pi 成为 delete 表达式的操作数时，该内存被释放。例如：

```
delete pi;
```

释放了 pi 指向的内存，结束了 int 型对象的生命期。通过把 delete 表达式放在程序中的适当位置上，程序员就可以控制在何时结束对象的生命期。delete 表达式调用库操作符 delete()，把内存还给空闲存储区。因为空闲存储区是有限的资源，所以当我们不再需要已分配的内存时，就应该马上将其返还给空闲存储区，这是很重要的。

看过前面的 delete 表达式，你可能会问，如果 pi 因为某种原因被设置为 0，又会怎么样呢？代码不应该像这样吗？

```
// 这样做有必要吗?
if ( pi != 0 )
    delete pi;
```

答案是不。如果指针操作数被设置为 0，则 C++会保证 delete 表达式不会调用操作符 delete()。没有必要测试其是否为 0（实际上，在多数实现下，如果增加了指针的显式测试，那么该测试实际上会被执行两次。）

在这里，讨论 pi 的生命期和 pi 指向的对象的生命期之间的区别是很重要的。指针 pi 本身是个在全局域中声明的全局对象。结果，pi 的存储区在程序开始之前就被分配，且一直保持到程序结束。这与 pi 指向的对象的生命期不同，后者是在程序执行过程中遇到 new 表达式时才被创建的。pi 指向的内存是动态分配的，它拥有的对象是动态分配的对象。因此，pi 是一个全局指针，指向一个动态分配的 int 型对象。当程序运行期间遇到 delete 表达式时，pi 指向的内存就被释放了。但是，指针 pi 的内存及其内容并没有受 delete 表达式的影响。在 delete 表达式之后，pi 被称作空悬指针，即指向无效内存的指针。空悬指针是程序错误的一个根源，它很难被检测到。一个比较好的办法是在指针指向的对象被释放后，将该指针设置为 0。这样可以清楚地表明该指针不再指向任何对象。

delete 表达式只能应用在指向的内存是用 new 表达式从空闲存储区分配的指针上。将 delete 表达式应用在指向空闲存储区以外内存的指针上，会使程序运行期间出现未定义的行为。但是，正如前面看到的，delete 表达式应用在值为 0 的指针（即不指向任何对象的指针）上，不会引起任何麻烦。下面的例子给出了安全的和不安全的 delete 表达式：

```
void f() {
    int i;
    string str = "dwarves";
```

```
    int *pi = &i;
    short *ps = 0;
    double *pd = new double(33);

    delete str;   // 糟糕: "dwarves" 不是动态对象
    delete pi;    // 糟糕: pi 指向 i, 一个局部对象

    delete ps;    // 安全
    delete pd;    // 安全
}
```

下面三个常见程序错误都与动态内存分配有关：

1．应用 delete 表达式失败，使内存无法返回空闲存储区。这被称作**内存泄漏**（memory leak）。

2．对同一内存区应用了两次 delete 表达式。这通常发生在两个指针指向同一个动态分配对象的时候。这是一个很难跟踪的问题。若多个指针指向同一个对象，当通过某一个指针释放了该对象时就会发生这样的情况。此时，该对象的内存被返回给空闲存储区，然后又被分配给某个别的对象。接着指向旧对象的第二个指针被释放，新对象也就跟着消失了。

3．在对象被释放后读写该对象。这常常会发生，因为 delete 表达式应用的指针没有被设置为 0。

这些操纵动态分配内存的错误比较容易出现，而且难于跟踪和修正。为帮助程序员更好地管理动态分配的内存，C++库提供了 auto_ptr 类类型的支持。这是下一小节的话题。在那之后，我们将会看到用 new 和 delete 表达式的第二种形式：动态分配和释放数组。

8.4.2 auto_ptr ※

auto_ptr 是 C++标准库提供的类模板，它可以帮助程序员自动管理用 new 表达式动态分配的单个对象（不幸的是，对用 new 表达式分配的数组管理没有类似的支持，我们不能用 auto_ptr 存储数组。如果这样做了，结果将是未定义的。）。

auto_ptr 对象被初始化为指向由 new 表达式创建的动态分配对象。当 auto_ptr 对象的生命期结束时，动态分配的对象被自动释放。在本小节中，我们将看看怎样把 auto_ptr 对象与 new 表达式创建的对象关联起来。

在使用 anto_ptr 类模板之前，必须包含下面的头文件：

```
#include <memory>
```

auto_ptr 对象的定义有下列三种形式：

```
auto_ptr< type_pointed_to > identifier( ptr_allocated_by_new );
auto_ptr< type_pointed_to > identifier( auto_ptr_of_same_type );
auto_ptr< type_pointed_to > identifier;
```

type_pointed_to 代表由 new 表达式创建的对象的类型。我们来依次看一下这些定义。在最常见的情况下，我们希望把 auto_ptr 直接初始化为 new 表达式返回的对象地址。我们可以这样做：

```
auto_ptr< int > pi( new int( 1024 ) );
```

pi 被初始化为由 new 表达式创建的对象的地址，且该对象的初始化值为 1024。我们可以

检查 auto_ptr 所指的对象的值，方式与普通指针相同：

```
if ( *pi != 1024 )
    // 喔，出错了
else *pi *= 2;
```

new 表达式创建的对象由 pi 指向，当 pi 的生命期结束时，它将被自动释放。如果 pi 是个局部对象，则 pi 所指的对象在定义 pi 的模块结束时被释放。如果 pi 是全局对象，则 pi 所指的对象在程序结束时被释放。

如果我们用一个 class 类型的对象初始化 auto_ptr 对象，比如标准 string 类型，会怎么样呢？例如：

```
auto_ptr< string >
    pstr_auto( new string( "Brontosaurus" ) );
```

假设我们现在希望访问一个字符串操作。对于普通的 string 指针，我们会这样做：

```
string *pstr_type = new string( "Brontosaurus" );

if ( pstr_type ->empty() )
    // 喔，出错了
```

那么，怎样用 auto_ptr 对象访问字符串操作 empty()呢？我们将使用相同的方式：

```
auto_ptr< string > pstr_auto( new string( "Brontosaurus" ) );
if ( pstr_auto->empty() )
    // 喔，出错了
```

auto_ptr 类模板背后的主要动机是支持与普通指针类型相同的语法，但是为 auto_ptr 对象所指对象的释放提供自动管理。根据一般的常识，你可能会认为这种额外的安全性一定来自于执行效率的开销，但实际情况并不是这样。因为对这些操作的支持都是内联的（它们由编译器在调用点上展开），所以使用 auto_ptr 对象并不比直接使用指针代价更高。

在下面的情况下，我们用 pstr_auto 的值初始化 pstr_auto2，并且 pstr_auto 的底层对象是 string，会怎样呢？

```
// 谁负责 string 的删除操作?
auto_ptr< string > pstr_auto2( pstr_auto );
```

假定直接用一个 string 指针初始化另一个，比如：

```
string *pstr_type2( pstr_type );
```

那么，这两个指针都持有程序空闲存储区内的字符串地址，我们必须小心地将 delete 表达式只应用在一个指针上。而 auto_ptr 类模板支持所有权概念。

当定义 pstr_auto 时，它知道自己对初始化字符串拥有所有权，并且有责任删除该字符串。这是所有权授予 auto_ptr 对象的责任。

问题是，当 pstr_auto2 被初始化为指向与 pstr_auto 相同的对象时，所有权会发生什么样的变化？我们不希望让两个 auto_ptr 对象都拥有同一个底层对象——这会引起重复删除对象的问题，这也是我们使用 auto_ptr 类型首先要防止的。

当一个 auto_ptr 对象被用另一个 auto_ptr 对象初始化或赋值时，左边被赋值或初始化的对象就拥有了空闲存储区内底层对象的所有权，而右边的 auto_ptr 对象则撤消所有责任。于是，在我们的例子中，将是用 pstr_auto2 删除字符串对象，向不是 pstr_auto，pstr_auto 不再

被用来指向字符串对象。

类似的行为也发生在赋值操作符上，已知下列两个 auto_ptr 对象：

```
auto_ptr< int > p1( new int( 1024 ) );
auto_ptr< int > p2( new int( 2048 ) );
```

赋值操作符可以将一个 auto_ptr 对象拷贝到另一个中，如下所示：

```
p1 = p2;
```

在赋值之前，由 p1 指向的对象被删除。赋值之后，p1 拥有 int 型对象的所有权，该对象值为 2,048。p2 不再被用来指向该对象。

在 auto_ptr 定义的第三种形式中，我们创建一个 auto_ptr 对象，但是没有用指针（指向空闲存储区中对象）将其初始化。例如：

```
// 没有指向任何对象
auto_ptr< int > p_auto_int;
```

因为 p_auto_int 没有被初始化指向一个对象，所以它的内部指针值被设置为 0。这意味着对它解除引用会使程序出现未定义的行为，就好像我们直接解引用一个值为 0 的指针时所发生的一样：

```
// 喔！解引用一个没有指向任何对象的 auto_ptr
if ( *p_auto_int != 1024 )
     *p_auto_int = 1024;
```

对于普通指针，我们只需测试是否为 0。例如：

```
int *pi = 0;
if ( pi != 0 ) ...;
```

但是怎样测试一个 auto_ptr 对象是否指向一个底层对象呢？操作 get()返回 auto_ptr 对象内部的底层指针。所以，为了判断 auto_ptr 对象是否指向一个对象，我们可以如下编程：

```
// 修改后的测试：保证 p_auto_int 指向一个对象
if ( p_auto_int.get() != 0 &&
     *p_auto_int != 1024 )
     *p_auto_int = 1024;
```

如果它没有指向一个对象，那么怎样使其指向一个呢——即，怎样设置一个 auto_ptr 对象的底层指针？我们可以用 reset()操作。例如：

```
else
     // ok，让我们设置 p_auto_int 的底层指针
     p_auto_int.reset( new int( 1024 ) );
```

我们不能够在 auto_ptr 对象被定义之后，再用 new 表达式创建对象的地址来直接向其赋值。因此，我们不能这样写：

```
void example()
{
     // 缺省，用 0 初始化
     auto_ptr< int > pi;
     {
          // 不支持
          pi = new int( 5 );
```

```
        }
    }
```

为了重置一个 auto_ptr 对象，我们必须使用 reset()函数。我们可以向 reset()传递一个指针；如果不希望设置（或者取消原来的设置）该 auto_ptr 对象的话，可以传进一个 0 值。如果 auto_ptr 当前指向一个对象并且该 auto_ptr 对象拥有该对象的所有权，则该对象在底层指针被重置之前，首先被删除。例如：

```
auto_ptr< string >
    pstr_auto( new string( "Brontosaurus" ) );

// 在重置之前删除对象 Brontosaurus
pstr_auto.reset( new string( "Long -neck" ) );
```

在这种情况下，用字符串操作 assign()对原有的字符串对象重新赋值，比删除原有的字符率对象并重新分配第二个字符串对象更为有效：

```
// 这种情况下，重置的更有效形式
// 用 string 的 assign() 设置新值
pstr_auto->assign( "Long-neck" );
```

程序设计的一件难事是：有时候仅仅得到正确的结果是不够的。有时候，我们不仅需要正确的结果，而且还需要一个可接受的性能。像调用 assign()有效释放和重分配一个字符串这样的小事情就是一个很好的例子，它说明在某些情况下，这些小细节会积聚成可怕的性能瓶颈。这些细节不应该烦忧那些试图为整个程序提供解决方案的人，但是这些细节是有经验的程序员应该考虑的。

auto_ptr 类模板为动态分配内存提供了大量的安全性和便利。但是，我们仍需小心，否则我就会陷入麻烦。我们可能会做错些什么呢？

1．我们必须小心，不能用一个指向“内存不是通过应用 new 表达式分配的”指针来初始化或赋值 auto_ptr。如果这样做了，delete 表达式会被应用在不是动态分配的指针上，这将导致未定义的程序行为。

2．我们必须小心，不能让两个 auto_ptr 对象拥有空闲存储区内同一对象的所有权。一种很显然犯这种错误的方法是，用同一个指针初始化或赋值两个 auto_ptr 对象。另一种途径是通过使用 get()操作。例如：

```
auto_ptr< string >
pstr_auto( new string( "Brontosaurus" ) );

// 喔！现在两个指针都指向同一个对象
// 并都拥有该对象的所有权
auto_ptr< string > pstr_auto2( pstr_auto.get() );
```

release()操作允许将一个 auto_ptr 对象的底层对象初始化或赋位给第二个对象，而不会使两个 auto_ptr 对象同时拥有同一对象的所有权。release()不仅像 get()操作一样返回底层对象的地址，而且还释放这对象的所有权。前面代码段可被正确改写如下：

```
// ok: 两个对象仍然指向同一个对象
// 但是，pstr_auto 不再拥有拥有权
auto_ptr< string >
    pstr_auto2( pstr_auto.release() );
```

8.4.3 数组的动态分配与释放

new 表达式也可以在空闲存储区中分配数组。在这种情况下，new 表达式中的类型指示符后面必须有一对方括号，里面的维数是数组的长度，且该组数可以是一个复杂的表达式。new 表达式返回指向数组第一个元素的指针。例如：

```
// 分配单个 int 型的对象
// 用 1024 初始化
int *pi = new int( 1024 );

// 分配一个含有 1024 个元素的数组
// 未被初始化
int *pia = new int[ 1024 ];

// 分配一个含 4x1024 个元素的二维数组
int (*pia2)[ 1024 ] = new int[ 4 ][ 1024 ];
```

pi 指向一个 int 型的单个对象，初始值为 1024。pia 指向数组的第十个元素，该数组有 1024 个元素。pia2 指向一个由四个 1024 个元素的数组构成的数组的第一个元素——即，pia2 指向一个有 1024 个元素的数组。

一般地，在空闲存储区上分配的数组不能给出初始化值集。（在 15.8 节，我们将了解在空闲存储区分配的类数组怎样用类的缺省构造函数进行初始化。）我们不可能在前面的 new 表达式中，通过指定初始值来初始化数组的元素。在空闲存储区中创建的内置类型的数组必须在 for 循环中被初始化，即数组的元素被一个接一个地初始化：

```
for ( int index = 0; index < 1024; ++index )
    pia[ index ] = 0;
```

动态分配数组的主要好处是，它的第一维不必是常量值：即，在编译时刻不需要知道维数，就像局部域或全局域中的定义所引入的数组的维数一样。这意味着我们可以分配符合当前程序所需要大小的内存。例如，在实际的 C++程序中，如果在程序执行期间，一个指针可能会指向许多个 C 风格的字符串，那么，被用来存放 C 风格字符串的内存（也就是该指针所指的字符串），通常是在程序执行期间根据字符串的长度动态分配所得。该技术比分配能够存放所有字符串的固定长度的数组更为有效。因为固定长度的字符串必须足够大，以便能够存放最大可能的字符串，尽管多数情况下字符串的长度可能都比较短。而且，如果有一个字符串实例比我们确定的固定长度还要长，则我们的程序就会失败。

下面的例子说明了怎样用 new 表达式将数组的第一维指定为运行时刻的一个值。假设有下列 C 风格的字符串：

```
const char *noerr = "success";

// ...
const char *err189 = "Error: a function declaration must "
                     "specify a function return type!";
```

由 new 表达式分配的数组的维数可被指定为一个在运行时刻才被计算出来的值，如下所示：

```
#include <cstring>
```

```
const char *errorTxt;
if (errorFound)
     errorTxt = err189;
else
     errorTxt = noerr;

int dimension = strlen( errorTxt ) + 1;
char *str1 = new char[ dimension ];

// 将错误文本复制到 str1
strcpy( str1, errorTxt );
```

我们也可以用一个在运行时刻才被计算的表达式代替 dimension：

```
// 典型的编程习惯
// 有时会让初学者迷惑
char *str1 = new char[ strlen( errorTxt ) + 1 ];
```

对 strlen()返回的值加 1 是必需的，这样才能容纳 C 风格字符串的结尾空字符。忘记分配这个空字符是个常见错误，并且很难跟踪，因为这样的错误通常是在程序的其他部分读写内存失败时才会表现出来。为什么呢？因为大多数处理 C 风格字符串数组的例程都要遍历数组直到结尾空字符。缺少该空字符常常导致严重的程序错误，因为程序会读写到其他不该读写的内存。我们建议使用 C++标准库 string，这正是避免此类错误的一个原因。

注意，对于用 new 表达式分配的数组，只有第一维可以用运行时刻计算的表达式来指定。其他维必须是在编译时刻已知的常量值。例如：

```
int getDim();

// 分配一个二维数组
int (*pia3)[ 1024 ] = new int[ getDim() ][ 1024 ]; //ok

// 错误：数组的第二维不是常量
int **pia4 = new int[ 4 ][ getDim() ];
```

用来释放数组的 delete 表达式形式如下：

```
delete [] str1;
```

空的方括号是必需的。它告诉编译器，该指针指向空闲存储区中的数组而不是单个对象。因为 str1 类型是 char 型的指针，所以，如果编译器没有看到空方括号对，它就无法判断出要被删除的存储区是否为数组。

如果不小心忘了该空括号对，会怎么样呢？编译器不会捕捉到这样的错误，并且不保证程序会正确执行（当数组的类型有析构函数时，这更加会是真的，如 14.4 节所述）

为避免动态分配数组的内存管理带来的问题，一般建议使用标准库 vector、list 或 string 容器类型，这些类型都会自动管理内存分配。string 类型在 3.4 节介绍，vector 见 3.10 节。容器类型在第 6 章详细讨论。

8.4.4 常量对象的动态分配与释放

程序员可能希望在空闲存储区创建一个对象，但是一旦它被初始化了就要防止程序改变

该对象的值。我们可以使用 new 表达式在空闲存储区内创建一个 const 对象，如下所示：

```
const int *pci = new const int(1024);
```

在空闲存储区创建的 const 对象有一些特殊的属性。首先，const 对象必须被初始化，如果省略了括号中的初始值，就会产生编译错误（除此之外，对于具有缺省构造函数的 class 类型的对象，初始值可以省略）。第二，用 new 表达式返回的值作为初始值的指针必须是一个指向 const 类型的指针。在前面的例子中，pci 是一个指向 const int 的指针类型。它指向由 new 表达式分配的 const int 对象。

对于一个位于空闲存储区内的对象，const 意味着什么呢？它意味着一旦该对象被初始化后，它的值就不能再被改变了。虽然该对象的值不能被修改，但是它的生命期也用 delete 表达式来结束。例如：

```
delete  pci;
```

即使 delete 表达式的操作数是一个指向 const int 的指针，delete 表达式仍然是有效的，并且使 pci 指向的内存被释放。

我们不能在空闲存储区内创建内置类型元素的 const 数组。一个简单的原因是，我们不能初始化用 new 表达式创建的内置类型数组的元素。所有在空闲存储区内被创建的 const 对象都必须被初始化，而且，因为 const 数组不能被初始化（除了类数组），所以试图用 new 表达式创建一个内置类型的 const 数组会导致编译错误：

```
const int *pci = new const int[100]; // 错误
```

8.4.5 定位 new 表达式

new 表达式的第三种形式可以允许程序员要求将对象创建在已经被分配好的内存中。这种形式的 new 表达式被称为定位 new 表达式（placement new expression）。程序员在 new 表达式中指定待创建对象所在的内存地址。new 表达式的形式如下：

```
new (place_address) type -specifier
```

place_address 必须是个指针。为了使用这种形式的 new 表达式，我们必须包含头文件 <new>。这项设施允许程序员预分配大量的内存，供以后通过这种形式的 new 表达式创建对象。例如：

```
#include <iostream>
#include <new>

const int chunk = 16;
class Foo {
public:
     int val() { return _val; }
     Foo() { _val = 0; }
private:
     int _val;
};

// 预分配内存，但没有 Foo 对象
char *buf = new char[ sizeof(Foo) * chunk ];
```

```
int main() {
    // 在 buf 中创建一个 Foo 对象
    Foo *pb = new (buf) Foo;

    // 检查一个对象是否被放在 buf 中
    if ( pb->val() == 0 )
        cout << "new expression worked!" << endl;

    // 到这里不能再使用 pb
    delete[] buf;
    return 0;
}
```

编译并执行该程序，产生下列输出：

```
new expression worked
```

不存在与定位 new 表达式相匹配的 delete 表达式。其实我们并不需要这样的 delete 表达式，因为定位 new 表达式并不分配内存。在前面的例子中，我们删除的不是指针 pb 指向的内存，而是 buf 指向的内存，这是必须的。当程序结尾处不再需要字符缓冲时，buf 指向的内存被删除。因为 buf 指向一个字符数组，所以 delete 表达式形式为：

```
delete [] buf;
```

当字符缓冲被删除时，它所包含的任何对象的生命期也就都结束了。在本例中，pb 不再指向一个有效的 Foo 型的对象。

练习 8.5

说明下列 new 表达式错误的原因。

```
(a) const float *pf = new const float[100];
(b) double *pd = new double[10][getDim()];
(c) int (*pia2)[ 1024 ] = new int[ ][ 1024 ];
(d) const int *pci = new const int;
```

练习 8.6

已知下面的 new 表达式，怎样删除 pa？

```
typedef int arr[10];
int *pa = new arr;
```

练习 8.7

下列 delete 表达式哪些有潜在的运行时刻错误？为什么？

```
int globalObj;
char buf[1000];
void f() {
    int *pi = &globalObj;
    double *pd = 0;
```

```
    float *pf = new float(0);
    int *pa = new(buf)int[20];
    delete pi;    // (a)
    delete pd;    // (b)
    delete pf;    // (c)
    delete[] pa; // (d)
}
```

练习 8.8

下列 auto_ptr 声明哪些是非法的或可能引起后续的程序出现错误？解释原因。

```
int ix = 1024;
int *pi = & ix;
int *pi2 = new int( 2048 );

(a) auto_ptr<int> p0(ix);    (b) auto_ptr<int> p1(pi);
(c) auto_ptr<int> p2(pi2);   (d) auto_ptr<int> p3(&ix);
(e) auto_ptr<int> p4(new int(2048));   (f) auto_ptr<int> p5(p2.get());
(g) auto_ptr<int> p6(p2.release());    (h) auto_ptr<int> p7(p2);
```

练习 8.9

说明下列两条语句的不同之处。

```
int *pi0 = p2.get();
int *pi1 = p2.release();
```

在什么情况下，调用哪一个会更合适？

练习 8.10

假设有下面的语句：

```
auto_ptr< string > ps( new string( "Daniel" ) );
```

下列两个 assign()调用的区别是什么？你认为哪个更合适？为什么？

```
ps.get()->assign( "Danny" );
ps->assign( "Danny" );
```

8.5 名字空间定义 ※

缺省情况下，在**全局域**（也被称作全局名字空间域，global namespace scope）中声明的每个对象、函数、类型或模板都引入了一个**全局实体**（global entity）。在全局名字空间域引入的全局实体必须有惟一的名字。例如，函数和对象不能有相同的名字，无论它们是否在同一程序文本文件中被声明。

这意味着，如果我们希望在程序中使用一个库，那么我们必须保证程序中的全局实体的名字不能与库中的全局实体名字冲突。如果程序是由许多厂商提供的库构成的，那么这将很难保证，各种库会将许多名字引入到全局名字空间域中。在组合不同厂商的库时，我们该怎

样确保程序中的全局实体的名字不会与这些库中声明的全局实体名冲突？名字冲突问题也被称为**全局名字空间污染**（global namespace pollution）问题。

程序员可以通过使全局实体名字很长，或与在程序中的名字前面加个特殊的字符序列前缀，从而避免这些问题。例如：

```
class cplusplus_primer_matrix { ... };
void inverse( cplusplus_primer_matrix & );
```

但是，这种方案不是很理想。用 C++写的程序中可能有相当数目的全局类、函数和模板在整个程序中都是可见的。对程序员来说，用这么长的名字写程序实在是个累赘。

名字空间允许我们更好地处理全局名字空间污染问题。库的作者可以定义一个名字空间，从而把库中的名字隐藏在全局名字空间之外。例如：

```
namespace cplusplus_primer {
    class matrix { /* ... */ };
    void inverse ( matrix & );
}
```

名字空间 cplusplus_primer 是用户声明的名字空间（和全局名字空间不同，后者被隐式声明，并且存在于每个程序之中）。

每个用户声明的名字空间代表一个不同的名字空间域。用户声明的名字空间域可以包含其他嵌套的名字空间定义，以及函数、对象、模板和类型的声明或定义。在一个名字空间内声明的实体被称为**名字空间成员**（namespace member）。与全局名字空间域的情形一样，用户声明的名宇空间中的每个名字必须指向该名字空间内的惟一实体。但是，因为不同的用户声明的名字空间引入了不同的域，所以不同的用户声明的名字空间可以具有相同名字的成员。

名字空间成员的名字会自动地与该名字空间名复合或被其**限定修饰**（qualified）。例如：在名字空间 cplusplus_primer 中声明的 matrix 类的名字是 cplusplus_primer::matrix，inverse 函数的名字是 cplusplusprimer::inverse()。

在程序中我们可以用限定修饰名来使用名字空间 cplusplus_primer 的成员，如下所示：

```
void func( cplusplus_primer::matrix &m )
{
    // ...
    cplusplus_primer::inverse(m);
    return m;
}
```

如果另一个用户声明的名字空间（如 DisneyFeatureAnimation）也提供了一个 matrix 类，而且我们希望使用这个类，而不是在名字空间 cplusplus_primer 中定义的类，则需要修改 func()，如下所示：

```
void func( DisneyFeatureAnimation::matrix &m )
{
    // ...
    DisneyFeatureAnimation::inverse(m);
    return m;
}
```

当然，总是用限定修饰名来引用名字空间成员会比较麻烦：

namespace_name::member_name

因为这个原因，C++提供了一些机制，比如**名字空间别名**（namespace aliase）、**using 声明**（using declaration）、**using 指示符**（using directive），使得在程序中使用名字空间成员更容易一些。我们将在 8.6 节中展示这些机制。

8.5.1 名字空间定义

用户声明的名字空间定义以关键字 namespace 开头，后面是名字空间的名字。该名字在它被定义的域中必须是惟一的。如果在同样的名字空间域中有其他实体与被定义的名字空间同名，就会发生错误。当然，这意味着名字空间定义并没有消除全局名字空间污染问题。但是，使用名字空间大大地缓解了这个问题。

在名字空间名之后是由花括号（{}）括起来的声明块。所有可以出现在全局名字空间域中的声明都可以被放在用户声明的名字空间中：类、变量（带有初始化）、函数（带有定义）以及模板。把一个声明放在用户声明的名字空间中并不会改变其意义。惟一的不同是，这样的声明所引入的名字要与名字空间名复合起来。例如：

```
namespace cplusplus_primer {
    class matrix { /* ... */ };
    void inverse ( matrix & );

    matrix operator+ ( const matrix &m1, const matrix &m2 )
        { /* ... */ }
    const double pi = 3.1416;
}
```

在名字空间 cplusplus_primer 中声明的类的名字是：

```
cplusplus_primer::matrix
```

函数的名字是：

```
cplusplus_primer::inverse()
```

常量的名字是：

```
cplusplus_primer::pi
```

类、函数、常量的名字被声明它的名字空间的名字限定修饰：这些名字被称为**限定修饰名**（qualified name）

名字空间的定义不一定是连续的。例如，可以如下定义前面的名字空间：

```
namespace cplusplus_primer {
    class matrix { /* ... */ };
    const double pi = 3.1416;
}

namespace cplusplus_primer {
    void inverse ( matrix & );
    matrix operator+ ( const matrix &m1, const matrix &m2 )
    { /* ... */ }
}
```

前面两个例子是等价的，它们定义的名字空间 cplusplus_primer 都包含类 matrix、函数

inverse()、常量 pi 以及 operator+()。因此，名字空间的定义是可累积的。

在下面这一行：

```
namespace namespace_name {
```

如果 namespace、name 没有引用前面已经定义过的名字空间，那么它就会定义一个新的名字空间，否则。它将打开原来的名字空间。以便加入新的声明。

名字空间的定义可以非连续，这对生成一个库很有帮助。它使我们更容易将库的源代码组织成接口和实现部分。例如：

```
// 名字空间的这部分定义了库接口
namespace cplusplus_primer {
    class matrix { /* ... */ };
    const double pi = 3.1416;
    matrix operator+ ( const matrix &m1, const matrix &m2 );
    void inverse ( matrix & );
}

// 名字空间的这部分定义了库实现

namespace cplusplus_primer {
    void inverse ( matrix &m )
        { /* ... */ }
    matrix operator+ ( const matrix &m1, const matrix &m2 )
        { /* ... */ }
}
```

该名字空间的第一部分给出了描述库接口的声明和定义：类型定义、常量定义、以及函数声明。该名字空间的第二部分给出了库的详细实现——即，函数定义。

对于组织一个库的源代码帮助更大的是，同一个名字空间的定义可以跨越几个不同的程序文本文件。不同程序文本文件的名字空间定义也可以积累起来。所以，我们的库可以组织成如下：

```
// ---- primer.h ----
namespace cplusplus_primer {
    class matrix { /* ... */ };
    const double pi = 3.1416;
    matrix operator+ ( const matrix &m1, const matrix &m2 );
    void inverse( matrix & );
}

// ---- primer.C ----
#include "primer.h"

namespace cplusplus_primer {
    void inverse( matrix &m )
        { /* ... */ }
    matrix operator+ ( const matrix &m1, const matrix &m2 )
        { /* ... */ }
}
```

使用我们的库的程序可能这样：

```
// ---- user.C ----
// 定义库的接口
#include "primer.h"

void func( cplusplus_primer::matrix &m )
{
    // ...
    cplusplus_primer::inverse( m );
}
```

这种程序组织方式使我们的库具有模块化特性，这种特性是“向用户隐藏实现细节”所必需的，它允许文件 primer.C 和 user.C 被编译链接到一个程序中，而不会有编译错误和链接错误。

8.5.2 域操作符（::）

用户声明的名字空间成员名自动被加上前缀，名字空间名后面加上域操作符（::）。名字空间成员名由该名字空间名进行限定修饰。

使用名字空间成员名，比如 matrix，而不用其名字空间名限定修饰是错误的。编译器不知道名字 matrix 指的是哪个声明：

```
// 定义库接口
#include "primer.h"

// 错误：不能找到 matrix 的声明
void func( matrix &m );
```

名字空间成员的声明被隐藏在其名字空间中。除非我们为编译器指定查找声明的名字空间，否则编译器将在当前的域及嵌套包含当前域的域中查找该名字的声明。例如，如果前面程序改写为

```
// 定义库接口
#include "primer.h"

class matrix { /* 用户定义 */ };

// ok: 找到全局 matrix 类型
void func( matrix &m );
```

则找到全局域中的类 matrix 的定义，该程序能正确编译。因为名字空间成员 matrix 的声明被隐藏在名字空间 cplusplus_primer 中，所以名字空间成员的名字与全局域中声明的类名没有冲突。这就是名字空间能够解决全局名字空间污染问题的原因：名字空间成员的名字不会被找到，除非用户使用域操作符并指定名字空间名字作为前缀。还有其他一些机制可使名字空间成员的声明在其外面也成为可见的。这样的机制被称作 using 声明（using declaration）和 using 指示符（using directive）。我们将在下一节中介绍它们。

注意，域操作符也可以被用来引用全局名字空间的成员。同为全局名字空间没有名字，

所以如下符号：

```
::member_name
```

指向的是全局名字空间的成员。当全局名字空间的成员被嵌套的局部域中声明的名字隐藏时，这对引用该名字非常有用。

下面的例子是一个计算斐波那契序列的函数，设计这个例子的用意是为了说明域操作符怎样被用来引用一个被隐藏的全局名字空间成员。变量 max 有两个定义。全局声明表示该序列的最大值，局部声明表示期望的序列长度（前面曾提到过，函数的参数被放在函数的局部域中。）max 的这两个声明在该函数中都必须被访问到。但是，不加修饰地使用 max 引用的是局部的声明。为访问全局声明，我们必须使用域操作符::max。下面是实现：

```
#include <iostream>
const int max = 65000;
const int lineLength = 12;

void fibonacci( int max )
{
     if ( max < 2 ) return;
          cout << "0 1 ";
     int v1 = 0, v2 = 1, cur;
     for ( int ix = 3; ix <= max; ++ix ) {
          cur = v1 + v2;
          if ( cur > ::max ) break;
          cout << cur << " ";
          v1 = v2;
          v2 = cur;
          if (ix % lineLength == 0) cout << endl;
     }
}
```

下面的 main()函数使用了这个函数：

```
#include <iostream>

void fibonacci( int );
int main() {
     cout << "Fibonacci Series: 16\n";
     fibonacci( 16 );
     return 0;
}
```

编译并执行该程序，产生下面的输出：

```
Fibonacci Series: 16
0 1 1 2 3 5 8 13 21 34 55 89
144 233 377 610
```

8.5.3 嵌套名字空间

前面曾提到过，用户声明的名字空间可以包含嵌套的名字空间。我们可以用嵌套的名字空间来进一步改善库中代码的组织结构。例如：

```
// ---- primer.h ----
```

```
namespace cplusplus_primer {
    // 第一个嵌套域
    // 定义了库的 matrix 部分
    namespace MatrixLib {
        class matrix { /* ... */ };
        const double pi = 3.1416;
        matrix operator+ ( const matrix &m1, const matrix &m2 );
        void inverse( matrix & );
        // ...
    }
    // 第二个嵌套域
    // 定义了库的 Zoology 部分
    namespace AnimalLib {
        class ZooAnimal { /* ... */ };
        class Bear : public ZooAnimal { /* ... */ };
        class Raccoon : public Bear { /* ... */ };
        // ...
    }
}
```

名字空间 cplusplus_primer 包含两个嵌套的名字空间：MatrixLib 和 AnimalLib。

名字空间 cplusplus_primer 可用来防止库中的名字与用户程序中全局名字空间中的名字发生冲突。这个库也被嵌套的域组织成更小的包，将有关的声明和定义分成组。名字空间 MatrixLib 包含 primer 库中的 matrix 部分，而 AnimalLib 含有库中的 ZooAnimal 部分。

嵌套名字空间的成员声明被隐藏在该嵌套域中。这样的成员会被自动地加上最外层名字空间名以及嵌套名字空间名形成的前缀。例如，在嵌套名字空间 MatrixLib 中声明的类的名字是：

```
cplusplus_primer::MatrixLib::matrix
```

函数的名字是：

```
cplusplus_primer::MatrixLib::inverse
```

程序可以按如下方式使用嵌套名字空间 cplusplus_primer::MatrixLib 的成员：

```
#include "primer.h"

// 是的，这很可怕
// 我们很快会引入使名字空间成员更易于使用的机制
void func( cplusplus_primer::MatrixLib::matrix &m )
{
    // ...
    cplusplus_primer::MatrixLib::inverse( m );
}
```

嵌套名字空间是包含它的名字空间中的一个嵌套域。在名字解析期间，嵌套名字空间的行为与嵌套块的类似。例如，当一个名字被用在一个名字空间的定义中时，编译器将会在包含其外的名字空间中查找声明。在下面的例子中，当查找 Type 的声明时，将考虑 Type 被使用之前的声明。在名字空间 MatrixLib 中的声明被首先考虑，然后再考虑名字空间 cplusplus_primer 中的，最后考虑的是全局域中的声明。

```
typedef double Type;
```

```
namespace cplusplus_primer {
    typedef int Type; // 隐藏 ::Type
    namespace MatrixLib {
        int val;

        // Type: 找到 cplusplus_primer 中的声明
        int func(Type t) {
            double val; // 隐藏 MatrixLib::val
            val = ...;
        }
        // ...
    }
}
```

在外围名字空间中声明的实体被嵌套的名字空间中声明的同名实体所隐藏。在前面的例子中，在全局域中的 Type 声明被名字空间 cplusplus_primer 中的 Type 声明隐藏。当在 MatrixLib 名字空间中使用的名字 Type 被解析时，找到在名字空间 cplusplus_primer 中的声明，func()被声明为带一个 int 型的参数。

类似的情况，在名字空间中声明的实体被局部域中声明的实体所隐藏。在上一个例子中，在名字空间 MatrixLib 中的 val 的声明被函数 func()局部域中的 val 声明所隐藏。当解析 func()中用到的名字 val 时，编译器会找到局部域中的声明，在 func()中的赋值是针对这个局部变量的。

8.5.4 名字空间成员定义

我们已经看到，名字空间成员的定义可以出现在名字空间定义内。例如，类 matrix 和常量 pi 是在嵌套的名字空间 MatrixLib 的定义内被定义的，而函数 operator+()和 inverse()的定义则是在程序的后面某个地方给出的：

```
// ---- primer.h ----
namespace cplusplus_primer {
    // 第一个嵌套域
    // 定义了库的 matrix 部分
    namespace MatrixLib {
        class matrix { /* ... */ };
        const double pi = 3.1416;
        matrix operator+ ( const matrix &m1, const matrix &m2 );
        void inverse( matrix & );
        // ...
    }
}
```

我们也可以在名字空间定义之外定义名字空间成员。在这种情况下，名字空间成员的名字必须被外围名字空间名限定修饰。例如，函数 operator+()可以在全局域中如下定义：

```
// ---- primer.C ----
#include "primer.h"

// 全局域定义
cplusplus_primer::MatrixLib::matrix
```

```
cplusplus_primer::MatrixLib::operator+
    ( const matrix& m1, const matrix &m2 )
    { /* ... */ }
```

在该定义中，名字 operator+()由名字空间 cplusplus_primer 和 MatrixLib 的名字限定修饰。但是，看一下 operator+()参数表中的类型 matrix 的用法。所用的名字没有被嵌套的名字空间名 cplusplus_primer::MatrixLib 限定修饰，怎么会这样呢？

operator+()的定义可以使用名字空间成员名的简短形式。这是因为名字空间成员定义是在其名字空间的域内。当 operator+()定义中用到的名字被解析时。编译器会考虑名字空间 MatrixLib 中的成员。但是请注意，返回类型必须被限定修饰，这是因为返回类型不在函数定义的域内。名字空间成员的简短形式只能用在下列成员名之后：

```
cplusplus_primer::MatrixLib::operator+
```

在 operator+()的参数表和函数体中，任何声明或表达式中都可以使用简短形式的名字空间成员名。例如，在 operator+()的局部声明中，我们可以如下创建一个 matrix 类型的对象：

```
// ---- primer.C ----
#include "primer.h"

cplusplus_primer::MatrixLib::matrix
    cplusplus_primer::MatrixLib::operator+
        ( const matrix &m1, const matrix &m2 )
    {
        // 声明一个类型为 cplusplus_primer::MatrixLib::matrix 的局部变量
        matrix res;
        // calculate the sum of two matrix objects
        return res;
    }
```

虽然名字空间成员可以被定义在名字空间定义之外。但是，对于哪些地方可以出现这样的定义还是有限制的。只有包围该成员声明的名字空间[20]才可能包含它的定义。例如，operator+()可在全局域、名字空间 cplusplus_primer 或名字空间 MatrixLib 中定义，而且只有这三种可能。名字空间 cplusplus_primer 中的定义可以是：

```
// ---- primer.C ----
#include "primer.h"

namespace cplusplus_primer {
    MatrixLib::matrix MatrixLib::operator+
        ( const matrix &m1, const matrix &m2 ) { /* ... */ }
}
```

注意，只有当一个名字空间成员在名字空间定义中已经被声明过，它才能在该名字空间定义之外被定义。如果下面的声明没有出现在 primer.h 中，那么刚才给出的 operator+()的定义将是一个错误：

```
namespace cplusplus_primer {
```

[20] 也就是该成员声明所在的名字空间及其外围名字空间。

```
        namespace MatrixLib {
            class matrix { /* ... */ };
            // 下列声明不能被省略
            matrix operator+ ( const matrix &m1, const matrix &m2 );
            // ...
        }
    }
```

8.5.5 ODR 和名字空间成员

正如前面所提到的，名字空间的定义可以是不连续的，可以跨越多个文件。因此，一个名字空间可以在多个文件中被声明。例如：

```
// primer.h
namespace cplusplus_primer {
    // ...
    void inverse( matrix & );
}

// use1.C
#include "primer.h"
// 在 use1.C 中声明 cplusplus_primer::inverse()

// use2.C
#include "primer.h"
// 在 use2.C 中声明 cplusplus_primer::inverse()
```

通过 use1.C 中的头文件 primer.h 声明的成员 cplusplus::inverse()，与 use2.C 中头文件 primer.h 声明的成员 cplusplus::inverse()引用到同一个函数。

虽然名字空间成员名是被限定修饰的，但是名字空间成员也是一个全局实体。还记得 8.2 节讨论的 ODR 要求吗？即非 inline 函数和对象在一个程序中只能被定义一次，这也同样适用于名字空间成员。为了符合这样的要求，使用名字空间的程序一般组织如下。

1．作为名字空间成员的函数和对象的声明被放在头文件中，该文件将被包含在要使用该名字空间的文件中。

```
// ---- primer.h ----
namespace cplusplus_primer {
    class matrix { /* ... */ };
    // 函数声明
    extern matrix operator+ ( const matrix &m1, const matrix &m2 );
    extern void inverse( matrix & );

// 对象声明
    extern bool error_state;
}
```

2．这些成员的定义可以出现在某一个实现文件中。

```
// ---- primer.C ----
#include "primer.h"
namespace cplusplus_primer {
    // 函数声明
```

```
        void inverse( matrix & )
            { /* ... */ }
        matrix operator+ ( const matrix &m1, const matrix &m2 )
            { /* ... */ }
        // 对象声明
        bool error_state = false;
}
```

与全局域中的对象声明一样，我们必须用关键字 extern 来指明只是声明名字空间成员，而不是定义它们。关键字 extern 也可以被用在名字空间成员函数的声明中。但是，如同全局函数的情形一样，在这个例子中，是否使用关键字 extern 是可选的。

8.5.6 未命名的名字空间

我们或许希望所定义的对象、函数、类类型或其他实体，它只在程序的一小段代码中可见。因为这样可以更进一步地缓解名字空间污染问题。因为我们知道该实体只被用在很有限的地方，所以可能不想再花费太多努力来保证这个实体有惟一的名字而不会与程序其他地方声明的名字冲突。当我们在一个函数或嵌套块中声明一个对象时，由该声明引入的名字只在声明它的块中可见。但是，如果程序员想让一个实体被多个函数使用，而又不想让该名字在整个程序中可用，又该怎么办呢？

例如，假设我们想实现一组排序函数，对 double 型 vector 的元素进行排序：

```
// ----- SortLib.h -----
void quickSort( double *, double * );
void bubbleSort( double *, double * );
void mergeSort( double *, double * );
void heapSort( double *, double * );
```

所有函数都使用同一个 swap()函数来交换 vector 中的元素。但是，我们不想让 swap()在整个程序中可见。我们希望保持该函数对于文件 SortLib.C 的局部性，因为只有上面四个函数调用 swap()。下面的代码没有给我们预期的结果。你能看出为什么吗？

```
// ----- SortLib.C -----
void swap( double *d1, double *d2 ) { /* ... */ }
// 只有下面四个函数使用 swap()
void quickSort( double *d1, double *d2 ) { /* ... */ }
void bubbleSort( double *d1, double *d2 ) { /* ... */ }
void mergeSort( double *d1, double *d2 ) { /* ... */ }
void heapSort( double *d1, double *d2 ) { /* ... */ }
```

即使函数 swap()在 SortLib.C 中定义，并且没有在描述排序库的接口的头文件 SortLib.h 中引入，但是，函数 swap()仍然是在全局域中声明的。因此，它是一个全局实体，它的名字不能与任何其他全局实体的名字冲突。

在 C++中，我们可以用**未命名的名字空间**（unnamed namespace）声明一个局部于某一文件的实体。未命名的名字空间以关键字 namespace 开头。同为该名字空间是没有名字的，所以在关键字 namespace 后面没有名字，而在关键字 namespace 后面使用花括号包含声明块。例如：

```
// ----- SortLib.C -----
namespace {
    void swap( double *d1, double *d2 ) { /* ... */ }
}
// 上面四个排序函数的定义
```

函数 swap()只在文件 SortLib.C 中可见。如果另一个文件也含有一个带有函数 swap()定义的未命名名字空间，则该定义引入的是一个不同的函数。函数 swap()存在两种定义但这并不是个错误，因为它们是不同的函数。不像其他名字空间，未命名的名字空间的定义局部于一个特定的文件，不能跨越多个文本文件。

在 SortLib.C 中，在未命名的名字空间的定义之后，我们可以用 swap()的简短格式引用它，没有必要用域操作符引用未命名名字空间的成员。

```
void quickSort( double *d1, double *d2 ) {
    // ...
    double* elem = d1;
    // ...
    // 引用未命名名字空间成员 swap()
    swap( d1, elem );

    // ...
}
```

由于未命名名字空间的成员是程序实体，所以函数 swap()可以在程序整个执行期间被调用。但是，未命名名字空间成员名只在特定的文件中可见，在构成程序的其他文件中是不可见的。

在引入标准 C++名字空间之前，解决此类声明局部化问题的常见方案是使用从 C 语言中继承来的关键字 static。未命名名字空间的成员与被声明为 static 的全局实体具有类似的特性。在 C 中，被声明为 static 的全局实体在声明它的文件之外是不可见的。例如，在 SortLib.C 中的声明可以按如下形式写成 C 程序，它会提供给 swap()相同的特性：

```
// SortLib.C
// swap() 在其他程序中不可见
static void swap( double *d1, double *d2 ) { /* ... */ }
// sort 函数定义同前
```

许多 C++实现都支持全局静态声明，但是，随着越来越多的 C++实现都支持名字空间，全局静态声明的用法将会被未命名的名字空间成员所取代。

练习 8.11

为什么要在程序中定义自己的名字空间？

练习 8.12

假设有下列 operator*()的声明，它是嵌套的名字空间 cplusplus_primer::MatrixLib 的成员：

```
namespace cplusplus_primer {
    namespace MatrixLib {
        class matrix { /* ... */ };
        matrix operator* ( const matrix &, const matrix & );
```

```
            // ...
        }
    }
```

怎样在全局域中定义该操作符？请为该操作符定义提供一个原型。

练习 8.13

说明在程序中使用未命名名字空间的原因。

8.6 使用名字空间成员 ※

总用限定修饰的名字形式 namespace_name::member_name 来引用名字空间成员，毫无疑问是非常麻烦的，尤其是当名字空间名很长的时候。如果不得不一直使用限定修饰名，我们可能会希望创建一些短名字的名字空间，不但因为它们易读，而且因为它们易于键入。但是，使用短的名字空间名会增加与程序中的其他全局名冲突的可能性，所以用长的名字空间名来发行我们的库更为合适一些。

幸运的是，有一些机制能够简化程序中的名字空间成员的用法。名字空间别名、using 声明、using 指示符是帮助我们克服名字空间名使用上的这些不便之处的机制。

8.6.1 名字空间别名

名字空间别名（namespace alias）可以用来把一个较短的同义词与一个名字空间名关联起来。例如，长名字空间名如

```
namespace International_Business_Machines
    { /* ... */ }
```

可以与一个较短的同义词相关联，如下：

```
namespace IBM = International_Business_Machines;
```

名字空间别名的声明以关键字 namespace 开头，后面是一个较短的别名，然后是赋值操作符，最后是原来的名字空间名。如果原来的名字空间名不是一个已知的名字空间名，则会出现错误。

名字空间别名也可以指向一个嵌套的名字空间，还记得早先介绍的 func()的可怕定义吗？如下所示：

```
#include "primer.h"

// 很难读
void func( cplusplus_primer::MatrixLib::matrix &m )
{
    // ...
    cplusplus_primer:: MatrixLib::inverse( m );
}
```

利用名字空间别名也可以引用嵌套的名字空间 cplusplusprimer::MatrixLib，从而使该定义更易读：

```
#include "primer.h"
// 短别名
namespace mlib = cplusplus_primer::MatrixLib;
// 较易读
void func( mlib::matrix &m )
{
    // ...
    mlib::inverse( m );
}
```

一个名字空间可以有许多同义词或别名，且所有别名和原来的名字空间名都可以交替使用。例如，假设别名 Lib 指向名字空间名 cplusplus_primer，则 func()的定义可以重写成下面的形式，它的意义不会改变：

```
// alias 指向名字空间 cplusplus_primer
namespace alias = Lib;
void func( Lib::matrix &m ) {
    // ...
    alias::inverse( m );
}
```

8.6.2 using 声明

通过使名字空间成员的名字可见，来在程序中用该名字的非限定修饰方式引用这个成员，而不用前缀 namespace_name::name。，也是可行的。如果该成员被用 using 指示符声明，那么这就能够做到这一点。

using 声明以关键字 using 开头，后面是名字空间成员名。using 声明中的成员名必须是限定修饰名。例如：

```
namespace cplusplus_primer {
    namespace MatrixLib {
        class matrix { /* ... */ };
        // ...
    }
}

// 名字空间成员 matrix 的 using 声明
using cplusplus_primer::MatrixLib::matrix;
```

using 声明在声明出现的域中引入了一个名字。例如，前面的 using 声明向全局域引入了名字 matrix。在遇到 using 声明之后，在全局域中或其嵌套的域中使用 matrix 都将引用该名字空间成员。例如，假设 using 声明后而又有下面的声明。

```
void func( matrix &m );
```

该声明声明了函数 func()，它有一个参数，类型是 cplusplus_primer::Matrix::matrix。

using 声明同其他声明的行为一样：它有一个域，它引入的名字从该声明开始直到其所在的域结束都是可见的。using 声明可以出现在全局域和任意名字空间中，同时它也可以出现在局部域中。与其他声明一样，using 声明引入的名字有以下特性：

- 它在该域中必须惟一。
- 由外围域中的声明引入的相同名字被其隐藏。
- 它被嵌套域中的相同名字的声明隐藏。

例如：

```
namespace blip {
    int bi = 16, bj = 15, bk = 23;
    // 其他声明
}
int bj = 0;
void manip() {
    using blip::bi;    // 函数 manip() 中的 bi 指向 blip::bi
    ++bi;              // 设置 blip::bi 为 17
    using blip::bj;    // 隐藏全局域中的 bj

                       // 在函数 manip() 中的 bj 指向 blip::bj
    ++bj;              // 设置 blip::bj 为 16
    int bk;            // bk 在局部域中声明
    using blip::bk;    // 错误：在 manip() 中重复定义 bk
}
int wrongInit = bk;    // 错误：bk 在这里不可见，应该用 blip::bk
```

函数 manip()中的 using 声明以简短形式引用名字空间 blip 的成员。using 声明在 manip()函数之外并不可见，用户只能在 manip()函数内部使用这些短名字。在该函数之外，仍然必须使用限定修饰名。

using 声明使名字空间成员易于使用。using 声明一次只能引入一个名字空间成员，它允许我们专门指定在程序中要使用的名字。在特定域中引入 using 声明，使我们可以明确地指定在哪些地方可使用名字空间成员的简短形式。在下一小节，我们将了解怎样一次引入一个名字空间的全部成员名。

8.6.3 using 指示符

名字空间是随标准 C++而引入的。标准 C++之前的实现并不支持名字空间，结果是，标准 C++之前的库也不把全局声明包装在名字空间中。在各种 C++实现支持名字空间之前，人们已经编写了大量重要的 C++代码及其应用程序。如果我们把一个库的内容封装到一个名字空间内，那么我们也就潜在地打破了这些使用旧版本库的旧版应用程序，如果我们把该库的内容包装到一个名字空间中，则该库中的所有名字都变成被限定修饰的，即以该名字空间名加上域操作符作为前缀。而所有以短形式使用该库中的名字的应用程序都不奏效了。

我们可以使用 using 声明使库中的名字变成可见的。例如，假设文件 primer.h 含有该库的新版本，它将全局声明包装到名字空间 cplusplus_primer 中。如果我们想让自己的程序能很快地和新库协同工作，那么就可以用两个 using 声明使名字空间 cplusplus_primer 中的类 matrix 和函数 func()的名字变成可见的。

```
#include "primer.h"
using cplusplus_primer::matrix;
using cplusplus_primer::inverse;

// 因为 using 声明，名字 matrix 和 inverse 可以不加限定修饰地被使用
void func( matrix &m ) {

// ...
inverse( m );
}
```

如果库非常大，且应用程序使用了库中许多的名字，则翻新一个使用名字空间库的新版本就可能需要使用大量的 using 声明。而且所有必需的 using 声明只是允许旧代码能像以前一样编译运行，这样的工作非常乏味，且容易出错。using 指示符可以用来解决这个问题，使得第一次转换到使用名字空间的库版本更加容易。

using 指示符以关键字 using 开头，后面是关键字 namespace，然后是名字空间名。如果该名字没有指向一个前面已经定义的名字空间，则这是一个错误。using 指示符允许我们让来自特定名字空间的所有名字的简短形式都可见。这些成员可以被直接使用，而不要求其名字被限定修饰。例如前面的代码例子可以重写如下：

```
#include "primer.h"

// using 指示符: cplusplus_primer 的所有成员都变成可见的
using namespace cplusplus_primer;

// 名字 matrix 和 inverse 可以不加限定修饰地被使用
void func( matrix &m ) {
    // ...
    inverse( m );
}
```

using 指示符使名字空间成员名可见，就好像它们是在名字空间被定义的地方之外被声明的一样。例如，由于 using 指示符，名字空间 cplusplus_primer 的成员就好像是在全局域中 func() 定义之前声明的一样。using 指示符并没有为名字空间成员的名字声明局部的别名，而是把名字空间的成员转移到包含该名字空间定义的那个域中。比如如下代码：

```
namespace A {
    int i, j;
}
```

对域中有如下 using 声明的代码来说：

```
using namespace A;
```

看起来就像：

```
int i, j;
```

我们来看个例子，它说明了 using 声明的影响（它保留了该名字空间域，但是将成员名与一个局部同义词相关联），以及 using 指示符的影响（其效果相当于去掉了该名字空间）：

```
namespace blip {
    int bi = 16, bj = 15, bk = 23;
    // 其他声明
```

```
}
int bj = 0;

void manip() {
    using namespace blip; // using 指示符 -
                          // ::bj 和 blip::bj 之间的冲突只在 bj 被使用时才被检测到
    ++bi;                 // 设置 blip::bi 为 17
    ++bj;                 // 错误: 二义性
                          // 全局 bj 还是 blip::bj?
    ++::bj;               // ok: 设置全局 bj 为 1
    ++blip::bj;           // ok: 设置 blip::bj 为 16
    int bk = 97;          // 局部 bk 隐藏 blip::bk
    ++bk;                 // 设置局部 bk 为 98
}
```

应该注意的第一个问题是 using 指示符是域内的。在 manip()中的 using 指示符只能应用在函数 manip()的块内。对函数 manip()来说，名字空间 blip 的成员就好像是在全局域中声明的一样。所以，函数 manip()可以以简短形式引用这些成员的名字。在函数 manip()之外的代码必须使用限定修饰名。

要注意的第二个问题是，由 using 指示符引起的二义性错误是在该名字被使用时才被检测到，而不是在遇到 using 指示符时。例如，成员 bj 在 manip()中出现，就好像它是在名字空间 blip 被定义的地方之外（即全局域中）被声明的一样。然而，在全局域中已经有一个名为 bj 的变量。因此，在函数 manip()中使用 bj 有二义性，该名字同时引用全局变量和名字空间 blio 的成员。但是，using 指示符并没有错，只有当 manip()函数用到 bj 时，编译器才会检测到二义性错误。如果 bj 在 manip()中没有被用到，则不会产生错误。

要注意的第三个问题是，使用限定修饰名不受 using 指示符的影响。当 manip()引用::bj 时，只有全局域中的变量才被考虑，当 manip()引用 blip::bj 时，只考虑名字空间 blip 引入的变量。

要注意的最后一个问题是，因为名字空间成员就好像是在“该名字空间定义所在的地方（at the location where the namespace definition is located）”之外被声明的一样，所以出现在 manip()中的成员就好像是在全局域中被定义的一样。这意味着，manip()中的局部声明可以隐藏某些名字空间成员名。局部变量 bk 隐藏了名字空间成员 blip::bk。在 manip()中引用 bk 没有二义性，它引用的是局部变量 bk。

using 指示符用起来很简单：只需要使用一个 using 指示符，所有的名字空间成员一下子就都可见了。尽管这可以看作是一个简单的解决方案，仍是，过多地使用 using 指示符可能会引入其自身的问题。如果一个应用使用了许多库，且这些库中的名字都用 using 指示符变为可见，则我们可能又回到了原来的问题：全局名字空间污染问题。例如：

```
namespace cplusplus_primer {
    class matrix { };
    // 其他省略
}

namespace DisneyFeatureAnimation {
```

```
        class matrix { };
        // 省略
}

using namespace cplus plus_primer;
using namespace DisneyFeatureAnimation;
matrix m; // 错误：二义性
// cplusplus_primer 的还是 DisneyFeatureAnimation 的?
```

由多个 using 指示符引起的二义性错误只能在使用点上被检测到。在前面的例子中，二义性错误只在 matrix 被使用时才被检测到。这种迟到的检测可能会使用户吃惊，即使头文件没有被改变，且没有新的声明被加入到程序中，以后仍然可能会出现错误：该错误经常出现在我们突然决定要使用库中的新特性的时候。

当我们把一个应用程序移植到一个包装在名字空间中的新库版本时，using 指示符非常有用。但是使用多个 using 指示符会引起全局名字空间污染问题。用多个选择性的 using 声明来代替 using 指示符会使这个问题最小化，由多个选择性的 using 声明引起的二义性错误在声明点就能被检测到。因此建议使用 using 声明而不是 using 指示符，以便更好地控制程序中的全局名字空间污染问题。

8.6.4 标准名字空间 std

标准 C++库中的所有组件都是在一个被称为 std 的名字空间中声明和定义的。在标准头文件（如<vector>或<iostream>）中声明的函数、对象和类模板，都被声明在名字空间 std 中。

如果所有的库组件都在名字空间 std 中被声明，那么下面这个来自 6.5 节中的例子所有的库组件的名字又会有什么错误？

```
#include <vector>
#include <string>
#include <iterator>
int main()
{
        // 与标准输出绑定的输入流迭代器
        istream_iterator<string> infile( cin );

        // 标记了 "流结束" 的输入流迭代器
        istream_iterator<string> eos;

        // 用 cin 输入的值初始化 svec
        vector<string> svec( infile, eos );
        // 处理 svec
}
```

对，代码没有通过编译，因为在上面的代码中，名字空间 std 的成员不能被不加限定修饰地访问。为了修正这个错误，我们可以选择下列方案之一：

- 用适当的限定修饰名代替例子中的名字空间 std 成员的名字。
- 用 using 声明使例子中用到的名字空间 std 的成员可见。
- 用 using 指示符使来自名字空间 std 的全部成员可见。

在例子中用到的名字空间 std 的成员有：类模板 iostream_iterator、程序的标准输入 cin、类 string，以及类模板 vector。

最简单的解决办案是在#include 指示符后面加上 using 指示符如下：

```
using namespace std;
```

该 using 指示符使名字空间 std 中的全部成员在例子中都可见。但是，在名字空间 std 中有太多的声明，我们更喜欢用 using 声明来减少“当我们向程序中增加新的全局声明时发生名字冲突的可能性”。

为使程序通过编译，我们只需下列 using 声明：

```
using std::istream_iterator;
using std::string;
using std::cin;
using std::vector;
```

但是，应该把它放在哪儿呢？如果程序是由许多文件构成的，则创建一个头文件，使它包含该应用程序所需的名字空间 std 成员的全部 using 声明，这样比较方便。该头文件将被包含在程序文本文件中的 C++标准库头文件之后。

在本书中，为使代码例子简短，且因为许多例子程序都是在不支持名字空间的编译器中被编译的，所以我们并没有显式地列出需要编译该例子的 using 声明，只是假设在代码例子中都已经提供了所用到的名字空间 std 成员的 using 声明。

练习 8.14

解释 using 声明和 using 指示符的区别。

练习 8.15

根据 6.14 节给出的完整例子，写出使名字空间 std 的成员在例子中可见所需要的 using 声明。

练习 8.16

考虑下面的代码例子：

```
namespace Exercise {
    int ivar = 0;
    double dvar = 0;
    const int limit = 1000;
}
int ivar = 0;

//1
void manip() {
    //2

    double dvar = 3.1416;
    int iobj = limit + 1;
```

```
        ++ivar;
        ++::ivar;
    }
```

如果将名字空间 Exercise 成员的 using 声明放在 //1 处，那么会对代码中的声明和表达式有什么样的影响？如果放在 //2 处呢？当用 using 指示符代替名字空间 Exercise 的 using 声明时，答案又是什么？

9

重 载 函 数

我们已经知道怎样声明和定义函数，以及怎样在程序中使用函数，在本章中我们将了解 C++支持的一种特殊函数：重载函数。如果两个函数名字相同，并且在相同的域中被声明，但是参数表不同，则它们就是重载函数（overloaded function）。在本章中，我们将首先了解怎样声明一组重载函数，以及这样做的好处。然后，再看看函数重载解析过程是怎样进行的——即，一个函数调用怎样被解析为一组重载函数中的某一个函数。函数重载解析过程是 C++中最复杂的内容之一。本章的结尾将为那些希望进一步详细了解重载函数的人提供了两个小节的高级主题。它们将更完整地描述参数类型转换和函数重载的解析。

9.1 重载函数声明

我们已经知道怎样声明和定义函数，以及怎样在程序中使用函数，现在我们将了解 C++支持的另一种函数新特性：重载函数。**函数重载**（function overloading）允许多个函数共享同一个函数名，但是针对不同参数类型提供共同的操作。

如果你曾经用一种程序设计语言写过算术表达式，那么你就已经使用过预定义的重载函数。例如，如下表达式：

```
1 + 3
```

调用了针对整数操作数的加法操作，而表达式：

```
1.0 + 3.0
```

调用了另外一个专门处理浮点操作数的不同的加法操作。实际被使用的操作对用户而言是透明的。加法操作被重载，以便处理不同的操作数类型。根据操作数的类型来区分不同的操作并应用适当的操作，是编译器的责任，而不是程序员的事情。

本章我们将了解怎样定义自己的重载函数。

9.1.1 为什么要重载一个函数名

正如内置加法操作的情形一样，我们可能希望定义一组函数，它们执行同样的一般性动

作，但是应用在不同的参数类型上。例如，假设我们希望定义一个函数，它返回参数中的最大值。

如果没有重载一个函数名的能力，那么我们就必须为每个函数给出一个惟一的名字。例如，我们可能如下定义一组 max()函数：

```
int i_max( int, int );
int vi_max( const vector<int> & );
int matrix_max( const matrix & );
```

但是，这些函数都执行了相同的一般性动作；都返回参数集合中的最大值。从用户的角度来看，只有一种操作，就是判断最大值。至于怎样完成其细节，函数的用户一点也不关心。

这种词汇上的复杂性不是“判断一组数中最大值”问题本身固有的，而是反映了程序设计环境的一种局限性：在同一个域中出现的名字必须指向一个唯实体（惟一的对象、函数、class 类型等等）。这种复杂性给程序员带来了一个实际问题，他们必须记住或查找每一个名字。函数重载把程序员从这种词汇复杂性中解放出来。

通过函数重载，程序员可以简单地这样写：

```
int ix = max( j, k );
vector<int> vec;
// ...
int iy = max( vec );
```

这项技术可以获得各种条件下的最大值。

9.1.2 怎样重载一个函数名

在 C++中，可以为两个或多个函数提供相同的名字，只要它们的每个参数表惟一就行：或者是参数的个数不同，或者是参数类型不同。下面是重载函数 max()的声明：

```
int max( int, int );
int max( const vector<int> & );
int max( const matrix & );
```

参数集惟一的每个重载声明都要求一个独立的 max()定义。

当一个函数名在一个特殊的域中被声明多次时，编译器按如下步骤解释第二个（以及后续的）的声明。

- 如果两个函数的参数表中参数的个数或类型不同，则认为这两个函数是重载的。例如：

```
// 重载函数
void print( const string & );
void print( vector<int> & );
```

- 如果两个函数的返回类型和参数表精确匹配，则第二个声明被视为第一个的重复声明。例如：

```
// 声明同一个函数
void print( const string &str );
void print( cons t string & );
```

参数表的比较过程与参数名无关。

- 如果两个函数的参数表相同，但是返回类型不同，则第一个声明被视为第一个的错误重复声明，会被标记为编译错误。例如：

```
unsigned int max( int i1, int i2 );
int max( int , int );  // 错误：只有返回类型不同
```

函数的返回类型不足以区分两个重载函数。

- 如果在两个函数的参数表中，只有缺省实参不同，则第二个声明被视为第一个的重复声明。例如：

```
// 声明同一函数
int max( int *ia, int sz );
int max( int *, int = 10 );
```

typedef 名为现有的数据类型提供了一个替换名：它并没有创建一个新类型。因此，如果两个函数参数表的区别只在于一个使用了 typedef，而另一个使用了与 typedef 相应的类型。则该参数表不被视为不同的。下列 calc()的两个函数声明被视为具有相同的参数表。第二个声明导致编译时刻错误，因为虽然它声明了相同的参数表，但是它声明了与第一个不同的返回类型。

```
// typedef 并不引入一个新类型
typedef double DOLLAR;

// 错误：相同参数表，不同返回类型
extern DOLLAR calc( DOLLAR );
extern int calc( double );
```

当一个参数类型是 const 或 volatile 时，在识别函数声明是否相同时，并不考虑 const 和 volatile 修饰符。例如，下列两个声明声明了同一个函数：

```
// 声明同一函数
void f( int );
void f( const int );
```

参数是 const，这只跟函数的定义有关系：它意味着，函数体内的表达式不能改变参数的值。但是，对于按值传递的参数，这对函数的用户是完全透明的：用户不会看到函数对按值传递的实参的改变。（按值传递的实参以及参数的其他传递方式在 7.3 节中讨论。），当实参被按值传递时，将参数声明为 const 不会改变可以被传递给该函数的实参种类。任何 int 型的实参都可以被用来调用函数 f(const int)。因为两个函数接受相同的实参集，所以刚才给出的两个声明并没有声明一个重载函数。函数 f()可以被定义为：

```
void f( int i ) { }
```

或：

```
void f( const int i ) { }
```

然而。在同一个程序中同时提供这两个定义将产生错误，因为这些定义把一个函数定义了两次。

但是，如果把 const 或 volatile 应用在指针或引用参数指向的类型上，则在判断函数声明

是否相同时，就要考虑 const 和 volatile 修饰符。

```
// 声明了不同的函数
void f( int* );
void f( const int* );

// 也声明了不同的函数
void f( int& );
void f( const int& );
```

9.1.3 何时不重载一个函数名

什么时候重载一个函数名没有好处？如果不同的函数名所提供的信息可使程序更易于理解的话，则再用重载函数就没有什么好处了。下面是一个例子，下列函数集合在一个公共数据抽象上进行操作，它们可能首先会被看作重载的对象：

```
void setDate( Date&, int, int, int );
Date &convertDate( const string & );
void printDate( const Date& );
```

这些函数在同一个数据类型（类 Date）上执行操作，但是并不共享同样的操作。在这种情况下，与函数名相关的词汇复杂性来自于程序员的习惯，他用这一组操作集和公共数据类型来命名函数。C++的类机制使得这种习惯变得不再必要。相反，这些函数应该成为类 Date 的成员，因为每个成员函数执行不同的操作，所以成员函数的名字应该表示它的操作。例如：

```
#include <string>
class Date {
public:
     set( int, int, int );
     Date &convert( const string & );
     void print();

     // ...
};
```

下面是另外一个例子，下列 Screen 类的五个成员函数在 Screen 的光标上执行各种移动操作，或许我们首先认为最好把这些函数以名字 move()重载：

```
Screen& moveHome();
Screen& moveAbs( int, int );
Screen& moveRel( int, int, char *direction );

Screen& moveX( int );
Screen& moveY( int );
```

最后两个实例并不能被重载，因为它们的参数表完全相同。为了提供一个惟一的标识，我们把两个函数如下压缩成一个：

```
// moveX() 和 moveY() 组合后的函数
Screen& move( int, char xy );
```

现在，每个函数都有了一个惟一的参数表，这样就能够用名字 move()重载该函数集合。

但是，根据我们的准则，重载函数是个坏主意：不同的函数名所提供的信息会被丢失，这使程序更难于理解。尽管光标移动是所有这些函数共享的通用操作，但是，这些函数之间移动的特性是惟一的。例如，moveHome()代表了光标移动的一个特殊实例。对程序的读者来说下面两个调用哪一个更易于理解？对 Screen 类的用户来说下面两个调用哪个更容易记忆？

```
// 哪一个更易于理解?
myScreen.home();  // 我们认为是这个
myScreen.move();
```

有时候，没有必要重载，可能也不需要不同的函数定义。在某些情况下，缺省实参可以把多个函数声明压缩为一个函数中。例如，两个光标函数：

```
moveAbs(int,int);
moveAbs(int,int,char*);
```

可以通过第三个 char*型参数的有无来区分。如果这两个函数的实现十分类似，并且在向函数传递参数时，如果能够找到一个 char*型缺省实参可以表示实参不存在时的意义，则这两个函数就可以被合并。现在，正好有个这样的缺省实参——值为 0 的指针：

```
move( int, int, char* = 0 );
```

程序员最好抱这样的观点：并不是每个语言特性都是你要攀登的下一座山峰。使用语言的特性应该遵从应用的逻辑，而不是简单地因为它的存在就必须要使用它。程序员不应该勉强使用重载函数。只有在必要的地方使用它们，才会让人感觉自然。

9.1.4 重载与域 ※

重载函数集合中的全部函数都应在同一个域中声明。例如，一个声明为局部的函数将隐藏而不是重载一个全局域中声明的函数。例如：

```
#include <string>
void print( const string & );
void print( double ); // overloads print()
void fooBar( int ival )
{
    // 独立的域：隐藏 print()的两个实例
    extern void print( int );

    // 错误：print( const string & )在这个域中被隐藏
    print( "Value : " );
    print( ival ); // ok: print( int ) 可见
}
```

我们也可以在一个类中声明一组重载函数。因为每个类都维持着自己的一个域，所以两个不同类的成员函数不能相互重载。类成员函数将在第 13 章描述，而类成员函数的重载解析将在第 15 章描述。

我们也可以在一个名字空间内声明一组重载函数。每个名字空间也都维持着自己的一个域，作为不同名字空间成员的函数不能相互重载。例如：

```
#include <string>
namespace IBM {
```

```
        extern void print( const string < );
        extern void print( double );  // 重载 print()
    }

    namespace Disney {
        // 独立的域:
        // 没有重载 IBM 的 print()
        extern void print( int );
    }
```

using 声明和 using 指示符可以使一个名字空间的成员在另一个中可见，这些机制对于重载函数的声明有一些影响。关于 using 声明和 using 指示符在 8.6 节介绍。

using 声明怎样影响重载函数呢？using 声明为一个名字空间的成员在该声明出现的域中提供了一个别名。下面程序中的 using 声明会怎么样呢？

```
    namespace libs_R_us {
        int max( int, int );
        int max( double, double );
        extern void print( int );
        extern void print( double );
    }
    // using 声明
    using libs_R_us::max;
    using libs_R_us::print( double );  // 错误
    void func()
    {
        max( 87, 65 );          // 调用 libs_R_us::max( int, int )
        max( 35.5, 76.6 );      // 调用 libs_R_us::max( double, double )
    }
```

第一个 using 声明向全局域中引入了两个 libs_R_us::max()函数。于是，我们便可以在 func()中调用这两个 max()函数。函数调用时的实参类型将决定哪个函数会被调用。第二个 using 声明是个错误。用户不能在 using 声明中为一个函数指定参数表。对于 libs_R_us::pring()惟一有效的 using 声明是：

```
    using libs_R_us::pring;
```

using 声明总是为重载函数集合的所有函数声明别名。为什么这个限制是有必要的呢？这个限制可以确保名字空间 libs_R_us 的接口不会被破坏。很清楚，对如下的函数调用：

```
    print( 88 );
```

名字空间的作者希望调用函数 libs_R_us::pring(int)。由于某种原因，库的作者给出了几个不同的函数。若允许用户有选择地把一组重载函数中的一个函数、而不是全部函数加入到一个域中，那么这将导致令人吃惊的程序行为。

如果 using 声明向一个域中引入了一个函数，而该域中已经存在一个同名的函数，又会怎样呢？记住，using 声明只是一个声明。由 using 声明引入的函数就好像在该声明出现的地方被声明一样。因此，由 using 声明引入的函数重载了在该声明所出现的域中同名函数的其他声明。例如：

```
#include <string>
namespace libs_R_us {
    extern void print( int );
    extern void print( double );
}
extern void print( const string & );

// libs_R_us::print( int ) 和 libs_R_us::print( double )
// 重载 print( const string & )
using libs_R_us::print;

void fooBar( int ival )
{
    print( "Value: " ); // 调用全局 print( const string & )
    print( ival );    // 调用 libs_R_us::print( int )
}
```

using 声明向全局域中加入了两个声明：一个是 print(int)，一个是 print(double)。这些声明为名字空间 libs_R_us 中的函数提供了别名。这些声明被加入到 print()的重载函数集合中，它已经包含了全局函数 print(const string&)。当 fooBar()调用函数时，所有的 print()函数都将被考虑。

如果 using 声明向一个域中引入了一个函数，而该域中已经有同名函数且具有相同的参数表，则该 using 声明就是错误的。如果在全局域中已经存在一个名为 print(int)的函数，则 using 声明不能为名字空间 libs_R_us 中的函数声明别名 print(int)。例如：

```
namespace libs_R_us {
    void print( int );
    void print( double );
}

void print( int );
using libs_R_us::print;  // 错误: print(int) 的重复声明
void fooBar( int ival )
{
    print( ival );    // 哪一个 print? ::print 还是 libs_R_us::print?
}
```

我们已经知道了 using 声明是怎样影响重载函数的，现在让我们来了解一下 using 指示符又是怎样影响重载函数的。using 指示符使名字空间成员就像在名字空间之外被声明的一样。通过去掉名字空间的边界，using 指示符把所有声明加入到当前名字空间被定义的域中。如果在当前域中声明的函数与某个名字空间成员函数名字相同，则该名字空间成员函数被加入到重载函数集合中。例如：

```
#include <string>

namespace libs_R_us {
    extern void print( int );
    extern void print( double );
}
```

```
extern void print( const string & );

// using 指示符:
// print(int), print(double) 和 print(const string &)
// 是重载函数集的一部分
using namespace libs_R_us;
void fooBar( int ival )
{
     print( "Value: " ); // 调用 global print(const string &)
     print( ival ); // 调用 libs_R_us::print(int)
}
```

如果使用多个 using 指示符，情况也是这样。具有相同的名字、但是来自不同名字空间的成员函数都将被加到同一重载函数集合中。例如：

```
namespace IBM {
     int print(int);
}
namespace Disney {
     double print(double);
}

// using 指示符:
// 从不同的名字空间形成函数的重载集合
using namespace IBM;
using namespace Disney;
long double print(long double);
int main() {
     print(1);       // 调用 IBM::print(int)
     print(3.1);     // 调用 Disney::print(double)
     return 0;
}
```

在全局域中的函数 print()的重载集合含有函数 print(double)、print(int)以及 print(long double)。这些函数是 main()中调用该函数时需要被考虑的重载函数集，尽管这些函数最初是在不同的名字空间域中被声明的。

因此，同一重载函数集合中的函数都是在同一个域中被声明的，即使这些声明可能是用“使名字空间成员好像在其他域中声明的一样可见的 using 声明或 using 指示符”引入的。

9.1.5 extern "c" 和重载函数 ※

如 7.7 节所示，我们可以用链接指示符 extern "C"，来表示 C++程序中的某一个函数是用程序设计语言 C 编写的。链接指示符 extern "C"对重载函数声明的影响又会怎样呢？重载函数集合中的某些函数可以是 C++函数，而另外一些是 C 函数吗？

链接指示符只能指定重载函数集中的一个函数，例如，包含下列两个声明的程序是非法的：

```
// 错误: 在一个重载函数集中有两个 extern "C" 函数
extern "C" void print( const char* );
extern "C" void print( int );
```

下面 calc()的重载说明了在一个重载函数集合上的典型的链接指示符的用法：

```
class SmallInt { /* ... */ };
class BigNum { /* ... */ };

// 这个 C 函数可以在 C 和 C++程序中调用
// C++函数可以处理 C++类参数
extern "C" double calc( double );
extern SmallInt calc( const SmallInt& );
extern BigNum calc( const BigNum& );
```

C 语言的 calc()函数可以被 C 程序调用，也可以被 C++程序调用。其他函数是 C++函数，它们含有类参数，只能在 C++程序中被调用。声明的顺序并不重要。

链接指示符并不影响函数调用时对于函数的选择：只用参数类型来选择将被调用的函数。被选中的函数是与实参类型精确匹配的那个。例如：

```
SmallInt si = 8;
int main() {
    calc( 34 ); // 调用 C 写的 calc( double )
    calc( si ); // 调用 C++ 写的 calc( const SmallInt & )

    // ...
    return 0;
}
```

9.1.6 指向重载函数的指针 ※

我们可以声明一个指向重载函数集合里的某一个函数的指针。怎样做呢？例如：

```
extern void ff( vector<double> );
extern void ff( unsigned int );

// pf1 指向哪个函数?
void ( *pf1 )( unsigned int ) = &ff;
```

因为 ff()是一个重载函数，所以只看初始化表达式&ff，编译器并不知道该选择哪个函数。为选择初始化该指针的函数，编译器要查找重载函数集合里与指针指向的函数类型只有相同的返回类型和参数表的函数。在上个例子中，选择的是 ff(unsigned int)。

如果没有函数与指针类型匹配，又该怎么办？如果是这样，将导致编译错误。例如：

```
extern void ff( vector<double> );
extern void ff( unsigned int );

// 错误：无匹配：无效参数表
void ( *pf2 )( int ) = &ff;

// 错误：无匹配：无效返同类型
double ( *pf3 )( vector<double> ) = &ff;
```

赋值的工作方式类似。如果一个重载函数的地址被赋值给一个函数指针，则该函数指针的类型被用来选择赋值符号右边的函数。如果编译器没有找到与指针类型匹配的函数，则赋值就是错误的，也就是说，在两个函数指针类型之间不能进行类型转换：

```
matrix calc( const matrix & );
```

```
int calc( int, int );
int ( *pc1 )( int, int ) = 0;
int ( *pc2 )( int, double ) = 0;

// ...
// ok: 匹配 int calc( int, int );
pc1 = &calc;

// 错误：无匹配： 无效的第二个参数类型
pc2 = &calc;
```

9.1.7 类型安全链接 ※

重载允许同一个函数名以不同参数表出现多次，这是程序源代码层次上的词法便利。但是，大多数编译系统的底层组件要求每个函数名必须惟一。这是因为大多数链接编辑器都是按照函数名来解析外部引用的。如果链接编辑器看到两个以上的名为 print 的实例，它就不能通过分析类型来区分不同的实体（在编译到这一点时，类型信息通常已经不存在了）。链接编辑器会标记 print 被定义多次，并退出。

为处理这个问题，每个函数名及其相关参数表都被作为一个惟一的内部名**编码**（encoded）。编译系统的底层组件只能看到编码后的名字。名字转换的细节并不重要：在不同的编译器实现中，它们可能不同。一般的做法是把参数的个数和类型都进行编码，然后再将其附在函数名后面。

正如在 8.2 节关于全局函数的介绍中我们所看到的，这种特殊的编码可确保同名函数的两个声明（它们有不同的参数表，处于不同的文件中）不会被链接编辑器当作同一个函数的声明。因为这种编码帮助链接阶段区分程序中的重载函数，所以我们把它称作**类型安全链接**（type-safe linkage）。

这种特殊编码不适用于用链接指示符 extern "C"声明的函数。这就是为什么在重载函数集合中只有一个函数可以被声明为 extern "C"的原因，具有不同的参数表的两个 extern "C" 的函数会被链接编辑器视为同一函数。

练习 9.1

为什么我们要声明重载函数？

练习 9.2

应该怎样声明下面 error()函数的重载函数集合以处理下列调用？

```
int index;
int upperBound;
char selectVal;

// ...
error( "Array out of bounds: ", index, upperBound );
error( "Division by zero" );
error( "Invalid selection", selectVal );
```

练习 9.3

说出下列声明集合中第二个声明所造成的影响。

```
(a)  int calc( int, int );
     int calc( const int, const int );
(b)  int get();
     double get();
(c)  int *reset( int * );
     double *reset( double * );
(d)  extern "C" int compute( int *, int );
     extern "C" double compute( double *, double );
```

练习 9.4

下列哪些初始化是错误的？为什么？

```
(a)  void reset( int * );
     void (*pf)( void * ) = reset;
(b)  int calc( int, int );
     int (*pf1)( int, int ) = calc;
(c)  extern "C" int compute( int *, int );
     int (*pf3)( int*, int ) = compute;
(d)  void (*pf4)( const matrix & ) = 0;
```

9.2 重载解析的三个步骤

函数重载解析（function overload resolution）是把函数调用与重载函数集合中的一个函数相关联的过程。在存在多个同名函数的情况下，根据函数调用中指定的实参选择其中一个函数。考虑下面的例子：

```
T t1, t2;
void f( int, int );
void f( float, float );
int main() {
     f( t1,t2 );
     return 0;
}
```

这里，根据给出的类型 T，函数重载解析过程将决定 f(t1,t2)调用的是 f(int, int)还是 f(float, float)，还要决定是因为用实参 t1 和 t2 不能调用任何一个函数，还是由于调用中指定的实参与两个函数都精确匹配引起了**二义性**（ambiguous），使调用出错了。

函数重载解析过程是 C++程序设计语言中最复杂的部分之一。C++初学者在开始时可能会被它的全部细节吓倒。因此，本节只大概地浏览一下重载函数解析的过程，使你对发生的

事情有个感性认识。希望进一步了解的读者将在下两节中看到有关函数重载解析的更详细地描述。

函数重载解析的过程有三个步骤，我们将用下面的例子解释这三步：

```
void f(); void f( int );
void f( double, double = 3.4 );
void f( char*, char* );
int main() {
    f( 5.6 );
    return 0;
}
```

函数重载解析的步骤如下：

1．确定函数调用考虑的重载函数的集合，确定函数调用中实参表的属性。

2．从重载函数集合中选择函数，该函数可以在（给出实参个数和类型）的情况下用调用中指定的实参进行调用。

3．选择与调用最匹配的函数。

下面我们将按顺序查看每一步。

函数重载解析的第一步是确定对该调用所考虑的重载函数集合。该集合中的函数被称为**候选函数**（candidate function）。候选函数是与被调用函数同名的函数，并且在调用点上，它的声明可见。

在这个例子中，有四个候选函数：f()、f(int)、f(double, double)以及 f(char*, char*)。

函数重载解析的第一步还要确定函数调用中的参数表的属性，即实参的数目和类型。在本例中。实参表由一个 double 型的实参构成。

函数重载解析的第二步是从第一步找到的候选函数中选择一个或多个函数，它们能够用该调用中指定的实参来调用。因此，选出来的函数被称为**可行函数**（viable function）。可行函数的参数个数与调用的实参表中的参数数目相同，或者可行函数的参数个数多一些，但是每个多出来的参数都要有相关的缺省实参。对于每个可行函数，调用中的实参与该函数的对应的参数类型之间必须存在**转换**（conversion）。

在这个例子中，有两个可行函数，它们能够用调用中指定的实参表进行调用。

- f(int)是一个可行函数，因为它只有一个参数而且存在从实参类型 double 到参数类型 int 之间的转换。
- f(double, double)也是一个可行函数，因为它的第二个参数给出了缺省值，而第一个参数类型是 double，与实参类型精确匹配。

如果函数重载解析过程的第二步没有找到可以用给定的实参表调用的可行函数，则该调用就是错误的。没有函数与调用匹配，则说是**无匹配情况**（no match situation）。

函数重载解析的第三步选择与调用最匹配的函数，该函数被称为**最佳可行函数**（best viable function）[通常也称为**最佳匹配函数**（best match function）]。为了选择这个函数，从实参类型到相应可行函数参数所用的转换都被划分**等级**（ranked）。最佳可行函数是被适用于如下规则的函数：

1．应用在实参上的转换不比调用其他可行函数所需的转换差。

2．在某些实参上的转换要比其他可行函数对该参数的转换好。

类型转换及其等级划分将在 9.3 节详细讨论。这里我们只简要地查看一下本例子中转换的等级。当考虑可行函数 f(int)时，应用的转换是个标准转换，它将 double 型的实参转换成 int 型。当考虑可行函数 f(double)时，实参的类型 double 与相应的参数精确匹配。因为精确匹配比标准转换好（不做转换比任何转换都好），所以该调用的最佳可行函数是 f(double, double)。

如果函数重载解析的第三步没有找到最佳可行函数，则该函数调用是有二义的，即没有找到一个比其他可行函数都好的函数。

有关函数重载解析步骤的详细情况可在 9.4 节中找到，当一个重载的类成员函数被调用时，或一个重载的操作符函数被调用时函数重载解析也是适用的。15.10 节将讨论类成员函数重载解析的规则，而 15.11 节将讨论重载操作符的函数重载解析规则。函数重载解析过程还必须考虑函数模板生成的函数，10.8 节将讨论函数模板如何影响函数重载解析过程。

练习 9.5

函数重载解析过程的最后一步（第三步）发生的是什么？

9.3 参数类型转换 ※

在函数重载解析的第二步中，编译器确定“可以应用在函数调用的实参上的、将其转换成每个可行函数中相应参数类型”的转换，并将其划分等级。这种等级有二种可能：

1．**精确匹配**（exact match）：实参与函数参数的类型精确匹配。例如，给出重载函数集中的下列三个 printo 函数，则后面三个 print()调用都导致精确匹配：

```
void print( unsigned int );
void print( const char* );
void print( char );

unsigned int a;
print( 'a' ); // 匹配 print( char );
print( "a" ); // 匹配 print( const char* );
print( a );   // 匹配 print( unsigned int );
```

2．与一个**类型转换**（type conversion）匹配。实参不直接与参数类型匹配，但是它能转换成这样的类型：

```
void ff( char );
ff( 0 ); // 从 int 到 char 转换实参
```

3．**无匹配**（no match）：实参不能与声明的函数的参数匹配，因为在实参与相应的函数参数之间无法进行类型转换。下列两个 print()调用导致无匹配：

```
// print() 声明如下
int *ip;
class SmallInt { /* ... */ };
SmallInt si;
```

```
print( ip );  // 错误：无匹配
print( si );  // 错误：无匹配
```

精确匹配的实参并不一定与参数的类型完全一致，有一些最小转换可以被应用到实参上。在精确匹配的等级类别中可能存在的转换如下：

- 从左值到右值的转换
- 从数组到指针的转换
- 从函数到指针的转换
- 限定修饰转换

我们会在后面更详细地介绍这些转换。

“与一个类型转换匹配”的等级类别是三个等级中最复杂的一个，几种类型转换都必须考虑到。可能的转换被分成三组：**提升**（promotion）、**标准转换**（standard conversion）和**用户定义的转换**（user-defined conversions）。提升和标准转换在本节后面介绍。用户定义的转换将在详细讨论类（class）之后介绍。用户定义的转换由**转换函数**（conversion function）来执行，它是类的一个成员函数，允许一个类定义自己的“标准”转换。在第 15 章我们将看到类的转换函数以及涉及用户定义的转换的函数重载解析过程。

为一个函数调用选择最佳可行函数时，编译器会选择在实参的类型转换方面“最好”的一个函数。函数转换被划分等级如下：精确匹配比提升好，提升比标准转换好，标准转换比用户定义的转换好。我们将在 9.4 节进一步了解类型转换的等级。但是现在，我们描述的是各种可能的类型转换，本节中的某些例子会给出简单的情况，即怎样用这种等级划分来选择最佳可行函数。

9.3.1 精确匹配的细节

精确匹配最简单的例子是实参与函数参数类型精确匹配。例如，已知下面 max()重载函数集中的两个函数，则后面两个 max()调用中的实参与重载集合的特定函数的参数精确匹配

```
int max( int, int );
double max( double, double );

int i1;

void calc( double d1 ) {
     max( 56, i1 );    // 精确匹配 max( int, int );
     max( d1, 66.9 );  // 精确匹配 max( double, double );
}
```

枚举类型定义了一个惟一的类型，它只与枚举类型中的枚举值以及被声明为该枚举类型的对象精确匹配。例如：

```
enum Tokens { INLINE = 128; VIRTUAL = 129; };
Tokens curTok = INLINE;

enum Stat { Fail, Pass };

extern void ff( Tokens );
extern void ff( Stat );
```

```
extern void ff( int );
int main() {
     ff( Pass );     // 精确匹配 ff( Stat )
     ff( 0 );        // 精确匹配 ff( int )
     ff( curTok );   // 精确匹配 ff( Tokens )
     // ...
}
```

正如前面所提到的，即使一个实参必须应用一些最小的类型转换才能将其转换为相应函数参数的类型，它仍然是精确匹配的。

这些转换的第一个就是从左值到右值的转换。左值代表了一个可被程序寻址的对象，可以从该对象读取一个值，除非该对象被声明为 const，否则它的值也可以被修改。相对来说，右值只是一个表达式，它表示了一个值，或一个引用了临时对象的表达式，用户不能寻址该对象，也不能改变它的值。下面是一个简单的例子：

```
int calc( int );
int main() {
     int lval, res;

     lval = 5;       // 左值: lval; 右值: 5
     res = calc( lval );
                     // 左值: res;
                     // 右值: 存放 calc() 的返回值的临时对象
     return 0;
}
```

在第一个赋值表达式中，lval 是个左值，文字常量 5 是个右值。在第二个赋值表达式中，res 是个左值，函数 calc()调用返回值的临时对象是个右值。

在某些情况下，当预计出现一个值的时候，我们也可以用一个左值表达式来实现。例如：

```
int obj1;
int obj2;
int main() {
     // ...
     int local = obj1 + obj2;
     return 0;
}
```

obj1 和 obj2 是左值表达式。但是 main()中的加法只需要存贮在 obj1 和 obj2 中的值。在执行加法前，从 obj1 和 obj2 中把这些值抽取出来。从一个左值表达式所表示的对象中抽取值的动作就是一个“从左值到右值的转换”。

当一个函数期望一个按值传递的实参，而该实参又是一个左值的时候，就会执行从左值到右值的转换。例如：

```
#include <string>

string color( "purple" );
void print( string );

int main() {
     print( color );   // 精确匹配: 从左值到右值的转换
```

```
    return 0;
}
```

因为 print()调用中的实参是按值传递的，所以发生了从左值到右值的转换，它从 color 中抽取出一个值，将其传递给 print(string)。即使发生了从左值到右值的转换，实参 color 也还是 print(string)的精确匹配。

不是所有函数调用都要求实参进行从左值到右值的转换。一个引用表示一个左值，所以当一个函数有一个引用参数时，被调用的函数接受一个左值。因此，不会有从左值到右值的转换被应用到相应的引用参数的实参上。例如，已知函数：

```
#include <list>
void print( list<int> & );
```

下列调用中的 li 是一个左值，代表被传递给函数 print()的 list<int>对象。

```
list<int> li(20);

int main() {
    // ...
    print( li ); // 精确匹配：没有从左值到右值的转换
    return 0;
}
```

li 与引用参数的绑定是个精确匹配。

精确匹配允许的第二种转换是从数组到指针的转换。如在 7.3 节中所提到的，函数参数没有数组类型，取而代之的是参数被转换成指向数组首元素的指针。类似地，类型为 NT 数组（这里 N 是数组元素的个数，T 是数组元素的类型）的实参总是被转换成 T 型的指针。实参类型的转换是从数组到指针的转换。即使发生了转换，实参仍然被看作是 T 型指针参数的精确匹配。例如：

```
int ai[3];
void putValues(int *);

int main() {
    // ...
    putValues(ai); // 精确匹配：从数组到指针的转换
    return 0;
}
```

在函数 putValues()被调用之前，发生了从数组到指针的转换，将实参 ai 从三个 int 的数组转换成 int 型的指针。即使 putValues()有一个指针参数，且在实参上发生了从数组到指针的转换，但是该实参也仍是 putValues()调用的精确匹配。

精确匹配允许的下一种转换是从函数到指针的转换，该转换在 7.9 节中已简要介绍过了。和数组类型参数一样，函数类型的参数自动被转换成指向函数的指针。函数类型的实参也自动被转换成函数指针类型。这种实参类型的转换被称为从函数到指针的转换。即使发生了这种转换，该实参仍被看作是函数指针类型参数的精确匹配。例如：

```
int lexicoCompare( const string &, const string & );

typedef int (*PFI)( const string &, const string & );
void sort( string *, string *, PFI );
```

```
string as[10];
int main()
{
    // ...
    sort( as,
        as + sizeof(as) / sizeof(as[0] - 1),
        lexicoCompare// 精确匹配：从函数到指针的转换
    );

    return 0;
}
```

在函数 sort()被调用之前，发生了从函数到指针的转换，它将实参 lexicoCompare 从函数类型转换成函数指针类型。即使该函数期望接收的是一个指针而实参是一个函数名，即使发生了从函数到指针的转换，该实参也仍然是 sort()的第三个参数的精确匹配。

精确匹配的最后一种转换是限定修饰转换，这种转换只影响指针。它将限定修饰符 const 或 volatile（或两者）加到指针指向的类型上。例如：

```
int a[5] = { 4454, 7864, 92, 421, 938 };
int *pi = a;
bool is_equal( const int * , const int * );
int func( int *parm ) {

    // pi 和 parm 的精确匹配：限定修饰转换
    if ( is_equal( pi, parm ) )

    // ...
    return 0;
}
```

在函数 is_equal()被调用之前，实参 pi 和 parm 被从 int 型指针转换成指向 const int 型的指针。该转换把 const 限定修饰符加到指针指向的类型上，所以是限定修饰转换。即使函数期望两个 const int 的指针，而两个实参是指向 int 型的指针，这两个实参也仍然是 is_equal()的参数的精确匹配。

限定修饰转换只应用在指针指向的类型上。当参数是 const 或 volatile 类型，而实参不是时，没有类型转换发生：

```
extern void takeCI( const int );
int main() {
    int ii = ...;

    takeCI(ii); // 无转换发生
    return 0;
}
```

在 takedCI()调用中，即使参数是 const int 型，也不会有限定修饰转换被应用在 int 型的实参 li 上 。该实参是函数参数类型的精确匹配。

如果实参是指针，且有 const 或 volatile 限定符应用在指针上，也是这样：

```
extern void init( int *const );
```

```
extern int *pi;
int main() {

    // ...
    init(pi); // 没有限制转换
    return 0;
}
```

由于 init()参数上的 const 限定修饰符只应用在指针本身上，而并没有应用在指针指向的类型上。因此，编译器在考虑应用在实参上的转换时不会考虑 const 限定修饰符。因为没有限定修饰转换被应用在实参 pi 上，所以该实参与函数参数类型精确匹配。

精确匹配类别中的前三种转换（从左值到右值、从数组到指针以及从函数到指针的转换）通常被称为**左值转换**（lvalue transformation）。正如在 9.4 节中即将看到的那样，虽然左值转换和限定修饰转换都属于精确匹配类别，但是只需要左值转换的精确匹配比需要限定修饰转换的要好。我们将在下节中更详细地讨论这些。

精确匹配可以用一个显式强制转换强行执行。例如，已知重载函数集合：

```
extern void ff(int);
extern void ff(void *);
```

如下调用；

```
ff( 0xffbc ); // 调用 ff(int)
```

与 ff(int)精确匹配，因为 0xffbc 是十六进制形式的 int 型文字常量。程序员可以如下提供一个显式转换来强制调用 ff(void*)。

```
ff( reinterpret_cast<void *>(0xffbc) );   // 调用 ff(void*)
```

显式强制转换应用在实参上时，实参的类型就变成强制转换的结果。使用显式强制转换的类型转换可以帮助指导函数重载解析。例如，如果因为实参与两个以上可行函数匹配，使得函数重载解析的结果是二义的，则可以用显式强制转换来打破二义性，使函数调用被解析为一个特殊的可行函数。

9.3.2 提升的细节

提升实际上就是下列转换之一：

- char、unsigned char 或 short 型的实参被提升为 int 型。如果机器上 int 型的字长比 short 整型的长，则 unsigned short 型的实参被提升到 int 型；否则，它被提升到 unsigned int 型。
- float 型的实参被提升到 double 类型。
- 枚举类型的实参被提升到下列第一个能够表示其所有枚举常量的类型：int、unsigned int、long 或 unsigned long。
- 布尔型的实参被提升为 int 型。

当实参的类型是上面描述的源类型之一，而函数参数的类型是相应被提升的类型时，则应用该提升。例如：

```
extern void manip( int );
```

```
int main() {
    manip( 'a' ); // 类型 char 被提升为 int
    return 0;
}
```

字符文字的类型是 char，它的提升类型是 int。因为提升的类型与函数 manip()的参数类型匹配，所以我们说函数调用要求提升它的实参。

假设有下列例子：

```
extern void print( unsigned int );
extern void print( int );
extern void print( char );

unsigned char uc;
print( uc ); // print( int ): uc 只需要提升
```

在 unsigned char 类型只占一个字节、而 int 型占四个字节的机器上，由于类型 int 可以表示 unsigned char 型的全部值，所以提升就是将一个 unsigned char 的实参变成 int 型。在上面给定的重载函数声明，以及刚刚描述的结构中，与 unsigned char 型实参最匹配的函数是 print(int)，要匹配其他两个函数则要求应用标准转换。

下面的例子说明了枚举型实参的提升：

```
enum Stat { Fail, Pass };

extern void ff( int );
extern void ff( char );

int main() {
    // ok: 枚举常量 Pass 被提升到 int
    ff( Pass );       // ff( int )
    ff( 0 );          // ff( int )
    return 0;
}
```

枚举类型的提升有时候会使人惊奇，编译器经常根据枚举常量的值来选择枚举类型的表示。例如，假设有前面描述的结构（char 有一个字节，int 有四个字节）以及下面的枚举类型：

```
enum e1 { a1, b1, c1 };
```

因为只有三个枚举常量——a1、b1 和 c1——它们的值分别为 0、1、2，该枚举类型的所有值都可以用 char 型表示，所以编译器常常会选择 char 型作为 e1 的表示。但是，假设我们有另外一个枚举类型 e2，它有不同的枚举常量值：

```
enum e2 { a2, b2, c2=0x80000000 };
```

因为有一个枚举常量的值是 0x80000000，所以该编译器被迫为 e2 选择一个能够表示值 0x80000000 的表示，这个表示就是 unsigned int。

因此，即使 e1 和 e2 都是枚举类型，它们的表示也并不相同。这使 e1 和 e2 被提升为不同的类型，例如：

```
#include <string>

string format( int );
string format( unsigned int );
```

```
int main() {
    format(e1);   // 调用 format( int )
    format(e2);   // 调用 format( unsigned int )
    return 0;
}
```

在第一个 format()的调用中，因为实参的类型是 char 型表示的类型 e1，所以实参被提升为 int 型，为该调用选择的函数是 format(int)。在 format()的第二个调用中，因为实参的类型是 unsigned int 型表示的 e2 型，所以实参被提升为类型 unsigned int。这使得函数 format(unsigned int)被选择给第二个调用。因此，你应该知道：两个枚举类型在重载函数解析期间的行为可能完全不同，解析过程根据枚举常量的值来决定它们被提升的类型。

9.3.3 标准转换的细节

有五种转换属于标准转换：

1．整值类型转换：从任何整值类型或枚举类型向其他整值类型的转换（不包括前面提升部分中列出的转换）。

2．浮点转换：从任何浮点类型到其他浮点类型的转换（不包括前面提升部分中列出的转换）。

3．浮点—整值转换：从任何浮点类型到任何整值类型或从任何整值类型到任何浮点类型的转换。

4．指针转换：整数值 0 到指针类型的转换和任何类型的指针到类型 void*的转换。

5．bool 转换：从任何整值类型、浮点类型、枚举类型或指针类型到 bool 型的转换。

下面是一些例子：

```
extern void print( void* );
extern void print( double );
int main() {
    int i;
    print( i );   // 匹配 print( double );
                  // i 被一个标准转换从 int 转换到 double
    print( &i );  // 匹配 print( void* );
                  // &i 被标准转换从 int* 转换到 void*

    return 0;
}
```

类别 1、2 和 3 中的转换是有潜在危险的转换，这是因为转换的目标类型不能表示源类型的全部值。例如，类型 float 不能表示出 int 类型的所有值的精度。这也是这些类别中的转换是标准转换而不是提升转换的原因。

```
int i;
void calc( float );

int main() {
    calc( i );    // 浮点—整值标准转换
                  // 潜在危险，取决于 i 值
    return 0;
```

当用户调用函数 calc()时，浮点—整值标准转换把实参从 int 型转换成 float 型。根据存储在 i 中值的情况，可能无法保证把 i 的值存储在类型 float 的参数内并且不损失精度。

所有的标准转换都被视为是等价的。例如，从 char 到 unsigned char 的转换并不比从 char 到 double 的转换优先级高。类型之间的接近程度不被考虑。即，如果有两个可行函数要求对实参进行标准转换以便匹配各自参数的类型，则该调用就是二义的，将被标记为编译错误。例如，下面给出的一对重载函数：

```
extern void manip( long );
extern void manip( float );
```

下列调用是二义的：

```
int main() {
     manip( 3.14 );      // 错误：二义性
                         // manip( float ) 也不会好到那里
     return 0;
}
```

文字常量 3.14 是 double 型的，通过标准转换两个函数都能匹配。因为可能存在有两种标准转换，所以该调用被标记为二义的。没有一个标准转换比其他的标准转换更为优先。程序员可以用显式强制转换来解决二义性的问题，比如：

```
manip( static_cast<long>( 3.14 ) );    // manip( long )
```

或通过用 float 常量后缀：

```
manip( 3.14F );    // manip( float )
```

下面是一些其他函数调用的例子，因为它们都与重载函数集中的多个函数匹配，所以它们都是二义的，并都被标记为错误：

```
extern void farith( unsigned int );
extern void farith( float );
int main() {
     // 每个调用都是二义的
     farith( 'a' );      // 实参类型为 char
     farith( 0 );        // 实参类型为 int
     farith( 2uL );      // 实参类型为 unsigned long
     farith( 3.14159 );  // 实参类型为 double
     farith( true );     // 实参类型为 bool
     return 0;
}
```

有时标准指针转换看起来有些违反直觉。尤其是，0 可以被转换成任何指针类型：这样创建的指针值被称为**空指针值**（null pointer value）。同时，值 0 也可以是任何整型常量表达式。例如：

```
void set(int*);

int main() {
     // 从 0 到 int* 的指针转换应用到两个实参上
     set( 0L );
     set( 0x00 );
     return 0;
```

```
    return 0;
}
```

常量表达式 0L（long int 型的 0）以及常量表达式 0x00（十六进制的 0）都属于整型类型因此能够被转换成 int*型的空指针值。

但是，因为枚举类型不是整型，仍为 0 的枚举型值不能被转换成指针类型。例如：

```
enum EN { zr = 0 };
set( zr );    // 错误：zr 不能被转换到 int*
```

对 set()的调用是错的，因为在枚举值 zr 和 int*型的参数之间不存在可能的转换，即使该枚举值为 0。

还有一些事情要注意，常量表达式 0 属于类型 int。把这常量表达式转换成指针类型的标准转换是必需的。如果重载函数集中有一个函数，它的参数是 int 型，则对于实参 0，该函数会被优先考虑。例如：

```
void print( int );
void print( void * );
void set( const char* );
void set( char* );

int main() {
    print( 0 );  // 调用 print( int )
    set( 0 ); // 二义
    return 0;
}
```

实参对 print(int)的调用是精确匹配。但是，为了调用 print(void*)，需要一个标准转换将 0 转换成指针类型。因为精确匹配比标准转换要好，所以该调用选择了函数 print(int)。对 set()的调用是二义的，因为通过应用标准转换，0 与两个 set()函数的参数都匹配。且两个函数对该调用一样好，所以它是二义的。

最后一种指针转换允许将任何指针类型的实参转换成 void*型的参数，因为 void*是通用的数据类型指针，所以它可以存放任何数据类型的指针值。下面是一些例子：

```
#include <string>

extern void reset( void * );
int func( int *pi, string *ps ) {
    // ...
    reset( pi ); // 指针转换：int* 到 void*
    // ...
    reset( ps ); // 指针转换：string* 到 void*
    return 0;
}
```

只有指向数据类型的指针才可以用指针标准转换将其转换成类型 void*，函数指针不能用标准转换转换成类型 void*。例如：

```
typedef int (*PFV)();
extern PFV testCases[10];    // 函数指针数组
extern void reset( void * );
```

```
int main() {
    // ...
    reset( testCases[0] );  // 错误：在 int(*)() 之间不存在标准转换

    return 0;
}
```

9.3.4 引用

函数调用的实参或函数参数都可以是引用。那么，引用又是怎样影响类型转换规则的呢？

首先，我们来看一下如果实参是一个引用时会发生什么情况。实参的类型永远不会是引用类型。当实参是一个引用时，该实参是一个左值，它的类型是引用所指的对象的类型。考虑下列例子：

```
int i;
int & ri = i;
void print( int );
int main() {
    print( i );   // int 型的左值实参
    print( ri ); // 同样
    return 0;
}
```

两个函数调用的实参都是 int 型，而在第二个调用中引用被用作实参，对实参类型没有任何影响。

当一个实参是类型 T 的引用时，所考虑的标准转换和提升与该实参是 T 型对象时的一样。例如：

```
int i;
int& ri = i;
void calc( double );
int main() {
    calc( i );    // 浮点–整值标准转换
    calc( ri );   // 同样
    return 0;
}
```

那么，引用参数是怎样影响应用在实参上的转换的呢？实参与引用参数的匹配结果有下面的两种可能。

1．实参是引用参数的合适的初始值。在这种情况下，我们说该实参是参数的精确匹配。例如：

```
void swap( int &, int & );
int manip( int i1, int i2 ) {
    // ...
    swap( i1, i2 );   // ok: 调用 swap( int &, int & )
    // ...
    return 0;
```

```
}
```

2．实参不能初始化引用参数。在这种情况下，没有匹配情况发生，实参不能被用来调用该函数。例如：

```
int obj;
void frd( double & );
int main() {
     frd( obj );   // 错误：参数必须是 const double &
     return 0;
}
```

对 frd()的调用是错误的。实参类型是 int，必须被转换成 double 以匹配引用参数的类型。该转换的结果是个临时值。因为这种引用不是 const 型的，所以临时值不能被用来初始化该引用。

下面是在引用参数与实参之间没有匹配的另外一个例子。

```
class B;
void takeB( B& );
B giveB();

int main() {
     takeB( give() );  // 错误：参数必须是 const B&
     return 0;
}
```

对 takeB()的调用是错误的。实参是函数调用的返回值，它是一个临时值，不能被用来初始化非 const 型的引用。

在这两种情况下，如果引用参数是 const 型的引用，则实参就是参数的精确匹配。针对下面给出的代码：

```
void print( int );
void print(int&);

int iobj;
int &ri = iobj;

int main() {
     print( iobj );     // 错误：二义
     print( ri );       // 错误：二义
     print( 86 );       // ok: 调用 print( int )
     return 0;
}
```

第一个函数调用是错误的。因为对象 iobj 是与两个函数 print()都精确匹配的实参，所以函数调用是二义的。对第二个函数调用也一样。引用 ri 指向一个对象，它是两个函数的精确匹配。但是，第三个调用是正确的。函数 print(int&)不是该调用的可行函数。整型常量是个右值，不是非 const 引用参数的有效的初始值。在调用 print(86)的可行函数集中只有一个函数 print(int)。因为它是惟一的可行函数，所以它是该调用选择的函数。

简而言之，对于引用参数来说，如果实参是该引用的有效初始值，则该实参是精确匹配。如果该实参不是引用的有效初始值，则不匹配。

练习 9.6

指出精确匹配中允许的两个最小转换。

练习 9.7

在下列函数调用中，实参上的每个转换的等级是什么？

```
(a)  void print( int *, int);
     int arr[6];
     print( arr, 6 );                  // 函数调用
(b)  void manip( int, int );
     manip( 'a', 'z' );                // 函数调用
(c)  int calc( int, int);
     double dobj;
     double = calc( 55.4, dobj );      // 函数调用
(d)  void set( const int * );
     int *pi;
     set( pi );                        // 函数调用
```

练习 9.8

下列哪个函数调用是因为实参与函数参数之间不存在类型转换而发生错误？

```
(a)  enum Stat { Fail, Pass };
     void test( Stat );
     test( 0 );                        // 函数调用
(b)  void reset( void * );
     reset( 0 );                       // 函数调用
(c) void set( void * );
     int *pi;
     set( pi );                        // 函数调用
(d) #include <list>
     list<int> oper();
     void print( list<int> & );
     print( oper() );                  // 函数调用
(e)  void print( const int );
     int iobj;
     print( iobj );                    // 函数调用
```

9.4 函数重载解析细节 ※

如 9.2 节所述，函数重载解析过程有三个步骤。这些步骤可以总结如下：

1．确定为该调用而考虑的候选函数，以及函数调用中的实参表属性。

2．从候选函数中选出可行函数。也就是说；根据调用中指定的实参、实参数目和类型，

选择可以被调用的函数。

3．对于“被用来将实参转换成可行函数参数类型的转换”划分等级，以便选出与调用最匹配的函数。

下面，我们来详细讨论这三个步骤。

9.4.1 候选函数

候选函数与被调用的函数具有同样的名字。可以用下面两种方式找到候选函数：

1．该函数的声明在调用点上可见。给出下列例子：

```
void f();
void f( int );
void f( double, double = 3.4 );
void f( char*, char* );
int main() {
     f( 5.6 );          // 这个调用有四个候选函数
     return 0;
}
```

因为在全局域中声明的四个 f()在调用点上都可见。所以它们都是候选函数集的一部分。

2．如果函数实参的类型是在一个名字空间中被声明的，则该名字空间中与被调用函数同名的成员函数也将被加入到候选函数集中。例如：

```
namespace NS {
     lass C { /* ... */ };
     oid takeC( C& );
}
// cobj 的类型是在名字空间 NS 中被声明的类 C
NS::C cobj;

int main() {
                          // 在调用点没有 takeC()可见
     takeC( cobj );       // ok: 调用 NS::takeC( C& )
                          // 因为实参类型是 NS::C
                          // 所以考虑在名字空间 NS 中声明的函数 takeC()
     return 0;
}
```

因此，候选函数是“在调用点上可见的函数”以及“在实参类型所在的名字空间中声明的同名函数”的集合。

当我们确定在调用点上可见的重载函数集合时，我们在前面看到的关于怎样生成重载函数集的规则仍然适用。

在嵌套的域中被声明的函数隐藏了而不是重载了外围域中的同名函数。这种情况下的候选函数是在嵌套域中被声明的函数，即没有被该函数调用隐藏的函数。在下面的例子中，在调用点上可见的候选函数是 format(double)和 format(char*)：

```
char* format( int );
void g() {
     char* format( double );
```

```
        char* format( char* );
        format(3);          // 调用 format( double )
}
```

因为在全局域中声明的函数 format(int)被隐藏，所以它没有被包含在候选函数集中。

在调用点上可见的 using 声明也可以引入候选函数。考虑下列例子：

```
namespace libs_R_us {
        int max( int, int );
        double max( double, double );
}

char max( char, char );

void func()
{
        // 名字空间的函数不可见
        // 这三个调用分别调用全局函数 max( char, char )
        max( 87, 65 );
        max( 35.5, 76.6);
        max( 'J', 'L' );
}
```

名字空间 libs_R_us 中定义的函数 max()在调用点上不可见，惟一可见的是全局域中声明的函数 max()。该函数是候选函数集中惟一的一个函数：它是 func()中三个调用所调用的函数。我们可以用 using 声明使名字空间 libs_R_us 中声明的函数 max()变为可见。那么，using 声明应该放在哪儿呢？如果把 using 声明放在全局域中，

```
char max( char, char );
using libs_R_us::max;     // using 声明
```

那么，来自名字空间 libs_R_us 中的函数 max()就将被加到重载函数集中，该集合同时还包含全局域中声明的函数 max()。现在，三个函数在 func()中都可见，并且都成为侯选函数集中的一部分。随着三个函数在调用点上可见，func()中的调用被解析如下：

```
void func()
{
        max( 87, 65 );        // 调用 libs_R_us::max( int, int )
        max( 35.5, 76.6 );    // 调用 libs_R_us::max( double, double )
        max( 'J', 'L' );      // 调用 max( char, char )
}
```

但是，如果我们在函数 func()的局部域中如下引入了 using 声明又会怎么样呢？

```
void func()
{
        // using 声明
        using libs_R_us::max;

        // 函数调用如上
}
```

候选函数集中会包含哪些 max()？请回忆一下 using 声明的嵌套。由于局部域中的 using 声明，全局函数 max(char, char)被隐藏。在调用点上可见的函数只是：

```
libs_R_us::max( int, int )
libs_R_us::max( double, double )
```

这两个函数是候选函数集中的函数，func()中的调用被解析如下：

```
void func()
{
    // using 声明
    // 全局 max( char, char ) 被隐藏
    using libs_R_us::max;

    max( 87, 65 );          // 调用 libs_R_us::max( int, int )
    max( 35.5, 76.6 );      // 调用 libs_R_us::max( double, double )
    max( 'J', 'L' );        // 调用 libs_R_us::max( int, int )
}
```

using 指示符也会影响候选函数集的构成。假设我们决定用 using 指示符而不是 using 声明使名字空间 lib_R_us 中的函数 max()在 func()中可见。例如，使用下面全局域中的 using 指示符，候选函数集就将包含全局函数 max(char,char)以及在名字空间 libs_R_us 中声明的函数 max(int, int)和 max(double, double)：

```
namespace libs_R_us {
    int max( int, int );
    double max( double, double );
}
char max( char, char );

using namespace libs_R_us; // using 指示符

void func()
{
    max( 87, 65 );          // 调用 libs_R_us::max( int, int )
    max( 35.5, 76.6 );      // 调用 libs_R_us::max( double, double )
    max( 'J', 'L' );        // 调用 ::max( char, char )
}
```

假若像下面这样，将 using 指示符放到 func()的局部域内，又会怎么样呢？

```
void func()
{
    // using 指示符
    using namespace libs_R_us;

    // 函数调用如上
}
```

哪些 max()会成为候选函数？记住，using 指示符使名字空间成员可见就好像它们是在名字空间之外、在定义名字空间的位置上被声明的一样。在我们的例子中，名字空间 libs_R_us 的成员在 func()的局部域内可见，就好像该成员已经在名字空间之外、全局域内被声明的样。这暗示着在 func()内可见的重载函数集与前面包含下列三个函数的一样：

```
max( char, char )
libs_R_us::max( int, int )
libs_R_us::max( double, double )
```

无论 using 指示符出现在全局域还是 func()的局部域内，都不会影响 func()中的调用的解析过程：

```
void func()
{
    using namespace libs_R_us;
    max( 87, 65 );       // 调用 libs_R_us::max( int, int )
    max( 35.5, 76.6 );   // 调用 libs_R_us::max( double, double )
    max( 'J', 'L' );     // 调用 ::max( char, char )
}
```

所以，候选函数集是在调用点上可见的函数（包括 using 声明和 using 指示符引入的函数）以及在与实参类型相关的名字空间内被声明的成员函数。例如：

```
namespace basicLib {
    int print( int );
    double print( double );
}
namespace matrixLib {
    class matrix { /* ... */ };
    void print( const matrix & );
}

void display()
{
    using basicLib::print;
    matrixLib::matrix mObj;
    print( mObj );       // 调用 matrixLib::print( const matrix& )
    print( 87 );         // 调用 basicLib::print( int )
}
```

哪些函数是调用 print(mObj)的候选函数？因为由函数 display()中的 using 声明引入的函数 basicLib::print(int)和 basicLib::print(double)在调用点上可见，所以它们都是候选函数。因为函数调用实参的类型是 matrixLib::matrix，所以在名字空间 matrixLib 中声明的函数 print()也是个候选函数。调用 print(87)的候选函数是哪些呢？在调用点上只有函数 basicLib::print(int)和 basicLib::print(double)可见，所以它们是候选函数。因为实参的类型是 int，所以编译器不会在其他名字空间中寻找其他候选函数。

9.4.2 可行函数

可行函数是候选函数集合中的函数。它的参数表或者与调用中的实参数目相同，或者有更多的参数。在后一种情况下，额外的参数会被给出缺省实参，以便可以用实参表中指定的实参调用该函数。可行函数是这样的函数：对于每个实参，都存在到函数参数表中相应的参数类型之间的转换。可被考虑的转换是 9.3 节中介绍的转换。

在下面的例子中，对调用 f(5.6)来说有两个可行函数：它们是 f(int)和 f(double)。

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

```
void f( char*, char* );
int main() {
    f( 5.6 );          // 两个可行函数: f( int ) 和 f( double )
    return 0;
}
```

f(int)是可行函数。因为它只有一个参数，这与函数调用中实参的数目匹配，并且存在着把实参从 double 型转换成 int 型的标准转换。f(double)也是个可行函数。这个可行函数只有一个参数，类型为 double，是调用中实参的精确匹配。候选函数 f()和 f(char*, char*)被排除在可行函数集合之外，是因为这些函数不能用一个实参调用。

在下面的例子中，调用 format(3)的唯一可行函数是函数 format(double)。虽然候选函数 format(char*)也可以用一个实参调用，但是在 int 型的实参和 char*型的参数之间不存在转换。就因为不存在该类型转换，所以该函数被排除在可行函数集合之外。

```
char* format( int );

void g() {
    // 全局函数 format( int ) 被隐藏
    char* format( double );
    char* format( char* );
    format(3);         // 只有一个可行函数: format( double )
}
```

在下面的例子中，三个候选函数都在 func()中 max()调用的可行函数集合中。这三个函数都可以用两个实参来调用。因为实参类型是 int，它是 libs_R_us::max(int,int)的参数的精确匹配，所以这两个实参可以通过“浮点—整值标准转换”转换成 libs_R_us::max(double,double)的参数，以及通过“整值标准转换”转换成 max(char,char)的参数。

```
namespace libs_R_us {
    int max( int, int );
    double max( double, double );
}

// using 声明
using libs_R_us::max;
char max( char, char );

void func()
{
    // 这三个 max() 都是可行函数
    max( 87, 65 );    // 调用 libs_R_us::max( int, int )
}
```

注意，对于有多个参数的候选函数来说，只要函数调用中的一个实参不能被转换成候选函数参数表中相应的参数，它就将马上被排除在可行函数集合之外，即使其他实参都存在转换。在下面的例子中，函数 min(char*,int)被排除在可行函数之外，因为在第一个实参 int 类型与相应函数参数 char*类型之间不存在转换。即使第二个实参是函数的第二个参数的精确匹配，该函数也会被排除：

```
extern double min( double, double );
extern int min( char*, int );
```

```
void func()
{
     // 候选函数 min( double, double )
     min( 87, 65 );    // 调用 min( double, double )
}
```

如果在去掉参数个数不同的候选函数（或去掉不存在合适的类型转换的候选函数）之后，没有可行函数存在，则该调用就会导致编译时刻错误。在这种情况下，我们就说没有找到匹配。

```
void print( unsigned int );
void print( char* );
void print( char );

int *ip;
class SmallInt { /* ... */ };
SmallInt si;

int main() {
     print( ip );      // 错误：没有可行函数：没有匹配
     print( si );      // 错误：没有可行函数：没有匹配
     return 0;
}
```

9.4.3 最佳可行函数

最佳可行函数是具有与实参类型匹配最好的参数的可行函数。对于每个可行函数来说，每个实参的类型转换都被划分了等级，以决定每个实参与其相应参数的匹配程度（9.2 节描述了得到支持的类型转换。）。最佳可行函数是满足下列条件的可行函数：

1．用在实参上的转换不比调用其他可行函数所需的转换更差

2．在某些实参上的转换要比其他可行函数对该参数的转换更好。

将实参转换成相应的函数参数时可能不只应用一种类型转换。例如，在下面的例子中：

```
int arr[3];
void putValues(const int *);

int main() {
     putValues(arr);   // 在转换序列中有 2 个转换
                       // 数组到指针、限定修饰转换
     return 0;
}
```

将实参 arr 从三个 int 元素的数组转换成 const int 指针类型应用了一个转换序列，该转换序列由下列转换构成：

1．从数组到指针的转换。将实参从三个 int 元素的数组转换成 int 型的指针。

2．限定修饰转换。把 int 型的指针转换成 const int 型的指针。

因此说一个**转换序列**（conversion sequence）被用来把实参转换成可行函数参数的类型更为合适。因为是一个转换序列而不是单个转换被应用到实参上将其转换成相应参数的类型，所以函数重载解析的第三步是将转换序列划分等级。

转换序列的等级是构成该序列最坏转换的等级。正如在 9.2 节中描述的，类型转换的等级划分如下：精确匹配好于提升，提升好于标准转换、在前面的例子中，序列中的两个转换都具有精确匹配的等级。

一个转换序列潜在地由下列转换以下列顺序构成：

左值转换——>

提升或者标准转换——>

限定修饰转换

左值转换（lvalue transformation）是指 9.2 节里讲的精确匹配类别中描述的前三个转换：从左值到右值的转换、从数组到指针的转换和从函数到指针的转换。转换序列的构成是这样的：首先是 0 个或一个左值转换，接着是 0 个或一个提升、或者 0 个或一个标准转换，再后面是 0 个或一个限定修饰转换。至多，每种转换会有一个被应用上，以将实参转换成相应的参数。

这种转换序列被称为**标准**（standard）转换序列。还有另外一种转换序列被称为**用户定义的**（user-defined）转换序列。用户定义的转换序列包含类成员转换函数，类成员转换函数和用户定义的转换序列将在 15 章讲述。

下面的例子中实参上的转换序列是什么？

```
namespace libs_R_us {
    int max( int, int );
    double max( double, double );
}
// using 声明
using libs_R_us::max;
void func()
{
    char c1, c2;
    max( c1, c2 );    // 调用 libs_R_us::max( int, int )
}
```

在 max()的调用中的实参是 char 型的。调用函数 libs_R_us::max(int, int)的实参上的转换序列如下：

1a．因为该实参按值传递，所以首先是从左值到右值转换，它从实参 c1 和 c2 中抽取值。

2a．提升转换将实参从 char 转换到 int。

调用函数 libs_R_us::max(double,double)的实参上的转换序列如下：

1b．先应用一个从左值到右值的转换，它从实参 c1 和 c2 抽取值。

2b．浮点——整值标准转换将实参从 char 转换成 double。

第一个转换序列的等级是提升（序列中最差的转换），而第二个转换序列的等级是标准转换。因为提升比标准转换好，所以函数 libs_R_us::max(int,int)被选为该调用的最佳可行函数，或最佳匹配函数。

如果通过对实参上的转换序列划分等级，仍然不能够判别出一个可行函数比其他函数更匹配实参的类型，则该调用就是二义的。在下面的例子中，calc()的两个实例都要求下列转换序列：

1．首先是一个从左值到右值的转换，它从实参 i 和 j 中抽取值。

2．通过一个标准转换把实参转换成相应的参数。

因为每个转换序列都和另一个一样好，所以该调用是二义的：

```
int i, j;
extern long calc( long, long );
extern double calc( double, double );

void jj() {
    // 错误：二义，没有最佳匹配
    calc( i, j );
}
```

限定转换（把 const 或 volatile 修饰符加到指针指向的类型上的转换）具有精确匹配的等级。但是，如果两个转换序列前面都相同，只是一个在序列尾部有一个额外的限定转换，则另一个没有额外限定转换的序列比较好。例如：

```
void reset( int * );
void reset( const int * );

int* pi;

int main() {
    reset( pi ); //没有限定转换的比较好；选择 reset( int * )
    return 0;
}
```

应用在调用第一个候选函数 reset(int*)的实参上的标准转换序列是个精确匹配：它只要求一个从左值到右值的转换来抽取实参的值。对第二个候选函数 reset(const int*)，也应用了一个从左值到右值的转换，接着是限定修饰转换把结果值从 int 指针转换成 const int 指针。这两个序列都是精确匹配，但是上面的函数调用不是二义的，因为这两个转换序列的第一个转换相同，但是第二个转换序列末尾有额外的限定修饰转换，所以第一个没有限定修饰转换的序列被认为是较好的匹配。因此，可行函数 reset(int*)是最佳可行函数。

下面是另一个例子，其中限定修饰转换影响了被选择的转换序列：

```
int extract( void * );
int extract( const void * );

int* pi;

int main() {
    extract( pi ); // 选择 extract( void * )
    return 0;
}
```

该调用有两个可行函数：extract(void*)和 extract(const void*)。在调用第一个可行函数 extract(void*)上应用的转换序列中，有一个从左值到右值的转换来抽取实参的值，接着是一个标准指针转换，它将该值从一个 int 指针转换成一个 void 指针。应用在调用第二个函数 extract(const void*)上的转换序列也是如此，只不过还应用了一个额外的限定修饰转换，它将结果从 void 指针转换成 const void 指针。因为这两个转换序列，除了第二个转换序列在尾部

是一个额外的限定修饰转换外，其余都是相同的，所以第一个转换序列被选择为更好的转换序列。函数 extract(void*)被选为该实参的最佳可行函数。

const 或 volatile 修饰符也能影响引用参数的初始化的等级。如同转换序列的情形一样如果两个引用初始化是相同的，只不过其中一个增加了一个额外的 const 或 volatile 修饰符，那么对于函数重载解析来说，没有额外修饰符的引用初始化是比较好的引用初始化，例如：

```
#include <vector>
void manip( vector<int> & );
void manip( const vector<int> & );
vector<int> f();
extern vector<int> vec;

int main() {
     manip( vec );  // 选择 manip( vector<int> & ) is selected
     manip( f() );  // 选择 manip( const vector<int> & ) is selected
     return 0;
}
```

在第一个调用中，两个调用的引用初始化都是精确匹配。但是该调用不是二义的．因为两个引用初始化都相同，除了第二个加上了 const 修饰符，所以没有额外限定修饰的初始化被认为是较好的初始化。因此，可行函数 manip(vector<int>&)是第一个调用的最佳可行函数。

在第二个调用中，该调用只有一个可行函数：manip(const vector<int>&)。因为实参是存放函数 f()返回值的临时单元，所以该实参是一个右值，它不能被用来初始化 manip(vector<int>&)的非 const 引用参数。因此，对于第二个调用的最佳可行函数，编译器只考虑一个可行函数：manip(const vector<int>&)。

当然，函数调用可以有一个以上的实参，选择最佳可行函数时必须考虑转换全部实参所需的转换序列的等级。我们来看一个例子：

```
extern int ff( char*, int );
extern int ff( int, int );

int main() {
     ff( 0, 'a' );// ff( int, int )
     return 0;
}
```

由于下述原因，有两个 int 型的参数的函数 ff()被选为最佳可行函数。

1．它的第一个实参较好。0 是 int 型的参数的精确匹配，而对于第一个 ff(char*,int)函数来说，它需要一个指针标准转换序列来匹配 char*型的参数。

2．它们的第二个实参一样好。实参‘a’的类型是 char，两个函数的第二个参数的匹配都要求一个提升等级的转换序列。

这里还有另外一个例子：

```
int compute( const int&, short );
int compute( int&, double );
extern int iobj;

int main() {
```

```
    compute( iobj, 'c' );   // compute( int&, double )
    return 0;
}
```

这两个函数 compute(const int&,short)和 compute(int&,double)都是可行函数。由于下述原因第二个函数被选为最佳可行函数。

1．它的第一个实参较好。第一个可行函数的引用初始化比较差，因为它加入了一个 const 限定修饰符，而第二个可行函数的初始化没有加入 const 限定修饰符。

2．它们的第二个实参一样好。实参‘c’的类型是 char，要匹配两个函数的第二个参数都要求一个标准转换等级的转换序列。

9.4.4 缺省实参

缺省实参可以使多个函数进入到可行函数集合中。可行函数是指可以用调用中指定的实参进行调用的函数。可行函数可以有比函数调用实参表中的实参个数更多的参数，只要每个多出来的参数都有相应的缺省实参即可：

```
extern void ff( int );
extern void ff( long, int = 0 );
int main() {
    ff( 2L );     // 匹配 ff( long, 0 );
    ff( 0, 0 );   // 匹配 ff( long, int );
    ff( 0 );      // 匹配 ff( int );
    ff( 3.14 );   // 错误：二义
}
```

对于第一个和第三个调用，即使该实参表中只有一个实参，第二个函数 ff()仍然是两个调用的可行函数，原因如下：

1．函数的第二个参数有相应的缺省实参。

2．函数的第一个参数是 long 型，与第一个调用的实参类型精确匹配。通过标准转换等级的转换序列，与第三个调用的实参类型也匹配。

最后一个调用是二义的，这是因为通过在第一个实参上应用标准转换，两个实例都可以匹配。这里不能选择 ff(int)作为更好的函数，因为它只有一个实参。

练习 9.9

解释在 main()中对 compute()的调用的函数重载解析过程中发生的事情。哪些函数是候选函数？哪些函数是可行函数？应用在实参上使其与每个可行函数的参数匹配的类型转换序列是什么？哪个函数（如果存在的话）是最佳可行函数？

```
namespace primerLib {
    void compute( );
    void compute( const void * );
}

using primerLib::compute;
void compute( int );
void compute( double, double = 3.4 );
```

```
void compute( char*, char* = 0 );
int main() {
     compute( 0 );
     return 0;
}
```

如果将 using 声明放在 main()中，但是在 compute()调用之前，会怎么样？回答与上面同样的问题。

10

函 数 模 板

本章将讲述什么是函数模板（function template），并讨论怎样定义和使用函数模板。使用函数模板其实相当简单，许多C++初学者在使用库中定义的函数模板时，甚至不知道他们在使用模板。只有高级C++用户才会像本章中描述的那样定义和使用函数模板。因此，本章的内容可被用作高级C++主题的介绍性资料。我们从描述什么是函数模板以及怎样定义函数模板开始。然后说明函数模板的简单用法。在这之后。再将焦点转移到更高级的话题上。首先，我们将了解怎样以更高级的方式使用函数模板：我们将详细了解模板实参的推演过程，看看当引用一个模板实例时，怎样指定显式模板参数。然后我们会了解编译器怎样初始化模板及其对程序组织结构上的要求，并讨论怎样定义函数模板实例的特化版本。然后，本章将给出一些让函数模板设计者感兴趣的话题。我们将解释函数模板怎样被重载，以及涉及函数模板的重载解析过程如何工作。我们还会介绍函数模板定义中的名字解析。以及函数模板如何才能被定义在名字空间中。最后，本章将以一个使用函数模板的例子作为结束。

10.1 函数模板定义

有时候，强类型语言对于实现相对简单的函数似乎是个障碍。例如，虽然下面的函数min()的算法很简单，但是，强类型语言要求我们为所有希望比较的类型都实现一个实例：

```
int min( int a, int b ) {
     return a < b ? a : b;
}

double min( double a, double b ) {
     return a < b ? a : b;
}
```

有一种方法可替代这种“为每个min()实例都显式定义一个函数”的方法，这种方法很有吸引力，但是也很危险，那就是用预处理器的宏扩展设施。例如：

```
#define min(a,b) ((a) < (b) ? (a) : (b))
```

虽然该定义对于简单的 min()调用都能正常工作，如

```
min( 10, 20 );
min( 10.0, 20.0 );
```

但是，在复杂调用下，它的行为是不可预期的，这是因为它的机制并不像函数调用那样工作，只是简单地提供参数的替换。结果是，它的两个参数值都被计算两次：一次是在 a 和 b 的测试中，另一次是在宏的返回值被计算期间，例如：

```
#include <iostream>
#define min(a,b) ((a) < (b) ? (a) : (b))

const int size = 10;
int ia[size];

int main() {
    int elem_cnt = 0;
    int *p = &ia[0];

    // 计数数组元素的个数
    while ( min(p++,&ia[size]) != &ia[size] )
        ++elem_cnt;

    cout << "elem_cnt : " << elem_cnt
        << "\texpecting: " << size << endl;
    return 0;
}
```

这个程序给出了计算整型数组 ia 的元素个数的一种明显绕弯的的方法。min()的宏扩展在这种情况下会失败，因为应用在指针实参 p 上的后置递增操作随每次扩展而被应用了两次。执行该程序的结果是下面不正确的计算结果：

```
elem_cnt : 5  expecting: 10
```

函数模板提供了一种机制，通过它我们可以保留函数定义和函数调用的语义（在一个程序位置上封装了一段代码，确保在函数调用之前实参只被计算一次），而无需像宏方案那样绕过 C++的强类型检查。

函数模板提供一个种用来自动生成各种类型函数实例的算法。程序员对于函数接口（参数和返回类型）中的全部或者部分类型进行**参数化**（parameterize），而函数体保持不变。如用一个函数的实现在一组实例上保持不变，并且每个实例都处理一种惟一的数据类型，如函数 min()，则该函数就是模板的最佳候选者。例如，下面是 min()的函数模板定义：

```
template <class Type>
Type min( Type a, Type b ) {
    return a < b ? a : b;
}

int main() {
    // ok: int min( int, int );
    min( 10, 20 );
    // ok: double min( double, double );
    min( 10.0, 20.0 );
```

```
    return 0;
}
```

如果用函数模板代替前面程序中的预处理器宏 min()，则程序的输出是正确的：

```
elem_cnt : 10 expecting: 10
```

［C++标准库为一些常用的算法，如这里定义的 min()，提供了函数模板。这些算法将在第 12 章描述。为了介绍函数模板，我们对于 C++标准库中定义的一些算法给出了相应的简化版本。］

关键字 template 总是放在模板的定义与声明的最前面。关键字后面是用逗号分隔的**模板参数表**（template parameter list），它用尖括号（<>，一个小于号和一个大于号）括起来。该列表是模板参数表，不能为空。模板参数可以是一个**模板类型参数**（template type parameter），它代表了一种类型；也可以是一个**模板非类型参数**（template nontype parameter），它代表了一个常量表达式。

模板类型参数由关键字 class 或 typename 后加一个标识符构成。在函数的模板参数表中，这两个关键字的意义相同。它们表示后面的参数名代表一个潜在的内置或用户定义的类型。模板参数名由程序员选择。在本例中，我们用 Type 来命名 min()的模板参数，但实际上可以是任何名字。譬如：

```
template <class Glorp>
    Glorp min( Glorp a, Glorp b ) {
        return a < b ? a : b;
}
```

当模板被实例化时，实际的内置或用户定义类型将替换模板的类型参数。类型 int、double、char*、vector<int>或 list<double>*都是有效的模板实参类型。

模板非类型参数由一个普通的参数声明构成。模板非类型参数表示该参数名代表了一个潜在的值，而该值代表了模板定义中的一个常量。例如，size 是一个模板非类型参数，它代表 arr 指向的数组的长度：

```
template <class Type, int size>
    Type min( Type (&arr) [size] );
```

当函数模板 min()被实例化时，size 的值会被一个编译时刻已知的常量值代替。

函数定义或声明跟在模板参数表后。除了模板参数是类型指示符或常量值外，函数模板的定义看起来与非模板函数的定义相同。我们来看一个例子：

```
template <class Type, int size>

Type min( const Type (&r_array)[size] )
{
    /* 找到数组中元素最小值的参数化函数 */
    Type min_val = r_array[0];

    for ( int i = 1; i < size; ++i )
        if ( r_array[i] < min_val )
            min_val = r_array[i];

    return min_val;
}
```

在我们的例子中，Type 表示 min()的返回类型、参数 r_array 的类型，以及局部变量 min_val 的类型。size 表示 r_array 引用的数组的长度。在程序的运行过程中，Type 会被各种内置类型和用户定义的类型所代替，而 size 会被各种常量值所取代，这些常量值是由实际使用的 min()决定的（记住，一个函数的两种用法是调用它和取它的地址。）。类型和值的替换过程被称为**模板实例化**（template instantiation）。我们将在下一节介绍模板实例化。

我们的 min()函数模板的函数参数表看起来可能有些短。如 7.3 节所讨论的，一个数组参数总是被作为指向数组首元素的指针来传递，数组实参的第一维在函数定义内是未知的。为了缓解这个问题，我们在此处把 min()的参数声明为数组的引用。这解决了用户必须传递第二个实参来指定数组长度的问题，但是缺点是用在不同长度的 int 数组时，会生成或实例化不同的 min()实例。

当一个名字被声明为模板参数之后，它就可以被使用了，一直到模板声明或定义结束为止。模板类型参数被用作一个类型指示符，可以出现在模板定义的余下部分。它的使用方式与内置或用户定义的类型完全一样，比如用来声明变量和强制类型转换。模板非类型参数被用作一个常量值，可以出现在模板定义的余下部分。它可以用在要求常量的地方，或许是在数组声明中指定数组的大小或作为枚举常量的初始值。

```
// size 指定数组参数的大小并初始化一个 const int 值
template <class Type, int size>
Type min( const Type (&r_array)[size] )
{
    const int loc_size = size;
    Type loc_array[loc_size];
    // ...
}
```

如果在全局域中声明了与模板参数同名的对象、函数或类型，则该全局名将被隐藏。在下面的例子中，tmp 的类型不是 double，是模板参数 Type：

```
typedef double Type;
template <class Type>
Type min( Type a, Type b )
{
    // tmp 类型为模板参数 Type
    // 不是全局 typedef
    Type tmp = a < b ? a : b;
    return tmp;
}
```

在函数模板定义中声明的对象或类型不能与模板参数同名：

```
template <class Type>
Type min( Type a, Type b )
{
    // 错误：重新声明模板参数 Type
    typedef double Type;
    Type tmp = a < b ? a : b;
    return tmp;
}
```

模板类型参数名可以被用来指定函数模板的返回位：

```
// ok: T1 表示 min() 的返回类型
// T2 和 T3 表示参数类型
template <class T1, class T2, class T3>
     T1 min( T2, T3 );
```

模板参数名在同一模板参数表中只能被使用一次。例如，下面代码就有编译错误：

```
// 错误：模板参数名 Type 的非法重复使用
template <class Type, class Type>
     Type min( Type, Type );
```

但是，模板参数名可以在多个函数模板声明或定义之间被重复使用：

```
// ok: 名字 Type 在不同模板之间重复使用
template <class Type>
     Type min( Type, Type );

template <class Type>
     Type max( Type, Type );
```

一个模板的定义和多个声明所使用的模板参数名无需相同。例如，下列三个 min()的声明都指向同一个函数模板：

```
// 三个 min() 的声明都指向同一个函数模板
// 模板的前向声明
template <class T> T min( T, T );
template <class U> U min( U, U );

// 模板的真正定义
template <class Type>
     Type min( Type a, Type b ) { /* ... */ }
```

模板参数在函数参数表中可以出现的次数没有限制。在下面的例子中，Type 用来表示两个不同函数参数的类型：

```
#include <vector>

// ok: 在模板函数的参数表中多次使用 Type
template <class Type>
     Type sum( const vector<Type> &, Type );
```

如果一个函数模板有一个以上的模板类型参数，则每个模板类型参数前面都必须有关键字 class 或 typename。

```
// ok: 关键字 typename 和 class 可以混用
template <typename T, class U>
     T minus( T*, U );

// 错误：必须是 <typename T, class U> 或 <typename T, typename U>
template <typename T, U>
     T sum( T*, U );
```

在函数模板参数表中，关键字 typename 和 class 的意义相同，可以互换使用。它们两个都可以被用来声明同一模板参数表中的不同模板类型参数［就如前面的函数模板 minus()所做的］。看起来，好像用 typename 而不是 class 来指派模板类型参数更为直观，毕竟，关键字 typename 名字能更清楚地表明后面的名字是个类型名。但是，关键字 typename 是最近才被

加入到标准 C++中的，早期的程序员可能更习惯使用关键字 class。（更不用说关键字 class 比 typename 要短一些，人们总是希望少打几个字嘛……）

通过将关键字 typename 加入到 C++中，使得我们可以对模板定义进行分析。这个话题有些过于高深，我们只简要地解释为什么需要关键字 typename。对于想了解更多内容的读者，建议阅读 Stroustrup 的书《Design and Evolution of C++》。

为了分析模板定义，编译器必须能够区分出是类型以及不是类型的表达式。对于编译器来说，它并不总是能够区分出模板定义中的哪些表达式是类型。例如，如果编译器在模板定义中遇到表达式 Parm::name，且 Parm 这个模板类型参数代表了一个类，那么 name 引用的是 Parm 的一个类型成员吗？

```
template <class Parm, class U>
     Parm minus( Parm* array, U value )
{
     Parm::name * p;    // 这是一个指针声明还是乘法？乘法
}
```

编译器不知道 name 是否为一个类型，因为它只有在模板被实例化之后才能找到 Parm 表示的类的定义。为了让编译器能够分析模板定义，用户必须指示编译器哪些表达式是类型表达式。告诉编译器一个表达式是类型表达式的机制是在表达式前加上关键字 typename。例如如果我们想让函数模板 minus()的表达式 Parm::name 是个类型名，因而使整个表达式是一个指针声明，我们应如下修改：

```
template <class Parm, class U>
     Parm minus( Parm* array, U value )
{
     typename Parm::name * p;    // ok: 指针声明
}
```

关键字 typename 也可以被用在模板参数表中，以指示一个模板参数是一个类型。

如同非模板函数一样，函数模板也可以被声明为 inline 或 extern。应该把指示符放在模板参数表后面，而不是在关键字 template 前面。

```
// ok: 关键字跟在模板参数表之后
template <typename Type>
     inline
     Type min( Type, Type );

// 错误: inline 指示符放置的位置错误
inline
template <typename Type>
     Type min( Array<Type>, int );
```

练习 10.1

指出下列函数模板定义中哪些是错误的，并将其改正。

```
(a)  template <class T, U, class V>
     void foo( T, U, V );

(b)  template <class T>
```

```
            T foo( int *T );
(c)   template <class T1, typename T2, class T3>
            T1 foo( T2, T3 );
(d)   inline template <typename T>
            T foo( T, unsigned int* );
(e)   template <class myT, class myT>
            void foo( myT, myT );
(f)   template <class T>
            foo( T, T );
(g)   typedef char Ctype;
      template <class Ctype>
            Ctype foo( Ctype a, Ctype b );
```

练习 10.2

下列模板重复声明中哪些是错的？为什么？

```
(a)   template <class Type>
            Type bar( Type, Type );
      template <class Type>
            Type bar( Type, Type );

(b)   template <class T1, class T2>
            void bar( T1, T2 );
      template <typename C1, typename C2>
            void bar( C1, C2 );
```

练习 10.3

将 7.3.3 小节中给出的函数 putValues()重写为模板函数。并且对函数模板进行参数化，使它有两个模板参数（一个是数组元素的类型，另一个是数组的长度）以及一个函数参数，该函数参数是一个数组的引用。同时给出函数模板定义。

10.2 函数模板实例化

函数模板指定了怎样根据一组或更多实际类型或值构造出独立的函数。这个构造过程被称为模板实例化（template instantiation）。这个过程是隐式发生的，它可以被看作是函数模板调用或取函数模板的地址的副作用。例如，在下面的程序中，min()被实例化两次：一次是针对 5 个 int 的数组类型，另一次是针对 6 个 double 的数组类型。

```
// 函数模板 min() 的定义
// 有一个类型参数 Type 和一个非类型参数 size
```

```
template <typename Type, int size>
     Type min( Type (&r_array)[size] )
{
     Type min_val = r_array[0];
     for ( int i = 1; i < size; ++i )
          if ( r_array[i] < min_val )
               min_val = r_array[i];
     return min_val;
}

// size 没有指定—ok
// size = 初始化表中的值的个数
int ia[] = { 10, 7, 14, 3, 25 };
double da[6] = { 10.2, 7.1, 14.5, 3.2, 25.0, 16.8 };

#include <iostream>
int main()
{
     // 为 5 个 int 的数组实例化 min()
     // Type => int, size => 5
     int i = min( ia );
     if ( i != 3 )
          cout << "??oops: integer min() failed\n";
     else cout << "!!ok: integer min() worked\n";

     // 为 6 个 double 的数组实例化 min()
     // Type => double, size => 6
     double d = min( da );
     if ( d != 3.2 )
          cout << "??oops: double min() failed\n";
     else cout << "!!ok: double min() worked\n";

     return 0;
}
```

调用

```
int i = min( ia );
```

被实例化为下面的 min()的整型实例，这里 Type 被 int、size 被 5 取代：

```
int min( int (&r_array)[5] )
{
     int min_val = r_array[0];
     for ( int ix = 1; ix < 5; ++ix )
          if ( r_array[ix] < min_val )
               min_val = r_array[ix];

     return min_val;
}
```

类似地，调用

```
double d = min( da );
```

也实例化了 min()的实例，这里 Type 被 double，size 被 6 取代。

类型参数 Type 和非类型参数 size 都被用作函数参数。为了判断用作模板实参的实际类型和值，编译器需要检查函数调用中提供的函数实参的类型。在我们的例子中，ia 的类型（即 5 个 int 的数组）和 da 的类型（即 6 个 double 的数组）被用来决定每个实例的模板实参。用函数实参的类型来决定模板实参的类型和值的过程被称为**模板实参推演**（template argument deduction）。我们将在下节更详细地介绍模板实参推演。我们也可以不依赖模板实参推演过程，而是显式地指定模板实参。我们将在 10.4 节了解怎样实现这种方式。）

函数模板在它被调用或取其地址时被实例化。在下面的例子中，指针 pf 被函数模板实例的地址初始化。编译器通过检查 pf 指向的函数的参数类型来决定模板实例的实参。

```
template <typename Type, int size>
     Type min( Type (&p_array)[size] ) { /* ... */ }

// pf 指向 int min( int (&)[10] )
int (*pf)(int (&)[10]) = &min;
```

pf 的类型是指向函数的指针，该函数有一个类型为 int(&)[10]的参数。当 min()被实例化时，该参数的类型决定了 Type 的模板实参的类型和 size 的模板实参的值。Type 的模板实参为 int，size 的模板实参为 10。被实例化的函数是 min(int(&)[10])，指针 pf 指向这个模板实例。

在取函数模板实例的地址时，必须能够通过上下文环境为一个模板实参决定一个惟一的类型或值。如果不能决定出这个惟一的类型或值，就会产生编译时刻错误。例如：

```
template <typename Type, int size>
     Type min( Type (&r_array)[size] ) { /* ... */ }
typedef int (&rai)[10];
typedef double (&rad)[20];

void func( int (*)(rai) );
void func( double (*)(rad) );

int main() {
     // 错误：哪一个 min() 的实例?
     func( &min );
}
```

因为函数 func()被重载了，所以编译器不可能通过查看 func()的参数类型，来为模板参数 Type 决定惟一的类型，以及为 size 的模板实参决定一个惟一值。调用 func()无法实例化下面的任何一个函数：

```
min( int (*)(int(&)[10]) )
min( double (*)(double(&)[20]) )
```

因为不可能为 func()指出一个惟一的实参的实例，所以在该上下文环境中取函数模板实例的地址会引起编译时刻错误。

如果我们用一个强制类型转换显式地指出实参的类型则可以消除编译时刻错误：

```
int main() {
     // ok: 强制转换指定实参类型
     func( static_cast< double(*)(rad) >(&min) );
}
```

更好的方案是用显式模板实参，这一点我们将在 10.4 节中说明。

10.3 模板实参推演 ※

当函数模板被调用时，对函数实参类型的检查决定了模板实参的类型和值、这个过程被称为**模板实参推演**（template argument deduction）。

函数模板 min()的函数参数是一个引用，它指向了一个 Type 类型的数组：

```
template <class Type, int size>
    Type min( Type (&r_array)[size] ) { /* ... */ }
```

为了匹配函数参数，函数实参必须也是一个表示数组类型的左值。下面的调用是个错误，因为 pval 是 int*类型而不是 int 数组类型的左值。

```
void f( int pval[9] ) {
    // 错误: Type (&)[] != int*
    int jval = min( pval );
}
```

在模板实参推演期间决定模板实参的类型时，编译器不考虑函数模板实例的返回类型。例如，对于如下的 min()调用：

```
double da[8] = { 10.3, 7.2, 14.0, 3.8, 25.7, 6.4, 5.5, 16.8 };
int i1 = min( da );
```

min()的实例有一个参数，它是一个指向 8 个 double 的数组的指针。出该实例返回的值的类型是 double 型。该返回值先被转换成 int 型，然后再用来初始化 i1。即使调用 min()的结果被用来初始化一个 int 型的对象，也不会影响模板实参的推演过程。

要想成功地进行模板实参推演，函数实参的类型不一定要严格匹配相应函数参数的类型。下列三种类型转换是允许的：左值转换、限定转换和到一个基类（该基类根据一个类模板实例化而来）的转换。让我们依次来看一看。

左值转换包括从左值到右值的转换、从数组到指针的转换或从函数到指针的转换（这些转换在 9.3 节中介绍）。为说明左值转换是怎样影响模板实参推演过程的，让我们考虑函数 min2()，它有一个名为 Type 的模板参数以及两个函数参数。min2()的第一个函数参数是一个 Type*型的指针。而 size 则不再像在 min()中定义的是个模板参数。size 变成一个函数参数当 min2()被调用时，我们必须显式地为它指定一个函数实参值：

```
template <class Type>
// 第一个参数是 Type*
Type min2( Type* array, int size )
{
    Type min_val = array[0];
    for ( int i = 1; i < size; ++i )
        if ( array[i] < min_val )
            min_val = array[i];
    return min_val;
}
```

我们可以用 4 个 int 的数组来作为第一个实参调用 min2()，如下：

```
int ai[4] = { 12, 8, 73, 45 };
int main() {
    int size = sizeof (ai) / sizeof (ai[0]);

    // ok: 从数组到指针的转换
    min2( ai, size );
}
```

函数实参 ai 的类型是 4 个 int 的数组，虽然这与相应的函数参数类型 Type*并不严格匹配。但是因为允许从数组到指针的转换，所以实参 ai 在模板实参 Type 被推演之前被转换成 int*型。Type 的模板实参接着被推演为 int，最终被实例化的函数模板是 min2(int*,int)。

限定修饰转换把 const 或 volatile 限定修饰符加到指针上（限定修饰转换在 9.3 节中介绍）。为说明限定修饰转换是怎样影响模板实参推演过程的，我们考虑函数 min3()，它的第一个函数参数的类型是 const Type*：

```
template <class Type>
    // 第一个参数是 const Type*
    Type min3( const Type* array, int size ) {
        // ...
    }
```

我们可以用 int*型的第一个参数调用 min3()，如下：

```
int *pi = &ai;

// ok: 到 const int* 的限定修饰转换
int i = min3( pi, 4 );
```

函数实参 pi 的类型是 int 指针，虽然与相应的函数参数类型 const Type*并不完全匹配。但是因为允许限定修饰转换，所以函数实参在模板实参被推演之前，就先被转换为 const Type*型了。然后 Type 的模板实参被椎演为 int，被实例化的函数模板是 min3(const int*, int)。

现在，再让我们来看看到一个基类（该基类根据一个类模板实例化而来）的转换。如果函数参数的类型是一个类模板，且如果实参是一个类，它有一个从被指定为函数参数的类模板实例化而来的基类，则模板实参的推演就可以进行。为说明这个转换，我们使用一个新的被称为 min4()的函数模板，它有一个类型为 Array<Type>&的参数（这里的 Array 是在 2.5 节中定义的类模板）。（第 16 章将给出类模板的完全讨论）。

```
template <class Type>
    class Array { /* ... */ };

template <class Type>
    Type min4( Array<Type>& array )
{
    Type min_val = array[0];
    for ( int i = 1; i < array.size(); ++i )
        if ( array[i] < min_val )
            min_val = array[i];
    return min_val;
}
```

我们可以用类型为 ArrayRC<int>的第一个实参调用 min4()（ArrayRC 也是第 2 章定义的

类模板。类继承将在 17 章和 18 章中讨论）。如下：

```
template <class Type>
    class ArrayRC : public Array<Type> { /* ... */ };

int main() {
    ArrayRC<int> ia_rc( ia, sizeof(ia)/sizeof(int) );
    min4( ia_rc );
}
```

函数实参 ia_rc 的类型是 ArrayRC<int>，它与相应的函数参数类型 Array<Type>&并不完全匹配。因为类 ArrayRC<int>有一个 Array<int>的基类，而 Array<int>是一个从被指定为函数参数的类模板实例化而来的类，并且派生类类型的函数实参还可以被用来推演一个模板实参，所以函数实参 ArrayRC<int>在模板实参被推演之前首先被转换成 Array<int>型，然后 Type 的模板实参再被推演为 int，被实例化的函数模板是 min4(Array<int>&)。

多个函数实参可以参加同一个模板实参的推演过程。如果模板参数在函数参数表中出现多次，则每个推演出来的类型都必须与根据模板实参推演出来的第一个类型完全匹配。例如

```
template <class T> T min5( T, T ) { /* ... */ }
unsigned int ui;

int main() {
    // 错误：不能实例化 min5( unsigned int, int )
    // 必须是：min( unsigned int, unsigned int ) 或
    // min( int, int )
    min5( ui, 1024 );
}
```

mins()的函数实参必须类型相同（要么都是 int，要么都是 unsigned int)，这是因为模板参数 T 必须被绑定在一个类型上。从第一个函数实参推演出的 T 的模板实参是 int，而从第二个函数实参推演出的是 unsigned int。因为对于两个函数实参，模板实参 T 的类型被推演成不同类型，所以模板实参推演将失败，并且模板实例化也会出错误。（一种解决办法是在调用 min5()时显式指定模板实参。我们将在 10.4 节介绍怎样实现它。）

这些可能的类型转换的限制只适用于参加模板实参推演过程的函数实参。对于所有其他实参，所有的类型转换都是允许的。下面的函数模板 sum()有两个参数。针对第一个参数的实参 op1 参与模板实参推演过程，而针对第二个参数的实参 op2 则没有参与。

```
template <class Type>
    Type sum( Type op1, int op2 ) { /* ... */ }
```

因为第二个实参不参与模板实参推演过程，所以当函数模板 sum()的实例被调用时。可以在第二个实参上应用任何类型转换。（9.3 节描述了可以应用在函数实参上的类型转换。）例如：

```
int ai[] = { ... };
double dd;

int main() {
    // sum( int, int ) 被实例化
    sum( ai[0], dd );
}
```

第一个函数实参 dd 的类型与相应的函数参数类型 int 不匹配。但是，对函数模板 sum() 实例的调用不是错的，这是因为第二个实参的类型是固定的，不依赖于模板参数。对于该调用，函数 sum(int,int)被实例化。实参 dd 被通过浮点—有序标准转换转换成类型 int。

所以模板实参推演的通用算法如下：

1．依次检查每个函数实参，以确定在每个函数参数的类型中出现的模板参数。

2．如果找到模板参数，则通过检查函数实参的类型，推演出相应的模板实参。

3．函数参数类型和函数实参类型不必完全匹配。下列类型转换可以被应用在函数实参上，以便将其转换成相应的函数参数的类型：

- 左值转换。
- 限定修饰转换。
- 从派生类到基类类型的转换。假定函数参数具有形式 T<args>、T<args>&或 T<args>*，则这里的参数表 args 至少含有一个模板参数。

4．如果在多个函数参数中找到同一个模板参数，则从每个相应函数实参推演出的模板实参必须相同。

练习 10.4

指出在模板实参推演过程中涉及到的函数实参时两种可行的类型转换。

练习 10.5

已知下列模板定义：

```
template <class Type>
     Type min3( const Type* array, int size ) { /* ... */ }
template <class Type>
     Type min5( Type p1,Type p2 ) { /* ... */ }
```

下列哪些调用是错误的？为什么？

```
double dobj1, dobj2;
float fobj1, fobj2;
char cobj1, cobj2;
int ai[5] = { 511, 16, 8, 63, 34 };
(a) min5( cobj2, 'c' );
(b) min5( dobj1, fobj1 );
(c) min3( ai, cobj1 );
```

10.4 显式模板实参 ※

在某些情况下编译器不可能推演出模板实参的类型。如在上节函数模板 min5()的例子中所看到的，如果模板实参推演过程为同一模板实参推演出两个不同的类型，则编译器会给出一个错误，指出模板实参推演失败。

在这种情况下，我们需要改变模板实参推演机制，并使用**显式指定**（explicitly specify）模板实参。模板实参被显式指定在逗号分隔的列表中，用尖括号（<>，一个小于号和一个

大于号）括起来，紧跟在函数模板实例的名字后面。例如，在我们前面使用的 min5()中，假定希望 T 的模板实参是 unsigned int，则函数模板实例 min5()的调用可以重写如下：

```
// min5( unsigned int, unsigned int ) 被实例化
min5< unsigned int >( ui, 1024 );
```

在这种情况下，模板实参表<unsigned int>显式地指定了模板实参的类型。因为模板实参已知，所以函数调用不再是一个错误。

注意，在调用函数 min5()时，第二个函数实参是 1024，它的类型是 int。因为第二个函数参数的类型通过显式模板实参已被固定为 unsigned int，所以第二个函数实参通过有序标准转换被转换成类型 unsigned int。

在上节中我们看到，能在函数实参上进行的只是类型转换是有限的。从 int 到 unsigned int 的有序标准转换就不允许。但是当模板实参被显式指定时，就没有必要推演模板实参了。函数参数的类型已经固定。当函数模板实参被显式指定时，把函数实参转换成相应函数参数的类型可以应用任何隐式类型转换。

除了允许在函数实参上的类型转换，显式模板参数还为解决其他的程序设计问题提供了方案。考虑下面的问题。我们希望定义一个名为 sum()的函数模板，以便从该模板实例化的函数可以返回某一种类型的值，该类型足够大，可以装下两种以任何顺序传递来的任何类型的两个值的和。我们该怎样做呢？应该指定 sum()的返回类型吗？

```
// 以 T 或 U 作为返回类型?
template <class T, class U>
    ??? sum( T, U );
```

在我们的例子中，答案是两个参数都不用。因为使用它们中的任何一个都会导致在某点上失败：

```
char ch; unsigned int ui;

// T 和 U 都不用作返回类型
sum( ch, ui );     // ok: U sum( T, U );
sum( ui, ch );     // ok: T sum( T, U );
```

一种方案是通过引入第三个模板参数来指明函数模板的返回类型。

```
// T1 不出现在函数模板参数表中
template <class T1, class T2, class T3>
    T1 sum( T2, T3 );
```

因为返回类型可能与函数实参类型不同，所以 T1 在函数参数表中不再被提起。这是一个潜在的问题，因为 T1 的模板实参不能从函数实参中被推演出来。但是，如果在 sum()的一个实例调用中给出一个显式模板实参。我们就能避免编译器错误地指出 T1 的模板实参不能被推演出来。例如：

```
typedef unsigned int ui_type;
ui_type calc( char ch, ui_type ui ) {

    // ...

    // 错误: T1 不能被推演出来
    ui_type loc1 = sum( ch, ui );
```

```
        // ok: 模板实参被显式指定
        // T1 和 T3 是 unsigned int, T2 是 char
        ui_type loc2 = sum< ui_type, char, ui_type >( ch, ui );
    }
```

我们真正期望的是，为 T1 指定一个显式模板实参，而省略 T2 和 T3 的显式模板实参。T2、T3 的模板实参可以从该调用的函数实参中推演出来。

在显式特化（explicit specification）中，我们只需列出不能被隐式推演的模板实参，如同缺省实参一样，我们只能省略尾部的实参。例如：

```
    // ok: T3 是 unsigned int
    // T3 从 ui 的类型中推演出来
    ui_type loc3 = sum< ui_type, char >( ch, ui );

    // ok: T2 是 char, T3 是 unsigned int
    // T2 和 T3 从 pf 的类型中推演出来
    ui_type (*pf)( char, ui_type ) = &sum< ui_type >;

    // 错误：只能省略尾部的实参
    ui_type loc4 = sum< ui_type, , ui_type >( ch, ui );
```

在其他情形下，编译器不可能从使用函数模板实例的上下文中推演出模板实参。没有显式模板实参。在这种上下文环境中就不可能使用函数模板实例。我们必须意识到需要支持这些清形，这导致了标准 C++对显式模板实参提供了支持。在下面的例子中，sum()实例的地址被取出，并作为重载函数 manipulate()调用的实参被传递。如在 10.2 节中所述，想只通过查看 manipulate()的参数表就能选择出作为实参传递的 sum()实例，这是不可能的。sum()的两个不同实例都可以被实例化，并满足该调用。但 manipulate()的调用是二义的。消除调用二义性的一个方案是提供一个显式强制类型转换，而更好的方案是使用显式模板实参。显式模板实参指明 sum()的哪个实例被使用，以及哪个 manipulate()被调用。例如：

```
template <class T1, class T2, class T3>
T1 sum( T2 op1, T3 op2 ) { /* ... */ }
void manipulate( int (*pf)( int,char ) );
void manipulate( double (*pf)( float,float ) );
int main( )
{
    // 错误：哪一个 sum 的实例?
    // int sum( int, char ) 还是
    // double sum( float, float ) ?
    manipulate( &sum );

    // 取实例: double sum( float, float )
    // 调用: void manipulate( double (*pf)( float, float ) );
    manipulate( &sum< double, float, float > );
}
```

我们必须指出，显式模板实参应该只被用在完全需要它们来解决二义性，或在模板实参不能被推演出来的上下文中使用模板实例时。首先，让编译器来决定模板实参的类型和值是比较容易的。其次，如果我们通过修改程序中的声明来改变在函数模板实例调用中的函数实参的类型，则编译器会自动用不同的模板实参实例化函数模板，而无需我们做任何事情。另

一方面，如果我们指定了显式模板参数，则必须检查显式模板实参对于函数实参的新类型是否仍然合适。所以建议在可能的时候省略显式模板实参。

练习 10.6

指出两种必须使用显式模板实参的情形。

练习 10.7

已知下面 sum()的模板定义

```
template <class T1, class T2, class T3>
     T1 sum( T2, T3 );
```

下列哪些调用是错误的，为什么？

```
double dobj1, dobj2;
float fobj1, fobj2;
char cobj1, cobj2;

(a) sum( dobj1, dobj2 );
(b) sum<double,double,double>( fobj1, fobj2 );
(c) sum<int>( cobj1, cobj2 );
(d) sum<double, ,double>( fobj2, dobj2 );
```

10.5 模板编译模式 ※

函数模板的定义可用来作为一个无限个函数实例集合定义的规范描述（prescription）。模板本身不能定义任何函数。例如，当编译器实现看到下面的模板定义时

```
template <typename Type>
     Type min( Type t1, Type t2 )
{
     return t1 < t2 ? t1 : t2;
}
```

它就保存了 min()的内部表示形式，但是不会使任何其他事情发生。后来，当它看到 min()被实际使用时，如

```
int i, j;
double dobj = min( i, j );
```

才根据模板定义为 min()实例化一个整型的定义。

这带来了几个问题：如果希望编译器能够实例化函数模板，那么函数模板 min()的定义必须在实例被调用之前就可见吗？例如，在上面的例子中，min()的整型实例被用在 dobj 的定义中，那么在此之前函数模板 min()的定义必须先出现吗？我们把函数模板定义放在头文件中（就好像对内联函数定义的做法一样），在使用函数模板实例的地方包含它们？或者我们只在头文件中给出函数模板声明，而把模板定义放在文本文件中（就好像对非内联函数的做法一样）？

为了回答这些问题，我们必须解释 **C++模板编译模式**（template compilation model），它

指定了对于定义和使用模板的程序的组织方式的要求。C++支持两种模板编译模式：**包含模式**（Inclusion Model）和**分离模式**（Separation Model）。本节余下部分将分别描述这两种模式，并具体说明它们的用法。

10.5.1 包含编译模式

在包含编译模式下，我们在每个模板被实例化的文件中包含函数模板的定义，并且往往把定义放在头文件中，像对内联函数所做的那样。我们已经把这种模式选为本书使用的模式，例如：

```
// model1.h
// 包含模式：模板定义放在头文件中
template <typename Type>
     Type min( Type t1, Type t2 ) {
          return t1 < t2 ? t1 : t2;
}
```

在每个使用 min()实例的文件中都包含了该头文件，例如：

```
// 在使用模板实例之前包含模板定义
#include "model1.h"
int i, j;
double dobj = min( i, j );
```

该头文件可以被包含在许多程序文本文件中。这意味着编译器必须在每个调用该实例的文件中实例化 min()的整型实例吗？不。该程序必须表现得好像 min()的整型实例只被实例化一次。但是，真正的实例化动作发生在何时何地，要取决于具体的编译器实现。现在就我们所关心的来说，我们只需要知道 min()的整型实例在程序中的某个地方被实例化。（如我们在本节结束时将看到的，用显式实例化声明可指定在何时何地进行模板实例化。这样的声明有时候必须在产品开发的后期被使，以来改善应用程序的性能。）

在头文件中提供函数模板定义有几个缺点。函数模板体（body）描述了实现细节，对于这些细节，用户可能想忽略，或者我们希望隐藏起来不让用户知道。实际上，如果函数模板的定义非常大，那么在头文件中给出的细节层次有可能是不可接受的。而且，在多个文件之间编译相同的函数模板定义增加了不必要的编译时间。分离编译模式允许我们分离函数模板的声明和定义，下面让我们看一下怎样使用它。

10.5.2 分离编译模式

在分离编译模式下，函数模板的声明被放在头文件中。在这种模式下，函数模板声明和定义的组织方式与程序中的非内联函数的声明和定义组织方式相同。例如：

```
// model2.h

// 分离模式：只提供模板声明
template <typename Type> Type min( Type t1, Type t2 );

// model2.C
// the template definition
export template <typename Type>
```

```
Type min( Type t1, Type t2 ) { /* ...*/ }
```

使用函数模板 min()实例的程序只需在使用该实例之前包含这个头文件：

```
// user.C
#include "model2.h"

int i, j;
double d = min( i, j ); // ok: 用法，需要一个实例
```

即使 min()的模板定义在 user.C 中不可见，仍然可以在这个文件中调用模板实例 min(int,int)。但是，为了实现它，模板 min()必须以一种特殊的方式被定义。你知道怎样做吗？如果仔细看一下定义了函数模板 min()的文件 model2.C，你就会注意到在模板定义中有一个关键字 export。模板 min()被定义成一个可导出的（exported）模板。关键字 export 告诉编译器在生成被其他文件使用的函数模板实例时可能需要这个模板定义。编译器必须保证，在生成这些实例时，该模板定义是可见的。

我们通过在模板定义中的关键字 template 之前加上关键字 export，来声明一个可导出的函数模板。当函数模板被导出时，我们就可以在任意程序文本文件中使用模板的实例，而我们所需要做的就是在使用之前声明该模板。如果省略了模板 min()定义中的关键字 export，则编译器实现可能不能实例化函数模板 min()的整型实例，而我们将不能正确链接我们的程序。

注意，有些编译器实现可能不要求用关键字 export。有些实现可能支持下列语言扩展：非导出的函数模板定义可能只出现在一个程序文本文件中，在其他程序文本文件中用到的实例仍然被正确地实例化。但是，这种行为只是一个扩展。如果在模板被实例化之前只有函数模板的声明在程序文本文件中可见，那么，标准 C++要求用户把函数模板定义标记为 export。

关键字 export 不需要出现在头文件的模板声明中。在 model2.h 中的 min()的声明中没有指定关键字 export。此关键字也可以出现在该声明中，但不是必需的。

在程序中，一个函数模板只能被定义为 export 一次。不幸的是，因为编译器每次只处理一个文件，所以它不能检测到一个函数模板在多个文本文件中被定义为 export 的情况。如果发生了这样的事情，下列行为就有可能随之发生：

1．可能产生一个链接错误，指出函数模板在多个文件中被定义。

2．编译器可能不只一次地为同一个模板实参集合实例化该函数模板，由于函数模板实例的重复定义，这会引起链接错误。

3．编译器可能用其中一个 export 函数模板定义来实例化函数模板，而忽略其他定义。

所以，在程序中提供多个 export 函数模板的定义不一定会产生错误。我们必须小心谨慎地组织程序，以便把 export 函数模板定义只放在一个程序文本文件中。

分离模式使我们能够很好地将函数模板的接口同其实现分开，进而组织好程序，以便把函数模板的接口放到头文件中，而把实现放在文本文件中。但是，并不是所有的编译器都支持分离模式，即使支持也未必总能支持得很好。支持分离模式需要更复杂的程序设计环境，所以它们不能在所有 C++编译器实现中提供。[21]

对于本书的目的而言，因为模板例子非常小，而且我们希望这些例子在许多 C++实现中

[21] 本书的姊妹篇《Inside the C++ Object Model》，描述了一个 C++编译器“the Edison Design Group compiler”支持的模板实例化机制。该书的中文简体版已经出版。

都能够方便地被编译，所以我们只使用包含模式。

10.5.3 显式实例化声明

当我们使用包含模式时，每个使用模板实例的程序文本文件都要包含函数模板的定义。我们已经看到：尽管程序不能确切地知道编译器在何时何地实例化函数模板，但是它必须表现得好像对一个特定的模板实参集合只实例化一次模板。在实际中，有些编译器（尤其是老的 C++编译器）对特殊的模板实参集合会多次实例化函数模板。在这种模式下，程序会选择这些实例中的一个作为最终的实例（当程序被链接或在某个预链接阶段）。而其他实例只是被简单地忽略掉。

无论函数模板被实例化一次还是多次，程序结果都不会受到影响，因为最终只有一个模板实例被程序使用。但是，如果函数模板被多次实例化，则程序的编译时间性能可能会受到很大影响。如果应用程序由许多文件构成，并且所有这些文件中的模板都被实例化，那么编译应用程序所需要的时间会显著地增加。

早期编译器的实例化问题使得模板用起来非常困难。为了解决这个问题，标准 C++提供了显式实例化声明来帮助程序员控制模板实例化发生的时间。

在显式实例化声明中，关键字 template 后面是函数模板实例的声明，其中显式地指定了模板实参。在下面的例子中，提供了 sum(int*,int)的显式实例化声明。

```
template <typename Type>
    Type sum( Type op1, int op2 ) { /* ... */ }

// 显式实例化声明
template int* sum< int* >( int*, int );
```

该显式实例化声明要求用模板实参 int*实例化模板 sum()。对于给定的函数模板实例。显式实例化声明在一个程序中只能出现一次。

在显式实例化声明所在的文件中，函数模板的定义必须被给出。如果该定义不可见，则该显式实例化声明是错误的。

```
#include <vector>
template <typename Type>
    Type sum( Type op1, int op2 ); // declaration only

// 声明一个 typedef 引用 vector< int >
typedef vector< int > VI;

// 错误：sum() 没有定义
template VI sum< VI >( VI , int );
```

当一个显式实例化声明在程序文本文件中出现时，在其他使用该函数模板实例的文件中又发生了什么事情？我们怎样告诉编译器，一个显式实例化声明已在另一个文件中出现，而在程序其他文件使用该模板函数时不能再对它实例化？

显式实例化声明是与另外一个编译选项联合使用的，该选项压制了程序中模板的隐式实例化。选项的名称随着编译器不同而小同。例如，对 IBM 编译器 Visual Age for C++ for Windows 版本 3.5，压制模板隐式实例化的选项为/ft-。当我们用这个选项编译应用程序时，

编译器假定我们将会用显式实例化声明处理模板实例，所以它不会隐式地实例化应用程序用到的模板。

当然，如果我们不为一个函数实例提供显式实例化声明，则在编译程序时指定选项/ft-就会产生一个链接错误，认为缺少函数模板实例化的定义。在这种情况下，模板不会被隐式地实例化。

练习 10.8

指出 C++支持的两种模板编译模式，并解释在这些模板编译模式下怎样组织含有函数模板定义的程序。

练习 10.9

给出下列 sum()的模板定义：

```
template <typename Type>
     Type sum( Type op1, char op2 );
```

怎样为 string 类型的模板实参声明一个显式实例化声明？

10.6 模板显式特化 ※

我们并不总是能够写出对所有可能被实例化的类型都是最合适的函数模板。在某些情况下，我们可能想利用类型的某些特性，来编写一些比模板实例化的函数更高效的函数。在有些时候，一般性的模板定义对于某种类型来说并不适用。例如，假设我们有函数模板 max()的定义：

```
// 通用的模板定义
template <class T>
     T max( T t1, T t2 ) {
          return (t1 > t2 ? t1 : t2);
}
```

如果函数模板用 const char*型的模板实参实例化，并且我们还想让每个实参都被解释为 C 风格的字符串，而不是字符的指针，则通用模板定义给出正确的语义就不正确了。为了获得正确的语义，我们必须为函数模板实例化提供特化的定义。

在**模板显式特化定义**（explicit specialization definition）中，先是关键字 template 和一对尖括号（<>，一个小于号和一个大于号），然后是函数模板特化的定义。该定义指出了模板名、被用来特化模板的模板实参，以及函数参数表和函数体。在下面的例子中，为 max(const char*, const char*)定义了一个显式特化：

```
#include <cstring>

// const char* 显式特化:
// 覆盖了来自通用模板定义的实例

typedef const char *PCC;
template<> PCC max< PCC >( PCC s1, PCC s2 ) {
```

```
    return ( strcmp( s1, s2 ) > 0 ? s1 : s2 );
}
```

由于有了这个显式特化，当在程序中调用函数 max(const char*,const char*)时，模板不会用类型 const char*来实例化。对所有用两个 const char*型实参进行调用的 max()，都会调用这个特化的定义。而对于其他的调用，根据通用模板定义实例化一个实例，然后再调用它。这些函数可能的调用如下：

```
#include <iostream>

// 函数模板 max() 的定义以及对 const char* 的特化

int main() {
    // 调用实例: int max< int >( int, int );
    int i = max( 10, 5 );

    // 调用显式特化: const char* max< const char* >( const char*, const char* );
    const char *p = max( "hello", "world" );

    cout << "i: " << i << " p: " << p << endl;
    return 0;
}
```

我们也可以声明一个函数模板的显式特化而不定义它。例如，函数 max(const char*,const char*)的显式特化可以被声明如下：

```
// 函数模板显式特化的声明
template<> PCC max< PCC >( PCC, PCC );
```

在声明或定义函数模板显式特化时，我们不能省略显式特化声明中的关键字 template 及其后的尖括号。类似地，函数参数表也不能从特化声明中省略掉。

```
// 错误: 无效的特化声明

// 缺少 template<>
PCC max< PCC >( PCC, PCC );

// 缺少函数参数表
template<> PCC max< PCC >;
```

但是，如果模板实参可以从函数参数中推演出来，则模板实参的显式特化可以从显式特化声明中省略。

```
// ok: 模板实参 const char* 可以从参数类型中推演出来
template<> PCC max( PCC , PCC );
```

在下面的例子中，函数模板 sum()被显式特化：

```
template <class T1, class T2, class T3>
    T1 sum( T2 op1, T3 op2 );

// 显式特化声明

// 错误: T1 的模板实参不能被推演出来
//       它必须显式指定
```

```
template<> double sum( float, float );

// ok: T1 的实参被显式指定
// T2 和 T3 可以从 float 推演出来
template<> double sum<double>( float, float );

// ok: 所有实参都显式指定
template<> int sum<int,char,char>( char , char );
```

省略显式特化声明中的 template<>并不总是错的。例如：

```
// 通用模板定义
template <class T>
T max( T t1, T t2 ) { /* ... */ }

// ok: 普通函数定义
const char* max( const char*, const char* );
```

但是，max()的声明并没有声明函数模板特化。它只是用与模板实例相匹配的返回值和参数表声明了一个普通函数。声明一个与模板实例相匹配的普通函数并不是个错误。

为什么我们要声明一个与模板实例相匹配的普通函数而不声明一个显式特化呢？如在 9.3 节所见到的，如果该实参参与模板实参的推演过程，则只能应用有限的一些类型转换把函数模板实例的实参转换成相应的函数参数类型。函数模板被显式特化的情形也是这样，只有 10.3 节描述的那些类型转换才可以被应用在函数模板显式特化的实参上。显式特化并没有帮助我们越过类型转换的限制。如果想进行该类型转换集合之外的转换，那么就必须定义一个普通函数而不是一个函数模板的特化。10.8 节将更详细地介绍，并给出怎样解析一个与普通函数和函数模板实例都匹配的调用。

即使函数模板显式特化所指定的函数模板只有声明而没有定义，我们仍然可以声明函数模板显式特化。在前面的例子中，函数模板 sum()在被特化之前只是被声明了一下。尽管不需要函数定义，但是模板声明也还是需要的。在名字 sum()被特化之前，编译器必须知道它是个模板。

在源文件中，使用函数模板显式特化之前，必须先进行声明。例如：

```
#include <iostream>
#include <cstring>

// 通用模板定义
template <class T>
T max( T t1, T t2 ) { /* ... */ }
int main() {
    // const char* max<const char*>( const char*, const char* ) 的实例
    // 使用通用模板定义
    const char *p = max( "hello", "world" );

    cout << " p: " << p << endl;
    return 0;
}
```

```
// 无效程序: const char* 显式特化覆盖了通用模板函数
typedef const char *PCC;
     template<> PCC max< PCC >( PCC s1, PCC s2 ) { /* ... */ }
```

因为上面的例子在声明显式特化之前使用了 max(const char*,const char*)的实例。所以，编译器只好假定该函数需要从通用模板定义中实例化。但是，一个程序不能对相同的模板实参集的同一模板同时有一个显式特化和一个实例。当在程序文本文件中再遇到 max(const char*,const char*)的显式特化时，编译器会提示一个编译时刻错误。

如果程序由一个以上的文件构成，则模板显式特化的声明必须在使用该特化的每个文件中都可见。像下面这样的情况是不允许的：在有些文件中，函数模板被根据通用模板定义实例化，而在其他文件中，对同一模板实参的集合却被特化。考虑下面的例子：

```
// ---- max.h ----
// 通用模板定义
template <class Type>
     Type max( Type t1, Type t2 ) { /* ... */ }

// ---- File1.C ----
#include <iostream>
#include "max.h"
void another();
int main() {
     // const char* max<const char*>( const char*, const char* )
     // 的实例
     const char *p = max( "hello", "world" );

     cout << " p: " << p << endl;
     another();
     return 0;
}

// ---- File2.C ----
#include <iostream>
#include <cstring>
#include "max.h"

// const char* 的模板显式特化
typedef const char *PCC;
     template<> PCC max< PCC >( PCC s1, PCC s2 ) { /*... */ }

void another() {
     // const char* max< const char* >( const char*, const char* )
     // 的显式特化;
     const char *p = max( "hi", "again" );

     cout << " p: " << p << endl;
     return 0;
}
```

上面的程序由两个文件构成。在 File1.C 中，没有显式特化 max(const char*,const char*)

的声明，函数模板被根据通用模板定义实例化。在 File2.C 中声明了显式特化，调用 max("hi", "again")就会调用该显式特化。因为同一程序在一个文件中实例化了函数模板实例 max(const char*,const char*)，而在另一个文件中又调用了这显式特化，所以这个程序是非法的。为了补救这个问题，显式特化的声明必须在文件 File1.C 中 max(const char*,const char*)调用之前被给出。

为了防止出现这样的错误，并确保模板显式特化 max(const char*,const char*)的声明被包含在每个用类型 const char*型实参调用函数模板 max()的文件中，显式特化的声明应该被放在头文件“max.h”中，并在所有使用函数模板 max()的程序中包含这个文件：

```
// ---- max.h ----
// 通用模板定义
template <class Type>
     Type max( Type t1, Type t2 ) { /* ... */ }

// const char* 模板显式特化的声明
typedef const char *PCC;
     template<> PCC max< PCC >( PCC s1, PCC s2 );

// ---- File1.C ----
#include <iostream>
#include "max.h"

void another();
int main() {
     // const char* max<const char*>( const char*, const char* ) 的特化;
     const char *p = max( "hello", "world" );

     //....
}
```

练习 10.10

请定义一个函数模板 count()，以记录数组中某个值出现的次数。写个程序调用它，并按顺序向其传递 double、int 和 char 型的数组，以及引入 count()函数的一个特化模板实例来处理字符串。最后，再重新运行调用该函数模板实例的程序。

10.7 重载函数模板 ※

函数模板可以被重载。例如，下面给出函数模板 min()的三个有效的重载声明：

```
// 类模板 Array 的定义
// (introduced in Section 2.4)
template <typename Type>
     class Array{ /* ... */ };

// min() 的三个函数模板声明
```

```
template <typename Type>
    Type min( const Array<Type>&, int ); // #1

template <typename Type>
    Type min( const Type*, int );        // #2

template <typename Type>
    Type min( Type, Type );              // #3
```

下面的 main()定义说明了这三个 min()声明可以被怎样调用：

```
#include <cmath>

int main()
{
    Array<int> iA(1024);// 类实例
    int ia[1024];

    // Type == int; min( const Array<int>&, int )
    int ival0 = min( iA, 1024 );

    // Type == int; min( const int*, int )
    int ival1 = min( ia, 1024 );

    // Type == double; min( double, double )
    double dval0 = min( sqrt( iA[0] ), sqrt( ia[0] ) );

    return 0;
}
```

当然，成功地声明一组重载函数模板并不能保证它们可以被成功地调用。在调用一个模板实例时，重载的函数模扳可能会导致二义性。下面是将出现这种二义性的一个例子。我们前面曾讲到，对于下列 min5()的模板定义：

```
template <typename T>
    int min5( T, T ) { /* ... */ }
```

即使用不同类型的实参调用 min5()，编译器也不能根据模板定义实例化函数。因为函数实参推演过程为 T 推演出两个不同的类型，所以模板实参推演失败，调用是错误的。

```
int i;
unsigned int ui;

// ok: 为 T 推演出: int
min5( 1024, i );

// 模板实参推演失败
// 为 T 推演出两个不同的类型
min5( i, ui );
```

为了解析第二个调用，我们可以重载 min5()，允许两个不同的实参类型：

```
template <typename T, typename U>
    int min5( T, U );
```

下列函数调用则调用了这个新函数模板的实例：

```
// ok: int min5( int, unsigned int )
min5( i, ui );
```

不幸的是，原先的调用现在已变成二义的了：

```
// 错误：二义性：来自 min5( T, T ) 和 min5( T, U ) 的两个可能的实例
min5( 1024, i );
```

min5()的第二个声明允许两个不同类型的函数实参。但是，它没有要求它们一定是不同的。在这种情况下，T 和 U 都可以是 int 型。对于两个实参类型相同的调用，这两个模板声明都可以被实例化。要指明哪个函数模板比较好、并且消除调用的二义性的惟一方法是显式指定模板实参（关于显式模板实参的讨论见 10.4 节）。例如：

```
// ok: 从 min5( T, U ) 实例化
min5<int, int>( 1024, i );
```

但是，在这种情况下，我们其实可以取消重载函数模板。因为 min5(T,U)处理的调用集是由 min5(T,T)处理的超集，所以应该只提供 min5(T,U)的声明，而 min5(T,T)应该被删除。因此，正如我们在第 9 章开始时所说的，尽管重载是可能的，但是我们在设计重载函数时，仍然必须小心确保重载是必需的。这些设计限制也同样适用于定义重载函数模板。

在某些情况下，即使对于一个函数调用，两个不同的函数模板都可以实例化，但是该函数调用仍然可能不是二义的。已知 sum()的下列两个模板定义，下面就是一种情况：虽然从这两个函数模板的任一个都可以生成一个实例，但是第一个模板定义比较好。

```
template <typename Type>
     Type sum( Type*, int );

template <typename Type>
     Type sum( Type, int );

int ia[1024];

// Type == int ; sum<int>( int*, int ); or
// Type == int*; sum<int*>( int*, int ); ??
int ival1 = sum<int>( ia, 1024 );
```

真让人吃惊，上面的调用居然没有二义性，该模板是用第一个模板定义实例化的。为该实例选择的模板函数是**最特化的**（most specialized）。因此，Type 的模板实参是 int 而不是 int*。

一个模板要比另一个更特化，两个模板必须有相同的名字、相同的参数个数，对于不同类型的相应函数参数，如上面的 T*和 T，一个参数必须能接受另一个模板中相应参数能够接受的实参的超集。例如，对模板 sum(Type*,int)，第一个函数参数只能匹配指针类型的实参。对于模板 sum(Type,int)，第一个函数参数可以匹配指针类型以及任意其他类型的实参。第二个模板接受第一个模极所能够接受的类型的超集，接受更有限的实参集合的模板被称为是更特化的。在我们的例子中，模板 sum(Type*,int)是更特化的，它是为了例子中的函数调用而被实例化的模板。

10.8 考虑模板函数实例的重载解析 ※

如上节所述，函数模板可以被重载，函数模板可以与一个普通非模板函数同名。例如：

```
// 函数模板
template <class Type>
     Type sum( Type, int ) { /* ... */ }
// 普通（非模板）函数
double sum( double, double );
```

当程序调用 sum()时，该调用可以被解析为函数模板的实例，或者被解析为普通函数。到底调用哪个函数取决于这些函数中的哪一个与函数实参类型匹配得最好。在第 9 章中介绍的函数重载解析过程被用来决定哪个函数与函数调用中的实参最匹配。例如，考虑下列代码：

```
void calc( int ii, double dd ) {
     // 调用模板实例还是普通函数?
     sum( dd, ii );
}
```

sum(dd,ii)调用由模板实例化的函数。还是调用普通非模板函数？为了回答这个问题，让我们逐步分析函数重载解析过程。函数重载解析的第一步是构造可以被调用的候选函数集。该集合由与被调用函数同名的函数构成，在调用点上，这些函数都有一个声明是可见的。

当存在一个函数模板时，如果用该函数调用的实参可以实例化一个函数，则从该模板实例化的实例也是一个候选函数。一个函数是否能被实例化，取决于模板实参推演过程是否能进行。（模板实参推演的过程在 10.3 节中说明。）在前面的例子中，函数实参 dd 被用来推演 Type 的模板实参。被推演出来的模板实参是 double，而模板实例 sum(double,int)被加入到候选函数集中。因此，该调用有两个候选函数。模板实例 sum(double,int)和普通函数 sum(double,double)。

一旦模板实例被加入到候选函数集中，函数重载解析过程就会像以前一样进行。

函数重载解析的第二步是从候选函数集中选择可行函数集。可行函数是这样的候选函数：对它而言，存在着能够把每个函数实参转换成相应的函数参数类型的类型转换。（9.3 节给出了可以被应用在函数实参上的类型转换。）对实例 sum(double,int)和非模板函数 sum(double,double)都存在类型转换，所以，这两个函数都是可行函数。

函数重载解析的第三步是为应用在实参上的类型转换划分等级，以便选择最佳可行函数。针对我们的例子，等级划分如下：

- 对于函数模板实例 sum(double,int)：

1．因为第一个实参的实参和参数类型都是 double，该转换是精确匹配。

2．因为第二个实参的实参和参数的类型都是 int，该转换也是精确匹配。

- 对非模板函数 sum(double,double)：

1．因为第一个实参的实参和参数类型都是 double，该转换也是精确匹配。

2．因为第二个实参的实参类型是 int，参数类型是 double，应用的转换是浮点——有序标准转换。

当只考虑第一个实参时。两个函数一样好。但是，函数模板实例对于第二个参数更适合

一些。所以，对该调用选择得到的最佳可行函数是实例 sum(double,int)。

只有当模板实参推演成功时，函数模板实例才能进入候选函数集。所以即使模板实参推演失败，它也不是一个错误。在这种情况下，没有函数实例被加入到候选函数集中。例如，假设函数模板 sum()被声明如下：

```
// 函数模板
template <class T>
     int sum( T*, int ) { .... }
```

如果与前面相同的函数调用，模板实参推演过程将会失败，因为对于一个 double 型的函数实参来说不可能有 T*型的相应参数。因为对该调用来说并没有实例可以从该函数模板产生，所以也就不会有实例被加入到候选函数集中。那么在候选函数集中惟一的函数就是非模板函数 sum(double,double)，则它就是此调用选择出来的函数，第二个实参将转换成 double 型。

如果模板实参推演成功，但是被推演的模板实参被显式特化，又会怎么样呢？进入候选函数集合的是显式特化，它将代替从通用模板定义实例化的函数。例如：

```
// 函数模板定义
template <class Type> Type sum( Type, int ) { /* ... */ }

// Type == double 的显式特化
template<> double sum<double>( double,int );

// 普通（非模板）函数
double sum( double, double );
void manip( int ii, double dd ) {
     // 调用模板显式特化 sum<double> ()
     sum( dd, ii );
}
```

对 manip()中的 sum()的调用，模板实参推演过程发现从通用模板定义生成的实例 sum(double,int)应该进入候选函数集合。但是，sum(double,int)有一个显式特化，所以进入候选函数集的是显式特化。实际上，同为后来发现该特化是该调用的最好匹配，所以它是被函数重载解析过程选中的函数。

模板显式特化不会自动进入候选函数集，只有模板实参推演成功的模板特化才被考虑。例如：

```
// 函数模板定义
template <class Type>
     Type min( Type, Type ) { /* ... */ }

// Type == double 的显式特化
template<> double min<double>( double,double );

void manip( int ii, double dd ) {
     // 错误：模板实参推演失败
     // 该调用没有候选函数
     min( dd, ii );
}
```

这里函数模板 min()将针对实参 double 进行特化。但是，这个特化并没有进入候选函数

集、manip()中对 min()调用时，模板实参推演会失败，因为从每个函数实参为 Type 推演出来的模板实参是不同的。对于第一个实参，为 Type 推演出来的类型是 double。而对于第二个实参，为 Type 推演出来的类型是 int。因为模板实参推演失败，所以没有实例进入候选函数集，并且特化 min(double,double)也被忽略。又因为该调用没有其他的候选函数，所以调用是错误的。

正如 10.6 节所提到的，一个普通函数也可以具有与“一个模板的实例化函数”完全匹配的返回类型和参数表。在下列例子中，函数 min(int,int)只是一个普通函数，而不是函数模板 min()的特化。你可能还记得，特化声明必须以符号 template<>开始：

```
// 函数模板声明
template <class T>
     T min( T, T );

// 普通函数声明
int min( int, int ) { }
```

一个函数调用可以与普通函数以及函数模板的实例化函数都匹配。在下列例子中，调用 min(ai[0],99)的两个实参类型都是 int。该调用有两个可行函数：普通函数 min(int,int)和从函数模板实例化的、带有相同返回类型和参数的函数。

```
int ai[4] = { 22, 33, 44, 55 };
int main() {
     // ／调用普通函数 min( int, int )
     min( ai[0], 99 );
}
```

但是，这样的调用不是二义的。当非模板函数存在时，因为该函数被显式实现，所以它将被给予更高的优先级。函数重载解析过程为该调用选择普通函数 min(int,int)。

一旦某个调用被函数重载解析过程解析为一个普通函数，如果该程序不包含该函数的定义，也不能回头了。如果不存在函数体，编译器也不能将函数模板实例化，为此函数生成函数体。取而代之的是产生一个链接时刻错误。在下面的例子中，程序调用了一个普通函数 min(int,int)，但是没有定义它。该程序产生了一个链接错误：

```
// 函数模板
template <class T>
     T min( T, T ) { .... }

// 这个普通函数在该程序中没有被定义
int min( int ,int );
int ai[4] = { 22, 33, 44, 55 };
int main() {
     // 链接错误: min(int, int) 被调用，但是没有被定义
     min( ai[0], 99 );
}
```

定义一个普通函数，让它的返回类型和参数表与另一个从模板实例化的函数的相同，又有什么用处呢？记住，当调用从模板实例化的函数时，只有有限的类型转换可以被应用在模板实参推演过程使用的函数实参上、如果声明一个普通函数，则可以考虑用所有的类型转换来转换实参，这是因为普通函数参数的类型是固定的。让我们看一个例子，它给出了声明一

个普通函数的原因。

假设我们想定义一个函数模板特化 min<int>(int，int)并希望：当用任何整型类型的实参调用 min()时，无论实参类型是否相同都调用这个函数。因为类型转换上的限制，所以用不同类型的整型实参的调用不会直接调用函数模板实例 min<int>(int,int)。我们可以通过指定显式模板实参直接调用该实例。但是，我们更喜欢一个不要求修改每个调用点的方案。通过定义一个普通函数，无论何时使用整型实参，我们的程序都会调用 min(int,int)的特化版本，而无需我们为每个调用都使用显式模板实参。例如：

```
// 函数模板定义
template <class Type>
     Type min( Type t1, Type t2 ) { ... }

int ai[4] = { 22, 33, 44, 55 };
short ss = 88;
void call_instantiation() {
     // 错误：这个调用没有候选函数
     min( ai[0], ss );
}
// 普通函数
int min( int a1, int a2 ) {
     min<int>( a1, a2 );
}
int main() {
     call_instantiation();
     // 调用普通函数
     min( ai[0], ss );
}
```

在 call_instantiation()中的调用 min(ai[0],ss)没有候选函数。因为，想要从函数模板 min()生成一个候选函数肯定要失败从函数实参将为 Type 推演出不同的模板实参，所以该调用是错误的。但是对于 main()中的调用 min(ai[0],ss)来说，普通函数 min(int,int)的声明是可见的。这个普通函数是该调用的可行函数：第一个实参的类型与相应参数的类型精确匹配第二个实参可以用提升转换为相应参数的类型。由于该普通函数是第二个调用的可行函数集中的惟一函数，所以它将被选中。

我们已经了解当涉及到同名的函数模板实例、函数模板特化以及普通函数时，函数重载解析是怎样进行的，现在让我们来总结一下，对于一个调用，考虑普通函数和函数模板的函数重载解析步骤：

1．生成候选函数集。

考虑与函数调用同名的函数模板。如果对于该函数调用的实参，模板实参推演能够成功则实例化一个函数模板，或者对于推演出来的模板实参存在一个模板特化，则该模板特化就是一个候选函数。

2．生成可行函数集，如 9.3 节所描述。

只保留候选函数集中可以用函数调用实参调用的函数。

3．对类型转换划分等级，如 9.3 节中描述。

a．如果只选择了一个函数，则调用该函数。

b．如果该调用是二义的，则从可行函数集中去掉函数模板实例。

4．只考虑可行函数集中的普通函数，完成重载解析过程，如 9.3 节中描述。

a．如果只选择了一个函数，则调用该函数。

b．否则，该调用是二义的。

让我们一步步地来看一个例子。下面是两个声明：一个函数模板声明，一个带有两个 double 型实参的普通函数。

```
template <class Type>
    Type max( Type, Type ) { .... }

// 普通函数
double max( double, double );
```

下面是三个 max()调用。你能分辨出每个调用分别会调用哪个实例吗？

```
int main() {
    int ival;
    double dval;
    float fd;

    // 向 ival, dval, 和 fd 赋某些值
    max( 0, ival );
    max( 0.25, dval );
    max( 0, fd );
}
```

让我们按顺序查看每个调用。

- max(o,ival)：两个实参的类型都是 int。对该调用存在两个候选函数：函数模板实例 max(int,int)以及普通函数 max(double,double)。用于函数模板实例对函数实参来说是精确匹配的，所以它将是被调用的函数。
- max(0.25,dval)：这两个实参都是 double 型。对该调用存在两个候选函数：函数模板实例 max(double,double)以及普通函数 max(double,double)。因为调用与两个函数都完全匹配，所以该调用存在二义。根据规则 3b 可知在这种情况下应选择普通函数。
- max(0,fd)：实参的类型分别是 int 和 float。对该调用只存在一个候选函数：普通函数 max(double,double)。因为从两个函数实参为 Type 推演出的模板实参不是一种类型，所以模板实参推演会失败，故也没有模板实例进入候选函数集。因为存在类型转换能把函数实参转换成相应函数参数的类型，该普通函数是个可行函数。因此，选择普通函数。如果该普通函数没有被定义，则该调用将是个错误。

如果我们已经为 max()定义了第二个普通函数，又会怎么样呢？例如：

```
template <class T> T max( T, T ) { .... }

// 两个普通函数
char max( char, char );
double max( double, double );
```

第三个调用的解析会不同吗？是的。

```
int main() {
    float fd;
    // 解析为哪个函数?
    max( 0, fd );
}
```

规则 3b 说明，因为该调用是二义的，所以只考虑普通函数。这些函数都没有被选为最佳可行函数，因为在实参上的类型转换对两个函数都是一样的不好：两个实参都要求标准转换以便匹配可行函数中的两个相应的参数。所以该调用是二义的，将被编译器标记为错误。

练习 10.11

让我们回到前面给出的例子

```
template <class Type>
    Type max( Type, Type ) { .... }

double max( double, double );

int main() {
    int ival;
    double dval;
    float fd;
    max( 0, ival );
    max( 0.25, dval );
    max( 0, fd );
}
```

下面的函数模板特化被加入到全局域的声明集中：

```
template <> char max<char>( char, char ) { .... }
```

重新考虑 main()中的函数调用，并为每个调用列出候选函数和可行函数。假定下面的函数调用被加到 main()中，则该调用将被解析为哪个函数？为什么？

```
int main() {
    // ...
    max (0, 'J' );
}
```

练习 10.12

假设有下列模板定义和特化，以及变量、函数声明的集合：

```
int i; unsigned int ui;
char str[24]; int ia[24];
template <class T> T calc( T*, int );
template <class T> T calc( T, T );
template<> char calc( char*, int );
double calc( double, double );
```

指出下列每个调用将会调用哪个模板实例或函数。为每个调用，列出候选函数、可行函数，并解释选择最佳可行函数的原因。

```
(a) calc( str, 24 );   (d) calc( i, ui );
(b) calc( ia, 24 );    (e) calc( ia, ui );
(c) calc( ia[0], i );  (f) calc( &i, i );
```

10.9 模板定义中的名字解析 ※

在模板定义中有些结构在两个模板实例之间有不同的意义，而另外一些结构在模板的所有实例之间意义相同。这取决于该结构是会涉及模板参数。例如：

```
template<typename Type>
Type min( Type* array, int size )
{
    Type min_val = array[0];

    for ( int i = 1; i < size; ++i )
        if ( array[i] < min_val )
            min_val = array[i];
    print( "Minimum value found: " );
    print( min_val );
    return min_val;
}
```

在 min()中，array 和 min_val 的类型取决于模板被实例化时取代 Type 的实际类型，而 size 的类型总是 int，与模板参数的类型无关。array 和 min_val 的类型随不同的模板实例而不同。因此，我们说这些变量的类型依赖于模板参数（depend on a template parameter），而 size 的类型则不依赖于模板参数。

因为不知道 min_val 的类型，所以当 min_val 出现在表达式中时，究竟哪个操作将会被用到也是未知的。例如，函数调用 print(min_val)将会调用哪个 print()函数？应该是 int 类型的 print()函数或是 float 类型的 print()函数？会不会因为没有 print()函数可以用 min_val 的类型的实参来调用而使得这个调用是错误的？只有当实例化该模板、知道了 min_val 的实际类型时，我们才能回答这些问题。因此我们也称调用 print(min_val)依赖于模板参数。

在 min()中，不依赖于模板参数的结构就没有这样的问题。例如，调用 print("Minimum value found")总知道应该调用哪个函数。这个函数被用来输出字符串，被调用的 print()函数不会随着模板实例的不同而不同。因此，我们称该调用不依赖于模板参数。

正如我们在第 7 章所了解的，在 C++中，函数必须在调用前被声明。在模板定义中，被调用的函数必须在模板定义出现之前被声明吗？在前面的例子中，在 min()的模板定义中，函数 print()必须在调用之前先声明吗？答案取决于我们引用了哪种名字。不依赖于模板参数的结构必须在使用之前先声明。因此前面给出的函数模板 min()的定义是不正确的。因为调用

```
print( "Minimum value found: " );
```

不依赖于模板参数，所以针对字符串的函数 print()必须在被模板定义使用它之前先被声明。为了解决这个问题，print()函数的声明必须在 min()的定义之前给出，如下所示：

```
// ---- primer.h ----
// 这个声明是必需的
// print( const char * ) 在 min() 中被调用
void print( const char* );

template<typename Type>
Type min( Type* array, int size ) {
    // ....
    print( "Minimum value found: " );
    print( min_val );
    return min_val;
}
```

另一方面，调用 print(min_val)使用的 print()函数的声明则是不需要的，因为我们还不知道要寻找的是哪个 print()。只有当 min_val 的类型是已知的时候，才可能知道应该调用哪个 print()函数。

那么应该在什么时候声明 print(min_val)调用的 print()函数呢？必须在实例化模板之前声明。例如：

```
#include <primer.h>

void print( int );
int ai[4] = { 12, 8, 73, 45 };

int main() {
    int size = sizeof(ai) / sizeof(int);

    // min( int*, int ) 的实例
    min( &ai[0], size );
}
```

函数 main()调用函数模板实例 min(int*,int)。在 min()的这个实例中，Type 被 int 取代，而变量 min_val 的类型是 int。所以 print(min_val)调用一个可以用 int 类型的实参来调用的函数。在实例化 min(int*,int)的时候，我们知道 print()函数的第二个调用有一个 int 型的实参。也正是在这个时候。要求可以用 int 型实参调用的 print()函数必须可见。在我们的例子中，被选择的函数是 print(int)。如果函数 print(int)在 min(int*,int)被实例化之前没有被声明，则该实例化会导致编译时刻错误。

所以，模板定义中的名字解析分两个步骤进行。首先：不依赖于模板参数的名字在模板定义时被解析；其次，依赖于模板参数的名字在模板被实例化时被解析。你可能会问，为什么要分两步呢？为什么不是所有的名字都在模板被实例化时被解析呢？

如果我们是函数模板的设计者，我们可以控制模板定义中的名字解析方式。假设函数模板 min()位于一个函数库中，该库还定义了其他一些模板和函数。我们希望 min()的实例在所有可能的时候都使用库中的其他组件。在前面的例子中，库的接口定义位于头文件<primer.h>中。函数 print(const char*)的声明和函数模板 min()的定义都是库的接口的一部分。我们希望 min()的实例调用该库提供的 print()函数。名字解析的第一步保证了这件事情。当用在模板定义中的名字不依赖于模板参数时，此名字肯定会指向库中定义的另一个组件——即，它会

指向和函数模板定义在一起、被打包在头文件<primer.h>中的声明。

事实上，函数模板的设计者必须确保为模板定义中用到的、所有不依赖于模板参数的名字提供声明。如果用在模板定义中的名字不依赖于模板参数，并且在定义该模板时没有找到该名字的声明，则模板定义是错误的。当模板被实例化时，编译器就不再考虑这样的错误。例如：

```
// ---- primer.h ----
template<typename Type>
Type min( Type* array, int size )
{
     Type min_val = array[0];
     // ...
     // 错误：没有找到 print ( const char* )
     print( "Minimum value found: " );
     // ok：依赖于模板参数
     print( min_val );
     // ...
}

// ---- user.C ----
#include <primer.h>

// print( const char* ) 的这个声明被忽略
void print( const char* );

void print( int );
int ai[4] = { 12, 8, 73, 45 };
int main() {
     int size = sizeof(ai) / sizeof(int);
     // min( int*, int ) 的实例
     min( &ai[0], size );
}
```

在 user.C 中的 print(const char*)的声明，在模板定义出现的地方是不可见的。但是，这样的声明在模板 min(int*,int)被实例化时是可见的，但该声明并不被调用 print("minimum value found:")所考虑，因为该调用不依赖于模板参数。除非模板定义中的结构依赖于模板参数，否则一个名字在模板定义的上下文中被解析，此解析过程不会在该模板实例化的上下文中被重新考虑。因此，确保被用在模板定义中的名字的声明和模板定义被正确地包含为库接口的一部分，这是模板设计者的责任。

让我们换个思路，假设该库是由其他人编写的，我们是头文件<primer.h>中的定义的用户。有这样一种情况，我们希望当程序在实例化库中的模板时考虑程序中定义的对象和函数。例如，假设我们的程序定义了一个称为 SmallInt 的类。我们希望实例化库<primer.h>中的函数 min()，以获得 SmallInt 型的对象的数组中的最小值。

当函数模板 min()被 SmallInt 型对象的数组实例化时，Type 的模板实参是 class 类型的 SmallInt。这意味着，在 min()的实例中，min_val 的类型是 SmallInt。在 min()的实例中，调用 print(min_val)应该被解析为哪个函数？

```
// ---- user.h ----
class SmallInt { /* ... */ };
void print( const SmallInt & );

// ---- user.C ----
#include <primer.h>
#include "user.h"
SmallInt asi[4];
int main() {
     // 设置 asi 的元素
     // min( SmallInt*, int ) 的实例
     int size = sizeof(asi) / sizeof(SmallInt);
     min( &asi[0], size );
}
```

对，我们希望考虑我们的函数 print(const SmallInt&)。只考虑在库<primer.h>中定义的函数是远远不够的。名字解析的第二步保证这样的事情可以发生。当用在模板定义中的名字依赖于模板参数时，在实例上下文中声明的名字被考虑。所以，对于函数模板中的该操作，如果模板实参是 SmallInt 型，那么可以处理 SmallInt 型的对象的函数肯定会被考虑。

在源代码中模板被实例化的位置被称为模板的**实例化点**（point of instantiation）。知道模板的实例化点是很重要的，因为它决定了对依赖于模板参数的名字所考虑的声明。函数模板的实例化点总是在名字空间域中，并且跟在引用该实例的函数后。例如，min(SmallInt*,int)的实例化点就紧跟在名字空间域中的函数 main()后：

```
// ...
int main() {
     // ...
     // 使用 min( SmallInt*, int )
     min( &asi[0], size );
}
// min( SmallInt*, int ) 的实例化点
// 好像实例定义如下出现
SmallInt min( SmallInt* array, int size )
     { /* ... */ }
```

但是，如果在一个源文件中，一个模板实例不只被使用一次，又该怎么办呢？实例化点在哪儿呢？可能你会问：为什么这很重要？在我们的例子中，它的重要性是因为函数 print(const SmallInt&)的声明必须出现在 min(SmallInt*,int)的实例化点之前。例如：

```
#include <primer.h>

void another();
SmallInt asi[4];
int main() {
     // 设置 asi 的元素
     int size = sizeof(asi) / sizeof(SmallInt);
     min( &asi[0], size );
     another();

     // ...
```

```
}
// 实例化点在这儿?

void another() {
     int size = sizeof(asi) / sizeof(SmallInt);
     min( &asi[0], size );
}
// 还是这儿?
```

当模板实例要多次使用时，在每个使用该实例的函数定义之后都有一个实例化点。编译器自由选择这些实例化点之一来真正实例化该函数模板。这意味着在组织代码时，我们必须小心地把解析依赖于模板参数的名字所需要的声明放置在模板的第一个实例化点之前。因此，在模板的任何一个实例被使用之前，应该头文件中给出所有必需的声明，并包含这个头文件。例如：

```
#include <primer.h>

// user.h 含有实例所需的声明
#include "user.h"

void another();
SmallInt asi[4];
int main() {
     // ...
}
// min( SmallInt*, int ) 的第一个实例化点
void another() {
     // ...
}
// min( SmallInt*, int ) 的第二个实例化点
```

如果不只在一个文件中使用模板实例，该怎样办呢？例如，如果函数 another()函数 main()在不同的文件中，该怎样办呢？那么在每个使用该模板的实例的文件中都存在一个实例化点。编译器自由选择这些文件中的任一个实例化点来实例化该函数模板。所以在组织代码时，我们必须也要小心地把头文件“user.h”包含在每个使用该函数模板实例的文件中。这保证 min(SmallInt*,int)的实例指向我们的函数 print(const SmallInt&)，这是我们所期望的，与编译器选择哪个实例化点无关。

练习 10.13

列出模板定义中名字解析的两个步骤。解释第一步如何解决库设计者关心的事情，以及第二步怎样提供模板用户所需要的灵活性。

练习 10.14

在 max(LongDouble*,SIZE)实例中的名字 display 和 SIZE 引用哪个声明？

```
// ---- exercise.h ----
void display( const void* );
typedef unsigned int SIZE;
template<typename Type>
```

```
        Type max( Type* array, SIZE size )
{
        Type max_val = array[0];
        for ( SIZE i = 1; i < size; ++i )
        if ( array[i] > max_val )
        max_val = array[i];
        display( "Maximum value found: " );
        display( max_val );
        return max_val;
}

// ---- user.h ----
class LongDouble { /* ... */ };
void display( const LongDouble & );
void display( const char * );
typedef int SIZE;

// ---- user.C ----
#include <exercise.h>
#include "user.h"
LongDouble ad[7];
int main() {
        // 设置 ad 的元素
        // max( LongDouble*, SIZE ) 的实例化
        SIZE size = sizeof(ad) / sizeof(LongDouble);
        max( &ad[0], size );
}
```

10.10 名字空间和函数模板 ※

与其他全局域定义一样，函数模板定义也可以被放在名字空间中。（关于名字空间的讨论见 8.5 节和 8.6 节。）这种模板定义的意义与在全局域中定义的一样，除了该模板的名字被隐藏在名字空间中。当在名字空间之外使用该模板时，该模板名必须被限定，或提供一个 usign 声明。

```
// ---- primer.h ----
namespace cplusplus_primer {
        // 模板定义被隐藏在名字空间中
        template<class Type>
            Type min( Type* array, int size ) { /* ... */ }
}

// ---- user.C ----
#include <primer.h>

int ai[4] = { 12, 8, 73, 45 };
int main() {
        int size = sizeof(ai) / sizeof(ai[0]);
```

```
        // 错误：没有找到函数 min()
        min( &ai[0], size );
        using cplusplus_primer::min;  // using 声明

        // ok: 指向名字空间 cplusplus_primer 中的 min()
        min( &ai[0], size );
}
```

如果我们的程序使用了一个在名字空间中定义的模板，而且我们希望为其提供一个特化，又会怎么样？（关于模板显式特化在 10.6 节介绍）。例如，我们想用名字空间 cplusplus_primer 中定义的模板 min()来找到 SmallInt 型的对象数组中的最小值。但是，我们意识到，在名字空间 cplusplus_primer 中提供的模板定义不能完全奏效。函数模板中的比较操作如下：

```
if ( array[i] < min_val )
```

该语句用小于（<）操作符比较两个 SmallInt 型的类对象。除非为类 SmallInt 定义了一个重载的 operator<()，否则该操作符不能被应用在两个类对象上。（我们将在 15 章看到怎样定义重载操作符。）假设我们为 min()函数模板定义一个特化，以便用一个名为 compareLess()的函数找到 SmallInt 类对象的数组中的最小值。下面是 compareLess()函数的声明：

```
// SmallInt 对象的比较函数
// 如果 param1 小于 parm2，返回 true
bool compareLess( const SmallInt &parm1, const SmallInt &parm2 );
```

这个函数的定义看起来会是什么样的呢？为了回答这个问题，我们需要更详细地看看我们的 SmallInt 的定义。这个 SmallInt 可以用来一些对象，能够存放定义与 8 位 unsigned char 范围相同的值，即 0－255。它的附加功能是能够捕捉到上溢和下溢错误。除此之外，我们还希望它的工作方式与 unsigned char 相同。类 SmallInt 的定义看起来如下：

```
class SmallInt {
public:
    SmallInt( int ival ) : value( ival ) { }
    friend bool compareLess( const SmallInt &, const SmallInt & );

private:
    int value;    // 数据成员
};
```

在这个类定义中，有些事情我们应该讨论一下、首先，该类有一个私有数据成员：value。它是存储 SmallInt 型对象值的数据成员。该类还包含一个构造函数：

```
// 类 SmallInt 的构造函数
SmallInt( int ival ) : value( ival ) { }
```

该构造函数有一个参数：ival。它执行的惟一动作是用参数 ival 的值初始化类数据成员 value。

现在，我们可以回答前面的问题了。函数 compareLess()怎样定义？该函数将比较它的两个 SmallInt 型参数的 value 数据成员，如下：

```
// 如果 parm1 小于 parm2，则返回 true
bool compareLess( const SmallInt &parm1, const SmallInt &parm2 ) {
    return parm1.value < parm2.value;
}
```

但是要注意，数据成员 value 是 SmallInt 类的私有数据成员。这个全局函数怎样引用该私有成员而不破坏类 SmallInt 的封装化且不会引起编译错误呢？如果看看 SmallInt 的定义，你会注意到这个类定义把全局函数 compareLess()声明为 friend。当一个函数是一个类的 friend 时，它就可以引用该类的私有成员，譬如我们的函数 compareLess()在 15.2 节中对类的 fiend 有进一步介绍）。

现在我们已经准备好为 min()定义我们的模板特化了。它如下使用 compareLess()函数：

```
// 针对 SmallInt 对象数组的 min() 特化
template<> SmallInt min<SmallInt>( SmallInt* array, int size )
{
    SmallInt min_val = array[0];
    for ( int i = 1; i < size; ++i )
    // 使用函数 compareLess() 比较
    if ( compareLess( array[i], min_val ) )
        min_val = array[i];
    print( "Minimum value found: " );
    print( min_val );
    return min_val;
}
```

我们应该在哪里声明这个特化？看看下面的做法怎么样：

```
// ---- primer.h ----
namespace cplusplus_primer {
    // 模板定义被隐藏在名字空间中
    template<class Type>
        Type min( Type* array, int size ) { /* ... */ }
}

// ---- user.h ----
class SmallInt { /* ... */ };
void print( const SmallInt & );
bool compareLess( const SmallInt &, const SmallInt & );

// ---- user.C ----
#include <primer.h>
#include "user.h"
// 错误：不是 cplusplus_primer::min() 的特化
template<> SmallInt min<SmallInt>( SmallInt* array, int size )
    { /* ... */ }
// ...
```

不幸的是，该代码不能完成任务。函数模板的显式特化声明必须被声明在该通用模板被定义的名字空间内。因此，我们必须在名字空间 cplusplus_primer 中定义 min()的特化。在我们的程序中，有两种实现它的方式。

请回忆一下，由于名字空间的定义可以是非连续的，所以我们可以重新打开名字空间 cplusplus_primer 的定义并加上该特化的定义，如下：

```
// ---- user.C ----
#include <primer.h>
#include "user.h"
```

```
namespace cplusplus_primer {
// cplusplus_primer::min() 的特化
template<> SmallInt min<SmallInt>( SmallInt* array, int size )
     { /* ... */ }
}
SmallInt asi[4];

int main() {
     // 用 set() 成员函数设置 asi 的元素
     using cplusplus_primer::min; // using declaration
     int size = sizeof(asi) / sizeof(SmallInt);
     // min(SmallInt*,int) 的实例化
     min( &asi[0], size );
}
```

或者我们可以用“在名字空间定义之外定义任何名字空间成员”的方式定义该特化：通过用外围名字空间名来限定修饰名字空间成员名。

```
// ---- user.C ----
#include <primer.h>
#include "user.h"

// cplusplus_primer::min() 的特化
// 此特化的名字被限定修饰
template<> SmallInt cplusplus_primer::
     min<SmallInt>( SmallInt* array, int size )
          { /* ... */ }

// ...
```

所以，作为包含模板定义的库的用户，如果我们想为库中的模板提供特化，那么我们就必须确保它们的定义被合适地放置在含有原始模板定义的名字空间内。

练习 10.15

现在我们把练习 10.14 中给出的头文件<exercise.h>的内容放到名字空间 cplusplus_primer 中。怎样改变函数 main()使其能够实例化位于名字空间 cplusplus_primer 中的函数模板 max()?

练习 10.16

再次参考练习 10.14，已知头文件<exercise.h>的内容被放在名字空间 cplusplus_primer 中，我们想为类 LongDouble 对象的数组定义函数模板 max()的特化。并且让该模板特化使用如下定义的函数 compareGreater()，来比较两个 LongDouble 型的对象：

```
// 比较两个 LongDouble 对象的函数
// 如果 parm1 大于 parm2，则返回 true
bool compareGreater( const LongDouble &parm1,
          const LongDouble &parm2 );
```

类 LongDouble 的定义如下：

```
class LongDouble {
```

```
public:
    LongDouble(double dval) : value(dval) { }
    void set(double dval) { value = dval; }

    friend bool compareGreater( const LongDouble &,
        const LongDouble & );
private:
    double value;
};
```

给出函数 compareGreater()以及使用该函数的 max()特化的定义。写一个用于设置数组 ad 元素的 main()函数，然后调用 max()的特化取得 ad 中的最大值：初始化数组 ad 的元素的值应该通过读取标准输入 cin 获得。

10.11 函数模板示例

本节将给出一个程序示例，用来说明怎样定义和使用函数模板。这个示例定义了一个函数模板 sort()，它对数组的元素进行排序。数组本身由在 2.5 节介绍的 Array 类模板表示。因此，函数模板 sort()可以被用来排序任意类型元素的数组。

我们在第 6 章中看到，C++标准库定义了一个容器类型被称为 Vector，它的行为和 2.5 节定义的 Array 非常相像。第 12 章将介绍泛型算法，它可以处理第 6 章描述的容器类型。其中一个算法被称为 sort()，它可以被用来排序 vector 的内容。在本节中，我们将定义自己的“泛型 sort()算法”，以处理我们的 Array 类。你在本节中所看到的是 C++标准库中的算法的简化版本。

我们针对 Array 类模板的 sort()函数模板定义如下：

```
#include "Array.h"

template <class elemType>
void sort( Array<elemType> &array, int low, int high ) {
    if ( low < high ) {
        int lo = low;
        int hi = high + 1;
        elemType elem = array[lo];

        for (;;) {
            while ( min( array[++lo], elem ) != elem && lo < high ) ;
            while ( min( array[--hi], elem ) == elem && hi > low ) ;
            if (lo < hi)
                swap( array, lo, hi );
            else break;
        }
        swap( array, low, hi );
        sort( array, low, hi-1 );
        sort( array, hi+1, high );
    }
}
```

函数 sort()使用了两个辅助函数：min()和 swap()。这两个函数都被定义为函数模板，而且能够处理可用来实例化 sort()的全部实参类型。min()被定义为函数模板，以便我们可以找到任意类型的两个数组元素的最小值：

```
template <class Type>
    Type min( Type a, Type b ) {
        return a < b ? a : b;
}
```

swap()被定义为函数模板，以便我们可以交换任意类型的两个数组元素：

```
#include "Array.h"
template <class elemType>
    void swap( Array<elemType> &array, int i, int j )
{
    elemType tmp = array[ i ];
    array[ i ] = array[ j ];
    array[ j ] = tmp;
}
```

为了确保我们的 sort()函数模板能够真正地工作，我们需要在数组被排序后显示数组的内容。因为 display()函数必须能够处理 Array 类模板被实例化后的任何数组，所以我们也必须把 display()定义为函数模板：

```
#include <iostream>

template <class elemType>
    void display( Array<elemType> &array )
{ // display format: < 0 1 2 3 4 5 >
    cout << "< ";

    for ( int ix = 0; ix < array.size(); ++ix )
        cout << array[ix] << " ";

    cout << ">\n";
}
```

在这个例子中，我们使用包含编译模式，并把函数模板放在头文件 Array.h 中，跟在 Array 类模板后面。

下一步是写一个函数练习这些函数模板。依次被传递给函数 sort()的数组为 double 型的数组、int 型的数组和 string 型的数组。下面是程序：

```
#include <iostream>
#include <string>

#include "Array.h"

double da[10] = {
    26.7, 5.7, 37.7, 1.7, 61.7, 11.7, 59.7,
    15.7, 48.7, 19.7 };

int ia[16] = {
    503, 87, 512, 61, 908, 170, 897, 275, 653,
```

```
        426, 154, 509, 612, 677, 765, 703 };

    string sa[11] = {
        "a", "heavy", "snow", "was", "falling", "when",
        "they", "left", "the", "police", "station" };

    int main() {
        // 调用构造函数初始化 arrd
        Array<double> arrd( da, sizeof(da)/sizeof(da[0]) );

        // 调用构造函数初始化 arri
        Array<int> arri( ia, sizeof(ia)/sizeof(ia[0]) );

        // 调用构造函数初始化 arrs
        Array<string> arrs( sa, sizeof(sa)/sizeof(sa[0]) );

        cout << "sort array of doubles (size == "
             << arrd.size() << ")" << endl;
        sort( arrd, 0, arrd.size()-1 );
        display(arrd);

        cout << "\sort array of ints (size == "
             << arri.size() << ")" << endl;
        sort( arri, 0, arri.size()-1 );
        display(arri);

        cout << "\sort array of strings (size == "
             << arrs.size() << ")" << endl;
        sort( arrs, 0, arrs.size()-1 );
        display(arrs);

        return 0;
    }
```

编译并运行该程序，产生下列输出（数组的输出已经被手工调整以适应篇幅）：

```
sort array of doubles (size == 10)
< 1.7 5.7 11.7 14.9 15.7 19.7 26.7
37.7 48.7 59.7 61.7 >

sort array of ints (size == 16)
< 61 87 154 170 275 426 503 509 512
612 653 677 703 765 897 908 >

sort array of strings (size == 11)
< "a" "falling" "heavy" "left" "police" "snow"
"station" "the" "they" "was" "when" >
```

在 C++标准库定义的泛型算法中（在第 12 章中），你还会找到一个 min()函数和一个 swap()函数。在第 12 章中我们将了解怎样在程序中使用它们。

11

异 常 处 理

异常处理是一种允许两个独立开发的程序组件在程序执行期间遇到程序不正常的情况（称为异常，exception）时，相互通信的机制。在本章中，我们将首先了解怎样在程序异常出现的位置产生（raise）或抛出（throw）异常。然后，我们再看一看怎样用 try 块把处理代码（或称作 catch 子句）和一组程序语句关联起来，并了解 catch 子句怎样处理异常。然后我们将会介绍异常规范，它是一种把一组异常和一个函数声明关联起来的机制，用于保证该函数不会抛出其他类型的异常。在本章最后，我们还将讨论程序使用异常时的一些注意事项。

11.1 抛出异常

异常（Exception）是程序可能检测到的，运行时刻不正常的情况，如被 0 除、数组越界访问或空闲存储内存耗尽等等。这样的异常存在于程序的正常函数之外。而且要求程序立即处理。C++提供了一些内置的语言特性来产生（raise）并处理异常。这些语言特性将激活了一种运行时刻机制，通过这种机制可在 C++程序的两个无关（常常是独立开发）的部分进行异常通信。

C++程序中出现异常时，检测到异常的程序段可以通过产生（raise ）或抛出（throw）异常来通知“异常已经发生”。为了了解在 C++中是怎样抛出异常的，我们来重新实现 4.15 节给出的 iStack 类，这次我们用异常来指出在栈的处理中不正常的情况。类 iStack 的定义看起来是这样的：

```
#include <vector>

class iStack {
public:
     iStack( int capacity )
          : _stack( capacity ), _top( 0 ) { }
     bool pop( int &top_value );
     bool push( int value );
     bool full();
     bool empty();
```

```
        void display();
        int size();

    private:
        int _top;
        vector< int > _stack;
    };
```

这里的栈用一个 int 型的 vector 来实现。当我们创建一个 iStack 对象时，iStack 的构造函数就会创建一个 int 型的 vector，其长度由初始值指定。该长度是 iStack 对象可以含有的元素的最大个数。例如，下面的代码创建了一个名为 myStack 的 iStack 对象，它可以含有 20 个 int 型的值：

```
iStack myStack(20);
```

处理 myStack 时会出现什么样的错误呢？下面是 iStack 类可能会遇到的不正常情况：

1．要求一个 pop()操作，但栈却是空的。
2．要求一个 push()操作，但栈却是满的。

如果这些不正常的情况都应该用异常通知给操纵 iStack 对象的函数，那么我们从哪儿开始呢？

首先，必须定义可以被抛出的异常。在 C++中，异常往往用类（class）来实现。虽然我们要到第 13 章才能完整地介绍类。但是在这里我们还是跟随 iStack 类定义了两个被用作异常的类，并把这些类定义放在头文件 stackExcp.h 中：

```
// stackExcp.h
class popOnEmpty { /* ... */ };
class popOnFull { /* ... */ };
```

在第 19 章我们将更详细地讨论 class 类型的异常，并讨论由 C++标准库提供的异常类继承层次。

我们必须要修改 pop()和 push()成员函数的定义，以便让它们可以抛出新定义的异常。抛出异常可通过 throw 表达式来实现，throw 表达式看起来非常像 return 语句。throw 表达式由关键字 throw 后面跟一个表达式构成，该表达式的类型是被抛出异常的类型。在 pop()中的 throw 表达式是什么样子的呢？我们来试一下：

```
// 喔！不是十分正确
throw  popOnEmpty;
```

不幸的是，这不完全正确。异常是个对象，pop()必须抛出一个 class 类型的对象。在 throw 表达式中的表达式不能只是一个类型。为创建一个 class 类型的对象，我们需要调用该类的构造函数。调用一个构造函数的 throw 表达式又是什么样的呢？下面是 pop()中的 throw 表达式：

```
// 表达式是一个构造函数调用
throw popOnEmpty();
```

该 throw 表达式创建一个 popOnEmpty 类型的异常对象。

成员函数 pop()和 push()被定义为返回一个 bool 型的值：返回 true 表示该操作成功，返回 false 表示失败。因为现在用异常表示 pop()和 push()操作的失败，那么这些函数的返回值

就不是必要的了。因而我们把这些成员函数的返回类型定义为 void，例如：

```
class iStack {
public:
     // ...

     // 不再返回一个值
     void pop( int &value );
     void push( int value );

private:
     // ...
};
```

使用 iStack 类的函数现在会假设：除非抛出异常，否则每件事情都是正常的，它们不再需要测试成员函数 pop()和 push()的返回值来了解操作是否成功。我们将在下两节了解怎样定义处理异常的函数。

现在我们准备给出 iStack 的成员函数 pop()和 push()的新实现代码：

```
#include "stackExcp.h"

void iStack::pop( int &top_value )
{
     if ( empty() )
          throw popOnEmpty();

     top_value = _stack[ --_top ];

     cout << "iStack::pop(): " << top_value << endl;
}

void iStack::push( int value )
{
     cout << "iStack::push( "iStack::push( " << value << " ) \n";

     if ( full() )
          throw pushOnFull();

     stack[ _top++ ] = value;
}
```

虽然异常往往是 class 类型的对象。但是 throw 表达式也可以抛出任何类型的对象。例如，（虽然很不常见）在下面的代码例子中，函数 mathFunc()抛出一个枚举类型的异常对象。下面是合法的 C++代码：

```
enum EHstate { noErr, zeroOp, negativeOp, severeError };

int mathFunc( int i ) {
     if ( i == 0 )
          throw zeroOp;    // 枚举类型的异常

          // 否则的话，继续正常处理流程
```

练习 11.1

下面的 throw 表达式哪些是错误的？为什么？对于合法的 throw 表达式，指出被抛出的异常的类型。

```
(a)  class exceptionType { };
     throw exceptionType { };
(b)  int excpObj;
     throw excpObj;
(c)  enum mathErr { overflow, underflow, zeroDivide };
     throw zeroDivide();
(d)  int *pi = &excpObj;
     throw pi;
```

练习 11.2

2.3 节定义的类 IntArray 有一个成员操作符函数 operator[]()，它用 assert()指示索引超越数组的边界。改变 operator[]()的定义，使其在该情况下抛出异常。定义一个异常类用作被抛出异常的类型。

11.2 try 块

下面的小程序用到了我们的 iStack 类，以及上节定义的 pop()和 push()成员函数。在 main()中的 for 循环迭代 50 次。它把 3 的倍数值：3、6、9 等等，压入到栈中。当遇到 4 的倍数时，如 4、8、12 等等，显示栈的内容。当值是 10 的倍数时，如 10、20、30 等等，则把栈的最后一项弹出，然后再次显示栈的内容。怎样改变 main()使其能够处理由 iStack 成员函数抛出的异常呢？

```
#include <iostream>
#include "iStack.h"
int main() {
     iStack stack( 32 );
     stack.display();

     for ( int ix = 1; ix < 51; ++ix )
     {
          if ( ix % 3 == 0 )
               stack.push( ix );
          if ( ix % 4 == 0 )
               stack.display();
          if ( ix % 10 == 0) {
               int dummy;
               stack.pop( dummy );
               stack.display();
          }
     }

     return 0;
```

```
}
```

try 块（try block）必须包围能够抛出异常的语句。try 块以关键字 try 开始，后面是花括号括起来的语句序列。在 try 块之后是一组处理代码，被称为 catch 子句。try 块把语句分成组，并将其与相应地处理这些语句可能抛出的异常的处理语句相关联。我们应该把 try 块放在 main()中的什么地方来处理 popOnEmpty 和 pushOnFull 异常？我们尝试一下：

```
for ( int ix = 1; ix < 51; ++ix ) {
    try { // pushOnFull 异常的 try 块
        if ( ix % 3 == 0 )
            stack.push( ix );
    }
    catch ( pushOnFull ) { ... }
    if ( ix % 4 == 0 )
        stack.display();

    try { // popOnEmpty 异常的 try 块
        if ( ix % 10 == 0 ) {
            int dummy;
            stack.pop( dummy );
            stack.display();
        }
    }
    catch ( popOnEmpty ) { ... }
}
```

我们实现的程序能够工作正常。但是，它的组织结构把异常处理和程序正常处理混在一起，因而不太理想。毕竟，异常是程序的非正常事件出现的情况。应把处理程序异常的代码与栈的正常操作的实现分离开，因为我们相信这个策略会使得代码更易于跟随和维护。下面是较好的方案：

```
try {
    for ( int ix = 1; ix < 51; ++ix )
    {
        if ( ix % 3 == 0 )
            stack.push( ix );

        if ( ix % 4 == 0 )
            stack.display();

        if ( ix % 10 == 0 ) {
            int dummy;
            stack.pop( dummy );
            stack.display();
        }
    }
}
catch ( pushOnFull ) { ... }
catch ( popOnEmpty ) { ... }
```

与 try 块相关联的是两个 catch 子句，它们能够处理 pushOnFull 和 popOnEmpty 异常，这两个异常可能会被 try 块中调用的 iStack 的成员函数 pop()和 push()抛出。每个 catch 子句在

括号中指定了它所处理的异常的类型。处理异常的代码被放在 catch 子句的复合语句中（在花括号之间）。我们将在下一节更详细地探讨 catch 子句。

在我们的例子中，程序的控制流是下列几种情况之一：

1．如果没有异常发生，则执行 try 块中的代码，和 try 块相关联的处理代码被忽略。程序 main()返回 0。

2．如果在 for 循环的第一个 if 语句中调用的成员函数 push()抛出一个异常，则 for 循环的第二个和第三个 if 语句被忽略，该 for 循环和 try 块被退出，执行 pushOnFull 类型异常的处理代码。

3．如果在 for 循环的第三个 if 语句中调用的成员函数 pop()抛出一个异常，则针对 display()的调用被忽略，for 循环和 try 块被退出，执行 popOnEmpty 类型异常的处理代码。

当某条语句抛出异常时，跟在该语句后面的语句将被跳过。程序执行权被转交给处理异常的 catch 子句。如果没有 catch 子句能够处理该异常，则程序执行权又将被转交给 C++标准库中定义的函数 terminate()。我们将在下一节讨论函数 terminate()。

try 块可以包含任何 C++语句——表达式以及声明。一个 try 块引入一个局部域，在 try 块内声明的变量不能在 try 块外被引用，包括在 catch 子句中。例如，我们可以重写函数 main()使得变量 stack 的声明出现在 try 块中。在这种情况下，在 catch 子句中不能引用 stack：

```
int main() {
    try {
        iStack stack( 32 );// ok: 在try块中声明
        stack.display();
        for ( int ix = 1; ix < 51; ++ix )
        {
            // 同上
        }
    }
    catch ( pushOnFull ) {
        // 这里不能引用stack
    }
    catch ( popOnEmpty ) {
        // 这里不能引用stack
    }
    // 这里不能引用 stack
    return 0;
}
```

我们也可以声明整个包含在 try 块中的函数。在这种情况下，我们不是把 try 块放在函数定义的内部，而是把函数体整个包含在一个函数 try 块（function try block）中。这种组织结构把程序的正常处理代码和异常处理代码分离得最为清楚。例如：

```
int main()
try {
    iStack stack( 32 );
    stack.display();
    for ( int ix = 1; ix < 51; ++ix )
    {
        // 与以前相同
```

```
        }
        return 0;
    }
    catch (pushOnFull) {
        // 这里不能引用 stack
    }
    catch ( popOnEmpty ) {
        // 这里不能引用 stack
    }
```

注意，关键字 try 在函数体的开始花括号之前，catch 子句列在函数体结束花括号之后。通过这种代码组织方式，main()中正常处理的代码被放在函数体中，与 catch 子句中处理异常的代码清楚地分开。但是，在 main()的函数体中声明的变量不能在 catch 子句中被引用。

一个函数 try 块把一组 catch 子句同一个函数体相关联。如果函数体中的语句抛出一个异常，则考虑用跟在函数体后面的处理代码来处理该异常。函数 try 块对类构造函数尤其有用。我们将在第 19 章重新回顾这种上下文环境中的函数加块。

练习 11.3

请编写一个程序，使它定义一个 IntArray 对象（这里 IntArray 是在 2.3 节中定义的 class 类型）并执行下列动作。我们有三个含有整型值的文件。

1．读取第一个文件，把读入的第 1、3、5……n 个（这里 n 为奇数）数值赋给 IntArray 对象，然后显示 IntArray 的内容。

2．读取第二个文件，并把读入的第 5、10……n 个（这里 n 为 5 的倍数）数值赋给 IntArray 对象，然后显示 IntArray 的内容。

3．读取第三个文件，把读入的第 2、4、6……n 个（这里 n 为偶数）数值赋给 IntArray 对象，然后显示 IntArray 的内容。

用练习 11.2 定义的 IntArray operator[]()向 IntArray 对象读写值。因为 operator[]()可能抛出异常，所以要在的程序中用一个或多个 try 块和 catch 子句以处理。operator[]()可能抛出的异常。说明你在程序中放置 try 块的位置的原因。

11.3 捕获异常

C++异常处理代码是 catch 子句（catch clause）。当一个异常被 try 块中的语句抛出时，系统通过查看跟在 try 块后面的 catch 子句列表，来查找能够处理该异常的 catch 子句。

一个 catch 子句由三部分构成：关键字 catch、在括号中的单个类型或单个对象声明（被称作异常声明，exception declaration）以及复合语句中的一组语句。如果选择了一个 catch 子句来处理一个异常，则执行相应的复合语句。让我们详细地看看 main()函数中的 pushOnFull 和 popOnEmpty 异常的 catch 子句。

```
    catch ( pushOnFull ) {
        cerr << "trying to push a value on a full stack\n";
        return errorCode88;
```

```
    }
    catch ( popOnEmpty ) {
        cerr << "trying to pop a value on an empty stack\n";
        return errorCode89;
    }
```

两个 catch 子句都有一个 class 类型的异常声明：第一个是 pushOnFull 类型，第二个是 popOnEmpty 类型。如果异常声明的类型与被抛出的异常类型匹配，则选择这段处理代码来处理异常。（我们将在第 19 章看到类型不必完全匹配：基类的处理代码可以处理从异常声明类型派生出来的 class 类型的异常。）例如，当 iStack 的成员函数 pop()抛出一个类型为 popOnEmpty 的异常时，则进入第二个 catch 子句。在发出一个错误消息给 cerr 之后，函数 main()返回 errorCode89。

如果这些 catch 子句不包含返回语句，那么程序的执行将继续到哪儿呢？在 catch 子句完成它的工作之后，程序的执行将在 catch 子句列表的最后子句之后继续进行。在我们的例子中，程序的执行在 main()的返回语句处继续，在 popOnEmpty 的 catch 子句向 cerr 产生一个错误消息之后，main()返回 0。

```
int main() {
    iStack stack( 32 );
    try {
        stack.display();
        for ( int ix = 1; ix < 51; ++ix )
        {
            // 同前
        }
    }
    catch ( pushOnFull ) {
        cerr << "trying to push a value on a full stack\n";
    }
    catch ( popOnEmpty ) {
        cerr << "trying to pop a value on an empty stack\n";
    }
    // 程序在这里继续
    return 0;
}
```

C++的异常处理机制被称为是**不可恢复的**（nonresumptive）：一旦异常被处理，程序的执行就不能够在异常被抛出的地方继续。在我们的例子中，一旦异常被处理，程序的执行就不能够在 pop()成员函数中异常被抛出的地方继续。

11.3.1 异常对象

catch 子句的异常声明可以是一个类型声明或一个对象声明。什么时候 catch 子句中的异常声明应该声明一个对象？当我们要获得 throw 表达式的值，或者要操纵 throw 表达式所创建的异常对象时，我们应该声明一个对象。假设我们设计自己的异常类，当该异常被抛出时，我们把信息存储在异常对象中，如果 catch 子句的异常声明声明了一个对象，则 catch 子句中的语句就可以用该对象来引用由 throw 表达式存储的信息。

例如，我们改变 pushOnFull 异常类的设计。我们在该异常对象中保存不能被压入到栈中的值。修改 catch 子句，向 cerr 发出错误信息时显示这个值。为了实现它，我们首先需要改变 pushOnFull 类的定义。下面是我们的新定义：

```
// 新异常类：
// 负责保存不能被压入到栈中的值
class pushOnFull {
public:
     pushOnFull( int i ) : _value( i ) { }
     int value() { return _value; }
private:
     int _value;
};
```

新的私有数据成员_value 拥有不能被压入栈中的那个值。构造函数取一个 int 型的值，把这个值存储在_value 数据成员中。下面是构造函数怎样被 throw 表达式调用，以便把不能压入到栈中的那个值存储在异常对象中：

```
void iStack::push( int value )
{
     if ( full() )
          // 把 value 存储在异常对象中
          throw pushOnFull( value );
     // ...
}
```

类 pushOnFull 也有一个新的成员函数 value()，它可以被用在 catch 子句中，以便显示异常对象中的值。下面是它的用法：

```
catch ( pushOnFull eObj ) {
     cerr << "trying to push the value " << eObj.value()
          << " on a full stack\n";
}
```

注意，catch 子句的异常声明声明了对象 eObj，用它来调用 pushOnFull 类的成员函数 value()。

异常对象总是在抛出点被创建，即使 throw 表达式不是一个构造函数调用，或者它没有表现出要创建一个异常对象，情况也是如此。例如：

```
enum EHstate { noErr, zeroOp, negativeOp, severeError };
enum EHstate state = noErr;

int mathFunc( int i ) {
     if ( i == 0 ) {
          state = zeroOp;
          throw state; // 创建异常对象
     }

     // 否则，正常处理流程继续
}
```

在这个例子中，对象 state 没有被用作异常对象。而是由 throw 表达式创建了一个类型为 EHstate 的异常对象，并且用全局对象 state 的值初始化该对象。程序是怎样分辨出该异常对

象不同于全局对象 state 呢？为了回答这个问题，我们必须先仔细地看看 catch 子句的异常声明。

catch 子句异常声明的行为特别像参数声明。当进入 catch 子句时，如果异常声明声明了一个对象，则用该异常对象的拷贝初始化这个对象。例如，下面的函数 calculate()调用前面定义的函数 mathFunc()。当进入 calculate()中的 catch 子句时，对象 eObj 由 throw 表达式创建的异常对象的拷贝进行初始化：

```
void calculate( int op ) {
    try {
        mathFunc( op );
    }
    catch ( EHstate eObj ) {
        // eObj 是被抛出的异常对象的拷贝
    }
}
```

这个例子中的异常声明类似于按值传递的参数。对象 eObj 用该异常对象的值初始化，就好像一个按值传递的函数参数用相应实参的值初始化一样（关于按值传递的参数的讨论见 7.3 节）。

与函数参数的情形一样，catch 子句中的异常声明也可以被改变成引用声明。于是，catch 子句就可以直接引用由 throw 表达式创建的异常对象，而不是创建一个局部拷贝了。例如：

```
void calculate( int op ) {
    try {
        mathFunc( op );
    }
    catch ( EHstate &eObj ) {
        // eObj 引用了被抛出的异常对象
    }
}
```

为了防止不必要地拷贝大型类对象，class 类型的参数应该被声明为引用。同样原因，如果 class 类型异常的异常声明被声明为引用，也是比较好的。

使用引用类型的异常声明，catch 子句能够修改异常对象。但是，由 throw 表达式指定的任何变量仍都不受影响。例如，在 catch 子句中修改 eObj 对象，不会影响由 throw 表达式指定的全局变量 state：

```
void calculate( int op ) {
    try {
        mathFunc( op );
    }
    catch ( EHstate &eObj ) {
        // 修正异常情况
        eObj = noErr; // 全局变量 state 没有被修改
    }
}
```

在修正异常情况之后，这个 catch 子句将对 eObj 重置为 noErr。因为 eObj 是个引用，所以我们可以期望该赋值修改全局变量 state。但是，该赋值只修改由 throw 表达式创建的异常对象。因为异常对象与全局变量 state 不同，所以修改了 catch 子句中的 eObj 后，state 仍保其

个变。

11.3.2 栈展开

找到一个 catch 子句，以处理被抛出的异常的过程如下：如果 throw 表达式位于 try 块中，则检查与 try 块相关联的 catch 子句，看是否有一个子句能够处理该异常。如果找到一个 catch 子句，则该异常被处理。如果没有找到 catch 子句，则在主调函数中继续查找。如果一个函数调用在退出时带着一个被抛出的异常，并且这个调用位于一个 try 块中，则检查与该 try 块相关联的 catch 子句，看是否有一个子句能够处理该异常。如果找到了一个 catch 子句，则该异常被处理。如果没有找到 catch 子句，则查找过程在主调函数中继续。这个过程沿着嵌套函数调用链向上继续，直到找到该异常的 catch 子句。只要一遇到能够处理该异常的 catch 子句，就会进入该 catch 子句，程序的执行在该处理代码中继续。

在我们的例子中，查找 catch 子句的第一个函数是 iStack 类的成员函数 pop()。因为 pop()中的 throw 表达式没有在 try 块中，所以 pop()带着一个异常而退出。要检查的下一个函数是调用成员函数 pop()的函数，在我们的例子中，它是 main()。在 main()中的 pop()调用位于一个 try 块中。系统考虑由与该 try 块关联的 catch 子句来处理该异常。一个 popOnEmpty 类型的异常的 catch 子句被找到，并进入它以处理该异常。

在查找用来处理被抛出异常的 catch 子句时，因为异常而退出复合语句和函数定义，这个过程被称作**栈展开**（stack unwinding）。随着栈的展开，在退出的复合语句和函数定义中声明的局部变量的生命期也结束了。C++保证，随着栈的展开，尽管局部类对象的生命期是因为抛出异常而被结束，但是这些局部类对象的析构函数也会被调用。我们将在第 19 章更详细地介绍这些内容。

如果一个程序没有为已被抛出的异常提供 catch 子句，该怎么办呢？异常不能够保持在未被处理的状态。异常对于一个程序非常重要，它表示程序不能够继续正常执行。如果没有找到处理代码，程序就调用 C++标准库中定义的函数 terminate()。terminate()的缺省行为是调用 abort()，指示从程序非正常退出。[在大多数情况下，调用 abort()已经足够了。但是在某些特殊情况下，我们有必要改变由 terminate()执行的动作。[STROUSTRUP97]给出了怎样做到这一点，并作了详细的讨论。]

到目前为止，你可能已经注意到在异常处理和函数调用之间的许多相似之处。throw 表达式的行为有点像函数调用，而 catch 子句有点像函数定义。这两种机制的一个主要区别是：建立函数调用所需要的全部信息在编译时刻已经获得，而对异常处理机制则不然。C++异常处理要求运行时刻的支持。例如，对于一个普通函数调用，通过函数重载解析过程，编译器知道在调用点上哪个函数会被真正调用。但是对于异常处理，编译器不知道特定的 throw 表达式的 catch 子句在哪个函数中，以及在处理异常之后执行权被转交到哪儿。这些决策必须在运行时刻进行。当一个异常不存在处理代码时，编译器无法通知用户。这就是为什么 terminate()函数存在的原因：它是一种运行时刻机制，当没有处理代码能够匹配被抛出的异常时由它通知用户。

11.3.3 重新抛出

在异常处理过程中也可能存在“单个 catch 子句不能完全处理异常”的情况。在某些修

正动作之后，catch 子句可能决定该异常必须由函数调用链中更上级的函数来处理，那么 catch 子句可以通过**重新抛出**（rethrow）该异常，把异常传递给函数调用链中更上级的另一个 catch 子句。rethrow 表达式的形式为：

```
throw;
```

rethrow 表达式重新抛出该异常对象，rethrow 只能出现在 catch 子句的复合语句中。例如：

```
catch ( exception eObj ) {
    if ( canHandle( eObj ) )
        // 处理异常
        return;
    else
        // 重新抛出它，并由另一个 catch 子句来处理
        throw;
}
```

被重新抛出的异常就是原来的异常对象。如果 catch 子句在重新抛出异常对象之前对它作了修改，那么这会有某些隐含的意义。下列代码没有修改原来的异常对象，你能看出为什么吗？

```
enum EHstate { noErr, zeroOp, negativeOp, severeError };

void calculate( int op ) {
    try {
        // 被 mathFunc() 抛出的异常的值为 zeroOp
        mathFunc( op );
    }
    catch ( EHstate eObj ) {
        // 做某些修正
        // 试图修改异常对象
        eObj = severeErr;

        // 希望重新抛出值为 severeErr 的异常对象
        throw;
    }
}
```

因为 eObj 不是引用，所以 catch 子句接收到的是异常对象的拷贝，在处理代码中对 eObj 所做的任何修改都只是改变了局部拷贝。它们不影响由 throw 表达式创建的原来的异常对象。因为在我们的例子中 catch 子句里没有修改原来的异常，所以被重新抛出的对象仍然具有初始的 zeroOp 值。

为了修改原来的异常对象，catch 子句中的异常声明必须被声明为引用。例如：

```
cacth ( EHstate &eObj ) {
    // 修改异常对象
    eObj = severeErr;

    // 被重新抛出的异常的值是 severeErr
    throw;
}
```

eObj 指向由 throw 表达式创建的异常对象，在 catch 子句中对 eObj 的修改影响了原来的异常对象。这些修改是被重新抛出的异常对象的一部分。

所以，把“catch 子句的异常声明”声明为引用的另一个原因是，确保应用在 catch 子句中的异常对象上的修改操作，能够反映到被重新抛出的异常对象上。我们将在 19.2 节了解“为什么 class 类型异常的异常声明应该是一个引用”的另外一个原因，在那里我们将了解 catch 子句怎样调用类的虚拟函数。

11.3.4 catch-all 处理代码

即使一个函数不能处理被抛出的异常，但是它也可能希望在带着异常退出之前执行一些动作。例如，函数可能获得了一些资源，如打开一个文件或在堆中分配了一些内存，它可能想在随着异常退出之前释放这些资源（关闭文件或释放内存）。例如：

```
void mainip() {
     resource res;
     res.lock();   // 锁定资源

     // 使用 res
     // 可能引起异常抛出的动作
     res.release();     // 如果抛出异常则跳过
}
```

如果有一个异常被抛出，则资源 res 的释放被跳过去。为保证该资源被释放，我们不是为每种可能的异常都写一个 catch 子句，因为我们不知道可能被抛出的全部异常，但是我们可以使用 catch 子句 catch-all。这种 catch 子句有一个形式为（...）的异常声明，这里的二个点被称为省略号（ellipsis）。对任何类型的异常，都会进入这个 catch 子句。例如：

```
// 对任何异常都会进入
catch ( ... ) {
     // 这里是我们的代码
}
```

catch(...)和 throw 表达式被组合起来使用。对于已经被锁定的资源，在异常被一个 rethrow 表达式传递给函数调用链中更上级的函数之前，它们在 catch 子句的复合语句中被释放：

```
void manip() {
     resource res;
     res.lock();

     try {
          // 使用 res
          // 某些能够引起异常被抛出的动作
     }
     catch ( ... ) {
          res.release();
          throw;
     }
     res.release();     // 如果抛出异常则跳过
}
```

为了确保该资源被正确地释放，如果一个异常被抛出并且 manip()随着异常而退出，则在

异常被传递给函数调用链的上层函数之前，我们可以用一个 catch(...)来释放该资源。我们也可以通过把资源封装在一个类中，以便管理资源的请求和释放，类的构造函数获得资源，而类的析构函数自动释放资源。我们将在第 19 章了解怎样实现这种策略。

catch(...)可以自己单独使用，也可以与其他 catch 子句联合使用。如果它与其他 catch 子句联合使用，那么在组织与 try 块相关的一组 catch 子句时我们必须小心。

catch 子句被检查的顺序与它们在 try 块之后出现的顺序相同。一旦找到了一个匹配，则后续的 catch 子句将不再检查。这意味着，如果 catch(...)与其他 catch 子句联合使用，它必须总是被放在异常处理代码表的最后，否则就会产生一个编译时刻错误。例如：

```
try {
     stack.display();
     for (int ix = 1; ix < 51; ++ix )
     {
          //  与前面相同
     }
}
catch ( pushOnFull ) {}
catch ( popOnEmpty ) { }
catch (...) { }    // 必须是最后一个 catch 子句
```

练习 11.4

请说明为什么说 C++的异常处理机制是不可恢复的。

练习 11.5

已知下列异常声明，请给出一个 throw 表达式，它可以创建一个能够被下列 catch 子句捕获的异常对象。

```
(a)  class exceptionType { };
     catch( exceptionType *pet ) {  }
(b)  catch( ... ) {  }
(c)  enum mathErr { overflow, underflow, zeroDivide }
     catch( mathErr &ref ) { }
(d)  typedef int EXCPTYPE;
     catch( EXCPTYPE ) {  }
```

练习 11.6

说明在栈展开过程中发生的事情。

练习 11.7

请给出 catch 子句的异常声明应该被声明为引用的两个原因。

练习 11.8

请用练习 11.3 开发的代码，修改你创建的异常类，以便将 operator[]()的非法索引存储在异常对象中，当该异常被抛出时，能够在后面用 catch 子句显示它。修改你的程序使得

operator[]()在程序执行期间抛出一个异常。

11.4 异常规范

通过查看 iStack 类的成员函数 pop()和 push()的声明，来判断这些函数可能会抛出异常，这是不可能的。一种可能的方案是，在每个成员函数的声明处附加上相关的注释。通过这种方式，出现在头文件中的类接口也给类成员函数可能抛出的异常做了文档：

```
class iStack {
public:
    // ...
    void pop( int &value ); // 抛出 popOnEmpty
    void push( int value ); // 抛出 pushOnFull

private:
    // ...
};
```

但这还是不太理想，因为无法保证该文档会随着 istack 类以后的发行而自动更新。它没有向编译器提供信息保证不会抛出其他种类的异常。**异常规范**（exception specification）提供了一种方案，它能够随着函数声明列出该函数可能抛出的异常。它保证该函数不会抛出任何其他类型的异常。

异常规范跟随在函数参数表之后。它用关键字 throw 来指定，后面是用括号括起来的异常类型表。例如，我们可以如下修改 iStack 类的成员函数的声明，以增加适当的异常规范：

```
class iStack {
public:
    // ...
    void pop( int &value ) throw(popOnEmpty);
    void push( int value ) throw(pushOnFull);

private:
    // ...
};
```

对于 pop()的调用，保证不会抛出任何 popOnEmpty 类型之外的异常。类似地，对于 push()的调用，保证不会抛出任何 pushOnFull 类型之外的异常。

异常声明是函数接口的一部分，它必须在头文件中的函数声明上指定。异常规范是函数和程序余下部分之间的协议。它保证该函数不会抛出任何没有出现在其异常规范中的异常。

如果函数声明指定了一个异常规范，则同一函数的重复声明必须指定同一类型的异常规范。同一函数的不同声明上的异常规范是不能累积的。例如：

```
// 同一函数的两个声明
extern int foo( int = 0 ) throw(string);

// 错误：异常规范被省略
extern int foo( int parm ) {  }
```

如果函数抛出了一个没有被列在异常规范中的异常会怎么样？程序只有在遇到某种不正常情况时，异常才会被抛出，在编译时刻编译器不可能知道，在执行时程序是否会遇到这些异常，因此，一个函数的异常规范的违例只能在运行时刻才能被检测出来。如果函数抛出了一个没有被列在其异常规范中的异常，则系统调用 C++标准库中定义的函数 unexpected()。unexpected()的缺省行为是调用 terminate()。[在某些条件下，可能有必要改变 unexpected()执行的动作。C++标准库提供了一种机制，可让我们改变 unexpected()的缺省行为。[STROUSTRUP97]更详细地讨论了这些。]

我们应该澄清一下，如果函数抛出了一个没有被列在其异常规范中的异常，系统未必就会调用 unexpected()。如果该函数自己处理该异常，并且该异常在“逃离”该函数之前被处理掉，那么一切都不会有问题。例如：

```
void recoup( int op1, int op2 ) throw(ExceptionType)
{
     try {
          // ...
          throw string("we're in control");
     }
     // 处理抛出的异常
     catch ( string ) {
          // 做一些必要的工作
     }
}    // ok, unexpected()没有被调用
```

即便在函数 recoup()中抛出 string 类型的异常，而且函数 recoup()保证不会抛出 ExceptionType 类型之外的其他异常，但是因为该异常在其“逃离”函数 recoup()之前被处理了，所以系统不会由于该函数抛出 string 类型的异常而调用函数 unexpected()。

函数异常规范的违例只有在运行时刻才能被检测到。如果一个表达式能够抛出一个不被规范允许的异常类型，则编译器不会产生编译时刻错误。如果这个表达式不会被执行，或者它从没有抛出违反异常规范的那个异常，则该程序会像期望的那样运行，而且该函数异常规范从不会被违反。例如：

```
extern void doit( int, int ) throw(string, exceptionType);
void action ( int op1, int op2) throw(string) {
     doit( op1, op2 ); // 没有编译错误
     // ...
}
```

函数 doit()可以抛出一个 exceptionType 类型的异常，它不是函数 action()的异常规范所允许的。即使函数 action()不允许这种类型的异常，该函数也能编译成功。编译器产生相应的代码以确保当违反异常规范的异常被抛出时。调用运行库函数 unexpected()。

空的异常规范保证函数不会抛出任何异常。例如，函数 no_problem()保证不会抛出任何异常：

```
extern void no_problem() throw();
```

如果一个函数声明没有指定异常规范，则该函数可以抛出任何类型的异常。

在被抛出的异常类型与异常规范中指定的类型之间不允许类型转换。例如：

```
int convert( int parm ) throw(string)
{
    // ...
    if ( somethingRather )
        // 程序错误:
        // convert() 不允许 const char* 型的异常
        throw "help!";
}
```

在函数 convert()中的 throw 表达式抛出一个 C 风格的字符串。由这个 throw 表达式创建的异常对象的类型为 const char*。通常，const char*型的表达式可以被转换成 string 类型。但是，异常规范不允许从被抛出的异常类型到异常规范指定的类型之间的转换，如果 convert()抛出该异常，则调用函数 unexpected()。为了修正这种情况，可以如下修改 throw 表达式，显式地把表达式的值转换成 string 类型：

```
throw string( "help!" );
```

11.4.1 异常规范与函数指针

我们也可以在函数指针的声明处给出一个异常规范。例如：

```
void (*pf) (int) throw(string);
```

该声明表示 pf 是一个函数指针，它只能抛出 string 类型的异常。和函数声明一样，同一指针的不同异常规范不能累积，指针 pf 的所有声明都必须指定相同的规范。例如：

```
extern void (*pf)( int ) throw(string);

// 错误: 缺少异常规范
void (*pf) ( int );
```

当带有异常规范的函数指针被初始化（或被赋值）时，对于用作初始值（或用作赋值右边的右值）的指针类型有一些限制。这两个指针的异常规范不必完全一样。但是，用作初始值或右值的指针异常规范必须与被初始化或赋值的指针异常规范一样或更严格。例如：

```
void recoup( int, int ) throw(exceptionType);
void no_problem() throw();
void doit( int, int ) throw(string, exceptionType);

// ok: recoup() 与 pf1 的异常规范一样严格
void (*pf1)( int, int ) throw(exceptionType) = &recoup;

// ok: no_problem() 比 pf2 更严格
void (*pf2)() throw(string) = &no_problem;

// 错误: doit()没有 pf3 严格
void (*pf3)( int, int ) throw(string) = &doit;
```

第三个初始化没有意义。该指针的声明保证 pf3 指向一个函数，该函数不会抛出除了 string 类型之外的任何异常。但是函数 doit()可能抛出一个 exceptionType 类型的异常。因为函数 doit()不能满足 pf3 的异常规范的保证，所以，函数 doit()不是 pf3 的合法初始值，因而会产生一个编译错误。

练习 11.9

请用练习 11.8 开发的代码，修改类 IntArray 的 operator[]()声明，加入适当的异常规范来描述这个操作符可能抛出的异常。修改你的程序，使 operator[]()抛出一个没有被列在异常规范中的异常。那么，会怎么样？

练习 11.10

如果函数有一个形式为 throw()的异常规范，那么它可以抛出什么异常？如果没有异常规范呢？

练习 11.11

下列哪些指针赋值是错误的？为什么？

```
void example() throw(string);
(a)  void (*pf1)() = example;
(b)  void (*pf2)() throw() = example;
```

11.5 异常与设计事项

在 C++程序设计中，有一些和异常处理的用法相关的设计事项。虽然对于异常处理的支持是被内置在语言中的，但并不是每个 C++程序都应该使用异常处理。因为抛出异常不像正常函数调用那样快，所以异常处理应该用在独立开发的不同程序部分之间，用于不正常情况的通信。例如，一个库的实现者可能决定用异常向库用户通知程序的异常情况。如果库函数遇到一种不能局部处理的意外情况，它可能会抛出一个异常通知使用该库的程序。

在我们的例子中，的库定义了 iStack 类及其成员函数。函数 main()使用这个库，我们应该假设写 main()的程序员不是库的实现者。类 iStack 的成员函数能够检测到在一个空栈上的 pop()操作请求，或在一个满的栈上的 push()操作请求，但是库的实现者不知道引起 pop()或 push()操作请求的程序状态，所以不能够在编写 pop()和 push()成员函数时，在局部函数内解决这种情况。因为这些错误不能在成员函数中被处理，所以我们决定抛出异常，以便通知使用该库的程序。

即使 C++支持异常处理，C++程序仍然应该使用其他的错误处理技术（比如在适当时返回一个错误代码。对“错误何时会变成异常”这个问题，没有明确的答案。确定什么是一种“意外的情况”，这实际上是库的实现者的责任。异常是库接口的一部分，决定一个库抛出哪些异常是库设计的一个重要阶段。如果该库希望被用在不会崩溃的程序中，那么该库必须自己处理问题，或者如果它不能处理的话，则必须把程序的不正常情况通知给使用该库的程序部分，在库代码本身没有可采取的有意义动作时，让调用者选择应该采取什么行动。决定“把哪些情况应该当作异常来处理”是库设计中很难的一部分。

在我们的 iStack 的例子中，成员函数 push()在栈满时是否应该抛出一个异常是有争议的。有些人可能会说这样更好：push()的实现可以局部地处理这种情况，在栈满时仍增长栈。毕

竟，惟一真正的限制是我们程序的可用内存。“在栈满时压入一个值就抛出异常”的决定可能是一个错误的考虑。我们则以重新实现成员函数 push()，在向一个满栈请求压入一个值时继续增长栈：

```
void iStack::push( int value)
{
    // 如果满，增长底层的 vector
    if ( full() )
        _stack.resize( 2 * _stack.size() );
    _stack[ _top++ ] = value;
}
```

类似地，当要求从空栈中抛出一个值时，pop()是否应该抛出异常？一个有趣的观察是，C++标准库的 stack 类（在第 6 章介绍）在要求一个弹出动作、而栈为空时，并没有抛出异常，而是这个操作有一个未定义行为：它不知道在要求这样的操作之后程序的行为是什么。在设计 C++标准库时，显然已经确定在这种情况下不应该抛出异常。“在遇到非法状态时，允许程序继续进行”在这种情况下被认为是合适的。正如我们所提到的，不同的库会有不同的异常。对于“什么构成了一个异常”的问题还没有正确的答案。

不是所有的程序都应该担心库会抛出异常。尽管有些系统不能忍受宕机的风险，因而应该处理异常事件情况，但不是每个系统都有这样的要求。异常处理是容错系统实现中的主要辅助手段。决定一个程序是否处理由库抛出的异常，或是让程序终止运行，是设计过程中很难的一部分。

程序设计中有关异常处理的最后一方面是，程序中的异常处理通常是分层的。一个程序通常是由一些组件构成，每个组件必须决定它将处理哪些异常，应该将哪些异常传递给程序的上一层。我们的组件指的是什么？例如，第 6 章介绍的文本查询系统可以分成三个组件或层。第一层是 C++标准库，它提供对 string、map 等等的基本操作的支持。第二层是文本查询系统本身，它定义了函数，如 string．caps()和 suffix_text()，它们操纵要被处理的文本并把 C++标准库用作于组件。第三层是使用文本查询系统的程序。每个组件或层被独立生成，并且都必须决定哪些异常情况它会本地处理，哪些异常会传递给程序的高层。

在一个层或组件中，不是每个函数都应该能够处理异常。通常，try 块和 catch 子句被一个程序组件的入口点函数使用。catch 子句处理当前组件中不适合传递给高层程序的异常。异常规范（11.4 节讨论）也用于一个组件的入口函数中，确保不希望传递给上一层程序的异常不会“逃离”。

我们将在第 19 章介绍了类和类层次结构之后，了解有关异常的程序设计的其他方面。

12

泛型算法

在第 2 章的 Array 类的实现中，我们提供了支持 min()、max()、find()和 sort()的成员操作。但是，标准 vector 类并没有提供这些显然很基本的操作。为了在 vector 的元素中找到最小或最大值，我们必须调用一个泛型算法（generic algorithm）：“算法”是因为它们实现公共的操作，如 min()、max()、find()和 sort()，“泛型”是因为它们操作在多种容器类型上——例如，不但有 vector 和 list 类型，还有内置数组类型。容器通过一对 iterator（迭代器，我们在 6.5 节简要讨论了 iterator）被绑定到某个泛型算法上，这对 iterator 标记了要遍历的元素范围、特殊的函数对象（function object）允许我们改变泛型算法的缺省操作语义。泛型算法、函数对象以及 iterator 的详细介绍形成了本章的主题。

12.1 概述

每个泛型算法的实现都独立于单独的容器类型。因为已经消除了算法的类型依赖性，所以单个的模板实例可以操作在各种容器以及内置数组类型上。考虑 find()算法，因为它独立于被适用的容器，所以它只要求下列一般性的步骤，这里假设资料集合未经排序。

1．顺次检查每个元素。

2．如果当前元素等于要被查找的值，那么返回该元素在集合中的位置。

3．否则，检查下一个元素，重复步骤 2，直到找到一个元素，或者检查完所有元素。

4．如果已经到了集合的末尾，而且还没有找到该值，则返回某个值指明该值在这个集合中不存在。

这个算法，正如我们所指出的，与被应用的容器类型以及被查找的值的类型无关。算法的要求如下：

1．我们需要某种遍历集合的方式，这包括“向前移到下一个元素，以及识别下一元素是否是末元素”的概念。典型情况下，对了内置数组类型（除了 C 风格字符串以外），我们传递两个实参来解决这个问题：首元素的指针、以及要遍历的元素的个数。对于 C 风格字符串，元素个数是不必要的，字符串的末尾由一个终止空字符来指示。

2．我们需要能够对容器中的元素与被查找元素进行比较。典型情况下，这可以通过使用与其关联的底层类型的“等于”操作符，或者传递一个执行该操作的函数的指针来解决。

3．我们需要一个公共类型来表示元素在容器中的位置，以及如果没有找到时使用的**无位置**（no position）。典型情况下，我们返回元素的索引、-1，或者指向该元素的指针或 0。

泛型算法用 iterator 抽象来解决第一个问题：对容器的遍历。iterator 提供了对指针的一个泛化。它至少支持下列操作符：递增操作符以用来访问下一个元素、解引用操作符用来访问实际的元素，以及等于和不等于操作符用来判定两个 iterator 是否相等。算法遍历的元素范围由一对 iterator 标记：一个 first iterator 指向要操作的首元素，和一个 last iterator 标记要操作的末元素的下一位置。由 last 指向的元素，不是要操作的元素。它被用作终止遍历的**哨兵**（sentinel），同时也被用作指示没有找到的返回值。如果找到了该值，则返回标记该位置的 iterator。

泛型算法解决第二个要求（值的比较）所用的方法是，为每个算法提供两个版本：一个使用元素底层类型的等于操作符，另一个使用函数对象或函数指针来实现比较（关于函数对象将在 12.3 节解释）。例如，下面是 find()的泛化实现，它使用了底层类型的等于操作符：

```
template < class ForwardIterator, class Type >
     ForwardIterator

find( ForwardIterator first, ForwardIterator last, Type value )
{
     for ( ; first != last; ++first )
          if ( value == *first )
               return first;
     return last;
}
```

ForwardIterator 是标准库预定义的五种 iterator 之一。ForwardIterator 支持读写它所指向的元素（这五种 iterator 将在 12.4 节给出。）

由于这个算法不直接访问容器的元素，因而获得了类型独立性，元素的全部访问和遍历都通过 iterator 实现。实际的容器类型（可能是一个容器类型，也可能是内置数组）未知。为支持内置数组类型，普通指针以及 iterator 都可以被传递给泛型算法。例如，下面的例子用 int 型的内置数组来使用 find()：

```
#include <algorithm>
#include <iostream>

int main()
{
     int search_value;
     int ia[ 6 ] = { 27, 210, 12, 47, 109, 83 };
     cout << "enter search value: ";
     cin >> search_value;
     int *presult = find( &ia[0], &ia[6], search_value );

     cout << "The value " << search_value
          << ( presult == &ia[6]
               ? " is not present" : " is present" )
```

```
            << endl;
    }
```

如果返回的指针等于 ia[6]的地址（即 ia 末元素的下一位置），则查找失败；否则，相应的值就被找到。

通常，向泛型算法传递指针时，我们可以写成：

```
int *presult = find( &ia[0], &ia[6], search_value );
```

或不太明确地写成：

```
int *presult = find( ia, ia+6, search_value );
```

如果希望传递一个子范围，我们只需修改传递给算法的地址索引。例如，在 find()的调用中，只查找第二个和第三个元素（记住元素从 0 开始计数）：

```
// 只查找元素 ia[1] 和 ia[2]
int *presult = find( &ia[1], &ia[3], search_value );
```

下面的例子用 vector 容器类型使用 find()：

```
#include <algorithm>
#include <vector>
#include <iostream>

int main()
{
    int search_value;
    int ia[ 6 ] = { 27, 210, 12, 47, 109, 83 };
    vector<int> vec( ia, ia+6 );

    cout << "enter search value: ";
    cin >> search_value;

    vector<int>::iterator presult;
    presult = find( vec.begin(), vec.end(), search_value );

    cout << "The value " << search_value
         << ( presult == vec.end()
              ? " is not present" : " is present" )
         << endl;
}
```

类似地，下面是对 list 容器类型的 find()用法：

```
#include <algorithm>
#include <list>
#include <iostream>

int main()
{
    int search_value;
    int ia[ 6 ] = { 27, 210, 12, 47, 109, 83 };
    list<int> ilist( ia, ia+6 );

    cout << "enter search value: ";
    cin >> search_value;
```

```
    list<int>::iterator presult;
    presult = find( ilist.begin(), ilist.end(), search_value );

    cout << "The value " << search_value
         << ( presult == ilist.end()
              ? " is not present" : " is present" )
         << endl;
}
```

在下一节中，我们将了解一个设计，它是一个用到了各种泛型算法的程序。在其后的小节中，我们将介绍函数对象。12.4 节将介绍关于 iterator 的更多细节。在 12.5 节中我们将简要地介绍泛型算法——每个算法的说明以及详细讨论被放到附录中。在本章最后，我们将讨论何时不宜使用泛型算法。

练习 12.1

对泛型算法的批评是：它的设计虽然很优雅，但却把正确性的责任放在程序员身上。例如，无效的 iterator 或标记了一个无效范围的 iterator 对，会导致未定义的运行时刻行为。这个批评对吗？对这些算法的使用应该只局限于很有经验的程序员吗？一般来说，程序员应该接受保护，以避免诸如泛型算法、指针和显式强制转换这样的语言结构中存在潜在的错误，是这样吗？

12.2 使用泛型算法

考虑如下的程序设计任务。我们想写一本儿童用书，希望知道适用于这本书的词汇层次。我们的想法如下：我们将阅读一定数量的儿童书籍，把其中的文本存储在 string vector 中（我们已经知道该怎样做——见 6.7 节）。下面是我们要做的：

1. 拷贝每个 vector。
2. 把 5 个 vector 合并成一个大的 vector。
3. 以字母顺序排列大的 vector。
4. 去掉所有重复的单词。
5. 再按单词的长度排序。
6. 计数超过 6 个字符的词的个数（长度是一个测量复杂度的依据，至少对词汇是这样）。
7. 去掉任何没有语义的中性词（如 and、if、or、but 等等）
8. 打印 vector。

这听起来像是要一章才能完成的工作。但是，使用泛型算法，我们可以把它缩短到本章的一个很短的小节中。

我们的函数的实参是一个 string vector 的 vector。我们以指针方式接受它，首先测试它是否非空：

```
#include <vector>
#include <string>

typedef vector<string> textwords;
```

```
void process_vocab( vector<textwords>*pvec )
{
    if ( ! pvec ) {
        // 给出警告信息
        return;
    }
    // ...
}
```

我们希望做的第一件事情是，创建一个 vector，它包含各个 vector 中的元素。我们可以用如下的 copy()泛型算法做到这一点（需要包含 algorithm 和 iterator 头文件）：

```
#include <algorithm>
#include <iterator>

void process_vocab( vector< textwords >*pvec )
{
    // ...
    vector< string > texts;
    vector<textwords>::iterator iter = pvec->begin();
    for ( ; iter != pvec->end(); ++iter )
        copy( (*iter).begin(), (*iter).end(),
            back_inserter( texts ));

    // ...
}
```

copy()算法把一对 iterator 当作前两个实参，用它们标记出要拷贝的元素范围。第三个实参是一个 iterator，它标记了被拷贝元素将被放置的起始位置。back_inserter 被称为 iterator 适配器：它使得元素被插入到作为实参的 vector 的尾部（我们将在 12.4 节详细查看 iterator 适配器）。

unique()虽然去掉了容器中的重复值，但是只去掉相邻的重复值。即，序列 01123211 的结果是 012321，而不是 0123。为了得到后一种结果，必须先对 vector 进行 sort()，即把序列 01111223 变成 0123。好，差不多了。实际上，结果是 01231223。

unique()的行为有些不符合直觉，它操作的容器的长度没有被改变。每个独一无二的元素被放到从头开始的下一个自由槽中。在我们的例子中，实际的结果是 01231223，序列 1223 表示的是算法的**废弃部分**（refuse）。unique()返回一个 iterator 指向这个废弃部分的开始处。典型情况下，这个 iterator 被传递给相关的容器操作 erase()来删除无效的元素。（因为内置数组不支持 erase()操作，所以 unique()算法族不太适合于内置数组类型。）下面是函数的一部分：

```
void process_vocab( vector< textwords >*pvec )
{
    // ...
    // 排序 texts 的元素
    sort( texts.begin(), texts.end() );

    // 删除重复的元素
```

```
    vector<string>::iterator it;
    it = unique( texts.begin(), texts.end() );
    texts.erase( it, texts.end() );
    // ...
}
```

下面是在 sort()之后但还没有调用 unique()之前、合并了两个小文本文件的 texts 的输出例子：

```
a a a a alice alive almost
alternately ancient and and and and and and
and as asks at at beautiful becomes bird
bird blows blue bounded but by calling coat
daddy daddy daddy dark darkened darkening distant each
either emma eternity falls fear fiery fiery flight
flowing for grow hair hair has he heaven,
held her her her her him him home
houses i immeasurable immensity in in in in
inexpressibly is is is it it it its
journeying lands leave leave life like long looks
magical mean more night, no not not not
now now of of on one one one
passion puts quite red rises row same says
she she shush shyly sight sky so so
star star still stone such tell tells tells
that that the the the the the the
the there there thing through time to to
to to trees unravel untamed wanting watch what
when wind with with you you you you
your your
```

在应用了 unique()，并调用了 erase()之后，texts vector 看起来是这样的：

```
a alice alive almost alternately ancient
and as asks at beautiful becomes bird blows
blue bounded but by calling coat daddy dark
darkened darkening distant each either emma eternity falls
fear fiery flight flowing for grow hair has
he heaven, held her him home houses i
immeasurable immensity in inexpressibly is it its journeying
lands leave life like long looks magical mean
more night, no not now of on one
passion puts quite red rises row same says
she shush shyly sight sky so star still
stone such tell tells that the there thing
through time to trees unravel untamed wanting watch
what when wind with you your
```

我们的下一个任务是按长度排序字符串。为实现它，我们不用 sort()而是用 stable_sort()算法。stable_sort()保留相等元素的相对位置，也就是说，对于长度相同的元素，当前的字母顺序被保留。为实现按长度排序，我们给出自己的小于等于操作，下面是一种实现方式：

```
bool less_than( const string & s1, const string & s2 )
{
    return s1.size() < s2.size();
```

```
}
void process_vocab( vector< textwords >*pvec )
{
    // ...

    // 按长度排序 texts 的元素
    // 保留元素的原始顺序
    stable_sort( texts.begin(), texts.end(), less_than );

    // ...
}
```

尽管这样已经完成了工作，但是比我们期望的效率要低得多。less_than()是作为单个语句而实现的。正常情况下，它应该被用作 inline 函数调用。但是，把它用作函数指针来传递，又阻止了它被 inline 的可能。替代的策略是使用函数对象来保留操作的 inline 特性。例如：

```
// 函数对象—小于操作被实现为 operator()的一个实例
class LessThan {
public:
    bool operator()( const string & s1, const string & s2 )
        { return s1.size() < s2.size(); }
};
```

函数对象是一个类，它重载了调用操作符（()）。调用操作符的函数体实现了函数的功能：小于比较。调用操作符的定义第一次看有点迷惑，因为出现了两个小括号。如下序列：

```
operator()
```

告诉编译器我们在重载调用操作符。第二对括号：

```
( const string & s1, const string & s2 )
```

指定了传递给调用操作符的重载实例的形式参数。如果比较这个定义和前面的 less_than()定义，我们注意到除了用 operator()代替 less_than 之外，这两个定义完全一样。

函数对象的定义方式与普通类对象一样，虽然在这种情况下，我们不必定义构造函数（没有数据成员需要被初始化）：

```
LessThan lt;
```

为了调用被重载的调用操作符，我们只是简单地把调用操作符应用在我们的类对象上，并向它提供必要的参数。例如：

```
string st1( "shakespeare" );
string st2( "marlowe" );

// 调用 lt.operator()( st1, st2 );
bool is_shakespeare_less = lt( st1, st2 );
```

下面是 process_vocab()的重新实现，这次我们向 stable_sort()传递了一个 LessThan 函数对象：

```
void process_vocab( vector< textwords >*pvec )
{
    // ...

    stable_sort( texts.be gin(), texts.end(), LessThan() );
```

```
    // ...
}
```

在 stable_sort()中，重载的调用操作符已经被内联展开。（stable_sort()能够接受的第三个实参可以是函数 less_than()的指针，也可以是类 LessThan 的对象，因为该实参是一个模板机制的类型参数。我们将在 12.3 节更详细地了解函数对象。）

下面是 texts 的 stable_sort()的结果：

```
a i
as at by he in is it no
of on so to and but for has
her him its not now one red row
she sky the you asks bird blue coat
dark each emma fear grow hair held home
life like long mean more puts same says
star such tell that time what when wind
with your alice alive blows daddy falls fiery
lands leave looks quite rises shush shyly sight
still stone tells there thing trees watch almost
either flight houses night, ancient becomes bounded calling
distant flowing heaven, magical passion through unravel untamed
wanting darkened eternity beautiful darkening immensity journeying
alternately
immeasurable inexpressibly
```

我们的下一个任务是计数长度小于 6 个字符的单词的个数。我们可以通过 count_if()泛型算法和第二个函数对象 GreatThan 来实现。GreatThan 是一个更加复杂的函数对象，因为我们要把它泛化，以便允许用户提供一个用于比较操作的显式长度值，所以在缺省情况下。用长度 6 初始化：

```
#include <iostream>

class GreaterThan {
public:
    GreaterThan( int sz = 6 ) : _size( sz ){}
    int size() { return _size; }
    bool operator()( const string & s1 )
        { return s1.size() > _size; }
private:
    int _size;
};
```

下面是它的用法：

```
void process_vocab( vector< textwords >*pvec )
{
    // ...
    // 计数长度大于 6 的字符串个数
    int cnt = count_if( texts.begin(), texts.end(),
        GreaterThan() );

    cout << "Number of words greater than length six are "
```

```
            << cnt << endl;
        // ...
    }
```

下面是这部分程序的输出：

```
Number of words greater than length six are 22
```

remove()的行为和 unique()相同，它并不是真正地改变容器的长度，而是把元素分成保留的（把它们按顺序拷贝到容器的前面）和要删除的（它们留在后面），它返回一个指向要被删除的第一个元素的 iterator。下面给出怎样用它来删除不希望留在 vector 中的常见词的集合：

```
void process_vocab( vector< textwords >*pvec )
{
    // ...

    static string rw[] = { "and", "if", "or", "but", "the" };
    vector< string > remove_words( rw, rw+5 );
    vector<string>::iterator it2 = remove_words.begin();

    for ( ; it2 != remove_words.end(); ++it2 ) {
        // 只是显示其他格式的 count()
        int cnt = count( texts.begin(), texts.end(), *it2 );

        cout << cnt << " instances removed: "
             << (*it2) << endl;

        texts.erase(
            remove(texts.begin(),texts.end(),*it2 ),
                texts.end()
        );
    }

    // ...
}
```

下面是 texts 的 remove()结果：

```
1 instances removed: and
0 instances removed: if
0 instances removed: or
1 instances removed: but
1 instances removed: the
```

最后我们想显示 vector 的内容。一种方式是迭代元素，按顺序一个个地显示，因为这种做法没有使用泛型算法，所以在这一节中不合适。我们更希望说明 for_each()泛型算法的用法，用它来输出 vector 的元素。for_each()把函数指针或函数对象应用在由一对 iterator 标记的容器的每个元素上。在我们的例子中，函数对象 printElem 把元素输出到标准输出上：

```
class PrintElem {
public:
    PrintElem( int lineLen = 8 )
        : _line_length( lineLen ), _cnt( 0 )
        {  }
    void operator()( const string &elem )
```

```
        {
            ++_cnt;
            if ( _cnt % _line_length == 0 )
                { cout << '\n'; }

            cout << elem << " ";
        }

    private:
        int _line_length;
        int _cnt;
    };

    void process_vocab( vector< textwords >*pvec )
    {
        // ...
        for_each( texts.begin(), texts.end(), PrintElem() );
    }
```

就是这样。我们完成了程序，几乎没做什么，只是把一串泛型算法调用连接在一起。为方便起见，我们在下面列出了完整的程序，并加上一个 main()函数驱动它（它提前使用了一个将在 12.4 节讨论的特殊的 iterator 类型）。我们列出了真正可执行的代码，它不完全是标准 C++。尤其是 count()和 count_if()算法提供的实现代表了一个旧版本，它不返回结果，而是要求传递一个额外的、用来存放结果值的实参。另外，iostream 库也反映了一个在标准 C++之前的实现，因为它要求使用 iostream.h 头文件。

```
#include <vector>
#include <string>
#include <algorithm>
#include <iterator>

// 标准 C++ 之前的 <iostream> 语法
#include <iostream.h>

class GreaterThan {
public:
    GreaterThan( int sz = 6 ) : _size( sz ){}
    int size() { return _size; }
    bool operator()( const string &s1 )
        { return s1.size() > _size; }

private:
    int _size;
};

class PrintElem {
public:
    PrintElem( int lineLen = 8 )
        : _line_length( lineLen ), _cnt( 0 )
```

```
        {}
    void operator()( const string &elem )
    {
        ++_cnt;

        if ( _cnt % _line_length == 0 )
            { cout << '\n'; }

        cout << elem << " ";
    }

private:
    int _line_length;
    int _cnt;
};

class LessThan {
public:
    bool operator()( const string & s1,
        const string & s2 )

    { return s1.size() < s2.size(); }
};

typedef vector<string, allocator> textwords;

void process_vocab( vector<textwords, allocator>*pvec )
{
    if ( ! pvec ) {
        // 给出警告消息
        return;
    }
    vector< string, allocator > texts;
    vector<textwords, allocator>::iterator iter;

    for ( iter = pvec->begin(); iter != pvec->end(); ++iter )
        copy( (*iter).begin(), (*iter).end(),
            back_inserter( texts ));

    // 排序 texts 的元素
    sort( texts.begin(), texts.end() );

    // ok, 我们来看一看我们有什么
    for_each( texts.begin(), texts.end(), PrintElem() );
    cout << "\n\n";   // 只是分隔显示输出

    // 删除重复元素
    vector<string, allocator>::iterator it;
    it = unique( texts.begin(), texts.end() );
```

```
    texts.erase( it, texts.end() );

    // ok, 让我们来看一看现在我们有什么了
    for_each( texts.begin(), texts.end(), PrintElem() );
    cout << "\n\n";

    // 根据缺省的长度 6 排序元素
    // stable_sort() 保留相等元素的顺序
    stable_sort( texts.begin(), texts.end(), LessThan() );
    for_each( texts.begin(), texts.end(), PrintElem() );
    cout << "\n\n";

    // 计数长度大于 6 的字符串的个数
    int cnt = 0;

    // count 的过时格式—标准 C++ 已经改变了它
    count_if( texts.begin(), texts.end(), GreaterThan(), cnt );
    cout << "Number of words greater than length six are "
         << cnt << endl;

    static string rw[] = { "and", "if", "or", "but", "the" };
    vector<string,allocator> remove_words( rw, rw+5 );
    vector<string, allocator>::iterator it2 =
         remove_wo rds.begin();

    for ( ; it2 != remove_words.end(); ++it2 )
    {
         int cnt = 0;

         // count 的过时格式—标准 C++ 已经改变了它
         count( texts.begin(), texts.end(), *it2, cnt );
         cout << cnt << " instances removed: "
              << (*it2) << endl;
         texts.erase(
              remove(texts.begin(),texts.end(),*it2),
              texts.end()
         );
    }

    cout << "\n\n";
    for_each( texts.begin(), texts.end(), PrintElem() );
}

// difference_type 类型能够包含一个容器的两个 iterator 的减法结果
// —在这种情况下，是 string vector 的 ...
// 通常，被缺省处理
typedef vector<string,allocator>::difference_type diff_type;

// 标准 C++ 之前的头文件语法
#include <fstream.h>
```

```
main()
{
    vector<textwords, allocator> sample;
    vector<string,allocator> t1, t2;
    string t1fn, t2fn;

    // 要求用户输入文件
    // 实际中的程序应该做错误检查
    cout << "text file #1: "; cin >> t1fn;
    cout << "text file #2: "; cin >> t2fn;

    // 打开文件
    ifstream infile1( t1fn.c_str());
    ifstream infile2( t2fn.c_str());

    // iterator 的特殊形式
    // 通常，diff_type 被缺省提供
    istream_iterator< string, diff_type > input_set1( infile1 ),
                                                      eos;
    istream_iterator< string, diff_type > input_set2( infile2 );

    // iterator 的特殊形式
    copy( input_set1, eos, back_inserter( t1 ));
    copy( input_set2, eos, back_inserter( t2 ));

    sample.push_back( t1 ); sample.push_back( t2 );
    process_vocab( &sample );
}
```

练习 12.2

单词的长度不是衡量一个文本的复杂度的惟一或可能的最好标准。另外一种可能的测试标准是句子的长度。请写一个程序，它读入一个文本文件，或者从标准输入读入，并为每个句子生成一个字符串 vector，然后把每个 vector 传递给 count()。按复杂性显示句子。一种有趣的做法是，把每个句子都作为长串存储在第一个字符串 vector 中，然后把该 vector 传递给 sort()，并提供一个函数对象，该函数对象提供了基于较短字符串的小于语义。（要了解某个特定的泛型算法的更详细描述或者其用法的进一步例子，请参见附录，它以字符顺序列出了这些算法。）

练习 12.3

对一段文字更可靠的难度测试是句子的结构复杂性。把每个逗号记 1 点、每个冒号或分号记 2 点、每个破折号记 3 点。修改练习 12.2 中的程序，计算每个句子的复杂性。用 count_if() 来确定句子 vector 中每个标点的出现次数，并按照复杂性顺序显示所有的句子。

12.3 函数对象

我们的 min()函数是模板机制的功能强大性与局限性的一个很好例子。

```
template <typename Type>
    const Type&
        min( const Type *p, int size )
{
    int minIndex = 0;
    for ( int ix = 1; ix < size; ++ix )
        if ( p[ ix ] < p[ minIndex ] )
            minIndex = ix;
    return p[ minIndex ];
}
```

功能强大性来自于“定义一个 min()的单个实例，它就可以被实例化为无限种类型”的能力。局限性在于，虽然 min()可以被实例化为无限种类型，但是它并不是对所有类型都完全适用。

局限性的焦点在于小于操作符的使用上。在一种情况下；底层类型可能不支持小于操作符。例如，一个 Image 类可能不提供小于操作符的实现，虽然我们现在还不知道，但是以后可能希望发现 Image 对象数组的最小帧数。但是用 Image 类数组来实例化 min()会导致编译时刻错误：

```
error: invalid types applied to the < operator: Image < Image
```

在第二种情况下，虽然存在小于操作符，但是提供的语义并不合适。例如，如果我们希望找到最小的字符串，但是希望只考虑字母，并且不对大小写敏感，则虽然小于操作符被支持，但支持的却是错误的语义。

传统的方案是参数化比较操作符，在这种情况下声明一个函数指针，该函数有两个参数并返回一个 bool 型的值：

```
template < typename Type,
    bool (*Comp)(const Type&, const Type&)>
    const Type&
        min( const Type *p, int size, Comp comp )
{
    int minIndex = 0;
    for ( int ix = 1; ix < size; ++ix )
        if ( Comp( p[ ix ], p[ minIndex ] ))
            minIndex = ix;
    return p[ minIndex ];
}
```

这种方案，与我们使用内置的小于操作符的第一个实现一起提供了对任何类型的一般性支持，同时也包括我们的 Image 类，只要我们实现两个 Image 小于比较的语义。函数指针的主要性能缺点是，它的间接引用使其不能被内联。

对函数指针的替代策略是函数对象（我们在前面的例子中看到了大量的例子）。函数对象是一个类，它重载了函数调用操作符［operator()］。该操作符封装了通常应该被实现为一

个函数的动作。典型情况下，函数对象被作为实参传递给泛型算法，当然我们也可以定义独立的函数对象。例如，如果把类 AddImage 定义为一个函数对象，它取两个图像，把它们合成在一起（即把两个加在一起），然后返回一个图像，则我们可以这样定义：

```
AddImages AI;
```

为了使函数对象执行其操作，我们应用调用操作符，提供必要的 Image 类操作数。例如：

```
Image im1("foreground.tiff"), im2("background.tiff");
// ...

// 调用 Image AddImages::operator()(const Image&,const Image&);
Image new_image = AI( im1, im2 );
```

函数对象与函数指针相比较，有两个方面的优点：首先，如果被重载的调用操作符是 inline 函数，则编译器能够执行内联编译，提供可能的性能好处：其次，函数对象可以拥有任意数目的额外数据，用这些数据可以缓冲结果，也可以缓冲有助于当前操作的数据。

下面是修改后的 min()实现（注意函数指针也可以用这个声明来传递，但是没有任何原型检查）：

```
template < typename Type,
     typename Comp >
const Type&
     min( const Type *p, int size, Comp comp )
{
     int minIndex = 0;

     for ( int ix = 1; ix < size; ++ix )
          if ( Comp( p[ ix ], p[ minIndex ] ))
               minIndex = ix;

     return p[ minIndex ];
}
```

泛型算法一般支持两种形式来应用操作：使用内置（或可能是被重载的）操作符，和使用函数指针或函数对象执行操作。

函数对象从哪里来？一般来说，有三种来源：

- 标准库预定义的一组算术、关系和逻辑函数对象。
- 一组预定义的函数适配器，允许我们对预定义的函数对象（甚至于任何函数对象）进行特殊化或者扩展。
- 我们可以定义自己的函数对象，将其传递给泛型算法，或将它们传给函数适配器。

本节，我们将按顺序了解这三种函数对象。

12.3.1 预定义函数对象

预定义函数对象被分成算术、关系和逻辑操作。每个对象都是一个类模板，其中操作数的类型被参数化。为了使用它们，我们必须包含下列头文件：

```
#include <functional>
```

例如，支持加法的函数对象是一个名为 plus 的类模板。为定义一个可以把两个整数相加的实例，我们可以这样写：

```
#include <functional>
plus< int > intAdd;

plus< int > int Add;
```

为了调用加法操作，我们把重载的调用操作符应用在 intAdd 上　就像在上节中我们对 AddImage 类所做的那样：

```
int ival1 = 10, ival2 = 20;
// 等价于 int sum = ival1 + ival2;
int sum = intAdd( ival1, ival2 );
```

类模板 plus 的实现调用了与其参数 int 类型相关联的加法操作符。这个类和其他预定义的类的函数对象的主要用法是作为泛型算法的实参，通常被用来改变缺省的操作。例如，缺省情况下，sort()用底层元素类型的小于操作符以升序排列容器的元素。为了以降序排列容器，我们传递预定义的类模板 greater，它调用底层元素类型的大于操作符：

```
vector< string > svec;
// ...
sort( svec.begin(), svec.end(), greater<string>() );
```

预定义的函数对象被分成算术、关系和逻辑三大类别，分别被列在下面的小节中。每个类对象都可以作为有名对象，也可以作为无名对象传递给一个函数，在下面的小节中分别进行了说明。我们使用下面的对象定义，包括一个简单类的定义（关于操作符重载将在第 15 章详细讨论）：

```
class Int {
public:
     Int( int ival = 0 ) : _val( ival ){}

     int operator-()           { return -_val; }
     int operator%(int ival)   { return _val % ival; }

     bool operator<(int ival)  { return _val < ival; }
     bool operator!()          { return _val == 0; }

private:
     int _val;
};

vector< string > svec;
string sval1, sval2, sres;
complex cval1, cval2, cres;
int ival1, ival2, ires;
Int Ival1, Ival2, Ires;
double dval1, dval2, dres;
```

另外，我们还定义了下列两个函数模板，我们向其传递各种没有名字的函数对象：

```
template <class FuncObject, class Type>
     Type UnaryFunc( FuncObject fob, const Type &val )
          { return fob( val ); }

template <class FuncObject, class Type>
     Type BinaryFunc( FuncObject fob,
```

```
                const Type &val1, const Type &val2 )
    { return fob( val1, val2 ); }
```

12.3.2 算术函数对象

预定义的算术函数对象支持加、减、乘、除、求余和取反。调用的操作符是与 Type 相关联的实例。对一个 class 类型，如果它提供了该操作符的重载实例，则调用该实例。

- 加法：plus<Types>

```
plus<string> stringAdd;

// 调用 string::operator+()
sres = stringAdd( sval1, sval2 );
dres = BinaryFunc( plus<double>(), dval1, dval2 );
```

- 减法：minus<Type>

```
minus<int> intSub;
ires = intSub( ival1, ival2 );
dres = BinaryFunc( minus<double>(), dval1, dval2 );
```

- 乘法：multiplies<Type>

```
multiplies<complex> complexMultiplies;
cres = complexMultiplies( cval1, cval2 );
dres = BinaryFunc( multiplies<double>(), dval1, dval2 );
```

- 除法：divides<Type>

```
divides<int> intDivides;
ires = intDivides( ival1, ival2 );
dres = BinaryFunc( divides<double>(), dval1, dval2 );
```

- 求余：modulus<Type>

```
modulus<int> IntModulus;
Ires = IntModulus( Ival1, Ival2 );
ires = BinaryFunc( modulus<int>(), ival2, ival1 );
```

- 取反：negate<Type>

```
negate<int> intNegate;
ires = intNegate( ires );
Ires = UnaryFunc( negate<int>(), Ival1 );
```

12.3.3 关系函数对象

预定义的关系函数对象支持等于、不等于、大于、大于等于、小于和小于等于。

- 等于：equal_to<Type>

```
equal_to<string> stringEqual;
sres = stringEqual( sval1, sval2 );
ires = count_if( svec.begin(), svec.end(),
     equal_to<string>(), sval1 );
```

- 不等于：not_equal_to<Type>

```
not_equal_to<complex> complexNotEqual;
cres = complexNotEqual( cval1, cval2 );
ires = count_if( svec.begin(), svec.end(),
     not_equal_to<string>(), sval1 );
```

- 大于：greater<Type>

```
greater<int> intGreater;
ires = intGreater( ival1, ival2 );
ires = count_if( svec.begin(), svec.end(),
     greater<string>(), sval1 );
```

- 大于等于：greater_equal<Type>

```
greater_equal<double> doubleGreaterEqual;
dres = doubleGreaterEqual( dval1, dval2 );
ires = count_if( svec.begin(), svec.end(),
     greater_equal<string>(), sval1 );
```

- 小于：less<Type>

```
less<int> IntLess;
Ires = IntLess( Ival1, Ival2 );
ires = count_if( svec.begin(), svec.end(),
     less<string>(), sval1 );
```

- 小于等于：less_equal<Type>

```
less_equal<int> intLessEqual;
ires = intLessEqual( ival1, ival2 );
ires = count_if( svec.begin(), svec.end(),
     less_equal<string>(), sval1 );
```

12.3.4 逻辑函数对象

预定义的逻辑函数对象支持逻辑与（两个操作数都为 true 时结果值为 true——应用与 Type 相关联的&&）、逻辑或（两个操作数中有一个为 true 返回 true——应用与 Type 相关联的||）和逻辑非（操作数为 false 则返回 true——应用与 Type 相关联的!操作符）。

- 逻辑与：logical_and<Type>

```
logical_and<int> intAnd;
ires = intAnd( ival1, ival2 );
dres = BinaryFunc( logical_and<double>(), dval1, dval2 );
```

- 逻辑或：logical_or<Type>

```
logical_or<int> intSub;
ires = intSub( ival1, ival2 );
dres = BinaryFunc( logical_or<double>(), dval1, dval2 );
```

- 逻辑非： logical_not<Type>

```
logical_not<int> IntNot;
Ires = IntNot( Ival1, Ival2 );
dres = UnaryFunc( logical_not<double>(), dval1 );
```

12.3.5 函数对象的函数适配器

标准库还提供了一组函数适配器，用来特殊化或者扩展一元和二元函数对象。适配器是一种特殊的类，它被分成下面两类：

1．绑定器（binder）：binder 通过把二元函数对象的一个实参绑定到一个特殊的值上，将其转换成一元函数对象。例如，为了计数一个容器中小于或等于 10 的元素的个数，我们可能会向 count_if()传递一个 less_equal 函数对象，以及一个被绑定为 10 的实参。在下一节中我们将了解怎样实现这种方法。

2．取反器（negator）：negator 是一个将函数对象的值翻转的函数适配器。例如，为了计数一个容器中所有大于 10 的元素的个数，我们可以向 count_if()传递 less_equal 函数对象的 negator，该函数对象有一个实参被绑定为 10。当然，在这种情况下，直接传递 greater 对象的 binder 并把一个实参绑定为 10 更为简洁明了。

C++标准库提供了两种预定义的 binder 适配器：bind1st 和 bind2nd。正如你所预料的 bind1st 把值绑定到二元函数对象的第一个实参上，bind2nd 把值绑定在第二个实参上。例如，为了计数容器中所有小于或等于 10 的元素的个数，我们可以这样向 count_if()传递：

```
count_if( vec.begin(), vec.end(),
     bind2nd( less_equal<int>(), 10 ));
```

标准库提供了两个预定义的 negator 适配器：not1 和 not2。同样正如你所料想的，not1 翻转一元预定义函数对象的真值，而 not2 翻转二元谓词函数的真值。为了取反 less_equal 函数对象的绑定，我们可以这样写：

```
count_if( vec.begin(), vec.end(),
     not1( bind2nd( less_equal<int>(), 10 )));
```

我们会在附录中使用泛型算法的例子中看到更多的使用 binder 和 negator 的例子。

12.3.6 实现函数对象

我们已经定义了大量函数对象来支持 12.2 节中的程序实现。在本节中，我们将看看定义类函数对象的步骤和各种变化。（在第 13 章将详细讲解类的一般性定义，第 15 章将讨论操作符重载。）

函数对象类定义的最简单形式包含一个被重载的函数调用操作符。例如，下面是一个二元函数对象，它判定一个值是否小于等于 10：

```
// 函数对象类的简单形式
class less_equal_ten {
public:
     bool operator() ( int val )
          { return val <= 10; }
};
```

使用这个对象的方式与使用预定义函数对象的方式相同。例如，下面是修改后的 count_if()调用，它使用了我们的函数对象：

```
count_if( vec.begin(), vec.end(), less_equal_ten() );
```

毫无疑问，这个类是相当局限的。我们可以应用一个 negator 来计数容器中大于 10 的元素的个数：

```
count_if( vec.begin(), vec.end(),
     not1(less_equal_ten()) );
```

我们也可以通过允许用户提供一个与每个元素比较的值来扩展我们的实现。一种做法是引入一个数据成员来存储被比较的值，以及一个构造函数把这个成员初始化为用户指定的值：

```
class less_equal_value {
public:
     less_equal_value( int val ) : _val( val ) {}
     bool operator() ( int val ) { return val <= _val; }

private:
     int _val;
};
```

我们现在用这个对象指定一个任意的整数值。例如，下面的调用计数小于等于 25 的元素的个数。

```
count_if( vec.begin(), vec.end(), less_equal_value( 25 ));
```

另外一种类的实现方式，不使用构造函数，它根据被比较的值对类参数化。例如：

```
template < int _val >

class less_equal_value {
public:
     bool operator() ( int val ) { return val <= _val; }
};
```

下面给出了怎样调用这个类来计数小于等于 25 的元素的个数。

```
count_if( vec.begin(), vec.end(), less_equal_value<25>() );
```

我们将在附录中看到更多自定义函数对象的例子，它们出现在每个使用泛型算法的例子中。

练习 12.4

请使用预定义的函数对象和函数适配器，创建一个能做下列事情的函数对象：

(a) 找到所有大于 1024 的值
(b) 找到所有不等于"pooh"的字符串
(c) 所有的值乘以 2

练习 12.5

请定义一个函数对象，使它能对三个对象进行运算。并返回中间的值。再定义一个函数让它做同样的操作。给出直接使用每个对象以及将它们传递给函数的例子，比较并对比每个的行为。

12.4 回顾 iterator

下列函数模板不能编译。你知道这是为什么吗？

```
// 无法通过编译
template < typename type >
int
count( const vector< type > &vec, type value )
{
     int count = 0;
     vector< type >::iterator iter = vec.begin();
     while ( iter != vec.end() ) {
          if ( *iter == value )
               ++count; ++iter; }
     return count;
}
```

问题在于 vec 是一个 const 引用，但是我们试图把一个非 const 的 iterator 绑定在它上面。如果允许这样做，那就没有什么能阻止我们在后面通过 iterator 修改这个 vector 的值了。为了防止这种情况的出现，C++语言要求绑定在 const vector 上的 iterator 也必须是 const iterator。我们这样做：

```
// ok: 这次可以通过编译了
vector< type >::const_iterator iter = vec.begin();
```

const 容器只能被绑定在 const iterator 上，这样的要求与 const 指针只能指向 const 数组的行为一样。在两种情况下，C++语言都努力保证 const 容器的内容不会被改变。

begin()和 end()两种操作都被重载，根据容器的常量性返回一个 const 或非 const iterator。例如，给出下列一对声明：

```
vector< int > vec0;
const vector< int > vec1;
```

在 vec0 上的 begin()和 end()调用返回一个非 const 的 iterator，而 vec1 上的调用返同一个 const 的 iterator。例如：

```
vector< int >::iterator iter0 = vec0.begin();
vector< int >::const_iterator iter1 = vec1.begin();
```

当然，给一个 const iterator 赋值一个非 const iterator 总是可以的，例如：

```
// ok: 把一个非 const iterator 初始化为一个 const
vector< int >::const_iterator iter2 = vec0.begin();
```

12.4.1 插入 iterator

下面是另外一个程序段，它有一个严重但是很微妙的问题。你看到问题了吗？

```
int ia[] = { 0, 1, 1, 2, 3, 5, 5, 8 };
vector< int > ivec( ia, ia+8 ), vres;

// ...
```

```
// 导致未定义的运行时刻行为
unique_copy( ivec.begin(), ivec.end(), vres.begin() );
```

这里的问题是，vres 中没有已被分配的空间来保存从 ivec 向其拷贝的 8 个整型值。unique_copy()算法用赋位操作拷贝每个元素值，但是赋值会失败，因为在 vres 中没有可用空间。

一种策略是提供 unique_copy()算法的两个版本：一个赋位元素，而另一个插入元素。插入实例会根据不同的需要支持在容器的前面、后面和任意位置插入元素的实例。

另外一种策略，也是一种被标准库采纳的策略是，定义一组（三个）插入 iterator 的适配器函数，它们返回特定的插入 iterator：

- back_inserter()：它使用容器的 push_back()插入操作代替赋值操作符。back_inserter()的实参是容器自己。例如，我们可以这样修正 ynique_copy()调用：

```
// ok: unique_copy() 现在用 vres.push_back() 插入
unique_copy( ivec.begin(), ivec.end(),
             back_inserter( vres );
```

- front_inserter()：它使用容器的 push_front()插入操作代替赋值操作符。front_inserter()的实参也是容器自己。但是，注意 vector 类不支持 push_front()操作，所以，试图在 vector 上使用它是错误的：

```
// 喔！错误
// vector 不支持 push_front() 操作
// 使用 deque 或 list
unique_copy( ivec.begin(), ivec.end(),
                 front_inserter( vres );
```

- inserter()：它调用容器的 insert()插入操作代替赋值操作符。inserter()要求两个实参：容器本身以及它的一个 iterator 指示起始插入的位置。例如：

```
unique_copy( ivec.begin(), ivec.end(),
                 inserter( vres, vres.begin() );
```

标记起始插入位置的 iterator 并不保持不变，而是随着每个被插入的元素而递增，这样每个元素就能顺序被插入。就好像我们已经写：

```
vector< int >::iterator iter = vres.begin(),
     iter2 = ivec.begin();
for ( ; iter2 != ivec.end(); ++iter, ++iter2 )
     vres.insert( iter, *iter2 );
```

12.4.2 反向 iterator

begin()和 end()操作分别返回指向容器的首元素和容器的末元素下一位置的 iterator。返回一个反向 iterator 也是有可能的：一个从末元素到首元素遍历容器的 iterator。对所有容器类型都能支持这种能力的操作是 rbegin()和 rend()。与前向 iterator 一样，它有 const 和非 const 两种实例。

```
vector< int > vec0;
const vector< int > vec1;
```

```
vector< int >::reverse_iterator r_iter0 = vec0.rbe gin();
vector< int >::const_reverse_iterator r_iter1 = vec1.rbegin();
```

反向 iterator 的遍历方式同前向 iterator 一样。不同的是 next（或 previous）操作的实现。对于前向 iterator，++操作访问容器中的下一个元素；对于反向 iterator，它访问的是前面的元素。例如，反向遍历一个 vector，我们可以这样写

```
// vector 中从后到前的反向 iterator
vector< type >::reverse_iterator r_iter;
for ( r_iter = vec0.rbegin();    // 将 r_iter 绑定到末元素
          r_iter != vec0.rend(); // 不等于首元素下一元素
          r_iter++ )             // 递减! iterator 一个元素
          { /* ... */ }
```

虽然这似乎混淆了递增和递减两个操作符的意义，但是它让程序员透明地为一个算法传递一对反向 iterator。例如，为了降序排列 vector，我们只要简单地向 sort()传递一对反向 iterator，如下

```
// 以升序排列 vector
sort( vec0.begin(), vec0.end() );

// 以降序排列 vector
sort( vec0.rbegin(), vec0.rend() );
```

12.4.3 iostream iterator

标准库为输入和输出 iostream 的 iterator 提供了支持，它们可以与标准库容器类型和泛型算法结合起来工作。istream_iterator 类支持在一个 istream、或其派生类（如 ifstream 输入文件流）上的 iterator 操作。类似地，ostream_iterator 支持在一个 ostream 或其派生类（如 ofstream 输出文件流）上的 iterator 操作。为了使用这两种 iterator，我们必须包含 iterator 头文件：

```
#include <iterator>
```

例如，在下列程序中，我们用一个 istream_iterator 从标准输入读入一个整数集到一个 vector 中，然后再用一个 ostream_iterator 作为 unique_copy()泛型算法的目标。

```
#include <iostream>
#include <iterator>
#include <algorithm>
#include <vector>
#include <functional>

/*
* 输入:
* 23 109 45 89 6 34 12 90 34 23 56 23 8 89 23
*
* 输出:
* 109 90 89 56 45 34 23 12 8 6
*/

int main()
```

```
{
    istream_iterator< int > input( cin );
    istream_iterator< int > end_of_stream;

    vector<int> vec;
    copy ( input, end_of_stream, inserter( vec, vec.begin() ));

    sort( vec.begin(), vec.end(), greater<int>() );

    ostream_iterator< int > output( cout, " " );
    unique_copy( vec.begin(), vec.end(), output );
}
```

12.4.4 istream_iterator

用下列一般形式声明一个 istream_iterator[22]：

```
istream_iterator<Type> identifier( istream& );
```

这里的 Type 表示任意一个已定义了输入操作符的内置或用户定义的类型。构造函数的实参可以是一个 istream 类对象，如 cin，或任意公有派生的 istream 子类型，如 ifstream。例如：

```
#include <iterator>
#include <fstream>
#include <string>
#include <complex>

// 从标准输入读入一个 complex 对象的序列
istream_iterator< complex > is_complex( cin );

// 从命名的文件中读入一个字符串序列
ifstream infile( "C++Primer" );

istream_iterator< string > is_string( infile );
```

应用在 istream_iterator 对象上的每个递增操作符，都用 operator>>()读入输入流的下一个元素。为了通过 istream_iterator 和一个泛型算法读取输入流，我们需要提供一对 iterator 指示文件内部的开始和结束位置。用一个 istream 对象初始化的 istream_iterator（比如 is_string）提供开始位置。为定义结束位置，我们使用专门的 istream_iterator 缺省构造函数：

```
// 构造一个 end-of-stream iterator 用作 iterator 对的结束标记 ...
istream_iterator< string > end_of_stream;
```

---

[22] 如果你的编译器还不支持模板参数的缺省值，那么需要给 istream_iterator 构造函数提供一个显式的第二个实参：用于存放元素的容器的 difference_type。difference_type 是能够保存“一个容器的两个 iterator 减法结果”的类型。例如，在 12.2 节中，在不支持模板参数缺省值的编译器下的程序的表示中，我们这样写：

```
typedef vector<string,allocator>::difference_type diff_type;
istream_iterator< string, diff_type > input_set1( infile1 ), eos;
istream_iterator< string, diff_type > input_set2( infile2 );
```

在一个完全兼容标准 C++的编译器下面，我们可以简化为下面的代码：

```
istream_iterator< string, input_set1( infile1 ), eos;
istream_iterator< string, input_set2( infile2 );
```

```
vector<string> text;

// Ok: 提供 iterator 对
copy( is_string, end_of_stream,
      inserter( text, text.begin() ));
```

12.4.5 ostream_iterator

用下列两种形式之一可以声明 ostream_iterator：

```
ostream_iterator<Type> identifier( ostream& )
ostream_iterator<Type> identifier( ostream&, char* delimiter )
```

这里的 Type 表示任意一个已定义了输出操作符（operator>>）的内置或用户定义的类型。delimiter 表示一个 C 风格字符串，它被输出到文件的每个元素后面。因为它是一个 C 风格字符串，所以它必须有空终止符，否则行为将是未定义的（一能会在运行时刻出现）。ostream 实参可以是一个实际的 ostream 类对象，如 cout，或任意公有派生的 ostream 子类型，如 ofstream。例如：

```
#include <iterator>
#include <fstream>
#include <string>
#include <complex>

// 向标准输出写一个 complex 对象的序列
// 用空格分割每个元素
ostream_iterator< complex > os_complex( cout, " " );

// 向一个命名文件写一个字符串序列
// 每一行放一个
ofstream outfile( "dictionary" );
ostream_iterator< string > os_string( outfile, "\n" );
```

下面是一个简单的例子，它读取标准输入，并将其回显在标准输出上，使用无名的 stream iterator 对象和 copy()泛型算法。

```
#include <iterator>
#include <algorithm>
#include <iostream>

int main()
{
    copy( istream_iterator< int >( cin ),
          istream_iterator< int >(),
          ostream_iterator< int >( cout ));
}
```

最后，下面这个小程序再次使用 copy()算法和一个 ostream_iterator 打开一个用户指定的文件，并将其回显在标准输出上：

```
#include <string>
#include <algorithm>
#include <fstream>
#include <iterator>
```

```
int main()
{
    string file_name;
    cout << "please enter a file to open: ";
    cin >> file_name;
    if ( file_name.empty() || !cin ) {
        cerr << "unable to read file name \n"; return -1;
    }
    ifstream infile( file_name.c_str());
    if ( ! infile ) {
        cerr << "unable to open " << file_name << endl;
        return -2;
    }
    istream_iterator< string > ins( infile ), eos;
    ostream_iterator< string > outs( cout, " " );
    copy( ins, eos, outs );
}
```

12.4.6 五种 iterator

为支持泛型算法全集，根据它们提供的操作集，标准库定义了五种 iterator：InputIterator、OutputIterator、ForwardIterator。BldirectionalIterator 和 RandomAccessIterator。下面是对它们各自特性的简要讨论：

1．InputIterator 可以被用来读取容器中的元素，但是不保证支持向容器的写入操作。InputIterator 必须提供下列最小支持（提供其他支持的 iterator 也可被用作 InputIterator，只要它们满足这个最小要求集）：两个 iterator 的相等和不相等测试、通过 operator（++）的前置和后置实例向前递增 iterator 指向下一个元素、通过解引用操作符 operator（*）读取一个元素。要求在这个层次上提供支持的泛型算法包括 find()、accumulate()和 equal()。任何一个算法如果要求 InputIterator，那么我们也可以向其传递第 3、4、5 项列出的 iterator 类别中的任一个。

2．OutputIterator 可以被认为是与 InputIterator 功能相反的 iterator，即它可以被用来向容器写入元素，但是不保证支持读取容器的内容。OutputIterator 一般被用作算法的第三个实参，标记出起始写入的位置。例如，copy()取 OutputIterator 作为第三个实参。任何一个算法如果要求 OutputIterator，那么我们也可以向其传递第 3、4、5 项列出的 iterator 类别中的任一个。

3．ForwardIterator 可以被用来以某一个遍历方向（是的，下一个类别支持双向遍历）向容器读或写。有些泛型算法至少要求 ForwardIterator，包括 adjacent_find()、swap_range()和 replace()。当然，任何要求 ForwardIterator 支持的算法都可以向其传递第 4 和 5 项定义的 iterator 类别。

4．BidirectionalIterator 从两个方向读或写一个容器，有些泛型算法至少要求 BidirectionalIterator，包括 inplace_merge()、next_permutation()和 reverse()。

5．RandomAccessIterator，除了支持 BidirectionalIterator 所有的功能之外，还提供了“在常数时间内访问容器的任意位置”的支持，要求 RandomAccessIterator 支持的泛型算法包括 binary_search()、sort_heap()和 nth_element()。

map、set 和 list 维护了双向 iterator。实际上，这意味着它们不能被用在要求 RandomAccessIterator 的泛型算法中，如 sort_heap()和 nth_element()。我们将在 12.6 节看到一个可用于 list 容器的替代操作。vector 和 deque 维护了随机访问的 iterator，因此可以被用于所有的泛型算法。

练习 12.6

说明下列代码错误的原因，指出哪些错误可以在编译期间被捕获到。

```
(a)  const vector<string> file_names( sa, sa+6 );
     vector<string>::iterator it = file_names.begin()+2;

(b)  const vector<int> ivec;
     fill( ivec.begin(), ivec.end(), ival );

(c)  sort( ivec.begin(), ivec.rend() );

(d)  list<int> ilist( ia, ia+6 );
     binary_search( ilist.begin(), ilist.end() );

(e)  sort( ivec1.begin(), ivec2.end() );
```

练习 12.7

请写一个程序，用 istream_iterator 从标准输入读入一个整数序列，用 ostream_iterator 把奇数写入一个文件，并且每个值以空格分开。再把偶数也用 ostream_iterator 写入第二个文件，每个值应该放在单独的一行中。

12.5 泛型算法

所有泛型算法（带有相当多异常，这些异常也构成了相应的规则）的前两个实参都是对 iterator，通常被称为 first 和 last，它们标记出要操作的容器或内置数组中的元素范围。元素范围概念（有时称为左闭合区间）通常写为：

```
// 读作：包含 first 以及到 但不包含 last 的所有元素
[ first, last )
```

表示范围从 first 开始，到 last 结束，但不包括 last。当如下条件时：

```
first == last
```

范围为空。

对于 iterator 对的一个要求是，从 first 开始通过反复应用递增操作符必须能够到达 last。但是，编译器自己不能保证这一点，若不能满足这个要求，将导致运行时刻未定义的行为——通常是程序的核心转储（core dump）。

每个算法的声明指示了它所要求支持的 iterator 的最小类别（关于 5 个 iterator 类别的简要讨论见 12.4 节）。例如，find()它实现对一个容器的一次只读遍历）最小要求 InputIterator。我们也可以向其传递 ForwardIterator、BidirectonalIterator 或 RandomAccessIterator。但是，向

其传递 OutputIterator 会导致错误。向其传递一个无效的 iterator 类别引起的错误，不保证会在编译时刻被捕获到，因为 iterator 类别不是实际的类型。它们是被传递给函数模板的类型参数。

有一些算法支持多个版本。一个用内置操作符，而第二个接受函数对象或函数指针，它们被用来作为该操作符的替换实现。例如，缺省情况下，unique()用容器底层元素的等于操作符来比较两个相邻的元素。但是，如果底层元素类型没有提供等于操作符，或者我们希望定义不同的元素相等语义，则可以提供一个给出期望语义的函数对象或函数指针。但是，其他算法被分成两个名字不同的实例。可指定条件的实例在所有情况下都以后缀 if 结束，如 find_if()。例如，有一个使用内置等于操作符的 replace()实例，和一个带函数对象或函数指针的 replace_if()。

对那些修改所操作的容器的算法，一般有两种版本：一种替换版本，它改变被应用的容器，一种版本返回一个带有这些变化的容器复本。例如，标准库中有 replace()和 replace_copy()两个算法。拷贝版本在名字中包含 copy。但是，并不是每个要对容器作变换的算法都会提供一个拷贝版本，例如 sort()算法就不提供相应的拷贝版本。在这种情况下，如果我们希望算法对一个拷贝进行操作，就需要自己生成并传递这个拷贝。

为使用这些泛型算法，必须包含相关的头文件：

```
#include <algorithm>
```

为了使用下列四个算术算法——adjacent_difference()、accumulate()、inner_product()和 partial_sum()，我们必须包含：

```
#include <numeric>
```

这些算法在下列几个类别中列出（对它们进行分类是为了便于表达，和标准库没有形式上的联系）。附录依次提供了按字母顺序对每个算法的讨论和说明。

12.5.1 查找算法

13 个查找算法为判断容器中是否存在一个值提供了各种策略。equal_range()、lower_bound()和 upper_bound()三个算法提供了二分查找的形式。它们指出了一个值应该被插入在容器中的哪个位置，同时保留容器的排列顺序。这 13 个算法是：

```
adjacent_find(), binary_search(), count(), count_if(), equal_range(),
find(), find_end(), find_first_of(), find_if(), lower_bound(),
upper_bound(), search(), search_n()
```

12.5.2 排序和通用整序算法

14 个排序（sorting）和通用整序（ordering）算法为容器中元素的排序提供了各种策略。分割（partition）算法把容器分成两组。第一组由满足某个条件的元素组成，第二组则由不满足条件的元素组成。例如，我们可以根据元素是奇是偶、或单词是否以大写字母开头分割一个容器。稳定（stable）算法维持了值相等或同等地满足某个条件的元素的相对关系。例如，给出序列：

```
{ "pshew", "honey", "Tigger", "Pooh" }
```

根据单词是否以大写字母开头的稳定分割将生成下面的序列（其中相等的类别中的原有相对顺序被保留下来）：

```
{ "Tigger", "Pooh", "pshew", "honey" }
```

算法的非稳定实例并不保证这一点。（注意排序算法不能被用在 list 或联合容器上，如 set 或 map。）

```
inplace_merge(), merge(), nth_element(), partial_sort(),
partial_sort_copy(), partition(), random_shuffle(), reverse(),
reverse_copy(), rotate(), rotate_copy(), sort(), stable_sort(),
stable_partition()
```

12.5.3 删除和替换算法

15 个删除和替换算法为替换或去掉一个或一组元素提供了各种策略。unique()去掉相邻的相等元素。iter_swap()交换由一对 iterator 指向的元素的值，但它不交换 iterator 本身。这 15 个算法是：

```
copy(), copy_backwards(), iter_swap(), remove(), remove_copy(),
remove_if(), remove_copy_if(), replace(), replace_copy(),
replace_if(), replace_copy_if(), swap(), swap_range(), unique(),
unique_copy()
```

12.5.4 排列组合算法

考虑由三个字符{a, b, c}组成的序列。这个序列有六种可能的排列：abc、acb、bac、bca、cab 和 cba。而且，这些排列根据 less_than 小于操作符做一个排序。即 abc 是第一排列。为什么？因为每个元素都小于它后面的元素。acb 是下一个排列，因为 a 是最小的元素，它被固定了。类似地，以 b 开头的排列要小于所有以 c 开头的排列。对于排列 bac 和 bca，bac 小于 bca，因为 ac 小于 ca。对于排列 bca，我们可以说它的上一个排列是 bac，下一个排列是 cab。abc 没有上一个排列，而 cba 没有下一个排列。

```
next_permutation(), prev_permutation()
```

12.5.5 算术算法

下列 4 个算法提供对于容器的算术操作。为了使用它们。必须包含头文件<numeric>。

```
accumulate(), partial_sum(), inner_product(), adjacent_difference()
```

12.5.6 生成和异变算法

6 个生成和异变算法用一组值填充一个新序列或替换现有的序列。

```
fill(), fill_n(),for_each(), generate(), generate_n(), transform().
```

12.5.7 关系算法

7 个关系算法为比较两个容器提供了各种策略（min()和 max()只是比较两个元素）。

lexicographical_compare()提供了一个字典排序操作（见附录中的讨论以及后面关于排列的讨论）。

```
equal(), includes(), lexicographical_compare(), max(), max_element(),
min(), min_element(), mismatch()
```

12.5.8 集合算法

4 个集合（set）算法提供了对于任何容器类型的通用集合操作。并（union）算法创建了一个包含两个容器中所有元素的有序序列。交（intersection）算法创建了一个包含“在两个容器中都出现的元素”的有序序列。差（difference）算法创建了“在一个容器中存在而在第二容器中不存在的元素”的有序序列。对称差（symmetric difference）算法创建了一个“在两个容器之一中存在、但不同时出现在两个容器中的元素”的有序序列。

```
set_union(), set_intersection(), set_difference(),
set_symmetric_difference()
```

12.5.9 堆算法

堆是以“数组来表示二叉树”的一种形式。标准库提供了最大堆（max-heap）表示，它里面每个节点的键值大于等于其子节点的键值。

```
make_heap(), pop_heap(), push_heap(), sort_heap()
```

12.6 何时不用泛型算法

关联容器，如 map 或 set，在内部维护元素的排序关系，以便允许快速查伐和获取。因此不允许在关联容器上应用重新排序的泛型算法，如 sort()或 partition()，如果要重新排序关联容器中的元素，我们必须先把它拷贝到顺序容器中，如 vector 或 list。

list 容器是一个双向链表：除了实际的数据，每个元素维持着两个分别指向下一个和上一个链表元素的链成员。list 的主要优势在于，我们可以有效地把一个元素或一段元素插入到 list 的任意位置上，或者从中删除。主要缺点是没有随机元素访问特性。例如，虽然我们可以写

```
vector<string>::iterator vec_iter = vec.begin() + 7;
```

用 vector 的第 8 个元素的地址初始化 vec_iter，但是，下面的代码：

```
// 错误：对 list 不支持 iterator 的算术运算
list<string>::iterator list_iter = slist.begin() + 7;
```

是非法的，因为 list 的元素是非连续存储的。为到达 list 的第 8 个元素，我们必须遍历中间的）元素。

因为 list 容器不支持随机访问，所以 merge()、remove()、reverse()、sort()和 unique()泛型算法最好不要用在 list 对象上，尽管这些算法都没有显式地要求一个 RandomAccessIterator。标准库为每个算法都提供了专门的 list 成员实例（比如专门为 list 的 splice()操作）：

- list::merge()：用第二个有序的 list 合并一个有序 list。
- list::remove()：删除等于某个值的元素。
- list::remove_if()：删除满足某个条件的元素。
- list::reverse()：将 list 中元素反向排列。
- list::sort()：排序 list 的元素。
- list::splice()：把一个 list 的元素移到另一个 list 中。
- list::unique()：删除某个元素的重复连续拷贝。

12.6.1 list::merge()

```
void list::merge( list rhs );
template <class Compare>
    void list::merge( list rhs, Compare comp );
```

根据底层元素类型的小于操作符或用户指定的比较操作，合并两个已排序的 list 的元素（注意调用 merge()时，rhs 的元素被移到 list 对象中，在该操作之后，rhs 是空的。）。例如：

```
int array1[ 10 ] = { 34, 0, 8, 3, 1, 13, 2, 5, 21, 1 };
int array2[ 5 ] = { 377, 89, 233, 55, 144 };

list< int > ilist1( array1, array1 + 10 );
list< int > ilist2( array2, array2+5 );

// merge 要求两个 list 已经排序
ilist1.sort(); ilist2.sort();
ilist1.merge( ilist2 );
```

在应用 merge()操作之后，ilist2 是空的，ilist1 含有升序的菲波那契序列的前 15 个元素。

12.6.2 list::remove()

```
void list::remove( const elemType &value );
```

remove()操作删除指定值的全部实例。例如：

```
ilist1.remove( 1 );
```

12.6.3 list::remove_if()

```
template < class Predicate >
    void list::remove_if( Predicate pred );
```

remove_if()操作删除所有满足指定条件为真的元素。例如：

```
class Even {
public:
    bool operator()( int elem ) { return ! ( elem % 2 ); }
};

ilist1.remove_if( Even() );
```

删除 12.6.1 中定义的 list 对象的所有偶数元素。

12.6.4 list::reverse()

```
void list::reverse();
```

reverse()操作反向排列 list 元素。

```
ilist1.reverse();
```

12.6.5 list::sort()

```
void list::sort();
template <class Compare>
    void list::sort( Compare comp );
```

缺省情况下，sort()操作根据底层元素类型的小于操作符以升序放置 list 的元素。也可以将一个替换的比较操作符指定为实参。例如：

```
list1.sort();
```

以升序排列序 list1，而

```
list1.sort( greater<int>() );
```

用大于操作符以降序排序 list1。

12.6.6 list::splice()

```
void list::splice( iterator pos, list rhs );
void list::splice( iterator pos, list rhs, iterator ix );
void list::splice( iterator pos, list rhs,
    iterator first, iterator last );
```

splice()把一个或一级元素从一个 list 移到另一个中去。它有三种形式：把一个 list 的全部元素搬移到另一个中去，把一个 list 中包含的一组元素搬移到另一个中去，以及把一个 list 中的单个元素搬移到另一个中去。每种形式都给出了一个指出插入一个或一组元素的位置的 iterator。例如，给出下列两个 list：

```
int array[ 10 ] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };

list< int > ilist1( array, array + 10 );
list< int > ilist2( array, array+2 ); // 包含 0, 1
```

下面使用 splice()把 ilist1 的第一个元素移到 ilist2 中。现在 ilist2 包含元素 0、1、0，而 ilist1 不再包含 0：

```
// ilist2.end() 指示要接合元素的位置
// 被接合的元素在该位置之前
// ilist1 指示从哪个 list 中移动元素
// ilist1.begin() 指示要被移动的元素

ilist2.splice( ilist2.end(), ilist1, ilist1.begin() )
```

在 splice()的下一个用法中，传递了两个 iterator，指示要移动元素的子范围：

```
list< int >::iterator first, last;
```

```
first = ilist1.find( 2 );
last = ilist1.find( 13 );
ilist2.splice( ilist2.begin(), ilist1, first, last );
```

在这种情况下，元素 2、3、5 和 8 被从 ilist1 移到 ilist2 的前部。现在 ilist1 含有五个元素 1、1、13、21 和 34。为了把剩下这些元素移到 ilist2 中，可以使用 splice()操作符的最后一种形式：

```
list< int >::iterator pos = ilist2.find( 5 );
ilist2.splice( pos, ilist1 );
```

现在，ilist1 是空的。剩下的五个元素被移到 ilist2 中，放在值为 5 的元素之前的位置。

12.6.7 list::unique()

```
void list::unique();
template <class BinaryPredicate>
    void list::unique( BinaryPredicate pred );
```

unique()操作去掉重复的连续拷贝。缺省情况下，它使用底层类型的等于操作符。例如，给出值{0,2,4,6,4,2,0}，应用 unique()，结果是一个完全没有变化的 7 个元素的 list，因为它没有连续重复的元素。如果先排列这个 list，产生{0, 0, 2, 2, 4, 4, 6} 。则应用 unique()的结果是四个惟一的值{0, 2, 4, 6}。

```
ilist.unique();
```

unique()的第二种形式接受一个比较操作符。例如：

```
class EvenPair {
public:
    bool operator()( int val1, int val2 )
        { return ! ( val2 % val1 ); }
};
ilist.unique( EvenPair() );
```

去掉第二个元素能被第一个元素整除的相邻元素。

对于一个 list 对象，这些成员操作应该比相应的泛型算法要被优先考虑。其他泛型算法，如 find()、transform()和 for_each()等等，在 list 对象上的执行效率相同（对于每一个泛型算法的详细讨论见附录）。

练习 12.8

用 list 而不是 vector 重新实现 12.2 节的程序。

第四篇

基于对象的程序设计

第四篇将集中讲述基于对象的程序设计——即，C++的类（class）设施的定义以及用法。我们可以用类来定义新的类型。并且操纵这些新的类型可以像内置类型一样容易。通过创建新的类型来描述问题域，C++使程序员能够编写出更易于理解的应用程序。类设施使得程序员能够将新类型的底层实现相关细节（只有新类型的实现者才关心这些），同该类型的接口和操作的定义（该类型的用户需要这些信息）分离了。随着这种分离，程序设计中各种乏味的琐碎工作也就越来越不用关心了。应用程序的基本类型时以被实现一次，并被多次重用。将数据和函数封装在一起的这种设施。可以支持新类型的实现，也大大简化了应用程序后续的维护以及演化过程。

第 13 章将集中讨论一般性的类机制：怎样定义一个类，**信息隐藏**（information hiding）的概念（即公有类接口和私有实现），怎样定义和操纵类的对象实例，以及关于类域、嵌套类和作为名字空间成员的类的讨论。第 14 章将详细讨论 C++通过名为**构造函数**（constructor）、**析构函数**（destructor）和**拷贝赋值操作符**（copy assignment operator）的特殊成员函数，为类对象的初始化、析构和赋值提供了特殊的支持。我们还将了解按成员初始化和拷贝的话题，说明怎样用一个类对象初始化或赋值该类型的另一个对象。

第 15 章将介绍类特有的操作符重载。操作符重载使我们能够用第 4 章描述的内置操作符来操作。class 类型的操作数。操作符重载使 class 类型对象的用法与内置类型对象的用法一样直观。第 15 章将首先给出操作符重载的一般概念和设计考虑，然后查看一些特殊的操作符，比如赋值、下标、调用以及类特有的 new 和 delete 操作符。有时候我们有必要把一个重载操作符声明为一个类的友元（friend），使其拥有特殊的访问权限。这一章还将解释有时候必须使用友元的原因。然后，这一章还给出另外一种特殊的类成员函数：转换函数（conversion functions），它允许程序员为 class 类型定义一组标准转换。当类对象被用作函数实参，或作为内置或重载操作符的操作数时，这些转换函数由编译器隐式地调用。在第 15 章结束时，展示了与类实参、类成员函数和重载操作符有关的函数重载解析规则。

类模板是第 16 章的主题。类模板是一个用于创建类的“处方”，其中有一个或多个类型或者值被参数化。例如，一个 vector 类可以将其包含的元素的类型参数化。一个 buffer 类可

以不仅仅参数化它所持有的元素的类型，而且还可以参数化 buffer 的长度。这一章还将讨论怎样定义一个类模板，以及怎样创建一个类模板的特定实例。在模板这一章将会再次讲到 C++ 对类的支持，因此，我们将再次讨论成员函数、友元声明和嵌套类型。这一章还将回顾第 10 章讨论的模板编译模式，说明它对类模板的影响。

13

类

使用 C++的类机制用户能够定义自己的数据类型。因此，类经常被称为用户定义的类型（user-defined type，UDT）。通过类，我们可以向一个已有的类型添加功能——比如第 2 章介绍的 IntArray，它比 int 类型的数组可以提供更多的功能。类也可以用来引入新的类型。如 Screen 类和 Account 类。在典型情况下，类也可被用来定义“不能与内置数据类型建立自然映射的抽象”。

在本章，我们将了解怎样定义类（class）类型以及怎样使用类对象。我们将介绍类定义怎样引入类的数据成员和类成员函数，数据成员定义类的内部表示，成员函数定义可以被应用在该类型的对象上的操作集。我们将给出，在类定义中；怎样使用信息隐藏（information hiding）来把类的内部表示和实现声明为私有的（private），而把在类对象上执行的操作声明为公有的（public）。私有内部表示被称为是封装的（encapsulated），而类的公有部分被称为类接口（class interface）。

然后，本章将查看一种特殊的类成员：静态成员。在这之后我们还将了解怎样用成员指针引用类数据成员或成员函数。我们还会介绍一种特殊的类类型，联合（union），它可以使不同类型的对象相互覆盖。本章将以类域和类域中名字解析的讨论作为结束。我们将在对于各种不同种类的类的讨论中说明这些话题，包括嵌套类、作为名字空间成员的类和局部类。

13.1 类定义

类定义包含两部分：**类头**（class head），由关键字 class 及其后面的类名构成。**类体**（class body），由一对花括号包围起来。类定义后面必须接一个分号或一列声明。例如：

```
class Screen { /* ... */ };
class Screen { /* ... */ } myScreen, yourScreen;
```

在类体中，对类的数据成员和成员函数进行声明，并指定这些类成员的访问级别。类体定义了**类成员表**（class member list）。

每个类定义引入一个不同的类（class）类型。即使两个类类型具有完全相同的成员表，它们仍是不同的类型。例如：

```
class First {
    int memi;
    double memd;
};
class Second {
    int memi;
    double memd;
};
class First obj1;
Second obj2 = obj1;    // 错误: obj1 和 obj2 类型不同
```

类体定义了一个域（scope），在类体中的类成员声明把这些成员名字引入到它们的类的域中。如果两个类有同名的成员，那么程序不会出错，并且这两个成员将指向不同的对象。我们将在 13.9 节更详细地介绍类域。

在引入类类型之后，我们可以以两种方式引用这种类类型：

1．指定关键字 class，后面紧跟类名。在前面例子中，obj1 的声明以这种方式引用类 First。

2．只指定类名。在前面例子中，obj2 的声明以这种方式引用 Second。

这两种引用类类型的方式是等价的。第一种方式是从 C 中借用的，在 C++的声明中用它引用类类型也是有效的。第二种方式是 C++引入的，它使类类型更容易被用在声明中。

13.1.1 数据成员

类数据成员的声明方式同变量声明相同，例如，Screen 类可以有下列数据成员：

```
#include <string>
class Screen {
    string _screen;   // string( _height * _width )
    string::size_type _cursor; // 当前屏幕（Screen）位置
    short _height;    // 行数
    short _width;     // 列数
};
```

因为已经决定采用 string 作为 Screen 类对象的内部表示，所以数据成员_screen 的类型是 string。_cursor 是 string 数据成员的索引，它指向当前的 Screen 位置。它的类型是：

```
string::size_type
```

这是一个可移植的类型，用来存放 string 的索引值（6.8 节引入了 size_type）。

与变量声明一样，我们没有必要分别声明两个 short 型的成员。下列定义与上面的 Screen 定义是等价的：

```
class Screen {
    /*
    * _Screen 指向一个长度为 _height * _width 的字符串;
    * _cursor 指向屏幕当前位置
    * _height 和 _width 指向行数和列数
    */
    string _screen;
    string::size_type _cursor;
    short _height, _width;
};
```

类数据成员可以是任意类型。例如：

```
class StackScreen {
    int topStack;
    void (*handler)();   // 函数的指针
    vector<Screen> stack; // 类的 vector
};
```

目前在本小节中看到的数据成员都是非静态（nonstatic）的数据成员。类也可以有静态（static）数据成员。静态数据成员有特殊的属性，这将在 13.5 节介绍。

正如你已经看到的，数据成员的声明看起来很像在块域或名字空间域中的变量声明。但是，除了静态（static）数据成员外，数据成员不能在类体中被显式地初始化。例如：

```
class First {
    int memi = 0; // 错误
    double memd = 0.0; // 错误
};
```

类的数据成员通过类的构造函数进行初始化。关于类的构造函数曾经在 2.3 节简要介绍过。我们将在第 14 章进一步详细讨论构造函数和类的初始化。

13.1.2 成员函数

用户会希望在 Screen 类型的对象上执行各种各样的操作。例如，要求一组光标移动操作。必须提供测试和设置部分屏幕的能力、用户应该能够把一个 Screen 对象拷贝到另一个上，用户也应该能够在运行时刻设置屏幕的实际维数。这些操作可以用类成员函数来实现。

类的成员函数被声明在类体中。成员函数的声明看起来像是名字空间域中所出现的函数声明。（记住，全局域也是一个名字空间域。8.2 节讨论了全局函数。8.5 节讨论了名字空间。）例如：

```
class Screen {
public:
    void home();
    void move( int, int );
    char get();
    char get( int, int );
    bool checkRange( int, int );
    // ...
};
```

成员函数的定义也可以被放在类体内。例如：

```
class Screen {
public:
    // home() and get() 的定义
    void home() { _cursor = 0; }
    char get() { return _screen[_cursor]; }

    // ...
};
```

函数 home()把光标定位在屏幕的左上角，函数 get()返回当前光标位置的字符值。成员函数与普通函数不同，表现在下面的属性上：

- 成员函数被声明在它的类中，这意味着该成员函数名在类域之外是不可见的。我们可以通过“点（.）或箭头（->）成员访问操作符”引用成员函数，如下所示：

```
ptrScreen->home();
myScreen.home();
```

13.9 节将详细讨论类域。

- 成员函数拥有访问该类的公有和私有成员的特权，而一般来说，普通函数只能访问类的公有成员。当然，一般而言，一个类的成员函数对另一个类的成员没有访问特权。

成员函数可以是重载的函数（重载函数在第 9 章中给出），但是，一个成员函数只能重载自己类的其他成员函数。一个类的成员函数与在其他类或名字空间中声明的函数无关，因此，也不能重载它们。例如 get(int,int)的声明只重载了前面在类 Screen 中声明的成员函数 get()：

```
class Screen {
public:
    // 重载成员函数 get() 的声明
    char get() { return _screen[_cursor]; }
    char get( int, int );
    // ...
};
```

我们将在 13.3 节更详细地介绍类成员函数。

13.1.3 成员访问

一个类类型的内部表示在初始使用之后被各种程序更改的事情经常发生。例如，假设我们对 Screen 类的用户进行了研究，发现所有定义的 Screen 类对象都是 80*24。在这种情况下，我们可能会希望实现一个少一些灵活性、但更有效的 Screen 类：

```
class Screen {
public:
    // 成员函数
private:
    // 静态成员初始化在 13.5 节讨论
    static const int _height = 24;
    static const int _width = 80;
    string _screen;
    string::size_type _cursor;
};
```

成员函数原来的实现——即它们怎样操纵类的数据成员——现在已经不再合适了。成员函数必须被重新实现。但是，这种变化不能要求类成员函数的接口（它们的参数表和返回类型）也跟着发生变化。

如果数据成员是公有的，则它们可以被程序的任何函数访问，在类的内部表示上的这种变化对于 Screen 类的用户的影响是什么呢？

- 旧的 Screen 类的实现中，所有直接访问数据成员的函数都被打破了。在程序可以被再次使用之前，我们必须找到这些代码并重写它们。
- 因为成员函数的接口没有改变，所以每个只通过 Screen 成员函数操纵 Screen 对象

的函数不要求改变原先已经生效的代码。但是，因为成员函数本身被重新实现了，所以我们必须重新编译程序。

信息隐藏（Information hiding）是为了防止程序的函数直接访问类类型的内部表示而提供的一种形式化机制。类成员的访问限制是通过类体内被标记为 public、private 以及 protected 的部分来指定的。关键字 public、private 和 protected 被称为**访问限定符**（access specifier）。在公有（public）区内被声明的成员是公有成员；在私有（private）或被保护的（protected）区域内被声明的成员是私有或被保护的成员。

- **公有成员**（public member）在程序的任何地方都可以被访问。实行信息隐藏的类将其 public 成员限制在成员函数上，这种函数定义了可以被一般程序用来操纵该类类型对象的操作。
- **私有成员**（private member）只能被成员函数和类的友元访问，实行信息隐藏的类把其数据成员声明为 private。
- **被保护成员**（protected member）对派生类（derived class）就像 public 成员一样，对其他程序则表现得像 private。［我们在第 2 章的 IntArray 类中看到了怎样使用 protected 成员的实例。关于 protected 成员的完全讨论要到第 17 章才进行，那时将介绍派生类以及继承（inheritance）的概念。］

下面 Screen 的定义指定了它的 public 区和 private 区：

```
class Screen {
public:
     void home(){ _cursor = 0; }
     char get() { return _screen[_cursor]; }
     char get( int, int );
     void move( int, int );
     // ...
private:
     string _screen;
     string::size_type _cursor;
     short _height, _width;
};
```

为了方便起见，类的 public 成员被放在前面。（关于为什么老的 C++代码把 private 成员放在前面，以及为什么这种风格在某些地方仍然存在的讨论，请参见［LIPPMAN96a］。）private 成员被列在类体的尾部。

一个类可以包含多个 public、private、protected 区。每个区一直有效，直到另一个区标签或类体的结束右括号出现为止。如果没有指定访问限定符，则缺省情况下，在类体的开始左括号后面的区是 private 区。

13.1.4 友元

在某些情况下，允许某个函数而不是整个程序可以访问类的私有成员，这样做会比较方便。友元（friend）机制允许一个类授权其他的函数访问它的非公有成员。

友元声明以关键字 friend 开头，它只能出现在类的声明中。由于友元不是授权友谊的类的成员。所以它们不受其在类体中被声明的 public、private 和 protected 区的影响。这里，我

们选择把所有友元声明组织起来放在类头之后，

```
class Screen {
    friend istream&
        operator>>( istream&, Screen& );
    friend ostream&
        operator<< ( ostream&, const Screen& );
public:
    // ... Screen 类的其他部分
};
```

输入输出操作符现在可以直接引用类 Screen 的成员，而不会发生错误。输入输出操作符的简单重新实现可能如下：

```
ostream& operator<< ( ostream& os, const Screen& s )
{
    // ok: 指向 height, _width, 和 _screen
    os << "<" << s._height
        << "," << s._width << ">";
    os << s._screen;
    return os;
}
```

一个友元或许是一个名字空间函数、另一个前面定义的类的一个成员函数，也可能是一个完整的类。在使一个类成为友元时，友元类的所有成员函数都被给予访问“授权友谊的类的非公有成员”的权力。（15.2 节将给出对友元的详细讨论。）

13.1.5 类声明和类定义

一旦到了类体的结尾，即结束右括号，我们就说一个类被定义了一次。一旦定义了一个类，则该类的所有成员就都是已知的，类的大小也是已知的了。

我们也可以声明一个类但是并不定义它。例如：

```
class Screen; // Screen 类的声明
```

这个声明向程序引入了一个名字 Screen，指示 Screen 为一个类类型。

但是我们只能以有限的方式使用已经被声明但还没有被定义的类类型。如果没有定义类那么我们就不能定义这类类型的对象，因为类类型的大小不知道，编译器不知道为这种类类型的对象预留多少存储空间。

但是，我们可以声明指向该类类型的指针或引用。允许指针和引用是因为它们都有固定的大小，这与它们指向的对象的大小无关。但是，因为该类的大小和类成员都是未知的，所以要等到完全定义了该类，我们才能将解引用操作符（*）应用在这样的指针上，或者使用指针或引用来指向某一个类成员。

只有已经看到了一个类的定义，我们才能把一个数据成员声明成该类的对象。在程序文本中还没有看到该类定义的地方，数据成员只能是该类类型的指针或引用。例如下面是类 StackScreen 的定义，它有一个数据成员是指向 Screen 类的指针，这里 Screen 只有声明没有定义：

```
class Screen; // 声明
```

```
class StackScreen {
    int topStack;

    // ok: 指向一个 Screen 对象
    Screen *stack;
    void (*handler)();
};
```

因为只有当一个类的类体已经完整时，它才被视为已经被定义，所以一个类不能有自身类型的数据成员。但是，当一个类的类头被看到时，它就被视为已经被声明了，所以一个类可以用指向自身类型的指针或引用作为数据成员。例如：

```
class LinkScreen {
    Screen window;
    LinkScreen *next;
    LinkScreen *prev;
};
```

练习 13.1

给出一个类 Person，它有下列两个数据成员

```
string _name;
string _address;
```

以及下列成员函数

```
Person( const string &n, const string &a )
    : _name( n ), _address( a ) { }
string name() { return _name; }
string address() { return _address; }
```

你会把哪些成员声明在公有区内，哪些声明在私有区内？说明理由。

练习 13.2

请说明类定义与类声明之间的区别，什么时候用类声明？什么时候用类定义？

13.2 类对象

类的定义，如类 Screen，不会引起存储区分配。只有当定义一个类的对象时，系统才会分配存储区。例如，给出下列 Screen 类的实现：

```
class Screen {
public:
    // 成员函数
private:
    string              _screen;
    string::size_type   _cursor;
    short               _height;
    short               _width;
};
```

如下定义

```
Screen myScreen;
```

将分配一块足够包含 Screen 类的四个数据成员的存储区。名字 myScreen 引用到这块存储区。每个类对象都有自己的类数据成员拷贝。修改 myScreen 的数据成员不会改变任何其他 Screen 对象的数据成员。

类类型的对象有一个域，它是由对象定义在程序文本文件中的位置决定的。一个类的对象可能被定义在一个与“类类型被定义的域”不同的域中。例如：

```
class Screen {
    // 成员列表
};
int main()
{
    Screen mainScreen;
}
```

类 Screen 在全局域中被声明，而 mainScreen 对象则在函数 main()的局部域中被声明。

类类型的对象也有生命期。根据对象是在一个名字空间域还是在一个局部域中被声明，以及它是否被声明为 static，对象可能在整个程序执行期间存在，或只在一个特殊的函数调用执行期间存在。当考虑域和生命期时，类类型的对象与其他对象非常相像。对象的域和生命期在第 8 章中介绍。

一个对象可以被同一类类型的另一个对象初始化或赋值。缺省情况下，拷贝一个类对象与拷贝它的全部数据成员等价。例如：

```
Screen bufScreen = myScreen;
// bufScreen._height   = myScreen._height
// bufScreen._width    = myScreen._width
// bufScreen._cursor   = myScreen._cursor
// bufScreen._screen   = myScreen._screen
```

我们也可以声明类对象的指针和引用。类类型的指针可以用同一类类型的类对象的地址做初始化或赋值。类似地，类类型的引用也可以用同一类类型的对象的左值作初始化。（面向对象的程序设计对此作了扩展，允许基类的引用或指针引用到派生类的对象）：

```
int main()
{
    Screen myScreen, bufScreen[10];
    Screen *ptr = new Screen;
    myScreen = *ptr;
    delete ptr;
    ptr = bufScreen;

    Screen &ref = *ptr;
    Screen &ref2 = bufScreen[6];
}
```

缺省情况下，当一个类对象被指定为函数实参或函数返回值时，它就被按值传递。我们也可以把一个函数参数或返回类型声明为一个类类型的指针或引用。7.3 节给出了类类型的指针或引用被作为参数的例子，并说明何时应该使用它们。7.4 节给出了类类型的指针或引用的返回类型，并说明应该何时使用它们。

我们必须用成员访问操作符来访问类对象的数据成员或成员函数，点成员访问操作符（.）与类对象或引用联用；箭头访问操作符（->）与类对象的指针联用。例如：

```
#include "Screen.h"
bool isEqual( Screen& s1, Screen *s2 )
{ // 如果不相等返回 false，相等则返回 true
    if ( s1.height() != s2->height() ||
         s1.width() != s2->width() )
        return false;

    for ( int ix = 0; ix < s1.height(); ++ix )
        for ( int jy = 0; jy < s2->width(); ++jy )
            if ( s1.get( ix, jy ) != s2->get( ix, jy ) )
                return false;

    return true; // 还在这里？那就是相等.
}
```

isEqual()是非成员函数，它比较两个 Screen 对象是否相等。isEqual()没有访问 Screen 的私有数据成员的特权，所以不能直接引用 s1 和 s2 的数据成员。它必须依赖于 Screen 类的公有成员函数。

isEqual()为了获得 Screen 的 height 和 width 的值，它必须使用成员函数 height()和 width()，它们被称为访问函数，这些函数提供了对类的私有数据成员的只读访问。它们的实现很简单：

```
class Screen {
public:
    int height() { return _height; }
    int width() { return _width; }
    // ...

private:
    short _height, _width;
    // ...
};
```

把箭头成员访问操作符应用在指向类对象的指针上，或者“把解引用操作符（*）应用在指针上获得其指向的类对象，然后再应用点成员访问操作符访问所需要的成员函数”，这两者是等价的。例如表达式

```
s2->height()
```

可以写为

```
(*s2).height
```

结果完全相同。

13.3 类成员函数

类的成员函数是一组操作的集合，用户可以在该类的对象上执行这些操作。能够在类 Screen 上执行的操作集由 Screen 类中的成员函数定义：

```
class Screen {
public:
     void home() { _cursor = 0; }
     void move( int, int );
     char get() { return _screen[_cursor]; }
     char get( int, int );
     bool checkRange( int, int );
     int height() { return _height; }
     int width() { return _width; }
     // ...
};
```

虽然每个类对象都有自己的类数据成员拷贝，但是，每个类成员函数的拷贝只有一份，例如：

```
Screen myScreen, groupScreen;
myScreen.home();
groupScreen.home();
```

当针对对象 myScreen 调用函数 home()时，在 home()中访问的成员_cursor 是对象 myScreen 的数据成员；当针对对象 groupScreen 调用 home()时，数据成员_cursor 引用的是对象 groupScreen 的数据成员。但是，两者调用的是同一个函数 home()。同一个成员函数怎样能引用两个不同类对象的数据成员呢？这种支持是通过 this 指针实现的，关于 this 指针将在下一节中介绍。

13.3.1 inline 和非 inline 成员函数

注意，函数 home()、get()、height()和 width()的定义是在类体内提供的。这些函数被称为“在类定义中定义的内联（inline）函数”。这些函数被自动作为 inline 函数处理。关于 inline 函数在 7.6 节介绍。

我们也可以通过在成员函数的返回类型前显式地指定关键字 inline，在类体内将这些成员函数声明为 inline 的，如下所示：

```
class Screen {
public:
     // 用 inline 关键字
     // 声明 inline 成员函数
     inline void home() { _cursor = 0; }
     inline char get() { return _screen[_cursor]; }
     // ...
};
```

在这个例子中，home()和 get()的定义与前面例子中的 home()和 get()的定义完全相同，在后者中省略了关键字 inline。因为这个关键字是冗余的，所以我们的例子在类体中没有为成员函数显式地指定关键字 inline。

一两行以上的成员函数最好被定义在类体之外。这要求一个特殊的声明语化，来标识一个函数是一个类的成员：成员函数名必须被它的类名限定修饰（qualified）。例如，下面是函数 checkRange()的定义，这里函数名用 Screen::限定修饰：

```
#include <iostream>
#include "Screen.h"
```

```
// 成员函数名用 Screen:: 限定修饰
bool Screen::checkRange( int row, int col )
{ // validate coordinates
     if ( row < 1 || row > _height ||
               col < 1 || col > _widt h ) {
          cerr << "Screen coordinates ( "
               << row << ", " << col
               << " ) out of bounds.\n";
          return false;
     }
     return true;
}
```

成员函数必须先在其类体内被声明，而且类体必须在成员函数被定义之前先出现。例如，如果在函数 checkRange()定义之前没有包含头文件 Screen.h，那么前面的程序就是错的。类体定义了类成员的完整列表。一旦类体结束，这个列表就不能再扩充。

通常，在类体外定义的成员函数不是 inline 的。但是，这样的函数也可以被声明为 inline 函数，可以通过显式地在类体中出现的函数声明上使用关键字 inline，或者通过在类体外出现的函数定义上显式使用关键字 inline，或者两者都用。例如，下面的实现定义了 move()是 Screen 的一个 inline 函数：

```
inline void Screen::move( int r, int c )
{ // 将 _cursor 称到绝对位置
     if ( checkRange( r, c ) )  // 位置合法吗?
     {
          int row = (r-1) * _width; // 行位置
          _cursor = row + c - 1;
     }
}
```

也可以如下指定关键字 inline，将函数 get(int, int)声明为内联的：

```
class Screen {
public:
     inline char get( int, int );
     // 其他函数声明未变
};
```

它的函数定义跟在类定义之后，关键字 inline 可以被省略：

```
char Screen::get( int r, int c )
{
     move( r, c );// _cursor 位置
     return get();// 另一个 get() 成员函数
}
```

由于内联函数必须在调用它的每个文本文件中被定义，所以没有在类体中定义的内联成员函数必须被放在类定义出现的头文件中。例如，前面给出的 move()和 get()的定义应该被放在头文件 Screen.h 中，且跟在类 Screen 的定义后面。

13.3.2 访问类成员

无论成员函数是在类体内还是外面，我们都说它在类域内。这有两个含义：

1．成员函数的定义可以引用任何一个类成员，无论该成员是私有的还是公有的，都不会破坏类访问限制。

2．成员函数可以直接访问它所属的类的成员，而无需使用点或箭头成员访问操作符。例如：

```
#include <string>

void Screen::copy( const Screen &sobj )
{
    // 如果这个 Screen 对象与 sobj 是同一个对象
    // 则无需拷贝
    // 我们将在 13.4 节介绍 this 指针
    if ( this != &sobj )
    {
        _height = sobj._height;
        _width = sobj._width;
        _cursor = 0;

        // 创建一个新字符串
        // 它的内容与 sobj._screen 相同
        _screen = sobj._screen;
    }
}
```

我们注意到，即使数据成员_screen、_height、_width 和_cursor 是类 Screen 的私有成员，成员函数 copy()仍可以引用这些私有成员，而没有错误。如果数据成员，如_screen、_height。_width 和_cursor，被使用的时候没有通过成员访问操作符，则成员函数引用的是调用者（一个类对象）的数据成员。例如，如果如下调用成员函数 copy()：

```
#include "Screen.h"
int main()
{
    Screen s1;

    // 设置 s1 的内容
    Screen s2;
    s2.copy(s1);

    // ...
}
```

在成员函数 copy()的定义中，参数 sobj 指 main()中定义的对象 s1。在点成员访问操作符之前提到的对象 s2 是调用成员函数 copy()的对象。对于这个 copy()调用，在 copy()的定义中，没有用成员访问操作符引用的数据成员_screen、_height、_width 和_cursor 实际上引用了对象 s2 的数据成员。在下一节，我们将更详细地介绍在成员函数定义中访问类成员，以及怎样通过 this 指针支持这种访问。

13.3.3 私有与公有成员函数

类成员函数可以被声明在类体的 public、protected 和 private 区内。怎样判断一个成员函

数应该被放在哪儿呢？公有成员函数定义了“类的用户可能想执行的操作”。公有函数集定义了类的接口（interface）。例如，类 Screen 的成员函数 home()、move()和 get()定义了可被程序用来操纵 Screen 型对象的操作。

因为我们通过把数据成员定义为私有的，以便向类的用户隐藏了类的内部表示，所以我们必须提供公有成员函数来操纵 Screen 对象，这就是我们所知晓的**信息隐藏**（information hidng）。信息隐藏使得代码免受“因一个类的内部表示的变化而带来的影响”。

防止类对象的内部状态被程序随机地修改也同等重要。较小的函数集提供了该对象的全部修改动作。如果出现错误，则错误的查找空间局限在这个函数集中，这大大地简化了程序的维护和修正问题。

到目前为止，我们只看到支持读取私有数据的成员函数。下面有两个 set()函数，它们允许用户修改 Screen 对象。首先，这两个新成员函数必须被加到类体中：

```
class Screen {
public:
    void set( const string &s );
    void set( char ch );
    // 其他的成员函数声明保持不变
};
```

这些成员函数的定义如下：

```
void Screen::set( const string &s )
{ // 在当前 _cursor 位置写字符串
    int space = remainingSpace();
    int len = s.size();
    if ( space < len ) {
        cerr << "Screen: warning: truncation: "
             << "space: " << space
             << "string length: " << len << endl;
        len = space;
    }

    _screen.replace( _cursor, len, s );
    _cursor += len - 1;
}

void Screen::set( char ch )
{
    if ( ch == '\0' )
        cerr << "Screen: warning: "
             << "null character (ignored).\n";
    else _screen[_cursor] = ch;
}
```

我们的 Screen 实现假定，Screen 对象不包含内嵌的空（null）字符。这是 set()不允许向 Screen 写空（null）字符的原因。

给出的函数是公有成员函数。它们可以在程序的任何位置上被调用。但是，私有成员函数只能被类的其他成员函数（和友元）调用，程序不能直接调用它们。在实现类抽象时，私有成员函数为其他成员函数提供支持。在函数 set(const, string&)中用到的函数 remainingSpace()

是这些函数之一。成员函数 remainingSpace()是类 Screen 的私有成员函数：

```
class Screen {
public:
     // 其他的成员函数声明保持不变
private:
     inline int remainingSpace();
};
```

remainingSpace()返回屏幕上剩余的空间数：

```
inline int Screen::remainingSpace()
{ // 当前位置不再是剩余的
     int sz = _width * _height;
     return( sz - _cursor );
}
```

关于 protected 成员的讨论将推延到第 17 章。

下面是一个小程序，它使用了到目前为止我们所实现的全部成员函数：

```
#include "Screen.h"
#include <iostream>

int main() {
     Screen sobj(3,3); // 13.3.4 节定义的构造函数
     string init("abcdefghi");
     cout << "Screen Object ("
          << sobj.height() << ", "
          << sobj.width() << " )\n\n";

     // 设置屏幕的内容
     string::size_type initpos = 0;
     for ( int ix = 1; ix <= sobj.width(); ++ix )
          for ( int iy = 1; iy <= sobj.height(); ++iy )
          {
               sobj.move( ix, iy );
               sobj.set( init[ initpos++ ] );
          }

     // 打印屏幕的内容
     for ( int ix = 1; ix <= sobj.width(); ++ix )
     {
          for ( int iy = 1; iy <= sobj.height(); ++iy )
               cout << sobj.get( ix, iy );
          cout << "\n";
     }

     return 0;
}
```

编译并运行这个程序，产生下列输出：

```
Screen Object ( 3, 3 )
abc
```

```
def
ghi
```

13.3.4 特殊的成员函数

有一组特殊的成员函数可以管理类对象并处理诸如初始化、赋值、内存管理、类型转换以及析构等活动。这些函数通常由编译器隐式调用。

初始化成员函数被称为**构造函数**（constructor）。每次定义一个类对象或由 new 表达式分配一个类对象时都会调用它。构造函数的名字必须与类名相同。下面是 Screen 类构造函数的声明，它为参数 hi、wid 和 bkground 提供了缺省实参值：

```
class Screen {
public:
     Screen( int hi = 8, int wid = 40, char bkground = '#');
     // 其他的成员函数声明保持不变
};
```

下面是 Screen 构造函数的定义：

```
Screen::Screen( int hi, int wid, char bk ) :
     _height( hi ),              // 用 hi 初始化 _height
     _width( wid ),              // 用 wid 初始化 _width
     _cursor ( 0 ),              // 初始化 _cursor 为 0
     _screen( hi * wid, bk )     // _Screen 的大小为 hi*wid
// 所有位置用 bk 的字符值初始化
{    // 所有的工作都由成员初始化列表完成
     // 14.5 节将讨论成员初始化列表
}
```

Screen 的构造函数自动初始化每个被声明的 Screen 对象。例如：

```
Screen s1;                      // Screen(8,40,'#')
Screen *ps = new Screen( 20 ); // Screen(20,40,'#')

int main() {
     Screen s(24,80,'*');       // Screen(24,80,'*')
     // ...
}
```

第 14 章将更详细地介绍构造函数、析构函数和赋值操作符，第 15 章将更详细地介绍转换函数和内存管理函数。

13.3.5 const 和 volatile 成员函数

通常，程序中任何试图修改 const 对象的动作都会被标记为编译错误：

```
const char blank = ' ';
blank = '\n'; // 错误
```

但是，程序通常不直接修改类对象，又是在必须修改类的对象时，才调用公有成员函数集来完成。为尊重类对象的常量性，编译器必须区分不安全与安全的成员函数（即区分试图修改类对象与不试图修改类对象的函数）。例如：

```
const Screen blankScreen;
```

```
blankScreen.display();          // 读类对象
blankScreen.set( '*' );         // 错误：修改类对象
```

类的设计者通过把成员函数声明为 const，以表明它们不修改类对象。例如：

```
class Screen {
public:
     char get() const { return _screen[_cursor]; }
     // ...
}
```

只有被声明为 const 的成员函数才能被一个 const 类对象调用。关键字 const 被放在成员函数的参数表和函数体之间。对于在类体之外定义的 const 成员函数，我们必须在它的定义和声明中同时指定关键字 const。例如：

```
class Screen {
public:
     bool isEqual( char ch ) const;
     // ...
private:
     string::size_type _cursor;
     string _screen;
     // ...
};

bool Screen::isEqual( char ch ) const
{
     return ch == _screen[_cursor];
}
```

把一个修改类数据成员的函数声明为 const 是非法的。例如，在如下简化的 Screen 定义中，

```
class Screen {
public:
     int ok() const { return _cursor; }
     void error( int ival ) const { _cursor = ival; }
     // ...

private:
     string::size_type _cursor;
     // ...
};
```

ok()的定义是一个有效的 const 成员函数定义，因为它没有改变_cursor 的值。但是，error()的定义修改了_cursor 的值，因此它不能被声明为一个 const 成员函数。这个函数定义将导致下面的编译器错误消息：

```
error: cannot modify a data member within a const member function
```

一般来说，任何一个类如果期望被广泛使用，就应该把那些不修改类数据成员的成员函数声明为 const 成员函数。但是，把一个成员函数声明为 const 并不能阻止程序员可能做到的所有修改动作。把一个成员函数声明为 const 可以保证这个成员函数不修改类的数据成员，但是，如果该类含有指针，那么在 const 成员函数中就能修改指针所指的对象。编译器不会把这种修改检测为错误，这常常令 C++初学者吃惊。例如：

```
#include <cstring>
class Text {
public:
     void bad( const string &parm ) const;
private:
     char *_text;
};

void Text::bad( const string &parm ) const
{
     _text = parm.c_str();              // 错误：不能修改 _text
     for ( int ix = 0; ix < parm.size(); ++ix )
          _text[ix] = parm[ix];         // 不好的风格，但不是错误的
}
```

尽管_text 不能被修改，但是_text 的类型是 char*，在类 Text 的 const 成员函数中可以修改_text 指向的字符。成员函数 bad()反映了一种不良的程序设计风格。但是编译器不能帮助检测出这样的情况，只有程序员自己保持警惕，因为 const 成员函数不能保证，在调用成员函数期间类对象引用的所有东西都保持不变。

const 成员函数可以被相同参数表的非 const 成员函数重载。例如：

```
class Screen {
public:
     char get(int x, int y);
     char get(int x, int y) const;
     // ...
};
```

在这种情况下，类对象的常量性决定了调用哪个函数：

```
int main() {
     const Screen cs;
     Screen s;

     char ch = cs.get(0,0);     // 调用 const 成员
     ch = s.get(0,0);           // 调用非 const 成员
}
```

构造函数和析构函数是两个例外，即使构造函数和析构函数不是 const 成员函数，const 类对象也可以调用它们。当构造函数执行结束、类对象已经被初始化时，类对象的常量性就被建立起来了。析构函数一被调用，常量性就消失。所以一个 const 类对象“从构造完成时刻到析构开始时刻”这段时间内被认为是 const。

也可以将成员函数声明为 volatile（volatile 限定修饰符在 3.13 节介绍）。如果一个类对象的值可能被修改的方式是编译器无法控制或检测的（例如，如果它是表示 I/O 端口的数据结构），则把它声明为 volatile。与 const 类对象类似，对于一个 volatile 类对象，只有 volatile 成员函数、构造函数和析构函数可以被调用：

```
class Screen {
public:
     char poll() volatile;

     // ...
```

```
};
char Screen::poll() volatile { ... }
```

13.3.6 mutable 数据成员

当我们把一个 Screen 类对象声明为 const 时出现了一些问题。我们期望的行为是，一旦 const Screen 对象被初始化，它的内容就不能被修改。但是我们应该能够监视到 Screen 对象的内容。例如，给出下面的 Screen 对象 cs，

```
const Screen cs( 5, 5 );
```

我们想监视在位置(3, 4)的内容。我们这样做：

```
// 读位置(3, 4)的内容
// 喔！不能工作
cs.move( 3, 4 );
char ch = cs.get();
```

但是，这不能工作。你知道为什么吗？move()不是 const 成员函数，而且不能很容易地变成 const 成员函数。move()的定义如下：

```
inline void Screen::move( int r, int c )
{
    if ( checkRange( r, c ) )
    {
        int row = (r-1) * _width;
        _cursor = row + c - 1; // 修改 _cursor
    }
}
```

我们注意到 move()修改了数据成员_cursor，因此若不加改动，它就不能被声明为 const。

但是，对一个 Screen 类的 const 对象不能修改_cursor，这看起来可能很奇怪，因为_cursor 只是一个索引，修改_cursor 不会修改 Screen 本身的内容。我们只是想记住要被监视的 Screen 位置。即使 Screen 对象是 const，也应该允许修改_cursor，因为这么做对于监视 Screen 对象的内容是必需的，而且又不会修改 Screen 本身的内容。

为了允许修改一个类的数据成员，即使它是一个 const 对象的数据成员，我们也可以把该数据成员声明为 mutable（**易变的**）。mutable 数据成员永远不会是 const 成员，即使它是一个 const 对象的数据成员。mutable 成员总可以被更新，即使是在一个 const 成员函数中。为把一个成员声明为 mutable 数据成员，我们必须把关键字 mutable 放在类成员表中的数据成员声明之前：

```
class Screen {
public:
    // 成员函数
private:
    string                      _screen;
    mutable string::size_type   _cursor;    // mutable 成员
    short                       _height;
    short                       _width;
};
```

现在任何 const 成员函数都可以修改_cursor，我们可以把成员函数 move()声明为 const：

即使 move()修改了数据成员_cursor，也不会有编译错误产生：

```
// move() 是一个 const 成员函数
inline void Screen::move( int r, int c ) const
{
    // ...
    // ok: const 成员函数可以修改 mutable 成员
    _cursor = row + c - 1;
    // ...
}
```

现在我们可以执行本小节开始时给出的操作来监视 Screen 对象 cs，而不会有错误发生。

我们注意到只有_cursor 被声明为 mutable 数据成员。而_screen、_height 和_width 都没有，因为这些数据成员的值在 const 的 Screen 类对象中是不应该被改变的。

练习 13.3

请解释下列调用中 copy()的行为：

```
Screen myScreen;
myScreen.copy( myScreen );
```

练习 13.4

其他的光标移动操作可能包含“向前（forward）或向后（backward）一次移动一个字符”。当到达屏幕的左上角或右下角时；光标会折回来。请实现 forward()和 backward()函数。

练习 13.5

上下移动一行可能是一种很有用的能力。当到达屏幕顶或底部时，光标不会折回，而是发出一个铃声并保持在原地。实现 up()和 down()函数，向 cout 写字符“007”会发出铃声。

练习 13.6

回顾到目前为止所介绍的 Screen 成员函数，在适当的地方把成员函数改变成 const 成员函数。说明你这样做的理由。

13.4 隐含的 this 指针

每个类对象都将维护自己的类数据成员的拷贝。例如：

```
int main() {
    Screen myScreen( 3, 3 ), bufScreen;

    myScreen.clear();
    myScreen.move( 2, 2 );
    myScreen.set( '*' );
    myScreen.display();

    bufScreen.reSize( 5, 5 );
```

```
        bufScreen.display();
    }
```

myScreen 有自己的_screen、_height、_width 和_cursor 数据成员。BufScreen 也有自己独立的一组成员，仍是每个类成员函数只存在一份拷贝。myScreen 和 bufScreen 都调用任何特定成员函数的同一份拷贝。

我们在上节已经看到，成员函数可以引用自己的类成员而无需使用成员访问操作符。例如，函数 move()的定义如下：

```
inline void Screen::move( int r, int c )
{
    if ( checkRange( r, c ) )  // 无效位置?
    {
        int row = (r-1) * _width; // 行位置
        _cursor = row + c - 1;
    }
}
```

如果调用了对象 myScreen 的函数 move()，那么在 move()中访问的数据成员_width 和_cursor 是 myScreen 的数据成员。如果调用了对象 bufScreen 的函数 move()，则访问的是 bufScreen 的数据成员。move()操纵的数据成员_cursor 怎样被依次绑定到属于 myScreen 和 bufScreen 的数据成员上呢？简短地回答起来，就是用 this 指针。

每个类成员函数都含有一个指向被调用对象的指针，这个指针被称为 this。在非 const 成员函数中，它的类型是指向该类类型的指针：在 const 成员函数中，是指向 const 类类型的指针。而在 volatile 成员函数中，是指向 volatile 类类型的指针。例如，在类 Screen 的成员函数 move()中，this 指针的类型是 Screen*。在类 List 的非 const 成员函数中，this 指针的类型是 List*。

因为 this 指针指向要调用其成员函数的类对象，所以如果函数 move()被对象 myScreen 调用，则 this 指针指向对象 myScreen。类似地，如果函数 move()被对象 bufScreen 调用，则 this 指针指向对象 bufScreen。Move()操纵的数据成员_cursor，依次被绑定在属于 myScreen 和 bufScreen 的数据成员上。

要想理解这一点，一种方法是看一看编译器是怎样实现 this 指针的。为支持 this 指针，必须要应用两个转变：

1．改变类成员函数的定义。用额外的参数：this 指针，来定义每个成员函数。例如：

```
// 伪代码，说明编译器对一个成员函数定义的展开形式
// 不是合法的 C++ 代码
inline void move( Screen* this, int r, int c )
{
    if ( checkRange( r, c ) )
    {
        int row = (r-1) * this->_width;
        this->_cursor = row + c - 1;
    }
}
```

在这个成员函数定义中，显式使用 this 指针来访问类数据成员_width 和_cursor。

2．改变每个类成员函数的调用，加上一个额外的实参——被调用对象的地址。例如：

```
myScreen.move( 2, 2)
```

被转化为：

```
move( &myScreen, 2, 2 )
```

程序员可以在成员函数定义中显式地引用 this 指针。例如，如下定义成员函数 home()是合法的，尽管这样做不是必要的：

```
inline void Screen::home()
{
    this->_cursor = 0;
}
```

但是，有一种情况下确实需要程序员显式地引用 this 指针，比如我们在前面定义的 Screen 成员函数 copy()。下一小节将给出一些示例。

13.4.1 何时使用 this 指针

函数 main()在对象 myScreen 和 bufScreen 上调用类 Screen 的成员函数，这样的动作是在独立的语句中。我们可以重新定义 Screen 类的成员函数，这样当多个成员函数应用到同一个 Screen 对象上时，我们可以将成员函数调用连接起来。例如，在函数 main()中的调用可以写为：

```
int main() {
    // ...
    myScreen.clear().move(2,2).set('*').display();
    bufScreen.reSize(5,5).display();
}
```

这看起来似乎符合操作 Screen 对象的直观方式，这里用到的动作序列是：先清 Screen 对象 myScreen，然后把光标移到位置(2,2)，接着再把该位置的字符设为‘*’、最后显示结果。

成员访问操作符点（.）和箭头（->）是左结合操作符，当看到这些操作符序列时，执行的顺序是从左到右。例如，先调用的是 myScreen.clear()，然后是 myScreen.move()等等。为使 myScreen.move()在 myScreen.clear()之后被调用，clear()必须返回类对象 myScreen。成员函数 clear()的定义必须返回被调用的类对象。正如我们已经看到的，在类成员函数内是通过 this 指针来访问该类对象的。下面是 clear()的实现：

```
// clear() 的声明在类体内
// 它指定了缺省实参 bkground = '#'
Screen& Screen::clear( char bkground )
{ // 重置 cursor 以及清屏幕
    _cursor = 0;

    _screen.assign(   // 赋给字符串
        _screen.size(), // size() 个字符
        bkground // 值都是 bkground
    );

    // 返回被调用的对象
    return *this;
```

```
}
```

注意，这个成员函数的返回类型是 Screen&，它表示该成员函数返回一个引用，指向它自己所属类类型的对象。为了允许在 main()中连接 Screen 的成员函数，成员函数 move()和 set()也需要作修改。它们的返回类型也必须从 void 改变成 Screen&，且必须在函数定义中返回 *this。

类似地，Screen 成员函数 display()必须重新实现如下：

```
Screen& Screen::display()
{
    typedef string::size_type idx_type;

    for ( idx_type ix = 0; ix < _height; ++ix )
    { // 针对每一行
        idx_type offset = _width * ix; // row position
        for ( idx_type iy = 0; iy < _width; ++iy )
            // 针对每一列，输出元素
            cout << _screen[ offset + iy ];
        cout << endl;
    }

    return *this;
}
```

Screen 的成员函数 resize()必须如下实现：

```
// reSize() 的声明在类体内
// 它指定了缺省实参 bkground = '#'
Screen& Screen::reSize( int h, int w, char bkground )
{ // 把屏幕的大小设置到高度 h 和 宽度 w
    // 记住屏幕的内容
    string local(_screen);

    // 替换 _screen 所引用的字符串
    _screen.assign(    // 赋给字符串
        h * w,         // h * w 个字符
        bkground       // 值都是 bkground
    );

    typedef string::size_type idx_type;
    idx_type local_pos = 0;

    // 把原来屏幕的内容拷贝到新的屏幕上
    for ( idx_type ix = 0; ix < _height; ++ix )
    { // 每一行
        idx_type offset = w * ix; // 行位置

        for ( idx_type iy = 0; iy < _width; ++iy )
            // 每一列，赋以原来的值
            _screen[ offset + iy ] = local[ local_pos++ ];
    }
```

```
        _height = h;
        _width = w;

        // _cursor 保持不变
        return *this;
    }
```

在成员函数中，this 指针的用处不全是返回该成员函数被应用的对象。在 13.3 节介绍成员函数 copy()时，我们看到了 this 指针的另一种用法：

```
void Screen::copy( const Screen& sobj )
{
    // 如果 Screen 对象与 sobj 是同一个对象
    // 无需拷贝
    if ( this != &sobj )
    {
        // 把 sobj 的值拷贝到 *this 中
    }
}
```

该 this 指针含有被调用的类对象的地址。如果 sobj 指向的对象的地址与 this 指针值相等，则 sobj 和 this 指向同一对象。拷贝动作是不必要的。我们会在 14.7 节介绍拷贝赋值操作符时再次看到这种结构。

练习 13.7

this 指针可以被用来修改其指向的类对象，也可以用同一类型的新对象覆盖该对象。例如，下面是类 classType 的成员函数 assign()。你能说明它的功能吗？

```
classType& classType::assign( const classType &source )
{
    if ( this != &source )
    {
        this->~classType();
        new (this) classType( source );
    }
    return *this;
}
```

记住，~classType()是析构函数的名字。new 表达式看起来可能有点滑稽，但是我们已经在 8.4 节看到过这种被称为定位 new 表达式（placement new expression）的 new 表达式。

你对这种编码风格有何看法？你认为这是一种安全的操作吗，为什么？

13.5 静态类成员

有时候某个特殊类类型的所有对象都需要访问一个全局对象。可能是要计数在程序的任意一点总共创建了多少个此类类型的对象，这个全局变量或者是指向该类型错误处理例程的指针，或者是指向该类类型对象的自由存储区的指针。在这些情况下，“提供一个所有对象共同使用的全局对象”比“每个类对象维持一个独立的数据成员”要更为有效。尽管这个对

象是一个全局对象，但是它的存在只是为了支持该类抽象的实现。

在这种情况下，类的静态数据成员提供了一个更好的方案。静态数据成员被当作该类类型的全局对象。对于非静态数据成员，每个类对象都有自己的拷贝，而静态数据成员对每个类类型只有一个拷贝。静态数据成员只有一份，由该类类型的所有对象共享访问。

同全局对象相比，使用静态数据成员有两个优势：

1．静态数据成员没有进入程序的全局名字空间，因此不存在与程序中其他全局名字冲突的可能性。

2．可以实现信息隐藏。静态成员可以是 private 成员，而全局对象不能。

在类体中的数据成员声明前面加上关键字 static，就使该数据成员成为静态的。static 数据成员遵从 public/private/protected 访问规则。例如，在下面定义的 Account 类中，_interestRate 是被声明为 double 型的私有静态成员。

```
class Account {
     Account( double amount, const string &owner );
     string owner() { return _owner; }
private:
     static double _interestRate;
     double _amount;
     string _owner;
};
```

为什么把_interestRate 声明为 static，而_amount 和_owner 不呢？这是因为每个 Account 对应不同的主人，有不同数目的钱，而所有 Account 的利率却是相同的。

因为在整个程序中只有一个_interestRate 数据成员，它被所有 Account 对象共享，所以把_interestRate 声明为静态数据成员减少了每个 Account 所需的存储空间。

尽管对于所有 Account 对象，_interestRate 的当前值相同，但是它的值可能随时间而被改变，所以，我们决定不把这个静态数据成员声明为 const。因为_interestRate 是静态的，所以它只需被更新一次，我们就可以保证每个 Account 对象都能够访问到被更新之后的值。要是每个类对象都维持自己的一个拷贝，那么每个拷贝都必须被更新，这将导致效率低下和更大的错误可能。

一般地，静态数据成员在该类定义之外被初始化。如同一个成员函数被定义在类定义之外一样，在这种定义中的静态成员的名字必须被其类名限定修饰。例如，下面是．interestRate 的初始化：

```
// 静态类成员的显式初始化
#include "account.h"
double Account::_interestRate = 0.0589;
```

与全局对象一样，对于静态数据成员，在程序中也只能提供一个定义。这意味着，静态数据成员的初始化不应该被放在头文件中，而应该放在含有类的非 inline 函数定义的文件中。静态数据成员可以被声明为任意类型。它们可以是 const 对象、数组或类对象等等。例如：

```
#include <string>
class Account {
     // ...
private:
```

```
        static const string name;
};

const string Account::name( "Savings Account" );
```

作为特例，有序型的 const 静态数据成员可以在类体中用一常量值初始化。例如，如果决定用一个字符数组而不是 string 来存储账户的姓名，那么我们可以用 int 型的 const 数据成员指定该数组的长度。例如：

```
// 头文件
class Account {
    // ...
private:
    static const int nameSize = 16;
    static const char name[nameSize];
};

// 文本文件
const int Account::nameSize;          // 必需的成员定义
const char Account::name[nameSize] = "Savings Account";
```

关于这个特例，有一些有趣的事情值得注意。用常量值作初始化的有序类型的 const 静态数据成员是一个**常量表达式**（constant expression）。如果需要在类体中使用这个被命名的值，那么，类设计者可声明这样的静态数据成员。例如，因为 const 静态数据成员 nameSize 是一个常量表达式，所以类的设计者可以用它来指定数组数据成员 name 的长度。

在类体内初始化一个 const 静态数据成员时，该成员必须仍然要被定义在类定义之外。但是，因为这个静态数据成员的初始值是在类体中指定的，所以在类定义之外的定义不能指定初始值。

因为 name 是一个数组（不是有序类型），所以它不能在类体内被初始化、任何试图这么做的行为都会导致编译时刻错误。例如：

```
class Account {
    // ...
private:
    static const int nameSize = 16;      // ok: 有序类型

    static const char name[nameSize] =
        "Savings Account";                // 错误
};
```

name 必须在类定义之外被初始化。

这个例子还说明了一点，我们注意到成员 nameSize 指定了数组 name 的长度，而数组 name 的定义出现在类定义之外：

```
const char Account::name[nameSize] = "Savings Account";
```

nameSize 没有被类名 Account 限定修饰。尽管 nameSize 是私有成员，但是 name 的定义仍没有错。怎么会这样？如同类成员函数的定义可以引用类的私有成员一样，静态数据成员的定义也可以。静态数据成员 name 的定义是在它的类的域内，当限定修饰名 Account::name 被看到之后，它就可以引用 Account 的私有数据成员。我们将在 13.9 节看到更多有关类域的内容。

在类的成员函数中可以直接访问该类的静态数据成员，而不必使用成员访问操作符：

```
inline double Account::dailyReturn()
{
    return( _interestRate / 365 * _amount );
}
```

但是在非成员函数中，我们必须以两种方式之一访问静态数据成员。可以使用成员访问操作符：

```
class Account {
    // ...
private:
    friend int compareRevenue( Account& , Account* );
    // 余下部分未变
};

// 引用和指针参数来说明对象和指针访问
int compareRevenue( Account &ac1, Account *ac2 )
{
    double ret1, ret2;
    ret1 = ac1._interestRate * ac1._amount;
    ret2 = ac2->_interestRate * ac2->_amount;

    // ...
}
```

ac1._interestRate 和 ac2._interestRate 都引用静态成员 Account::interestRate。

因为类静态数据成员只有一个拷贝，所以它不一定要通过对象或指针来访问。访问静态数据成员的另一种方法是，用被类名限定修饰的名字直接访问它：

```
// 用限定修饰名访问静态成员
if ( Account::_interestRate < 0.05 )
```

当我们不通过类的成员访问操作符访问静态数据成员时，必须指定类名以及紧跟其后的域操作符

```
Account::
```

因为静态成员不是全局对象，所以我们不能在全局域中找到它。下面的 friend 函数 compareRevenue()的定义与刚刚给出的等价：

```
int compareRevenue( Account &ac1, Account *ac2 )
{
    double ret1, ret2;
    ret1 = Account::_interestRate * ac1._amount;
    ret2 = Account::_interestRate * ac2->_amount;
    // ...
}
```

静态数据成员的“惟一性”本质（独立于类的任何对象而存在的惟一实例），使它能够以独特的方式被使用，这些方式对于非 static 数据成员来说是非法的；

1．静态数据成员的类型可以是其所属类，而非 static 数据成员只能被声明为该类的对象的指针或引用。例如：

```
class Bar {
public:
```

```
    // ...
private:
    static Bar mem1;  // ok
    Bar *mem2;    // ok
    Bar mem3;     // 错误
};
```

2．静态数据成员可以被作为类成员函数的缺省实参，而非 static 成员不能。例如：

```
extern int var;
class Foo {
private:
    int var;
    static int stcvar;
public:
    // 错误: 被解析为非 static 的 Foo::var
    // 没有相关的类对象
    int mem1( int = var );

    // ok: 解析为 static 的 Foo::stcvar
    // 无需相关的类对象
    int mem2( int = stcvar );

    // ok: int var 的全局实例
    int mem3( int = ::var );
};
```

13.5.1 静态成员函数

成员函数 raiseInterest()和 interest()访问静态数据成员_interestRate：

```
class Account {
public:
    void raiseInterest( double incr );
    double interest() { return _interestRate; }
private:
    static double _interestRate;
};

inline void Account::raiseInterest( double incr )
{
    _interestRate += incr;
}
```

问题在于，我们必须通过在某个特定的类对象上应用成员访问操作符，才能调用每个成员函数。因为这些成员函数除了静态数据成员_interestRate 之外不访问任何其他的数据成员，所以它们与用哪个对象来调用这个函数无关。这种调用的结果不会访问或修改任何对象（非 static）数据成员。

较好的方案是将这样的成员函数声明为静态成员函数，可以如下实现：

```
class Account {
public:
    static void raiseInterest( double incr );
```

```
    static double interest() { return _interestRate; }
private:
    static double _interestRate;
};

inline void Account::raiseInterest( double incr )
{
    _interestRate += incr;
}
```

静态成员函数的声明除了在类体中的函数声明前加上关键字 static，以及不能声明为 const 或 volatile 之外，与非静态成员函数相同。出现在类体外的函数定义不能指定关键字 static。

静态成员函数没有 this 指针，因此在静态成员函数中隐式或显式地引用这个指针都将导致编译时刻错误。试图访问隐式引用 this 指针的非静态数据成员也会导致编译时刻错误。例如，前面给出的成员函数 dailyReturn()就不能被声明为静态成员函数，因为它访问了非静态数据成员 amount。

我们可以用成员访问操作符点（.）和箭头（->）为一个类对象或指向类对象的指针调用静态成员函数，也可以用限定修饰名直接访问或调用静态成员函数，而无需声明类对象。下面的小程序说明了静态类成员的用法：

```
#include <iostream>
#include "account.h"

bool limitTest( double limit )
{
    // 还没有定义 Account 类对象
    // ok: 调用 static 成员函数
    return limit <= Account::interest();
}

int main() {
    double limit = 0.05;

    if ( limitTest( limit ) )
    {
        // static 类成员的指针被声明为普通指针
        void (*psf)(double) = &Account::raiseInterest;
        psf( 0.0025 );
    }

    Account ac1( 5000, "Asterix" );
    Account ac2( 10000, "Obelix" );
    if ( compareRevenue( ac1, &ac2 ) > 0 )
        cout << ac1.owner()
             << " is richer than "
             << ac2.owner() << "\n";
```

```
        else
            cout << ac1.owner()
                 << " is poorer than "
                 << ac2.owner() << "\n";
        return 0;
    }
```

练习 13.8

已知下面的类 Y，以及它的两个静态数据成员和两个静态成员函数，

```
class X {
public:
    X( int i ) { _val = i; }
    int val() { return _val; }
private:
    int _val;
};

class Y {
public:
    Y( int i );
    static X xval();
    static int callsXval();
private:
    static X _xval;
    static int _callsXval;
};
```

请把_xval 初始化为 20，_callsXval 初始化为 0。

练习 13.9

用练习 13.8 中的类，实现类 Y 的两个静态成员访问函数。callsXval()只是记录 xval()被调用的次数。

练习 13.10

下列静态数据成员的声明和定义哪些是错误的？说明原因。

```
// example.h
class Example {
public:
    static double rate = 6.5;
    static const int vecSize = 20;
    static vector<double> vec(vecSize);
};

// example.C
#include "example.h"
double Example::rate;
vector<double> Example::vec;
```

13.6 指向类成员的指针

假定 Screen 类定义了四个新成员函数——forward()、back()、up()和 down()，它们分别向右、向左、向上和向下移动光标。首先，我们在类体中声明这些新的成员函数：

```
class Screen {
public:
     inline Screen& forward();
     inline Screen& back();
     inline Screen& end();
     inline Screen& up();
     inline Screen& down();
     // 其他成员函数同前
private:
     inline int row();
     // 其他私有成员同前
};
```

成员函数 forward()和 back()每次把光标移动一个字符。在到达屏幕左上角或右下角时光标会折回。

```
inline Screen& Screen::forward()
{ // 向前移动_cursor一个屏幕元素
     ++_cursor;

     // 检查是否到达右下角，若是则折回
     if ( _cursor == _screen.size() )
          home();

     return *this;
}
inline Screen& Screen::back()
{ // 向后移动 _cursor 一个屏幕元素

     // 检查是否到达左上角，若是则折回
     if ( _cursor == 0 )
          end();
     else
          --_cursor;

     return *this;
}
```

end()把光标设置在屏幕的右下角，与以前介绍的成员函数 home()互补：

```
inline Screen& Screen::end()
{
     _cursor = _width * _height - 1;
     return *this;
}
```

up()和 down()把光标上下移动一行。在到达屏幕最上一行或最末行时，光标并不折回，

而是留在原地并发出一个铃声：

```
const char BELL = '\007';

inline Screen& Screen::up()
{ // 移动 _cursor 向上一行
     // 不折回，而是发出铃声
     if ( row() == 1 ) // 到顶了?
          cout << BELL << endl;
     else
          _cursor -= _width;

     return *this;
}

inline Screen& Screen::down()
{
     if ( row() == _height ) // 到底了?
          cout << BELL << endl;
     else
          _cursor += _width;
     return *this;
}
```

row()是一个私有成员函数，它支持 up()和 down()的实现，返回光标位置的当前行：

```
inline int Screen::row()
{ // 返回当前行
     return ( _cursor + _width ) / _width;
}
```

Screen 类的用户要求一个函数 repeat()，它执行用户指定的操作 n 次。它的非通用的实现如下：

```
Screen &repeat( char op, int times )
{
     switch( op ) {
     case DOWN:          // 调用 Screen::down() n 次
          break;
     case UP:            // 调用 Screen::up() n 次
          break;
     // ...
     }
}
```

虽然这种实现能够工作，但是它有许多缺点。一个问题是，它依赖于 Screen 的成员函数保持不变。每次增加或删减一个成员函数，都必须更新 repeat()。第二个问题是它的大小。由于必须测试每个可能的成员函数，所以 repeat()的完整列表非常大而且不必要地复杂。

替换的办法是一种更通用的实现，用 Screen 成员函数的指针类型的参数取代 OP。repeat()不需要再确定用户期望什么样的操作。整个 switch 语句也可以被去掉。类成员指针的用法和定义是下一小节的话题。

13.6.1 类成员的类型

函数指针不能被赋值为成员函数的地址，即使返回类型和参数表完全匹配。例如，下面的 pfi 是一个函数指针，该函数没有参数，返回类型为 int：

```
int (*pfi) ();
```

给出两个全局函数，HeightIs()和 WidthIs()。

```
int HeightIs();
int WidthIs();
```

把 HeightIs()和 WidthIs()中的任何一个或两个赋值给指针 pfi 都是合法且正确的：

```
pfi = HeightIs;
pfi = WidthIs;
```

类 Screen 也定义了两个访问函数——height()和 width()——它们也没有参数，返回类型也是 int：

```
inline int Screen::height() { return _height; }
inline int Screen::width() { return _width; }
```

但是把 height()或 width()任何一个赋给指针 pfi 都是类型违例，都将导致编译时刻错误：

```
// 非法赋值：类型违例
pfi = &Screen::height;
```

为什么会出现类型违例呢？成员函数有一个非成员函数不具有的属性——它的类（its class）。指向成员函数的指针必须与向其赋值的函数类型匹配，不是两个而是三个方面都要匹配：1）参数的类型和个数；2）返回类型；3）它所属的类类型。

在成员函数指针和（普通）函数指针之间的不匹配是由于这两种指针在表示上的区别。函数指针存储函数的地址，可以被用来直接调用那个函数（关于函数指针在 7.9 节讨论）。成员函数指针首先必须被绑定在一个对象或者一个指针上，才能得到被调用对象的 this 指针，然后才调用指针所指的成员函数。（在下一小节，我们将看到成员函数指针怎样被绑定到一个对象或指针上，以便调用一个成员函数。）虽然普通函数指针和成员函数指针都被称作指针，但是它们是不同的事物。

成员函数指针的声明要求扩展的语法，它要考虑类的类型。对指向类数据成员的指针也是这样。考虑 Screen 类的成员 height 的类型。它的完整类型是：“short 型的 Screen 类的成员”。指向_height 的指针的完整类型是“指向 short 型的 Screen 类的成员的指针”。这可以写为：

```
short Screen::*
```

指向 short 型的 Screen 类的成员的指针的定义如下：

```
short Screen::*ps_Screen;
```

ps_Screen 可以用_height 的地址初始化，如下：

```
short Screen::*ps_Screen = &Screen::_height;
```

类似地，它也可以用_width 的地址赋值，如下：

```
ps_Screen = &Screen::_width;
```

ps_Screen 可以被设置为_width 和_height 中的任一个，因为 Screen 的这两数据成员的类型都是 short。

在数据成员指针和普通指针之间的不匹配也是由于这两种指针的表示上的区别。普通指针含有引用一个对象所需的全部信息。数据成员指针在被用来访问数据成员之前，必须先被绑定到一个对象或指针上。（在下一小节，我们将介绍一个数据成员指针怎样被绑定到一个对象或指针上。）（本书的姐妹书 《Inside the C++ Object Model》［LIPPMAN96a］也讨论了成员指针的表示。）

定义一个成员函数指针需要指定函数返回类型、参数表和类。例如，指向 Screen 成员函数并且能够引用成员函数 height()和 width()的指针类型如下：

```
int (Screen::*) ()
```

这种类型指定了一个指向类 Screen 的成员函数的指针，它没有参数，返回值类型为 int。

指向成员函数的指针可被声明、初始化及赋值如下：

```
// 所有指向类成员的指针都可以用 0 赋值
int (Screen::*pmf1)() = 0;
int (Screen::*pmf2)() = &Screen::height;

pmf1 = pmf2;
pmf2 = &Screen::width;
```

使用 typedef 可以使成员指针的语法更易读。例如，下面的类型定义：

```
Screen& ( Screen::* ) ()
```

也就是指向 Screen 类成员函数的一个指针，该函数没有参数，返回 Screen 类对象的引用。它可以被下列 typedef 定义 Action 所取代：

```
typedef Screen& (Screen::*Action)();

Action default = &Screen::home;
Action next = &Screen::forward;
```

指向成员函数类型的指针可以被用来声明函数参数和函数返回类型。我们也可以为成员函数类型的参数指定缺省实参。例如：

```
Screen& action( Screen&, Action );
```

action()被声明为有两个参数：一个 Screen 类对象的引用，和一个指向类 Screen 的成员函数的指针，该函数没有参数，返回 Screen 类对象的引用。action()可以以下列任意方式被调用：

```
Screen myScreen;
typedef Screen& (Screen::*Action)();
Action default = &Screen::home;

extern Screen& action( Screen&, Action = &Screen::display );

void ff()
{
```

```
        action( myScreen );
        action( myScreen, default );
        action( myScreen, &Screen::end );
    }
```

类成员的指针的调用和用法是下一小节的内容。

13.6.2 使用指向类成员的指针

类成员的指针必须总是通过特定的对象或指向该类类型的对象的指针来访问。我们通过使用两个指向成员操作符的指针（针对类对象和引用的.*，以及针对指向类对象的指针的->*）来做到这一点。例如，如下所示，我们通过成员函数的指针调用成员函数：

```
int (Screen::*pmfi)() = &Screen::height;
Screen& (Screen::*pmfS)( const Screen& ) = &Screen::copy;

Screen myScreen, *bufScreen;

// 直接调用成员函数
if ( myScreen.height() == bufScreen->height() )
     bufScreen->copy( myScreen );

// 通过成员指针的等价调用
if ( (myScreen.*pmfi)() == (bufScreen->*pmfi)() )
     (bufScreen->*pmfS)( myScreen );
```

如下调用：

```
(myScreen.*pmfi)()
(bufScreen->*pmfi)()
```

要求有括号，因为调用操作符——()——的优先级高于成员操作符指针的优先级。没有括号，

```
myScreen.*pmfi()
```

将会被解释为：

```
myScreen.*(pmfi())
```

它会先调用函数 pmfi()，把它的返回值与成员对象操作符的指针（.*）绑定。当然 pmfi 的类型不支持这种用法，会产生一个编译时刻错误。

类似地，指向数据成员的指针可以按下列方式被访问：

```
typedef short Screen::*ps_Screen;

Screen myScreen, *tmpScreen = new Screen( 10, 10 );

ps_Screen pH = &Screen::_height;
ps_Screen pW = &Screen::_width;

tmpScreen->*pH = myScreen.*pH;
tmpScreen->*pW = myScreen.*pW;
```

下面是在本节开始时讨论的成员函数 repeat()的实现，它被修改为用一个成员函数的指针作为参数：

```
typedef Screen& (Screen::*Action)();

Screen& Screen::repeat( Action op, int time s )
{
     for ( int i = 0; i < times; ++i )
          (this->*op)();

     return *this;
}
```

参数 op 是一个成员函数的指针，它指向要被调用 times 次的成员函数。若要为 repeat()的参数提供缺省实参，则声明如下：

```
class Screen {
public:
     Screen &repeat( Action = &Screen::forward, int = 1 );
     // ...
};
```

repeat()的调用如下：

```
Screen myScreen;
myScreen.repeat();           // repeat( &Screen::forward, 1 );
myScreen.repeat( &Screen::down, 20 );
```

我们也可以定义指向成员函数的指针表。在下面的例子中，Menu 是为了光标移动而提供的 Screen 成员函数指引表，cursorMovements 是一个枚举类型。它为 Menu 提供一组索引。

```
Action Menu[] = {
     &Screen::home,
     &Screen::forward,
     &Screen::back,
     &Screen::up,
     &Screen::down,
     &Screen::end
};

enum CursorMovements {
     HOME, FORWARD, BACK, UP, DOWN, END
};
```

我们可以定义 move()的重载实例，它接受一个 CursorMovements 参数。用 Menu 表来调用被选中的成员函数，下面是实现：

```
Screen& Screen::move( CursorMovements cm )
{
     ( this->*Menu[ cm ] )();

     return *this;
}
```

下标操作符（[]）的优先级高于指向成员操作符的指针（->*）操作。move()的第一个语句通过索引 Menu 表，选择要被调用的成员函数。然后用 this 指针和指向成员操作符的指针来调用该成员函数。成员函数 move()可以被用在一个交互程序中，用户从一个显示在屏幕上的菜单中选择光标移动动作。

13.6.3 静态类成员的指针

在非静态类成员的指针和静态类成员的指引之间有一个区别。指向类成员的指针语法不能被用来引用类的静态成员。静态类成员是属于该类的全局对象和函数。它们的指针是普通指针。（请记住静态成员函数没有 this 指针。）

指向静态类成员的指针的声明看起来与非类成员的指针相同。解引用该指针不需要类对象。例如，我们再来看一下类 Account：

```
class Account {
public:
    static void raiseInterest( double incr );
    static double interest() { return _interestRate; }
    double amount() { return _amount; }
private:
    static double _interestRate;
    double _amount;
    string _owner;
};

inline void Account::raiseInterest( double incr )
{
    _interestRate += incr;
}
```

&_interestRate 的类型是 double*，而不是：

```
// 不是 &_interestRate 的类型
double Account::*
```

指向_interestRate 的指针定义如下：

```
// OK: 是 double*, 而不是 double Account::*
double *pd = &Account::_interestRate;
```

它被解引用的方式与普通指针一样，不需要相关的类对象。例如：

```
Account unit;

// 用普通的解引用操作符
double daily = *pd /365 * unit._amount;
```

但是，因为_interestRate 和_amount 都是私有成员，所以我们需要使用静态成员函数 interest()和非静态成员函数 amount()。

指向 interest()的指针的类型是一个普通函数指针：

```
// 正确
double (*) ()
```

而不是类 Account 的成员函数的指针：

```
// 不正确
double (Account::*) ()
```

这个指针的定义和对 interest()的间接调用处理方式与非类的指针相同：

```
// ok: douple(*pf) () 不是 double(Account::*pf)()
double (*pf)() = &Account::interest;
```

```
double daily = pf () / 365 * unit.amount();
```

练习 13.11

Screen 类成员_screen 和_cursor 的类型是什么？

练习 13.12

定义一个指向成员的指针，并用 Screen::_screen 的值初始化。再定义一个指向成员的指针，并把 Screen::_cursor 的值赋给它。

练习 13.13

为 Screen 的成员函数的每个不同类型定义一个 typedef。

练习 13.14

指向成员的指针也可以被声明为类的成员。修改 Screen 类的定义，使其含有一个指向 Screen 成员函数的指针，指针的类型与 home()和 end()相同。

练习 13.15

修改现有的 Screen 构造函数（或引入一个新的构造函数），使其含有一个指向成员函数的指针类型的参数，该成员函数的参数表和返回类型与成员函数 home()和 end()相同。为该参数提供一个缺省实参。用这个参数初始化练习 13.14 中引入的数据成员。并且再提供一个 Screen 成员函数，以便允许用户设置这个成员。

练习 13.16

定义一个 repeat()的重载实例，它有一个 cursorMovements 类型的参数。

13.7 联合：一种节省空间的类

联合（union）是一种特殊的类，一个联合中的数据成员在内存中的存储是互相重叠的。每个数据成员都在相同的内存地址开始。分配给联合的存储区数量是“要包含它最大的数据成员”所需的内存数。同一时刻只有一个成员可以被赋给一个值。

我们来看一个例子，说明为什么及怎样使用联合。编译器的词法分析器把用户的程序分成语法单元（token）序列。如下语句：

```
int i = 0;
```

被转换成内含 5 个语法单元的序列：

1．类型关键字 int

2．标识符 i

3．操作符=

4．int 型的常量 0

5．分号

词法分析器把这些语法单元传递给解析器。解析器的第一步是识别它收到的语法单元序列。所提供的信息必须能让解析器识别出这个语法单元序列是一个声明，因此，每个语法单元都有相关的信息，这些信息允许解析器能识别前面的语法单元序列如下：

```
Type ID Assign Constant Semicolon
```

一旦解析器识别出这个语法单元序列是一个声明，接着它就分析每个语法单元的值。在本例中，它判断出：

```
Type <==> int
ID <==> i
Constant <==> 0
```

对于赋值和分号它不需要更多的信息，因为这两个语法单元只有一个可能的值：=和;：

所以，一个语法单元的一个表示可能要用到两个成员——token 和 value。token 是一个唯一的编码，它指定该语法单元为下列之一：Type、ID、Assign、Constant 或者 Semicolon。例如，这个惟一编码可能是一个整数值，用 85 表示 ID，而用 72 表示 Semicolon。value 包含该语法单元的一个特定的值。例如，对前面声明中的语法单元 ID，value 含有字符串 I；对语法单元 Type，value 含有一个 int 类型的表示。

数据成员 value 的表示是一个问题，尽管对于任意给定的语法单元，它只包含一个值，但是 value 可以拥有多种数据类型的值。对语法单元 ID，value 表示字符串，对语法单元 Constant，它表示的是一个整数值。

当然，表示多种数据类型的一种可能方式是用一个类。编译器的作者可以把 value 声明为类类型，对 value 可能表示的数据类型它都含有一个成员。

用类作为 value 的表示解决了这个问题。但是，对于一个给定的语法单元，value 只能是多种可能的数据类型中的一种，它只使用了多个类成员中的一个。但是，类类型为所有的数据成员都保留了空间。比较理想的做法是，这个类一次只维持足够的空间来存放多个可能的数据类型中的一个，而不是维持全部成员的空间。联合正好允许这样。下面是表示这种语祛单元数据类型的 union 的定义：

```
union TokenValue {
    char _cval;
    int _ival;
    char *_sval;
    double _dval;
};
```

如果在 TokenValue 的成员中最大的数据类型是_dval，则 TokenValue 的大小是 double 型对象的大小。缺省情况下，union 的成员都是公有成员。union 的名字可以被用在任何类名可以被使用的地方。例如：

```
// TokenValue 类型的对象
TokenValue last_token;

// TokenValue 类型对象的指针
TokenValue *pt = new TokenValue;
```

union 的成员通过类成员访问操作符（.和->）来访问，就像类成员一样，在操作符前面加上一个 union 对象或指向 union 对象的指针，例如：

```
last_token._ival = 97;
char ch = pt->_cval;
```

union 的成员可以被声明为公有、私有或保护的：

```
union TokenValue {
public:
    char _cval;
    // ...
private:
    int priv;
};

int main() {
    TokenValue tp;
    tp._cval = '\n';  // ok

    // 错误: main() 不能访问私有成员 TokenValue::priv
    tp.priv = 1024;
}
```

union 不能有静态数据成员或是引用成员。如果一个类类型定义了构造函数、析构函数或拷贝赋值操作符，则它不能成为 union 的成员类型。例如：

```
union illegal_members {
    Screen s;          // 错误: 有构造函数
    Screen *ps;        // ok
    static int is;     // 错误: 静态成员
    int &rfi;          // 错误: 引用成员
};
```

我们可以为 union 定义成员函数，包括构造函数和析构函数。

```
union TokenValue {
public:
    TokenValue(int ix) : _ival(ix) { }
    TokenValue(char ch) : _cval(ch) { }
    // ...
    int ival() { return _ival; }
    char cval() { return _cval; }
private:
    int _ival;
    char _cval;
    // ...
};

int main() {
    TokenValue tp(10);
    int ix = tp.ival();
    // ...
}
```

下面给出了怎样使用 union TokenValue 的例子：

```
enum TokenKind { ID, Constant /* 及其他语法单元 */ };
class Token {
public:
     TokenKind tok;
     TokenValue val;
};
```

我们可以如下使用 Token 类型的对象：

```
int lex() {
     Token curToken;
     char *curString;
     int curIval;

     // ...
     case ID: // 标识符
          curToken.tok = ID;
          curToken.val._sval = curString;
          break;
     case Constant: // 整数常量
          curToken.tok = Constant;
          curToken.val._ival = curIval;
          break;

     // ... etc.
}
```

使用 union 的危险是，通过一个不适当的数据成员意外地获取到当前存储在 union 中的值。例如，如果最后一次赋值是针对_ival 的，则程序员不能通过成员_sval 获取该值。这样做一定会导致程序错误。

为防止这样的错误，程序员应该定义一个额外的对象，来跟踪当前被存储在 union 中的值的类现 这个额外的对象被称为 union 的判别式（discriminant）。就是类 Token 中的成员 tok 的角色。例如：

```
char *idVal;

// 在引用 sval 之前检查判别式的值
if ( curToken.tok == ID )
     idVal = curToken.val._sval;
```

一个比较好的经验是，在处理作为类成员的 union 对象时，为所有 union 数据类型提供一组访问函数。例如：

```
#include <cassert>

// union 成员 sval 的访问函数
string Token::sval() {
     assert( tok==ID );
     return val._sval;
}
```

在定义 union 时，union 的名字是可选的。如果在程序中不需要用 union 的名字作为类型名去声明其他的对象，则定义 union 类型时就没必要提供名字了。例如，下面的 Token 的定义与前面的定义等价。惟一的区别是这个 union 没有名字：

```
class Token {
public:
     TokenKind tok;

     // union 类型名被省略
     union {
          char _cval;
          int _ival;
          char *_sval;
          double _dval;
     } val;
};
```

有一种特殊的 union 实例被称为匿名 union（anonymous union）。匿名 union 是没有名字的 union，它后面也没有跟着对象定义。例如，下面的 Token 类定义含有一个匿名 union：

```
class Token {
public:
     TokenKind tok;
     // 匿名 union
     union {
          char _cval;
          int _ival;
          char *_sval;
          double _dval;
     };
};
```

匿名 union 的数据成员可以在定义匿名 union 的域中被直接访问。例如，下面是重新编码的 lex()函数，它用到了含有匿名 union 的 Token 类定义：

```
int lex() {
     Token curToken;
     char *curString;
     int curIval;

     // ... 确定语法单元
     // ... 设置 curToken
     case ID:
          curToken.tok = ID;
          curToken._sval = curString;
          break;
     case Constant:    // 整数常量
          curToken.tok = Constant;
          curToken._ival = curIval;
          break;

     // ... etc.
}
```

匿名 union 去掉了一层成员访问操作符，因为 union 的成员名可以像 Token 类的成员一样被访问。所以匿名 union 不能有私有或保护的成员，也不能定义成员函数。在全局域中定义的匿名 union 必须被声明在未命名的名字空间中（或者被声明为 static）。

13.8 位域（bit-field）：一种节省空间的成员

有一种被称为**位域**（bit-field）的特殊的类数据成员，它可以被声明用来存放特定数目的位。位域必须是有序数据类型。它可以有符号，也可以无符号。例如：

```
class File {
    // ...
    unsigned int modified : 1; // 位域 (bit-field)
};
```

位域标识符后面跟有一个冒号，然后是一个常量表达式指定位数。例如，modified 是一个只有一位构成的位域。

在类体中相邻定义的位域，如果可能的话，它们会被放在同一个整数的连续位中，并以此提供空间压缩。例如，在下列声明中，5 个位域被存储在单个 unsigned int 中，它首先与位域 mode 相关联。

```
typedef unsigned int Bit;
class File {
public:
    Bit mode: 2;
    Bit modified: 1;
    Bit prot_owner: 3;
    Bit prot_group: 3;
    Bit prot_world: 3;
    // ...
};
```

对于位域的访问方式与其他类数据成员相同。例如，类的私有位域只能在类的成员函数和友元中被访问：

```
void File::write()
{
    modified = 1;
    // ...
}

void File::close()
{
    if ( modified )
        // ... 内容从略
}
```

下面的例子说明了怎样使用大于 1 位的位域（例子中用到了 4.4 节讨论的按位操作符）

```
enum { READ = 01, WRITE = 02 }; // 文件模式

int main() {
    File myFile;
    myFile.mode |= READ;

    if ( myFile.mode & READ )
        cout << "myFile.mode is set to READ\n";
```

```
}
```

通常情况下，我们会定义一组 inline 成员函数，来测试每个位域成员的值。例如，类 File 可以定义成员 isRead()和 isWrite()。

```
inline int File::isRead() { return mode & READ; }
inline int File::isWrite() { return mode & WRITE; }

if ( myFile.isRead() ) /* ... */
```

有了这些成员函数，现在位域可以被声明为类 File 的私有成员。

由于取地址操作符（&）不能被应用在位域上，所以也没有能指向类的位域的指针。位域也不能是类的静态成员。

C++标准库提供了一个 bitset 类模板，它可以辅助操纵位的集合。在可能的情况下，应尽可能使用它来取代位域。bitset 类模板及其操作在 4.12 节给出。

练习 13.17

请改写本节中的例子，使类 File 使用 4.12 节给出的 bitset 类及其操作符，而不是直接声明和操纵位域数据成员。

13.9 类域 ※

类体就定义了一个域。在类体中，每一个类成员的声明都向它的类域中引入了一个成员名。

成员访问操作符（点和箭头）以及域解析操作符（::）可以被用在程序中，来访问类域中声明的成员。使用点和箭头操作符时，在操作符前面的名字给出了一个类类型的对象或指向一个类对象的指针，对于操作符后面的成员名，编译器将在这个类的类域中查找。类似地，在使用域解析操作符时，对于操作符后面的名字，编译器要在“操作符前面的名字指定的类的类域”中查找。（在第 17 章和第 18 章我们将看到派生类也可以引用其基类的成员。）

我们并不总是需要用成员访问操作符或域解析操作符来引用类的成员。某些程序代码本身就在类域中，这些程序部分可以直接访问类成员。位于类域内的第一部分程序文本就是类定义本身。类成员名可被用在类体中其声明之后。例如：

```
class String {
public:
    typedef int index_type;

    // 参数类型引用 String::index_type
    char& operator[]( index_type );
};
```

类成员在类体中被声明的顺序很重要。在类体中，后声明的成员不能被先声明的成员声明使用，例如，如果成员 operator[]()的声明出现在 typedef index_type 声明之前，则 operator[]()的声明就是错的，因为它使用了未声明的名字 index_type：

```
class String {
public:
```

```
        // 错误：名字 index_type 还没有被声明
        char& operator[]( index_type );
        typedef int index_type;
};
```

在类定义中用到的名字必须在使用前首先被声明，这个规则有两种例外的情况。第一个例外是对于被用在 inline 成员函数定义中的名字，第二个例外是于被用作缺省实参的名字。我们来依次了解每一种情况。

对于用在 inline 成员函数定义中的名字的解析有两步。首先函数声明（即函数返回类型和参数表）在其所在的类定义中出现的位置被处理，然后函数体在完整的类域中被处理——所有的成员的声明都被看到。我们来看一个例子，它在类体中定义了 inline 成员 operator[]()：

```
class String {
public:
        typedef int index_type;
        char& operator[] ( index_type elem )
            {return _string[ elem ]; }
private:
        char * _string;
};
```

名字解析的第一步是查找在成员 operator[]()声明中被用到的名字。在这第一步中，参数类型名 index_type 首先被检查。因为第一步发生在类体中遇到成员函数声明时，所以名字 index_type 必须在成员 operator[]()的定义之前被声明。

注意，成员_string 是在类体中 operator[]()的定义之后被声明的。这是可以的，在 operator[]()函数体中，_string 不是一个未声明的名字。成员函数体内的名字是在“inline 成员函数定义的名字解析的第二步”中被检查的。该名字解析发生在类的完整域中。就好像成员函数体是在类体结束前被最后处理一样。在这个点上，类的所有成员都已经被声明了。

缺省实参也是在名字解析的第二步——在类的完整域中被解析。例如，成员函数 clear()的声明使用了定义在后面的静态成员 bkground 的名字：

```
class Screen {
public:
        // bkground 指向在类定义中后来声明的静态成员
        Screen& clear( char = bkground );
private:
        static const char bkground = '#';
};
```

尽管在成员函数声明中的缺省实参是在类的完整域中被解析的，但是如果缺省实参引用了一个非静态成员，则程序仍然是错的。非静态数据成员在它的值被程序使用之前，必须先被绑定到该类类型的对象上，或绑定到指向该类类型的对象的指针上。把非静态数据成员用作缺省实参违背了这个限制。例如，如果把前面的例子改写为：

```
class Screen {
public:
        // ...

        // 错误：bkground 是一个非 static 成员
        Screen& clear( char = bkground );
```

```
private:
    const char bkground;
};
```

缺省实参名被解析为非静态数据成员 bkground，因此，这个缺省实参是错的。

出现在类体外的类成员定义是该类域中另一部分程序文本（与类定义相比）。对于这部分程序文本，即使不通过成员访问操作符或域解析操作符，也能找到类成员的名字，我们来看一下这些成员定义中名字解析过程是怎样进行的。

一般地，如果类成员的定义出现在类体之外，则跟在被定义的成员名后面的程序，直到该成员定义结束，都被认为是在类域之中。例如，我们把 operator[]()的定义移到类 String 之外：

```
class String {
public:
    typedef int index_type;
    char& operator[]( index_type );
private:
    char *_string;
};

// operator[]() 访问 index_type 和 _string
inline char& String::operator[]( index_type elem )
{
    return _string[ elem ];
}
```

注意，参数表直接引用成员 typedef index_type，而没有用 string::限定修饰这个名字。跟在成员名 string::operator[]()之后的程序文本，直到该成员函数定义结束，都在类域中。在类 String 域中声明的类型都被考虑用来解析成员函数参数表中的类型名。

静态数据成员的定义也可以出现在类定义之外。在这些定义中，在被定义的静态成员名之后的程序文本，直到成员定义结束，也都被认为是在类域中、例如，静态成员的初始值可以直接引用类成员，而无需使用点、箭头或域解析操作符：

```
class Account {
    // ...
private:
    static double _interestRate;
    static double initInterest();
};

// 引用 Account::initInterest()
double Account::_interestRate = initInterest();
```

interestRate 的初始值调用了静态成员函数 Account::initInterest()，而没有用限定修饰的名字来引用 initInterest()。

不仅是初始值，在静态成员 interest 名的后面，直到结束静态成员定义的分号，所有东西都在 Account 类的域中。于是，如下所示，静态成员 name 的定义也可以引用类成员

```
nameSize:
class Account {
// ...
```

```
private:
    static const int nameSize = 16;
    static const char name[nameSize];
};

// nameSize 没有被 Account 限定修饰
const char Account::name[nameSize] = "Savings Account";
```

即使成员 nameSize 没有用类名 Account 限定修饰，name 的定义仍然没有错误。静态数据成员 name 的定义是在其类的域内，在限定修饰的名字 Account::name 可见之后，它就可以引用类 Account 的成员。

在类体之外的类成员定义中，在被定义的成员名字之前的程序文本，不在该类的域内。如果这些程序文本要引用类成员，则必须使用域解析操作符。例如，如果静态成员的类型是类 Account 的 typedef 成员 Money，则在类体之外定义静态数据成员时，名字 Money 必须被限定修饰：

```
class Account {
    typedef double Money;
    // ...
private:
    static Money _interestRate;
    static Money initInterest();
};
// Money 必须用 Account:: 限定修饰
Account::Money Account::_interestRate = initInterest();
```

每个类维护自己相关联的域，两个不同的类有两个不同的类域。一般地，一个类的成员不能直接被用在其他类的成员定义中，除非一个类是另一个类的基类。关于继承和基类的介绍请参见第 17 章和 18 章。

13.9.1 类域中的名字解析

当然，用在类域中的名字不见得总是类的成员名字。在类域中的名字解析过程也能找到在其他域中声明的名字。在名字解析期间，如果在类域中，一个用到的名字不能被解析成一个类成员名，则系统就会在包含这个类或成员定义的域中查找该名字的声明。在本小节，我们将了解怎样解析在类域中被用到的名字。

用在类定义中的名字（除了在 inline 成员函数定义中的名字和缺省实参的名字）其解析过程如下：

1．在名字使用之前出现的类成员的声明应予以考虑。

2．如果步骤 1 的解析没有成功，则在类定义之前的名字空间域中出现的声明应予以考虑。（记住，全局域也是一个名字空间域。）（8.5 节介绍了名字空间。）例如：

```
typedef double Money;
class Account {
    // ...
private:
    static Money _interestRate;
    static Money initInterest();
```

```
    // ...
};
```

编译器首先在类 Account 的域中查找 Money 的声明，它只考虑在使用 Money 之前的成员声明。因为没有找到这样的成员声明，所以编译器接着在全局域中查找 Money 的声明。又因为只有在类 Account 定义之前的声明才会被考虑，所以最终找到了全局 typedef Money 的声明，它是在_interestRate 和 initInterest()的声明中被使用的类型。

被用在类成员函数定义中的名字的解析过程如下：

1．在成员函数局部域中的声明首先被考虑（局部域和局部声明在 8.1 节介绍）。

2．如果在步骤 1 中的解析不成功，则考虑所有的类成员声明。

3．如果在步骤 2 中的解析不成功，则考虑在成员函数定义之前的名字空间域中出现的声明。

我们来看一看，用在 inline 成员函数体内的名字是怎样被解析的：

```
int _height;

class Screen {
public:
    Screen( int _height ) {
        _height = 0;          // 哪一个 _height? 参数
    }
private:
    short _height;
};
```

为了查找 Screen 构造函数中的名字_height 的声明，编译器首先在构造函数的局部域内查找。函数的参数是在该函数的局部域内被声明的。在构造函数定义中用到的名字_height 引用了这个参数声明。

如果这个参数声明没有被找到的话，则编译器接着会查看类域，考虑 Screen 的所有成员声明，这样就会找到成员_height 的声明。我们称，类成员_height 的声明被构造函数的参数声明隐藏起来。即使这个类成员被隐藏，我们仍然可以通过用类名或 this 指针显式地限定修饰这个成员名，以便在构造函数中使用它。例如：

```
int _height;

class Screen {
public:
    Screen( long _height ) {
        this->_height = 0;    // 指向 Screen::_height
        // 这样也有效
        // Screen::_height = 0;
    }
private:
    short _height;
};
```

假设参数声明和成员声明都没有被找到，那么编译器就会在外围的名字空间中查找。在本例中，考虑在 Screen 类定义之前让全局域中可见的声明。这样就会找到全局对象_height 的声明。我们称，全局对象_height 被类成员声明所隐藏。即使全局对象被隐藏，我们仍然可

以通过使用全局域解析操作符限定修饰全局名字，以便在构造函数中使用它：

```
int _height;
class Screen {
public:
     Screen( long _height ) {
          ::_height = 0;  // 指向全局对象
     }
private:
     short _height;
};
```

如果构造函数在类定义之外被声明，则名字解析的第 3 步不但要考虑出现在 Screen 类定义之前的全局域中的声明，而且还要考虑出现在成员函数定义之前的全局域中的声明。例如

```
class Screen {
public:
     // ...
     void setHeight( int );
private:
     short _height;
};

int verify(int);

void Screen::setHeight( int var ) {
     // var: 指向参数
     // _height: 指向类成员
     // verify: 指向全局函数
     _height = verify( var );
}
```

我们注意到，全局函数 verify()的声明在类 Screen 定义之前是不可见的。但是，名字解析的第 3 步要考虑在成员定义之前可见的名字空间域中的声明，所以找到了全局函数 verify()的声明。

用在类静态成员定义中的名字解析过程如下：

1．考虑所有类成员的声明。

2．如果第 1 步失败，则考虑在静态成员定义之前的名字空间域中出现的声明。

在第 2 步期间，编译器考虑出现在静态数据成员定义之前的名字空间域中的声明，而不只是出现在类定义之前的声明。

练习 13.18

类域中的程序文本包括哪些部分？请列出来。

练习 13.19

对于类域中的哪些程序文本，需要考虑该类的完整类域（即考虑所有在类体中被声明的成员）？请列出来。

练习 13.20

当名字 Type 被用在类 Exercise 和成员函数 setVal()的定义中时，它引用的是哪个声明？（记住，不同的用法可能引用不同的声明。）当名字 initVal 被用在成员函数 setVal()的定义中时，它引用的是哪个声明？

```
typedef int Type;
Type initVal();
class Exercise {
public:
     // ...
     typedef double Type;
     Type setVal( Type );
     Type initVal();
private:
     int val;
};

Type Exercise::setVal( Type parm ) {
     val = parm + initVal();
}
```

成员函数 setVal()的定义是错误的。你能看出为什么吗？请做出必要的修改，以使类 Exercise 使用全局 typedef Type 和全局函数 initVal()。

13.10 嵌套类

一个类可以在另一个类中定义，这样的类被称为嵌套类（nested class）。嵌套类是其外围类的一个成员。嵌套类的定义可以出现在其外围类的公有、私有或保护区中。

嵌套类的名字在其外围类域中是可见的，但是在其他类域或名字空间中是不可见的。这意味着，嵌套类的名字不会与外围域中声明的相同名字冲突。例如：

```
class Node { /* ... */ };
class Tree {
public:
     // Node 被封装在 Tree 的域中
     // 在类域中 Tree::Node 隐藏了 ::Node
     class Node {...};

     // ok: 被解析为嵌套类: Tree::Node
     Node *tree;
};

// Tree::Node 在全局域中不可见
// Node 被解析为全局的 Node 声明
Node *pnode;
class List {
public:
     // Node 被封装在 List 的域中
```

```
        // 在类域 List::Node 中隐藏了 ::Node
        class Node {...};

        // ok: 解析为: List::Node
        Node *list;
    };
```

与非嵌套类一样，嵌套类可以有与自身同样类型的成员：

```
    // Not ideal configuration: evolving class definition
    class List {
    public:
        class ListItem {
            friend class List;          // 友元声明
            ListItem( int val = 0 );    // 构造函数
            ListItem *next;             // 指向自己类的指针
            int value;
        };
        // ...
    private:
        ListItem *list;
        ListItem *at_end;
    };
```

私有成员是指这样的成员，它只能在该类的成员或友元定义中被访问。除非外围类被声明为嵌套类的友元，否则它没有权利访问嵌套类的私有成员。这就是为什么 ListItem 把 List 声明为友元的原因：为了允许类 List 的成员定义访问 ListItem 的私有成员。嵌套类也没有任何特权访问其外围类的私有成员。如果我们想授权 ListItem，允许它访问类 List 的私有成员，那么该外围类 List 必须把嵌套类 ListItem 声明为友元。在前面的例子中 ListItem 不是 List 的友元，所以它不能引用 List 的私有成员。

把类 ListItem 声明为 List 类的公有成员意味着，该嵌套类可以在整个程序中（在类 List 的友元和成员定义之外）用作类型。例如：

```
    // ok: 全局域中的声明
    List::ListItem *headptr;
```

这超出了我们的本意。嵌套类 ListItem 支持 List 类的抽象，我们不希望让 ListItem 类型在整个程序中都可以被访问。那么，比较好的设计是把 ListItem 嵌套类定义为类 List 的私有成员：

```
    // 不理想的配置：要改进的类定义
    class List {
    public:
        // ...
    private:
        class ListItem {
            // ...
        };
        ListItem *list;
        ListItem *at_end;
    };
```

现在，只有 List 的成员和友元的定义可以访问类型 ListItem。把类 ListItem 的所有成员

都声明为公有的也不再有任何坏处。因为 ListItem 类是 List 的私有成员，所以只有 List 类的友元和成员可以访问 ListItem 的成员。有了这个新的设计，我们就不再需要友元声明了。下面是类 List 的新定义：

```
// 较好的设计!
class List {
public:
     // ...
private:
     // 现在 ListItem 是一个私有的嵌套类型
     class ListItem {
     // 它的成员都是公有的
     public:
          ListItem( int val = 0 );
          ListItem *next;
          int value;
     };
     ListItem *list;
     ListItem *at_end;
};
```

在类 ListItem 的定义中没有把构造函数定义为 inline（内联的），构造的数必须在类定义之外被定义。在哪儿可以定义它呢？ListItem 的构造函数不是类 List 的成员，所以我们不能在类 List 的体内定义。ListItem 的构造函数必须被定义在全局域中——该域含有其外围类的定义。当我们没有在嵌套类体内以 inline 形式定义嵌套类的成员函数时，我们就必须在最外围的类之外定义这些成员函数。

下面是 ListItem 构造函数的一种可能的定义。但是，对于全局域定义的语法来说这是不正确的：

```
class List {
public:
     // ...
private:
     class ListItem {
     public:
          ListItem( int val = 0 );
          // ...
     };
};
// 错误: ListItem 不在全局域中
ListItem::ListItem( int val ) { ... }
```

问题在于，名字 ListItem 在全局域中是不可见的。在全局域中使用 ListItem 必须指明 ListItem 是类 List 中嵌套的类。可以通过用其外围类名 List 限定修饰类名 ListItem 来做到这一点。下面是正确的语法：

```
// 用外围类名限定修饰嵌套类名
List::ListItem::ListItem( int val ) {
     value = val;
     next = 0;
}
```

注意，只有嵌套类名是限定修饰的。第一个限定修饰符 List::指外围类，它限定修饰其后的名字——嵌套类 ListItem。第二个 ListItem 是指构造函数而不是嵌套类。下列定义中的成员名字是不正确的：

```
// 错误：构造函数名是 ListItem 而不是 List::ListItem
List::ListItem::List::ListItem( int val ) {
    value = val;
    next = 0;
}
```

如果 ListItem 已经声明了一个静态成员，那么它的定义也要放在全局域中。在这样的定义中，静态成员名看起来如下所示：

```
int List::ListItem::static_mem = 1024;
```

注意，对于成员函数和静态数据成员而言，不一定只有嵌套类的公有成员，才能在类定义之外被定义。类 ListItem 的私有成员也可以被定义在全局域中。

嵌套类也可以被定义在其外围类之外。例如。Lisiltem 的定义也可以在全局域中被给出，如下：

```
class List {
public:
    // ...
private:
    // 这个声明是必需的
    class ListItem;
    ListItem *list;
    ListItem *at_end;
};
// 用外围类名限定修饰嵌套类名
class List::ListItem {
public:
    ListItem( int val = 0 );
    ListItem *next;
    int value;
};
```

在全局定义中，嵌套类 ListItem 的名字必须由其外围类 List 的名字限定修饰。注意，在类 List 体内的 ListItem 的声明不能省略。如果嵌套类没有先被声明为其外围类的一个成员，则全局域中的定义不能被指定为嵌套类。在全局域中定义的嵌套类不一定是其外围类的公有成员。

在嵌套类的定义被看到之前，我们只能声明嵌套类的指针和引用。即使类 ListItem 是在全局域中被定义的，List 的数据成员 list 和 at_end 仍然是有效的，因为这两个成员都是指针。如果这两个成员中有一个是对象而不是指针，那么类 List 的成员声明将会引发一个编译错误。例如：

```
class List {
public:
    // ...
private:
    // 这个声明是必需的
```

```
        class ListItem;
        ListItem *list;
        ListItem at_end;  // 错误：未定义嵌套类 ListItem
};
```

为什么会希望在类定义之外定义嵌套类呢？或许嵌套类支持外围类的实现细节，我们不想让 List 类的用户看到 ListItem 的细节。因此，我们不愿把嵌套类的定义放在含有 List 类接口的头文件中。于是，我们只能在含有 List 类及其成员实现的文本文件中给出嵌套类 ListItem 的定义。

嵌套类可以先被声明，然后再在外围类体中被定义。这允许多个嵌套类具有互相引用的成员。例如：

```
class List {
public:
    // ...
private:
    // List::ListItem 的声明
    class ListItem;
    class Ref {
        ListItem *pli;       // pli 类型为：List::ListItem*
    };

    // List::ListItem 的定义
    class ListItem {
        Ref *pref;           // pref 的类型为：List::Ref*
    };
};
```

如果类 ListItem 没有在类 Ref 之前先被声明，那么成员 pli 的声明就是错的，因为名字 ListItem 没有被声明。

嵌套类不能直接访问其外围类的非静态成员，即使这些成员是公有的，任何对外围类的非静态成员的访问都要求通过外围类的指针、引用或对象来完成。例如：

```
class List {
public:
    int init( int );
private:
    class ListItem {
    public:
        ListItem( int val = 0 );
        void mf( const List & );
        int value;
        int memb;
    };
};
List::ListItem::ListItem( int val )
{
    // List::init() 是类 List 的非静态成员
    // 必须通过 List 类型的对象或指针来使用
    value = init( val );    // 错误：非法使用 init
}
```

使用类的非静态成员时，编译器必须能够识别出非静态成员属于哪个对象。在类 ListItem 的成员函数中，this 指针只能被隐式地应用在类 ListItem 的成员上，而不是外围类的成员上。由于隐式的 this 指针，我们知道数据成员 value 指向被调用构造函数的对象。在 ListItem 的构造函数中的 this 指针的类型是 ListItem*。而要访问成员 init()所需的是 List 类型的对象或 List*类型的指针。

下面是成员函数 mf()通用引用参数引用 init()。从这里我们能够知道，成员 init()是针对函数实参指定的对象而被调用的：

```
void List::ListItem::mf( const List &il ) {
    memb = il.init(); // ok: 通过引用调用 init()
}
```

尽管访问外围类的非静态数据成员需要通过对象，指针或引用才能完成，但是嵌套类可以直接访问外围类的静态成员、类型名、枚举值（假定这些成员是公有的）。类型名是一个 typedef 名字、枚举类型名、或是一个类名。例如：

```
class List {
public:
    typedef int (*pFunc)();
    enum ListStatus { Good, Empty, Corrupted };
    // ...
private:
    class ListItem {
    public:
        void check_status();
        ListStatus status; // ok
        pFunc action;// ok
        // ...
    };
    // ...
};
```

pFunc、ListStatus 和 ListItem 都是外围类 List 的域内部的嵌套类型名。这三个名字以及 ListStatus 的枚举值都可以被用在 ListItem 的域中，这些成员可以不加限定修饰地被引用：

```
void List::ListItem::check_status()
{
    ListStatus s = status;
    switch ( s ) {
        case Empty: ...
        case Corrupted: ...
        case Good: ...
    }
}
```

在 ListItem 的域之外，以及在外围类 List 域之外引用外围类的静态成员、类型名和枚举名都要求域解析操作符。例如：

```
List::pFunc myAction;  // ok
List::ListStatus stat = List::Empty;   // ok
```

当引用一个枚举值时，我们不能写：

```
List::ListStatus::Empty
```

这是因为枚举值可以在定义枚举的域内被直接访问。为什么？因为枚举定义并不像类定义一样维护了自己相关的域。

13.10.1 在嵌套类域中的名字解析

让我们来看看在联套类的定义，及其成员定义中的名字解析是怎样进行的。

被用在嵌套类的定义中的名字（除了 inline 成员函数定义中的名字和缺省实参的名字之外）其解析过程如下：

1．考虑出现在名字使用点之前的嵌套类的成员声明。

2．如果第 1 步没有成功，则考虑出现在名字使用点之前的外围类的成员声明。

3．如果第 2 步没有成功，则考虑出现在嵌套类定义之前的名字空间域中的声明。

例如：

```
enum ListStatus { Good, Empty, Corrupted };
class List {
public:
     // ...
private:
     class ListItem {
     public:
          // 查找：
          // 1) 在 List::ListItem 中
          // 2) 在 List 中
          // 3) 在全局域中
          ListStatus status; // 引用全局枚举
          // ...
     };
     // ...
};
```

编译器首先在类 ListItem 的域中查找 ListStatus 的声明。因为没有找到成员声明，所以编译器接着在类 List 的域中查找 ListStatus 的声明。因为在 List 类中也没有找到声明，于是编译器在全局域中查找 ListStatus 的声明。在这三个域中，只有位于 ListStatus 使用点之前的声明才会被编译器考虑。编译器找到了全局枚举 ListStatus 的声明，它是被用在 Status 声明中的类型。

如果在全局域中。在外围域 List 之外定义嵌套类 ListItem，则 List 类的所有成员都已经被声明完毕，因而编译器将考虑其所有声明：

```
class List {
private:
     class ListItem;
     // ...
public:
     enum ListStatus { Good, Empty, Corrupted };
     // ...
};
class List::ListItem {
public:
     // 查找：
```

```
        // 1) 在 List::ListItem 中
        // 2) 在 List 中
        // 3) 在全局域中
        ListStatus status;  // List::ListStatus
        // ...
};
```

ListItem 的名字解析过程首先在类 ListItem 的域中开始查找。因为没有找到成员声明，所以编译器在类 List 的域内查找 ListStatus 的声明。因为类 List 的完整定义都已经能够看得到，所以这一步查找考虑 List 的所有成员。于是找到 List 中嵌套的 enumListStatus，尽管它是在 ListItem 之后被声明的。status 是 List 的 ListStatus 类型的一个枚举对象。如果 List 没有名为 ListStatus 的成员，则名字查找过程会在全局域中。在嵌套类 ListItem 定义之前查找声明。

被用在嵌套类的成员函数定义中的名字，其解析过程如下：

1．首先考虑在成员函数局部域中的声明。

2．如果第 1 步没有成功，则考虑所有嵌套类成员的声明。

3．如果第 2 步没有成功，则考虑所有外围类成员的声明。

4，如果第 3 步没有成功，则考虑在成员函数定义之前的名字空间域中出现的声明。

在下面的代码段中，成员函数 check_status()定义中的 list 引用了哪个声明？

```
class List {
public:
    enum ListStatus { Good, Empty, Corrupted };
    // ...
private:
    class ListItem {
    public:
        void check_status();
        ListStatus status;    // ok
        // ...
    };
    ListItem *list;
    // ...
};
int list = 0;
void List::ListItem::check_status()
{
    int value = list;          // 哪个 list?
}
```

很有可能程序员想让 check_status()中的 List 引用全局对象：

- value 和全局对象 List 的类型都是 int。List::list 成员是指针类型，在没有显式转换的情况它不能被赋给 value。
- 不允许 ListItem 访问其外围类的私有数据成员，如 List。
- list 是一个非静态数据成员，在 ListItem 的成员函数中必须通过对象、指针或引用来访问它。

但是，尽管有这些原因，在成员 check_status()中用到的名字 List 仍被解析为类 List 的数据成员 list。记住，如果在联套类 ListItem 的域中没有找到该名字；则在查找全局域之前，下

一个要查找的是其外围类的域。外围类 List 的成员 list 隐藏了全局域中的对象。于是产生一个错误消息，因为在 check_status()中使用指针 list 是无效的。

只有在名字解析成功之后，编译器才会检查访问许可和类型兼容性。如果名字的用法本身就是错误的，则名字解析过程将不会再去查找更适合于该名字用法的声明，而是产生一个错误消息。

为了访问全局对象 list，必须使用全局域解析操作符：

```
void List::ListItem:: check_status() {
    value = ::list;   // ok
}
```

如果成员函数 check_status()被定义成位于 ListItem 类体中的内联函数．则上面所讲到的最后一步修改会使编译器产生一个错误消息，报告说全局域中的 list 没有被声明。

```
class List {
public:
    // ...
private:
    class ListItem {
    public:
        // 错误：没有可见的 ::list 声明
        void check_status() { int value = ::list; }
        // ...
    };
    ListItem *list;
    // ...
};
int list = 0;
```

全局对象 list 是在类 List 定义之后被声明的，对于在类体中内联定义的成员函数，只考虑在外围类定义之前可见的全局声明。如果 check_status()的定义出现在 List 的定义之后，则编译器考虑在 check_status()定义之前可见的全局声明，于是找到对象 list 的全局声明。

练习 13.21

第 11 章有一个使用类 iStack 的运行示例，请修改这个例子，把异常类 pushOnFull 和 popOnEmpty 声明为类 iStack 的公有嵌套类。修改第 11 章给出的类 iStack 的定义和它的成员函数定义，以及 main()的定义来引用这些嵌套类。

13.11 作为名字空间成员的类 ※

目前给出的在名字空间中的类都是在全局名字空间域中定义的类。类也可以被定义在用户声明的名字空间中。在用户声明的名字空间中定义的类名只在该名字空间的域中可见，而在全局域或其他名字空间中不可见。这意味着，该类名不会与在其他名字空间中声明的名字冲突。例如：

```
namespace cplusplus_pri mer {
    class Node { /* ... */ };
}
```

```
namespace DisneyFeatureAnimation {
    class Node { /* ... */ };
}
Node *pnode;        // 错误: Node 在全局域中不可见

// OK: 声明 nodeObj 的类型为 DisneyFeatureAnimation::Node
DisneyFeatureAnimation::Node nodeObj;

// using 声明: 使得 node 在全局域中可见
using cplusplus_primer::Node;
Node another;       // cplusplus_primer::Node
```

正如前两节所说明的，类成员（即成员函数、静态数据成员或者嵌套类）可以在其类体外被定义。作为库的实现者，如果我们把自己的类定义放在一个用户声明的名字空间中，那么我们又可以把在类体外声明的类成员的定义放在哪儿？这些声明可以被放在包含最外围类定义的名字空间中，或者它的某一个外围名字空间中。这允许我们按下面的方式组织库中的代码：

```
// --- primer.h ---
namespace cplusplus_primer {
    class List {
        // ...
    private:
        class ListItem {
        public:
            void check_status();
            int action();
            // ...
        };
    };
}

// --- primer.C ---
#include "primer.h"
namespace cplusplus_primer {
    // ok: check_status() 在与 List 相同的名字空间中定义
    void List::ListItem::check_status() { }
}
// ok: action() 在全局域中定义,
// 在一个包含类 List 定义的名字空间中
// 成员名用名字空间名限定修饰
int cplusplus_primer::List::ListItem::action() { }
```

嵌套类 ListItem 的成员可以被定义在名字空间 cplusplus_primer（它含有类 List 的定义）中，或者被定义在全局名字空间（它包含名字空间 cplusplus_primer 的定义）中。在这两种情况下，定义中的成员名字都必须用其外围类名适当地限定修饰，如果是在其外围的用户声明的名字空间外面被声明的，则还要用该名字空间名限定修饰。

在用户声明的名字空间里出现的成员定义中，名字解析是怎样进行的呢？例如，在 action() 定义中的名字解析过程怎样找到 someVal 的声明？

```
int cplusplus_primer::List::ListItem::action() {
    int local = someVal;
    // ...
}
```

成员函数定义中的局部域首先被检查，然后是类 ListItem 的完整域，接着查找的是类 List 的完整域。到目前为止，这些都与 13.10 节描述的名字解析过程相同。然后，名字空间 cplusplus_primer 中的声明被检查，最后，全局域中的声明被检查。当考虑名字空间 cplusplus_primer 或全局域中的声明时，只考虑位于成员函数 action()定义之前的声明。例如：

```
// --- primer.h ---
namespace cplusplus_primer {
    class List {
        // ...
    private:
        class ListItem {
    public:
        int action();
        // ...
    };
    };
    const int someVal = 365;
}

// --- primer.C ---
#include "primer.h"

namespace cplusplus_primer {
    int List::ListItem::action() {
        // ok: cplusplus_primer::someVal
        int local = someVal;
        // 错误: calc() 还没有声明
        double result = calc( local );
        // ...
    }
    double calc(int) { }
    // ...
}
```

名字空间 cplusplus_primer 的定义不是连续的。类 List 和对象 someVal 的定义出现在名字空间定义的第一部分中，它被放在头文件 primer.h 中。函数 calc()的定义出现在实现文件 primer.C 提供的名字空间定义中。在 action()中使用 calc()是错误的，因为它是在 action()定义中的使用点之后被声明的。如果 cllc()是名字空间 cplusplus_primer 接口的一部分，则它应该在头文件中出现的名字空间部分被声明，如下所示：

```
// --- primer.h ---
namespace cplusplus_primer {
    class List {
        // ...
    };
    const int someVal = 365;
```

```
        double calc(int);
    }
```

否则，如果 calc()只被用在 action()中以辅助它的实现，而不是名字空间接口的一部分，那么它必须在 action()之前被声明，以便可以在 action()定义中引用它。

这与我们在前面的章节中看到的在全局域中查找声明类似：只考虑在成员定义之前出现的声明，而忽略在成员定义之后的声明。

对于在类定义之外的成员定义，在查找成员定义中出现的名字时，这里有一个小技巧可以帮助你记住查找域的顺序。限定修饰成员名的名字指示了要被查找的域的顺序。例如，上个例子中 action()成员名被修饰如下：

```
cplusplus_primer::List::ListItem::action()
```

限定修饰符 cplusplus_primer::List::ListItem 以反向顺序指出了要被查找的名字空间和类域。第一个查找的域是类 ListItem 的域。然后查找其外围类 List 的类域。最后在检查含有 action()定义的域之前，先查找名字空间 cplusplus_primer。在这样的查找过程里，在所有类域中总是考虑所有的成员声明，而在名字空间中则只考虑在成员定义之前可见的声明。

在名字空间域中定义的类对于整个程序都是可见的。如果头文件 primer.h 不只被包含在一个程序文本文件中，那么就可以在不同的文件中使用 cplusplus_primer::List 引用同一个类。类是一个程序实体，在一个程序中可以为其提供一个以上的定义。在每个类或者其成员被定义或使用的文件中，一个类的定义只能被提供一次。但是；这些类定义在它出现的所有文本文件中必须完全相同。因此，名字空间的类定义应该被放在一个头文件中，如 primer.h。然后，这个头文件可以被包含在每个使用或定义类成员的文本文件中，这样可以防止不匹配的情况发生，比如一个类定义不只被编写一次而且不匹配。

名字空间类的非 inline 成员函数和静态数据成员也是程序实体。但是，在整个程序中只能为这些成员提供一个定义。因此这些成员的定义不应该被放在头文件的类定义中，而是放在它们自己单独的文本文件中，比如程序文本文件 primer.C 中。

练习 13.22

现在请用在练习 13.21 中定义的类 iStack，把异常类 pushOnFull 和 popOnEmpty 声明为名字空间 LibException 的成员，如下：

```
namespace LibException {
    class pushOnFull{ };
    class popOnEmpty{ };
}
```

然后把类 iStack 声明为名字空间 Container 的成员。修改类 iStack 的定义及其成员函数的定义，以及 main()的定义，以便引用这些作为名字空间成员的类。

13.12 局部类 ※

类也可以定义在函数体内，这样的类被称为局部类（local class）。局部类只在定义它的局部域内可见。与嵌套类不同的是，在定义该类的局部域外没有语法能够引用局部类的成员。

因此，局部类的成员函数必须被定义在类定义中。在实际中，这就把局部类的成员函数的复杂性限制在几行代码中。否则，对读者来说，代码将变得很难理解。

因为没有语法能够在名字空间域内定义局部类的成员，所以也不允许局部类声明静态数据成员。

在局部类中嵌套的类可以在其类定义之外被定义。但是，该定义必须出现在包含外围局部类定义的局部域内。在局部域定义中的嵌套类的名字必须由其外围类名限定修饰。在外围类中，该嵌套类的声明不能被省略，例如：

```
void foo( int val )
{
    class Bar {
    public:
        int barVal;
        class nested;          // 嵌套类的声明是必需的
    };

    // 嵌套类定义
    class Bar::nested {
        // ...
    };
}
```

外围函数没有特权访问局部类的私有成员。当然，这可以通过使外围函数成为局部类的友元来实现。但是，看起来，局部类几乎从不需要私有成员。能够访问局部类的程序部分只有很少的一部分。局部类被封装在它的局部域中，通过信息隐藏进一步封装好像有点太过了。在实际中，很难找到一个理由不把局部类的所有成员都声明为公有的。

同嵌套类一样，局部类可以访问的外围域中的名字也是有限的。局部类只能访问在外围局部域中定义的类型名、静态变量以及枚举值。例如：

```
int a, val;

void foo( int val )
{
    static int si;
    enum Loc { a = 1024, b };

    class Bar {
    public:
        Loc locVal;                    // ok;
        int barVal;
        void fooBar( Loc l = a ) {     // ok: Loc::a
            barVal = val;              // 错误：局部对象
            barVal = ::val;            // OK: 全局对象
            barVal = si;               // ok: 静态局部对象
            locVal = b;                // ok: 枚举值
        }
    };
    // ...
}
```

在局部类体内（不包括成员函数定义中的）的名字解析过程是：在外围域中查找出现在局部类定义之前的声明。在局部类的成员函数体内的名字的解析过程是：在查找外围域之前，首先直找该类的完整域。

还是一样，如果先找到的声明使该名字的用法无效，则不考虑其他声明。即使在 fooBar() 中使用 val 是错的，编译器也不会找到全局变量 val，除非用全局域解析操作符限定修饰 val。

14

类的初始化、赋值和析构

本章将详细介绍程序中的类对象的自动初始化、赋值和析构。初始化由构造函数（constructor）支持。构造函数是一个可能被重载的用户定义函数，它是由类设计者提供的，在程序中的对象第一次被使用之前，构造函数被自动应用在每个类对象上。析构函数（destructor）是与构造函数互补的用户自定义成员函数，在对象的最后一次被使用之后它被自动应用在每个类对象上。析构函数主要被用来释放在类的构造函数中或整个生命期中获得的资源。

缺省情况下，用一个类的对象初始化该类的另一个对象，或者向该类另一个对象赋值，都由缺省的按成员语义（default memberwise semantics）支持，每个类成员被依次拷贝。在通常情况下，这种按成员语义已经足够了，然而在某些环境下，它对类的安全性和处理正确性还不够。在这些情况下，需要类的设计者提供特殊的拷贝构造函数（copy constructor）和拷贝赋值操作符（copy assignment operator）的定义。通常，提供这些特殊的成员函数最困难的一步是：意识到我们的确需要提供它们。

14.1 类的初始化

考虑下面的类定义：

```
class Data {
public:
    int ival;
    char *ptr;
};
```

为了能够安全地使用这个类的对象，我们必须确保它的两个成员都被正确地初始化。但是，对于不同的类这意味着什么呢？直到理解了这个类所代表的抽象时我们才能回答。如果它代表某个公司的雇员，那么 ptr 可能被设置为雇员的名字，而 ival 是雇员的惟一雇员号。负数或 0 是无效的。如果该类表示一个城市当前的气温，则负数、0 或正数都有效。另一种可能是，Data 表示一个被引用计数的字符串（reference-counted string）：ival 的值是以 ptr 指向的字符串被引用的次数。在这种抽象下，ival 用初始值 1 做初始化。如果该值降为 0，则删

除该对象。

当然，类及其数据成员的助记名会让程序的读者明白该类的意图，但是这对编译器没有提供任何额外的信息。为了使编译器能够了解我们的意图，我们必须提供一个或一组重载的特殊初始化构造函数。根据对象定义中指定的一组初始值，编译器会选择适当的构造函数。例如，下列每个函数都表示了一个合法的、惟一的 Data 类对象的初始化操作。

```
Data dat01( "Venus and the Graces", 107925 );
Data dat02( "about" );
Data dat03( 107925 );
Data dat04;
```

当我们需要一个类对象而又不知道初始值应该是什么的时候，在程序中这种情况也是有可能的（如 dat04），或许这些值只能在后面才可以确定。但是我们仍需要提供一些初始值，或者只是表明现在还没有提供初始值。在某些意义上来说，有时候需要初始化一个类对象表明它还没有被初始化。多数类都提供了一个特殊的**缺省构造函数**（default constructor），它不需要指定初始值。典型情况下，如果类对象是由缺省构造函数初始化的，则我们可以认为它还没有被初始化。

Data 类需要提供一个构造函数吗？正如它的定义所示，它不需要，因为它的所有数据成员都是公有的。从 C 语言继承来的机制支持显式初始化表，类似于用在初始化数组上的初始化表。例如：

```
int main()
{
    // local1.ival = 0; local1.ptr = 0
    Data local1 = { 0, 0 };

    // local2.ival = 1024;
    // local2.ptr = "Anna Livia Plurabelle"
    Data local2 = { 1024, "Anna Livia Plurabelle" };

    // ...
}
```

根据数据成员被声明的顺序，这些值按位置被解析。例如，下面是一个编译错误，因为 ival 在 ptr 之前被声明：

```
// 错误: ival = "Anna Livia Plurabelle"
//       ptr = 1024
Data local2 = { "Anna Livia Plurabelle" , 1024 };
```

显式初始化表有两个主要缺点。它只能被应用在所有数据成员都是公有的类的对象上（即显式初始化表不支持使用数据封装和抽象数据类型——这些在 C 语言中都没有，而这种形式的初始化正是从 C 语言继承来的）；它要求程序员的显式干涉，增加了意外（忘了提供初始化表）和错误（弄错了初始值顺序）的可能性。

既然已知这些缺点，那么使用显式初始化表代替构造函数有什么理由吗？在实际中是有的。在某些应用中，通过显式初始化表，用常量值初始化大型数据结构比较有效。例如，或

许我们正在创建一个调色板，或者向一个程序文本中注入大量常量值，如一个复杂地理模型中的控制点和节点值。在这些情况下，显式初始化可以在装载时刻完成，从而节省了构造函数的启动开销（即使它被定义为 inline），尤其是对全局对象。[23]

但是，通常来说，比较好的类初始化机制是构造函数，它保证在每个对象的首次使用之前被编译器自动应用在每个类对象上：在下一节，我们将详细了解类的构造函数。

14.2 类的构造函数

构造函数与类同名，我们以此来标识构造函数。为了声明一个缺省的构造函数，我们这样写[24]：

```
class Account {
public:
    // 缺省构造函数
    Account();

    // ...
private:
    char *_name;
    unsigned int _acct_nmbr;
    double _balance;
};
```

构造函数上惟一的语法限制是，它不能指定返回类型，甚至 void 也不行。例如，下列两个声明都是错误的：

```
// 错误：构造函数不能指定返回值。
void Account::Account() { ... }
Account* Account::Account( const char *pc ) { ... }
```

C++语言对于一个类可以声明多少个构造函数没有限制，只要每个构造函数的参数表是惟一的即可。

我们怎么能知道要定义哪个或多少个构造函数呢？在最小情况下，我们必须允许用户为每一个需要设置的数据成员提供一个初始值。例如，一个账户号码要能被设置或自动生成来保证其惟一性。对于我们的目的，我们让它自动生成。这就要求我们初始化两个成员_name 和_balance：

```
Account( const char *name, double open_balance );
```

用这个构造函数初始化的 Account 对象可以被定义如下：

```
Account newAcct( "Mikey Metz", 0 );
```

如果有许多账户以开户余额 0 开始，那么用户可能只要求指定一个名字，让构造函数自动把_balance 初始化为 0。另一种方案是提供第二种形式的构造函数：

```
Account( const char *name );
```

[23] 带有例子和粗略性能分析的详细讨论见［LIPPMAN96a］。

[24] 通常我们会将_name 声明为 string 类型。我们把它声明为 C 风格字符串，为的是将关于“类数据成员的初始化”的讨论推迟到 14.4 节。

另一种方案是给双参数的构造函数，已提供一个缺省值 0：

```
Account( const char *name, double open_balance = 0.0 );
```

这两个构造函数都提供了用户要求的功能，在这个意义上讲，两个方案都是可接受的。更好的方案是使用缺省实参，因为它减少了与类相关的构造函数的数目。

我们还应该为“指定一个开户余额而没有客户名的情况”提供支持吗？正如 Account 类定义所示，类规范显式地禁止了它。我们的双参数构造函数的第一个参数带有缺省实参，它为接受 Account 类数据成员的初始值提供了完整的接口，这些数据成员可以由用户设置：

```
class Account {
public:
    // 缺省构造函数
    Account();

    // 声明中的参数名不是必需的
    Account( const char*, double=0.0 );
    const char* name() { return _name; }

    // ...

private:
    // ...
};
```

下面是两个合法的 Account 类对象定义，它们向构造函数传递了一个或两个实参：

```
int main()
{
    // ok: 都调用双参数构造函数
    Account acct( "Ethan Stern" );
    Account *pact = new Account( "Michael Lieberman", 5000 );

    if ( strcmp( acct.name(), pact->name() ))
        // ...
}
```

C++要求，在类对象首次被使用之前，构造函数将被应用在该对象上。这意味着，在 if 语句的条件测试之前，构造函数首先被应用在 acct 和指针 pact 所指的对象上。

在内部，编译器会重写我们的程序，插入对构造函数的调用。下面给出了在 main()中 acct 的定义可能被展开的样子：

```
// C++ 伪代码
// 说明内部的构造函数插入情况
int main()
{
    Account acct;
    acct.Account::Account("Ethan Stern", 0.0);

    // ...
}
```

当然，如果构造函数实例被定义为 inline，那么它就会在调用点上被展开。

虽然 new 表达式的处理有点复杂，但是只有 new 表达式成功地换得所需内存，构造函数才会被调用。pact 定义的扩展可能如下（有些简化）：

```
// C++ 伪代码
// 使用 new 表达式的构造函数插入情况
int main()
{
     // ...

     Account *pacct;
     try {
          pact = _new( sizeof( Account ));
          pact->Account::Account(
               "Michael Lieberman", 5000.0);
     }

     catch( std::bad_alloc ) {
          // 操作符 new 失败
          // 构造函数没有被执行
     }

     // ...
}
```

为构造函数指定实参有三种等价形式：

```
// 一般等价的形式
Account acct1( "Anna Press" );
Account acct2 = Account( "Anna Press" );
Account acct3 = "Anna Press";
```

acct3 的形式只能被用于指定单个实参的情形。对于两个以上的实参，只能使用 acct1 和 acct2 的形式。一般来说，我们推荐使用 acct1 的形式：

```
// 推荐的构造函数形式
Account acct1( "Anna Press" );
```

C++语言新手常犯的错误是，按如下方式声明一个用缺省构造函数初始化的对象：

```
// 喔！并没有像期望的那样工作
Account newAccount();
```

它能通过编译，但是，当我们试图使用它时：

```
// 编译错误
if ( !newAccount.name() ) ...
```

编译器会抱怨我们不能把成员访问符应用到函数上，定义

```
// 定义了一个函数 newAccount,
// 不是一个 Account 类对象
Account newAccount();
```

被编译器解释为定义了一个没有参数、返回一个 Account 类型对象的函数——完全不是我们的意图。用缺省构造函数初始化类对象的正确声明是去掉尾部的小括号：

```
// ok: 定义了一个类对象
Account newAccount;
```

只有当没有构造函数或声明了缺省构造函数时，我们才能不指定实参集来定义类对象。一旦一个类声明了一个或者多个构造函数，类对象就不能被定义为不调用任何构造函数的实例。尤其是，如果一个类声明了一个包含多个参数的构造函数，但没有声明缺省构造函数，则每个类对象的定义都必须提供所需的实参。例如，对于我们的项目中有人可能会说，定义缺省构造函数没有任何意义，因为每个有效的帐户都必须有一个用户名。修改后的 Account 类可能去掉了缺省的构造函数。

```
class Account {
public:
    // 声明中的参数名不是必需的
    Account( const char*, double=0.0 );
    const char* name() { return _name; }

    // ...

private:
    // ...
};
```

现在，每个 Account 类对象必须至少为构造函数提供一个 C 风格字符串实参，才能是有效的类对象定义。尽管这或许严格遵守了 Account 类抽象的规范，但是，在实践中，这被证明是不切合实际的。为什么？因为容器类（比如 vector）要求它们的类元素或者提供缺省的构造函数，或者不提供构造函数。类似地，对于类对象的动态数组，在分配内存的时候也要求或者有缺省构造函数，或者没有构造函数。例如，如果用户这样写，则我们新定义的 Account 类就会失败：

```
// 错误: 要求缺省构造函数
Account *pact = new Account[ new_client_cnt ];
```

在实践中，如果定义了其他构造函数，则也有必要提供一个缺省构造函数。

如果一个类没有合适的缺省值怎么办呢？例如，我们的 Account 类要求为每个有效的 Account 类对象指定一个名字。在这种情况下，我们能做到的最好情况是，初始化该对象。指明它还没有被有效的值初始化。例如：

```
// 缺省 Account 构造函数
inline Account::
Account() {
    _name = 0;
    _balance = 0.0;
    _acct_nmbr = 0;
}
```

于是，在使用 Account 类对象之前，我们需要在 Account 成员函数中进行检查以保证它的完整性。

类的初始化有一种可替换的语法：**成员初始化表**（member initialization list,），是由逗号分开的成员名及其初值的列表。例如；缺省的 Account 构造函数可以这样写：

```
// 使用成员初始化表的缺省 Account 构造函数
inline Account::
```

```
Account()
: _name( 0 ),
     _balance( 0.0 ), _acct_nmbr( 0 )
{}
```

成员初始化表只能在构造函数定义中被指定，而不是在其声明中。该初始化表被放在参数表和构造函数体之间。由冒号开始。下面是双参数的构造函数，部分利用了成员初始化表：

```
inline Account::
Account( const char* name, double opening_bal )
: _balance( opening_bal )
{
     _name = new char[ strlen(name)+1 ];
     strcpy( _name, name );

     _acct_nmbr = get_unique_acct_nmbr();
}
```

get_unique_acct_nmbr()是一个非公有成员，它返回一个保证没有被使用的帐号。

构造函数不能用 const 或 volatile 关键字来声明（见 13.3.5 节的讨论）。例如，如下这样写是非法的：

```
class Account{
public:
     Account() const;  // error
     Account() volatile; // error

     // ...
};
```

显然，这并不意味着 const 和 volatile 类对象不能用构造函数来初始化。而是说，被应用到类对象上的适当的构造函数与该对象是 const、非 const 或 volatile 无关。只有当构造函数执行完毕。类对象已经被初始化的时候，该类对象的常量性才被建立起来。一旦析构函数被调用，常量性就消失了。因此，一个 const 类对象在“从其构造函数完成到析构函数开始”这段时间内才被认为是 const 的，对 volatile 类对象也一样。

考虑如下程序段：

```
// 在某个头文件中
extern void print( const Account &acct );

// ...

int main()
{
     // 把 "oops" 转换成一个 Account 对象
     // 用 Account::Account( "oops", 0.0 )
     print( "oops" );

     // ...
}
```

缺省情况下，单参数构造函数（或者有多个参数，除了第一个参数外，其他都有缺省实参）被用作转换操作符。在上面的程序段中的 print()的调用里，Account 构造函数被编译器隐

式地应用，以便把一个文字字符串转换成一个 Account 对象，尽管这种转换在这种情况下并不合适。

无意的隐式类转换，如把“oops”转换成一个 Account 对象，已经被证明是很难跟踪的错误源。关键字 explicit 被引入到标准 C++中，以帮助我们抑制这种不受欢迎的编译器辅助行为。explicit 修饰符通知编译器不要提供隐式转换：

```
class Account{
public:
    explicit Account( const char*, double=0.0 );
    // ...
};
```

explicit 只能被应用在构造函数上（关于转换操作符和关键字 explicit 的讨论请参见 15.9.2 节）

14.2.1 缺省构造函数

缺省构造函数是指不需要用户指定实参就能够被调用的构造函数，这并不意味着它不能接受实参。只意味着构造函数的每个参数都有一个缺省值与之关联。例如，下列每个函数都表示一个缺省构造函数：

```
// 每个都是缺省构造函数
Account::Account() { ... }
iStack::iStack( int size = 0 ) { ... }
Complex::Complex(double re=0.0,double im=0.0) { ... }
```

当我们写：

```
int main()
{
    Account acct;
    // ...
}
```

编译器首先检查 Account 类是否定义了缺省构造函数。以下情况之一会发生：

1．定义了缺省构造函数，它被应用到 acct 上。

2．定义了缺省构造函数，但它不是公有的。acct 的定义被标记为编译时刻错误：main()没有访问权限。

3．没有定义缺省构造函数，但是定义了一个或者多个要求实参的构造函数。acct 的定义被标记为编译时刻错误；实参太少。

4．没有定义缺省构造函数，也没有定义其他构造函数。该定义是合法的。acct 没有被初始化，没有调用任何构造函数。

到现在为止，第 1 和第 3 项很容易理解（若不理解，请回头看一看本章从开始到现在的内容——希望这不会导致无限阅读循环！）。让我们更仔细地看看第 2 和第 4 项。

如果 Account 类把它的所有成员都声明成公有的，但没有声明构造函数，如

```
class Account {
public:
    char *_name;
    unsigned int _acct_nmbr;
```

```
        double _balance;
};
```

那么，每个 Account 类对象的定义都不会导致“类特有的初始化”发生。三个成员的初始值取决于每个对象定义的上下文环境。如果声明一个静态对象如下：

```
// 静态范围
// 每个对象相关的内存被初始化为 0

Account global_scope_acct;
static Account file_scope_acct;

Account foo()
{
    static Account local_static_acct;
    // ...
}
```

则保证成员被初始化为 0（对非类对象也一样）。

但是，局部定义或动态分配的对象会被一个随机值初始化，该值是程序运行栈中该内存上一次被使用的结果。例如：

```
// 局部或堆对象在被初始化或赋值之前
// 不会被初始化

Account bar()
{
    Account local_acct;
    Account *heap_acct = new Account;
    // ...
}
```

新用户常常会错误地认为，如果不存在缺省构造函数，则编译器会自动生成一个缺省构造函数，并将其应用在对象上，以初始化类的数据成员。对于我们定义的 Account 类来说，这就不是真的：系统既没有生成缺省构造函数也没有调用它。对于含有类数据成员或继承来的比较复杂的类，这在部分上是对的；可能会生成一个缺省构造函数，但是它不会为内置或复合型的数据成员（如指针或数组）提供初始值。

如果我们想初始化内置或复合型的数据成员，则我们必须在一个或一组构造函数中显式地完成。如果不这样做，就不可能知道“局部或动态分配的类对象中的内置和复合型数据成员”是一个有效值，还是一个未初始化的值。[25]

14.2.2 限制对象创建

构造函数的可访问性由其声明所在的访问区来决定。我们可以通过把相关的构造函数放

[25] 对于在 C 中写程序的人来说，Account 定义的行为与在 C 中所写的下列代码一样：

```
typedef struct {
    char *_name;
    unsigned int _acct_nmbr;
    double _balance;
} Account;
```

到非公有访问区内，从而限制或显式禁止某些形式的对象创建动作。在下面的例子中，缺省的 Account 构造函数被声明为私有的，而双参数构造函数则被声明为公有的：

```
class Account {
    friend class vector< Account >;
public:
    explicit Account( const char*, double = 0.0 );
    // ...
private:
    Account();
    // ...
};
```

一般程序只能用关联名或账户名和开户余额定义 Account 对象。Account 的成员函数及其友元 vector 可以用任何一个构造函数来定义 Account 对象。

在实际的 C++程序中，非公有的构造函数的主要用处是：

1．防止用一个类的对象向该类另一个对象作拷贝（在下一小节讨论）；

2．指出只有当一个类在继承层次中被用作基类，而不能直接被应用程序操纵时，构造函数才能被调用（见第 17 章，关于继承和面向对象程序设计的讨论）。

14.2.3 拷贝构造函数

用一个类对象初始化该类的另一个对象被称为缺省按成员初始化（default memberwise initialization）。在概念上，一个类对象向该类的另一个对象作拷贝是通过依次拷贝每个非静态数据成员来实现的。类的设计者也可以通过提供特殊的拷贝构造函数（copy constructor）来改变缺省的行为。如果定义了拷贝构造函数，则在用一个类对象初始化该类另一个对象时它就会被调用。

缺省按成员的初始化对于类的正确行为常常是不合适的。通过定义拷贝构造函数的显式实例，我们可以改变缺省的行为。我们的 Account 类要求我们这样做，否则两个 Account 对象会有相同的帐号，这在该类的规范中显然是不允许的。

拷贝构造函数有一个指向类对象的引用作为形式参数（传统上被声明为 const）。下面是它的实现：

```
inline Account::
Account( const Accout &rhs )
: _balance( rhs._balance )
{
    _name = new char[ strlen(rhs._name)+1 ];
    strcpy( _name, rhs._name );

    // 不能拷贝 rhs._acct_nmbr
    _acct_nmbr = get_unique_acct_nmbr();
}
```

当我们写：

```
Account acct2( acct1 );
```

编译器判断是否为 Account 类声明了一个显式的拷贝构造函数。如果声明了拷贝构造函数，并且是可以访问的，则调用它。如果声明了拷贝构造函数但是不可访问，则 acct2 的定义

就是一个编译时刻错误。如果没有声明拷贝构造函数的实例，则执行缺省的按成员初始化。如果我们后来引入或去掉了一个拷贝构造函数的声明，则用户程序无需改变，但是，需要重新编译它们（14.6 节将详细讲解按成员初始化）

练习 14.1

下列说明哪些不是真的？为什么？

(a) 一个类必须至少提供一个构造函数。
(b) 缺省构造函数是参数表中没有参数的构造函数。
(c) 如果对于一个类不存在有意义的缺省值，则该类不应该提供一个缺省构造函数。
(d) 如果一个类不显式地提供缺省构造函数，则编译器会自动生成一个，用相关类型的缺省值初始化每个数据成员。

练习 14.2

请为下列数据成员提供一个或一组构造函数，并说明你的选择。

```
class NoName{
public:
     // constructor(s) go here ...
     // ...
protected:
     char *pstring;
     int ival;
     double dval;
};
```

练习 14.3

选择下列抽象中的一个（或选择你自己的抽象）。判断对于该类什么数据（可以由用户设置的）比较合适？请提供适当的构造函数集，并说明你的选择。

(a) Book
(b) Date
(c) Employee
(d) Vehicle
(e) Object
(f) Tree

练习 14.4

使用下列 Account 类定义

```
class Account{
public:
     Account();
     explicit Account( const char*, double=0.0 );
     // ...
};
```

请说明下面的语句会发生什么事情？

```
(a)  Account acct;
(b)  Account acct2 = acct;
(c)  Account acct3 = "Rena Stern";
(d)  Account acct4( "Anna Engel", 400.00 );
(e)  Account acct5 = Account( acct3 );
```

练习 14.5

拷贝构造函数的参数虽然不一定是 const，但它却必须是引用。下面语句为什么是错的？

```
Account::Account( const Account rhs );
```

14.3 类的析构函数

提供构造函数的一个目的是为了自动获取资源、我们已经看到了这样的例子，在 Account 类的构造函数中，通过应用 new 表达式分配了一个字符数组，并保证帐号的惟一性。另一种可能的情况是。我们或许希望在共享内存区或线程的临界区设置一个互斥锁。但是我们还缺少一种对称的操作，它为生命期即将结束的类对象返还相关的资源或者自动释放资源。**析构函数**（destructor）就是这样一个特殊的类成员函数。它是构造函数的互补。

析构函数是一个特殊的由用户定义的成员函数，当该类的对象离开了它的域，或者 delete 表达式应用到一个该类的对象的指针上时，析构函数会自动被调用。析构函数的名字是在类名前加上波浪线（~），它不返回任何值也没有任何参数。因为它不能指定任何参数，所以它也不能被重载。尽管我们可以为一个类定义多个构造函数，但是我们只能提供一个析构函数，它将被应用在类的所有对象上。下面是 Account 类的析构函数：

```
class Account {
public:
     Account();
     explicit Account( const char*, double=0.0 );
     Account( const Account& );
     ~Account();
     // ...
private:
     char *_name;
     unsigned int _acct_nmbr;
     double _balance;
};
inline
Account::~Account()
{
     delete [] _name;
     return_acct_nmbr( _acct_nmbr );
}
```

注意，我们的析构函数并没有复位（reset）数据成员的值：

```
inline
Account::~Account()
```

```
{
    // 这是必要的
    delete [] _name;
    return_acct_nmbr( _acct_nmbr );
    // 没有必要
    _name = 0;
    _balance = 0.0;
    _acct_nmbr = 0;
}
```

尽管这么做没有错，但是没有必要这样做，因为与这些成员相关联的存储区将被归还。更一般的情况，我们考虑下面的类

```
class Point3d {
public:
    // ...
private:
    float x, y, z;
};
```

为了允许用户初始化这三个坐标的内存，构造函数是必需的。但是析构函数呢？在这种情况下，析构函数不是必需的。Point3d 类对象没有资源需要被释放，三个坐标成员的存储区在每个类对象生命期的开始和结束时，由编译器自动分配和释放。

一般地，如果一个类的数据成员是按值存储的，比如 Point3d 的三个坐标成员，则无需析构函数。并不是每一个类都要求有析构函数，即使我们为该类定义了一个或多个构造函数。析构函数主要被用来放弃在类对象的构造函数或生命期中获得的资源，如释放互斥锁或删除由操作符 new 分配的内存。

析构函数不局限在放弃资源上。一般地，析构函数可以执行“类设计者希望在最后一次使用对象之后执行的任何操作”。例如，用于程序性能工具的一项常见技术是定义一个 Timer 类。Timer 类的构造函数启动某种形式的程序时钟，它的析构函数停止时钟，并以某种形式显示结果。于是，我们可以有条件地在希望计时的关键程序段内定义一个 Timer 类对象，如下所示：

```
{
    // 关键代码段开始
#ifdef PROFILE
    Timer t;
#endif
    // 关键代码段
    // t 的析构函数自动显示累计时间
}
```

为了证实我们确实理解了析构函数（和构造函数）的行为，让我们通过下列程序段浏览一个例子：

```
(1)  #include "Account.h"
(2)  Account global( "James Joyce" );
(3)  int main()
(4)  {
(5)      Account local( "Anna Livia Plurabelle", 10000 );
(6)      Account &loc_ref = global;
```

```
(7)       Account *pact = 0;
(8)
(9)            {
(10)      Account local_too( "Stephen Hero" );
(11)      pact = new Account( "Stephen Dedalus" );
(12)      }
(13)
(14)      delete pact;
(15) }
```

调用了多少个构造函数？四个：一个是针对第(2)行的全局对象 gobal；两个针对局部对象 local 和 local_too 调用的，分别在第(5)行和(10)行；还有一个针对在(11)行分配的堆对象。在第(6)行的类对象引用 loc_ref 的声明，以及在第(7)行的类对象指针 pact 的声明都不会引起调用构造函数。引用被用作一个已被构造的对象的别名。在这里，local_ref 被用作 global 的别名。指针也一样，它只是指向一个已被构造的对象（在本例中，指向第(11)行堆上分配的对象）或不指向任何对象（第(7)行）。

类似地，程序也调用了四个析构函数：一个是针对第(2)行声明的全局对象 global；另两个是两个局部对象；还有一个是针对第(14)行删除的堆对象。但是，与构造函数不同的是，没有相关的源代码语句指出在一个类对象上调用了析构函数；而是编译器在类对象的最后一次被使用之后。在相关域结束之前插入了调用。

在程序执行的初始化阶段和收尾阶段，全局类对象调用它们自己的构造函数和析构函数。尽管在全局对象被定义的文件中，它们这种表现非常好，但是在几个独立编译的文件之间引用全局对象的时候，其安全和使用效率成了 C++中富有挑战性的设计问题。[26]

当类对象的指针或引用离开域时（被引用的对象还没有结束生命期），析构函数不会被调用。

C++语言在内部保证，不会用 delete 操作符删除不指向任何对象的指针，所以我们无需再编写代码来保证这一点：

```
// 没有必要—由编译器隐式执行
if ( pact != 0 ) delete pact;
```

无论何时，当在一个函数内删除一个独立的堆对象时，最好是用 auto_ptr 类对象而不是一个实际的指针（关于 auto_ptr 的讨论见 8.4 节）。对于堆上的类对象尤其应该这样做，否则的话，如果应用 delete 表达式失败，比如一个异常被抛出的情况下，不仅会导致内存泄漏。而且析构函数也不会被调用。例如，下面是我们用 auto_ptr 重写之后的程序示例（它被稍做修改，因为 auto_ptr 对象不支持被显式地重置以指向第二个对象，除非赋值第二个 auto_ptr）。

```
#include <memory>
#include "Account.h"

Account global( "James Joyce" );
int main()
{
     Account local( "Anna Livia Plurabelle", 10000 );
     Account &loc_ref = global;
     auto_ptr<Account> pact( new Account( "Stephen Dedalus" ));
```

[26] 这个问题的原始讨论以及最广泛的解决方案见 Jerry Schwarz 在［LIPPMAN96b］中的文章。

```
        {
            Account local_too( "Stephen Hero" );
        }
        // auto_ptr object destroyed here
    }
```

14.3.1 显式的析构调用

```
// ...
switch( swt ) {
case 0:
     return;
case 1:
     // 进行操作
     return;
case 2:
     // 进行其他操作
     return;
// 等等
}
```

在每个 return 语句之前，析构函数都必须被内联地展开。在 Account 类的析构函数的情况下，由于它的长度较小，所以多次展开的相关开销也较小。但是，如果已经发现它确实是一个问题，则解决方案是：或者把析构函数声明为非内联的，或者重新改写程序代码。一种可能的重写方案是在每个 case 标签中用 break 语句代替 return 语句，然后在 switch 语句后面引入一个 return 语句：

```
// 重写来提供一个返回点
switch( swt ) {
case 0:
     break;
case 1:
     // 进行操作
     break;
case 2:
     // 进行另一些操作
     break;
// 等等
}

// 单个返回点
return;
```

练习 14.6

给出如下的一组数据成员，其中 pstring 指向一个动态字符数组，请写一个合适的析构函数。

```
class NoName {
public:
     ~NoName();
     // ...
private:
     char *pstring;
     int ival;
     double dval;
};
```

练习 14.7

对于 14.2 节的练习 14.3 中选择的类，判断其是否需要析构函数，如果不需要，请说明原

因。否则，请给出它的实现。

练习 14.8

在下列代码段中出现了多少析构函数的调用？

```
void mumble( const char *name, double balance, char acct_type )
{
    Accout acct;

    if ( ! name )
        return;

    if ( balance <= 99 )
        return;

    switch( acct_type ) {
        case 'z': return;
        case 'a':
        case 'b': return;
    }

    // ...
}
```

14.4 类对象数组和 vector

类对象数组与内置类型数组的定义方式相同。例如，

```
Account table[ 16 ];
```

定义了一个含有 16 个 Account 对象的数组，且每个元素依次用 Account 缺省构造函数初始化。如果我们愿意的话，则可以用放在括号中的初始化表给构造函数提供显式实参。例如：

```
Account pooh_pals[] = { "Piglet", "Eeyore", "Tigger" };
```

定义了三个元素的数组，三个元素依次用构造函数

```
Account( "Piglet", 0.0 );    // 第一个元素
Account( "Eeyore", 0.0 );    // 第二个元素
Account( "Tigger", 0.0 );    // 第三个元素
```

初始化。

构造函数的单个实参可以简单地被显式指定，如上面的例子所示。如果我们希望指定多个实参，则需要使用完整的构造函数语法。例如：

```
Account pooh_pals[] = {
    Account( "Piglet", 1000.0 ),
    Account( "Eeyore", 1000.0 ),
    Account( "Tigger", 1000.0 )
};
```

如果要在数组的初始化表中指定缺省构造函数，我们可以使用带有空参数表的完整构造函数语法。例如：

```
Account pooh_pols[] = {
    Account( "Woozle", 10.0 ),
    Account( "Heffalump", 10.0 ),
    Account()
};
```

我们也可以按下面的写法，获得三个元素的等价数组：

```
Account pooh_pols[3] = {
    Account( "Woozle", 10.0 ),
    Account( "Heffalump", 10.0 )
};
```

即，数组初始化表被依次应用到数组的相继元素上。对于那些没有显式构造函数实参集合的元素，用类的缺省构造函数初始化。如果该类没有定义缺省构造函数，则初始化表必须为数组的每个元素提供构造函数实参。

用下标操作符可以访问类数组的单个元素，就像内置类型一样。因此，例如，

```
pooh_pals[0];
```

访问 Piglet，而

```
pooh_pals[1];
```

访问 Eeyore 等等。为了访问某个特定数组元素的类成员，我们把下标操作符和成员访问操作符组合起来。例如：

```
pooh_pals[1]._name != pooh_pals[2]._name();
```

对于在堆中分配的类对象数组的元素，我们没有办法提供一组显式的值来做初始化。如果希望支持通过 new 表达式分配数组，则类必须提供一个缺省构造函数，或不提供构造函数。在实践中，几乎所有的类都提供一个缺省构造函数。

声明

```
Account *pact = new Account[ 10 ];
```

创建了一个在堆中分配的、包含 10 个 Account 类对象的数组，它们都用 Account 类缺省构造函数初始化。

为了释放 pact 指向的数组，我们必须应用 delete 表达式，但是，简单地写

```
// 喔！不太对
delete pact;
```

是不够的，因为它不能识别出 pact 是一个指向类对象数组的指针。结果是，只有首元素被应用了 Account 析构函数，这并不是我们所希望的。为了在每个数组元素上应用析构函数，我们必须要在 delete 操作符和被删除的对象的地址之间加上一个空的方括号：

```
// ok:
// 表明 pact 指向一个数组
delete [] pact;
```

空的方括号表明 pact 指向一个数组。编译器获得数组中元素的个数，保证在每个元素上

都应用析构函数。

14.4.1 堆数组的初始化 ※

缺省情况下，在堆中分配的类对象数组的初始化要求两步：1）实际分配数组，并且，如果定义了缺省构造函数，则把它应用到每个元素上；2）每个元素后来被赋以一个特定的值。

为了提供一个完整的初始化步骤，程序员自己必须介入，支持以下语义：为数组元素的全部或者一部分指定初始值，并确保缺省构造函数被应用到那些没有获得初始值的元素上。下面是多种解决方案中的一种，它利用了定位 new 操作符。

```
#include <utility>
#include <vector>
#include <new>
#include <cstddef>
#include "Accounts.h"
typedef pair<char*,double> value_pair;
/* init_heap_array(),
* 被声明为静态成员函数
* 提供类对象堆数组的分配和初始化
*
* init_values: 数组元素的初始值对
* elem_count: 数组元素个数
* 如果为 0, 数组大小与 init_values vector 一样
*/
Account*
Account::
init_heap_array(
vector<value_pair> &init_values,
vector<value_pair>::size_type elem_count = 0 )
{
     vector<value_pair>::size_type
          vec_size = init_values.size();
     if ( vec_size 0 && elem_count 0 )
          return 0;
     // 分配的数组大小是 elem_count
     // 或者, 若 elem_count 为 0, 则为 vector 的大小
     size_t elems = elem_count
          ? elem_count : vec_size;

     // 找到一块不用的内存来保存数组
     char *p = new char[sizeof(Account)*elems];

     // 每个元素的独立初始化
     int offset = sizeof( Account );
     for ( int ix = 0; ix < elems; ++ix )
     {
          // 偏移到第 ix 个元素
          // 如果提供了一个初始值对
          // 把该对传递给构造函数
          // 否则, 调用缺省构造函数
```

```
            if ( ix < vec_size )
                new( p+offset*ix ) Account( init_values[ix].first,
                    init_values[ix].second );
            else new( p+offset*ix ) Account;
        }

        // ok: 元素被分配并初始化
        // 返回第一个元素的指针
        return (Account*)p;
    }
```

这里用到的技巧是“预分配”了一块足够的内存来存放要求的类数组。我们按照未经格式处理的形式（即 raw memory）来分配这块内存，这是为了避免对每个数组元素调用缺省构造函数。这是下面语句所做的：

```
    char *p = new char[sizeof(Account)*elems];
```

程序接下来遍历这块内存，对 p 做偏移，使得它指向下一个 Account 元素。如果提供了对初始值，则调用双参数的构造函数，否则调用缺省构造函数：

```
    for ( int ix = 0; ix < elems; ++ix ) {
        if ( ix < vec_size )
            new( p+offset*ix ) Account( init_values[ix].first,
                    init_values[ix].second );
        else new( p+offset*ix ) Account;
    }
```

正如在 14.3 节中所见，定位 new 操作符允许我们把一个类构造函数应用到预分配的内存上。在这种情况下，我们用定位操作符 new 在每个预分配的数组元素上依次应用 Account 类型的构造函数。因为在生成初始化的堆数组类时，我们已经改变了普通的分配机制，所以我们也必须为它的释放提供支持。应用传统的 delete 操作符不会奏效：

```
    delete [] ps;
```

为什么呢？因为 ps（我们假设它是通过调用 init_heap_array()而被初始化的）不是用普通的数组操作符 new 分配的，因此与 ps 相关联的元素个数是未知的，我们必须自己来做这项工作：

```
    void
    Account::
    dealloc_heap_array( Account *ps, size_t elems )
    {
        for ( int ix = 0; ix < elems; ++ix )
            ps[ix].Account::~Account();

        delete [] reinterpret_cast<char*>(ps);
    }
```

请回忆前面，在初始化函数中，我们用指针的算术运算访问数组的每个元素，

```
    new( p+offset*ix ) Account;
```

而这里我们通过索引 ps 来访问每个元素，

```
    ps[ix].Account::~Account();
```

区别是，尽管 ps 和 p 指向相同的内存区，但是 ps 被声明为类 Account 对象的指针，而 p 被声明为 char 的指针。索引 p 会产生数组的第 ix 个字节，而不是第 ix 个 Account 类对象。由于对于 p，相关联的类型无效，所以要求我们自己编写指针算术运算。

我们把这两个函数声明为 Account 类的静态成员函数：

```
typedef pair<char*,double> value_pair;
class Account {
public:
    // ...

    static Account* init_heap_array(
        vector<value_pair> &init_values,
        vector<value_pair>::size_type elem_count = 0 );

    static void dealloc_heap_array( Account*, size_t );

    // ...
};
```

14.4.2 类对象的 vector

当我们定义一个含有五个类对象的 vector 时，如：

```
vector< Point > vec( 5 );
```

元素的初始化过程如下：[27]

1．创建一个底层类类型的临时对象，在其上应用该类的缺省构造函数。

2．在 vector 的每个元素上依次应用拷贝构造函数，用临时类对象的拷贝初始化每一个类对象。

3．删除临时类对象。

尽管最终结果等同于定义五个类对象的数组，比如：

```
Point pa[ 5 ];
```

但是，初始化 vector 代价比较大：临时对象的构造和析构，以及拷贝构造函数往往比缺省构造函数计算上更复杂。

作为一般的设计原则，类对象的 vector 仅仅最适合于元素的插入操作，也就是，我们定义一个空的 vector。如果我们先预算出要插入的元素的数目，或者能够比较准确地猜出来，那么我们可以预留相应的存储区。然后，而进行元素插入。例如：

```
vector< Point > cvs;   // 空
int cv_cnt = calc_control_vertices();

// 预留内存以便存放 cv_cnt Point 对象
// cvs 仍然是空的...
cvs.reserve( cv_cnt );

// 打开一个文件并准备迭代它
```

[27] 相关构造函数的原型特征如下，拷贝构造函数依次把一个值应用到每个元素上。通过提供一个类对象作为第二个实参，而临时对象的创建不是必需的。

explicit vector (size_type n, const T& value=T(),const Allocator&=Allocator();

```
ifstream infile( "spriteModel" );
istream_iterator<Point> cvfile( infile ),eos;

// ok, 现在插入元素
copy( cvfile, eos, inserter( cvs, cvs.begin() ));
```

[copy()、inserter 类和 istream_iterator 在第 12 章讨论。] list 和 deque 对象的定义与 vector 对象有相同的行为。把一个类对象插入到每一种容器类型中，都是通过拷贝构造函数来实现的。

练习 14.9

下列语句哪些不正确？请改正你认为不正确的实例。

```
(a)  Account *parray[10] = new Account[10];
(b)  Account iA[1024] = {
         "Nhi", "Le", "Jon", "Mike", "Greg", "Brent", "Hank"
         "Roy", "Elena" };
(c)  string *ps=string[5]("Tina","Tim","Chyuan","Mira","Mike");
(d)  string as[] = *ps;
```

练习 14.10

在下列情况下，使用哪个类型更合适：静态数组，如 Account pa[10]、动态数组，还是 vector？请说明原因。

(a) 在一个名为 Lut()的函数中，有一个包含 256 个元素的集合，用来存放 Color 类对象，且元素的值是一个常量。

(b) 需要一个未知个数的 Account 元素集合，而每个帐户的数据都需要从一个文件中读取。

(c) 一个 elem_size 大小的 string 集合，由函数 gen_words(elem_size)产生，并传回给文本管理器。

练习 14.11

使用动态类数组的一个潜在缺点是，我们时常忘记放上一个方括号以表明指针指向一个数组，即写成：

```
// 喔：不检查 parray 是否指向一个数组
delete parray;
```

而不是：

```
// ok: 获取 parray 所指数组的大小
delete [] parray;
```

方括号的存在会使编译器获取数组的大小（size）。然后析构函数再被依次应用在每个元素上，一共 size 次。否则，只有一个元素被析构。无论哪种情况下，分配的全部空间都被返回给自由存储区。

在原来的语言设计中，对是否要求方括号来激发一个取大小操作、还是保持原先的语言

要求（由程序员在方括号中显式提供数组的长度），曾展开了激烈的讨论。原来的要求如下：

```
// 原来的语言设计要求提供显式大小
delete [10] parray;
```

为什么语言会改变成现在的样子，即，不要求用户提供显式的数组大小，而是要求存储和获取大小值？为什么不改成省略 delete 表达式中的方括号，而要求程序员记住该指针是指向单个对象还是一个数组？你会怎样设计语言？

14.5 成员初始化表

让我们修改 Account 类，重新声明它的_name 成员为 string 型：

```
#include <string>

class Account {
public:
     // ...

private:
     unsigned int _acct_nmbr;
     double _balance;
     string _name;
};
```

我们还需要修改构造函数。这涉及到两个主题：1）维持与原始接口的兼容性，同时适合新的类型；2）用一组相关的构造函数来初始化一个成员类对象。

原来的双参数 Account 构造函数：

```
Account ( const char*, double = 0.0 );
```

对于我们的新 string 类类型不够用。例如如下代码将失败：

```
string new_client( "Steve Hall" );
Account new_acct( new_client, 25000 );
```

因为在从 string 对象到 char*之间没有隐式转换。而写成：

```
Account new_acct( new_client.c_str(), 25000 );
```

虽然会奏效，但是可能会使用户困惑。一个解决方案就是简单地增加一个形式如下的新构造函数：

```
Account( string, double = 0.0 );
```

现在，当我们写以下语句：

```
Account new_acct( new_client, 25000 );
```

就会调用该实例，而旧的代码如：

```
Account *open_new_account( const char *nm )
{
     Account *pact = new Account( nm );

     // ...

     return pact;
}
```

将继续使用原来的双参数构造函数。

因为 string 类为 char*到 string 对象的转换提供了支持（关于类转换将在下一章讨论），所以我们也可以只简单地用带有 string 型第一个参数的实例，取代原来的双参数构造函数。在这种情况下，我们写：

```
Account myAcct( "Tinkerbell" );
```

“Tinkerbell”被转换成一个临时的 string 对象，然后该对象再被传递给一个双参数的构造函数，它的第一个参数是 string 类型。

我们的考虑是在“类 Account 构造函数的数量增加”和“处理 char*实参的低效（由于创建临时 string）”之间进行折衷和平衡。我们的设计选择是提供两个版本的双参数构造函数。重写的 Account 构造函数如下：

```
#include <string>

class Account {
public:
     Account();
     Account( const char*, double=0.0 );
     Account( const string&, double=0.0 );
     Account( const Account& );
     // ...
private:
     // ...
};
```

下一个要点是，怎样正确地初始化含有一组相关构造函数的类数据成员。这被分成三个子条目：

1．我们怎样调用它的缺省构造函数？我们将需要在缺省 Account 构造函数中做到这一点。

2．我们怎样调用它的拷贝构造函数？我们需要在 Account 拷贝构造函数和带有 string 型参数的双参数 Account 构造函数中做到这一点。

3．更一般的情况下，我们怎样把实参传递给一个成员类对象的构造函数？我们需要在有 char*型首参数的双参数 Account 构造函数中做到这一点。

解决方案是成员初始化表（在 14.2 节简要介绍过）。通过成员初始化表，类数据成员可以被显式初始化。成员初始化表是由逗号分隔的成员、名字实参对。例如，下面的双参数构造函数的新实现，就使用了成员初始化表（_name 现在是 string 型的成员类对象）：

```
inline Account::
Account( const char* name, double opening_bal )
: _name( name ), _balance( opening_bal )
{
     _acct_nmbr = get_unique_acct_nmbr();
}
```

成员初始化表跟在构造函数的原型后，由冒号开头。成员名是被指定的，后面是括在括号中的初始值，类似于函数调用的语法。如果成员是类对象，则初始值变成被传递给适当的构造函数的实参，该构造函数然后被应用在成员类对象上。在我们的例子中，name 被传递给应用在_name 上的 string 构造函数。_balance 用参数 opening_bal 初始化。

类似地，下面是另一个双参数 Account 构造函数：

```
inline Account::
Account( const string& name, double opening_bal )
     : _name( name ), _balance( opening_bal )
{
     _acct_nmbr = get_unique_acct_nmbr();
}
```

在这种情况下，string 的拷贝构造函数被调用，把成员类对象_name 初始化成 string 参数 name。

C++新手关注的一个常见问题是，使用初始化表和在构造函数内使用数据成员的赋值之间有什么区别。例如，以下代码

```
inline Account::
Account( const char *name, double opening_bal )
     : _name( name ), _balance( opening_bal )
{
     _acct_nmbr = get_unique_acct_nmbr();
}
```

和：

```
inline Account::
Account( const char *name, double opening_bal )
{
     _name = name;
     _balance = opening_bal;
     _acct_nmbr = get_unique_acct_nmbr();
}
```

它们的区别是什么？

两种实现的最终结果是一样的。在两个构造函数调用的结束处，三个成员都含有相同的值，区别是成员初始化表只提供该类数据成员的初始化。在构造函数体内对数据成员设置值是一个赋值操作。区别的重要性取决于数据成员的类型。

在概念上，很重要的一点是，我们可以认为构造函数的执行过程被分成两个阶段：隐式或显式初始化阶段，以及一般的计算阶段。计算阶段由构造函数体内的所有语句构成。在计算阶段中，数据成员的设置被认为是赋值，而不是初始化。没有清楚地认识到这个区别是程序错误和低效的常见源泉。

初始化阶段可以是显式的或隐式的，取决于是否存在成员初始化表。隐式初始化阶段按照声明的顺序依次调用所有基类的缺省构造函数。然后是所有成员类对象的缺省构造函数（我们将在第 17 章讨论面向对象程序设计时考虑基类）。例如，当我们写如下代码：

```
inline Account::
Account()
{
     _name = "";
     _balance = 0.0;
     _acct_nmbr = 0;
}
```

则初始化阶段是隐式的，在构造函数体被执行之前，先调用与_name 相关联的缺省 string

构造函数，这意味着把空串赋给_name 的赋值操作是没有必要的。

对于类对象，在初始化和赋值之间的区别是巨大的。成员类对象应该总是在成员初始化表中被初始化，而不是在构造函数体内被赋值。缺省 Account 构造函数的更正确的实现如下：

```
inline Account::
Account() : _name( string() )
{
    _balance = 0.0;
    _acct_nmbr = 0;
}
```

它之所以更正确，是因为我们已经去掉了在构造函数体内不必要的对_name 的赋值。但是，对于缺省构造函数的显式调用也是不必要的，下面是更紧凑但却等价的实现：

```
inline Account::
Account()
{
    _balance = 0.0;
    _acct_nmbr = 0;
}
```

剩下的问题是，对于两个被声明为内置类型的数据成员，其初始化情况如何？例如，用成员初始化表和在构造函数体内初始化_balance 是否等价？回答是不。对于非类数据成员的初始化或赋值，除了两个例外，两者在结果和性能上都是等价的。即，更受欢迎的实现是用成员切始化表：

```
// 更受欢迎的初始化风格
inline Account::
Account() : _balanae( 0.0 ), _acct_nmbr( 0 )
    { }
```

两个例外是指任何类型的 const 和引用数据成员。const 和引用数据成员也必须是在成员初始化表中被初始化，否则，就会产生编译时刻错误。例如，下列构造函数的实现将导致编译时刻错误：

```
class ConstRef {
public:
    ConstRef( int ii );

private:
    int i;
    const int ci;
    int &ri;
};

ConstRef::
ConstRef( int ii )
{ // 赋值
    i = ii;          // ok
    ci = ii;         // 错误：不能给一个 const 赋值
    ri = i;          // 错误：  ri 没有被初始化
}
```

当构造函数体开始执行时，所有 const 和引用的初始化必须都已经发生。因此，只有将它

们在成员初始化表中指定这才有可能。正确的实现如下：

```
// ok: 初始化引用和 const
ConstRef::
ConstRef( int ii )
     : ci( ii ), ri( i )
     { i = ii; }
```

每个成员在成员初始化表中只能出现一次，初始化的顺序不是由名字在初始化表中的顺序决定，而是由成员在类中被声明的顺序决定的。例如。给出下面的 Account 数据成员的声明顺序：

```
class Account {
public:
     // ...
private:
     unsigned int _acct_nmbr;
     double _balance;
     string _name;
};
```

下面的缺省构造函数：

```
inline Account::
Account() : _name( string() ), _balance( 0.0 ), _acct_nmbr( 0 )
     {}
```

的初始化顺序为 acct_nmbr、_balance，然后是_name。但是在初始化表中出现（或者在被隐式初始化的成员类对象中）的成员，总是在构造函数体内成员的赋值之前被初始化。例如，在下面的构造函数中：

```
inline Account::
Account( const char *name, double bal )
     : _name( name ), _balance( bal )
{
     _acct_nmbr = get_unique_acct_nmbr();
}
```

初始化的顺序是_balance、_name，然后是_acct_nmbr。

由于这种“实际的初始化顺序”与“初始化表内的顺序”之间的明显不一致，有可能导致以下难于发现的错误，当用一个类成员初始化另一个时：

```
class X {
     int i;
     int j;
public:
     // 喔！你看到问题了吗?
     X( int val )
          : j( val ), i( j )
          {}
     // ...
};
```

尽管看起来 j 好像是用 val 初始化的，而且发生在它被用来初始化 i 之前，但实际上是 i 先被初始化的，因此它是用一个还没有被初始化的 j 初始化的。我们的建议是，把“用一个成员对

另一个成员进行初始化（如果你真的认为有必要）”的代码放到构造函数体内，如下所示：

```
// 更好的习惯用法
X::X( int val ) : i( val ) { j = i; }
```

练习 14.12

下列构造函数的定义有什么问题？应该怎样修正？

```
(a)  Word::Word( char *ps, int count = 1 )
         : _ps( new char[strlen(ps)+1] ),
             _count( count )
     {
         if ( ps )
             strcpy( _ps, ps );
         else {
             _ps = 0;
             _count = 0;
         }
     }
(b)  class CL1 {
     public:
         CL1() { c.real(0.0); c.imag(0.0); s = "not set"; }
         // ...
     private:
         complex<double> c;
         string s;
     };
(c)  class CL2 {
     public:
         CL2( map<string,location> *pmap, string key )
             : _text( key ), _loc( (*pmap)[key] ) {}
         // ...
     private:
         location _loc;
         string _text;
     };
```

14.6 按成员初始化 ※

用一个类对象初始化另一个类对象，比如

```
Account oldAcct( "Anna Livia Plurabelle" );
Account newAcct( oldAcct );
```

被称为缺省的**按成员初始化**（default memberwise initialization）。缺省是因为它自动发生，无论我们是否提供显式构造函数。按成员是因为初始化的单元是单个非静态数据成员，而不是对整个类对象的按位拷贝。

最简单的按成员初始化的概念模型是，想像编译器内部如何产生一个特殊的拷贝构造的数。在拷贝构造函数中，每个非静态数据成员按照声明的顺序被依次初始化。例如，已知

Account 类的第一个定义：

```
class Account {
public:
     // ...

private:
     char *_name;
     unsigned int _acct_nmbr;
     double _balance;
};
```

我们可以认为缺省的 Account 拷贝构造函数被定义如下：

```
inline Account::
Account( const Account &rhs )
{
     _name = rhs._name;
     _acct_nmbr = rhs._acct_nmbr;
     _balance = rhs._balance;
}
```

“用一个类对象初始化该类另一个对象”发生在下列程序情况下：

1．用一个类对象显式地初始化另一个类对象，例如：

```
Account newAcct( oldAcct );
```

2．把一个类对象作为实参传递给一个函数，例如：

```
extern bool cash_on_hand( Account acct );
if ( cash_on_hand( oldAcct ))
     // ...
```

把一个类对象作为一个函数的返回值传递回来，例如：

```
extern Account
     consolidate_accts( const vector< Account >& )
{
     Account final_acct;

     // do the finances ...

     return final_acct;
}
```

3．非空顺序容器类型的定义，例如：

```
// 五个 string 拷贝构造函数被调用
vector < string > svec( 5 );
```

（在本例中，用 string 缺省构造函数创建一个临时对象，然后通过 string 拷贝构造函数，该临时对象被依次拷贝到 vector 的五个元素中。）

4．把一个类对象插入到一个容器类型中，例如：

```
svec.push_back( string( "pooh" ));
```

对于大多数实际的类定义，由于考虑到类的安全性以及用法正确性，所以说缺省的按成员初始化是不够的。最经常出现的情况是，一个类的数据成员是一个指向堆内存的指针，并且这块内存将由该类的析构函数删除，就如 Account 类中的_name 成员一样。

在缺省按成员初始化之后，newAcct._name 和 oldAcct._name 指向同一个 C 风格字符串。如果 oldAcct 离开了域，并且 Account 的析构函数被应用在其上，则 newAcct._name 现在指向一个被删除了的内存区。另一种情况是，如果 newAcct 修改了由_name 指向的字符串，则 oldAcct 也会受到影响。这种指向错误很难跟踪。

指针“别名（aliasing）”问题的一种解决方案是，分配该字符串的第二个拷贝，并初始化 newAcct._name 以指向这份新的拷贝。为实现这一点，我们必须改变 Account 类的缺省按成员初始化。我们通过提供一个显式的拷贝构造函数来做到这一点，由这个拷贝构造函数实现这种正确的初始化语意。

类的内部语义也可能使缺省的按成员初始化无效。不如前面所解释的，不能有两个 Account 类的对象持有同一个帐号。为了保证这一点，我们必须改变 Account 类的缺省按成员初始化。下面是解决这两个问题的拷贝构造函数：

```
inline Account::
Account( const Account &rhs )
{
    // 处理指针别名问题
    _name = new char[ strlen(rhs._name)+1 ];
    strcpy( _name, rhs._name );

    // 处理帐号惟一性问题
    _acct_nmbr = get_unique_acct_nmbr();

    // ok: 现在可以按成员拷贝
    _balance = rhs._balance;
}
```

在许多情况下，一个方案最难的部分就是要意识到它的必要性。

除了提供拷贝构造函数，另一种替代的方案是完全不允许按成员初始化。这可以通过下列两个步骤实现：

1．把拷贝构造函数声明为私有的，这可以防止按成员初始化发生在程序的任何一个地方（除了类的成员函数和友元之外）

2．通过有意不提供一个定义（但是，我们仍然需要第 1 步中的声明），可以防止在类的成员函数和友元中出现按成员初始化。C++语言不会允许我们阻止类的成员函数和友元访问任何私有类成员。但是通过不提供定义，任何试图调用拷贝构造函数的动作虽然在编译系统中是合法的，但是会产生链接错误，因为无法为它找到可解析的定义。

例如，为了不允许 Account 类的按成员初始化，我们必须如下声明该类：

```
class Account {
public:
    Account();
    Account( const char*, double=0.0 );
    // ...
private:
    Account( const Account& );
    // ...
};
```

14.6.1 成员类对象的初始化

把“C 风格字符串的_name 声明”替换成“string 类类型的_name 声明”会发生什么变化？会怎样影响缺省的按成员初始化的行为？我们的显式拷贝构造函数需要怎样改变？我们将在本小节依次讨论这些问题。

缺省的按成员初始化依次检查每个成员，如果成员是内置或复合数据类型，则直接执行从成员到成员的初始化。例如，在我们原来的 Account 类定义中，因为_name 是一个指针，所以它直接被初始化：

```
newAcct._name = oldAcct._name;
```

但是成员类对象的处理则不同。当我们写以下语句时：

```
Account newAcct( oldAcct );
```

这两个对象就被识别为 Account 类对象。如果 Account 类提供了一个显式的拷贝构造函数，则调用它以完成初始化，否则应用缺省的按成员初始化。

类似地，当一个成员类对象被识别出来时，则递归应用相同的过程。该类提供了显式的拷贝构造函数了吗？如果是，则调用该构造函数初始化成员类对象。否则就在成员类对象上应用缺省的按成员初始化。如果类的所有成员都是内置或复合数据类型，则依次对它们进行初始化，这样就完成了成员类对象的初始化。否则，如果有一个或多个成员本身是成员类对象，则递归应用这个过程，直到每个内置和复合数据类型都被处理完。

在我们的例子中，string 类提供了显式拷贝构造函数。通过调用该拷贝构造函数，_name 被初始化。现在我们可以认为，缺省 Account 拷贝构造函数被定义如下：

```
inline Account::
Account( const Account &rhs )
{
    _acct_nmbr = rhs._acct_nmbr;
    _balance = rhs._balance;

    // C++伪代码
    // 说明调用了一个类成员
    // 对象的拷贝构造函数
    _name.string::string( rhs._name );
}
```

Account 类的缺省按成员初始化过程现在可以正确地处理_name 的分配和释放，但是，拷贝帐号仍然不正确。因此，我们仍然必须提供一个显式的拷贝构造函数，下面的代码不是十分正确。你能看出为什么吗？

```
// 不太对
inline Account::
Account( const Account &rhs )
{
    _name = rhs._name;
    _balance = rhs._balance;
    _acct_nmbr = get_unique_acct_nmbr();
}
```

该实现不完全正确是因为我们没有区分开初始化和赋值。结果，调用的不是 string 拷贝构造函数，而是在隐式初始化阶段调用了缺省的 string 构造函数，并且在构造函数体内调用了 string 拷贝赋值操作符。修正很简单。

```
inline Account::
Account( const Account &rhs )
    : _name( rhs._name )
{
    _balance = rhs._balance;
    _acct_nmbr = get_unique_acct_nmbr();
}
```

再次强调，真正的工作是在一开始就意识到我们需要提供一个修正（两个实现的结果都是_name 持有 rhs._name 的值。只不过，第一个实现要求做两次重复工作）。一个一般性的规则是，在成员初始化表中初始化所有的成员类对象。

练习 14.13

下列哪些类需要拷贝构造函数？

(a) 含有四个 float 成员的 Point3w 表示。
(b) matrix 类，其中实际的矩阵是在构造函数中动态分配的，在析构函数中删除的。
(c) payroll 类，其中为每个对象提供一个惟一的 ID。
(d) word 类，含有一个 string 对象和行列位置对的 vector 对象。

练习 14.14

已知下列类，请为每个类实现一个拷贝构造函数，以及一个缺省构造函数和一个析构函数。

(a)
```
class BinStrTreeNode {
public:
    // ...
private:
    string _value;
    int _count;
    BinStrTreeNode *_leftchild;
    BinStrTreeNode *_rightchild;
};
```

(b)
```
class BinStrTree {
public:
    // ...
private:
    BinStrTreeNode *_root;
};
```

(c)
```
class iMatrix {
public:
    // ...
private:
```

```
        int _rows;
        int _cols;
        int *_matrix;
    };

(d) class theBigMix {
    public:
        // ...
    private:
        BinStrTree _bst;
        iMatrix _im;
        string _name;
        vector<float> *_pvec;
    };
```

练习 14.15

对 14.2 节的练习 14.3 中选择的类，判断是否需要一个拷贝构造函数。如果不需要，请说明原因。否则，请给出实现。

练习 14.16

在下边代码段中，请找出每个按成员初始化的实例：

```
Point global;
Point foo_bar( Point arg )
{
    Point local = arg;
    Point *heap = new Point( global );
    *heap = local;
    Point pa[ 4 ] = { local, *heap };
    return *heap;
}
```

14.7 按成员赋值 ※

缺省的按成员赋值（default memberwise assignment）所处理的是，用一个类对象向该类的另一个对象的赋值操作。其机制基本上与缺省的按成员初始化相同，但是它利用了一个隐式的拷贝赋值操作符来取代拷贝构造函数。例如：

```
newAcct = oldAcct;
```

在缺省情况下，用 oldAcct 的相应成员的值依次向 newAcct 的每个非静态成员赋值。在概念上就好像编译器已经生成下列拷贝赋值操作符：

```
inline Account&
Account::
    operator=( const Account &rhs )
    {
        _name = rhs._name;
        _balance = rhs._balance;
```

```
        _acct_nmbr = rhs._acct_nmbr;
    }
```

一般来说，如果缺省的按成员初始化对于一个类不合适，则缺省的按成员赋值也不合适。例如，对于原来的 Account 类的定义来说（其中_name 被声明为 char*类型），_name 和_acct_nmbr 的按成员赋值就都不合适了。

通过提供一个显式的拷贝赋值操作符的实例，可以改变缺省的按成员赋值。我们在这个操作符实例中实现了正确的类拷贝语义。拷贝赋值操作符的一般形式如下：

```
// 拷贝赋值操作符的一般形式
className&
className::
operator=( const className &rhs )
{
    // 保证不会自我拷贝
    if ( this != &rhs )
    {
        // 类拷贝语义在这里
    }

    // 返回被赋值的对象
    return *this;
}
```

这里条件测试是：

```
if ( this != &rhs )
```

应该防止一个类对象向自己赋值。因为对于“先释放与该对象当前相关的资源，以便分配与被拷贝对象相关的资源”这样的拷贝赋值操作符，拷贝自身尤其不合适。例如，考虑 Account 拷贝赋值操作符：

```
Account&
Account::
operator=( const Account &rhs )
{
    // 避免向自身赋值
    if ( this != &rhs )
    {
        delete [] _name;
        _name = new char[strlen(rhs._name)+1];
        strcpy( _name,rhs._name );
        _balance = rhs._balance;
        _acct_nmbr = rhs._acct_nmbr;
    }
    return *this;
}
```

当一个类对象被赋值给该类的另一个对象时，如

```
newAcct = oldAcct;
```

下面几个步骤就会发生：

1．检查该类，判断它是否提供了一个显式的拷贝赋值操作符。

2．如果是，则检查访问权限，判断是否在这个程序部分它可以被调用。

3．如果它不能被调用，则会产生一个编译时刻错误。否则，调用它执行赋值操作。

4．如果该类没有提供显式的拷贝赋值操作符，则执行缺省按成员赋值。

5．在缺省按成员赋值下，每个内置类型或复合类型的数据成员被赋值给相应的成员。

6．对于每个类成员对象，递归执行1到6步，直到所有内置或复合类型的数据成员都被赋值。

例如，如果我们再次修改 Account 类的定义，使_name 为一个 string 类型的成员类对象。则

```
newAcct = oldAcct;
```

会调用缺省的按成员赋值，就好像编译器为我们生成了下面的拷贝赋值操作符：

```
inline Account&
Account::
operator=( const Account &rhs )
{
    _balance = rhs._balance;
    _acct_nmbr = rhs._acct_nmbr;

    // 即使在程序员这个层次上，
    // 这个调用也是正确的
    // 等同于简短形式: _name = rhs._name
    _name.string::operator=( rhs._name );
}
```

但是，Account 类对象的缺省按成员赋值仍然不合适，同为_acct_nmbr 成员也被按成员拷贝了。我们仍然必须提供一个显式的拷贝赋值操作符，但是它以成员类 string 对象的方式来处理 name：

```
Account&
Account::
operator=( const Account &rhs )
{
    // 避免类对象向自身赋值
    if ( this != &rhs )
    {
        // 调用 string::operator=(const string& )
        _name = rhs._name;
        _balance = rhs._balance;
    }

    return *this;
}
```

如果希望完全禁止按成员拷贝的行为，那么就需要像禁止按成员初始化一样：将操作符声明为 private，并且不提供实际的定义。

一般来说，应该将拷贝构造函数和拷贝赋值操作符视为一个个体单元。因为在我们需要其中一个的时候，往往也需要另外一个。而试图禁止一个的时候，也很可能需要禁止另一个：

练习 14.17

请为 14.6 节的练习 14.14 中定义的每一个类提供拷贝赋值操作符。

练习 14.18

对于 14.2 节的练习 14.3 中选择的类，请判断是否需要一个拷贝赋值操作符。如果是，请给出。否则，请说明原因。

14.8 效率问题 ※

一般来说，通过指针或引用向一个函数传递一个类对象比传值更有效率。例如，函数原型：

```
bool sufficient_funds{ Account acct, double };
```

要求每个调用都用“实际被传递进来的 Account 对象”按成员初始化参数 acct。下面是修订后的函数版本：

```
bool sufficient_funds( const Account, double );
bool sufficient_funds( const Account &acct, double );
```

它只要求拷贝 Account 对象的地址值，而不会发生类的初始化操作（关于引用和指针参数之间的关系的讨论见 7.3 节）。

虽然返回类对象的指针和引用也比按值返回类对象有效，但是，正确的编程实现却非常困难。例如，考虑下面的加法操作符：

```
// 完成了工作，但是，对于大型 matrix 对象
// 可能效率低得不能接受
Matrix
     opelator+( const Matrix& m1, const Matrix& m2 )
{
     Matrix result;

     // 算术运算

     return result;
}
```

重载的加法操作符允许用户写：

```
Matrix a, b;

// ...
// 都调用 operator+()
Matrix c = a + b;
a = b + c;
```

但是，如果 Matrix 是个很大、很复杂的类，则按值返回 result 可能在效率上难以被接受——尤其是，如果这个操作被频繁调用的话，那么就会发现它是个性能瓶颈。

尽管下面修改后的实现被证明显著地提高了系统性能：

```
// 更有效，但是在返回之后地址无效
// —可能导致运行时刻程序失败
```

```
Matrix&
     operator+( const Matrix& m1, const Matrix& m2 )
{
     Matrix result;
     // 做加法操作 ...
     return result;
}
```

但是，它也会导致程序运行时刻错误：_result 的地址值在函数完成之后是未定义的。（实际上我们返回一个指向局部对象的引用，该对象在函数完成后不再存在。）

我们返回的地址值必须在函数完成后仍保持有效，尽管下面的实现提供了一个永久的地址值：

```
// 没有办法保证内存不会丢失
// 因为 Matrix 可能很大，所以这种丢失可能很严重
Matrix&
     operator+( const Matrix& m1, const Matrix& m2 )
{
     Matrix &result = new Matrix;

     // 做加法操作 ...

     return *result;
}
```

但是它也导致了严重的内存泄漏：在该对象最后一次被使用之后，没有程序能够对其应用 delete 表达式。因此，在实践中，它也是不能被接受的。

从程序设计的角度，一种解决方案是重新定义加法操作符，使其含有第三个引用参数，以存储结果：

```
// 这提供了我们要求的效率
// 但是用户用起来却不直观。
void
mat_add( Matrix &result,
     const Matrix& m1, const Matrix& m2 )
{
     // 直接计算到 result 中
}
```

这解决了性能问题，但是该类不能再以操作符的语法被使用了。也不能使用它的一般性功能，包括初始化对象：

```
// 不再支持
Matrix c = a + b;
```

或使其作为子表达式参与计算：

```
// 也不再支持
if ( a + b > c ) ...
```

C++语言不能有效地返回一个类对象，这被视为 C++语言的一个重大缺陷。有人提出了一种解决方案，即，对语言进行扩展，在语言层次上指定函数返回的类对象的名字。例如：

```
Matrix
     operator+( const Matrix& m1, const Matrix& m2 )
```

```
{
    Matrix result;
    //...
    return result;
}
```

然后编译器会在内部重写该函数，使其含有第二个引用参数：

```
// 内部重写的函数
// 在提出的语言扩展之下
void
operator+( Matrix &result,
    const Matrix& m1, const Matrix& m2 )
{
    // 直接计算到 result 中
}
```

并且把所有使用函数的地方都转换成在目的类对象上直接计算结果。例如：

```
Matrix c = a + b;
```

被内部转换为：

```
Matrix c;
operator+( c, a, b );
```

这种命名返回值的扩展从未成为语言规范的一部分——但却是一种优化。它使编译器能够知道一个类对象被返回，并且无需显式的语言扩展就可以提供返回值的转换。下面给出了一个一般形式的函数：

```
classType
    functionName( paramList )
{
    classType namedResult;

    // do the work

    return n;
}
```

编译器会在内部把这个函数及其用法转换成下面的形式：

```
void
functionName( classType &namedResult, paramList )
{
    // 直接计算到 namedResult 中
}
```

从而消除了类对象的按值返回，以及对类的拷贝构造函数调用的需要。为了触发它，被返回的类对象必须与函数中每个返回点的对象名相同。

关于 C++类对象的效率问题，还有最后一点要说明。比如：

```
Mactix c = a + b;
```

这样的类对象的初始化总是比赋值更有效率。例如，虽然程序结果完全相同，但是写成

```
Matrix c;
c = a + b;
```

就会要求更多的计算才能得到结果。类似地，下面的写法：

```
for ( int ix = 0; ix < size -2; ++ix ) {
    Matrix matSum = mat[ix] + mat[ix+1];

    // ...
}
```

比以下写法有效很多：

```
Matrix matSum;
for ( int ix = 0; ix < size -2; ++ix ) {
    matSum = mat[ix] + mat[ix+1];

    // ...
}
```

赋值的开销总是要多一些，这是因为一般情况下，我们并不能直接用被返回的局部对象代替赋值的目的对象。也就是说，虽然：

```
Point3d p3 = operator+( p1, p2 );
```

可以被安全地转换为：

```
// C++伪代码
Point3d p3;
operator+( p3, p1, p2 );
```

但是把：

```
Point3d p3;
p3 = operator+( p1, p2 );
```

转换成：

```
// C++ 伪代码
// 在赋值的情况下不安全
operator+( p3, p1, p2 );
```

并不安全。

问题是，被转换的函数要求传递给它的对象代表了一块未使用的存储区（raw storage）。为什么？因为对该对象做的第一件事情就是，对其应用构造函数。如果被传递的对象已经被构造，则可能因为两次构造而造成语义上潜在的灾难。

一个正在被初始化的对象一定代表了一块未被使用的存储区 raw storage，而正在被赋值的对象。如果该类声明了相关的构造函数（这正是我们所考虑的情况），则它一定不会代表一块未被使用的存储区，因此不能被直接安全地传递给函数。

相反，编译器必须以临时类对象的形式创建未被使用的存储区，并把这个对象传递给函数，然后将临时对象按成员赋值给目标对象。最后，如果该类有相关的析构函数，还必须把析构函数应用到临时对象上。例如：

```
Point3d p3;
p3 = operator+( p1, p2 );
```

可能转换成：

```
// C++ 伪代码
Point3d temp;
operator+( temp, p1, p2 );
p3.Point3d::operator=( temp );
```

```
temp.Point3d::~Point3d();
```

Michael Tiemann，GNU C++编译器的作者，提出了一个专门用语：**返回值语言扩展**（return value language extension）。他对这部分内容的讨论可以在［LIPPMAN96b］中找到。在本书的姐妹书《Inside the C++ Object Model》（［LIPPMAN96a］）中给出了有于本章介绍的话题的更高级的讨论。

15

重载操作符和用户定义的转换

在第 15 章中，我们将了解两种特殊类型的函数：重载操作符和转换函数。这些函数使得类（class）类型的对象能够被用在表达式中，而且使用方式与内置类型同样直观。本章将首先给出操作符重载的一般概念以及设计考虑。然后再给出有特殊访问权限的类的友元（friend）的概念，以及对于友元必要性（尤其是在实现重载操作符时）的讨论。接着，本章还将介绍在定义类类型时要求特别注意的特殊重载操作符：赋值、下标、调用、成员访问箭头、递增和递减，以及类特有的 new 和 delete 操作符。第二种特殊类型的函数是本章的下一个焦点：成员转换函数，它为一个类类型定义了一组“标准转换”。当类对象被用作函数实参；或者内置和重载操作符的操作数时，编译器会隐式地应用这些转换函数。在本章的最后将讨论更高级的话题——函数重载解析规则，它涉及类实参、类成员函数以及重载操作符。

15.1 操作符重载

正如在上一章的例子中所见的，操作符重载使得程序员能够为类类型的操作数定义预定义的操作符版本（如第 4 章所讨论）。例如，在 3.15 节给出的 String 类就定义了许多重载的操作符，下面是 String 类的定义：

```
#include <iostream>

class String;
istream& operator>>( istream &, String & );
ostream& operator<<( ostream &, const String & );

class String {
public:
    // 构造函数的重载集合
    // 提供自动初始化
    String( const char * = 0 );
    String( const String & );

    // 析构函数：自动释放初始化的对象
```

```
        ~String();

        // 赋值操作符的重载集合
        String& operator=( const String & );
        String& operator=( const char * );

        // 重载的下标操作符
        char& operator[]( int ) const;

        // 等于操作符的重载集合
        // str1 == str2;
        bool operator==( const char * ) const;
        bool operator==( const String & ) const;

        // 成员访问函数
        int size() { return _size; }
        char* c_str() { return _string; }

    private:
        int _size;
        char *_string;
    };
```

String 类有三个重载函数集，第一个集合为 String 类定义了赋值操作符：

```
// 赋值操作符的重载集合
String& operator=( const String & );
String& operator=( const char * );
```

第一个赋值操作符是拷贝赋值操作符，它支持从一个 String 类对象向另一个该类对象的赋值操作。关于拷贝赋值操作符我们已经在 14.7 节详细讨论过。第二个赋值操作符支持从 C 风格字符串向 String 类型对象的赋值操作，如下所示：

```
String name;
name = "Sherlock"; // use of operator=( char * )
```

我们将在 15.3 节介绍非拷贝赋值操作符的赋值操作符。

第二个重载操作符集合定义了一个操作符——下标操作符：

```
// 重载的下标操作符
char& operator[]( int ) const;
```

这个操作符使得程序能够像索引内置数组对象一样索引 String 类对象：

```
if ( name[0] != 'S' )
    cout << "oops, something went wrong\n";
```

我们将在 15.4 节更详细地介绍重载的下标操作符。

第二个重载操作符集合为 String 类对象定义了等于操作符。使得程序可以比较两个 String 类对象是否相等，或者比较一个 String 类对象是否等于一个 C 风格字符串：

```
// 等于操作符的重载集合
// str1 == str2;
bool operator==( const String & ) const;
```

```
bool operator==( const char * ) const;
```

我们将在下一小节详细介绍这个操作符。

重载的操作符使得类类型对象可以与第 4 章定义的操作符一起被使用，使得对于类对象的操纵与内置类型的对象一样直观。例如，如果我们想定义一个操作来支持把两个 String 类对象连接起来。我们可能会决定把这个新的操作实现为一个被称为 concat()的成员函数。但是，为什么选择名字 concat()，而不是 append()呢？尽管我们选择的名字有逻辑性和助记性，但是，用户还是可能会忘记我们选择的名字。如果我们把它定义为一个重载的操作符，则记住操作的名字通常会很容易。例如，我们不是用 concat()，我们更喜欢把新的 String 操作命名为 operator+=()。这个新操作符可以被如上使用：

```
#include "String.h"

int main() {
    String name1 = "Sherlock";
    String name2 = "Holmes";
    name1 += " ";
    name1 += name2;
    if ( ! ( name1 == "Sherlock Holmes" ) )
        cout << "concatenation did not work\n";
}
```

重载的操作符在类体中被声明，声明方式同普通成员函数一样，只不过它的名字包含关键字 operator，以及紧随其后的一个预定义操作符（该操作符必须来自 C++预定义操作符的一个子集，见表 15.1）。在 String 类中可以如下声明 operator+=()：

```
class String {
public:
    // += 操作符的重载集合
    String& operator+=( const String & );
    String& operator+=( const char * );

    // ...
private:
    // ...
};
```

并定义如下：

```
#include <cstring>

inline String& String::operator+=( const String &rhs )
{
    // 如果 rhs 引用的 String 不为空
    if ( rhs._string )
    {
        String tmp( *this );

        // 创建足够大的存储区
        // 以便包含被连接之后的 String
        _size += rhs._size;
        delete[] _string;
```

```
            _string = new char[ _size + 1 ];

            // 首先，把原来的 String 拷贝到新的存储区中，
            // 然后附加上 rhs 所指的 String
            strcpy( _string, tmp._string );
            strcpy( _string + tmp._size, rhs._string );
        }
        return *this;
    }

    inline String& String::operator+=( const char *s )
    {
        // 如果 s 不是空指针
        if ( s )
        {
            String tmp( *this );
            // 创建足够大的存储区
            // 以便包含被连接之后的 String
            _size += strlen( s );
            delete[] _string;
            _string = new char[ _size + 1 ];

            // 首先把原来的 String 拷贝到新的存储区中
            // 然后，附加上 s 所指的 C 风格字符串
            strcpy( _string, tmp._string );
            strcpy( _string + tmp._size, s );
        }
        return *this;
    }
```

15.1.1 类成员与非成员

让我们再仔细地看一看 String 类的等于操作符。第一个操作符使我们能够比较两个 String 类对象是否相等，第二个允许我们比较一个 String 类对象是否等于一个 C 风格的字符串。例如：

```
    #include "String.h"
    int main() {
        String flower;

        // 设置 flower
        if ( flower == "lily" ) // ok
            // ...
        else
            if ( "tulip" == flower )  // 错误
            // ...
    }
```

在 main()中，等于操作符的第一个用法调用了 String 类重载的 operator==(const char*)。但是，第二次使用等于操作符却导致了一个编译错误。怎么会这样呢？

问题在于，只有在左（left）操作数是该类类型的对象时，才会考虑使用作为类成员的重

载操作符。因为这里的左操作数不是类类型，所以编译器试图找到一个内置操作符，它可以有一个 C 风格字符串的左操作数和一个 String 类型的右操作数。事实上并不存在这样的操作符，所以编译器为 main()中第二次使用等于操作符就会产生一个错误信息。

但是，你可能会说，我们可以用类构造函数，从一个 C 风格字符串创建一个 String 类的对象。为什么编译器不能隐式地做如上的转换呢？

```
if ( String( "tulip" ) == flower )        // ok: 调用成员操作符
```

简要的答案是效率。重载的操作符并不要求两个操作数的类型一定相同。例如，一个 Text 类定义了如下的等于操作符作为成员函数：

```
class Text {
public:
    Text( const char * = 0 );
    Text( const Text & );

    // 等于操作符的重载集合
    bool operator==( const char * ) const;
    bool operator==( const String & ) const;
    bool operator==( const Text & ) const;

    // ....
};
```

在 main()中的表达式可以重写如下：

```
if ( Text( "tulip" ) == flower )          // 调用 Text::operator==()
```

因此，为了给这个比较操作找到等于操作符，编泽器必须查着所有的类定义，以找到所有能够把左操作数转换成类类型的构造函数，然后再为每一个类类型找到相关的重载等于操作符，看是否有一个能执行等于操作。接着，编译器还需要判断哪一个“构造函数和等于操作符”的组合对于右操作数是最佳匹配！如果要求编译器这样做的话，那么，编译 C++程序所需时间会显著增加。因此，编译器只考虑在左操作数的类中定义的成员重载操作符（以及在其基类中定义的重载操作符，正如我们将在第 19 章中看到的那样）

但是，声明非类成员的重载操作符也是可以的。非类成员的重载操作符对于 main()中的比较而言是错误的。对于这个比较，C 风格字符串是左操作数，如果我们用名字空间域中声明的等于操作符（如下所示）来代替 String 中的成员等于操作符，则这个比较操作就会有效：

```
bool operator==( const String &, const String & );
bool operator==( const String &, const char * );
```

我们注意到这些全局重载操作符比成员重载操作符多了一个参数。对于成员操作符，隐式的 this 指针被用作隐式的第一个参数。例如，对于成员操作符，如下表达式：

```
flower == "lily"
```

被编译器重写为

```
flower.operator==( "lily" )
```

在成员重载操作符的定义中，我们通过 this 指针可以引用左操作数 flower（关于 this 指针在 13.4 节介绍）。对于全局重载操作符，代表左操作数的参数必须被显式指定。

有了针对 String 类的全局重载操作符，如下表达式：

```
flower == "lily"
```

将调用操作符：

```
bool operator==( const String &, const char * );
```

等于操作符的第二个用法将调用哪个操作符？

```
"tulip" == flower
```

我们没有定义下列重载操作符：

```
bool operator==( const char *, const String & );
```

我们需要这样做吗？我们可以这样做，但不是必需的。当一个重载操作符是一个名字空间的函数时，对于操作符的第一个和第二个参数，即等于操作符的左和右两个操作数都会考虑转换。这意味着编译器将解释等于操作符的第二个用法如下：

```
operator==( String("tulip") , flower );
```

并调用下列重载操作符执行比较：

```
bool operator==( const String &, const String & );
```

好。现在你可能想知道为什么我们要提供第二个重载操作符：

```
bool operator==( const String &, const char * );
```

从 C 风格字符串到 String 类的类型转换也可以被应用到右操作数上。如果我们只定义一个名字空间重载操作符，它接受两个 String 类的操作数，那么函数 main()也能够没有错误地通过编译：

```
bool operator==( const String &, const String & );
```

我们是只提供这样一个重载操作符，还是再提供另外两个操作符：

```
bool operator==( const char *, const String & );
bool operator==( const String &, const char * );
```

这将取决于从 C 风格字符串到 String 的类型转换开销，即它会取决于应用程序中调用 String 构造函数引起的额外开销。如果我们预料到会频繁地使用等于操作符来比较 C 风格字符串和 String 型的对象，则提供所有这三个名字空间全局操作符就不失为一个好主意（我们将在下一节“友元”中对性能进行更多讨论）。

我们将在 15.9 节中更详细地讨论使用构造函数的类类型转换；此外还将在 15.10 节重新回顾函数重载解析过程，着重讨论类类型的转换，以及在 15.12 节介绍涉及重载操作符的函数重载解析过程。

那么，一般应该怎样决定是把一个操作符声明为类成员还是名字空间成员呢？在某些情况下，程序员没有选择的余地。

- 如果一个重载操作符是类成员，那么只有当跟它一起被使用的左操作数是该类的对象时，它才会被调用。如果该操作符的左操作数必须是其他的类型，那么重载操作符必须是名字空间成员。

- C++要求，赋值（=）、下标（[]）、调用（()）和成员访问箭头（->）操作符必须被定义为类成员操作符。任何把这些操作符定义为名字空间成员的定义都会被标记为编译时刻错误。例如：

```
// 错误：必须是类成员
char& operator[]( String & ,int ix );
```

我们将在 15.3 节里更详细地介绍赋值操作符，在 15.4 节介绍下标操作符，在 15.5 节介绍调用操作符，在 15.6 节介绍成员访问操作符箭头->。

除此之外，由类设计者选择把操作符声明为一个类成员还是一个名字空间成员。如果有一个操作数是类类型，如 String 类的情形，那么对于对称操作符，比如等于操作符，最好定义为名字空间成员。

在结束本小节之前，让我们为 String 类定义名字空间等于操作符：

```
bool operator==( const String &str1, const String &str2 )
{
    if ( str1.size() != str2.size() )
        return false;

    return strcmp( str1.c_str(), str2.c_str() ) ? false : true;
}
```

15.1.2 重载操作符的名字

只有在 C++预定义操作符集中的操作符才可以被重载，表 15.1 列出了可以被重载的操作符。

表 15.1 可以被重载的操作符

+	-	*	/	%	^	&	|	~
!	,	=	<	>	<=	>=	++	--
<<	>>	==	!=	&&	||	+=	-=	/=
%=	^=	&=	|=	*=	<<=?	>>=	[]	()
->	->*	new	new[]	delete	delete[]			

类的设计者不能声明一个没有出现在表格中的重载操作符。例如，如果声明了一个重载操作符 operator**以提供指数算法，就会产生编译错误。

下列四个 C++操作符不能被重载：

```
// 不能被重载的操作符
::.*.?:
```

对于内置类型的操作符，它的预定义意义不能被改变。例如，内置整型加操作符不能被

取代为检查溢出：

```
// 错误：不能为 int 重新定义内置的操作符
int operator+( int, int );
```

也不能为内置数据类型定义其他的操作符。例如，有两个数组类型操作数的 operator+不能被加入到内置操作集中。

程序员只能为类类型或枚举类型的操作数定义重载操作符。我们可以这样来实现：把重载操作符声明为类的成员，或者声明为名字空间成员，同时至少有一个类或枚举类型的参数（按值传递或按引用传递）

预定义的操作符优先级（4.13 节讨论了操作符优先级）不能被改变。无论类类型还是操作符的实现：

```
x == y + z;
```

总是在 operator==之前执行 operator+。与预定义操作符一样，在使用重载操作符时，可以用小括号来改变优先级。

操作符预定义的操作数个数（arity）必须被保留。例如，一元的逻辑非操作符不能被定义为针对两个 String 对象的二元操作符。下面是一个非法的实现，将导致编译时刻错误：

```
// 非法：！是一元操作符
bool operator!( const String &s1, const String &s2 )
{
    return( strcmp( s1.c_str(), s2.c_str() ) != 0 );
}
```

对于内置类型，四个预定义的操作符（“+”、“-”、“*”和“&”）既可被用作一元操作符，也可被用作二元操作符。操作符的这两种版本都可以被重载。

除了对 operator()外，对其他重载操作符提供缺省实参都是非法的。

15.1.3 重载操作符的设计

赋值、取地址以及逗句操作符对于类类型的操作数有预定义的意义。对于类操作数，这些操作符也可以被重载。为使其他操作符在被应用到类类型的操作数上时也有意义，类的设计者必须显式地定义它。一个类最终需要提供哪些操作符，是由该类预期的用途来决定的。

我们总是以定义一个类的公有接口开始设计。一个类必须为用户提供哪些操作呢？这些将是公有成员函数的最小集合。一旦定义了这个集合，就可以考虑应该把哪些操作定义为重载操作符。

一旦定义了类的公有接口，我们就在每个操作和操作符之间寻找逻辑匹配关系：

- isEmpty()变成逻辑非操作符，operator!()。
- isEqual()变成等于操作符，operator==()。
- copy()变成赋值操作符，operator=()。

每个操作符都有与预定义用法相关联的意义。例如，二元+，就被强烈地认为与加法有关。在类类型中，将二元+映射成类类型中某个类似的操作可以提供一个方便的助记符。例如，matrix 类型的加法（把两个 matrix 型的对象相加）就是二元+的一个合适的扩展。

对于操作符重载，最糟糕的错误用法不是像把减法定义成 operator-()这样的做法。任何

一个有责任心的程序员都不会这样做。最糟糕的误用是，操作符的操作对于用户来说含义不清。

在这种意义上，所谓具有二义性的操作符，意思是指对于大量不同的解释它都同等程度地支持。对于 String 类的用户来说，operator+()应该被认为是一个连接操作符，而无需更多的解释。当重载操作符的意义不是十分明显时，最好的做法是不要提供它。

对于内置类型的操作数，在复合操作符和相应独立操作符的意义之间存在等价关系（例如；在+后跟。和复合操作符+=之间的等价）。对于类而言，我们必须显式地定义这种等价关系。例如，如果为 String 类定义了 operator+()和 openator=()，分别支持连接操作和按成员拷贝操作，

```
String s1( "C" );
String s2( "++" );
s1 = s1 + s2; // s1 == "C++"
```

这些重载操作符并不隐式地支持等价的复合操作符。perator+=：

```
s1 += s2;
```

该复合赋值操作符必须被显式定义。如果定义了，则应该提供期望的意义。

练习 15.1

为什么下面的语句没有调用重载的 operator==(const String&, const String&)?

```
"cobble" == "stone"
```

练习 15.2

请提供重载的不等于操作符，它可以处理下列三种情况：

```
String != String
String != C 风格字符串
C 风格字符串 != String
```

说明选择实现一个或多个操作符的原因。

练习 15.3

在 13.3 节、13.4 节和 13.6 节中实现的 Screen 类的成员函数中，哪些适合于被做成重载操作符？

练习 15.4

请说明为什么 3.15 节中的 String 类定义的输入和输出操作符被声明为全局函数而不是成员函数。

练习 15.5

请为第 13 章中定义的 Screen 类实现重载的输入和输出操作符。

15.2 友元

让我们回顾一下上节介绍的重载的等于操作符的定义，它是为名字空间域中定义的String 类而提供的。针对两个 String 对象的等于操作符如下：

```
bool operator==( const String &str1, const String &str2 )
{
    if ( str1.size() != str2.size() )
        return false;
    return strcmp( str1.c_str(), str2.c_str() ) ? false : true;
}
```

把这个定义与被定义为成员函数的操作符定义相比较：

```
bool String::operator==( const String &rhs ) const
{
    if ( _size != rhs._size )
        return false;
    return strcmp( _string, rhs._string ) ? false : true;
}
```

你看到区别了吗？我们注意到必须要修改函数定义内部对于 String 类私有数据成员的引用方式。因为新的等于操作符是全局函数，不是类成员函数，它不能直接引用 String 的私有数据成员。它使用访问成员函数 size()和 c_str()来获得 String 对象的大小以及底层的 C 风格字符串。

另外一种可能的实现是；把全局等于操作符声明为 String 类的友元（friend）。通过把函数或操作符声明为友元，一个类可以授予这个函数或操作符访问其非公有成员的权利。

友元声明以关键字 friend 开始，它只能出现在类定义中。因为友元不是授权类的成员，所以它不受其所在类的声明区域（public、private 和 protected）的影响。这里我们选择把所有友元声明组织在一起并放在类头之后：

```
class String {
    friend bool operator==( const String &, const String & );
    friend bool operator==( const char *, const String & );
    friend bool operator==( const String &, const char * );
public:
    // ... String 类中的其他部分
};
```

String 类中的三个友元声明把全局域中声明的三个重载的比较操作符（在上节介绍）声明为 String 类的友元。

既然这些等于操作符已经被声明为友元，那么它们的定义就可以直接引用 String 的私有成员了：

```
// friend 操作符: 直接引用 String 的私有成员
// friend operators: refer to String private members directly
bool operator==( const String &str1, const String &str2 )
{
    if ( str1._size != str2._size )
        return false;
    return strcmp( str1._string, str2._string ) ? false : true;
```

```
}
inline bool operator==( const String &str, const char *s )
{
    return strcmp( str._string, s ) ? false : true;
}
// 以下略
```

有人可能会说，在这种情况下，由于 c_str()和 size()是内联的，它们提供了等价的效率，并且保留了成员封装，所以没必要直接访问_size 和_string。这是对的，使用成员访问函数而不是直接访问成员，并不总是意味着它的效率较低。由于存在这些访问函数，所以没有必要把等于操作符声明为 String 类的友元。

那么，我们怎样判断一个非类成员的操作符应该是类的友元，还是应该使用成员访问函数呢？一般来说，类的实现者应该尽量使得名字空间函数和访问类内部表示的操作符的数目最小化。如果已经提供了访问成员函数并且它们具有等同的效率，那么最好是使用这些成员函数，并且把名字空间操作符与类表示中的变化隔离开。但是，如果类的实现者决定不为该类的某些私有成员提供访问成员函数，而且名字空间操作符需要引用这些私有成员才能完成它们的操作，那么就必须使用友元机制。

友元声明的最常见用法是，允许非成员的重载操作符访问一个“视其为朋友”的类的私有成员。原因是，除了提供左和右操作数的对称性外，非成员的重载操作符就像成员函数一样，能够完全访问一个类的私有成员。

虽然友元声明的主要用处是在重载操作符上，但是，在某些情况下，一个名字空间函数、另一个在此之前被定义的类的成员函数、或者一个完整的类必须声明为友元。在使一个类成为另一个类的友元时，友元类的成员函数被赋予访问授权类的非公有成员的权利。下面我们将更详细地了解函数而不是操作符的友元声明。

一个类必须把“它希望与之建立友元关系”的重载函数集中的每个函数都声明为友元。例如：

```
extern ostream& storeOn( ostream &, Screen & );
extern BitMap& storeOn( BitMap &, Screen & );

// ...
class Screen
{
    friend ostream& storeOn( ostream &, Screen & );
    friend BitMap& storeOn( BitMap &, Screen & );
    // ...
};
```

如果一个函数操纵两个不同类类型的对象，而且该函数需要访问这两个类的非公有成员，则这个函数可以被声明为这两个类的友元，或者作为一个类的成员函数，并声明为另一个类的友元。让我们来看一看怎样做。

如果我们决定一个函数必须被声明为两个类的友元，则友元声明如下：

```
class Window; // 只声明
class Screen {
    friend bool is_equal( Screen &, Window & );
    // ...
```

```
};
class Window {
     friend bool is_equal( Screen &, Window & );
     // ...
};
```

如果我们决定该函数必须作为一个类的成员函数，并又是另一个类的友元，则成员函数声明和友元声明如下：

```
class Window;
class Screen {
public:
     // copy 是类 Screen 的成员
     Screen& copy( Window & );

     // ...
};

class Window {
     // copy 是类 Window 的一个友元
     friend Screen& Screen::copy( Window & );

     // ...
};
```

只有当一个类的定义已经被看到时，它的成员函数才能被声明为另一个类的友元。这并不总是能够做到的。例如，如果 Screen 类必须把 Window 类的成员函数声明为友元，而 Window 类必须把 Screen 类的成员函数声明为友元，该怎么办呢？在这种情况下，可以把整个 Window 类声明为 Screen 类的友元。例如：

```
class Window;
class Screen {
     friend class Window;
     // ...
};
```

Screen 类的非公有成员现在可以被 Window 的每个成员函数访问。

练习 15.6

请重新实现例 15.5 中为 Screen 类定义的输入和输出操作符，使其成为友元函数，并修改它们的定义，直接访问类的私有成员，哪一个实现更好一些？说明原因。

15.3 操作符=

一个类对象向该类的另一个对象的赋值可通过拷贝赋值操作符来执行。这个特殊的操作符已在 14.7 节描述过。

我们也可以为一个类（class）类型定义其他的赋值操作符。如果一个类类型的对象被赋以一个不是它自己类类型的值，那么它可以定义接受这种其他类型参数的赋值操作符。例如：为了支持用 C 风格字符串向 String 类对象的赋值，如：

```
String car ("Volks");
car = "Studebaker";
```

我们给出如上能够接受 const char*型的赋值操作符，该操作符已经在前面的 String 类中被声明：

```
class String {
public:
    // char* 的赋值操作符
    String& operator=( const char * );

    // ....
private:
    int _size;
    char *_string;
};
```

这个赋值操作符被实现如下。如果 String 对象被赋以一个空指针值，则 String 对象被设置为空，否则 String 对象的内容是被赋进来的 C 风格字符串的拷贝：

```
String& String::operator=( const char *sobj )
{
    // sobj 是个空指针
    if ( ! sobj ) {
        _size = 0;
        delete[] _string;
        _string = 0;
    }
    else {
        _size = strlen( sobj );
        delete[] _string;
        _string = new char[ _size + 1 ];
        strcpy( _string, sobj );
    }
    return *this;
}
```

_string 引用了参数 sobj 指向的 C 风格字符串的拷贝。为什么是一个拷贝？因为我们不能直接把 sobj 赋给_string：

```
_string = sobj;          // 错误：类型不匹配
```

sobj 是一个指向 const 的指针，指向 const 的指针不能被赋给一个指向非 const 的指针（如 3.5 节中说明的）。若我们决定如下定义赋值操作符：

```
String& String::operator=( char *sobj ) { // ...
```

它现在允许_string 直接引用 sobj 指向的 C 风格字符串，但是这又产生了其他的问题。记住，一个 C 风格字符串的类型是 const char*。把参数定义为指向非 const 的指针禁止了下面这样的赋值表达式（我们通常会这样做）！

```
car = "Studebaker";    // operator=( char * ) 不允许这样的赋值
```

我们没有选择。参数必须是 const char*，以便允许把 C 风格字符串赋给 String 对象。如果_string 能够直接引用 sobj 指向的对象，那么还有其他问题。我们不知道 sobj 指向什么。它可能引用一个字符串数组，并且该字符串数组能够以 String 对象未知的方式被修改。例

如：

```
char ia[] = { 'd', 'a', 'n', 'c', 'e', 'r' };
String trap = ia;       // trap._string 指向 ia
ia[3] = 'g';            // 不是我们希望的：改变 ia 和 trap._string
```

如果 trap._string 可以直接引用 ia，那么对象 trap 会有很令人吃惊的行为：在没有调用任何 String 成员函数的情况下，它的值也会被改变。因此，我们认为重新分配内存来保存_string 引用的 C 风格字符串的值，可以使 String 类对象少一些令人吃惊的行为。

注意，我们的赋值操作符使用了 delete 表达式。_string 指向一个在堆中被分配的字符数组。为了防止内存泄漏，在_string 指向“被分配来保存新字符串值的内存”之前，_string 原来引用的 C 风格字符串被通过 delete 表达式释放。因为_string 指向一个字符数组，所以必须使用 delete 表达式的数组版本（关于数组 delete 表达式在 8.4 节讨论）

关于赋值操作符，要注意的最后一件事情是：赋值操作符的返同类型是 String 类的一个引用。为什么我们要将这个赋值操作符为声明返回一个引用呢？因为对于内置类型，赋值操作符可以被串联在一起，如下所示：

```
// 赋值操作符串
int iobj, jobj;
iobj = jobj = 63;
```

赋值操作符是从右到左结合的。上面赋值的顺序如下：

```
iobj = (jobj = 63);
```

我们希望为 String 类对象的赋值保留这种行为，例如，支持：

```
String verb, noun;
verb = noun = "count";
```

在这个串中的第一个赋值调用前面为 const char*定义的赋值操作符。这个赋值的结果的类型必须能作为 String 类拷贝赋值操作符的实参。因此，即使赋值操作符的参数是 const char*，但是它的返回类型仍然是 String 类的引用。

赋值操作符也可以被重载。在我们的 String 类中，重载的赋值操作符集如下：

```
// 赋值操作符的重载集合
String& operator=( const String & );
String& operator=( const char * );
```

对每一种必须被赋给一个 String 对象的类型，都可能有一个赋值操作符，但是，每个赋值操作符都必须被定义为类的一个成员函数。

15.4 操作符[]

我们可以为“表示容器抽象并能够获取其单独元素的类”定义下标操作符 operator[]()。我们的 String 类、在第二章中给出的 IntArray 类、或者 C++标准库中定义的 vector 类模板，都是容器类的例子，对它们声明下标操作符很有意义。下标操作符必须被定义为类的成员函数。

String 类的用户需要对类成员_string 的单个字符进行读写。我们希望支持 String 类对象

的下列用法：

```
String entry( "extravagant" );
String mycopy;
for ( int ix = 0; ix < entry.size(); ++ix )
    mycopy[ ix ] = entry[ ix ];
```

下标操作符必须能够出现在一个赋值操作符的左右两边。为了能在左边出现，它的返回值必须是一个左值。这可以通过把返问类型指定为一个引用来实现：

```
#include <cassert>

inline char&
String::operator[]( int elem ) const
{
    assert( elem >= 0 && elem < _size );
    return _string[ elem ];
}
```

下标操作符的返回值是被索引的元素的左值，这是它能够出现在赋值的左边的原因。例如，下列语句为 color._string 的第 0 个元素赋一个字符：

```
String color( "violet" );
color[0] = 'V';
```

注意，在下标操作符的定义中，它对于接收到的索引值进行边界检查。这里我们决定用 c 库函数 assert()执行这种检查。我们也可以抛出一个异常，来指明 elem 值为负或大于_string 指向的 C 风格字符串的长度（关于异常处理和 throw 表达式在第 11 章讨论）。

15.5 操作符 operator()

我们可以为类类型的对象重载函数调用操作符。我们在 12.3 节介绍函数对象时，已经看到了这种重载操作符的用法。如果一个类类型被定义来表示一个操作时，则可以为这个类类型重载函数调用操作符，以便调用这个操作。例如，absInt 类被定义为将“取一个 int 型操作数的绝对值的操作”封装起来：

```
class absInt {
public:
    int operator()( int val ) {
        int result = val < 0 ? - val : val;
        return result;
    }
};
```

重载的 operator()必须被声明为成员函数，它的参数表可以有任意数目的参数，且参数类型可以是 7.2 节和 7.3 节中给出的允许被作为函数参数的任何类型。重载的 operator()的返回值可以是 7.2 节和 7.4 节中允许被作为函数返问值类型的任何类型。

我们通过向一个类类型的对象应用一个实参表，调用该类重载的 operator()操作符。我们将会看到，absInt 类的重载的 operator()怎样被第 12 章中定义的一个泛型算法使用。下面的例子调用了泛型算法 transform()，并把 absInt 定义的操作应用在向量 ivec 所含有的所有元素

上：即，把每个向量元素设置为其绝对值：

```
#include <vector>
#include <algorithm>

int main() {
    int ia[] = { - 0, 1, - 1, - 2, 3, 5, - 5, 8 };
    vector< int > ivec( ia, ia+8 );

    // 把 ivec 的每个元素设置为其绝对值
    transform( ivec.begin(), ivec.end(), ivec.begin(), absInt() );

    // ....
}
```

transform()的第一个和第二个实参指示了 absInt 操作被应用的元素范围。第三个实参指向“被用来存储 absInt 操作结果的向量”的开始。

transform()的第四个实参是一个 absInt 类的临时对象，它通过调用 absInt 的缺省构造函数来创建。main()调用的泛型算法 transform()的实例看起来像这样：

```
typedef vector<int>::iterator iter_type;

// transform() 的实例把
// 操作 absInt 应用到 int 型
// vector 的所有元素上
iter_type transform( iter_type iter, iter_type last,
                     iter_type result, absInt func )
{
    while ( iter != last )
        *result++ = func( *iter++ ); // 调用 absInt::operator()
    return iter;
}
```

func 是一个类类型的对象，它的类型代表了 absInt 操作，该操作把 int 型的值设置为其绝对值。对象 func 被用来调用 absInt 类的重载的 operator()。传递给这个重载操作符的实参是*iter，它指向我们想获取其绝对值的那个向量元素。

15.6 操作符->

我们也可以为类类型的对象重载成员访问操作符箭头，它必须被定义为一个类的成员函数。它的作用是赋予一个类类型与指针类似的行为。它通常被用于一个代表“智能指针（smart pointer）”的类类型。也就是说，一个类的行为很像内置的指针类型，但是支持某些额外的功能。

例如，假设我们想定义一个类类型来代表一个指向 Screen 类对象的指针，这里的 Screen 类是第 13 章介绍的。定义如下：

```
class ScreenPtr {
    // ...
private:
```

```
    Screen *ptr;
};
```

我们希望定义 ScreenPtr 类，来保证这种类的对象总是指向 Screen 对象。它不能不指向对象，如同内置指针一样。我们的应用程序可以直接使用 ScreenPtr 类型的对象，而不用先测试它是否指向一个 Screen 对象。为了获得这种行为，我们定义一个带有构造函数的 ScreenPtr 类，但是它没有缺省构造函数（关于构造函数在 14.2 节详细讨论）：

```
class ScreenPtr {
public:
    ScreenPtr( Screen &s ) : ptr( &s ) { }

    //....
};
```

ScreenPtr 类型的对象的定义必须提供初始值：一个 Screen 类型的对象，ScreenPtr 对象将指向它，否则 ScreenPtr 对象的定义就是错误的：

```
ScreenPtr p1;                  // 错误：ScreenPtr 没有缺省构造函数
Screen myScreen( 4, 4 );
ScreenPtr ps( myScreen );      // ok
```

为使 ScreenPtr 类的行为像内置指针，我们必须再定义一些重载操作符，我们定义的两个操作符是：解引用操作符（*）和成员操作符箭头（->）：

```
// 支持指针行为的重载操作符
class ScreenPtr {
public:
    Screen& operator*() { return *ptr; }
    Screen* operator->() { return ptr; }

    //....
};
```

成员访问操作符箭头被重载为一元操作符，即它没有参数。当它被用在表达式中时，只能根据左边操作数的类型来选择它。例如，下面给出的语句：

```
point->action();
```

将检查 point 以决定其类型。如果 point 是某一个类类型的指针，则这个语句使用内置成员访问操作符箭头的语义。如果 point 是某一个类类型的对象或引用，则查找这个类的重载的成员操作符箭头。如果没有定义成员操作符，则该语句就是错的，因为类对象或引用通常必须使用点成员访问操作符来引用类成员。如果定义了重载的成员访问操作符箭头，则它被绑定到 point 上，并被调用。

重载的成员访问操作符箭头的返回类型必须是一个类类型的指针，或者是“定义该成员访问操作符箭头的类”的一个对象。如果返回类型是一个类类型的指针，则内置成员访问操作符箭头的语义被应用在返回值上。如果返回值是另外一个类的对象或引用，则递归应用该过程，直到返回的是指针类型或语句错误。例如，我们可以用 ScreenPtr 对象 ps 访问 Screen 类的成员，如下所示：

```
ps->move( 2, 3 );
```

因为成员访问操作符箭头的左操作数的类型是 ScreenPtr，所以使用该类的重载操作符。

该操作符返回一个指向 Screen 类对象的指针，内置成员访问操作符箭头被依次应用在这个返回值上，以调用 Screen 类的成员函数 move()。

下面的小程序使用了我们的 ScreenPtr 类，ScreenPtr 类型的对象用起来就像 Screen*类型的对象一样：

```
#include <iostream>
#include <string>
#include "Screen.h"

void printScreen( ScreenPtr &ps )
{
     cout << "Screen Object ("
          << ps->height() << ", "
          << ps->width() << " )\n\n";
     for ( int ix = 1; ix <= ps ->height(); ++ix )
     {
          for ( int iy = 1; iy <= ps ->width(); ++iy )
               cout << ps->get( ix, iy );
          cout << "\n";
     }
}

int main() {
     Screen sobj( 2, 5 );
     string iint( "HelloWorld" );
     ScreenPtr ps( sobj );

     // 设置屏幕的内容
     string::size_type iintpos = 0;
     for ( int ix = 1; ix <= ps->height(); ++ix )
          for ( int iy = 1; iy <= ps ->width(); ++iy )
          {
               ps->move( ix, iy );
               ps->set( iint[ iintpos++ ] );
          }

     // 输出屏幕的内容
     printScreen( ps );
     return 0;
}
```

当然，这种操纵类对象的指针不像使用内置指针类型一样有效率。所以，智能指针类必须提供其他一些对于我们的程序设计很重要的功能，以便抵消使用它产生的额外开销。

15.7 操作符++和--

为了继续实现上一节介绍的 ScreenPtr 类，我们为这个类定义内置指针类型所支持的另外两个操作符：递增（++）和递减（--）。希望能够用 ScreenPtr 类来引用 Screen 对象的数组的元素。为了做到这一点，我们需要向 ScreenPtr 类增加一些数据成员。

我们先定义一个被称为 size 的新数据成员，它含有 0（表明 ScreenPtr 对象指向单个对象）或含为 ScreenPtr 对象所指数组的大小。再定义一个被称为 offset 的数据成员，用来记录 ScreenPtr 对象所指数组中的偏移量：

```
class ScreenPtr {
public:
     // ...
private:
     int size;     // 数组的大小，对于单个对象，其值为 0
     int offset;   // ptr 在数组中的偏移
     Screen *ptr;
};
```

由于有了这些额外的功能和这些新的数据成员，我们必须修改 ScreenPtr 类的构造函数。如果被创建的 ScreenPtr 对象指向一个数组，则 ScreenPtr 的用户必须向构造函数提供额外的实参：

```
class ScreenPtr {
public:
     ScreenPtr( Screen &s , int arraySize = 0 )
          : ptr( &s ), size ( arraySize ), offset( 0 ) { }
private:
     int size;
     int offset;
     Screen *ptr;
};
```

构造函数中额外的实参指出了数组的大小。为了保留原来的功能，构造函数的第二个参数得到一个缺省实参，并把 size 的值设置为 0。如果在创建 ScreenPtr 对象时，没有提供第二个实参，则使用缺省实参。在这种情况下，将假设该对象引用单个的 Screen 对象。新的 ScreenPtr 类的对象可以被定义如下：

```
Screen myScreen( 4, 4 );
ScreenPtr pobj( myScreen );            // ok: 指向单个对象
const int arrSize = 10;
Screen *parray = new Screen[ arrSize ];
ScreenPtr parr( *parray, arrSize );    // ok: 指向数组
```

现在，我们已经为 ScreenPtr 重载递增和递减操作符做好了准备。还有一个小问题是，存在两种递增和递减操作符：前置版本和后置版本。幸运的是，重载的递增和递减操作符的前置和后置实例都可以被定义。前置操作符的声明看起来就像你所期望的那样：

```
class ScreenPtr {
public:
     Screen& operator++();
     Screen& operator- - ();
     // ...
};
```

前置的递增和递减操作符被定义为一元操作符函数。例如，我们可以如下使用前置递增操作符：

```
const int arrSize = 10;
Screen *parray = new Screen[ arrSize ];
```

```
ScreenPtr parr( *parray, arrSize );
for ( int ix = 0;
      ix < arrSize;
      ++ix, ++parr // 等价于 parr.operator++()
)
      printScreen( parr );
```

这些重载的操作符可被定义如下：

```
Screen& ScreenPtr::operator++()
{
      if ( size == 0 ) {
            cerr << "cannot increment pointer to single object\n";
            return *ptr;
      }

      if ( offset >= size - 1 ) {
            cerr << "already at the end of the array\n";
            return *ptr;
      }
      ++offset;
      return *++ptr;
}

Screen& ScreenPtr::operator--()
{
      if ( size == 0 ) {
            cerr << "cannot decrement pointer to single object\n";
            return *ptr;
      }

      if ( offset <= 0 ) {
            cerr << "already at the beginning of the array\n";
            return *ptr;
      }
      --offset;
      return *--ptr;
}
```

为区分后置操作符与前置操作符的声明，重载的递增和递减后置操作符的声明有一个额外的 int 类型的参数。在下面的例子中，它声明了 ScreenPtr 类的前置和后置操作符对：

```
class ScreenPtr {
public:
      Screen& operator++();      // 前置操作符
      Screen& operator- - ();
      Screen& operator++(int);   // 后置操作符
      Screen& operator- - (int);
      // ...
};
```

后置操作符可被实现如下：

```
Screen& ScreenPtr::operator++(int)
{
```

```
        if ( size == 0 ) {
            cerr << "cannot increment pointer to single object\n";
            return *ptr;
        }
        if ( offset == size ) {
            cerr << "already one past the end of the array\n";
            return *ptr;
        }
        ++offset;
        return *ptr++;
    }

    Screen& ScreenPtr::operator--(int)
    {
        if ( size == 0 ) {
            cerr << "cannot decrement pointer to single object\n";
            return *ptr;
        }
        if ( offset == - 1 ) {
            cerr << "already one before the beginning of the array\n";
            return *ptr;
        }
        --offset;
        return *ptr--;
    }
```

注意，这里不需要给出参数名，因为它没有被用在操作符定义中。额外的整型参数对于后置操作符的用户是透明的，编译器为它提供了缺省值，因而该参数也可以被忽略。这就是参数没有被命名的原因。下面的例子使用了后置操作符：

```
    const int arrSize = 10;
    Screen *parray = new Screen[ arrSize ];
    ScreenPtr parr( *parray, arrSize );

    for ( int ix = 0; ix < arrSize; ++ix )
        printScreen( parr++ );
```

对于后置操作符的显式调用要求为第二个整型实参指定一个实际的值。对于我们的 ScreenPtr 类，为该显式调用而指定的实参被忽略，因为它没被用在重载操作符的定义中。

```
    parr.operator++(1024);          // 调用后置操作符++
```

重载的递增和递减操作符也可以被声明为友元函数。例如，我们可以改变 ScreenPtr 的定义，把这些操作符声明为友元函数，如下所示：

```
    class ScreenPtr {
        // 非成员声明
        friend Screen& operator++( ScreenPtr & );       // 前置
        friend Screen& operator- - ( ScreenPtr & );
        friend Screen& operator++( ScreenPtr &, int );  // 后置
        friend Screen& operator- - ( ScreenPtr &, int );
    public:
        // 成员定义
    };
```

练习 15.7

当重载的递增和递减操作符被声明为友元函数时，请为 ScreenPtr 类提供它们的定义。

练习 15.8

ScreenPtr 类现在可以表示一个指向 Screen 类数组的指针。请修改重载的 operator*()和重载的 operator一>()（在 15.6 节定义），确保当 ScreenPtr 对象指向一个数组元素，该对象不会指向数组前元素之前和末元素之后的元素。提示：这些重载的操作符应该使用新的数据成员 size 和 offset。

15.8 操作符 new 和 delete

在缺省情况下，空闲存贮区中的类对象的分配和释放，由在 C++标准库中定义的全局操作符 new()和 delete()来执行（我靡蜈 8.4 节介绍了这些操作符）。如果一个类提供了两个分别被称为操作符 new()和操作符 delete()的成员函数，那么它就可以承接自己的内存管理权。如果在类中定义了这些成员操作符，则它们会被调用，以取代全局操作符来分配和释放该类类型的对象。作为一个例子，让我们把操作符 new()和 delete()定义为 Screen 类的成员。

类成员操作符 new()的返回类型必须是 void*型，并且有一个 size_t 类型的参数，这里的 size_t 是一个在系统头文件<cstddef>中被定义的 typedef。下面是 Screen 类操作符 new()的声明：

```
class Screen {
public:
     void *operator new( size_t );
     // ...
};
```

当 new 表达式创建一个类类型的对象时，编译器查看该类是否有一个成员操作符 new()。如果有，则选择这个操作符为该类对象分配内存；否则，调用全局操作符 new()。例如，下面的 new 表达式：

```
Screen *ps = new Screen;
```

在空闲存贮区中创建了一个 Screen 类型的对象，因为 Screen 类有一个成员操作符 new()，所以调用该成员操作符 new()。操作符的 size_t 参数自动被初始化为 Screen 类的大小（按字节计数）

增加或删除一个类的操作符 new()并不要求改变用户的代码。new 表达式在调用全局操作符 new()或者类成员操作符 new()时，其形式相同。如果 Screen 类没有定义自己的操作符 new()：则 new()表达式仍然有效，只不过调用的是全局操作符 new()。

程序员可以使用“全局域解析操作符”来选择调用全局操作符 new()。例如：

```
Screen *ps = ::new Screen;
```

调用了全局操作符 new()，即使 Screen 类也定义了 new()操作符作为其成员。

类成员操作符 delete()的返回类型必须是 void，并且第一个参数的类型是 void*。下面是

Screen 类的操作符 delete()的声明：

```
class Screen {
public:
     void operator delete( void* );
};
```

当 delete 表达式的操作数是指向一个类类型对象的指针时，编译器检查该类是否向一个成员操作符 delete()。如果有，则选择该操作符为类对象释放内存；否则，调用全局操作符 delete()。下面的 delete 表达式：

```
delete ps;
```

释放了 ps 所指的 screen 类对象的内存。因为 Screen 类有成员操作符 delete()，所以调用了该类成员操作符 delete()。操作符的 void*参数自动被初始化为 ps 值。

增加或去掉一个类的操作符 delete()并不要求改变用户的代码。delete 表达式在调用全局操作符 delete()和类成员操作符 delete()时，其形式相同。如果 Screen 类没有定义自己的操作符 delete()，则上面的 delete 表达式仍然有效，只不过调用了全局操作符 delete()。

程序员可以通过使用“全局域解析操作符”有选择地调用全局操作符 delete()。例如：

```
::delete ps;
```

调用全局域中定义的操作符 delete()，即使 Screen 类把操作符 delete()定义为它的一个成员。一般来说，被使用的操作符 delete()应该与用来分配存贮区的 new()操作符相匹配。例如，如果 ps 所指的存贮区是通过调用全局操作符 new()的 new 表达式分配的，那么 delete 表达式也应该调用全局操作符 delete()。

为一个类类型而定义的 delete()操作符，如果它是被 delete 表达式调用的，则它可以有两个参数而不是一个。第一个参数仍然必须是 void*型，而第二个参数必须是预定义类型 size_t（记住，它被包含在库头文件<cstddef>中）。例如：

```
class Screen {
public:
     // replaces:
     // void operator delete( void* );
     void operator delete( void *, size_t );
};
```

如果还存在额外的参数，则它将被编译器用第一个参数所指对象的字节大小自动初始化。［这个参数在面向对象的类层次结构中是非常基本的，在一个面向对象的类层次结构中，操作符 delete()可以被一个派生类继承，第 17 章将详细讨论继承关系。］

让我们更详细地看一看 Screen 类成员操作符 new()和 delete()的实现。我们的内存分配策略是管理一个链表，它包含许多可供使用的 Screen 类对象，并由 freeStore 指针指示位置。Screen 成员操作符 new()的每次调用都返回 freeStore 所指的下一个类对象。成员操作符 delete()的每次调用都把类对象返回到 freeStore 所指的链表开始处。如果由 freeStore 指向的类对象链表为空，则调用全局操作符 new()分配一块能够包含 screenChunk 个 Screen 对象的存贮区。

screenChunk 和 freeStore 包含的值只对 Screen 类才有意义。因此，我们把它们封装为 Screen 类的私有成员。另外，对于所有被创建的 Screen 类对象，这些数据成员必须只能有一个实例。所以这些成员被声明为 static 成员。第三个数据成员 next，被用来维护 Screen 对象

的链表。如下所示：

```
class Screen {
public:
     void *operator new( size_t );
     void operator delete( void *, size_t );
     // ...

private:
     Screen *next;
     static Screen *freeStore;
     static const int screenChunk;
};
```

下面是 Screen 成员操作符 new()的一种实现：

```
#include "Screen.h"
#include <cstddef>

// 静态成员在程序文本文件中被初始化，而不是头文件中
Screen *Screen::freeStore = 0;
const int Screen::screenChunk = 24;

void *Screen::operator new( size_t size )
{
     Screen *p;
     if ( !freeStore ) {
          // 链表空：抓取一块存储区
          // 这里调用全局 new
          size_t chunk = screenChunk * size;
          freeStore = p =
               reinterpret_cast< Screen* >( new char[ chunk ] );

          // 现在把已经分配的内存串起来
          for ( ;
                    p != &freeStore[ screenChunk - 1 ];
                    ++p )
               p- >next = p+1;
          p- >next = 0;
     }
     p = freeStore;
     freeStore = freeStore - >next;
     return p;
}
```

下面是 Screen 成员操作符 delete()的一种可能的实现：

```
void Screen::operator delete( void *p, size_t )
{
     // 将被删除的对象插入到空闲链表尾
     ( static_cast< Screen* >( p ) )- >next = freeStore;
     freeStore = static_cast< Screen* >( p );
}
```

我们可以为一个类只声明操作符 new()而不声明相应的操作符 delete()。在这种情况下，

通过全局操作符 delete()删除类对象。我们也可以只为一个类声明操作符 delete()，而不声明操作符 new()。在这种情况下，该类对象被通过全局操作符 new()创建。但是，这些操作符通常成对出现，如本例的情形，类设计者经常需要同时提供它们两个。

操作符 new()和 delete()都是类的静态（static）成员，它们遵从静态成员函数的一般限制。这些操作符被自动做成静态成员函数，而无需程序员显式地把它们声明为静态的。尤其要记住的是静态成员函数没有 this 指针，因此它们只能访问静态数据成员（关于静态成员函数的讨论见 13.5 节）。其原因是，这些操作符被调用的时候，要么是在该类对象被创建之前［操作符 new()］，要么是在其被销毁之后［操作符 delete()］。

用操作符 new()的分配动作，如：

```
Screen *ptr = new Screen( 10, 20 );
```

与下列双语句序列等价：

```
// C++伪码
ptr = Screen::operator new( sizeof( Screen ) );
Screen::Screen( ptr, 10, 20 );
```

即，new 表达式先调用该类的操作符 new()来分配存贮区，然后再调用构造函数初始化该对象。如果操作符 new()失败，则抛出 bad_alloc 类型的异常，并且不会调用构造函数。

用操作符 delete()释放存贮区的动作，如：

```
delete ptr;
```

与下列双语句序列等价：

```
// C++伪玛
Screen::~Screen( ptr );
Screen::operator delete( ptr, sizeof( *ptr ) );
```

即，delete 表达式首先调用该对象的析构函数，然后再调用该类的操作符 delete()释放存贮区。如果 ptr 的值是 0，则不会调用析构函数和操作符 delete()。

15.8.1 数组操作符 new[]和 delete[]

在上一小节中定义的类操作符 new()只能被用来分配单个类对象。例如，下面的 new 表达式调用了 Screen 操作符 new()：

```
// 调用 screen::operator new()
Screen *ps=new Screen( 24, 88 );
```

而下面的 new 表达式通过调用全局操作符 new[]()，在空闲存贮区中分配 Screen 对象数组：

```
// 调用 ::operator new[]()
Screen *psa = ne w Screen[10];
```

我们也可以把针对数组分配的操作符 new[]()和 delete[]()声明为类的成员。类成员操作符 new[]()的返回类型必须是 void*，并且第一个参数的类型是 size_t。下面是 Screen 类操作符 new[]()的声明：

```
class Screen {
```

```
public:
    void *operator new[]( size_t );
    // ...
};
```

当一个 new 表达式创建一个类类型对象的数组时，编译器将检查该类是否有成员操作符 new[]()。如果有，则用该操作符来分配数组的内存，否则将调用全局操作符 new[]()。下面的 new 表达式在空闲存贮区中创建了一个包含 10 个 Screen 类对象的数组：

```
Screen *ps = new Screen[10];
```

用为 Screen 类有成员操作符 new[]()，所以，new 表达式调用该操作符。操作符的 size_t 参数被自动初始化，其值等于存放 10 个 Screen 对象的数组所需内存的字节大小。

即使 Screen 类有一个成员操作符 new[]()，程序员也可以通过全局域解析操作符，来调用全局 new[]()来创建 Screen 对象的数组。例如，下面的 new 表达式调用了全局域中定义的操作符 new[]()：

```
Screen *ps = ::new Screen[10];
```

成员操作符 delete[]()的返回类型必须是 void，它的第一个参数必须是 void*类型。例如，下面是 Screen 类操作符 delete[]()的声明：

```
class Screen {
public:
    void operator delete[]( void* );
};
```

为删除一个类的数组，delete 表达式必须使用数组语法：

```
delete[] ps;
```

当这样的 delete 表达式的操作数是一个指向类类型的指针时，编译器就会检查该类是否有成员操作符 delete[]()。如果有，则用该操作符来释放数组的内存；否则，调用全局操作符 delete[]()。操作符的 void*参数被自动初始化，其值等于数组存储区的起始处。

即使 Screen 类有一个成员操作符 delete[]()，程序员也可以通过全局域解析操作符有选择地调用全局操作符 delete[]()。例如：

```
::delete[] ps;
```

调用了全局域中定义的操作符 delete[]()。

增加或去掉类中的操作符 new[]()或操作符 delete[]()都不要求改变用户的代码。new 表达式和 delete 表达式在调用全局操作符或类成员操作符时，其形式相同。

创建数组的 new 表达式首先调用类操作符 new[]()来分配存贮区，然后再调用缺省构造函数依次初始化数组的每一个元素。如果这类定义了构造函数，但是没有缺省构造函数，则相应的 new 表达式就是错误的。因为没有任何 C++语法可以为数组元素指定初始值，或者在数组版本的 new 表达式中为类的构造函数指定实参。

删除数组的 delete 表达式先调用类的析构函数依次销毁数组的每一个元素，然后再调用类操作符 delete[]()来释放内存。针对数组的 delete 表达式必须要使用数组语法，这很重要。在下列语句中：

```
delete ps;
```

如果 ps 指向一个类对象的数组，则缺少[]可能导致只在数组的首元素上调用析构函数，尽管被释放的内存数量可能是正确的。

一个类的操作符 delete[]()也可以有两个参数，第二个参数的类型是预定义类型 size_t。例如：

```
class Screen {
public:
    // 代替:
    // void operator delete( void* );
    void operator delete[]( void*, size_t );
};
```

如果存在额外的参数，则它由编译器自动初始化，其值等于存贮数组所需内存的字节大小。

15.8.2 定位操作符 new()和 delete()

只要每个声明都有惟一的参数表，我们也可以重载类的成员操作符 new()。但是任何一个类操作符 new()的第一个参数的类型都必须是 size_t。例如：

```
class Screen {
public:
    void *operator new( size_t );
    void *operator new( size_t, Screen* );

    // ...
};
```

额外的参数可以被 new 表达式中指定的定位实参初始化。例如：

```
void func( Screen *start ) {
    Screen *ps = new (start) Screen;
    // ...
}
```

在 new 表达式中的关键字 new 后面、出现在括号中的部分表示定位实参。上面的 new 表达式调用双参数的成员操作符 new()。第一个参数被自动初始化为 Screen 类的字节大小值。第二个参数被初始化为定位实参 start 的值。

我们也可以重载类成员操作符 delete()。但是，这样的操作符不会以 delete 表达式的方式被调用。如果 new 表达式调用的构造函数（是的，这不是打字错误，真的是指 new 表达式）抛出一个异常的话，重载的操作符 delete()只能被编译器隐式地调用。让我们仔细地看一看何时使用这样的 delete()操作符。

下面的 new 表达式：

```
Screen *ps = new ( start ) Screen;
```

它的动作如下：

1．调用类 Screen 的操作符 new（size_t, Screen*）；

2．接着，调用类 Screen 的缺省构造函数初始化该对象；

3．然后，用 Screen 对象的地址初始化 ps。

让我们假设类操作符 new(size_t, Screen*)通过调用全局操作符 new()分配内存。如果在第二步调用的 Screen 构造函数抛出一个异常，那么类的设计者怎样确保由这个操作符 new()分

配的内存被正确地释放？类的设计者怎样保护用户的代码不会产生内存泄漏？类的设计者可以提供一个重载的操作符 delete()，用于在这种情况下（且只能在这种情况下）被调用。

如果类的设计者提供了一个重载的操作符 delete()，且它的参数类型与操作符 new()的参数类型匹配，则编译器就会自动调用这个操作符 delete()来释放存储区。例如，已知下列定位 new 表达式：

```
Screen *ps = new (start) Screen;
```

如果 Screen 类的缺省构造函数抛出一个异常并退出，则编译器会在 Screen 类的域中查找一个操作符 delete()。要想使操作符 delete()被考虑，它必须具有与被调用的 new()操作符的参数类型相匹配的参数。因为操作符 new()的第一个参数的类型总是 size_t，而操作符 delete()的第一个参数是 void*类型，所以每个函数的第一个参数不会被考虑用来做这种比较。编译器在 Screen 类中查找下面形式的操作符 delete()：

```
void operator delete( void*, Screen* );
```

如果 new 表达式调用的构造函数抛出一个异常，并且在 Screen 类中找到了这样的 delete()操作符，则编译器就调用它来释放内存。如果没有找到这样的操作符 delete()，则不会调用任何 delete()操作符。

根据操作符 new()是否分配内存或者是否重新使用已分配的内存，类设计者可以决定是否提供与特定操作符 new()相匹配的操作符 delete()。如果说操作符 new()分配了内存，则应该提供定位操作符 delete()，以便当“new 表达式调用的构造函数抛出异常”时可以正确地释放内存。如果定位操作符 new()没有分配内存，则无需提供相匹配的操作符 delete()来释放内存。

我们也可以重载针对数组的定位操作符 new[]()和 delete[]()：

```
class Screen {
public:
    void *operator new[]( size_t );
    void *operator new[]( size_t, Screen* );
    void operator delete[]( void*, size_t );
    void operator delete[]( void*, Screen* );

    // ...
};
```

当分配数组的 new 表达式指定匹配的定位实参时，定位操作符 new[]()就会被使用。例如：

```
void func( Screen *start ) {
    // 调用 Screen::operator new[]( size_t, Screen* )
    Screen *ps = new (start) Screen[10];

    // ...
}
```

如果由该 new 表达式调用的构造函数抛出一个异常，则编译器就会自动调用在 Screen 类中定义的、匹配的重载操作符 deletee[]()。

练习 15.9

请说明下列初始化哪些是错误的，并说明原因。

```
class iStack {
public:
```

```
        iStack( int capacity )
            : _stack( capacity ), _top( 0 ) {}
        // ...
    private:
        int _top;
        vector< int > _stack;
    };

    (a)  iStack *ps = new iStack(20);
    (b)  iStack *ps2 = new const iStack(15);
    (c)  iStack *ps 3 = new iStack[ 100 ];
```

练习 15.10

请说明在下列 new 和 delete 表达式中发生了什么事情：

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

Exercise *pe = new Exercise[20];
delete[] pe;
```

改变 new 和 delete 表达式，以便调用全局操作符 new()和 delete()。

练习 15.11

请说明类设计者为什么应该提供类的定位操作符 delete()。

15.9 用户定义的转换

我们已经看到了类型转换怎样被应用到内置类型的操作数上。在 4.14 节我们了解了类型转换怎样被应用到内置操作符的操作数上。在 9.3 节我们查看了类型转换怎样被应用在函数调用的实参上，使其转换成函数参数的类型。例如，类型转换被应用在下列六个加法操作的操作数上：

```
char ch; short sh; int ival;

/* 每个加法中的一个操作数
 * 要求一个类型转换 */
ch + ival; ival + ch;
ch + sh; sh + ch;
ival + sh; sh + ival;
```

操作数 ch 和 sh 被提升为 int 型，所做的加法是两个 int 型值的加法，提升动作由编译器隐式处理，因此对用户是透明的。

在本节中，我们将考虑类的设计者怎样为类类型的对象提供一组用户定义的转换。这些用户定义的转换也是由编译器在需要时隐式地调用的。为了说明为什么需要用户定义的转换，

我们再次使用 10.9 节介绍的 SmallInt 类。

回忆一下，这个类允许我们定义一些对象。它们含有与 8 位 unsigned char 相同范围的值——即 0 到 255。另外，这个类能捕获到上溢和下溢错误。除此之外，它的行为与一个 unsigned char 相同。

我们希望能够在 SmallInt 对象与其他 SmallInt 对象或者内置算术类型的对象之间进行加减操作。我们要通过提供 6 个 SmallInt 操作符函数来实现对这些操作的支持：

```
class SmallInt {
     friend operator+( const SmallInt &, int );
     friend operator- ( const SmallInt &, int );
     friend operator- ( int, const SmallInt & );
     friend operator+( int, const SmallInt & );
public:
     SmallInt( int ival ) : value( ival ) { }
     operator+( const SmallInt & );
     operator- ( const SmallInt & );
     // ...
private:
     int value;
};
```

两个成员操作符允许我们加减两个 SmallInt 对象。友元全局操作符允许我们在 SmallInt 对象和内置算术类型的对象之间进行加减操作。之所以只需要 6 个操作符，是因为任何内置算术类型都可以被转换为与 int 型参数相匹配。例如，表达式：

```
SmallInt si( 3 );
si + 3.14159
```

分两步被解析时：

1．double 文字常量 3.14159 被转换成整型值 3；

2．调用操作符 operator+(const SmallInt&, int)，返回值 6。

如果我们还想支持按位操作符、逻辑操作符、关系操作符和复合赋值操作符，则要求的操作符的数目就变得非常可怕了。我们更希望的，不是提供所有的重载操作符，而是一种将 SmallInt 类对象自动转换成 int 型对象的方式。

C++提供了一种机制，通过它，每个类都可以定义一组“可被应用在该类型对象上的转换”。对于 SmallInt，我们定义了一个从 SmallInt 对象到 int 型的转换。下面是实现：

```
class SmallInt {
public:
     SmallInt( int ival ) : value( ival ) { }

     // 转换操作符
     // SmallInt ==> int
     operator int() { return value; }

     // 没有提供重载操作符
private:
     int value;
};
```

操作符 int()是一个**转换函数**（conversion function），它定义了一个**用户定义的转换**（use-defined conversion）。用户定义的转换是在类类型和转换函数中指定的类型之间的转换。在本例中，是类型 int。转换函数定义了转换的意义以及应用转换时编译器必须执行的动作。从 SmallInt 类对象到 int 型转换的意义是，返回存贮在数据成员 value 中的 int 型的值。

现在，SmallInt 类对象可以被用在任何可以使用 int 对象的地力。假设不再提供重载的操作符、并且类 SmallInt 提供了一个向 int 型转换的函数，下列加法：

```
SmallInt si( 3 );
si + 3.14159;
```

被解析为如下两步：

1．调用 SmallInt 转换函数，产生整型值 3；

2．整型值 3 被提升为 3.0，并与 double 文字常量 3.14159 相加，生成 double 型 6.14159。

与我们前面定义的重载操作符的行为相比，这种行为更接近于内置类型操作数的行为。当一个 int 型的值被加到 double 型的值上时，执行的操作是 double 型操作数的加法（int 型的值被转换成 double 型的值），结果是产生一个 double 型的值。下面的程序说明了 SmallInt 类的用法：

```
#include <iostream>
#include "SmallInt.h"

SmallInt si1, si2;
int main() {
     cout << "enter a SmallInt, please: ";
     while ( cin >> si1 ) {
          cout << "The value read is "
               << si1 << "\nIt is ";

          // SmallInt::operator int() 被调用两次
          cout << ( ( si1 > 127 )
                    ? "greater than "
                    : ( ( si1 < 127 )
                         ? "less than "
                         : "equal to ") ) << "127\n";
          cout << "\nenter a SmallInt, please \
               (ctrl-d to exit): ";
     }
     cout << "bye now\n";
}
```

编译并运行程序，产生上列结果：

```
enter a SmallInt, please: 127

The value read is 127
It is equal to 127

enter a SmallInt, please (ctrl -d to exit): 126

The value read is 126
It is less than 127
```

```
enter a SmallInt, please (ctrl -d to exit): 128
The value read is 128
It is greater than 127
enter a SmallInt, please (ctrl -d to exit): 256
***SmallInt range error: 256 ***
```

实现 SmallInt 类的代码已经被修改了，以便增加额外的支持。如下所示：

```
#include <iostream>

class SmallInt {
    friend istream&
        operator>>( istream &is, SmallInt &s );
    friend ostream&
        operator<< ( ostream &os, const SmallInt &s )
        { return os << s.value; }
public:
    SmallInt( int i=0 ) : value( rangeCheck( i ) ){}
    int operator=( int i )
        { return( value = rangeCheck( i ) ); }
    operator int() { return value; }

private:
    int rangeCheck( int );
    int value;
};
```

在类体之外被定义的成员函数的定义如下：

```
istream& operator>>( istream &is, SmallInt &si ) {
    int ix;
    is >> ix;
    si = ix; // SmallInt::operator=(int)
    return is;
}

int SmallInt::rangeCheck( int i )
{
    /* 如果前 8 位以外的位被置位
     * 则报告值太大了：然后退出 */
    if ( i & ~0377 ) {
        cerr << "\en***SmallInt range error: "
             << i << " ***" << endl;
        exit( - 1 );
    }

    return i;
}
```

15.9.1 转换函数

转换函数（conversion function）是一种特殊类型的类成员函数。它定义了一个由用户定义的转换，以便把一个类对象转换成某种其他的类型。在类体中通过指定关键字 operator，

并在其后加上转换的目标类型后，我们就可以声明转换函数。

在转换函数的声明中，关键字 operator 后面的名字不一定必须是内置类型的名字。接下来定义的 Token 类定义了多个转换函数，其中一个被定义为使用 typedef 名 tName，而另一个定义了向类类型 SmallInt 的转换：

```
#include "SmallInt.h"

typedef char *tName;
class Token {
public:
     Token( char*, int );
     operator SmallInt() { return val; }
     operator tName() { return name; }
     operator int() { return val; }
     // 其他公有成员
private:
     SmallInt val;
     char *name;
};
```

请注意向 SmallInt 进行转换的和 int 转换函数的定义是相同的。成员 val 的值被转换函数 Token::operator int()返回。因为 val 的类型是 SmallInt，所以编译器隐式地应用转换函数 SmallInt::operator int()，把 val 转换成 int 型。而编译器又隐式地应用 Token::operator int()，把 Token 型的对象转换成 int 型的值。例如，编译器用这个转换函数隐式地把 Token 型的实参 t1 和 t2 转换成 int 型，print()的参数类型：

```
#include "Token.h"

void print( int i )
{
     cout << "print( int ) : " << i << endl;
}

Token t1( "integer constant", 127 );
Token t2( "friend", 255 );

int main()
{
     print( t1 ); // t1.operator int()
     print( t2 ); // t2.operator int()
     return 0;
}
```

编译并运行这个小程序，生成下列输出：

```
print( int ) : 127
print( int ) : 255
```

转换函数采用如下的一般形式：

```
operator type();
```

这里的 type 可用内置类型、类类型或 typedef 名取代。但是不允许 type 表示数组或函数类型。转换函数必须是成员函数，它的声明不能指定返回类型和参数表。例如，下列声明都

是错误的：

```
operator int( SmallInt & );    // 错误：不是成员
class SmallInt {
public:
     int operator int();       // 错误：返回类型
     operator int( int = 0 );  // 错误：参数表
     // ...
};
```

显式的强制类型转换会导致调用转换函数。如果被转换值的类型是一个类类型，它有个转换函数，并且该转换函数的类型是强制转换所指定的类型，则调用这个类的转换函数。例如：

```
#include "Token.h"

Token tok( "function", 78 );

// 函数型的表示法：调用 Token::operator SmallInt()
SmallInt tokVal = SmallInt( tok );

// static_cast：调用 Token::operator tName()
char *tokName = static_cast< char * >( tok );
```

转换函数 Token::operator char*()可能会有意料不到的副作用。你知道是什么吗？试图直接访问私有成员 Token::name，将被编译器标记为错误：

```
char *tokName = tok.name;    // 错误：Token::name 是 private 的
```

但是现在，我们的转换函数提供的却正是这种访问，但我们并不希望，用户能够直接修改 Token::name。下面的例子说明了这样的修改是如何进行的：

```
#include "Token.h"

Token tok( "function", 78 );
char *tokName = tok;          // ok：隐式转换
*tokName = 'P';               // 喔！Token 的 name 成员现在是 "Punction"！
```

我们的意图是，只允许对被转换的 Token 类对象进行只读访问。为了实现这个目的，转换操作符必须返回一个 const char*：

```
typedef const char *cchar;
class Token {
public:
     operator cchar() { return name; }

     // ...
};

// 错误：不允许把 char* 转换成 const char*
char *pn = tok;
const char *pn2 = tok; // ok
```

另外一种解决方案是，改变 Token 的定义，让它使用在 C++标准库中定义的 string 类型。例如：

```
class Token {
```

```
public:
    Token( string, int );
    operator SmallInt() { return val; }
    operator string() { return name; }
    operator int() { return val; }
    // 其他公有成员

private:
    SmallInt val;
    string name;
};
```

Token::operator string()的语义是以传值方式返回代表 Token 名字的 string，这防止了程序无意中修改 Token 的私有成员 name 的值。

使用转换函数时，转换的目标类型必须与转换函数的类型完全匹配吗？例如，下列代码会调用 Token 类中定义的转换函数 operator int()吗？

```
extern void calc( double );
Token tok( "constant", 44 );

// 调用 tok.operator int() 吗? 是的
// int --> double 通过标准转换
calc( tok );
```

如果转换的目标（本例中的 double）与转换函数的类型（本例中的 int 类型）不完全匹配，且目标类型可以通过标准转换序列到达，则仍可调用转换函数（9.3 节讲述了标准转换序列）。为了调用函数 calc()，应该调用 Token::operator int()，以便把 tok 从 Token 类型转换成 int 型。然后再应用一个标准转换把用户定义的转换结果从 int 型转换成 double 型。

在用户定义的转换之后只允许标准转换序列。如果为了到达目标类型，必须应用第二个用户定义的转换，则编译器不会隐式应用任何转换。例如，如果 Token 没有定义 operator int()，则下列调用是非法的：

```
extern void calc( int );
Token tok( "pointer", 37 );

// 没有定义 Token::operator Int()
// 这个调用会产生编译时刻错误
calc( tok );
```

如果没有定义 Token::operator int()，tok 向 int 型的转换就会要求调用两个用户定义的转换函数。实参 tok 将首先需要从 Token 转换到 SmallInt，使用转换函数：

```
Token::operator SmallInt()
```

然后还需要用转换函数：

```
SmallInt::operator int()
```

把用户定义的转换的结果转换成 int 型。

如果没有定义 Token::openator int()，因为从类型 Token 到 int 之间不存在隐式的转换，调用 calc(tok) 将被标记为编译时刻错误。

如果在转换函数的类型和类类型之间没有逻辑匹配，则提供转换函数可能反而会使阅读程序的人糊涂。例如：

```
class Date {
public:
    // 猜猜会返回哪一个成员!
    operator int();

private:
    int month, day, year;
};
```

Date 的转换函数 operator int()应该返回什么值？无论因为何种原因，做出何种选择，对于程序的用户来说 Date 对象的用法都是不清楚的，这是因为在 Date 类型的对象与 int 型的值之间没有逻辑上的一对一映射关系。这种情况下，最好不要定义转换函数。

15.9.2 用构造函数作为转换函数

在一个类的构造函数中，凡是只带一个参数的构造函数，例如 SmallInt 的构造函数 SmallInt(int)，都定义了一组隐式转换，把构造函数的参数类型转换为该类的类型（如 SmallInt）。例如，SmallInt(int)把 int 型的值转换成 SmallInt 值：

```
extern void calc( SmallInt );
int i;

// 需要把 i 转换成一个 SmallInt 值
// SmallInt(int) 可以做到这一点
calc( i );
```

在调用 calc(i)中，通过调用构造函数 SmallInt(int)把 i 转换成 SmallInt 类型的值。编译器调用构造函数创建一个 SmallInt 类型的临时对象。然后，再将这个对象的值的拷贝传递给 calc()。可以将上面的函数调用理解成这样：

```
// C++伪码
// 创建一个临时 SmallInt 对象
{
    SmallInt temp = SmallInt( i );
    calc( temp );
}
```

这个例子中的花括号指出了 SmallInt 临时对象的生命期，即，临时对象会在函数调用语句结束时被销毁。

构造函数参数的类型也可能是另外一种类类型。例如：

```
class Number {
public:
    // 从一个 SmallInt 值创建一个 Number 值
    Number( const SmallInt& );

    // ...
};
```

在这种情况下，SmallInt 型的值可以被用在任何需要 Number 类型的值的地方。例如：

```
extern void func( Number );
SmallInt si( 87 );
int si(87);
```

```
{ // 调用 Number( const SmallInt & )
    func( si );
    // ...
}
```

使用构造函数执行隐式转换时，构造函数的参数类型必须与要被转换的值的类型完全匹配吗？例如，下列代码会调用 SmallInt 类中定义的构造函数 SmallInt(int)，来把 dobj 转换成类型 SmallInt 吗？

```
extern void calc( SmallInt );
double dobj;

// 会调用 SmallInt( int ) 吗? 是的.
// 通过标准转换, dobj 被从 double 转换到 int
calc( dobj );
```

如果需要，编译器会在调用构造函数执行用户定义的转换之前，在实参上应用标准转换序列。为了调用函数 calc()，先应用一个标准转换，以便把 dobj 从 double 型转换成 int 型。然后再调用 SmallInt(int)把转换的结果转换成 SmallInt。

编译器隐式地用单参数构造函数，将参数类型的值转换成构造函数类类型的值。这可能不是我们所希望的。我们可能决定，构造函数 Number(const SmallInt&）必须只能被用来“以 SmallInt 型的值初始化 Number 型的对象”，而在其他情况下编译器不会使用这个构造函数进行隐式类型转换。为防止使用该构造函数进行隐式类型转换，我们可以把它声明为显式的（explicit）：

```
class Number {
public:
    // 不会被用来执行隐式转换
    explicit Number( const SmallInt & );

    // ...
};
```

编译器不会使用一个显式构造函数来执行隐式类型转换。例如：

```
extern void func( Number );
SmallInt si(87);
int main()
{ // 错误: 从 SmallInt 到 Number 没有隐式转换
    func( si );

    // ...
}
```

但是，该构造函数仍然可以被用来执行类型转换，只要程序以强制转换的形式显式地要求转换即可。例如：

```
SmallInt si(87);
int main()
{ // 错误: 从 SmallInt 到 Number 没有隐式转换
    func( si );
    func( Number( si ) );   // ok: cast
    func( static_cast< Number >( si ) ); // ok: 强制转换
}
```

15.10 选择一个转换 ※

用户定义的转换是由转换函数或构造函数执行的转换。正如前面已经看到的，在转换函数执行转换之后，接着可以有一个标准转换把转换函数的结果转换成目标类型。类似地，在构造函数执行转换之前，也可以有一个标准转换把要被转换的值变成构造函数参数的类型。

用户定义的转换序列（user-defined conversion sequence）是用户定义的转换与“需要用来把值变成转换目标类型的标准转换”的组合。用户定义的转换序列形式如下：

标准转换序列——>

用户定义的转换——>

标准转换序列

这里，用户定义的转换或者调用转换函数，或者调用构造函数。

当试图转换一个值时，有可能存在两个不同的用户定义的转换序列，它们都能够被用来把该值转换成目标类型。当不只一个转换序列可以被应用时，编译器必须选择最好的序列执行转换。在本节中，我们将了解编译器是怎样做的。

一个类可以定义许多转换函数用来执行转换操作。例如，我们的 Number 类可能定义了两个转换函数：operator int()和 operator float()。这两个函数都可以用来把 Number 类型的对象转换成 float 型的值。当然，编译器可以直接用 Token::operator float()获得 float 型的值。它也可以用 Token::operator int()来完成该操作，因为通过标准转换，可以把转换函数的结果 int 型的值转换成 float 型的值。是否因为存在两个用户定义的转换序列而使该转换成为二义的呢？或者其中一个用户定义的转换序列更好？

```
920class Number {
public:
     operator float();
     operator int();
     // ...
};
Number num;
float ff = num;    // 哪一个转换函数? operator float()
```

如果两个转换函数都可以使用，则跟在转换函数之后的标准转换序列会成为“选择最佳的用户定义转换序列”的依据。在上一个例子中，应用了下列两个用户定义的转换序列来执行转换：

1．operator float()—>精确匹配；

2．operator int()—>标准转换。

正如 9.3 节所示，精确匹配优于标准转换。因此，第一个转换序列是较好的转换序列，于是选择转换函数 Token::operator float()来做转换。

类似地，很可能有两个构造函数都能被用来把一个值变成转换的目标类型。在这种情况下，在用户定义转换之前的标准转换序列会成为“选择最佳的用户定义转换序列”的依据。例如：

```
class SmallInt {
public:
```

```
        SmallInt( int ival ) : value( ival ) { }
        SmallInt( double dval )
            : value( static_cast< int >( dval ) )
                { }
};

extern void manip( const SmallInt & );
int main() {
    double dobj;
    manip( dobj ); //ok: SmallInt( double )
}
```

这里 SmallInt 类定义了两个构造函数——SmallInt(int)和 SmallInt(double)，可用来把 double 型的对象 dobj 转换成 SmallInt 型的值。可以用构造函数 SmallInt(double)来执行转换，这是因为它直接取 double 型的参数。也可以用 SmallInt(int)来执行转换，因为可以先用标准转换把 dobj 从 double 型转换为 int 型，然后再把这个结果用作构造函数 SmallInt(int)的实参。下面的两个用户定义转换序列都可以被用来做转换：

1．精确匹配—>SmallInt(double);

2．标准转换—>SmallInt(int)。

因为精确匹配要好于标准转换，所以使用构造函数 SmallInt(double)来执行转换。

编译器并不是总能选择出一个用户定义的转换序列作为最佳序列来执行转换。也许所有可能的转换都一样好。在这种情况下，我们称转换是二义的。在这种情况下，编译器不会隐式地应用任何转换。例如，使用 Number 类定义的两个转换函数：

```
class Number {
public:
    operator float();
    operator int();

    // ...
};
```

就不可能把 Number 型的对象隐式地转换成 long 型的对象。下面的语句将被标记为错误的，因为用户定义的转换序列是二义的：

```
// 错误：两个操作符 float() 和 int() 都可以应用
long lval = num;
```

可以用来把 num 转换成 long 型的用户定义的转换序列如下：

1．operator float()—>标准转换；

2．operator int()—>标准转换。

因为在用户定义的转换后面跟的都是标准转换，所以这两个转换序列的一样好，编译器不能为隐式转换选择一个惟一的转换序列。

程序员可以通过指定一个显式强制转换来指明要使用哪一个转换序列。例如：

```
// ok: 显式强制转换
long lval = static_cast< int >( num );
```

这个转换是显式的，使用转换函数 Number::operator int()，然后是从 int 到 long 的标准转换。

在为隐式转换选择一个用户定义的转换序列时，如果两个类定义了互相转换的函数，则二义性也可能会出现。例如：

```
class SmallInt {
public:
     SmallInt( const Number & );
     // ...
};

class Number {
public:
     operator SmallInt();
     // ...
};

extern void compute( SmallInt );
extern Number num;
compute( num );                    // 错误：两个可能的转换
```

实参 num 可以用两种不同的方式被转换成 SmallInt 型，可以使用构造函数 SmallInt::SmallInt(const Number&)或转换函数 Number::operator SmallInt()。因为这两个函数一样好，所以这个调用是错的。

程序员可以通过显式地调用 Number 类的转换函数，解决这种二义性问题。

```
// ok：显式调用以便解决二义性
compute( num.operator SmallInt() );
```

但是，程序员不能用显式强制类型转换来解决二义性，因为针对这个显式强制转换，转换函数和构造函数都会被考虑：

```
compute( SmallInt( num ) );     // 错误：仍然是二义的
```

正如你所看到的，提供多个转换函数和构造函数来执行隐式转换，也可能会导致意想不到的结果。我们应该明智地使用转换函数和构造函数，或者通过把构造函数声明为 explicit，以限制构造函数在隐式转换中的作用（因此也限制了其令人惊讶的影响）。

15.10.1 函数重载解析——回顾

在第 9 章中，我们详细描述了怎样解析重载函数的调用。当函数调用的实参是类类型。指向类类型的指针或者是指向成员的指针时，则需要为该调用考虑一个很大的函数集，以作为可能的候选函数。因此，使用类类型的实参影响了函数重载解析过程的第一步——选择候选函数集。

在函数重载解析过程的第三步中，最佳匹配函数被选择出来。为了执行这个选择，凡是能够把实参转换成相应参数类型的类型转换，都被划分等级。对于类类型的实参和参数，可能的转换集必须包括我们在上节介绍的用户定义的转换序列。因此，重载函数解析过程的第三步必须对用户定义的转换序列划分等级。

在本节中，我们将详细地看一看类类型的实参和参数怎样影响候选函数集，以及用户定义的转换怎样影响最佳匹配函数的选择。

15.10.2 候选函数

候选函数是与函数调用同名的函数。例如，我们有如下的函数调用：

```
SmallInt si(15);
add( si, 566 );
```

该调用的候选函数必须被命名为 add，而函数 add()的哪些声明会被考虑呢？

与任何函数调用一样，在调用点可见的函数 add()的声明都是候选函数。例如，在全局域中声明的两个函数 add()是下列调用的候选函数：

```
const matrix& add( const matrix &, int );
double add( double, double );

int main() {
    SmallInt si(15);
    add( si, 566 );
    // ...
}
```

“考虑在调用点可见的函数”这条原则并不是专门针对带有类类型实参的函数调用的。但是，针对这种函数调用，还会在另外两个域中查找函数声明：

1．如果实参是一个类类型的对象、类类型的指针、类类型的引用或者指向类成员的指针，并且该类类型是在一个用户声明的名字空间中被声明的，则在该名字空间中声明的、与函数调用同名的函数也被加入到候选函数集中。例如：

```
namespace NS {
    class SmallInt { /* ... */ };
    class String { /* ... */ };

    String add( const String &, const String & );
}

int main() {
    // si 的类型是 SmallInt 类
    // 该类在名字空间 NS 中被声明
    NS::SmallInt si(15);
    add( si, 566 ); // NS::add() 是一个候选函数
    return 0;
}
```

实参由是 SmallInt 类型的，该类型在名字空间 NS 中被声明。因此在名字空间 NS 中声明的函数 add(const string&,const String&)也被加入到候选函数集中。

2．如果实参是一个类类型的对象、类类型的指针、类类型的引用或者指向类成员的指针，并且该类有与函数调用同名的友元（friend）函数，则该友元函数被加入到候选函数集中。例如：

```
namespace NS {
    class SmallInt {
        friend SmallInt add( SmallInt, int ) { /* ... */ }
    };
}
```

```
int main() {
    NS::SmallInt si(15);

    add( si, 566 ); // 友元 add() 是一个候选函数
    return 0;
}
```

函数实参 si 的类型是 SmallInt，它的友元函数 add(SmallInt,int)是名字空间 NS 的一个成员，即便它没有在名字空间 NS 中被直接声明。正常情况下，在名字空间 NS 中的查找过程不会找到友元函数。但是，带有 SmallInt 型实参的 add()函数的调用会考虑在 SmallInt 类的成员表中声明的友元函数，并把它们加入到候选函数集中。

因此，如果一个函数调用的实参是一个类类型的对象、类类型的指针、类类型的引用或者是指向类成员的指针，则候选函数是以下各个函数集合的并集：在调用点可见的函数、在定义该类的名字空间中声明的函数，以及在类成员表中声明为友元的函数。

如果我们把前面的例子的各部分都放到一起：

```
namespace NS {
    class SmallInt {
        friend SmallInt add( SmallInt, int ) { /* ... */ }
    };

    class String { /* ... */ };
    String add( const String &, const String & );
}

const matrix& add( const matrix &, int );
double add( double, double );

int main() {
    // si 的类型是 SmallInt 类
    // 该类在名字空间 NS 中被声明
    NS::SmallInt si(15);
    add( si, 566 );   // 调用友元 function
    return 0;
}
```

则候选函数是：

1．全局函数：

```
add( const matrix &, int )
add( double, double )
```

2．名字空间函数：

```
NS::add( const String &, const String & )
```

3．友元函数：

```
NS::add( SmallInt, int )
```

函数重载解析过程选择 SmallInt 类的友元函数 NS::add(SmallInt,int)作为该调用的最佳匹配函数，是因为在调用中指定的两个实参都与友元函数的参数精确匹配。

当然，函数调用可能有多个实参是类类型、类类型的指针、类类型的引用或者类成员的

指针，且与每个实参对应的类类型可能也不相同。因此每个实参都被依次检查，并从定义相应的类的名字空间中和它的友元函数中查找候选函数。因此，带有类类型实参的调用的候选函数集，可能包含不同名字空间的函数，以及不同类中声明的友元函数。

15.10.3 类域中的函数所调用的候选函数

当形式为：

```
calc(t)
```

的函数调用出现在类域中时（例如在成员函数中），在上一小节中描述的候选函数的第一集合（即含有在调用点可见的函数声明的集合）可能包含非成员函数。通过名字解析可以找到在调用点可见的候选函数集合。关于类域中的名字解析，曾经在 13.9 节（针对类域）、13.10 节（针对嵌套类）、13.11 节（针对作为名字空间成员的类）以及 13.12 节（针对局部类）中都有相关的讨论。

比我们来看一个例子：

```
namespace NS {
     struct myClass {
          void k( int );
          static void k( char* );
          void mf();
     };

     int k( double );
};

void h(char);
void NS::myClass::mf() {
     h('a');  // 调用全局 h( char )
     k(4);    // 调用 myClass::k( int )
}
```

正如在 13.11 节中提到的，编译器以相反的顺序搜寻限定修饰符 NS::myClass::，即先在 myClass 中，然后再在名字空间 NS 中，针对成员函数 mf()的定义中用到的名字，查找可见的声明。我们先考虑调用：

```
h( 'a' );
```

在成员函数 mf()定义中的名字解析过程首先针对 h()的调用，考虑 myClass 的成员函数。因为在 myClass 的域中没有找到名字为 h()的成员函数，所以接下来在名字空间 NS 中查找候选函数。因为在名字空间 NS 域中也没有找到名为 h()的函数，故而接下来在全局域中查找候选函数。在这里找到了全局函数 h(char)，并且它是在调用点可见的候选函数集中惟一的函数。

在这个查找过程中，一旦找到了一个函数声明，则查找“在调用点可见的候选函数”的过程马上结束。该集合只包含名字解析成功的域中声明的函数。这也可以在调用：

```
k( 4 );
```

的候选函数集中看到。

为了查找针对该调用的候选函数，首先考虑 myClass 类的域。找到了两个成员函数 k(int)

和 k(char*)。因为在调用点可见的候选函数集只包含名字解析成功的域中声明的函数，所以不再在名字空间 NS 的域中查找候选函数，并且函数 k(double)被排除在候选函数集之外。

如果重载解析过程因为在候选函数集中没有最佳匹配函数而发现该调用是二义的，则该函数调用就是错误的。编译器不会进一步查找外围域中是否存在与函数调用实参匹配更好的其他候选函数。

15.10.4 对用户定义的转换序列划分等级

通过一个用户定义的转换序列，函数调用的实参可以被隐式地转换成函数参数的类型。用户定义的转换序列怎样影响函数重载解析过程呢？例如，给出如下的 calc()调用，哪个函数会被调用呢？

```
class SmallInt {
public
     SmallInt( int );
};

extern void calc( double );
extern void calc( SmallInt );
int ival;

int main() {
     calc( ival ); // 调用哪个 calc()?
}
```

被选中的是“参数与函数调用实参类型最佳匹配的函数”，该函数被称为最佳匹配或最佳可行函数。为了选择最佳可行函数，函数实参上的隐式转换被划分等级。最佳可行函数是这样的函数：其应用在函数实参上的转换不比调用其他可行函数所需的转换更差，而且在某些实参上应用的转换比调用其他可行函数在同样实参上需要应用的转换更好。

标准转换序列总是好于用户定义的转换序列。例如，对于前面例子中的 calc()调用，两个 calc()函数都是可行函数。calc(double)是一个可行函数，因为存在标准转换把 int 型的实参转换成参数的类型(double)。calc(SmallInt)也是一个可行函数，因为存在用户定义的转换可以把 int 型的实参转换成函数参数的类型 SmallInt。这里的用户定义转换使用构造函数 SmallInt(int)来完成。因为标准转换序列优于用户定义的转换序列，所以编译器为该调用选择可行函数 calc(double)作为最佳可行函数。

但是，如果比较两个用户定义的转换序列会怎么样呢？如果两个用户定义的转换序列使用不同的转换函数或不同的构造函数，则两个转换序列被认为程度一样好。例如：

```
class Number {
public:
     operator SmallInt();
     operator int();
     // ...
};
extern void calc( int );
extern void calc( SmallInt );
extern Number num;
```

```
calc( num ); // 错误：二义的
```

calc(int)和 calc(SmallInt)都是可行函数。calc(int)是一个可行函数，因为转换函数 Number::operator int()可以把 Number 类型的实参转换成参数类型 int。calc(SmallInt)也是转换函数，因为转换函数 Number::operator SmallInt()可以把 Number 类型的实参转换成函数参数类型 SmallInt。因为用户定义的转换序列的等级总是相同，所以编译器无法判定哪个用户定义的转换序列更好一些。因此，前面的函数调用是二义的，被标记为编译时刻错误。

程序可以通过显式指定转换来解决二义性问题，例如：

```
// 显式转换解决二义性
calc( static_cast< int >( num ) );
```

显式强制转换强迫编译器用转换函数 Number::operator int()把实参 num 转换成 int 型。那么，现在实参的类型是 int，与函数 calc(int)精确匹配，所以该函数被选为最佳可行函数。

我们假设 Number 类没有定义转换函数 Number::operator int()。那么，调用：

```
// 只定义了 Number::operator SmallInt()
calc( num );  // 还是二义的吗?
```

仍然是二义的吗？记住，SmallInt 类也定义了一个转换函数，它可以把 SmallInt 类型的值转换成 int 型的值。

```
class SmallInt {
public:
     operator int();
     // ...
};
```

有人可能认为仍然会调用函数 calc(int)，其过程是：先通过转换函数 Number::oerator SmallInt()把实参 num 从 Number 类型转换成 SmallInt 类型，然后再用转换函数 SmallInt::operator int()把结果转换为 int 型。但是，情况并不是这样。记住，用户定义的转换序列中只能有一个“用户定义的转换”，所以在第一个用户定义的转换之后只能考虑标准转换。如果没有定义 Number::operator int()，则编译器认为 calc(int)不是可行函数，因为没有隐式转换能够把实参 num 转换成函数参数的类型 int。

因此，如果没有定义 Number::operator int()，则 calc(SmallInt)就是惟一的可行函数。它将被选为最佳可行函数。

如果两个“用户定义的转换序列”使用了相同的转换函数，则转换函数后面的标准转换的等级被作为“选择最佳用户定义转换序列”的依据。例如：

```
class SmallInt {
public:
     operator int();
     // ...
};

void manip( int );
void manip( char );

SmallInt si ( 68 );
```

```
main() {
    manip( si ); // 调用 manip( int )
}
```

manip(int)和 manip(char)都是可行函数。manip(int)是一个可行函数，因为转换函数 SmallInt::operator int()可以把 SmallInt 类型的实参转换成参数类型 int。manip(char)也是一个可行函数，因为转换函数 SmallInt::operator int()可以把 SmallInt 类型的实参转换成 int 型，然后再通过标准转换把结果转换为 char 型。于是，用户定义的序列是：

```
manip(int) : operator int() -> 精确匹配
manip(char): operator int() -> 标准转换
```

因为两个用户定义的转换序列都使用了相同的转换函数，所以用标准转换的等级来决定最佳转换序列。因为精确匹配优于标准转换，所以选择函数 manip(int)作为最佳可行函数。

注意，如果两个“用户定义的转换序列”使用了相同的转换函数，则用户定义的转换后面的标准转换序列是选择的惟一标准。这与 15.9 节末给出的例子有点不同。在那一节，我们说明了编译器怎样选择用户定义的转换来把一个特殊类型的值转换成一个给定的目标类型。在那种情况下，源与目的类型都是固定的，编译器可以在所有能够转换这两种类型的不同的用户定义转换之间进行选择。而这里考虑的是两个不同的函数，带有不同的参数类型，且目标类型是变化的。如果两个参数类型要求不同的用户定义的转换，则不可能选择哪一个参数类型比另一个更好，除非用户定义的转换涉及到相同的转换函数。在这种情况下，我们可以用“在用户定义转换之后的标准转换”来选择最佳目标类型。例如：

```
class SmallInt {
public:
    operator int();
    operator float();
    // ...
};

void compute( float );
void compute( char );

SmallInt si ( 68 );

main() {
    compute( si ); // 二义的
}
```

compute(float)和 compute(char)都是可行函数。compute(float)是一个可行函数，因为转换函数 SmallInt::operator float()可以把 SmallInt 类型的实参转换成函数参数类型 float。compute(char)也是一个可行函数，因为转换函数 SmallInt::operator int()可以把 SmallInt 类型的实参转换成 int 型；然后再通过标准转换可以把结果转换成 char 型。所以用户定义的序列是：

```
compute(float): operator float() -> 精确匹配
compute(char): operator int() -> 标准转换
```

因为两个“用户定义的转换序列”使用了不同的转换函数，所以不能判定哪个函数对该调用有最佳的参数类型。标准转换序列的等级也不能被用来判定最佳转换序列，以及最佳参数类型。所以，该调用被编译器标记为二义的。

练习 15.12

C++标准库中的类没有定义转换函数，许多带一个实参的构造函数被声明为显式的（explicit）。但是，C++标准库中的类定义了许多重载的操作符。你认为选择这种设计方案的原因是什么？

练习 15.13

为什么在 15.9 节开始处定义的 SmallInt 类的重载输入操作符没有以下列方式实现？

```
istream& operator>>( istream &is, SmallInt &si )
{
    return ( is >> si.value );
}
```

练习 15.14

为下列每个初始化给出可能的用户定义的转换序列？每个初始化的结果是什么？

```
class LongDouble {
    operator double();
    operator float();
};
extern LongDouble ldObj;

(a)  int ex1 = ldObj;
(b)  float ex2 = ldObj;
```

练习 15.15

如果函数实参的类型是类（class）类型，请给出函数重载解析期间考虑的候选函数的三个集合。

练习 15.16

对于下列调用，哪个 calc()函数会被选为最佳可行函数？给出调用每个函数所需的转换序列，并说明选择最佳可行函数的原因。

```
class LongDouble {
public
    LongDouble( double );
    // ...
};

extern void calc( int );
extern void calc( LongDouble );
double dval;

int main() {
    calc( dval );// 哪个函数?
}
```

15.11 重载解析和成员函数 ※

成员函数也可以被重载。对于一个成员函数的调用，编译器也使用函数重载解析过程来选择最佳可行函数。成员函数的重载解析与非成员函数的重载解析十分类似。该过程由下列相同的三步骤组成：

1．选择候选函数；

2．选择可行函数；

3．选择最佳匹配函数。

为“成员函数调用”选择候选函数和可行函数，其做法与以前有一点小小的不同，我们将在本节介绍这些区别。

15.11.1 重载成员函数的声明

类的成员函数也可以被重载。例如：

```
class myClass {
public:
     void f( double );
     char f( char, char );   // 重载 myClass::f( double )
     // ...
};
```

如同在名字空间域中声明的函数一样，在类中声明的成员函数可以有相同的名字，只要参数表惟一：或者在参数的数目上不同，或者在参数的类型上不同。如果以同一名字声明的两个成员数只有返回类型不同，则第二个声明被视为错误的声明，被标记为编译时刻错误。例如：

```
class myClass {
public:
     void mf();
     double mf();       // 错误：不是有效的重载声明

     // ...
};
```

但是，不像名字空间函数，成员函数在类成员表中只能被声明一次。如果两个同名的成员函数声明的返回类型和参数表都完全匹配，则第二个声明被编译器标记为无效的成员函数重复声明。例如：

```
class myClass {
public:
     void mf();
     void mf();         // 错误：无效的重复声明

     // ...
};
```

重载函数集中的所有函数都在同一个域中被声明。因此，成员函数不会重载在名字空间域中声明的函数。又因为每个类都维护了自己的域，所以两个不同类的成员的函数也不会相

互重载。

重载成员函数集可以包含静态和非静态成员函数。例如：

```
class myClass {
public:
     void mcf( double );
     static void mcf( int* );    // 重载 myClass::mcf( double )
     // ...
};
```

调用静态还是非静态成员函数取决于函数重载解析的结果。下一节我们将详细讨论，当静态和非静态函数都是可行函数时，函数重载解析的情况。

15.11.2 候选函数

对下列形式的成员函数调用，

```
mc.mf( arg );
pmc->mf( arg );
```

如果 me 是一个 myClass 类型的表达式，而 pmc 是指向 myClass 类型的指针，那么这两个调用的候选成员函数集由“在 myClass 类的域中找到的 mf()函数声明”构成。

类似地，对形式为：

```
myClass::mf( arg );
```

的函数调用，候选函数集由“在 myClass 类域中查找 mf()声明时找到的函数”构成。例如：

```
class myClass {
public:
     void mf( double );
     char mf( char, char = '\n' );
     static void mf( int* );

     // ...
};

int main() {
     myClass mc;
     int iobj;
     mc.mf( iobj );
}
```

main()中函数调用的候选函数是 myClass 类中声明的三个成员函数 mf()：

```
void mf( double );
char mf( char, char = '\n' );
static void mf( int* );
```

如果在 myClass 类中不存在名为 mf()的成员函数，则候选函数集为空。（实际上，接下来会考虑基类中的函数。基类的成员函数怎样进入候选函数集将在 19.3 节中讨论。）如果对一个函数调用不存在候选函数，则该调用被标记为编译时刻错误。

15.11.3 可行函数

可行成员函数是来自候选成员函数集中的函数，它可以用调用中指定的实参表被调用。

在实参的类型和它的函数参数类型之间存在隐式类型转换。例如：

```
class myClass {
public:
     void mf( double );
     char mf( char, char = '\n' );
     static void mf( int* );
     // ...
};

int main() {
     myClass mc;
     int iobj;
     mc.mf( iobj ); // 哪一个成员函数 mf()？它是二义的
}
```

对 main()中的函数调用 mf()，存在两个可行成员函数：

```
void mf( double );
char mf( char, char = '\n' );
```

1．mf(doubl）是可行成员函数，因为它只有一个参数，并且存在标准转换可以把 im 型的实参 b 转换成参数类型 dOOble。

2.mf(cha,char)是可行成员函数，因为函数的第二个参数有缺省实参，并且存在标准转换可以把 int 型的实参 iobj 转换成第一个参数的类型 char。

在选择最可行成员函数时，在每个实参上的类型转换被划分等级。最佳可行成员函数是这样的可行成员函数：其应用在实参上的转换不比调用任何其他可行函数所需的转换更差，而且在某些实参上的转换要比调用其他可行函数在相同实参上所需的转换更好。

在上一个例子中，为了匹配两个可行成员函数的参数而被应用在实参上的转换都是标准转换。所以该函数调用是二义的，因为对于函数调用中指定的实参，两个成员函数一样好。

静态成员函数和非静态成员函数都可以被包含在可行函数集中，这与函数调用的形式无关。例如：

```
class myClass {
public:
     static void mf( int );
     char mf( char );
};

int main() {
     char cobj;
     myClass::mf( cobj ); // 哪个成员函数?
}
```

即使成员函数 mf()通过类名和域解析操作符［myClass()::mf()］来调用，或者 mf()没有通过对象或指向对象的指针和类成员访问操作符点（.）或箭头（->）来调用，非静态成员函数 mf(char)仍然被包含在该调用的可行函数集中，和静态成员函数 mf(int)一样。

函数重载解析过程接下来通过对“作用在函数实参上的类型转换”划分等级，以选择最佳可行函数。char 型的实参 cobj 是 mf(char)的参数的精确匹配。该实参通过“一个提升（promotion）”可以被转换成 mf(int)的参数类型。根据本例中应用在实参上的转换的等级，

就可以选择成员函数 mf(char)作为最佳可行函数。

但是，被选择的成员函数是非静态成员函数，不能被直接调用。它必须通过 myClass 类对象或对象的指针和成员访问操作符点（.）或箭头（->）才能被调用。那么，会发生什么事情呢？如果被选中的最佳可行函数是非静态成员函数，并且该调用不能真正发生，则会因为该调用没有指定对象（像本例这种情况），而使该调用被编译器标记为错误。

当选择可行函数集时，必须考虑成员函数的另一个方面，那就是非静态成员函数的 const 或 volatile 属性（13.3 节介绍了 const 和 volatile 成员函数）。这些属性又是怎样影响函数重载解析过程的呢？给出 myClass 类的下列成员函数：

```
class myClass {
public:
     static void mf( int* );
     void mf( double );
     void mf( int ) const;
     // ...
};
```

静态成员函数 mf(int*)、const 成员函数 mf(int)以及非 const 成员函数 mf(double)都被包含在下面函数调用的候选函数集中。又有哪些成员函数被包含在可行函数集中呢？

```
int main() {
     const myClass mc;
     double dobj;
     mc.mf( dobj ); // 哪个成员函数 mf()?
}
```

对于“可被应用到函数实参上的转换”进行检查，mf(double)和 mf(int)是可行函数。double 型的实参 dobj 是 mf(double)的参数的精确匹配。通过标准转换，实参 dobj 可以被转换成 mf(int)的参数的类型。

当通过点或箭头成员访问操作符调用成员函数时，在选择可行函数集中的函数时，需要考虑在调用成员函数时所使用的对象或指针的类型。

mc 是 const 对象。对于 const 对象，只有 const 非静态成员函数才可以被调用。因为不能调用非 const 非静态成员函数 mf(double)），所以它被排除在可行函数集之外。该调用的惟一可行函数是 const 成员函数 mf(int)，它被选为该调用的最佳可行函数。

如果用 const 对象调用静态成员函数会怎么样？静态成员函数不能被声明为 const 或 volatile，那么静态成员函数可以通过 const 对象而被调用吗？例如：

```
class myClass {
public:
     static void mf( int );
     char mf( char );
};
int main() {
     const myClass mc;
     int iobj;
     mc.mf( iobj ); // 可以调用静态成员函数吗?
}
```

静态成员函数对一个特定类类型的所有对象都是通用的。静态成员函数只能直接访问类

的静态成员。const 对象 mc 的非静态成员不能被静态成员函数 mf(int)访问，因此用点或箭头操作符调用 const 对象的静态成员函数是有效的。

所以静态成员函数从不会因为“被调对象或指针的限定修饰符 const 或 volatile”，而被排除在可行函数集之外。静态成员函数被认为匹配“它所在类类型的任何对象或指针”。

在前面的例子中，因为 mf 是 const 对象，所以成员函数 mf(char)被排除在可行函数集之外。但是成员函数 mf(int)被包含在可行函数集中，因为它是静态函数。因为它是该调用的惟一可行函数，所以 mf(int)被选为最佳可行函数。

15.12 重载解析和操作符 ※

正如在前面几节中所看到的，我们可以为类类型声明重载的操作符和转换函数。当遇到下面的初始化中的加法操作符时：

```
SomeClass sc;
int iobj = sc + 3;
```

编译器怎样判断是使用 SomeClass 类的重载加法操作符，还是“先使用转换函数把操作数 sc 转换成内置类型，然后再用内置的加法操作符”呢？

答案取决于为 SomeClass 类定义的重载操作符集和转换函数集。编译器通过函数重载解析过程，来选择用于执行加法的操作符。在本节中，我们将了解，当重载解析过程被用在类类型的操作数上时，它如何选择操作符。

针对重载操作符的重载解析过程也遵循 9.2 节中给出的三步骤，如下：

1．选择候选函数；

2．选择可行函数；

3．选择最佳可行函数。

我们将在本节详细讲述这三步。

如果一个操作符只有内置类型的操作数，则函数重载解析过程不会被应用。对于这样的操作数，编译器保证使用内置的操作符（关于“带有内置类型操作数的操作符”的用法在第 4 章讲述）。例如：

```
class SmallInt {
public:
     SmallInt( int );
};

SmallInt operator+ ( const SmallInt &, const SmallInt & );

void func() {
     int i1, i2;
     int i3 = i1 + i2;
}
```

因为 i1 和 i2 都是 int 型，不是类类型的操作数，所以用内置操作符做加法 i1+i2，而重载的操作符 operator+(const SmallInt&,const SmallInt&)被忽略，即使这些操作数可以通过用户定义的转换［调用构造函数 SmallInt(int)］被转换成 SmallInt 类型。本节描述的重载解析过

程不会用于这样的情形。

本节描述的操作符的重载解析过程也只适用于使用操作符语法时。例如：

```
void func() {
    SmallInt si(98);

    int iobj = 65;
    int res = si + iobj;          // 使用了操作符语法
}
```

如果使用了函数调用语法，如：

```
int res = operator+( si , iobj );  // 使用了函数调用语法
```

则应用 15.10 节讲述的名字空间函数的重载解析过程。如果使用了成员函数调用语法，如：

```
// 使用了成员函数调用语法
int res = si.operator+( iobj );
```

则使用 15.11 节描述的类成员函数的重载解析过程。

15.12.1 候选的操作符函数

候选的操作符函数是与被调用函数同名的函数。对下面加法操作符的用法：

```
SmallInt si(98);
int iobj = 65;
int res = si + iobj;
```

候选的操作符函数名为 operator+。考虑哪些 operator+()的声明呢？

潜在地，编译器为“使用操作符语法并带有类类型操作数的操作符”生成了五个候选操作符集合，前三个集合与“带有类类型实参的普通函数调用”的相同：

1．在调用点可见的操作符的集合。使用操作符时，可见的 operator+()的声明是候选操作符函数。例如，对于下面 main()中使用的 operator+()。在全局域中被声明的 operator+()是一个候选函数：

```
SmallInt operator+ ( const SmallInt &, const SmallInt & );

int main() {
    SmallInt si(98);
    int iobj = 65;
    int res = si + iobj;   // ::operator+() 是一个候选函数
}
```

2．在操作数类型被定义的名字空间中声明的操作符的集合。如果一个操作数的类型是类（class），并且该类被声明在用户声明的名字空间中，则在该名字空间声明的同名操作符函数都是候选操作符函数。例如：

```
namespace NS {
    class SmallInt { /* ... */ };
    SmallInt operator+ ( const SmallInt&, double );
}

int main() {
```

```
    // si 的类型是 SmallInt 类
    // 该类被声明在名字空间 NS 中
    NS::SmallInt si(15);

    // NS::operator+() 是一个候选函数
    int res = si + 566;
    return 0;
}
```

操作数 si 的类型是 SmallInt，它是在名字空间 NS 中被声明的类类型。在名字空间 NS 中的重载操作符 operator+(const SmallInt&,double)被加入到候选操作符函数集合中。

3．被声明为“操作数类类型的友元（friend）”的操作符集合。对于类类型的操作数，如果该类的定义声明了与所用操作符同名的友元操作符函数，则这些友元操作符函数被加入到候选函数集中。例如：

```
namespace NS {
    class SmallInt {
        friend SmallInt operator+(const SmallInt&, int )
            { /* ... */ }
    };
}

int main() {
    NS::SmallInt si(15);
    // friend operator+() 是一个候选函数
    int res = si + 566;
    return 0;
}
```

操作数 si 的类型是 SmallInt。它的友元操作符函数 operator+(const SmallInt&,int)是名字空间 NS 的成员，即使它没有直接被声明在名字空间 NS 中。在名字空间 NS 中的普通找法不会找到该友元操作符函数。但是，如果在使用 operator+()的时候带有 SmallInt 类型的实参，则编译器会考虑在 SmallInt 类域中声明的友元函数，并把它们加入到候选函数集中。

前面三个候选操作符函数集合的建立方式，与针对带有类类型实参的函数调用的候选函数集合相同。但是，对于以操作符语法使用的操作符，编译器还会建立另外两个候选操作符函数集合，它们组成了五个候选操作符函数集合。

4．在左操作数的类中被声明的成员操作符集合。如果 operator+()的左操作数是一个类类型，则通过在左操作数的类中查找成员 operator+()的声明，建立起成员操作符候选函数集合。例如：

```
class myFloat {
    myFloat( double );
};

class Smallint {
public:
    SmallInt( int );
    SmallInt operator+ ( const myFloat & );
};
int main() {
```

```
    SmallInt si(15);
    int res = si + 5.66; // 成员 operator+() 是一个候选函数
}
```

对于 main()中使用的 operator+()，在 SmallInt 类中定义的成员操作符 SmallInt::operator+(const myFloat&)被包含在候选函数集中。

5．内置操作符集合。已知可与内置 operator+()一起被使用的类型，对于内置的二元 operator+()。候选操作符函数如下：

```
int operator+( int, int )
double operator+( double, double )
T* operator+( T*, I )
T* operator+( I, T* )
```

第一个声明表示用于两个整型值相加的内置操作符。第二个声明表示用于两个浮点型值相加的内置操作符。第三和第四个声明表示针对指针类型的内置操作符，它可以被用来把一个整型值加到指针值上。这些声明只是符号性的。它们被用来描述内置操作符的候选集合，编译器对于任何加法操作都会考虑它们。

在组成前四个候选操作符函数集合时，有些候选集可能是空的。例如，如果在 SmallInt 类中没有找到名为 operator+()的成员函数，则成员候选操作符函数集合（即第四个集合）是空的。

候选的操作符函数集是前面列出的五个候选函数集的并集。例如：

```
namespace NS {
    class myFloat {
        myFloat( double );
    };

    class SmallInt {
        friend SmallInt operator+(const SmallInt &, int ) { /* ... */ }
    public:
        SmallInt( int );
        operator int();
        SmallInt operator+( const myFloat & );
        // ...
    };

    SmallInt operator+( const SmallInt &, double );
}

int main() {
    // si 的类型是类 SmallInt:
    // 该类在名字空间 NS 中声明
    NS::SmallInt si(15);
    int res = si + 5.66; // which operator+ ?
    return 0;
}
```

对于 main()中使用的 operator+()，这五个候选函数集合给出了七个候选操作符函数：

1．第一个候选函数集合是空的。在 main()中，operator+()是在全局域中被使用的，全局域中没有可见的重载操作符 operator+()的声明。

2．第二个候选函数集合包含了定义 SmallInt 类的名字空间 NS 中声明的操作符。下面的操作符在名字空间 NS 中被定义：

```
NS::SmallInt NS::operator+( const SmallInt&, double );
```

3．第三个候选函数集合包含了“被声明为 SmallInt 类的友元”的操作符。下面的操作符是 SmallInt 类的友元：

```
NSNS::SmallInt NS::operator+( const SmallInt &, int );
```

4．第四个候选函数集合包含了“被声明为 SmallInt 类的成员”的操作符。下面的操作符是 SmallInt 类的成员。

```
NS::SmallInt NS::SmallInt::operator+( const myFloat & );
```

5．第五个候选函数集合包含了内置的二元操作符：

```
int operator+( int, int )
double operator+( double, double )

T* operator+( T*, I )
T* operator+( I, T* )
```

呀！是的，为一个以操作符语法使用的操作符建立候选操作符函数表，确实需要做许多工作。一旦建立了候选函数集，则如同以前一样，通过分析可被应用在候选操作符的操作数上的转换，就可以找到可行函数和最佳可行函数。

15.12.2 可行函数

可行操作符函数集是从候选操作符函数集中选择出来的，方法是：只选择出那些“能够用（在操作符被使用时）指定的操作数来调用的操作符函数”。例如，在我们的例子中的七个候选操作符，哪些是可行函数？操作符的用法如下：

```
NS::SmallInt si(15);
si + 5.66;
```

左操作数的类型是 SmallInt，而右操作数的类型是 double。

第一个候选函数是 operator+()这种用法的可行函数：

```
NS::SmallInt NS::operator+( const SmallInt &, double );
```

SmallInt 类型的左操作数 si 是“重载操作符的引用参数的初始值”的精确匹配。而右操作数是 double 型的值，它也是重载操作符的第二个参数的精确匹配。

第二个候选函数也是 operator+()这种用法的可行函数：

```
NS::SmallInt NS::operator+( cons t SmallInt &, int );
```

SmallInt 类型的左操作数 si 是“重载操作符的引用参数的初始值”的精确匹配。右操作数是 int 型的值，通过标准转换，它可以被转换成重载操作符的第二个参数。

第二个候选函数也是 operator+()这种用法的可行函数：

```
NS::SmallInt NS::SmallInt::operator+( const myFloat & );
```

SmallInt 类型的左操作数 si 是一个类类型的对象，在该类中。这个重载操作符被定义为一个成员函数。右操作数是 int 型的值，它也能够通过用户定义的转换序列［用构造函数

myFloat(double)］被转换成 myFloat 类类型。

第四和第五个可行函数是内置操作符：

```
int operator+( int, int )
double operator+( double, double )
```

SmallInt 类含有一个转换函数，它可以把 SmallInt 的值转换成 int 型的值。对于第一个内置操作符，通过这个转换函数，可以把 SmallInt 类型的左操作数转换成 int 型，通过标准转换，可以把 double 型的第二个操作数转换成 int 型。对于第二个内置操作符，通过转换函数，可以把 SmallInt 类型的左操作数转换成 int 型，然后通过标准转换把结果转换成 double 型的值；double 型的第二个操作数是第二个参数的精确匹配。

在五个可行函数中，最佳可行函数是第一个可行函数，即在名字空间 NS 中声明的重载的 operator+()：

```
NS::SmallInt NS::operator+ ( const SmallInt &, double );
```

对于这个重载操作符，两个操作数都与参数完全匹配。

15.12.3 二义性

为同一个类类型既提供转换函数来实现“到内置类型之间的隐式转换”，又提供重载的操作符，可能会导致在重载操作符和内置操作符之间的二义性。例如，给出下面的 String 类，它带有相应的比较函数：

```
class String {
// ...
public:
     String( const char * = 0 );
     bool operator== ( const String & ) const;

     // 没有提供 operator== ( const char * )
};
```

以及下列 operator==()的用法：

```
String flower( "tulip" );

void foo( const char *pf ) {
     // 调用重载的 String::operator==()
     if ( flower == pf )
          cout << pf << " is a flower!\n";

     // ...
}
```

如下比较操作：

```
flower == pf
```

调用了 String 的等于成员操作符：

```
String::operator==( const String & ) const;
```

而如下调用了构造函数的用户定义转换：

```
String( const char * )
```

被用来把右操作数 pf 从 const char*型转换成 String 类型：即成员 operator==()的参数类型。

如果转换函数操作符 const char*()被加入到 String 类的定义中：

```
class String {
    // ...
public:
    String( const char * = 0 );
    bool operator== ( const String & ) const;
    operator const char*(); // 新的转换函数
};
```

则前面 operator==()的用法是二义的：

```
// 等于测试不能通过编译!
if ( flower == pf )
```

因为引入了转换函数操作符 const char*()，所以内置的比较操作符：

```
bool operator==( const char *, const char * )
```

现在也是可行函数。通过新的用户定义转换，String 类型的左操作数 flower 可以被转换成 const char*型。

现在对 foo()中 operator==()的用法有两个可行操作符函数，第一个可行函数：

```
String::operator==( const String & ) const;
```

要求一个用户定义的转换，以便把右操作数 pf 从 const char*型转换成 String 类型。第二个可行函数：

```
bool operator==( const char * , const char * )
```

也要求一个用户定义的转换，以便把左操作数 flower 从 String 类型转换成 const char*型。

所以，第一个可行函数对于左操作数比较好，而第二个可行函数对于右操作数比较好。于是，该调用被标记为二义的，这是因为没有找到最佳可行函数。

这就是为什么我们要在设计类的接口时，以及为特殊的类类型声明重载操作符、构造函数和转换函数时必须小心谨慎的原因。用户定义的转换被编译器隐式地应用，可能使得当一个操作符与类类型的操作数一起被使用时，内置操作符变成可行函数。因此，我们应该明智地使用转换函数和非显式的构造函数（即没有被声明为 explicit 的构造函数）

练习 15.17

请指出在函数重载解析过程中，如果操作符被用在类类型的操作数上，编译器应该考虑的五个候选函数集。

练习 15.18

下列哪个 operator+()被选为 main()中加法操作的最佳可行函数？请列出候选函数集、可行函数集，以及针对每个可行函数的实参的类型转换。

```
namespace NS {
    class complex {
```

```
        complex( double );
        // ...
    };

    class LongDouble {
        friend LongDouble operator+( LongDouble &, int ) { /*...*/ }
    public:
        LongDouble( int );
        operator double();
        LongDouble operator+( const complex & );
        // ...
    };

    LongDouble operator+( const LongDouble &, double );
}

int main() {
    // si 的类型是 SmallInt 类
    // 该类被声明在名字空间 NS 中
    NS:: LongDouble ld(16.08);

    double res = ld + 15.05; // which operator+ ?
    return 0;
}
```

16

类 模 板

本章将讲述类模板。以及怎样定义和使用它们。类模板是一种“规范描述（prescription）”，规定了如何创建一个类，而且在这样的类中有一个或多个类型或值被参数化、C++初学者可以直接使用类模板，而无需了解模板定义和初始化背后的机制，这完全是有可能的。实际上，在这本书中，我们已经使用了 C++标准库中定义的类模板（像 vector、list 等等），而没有详细描述模板机制。只有较高级的 C++程序员才会定义自己的类模板，并使用本章描述的机制。因此，本章的内容是 C++的高级话题的介绍资料。

本章分为入门和高级两部分。入门部分将给出怎样定义类模板，并说明类模板的简单用法，同时将讨论怎样实例化类模板。在入门部分还将看到，我们可以为类模板定义不同类型的成员：成员函数、静态数据成员和嵌套的类型。高级部分将给出一些只有写产品级应用程序才需要的资料。我们将首先查看编译器怎样实例化模板，及其对程序组织的要求。接着、我们将给出怎样为类模板或者类模板的一个成员定义特化和部分特化。然后，本章将介绍两个对类模板的设计者感兴趣的话题：类模板定义中的名字怎样被解析，以及怎样在名字空间中定义类模板。本章将以一个定义并使用类模板的较大规模的例子作为结束。

16.1 类模板定义

假设我们想定义一个类来支持队列（queue）的机制。队列是一个专门用于对象集合的数据结构，对象被加入到队列的尾部，而从队列的顶部被删除。队列的行为被称为先进先出（first in first out），或 FIFO。（C++标准库定义的个队列类型在 16.7 节已经简要描述过。在本章中，我们将通过定义一个自己的简单队列类型来介绍类模板。）

我们决定，我们的 Queue（队列）类将支持下列操作：

- 在队尾加入一项：

```
void add( item );
```

- 从队首删除一项：

```
item remove();
```

- 判断队列是否为空：

```
bool is_empty();
```

- 判断队列是否已满：

```
bool is_full();
```

Queue 类的定义看起来可能是这样的：

```
class Queue {
public:
    Queue();
    ~Queue();

    Type& remove();
    void add( const Type & );

    bool is_empty();
    bool is_full();
private:
    // ....
};
```

问题是，对于 Type，我们应该使用什么类型？假设我们选择用 int 代替 Type 来实现我们的 Queue 类，那么 Queue 类就被定义来处理 int 型对象的集合。如果程序员把另一种类型的值赋给这些对象，则被赋的值将被转换为 int 型，或者如果不存在转换，则这个赋值将被标记为编译时刻错误。例如：

```
Queue qObj;
string str( "vivisection" );

qObj.add( 3.14159 );     // ok: 加入到队列中的项 == 3
qObj.add( str );         // 错误: 从 string 到 int 没有转换
```

因为集合中的每个对象都是 int 型的，所以 C++类型系统保证只有 int 型的值或者能被转换成 int 型的值，才能被赋给 Queue 类型的对象。当然，当程序员希望使用 int 型对象的队列时，这很不错。但是，当程序员希望用 Queue 类表示 double、char、复数或者 string 型对象的集合时，就不好了。

解决问题的一种方法是简单地使用“蛮力”，程序员拷贝整个 Queue 类的实现，并修改它，使其能够在 double 类型上进行工作，然后是复数、string 等类型。由于类名不能被重载，所以必须对每一个实现给予惟一的名字：IntQueue、DoubleQueue、ComplexQueue 和 StringQueue。每当需要一个新类时，我们就拷贝、修改并重命名代码。

这种类类型复制的方法有什么问题呢？每一个 Queue 都有一个惟一的名字，这就有一个词汇复杂性的问题。而且，也会导致管理上的复杂性——想像一下，Queue 是一个通用的实现，从 Queue 到 IntQueue 类需要做一次修改，对于每一个特定的实例都需要做这样的修改操作。一般来说，为单个类型提供手工生成的拷贝是一个无休止的过程，也是一个无限复杂的维护过程。

幸运的是，C++为自动生成类类型提供了模板机制。我们可以用类模板为每个特殊类型的队列自动生成 Queue 类。Queue 类的模板定义看起来可能是这样的：

```
template <class Type>
class Queue {
public:
     Queue();
     ~Queue();

     Type& remove();
     void add( const Type & );

     bool is_empty();
     bool is_full();
private:
     // ....
};
```

程序员用下面的代码：

```
Queue<int> qi;
Queue< complex<double> > qc;
Queue<string> qs;
```

依次生成 int、复数和 string 类型的 Queue 类。

Queue 类的实现将在随后几节中给出，以便说明类模板的定义和用法。该实现使用了一对类模板抽象：

1．类模板 Queue 本身提供了前面描述的公有接口以及一对数据成员：front 和 back。类模板 Queue 被实现为一个链表。

2．类模板 QueueItem 表示 Queue 链表的一个节点。加入到队列中的每一项都被存储在一个 QueueItem 对象中。QueueItem 对象含有一对数据成员：value 和 next。value 的实际类型随 Queue 实例的变化而变化，next 是指向队列中下一个 QueueItem 对象的链接。

在更详细地查看这些模板的实现之前，让我们先仔细看看怎样声明和定义模板。下面是类模板 QueueItem 的声明：

```
template <class T>
     class QueueItem;
```

类模板的定义和声明都以关键字 template 开头，关键字后面是一个用逗号分隔的模板参数表，用尖括号（<>）括起来。这个表被称为类模板的模板参数表（template parameter list），它不能为空。模板参数可以是一个类型参数，也可以是一个非类型参数。如果是非类型参数，则代表一个常量表达式。

模板的类型参数（type parameter）由关键字 class 或关键字 typename 及其后的标识符构成。在模板参数表中，关键字 class 和 typename 的意义相同。（在标准 C++之前，关键字 typename 没有被支持。10.1 节更详细地讨论了把这个关键字加入到 C++中的原因：因为有时必须要靠它来指导编译器解释模板定义。）这两个关键字表明后面的参数名代表一个内置的或用户定义的类型。例如，在前面给出的类模板 QueueItem 的前向声明中，有一个名为 T 的模板类型参数。任何内置的或用户定义的类型，如 int、double、char*、complex 或 string 都是 T 的有效实参。

一个类模板可以有多个类型参数：

```
template <class T1, class T2, class T3>
```

```
    class Container;
```

但是，每个模板类型参数的前面都必须有关键字 class 或 typename。例如，下面的模板声明是错误的：

```
// 错误：必须是 <typename T, class U> 或 <typename T, typename U>
template <typename T, U>
    class collection;
```

一旦声明了类型参数。那么在类模板定义的余下部分中，它就可以被用作类型指示符。它在类模板中的使用方式与“内置的或用户定义的类型在非模板类定义中的用法”一样。例如，类型参数可以被用来声明数据成员、成员函数和嵌套类的成员等等。

模板**非类型参数**（nontype parameter）由一个普通参数声明构成。一个非类型参数指示该参数代表了一个潜在的值，而这个值又代表类模板定义中的一个常量。例如，一个 Buffer 类模板可以有一个类型参数来表示它所包含的元素类型，和一个非类型参数来表示其大小的常量值。例如：

```
template <class Type, int size>
    class Buffer;
```

一个类定义或声明紧跟在模板参数表后面。除了模板参数外，类模板的定义看起来和非模板类相同。

```
template <class Type>
class QueueItem {
public:
    // ...
private:
    // Type 表示数据成员的类型
    Type item;
    QueueItem *next;
};
```

在本例中，Type 被用来指明数据成员 item 的类型。在程序中，Type 会被各种内置的和用户定义的类型代替。类型替换的过程被称为**模板实例化**（template instantiation）。

模板参数的名字，在它被声明为模板参数后，一直到模板声明或定义的结束，都可以被使用。如果在全局域中声明了与模板参数同名的变量，则该变量被隐藏掉。在下面的例子中，item 的类型不是 double，它的类型是模板参数的类型：

```
typedef double Type;

template <class Type>
class QueueItem {
public:
    // ...
private:
    // item 不是 double 类型
    Type item;
    QueueItem *next;
};
```

模板参数名不能被用作在类模板定义中声明的类成员的名字：

```
template <class Type>
class QueueItem {
public:
    // ...
private:
    // 错误：成员名不能与模板参数 Type 同名
    typedef double Type;
    Type item;
    QueueItem *next;
};
```

模板参数的名字在模板参数表中只能被引入一次。例如，下面语句将被标记为编译时刻错误：

```
// 错误：重复使用名为 Type 的模板参数
template <class Type, class Type>
    class container;
```

在不同的类模板声明或定义之间，模板参数的名字可以被重复使用：

```
// ok: 名字 'Type' 在不同模板之间可被重复使用
template <class Type>
    class QueueItem;

template <class Type>
    class Queue;
```

在类模板的前向声明和类模板定义中，模板参数的名字可以不同。例如，下面三个 QueueItem 都引用同一个类模板：

```
// 所有三个 QueueItem 声明都引用同一个类模板
// 模板的声明
template <class T> class QueueItem;
template <class U> class QueueItem;

// 模板的真正定义
template <class Type>
    class QueueItem { ... };
```

类模板的参数可以有缺省实参，这对类型参数和非类型参数都一样。像函数参数的缺省实参一样在 7.3 节介绍），模板参数的缺省实参是一个类型或值。当模板被实例化时，如果没有指定实参，则使用该类型或者值。缺省实参应该是一个“对类模板实例的多数情况都适合”的类型或值。在下面的例子中，如果模板实例的名字没有指定 Buffer 的大小，则实例化的 Buffer 的大小是 1024 个项。

```
template <class Type, int size = 1024>
    class Buffer;
```

类模板的后续声明可以为模板参数提供附加的缺省实参。正如函数参数的缺省实参的情形一样，在向左边的参数提供缺省实参之前，必须首先给最右边未初始化的参数提供缺省实参。例如：

```
template <class Type, int size = 1024>
    class Buffer;
```

```
// ok: 考虑两个声明中的缺省实参
template <class Type = string , int size>
     class Buffer;
```

（注意，标准 C++之前的编译器并不支持模板参数的缺省实参。在本书的许多例子中，例如第 12 章中的例子，为了能够在标准 C++之前的编译器上通过编译，它们都不用模板参数的缺省实参。）

在类模板定义中，类模板的名字可以被用作一个类型指示符，凡是可以使用非模板类名的地方都可以用它。例如，下面是一个更完整的 QueueItem 类模板的版本：

```
template <class Type>
class QueueItem {
public:
     QueueItem( const Type & );
private:
     Type item;
     QueueItem *next;
};
```

我们注意到，在类模板定义中，QueueItem 类模板名的每次出现都是以上形式的缩写：

```
QueueItem<Type>
```

这种简写形式只能被用在类模板 QueueItem 自己的定义中（以及在类模板定义之外出现的成员定义中，在接下去几节我们将会看到）。当 QueueItem 在其他模板定义中被用作一个类型指示符时，我们必须指定完整的模板参数表。在下面的例子中，类模板被用在函数模板 display 的定义中。在这种情况下，类模板 QueueItem 的名字必须跟有模板参数，就像在 QueueItem<Type>中一样：

```
template <class Type>
void display( QueueItem<Type> &qi )
{
     QueueItem<Type> *pqi = &qi;

     // ...
}
```

16.1.1 Queue 和 QueueItem 类模板的定义

下面是类模板 Queue 的定义，它和类模板 QueueItem 的定义都被放在一个名为 Queue.h 的头文件中：

```
#ifndef QUEUE_H
#define QUEUE_H

// QueueItem 的声明
template <class T> class QueueItem;

template <class Type>
class Queue {
public:
     Queue() : front( 0 ), back ( 0 ) { }
```

```
    ~Queue();
    Type& remove();

    void add( const Type & );
    bool is_empty() const {
        return front == 0;
    }
private:
    QueueItem<Type> *front;
    QueueItem<Type> *back;
};
#endif
```

在类模板 Queue 的定义中，使用名字 Queue 时可以省略参数表<type>。但是，当 Queue 的定义引用到类模板 QueueItem 时不能省略参数表。例如，下面的 front 声明就是错误的：

```
template <class Type>
class Queue {
public:
    // ...
private:
    // 错误：QueueItem 不是一个已知类型
    QueueItem *front;
};
```

练习 16.1

请指出下列模板类声明（或声明对）中哪些是非法的。

```
(a)  template <class Type>
         class Container1;
     template <class Type, int size>
         class Container1;

(b)  template <class T, U, class V>
         class Container2;

(c)  template <class C1, typename C2>
         class Container3 {};

(d)  template <typename myT, class myT>
         class Container4 {};

(e)  template <class Type, int *ptr>
         class Container5;
     template <class T, int *pi>
         class Container5;

(f)  template <class Type, int val = 0>
         class Container6;
     template <class T = complex<double>, int v>
         class Container6;
```

练习 16.2

下面 List 的定义不正确，应该怎样改正它？

```
template <class elemenType>
class ListItem;

template <class elemType>
class List {
public:
     List<elemType>()
          : _at_front( 0 ), _at_end( 0 ), _current( 0 ), _size( 0 )
          {}
     List<elemType>( const List<elemType> & );
     List<elemType>& operator=( const List<elemType> & );

     ~List();

     void insert( ListItem *ptr, elemType value );
     int remove( elemType value );
     ListItem *find( elemType value );
     void display( ostream &os = cout );
     int size() { return _size; }
private:
     ListItem *_at_front;
     ListItem *_at_end;
     ListItem *_current;
     int _size;
};
```

16.2 类模板实例化

类模板定义指定了怎样根据一个或多个实际的类型或值的集合来构造单独的类。Queue 的类模板定义被用作“Queue 类的特定类型的实例”的自动生成模板。例如，当程序员这样写时：

```
Queue<int> qi;
```

一个针对 int 型对象的 Queue 类就被从通用类模板定义中创建出来。

从通用的类模板定义中生成类的过程被称为**模板实例化**（template instantiation），当一个针对 int 型对象的 Queue 类被实例化时，类模板定义中每次出现的模板参数 Type 都被 int 取代。针对 int 的 Queue 类逐字变成：

```
class Queue<int> {
public:
     Queue<int>() : front( 0 ), back ( 0 ) { }
     ~Queue<int>();

     int& remove();
     void add( const int & );
```

```
        bool is_empty() const {
            return front == 0;
        }
    private:
        QueueItem<int> *front;
        QueueItem<int> *back;
    };
```

类似地，为了创建一个针对 string 类型对象的 Queue 类，程序员要写：

```
Queue<string> qs;
```

在这种情况下，类模板定义中出现的模板参数 Type 被类型 string 取代。对象 qi 和 qs 都是类类型的对象。

同一个类模板针对不同类型的实例之间并没有特殊的关系。类模板的每个实例都构成一个独立的类类型。例如，int 型的 Queue 实例没有权利访问 string 类型的 Queue 实例的非公有成员。

类模板实例的名字是 Queue<int>或 Queue<string>。在类模板名 Queue 后面的符号<int>或<string>被称为模板实参。模板实参必须是在一个由逗号分隔的列表中被指定，并放在一对尖括号（<>，一个小于号和一个大于号）中。类模板的实例名必须总是显式地指定模板实参。与函数模板实例的模板实参不同的是，根据类模板实例被使用的上下文环境，编译器无法推断出类模板实例的模板实参：

```
Queue qs;      // 错误：哪一个模板实例?
```

类模板 Queue 的实例可以被一般的程序使用，凡是能够使用非模板类型的地方就可以用它：

```
// 返回类型和两个参数都是 Queue 的实例
extern Queue< complex<double> >
foo( Queue< complex<double> > &, Queue< complex<double> > & );

// 指向 Queue 实例的成员函数的指针
bool (Queue<double>::*pmf)() = 0;

// 从 0 到 Queue 实例的显式强制转换
Queue<char*> *pqc = static_cast< Queue<char*>* > ( 0 );
```

类模板 Queue 的实例类型的对象的声明和使用方式，与非模板类类型的对象相同：

```
extern Queue<double> eqd;

Queue<int> *pqi = new Queue<int>;
Queue<int> aqi[1024];

int main() {
    int ix;
    if ( ! pqi->is_empty() )
        ix = pqi->remove();
    // ...
    for ( ix = 0; ix < 1024; ++ix )
        eqd[ ix ].add( ix );
    // ...
}
```

模板声明或定义可以引用类模板或类模板的实例：

```
// 函数模板声明
template <class Type>
void bar( Queue<Type> &,          // 引用通用的模板
     Queue<double> &              // 和模板实例
)
```

但是，在模板定义的上下文环境之外，只能使用类模板实例。例如，非模板函数必须指定它使用类模板 Queue 的哪个特殊实例：

```
void foo( Queue<int> &qi )
{
     Queue<int> *pqi = &qi;

     // ...
}
```

只有当代码中使用了类模板的一个实例的名字，并且上下文环境要求必须存在类的定义时，这个类模板才被实例化。并不是每次使用一个类都要求知道该类的定义。例如，如果我们只是声明一个类的指针和引用，就没有必要知道类的定义。例如：

```
class Matrix;

Matrix *pm;                       // ok: 不需要类 Matrix 的定义
void inverse( Matrix & );         // ok 也不需要
```

所以，声明一个类模板实例的指针和引用不会引起类模板被实例化。（这里我们应该指出，有些标准 C++之前的编译器在程序文本中第一次遇到实例名时，就实例化该模板。）例如，下面的函数 foo()声明了类模板实例 Queue<int>的一个指针和一个引用。但是，这些声明并没有引起模板 Queue 被实例化：

```
// Queue<int> 没有为其在 foo() 中的使用实例化
void foo( Queue<int> &qi )
{
     Queue<int> *pqi = &qi;

     // ...
}
```

定义一个类类型的对象时需要该类的定义。例如，在下面的例子中，obj1 的定义就是错的。这个对象的定义要求让编译器知道 Matrix 的大小，以便为 obj1 分配正确的内存数：

```
class Matrix;
Matrix obj1;                      // 错误: Matrix 没有被定义

class Matrix { ... };
Matrix obj2;                      // ok
```

所以，如果一个对象的类型是一个类模板的实例，那么当对象被定义时，类模板也被实例化。在下面的例子中，对象 qi 的定义引起模板 Queue<int>被实例化：

```
Queue<int> qi;                    // Queue<int> 被实例化
```

在这一点上，Queue<int>类的定义对于编译器变成已知的，该点被称为类 Queue<int>的**实例化点**（point of instantiation）。

类似地，如果一个指针或引用指向一个类模板实例，那么只有当检查这个指针或引用所指的那个对象时，类模板才会被实例化。在前面定义的函数 foo()中，如果指针 pqi 被解引用，qi 被用来获得它指向的对象值，或者 pqi 或 qi 被用来访问 Queue<int>的数据成员或成员函数时，Queue<int>才会被实例化：

```
void foo( Queue<int> &qi )
{
    Queue<int> *pqi = &qi;

    // 因为成员函数被调用，所以 Queue<int> 被实例化
    pqi->add( 255 );

    // ...
}
```

编译器必须在 foo()调用 Queue<int>类的成员函数 add()之前，先知道 Queue<int>类的定义。

回忆前面，类模板 Queue 的定义引用了类模板 QueueItem，如下所示：

```
template <class Type>
class Queue {
public:
    // ...
private:
    QueueItem<Type> *front;
    QueueItem<Type> *back;
};
```

当针对类型 int 实例化 Queue 时，实例 Queue<int>的成员 front 和 back 是指向 QueueItem<int>的指针。所以 Queue<int>的实例引用了类模板 QueueItem 的 int 型实例。但是，因为这些成员是指针，所以，只有当这些成员在 Queue<int>类的成员函数中被解引用时，类型 QueueItem<int>才被实例化。

我们已经决定把 QueueItem 作为辅助类来帮助实现 Queue 类，它不希望被普通程序使用。所以，普通程序只能操纵 Queue 类对象。类模板 QueueItem 的实例化只能由“类模板 Queue 的实例化及其成员的实例化”引起。在后面的几节中，我们将了解类模板成员的实例化。

在定义模板时，根据模板被实例化的类型，我们必须做一些设计上的考虑。例如，你知道为什么下面的 QueueItem 的构造函数定义可能无法被一个大范围的类型实例所接受吗？

```
template <class Type>
class QueueItem {
public:
    QueueItem( Type );   // 不好的设计选择
    // ...
};
```

QueueItem 构造函数的定义实现了按值传递的实参语意。当 QueueItem 被一个内置类型（比如在 QueueItem<int>的实例中）实例化时，这足以完成任务。但是，当 QueueItem 被一个大型的类类型（例如 Matrix）实例化时，这种选择所带来的运行时刻影响是不能被接受的（7.3 节讨论了声明传值参数和声明引用参数的性能影响）。这也正是构造函数被声明为 const 引用类型的原因：

```
QueueItem( const Type& );
```

另外一种设计考虑在于这个构造函数的实现之中。如果用来实例化 QueueItem 的类型没有相关的构造函数，则可以接受下面的构造函数定义：

```
template <class Type>
class QueueItem {
    // ...
public:
    // 可能效率很低
    QueueItem( const Type &t ) {
        item = t; next = 0;
    }
};
```

如果模板实参是一个具有构造函数的类（例如 string），它将导致 item 被初始化两次！在 QueueItem 的构造函数体执行之前，string 的缺省构造函数被调用来初始化 item。然后，新构造的 item 又被按成员赋值。在 QueueItem 构造函数的定义中，我们只需在构造函数成员初始化表中显式地初始化 item，就可以解决这个问题：

```
template <class Type>
class QueueItem {
    // ...
public:
    // 在构造函数成员初始化表中初始化 item
    QueueItem( const Type &t )
        : item(t) { next = 0; }
};
```

（14.5 节讨论了成员初始化表，以及何时、怎样使用它们。）

16.2.1 非类型参数的模板实参

类模板参数也可以是一个非类型模板参数。对“可以被用于这种非类型模板参数的模板实参”的种类有一些限制，在这里我们将介绍这些限制。下面的例子使用了第 13 章中介绍的 Screen 类。这里把它重新定义为模板，将它的高度（height）和宽度（width）参数化：

```
template <int hi, int wid>
class Screen {
public:
    Screen() : _height( hi ), _width( wid ), _cursor ( 0 ),
        _screen( hi * wid, `#' )
        { }
    // ...
private:
    string _screen;
    string::size_type _cursor;
    short _height;
    short _width;
};

typedef Screen<24,80> termScreen;
termScreen hp2621;
```

```
Screen<8,24> ancientScreen;
```

绑定给非类型参数的表达式必须是一个常量表达式。即，它必须能在编译时被计算出结果。在前面的例子中，typedef termScreen 引用到模板实例 Screen<24,80>。hi 的模板实参是 24，而 wid 的实参是 80。在这两种情况下，模板实参都是常量表达式。

但是，如果给定下面定义的类模板 BuffPtr，那么它的实例将导致编译错误，因为来自操作符 new()调用结果的指针值只有到运行时刻才能被知道：

```
template <int *ptr> class BufPtr { ... };

// 错误：模板实参不能在编译时刻被计算出来
BufPtr< new int[24] > bp;
```

类似地，非 const 对象的值不是一个常量表达式，它不能被用作非类型模板参数的实参。但是，名字空间域中任何对象的地址（即使该对象不是 const 类型）是一个常量表达式（而局部对象的地址则不是）。因此，名字空间域的对象的地址可以被用作非类型模板参数的实参。类似地，sizeof 表达式的结果是一个常量表达式，所以它可以被用作非类型模板参数的实参：

```
template <int size> Buf{ ... };
template <int *ptr> class BufPtr { ... };

int size_val = 1024;
const int c_size_val = 1024;

Buf< 1024 > buf0;                // ok
Buf< c_size_val > buf1;          // ok
Buf< sizeof(size_val) > buf2;    // ok: sizeof(int)
BufPtr< &size_val > bp0;         // ok

// 错误：不能在编译时刻被计算出来
Buf< size_val > buf3;
```

这里的另一个例子说明了非类型模板参数怎样被用来表示类模板定义中的常量值，以及怎样用模板实参为该模板参数指定一个值：

```
template <class Type, int size>
class Fixed_Array {
public:
     Fixed_Array( Type *ar) : count( size )
     {
          for ( int ix = 0; ix < size; ++ix )
               array[ ix ] = ar[ ix ];
     }
private:
     Type array[ size ];
     int count;
};

int ia[4] = { 0, 1, 2, 3 };
Fixed_Array< int, sizeof( ia ) / sizeof( int ) > iA( ia );
```

对于一个模板非类型参数，如果两个不同的表达式的求值结果相同，则它们被认为是等

价的模板实参。例如，下面三个 Screen 实例都引用到同一个模板实例 Screen<24.80>中：

```
const int width = 24;
const int height = 80;

// 三者都是类型 Screen< 24, 80 >
Screen< 2*12, 40*2 > scr0;
Screen< 6+6+6+6, 20*2+40 > scr1;
Screen< width, height > scr2;
```

在模板实参的类型和非类型模板参数的类型之间允许进行一些转换，能被允许的转换集是“函数实参上被允许的转换”的子集：

1．左值转换，包括从左值到右值的转换、从数组到指针的转换，以及从函数到指针的转换。例如：

```
template <int *ptr> class BufPtr { ... };

int array[10];
BufPtr< array > bpObj;      // 数组到指针的转换
```

2．限定修饰转换。例如：

```
template <const int *ptr> class Ptr { ... };

int iObj;
Ptr< &iObj > pObj;          // 从 int* 到 const int* 的转换
```

3．提升，例如：

```
template <int hi, int wid> class Screen { ... };

const short shi = 40;
const short swi = 132;
Screen< shi, swi > bpObj2;  // 从 short 到 int 的提升
```

4．整值转换，例如：

```
template <lunsigned int size> Buf{ ... };
Buf<l 1024 > bObj;          // 从 int 到 unsigned int 的转换
```

（这些转换在 9.3 节详细讨论过）

再考虑上面一组声明：

```
extern void foo( char * );
extern void bar( void * );

typedef void (*PFV)( void * );
const unsigned int x = 1024;

template <class Type,
     unsigned int size,
     PFV handler> class Array { ... };

Array<int, 1024U, bar> a0;   // ok: 不需要转换
Array<int, 1024U, foo> a1;   // 错误: foo != PFV
```

```
Array<int, 1024, bar> a2;    // ok: 1024 被转换成 unsigned int
Array<int, 1024, foo> a3;    // 错误: foo != PFV

Array<int, x, bar> a4;       // ok: 不需要转换
Array<int, x, foo> a5;       // 错误: foo != PFV
```

Array 类对象 a0 和 a4 定义正确，因为模板实参与相应的模板参数完全匹配。Army 类对象 a2 定义也正确，因为 int 型的模板实参 1024 被通过一个有序转换，转换成非类型模板参数 size 的类型（unsigned int）。Array 类对象 a1、a3 和 a5 的声明是错误的，因为在任何两个函数类型之间不存在转换。

下面把整型 0 转换成指针值的转换是不允许的：

```
template <int *ptr>
class BufPtr { ... };

// 错误: 0 的类型是 int
// 不能通过“隐式转换”隐式地转换到空指针
BufPtr< 0 > nil;
```

练习 16.3

请指出下列哪些模板实例的使用会引起模板被实例化。

```
template < class Type >
class Stack { };

void f1( Stack< char > );                    // (a)

class Exercise {
     // ...
     Stack< double > &rsd;                   // (b)
     Stack< int > si;                        // (c)
};

int main() {
     Stack< char > *sc;                      // (d)
     f1( *sc );                              // (e)
     int iObj = sizeof( Stack< string > );   // (f)
}
```

练习 16.4

请指出下列哪些模板实例化是有效的，并说明原因。

```
template < int *ptr > class Ptr { ... };
template < class Type, int size > class Fixed_Array { ... };
template < int hi, int wid > class Screen { ... };

(a)  const int size = 1024;
     Ptr< &size > bp1;
(b)  int arr[10];
```

```
        Ptr< arr > bp2;
(c)     Ptr < 0 > bp3;
(d)     const int hi = 40;
        const int wi = 80;
        Screen< hi, wi+32 > sObj;
(e)     const int size_val = 1024;
        Fixed_Array< string, size_val > fa1;
(f)     unsigned int fasize = 255;
        Fixed_Array< int, fasize > fa2;
(g)     const double db = 3.1415;
        Fixed_Array< double, db > fa3;
```

16.3 类模板的成员函数

与非模板类一样，类模板的成员函数也可以在类模板的定义中定义，在这种情况下，该成员函数是 inline 成员函数。或者，成员函数也可以被定义在类模板定义之外。在介绍类模板 Queue 时，我们已经看到了 inline 成员函数的例子。例如，在类模板定义中的 Queue 构造函数被定义为 inline：

```
template <class Type>
class Queue {
    // ...
public:
    // inline 构造成员函数
    Queue() : front( 0 ), back ( 0 ) { }

    // ...
};
```

被定义在类模板定义之外的成员函数必须使用特殊的语法，来指明它是一个类模板的成员。成员函数定义的前面必须加上关键字 template 以及模板参数。例如，在类模板定义之外可定义 Queue 构造函数如下：

```
template <class Type>
class Queue {
public:
    Queue( );
private:
    // ...
};

template <class Type>
inline Queue<Type>::
    Queue( ) { front = back = 0; }
```

Queue 的第一次出现（在域操作符::之前）后面紧跟着模板参数表，这表示成员函数所属的类模板。Queue 的第二次出现（紧跟在域操作符之后）表示构造函数的名字。它的名字可以（但是不必）紧跟在模板参数表后面。在成员函数名字之后是函数定义，它看起来与非模

板函数定义十分相像。但是，凡是在普通函数定义中可以使用类型名的地方，类模板的成员函数定义也可以引用模板参数 Type。

类模板的成员函数本身也是一个模板。标准 C++要求这样的成员函数只有在被调用或者取地址时，才被实例化。（标准 C++之前的有些编译器在实例化类模板时，就实例化类模板的成员函数。）用来实例化成员函数的类型，就是其成员函数要调用的那个类对象的类型。例如：

```
Queue<string> qs;
```

对象 qs 的类型是 Queue<string>。当初始化这个类对象时，Queue<string>类的构造函数被调用。在这种情况下，用来实例化构造函数的模板实参是 string。

当类模板被实例化时，类模板的成员函数并不自动被实例化。只有当一个成员函数被程序用到（函数调用或取地址）时，它才被实例化。类模板的成员函数被实例化的时间会影响到“在类模板成员函数定义中名字的解析”（在 16.11 节进一步讨论），以及“可以声明一个成员函数特化的时间”（在 16.9 节进一步讨论）。

16.3.1 Queue 和 QueueItem 模板成员函数

为了更加熟悉类模板成员函数的定义和用法，让我们再详细地看一看类模板 Queue 及其成员函数：

```
template <class Type>
class Queue {
public:
    Queue() : front( 0 ), back ( 0 ) { }
    ~Queue();
    Type remove();
    void add( const Type & );
    bool is_empty() const {
        return front == 0 ;
    }
private:
    QueueItem<Type> *front;
    QueueItem<Type> *back;
};
```

析构函数以及成员函数 remove()和 add()都没有被定义在类模板定义中。如下面例子所示，这些成员函数在类模板定义之外被定义。Queue 析构函数清空队列中的项：

```
template <class Type>
Queue<Type>::~Queue()
{
    while ( ! is_empty() )
        remove();
}
```

成员函数 Queue<Type>::add()在队尾加入了一个新项，下面是实现过程：

```
template <class Type>
void Queue<Type>::add( const Type &val )
{
```

```
        // 分配新的 QueueItem 对象
        QueueItem<Type> *pt =
            new QueueItem<Type>( val );
        if ( is_empty() )
            front = back = pt;
        else
        {
            back->next = pt;
            back = pt;
        }
    }
```

成员函数 Queue<Type>::remove()返回在队首的项，且相关的 QueueItem 对象被删除：

```
    #include <iostream>
    #include <cstdlib>

    template <class Type>
    Type Queue<Type>::remove()
    {
        if ( is_empty() )
        {
            cerr << "remove() on empty queue \n";
            exit( - 1 );
        }
        QueueItem<Type> *pt = front;
        front = front->next;
        Type retval = pt->item;

        delete pt;
        return retval;
    }
```

我们决定在头文件 Queue 上中增加成员函数的定义，并将在每个使用这些成员函数实例的文件中包含这些定义（我们将在 16.8 节介绍这样做的原因，以及模板编译模式的更一般的问题）。

下列程序说明了怎样使用和实例化 Queue 类模板的成员函数：

```
    #include <iostream>
    #include "Queue.h"

    int main()
    {
        //  Queue<int> 类被实例化
        // new 表达式要求 Queue<int> 必须被定义
        Queue<int> *p_qi = new Queue<int>;
        int ival;
        for ( ival = 0; ival < 10; ++ival )
            // 成员函数 add() 被实例化
            p_qi->add( ival );

        int err_cnt = 0;
        for ( ival = 0; ival < 10; ++ival ) {
```

```
        // 成员函数 remove() 被实例化
        int qval = p_qi->remove();
        if ( ival != qval ) err_cnt++;
    }

    if ( !err_cnt )
        cout << "!! queue executed ok\n";
    else cerr << "?? queue errors: " << err_cnt << endl;
    return 0;
}
```

编译并执行程序，产生下列输出：

```
!! queue executed ok
```

练习 16.5

请用 16.2 节中定义的类模板 Screen，重新将“第 13 章中 13.3 节、13.4 节和 13.6 节中实现的 Screen 类的成员函数”实现为模板成员函数。

16.4 类模板中的友元声明

有三种友元声明可以出现在类模板中：

1．非模板友元类或友元函数。在下面的例子中，函数 foo()、成员函数 bar()以及 foobar 类都是类模板 QueueItem 的所有实例的友元。

```
class Foo {
    void bar();
};

template <class T>
class QueueItem {
    friend class foobar;
    friend void foo();
    friend void Foo::bar();
    // ...
};
```

在类模板 QueueItem 把 foobar 类和函数 foo()声明为友元之前，它们不必在全局域中被声明或定义。但是，在 QueueItem 类把 Foo 类的一个成员声明为友元之前，Foo 类必须已经被定义。记住，一个类成员只能由该类的定义引入。在 Foo 的类定义可见之前，QueueItem 不能引用 Foo::bar()。

2．绑定的（bound）友元类模板或函数模板。下列例子中，在类模板 QueueItem 的实例和它的友元（也是模板实例）之间定义了一对一的映射。对 QueueItem 的每一个类型的实例，foobar、foo()和 Queue<T>::bar()的单个相关的实例都是友元。如上所示：

```
template <class Type>
    class foobar{ ... };

template <class Type>
```

```
        void foo( QueueItem<Type> );

    template <class Type>
    class Queue {
        void bar();

        // ...
    };

    template <class Type>
    class QueueItem {
        friend class foobar<Type>;
        friend void foo<Type>( QueueItem<Type> );
        friend void Queue<Type>::bar();

        // ...
    };
```

在一个模板可以被用在一个类模板的友元声明中之前，它的声明或定义必须先被给出。在我们的例子中，在 QueueItem 类中的友元声明之前，必须先声明类模板 foobar 和 Queue，以及函数模板 foo()。

foo()的友元声明的语法看起来或许令人吃惊：

```
    friend void foo<Type>( QueueItem<Type> );
```

函数名后面紧跟着显式的模板实参表：foo<type>。这种语法可用来指定该友元声明所引用的函数模板 foo()的实例。如果省略了显式的模板实参，如下所示：

```
    friend void foo( QueueItem<Type> );
```

则友元声明会被解释为引用了一个非模板函数，且该函数的参数类型是类模板 QueueItem 的一个实例。正如在 10.6 节中提到的，模板函数和同名的非模板函数可以共存。虽然在 QueueItem 类的定义之前存在函数模板声明，但是这不会强迫友元声明指向该模板。所以，我们必须为“引用函数模板实例的友元声明”指定显式的模板参数表。

3．非绑定的（unbound）友元类模板或函数模板。在下面的例子中，在类模板 QueueItem 的实例和其友元之间定义了一对多的映射。对 QueueItem 的每一个类型的实例，foobar、foo()和 Queue<T>::bar()的所有实例都是友元。如下所示：

```
    template <class Type>
    class QueueItem {
        template <class T>
            friend class foobar;

        template <class T>
            friend void foo( QueueItem<T> );

        template <class T>
            friend void Queue<T>::bar();

        // ...
    };
```

我们应该注意，在标准 C++之前的编译器不支持类模板中的这种友元声明。

16.4.1 Queue 和 QueueItem 的友元声明

因为 QueueItem 不想被一般的程序使用，所以 QueueItem 构造函数的声明被移到类模板 QueueItem 的私有区内。现在，为了创建和操纵 QueueItem 类对象，Queue 必须被声明为 QueueItem 的友元。

有两种方法可以把一个类模板声明为友元。第一种方法是，将所有的 Queue 实例都声明为每个 QueueItem 实例的友元：

```
template <class Type>
class QueueItem {
    // 所有的 Queue 实例都是
    // 每个 QueueItem 实例的友元

    template <class T> friend class Queue;
};
```

但是，这不是真正的设计意图。例如，“用类型 string 实例化的 Queue”成为“用类型 complex<double>实例化的 QueueItem”的友元没有任何意义。Queue<string>只应该是 string 类型的 QueueItem 实例的友元。即，我们想要一个位于 Queue 和 QueueItem 的每个类型实例之间的一一映射。这可以通过第二种友元声明来获得：

```
template <class Type>
class QueueItem {
    // 每个 QueueItem 实例都有相关的
    // Queue 实例作为友元
    friend class Queue<Type>;

    // ...
};
```

该声明指定了对于 QueueItem 的每个特殊类型的实例，相应的 Queue 实例是友元。即，int 型的 Queue 实例只是 int 型的 QueueItem 实例的友元，而不是 complex<double>或 string 型的实例的友元。

在任何给定点上，用户可能需要能够显示 Queue 对象的内容。要做到这一点，一种方法是提供输出操作符的重载实例。这个操作符需要被声明为 Queue 类模板的友元函数，因为它必须访问该类的私有成员。操作符的原型应该是什么样呢？

```
// 什么形式的 Queue 实参?
ostream& operator<< ( ostream &, ??? );
```

既然 Queue 是一个类模板，那么，模板实例的名字就必须指定完整的实参表。例如：

```
ostream& operator<< ( ostream &, const Queue<int> & );
```

这定义了“专门针对 int 型数据项的 Queue 类模板实例”的输出操作符。但是，如果 Queue 的数据项是 string，又会怎么样呢？

```
ostream& operator<< ( ostream &, const Queue<string> & );
```

我们并不需要显式地定义每一个特殊的输出操作符，而是可以定义一个一般化的输出操作符，它能够处理 Queue 的全部实例。例如：

```
osostream& operator<< ( ostream &, const Queue<Type> & );
```

但是，为了使其能够工作，接下来我们必须使这个重载的输出操作符成为一个函数模板：

```
template <class Type> ostream&
    operator<< ( ostream &, const Queue<Type> & );
```

这样就可以了。每次将 Queue 实例传递给 ostream 时，函数模板就被实例化和调用一次。下面是作为函数模板的输出操作符的一种可能的实现：

```
template <class Type>
ostream& operator<< ( ostream &os, const Queue<Type> &q )
{
    os << "< ";
    QueueItem<Type> *p;

    for ( p = q.front; p; p = p->next )
        os << *p << " ";
    os << " >";

    return os;
}
```

如果 int 型对象的 Queue 含有值 3、5、8 和 13，则该 Queue 显示的输出如下：

```
< 3 5 8 13 >
```

注意，输出操作符引用了 Queue 的私有成员 front。下一件我们需要做的事情是把这个操作符声明为 Queue 的友元：

```
template <class Type>
class Queue {
    friend ostream&
        operator<< ( ostream &, const Queue<Type> & );

    // ...
};
```

我们注意到，在这种情况下，与本节中前面的类模板 Queue 中的友元声明一样，这种声明创建了 Queue 的实例与相应的 operator<<()实例之间的一一映射的关系。

Queue 中元素的实际显示输出依赖于 QueueItem 的输出操作符 operator<<()：

```
os << *p;
```

QueueItem 的输出操作符也要被实现为一个模板函数，从而保证在需要的时候自动实例化一个合适的 operator<<()：

```
template <class Type>
ostream& operator<< ( ostream &os, const QueueItem<Type> &qi )
{
    os << qi.item;
    return os;
}
```

因为这个操作符访问了 QueueItem 的私有成员 item，所以这个操作符必须被声明为类模板 QueueItem 的友元。实现如下：

```
template <class Type>
class QueueItem {
    friend class Queue<Type>;
```

```
        friend ostream&
            operator<< ( ostream &, const QueueItem<Type> & );
        // ...
};
```

QueueItem 的输出操作符 operator<<()依赖于 item 本身来处理实际的输出：

```
os << qi.item;
```

这给 Queue 实例引入了一种微妙的类型依赖。实例上，绑定在 Queue 上的每一个用户定义的类类型要想显示自己，就必须提供一个输出操作符。没有一种语言层次上的机制可以在类模板 Queue 的定义中指定或者强加这种依赖性。相反，如果“被用来实例化模板 Queue 的类型”没有定义相应的输出操作符，则试图显示该实例的内容时，输出操作符就会无效，从而导致一个编译时刻错误。当然，对于未定义输出操作符的类型，Queue 也可以被实例化，只不过不要企图显示 Queue 的内容。

下面的程序说明了怎样实例化和使用 Queue 类模板的友元函数和 QueueItem 类模板的友元函数：

```
#include <iostream>
#include "Queue.h"
int main() {
    Queue<int> qi;

    // 实例化两个实例：
    // ostream& operator<< (ostream &os, const Queue<int> &)
    // ostream& operator<< (ostream &os, const QueueItem<int> &)
    cout << qi << endl;
    int ival;
    for ( ival = 0; ival < 10; ++ival )
        qi.add( ival );
    cout << qi << endl;

    int err_cnt = 0;
    for ( ival = 0; ival < 10; ++ival ) {
        int qval = qi.remove();
        if ( ival != qval ) err_cnt++;
    }
    cout << qi << endl;

    if ( !err_cnt )
        cout << "!! queue executed ok\n";
    else cout << "<<<< queue errors: " << err_cnt << endl;

    return 0;
}
```

编译并运行程序，产生如下输出：

```
< >
< 0 1 2 3 4 5 6 7 8 9 >
< >
!! queue executed ok
```

练习 16.6

请使用练习 16.5 中定义的类模板 Screen，将原来为 Screen 类定义的输入和输出操作符（15.2 节的练习 15.6）重新实现为模板，并说明你选择加入到类模板 Screen 中的友元声明的原因。

16.5 类模板的静态数据成员

类模板也可以声明静态数据成员。类模板的每个实例都有自己的一组静态数据成员。为说明这一点，让我们为类模板 QueueItem 引入操作符 new()和 delete()。为此，我们需要为 QueueItem 类增加两个静态的数据成员：

```
static QueueItem<Type> *free_list;
static const unsigned QueueItem_chunk;
```

对类模板 QueueItem 定义的修改如下：

```
#include <cstddef>
template <class Type>
class QueueItem {
     // ...
private:
     void *operator new( size_t );
     void operator delete( void *, size_t );

     // ...
     static QueueItem *free_list;
     static const unsigned QueueItem_chunk;
     // ...
};
```

操作符 new()和 delete()被声明为 private，以此来防止普通的程序在自由存储区中创建 QueueItem 类型的对象。只有 QueueItem 的成员和友元（比如模板 Queue）才能够在自由存储区中创建（和删除）QueueItem 类型的对象。

操作符 new()的定义可以如下实现：

```
template <class Type> void*
QueueItem<Type>::operator new( size_t size )
{
     QueueItem<Type> *p;

     if ( ! free_list )
     {
          size_t chunk = QueueItem_chunk * size;
          free_list = p =
               reinterpret_cast< QueueItem<Type>* >
                    ( new char[chunk] );
          for ( ; p != &free_list[ QueueItem_chunk - 1 ]; ++p )
               p->next = p + 1;
          p->next = 0;
     }
```

```
        p = free_list;
        free_list = free_list->next;

        return p;
    }
```

下面是操作符 delete()的模板实现：

```
template <class Type>
void QueueItem<Type>::
        operator delete( void *p, size_t )
{
        static_cast< QueueItem<Type>* >( p )->next = free_list;
        free_list = static_cast< QueueItem<Type>* > ( p );
}
```

剩下要做的事情就是初始化静态成员 free_list 和 QueueItem_chunk。模板形式的静态数据成员定义如下所示：

```
/* 为每个 Queueltem 实例生成相关的 free_list,
 * 并把它初始化为 0
 */
template <class T>
        QueueItem<T> *QueueItem<T>::free_list = 0;

/* 为每个 Queueltem 实例生成相关的 QueueItem_chunk,
 * 并把它初始化为 24
 */
template <class T>
        const unsigned int
QueueItem<T>::QueueItem_chunk = 24;
```

静态数据成员的模板定义必须出现在类模板定义之外。因此，模板定义以关键字 template 开始，后面是类模板参数表<class T>。静态数据成员的名字前需要加上前缀 QueueItem<T>::，表明该成员属于类模板 QueueItem。这些静态数据成员的定义被加入到头文件 Queue.h 中，且必须被包含在使用这些静态数据成员的实例的文件中。（我们将在 16.8 节了解之所以认定这样做的原因，以及模板编译模式的相关话题。）

只有当程序使用静态数据成员时，它才会从模板定义中被真正实例化。类模板的静态成员本身就是一个模板，静态数据成员的模板定义不会引起任何内存被分配。只有对静态数据成员的某个特定的实例才会分配内存。每个静态数据成员实例都与一个类模板实例相对应。因此，一个静态数据成员的实例在被引用的时候，总是要通过一个特定的类模板实例。例如：

```
// 错误: QueueItem 不是一个真正的实例
int ival0 = QueueItem::QueueItem_chunk;

int ival1 = QueueItem<string>::QueueItem_chunk; // ok
int ival2 = QueueItem<int>::QueueItem_chunk; // ok
```

练习 16.7

请使用 15.8 节中定义的操作符 new()和 delete()，以及与它们相关的静态成员 screenChunk

和 freeStore，为练习 16.6 中定义的类模板 Screen 实现这些操作符和静态成员。

16.6 类模板的嵌套类型

类模板 QueueItem 只被设计用来辅助 Queue 的实现。为了达到这个目的，QueueItem 向一个私有构造函数，它允许其友元类模板 Queue 的成员函数，而不允许其他类或函数（除了它自己的成员函数）创建 QueueItem 类型的对象。虽然 QueueItem 是一个对整个程序都可见的类模板，但是程序不调用 Queue 的成员函数就不能创建 QueueItem 对象或引用 QueueItem 的任何成员。

另外一种策略是，在类模板 Queue 的私有区中嵌入类模板 QueueItem 的定义。既然 QueueItem 成为嵌套私有类型，它对一般程序就变成不可见的。因为它是一个私有的嵌套类型，所以只有类模板 Queue 和 Queue 的友元（输出操作符）才可以访问它。如果我们让 QueueItem 的成员都成为公有的。则没有必要再把 Queue 声明为 QueueItem 的友元。

这种实现保留了原始实现的语义，并已更加优雅地建立了 QueueItem 与 Queue 类模板之间的关系模型。

因为 Queue 要求每个被实例化的类型都有相关的 QueueItem 类，所以嵌套的类也是一个类模板。类模板的嵌套类自动成为一个类模板，在嵌套类模板内部可以使用外围类模板的模板参数。例如：

```
template <class Type>
class Queue {
    // ...
private:
    class QueueItem {
    public:
        QueueItem( Type val )
            : item( val ), next( 0 ) { ... }
        Type item;
        QueueItem *next;
    };

    // 因为 QueueItem 是一个嵌套类型
    // 不是在 Queue 外定义的模板
    // 所以可以省略 QueueItem 之后的模板实参 <Type>
    QueueItem *front, *back;
    // ...
};
```

Queue 的每个实例都用适当的 Type 模板实参生成自己的 QueueItem 类。这种在类模板 QueueItem 的实例和外围类模板 Queue 的实例之间的映射是一对一的。

当外围类模板被实例化时，它的嵌套类不会自动被实例化。只有当上下文环境确实需要嵌套类的完整类类型时，嵌合类才会被实例化。例如，我们在 16.2 节中提到，如果类模板 Queue 针对 int 类型被实例化了，则类型 QueueItem<int>不会被自动实例化。成员 front 和 back 是指向 QueueItem<int>的指针。如果只是声明了这种类类型的指针，则不需要实例化类型 QueueItem<int>。将 QueueItem 做成类模板 Queue 的一个嵌套类，并不会改变这一点。只有

当 Queue<int>的成员函数解引用（dereference）成员 front 和 back 时，QueueItem<int>才会被实例化。

在类模板中也可以声明枚举和 typedef。例如：

```
template <class Type, int size>
class Buffer {
public:
    enum Buf_vals { last = size -1, Buf_size };
    typedef Type BufType;
    BufType array[ size ];

    // ...
};
```

我们并没有提供一个显式的 Buf_size 数据成员，而是 Buffer 类模板声明了一个枚举类型，并且用模板参数来初始化其中的嵌套枚举值。例如，声明：

```
Buffer<int, 512> small_buf;
```

把 Buf_size 设置为 512，而 last 为 511。类似地，声明：

```
Buffer<int, 1024> medium_buf;
```

把 Buf_size 设置为 1024，而 last 为 1023。

公有的嵌套类型可以被用在类定义之外。然而，对于类模板的公有嵌套类型（或嵌套枚举类型的一个枚举值）。一般的程序只能引用该嵌套类型的一个实例。在这种情况上，嵌套类型的名字前必须要加上类模板实例的名字。例如：

```
// 错误: Buffer 的哪一个实例?
Buffer::Buf_vals bfv0;
Buffer<int,512>::Buf_vals bfv1;      // ok
```

即使嵌套类型并没有使用外围类模板的参数，这条规则也同样适用。例如：

```
template <class T> class Q {
public:
    enum QA { empty, full };         // 不变量
    QA status;

    // ...
};
#include <iostream>
int main() {
    Q<double> qd;
    Q<int> qi;

    qd.status = Q::empty;            // 错误: Q 的哪一个实例?
    qd.status = Q<double>::empty;    // ok

    int val1 = Q<double>::empty;
    int val2 = Q<int>::empty;

    if ( val1 != val2 )
        cerr << "implementation error!" << endl;
    return 0;
```

```
}
```

虽然在 Q 的每一个实例中，empty 的值都相同，但是，引用 empty 的代码也必须指定这个枚举值所属的 Q 的实例。

练习 16.8

请把 13.10 节定义的 List 和其嵌套的类 ListItem 定义为类模吧　并为相关的类成员提供模板定义。

16.7 成员模板　※

函数或类模板可以是一个普通类的成员，也可以是一个类模板的成员。成员模板的定义看起来像一般模板的定义　成员定义前面加上关键字 template 及模板参数表。例如：

```
template <class T>
class Queue {
private:
     // 类成员模板
     template <class Type>
     class CL
     {
          Type member;
          T mem;
     };
     // ...
public:
     // 函数成员模板
     template <class Iter>
     void assign( Iter first, Iter last )
     {
          while ( ! is_empty() )
               remove();   // calls Queue<T>::remove()
          for ( ; first != last; ++first )
               add( *first ); // calls Queue<T>::add( const T & )
     }
};
```

（注意，标准 C++之前的编译器不支持成员模板。这种特性被加入到 C++中，是为了支持第 6 章中给出的抽象容器类型的实现，正如下面的段落所解释的。）

成员模板的声明有它自己的模板参数。例如，类成员模板 CL 有自己的名为 Type 的模板参数，而函数成员模板 assign()也有自己的模板参数 Iter。另外，成员模板的定义也可以使用外围类模板的模板参数。例如，类成员模板 CL 有类型为 T 的数据成员，而 T 是外围类模板 Queue 的模板参数。

在类模板 Queue 中声明一个成员模板意味着，Queue 的一个实例包含了“可能无限多个联套类 CL”，和“可能无限多个成员函数 assign()”。例如，Queue<int>的实例可能含有下列嵌套类型：

```
Queue<int>::CL<char>
```

```
Queue<int>::CL<string>
```

类似地，Queue<int>也可能会有下列成员函数：

```
void Queue<int>::assign( int *, int * )
void Queue<int>::assign( vector<int>::iterator,
                         vector<int>::iterator )
```

成员模板遵循与其他类成员相同的访问规则。因为类成员模板 CL 是类模板 Queue 的一个私有成员，所有只有 Queue 的成员函数和友元（friend）才能引用这个类成员模板的实例。而函数成员模板 assign()是一个公有成员，所以它可以被整个程序使用。

只有当成员模板被程序中使用时，它才被实例化。例如：

```
int main()
{
    // Queue<int> 的实例
    Queue<int> qi;

    // Queue<int>::assign( int *, int * ) 的实例
    int ai[4] = { 0, 3, 6, 9 };
    qi.assign( ai, ai + 4 );

    // Queue<int>::assign( vector<int>::iterator,
    //                     vector<int>::iterator) 的实例
    vector<int> vi( ai, ai + 4 );

    qi.assign( vi.begin(), vi.end() );
}
```

类模板 Queue 的函数成员模板 assign()是说明“为什么支持容器类型需要成员模板”的一个很好的例子。例如，给定一个 Queue<int>类型的队列，我们希望能够把其他容器（list、vector 或简单数组）的内容加入到队列中，这些容器的元素要么是 int 型，要么是其他可以转换成 int 的类型。成员模板 assign()正好允许我们这样做。因为它可以使用任何容器类型，所以我们编写函数成员模板 assign()时，用迭代器作为接口，这样就能够把它的实现与迭代器指向的实际类型分离开。

在函数 main()中，首先用类型 int*实例化成员模板 assign()，以允许把 int 数组的内容赋值给 qi。然后再用类型 vector<int>::iterator 实例化成员模吧　以允许把 int 型 vector 的内容赋值给 qi。这些可以把元素赋给队列的容器，而不一定含有 int 型的元素，凡是可被转换成 int 的类型都有效。为了说明原因，我们来看一下 assign()的定义

```
template <class Iter>
void assign( Iter first, Iter last )
{
    // 从队列中删除项
    for ( ; first != last; ++first )
        add( *first );
}
```

由 assign()调用的函数 add()是成员函数 Queue<Type>::add()。在 int 型的 Queue 实例中。该成员函数有下列原型：

```
void Queue<int>::add( const int &val )
```

实参*first 必须是 int 型，或者是一个可以初始化 const int 类型的引用参数的类型。类型转换也是允许的。例如，为了重用 15.9 节定义的 SmallInt 类，我们可以通过函数成员模板 Assign()，把“元素类型为 SmallInt 的容器”的内容赋给 Queue<int>类型的队列。这也是可以的，因为类 SmallInt 含有转换函数，可以把 SmallInt 类型的值转换成 int 型的值：

```
class SmallInt {
public:
    SmallInt( int ival = 0 ) : value( ival ) { }

    // 转换函数: SmallInt ==> int
    operator int() { return value; }

    // ...
private:
    int value;
};

int main()
{
    // Queue<int> 的实例
    Queue<int> qi;
    vector<SmallInt> vsi;

    // 设置 vector 的内容
    // Queue<int>::assign( vector<SmallInt>::iterator,
    //                     vector<SmallInt>::iterator ) 的实例
    qi.assign( vsi.begin(), vsi.end() );
    list<int*> lpi;

    // 设置 list 的内容
    // 错误: 当成员模板 assign() 被实例化时
    // 从 int* 到 int 不存在转换
    qi.assign( lpi.begin(), lpi.end() );
}
```

assign()的第一次实例化是有效的，因为存在从 SmallInt 到 int 型的隐式转换，且 assign()的第一次实例化中的 add()调用是有效的。第二个实例化是错误的，因为 int*型的对象不能初始化一个指向 const int 类型的引用。在 assign()的第二个实例中的 add()调用是错误的。

C++标准库中定义的容器类型有一个被称为 assign()的函数成员模吧　它的行为与我们的类模板 Queue 的成员模板 assign()完全一样。

任何成员函数都可以被定义为成员模板。例如，构造函数也可以被定义为成员模板。我们可以为类模板 Queue 定义这样的构造函数：

```
template <class T>
class Queue {
    // ...
public:
    // 构造函数成员模板
    template <class Iter>
```

```
        Queue( Iter first, Iter last )
        : front( 0 ), back( 0 )
        {
            for ( ; first != last; ++first )
                add( *first );
        }
};
```

这样的构造函数允许用另一个容器的内容进行初始化。C++标准库中定义的容器类型也有构造函数成员模吧　以便允许用其他容器类型的内容进行初始化。实际上，本节中的第一个 main()定义就使用了 vector 的构造函数成员模吧

```
vector<int> vi( ai, ai + 4 );
```

这个定义用类型 int*实例化了容器 vector<int>的构造函数成员模吧　以便允许用 int 型元素数组的内容来初始化这个 vector。

像非模板成员一样，一个成员模板也可以被定义在其外围类或类模板定义之外。例如，类成员模板 CL 或者成员函数模板 assign()可以被定义在类模板 Queue 之外，如下所示：

```
template <class T>
class Queue {
private:
    template <class Type> class CL;
    // ...
public:
    template <class Iter>
    void assign( Iter first, Iter last );
    // ...
};

template <class T> template <class Type>
class Queue<T>::CL<Type>
{
    Type member;
    T mem;
};

template <class T> template <class Iter>
void Queue<T>::assign( Iter first, Iter last )
{
    while ( ! is_empty() )
        remove();

    for ( ; first != last; ++first )
        add( *first );
}
```

如果一个成员模板被定义在类模板定义之外，则在它的定义前面就必须加上类模板参数表，然后再跟上它自己的模板参数表。这就是成员函数模板 assign()以：

```
template <class T> template <class Iter>
```

开头的原因。

第一个模板参数表 template<class T>是类模板 Queue 的，而第二个模板参数表

template<class Iter>是成员模板 assign()的。模板参数不一定与类模板定义中指定的名字相同。例如，下面的语句仍然定义了类模板 Queue 的函数成员模板 assign()：

```
template <class TT> template <class IterType>
    void Queue<TT>::assign( IterType first, IterType last )
    { ... }
```

16.8 类模板和编译模式 ※

类模板定义只是“无限多个类类型的定义”的“规范描述（prescription）”而已，模板定义本身并没有定义任何一个类类型。例如，当编译器看到如下类模板定义时：

```
template <class Type>
    class Queue { ... };
```

它仅保存 Queue 的内部表示。而以后当编译器看到这种类模板实例真正被使用时，如：

```
int main() {
    Queue<int> *p_qi = new Queue<int>;
}
```

它就用保存下来的 Queue 模板定义的内部表示来实例化类类型 Queue<int>。

只有当上下文环境要求类模板的完整类定义时，类模板才被实例化（这在 16.2 节更详细地讨论过）。在前面的例子中，类模板实例 Queue<int>被实例化，因为编译器必须知道类类型 Queue<int>的大小，以便为 new 表达式创建的对象分配正确的存储区。

只有当编译器看到了实际的类模板定义　而不仅仅只是声明时，它才能实例化类模板。当程序使用一个类模板并且要求其实例时，程序必须首先提供类模板的定义

```
// 类模板的声明
template <class Type>
class Queue;

Queue<int>* global_pi = 0;   // ok: 不需要类定义

int main() {
    // 错误：需要实例
    // 类模板定义必须可见
    Queue<int> *p_qi = new Queue<int>;
}
```

类模板可以在多个文件中针对同一类型被实例化。我们知道，对于类类型的情况，在每个使用类成员的文件中必须提供类定义。同样地，在要求“类模板实例的完整类定义”的每个文件中，编译器针对特定的类型实例化该类模板。为了确保在每个必须实例化类模板的文件中都有类模板的定义　类模板定义应该被放在头文件中。

类模板的成员函数、静态数据成员和嵌套类的行为与模板本身十分相像。类模板的成员的定义被用来为每个特定类的模板实例生成成员实例。例如，当编译器看到如下成员函数定义时：

```
template <class Type>
void Queue<Type>::add( const Type &val )
    { ... }
```

它就保存 Queue<Type>::add()的内部表示。以后，当编译器看到这个成员函数被真正使用时（例如，通过一个 Queue<int>类型的对象），它才根据保存下来的成员函数定义的内部表示，实例化 Queue<int>::add(const int&)：

```
#include "Queue.h"
int main() {
    // Queue<int> 的实例
    Queue<int> *p_qi = new Queue<int>;
    int ival;
    // ...
    // Queue<int>::add( const int & ) 的实例
    p_qi->add( ival );
    // ...
}
```

针对一个特定类型而实例化类模吧　不会引起“针对同一类型自动实例化类模板成员的定义”。只有当程序需要知道成员的定义时（即，如果嵌套类被使用时要求它的完整类类型，或如果调用成员函数，或如果做成员地址，或如果查看静态数据成员的值），成员才会被实例化。

类模板的静态数据成员的实例化会带来我们在 10.5 节中关于函数模板讨论的同样问题要使编译器能够实例化一个类模板的成员函数或者静态成员，在使用成员的一个实例时，成员的定义必须可见吗？例如，在 main()中成员函数 add()的整型实例被调用之前，它的定义必须出现吗？我们把成员函数和静态数据成员的定义放在头文件中（正如我们对于内联成员函数定义所做的那样），以便使它们被包含在每个使用其实例的地方，或者类模板定义的实例已经足够允许这些成员被使用了，所以可把这些成员定义放在文本文件中（我们通常把类类型的非内联成员函数和静态数据成员的定义放在哪里）？

要回答这些问题，我们必须回顾 **C++的模板编译模式**（template compilation model），它指定了那些定义和使用模板的程序应该如何组织代码。在 10.5 节我们描述的两种模板编译模式——包含模式和分离模式——都适用于类模板的成员函数和静态数据成员的定义。本节余下部分将描述这两种模式，以及它们被如何用于这些成员定义。

16.8.1 包含编译模式

在包含编译模式下，类模板的成员函数和静态成员的定义必须被包含在“要将它们实例化”的所有文件中，对于类模板定义中被定义为 inline 的内联成员函数，这是自动发生的。但是，如果一个成员函数被定义在类模板定义之外，那么这些定义应该被放在含有该类模板定义的头文件中，这是本书选择的模式。例如，模板 Queue 和 QueueItem 的定义　以及它们的成员函数和静态数据成员的模板定义都被放在头文件 Queue.h 中。

与函数模板定义的情形一样，在头文件中提供类模板的成员定义有一些缺点。成员函数定义可能很大，可能会描述实现细节，而用户可能不想知道或者我们希望向用户隐藏这些细节。而且在多个文件之间编译相同的函数模板定义可能会增加不必要的编译时间。如果提供了分离编译模式，则允许我们把类模板接口（即类模板定义）与它的实现（即它的成员函数和静态数据成员的定义）分离。让我们来看看怎样使用它。

16.8.2 分离编译模式

在分离编译模式下，类模板定义和其 inline 成员函数定义都被放在头文件中，而非 inline 成员函数和静态数据成员被放在程序文本文件中。在这种模式下，类模板及其成员的定义的组织方式，与我们组织非模板类及其成员的定义的方式相同。例如：

```
// ----- Queue.h -----
// 声明 Queue 是一个可导出的 (exported) 类模板
export template <class Type>
class Queue {
     // ...
public:
     Type& remove();
     void add( const Type & );
     // ....
};

// ----- Queue.C -----
// exported definition of class template Queue in Queue.h
#include "Queue.h"

template <class Type>
void Queue<Type>::add( const Type &val ) { ... }
template <class Type>
Type& Queue<Type>::remove() { ... }
```

使用成员函数实例的程序只需要在使用它之前包含这个头文件：

```
// ----- User.C -----
#include "Queue.h"
int main() {
     // Queue<int>  的实例
     Queue<int> *p_qi = new Queue<int>;
     int ival;
     // ...

     // ok: Queue<int>::add( const int & )  的实例
     p_qi->add( ival );
     // ...
}
```

即使成员函数 add()的模板定义在 User.C 中不可见，但是，在这个文件中仍然可以调用模板实例 Queue<int>::add(const int&)。然而，为了使其成为可能，类模板必须以一种特殊的方式来声明——声明为 exported（可导出的）类模板。

可导出的类模板是指这样一个模吧　当它的成员函数实例或静态数据成员实例被使用时，编译器只要求类模板的定义。如果一个文件要使用这些成员，它可以省略这些成员的定义。

可导出的类模板的声明是在类模板的定义　或者声明）的关键字 template 之前加上关键字 export：

```
export template <class Type>
```

```
class Queue { ... };
```

在我们的例子中，关键字 export 被应用在文件 Queue.h 中的类模板 Queue 上，这个头文件被包含在 Queue.C 中，而 Queue.C 包含了类模板成员函数的定义。然后，成员函数 add()和 remove()的定义被自动声明为可导出的（exported）。在其他文件使用这些成员函数的实例之前，这些成员的定义可以不出现。

注意，即使一个类模板被声明为可导出的，类模板自身的定义也不能从 User.C 中省略掉。在 User.C 中的 Queue<int>类的实例提供了“声明成员函数 Queue<int>::add()和 Queue<int>::remove()”的类定义 在可以调用这些成员函数之前，这些声明是必需的。因此即使类模板自身被声明为可导出的，关键字 export 也只影响类模板的成员函数和静态数据成员。

我们也可以只把类模板的个别成员声明为可导出的。在这种情况上，关键字 export 不是被指定在类模板上，而是被指定在要被导出的成员定义上、例如，如果类模板 Queue 的作者只想让成员函数 Queue<type>::add()被导出（即，只想从头文件 Queue.h 中去掉这个成员函数的定义 ，则关键字 export 可以被指定在成员函数 add()的定义上：

```
// ----- Queue.h -----
template <class Type>
class Queue {
     // ...
public:
     Type& remove();
     void add( const Type & );
};

// 必需的，因为 remove() 不是可导出的
template <class Type>
Type& Queue<Type>::remove() { ... }

// ----- Queue.C -----
#include "Queue.h"
// 只有成员函数 add() 是可导出的
export template <class Type>
void Queue<Type>::add( const Type &val ) { ... }
```

注意，成员函数 remove()的模板定义被移到头文件 Queue.h 中是必须的，因为 remove()不再是一个可导出的模吧 因此它的定义在调用 remove()实例的文件中必须可见。

“类模板成员函数或静态数据成员的定义在一个程序中被定义为可导出的”只能有一次。不幸的是，因为编译器每次只处理一个文件，所以当这些成员在多个程序文本文件中作为可导出成员被定义时，它不能检测出来。如果有这种情况发生，可能发生以下这些行为：

1.可能产生一个链接时刻错误，指出为一个类模板的同一个成员提供了多个模板定义。

2.编译器可能为同一组模板实参多次实例化该成员，引起链接时刻错误，因为模板实例的重复定义。

3.编译器可能用一个可导出的模板定义实例化该成员，而忽略其他定义。

所以，我们不能确定，如果在程序中为一个类模板的可导出成员提供了多个定义 是否会产生一个错误。因此，在组织程序时我们必须小心谨慎地把这些成员定义只放在一个程序

文本文件中。

分离编译模式使我们能够更好地把类模板的接口同其实现分离开，它使我们能够这样来组织程序：把类模板的接口放在头文件中，而把具体实现放在文本文件中。但是，不是所有的编译器都支持分离模式，或者虽然支持但支持得不是很好。支持分离模式需要更复杂的程序设计环境，而这些不是在所有的 C++编译器实现中都可以获得的。

对于本书的目的而言，因为我们的模板例子都很小，而且我们想让这些例子能在更多的 C++编译器中易于编译，所以我们只限于使用包含编译模式。

16.8.3 显式实例声明

当使用包含编译模式时，类模板成员的定义被包含在使用其实例的所有程序文本文件中。何时何地编译器实例化类模板成员的定义 我们并不能精确地知晓。一些编泽器（尤其是比较老的 C++编译器）实际上可能针对同一组特定的模板实参，多次实例化成员定义 然后选择其中之一作为程序使用的实例（当链接程序时或在某一个预链接阶段），而其他实例只是被简单地忽略。

一个成员被实例化一次还是多次，都不会影响程序的结果，因为最后只有一个模板实例被程序使用。但是如果模板被多次实例化，则程序的编译时间性能可能会大大地受影响。如果应用程序由大量文件构成，并且一个模板在所有文件中都被实例化，则编译应用程序的时间可能会显著地增加。

早期编译器的实例化问题使得模板难于被使用。为了帮助解决这个问题，标准 C++提供了**显式实例声明**（explicit instantiation declaration），以允许程序员控制模板实例化发生的时间。

在显式实例声明中，关键字 template 后面跟着关键字 class 以及类模板实例的名字。下面的例子声明了 Queue<int>类的显式实例。这个显式实例声明要求用模板实参 int 来实例化类模板 Queue：

```
#include Queue.h

// 显式实例声明
template class Queue<int>;
```

显式实例化类模板时，它的所有成员也被显式实例化，而且针对同一组模板实参类型。这暗示着。在显式实例声明出现的地方不但要提供类模板的定义 而且还要提供类模板成员的全部定义。如果不存在这些定义 则显式实例声明是错误的。例如：

```
template <class Type>
class Queue;

// 错误：没有定义模板 Queue 及其成员
template class Que ue<int>;
```

当一个显式实例声明出现在程序文本文件中时，如果其他文本文件也使用了该类模板实例，则会发生什么？我们怎样告诉编译器一个显式实例声明出现在另外一个程序文本文件中，该类模板及其成员在程序的其他文本文件中被使用时不能再被实例化？

这种情形的解决方案与 10.5.3 节讨论函数模板时给出的方案相同，我们必须使用“抑制

模板隐式实例化”的编译器选项。当用这个选项编译我们的应用程序时。编译器假设我们将用显式实例声明来处理模板实例化，对于应用程序中用到的模吧 它将不会隐式实例化它们。

练习 16.9

如果你所使用的编译器支持分离编译模式，你会把类模板的成员函数和静态数据成员定义放在什么地方？请说明原因。

练习 16.10

已知在上节练习中开发的类模板 Screen（尤其是在 16.3 节的练习 16.5 中定义的成员函数，以及在 16.5 节的练习 16.7 中定义的静态成员），请利用模板的分离编译模式组织这些定义。

16.9 类模板特化 ※

在介绍类模板特化以及程序怎样定义它们之前，让我们为类模板 Queue 增加两个新的成员函数。成员函数 min()和 max()分别对 Queue 中的数据项进行迭代，以找到最小值和最大值（当然，最好是用第 12 章给出的泛型算法 min()和 max()。但是，为了介绍模板特化，我们把这些函数定义为类模板 Queue 的成员函数）：

```
template <class Type>
class Queue {
     // ...
public:
     Type min();
     Type max();
     // ...
};

// 找到 Queue 中的最小值
template <class Type>
Type Queue<Type>::min( )
{
     assert( ! is_empty() );
     Type min_val = front->item;
     for ( QueueItem *pq = front->next; pq != 0; pq = pq->next )
          if ( pq->item < min_val )
               min_val = pq->item;
     return min_val;
}
// 找到 Queue 中的最大值
template <class Type>
Type Queue<Type>::max( )
{
     assert( ! is_empty() );
     Type max_val = front ->item;
          for ( QueueItem *pq = front->next; pq != 0; pq = pq->next )
```

```
            if ( pq->item > max_val )
                max_val = pq->item;
        return max_val;
    }
```

在成员函数 min()中的下列语句用于比较 Queue 中的两个数据项：

```
pq->item< min_val
```

这引入了对于“Queue 类模板被实例化所针对的类型”的隐含要求：被用作模板实参的类型必须能够使用为内置类型而预定义的小于操作符，或者是定义了 operator<()的用户定义的类类型。如果没有为这样的类型定义 operator<()，而且在该类型数据项的 Queue 上调用了 min()，那么在 min()中使用无效的比较操作符的地方会导致一个编译时刻错误。（类似的问题也存在于成员函数 max()中，它使用了 operator()>。）

假设有如下类型，我们想用它实例化类模板 Queue：

```
class LongDouble {
public:
    LongDouble( double dval ) : value( dval ) { }
    bool compareLess( const Lo ngDouble & );
private:
    double value;
};
```

但是，因为不存在比较两个 LongDouble 型值的 operator<()，所以成员函数 min()和 max()不能被用在 Queue<LongDouble>类型的 Queue 上。这个问题的一种解决方案是，定义全局操作符 operator<()和 operator>()，它们使用 LongDouble 的成员函数 compareLess()来比较两个 Queue<LongDouble>类型的值。然后，再在 min()和 max()中，这些全局操作符被自动调用来比较 Queue<LongDouble>类型的 Queue 中的数据项。但是，为了介绍类模板特化，我们考虑另外一种方案。如果模板实参是 LongDouble 类型，则我们不希望使用类模板 Queue 的通用成员函数定义 来实例化成员函数 mim()和 max()。我们希望专门定义 Queue<LongDouble>::mln()和 Queue<LongDouble>::max()实例，让它们使用 Long(double)成员函数 compareLess()。

为此，我们可以通过一个**显式特化定义**（explicit specialization definition），为类模板实例的一个成员提供一个特化定义。显式特化定义包括关键字 template、后跟一对尖括号（<>，一个小于号和一个大于号），以及后面的类成员的特化定义。下面的例子为类模板实例 Queue<LongDouble>的成员函数 min()和 max()定义了显式特化。

```
// 显式特化定义
// explicit specialization definitions
template<> LongDouble Queue<LongDouble>::min( )
{
    assert( ! is_empty() );
    LongDouble min_val = front->item;
    for ( QueueItem *pq = front->next; pq != 0; pq = pq->next )
        if ( pq->item.compareLess( min_val ) )
            min_val = pq->item;
    return min_val;
}
template<> LongDouble Queue<LongDouble>::max( )
```

```
{
    assert( ! is_empty() );
    LongDouble max_val = front->item;
    for ( QueueItem *pq = front->next; pq != 0; pq = pq->next )
        if ( max_val.compareLess( pq->item ) )
            max_val = pq->item;
    return max_val;
}
```

即使类类型 Queue<LongDouble>是根据通用类模板定义而被实例化的，Queue<LongDouble>的每个对象仍可以使用成员函数 min()和 max()的特化——这些成员函数并不根据类模板 Queue 的通用成员定义而被实例化。

因为成员函数 min()和 max()的显式特化定义是函数定义而不是模板定义 而且因为这些定义没有被声明为内联的），所以它们不能被放在头文件中，必须被放在程序文本文件中。幸运的是，我们可以只是声明函数模板显式特化而不定义它。例如，成员函数 min()和 max()的显式特化可以被声明如下：

```
// 函数模板显式特化声明
template<> LongDouble Queue<LongDouble>::min( );
template<> LongDouble Queue<LongDouble>::max( );
```

把这些声明放在头文件中，以及把相关的定义放在程序文本文件中，我们就可以像对其他非模板类成员定义一样地组织显式特化的代码。

在某些情况下，整个类模板的定义对于某个特殊的类型并不合适。在这样的情况下，程序员可以提供一个定义来特殊化整个类模板。例如，程序员可以针对 Queue<LongDouble>提供一个完整的定义

```
// QueueLD.h: 定义类的特化 Queue<LongDouble>
#include "Queue.h"

template<> class Queue<LongDouble> {
    Queue<LongDouble>();
    ~Queue<LongDouble>();
    LongDouble& remove();
    void add( const LongDouble & );
    bool is_empty() const;
    LongDouble min();
    LongDouble max();
private:
    // 某些特殊的实现
};
```

只有当通用的类模板被声明（不一定被定义）之后，它的显式特化才可以被定义。即，在模板被特化之前，编译器必须知道类模板的名字。在前面的例子中，如果不在模板显式特化的定义之前包含头文件 Queue.h，则编译器就会产生一个错误，指出 Queue 不是一个模板名。

即使我们定义了一个类模板特化，也必须定义与这个特化相关的所有成员函数或静态数据成员。类模板的通用成员定义不会被用来创建显式特化的成员的定义，这是因为类模板特化可能拥有与通用模板完全不同的成员集合。如果我们决定为类类型 Queue<LongDouble>提

供一个显式特化定义 那么我们不但要为成员函数 min()和 max()提供定义 而且还必须为所有其他成员函数提供定义。

如果整个类被特化了，那么，标记特化定义的符号 template<>只能被放在类模板的显式特化的定义之前，类模板特化的成员定义不能以符号 template<>作为打头。例如：

```
#include "QueueLD.h"

// 定义类模板特化的成员函数 min()
LongDouble Queue<LongDouble>::min( ) { }
```

类模板不能够在某些文件中根据通用模板定义被实例化，而在其他文件中却针对同一组模报实参被特化。例如，给出了模板 Queue<LongDouble>的特化，则必须在使用它的每个文件中都声明该特化：

```
// ---- File1.C ----
#include "Queue.h"
void ReadIn( Queue<LongDouble> *pq ) {
     // pq->add() 的使用引起 Queue<LongDouble> 被实例化
}

// ---- File2.C ----
#include "QueueLD.h"

void ReadIn( Queue<LongDouble> * );
int main() {
     // 使用 Queue<LongDouble> 特化定义
     Queue<LongDouble> *qld = new Queue<LongDouble>;
     ReadIn( qld );

     // ...
}
```

上面的程序是错的。荆菅编译器通常诊断不出这样的错误，但为防止此类错误，我们应该在每个使用 Queue<LongDouble>的文件中，在它被第一次使用之前包含头文件 QueueLD.h。

16.10 类模板部分特化 ※

如果类模板有一个以上的模板参数，则有些人就可能希望为一个特定的模板实参或者一组模板实参特化类模吧 而不是为所有的模板参数特化该类模板。即，有人可能希望提供这样一个模吧 它仍然是一个通用的模吧 只不过某些模板参数已经被实际的类型或值取代。通过使用类模板部分特化（partial specialization），这是有可能实现的。相比“通用模板定义针对一组特定的模板实参被实例化之后的类版本”而言，类模板的部分特化可能被用来定义一个更加适当、更加高效的实现版本。

例如，让我们使用 16.2 节介绍的类模板 Screen。它的部分特化 Screen()<hi,80>为 80 列的屏幕提供了更加有效的实现：

```
template <int hi, int wid>
```

```
class Screen {
    // ...
};

// 类模板 Screen 的部分特化
template <int hi>
class Screen<hi, 80> {
public:
    Screen();
    // ...
private:
    string _screen;
    string::size_type _cursor;
    short _height;
    // 为 80 列的屏幕使用特殊的算法
};
```

类模板部分特化也是一个模吧　它的定义看起来就像一个模板定义。这样的定义以关键字 template 开始，后面是尖括号中的模板参数表。类模板部分特化的参数表与对应的通用类模板定义的参数表不同。Screen 的部分特化只有一个非类型模板参数 hi，因为 wid 的模板实参已知为 80。而部分特化的模板参数表只列出模板实参仍然未知的那些参数。

部分特化与对应的通用模板同名，也叫 Screen。但是，类模板部分特化的名字后面总是跟着一个模板实参表。在上一个例子中，模板实参表是<hi,80>。因为第一个模板参数的实参值未知，所以在实参表中，模板参数 hi 的名字被用作占位符。而另一个实参是一个值 80，该模板是针对这个值而被部分特化的。

当程序使用类模板部分特化时，它是被隐式实例化的。在下面的例子中，类模板部分特化将用 24 作为 hi 的模板实参而被实例化：

```
Screen<24,80> hp2621;
```

我们注意到，Screen<24,80>的实例既能从通用类模板定义而被实例化，也能从部分特化的定义而被实例化。那么编译器为什么会选择部分特化来实例化模板呢？当程序声明了类模板部分特化时，编译器选择“针对该实例而言最为特化的模板定义”进行实例化。当没有特化可被使用时，才使用通用模板定义。例如，当 Screen<40,132>必须被实例化时，该实例与程序提供的部分特化并不匹配，这里的部分特化只被用来实例化 80 列的 Screen。

部分特化的定义与通用模板的定义完全无关。部分特化可能拥有与通用类模板完全不同的成员集合。类模板部分特化必须有它自己对成员函数、静态数据成员和嵌套类的定义。类模板成员的通用定义不能被用来实例化类模板部分特化的成员。例如，我们必须定义部分特化 Screen<hi,80>的构造函数。下面是一种可能的定义

```
// 部分特化 Screen<hi, 80> 的构造函数
template<int hi>
Screen<hi,80>::Screen() : _height( hi ), _cursor ( 0 ),
          _screen( hi * 80, bk )
              { }
```

如果程序没有提供 Screen<hi,80>的构造函数模板定义　而该部分特化又被用来实例化一个类类型，则通用类模板的构造函数定义不会被用来实例化这个构造函数成员。

16.11 类模板中的名字解析 ※

在 10.9 节关于函数模板的名字解析的讨论中，我们说过，这个解析过程分两步进行。对于类模板定义及其成员定义中的名字解析，这两步仍然适用。每一个步骤分别应用在不同种类的名字上：第一步应用于“在类模板的所有实例中具有相同意义”的名字，第二步应用于“在不同模板实例中意义不同”的名字。让我们来看一些例子，它们使用了类模板 Queue 的成员函数 remove()：

```
// Queue.h:
#include <iostream>
#include <cstdlib>

// Queue 类的定义
template <class Type>
Type Queue<Type>::remove() {
     if ( is_empty() ) {
          cerr << "remove() on empty queue\n";
          exit( - 1 );
     }

     QueueItem<Type> *pt = front;
     front = front->next;

     Type retval = pt->item;
     delete pt;

     cout << "value removed: ";
     cout << retval << endl;

     return retval;
}
```

在如下表达式中：

```
cout << retval << endl;
```

retval 的类型是 Type，它的真正类型要到成员函数 remove()被实例化时才能知道。被选中的 operator<<()依赖于 retval 的实际类型，即依赖于代替模板参数 Type 的类型。所以，直到 remove()被实例化时，才可能知道哪一个 operator<<()会被调用。不同的 remove()实例将可能调用不同的 operator<<()。因此，我们称，被选择的 operator<<()依赖于模板参数。

但是，对 exit()的调用，情况则不同。调用 exit()的函数实参是一个文字常量，在成员函数 remove()的所有实例中它的值都相同。因为这个函数调用没有用到“依赖于模板参数 Type 类型”的实参，所以编译器可以保证，在所有实例中对 exit()的调用，都会调用到在头文件 cstdlib 中声明的函数 exit()。类似的情况，我们知道对于表达式：

```
cout << "value removed: ";
```

总是调用全局操作符：

```
ostream& operator<< ( ostream &, const char * );
```

实参“value removed: ”是 C 风格的字符串，它的类型不依赖于模板参数 Type。所以可以保证，在函数 remove()的所有实例中，operator<<()的这个用法的意义相同。在模板的所有实例中，意义相同的语法结构体是指不依赖于模板参数的语法结构体。

在类模板定义中或类模板成员的定义中，名字解析的两个步骤如下：

1.在模板被定义时，解析出不依赖于模板参数的名字。

2.在模板被实例化时，解析出依赖于模板参数的名字。

这种两阶段的方法可以同时满足来自类模板设计者和类模板用户的要求。作为类模板设计者，我们想尽可能控制模板定义中的名字解析过程。如果类模板是一个库的一部分，并且这个库还定义了其他的模板和函数，那么，我们希望让类模板的实例及其成员尽可能使用库中的其他组件，名字解析过程的第一步可以保证这个要求、当模板定义中用到的名字不依赖于模板参数时，名字解析过程只考虑在模板定义之前头文件中可见的声明。

实际上，类模板的设计者必须确保为“所有用在模板定义中、且不依赖于模板参数的名字”提供声明。如果在模板定义中用到的名字不依赖于模板参数，日该名字的声明在模板被定义时未能找到，则模板定义是错误的。如果在类模板 Queue 的成员函数 remove()的定义之前没有包含头文件 iostream 和 cstdlib，则如下表达式：

```
cout << "value removed: ";
```

或者对 exit()的调用都将是错误的。

假设用某一个类型实例化一个模吧　那么在考虑与这个类型相关的函数和操作符时，名字解析的第二步是必须的。例如，如果我们用在 16.9 节定义的类类型 LongDouble 实例化类模板 Queue，希望在 Queue 的成员函数 remove()中的下列表达式：

```
cout << retval << endl;
```

调用到与 LongDouble 类相关的输出操作符 operator<<()。例如：

```
#include "Queue.h"
#include "ldouble.h"

// contains:
// class LongDouble { ... };
// ostream& operator<< ( ostream &, const LongDouble & );
int main() {
    // Queue<LongDouble> 的实例
    Queue<LongDouble> *qld = new Queue<LongDouble>;
    // Queue<LongDouble>::remove() 的实例
    // 调用 LongDouble 的输出操作符
    qld->remove();

    //...
}
```

一个模板被实例化的确切位置被称作模板的**实例化点**（point of instantiation）。知道一个模板的实例化点的位置很重要，因为它决定了为“依赖于模板参数的名字”所考虑的声明。

类模板的实例化点总是在名字空间域中，而且它总是在“引用类模板实例的声明或定义”之前。类模板的成员函数或静态数据成员的实例化点也总是跟在“引用类模板成员实例的声明或定义”之后。

在上一个例子中，Queue<LongDouble>的实例化点就在 main()之前，编译器考虑在该点之前的所有声明，以便解析“在模板 Queue 定义中用到的、依赖于模板参数”的名字。成员函数 remove()的实例化点紧跟在 main()之后，编译器考虑在该点之前的所有声明，以便解析“在成员函数 remove()的定义中用到的、依赖于模板参数”的名字。

正如 16.2 节提到的，如果一个类模板被用在“要求一个完整类定义的上下文环境”中，则它将被实例化。当类模板被实例化时，类模板实例的成员不会自动被实例化。只有当程序用到这些成员时，它们才被实例化。因此，模板的实例化点可能与它的成员的实例化点不同，并且不同的成员会有不同的实例化点。为防止出错，对于在类模板的定义和它的成员定义中用到的名字，这些名字的声明应该被放到头文件中，而且，在类模板的第一次实例化和其成员的实例化之前，该头文件被包含到代码中。

16.12 名字空间和类模板 ※

与任何其他的全局域定义一样，类模板定义也可以被放在名字空间中（关于名字空间的讨论见 85 节和 8.6 节）。对于这样的类模板定义 其意义与在全局域中定义的类模板相同。只不过模板名被隐藏在名字空间中。当在名字空间之外使用模板时，模板名字必须被名字空间名限定修饰，或者提供一个 using 声明。例如：

```
#include <iostream>
#include <cstdlib>

namespace cplusplus_primer {
    template <class Type>
    class Queue { // ...
    };

    template <class Type>
    Type Queue<Type>::remove()
    {
        // ...
    }
}
```

当类模板名字 Queue 被用在名字空间之外时，它必须被名字空间名 cplusplus_primer 限定修饰，或者通过一个 using 声明而被引入。类模板 Queue 的用法与本章前面描述的相同——以相同的方式实例化，它可以有成员函数、静态数据成员以及嵌套类等等。例如：

```
int main() {
    using cplusplus_primer::Queue;// using 声明

    // 引用名字空间 cplusplus_primer 的类模板
    Queue<tint> *p_qi = new Queue<int>;

    // ...
    p_qi->remove();
}
```

由于下面的 new 表达式：

```
... = new Queue<int>;
```

使用了模板 cplusplus_primer::Queue，所以 cplusplus_primer::Queue<int>被实例化，p_qi 是指向 cplusplus_primer::Queue<int>类类型的指针。当该指针被用来引用成员函数 remove()时，它引用这个模板实例的成员函数 remove()。

在名字空间中声明类模板也会影响“该类模板及其成员的特化和部分特化声明”的方式（关于特化在 16.9 节讨论，部分特化在 16.10 节）。类模板或类模板成员的特化声明必须被声明在“定义通用模板的名字空间”中。

在下面的例子中，类类型 Queue<char*>的特化声明和类类型 Queue<double>的成员函数 remove()的特化声明，都在名字空间 cplusplus_primer 中被声明。

```
#include <iostream>
#include <cstdlib>

namespace cplusplus_primer {
    template <class Type>
    class Queue { ... };

    template <class Type>
    Type Queue<Type>::remove() { ... }

    // cplusplus_primer::Queue<char*> 的特化声明
    template<> class Queue<char*> { ... };

    // cplusplus_primer::Queue<double>::remove() 成员函数的特化声明
    template<> double Queue<double>::remove() { ... }
}
```

尽管这些特化是名字空间 cplusplus_primer 的成员，但是，它们的定义本身不一定出现在名字空间 cplusplus_primer 中。我们也可以在名字空间之外定义模板特化，只要该特化的定义出现在名字空间 cplusplus_primer 的外围名字空间中、并且特化的名字被正确的名字空间名限定修饰。例如：

```
namespace cplusplus_primer
{
    // Queue 及其成员函数的定义
}

// cplusplus_primer::Queue<char*> 的特化声明
template<> class cplusplus_primer::Queue<char*> { ... };

// 成员函数 cplusplus_primer::Queue<double>::remove() 的特化声明
template<> double cplusplus_primer::Queue<double>::remove()
    { ... }
```

cplusplus_primer::Queue<char*>和类类型 cplusplus_primer::Queue<double>的成员函数 remove()的特化声明都是在全局域中被提供的。因为全局域包含了名字空间 cplusplus_primer，所以这些定义对于在名字空间 cplusplus_primer 定义的类模板 Queue 来说，都是有效的特化

定义。

16.13 模板数组类

在本节我们将完成 2.5 节介绍的 Array 类模板的实现（这个类模板将通过 18.3 节的单继承被扩展，以及通过 18.6 节的多继承被进一步扩展）。下面是 Array 类模板的完整的头文件：

```
#ifndef ARRAY_H
#define ARRAY_H
#include <iostream>

template <class elemType> class Array;
template <class elemType> ostream&
     operator<< ( ostream &, const Array<elemType> & );

template <class elemType>
class Array {
public:
     explicit Array( int sz = DefaultArraySize )
          { init( 0, sz ); }
     Array( const elemType *ar, int sz )
          { init( ar, sz ); }
     Array( const Array &iA )
          { init( iA._ia, iA._size ); }
     ~Array() { delete[] _ia; }

     Array & operator=( const Array & );
     int size() const { return _size; }

     elemType& operator[]( int ix ) const
          { return _ia[ix]; }

     ostream &print( ostream &os = cout ) const;
     void grow();
     void sort( int,int );
     int find( elemType );
     elemType min();
     elemType max();
private:
     void init( const elemType *, int );
     void swap( int, int );

     static const int DefaultArraySize = 12;
     int _size;
     elemType *_ia;
};
#endif
```

三个构造函数中的公共代码被抽取到一个独立的成员函数 init()中。由于它不希望被 Array 类模板的用户直接调用，所以它被声明为私有成员。

```
template <class elemType>
void Array<elemType>::init( const elemType *array, int sz )
{
    _size = sz;
    _ia = new elemType[ _size ];
    for ( int ix = 0; ix < _size; ++ix )
        if ( ! array )
            _ia[ ix ] = 0;
        else _ia[ ix ] = array[ ix ];
}
```

拷贝赋值操作符的实现很简单。正如在 14.7 节中提到的，该操作符的实现保证不会拷贝对象自身：

```
template <class elemType> Array<elemType>&
Array<elemType>::operator=( const Array<elemType> &iA )
{
    if ( this != &iA ) {
        delete[] _ia;
        init( iA._ia, iA._size );
    }
    return *this;
}
```

成员函数 print()处理对象的实际输出，该对象的类型是 Array 类模板的实例。它的输出可能过于复杂，但是可以在一页上很漂亮地显示内容。已知 Array<int>类型的实例含有元素 3、5、8、13 和 21，则该对象的输出为：

```
(5) < 3, 5, 8, 13, 21 >
```

ostream 输出操作符只是简单地调用 print()。下面是两个函数的实现：

```
template <class elemType> ostream&
operator<< ( ostream &os, const Array<elemType> &ar )
{
    return ar.print( os );
}

template <class elemType>
ostream & Array<elemType>::print( ostream &os ) const
{
    const int lineLe ngth = 12;
    os << "( " << _size << " )< ";

    for ( int ix = 0; ix < _size; ++ix )
    {
        if ( ix % lineLength == 0 && ix )
            os << "\n\t";
        os << _ia[ ix ];

        // 对于一行的最后一个元素，或者数组的
        // 最后一个元素不产生逗号
        if ( ix % lineLength != lineLength-1 &&
                ix != _size -1 )
            os << ", ";
```

```
        }
        os << " >\n";
        return os;
    }
```

成员函数 print()中的下述语句处理 Array 元素值的真正输出：

```
    os << _ia[ ix ];
```

该语句引入了对于“被用来实例化 Array 类模板” 的类型的隐含要求：被用作模板实参的类型必须是内置类型，或者是定义了自己的输出操作符的用户自定义类类型。如果没有为这样的类型定义输出操作符，则在使用该输出操作符的地方，试图显示这种类型的 Array 实例的操作会导致编译时刻错误。

grow()成员函数增长 array 对象的大小，在我们的例子中，它只把 Array 对象增加为当前大小的一倍半：

```
template <class elemType>
void Array<elemType>::grow()
{
    elemType *oldia = _ia;
    int oldSize = _size;

    _size = oldSize + oldSize/2 + 1;
    _ia = new elemType[_size];

    int ix;
    for ( ix = 0; ix < oldSize; ++ix)
        _ia[ix] = oldia[ix];
    for ( ; ix < _size; ++ix )
        _ia[ix] = elemType();
    delete[] oldia;
}
```

成员函数 find()、min()和 max()实现了在内部数组 ia 上的迭代查找。当然，如果数组是有序的，则这些成员函数可以被实现得更为有效。

```
template <class elemType>
elemType Array<elemType>::min( )
{
    assert( _ia != 0 );
    elemType min_val = _ia[0];
    for ( int ix = 1; ix < _size; ++ix )
        if ( _ia[ix] < min_val )
            min_val = _ia[ix];
    return min_val;
}

template <class elemType>
elemType Array<elemType>::max()
{
    assert( _ia != 0 );
    elemType max_val = _ia[0];
    for ( int ix = 1; ix < _size; ++ix )
        if ( max_val < _ia[ix] )
```

```
                max_val = _ia[ix];
        return max_val;
}
template <class elemType>
int Array<elemType>::find( elemType val )
{
        for ( int ix = 0; ix < _size; ++ix )
            if ( val == _ia[ix] ) return ix;
        return - 1;
}
```

最后，类模板 Array 提供了一个成员函数 sort()。它实现了快速排序算法（quicksort）。该成员函数看起来与 10.11 节定义的非成员函数模板实现有些类似。swap()只是被用作 sort()的辅助函数，它不是类模板 Array 公有接口的组成部分，所以被声明为私有成员：

```
template <class elemType>
void Array<elemType>::swap( int i, int j )
{
        elemType tmp = _ia[i];
        _ia[i] = _ia[j];
        _ia[j] = tmp;
}

template <class elemType>
void Array<elemType>::sort( int low, int high )
{
        if ( low >= high ) return;
        int lo = low;
        int hi = high + 1;
        elemType elem = _ia[low];
        for (;;) {
            while ( _ia[++lo] < elem && lo < high ) ;
                while ( _ia[--hi] > elem && hi > low ) ;
                    if ( lo < hi )
                        swap( lo, hi );
                    else break;
        }
        swap( low, hi );
        sort( low, hi-1 );
        sort( hi+1, high );
}
```

当然，实现了以上代码，并不能保证这些代码就可以真正工作了。try_array()是一个模板函数，可用来测试我们的 Array 类模板的实现。它看起来如下：

```
#include "Array.h"

template <class elemType>
void try_array( Array<elemType> &iA )
{
        cout << "try_array: initial array values:\n";
        cout << iA << endl;
```

```
    elemType find_val = iA [ iA.size()-1 ];
    iA[ iA.size()-1 ] = iA.min();
    int mid = iA.size()/2;
    iA[0] = iA.max();
    iA[mid] = iA[0];
    cout << "try_array: after assignments:\n";
    cout << iA << endl;

    Array<elemType> iA2 = iA;
    iA2[mid/2] = iA2[mid];
    cout << "try_array: memberwise initialization\n";
    cout << iA << endl;

    iA = iA2;
    cout << "try_array: after memberwise copy\n";
    cout << iA << endl;
    iA.grow();
    cout << "try_array: after grow\n";
    cout << iA << endl;

    int index = iA.find( find_val );
    cout << "value to find: " << find_val;
    cout << "\tindex returned: " << index << endl;

    elemType value = iA[index];
    cout << "value found at index: ";
    cout << value << endl;
}
```

让我们来看一看函数模板 try_array()。第一步是输出原始的 Array。这可以证实模板的输出操作符被实例化，并为我们提供原始数组的内容，我们可以用来比较以后对 Array 的修改正确与否。find_val 包含一个值，稍后它被传给 find()。如果 try_array()是一个非模板函数，则该值将是一个常量文字。但是，同为没有一个值能够被用于每一种可能的类型，所以该值不能是一个常量文字。我们随机地用 Array 的一些元素赋给 Array 的其他元素，以此来练习 min()、max()、size()以及下标操作符。

通过调用类模板 Army 的拷贝构造函数，iA2 被按成员初始化，初始值为 iA。然后再通过对元素 mid/2 的赋值，来练习 iA2 的下标操作符（当 iA 是 Array 的派生子类型，并且下标操作符被声明为虚函数时，这两行将更有趣。我们将在第 18 章中关于继承的讨论中再次看到这一点。）随后，通过调用 Array 类的赋值操作符，把修改后的 iA2 按成员拷贝给 iA。之后，程序练习 grow()和 find()成员函数。函数在测试 find()的返回值时会失败。记住，如果没有找到的话，则 find()返回-1。用-1 索引 Array 会导致下溢错误（在第 18 章中，从 Array 派生的、带有边界检查的 Array 类模板会捕获到这个错误）。

我们想证实我们的模板实现是否能在各种数据类型上奏效——例如整数、浮点值和字符串。下面的 main()应用了这三种数据类型的 try_array()：

```
#include "Array.C"
#include "try_array.C"
#include <string>
```

```
int main()
{
    static int ia[] = { 12,7,14,9,128,17,6,3,27,5 };
    static double da[] = {12.3,7.9,14.6,9.8,128.0 };
    static string sa[] = { "Eeyore", "Pooh", "Tigger",
                           "Piglet", "Owl", "Gopher", "Heffalump" };

    Array<int> iA( ia, sizeof(ia)/sizeof(int) );
    Array<double> dA( da, sizeof(da)/sizeof(double) );
    Array<string> sA( sa, sizeof(sa)/sizeof(string) );

    cout << "template Array<int> class\n" << endl;
    try_array(iA);

    cout << "template Array<double> class\n" << endl;
    try_array(dA);

    cout << "template Array<stri ng> class\n" << endl;
    try_array(sA);

    return 0;
}
```

下面是 double 型 Array 类模板实例的输出：

```
try_array: initial array values:
( 5 )< 12.3, 7.9, 14.6, 9.8, 128 >
try_array: after assignments:

( 5 )< 14.6, 7.9, 14.6, 9.8, 7.9 >
try_array: memberwise initialization

( 5 )< 14.6, 7.9, 14.6, 9.8, 7.9 >
try_array: after memberwise copy

( 5 )< 14.6, 14.6, 14.6, 9.8, 7.9 >
try_array: after grow

( 8 )< 14.6, 14.6, 14.6, 9.8, 7.9, 0
0, 0 >

value to find: 128index returned: -1
value found at index: 3.35965e-322
```

越界索引会导致程序返回的最后值是无效的。同样的越界索引会导致类模板 Array 的 string 实例在执行期间崩溃。下面是输出：

```
template Array<String> class
```

```
try_array: initial array values:

( 7 )< Eeyore, Pooh, Tigger, Piglet, Owl, Gopher
     Heffalump >
try_array: after assignments:

( 7 )< Tigger, Pooh, Tigger, Tigger, Owl, Gopher
     Eeyore >
try_array: memberwise initialization

( 7 )< Tigger, Pooh, Tigger, Tigger, Owl, Gopher
     Eeyore >
try_array: after memberwise copy

( 7 )< Tigger, Tigger, Tigger, Tigger, Owl, Gopher
     Eeyore >
try_array: after grow

( 11 )< Tigger, Tigger, Tigger, Tigger, Owl, Gopher
     Eeyore, <empty>, <empty>, <empty>, <empty> >

value to find: Heffalumpindex returned: -1
Memory fault(coredump)
```

练习 16.11

请修改在本节中定义的类模板 Array，去掉成员函数 sort()、find()、max()和 swap()，再把函数模板 try_array()改为使用泛型算法（在第 12 章中定义）。

第五篇

面向对象的程序设计

面向对象的程序设计扩展了基于对象的程序设计，可以提供类型 / 子类型的关系。这是通过一种被称为**继承**（inheritance）的机制而获得的。类不再是重新实现共享的特征，而是继承了其父类的数据成员和成员函数。C++通过一种被称为**类派生**（class derivation）的机制来支持继承。被继承的类为**基类**（base class），而新的类为**派生类**（derived class）。我们把基类和派生类实例的集合称作**类继承层次结构**（hierarchy）

例如，在 3D 计算机图形中，OrthographicCamera 和 PerspectiveCamera 都是从一个抽象基类 Camera 派生而来的。所有相机（camera）共同的操作和数据都被定义在这个抽象的 Camera 类中。每个派生类只实现“它与抽象的 Camera 不同”的部分，或者为继承来的成员函数提供替代的实现，或者引入新的成员。

如果基类和派生类共享相同的公有接口，则派生类被称作基类的**子类型**（subtype）。例如 PerspectiveCamera 是 Camera 的一个子类型。在 C++中，存在特殊的类型 / 子类型关系，基类指针或引用可以直接引用其任何派生子类，而无需程序员介入。[这种“用基类的指针或引用操纵多个类型”的能力被称为**多态**（polymorphism）]。例如，已知函数：

```
void lookAt( const Camera *pcamera );
```

我们在实现 lookAt()时，只要对基类 Camera 的接口进行编程，而与“pcamera 指向的是 PerspectiveCamera、OrthographicCamera 还是某个尚未定义的、从 Camera 派生来的子类型”无关。

每个单独的 lookAt()调用都会被传入一个 Camera 子类对象的地址。编译器会自动地把它转换成适当的基类指针。例如：

```
// ok: 自动被转换成 Camera*
OrthographicCamera ocam;
lookAt( &ocam );

// ...

// ok: 自动被转换成 Camera*
PerspectiveCamera *pcam = new PerspectiveCamera;
lookAt( pcam );
```

lookAt()的实现被屏蔽在应用程序的实际 Camera 子类之外。如果以后我们希望增加或去掉一个子类，那么，无需改变 lookAt()。

子类多态性使得我们在编写应用程序的核心时，可以不用考虑将来需要维护的单个类型。我们利用基类指针和引用，对抽象的基类的公有接口进行编程。在运行时刻，真正要引用的类型被解析出来，并且调用适当的公有接口实例。

在运行时刻需要解析出被调用的函数，这个解析过程被称为**动态绑定**（dynamic bindng）（缺省情况下，函数是在编译时刻被**静态解析**的）。在 C++中，通过一种被称为**虚拟函数**（virtual function）的机制来支持动态绑定。通过继承和动态绑定，子类型多态性为面向对象的程序设计提供了基础，它是以下章节的主题。

第 17 章将覆盖 C++中支持面向对象程序设计的各种设施，并将讨论继承机制怎样影响诸如构造函数、析构函数、按成员初始化和赋值这样一些类的机制。为了使讨论更加形象，我们将开发一个 Query 类层次结构来支持第 6 章引入的文本查询系统。

第 18 章我们将讨论通过多继承和虚拟继承得到的更复杂的继承层次。它将用多继承和虚拟继承，把第 16 章的模板类例子扩展成一个三层的类模板层次结构。

第 19 章我们将讨论运行时刻类型识别（Run-Time Type Identification，RTTI）机制，同时将对于“在继承机制之下的重载函数解析过程”做深入的介绍。此外，我们还将重新检查异常处理设施，以讨论标准库的异常类层次，并说明怎样定义和处理我们自己的异常类。

第 20 章将深入讨论 iostream 库。iostream 库是一个类层次结构，它支持虚拟继承和多继承。

17

类继承和子类型

在第 6 章中，为了引出并展开对于抽象容器类型的讨论，我们介绍了文本查询系统的部分实现，并且在 TextQuery 类中封装了它的最后形式。但是，我们没有实现它的前端——实际的用户查询支持，而是把它推延到我们全面介绍面向对象程序设计的时候。在本章中，我们将把前端查询语言实现为一个单继承的 Query 类层次结构，来介绍和探讨 C++中的面向对象的设计与编程。另外，我们将修改和扩展第 6 章的 TextQuery 类，以便提供一个完整的文本查询系统。

运行我们的文本查询系统的程序如下：

```
#include "TextQuery.h"

int main()
{
    TextQuery tq;
    tq.build_text_map();
    tq.query_text();
}
```

build_text_map()是第 6 章中成员函数 doit()稍作修改之后的形式。它的主要任务是生成个单词位置映射表，由文本中每个重要的词作为索引。（如果你回想一下，可能还记得我们并没有存储无语义的词，如 if、and、but 等等。另外，我们去掉了大写的字母，处理了复数后缀，如把 testifies 转换成 testify，把 marches 转换成 march。）与每个词相关联的是一个位置向量，其中每个向量元素存储了该单词在文本中出现的行和列位置。

query_text()请求每一个用户查询，并利用单继承和动态绑定机制把每个用户查询转换成一个内部的面向对象的 Query 类层次。build_text_map()构造得到的单词位置映射表，可被用来计算每个查询的内部表示。查询的结果是一个惟一的集合，由文本文件中所有满足查询准则的行为构成。例如：

```
Enter a query-please separate each item by a space.
Terminate query (or session) with a dot( . ).

==> fiery && ( bird || shyly )
```

```
        fiery ( 1 ) lines match
        bird ( 1 ) lines match
        shyly ( 1 ) lines match
        ( bird || shyly ) ( 2 ) lines match
        fiery && ( bird || shyly ) ( 1 ) lines match

    Requested query: fiery && ( bird || shyly )

    ( 3 ) like a fiery bird in flight. A beautiful fiery bird, he tells her,
```

我们选择支持的查询设施由下列元素构成：

1．单个词，如 Alice 或 untamed。凡是有单词出现的所有行，其行号都会被显示在小括号中，并按递增的顺序来显示。例如：

```
    ==> daddy

    .
        daddy ( 3 ) lines match

        Requested query: daddy

        ( 1 ) Alice Emma has long flowing red hair. Her Daddy says
        ( 4 ) magical but untamed. "Daddy, shush, there is no such thing,"
        ( 6 ) Shyly, she asks, "I mean, Daddy, is there?"
```

2．非（Not）查询，使用!操作符。凡是文本中该名字没有出现的所有行都被显示出来。例如，下面是第 1 项的非：

```
    ==> ! daddy

    .
        daddy ( 3 ) lines match
        ! daddy ( 3 ) lines match

    Requested query: ! daddy
    ( 2 ) when the wind blows through her hair, it looks almost alive,
    ( 3 ) like a fiery bird in flight. A beautiful fiery bird, he tells her,
    ( 5 ) she tells him, at the same time wanting him to tell her more.
```

3．或（Or）查询，使用||操作符。任何一行只要包含两个名字中的一个，该行就被显示出来。例如：

```
    ==> fiery || untamed

    .
        fiery ( 1 ) lines match
        untamed ( 1 ) lines match
        fiery || untamed ( 2 ) lines match

    Requested query: fiery || untamed
    ( 3 ) like a fiery bird in flight. A beautiful fiery bird, he tells her,
    ( 4 ) magical but untamed. "Daddy, shush, there is no such thing,"
```

4．与（And）查询，使用&&操作符。任何一行如果同时包含两个单词，并且相邻，则

该行被显示出来。这也包括“一行的最后一个单词以及下一行的第一个单词”的情形。例如：

```
==> untamed && Daddy
.
     untamed ( 1 ) lines match
     daddy ( 3 ) lines match
     untamed && daddy ( 1 ) lines match

Requested query: untamed && daddy

( 4 ) magical but untamed. "Daddy, shush, there is no such thing,"
```

这些元素也可以被组合起来，比如：

```
fiery && bird || shyly
```

但是，运算的顺序是从左向右，每个元素维持相同的优先级。所以上一个复合查询的运算结果是 fiery bird 或 shyly，而不是 fiery bird 或 fiery shyly：

```
==> fiery && bird || shyly
.
     fiery ( 1 ) lines match
     bird ( 1 ) lines match
     fiery && bird ( 1 ) lines match
     shyly ( 1 ) lines match
     fiery && bird || shyly ( 2 ) lines match

Requested query: fiery && bird || shyly

( 3 ) like a fiery bird in flight. A beautiful fiery bird, he tells her,
( 6 ) Shyly, she asks, "I mean, Daddy, is there?"
```

为了允许对一个查询划分子组（subgrouping），我们的查询设施必须支持括号。例如：

```
fiery && ( bird || shyly )
```

找到所有对于 fiery bird 或 fiery shyly 的引用，查询的结果在本节开始处显示过。

我们的系统必须足够智能，以确保不会多次显示相同的行。

17.1 定义一个类层次结构

本章中，我们主要的焦点是建立一个类层次结构，来代表用户查询。我们的初始设计是把每个查询操作表示成一个单独的类：

```
NameQuery     // Shakespeare
NotQuery      // !Shakespeare
OrQuery       // Shakespeare || Marlowe
AndQuery      // William[28] && Shakespeare
```

每个类都定义了一个 eval()成员函数，用来计算出每个操作所代表的查询。例如，

[28] 为了简化我们的系统，我们要求用空格分割每一个单词，包括括号和查询操作符、而在实际的系统中，这是不合理的，我们认为在这种以介绍性为主的文本中，这个要求是可以被接受的。

NameQuere 的成员函数 eval()只是简单地返回该单词出现的行列数位置向量（见 6.8 节）。但是 OrQuery 的成员函数 eval()必须建立起它的两个操作数的位置向量的并集，等等。

因此，查询：

```
untamed || fiery
```

由一个 OrQuery 类对象构成，它包含两个 NameQuery 对象作为操作数。这样就可以支持简单的查询，但是，在处理下面的复合查询时会出现一个问题：

```
Alice || Emma && Weeks
```

该查询由两个子查询构成：一个 OrQuery 对象，它含有 NameQuery 对象 Alice 和 Emma，以及一个 AndQuery 对象。AndQuery 对象的右操作数是 NameQuery Weeks。

```
AndQuery
    OrQuery
        NameQuery ("Alice")
        NameQuery ("Emma")
NameQuery ("Weeks")
```

但是左操作数是它前面的 OrQuery 对象。它也很可能代表了一个 NotQuery 或另一个 NameQuery 对象。当一个操作数是四种可能的查询（query）类类型之一时，我们在内部怎样表示这个操作数？这个问题有两个方面：

1．我们需要能够声明操作数的类型，它可能是 OrQuery、AndQuery 和 NotQuery，这样每种查询类都可以包含四种不同的查询类类型。

2．不管第 1 项如何解决，我们需要能够在运行时刻针对每一个操作数，调用“特定于一个类（class-specific）”的 eval()成员函数实例。

非面向对象的方案是，把操作数定义成一个 union 型，并提供一个**判别式**（discriminant）来指示操作数实际的类型：

```
// 非面向对象的方案
union op_type {
    // union 不能包含带有相关构造函数的类对象
    NotQuery *nq;
    OrQuery *oq;
    AndQuery *aq;
    string *word;
};

enum opTypes {
    Not_query=1, Or_query, And_query, Name_query
};

class AndQuery {
public:
    // ...

private:
    /*
    * op_types 含有查询的实际操作数类型
    * opTypes 确定每个操作数的类型
    */
```

```
        op_type _lop, _rop;
        opTypes _lop_type, _rop_type;
};
```

另外一种方案是，可以抛开 union，通过一个 void*指针来存储对象：

```
class AndQuery {
public:
        // ...
private:
        void *_lop, *_rop;
        opTypes _lop_type, _rop_type;
};
```

这里仍然需要判别式，因为我们不能直接使用由 void*指针指向的对象，没有办法可以查询指针本身的类型。（在 C++下，我们不建议使用这种方案，但是，它是 C 语言中一个常见的程序设计习惯。）

这两种方案的主要缺点是把类型解析的负担交给了程序员。例如，在 void*方案中，AndQuery 的 eval()操作可能被实现如下：

```
void
AndQuery::
eval()
{
        // 非面向对象的方案
        // 把类型解析的负担留给了程序员
        // 指出左操作数的实际类型
        switch( _lop_type ) {
            case And_query:
                AndQuery *paq = static_cast<AndQuery*>(_lop);
                paq->eval();
                break;
            case Or_query:
                OrQuery *poq = static_cast<OrQuery*>(_lop);
                poq->eval();
                break;
            case Not_query:
                NotQuery *pnotq = static_cast<NotQuery*>(_lop);
                pnotq->eval();
                break;
            case Name_query:
                NameQuery *pnmq = static_cast<NameQuery*>(_lop);
                pnmq->eval();
                break;
        }
        // 对右操作数同样
}
```

“由程序员显式管理类型解析”方案的主要弊处是：直接处理每个类型而带来的代码长度和复杂性，以及对于所支持的类型集合，很难增加或删除一个类型而不打断现有的代码。

面向对象的程序设计提供了另外一种方案，它把类型解析的负担从程序员身上转移到编译器上。例如，在面向对象设计下，AndQuera 的 eval()操作被重新实现［eval()被声明为虚拟

函数］如下：

```
// 面向对象的方案
// 类型解析的负担被转移到编译器上
// note: _low 和 _rop 现在是类类型的对象
// 它们的定义在以后给出
void
AndQuery::
eval()
{
    _lop->eval();
    _rop->eval();
}
```

即使我们从所支持的类型集中增加或删除一个类型，这部分代码也无需修改或重新编译。

17.1.1 面向对象的设计

如果用面向对象的设计方法，四种查询（query）类型应该由什么构成呢？怎样解决前面所说的两个问题呢？

通过继承，我们为前面四种独立的查询类类型之间定义了一种关系。我们引入一个抽象的 Query 类作为基类，其他几个类都从它派生（或生成）。一个抽象类可以被看作一个不完整的类，它由每个后续的派生类或多或少加以补充才得以完成——在我们的例子中，派生类是四个查询类型 AndQuery、OrQuery、NotQuery 和 NameQuery。

我们的抽象 Query 基类定义了所有查询类型公共的数据成员和成员函数集。Query 的派生类，如 AndQuery，定义了只与每个特定的查询才有关的内容。例如，NameQuery 是 Query 的一个特殊实例，它的操作数总是 string。我们称 NameQuery 为派生类。我们也说 Query 被用作基类。（对其他查询类型也一样。）派生类继承了其基类的数据成员和成员函数，并且可以直接使用它们，就好像它们是派生类的成员一样。

继承层次结构的主要好处是，我们可以针对抽象基类的公有接口进行编程，而不是针对组成继承层次的个别类型，通过这种方式，我们的代码可以不受层次结构变化的影响。例如。我们把 eval()定义为抽象 Query 基类的公有虚拟函数，通过下面的代码

```
_rop->eval();
```

用户代码可以不受查询语言变化的影响。这不但允许增加、删除、修改类型而无需改变用户程序，而且新的查询类型的提供者不用重新编写“对层次结构中的全部类都相同”的行为或动作。这是由继承机制的两种特殊性质来支持的：多态和动态绑定。

当我们在 C++中说到多态性时，我们主要指基类的指针或引用可以指向其任意派生类的能力。例如，如果我们定义了非成员函数 eval()如下，

```
// pquery 可以指向任何从 Query 派生的类型
// pquery can address any of the classes derived from Query
void eval( const Query *pquery )
{
    pquery->eval();
}
```

我们可以传递任何一种查询类型对象的地址，来合法地调用它：

```
int main()
{
    AndQuery aq;
    NotQuery notq;
    OrQuery *oq = new OrQuery;
    NameQuery nq( "Botticelli" );

    // ok: 都是从 Query 派生的
    // 编译器自动转换到基类
    eval( &aq );
    eval( &notq );
    eval( oq );
    eval( &nq );
}
```

如果试图用不是从 Query 派生的类对象的地址来调用 eval()，则导致编译时刻错误:

```
int main()
{
    string name( "Scooby-Doo" );

    // 错误: string 不是从 Query 派生的
    eval( &name );
}
```

在 eval()中，

```
pquery->eval();
```

执行时必须根据 pquery 指向的实际类对象，来调用适当的 eval()虚拟成员函数。在上一个例子中，pquery 依次指向 AndQuery 对象、NotQuery 对象、OrQuery 对象和 NameQuery 对象。在程序执行期间的每个调用点上，pquery 所指的真正的类类型才被确定下来，并调用适当的 evel()实例。动态绑定正是完成这项工作的机制。（我们将在 17.5 节详细介绍虚拟函数的设计和使用。）

在面向对象的程序设计中，程序员操纵某一个绑定的一个未知的实例，该绑定的类型是一个无限的集合。（这些类型通过继承层次结构被绑定起来。然而，在理论上，对于层次的广度和深度没有限制。）在 C++中，这只能通过操纵基类指针和引用来实现。在基于对象的程序设计方法中，程序员操纵一个确定类型的实例，该类型在编译点的时候已经被完全定义了。

虽然一个对象的多态操纵行为要求“通过指针或引用来访问该对象”，但是，在 C++中，指针或引用的操作本身不一定导致多态性。例如，考虑如下代码：

```
// 没有多态
int *pi;

// 没有语言支持的多态
void *pvi;

// ok: pquery 可以指向任何 Query 派生类
Query *pquery;
```

在 C++中，多态性只存在子类继承层次中。void*型的指针可以被描述为多态，但是语言本身并没有显式地支持它们——即，它们必须由程序员自己来管理，程序员可以通过“显式强制类型转换，以及记录实际类型的判别式”来做到这一点。[有人可能会说它们不是第一等（first-class）的多态对象。]

C++语言以下列几种方式支持多态性：

1．通过一个隐式转换，从“派生类指针或引用”转换到“其公有基类类型的指针或引用”

```
Query *pquery = new NameQuery( "Glass" );
```

2．通过虚拟函数机制：

```
pquery->eval();
```

3．通过 dynamic_cast 和 typeid 操作符（将在 19.1 节详细讨论）：

```
if ( NameQuery *pnq =
     dynamic_cast< NameQuery* >( pquery )) ...
```

我们通过把 AndQuery、NotQuery 和 OrQuery 类型的每个操作数定义为 Query*的指针，来解决我们的操作数表示问题。例如：

```
class AndQuery {
public:
     // ...
private:
     Query *_lop;
     Query *_rop;
};
```

这两个操作数现在都可以指向“从抽象 Query 基类派生”的任何类型的查询类类型，无论是现在已经定义的还是将来要定义的。由于虚拟机制，发生在程序执行期间的每个操作数的计算过程与它的实际类型无关：

```
_rop->eval();
```

图 17.1 说明了抽象 Query 类及其派生类的继承层次结构。怎样把图 17.1 转换成 C++程序代码呢？

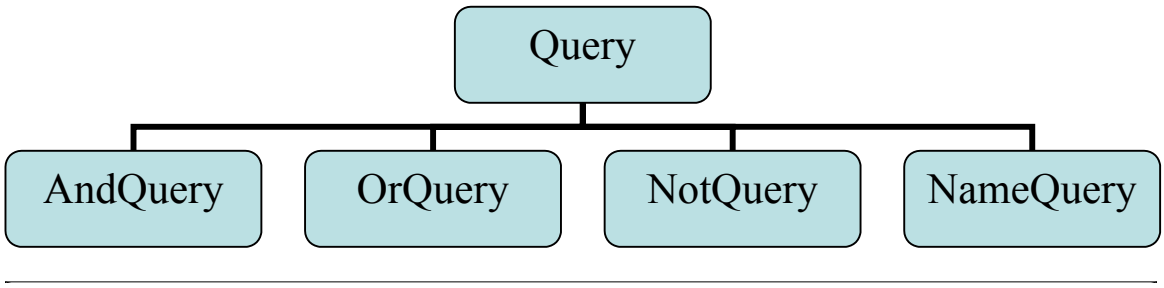


图 17.1 Query 类层次结构

在 2.4 节中，我们看到过 IntArray 类层次的实现。对于图 17.1 给出的 Query 类层次，其定义语法与此类似：

```
class Query { ... };
```

```
class AndQuery : public Query { ... };
class OrQuery : public Query { ... };
class NotQuery : public Query { ... };
class NameQuery : public Query { ... };
```

继承关系通过类派生表（class derication list）来指定。在单继承下，它的一般形式为：

```
: access-level base-class
```

这里 access-level 是 public、protected 或 private 之一（关于 protected 和 private 继承的意义将在 18.3 节讨论），而 base-class 是前面已经定义过的类名。例如，Query 可被用作四个查询类型的公有基类。

在派生表中指定的类必须首先被定义好，方可被指定为基类。例如，下面的 Query 的前向声明不足以使其被用作基类：

```
// 错误：Query 必须已经被定义
class Query;
class NameQuery : public Query { ... };
```

派生类的前向声明不能包括它的派生表，而只是类名——与非派生类一样。例如，下面的 NameQuery 的前向声明导致编译时刻错误:

```
// 错误：前向声明不能包含派生类的派生表
class NameQuery : public Query;
```

正确的前向声明如下：

```
// 派生类与非派生类的前向声明只列出类名
class Query;
class NameQuery;
```

Query 基类与 2.4 节的 IntArray 基类的主要区别是，Query 不代表我们的应用领域中的一个真正类型。IntArray 类层次的用户可以直接定义并操纵 IntArrny 类对象，而 Query 类层次的用户只能定义 Query 指针和引用。它们被用来间接操纵“从 Query 派生的类对象”。Query 被称为**抽象基类**（abstract base class），而 IntArray 则是**实体基类**（concrete base class）。面向对象设计的主要形式是一个抽象基类的定义（如 Query），以及它的公有派生。

练习 17.1

一个图书馆支持下列类别的借阅资料，每个类别都有自己的借阅、登记策略。请把它们组织到一个继承层次中：

```
book            audio book
record          children's puppet
video           sega video game
rental book     sony playstation video game
cdrom book      nintendo video game
```

练习 17.2

请从下面含有一组类型的一般抽象中选择一个（或选择你自己的一个抽象）。把这些类型组织到一个继承层次中：

(1) 图形文件格式（如 gif、tiff、jpeg、bmp）
(2) 几何图形（如方、圆、球、圆锥）
(3) C++语言类型（如类、函数、成员函数）

17.2 确定层次的成员

如 2.4 节所描述，在基于对象的设计中，一般有一个类的提供者和多个使用者，提供者设计这个类，通常也会实现这个类。用户则使用由提供者给出的公有接口。这样的行为分离反映在对类 private 和 public 访问级别的划分上。

在继承机制下，有多个类的提供者：一个提供基类实现（可能有某些派生类）以及一个或另一些在继承层次的生命期中提供派生类。这也是一种实现行为。子类的提供者常常（但不总是）需要访问基类的实现。为提供这样的能力，并且仍然防止对于实现细节的一般性访问，C++提供了另一种访问级别：protected。一个类的 protected 区域中的数据成员和成员函数，虽然对于一般程序仍然是不可访问的，但是，对于派个类却是可用的。（放在其类的 priivat 区域中的项只被提供给基类自己，而不是任何一个派生类。）

把一个成员指定为 public 的标准在基于对象和面向对象的设计之间没有区别。真正的变化在于，是把一个非公有成员声明为 protected 还是 private。如果我们希望防止后来的派生类直接访问成员，则把它声明为 private（对基类而言）。如果我们认为一个成员为“后来的、要求直接访问该成员的派生类”提供了一个操作或数据存储，以使派生类的实现更为有效，则把这个成员声明为 protected。在设计一个基类时，设计者还要考虑的是“确定哪些成员函数是类型相关的”。它们是类层次结构中的虚拟函数。

我们的 Query 类层次结构的下一个设计步骤是确定：

1．Query 类层次结构的公有接口应该提供哪些操作？
2．这些操作之中，哪些应该被声明为虚拟的？
3．单个的派生类需要哪些其他的操作？
4．在抽象 Query 类中应该声明哪些数据成员？
5．单个的派生类要求哪些数据成员？

不幸的是，对于这些问题的答案没有任何神奇的公式，而且，就算回答了也不能保证正确性和完整性。正如我们将要看到的，面向对象设计过程是反复迭代的，它要求对不断演化过程中的类层次结构进行添加和修改。在本节余一下部分，我们将介绍 Query 类层次结构的迭代演化过程。

17.2.1 定义基类

Query 类的成员代表了：

1．被所有的派生查询类型支持的操作集。这包括由派生类类型改写的虚拟操作以及在派生类之间共享的非虚拟操作。对于每种情况我们都将介绍一个例子。

2．对于派生类公共的数据成员集。通过把这些成员从派生类抽取到抽象 Query 类中，我们就能够访问这些“独立于正在操作的实际类型”的成员。我们也将看到两个例子。

已知如下的查询：

```
fiery || untamed
```

两个主要的操作是 1）求出与该查询相匹配的文本行；2）向用户显示匹配的行。我们分别把这两个操作命名为 eval()和 display()。

eval()的计算对于每个派生的查询类类型是特有的，所以必须在 Query 类定义中声明为虚拟的。每个派生类必须提供自己的实现。Query 基类提供公有接口 供我们编程使用。

对于文本中已经匹配到的行的显示操作 display()，它独立于实际的派生类查询类型。该算法需要访问文本本身的内容以及与查询相匹配的行的列表。无论该操作是 AndQuery。OrQuery 或其他什么，这个算法都保持不变。我们不是复制该操作以及每个派生类中支持的数据，而是在 Query 中定义了惟一一份实例，然后让每个派生类继承该操作。

在这种设计下，我们可以调用任何操作而无需知道正在操作的对象的实际类型。例如：

```
void
doit( Query *pq )
{
    // 虚拟调用
    pq->eval();

    // 静态调用 Query::display()
    pq->display();
}
```

应该怎样表示文本中被匹配到的行呢？对于查询中的每个词，它的出现由一个相关的位置向量来指示，这些位置向量是在处理文本时生成的。位置是一个 short 整数的行列对。由 build_text_map()构造的单词位置映射表包含所向被系统识别的单词的位置向量，并且以一个代表相应单词的字符串作为索引。例如，已知输入文本：

```
Alice Emma has long flowing red hair. Her Daddy says
when the wind blows through her hair, it looks almost alive,
like a fiery bird in flight. A beautiful fiery bird, he tells her,
magical but untamed. "Daddy, shush, there is no such thing,"
she tells him, at the same time wanting him to tell her more.
Shyly, she asks, "I mean, Daddy, is there?"
```

下面是文本位置映射表，对于某些单词有多个项［单词表示映射的键（key），括号中的值对表示位置向量的元素——注意，行和列都以 0 开始计数］：

```
bird      ((2,3),(2,9))
daddy     ((0,8),(3,3),(5,5))
fiery     ((2,2),(2,8))
hair      ((0,6),(1,6))
her       ((0,7),(1,5),(2,12),(4,11))
him       ((4,2),(4,8))
she       ((4,0),(5,1))
tell      ((2,11),(4,1),(4,10))
```

但是，位置向量不能表示查询的结果。例如，虽然 fiery 出现在两个位置中，但是，它只代表一个实际要显示的行。

我们需要计算由位置向量表示的行的惟一集合。一种策略是，创建一个记录行数的向量（vector），它包含了每个位置向量中的行数。把这个向量传递给 unique()泛型算法，消除重复的元素［关于 unique()用法的讨论和示例见附录］。余下的行应当是升序的。为了确保这

一点，我们可以对行向量应用 sort()泛型算法。

我们选择的另一种策略是，生成一个 set 对象，其中包含位置向量中的行数，set 对象自动以升序包含一组没有重复的值。于是，我们需要一个函数把位置向量转换成惟一行数的集合：

```
set<short>* Query::_vec2set( const vector< location >* );
```

我们把_vec2set 声明为 protected 的 Query 成员函数。它不是 public 的，因为它不属于“期望被 Query 类层次结构用户调用的操作集”。它也不是 private 的，因为它是一个辅助函数，希望能够被派生类使用（下划线表示它不是 Query 类层次结构的公有接口部分）。

例如，bird 的位置向量含有两项，但这两项属于同一行，所以它的结果集（solution set）只包含一个项：(2)。tell 的位置向量含有三项，其中两项属于同一行，所以它的结果集包含两个项：(2,4)。下面是前面给出的位置向量所对应的结果集：

```
bird      (2)
daddy     (0,3,5)
fiery     (2)
hair      (0,1)
her       (0,1,2,4)
him       (4)
she       (4,5)
tell      (2,4)
```

NameQuery 的计算很简单，获得与该名字相关的位置向量，并把该向量转成一个无重复行的集合，然后再显示文本的相关行。

NotQuery 表示“操作数没有出现的所有行”。以下查询：

```
! daddy
```

表示行(1,2,4)的集合。为了计算这个集合，我们需要知道文本中包含多少行。（我们从来没有计算过这个信息，因为在此之前，我们并不需要它。但是，很快我们就会用到它，甚至还需要更多的信息！）为了使计算 NotQuery 操作更容易，最方便的做法是，生成文本中所有行的集合(0,1,2,3,4,5)。然后，我们就可以通过取两个集合的差 set_difference()来解决这个问题（针对“daddy”的 NameQuery 的集合是(0,3,5)。

OrQuery 表示其每个操作数出现的所有行的并集。例如，给定查询：

```
fiery || her
```

无重复的行集合是(0,1,2,4)。这是“与 fiery 相关的行(2)”以及“与 her 相关的行(0,1,2,4)”的并集。行结果集合不能含有重复的行数，而且必须是升序。

到现在为止，通过处理无重复的行数的集合，我们已经能够计算每个查询。但是，AndQuery 要求检查每个位置对的行列值。例如，如下查询的操作数：

```
her && hair
```

出现在四个独立的行中。正如前面已经定义的，AndQuery 的语义要求一个匹配的行必须包含严格顺序的 her hair。第一行出现了这两个单词，尽管它们是相邻的，但也不算匹配：

```
Alice Emma has long flowing red hair. Her Daddy says
```

而两个单词在第二行中的出现完全匹配：

```
when the wind blows through her hair, it looks almost alive,
```

her 的另两次出现也包含了不相邻的 hair——显然不匹配。所以该查询的结果是文本的第二行：(1)。

如果不是因为 AndQuery 操作，我们就无需为每个操作维护位置向量。但是，因为任何派生的查询类型都可能是 AndQuery 的操作数，所以每个类型都必须计算并维护无重复行的集合，以及行列位置对。例如，考虑下列两个查询：

```
fiery && ( hair || bird || potato )
fiery && ( ! burr )
```

NotQuery 有可能成为 AndQuery 的操作数，这意味着我们不能只是简单地创建一个仅含有每个行数项的向量，还需要有一个向量包含文本中每个行和列的位置对。[在 7.5 节介绍 NotQuery 的 eval()函数时，我们会再次看到这一点。]

所以，一个很必要的数据成员是“与每个操作的计算过程相关联的位置向量”。我们可以把它声明为每个派生类的一个成员，也可以把它声明为抽象 Query 基类的一个成员，然后再被每个派生类继承，我们必须两者之中在做出选择。这两种表示方式所需的内存空间是相同的。通过把它放到公共 Query 基类中，我们可以把“对它的初始化和访问支持”限制在局部区域中（即基类中），这是我们的选择。

无论是把无重复行数的集合［我们将其称为**行结果集合**（line solution set）］表示为数据成员，还是在每次需要时再动态计算它，都只是一个实现决策而已。我们选择按需计算，然后，把它的地址收藏起来，供以后使用。同时，它也被声明为抽象 Query 基类的一个成员。

为了显示已经匹配的行的集合，我们需要行结果集合以及存储这些行的实际文本文件。尽管每一个操作都要求自己的位置向量，但是，它们只需要一个共享的文本文件的实例。所以，我们把它定义为 Query 的静态数据成员。[display()函数的实现只依赖于这两个成员。]

下面是 Query 抽象基类的第一个版本，但是还没有声明任何构造函数、析构函数和拷贝赋值操作符（这些将分别在 17.4 节和 17.6 节完成）：

```
#include <vector>
#include <set>

#include <string>
#include <utility>

typedef pair< short, short > location;
class Query {
public:
    // 构造函数和析构函数在 17.4 节讨论
    // 拷贝构造函数和拷贝赋值操作符在 17.6 节讨论
    // 支持公有接口的操作
    virtual void eval() = 0;
    void display () const;

    // 读访问函数
    const set<short> *solution() const;
```

```
    const vector<location> *locations() const { return &_loc; }

    static const vector<string> *text_file() {return _text_file;}
protected:
    set<short> *_vec2set( const vector<location>* );

    static vector<string> *_text_file;

    set<short> *_solution;
    vector<location> _loc;
};

inline const set<short>*
Query::
solution()
{
    return _solution
        ? _solution
        : _solution = _vec2set( &_loc );
}
```

特殊的语法：

```
virtual void eval() = 0;
```

表明抽象基类 Query 没有为函数 evel()提供虚拟定义。为什么？因为没有有意义的算法可供定义。这样的 eval()实例被称作一个**纯虚拟函数**（pure virtual function）。它被用作类层次结构的公有接口中的一个占位符。它也不希望在程序中被调用，而每个后续的派生类都将提供一个实际的实例。（我们将在 17.5 详细讨论虚拟函数。）

17.2.2 定义派生类

每个派生类都继承了其基类的数据成员和成员函数，派生类只需编写与基类行为不同或扩展的方面。例如，NameQuety 必须定义 eval()。另外，它需要为单词的名字提供支持。我们将用 string 成员类对象表示该名字。最后，为了获取相关联的位置向量，我们还必须使用单词位置映射表。因为所有的 NameQuery 类对象实例只需要共享一个单词位置映射表，所以我们把它声明为静态数据成员。下面是 NameQuery 类的初始定义（现在，我们暂时不考虑构造函数、析构函数和拷贝赋值操作符）：

```
typedef vector<location> loc;

class NameQuery : public Query {
public:
    // ...
    // 改写 virtual Query::eval() 实例[29]
    void eval();

    // 读访问函数
```

[29] 继承而来的派生类虚拟函数实例比如 eval()。不再需要（但也可以）指定关键字 virtual。编译器会比较函数的原型，从而识别出这个实例。

```
        string name() const { return _name; }

        static const map<string,loc*> *word_map() { return _word_map; }
    protected:
        string _name;
        static map<string,loc*> *_word_map;
    };
```

除了为 eval()虚拟函数提供自己的实例外，NotQuery 类还必须为它的单个操作数提供支持。因为这个操作数可以是任何派生的查询类类型，所以我们将其定义为 Query 类型的指针。记住，NotQuery 不仅要提供“操作数不出现的文本行”。而且还要提供每行中的所有列位置。例如，已知 NotQuery：

```
!daddy
```

它的 NameQuery 操作数含有下列位置向量：

```
daddy ((0,8),(3,3),(5,5))
```

NotQuery 位置向且必须包含行（1,2,4)的所有列。另外，它还必须包含行(0)的除了列(8)之外的所有列，行(3)的除了列(3)之外的所有列，行(5)的除了列(5)之外的所有列。

计算这些信息的最简单方式是，有一个共享的位置向量，它含有文本中每个单词出现的行列对。在 17.5 节给出 NotQuery 的函数 eval()时，我们将介绍这个实现。总之，我们把这个成员定义为 NotQuery 的静态成员。

下面是 NotQuery 的初始类定义（再次说明，我们没有考虑构造函数、析构函数以及拷贝赋值操作符）：

```
class NotQuery : public Query {
public:
    // ...

    // 另外一种语法：显式的 virtual 关键词
    // 改写 Query::eval()
    virtual void eval();

    // 读访问函数
    const Query *op() const { return _op; }
    static const vector<location>*all_locs(){ return _all_locs; }

protected:
    Query *_op;
    static const vector< location > *_all_locs;
};
```

AndQuery 和 OrQuery 类都是一元操作符，所以必须支持左右操作数。两个操作数都可以是任何派生的查询类类型，因此，我们把这两个成员都定义为 Query 的指针类型。它们还必须分别提供虚拟函数 eval()的实例。下面是 OrQuery 类的初始定义：

```
class OrQuery : public Query {
public:
    // ...
```

```
        virtual void eval();

        const Query *rop() const { return _rop; }
        const Query *lop() const { return _lop; }

    protected:
        Query *_lop;
        Query *_rop;
    };
```

另外，每个 AndQuery 类对象必须能够访问每一行包含的单词的个数。否则 AndQuery 计算时将不能找到跨越两行的相邻的单词。例如，已知查询：

```
tell && her && magical
```

满足匹配的序列跨越了第三和第四行：

```
like a fiery bird in flight. A beautiful fiery bird, he tells her,
magical but untamed. "Daddy, shush, there is no such thing,"
```

三个单词的相关位置向量如下：

```
her       ((0,7),(1,5),(2,12),(4,11))
magical   ((3,0))
tell      ((2,11),(4,1),(4,10))
```

除非 AndQuery 的函数 eval()能够确定行(2)有 12 个单词，否则它不能判定 magical 与 her 相邻。我们将通过一个名为_max_col 的静态数据成员，来提供这一份实例。[eval()的实现将在 17.5 节详细给出。]下面是 AndQuery 类的初始定义：

```
class AndQuery : public Query {
public:
    // 构造函数在 17.4 节讨论
    virtual void eval();

    const Query *rop() const { return _rop; }
    const Query *lop() const { return _lop; }

    static void max_col( const vector< int > *pcol )
        { if ( !_max_col ) _max_col = pcol; }

protected:
    Query *_lop;
    Query *_rop;

    static const vector< int > *_max_col;
};
```

17.2.3 小结

每个派生类的公有接口都是由“通过继承得到的 Query 的公有成员”和“派生类自己的公有成员”构成。当我们写下如下语句时：

```
Query *pq = new NameQuery( "Monet" );
```

通过 pq 只能访问 Query 的公有接口。当写如下语句时：

```
pq->eval();
```

因为 eval()被声明为虚拟函数，所以真正被调用的是“与 pq 实际所指的派生类对象相关联”的 eval()实例。在这种情况下，NameQuery 实例被调用。当写如下语句时：

```
pq->display();
```

总是调用非虚拟的 Query 函数 display()。但是，isplay()仍然反映了由 pq 指向的实际派生类对象的结果集。这种情况下，我们不是依赖于虚拟机制，而是把共享的操作及其支持数据抽取到公共的抽象 Query 基类中。display()是一个多态程序设计的例子，但是它不是由虚拟函数机制支持的，而仅仅由继承机制来支持。下面是我们的实现（正如我们将在最后一节所看到的，这只是一个过渡方案）：

```
void
Query::
display()
{
    if ( ! _solution->size() ) {
        cout << "\n\tSorry, "
             << " no matching lines were found in text.\n"
             << endl;
    }

    set<short>::const_iterator
        it = _solution->begin(),
        end_it = _solution->end();

    for ( ; it != end_it; ++it ) {
        int line = *it;

    // 文本行不要从 0 开始，这样会把用户弄糊涂...
        cout << "( " << line+1 << " ) "
             << (*_text_file)[line] << '\n';
    }
    cout << endl;
}
```

在本节中，我们已经提供了 Query 类层次结构的初次定义。我们还没有考虑的问题是，怎样用类层次结构来构建实际的数据结构，并以此代表用户的查询。实际上，为了实现这一点，我们需要修改和扩展刚刚给出的定义。在讨论那些内容之前，我们需要更详细地看一看 C++下的继承机制。

练习 17.3

考虑 17.1 节末尾的练习 17.1 的图书馆类层次结构的下列成员。请指出哪些实例可能是虚拟函数的候选者，哪些可能会在所有库资料之间是公共的，因此可以出现在基类中。（注意，LibMember 表示“能够借阅图书馆资料的图书馆成员”的抽象。Date 是一个类，用来表示特定年份的一个日期。

```
class Library {
public:
```

```
    bool check_out( LibMember* );
    bool check_in ( LibMember* );
    bool is_late( const Date& today );
    double apply_fine();
    ostream& print( ostream&=cout );

    Date* due_date() const;
    Date* date_borrowed() const;

    string title() const;
    const LibMember* member() const;
};
```

练习 17.4

请指出 17.1 节的练习 17.2 中选择的类层次结构的基类和派生类成员，指出虚拟函数以及公有和被保护的成员。

练习 17.5

下列语句哪些不正确？

```
class Base { ... };

(a) class Derived : public Derived { ... };
(b) class Derived : Base { ... };
(c) class Derived : private Base { ... };
(d) class Derived : public Base;
(e) class Derived inherits Base { ... };
```

17.3 基类成员访问

派生类对象实际上是由多个部分组成的。每个基类代表了一个由该基类的非静态数据成员组成的子对象（subobject）。派生类对象由其基类子对象以及“由派生类的非静态数据成员构成的派生部分”组成。例如。NameQuery 类对象，由 Query 子对象（包含继承而得的数据成员_loc 和_solution）以及 NameQuery 类部分（包含数据成员_name）组成。

在派生类中，继承得到的基类子对象的成员可以被直接访问，就好像它们是派生类的成员一样。（继承链的深度不会限制对这些成员的访问，也不会增加访问开销。）例如：

```
void
NameQuery::
display_partial_solution( ostream &os )
{
    os << _name
       << " is found in "
       << (_solution ? _solution->size() : 0)
       << " lines of text\n";
}
```

对于继承而来的基类成员函数的访问也一样；我们调用它们，就好像它们是派生类的成

员，或者通过该类的一个对象：

```
NameQuery nq( "Frost" );

// 调用 NameQuery::eval()
nq.eval();

// 调用 Query::dysplay()
nq.display();
```

或者直接在成员函数中调用：

```
void
NameQuery::
match_count()
{
    if ( ! _solution )
        // 调用 Query::_vec2set()
        _solution = _vec2set( &_loc );
    return _solution->size();
}
```

在派生类中直接访问基类成员有一个例外——当基类成员名在派生类中被重用时。例如：

```
class Diffident {
public: // ...
protected:
    int _mumble;
    // ...
};

class Shy : public Diffident {
public: // ...
protected:
    // 隐藏了 Diffident::_mumble 的可视性
    string _mumble;
    // ...
};
```

在 Shy 的域中，非限定修饰地使用_mumble，总是被解析为 Shy 类的 string 成员_mumble。即使这样的用法是非法的，编译器也会这样解析。例如：

```
void
Shy::
turn_eyes_down()
{
    // ...
    _mumble = "excuse me"; // ok

    // 错误: int Diffident::_mumble 被隐藏
    _mumble = -1;
}
```

当编译器把此类用法标记为错误时，我们经常听到程序员抱怨编译器太愚蠢。尽管我们

能够明白程序员的意思，但是，编译器却需要一点帮助才行。为了用“已被派生类重用的名字”来访问基类的成员，我们必须用它的类域操作符限定修饰基类的成员。例如，下面是turn_eyes_down()的正确实现：

```
void
Shy::
turn_eyes_down()
{
    // ...
    _mumble = "excuse me";  // ok

    // ok: 限定修饰基类的实例
    Diffident::_mumble = -1;
}
```

C++语言初学者的一个常见的误解是，希望基类和派生类的成员函数构成一个重载函数集。例如：

```
class Diffident {
public:
    void mumble( int softness );
    // ...
};

class Shy : public Diffident {
public:
    // 隐藏了 Diffident::mumble 的可视性
    // 它们没有形成一对重载实例
    void mumble( string whatYaSay );
    void print( int soft, string words );

    // ...
};
```

但是，试图在派生类中调用基类实例却导致一个编译时刻错误。例如：

```
Shy simon;
// ok: Shy::mumble( string )
simon.mumble( "pardon me" );

// 错误: 期望第一个实参是 string 类型
// Diffident::mumble( int ) 不可见
simon.mumble( 2 );
```

虽然基类的成员可以被直接访问，但是它们仍然属于基类的域。一个名字的重载候选函数必须都出现在同一个域中。如果不是这样的话，那么，下面非虚拟成员函数 turn_aside()的两个实例：

```
class Diffident {
public:
    void turn_aside();
    // ...
};

class Shy : public Diffident {
```

```
public:
    // 隐藏了 Diffident::turn_aside() 的可视性
    void turn_aside();

    // ...
};
```

将导致重复定义错误，因为这两个实例的原型相同。它们不会出错，因为它们都属于各自被定义的类域中。

如果我们真的希望为基类和派生类的成员实例提供一个重载函数集合，该怎么办呢？我们需要在调用基类实例的派生类中写一个小的 inline 存根函数吗？显然，这样做实现了我们的目标，

```
class Shy : public Diffident {
public:
    // ok: 方法之一：为基类和派生类的成员
    // 提供一个重载函数集合
    void mumble( string whatYaSay );
    void mumble( int softness ) {
        Diffident::mumble( softness ); }

    // ...
};
```

但是在标准 C++中，这不是必需的。我们可以使用 using 声明（using declaration）获得同样的结果，如下所示：

```
class Shy : public Diffident {
public:
    // ok: 在标准 C++ 下，通过 using 声明
    // 创建了基类和派生类成员的重载集合
    void mumble( string whatYaSay );
    using Diffident::mumble;

    // ...
};
```

实际上，using 声明把基类中每个被命名的成员都引入到派生类的域中。现在，基类成员就可以进入到“与派生类中的成员函数名字相关”的重载实例集中。（针对一个成员函数的 using 声明不能指定参数表，只能指定成员函数名。这意味着，如果该函数在基类中被重载，则所有的重载实例都被加入到派生类类型的域中，我们不能只增加基类的重载成员函数集中的一个实例。）

C++新手程序员的另一个常见误解是访问基类的 protected 成员的范围。当我们写如下代码：

```
class Query {
public:
    const vector<location>* locations() const { return &_loc; }
    // ...
protected:
    vector<location> _loc;
    // ...
};
```

我们是在说从 Query 派生的类可以直接访问数据成员_loc，而程序的其余部分必须使用公有访问函数。但是，派生类访问其自身的基类子对象的 protected 数据成员_loc 又意味着什么？派生类不能访问另一个独立的基类对象的 protected 成员。例如：

```
bool
NameQuery::
compare( const Query *pquery )
{
    // ok: 自己的 Query 子对象的 protected 成员
    int myMatches = _loc.size();

    // 错误：没有 "直接访问另一个独立的 Query
    // 对象的 protected 成员" 的权利
    int itsMatches = pquery->_loc.size();
    return myMatches == itsMatches;
}
```

NameQuery 只能访问一个 Query 类对象的 protected 成员。它自己的 Query 子对象。这些 protectdd 成员在派生类中通过隐式 this 指针被访问。关于 this 指针的介绍见 13.4 节。。）解决这个编译时刻错误的最直接办法是利用公有成员函数 location()重写 compare()函数：

```
bool
NameQuery::
compare( const Query *pquery )
{
    // ok: 其 Query 子对象的 protected 成员
    int myMatches = _loc.size();

    // ok: 使用公有访问的方法
    int itsMatches = pquery->locations()->size();
    return myMatches == itsMatches;
}
```

但是，真正的问题在于不正确的设计。因为_loc 是 Query 基类的成员，compare()应该属于 Query 类的成员，而不是派生的 NameQuery 类的成员。通常，派生类和基类之间的成员访问问题，都可以通过把操作移到“包含该不可访问的成员的类”中来解决，正如这种情况所示。

这种形式的成员访问限制不适用于自己类的其他对象。例如：

```
bool
NameQuery::
compare( const NameQuery *pname )
{
    int myMatches = _loc.size();          // ok
    int itsMatches = pname->_loc.size();  // ok as well

    return myMatches == itsMatches;
}
```

派生类可以直接访问该类其他对象的 protected 基类成员，以及该类其他对象的 protected 和 private 成员。

考虑下列初始化，它用一个派生类 NameQuery 对象的地址初始化一个 Query 基类指针：

```
Query *pb = new NameQuery( "sprite" );
```

如果我们调用在 Query 基类中定义的虚拟函数，如：

```
pb->eval();   // 调用 NameQuery::eval()
```

则调用派生的 NameQuery 类实例。除了“在 Query 基类中被声明、并且在 NameQuery 派类中被改写”的虚拟函数之外，我们没有办法通过 pb 直接访问 NameQuery 的成员：

1．如果 Query 和 NameQuery 都声明了一个同名的非虚拟成员函数，则通过 pq 调用的总是 Query 的实例。

2．类似地，如果 Query 和 NameQuery 都声明了一个同名的数据成员，则通过 pq 总是访问 Query 的实例。

3．如果 NameQuery 引入了一个在 Query 中不存在的虚拟函数［比如 suffix()］，那么，试图通过 pq 调用它就会导致一个编译时刻错误:

```
// 错误: suffix() 不是 Query 的成员
pb->suffix();
```

4．类似地，如果我们试图通过 pq 访问 NameQuery 的数据成员或非虚拟成员函数，也会产生一个编译时刻错误:

```
// 错误: _name 不是 Query 的成员
pb->_name;
```

在这种情况下，即使对要访问的成员进行限定修饰，也不起作用：

```
// 错误: Query 没有 NameQuery 基类
pb->NameQuery::name();
```

在 C++中，基类指针只能访问在该类中被声明（或继承）的数据成员和成员函数，包括虚拟成员函数，而与它可能指向的实际对象无关。把一个成员函数声明为虚拟的，只推延了“在程序执行期间根据 pq 指向的实际类类型，对于要调用的实例的解析过程”。

虽然，这看起来似乎不太灵活，但是，它带来了两个重要的好处：

1．虚拟成员函数的执行从不会同为实际类类型不存在函数实例而失败。如果不存在某个适当的实例，则程序不能被编译。

2．虚拟机制可以被优化。虚拟函数调用通常不会比“通过指针间接调用函数”的开销更大（完全讨论见［LIPPMAN96a］）。（我们将在 17.5 节详细查看虚拟函数。）

Query 基类定义了一个静态数据成员_text_file：

```
static vector<string> *_text_file;
```

派生类 NameQuery 会创建第二个_text_file 实例（对于 NameQuery 类而言是惟一的）吗？不。所有派生类对象都引用这个相同的、单一的、共享的静态成员。不论从 Query 派生了多少类，_text_file 只存在一个实例。如果愿意的话，我们可以通过派生类对象用成员访问语法来访问它：

```
nameQueryObject._text_file; // ok
```

最后，如果一个派生类希望直接访问其基类的私有成员，则该基类必须显式地把派生类

声明为一个友元（friend）。例如：

```
class Query {
     friend class NameQuery;
public:
     // ...
};
```

现在，NameQuery 不但可以访问它自己的基类子对象的私有成员，而且还可以访问所有 Query 对象的私有和被保护的成员。

如果我们想从 NameQuery 派生一个 StringQuery 类，该怎么办呢？StringQuery 支持 NameQuery 的简写形式，以便用户不用写：

```
beautiful && fiery && bird
```

而可以简单地写：

```
"beautiful fiery bird"
```

StringQuery 继承了 NameQuery 与 Query 的友元关系吗？不。友元关系没有被继承。派生类没有成为“向它的基类授权友谊的类”的友元。如果这个派生类要求这种友元关系，则它必须被相应的类显式地授权。例如，StringQuery 没有对 Query 的访问特权。如果它需要特权访问，则 Query 必须显式地向它授权。

练习 17.6

已知如下基类和派生类定义：

```
class Base {
public:
     foo( int );
     // ...
protected:
     int _bar;
     double _foo_bar;
};

class Derived : public Base {
public:
     foo( string );
     bool bar( Base *pb );
     void foobar();
     // ...
protected:
     string _bar;
};
```

请指出下列代码段的错误以及修正方式：

```
(a)  Derived d; d.foo( 1024 );
(b)  void Derived::foobar() { _bar = 1024; }
(c)  bool Derived::bar( Base *pb )
     { return _foo_bar == pb->_foo_bar; }
```

17.4 基类和派生类的构造

派生类由一个或多个基类子对象以及派生类部分构成。例如，NameQuery 由一个 Query 子对象和一个 string 成员类对象构成。为了说明派生类构造函数的行为，我们引入一个内置数据成员：

```
class NameQuery : public Query {
public:
    // ...
protected:
    bool _present;
    string _name;
};
```

如果_present 被设置为 false，则表明 name 没有出现在文本中。

我们先考虑没有定义 NameQuery 类构造函数的情况。在这种情况下，当定义一个 NameQuery 对象时：

```
NameQuery nq;
```

先调用缺省的 Query 类构造函数，然后再调用缺省的 string 类构造函数（与成员类对象 _name 相对应）。_present 没有被初始化，这是一个潜在的程序错误根源。

为了初始化_present，我们可以定义缺省的 NameQuery 构造函数如下：

```
inline NameQuery::NameQuery(){ _present = false; }
```

现在，nq 的定义调用三个构造函数：缺省的 Query 基类构造函数、string 缺省构造函数以便初始化数据成员_name，以及 NameQuery 的缺省构造函数。

如果我们希望把一个实参传递给 Query 的基类构造函数，该怎么办呢？我们可能会怎样做呢？我们可以通过分析来回答这个问题。

为了把一个或多个实参传递给成员类对象的构造函数，我们通过成员初始化表来实现（我们也可以用成员初始化表来初始化非类的数据成员——见 14.5 节的讨论）。例如：

```
inline NameQuery::
NameQuery( const string &name )
    : _name( name ), _present( false )
{}
```

为了向基类构造函数传递一个或多个参数，我们也使用成员初始化表。在下面的例子中，我们向 string 构造函数传递实参 name，向 Query 基类构造函数传递由 ploc 指向的对象：

```
inline
NameQuery::
NameQuery( const string &name,
           vector<location> *ploc )
    : _name( name ), Query( *ploc ), _present( true )
{}
```

尽管 Query 被放在成员初始化表中的第一位，但是它总是在“与_name 相关联的 string 构造函数”之前被调用。构造函数的调用顺序总是如下：

1．基类构造函数。如果有多个基类，则构造函数的调用顺序是某类在类派生表中出现的顺序，而不是它们在成员初始化表中的顺序。（我们将在第 18 章讨论多继承。）

2．成员类对象构造函数。如果有多个成员类对象，则构造函数的调用顺序是对象在类中被声明的顺序，而不是它们出现在成员初始化表中的顺序（详细讨论见 14.5 节）。

3．派生类构造函数。

作为一般规则，派生类构造函数应该不能直接向一个基类数据成员赋值，而是把值传递给适当的基类构造函数。否则，两个类的实现变成**紧耦合的**（tightly coupled），将更加难于正确地修改或扩展基类的实现。（基类设计者的责任是提供一组适当的基类构造函数。）

在本节余下部分，我们将按顺序讨论 Query 基类构造函数以及四个派生类构造函数的设计。然后，再简要考虑另一个两层以上的 Query 层次结构的设计。我们将以类析构函数的讨论结束本节。

17.4.1 基类构造函数

我们的 Query 类声明了两个非静态数据成员_solution 和_loc：

```
class Query {
public:
    // ...
    protected:
    set<short> *_solution;
vector<location> _loc;
    // ...
};
```

缺省的 Query 类构造函数只需要显式地初始化_solution。而缺省的 vector 构造函数被自动调用，以便将_loc 初始化。下面是它的实现：

```
inline Query::Query(): _solution( 0 ) {}
```

我们还需要定义第二个 Query 类构造函数，它有一个指向 vector<location>的引用作为参数：

```
inline
Query::
Query( const vector< location > &loc )
    : _solution( 0 ), _loc( loc )
{}
```

对于此第二个 Query 构造函数，只有当 NameQuery 对象代表了文本中出现的单词时，它才在 NameQuery 构造函数中被调用。在这种情况下，与单词相关联的、预先被计算好的位置向量（location vector）被传递过来。另外三个派生类型在其相关的成员函数 eval()中计算它们的位置向量。（我们将在下一小节看到这样的一个例子，成员函数 eval()的实现将在 17.5 节讨论虚拟函数时给出。）

现在的问题是，应该把构造函数声明成什么样的访问级别？我们不希望把它们声明为公有的，因为 Query 类对象只想在程序中作为“其派生子类型对象中的子对象”而存在。我们通过将构造函数声明为 protected 而不是 public 来指出这一点。

```
class Query {
```

```
public:
    // ...
protected:
    Query();
    // ...
};
```

这第二个 Query 构造函数在一个更为严格的条件下被调用：它不但应该只构造一个 Query 子对象，而且它还应该只构造一个 NameQuery 对象的 Query 子对象。我们可以通过把这第二个构造函数声明为 private，并把 NameQuery 声明为 Query 类的友元来保证这一点。正如在上节讨论的，派生类只能访问其某类的 public 和 protected 成员。现在，在 OrQuery、AndQuery 和 NotQuery 中任何企图调用第二个构造函数都会导致编译时刻错误。）

```
class Query {
    friend class NameQuery;
public:
    // ...
protected:
    Query();
    // ...
private:
    explicit Query( const vector<location>& );
};
```

（有人可能会对这第二个构造函数提出异议，主张在 NameQuery 成员函数 eval()中填充 _loc 更为合适。但是，这种双构造函数的设计，对于说明基类的构造函数用法而言，能较好地体现我们的意图。）

17.4.2 派生类构造函数

NameQuery 也定义了两个构造函数。它们是公有的，因为 NameQuery 对象希望能被定义在我们的应用程序中。如下所示：

```
class NameQuery : public Query {
public:
    explicit NameQuery( const string& );
    NameQuery( const string&, vector<location>* );
    // ...
protected:
    // ...
};
```

单参数的构造函数接受一个 string 参数。它被传递给 string 构造函数，该构造函数被用来初始化 string 数据成员_name。缺省的 Query 基类构造函数被隐式调用：

```
inline
NameQuery::
NameQuery( const string &name )
    // Query::Query() 被隐式调用
    : _name( name )
{}
```

双参数构造函数也接受了一个 string 参数，此外它还接受一个指向 vector<location>的指

针类型的另一个参数。它被传递给私有的 Query 基类构造函数。注意在这里我们不再把 _present 视为 NameQuery 的数据成员）：

```
inline
NameQuery::
NameQuery( const string &name, vector<location> *ploc )
     : _name( name ), Query( *ploc )
{}
```

下面是它们的用法：

```
string title( "Alice" );
NameQuery *pname;

// 看 "Alice" 是否出现在单词文本映射表中，
// 如果是，则获取其相关的位置向量
if ( vector<location> *ploc = retrieve_location( title ))
     pname = new NameQuery( title, ploc );
else pname = new NameQuery( title );
```

NotQuery、OrQuery 和 AndQuery 都如下定义了一个构造函数，其中隐式地调用了 Query 基类的缺省构造函数：

```
inline NotQuery::
NotQuery( Query *op = 0 ) : _op( op ) {}

inline OrQuery::
OrQuery( Query *lop = 0, Query *rop = 0 )
     : _lop( lop ), _rop( rop )
{}

inline AndQuery::
AndQuery( Query *lop = 0, Query *rop = 0 )
     : _lop( lop ), _rop( rop )
{}
```

在 17.7 节中，我们将生成独立的派生类对象来表示每一个用户查询。

17.4.3 另外一个类层次结构

虽然我们的 Query 类层次结构已经设计得很充分了，但是，它不是唯一可能的设计。例如，因为 AndQuery 和 OrQuery 类都支持一个二元操作，所以，在这两个类之间有一些重复。我们可以把这两个类共同的数据成员和成员函数抽取到一个抽象的 BinaryQuery 基类中。图 17.2 给出了 Query 层次结构的新子树。

BinaryQuery 也是一个抽象基类——即，该类的实际对象不会出现在应用程序中。同为 BinaryQuery 类不存在有意义的 eval()实现，所以我们不为它提供 eval()虚拟函数的定义。在 BinaryQuery 类中，eval()的纯虚拟实例也是有效的（我们将在 17.5 节详细查看纯虚拟函数）。

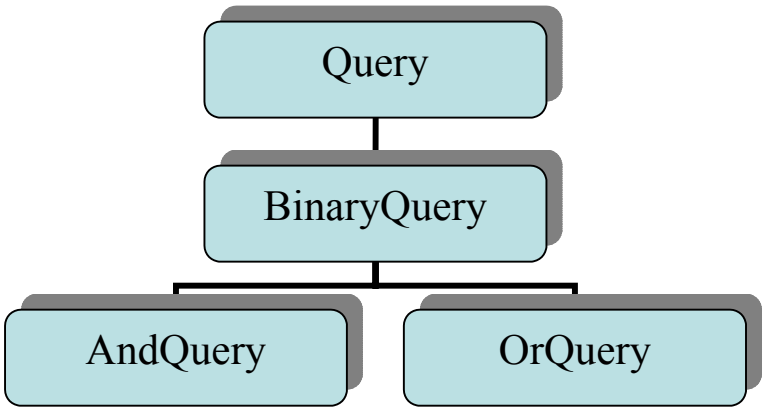


图 17.2 另一个类层次结构

由于两个访问成员函数 lop()和 rop()，对于两个派生类是公共的。所以把它们提出来放到 BinaryQuery 类中，它们被定义为非静态内联成员函数。类似地，本来由派生类声明的两个数据成员 lop 和_rop 被提升到 BinaryQuery 类中，它们被声明为 protected 非静态数据成员。而两个派生类的公有构造函数被组合到一个 protected 的 BinaryQuery 构造函数中：

```
class BinaryQuery : public Query {
public:
    const Query *lop() { return _lop; }
    const Query *rop() { return _rop; }
protected:
    BinaryQuery( Query *lop, Query *rop )
        : _lop( lop ), _rop( rop )
    {}
    Query *_lop;
    Query *_rop;
};
```

现在，这两个派生类看起来好像只需提供适当的 eval()实例：

```
// 喔！这些类定义不正确
class OrQuery : public BinaryQuery {
public:
    virtual void eval();
};

class AndQuery : public BinaryQuery {
public:
    virtual void eval();
};
```

即使我们已经定义了 eval 实例，但它们仍是不完整的。如果我们编译这两个类定义，令人吃惊的是，它们居然能无错误地通过。如果试图定义一个实际的类对象，比如：

```
// 错误：缺少 AndQuery 类构造函数
AndQuery proust( new NameQuery( "marcel" ),
```

```
        new NameQuery( "proust " ));
```

则 proust 的定义被标记为错误：告诉我们 AndQuery 类缺少一个支持双参数的构造函数。

我们已经假设 AndQuery 和 OrQuery 都继承了成员访问函数 lop()和 rop()，并以同样的方式继承了 BinaryQuery 的构造函数。但是，实际上它们没有继承。派生类并不继承基类的构造函数。（原因是，这样太容易引入“未初始化派生类成员”错误。例如，假设我们后来为 AndQuery 增加了另外一个非类的数据成员。那么，继承而来的基类构造函数就不再足以初始化派生类 AndQuery。但是，增加新成员的程序员可能意识不到这一点。然而，该错误不会在 AndQuery 对象的构造过程中被揭示出来，而是在这个对象的某个使用点上才被暴露出来。实践证明，这种错误很难跟踪。继承基类的 new 和 delete 操作符的重载实例，有时也会导致这种问题。）

每个派生类都必须提供自己的构造函数集。在 AndQuery 和 OrQuery 类的情况下，构造函数被当作一个接口，用来向 BinaryQuery 构造函数传递两个操作数。下面是正确的实现：

```
// ok: 这些类定义是正确的
class OrQuery : public BinaryQuery {
public:
    OrQuery( Query *lop, Query *rop )
        : BinaryQuery( lop, rop ) {}

    virtual void eval();
};

class AndQuery : public BinaryQuery {
public:
    AndQuery( Query *lop, Query *rop )
        : BinaryQuery( lop, rop ) {}

    virtual void eval();
};
```

如果再次看一看图 17.2，我们会发现 BinaryQuery 是 AndQuery 和 OrQuery 的直接基类，Query 是 BinaryQuery 的直接基类，而 Query 是 AndQuery 和 OrQuery 的非直接基类。

派生类构造函数只能合法地调用其直接基类的构造函数（虚拟继承为这条规则提供了一个特例，正如它对许多其他规则所做的一样，见 18.5 节）。例如，AndQuery 在其成员初始化表中调用 Query 构造函数就是错误的。

现在，AndQuery 或 OrQuery 类对象的定义将引起对于下列三个构造函数的调用：非直接基类 Query 的构造函数、直接基类 BinaryQuery 的构造函数，以及 AndQuery 或 OrQuery 派生类构造函数（基类构造函数被调用的顺序反映了派生类继承层次结构中深度优先的遍历过程）。但是，附加的 BinaryQuery 类派生对于性能的影响可以被忽略，同为我们已经把它定义为 inline。

因为修改之后的层次结构保留了原始设计的公有接口，所以这样的改变不会影响到原来使用老的层次结构的代码。虽然用户的源代码无需修改，但是，它需要用新的类层次结构的定义重新编译。然而，因为要求重新编译一个完整的系统，所以某些用户可能会不愿意加快向新设计的迁移步伐。

17.4.4 迟缓型错误检测（Lazy error detection）

C++新手常常很吃惊，AndQuery 和 OrQuery 的非法定义可以无错误地通过编译（它们都缺少必要的构造函数声明）。也许我们并不打算定义一个实际的 AndQuery 对象，但可能已经把修改后的类层次结构递交出去了，并且没有改正我们的错误。接下来会怎么样？考虑：

1．如果一个错误在声明点上被标记出来，则我们不能继续编译程序，有到错误被改正为止。但是，如果发生冲突的声明是库的一部分，而我们不能访问它的源码，则解决这样的冲实可能真的不是小事一桩。而且，我们可能永远不会有机会触发应用程序中的错误，以至于虽然这个声明代表了一个潜在的错误，但是该错误却从不会在我们的代码中被识别出来。

2．另一方面，如果直到一个使用点上，该错误才被标记出来，则我们的代码可能充满了各种未触发的语言错误，而不小心的程序员可能随时激活它们。在这种策略下，成功编译代码并不能确保它没有语义错误，只能保证程序没有违反语言的语义规则。

在使用点上产生一个错误消息是一种**迟缓型计算**（lazy evaluation）的形式，是提高程序性能的常见设计策略。它常常被应用在昂贵资源的初始化和分配上，直到真正需要这些资源时才分配或者初始化。如果这些资源永远也没有被真正用到，则我们就可以节省下不必要的运行开销。如果需要这些资源，但不是一次需要全部资源，则我们可以分散程序的初始化开销。

在 C++中，在处理重载函数、模板及类层次结构时，通常会产生一种潜在的组合错误，这种错误（往往会在使用点而不是声明点上被揭示出来。无论你是否相信这是一个正确的策略（实践中，我们认为它是正确的。在组合多个组件时，要想解决每一个潜在的错误，并不符合生产规律），它确实是正在使用的策略。这意味着我们必须小心谨慎地测试自己的代码，以便找到并解决潜在的错误。在组合两个或多个大型组件时，少量潜在的错误是可以被较受的。但是，在单个组件中，如 Query 类层次结构中，则往往是不可接受的

17.4.5 析构函数

当派生类对象的生命期结束时，如果派生类和基类析构函数都被定义了，则它们会被自动调用，并且所有成员类对象的析构函数也会被自动调用。例如，已知下列 NameQuery 类对象。

```
NameQuery nq( "hyperion" );
```

析构函数的调用顺序是：1）NameQuery 类析构函数；2）数据成员_name 的 string 析构函数，以及 3）Query 基类析构函数。更一般的情况下，派生类的析构函数调用顺序与它的构造函数调用顺序相反。

下面是 Query 基类和派生类析构函数（它们分别被声明为相应类的 public 成员）：

```
inline Query::
~Query(){ delete _solution; }

inline NotQuery::
~NotQuery(){ delete _op; }

inline OrQuery::
~OrQuery(){ delete _lop; delete _rop; }
```

```
inline AndQuery::
~AndQuery(){ delete _lop; delete _rop; }
```

这里有两点需要注意：1）我们没有提供显式的 NameQuery 析构函数。为什么？因为我们并不需要提供程序层次上的清除工作。Query 基类析构函数和_name 的 string 析构函数会被自动调用。2）在派生类析构函数中，delete 表达式被应用到 Query*指针上。但是我们想调用的不是 Query 的析构函数。我们希望调用指针所指的实际对象类型的析构函数。为做到这一点，我们必须把 Query 基类的析构函数声明为虚拟的。我们将在下一节查看虚拟析构函数，以及一般的虚拟函数。

还有最后一件事需要提醒。我们的实现中有一个隐含的假设是，在 NotQuery、OrQuery 和 AndQuery 类对象中引用的操作数是被分配在堆上的，这也正是我们在相应的析构函数中对每个操作数调用 delete 操作符的原因。但是，这不是语言层次上强制的假设。语言本身并没有区分堆（heap）上的地址与非堆（non-heap）上的地址。因此，我们的实现在某种层次上是不安全的。

正如我们将在 17.7 节所看到的，把 Query 层次结构的分配和构造过程封装在一个 UserQuery 管理类中，使我们有足够的信心相信，至少对于本书而言，我们的设想没有违例。但是，作为一般目的的库，我们还需要进一步的保证。程序层次上的有效策略是，为层次结构中的类重载 new 和 delete 操作符。一种可能的程序层次上的策略如下：new 操作符把对象标记为“在堆中分配”，然后再用 new 表达式分配该对象。而 delete 操作符检查这个标记是否存在。如果存在，则对操作数应用 delete 表达式。

练习 17.7

请为 17.1 节的练习 17.2 选择的类层次结构定义基类和派生类构造函数和析构函数。

练习 17.8

请重新实现 OrQuery 类，使其从一个抽象的 UnaryQuery 类派生。

练习 17.9

下列类定义有什么错？

```
class Object {
public:
     virtual ~Object();
     virtual string isA();
protected:
     string _isA;
private:
     Object( string s ) : _isA( s ){}
};
```

练习 17.10

给出下列基类定义：

```
class ConcreteBase {
public:
     explicit ConcreteBase( int );
     virtual ostream& print( ostream& );
     virtual ~Base();
     static int object_count();
protected:
     int _id;
     static int _object_count;
};
```

下列代码错在哪里？

```
(a)  class C1 : public ConcreteBase {
     public:
          C1( int val )
               : _id( _object_count++ ){}
          // ...
     };
(b)  class C2 : public C1 {
     public:
          C2( int val )
               : ConcreteBase( val ), C1( val ){}
          // ...
     };
(c)  class C3 : public C2 {
     public:
          C3( int val )
               : C2( val ), _object_count( val ){}
          // ...
     };
(d)  class C4 : public ConcreteBase {
     public:
          C4( int val )
               : ConcreteBase( _id+val ){}
          // ...
     };
```

练习 17.11

在 C++的原始定义中，成员初始化表中的初始化顺序决定了构造函数的调用顺序。直到 1986 年左右才改成了现在的语言规则。你认为改变原始语言规则的原因是什么？

17.5 基类和派生类虚拟函数

缺省情况下，类的成员函数是非虚拟的（nonvirtual）。当一个成员函数为非虚拟的时候，通过一个类对象（指针或引用）而被调用的该成员函数，就是该类对象的静态类型中定义的成员函数。例如：

```
void Query::display( Query *pb )
{
    set<short> *ps = pb->solutions();
    // ...
    display();
}
```

pb 的静态类型是 Query*。非虚拟的 solutions()调用的是 Query 的成员函数。而非虚拟函数 display()通过隐式 this 指针被调用。this 指针的静态类型也是 Query*，所以被调用的函数是 Query 的成员函数。

要把对象声明为虚拟的，我们只需指定关键字 virtual：

```
class Query {
public:
    virtual ostream& print( ostream& = cout ) const;
    // ...
};
```

当成员函数是虚拟的时候，通过一个类对象（指针或引用）而被调用的该成员函数，是在该类对象的动态类型中被定义的成员函数。但是，正如所发生的，一个类对象的静态和动态类型是相同的。所以，虚拟函数机制只在使用指针和引用时才会如预期般地起作用。

只有在通过基类指针或引用间接指向派生类子类型时，多态性才会起作用。使用基类对象并不会保留派生类的类型身份。例如，考虑下列代码段：

```
NameQuery nq( "lilacs" );

// ok: 但是 nq 被 "切割" 成一个 Query 子对象
Query qobject = nq;
```

用 nq 初始化 qobject 是合法的：qobject 现在等于 Query 基类子类型 nq。但是，qobject 并不是一个 NameQuery 对象。在初始化 qobject 之前，nq 的 NameQuery 部分被“切除”。在 C++的面向对象程序设计中，具有讽刺意味的是，我们必须使用指针以引用而不是对象来支持它（指面向对象程序设计）。例如，在下面的代码段中：

```
void print( Query object,
            const Query *pointer,
            const Query &reference )
{
    // 直到运行时刻才能确定
    // 调用哪个 print() 实例
    pointer->print();
    reference.print();

    // 总是调用 Query::print()
    object.print();
}

int main()
{
    NameQuery firebird( "firebird" );
    print( firebird, &firebird, firebird );
}
```

通过 pointer 和 reference 的调用被解析为它们的动态类型。在这个例子中，它们都调用 NameQuery::print()。而通过 object 的调用则总是调用 Query::print()。我们将在 18.6.2 节中看到一个例子，它说明了这种切除造成的影响。）

在以下各小节中，我们将通过查看各种实例的实现，来说明虚拟函数的定义和用法。每一个虚拟成员函数都说明了面向对象设计中的一个不同方面。

17.5.1 虚拟的输入 / 输出

我们要给出的第一个虚拟操作是向标准输出或文件打印一个查询：

```
ostream& print( ostream &os = cout ) const;
```

我们必须把 print()声明为虚拟的，因为每个 print()实现都依赖子类型。但是，我们必须能够通过 Query*指针来调用。例如，AndQuery 的 print()看起来是这样的：

```
ostream&
AndQuery::print( ostream &os ) const
{
    _lop->print( os );

    os << " && ";

    _rop->print( os );
}
```

我们必须把 print()声明为 Query 抽象类的虚拟函数。否则，我们就不能通过“AndQuery、OrQuery 和 NotQuery 类的 Query*操作数数据成员”调用 print()。但是，Query 基类的 print()没有意义。现在，我们只是简单地把它定义为一个空函数。（以后，我们会将它重新定义为纯虚拟函数。）

```
class Query {
public:
    virtual ostream& print( ostream &os=cout ) const {}
    // ...
};
```

第一次引入虚拟函数的基类时，必须在类声明中指定 virtual 关键字。如果定义被放在类的外面，则不能再次指定关键字 virtual。例如，下面的 print()定义将导致编译时刻错误：

```
// 错误：关键字 virtual 只能出现在类定义中
virtual ostream& Query::print( ostream& ) const { ... }
```

正确的定义必须不包括关键字 virtual。

引入虚拟函数的类必须定义它，或者把它声明为纯虚拟函数（再次说明，我们现在只是把它定义为空函数）。如果派生类可以提供自己的实例，那么此实例将成为该派生类的活动实例，或各派生类也可以继承基类的活动实例。如果派生类定义了实例，则称之为改写（override，也译为改变或者覆盖）了基类的实例。

在我们开始讨论四个派生类实现之前，首先需要考虑在查询中出现的括号。例如，已知查询：

```
fiery && bird || shyly
```

是用户在查找单词对：

```
fiery bird
```

或单个副词：

```
shyly
```

的任意次出现。

另一方面，查询：

```
fiery && ( bird || hair )
```

查找单词对：

```
fiery bird
```

或：

```
fiery hair
```

的任意次出现。

如果我们的 print()实现不能重现原来的括号，那么，它们对于用户的价值就会小很多。为了追踪记录必要的左右括号，我们对 Query 抽象基类进行改进，引入了一对非静态数据成员，以及相应的访问支持函数（这种通过改进引入成员，是一个类层次结构进化的正常行为）：

```
class Query {
public:
    // ...

    // 设置 _lparen 和 _rparen
    void lparen( short lp ) { _lparen = lp; }
    void rparen( short rp ) { _rparen = rp; }

    // 获取 _lparen 和 _rparen 的值
    short lparen() { return _lparen; }
    short rparen() { return _rparen; }

    // 打印左右括号
    void print_lparen( short cnt, ostream& os ) const;
    void print_rparen( short cnt, ostream& os ) const;
protected:
    // 拥有左右括号的数目
    short _lparen;
    short _rparen;

    // ...
};
```

_lparen 表示一个特定的对象应该输出的左括号数，_paren 表示右括号的数目。（在 17.7 节中，我们将了解怎样计算这些数目，以及怎样对这两个成员赋值。）下面是一个例子，它处理高度“括号化”的查询：

```
==> ( untamed || ( fiery || ( shyly ) ) )
evaluate word: untamed
_lparen: 1
_rparen: 0
```

```
evaluate Or
_lparen: 0
_rparen: 0

evaluate word: fiery
_lparen: 1
_rparen: 0

evaluate Or
_lparen: 0
_rparen: 0

evaluate word: shyly
_lparen: 1
_rparen: 0

evaluate right parens:
_rparen: 3

( untamed ( 1 ) lines match
( fiery ( 1 ) lines match
( shyly ( 1 ) lines match
( fiery || ( shyly ( 2 ) lines match [30]
( untamed || ( fiery || ( shyly ))) ( 3 ) lines match
Requested query: ( untamed || ( fiery || ( shyly )))

( 3 ) like a fiery bird in flight. A beautiful fiery bird, he tells her,
( 4 ) magical but untamed. "Daddy, shush, there is no such thing,"
( 6 ) Shyly, she asks, "I mean, Daddy, is there?"
```

下面是 NameQuery 的实现：

```
ostream&
NameQuery::
print( ostream &os ) const
{
    if ( _lparen )
        print_lparen( _lparen, os );
    os << _name;

    if ( _rparen )
        print_rparen( _rparen, os );
    return os;
}
```

下面是它的声明：

```
class NameQuery : public Query {
public:
    virtual ostream& print( ostream &os ) const;
    // ...
};
```

[30] 直到 OrQuery 显示其部分结果后，右括号才被识别出来。

为了使虚拟函数的派生类实例能够改写其基类的活动实例，它的原型必须与基类完全匹配。例如，如果我们去掉了 const，或声明了第二个参数，则 NameQuery 实例将不能改写活动的基类实例。此外，返回值也必须相同，但有一个特例：派生类实例的返回值可以是基类实例返回类型的公有派生类类型。例如，如果基类实例返回一个 Query*，则派生类实例可以返回 NameQuery*。[在实现 clone()函数时，我们将看到一个例子，以说明可以这样做的原因。]下面是 NotQuery 的声明以及 print()的实现：

```
class NotQuery : public Query {
public:
    virtual ostream& print( ostream &os ) const;
    // ...
};

ostream&
NotQuery::
print( ostream &os ) const
{
    os << " ! ";
    if ( _lparen )
        print_lparen( _lparen, os );
    _op->print( os );
    if ( _rparen )
        print_rparen( _rparen, os );
    return os;
}
```

当然，通过 op 对 print()的调用是虚拟调用。

AndQuery 和 OrQuery 的 print()声明和实现本质上是类似的，我们只显示 AndQuery 的 print()：

```
class AndQuery : public Query {
public:
    virtual ostream& print( ostream &os ) const;
    // ...
};

ostream&
AndQuery::
print( ostream &os ) const
{
    if ( _lparen )
        print_lparen( _lparen, os );

    _lop->print( os );
    os << " && ";
    _rop->print( os );

    if ( _rparen )
        print_rparen( _rparen, os );

    return os;
}
```

这些虚拟函数 print()的实现使我们能够向 ostream 或者“从 ostream 派生的任何 stream”输出任何 Query 子类型。例如：

```
cout << "The query request is ";
Query *pq = retrieveQuery();
pq->print( cout );
```

虽然这种设施很有用，但还是不够。我们还希望能够用 iostream 输出操作符，输出任何当前的或者将来从 Query 派生的类类型。例如：

```
Query *pq = retrieveQuery();

cout << "The query request "
     << *pq
     << " generated the following results:\n";
```

由于输出操作符已经是 ostream 类的成员，所以我们不能直接提供一个虚拟的输出操作符，相反，我们必须提供一个间接的虚拟函数，如下所示：

```
inline ostream&
operator<<( ostream &os, const Query &q )
{
     // print() 的虚拟调用
     return q.print( os );
}
```

当我们写如下代码时：

```
AndQuery query;

// 设置 query ...
cout << query << endl;
```

我们的 ostream 操作符就被调用，然后再调用：

```
q.print( os )
```

q 被绑定到 AndQuery 类对象 query 上，而 os 则被绑定到 cout 上。如果我们写：

```
NameQuery query2( "Salinger" );
cout << query2 << endl;
```

则 NameQuery 的 print()实例被调用。下面的调用：

```
Query *pquery = retrieveQuery();
cout << *pquery *Lt; endl;
```

调用了在程序该执行点上与 pquery 指向的对象相关联的 print()实例。

17.5.2 纯虚拟函数

为了支持用户的查询，我们主要的编码任务就是实现与每个查询操作符相关联的、与具体类型有关的各种操作。为了做到这一点，我们定义了四个实体类类型：NameQuery 及 OrQuery 等等。但是，我们主要的设计任务是在类型无关的接口背后封装每个实体查询的处理。这使我们能够提供应用程序的核心，而其编码不受某些特殊类型的增删影响。

为了实现这一点，我们定义了抽象 Query 类类型。我们并没有编写用户可能指定的查询类型。而是给出了可以应用在所有查询类型上的抽象动作。例如：

```
void doit_and_bedone( vector< Query* > *pvec )
{
    vector<Query*>::iterator
        it = pvec->begin(),
        end_it = pvec->end();
    for ( ; it != end_it; ++it )
    {
        Query *pq = *it;
        cout << "processing " << *pq << endl;
        pq->eval();
        pq->display();
        delete pq;
    }
}
```

在理论上，这种做法支持将来无限增加的查询类型，而无需修改或者重新编译我们的系统核心——只要抽象基类的公有接口足够支持每个新的查询类型。

提供 Query 类的公有接口的目的是，定义一组“足够支持当前和将来的所有查询类型”的操作。在实践中，这是个相当高的要求，我们不可能对将来所有查询操作的类型都做保证。另一方面，为那些已知的类型提供公共的接口肯定是可以做到的。对于任何超出这个公共接口的东西，我们最好以怀疑的眼光去审视它们。

因为 Query 是一个抽象类，它不能真正出现在我们的应用程序中，所以我们不能为它的虚拟函数提供有意义的实现。它们只是被用作占位符，被后来的派生子类型改写，它们不希望被直接调用。

C++语言为我们提供了一种语法结构，通过它可以指明，一个虚拟函数只是提供了一个可被子类型改写的接口。但是，它本身并不能通过虚拟机制被调用。这就是纯虚拟函数（pure virtual function）。纯虚拟函数的声明如下所示：

```
class Query {
public:
    // 声明纯虚拟函数
    virtual ostream& print( ostream&=cout ) const = 0;

    // ...
};
```

这里函数声明后面紧跟赋值 0。

包含（或继承）一个或多个纯虚拟函数的类被编译器识别为抽象基类。试图创建一个抽象基类的独立类对象会导致编译时刻错误。（类似地，通过虚拟机制调用纯虚拟函数也是错误的。）例如：

```
// Query 声明了纯虚拟函数
// 所以，程序员不能创建独立的 Query 类对象
// ok: NameQuery 中的 Query 子对象
Query *pq = new NameQuery( "Nostromo" );

// 错误: new 表达式分配 Query 对象
Query *pq2 = new Query;
```

抽象基类只能作为子对象出现在后续的派生类中，这些正是我们所期望的 Query 基类的

语义。

17.5.3 虚拟函数的静态调用

当用类域操作符调用虚拟函数时，我们改变了虚拟机制，使得虚拟函数在编译时刻被静态解析。例如，假设我们已经为 Query 层次结构的所有基类和派生类定义了虚拟函数 isA()：

```
Query *pquery = new NameQuery( "dumbo" );

// 通过虚拟机制动态调用 isA()
// 调用 NameQuery::isA() 实例
pquery->isA();

// 在编译时刻静态调用 isA
// 调用 Query::isA 实例
pquery->Query::isA();
```

Query::isA()的显式调用在编译时刻被解析为基类 Query 的实例，即使 pquery 刚好指向一个 NameQuery 对象。

为什么我们要改变虚拟机制呢？常常是为了效率。在一个派生类虚拟函数中，有时需要调用基类的实例来完成“已经在基类和派生类实例之间被抽取出来”的操作。例如，Camera 虚拟 display()函数可能显示对所有 Camera 公共的信息。PerspectiveCamera 的 display()实例只显示对于 PerspectiveCamera 惟一的信息。我们不是在 PerspectiveCamera 的 display()实现中复制 Camera 的操作，而是调用 Camera 的实例。我们知道自己希望调用哪个实例，所以，如果我们可以改变它，就不需要通过虚拟机制。而且，如果 Camera 的实例被声明为 inline，则编译时刻调用它将引起内联展开。

以下是期望改变虚拟机制的另外一种解释。它也向我们展示了纯虚拟函数的另一方面，C++的新手常常感觉它违背了直觉。

AndQuery 和 OrQuery 的 print()实例几乎完全相同，只有表示操作符的文字字符中有所不同。我们并不为它们提供两个不同的实例，而是实现一个可供两者共享的单个实例。为了做到这一点，我们再次定义抽象基类 BinaryQuery、AndQuery 和 OrQuery 从它派生。BinaryQuery 定义了两个操作数，以及一个额外的 string 数据成员来存放操作符的值。因为它是一个抽象类，所以我们把 print()声明为纯虚拟函数：

```
class BinaryQuery : public Query {
public:
    BinaryQuery( Query *lop, Query *rop, string oper )
        : _lop(lop), _rop(rop), _oper(oper){}
    ~BinaryQuery() { delete _lop; delete _rop; }
    ostream &print( ostream& =cout ) const = 0;
protected:
    Query *_lop;
    Query *_rop;
    string _oper;
};
```

下面是 BinaryQuery 的 print()实例，由派生类 AndQuery 和 OrQuery 调用：

```
inline ostream&
BinaryQuery::
print( ostream &os ) const
{
    if ( _lparen )
        print_lparen( _lparen, os );

    _lop->print( os );
    os << ' ' << _oper << ' ';
    _rop->print( os );

    if ( _rparen )
        print_rparen( _rparen, os );
    return os;
}
```

我们似乎为自己引入了一个矛盾。一方面，我们认为有必要把 print()声明为纯虚拟函数，以告诉编译器 BinaryQuery 是一个抽象基类。现在可以保证在应用程序中不会定义 BinaryQuery 的独立对象。

另一方面，我们必须定义 BinaryQuery 的虚拟 print()实例，并通过 AndQuery 和 OrQuery 类的对象调用它。

与许多矛盾一样，我们忽略了一个重要的信息：纯虚拟函数，虽然它可以通过虚拟机制被调用。但也可以被静态调用。例如：

```
inline ostream&
AndQuery::
print( ostream &os ) const
{
    // ok: 抑制虚拟机制
    // 静态调用 BinaryQuery::print
    BinaryQuery::print( os );
}
```

17.5.4 虚拟函数和缺省实参

考虑下面简单的类层次结构：

```
#include <iostream>

class base {
public:
    virtual int foo( int ival = 1024 ) {
        cout < "base::foo() -- ival: " < ival < endl;
        return ival;
    }
    // ...
};

class derived : public base {
public:
    virtual int foo( int ival = 2048 ) {
```

```
            cout << "derived::foo() -- ival: " << ival << endl;
            return ival;
        }
        // ...
};
```

类设计者的目的是，如果不带实参而调用 foo()的基类实例，则应该传递给它缺省实参 1024。例如：

```
base b;
base *pb = &b;

// 调用 base::foo( int )
// 意图是，应该返回 1024
pb->foo();
```

类似地，若程序员的目标是 foo()的派生类实例，如果不带实参调用，则应该传递给缺省实参 2048。例如：

```
derived d;
base *pb = &d;

// 调用 derived::foo( int )
// 意图是，应该返回 2048
pb->foo();
```

正如所发生的，这不是 C++虚拟机制的语义行为。例如，下面的小程序使用了我们的类层次结构：

```
int main()
{
    derived *pd = new derived;
    base *pb = pd;

    int val = pb->foo();
    cout << "main() : val through base: "
         << val << endl;

    val = pd->foo();
    cout << "main() : val through derived: "
         << val << endl;
}
```

编译并运行它，程序产生下列输出：

```
derived::foo() -- ival: 1024
main() : val through base: 1024

derived::foo() -- ival: 2048
main() : val through derived: 2048
```

在这两个调用中，foo()的派生类实例被正确调用。这是因为，foo()调用的真正实例是在运行时刻根据 pd 和 pb 指向的实际类型决定的。然而，传递给 foo()的缺省实参不是在运行时刻决定的，而是在编译时刻根据被调用函数的对象的类型决定的。当通过 pb 调用 foo()时，

缺省实参中 base::foo()的声明决定，为 1024。当通过 pd 调用 foo()时，缺省实参由 derived::foo()的声明决定，为 2048。

如果通过基类指针或引用调用派生类实例，则传递给它的缺省实参是由基类指定的，那么为什么还要在派生类实例中指定缺省实参呢？

我们可能希望有不同的缺省实参，不是根据被调用函数 foo()的特定子类型，而是根据“调用该函数时所使用的指针或引用”的类型。例如，1024 和 2048 可能代表不同的图像大小。如果希望产生更详细的图像，则可以通过基类调用 foo()。如果希望更好的分辨率，也通过派生类调用 foo()。

但是，如果我们确实希望传递给 foo()的实际缺省实参是根据被调用函数的实际实例而决定的，那么该怎么办呢？不幸的是，虚拟机制不直接支持这种行为。一种程序设计的方案是指定一个缺省实参，它可以被用来指示“用户并没有传递相应的值”。而真正的缺省实参被声明为函数中的局部值，如果没有传递进来显式的值，则使用该局部值。例如：

```
void
base::
foo( int ival = base_default_value )
{
    int real_default_value = 1024;

    if ( ival == base_default_value )
        ival = real_default_value;
    // ...
}
```

这里的 base_default_value 对于整个层次结构都是一致的，如果它出现了，则表示用户没有提供一个显式的值。派生类实例以类似的方式实现：

```
void
derived::
foo( int ival = base_default_value )
{
    int real_default_value = 2048;

    if ( ival == base_default_value )
        ival = real_default_value;
    // ...
}
```

17.5.5 虚拟析构函数

在下面的函数中，我们如下应用 delete 表达式：

```
void doit_and_bedone ( vector< Query* > *pvec )
{
    // ...
    for ( ; it != end_it; ++it )
    {
        Query *pq = *it;
        // ...
```

```
            delete pq;
        }
    }
```

为了使函数能够正确执行，在应用 delete 表达式时，必须调用 pq 指向的动态类型的析构函数。为此，必须把 Query 类析构函数声明为虚拟的：

```
class Query {
public:
    virtual ~Query() { delete _solution; }
    // ...
};
```

现在，每个后来派生的类的析构函数都被自动调用，doit_and_bedone()正确执行。

在继承机制下的析构函数的行为如下：派生类的析构函数先被调用。在 pq 的情况下，它是一个虚拟函数调用。完成之后，直接基类的析构函数被静态调用——如果被声明为 inline，则被内联展开。例如，如果 pq 指向一个 AndQuery 对象：

```
delete pq;
```

则通过虚拟机制调用 AndQuery 析构函数。然后再静态调用 BinaryQuery 析构函数。而这之后，Query 析构函数也被静态调用。

已知下列类层次结构：

```
class Query {
public: // ...

protected:
    virtual ~Query();

    // ...
};

class NotQuery : public Query {
public:
    ~NotQuery();

    // ...
};
```

当通过 NotQuery 对象调用 NotQuery 析构函数时，它的访问级别是 public，但是，当通过 Query 指针或引用来调用析构函数时，它是 protected。即，虚拟函数承接了“调用者所属类类型”的访问级别。因此：

```
int main()
{
    Query *pq = new NotQuery;

    // 非法：析构函数是 protected
    delete pq;
}
```

作为一般规则，我们建议将类层次结构的根基类（声明了一个或多个虚拟函数）的析构函数声明为虚拟的。但是，不像基类的构造函数，一般地，基类的析构函数不应该是 protected。

17.5.6 eval()虚拟函数

Query 类层次结构的核心是 eval()虚拟函数（实际上它与语言特性的关系最小）。再次说明，因为在抽象 Query 类中 eval()没有有意义的实现，所以我们把它声明为纯虚拟实例：

```
class Query {
public:
    virtual void eval() = 0;
    // ...
};
```

一个名字的实际计算过程发生在构建单词位置映射表期间。如果一个单词在文本中出现，则它的位置向量也一定出现在映射表中。在我们的实现中，如果存在位置向量，则它和单词的名字一起被传递给 NameQuery 的构造函数。NameQuery 的 eval()实例什么也没做。

但是，我们不能允许它继承 Query 声明的纯虚拟实例。为什么？因为 NameQuery 是一个实体类，它代表了应用程序领域中的实际对象。如果它继承了 Query 的纯虚拟函数，则 NameQuery 将是一个抽象类，这将禁止我们创建 NameQuery 类对象。所以，我们只是把 eval()定义为空函数：

```
class NameQuery : public Query {
public:
    virtual void eval() {}
    // ...
};
```

NotQuery 找到“没有出现其操作数”的所有文本行。这些行的行列对被放入_loc 的 NotQuery 的实例中。下面是我们的实现：

```
void NotQuery::eval()
{
    // 确保操作数已被计算
    _op->eval();

    // all_locs 是所有文本位置对的 vector
    // 它是 NotQuery 的静态成员：
    // static const vector<location>* _all_locs
    vector< location >::const_iterator
    iter = _all_locs->begin(),
    iter_end = _all_locs->end();

    // 获取操作数出现的行的集合
    set<short> *ps = _vec2set( _op->locations() );

    // 为没有找到操作数的每一行
    // 把所有的位置对拷贝到 _loc 中
    for ( ; iter != iter_end; ++iter )
    {
        if ( ! ps->count( (*iter).first ))
            _loc.push_back( *iter );
    }
}
```

下面是 Not 查询的计算过程。操作数出现在文本的 0、3、5 行。（在内部，我们在一个

字符串向量中索引文本，因此，计数从 0 开始。当向用户显示行数时，计数从 1 开始。）所以，NotQuery 的计算过程创建了一个位置向量，1、2、4 行的所有行列对都放在里面。（为了使显示最小化，我们已经对位置向量作了编辑修改。）

```
==> ! daddy
daddy ( 3 ) lines match
display_location vector:
     first: 0 second: 8
     first: 3 second: 3
     first: 5 second: 5
! daddy ( 3 ) lines match
display_location vector:
     first: 1 second: 0
     first: 1 second: 1
     first: 1 second: 2
     ...
     first: 1 second: 10
     first: 2 second: 0
     first: 2 second: 1
     ...
     first: 2 second: 12
     first: 4 second: 0
     first: 4 second: 1
     ...
     first: 4 second: 12
     Requested query: ! daddy
( 2 ) when the wind blows through her hair, it looks almost alive,
( 3 ) like a fiery bird in flight. A beautiful fiery bird, he tells her,
( 5 ) she tells him, at the same time wanting him to tell her more.
```

OrQuery 利用泛型算法 merge()合并其两个操作数的位置向量。为了使 merge()能够对“行列对”进行排序，我们定义了一个函数对象来判断两个行列对中哪一个更小。下面是我们的实现：

```
class less_than_pair {
public:
     bool operator()( location loc1, location loc2 )
     {
          return (( loc1.first < loc2.first ) ||
          ( loc1.first == loc2.first ) &&
          ( loc1.second < loc2.second ));
     }
};
void OrQuery::eval()
{
     // 计算左右操作数
     _lop->eval();
     _rop->eval();
     // 准备合并这两个位置向量
     vector< location, allocator >::const_iterator
          riter = _rop->locations()->begin(),
          liter = _lop->locations()->begin(),
```

```
            riter_end = _rop->locations()->end(),
            liter_end = _lop->locations()->end();

        merge( liter, liter_end, riter, riter_end,
            inserter( _loc, _loc.begin() ),
            less_than_pair() );
    }
```

下面是 Or 查询的计算过程，其中显示了每个 OrQuery 操作数以及 merge()结果的位置向量。（记住向用户显示的行数从 1 开始，而内部从 0 开始。）

```
    ==> fiery || untamed
    fiery ( 1 ) lines match
    display_location vector:
        first: 2 second: 2
        first: 2 second: 8

    untamed ( 1 ) lines match
    display_location vector:
        first: 3 second: 2
    fiery || untamed ( 2 ) lines match
    display_location vector:
        first: 2 second: 2
        first: 2 second: 8
        first: 3 second: 2
    Requested query: fiery || untamed
    ( 3 ) like a fiery bird in flight. A beautiful fiery bird, he tells her,
    ( 4 ) magical but untamed. "Daddy, shush, there is no such thing,"
```

AndQuery 的实现需要对两个操作数的位置向量进行迭代，以查找相邻的单词。每一个被找到的行列位置对被插入到_loc 中。实现过程中的主要工作是保持两个操作数位置的同步，以便我们可以比较它们是否相邻。如下所示：

```
    void AndQuery::eval()
    {
        // 计算左右操作数
        _lop->eval();
        _rop->eval();

        // grab the iterators
        vector< location, allocator >::const_iterator
            riter = _rop->locations()->begin(),
            liter = _lop->locations()->begin(),
            riter_end = _rop->locations()->end(),
            liter_end = _lop->locations()->end();

        // 当它们都有元素需要比较时，循环
        while ( liter != liter_end && riter != riter_end )
        {
            // 当左行数大于右行数时
            while ( (*liter).first > (*riter).first ) {
                ++riter;
                if ( riter == riter_end ) return;
```

```
            }
            // 当左行数小于右行数时
            while ( (*liter).first < (*riter).first )
            {
                // 如果在一行的最后一个单词
                // 和下一行的首单词发现匹配
                // _max_col: 标识了一行的最后一个单词
                if ( (*liter).first == (*riter).first-1 &&
                     (*riter).second == 0 &&
                     (*liter).second == (*_max_col)[ (*liter).fir
                {
                     _loc.push_back( *liter );
                     _loc.push_back( *riter );
                     ++riter;
                     if ( riter == riter_end ) return;
                }
                ++liter;
                if ( liter == liter_end ) return;
            }
            // 当都在同一行时
            while ( (*liter).first == (*riter).first )
            {
                if ( (*liter).second+1 == ((*riter).second) )
                { // ok: 一个相邻的匹配
                     _loc.push_back( *liter ); ++liter;
                     _loc.push_back( *riter ); ++riter;
                }
                else
                     if ( (*liter).second <= (*riter).second )
                          ++liter;
                     else ++riter;
                     if ( liter == liter_end || riter == riter_end )
                          return;
            }
        }
    }
```

下面是 And 查询的计算过程，我们显示了 AndQuery 操作数的位置向量以及最终计算结果的位置向量。（再次提醒，显示给用户的行数从 1 开始，而在内部从 0 开始。）

```
==> fiery && bird
fiery ( 1 ) lines match
display_location vector:
     first: 2 second: 2
     first: 2 second: 8
bird ( 1 ) lines match
display_location vector:
     first: 2 second: 3
     first: 2 second: 9
fiery && bird ( 1 ) lines match
display_location vector:
     first: 2 second: 2
```

```
        first: 2 second: 3
        first: 2 second: 8
        first: 2 second: 9
    Requested query: fiery && bird
    ( 3 ) like a fiery bird in flight. A beautiful fiery bird, he tells her,
```

最后，是 And-Or 复合查询的计算过程。每一个中间结果的位置向量被显示出来。而最终的结果位置向量也被显示出来。

```
    ==> fiery && ( bird || untamed )
    fiery ( 1 ) lines match
    display_location vector:
        first: 2 second: 2
        first: 2 second: 8
    bird ( 1 ) lines match
    display_location vector:
        first: 2 second: 3
        first: 2 second: 9
    untamed ( 1 ) lines match
    display_location vector:
        first: 3 second: 2
    ( bird || untamed ) ( 2 ) lines match
    display_location vector:
        first: 2 second: 3
        first: 2 second: 9
        first: 3 second: 2
    fiery && ( bird || untamed ) ( 1 ) lines match
    display_location vector:
        first: 2 second: 2
        first: 2 second: 3
        first: 2 second: 8
        first: 2 second: 9
    Requested query: fiery && ( bird || untamed )
    ( 3 ) like a fiery bird in flight. A beautiful fiery bird, he tells her,
```

17.5.7 虚拟 new 操作符

已知一个指针指向一个实体查询子类型，那么在堆中分配一个复制的对象就是一件很容易的事情。例如：

```
    NotQuery *pnq;
    // set pnq ...

    // new 表达式调用 NotQuery 的拷贝构造函数
    NotQuery *pnq2 = new NotQuery( *pnq );
```

但是，如果已知一个指向抽象 Query 的指针，那么，分配一个复制的对象就不那么容易了。例如：

```
    const Query *pq = pnq->op();
    // 怎样复制 pq?
```

如果我们能够声明一个操作符 new 的虚拟实例，则问题就解决了。正确的 new 操作符实

例将被自动调用。不幸的是，new 操作符不能被声明为虚拟的，因为它是一个静态成员函数，在构造类对象之前被应用到未使用的内存上（见 15.8 节讨论）

虽然我们不能把 new 操作符声明为虚拟的，但是我们可以提供一个代理 new 操作符，来把我们的对象分配并拷贝到空闲存储区中，这个代理通常被称为 clone()：

```
class Query {
public:
    virtual Query *clone() = 0;

    // ...
};
```

下面是 NameQuery 实例的一种实现方式：

```
class NameQuery : public Query {
public:
    virtual Query *clone()
        // 调用 NameQuery 的拷贝构造函数
        { return new NameQuery( *this ); }

    // ...
};
```

当目标指针的类型是 Query*时，这是完全正确的，如：

```
Query *pq = new NameQuery( "Valery" );
Query *pq2 = pq->clone();
```

当目标指针类型是实际的 NameQuery*时，它工作得就有些不太好了。在这种情况下，要求我们提供一个向下转换把 Query*指针转换回 NameQuery*：

```
NameQuery *pnq = new NameQuery( "Rilke" );
NameQuery *pnq2 =
    static_cast<NameQuery*>( pnq->clone() );
```

（需要向下转换的原因在 19.ll 节解释。）

前面我们说过，对于“要求派生类的返回类型必须与其基类实例的返回类型完全匹配”有一个例外，这个例外就是用来支持这种情况的。如果虚拟函数的基类实例返回一个类类型（或指向类类型的指针或引用），则派生类实例可以返回一个“从基类实例返回的类公有派生出来”的类（或指向类类型的指针或引用）：

```
class NameQuery : public Query {
public:
    virtual NameQuery *clone()
        { return new NameQuery( *this ); }

    // ...
};
```

现在，pq2 和 pnq2 的初始化都可以实现，而无需显式强制转换。

```
// Query *pq = new NameQuery( "Broch" );
Query *pq2 = pq->clone();          // ok

// NameQuery *pnq = new NameQuery( "Rilke" );
NameQuery *pnq2 = pnq->clone();    // ok
```

下面是 NotQuery 的 clone 实现：

```
class NotQuery : public Query {
public:
    virtual NotQuery *clone()
        { return new NotQuery( *this ); }
    // ...
};
```

AndQuery 和 OrQuery 实例以类似的方式实现。为了这些 clone()的实现能够成功，我们必须提供显式的 AndQuery、NotQuery 和 OrQuery 的拷贝构造函数实例。我们将在 17.6 节这样做。

17.5.8 虚拟函数、构造函数和析构函数

正如 17.4 节所示，派生类对象中构造函数的调用顺序是，先调用基类的构造函数，然后是派生类的构造函数。例如，当我们定义一个 NameQuery 类对象时，如：

```
NameQuery poet( "Orlen" );
```

构造函数的调用顺序是先 Query，然后 NameQuery。

当执行 Query 基类的构造函数时，poet 的 NameQuery 部分还没有被初始化。实际上，poet 还不是一个 NameQuery，只有它的 Query 子对象被构造了。

如果在基类的构造函数中调用了一个虚拟函数，而基类和派生类都定义了该函数的实例，将会怎么样？应该调用哪一个函数实例？如果可以调用虚拟函数的派生类实例，并且它访问任意的派生类成员，那么调用的结果在逻辑上是未定义的。而程序可能会崩溃。

为了防止这样的事情发生，在基类构造函数中调用的虚拟实例总是在基类中活动的虚拟实例。实际上，在基类构造函数中，派生类对象只不过是一个基类类型的对象而已。

对于派生类对象，在基类析构函数中也是如此：派生类部分也是未定义的，但是，这一次不是因为它还没有被构造，而是因为它已经被销毁。

练习 17.12

在 NameQuery 中，位置向量最直接的内部表示，是用“文本位置映射表中存储的指针”进行初始化的指针。这也是最有效的方式，因为我们只拷贝一个地址而不是向量中的所有位置对。AndQuery、OrQuery 和 NotQuery 类必须根据其操作数的计算结果构造自己的位置向量。当这样的类对象的生命期结束时，相关的位置向量也必须被删除。当 NameQuery 对象的生命期结束时，位置向量不能被删除。我们怎样才能把这些位置向量作为指针存储在 Query 基类中，并且删除掉与 AndQuery、OrQuery 和 NotQuery 相关联的实例而保留 NameQuery 类对象的实例呢？（注意，不准许我们在 Query 中增加一个标志，指明是否删除位置向量指针！）

练习 17.13

下面的类定义有何错误？

```
class AbstractObject {
public:
```

```
    ~AbstractObject();

    virtual void doit() = 0;

    // ...
};
```

练习 17.14

已知：

```
NameQuery nq( "Sneezy" );
Query q( nq );
Query *pq = &nq;
```

为什么如下调用：

```
pq->eval();
```

调用的是 NameQuery 的 eval()实例，而：

```
q.eval();
```

调用的是 Query 实例？

练习 17.15

下面派生类的虚拟函数的重新声明哪些是错误的？

```
(a)  Base* Base::copy( Base* );
     Base* Derived::copy( Derived* );
(b)  Base* Base::copy( Base* );
     Derived* Derived::copy( Base* );
(c)  ostream& Base::print( int, ostream&=cout );
     ostream& Derived::print( int, ostream& );
(d)  void Base::eval() const;
     void Derived::eval();
```

练习 17.16

在实践中，我们的程序在初次练习或初次用实际数据练习时，几乎不可能正确运行。把调试策略整合到类的设计中，常常会很有用。请为我们的 Query 类层次结构实现一个 debug()虚拟函数，它显示了各个类的数据成员。同时支持一个细节控制级别：（a）以 debug()函数的实参，（b）以类的数据成员。（后者允许单独的类对象打开或关闭调试信息的显示。）

练习 17.17

在下面的继承层次中有什么错误？

```
class Object {
public:
    virtual void doit() = 0;
    // ...
protected:
```

```
        virtual ~Object();
};

class MyObject : public Object {
public:
        MyObject( string isA );
        string isA() const;

protected:
        string _isA;
};
```

17.6 按成员初始化和赋值 ※

设计一个类的责任之一就是确保在按成员初始化（在 14.6 介绍）和按成员赋值（在 14.7 节介绍）下，该类能正确有效地工作。本节我们将考虑在继承机制下的这些操作。

到目前为止，我们还没有为按成员初始化提供显式处理。现在让我们来一步一步地看看，在缺省情况下我们的 Query 层次结构会怎么样。

抽象 Query 基类定义了三个非静态数据成员：

```
class Query {
public: // ...

protected:
        int _paren;
        set<short> *_solution;
        vector<location> _loc;
        // ...
};
```

如果_solution 被设置了，则它指向由成员函数_vec2set()在空闲存储区中分配的 set。Query 的析构函数把 delete 表达式应用到_solution 上面。

Query 类需要提供一个显式的拷贝构造函数和一个显式的拷贝赋值操作符（如果你还不太清楚这些，请复习 14.6 节）。但是我们在提供这两者之前，先来看看缺省的按成员拷贝。

NameQuery 派生类包含一个 string 成员类对象以及一个 Query 基类子对象。已知如下的 NameQuery 对象“folk”：

```
NameQuery folk( "folk" );
```

用 folk 对 music 进行初始化：

```
NameQuery music = folk;
```

会导致以下事情发生：

1．编译器检查 NameQuery 是否定义了一个显式的拷贝构造函数实例。答案是没有。所以，编译器准备应用缺省的按成员初始化。

2．编译器接下来检查 NameQuery 类是否含有基类子对象。是的，它含有 Query 基类子对象。

3．编译器检查 Query 基类是否定义了显式的拷贝构造函数实例。答案也是没有。所以编译器准备应用缺省的按成员初始化。

4．编译器检查 Query 类是否含有基类子对象。没有。

5．编译器以声明的顺序检查 Query 的每个非静态成员。如果成员是非类对象，如_paren 和_solution，则它用 folk 的成员值初始化 music 对象的成员。如果成员是类对象，如_loc，则它递归地应用步骤 1。是的，vector 类定义了一个显式的拷贝构造函数实例。该拷贝构造函数被调用，用 folk._loc 初始化 music._loc。

6．然后编译器按声明的顺序检查 NameQuery 类型的每个非静态成员。string 成员类对象被识别出来，它有一个显式的拷贝构造函数。于是调用该拷贝构造函数，用 folk._name 初始化 music._name。

现在，用 folk 初始化 music 的缺省初始化已经完成了。除了_solution 的缺省拷贝（如果允许的话）可能引起程序失败之外，程序执行得很好。我们提供显式的 Query 类拷贝构造函数，以改变缺省的处理。一个解决方案是，拷贝整个结果集（_solution），如下。

```
Query::Query( const Query &rhs )
    : _loc( rhs._loc ), _paren(rhs._paren)
{
    if ( rhs._solution )
    {
        _solution = new set<short>;
        set<short>::iterator
            it = rhs._solution->begin(),
            end_it = rhs._solution->end();
        for ( ; it != end_it; ++it )
            _solution->insert( *it );
    }
    else _solution = 0;
}
```

但是，因为我们对于结果集的实现是按需计算的，所以没有必要拷贝它。我们的拷贝构造函数的目的就是防止这种缺省拷贝，把_solution 初始化为 0 已经足够了：

```
Query::Query( const Query &rhs )
    : _loc( rhs._loc ),
    _paren(rhs._paren), _solution( 0 )
{ }
```

用 folk 初始化 music 遵循同样的步骤 1 和步骤 2。现在步骤 3 发现 Query 定义了一个显式的拷贝构造函数。这个拷贝构造函数被调用，而第 4 步和第 5 步不再执行。第 6 步和以前一样执行。

于是，用 folk 初始化 music 的按成员初始化就完成了，它执行得很好。NameQuery 不需要提供显式的拷贝构造函数。

NotQuery 派生类含有一个 Query 基类子对象和一个 Query*数据成员_op（指向在空闲存储区中分配的操作数）。NotQuery 的析构函数在该操作数上应用 delete 表达式。

NotQuery 类不能安全地允许对其_op 成员进行缺省的按成员初始化，所以必须提供一个显式的拷贝构造函数。它的实现利用了上节定义的虚拟 clone()函数。如下所示：

```
inline NotQuery::
NotQuery( const NotQuery &rhs )
    // 调用 Query::Query( const Query &rhs )
```

```
    : Query( rhs )
    { _op = rhs._op->clone(); }
```

用一个 NotQuery 类对象按成员初始化另一个 NotQuery 类对象会引发下列两步动作：

1．编译器检查 NotQuery 是否定义了一个显式的拷贝构造函数实例。是的。

2．调用 NotQuery 拷贝构造函数执行按成员初始化。

就是这样。NotQuery 的拷贝构造函数有责任执行基类子对象和非静态数据成员的正确初始化工作。而 AndQuery 和 OrQuery 实例与 NotQuery 类似，留给读者作练习。

按成员赋值与按成员初始化类似。如果存在一个显式的拷贝赋值操作符，则它被调用，用一个类对象向另一个类对象赋值。否则，应用缺省的按成员赋值。

如果存在基类，则首先对基类子对象按成员赋值。如果基类提供了一个显式的拷贝赋值操作符，则调用它。否则，递归地对基类和基类子对象的成员应用缺省按成员赋值的动作。

编译器按声明的顺序检查每个非静态数据成员。如果它是非类类型，则右边的实例被拷贝到左边。如果它是类类型，并且该类定义了显式的拷贝赋值操作符，则调用该操作符；否则，递归地对基类和成员类对象的成员应用缺省按成员赋值的动作。

下面是 Query 的拷贝赋值操作符。现在我们并不需要拷贝结果集，而只是防止它的缺省拷贝：

```
Query&
Query::
operator=( const Query &rhs )
{
    // 阻止自我赋值
    if ( &rhs != this )
    {
        _paren = rhs._paren;
        _loc = rhs._loc;

        delete _solution;

        _solution = 0;
    }
    return *this;
}
```

NameQuery 不要求显式的拷贝赋值操作符。用一个 NameQuery 对象向另一个 NameQuery 对象赋值将导致下列两步动作：

1．调用显式的 Query 拷贝赋值操作符，在两个 NameQuery 对象的 Query 子对象之间赋

2．调用显式的 string 拷贝赋值操作符，在两个 NameQuery 对象的 string 成员类对象之间

对于 NameQuery 的显式赋值操作符我们不能再优化了，缺省按成员赋值所做的已经足够了。

NameQuery、OrQuery 和 AndQuery 都要求显式的拷贝赋值操作符，以便安全地拷贝各自的操作数。下面是 NotQuery 的实例：

```
inline NotQuery&
NotQuery::
```

```
operator=( const NotQuery &rhs )
{
    // 阻止自我赋值
    if ( &rhs != this )
    {
        // 调用 Query 拷贝赋值操作符
        this->Query::operator=( rhs );

        // 拷贝操作数
        _op = rhs._op->clone();
    }
    return *this;
}
```

不像拷贝构造函数，拷贝赋值操作符没有特殊的部分可以通过它来调用基类的赋值操作符。要想调用基类的赋值操作符，可以有两种语法形式：通过显式的调用（如上所示），以及通过如下的显式强制转换：

```
(*static_cast<Query*>(this)) = rhs;
```

AndQuery 和 OrQuery 拷贝赋值实例与此类似，留作练习。

下面的小测试程序使用了我们的实现——检查它实际的工作情况。我们只是创建或拷贝一个对象，然后输出它的值：

```
#include "Query.h"
int
main()
{
    NameQuery nm( "alice" );
    NameQuery nm2( "emma" );

    NotQuery nq1( &nm );
    cout << "notQuery 1: " << nq1 << endl;

    NotQuery nq2( nq1 );
    cout << "notQuery 2: " << nq2 << endl;

    NotQuery nq3( &nm2 );
    cout << "notQuery 3: " << nq3 << endl;

    nq3 = nq2;
    cout << "notQuery 3 assigned nq2: " << nq3 << endl;

    AndQuery aq( &nq1, &nm2 );
    cout << "AndQuery : " << aq << endl;

    AndQuery aq2( aq );
    cout << "AndQuery 2: " << aq2 << endl;

    AndQuery aq3( &nm, &nm2 );
```

```
    cout << "AndQuery 3: " << aq3 << endl;

    aq2 = aq3;
    cout << "AndQuery 2 after assign: " << aq2 << endl;
}
```

编译并运行该程序，产生如下输出：

```
notQuery 1: ! alice
notQuery 2: ! alice
notQuery 3: ! emma
notQuery 3 assigned nq2: ! alice
AndQuery : ! alice && emma
AndQuery 2: ! alice && emma
AndQuery 3: alice && emma
AndQuery 2 after assign: alice && emma
```

练习 17.18

请实现 AndQuery 和 OQuery 的拷贝构造函数。

练习 17.19

请实现 AndQuery 和 OrQuery 的拷贝赋值操作符。

练习 17.20

哪些因素可以用来表明一个类需要显式的拷贝构造函数和拷贝赋值操作符？

17.7 UserQuery 管理类

给出查询，如：

```
fiery && ( bird || potato )
```

我们的任务是建立一个等价的 Query 层次结构：

```
AndQuery
    NameQuery( "fiery" )
    OrQuery
        NameQuery( "bird" )
        NameQuery( "potato" )
```

问题是，怎样才能把它做到最好？计算一个查询的过程类似于一个有限状态机。我们以空状态开始，然后，每一个查询元素从一个状态转换到另一个，直到整个查询都被计算过为止。我们实现的核心是一个大型的 switch 语句，它位于一个名为 eval_query()的操作中。用户查询的每个单词依次从 string vector 中读入，并测试其每一种可能：

```
vector<string >::iterator
    it = user_query->begin(),
    end_it = user_query->end();
```

```
for ( ; it != end_it; ++it )
    switch( evalQueryString( *it ))
    {
    case WORD:
        evalWord( *it );
        break;

    case AND:
        evalAnd();
        break;

    case OR:
        evalOr();
        break;

    case NOT:
        evalNot();
        break;
    case LPAREN:
        ++_paren;
        ++_lparenOn;
        break;
    case RPAREN:
        --_paren;
        ++_rparenOn;
        evalRParen();
        break;
    }
```

五个计算操作［_evalWord()、evalAnd()、evalOr()、endaNot()和 evalRparen()］完成了 Query 层次结构的实际创建工作。在查看它们的详细实现之前，我们先来考虑它们的组织。

一种策略是把它们都定义成单独的函数，正如在第 6 章文本查询例子程序中所做的那样。用户查询和派生的 Query 子类型代表了独立的数据，供这些函数使用。这是一种过程化程序设计模型，它不是我们所追求的。

就像在 6.14 节所做的：通过引入一个 TextQuery 类来封装第 6 章的操作和数据。而这里，我们希望引入一个 UserQuery 类来封装和管理这些操作和数据。

第一个数据成员是 string vector，它包含了实际的用户查询。第二个数据成员是 Query* 类型的指针，指向在 eval_query()中建立起来的查询层次表示。我们还定义了三个额外的成员来处理括号：_paren，帮助我们改变操作符计算的缺省优先级（稍后我们将给出一个例子），_lparenOn 和_rparenOn 记录了与当前查询节点相关联的括号的种类和数目［（我们在 17.5.1 节讨论虚拟 print()函数中看到了它们的用法］。

除了这五个成员之外，我们还需要两个。考虑下列查询：

```
fiery || untamed
```

我们的真正目标是把这个查询表示成下面的 OrQuery 对象：

```
OrQuery
    NameQuery( "fiery" )
    NameQuery( "untamed" )
```

但是，处理查询的顺序有点问题。首先，我们定义了一个 NameQuery，但是没有定义“将其加到上面”的 OrQuery。我们需要一个地方来暂时存储 NameQuery 对象，以备后来可以获取到。

为了“把某个事物放在某地以备后来可以获取到”的传统数据结构是栈（stack），我们将把 NameQuery 对象放在栈中。当下次遇到 OrQuery 操作符时，我们将获取到 NameQuery。并把它作为左操作数传递给 OrQuery。

完成之后，对于 OrQuery 对象我们该做什么呢？在这时，我们的 OrQuery 对象是不完整的，它缺少右操作数。我们需要把它放在一边，直到右操作数可供使用。

我们可以把它与 NameQuery 对象存放在同一个栈中。但是，OrQuery 对象代表了一个不同的处理状态：它是一个不完整操作符。我们更愿意定义两个栈：一个用来存放复合查询中的完整操作数（我们放置 NameQuery 对象的地方，并把它命名为_query_stack），另一个用来存放缺少右操作数的不完整操作符。我们认为这个栈含有当前要完成的操作，所以将它命名为_current_op。这正是放置 OrQuery 对象的地方。当我们定义了第二个 NameQuery 对象时，就从_current_op 中获取到 orQuery 对象，并把该 NameQuery 对象作为第二个操作数加上去。现在 OrQuery 是完整的了，我们把它压入到_query_stack 中。

当用户查询的处理工作完成时，如果每件事情都进行得很好，则_current_op 是空的。且_query_stack 应该包含一个对象。该对象是用户查询的完整表示，在我们的例子中，也就是 OrQuery 对象。

为了了解这是怎样工作的，我们来看一些实际的查询。第一个例子是一个简单的 NotQuery：

```
! daddy
```

下面是该查询的实际处理过程，在_query_stack 中的最终对象是 NotQuery 对象：

```
evalNot() : incomplete!
     push on _current_op (size == 1 )
evalWord() : daddy
     pop _current_op : NotQuery
     add operand: WordQuery : NotQuery complete!
     push NotQuery on _query stack
```

在 eval 操作下面的缩进文字表明了正在执行的操作。第二个例子是一个复合的 OrQuery。它说明了向_query_stack 压入一个完整的操作符的情况。

```
==> fiery || untamed || shyly
evalWord() : fiery
     push word on _query stack
evalOr() : incomplete!
     pop _query_stack : fiery
     add operand : WordQuery : OrQuery incomplete!
     push OrQuery on _current_op (size == 1 )
evalWord() : untamed
     pop _current_op : OrQuery
     add operand: WordQuery : OrQuery complete!
     push OrQuery on _query stack
evalOr() : incomplete!
```

```
        pop _query_stack : OrQuery
        add operand : OrQuery : OrQuery incomplete!
        push OrQuery on _current_op (size == 1 )
    evalWord() : shyly
        pop _current_op : OrQuery
        add operand: WordQuery : OrQuery complete!
        push OrQuery on _query stack
```

我们的最后一个例子说明了复合查询和用括号改变计算顺序的用法：

```
    ==> fiery && ( bird || untamed )
    evalWord() : fiery
        push word on _query stack
    evalAnd() : incomplete!
        pop _query_stack : fiery
        add operand : WordQuery : AndQuery incomplete!
        push AndQuery on _current_op (size == 1 )
    evalWord() : bird
        _paren is set to 1
        push word on _query stack
    evalOr() : incomplete!
        pop _query_stack : bird
        add operand : WordQuery : OrQuery incomplete!
        push OrQuery on _current_op (size == 2 )
    evalWord() : untamed
        pop _current_op : OrQuery
        add operand: WordQuery : OrQuery complete!
        push OrQuery on _query stack
    evalRParen() :
        _paren: 0 _curent_op.size(): 1
        pop _query_stack : OrQuery
        pop _current_op: AndQuery
        add operand : OrQuery : AndQuery complete!
        push AndQuery on _query_stack
```

我们实现的文本查询系统由三个组件构成：①基于对象的 TextQuery 类，它进行实际的文本处理（在 6.14 节详细介绍）；②面向对象的 Query 类层次结构，表达和计算每一个用户查询；③基于对象的 UserQuery 类，它表示一个有限状态机，用来建立 Query 层次结构。

到现在为止，我们已经实现了这三个互相独立的大型组件，而且在它们之间没有冲突。不幸的是，实际情况并不是这样。你看到问题了吗？Query 类层次结构不支持由“刚刚跟踪过的 UserQuery 实现”对其提出的对象构造要求！

1．当前的 AndQuery、OrQuery 和 NotQuery 类型都要求在定义每个对象时，每个操作数都必须存在。但是，我们的处理过程却要求我们定义不完整的对象。

2．我们的处理过程要求我们能够后续地向 AndQuery、OrQuery 和 NotQuery 类增加操作数。而且，这必须是一个虚拟操作。我们必须通过“被压入到_current_op 中的 Query*指针”来增加操作数。但是，增加操作数是依赖于类型的，取决于它是一元操作（NotQuery）或二元操作（OrQuery 和 AndQuery）。正如定义所示，我们的 Query 类层次结构没有提供这样的操作。

这里发生的情况是，领域分析产生的接口与设计的实际实现有所不同。并不是这种分析

不正确，而是不完整。这或多或少是一个程度的问题，分析、设计和实现是独立的三个阶段，并且都被视为顺序瀑布（waterfall）模型，不允许反馈和修改。从根本上，我们必须承认这个事实，我们不能想到或预料到所有的事情。困难在于，（在我们的头脑意识中，以及在我们的管理中）如何区分“复杂处理过程中不可避免的错误”，以及“由于个人缺少时间或不注意而引起的错误”。

在这种情况下，我们必须回过头来修改 Query 类层次结构，或者对这些改变进行协商。在一个低效的组织中，互相指责只会耽误开发进度。项目组将会变得没有积极性，并且充满官僚气氛。在我们的情况下，作为本书的作者，我们只是进入到代码中，摆弄这些代码，并修改子类型的构造函数，以及增加一个虚拟的 add_op()成员函数，以支持“在一个操作符对象被定义之后再增加操作数”的特性［我们将很快在查看 evalRParen()和 evalWord()操作时了解它的用法］。

17.7.1 定义 UserQuery 类

UserQuery 对象可以用“一个表示用户查询的字符串向量的指针”来初始化，或者在以后通过 query()成员函数，向它传递一个用户查询的地址。这使得一个查询对象可以被用于多个用户查询。Query 类层次结构的实际建立过程由 eval_query()操作执行。例如：

```
// 仅仅定义一个实例，没有相关联的用户查询
UserQuery user_query;
string text;
vector<string> query_text;
// 对每个用户请求进行循环
do {
     while( cin >> text )
          query_text.push_back( text );

     // 把查询传递给 UserQuery 对象
     user_query.query( &query_text );

     // 计算查询并返回
     // Query* 层次结构的根
     Query *query = user_query.eval_query();
}
while ( /* 继续下一个用户查询 */ );
```

下面是 UserQuery 类的定义：

```
#ifndef USER_QUERY_H
#define USER_QUERY_H

#include <string>
#include <vector>
#include <map>
#include <stack>

typedef pair<short,short>    location;
typedef vector<location,allocator> loc;
```

```
#include "Query.h"

class UserQuery {
public:
    UserQuery( vector< string,allocator > *pquery = 0 )
        : _query( pquery ), _eval( 0 ), _paren( 0 ) {}

    Query *eval_query();        // 建立层次结构
    void query( vector< string,allocator > *pq );
    void displayQuery();

    static void
    word_map( map<string,loc*,less<string>,allocator> *pwm )
        { if ( !_word_map ) _word_map = pwm; }
private:
    enum QueryType { WORD = 1, AND, OR, NOT, RPAREN, LPAREN };

    QueryType    evalQueryString( const string &query );
    Void         evalWord( const string &query );
    void         evalAnd();
    void         evalOr();
    void         evalNot();
    void         evalRParen();
    bool         integrity_check();

    int               _paren;
    Query             *_eval;
    vector<string>    *_query;

    stack<Query*,vector<Query*> > _query_stack;
    stack<Query*,vector<Query*> > _current_op;

    static short _lparenOn, _rparenOn;
    static map<string,loc*,less<string>,allocator>
        *_word_map;
};
#endif
```

注意，我们声明的两个栈都包含了 Query 指针类型的元素，而不是 Query 对象本身。虽然这两种实现方法都能够支持正确的应用程序行为，但是直接存储对象的效率很低：每个对象（和它的操作数）必须按成员拷贝到栈中（每个操作数被通过 clone()虚拟调用而按成员拷贝），然后再被销毁。除非我们真的需要修改放在容器中的类对象，否则通过指针存储它们能够更有效。

下面是各种计算操作，我们已经把它们定义为 inline 的。evalAnd()和 evalOr()执行下列步骤：都对_query_stack 执行“弹出”动作［这让标准库的 stack 类中采取了两个操作：top()获得元素，pop()将其从栈中弹出］。从堆中分配一个 AndQuery 或 OrQuery 对象，并把从_query_stack 中获取的对象传递给它。它们都向 AndQuery 或 OrQuery 对象传递左、右括号的数目，以后操作符在显示自身时需要这些信息。最后，它们都把不完整的操作符压入到

_current_op 中：

```
inline void
UserQuery::
evalAnd()
{
     Query *pop = _query_stack.top(); _query_stack.pop();
     AndQuery *pq = new AndQuery( pop );

     if ( _lparenOn )
          { pq->lparen( _lparenOn ); _lparenOn = 0; }

     if ( _rparenOn )
          { pq->rparen( _rparenOn ); _rparenOn = 0; }

     _current_op.push( pq );
}

inline void
UserQuery::
evalOr()
{
     Query *pop = _query_stack.top(); _query_stack.pop();
     OrQuery *pq = new OrQuery( pop );

     if ( _lparenOn )
          { pq->lparen( _lparenOn ); _lparenOn = 0; }

     if ( _rparenOn )
          { pq->rparen( _rparenOn ); _rparenOn = 0; }

     _current_op.push( pq );
}
```

evalNot()操作的步骤如下：它从堆中分配一个新的 NotQuery 对象。并且向 NotQuery 对象传递左右括号的数目，以后操作符在显示自身时需要这些信息。最后，它把这个不完整的操作符压入到_current_op 中：

```
inline void
UserQuery::
evalNot()
{
     NotQuery *pq = new NotQuery;
     if ( _lparenOn )
          { pq->lparen( _lparenOn ); _lparenOn = 0; }

     if ( _rparenOn )
          { pq->rparen( _rparenOn ); _rparenOn = 0; }
     _current_op.push( pq );
}
```

当遇到一个右括号时，evalRParen()操作被调用。如果活动的左括号数大于在_current_op 中元素的数目，则该操作什么也不做。否则，它将执行下列步骤：弹出_query_stack，以便获

取当前未分配的操作数；弹出_current_op，以获取当前不完整的操作符；调用 Query 的虚拟 add_op()成员函数，把尚未分配的操作数传递给不完整的操作符。最后，再把现在已经完整的操作符压入到_query_stack 中：

```
inline void
UserQuery::
evalRParen()
{
    if ( _paren < _current_op.size() )
    {
        Query *poperand = _query_stack.top();
        _query_stack.pop();
        Query *pop = _current_op.top();
        _current_op.pop();
        pop->add_op( poperand );
        _query_stack.push( pop );
    }
}
```

evalWord()操作执行下列步骤：它在文本文件相关的_word_map 中查找单词。如果单词存在，则获取该位置向量，并调用双参数的 NameQuery 构造函数，在堆中分配一个新的 NameQuery 对象。如果单词不存在，则调用单参数的 NameQuery 构造函数，在堆中分配一个新的 NameQuery 对象。如果在_current_op 中的元素个数小于等于已经看到的括号数，就会有不完整操作符在等待 NameQuery 操作数。所以 NameQuery 将被压入到_query_stack 中，或获取到_current_op 中不完整的操作符。调用 Query 的虚拟 add_op()成员函数，把 NameQuery 对象传递给不完整的操作符，最后，再把现在已经完整的操作符压入到_query_stack 中：

```
inline void
UserQuery::
evalWord( const string &query )
{
    NameQuery *pq;
    loc *ploc;

    if ( ! _word_map->count( query ))
        pq = new NameQuery( query );
    else {
        ploc = ( *_word_map )[ query ];
        pq = new NameQuery( query, *ploc );
    }

    if ( _current_op.size() <= _paren )
        _query_stack.push( pq );
    else {
        Query *pop = _current_op.top();
        _current_op.pop();
        pop->add_op( pq );
        _query_stack.push( pop );
    }
}
```

练习 17.21

请为 UserQuery 类提供析构函数、拷贝构造函数和拷贝赋值操作符。

练习 17.22

请为 UserQuery 提供 print()函数，并解释你所选择的显示信息。

17.8 组合起来

我们的文本查询应用程序的 main()程序如下所示：

```
#include "TextQuery.h"
int main()
{
    TextQuery tq;
    tq.build_text_map();
    tq.query_text();
}
```

build_text_map()成员函数是 6.14 节的成员函数 doit()重命名之后的函数：

```
inline void
TextQuery::
build_text_map()
{
    retrieve_text();
    separate_words();
    filter_text();
    suffix_text();
    strip_caps();
    build_word_map();
}
```

query_text()成员函数取代了 6.14 节定义的同名实例。在原来的 query_text()实现中，它承担了“接受用户查询”和“显示计算结果”的责任。我们决定在新的 query_text()中继续让它承担这些责任，同时重新实现它所执行的动作：

```
void
TextQuery::query_text()
{
    /* 局部对象:
     *
     * text: 按顺序存放查询中的每个单词
     * query_text: 保存用户查询的 vector
     * caps: 支持 "把大写转换为小写" 的过滤器
     *
     * user_query : UserQuery 对象,
     *              封装了用户查询的实际计算过程
     */

    string text;
```

```
string caps( "ABCDEFGHIJKLMNOPQRSTUVWXYZ" );
vector<string, allocator> query_text;
UserQuery user_query;

// 初始化 UserQuery 的静态数据成员
NotQuery::all_locs( text_locations->second );
AndQuery::max_col( &line_cnt );
UserQuery::word_map( word_map );

do {
    // 如果有的话，删除以前的查询
    query_text.clear();
    cout << "Enter a query - please separate each item "
         << "by a space.\n"
         << "Terminate query (or session) "
         << "with a dot( . ).\n\n"
         << "==> ";
    /*
     * 从标准输入获取查询，
     * 删除所有的大写字母
     * 大量输入 query_text ...
     *
     * 注意：应该完成用户查询的所有处理
     */
    while( cin >> text )
    {
        if ( text == "." )
            break;
        string::size_type pos = 0;
        while (( pos = text.find_first_of( caps, pos ))
                 != string::npos )
            text[pos] = tolower( text[pos] );
        query_text.push_back( text );
    }

    // ok: 如果有查询，处理它...
    if ( ! query_text.empty() )
    {
        // 把查询传递给 UserQuery 对象
        user_query.query( &query_text );

        // 计算 UserQuery
        // 返回 Query* 层次结构
        // 17.7 节描述了这一点
        // query 是 TextQuery 的 Query* 成员
        query = user_query.eval_query();

        // 计算 Query 层次结构
        // 实现见 17.5 节
        query->eval();
```

```
                // ok: 显示结果
                // 一个 TextQuery 成员函数
                display_solution();

                // 在用户终端上给出额外一行
                cout << endl;
            }
        }
        while ( ! query_text.empty() ); cout << "Ok, bye!\n";
    }
```

在 Addison-Wesley 的 ftp 站点上可以找到这个程序的完整代码。作为该程序的最后一个练习，我们将其应用在大量的在线文本上。第一个查询针对 Herman Melville 的短篇小说《Bartleby》。它演示了一个复合的 AndQuery，找到连续文本行中的相邻匹配。（注意，放在斜线中的单词，表示它们是斜体的。）

```
Enter a query - please separate each item by a space.
Terminate query (or session) with a dot( . ).
==> John && Jacob && Astor

    john ( 3 ) lines match
    jacob ( 3 ) lines match
    john && jacob ( 3 ) lines match
    astor ( 3 ) lines match
    john && jacob && astor ( 5 ) lines match

Requested query: john && jacob && astor
( 34 ) All who know me consider me an eminently /safe/ man. The late
 John Jacob
( 35 ) Astor, a personage little given to poetic enthusiasm, had
 no hesitation in
( 38 ) my profession by the late John Jacob Astor, a name which,
 I admit, I love
( 40 ) bullion. I will freely add that I was not insensible to the
 late John Jacob
( 41 ) Astor's good opinion.
```

在下一个查询中，应用了括号和复合操作符，是针对 Joseph Conrad 的小说 《Heart of Darkness》的：

```
==> horror || ( absurd && mystery ) || ( North && Pole )
    horror ( 5 ) lines match
    absurd ( 8 ) lines match
    mystery ( 12 ) lines match
    ( absurd && mystery ) ( 1 ) lines match
    horror || ( absurd && mystery ) ( 6 ) lines match
    north ( 2 ) lines match
    pole ( 7 ) lines match
    ( north && pole ) ( 1 ) lines match
    horror || ( absurd && mystery ) || ( north && pole )
        ( 7 ) lines match
Requested query: horror || ( absurd && mystery ) || ( north && pole )
( 257 ) up I will go there.' The North Pole was one of these
```

```
( 952 ) horrors. The heavy pole had skinned his poor nose.
( 3055 ) some lightless region of subtle horrors, where pure,
( 3673 ) " 'The horror! The horror!'
( 3913 ) the whispered cry, 'The horror! The horror! '
( 3957 ) absurd mysteries not fit for a human being to behold.
( 4088 ) wind. 'The horror! The horror!'
```

最后一个查询应用在 Henry James 的《Portrait of Lady》的片段上。它演示了一个复合查询，以及对于大型文本文件的处理：

```
==> clever && trick || devious

    clever ( 46 ) lines match
    trick ( 12 ) lines match
    clever && trick ( 2 ) lines match
    devious ( 1 ) lines match
    clever && trick || devious ( 3 ) lines match

Requested query: clever && trick || devious
( 13914 ) clever trick she had guessed. Isabel, as she herself grew older,
( 13935 ) lost the desire to know this lady's clever trick. If she had
( 14974 ) desultory, so devious, so much the reverse of processional.
There were
```

练习 17.23

我们对于用户查询的处理还不是很成功，因为我们没有像构建文本内容时所做的那样，把同样的预处理过程应用在每个单词上，参见 6.9 节和 6.10 节。因此。如果用户希望找到 maps，却发现我们的文本表示只能识别 map，那么就应该通过修改 query_text()来提供等价的预处理。

练习 17.24

我们的查询系统可以通过增加一个 InclusiveAndQuery 进一步完善。这种查询可以用一个&来表示。如果两个单词在同一行，但不一定相邻，这时它的计算结果为 true。例如，已知行：

```
We were her pride of ten, she named us
```

那么，下面的 InclusiveAndQuery：

```
pride & ten
```

计算结果为 true，而原来的 AndQuery：

```
pride && ten
```

计算结果为 false。请为 InclusiveAndQuery 提供必要的支持。

练习 17.25

我们当前实现的 display_solution()（见下面）只是打印到标准输出上。更合理的实现是，将允许用户指定 ostream，用它来指导显示。请修改 display_solution()允许用户指定 ostream 对象。在 UserQuery 类定义中要作其他哪些修改？

```
void TextQuery::
display_solution()
{
    cout << "\n"
         << "Requested query: "
         << *query << "\n\n";

    const set<short> *solution = query->solution();
    if ( ! solution->size() ) {
        cout << "\n\tSorry, "
             << " no matching lines were found in text.\n"
             << endl;
        return;
    }

    set<short>::const_iterator
        it = solution->begin(),
        end_it = solution->end();

    for ( ; it != end_it; ++it ) {
        int line = *it;
        // 文本行从 0 开始，别把用户弄糊涂了 ...
        cout << "( " << line+1 << " ) "
             << (*lines_of_text)[line] << '\n';
    }

    cout << endl;
}
```

练习 17.26

TextQuery 类真正需要的是能够从用户那里接受命令行选项。

(a) 为文本查询系统确定的命令行语法。
(b) 给出其他必要的数据成员和成员函数。
(c) 粗略地描述一个命令行设施的实现方法（见 7.8 节例子）。

练习 17.27

作为一个可能的程序设计项目，考虑下列对查询系统的改进：

(a) 允许用一个 string 来表示 AndQuery，比如“Motion Picture Screen Cartoonists”。
(b) 提供这样的支持：根据“几个单词是否出现在同一个句子中，而不是同一行中”的规则来计算结果。
(c) 引入一个历史系统，用户可以通过号码（一个数）引用以前的查询，可能在原来查询的基础上增加新的动作，或者将其与另一个查询组合起来。
(d) 不再显示匹配的数目以及所有的匹配行，而是允许用户为中间查询以及最终查询指定显示行的范围。例如：

```
==> John && Jacob && Astor
```

```
(1)  john ( 3 ) lines match
(2)  jacob ( 3 ) lines match
(3)  john && jacob ( 3 ) lines match
(4)  astor ( 3 ) lines match
(5)  john && jacob && astor ( 5 ) lines match

// 新设施：让用户选择该显示哪个查询
// 用户键入号码
==> display? 3

// 系统然后询问显示多少行
// 回车表示全部，否则用户可以输入一个行数或一段范围
==> how many( return displays all, else enter single line or range) 1-3
```

18

多继承和虚拟继承

在实际的 C++应用程序中，主要的继承模型是“从单个基类的公有继承”模型。一般地，我们可以期望大多数的继承用法都属于这个模型。但是在某些情况下，单继承不足以解决问题，因为：①它不能为一个程序域的抽象建模；②它提供的模型不必要的复杂而且不直观。在这些情况下，多继承，或者它的特例——虚拟继承——就是一个较好的方案。C++为多继承和虚拟继承提供的支持是本章的焦点。

18.1 准备阶段

在了解多继承和虚拟继承之前，让我们先来简要地看一看它们的用法动机。我们的第一个例子来自 3D 计算机图形。但是，在我们引入问题之前，必须先引入**问题域**（problem domain）。

场景在计算机中被表示为**场景图**（scene graph）。场景包含一些几何图形（3D 模型），一个或多个灯光（若没有灯光，则模型被笼罩在黑暗中），相机（若没有相机，我们就不能看到场景）和几个定位元素的转换节点。

着色（rendering）是把灯光和相机信息应用到几何图形上，产生一个 2D 图像用于显示的过程。Rendering 算法的两个主要关注点是：①照亮场景的光源的自然属件；②几何面的材料属性，如颜色、粗糙度、透明度及半透明度。例如，天使的月白色翅膀的羽毛与从它眼睛中落下来的钻石般的泪滴的表现完全不同，虽然它们都在同样的银色灯光下。

为每个场景增加灯光和几何图、重新布置它们以及改变它们的属性是计算机艺术家的一项艰苦的任务。我们的任务是为“在屏幕上操纵场景图”提供交互支持。在我们的工具的当前版本中，假设已经选择用 Open Inventor C++框架（见［WERNECKE94］）来实现底层的场景图，则可以通过子类型机制对它进行扩展，以提供我们自己必需的类抽象。例如，Open Inventor 提供了下列三种从抽象 SoLight 基类派生的内置光源：

```
class SoSpotLight : public SoLight { ... };
class SoPointLight : public SoLight { ... };
class SoDirectionalLight : public SoLight { ... };
```

这里的 So 是一个词汇前辍，被用来提供一个惟一的名字，以区别于其他的图形域名字

（该框架是在引入名字空间之前被设计的）。**点光源**（point light）是向各个方向照射的光源（想像一下太阳）。**有向光**（Directional light）向一个特定的方向照射。**聚光灯**（Spotlight）是一个圆锥型灯光，如用在舞台作品上来照亮舞台的某一部分。

缺省情况下，Open Inventor 用 OpenGL（见［NEIDER93］）着色场景图，显示到屏幕上。虽然这对于交互显示来说已经足够了，但是几乎所有为电影厂使用而生成的图像都是用 Pixar 的 RenderMan 着色的（见［UPSTILL90］）。为了用 RenderMan 着色场景图，我们需要提供自己的特殊的灯光子类型：

```
class RiSpotLight : public SoSpotLight { ... };
class RiPointLight : public SoPointLight { ... };
class RiDirectionalLight : public SoDirectionalLight { ... };
```

正如期望的那样，这是可以工作的。我们的新子类型包含了通过 RenderMan 着色所需要的额外信息。Open Inventor 基类使得我们可同时通过 OpenGl 进行着色。但是当我们需要扩展对于**阴影**（shadow）的支持时，事情就变得很糟糕。

在 RenderMan 中，聚光灯和有向光支持产生阴影（我们将它们称为 SCLS［shadow-capable light sources，有阴影的光源］），而点光源则不是。一般的算法都是要求我们迭代场景中所有的光源，并为每个打开的 SCLS 生成一个 shadow man。问题是，灯光在场景中是以多态的 SoLight 对象被存储的。尽管我们可以把公共数据和必要的操作封装在 SCLS 类中，但是怎样把这个类插入到现有的 Open Inventor 层次结构中还是不清楚。

在 Open Inventor 的 SoLight 子树中，没有合适的位置可以派生出 SCLS 类，进而使得 SdRiSpotLight 和 SdRiDirectionalLight 都可以从它继续往下派生。在缺少多继承的情况下，我们能够做的最好办法就是针对每一种 SCLS 灯光，将一个 SCLS 成员类对象与一个方法组合起来，以调用适当的操作：

```
SoLight *plight = next_scene_light();
if ( RiDirectionalLight *pdilite =
          dynamic_cast<RiDirectionalLight*>( plight ))
     pdilite->scls.cast_shadow_map();
else
     if ( RiSpotLight *pslite =
                    dynamic_cast<RiSpotLight*>( plight ))
          pslite->scls.cast_shadow_map();
// 下略 ...
```

［dynamic_cast 操作符是 RTTI（运行时刻类型识别，Run-Time Type Identification）的一部分。它支持在运行时刻查询一个多态指针或引用所指向的对象的实际类型（RTTI 在 19 章中讨论）］

有了多继承，我们就可以封装 SCLS 子类型，使我们的代码不用随着每一种 SCLS 光源的增加和删除而修改。如图 18.1 所示：

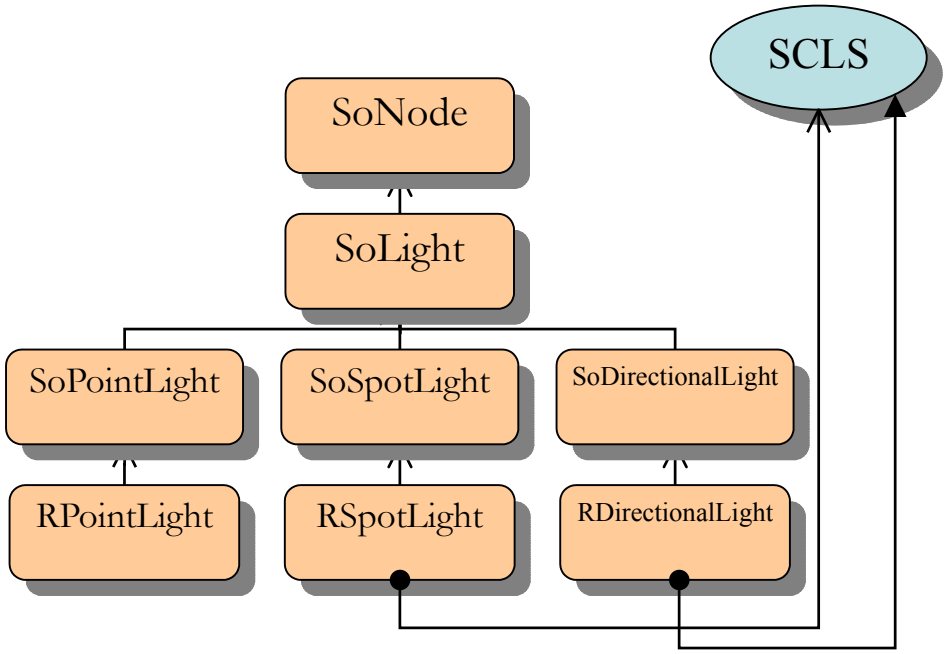


图 18.1 多继承 Light 层次结构

```
class RiDirectionalLight :
     public SoDirectionalLight, public SCLS { ... };
class RiSpotLight :
     public SoSpotLight, public SCLS { ... };

// ...
SoLight *plight = next_scene_light();
if ( SCLS *pscls = dynamic_cast<SCLS*>(plight))
     pscls->cast_shadow_map();
```

这还不是最完美的方案。如果我们能够访问 Open Inventor 的源代码，则可以通过向 SoLight 增加一个 SCLS 指针成员以及对 cast_shadow_map()操作的支持，来避开多继承，从而达到同样的目的：

```
class SoLight : public SoNode {
public:
     void cast_shadow_map()
          { if ( _scls ) _scls->cast_shadow_map(); }
     // ...
     protected:
          SCLS *_scls;
};
// ...
SdSoLight *plight = next_scene_light();
plight->cast_shadow_map();
```

在实际的应用程序中，最广泛使用多继承（和虚拟继承）的便是标准 C++的输入/输出 iostream 库。对用户可见的、两个最主要的 iostream 类是 istream 类（输入的）和 ostream 类（输出的）。这两个类共同的属性包括：

1．格式状态信息（以十进制、八进制或十六进制表示整数值，以及用小数或科学计数法表示浮点数等等）。

2．条件状态信息（该流对象处于正常还是失效状态等等）。

3．本地化信息（先显示日还是月，如 7/4/76 等等）

4．用来存放被读入或被写出的数据的实际缓冲区。

这些公共属性被抽取到一个抽象的 ios 基类中，而 istream 和 ostream 类部从它派生。

iostream 类是多继承的第二个例子。它支持对同一文件进行读和写的操作，且从 istream 和 ostream 类派生。但不幸的是，在缺省情况下，虽然它继承了 ios 基类的两份单独的实例，然而我们却很难管理也不需要它们。

虚拟继承为这种问题（即，继承了多个基类实例，但是只需要一份单独的共享实例）提供了解决方案。简化的 iostream 类层次结构如图 18.2 所示：

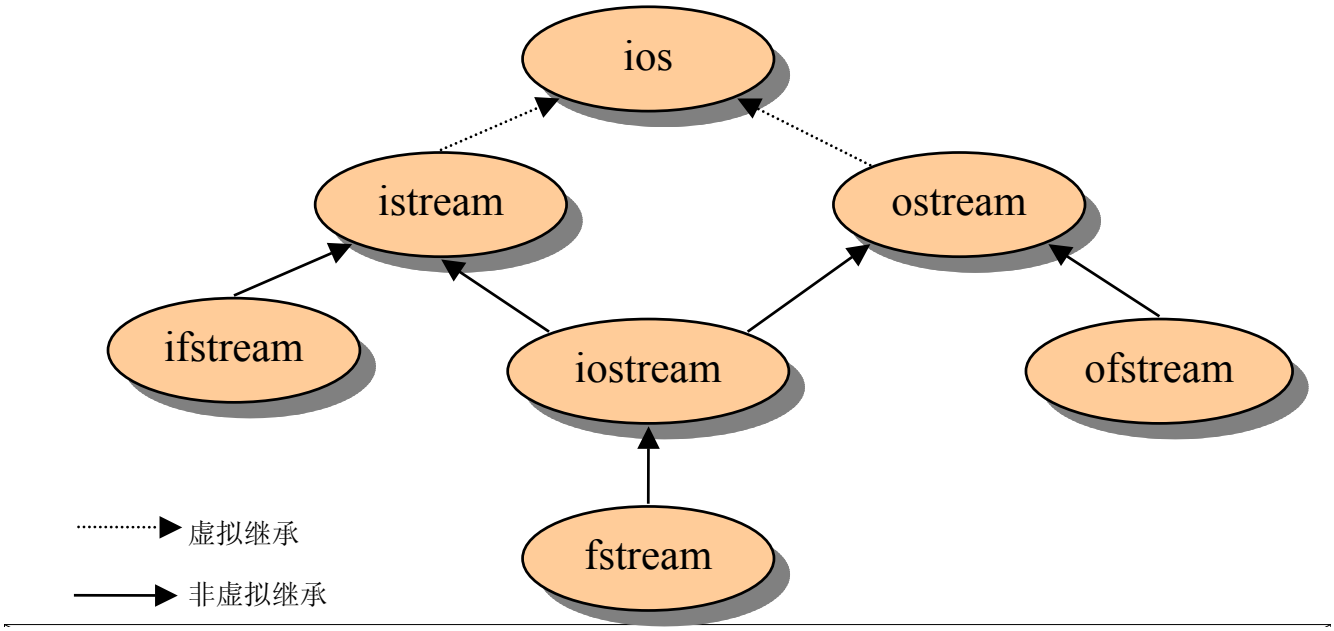


图 18.2 虚拟继承 iostream 层次结构（简化的）

支持分布式对象计算是虚拟继承和多继承的另一个实际例子。详细的讨论和说明见 Douglas Schmidt 和 Steve Vinoski 在［LIPPMAN96b］中的关于分布式对象计算的文章。多继承和虚拟继承是本章的主要焦点。现在我们的焦点是多继承和虚拟继承的用法和行为。在本书的姐妹篇《Inside the C++ Object Model》中介绍了更高级的性能和设计主题。

在下面的讨论中，我们将选择一个教学例子——动物园里动物的层次结构例子。动物园中的动物存在于不同的抽象级别上。当然，有独立的动物，如 Ling-ling、Mowgli 和 Balou。每个动物属于一个种（specips）：例如，Ling-ling 属于一个大熊猫种。每个种又是一个科（family）的成员，大熊猫是熊科的一个成员。虽然我们将在 18.5 节谈到，很长时间以来这种关系是动物学分类领域争论的焦点，每个科又是一个动物域（kingdom）的成员——在这种情况下，是一个特定的动物园中更为有限的域。

每一层抽象都含有各自的数据和操作，但它们都支持广泛的用户范围。例如，抽象的 ZooAnimal 类含有所有动物公共的信息，并且为所有的一般查询提供公有接口。Bear 类会有 Bear 科惟一的信息，等等。

除了实际的动物类，还有一些辅助类，它们封装了各种抽象，如濒临灭绝的动物。例如，在 Panda 类的实现中。Panda 是从 Bear 和 Endangered（濒临灭绝）多重派生而来的。

18.2 多继承

为支持多继承，一个类的派生表：

```
class Bear : public ZooAnimal { ... };
```

被扩展成支持逗号分割的基类表。例如：

```
class Panda : public Bear, public Endangered { ... };
```

每个被列出的基类还必须指定其访问级别：public、protected 或 private 之一。与单继承一样，只有当一个类的定义已经出现后，它才能被列在多继承的基类表中。

对于一个派生类的基类的数目，C++没有限制。实际中，看起来两个基类是最常见的，一个基类常常用于表示一个公有抽象接口，第二个基类提供私有实现（虽然我们前面的两个例子都没有说明这一点）。从三个或者更多个直接基类继承而来的派生类遵循 mixin-based 设计风格，其中每个基类都表示该派生类完整接口的一个方面（facet）。

在多继承下，派生类含有每个基类的一个基类子对象（关于派生类的基类子对象的讨论见 17.3 节）。例如，当我们写：

```
Panda ying_yang;
```

时，ying_yang 由一个 Bear 类子对象（它又含有一个 ZooAnimal 基类子对象）、一个 Endangered 类子对象，以及在 Panda 类中声明的非静态数据成员组成（见图 18.3）

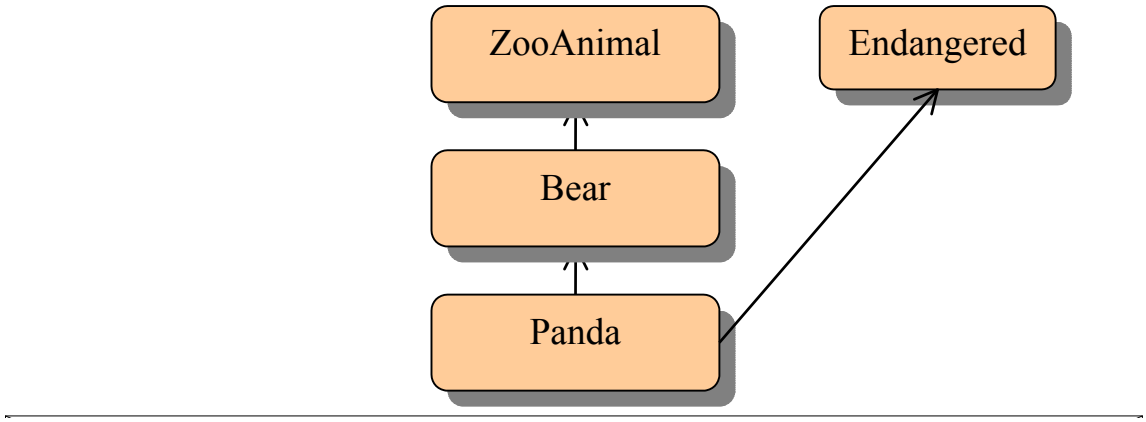


图 18.3 多继承 Panda 层次结构

基类构造函数被调用的顺序以类派生表中声明的顺序为准。例如，对 ying_yang 来说，构造函数被调用的顺序是：Bear 构造函数（因为 Bear 是从 ZooAnimal 派生的，所以在 Bear 构造函数执行之前，ZooAnimal 的构造函数先被调用），Endangered 构造函数，然后是 Panda 构造函数。

正如 17.4 节所讨论的，构造函数调用顺序不受“基类在成员初始化表中是否存在以及被列出的顺序”的影响。即，如果 Bear 缺省构造函数被隐式调用，也就是说，它没有出现在成员初始化表中，如下所示：

```
// Bear 缺省构造函数在
// Endangered 的双参数
```

```
// 构造函数之前被调用
Panda::Panda()
        : Endangered( Endangered::environment,
    Endangered::critical )
{ ... }
```

那么，Bear 的缺省构造函数仍然在显式列出的双参数 Endangered 构造函数之前被调用。

类似地，析构函数调用顺序总是与构造函数顺序相反。在我们的例子中，析构函数调用顺序是：~Panda()、~Endangered()、~Bear()，最后是~ZooAnimal()。

在单继承下，如 17.3 节所示，基类的 public 和 protected 成员可以直接被访问，就像它们是派生类的成员一样，对多继承这也是正确的。但是在多继承下。派生类可以从两个或者更多个基类中继承同名的成员。然而在这种情况下，直接访问是二义的，将导致编译时刻错误。

但是，这个编译时刻错误不是由于“对两个成员的非限定修饰访问存在潜在的二义性”所触发的，而是由于“企图真正访问这两个成员”（见 17.4.4 节讨论）而触发的。例如，如果 Bear 和 Endangered 都定义了一个成员函数 print()，则如下语句：

```
ying_yang.print( cout );
```

将导致编译时刻错误，即使这两个通过继承得到的成员函数定义了不同的参数类型：

```
Error: ying_yang.print( cout ) -- ambiguous, one of
                  Bear::print( ostream& )
                  ndangered::print( ostream&, int )
```

原因在于，继承得到的成员函数没有构成派生类中的重载函数（见 17.3 节的讨论）。因此，对于 print()调用，编译器在解析的时候，只是使用了针对 print 的名字解析，而不是使用“基于传递给 print()的实际实参类型的重载解析”（我们将在 18.4 节看到它是怎样被解析的）。

在单继承下，如果有必要的话，派生类的指针、引用或对象将自动被转换成公有派生的基类的指针、引用或对象。对于多继承，这也是正确的。例如，一个 Panda 指针、引用或对象可以被转换成 ZooAnimal、Bear 或 Endangered 类的指针、引用或对象。例如：

```
extern void display( const Bear& );
extern void highlight( const Endangered& );

Panda ying_yang;

display( ying_yang );  // ok
highlight( ying_yang );  // ok

extern ostream&
    operator<<( ostream&, const ZooAnimal& );

cout << ying_yang << endl;   // ok
```

但是，在多继承下，二义转换的可能性非常大。例如，考虑下列两个函数：

```
extern void display( const Bear& );
```

```
extern void display( const Endangered&);
```

用非限定修饰的 Panda 对象调用 display()：

```
Panda ying_yang;
display( ying_yang );  // 错误：二义性
```

将导致下列一般形式的编译时刻错误:

```
Error: display( ying_yang ) -- ambiguous, one of
                        extern void display( const Bear&);
                        extern void display( const Endangered&);
```

编译器没有办法区分应该使用哪一个直接基类（它们都可以从派生类转换而来）。每个转换的可应用程度都是等价的（我们将在 18.4.1 节了解这是如何被解析的）。

为了了解多继承怎样影响虚拟函数机制，让我们为每个 Panda 的直接基类定义一组虚拟函数（虚拟函数在 17.2 节介绍，在 17.5 节详细讨论。）

```
class Bear : public ZooAnimal {
public:
    virtual ~Bear();
    virtual ostream&print( ostream&) const;
    virtual string isA() const;
    // ...
};

class Endangered {
public:
    virtual ~Endangered();
    virtual ostream&print( ostream&) const;
    virtual void highlight() const;
    // ...
};
```

现在，我们来定义 Panda，它提供了自己的 print()实例、析构函数，并引入了一个新的虚拟函数 cuddle()，如下：

```
class Panda : public Bear, public Endangered
{
public:
    virtual ~Panda();
    virtual ostream&print( ostream&) const;
    virtual void cuddle();
    // ...
};
```

可以直接从 Panda 对象调用的虚拟函数集如下表所示：

表 18.1 活动的 Panda 虚拟函数

虚拟函数名	活动实例
destructor（构析函数）	Panda::~Panda()
print(ostream&) const	Panda::print(ostream&)

虚拟函数名	活动实例
isA() const	Bear::isA()
highlight() const	Endangered::highlight()
cuddle()	Panda::cuddle()

当用 Panda 类对象的地址初始化或赋值 Bear 或 ZooAnimal 指针或引用时，Panda 接口中“Panda 特有的部分”以及“Endangered 部分”就都不能再被访问。例如：

```
Bear *pb = new Panda;
pb->print( cout );      // ok: Panda::print(ostream&)
pb->isA();              // ok: Bear::isA()
pb->cuddle();           // 错误：不是 Bear 接口的部分
pb->highlight();        // 错误：不是 Bear 接口的部分
delete pb;              // ok: Panda::~Panda()
```

（注意，如果 Panda 对象已经被赋值给一个 ZooAnimal 指针，则上述这一组调用的解析结果相同。）

类似地，当用 Panda 类对象的地址初始化或赋值 Endangered 指针或引用时，Panda 接口中“Panda 特有的部分”以及“Bear 部分”都不能再被访问。例如：

```
Endangered *pe = new Panda;
pe->print( cout );// ok: Panda::print(ostream&)

// 错误：不是 Endangered 的接口部分
pe->isA();

// 错误：不是 Endangered 的接口部分
pe->cuddle();

pe->highlight();        // ok: Endangered::highlight()
delete pe;              // ok: Panda::~Panda()
```

无论我们删除对象所使用的指针类型是什么，虚拟析构函数的处理都是一致的。例如，已知：

```
// ZooAnimal *pz = new Panda;
delete pz;

// Bear *pb = new Panda;
delete pb;

// Panda *pp = new Panda;
delete pp;

// Endangered *pe = new Panda;
delete pe;
```

在上述例子中，析构函数的调用顺序完全相同。析构函数调用顺序与构造函数的顺序相

反：Panda 析构函数被通过虚拟机制调用。在 Panda 析构函数执行之后，依次静态调用 Endangered、Bear 和 ZooAnimal 析构函数。

通过多继承得到的派生类，它的按成员初始化和赋值与单继承下的派生类相同（见 17.6 节讨论）。例如，已知 Panda 的声明：

```
class Panda : public Bear, public Endangered
{ ... };
```

ling_ling 的按成员初始化：

```
Panda yin_yang;
Panda ling_ling = yin_yang;
```

调用了 Bear 拷贝构造函数（但是，因为 Bear 是从 ZooAnimal 派生来的，所以在执行 Bear 拷贝构造函数之前，先调用 ZooAnimal 拷贝构造函数），然后再调用 Endangered 拷贝构造函数，以及执行 Panda 拷贝构造函数。按成员赋值与此类似。

练习 18.1

下列声明哪些是错误的？请说明原因。

```
(a)  class CADVehicle : public CAD, Vehicle { ... };
(b)  class DoublyLinkedList:
     public List, public List { ... };
(c)  class iostream:
     private istream, private ostream { ... };
```

练习 18. 2

已知下列类层次结构，它们都定义了缺省的构造函数：

```
class A { ... };
class B : public A { ... };
class C : public B { ... };
class X { ... };
class Y { ... };
class Z : public X, public Y { ... };
class MI : public C, public Z { ... };
```

下面定义的构造函数的执行顺序是什么？

```
MI mi;
```

练习 18.3

已知下列类层次结构，它们都定义了缺省的构造函数：

```
class X { ... };
class A { ... };
class B : public A { ... };
class C : private B { ... };
class D : public X, public C { ... };
```

下列哪些转换是不允许的？

```
D *pd = new D;
(a) X *px = pd; (c) B *pb = pd;
(b) A *pa = pd; (d) C *pc = pd;
```

练习 18.4

已知下列类层次结构，以及虚拟函数集：

```
class Base {
public:
     virtual ~Base();
     virtual ostream&print();
     virtual void log();
     virtual void debug();
     virtual void readOn();
     virtual void writeOn();
     // ...
};

class Derived1 : virtual public Base {
public:
     virtual ~Derived1();
     virtual void writeOn();
     // ...
};

class Derived2 : virtual public Base {
public:
     virtual ~Derived2();
     virtual void readOn();
     // ...
};

class MI : public Derived1, public Derived2 {
public:
     virtual ~MI();
     virtual ostream&print();
     virtual void debug();
     // ...
};
```

下列语句调用哪一个函数实例？

```
Base *pb = new MI;
(a) pb->print();   (c) pb->readOn();    (e) pb->log();
(b) pb->debug();   (d) pb->writeOn();   (f) delete pb;
```

练习 18.5

使用练习 18.4 定义的类层次结构，请指出当通过(a) pd1 和(b) d2 调用时，哪些虚拟函数是活动的：

```
(a)  Derived1 *pd1 = new MI;
(b)  MI obj;
     Derived2 d2 = obj;
```

18.3 public、private 和 protected 继承

public 派生被称为**类型继承**（type inheritance）。派生类是基类的子类型，它改写了基类中所有与类型相关的成员函数，而继承了共享的成员函数。派生类往往反映了一种 is-a（是一种）关系，它提供了“较一般的基类”的一种特化。Bear 是一种 ZooAnimal。AudioBook 是一种 LibBook，它们都是一种 LibraryLendingMaterial。我们说 Bear 是 ZooAnimal 的子类型，Panda 也一样。类似地，我们说 AndioBook 是 LibBook 的子类型，它们都是 LibraryLendingMaterial 的子类型。子类型在“程序中任何希望使用公有基类类型”的地方都可以被透明地替换掉，并且继续正确执行（当然前提是，子类型必须被正确实现）。到现在为止，所有关于继承的例子都反映了子类型继承。

private 派生被称为**实现继承**（implementation inheritance），派生类不直接支持基类的公有接口。相反，当它提供自己的公有接口时，它希望重用基类的实现。为说明涉及到的主题，我们来实现一个 PeekbackStack。

PeekbackStack 持用 peekback()方法查看栈：

```
bool
PeekbackStack::
peekback( int index, type &value ) { ... }
```

如果 peedback()返回 true，则 value 含有在 index 处的元素。如果 peekback()返回 false，则 index 无效，且 value 被设置为栈顶的元素。

在 peedback()的实现中，有两个可能出错的地方：

1．PeekbackStack 抽象的实现。即，我们正确地实现了它的行为吗？

2．底层表示方式的实现。即，我们是否正确地管理内存的分配和释放，以及对象的拷贝等等？

栈一般用数组或元素的链表来实现（标准库的栈 stack 缺省情况下由 deque 组成，但如果我们愿意的话，可以指定 vector——见第 6 章）。我们想要的是一个完全保证的（至少是一个经过测试的、且完全支持的）数组或者 list 的实现，可以直接把它插入到 PeekbackStack 中。如果可以，我们就可以把精力集中在栈的正确行为上。

正巧我们有一个 IntArray 类，是在 2.3 节实现的（是的。为了讨论的目的，我们忽略了标准库 deque，也不考虑为 int 之外的数据类型提供支持）。那么，问题就是：怎样在 PeekbackStack 的实现中更好地重用 IntArray 类。当然，第一个想法是继承。（注意，我们需要修改 IntArray 类，将其成员从 private 改为 protected。）下面是实现：

```
#include <IntArray.h>

class PeekbackStack : public IntArray {
private:
     const int static bos = -1;
```

```
public:
    explicit PeekbackStack( int size )
        : IntArray( size ), _top( bos ){}

    bool empty() const { return _top == bos; }
    bool full() const { return _top == size()-1; }

    int top() const { return _top; }
    int pop() {
        if ( empty() )
            /* 处理错误情况 */ ;
            return ia[ _top-- ];
    }

    void push( int value ) {
        if ( full() )
            /* 处理错误情况 */ ;
            ia[ ++_top ] = value;
    }
    bool peekback( int index, int &value ) const;
private:
    int _top;
};
inline bool
PeekbackStack::
peekback( int index, int &value ) const
{
    if ( empty() )
        /* 处理错误情况 */ ;

    if ( index < 0 || index > _top )
    {
        value = ia[ _top ];
        return false;
    }
    value = ia[ index ];
    return true;
}
```

这正好是我们所希望的——而且更多。使用我们新的 PeekbackStack 类的程序代码也可能会不正确地使用 IntArray 基类的公有接口。例如：

```
extern void swap( IntArray&, int, int );
PeekbackStack is( 1024 );

// 喔！误用了 PeekbackStack
swap(is,i,j);
is.sort();
is[0] = is[512];
```

PeekbackStack 类抽象应该保证，在访问它所包含的元素时遵守“先进先出”的行为规范。但是 IntArray 额外的接口严重地破坏了对这种行为规范的保证。

问题是，public 派生定义了一种 is-a 关系。但 PeekbackStack 不是一种 IntArray 而是把 IntArry 作为实现的一部分。IntArray 的公有接口并不是 PeekbackStack 类公有接口的一部分。PeekbackStack 希望重用 IntArray 类的实现，但是，PeekbackStack 不是 IntArray 的子类型。

一个 private 基类反映了一种“并非基于子类型关系”的继承形式。基类的整个公有接口在派生类中变成 private。除了在派生类的友元和成员函数中，前面对于 PeekbackStack 实现的所有误用现在都是非法的了。

前面的 PeekbackStack 定义惟一需要改变的是在类的派生表中用关键字 private 取代 public，则类定义内部的关键字 private 和 public 无需改变：

```
class PeekbackStack : private IntArray { ... };
```

18.3.1 继承与组合（composition）

作为 IntArray 类私有派生的 PeekbackStack 类可以工作——但是，这样做是必要的吗？在这种情况下，通过继承有什么收获吗？其实没有。

对于支持 is-a 子类型关系来说，public 继承是一个很有力的机制。但是，PeekbackStack 的实现表示的是，它和 IntArray 类之间的 has-a（有一个）关系。PeekbackStack 类把 IntArray 作为实现的一部分。has-a 关系一般由是组合（composition）而不是继承来支持。实现组合很容易，只须使一个类成为另一个类的成员。在本例中，就是让 IntArray 作为 PeekbackStack 的成员。下面是通过 has-a 关系建立起来的 PeekbackStack 实例：

```
class PeekbackStack {
private:
    const int static bos = -1;
public:
    explicit PeekbackStack( int size ) :
        stack( size ), _top( bos ){}
    bool empty() const { return _top == bos; }
    bool full() const { return _top == stack.size()-1; }
    int top() const { return _top; }

    int pop() {
        if ( empty() )
        /* 错误处理 */ ;
            return stack[ _top-- ];
    }

    void push( int value ) {
        if ( full() )
            /* 错误处理 */ ;
            stack[ ++_top ] = value;
    }

    bool peekback( int index, int &value ) const;
    private:
        int _top;
        IntArray stack;
};
```

```
inline bool
PeekbackStack::
peekback( int index, int &value ) const
{
    if ( empty() )
        /* 错误处理 */ ;
    if ( index < 0 || index > _top )
    {
        value = stack[ _top ];
        return false;
    }

    value = stack[ index ];
    return true;
}
```

下面给出了一个关于“在包含 has-a 关系的类设计中”是否使用组合或私有继承的广泛建议：

- 如果我们希望改写一个类的虚拟函数，则必须使用私有继承。
  如果我们希望一个类能够引用“一个包含多种可能类型的层次结构”中的一个类，那么就必须通过引用使用组合，我们将在 18.3.4 详细讨论。
- 和 PeekbackStack 类一样，如果只是希望简单地重用实现，则按值组合比继承更好。
  如果希望对象的迟缓型分配，则按引用（使用一个指针）组合通常是一个不错的设计选择。

18.3.2 免除（exempting）个别成员的私有继承影响

在上述以私有方式继承 IntArray 的 PeekbackStack 中，IntArray 类的所有 protected 和 public 成员全被继承为 PeekbackStack 的私有成员。但是，如果 PeekbackStack 的用户能够查询 PeekbackStack 实例的大小，那么这项功能还是很有用的：

```
is.size();
```

类的设计者可以针对基类的个别成员，使其免除非公有派生的影响。例如，下面语句免除了 IntArray 的成员函数 size()：

```
class PeekbackStack : private IntArray {
public:
    // 维持公有访问级别
    using IntArray::size;
    // ...
};
```

免除个别成员的另一个原因是允许后续的派生类访问私有基类的 protected 成员。例如，假设用户希望 PeekbackStack 子类型可以动态增长。为了提供这样的能力，从 PeekbackStack 派生的类需要访问 IntArray 的 protected 元素 ia 和 size：

```
template <class Type>
class PeekbackStack : private IntArray {
public:
```

```
        using IntArray::size;
        // ...

    protected:
        using IntArray::_size;
        using IntArray::ia;
        // ...
    };
```

派生类只能将继承得到的成员恢复到原来的访问级别，该访问级别不能比基类中原来指定的级别更严格或更不严格。

一种常见的多继承形式是，继承一个类的公有接口和第二个类的私有实现。例如，Booch Components（一个 C++类库）包含如下实现的可增长的 Queue（更详细的内容见 Michael Vilot 和 Grady Booch 在［LIPPMAN96b］中的文章）：

```
template < class item, class container >
class Unbounded_Queue:
    private Simple_List< item >,   // 实现
    public Queue< item >// 接口
{ ... };
```

18.3.3 protected 继承

第三种派生形式是 protected 继承。在 protected 继承下，基类的所有公有成员都成为派生类的 protected 成员。这意味着它们可以被“后来从该类派生的类”访问，但是不能在层次结构之外被访问。例如，如果希望 PeekbackStack 从 Stack 派生，则私有派生：

```
// 喔！这不支持 PeekbackStack 的后续派生
// 所有 IntArray 成员现在都是 private
class Stack : private IntArray { ... };
```

有些过于严格，因为“在 Stack 中使 IntArray 成员都是 private 的”会禁止后来的派生类访问这些成员。为了支持：

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

Stack 必须是 IntArray 的 protected 派生：

```
class Stack : protected IntArray { ... };
```

18.3.4 对象组合

实际有两种形式的对象组合：

1．**按值组合**（Composition by value），类的实际对象被声明为一个成员，正如前两个小节修订 PeekbackStack 实现时所解释的。

2．**按引用组合**（Composition by reference），通过类对象的引用或指针成员间接指向一个对象。

按值组合提供了对于“对象生命期和拷贝语义”的自动管理，并且使得对于对象本身的访问更加有效、更加直接。在什么情况下按引用组合更合适呢？

例如，我们决定通过组合而不是继承来表示 Endangered 更好一些。则应该在 ZooAnimal

中直接定义 Endangered 对象，还是通过指针或引用间接引用它呢？让我们考虑：①是否所有的 ZooAnimal 都表现了这个特性？②如果不，是否它随时间而改变，即，这个特性是否可能随时间的变化而增加或删除？

如果对问题①的回答是所有 ZooAnimal 对象都表现了这个特性，则按值组合一般比较好。［一般情况下，对于大型类对象来说，按值组合并不一定是最有效的表现策略，尤其是在它们经常被拷贝的情况下。当与引用计数策略联合使用时，按引用组合可以使我们避免不必要的拷贝，这被称为“写时拷贝（copy on write）”（代价是增加了管理对象的复杂度）。然而，对这项技术的讨论超出了 C++语言入门的范围，细致的解释可以在［KOENIG97］的第 6、7 章中找到。］

如果对问题①的回答是只有某些 ZooAnimal 表现了这个特性，则按引用组合一般比较好。（为什么并不濒临灭绝的对象也要带上一个 Endangered 类对象？）

因为 Endangered 类对象可能不存在，所以我们必须用指针而不是引用来表示它。（设置为 0 的指针被认为不指向任何对象，引用必须总是指向一个对象。3.6 节详细说明了这种区另。）

如果对问题②的回答是“是的”。则我们必须提供运行时刻访问函数来插入和删除 Endangered 对象。

在我们的例子中，Endangered（濒临灭绝的）是 ZooAnimal 子类型中少数对象的特性。除此之外，至少在理论上，它是一种可以被转换的条件，我们的 Panda 可能有一天不再受灭绝的威胁：

```
class ZooAnimal {
public:
    // ...
    const Endangered* Endangered() const;
    void addEndangered( Endangered* );
    void removeEndangered();
    // ...
protected:
    Endangered *_endangered;
    // ...
};
```

如果我们的应用程序希望在多种平台上运行，那么，把依赖平台的信息封装在一个抽象类层次结构中会更有用，这使得应用程序可以针对一个“与平台无关的抽象接口”进行编程。例如，为了在 UNIX 和 PC 下都能显示 ZooAnimal 对象，我们可以定义一个 DisplayManager 类层次结构：

```
class DisplayManager{ ... };
class DisplayUnix : public DisplayManager{ ... };
class DisplayPC : public DisplayManager{ ... };
```

ZooAnimal 不是一种 DisplayManager，而是有一个 DisplayManager 实例。通过组合而不是继承，ZooAnimal 包含一个 DisplayManager 对象。我们的第一个问题是按值组合还是按引用组合？

通过按值组合，我们无法表示一个 DisplayManager 对象，也无法通过这个对象引用到一

个实际的 DisplayUnix 或 DisplayPC 对象。只有一个 DisplayManager 引用或指针类型的 ZooAnimal 数据成员才能使我们在运行时刻操纵 DisplayManager 的子类型。即，只有按引用组合才能支持面向对象的程序设计（详细说明见［LIPPMAN96a］）。

第二个问题是，怎样决定把 ZooAnimal 的成员声明成 DisplayManager 引用还是指针？

1．如果在 ZooAnimal 对象被创建时，提供了实际的 DisplayManager 子类型，而且它不会随着程序的执行过程而改变，那么只有这样才可以把 DisplayManager 成员声明为一个引用。

2．如果应用了**迟缓型分配策略**（lazy allocation strategy），直到真正要显示一个对象时才分配实际的 DisplayManager 子类型，那么必须把成员表示成指针，并初始化为 0。

3．如果我们希望在运行期间切换（toggle）显示模式，则必须把 DisplayManager 成员表示成指针，并且将它初始化为 0。切换的意思是允许用户随着程序的执行过程而确定 DisplayManager 子类型，或者改变 DisplayManager 子类型。

在实践中，当然不可能应用程序的每一个 ZooAnimal 子对象都要求用它自己的 DisplayManager 子类型来显示自己。在这种情况下，最可能的设计选择是 ZooAnimal 的静态 DisplayManager 指针。

练习 18.6

请指出下列选项中哪些是类型继承，哪些是实现继承。

```
(a)  Queue : List
(b)  EncryptedString : String
(c)  Gif : FileFormat
(d)  Circle : Point
(e)  Dqueue : Queue, List
(f)  DrawableGeom : Geom, Canvas
```

练习 18.7

请用标准库的 deque 代替 18.3.1 节中的 PeekbackStack 的 Array 成员，并写个小程序来应用它。

练习 18.8

请对照按值组合与按引用组合，给出每种用法的例子来说明你的讨论。

18.4 继承下的类域

每个类都维护自己的类域，并在类域中定义成员名和嵌套类型名（详细讨论见 13.9 节和 13.10 节）。在继承下，派生类的域被嵌套在直接基类的域中。如果一个名字在派生类域中没有被解析出来，则编译器在外围基类域中查找该名字的定义。

正是这种“继承下的类域的层次嵌套”使得基类的成员能被直接访问，仿佛它们就是派生类的成员。让我们先看一些单继承的例子，然后再将讨论扩展到考虑多继承的情形。已知下面简化的 ZooAnimal 类定义，

```
class ZooAnimal {
```

```
public:
    ostream &print( ostream&) const;

    // 为了向外域暴露设为 public
    string is_a;
    int ival;
private::
    double dval;
};
```

以及简化的派生类 Bear 的定义，如下所示：

```
class Bear : public ZooAnimal {
public:
    ostream &print( ostream&) const;
    int mumble( int );

    // 为了向外域暴露设为 public
    string name;
    int ival;
};
```

当我们这样写如下语句时：

```
Bear bear;
bear.is_a;
```

名字解析的实际过程如下：

1．Bear 是 Bear 类的对象。为了查找 is_a，首先在 Bear 类域中进行。没有找到。

2．由于 Bear 是从 ZooAnimal 派生来的，所以接下来在 ZooAnimal 类域中查找 is_a 的声明。发现它是 ZooAnimal 基类的一个成员。该引用被成功解析。

虽然基类的成员可以像派生类的成员那样被直接访问，但是，实际上，它保持了自己与基类之间的成员关系。一般，我们并不关心实际上哪个对象含有这个成员。当基类和派生类成员共享同一个名字时，我们就必须要考虑这一点了。例如，当我们写：

```
bear.ival;
```

访问的 ival 实例是前述查找过程中第 1 步找到的 Bear 成员。

实际上，与基类成员同名的派生类成员隐藏了对基类成员的直接访问。为了访问基类成员，我们必须使用类域操作符来限定修饰它：

```
bear.ZooAnimal::ival;
```

这引导编译器直接在 ZooAnimal 类域中查找 ival 的声明。

让我们用一个有点荒谬的例子来说明类域操作符的用法（所谓有点荒谬，意思是你永远不应该在真正的产品代码中真的这样做！）：

```
int ival;
int Bear::mumble( int ival )
{
    return ival +      // 参数实例
        ::ival +       // 全局实例
        ZooAnimal::ival +
```

```
        Bear::ival;
}
```

对 ival 未限定修饰的引用被解析为形式参数实例，（如果 mumble()中没有定义 ival，则访问 Bear 的 ival 成员实例。如果在类 Bear 中也没有定义 ival，则访问 ZooAnimal 的 ival 成员实例。如果在 ZooAnimal 类中也没有定义 ival，则访问全局的 ival 实例。）

编译器总是先解析一个类成员，然后再判断该访问是否合法，这可能粗看起来似乎违反直觉。例如，考虑修改后的 mumble()实现：

```
int dval;
int Bear::mumble( int ival )
{
    // 错误：解析为 ZooAnimal::dval 的私有成员函数
    return ival + dval;
}
```

我们或许会说，查找算法应该把它解析为第一个合法的、可访问的标识符，而不是最直接的标识符。但是，它没有。在这个例子中，查找算法执行如下：

1．在 Bear 成员函数的局部域中定义了 dval 吗？没有；

2．在 Bear 类中定义了 dval 吗？没有；

3．在基类 ZooAnimal 域中定义了 dval 吗？是的。该引用被解析为这个实例。

既然实例已经被解析了，那么编译器将检查对该实例的访问是否合法。在本例中，是不合法的，dval 是私有的数据成员，不能在 mumble()中被直接访问。要想正确地（可能正符合意图）解析，要求显式的域操作符：

```
return ival + ::dval;  // ok
```

在考虑成员的访问级别之前对它进行解析，基本的出发点是要防止程序语义的微妙改动，而这种语义一般与成员访问级别无关。例如，考虑下列调用：

```
int dval;
int Bear::mumble( int ival )
{
    foo( dval );
    // ...
}
```

如果 foo()是重载函数，那么把 ZooAnimal::dval 从 private 变为 protected 成员可能大大地改变了 mumble()中的整个调用顺序——可能类的设计者在改变成员的访问级别时完全不知道这些改变。

在基类和派生类之间名字相同并且原型也相同的成员函数，其行为与同名的数据成员一样，派生类的成员在派生类域中隐藏了基类的成员。为了调用基类的成员，我们必须使用基类域操作符。例如：

```
ostream& Bear::print( ostream &os ) const
{
    // 调用 ZooAnimal::print(os)
    ZooAnimal::print( os );

    os << name;
```

```
        return os;
    }
```

18.4.1 多继承下的类域

多继承的引入怎样影响类域的查找过程呢？所有直接基类被同时查找，如果从两个或多个基类继承了同名的成员，则增加了二义引用的可能性。我们先来看几个例子，了解一下二义性怎样出现以及解析它的不同策略。首先，考虑下面一组类：

```
class Endangered {
public:
    ostream&print( ostream&) const;
    void highlight();
    // ...
};

class ZooAnimal {
public:
    bool onExhibit() const;
    // ...
private:
    bool highlight( int zoo_location );
    // ...
};

class Bear : public ZooAnimal {
public:
    ostream&print( ostream&) const;
    void dance( dance_type ) const;
    // ...
};
```

当用多继承派生 Panda 类时：

```
class Panda : public Bear, public Endangered {
public:
    void cuddle() const;
    // ...
};
```

虽然从 Bear 和 Endangered 基类中继承 print()和 highlight()都存在潜在的二义性，但是，在真正引用这些函数之前，编译器并不会报告二义错误。

两个通过继承得到的 print()成员的二义性是显然的，但是，在两个 highlight()之间的冲突有些令人吃惊（毫无疑问，这正是目的所在）。毕竟，两个实例有不同的访问级别和不同的函数原型。而且 Endangered 中的实例是两个直接基类之一的成员，而 ZooAnimal 中的实例是第二个直接基类的基类的成员。

没关系（我们将会看到，实际上是有关系的，但是，是在虚拟继承下），Bear 继承了 ZooAnimal 的私有 highlight()成员。虽然它在 Bear 或 Panda 中是可见的，但是对它进行调用却是非法的。Panda 继承了两个名为 highlight 的成员，并且都是可见的。因此，任何无限定修饰的引用都会引起编译时刻错误。

在查找标识符时，首先从出现该引用的直接域中开始。例如，如果写：

```
int main()
{
    Panda yin_yang;
    yin_yang.dance( Bear::macarena );
}
```

那么直接域就是 yin_yang 所属的类的域——Panda。如果我们写：

```
void Panda::mumble()
{
    dance( Bear::macarena );
    // ...
}
```

那么直接域就是成员函数 mumble()的局部域。当然，如果找到了一个声明，则该标识符就被解析，且查找过程结束。否则，查找外围的域。

在多继承下，查找过程对每个基类的继承子树同时进行检查——在我们的例子中，包括 Endangered 和 Bear/ZooAnimal 子树。如果只在其中一个基类子树中找到了声明，则该标识将被解析，查找算法结束。这正是在 dance()调用中所发生的：

```
// ok: Bear::dance()
yin_yang.dance( Bear::macarena );
```

如果在两个或多个基类子树中都找到了声明，则表示这个引用是二义的，会产生一个编译时刻错误消息。这正是非限定修饰调用 print()的情况：

```
int main()
{
    // 错误：二义
    // Bear::print( ostream&) const
    // Endangered::print( ostream&) const
    Panda yin_yang;
    yin_yang.print( cout );
}
```

在程序层次上，解决成员二义性的方案是用类域操作符显式地限定修饰“期望被调用的实例”。例如：

```
int main()
{
    // ok: 但不推荐
    Panda yin_yang;
    yin_yang.Bear::print( cout );
}
```

虽然这样做已经能解决问题，但它并不是最令人满意的解决方案。原因是，现在必须由用户来决定对于 Panda 类正确的行为是什么。这种负担不应该被强加在类的用户身上，而应该是类的设计者要注意这些细节。较好的解决方案是 Panda 类自己解决它的继承层次结构中的二义性。最简单的方式是，在提供预期行为的派生类中定义一个同名实例。例如：

```
inline void Panda::highlight() {
    Endangered::highlight();
}

inline ostream&
```

```
Panda::print( ostream &os ) const
{
    Bear::print( os );
    Endangered::print( os );
    return os;
}
```

对于多继承的派生类，即使它的声明被成功编译，也不能保证没有潜在的二义性问题，所以强烈建议在这种类的单元测试中对它的所有方法进行测试（虽然这样做很繁琐）

练习 18.9

已知下列类层次结构，以及数据成员集：

```
class Base1 {
public:
    // ...
protected:
    int ival;
    double dval;
    char cval;
    // ...
private:
    int *id;
    // ...
};
class Base2 {
public:
    // ...
protected:
    float fval;
    // ...
private:
    double dval;
    // ...
};
class Derived : public Base1 {
public:
    // ...
protected:
    string sval;
    double dval;
    // ...
};
class MI : public Derived, public Base2 {
public:
    // ...
protected:
    int *ival;
    complex<double> cval;
    // ...
};
```

以及 MI::foo()成员函数的骨架：

```
int ival;
double dval;
void MI::
foo( double dval )
{
     int id;
     //...
}
```

(a) 请指出在 MI 中可见的成员有哪些，有来自多个基类中的可见成员吗?
(b) 请指出在 MI::foo()中可见的成员有哪些?

练习 18.10

请用在练习 18.9 中定义的类层次结构，指出成员函数 MI::bar()中的赋值哪些是错误的。

```
void MI::
bar()
{
     int sval;
     // 练习答案放在此处
}

(a) dval = 3.14159;    (d) fval = 0;
(b) cval = 'a';    (e) sval = *ival;
(c) id = 1;
```

练习 18.11

请用在练习 18.9 中定义的类层次结构，以及下面成员函数 MI::foobar()的骨架，

```
int id;
void MI::
foobar( float cval )
{
     int dval;
     // 练习答案放在此处
}
```

(a) 将 Base1 的 dval 成员加上 Derived 的 dval 成员，并将结果赋值给局部的 dval 实例。
(b) 将 MI 的 cval 的实数部分赋值给 Base2 的 fval 成员。
(c) 将 Base1 的 cval 成员赋值给 Derived 的 sval 成员的第一个字符。

练习 18.12

已知下列类层次结构，以及下列名为 print()的成员函数：

```
class Base {
public:
     void print( string ) const;
```

```
    // ...
};

class Derived1 : public Base {
public:
    void print( int ) const;
    // ...
};

class Derived2 : public Base {
public:
    void print( double ) const;
    // ...
};

class MI : public Derived1, public Derived2 {
public:
    void print( complex<double> )const;
    // ...
};
```

1. 为什么下列语句会导致编译时刻错误？

```
MI mi;
string dancer( "Nijinsky" );
mi.print( dancer );
```

2. 怎样修改 MI 的定义以使它正确编译和执行？

18.5 虚拟继承 ※

在缺省情况下，C++中的继承是按值组合的一种特殊情况。当我们写：

```
class Bear : public ZooAnimal { ... };
```

每个 Bear 类对象都含有其 ZooAnimal 基类子对象的所有非静态数据成员，以及在 Bear 中声明的非静态数据成员。类似地，当派生类自己也作为一个基类对象时，如：

```
class PolarBear : public Bear { ... };
```

则 PolarBear 类对象含有在 PolarBear 中声明的所有非静态数据成员，以及其 Bear 子对象的所有非静态数据成员和 ZooAnimal 子对象的所有非静态数据成员。

在单继承下，这种由继承支持的、特殊形式的按值组合提供了最有效的、最紧凑的对象表示。在多继承下，当一个基类在派生层次中出现多次时就会有问题。最主要的实际例子是 iostream 类层次结构，我们记得在图 18.2 中，ostream 和 istream 类都从抽象 ios 基类派生而来。而 iostream 类又是从 ostream 和 istream 派生：

```
class iostream :
    public istream, public ostream { ... };
```

缺省情况下，每个 iostream 类对象含有两个 ios 子对象：在 istream 子对象中的实例以及在 ostream 子对象中的实例。这为什么不好？从效率上而言，存储 ios 子对象的两个复本，浪费了存储区，因为 iostream 只需要一个实例。而且，ios 构造函数被调用了两次，每个子对

象一次。更严重的问题是由于两个实例引起的二义性。例如，任何未限定修饰地访问 ios 的成员都将导致编译时刻错误：到底访问哪个实例？如果 ostream 和 istream 对其 ios 子对象的初始化稍稍不同，会怎样呢？怎样通过 iostream 类保证这一对 ios 值的一致性？在缺省的按值组合机制下，真的没有好办法可以保证这一点。

C++语言的解决方案是，提供另一种可替代“按引用组合”的继承机制：虚拟继承（virtual inheritance）。在虚拟继承下，只有一个共享的基类子对象被继承，而无论该基类在派生层次中出现多少次。共享的基类子对象被称为虚拟基类（virtual base class）。在虚拟继承下，基类子对象的复制及由此而引起的二义性都被消除了。

为了讨论虚拟继承的语法和语义，我们选择用 Panda 类作为教学示例。在动物学领域中，人们对 Panda（熊猫）属于浣熊科（Raccoon）还是熊（Bear）科，已经激烈争论了 100 多年。由于软件设计主要是一种服务性工业，所以，我们最实际的解决方案是同时从两者派生：

```
class Panda : public Bear,
    public Raccoon, public Endangered { ... };
```

虚拟继承 Panda 层次结构如图 18.4 所示，其中两个虚箭头分别表示 Bear 和 Raccoon 从 ZooAnimal 的虚拟派生，而三个实箭头分别表示 Panda 从 Bear、Raccoon 和 18.2 节的 Endangered 的非虚拟派生。

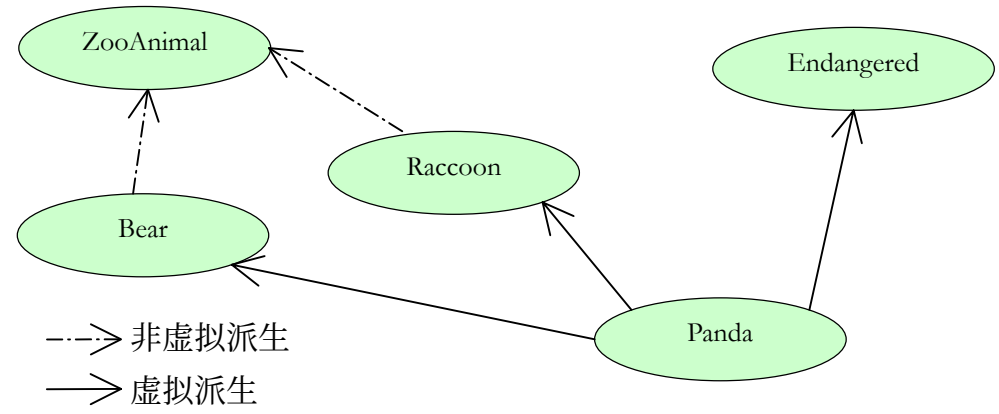


图 18.4 虚拟继承 Panda 层次结构

如果仔细查看图 18.4，我们会注意到虚拟继承的不直观部分：虚拟派生（本例中的 Bear 和 Raccoon）在先，实际上应该在后。只有伴随着 Panda 的声明，虚拟继承才是必要的。但是，如果 Bear 和 Raccoon 还没有实现虚拟派生，则 Panda 类的设计者就不走运了。

这是否意味着，我们应该尽可能地以虚拟方式派生我们的基类，以便层次结构中后续的派生类可能会需要虚拟继承，是这样吗？不。我们强烈反对。那样做对性能的影响会很严重（而且增加了后续类派生的复杂性），关于性能评测和讨论见［LIPPMAN96a］。

那么，我们从不应该使用虚拟继承吗？不是。在实践中，几乎所有成功使用虚拟继承的例子中，凡是需要虚拟继承的整个层次结构子树，如 iostream 库或 Panda 子树，都是由同一个人或项目设计组一次设计完成的。

一般地，除非虚拟继承为一个眼前的设计问题提供了解决方案，否则建议不要使用它。当然，尽管如此，现在我们仍然要看看怎样使用它。

18.5.1 虚拟基类声明

通过用关键字 virtual 修改一个基类的声明可以将它指定为被虚拟派生。例如，下列声明使得 ZooAnimal 成为 Bear 和 Raccoon 的虚拟基类：

```
// 关键字 public 和 virtual
// 的顺序不重要
class Bear : public virtual ZooAnimal { ... };
class Raccoon : virtual public ZooAnimal { ... };
```

虚拟派生不是基类本身的一个显式特性，而是它与派生类的关系。如前面所说明的，虚拟继承提供了“按引用组合”。也就是说，对于子对象及其非静态成员的访问是间接进行的。这使得在多继承情况下，把多个虚拟基类子对象组合成派生类中的一个共享实例，从而提供了必要的灵活性。同时；即使一个基类是虚拟的，我们仍然可以通过该基类类型的指针或引用，来操纵派生类的对象。例如，尽管 Panda 被设计为虚拟继承层次结构，下面的 Panda 基类转换也可以正确执行：

```
extern void dance( const Bear* );
extern void rummage( const Raccoon* );
extern ostream&
operator<<( ostream&, const ZooAnimal& );
int main()
{
    Panda yin_yang;
    dance( &yin_yang ); // ok
    rummage( &yin_yang );   // ok
    cout << yin_yang; // ok
    // ...
}
```

如果一个类可以被指定为基类，那么我们就可以将它指定为虚拟基类，而且它可以包含非虚拟基类支持的所有元素。例如，下面是 ZooAnimal 类声明：

```
#include <iostream>
#include <string>

class ZooAnimal;
extern ostream&
    operator<<( ostream&, const ZooAnimal& );

class ZooAnimal {
public:
    ZooAnimal( string name,
        bool onExhibit, string fam_name )
        : _name( name ),
        _onExhibit( onExhibit), _fam_name( fam_name )
    {}

    virtual ~ZooAnimal();
    virtual ostream& print( ostream& ) const;

    string name() const { return _name; };
```

```
        string family_name() const { return _fam_name; }
        // ...
    protected:
        bool _onExhibit;
        string _name;
        string _fam_name;
        // ...
    };
```

直接派生类实例的声明和实现与非虚拟派生的情形相同，只是要用到关键字 virtual。例如，下面是 Bear 类声明：

```
    class Bear : public virtual ZooAnimal {
    public:
        enum DanceType {
            two_left_feet, macarena, fandango, waltz };
        Bear( string name, bool onExhibit=true )
            : ZooAnimal( name, onExhibit, "Bear" ),
                      _dance( two_left_feet )
        {}

        virtual ostream&print( ostream& ) const;
        void dance( DanceType );

        // ...
    protected:
        DanceType _dance;
        // ...
    };
```

类似地，下面是 Raccoon 类的声明：

```
    class Raccoon : public virtual ZooAnimal {
    public:
        Raccoon( string name, bool onExhibit=true )
            : ZooAnimal( name, onExhibit, "Raccoon" ),
            _pettable( false )
        {}

        virtual ostream&print( ostream&) const;
        bool pettable() const { return _pettable; }
        void pettable( bool petval ) { _pettable = petval; }

        // ...
    protected:
        bool _pettable;
        // ...
    };
```

18.5.2 特殊的初始化语义

如果在一个派生类中有一个或多个虚拟基类间接出现，那么它就需要有特殊的初始化语

义。稍后我们将看一看上节中的 Bear 和 Raccoon 类的实现。你能看出由 Panda 类派生引起的问题吗？

```
class Panda : public Bear,
     public Raccoon, public Endangered {
public:
     Panda( string name, bool onExhibit=true );
     virtual ostream& print( ostream& ) const;

     bool sleeping() const { return _sleeping; }
     void sleeping( bool newval ) { _sleeping = newval; }

     // ...
protected:
     bool _sleeping;
     // ...
};
```

对。问题在于 Bear 和 Raccoon 的基类构造函数都提供了一个带有显式实参集合的 ZooAnimal 构造函数。更加糟糕的是，在我们的例子中，这个被用作科目名（name）的实参不但不相同，而且对 Panda 类无效。

在非虚拟派生中，派生类只能显式初始化其直接基类（见 17.4 节的讨论）。例如，在 ZooAnimal 的非虚拟派生中，Panda 类不能在 Panda 成员初始化表中直接调用 ZooAnimal 的构造函数。然而，在虚拟派生中，只有 Panda 可以直接调用其 ZooAnimal 虚拟基类的构造函数。

虚拟基类的初始化变成了**最终派生类**（most derived class）的责任，这个最终派生类是由每个特定类对象的声明来决定的。例如，我们在声明 Bear 类对象时：

```
Bear winnie( "pooh" );
```

Bear 是 winnie 对象的最终派生类，它所调用的 ZooAnimal 构造函数被执行。当我们写如下语句时：

```
cout << winnie.family_name();
```

输出的是：

```
The family name for pooh is Bear.
```

类似地，如下声明：

```
Raccoon meeko( "meeko" );
```

声明 Raccoon 是 meeko 类对象的最终派生类时，因此应执订 Raccoon 调用的 ZooAnimal 构造函数。当我们写如下语句时：

```
cout << meeko.family_name();
```

输出的是：

```
The family name for meeko is Raccoon.
```

现在，当我们声明 Panda 类对象时，比如：

```
Panda yolo( "yolo" );
```

Panda 是 yolo 类对象的最终派生类，所以初始化 ZooAnimal 成为 Panda 类的责任。

当一个 Panda 对象被初始化时，①在 Raccoon 和 Bear 的构造函数执行过程中，它们对于 ZooAnimal 构造函数的调用不再被执行；②ZooAnimal 构造函数被调用时，其实参是在 Panda 的初始化表中被指定的。下面是具体实现：

```
Panda::Panda( string name, bool onExhibit=true )
            : ZooAnimal( name, onExhibit, "Panda" ),
            Bear( name, onExhibit ),
            Raccoon( name, onExhibit ),
            Endangered( Endangered::environment,
                        Endangered::critical )
            _sleeping( false )
{}
```

如果 Panda 的构造函数没有显式地为 ZooAnimal 构造函数指定实参，则发生下面两个动作之一：调用 ZooAnimal 的缺省构造函数，或者，如果没有缺省构造函数，则编译器在编译 Panda 构造函数的定义时会给出一个错误消息。

当我们写如下语句时：

```
cout << yolo.family_name();
```

输出的是：

```
The family name for yolo is Panda.
```

在 Panda 中，Bear 和 Raccoon 类都被用作中间派生类而不是最终派生类。作为中间派生类，所有对虚拟基类构造函数的调用都被自动抑制了。如果 Panda 又被其他类派生，则 Panda 也将成为中间派生类，它对 ZooAnimal 构造函数的调用也将被自动抑制住。

或许你已经注意到，当 Bear 和 Raccoon 类被用作中间派生类时，向 Bear 和 Raccoon 构造函数传递的两个实参是不必要的。避免这种不必要的参数传递的解决方案是，提供一个显式的构造函数，用于“当它被作为中间派生类时”的情形。例如，中间类 Bear 的构造函数可以修改如下

```
class Bear : public virtual ZooAnimal {
public:
    // 当作为最终派生类时
    Bear( string name, bool onExhibit=true )
        : ZooAnimal( name, onExhibit, "Bear" ),
        _dance( two_left_feet )
    {}

    // ... rest the same

protected:
    // 当作为一个中间派生类时
    Bear() : _dance( two_left_feet ) {}

    // ... rest the same
};
```

我们将这个实例指定为 protected，因为它只希望在后续的派生类中被调用。假设我们已经为 Raccoon 提供了类似的缺省构造函数，则可以如下修改 Panda 构造函数：

```
Panda::Panda( string name, bool onExhibit = true )
            : ZooAnimal( name, onExhibit, "Panda" ),
              Endangered( Endangered::environment,
                          Endangered::critical )
              _sleeping( false )
{}
```

18.5.3 构造函数与析构函数顺序

无论虚拟基类出现在继承层次中的哪个位置上，它们都是在非虚拟基类之前被构造。例如，在下面这个有点古怪的 TeddyBear 派生类中，有两个虚拟基类：直接的 ToyAnimal 实例，以及来自 Bear 的 ZooAnimal 实例：

```
class Character { ... };
class BookCharacter : public Character { ... };
class ToyAnimal { ... };
class TeddyBear : public BookCharacter,
                  public Bear, public virtual ToyAnimal
{ ... };
```

层次结构如图 18.5 所示，这里的虚拟派生用虚箭头表示，而非虚拟派生用实箭头表示。

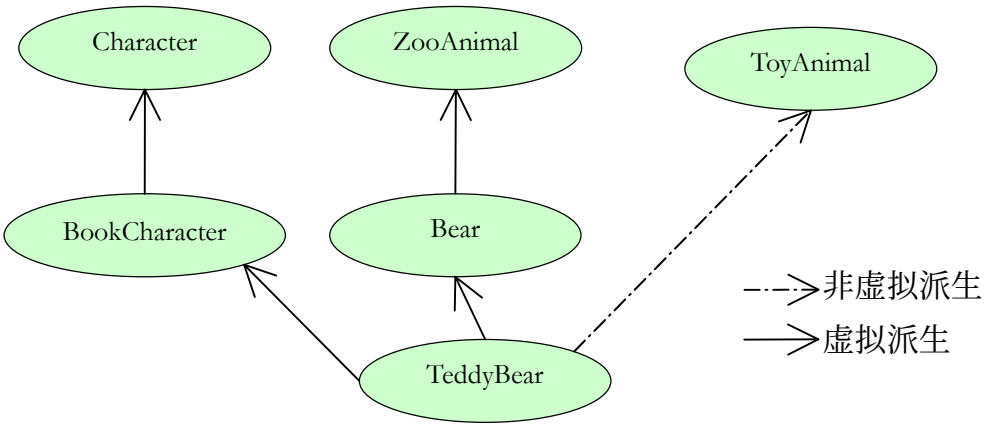


图 18.5 虚拟继承 TeddyBear 层次结构

编译器按照直接基类在声明中的顺序，来检查虚拟基类的出现情况。在我们的例子中，BookCharacter 的继承子树首先被检查，然后是 Bear，最后是 ToyAnimal。每个子树按深度优先的顺序被检查。即，查找从树根类开始，然后向下移动。对于 BookCharacter 子树，先检查 Character，然后是 BookCharacter。对于 Bear 子树而言，则先检查 ZooAnimal，然后是 Bear。

在这个查找算法下，TeddyBear 的虚拟基类构造函数的调用顺序是，先 ZooAnimal，后跟 ToyAnimal。

一旦调用了虚拟基类的构造函数，则非虚拟基类构造函数就按照声明的顺序被调用：先是 BookCharater，然后是 Bear。在 BookCharacter 构造函数执行之前，它的基类 Character 构造函数先被调用。

已知声明：

```
TeddyBear Paddington;
```

基类构造函数的调用顺序如下：

```
ZooAnimal();            // Bear 的虚拟基类
ToyAnimal();            // 直接虚拟基类
Character();            // BookCharacter 的非虚拟基类
BookCharacter();        // 直接非虚拟基类
Bear();                 // 直接非虚拟基类
TeddyBear();            // 最终派生类
```

这里初始化 ZooAnimal 和 ToyAnimal 是 TeddyBear 的责任，因为它是 Paddington 类对象的最终派生类。

“按成员初始化”下的拷贝构造函数（以及“按成员赋值”下的拷贝赋值操作符）的调用顺序，也是如此。基类析构函数的调用顺序则保证与构造函数的调用顺序相反。

18.5.4 虚拟基类成员的可视性

让我们重新定义 Bear 类，以提供它自己的 onExhibit()成员函数的实例（原来的 onExhibit()成员实例从 ZooAnimal 继承而来）：

```
bool Bear::onExhibit() { ... }
```

通过 Bear 类对象引用的 onExhibit()现在被解析为 Bear 的实例：

```
Bear winnie( "a lover of honey" );
winnie.onExhibit();    // Bear::onExhibit()
```

通过 Raccoon 类对象引用的 Raccoon meeko("a lover of all foods");

```
Raccoon meeko( "a lover of all foods" );
meeko.onExhibit(); // ZooAnimal::onExhibit()
```

派生类 Panda 从它的两个基类所继承而来的成员可被分为以下三类：

1．ZooAnimal 虚拟基类实例，如 name()和 family_name()，它们没有被 Bear 和 Raccoon 改写。

2．继承自 Raccoon、属于 ZooAnimal 虚拟基类的 onExhibit()实例，以及 Bear 定义的、被改写了的 onExhibit()实例。

3．继承自 ZooAnimal、分别被 Bear 和 Raccoon 特化了的 print()实例。

对于这些继承得到的成员，哪些可以在 Panda 类域中被直接地、无二义地访问？在非虚拟派生下，答案是没有，所有非限定修饰的引用都是二义的。在虚拟派生下，第 1 项和第 2 项的所有成员都可以被直接地、无二义地访问。例如，已知 Panda 类对象：

```
Panda spot( "Spottie" );
```

下面的调用：

```
spot.name();
```

调用了共享的 ZooAnimal 虚拟基类成员函数 name()。而下面的调用：

```
spot.onExhibit();
```

调用了派生的 Bear 成员函数 onExhibit()。

当两个以上的成员实例分别通过不同的派生路径被继承（不但适用于成员函数，也适用

于数据成员和联套类型），并且它们都代表了相同的虚拟基类成员时，则不存在二义性，因为它们共享了该成员的单个实例（第 1 项）。如果一个代表虚拟基类的成员，而另一个是后续派生类的成员，则也不会有二义性（特化的派生类实例的优先级高于共享的虚拟基类实例［第 2 项］）。但是，如果它们都代表后续派生类的实例，则直接访问该成员就是二义的。最好的解决办法是在派生类中给出一个改写的实例（第 3 项）

例如，在非虚拟派生下，通过 Panda 类对象对 onExhibit()的非限定修饰引用就是二义的：

```
// 错误：在非虚拟派生下，二义
Panda yolo( "a lover of bamboo" );
yolo.onExhibit();
```

在非虚拟派生下的解析引用过程中，每个继承得到的实例都具有同样的权值，所以未限定修饰的引用将导致编译时刻二义性错误（见 18.4.1 节讨论）

在虚拟派生下，对于虚拟基类成员的继承比“该成员后来重新定义的实例”的权值小。继承得到的 Bear 的 onExhibit）实例，比通过 Raccoon 继承得到的 ZooAnimal 实例优先。

```
// ok：在虚拟继承下没有二义
// 调用 Bear::onExhibit()
yolo.onExhibit();
```

如果在同一派生级别上有两个或多个基类重新定义了一个虚拟基类成员，则在派生类中，它们有相同的优先级。例如，如果 Raccoon 也定义了一个 onExhibit()成员，则 Panda 需要用适当的类域操作符来限定修饰每个访问：

```
bool Panda::onExhibit()
{
    return Bear::onExhibit() &&
           Raccoon::onExhibit() &&
           ! _sleeping;
}
```

练习 18.13

已知下面的类层次结构：

```
class Class { ... };
class Base : public Class { ... };
class Derived1 : virtual public Base { ... };
class Derived2 : virtual public Base { ... };
class MI : public Derived1,
    public Derived2 { ... };
class Final : public MI, public Class { ... };
```

(a) Final 类对象的构造函数和析构函数的调用顺序如何？
(b) Final 类对象含有多少个 Base 子对象，多少个 Class 子对象？
(c) 下列哪个赋值会引起编译时刻错误？

```
Base *pb;
MI *pmi;
Class *pc;
Derived2 *pd2;
```

```
(i) pb = new Class;    (iii) pmi = pb;
(ii) pc = new Final;   (iv) pd2 = pmi;
```

练习 18.14

已知下列类层次结构，以及下列成员：

```
class Base {
public:
     bar( int );
     // ...
protected:
     int ival;
     // ...
};

class Derived1 : virtual public Base {
public:
     bar( char );
     foo( char );
     // ...
protected:
     char cval;
     // ...
};
class Derived2 : virtual public Base {
public:
     foo( int );
     // ...
protected:
     int ival;
     char cval;
     // ...
};
class VMI : public Derived1, public Derived2 {};
```

哪些继承得到的成员可在 VMI 类中不用限定修饰就可以被访问，哪些又需要限定修饰？

练习 18.15

已知下面的 Base 类及其三个构造函数：

```
class Base {
public:
     Base();
     Base( string );
     Base( const Base&);
     // ...
protected:
     string _name;
};
```

请为下面的类定义相应的三个构造函数：

```
(a)  两者之一
     class Derived1 : virtual public Base{ ... };
     class Derived2 : virtual public Base{ ... };
(b)  class VMI : public Derived1, public Derived2{ ... };
(c)  class Final : public VMI{ ... };
```

18.6 多继承及虚拟继承实例 ※

本节我们将通过实现 2.4 节引入的 Array 类模板层次结构，来说明多继承、虚拟继承的定义和用法。我们的实现将以第 16 章给出的 Array 类模板为基础，将它修改为一个实体基类。作为开始，我们将首先简要地讨论如何使用带有继承的类模板。

类模板的实例可以被用作一个显式的基类，如下：

```
class IntStack : private Array<int> {};
```

除此之外，类模板也可以从非模板基类派生，如下所示：

```
class Base {};

template < class Type >
     class Derived : public Base {};
```

或同时被用作派生类和基类：

```
template < class Type >
     class Array_RC : public virtual Array<Type> {};
```

在第一个例子中，Array 类模板的整型实例被用作非模板类 IntStack 的私有基类。在第二个例子中，非模板类 Base 被用作每个 Derived 类模板实例的基类。在第三个例子中，模板类 Array_RC 的每个实例都把相应的 Array 类模板实例作为基类。例如：

```
Array_RC<int> ia;
```

生成了 Array 和 Array_RC 类模板的整型实例。

另外，模板参数自己也可以被用作基类、例如，[MURRAY93] 说明了这种情况：

```
template < typename Type >
     class Persistent : public Type{ ... };
```

它为每个被实例化的类型定义了一个派生的 Persistent 子类型。正如 Murray 注记所说，Type 的隐含限制是，它必须是一个类类型。例如：

```
Persistent< int > pi;  // 喔！错误
```

将因为内置类型不能再被派生，而导致编译时刻错误。

当一个类模板被用作基类时，我们必须用它完整的参数表对其进行修饰。例如，已知下列类模板定义：

```
template < class T > class Base {};
```

我们应该写：

```
template < class Type >
     class Derived : public Base<Type> {};
```

而不是：

```
// 错误：Base 是一个模板
// 必须指定模板实参
template < class Type >
     class Derived : public Base{};
```

在下一节中，第 16 章定义的 Array 类模板将被用作这样几个类的虚拟基类：①带有范围检查的 Array 子类型；②排序的 Array 子类型；③既排序又进行范围检查的 Array 子类型。但是原始的 Array 类模板定义不适合被用于派生：

- 所有的数据成员和辅助函数都是 private 的，而不是 protected。
- 与类型相关的函数（如下标操作符）都没有被指定为 virtual。

这是否意味着我们原来的实现是错误的？不。在我们理解的范围内，它是正确的。在实现原来的类模板 Array 时，我们没有意识到将来需要对 Array 类型进行特化。但是现在我们知道了，所以我们需要修订 Array 类模板的定义（成员函数的实现仍然相同）。下面是新的 Array 类模板定义：

```
#ifndef ARRAY_H
#define ARRAY_H
#include <iostream>

// 为了 operator<< 而需要的前向声明
template <class Type> class Array;
template <class Type> ostream&
     operator<<( ostream&, const Array<Type>&);
template <class Type>

class Array {
     static const int ArraySize = 12;
public:
     explicit Array( int sz = ArraySize ) { init( 0, sz ); }
     Array( const Type *ar, int sz )      { init( ar, sz); }
     Array( const Array &iA )    { init( iA.ia, iA.size()); }
     virtual ~Array()            { delete [] ia; }

     Array&operator=( const Array&);
     int size() { return _size; }
     virtual void grow();

     virtual void print( ostream&= cout );

     Type at( int ix ) const { return ia[ ix ]; }
     virtual Type&operator[]( int ix ) { return ia[ix]; }

     virtual void sort( int,int );
     virtual int find( Type );
     virtual Type min();
     virtual Type max();
protected:
     void swap( int,int );
```

```
    void init( const Type*, int );
    int _size;
    Type *ia;
};
#endif
```

转变为多态设计带来的问题是，下标操作符的一般用法已经从一个 inline 内存访问转变成相当复杂的虚拟函数调用。例如，在下面的函数中，无论 ia 指向什么类型，一个简单的内联读取元素的操作就已经足够了：

```
int find( const Array< int > &ia, int value )
{
    for ( int ix = 0; ix < ia.size(); ++ix )
        // 现在变成了一个虚拟函数调用
        if ( ia[ ix ] == value )
            return ix;
    return -1;
}
```

出于性能考虑，我们给出了 inline 成员函数 at()，以用来直接读取元素。

18.6.1 带有范围检查的 Array 派生类

在 16.13 节的函数时 try_array()（该函数被用来练习早先实现的 Array 类模板）中，有下列两条语句：

```
int index = iA.find( find_val );
Type value = iA[ index ];
```

find()返回 find_val 第一次出现时的索引，如果该值在数组中没有出现，则返回-1。此代码是不正确的，因为它没有测试返回值可能为-1。由于-1 落在数组边界之外，所以 value 的初始化可能是无效的，程序的执行过程有潜在的错误。让我们来定义一个带有范围检查的 Array 子类型，把它称作 Array_RC，并把它定义在名为 Array_RC.h 的头文件中：

```
#ifndef ARRAY_RC_H
#define ARRAY_RC_H

#include "Array.h"

template <class Type>
class Array_RC : public virtual Array<Type> {
public:
    Array_RC( int sz = ArraySize )
              : Array<Type>( sz ) {}

    Array_RC( const Array_RC&r );
    Array_RC( const Type *ar, int sz );
    Type&operator[]( int ix );
};

#endif
```

在派生类的定义中，每次用到模板基类类型的指示符时，我们都必须使用它的完整形式参数表来限定修饰。我们应该写：

```
Array_RC( int sz = ArraySize )
    : Array<Type>( sz ) {}
```

而不是：

```
// 错误：Array 不是一个类型指示符
Array_RC( int sz = ArraySize ) : Array( sz ) {}
```

Array_RC 类惟一特殊的行为是，它的下标操作符会执行范围检查。否则 Array 类模板的实现可以被直接重用。但是，因为构造函数没有被继承，所以 Array_RC 类定义了三个构造函数。从 Array 到 Array_RC 的虚拟派生为后续的多重派生做好了准备，稍后我们将会看到。

下面是 Array_RC 成员函数完整的实现，放在名为 Arrny_RC.C 的文件中（因为我们使用了包含模式的模板实例［见 16.8.1 节讨论］，所以将 Array 的函数定义放在 Array.C 头文件中）：

```
#include "Array_RC.h"
#include "Array.C"
#include <assert.h>

template <class Type>
Array_RC<Type>::Array_RC( const Array_RC<Type> &r )
    : Array<Type>( r ) {}

template <class Type>
Array_RC<Type>::Array_RC( const Type *ar, int sz )
    : Array<Type>( ar, sz ) {}

template <class Type>
Type &Array_RC<Type>::operator[]( int ix ) {
    assert( ix >= 0 &&ix < Array<Type>::_size );
    return ia[ ix ];
}
```

为什么要对引用到的 Array 基类成员进行限定修饰呢？比如下面的_size 的限定修饰：

```
Array<Type>::_size;
```

我们必须这样做，才能保证直到模板被实例化之后 Array 基类才会被检查。我们通过“对基类的引用依赖于模板参数”来做到这一点。在 Array_RC 定义中的名字，除了显式依赖于模板参数的名字之外，其余的都是在定义模板的时候被解析的。当使用非限定修饰的名字_size 时，编译器必须找到_size 的定义，除非该名字显式地依赖于模板参数。为了让名字_size 依赖于模板参数，我们将它加上基类名 Array<type>的前缀。于是，直到模板被实例化时，编译器才解析名字_size（在 Array_Sort 的类定义中，我们将会看到更多这样的例子）

Array_RC 的每一个实例化都生成一个相对应的 Array 类实例，例如：

```
Array_RC<string> sa;
```

生成一个 string Array_RC 以及一个相应的 string Array 实例。下面的程序重新运行 try_array()（实现见 16.13 节），并将一个 Array_RC 子类型的对象传递给它。如果我们的实现是正确的，则边界违例将被捕获到：

```
#include "Array_RC.C"
#include "try_array.C"

main()
{
     static int ia[10] = { 12,7,14,9,128,17,6,3,27,5 };
     Array_RC<int> iA( ia,10 );

     cout << "class template instantiation Array_RC<int>\n";
     try_array( iA );

     return 0;
}
```

编译并运行该程序，产生下列输出：

```
class template instantiation Array_RC<int>
try_array: initial array values:
( 10 )< 12, 7, 14, 9, 128, 17
        6, 3, 27, 5 >
try_array: after assignments:
( 10 )< 128, 7, 14, 9, 128, 128
        6, 3, 27, 3 >
try_array: memberwise initialization
( 10 )< 128, 7, 14, 9, 128, 128
        6, 3, 27, 3 >
try_array: after memberwise copy
( 10 )< 128, 7, 128, 9, 128, 128
        6, 3, 27, 3 >
try_array: after grow
( 16 )< 128, 7, 128, 9, 128, 128
        6, 3, 27, 3, 0, 0
        0, 0, 0, 0 >

value to find: 5 index returned: -1
Assertion failed: ix >= 0 && ix < _size
```

18.6.2 排序的 Array 派生类

下面是 Array 的第二个特殊化子类型——排序的 Array 子类型。我们把它称作 Array_Sort，并且将其定义在头文件 Array_S.h 中。如下所示：

```
#ifndef ARRAY_S_H
#define ARRAY_S_H

#include "Array.h"

template <class Type>
class Array_Sort : public virtual Array<Type> {
protected:
     void set_bit() { dirty_bit = true; }
```

```
        void clear_bit() { dirty_bit = false; }
        void check_bit() {
            if ( dirty_bit ) {
                sort( 0, Array<Type>::_size-1 );
                clear_bit();
            }
        }
    public:
        Array_Sort( const Array_Sort&);
        Array_Sort( int sz = Array<Type>::ArraySize )
            : Array<Type>( sz )
            { clear_bit(); }

        Array_Sort( const Type* arr, int sz )
            : Array<Type>( arr, sz )
            { sort( 0,Array<Type>::_size-1 ); clear_bit(); }

        Type&operator[]( int ix )
        { set_bit(); return ia[ ix ]; }
        void print( ostream&os = cout )
        { check_bit(); Array<Type>::print( os ); }

        Type min() { check_bit(); return ia[ 0 ]; }
        Type max() { check_bit(); return ia[ Array<Type>::_size-1 ]; }
        bool is_dirty() const { return dirty_bit; }

        int find( Type );
        void grow();
    protected:
        bool dirty_bit;
    };

    #endif
```

Array_Sort 引入了一个额外的数据成员 dirty_bit。如果 dirty_bit 被设置了，则不再保证数组是排过序的。它还提供了许多支持访问的函数：is_dirty()返回 dirty_bit 的值，set_bit()将 dirty_bit 置为 true，clear_bit()将 dirty_bit 置为 false，如果 dirty_bit 被设置为 true，则 check_bit()重新排序数组，然后清除 dirty_bit。任何潜在的能使数组打破顺序的操作都将调用 set_bit()。

对 Array 基类模板的所有引用都必须指定该类的完整参数表。例如：

```
Array<Type>::print( os );
```

调用的是基类的 print()函数；这是与每个 Array_Sort 实例相对应的 Array 类实例。例如：

Array_Sort<string> sas;

实例化了一个 string Array_Sort 实例以及一个 string Array 类实例：

```
cout << sas;
```

实例化了一个输出操作符的 string Array 实例，sas 被传递给该输出操作符。在操作符中，调用：

```
ar.print( os );
```

调用了 string Array_Sort 虚拟实例的 print()。首先，check_bit()被调用。然后，string Array 实例的 print()被静态调用。（静态调用意味着该函数在编译时刻被解析，并且在适当情况下被内联扩展开。）虚拟函数通常会在运行时刻根据 ar 指向的实际对象而被动态调。当用类域操作符显式地调用虚拟函数时，我们就改变了虚拟机制，就好像 Array::print()那样。当在派生类的虚拟函数实例中显式地调用基类的虚拟函数实例时（比如在 Array_Sort 的 print()实例中），这是一个很有效的辅助手段（讨论见 17.5 节）

在类定义之外定义的成员函数被放在文件 Array_S.C 中。由于模板的语法，该声明看起来非常复杂。但是，除了参数表之外，该声明与非模板类相同：

```
template <class Type>
Array_Sort<Type>::
Array_Sort( const Array_Sort<Type> &as )
    : Array<Type>( as )
{
    // 注意: as.check_bit() 不能工作!
    // ——见后面的说明 ...
    if ( as.is_dirty() )
        sort( 0, Array<Type>::_size-1 );
    clear_bit();
}
```

如果一个模板名字被用作类型指示符，则必须用完整的参数表对它进行限定修饰。因而我们应该写成：

```
template <class Type>
Array_Sort<Type>::
Array_Sort( const Array_Sort<Type> &as )
```

而不是：

```
template <class Type>
Array_Sort<Type>::
Array_Sort<Type>( // 错误: 不是类型指示符
```

同为第二个出现的 Array_Sort 是用作函数名而不是类型指示符。

我们之所以写：

```
if ( as.is_dirty() )
    sort( 0, _size );
```

而不是：

```
as.check_bit();
```

的原因有两个。第一个原因是类型安全：check_it()是非 const 成员函数——它要修改其相关的类对象。实参 as 是作为一个常量对象的引用被传递进来的，所以 as 上的 check_bit()调用违反了常量性，在编译时刻被标记为错误。

第二个原因是拷贝构造函数并不关心与 as 相关联的数组，而只判断新创建的 Array_Sort 对象是否需要排序。记住，与新的 Array_Sort 对象相关联的 dirty_bit 数据成员还没有被初始化。当 Array_Sort 构造函数体开始时，只有从 Array 类继承的成员 ia 和_size 被初始化了。Array_Sort 的构造函数必须初始化它自己额外的数据成员（通过调用 clear_bit()），以及实现

该子类型的特殊行为（通过调用 sort()），Array_Sort 构造函数的另一种实现如下：

```
// 另一种实现
template <class Type>
Array_Sort<Type>::
Array_Sort( const Array_Sort<Type> &as )
     : Array<Type>( as )
{
     dirty_bit = as.dirty_bit;
     check_bit();
}
```

下面是成员函数 grow()的实现。[31]策略是，重用从 Array 类继承来的 grow()实例以分配额外的内存，然后再重新排序数组元素，并清除 dirty_bit：

```
template <class Type>
void Array_Sort<Type>::grow()
{
     Array<Type>::grow();
     sort( 0, Array<Type>::_size-1 );
     clear_bit();
}
```

下面是 Array_Sort 的 find()实例的二分查找的具体实现：

```
template <class Type>
int Array_Sort<Type>::find( Type val )
{
     int low = 0;
     int high = Array<Type>::_size-1;
     check_bit();

     while ( low <= high ) {
          int mid = ( low + high )/2;
          if ( val == ia[ mid ])
               return mid;
          if ( val < ia[ mid ] )
               high = mid-1;
          else low = mid+1;
     }
     return -1;
}
```

让我们用 try_array()函数测试 Array_Sort 的实现。下面的程序测试了 Array_Sort 类的整型和 string 实例：

```
#include "Array_S.C"
#include "try_array.C"
#include <string>

main()
{
```

[31] 如果在 grow()把“原来数组中的一个元素”拷贝到一个新的位置之前，客户已经保存了（通过引用返回的）该元素的地址，那么便可能出现空悬引用（dangling reference）。详细的讨论见［LIPPMAN96b］中 Tom Cargill 的文章。

```
    static int ia[ 10 ] = { 12,7,14,9,128,17,6,3,27,5 };
    static string sa[ 7 ] = {
        "Eeyore", "Pooh", "Tigger",
        "Piglet", "Owl", "Gopher", "Heffalump"
    };

    Array_Sort<int> iA( ia,10 );
    Array_Sort<string> SA( sa,7 );

    cout << "class template instantiation Array_Sort<int>"
        << endl;
    try_array( iA );
    cout << "class template instantiation Array_Sort<string>"
        << endl;
    try_array( SA );

    return 0;
}
```

当编译并执行程序时，string 实例的输出看起来如下所示——注意，当它试图显示一个用“越界的值-1”作为索引的元素时，执行就会失败。

```
class template instantiation Array_Sort<string>

try_array: initial array values:
( 7 )< Eeyore, Gopher, Heffalump, Owl, Piglet, Pooh
    Tigger >

try_array: after assignments:
( 7 )< Eeyore, Gopher, Owl, Piglet, Pooh, Pooh
    Pooh >

try_array: memberwise initialization
( 7 )< Eeyore, Gopher, Owl, Piglet, Pooh, Pooh
    Pooh >

try_array: after memberwise copy
( 7 )< Eeyore, Piglet, Owl, Piglet, Pooh, Pooh
    Pooh >

try_array: after grow
( 11 )<<;empty>, <empty>, <empty>, <empty>, Eeyore, Owl
    Piglet, Piglet, Pooh, Pooh, Pooh >
value to find: Tigger index returned: -1
Memory fault(coredump)
```

注意，按成员拷贝的 Array 类 string 实例的显示并没有被排序，为什么会这样？这是因为此处的虚拟函数是通过类的对象被调用的，而不是通过指针或引用。正如 17.5 节说明的，当通过类对象调用时，被调用的实例反映了该对象的类类型的活动虚拟函数，而不是可能已经被赋值给它的对象的类类型，因此无法通过 Array 类对象调用 Sort 实例。（我们只是把它当作一个示例，在实际的产品代码中并不会这样做。）

18.6.3 多重派生的 Array 类

最后，让我们来定义一个排序的、带有类型检查的数组。我们可以通过继承 Array_RC 和 Array_Sort 来定义它。下面是实现（我们的实现仅限于三个构造函数和一个下标操作符，代码被放在名为 Array_RC_S.h 的头文件中）：

```
#ifndef ARRAY_RC_S_H
#define ARRAY_RC_S_H

#include "Array_S.C"
#include "Array_RC.C"

template <class Type>
     class Array_RC_S : public Array_RC<Type>,
                        public Array_Sort<Type>
{
public:
     Array_RC_S( int sz = Array<Type>::ArraySize )
          : Array<Type>( sz )
          { clear_bit(); }

     Array_RC_S( const Array_RC_S &rca )
          : Array<Type>( rca )
          { sort( 0,Array<Type>::_size-1 ); clear_bit(); }

     Array_RC_S( const Type* arr, int sz )
          : Array<Type>( arr, sz )
          { sort( 0,Array<Type>::_size-1 ); clear_bit(); }

     Type&operator[]( int index ) {
          set_bit();
          return Array_RC<Type>::operator[]( index );
     }
};

#endif
```

该类继承了 Array 类每个接口函数的两份实现：一份是 Array_Sort 的，另一份是通过 Array_RC 继承的虚拟 Array 基类的（除了下标操作符之外，它的继承分别来自两个基类）。在非虚拟派生中，例如，调用 find()被标记为二义的——到底应该调用哪一个继承而来的实例呢？但是，在虚拟派生中，“Array_Sort 中被改写的这些成员函数实例”比“通过 Array_RC 继承而来的虚拟基类实例”优先（在 18.5.4 节详细讨论过这一点）。在虚拟继承下，find()的非限定修饰调用被解析为 Array_Sort 类的实例。

由于在 Array_RC 和 Array.Sort 两个基类中，下标操作符都被重新定义了，所以优先级相同。在 Array_RC_Sort 中，未限定修饰的下标操作符的调用是二义的。该类必须提供自己的实例，或者它的用户不会对该类的对象直接应用下标操作符。在语义上，针对 Array_RC_Sort

类调用下标操作符意味着什么？为了反映它的排序属性，它必须设置继承而来的 dirty_bit 数据成员。为了反映它的范围检查属性，它必须对传递进来的索引值进行测试。然后，才返回被索引的元素。后两步由继承得到的 Array_RC_Sort 下标操作符提供。下面的语句：

```
return Array_RC::operator[]( index );
```

显式地调用这个操作符。因为它是一个显式调用，所以改变了虚拟机制。又因为它是 inline 函数。所以静态解析将导致其代码的内联展开。

让我们通过执行 try_array()函数来测试这个实现，并依次提供 Array_RC_Sort 模板类的整型和 string 类型的实例。下面是程序：

```
#include "Array_RC_S.h"
#include "try_array.C"
#include <string>

int main()
{
     static int ia[ 10 ] = { 12,7,14,9,128,17,6,3,27,5 };
     static string sa[ 7 ] = {
          "Eeyore", "Pooh", "Tigger",
          "Piglet", "Owl", "Gopher", "Heffalump"
     };

     Array_RC_S<int> iA( ia,10 );
     Array_RC_S<string> SA( sa,7 );

     cout << "class template instantiation Array_RC_S<int>"
          << endl;
     try_array( iA );

     cout << "class template instantiation Array_RC_S<string>"
          << endl;
     try_array( SA );

     return 0;
}
```

下面是模板 Array_RC_Sort 类的 string 实例的输出。现在，索引越界错误被捕获到：

```
class template instantiation Array_RC_S<string>

try_array: initial array values:
( 7 )< Eeyore, Gopher, Heffalump, Owl, Piglet, Pooh
     Tigger >

try_array: after assignments:
( 7 )< Eeyore, Gopher, Owl, Piglet, Pooh, Pooh
     Pooh >
try_array: memberwise initialization
```

```
( 7 )< Eeyore, Gopher, Owl, Piglet, Pooh, Pooh
     Pooh >

try_array: after memberwise copy
( 7 )< Eeyore, Piglet, Owl, Piglet, Pooh, Pooh
     Pooh >

try_array: after grow
( 11 )< <empty>, <empty>, <empty>, <empty>, Eeyore, Owl
     Piglet, Piglet, Pooh, Pooh, Pooh >

value to find: Tigger index returned: .1
Assertion failed: ix >= 0 &&ix < size
```

我们在这里展示 Array 的层次结构，只是为了说明多继承和虚拟继承的定义和用法。有关数组类设计的更高级的讨论见［NACKMAN94］。当然，对于数组的大多数需求，标准库 vector 类已经足够了。

练习 18.16

向 Array 添加一个额外的成员函数：spy()。实现当它被调用时，能够记住被应用在该类对象上的操作：a）索引访问的次数，b）每个成员函数被调用的次数，c）当调用 find()时，被搜索的元素值，d）元素搜索成功的次数。请说明你的设计，并修改整个 Array 子类型以便 spy()可以正常工作。

练习 18.17

关联数组是标准库 map 的另一个名字，因为它支持根据键值进行索引。你认为关联数组适合做成 Array 类的子类型吗，为什么？

练习 18.18

请用标准库模板容器类以及尽可能多的泛型算法重新实现 Array 层次结构。

19

C++中继承的用法

有了继承，指向基类类型的指针或引用就可以被用来指向派生类类型的对象。然后，我们就可以编写程序来操纵这些指针或引用，而不用考虑它们所指向的对象的实际类型。用一个基类指针或引用来操纵多个派生类型的能力被称为多态性。本章中，我们将了解C++中的三个语言特性，它们为多态性提供了特殊的支持。首先我们将了解RTTI（Run-Time Type Identification，运行时刻类型识别）的特性，它使程序能够获取由基类指针或引用所指的对象的实际派生类型。接着，我们将了解类的继承怎样影响异常处理：怎样把异常定义为类层次结构，以及基类类型的异常处理代码怎样能够处理派生类类型的异常。最后，我们将回顾函数重载解析的规则，看看类继承对于函数实参上可能的类型转换的影响，以及对于选择最佳可行函数的影响。

19.1 RTTI

RTTI（运行时刻类型识别）允许“用指向基类的指针或引用来操纵对象”的程序能够获取到“这些指针或引用所指对象”的实际派生类型。在c++中，为了支持RTTI提供了两个操作符：

1．dynamic_cast操作符，它允许在运行时刻进行类型转换，从而使程序能够在一个类层次结构中安全地转换类型，把基类指针转换成派生类指针，或把指向基类的左值转换成派生类的引用，当然只有在保证转换能够成功的情况下才可以。

2．typeid操作符，它指出指针或引用指向的对象的实际派生类型。

但是，对于要获得的派生类类型的信息，dynamic_cast和typeid操作符的操作数的类型必须是带有一个或多个虚拟函数的类类型。即，对于带有虚拟函数的类而言，RTTI操作符是运行时刻的事件，而对于其他类而言，它只是编译时刻的事件。在本节，我们将更详细地了解这两个操作符所提供的支持。

在实现某些应用程序（比如调试器或数据库程序）时，RTTI的使用是很有必要的。在这些应用程序中，只有在运行时刻通过检查“与对象的类类型一起存储的RTTI信息”，我们才能知道对象的类型。但是，我们应该尽量减少使用RTTI，而尽可能多地使用C++的静态类型系统（即编译时刻类型检查），因为它是更加安全有效的。

19.1.1 dynamic_cast 操作符

dynamic_cast 操作符可以用来把一个类类型对象的指针转换成同一类层次结构中的其他类的指针，同时也可以用它把一个类类型对象的左值转换成同一类层次结构中其他类的引用。与 C++支持的其他强制转换不同的是，dynamic_cast 是在运行时刻执行的。如果指针或左值操作数不能被转换成目标类型，则 dynamic_cast 将失败。如果针对指针类型的 dynamic_cast 失败，则 dynamic_cast 的结果是 0。如果针对引用类型的 dynamic_cast 失败，则 dynamic_cast 会抛出一个异常。在后面，我们会给出失败的 dynamic_cast 的示例。

在进一步详细了解 dynamic_cast 的行为之前，我们先来了解为什么在 C++程序中用户需要使用 dynamic_cast。假设我们的程序用类库来表示公司中不同的雇员。这个层次结构中的类都支持某些成员函数，以计算公司的薪金。例如：

```
class employee {
public:
     virtual int salary();
};

class manager : public employee {
public:
     int salary();
};

class programmer : public employee {
public:
     int salary();
};

void company::payroll( employee *pe ) {
     // 使用 pe->salary()
}
```

我们的公司有不同类型的雇员。company 成员函数 payroll()的参数是指向 employee（雇员）类的指针，它可能指向 manager（经理）类，也可能指向 programmer（程序员）类。因为 payroll()调用虚拟函数 salary()，所以，根据 pe 指向的雇员的类型，它分别调用 manager 或 programer 类改写的函数。

假设。employee 类不再能满足我们的需要，我们想要修改它。希望增加一个名为 bonus()的成员函数，在计算公司的薪金时，能与成员函数 salary()一起被使用。则可以通过向 employee 层次结构中的类增加一个虚拟成员函数来实现。例如：

```
class employee {
public:
     virtual int salary();
     virtual int bonus();
};
class manager : public employee {
public:
     int salary();
};
```

```
class programmer : public employee {
public:
     int salary();
     int bonus();
};

void company::payroll( employee *pe )
{
     // 使用 pe->salary() 和 pe->bonus()
}
```

如果 payroll()的参数 pe 指向一个 manager 类型的对象，则调用其类 employee 中定义的虚拟函数 bonus()，因为 manager 类型的对象没有改写 employee 类中定义的虚拟函数 bonus()。如果 payroll()的参数 pe 指向一个 programmer 类型的对象，则调用在 programmer 类中定义的虚拟成员函数 bonus()。

为类层次结构增加虚拟函数时，我们必需重新编译类层次结构中的所有类成员函数。如果我们能够访问类 employee、manager 和 programmer 的成员函数的源代码，那么，就可以增加虚拟函数 bonus()。但这样做并不总是可能的。如果前面的类层次结构是由第三方库提供商提供的，那么，该提供商可能只提供库中定义类接口的头文件以及库的目标文件。但他们可能不会提供类成员函数的源代码。在这种情况下，重新编译该层次结构下的类成员函数是不可能的。

如果我们希望扩展这个类库，则不能增加虚拟成员函数。但我们可能仍然希望增加功能。在这种情况下，就必需使用 dynamic_cast。

dynamic_cast 操作符可用来获得派生类的指针，以便使用派生类的某些细节，这些细节没有其他办法能够得到。例如，假设我们通过向 programmer 类增加 bonus()成员函数来扩展这个库。我们可以在 programmer 类定义（在头文件中给出）中增加这个成员函数的声明，并在我们自己的程序文本文件中定义这个新的成员用数：

```
class employee {
public:
     virtual int salary();
};

class manager : public employee {
public:
     int salary();
};

class programmer : public employee {
public:
     int salary();
     int bonus();
};
```

记住，函数 payroll()接收一个指向基类 employee 的指针作为参数。我们可以用 dynamic_cast 操作符获得派生类 programmer 的指针，并用这个指针调用成员函数 bonus()，如下所示：

```
void company::payroll( employee *pe )
{
     programmer *pm = dynamic_cast< programmer* >( pe );

     // 如果 pe 指向 programmer 类型的一个对象
     // 则 dynamic_cast 成功
     // 并且 pm 指向 programmer 对象的开始
     if ( pm ) {
          // 用 pm 调用 programmer::bonus()
     }
     // 如果 pe 不是指向 programmmer 类型的一个对象
     // 则 dynamic_cast 失败,
     // 并且 pm 的值为 0
     else {
          // 使用 employee 的成员函数
     }
}

dynamic_cast 转换
dynamic_cast< programmer* >( pe )
```

把它的操作数 pe 转换成 programmer*型。如果 pe 指向 programmer 类型的对象，则该强制转换成功。否则，转换失败，dynamic_cast 的结果为 0。

所以 dynamic_cast 操作符一次执行两个操作。它检验所请求的转换是否真的有效，只有在有效时，它才会执行转换，而检验过程发生在运行时刻。dynamic_cast 比其他 C++转换操作要安全，因为其他转换不会检验转换是否真正能被执行。

在上个例子中，如果在运行时刻 pe 实际上指向一个 programmer 对象，则 dynamic_cast 成功，并且 pm 被初始化，指向一个 programmer 对象。否则，dynamic_cast 操作的结果是 0，而 pm 被初始化为 0。通过检查 pm 的值，函数 company::payroll()知道 pe 何时指向一个 programmer 对象。然后，它可以用成员函数 programmr::Bonus()计算程序员的薪金。如果同为 pc 指向一个 manager 对象，而导致 dynamic_cast 失败，则使用一般的薪金计算，不使用新的成员函数 programmer::bonus()。

dynamic_cast 被用来执行从基类指针到派生类指针的安全转换，它常常被称为安全的向下转换（downcasting）。当我们必须使用派生类的特性，而该特性又没有出现在基类中时，我们常常使用 dynamic_cast。用指向基类类型的指针来操纵派生类类型的对象，通常通过虚拟函数自动处理。但是，在某些情形下，使用虚拟函数是不可能的。dynamic_cast 为这些情形提供了替代的机制，但是这种机制比虚拟成员函数更易出错，应该小心使用。

一种可能的错误是，在测试 dynamic_cast 的结果是否为 0 之前就使用它。如果这样做了，则与之相应的结果不见得是正确的，它未必指向一个类对象。例如：

```
void company::payroll( employee *pe ) {
     programmer *pm = dynamic_cast< programmer* >( pe );

     // 潜在的错误: 在测试 pm 的值之前使用 pm
     static int variablePay = 0;
     variablePay += pm->bonus();
     // ...
```

```
}
```

在使用结果指针之前，我们必须通过测试 dynamic_cast 操作符的结果来检验转换是否成功。company::payroll()函数的一个较好的定义如下：

```
void company::payroll( employee *pe )
{
    // dynamic_cast 和测试在同一条件表达式中
    if ( programmer *pm = dynamic_cast< programmer* >( pe ) ) {
        // 使用 pm 调用 programmer::bonus()
    }
    else {
        // 使用 employee 的成员函数
    }
}
```

dynamic_cast 操作的结果被用来初始化 if 语句条件表达式中的变量 pm，这也是必要的，因为条件中的声明也会产生值。如果 pm 不是 0，也就是说，如果因为指针 pe 实际指向一个 programmer 对象，所以 dynamic_cast 成功，则执行 if 语句的 true 分支。否则，条件中的声明产生结果 0，并执行 else 分支。因为 dynamic_cast 操作和其结果的测试出现在同一语句中，所以不可能在 dynamic_cast 和测试之间错误地插人代码，从而不可能在测试之前使用 pm。

在上个例子中，dynamic_cast 操作把一个基类指针转换成派生类指针。dynamic_cast 也可以用来把一个基类类型的左值转换成派生类类型的引用。这种 dynamic_cast 的语法如下：

```
dynamic_cast< Type& >( lval )
```

这里的 Type&是转换的目标类型，而 lval 是基类类型的左值。只要 Ival 指向的对象的类型“有一个基类或派生类是 Type 类型”，那么 dynamic_cast 操作就会把操作数 lval 转换成期望的 Type&。

因为不存在空引用（见 3.6 节），所以不可能通过比较 dynamic_cast 的结果（dynamic_cast 的结果引用）是否为 0 来检验 dynamic_cast 是否成功。如果上个例子想使用引用而不是指针，则条件：

```
if ( programmer *pm = dynamic_cast< programmer* >( pe ) )
```

就不能被改写为：

```
if ( programmer &pm = dynamic_cast< programmer& >( pe ) )
```

当 dynamic_cast 被用来转换引用类型时，它会以一种不同的方式报告错误情况。如果一个引用的 dynamic_cast 失败，则会抛出一个异常。

那么，为了使用引用类型的 dynamic_cast，上个例子必须被重写如下：

```
#include <type_info>

void company::payroll( employee &re )
{
    try {
        programmer &rm = dynamic_cast< programmer & >( re );
        // 用 rm 调用 programmer::bonus()
    }
    catch ( std::bad_cast ) {
        // 使用 employee 的成员函数
```

```
        }
    }
```

如果一个引用的 dynamic_cast 失败，则它会抛出一个 bad_cast 类型的异常。类类型 bad_cast 被定义在 C++标准库中。为了像这个例子那样在程序中引用该类型，我们必须包含头文件<type_info>（我们将在下一节了解 C++标准库中定义的异常）。

什么时候应该使用针对引用的 dynamic_cast，而不是针对指针的？这实在是程序员的选择。对引用的 dynamic_cast，不可能忽略失败的转换并在没有测试其结果前使用它而指针是有可能的。但是，使用异常给程序增加了相应的运行开销（如第 11 章所说明的），所以有些程序员可能更喜欢使用指针的 dynamic_cast。

19.1.2 typeid 操作符

RTTI 提供的第二个操作符是 typeid 操作符，它在程序中可用于获取一个表达式的类型。如果表达式是一个类类型，并且含有一个或多个虚拟成员函数，则答案会不同于表达式本身的类型。例如，如果表达式是一个基类的引用，则 typeid 会指出底层对象的派生类类型。例如：

```
#include <type_info>

programmer pobj;
employee &re = pobj;

// 我们将在下面关于 type_info 的小节中
// 看到 name()
// name() 返回 C 风格字符串: "programmer"
cout << typeid( re ).name() << endl;
```

typeid 操作符的操作数 re 的类型是 employee。但是，因为 re 是带有虚拟函数的类类型的引用，所以 typeid 操作符的结果指出，底层对象的类型是 programmer 类型（而不是操作数的类型 employee）。使用 typeid 操作符时，程序文本文件必须包含 C++标准库中定义的头文件<type_info>，如这个例子所示。

typeid 操作符可用来做什么？它用在高级的系统程序设计开发中，例如，设计构造调试器，或用来处理从数据库获取到的永久性对象。在这样的系统中，在一个调试会话中，或者向一个数据库存贮或获取对象期间，当程序通过基类指针或引用操纵一个对象时，程序需要找到被操纵的对象的实际类型，以便正确地列出对象的属性。为了找到对象的实际类型，我们可以使用 typeid。

typeid 操作符必须与表达式或类型名一起使用。例如，内置类型的表达式和常量可以被用作 typeid 的操作数。当操作数不是类类型时，typeid 操作符会指出操作数的类型：

```
int iobj;

cout << typeid( iobj ).name() << endl;     // 打印: int
cout << typeid( 8.16 ).name() << endl;     // 打印: double
```

当 typeid 操作符的操作数是类类型，但不是带有虚拟函数的类类型时，typeid 操作符会指出操作数的类型，而不是底层对象的类型：

```
class Base { /* 没有虚拟函数 */ };
class Derived : public Base { /* 没有虚拟函数 */ };

Derived dobj;
Base *pb = &dobj;

cout << typeid( *pb ).name() << endl; // 打印: Base
```

typeid 操作符的操作数是 Base 类型的，即表达式*pb 的类型。因为 Base 不是一个带有虚拟函数的类类型，所以 typeid 的结果指出，表达式的类型是 Base，尽管 pb 指向的底层对象的类型是 Derived。

可以对 typeid 的结果进行比较。例如：

```
#include <type_info>

employee *pe = new manager;
employee& re = *pe;

if ( typeid( pe ) == typeid( employee* ) )   // true
     // do something

/*
   if ( typeid( pe ) == typeid( manager* ) )       // false
   if ( typeid( pe ) == typeid( employee ) )       // false
   if ( typeid( pe ) == typeid( manager ) )        // false
*/
```

if 语句的条件子句比较“在一个表达式上应用 typeid 操作符的结果”和“用在类型名操作数上的 typeid 操作符的结果”。注意比较：

```
typeid( pe ) == typeid( employee* )
```

的结果为 true。这使得习惯写：

```
// 调用虚拟函数
pe->salary();
```

的用户有些吃惊，它导致调用 manager 派生类的函数 salary()。typeid(pe)与虚拟函数调用机制不同。这是因为操作数 pe 是一个指针，而不是一个类类型。为了要获取到派生类类型，typeid 的操作数必须是一个类类型（带有虚拟函数）。表达式 typeid(pe)指出 pe 的类型，即指向 employee 的指针。它与表达式 typeid(employee*)相等，而其他比较的结果都是 false。

当表达式*pe 被用在 typeid 上时，结果指出 pe 指向的底层对象的类型：

```
typeid( *pe ) == typeid( manager )      // true
typeid( *pe ) == typeid( employee )     // false
```

在这两个比较中，因为*pe 是一个类类型的表达式，该类带有虚拟函数，所以 typeid 的结果指出操作数所指的底层对象的类型，即 manager。

typeid 操作符也可以被用在引用上，例如：

```
typeid( re ) == typeid( manager )       // true
typeid( re ) == typeid( employee )      // false
typeid( &re ) == typeid( employee* )    // true
typeid( &re ) == typeid( manager* )     // false
```

在前两个比较中，操作数 re 是带有虚拟函数的类类型，因此 typeid 操作数的结果指出 re 指向的底层对象的类型。在后两个比较中，操作数&re 是一个类型指针。因此 typeid 操作符的结果指出操作数的类型，即 employee*。

typeid 操作符实际上返回一个类型为 type_info 的类对象。type_info 类类型被定义在头文件<type_info>中，它的类接口描述了我们可以对 typeid 操作符的结果做什么操作。我们将在下一小节看到这个接口。

19.1.3 type_info 类

type_info 类的确切定义是与编译器实现相关的，但是这个类的某些特性对每个 C++程序却都是相同的：

```
class type_info {
     // 依赖于编译器的实现
private:
     type_info( const type_info& );
     type_info& operator= ( const type_info& );
public:
     virtual ~type_info();

     int operator==( const type_info& ) const;
     int operator!=( const type_info& ) const;

     const char * name() const;
};
```

因为 type_info 类的拷贝构造函数和拷贝赋值操作符都是私有成员，所以用户不能在自己的程序中定义 type_info 对象。例如：

```
#include <typeinfo>

type_info t1; // 错误：没有缺省构造函数
              // 错误：拷贝构造函数是 private 的
type_info t2 ( typeid( unsigned int ) );
```

在程序中创建 type_info 对象的惟一途径是使用 typeid 操作符。

该类还有重载的比较操作符。这些操作符允许比较两个 type_info 对象，因此允许比较“用 typeid 操作符获得的结果”，如上小节所示：

```
typeid( re ) == typeid( manager )      // true
typeid( *pe ) != typeid( employee )    // false
```

函数 name()返回一个 C 风格字符串，它是 type_info 对象所表示的类型的名字。该函数可以被用在我们的程序中，如下所示：

```
#include <typeinfo>

int main() {
     employee *pe = new manager;

     // 输出："manager"
     cout << typeid( *pe ).name() << endl;
}
```

为了使用成员函数 name()，我们不能忘了包含头文件<typeinfo>。

类型名是惟一保证被所有 C++编译器实现提供的信息，可通过 type_info 成员函数 name()获得。正如在本节开始提到的，对 RTTI 的支持是与编译器实现相关的，而且，某些编译器可能为类 type_info 提供了其他成员函数，而没有在上面列出来。你应该查询编译器手册来找到确切的 RTTI 支持。可能提供了哪些额外的支持？基本上，编译器为一个类型提供的任何可能的信息都可以被加进来。例如：

1．类成员函数清单；

2．内存中该类类型对象的布局是什么样的，即，成员和基类子对象是怎样被映射的。

编译器用来扩展 RTTI 支持的一种常见技术是，为“从 type_info 派生的类类型”增加额外的信息、因为 type_info 类含有一个虚拟析构函数，所以 dynamic_cast 操作符可以被用来判断是否有可用的特殊类型的 RTTI 扩展支持。例如，我们假设一个编译器通过一个名为 extended_type_info 的类为 RTTI 提供额外的支持。extended_type_info 是一个从 type_info 派生的类。通过使用 dynamic_cast，一个程序可以发现“typeid 操作符返回的 type_info 对象”是否为 extended_type_info 类型，在程序中是否可以使用额外的 RTTI 支持，如下：

```
#include <typeinfo>

// typeinfo 头文件包含 extended_type_info 的定义
typedef extended_type_info eti;

void func( employee* p )
{
    // 从 type_info* 到 extended_type_info* 向下转换
    if ( eti *eti_p = dynamic_cast<eti *>( &typeid( *p ) ) )
    {
        // 如果 dynamic_cast 成功
        // 通过 eti_p 使用 extended_type_info 信息
    }
    else
    {
        // 如果 dynamic_cast 失败
        // 使用标准 type_info 信息
    }
}
```

如果 dynamic_cast 成功，则 typeid 操作符返回一个 extended_type_info 类型的对象，意味着该编译器提供了额外的 RTTI 支持，可供程序使用。如果 dynamic_cast 失败，则只有基本的 RTTI 支持可以被程序使用。

练习 19.1

已知下列类层次结构，其中每个类都定义了一个缺省构造函数和一个虚拟析构函数：

```
class X { ... };
class A { ... };
class B : public A { ... };
```

```
class C : public B { ... };
class D : public X, public C { ... };
```

则下列哪些 dynamic_cast 会失败？

```
(a)  D *pd = new D;
     A *pa = dynamic_cast< A* >( pd );

(b)  A *pa = new C;
     C *pc = dynamic_cast< C* >( pa );

(c)  B *pb = new B;
     D *pd = dynamic_cast< D* >( pb );

(d)  A *pa = new D;
     X *px = dynamic_cast< X* >( pa );
```

练习 19.2

请说明应该在什么时候用 dynamic_cast 代替虚拟函数。

练习 19.3

请用练习 19.1 定义的类层次结构，重新改写下列代码段来执行一个针对引用的 dynamic_cast，把*pa 表达式转换成类型 D&：

```
if ( D *pd = dynamic_cast< D* >( pa ) )
     // 使用 D 的成员
}
else {
     // 使用 A 的成员
}
```

练习 19.4

已知下列类层次结构，其中每个类都定义了一个缺省构造函数和一个虚拟析构函数：

```
class X { ... };
class A { ... };
class B : public A { ... };
class C : public B { ... };
class D : public X, public C { ... };
```

下列情况都会向标准输出打印出哪个类型名？

```
(a)  A *pa = new D;
     cout << typeid( pa ).name() << endl;

(b)  X *px = new D;
     cout << typeid( *px ).name() << endl;

(c)  C cobj;
```

```
        A& ra = cobj;
        cout << typeid( &ra ).name() << endl;

(d)     X *px = new D;
        A& ra = *px;
        cout << typeid( ra ).name() << endl;
```

19.2 异常和继承

异常处理是针对运行时刻的程序异常而提供的语言层次上的标准设施。C++为异常处理提供了一套统一的语法和风格，同时也允许程序员对异常处理设施进行细微的调整。第 11 章曾经介绍过 C++对于异常处理的基本支持，说明了一个程序如何抛出一个异常，当异常被抛出的时候程序的控制权如何被转移到异常处理代码中，以及异常处理代码怎样与 try 块相关联。

当类类型的层次结构被用于异常时，异常处理的方式变得更加多样化。本节我们将了解怎样写程序来抛出和处理来自这样的层次结构的异常。

19.2.1 定义为类层次结构的异常

在第 11 章中，我们用两个类类型来描述由 iStack 类成员函数抛出的异常种类：

```
class popOnEmpty { ... };
class pushOnFull { ... };
```

在实际的 C++程序中，表示异常的类类型通常被组织成一个组（group）或一个层次结构。对于我们这两个异常类，异常的层次结构会是什么样的呢？

我们可以定义一个被称为 Excp 的基类，然后再从它派生出两个异常类。该基类封装了两个派生类公共的数据成员和成员函数：

```
class Excp { ... };
class popOnEmpty : public Excp { ... };
class pushOnFull : public Excp { ... };
```

基类 Excp 可以提供的一个操作是打印错误信息。层次结构中的两个异常类都可以使用这一措施：

```
class Excp {
public:
    // 打印错误信息
    static void print( string msg ) {
        cerr << msg << endl;
    }
};
```

我们可以进一步精炼该异常类层次结构。我们可以从基类 Excp 派生其他的类，以更细致的方式描述程序可能检测到的异常。

```
class Excp { ... };

class stackExcp : public Excp { ... };
```

```
        class popOnEmpty : public stackExcp { ... };
        class pushOnFull : public stackExcp { ... };
    class mathExcp : public Excp { ... };
        class zeroOp : public mathExcp { ... };
        class divideByZero : public mathExcp { ... };
```

这些进一步的精炼允许更精确地识别出在程序中发生的不正常情况。其他的异常类被组织成几个层次。随着层次结构的加深，每一层都变成一个更加特定的异常。例如，上面给出的异常结构中的第一层，也是最一般化的层由 Excp 类表示。第二层把 Excp 类特化成两个不同的类：stackExcp（针对在操纵 iStack 类时出现的异常），以及 mathExcp（针对在数学库的函数中发生的异常）。层次结构的第三层，也是最特化的层次更进一步细化异常类。类 popOnEmpty 和 pushOnFull 定义了两种 stackExcp 异常，而类 zeroOp 和 divideByZero 定义了两种 mathExcp 异常。

在下一小节中，我们将了解怎样抛出和处理刚刚定义的层次结构中的异常类。

19.2.2 抛出类类型的异常

既然我们已经较详细地了解了类类型，那么现在让我们来看一看，当 iStack 成员函数 push()抛出一个异常时会怎么样：

```
void iStack::push( int value )
{
    if ( full() )
        // value 被存储在异常对象中.
        throw pushOnFull( value );
    // ...
}
```

执行该 throw 表达式会发生许多个步骤：

1．throw 表达式通过调用类类型 pushOnFull 的构造函数创建一个该类的临时对象。

2．创建一个 pushOnFull 类型的异常对象，并传递给异常处理代码，该异常对象是第 1 步 throw 表达式创建的临时对象的拷贝。它通过调用 pushOnFull 类的拷贝构造函数而创建。

3．在开始查找异常处理代码之前，在第 1 步中由 throw 表达式创建的临时对象被销毁。

你或许想知道为什么需要第 2 步，即，为什么要创建一个异常对象？throw 表达式：

```
pushOnFull( value );
```

创建了一个临时对象，它在 throw 表达式结束时被销毁。但是，在找到异常处理代码之前，该异常必须一直持续存在，而为了找到异常处理代码，或许要经过函数调用链中更上层的许多函数。所以，必须把这个临时对象拷贝到一个被称为异常对象（exception object）的存贮区中，它保证会持续到异常被处理完毕。在某些情况下，编译器可能能够直接创建异常对象，而不需要创建第 1 步的临时对象。但是，消除该临时对象并不是 C++标准的要求，而且并不总是能够做到的。

因为异常对象是通过拷贝 throw 表达式的值而创建的，所以，抛出来的异常总是在 throw 表达式中指定确切的类型。例如：

```
void iStack::push( int value ) {
```

```
        if ( full() ) {
            pushOnFull except( value );
            stackExcp *pse = &except;
            throw *pse;   // 异常对象的类型为 stackExcp
        }
        // ...
    }
```

表达式*pse 的类型为 stackExcp。被创建的异常对象的类型是 stackExcp，即使 pse 指向一个实际类型为 pushonFull 的对象。由 throw 表达式指向的对象的实际类型也不会被用来创建异常对象。所以该异常不能被 pushOnFull 类型的 catch 子句处理。

throw 表达式的动作暗示了：可被用来创建异常对象的类的种类会受到某些限制。如果下列的情形发生，则在 iStack 成员函数 push()中的 throw 表达式是错误的：

1．pushOnFull 类没有能接收 int 型实参的构造函数，或者，该构造函数不可访问；

2．pushOnFull 的拷贝构造函数或析构函数不可访问；

3．pushOnFull 是抽象基类，因为程序不能创建抽象类类型的对象（如 17.1 节所说明）。

19.2.3 处理类类型的异常

当异常被组织成类层次结构时，类类型的异常可能会被该类类型的公有基类的 catch 子句捕获到。例如，pushOnFull 类型的异常可以被 stackExcp 或 Excp 类型异常所对应的 catch 子句处理。

```
int main( ) {
    try {
        // ...
    }
    catch ( Excp ) {
        // 处理 popOnEmpty 和 pushOnFull 异常
    }
    catch ( pushOnFull ) {
        // 处理 pushOnFull 异常
    }
}
```

在上个例子中，cath 子句的顺序不是最优的。你能看出为什么吗？记住，catch 子句的检查顺序是它们在 try 块后出现的顺序。一旦编译器为一个异常找到了一个 catch 子句，就不会再检查进一步的 catch 子句。在上个例子中，因为 Excp 的 catch 子句也处理 pushOnFull 类型的异常，所以为 pushOnFull 指定的 catch 子句永远也不会被执行。catch 子句的正确顺序如下所示：

```
catch ( pushOnFull ) {
    // 处理 pushOnFull 异常
}
catch ( Excp ) {
    // 处理其他异常
}
```

派生类类型的 catch 子句必须先出现。这确保了只有在没有其他 catch 子句适用时，才会进入基类类型的 catch 子句。

当异常被组织成类层次结构时，类库的用户就可以选择粒度层次，应用程序将在一定的层次上处理从库中抛出的异常。例如，在写函数 main()时，我们决定应用程序将以某种特定的方式处理 pushOnFull 类型的异常，这也是我们为这种异常提供专门 catch 子句的原因。我们还决定，应用程序将以更一般的方式处理所有其他的异常。例如：

```
catch ( pushOnFull eObj ) {
    // 使用 pushOnFull 类的成员函数 value()
    // 见 11.3 节
    cerr << "trying to push the value " << eObj.value()
         << " on a full stack\n";
}
catch ( Excp ) {
    // 使用基类成员函数 print()
    Excp::print( "an exception was encountered" );
}
```

如 11.3 节所提到的，为抛出的异常找到 catch 子句的过程不像函数重载解析，在函数解析期间，选择最佳可行函数时，要考虑所有在调用点可见的候选函数。在异常处理期间，异常的 catch 子句不必是与异常最匹配的 catch 子句。被选中的 catch 子句是最先匹配到的，即，遇到的第一个可以处理该异常的 catch 子句。这就是为什么“在 catch 子句列表中最特化的 catch 子句必须先出现”的原因。

catch 子句的异常声明，即关键字 catch 后面的括号中的声明，与函数参数的声明十分类似。在上一个例子中，异常声明类似于一个按值传递的参数。对象 eobj 以异常对象的值的拷贝作为初始值，其方式与“用实参值的一个拷贝初始化相应的按值传递的函数参数”相同。如同函数参数的情形一样，catch 子句的异常声明也可以被改变成引用声明。那么，catch 子句可以直接引用被 throw 表达式创建的异常对象，而不是创建自己的局部拷贝。我们知道，类类型的参数应该被声明为引用，以防止大型类对象的不必要的拷贝动作，基于同样的原因，对于类类型的异常，其异常声明最好也被声明为引用。根据异常声明是一个对象还是一个引用，catch 子句的行为会有所不同（我们将在本节了解这些内容）。

第 11 章介绍了 rethrow 表达式，它被一个 catch 子句用来把一个异常传递给函数调用列表中、更上层的另一个 catch 子句。rethrow 表达式的形式如下：

```
throw;
```

当这样的 rethrow 表达式被放在基类类型的 catch 子句中时，它的行为会怎么样呢？例如，如果 mathFunc()抛出一个 divideByZero 类型的异常，那么被重新抛出的异常类型是什么？

```
void calculate( int parm ) {
    try {
        mathFunc( parm );  // 抛出 divideByZero 异常
    }
    catch ( mathExcp mExcp ) {
        // 部分地处理当前异常
        // 并重新抛出该异常对象
        throw;
    }
}
```

被重新抛出的异常类型是 mathFunc()抛出的异常的类型（即 divideByZero），或者是在

catch 子句的异常声明中的类型（即 mathExcp）吗？

记住，throw 表达式重新抛出的是原来的异常对象。由于原来的异常对象是 divideByZero，所以重新抛出的异常也是 divideByZero。在 catch 子句中，对象 mExcp 以“divideByZero 异常对象的 MathExcp 基类子对象的一份拷贝”作为初始值。该拷贝只在 catch 子句中被访问，而不是被重新抛出的原来的异常对象。

假设在我们的异常层次结构中的类类型有析构函数，例如：

```
class pushOnFull {
public:
    pushOnFull( int i ) : _value( i ) { }
    int value() { return _value; }
    ~pushOnFull();    // 新声明的析构函数
private:
    int _value;
};
```

应该在何时调用这个析构函数？为了回答该问题，我们需要更详细地检查 catch 子句：

```
catch ( pushOnFull eObj ) {
    cerr << "trying to push the value " << eObj.value()
         << " on a full stack\n";
}
```

因为异常声明把 eObj 声明为 catch 子句的局部对象，而且 pushOnFull 类本身就有析构函数，所以当 catch 子句退出时，eObj 也被销毁了。但是，对于异常抛出时创建的异常对象，它的析构函数何时才被调用呢？

你可能会有几种猜测，一种是在进入 catch 子句时，另一种是在 catch 子句结束时。但是，如果异常对象在这两点被销毁，则它可能被销毁得太早了。你能看出其中的原因吗？如果 catch 子句重新抛出异常，并把这个异常对象传递给在函数调用链中的上一级 catch 子句，那么，在到达最后一个处理该异常的 catch 子句之前，异常对象是不能被销毁的。因此，对于一个异常对象，直到该异常的最后一个 catch 子句退出时，它才被销毁。

19.2.4 异常对象和虚拟函数

如果被抛出的异常对象是派生类类型的，并且它被针对基类的 catch 子句处理，则 catch 于句一般不能使用派生类类型的特性。例如，在异常类 pushOnFull 中声明的成员函数 value()，不能被用在处理 Excp 类型异常的 catch 子句中。

```
catch ( Excp &eObj )
{
    // 错误: Excp 没有成员函数 value()
    cerr << "trying to push the value " << eObj.value()
         << " on a full stack\n";
}
```

我们可以通过重新设计异常类层次结构来定义虚拟函数，然后再在针对基类 Excp 的 catch 子句中，通过这些虚拟函数来调用派生类类型中更为特化的成员函数。例如：

```
// 定义了虚拟函数的新类定义
class Excp {
public:
```

```
        virtual void print() {
            cerr << "An exception has occurred"
                 << endl;
        }
    };
    class stackExcp : public Excp { };

    class pushOnFull : public stackExcp {
    public:
        virtual void print() {
            cerr << "trying to push the value " << _value
                 << " on a full stack\n";
        }
        // ...
    };
```

则 print()函数可以被用在 catch 子句中，如下所示：

```
    int main( ) {
        try {
            // iStack::push() throws a pushOnFull exception
        } catch ( Excp eObj ) {
            eobj.print();     // 调用虚拟函数
                              // 喔！调用基类实例
        }
    }
```

即使被抛出的异常是 pushOnFull 类型的，并且函数 print()是一个虚拟函数，但是语句 eObj.print()仍输出下列行：

```
    An exception has occurred
```

被调用的 print()函数是基类 Excp 的成员函数，而不是派生类 pushOnFull 改写的函数。为什么调用的不是派生类的 print()函数呢？

记住，catch 子句的异常声明的行为与参数声明十分相似，在进入 catch 子句时，因为异常声明在这里声明了一个对象，所以 eObj 以“异常对象的基类子对象 Excp 的一个拷贝”作为初始值。eobj 是 Excp 类型的对象，而不是 pushOnFull 类型的对象。为了调用派生类对象的虚拟函数，异常声明必须声明一个指针或引用。例如：

```
    int main( ) {
        try {
            // iStack::push() 抛出一个 pushOnFull 异常
        }
        catch ( Excp &eObj ) {
            eobj.print();        // 调用虚拟函数 pushOnFull::print()
        }
    }
```

在这个例子中，catch 子句的异常声明也是基类 Excp 类型的，但是，因为 eObj 是一个引用，而且 eObj 指向了 pushOnFull 类型的异常对象，所以 eObj 可以被用来调用 pushOnFull 类型定义的虚拟函数。当 catch 子句调用 print()虚拟函数时。调用的是派生类 pushOnFull 的函数 print()，程序将输出下列行：

```
    trying to push the value 879 on a full stack
```

所以，把“catch 子句的异常声明”声明为引用的另一个原因是，确保能正确地调用与异常类型相关联的虚拟函数。

19.2.5 栈展开和析构函数调用

当一个异常被抛出时，为了寻找能处理该异常的 catch 子句，需要从抛出异常的函数内开始，向上通过嵌套的函数调用链，直到找到该异常的 catch 子句为止。在函数调用链中查找 catch 子句的过程被称作**栈展开**（stack unwinding），我们在 11.3 节第一次引入了栈展开的概念。

在栈展开期间，随着函数调用链中的函数在 catch 子句的查找期间退出，每个函数执行的动作也被立即结束。如果一个函数获得了资源（例如，如果它打开了一个文件或在空闲存贮区中分配了空间），那么就不大好了，这些资源将永远不会被释放。

有一种程序设计技术，使程序员能够避免这样的限制。在栈展开期间，由于栈被展开，每次一个复合语句或语句块退出时，如果在退出的块中有某一个局部对象是类类型的，则在复合语句或函数退出时，栈展开过程将自动调用该对象的析构函数（关于局部对象在 8.1 节描述）。

例如，下面的类在其构造函数中封装了“在空闲存贮区中分配一个 int 数组”的动作，并在其析构函数中封装了“释放这块内存”的逆动作：

```
class PTR {
public:
    PTR() { ptr = new int[ chunk ]; }
    ~PTR() { delete[] ptr; }
private:
    int *ptr;
};
```

如下面的函数 mainp()所示，在函数 mathFunc()被调用之前，这种类型的局部对象首先被创建：

```
void manip( int parm ) {
    PTR localPtr;
    // ...
    mathFunc( parm ); // 抛出 divideByZero 异常
    // ...
}
```

如果 mathFunc()抛出一个 divideByZero 类型的异常，则栈展开过程在函数调用链中向查找该异常的 catch 子句。在栈展开过程中，函数 manip()会被检查到。因为函数 mathFunc()调用没有被放在 try 块中，所以不会在 manip()中查找针对该异常的 catch 子句。栈展开过程继续向上遍历函数调用链，到达调用函数 manip()的函数（即上一个调用函数）。然而，在 manip()带着这个未处理的异常退出之前，栈展开过程会销毁 manip()中所有“在调用函数 mathFunc()之前被创建的”局部类对象。栈展开过程在函数调用链中继续向上前进，在前进之前，局部对象 localPtr 先被销毁，由 localPtr 指向的空闲存贮区也被释放，以防止内存泄漏。

这就是为什么我们说，C++异常处理过程反映的是被称为“资源获取是初始化，资源释放是析构”的程序设计技术的原因。如果一个资源被实现为一个类，并且“要求获得资源的

动作”被封装在类的构造函数中，“释放资源的动作”被封装在类的析构函数中，比如我们的 PTR 类，那么，如果一个函数带着未处理的异常退出时，函数中这种类类型的局部对象将被自动销毁。因此，任何必须发生的资源释放动作，当它们被封装在类的析构函数中时，都不会被栈展开过程跳过去。

你可能还记得在 8.4 节中介绍的 auto_ptr 设施，它在 C++标准库中被定义，该设施与 PTR 类十分相似。它在构造函数中封装了对空闲存贮区的分配动作，而在析构函数中封装了对该内存的释放动作。当我们用 auto_ptr 在空闲存贮区中分配单个对象时，可以保证在复合语句或函数带着未处理的异常退出时（在栈展开期间），该空闲存贮区会被正确地释放。

19.2.6 异常规范

通过使用异常规范（exception specification），函数声明可以指定该函数能够直接或间接抛出的异常集合。异常规范保证该函数不会抛出任何没有被列在异常规范中的异常。11.4 节首次介绍了异常规范，关于异常规范和类类型我们还有一些事情需要提及。

首先，异常规范可以为类成员函数指定，与非成员函数一样，成员函数声明的异常规范也是跟在函数参数表的后面。例如，C++标准库中的 bad_alloc 类被定义成：它的所有成员函数都有一个空的异常规范 throw()，这表示它的成员函数保证不会抛出任何异常：

```
class bad_alloc : public exception {
     // ...
public:
     bad_alloc() throw();
     bad_alloc( const bad_alloc & ) throw();
     bad_alloc & operator=( const bad_alloc & ) throw();
     virtual ~bad_alloc() throw();
     virtual const char* what() const throw();
};
```

我们注意到，如果一个成员函数被声明为 const 或 volatile 成员函数［如上个例子中的 what()］，则异常规范跟在函数声明的 const 和 volatile 限定修饰符之后。

同一个函数所有声明中的异常规范都必须指定相同的类型。对于成员函数，如果该函数被定义在类定义之外，则其定义所指定的异常规范，必须与“类定义中该成员函数声明中的异常规范”相同。例如：

```
#include <stdexcept>

// <stdexcept> defines class overflow_error
class transport {
     // ...
public:
     double cost( double, double ) throw ( overflow_error );
     // ....
};
// 错误：异常规范不同于类成员表中的声明
double transport::cost( double rate, double distance ) { }
```

基类中虚拟函数的异常规范，可以与派生类改写的成员函数的异常规范不同。但是，派生类虚拟函数的异常规范必须与基类虚拟函数的异常规范一样或者更严格。例如：

```
class Base {
public:
     virtual double f1( double ) throw ();
     virtual int f2( int ) throw ( int );
     virtual string f3( ) throw ( int, string );
     // ...
};
class Derived : public Base {
public:
     // 错误：异常规范没有 base::f1() 的严格
     double f1( double ) throw ( string );
     // ok: 与 base::f2() 相同的异常规范
     int f2( int ) throw ( int );
     // ok: 派生 f3() 更严格
     string f3( ) throw ( int );
     // ...
};
```

为什么派生类成员函数的异常规范必须与基类函数一样严格或者更严格？因为这可以确保当派生类的虚拟函数被通过基类类型的指针调用时，该调用保证不会违背基类成员函数的异常规范。例如：

```
// 保证不会抛出异常
void compute( Base *pb ) throw()
{
     try {
          pb->f3( );    // 可能抛出 int 或者 string 类型的异常
     }
     // 处理来自 Base::f3() 的异常
     catch ( const string & ) { }
     catch ( int ) { }
}
```

Base 类中的 f3()声明保证该函数只会抛出 string 和 int 类型的异常。函数 compute()根据这个保证，只定义了处理这些异常的 catch 子句。因为 Derived 中的 f3()比 Base 中的 f3()更严格，所以当我们对 Base 类的接口进行编程时，程序不会违背我们的规范。

最后，在第 11 章中我们提到过，在“被抛出的异常的类型”与“异常规范中指定的类型”之间不允许进行类型转换。这个规则有一个小小的例外：当异常规范指定一个类类型或类类型的指针时。如果一个异常规范指定了一个类，则该函数可以抛出“从该类公有派生的类类型”的异常对象。（对指针来说也是类似的，如果异常规范指定了一个类的指针，则该函数可以抛出“从该类公有派生的类的指针类型”的异常对象。）例如：

```
class stackExcp : public Excp { };
class popOnEmpty : public stackExcp { };
class pushOnFull : public stackExcp { };

void stackManip() throw( stackExcp )
{
     // ...
}
```

异常规范不但指出 stackManip()可以抛出 stackExcp 类型的异常，而且还可以抛出 popOnEmpty 或 pushOnFull 类型的异常。以前曾经提到过，从一个基类公有派生出的派生类反映了一种 is-a 的关系。并且为更一般化的基类提供了一个特化。因为 popOnEmpty 和 pushOnFull 异常是一种 stackExcp，所以这些异常并没有违背 stackManip()的异常规范。

19.2.7 构造函数和函数 try 块

我们可以把整个函数体包含在一个 try 块中，这种 try 块被称为函数 try 块（function try block）（我们在 11.2 节首次讨论了函数 try 块）。例如：

```
int main()
try {
    // main() 的函数体
}
catch ( pushOnFull ) {
    // ...
}
catch ( popOnEmpty ) {
    // ...
}
```

函数 try 块把一组 catch 子句与一个函数体相关联起来。如果函数体中的语句抛出了一个异常，则考虑用函数体后面的处理代码来处理这个异常。

函数 try 块对于类的构造函数来说是必需的。让我们来看看原因，构造函数的定义形式如下：

```
class_name( parameter_list )
    // 成员初始化表
    : member1( expression1 ) , // member1 的初始化
      member2( expression2 )   // member2 的初始化
// function body:
{ /* ... */ }
```

expression1 和 expression2 可能是任何类型的表达式。尤其是，这些表达式调用的函数可能会抛出异常。

我们重新使用第 14 章定义的 Account 类，以展示一个更具体的例子。Account 构造函数可以被重新定义如下：

```
inline Account::
Account( const char* name, double opening_bal )
    : _balance( opening_bal - ServiceCharge() )
{
    _name = new char[ strlen(name)+1 ];

    strcpy( _name, name );
    _acct_nmbr = get_unique_acct_nmbr();
}
```

在成员初始化表中，对于成员 balance 的初始化需要调用 ServiceCharge()函数，它可能会抛出一个异常。如果我们想处理“在 Account 类型对象的构造期间所调用的函数”抛出的所有异常，那么，我们该怎样实现这个构造函数呢？

显然，把 try 块放在函数体中不会起任何作用。例如：

```
inline Account::
Account( const char* name, double opening_bal )
        : _balance( opening_bal - serviceCharge() )
{
     try {
          _name = new char[ strlen(name)+1 ];
          strcpy( _name, name );
          _acct_nmbr = get_unique_acct_nmbr();
     }
     catch ( ...) {
          // 特殊处理
          // 不能捕获来自成员初始化表的异常
     }
}
```

因为 try 块不会包含成员初始化表，所以在构造函数体尾部的 catch 子句不会考虑由“在成员初始化表中函数 ServiceCharge()”抛出的异常。

使用函数 try 块是保证“在构造函数中捕获所有在对象构造期间抛出的异常”的惟一解决方案。我们可以为 Account 类的构造函数定义函数 try 块，如下所示：

```
inline Account::
Account( const char* name, double opening_bal )
try
     : _balance( opening_bal - serviceCharge() )
{
     _name = new char[ strlen(name)+1 ];
     strcpy( _name, name );
     _acct_nmbr = get_unique_acct_nmbr();
}
catch ( ... )
{
     // 特殊处理
     // 现在能够捕获来自 ServiceCharge() 的异常了
}
```

我们注意到，关键字 try 被放在成员初始化表之前，try 块的复合语句包围了构造函数体。现在 catch 子句 catch(...)被考虑用来处理“从成员初始化表或构造函数体内抛出的所有异常”。

19.2.8 C++标准库的异常类层次结构

在本节开始时，我们引入了一个异常类层次结构，我们的程序可以用这个层次结构来报告程序不正常情况。C++标准库也提供了一个异常类层次结构，它被用来报告 C++标准库中的函数执行期间遇到的程序不正常情况。这些异常类也可以被用在我们编写的程序中，或者被进一步派生，以便描述我们所写的程序中的异常。

C++标准库中的异常层次的根类被称为 exception。这个类被定义在库的头文件<exception>中，它是 C++标准库函数抛出的所有异常的基类。exception 类的接口如下：

```
namespace std {
```

```
        class exception {
        public:
            exception() throw();
            exception( const exception & ) throw();
            exception& operator=( const exception& ) throw();
            virtual ~exception() throw();
            virtual const char* what() const throw();
        };
    }
```

与 C++标准库中定义的所有类一样，exception 类被放在名字空间 std 中，以防止污染程序中的全局名字空间。

类定义中的前四个函数分别是缺省构造函数、拷贝构造函数、拷贝赋值操作符以及析构函数。因为这些成员函数都是公有成员函数，所以任何程序都可以自由地创建、拷贝和赋值异常对象。析构函数是一个虚拟函数，这使得从 exception 类派生的类的定义更加容易。

在这个成员函数清单中，最有趣的函数是 what()，它返回一个 C 风格字符串。C 风格字符串的目的是为被抛出的异常提供某种文本描述。函数 what()是一个虚拟函数。从 exception 类派生的类可以用自己的版本改写函数 what()，以便更好地描述派生的异常对象。

我们注意到，exception 类定义中的所有函数都有一个空的异常规范 throw()。这表示 exception 类的成员函数不会抛出任何异常。程序可以随意地操纵 exception 对象（例如在 exception 类型的异常的 catch 子句中），而无需担心内部函数在创建、拷贝以及销毁 exception 对象时会抛出异常。

除了根 exception 类，C++标准库还提供了一些类，它们可被用在我们所编写的程序中，以报告程序的不正常情况。在这些预定义的类所反映的错误模型中，错误被分成两个大类：**逻辑错误**（logic error）和**运行时刻错误**（run-time error）

逻辑错误是那些由于程序的内部逻辑而导致的错误。逻辑错误是可以避免的，且在程序开始执行之前，能够被检测到。例如，违反了逻辑的先决条件，或者违反了类的不变性，两者都是逻辑错误。在 C++标准库中定义的逻辑错误如下：

```
    namespace std {
        class logic_error : public exception {
        public:
            explicit logic_error( const string &what_arg );
        };
        class invalid_argument : public logic_error {
        public:
            explicit invalid_argument( const string &what_arg );
        };
        class out_of_range : public logic_error {
        public:
            explicit out_of_range( const string &what_arg );
        };
        class length_error : public logic_error {
        public:
            explicit length_error( const string &what_arg );
        };
        class domain_error : public logic_error {
```

```
        public:
            explicit domain_error( const string &what_arg );
        };
    }
```

如果函数接收到一个无效的实参，那么就会抛出一个 invalid_argument 异常。而如果函数接收到一个不在期望范围内的实参，则它可以抛出一个 out_of_range 异常。函数还可以通过抛出一个 length_error 异常，来报告企图产生一个“长度值超出最大允许值”的对象。另外编译器可能会抛出一个 domain_error 异常来报告域错误（domain error）。

与此相对，运行时刻错误是由于程序域之外的事件而引起的错误。运行时刻错误只在程序执行时才是可检测的。在 C++标准库中定义的运行时刻错误如下：

```
namespace std {
    class runtime_error : public exception {
    public:
        explicit runtime_error( const string &what_arg );
    };
    class range_error : public runtime_error {
    public:
        explicit range_error( const string &what_arg );
    };
    class overflow_error : public runtime_error {
    public:
        explicit overflow_error( const string &what_arg );
    };
    class underflow_error : public runtime_error {
    public:
        explicit underflow_error( const string &what_arg );
    };
}
```

函数可以通过抛出 range_error 异常，来报告内部计算中的范围错误。函数既可以抛出 overflow_error 异常来报告算术溢出错误，又可以抛出 underflow_error 异常来报告算术下溢错误。

exception 类也是 bad_alloc 异常的基类。当 new()操作符不能分配所要求的存贮区时（如 8.4 节所提及的），它会抛出一个 bad_alloc 异常。exception 类也是 bad_cast 异常的基类，当引用 dynamic_cast 失败时，程序会抛出 bad_cast 异常（如 19.1 节所提及）。

让我们为 16.12 节给出的类模板 Array 重新定义 operator[]()，如果索引值越界，那么它会抛出一个 out_of_range 类型的异常：

```
#include <stdexcept>
#include <string>
template <class elemType>
class Array {
public:
    // ...
    elemType& operator[]( int ix ) const
    {
        if ( ix < 0 || ix >= _size )
```

```
            {
                string eObj =
                    "out_of_range error in Array<elemType>::operator[]()";
                throw out_of_range( eObj );
            }
            return _ia[ix];
        }
        // ...
    private:
        int _size;
        elemType *_ia;
    };
```

为了使用预定义的异常类，我们的程序必须包含头文件<stdexcept>。传递给 out_of_range 构造函数的 string 对象 eObj 描述了被抛出的异常。当该异常被捕获到时，通过 exception 类的 what()成员函数可以获取这些信息，如下所示：

```
int main()
{
    try {
        // main() 函数同 16.2 节中定义
    }
    catch ( const out_of_range &excp ) {
        // 打印：
        // out_of_range error in Array<elemType>::operator[]()
        cerr << excp.what() << "\n";
        return -1;
    }
}
```

有了这份实现，函数 try_array()中的越界索引值将导致 Array 的 operator[]()抛出一个 out_of_range 类型的异常，它将在 main()中被捕获到。

练习 19.5

下列函数可能会抛出哪些异常？

```
#include <stdexcept>
(a) void operate() throw( logic_error );
(b) int mathOper( int ) throw( underflow_error, overflow_error );
(c) char manip( string ) throw( );
```

练习 19.6

请说明 C++异常处理怎样支持名为“资源获取是初始化，资源释放是析构”的程序设计技术。

练习 19.7

为什么 try 块后的 catch 子句表是不正确的，怎样修正它？

```
#include <stdexcept>
int main() {
```

```
        try {
            // 使用 C++ 标准库
        }
        catch( exception ) {
        }
        catch( const runtime_error &re ) {
        }
        catch( overflow_error eobj ) {
        }
    }
```

练习 19.8

已知一个基本的 C++程序：

```
int main() {
    // 使用 C++ 标准库
}
```

请修改 main()，以捕获 C++标准库中的函数所抛出的任何异常。异常处理代码应该在调用 abort()（在头文件<cstdlib>中定义）已结束 main()之前，打印出与这个异常相关联的错误信息。

19.3 重载解析过程和继承 ※

类继承会影响到函数重载解析过程的所有方面。记住，函数重载解析过程有以下三步：

1．选择候选函数；

2．选择可行函数；

3．选择最佳匹配函数。

（见 9.2 节的详细讨论。）

候选函数的选择会受到继承机制的影响，是因为与基类相关联的函数，即基类的成员函数或者是在“基类被定义的名字空间”内声明的函数，都将在选择候选函数时被考虑。可行函数的选择会受到继承机制的影响，是因为对于从实参到可行函数参数的类型转换，将考虑一个更大的用户定义转换集。最佳可行函数的选择也受到继承机制的影响，因为继承机制影响了“可以把实参转换成函数参数类型”的转换序列的等级。在本节中，我们将详细地了解继承机制对于函数重载解析过程三步骤的影响。

19.3.1 候选函数

继承机制影响函数重载解析过程的第一步——为一个调用建立候选函数集。继承机制在第一步上的影响会根据该调用是如下形式的普通函数调用：

```
func( args );
```

还是用成员访问操作符点（.）或箭头（->）调用成员函数：

```
object.memfunc( args );
pointer->memfunc( args );
```

而有所不同。在这个小节中我们将依次了解继承机制对于每一种情况的影响。

假如普通函数调用的实参是类类型、类类型的引用或类类型的指针，并且该类类型是在一个名字空间内被定义的，则在该名字空间内声明的、与被调函数同名的函数都是候选函数，即使这些函数在调用点上并不可见（在 15.10 节对此有更详细的讨论）。在继承机制下，如果一个实参是类类型、类类型的引用或者类类型的指针，并且该类有基类（可能会有许多基类），则在“定义基类的名字空间”内声明的、并且与被调函数同名的函数，也被加入到候选函数集中。例如：

```
namespace NS {
    class ZooAnimal { /* ... */ };
    void display( const ZooAnimal& );
}

// Bear 基类在名字空间 NS 中声明
class Bear : public NS::ZooAnimal { };

int main() {
    Bear baloo;
    display( baloo );
    return 0;
}
```

实参 baloo 的类型为 Bear，display()调用的候选函数不但包括在调用点上可见的函数，而且还包括在声明 Bear 类及其基类 ZooAnimal 的名字空间中的函数。在名字空间 NS 中的函数 display(const ZooAnimal&）就被加入到候选函数集中。

如果实参的类型是一个类，并且该类的定义声明了与被调函数同名的友元函数，则这些友元函数也是候选函数，即使这些友元函数的声明在调用点上不可见（如 15.10 节所示）。在继承机制下，如果实参的类型是一个带有基类的类，那么，与被调函数同名的、在基类定义中声明的友元函数也将被加入到候选函数集中。例如，我们把前面所示的 dsplay()函数声明为 ZooAnimal 类的友元函数：

```
namespace NS {
    class ZooAnimal {
        friend void display( const ZooAnimal& );
    };
}

// Bear 基类在名字空间 NS 中声明
class Bear : public NS::ZooAnimal { };

int main() {
    Bear baloo;
    display( baloo );
    return 0;
}
```

函数实参 baloo 的类型为 Bear，它的基类 ZooAnimal 把 display()函数声明为友元。display()函数是名字空间 NS 的一个成员，即使它从未在名字空间 NS 中被直接声明过。在名字空间

NS 中进行一般的查找，是找不到友元函数的。但是，因为函数 display()调用的实参类型是 Bear，所以在 Bear 的基类 ZooAnimal 中声明的友元函数将被加入到候选函数集中。

所以，如果普通函数调用的实参是类类型的对象、类类型的指针或类类型的引用，则候选函数是以下集合的并集：

1．在调用点上可见的函数；

2．在“定义该类类型的名字空间”或“定义该类的基类的名字空间”中声明的函数；

3．该类或其基类的友元函数。

对于用点或箭头成员访问操作符调用的成员函数，继承机制也会影响到它的候选函数集的建立。正如在 18.4 节所看到的，在派生类中的成员函数声明并没有重载基类中声明的同名成员函数。相反，派生类中的成员函数隐藏了基类中同名成员函数的声明，即使函数参数表并不相同。例如：

```
class ZooAnimal {
public:
     Time feeding_time( string );
     // ...
};
class Bear : public ZooAnimal {
public:
     // 隐藏 ZooAnimal::feeding_time( string )
     Time feeding_time( int );
     // ....
};

Bear Winnie;

// 错误: ZooAnimal::feeding_time( string ) 被隐藏
Winnie.feeding_time( "Winnie" );
```

在 Bear 类中声明的成员函数 feeding_time(int)隐藏了在 Bear 的基类 ZooAnimal 中声明的函数 feeding_time(string)。因为成员函数调用是通过 Bear 类型的对象 Winnie 而进行的，所以编译器只搜索 Bear 类域，以寻找成员函数调用的候选函数。在 Bear 域中惟一可见的声明是 feeding_time(int)，所以它是这个成员函数调用的候选函数集中惟一的函数，因此该调用是错误的。

为了纠正这种情况，使基类的成员函数重载派生类的成员函数，派生类的设计者可以用 using 声明把基类成员函数引入到派生类的域中。例如：

```
class Bear : public ZooAnimal {
public:
     // feeding_time( int ) is overloaded with ZooAnimal's
     using ZooAnimal::feeding_time;
     Time feeding_time( int );
     // ...
};
```

现在，两个 feeding_time()函数都在派生类 Bear 的域中，它们都是下列调用：

```
// ok: 调用 ZooAnimal::feeding_time( string )
Winnie.feeding_time( "Winnie" );
```

的候选函数的一部分，并且编译器为该调用选择了基类成员函数 feeding_time(string)。

在多继承下建立候选成员函数集时，成员函数的声明必须在同一个基类中被找到，否则该调用就是错误的。例如：

```
class Endangered {
public:
     ostream& print( ostream& );
     // ...
};

class Bear : public ZooAnimal {
public:
     void print( );
     using ZooAnimal::feeding_time;
     Time feeding_time( int );
     // ...
};

class Panda : public Bear, public Endangered {
public:
     // ...
};
int main()
{
     Panda yin_yang;
     // 错误：有二义性，该是哪一个?
     //      Bear::print()
     //      Endangered::print( ostream&)
     yin_yang.print( cout );

     // ok: 调用 Bear::feeding_time()
     yin_yang.feeding_time( 56 );
}
```

当在 Panda 的域中查找成员函数 print()的声明时，找到了 Bear::print()和 Endangered::print()。因为 print()的这两个声明不是在同一个基类中被找到的，所以即使这两个 print()函数有不同的参数表，该调用的候选函数集也是空的，而这个成员函数的调用是错误的。为了改正这个错误，Panda 类必须引入自己的 print()函数。当在 Panda 域中直找成员函数 feeding_time()的声明时，在 Bear 类域中找到了 ZooAnimal::feeding_time()和 Bear::feeding_time()。因为这两个声明是在同一个基类中被找到的，所以该调用的候选函数集包含了这两个函数，并且成员函数 Bear::feeding_time()最终被选择出来。

19.3.2 可行函数和用户定义的转换序列

继承机制也会影响到函数重载解析过程的第二步，即，从候选函数集中选择可以被调用的可行函数。可行函数是指这样的函数：从函数调用的每个实参到相应的可行函数参数之间都存在类型转换。

在 15.9 节，我们描述了一个类的设计者可以怎样为类类型的对象提供一组用户定义的转

换。这些用户定义的转换由编译器隐式地调用，以便把函数调用的实参转换成对应的可行函数参数。用户定义的转换或者是一个转换函数，或者是一个单参数的非显式构造函数。在继承机制下，在函数重载解析过程的第二步期间，将会考虑一个更大的用户定义转换的集合。

转换函数像其他的类成员函数一样会被继承。例如。我们可能为 ZooAnimal 类定义了一个转换函数，如下所示：

```
class ZooAnimal {
public:

     // 转换: ZooAnimal ==> const char*
     operator const char*();

     // ...
};
```

派生类 Bear 从基类 ZooAnimal 继承了这个转换函数。当 Bear 类型的值被用在“期望获得一个 const char*类型的操作数”的地方时，这个转换函数就被隐式地调用，并把 Bear 值转换成类型 const char*。例如：

```
extern void display( const char* );
Bear yogi;

// ok: yogi ==> const char*
display( yogi );
```

单参数的非显式构造函数定义了另外一组隐式转换。构造函数可以把其参数类型的值转换成该类类型的值。例如，我们为 ZooAnimal 定义了如下的构造函数：

```
class ZooAnimal {
public:
     // 转换: int ==> ZooAnimal
     ZooAnimal( int );

     // ...
};
```

这个构造函数可以用来把整型值转换成 ZooAnimal 类型的值，但是构造函数不被继承。当需要一个从 ZooAnimal 派生的类类型时，ZooAnimal 构造函数不能被用来转换对象。例如：

```
const int cageNumber = 8788;

void mumble( const Bear & );

// 错误: 没有使用 ZooAnimal(int)
mumble( cageNumber );
```

因为转换的目标类型是 Bear，即 mumble()参数的类型，所以只考虑 Bear 类中的构造函数。

19.3.3 最佳可行函数

继承机制也会影响到函数重载解析过程的第三步，即选择最佳可行函数。为了选择最佳可行函数，“用来将实参转换成相应函数参数类型”的类型转换被划分等级。下列隐式转换的等级是什么？

1．把派生类类型的实参转换成任何一个基类类型的参数；

2．把派生类类型的指针转换成任何一个基类类型的指针；

3．用派生类类型的左值初始化基类类型的一个引用。

在为实参上的转换划分等级时，这些转换具有标准转换的等级。（其他标准转换在 9.3 节中描述。）这些转换不是用户定义的转换。因为它们不依赖于类设计者定义的转换函数以及构造函数。例如：

```
extern void release( const ZooAnimal& );
Panda yinYang;

// 标准转换：Panda -> ZooAnimal
release( yinYang );
```

因为用 Panda 类型的实参 yinYang 初始化了一个基类类型的引用，所以该转换的等级是标准转换。

在 15.10 节中，我们知道，在对类型转换进行等级划分以便选择最佳可行函数时，标准转换序列好于用户定义的转换序列。例如：

```
class Panda : public Bear,
              public Endangered
{
     // 继承 ZooAnimal::operator const char *()
};
Panda yinYang;

extern void release( const ZooAnimal& );
extern void release( const char * );

// 标准转换：Panda -> ZooAnimal
// 选择：release( const ZooAnimal& )
release( yinYang );
```

release(const char*)和 release(const ZooAnimal&)都是可行函数。函数 release(const ZooAnimal&)是可行函数，是因为通过标准转换，可以用实参对其引用参数进行初始化。函数 release(const char*)是可行函数，是因为使用转换函数 ZooAnimal::operator const char*()的用户定义转换，可以把实参转换成 const char*类型。因为标准转换序列好于用户定义的转换序列，所以函数 release(const ZooAnimal&)被选为最佳可行函数。

对于从派生类类型到不同基类类型的不同标准转换进行等级划分时，对于从派生类类型到基类类型移动较少（距离较近）的转换，被认为好于移动较多（距离较远）的转换。例如，下面的调用不是二义的，尽管在两种情况下都要求一个标准转换。到基类 Bear 的转换被认为好于到基类 ZooAnimal 的转换，因为对于基类 Bear，它要求从派生类 Panda 移动的较少。所

以这个调用的最佳可行函数是 release(const Bear&)：

```
extern void release( const ZooAnimal& );
extern void release( const Bear& );

// ok: release( const Bear& );
release( yinYang );
```

类似的规则也适用于指针，对从派生类类型指针到不同基类类型的指针的不同标准转换进行等级划分时，从派生类类型到基类类型移动较少的转换被认为是较好的转换。类似的规则也可以扩展到 void*的处理上，到基类类型指针的标准转换，好于到 void*的转换。例如：

已知下列重载函数对：

```
void receive( void* );
void receive( ZooAnimal* );
```

函数 receive(ZooAnimal*)是 Panda*类型实参的最佳可行函数。

多继承也可能引起同样的问题：如果从派生类类型到两个基类类型的移动距离相等，则从派生类到两个不同的基类类型的标准转换等级相同。例如，Panda 是从 Bear 和 Endangered 派生而来。从派生类 Panda 到这两个基类的移动距离相同，所以从 Panda 类对象到这两个某类的转换一样好。由于两个转换一样好，所以就不能为以下的调用选择最佳可行函数，该调用是错误的：

```
extern void mumble( const Bear& );
extern void mumble( const Endangered& );

/* 错误：二义调用
 * mumble() 的选择：
 * void mumble( const Bear & );
 * void mumble( const Endangered & );
 */

mumble( yinYang );
```

为了能够解析这个调用，程序员必须给出显式强制类型转换：

```
mumble( static_cast< Bear >( yinYang ) ); // ok
```

用一个基类类型的对象初始化一个派生类对象，或初始化一个派生类类型的引用，或者从一个基类类型的指针到派生类类型的指针的转换，都不会被当作一个隐式转换来应用（然而，这样的转换可以用一个显式的 dynamic_cast 来执行，如 19.1 节所示。）对于下面的调用，不存在可行函数，因为从 ZooAnimal 类型的实参到派生类类型之间不存在隐式转换。

```
extern void release( const Bear& );
extern void release( const Panda& );
ZooAnimal za;

// 错误：没有匹配
release( za );
```

在下面的例子中，release()调用的最佳可行函数是 release(const char*)。这似乎令人有些吃惊，因为应用在实参上的、将其转换成函数参数类型的转换序列的等级是用户定义的转换

序列，它使用了 ZooAnimal 的转换函数 const char*()。然而，因为从基类类型到派生类类型没有隐式的转换，所以函数 release(const Bear&)不是可行函数，而 release(const char*)就成了该调用的惟一可行函数：

```
class ZooAnimal {
public:
     // 转换: ZooAnimal ==> const char*
     operator const char*();
     // ...
};

extern void release( const char* );
extern void release( const Bear& );
ZooAnimal za;

// za ==> const char*
// ok: release( const char* )
release( za );
```

练习 19.9

已知下面的类层次结构以及成员函数集合：

```
class Base1 {
public:
     ostream& print();
     void debug();
     void writeOn();
     void log( string );
     void reset( void *);
     // ...
};

class Base2 {
public:
     void debug();
     void readOn();
     void log( double );
     // ...
};

class MI : public Base1, public Base2 {
public:
     ostream& print();
     using Base1::reset;
     void reset( char * );
     using Base2::log;
     using Base1::log;
     // ...
};
```

对于以下的成员函数调用，候选成员函数集中有哪些函数？

```
MI *pi = new MI;

(a) pi->print();   (c) pi->readOn();    (e) pi->log( num );
(b) pi->debug();   (d) pi->reset(0);    (f) pi->writeOn();
```

练习 19.10

已知下列类层次结构以及转换函数集：

```
class Base {
public:
     operator int();
     operator const char *();
     // ...
};

class Derived : public Base {
public:
     operator double();
     // ...
};
```

对于下面的调用，哪个函数会被选为最佳可行函数（如果有的话）？请列出候选函数。可行函数，以及应用在每个可行函数的实参上的类型转换。

```
(a)  void operate( double );
     void operate( string );
     void operate( const Base & );

     Derived *pd = new Derived;
     operate( *pd );

(b)  void calc( int );
     void calc( double );
     void calc( const Derived & );

     Base *pb = new Derived;
     operate( *pb );
```

20

iostream 库

C++的输入／输出设施是由 iostream 库（iostream library）提供的，它是一个利用多继承和虚拟继承实现的面向对象类层次结构，是作为 C++标准库的一个组件而提供的。它为内置数据类型的输入输出提供了支持，同时也支持文件的输入输出。除此之外，类的设计者还可以通过扩展 iostream 库，来读写新的类类型。

为了在我们的程序中使用 iostream 库，我们必须包含相关的头文件，如下：

```
#include <iostream>
```

输入输出操作是由 istream（输入流）和 ostream（输出流）类提供的（第三个类 iostream 类同时从 istream 和 ostream 派生，允许双向输入／输出）。为了方便，这个库定义了下列三个标准流对象：

1．cin，发音为 see-in，代表**标准输入**（standard input）的 istream 类对象。一般地，cin 使我们能够从用户终端读入数据。

2．cout，发音为 see-out，代表**标准输出**（standard output）的 ostream 类对象。一般地，cout 使我们能够向用户终端写数据。

3．cerr，发音为 see-err，代表**标准错误**（standard error）的 ostream 类对象。cerr 是导出程序错误消息的地方。

输出主要由重载的左移操作符（<<）来完成。类似地，输入主要由重载的右移操作符（>>）来完成。例如：

```
#include <iostream>
#include <string>
int main() {
    string in_string;

    // 向用户终端写字符串
    cout << "Please enter your name: ";

    // 把用户输入的读取到 in_string 中
    cin >> in_string;
```

```
        if ( in_string.empty() )
            // 产生一个错误消息，输出到用户终端
            cerr << "error: input string is empty!\n";
        else cout << "hello, " << in_string << "!\n";
    }
```

怎么理解这两个操作符呢？一种很有用的思考方式是，它们指出了数据移动的方向。例如：

```
    >> X
```

把数据放入 x 中，而：

```
    << X
```

从 x 中拿出数据。20.1 节将介绍 iostream 库为数据输入提供的支持。20.5 节将了解怎样扩展 iostream 库，以支持新类类型的数据输入。类似地，20.2 节将介绍 iostream 库为数据输出提供的支持，而在 20.4 节我们将了解怎样扩展这个库，以允许新类类型的数据输出。

除了对用户终端的读写操作之外，iostream 库还支持对文件的读写。下列三种类类型提供了文件支持：

1．ifstream，从 istream 派生，把一个文件绑到程序上用来输入。

2．ofstream，从 ostream 派生，把一个文件绑到程序上用来输出。

3．fstream，从 iostream 派生，把一个文件绑到程序上用来输入和输出。

为了使用 iostream 库的文件流组件，我们必须包含相关的头文件：

```
    #include <fstream>
```

（由于在 fstream 头文件中也包含了 iostream 头文件，所以我们不需要同时包含这两个文件。）C++对于文件的输入 / 输出也支持同样的输入和输出操作符。例如：

```
    #include <fstream>
    #include <string>
    #include <vector>
    #include <algorithm>

    int main()
    {
        string ifile;
        cout << "Please enter file to sort: ";
        cin >> ifile;

        // 构造一个 ifstream 输入文件对象
        ifstream infile( ifile.c_str() );

        if( ! infile ) {
            cerr << "error: unable to open input file: "
                 << ifile << endl;
            return -1;
        }
        string ofile = ifile + ".sort";
```

```
    // 构造一个 ofstream 输出文件对象
    ofstream outfile( ofile.c_str() );
    if( !outfile ) {
        cerr << "error: unable to open output file: "
             << ofile << endl;
        return -2;
    }

    string buffer;
    vector< string, allocator > text;

    int cnt = 1;
    while ( infile >> buffer ) {
        text.push_back( buffer );
        cout << buffer << ( cnt++ % 8 ? " " : "\n" );
    }
    sort( text.begin(), text.end() );

    // ok: 把排序后的词打印到 outfile
    vector<string, allocator>::iterator iter = text.begin();
    for ( cnt = 1; iter != text.end(); ++iter, ++cnt )
        outfile << *iter
             << (cnt%8 ? " " : "\n" );
    return 0;
}
```

下面是运行该程序的一个例子：要求我们输入一个文件以便排序。我们键入 alice_emma（我们的输入在示例输出中以黑体显示）。程序把读入的每个单词回显到标准输出上：

```
Please enter file to sort: alice_emma
Alice Emma has long flowing red hair. Her
Daddy says when the wind blows through her
hair, it looks almost alive, like a fiery
bird in flight. A beautiful fiery bird, he
tells her, magical buf untamed. "Daddy, shush, there
is no such thing," she tells him, at
the same time wanting him to tell her
more. Shyly, she asks, "I mean, Daddy, is
there?"
```

接着程序再把排序后的字符串序列写到 outfile 中。当然，标点符号也会影响单词的顺序（我们将在下节修正它）：

```
"Daddy, "I A Alice Daddy Daddy, Emma Her
Shyly, a alive, almost asks, at beautiful bird
bird, blows but fiery fiery flight. flowing hair,
hair. has he her her her, him him,
in is is it like long looks magical
mean, more. no red same says she she
shush, such tell tells tells the the there
there?" thing," through time to untamed. wanting when
wind
```

在 20.6 节我们将详细介绍文件输入 / 输出。

iostream 库还支持**内存输入 / 输出**（in-memory input/output），当流被附着在程序内存中的一个字符串上时，我们可以用 iostream 输入和输出操作符来对它进行读写。可以通过定义下列三种类类型中的一个实例来定义一个 iostream 字符串对象：

1．istringstream，从 istream 派生，从一个字符串中读取数据；

2．ostringstream，从 ostream 派生，写入到一个字符串中；

3．stringstream，从 iostream 派生，从字符串中读取，或者写入到字符串中。

要使用这些类，我们必须包含相关的头文件：

```
#include <sstream>
```

（sstream 头文件包含了 iostream 头文件，因此我们无需同时包含这两个头文件。）在下面的代码段中，ostringstream 被用来格式化一个错误信息，然后再返回底层的字符串：

```
#include <sstream>

string program_name( "our_program" );
string version( "0.01" );

// ...

string mumble( int *array, int size )
{
    if ( ! array ) {
        ostringstream out_message;

        out_message << "error: "
            << program_name << "--" << version
            << ": " << __FILE__ << ": " << __LINE__
            <<" -- ptr is set to 0; "
            << " must address some array.\n";

        // 返回底层 string 对象
        return out_message.str();
    }
    // ...
}
```

20.8 节将详细介绍 iostream 字符串对象。

在实践中，iostream 支持两种预定义的字符类型：char 和 wchar_t。目前我们所描述的 iostream 类（以及我们在本章余下部分要关注的）读写的是 char 型的流。与此互补的是另外一组支持 wchar_t 型的 iostream 对象和类。每个类与类对象都加了前缀“w”，以便与相应的 char 型区分开。因此，wchar_t 标准输入被命名为 wcin，标准输出为 wcout，以及标准错误 wcerr。然而，char 和 wchar_t 型的 stream 类和类对象所需要的头文件是相同的。

wchar_t 输入和输出类是 wistream、wostream 和 wiostream。文件输入和输出类是 wifstream、wofstream 和 wfstream。iostream 字符串输入输出类是 wistringstream、wostringstream 以及 wstringstream。

20.1 输出操作符<<

最常用的输出方法是在 cout 上应用左移操作符（<<）。例如：

```
#include <iostream>

int main() {
     cout << "gossipaceous Anna Livia\n";
}
```

在用户终端上输出以下内容：

```
gossipaceous Anna Livia
```

输出操作符可以接受任何内置数据类型的实参，包括 const char*，以及标准库 string 和 complex 类类型。任何表达式包括函数调用，都可以是输出操作符的实参，只要它的计算结果是一个能被输出操作符实例接受的数据类型即可。例如：

```
#include <iostream>
#include <string.h>

int main()
{
     cout << "The length of "ulysses" is:\t";
     cout << strlen( "ulysses" );
     cout << '\n';

     cout << "The size of "ulysses" is:\t";
     cout << sizeof( "ulysses" );
     cout << endl;
}
```

在用户终端上输出如下内容：

```
The length of "ulysses" is:  7
The size of "ulysses" is:    8
```

［endl 是一个 ostream 操纵符（manipulator），它把一个换行符插入到输出流中，然后再刷新 ostream 缓冲区。我们将在 20.9 节介绍有关缓冲的做法］。

把输出操作符连接成一条语句常常会更方便一些。例如，上面的程序可以重写为：

```
#include <iostream>
#include <string.h>

int main()
{
     // 输出操作符可以被连接在一起
     cout << "The length of "ulysses" is:\t"
          << strlen( "ulysses" ) << 'n';
     cout << "The size of "ulysses" is:\t"
          << sizeof( "ulysses" ) << endl;
}
```

输出操作符序列（以及输入操作符序列）能够被连接是原因表达式：

```
cout << "some string"
```

计算的结果是左边的 ostream 操作数。也就是说，表达式的结果是 cout 对象自己，于是，通过这个序列，它又被应用到下一个输出操作符上，等等（我们说操作符<<从左向右结合）。

iostream 库还提供了指针类型的预定义输出操作符，允许显示对象的地址。缺省情况下，这些值以十六进制的形式显示。例如：

```
#include <iostream>

int main()
{
    int i = 1024;
    int *pi = &i;

    cout << "i: " << i
         << "\t&i:\t" << &i << '\n';

    cout << "*pi: " << *pi
         << "\tpi:\t" << pi << endl
         << "\t\t&pi:\t" << &pi << endl;
}
```

在终端上输出：

```
i:   1024   &i: 0x7ffff0b4
*pi: 1024   pi: 0x7ffff0b4
           &pi: 0x7ffff0b0
```

后面我们将会看到怎样用十进制数形式打印地址。

下面的程序展示了一种令人迷惑的现象，我们的目的是输出 pstr 所包含的地址值：

```
#include <iostream>

const char *str = "vermeer";

int main()
{
    const char *pstr = str;
    cout << "The address of pstr is: "
         << pstr << endl;
}
```

但是，在编译并运行该程序后，却产生了以下意料之外的输出：

```
The address of pstr is: vermeer
```

问题是，类型 const char*没有被解释成地址值，而是解释为 C 风格字符串。为了输出 pstr 包含的地址值。我们必须改变 const char*的缺省处理。我们将分两步完成此处理，首先把 const 强制转换掉，然后再把 pstr 强制转换成 void*类型：

```
<< static_cast<void*>(const_cast<char*>(pstr))
```

编译并运行程序，得到了我们所期望的输出：

```
The address of pstr is: 0x116e8
```

下面是另一个令人迷惑的现象，我们的目的是显示两个值中的较大值：

```
#include <iostream>

inline void
max_out( int val1, int val2 ) {
     cout << ( val1 > val2 ) ? val1 : val2;
}

int main()
{
     int ix = 10, jx = 20;

     cout << "The larger of " << ix;
     cout << ", " << jx << " is ";

     max_out( ix, jx );

     cout << endl;
}
```

但是，在编译并运行程序后，却生成了如下不正确的结果：

```
The larger of 10, 20 is 0
```

问题在于输出操作符的优先级高于条件操作符，所以输出 val1 和 val2 比较结果的 true/false 值。即，表达式：

```
cout << ( val1 > val2 ) ? val1 : val2;
```

被计算为：

```
(cout << ( val1 > val2 )) ? val1 : val2;
```

因为 val1 不大于 val2，所以计算结果为 false，它被输出为 0。为了改变预定义的操作符优先顺序，整个条件操作符表达式必须被放在括号中：

```
cout << (val1 > val2 ? val1 : val2);
```

这次产生了正确的输出：

```
The larger of 10, 20 is 20
```

如果 bool 文字值 true 和 false 以字符串的形式输出，而不是 0 或 1——即，如果输出为：

```
The larger of 10, 20 is false
```

则前面不正确的输出可能会更容易调试，从而不至于让程序员产生迷惑。

缺省情况下，false 文字值被输出为 0，而 true 为 1。我们可以通过应用 boolalpha 操纵符来改变这种缺省行为。下面的程序正是这样做的：

```
int main()
{
     cout << "default bool values: "
          << true << " " << false
          << "\nalpha bool values: "
          << boolalpha
          << true << " " << false
```

```
            << endl;
    }
```

程序执行时产生下面的输出：

```
default bool values: 1 0
alpha bool values: true false
```

对于内置数组及容器类型（如 vector 或 map）的输出，要求迭代一遍，并输出每个单独的元素。例如：

```
#include <iostream>
#include <vector>
#include <string>

string pooh_pals[] = {
    "Tigger", "Piglet", "Eeyore", "Rabbit"
};

int main()
{
    vector<string> ppals( pooh_pals, pooh_pals+4 );
    vector<string>::iterator iter = ppals.begin();
    vector<tring>::iterator iter_end = ppals.end();

    cout << "These are Pooh's pals: ";

    for ( ; iter != iter_end; iter++ )
        cout << *iter << " ";
    cout << endl;
}
```

我们可以不用显式地“对容器中的元素进行迭代，并依次输出每个元素”，ostream_iterator 可以用来实现同样的效果。例如，下面是一个等价的程序，它使用了 ostream_iterator（关于 ostream_iterator 的详细讨论见 12.4 节）：

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>

string pooh_pals[] = {
    "Tigger", "Piglet", "Eeyore", "Rabbit"
};

int main()
{
    vector<string> ppals( pooh_pals, pooh_pals+4 );
    vector<string>::iterator iter = ppals.begin();
    vector<string>::iterator iter_end = ppals.end();

    cout << "These are Pooh's pals: ";

    // 把每个元素拷贝到 cout ...
```

```
    ostream_iterator< string > output( cout, " " );
    copy( iter, iter_end, output );

    cout << endl;
}
```

编译并运行程序，产生如下输出：

```
These are Pooh's pals: Tigger Piglet Eeyore Rabbit
```

练习 20.1

已知下列对象定义：

```
string sa[4] = { "pooh", "tigger", "piglet", "eeyore" };
vector< string > svec( sa, sa+4 );
string robin( "christopher robin" );
const char *pc = robin.c_str();

int ival = 1024;
char blank = ' ';

double dval = 3.14159;
complex purei( 0, 7 );
```

(a) 在标准输出上打印出每个对象的值：
(b) 输出 pc 的地址值：
(c) 利用以下条件操作符的结果，输出 ival 和 dval 的最小值。

```
ival < dval ? ival : dval
```

20.2 输入

输入主要由右移操作符（>>）来支持。例如，在下面的程序中，从标准输入读入一个 int 型的值序列，并把它放在一个 vector 中：

```
#include <iostream>
#include <vector>
int main()
{
    vector<int> ivec;
    int ival;
    while ( cin >> ival )
        ivec.push_back( ival );
    // ...
}
```

子表达式：

```
cin >> ival
```

从标准输入读入一个整数值，如果成功，则把该值拷贝到 ival 中。这个子表达式的结果是左边的 istream 对象——在这种情况下，即 cin 自己。（我们马上就会看到，这使得输入操

作符能够连接起来。）

表达式：

```
while ( cin >> ival )
```

从标准输入读入一个序列，直到 cin 为 false 为止。有两种情况会使一个 istream 对象被计算为 false：读到文件结束（在这种情况下，我们已经正确地读完文件中所有的值）或遇到一个无效的值，比如 3.14159（小数点是非法的）、1e-1（字符文字 e 是非法的），或者一般的任意字符串文字。在读入一个无效值的情况下，istream 对象被放置到一种错误的状态中，并且对于值的所有读入动作都将停止（在 20.7 节，我们将详细地讨论该错误条件）。

预定义的输入操作符可以接受任何的内置数据类型，包括 C 风格字符由以及标准库 string 和 complex 类类型。例如：

```
#include <iostream>
#include <string>

int main()
{
    int item_number;
    string item_name;
    double item_price;

    cout << "Please enter the item_number, item_name, and price: "
         << endl;

    cin >> item_number;
    cin >> item_name;
    cin >> item_price;

    cout << "The values entered are: item# "
         << item_number << " "
         << item_name << " @$"
         << item_price << endl;
}
```

下面是程序的执行示例情况：

```
Please enter the item_number, item_name, and price:
10247 widget 19.99
The values entered are: item# 10247 widget @$19.99
```

如果在另一行上输入每个项会怎样呢？这不是问题。缺省情况下，输入操作符丢弃任何中间空白（空格、制表符、换行符、走纸以及回车。关于如何改变这种缺省行为的讨论见 20.9 节）：

```
Please enter the item_number, item_name, and price:
10247
widget
19.99
The values entered are: item# 10247 widget @$19.99
```

“数值的读操作”比“数值的写操作”更可能导致 iostream 错误。例如，如果输入项的序列是：

```
// 错误：item_name 应该在第二个位置上
BuzzLightyear 10009 8.99
```

语句：

```
cin >> item_number;
```

将导致输入错误，这是因为 BuzzLightyear 不是一个 int 类型的值。当出现输入错误时，如果对 istream 对象进行测试，则测试结果为 false。另一种更为健壮的做法如下所示：

```
cin >> item_number;
if ( ! cin )
    cerr << "error: invalid item_number type entered!\n";
```

尽管 iostream 库支持输入操作符的连接，但是这种做法使我们无法测试个别读操作的可能错误。因此，我们只能在确实没有错误机会的情况下才能使用连接形式的输入操作符。下面是改写之后的程序，用到了连接形式的输入操作符：

```
#include <iostream>
#include <string>

int main()
{
    int item_number;
    string item_name;
    double item_price;

    cout << "Please enter the item_number, item_name, and price: "
         << endl;

    // ok: 但更容易出错
    cin >> item_number >> item_name >> item_price;

    cout << "The values entered are: item# "
         << item_number << " "
         << item_name << " @$"
         << item_price << endl;
}
```

字符序列：

```
ab c
d e
```

由下列九个字符构成：‘a’、‘b’、‘ ’（空格）、‘c’、‘\n’（换行符）、‘d’、‘\t’（制表符）、‘c’和‘\n’。然而，下面的程序用输入操作符只读取了五个字母字符：

```
#include <iostream>

int main()
{
    char ch;

    // 读入每个字符，然后输出
    while ( cin >> ch )
        cout << ch;
    cout << endl;
```

```
    // ...
}
```

程序执行时输出如下内容：

```
abcde
```

缺省情况下，所有的空白字符都被抛弃掉。如果我们希望读入空白字符，或许是为了保留原始的输入格式，或许是为了处理空白字符（比如计算换行符的个数），一种方法是使用 istream 的 get()成员函数（ostream 的 put()成员函数一般与 get()配合使用——稍后我们将更详细地看看这些函数）。例如：

```
#include <iostream>

int main()
{
    char ch;

    // 获取每个字符，包括空白字符
    while ( cin.get( ch ))
        cout.put( ch );

    // ...
}
```

（第二种方法是使用 noskipws 操纵符。）

对于下面的两个字符串序列，如果由 const char*或 string 输入操作符来读入，则它们都被视为“包含五个由空白字符分隔的字符串”：

```
A fine and private place
"A fine and private place"
```

引号的存在并没有导致内联的空白字符被当作扩展字符串的一部分。相反，这两个引号成为第一个词的首字符，以及最后一个词的末字符。

我们可以不是显式地从标准输入上依次读入每个单独的元素，istream_iterator 可以被用来达到相同的行为效果。例如：

```
#include <algorithm>
#include <string>
#include <vector>
#include <iostream>

int main()
{
    istream_iterator< string > in( cin ), eos ;
    vector< string > text ;

    // 从标准输入向 text 拷贝值
    copy( in , eos , back_inserter( text ) ) ;
    sort( text.begin() , text.end() ) ;

    // 删除所有重复的值
    vector< string >::iterator it ;
```

```
        it = unique( text.begin() , text.end() ) ;
        text.erase( it , text.end() ) ;

        // 显示结果 vector
        int line_cnt = 1 ;
        for ( vector< string >::iterator iter = text.begin();
                iter != text.end() ; ++iter , ++line_cnt )
            cout << *iter
                << ( line_cnt % 9 ? " " : "\n" ) ;
        cout << endl;
}
```

程序的输入是程序代码文本本身。这些代码文本已经被存储在名为 istream_iter.C 的文件中。在 UNIX 下，我们可以把文件重定向到标准输入（istream_iter 是程序的名字）：

```
istream_iter < istream_iter.C
```

（对于非 UNIX 系统，请查询程序员指南手册。）当程序执行时将产生以下输出：

```
!= " "\n" #include % ( ) *iter ++iter
++line_cnt , 1 9 : ; << <algorithm> <iostream.h>
<string> <vector> = > >::difference_type >::iterator ? allocator
back_inserter(
cin copy( cout diff_type eos for in in( int
istream_iterator< it iter line_cnt main() sort( string text
text.begin()
text.end() text.erase( typedef unique( vector< { }
```

（关于 iostream_iterator 在 12.4 节讨论。）

除了预定义的输入操作符以外，重载的输入操作符可以支持读入用户定义的类类型。20.5 节将详细介绍重载的输入操作符。

20.2.1 字符串输入

我们既可以以 C 风格字符数组的形式读入字符串，也可以以 string 类类型的形式读入字符串。会建议使用 string 类类型，使用它的主要好处是，与字符串相关的内存可被自动管理。例如，为了把字符串当作 C 风格字符数组，我们必须判断数组的长度——该长度应该足够容纳每一个可能的字符串。典型情况下，我们将每个字符串读入到一个数组缓冲区中。然后从空闲存储区中分配“正好可以存放这个字符串”的内存，并把缓冲区拷贝到这个按需分配的内存区中。例如：

```
#include <iostream>
#include <string.h>
char inBuf[ 1024 ];
try
{
    while ( cin >> inBuf ) {
        char *str = new char[ strlen( inBuf )+1 ];
        strcpy( str, inBuf );
        // ... 操作 str
        delete [] str;
    }
}
```

```
catch( ... ) { delete [] str; throw; }
```

string 类型非常易于管理：

```
#include <iostream>
#include <string>

string str;
while ( cin >> str )
    // ... 操作 string
```

在本小节的余下部分，我们将了解如何用 C 风格字符数组和 string 类输入操作符来读入字符串。我们仍用“young Alice Emma”作为输入文本：

```
Alice Emma has long flowing red hair. Her Daddy says
when the wind blows through her hair, it looks almost
alive, like a fiery bird in flight. A beautiful fiery
bird, he tells her, magical but untamed. "Daddy, shush,
there is no such creature," she tells him, at the same time
wanting him to tell her more. Shyly, she asks, "I mean,
Daddy, is there?"
```

我们将把它键入到名为 alice_emma 的文本文件中，然后再将其重定向到程序的标准输入。以后介绍文件输入时，我们将直接打开并读取它。以下程序将从标准输入中以 C 风格字符数组形式读取字符串序列。并确定哪一个字符串最长。

```
#include <iostream>
#include <string.h>

int main()
{
    const int bufSize = 24;
    char buf[ bufSize ], largest[ bufSize ];

    // 存放统计数;
    int curLen, max = -1, cnt = 0;
    while ( cin >> buf )
    {
        curLen = strlen( buf );
        ++cnt;

        // new longest word? save it.
        if ( curLen > max ) {
            max = curLen;
            strcpy( largest, buf );
        }
    }

    cout << "The number of words read is "
         << cnt << endl;
    cout << "The longest word has a length of "
         << max << endl;
    cout << "The longest word is "
         << largest << endl;
```

```
}
```

编译并执行程序，产生以下输出：

```
The number of words read is 65
The longest word has a length of 10
The longest word is creature,"
```

实际上，这个结果是不正确的。beautiful 是文本中最长的词，长度为 9。但是，被选中的是 creature，因为有一个逗句和一个引号附在它的后面。为了使程序能像用户期望的那样解释字符串，我们需要过滤掉非字母元素。

但是，在开始之前，我们应该再仔细地看一看这个程序。在程序中，每个字符串被存储在 buf 中，它被声明成长度为 24 的数组。如果读入的字符串长度等于或超出 24，buf 就会溢出。例如，上个例子可能会被修改如下：

```
while ( cin >> setw( bufSize ) >> buf )
```

这里 bufSize 是字符数组 buf 的长度。setw()把长度等于或大于 bufSize 的字符串分成最大长度为：

```
bufSize - 1
```

的两个或多个字符串。

在每个新串的末尾放一个空字符。为了使用 setw()，要求程序包含 iomanip 头文件：

```
#include <iomanip>
```

如果可见的 buf 声明没有指定维数：

```
char buf[] = "An unrealistic example";
```

则程序员可以应用 sizeof 操作符——只要标识符是一个数组的名字，并且在表达式可见的域中：

```
while ( cin >> setw(sizeof( buf )) >> buf );
```

在下面的程序中，使用 sizeof 操作符将导致我们没有预料到的程序行为的发生：

```
#include <iostream>
#include <iomanip>

main()
{
    const int bufSize = 24;
    char buf[ bufSize ];

    char *pbuf = buf;

    // 每个大于 sizeof(char*) 的字符串
    // 被分成两个或多个字符串
    while ( cin >> setw(sizeof(pbuf)) >> pbuf )
        cout << pbuf << endl;
}
```

编译并执行该程序，产生下列未预料的结果：

```
$ a.out
```

```
The winter of our discontent

The
win
ter
of
our
dis
con
ten
t
```

出现的这个问题在于，传递给 setw()的是字符指针的大小而不是其指向的字符数组的大小。在这台特定的机器上，字符指针是四字节，所以原始输入被分成长度为 3 的字符串序列。

下面的代码试图修正这个错误，但实际上将导致更严重的错误：

```
while ( cin >> setw(sizeof(*pbuf)) >> pbuf )
```

意图是向 setw()传递 pbuf 指向的数组的大小。但是，符号：

```
*pbuf
```

只产生一个 char。在这种情况下。被传递给 setw()的是 1。while 循环每次执行都会把一个空字符读入到 pbuf 指向的数组中。所以从来不会读取标准输入，且该循环会无限进行下去。

如果使用 string 类类型，则所有这些内存分配的问题就都不存在了，string 会自动管理内存。下面是用 string 重写的程序：

```
#include <iostream>
#include <string>

int main()
{
    string buf, largest;

    // 存放统计数：
    int curLen, max = -1, cnt = 0;
    while ( cin >> buf ) {
        curLen = buf.size();
        ++cnt;

        // 又出现最长单词了？保存
        if ( curLen > max ) {
            max = curLen;
            largest = buf;
        }
    }
    // ... 其余同上
}
```

由于逗号和引号被解释成字符串的一部分，所以输出仍然不正确。让我们来写一个函数来从字符串中滤掉这些元素：

```
#include <string>
void filter_string( string &str )
{
    // 过滤元素
    string filt_elems( "\",?.");
    string::size_type pos = 0;
    while (( pos = str.find_first_of( filt_elems, pos ))
            != string::npos )
        str.erase( pos, 1 );
}
```

这样做能工作得很好，但是，我们希望去掉的元素被固定在代码中了。较好的策略是，允许用户传递一个包含这些元素的字符串。如果用户希望使用这些缺省的元素，则他们可以传递一个空字符串：

```
#include <string>
void filter_string( string &str,
    string filt_elems = string("\",?."))
{
    string::size_type pos = 0;
    while (( pos = str.find_first_of( filt_elems, pos ))
            != string::npos )
        str.erase( pos, 1 );
}
```

以下是 filter_string()的更一般化的版本，它接受一对 iterator，由它们标记出需要过滤的元素范围：

```
template <class InputIterator>
void filter_string( InputIterator first, InputIterator last,
string filt_elems = string("\",?."))
{
    for ( ; first != last; first++ )
    {
        string::size_type pos = 0;
        while (( pos = (*first).find_first_of( filt_elems, pos ))
                != string::npos )
        (*first).erase( pos, 1 );
    }
}
```

使用该函数的程序可能会这样实现：

```
#include <string>
#include <algorithm>
#include <iterator>
#include <vector>
#include <iostream>

bool length_less( string s1, string s2 )
{ return s1.size() < s2.size(); }

int main()
{
```

```
    istream_iterator< string > input( cin ), eos;
    vector< string > text;

    // copy 是一个泛型算法
    copy( input, eos, back_inserter( text ));

    string filt_elems( "\",.?;:");
    filter_string( text.begin(), text.end(), filt_elems );
    int cnt = text.size();

    // max_element 是一个泛型算法
    string *max = max_element( text.begin(), text.end(),
                  length_less );
    int len = max->size();

    cout << "The number of words read is "
         << cnt << endl;

    cout << "The longest word has a length of "
         << len << endl;

    cout << "The longest word is "
         << *max << endl;
}
```

当我们在 max_element()中使用缺省的 string 小于操作符时，程序的输出令我们吃惊：

```
The number of words read is 65
The longest word has a length of 4
The longest word is wind
```

wind 显然不是最长的元素。对这个结果迷惑了一会儿之后，我们意识到，string 的小于操作符计算的不是字符串的长度而是其字母顺序关系。在那种意义上，wind 是文本中的最大字符串。为了找到最大长度的字符串，我们需要提供另外一个小于操作符：length_less()：

```
The number of words read is 65
The longest word has a length of 9
The longest word is beautiful
```

练习 20.2

从标准输入读入类型为：string、double、string、int 以及 string 的一个序列。检查是否有输入错误发生。

练习 20.3

从标准输入读入未知数目的字符串，并把它们存储在 list 中。然后，判断最长和最短的字符串。

20.3 其他输入／输出操作符

在某些场合下，我们需要把输入流当作一个未经解释的字节序列来读取，而不是特定数据类型（如 char、int、string 等等）的序列。istream 成员函数 get()一次读入一个字节，getline()一次读入一块字节，或者由一个换行符作为结束，或者由某个用户定义的终止字符作为结束。成员函数 get()有三种形式：

1．get(char& ch)从输入流中提取一个字符，包括空白字符，并将它存储在 ch 中。它返回被应用的 istream 对象。例如，以下程序收集了在输入流上的各种统计信息，然后直接将其拷贝到输出流上：

```
#include <iostream>
int main()
{
    char ch;
    int tab_cnt = 0, nl_cnt = 0, space_cnt = 0,
        period_cnt = 0, comma_cnt = 0;
    while ( cin.get( ch )) {
        switch( ch ) {
            case ' ': space_cnt++;   break;
            case '\t': tab_cnt++; break;
            case '\n': nl_cnt++;  break;
            case '.': period_cnt++;  break;
            case ',': comma_cnt++;   break;
        }
        cout.put( ch );
    }
    cout << "\nour statistics:\n\t"
        << "spaces: "<< space_cnt << '\t'
        << "new lines: "   << nl_cnt << '\t'
        << "tabs: "  << tab_cnt   << "\n\t"
        << "periods: "  << period_cnt << '\t'
        << "commas: "<< comma_cnt << endl;
}
```

ostream 成员函数 put()提供了另外一种方法，用来将字符输出到输出流中。put()接受 char 型的实参并返回被调用的 ostream 类对象。

编译并执行程序，产生下列输出：

```
Alice Emma has long flowing red hair. Her Daddy says
when the wind blows through her hair, it looks almost alive,
like a fiery bird in flight. A beautiful fiery bird, he tells her,
magical but untamed. "Daddy, shush, there is no such creature,"
she tells him, at the same time wanting him to tell her more.
Shyly, she asks, "I mean, Daddy, is there?"

our statistics:
    spaces: 59    new lines: 6  tabs: 0
    periods: 4    commas: 12
```

2．get()的第二个版本也从输入流读入一个字符。区别是，它返回该字符值而不是被应用的 istream 对象。它的返回类型是 int 而不是 char，因为它也返回文件尾的标志（end-of-file），该标志通常用-1 来表示，以便与字符集区分开。为测试返回值是否为文件尾，我们将它与 iostream 头文件中定义的常量 EOF 做比较。被指定用来存放 get()返回值的变量，应该被声明为 int 类型，以便包含字符值和 EOF。下面是一个简单的例子：

```
#include <iostream>

int main()
{
     int ch;

     // 或使用:
     // while (( ch = cin.get() ) && ch != EOF )
     while (( ch = cin.get()) != EOF )
          cout.put(ch);

     return 0;
}
```

使用前两个 get()中任何一个，读取下列字符序列需要七次迭代：

```
a b c
d
```

读入的七个字符是（‘a’、空格、‘b’、空格、‘c’、换行和‘d’）。第八次迭代遇到 EOF。对于输入操作符（>>），因为它在缺省情况下跳过空白字符，所以它读取这个序列只需四次迭代，依次返回‘a’、‘b’、‘c’和‘d’。而 get()函数的下一种形式可以用两次迭代读取这个序列。

3．get()的第三个版本具有下列原型：

```
get(char *sink, streamsize size, char delimiter='\n')
```

sink 代表一个字符数组，用来存放被读取到的字符。size 代表可以从 istream 中读入的字符的最大数目。delimiter 表示，如果遇到它就结束读取字符的动作。delimiter 字符本身不会被读入，而是留在 istream 中，作为 istream 的下一个字符。一种常见的错误是，在执行第二个 get()之前忘了去掉 delimiter。在下面的程序例子中，我们用 istream 成员函数 ignore()来去掉 delimiter。缺省情况下，换行符被用作 delimiter。

字符读取过程一直进行，直到以下任何一个条件发生。在发生了任何一个条件之后，一个空字符被放在数组中的下一个位置上。

- size-1 个字符被读入；
- 遇到文件结束符（end-of-file）；
- 遇到 delimiter 字符（再次说明，它不会被放在数组中，而是留作 istream 的下一个字符）。

get()的返回值是被调用的 istream 对象。（gcount()返回实际被读入的字符个数。）下面是其用法的一个简单示例：

```
#include <iostream>
```

```
int main()
{
    const int max_line = 1024;
    char line[ max_line ];

    while ( cin.get( line, max_line ))
    {
        // 最大读取数量 max_line - 1, 也可以为 null
        int get_count = cin.gcount();
        cout << "characters actually read: "
             << get_count << endl;

        // 处理每一行
        // 如果遇到换行符
        // 在读下一行之前去掉它
        if ( get_count & max_line-1 )
            cin.ignore();
    }
}
```

针对 young Alice Emma 执行程序时，将产生以下输出：

```
characters actually read: 52
characters actually read: 60
characters actually read: 66
characters actually read: 63
characters actually read: 61
characters actually read: 43
```

为了更好地测试它的行为，我们创建了一行，超过了 max_line 个字符，然后再把它放在包含 Alice Emma 的文件的最前面：

```
characters actually read: 1023
characters actually read: 528
characters actually read: 52
characters actually read: 60
characters actually read: 66
characters actually read: 63
characters actually read: 61
characters actually read: 43
```

缺省情况下，ignore()从被调用的 istream 对象中读入一个字符并丢弃掉，但是我们也可以指定显式的长度和 delimiter。它的原型如下：

```
ignore( streamsize length = 1, int delim = traits::eof )
```

ignore()从 istream 中读入并丢弃 length 个字符，或者遇到 delimiter 之前包含 delimiter 在内的所有字符，或者直到文件结尾。它返回当前被应用的 istream 对象。

因为程序员常常忘了在应用 get()之前丢弃 delimiter，所以使用成员函数 getline()要比 get()更好，因为它丢弃 delimiter 而不是将其留作 istream 的下一个字符。getline()的语法与 get()的三参数形式相同（它也返回被调用的 istream 对象）：

```
getline(char *sink, streamsize size, char delimiter='\n')
```

因为 getline()和 get()的三参数形式都可以读入 size 个或少于 size 个字符，所以我们有必要查询 istream，以确定实际读入了多少个字符。istream 成员函数 gcount()正好提供了这样的信息，它返回由最后的 get()或 getline()调用实际提取的字符数。

ostream 成员函数 write()提供了另外一种方法，可以输出字符数组。它不是输出“直到终止空字符为止的所有字符”，而是输出某个长度的字符序列，包括内含的空字符。它的函数原型如下：

```
write( const char *sink, streamsize length )
```

length 指定要显示的字符个数。write()返回当前被调用的 ostream 类对象。

与 ostream 的 write()函数相对应的函数是 istream 的 read()函数，它的原型被定义如下：

```
read( char* addr, streamsize size )
```

read()从输入流中提取 size 个连续的字节，并将其放在地址从 addr 开始的内存中。gcount()返回由最后一个 read()调用提取的字节数，而 read()返回当前被调用的 istream 类对象。下面是使用 getline()、gcount()和 write()的例子：

```
#include <iostream>

int main()
{
     const lineSize = 1024;
     int lcnt = 0;      // 读入多少行
     int max = -1;      // 最长行的长度
     char inBuf[ lineSize ];

     // 读取 1024 个字符或者遇到换行符
     while (cin.getline( inBuf, lineSize ))
     {
          // 实际读入多少字符
          int readin = cin.gcount();

          // 统计：行数、最长行
          ++lcnt;
          if ( readin > max )
               max = readin;

          cout << "Line #" << lcnt
               << "\tChars read: " << readin << endl;
          cout.write( inBuf, readin).put('\n').put('\n');
     }

     cout << "Total lines read: " << lcnt << endl;
     cout << "Longest line read: " << max << endl;
}
```

当对 Moby Dick 的前几个句子执行这个程序时，程序产生下列输出：

```
Line #1 Chars read: 45
Call me Ishmael. Some years ago, never mind

Line #2 Chars read: 46
```

```
how long precisely, having little or no money

Line #3 Chars read: 48
in my purse, and nothing particular to interest

Line #4 Chars read: 51
me on shore, I thought I would sail about a little

Line #5 Chars read: 47
and see the watery part of the world. It is a

Line #6 Chars read: 43
way I have of driving off the spleen, and

Line #7 Chars read: 28
regulating the circulation.

Total lines read: 7
Longest line read: 51
```

istream 的 getline()函数只支持输入到一个字符数组中。但是，标准库给出了非成员的 getline()实例，它可以输入到一个 string 对象中，原型如下：

```
getline( istream &is, string str, char delimiter );
```

这个 getline()实例的行为如下：读入最大数目为 str::max_size-1 个字符。如果输入序列超出这个限制，则读操作失败，并且 istream 对象被设置为错误状态；否则，当读到 delimiter（它被从 istream 中丢弃，但没有被插入到 string 中）或遇到文件结束符时，输入结束。

另外还有三个 istream 操作符：

```
// 将字符放回 iostream
putback( char c );
// 往回重置“下一个” istream 项
unget();
// 返回下一个字符（或 EOF）
// 但不要提取出来
peek();
```

下面代码段说明了怎样使用这些操作符：

```
char ch, next, lookahead;
while ( cin.get( ch ))
{
     switch (ch) {
     case '/':
          // 是注释行吗? 用 peek() 看一看:
          // 是的? ignore() 余下的行
          next = cin.peek();
          if ( next == '/' )
               cin.ignore( lineSize, '\n' );
          break;
     case '>':
          // 查找 >>=
```

```
            next = cin.peek();
            if ( next == '>' ) {
                lookahead = cin.get();
                next = cin.peek();
                if ( next != '=' )
                    cin.putback( lookahead );
            }
        // ...
    }
```

练习 22.4

请从标准输入读入以下字符序列，包括所有空白字符，并依次回显在标准输出上：

```
a b c
d e
f
```

练习 20.5

请读入句子“riverrun，from bend of bay to swerve of shore”，将它当作 (a) 九个字符串的序列；(b)一个单个字符串。

练习 20.6

请用 getline()和 gcount()从标准输入读入一系列文字行。确定读入的最长行（如果在读入一行时，要求应用多个 getline()，则确保这一行仍被看作单独的一行）。

20.4 重载输出操作符<<

当实现一个类类型时，如果我们希望这个类支持输入和输出操作，那么必须提供重载的输入和输出操作符的实例。在本节，我们将了解怎样重载输出操作符，重载输入操作符是下一节的主题。下面是 WordCount 类的输出操作符重载实例：

```
#include <iostream>

class WordCount {
    friend ostream&
    operator<<(ostream&, const WordCount&);
public:
    WordCount( string word, int cnt=1 );
    // ...
private:
    string word;
    int occurs;
};
ostream&
operator <<( ostream& os, const WordCount& wd )
{ // 格式: <occurs> word
    os << "< " << wd.occurs << " > "
```

```
            << wd.word;
        return os;
    }
```

这里有一个问题，类的输出操作符是否应该产生尾部换行符？由于内置数据类型的输出操作符并不产生这样的换行符，所以用户一般不会期望一个类的实例会提供换行符。因此，对于一个类的输出操作符而言，较好的设计选择是不产生尾部的换行符。

一旦定义了 WordCount 的输出操作符，我们就可以将它与其他输出操作符自由地混合使用了。例如：

```
#include <iostream>
#include "WordCount.h"

int main()
{
    WordCount wd( "sadness", 12 );
    cout << "wd:\n" << wd << endl;

    return 0;
}
```

在用户终端上输出如下：

```
wd:
< 12 > sadness
```

输出操作符是一个双目操作符，它返回一个 ostream 引用。重载定义的通用框架如下：

```
// 重载 output 操作符的通用框架
ostream&
operator <<( ostream& os, const ClassType &object )
{
    // 准备对象的特定逻辑
    // 成员的实际输出
    os << // ...

    // 返回 ostream 对象
    return os;
}
```

它的第一个实参是一个 ostream 对象的引用。第二个一般是一个特定类类型的 const 引用，返同类型是一个 ostream 引用，且它的值总是该输出操作符所应用的 ostream 对象。

因为第一个实参是一个 ostream 引用，所以输出操作符必须定义为非成员函数。（详细讨论见 15.1 节。）当输出操作符要求访问非公有成员时，必须将它声明为该类的友元。（关于友元的讨论见 15.2 节。）

Location 是一个类，它包含了一个单词每次出现所在的行列数。下面是它的定义：

```
#include <iostream>

class Location {
friend ostream& operator<<( ostream&, const Location& );
public:
    Location( int line=0, int col=0 )
        : _line( line ), _col( col ) {}
```

```
private:
    short _line;
    short _col;
};
ostream& operator <<( ostream& os, const Location& lc )
{
    // output of a Location object: < 10,37 >
    os << "<" << lc._line
        << "," << lc._col << "> ";
    return os;
}
```

让我们来重新定义 WordCount，使它包含一个 Location 类对象的 vector：_occurList，以及一个 string 类对象_word：

```
#include <vector>
#include <string>
#include <iostream>
#include "Location.h"

class WordCount {
    friend ostream& operator<<(ostream&, const WordCount&);

public:
    WordCount(){}
    WordCount( const string &word ) : _word(word){}
    WordCount( const string &word, int ln, int col )
            : _word( word ){ insert_location( ln, col ); }

    string word() const { return _word; }
    int occurs() const { return _occurList.size(); }
    void found( int ln, int col )
        { insert_location( ln, col ); }
private:
    void insert_location( int ln, int col )
    { _occurList.push_back( Location( ln, col )); }
    string _word;
    vector< Location > _occurList;
};
```

string 类和 Location 类都定义了 operator<<()的实例。下面是 WordCount 输出操作符的新定义：

```
ostream&
operator <<( ostream& os, const WordCount& wd )
{
    os << "<" << wd._occurList.size() << "> "
        << wd._word << endl;

    int cnt = 0, onLine = 6;

    vector< Location >::const_iterator first =
        wd._occurList.begin();
```

```
        vector< Location >::const_iterator last =
            wd._occurList.end();

        for ( ; first != last, ++first )
        {
            // os << Location
            os << *first << " ";

            // 格式化：6 个一行
            if ( ++cnt == onLine )
                { os << "\n"; cnt = 0; }
        }
        return os;
    }
```

下面的程序利用了 WordCount 的新定义。为简单起见，单词的出现位置被手工编在代码中。如下所示：

```
    #include <iostream>
    #include "WordCount.h"

    int main()
    {
        WordCount search( "rosebud" );

        // 为简单起见，直接写入出现次数
        search.found(11,3); search.found(11,8);
        search.found(14,2); search.found(34,6);
        search.found(49,7); search.found(67,5);
        search.found(81,2); search.found(82,3);
        search.found(91,4); search.found(97,8);

        cout << "Occurrences: " << "\n"
            << search << endl;

        return 0;
    }
```

编译并执行该程序，将产生下列输出：

```
    Occurrences:
    <10> rosebud
    <11,3> <11,8> <14,2> <34,6> <49,7> <67,5>
    <81,2> <82,3> <91,4> <97,8>
```

该程序的输出被保存在名为 output 的文件中，我们接下来的努力目标是定义一个输入操作符来把它读回来。

练习 20.7

已知下面的 Date 类定义：

```
    class Date {
    public:
```

```
        // ...
    private:
        int month, day, year;
    };
```

请给出输出操作符的重载实例，

(a) 产生以下格式：

```
// 拼写月份
September 8th, 1997
```

(b) 产生以下格式：

```
9 / 8 / 97
```

(c) 哪一个更好，为什么？

(d) Date 的输出操作符应该是一个友元函数吗，为什么？

练习 20.8

请为下面的 CheckoutRecord 类定义输出操作符：

```
class CheckoutRecord {
public:
    // ...
private:
    double book_id;
    string title;
    Date date_borrowed;
    Date date_due;

    pair<string,string> borrower;
    vector< pair<string,string>* > wait_list;
};
```

20.5 重载输入操作符>>

重载输入操作符（>>）与重载输出操作符类似，只不过出错的可能性更大。例如，下面是 WordCount 输入操作符的实现：

```
#include <iostream>
#include "WordCount.h"

/* 必须修改 WordCount，指定输入操作符为友元
class WordCount {
friend ostream& operator<<( ostream&, const WordCount& );
friend istream& operator>>( istream&, WordCount& );
*/

istream&
operator>>( istream &is, WordCount &wd )
{
    /* WordCount 对象被读入的格式：
     * <2> string
     * <7,3> <12,36>
```

```
         */

        int ch;

        /* 读入小于符号，如果不存在
         * 则设置 istream 为失败状态并退出
         */

        if ((ch = is.get()) != '<' )
        {
            is.setstate( ios_base::failbit );
            return is;
        }

        // 读入多少个
        int occurs;
        is >> occurs;

        // 取 >; 不检查错误
        while ( is && (ch = is.get()) != '>' ) ;
        is >> wd._word;

        // 读入位置
        // 每个位置的格式: < line, col >
        for ( int ix = 0; ix < occurs; ++ix )
        {
            int line, col;

            // 提取值
            while (is && (ch = is.get())!= '<' ) ;
            is >> line;
            while (is && (ch = is.get())!= ',' ) ;
            is >> col;
            while (is && (ch = is.get())!= '>' ) ;
            wd.occurList.push_back( Location( line, col ));
        }
        return is;
    }
```

这个例子说明了一些可能的、与 iostream 错误状态相关的话题：

1．由于不正确的格式而导致失败，istream 应该把状态标记为 fail：

is.setstate(ios_base::failbit)

2．对于错误状态中的 iostream，插入和提取操作没有影响。例如：

while ((ch = is.get()) != lbrace)

如果 istream 对象 is 处于错误状态，则该循环将永远进行下去。这就是为什么在每次调用 get()之前必须测试条件的原因：

```
// 测试 "is" 是否处于正常状态
while ( is && (ch = is.get()) != lbrace)
```

如果 istream 对象没有处于正常状态，则它被测试为 false（我们将在 20.7 节更详细地了解 iostream 对象的条件状态）。

下面的程序读取一个 WordCount 类对象，其内容正是通过上节定义的重载输出操作符写入的：

```
#include <iostream>
#include "WordCount.h"

int main()
{
    WordCount readIn;

    // operator>>( cin, readIn )
    cin >> readIn;

    if ( !cin ) {
        cerr << "WordCount input error" << endl;
        return -1;
    }

    // operator<<( cout, readIn )
    cout << readIn << endl;
}
```

编译并执行该程序，产生下列输出：

```
<10> rosebud
<1,3> <11,8> <14,2> <34,6> <49,7> <67,5>
<81,2> <82,3> <91,4> <97,8>
```

练习 20.9

WordCount 输入操作符直接处理单独的 Location 项的输入，请把这部分代码抽取到一个独立的 Location 输入操作符中。

练习 20.10

请为在 20.4 节的练习 20.7 中定义的 Date 类提供一个输入操作符。

练习 20.11

请为在 20.4 节的练习 20.8 中定义的 CheckoutRecord 类提供一个输入操作符。

20.6 文件输入和输出

如果用户希望把一个文件连接到程序上，以便用来输入或输出，则必须包含 fstream 头文件（而这个头文件又包含了 iostream 头文件）：

```
#include <fstream>
```

为了打开一个仅被用于输出的文件。我们可以定义一个 ofstream（输出文件流）类对象。

例如：

```
ofstream outfile( "copy.out", ios_base::out );
```

传递给 ofstream 构造函数的实参分别指定了要打开的文件名和打开模式。ofstream 文件可以被打开为输出模式（ios_base::out）或附加模式（ios_base::app）。（在缺省情况下，ostream 文件以输出模式打开。）outfile2 的定义与 outfile 的定义等价：

```
// 缺省情况下，以输出模式打开
ofstream outfile2( "copy.out" );
```

如果在输出模式下打开已经存在的文件，则所有存储在该文件中的数据都将被丢弃。如果我们希望增加而不是替换现有文件中的数据，则应该以附加模式打开文件。于是，新写到文件中的数据将添加到文件尾部。在这两种模式下，如果文件不存在，程序都会创建一个新文件。

在试图读写文件之前，先判断它是否已成功打开，这是一个不错的主意，我们可以按如下方式测试 outfile：

```
if ( ! outFile ) { // 打开失败
     cerr << "cannot open "copy.out" for output\n";
     exit( -1 );
}
```

ofstream 从 ostrearn 类派生，且所有 ostream 操作都可以应用到一个 ofstream 类对象上，例如：

```
char ch = ' ';
outFile.put( '1' ).put( ')' ).put( ch );

outFile << "1 + 1 = " << (1 + 1) << endl;
```

向 outfile 中插入：

```
1) 1 + 1 = 2
```

以下的程序从标准输入获取字符，并将它输出到 copy.out 中：

```
#include <fstream>

int main()
{
     // 打开文件 copy.out 用于输出
     ofstream outFile( "copy.out" );

     if ( ! outFile ) {
          cerr << "Cannot open "copy.out" for output\n";
          return -1 ;
     }

     char ch;

     while ( cin.get( ch ) )
          outFile.put( ch );
}
```

用户定义的输出操作符实例也可以应用到 ofstream 类对象上。下面的程序调用了上节定义的 WordCount 输出操作符：

```
#include <fstream>
#include "WordCount.h"

int main()
{
    // 打开文件 word.out 用于输出
    ofstream oFile( "word.out" );

    // 在这里测试是否打开成功...
    // 手工创建和设置 WordCount 对象
    WordCount artist( "Renoir" );
    artist.found( 7, 12 ); artist.found( 34, 18 );

    // 调用 operator<<(ostream&, const WordCount&);
    oFile << artist;
}
```

为了打开一个仅用于输入的文件，我们可以使用 ifstream 类对象，ifstream 类从 istream 类派生。下面的程序读取了一个由用户指定的文件，并将内容写到标准输出上：

```
#include <fstream>
int main()
{
    cout << "filename: ";
    string file_name;
    cin >> file_name;

    // 打开文件 copy.out 用于输入
    ifstream inFile( file_name.c_str() );
    if ( !inFile ) {
        cerr << "unable to open input file: "
             << file_name << " -- bailing out!\n";
        return -1;
    }
    char ch;

    while ( inFile.get( ch ))
        cout.put( ch );
}
```

下面的程序依次读取 Alice Emma 文本文件，用 filter_string()过滤它（Alice Emma 文本以及 filter_string()的定义见 20.2.1 节），排序字符串以及去掉重复单词，然后再把结果文本写入到一个输出文件中：

```
#include <fstream>
#include <iterator>
#include <vector>
#include <algorithm>

template <class InputIterator>
void filter_string( InputIterator first, InputIterator last,
    string filt_elems = string("\",?."))
```

```
{
    for ( ; first != last; first++ )
    {
        string::size_type pos = 0;
        while ((pos=(*first).find_first_of(filt_elems,pos))
                != string::npos )
            (*first).erase( pos, 1 );
    }
}

int main()
{
    ifstream infile( "alice_emma" );

    istream_iterator<string> ifile( infile );
    istream_iterator<string> eos;

    vector< string > text;
    copy( ifile, eos, inserter( text, text.begin() ));
    string filt_elems( "\",.?;:" );
    filter_string( text.begin(), text.end(), filt_elems );

    vector<string>::iterator iter;
    sort( text.begin(), text.end() );
    iter = unique( text.begin(), text.end() );
    text.erase( iter, text.end() );

    ofstream outfile( "alice_emma_sort" );
    iter = text.begin();
    for ( int line_cnt = 1; iter != text.end();
            ++iter, ++line_cnt )
    {
        outfile << *iter << " ";
        if ( ! ( line_cnt % 8 ))
            outfile << '\n';
    }
    outfile << endl;
}
```

编译并运行该程序，产生如下输出：

```
A Alice Daddy Emma Her I Shyly a
alive almost asks at beautiful bird blows but
creature fiery flight flowing hair has he her
him in is it like long looks magical
mean more no red same says she shush
such tell tells the there through time to
untamed wanting when wind
```

在定义 ifstream 和 ofstream 类对象时，我们也可以不指定文件。以后可以通过成员函数 open()显式地把一个文件连接到一个类对象上。例如：

```
ifstream curFile;
```

```
// ...
curFile.open( filename.c_str() );
if ( ! curFile )  // 打开失败了吗?
    // ...
```

我们可以通过成员函数 close()断开一个文件与程序的连接。例如：

curFile.close();

在下面的程序中，用同一个 ifstream 类对象依次打开和关闭了五个文件：

```
#include <fstream>

const int fileCnt = 5;
string fileTabl[ fileCnt ] = {
    "Melville","Joyce","Musil","Proust","Kafka"
};

int main()
{
    ifstream inFile;  // 没有连接任何文件
    for ( int ix = 0; ix < fileCnt; ++ix )
    {
        inFile.open( fileTabl[ix].c_str() );

        // ... 判断是合打开成功
        // ... 处理文件
        inFile.close();
    }
}
```

fstream 类对象可以打开一个被用于输出或者输入的文件，fstream 类从 iostream 类派生而来。在下面的例子中，使用 fstream 类对象文件，对 word.out 先读后写。文件 word.out，在本节开始时被创建，它包含一个 WordCount 对象。如下所示：

```
#include <fstream>
#include "WordCount.h"

int main()
{
    WordCount wd;
    fstream file;

    file.open( "word.out", ios_base::in );
    file >> wd;
    file.close();
    cout << "Read in: " << wd << endl;

    // ios_base::out 将丢弃当前的数据
    file.open( "word.out", ios_base::app );
    file << endl << wd << endl;
    file.close();
}
```

fstream 类对象还可以打开一个同时被用于输入和输出的文件。例如，下面的定义以输入

和输出模式打开 word.out：

```
fstream io( "word.out", ios_base::in|ios_base::app );
```

按位或（OR）操作符被用来指定一种以上的模式。通过 seekg()或 seekp()成员函数，我们可以对 fstream 类对象重新定位［g 表示为了获取（getting）字符而定位（用于 ifstream 类对象），而 p 表示为放置字符（putting）而定位（用于 ofstream 类对象）］。这些函数移动到文件中的一个“绝对”地址，或者从特定位置移动一个偏移。seekg()和 seekp()有以下两种形式：

```
// 设置到文件中固定的位置上
seekg( pos_type current_position );

// 从当前位置向某个方向进行偏移
seekg( off_type offset_position, ios_base::seekdir dir );
```

在第一个版本中，当前位置被设置为由 current_position 指定的某个固定的位置，这里 0 是文件的开始。例如，如果一个文件由下列字符：

```
abc def ghi jkl
```

构成，则下列调用：

```
io.seekg( 6 );
```

把 io 重新定位到字符位置 6，在我们的例子中即字符 f。第二种形式使用一个偏移来重新定位文件，该偏移值或者是从当前位置开始计算，或者是到文件开始处的偏移，或者是从文件尾部倒退向后计算，这由第二个实参来指定。dir 可以被设置为以下选项之一：

1．ios_base::beg，文件的开始；

2．ios_base::cur，文件的当前位置；

3．ios_base::end，文件的结尾。

在下面的例子中，在每次迭代时，seekg()的每次调用都将文件重新定位在第 i 个记录项上：

```
for ( int i = 0; i < recordCnt; ++i )
    readFile.seekg( i * sizeof(Record), ios_base::beg );
```

第一个实参可以被指定为负数。例如，下面语句从当前位置向后移动 10 个字节：

```
readFile.seekg( -10, ios_base::cur );
```

fstream 文件中的当前位置由下面两个成员函数之一返回：tellg()或 tellp()（用在 ifstream 上，‘p’表示 putting；用在 ofstream 上，‘g’表示 getting）。例如：

```
// 标记出当前位置
ios_base::pos_type mark = writeFile.tellp();

// ...

if ( cancelEntry )
    // 返回到原先标记的位置上
    writeFile.seekp( mark );
```

如果程序员希望从文件的当前位置向前移动一个 Record，则可以用以下两种方法实现：

```
// 等价的做法：用于重新定位的 seek 调用
readFile.seekg( readFile.tellg() + sizeof(Record) );

// 这种方法被认为效率更高
readFile.seekg( sizeof(Record), ios_base::cur );
```

让我们来仔细地看一个实际的程序设计示例。给定一个要读取的文本文件，我们将计算文件的字节大小，并将它存储在文件尾部。另外，每次遇到一个换行符，我们都将当前的字节大小（包括换行符）存储在文件末尾。例如，已知文本文件：

```
abcd
efg
hi
j
```

程序应该生成下面修改之后的文本文件：

```
abcd
efg
hi
j
5 9 12 14 24
```

下面是我们的原始实现：

```
#include <iostream>
#include <fstream>

main()
{
    // 以输入和附加模式打开
    fstream inOut( "copy.out", ios_base::in|ios_base::app );
    int cnt = 0; // 字节计数器
    char ch;

    while ( inOut.get( ch ) )
    {
        cout.put( ch ); // 在终端回显
        ++cnt;
        if ( ch == '\n' ) {
            inOut << cnt ;
            inOut.put( ' ' ); // 空格
        }
    }

    // 输出最终的字节数
    inOut << cnt << endl;
    cout << "[ " << cnt << " ]" << endl;
    return 0;
}
```

inOut 是附在文件 copy.out 上的 fstream 类对象，它以输入和附加（append）两种模式打开。以附加模式打开的文件将把数据写到文件尾部。

每次读入一个字符时，包括空白字符但不包括文件结束符，我们把 cnt 加 1 并在用户终端上回显这个字符。将输入的字符回显到终端上，目的是让我们可以看清程序是否像期望的

那样工作。

每次遇到换行符时，我们都把 cnt 当前值写到 inOut 中。读到文件结束符则终止循环，再把 cnt 的最后值写到 inOut 和屏幕上。

编译程序，它似乎是正确的。我们用的文件含有 Moby Dick 的前几个句子，这是 19 世纪美国小说家 Herman Melville 写的作品：

```
Call me Ishmael. Some years ago, never mind
how long precisely, having little or no money
in my purse, and nothing particular to interest
me on shore, I thought I would sail about a little
and see the watery part of the world. It is a
way I have of driving off the spleen, and
regulating the circulation.
```

程序执行时，产生如下输出：

```
[ 0 ]
```

没有任何字符被显示出来，程序认为文本文件是空的，这显然不正确。我们对某些基本的概念理解有误。请别着急，也别沮丧，现在要做的事情就是仔细地想一想。问题在于，文件是以附加（append）模式打开的，所以它一开始就被定位在文件尾。当：

```
inOut.get( ch )
```

执行时。遇到了文件结束符，while 循环终止，因此 cnt 的值是 0。

虽然程序执行的结果很糟糕，但是，一旦找到了问题，解决的方案也就很简单了。我们所要做的，就是在开始读之前，把文件重新定位到文件的开始处。语句：

```
inOut.seekg( 0 );
```

所实现的正是这样的效果。重新编译并运行程序，这次产生如下输出：

```
Call me Ishmael. Some years ago, never mind
[ 45 ]
```

它只为文本文件的第一行产生了显示和计数结果，而余下的六行都被忽略了。唉，谁说程序设计很容易？这是程序员成长过程的一部分，尤其是当我们正在学习新东西的时候。（有时候记录下我们误解或做错的事情非常有用——尤其是以后当我们面对比我们更没有经验的人而失去耐心的时候。）现在我们需要做一个深呼吸，想想自己正在努力地做什么事情，以及做了什么事情。你发现问题了吗？

问题在于，文件以附加（append）模式被打开。第一次写 cnt 时，文件被重新定位到文件末尾，后面的 get()遇到文件结束符，因此结束了 while 循环。

这次的解决方案是把文件重新定位到写 cnt 之前的位置上，可以用下面两条语句来完成：

```
// 标记出当前位置
ios_base::pos_type mark = inOut.tellg();
inOut << cnt << sp;

inOut.seekg( mark );   // 恢复位置
```

重新编译并执行程序，终端上的输出结果是正确的。但是，检查输出文件，发现它仍然不对，最终字节数虽然已经被写到终端上，但是没有写到文件中。while 循环后面的输出操作

符没有被执行。

这次的问题是，inOut 处于“遇到文件结束符”的状态。只要 inOut 处于这种状态，就不能再执行输入和输出操作。解决方案是调用 clear()，清除文件的这种状态，可以用以下语句完成：

```
inOut.clear();     // 清除状态标记
```

完整的程序如下所示：

```
#include <iostream>
#include <fstream>

int main()
{
     fstream inOut( "copy.out", ios_base::in|ios_base::app );
     int cnt=0;
     char ch;
     inOut.seekg(0);

     while ( inOut.get( ch ) )
     {
          cout.put( ch );
          cnt++;
          if ( ch == '\n' )
          {
               // 标记当前位置
               ios_base::pos_type mark = inOut.tellg();
               inOut << cnt << ' ';
               inOut.seekg( mark ); // 恢复位置
          }
     }
     inOut.clear();
     inOut << cnt << endl;
     cout << "[ " << cnt << " ]\n";
     return 0;
}
```

重新编译并执行程序，终于产生了正确的输出。我们在实现程序时的错误是，没有为需要支持的行为提供显式的语句。每一个后续的方案都是针对一个出现的问题，而不是先分析整个问题，然后再提出完整的方案。虽然最终还是得到了同样的结果，但是与“开始时就全部认真地考虑问题”相比较，我们付出了更多的劳动和代价。

练习 20.12

请使用练习 20.10 中的 Date 类或练习 20.11 中的 CheckoutRecord 类定义的输出操作（都在 20.5 节），来编写一个程序以创建一个输出文件，并将数据写入其中。

练习 20.13

请写一个程序以打开并读取在练习 20.12 中创建的文件，然后再在标准输出上显示该文件的内容。

练习 20.14

请写一个程序，打开在练习 20.12 创建的文件，同时用于输入和输出。并且分别在以下位置输出一个 Date 类或 CheckoutRecord 类的实例：(a) 在文件的开始处；(b) 在第二个已有的对象之后；(c) 在文件末尾。

20.7 条件状态

一般来说，作为 iostream 库的用户，我们主要关心的是，一个流是否处于非错误状态。例如，如果我们写：

```
int ival;
cin >> ival;
```

并键入“Borges”。那么，在试图把一个字符串文字赋值给一个整型变量失败后，cin 被设置为一种错误状态。如果我们键入 1024，则读入成功，而 cin 仍处于正常状态。只有在正常状态下，输入流才执行读操作。

为了判断流对象是否处于正常状态，我们通常只是测试它的真值：

```
if ( !cin )
     // 读操作失败或遇到文件尾
```

为了读入未知数目的元素，我们一般如下编写 while 循环：

```
while ( cin >> word )
     // ok: 读操作成功
```

当到达文件结束符，或者在读操作期间发生错误条件时，while 循环的条件测试为 false。大多数情况下，对于流对象，这种形式的 trur/false 结果已经足够了。但是，在实现 20.5 节中的 Word 输入操作符的过程里，我们需要更细致地访问流的条件状态。

每个流对象都维护了一组条件标志，通过这些条件标志，我们可以监视流的当前状态。可以调用以下四个谓词成员函数：

1．如果一个流遇到文件结束符，则 eof()返回 true。例如：

```
if ( inOut.eof() )
     // ok, 都读进来了 ...
```

2．如果试图做一个无效的操作，比如 seeking 重定位操作超出了文件尾。则 bad()返回 true。一般地，这表示该流由于某种未定义的方式而被破坏了。

3．如果操作不成功，比如打开一个文件流对象失败或遇到一种无效的输入格式，则 fail()返回 true。例如：

```
ifstream iFile( filename, ios_base::in );

if ( iFile.fail() ) // 不能打开
     error_message( ... );
```

4．如果其他条件都不为 true，则 good()返回 true。例如：

```
if ( inOut.good() )
```

显式地修改流对象的条件状态有两种方式。第一，使用 clear()成员函数，可以把条件状态复位到一个显式的值。第二，使用 setstate()成员函数。我们可以不复位条件状态，而是在对象现有条件状态的基础上再增加一个条件。例如，在 WordCount 类的输入操作符中，当遇到一种无效格式时，我们用 setstate()为 istream 对象增加一个失败的条件：

```
if ((ch = is.get()) != '<' )
{
    is.setstate( ios_base::failbit );

    return is;
}
```

所有可用的条件值如下：

```
ios_base::badbit
ios_base::eofbit
ios_base::failbit
ios_base::goodbit
```

为了设置多个条件状态，我们可以如下使用按位 OR 操作符：

```
is.setstate( ios_base::badbit | ios_base::failbit );
```

在测试 20.5 节中的 WordCount 输入操作符时，我们写过：

```
if ( !( cin >> readIn ))
{
    cerr << "WordCount input error" << endl;
    exit( -1 );
}
```

作为其他的选择方案，我们或许会希望继续自己的程序，或许警告用户发生了输入错误，并要求再次输入。为了从 cin 中读取其他的输入，我们必须将它重新置于正常的状态，这可以通过使用 clear()成员函数来完成：

```
cin.clear();  // 将 cin 重设为正常
```

更一般地，clear()可以用来清除流对象的现有条件状态，并且设置 0 个或多个新的条件状态。例如：

```
cin.clear( ios_base::goodbit );
```

这可以显式地使 cin 恢复为正常的状态。（上面这两个调用是等价的，因为 clear()调用的缺省值是 goodbit 值。）

rdstate()成员函数使我们能够显式地访问 iostream 类对象的状态，例如：

```
ios_base::iostate old_state = cin.rdstate();
cin.clear();
process_input();

// 现在，cin 被重置为原来的状态
cin.clear( old_state );
```

练习 20.15

请改写练习 20.7 中的 Date 类和练习 20.8 中的 CheckoutRecord 类的输入操作符，以便设置 istream 对象的条件状态。另外再修改用来练习操作符的程序，以便检查显式设置的条件状态，并且一旦报告条件状态，就复位 istream 对象的条件状态。通过提供正常的和错误的格式，来使用修改后的程序。

20.8 string 流

iostream 库支持在 string 对象上的内存操作。ostringstream 类向一个 string 插入字符，istringstream 类从一个 string 对象读取字符，而 stringstream 类可以用来支持读和写两种操作。为了使用 string 流（字符串流），我们必须包含相关的头文件：

```
#include <sstream>
```

例如，下面的函数把整个文件 alice_emma 读到一个 ostringstream 类对象 buf 中。buf 随输入的字符而增长，以便容纳所有的字符：

```
#include <string>
#include <fstream>
#include <sstream>

string read_file_into_string()
{
    ifstream ifile( "alice_emma" );
    ostringstream buf;
    char ch;

    while ( buf && ifile.get( ch ))
        buf.put( ch );
    return buf.str();
}
```

成员函数 str()返回与 ostringstream 类对象相关联的 string 对象。我们可以用与处理普通 string 对象一样的方式，来操纵这个 string 对象。例如，在下面的程序中，我们用与 buf 关联的 string 对象按成员初始化 text：

```
int main()
{
    string text = read_file_into_string();

    // 标记出文本中每个换行符的位置
    vector< string::size_type > lines_of_text;
    string::size_type pos = 0;
    while ( pos != string::npos )
    {
        pos = text.find( '\n', pos );
        lines_of_text.push_back( pos );
    }
```

```
    // ...
}
```

ostringstream 对象也可以用来支持复合 string 的自动格式化，即，由多种数据类型构成的字符串。例如，输出操作符自动地将类型转换成相应的字符串表示，而不用担心所需的存储区大小：

```
#include <iostream>
#include <sstream>

int main()
{
    int ival = 1024; int *pival = &ival;
    double dval = 3.14159; double *pdval = &dval;
    ostringstream format_message;

    // ok: 把值转换成 string 表示
    format_message << "ival: " << ival
        << " ival's address: " << pival << '\n'
        << "dval: " << dval
        << " dval's address: " << pdval << endl;

    string msg = format_message.str();
    cout << " size of message string: " << msg.size()
        << " message: " << msg << endl;
}
```

在某些情况下，把非致命的诊断错误和警告集中在一起，而不是在遇到的地方显示出来，更受欢迎。有一种简单的方法可以做到这一点，就是提供一组一般形式的格式化重载函数：

```
string
format( string msg, int expected, int received )
{
    ostringstream message;

    message << msg << " expected: " << expected
            << " received: " << received << "\n";
    return message.str();
}

string format( string msg, vector<int> *values );

// ... 等等
```

于是，应用程序可以存储这些字符串，便于以后显示，或许可以按严重程度进行分类。一般地，它们可能会被分为 Notify、Log 或 Error 类。

istringstream 由一个 string 对象构造而来，它可以读取该 string 对象。istringstream 的一种用法是将数值字符串转换成算术值。例如：

```
#include <iostream>
#include <sstream>
#include <string>
```

```
int main()
{
    int ival = 1024; int *pival = &ival;
    double dval = 3.14159; double *pdval = &dval;

    // 创建一个字符串，存储每个值并
    // 用空格作为分割
    format_string << ival << " " << pival << " "
        << dval << " " << pdval << endl;

    // 提取出被存储起来的 ascii 值
    // 把它们依次放在四个对象中
    istringstream input_istring( format_string.str() );
    input_istring >> ival >> pival
        >> dval >> pdval;
}
```

练习 20.16

在 C 中，输出消息的格式化是使用标准 C 的 printf()函数族。例如，下列代码段：

```
int ival = 1024;
double dval = 3.14159;
char cval = 'a';
char *sval = "the end";
printf( "ival: %d\tdval: %g\tcval: %c\tsval: %s",
        val, dval, cval, sval );
```

产生：

```
ival: 1024    dval: 3.14159  cval: a    sval: the end
```

print()的第一个实参是一个格式字符串。每个%字符表示它将被一个实参值替代，而%后面的字符表示它的类型。下面是 printf()支持的某些可能的类型：

```
%d   integer
%g   floating point
%c   char
%s   C-style string
```

（完整的讨论见［KERNIGHAN88］。）

printf()的其他实参与每个“%格式对”按位置一一匹配，格式字符串的其他字符被视为文字常量，被直接输出。

printf()函数族的两个主要缺点是：①格式化字符串不能被扩展，因而不能识别用户定义的类型；②如果只是实参的类型和数目与格式字符串不匹配，则错误不能被检测出来，输出将很难看。printf()函数族的主要好处是，格式字符串非常紧凑。

(a) 请用一个 ostringstream 对象，产生等价的格式化输出：

(b) 请比较两种方法的优缺点。

20.9 格式状态

每一个 iostream 库对象都维护一个格式状态（format state），它控制格式化操作的细节，比如整型值的进制基数或浮点数值的精度。C++为程序员提供了一组预定义的操纵符（manipulator），可用来修改对象的格式状态。[32]

操纵符在流对象上的应用方式，与数据一样。但是，操纵符不会导致读写数据，只是修改流对象的内部状态。例如，缺省情况下，true 值的 bool 对象被写成整数值 1：

```
#include <iostream>

int main()
{
    bool illustrate = true;;
    cout << "bool object illustrate set to true: "
        << illustrate << '\n';
}
```

为了修改 cout，使它能够把 illustrate 显示为 true，我们应用 boolalpha 操作符：

```
#include <iostream>

int main()
{
    bool illustrate = true;;
    cout << "bool object illustrate set to true: ";

    // 改变 cout 的状态
    // 用字符串 true 和 false 输出 bool 值
    cout << boolalpha;
    cout << illustrate << '\n';
}
```

因为在应用操作符之后，仍然会返回原来被应用的流对象，所以我们可以把它的应用与数据的应用连接起来（或者与其他操作符的应用连接起来）。下面是重写之后的小程序，它混合了数据和操作符：

```
#include <iostream>
int main()
{
    bool illustrate = true;
    cout << "bool object illustrate: "
        << illustrate
        << "\nwith boolalpha applied: "
        << boolalpha << illustrate << '\n';

    // ...
```

[32] 另外，程序员还可以使用成员函数 setf()和 unsetf()。直接设置或解除格式状态标志。我们没有介绍这些操作，关于这种方法的讨论见［STROUSTRUP97］。

```
}
```

像这样，把操作符和数据混合起来容易产生误导作用。应用操作符之后，不只改变了后面输出值的表示形式，而且还修改了 ostream 的内部格式状态。在我们的例子中，整个程序的余下部分都将把 bool 值显示为 true 或 false。

为了消除对 cout 的修改，必须应用 noboolalpha 操作符：

```
cout << boolalpha      // 设置 cout 的内部状态
     << illustrate
     << noboolalpha    // 解除 cout 内部状态
```

我们将会看到，许多操作符都有类似的“设置/消除（set/unset）”对。

缺省情况下，算术值以十进制形式被读写。程序员可以通过使用 hex、oct 和 dec 操作符，把整数值的进制基数改为八进制或十六进制，或改回十进制（浮点值的表示不受影响）。例如：

```
#include <iostream>

int main()
{
     int ival = 16;
     double dval = 16.0;

     cout << "ival: " << ival
          << " oct set: " << oct << ival << "\n";
     cout << "dval: " << dval
          << " hex set: " << hex << dval << "\n";
     cout << "ival: " << ival
          << " dec set: " << dec << ival << "\n";
}
```

编译并执行该程序，产生下列输出：

```
ival: 16 oct set: 20
dval: 16 hex set: 16
ival: 10 dec set: 16
```

我们这个程序有一个问题，就是在看到一个值的时候无法知道它的进制基数。例如，20 是真正的 20，还是 16 的八进制表示？操纵符 showbase 可以让一个整数值在输出时指明它的基数，形式如下：

1．0x 开头表明是十六进制数（如果希望显示为大写字母，则可以应用 uppercase 操作符：为了转回小写的 x，我们可以应用 nouppercase 操作符）：

2．以 0 开头表示八进制数：

3．没有任何前导字符，表示十进制数。

下面是用 showbase 改写过的程序：

```
#include <iostream>

int main()
{
     int ival = 16;
     double dval = 16.0;
```

```
        cout << showbase;
        cout << "ival: " << ival
             << " oct set: " << oct << ival << "\n";
        cout << "dval: " << dval
             << " hex set: " << hex << dval << "\n";
        cout << "ival: " << ival << " dec set: "
             << dec << ival << "\n";
        cout << noshowbase;
    }
```

下面是修改后的输出：

```
ival: 16 oct set: 020
dval: 16 hex set: 16
ival: 0x10 dec set: 16
```

noshowbase 操作符重新设置 cout，使它不再显示整数值的进制基数。

缺省情况下，浮点值有 6 位的精度。这个值可以用成员函数 precision（int）或流操作符 setprecision()来修改（若使用后者，则必须包含 iomanip 头文件），precision()返回当前的精度值。例如：

```
#include <iostream>
#include <iomanip>
#include <math.h>

int main()
{
    cout << "Precision: "
         << cout.precision() << endl
         << sqrt(2.0) << endl;

    cout.precision(12);

    cout << "\nPrecision: "
         << cout.precision() << endl
         << sqrt(2.0) << endl;

    cout << "\nPrecision: " << setprecision(3)
         << cout.precision() << endl
         << sqrt(2.0) << endl;

    return 0;
}
```

编译并执行该程序，产生以下输出：

```
Precision: 6
1.41421
Precision: 12
1.41421356237
```

```
Precision: 3
1.41
```

当操作符带有一个实参，比如前面见到的 setprecision()和 setw()，就必须包含 iomanip 头文件：

```
#include <iomanip>
```

我们的例子没有说明有关 setprecision()的两个更深入的方面：①整数值不受影响；②浮点值被四舍五入而不是被截取。因此当精度为 4 时，3.14159 变成 3.142，精度为 3 时变成 3.14。

在缺省情况下，当小数部分为 0 时，不显示小数点。例如：

```
cout << 10.00
```

输出为：

```
10
```

为了强制显示小数点，使用 showpoint 操作符：

```
cout << showpoint
     << 10.0
     << noshowpoint << '\n';
```

noshowpoint 操作符重新设置缺省行为。

在缺省情况下，浮点值以定点小数法显示。为了改变为科学计数法，使用 scientific 操作符。为了改回到定点小数法，使用 fixed 操作符：

```
cout << "scientific: " << scientific
     << 10.0
     << "fixed decimal: " << fixed
     << 10.0 << '\n';
```

将产生：

```
scientific: 1.0e+01
fixed decimal: 10
```

如果希望把‘e’输出为‘E’，则可以使用 uppercase 操作符。如果想要转回小写字母，则使用 nouppercase 操作符。（uppercase 操作符不会使所有字母字符都显示为大写！）。

缺省情况下，重载的输入操作符跳过空白字符（空格、制表符、换行符、走纸以及回车）。已知序列：

```
a b c
d
```

则如下循环：

```
char ch;
while ( cin >> ch )
     // ...
```

将执行四次，以读入从 a 到 d 的四个字符，跳过中间的空格、可能的制表符和换行符。操作符 noskipws 使输入操作符不跳过空白字符：

```
char ch;
cin >> noskipws;
```

```
while ( cin >> ch )
    // ...
cin >> skipws;
```

现在，while 循环需要迭代七次，才能读入字符 a 到 d。为了转回到缺省行为，我们在 cin 上应用操纵符 skipws。

当我们写：

```
cout << "please enter a value: ";
```

文字字符串被存储在与 cout 相关联的缓冲区中。有许多种情况都可以引起缓冲区被刷新（即，清空）在我们的例子中，也就是将缓冲区写到标准输出上：

1．缓冲区可能会满，在这种情况下，它必须被刷新，以便读取后面的值。

2．我们可通过显式地使用 flush、ends 或 endl 操作符来刷新缓冲区。

```
// 清空缓冲区
cout << "hi!" << flush;

// 插入一个空字符然后刷新缓冲区
char ch[2]; ch[0] = 'a'; ch[1] = 'b';
cout << ch << ends;

// 插入一个换行符然后刷新缓冲区
cout << "hi!" << endl;
```

3．unitbuf，一个内部的流状态变量，若它被设置，则每次输出操作后都会清空缓冲区。

4．ostream 对象可以捆绑到 istream 上，在这种情况下，当 istream 从输入流读取数据时，ostream 的缓冲区就会被刷新。cout 被预定义为“捆绑”在 cin 上：

```
cin.tie( &cout );
```

语句：

```
cin >> ival;
```

使得与 cout 相关联的缓冲区被刷新。

一个 ostream 对象一次只能被捆绑到一个 istream 对象上，为了打破现有的捆绑，我们可以传递一个实参 0。例如：

```
istream is;
ostream new_os;

// ...

// tie() 返回现有的捆绑
ostream *old_tie = is.tie();
is.tie( 0 );            // 打破现有的捆绑
is.tie( &new_os );      // 设置新的捆绑

// ...
is.tie( 0 );            // 打破现有的捆绑
is.tie( old_tie );      // 重新建立原来的捆绑
```

我们可以用 setw()操作符来控制数字或字符串值的宽度。例如程序：

```
#include <iostream>
#include <iomanip>

int main()
{
    int ival = 16;
    double dval = 3.14159;

    cout << "ival: " << setw(12) << ival << '\n'
         << "dval: " << setw(12) << dval << '\n';
}
```

产生了以下输出：

```
ival:           16
dval:      3.14159
```

第二个 setw()是必需的，因为不像其他操作符，setw()不修改 ostream 对象的格式状态。

要想使输出的值左对齐，我们可以用 left 操作符（通过 right 操作符可以重新设回到缺省状态）来实现。如果我们希望产生：

```
   16
-   3
```

则可以应用 internal 操作符，它使得正负符号左对齐，而值右对齐，中间添加空格。如果希望用其他字符填充中间的空白，则可以应用 setfill()操作符。

```
cout << setw(6) << setfill('%') << 100 << endl;
```

产生：

```
%%%100
```

预定义的所有操作符都被列在表 20.1 中

表 20.1 操 作 符

操 作 符	含　　义
boolalpha	把 true 和 false 表示为字符串
*noboolalpha	把 true 和 false 表示为 0、1
showbase	产生前缀，指示数值的进制基数
*noshowbase	不产生进制基数前缀
showpoint	总是显示小数点
*noshowpoint	只有当小数部分存在时才显示小数点
Showpos	在非负数值中显示+
*noshowpos	在非负数值中不显示+

续表

操 作 符	含 义
*skipws	输入操作符跳过空白字符
noskipws	输入操作符不跳过空白字符
uppercase	在十六进制下显示 0X，科学计数法中显示 E
*nouppercase	在十六进制下显示 0x，科学计数法中显示 e
*dec	以十进制显示
hex	以十六进制显示
oct	以八进制显示
left	将填充字符加到数值的右边
right	将填充字符加到数值的左边
Internal	将填充字符加到符号和数值的中间
*fixed	以小数形式显示浮点数
scientific	以科学计数法形式显示浮点数
flush	刷新 ostream 缓冲区
ends	插入空字符，然后刷新 ostream 缓冲区
endl	插入换行符，然后刷新 ostream 缓冲区
ws	“吃掉” 空白字符
// 以下这些要求 #include <iomanip>	
setfill(ch)	用 ch 填充空白字符
setprecision(n)	将浮点精度设置为 n
setw(w)	按照 w 个字符来读或者写数值
setbase(b)	以进制基数 b 输出整数值

注：*表示缺省的流状态。

20.10 强类型库

iostream 库是强类型的。例如，试图从一个 ostream 读数据，或者写数据到一个 istream，都会在编译时刻被捕获到，并标记为类型违例。例如，已知下列声明：

```
#include <iostream>
#include <fstream>

class Screen;
extern istream& operator>>( istream&, const Screen& );
```

```
extern void print( ostream& );
ifstream inFile;
```

下面的两条语句都将导致编译时刻类型违例：

```
int main()
{
    Screen myScreen;

    // 错误：期望一个 ostream&
    print( cin >> myScreen );

    // 错误：期望 >> operator
    inFile << "error: output operator";
}
```

输入/输出设施是 C++标准库的一个组件。第 20 章没有描述整个 iostream 库——特别是，像“创建用户定义的操纵符和缓冲区类”这样的主题超出了本书的范围。我们关注的是 iostream 库的基础部分，通过这些基础部分程序员可以提供基本的输入/输出功能。

附 录

泛型算法（按字母排序）

在本附录中，我们将依次介绍每个单独的算法，我们选择以字母顺序给出这些算法（除了少数例外），以便可以很容易地查阅它们。本附录给出这些算法的一般形式是，①列出函数原型；②提供一两段说明性的文字，指出某些不直观的行为或可能性；以及最重要的③提供一个程序例子来说明怎样使用该算法。

所有泛型算法的前两个实参都是一对 iterator（迭代器），通常称为 first 和 last，标记出内置数组或容器中要操作的元素的范围。元素范围的表示法（有时也被称为左包含区间）通常被写作：

```
// 被读作：包括第一个以及它后面的每一个元素，但是不包括
// 最后一个.
[ first, last )
```

表示该范围从 first 开始，直到 last 结束，但是不包括 last。当表示为以下形式时：

```
first == last
```

该范围被称为是空的。

对于 iterator 对的要求是，它必须能够从 first 开始，通过反复应用递增操作符可以到达 last。但是，编译器自己不能保证这一点。不能满足这个要求将导致未定义的运行时刻行为——通常是程序的核心转储。

每个算法的声明都指出了其 iterator 必须支持的最小分类（关于五个 iterator 分类的简要讨论见 12.4 节）。例如，find()实现了对一个容器的单遍只读遍历，它至少需要一个 InputIterator。它也可以被传递一个 ForwardIterator、BidirectionalIterator 或 RamdomAccessIterator。然而，如果向它传递一个 OutputIterator 就会引起错误。给一个算法传递一个无效的 iterator 类别这样的错误并不一定会在编译时刻被检查出来，因为 iterator 类别不是实际的类型，而是被传递给函数模板的类型参数。

有些算法支持多个版本，一个版本利用内置操作符，而另一个版本接受一个函数对象或指向函数的指针，以便提供该操作符的替代实现。例如，缺省的 unique()利用容器的底层元素类型的等于操作符，来比较两个相邻的元素。但是，如果底层元素类型没有提供等于操作

符，或者我们希望定义不同的元素相等语义，那么可以传递一个函数的对象或指向函数的指针，然后再由该函数提供期望的语义。然而，另外一些算法被分成两个不同名字的实例，其中，第二个版本的实例都有后缀_if，比如 find_if()。例如，有一个使用内置等于操作符的 replace()实例，和一个带有函数对象或函数指针的 replace_if()实例。

对那些修改所操作容器的算法，一般有两个版本：一个是实地（in-place）版本，改变当前正被应用的容器；而在另一个版本中，将返回容器的一个拷贝，所做的修改被应用在这份拷贝上。例如，有 replace()和 replace_copy()两个算法，拷贝版本在名字中总会有_copy。但是，并不是每一个要改变相关容器的算法都有拷贝版本。例如，sort()算法就没有提供拷贝版本。在这种情况下，如果我们希望该算法在拷贝上进行操作，则需要自己做一份拷贝，并传递给算法。

要使用泛型算法，我们必须包含相关的头文件：

```
#include <algorithm>
```

如果要使用以下四个算术算法：adjacent_difference()、accumulate()、inner_product()以及 partial_sum()，则必须包含：

```
#include <numeric>
```

本附录中，实现算法的代码以及这些算法所操作的容器类型反映了当前可用的标准库实现。iostream 库反映了标准 C++之前的实现版本，例如，包括“使用 iostream.h 头文件”这样的行为。在模板机制中，模板参数不支持缺省实参。为了使程序能在读者当前的系统上运行，或许需要修改某些声明。

在[MUSSER96]，我们可以找到关于泛型算法更完美、更详细的讨论，虽然对于最终的 C++标准库而言，这些讨论有些过时。

accumulate()

```
template < class InputIterator, class Type >
Type accumulate(
                InputIterator first, InputIterator last,
                Type init );
template < class InputIterator, class Type,
          class BinaryOperation >
Type accumulate(
                InputIterator first, InputIterator last,
                Type init, BinaryOperation op );
```

accumulate()的第一个版本把由“iterator 对[first,last)”标记的序列中的元素之和，加到一个由 init 指定的初始值上。例如，已知序列{1,1,2,3,5,8}和初始值 0，则结果是 20。在第二个版本中，不再是做加法，而是传递进来的二元操作被应用在元素上。例如，如果向 accumulate()传递函数对象 times<int>，则结果是 240，当然，假设初始值是 1，而不是 0。accumulate()是一个算术算法。要使用它，我们必须包含<numeric>头文件。

```
#include <numeric>
#include <list>
```

```
#include <functional>
#include <iostream.h>

/*
 * 输出为：
   accumulate()
   operating on values {1,2,3,4}
   result with default addition: 10
   result with plus<int> function object: 10
*/

int main()
{
     int ia[] = { 1, 2, 3, 4 };
     list<int,allocator> ilist( ia, ia+4 );

     int ia_result = accumulate(&ia[0], &ia[4], 0);
     int ilist_res = accumulate(
          ilist.begin(), ilist.end(), 0, plus<int>() );

     cout << "accumulate()\n\t"
          << "operating on values {1,2,3,4}\n\t"
          << "result with default addition: "
          << ia_result << "\n\t"
          << "result with plus<int> function object: "
          << ilist_res
          << endl;
}
```

adjacent_difference()

```
template < class InputIterator, class OutputIterator >
OutputIterator adjacent_difference(
     InputIterator first, InputIterator last,
     OutputIterator result );
template < class InputIterator, class OutputIterator,
           class BinaryOperation >
OutputIterator adjacent_difference(
     InputIterator first, InputIterator last,
     OutputIterator result, BinaryOperation op );
```

adjacent_differece()的第一个版本创建了一个新的序列，该序列中的每个新值（第一个元素除外）都代表了当前元素与上一个元素的差。例如，已知序列{0,1,1,2,3,5,8}，则新序列的第一个元素只是原来序列第一个元素的拷贝：0。第二个元素是前两个元素的差：1。第三个元素是第二个和第三个元素的差，即 1-1，为 0，等等。新序列是{0,1,0,1,1,2,3}。

第二个版本用指定的二元操作计算相邻元素的差。例如，使用同一个序列，让我们传递 times<int>函数对象。同样，新序列的第一个元素只是原来序列第一个元素的拷贝：0。第二个元素是原来第一个和第二个元素的积，也是 0。第三个元素是第二和第三个元素的积，即

1*1，为 1 等等。新序列是{0,0,1,2,6,15,40}。

在两个版本中，OutputIterator 总是指向新序列末元素的下一个位置。adjacent_difference()是一种算术算法。使用这两个版本都必须包含头文件<numeric>。

```
#include <numeric>
#include <list>
#include <functional>
#include <iterator>
#include <iostream.h>

int main()
{
     int ia[] = { 1, 1, 2, 3, 5, 8 };
     list<int,allocator> ilist(ia, ia+6);
     list<int,allocator> ilist_result(ilist.size());
     adjacent_difference(ilist.begin(), ilist.end(),
          ilist_result.begin() );

     // 输出为:
     // 1 0 1 1 2 3
     copy( ilist_result.begin(), ilist_result.end(),
          ostream_iterator<int>(cout," "));
     cout << endl;
     adjacent_difference( ilist.begin(), ilist.end(),
          ilist_result.begin(), times<int>() );

     // 输出为:
     // 1 1 2 6 15 40
     copy( ilist_result.begin(), ilist_result.end(),
          ostream_iterator<int>(cout," "));
     cout << endl;
}
```

adjacent_find()

```
template < class ForwardIterator >
           ForwardIterator
adjacent_find( ForwardIterator first, ForwardIterator last );
template < class ForwardIterator, class BinaryPredicate >
           ForwardIterator
adjacent_find( ForwardIterator first,
                ForwardIterator last, Predicate pred );
```

adjacent_find()在由[first,last)标记的元素范围内，查找第一对相邻的重复元素。如果找到，则返回一个 ForwardIterator，并指向这对元素的第一个元素；否则返回 last。例如，已知序列{0,1,1,2,2,4}，元素对{1,1}被找到，函数返回指向第一个 1 的 iterator：

```
#include <algorithm>
#include <vector>
```

```
#include <iostream.h>
#include <assert.h>
class TwiceOver {
public:
     bool operator() ( int val1, int val2 )
     { return val1 == val2/2 ? true : false; }
};

int main()
{
     int ia[] = { 1, 4, 4, 8 };
     vector< int, allocator > vec( ia, ia+4 );
     int *piter;
     vector< int, allocator >::iterator iter;

     // piter 指向 ia[1]
     piter = adjacent_find( ia, ia+4 );
     assert( *piter == ia[ 1 ] );

     // iter 指向 vec[2]
     iter = adjacent_find( vec.begin(), vec.end(), TwiceOver() );
     assert( *iter == vec[ 2 ] );

     // 到达这里表示一切顺利
     cout < "ok: adjacent-find() succeeded!\n";
}
```

binary_search()

```
template< class ForwardIterator, class Type >
bool
binary_search( ForwardIterator first,
               ForwardIterator last, const Type &value );
bool
binary_search( ForwardIterator first,
               ForwardIterator last, const Type &value,
               Compare comp );
```

binary_search()在由[first,last)标记的有序序列中查找 value。如果找到，则返回 true。否则，返回 false。第一个版本假设该容器是用底层类型的小于操作符排序的。在第二个版本中，我们指出了该容器是用指定的函数对象进行排序的：

```
#include <algorithm>
#include <vector>
#include <assert.h>
int main()
{
     int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40};
     sort( &ia[0], &ia[12] );
     bool found_it = binary_search( &ia[0], &ia[12], 18 );
     assert( found_it == false );
```

```
    vector< int > vec( ia, ia+12 );
    sort( vec.begin(), vec.end(), greater<int>() );
    found_it = binary_search( vec.begin(), vec.end(),
        26, greater<int>() );
    assert( found_it == true );
}
```

copy()

```
template < class InputIterator, class OutputIterator >
OutputIterator
copy( InputIterator first1, InputIterator last,
      OutputIterator first2 );
```

copy()把由[first,last)标记的序列中的元素，拷贝到由 first2 标记为开始的地方。它返回 first2，但此时 first2 已经被移动到最后一个插入元素的下一位置。例如，已知序列{0,1,2,3,4,5}，我们可以用下列调用将序列左移 1 位：

```
int ia[] = { 0, 1, 2, 3, 4, 5 };

// 左移 1 位，结果为 {1,2,3,4,5,5}
copy( ia+1, ia+6, ia );
```

copy()从 ia 的第二个元素开始，把 1 拷贝到第一个位置上……，直到所有元素都被拷贝到它左边的位置上：

```
#include <algorithm>
#include <vector>
#include <iterator>
#include <iostream.h>

/* 生成：
   0 1 1 3 5 8 13
   将数组序列左移 1 位:
   1 1 3 5 8 13 13
   将 vector 序列左移 2 位:
   1 3 5 8 13 8 13
*/

int main()
{
    int ia[] = { 0, 1, 1, 3, 5, 8, 13 };
    vector< int, allocator > vec( ia, ia+7 );
    ostream_iterator< int > ofile( cout, " " );

    cout << "original element sequence:\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';

    // 左移 1 位
    copy( ia+1, ia+7, ia );
    cout << "shifting array sequence left by 1:\n";
    copy( ia, ia+7, ofile ); cout << '\n';
```

```
    // 左移 2 位
    copy( vec.begin()+2, vec.end(), vec.begin() );
    cout << "shifting vector sequence left by 2:\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';
}
```

copy_backward()

```
template < class BidirectionalIterator1,
           class BidirectionalIterator2 >
BidirectionalIterator2
copy_backward( BidirectionalIterator1 first,
               BidirectionalIterator1 last1,
               BidirectionalIterator2 last2 );
```

copy_backward()除了元素以相反的顺序被拷贝外，其他行为与 copy()相同。也就是说，拷贝操作从 last-1 开始，直到 first。这些元素也被从后向前拷贝到目标容器中，从 last2-1 开始，一直拷贝 last1-first 个元素。

例如，已知序列{0,1,2,3,4,5}，我们可以把最后三个元素(3,4,5)拷贝到前三个(0,1,2)中。做法是，把 first 设为值 0 的地址，last1 设为值 3 的地址，而 last2 设为值 5 的后一个位置。值为 5 的元素被赋给前面值为 2 的元素，而元素 4 被赋给前面值为 1 的元素。最后，元素 3 被赋给前面值为 0 的元素。结果序列是{3,4,5,3,4,5}。

```
#include <algorithm>
#include <vector>
#include <iterator>
#include <iostream.h>

class print_elements {
public:
    void operator()( string elem ) {
        cout << elem
             << ( _line_cnt++%8 ? " " : "\n\t" );
    }

    static void reset_line_cnt() { _line_cnt = 1; }
private:
    static int _line_cnt;
};

int print_elements::_line_cnt = 1;

/* 生成：
   原字符本为：
   The light untonsured hair grained and hued like
   pale oak
```

```
   copy_backward( begin+1, end-3, end ) 后的序列为:
   The light untonsured hair light untonsured hair grained
   and hued
*/

int main()
{
     string sa[] = {
          "The", "light", "untonsured", "hair",
          "grained", "and", "hued", "like", "pale", "oak" };

     vector< string, allocator > svec( sa, sa+10 );

     cout << "original list of strings:\n\t";
     for_each( svec.begin(), svec.end(), print_elements() );
     cout << "\n\n";

     copy_backward( svec.begin()+1, svec.end()-3, svec.end() );

     print_elements::reset_line_cnt();

     cout << "sequence after "
          << "copy_backward( begin+1, end-3, end ):\n";

     for_each( svec.begin(), svec.end(), print_elements() );
     cout << "\n";
}
```

count()

```
template< class InputIterator, class Type >
iterator_traits<InputIterator>::distance_type
count( InputIterator first,
       InputIterator last, const Type& value );
```

count()利用等于操作符，把[first,last)标记范围内的元素与 value 进行比较。并返回容器中与 value 相等的元素的个数。[注意，标准库的实现支持早期的 count()版本。]

```
#include <algorithm>
#include <string>
#include <list>
#include <iterator>
#include <assert.h>
#include <iostream.h>
#include <fstream.h>

/*****************************************************************
* text read:
Alice Emma has long flowing red hair. Her Daddy says
when the wind blows through her hair, it looks almost alive,
like a fiery bird in flight. A beautiful fiery bird, he tells her,
```

```
magical but untamed. "Daddy, shush, there is no such thing,"
she tells him, at the same time wanting him to tell her more.
Shyly, she asks, "I mean, Daddy, is there?"
*****************************************************************
* 程序输出：
* count(): fiery occurs 2 times
*****************************************************************
*/

int main()
{
     ifstream infile( "alice_emma" );
     assert ( infile != 0 );
     list<string,allocator> textlines;

     typedef list<string,allocator>::difference_type diff_type;
     istream_iterator< string, diff_type > instream( infile ),
                                                     eos;
     copy( instream, eos, back_inserter( textlines ));
     string search_item( "fiery" );

     /*****************************************************************
     * 注意：这是使用 count() 的标准 C++ 接口
     *       但是目前的 RogueWave 实现
     *       支持的是早期版本，其中没有开发 distance_type
     *       因此 count() 将通过
     *       一个参数返回值
     *
     * 调用方式如下：
     * typedef iterator_traits<InputIterator>::
     * distance_type dis_type;
     *
     * dis_type elem_count;
     * elem_count = count( textlines.begin(), textlines.end(),
     *                     search_item );
     *****************************************************************/
     int elem_count = 0;
     list<string,allocator>::iterator
          ibegin = textlines.begin(),
          iend = textlines.end();

     // count() 的过时形式
     count( ibegin, iend, search_item, elem_count );
     cout << "count(): " << search_item
          << " occurs " << elem_count << " times\n";
}
```

count_if()

```
template< class InputIterator, class Predicate >
```

```
iterator_traits<InputIterator>::distance_type
count_if( InputIterator first,
          InputIterator last, Predicate pred );
```

count_if()对于[first,last]标记范围内的每个元素都应用 pred，并返回 pred 计算结果为 true 的次数。

```
#include <algorithm>
#include <list>
#include <iostream.h>

class Even {
public:
     bool operator()( int val )
          { return val%2 ? false : true; }
};

int main()
{
     int ia[] = {0,1,1,2,3,5,8,13,21,34};
     list< int,allocator > ilist( ia, ia+10 );
/*
* 目前编译器不支持
*****************************************************
typedef
     iterator_traits<InputIterator>::distance_type
     distance_type;
     distance_type ia_count, list_count;

     // 计算偶数元素：4
     ia_count = count_if( &ia[0], &ia[10], Even() );
     list_count = count_if( ilist.begin(), ilist_end(),
                            bind2nd(less<int>(),10) );
*****************************************************
*/
     int ia_count = 0;
     count_if( &ia[0], &ia[10], Even(), ia_count );

     // 生成结果为：
     //    count_if(): there are 4 elements that are even.
     cout << "count_if(): there are "
          << ia_count < " elements that are even.\n";

     int list_count = 0;
     count_if( ilist.begin(), ilist.end(),
          bind2nd(less<int>(),10), list_count );

     // 生成结果为：
     // count_if(): there are 7 elements that are less than 10.
     cout << "count_if(): there are "
          << list_count
```

```
            << " elements that are less than 10.\n";
    }
```

equal()

```
template< class InputIterator1, class InputIterator2 >
bool
equal( InputIterator1 first1,
       InputIterator1 last, InputIterator2 first2 );
template< class InputIterator1, class InputIterator2,
          class BinaryPredicate >

bool
equal( InputIterator1 first1, InputIterator1 last,
       InputIterator2 first2, BinaryPredicate pred );
```

如果两个序列在范围[first,last)内包含的元素都相等，则 equal()返回 true。如果第二序列包含更多的元素，则不会考虑这些元素。如果我们希望保证两个序列完全相等；则需要写：

```
if ( vec1.size() == vec2.size() &&
     equal( vec1.begin(), vec1.end(), vec2.begin() );
```

或使用该容器的等于操作符，比如 vecl==vec2。如果第二个容器比第一个容器的元素少，算法的迭代过程应该超过其末尾，则运行时刻的行为是未定义的。缺省情况下，底层元素类型的等于操作符用来作比较，第二个版本应用 pred。

```
#include <algorithm>
#include <list>
#include <iostream.h>

class equal_and_odd{
public:
     bool
     operator()( int val1, int val2 )
     {
          return ( val1 == val2 &&
               ( val1 == 0 || val1 % 2 ));
     }
};

int main()
{
     int ia[] = { 0,1,1,2,3,5,8,13 };
     int ia2[] = { 0,1,1,2,3,5,8,13,21,34 };
     bool res;

     // true: 都等于 ia. 的长度
     // 生成结果为: int ia[7] equal to int ia2[9]? true.
     res = equal( &ia[0], &ia[7], &ia2[0] );
     cout << "int ia[7] equal to int ia2[9]? "
          << ( res ? "true" : "false" ) << ".\n";
     list< int, allocator > ilist( ia, ia+7 );
```

```
    list< int, allocator > ilist2( ia2, ia2+9 );

    // 生成结果为: list ilist equal to ilist2? true.
    res = equal( ilist.begin(), ilist.end(), ilist2.begin() );
    cout << "list ilist equal to ilist2? "
         << ( res ? "true" : "false" ) << ".\n";

    // false: 0, 2, 8 不相等，也不是奇数
    // 生成结果为: list ilist equal_and_odd() to ilist2? false.
    res = equal( ilist.begin(), ilist.end(),
         ilist2.begin(), equal_and_odd() );
    cout << "list ilist equal_and_odd() to ilist2? "
         << ( res ? "true" : "false" ) << ".\n";
    return 0;
}
```

equal_range()

```
template< class ForwardIterator, class Type >
pair< ForwardIterator, ForwardIterator >
equal_range( ForwardIterator first,
             ForwardIterator last, const Type &value );

template< class ForwardIterator, class Type, class Compare >
pair< ForwardIterator, ForwardIterator >
equal_range( ForwardIterator first,
             ForwardIterator last, const Type &value,
             Compare comp );
```

equal_range()返回一对 iterator，第一个 iterator 表示由 lower_bound()返回的 iterator 值，第二个表示由 upper_bound()返回的 iterator 值，它们的语义描述见相应的算法。例如，已知下面的序列：

```
int ia[] = {12,15,17,19,20,22,23,26,29,35,40,51};
```

用值 21 调用 equal_range()，返回一对 iterator，这两个 iterator 都指向值 22。用值 22 调用 equal_range()，返回一对 iterator，其中 first 指向值 22，second 指向值 23。第一个版本使用底层类型的小于操作符，第二个版本则用 comp 对元素进行排序：

```
#include <algorithm>
#include <vector>
#include <utility>
#include <iostream.h>

/* 生成结果为:
   array element sequence after sort:
   12 15 17 19 20 22 23 26 29 35 40 51

   equal_range result of search for value 23:
         *ia_iter.first: 23 *ia_iter.second: 26
```

```
    equal_range result of search for absent value 21:
          *ia_iter.first: 22 *ia_iter.second: 22

    vector element sequence after sort:
    51 40 35 29 26 23 22 20 19 17 15 12

    equal_range result of search for value 26:
          *ivec_iter.first: 26   *ivec_iter.second: 23

    equal_range result of search for absent value 21:
          *ivec_iter.first: 20   *ivec_iter.second: 20
*/

int main()
{
    int ia[] = { 29,23,20,22,17,15,26,51,19,12,35,40 };
    vector< int, allocator > ivec( ia, ia+12 );
    ostream_iterator< int > ofile( cout, " " );

    sort( &ia[0], &ia[12] );

    cout << "array element sequence after sort:\n";
    copy( ia, ia+12, ofile ); cout << "\n\n";

    pair< int*,int* > ia_iter;
    ia_iter = equal_range( &ia[0], &ia[12], 23 );

    cout << "equal_range result of search for value 23:\n\t"
         << "*ia_iter.first: " << *ia_iter.first < "\t"
         << "*ia_iter.second: " << *ia_iter.second < "\n\n";

    ia_iter = equal_range( &ia[0], &ia[12], 21 );

    cout << "equal_range result of search for "
         << "absent value 21:\n\t"
         << "*ia_iter.first: " << *ia_iter.first << "\t"
         << "*ia_iter.second: " << *ia_iter.second << "\n\n";

    sort( ivec.begin(), ivec.end(), greater<int>() );

    cout << "vector element sequence after sort:\n";
    copy( ivec.begin(), ivec.end(), ofile ); cout << "\n\n";

    typedef vector< int, allocator >::iterator iter_ivec;
    pair< iter_ivec, iter_ivec > ivec_iter;

    ivec_iter = equal_range( ivec.begin(), ivec.end(), 26,
        greater<int>() );
```

```
        cout << "equal_range result of search for value 26:\n\t"
             << "*ivec_iter.first: " << *ivec_iter.first << "\t"
             << "*ivec_iter.second: " << *ivec_iter.second
             << "n\n\";

        ivec_iter = equal_range( ivec.begin(), ivec.end(), 21,
             greater<int>() );

        cout << "equal_range result of search for "
             << "absent value 21:\n\t"
             << "*ivec_iter.first: " << *ivec_iter.first << "\t"
             << "*ivec_iter.second: " << *ivec_iter.second
             << "n\n\";
    }
```

fill()

```
template< class ForwardIterator, class Type >
void
fill( ForwardIterator first,
      ForwardIterator last, const Type& value );
```

fill()将 value 的拷贝赋给[first,last)范围内的所有元素。

```
#include <algorithm>
#include <list>
#include <string>
#include <iostream.h>

/* 结果为:
   original array element sequence:
   0 1 1 2 3 5 8

   array after fill(ia+1,ia+6):
   0 9 9 9 9 9 8

   original list element sequence:
   c eiffel java ada perl

   list after fill(++ibegin,--iend):
   c c++ c++ c++ perl
*/
int main()
{
    const int value = 9;
    int ia[] = { 0, 1, 1, 2, 3, 5, 8 };
    ostream_iterator< int > ofile( cout, " " );

    cout << "original array element sequence:\n";
    copy( ia, ia+7, ofile ); cout << "\n\n";

    fill( ia+1, ia+6, value );
```

```
    cout << "array after fill(ia+1,ia+6):\n";
    copy( ia, ia+7, ofile ); cout << "\n\n";

    string the_lang( "c++" );
    string langs[5] = { "c", "eiffel", "java", "ada", "perl" };

    list< string, allocator > il( langs, langs+5 );
    ostream_iterator< string > sofile( cout, " " );

    cout << "original list element sequence:\n";
    copy( il.begin(), il.end(), sofile ); cout << "\n\n";

    typedef list<string,allocator>::iterator iterator;

    iterator ibegin = il.begin(), iend = il.end();
    fill( ++ibegin, --iend, the_lang );

    cout << "list after fill(++ibegin,--iend):\n";
    copy( il.begin(), il.end(), sofile ); cout << "\n\n";
}
```

fill_n()

```
template< class ForwardIterator, class Size, class Type >
void
fill_n( ForwardIterator first,
        Size n, const Type& value );
```

fill_n()把 value 的拷贝赋给[first,first+count)范围内的 count 个元素：

```
#include <algorithm>
#include <vector>
#include <string>
#include <iostream.h>

class print_elements {
public:
    void operator()( string elem ) {
        cout << elem
             << ( _line_cnt++%8 ? " " : "\n\t" );
    }
    static void reset_line_cnt() { _line_cnt = 1; }
private:
    static int _line_cnt;
};

int print_elements::_line_cnt = 1;
/* 结果为:
   original element sequence of array container:
   0 1 1 2 3 5 8

   array after fill_n( ia+2, 3, 9 ):
```

```
    0 1 9 9 9 5 8

    原字符串序列为：
        Stephen closed his eyes to hear his boots
        crush crackling wrack and shells
        sequence after fill_n() applied:
        Stephen closed his xxxxx xxxxx xxxxx xxxxx xxxxx
        xxxxx crackling wrack and shells
*/

int main()
{
    int value = 9; int count = 3;
    int ia[] = { 0, 1, 1, 2, 3, 5, 8 };
    ostream_iterator< int > iofile( cout, " " );

    cout << "original element sequence of array container:\n";
    copy( ia, ia+7, iofile ); cout << "\n\n";
    fill_n( ia+2, count, value );
    cout << "array after fill_n( ia+2, 3, 9 ):\n";
    copy( ia, ia+7, iofile ); cout << "\n\n";

    string replacement( "xxxxx" );
    string sa[] = { "Stephen", "closed", "his", "eyes", "to",
        "hear", "his", "boots", "crush", "crackling",
        "wrack", "and", "shells" };

    vector< string, allocator > svec( sa, sa+13 );
    cout << "original sequence of strings:\n\t";
    for_each( svec.begin(), svec.end(), print_elements() );

    cout << "\n\n";
    fill_n( svec.begin()+3, count*2, replacement );
    print_elements::reset_line_cnt();

    cout << "sequence after fill_n() applied:\n\t";
    for_each( svec.begin(), svec.end(), print_elements() );
    cout << "\n";
}
```

find()

```
template< class InputIterator, class T >
InputIterator
find( InputIterator first,
      InputIterator last, const T &value );
```

find()利用底层元素类型的等于操作符，对[first,last)范围内的元素与 value 进行比较。当发现匹配时，结束搜索过程，且 find()返回指向该元素的一个 InputIterator。如果没有发现匹配，则返回 last：

```
#include <algorithm>
#include <iostream.h>
#include <list>
#include <string>

int main()
{
     int array[ 17 ] = { 7,3,3,7,6,5,8,7,2,1,3,8,7,3,8,4,3 };
     int elem = array[ 9 ];
     int *found_it;
     found_it = find( &array[0], &array[17], elem );

     // 结果: find the first occurrence of 1 found!
     cout << "find the first occurrence of "
          << elem < "\t"
          << ( found_it ? "found!\n" : "not found!\n" );

     string beethoven[] = {
          "Sonata31", "Sonata32", "Quartet14", "Quartet15",
          "Archduke", "Symphony7" };
     string s_elem( beethoven[ 1 ] );

     list< string, allocator > slist( beethoven, beethoven+6 );
     list< string, allocator >::iterator iter;
     iter = find( slist.begin(), slist.end(), s_elem );

     // 结果: find the first occurrence of Sonata32 found!
     cout << "find the first occurrence of "
          << s_elem < "\t"
          << ( iter != slist.end() ? "found!\n" : "not found!\n" );
}
```

find_if()

```
template< class InputIterator, class Predicate >
InputIterator
find_if( InputIterator first,
         InputIterator last, Predicate pred );
```

依次检查[first,last)范围内的元素，并把 pred 应用在这些元素上面。如果 pred 计算结果为 true，则搜索过程结束。find_if()返回指向该元素的 InputIterator。如果没有找到匹配，则返回 last。

```
#include <algorithm>
#include <list>
#include <set>
#include <string>
#include <iostream.h>

// 提供另一种等于操作符
```

```
// 如字符串包含在成员对象的
// 友元集中返回 true
class OurFriends {
public:
     bool operator()( const string& str ) {
          return ( friendset.count( str ));
     }
     static void
     FriendSet( const string *fs, int count ) {
          copy( fs, fs+count,
               inserter( friendset, friendset.end() ));
     }
private:
     static set< string, less<string>, allocator > friendset;
};

set< string, less<string>, allocator > OurFriends::friendset;
int main()
{
     string Pooh_friends[] = { "Piglet", "Tigger", "Eyeore" };
     string more_friends[] = { "Quasimodo", "Chip", "Piglet" };
     list<string,allocator> lf( more_friends, more_friends+3 );

     // 生成 pooh_friends 列表
     OurFriends::FriendSet( Pooh_friends, 3 );
     list<string,allocator>::iterator our_mutual_friend;
     our_mutual_friend =
          find_if( lf.begin(), lf.end(), OurFriends());

     // 结果:
     // Ah, imagine our friend Piglet is also a friend of Pooh.
     if ( our_mutual_friend != lf.end() )
          cout << "Ah, imagine our friend "
               << *our_mutual_friend
               << " is also a friend of Pooh.\n";
     return 0;
}
```

find_end()

```
template< class ForwardIterator1, class ForwardIterator2 >
ForwardIterator1
find_end( ForwardIterator1 first1, ForwardIterator1 last1,
          ForwardIterator2 first2, ForwardIterator2 last2 );
template< class ForwardIterator1, class ForwardIterator2,
          class BinaryPredicate >
ForwardIterator1
find_end( ForwardIterator1 first1, ForwardIterator1 last1,
          ForwardIterator2 first2, ForwardIterator2 last2,
          BinaryPredicate pred );
```

在由[first,last)标记的序列中查找“由 iterator 对[first2,last2)标记的第二个序列”的最后一次出现。例如，已知字符序列 mississippi 和第二个序列 ss，则 find_end()返回一个 ForwardIterator。指向第二个 ss 序列的第一个 s。如果在第一个序列中没有找到第二个序列，则返回 last1。在第一个版本中，使用底层的等于操作符。在第二个版本中，使用用户传递进来的二元操作 pred：

```
#include <algorithm>
#include <vector>
#include <iostream.h>
#include <assert.h>

int main()
{
     int array[ 17 ] = { 7,3,3,7,6,5,8,7,2,1,3,7,6,3,8,4,3 };
     int subarray[ 3 ] = { 3, 7, 6 };
     int *found_it;

     // 在数组中查找最后一次出现的 3,7,6 序列
     // 返回首元素的地址...
     found_it = find_end( &array[0], &array[17],
          &subarray[0], &subarray[3] );
     assert( found_it == &array[10] );

     vector< int, allocator > ivec( array, array+17 );
     vector< int, allocator > subvec( subarray, subarray+3 );
     vector< int, allocator >::iterator found_it2;

     found_it2 = find_end( ivec.begin(), ivec.end(),
          subvec.begin(), subvec.end(),
     equal_to<int>() );

     assert( found_it2 == ivec.begin()+10 );

     cout << "ok: find_end correctly returned beginning of "
          << "last matching sequence: 3,7,6!\n";
}
```

find_first_of()

```
template< class ForwardIterator1, class ForwardIterator2 >
ForwardIterator1
find_first_of( ForwardIterator1 first1, ForwardIterator1 last1,
               ForwardIterator2 first2, ForwardIterator2 last2
template< class ForwardIterator1, class ForwardIterator2,
          class BinaryPredicate >
ForwardIterator1
find_first_of( ForwardIterator1 first1, ForwardIterator1 last1,
               ForwardIterator2 first2, ForwardIterator2 last2,
               BinaryPredicate pred );
```

由[first2,last2)标记的序列包含了一组元素的集合，find_first_of()将在由[first1,last1)标记的序列中搜索这些元素。例如，假设我们希望在字符序列 synesthesia 中找到第一个元音。为了做到这一点，我们把第二个序列定义为 aeiou。find_first_of()返回一个 ForwardIterator，指向元音序列中的元素的第一个出现，本例中，指向第一个 e。如果第一个序列不含有第二个序列中的任何元素，则返回 last1。在第一个版本中，使用底层元素类型的等于操作符。在第二个版本中，使用二元操作 pred：

```
#include <algorithm>
#include <vector>
#include <string>
#include <iostream.h>
int main()
{
     string s_array[] = { "Ee", "eE", "ee", "Oo", "oo", "ee" };
     string to_find[] = { "oo", "gg", "ee" };

     // 返回第一次出现的 "ee" -- &s_array[2]
     string *found_it =
          find_first_of( s_array, s_array+6,
          to_find, to_find+3 );

     // 结果:
     // found it: ee
     //       &s_array[2]:   0x7fff2dac
     //       &found_it: 0x7fff2dac
     if ( found_it != &s_array[6] )
          cout << "found it: " << *found_it << "\n\t"
               << "&s_array[2]:\t" << &s_array[2] << "\n\t"
               << "&found_it:\t" << found_it << "n\n\";

     vector< string, allocator > svec( s_array, s_array+6);
     vector< string, allocator > svec_find( to_find, to_find+3 );

     // 返回找到的 "oo" -- svec.end()-2
     vector< string, allocator >::iterator found_it2;
     found_it2 = find_first_of(
          svec.begin(), svec.end(),
          svec_find.begin(), svec_find.end(
               equal_to<string>() );

     // 结果:
     // found it, too: oo
     //       &svec.end()-2: 0x100067b0
     //       &found_it2: 0x100067b0
     if ( found_it2 != svec.end() )
          cout << "found it, too: " << *found_it2 << "\n\t"
               << "&svec.end()-2:\t" << svec.end()-2 << "\n\t"
               << "&found_it2:\t" << found_it2 << "\n";
}
```

for_each()

```
template< class InputIterator, class Function >
Function
for_each( InputIterator first,
          InputIterator last, Function func );
```

for_each()依次对[first,last)范围内的所有元素应用函数 func，func 不能对元素执行写操作（因为前两个参数都是 InputIterator，所以不能保证支持赋值操作）。如果我们希望修改元素，则应该使用 transform()算法。func 可以返回值，但是该值会被忽略：

```
#include <algorithm>
#include <vector>
#include <iostream.h>

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }

int main()
{
     vector< int, allocator > ivec;
     for ( int ix = 0; ix < 10; ix++ )
          ivec.push_back( ix );
     void (*pfi)( int ) = print_elements;

     for_each( ivec.begin(), ivec.end(), pfi );
     return 0;
}
```

generate()

```
template< class ForwardIterator, class Generator >
void
generate( ForwardIterator first,
          ForwardIterator last, Generator gen );
```

generate()通过对 gen 的连续调用，来填充一个序列的[first,last)范围。gen 可以是函数对象或函数指针：

```
#include <algorithm>
#include <list>
#include <iostream.h>

int odd_by_twos() {
     static int seed = -1;
     return seed += 2;
}

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }
```

```
int main()
{
    list< int, allocator > ilist( 10 );
    void (*pfi)( int ) = print_elements;
    generate( ilist.begin(), ilist.end(), odd_by_twos );

    // 结果:
    // elements within list the first invocation:
    // 1 3 5 7 9 11 13 15 17 19
    cout << "elements within list the first invocation:\n";
    for_each( ilist.begin(), ilist.end(), pfi );
    generate( ilist.begin(), ilist.end(), odd_by_twos );

    // 结果:
    // elements within list the second iteration:
    // 21 23 25 27 29 31 33 35 37 39
    cout << "\n\nelements within list the second iteration:\n";
    for_each( ilist.begin(), ilist.end(), pfi );

    return 0;
}
```

generate_n()

```
template< class ForwardIterator,
          class Size, class Generator >
void
generate_n( OutputIterator first, Size n, Generator gen );
```

generate_n()通过对 gen 的 n 次连续调用，来填充一个序列中从 first 开始的 n 个元素。gen 可以是函数对象或函数指针：

```
#include <algorithm>
#include <iostream.h>
#include <list>

class even_by_twos {
public:
    even_by_twos( int seed = 0 ) : _seed( seed ){}
    int operator()() { return _seed += 2; }
private:
    int _seed;
};

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }

int main()
{
    list< int, allocator > ilist( 10 );
    void (*pfi)( int ) = print_elements;
```

```
    generate_n( ilist.begin(), ilist.size(), even_by_twos() );

    // 结果:
    // generate_n with even_by_twos():
    // 2 4 6 8 10 12 14 16 18 20
    cout << "generate_n with even_by_twos():\n";
    for_each( ilist.begin(), ilist.end(), pfi ); cout << "\n";
    generate_n(ilist.begin(),ilist.size(),even_by_twos(100));

    // 结果:
    // generate_n with even_by_twos( 100 ):
    // 102 104 106 108 110 112 114 116 118 120
    cout << "generate_n with even_by_twos( 100 ):\n";
    for_each( ilist.begin(), ilist.end(), pfi );
}
```

includes()

```
template< class InputIterator1, class InputIterator2 >
bool
includes( InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2 );
template< class InputIterator1, class InputIterator2,
          class Compare >
bool
includes( InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          Compare comp );
```

includes()判断[first1,last1)的每一个元素是否被包含在序列[first2,last2)中。第一个版本假设这两个序列是用底层元素类型的小于操作符排序的，第二个版本用 comp 来判定元素顺序。

```
#include <algorithm>
#include <vector>
#include <iostream.h>
int main()
{
    int ia1[] = { 13, 1, 21, 2, 0, 34, 5, 1, 8, 3, 21, 34 };
    int ia2[] = { 21, 2, 8, 3, 5, 1 };

    // includes 必须传入已排序的容器
    sort( ia1, ia1+12 ); sort( ia2, ia2+6 );

    // 结果: every element of ia2 contained in ia1? true
    bool res = includes( ia1, ia1+12, ia2, ia2+6 );
    cout << "every element of ia2 contained in ia1? "
         << (res ? "true" : "false") << endl;

    vector< int, allocator > ivect1( ia1, ia1+12 );
    vector< int, allocator > ivect2( ia2, ia2+6 );
```

```
    // 按降序排序
    sort( ivect1.begin(), ivect1.end(), greater<int>() );
    sort( ivect2.begin(), ivect2.end(), greater<int>() );

    res = includes( ivect1.begin(), ivect1.end(),
                    ivect2.begin(), ivect2.end(),
                    greater<int>() );

    // 结果:
    // every element of ivect2 contained in ivect1? true
    cout << "every element of ivect2 contained in ivect1? "
         << (res ? "true" : "false") << endl;
}
```

inner_product()

```
template < class InputIterator1, class InputIterator2,
           class Type >
Type
inner_product(
              InputIterator1 first1, InputIterator1 last,
              InputIterator2 first2, Type init );
template < class InputIterator1, class InputIterator2,
           class Type,
           class BinaryOperation1, class BinaryOperation2 >
Type
inner_product(
              InputIterator1 first1, InputIterator1 last,
              InputIterator2 first2, Type init,
              BinaryOperation1 op1, BinaryOperation2 op2 );
```

inner_product()的第一个版本对两个序列做内积（对应的元素相乘，再求和），并将内积加到一个由 init 指定的初始值上。第一个序列由[first1,last)标记，第二个序列由 first2 开始，随着第一个序列而逐渐递增。例如，已知序列{2,3,5,8}和{1,2,3,4}，则下列乘积对的和就是结果：

```
2*1 + 3*2 + 5*3 + 8*4
```

如果提供初始值 0，则结果是 55。

第二个版本用二元操作 op1 代替缺省的加法操作，用二元操作 op2 代替缺省的乘法操作。例如，如果同样用上两个序列，指定 op1 为减法，op2 为加法，则结果是下列加法对的差：

```
(2+1) - (3+2) - (5+3) - (8+4)
```

inner_product()是一个算术算法。要使用它，必须包含头文件<numeric>：

```
#include <numeric>
#include <vector>
#include <iostream.h>

int main()
{
```

```
    int ia[] = { 2, 3, 5, 8 };
    int ia2[] = { 1, 2, 3, 4 };

    // 两个数组的元素两两相乘，
    // 并将结果添加到初始值：0
    int res = inner_product( &ia[0], &ia[4], &ia2[0], 0);

    // 结果：inner product of arrays: 55
    cout << "inner product of arrays: "
         << res << endl;
    vector<int, allocator> vec( ia, ia+4 );
    vector<int, allocator> vec2( ia2, ia2+4 );

    // 两个向量中的元素相加
    // 并从初始值中减去和：0
    res = inner_product( vec.begin(), vec.end(),
         vec2.begin(), 0,
         minus<int>(), plus<int>() );

    // 结果：inner product of vectors: -28
    cout << "inner product of vectors: "
         << res << endl;
    return 0;
}
```

inplace_merge()

```
template< class BidirectionalIterator >
void
inplace_merge( BidirectionalIterator first,
               BidirectionalIterator middle,
               BidirectionalIterator last );
template< class BidirectionalIterator, class Compare >
void
inplace_merge( BidirectionalIterator first,
                BidirectionalIterator middle,
                BidirectionalIterator last, Compare comp );
```

inplace_merge()合并两个排过序的连续序列，分别由[first,middle)和[middle,last)标记。结果序列覆盖了由 first 开始的这两段范围。第一个版本使用底层类型的小于操作符对元素进行排序，第二个版本根据程序员传递的二元比较操作对元素进行排序：

```
#include <algorithm>
#include <vector>
#include <iostream.h>

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }

/*
 * 结果：
```

```
        ia sorted into two subarrays:
        12 15 17 20 23 26 29 35 40 51 10 16 21 41 44 54 62 65 71 74

        ia inplace_merge:
        10 12 15 16 17 20 21 23 26 29 35 40 41 44 51 54 62 65 71 74

        ivec sorted into two subvectors:
        51 40 35 29 26 23 20 17 15 12 74 71 65 62 54 44 41 21 16 10

        ivec inplace_merge:
        74 71 65 62 54 51 44 41 40 35 29 26 23 21 20 17 16 15 12 10
    */
    int main()
    {
        int ia[] = { 29,23,20,17,15,26,51,12,35,40,
            74,16,54,21,44,62,10,41,65,71 };
        vector< int, allocator > ivec( ia, ia+20 );
        void (*pfi)( int ) = print_elements;

        // 以一事实上排序排列两上子序列
        sort( &ia[0], &ia[10] );
        sort( &ia[10], &ia[20] );

        cout << "ia sorted into two sub-arrays: \n";
        for_each( ia, ia+20, pfi ); cout << "\n\n";

        inplace_merge( ia, ia+10, ia+20 );
        cout << "ia inplace_merge:\n";
        for_each( ia, ia+20, pfi ); cout << "\n\n";

        sort( ivec.begin(), ivec.begin()+10, greater<int>() );
        sort( ivec.begin()+10, ivec.end(), greater<int>() );

        cout << "ivec sorted into two sub-vectors: \n";
        for_each( ivec.begin(), ivec.end(), pfi ); cout << "\n\n";
        inplace_merge( ivec.begin(), ivec.begin()+10,
            ivec.end(), greater<int>() );

        cout << "ivec inplace_merge:\n";
        for_each( ivec.begin(), ivec.end(), pfi ); cout << endl;
    }
```

iter_swap()

```
    template <class ForwardIterator1, class ForwardIterator2>
    void
    iter_swap ( ForwardIterator1 a, ForwardIterator2 b );
```

iter_swap()交换由两个 ForwardIterator：a 和 b 所指向的元素中的值。

```
    #include <algorithm>
```

```
#include <list>
#include <iostream.h>
int main()
{
     int ia[] = { 5, 4, 3, 2, 1, 0 };
     list< int,allocator > ilist( ia, ia+6 );
     typedef list< int, allocator >::iterator iterator;
     iterator iter1 = ilist.begin(), iter2,
          iter_end = ilist.end();

     // 对列表进行冒泡排序...
     for ( ; iter1 != iter_end; ++iter1 )
          for ( iter2 = iter1; iter2 != iter_end; ++iter2 )
               if ( *iter2 < *iter1 )
                    iter_swap( iter1, iter2 );

     // 输出结果为:
     // ilist after bubble sort using iter_swap():
     // { 0 1 2 3 4 5 }
     cout << "ilist afer bubble sort using iter_swap(): { ";
     for ( iter1 = ilist.begin(); iter1 != iter_end; ++iter1 )
          cout << *iter1 << " ";
     cout << "}\n";
}
```

lexicographical_compare()

```
template <class InputIterator1, class InputIterator2 >
bool
lexicographical_compare(
          InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2 );
template < class InputIterator1, class InputIterator2,
           class Compare >
bool
lexicographical_compare(
               InputIterator1 first1, InputIterator1 last1,
               InputIterator2 first2, InputIterator2 last2,
               Compare comp );
```

lexicographical_compare()比较由[first1,last1)和[first2,last2)标识的两个序列的对应元素对。比较操作将一直进行下去，直到某个元素对不匹配，或者到达[last1,last2]对，或者到达last1 或 last2（如果两个序列长度不等）。对于第一个不匹配的元素对，发生以下事情：

- 如果第一个序列的元素小，则返回 true，否则返回 false。
- 如果到达 last1，而 last2 未到，则返回 true。
- 如果到达 last2，而未到达 last1，则返回 false。
- 如果 last1 和 last2 都已到达（所有元素都匹配），则返回 false。即第一个序列在字典序上不小于第二个序列。

例如，已知下列两个序列：

```
string arr1[] = { "Piglet", "Pooh", "Tigger" };
string arr2[] = { "Piglet", "Pooch", "Eeyore" };
```

本算法在第一个元素对上匹配，但是在第二个上不匹配，Pooh 大于 Pooch，因为 c 在字典序上小于 h（想像一下字典中的单词是如何排序的）。算法在这一点上停止（不再比较第三个元素），比较的结果是 false。

算法的第二个版本使用了比较对象，而不再使用底层元素类型的小于操作符。

```
#include <algorithm>
#include <list>
#include <string>
#include <assert.h>
#include <iostream.h>

class size_compare {
public:
    bool operator()( const string &a, const string &b ) {
        return a.length() <= b.length();
    }
};

int main()
{
    string arr1[] = { "Piglet", "Pooh", "Tigger" };
    string arr2[] = { "Piglet", "Pooch", "Eeyore" };
    bool res;

    // 第二个元素值为 false
    // Pooch 小于 Pooh
    // 第二个元素值也为 false
    res = lexicographical_compare( arr1, arr1+3,
        arr2, arr2+3 );
    assert( res == false );

    // 值为 true: ilist2 每个元素的
    // 长度都小于或等于
    // 对应的 ilist1 的元素

    list< string, allocator > ilist1( arr1, arr1+3 );
    list< string, allocator > ilist2( arr2, arr2+3 );

    res = lexicographical_compare(
        ilist1.begin(), ilist1.end(),
        ilist2.begin(), ilist2.end(), size_compare() );

    assert( res == true );
    cout << "ok: lexicographical_compare succeeded!\n";
}
```

lower_bound()

```
template< class ForwardIterator, class Type >
ForwardIterator
lower_bound( ForwardIterator first,
             ForwardIterator last, const Type &value );
template< class ForwardIterator, class Type, class Compare >
ForwardIterator
lower_bound( ForwardIterator first,
             ForwardIterator last, const Type &value,
             Compare comp );
```

lower_bound()返回一个 iterator，它指向在[first,last)标记的有序序列中可以插入 value、而不会破坏容器顺序的第一个位置，而这个位置标记了一个大于等于 value 的值。例如，已知下列序列：

```
int ia[] = {12,15,17,19,20,22,23,26,29,35,40,51};
```

用值 21 调用 lower_bound()，返回一个指向值 22 的 iterator。用值 22 调用 lower_bound()，也返回一个指向值 22 的 iterator。第一个版本使用底层类型的小于操作符，第二个版本根据 comp 对元素进行排序和比较。

```
#include <algorithm>
#include <vector>
#include <iostream.h>

int main()
{
    int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40};
    sort( &ia[0], &ia[12] );
    int search_value = 18;
    int *ptr = lower_bound( ia, ia+12, search_value );

    // 结果:
    // The first element 18 can be inserted in front of is 19
    // The previous value is 17
    cout << "The first element "
         << search_value
         << " can be inserted in front of is "
         << *ptr << endl
         << "The previous value is "
         << *(ptr-1) << endl;

    vector< int, allocator > ivec( ia, ia+12 );

    // 降序排序...
    sort( ivec.begin(), ivec.end(), greater<int>() );
    search_value = 26;
    vector< int, allocator >::iterator iter;

    // 告诉它这里所用的
```

```
        // 正确的排序关系...
        iter = lower_bound( ivec.begin(), ivec.end(),
        search_value, greater<int>() );
        // 结果:
        // The first element 26 can be inserted in front of is 26
        // The previous value is 29
        cout << "The first element "
             << search_value
             << " can be inserted in front of is "
             << *iter << endl
             << "The previous value is "
             << *(iter-1) << endl;
}
```

max()

```
template< class Type >
const Type&
max( const Type &aval, const Type &bval );
template< class Type, class Compare >
const Type&
max( const Type &aval, const Type &bval, Compare comp );
```

max()返回 aval 和 bval 两个元素中较大的一个。第一个版本使用与 Type 相关联的大于操作符，第二个版本使用比较操作 comp：

max_element()

```
template< class ForwardIterator >
ForwardIterator
max_element( ForwardIterator first,
             ForwardIterator last );
template< class ForwardIterator, class Compare >
ForwardIterator
max_element( ForwardIterator first,
             ForwardIterator last, Compare comp );
```

max_element()返回一个 iterator，指向[first,last)序列中值为最大的元素。第一个版本使用底层元素类型的大于操作符，第二个版本使用比较操作 comp：

min()

```
template< class Type >
const Type&
min( const Type &aval, const Type &bval );
template< class Type, class Compare >
const Type&
min( const Type &aval, const Type &bval, Compare comp );
```

min()返回 aval 和 bval 两个元素中较小的一个。第一个版本使用与 Type 相关联的小于操

作符，第二个版本使用比较操作 comp：

min_element()

```
template< class ForwardIterator >
ForwardIterator
min_element( ForwardIterator first,
             ForwardIterator last );
template< class ForwardIterator, class Compare >
ForwardIterator
min_element( ForwardIterator first,
             ForwardIterator last, Compare comp );
```

min_element()返回一个 iterator，指向[first,last)序列中值为最小的元素。第一个版本使用底层元素类型的小于操作符，第二个版本使用比较操作 comp：

```
// 说明 max(), min(), max_element(), min_element() 的用法
#include <algorithm>
#include <vector>
#include <iostream.h>

int main()
{
     int ia[] = { 7, 5, 2, 4, 3 };
     const vector< int, allocator > ivec( ia, ia+5 );

     int mval = max( max( max( max( ivec[4], ivec[3]),
          ivec[2]),ivec[1]),ivec[0]);

     // 输出: the result of nested invocations of max() is: 7
     cout << "the result of nested invocations of max() is: "
          << mval << endl;
     mval = min( min( min( min( ivec[4], ivec[3]),
     ivec[2]),ivec[1]),ivec[0]);

     // 输出: the result of nested invocations of min() is: 2
     cout << "the result of nested invocations of min() is: "
          << mval << endl;
     vector< int, allocator >::const_iterator iter;
     iter = max_element( ivec.begin(), ivec.end() );

     // 输出: the result of invoking max_element() is also: 7
     cout << "the result of invoking max_element() is also: "
          << *iter << endl;
     iter = min_element( ivec.begin(), ivec.end() );

     // 输出: the result of invoking min_element() is also: 2
     cout << "the result of invoking min_element() is also: "
          << *iter << endl;
}
```

merge()

```
template< class InputIterator1, class InputIterator2,
          class OutputIterator >
OutputIterator
merge( InputIterator1 first1, InputIterator1 last1,
       InputIterator2 first2, InputIterator2 last2,
       OutputIterator result );
template< class InputIterator1, class InputIterator2,
          class OutputIterator, class Compare >
OutputIterator
merge( InputIterator1 first1, InputIterator1 last1,
       InputIterator2 first2, InputIterator2 last2,
       OutputIterator result, Compare comp );
```

merge()把两个分别由[first1,last1)和[first2,last2)标记的有序序列，合并到一个从 result 开始的单个序列中，并返回一个 OutputIterator，指向新序列中最后一个元素的下一位置。第一个版本使用底层类型的小于操作符对元素进行排序，第二个版本根据 comp 对元素进行排序：

```
#include <algorithm>
#include <vector>
#include <list>
#include <deque>
#include <iostream.h>

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }
void (*pfi)( int ) = print_elements;

int main()
{
     int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40};
     int ia2[] = {74,16,39,54,21,44,62,10,27,41,65,71};
     vector< int, allocator > vec1( ia, ia +12 ),
          vec2( ia2, ia2+12 );
     int ia_result[24];
     vector<int,allocator> vec_result(vec1.size()+vec2.size());
     sort( ia, ia +12 );
     sort( ia2, ia2+12 );

     // 输出：
     // 10 12 15 16 17 19 20 21 22 23 26 27 29 35
     //       39 40 41 44 51 54 62 65 71 74
     merge( ia, ia+12, ia2, ia2+12, ia_result );
     for_each( ia_result, ia_result+24, pfi ); cout << "\n\n";

     sort( vec1.begin(), vec1.end(), greater<int>() );
     sort( vec2.begin(), vec2.end(), greater<int>() );

     merge( vec1.begin(), vec1.end(),
```

```
                vec2.begin(), vec2.end(),
                vec_result.begin(), greater<int>() );

        // 输出：
        // 74 71 65 62 54 51 44 41 40 39 35 29 27 26 23 22
        // 21 20 19 17 16 15 12 10
        for_each( vec_result.begin(), vec_result.end(), pfi );
        cout << "\n\n";
}
```

mismatch()

```
template< class InputIterator1, class InputIterator2 >
pair<InputIterator1, InputIterator2>
mismatch( InputIterator1 first1,
          InputIterator1 last, InputIterator2 first2 );
template< class InputIterator1, class InputIterator2,
          class BinaryPredicate >
pair<InputIterator1, InputIterator2>
mismatch( InputIterator1 first1, InputIterator1 last,
          InputIterator2 first2, BinaryPredicate pred );
```

mismatch()并行地比较两个序列，指出第一个“元素不匹配”的位置。它返回一对 iterator，标识出第一个元素不匹配的位置。如果所有的元素都匹配，则返回指向每个容器 last 元素的 iterator。例如，已知序列 meet 和 meat，则两个被返回的 iterator 分别指向第三个元素。缺省情况下，用等于操作符对元素进行比较。第二个版本允许用户指定一个比较操作。如果第二个序列比第一个序列的元素多，这些元素将被忽略。如果第二个序列比第一个序列的元素少，则运行时刻的行为是未定义的：

```
#include <algorithm>
#include <list>
#include <utility>
#include <iostream.h>

class equal_and_odd{
public:
    bool operator()( int ival1, int ival2 )
    {
        // 两个值相等吗，或
        // 都为 0 或都为奇数
        return ( ival1 == ival2 &&
            ( ival1 == 0 || ival1%2 ));
    }
};
int main()
{
    int ia[] = { 0,1,1,2,3,5,8,13 };
    int ia2[] = { 0,1,1,2,4,6,10 };

    pair<int*,int*> pair_ia = mismatch( ia, ia+7, ia2 );
```

```
    // 输出: first mismatched pair: ia: 3 and ia2: 4
    cout << "first mismatched pair: ia: "
         << *pair_ia.first << " and ia2: "
         << *pair_ia.second << endl;
    list<int,allocator> ilist( ia, ia+7 );
    list<int,allocator> ilist2( ia2, ia2+7 );
    typedef list<int,allocator>::iterator iter;
    pair< iter,iter > pair_ilist =
         mismatch( ilist.begin(), ilist.end(),
         ilist2.begin(), equal_and_odd() );
    // 输出:
    // first mismatched pair either not equal or not odd:
    //       ilist: 2 and ilist2: 2
    cout << "first mismatched pair either not equal "
         << "or not odd: \n\tilist: "
         << *pair_ilist.first << " and ilist2: "
         << *pair_ilist.second << endl;
}
```

next_permutation()

```
template< class BidirectionalIterator >
bool
next_permutation( BidirectionalIterator first,
                  BidirectionalIterator last );
template< class BidirectionalIterator, class Compare >
bool
next_permutation( BidirectionalIterator first,
                  BidirectionalIterator last, Compare comp );
```

next_permutation()取出由[first,last)标记的排列，并将其重新扫序为下一个排列（关于怎样确定上一个排列的讨论见 12.5.4 节）。如果不存在下一个排列，则返回 false。否则，返回 true。第一个版本使用底层类型的小于操作符来确定下一个排列，第二个版本根据 comp 对元素进行排序。如果原始字符串是排过序的，则连续调用 next_permutation()生成整个排列集合。例如，在下列程序中，如果我们不能把 musil 排序成 ilmsu，则不能生成排列的全集：

```
#include <algorithm>
#include <vector>
#include <iostream.h>

void print_char( char elem ) { cout << elem ; }
void (*ppc)( char ) = print_char;

/* 输出:
ilmsu ilmus ilsmu ilsum ilums ilusm imlsu imlus
imslu imsul imuls imusl islmu islum ismlu ismul
isulm isuml iulms iulsm iumls iumsl iuslm iusml
limsu limus lismu lisum liums liusm lmisu lmius
lmsiu lmsui lmuis lmusi lsimu lsium lsmiu lsmui
lsuim lsumi luims luism lumis lumsi lusim lusmi
```

```
milsu milus mislu misul miuls miusl mlisu mlius
mlsiu mlsui mluis mlusi msilu msiul msliu mslui
msuil msuli muils muisl mulis mulsi musil musli
silmu silum simlu simul siulm siuml slimu slium
slmiu slmui sluim slumi smilu smiul smliu smlui
smuil smuli suilm suiml sulim sulmi sumil sumli
uilms uilsm uimls uimsl uislm uisml ulims ulism
ulmis ulmsi ulsim ulsmi umils umisl umlis umlsi
umsil umsli usilm usiml uslim uslmi usmil usmli
*/
int main()
{
     vector<char,allocator> vec(5);

     // 字符顺序: musil
     vec[0] = 'm'; vec[1] = 'u'; vec[2] = 's';
     vec[3] = 'i'; vec[4] = 'l';
     int cnt = 2;
     sort( vec.begin(), vec.end() );
     for_each( vec.begin(), vec.end(), ppc ); cout << "\t";

     // 生成 "musil" 的所有排列组合
     while( next_permutation( vec.begin(), vec.end()))
     {
          for_each( vec.begin(), vec.end(), ppc );
          cout << "\t";
          if ( ! ( cnt++ % 8 )) {
               cout << "\n";
               cnt = 1;
          }
     }
     cout << "\n\n";
     return 0;
}
```

nnth_element()

```
template< class RandomAccessIterator >
void
nth_element( RandomAccessIterator first,
             RandomAccessIterator nth,
             RandomAccessIterator last );
template< class RandomAccessIterator, class Compare >
void
nth_element( RandomAccessIterator first,
             RandomAccessIterator nth,
             RandomAccessIterator last, Compare comp );
```

nnth_element()将[first,last)标记的序列重新排序，使所有小于第 n 个元素的元素都出现在它前面，而大于它的元素出现在它后面。例如，已知数组：

```
int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40 };
```

下面的 nth_element()调用使第七个元素为第 n 个（它的值是 26）：

```
nth_element( &ia[0], &ia[6], &ia[12] );
```

产生一个序列，其中小于 26 的七个元素在它的左边，余下大于 26 的四个元素在它的右边：{23,20,22,17,15,19,12,26,51,35,40,29}，但是，第 n 个元素两边的元素并不保证存在某种特定的顺序。第一个版本使用底层类型的小于操作符。第二个版本根据程序员传递的二元比较操作，对元素调整顺序：

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/*
* 输出：
original order of the vector: 29 23 20 22 17 15 26 51 19 12 35 40
sorting vector based on element 26
12 15 17 19 20 22 23 26 51 29 35 40
sorting vector in descending order based on element 23
40 35 29 51 26 23 22 20 19 17 15 12
*/

int main()
{
    int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40};
    vector< int,allocator > vec( ia, ia+12 );
    ostream_iterator<int> out( cout," " );

    cout << "original order of the vector: ";
    copy( vec.begin(), vec.end(), out ); cout << endl;
    cout << "sorting vector based on element "
         << *( vec.begin()+6 ) << endl;

    nth_element( vec.begin(), vec.begin()+6, vec.end() );
    copy( vec.begin(), vec.end(), out ); cout << endl;
    cout << "sorting vector in descending order "
         << "based on element "
         << *( vec.begin()+6 ) << endl;

    nth_element( vec.begin(), vec.begin()+6,
         vec.end(), greater<int>() );
    copy( vec.begin(), vec.end(), out ); cout << endl;
}
```

partial_sort()

```
template< class RandomAccessIterator >
void
partial_sort( RandomAccessIterator first,
              RandomAccessIterator middle,
```

```
                RandomAccessIterator last );
template< class RandomAccessIterator, class Compare >
void
partial_sort( RandomAccessIterator first,
              RandomAccessIterator middle,
              RandomAccessIterator last, Compare comp );
```

partial_sort()对整个序列作部分排序，被排序元素的个数正好可以被放到[first,middle)范围内。在[middle,last)中的元素是未经排序的，它们都落在实际被排序的序列之外。例如，已知数组：

```
int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40 };
```

调用 partial_sort()，使第六个元素为 middle：

```
stable_sort( &ia[0], &ia[5], &ia[12] );
```

则产生了一个序列，其中五个最小的元素被排序（即 middle-first 个元素）：{12,15,17,19,20,29,23,22,26,51,35,40}。从 middle 到 last-1 的元素并没有按任何特定的顺序，但是它们的值都落在实际被排序的序列之外。第一个版本用底层类型的小于操作符，第二个版本根据 comp 对元素进行排序：

partial_sort_copy()

```
template< class InputIterator, class RandomAccessIterator >
RandomAccessIterator
partial_sort_copy( InputIterator first, InputIterator last,
                   RandomAccessIterator result_first,
                   RandomAccessIterator result_last );
template< class InputIterator, class RandomAccessIterator,
          class Compare >
RandomAccessIterator
partial_sort_copy( InputIterator first, InputIterator last,
                   RandomAccessIterator result_first,
                   RandomAccessIterator result_last,
                   Compare comp );
```

partial_sort_copy()的行为与 partial_sort()相同，只不过它把经过部分排序的序列拷贝到由[result_first,result_last)标记的容器中。因此，如果我们指定了一个独立的容器去接受拷贝，则结果是一个完全排序的序列。例如，已知两个数组：

```
int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40 };
int ia2[5];
```

指定第八个元素为 middle 的 partial_sort_copy()调用：

```
stable_sort( &ia[0], &ia[7], &ia[12],
     &ia2[0], &ia[5] );
```

用五个排过序的元素填充 ia2：{12,15,17,19,20}，而另外两个排过序的元素没有被使用。

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/*
```

```
* 输出：
original order of vector: 69 23 80 42 17 15 26 51 19 12 35 8
partial sort of vector: seven elements
8 12 15 17 19 23 26 80 69 51 42 35
partial_sort_copy() of first seven elements
of vector in descending order
26 23 19 17 15 12 8
*/

int main()
{
     int ia[] = {69,23,80,42,17,15,26,51,19,12,35,8 };
     vector< int,allocator > vec( ia, ia+12 );
     ostream_iterator<int> out( cout," " );
     cout << "original order of vector: ";
     copy( vec.begin(), vec.end(), out ); cout << endl;
     cout << "partial sort of vector: seven elements\n";
     partial_sort( vec.begin(), vec.begin()+7, vec.end() );
     copy( vec.begin(), vec.end(), out ); cout << endl;
     vector< int, allocator > res(7);
     cout << "partial_sort_copy() of first seven elements\n\t"
          << "of vector in descending order\n";
     partial_sort_copy( vec.begin(), vec.begin()+7, res.begin(),
          res.end(), greater<int>() );
     copy( res.begin(), res.end(), out ); cout << endl;
}
```

partial_sum()

```
template < class InputIterator, Class OutputIterator >
OutputIterator
partial_sum(
             InputIterator first, InputIterator last,
             OutputIterator result );
template < class InputIterator, Class OutputIterator,
           class BinaryOperation >
OutputIterator
partial_sum(
             InputIterator first, InputIterator last,
             OutputIterator result, BinaryOperation op );
```

partial_sum()的第一个版本创建一个新的元素序列，其中每个新元素的值代表了[first,last)序列中这位置之前（包括该位置）所有元素的和。例如，已知序列{0,1,1,2,3, 5,8}。则新序列是{0,1,2,4,7,12,20}。例如，第四个元素是前三个值{0,1,1}的部分和加上它自己(2)，产生值 4。

第二个版本使用程序员传递的二元操作。例如，已知序列{1,2,3,4}，我们传递函数对象times<int>。结果序列是{1,2,6,24}。在两个版本中，OutputIterator 指向新序列末元素的下一个位置。

partial_sum()是一个算术算法，我们必须包含标准头文件<numeric>：

```
#include <numeric>
#include <vector>
#include <iostream.h>

/*
* 输出：
elements: 1 3 4 5 7 8 9
partial sum of elements:
1 4 8 13 20 28 37
partial sum of elements using times<int>():
1 3 12 60 420 3360 30240
*/

int main()
{
     const int ia_size = 7;
     int ia[ ia_size ] = { 1, 3, 4, 5, 7, 8, 9 };
     int ia_res[ ia_size ];
     ostream_iterator< int > outfile( cout, " " );
     vector< int, allocator > vec( ia, ia+ia_size );
     vector< int, allocator > vec_res( vec.size() );
     cout << "elements: ";
     copy( ia, ia+ia_size, outfile ); cout << endl;
     cout << "partial sum of elements:\n";
     partial_sum( ia, ia+ia_size, ia_res );
     copy( ia_res, ia_res+ia_size, outfile ); cout << endl;
     cout << "partial sum of elements using times<int>():\n";
     partial_sum( vec.begin(), vec.end(), vec_res.begin(),
          times<int>() );
     copy( vec_res.begin(), vec_res.end(), outfile );
     cout << endl;
}
partition()
template < class BidirectionalIterator, class UnaryPredicate >
BidirectionalIterator
partition( BidirectionalIterator first,
     BidirectionalIterator last, UnaryPredicate pred );
```

partition()对[first,last)范围内的元素重新排序。当向它传递一个一元谓词操作 pred 时，所有计算结果为 true 的元素都被放在计算结果为 false 的元素前面。例如，已知序列{0,1,2,3,4,5,6}，以及一个“测试元素是否为偶数”的一元谓词操作，则 true 和 false 的元素范围分别是{0,2,4,6}和{1,3,5}。虽然所有的偶元素保证放在奇元素的前面，但是，结果序列并不保证保留元素的相对位置。即，4 可能放在 2 的前面，或者 5 放在 3 之前。后面讨论的 stable_partition()会保证保留容器内元素的相对顺序：

这是一页的末尾

```
#include <algorithm>
#include <vector>
#include <iostream.h>
```

```
class even_elem {
public:
     bool operator()( int elem )
          { return elem%2 ? false : true; }
};

/*
* 输出:
     original order of elements:
     29 23 20 22 17 15 26 51 19 12 35 40
     partition based on whether element is even:
     40 12 20 22 26 15 17 51 19 23 35 29
     partition based on whether element is less than 25:
     12 23 20 22 17 15 19 51 26 29 35 40
*/

int main()
{
     const int ia_size = 12;
     int ia[ia_size] = { 29,23,20,22,17,15,26,51,19,12,35,40 };
     vector< int, allocator > vec( ia, ia+ia_size );
     ostream_iterator< int > outfile( cout, " " );

     cout << "original order of elements: \n";
     copy( vec.begin(), vec.end(), outfile ); cout << endl;

     cout << "partition based on whether element is even:\n";
     partition( &ia[0], &ia[ia_size], even_elem() );
     copy( ia, ia+ia_size, outfile ); cout << endl;

     cout << "partition based on whether element "
          << "is less than 25:\n";

     partition( vec.begin(), vec.end(), bind2nd(less<int>(),25) );
     copy( vec.begin(), vec.end(), outfile ); cout << endl;
}
```

prev_permutation()

```
template < class BidirectionalIterator >
bool
prev_permutation( BidirectionalIterator first,
                  BidirectionalIterator last );
template < class BidirectionalIterator, class Compare >
bool
prev_permutation( BidirectionalIterator first,
                  BidirectionalIterator last, Compare comp );
```

prev_permutation 取出由[first,last)标记的排列，并将它重新排序为上一个排列（关于怎样判断上一个排列的讨论见 12.5.4 节）。如果不存在上一个排列，则返回 false。否则，返回 true。

第一个版本使用底层类型的小于操作符，来确定上一个排列，第二个版本根据程序员传递的二元比较操作，对元素进行排序：

```
#include <algorithm>
#include <vector>
#include <iostream.h>

// 输出：
// n d an a d  d n a  d a n  a n d  a d n

int main()
{
    vector< char, allocator > vec( 3 );
    ostream_iterator< char > out_stream( cout, " " );
    vec[0] = 'n'; vec[1] = 'd'; vec[2] = 'a';
    copy( vec.begin(), vec.end(), out_stream ); cout << "\t";

    // 生成 "dan" 的所有排列
    while( prev_permutation( vec.begin(), vec.end() )) {
        copy( vec.begin(), vec.end(), out_stream );
        cout << "\t";
    }

    cout << "\n\n";
}
```

random_shuffle()

```
template< class RandomAccessIterator >
void
random_shuffle( RandomAccessIterator first,
                RandomAccessIterator last );
template< class RandomAccessIterator,
          class RandomNumberGenerator >
void
random_shuffle( RandomAccessIterator first,
                RandomAccessIterator last,
                RandomNumberGenerator rand );
```

random_shuffle()对[first,last)范围内的元素随机调整顺序。第二个版本使用一个专门产生随机数的函数对象或函数指针。rand 返回一个 double 类型的。位于区间[0,1]内的值。

```
#include <algorithm>
#include <vector>
#include <iostream.h>
int main()
{
    vector< int, allocator > vec;
    for ( int ix = 0; ix < 20; ix++ )
        vec.push_back( ix );

    random_shuffle( vec.begin(), vec.end() );
```

```
    // 输出：
    // random_shuffle of sequence of values 1 .. 20:
    // 6 11 9 2 18 12 17 7 0 15 4 8 10 5 1 19 13 3 14 16
    cout << "random_shuffle of sequence of values 1 .. 20:\n";
    copy( vec.begin(), vec.end(),
    ostream_iterator< int >( cout," " ));
}
```

remove()

```
template< class ForwardIterator, class Type >
ForwardIterator
remove( ForwardIterator first,
        ForwardIterator last, const Type &value );
```

remove()删除在[first,last)范围内的所有 value 实例。remove()以及 remove_if()并不真正地把匹配到的元素从容器中清除（即容器的大小保留不变），而是每个不匹配的元素依次被赋值给从 first 开始的下一个空闲位置上，返回的 ForwardIterator 标记了新的元素范围的下一个位置。例如，考虑序列{0,1,0,2,0,3,0,4}。假设我们希望删除所有的 0，则结果序列是{1,2,3,4,0,3,0,4}。1 被拷贝到第一个位置上，2 被拷贝到第二个位置上，3 被拷贝到第三个位置上，4 被拷贝到第四个位置上。返回的 ForwardIterator 指向第五个位置上的 0。典型的做法是，该 iterator 接着被传递给 erase()，以便删除无效的元素。（内置数组不适合于使用 remove()和 remove_if()算法，因为它们不能很容易地被改变大小。由于这个原因，对于数组而言，remove_copy()和 remove_copy_if()是更受欢迎的算法。

remove_copy()

```
template< class InputIterator, class OutputIterator,
          class Type >
OutputIterator
remove_copy( InputIterator first, InputIterator last,
             OutputIterator result, const Type &value );
```

remove_copy()把所有不匹配的元素都拷贝到由 result 指定的容器中。返回的 OutputIterator 指向被拷贝的末元素的下一个位置，但原始容器没有被改变：

```
#include <algorithm>
#include <vector>
#include <iostream.h>
/* 输出：
   original vector sequence:
   0 1 0 2 0 3 0 4 0 5
   vector after remove, without applying erase():
   1 2 3 4 5 3 0 4 0 5
   vector after erase():
   1 2 3 4 5
   array after remove_copy():
   1 2 3 4 5
*/
```

```
int main()
{
    int value = 0;
    int ia[] = { 0, 1, 0, 2, 0, 3, 0, 4, 0, 5 };
    vector< int, allocator > vec( ia, ia+10 );
    ostream_iterator< int > ofile( cout," ");
    vector< int, allocator >::iterator vec_iter;

    cout << "original vector sequence:\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';
    vec_iter = remove( vec.begin(), vec.end(), value );
    cout << "vector after remove, without applying erase():\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';

    // erase the invalid elements from container
    vec.erase( vec_iter, vec.end() );
    cout << "vector after erase():\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';

    int ia2[5];
    vector< int, allocator > vec2( ia, ia+10 );
    remove_copy( vec2.begin(), vec2.end(), ia2, value );

    cout << "array after remove_copy():\n";
    copy( ia2, ia2+5 ofile ); cout << endl;
}
```

remove_if()

```
template< class ForwardIterator, class Predicate >
ForwardIterator
remove_if( ForwardIterator first,
           ForwardIterator last, Predicate pred );
```

remove_if()删除所有在[first,last)范围内、并且 pred 计算结果为 true 的元素。remove_if()以及 remove()并不真正地把匹配到的元素从容器中清除，而是将每个不匹配的元素依次赋值给从 first 开始的下一个空闲位置上。返回的 ForwardIterator 标记了新的元素范围的下一个位置。一般是将这个 iterator 传递给 erase()，以便真正地删除掉无效的元素。（remove_copy_if()更加适用于内置数组。）

remove_copy_if()

```
template< class InputIterator, class OutputIterator,
    class Predicate >
OutputIterator
remove_copy_if( InputIterator first, InputIterator last,
    OutputIterator result, Predicate pred );
```

remove_copy_if()把所有不匹配的元素拷贝到由 result 指定的容器中。返回的 OutputIterator

标记了被拷贝的末元素的下一个位置，原始容器没有被改变：

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/* 输出：
   original element sequence:
   0 1 1 2 3 5 8 13 21 34
   sequence after applying remove_if < 10:
   13 21 34
   sequence after applying remove_copy_if even:
   1 1 3 5 13 21
*/
class EvenValue {
public:
     bool operator()( int value ) {
          return value % 2 ? false : true; }
};
int main()
{
     int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };

     vector< int, allocator >::iterator iter;
     vector< int, allocator > vec( ia, ia+10 );
     ostream_iterator< int > ofile( cout, " " );
     cout << "original element sequence:\n";
     copy( vec.begin(), vec.end(), ofile ); cout << '\n';

     iter = remove_if( vec.begin(), vec.end(),
          bind2nd(less<int>(),10) );
     vec.erase( iter, vec.end() );

     cout << "sequence after applying remove_if < 10:\n";
     copy( vec.begin(), vec.end(), ofile ); cout << '\n';
     vector< int, allocator > vec_res( 10 );
     iter = remove_copy_if( ia, ia+10,
          vec_res.begin(), EvenValue() );

     cout << "sequence after applying remove_copy_if even:\n";
     copy( vec_res.begin(), iter, ofile ); cout << '\n';
}
```

replace()

```
template< class ForwardIterator, class Type >
void
replace( ForwardIterator first, ForwardIterator last,
const Type& old_value, const Type& new_value );
```

replace()将[first,last)范围内的所有 old_value 实例都用 new_value 替代。

replace_copy()

```
template< class InputIterator, class OutputIterator,
          class Type >
OutputIterator
replace_copy( InputIterator first, InputIterator last,
              OutputIterator result,
              const Type& old_value, const Type& new_value );
```

replace_copy()的行为与 replace()类似，只不过是把新序列拷贝到由 result 开始的容器内。返回的 OutputIterator 指向被拷贝的末元素的下一个位置，但原始序列没有被改变。

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/* 输出:
   original element sequence:
   Christopher Robin Mr. Winnie the Pooh Piglet Tigger Eeyore
   sequence after applying replace():
   Christopher Robin Pooh Piglet Tigger Eeyore
   sequence after applying replace_copy():
   Christopher Robin Mr. Winnie the Pooh Piglet Tigger Eeyore
*/
int main()
{
     string oldval( "Mr. Winnie the Pooh" );
     string newval( "Pooh" );

     ostream_iterator< string > ofile( cout, " " );
     string sa[] = {
          "Christopher Robin", "Mr. Winnie the Pooh",
          "Piglet", "Tigger", "Eeyore"
     };
     vector< string, allocator > vec( sa, sa+5 );
     cout << "original element sequence:\n";
     copy( vec.begin(), vec.end(), ofile ); cout << '\n';

     replace( vec.begin(), vec.end(), oldval, newval );
     cout << "sequence after applying replace():\n";
     copy( vec.begin(), vec.end(), ofile ); cout << '\n';

     vector< string, allocator > vec2;
     replace_copy( vec.begin(), vec.end(),
          inserter( vec2, vec2.begin() ),

     newval, oldval );
     cout << "sequence after applying replace_copy():\n";
     copy( vec2.begin(), vec2.end(), ofile ); cout << '\n';
}
```

replace_if()

```
template< class ForwardIterator, class Predicate, class Type >
void
replace_if( ForwardIterator first, ForwardIterator last,
            Predicate pred, const Type& new_value );
```

replace_if()将[first,last)范围内的、pred 计算结果为 true 的所有元素，都用 new_value 替代。

replace_copy_if()

```
template< class ForwardIterator, class OutputIterator,
          class Predicate, class Type >
OutputIterator
replace_copy_if( ForwardIterator first, ForwardIterator last,
                 OutputIterator result,
                 Predicate pred, const Type& new_value );
```

replace_copy_if()的行为与 replace_if()类似，只不过是把新序列拷贝到由 result 开始的容器中。返回的 OutputIterator 指向被拷贝的末元素的下一个位置，原始序列没有被改变。

```
#include <algorithm>
#include <vector>
#include <iostream.h>
/*
* 输出:
  original element sequence:
  0 1 1 2 3 5 8 13 21 34
  sequence after applying replace_if < 10 with 0:
  0 0 0 0 0 0 0 13 21 34
  sequence after applying replace_if even with 0:
  0 1 1 0 3 5 0 13 21 0
*/
class EvenValue {
public:
     bool operator()( int value ) {
          return value % 2 ? false : true; }
};
int main()
{
     int new_value = 0;
     int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
     vector< int, allocator > vec( ia, ia+10 );
     ostream_iterator< int > ofile( cout, " " );

     cout << "original element sequence:\n";
     copy( ia, ia+10, ofile ); cout << '\n';

     replace_if( &ia[0], &ia[10],
```

```
            bind2nd(less<int>(),10), new_value );
        cout << "sequence after applying replace_if < 10 with 0:\n";
        copy( ia, ia+10, ofile ); cout << '\n';
        replace_if( vec.begin(), vec.end(),
            EvenValue(), new_value );
        cout << "sequence after applying replace_if even with 0:\n";
        copy( vec.begin(), vec.end(), ofile ); cout << '\n';
}
```

reverse()

```
template< class BidirectionalIterator >
void
reverse( BidirectionalIterator first,
         BidirectionalIterator last );
```

reverse()对于容器中[first,last)范围内的元素重新按反序排列。例如，已知序列{0,1,1,2,3}，则反序序列是{3,2,1,1,0}。

reverse_copy()

```
template< class BidirectionalIterator, class OutputIterator >
OutputIterator
reverse_copy( BidirectionalIterator first,
              BidirectionalIterator last, OutputIterator result );
```

reverse_copy()的行为与 reverse()类似，只不过把新序列拷贝到由 result 开始的容器中。返回的 OutputIterator 指向被拷贝的元素的下一个位置。原始序列没有被改变。

```
#include <algorithm>
#include <list>
#include <string>
#include <iostream.h>

/*
* 输出:
  Original sequence of strings:
  Signature of all things I am here to
  read seaspawn and seawrack that rusty boot
  Sequence after reverse() applied:
  boot rusty that seawrack and seaspawn read to
  here am I things all of Signature
*/

class print_elements {
public:
     void operator()( string elem ) {
          cout << elem
               << ( _line_cnt++%8 ? " " : "\n\t" );
```

```
        }
        static void reset_line_cnt() { _line_cnt = 1; }
    private:
        static int _line_cnt;
    };

    int print_elements::_line_cnt = 1;

    int main()
    {
        string sa[] = { "Signature", "of", "all", "things",
            "I", "am", "here", "to", "read",
            "seaspawn", "and", "seawrack", "that",
            "rusty", "boot"
        };
        list< string, allocator > slist( sa, sa+15 );

        cout << "Original sequence of strings:\n\t";
        for_each( slist.begin(), slist.end(), print_elements() );
        cout << "\n\n";

        reverse( slist.begin(), slist.end() );
        print_elements::reset_line_cnt();

        cout << "Sequence after reverse() applied:\n\t";
        for_each( slist.begin(), slist.end(), print_elements() );
        cout << "\n";

        list< string, allocator > slist_copy( slist.size() );
        reverse_copy( slist.begin(), slist.end(),
            slist_copy.begin() );
    }
```

rotate()

```
template< class ForwardIterator >
void
rotate( ForwardIterator first,
        ForwardIterator middle, ForwardIterator last );
```

rotate()把[first,middle)范围内的元素移到容器末尾，由 middle 指向的元素成为容器的第一个元素。例如，已知单词“hissboo”，则以元素“b”为轴的旋转将单词变成“boohiss”。

rotate_copy()

```
template< class ForwardIterator, class OutputIterator >
OutputIterator
rotate_copy( ForwardIterator first, ForwardIterator middle,
             ForwardIterator last, OutputIterator result );
```

rotate_copy()的行为与 rotate()类似，只不过把旋转后的序列拷贝到由 result 标记的容器中。返回的 OutputIterator 指向被拷贝的末元素的下一个位置。原始序列没有被改变。

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/*
* 输出:
  original element sequence:
  1 3 5 7 9 0 2 4 6 8 10
  rotate on middle element(0) ::
  0 2 4 6 8 10 1 3 5 7 9
  rotate on next to last element(8) ::
  8 10 1 3 5 7 9 0 2 4 6
  rotate_copy on middle element ::
  7 9 0 2 4 6 8 10 1 3 5
*/

int main()
{
     int ia[] = { 1, 3, 5, 7, 9, 0, 2, 4, 6, 8, 10 };

     vector< int, allocator > vec( ia, ia+11 );
     ostream_iterator< int > ofile( cout, " " );
     cout << "original element sequence:\n";

     copy( vec.begin(), vec.end(), ofile ); cout << '\n';
     rotate( &ia[0], &ia[5], &ia[11] );
     cout << "rotate on middle element(0) ::\n";

     copy( ia, ia+11, ofile ); cout << '\n';
     rotate( vec.begin(), vec.end()-2, vec.end() );
     cout << "rotate on next to last element(8) ::\n";

     copy( vec.begin(), vec.end(), ofile ); cout << '\n';
     vector< int, allocator > vec_res( vec.size() );
     rotate_copy( vec.begin(), vec.begin()+vec.size()/2,
                  vec.end(), vec_res.begin() );

     cout << "rotate_copy on middle element ::\n";
     copy( vec_res.begin(), vec_res.end(), ofile );
     cout << '\n';
}
```

search()

```
template< class ForwardIterator1, class ForwardIterator2 >
ForwardIterator
search( ForwardIterator1 first1, ForwardIterator1 last1,
```

```
        ForwardIterator2 first2, ForwardIterator2 last2 );
template< class ForwardIterator1, class ForwardIterator2,
          class BinaryPredicate >
ForwardIterator
search( ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2,
        BinaryPredicate pred );
```

给出了两个范围，search()返回一个 iterator，指向在[first1,last1)范围内第一次出现子序列[first2,last2]的位置。如果子序列未出现，则返回 last1。例如，在 mississippi 中，子序列 iss 出现两次，则 search()返回一个 iterator，指向第一个实例的起始处。缺省情况下，使用等于操作符进行元素的比较，第二个版本允许用户提供一个比较操作：

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/*
* 输出：
  Expecting to find the substring 'ate': a t e
  Expecting to find the substring 'vat': v a t
*/

int main()
{
     ostream_iterator< char > ofile( cout, " " );

     char str[ 25 ] = "a fine and private place";
     char substr[] = "ate";

     char *found_str = search(str,str+25,substr,substr+3);
     cout << "Expecting to find the substring 'ate': ";
     copy( found_str, found_str+3, ofile ); cout << '\n';

     vector< char, allocator > vec( str, str+24 );
     vector< char, allocator > subvec(3);

     subvec[0]='v'; subvec[1]='a'; subvec[2]='t';
     vector< char, allocator >::iterator iter;
     iter = search( vec.begin(), vec.end(),
          subvec.begin(), subvec.end(),

     equal_to< char >() );

     cout << "Expecting to find the substring 'vat': ";
     copy( iter, iter+3, ofile ); cout << '\n';
}
```

search_n()

```
template< class ForwardIterator, class Size, class Type >
```

```
ForwardIterator
search_n( ForwardIterator first, ForwardIterator last,
          Size count, const Type &value );
template< class ForwardIterator, class Size,
          class Type, class BinaryPredicate >
ForwardIterator
search_n( ForwardIterator first, ForwardIterator last,
          Size count, const Type &value, BinaryPredicate pred );
```

search_n()在[first,last)序列中查找“value 出现 count 次”的子序列。如果没有找到“value 的 count 次出现”，则返回 last。例如，为了在序列 mississippi 中找到了序列 ss，value 将被设置为“s”，而 count 为 2。为了找到子串“ssi”的两个实例，value 应该为“ssi”，而 count 仍是 2。search_n()返回一个 iterator，指向被找到的 value 的第一个元素。缺省情况下，使用等于操作符来比较元素，第二版本允许用户提供一个比较操作。

```
#include <algorithm>
#include <vector>
#include <iostream.h>
/*
* 输出：
  Expecting to find two instances of 'o': o o
  Expecting to find the substring 'mou': m o u
*/
int main()
{
     ostream_iterator< char > ofile( cout, " " );

     const char blank = ' ';
     const char oh = 'o';

     char str[ 26 ] = "oh my a mouse ate a moose";
     char *found_str = search_n( str, str+25, 2, oh );

     cout << "Expecting to find two instances of 'o': ";
     copy( found_str, found_str+2, ofile ); cout << '\n';

     vector< char, allocator > vec( str, str+25 );

     // 寻找第一个这样的序列
     // 其中三个字符都不是空格：mouse 中的 mou
     vector< char, allocator >::iterator iter;
     iter = search_n( vec.begin(), vec.end(), 3,
         blank, not_equal_to< char >() );
     cout << "Expecting to find the substring 'mou': ";
     copy( iter, iter+3, ofile ); cout << '\n';
}
```

set_difference()

```
template < class InputIterator1, class InputIterator2,
```

```
            class OutputIterator >
OutputIterator
set_difference( InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2,
                OutputIterator result );
template < class InputIterator1, class InputIterator2,
           class OutputIterator, class Compare >
OutputIterator
set_difference( InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2,
                OutputIterator result, Compare comp );
```

set_difference()构造一个排过序的序列，其中的元素出现在第一个序列中（由[first,last)标记），但是不包含在第二个序列中（由[first2,last2]标记）。例如，已失两个序列{0,1,2,3}和{0,2,4,6}，则差集为{1,3}。返回的 OutputIterator 指向被放入 result 所标记的容器中的最后元素的下一个位置。第一个版本假设该序列是用底层元素类型的小于操作符来排序的，第二个版本假设该序列是用 comp 来排序的。

set_intersection()

```
template < class InputIterator1, class InputIterator2,
           class OutputIterator >
OutputIterator
set_intersection( InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, InputIterator2 last2,
                  OutputIterator result );
template < class InputIterator1, class InputIterator2,
           class OutputIterator, class Compare >
OutputIterator
set_intersection( InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, InputIterator2 last2,
                  OutputIterator result, Compare comp );
```

set_intersection()构造一个排过序的序列，其中的元素在[first1,last1)和[first2,last2)序列中都存在。例如，已知序列{0,1,2,3}和{0,2,4,6}，则交集为{0,2}。这些元素被从第一个序列中拷贝出来。返回的 OutputIterator 指向被放入 result 所标记的容器内的最后元素的下一个位置。第一个版本假设该序列是用底层类型的小于操作符来排序的，而第二个版本假设该序列是用 comp 来排序的。

set_symmetric_difference()

```
template < class InputIterator1, class InputIterator2,
           class OutputIterator >
OutputIterator
set_symmetric_difference(
           InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, InputIterator2 last2,
           OutputIterator result );
```

```
template < class InputIterator1, class InputIterator2,
           class OutputIterator, class Compare >
OutputIterator
set_symmetric_difference(
           InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, InputIterator2 last2,
           OutputIterator result, Compare comp );
```

set_symmetric_difference()构造一个排过序的序列，其中的元素在第一个序列中出现、但不出现在第二个序列中，或者在第二个序列中出现、但不出现在第一个序列中。例如，已知两个序列{0,1,2,3}和{0,2,4,6}，则对称差集是{1,3,4,6}。返回的 OutputIterator 指向被放入 result 所标记的容器内的最后元素的下一个位置。第一个版本假设该序列是用底层类型的小于操作符来排序的，而第二个版本假设该序列是用 comp 来排序的。

set_union()

```
template < class InputIterator1, class InputIterator2,
           class OutputIterator >
OutputIterator
set_union( InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, InputIterator2 last2,
           OutputIterator result );
template < class InputIterator1, class InputIterator2,
           class OutputIterator, class Compare >
OutputIterator
set_union( InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, InputIterator2 last2,
           OutputIterator result, Compare comp );
```

set_union()构造一个排过序的序列，它包含了[first1,last1)和[first2,last2)这两个范围内的所有元素。例如，已知两个序列{0,1,2,3}和{0,2,4,6}，则并集为{0,1,2,3,4,6}，如果一个元素在两个容器中都存在，比如 0 和 2，则拷贝第一个容器中的元素。返回的 OutputIterator 指向被放入 result 所标记的容器内的最后元素的下一个位置。第一个版本假设该序列是用底层类型的小于操作符来排序的，而第二个版本假设该序列是用 comp 来排序的。

```
#include <algorithm>
#include <set>
#include <string>
#include <iostream.h>

/*
* 输出:
  set #1 elements:
         Eeyore Piglet Pooh Tigger
  set #2 elements:
         Heffalump Pooh Woozles
  set_union() elements:
         Eeyore Heffalump Piglet Pooh Tigger Woozles
  set_intersection() elements:
         Pooh
```

```
  set_difference() elements:
        Eeyore Piglet Tigger
  set_symmetric_difference() elements:
        Eeyore Heffalump Piglet Tigger Woozles
*/
int main()
{
    string str1[] = { "Pooh", "Piglet", "Tigger", "Eeyore" };
    string str2[] = { "Pooh", "Heffalump", "Woozles" };
    ostream_iterator< string > ofile( cout, " " );

    set<string,less<string>,allocator> set1( str1, str1+4 );
    set<string,less<string>,allocator> set2( str2, str2+3 );

    cout << "set #1 elements:\n\t";
    copy( set1.begin(), set1.end(), ofile ); cout << "\n\n";
    cout << "set #2 elements:\n\t";
    copy( set2.begin(), set2.end(), ofile ); cout << "\n\n";
    set<string,less<string>,allocator> res;
    set_union( set1.begin(), set1.end(),
        set2.begin(), set2.end(),
        inserter( res, res.begin() ));

    cout << "set_union() elements:\n\t";
    copy( res.begin(), res.end(), ofile ); cout << "\n\n";
    res.clear();
    set_intersection( set1.begin(), set1.end(),
        set2.begin(), set2.end(),
        inserter( res, res.begin() ));

    cout << "set_intersection() elements:\n\t";
    copy( res.begin(), res.end(), ofile ); cout << "\n\n";
    res.clear();
    set_difference( set1.begin(), set1.end(),
        set2.begin(), set2.end(),
        inserter( res, res.begin() ));

    cout << "set_difference() elements:\n\t";
    copy( res.begin(), res.end(), ofile ); cout << "\n\n";
    res.clear();
    set_symmetric_difference( set1.begin(), set1.end(),
        set2.begin(), set2.end(),
        inserter( res, res.begin() ));
    cout << "set_symmetric_difference() elements:\n\t";
    copy( res.begin(), res.end(), ofile ); cout << "\n\n";
}
```

sort()

```
template< class RandomAccessIterator >
```

```
void
sort( RandomAccessIterator first,
      RandomAccessIterator last );
template< class RandomAccessIterator, class Compare >
void
sort( RandomAccessIterator first,
      RandomAccessIterator last, Compare comp );
```

sort()利用底层元素的小于操作符，以升序重新排列[first,last)范围内的元素。第二版本根据 comp 对元素进行排序（为了保留相等元素之间的顺序关系，要使用 stable_sort()，而不是 sort()）。我们不提供专门的程序来说明 sort()的用法，因为它在许多其他的例子中会被用到，比如 binary_search()、equal_range()和 inplace_merge()。

stable_partition()

```
template< class BidirectionalIterator, class Predicate >
BidirectionalIterator
stable_partition( BidirectionalIterator first,
                  BidirectionalIterator last,
                  Predicate pred );
```

stable_partition()的行为与 partition()类似，只不过它保证会保留容器中元素的相对顺序。下面是与 partition()的例子相同的一个程序，但是它被修改为调用 stable_partition()。

```
#include <algorithm>
#include <vector>
#include <iostream.h>
/*
* generates:
  original element sequence:
  29 23 20 22 17 15 26 51 19 12 35 40
  stable_partition on even element:
  20 22 26 12 40 29 23 17 15 51 19
  stable_partition of less than 25:
  23 20 22 17 15 19 12 29 26 51 35 40
*/
class even_elem {
public:
     bool operator()( int elem ) {
          return elem%2 ? false : true;
     }
};
int main()
{
     int ia[] = { 29,23,20,22,17,15,26,51,19,12,35,40 };

     vector< int, allocator > vec( ia, ia+12 );
     ostream_iterator< int > ofile( cout, " " );

     cout << "original element sequence:\n";
     copy( vec.begin(), vec.end(), ofile ); cout << '\n';
```

```
    stable_partition( &ia[0], &ia[12], even_elem() );
    cout << "stable_partition on even element:\n";
    copy( ia, ia+11, ofile ); cout << '\n';
    stable_partition( vec.begin(), vec.end(),
        bind2nd(less<int>(),25) );
    cout << "stable_partition of less than 25:\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';
}
```

stable_sort()

```
template< class RandomAccessIterator >
void
stable_sort( RandomAccessIterator first,
             RandomAccessIterator last );
template< class RandomAccessIterator, class Compare >
void
stable_sort( RandomAccessIterator first,
             RandomAccessIterator last, Compare comp );
```

stable_sort()利用底层类型的小于操作符，以升序重新排列[first,last)范围内的元素，并且保留相等元素之间的顺序关系。第二版本根据 comp 对元素进行排序。

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/*
* 输出:
  original element sequence:
  29 23 20 22 12 17 15 26 51 19 12 23 35 40
  stable sort -- default ascending order:
  12 12 15 17 19 20 22 23 23 26 29 35 40 51
  stable sort: descending order:
  51 40 35 29 26 23 23 22 20 19 17 15 12 12
*/
int main()
{
    int ia[] = { 29,23,20,22,12,17,15,26,51,19,12,23,35,40 };
    vector< int, allocator > vec( ia, ia+14 );
    ostream_iterator< int > ofile( cout, " " );

    cout << "original element sequence:\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';

    stable_sort( &ia[0], &ia[14] );

    cout << "stable sort -- default ascending order:\n";
    copy( ia, ia+14, ofile ); cout << '\n';

    stable_sort( vec.begin(), vec.end(), greater<int>() );
```

```
    cout << "stable sort: descending order:\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';
}
```

swap()

```
template< class Type >
void
swap ( Type &ob1, Type &ob2 );
```

swap()交换存贮在对象 ob1 和 ob2 中的值。

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/*
* 输出：
  original element sequence:
  3 4 5 0 1 2
  sequence applying swap() to support bubble sort:
  0 1 2 3 4 5
*/

int main()
{
    int ia[] = { 3, 4, 5, 0, 1, 2 };
    vector< int, allocator > vec( ia, ia+6 );
    for ( int ix = 0; ix < 6; ++ix )
        for ( int iy = ix; iy < 6; ++iy ) {
            if ( vec[iy] < vec[ ix ] )
                swap( vec[iy], vec[ix] );
        }

    ostream_iterator< int > ofile( cout, " " );
    cout << "original element sequence:\n";

    copy( ia, ia+6, ofile ); cout << '\n';
    cout << "sequence applying swap() "
        << "to support bubble sort:\n";

    copy( vec.begin(), vec.end(), ofile ); cout << '\n';
}
```

swap_range()

```
template <class ForwardIterator1, class ForwardIterator2 >
ForwardIterator2
swap_range( ForwardIterator1 first1, ForwardIterator1 last,
            ForwardIterator2 first2 );
```

swap_range()将[first,last)标记的元素值与“从 first2 开始、相同个数”的元素值进行交

换。这两个序列可以是同一容器中不相连的序列，也可以位于两个独立的容器中。如果从 first2 开始的序列小于由[first1,last)标记的序列，或者两个序列在同一容器中有重叠，则该算法的运行时刻行为是未定义的。swap_range()返回第二个序列的 iterator，指向最后一个被交换的元素的下一个位置。

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/*
* 输出：
  original element sequence of first container:
  0 1 2 3 4 5 6 7 8 9
  original element sequence of second container:
  5 6 7 8 9
  array after swap_ranges() in middle of array:
  5 6 7 8 9 0 1 2 3 4
  first container after swap_ranges() of two vectors:
  5 6 7 8 9 5 6 7 8 9
  second container after swap_ranges() of two vectors:
  0 1 2 3 4
*/

int main()
{
     int ia[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
     int ia2[] = { 5, 6, 7, 8, 9 };

     vector< int, allocator > vec( ia, ia+10 );
     vector< int, allocator > vec2( ia2, ia2+5 );

     ostream_iterator< int > ofile( cout, " " );
     cout << "original element sequence of first container:\n";

     copy( vec.begin(), vec.end(), ofile ); cout << '\n';
     cout << "original element sequence of second container:\n";

     copy( vec2.begin(), vec2.end(), ofile ); cout << '\n';

     // 在同一序列中进行交换
     swap_ranges( &ia[0], &ia[5], &ia[5] );
     cout << "array after swap_ranges() in middle of array:\n";
     copy( ia, ia+10, ofile ); cout << '\n';

     // 跨容器交换
     vector< int, allocator >::iterator last =
          find( vec.begin(), vec.end(), 5 );
     swap_ranges( vec.begin(), last, vec2.begin() );
     cout << "first container after "
          << "swap_ranges() of two vectors:\n";
```

```
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';

    cout << "second container after "
         << "swap_ranges() of two vectors:\n";

    copy( vec2.begin(), vec2.end(), ofile ); cout << '\n';
}
```

transform()

```
template< class InputIterator, class OutputIterator,
          class UnaryOperation >
OutputIterator
transform( InputIterator first, InputIterator last,
           OutputIterator result, UnaryOperation op );
template< class InputIterator1, class InputIterator2,
          class OutputIterator, class BinaryOperation >
OutputIterator
transform( InputIterator1 first1, InputIterator1 last,
           InputIterator2 first2, OutputIterator result,
           BinaryOperation bop );
```

transform()的第一个版本将 op 作用在[first,last)范围内的每个元素上，从而产生一个新的序列。例如，已知序列{0,1,1,2,3,5}和函数对象 Double（它使每个元素加倍），那么，结果序列是{0,2,2,4,6,10}。

第二个版本将 bop 作用在一对元素上，其中一个元素来自序列[first1,last)，另一个来自由 first2 开始的序列，最终产生一个新的序列。如果第二个序列包含的元素少于第一个序列，则运行时刻行为是未定义的。例如，已知序列{1,3,5,9}和{2,4,6,8}，以及函数对象 AddAndDouble（它把两个元素相加，并将和加倍），则结果序列是{6,14,22,34}。

两个版本的 transform()都把结果序列放在由 result 标记的容器中。result 可以指向两个输入容器之一，则实际达到的效果是，用 transform()返回的元素取代当前的元素。返回的 OutputIterator 指向最后被赋给 result 的元素的下一个位置。

```
#include <algorithm>
#include <vector>
#include <math.h>
#include <iostream.h>
/*
* 输出：
  original array values: 3 5 8 13 21
  transform each element by doubling: 6 10 16 26 42
  transform each element by difference: 3 5 8 13 21
*/
int double_val( int val ) { return val + val; }
int difference( int val1, int val2 ) {
     return abs( val1 - val2 ); }

int main()
{
```

```
    int ia[] = { 3, 5, 8, 13, 21 };
    vector<int, allocator> vec( 5 );
    ostream_iterator<int> outfile( cout, " " );
    cout << "original array values: ";
    copy( ia, ia+5, outfile ); cout << endl;
    cout << "transform each element by doubling: ";
    transform( ia, ia+5, vec.begin(), double_val );
    copy( vec.begin(), vec.end(), outfile ); cout << endl;
    cout << "transform each element by difference: ";
    transform( ia, ia+5, vec.begin(), outfile, difference );
    cout << endl;
}
```

unique()

```
template< class ForwardIterator >
ForwardIterator
unique( ForwardIterator first,
        ForwardIterator last );
template< class ForwardIterator, class BinaryPredicate >
ForwardIterator
unique( ForwardIterator first,
        ForwardIterator last, BinaryPredicate pred );
```

对于连续的元素，如果它们包含相同的值（使用底层类型的等于操作符来判断），或者把它们传给 pred 的计算结果都为 true，则这些元素被折叠成一个元素。例如，在单词 mississippi 中，语义上的结果是 misisipi。注意，四个 i 不是连续的，所以不会被折叠。类似地，因为两对 s 也是不连续的，所以也没有被折叠成单个实例。为了保证所有重复的实例都被折叠起来，我们必须先对容器进行排序。

实际上，unique()的行为有些不太直观，类似于 remove()算法。在这两种情况下，容器的实际大小并没有变化，每个惟一的元素都被依次拷贝到从 first 开始的下一个空闲位置上。

因此，在我们的例子中，实际的结果是 misisipippi，这里的 ppi 字符序列可以说是算法的残留物。返回的 ForwordIterator 指向残留物的起始处。典型的做法是，这个 iterator 被传递给 erase()，以便删除无效的元素。（由于内置数组不支持 erase()操作，所以 unique()不太适合于数组：unique_copy()对数组更为合适一些。）

unique_copy()

```
template< class InputIterator, class OutputIterator >
OutputIterator
unique_copy( InputIterator first, InputIterator last,
     OutputIterator result );
template< class InputIterator, class OutputIterator,
     class BinaryPredicate >
OutputIterator
unique_copy( InputIterator first, InputIterator last,
     OutputIterator result, BinaryPredicate pred );
```

unique_copy()把每组“含有相同的值（使用底层类型的等于操作符来判断）”或“被传递给 pred 时计算结果为 true（描述见 unique()）”的连续元素，拷贝一个实例。为了保证所有重复的元素都被清除掉，必须先对容器进行排序，返回的 OutputIterator 指向目标容器的尾部。

```
#include <algorithm>
#include <vector>
#include <string>
#include <iterator>
#include <iostream.h>

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }
void (*pfi)( int ) = print_elements;
void (*pfs)( string ) = print_elements;

int main()
{
    int ia[] = { 0, 1, 0, 2, 0, 3, 0, 4, 0, 5 };
    vector<int,allocator> vec( ia, ia+10 );
    vector<int,allocator>::iterator vec_iter;

    // 生成不能交换的序列：没有连续的 0
    // 结果：0 1 0 2 0 3 0 4 0 5
    vec_iter = unique( vec.begin(), vec.end() );
    for_each( vec.begin(), vec.end(), pfi ); cout << "\n\n";

    // 排了序的向量：0 0 0 0 0 1 2 3 4 5
    // 应用 unique() 后：
    // 结果：0 1 2 3 4 5 2 3 4 5
    sort( vec.begin(), vec.end() );
    vec_iter = unique( vec.begin(), vec.end() );
    for_each( vec.begin(), vec.end(), pfi ); cout << "\n\n";

    // 从容器中删除无效元素
    // 结果：0 1 2 3 4 5
    vec.erase( vec_iter, vec.end() );
    for_each( vec.begin(), vec.end(), pfi ); cout << "\n\n";

    string sa[] = { "enough", "is", "enough",
        "enough", "is", "good"
    };

    vector<string,allocator> svec( sa, sa+6 );
    vector<string,allocator> vec_result( svec.size() );
    vector<string,allocator>::iterator svec_iter;

    sort( svec.begin(), svec.end() );
    svec_iter = unique_copy( svec.begin(), svec.end(),
        vec_result.begin() );
```

```
    // 结果: enough good is
    for_each( vec_result.begin(), svec_iter, pfs );
    cout << "\n\n";
}
```

upper_bound()

```
template< class ForwardIterator, class Type >
ForwardIterator
upper_bound( ForwardIterator first,
             ForwardIterator last, const Type &value );
template< class ForwardIterator, class Type, class Compare >
ForwardIterator
upper_bound( ForwardIterator first,
             ForwardIterator last, const Type &value,
             Compare comp );
```

upper_bound()返同一个 iterator，它指向在[first,last)标记的有序序列中可以插入 value。而不会破坏容器顺序的最后一个位置。这个位置标记了一个大于 value 的值。例如，已知序列：

```
int ia[] = {12,15,17,19,20,22,23,26,29,35,40,51};
```

用值 21 调用 upper_bound()，返回一个指向值 22 的 iterator。用值 22 调用 upper_bound()，则返回一个指向值 23 的 iterator。第一个版本使用底层类型的小于操作符，而第二个版本根据 comp 对元素进行排序和比较。

```
#include <algorithm>
#include <vector>
#include <assert.h>
#include <iostream.h>

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }

void (*pfi)( int ) = print_elements;

int main()
{
    int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40};
    vector<int,allocator> vec(ia,ia+12);

    sort(ia,ia+12);
    int *iter = upper_bound(ia,ia+12,19);
    assert( *iter == 20 );

    sort( vec.begin(), vec.end(), greater<int>() );
    vector<int,allocator>::iterator iter_vec;

    iter_vec = upper_bound( vec.begin(), vec.end(),
        27, greater<int>() );
```

```
    assert( *iter_vec == 26 );

    // 输出: 51 40 35 29 27 26 23 22 20 19 17 15 12
    vec.insert( iter_vec, 27 );
    for_each( vec.begin(), vec.end(), pfi ); cout << "\n\n";
}
```

堆算法

标准库提供的 heap（堆）是一个 max-heap。所谓 max-heap 是一个用数组表示的二叉树，它的每个节点上的键值大于或等于其儿子节点的键值（完整的讨论见［SEDGEWIOK88］）（另外一种表示是 min-heap，其中每个节点的键值小于或等于其儿子节点的键值。）在标准库的表示中，最大的键值（可以把它想像成树的根）总是在数组的开始处。例如，以下的字母序列满足堆的要求：

满足堆要求的字母序列

X T O G S M N A E R A I

在这个例子中，X 是根节点，有一个左儿子 T 和右儿子 O。注意，两个儿子之间的顺序是不要求的（即，左儿子不必小于右儿子）。G 和 S 是 T 的儿子，而 M 和 N 是 O 的儿子。类似地，A 和 E 是 G 的儿子，R 和 A 是 S 的儿子，I 是 M 的左儿子，而 N 是叶节点，没有儿子。

四个泛型堆算法。make_heap()、pop_heap()、push_heap()和 sort_heap()为堆的创建和操纵提供了支持。后三个算法假定：由 iterator 对标记的序列代表了一个真正的堆（如果该序列不是一个堆的话，则算法的运行时刻行为是未定义的）。注意，list 容器不能被用作堆，因为它不支持随机访问。内置数组可以被用来支持一个堆，也是 pop_heap()和 push_heap()算法难以与数组一起使用，因为这两个算法要求改变数组的大小。我们先简要介绍这四个算法，然后用一个小程序说明它们的用法。

make_heap()

```
template< class RandomAccessIterator >
void
make_heap( RandomAccessIterator first,
           RandomAccessIterator last );
template< class RandomAccessIterator, class Compare >
void
make_heap( RandomAccessIterator first,
           RandomAccessIterator last, Compare comp );
```

make_heap()把[first,last)范围内的元素做成一个堆。双参数版本使用底层类型的小于操作符作为排序准则，第二个版本根据 comp 对元素进行排序。

pop_heap()

```
template< class RandomAccessIterator >
void
pop_heap( RandomAccessIterator first,
          RandomAccessIterator last );
```

```
template< class RandomAccessIterator, class Compare >
void
pop_heap( RandomAccessIterator first,
          RandomAccessIterator last, Compare comp );
```

pop_heap()并不真正地把最大元素从堆中弹出，而是重新排序堆。它把 first 和 last-1 交换，然后将[first,last-1)范围的序列重新做成一个堆。之后，我们就可以用容器的成员操作 back()，来访问“被弹出”的元素，或者用 pop_back()将它真正删除掉。双参数版本使用底层类型的小于操作符作为排序难则，第二个版本根据 comp 对元素进行排序。

push_heap()

```
template< class RandomAccessIterator >
void
push_heap( RandomAccessIterator first,
           RandomAccessIterator last );
template< class RandomAccessIterator, class Compare >
void
push_heap( RandomAccessIterator first,
           RandomAccessIterator last, Compare comp );
```

push_heap()假设由[first,last-1)标记的序列是一个有效的堆，要被加到堆中的新元素在位置 last-1 上，它将[first,last)序列重新做成一个堆。在调用 push_heap()之前，我们必须先把元素插入到容器的后面，或许可以使用 push_back()操作符（这将在下一个程序例子中说明）。双参数版本使用底层类型的小于操作符作为排序准则，第二个版本根据 comp 对元素进行排序。

sort_heap()

```
template< class RandomAccessIterator >
void
sort_heap( RandomAccessIterator first,
           RandomAccessIterator last );
template< class RandomAccessIterator, class Compare >
void
sort_heap( RandomAccessIterator first,
           RandomAccessIterator last, Compare comp );
```

sort_heap()对范围[first,last)中的序列进行排序，它假设该序列是一个有效的堆（否则，它的行为是未定义的）。（当然，经过排序之后的堆就不再是一个有效的堆！）双参数版本使用底层类型的小于操作符作为排序准则，第二个版本根据 comp 对元素进行排序。

```
#include <algorithm>
#include <vector>
#include <iostream.h>

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }

int main()
{
     int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40 };
```

```
    vector< int, allocator > vec( ia, ia+12 );

    // 结果: 51 35 40 23 29 20 26 22 19 12 17 15
    make_heap( &ia[0], &ia[12] );
    void (*pfi)( int ) = print_elements;
    for_each( ia, ia+12, pfi ); cout << "\n\n";

    // 结果: 12 17 15 19 23 20 26 51 22 29 35 40
    // a min-heap: root is smallest element
    make_heap( vec.begin(), vec.end(), greater<int>() );
    for_each( vec.begin(), vec.end(), pfi ); cout << "\n\n";

    // 结果: 12 15 17 19 20 22 23 26 29 35 40 51
    sort_heap( ia, ia+12 );
    for_each( ia, ia+12, pfi ); cout << "\n\n";

    // 再加一个新的最小的元素:
    vec.push_back( 8 );

    // 结果: 8 17 12 19 23 15 26 51 22 29 35 40 20
    // 将最新最小的元素放在根处

    push_heap( vec.begin(), vec.end(), greater<int>() );
    for_each( vec.begin(), vec.end(), pfi ); cout << "\n\n";

    // 结果: 12 17 15 19 23 20 26 51 22 29 35 40 8
    // 应用次最小的元素替代最小的

    pop_heap( vec.begin(), vec.end(), greater<int>() );
    for_each( vec.begin(), vec.end(), pfi ); cout << "\n\n";
}
```

英汉对照索引

凡例

1.本索引共分三级:第一级为加粗显示，第二级和第三级各自缩进一级。其中的意义第二级应加上第一级才完整。第三级则应该加上前两级才完整。

2.每级索引结构相同，分成英文，中文，多个索引项以及参见部分。索引项中，冒号前面是章节号(附录用 A 表示)，冒号后的是页码。章节号加粗的表示义项为该章节的主题。

符号

B

G

H

I

M

N

P

Q

R

S

T

U

V

W