第一章 绪论

1．1 程序设计与程序设计语言

半个多世纪以来，计算机技术无论是作为科学学科还是作为现代产业，已从一颗幼苗成长为枝繁叶茂的参天大树。回顾其发展历程，计算机科学与工程的发展（包括“网络时代”的今天）一直是围绕着程序设计这个中心课题进行的。

1.1.1 计算机与程序设计

计算机也许是廿世纪人类带给廿一世纪的最有价值的礼物，是人类文明历史上最伟大的发明之一，现在估计它对人类生活将会产生多么大的影响也许还为时尚早。不过我们总是能够把计算机与人类其它伟大发明，如飞机、电灯、汽车、电视机等等相比较，从其差别中可以感到它的威力、影响和壮美前景。

计算机与其它发明的主要差别是：

·人类的发明都是对自己的器官的延长或替代，别的发明都可归结为人的四肢和五官的延长或替代；而计算机则是人类大脑功能的延长或某种替代，所以被称为“电脑”。

·人类的发明可以应用在各个不同的局部领域，计算机与众不同之处在于它可以应用在几乎所有的人类活动的领域。

目前，计算机的这两个特征还在发展之中，计算机可以在怎样的程度上延长或代替大脑的活动，计算机可以在何种程度上被广泛而深入地应用于各个领域，谁也不能指出一个“到顶”不再发展的时间，不过我们现在可以指出的是，使计算机具有如此影响力的根本原因是，计算机不是一个一次性的直接服务产品，它为人类服务是有条件的，这个条件就是程序设计。

没有程序和程序设计，计算机可以说是一堆废物，也可以换一个方法说：程序(软件)是计算机的必要组成部分。计算机首先要求人们不断地在程序设计上付出大量的创造性劳动，然后才能享受到它的服务。

计算机本身是人类智慧的产物，它的诞生又导致了人们投入十倍、百倍的精力和智慧用于程序设计和软件开发,从而又引发出无穷无尽的新的发明创造。

有关计算机科学与技术的大部分研究工作都是围绕程序设计进行的，特别是形势发展到今天，新型计算机本身(主要指计算机硬件核心的芯片)的设计也归结为使用高级硬件描述语言的‘程序设计”,所以也可以说整个计算机产业(硬件和软件的研究、设计和生产)就是在进行程序的设计与开发。因此计算机专家和专业人员的培养和训练的最重要任务就是让学生掌握程序设计及其相关理论的研究和开发能力。

其实，把计算机比作电视机或电冰箱并不恰当，它更像是人所驯养的一匹马，能按照人的指令去完成各种任务，所谓程序，就是要计算机完成某一任务所规定的一系列动作步骤。计算机好像是唯人的命令是从的仆人，严格地按照程序规定的步骤完成任务。当然，计算机程序不是简单的几条或几十条命令放在一起，现代的计算机每秒钟可以执行成千上万条指令（目前，最快的计算机的峰值速度可达每秒钟处理 35 万亿条指令，现在人们正在研制千万亿次的 P 级高性能计算机）。因此，计算机程序规模很大，内容十分复杂。为计算机编程序是一种非常复杂，具有挑战性的工作。也可以说，自计算机问世的半个世纪以来，人们都是在研究设计各种各样的程序，使计算机完成各种各样的任务。

1946 年美国的 Mauchly 和 Eckert 研制的第一台电脑 ENIAC 上应用的程序是用来计算火炮的弹道函数的。微软公司开发的 Windows 系列是一种用来管理计算机资源的图形界面操作系统，它也是一个大规模的程序。这个程序已为开发商创造了数百亿美元的财富。1997 年 5 月份，另一个计算机程序的开发也引起了新闻界的注意，这就是击败了国际象棋世界冠军卡斯帕罗夫的计算机程序“ Deeper Blue ”(“ 超蓝 ”)。这是一个能够下棋的程序，它以 4:2 击败了当代最强的国际象棋大师，赢得了 70 万美元的奖金。（实际上，IBM 公司为开发“ Deeper Blue ”的投入比这笔奖金要多得多。）

因此，程序设计是一件工作量永无止境，极其困难复杂而又富有魅力和创造乐趣的工作。这样一项职业，每年都吸引着数以十万计的优秀人才投入其中，容纳下他们的全部智慧、想象力和创造性的工作，换来了计算机产业和计算学科的日新月异的发展。

1.1.2 程序设计语言，从低级到高级

程序设计的任务就是用计算机懂得的语言即程序设计语言编写程序，然后交给它去执行。

1．计算机指令系统

严格地说，一台“纯粹”的计算机(或称“裸机”)并没有特别高超的本领，必须由人通过“程序”指挥它完成各种任务。

· 计算机本身只会完成几十种(或一百多种)不同的简单“动作”，例如把内存的某地址的数取到某寄存器；把某地址的两个数相加送到某寄存器；判断某个值是否为 0 等等。

· 计算机设计者把计算机可以完成的动作编辑成一个指令表，每种动作赋予一个二进制代码，并为机器的每种动作设计一种通用的格式：由指令码和内存地址组成的指令。一条指令就是一个固定长度的由指令码和地址码组成的二进制位串，这就是计算机唯一可以读懂的语言。一般称作机器语言。

·程序员把要计算机完成的任务分解为一系列其指令表(或指令系统)包括的“动作”，以指令序列的形式写出来，这就是机器语言程序设计。

2．低级编程语言

1949 年，普林斯顿大学的冯·诺依曼（von Neumann）研制的 EDVAC 计算机，首次把上述指令序列形式的程序与数据一同置于磁芯存储器中。从那时开始，计算机通过识别置于存储器中的由 0 和 1 组成的二进制指令序列，来依次执行指定的操作，这种工作方式一直延续到今天。

在计算机应用的最初的十几年中，大多数计算机程序就是用所谓“机器语言”编写的。例如，做一次加法 TOTAL=PRICE+TAX, 程序为：

```
156C        // 取 6C 内容送寄存器 5
166D        // 取 6D 内容送寄存器 6
5056        // 把二值相加，结果送寄存器 0
306E        // 把寄存器 0 中的结果送地址 6E
C000        // 停机
```

这种“语言”虽然十分简单，机器可以“看”懂，但对于程序员来说却很不方便。完成一个简单的计算公式也要编写几十条指令，编程工作枯燥而繁琐，程序冗长而难读，调试、修改、移植和维护都是难题。因此，早期的计算机应用不广，编程的专业性极强，人们逐渐感到用机器语言编程是计算机应用向前发展的瓶颈。

50 年代的计算机专家们，一方面以“机器语言”的形式写出了大量的应用程序，例如在那个年代实现了天气预报和卫星上天，计算机曾做出了巨大的贡献。另一方面，他们也在研究，可否在语言问题上兼顾对话的双方：计算机与人。“机器语言”——太“低级”了，机器容易懂，但对人却太不方便。于是，一种“汇编系统”程序问世了，这种程序的功能是把一种“汇编语言”翻译为

“机器语言”。例如，上面的加法程序用汇编语言可以写为：

```
LD R5,PRICE
LD R6,TAX
ADDI R0,R5,R6
ST R0,TOTAL
HLT
```

汇编语言用符号代替二进制地址表示参加操作的数据，这样大大减少了编程工作的困难。后来又改进为“宏汇编语言”。一条宏汇编指令可以代替多条机器指令。人们用汇编语言或宏汇编语言写程序，通过汇编系统（Assembler）把它们翻译成计算机唯一“看”得懂的机器语言程序，然后再令其执行。

使用汇编语言编程比使用机器语言编程要容易，另外由于汇编语言指令与机器语言指令基本上一条对一条或一条对几条，所以汇编系统的程序开发也不太复杂。因此，汇编语言编程很快取代了机器语言编程。到了 60 年代，机器语言编程已经比较少了。后来以 FORTRAN 和 ALGOL60 为代表的高级语言逐渐流行，汇编语言的应用从应用程序的开发转向了系统程序，如操作系统（Operating System，OS），编译系统（Compiler）的开发。到了 70 年代，新一代的高级语言 Pascal 和 C 语言问世，又逐渐在系统程序的开发领域取代了汇编语言。然而到了 80 年代，计算机一些新的应用形式如单板机、单片机等的简单编程仍然以汇编语言为主。到了 90 年代，汇编语言的应用才相对减少了。

汇编语言和机器语言都属于低级语言，这是因为它与人的习惯语言方式距离较远，所以它们的共同缺点是：

- 依赖于机器，可移植性差；
- 代码冗长，不易于编写大规模程序；
- 可读性差，可维护性差。

总之，汇编语言没有解决计算机编程难的基本问题。低级语言必然为高级语言取代，没有高级语言就没有今天的计算机应用的网络化、多媒体化、智能化等等。

３．高级程序设计语言

有了计算机以后，许多科学与工程计算问题交给计算机完成，大多获得了成功。计算机比人们用手工计算速度快，精度高，为人们帮了大忙，那么编程序是否也可以请计算机帮忙呢?

从廿世纪 50 年代开始，作为当时的尖端科学技术，许多计算机科学家开始研究一种“可以编程序的程序”，有些研究者把它称为“程序设计自动化”。

这种能够把用高级语言写出的程序翻译为机器语言的指令序列的程序被称

为编译程序（compiler）、编译系统或编译器。

最早取得成功的是 IBM 公司的 John Backus，他领导的研究小组于 1954 年提出了一个高级语言规范“IBM mathematical FORmula TRANslation system”（FORTRAN），两年后完成了编译系统的研制。而正式公布的 FORTRAN Ⅰ语言和编译系统则是在 1957 年 4 月。此后，许多不同的高级语言及其编译系统被人们开发和使用，常见的有 ALGOL60，BASIC，Pascal，C，C++，JAVA，C#等。

程序设计语言作为人和计算机之间通讯的媒介，从低级向高级，从第一代语言，到第二代语言，第三代语言...的演变，就是从符合计算机的特征向着符合人的特征的方向演变，语言越是“高级”，其语法越是符合人的习惯，便于普通用户编程。这里的“高级”并非“高深”，相反地，低级语言编程是更难的任务。

FORTRAN 语言的诞生标志着计算机技术的一个新的里程碑。有了高级程序设计语言及其编译系统的帮助，人们可编制出规模越来越大，结构越来越复杂的程序。计算机的应用领域不断扩展，计算机技术能够发展到今天的局面，没有高级语言及其编译技术的突破是不可能实现的。

1.1.3 程序设计方法学的发展

程序设计技术的发展，是一个逐步提高的过程，是一个与实际应用需要互相制约和促进的螺旋式的发展进程。我们可以把这个过程描述如下：

１．程序设计技术的初级阶段

计算机诞生，von Neumann 模式形成，低级语言编程是主要开发形式。这个时期计算机在工程计算领域的成功应用使人们发现低级语言程序设计成为推广计算机应用的主要障碍。

第一代高级语言（以 FORTRAN 和 ALGOL60 为代表）诞生，从低级语言编程转向高级语言编程，可以说是计算机“生产力”的一次解放。

当时计算机应用的主要领域是数值计算，因此，语言中变量、表达式、向量和矩阵(数组)的处理是其主要部分；例如 FORTRAN 语言名称的含义就是“数学公式翻译语言”。而像编译系统这样的系统程序，被认为是更难编程的。这类程序仍由低级语言，主要是汇编语言编写。

这个时期的程序设计技术重视程序设计技巧，注意节省内存和目标代码的效率。

高级语言的出现使得程序设计的难度降低，导致了计算机应用在五六十年代的发展(包括宇航和机器人等大型应用程序的开发、使用)进入新的阶段。

60 年代，以大规模程序频频出错(例如 1962 年，因软件出错导致美国金星探测器水手 II 号卫星发射失败)为特征的“软件危机”发生，引起关于“Goto 语句”的辩论。

2．结构程序设计（Structured Programming，SP）阶段

为了完成一些规模大，复杂度高，使用周期长，投入人力、物力多的大型程序设计任务，结构程序设计(和软件工程)思想被提倡，以 Pascal 语言和 C 语言为代表，强调数据类型、程序结构，程序设计的目标把可靠性、可维护性的要求放在了比高效率更重要的位置上，主张宁肯牺牲一点效率也要保证程序无错或少错。

结构程序设计思想的要点是：

·注意程序的可读性，即程序书写的静态结构要清晰，符合人的阅读习惯，以及注意编程格式，给标识符一个有意义的命名，增加必不可少的注释等等。

·采用自顶向下、逐步求精的设计方法符合人们解决复杂问题的普遍规则，可提高软件开发的成功率；由此开发出的程序具有清晰的层次结构，易于阅读、理解及修改、调试、扩充。

· 尽可能地使程序运行的动态结构与程序书写的静态结构相对地比较一致。这依赖于下面几点：

1）程序语言只包括顺序、选择、循环三种控制结构，其目标是使程序的各个局部符合控制的单入口单出口原则。

与单入口单出口原则相关的是首先由 E.Dijkstra 提出的应从高级语言中取消 Goto 语句的建议，因为造成程序混乱的根源是 Goto 语句。虽然在当时讨论的结论是在高级语言中(包括 C 和 Pascal 语言)保留 Goto 语句，但建议程序员不用或少用它。但到了 90 年代，上述 E.Dijkstra 的意见在一种新的由 C++语言衍生而来的 JAVA 语言中得到了采纳，Goto 语句被取消了。

2）努力规范和简化模块之间的联系，充分利用过程或函数的参数机制，规范程序模块间的接口。

·严格区分数据类型，所有程序中出现的数据都必须进行类型说明和任何数据操作都进行类型检查。

体现 SP 思想的语言是 Pascal 语言和 C 语言。

N.Wirth 设计的 Pascal 语言于 1971 年问世，它一方面从 ALGOL60 等语言中吸收了许多有益的成分，另一方面，它是结构程序设计思想的最集中的体现，语言规模适中，精致简明，其全部语法规则采用形式化的语法描述形式 BNF（Backus 范式）和语法图，更受到人们的欢迎。Pascal 语言最初是为了教学和编写系统软件，这两个目标都达到了，它也许是最早的可以用来编制系统软件

的高级语言。

C 语言是这个时期另一个最重要的语言。虽然，它与 Pascal 语言都体现了 SP 思想，但是二者又有相当大的差别。前者注重抽象，注重简明严格，是程序设计理论专家的研究成果。而 C 语言则更注重实用，注重表达式的简捷高效；为了方便设计系统软件，不惜引入低级操作功能。到了 80 年代，C 语言受到了软件开发人员的欢迎，逐渐成为软件开发的主流语言。至今，大多数优秀的程序员是 C/C++语言的熟练掌握者，大多数成功的软件是 C 或 C++程序。

同一时期还有许多高级语言受到重视，其中应该提到的是 Ada 语言。

程序设计技术的进步使计算机在更广泛的领域中应用成为可能，20 世纪七、八十年代，包括系统程序(操作系统和编译系统)，大规模的信息管理系统程序和一些人工智能程序被成功地开发出来。例如，为美国国防部研制的导弹预警系统程序长达 380 万句。

计算机技术的发展，使得结构程序设计方法面对诸如图形界面、窗口系统等大规模的程序的软件商品化、部件化的要求，编程工作成为新的瓶颈。

３．面向对象程序设计（Object-Oriented Programming，OOP）阶段

80 年代，面向对象程序设计逐渐从理论转向实践，以 Smalltalk，C++等语言为代表，程序设计理论步入成熟期。

虽然类（Class）的概念在 60 年代后期就已经为 Simula_67 语言所采用，但面向对象程序设计的思想还是被认为是起源于 80 年代初。在这个时期，A.Kay 研制了 Smalltalk 语言，B.Stroustrup 则开发了 C++ 语言，然而面向对象程序设计方法真正在软件的工业开发领域流行则是 80 年代末期的事情。

OOP 方法在 90 年代的盛行主要有两个基本的原因：

其一是 OOP 方法从思想上与 SP 方法相比是抓住了软件开发的本质和规律，经过十年的研究和工程试验，已为软件专家和软件开发商所接受。 OOP 方法开发的软件产品，易重用，易修改，易测试，易维护，易扩充。从程序设计方法学的研究角度出发， OOP 方法是人们经过五十多年的程序设计实践，总结出的成熟的技术。现在已经没有人怀疑， OOP 方法是 21 世纪的主流开发手段。

其二就是计算机所要解决的问题越来越重要，越来越复杂。从航天飞机的发射，到机场的调度，以至于办公大楼的电梯，电网的供电系统，庞大的电话自动交换机的控制……问题的规模和复杂度呈几何级数增长(例如美国航天飞机系统的程序长达 4000 万句)，原来的程序设计技术不再能满足要求。最近十年，软件的市场商品化得到了巨大的发展。一方面，以软件包形式出现的作为开发环境与工具的软件产品年销售额已达数百亿美元；另一方面，专为用户设

计的定制软件也越来越大，越来越复杂。其开发代价常常是以百万、千万美元计算。例如，金融证券的管理系统，为一种新型飞机的设计和投产，包括从飞行控制系统到基地维护设备，从飞行训练模拟系统到飞行中的娱乐系统……全世界每年定制软件的产值也达千亿美元。如此大量的程序设计任务，已经导致人们不得不探索新的更可靠更高效的设计开发技术。 OOP 方法正是在这样的形势下，成为唯一可行的选择。

OOP 技术之所以能适应今天软件产业的需要，是因为它比较好地解决了软件模块化、信息隐蔽和抽象的目标。通过类和对象，把程序所涉及的数据结构和对它施行的操作有机地组成模块，对于数据和对数据的处理细节进行了最大限度的封装，其密封性、独立性和接口的清晰性都得到了加强。90 年代的程序设计方式，由于有了 OOP 技术的支持，发生了质的变化，未来 20 年的软件开发将越来越向着可重用部件的组装方式发展，而可重用部件的存在形式就是类及其派生系统。

支持 OOP 方法的高级语言主要产生在 80 年代，近百种语言主要是在高等学校和科研单位中供研究使用的。由于 C++语言强大的功能，加上它对已经十分庞大的 C 程序员群体具有自然的吸引力，由它取代 C 语言成为软件开发领域的主要开发语言之一，几乎成了顺理成章的事。

Smalltalk 语言也是一个影响很大的 OOP 语言，目前它在金融领域中 L/S 模型结构应用领域有很大市场。

较晚(80 年代中期)出现的 EIFFEL 语言，和 Ada 语言的 OOP 版本 Ada9x 语言都是很受重视的面向对象程序设计工具。

值得一提的是，1994 年出现的 JAVA 语言和 2001 年推出的 C#语言，也是 OOP 语言。 JAVA 语言的产生，开始是用来设计 Internet 上的 Web 浏览器，它们脱胎于 C++语言，与 C++语言的语法规则大体一致。

1.1.4 程序设计的范型

上面介绍的程序设计方式一般被称为命令型（imperative）程序设计或过程型（procedural）程序设计，不同的程序设计方式也称作不同的程序设计范型（programming paradigms)。

1．命令型程序设计

命令型程序设计亦称过程型程序设计，这是用计算机求解问题的基本方式，面对问题，设计求解问题的算法，根据算法编写指令序列。机器语言、FORTRAN、ALGOL60、BASIC、Pascal、C(C++)、Ada 等语言都是为支持命令型程

序设计范型而设计的。

2．面向对象程序设计

面向对象程序设计是命令（过程）型程序设计的发展，是命令型程序设计的高级、成熟阶段，它们是半个多世纪以来程序设计技术的主流。Smalltalk、C++、JAVA、C# 等语言都是为支持面向对象程序设计范型而设计的。

命令型程序设计中的数据是死的、被动的（passive），好像水、煤这样的物质，程序就是为计算机处理数据安排的动作序列。而面向对象程序设计把数据和对数据的操作（方法）有机地组合成类和对象，形成可动作地部件，好像计算机系统的电源、光驱、窗口等等。

面向对象程序的模块化结构清晰，安全性好，可重用性好，特别适合大规模网络化软件的开发，因此，目前它已成为软件开发的主要形式。

还有两种不同的程序设计范型有必要介绍给读者：

3．函数型程序设计

函数型程序设计的思想是把通过程序设计求解问题视为一个“ black boxes”结构，用数学家的说法，就是一个函数。编程过程就是把一个复杂的函数构造为若干简单函数的嵌套。

例如，计算若干整数的平均值的函数可由三个简单函数 Div,Sum,Count 组合而成：(Div （Sum Numbers）(Count Numbers))，

函数 Sum 完成 Numbers:$x_1,x_2,\ldots x_n$的求和计算；

函数 Count 完成 numbers: $x_1,x_2,\ldots x_n$的计数计算；

函数 Div 完成 Sum 和 Count 结果的除法计算。

LISP 语言是非常值得一提的函数型程序设计语言。它在 60 年代初由人工智能研究的创始人之一 John McCarthy 提出。这是一个用于符号处理的函数式程序设计语言，巧妙地用 S-表达式来统一处理程序和数据，其运算符的前缀表示法，递归的数据和控制结构等都十分有趣。至今 LISP 语言(及 APL 语言)仍是函数式程序设计语言的主要代表，用于人工智能和计算机硬件设计等领域。

4．逻辑型程序设计

逻辑型(logical)程序设计亦称申述型（declarative）程序设计，以逻辑程序设计思想为基础，把事实和规则作为知识，通过推理机制产生计算结果。逻辑程序设计的关键是要给出一个精确的问题描述（而不是设计求解算法），执行一个通用的求解算法，也就是说，编程的任务是解决“问题是什么？”而不是“用什么算法解决问题？”，这种程序设计范型适用于一些特定的应用领域。主要代表语言是诞生于 70 年代初的 PROLOG 语言，它是人工智能研究领域的重要工具，80 年代日本投巨资开发的“第五代计算机系统”即以 PROLOG 为

主要语言。由于近年来形式逻辑领域的研究取得进展，逻辑程序设计越来越受到关注。

1.1.5 程序设计技术的四个层次

程序设计技术的研究与培训一般是在不同的层次上进行的，高水平的程序员必须在各个不同的层次上接受充分的训练，以获得比较深的造诣。这四个层次是：

- 算法
- 程序设计方法学
- 程序设计语言
- 程序设计环境与工具

1．算法的设计与分析

所谓计算机算法就是用计算机解决问题的步骤。算法理论的研究除了要设计出正确的解法之外，主要研究算法在时间和空间这两个方面的复杂度，即如何以最少的时间和空间代价解一个给定的问题。

算法理论是整个计算机科学的核心。一个高级程序员必须掌握有关算法设计与分析的知识，否则他不会编出高水平、高质量的程序，更不要说有创造性的程序了。例如，多媒体技术是 90 年代软件技术的新发展，由于图形，图像和声音信息扩充了原来的单纯的字符信息模式，一下子使得计算机处理的数据增加了几千倍，于是数据传输成了大问题，其中关键问题是数据的压缩和解压缩算法。数据压缩算法问题的解决保证了计算机多媒体化的实现。

算法研究的特点是：

· 需要深入的基础理论知识。许多著名的计算机科学家都在算法设计与分析领域有深入的研究，从而对计算机的发展做出巨大的贡献。

· 在时间上有相对的稳定性，与描述算法的语言无关，其研究的内容和成果可以在较长的时间发挥作用。例如，英国计算机科学家 C.A.R.Hoare 60 年代设计的 Quicksort 算法至今仍是有效的排序算法。50 年代人们在研制编译程序提出的 Hash 算法，到了 90 年代仍然是重要的数据存储与检索算法，在大型数据库和人工智能系统的开发中仍在不断地被改进和采用。

· 在效果上往往在广泛的领域中起到突破性作用。例如，70 年代末提出的一种大素数判别的概率算法，后来被用到密码学领域而形成“公钥密码系统”，对于保密通讯产生了巨大的影响。它不仅在 80 年代广泛被用于国防、外交和商业的密码通讯，到了 90 年代计算机技术进入到网络化时代，数据的加密解密算

法的研究更成为网络系统安全机制的关键。

２．程序设计方法学

程序设计方法的研究主要是从 60 年代末期的“软件危机”后受到重视，程序设计方法学的目标是在开发正确可靠的软件的前提下，如何缩短软件开发周期，进而如何延长所设计的软件使用周期，这里就包括了程序的可读性、可测试性、可维护性、可扩充性和可重用性。

近十几年来面向对象程序设计方法的大量研究工作已把人们对于程序的认识推向了一个全新的阶段。

程序的设计与开发不是采用支持 OOP 的语言就可以写出面向对象的程序。程序员必须从程序设计方法学上有所领悟，这是新时期程序设计技术对于程序员的新的要求。程序设计方法学的研究有下面的特征：

· 其总的目标是缩短软件开发周期，延长软件生存周期，降低软件的开发成本；

· 它的理论更新与计算机自身应用规模的发展同步，往往不是可以由人们自由选择这种或那种方法，而是一种难以抗拒的必然趋势。

· 它的研究成果虽然相对于计算机算法的成果变化较快，但相对于各种语言及其版本的变化则相对稳定，至少在十年、廿年的范围内应有一个相对的稳定期。

３．程序设计语言

本书的重点在于介绍一种语言的使用方法，学习语言可以说是掌握编程技术的关键，任何人学习程序设计都必须从学习一种语言开始，其关键是选择一种最合适的最有前途的语言。学习语言是程序设计的基础，因此，在本节提出的程序设计技术的四个层次中，语言是最基本最起码的要求。而掌握了程序设计方法学的理论与方法可以指导你写出结构好的程序，学好计算机算法设计与分析的理论可以帮助你写出高水平甚至有突破性的程序，至于第四个层次，程序设计的环境与工具则有利于程序员快速、高效地完成设计工作。

４．程序设计环境与工具

为程序员进行程序设计开发提供环境条件和辅助工具，这一点可以说是从有了机器语言编程就开始了，那时系统为程序员提供“标准子程序”，对于像三角函数、对数函数等常用的数学函数，已由专家编好程序，供程序员设计时调用。可以说几乎从一开始，人们就想到要通过计算机为程序员提供帮助。

有了高级语言后，除了系统提供的标准函数库之外，编辑器（Editor）、连接器（Linker）和调试器（Debuger）是系统为程序员提供帮助的主要形式。在这里，程序设计的主体是程序员，“工具”所起到的作用，绝对是辅助的。

软件开发技术的发展，使设计环境与工具在程序设计中所发挥的作用发生了某种质的变化：

· 工具越来越多，设计分析工具、测试分析工具、测试实例生成、模块代码生成工具、文档生成工具、工程进度管理工具等等统称 CASE(计算机辅助软件工程)工具。

· 工具越来越具有智能，代替和“强迫”程序员按照方法学和语法进行设计工作，如语法制导的编辑器，包括了程序框架和语法检查的功能。它们可以在程序员水平不高的条件下，在一定程度上保证代码的质量。

· 把编程的某些部分工作归结为“组装”和“集成”，例如像 Visual C++、Borland C++、Visual BASIC 等版语言都以大量的“篇幅”为用户提供了标准模式的编程工具，它们可以代替程序员把现成的窗口、菜单、对话框、滚动条、图标等标准的类和对象提供给用户，在设计应用程序时，可以像是部件组装那样，节省大量的编程时间。从某种意义上说，编程方式本身也正在发生质的变化。与电子产品的设计与生产一样，在五六十年代需要以电阻、电容，晶体管等元件来设计安装调试，而到了八九十年代，这样的工作已为各种集成电路部件的组装所代替。未来程序设计越来越明显的部件组装化的趋势是肯定的。

程序设计环境与工具，在我们的四个层次中处于最底层，其特征是：

· 它将越来越重要，逐渐成为程序设计的必要条件。

· 它将越来越面向一般用户，易学好用，灵活方便。

· 它在四个层次中最不稳定，几乎每年都在变化更新。

1．2　C++语言概述

据统计，目前世界上支持面向对象程序设计的语言已近百种，除了大多数是供研究用的非商业软件外，具有强大竞争力的 OOP 语言也不在少数，其中一类是以 Smalltalk 和 EIFFEL 语言为代表的新的面向对象语言。它们以其全新的语言规范和与开发工具紧密结合的编程机制为特征，特别是 Smalltalk 作为第一个面向对象语言在程序设计领域的影响当然是巨大的。另一类则是传统的重要编程语言向面向对象程序设计靠拢的结果，除了 C++语言之外，几乎所有成功的语言，包括 Pascal 语言，Ada 语言，C 语言，COBOL 语言，FORTRAN 语言，都发展了它们自己的面向对象的新版本，其中常见的有 Ada95，Borland 开发的 Turbo Pascal 5.5 和 6.0，Object C，Forth 等等。C++语言最初被称为“带类的 C”（C with Class）也应属于这类。这种情形在结构程序设计时期也曾出现，一些传统的编程语言如 FORTRAN、BASIC 语言都开发了结构化的新版本，不过其影响还是没有超过 Pascal 和 C 语言。

在这样的面向对象的“语言之林”中，C++语言能够获得成功，应该说不是偶然的。本章的内容也许会有助于搞清下面几个问题，为什么到了 90 年代，数以百万计的标有 Visual，Borland，Zortech 等等标记的 C++语言和编译系统被销售出去，为什么人们选择 C++语言作为软件的开发语言，为什么人们选择以 C++语言为样本语言来讲解 OOP 技术，为什么人们开始用 C++语言代替 Pascal 来描述算法和数据结构，为什么目前最热的 Internet 编程语言 Java 也是从 C++语言衍生脱胎而成的。

1.2.1　为什么选择 C++语言

面对软件开发任务的公司经理需要选择一种最合适的开发语言，讲授程序设计课程的计算机专家或教师也需要选择教学语言，投身于计算机技术领域的青年人才，其中有的人已经学习过 BASIC，Pascal，或 C 语言，有些人则刚刚开始，他们也需要考虑学习一种适当的高级语言。应该说，供他们比较和选择的重要语言中，C++是最强有力的候选者。

(1) 面向对象程序设计正在逐渐成为主流设计技术。十几年前，OOP 主要是在高校和科研机关作为研究课题，而目前，已有越来越多的人认为 OOP 已经不是供人们选择的一种方法，而是程序设计技术发展的必然趋势。无论是在

开发领域还是在教学领域,以结构化程序设计为主要特征的 C 和 Pascal 已经不太适应形势的需要了。

(2) OOP 技术并不取代 SP 和一般的程序设计的技能技巧。C++语言对于后者有较大的容纳，这一点使它比较容易受到欢迎。

(3) 由于各大公司的竞相开发，C++语言在各种不同机型上都有优秀的编译系统和相关的环境与工具。

(4) C++语言最可能取代 C 而成为主流的软件开发语言之一。所以能为程序员和软件开发商所接受，而一些新的软件开发语言如 JAVA，C＃语言与 C＋＋语言基本一致。

(5) 目前，在计算机教学领域，Pascal 和 C 仍被一些学校采用，然而，最近在国外出现了一批比较有水平的讲授 C++语言的教科书，以 C++语言为主的讲授面向对象设计技术，还有一些以 C++语言为主要描述语言的计算机专业如数据结构、算法设计与分析的相关教科书。这些动向说明，C++语言已成为计算机专业主要的教学语言。

1.2.2 C++语言简史

C++语言的作者是美国 AT&T 公司 Bell 实验室的 Bjarne Stroustrup 。

C++语言的设计是在实验室实际研究工作中发展起来的，它更注意原来的基础和可行性。

C++语言不同于 Smalltalk 语言， Smalltalk 语言由美国加利福尼亚州的 Xerox 研究中心推出（ Smalltalk76，Smalltalk80），是一种独立设计的面向对象语言。C++语言基本上走的是在已有的 C 语言的基础上扩张面向对象特征之路。

１． 带类的 C

1979 年 10 月在 Bell 实验室为 C 语言开发了一个名为 Cpre 的预处理器，这个预处理器使 C 语言中加入了从 Simula_67 中引入的类。在此后的数年中，这个“带类的 C”（ C with Class）被用于十几个系统的开发，取得了初步的成功。

据 Stroustrup 本人说，他在剑桥大学期间曾使用 Simula 语言，“感觉上对 Pascal 的死板和 Simula 的灵活性之间的对比是开发 C++的基本出发点。其中 Simula 的类的概念被视为关键的区别”。然而， Simula 的实现却只适用于小程序，编译系统的低效率，使得作者不得不改用 C 语言的前身 BCPL（Basic Combined Programming Language）语言来完成他的实验。这也许是产生在 C 语言上嫁接 Simula67 中的类的初衷吧。

选择 C 语言作为基础，是 C++语言开发者当时慎重考虑的另一个问题。虽然在那时，C 语言和 Pascal 语言刚刚成为引人注目的结构程序设计语言，但这两种语言的对比还是清清楚楚的。

在程序的清晰可读和安全性方面，Pascal 语言占有优势，而 C 语言则在灵活性、高效率、易获得(指在各种计算机上都有了 C 语言的实现)和可移植等方面优于 Pascal 语言。C++语言开发者说：“在以后的十年中，选择 C 作为基础，使其置身于系统程序设计的主流之中，而这正是我所希望的”。

２． C++语言的诞生

1985 年，由 C++语言开发者编写的《C++程序设计语言》一书的出版宣布了 C++ 1.0 版的诞生。在这之前从“带类的 C” Cpre（和 Cfront）到 C++语言是经过一系列的演变过程的，其中值得一提的是：1983 年底，贝尔实验室的项目负责人决定把 C++语言的开发作为关键项目予以支持，对设计者提出的第一个要求就是要与 C 语言“100%的兼容”。这个要求决定了 C++语言成为 C 语言的超集。

C++语言受欢迎或许与此有关，C++系统的用户数，从 1985 年的 500，增至 1991 年的几十万。例如 Borland 公司到 1991 年 10 月就售出 C++编译器达 50 万套。

另一方面，C++语言是不断研究和实践改进的结果，在 C++1.0 发表后的十几年中，C++语言的基本概念仍然在不断地补充和形成。

３． C++语言的发展

语言的核心特征是逐步地完善起来的，这也许是 C++语言不同于其它语言的独特之处。

- 在“ C with Class”阶段，在 C 语言的基础上加进去的特征主要有：
  类及派生类；
  公有和私有成员的区分；
  类的构造函数和析构函数；
  友元；
  内联函数；
  赋值运算符的重载；等等。
- 1985 年公布的 C++ 1.0 版的内容中又增加了一些重要特征：
  虚函数的概念；
  函数和运算符的重载；
  引用；
  常量（Const）；等等。

· 1989 年推出的 2.0 版才形成了更加完善的支持面向对象程序设计的 C++语言，新增加的概念包括：

类的保护成员；

多重继承；

对象的初始化与赋值的递归机制；

抽象类；

静态成员函数；

const 成员函数；等等。

· 1993 年的 C++ 3.0 版本是 C++语言的进一步完善，其中最重要的新特征是模板（template）；此外解决了多重继承产生的二义性问题和相应的构造函数与析构函数的处理，等等。

· 1998 年,C++标准（ISO/IEC 14882 Standard for the C++ Programming Language）得到了国际标准化组织（ISO）和美国国家标准局（ANSI）的批准，标准 C++及其标准库更体现了 C++语言的设计的初衷。名字空间的概念，标准模板库（STL）中增加了标准容器类、通用算法类和字符串类型等等使得 C++语言更为实用。

C++语言开发的宗旨是使面向对象程序设计技术和数据抽象成为软件开发者的一种真正的实用技术。所以 C++语言的形成是一个发展和完善的过程。其研制和开发过程是以编译系统的有效实现为前提的，这或许正是 C++语言能够成功的原因。

1.2.3 C++语言的特点

在程序设计语言的历史上，在所有比较成功的高级语言中，C++语言有着不少与众不同的地方，它的这些特点既是人们愿意选择它的原因，同时也是要想学好 C++语言必须搞清的一些问题。

(1) C++语言是支持面向对象程序设计的最主要的代表语言之一。

如前所述，C++语言包括了几乎所有的支持 OOP 的语法特征，基本上反映了 80 年代到 90 年代以来的所有程序设计和软件开发的新思想和新技术。其中包括：

· 封装和信息隐藏。把数据和对数据的操作一起封装到类和对象之中。

· 抽象数据类型。一种新的类的定义就是一种新的抽象数据类型，它可以用在不同的程序系统之中。

· 以继承和派生的方式实现程序的重用机制，为程序的重用找到了一种可

靠而方便的方式。

· 通过函数与运算符的重载，通过派生类中虚函数的多重定义，实现多种情形下的多态特征，明显提高了程序的水平。

· 通过模板等特征实现了类型和函数定义的参数化，把抽象又提升了一级。

C++语言这些典型的 OOP 特征，在 OOP 方法已被软件开发领域接受为新一代开发技术的今天，它不仅已为许多软件开发者所接受，而且也为计算机的专家所接受。C++的 3.0 版本公布之后，一些计算机领域的教科书，如算法与数据结构方面的书中，对于其中的算法与数据结构的描述，已从过去大部分采用 Pascal 或类 Pascal 语言转而采用 C++语言，用类和模板描述算法和数据结构具有通用性，更加简洁合理。

(2) C++语言是程序员和软件开发者在实践中的创造，无论是语言的各个特征还是整个研制过程，时时处处体现了面向实用，面向软件开发者的思想，这样开发出来的系统，比较实用，更加高效，容易受到程序员的欢迎。

从语言的发展史来看，总是比较“务实”的语言最终占上风。例如，在高级语言的初期阶段，虽然 FORTRAN 语言和 ALGOL60 语言都是专家研制的成果，但二者在设计思想上有着明显的区别。

FORTRAN 是 IBM 公司的产品，在设计语言规范的过程中，处处注重编译后的目标码的运行效率以及包括 I/O 系统在内的可执行性。FORTRAN 问世以后，所以得到普遍流行，固然与它是第一个最早的高级语言有关，但其开发中的“务实”观点肯定是其成功的关键。 ALGOL60 语言是由一批计算机语言和编译系统方面的专家作为国际标准的程序设计语言而研制的，它对于语言的语法和语义及编译技术方面确有创造性的发展，对程序设计技术和理论的影响比 FORTRAN 大得多。然而，在目标代码的效率上，在 I/O 功能实现上以及争取大公司的支持上，却输给了 FORTRAN，反而受到冷落。

另一个例子是结构程序设计时期的 Pascal 语言和 C 语言，这又是两个影响大而在设计思想上存在类似区别的语言。

Pascal 语言的设计者是瑞士的 N.Wirth 教授，他本人也曾参与 ALGOL_68 语言的研制。 Pascal 语言完全体现了结构程序设计思想，目标是用于教学和系统程序设计的精致简明的语言，从 70 年代末期开始受到人们广泛重视。至今，Pascal 语言仍是世界各国计算机专业的主要教学语言。然而在软件的开发领域，在与 C 语言的竞争中却处于劣势。C 语言的开发者是美国 Bell 实验室的 K.Thompson 和 D.M.Ritchie，他们也是 UNIX 操作系统的开发者，C 语言是他们为了用高级语言描述和实现 UNIX 而设计的工作语言。为了适合系统软件的开发，C 语言引入了一些低级操作，其表达式简洁，运算符丰富，非常注意程

序描述的简洁和目标代码的效率。与 Pascal 语言相比不那么高级(有人称之为“中级语言”),不很抽象,安全性和可读性较差,但在 80 年代的软件开发领域,“务实”的 C 语言明显地占了上风,成了商业软件开发的主流语言。

到了 90 年代,程序设计开始进入面向对象特征时期。Smalltalk 语法和 C++语法相比,仍然是“务实”性不足。例如,其运行效率仍然是明显劣于 C++(有人指出约相差 3～5 倍),这也是 C++语言在软件产业界更受欢迎的根本原因。

(3) C++语言是 C 语言的超集。从 C++ 1.0 版本问世,就确定了“与 C 语言 100%兼容”的宗旨,虽然这样做也为 C++语言带来了一些问题,但利大于弊,这也是 C++成功的关键。

C++与 C 语言相比:

· 从 C 语言中继承了其独有的那种为程序员所喜爱的简明高效的表达式形式;

· 比较容易地解决了目标代码高质量高效率的问题;

· 吸纳下了 80 年代培养的一批高水平的 C 程序员,他们比较自然地转向 C++;

· 可以与 80 年代以来的大批 C 程序软件兼容,可以把它们在 C++环境下继续维护使用。

为了有利于 C++语言的发展,C++语言的设计者和开发商还注意解决与 C 语言相关的问题。一方面,在相当长的时间内,C 与 C++语言及其编译系统同时发售,促进了 C 程序员向 C++语言的转化过程。另一方面,对于 C 语言的语法成分,做了许多有成效的取代工作,例如:

· 引入 const 常量和内联函数概念,取代宏(#define)定义;

· 引入 refrence 引用概念,部分取代过于灵活而影响安全性的指针;

· 引入动态内存分配运算符(new),取代比较低级的有关库函数;

· 引入用于数据 I/O 的流类,取代可读性差的 C 语言 I/O 库函数,等等。

全部包容 C 语言的策略,也造成了一些问题,例如:

· 语法成分中出现一些冗余或重复的现象;

· 同时支持两种程序框架——SP 框架和 OOP 框架;

· 容易引起误解:C++语言是 C 语言的扩集,是 C 语言的改善,实际上在设计方法、思想和风格上,二者有着本质的差别。

要想学好 C++语言,必须把这些问题搞清。

1.3 本书的宗旨及内容安排

虽然 C++语言已问世很长时间，各种介绍的书籍比比皆是，但学有成效者不多。比起学习其它语言，例如 BASIC，Pascal 和 C 语言，学习 C++语言要难得多，必须有决心下功夫，有好的方法指导，有好的教材和上机条件，才能有好的学习效果。

1.3.1　讲授 C++语言的困难

采用面向对象程序设计语言可以设计出好的程序，但学习起来比以前学过的其它语言要困难，这是因为下面的原因。

1．C++语言的规模较大

在最近的 15 年内，人们学习最多的语言大概要算是 BASIC，Pascal 和 C 语言了。直观上讲，学会任何一种语言，也就学会了通过编程令计算机完成各种各样任务的方法。但就语言的规模来衡量，不同的语言差别是相当大的。

最简单的语言是 BASIC 语言，内容少，编程和调试都很简单，教科书不仅讲述详细，而且例子很多，一个打印语句就可以花上几页的篇幅说明。大多数人在学习中不会感到困难。

Pascal 语言比 BASIC 语言要复杂得多，丰富的数据类型，过程和函数的参数调用，指针和记录类型的使用等等，需要学习和讲授的内容几乎增加了一倍。许多人就是从学习 Pascal 语法作为基础，比较容易地学习 C 或 Ada 等更复杂的语言的。

C 语言虽然与 Pascal 同是 SP 语言，但语言中的语法成分却要比 Pascal 增加许多内容。例如 C 语言的运算符几乎多了一倍，循环语句要灵活得多，数组和字符串的处理复杂而多变，C 的指针比 Pascal 指针要自由得多，学好 C 语言，也许要比学习 Pascal、BASIC 语言花费更多的努力。

C++语言是 C 语言的扩集，这种扩充有三个方面：

· 有关面向对象的新概念、新特征，这部分内容在深度上、数量上是不容忽视的。

· 原有 C 语言的概念和特征又补充了新的代替物，如 I/O，const，refrence，等等。

· 新的概念又延伸了老的特征，例如有了类和对象，原有的数组、指针又增加了对象数组、对象指针，原有的函数又有成员函数及对象参数等等。

总之在语言规模上，C++语言与 BASIC，C，Pascal 相比是大很多，C++语言在规模上与普通的 BASIC 语言相比多许多倍，但介绍 C++语言的书却不

能比 BASIC 的书多许多倍，学习 C++语言的时间也不可能多许多倍。

2． C++语言中的新概念不易掌握

OOP 理论是程序设计技术经过 50 年的实践和研究的成果，被认为是成熟阶段的方法论。因此，OOP 语言中的一些重要概念和特征，例如，虚函数，抽象类，构造函数的隐式调用，运算符的重载，类模板和函数模板等等，与语言中的普通概念例如数组、分支语句、子程序和过程等等比较起来更难深入理解和掌握。

这一情况说明，人类关于程序设计方面的认识不断在深化；另一方面，人们的认识能力也总是随时间的延伸而不断地提高，像上面提到的有关程序设计的一些概念，例如变量，数组，循环语句等等，现在大家都认为是很简单的，但在 30 年前，这些概念也是十分难以掌握的。

3．OOP 和 OOP 语言的培训需要完备条件

从传统的程序设计转向 OOP，是一个巨大的转变，程序设计的方法和风格有着本质的不同。因此，向 OOP 的转变比起 30 年前向 SP 的转变更难，可能需要更长的转变期，当然这一转变对于程序设计和软件开发领域的影响也可能更大。

虽然从 Smalltalk80 公布算起，面向对象程序设计技术已快有 20 年了，但必须承认，把 OOP 技术从研究室引到商业软件的开发领域并使大多数人认定 OOP 技术是今后软件开发的主流技术，仅仅是 90 年代以后的事。如何大规模地培训 OOP 人才，则是刚刚提到日程上的问题，下面的现状可以说明这种情形：

· 几乎所有的计算机专业都意识到了面向对象程序设计培训的重要性，但一些学校中的教学模式仍停留在首先学习 Pascal 或 C 语言，并在此基础上学习大多数相关课程(如数据结构，编译方法，操作系统，数据库系统等等)，最后以一个选修课的形式引入 OOP 的理论和语言。这种模式很难达到培养的目标，少数掌握了面向对象技术的高级程序员更多地是靠自己的理解。

· 有关的书籍虽然很多，但大多数难以产生明显的效果，其中一种以 OOP 理论为主，于编程实践的直接指导不够，显得比较抽象，距离较远。另一种介绍编程语言，尤以 C++语言的书居多，但其中大部分是下面的三类书籍：一类是为 C 程序员写的，只讲 C++中不同于 C 的语法成分，往往学习的结果是使 C 程序员学会了使用 C++语法及其编译系统，编出的程序仍然是结构化的 C 程序。另一类书籍虽然包含很多 C++的篇幅，然而却是典型的 C 语言的扩充。第三类 C++的书主要是提供给编程环境与工具的“用户”的，其目标是指导读者如何利用系统提供的标准类和函数库实现诸如“ Windows 编程”这类目的的，它

的目标是指导用户如何利用面向对象技术的可重用的成果。

· 负责程序设计课程的教师虽然主张抓好 OOP 教学，但他们自学比较困难，缺乏大规模的编程实践机会，同时他们发现这种课程开设早了对学生会比较难，开设晚了，又收不到预期的效果。

· 实际上问题还是出在面向对象技术本身。一方面，OOP 技术并不排斥一般程序设计的技巧训练和结构化方法，而 C++语言恰好容纳了这几方面的特征，但这就必然造成学习内容的庞杂。另一方面，OOP 技术是针对大中规模程序的，很难找到短小的实际程序为例，没有实际程序，书就会显得空洞枯燥。传统的程序设计可以通过大量的短小的程序实例和上机练习，进行生动自然的讲授和训练，但面向对象程序设计，任何一个实用的类定义都可能要几百行，这确实是一个不太好解决的问题。

总之，如何有效地培养高水平的 C++程序员，这是一个摆在计算机教育界面前的一个挑战性的问题。本书的编写就是期望在解决这样的重要问题中做一些探索。可以肯定的是，既然 OOP 技术盛行于世纪之交，计算机学科和计算机产业的发展又是必然趋势，那么这个困难就一定会被克服。

1.3.2　本书的指导思想

由于上面介绍的原因，编写一本好的 C++教材不是一件容易的事，下面是作者构思这本书所遵循的思路。

1．训练程序设计技术的教科书。

任何一本讲解编程语言的教材，都同时肩负着两个任务：

- 使学生掌握编程的方法和技巧（读程序、写程序）。
- 讲解编程语言的语法规则（讲语法）。

二者是相互关联的，但写书却有不同的写法。

- 由浅入深地介绍精选的程序实例，同时根据需要讲解相关的语法规则，最终目的是使学生掌握编写较高水平程序的能力；
- 以讲解语法规则为主，每讲一种规则，都安排适当的简单程序实例来说明，最终目的是使学生对 C++语言的规则有全面的掌握。

由于上节中谈到的 C++语言的特点，目前许多 C++（或 JAVA）语言的书，多半是按第二种思路编写的，完全以语法制导，程序用来讲解语法，书中很少有较复杂、较实用的程序，因此其内容比较浮浅，不适于作为讲解程序设计技术的教材。本书的编写，努力按照前一种思路，在许多章节，采用先读程序，再讲有关语法的方式，在书中尽量安排一些有应用背景的大程序（如第十一章

大厦电梯仿真程序)。尽管如此，由于 C++语言的规模和 C++实用程序的规模都很大，实现这一点并不容易。

2．讲解面向对象程序设计技术的教科书。

(1) C++语言是一个面向对象程序设计语言，但它也支持结构化程序设计，因此，有的书中称它是一种“混合”语言。对于编程的学习来说，这是它的一种优点。不过在学习中应特别注意的是，首先要把注意力放在 OOP 方法和风格的学习上。对于在编程中经常碰到的涉及程序结构化和具体编程技巧方面的问题，例如数据类型的检查和转换，函数参数的调用处理机制，数组下标表达式，字符串指针和函数指针等等，当然要搞清、学好，但是又不能把精力全放在这上面(如像学习 C 语言中那样)。为了学好数组的使用，学好 for 语句的使用，学好数组和字符串指针的使用，要做大量的例题，反复理解。但是，在我们的书中，有意地减少这方面的内容篇幅。因为我们主张，这些内容对于高水平的程序员当然是必要的，但是，它们应在掌握 OOP 编程方法和技术的前提下在编程实践中再学习和补充，否则容易造成喧宾夺主，因小失大的结果。

(2) 不宜采用传统的程序设计语言教科书中把语法划分成若干章节，按语法顺序补充实例逐步讲解的方法。一个“捷径”是(当然不是“十日通”那样的速成)按照规模和难度，从易到难选学一些程序设计实例，直接学习编程技术。在实际程序的学习中，掌握最主要的语法规则和碰到的所有语法现象，遇到什么学什么，不要求完善和全部细节，只要求掌握基本的方法和思想，通过实例对语法规则进行深入的理解，把系统全面的语法学习放在第二步，甚至可以在编程实践过程中，遇到了问题再学习语法。

(3) 处理好与 C 语言的关系，不主张先学好 C 语言(或 Pascal 语言)，再补充 C++的特征，最好是把 C++语言作为一个独立的语言来学习。如果已经是熟悉 C 语言程序员，也不建议再找一本专讲 C++与 C 区别那种书，还是选择把 C++语言作为独立的 OOP 语言的书来学习为好。水平较高的 C 程序员容易把转变为 C++程序员看成是简单的事，事实上这种转变还是需要付出努力的。

1.3.3 本书的内容安排

全书共分十一章，大致可分为四部分：

第一部分由第一章绪论组成，由远及近地分别介绍程序设计语言、C++语言以及本书内容。这一部分初学者可能会感到内容较深，但学完其它几部分再读，必有新的理解。

第二部分由第二至六章组成，主要介绍一般的程序设计技术，对于初学者来说，打好基础是最重要的，一定要学好，在讲授的过程中按照由浅入深的原则，分为五章。根据程序设计语言的两大要素即数据的类型和对数据的处理，形成两条主线：

* 第二章介绍预备知识：基本符号，词汇，语句到最简单的程序；介绍如何编辑、编译、连接、运行一个简单的程序的必要知识。其中部分用小字排版，可以在讲授中略过。

* 第三章介绍基本数据类型及其派生类型，其中也包括枚举类型。另外包括由系统提供的基本运算符；它们组成的表达式语句使得程序能够对数据进行运算。

* 第四章引入导出数据类型数组及结构类型，同时介绍了 C++程序的基本控制结构。有了控制语句的 C++程序才有了判断能力，才可以实现各种不同应用。

* 第五章介绍函数概念。它帮助人们编出结构良好的 C++程序。函数是 SP 框架的 C++程序的核心，同时也是 OOP 框架的 C++程序的主要组成部分。

* 第六章是 C++程序中更具技巧性的一部分，通过引入指针和引用类型，加上运算符 new，delete 的配合，更为灵活的 C++程序才编写出来。

第三部分为第七至九章，是 C++程序支持面向对象程序设计的主要部分，对于初学者来说难度较大。

* 第七章讲述类和对象的概念。类一方面是最合理的由数据和函数组成的程序模块，又是完全由用户定义的新的数据类型。

* 第八章是类的继承和派生，其中虚函数等概念体现了 OOP 程序的优越性。

* 第九章介绍模板。模板是类的抽象，读者将在后续课程“数据结构”中看到它的应用。学习面向对象程序设计的能力不仅在本书的学习中，也应该在今后的其它相关课程的学习中不断提高。

本书的第四部分，是最后的三章，目的是向读者介绍实用的 OOP 程序的构造和编程方法。

* 第十章介绍系统提供的 I/O 流类系统，其目标不仅是为了让读者全面掌握 C++程序的输入输出功能，也是向读者展示一个由若干类组成的系统的结构及运作方式。

* 第十一章则介绍实用的 OOP 程序的自顶向下的开发过程，这一章也可以不在课堂讲授，但对于学生学习面向对象程序的设计方法是一个十分有效的参考,建议读者予以注意。

* 第十二章介绍 C++语言提供的异常处理机制，适于在大型程序中使用，

不一定在课堂讲授。

思考题

1. 简述计算机应用与程序设计的关系。
2. 为什么说计算机与人类的其它发明不同？它有什么特点?
3. 你怎样理解程序设计语言？高级语言和低级语言是怎样划分的?
4. 什么是机器语言，汇编语言?
5. 高级程序设计语言的主要特征是什么?学习本书前接触过哪些高级语言?
6. 程序设计技术的发展可分成几个阶段？简述各阶段的主要特征。
7. 为了成为高水平的程序员，应该在哪几个方面进行学习和训练?
8. 谈谈选择 C++语言的理论和原因。
9. C++语言的形成与其它语言有何不同?
10. C++语言受到欢迎的主要原因是什么?
11. 试分析 C++语言与 C 语言完全兼容的利弊。
12. 简述 C++语言的主要特点。
13. 学习 C++语言的主要困难之处在哪里?
14. 如何学好 C++语言?在学习中应注意的有哪些问题?
15. 在学习 C++语言中，应如何处理 C++语言中仍保留 C 语言中的语法成分?

第二章 C++语言初步

程序设计与作家写小说有类似的规律。许多文学家的水平是在阅读大量名家作品和不断的写作实践中形成的。学习用 C++语言编程，自然要掌握 C++语言的语法规则，不过，更重要的是读程序，所有的语法规则都必须与读程序结合起来学习。因此，本章第一节就是给出一个比较完整而又不长的 C++程序。

2.1 初识 C++程序

2.1.1 程序实例

在讲解 C++语言的语法规则之前，我们先给出两个 C++程序实例。

先给出一个简单的 C++程序。

```
//program2_1.cpp
#include <iostream.h>
void main(void)
{
    cout<<"Let´s learn to write a C++ Program.";
}
```

此程序由 6 行代码组成，这是一个完整的 C++程序，其功能是输出一个英语句子,下面对程序的各个部分所起的作用做简单的介绍，有些名词如“编译预处理”,“主函数”,“参数表”等将在后面逐步讲解。

说明：

（1）第 1 行为注释，程序的每行如出现符号“//”，则该行在其右的所有符号为注释。注释是帮助阅读程序的说明，与程序的运行没有关系，在程序被编译时，注释被当作空格处理。此行指出本程序以文件名“program2_1.cpp”存储。有时候注释的内容比较长，用“//”区分注释部分不方便，C++语言还提供另一种在程序中插入注释的方法，即用“/*”和“*/”括起来的部分，无论

几行，都作为注释（见 2.3.5 节说明）。%

（2）第 2 行 #include 是一条编译预处理指令，它告诉编译系统在编译本程序时把系统提供的头文件 iostream.h 的内容插入到第 2 行的位置，它在程序中的作用与第 5 行的输出语句有关。

（3）第 3～6 行是程序的主体，由一个主函数组成。其中 main 是主函数名，第一个 void 指出该函数无返回值。括号( )表示函数，括号内应为函数的参数表，但此函数无参数，故用 void 表示，(void)与空白( )效果相同。

第 4～6 行称为函数体，用 { } 括起来，函数体内可以包含任意多行语句。

第 5 行是本程序中主函数的唯一要执行的任务：向屏幕输出(显示)一串字符组成的英语语句。

cout 是一个标准输出文件名，这里表示屏幕。符号“＜＜”是运算符，它指示计算机把其右端用双引号括起来的字符串输送到 cout 文件即屏幕(或打印机)。由于“cout”,“＜＜”的说明都在系统提供的头文件 iostream.h 之中。因此，凡是程序中需要使用 cout，＜＜等标准输入输出机制时，第 2 行包含的指令就必须列出。

（4） 此程序的执行结果是在屏幕上显示：

```
Let's learn to write a C++ Program.
```

下面给出一个比较复杂的程序实例，它包括了 C++程序的一般结构。当然不能期待初学者一看就懂，但我们首先可以得到一个直观的感觉。

这个程序的功能是求解一道中国古代数学问题，已知公鸡 5 元一只，母鸡 3 元一只，雏鸡三只一元，问花 100 元买 100 只，应各有几只。这个问题可能有多个答案，它很适合用计算机编程求解。

```
//program2_2.cpp
#include <iostream.h>
void main(void)
{
int chicken,hen,cock;
cout<<"chicken    hen      cock"<<endl;
for(chicken=0; chicken<100; chicken+=3)
        for(hen=0;  hen<=33;  hen++)
              if((cock=100-chicken-hen)>-1)
```

```
        if(int(chicken/3)+hen*3+cock*5==100)
        cout<<"    "<<chicken<<"        "<<hen<<"      "<<cock<<endl;
}
```

这个程序已经比上一个程序复杂，有变量、循环语句、分支语句等一些语法成分。简单说明如下：

（1）由#include 开头的第二行是编译预处理命令，用于输出数据。

（2）由 void main(void) 开头的主函数，包括 10 行程序，它是整个程序的主体。

（3）第 5 行定义了三个变量：chicken 表示雏鸡的个数，hen 表示母鸡的个数，cock 表示公鸡的个数。变量及其类型的概念在第三章介绍。

（4）第 6，11 行是类似上例的输出语句，不过这里更为复杂，其中第 11 行的输出语句被执行了多次。有关输入输出（I/O）语句在下一节介绍。

（5）第 7－10 行包含 for(循环)语句和 if（分支）语句，用来对不同数量的雏鸡、公鸡、母鸡分别进行判断，凡是满足“百鸡百钱”条件的数值作为结果输出。具体算法是，令 chicken 取值 0－100，hen 取值 0－33，令 cock 取值为 cock=100-chicken-hen,在这样一个范围内搜索符合条件的解，这个条件就是： chicken/3+hen*3+cock*5=100 .

有关 for(循环)语句和 if（分支）语句的用法在第四章介绍。

（6）这个 C++程序的执行结果是在屏幕上输出：

```
chicken  hen   cock
   75     25     0
   78     18     4
   81     11     8
   84     4     12
```

结果说明这个题有四组解，其正确性不难验证。

可以看到，program2_2.cpp 比 program2_1.cpp 要复杂一些，更复杂的应用程序将在后续章节中见到，例如，在第九章所介绍的 C++程序，包含了 C++语言的大多数知识，需要学习本书的几乎全部内容后，才能读懂和设计。

2.1.2 I/O 语句

将程序中的数据送到外部设备如屏幕、打印机称为程序的输出（output），反之，程序从外部设备获得数据称为输入（input）。

大多数程序都要有数据的输入输出,如 program2_1,2_2 中,都有输出语句，

因此，在系统讲解语法规则之前，先简单地介绍 C++程序中最常用的标准输入输出语句。

1. 标准输出语句

```
cout<<"Let's learn to write a C++ Program.";
cout<<endl;
cout<<"chicken hen cock"<<endl;
cout<<" "<<chicken<<" "<<hen<<" "<<cock<<endl;
```

是四个标准输出语句。cout 是标准输出文件（指屏幕或打印机）的名字，“ << ”称为插入运算符。它们完成向显示器屏幕输出指定的内容。

第一个语句输出一个字符串常量，即在屏幕上显示包括空格和标点符号在内的一个字符串；

第二个语句中，endl 有特定的含义，它要求“ 回车换行 ”，即要求下一次输出数据从下一行左端开始。

第三个语句则连续向屏幕输出两项内容，先是字符串常量"chicken hen cock"，它是输出数据的标题行，然后“ 输出 ”的 endl, 则并不真的输出什么，而是起到了“ 回车换行 ”的作用。它等价于两条语句：

```
cout<<"chicken hen cock";
cout<<endl;
```

的效果。执行后在屏幕上显示：

```
chicken hen cock
```

第四个语句等价于七条输出语句：

```
cout<<" ";
cout<<chicken;
cout<<" ";
cout<<hen;
cout<<" ";
cout<<cock;
cout<<endl;
```

其中，chicken,hen,cock 不是字符串（它们没有引号），而是变量，要输出变量当前的值。

语句 cout<<" "<<chicken<<" "<<hen<<" "<<cock<<endl; 在 program2_2 的运行中共执行了四次，分别印出四行数据，如果取消 endl,结果将印为一行：

```
75 25 0 78 18 4 81 11 8 84 4 12
```

由此可以看出语句 cout<<endl;的作用不是决定向屏幕（或打印机）输出什么内容，而是输出的格式，关于输出的格式，将在第四章和第十章中有详细的介绍。

2． 标准输入语句

下面的程序使用了标准输入语句 cin<<myage;：

```
//program2_3.cpp
#include <iostream.h>
void main(void)
  {
    int   myage;
    cout<<"My age is ";
    cin>>myage;                    //输入年龄（一个整数）
    cout<<myage<<endl;
  }
```

这个 C++程序允许从键盘输入一个整数，例如 21，结果在屏幕上显示：

My age is 21

cin 与 cout 类似，是标准输入文件（指键盘）的名字。“＞＞”称作提取运算符。myage 是一个程序员定义的整型变量名，变量不同于常量，在程序运行中其值可以改变。该语句的作用是把从键盘输入的字符串赋给变量 myage。

3． 标准流类

任何一种高级语言，都必须提供数据 I/O 的机制。Pascal 语言，C 语言等是通过由系统提供的标准 I/O 函数的形式实现的，C++语言除了保留 C 语言的标准 I/O 函数功能之外，提供了一套完善的标准 I/O 流类系统，其结构之合理，使用之方便，远远超过 C 语言的 I/O 函数库。因此，本书只介绍标准流类系统的用法，不过在本节由于准备知识不足，只能就目前程序碰到的部分作简单的讲解，其用法将在后面陆续介绍。而 C++系统的标准流类的结构和机制，将在第十章作为 OOP 程序设计的典型用例予以讲授。

cout，cin 这两个标识符以及“＜＜”,“＞＞”都是系统提供的标准流类中定义的。因此，凡是在程序中使用它们，就必须把对它们有所说明的头文件 iostream.h 包含进来，即：

#include ＜iostream.h＞

是必不可少的。

由于 C++语言保留了 C 语言的所有语法及函数功能，以上介绍的输入输出

方法也完全可以用 C 语言的方式实现。就是利用从 C 语言中继承过来的用于数据 I/O 的库函数，即用 I/O 库函数调用的方式完成输入输出任务，例如上面所举的例子也可以用下面的语句实现：

```
printf（"Let's  learn to write a  C++  Program．"）；
等价于 cout<<"Let's  learn to write a  C++  Program．"；
pringf（"\n"）；
等价于 cout<<endl；
printf（" chicken  hen  cock "）；
等价于 cout<<"chicken  hen  cock"<<endl;
printf（"  %d  %d  %d\n"，chicken,hen,cock）；
等价于 cout<<"  "<<chicken<<"  "<<hen<<"  "<<cock<<endl;
scanf（"%d"，&myage）；
等价于 cin>>myage;
```

如果采用标准输出函数 printf（）和输入函数 scanf（），则应把程序开始的预处理命令改为：

```
#  include〈stdio.h〉
```

这是因为库函数 printf（）和 scanf（）是在头文件 stdio.h 中说明的。

2.2 C++语言的基本符号

所谓语言就是一个字符集合加上一个语法规则集合，按规则组成的符号序列就是文章。

计算机语言同样是一个基本符号集合和一个规则集合的组合，计算机程序就是“文章”。

C++语言是用来编写计算机程序的高级语言，所有的 C++程序都是一个按 C++语法规则组织的由基本符号组成的序列。

本节介绍 C++语言的基本符号集。

2.2.1 基本符号分类

出现在 C++程序中的基本符号可以分为三类：

1．字母

由大小写英文字母

A，B，C，…　，X ，Y，Z；

a，b，c，…　，x ，y ，z 。

共 52 个符号组成第一类基本符号。

2．数字

0，1，2，… ，9。

共 10 个符号组成第二类基本符号。

3．特殊符号

⌣ ，! ，# ，$ ，% ，& ，　，　，' ，" ，(,) ，* ，+ ，，，- ，• ，／，：，；，

＜ ，= ，＞，？，@ ，[，＼，] ，^ ，_ ，{ ，| ，} ，～ 。

共 33 个符号组成第三类基本符号（‘⌣’表示空格）。

三类基本符号共计 95 个，组成基本符号集。

上节中给出的完整 C++程序，就可以视为由上述符号按一定语法规则组成的符号序列。

2.2.2　基本符号的 ASCII 编码

计算机本身不能直接区分不同的字母、数字或特殊符号，它是根据每个符号对应的编码来识别这些基本符号的。

ASCII 是美国标准信息交换码（American Standard Code for Information Interchange）的英文缩写，ASCII 码表把 95 个基本（可打印）符号和 33 个控制字符共 128 个字符与七位二进制数 0000000～1111111 共 128 个数码建立了对应关系，实际上任何一个基本符号在计算机内的表示形式就是这样一个二进制数码。

ASCII 码表见表 2.1。

表 2.1　ASCII 码表

代码	字符	名称	代码	字符	名称	代码	字符	名称	代码	字符	名称
000	NUL	无效	032		空格	064	@		096	‘	单引
001	SOH	标题始	033	!	叹号	065	A		097	a	
002	STX	正文始	034	"	双引	066	B		098	b	
003	ETX	正文尾	035	#	#号	067	C		099	c	
004	EOT	传递止	036	$	币号	068	D		100	d	
005	ENQ	查询	037	%	百分号	069	E		101	e	
006	ACK	信号确认	038	&	与号	070	F		102	f	
007	BEL	响铃	039	'	单引	071	G		103	g	
008	BS	退格	040	(	圆括号	072	H		104	h	

009	HT	水平制表	04l	)	圆括号	073	I		105	I	
010	LF	换行	042	*	乘号	074	J		106	j	
011	VT	垂直制表	043	+	加号	075	K		107	k	
012	FF	换页	044	,	逗号	076	L		108	l	
0l3	CR	回车	045	-	减号	077	M		109	m	
014	SO	移出	046	·	圆点	078	N		110	n	
015	SI	移入	047	／	除号	079	O		111	o	
016	DLE	换码	048	0		080	P		112	n	
017	DC1	设备控制	049	1		081	Q		113	q	
018	DC2	设备控制	050	2		082	R		114	r	
019	DC3	设备控制	051	3		083	S		115	s	
020	DC4	设备停机	052	4		084	T		116	t	
02l	NAK	信息否认	053	5		085	U		117	u	
022	SYN	同步	054	6		086	V		118	v	
023	ETB	块传送止	055	7		087	W		119	w	
024	CAN	作废	056	8		088	X		120	x	
025	EM	纸用完	057	9		089	Y		121	y	
026	SUB	置换	058	:	冒号	090	Z		122	z	
027	ESC	换码	059	;	分号	091	[	方括号	123	{	括号
028	FS	文件分离	060	<	小于	092	＼	反斜扛	124	|	竖线
029	GS	分组	06l	=	等于	093	]	方括号	125	}	括号
030	RS	记录分离	062	>	大于	094	^		126	~	波纹
031	US	元素分离	063	?	问号	095	_	下横线	127	DEL	删除

其中，0～31 和 127 的二进制码对应的是控制字符，例如

013 CR 为“回车”符，(carriage return)

010 LF 为“换行”符，(line feed)

这些字符与计算机的控制有关，严格地说它们并不是 C++语言的一部分。

从 032～126 共 95 个可打印字符构成了基本字符集。

由于计算机的基本存储单元是字节，一个字节有 8 个二进制位，一般一个基本字符的代码存为一个字节。用 8 位编码可表示 256 个字符，因此在不同的计算机中有不同的扩展 ASCII 码表，在扩展的 ASCII 码表中，128～255 号字符一般是不同语言的特殊字符，特殊制表符，及其它特殊符号。增加了这些符号，方便了制表输出和特殊符号输出，但它们仍然不能作为 C++语言的基本符号。在许多高级语言（如 Basic，C ）的书中，可以找到扩展的 ASCII 码表，其中扩展部分只供读者参考，因为其符号与编码的对应可能不是惟一的。

在 C++语言中，有一个关于字符的特别的处理方法，就是它把字符和它的编码混淆在一起，在表示上不予区分。

例如，一个字符类型的数据可以代表该字符本身，也可以代表该字符的

ASCII 码：

```
char  c1  = 'a' ;
int    i=  c1;
 cout＜＜c1＜＜  i;
```

c1 是字符型变量，把字符 a 作为它的值。但如果把 c1 的值赋给整型变量 i 时，赋给 i 的就是字母 a 的 ASCII 码 97。因此输出 c1 和 i 的值时，将分别打印出字母 a 和数字 97。

由此可以看出变量 c1 的值既是 a，也是 97，不需要像在其它语言（如 Pascal，BASIC 等）要有个字符和编码的转换函数。

2.3　C++语言的词汇

单词是语言中的基本语义单元，每一个单词由一个或多个基本符号组成。

C++ 程序中的基本符号都是组合成有一定含义的单词形式出现的，因此也可以说 C++程序是符合规则的单词的组合。

什么是符合规则的单词呢，凡属于下面列出的五类单词，都是 C++程序中合法的单词。

2.3.1 关键字（key word）

关键字是这样一类有特定的专门含义的单词。对于 C++ 语言来说，凡是列入关键字表的单词，一律不得移作它用。因此，关键字又称为保留字（reserved word）。在上节的程序中，void,int,for,if 等单词就属于关键字。

例如，for 是一个关键字，它在 C++ 程序中常常出现，必须用在 for 语句（一种循环语句）的开头。换句话说，在 C++程序中，关键字 for 指明，在它后面的应是一个 for 语句，关键字 for 只有这样一种用法。

再如，const 是另一个关键字，它用在常量说明的开头，它指出在它后面说明的是常量。不过关键字 const 在 C++ 程序中的用法不唯一，例如，const 还可以出现在函数的参数表中，可以用 const 指明某一引用型参数是不被改变的，等等，所以关键字 const 有多于一种的含义和用法。

在本节我们只能列出关键字表（见表 2.2），而大多数关键字的含义和用法应在有关的章节中找到其说明。

关于 C++语言的关键字，有如下说明：

（1）C++ 语言的关键字一般包含了几乎所有的 C 语言的关键字。

（2）随着 C++语言的不断完善，其关键字集也在不断变化。例如，在由 Stroustrup 提出的原始版本的 C++语言中，尚不包括 private，protected 等关键字，至于像 template 等涉及模板概念的关键字，更是在最近的 C++ 版本中所增加的。

表 2.2 基本关键字表

Asm	auto	Bool	break
case	catch	Char	class
const	continue	Default	delete
do	double	elae	enum
extern	false	float	for
friend	goto	if	inline
int	long	namespace	new
operator	private	protected	public
register	return	short	signed
sizeof	static	struct	switch
template	this	throw	true
try	typedef	union	unsigned
using	virtual	void	volatile
wchar_t	while		

（3）各不同版本的 C++ 语言的实现可能有不少涉及其应用领域的关键字的设置，如与微机有关的 far，near 等关键字。还有一些个别关键字，虽然包含于两种不同的版本之中，却有不同的含义，如 volatile 关键字的使用。

总之，关键字集合是使用 C++语言编程前应首先弄清楚的，特别是对少数个别的关键字的设置应有所了解，以免在编程中产生错误，至少应避免在设定标识符时与关键字重名。

在本节中当然不可能介绍所有关键字的作用，这些关键字将陆续出现在以后的章节中，逐渐使读者准确地了解所有关键字的意义和用法。

2.3.2 标识符（identifier）

标识符是由程序员为程序中的各种成分：变量，有名常量，用户定义的类型，枚举类型的值，函数及其参数，类，对象等所起的名字。名字不能随便起，必须符合标识符的组成规则：

（1）标识符是一个以字母或下横线‘_’开头的，由字母、数字、下横线组成的字符串，如：

abcd，c5,_PERSON_H 都是合法的标识符，而 3A，A*B,$43.5A 都是不合法的，一个标识符中间不可插入空格。

（2）标识符应与任一关键字有区别，如 for，if,case 等都不可作标识符。

（3）标识符中字母区分大小写，即 Abc 与 abc，被认为是不同的两个标识符。与此相反，关键字不区分大小写，如 FOR，For,for,foR 都认为是同一关键字，且都不可作为标识符。

（4）标识符的有效长度。如果程序中的标识符过长，系统将对有效长度之外的字符忽略不计，一般 C++语言设其有效长度为 32，用户也可在编译时指定标识符有效长度，其方法是：

在集成设计环境 IDE 下，选择菜单：

option\compiler\source\Identifier Length

在 1‥32 范围内选择一个整数。

在 program2.2 和 program2.3 中，chicken,hen,cock,myage，……都是由用户定义的标识符。

除了符合规则之外，为了在大型程序中区分和记忆，用户在为常量、变量、函数等起名字时，往往不是简单地用 a,b,c,n1,n2….这样的名字，而是使标识符有一定的描述性，表示母鸡数量的变量名为 hen，表示我的年纪的变量名为 myage 或 my_age、myAge,还有一种“匈牙利标记法”，在变量的名字中，不但要表示其含义，还要表示数据的类型，例如，imyAge 表示整型变量，ipmyAge 表示整型指针变量。

2.3.3　字面常量（literal constant）

C++程序中的常量是指固定不变的量。一般常量有两种表示形式：一种称为有名常量，一种称为字面常量。例如圆周率 pai=3.1416，其中 pai 就是一个有名常量，pai 是量 3.1416 的名字，而 3.1416 称为字面常量。

C++程序中有名常量的名字就是一个标识符，而字面常量是一类特殊的单词，它也是程序所要处理的数据的值。

字面常量分为四类：int 型常量，float 型常量，char 型常量和字符串常量。

1. int 型常量

int 型常量即整型常量，实际上就是整数。C++程序中除允许一般的十进制

整数之外，还允许八进制整数和十六进制整数出现。

（1）十进制整数。与一般数学中整数表示完全相同。例如：

4798，-23，0，5，79834

十进整数可以不经任何说明出现在某些表达式等适当位置。不过一般要注意它的大小范围，例如，上面列出的 79834 这个数已超过了一般整数（占 2 字节）的表示范围，但系统可自动按长整数（long int）（占 4 字节）来处理。如果程序中出现的字面整数超过长整数范围，系统将发出警告。

程序员可以在字面整数后加上字符后缀以表示该整数的类型：用 u，U 表示 unsigned int 类型(无符号整数)，用 l，L 表示 long int 类型（长整数），如 435u，41275u，83l，987532L 等等。

（2）八进制整数。以零开头的整数为八进制整数。例如：02，017，0475。

（3）以零和字母 X(x)开头的整数为十六进制整数。例如：0x0,0x4d,0xA3D。

这里要说明的是：

十进制整数的基本字符为 0，1，…，9；

八进制整数的基本字符为 0，1，…，7；

十六进制整数的基本字符为 0,1,…,9,A,B,C,D,E,F 或 0,1,…,9,a,b,c,d,e,f；

例如，下面三个输出语句：

cout<<023； cout<<23； cout<<0x23;

将输出不同的十进整数：19，23，35。

八进制和十六进制数也可以加上 u，U，l，L 字符后缀。

2．float 型常量

float 型常量即浮点常量，浮点常量有两种表示法：

（1）小数点表示法：4.75，2.0，-473.385。一般，用 2 表示整数 2，用 2.0 表示浮点数 2。

（2）科学表示法：1.2e35， -7.37e-3。

$$WeJ = W \times 10^{J}$$

小数点表示法与一般数学中实数（小数）表示一致，科学表示法则是一种计算机表示方法，字符 e 左面为尾数 W，右面为阶码 J，例如

1.2e4= 12000， -7.37e-3＝-0.00737。

一般设尾数为 W，阶码为 J，$Wej = W \times 10^{J}$

C++语言中，浮点常量按 double 类型数据（占 8 个字节）处理，如果浮点常数大小超过了倍精度浮点（double）数据的表示范围，系统将给出提示信息。

如果用户不需按 double 型数据处理，可指定按 float 类型数据（占 4 个字节）处理，方法是在浮点常量后加字符后缀 f，F,例如：3.58f，73e-5F。关于个类型数据的表示范围可在第四章表 4-1 中查到。

3．char 常量

char 常量即字符常量。

字符也是程序要处理的一种数据的形式，用单引号括起来的基本符号就是一个字符常量：

'A', 'g', '3', '!'

事实上，一个字符常量在机内的存储形式仍然是一个整数。在使用 ASCII 码的系统中，字符和对应的整数是一一对应关系。因此，一个字符常量和字符型变量一样，有双重属性，它既是一个字符，又是一个整数。引入字符常量有利于增加系统的可移植性，对于采用不同的字符集编码系统的程序来说，直接使用字符常量就避免了二义性。

C++语言在 95 个字符常量之外，又定义了一些特殊的字符常量，这些字符常量的表示全是用反斜杠“＼”开头的，下面列出其表示及含义。

字符	ASCII 码	含义	
\a	007	响铃	BEL
\b	008	退格	BS
\f	012	走纸	FF
\n	010	换行	LF
\r	013	回车	CR
\t	111	水平制表	HT
\v	011	垂直制表	VT
\\	092	反斜杠	＼
\'	044	单引号	'
\"	034	双引号	"
\?	063	问号	?
\0	048	整数 0	NUL
\DDD	ODDD	八进制整数	
\xHHH	0xHHH	十六进制整数	

表中所列的以反斜杠开头的“字符”在程序中都作为一个单个字符出现，例如：'\n', '\r' ，'\"' 都表示一个字符常量，其在程序中的作用与'a',

'F ' 等相同。

表中最后两项指出，可以用八进制或十六进制数来表示字符常量，例如：

'\62' 表示'2'，其 ASCII 码为 050
'\135' 表示']'，其 ASCII 码为 093
'\x44' 表示'D'，其 ASCII 码为 068
'\x05f' 表示'_'，其 ASCII 码为 095
'\5' 表示'ENQ'，其 ASCII 码为 005

从这些例子可以看出，可以用八进制或十六进制的 ASCII 码直接表示字符,例如，字母 a 可以表示为：' a','\141','\x61'。

4. 字符串常量

用双引号 " " 括起来的字符序列：

```
"string constant"
```

称为字符串常量。

一个字符串常量是一个特殊的字符序列或字符数组，其长度为该字符串中所有字符的个数加 1。原因是除了保存串中字符（包括空格）之外，在最后存一串尾符'\0'。例如，字符串常量"Pascal"的长度为 7。

字符串中也可包括特殊字符，例如：

```
"Can't open the file!\n",
"beep at end of message\07\n"。
```

不过在使用特殊符号'\0'时，会产生问题，系统将把它作为串尾符而忽略了它后面的其它字符。另外在使用\DDD 或\xHHH 形式的字符时，应特别注意避免产生二义性：

'\x4fa'可能难以区分'\4' 'f' 'a'和'\x4f' 'a'；

'\0274'可能难以区分'\02' '7' '4'和'\027' '4'。

字符串常量在程序中应用很多，有关的处理一般与字符数组和指针有关。

2.3.4 运算符（operator）

C++中另一类重要单词是运算符，主要由字母、数字之外的第三类基本符号组成，少数的例外是个别关键字如 sizeof，new，delete，也被列入运算符之列，其余运算符为：

+，-，* ，／，%，==，!= ，＜，＜=，＞，＞=，!，&&，|| ，&，^ ，|，～，++ ，-- ，+=，-=，*= ，／=，%=，＜＜=，＞＞=，&= ，^= ，|= ，？:，=，()，[]，· ，-＞ ，＜＜，＞＞，' ， ::

关于运算符的分类，功能和用法将在第 4、5 章中分别介绍，这里仅作几点说明。

（1）C++语言与大多数常见的高级语言相比，是运算符和运算形式最为丰富的一种语言，学习 C++语言应努力掌握其运算符的用法，特别是其中不少包含混合操作的运算符的用法。例如，前后缀增量、减量运算符++，--的功能和用法，复合赋值运算符+=，-=等的功能和用法。一般情况下，正确使用这些运算符，可使程序简明、清晰。

（2）在上面所列的运算符中，许多运算符是一身兼二任或兼多任，例如：运算符* ：

对于整数 m，n：m*n 表示整数乘法；

对于实数 a，b: a*b 表示浮点数乘法；

对于某类指针 p：*p 表示 p 指向变量的内容。

此外，* 除了作运算符之外，还有另外的功能，例如：

```
int * i, j;
```

这里的* 已不是运算符，可以说它起到一个关键字或分隔符的作用。由此可知，C++语言中单词的分类不是绝对的。

（3）运算符的概念和用法来自数学，不过在 C++语言中运算符和运算的概念对于人们日常习惯的理解已有所扩展。例如，在 C++语言中，赋值（"="），表达式的并列（<表达式>,<表达式>），下标（a [i]中的[]）都一律作为运算和运算符来处理，对此，读者应予以注意。

2.3.5　分割符（separator）

分割符本身没有明确的含义，但程序中却必不可少，一般用来界定或分割其它语法成分的单词称为分割符（separator)。程序中的分割符有点像文章中的标点符号。例如：

";": 表示一个语句的结束。

"": 表示一个字符串的开始与结束。

分割符包括：

⌣（空格)，"，#，(,)，/*，*/，//，'，　;，{,}

其中较为特殊的是空格，程序中空格的使用十分重要。C++程序允许连续的空格出现，从语法功能看，连续多个空格与一个空格作用相同，在两个相邻的关键字或标识符之间起到了分割的作用。另外，在程序中连续的空格可改善程序的格式，提高程序的可读性。

分割符/* ，*/，和//用于注释。/* 与*/应成对地出现，其间的任何符号序列，在程序中等价于一个空格，这就为程序员在程序中加入注释以提高可读性提供了方便。分割符//有类似的功能，它使其后直至行尾的任何字符序列成为注释。

另外，分割符“，”可用来分割并列的变量、对象、参数等，同时“，”也是运算符。

读者可在 program2.1-2.3 中找到分割符的使用实例。

2.4 C++程序的基本框架

上节中，从微观上介绍了 C++程序的最基本的组成单元，本节则从宏观上介绍一个完整 C++程序的基本组成。如第二章中所说，C++语言与支持结构程序设计（Structured Programming）的高级语言 Pascal 和 C 不同，也与支持面向对象程序设计（Object_orinted Programming）的高级语言 Smalltalk，Java 等也不同，它同时具有两种框架，分别支持结构程序设计(SP)和面向对象程序设计（OOP)。

2．4．1 主函数

函数是 C++语言中最重要的概念之一，有关函数的设计还将在第六章详细介绍，主函数也是函数，还有一些细节将在那里介绍。

以 main 命名的主函数是 C++程序中具有特殊性质和功能的函数。我们已经在 program2_1，program2_2 中见到。

在 SP 框架的 C++程序中，main（）函数是整个程序的主控模块。

在 OOP 框架的 C++程序中，main（）函数是整个程序的入口。

（1）主函数是任何一个 C++程序中唯一必不可少的函数。一个最短的合法 C++程序为：

```
main（ ）{ }
```

这 8 个字符是所有合法的 C++程序的交。

（2）主函数的函数名是标识符 main，它是由系统指定的。

（3）主函数的类型（返回类型）为 void 型或 int 型，其中 int 型可以缺省说明。当主函数为 int 型时，它可以返回一整型数值，这个值是传送给操作系统的。

（4）主函数的参数可有下面的两种形式：

①无参：　void　main（ ）或 main（ ）

　　void　main（void）或 main（void）

②有参：　void　main（int　argc，char * argv[]）

　　或 main（int　argc，char * argv[]）

即主函数的形参表也是由系统规定的，用户不得随意设计。

（5）主函数可调用任何其它函数，但它本身不可由任何函数调用，而只能由 C++程序被编译后的执行代码将要在其上运行的操作系统自动调用，实际上它是为系统运行该程序标识出启动地址。因此，主函数也是任一 C++程序运行的执行入口。

（6）主函数不可作其它属性说明，如不可说明为静态（static）的，内联（inline）的等等。

总之，C++程序的主函数是：

① 整个程序的主控模块。

② 程序的入口。

③ 程序和它的运行环境的接口。

主函数以返回值和参数的方法提供了程序和它的运行环境之间交换信息的手段。（在本书中，用小号字排版的部分，建议读者自学，可不在课堂上讲授，下同）

在 main（）取 int 型时，允许程序在退出之前给系统返回一个整数信息。

当 main（）取有参形式时，允许程序应用者通过系统运行本程序时，同时向系统提供几个字符串型的输入信息。此法是为了通过操作系统以命令行的形式，在运行该程序时指定运行参数而设置的。

主函数的有参形式为：　void　main（int　argc，char * argv[]）

　　或 main（int　argc，char * argv[]）

argc：存放命令行中字符串的个数，即命令名加上参数的个数。它总是大于或等于 1（等于 1 的时候相当于没有输入参数）。

argv：字符串数组指针。保存了命令行中输入的各个字符串。argv[0]指向命令行的第一个字符串（命令名），argv[1]指向命令行第二个字符串（第一个参数），以此类推。

例如，设计一个 C++程序 comp，它可以完成对于两个文本文件的比较，指出其差异。在执行 comp 程序时，需同时指出要进行比较的两个文件的文件名，以便在比较时对它们进行读取。可利用带参数的主函数设计方法：

```
//comp.cpp
#include　<iostream.h>
```

```
#include <process.h>
…
main（int argc，char * argv[ ]）{
    if（argc!＝3）{
        cout＜＜"correct command format ："＜＜endl；
        cout＜＜"comp filenamel filename2"＜＜endl；
        return 1；                    //退出程序，并把值 1 返回给操作系统
    }
    cout＜＜"The first file is ："argv[1] ＜＜endl；
    cout＜＜"The second file is ："argv[2] ＜＜endl；
    …                                 //comp 程序的正常运行代码
    …
    return 0；                        //退出程序，把值 0 返回给操作系统
}
```

上面的 comp **程序的运行命令形式为：**

```
＞comp filenamel filename2 ↙
```

例如有文本文件 text1，text2，text3，**可执行** comp **程序：**

```
＞comp textl text2 ↙                    // 比较 textl 与 text2
```

屏幕会显示：

```
The first file is ：text1
The second file is ：text2
…
```

也可以对 text2 **和** text3 **进行比较：**

```
＞comp text2 text3 ↙                    // 比较 text2 与 text3
The first file is ：text2
The second file is ：text3
…
```

如果命令行错：

```
＞comp ↙或
＞comp text2 ↙或
＞comp textl text2 text3 ↙
```

程序将停止运行并输出信息：

```
correct command format ：
comp filenamel filename2
```

应按正确命令行格式重新执行。

2.4.2　预处理命令

C++语言允许在程序中包含若干编译预处理命令，它们都以符号“ # ”开头，指出在对该程序进行编译之前应做的“ 预处理 ”工作。

在程序中加入若干预处理命令，有利于程序的组织和管理，对于较大规模的程序，它有助于各程序模块之间的协调和代码共享。

C++语言的编译系统包含一个附加的编译预处理程序，它是一个文本处理程序，在源程序被正式编译之前，首先进行预处理。其功能是读取和识别程序中的预处理命令，并按预处理命令对待编译的程序进行指定的预处理。下面介绍主要的预处理命令。

在后面的所有程序中，都有预处理命令的使用。因此，初学者对于本节的内容的理解应在今后各章节的程序阅读中逐渐体会。

2.4.2.1　文件嵌入命令

文件嵌入命令（#include）要求系统在编译之前把它指明的文件嵌入到该命令所在位置，其格式为：

#include　< <文件名> >

#include　" <文件名> "

符号‘ # ’：指明其后为预处理命令。

include：关键字 include 专门用于文件嵌入命令。

文件名：要嵌入的文件的全名。要嵌入到程序中的文件的内容当然应是一个 C++程序段，一般是若干说明语句。这样的文件称为“ 头文件 ”，或“ 头部文件 ”，习惯上用扩展名“ .h ”。

当头文件用< >括起来时，要求系统首先在由系统提供的 include 目录下搜索该文件，故当要求嵌入的是系统提供的头文件时，用尖括号括起来较好。

当头文件用双引号"和"括起来时，要求系统首先在源程序所在的当前目录中查找，故当要嵌入的头文件是用户自己设计时，用双引号括起来，这样可减少搜索时间。在当前目录未找到该文件时，再转到包含目录中查找。

如果未查到嵌入文件，将报告出错。

书写嵌入命令应注意：

（1）每一 include 命令只指定一个包含文件。

（2）文件的包含可以嵌套，即被包含文件中又有 include 命令。这时一个嵌

入命令实际上可包含多个文件。应避免互相包含的情形。

（3）文件的包含把两个或多个程序模块组合成一个源程序，作为一个整体进行编译，它与联接（link）的概念不同。

文件嵌入命令在程序的设计和组织中是很有用的手段，它使得整个程序的结构更为清晰。例如：把系统提供的库函数按功能划分为若干组，每个组把与该组的库函数有关的全局量说明和函数原型形成一个头文件。当程序中需要调用该组库函数时，只需把相关的头文件包含进来。在编译时，自然把该组库函数的实现代码联接到一起。这种方式既清楚，又高效，已为人们所接受。

对于用户程序的全局量说明和函数原型，也可以用头文件的形式单独形成一个文件，在所有与它们有关的程序段中，只需用一 include 命令包含即可，这种方法不仅节省了程序的书写的麻烦，在某些常量修改的时候也可以只对头文件进行变动，而不必改动所有程序文件。

顺便指出，include 命令不一定只把扩展名为.h 的头文件包含进来，因为头文件同样是一个.cpp 文件。头文件也不是一个严格定义的概念。实际应用中包含文件以说明语句为主，它是程序的“头”（首部），因而得到了习惯名称。

2.4.2.2 宏定义命令

宏定义（＃define）命令亦称宏替换命令，与其相关的还有取消宏定义（＃undef）命令。

宏定义命令的功能是用一个称为宏名的标识符代表一段字符串，后者称为宏替换体。在预处理过程中，把程序中出现所有该宏名，用替换体取而代之，其格式为：

＃define <宏名> <宏替换体>

例如：

```
＃define   YES   1
＃define   PI   3.14159
＃define   RAD   PI/180
＃define   PRT1   cout＜＜"\n The Begin"
＃define   Max（a，b）   ((a）＞（b）？（a)：(b))
＃define   REGISTER
＃define   ZERO   0.000001
```

其中

符号＃：指明为预处理命令。

define：指明为宏定义命令。

宏名：为一标识符，习惯上用大写字母组成，它代替宏替换体出现在程序中。

宏替换体：为一任意字符串。它可能是一个数字常量，可计算值的算术表达式，字符串常量，也可以是一个语句。

几点说明：

（1）宏定义可以作为常量说明。例如：符号常量 YES 为整数 1，PI 为浮点常量 3.14159，其作用类似于后面介绍的常量说明，但二者是有区别的。

（2）宏名也可以出现在另一个宏定义的宏替换体中，例如：

```
#define  RAD  PI/180
```

当然在这个宏定义之前应已有宏名 PI 的定义。在预处理过程中要经过两重替换。

（3）宏替换体也可以是符号，或字符串，例如：

```
#define  BEGIN  {
#define  END  }
#define  ERROR  "\n  can't find the file！\n"
```

前两个宏定义，使得在程序中可以像 Pascal 语言那样用 BEGIN，END 表示程序块的开始和结束；后一宏定义可用 ERROR 代替字符串"can't find the file！"输出。

（4）宏替换可以是可执行的表达式语句，如上例中 PRT1 的定义，在程序中使用

```
PRT1；
```

实际上是执行一个字符串输出语句 cout＜＜"\n The Begin"；。

（5）宏替换体也可以是空。其作用可在后文中见到，它表示该宏名已经被定义过。

（6）一种复杂的宏替换是带参数的宏替换，其格式为：

```
#define  〈宏名〉〈形参〉〈宏替换体〉
```

其中的形参，可以在程序调用时，用实参（表达式）替换。

下面的例子说明其使用方法：

```
#incunde  <iostream.h>
#define  PUTOUT（k）  cout＜＜k＜＜endl
#define  MAX（a，b）  ((a) ＞ (b) ？ (a)：(b))
void  main（）{
   int  m=2，n=3，p；
   p=MAX（2*m，m+n）；
```

```
    PUTOUT（p)；
    PUTOUT（MAX（4，m))；
}
```

在 5.2 节，介绍了更多的带参数的宏定义的例子，可以看到如此定义的宏替换实际的效果相当于后文中的内联函数，然而它与内联函数在机制上是有区别的。C++语言中的宏定义命令是从 C 语言中继承过来的，由于 C++语言中设置了 const 常量说明和 inline 函数，在大多数情况下，后者更严格、方便，可以取代宏定义。因此，宏定义命令地使用对于 C++语言编程来说，远不如在 C 语言中那样重要，这里的介绍主要为了读者阅读 C 程序，及偶尔在 C++程序中使用。

（7）取消宏定义命令，是用来改变原有定义内容的手段，例如：

```
#define   ZERO   0.000001
#undef    ZERO
#define   ZERO   0.001
```

这样可以在程序内部改变了定义的常量的值。

2.4.2.3 条件编译命令

条件编译命令使得程序可以根据条件来决定哪部分程序参加编译，哪部分程序不参加编译（建议读者自学，可不在课堂上讲授)。这种功能有助于程序员：

(1）避免生成不必要的程序代码；

(2）在调试中加入若干调试语句；

(3）使程序对不同计算机和操作系统具有可移植性。

条件编译命令有四组：

#if 和#endif

#ifdef 和#endif

#ifndef 和#endif

#elif 和#else，#endif

其格式及规则为：

```
#if < 常量表达式 >
    < 程序段 >
#endif
```

常量表达式：由宏常量，const 常量，枚举值或字面常量组成的算术表达式，在预处理时可以计算出表达式的值，值为 0 后面的程序段不参加编译，值非 0 参加编译。

```
#ifdef < 标识符 >
```

< 程序段 >

#endif

标识符：如果该标识符是一个已经定义过的宏名，则该程序段参加编译，否则跳过这一程序段。

#ifndef < 标识符 >

< 程序段 >

#endif

标识符：如果该标识符是一个已定义过的宏名，则该程序段不参加编译，否则参加编译。这一类型的预处理命令经常在实用程序中使用。

#elif < 常量表达式 >

< 程序段 1 >

#else

< 程序段 2 >

#endif

常量表达式：其值为 0 跳过程序段 1，编译程序段 2，其值非 0 则编译程序段 1，跳过程序段 2。

还有一种预处理命令也可以归为条件编译命令之中：错误报告命令#error，它用来指出在预处理过程中，在可能出现错误的地方，输出错误报告，并停止编译预处理过程。其格式为：

#error < 错误信息 >，

例如：

```
#include <iostream.h>
#define WHITE 0
#define BLACK 1
#define COLOR WHITE
void main ( ) {
  #if COLOR==WHILE
      cout<<"Se1ect white stone "<<endl;
  #elif COLOR==BLACK
      cout<<"Se1ect black stone "<<endl;
  #else ;
  #error you select color incorrectly;
  #endif
  …
```

```
}
```

当宏定义在修改后，产生条件编译的错误，如＃define COLOR 2 可使条件编译出错，输出

```
you select color incorrectly
```

并停止编译。

2.4.3 C++程序的 SP 框架

结构程序设计（SP）的基本思想是自顶向下逐步求精，在编程时总是把一个大的复杂的任务逐步划分为若干子任务，形成用一个个相对简单的独立的程序功能模块，在 C++程序中就是函数（在 Pascal 程序中也称为过程）。

例如，为了从某市的车辆档案中具有某些特征的车辆的车主信息，这样的程序首先可以分为三个模块：

A1．处理数据输入，接收输入信息，并送至 A2；

例如，白色，夏利 2000；

A2．在车辆档案中搜索具有指定特征的车辆记录（例如有 146 条），提取相关信息送 A3。

A3．完成所需信息的输出，可以是一个文本文件，或直接显示到屏幕上浏览。

每个模块的任务又可以分为若干子任务，例如模块 A2 又可分为：

A21．根据输入信息确定打开相关数据档案(例如 2000－2004 年的轿车档案)；

A22．检索档案中与输入信息匹配的记录，送 A23；

A23．把匹配记录中的所需信息（例如：车主姓名，身份证号，住址，电话）送 A3；

A24．关闭相关文档。

这些模块又可划分，例如，模块 A22 可分为：

A221．从文档中取出一条记录；

A222．把该记录相关域的数据与输入信息比较，若相同送至 A23；

A223．判断文档搜索是否完成，若完成转 A24，否则读下一条记录，转 A222。

在功能分解过程中，程序员在具体编写每个模块时，所面对的问题是很简单的。

结构程序设计（SP）以函数为核心划分成若干程序模块，在函数与函数之间，通过全局量、函数参数和函数返回值进行数据通讯。从结构程序设计的观

点，一个 C++程序由下面几个组成部分组成。

（1）主函数：它是整个程序的主控模块，也是整个程序的人口和出口，最简单的程序只有主函数。如 program2.1。

（2）用户定义函数：它由一个或若干个函数定义组成，大的程序可以有几十，几百个函数定义，它们是整个 C++程序的主体。

用户定义函数可以为主函数调用，也可为其它用户定义函数（包括它自己）所调用。

用户定义函数可以调用其它用户定义函数和库函数。

SP 程序设计的主要工作是完成用户定义函数的设计。

用户定义函数的设计将在第五章中介绍。

（3）库函数：原则上说系统提供的库函数与用户定义的函数并无区别，因此，也可以把用户定义函数和库函数归为一类。

（4）全局说明：由各个函数共同使用的常量，变量，用户定义的类型（也包括类）的说明，以及已定义的函数的原型组成。

程序的全局说明实际上是各个函数之间的通讯媒介。

程序的这四个组成部分可以有如图 2.1 所示的关系。

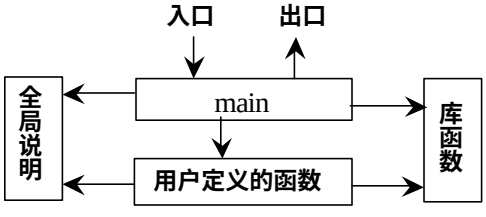


图 2.1　程序四个组成部分之间的关系

C++程序还有另外两个非正式的组成部分，那就是预处理命令和注释。预处理命令已经在 2.4.2 节中介绍。

对于比较大的 C++程序，可以把它们划分为几个程序模块，最简单而有效的划分方法是：

① 根据主函数和各用户定义的函数的功能及相互关系,把它们划分成若干个.CPP 文件。

② 按与每个.CPP 程序文件中的函数有关的全局说明组成一个或多个.h（头）文件。

③ 已有库函数组成的若干.CPP 文件和对应的.h 文件。

在预处理命令的帮助下，一个 C++程序被划分为若干.CPP 和.h 程序模块。在包含命令的帮助下，这些文件形成了一个有机的整体。

（5）注释：一个完整的 C++程序必须包含清楚详细的注释。

虽然从编译的观点来看，它把程序的注释部分视为空格，没有任何作用。但从程序设计和软件开发的角度看，程序中的注释是软件产品的不可缺少的组成部分。

C++程序中的注释，可有两种形式：

① 处于符号“ /* ”和“ */ ”之间的全部符号；

② 在一行中位于双斜杠“ // ”右边的全部符号。

程序员应充分利用加注释的方法，对各程序段作出说明和解释，这将有利于程序的阅读、调试、修改和重用。

注释与程序的功能、编译及运行无关。

有关 C++程序的 SP 框架的实例，将在第五章介绍。

虽然,前面已经指出本章介绍的以函数设计为中心的 SP 框架在当前和今后仍有保留的必要，但是，我们还是要向读者推荐更好的面向对象的程序设计方法。为此，简单地讨论一下 SP 方法的缺点：

（1）SP 方法没有充分利用 C++语言所提供的有力手段。类和对象是 C++语言的主要特征，但在 SP 框架下，类只起到一个配角的作用，它可能出现在全局说明中，作为一种新的用户定义的数据类型出现，而类作为程序模块划分的作用却未得到利用。

(2)以函数为中心对程序进行模块划分,主要是依照程序模块的功能特征，其划分具有相当大的随意性。在大多数情况下，如果划分不科学不合理，自然会影响到整个程序的可读性、可维护性、可重用性等等。

（3）C++语言中的函数没有层次关系，除了主函数之外，所有的函数都是“ 平等 ”的，可以说是一个无序的集合。由这样一些函数组成的模块，其本身没有必然的强内聚性。函数间，模块间的联系较多，不利于程序的编制、调试、修改、扩充和重用。

总之，C++语言仍然支持 SP 框架的程序设计，当开发小规模、短期使用的程序时，这种方式仍然有用。若与面向对象程序设计方式相比，它包含若干不可克服的缺点，学习 C++语言程序设计，显然应该投入更多的精力，努力掌握 OOP 的思想和技术。

2.4.4 C++程序的 OOP 框架

C++语言是面向对象的程序设计语言，是这类语言的主要代表。学习 C++语言的主要目的，就是要学会如何运用 C++语言编出面向对象的程序。

按结构程序设计思想设计的程序，其主体是若干函数定义的集合。按面向对象程序设计思想设计的程序，它的主体是若干类定义的集合：

类说明部分：

　类 C1 的说明，

　类 C2 的说明，

　…

　类 Cn 的说明，

主函数 main（ ），

类成员函数的定义：

　类 Cl 的成员函数定义，

　类 C2 的成员函数定义，

　…

　类 Cn 的成员函数定义。

从程序设计方法学的观点看，OOP 更科学更严谨，是程序设计理论经过五十年的发展历程，从而达到了成熟时期的结果。

面向对象的程序设计把数据和对处理进行数据的过程作为一个整体——对象（Object）。每一个对象是类的一个实例，类是同一类对象的抽象，可以把类比作图纸，对象就是按图纸和具体要求生产的一个个零件。一个运行的系统由许多对象组成，这些对象之间通过信息传递相互作用，形成有机的整体，所传递的信息称为消息（Message），类、对象、消息是面向对象的程序设计的三个基本概念。它们具有三个基本特征：

1．封装性. 好像系统的一个部件（例如计算机系统中的图形卡、声卡、打印机），它是作为一个整体参加系统工作的，其内部工作原理被封装起来，只把功能和基本操作对外公开，便于系统使用，其数据只能通过规定的基本操作进行处理，这就是封装和数据隐藏原理。

2．继承性. 在设计好的类的基础上可以生成派生类,派生类继承原类的设计，又增加新的功能，于是类之间可以有继承和派生关系，有利于简化程序设计。这与函数之间的关系不同。

3．多态性. 一个类的不同对象可以对同一消息产生不同的响应。

通过设计一个个的类，最终组合成一个完整的程序或软件，就好像通过设计一个个部件来设计机床、汽车、电视机、计算机这样的复杂产品。软件是一个复

杂产品，如何把它们分解为若干可重用部件，软件的可重用性是一个程序设计方法学理论的重要问题。由于涉及的内容比较深入，我们只能在本书后面的章节中去介绍。

2.5 运行 C++程序

把设计好的 C++ 程序交计算机运行，并最终获得结果，这一系列的工作主要由计算机自身完成，人只需做一些比较简单的操作。

2.5.1 编辑 C++程序

第一项工作是把程序作为一个文本(字符)文件输入到计算机中。编辑的工作主要包括创建新程序文件(一般是.cpp 文件)，输入，浏览，修改，插入，删除等等。

任何一种文本编辑器都可以完成这项工作，大多数 C++系统例如 Visual C++，Borland C++等都提供了集成开发环境 IDE（Integrated Develop Environment)。IDE 不仅为用户提供了方便的编辑功能，也包括了文件管理、编译、链接和项目管理等多方面的支持。

由于 IDE 为用户提供了编辑、编译、链接、调试等综合环境，例如在编好一个程序后，可以当即编译，运行，再根据编译和运行情况(包括系统提供的出错信息和调试手段)修改程序，使程序迅速调试通过或修改得更好。

2.5.2 编译和链接过程

C++程序不能直接交计算机运行，需经编译系统（Compiler）编译，变成由机器指令组成的可执行程序，然后由计算机运行。

编译前的 C++程序称为源程序，编译后的机器语言程序称为目标程序，见图 2.2。

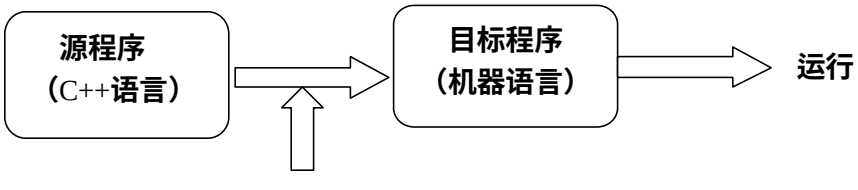


图 2.2 源程序编译为目标程序

实际上从 C++源程序到可执行的机器语言程序还有两个步骤必须完成：

(1) 编译预处理。在源程序被编译之前，须经“预处理”，其任务是根据程序中的预处理指令，完成指定的预处理任务。在本节给出的三个简单程序中都包含一条预处理指令：#include〈iostream.h〉，其具体操作是在系统提供的包含文件库中找出头文件 iostream.h，并把它嵌入到该预处理指令的位置。

(2) 链接（linking)。链接的任务是把已编译好的目标程序与其它需共同运行的目标程序和系统提供的库程序链接起来，形成可执行程序。在上述简单程序中就需要把目标程序与完成键盘输入和屏幕输出的库程序链接起来再去执行。

因此，更详细的步骤应包括：编辑、预处理，编译，链接和调试，见图 2.3。

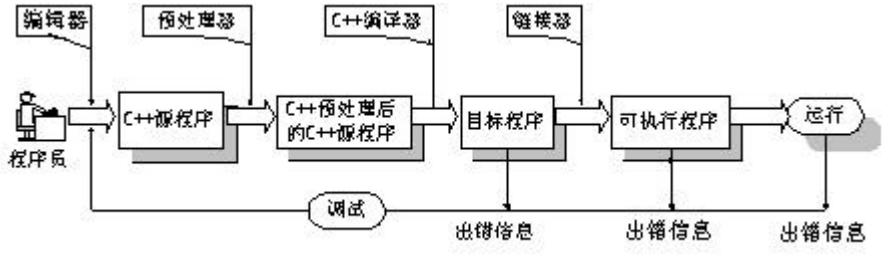


图 2.3 C++程序的编写，编译，运行过程

其中,C++源程序为 .cpp 文件,目标程序为 .obj 文件,可执行程序为 .exe 文件，在调试中根据编译过程、链接过程、运行过程和结果中发现的出错信息，对源程序进行反复修改，最终得到正确的程序和结果。

目前 C++语言的实现系统（编译系统）很多，版本很多，但上述过程是基本一致的，都按下面的方式处理：

(1) 一般用户编写的 C++程序可以按.cpp 文件的形式保存。一个.cpp 文件可以包括一个或多个程序单元，一个程序单元可以存为一个.cpp 文件，也可分存在.h 和.cpp 两个文件中。有的 C++系统的保存形式不是.cpp 文件而是.cc 文件。

(2) 编译预处理不产生中间文件，预处理后自动进入编译，编译后的目标文件一般为.obj 文件。

(3) 经过链接处理的可执行程序记为.exe 文件（.exe 文件是可执行文件)。

(4) 运行这个可执行程序。

2．5．3 运行一个简单的实例

目前,比较流行的 C++编译系统有 Microsoft 的 Visual C++,Inpries(原 Borland)的 C++ Builder(4.*),Borland 的 Borland(Turbo)C++(3.*)等。这里，作为一个实例，介绍 program2_1 在 Visual C++的 IDE 上从编辑到运行的主要过程 。

1. 在“程序”表中选择 Microsoft Visual C++ 6.0 ，进入到 Visual C++ 6.0 的集成环境窗口。
2. 在“File”菜单下选择“new”项（编一个新程序)，出现“new”对话窗口。
3. 在“new”窗口，点击“Projects”标签，在对话框中，选择“win32 console Application”(win32 控制台应用)；在“Projects name”位置下输入项目名称，例如 demo1（把我们的编程工作称为一个名为 demo1 的 Project)；在“Location”位置下，选择工作区空间（指定编程及运行中所生成的文件存放的的位置）的地址，就是确定新建的以 demo1 为名的工作区目录的地址，完成后按“OK”。
4. 在下一对话窗口中，由用户选择项目类型，例如，可选择“An empty project” 完成后按“Finish”按纽。
5. 进入编程环境窗口，窗口分为四部分：上部：菜单和工具条；中右：视图区，是显示和编辑程序文件的操作区；中左：工作区（workspace）显示窗口，这里显示工作中与项目相关的各种文件种类，工作区窗口可用 View 菜单关闭或打开；下部：输出区，程序调试过程中，进行编译、链接、运行中输出的相关信息在这里显示，它也可由 View 菜单关闭或打开；
6. 在上部的菜单中选择“Project/Add To Project/new”，出现对话框，点击“Files”，在出现的文件类型表中选择“C++ Sonrce File”，填入项目名，“demo1”，源文件名“main”和工作区目录的位置路径，按“OK”回到主环境窗口；
7. 在左侧的工作区，点“File View”，打开 demo1 files : Source File /

main.cpp；

8. 这时，右侧的视图区是空的，把 program2-1(或其它源程序)输入其中；
9. 在上方的“ Build”菜单下，选“compiler main.app”，进行编译；查看下部 output 区的输出信息，如有出错信息，修改源程序重新编译；
10. 编译通过后，在上方的“ Build”菜单下，选“ Build demo.exe”，进行链接，如有出错信息，进行修改；
11. 在“ Build”菜单下，选“ Execute demo.exe”运行程序，得到输出结果。

这些操作一般是比较灵活的，上面的 9－11 步可以用其它方式完成，例如：选择“ Start Debug / Go ”可以自动完成上面三个步骤。读者应在大量的操作练习中逐步掌握。

在附录 A 中，我们介绍了 Visual C++ 6.0 的集成环境。读者可以结合该环境学习 C++程序设计。

思考题

1. C++的基本符号有哪些？
2. 什么是关键字？什么是标识符？什么是合法的标识符？
3. 什么是常量？C++中常量的表示形式有哪些？C++的字面常量有哪几类？
4. 从结构程序设计的观点看，C++程序是由哪几部分组成的？它们之间的关系如何？
5. C++程序的主函数与其它函数有什么不同？
6. 什么是预处理？预处理在 C++程序中起什么作用？
7. C++注释的格式有哪几种形式？
8. 简述结构程序设计（SP）的优缺点。
9. 文件嵌入命令的作用是什么？什么是宏定义命令？
10. OOP 框架设计的 C++程序是由哪几部分组成的？
11. C++程序处理输入输出的方式有哪些？
12. 与 SP 比较，OOP 有哪些优越之处？
13. 简述从源程序到程序运行中间的各个处理步骤。

练习题

1. 选择题

(1) 标识符中，( ) 符号不能组成标识符。

A.分割符　　B.下划线　　C.大小写字母　　D.数字字符

(2) 下面标识符中，( ) 是不合法的标识符。

A.Pad　　B.max　　C.P#d　　D.t1mp

(3) 下面标识符中，( ) 是合法的标识符。

A.b-b　　B.CCP　　C.Scanf　　D.*jer

(4) 下面哪些不是转义字符？

A.\\　　B.\t　　C.\x11　　D.\ff

(5) 预处理命令都是以 ( ) 开头

A.*　　B.#　　C.：　　D./

2. 在计算机上对程序 program2_1.cpp，program2_2.cpp 进行编辑，编译，运行。

3. 写出下面程序的输出结果。

(1)

```
#include <iostream.h>
void main()  {
cout<<"Hello,everyone!"<<"  ";
cout<<"How are you!"<<"\n";
cout<<endl<<"Goodbye!";
}
```

(2) //testmain.cpp

```
#include <iostream.h>
void main (int  argc, char * argv[ ]) {
   if (argc > 2){
     cout<<"Error: only one parameter.";
     return;
   }
   if (argc == 2)
     cout<<"The parameter is : "<< argv[1];
   else
     cout<<"There is no parameter";
}
```

第三章 基本数据类型与基本运算

虽然上一章已经介绍了有关 C++语言的许多内容，但是，我们还只能写出像 Program2.1 那样印出一句话的程序。写程序主要是对数据进行计算或处理，本章引入 C++语言支持的最基本的数据类型和最基本的运算，也就是系统提供的运算符。

让我们首先看两个运用运算符对基本类型的数据进行计算的程序实例。

3.1 包含简单计算的 C++程序

3.1.1 程序实例:求两数之和

```
//program 3-1.cpp
#include<iostream.h>
void main()
{
    int a,b,sum;
    a = 43;
    b = 37;
    sum = a + b;
    cout << "The sum is " << sum;
    cout << endl;
}
```

运行结果：

```
The sum is 80
```

说明：

(1) 第 5 行的说明语句把三个标识符 a，b，sum 定义成三个 int 型变量。

(2) 第 6～7 行把两个整数 43 和 37 赋给变量 a，b，于是变量 a，b 的当前值为 43 和 37。

(3) 第 8 行先令 a 和 b 相加，然后把结果值 80 赋给变量 sum。在 6～8 行出现的符号“=”称为赋值运算符，与数学中的符号“=”不完全一样，它是令其右边的表达式求值，然后把值赋给左边的变量。这里的“+”和“=”都称为 C++的算术运算符。

(4) 此程序执行的结果是在屏幕上显示：The sum is 80

(5) 第 10 行中 endl 要求在屏幕上输出回车符进行换行。

3.1.2 程序实例：计算圆面积

上节的程序没有多少实用价值，因此我们再引入一个计算圆面积的程序：

```
//program 3-2.cpp
#include <iostream.h>
void main()
{
    const float pai = 3.14;
    float radius;
    cout <<" Enter radius:";
    cin >> radius;
    float area = pai * radius * radius;
    cout << "\n The area of circle is ";
    cout << area << endl;
}
```

运行结果：

```
Enter radius:4.5
The area of circle is 63.585
```

说明：%

(1) 主函数 main（）前的 void 指明不必返回参数(函数的返回类型亦称该函数的类型)，C++规定不指明返回类型的主函数为 int 类型，即应在主函数执行完返回一整数值给调用它的操作系统。%

(2) 第 5 行说明了一个浮点型的常量 pai，并定义其值为 3.14，由于 pai 是常量，故在程序运行中，这个值不会改变。%

(3) 第 6 行和第 9 行说明了两个浮点（float）型的变量 radius 和 area，后者立即被赋值，其值是圆面积的计算结果。%

(4) 第 8 行是输入语句，cin 也是在 iostream.h 说明的标准输入文件，它指的是键盘，也就是说，程序执行到语句 cin＞＞radius；时它将等待，在用户从键盘上键入一个浮点数——例如键入：2.5 后，把输入的浮点数 2.5 送入变量 radius，相当于为变量 radius 赋值 2.5。

(5) 第 7 行和第 10，11 行都是向屏幕输出字符串，该字符串由双引号括起，按原样输出，其中第 10 行的字符串中开头“＼n”是由反斜杠“＼”和字 符‘n’组成的，是特殊字符，它是不可打印的控制字符。＼n 要求回车换行，即 cout＜＜"＼n"；其效果与 cout＜＜endl；相同。

(6) 此程序的运行结果将随输入数据的变化而不同，若输入半径值改为 2.5，则在屏幕上显示出：%

```
Enter radius:2.5
The area of circle is 19.625 %
```

这个程序与 program3_1.cpp 比较有一个明显的优点，就是它的输人数据不必在编程时确定，而是在运行时由操作者直接从键盘输入。

本章将介绍 C++语言提供的基本数据类型和运算符，以及如何使用它们编出简单的程序。

3.2 基本类型及其派生类型

3．2．1 数据类型的概念

任何一个程序都可以分为三个部分：数据的输入，数据的加工，数据的输出，所以数据是程序运行的对象和结果，是程序设计的第一要素。

一种程序设计语言优劣的首要衡量标准就是它所能提供的数据类型，能不能使用户把客观世界多种多样形态的实际问题方便又灵活地抽象为适当类型的数据加以处理。

C++语言脱胎于结构程序设计语言，因此它继承了 C 和 Pascal 语言丰富多样的数据类型。类型丰富是结构程序设计语言的特点，它为程序员提供方便的选择空间。另一方面，C++语言作为面向对象程序设计语言，它又为程序员根据现实世界的信息特征，自由地设计千变万化的抽象数据类型创造了良好的环境。这一点使得它的应用前景越来越好。

数据类型的概念虽然从有了计算机和程序设计时就已经存在，但早期的程序设计语言如 Basic,Fortran 中没有明确的类型概念，对变量和常量不作类型说明。著名的计算机科学家 C.A.R.Hoare 在 60 年代曾说“类型的信息决定了变量值的表示方法以及所需的内存量，类型的信息还决定了相关的算术运算的意义。”他还曾为类型的概念总结了六条“突出本质”。现在看来，类型概念的几个要点是：

（1）每一项数据应唯一地属于某种类型。

（2）每一数据类型意味着一个有明确定义的值的集合。

（3）同一类型的数据占用相同大小的存储空间。

（4）同一类型的数据具有相同的（允许对其施加的）运算操作集。

C++程序中的数据类型可以如下不同类型的形式出现：

1．基本类型：由系统定义，几乎各种语言都须具备的数据类型，C++在五种基本类型基础上又给出了若干类型的派生类型。

2．用户定义类型：由用户自己根据问题的数据特征，定义所需要的数据类型。是面向对象语言区别于一般结构化程序设计语言的主要特征之一。

作为面向对象语言，C++把类（class）视为其核心概念，引入了类（class）的概念，就使得 C++语言与 C 语言相比发生了本质的变化。

读者不要被“C++语言是 C 的扩集”，“C++的语言是带 class 的 C”等等这样一些表面上的事实和说法所迷惑。一个合格的 C++程序员应该与 C 或 Pascal 程序员在观念上有本质的区别。

class 和 object 的概念在 C++编程中所起的作用，可以从两个角度来分析。

第一，从程序的组织角度。C++通过 class 把数据以及对这些数据进行处理和运算的函数封装为互相关联的程序模块，这与 C 和 Pascal 等语言把程序划分为具有互相调用关系的函数或过程是不同的。

第二，从数据类型的角度，C++通过 class 引入了抽象数据类型的概念，一个由数据成员和函数成员组成的一个类就是一种新的数据类型，C++语言为用户提供了设计反映不同应用背景特征的千变万化的数据类型的可能性。

在 C++程序中，程序员可以根据需要定义多种多样的数据类型。

如 stack（栈），queue（队列），set（集合），complex (复数)， vector（向量），matrix（矩阵）等。

其它通用语言可能设置的类型，C++程序员都可以方便地定义。同时与具体问题密切结合的类型也出现在 C++程序中，如：

window（窗口），menu（菜单），student（学生），employee（雇员），car（小轿车），elevator（电梯），……

于是，编程的过程发生了变化，例如过去的语言，如 Basic,Fortran 等，只有整数、实数、字符等类型，程序员必须把实际问题中的各种数据如温度、坐标、卡片、窗口等化为基本数据类型进行运算，而 C++ 语言则要求程序员直接根据实际问题的需要定义新的数据类型。另外，如枚举类型（还有子界类型）我们也把它归到这一类中，称为用户参与定义的类型。

3． 导出类型：由已定义类型以某种确定的方式产生的新类型，这主要是指数组（array）、结构（struct）、指针（pointer）、和引用（reference）类型。导出类型的使用在实际应用中十分广泛，无论是系统提供的基本类型，还是用户定义的类型（类），都可以产生相应的导出类型。

C++语言从 C 语言中继承了枚举（enum）类型、结构（struct）和联合（union）类型。在 C++语言中，枚举类型实际上是整数（int）类型的子集，而结构和联合与 C 语言中已经不同，它们可以作为特殊的类来处理。

C++语言中的类型可以按下表划分：

表 3_1 C++语言中类型的划分

系统提供	基本类型	int，float，double，char，bool，void
	派生类型	（修饰符＋基本类型）
用户定义	完全由用户定义	class，（struct，union）
	部分由用户定义	enum （int 类型的子集）
由其它类型导出		array，struct，pointer，reference

3.2.2 基本类型

基本类型是具有下面三个特征的数据类型：

- 由系统定义和提供；
- 它们是构造所有其它类型的原始出发点；
- 它们是几乎所有程序设计（不管是哪一代的）语言都包含的。

C++语言的基本数据类型有：int 型，float 型，double 型，char 型和 void 型。

1． int 型

int 型又称整型。是最常用最基本的数据类型。

int 型的数值集原则上是所有整数，实际的值集为计算机所能表示的所有整数。这个范围是有限的，因此程序员在编程时必须经常考虑因数据过大而溢出的问题。

int 型数据占用存储空间依不同的计算机和编译系统而有所差别，目前在 PC 机上运行的各种 C++语言规定 int 型数据占用 4 bytes 即 32 bits 空间（在较早的系统中则占 2 bytes 即 16 bits 空间）。

int 型数据允许算术运算，关系运算等许多种运算。

由于 C++语言从 C 语言那里继承了在类型和运算方面的许多“灵活”的处理，例如 char 型与 int 型的相容，以 int 型的值 0 和 1，或者以 0 和非 0 的值表示逻辑值 false 和 true 。因此，int 型数据也可以参加逻辑运算、位运算等几乎所有的运算。这给程序员提供了某种方便，但也容易造成混乱，应予注意。

2．float 型和 double 型

float 型又称浮点型。Pascal 语言中称为实型，它对应着数学中的实数概念，即带小数点的数。

float 型的值集，原则上是任意大小和精度的小数，实际的值集虽然不可能是任意大小，但由于采用尾数+阶码的表示形式，所以其可表示浮点数的范围可大到$\pm 3.4\times10^{38}$，表示的精度可以小到1.0×10^{-38}。因此，在一般的应用问题中，float 型数据总是可以满足精度和大小的要求，不会出现溢出现象。

float 型数据一般占用 4 bytes，即 32 bits 空间。当精度较高或数值较大时，人们往往使用 double 型，占用 8 bytes,即 64 bits 空间。

float 型数据与 int 型数据的区别在于它们所参加的运算操作类型是不同的。例如在后文将看到增量运算（++，--）可以施于 int 型而不可用于 float 型；在 switch 语句中出现的情况表达式可以是 int 型但不可是 float 型。

3．char 型

char 型又称字符型，即把单个字符作为一种数据处理。

char 型的值集是全部基本字符，ASCII 码集或扩充的 ASCII 码集对应之全部符号。

char 型的数据占用 1 byte 即 8 bits 空间。

char 型数据由于与数字 int 型有某种通用关系，而它本身又是一种字符类型，故可参加的运算相当广泛。

C++语言提供的 char 型与 int 型有着密切的关系，C++语言的设计者 B.Stroustrup 虽然把 char 型列为基本类型，但却把 char 型与 int 一起归为用来表示整数的一类，即可以用来表示字符的整型。由于字符集可与单字节整数有一个对应关系（ASCII 码），因此我们还可把 char 型看作是可以用来表示单字节

整数的字符型。下面的程序说明这一情况：

```
void main() {
  char c1='A',c2='t',c3='\t',c4='!';      //'\t'为水平制表符
  cout<<"c1="<<c1<<", int(c1)="<<int(c1)<<endl;
  cout<<"c2="<<c2<<", int(c2)="<<int(c2)<<endl;
  cout<<"c3="<<c3<<", int(c3)="<<int(c3)<<endl;
  cout<<"c4="<<c4<<", int(c4)="<<int(c4)<<endl;
}                              //int(c)把 char 型的 c 变为 int 型
```

输出结果：

```
c1=A, int(c1)=65
c2=t, int(c2)=116
c3=    , int(c3)=9
c4=!, int(c4)=33
```

新版本的C++/C系统增加了双字节字符型wchar_t,用来支持具有大字符集的unicode标准码，它不属于基本类型。

4.bool 型

bool型在新版的C++语言中被列为基本类型，它只有两个值：false,true，表示逻辑的真和假，可参加逻辑运算和作为逻辑表达式及关系表达式的结果。

5. void 型

void型称为无值型。void型是一种较为抽象的概念。在C++语言中它用来说明函数及其参数，没有返回值的函数说明为void类型的函数，没有参数的函数，其形参表由void表示。

void类型的值集为一空集。

有了void类型，C++语言规定，所有函数说明（main函数除外）都必须指明（返回）类型，都必须包含参数说明。

3．2．3 基本类型的派生类型

基本类型经过简单的字长及范围放大或缩小，就形成了基本类型的简单派生类型。

派生类型的说明符由int，float，double，char前面加上类型修饰符组成。类型修饰符包括：

short：短的，缩短字长。
1ong：长的，加长字长。
signed：有符号的，值的范围包括正负值。
unsigned：无符号的，值的范围只包括正值。

在下面的表 3_2 中我们列出了基本类型及其常用的派生类型的情形，由于各种类型字节及取值范围依机器与系统不同而可能有变，故我们需指出所列数据为一般 PC 机上常见 C++系统的取值范围。

表 3_2 基本类型及派生类型的字长及值域

类型名	字长（byte）	取值范围
bool		{false,true}
char	1	-128～127
(signed) char	1	-128～127
unsigned char	1	0～255
short (int)	2	-32768～32767
(signed) short (int)	2	-32768～32767
unsigned short (int)	2	0～65535
int	4	-2147483648～2147483647
(signed) int	4	-2147483648～2147483647
unsigned (int)	4	-2147483648～2147483647
long (int)	4	-2147483648～2147483647
(signed) long (int)	4	-2147483648～2147483647
unsigned long (int)	4	0～4294967295
float	4	-3.4E(+/-)38～3.4E(+/-)38
double	8	-1.7E(+/-)308～1.7E(+/-)308
long double	10	-3.4E(+/-)4932～3.4E(+/-)4932
void	0	{ }

实际使用的派生类型名可简化，如 1ong int 可以用 long 代替，unsigned long int 可用 unsigned long 代替，表中括号内的部分可省略。

从表 3_2 中可发现修饰符 long 没有起作用，long int 与 int 型是相同的。实际上这与系统有关，例如在其它系统中，可能规定 int 类型数据字长为 2 byte 或 3 byte，这时 long 表示的长整型占 4 byte，区别就表现出来了。

3.2.4 enum 类型

enum 类型又称枚举类型，它是一种由用户参与定义的类型。其格式为：

enum <enum 类型名>{<枚举值表>} <枚举变量表>;

enum：关键字 enum 指明为枚举类型说明。

enum 类型名：标识符，由程序员给出的这个具体的枚举类型名。

枚举值表：若干枚举值用逗号分开。

枚举值可有两种形式：

（1）值名，是一个标识符；

（2）<值名> = <整型常量> 。

枚举变量表：这个变量表可以缺省，有此表意味着该类型说明语句兼作变量说明。

枚举变量说明可另写，格式为：

enum <枚举类型名> <枚举变量表> ；

其中关键字 enum 可缺省，变量表中每一表项可以是变量名，也可同时赋初值。例如：

```
enum color {RED= 1, YELLOW, BLUE} c1= BLUE, c2;
enum color  a, b= RED, c ;
enum day { Sun,Mon,Tue,Wed,Thu,Fri,Sat };
enum day1 { Mon=1, Tue,Wed,Thu,Fri,Sat,Sun };
```

关于 enum 类型有四点说明：

（1）一个 enum 类型实际上是 int 型的一个子集，其每一个值对应一个整数。

（2）n 个枚举值全未赋常量值时，它们自左至右分别与整数 0‥n- 1 对应。例如，枚举类型 day 的七个值为：Sun＝0,Mon＝1,Tue＝2,Wed＝3,Thu＝4,Fri＝5,Sat＝6。

（3）若第 i 个枚举值赋常量值为整数 m，则其未赋常量值的后续枚举值分别与整数 m＋1，m＋2，…对应，直到下一个赋了值的枚举值或结束。因此，为枚举值所赋的整型常量值应自左至右递增。例如，枚举类型 day1 的七个值为：Mon＝1,Tue＝2,Wed＝3,Thu＝4,Fri＝5,Sat＝6,Sun＝7。

（4）枚举类型变量只能赋予其值表中的值，且不能直接赋予数值。例如，

c2＝Fri；

c2＝3； 都是错误的。

（5）枚举类型的说明亦可作为成组说明若干整型符号常量的方法。enum 类型是从 C 语言继承下来的，在 C 语言中它被列为用户定义类型，严格他说，它不是完全由用户定义的。在 C++语言中以类说明的

形式定义的类型才是完全的用户定义类型。

3．2．5 定点类型与浮点类型

基本类型及其派生类型可以划分为两大类：整数类型（定点类型）和浮点类型（实数类型）。

整数类型包括：

bool 类型；

enum 类型；

char 类型及其派生类型；

int 类型及其派生类型；

它们的值集合都是整数集合的子集。

实际上虽然 bool 型，enum 型的值用标识符表示，但输出为整数值，例如：

```
void main(){
  bool flag1=false,flag2=true;
  cout<<"flag1="<<flag1<<endl;
  cout<<"flag2="<<flag2<<endl;
}
```

输出结果：

```
flag1=0
flag2=1
```

浮点类型包括：

float 类型及其派生类型 double，long double；

它们的值集合都是实数集合的子集。

后面介绍的有些运算只允许整数类型（定点类型）参加。

3.3 说明语句

3．3．1 语句

语句是 C++程序中的基本功能单元。程序中的任何一个语句都意味着为完成某一任务而进行的某种处理动作。

最简单的语句可能仅由一个或几个单词构成，而一个复杂的语句可能又包括若干语句，有时可长达几十行，几百行。

可以说 C++程序是由若干语句组成，这些语句可以分为四类：

（1）说明语句．程序中所有由程序员给出的名字标识符，包括变量、常量、对象、类、类型、函数、参数等等都要在使用前进行说明或定义。一般地，C++程序中的说明语句并不单纯说明，也往往包含创建和初始化等工作。

（2）表达式语句．亦可称为处理语句。它是程序中要求计算机对数据进行处理和操作的语句。一般语言中，有陈述句和祈使句。说明语句就是陈述句，表达式语句就是祈使句。C++语言的表达式概念与数学表达式的概念基本一致，a+b，(n1+n2)/2 都是表达式，不过，范围有所扩大，例如：

在 program 3-1 和 program 3-2 中，cout << "The sum is " << sum;　cout <<" Enter radius:";　cin >> radius; 是三个输入输出(I / O)语句，　a = 43;　sum = a + b; 是两条赋值语句，它们都是表达式语句，　由表达式加分号“ ;”组成。　语句　float area = pai * radius * radius; 虽然也起到了赋值的作用，但它是一个说明语句。

（3）控制语句．一般计算机完成的任务复杂而多变，很少有程序可以按顺序完成一步、两步、三步…… 就可简单达到目的的情况。控制语句就是程序中用来控制语句执行次序的语句，语句是程序要完成的一个个动作或任务，控制语句是调度者，决定下一步执行哪一个语句，它也是程序中不可缺少的。

（4）复合语句和空语句．严格他说，复合语句并不是独立的一类语句，若干语句用分割符“ { ”和“ } ”括起来组成一个复合语句，所以复合语句又称为“ 块 ”或块语句。至于空语句，由一个分号“ ;”组成，可以看成一个特殊的表达式语句，不做任何事情。

各种语句的用法将在后面详细介绍。

3．3．2　常量和变量

C++程序中的数据可分为常量（constant）与变量（variable）两大类。在程序执行过程中其值不能被改变的数据称为常量，其值可以改变的称为变量。常量又分有名常量和字面常量。无论是常量还是变量，都是数据，都应有确定的类型，并须按所属类型的规定参加运算。

字面常量的类型是根据书写形式来区分的。例如，475，200 是 int 型，3.1416，200.0 是 float 型，’a’，’4’，’@’是 char 型。

有名常量和变量在程序中必须遵循“ 先声明，后使用 ”的原则，程序中出

现的所有有名常量和变量都必须在使用前由常量说明语句和变量说明语句说明。

有名常量的使用，例如：

```
const float pai = 3.1416;
```

在程序中用 pai 代替 3.1416,常可提高程序的可读性和可维护性。

变量是程序设计中最重要的概念，在程序中说明了一个变量，实际上是做了下面几项工作：

- 赋给该变量一标识符作为其名；
- 指定其类型；
- 在内存中分配给它一片存放空间（其大小由类型决定）；
- 通过赋初值或赋值语句给它一个当前值。例如

```
float temprature = 37.1;
int a, b=5;
```

3．3．3　常量说明

常量说明的目的是对于有名常量赋予类型和值。

常量说明语句的格式为：

```
const  <类型名><常量名> = <表达式>;
```

例如：

```
const  int  N = 2000;
const  float  pai = 3.1416;
```

必须以关键字 const 开头。

类型名：限定为基本类型（int，float，char，bool）及其派生类型。

常量名：标识符。

表达式：其值应与该常量类型一致的表达式（常量和变量也是表达式）。

由于常量和变量同样要求系统为其分配内存单元，所以可以把有名常量视为一种不允许赋值改变的或只读不写的变量，称其为 const 变量。

C++语言另外还从 C 语言中继承了一种定义常量的方法，即在编译预处理命令中的宏定义（或宏替换）方法，例如：

```
# define  N  1000
# define  pai  3.1416
```

也能起到类似的作用，不过，用宏替换的方法定义符号常量与 const 方式的实现机制是不同的：宏替换是在编译时把程序中出现的所有标识符 N 或 pai 都用 1000 和 3.1416 来替换，这里并没有一个只读不写的 const 变量存在；宏

替换的方式中没有类型、值的概念，仅是两个字符串的代换，容易产生问题。因此，在大多数情况下建议使用 const 常量。

在有些 C++语言的系统中，关键字 volatile 与 const 有关，const 把“变量”pai 说明成不变的，而 volatile 则把“常量”说明成可变的。如

```
const int n=1000;
…
volatile int n;
…
```

后一说明，把不允许变化的常量 n 改变为“可变”的。

在有的系统中，volatile 也用于把 register 变量恢复为 auto 变量。

3．3．4 变量说明

变量是数据在程序中出现的主要形式，在变量第一次被使用前它应被说明。变量说明的格式为：

[<存储类>] <类型名或类型定义><变量名表>；

例如：
```
int size，high，temp= 37；
static long sum；
auto float t= 0.5；
```

存储类：程序员可有五种选择：

auto： 把变量说明为自动变量

register： 把变量说明为寄存器变量

static： 把变量说明为静态变量

extern： 把变量说明为外部变量

第五种选择为缺省，按自动变量处理。[<存储类>] 的方括号表示可以缺省。

有关存储类的说明稍后还有详细介绍。

类型名或类型定义：任何变量说明语句中，必须包含数据类型的说明，不可缺省。在上面的例中：int，long（long int），float 就是变量的类型说明。对于有些用户定义的类型，也可以直接把类型定义本身作为变量的类型说明。例如：`enum color {RED= 1，YELLOW，BLUE} c1= BLUE，c2；`

变量名表：列出该说明语句所定义的同一类型的变量及其初值，其格式为：

变量名表：<变量名>[=<表达式>]，<变量名表>

例如：

```
float c1;
char ch1 = 'e', ch2;
int a, b, c = 5;
```

下面就有关变量的若干概念分别说明如下：

1．全局变量和局部变量

C++语言规定变量说明语句可以出现在程序的任何位置。

全局变量：其说明语句不在任何一个类定义、函数定义和复合语句（程序块）之内的变量。

局部变量：其说明语句在某一类定义、函数定义或复合语句之内的变量。

简单地说，说明语句包含在某一对分割符“{”和“}”之内的变量为局部变量，否则为全局变量。

全局变量所占用的内存空间在内存的数据区，在程序运行的整个过程中位置保持不变。

局部变量占用的空间一般位于为程序运行时设置的临时工作区，以堆栈的形式允许反复占用和释放。

可以把变量比作人，说明一个变量，首先要为其分配地址空间，类似为某人分配住房，分配了永久住房的相当于全局变量，参加一次会议，分配临时房间的相当于局部变量，会议结束时房间要退掉。

2．生存期与作用域

生存期：变量 a 的生存期是指从 a 被说明且分配了内存开始，直到该说明语句失去效力，相应内存被释放为止，称为该变量的生存期。

一个全局变量的生存期是指从它被说明（定义点）开始，直到程序结束。

一个局部变量的生存期是指从它被说明（定义点）开始，直到包含它的最近的一个程序块的结束。

若变量说明在某函数定义内，在函数调用过程中其生存期结束于所在的程序块尾。如果该变量是某一个类的数据成员，则显然其生存期应与该类的对象的生存期一致。（对象说明语句在后面介绍）。

作用域：变量 a 的作用域是指标识符 a 可以代表该变量的范围。

一般作用域与生存期一致，但由于 C++语言允许在不同程序部分为不同的变量取同一名字，因此一个变量名的作用域可能小于其生存期，例如，在下面的程序块中：

```
1    {…
2      int   a, b= 5;            //int 变量 b 的作用域为 2－10 行
3      …                          //int 变量 a 的作用域为 2－5 和 9－10 行
```

```
4       if  f(a)
5         {…
6               float    a=2.0；     // float 变量 a 的作用域为 6－8 行
7               …
8         }；
9         …
10  }
```

上例中定义了三个变量：int 型变量 a、b 和 float 型变量 a。int 型变量 a、b 的生存期是 2～10 行，float 型变量 a 的生存期是 6～8 行，而 int 型变量 a 的作用域却只能是 2～5 行和 9～10 行。在 6～8 行范围内整型变量 a 虽然被完好地保存着，但却不能使用。

3．变量的存储类属性

变量除了划分为不同类型之外，还因存储分配等特征区分为不同的存储类。在 C++语言中把它们划分为以下四类：

auto 变量：用关键字 auto 说明的局部变量，称为自动变量。该变量在程序的临时工作区中获得存储空间，如说明语句未赋初值，系统不会自动为其赋初值，随着变量生存期结束，这段临时空间将被释放，可能为其它自动变量占用。变量的 auto 属性为缺省属性，即不写 auto 与写上的效果相同。

register 变量：用 register 说明的局部变量，称为寄存器变量，该变量将可能以寄存器作为存储空间。register 说明仅能建议（而不是强制）系统使用寄存器，这是因为寄存器虽存取速度快，但空间有限，当寄存器不够用时，该变量仍按自动变量处理。

一般在短时间内被频繁访问的变量置于寄存器中可提高效率。不过本书并不建议经常使用 register 变量，理由如下：

（1）在许多情况下使用寄存器变量效果不明显。

（2）有的版本的 C++编译系统具有对局部变量按某种策略自动决定可否占用可用寄存器的功能，效果比程序员决定可能好一些。

（3）局部变量存于寄存器时它将没有内存地址，可能影响与寻址有关的操作，如寻址运算符＆的操作。因此有的版本 C++语言使用关键字 volatile 来专门说明“非寄存器变量”只可占用内存。

static 变量：用 static 说明的变量称为静态变量，任何静态变量的生存期将延续到整个程序的终止，其要点为：

（1）静态变量和全局变量一样，在内存数据区分配空间，在整个程序运行过程中不再释放。

（2）静态变量如未赋初值，系统将自动为其赋缺省初值 0（NULL）。

（3）静态变量的说明语句在程序执行过程中多次运行或多次被同样说明时，其第一次称为定义性说明，进行内存分配和赋初值操作，在以后的重复说明时仅维持原状，不再做赋初值的操作。

例如：

```
static float r,s = 2.5;
```

static 变量在程序设计中常被用到，由于 static 变量是“永久”占用空间，可以保存函数调用过程中某些局部量的结果，并把它传送到该函数的下次调用之中。

由于静态变量容易浪费空间，所以不宜过多使用。

extern 变量：用关键字 extern 说明的变量称为外部变量。

一个变量被说明为外部变量，其含义是告诉系统不必为其按一般变量那样分配内存，该变量已在这一局部的外面定义。

外部变量一般用于由多个文件组成的程序中，有些变量在多个文件中被说明，但却是同一变量，指出某一变量为外部变量就避免了重复分配内存，产生错误。

4．变量的初始化

C++语法为在变量的说明语句中进行变量初始化提供了方便（与变量初始化相类似的还有对象的初始化和函数参数的初始化，C++语言也同样提供了方便）。除了基本类型及其派生类型的变量初始化比较简单，已在前面的例中介绍之外，对于数组、结构、指针等类型的变量初始化也较易实现。

数组变量的初始化只须用｛和｝把初始值表括起来即可，例如：

```
int  a[4] = {1，2，3，4} ，b[2][3]={ 1，2，2，3，3，4};
int  c[3] = {6，9}，  d[2][2] = {{2，4}，{6，8}};
```

一般把值列出即可，亦可部分元素赋初值，系统按顺序为前面的元素赋初值。高维数组的初值也可用｛,｝再分组。

结构变量的初始化与数组类似，应注意结构元素的类型和顺序，例如：

```
struct  st{int  n，m；float  a；}；
st  stl = {2，3，4.0}，st2 = {0，0，1.0}；
```

指针变量的初始化情况有所不同，一般需要取地址的运算，例如：

```
int  a[4]，b；
int * pa = a+2，  * pb = &b；
```

指针变量 pa 的初值为数组元素 a[2] 的地址，指针 pb 的初值为变量 6 的地址。

有关数组、结构、指针及引用类型的说明和使用将在后文详细介绍。

3．3．5 名字空间（namespace）

名字空间（namespace）亦称命名空间，是C++语言的新标准引入的概念，用来解决大型程序中标识符重名的问题。由标识符命名的变量、常量、函数、类、对象等等可以根据它们的逻辑关系分组，每个组就是一个名字空间，说明一个名字空间的语法格式为：

```
namespace <标识符> { <若干说明或定义> }
```

例如，

```
namespace X {
  int i,j,k;
}
namespace SpaceY {
  int i;
  float a;
  void F1(){
    i = i+3;
  }
}
```

第一个名字空间包含三个int型变量的说明，第二个名字空间包含两个变量说明和一个函数定义。其中两个变量i是两个不同的变量。

名字空间中说明的变量在引用时要加限定符，例如：

```
X::i, X::k, SpaceY::i, SpaceY::F1()
```

这样可以减少程序中出错的机会，符合结构化程序设计的要求。

引用名字空间的变量时都要加上名字空间前缀，会使人感到太麻烦，C++语言又引入使用指令：

```
using namespace <名字空间名>
```

在使用指令说明的范围内，引用该名字空间的变量或函数可省略前缀，例如：

```
namespace X {
  int i,j,k;
}
int k;                //全局变量k
void f() {
```

```
    int j=0;            //局部变量 j
    using namespace X;
    i=52;
    X::k=i+4;           //X 中的变量 k
    j=j+1;              //局部变量 j
}
```

这个例子也说明了名字空间变量与全局变量、局部变量的引用规则。

目前 C++语言中的标准函数库的变量与函数都属于名字空间 sdt，如 cin，cout 等，应写成 std::cin，std::cout，不过，由于在相应的头文件中已包括使用指令：using namespace std; 因此，前缀 std:: 可以省略,若程序中又把 cin 说明为变量，则在相应的作用域 cin 代表该变量，而在输入语句中需加限定符：std::cin>>… 。

名字空间的说明在形式上类似于类的说明，但实际机制是不同的。

3．3．6　类型说明

C++语言除了提供由系统定义的标准数据类型 int，float，double, char，void 及其派生类型（如 short int, unsigned int, long int, long double 等），和导出类型（如数组，指针和引用）之外，还允许用户自己创建新的数据类型，这种用户定义的类型可以分为如下三类。

1．typedef 类型说明

用关键字 typedef 引导的类型说明语句，可由用户为一个已定义的类型赋予一个新的类型名，其格式为：

typedef <已定义类型名> <新类型名>

例如：

```
typedef  int  id_number；
typedef  float  temprature；
```

经过这两个语句说明之后，in_number 和 temprature 就可以作为与 int 和 float 一样的类型名在程序中使用。在大型软件中为了提高程序的可读性，常常采用这种形式。

2．用户参与定义的新类型说明

C++语言规定数据结构的框架和运算规则，由用户定义其细节和类型名。例如 enum（枚举）类型。其类型说明格式为：

enum <枚举类型名> {<枚举值表>} [<枚举变量表>]；

例如：

```
enum  color {
  RED = 1, YELLOW, BLUE
} a = BLUE, b;
```

在后面的程序中可以看到 enum 类型的使用。

3．完全由用户定义的数据类型

C++语言为完全由用户定义新的数据类型创造了条件，只有面向对象语言才有此特征。事实上一个完整的数据类型不仅包括数据的结构，数据的取值范围，也包括该类型允许的运算和操作处理。而作为面向对象语言其主要特征的类（class），正好适应了新类型定义的需要。因此，类的说明语句也就是完全由用户定义的新类型说明语句。对此，还必须向读者指出下面几点。

（1）把类的说明作为一个新数据类型的说明，这一点读者可以在第七章的几个实例中清楚地了解到。如此定义类型的实例一般称为对象，并由此应体会到变量和对象概念的内在联系。当然，二者的区别也是明显的，变量所参加的运算和操作是系统提供的，如整型变量可参加加减乘除运算，无须用户定义，而对象所参加的运算或操作则是在类说明中由程序提供的。

（2）在 C++语言中，类和对象的概念，不单纯是用来创建新的数据类型，这还有作为程序的基本组成模块的功能。因此，有关类的说明，我们把它放在第七章和第八章中详细介绍。

（3）由于 C++语言允许 struct 和 union 中也包含函数成员，因此结构和联合就也具备了类似于类的性质。然而这种类似和重复使得 struct 和 union 反而变得不重要了。一般的 C++程序员仍然是在只有数据成员时采用 struct 或 union，而更多的时候直接用类。

在 C++ 语言中，还有另外一些说明语句，这主要是函数说明、类说明（它也可视为类型说明的一种）、对象说明等，这将在后面介绍。

3.4 基本运算符

3．4．1 运算符和表达式的概念

高级程序设计语言中的运算符与表达式的概念，来源于数学，把两个整数相乘然后再与第三个整数相加表示为：

```
a * b + c
```

这可以说是人类数百年数学研究的一个非常优秀的成果，a*b+c 称为表达式，符号+、*则称为运算符，既简洁又科学。然而，计算机却看不懂它，让计算机完成一个表达式的计算，要编一系列的指令，冗长而难读。

最早的高级语言，例如 FORTRAN 的目标就是把数学中的运算符和表达式的形式引入到语言中，可以说这是第一代高级程序设计语言的主要特征。

表达式由运算分量和运算符按一定规则组成，其中的运算符指明所进行运算的类型，运算分量可以是常量、变量或表达式，单个的常量或变量也是一个表达式。例如，a*b-c，n=n+2，35，d3 等等。按参加某一类运算的运算分量的个数，运算分为单目运算，双目运算以及三目、多目运算。

单目：<单目运算符><运算分量>或<运算分量><单目运算符>

例如：-24，-a，i++

双目：<左运算分量><双目运算符><右运算分量>

例如：a+b，x=a+b（a+b 为右运算分量）

三目和多目：C++语言中只有一种三目运算即条件运算符，而把函数调用视为一般多目运算。

3．4．2 运算类型与运算符

每一种运算区别于其它运算的是：

1. 参加运算的运算分量的数量和类型；
2. 运算结果的类型；
3. 运算的具体操作；

例如：

整型加法运算：有两个 int 型运算分量，运算结果为 int 型，具体操作为两数求和。

小于（<）关系运算：有两个 int 型（或 float 型，char 型）运算分量，运算结果为 bool 值 false 或 true（对应整数 0 或 1），具体操作为对于两个运算分量进行比较。

增量运算（++）：只有一个 int 型运算分量（int 型变量），结果为 int 型。具体操作为取运算分量值，加整数 1，然后把结果赋给作为运算分量的变量。

C++语言为每一种允许的运算提供一种运算符，例如整数加法为‘ + ’，小于关系运算符为‘ < ’，增量运算为‘ ++ ’。

为了符合人们的习惯和节省专用符号，有些运算符号一符多用，即一个运

算符对应于多种运算。例如，一般都知道，‘ * ’是乘法运算符，严格地说，‘ * ’是 int 型乘法运算，float 型乘法运算，char 型乘法运算的运算符，同时它还是单目的取值运算（用于指针类型）的运算符。

一类运算是一个运算类型，具有相同运算分量和结果类型的运算划分为同一类。比如赋值运算，算术运算，关系运算，逻辑运算，位运算等等。

3．4．3 赋值运算

赋值运算是一种双目运算，其形式为：<变量名>=<表达式>

右运算分量为一表达式。

‘ = ’为赋值运算符（与数学中的等号含义不同）。

左运算分量为与表达式类型相同的变量。

赋值运算的操作为：1.计算表达式的值;2.把该值赋给左端变量。例如：

```
a = b + c * a
x = abs (y)
s = s + n
```

在程序中常出现类似于 s=s+n 这样的赋值表达式,C++语言允许采用的更为简洁的方式描述为 s+=n。于是运算符+=表示一种复合的赋值运算；把右侧表达式计算结果加上左端的变量值，然后把和赋给变量。

除赋值运算符‘ = ’之外，C++提供另外十种复合赋值运算符，它们是：

+=，-=，*=，/=，%=，>>=，<<=，&=，|=，^=，

这些符号仅是一种简洁的表示方法，例如：

```
s += n      等价于 s = s + n
p *= a + c 等价于 p = p *  (a + c)
```

后几种赋值运算符的使用，在下文介绍了运算符 %,>>,<<,&,|,^之后，不难按照类似的方式学会使用。

3．4．4 算术运算

算术运算是指 int 型，float 型（也包括 char 型）的数值类数据计算后，得到同一类型数据的运算，包括：

单目的减（-），增量（++）和减量（--）运算。

双目的加（+），减（-），乘（*），除（/）和模（%）运算。

单目减：其形式为：- <表达式>，相当于用-1 乘上运算分量的值。例如：

-a，-（x + y），-sqrt（a * a + b * b）

增量运算有两种形式：

前缀增量：++<运算分量>

后缀增量：<运算分量>++

其中运算分量为 int 型（包括 char 型）变量，具体操作为令变量值加 1

前增量++i 与后增量 i++的区别在于：如果++i 和 i++作为表达式又参加其它运算的话，前者是先令 i 加 1 然后参加其它运算；而后者则是先令 i 参加其它运算，然后再令 i 加 1。

例如下面的几个语句：

```
i ++；        // 结果等价于 i = i + 1
++ i；        // 结果等价于 i = i + 1
a = i ++；    // 结果等价于 {a = i；i = i + 1}
a = ++ i；    // 结果等价于 {i = i + 1；a = i}
```

前缀减量：--<运算分量>

后缀减量：<运算分量>--

上面两种运算规则除了如前所述的加法改为现在的减法外，和增量运算完全相同。增量减量运算是一种复合运算，虽然在程序设计中经常使用，并能使程序简洁高效，不过有利也有弊，由于其前后缀的用法较为复杂，有时在与加法、减法运算符混合使用时容易出现混乱，在编程时应注意避免出现诸如 ++a++, a+++++a 这样的表达式。

双目运算的算术运算符+，-，*，/，%是人们最常用最熟悉的，其格式为：

<运算分量><双目算术运算符><运算分量>

例如：

a + b，a-b，a*b，a/b，a%b

运算分量应为数值类型，结果也应该为同一类型。所进行的具体操作即为相应的加、减、乘、除和取模运算。

关于运算分量的类型，要作下面的几点说明：

1． 两个运算分量应为同一类型，如果不同，应该遵循类型转换原则，即由“短”类型向“长”类型的自动转换，即按照：

char→int→float→double 的次序进行自动转换。例如：

```
int a，b；
float x，y；
x = b * a + y；
```

表达式中 a，b 和 y 虽然不是同一类型，a * b 的结果是 int 型，它相对

于 y 的 float 类型是“短”类型，于是 a * b 的结果转化为 float 型和 y 相加，然后赋值给 x。

2． 两个 int 型数据相除，结果应为 int 型，若商不是整数，也要取整。 int 型与 float 或 double 型相除，结果应为 float 或 double 型。例如：

```
int a = 3, b=2；
float y = 2.0;
```

规定 a/b 的值是 1 而不是 1.5，而 a/y 的值是 1.5。这是因为 a/b 的结果是 int 型，所以取 1。而 a/y 中 y 是 float 型，结果应该是 float 型，所以取 1.5。

3． 取模运算符%主要应用于整形数值计算。a%b 表示用 b 除 a 所得到的余数。例如，47%4 的值为 3，33%19 的值为 14。因此对于整数（int 型和 char 型）来说，除法运算和取模运算有如下关系：

```
a – b *（a / b）= a % b  //这里“ = ”为等号
```

3．4．5 关系运算

C++提供六种关系运算，它们相应的运算符为：

<，<=，>，>=，==，!=

使用的格式为：

<运算分量><关系运算符><运算分量>

例如：

-a<b，x>=y，a==b，x!=y*y

运算分量应该为数值类型（或称算术类型）和指针类型的表达式，运算符两边类型应该相同。运算的结果类型为 bool 型，但实际上仅有两个可能的结果值，也就是逻辑值 false 和 true，这两个逻辑值对应着整数 0 和 1。

关系运算的具体操作，是在计算两个运算分量的值以后对它进行比较操作，符合运算符指出的关系，其结果为 1（true），否则为 0（false）。

C++语言中，bool 型和 int 型都属于整数类型，bool 值 false 和 true 同时具有 int 型的值 0 和 1，整数也可以隐式地转换为 bool 值：非零整数转换为 true,而 0 转换为 false。例如：

```
bool b=7;
int n=true;
```

是允许的，整数 7 被转换为 bool 值 true，赋给 bool 型变量 b，bool 值 true 被转换为 int 型值 1，赋给 int 型变量 n。

3．4．6　逻辑运算

C++语言提供三种逻辑运算符：

!，&&，||

分别称为逻辑非，逻辑与，逻辑或运算。其使用格式为：

<逻辑运算符!><运算分量>

<运算分量><逻辑运算符&&或||><运算分量>

运算分量应该是相同的数值(算术)类型或指针类型的表达式，其运算结果为bool (int)型，且只能取 false (0)或 true (1)。

逻辑运算的具体操作为：

1．计算两边的运算分量的值；

2．以值 0 为假（FALSE = 0 或 F），非 0 为真（TRUE = 1 或 T）；

3．按不同逻辑运算符计算返回值。

■　单目逻辑运算符 !：

分量	0	1
结果	1	0

例如：

```
int a = 2，b = 5；
cout << !a << "    " << ! (b / a – a) << endl;
```

结果输出：

```
0    1
```

■　双目逻辑与运算符&&：

分量 1	0	1	0	1
分量 2	0	0	1	1
结果值	0	0	0	1

例如：

```
cout << (a && b) << "      " << (a && (b / a – a)) << endl；
```

结果输出：

```
1    0
```

■　双目逻辑或运算符||：

分量 1	0	1	0	1
分量 2	0	0	1	1
结果值	0	1	1	1

例如：

```
cout << (a || b )<< "    " << (a || (b / - a) )<< endl；
```

结果输出：

```
1    1
```

一般逻辑运算（以及关系运算==）的运算分量，不取浮点类型，因为在判定浮点数是否为 0 或两值是否相等时容易出错。

3．4．7　位运算

位运算是高级语言中的“低级”运算，其操作的对象是整型数，在机器内部二进制表示的每一位（bit）。C++提供如下六种位运算符：

1．　双目位运算符：&，|，^，>>，<<

2．　单目位运算符：~

它们的使用格式为：

<运算分量> <双目位运算符&，| 或^> <运算分量>

<运算分量> <双目位运算符>>或<<> <整数 n>

<单目位运算符~> <运算分量>

其中运算分量为相同的 int 或 char 型及其派生类型。整数 n 为移位数。运算结果仍是整型。

例如：

```
char a = 39，b = 45；
```

■　按位与运算&：

则 a & b 的值是 37，计算过程是按位进行逻辑与运算：

39	0	0	1	0	0	1	1	1
45	0	0	1	0	1	1	0	1
37	0	0	1	0	0	1	0	1

■　按位或运算|：

则 a | b 的值是 47，计算过程是按位进行逻辑或运算：

39	0	0	1	0	0	1	1	1
45	0	0	1	0	1	1	0	1
47	0	0	1	0	1	1	1	1

■　按位异或运算^：

则 a ^ b 的值是 10，计算过程是按位进行逻辑异或运算：

39	0	0	1	0	0	1	1	1
45	0	0	1	0	1	1	0	1

10	0	0	0	0	1	0	1	0

■ **单目按位取反运算~：**

则~a **的值是** 11011000

■ **按位左移运算**<<：

则 a << 1 **等于** 01001110，a << 3 **等于** 00111000。

■ **按位右移运算**>>：

则 a >> 1 **等于** 00010011，a >> 3 **等于** 00000100。

3．4．8　其它运算

C++**除了把人们习惯的算术运算，逻辑运算等列为它所支持的运算之外，还有许多特别的操作，也统称为运算，且设置运算符。**

■ **条件运算符**

条件运算符是三目运算，运算符是？ :，它的使用格式为：

<**表达式** 1>?<**表达式** 2>:<**表达式** 3>

表达式 1 **为数值表达式，表达式** 2 **和表达式** 3 **为任意表达式。**

其具体操作过程如下：

1． **计算表达式** 1，**如果值非** 0（**真**）**则转**（3）；

2． **计算表达式** 3，**转**（4）；

3． **计算表达式** 2；

4． **完成。**

实际上如果把运算结果送到变量 s **中，则可表示如下：**

```
int a，b，s；
……
s =  （a > b）? a : b;
```

这个语句等价于条件语句（下一章）：

```
if (a > b)    s = a；  else    s= b；
```

■ **指针运算符**

指针运算符包括取地址运算符&**和值引用运算符***，**它们都是单目运算符，使用的格式分别为：**

& <**变量**>

变量是一个已经说明过的变量名，例如：

```
int a，b = 3，*pa，*pb = &b;
```

&b **表示变量** b **的地址。**

* <指针变量>

指针变量是一个指针变量名，例如：

* pb

因为 pb 已经赋初值为&b，所以*pb 表示指针变量 pb 所指向的变量 b。例如：

a = * pb

实际上是把变量 b 中的值 3 赋给了变量 a。

指针运算符的使用将在第六章介绍。

■ 逗号运算符

C++（和 C）语言把逗号‘，’也列为一种运算符，这不是指在变量说明语句中的变量表内和函数说明中的参数表内所使用的‘，’，在那里它是分割符。逗号运算的基本格式是：

<表达式 1>,<表达式 2>

表达式 1 和表达式 2 为任意表达式，运算的结果为表达式 2 的值，运算过程为先计算表达式 1，然后计算表达式 2。例如：

设 i，j，s，t，a，b 为 int 型变量，

if （t = s，a < b）s = a；else s = b；

for （i = 0，j = 10；i < j；i ++，j--）{……}

上面的两个语句（将在下一章介绍的 if 语句和 for 语句）中都使用了逗号运算符组成的逗号表达式。前者用表达式（t = s，a < b）完成了两件事：把 s 的值赋给 t，计算 a < b 的值以决定 if 语句的选择。后者则在 for 语句中巧妙地采用两个循环变量 i 和 j，用逗号表达式 i = 0，j = 10 为它们赋初值，同时用另一个逗号表达式 i ++，j--进行增量和减量。

逗号运算符并未进行实际的数据处理，但它可以使程序简明。

■ 函数调用符

把圆括号（ ）称为运算符并不符合人们的习惯，这是因为 C++语言把函数调用和类型强制转换也归类为表达式。（ ）作为运算符有两种情况。

（1） 用于函数调用，格式为：

<函数名> (<实参表>)

例如：add（a，b），main（）等。

（2） 用于强制类型转换。其格式为：

<类型名> (<表达式>) 或 (<类型名>) <表达式>

例如：int a = 2，b = 3；

cout << a / b << " " << float（a）/ b << endl；

其输出结果为：

0　　0.666

其中关键是 float（a）把 int 型变量 a 的值转变为浮点型，按规定它和 b 的商，也应该是浮点型，从而造成了 a / b 与第二个表达式 float（a）/ b 的结果不同。

■ 字长提取符

字长提取符 sizeof，实际上是一个系统提供的函数。其使用格式为：

sizeof （<运算分量>）

这个单目运算和一般函数的使用有两点区别：

（1） 运算分量是一个变量名，也可以是一个类型名；

（2） 当运算分量是变量名时，括号（ ）可以省略。运算的结果是一整数，该整数表示指定变量或类型的长度也就是字节数。

例如：

```
int a，b；
sizeof（char）          //返回值为 1
sizeof  a               //返回值为 4
sizeof（b）             //返回值为 4
```

sizeof 运算为用户提供了一种了解对不同类型数据在系统内实际分配的内存情况的途径。例如，我们在介绍基本数据类型时曾提出，不同的系统对某些数据类型（如 int 型）所分配的内存大小可能是不同的，这往往造成程序的不可移植性，sizeof 运算的使用，可以有助于解决这个问题。下面的程序利用字长提取符验证表 3.2 所列字长数据是否与你所用的系统一致。

```
void main(){
  cout<<"Number of bytes used:\n";
  cout<<"\t          char: "<<sizeof(char)<<endl;
  cout<<"\t         short: "<<sizeof(short)<<endl;
  cout<<"\t           int: "<<sizeof(int)<<endl;
  cout<<"\t          long: "<<sizeof(long)<<endl;
  cout<<"\t unsigned int: "<<sizeof(unsigned int)<<endl;
  cout<<"\t         float: "<<sizeof(float)<<endl;
  cout<<"\t        double: "<<sizeof(double)<<endl;
```

```
}
```

输出结果：

```
Number of bytes used:
                    char: 1
                   short: 2
                     int: 4
                    long:  4
           unsigned  int:  4
                   float:  4
                  double:  8
```

不难看出，sizeof **运算也是高级语言** C++**中的又一种“低级”操作。**

■ **动态分配符**

new **和** delete **是** C++**语言提供的用于动态数据生成和释放的单目运算符，其格式为：**

new <**类型名**>

new <**类型名**>[size]

new <**类型名**> (<**初值**>)

delete <**指针变量**>

delete [ ] <**指针变量**>

（1） new **运算用来生成一个无名的动态变量，它返回一个该类型的指针值，在程序中利用指针对这个变量进行操作。**

（2） delete **运算用来释放或撤销由** new **生成的动态变量。**

关于动态分配运算符 new **和** delete **的使用，将在第六章介绍。**

■ **数组下标运算符**

数组下标运算符[ ]**和函数运算符（）一样也是** C/C++**语言中的特殊概念。其格式为：**

<**数组名**> [<**下标表达式**>]

其运算的过程是计算下标表达式，并以该值作为下标返回数组的对应元素，其作用相当于一个变量，称为下标变量。例如：

```
float a[100]，i；
for （i = 1；i = 100；i ++） a[i-1] = i；
```

其中 a[i – 1]**就是下标变量。（详见第四章）**

■ **限定运算符**

最后一类没有明确运算含义的运算符是限定运算符：:，.，->，下面分别介绍：

（1）　作用域限定符：: 有两个作用：

1．　用于类的成员，其使用格式为：

<类名>：: <类成员名>

在程序中引用类的静态成员时，或是在类说明外定义该类的成员函数时，都需要使用“<类名>：:”来限定所引用的数据或函数成员的归属。

2．　用于全局变量或名字空间变量，其使用格式为：

：: a　//指明 a 为全局变量

<名字空间名>　：: <变量名>　例如：spaceY：: i

当在某个程序局部说明了与该全局变量同名的变量时，在这个程序块中，此变量名就仅指向局部变量，这时如果需要使用全局变量的话，则要通过限定运算符：: 加以区分。

（2）　成员选择符.和->用来限定对象的成员。一个类（或结构和联合）的对象在引用其成员时，一般有两种方式：

1．<对象名>.<成员(数据或函数)>

例如：

```
point p1，p2；
float fx = p1.getx()；
float fy = p2.gety()；
```

2．<对象指针>-><成员>

例如：

```
point * pp = &p1；
float fx = pp->getx()；
```

限定运算符的使用，读者可结合后文，特别是第七章以后的内容学习。在这里读者只要大致对所介绍的限定符有一个了解就可以了，C++的运算符是十分灵活的，要在实践中熟练使用还要做大量的练习。

3．4．9　运算的优先级

在大多数情况下，程序中出现的表达式包含不止一个运算。这可能是：

（1）　多种运算的重复。例如：

```
size * size * size
```

temp = a，a = b，b = temp，i ++

（2） 多个同类运算的相连。例如：

a + b * c / （a – b）

（3） 多个不同类的运算的混合。例如：

a ++ < = b + 5 && 4.5 * a == *ptr

在这样的一些表达式中，许多运算放在一起，为了避免二义性，必须有一个严格的规定，以保证每一种合法的表达式都只有唯一正确的计算顺序，这就是优先问题。

C++语言中解决优先问题有三种方法：

（1） 括号（）优先

在表达式中可以对于其任意子表达式使用括号，括号内的部分优先计算。例如：

a *（b + c）；

! (p && (q||r))；

（2） 没有括号的地方，不同的运算按照优先级的顺序进行计算，优先级高的先计算。例如：

a + b * c；

！p && q || r；

a ++ < = p[0] + 5 && 4.5 * a == * ptr；

由于算术运算符中 8 种运算分为三个不同的优先级：

- （单目），++，--

*，/，%

+，-（双目）

所以表达式 a + b * c 等价于 a +（b * c）

由于逻辑运算符的三种运算分为三个优先级，按从高到低的次序为！、&&、‖。所以逻辑表达式！p&& q || r 的运算次序也是唯一确定的，它等价于（（！p）&& q）‖ r。

至于第三个混合表达式，我们可以根据下节提供的运算符总表中的优先级信息确定，上式的计算顺序等价于：

（（a ++）< =（（p[0]）+ 5））&&（（4.5 * a）==（*ptr））

运算符的优先级规定保证了这样一个复杂的表达式在计算次序上不需要加上任何括号也可以唯一确定。在下节的运算符总表中，用若干条横线区分出不同的优先等级，在相邻两横线中的运算符具有相同优先级，在不同的组的运算符有不同优先级，上面组的运算优先级大于

下面组的运算优先级。

（3） 具有相同优先级的运算符，按两种顺序规定决定先后次序：

1． 左结合规则：即从左向右依次运算，例如：

a + b – c 等价于（a + b）- c

双目的算术运算符，关系运算符，逻辑运算符，位运算符，逗号运算符都是左结合的。

2． 右结合规则：即从右向左依次运算，例如：

* & array 等价于*（&array）

d = c = 4 * a * a 等价于 d =（c = 4 * a * a）

可以连续运算的单目运算符，赋值运算符，条件运算符都是右结合的。

有了上述决定运算顺序的优先规则，C++程序中出现的任何合法表达式都可以无二义性地唯一的确定计算次序。

3．4．10　运算与运算符小结

在本节我们共介绍了 48 种运算，按不同的优先级列在表中。

运算	运算符	格式	例	左/右结合
类限定	::	<类名>::<成员>	stack::pop()	
全局限定	::	::<全局变量>	::a	
成员选择	.	<对象>.<成员>	st1.push(a)	右
成员选择	—〉	<对象指针>-><成员>	stp->push(a)	右
下标	[ ]	<数组名>[<下标>]	array[i+1]	
函数调用	( )	<函数名>(<实参表>)	sin(x)	
后增量	++	<变量>++	a++	右
后减量	--	<变量>--	a—	右
变量长度	sizeof	sizeof<变量>	sizeof (a)	
类型长度	sizeof	sizeof<类型>	sizeof(char)	
取地址	&	&<变量/对象>	&a	右
指针引用	*	*<指针>	*pa	右
单目减	-	-<变量>	-a	右
前增量	++	++<变量>	++a	右
前减量	--	--<变量>	--a	右
逻辑非	!	!(<逻辑表达式>)	!(a>b)	右
按位取反	~	~(<整数表达式>)	~(i+3)	右
动态分配	new delete	new<类型>	new char	

动态撤销 数组撤销	delete	delete<指针> delete[ ]<指针>	delete pc delete[ ] pc	
乘法	*	<表达式>*<表达式>	a * b	左
除法	/	<表达式>/<表达式>	a / b	左
取模	%	<表达式>%<表达式>	a % b	左
加法	+	<表达式>+<表达式>	a + b	左
减法	—	<表达式>-<表达式>	a – b	左
左移	<<	<整型量><<<表达式>	a << 5	左
右移	>>	<整型量>>><表达式>	a >> i	左
小于	<	<表达式><<表达式>	a < b	左
不大于	<=	<表达式><=<表达式>	a <= b	左
大于	>	<表达式>><表达式>	a > b	左
不小于	>=	<表达式>>=<表达式>	a >= b	左
等于	==	<表达式>==<表达式>	a == b	左
不等于	!=	<表达式>!=<表达式>	a != b	左
按位与	&	<整表达式>&<整表达式>	a & b	左
按位异或	^	<整表达式>^<整表达式>	a ^ b	左
按位或	|	<整表达式>|<整表达式>	a | b	左
逻辑与	&&	<表达式>&&<表达式>	(a>b)&&c	左
逻辑或	||	<表达式>||<表达式>	(a>b)||c	左
条件	? ：	<表达式>?<表达式>:<表达式>	a>b?a:b	右
赋值	=	<变量> = <表达式>	a = a + b	右
乘赋值	*=	<变量> *= <表达式>	a *= b	右
除赋值	/=	<变量> /= <表达式>	a /= b	右
模赋值	%=	<变量> %= <表达式>	a %= b	右
加赋值	+=	<变量> += <表达式>	a += b	右
减赋值	-=	<变量> -= <表达式>	a -= b	右
左移赋值	<<=	<变量> <<= <表达式>	a <<= b	右
右移赋值	>>=	<变量> >>= <表达式>	a >>= b	右
与赋值	&=	<变量> &= <表达式>	a &= b	右
或赋值	|=	<变量> |= <表达式>	a |= b	右
异或赋值	^=	<变量> ^= <表达式>	a ^= b	右
逗号	,	<表达式>,<表达式>	a = 3,a ++	左

优先级表包括了几乎全部运算符，基本上解决了复杂表达式的运算顺序唯一性问题，不过，在许多细节情形中，并不能完全解决问题，下面让我们简单谈一谈 C++语言的语法和实现它的编译系统的关系。

3．4．11 关于 C++语法及其实现系统的注释

像汉语、英语这样的自然语言，一方面有其自己的语法规则，另一方面又允许有许多不同的方言存在。汉语的不同方言可以有很大区别。

C++语言作为一种程序设计语言，有一套标准的语法规则，同时它又允许各种不同的实现（编译）系统在语法规则之外的细节上有所区别。有经验的程序员知道，要了解这些细节，不能从教科书中学习，必须在上机实践中逐步掌握。下面是这类的例子：

负整数的除法，例如：-11/2　C++的规则允许取值为接近商-5.5 的整数-5 或-6, 编译系统可以决定取为-5 还是-6。程序员应通过上机确定。弄清这一点也影响到取模运算%，例如，为了确定 7%-2 的值，首先应确定 7/-2 的值：

如果 7/-2 = -4,　则由公式 a%b = a-b*(a/b) 得到 7%-2 = -1

如果 7/-2 = -3,　则由公式 a%b = a-b*(a/b) 得到 7%-2 = 1

当编程中遇到负整数的除法或取模运算时，程序员必须通过测试确定。

逻辑与运算&& 的处理也会产生问题，有的系统采用“短路规则”，在计算表达式 a && b 时，若 a 的值为 false, 则不再计算 b 的值，直接得到表达式 a && b 的值为 false。这样就会产生二义性，例如：

```
int a=3, b=4, c=5, d=6, e=7;
b=(a<b) && (c=d>e);
cout<< a <<" "<< b <<" "<< c <<endl;
```

在正常情况下，输出的 a,b,c 的值为：3, 0, 0; 但是，如果系统采用“短路规则”，输出的 a,b,c 的值变为：3, 0, 5，而“短路规则”当然不是 C++语言的规定。

思考题

1. C++语言中的基本数据类型有哪些？它们的含义是什么？
2. 基本类型的派生类型有哪些？
3. 怎样使用用户定义类型？
4. 导出类型有哪几种?怎样使用导出类型？
5. 请你构想一下是否 C++提供的数据类型已经能够表示这个大千世界？
6. 常量和变量的区别是什么？为什么要区分常量和变量?使用常量和变量有什么好处？
7. 在变量的声明和使用过程中，计算机都作了那些工作？
8. 为什么变量会有全局变量和局部变量之分?为什么变量会有作用域和生存期？变量的作用域和生存期是否一致？
9. 试述==运算符和=运算符的不同。
10. 你现在是否能够通过使用定义自己的数据类型来表示生活中的一些事物(例如房屋、动物、人等)？

11．概述算术运算符、关系运算符和逻辑运算符的相对优先级关系。

习题

1．设有如下的说明：

```
int i = 8, j = 3, k, a, b;
unsigned long w = 5, u;
double x = 1.42, y = 5.2, t;
```

请判断以下表达式是否正确，如果正确请写出表达式的值。

```
k = i++, w += -2, y += x++, i /= j+12, k = --i, f = 3/2* (t=20.2-32.1)
k= (a=2, b=3, a+b), a = 2*a = 3, ++ (i+j), 2% (-3), -2% (-3)
```

2．已知条件：int x = 3, int y = 1, int z = 32, float f = 3.1

有如下表达式:

```
(x--) + (x++) + (++x) + (--x);
x|| (x-y) && f && !(f||x)
(x<<y) & z |～ (z>>y|x)
x ^ y & ～z
```

试说出这些表达式的结果。

3．分析下列程序的运行结果：

程序（1）：

```
#include <stdio.h>
main(){
  int i = 3, k;
  printf ("%d,%d\n", (i++) + (i++) + (i++), i);
  i = 3;
  printf ("%d,%d\n", ++i+ (++i) + (++i), i);
  i = 3;
  k = i+++i+++i++;
  printf ("%d,%d\n", k, i);
}
```

程序（2）：

```
#include <stdio.h>
main(){
  int a, b, c;
  long int d;
```

```
    b = (a = 32767, a+1);
    c = d = 0xffff;
    printf ("a=%x,b=%d,c=%d,d=%d\n", a,b,c,d);
}
```

4. 已知 int i = 32, j = 1, k = 3，确定下面的复合关系测试产生真或假。

```
!(!(True||False))
(True && False) && !(False||True)
!(True && False) || (True||False)
True|| (False && False) ||False
(i && j) && k
34 – i||k
j != k && i !=k
```

5. 将下面的表达式加上括号，使得其计算顺序不变。

```
a ++ b ++
a = b = c
a ++ > b +++ c || 4.3 * a = b *2 + c
a = b++, b = c++
a && b + c < 2||d++ == 4
```

6. 编写一个程序，当用户输入两个时间以后，求出这两个时刻的时间差（按秒计算）并打印到屏幕上。

7. 设计一个简单的英文加密程序，加密规则如下，对每一个英文字母用该字母后面的第 2 个字母和这个字母本身进行异或运算，设置 9 个变量，分别存储字符串 WeiXiong 中的每一个字符，进行运算，检查输出结果和理论计算的结果是否一致。

第四章 基本控制结构与导出数据类型

所谓程序设计或编程就是对一系列要计算机完成的命令的先后次序进行安排，上一章介绍的程序是一种单纯按书写顺序执行的程序，是简单结构程序，其执行规则是：

执行顺序＝书写顺序

其程序模式为：

S_1；

S_2；

……

S_n；

执行顺序亦为 S_1；S_2；…；S_n。

这种程序虽然写起来简单，但

（1）无“智能”可言，它不会根据情况进行判断，决定不同的处理，所有的操作的顺序都是由程序员在编程时安排好的。而几乎所有的实际计算机应用都要求计算机具有应变的能力，因此执行简单顺序程序的计算机也就不能被称为“电脑”了。

（2）写多长的程序就执行多长，在目前每秒钟能完成上百万次浮点运算的计算机面前，程序的长度该有多长呢，编程变成人类无法完成的任务。

因此，这种简单程序实际上几乎没有实用价值。本章介绍 C++语言提供的控制语句可以改变程序的这种简单结构，例如，分支语句可以根据程序执行到某一步的具体情况决定下一步转向不同的程序段；循环语句则可以令计算机反复执行某一段程序多次，有了循环语句，才使得一些不很长的程序却可能在高速计算机上运行很长时间。本章介绍的一种导出数据类型——数组乃是一种最常用的数据组织形式，数组的处理往往采用循环语句。

4.1 控制语句、复合语句和空语句

4.1.1 简单的计算器程序

首先通过一个实例来说明控制语句在程序中的作用，简单的计算器功能是：输入两个数，然后在屏幕上显示某一双目算术运算结果。

```
// program 4_1.cpp
#include<iostream.h>
void  main()
 {
  int  x, y;
  cout<<"first  integer:";
  cin>>x;            // 输入一个整数
  cout<<endl<<"Second  integer:";
  cin>>y;            // 输入一个整数
 cout<<endl<<x<<"+"<<y<<"="<<x+y<<edl;
}
```

执行结果：

first integer：12 %

second integer：7 %

12 +7=19 %

这个“计算器”利用输入语句可以输入不同整数，但只可作加法，功能很差。为了改进它，使用户可以选择不同的算术运算，利用本章将介绍的开关语句（一种控制语句）设计出下面的程序。

```
// program 4_2.cpp
#include<iostream.h>
void  main()
 {
  int  x, y;
  char  op;
  cout<<"first  integer:";
  cin>>x;
  cout<<endl<<"Second  integer:";
  cin>>y;
```

```
    cout<<endl<<"operator (+, -, * , /, %):";
    cin>>op;
    switch(op)                    // 开关语句
    {
      case'+':
        cout<<endl<<x<<"+"<<y<<"="<<x+y;
        break;
      case'-':
        cout<<endl<<x<<"-"<<y<<"="<<x-y;
        break;
      case'*':
        cout<<endl<<x<<"*"<<y<<"="<<x * y;
        break;
      case'/':
        cout<<endl<<x<<"/"<<y<<"="<<x/y;
        break;
      case'%':
        cout<<endl<<x<<"%"<<y<<"="<<x%y;
        break;
      default:
        cout<<endl<<"Wrong!";
    }
}
```

此程序运行中可任意输入两整数及运算符，并显示运行结果：%

```
first  integer:12 %
second  integer:7 %
operator (+, -, *3 , /, %):*3 %
12 * 7=84 %
```

这个计算器可以由用户选择两个整数和算术运算符，确实是一种改进，其中开关语句可以根据用户输入的运算符 op 的值,决定执行开关语句中所列的六种情况之一。不过，这个计算器还有一个明显的缺点，每次只能计算一题，能否进一步改进，使它能够一次运行计算多个不同的算术题，同样可通过控制语

句的使用实现。

```
//program 4_3.cpp
#include<iostream.h>
void  main()
 {
  int  x, y;
  char  op, cont;
 bool  quit = true ;
  while (quit) {
   cout<<"first  integer:";
   cin>>x;
   cout<<endl<<"Second  integer:";
   cin>>y;
   cout<<endl<<"operator (+, -, * , /, %):";
   in>>op;
   switch(op) {              // 开关语句
    case'+':
      cout<<endl<<x<<"+"<<y<<"="<<x+y;
      break;
    case'-':
      cout<<endl<<x<<"-"<<y<<"="<<x-y;
      break;
    case'*':
      cout<<endl<<x<<"*"<<y<<"="<<x*y;
      break;
    case'/':
      cout<<endl<<x<<"/"<<y<<"="<<x/y;
      break;
    case'%':
      cout<<endl<<x<<"%"<<y<<"="<<x%y;
      break;
    default:
      cout<<endl<<"Wrong!";
```

```
    }
  cout << endl <<"Do you want to continue ? (y or n) ";
  cin >> cont;
  if (cont == 'n') quit = false;
   }
 }
```

此程序运行中可作任意多次算术运算，可能有如下的显示运行结果：%

```
first  integer：12 %
second  integer：7 %
operator（+，-，*3 ，/，%）：*3 %
12 * 7=84 %
first  integer：24 %
second  integer：13 %
operator（+，-，*3 ，/，%）：&3 %
Wrong!
first  integer：47 %
second  integer：15 %
operator（+，-，*3 ，/，%）：+3 %
47 + 15=62
```

程序中的 while 语句是一种循环语句，当变量 quit 的值为 true 时，程序重复执行，直到 quit 的值变为 false。if 语句是一种分支语句，当 char 类型变量 cont 的值为 ' n ' 时，执行语句 quit＝false ;，否则跳过它，执行下面的语句。它们的用法在下节介绍。%

在学习了控制语句的语法规则后，读者还可以进一步改进上面的计算器程序，使它具有更多更强的功能。

4.1.2 控制语句

控制语句是用来控制程序中各语句执行的次序，更确切的说，控制语句是用来改变或打破程序中按语句的先后次序顺序执行规律的。C++语言中的控制语句分条件控制语句和无条件控制语句，包括如图 4.1 所示的九种不同功能控制语句。

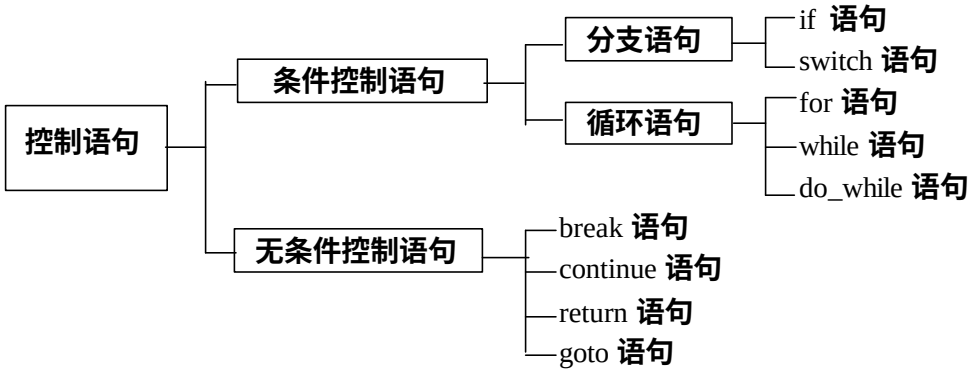


图 4.1 控制语句

程序中控制语句的使用具有关键性的作用。可以说正确灵活的使用控制语句，才能设计出多种多样的软件系统。包含控制语句的计算机程序的执行过程一般称为程序的动态结构，而程序的书写顺序称为静态结构。一般程序的动态结构与静态结构有较大的差别。一方面，如上所说，实际的程序不得不如此；另一方面，程序的动态结构与静态结构的差别，也给程序设计、调试和维护造成了困难，特别是当程序的规模较大时，软件的开发与维护是一项非常复杂且很难排除错误的工作，例如目前流行的 Windows 系统，虽然已为成千上万个用户采用，仍不能保证其中没有任何错误。

4.1.3 复合语句和空语句

C++程序中还有另外一种语句，称为复合语句，与说明语句、表达式语句、控制语句不同，它不是另外一种不同类型的语句，它是若干语句的组合。

复合语句又称为块语句（block）或程序块。它不是一类独立的语句，其格式为：

```
{<语句 S1>
 <语句 S2>
 …
 <语句 Sn>
}
```

语句 S_i：可能是说明语句、表达式语句、控制语句、也可能是一个复合语句。

分割符“{”和“}”的作用，是把 S_1，S_2，…，S_n 这个语句序列组合成一个语句，虽然包含多个语句，但在逻辑或概念上，它作为一个语句出现。一个语句由若干语句组成，称为语句的嵌套，这种嵌套的概念，对于理解 C++语言的语法规则是非常重要的。

块语句可以出现在程序中的任何地方，但块语句最主要的出现位置在下面情况：

（1）作为函数体；

（2）作为循环体；

（3）作为 if 语句的一个分支。

另一种特殊的语句是空语句。

空语句，就是什么都不做。其格式为：

;

也可以把空语句列入表达式语句之中，但把它与复合语句列到一类，更能体现出其概念上的含义。

有了空语句的概念，程序往往变得灵活。例如，读者在下文将看到：

（1）for 语句中，控制部分的三个表达式语句 E_1，E_2，E_3 和循环体 S 都可以是空语句；

（2）goto 语句使用中的标号语句可以是空语句：

L1： ;

这样的标号语句的使用常常是方便的。

4.2 分支语句

4.2.1 分支语句

分支语句亦称选择语句，属于有条件控制语句，它是计算机具有按条件判断、“思维”的基础。

C++提供两类分支语句，if 语句和 switch 语句。

1．if 语句

if 语句亦称条件语句，如果语句，它有两种形式：

if（<表达式 E>）<语句 S>

if（<表达式 E>）<语句 S_1> else <语句 S_2>

例如：

if（a<= b）{c=a;a=b;b=c;}

if（a<c） cout<<a；else cout<<c；

关键字 if：指明该语句为条件语句。

表达式 E：在 if 后面，用（,）括起来，该表达式应为一 int 型或 char 型的表达式，因此，int 型，long 型，char 型，以及 enum 型表达式都是合法的，其值等于 0 为假，非 0 为真。

语句 S：可以是任何类型的语句，也可以是块语句。语句 S_1，语句 S_2 同 S。

关键字 else：出现在第二类 if 语句中，它表示当表达式 E 取假值（0）时跳过语句 S_1，执行 S_2。

图 4.2 是两种 if 语句的流程图。

两点说明：

（1） 严格的说，表达式 E 应为一布尔表达式，其值应为 false 和 true。C++ 语言对 E 的要求较松，比较方便灵活。

（2） if 语句中的语句 S，S_1，S_2 也可以是 if 语句，即 if 语句的嵌套。

在 if 语句嵌套时，C++语言规定，为了避免二义性，每个 else 只与它前面的最近的那个未配对的 if 配对。

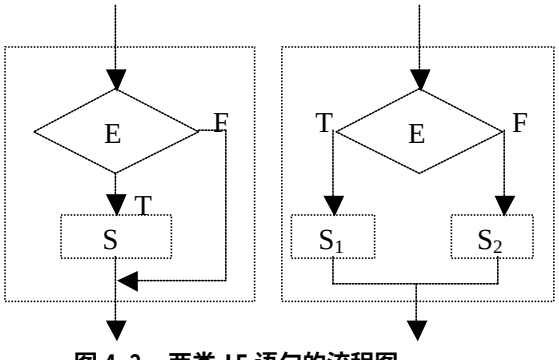


图 4.2 两类 IF 语句的流程图

例如，求三个整数中的最大元的程序：

```
// program 4_4.cpp
#include<iostream.h>
void  main ()
 {
  int a, b, c, max;
  cout << "first  integer:";
```

```
    cin >> a;
    cout << endl << "second  integer:";
    cin >> b;
    cout << endl << "third  integer:";
    cin >> c;
    if  (a<b)                          //  以下 6 行程序用来求最大元
      if  (c<b)  max = b;
      else  max=c;
     else
      if  (c<a)  max = a;
      else  max = c;
    cout << endl <<"maximum element: " << max << endl;
}
```

运行结果：

first integer: 35
second integer: 47
third integer: 12
maximum element: 47

这里第 13 行的 else 必须与第 12 行的 if 配对，反之，如把第 12 行视为一个完整的 if 语句，把第 13 行的 else 和第 11 行的 if 配对，显然程序就会出现错误。所以，在使用分支语句时，必须记住“每个 else 只与它前面的最近的那个未配对的 if 配对”这一规则。。

2．switch 语句

switch 语句又称开关语句，或分情况语句，它是一种多分支语句。其基本格式为：

switch（<表达式 E>）{
case <情况常量 C_1>：<语句序列 S_1>
case <情况常量 C_2>：<语句序列 S_2>
…
case <情况常量 C_n>：<语句序列 S_n>
default：< 语句序列 S_{n+1}>

关键字 switch：指明该语句为 switch 语句。

表达式 E：int **型，**char **型及其派生类型和枚举类型的表达式。**

关键字 case：用来引出一个分支。

情况常量 C_i：为表达式 E **的一个可能值。当计算表达式** E **的值为** C_i **时，控制跳过前面的** S_1，…S_{i-1} **语句序列，直接执行语句序列** S_i。

default：关键字 default **表示当表达式** E **的值不等于已列出的情况常量** C_1，C_2，…，C_n **时，控制直接转向语句序列** S_{i+1} **（跳过** S_1，S_2，…，S_n）。default **分支可以缺省，它相当于** S_{n+1} **为一空语句。**

语句序列 S_1,S_2，…，S_{n+1} **是** switch **语句的各个分支，** S_i **可以是任意多个语句的序列。**

三点说明：

（1）表达式 E **和情况常量** C_1,C_2，…,C_n **应为同一类型，且** C_1，C_2，…，C_n **互不相同，当需要对于表达式** E **的若干不同值对应于同一语句序列时，可把它们列在一起：**

```
case  C1  ：
case  C2  ：
case  C3  ：S3
```

这样就可实现当表达式取 C_1，C_2 **或** C_3 **时都转向语句序列** S_3。

（2）注意图 4.3 **给出的流程图，当表达式** E **取值** C_i **时，控制转到语句序列** S_i（**跳过** S_1，S_2，…，S_{i-1}），**在执行完** S_i **之后，不是跳出整个开关语句，而是继续执行语句序列** S_{i+1}，…，S_n，S_{n+1}，**然后才转出开关语句继续执行后面的程序；**

(3)如果实际的问题要求程序在执行各情况分支之后，直接跳出开关语句时（见图 4.4)，开关语句的使用须略作改动，即在每个情况分支的语句序列之后增加一 break **语句。其形式为：**

```
switch（E）{
    case C1：S1
        break;
    case C2：S2
        break;
    case Cn：Sn
        break;
    default：Sn+1
}
```

其中的 break **语句在后面介绍。**

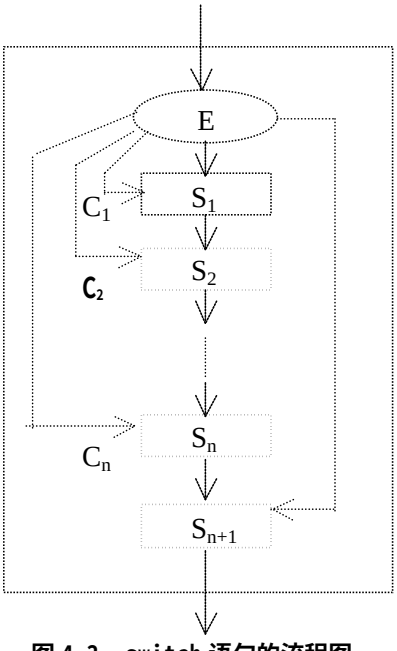


图 4.3 switch 语句的流程图

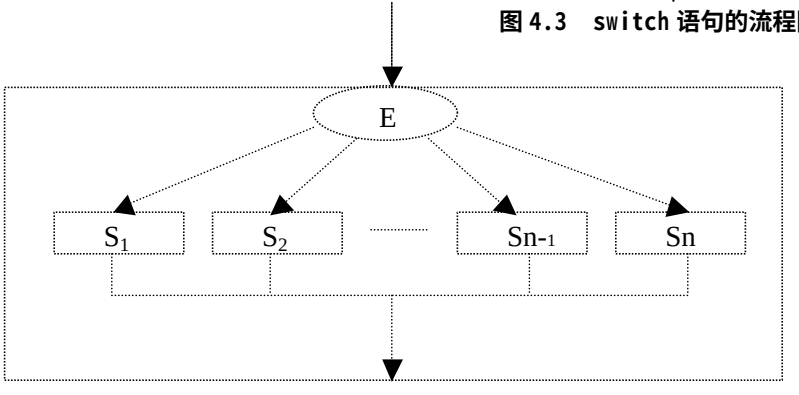


图 4.4 switch 语句的第二种流程图

4.2.2 温度值变换程序

```
program4_5
  //program4_5.cpp
  #include<iostream.h>
  void  main ()
```

```
{
   float  t，tc，tf；
   char  corf；
   const  float  fac=1.8，inc=32.0；
   cout＜＜"Enter  temperature："；
   cin＞＞t；                                    〃输入温度值
   cin＞＞corf；                                 〃指出是摄氏(c)或华氏(f)
   if（corf==’c’‖corf==’C’）
   {
        tc=t；
        tf=t＊fac+inc；
   }
   else
        if（corf==’F’‖corf==’f’）
        {
            tf=t；
            tc=（t＊inc）/fac；
        }
        else  tc=tf=0.0；
   cout＜＜endl＜＜"The  temperature  is："；
   cout＜＜tc＜＜"C="＜＜tf＜＜"F＼n"；
}
```

此程序用于温度值(摄氏温度与华氏温度)的换算，输入温度值并指出该值是摄氏（C）还是华氏(F)温度，然后程序将根据不同的输入(摄氏或华氏)进行不同的换算。例如输入 40.2C，计算结果将在屏幕上显示：%

```
Enter  temperature：40.2C %
The  temperature  is：40.2C=104.36F %
```

说明：%

(1) 第 11～25 行是条件语句，其中第 11 行是条件表达式，12～15 行为语句 S1，它是一个由两个语句组成的块语句；16～22 为语句 S2，它又是一个条件语句，它的条件表达式在 16 行，17～21 是它的语句 S1，而它的 S2 在 22 行。11～25 行程序的执行过程是：%

如果键盘输入的字符(存在 corf 中)为 C 或 c，即 corf==’C’或 corf==’c’，则

程序继续执行 12～15 行，执行完，转到 23 行输出。%

如果键盘输入的字符为 F 或 f，则 11 行的条件表达式计算的结果为 0(表示为假),程序跳过 12～15 行,转到 16 行执行第二个 if(条件)语句。这时 corf 为'F'或'f'，故该条件表达式为真，即结果值为 1，则执行 18～21 的程序，然后跳到 23 行输出。%

如果键盘输入的字符非 C，c，F，f，则认为出错，转而执行 22 行，输出为 0.0C=0.0F。

图 4.5 是上面程序的运行流程图。

（2）程序中 12～15，18～21 就是这样的块语句。读者可以发现，在块语句后，else 前省略了语句后面所要求的“ ;”。

（3）条件语句是一种分支语句，它可以根据条件，在两种不同的处理中选择其一。在本节的例中，实际上是有三种不同的处理方式，%

它是采用两个条件语句来实现区分控制的。在实际问题中，常有按多种不同条件分别进行处理的问题，例如，某数学竞赛，按成绩划分获奖等级，95 分以上为一等奖，90 分以上为二等奖，80 分以上为三等奖，其余不得奖。可以通过三重条件语句区分出四种情形分别处理。其形式为：

```
if（score＜95）%
    if（score＜90）%
       if（score＜80）    s4；%
       else   s3；%
    else   s2；%
else   s1；
```

C++语句为这类多选择分支的情形提供一种更方便的分支控制语句 :即开关（switch）语句。

4.3 循环语句

4.3.1 循环语句

在实际应用中，人们常需要计算机重复多次完成相同或相似的动作，这时，循环语句就非常有效了。正是由于循环语句的使用，使得人们可以写出很短的程序令计算机作大量的工作。

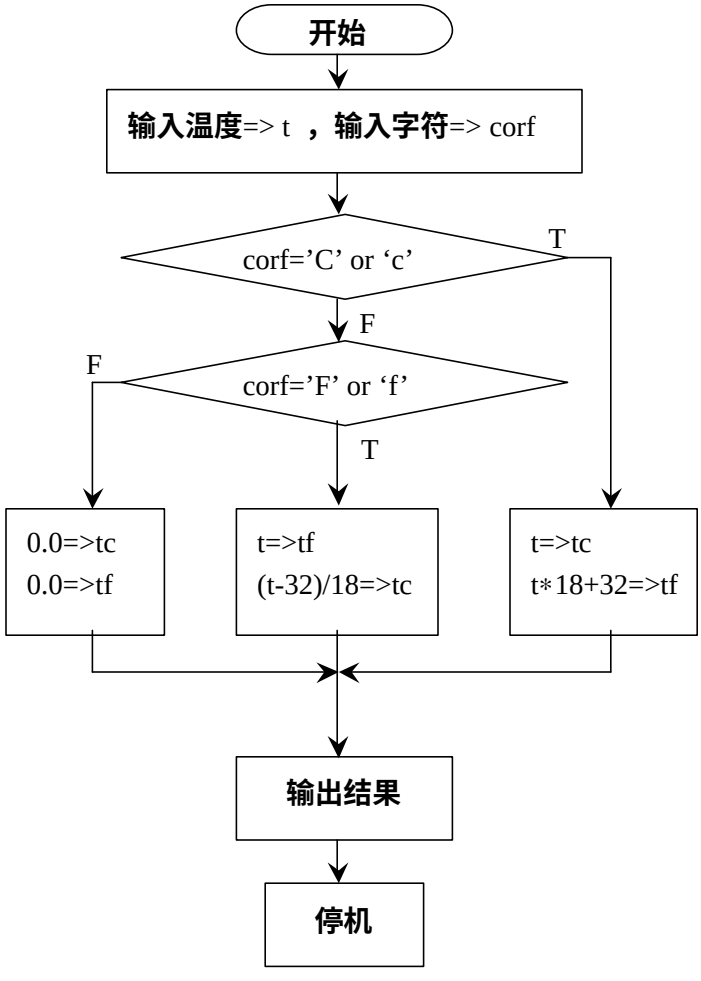


图 4.5 温度程序的运行流程图

%

循环语句亦称重复语句，它可以按一定规则控制一段程序（循环体）重复执行若干次。

C++语言提供三种循环语句。

1．for 语句

for 语句是 C++程序中最常用的，功能最强的循环语句，for 语句的流程图见图 5.6，其格式为：

for（<表达式 Ei>；<表达式 E2>；<表达式 E3>）<语句 S>

for：关键字 for 指明该语句为 for 语句。

表达式 E1：一般用于为循环控制变量赋初值，故又称初值表达式，但从语法规则看它并不一定是一赋值表达式。

表达式 E2：用于判定循环是否继续，故称为条件表达式，E2 必须是具有整型值的 int 型，char 型及其派生类型和枚举类型的表达式。

表达式 E3：一般用于循环变量的增（减）值，故称为增量表达式，但从语法规则看它可以是任何表达式语句。

语句 S：它可以是单个语句，也可以是（在多数情形下）块语句，它是要被循环重复执行的程序段，故又称为循环体。

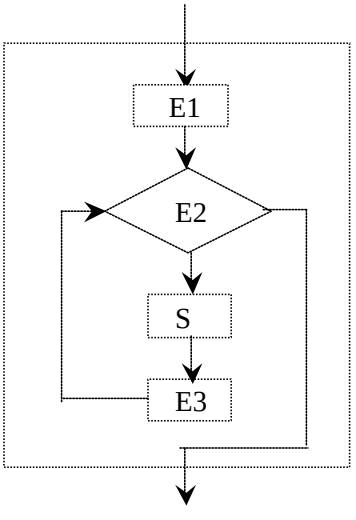


图 4.6 for 语句的流程图

例如：

```
for（i= 0；i＜n；i++）
    cout＜＜i+1；
```

几点说明：

（1）表达式 E1，E2，E3 都可以为空。其中 E2 为空语句，意味 E2 永远取值为真，循环体将可能无限循环下去，除非 S 中含有在一定条件下执行跳转的语句。

（2）表达式 E1 只执行一次，不仅可初始化循环变量，也可使用逗号表达式，初始化其它成分，例如：

```
for（i=1，j=10；i＜n；i++，j--)
    cout＜＜i+j；
```

（3）表达式 E2 和 E3 在每次执行 S 的前后要分别执行一次，如果采用逗号表达式的形式，则 E2，E3 除了完成判定循环条件和循环变量增量的功能之外，还可进行其它数据处理。例如：

```
for（i=1，j=1；x=i+j,i＜n；y=x+1，i++）；
```

由此看来，for 语句中被重复执行的程序不一定只包含在循环体 S 中，也可能包含在 E2 和 E3 之中。

（4）S 可以是空语句，例如：

```
for（t=0；t＜some-value；t++）；
```

可用于某种延时，

```
for（i=0；j＜n；s+=i,i++）；
```

它等价于

```
for（i=0；i<n；i++）s+=i;
```

2．while 语句

while 语句是一种形式简单的循环语句，它实际上是 for 语句在表达式 El 和 E3 为空的特殊情形。while 流程图见图 4.7，其格式为：

while（〈表达式〉）语句 S

例如：

```
while（inp！=’Q’）{
    cin>>inp；
    cout<<inp；
}
```

while：关键字 while 指明为 while 语句，注意 whi1e 也出现在 do-while 语句中。这一点它与 for，switch，if,else，do 等关键字不同。

表达式 E：用于判定循环是否继续，首先计算 E 的值，为非零整数即为真，执行循环体，循环体执行后再计算 E,直到 E 取 0 值（假）时跳出循环。

语句 S：为循环体。

由图 5.7 不难看出：

while（E） S

相当于

for（;E; ） S

3．do-while 语句

do-while 语句类似于 while 语句，只是把循环的判定移至循环体之后（图 4.8），其格式为：

do <语句 S> while（<表达式 E>）；

例如：

```
int  s=0,i=1;
do
    s+=i++；
while（i<=100）;
```

do：关键字 do 指明该语句为 do-while 语句。

语句 S：为循环体。

while：关键字 while 引出循环条件表达式。

表达式 E：判定循环是否继续，其值为真（非 0 整数）时返回执行循环体 S，否则跳出。

do-while 语句与 while 语句的主要区别是，它的循环体至少被执行一次；而 while 语句是首先判断条件，可能一次也不执行就跳出。

do-while 语句的条件表达式 E 的作用与 Pascal 语言中的 Repeat-Until 语句不同，后者是条件为真时跳出。

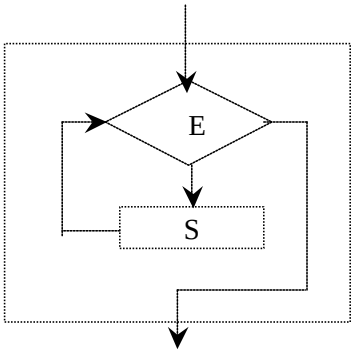


图 4.7 while 语句流程图

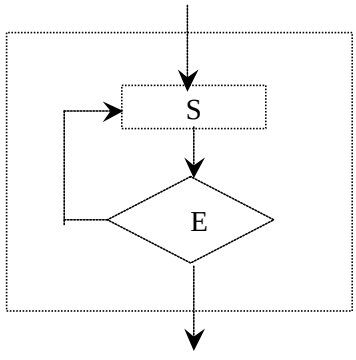


图 4.8 do-while 语句流程图

4.3.2 求素数

下面的程序用来求出 1000 以内的全部素数。

```
// program 4_6.cpp
#include <iostream.h>
void main()
{
   const int m=1000;
   int i, j, isprime;
   for(i=2;i<=m;i++)
   {
      isprime=1;
      for(j=i-1;j>1;j--)
         if(i %j==0) isprime=0;
      if(isprime) cout<<i<<', ';
      if(i %30==0) cout<<endl;
```

```
    }
  }      %
```

运行结果为：

```
2，  3，5，7，11，13，17，19，23，29，
31，37，41，43，47，53，59，
°°°°°°
```

说明：%

（1）当 m 值比较大时，这个程序计算量是比较大的，事实上这是用来求出小于 m 的所有素数的最简单，也是最“笨”的算法。%

一种改进是把第 10 行修改为：%

```
  int   i1=int（sqrt（i））；%
  for（j=i1；j＞1；j--）%
```

为了确定整数 i 是不是素数，不需要用 2，3，…，i-1 来试除它，只需用 2，3，…，sqrt（i）试除就可以了。%

这里 sqrt（i）是标准函数，功能是计算平方根，而 int（）则是把浮点值转化为整型值。%

另一种算法是埃拉脱散（Eratosthenes）筛法，将在下章介绍。%

（2）for 语句是一种最常用的循环语句，其作用是令循环变量 i 分别取值 2，3，…，m，对每个 i，测试其是否为素数，如果是素数则把它输出。%

（3）for 语句循环体内还有一个 for 语句，它的作用是检查变量 i 的当前值)是否为素数，具体方法是令循环变量 j 分别取 i-1，i-2，…，3，2，让 j 分别去除 i，如果有这样的 j 能整除 i，即 i%j==0 为真，则令 isprime 为 0，当这个 for 语句执行完转到 12 行时，测试变量 isprime 为 0 意味着当前的 i 有因子（2～i-1），反之如 isprime 为 1 意味着 2～i-1 都不能整除 i，这个 i 为素数。% 由此可见循环语句可以嵌套，处于内层循环语句的循环体中的语句，重复执行次数最多。

（4）最后是素数输出。一般数据输出，无论是显示或打印，设计者应适当考虑输出的格式，本程序对格式的处理是：%

每输出一个素数之后，跟随一个逗号，以便区分相邻的两个素数。%

每当 i 值取 30，60，90，…值时令输出头回车换行。于是输出的形式将为：

2，3，5，7，11，13，17，19，23，29，%

31，37，41，43，47，53，59，%

…%

4.3.3 计算常数 e 的值

从本节提供的程序中，读者可以开始体会到计算机的力量了。%

$$e=1+\frac{1}{1!}+\frac{1}{2!}+\frac{1}{3!}+\cdots+\frac{1}{(n-1)!}+r_n$$

e 是自然对数的底，它和π一样是数学中最常用的无理数常量。其近似值的计算公式为：

当 n 充分大时，这个公式可以计算任意精度 e 的近似值。为了保证误差 $r_n < e$，只需 $\frac{1}{(n-1)!}(>r_n)<e$。故有下面的程序。

```
//program 4_7.cpp
#include<iostream.h>
void  main( )
 {
   const  double  eps=0.1e-10;
   int  n=1;
   float  e=1.0, r=1.0;
   do  {
         e+=r;
         n++;                                  //相当于 n=n+1
         r/=n;                                 //等价于 r=r/n
       }
   while (r>eps) ;
   cout<<"The approximate Value of natural logarithm base is  ";
   cout<<e<<endl;
 }      %
```

此程序计算出误差小于 eps 的 e 值。%

说明：%

（1）eps 作为双精度浮点型（double）常量在程序开头说明，在 C++程序中，常量就是赋了初值后不可改变的变量。

（2）由 do-while 循环读者可以看出，用户提供了常数 eps(它是所需要的精

度)，并不知道为了达到这个精度，应该计算级数的多少项，这一点是程序通过它自己的判断完成的。%

(3) 13 行中表达式 r＞eps 是一个关系表达式，类似的关系表达式还有：%

a==b， a-b! =0， i＜=n-1

等等。关系表达式的值可能为 0(假)或 1(真)，在 C++程序中，关系表达式的值按整型处理。%

(4) 这个程序虽然很短，但已开始显出计算机的威力：%

① 它可以自己判断累加多少项时达到精度要求，一般程序员不经实际计算很难事先知道。%

② 它可以控制程序重复很多遍，而且每次重复时，参数可能在变化。有的程序需要在高速计算机上运行几分钟，甚至几小时，这意味着数以亿计的基本操作，但程序不过几十行，几百行，原因就在于循环语句的使用。%

4.4 转向语句

C++语言提供了四种无条件转向语句。

1.break 语句

break 语句又称跳出语句。它用于循环语句和 switch 语句，其功能是把程序的运行流程跳转到它所在的循环语句或 switch 语句的出口。其格式为；

```
break；
```

仅由一个关键字 break 组成。

break 语句使用方便，含义也很明确，不像 goto 语句那样容易产生混乱。

2．continue 语句

continue 语句又称继续语句，是仅用于循环语句中的一种控制语句。其功能是把程序运行流程跳转到该循环体（不是该循环语句）的出口点，结束循环体的一次重复，其格式为：

```
continue；
```

continue 语句和 break 语句类似，虽然都是无条件跳转语句，但其跳转的范围明确，不易产生问题，是推荐使用的无条件转移控制语句。

3．return 语句

return 语句又称返回语句。只用于函数定义，其功能为：

(1) 把程序运行的流程跳转到该函数的调用点，或函数调用的出口点。

（2）计算返回表达式 E，并把其值作为该函数的返回值。其格式为：

return；或

return〈表达式 E〉；

return：关键字 return 仅用于返回语句。

表达式 E：当函数的返回类型为 void 型时，返回语句应取第一种方式，否则，取第二种方式。表达式 E 的类型应与函数的（返回）类型相一致。对于非 void 类型的函数来说，其函数体中的 return 语句是必不可少的。

4．goto 语句

goto 语句又称转向语句，其功能是令程序跳转到程序指定的某标号语句处。由于标号语句的设定较为灵活，因此，goto 语句的跳转控制比上两三种跳转语句有较大的随意性。其格式为：

goto〈标号〉;

goto：关键字 goto 指明该语句为 goto 语句，其后必须有一标号。

标号：标号是一个标识符，其定义出现在标号语句：

〈标号〉:〈语句〉

例如：

```
……
L1：〈语句 S1〉
……
goto  L2；
……
L2：〈语句 S2〉
……
goto  L1；
……
```

语句 goto L2；把控制流程跳转到 L2：〈语句 S2〉；而语句 goto L1，又跳转到 L1：〈语句 S1〉。标号语句可以出现在转向它的 goto 语句之后，也可以出现在其之前。goto 语句和 break，continue，return 语句都是无条件转移语句，无须任何判断，程序执行到此时，必须跳转到一个确定的位置。不过，goto 语句与前三者不同的是：goto 语句转向的目标位置由程序员任意确定，而 break，continue 和 return 语句的转向目标位置是唯一地由其本身的位置所决定。然而，goto 语句的转向目标——标号语句的位置虽然可由程序员确定，但不是完全任意的，它应满足下面的限制：

（1）一个函数定义（函数体）内的 goto 语句不可转向到函数之外。换句话说，函数的出口点只能是 return 语句或函数体结束。

（2）一个块语句（包括函数体和循环体）外的 goto 语句不可转向到该程序块之内，因为这往往会产生计算机难于处理的局面。

即使有这样的限制，goto 语句仍然是十分自由的，goto 语句的使用容易造成：

①使程序的静态结构与程序的动态结构差别增大，使程序段之间形成“交叉”的关系，不利于程序的维护和调试。

②一个好的程序，它的各个程序段最好是单入口单出口的，没有死循环和死区（不可到达的程序段）。但 goto 语句的使用容易破坏这种状态。

③历史上关于 goto 语句的讨论是程序设计和软件开发史上的重大事件之一，最近一次的讨论，是由“goto 语句是有害的”这样一篇著名论文引起，最终以“限制使用 goto 语句”作为结论，从而对软件开发的发展产生了重要影响。

因此，本书向读者建议，不用或少用 goto 语句。

4.5 数据导出类型,数组（Array）

在计算机所处理的数据中，最常见的，也是最需要由计算机高速处理的数据是成批出现的同一类型的数据，C++语言中把这类数据称为数组。例如：%

· 监测系统在一段时间内获得的检测、采样值，可能是 int 型或 float 型数据的数组；

· 一个管理系统的数据库，可能是一系列结构类型(记录)数据的序列；

·一幅电视图像,可能是其每个像素的颜色及灰度值(整型数)的二维阵列。%

数组不是一种独立的数据类型，在本书中我们称之为导出类型。任何一种类型的一批数据，都可以组成数组，例如 int 型，float 型，char 型数据可以组成数组；这些基本类型的派生类型(如 long 型，double 型，unsigned char 型)的数据，枚举、结构、联合类型的数据，以至由用户或系统定义的类(类型)的对象，都可以组成数组。%

循环语句的使用常常是与数组相联系。

4.5.1 导出类型的概念

C++语言的重要特征之一是数据类型极为丰富，在第四章已介绍了由系统定义的基本数据类型，在后面的章节中，我们还将介绍由用户定义的数据类型。C++语言还提供另一类数据类型即导出数据类型,其特点是这种类型的定义是在

其它已定义类型的基础上定义的，而且其运算也是确定的。在 C++语言中，主要有三种，即：

还有“第四种”导出数据类型，结构（及联合）类型，由若干不同类型数据（或对象）组合而成，将在下节介绍。不过，C++语言允许结构（及联合）包含函数成员，这种包含函数成员的结构（及联合）已不是导出数据类型，将在第七章介绍。

数组是程序设计中最常见的数据表示形式，严格他说，数组并不是一种新的数据类型，而是某一特定类型（可以是基本类型及其派生类型，也可以是用户定义的类型）的若干变量（或对象）成组说明的一种形式，因此，我们把数组及后文介绍的指针和引用类型称为导出类型。例如：

```
int  a，b，c，d；
int  array [4]；
```

这两个说明语句都说明了四个 int 型变量。

第一个说明语句为四个整型变量分别起了名字，分配了内存，它们是互相独立的，在使用中分别用它们的名字存取。

第二个说明语句虽然也说明了四个整型变量，并为其分配了内存，不过这四个变量有一个共同的名字 array，它们之间则由“下标”来区分。或者说，它们放在一起名为 array，而每一个则分别取名为 array[0]，array[1]，array[2]，array[3]。如果有 100 个整型变量，要求分别赋给 1～100 这 100 个整数，有了数组的形式，这件事就很容易实现：

```
int  list[100] ；
for（int  i=0；i＜100；i++)
     list[i]=i+1；
```

试想，倘若分别说明 100 个独立的整型变量，再分别赋值，那就很麻烦：

（1）要为它们起 100 个名字，使得说明语句很长；

（2）再用赋初值或赋值语句的方式，把 100 个值赋给它们，这样致使程序冗长而不清晰。

4.5.2 一维数组

由 n 个同一类型数据组成的一维序列，构成一维数组，一维数组的下标为 0，…，n-1。

1．一维数组的说明

任何一种已知类型数据都可说明为数组，数组说明的格式为：

<类型名><数组名>[<元素数>]＝{<初值表>} ；

类型名：可以是基本类型名，基本类型的派生类型名，类名，枚举类型名，结构，联合类型名（也可以是枚举、结构联合的类型说明）。

关于指针数组将在下节说明。还有数组类型的数组就是多维数组了，也将在下面介绍。

数组名：标识符，数组名有两个附加作用：

（1）是表示数组元素的下标变量。如 array[2]表示数组的第三个元素。

（2）它还是一个指向数组的首元指针。

元素数：一个正整数，指出数组的元素个数，或数组的大小（size）。元素数要用方括号“ [] ”括起来。方括号 [] 不可缺省，元素数则有时可缺省，这时必须赋初值，系统按所赋的初值个数确定数组的大小。

初值表：可缺省。是由花括号 {,} 括起来且用逗号‘ , ’分开的初始化常量值。例如：

```
int  list [100], A[10][10] , B[]={4,3,2};
char  ch[26];
complex  com[4]={ (3.2, 4.7), (0.0, 0.0) };
```

其中，数组 list[100]未赋初值。

数组 A[10][10]为一个二维数组，未赋初值。

数组 B[]已赋初值，元素数为 3。

数组 com[4]是用户定义的 complex 类型的数组，数组有 4 个元素，其中前两个元素已赋初值，初值分别为（3.2，4.7），(0.0，0.0)。

2．数组的操作——下标变量

C++语言未提供对于数组整体进行操作的运算符和运算。对于数组的操作是通过对于其元素，即下标变量进行的。下标变量的格式为：

<数组名>[<下标>]

下标：整数，或整型表达式，其取值范围为 0～n-1,n=数组元素总数，例如：

```
int  A[4]；
```

共有四个下标变量：A[0]，A[1]，A[2]，A[3]。

对于下标变量可进行下列操作。

（1）赋值。可以三种方式进行：

①初始化：

```
int  A [4] ={1，2，3} ；
```

相当于一次为 A[0]，A[1]，A[2] 赋值为 1，2，3。

②赋值语句：

```
A[3]= 4* A[1] ；
```

相当于把 4*2=8 赋值 A[3]。

③输入语句：

```
cin＞＞A[0] ；
```

或

```
for（int i=0；i＜4；i++）  cin＞＞A[i]；
```

后者可通过键盘操作，为数组的四个元素依次赋值。

（2）一般运算。

下标变量可与同一类型的一般变量一样参加它所允许的运算。如：

```
A[0]+= A[2]++；
cout＜＜A[0]＜＜″ ″＜＜A[1]*A[2]；
```

等等。

下标也可以是一个表达式，如：

```
int  A[4]＝{1，2，3，4}；
A[3]＝A[A[2]-A[1]]*4；
```

下标表达式应注意其值应保持在 0～n-1 范围之内。

4.5.3 多维数组

若干同一类型的数据 m 列 n 行的矩阵，则可称为二维数组，二维数组亦可视为一维数组的数组。

```
int  A[m][n]；
```

说明了一个二维数组,它有 m×n 个元素,它也可以视为由 n 个一维数组 int A[m] 组成的（一维）数组。

类似地，还可以说明三维，四维数组，二维以上的数组统称为多维数组。

1．说明和初始化

多维数组的说明（以二维为例）：

<类型名><数组名>[<行数>][<列数>]

类型名：同前节说明。

数组名：同前节说明。

行数：正整数。

列数：正整数。

例如：

```
char  ch[2][3];
```

该二维数组共有 2 行 3 列，元素个数为 2*3= 6。

在二维数组中，为数组赋初值的方式有：

```
int a[2][3]={{1,2,3},{4,5,6}};
```

亦可写为：

```
int a[2][3]={1, 2, 3, 4, 5, 6};
```

二者效果是一样的。系统将按逐行的次序顺序为各元素赋值。其次序为：

```
a[0][0],a[0][1],a[0][2],
a[1][0],a[1][1],a[1][2]
```

对于三维以上的高维数组，其赋值方法是一样的。

2．二维数组的操作

对于二维数组的操作，同样是通过对其元素即下标变量的操作来进行的。操作中应注意：

（1）注意下标表达式的取值范围：

```
char  A[m][n];
```

则下标变量 A[i][j]中 i 的值应在 0～m-1 之间，j 的值应在 0～n-1 之间。

（2）当只有一个下标时：

```
A[i] (0 <= i <= n-1)
```

表示的是一个一维数组，其元素个数为 n，这些元素可用下标变量：

```
A[i][0],A[i][1],…,A[i][n-1]
```

来表示。

（3）更高维的数组，其定义和操作类似。

4.5.4 数组与字符串

从表面上看，一个字符串就是一个字符数组，但在 C++语言中，二者并不完全相同，读者在涉及到字符串处理时，必须注意。

字符串是一个以串尾符'＼0'结尾的字符类型数组。从下面的例子可看出其区别：

```
char  string1[7]="China";
char  string2[ ] = "China";
char  string3[7]= { 'c', 'h', 'i', 'n', 'a' };
char  string4[7]= { 'c', 'h', 'i' , 'n', 'a', '\0'};
```

这说明了四个字符型数组，它们是互不相同的，分述如下。

其中，“China”是一个字符串常量，其字符串的长为 5，由 5 个字符组成，但存放这一串常量却用 6 个字节，最后一个字节放串尾符‘\0’，因此：

字符数组 string1 的长度为 7，赋初值以后其前六个元素已被赋值，分别为'c', 'h', 'i', 'n', 'a'和'\0' 。

字符数组 string2 的长度为 6，赋初值后各元素为'C', 'h', 'i', 'n', 'a'和'\0'。

字符数组 string3 的长度为 7，赋初值后前 5 个元素的值为'C','h','i','n'和'a'。

字符数组 string4 的长度为 7，赋初值后应与 string1 相同。

字符串的操作与指针有关，这部分内容将在下一章介绍。

4.6 结构（struct）类型

实际应用中往往需要把若干种不同类型的数据组合为一个导出类型，这就是结构类型。数组是把若干相同类型的数据放在一起，结构则是把若干不同的数据放在一起。例如，一个公司雇员的数据可能包括：

```
char name[20];
enum {male,female}sex;
float salary;
char phone[11];
```

对公司雇员数据进行处理或检索时，把它们组成结构类型会更方便：

```
struct Employee {
  char name[20];
  enum {male,female}sex;
  float salary;
```

```
  char phone[11];
};
```

C++语言中的结构是作为类似于类的概念处理的，这一点将在第七章说明，在本章结构（以及联合）类型仅作为一种导出数据类型。

4．6．1 结构类型与结构变量说明

结构类型说明的格式为：

struct <类型名> { <成员表>};

类型名：标识符。

成员表：<类型> <成员 1>;<类型> <成员 2>;...<类型> <成员 n>;

结构类型的变量说明的格式为：

[struct] <结构类型名> <变量名表>;

变量说明中的关键字 struct 可以省略（C 语言中不可），变量名表与一般变量说明一样，其赋初始值的方法可从下面的例子中了解：

```
struct Employee {
  char name[20];
  enum {male,female}sex;
  float salary;
  char phone[11];
};                                  // 结构类型 Employee 的定义
struct Employee gy1,gy2;            // 变量 gy1,gy2 的说明
Employee gy3,gy4={"John Smith",male,2107.5,02223501234};
                                    // 变量 gy3,gy4(赋初值)的说明
```

结构类型及其变量的说明也可放在一起：

```
struct Employee {
  char name[20];
  enum {male,female}sex;
  float salary;
  char phone[11];
}gy1,gy2;
```

4.6.2 结构变量的引用和赋值

一个结构变量由若干分量组成，对结构分量的存取由圆点运算符“．”实现，例如，在上例中，雇员 gy1 的姓名可表示为 gy1.name,其电话号码可表示为

gy1.phone。因此，为结构变量赋值，除了在变量说明中赋初值（如为 gy4 赋初值）的方法外，还可以用赋值语句或输入语句为结构分量赋值的方法：

```
gy3.name = "Tom Green";
gy3.sex = male;
cin >> gy3.salary;
cin >> gy3.phone;
```

无论采用哪种方法为结构分量赋值，都必须保证类型一致。

C++语言还允许直接对结构变量赋值，下面的赋值语句也是允许的：

```
gy1 = gy4;
gy3 = gy4;
```

一般，数组变量不能这样直接赋值。

4．6．3 结构数组

结构类型的数据也可以组成数组，称为结构数组。结构数组在许多实际应用问题中的采用非常普遍，例如，通讯录，学生成绩单，商业销售记录，人事档案，资料登记表等等。下面的结构数组是一个公司的雇员档案：

```
struct Employee {
  char name[20];
  enum {male,female}sex;
  float salary;
  char phone[11];
};
Employee efile[100];
```

数组的每一个分量是一个雇员的档案数据。

4.7 C++程序实例

4.7.1 统计学生成绩

已知 n 个学生的注册号和成绩，计算他们的平均成绩，并列出成绩最好的前 t 名学生的注册号和分数。

```
// program4_8.cpp
# include<iostream.h>
```

```
void main()
{
    const int n=150;
    const int t=10;
    int index[n];
    float score[n];                                    //存放注册号
    for(int i=0;i<n;i++)                               //存放成绩
        cin>>index[i]>>score[i];                       //从键盘输入数据
    float sum=0;
    for(i=0;i<n;i++)
        sum+=score[i];                                 //计算分数总和
    cout.precision(2);                                 //设置输出精度
    cout<<endl<<"Average score:"<<sum/n;
    cout.width(28);                                    //设置输出宽度
    cout<<endl<<"register number    score";
    for(i=0;i<t;i++)
                    /* 选取前 t 名分数最高的学生，输出其注册号及成绩3 */
    {   float s=score[i];
        int j1=i;
        for(int j=i+1;j<n;j++)
            if(s<score[j])
            {
                s=score[j];
                j1=j;
            }
        if(j1>i)
        {
            score[j1]=score[i];
            score[i]=s;
            j=index[j1];
            index[j1]=index[i];
            index[i]=j;
        }
        cout.width(4);                                 //输出序号，注册号和分数
```

```
        cout<<endl<<i+1;
        cout.width(11);
        cout<<index[i];
        cout.width(12);
        cout.precision(2);
        cout<<score[i];
    }
}      %
```

运行这个程序，应先按顺序输入学生注册号和成绩，如：%

```
    1  83  2  74  3  65…%
```

这个程序的设计考虑了输出的格式，其运算结果以下面形式显示：%

```
    Average  score:73.41 %
      Register number   score %
    1    142             93.0 %
    2     25             91.5 %
    3     89             91.0 %
    4    112             91.0 %
     …   %
    10    77             65.5 %
```

说明：%

（1）5～6 行说明了两个常量，不把 n 和 t 的值直接用某个常数 150，10 写到语句中，有利于在改变常数时可以简单方便地修改程序。%

（2）数组的变量说明与一般变量说明的区别在于用 [] 指出数组的大小（size)。读者应注意：在数组说明中 [] 内指出的是数组的大小，而在数组变量的使用中，如 score [i]，方括号 [] 中的变量或表达式的值是数组变量的下标。下标表示该下标变量是数组的第几个变量。一个数组的分量——下标变量，在程序中的使用和一个一般变量一样，如程序中的 index [i] 是一个 int 变量，而 score [j] 是一个 float 变量。%

（3）第 9～10 行是一个 for 语句，它循环重复 n 次，每次从键盘分别输入一个学生的注册号和分数到相应的数组。%

在后面的章节还可能碰到从外存储器中的一个文件，或从程序中已经存在的数组中获得数据，在本章，我们暂且只用键盘输入数据这种方式。%

（4）11～13 行完成了 n 个学生总分的计算，结果被累加到浮点变量 sum 中。%

（5）14～15 行完成平均分数的输出，其中 14 行设置了后面要输出的浮点型数据的精度，即在小数点后取 2 位。%

cout.precision（2）；是一个函数调用语句，函数名是 precision，cout 说明此函数是一个标准类的一个标准对象 cout 的函数。现在第一次碰到类和对象，不可能详细讲解，所加的“标准”一词，其含义是指所涉及的流（stream）类和它的对象 cout 都是由系统已经定义好的，其说明可在文件 iostream.h 中找到。%

上面的程序中第 10，14～17，36～42 各行的语句都要求第 2 行的“包含”命令。＃include 命令的作用相当于把已在系统中存在的文本文件 iostream.h 全文插入到包含命令的位置。在这些行的语句中出现的标识符(对象名，函数名等)的说明都可在文件中找到；不然，程序碰到未说明的标识符就无法运行下去了。%

函数 precision 要求一个 int 型的参数，（)中的 2 就是赋给函数的实参。它指出在后面输出的浮点数的精度取到小数点后两位。%

（6）16 行是流类的标准对象 cout 的另外一个函数，它为其后输出的数据设定宽度。

设定宽度为 28，而在 17 行输出的字符串长度为 22，故其左端应填充六个空格。%

（7）18～43 行是一个 for 语句，该语句将重复执行 t 次，每次完成三步工作：%

① 21～26 行从第 i 项到第 n 项中选出最高分，把最高分存到 s 中，其下标存到 j1 中。

② 28～35 行把所选到的最高分 score［j1］与当前的 score［i］互相交换，把 index［j1］与 index［i］互相交换，即把第 i＋1 个最高分移到前第 i＋1 个位置上去。注意，数组的下标是从 0 开始的。%

③ 36～42 行把第 i＋1 个最好成绩按序号，注册号和得分自左至右打印出一行。%

（8）在 18～19 行出现了另一种注释的形式，即由／*和*／括起来的注释，C＋＋允许以下两种形式的注释：%

① 由〃引导的注释，其长度只延续到本行的最右端。%

② 由／*和*／括起来的注释，其长度可以延续任意长度。%

无论多么长的注释，在编译时都当作一个空格处理。%

program4_8 中可以采用结构数组作为学生的成绩表，其效果会更好。

〃program4_8.cpp

```
# include＜iostream.h＞
void  main（）
 {
   const  int  n＝150；
   const  int  t＝10；
   struct  student{
     int  index;
      float  score;
   } scoretab  [n];                                    //  学生成绩表
   float  sum＝0；
   for（int  i＝0；i＜n；i＋＋＝{
       cin＞＞scoretab［i］.index＞＞scoretab［i］.score；
       sum += scoretab [ i ].score;
  }                                                    〃从键盘输入数据并统计总和
   cout.precision（2）；                               〃设置输出精度
   cout＜＜endl＜＜"Average  score："＜＜sum／n；
   cout.width（28）；                                  〃设置输出宽度
   cout＜＜endl＜＜"          register  number     score"；
   for（i＝0；i＜t；i＋＋）
                    ／* 选取前 t 名分数最高的学生，输出其注册号及成绩3 *／
   {  float  s＝scoretab［i］.score；
      int  j1＝i；
      for（int  j＝i＋1；j＜n；j＋＋＝
          if（s＜scoretab［j］. score）
          {
              s＝scoretab［j］. score；
              j1＝j；
          }
      if（j1＞i）
      {
          scoretab［j1］. score＝scoretab［i］. score；
          scoretab［i］. score＝s；
          j＝scoretab［j1］. index；
          scoretab［j1］. index＝scoretab［i］. index；
```

```
            scoretab［i］. index=j；
        }
        cout.width（4）；                                    //输出序号，注册号和分数
        cout<<endl<<i+1；
        cout.width（11）；
        cout<<scoretab［i］. index；
        cout.width（12）；
        cout.precision（2）；
        cout<<scoretab［i］. score；
    }
}       %
```

4.7.2 输出三角函数表

输出从 0 到 90 度之间每隔 15 度的正弦、余弦，正切函数值。

```
//program4_10.cpp
# include<iostream.h>
# include<math.h>
void  main（）
 {
    const  double  pai=3.1416；
    const  int  interval=15；
    cout.width（10）；
    cout<<"Angle  x"；
    cout.width（10）；
    cout<<"sin（x）"；
    cout.width（10）；
    cout<<"cos（x）"；
    cout.width（10）；
    cout<<"tan（x）"；
    float  arc；
    cout.precision（4）；
    for（int  doa=0；doa<=90；doa+=interval）
```

```
    {
        arc＝pai*doa／180；
        cout＜＜endl；
        cout.width（10）；
        cout＜＜doa；
        cout.width（10）；
        cout＜＜sin（arc）；
        cout.width（10）；
        cout＜＜cos（arc）；
        cout.width（10）；
        if（doa==90）
            cout＜＜"－"；
        else
            cout＜＜tan（arc）；
    }
}      %
```

运行这个程序，将会输出一个三角函数表：%

```
Angle x   sin (x)     cos (x)      tan (x)%
0         0.0000      1.0000       0.0000 %
15    "      ┊   "         ┊        "   ┊%
30 %
 ┊   "       ┊             ┊   "        ┊   " %
75 %
90        1.0000      0.0000         －%
```

说明：%

（1）本程序在 25，27，29 行使用系统提供的标准函数(又称库函数)sin ()，cos ()，tan ()，它们都在头文件 math.h 中说明，因此在 3 行包含了此文件。

（2）由于当 doa＝90 时，其正切函数趋于无穷大，故在 29 行对它有特殊处理，否则，如按正弦、余弦一样地直接输出，当 doa 变到 90 时，运行将可能溢出或打印出一个超界的大数。%

4.7.3 画一个四叶玫瑰线图形

我们在屏幕上画一个四叶玫瑰线图形，其函数ρ＝a·sin 2φ φ∈［0°，360°），即当自变量从0°逐步增加到360°时，函数应在平面上画出如图4.9所示的四叶玫瑰线的图形。%

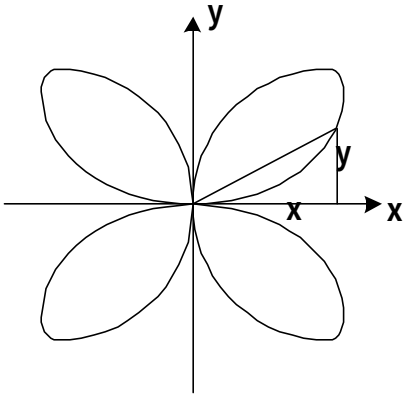


图4.9 四叶玫瑰线图形

把360°分为128份，每个角度对应一个函数值,从而得到平面上一个点，我们用符号‘*’来表示，这样一个“点”的序列就描画出(粗略的)图形。%

一般显示器屏幕可显示25行,每行80个字符，从而有下面的程序：

```
//program4_11.cpp
#include<iostream.h>
#include<math.h>
void main ( )
 {
   const double pai=3.14159，a=16.0；
   const int aspect=2；
   double angle，p；
   int x，y；
   char rose［25］［80］；
   for（x=0；x<80；x++）
        for（y=0；y<25；y++）
             rose［y］［x］=’’；
   for（int i=0；i<128；i++）
   {
      angle=i*pai／64；
      p=a*sin（2*angle）；
      x=int（p*cos（angle））*aspect+40；
      y=int（p*sin（angle））+13；
      rose［y］［x］=’*’；
  }
  for（y=0；y<25；y++）
  {
```

```
        for（x=0；x<80；x++）
        cout<<rose［y］［x］；
        cout<<endl；
    }
}      %
```

说明：%

（1）这是一个输出图形或函数图像的例子，读者在实际应用编程中将经常碰到这类的情况。任何图形，例如，一条直线段、一个圆、椭圆，以至任一函数曲线都是由一个个的点构成的,而在这个例子中四叶玫瑰线就是由若干个‘*’组成的，由于在屏幕上一个字符的高宽比是 2：1，所以在 7 行，定义了常量 aspect=2，在生成曲线时每个点的 x 坐标值乘以 2。%

（2）第 10 行说明了一个二维数组 rose［25］［80］，这是由 25×80=2000 个字符型分量组成的数组。二维以上的数组称为多维数组，它可以视为由 25 个一维数组组成的数组。因此下面的下标变量的含义是清楚的：%

rose［0］［0］，是个 char 型变量，代表二维数组的第一个分量。%

rose［4］［15］，代表数组 rose 的第 4×80+15=335 个分量。rose［24］［79］是其最后一个分量。%

rose［0］是一个一维数组，由二维数组 rose 的前 80 个 char 型分量组成。%

整个二维数组由 25×80 个字符组成，在这里它代表整个屏幕。一个下标变量 rose［i］［j］表示其第 i+1 行的第 j+1 个符号。而一维数组 rose［i］则表示屏幕的第 i+1 行。

（3）11～13 行把整个数组置为空白。%

（4）14～20 行把曲线所经过的位置，置为“*”。%

令角 angle（φ）从 0 变到 2π(一周)，共分 128 份，故每一份为π／64，即每一步的角度值为 i·π／64。%

按公式计算的ρ值，已可确定平面一点，其 x，y 坐标值是个浮点数，在 17～18 行，通过类型变换，化为整数，在相应的点 rose［x］［y］置为“*”。%

类型变换是一种运算，它用来把一种类型的数据，强制变化为另一类型。例如，int（3.47）就是把浮点数 3.47 强制转化为整数 3。float（4）则把整数 4，变为浮点数 4.0，后者在数值上虽然没有变化，但实际在计算机中已发生变化，整数 4 在存储器中由两个字节（byte）表示，而浮点数 4.0 则占用 4 个字节。%

C++语言允许用两种形式表示上述类型转换：int（a）或（int）a 都起到把变量 a（可能是浮点类型)变为 int 型的作用。%

(5) 22～27 行为输出，它产生一个由 128 个“ * ”组成的粗略的四叶玫瑰线。%

(6) 为了使图像更准确，也可以把总的点数 128 增加，角度的变化更精细。%

不过，整个屏幕只有 25×80 个“ 点 ”，已经很难更“ 逼真 ”了。确切地说，我们这个例子主要是为了让读者弄懂曲线如何生成，在实际应用中曲线一般不是由“ 字符 ”来画的，而是由真正的“ 点 ”——像素来画出来的。例如常见的显示器屏幕由 480×640、768×1024、……个“ 点 ”组成，由这样多的点组成的图形和图像自然要逼真得多,对于每一个点,程序可以置为亮或暗,(或高亮)，还可置为不同颜色。为了能对“ 点 ”而不是“ 字符 ”进行操作，要对计算机的“ 工作模式 ”进行变化：%

一般操作模式称为“ 文本模式 ”，我们前面所介绍的所有程序都是在文本模式下工作的。

另一种模式称为“ 图形模式 ”，当把机器的工作状态设置为图形模式时，程序才能按“ 点 ”进行操作。C++语言提供了进行图形处理的一组标准(库)函数，一般在 graphics.h 或 graph.h 中说明。%

然而，本节的例子还是指出了如何用计算机画一个图形或图像。无论是用“ 字符 ”还是用“ 点 ”，计算机画图的基本原理都是一样的。%

4.7.4 Eratosthenes 筛法求素数

本节将用程序求出 1000 以内所有的素数。

求素数最简单的想法就是对 2～1000 的所有数进行检测、用小于该数的数试除，如果都不能除尽，就找到了一个素数。本节的程序也是用来求 1000 以内的所有素数，但其思想简单而巧妙。它是把 2～1000 所有的整数放到一起，首先把 2 保留，把 2 的所有倍数从中去掉，再保留 3，同时删去所有 3 的倍数，下一个数是 5，7，…。好像是一个筛子，把不需要的数逐步筛去，留下的正是所要求的。

```
// Program4_12.cpp
#include<iostream.h>
#include<iomanip.h>
void  main ()
```

```
{
    const  int  n=1000;
    int  i=1, j, count=0;
    int  sieve [n+1];
   for ( j=1;j<n+1;j++)  sieve [j] =j;
    while (i<n+1)
    {
        while (sieve [i] ==1) i++;
        cout<<setw (5) <<sieve [i];
        count++;
        if (count%15==0)   cout<<endl;
        for (j=i;j<n+1;j+=i)
        sieve [j] =1;
    }
}
```

运行结果输出 1000 以内的全部 168 个素数：

```
  2   3   5   7  11  13  17  19  23  29  31  37  41  43  47
 53  59  61  67  71  73  79  83  89  97 101 103 107 109 113
127 131 137 139 149 151 157 163 167 173 179 181 191 193 197
199 211 223 227 229 233 239 241 251 257 263 269 271 277 281
283 293 307 311 313 317 331 337 347 349 353 359 367 373 379
383 389 397 401 409 419 421 431 433 439 443 449 457 461 463
467 479 487 491 499 503 509 521 523 541 547 557 563 569 571
577 587 593 599 601 607 613 617 619 631 641 643 647 653 659
661 673 677 683 691 701 709 719 727 733 739 743 751 757 761
769 773 787 797 809 811 821 823 827 829 839 853 857 859 863
877 881 883 887 907 911 919 929 937 941 947 953 967 971 977
983 991 997
```

此程序以每行 15 个素数从小到大依次在屏幕上显示出给定常数 n 以内的所有素数。%

说明：%

(1) 程序的操作主要是在名为 sieve（筛子）的一维数组上进行的：%

第 8 行把整数 1～n 存入 sieve [n]。%

8～17 行完成全部素数产生和输出任务，其中：%

第 11 行跳过值为 1 的分量，意味着其下一个分量必为一素数。%

第 12 行以宽度为 4 的格式输出一个素数。%

13～14 行保证每输出 15 个素数时回车换行。%

15～16 行把该素数和它的所有倍数的值改置为 1。%

（2）第 12 行采用另一种设置格式的方式，它等价于：

```
cout.width（5）；
cout＜＜sieve［i］； %
```

这一方式更为方便，有关说明可在关于 I/O 流类的说明中找到。%

（3）程序中的数组 sieve[]也可以改为 bool 类型，节省了空间，结果相同，程序显得更为巧妙：

```
//Program4_13.cpp
#include＜iostream.h＞
#include＜iomanip.h＞
void  main（）
 {
    const  int  n＝1000；
    int  i＝1，j，count＝0;
    bool  sieve［n＋1］；
   for（ j＝1；j＜n＋1；j＋＋）  sieve［j］＝1；        // 1 相当于 true
    while（i＜n）                                    // n=1000 不必考虑
    {
        i＋＋；
       if  （sieve[ i ]）{
           cout＜＜setw（5）＜＜i；
           count＋＋；
           if（count%15＝＝0）   cout＜＜endl；
           for（j＝i；j＜n＋1；j＋＝i）
           sieve［j］＝0；                          // 0 相当于 false
       }
  }
```

程序执行后，屏幕输出结果与 program4_12 相同。

思考题

1．什么是控制语句？C++中的控制语句有哪几种？
2．什么是复合语句？
3．空语句的作用是什么？
4．分支语句的作用是什么？它如何控制程序的运行次序的？
5．循环语句的重要作用是什么？
6．C++提供了哪几种转向语句？它们一般用于什么场合？
7．如何定义和使用数组？
8．为什么把数组类型归为导出类型，它与基本类型和其它用户定义类型有何不同?%

练习题

1．选择题

(1) 下面关于条件语句的叙述中，( ) 是错误的

A.if 语句只有一个 else 语句

B.if 语句中可以有多个 else if 语句

C.if 语句中 if 体内不能是 switch 语句

D.if 语句的 if 体中可以是循环语句

(2) 下面语句 for 循环的次数是 ( )

```
for(int i=0,x=0; ! x&&i<=5; i++)
```

(3) 当 a=1 ，b=2，c=3 时，执行以下 if 语句 a ，b，c 各为多少。

```
if(a>c)；b=a；a=c；c=b；
```

A.a=1 b=2 c=3

B.a=3 b=1 c=1

C.a=2 b=3 c=1

D.a=3 b=1 c=2

2．设计一个能自动转换公制和英制尺寸的程序，要求用户输入其身高，可以是厘米（cm）数，也可以是英尺（inch）数。已知每英尺为 25.4cm，根据输入做制式的变换，输出用户的公制和英制的身高。%

3．由用户给出任意三个实数，求其中的最大值和最小值。%

4．利用迭代公式：%

$$\begin{cases} y_0 = x; \\ y_{n+1} = \dfrac{1}{3}(2y_n + \dfrac{x}{y_n^2}) \end{cases}$$

计算实数 x 的立方根 $y = \sqrt[3]{x}$，当 $|y_{n+1} - y_n| \le \varepsilon$ 时，y_{n+1} 为 $y = \sqrt[3]{x}$ 的近似值。%

5. 已知某年的 1 月 1 日是星期几，编一个可以计算该年的任意一天(由用户指定)是星期几的程序。注意 2 月份，闰年为 29 天，非闰年为 28 天，可被 4 整除而不可被 100 整除的年份为闰年。由用户输入年份和该年元旦是星期几，并输入任意一个月份日期，由程序计算出这一天是星期几。%

6. 已知圆周率π的计算公式：%

π/4=1－ 1/3 ＋ 1/5 －1/7 ＋ 1/9 －…

编一个程序计算π的近似值。精度要求：%

(1) 计算 200 项得到的近似π；%

(2) 要求误差小于 0.000001 的π的近似值。%

7. 用程序验证 100 以内的奇整数，其平方被 8 除余数为 1。%

8. 某商店出售商品四种：A 商品每公斤 2.75 元，B 商品每个 12 元，C 商品每米 14.5 元，D 商品每台 255 元，超过 3 台优惠 10%，为其设计一个计算价钱的程序。%

9. 设计一个程序，对于用户输入的任意整数 a（a≥0）和 b（b≥2），求出满足 $a \ge b^n$ 的整数 n。

10. 对于整型数组 a [20] 和 b [20]，编制程序完成下列任务：%

(1) 由用户为两个数组输入值；%

(2) 求出两个数组的最大元和最小元；%

(3) 把数组 a 和 b 中的整数分别从小到大和从大到小排序；%

(4) 把两个有序的数组 a 和 b 组成一个长度为 40 的有序数组 c [40]。%

11. 已知二维数组 dig[10] [15]为一字符型矩阵，其每一元素为一数字字符(“0”,“1”,…,“9”)，编一程序，统计矩阵中各个数字的出现次数。%

12. 由 20 个正整数排成一圈，编一个程序找出连续的四个数，其和是最大的，(不小于圈上任何其它连续的四个数之和)。%

13. 设计一个程序，它可以把最近的 20 天的气温值以柱形图的形式显示出来，假设气温在 22～35 度之间(取整)，温度的高低以由‘*’组成的“柱”的高低来表示。

14. 平面上有 n 个点，已知任两点之间的距离为 dist（i，j）（i，j∈ [1‥n])，定义

id（i）＝min i≠j dist（i，j）

为点 i 的孤立半径，求 n 个点中具有最大孤立半径的点。

第五章 函数，函数与运算符的重载

函数概念是 C++语言中最重要的概念之一，函数设计是程序设计的主要部分或实质部分。函数在 C++程序设计中的意义体现在四个方面。

（1）从历史上说，函数的思想来源于子程序，（如 BASIC 语言中的子程序），把程序中反复出现的相同或相近的程序改写成子程序，可以大大缩短程序的长度。函数实际上是参数化的子程序。

（2）从结构化程序设计（SP）的观点来看，函数绝不仅仅是为了缩短程序长度，更重要的是通过函数设计，可以把整个程序要完成的整体的复杂的计算任务，分解为一个个较小的，相对简单的子任务。这种模块化的程序易设计，易阅读，易调试，易维护，较少出错。

（3）从运算的角度说，函数就是 C++语言提供的由用户定义的运算。运算符是系统提供的运算，而函数是由用户自己定义的运算。

（4）作为面向对象程序设计（OOP）语言的 C++，以类为核心，类由数据和方法组成，方法就是对数据的运算和处理，亦即类的函数成员。故函数设计同样是 OOP 的重要组成部分。

在 5.1 节中我们用实例来说明前三点，而对于函数作为类的成员，将在后面的章节中说明。

5.1 三次方程求根程序的设计

按照 Cardan 公式，计算三次方程 $x^3+px+q=0$ 的一个实根的公式为：

$$xr=\sqrt[3]{-\frac{q}{2}+\sqrt{(\frac{q}{2})^2+(\frac{p}{3})^3}}+\sqrt[3]{-\frac{q}{2}-\sqrt{(\frac{q}{2})^2+(\frac{p}{3})^3}} \qquad (5.1)$$

在计算实根 *xr* 的程序中，把计算一个浮点数的立方根的程序作为一个用户定义的函数，而在主程序中两次调用这个函数。

在计算这个比较复杂的公式的过程中，我们采用自顶向下逐步求精的方法，

把一个复杂的问题分解为若干比较简单的子问题，以减少编程的难度，编写用户定义的函数可以帮助我们达到这个目的。

为了从系数 p、q 计算实根 *xr*,把公式（5.1）的计算分解为下面几步：

1） 令 实数 xr＝A+B ;

2） 令 实数 A, B 分别为实数 R,S 的立方根：

3） 令 R = -q/2 + a, S = -q/2－a ; (5.2)

4） 令 a = sqrt ((q/2) *(q/2) + (q/3) * (q/3) * (q/3)) ; （5.3）

实际的计算过程为：

用公式(5.3)计算得到 a ;

用公式(5.2)计算得到 R 和 S ；

求出 R 和 S 的立方根 A 和 B ;

最后得到实根 xr。

其中，计算立方根比较复杂，且须计算两次，用一个函数 cuberoot (y)的形式表示，计算过程可以用下面的程序表示：

```
float  p, q, xr；
cout＜＜"Input parameters p,q: "；
cin＞＞p＞＞q;
float  a＝sqrt ((q/2) *(q/2) + (q/3) * (q/3) * (q/3)) ;
xr＝cuberoot（－q / 2＋a）＋cuberoot（－q / 2－a）;
cout＜＜endl＜＜"The real root of the equation is"＜＜xr；
```

在计算 $\sqrt[3]{y}$ 过程中，使用数学中常用的迭代公式：

$$r_0=y, \qquad r_{n+1}=\frac{2}{3}r_n+\frac{y}{3r_n^2} \qquad (5.4)$$

计算实数 y 的立方根的函数 cuberoot(y)的程序为（精确到小数点后 6 位）：

```
float  cuberoot（float  x）
{
  float root , croot；
  const  float  eps＝1e－6；
  croot＝x；
  do ｛
```

```
        root＝croot；
        croot＝（2*root＋x/（root*root））/3；
    }
    while（fabs（croot－root）＞eps）；
    return（croot）；
}
```

编写函数的语法规则在下节介绍，完整的求解程序如下：

```
//program5_1.cpp
# include＜iostream.h＞
# include＜math.h＞
float   cuberoot（float）；
void main（void）
{
  float   p, q, xr；
    cout＜＜"Input parameters p,q: "；
    cin＞＞p＞＞q;
    float   a＝sqrt ((q/2) *(q/2) + (q/3) * (q/3) * (q/3)) ;
    xr＝cuberoot（－q／2＋a）＋cuberoot（－q／2－a）；
    cout＜＜endl＜＜"The real root of the equation is"＜＜xr；
}
float   cuberoot（float   x）
{
  float root , croot；
    const  float   eps＝1e－6；
    croot＝x；
    do｛
        root＝croot；
        croot＝（2*root＋x/（root*root））/3；
    }
    while（fabs（croot－root）＞eps）；
    return（croot）；
}
```

这个程序包含了一个用户定义的函数 cuberoot（y），**它负责计算浮点型自**

变量 y 的立方根的值。在主函数 main（）中，这个函数两次被调用。试想如果把所有的计算都写到 main（）中，该程序可能变长：

```
void main（void）
{
  float p，q，xr，root；
  const float eps＝1e－6；
  cout<<＂Input parameters p,q；＂；
  cin>>p>>q；
 float  a＝sqrt(q／2*q／2＋p／3*p/3*p／3)；
  float croot＝－q／2＋a,x＝croot；
  do {
        root＝croot；
          croot＝(2*root＋x/(root*root))／3；
       }while(ads(croot  －root)>eps)；
  xr＝croot；
  croot＝  －q／2－a；  x＝croot；
  do {
         root＝croot；
         croot＝(2*＋／(*root))／3；
      }while  (abs(croot－root)>eps)；
  xr＋＝croot；
  cout<<endl<<?----------?<<xr；
 }
```

在程序中,计算 croot 的立方根的运算进行了两次,所以该计算程序要重复两次, 当程序较长,或计算次数更多时，采用“子程序”的方案可以大大缩短程序的长度。

其实，即使在 main（）中只计算 cuberoot（x）一次，也是把它分出来更好。特别是当程序比较复杂时，可以使得程序显得清晰，在 program5_1 中，main（）中不涉及计算立方根的细节，显得简洁，而在 cuberoot（）中只解决一个浮点数的立方根的计算，也很清楚。

另外,还可以把立方根的计算与 C++语言中的运算符和标准函数对应起来，当在程序中对于 cuberoot（y）给出了定义之后，就可以在主函数或其它用户定义的函数中，像运算符或标准函数那样使用了，如 cuberoot（x）的使用与 a+b，sin（x）的使用没有什么区别。

5.2 函数的说明与使用

5.2.1 函数说明

C++程序允许两种函数说明语句的形式，我们把它们分别称为函数原型（或函数声明）和函数定义。

1．函数原型

函数原型（亦称函数声明）用来指出函数的名称，类型和参数，其格式为：

[〈属性说明〉]〈类型〉〈函数名〉（〈参数表〉）；

例如：

```
int  add (int a, int b);
inline void swap (float & s, float & t);
void print (char *) ；
```

属性说明：可缺省，一般可以是下面的关键字之一：inline，static，virtual, friend 等。

inline 表示该函数为内联函数；

static 表示该函数为静态函数；

virtual 表示该函数为虚函数；

friend 表示该函数为某类(class)的友元函数。

其含义的细节在后面有关章节介绍。

类型：指函数的返回类型。C ++语言规定除了特别情形（main（）函数和类的构造函数，析构函数）之外，所有函数都必须在说明中指出返回类型。一个函数可能有多个结果，不一定都用返回值的方式输出（还可通过全局变量，引用参数和指针等方式传出）计算结果，但函数调用作为一个表达式，该表达式的值则只能是函数的返回值。

函数名：一个标识符。

参数表：它可能为空，void 或〈类型〉〈参数名〉,〈类型〉〈参数名〉…的形式。

例如：

```
main()
print(void)
```

```
cuberoot(float x)
add(int a,int b)
```

其中，函数原型中的参数表可忽略参数名，如：

```
int add(int, int)
void swap(float &, float &)
```

它们与函数原型 int add (int a, int b); void swap (float & s, float & t); 是等价的，事实上在编译时，函数原型中的参数名是被忽略的，后者的参数表中的符号`&`表示该参数为引用型参数，用以区别于一般的赋值型参数，其用法在下面介绍。

在上节 program5_1 中，我们给出了函数 cuberoot（float x）的原型和定义。函数原型一般在两种情形下被使用：

①在程序中某函数的调用语句出现在该函数的定义之前，例如在上面 program5_1 中，这时必须在调用语句之前列出函数原型。

②为了类定义的简明清晰，一般把较大的函数成员定义移到类说明之外，这时应把该函数的原型列于类说明之中。类的定义在第八章中介绍。

2．函数定义

函数定义与函数原型的主要区别是它还包括函数体，其格式为：

[〈属性说明〉]〈类型〉〈函数名〉（〈参数表〉）〈函数体〉

属性说明，返回类型，函数名与函数原型一致，参数表中不可省略参数名。

函数体：由｛和｝括起来的复合语句即程序块。

program 5_1 的最后 12 行就是一个函数定义。

从函数说明语句的介绍，我们可以看到程序中各种语句的分类（说明语句，表达式语句，控制语句和复合语句）并不是一个简单的划分概念。函数定义本身是一种说明语句，但其函数体则是一个复合语句。而在函数体内部又可能包含说明语句，表达式语句，控制语句和复合语句。

总之，这种分类和区分是明确的，但它们之间又有互相包含的关系。

5.2.2 函数调用

函数调用是已定义函数的一次实际运行，（与某类型的一个变量和后文中某类的一个对象类似），函数调用是函数定义的一个“实例”。

函数说明中的参数称为形式参数（形参），函数调用中的参数称为实际参数（实参），函数调用一般出现在表达式中或独立形成一个函数调用语句，例如：

```
xr＝cuberoot（－q／2＋a）＋cuberoot（－q／2－a）；
```

swap (a, b)；

函数调用的两要素是函数名和实参表：

<函数名> (<实参表>)

实参表中的参数的类型、个数、顺序必须与函数说明的参数表（形参表）相一致。赋值型的实参可以是一个表达式。

在上节 program5_1 的 main()中两次出现函数 cuberoot(x)调用,其具体的调用实施过程如下：

(1) 根据调用语句中的函数名(cuberoot)在整个程序中搜索同名函数定义；

(2) 对实参数的参数个数，类型，顺序进行核对，判定是否与函数定义中的形参表对应一致，在上例中只有一个浮点型参数；

(3) 根据参数的类型（值参数或引用参数）进行值参数的值传递或引用参数的换名，在上例中即是要把实参表达式的值计算出来赋给形参 x；

(4) 运行函数体代码；

(5) 返回调用点，并返回所要求的函数值，即返回计算结果 croot 的值。

5.2.3 函数的返回

函数的返回完成两项任务：

(1) 把运行控制从函数体返回到函数调用点。在上例中就是在计算 cuberoot (-q/2+a)之后再返回到语句 xr = cuberoot () + cuberoot () 的计算过程中。

(2) 根据返回值要求，返回所需要的数据值。

函数的返回值有下面几种情形：

1.返回 void 类型

如果函数无值返回，应说明为 void 类型。例如：

```
void  print（ ）  { cout<<＂Hello World !＂}；
void  show（ ）  { cout<<＂Wonderful C++!＂}；
```

函数仅需完成打印和显示工作，不需返回任何数据，这类函数调用一般形成一个函数调用语句。

未作类型说明的函数，系统认为是 int 类型函数，应返回一整型值。

2．返回数值类型

最常见的函数是返回一个数值的函数。例如：

```
int  add（int a，int b）；
float  cuberoot（float x）；
```

这类函数的调用表达式可以出现在表达式语句中。

当函数要返回的数值不止一个时，情况比较复杂，一般它可以以结构或类的形式，也可以以结构，数组或对象指针类型方式实现，这样的实例在后面的章节可以见到。

3．返回引用类型

值返回方式是 C 和 Pascal 语言中唯一的返回方式，C++语言提供的引用返回概念是其特有的一种很强的功能，当函数定义中把该函数说明为某类型的引用类型时，该函数调用后返回的不单是值，而是包含返回值的变量（或对象）。由于返回引用与引用类型有关，所以这样的实例将在下节介绍。

5.2.4 函数的参数

函数的参数的设置和使用是函数设计中非常重要的部分。

C++语言允许函数无参，有一个或多个参数，而且还支持不定个数参数的函数。

(1) 无参函数：其函数说明为下列形式：

```
void print（void）；        // 输出指定的数据
int getx（）；              // 输出变量 x 的值
```

用 void 或空表示无参。

(2) 一个或多个参数：

多数函数有一个或多个确定的个数、确定的类型和顺序的参数，例如：

```
void sort（int n，char* array）  {…}；//对 n 元整形数组 array
                                        //进行排序
float max（float a，float b，float c）  {…}；
                                  /* 求三个实数 a,b,c 中最大者。*/
```

等等。定义中应注意参数表的组成。因为在 C++语言中，不同的函数是根据函数名和参数表二者来区分的，只有二者完全一致才是同一函数。（有关指针类型，将在下章介绍）

(3) 不定个数参数：

有些应用问题中参数个数是变化的。例如设计一个电话计费函数，为了计算通话费，不同的通话类型（如市话，长途，数据与通讯，BP 机等）有不同数目的参数。

处理参数个数不定的情形，可有不同的途径。例如：

```
void sort（int n，float * a）；
```

这个函数可对 n 长的浮点数组进行排序，n 是变化的；由于 a 是数组的首

元指针，因此这个函数实际上是可以进行对任意多个浮点数排序的处理。

C++语言有的版本还提供一些库函数，支持处理形如：

```
void abc（int i,…）；
```

的不定参数函数。

C++语言,允许参数表中包含无名参数，主要是为了区分函数，例如：

```
int f（int a，int b）{return  a+b*b；}
int  f（int a，int b，int）{return  a*a+b；}
```

两个不同的函数同名，但由于第二个函数包含一无名参数，使得在调用时能够被区分，f（x，y）是第一个函数的调用，f（x，y，0）是第二个函数的调用。

C++程序还允许为函数定义可缺省参数，这种函数调用时具有灵活性，例如：

```
int sqrsum（int a，int b，int c＝0）{
  return  a*a＋b*b＋c*c；
}
```

其中参数 c 为可缺省参数，下面的调用方式都是合法的（x，y，z 为 int 型变量）：

```
sqrsum（x，y，z）
sqrsum（x＋y，x－y）
sqrsum（x，y）
```

参数表中可有任意多个参数指定为可缺省参数，但所有可缺省参数必须列后。在调用该函数时，一般不允许部分缺省，即要么省去全部缺省参数，要么一个也不省。

5.2.5 值调用与引用调用

C++语言在进行函数调用时，对参数的处理有两种方式，赋值型和引用型，即值调用方式和引用调用方式。前者是普通的形式，在 C 语言中只有这种方式；C++语言中增加了引用调用形式，这种形式与 pascal 语言中的变量参数调用方式相似。

1．赋值调用方式

在执行函数调用时，在检查函数名及参数表之后，首先为值参数分配内存，然后计算各对应的实参表达式，并把计算的值赋给刚刚创建的参数变量，进而开始函数体的运行。

赋值形参：在函数定义的参数中，除了被说明为引用（&）的参数之外，其余所有类型的形参都属于赋值形参。

凡是赋值形参，在函数的每次调用时，都必须为每一个赋值形参创建一个新的参数变量。

实参表达式：另一方面，函数调用语句中，与赋值形参相对应的实参可以是指定类型的常量、变量或表达式。在执行函数调用时应把该表达式的值计算出来，作为初值赋给刚刚为赋值形参创建的参数变量。这是赋值调用方式名称的由来。

为赋值形参创建的参数变量是局限于函数体运行的局部变量，它作为该形参的一个实例，参加函数体程序块的这次运行，一旦运行完毕， 这个参数变量就被撤消。例如，定义函数：

```
int multiple3(int n){
  n=n*3;
  return n;
}
```

调用函数 multiple3 三次：

```
int n=5;
cout<<multiple3(2)<<endl;
cout<<multiple3(n)<<endl;
cout<<multiple3(3 * n)<<endl;
cout<<n<<endl;
```

执行结果输出为：

```
6
15
45
5
```

这里变量 n 虽与形参 n 同名，但却是没有直接的关系的两个不同的参数，即使以变量 n 作为实参，调用该函数，其函数体中对于形参 n 所进行的赋值（或任何别的操作）与变量 n 是无关的。仅仅传递实参的值，实参本身与函数调用过程无关，变量 n 在调用之后其值没有改变。

实参表达式一般应使用只计算值的表达式如 3*n，x+y，&x，multiple3（n）等等，对于可能改变变量值的表达式如 n++，m＝s+m*3 等，应注意的是，当一个函数有多个赋值形参时，在进行值传递过程中，多个实参表达式计算的次序。例如：

函数 int reaipara (int a,int b) { return 2*a+b;};

… n=5; m=realpara（n++，n--）；…

的调用过程决定于两个实参表达式计算次序，先计算 n++赋值给形参 a， 还是先计算 n--赋值给形参 b,显然结果是不同的， C++语言对此并无明确规定，它将依赖于具体的编译系统。读者如采用此种形式，应首先测试一下你所采用系统的处理方式。

2．引用调用方式

引用形参：函数定义的参数表中，名字前加上符号＆的参数为引用形参。例如：

```
void swap (int& a, int& b) {
  int temp=a;
  a=b;
  b=temp;
}
```

形参 a，b 为引用形参，这时其函数原型可写为：

```
void swap (int&, int&);
```

引用形参在调用过程中的参数传递机制不同于赋值形参。其要点是：

（1）函数的调用语句中对应于引用形参的实参必须是同一类型的变量，非变量的表达式则不允许。

（2）参数传递的内容不是实参的值，而是地址，其实际的效果是令对应的引用形参在调用过程中，作为一个变量名指向作为实参的这个变量，与赋值形参的不同在这里体现出来，在引用调用过程中并不创建新的参数变量！

（3）在函数体程序块的运行中，引用形参的每次出现，由于它现在已经是指向实参变量，因此相当于全用实参变量所代替。即起到了所谓的“换名”的作用。

（4）在函数体程序运行结束，控制转回调用点时，该引用形参与实参变量的对应关系也就终止了。但是在调用过程中对于这个实参变量的所有处理和操作的结果，却保留下来。这一点也是区别于赋值调用的。例如，在使用上面的 swap 函数时：

```
int x=5, y=3, z=7;
swap (x, y);              //结果 x=3,y=5,
swap (y, z);              //结果 x=3,y=7,z=5,
```

并不需要借助于指针类型，直接可以把函数处理结果带出函数。

C++语言提供的引用调用方式，表面上看它与利用下章的指针类型的形参的赋值调用所起的效果相同，实际上它优于指针方法。设计函数在下面两种情形时，①需要改变某些变量的值（上述函数 swap 就是一例）；

②对于占内存较多的数据参数，为了不另建新的参数变量以节省内存，建议采用引用参数。在后一种情况，为了保证实参不在函数中被修改，可在形参说明中加上 const 说明，例如：

complex add（const complex& a，const complex& b）；

而对于赋值形参，则无此必要。

5.2.6 内联函数

内联（inline）函数的设置是 C++不同于 C 的特征之一。

（1）在 C++程序中符合下列条件的函数：

①函数说明前冠以“ inline ”关键字的函数；

②类内定义的函数成员,将在第七章介绍。

（2）在编译过程中，凡内联函数，系统把它的执行代码插入到该函数的每个调用点，从而使程序执行过程中，每次该函数调用时不需控制转移，但该函数代码可能有多个拷贝出现在目标程序中。也就是说，把多次调用的函数说明为内联函数，会使目标程序占用的空间加大，而运行时间得到节省。

（3）为了优化程序、提高程序的运行效率，一般把函数体短小而又频繁调用的函数说明为内联函数较好。

（4）利用编译预处理的宏定义方式，也可以实现类似于内联函数的功能。不过，内联函数的方式更为方便和可靠。例如，下面几个内联函数定义可以用带参数的宏定义方式实现：

```
inline int MAX (int a, int b) {if (a>b) return a; return b;}
inline float ABS (float a) { return ( a>=0)? a: 0-a; }
inline float PERCENT (float a, float b) {return 100.0 * a / b;}
inline void SWAP (int a, int b) {int t; t=a; a=b; b=t; return;}
inline bool ISODD (int x) {return (x %2 = = 1)?1: 0; }
```

对应的宏定义为：

```
# define MAX (a,b)  ((a>b)? a: b)                    // 求较大者
# define ABS (a)  (( a >= 0 )? a: 0-a )              // 求绝对值
# define PERCENT (a,b)  (100.0 * a / b )             // 求百分比
# define SWAP (t, a, b)  {int t=a; a=b; b=t;}        // 交换 a,b 的值
# define ISODD (x)  ((x %2 = = 1) 1: 0 )             // 判断 x 是否为奇数
```

虽然这些带参数的宏定义的功能与对应的内联函数基本一致，但仍然是有

差别的，宏定义中的参数和计算结果没有类型说明，编译时不可能进行类型检查，是不安全的，更无法区分赋值参数和引用参数，很容易出错。因此，C++语言的编程中，当某段计算短小而又经常被重复时，建议采用内联函数，少用宏定义实现。

5.3 函数的嵌套与递归

5.3.1. 函数嵌套

一个函数的函数体中包含一个或多个函数调用语句，即称为函数嵌套。

嵌套的含义是，如果函数 A 要调用函数 B，也就是说，函数 A 的定义要依赖于函数 B 的定义。因此函数 B 的定义或函数 B 的原型必须出现在函数 A 的定义语句之前。

另一方面，函数 A 调用函数 B，在调用 A 的过程中，即执行 A 的函数体过程中，调用 B，也就是中途把程序控制转到 B 的函数体，在执行结束后再返回到 A 的函数体中。

函数嵌套调用所占用的空间（如赋值参数的创建等等）用堆栈（stack）的方式管理。一般这种堆栈所分配的空间是有限的，因此函数互相嵌套的层数也是有限的，依编译系统不同，其允许的嵌套层数也可能不同。从下面的程序中可以了解函数嵌套调用机制和栈操作的原理。

```
// program5_2
# include <iostream.h>
void f1 (int,int);
void f2 (int);
void main () {
  int a, b ;
  cout << “a:” ;
  cin >> a ;
  cout <<”，    b:”;
  cin >>b;
  f1 (a, a+b);
   cout << endl << a+b << endl;
}
```

```
void f1 (int x, int y) {
  int m = 2;
  x *= m ;
   y++;
  f2 ((x+y)/m);
}
void f2 (int p) {
  if (p <= 100) cout << endl << p;
}
```

其运行结果可能是：

```
a:23   b:7
38
30
```

在这个例子中,主函数 main **调用函数** f1, **函数** f1 **又调用函数** f2**，其运行机制讨论如下：**

为了简化，函数 f1 **和** f2 **都只有整型的赋值参数，且没有返回值。**

在主函数中执行调用语句 f1(a, a+b); **时，首先计算表达式** a **和** a+b **的值，传给** f1 **的参数** x **和** y**，然后把程序控制转到** f1 **的函数体。**

在函数 f1 **中执行调用语句** f2 ((x+y)/m); **时， 则首先根据** f1 **的实参变量** x,y **和局部变量** m **的当前值，计算表达式（**x+y**）**/m **的值，传给** f2 **的实参变量** p**，然后把程序控制转到** f2 **的函数体。**

函数 f2 **的这次运行结束，程序控制转回到调用它的函数** f1 **中的调用点，继续函数** f1 **的运行，函数** f1 **的这次运行结束后，程序的控制再转回到调用** f1 **的主函数继续运行。**

为了保证返回后继续正常运行，函数调用过程开始时需要做更多的事情：

1**） 建立被调用函数的栈空间；**

2**） 保存调用函数的运行状态和返回地址；**

3**） 传递参数；**

4**） 把程序控制转交给被调用函数。**

程序 program5_2 **在运行中，函数** main,f1,f2 **的栈空间按照“ 后进先出 ”原则，下面是在栈区它们的栈空间分配示意图：**

		栈区
f2（）	实参 p	38
		返回地址
		f1()运行状态
f1（）	变量 m	2
	实参 x	23 —> 46
	实参 y	30 —> 31
		返回地址
		Main()运行状态
Main（）	变量 a	23
	变量 b	7
	参数	无
		返回地址
		OS 运行状态

图 5.1 函数调用过程中的栈结构示意图

5．3．2 函数的递归

递归函数的设计在程序设计中占有重要的地位。

函数 A 在其函数体中直接包含它自己的调用语句，这种调用称为直接递归调用，函数 A 称为（直接）递归函数。

函数 A 在其函数体中间接地包含对它自己的调用，例如 A 调用函数 B,但函数 B 又调用函数 A，则函数 A 称为（间接）递归函数。下面是一个（直接）递归函数的例子：

```
float fac（int n）{
  if  （n＜2＝return 1；
  return  fac（n－1）*n；
}
```

递归函数的实际计算过程与递归次序相反，例如，设 n=4，计算 fac(4)的递归次序是：fac(4)－>fac(3)－>fac(2)－>fac(1)，而实际计算过程则正好相反：fac(1)=1, fac(2)=fac(1)*2=2，fac(3)=fac(2)*3=6，fac(4)=fac(3)*4=24.

从下面的实例可以进一步了解递归过程的执行机制。

反序输出：从键盘输入 10 个 int 型数，而后按输入的相反顺序输出它们。

```
// program5_3
# include <iostream.h>
```

```
void inv (int n)
{
    int i;
    cin>>i;
    if (n>1)
        inv (n-1);
    else
        cout<<"---- The result ----"<<endl;
    cout<< i <<" ";
}

void  main(void)
{
    cout<<"Input 10 integers:"<<endl;
    inv(10);
    cout<<endl;
}
```

可以有下面的执行结果：

```
Input 10 integers:
1 2 3 4 5 6 7 8 9 10
---- The result ----
10 9 8 7 6 5 4 3 2 1
```

在许多情形下递归函数易写易读，像解著名的 Hanoi **塔问题的递归函数，其递归程序很短，也极易理解(见** 5.6 **节)，但是，如果不用递归方法，程序将十分复杂，很难编写。下面的实例可以说明这种情形。**

反序输出一个正整数的各位数值，如输入 231**，应输出** 132**。**

```
// program5_4
# include <iostream.h>
  int conv（int n）{
     int  m；
     if  (n<10) return n;
     cout  <<  n%10;
     return  conve(n/10);
```

```
}
void main(void){
  int t;
  cout <<"Input a positive number:";
  cin >> t;
  cout << endl;
  conv(t);
}
```

输出结果为：

```
Input a positive number: 4781
1874
```

这个函数的设计包含了函数的递归，显示了递归设计技术的效果。如果不用递归函数设计，程序不如递归形式清晰：

```
int conv（int n）{
  int m；
  if (n<0) cout << "Please input a positive number!";
  else do {
          cout << n%10;
          n=/10;
       } while (n!=0);
}
```

两个函数在处理输入为负数时，方法略有不同。

在一般情况下，读者应注意：

(1) **无论是直接递归还是间接递归都必须保证在有限次调用之后能够结束。例如函数** fac **中的参数** n **在递归调用中每次减** 1**，总可达到**<2 **的状态而结束。**

(2) **函数调用时系统要付出时间和空间代价，在环境条件相同的情形下，总是非递归程序效率较高。**

5.4 函数与运算符的重载（overload）

5.4.1 函数重载

函数重载实际上是函数名重载，即支持多个不同的函数采用同一名字。例如：

```
int abs（int n）{return（n＜0？-n：n；}
float abs（float f）{if（f＜0）f＝-f；  return  f；}
double abs（double d）{if（d＜0）return-d；return  d；}
```

三个函数都是求绝对值，采用同一个函数名，更符合人们的习惯（这一点在 C 语言和其它语言中是不允许的。它们规定，在同一个程序块中一个名字只允许有一种含义）。在程序中经常出现这样的情况：对若干种不同的数据类型求和，虽然数据本身差别很大（例如整数求和，向量求和，矩阵求和），具体的求和操作差别也很大，但完成不同求和操作的函数却可以取相同的名字（例如 sum，add 等）。

许多差别很大的打印函数可以都用名 print，显示函数可以都用 display，从键盘或文件获取信息都称作 get，发送称为 send，接收称为 receive 等等。

函数名的重载并不是为了节省标识符（标识符的数量是足够的），而是为了方便程序员的使用，这一点很重要。

实现函数的重载必须满足下列条件之一：

（1）参数表中对应的参数类型不同；

（2）参数表中参数个数不同；

（3）参数表中不同类型参数的次序不同。

例如：

```
void print（int）；          //整型
void print（point）；        //类 point 的对象
int sum（int ，int）；
int sum（int ，int ，int）；
int get（int n，float a[ ]）；
int get（int n，float a[ ]，int n）；
```

在定义同名函数时应注意：

（1）返回类型不能区分函数，例如：

```
float add（int float）；
int add（int float）；          //错误
```

在调用时系统无法区分选择对应函数。

（2）采用引用参数不能区分函数，例如：

```
void print（double）；
```

```
void print（double &）;          //错误
void print（const double &）；  //错误
```

（3）有些派生基本类型的参数虽然可以区分同名函数，但在使用中必须注意，例如：

```
int abs（int）；
unsigned int abs（unsigned int）；
```

对这种函数的调用必须注意：

```
int a=3，b；
unsigned int c=5，d；
```

调用语句

```
b＝abs(a)；
d＝abs(c)；
b＝abs（-3）；
b＝abs（3）；
d＝abs（3*a）；
```

其中 abs（3）因实参 3 既可以是 int 型，也可以 unsigned int 型，故这里可能出现二义性。

（4）包含可缺省参数时，可能造成二义性，例如：

```
int sum（int a，int b，int c＝0）；
int sum（int a，int b）；
```

程序设计中应避免这种情形出现。

上面的二义性的情形是指系统面对某一函数调用语句，有两个或两个以上同名函数与之匹配；而另一种情况下，由于数据类型之间的复杂转换关系也可能造成找不到与之匹配的函数定义。

遇到无准确匹配的函数定义时，C++系统并不马上按出错处理，它按下面的方式处理：

（1）通过数组名与指针变量，函数名与函数指针，某类型变量与 const 常量之间的转换，再查是否可实现匹配；

（2）把实参类型从低到高（按字长由短到长）进行基本类型及其派生类型的转换，再检查是否可匹配；

（3）查有无已定义的可变个数参数的函数，如有把它归为该函数。

在进行上述尝试性的处理之后可能出现仍无匹配或匹配不唯一的情况，这时可能输出出错信息或错误地运行。

5.4.2 可重载运算符

C++语言中的运算符实际上是函数的方便表示形式，例如，算术运算符"+"也可以表示为函数形式：

```
int add (int a, int b) { return a+b;};
```

这时，a+b 和 add (a,b) 的含义是一样的。因此，运算符也可以重载。

C++语言规定，大多数运算符都可以重载，我们把它们列于后：

单目运算符：

-，~，!，++，--，new，delete

双目运算符：

+，-，*，／，%	（算术运算）
&，｜，^，<<，>>	（位运算）
&&，||	（逻辑运算）
==，!=，<，<=-，>，>=	（关系运算）
＝	（赋值运算）
＋＝，－＝，*＝，／＝，%＝	（赋值运算）
^＝，＆＝，｜＝，>>＝，<<＝	（赋值运算）
，	（逗号运算）
<<，>>	（I／O 运算）
（），[]	（其它）

关于这些可重载运算符，有几点需要说明：

（1）所列可重载运算符几乎包含了 C++的全部运算符集，例外的是：

限定符.，::，

条件运算符？ :，

取长度运算符 sizeof

它们不可重载（不可赋予不同的操作）。

（2）在可重载的运算符中有几种不同情况：

①算术运算符，逻辑运算符，位运算符和关系运算符中的<，>，<＝，>＝，这些运算都与基本数据类型有关，通过运算符重载函数的定义，使它们也用于某些用户定义的数据类型，这是重载的主要目的。

②赋值运算符＝，关系运算符＝＝，！＝，指针运算符＆和*，下标运算符 [] 等，它们的运算所涉及的数据类型按 C++程序规定，并非只限于基本数值类型。因此，这些运算符可以自动地扩展到任何用户定义的数据类型，一般不需作重载定义就可“自动”地实现重载。

不过在某些特定情况下，也可由用户定义其重载函数。这种情形相对较少，大多包含指定的特殊操作或程序员灵活的构思。

③单目运算符++和--实际上各有两种用法，前缀增（减）量和后缀增（减）量。其运算符重载函数的定义当然是不同的，对两种不同的运算无法从重载函数的原型上予以区分：函数名（operator ++）和参数表完全一样。为了区别前缀++和后缀++，C++语言规定，在后缀++的重载函数的原型参数表中增加一个int 型的无名参数。其原型形式为：

```
前缀++：  〈类型〉operator  ++  （ ）          //作为类成员
          〈类型〉operator  ++（〈类型〉）      //作为类外函数
后缀++：  〈类型〉operator  ++  （int）        //作为类成员
          〈类型〉operator  ++（〈类型〉，int）  //作为类外函数
```

关于减量运算符--的重载方式相同。其调用方法为：

```
前缀++：  ++a 或 a.operator++（）
                operator++（a）
后缀++：  a++ 或 a.operator++（0）
                operator++（a，0）
```

5.4.3 运算符重载函数的定义

运算符的重载是一个特殊函数定义过程，这类函数总是以 operator<运算符>作为函数名。

由于运算符重载总是要涉及用户定义的数据类型，而 C++程序中的用户定义类型大多以类（结构，联合）的定义形式出现，因此，其实例须在第七章以后引进，因此，在本节仅对 enum（枚举）类型作为重载的简单实例。

假设程序中定义了一个枚举类型的 bool 类型：

```
enum bool{FALSE，TRUE}；
```

用运算符＋（双目），*（双目），－（单目）来表示或、与、非运算是十分方便的：

```
bool operator + （bool a ，bool b）{
  if（(a==FALSE）&&（b==FALSE)） return  FALSE；
  return  TRUE；
}
bool  operator*（bool a，bool b）{
  if（(a==TRUE）&&（b==TRUE)） return  TRUE；
```

```
    return FALSE;
  }
  bool operator-(bool a){
    if (a==FALSE) return TRUE;
    return FALSE;
  }
```

这样我们就为 Bool 型的变量或常量定义了双目运算+，*和单目运算—，实现了布尔量的或，与，非的运算（如果改用运算符&,|,!也可以起到同样的作用），例如，我们可以在程序中方便的表示其运算：

```
b1=b1+b2;
b1=-b3;
b1=(b1+b3)*FALSE;
```

运算符重载函数的调用可有两种方式：

1. 与原运算符相同的调用方式，如上例中的 b1+b2，b1*b2，等等。

2. 一般函数调用方式，如 b1+b2，也可以写为 operator+（b1，b2）被重载的运算符的调用方式，优先级和运算顺序都与原运算符一致，其运算分量的个数也不可改变。

3. 运算符重载主要用于用类的形式定义的用户定义类型，例如，复数类型，集合类型，向量类型等等，通过运算符重载把人们习惯的运算符引入到计算操作之中，会收到很好的效果。这样的实例将在第七章介绍。

5.5 函数与 C++程序结构

函数是 C++程序中重要的语法单位和程序模块，在 2.4 节中，已经介绍了有关编译预处理、主函数、C++程序的 SP 框架和 OOP 框架等涉及程序结构的初步知识，本节将进一步讨论这方面的内容。

5.5.1 库函数的使用

库函数又称标准函数，是 C++语言编译系统为用户提供的内部函数，其编写与一般用户定义的函数相同，程序员可在程序中直接使用，但是要在程序开头说明库函数所在的头文件名，例如：

```
#include <iostream.h>
#include <math.h>
```

C++系统中有一个很大的标准函数库（和标准类库），包括许多在各种程序中常用的基本任务处理函数，这些库函数被分成不同的组，例如，数学计算、字符处理、字符串处理、I/O 操作、图形处理等等，当#include 命令包含了某个头文件时，实际上就是把这一组库函数的说明包含在程序中。因此，库函数的使用与一般用户定义函数基本相同，在编译过程中，系统能自动把库函数的目标代码与用户程序的目标代码进行链接。

在本书的附录 B 中，介绍了 Visual C++提供的部分主要库函数，在系统的 Help 文档中，有关于库函数的完整说明。

在本节讨论程序结构时，把库函数视为一般的用户定义函数，因此，在 C++程序中，只有主函数和用户定义函数两种。

5.5.2 SP 框架结构

按结构程序设计（SP）思想设计的程序结构称为 SP 框架。函数是 SP 框架的核心。

一个 SP 框架的完整 C++程序由下面几部分组成：

1） 一个主函数。它可调用其它函数，但不能被调用。

2） 任意多个用户定义函数。都处于同一“等级”，可以互相调用。

3） 全局说明。在所有函数定义之外的变量说明和函数原型。

4） 预处理命令。在进行预处理后，这部分被取代。

5） 注释。只起方便阅读的作用，编译后被删除。

实际上，这样的一个程序，无论大小，都是由一个主函数、若干个自定义函数和全局说明组成。

对于比较大的程序，可以把它们划分为几个程序文件，这些程序模块可能由一个或多个程序员编写，最简单而有效的划分方法是：

① 根据主函数和各用户定义的函数的功能及相互关系,把它们划分成若干个.CPP 文件。

② 按与每个.CPP 程序文件中的函数有关的全局说明组成一个或多个.h（头）文件。

③ 程序中使用的库函数组成的若干.CPP 文件和对应的.h 文件。

在预处理命令的帮助下，一个 C++程序被划分为若干.CPP 和.h 程序文件。在包含命令的帮助下，这些文件形成了一个有机的整体。

在这样的模块结构中，各个.CPP 文件是全部函数的划分，它们组成了程序代码的主体。另一方面，由于任何函数的调用都必须以预先声明该函数（列出

其原型）为前提，因此包含若干全局说明的头文件，可能在各个模块中反复出现，这样一方面给程序员提供了方便：要调用哪部分函数，只须在程序的开始把有关的头文件包含进来即可。

采用这种方法往往产生一个问题，同一个头文件在各程序模块中多次被包含的情形很容易出现，虽然，这对程序员没有什么代价，但在预处理过程之后提交编译的程序中，就可能包含许多重复的程序段，因此，在许多划归一个头文件的程序中，利用条件编译命令有效地避免了这种情形的发生。见下例：

```
//ABC.h
        #ifndef ABC_H
        #define ABC_H
        //头文件的程序内容
        #endif
```

预处理程序第一次碰到

```
        #include "ABC.h"
```

时，把上面的 ABC.h 的程序内容嵌入到 include 命令的位置。然后执行条件编译命令：

```
        #ifndef ABC_H
```

由于尚未对 ABC_H 进行宏定义，故处理该文件全部内容，包括执行宏定义：

```
        #define ABC_H
```

和把后面的 C++程序（全局说明）提交给编译器。如果，头文件 ABC.h 在程序中多次被包含，那么在预处理器第二次遇到

```
        #include "ABC.h"
```

时，它仍把 ABC.h 的内容嵌入，但这时在进一步执行条件编译命令

```
        #ifndef ABC_H
```

时，由于宏 ABC_H 已被定义过，因此预处理器跳过这段程序（直到文件尾的#endif)。这就保证了一个头文件虽多次被包含，但在编译时却不必重复编译多次。

虽然，前面已经指出本章介绍的以函数设计为中心的 SP 框架在当前和今后仍有保留的必要,而且 C++语言作为结构程序设计语言,它与 C 语言和 Pascal 语言相比，有其明显的优势。但是，我们还是要向读者推荐更好的面向对象的程序设计方法。为此，简单地讨论一下 SP 方法的缺点：

（1）SP 方法没有充分利用 C++语言所提供的有力手段。类和对象是 C++语言的主要特征，但在 SP 框架下，类只起到一个配角的作用，它可能出现在全局说明中，作为一种新的用户定义的数据类型出现，而类作为程序模块划分的作用却未得到利用。

(2)以函数为中心对程序进行模块划分,主要是依照程序模块的功能特征，其划分具有相当大的随意性。在大多数情况下，如果划分不科学不合理，自然会影响到整个程序的可读性、可维护性、可重用性等等。

(3) C++语言中的函数没有层次关系，除了主函数之外，所有的函数都是“平等”的，可以说是一个无序的集合。由这样一些函数组成的模块，其本身没有必然的强内聚性。函数间，模块间的联系较多，不利于程序的编制、调试、修改、扩充和重用。

总之，C++语言仍然支持 SP 框架的程序设计，当开发小规模、短期使用的程序时，这种方式仍然有用。若与面向对象程序设计方式相比，它包含若干不可克服的缺点，学习 C++语言程序设计，显然应该投入更多的精力，努力掌握 OOP 的思想和技术。

5.5.3 函数间的数据传递

函数是程序的基本模块，这种划分是十分必要的，一个程序中的各个函数又是互相联系的，这种联系体现为函数间的数据的传递。没有函数间的数据的传递，程序不能成为一个有机的整体，就不能工作。

C++语言为函数间的数据的传递提供了多种机制，这些机制是学习程序设计所必须掌握的。

1．通过赋值参数和返回语句

这是最简单、最安全的数据传递方式，对于一个被调用函数来说，通过赋值参数从调用它的函数接收数据，通过返回（return）语句向调用它的函数传回数据，两者都是单方向的，学习者容易掌握且不容易出错。

2．通过全局变量

全局变量的定义域可延续到整个程序执行结束，因此，只要在函数中没有把该全局变量名说明为其它变量，在所有的函数中都可以直接访问它，也就是说，函数间的数据传递还可以通过全局变量实现，这种传递可以是双向的，例如：

```
#include <iostream.h>
int length,width,area;

void rectangle()                    //无参函数 area()
{
   area = length * width;
}
```

```
void main()
{
cout<<"input length and width:";
cin >> length >> width;     //输入全局变量 length，width 的值
cout <<"the area of the rectangle is :";
rectangle ();                       //调用无参函数 area()
cout << area << endl;
}
```

在这个程序中,全局变量 length,width,area 在函数 main()和 rectangle()中都是可见的，rectangle()的输入数据通过全局变量 length,width 从主函数传进，计算结果通过全局变量 area 传出，因此，全局变量可以为函数进行数据的双向传递。在早期的程序设计语言，例如 BASIC 语言中，子程序（类似于函数）的数据传递主要以这种方式进行，不过，从结构程序设计（SP）的观点来看，这种方式不够安全，在较大的程序运行过程中，函数被调用的次数很难预测，容易出错，因此，采用参数和返回语句的方式更符合结构程序设计的思想，

3.通过指针类型参数和引用参数

通过指针类型的赋值参数给函数传进的不是数据本身，而是数据的地址，在函数体的运行时，可以从该地址获得数据或输出数据，其作用类似于全局变量，数据的传递可以是双向的，指针类型的使用将在第六章介绍。

采用引用参数的形式完成函数间的数据传递的原理在 5.2 节。

4.函数的数组类型参数

数组可以作为函数的参数，其调用机制与一般的赋值参数不同，由于一般数组占用较大空间，不宜在函数调用的栈空间分配一片数据存放地址，因此，C++语言规定对于数组参数在调用时只传递该数组的首地址，实际上，就是该数组的指针，其效果与指针类型参数相同，这种数据的传递也是双向的。

数组参数只传递数组地址，不仅节省了存储空间，也节省了数据的传输时间，使得数据的传入和传出更为方便，不过，数组的 size 信息未包含在作为参数的数组名中，因此，一般把数组作为参数的同时，也把该数组的大小作为一个整型参数，下面的实例说明其用法。

计算数组 a 中前 n 个整数累加和的递归函数 sum：

```
#include <iostream.h>
int sum(int a[], int n)
{
```

```
    if ( n==1 )
        return a[0];
    else
        return ( a[n-1]+sum(a, n-1) );
}
void main()
{
    const int n=6;
    int A[n];
    cout<<"Input "<<n<<" integers:"<<endl;
    for(int i=0; i<n; i++)
        cin>>A[i];
    int s=sum(A, n);
    cout<<"s="<<s<<endl;
}
```

程序执行后的输出结果为：

```
Input 6 integers:
22 4 -2 9 100 3
s=136
```

5.5.4 变量与函数的作用域

程序中出现的所有名字（标识符）都必须说明，每个名字（变量名、常量名、参数名、函数名、类名、对象名等）都在程序的一定范围内有意义，就是该名字的作用域。在第三章已经初步讨论了变量的生存期与作用域，这里我们做进一步的介绍。

1．外部存储属性与静态存储属性

一个 C++程序由一个主函数和若干用户定义函数（或类）组成，当程序的规模较大时，整个程序可能被划分为几个程序文件，关键字 extern(外部存储属性)和 static(静态存储属性)可以规定所说明的变量名或函数名的作用域在一个程序文件范围内还是扩展到程序文件之外。

外部存储属性 extren. 在所有的函数、类和名字空间之外说明的变量的作用域从被说明点开始，到所在的程序文件结束（注意，其生存期则是直到程序结束）。在另一个程序文件中如果需要使用同一个变量的话，必须把这个变量说明为外部（extern）的，表示这个变量的说明不在本程序文件中，而在原文件

中。例如；

程序 prog_5 由两个程序文件 prog_5a.cpp,和 prog_5b.cpp 组成，变量 t 在 prog_5a.cpp 中被说明，若在 prog_5b.cpp 中使用同一个变量 t，必须在后者中说明为外部变量。

```
//prog_5a.cpp
#include <iosteam.h>
int t;
void fa(int s);
int  fb();
void main(){
  t=5;
  fa(t+1);
  cout << t <<endl;
}
void fa(int s){
  cout << s + fb() <<endl;
}
//prog_5b.cpp
extern int t;
int fb(){
  t *= 2;
  return t ;
}
```

运行结果为：

```
16
10
```

在 prog_5b.cpp 中使用的变量 t 是在 prog_5a.cpp 中定义的，如果没有外部说明语句 extern int t;,变量 t 的作用域仅在 prog_5a.cpp 中有效。这里对变量 t 的外部说明并不为其分配新的空间，而是指出变量 t 就是外部文件 prog_5a.cpp 中定义的那个变量 t。

另外要说明的是，为什么在文件 prog_5b.cpp 中定义的函数 fb()却可以在文件 prog_5a 中调用，而不必作外部说明呢？这是因为 C++语言规定，函数的

说明和原型总是默认为外部（extern）的，换句话说，（类外）函数的作用域是整个程序。

静态存储属性 static. 对于变量和函数的说明，静态存储属性可以使其作用域为所在的程序文件，分三种情形：

1）对于全局变量：增加 static 说明，使其作用域仅限于本程序文件。例如：

```
int n;
```

全局变量 n 的生存期为整个程序；作用域为本程序文件，但可以通过外部属性说明把它的作用域扩展到整个程序。

```
Static int m;
```

静态全局变量 m 的生存期为整个程序，作用域为本程序文件，不可扩展。

2）对于局部变量：增加 static 说明，使其生存期和作用域扩展到整个程序，例如：

```
{ ……int n; ……}
```

局部变量 n 的生存期和作用域限于其所在的语句块或函数体，所占用的空间位于临时数据区。

```
{ ……static int m; …… }
```

静态局部变量 m 的生存期为整个程序（参阅第三章关于静态变量的介绍），程序第一次执行该语句时，为变量 m 分配空间，赋缺省初值 0，第二次以后执行该语句时，不再做初始化工作，每次执行该程序块时，变量 m 都是指向同一地址。

3）对于函数：一般（类外）函数的生存期和作用域为整个程序，静态属性的函数的作用域被限于所在的程序文件，这时，该函数不能在其它程序文件中使用。

2．名字的生存期与作用域

程序中的名字（变量名、函数名等等）的生存期与其对应的语法实体被分配的存储空间相关，全局的、静态的、外部的语法实体被分配到全局数据区，在整个程序运行过程中，全局数据区的存储分配是不变的，因此，这些名字的生存期为整个程序。局部的（在函数内，程序块内说明的）语法实体被分配到局部数据区（栈区等），这种分配是临时的，一旦该函数体或程序块这些结束，所分配的空间被撤销，因此，局部名字的生存期从被说明开始，到程序块结束。从这里也可看到，生存期与一个程序被划分为一个或多个程序文件是没有关系的。

名字的作用域与程序文件有关，作用域的概念是从这个名字能够被引用的

角度考虑的，一个名字在某处可以被引用，也称它在这里是可见的，它可被引用的范围，就是该名字的作用域。在第三章已经指出，如果在 int 变量 a 的作用域内又说明了一个同名 float 变量 a，则前者的作用域应减去后者的作用域。

C++程序中的名字（标识符）的作用域有下面几个等级：

整个程序．函数（类）的作用域，全局变量（在 extern 说明帮助下）的作用域。

程序文件．在多文件结构的程序中，一个程序文件的范围小于整个程序，静态函数的作用域和静态变量的作用域都是其所在的程序文件。对于单文件程序，程序文件也就是整个程序。

函数．在函数内说明的局部变量的作用域在函数体内。

程序块．在任何一个复合语句、循环体（由符号{,}包含的若干程序语句）中说明的局部变量的作用域在该程序块内。

与类和对象有关的作用域问题，将在后文中遇到。

5.6　程序实例

我们学习编程的关键是多写程序，现在我们已经学习了函数的使用方法，通过这种方法我们可以初步地编写一些稍微大型一些的程序。下面我们举两个程序实例，来看一下函数在程序中的作用，特别是学习递归程序的编制方法。

5.6.1 “三色冰激凌”程序

这是一个由冰激淋商提出来的问题，有 28 种颜色的原料，可以组合成多少种三色冰激淋。一种答案是有 19656＝28×27×26 种，称为（28，3）的排列数 A（28，3），它是把同样三种颜色的不同排列数也计算在内了；另一答案是 3276＝19656／3!种，称为（28，3）的组合数 C（28，3）。其中 28 为元素数，3 为选择数，下面的程序对输入的元素数和选择数计算相对应的排列数和组合数。

```
1  //program 5_2.cpp %
2  #include <iostream.h>%
3  long  factorial (int number); %
4  void main (void) %
5  {
6    int  i, selections, elements; %
7    cout << "Number  of  selections:"; %
```

```
8   cin >> selections；%
9   cout << "Out of how many elements："；%
10  cin >> elements；%
11  double answer＝elements；%
12  for (i＝1；i＜selections；i++) %
13    answer *＝ --elements；%
14  cout <<"A("<<elements+selections-1<<","<<selections<<")="；
15  cout << answer << endl；%
16  answer ／＝ factorial（selections）；%
17  cout<<"C("<<elements+selections-1<<","<<selections<<")="；
18  cout＜＜answer＜＜endl；%
19  } %
20  long factorial (int number) %
21  {%
22    long value＝1；%
23    while (number＞1) value *＝ number--；
24    return value；
25  }
```

说明：%

(1) 这个程序看起来不复杂，在 12～13 行计算排列数 A（m，n）＝m（m-1）3 …3 （m-n+1），即从 m 开始计算 m，（m-1），…共 n 个整数的连乘积。而 C（m，n）＝A（m，n）／n!这个计算在 16 行通过函数 factorial（ ）的调用完成。显然输入的整数 n（即 selections）和 m（即 elements）应为正整数且值不宜过大。%

(2) 由于计算中的 A（m，n），C（m，n）和 n！一般是较大的整数，而 int 型数(在 PC 机上)占两个字节，不能超过 2^{15}-1＝32767，所以程序中在数据类型的选择上值得读者注意学习：%

① 变量 i，selections， elements 为 int 型。%

② 在 3,20 行函数 factorial( )的返回类型为 int 型的派生类型 long(即 long int 型)，称为长整型，它占四个字节，因而可以表示更大范围的整数，该函数的参数 number 为 int 型,结果是 number 可能很大,故返回类型为 long。

③ 因同样的原因，在 11 行用来存放计算结果的变量 answer 被说明为 double 类型，称为双精度浮点数，这种类型可以表示数的范围就更大了。%

试想若把函数和变量 answer 说明为 int 型，那么这个程序将很容易因数据

超限而溢出，系统可能出错停机。%

(3) 在 13，23 行的表达式中，使用了 C／C++语言特有的复合运算符“ *= ”和“ -- ”，它们的使用有利于使程序简短，书写方便，不过应准确掌握其性能。%

“ *＝ ”是赋值号“ ＝ ”与乘法运算符“ * ”复合而成，也是一种双目运算：“ a*＝b ”等价于“ a＝a*b ”，也可以把它视为“ a＝a*b ”的简化写法。“ -- ”(和“ ++ ”)称为减量(增量)运算符，属于单目运算，“ a-- ”等价于“ a＝a-1 ”，“ --a ”也等价于“ a＝a-1 ”。“ a-- ”和“ --a ”的区别只有当它与其它运算共同组成表达式时才能显现出来。例如在 13 行是“ 前缀减量 ”运算与“ 乘赋值 ”运算组合成表达式，其运算顺序是：%

首先完成--elements，即 elements＝elements-1 %

然后完成 answer*＝elements，即 answer＝answer*elements %

设执行之前 answer＝20，elements＝10，则执行 answer*＝--elements 之后，首先令 elements＝9，然后 answer＝20*9＝180 %

我们再看 22 行的 value*＝number--，运算顺序为：%

首先完成 value*＝number 即 value＝value*number，%

然后完成 number--，%

设执行前 value＝20，number＝10，则执行 value*＝number--之后，首先 value＝20*10＝200，然后 numer＝10-1＝9，%

由此，就可看出--a 与 a--；++a 与 a++之间的区别了。%

(4) 20～25 行是一个计算正整数阶乘的函数。当实参为小于 1 的整数时，函数返回 1，否则返回 number！。%

这个函数也可以采用递归函数的方式设计，即在函数体内又(直接或间接地)调用该函数本身，函数定义如下：

```
20 long  factorial (int  number) {%
21   if (number＜＝1) return  1；%
22   else  return  number * fectorial (number-1)；%
23 }
```

这个程序与 n！的数学定义式%

$$n!=\begin{cases}1 & n<=1\\ n\times(n-1)! & n>1\end{cases}$$

相吻合。%

递归程序往往使程序思路清晰，易编易读，下节的实例进一步说明掌握递

归程序设计方法的重要性。%

5.4.2 Hanoi 塔问题

Hanoi 塔问题是一个古印度著名的智力测验。该问题是这样的：有三个立柱 A、B、C，A 柱上穿有大小不等的圆盘 64 个，较大的圆盘在下，较小者在上。要求借助于 B 柱把 A 柱上的 64 个圆盘移到 C 柱，规则是：%

(1) 每次只能把一个柱上最上面的圆盘移至另一个柱的最上层。%

(2) 保持在每个柱上的圆盘较大者在下，较小者在上。%

我们给出的程序，是实现对任意的 n 个盘子从 A 柱借助于 B 柱移到 C 柱，并指出全部移动过程。

```
1  //program  5_3.cpp %
2  #include  <iostream.h>
3  void  hanoi(int,  char，char，char)；
4  void  main  (void) %
5  {%
6    int  m；%
7   cout << "Input  the  number  of  disks："；
8   cin >> m；%
9    cout<<endl<<"The  step  to  moving"<<m<<"  disks："；
10   hanoi（m，'A'，'B'，'C')；%
11 } %
12 void  move（char  from,char  to)
13 {
14 cout << endl << from<<"==>"<<to；%
15 } %
16 void  hanoi(int  n，char  a，char  b，char  c)
17 {%
18  if（n==1) %
19     move（a，c)；%
20  else %
21     {%
22        hanoi（n-1，a，c，b)；%
23        move（a，c)；%
```

```
24          hanoi (n-1, b, a, c);%
25        }
26  }
```

说明：%

(1) 第 10 行是本程序的目标：从 A 柱，借助于 B 柱，向 C 柱按规则移动 m 个盘子。%

(2) 16～26 把完成上述任务归结为：%

如果盘子数为 1，只须直接把该盘从 A 柱移至 C 柱；%

如果盘子数大于 1，则完成下面三步：%

① 把 A 柱上面的 n-1 个盘子借助于 C 柱,按规则移到 B 柱(一次递归调用)；%

② 把 A 柱上留下的(最大的)圆盘移到 C 柱；%

③ 把 B 柱上的 n-1 个圆盘，借助于 A 柱，移到 C 柱(又一次递归调用)。

经过此三步，已经完成了把 A 柱上的 n 个圆盘，借助于 B 柱，按规则移到 C 柱的任务。

(3) 函数 hanoi（ ）是一个递归函数，它所解决的 Hanoi 塔问题比上一节中计算阶乘问题要复杂得多。在上一节中我们毫不费力地为阶乘的计算设计了非递归和递归的两个函数，然而对于 Hanoi 塔问题，情况却不相同，其递归程序是相当简单的，但为其设计一个非递归程序却不是一件简单易行的事情。

(4) 程序 program5_3 执行结果如下：

```
Input the number of disks：3 %
The step to moving 3 disks：%
A==>C %
A==>B %
C==>B %
A==>C %
B==>A %
B==>C %
A==>C  %
```

思考题：

1. 什么是函数抽象，程序设计语言中的函数与数学中的函数有何区别？
2. 函数的属性说明有几种？每一种代表的含义是什么？
3. 函数的返回类型有几种？它们的含义和使用方法是什么？

4. 函数的参数调用有几种方式？它们的使用方法和相互的区别是什么？
5. 什么是函数嵌套？
6. 什么是函数递归？使用函数递归有什么好处和不利？
7. 什么是函数的重载？实现函数重载必须满足那几个条件？
8. 函数重载带给我们什么好处？
9. 什么是运算符的重载？各种不同的运算符是怎样进行重载的？

练习题：

1. 仔细分析下面的程序，指出他的输出结果是什么？在计算机上运行程序看结果是否和预测的一致？

```
#include <stdio.h>
add (int x, int y) {
  return x + y;
}
main () {
  int x, i;
  printf ("x1=%d\n", add (4, 8));
  printf ("x2=%d\n", add (3, add (1, 2)));
  x = 0;
  for (i = 1; i<=6; i++)
    x = add (x, i);
  printf ("x3=%d\n", x);
  printf ("x4=%d\n", add(add(add(add(add(1,1),1),1),1),1));
  printf ("x5=%d\n", add(add(add(add(add(1,2),3),4),5),6));
}
```

2. 仔细分析下面的程序，指出程序完成什么功能。

```
int Func (char *x)
{
  char *y = x;
  while (*y++);
  return (y-x-1);
}
```

3. 对给定的字符串“AaBbCcDdEeFf”，将它的前 n 个字符从小到大进行排序，将下面的程序填写完整，并上机进行调试。

```
main () {
  char *str = "AaBbCcDdEeFf";
  sort (str, 8);
  printf ("%s\n",str);
}
sort (_______) {
  int i, j = count;
  __________;
  while (j-->1)
    if (________) {
      temp = str[i];
      str[i] = str[i+1];
      str[i-1] = temp;
    }
}
```

4. 编写一个函数，能将通过参数传来的十进制整数转换成八进制整数。
5. 设 P（x）表示十进制正整数 x 的各位数字之积，编一个程序求出满足

   P（x） = 10x - 22
6. 设计一个函数，可以对 n 个浮点数进行排序，其方法是每次取未排序的元素集中的最小元素，重复 n – 1 次完成 n 元排序。
7. 编写一个函数 Inverse（int n）可以把正整数 n 及其各个位上的数字的逆序输出，例如：n = 2375，输出为 5732。
8. 解释 Hanoi 塔问题程序解答中的函数调用关系，自行设计一种图示方法将这个程序中的函数调用关系清楚地表示出来。
9. 编写函数 CacuN（int n）计算 N !，使用递归方法和非递归方法。

第六章 指针，引用与动态内存分配

在第四章，我们曾引入数组类型，它与本章介绍的指针和引用类型是 C++语言提供的三种导出类型，正确灵活地使用数组，指针和引用类型，是设计高质量的 C++程序所必经掌握的技术。结构在只有数据成员时，也是一种导出数据类型，不过，C++语言允许用户为结构设计函数成员，实际上是用户为该结构类型设计的运算，因此，结构也是用户定义的类或类型，将在下一章介绍。

C++程序中所处理的数据主要以变量（和对象）的形式出现。每个变量有两个最主要的属性，那就是变量的内容和地址。例如：

```
int a=3,b=5,c;
c=a+b;
```

在上面语句中，变量 a,b,c 已被赋值。在它们被重新赋值或被释放之前，变量名 a,b,c 就代表着它们的内容，即整型数 3，5，8。由变量 a,b,c 参加的任何运算，都是对它们的值 3，5，8 进行的运算。

另一方面，C++语言也为用户对于变量所占空间的地址提供了进行操作的机制,这就是指针类型和引用类型。指针和引用不是一种独立的数据类型，而是一种导出数据类型。指针类型也是 C 语言和 Pascal 语言的重要类型，而引用类型是 C++语言中新的重要类型，在 Java 语言和 C#语言等一些新的语言中，引用类型取代了指针类型，因此，两者的使用都应该熟练掌握。

我们首先通过实例来说明指针的使用。

6.1 选择排序算法

排序的任务是把已经存在一个数组里的 n 个数按从大到小的顺序排列，可以采用多种不同的方法实现，选择排序算法的思路简单，容易理解，每次总是从无序的序列中选出最大者，交换到序列的左端，于是，无序的序列越来越短，经过 n-1 步，达到排序的目的。程序 program6_1 的主要部分是介绍实现选择排序算法的函数 void ssort(float *,int)，在阅读下面的程序时，应特别注意指针类型变量和函数参数的使用：

```
//program6_1.cpp
#include<iostream.h>
#include<stdlib.h>
#include<iomanip.h>
void ssort(float *, int );
void main(void){
       int   i, seed, n=20;
       cout<<endl<<"seed="  ;
       cin>>seed;
       cout<<endl;
       srand(seed);
       float    list[20], *pf;
      for(i=0; i<n; i++){
         list[i] = rand( );
         cout<<setw(8)<<list[i];
         if ((i+1)%8==0)
              cout<<endl;}
       i=0; cout<<endl;
       pf=&list[0];
       ssort (pf,n);
      while ( i<n ) {
       cout<<setw(8)<<list[i];
       i++;
       if (i % 8 == 0)
              cout<<endl;
       }
       cout<<endl;
}
void swap1(float *a, float *b){
       float  temp;
       temp = *a;
       *a = *b;
       *b = temp;
}
```

```
void ssort (float *a, int m){
    int   i, j, index;
    float elem;
    for (i=0; i<m-1; i++){
        elem = *(a+i);
        index = i;
        for (j=i+1; j<m; j++)
            if (*(a+j)>elem){
                elem = *(a+j);
                index = j;
            }
        swap1(a+i, a+index);
    }
}
```

运行结果：

```
seed=6523
 21339   28822   26220    8712   18120    1280   27491    8647
  7137    2030   24101   20505    4452   26942   10989   29919
 27130   20996   17227   13559
 29919   28822   27491   27130   26942   26220   24101   21339
 20996   20505   18120   17227   13559   10989    8712    8647
  7137    4452    2030    1280
```

程序说明：

这个程序重点是介绍函数 ssort( ) **和** swap1( ) **的设计，应特别注意指针变量 pf、函数 ssort()的指针类型参数 a 和函数 swap()的指针类型参数 a,b 的说明和使用方法。**

主函数做了三件事：

1) 为长度为 n 的数组 list[n]输入 n 个随机数，作为排序算法处理的对象，程序中取 n=20,是为了显示输出结果方便，也可取 n=1000 或更大，实际输入的值是整数也是为了显示清楚。利用库函数 rand()产生伪随机数形成无序数组，由用户输入一个随机数（例如 6523）作为“seed”可使无序数组的 20 个数的随机性较好。

2）调用函数 ssort()进行排序。

3）把经过排序的数组 list[20]的值输出。

函数 swap1( )的定义中使用了指针类型的参数。参数 a,b 是 float 型的指针变量，即它们的值应是浮点型变量的地址。

在 swap1( ) 的调用语句中，实参 a+i, a+index, 其中 a 为 float 型的指针，i, index 为整型变量，C++ 语言允许指针变量+/-整数，其结果仍为指针。即 a+i, index 应表示数组 List[ ]中 a 指向的元素后面的元素变量的地址。

在函数 swap1( )中使用了*a 和*b，这里符号‘ * ’不是乘法运算符，而是表示指针变量（a 或 b）所指向的那个变量。例如：

```
int n=4, *p;
p=&n;
*p++;
```

表示有整型变量 n,和整型指针变量 p,用&n 表示变量 n 的地址，p=&n 使指针 p 指向变量 n,而这时*p 正是表示变量 n,*p++相当于 n++。

由此可以看出，‘ & ’和‘ * ’是一对用于指针操作的互逆的单目运算符，（在这里‘ & ’不是“ 引用 ”的意思）。&n 表示变量 n 的地址，而*p 表示指针 p 指向的变量或该变量的值。

对于函数swap1(float *a,float *b){…},读者可与上一章中介绍的swap(int &a, int &b){…}函数作一比较，虽然它们都达到交换两个变量内容的目的，但采用的方式却是不同的。在 swap1( )中有两个指针型参数，属于赋值形参，在调用时创建两个新的指针变量 a 和 b，把实参表达式计算的结果地址赋给 a 和 b，从而使指针 a 和 b 指向相应的变量。而在 swap( )中形参 a,b 为引用型，它是在调用时直接由实参的变量代替 a,b 参加 swap( )函数的函数体操作。

函数 ssort( )是一个用于对 m 个浮点数组进行排序的函数，其参数为 float 型指针和 int 型整数。前者指明数组的首地址，后者为数组中元素的个数，执行结果是把这个数组中的浮点数按从大到小的顺序排列好。

按照 C++语言的规定，数组名 list 除了作为名字之外，它还可表示该数组的首地址，即可以用 list 直接表示&list[0]。

因此，19～20 行可以用下面一个调用语句取代：

```
ssort(list,n)
```

其效果完全相同。

函数 ssort(tist,n)的函数体是具体完成排序的过程，请注意如何表示数组的各个元素。

从这里可以更清楚地了解到：程序中说明的数组 list[20],实际上是 n（=20）个 float 型变量，它们除可以用下标变量 list[0],list[1],..,list[19]表示之外，也可以用*list,*(list+1),,*(list+19)来表示。指针可以和整数做加减法，这一点已在上

文指出，指针变量做加法要小心，C/C++的指针操作有些“过于自由”了。

把 n 个数排好次序可以有许多种不同的算法，这个函数是每次把尚未排好的元素中的最大元素选出，移到左端，最终形成从大到小的排列，故被称为选择排序算法。

下节介绍有关指针类型的语法规则。

6.2 指针（Pointer）类型

6.2.1 指针变量说明

指针类型的变量说明格式为：

〈类型名〉*〈指针变量名〉=〈初值〉;

类型名：任一基本类型名，基本类型的派生类型名，用户定义的类、枚举类型、结构及联合类型名。

类型名为 void 时，称为不确定类型的指针类型。

类型名也可以是由〈类型名〉*表示的指针类型名，这时称为多级指针（指针类型的指针）。

符号*：表示其后说明的变量为指针变量（这里的"*"不是运算符）。

指针变量名：标识符。

初值：可缺省。它可以是该类型的某一变量的地址。例如：

```
int *ptr;
float a=3.0, b, c[4];  float * pa=&a;
float *p[2]={&a, &b};
float *pn=NULL;
float *pc=c;
float *pf=new float;
point *pp;
```

其中：

ptr 是一个未赋初值的 int 型的指针变量。

pa 是 float 型的指针变量，被赋初值为&a,即 pa 当前的值为变量 a 的地址，或说 pa 指向变量 a。如把&a 赋给 ptr 是不合法的，因为后者是整型指针。

p[ ]是浮点型指针数组，它由两个指针元素组成，分别被赋值&a，&b。

pn 是一被赋初值为 NULL 的指针变量，规则规定 NULL 与整数 0 通用，它是唯一可以赋给任一类型指针变量的值，表示当前该指针未指向任一变量。因

此，一个指针变量，可有三种状态：

（1） 未赋任何值，“ 悬空 ” 状态。

（2） 被赋予 NULL 值。未指向任一变量。

（3） 指向某一变量。

pc 是一浮点型指针变量，由于 c 是浮点型数组的名，因此 c 又是指向数组的首元的指针，这时把 c 的值赋给 pc,意味着 pc 的值为数组 c[ ]的首元 c[0]的地址。

pf 也是浮点型指针变量，它的初值是通过动态内存分配运算符 new 的“ 运算 ” 结果，生成一无名的 float 型变量，其地址赋给了 pf。new 的使用将在本章介绍。由于这样产生的 float 型变量是无名的，因此在程序中对它的引用，主要是靠 pf（用*pf 表示）。

pp 是一个对象指针，因为 point 是一个已定义的类名，pp 将可用来存放一个 point 类对象的地址，或说可用 pp 指向 point 类的一个对象（关于类和对象在第七章以后介绍）。

6.2.2 指针变量的操作

本节介绍与指针变量有关的运算。

1．取地址运算&和取内容运算*

取地址运算为一单目运算，运算符为&,例如：

```
int a=3,*pa=&a;
```

其中&a 表示变量 a 的地址。它可以作为指针 pa 的值。使用&时应注意：

（1）其后只可用一变量名，而不可为一字面常量或一般表达式。

（2）赋值时，pa 与 a 的类型应一致。

取内容运算为一单目运算，运算符为*，例如：

```
int a=3,*pa=&a;
*pa=5;
cout<<a<<*pa;
```

其中*pa＝5 和 cout<<a<<*pa; 中的*pa 表示指针 pa 当前所指向的变量 a, 因此，*pa=5;相当于 a=5; ,这里的*pa 被称为“ 左值 “ ，相当于变量 a 占用的空间，而后一个*pa 则相当于变量 a 的值，故二者显示出的值应都是 5。

运算*与&互为逆运算，这两个运算符都有不只一种含义，应注意不要混淆。

2．数组指针的算术运算

如果指针变量是指向数组的，例如：

```
int  a[20]；
int  *pa=a；
```

则指针 pa 可以有限制地进行＋／－运算或增量（＋＋）减量（－－）运算。

这时，可以用三种方式表示该数组的元素：

（1）下标变量的形式：

a[0], a[1], a[2], a[3], …, a[19]；

（2）用数组名 a（一个常量指针）：

*a, *(a+1), *(a+2), *(a+3), … ，*(a+19);

（3）用指针变量 pa（＝a）：

pa，(pa+1), *(pa+2), *(pa+3),…，*(pa+19)；

虽然三种方式都可以，不过应注意的是，这里 pa 是变量，它可能是变化的，例如：

```
pa+=3；
```

执行后，pa 已是指向 a[3]，即*pa 为 a[3]。

```
pa--；
```

执行后，pa 已是指向 a[2]。

下面两个程序效果是一样的：

```
#include <iostream.h>
    void  main(){
    int i,a[5];
     cout<<"input 5 integers:"<<endl;
     for(i=0; i<5; i++)
          cin>>a[i];
     cout<<"==== Reverse output: ===="<<endl;
     for(i=4; i>=0; i--)
          cout<<a[i]<<"    ";
     cout<<endl;
    }
#include <iostream.h>
    void  main(){
     int i,a[5];
     cout<<"input 5 integers:"<<endl;
     for(i=0; i<5; i++)
          cin>>*(a+i);
```

```
        //*(a+i)全同于 a[i] , a 为数组首地址(一个常量指针)
    cout<<"==== Reverse output: ===="<<endl;
    for(i=4; i>=0; i--)
        cout<<*(a+i)<<"   ";
        //*(a+i)全同于 a[i]
    cout<<endl;
    }
```

当数据为多维数组时，上述对应关系为：

```
int   a[2][3]；
int   *  pa=a；      // 或*pa=a[0][0];
```

三种方式表示其六个元素：

（1）a[0][0]，a[0][1]，a[0][2]，a[1][0]，a[1][1]，a[1][2]。

（2）*a，*(a＋1)，*(a＋2)，*(a＋3)，*(a＋4)，*(a＋5)。

（3）*pa，*(pa+1)，*(pa+2)，*(pa+3)，*(pa+4)，*(pa+5)。

其中

*（a＋4）即为 a[1][1]，

*（pa＋2）即为 a[0][2]，

应注意：

（1）随时检查当前指针 pa 指向数组的哪个元素。例如 pa=pa＋4；执行后，pa 指向了 a[1][1]。这时如果再做：pa＋＋；pa＋＋；就会出现指针 pa“超界”，已不知它该指向谁了。

（2）有时可以把数组指针在限定范围（数组元素数）内，当作整数来运算，例如：

```
int   a[4][5]，n；
int  *pa=a，*pb＝&a[3][4]；
n=pb-pa+1；
cout＜＜n＜＜endl;
```

输出的是数组的元素总数。

C++的数组指针的+/-运算，使用起来比较方便，但很不安全，极容易出错，必须予以注意。有的人把指针称为“数据结构中的 goto 语句”意思就是它的过于灵活容易造成程序出错，这也是一些新的语言，如 JAVA,C#语言中取消指针类型的原因之一。。

3．指针的关系运算

指针变量可以参加关系运算。这要分三种情况：

（1）一般指针可以进行相等和不等的比较，指向同一变量（地址）者为相等，否则不等。

（2）任一指针可以和指针常量 NULL 进行相等和不等的比较。如一指针 p 已经指向了某变量，则它不等于 NULL。

（3）数组指针，可以指向该数组的各个元素，且一个数组的各个元素在内存中是顺序存放的，故数组指针之间不但可以进行相等和不等比较，也可以进行大于，小于，大于等于，小于等于等的比较。

下面是有关的例子：

```
float   a，b，c[10]；
float   *pl=&a，*p2=NULL，*p3=C，*p4；
if（pl != p2）p2=p3＋2；
if（p2＞p3）p2＋=1；
```

顺便说明，指针和所有类型一样，它也可进行赋值运算。

6.2.3 指针与数组

指针与数组都是导出类型，它们都是在其它类型的基础上被定义的。那么它们互相之间可否“导出”呢？

1．指向数组元素的指针

前面被简单地称为数组指针的，实际上应更确切地称之为指向数组元素的指针，例如：

```
int   n［10］；
int   *pn=n;
```

则指针 pn 指向数组 n 的首元 n[0]。在这里

```
pn=n
```

与

```
pn=&n[0]；
```

起相同的效果，因为数组名就是一个指向其首元的一个指针常量。

2．指向数组的指针

把数组作为整体，称为指向数组的指针，对于这类指针，其说明语句格式为：

〈类型名〉（*〈指针变量名〉）[〈数组元素数〉]；

而指向多维数组的指针说明中，最后一项[〈数组元素数〉]可有多个。

可以发现，指向多维（或一维）数组的指针说明与多维（或一维）数组说

明只相差符号*和一对括号（*）。这个括号是很重要的，在下文介绍的指针数组恰恰没有这对括号。例如：

```
int （*pa）[4]；
```

表示指针变量 pa 是一个指向一维整型数组的指针，该数组包含四个整型元素。

一般指向数组的指针由多维数组的操作中被赋值：

```
float A[2][4]；
float（*pa）[4]；
pa=A；
```

这时指针 pa 指向一维数组 A[0]，（A[0]有四个元素），执行增量运算

```
pa++；
```

后，pa 指向一维数组 A[1]，数组 A[1]的四个元素是 A[1][0]，A[1][1]，A[1][2]，A[1][3]。

它们也可表示为（*pa）[0]，（*pa）[1]，（*pa）[2]，（*pa）[3]。

从指向数组指针和指向数组元素指针的区别，可以看到：虽然指针变量的内容都是地址，但指针变量的类型（指向的对象类型）则决定了该地址指向的存储区的大小，例如：

```
float *P=&A[0][0]；
float（*pa）[4]；
pa=A；
```

虽然这时指针 p 和指针 pa 指向的地址是一样的，但 P 是指向一个整型下标变量 A[0][0]，而 pa 则是指向整型一维数组 A[0]，前者占内存空间可能是 4bytes，而后者则占 16bytes。

3．指针数组

由指针组成的数组，常常在程序中出现，其说明格式为：

〈类型名〉*〈数组名〉[〈元素数〉]

n 维指针数组为：

〈类型名〉*〈数组名〉[〈元素数＞][〈元素数 2〉]……[〈元素数 n〉]；

从上面格式可知：

（1）它比一般数组增加了符号‘*’。

（2）它比指向数组的指针，少了一对括号“()”。例如：

```
int a，b，c，d，A[2][4]；
int *p1[4]={&a,&b,&c,&d};
int *p2[2]={A[0],A[1]};
```

其中，指针数组 pl 由四个整型指针组成，把四个整型变量 a，b，c，d 的

地址赋给它们作初值是合法的。

指针数组 p2 由两个整型指针组成，A[0]，A[1]是两个一维整型（含四个元素的）数组名，它们表示 A[0][0]，A[1][0]的地址，作为 p2 的元素的初值也是合法的。

从上面三个小节的内容可以看出下面定义的指针的区别：

```
int *p;
int *p1[2]；
int（*p2）[4]；
```

p 是指向 int 型变量的指针，p1 是由两个整型指针组成的指针数组，而 p2 则是一个指向一维整型数组的指针。

6.2.4 字符串指针

字符串是一种特殊的数据形式，在数组一节，我们已经指出字符串和字符数组的关系。C++语言对于字符串的操作提供了多种手段，其中利用字符串指针是最方便的一种。

1. 说明与初始化

字符串指针没有自己独立的形式，其说明语句的格式与字符类型指针相同，例如：

```
char  ch= 'a'：
char  *pc1=&ch；
char  *pc2="world"；
char  as[10]="Chinese"；
char  *pc3=as；
```

在上面的几个说明语句中：

变量 ch 是字符型，它被赋初值字符‘ a ’。

变量 pcl 是字符指针，它被赋初值为字符变量的地址。这时 pcl 是一个字符型指针。

变量 pc2 和 pcl 一样说明为字符指针，但它被赋初值为字符串常量“ world ”，于是 pc2 就成为一个字符串指针。它的内容是字符串第一个字符‘ w ’的存储地址。

数组 as[ ]是一个长度为 10 的字符数组。为 as 所赋的初值为字符串常量“ Chinese ”，这个常量在数组中占八个分量的位置，七个字母和一个串尾符‘ ＼ 0 ’。

变量 pc3 说明为字符指针，并令其指向字符数组 as[]的首元，这时，pc3 也是一个字符串指针。

字符串指针可以按一般字符指针的规则工作，例如：

```
while（*pc2 != '\0'）
cout＜＜*pc2++；
```

执行上述 while 语句将在屏幕上显示 world。类似地：

```
while（*pc3 != '\0'）
cout＜＜*pc3++；
```

执行的结果是显示 Chinese。

字符串指针有时可以按不同于一般指针变量的方式操作，即有时它可以表示字符串整体，而不是单单表示当前指向的那个字符。这一点为用户提供了方便，但也应在概念上搞清楚，以免混淆,请看下面介绍。

2．字符串的整体输入输出

把字符串作为字符数组,然后按字符依次进行输入输出是可行的(如上述)，但比较麻烦。因此，再提供整体输入输出的方法。

把字符串变量用字符数组或字符串指针的形式表示，然后直接进行 I/O 操作：

```
cin＞＞〈字符串〉；
cout＜＜〈字符串〉；
```

这里的字符串可以用字符串指针 pc2，pc3 或字符数组名 as 来表示，例如：

```
char   as  [10]="world"；
char   *pc2＝"world"；
char   *pc3=as；
cout＜＜as；
cout＜＜pc2；
cout＜＜pc3；
cout＜＜"world"；
```

后四个输出语句产生相同效果，都是输出了字符串 world。

```
char   str  [20]；
char   *pc2=new  char[20]；
char   *pc3=str；
cin＞＞str；
cin＞＞pc2；
cin＞＞pc3；
```

后三个输入语句允许输入长度不超过 20 的字符串，忽略前导空格，以空格作为串尾。

3. 字符串指针数组

以字符串指针组成的数组使用起来比较方便，例如：

```
char *menu［］＝{"File"，"Edit"，"Search"，"Help"}；
```

就是一个字符串指针数组，它的每个元素指向一个字符串常量。于是 menu[0]，menu[1]，menu[2]，menu[3]将分别表示“File”,“Edit”,“Search”,“He1p”。

4．利用字符串指针的标准函数

在头文件 string.h 中，C＋＋系统提供若干串处理函数的说明，它们大都利用字符串指针，下面列出几例：

（1）求字符串长度。

```
unsigned strlen（const char *str ）；
```

返回字符串 str 的长度（不包括串尾符）。

（2）字符串拷贝：

```
char *strcpy（char *str1 ，char *str2）；
```

把字符串 str2 拷贝给字符串 strl（覆盖原来内容），返回 strl 的地址。

（3）字符串连接：

```
char *strcat（char *str1 ，char *str2）；
```

把字符串 str2 连接到字符串 str1 的后面（使 str1 加长），返回 str1 的地址。

（4）字符串比较：

```
int strcmp（char *str1 ，char *str2）；
```

此函数把 ASCII 码表作为字符表，按字典序比较两个字符串 str1 和 str2。例如串“abc”小于串“abd”，串“xy”小于串“xyz”等等。根据两个串之间的大于，等于，小于关系，返回正整数，0 和负整数。

在 C++程序中，有关指针、字符数组和字符串的操作是初学者较难掌握的，下面的程序实例可以帮助读者加深理解。

```
#include <iostream.h>
void main()
{
    char s[21], *ps=s;
    for (int i=0; i<20; i++)
        s[i]='A'+i;
```

```
        s[20]='\0';
        ps++;
        cout<<"ps="<<ps<<endl;
        ps+=2;
        cout<<"ps="<<ps<<endl;
        for (ps=&s[19]; ps>&s[11]; ps-=2)
        {
            cout<<"*ps="<<*ps<<endl;
            cout<<"ps="<<ps<<endl;
        }
}
```

运行结果：

```
ps=BCDEFGHIJKLMNOPQRST
ps=DEFGHIJKLMNOPQRST
*ps=T
ps=T
*ps=R
ps=RST
*ps=P
ps=PQRST
*ps=N
ps=NOPQRST
```

6.2.5 指针与函数

指针与函数的关系密切，主要分三部分，指针作函数参数，作函数返回类型和指向函数的指针。函数指针在 Pascal 语言是没有的。

1．指针作函数参数

指针是一种特殊类型的数据，它的值是另外某种类型的变量的地址。因此，指针作函数的参数可以起到其它类型参数所起不到的作用。

C＋＋语言中，函数的参数（引用型参数除外）在调用过程中，须把实参（表达式）的值赋给仅在调用过程中有意义的形参，参加到函数体的运算之中。这样的机制很难实现对相对于函数的外部的“ 全局 ”变量作某些处理。例如要

编写一个交换两个浮点型变量内容（值）的函数：

```
float  a=1.0, b=2.1，c=3.2，d=4.5;
```

为了交换变量 a，b 的值，当然可以直接用三条语句完成：

```
float  t＝a;
a=b;
b=t;
```

就可达到目的。若要以函数形式表示，该函数应为无参函数：

```
void  swap（void）{
float  t=a;
a=b;
b＝t;
};
```

这个函数当然是对的。其中变量 a,b 是在函数外定义的全局变量。它的明显缺点是没有“参数化”，没有通用性。当我们需要交换 b 和 c，或交换 c 和 d 的值时，它就不能用，但是把它写成有参的形式：

```
void swap（float x，float y）{
float t＝x;
x=y;
y=t;
};
```

这个函数就是错的，当要把变量 a,b 内容交换时，函数调用：

```
swap（a，b）；
```

不会把变量 a 和 b 的内容真的交换，它只是在函数执行时把临时的形参 x 和 y 的内容交换了，而这个 x 和 y 在函数执行完成就被释放掉。

利用指针可以解决上面的问题，swap（ ）函数可以定义为：

```
void  swap（float *pa，float *pb）{
float  t=*pa；
*pa=*pb；
*pb=t；
};
```

为了交换 a 和 b，交换 c 和 d,只需调用二次：

```
swap（&a，&b）；
swap（&c，&d）；
```

因为它通过参数传进了要改变内容的全局变量的地址，从而可以达到以参

数形式输入，又能改变函数外变量内容的目的。

2．函数返回指针

返回值为指针的函数称为指针型函数，其返回类型的说明应指明指针的对象类型后加“*”符号。例如：

```
char *menu[ ]={"Error!","File","Edit","Search","Help"};
char *menuitem（int m）{
return（m<1 || m>4）？menu[0]：menu[m]；
}；
```

这个函数 menuitem()返回的是字符串指针，可直接调用函数 menuitem（ ）来进行字符串的输入输出。

指针型函数的设计，应注意返回的指针在该函数被调用的域内是有确切的对象变量的。切不可返回指向函数内说明的局部变量或参数变量的指针，或无指向的指针。

3．函数指针

函数不是数据，但它与变量还是有两点相通之处：一个是它有类型（返回类型），另一个是它也有地址，称为人口地址，故在有的书中也勉强地把函数仍归为一种特殊的数据“类型”。

函数的地址也可作指针的值，这就是函数指针。

函数指针的说明格式与函数的原型相似，主要区别是：原来的〈函数名〉，用 *〈函数指针名〉所代替，例如：

```
int（*pf）（float）；
```

其中 pf 是一个函数指针变量，由于对 pf 的说明中已规定了函数的返回类型（有时还包括存储类型）和参数表，因此，指针 pf 只能够指向这类函数。例如：

```
int f1（float）；
int f2（char）；
int f3（float）；
int f4（float）；
```

设 f1,f2,f3,f4 是 4 个已说明的函数，这时，下面的说明和赋值，就有合法与不合法的区别：

```
int（*pf）（float）=&f1；          //合法
int（*pf1）（char）=&f1;           //不合法
pf=&f4；                           //合法
pf=&f2；                           //不合法
```

```
pf=&f3；                                    //合法
```

C＋＋语言本身不允许把函数作为参数，然而有了函数指针，就可以通过函数指针，起到把函数作为参数的作用。 例如：

用来计算函数定积分的函数 simpson（），对于不同的函数计算其定积分值，应该有一个函数参数，在 C＋＋程序中用函数指针可以方便地解决这个问题：

```
float simpson（float a，float b，float（*pf）（float））；
```

参数 a,b 给出定积分的上下限，函数指针 pf 则指向被积函数（其函数体从略）。在使用时可以对不同的浮点函数和上下限，调用 simpson（）计算其定积分：

```
float  a=3.2，b=4.2;
float  f1（float）{…}；                    //函数体从略
float  f2（float）{…}；                    //函数体从略
float（*pfl）（float）=&f1：               //函数指针 pf1 指向函数 f1
cout＜＜simpson（a，b，pfl）＜＜endl   ；
a＝4.0；b=10.5；
pfl=&f2；                                  //pf1 指向函数 f2
cout＜＜simpson（a，b，pfl）＜＜endl；
```

两次调用分别计算两个不同函数 fl 和 f2 的定积分值，这样的功能没有函数指针是难以实现的。

函数指针的使用属于较深入的话题，这里就不进一步展开了。

6.3 指针与动态内存分配

6.3.1 动态分配运算符

在第四章，我们已经介绍了动态分配运算符 new 和 delete。new 和 delete 的使用，也是 C++语言优于 C 语言的特征之一。

New 运算是程序中除了变量说明之外，又一种生成变量的方法，不过用 new 生成的变量为无名动态变量，它返回的是一个该类型的指针值，程序通过指针对这个变量进行操作。例如：

```
int   *pi，a=5;
char   *pc；
float   *pf，f=3.0；
```

```
pi=new  int；                          //生成一动态 int 类型变量
pc=new  char[4]；                      //生成动态 char 类型数组
pf=new  float（4.7）；                 //生成动态 float 类型变量且赋初值
*pi＝a*a;                              //动态变量*pi 被赋值
for（int i=0；i＜4；i＋＋）            //动态 char 型数组的每个分量被赋值
    *（pc＋i）＝‘a’；
f+=*pf；                               //动态变量*pf 参加运算
```

我们可以把用 new 生成的一个动态变量与用变量说明语句说明的一个变量作比较：

·都要指出变量的类型，类型名要放在 new 之后。

·都可以赋初值，不是用=〈初值〉的方式而是用括号（〈初值〉）的方式。

·都可以说明为数组，加数组运算符[ ]。

·动态变量没有变量名，须用指针变量接收到它的地址后，通过指针运算符‘*’进行操作。

Delete 运算用来撤消或释放由 neW 生成的动态变量，例如：

```
delete   pa；          //释放 pa 指向的动态 int 变量
delete   pf；          //释放 pf 指向的动态 float 变量
delete［］pc；         //释放 pc 指向的动态数组
```

动态变量与一般变量的主要区别就是它可以在程序运行过程中任意被撤销。而一般变量则必须在其所说明的程序块结束时自动撤销。

6.3.2 用指针进行内存动态分配

C＋＋程序中对变量、数组等的说明实际上就是为它们进行内存分配，所分配的空间及其内容，通过变量名引用。用这种方式为变量分配空间，其占用形式比较死板。有了运算符 new 和 delete,就可以实现一种动态分配内存的形式，即通过指针引用，而内存的分配和释放可以在程序的任何地方进行。C 语言中没有运算符 new 和 delete,其动态内存分配依靠若干库函数完成，比较复杂，这也是 C++语言优于 C 语言的特点之一。例如：

```
int    *pint;
char    *pchar;
float   *pfloat;
pfloat = new float;
```

```
pchar = new char;
pint = new int;
```

这样就生成了三个（浮点型、字符型、整型）变量，但它们没有名字。由于三个变量的地址分别存在指针 pfloat，pchar 和 pint，故在程序中使用这三个变量时，全通过指针：

```
*pchar = 'A';
*pint = 5;
*pfloat = 4.7;
*pint++;
```

当不再需要这些变量时，可在程序的任何地点释放掉它们：

```
delete  pchar;
delete  pint;
delete  pfloat;
```

注意，这里释放的是动态的（char,int,float）变量，而不是指针变量（pchar, pint, pfloat）。

这种动态生成的变量，在使用上方便灵活，其必要条件就是应用相应的指针来操纵它。

读者在使用动态变量时应注意的是，要保护动态变量的地址。例如在执行

```
pi=new int;
```

之后，不要轻易地冲掉指针 pi 中的值，假如执行了 pi=&a；语句之后，再释放原来生成的动态变量：

```
delete pi;
```

已经达不到原来的目标了。

C＋＋中 new 运算类似于 Pascal 语言中 new 过程的使用，不过它不作为相应的函数处理。C＋＋还保留了从 C 语言中继承下来用于动态存储分配的 malloc（），free（ ）等标准函数，它们已经可以为 new 和 delete 运算所取代。

6.4 导出数据类型（3），引用（Reference）

引用（reference）是 C＋＋语言特有的（C 和 Pascal 语言不具备）数据形式。它的存在不仅像数组和指针那样依赖于已有的类型，而且它还依赖于一个已有的变量。从直观上说，一个引用变量是一个已定义的变量的别名。这种引

用型变量在作为函数的参数和返回类型时，其使用方式与指针类型有相似性。上文已经提到指针类型的缺点，因此建议读者在学习和掌握C＋＋语言过程中，应体会引用类型的重要性，理解它的原理和特征，掌握它的使用方法，在既可以使用指针类型又可以使用引用类型时，多用引用类型。由于一些新的语言，如 JAVA,C#等，取消了指针类型而仅保留引用类型，引用类型的重要性就更容易理解了。

1.引用变量的说明

引用变量的说明格式与指针变量说明相似：

〈类型名〉＆〈变量名〉=〈对象变量名〉；

与指针说明的区别是：

（1）用符号‘＆’代替符号‘＊’。

（2）赋初值部分不可缺省。

例如：

```
int size=5，color；
int ＆refs=size；
int ＆refc=color；
```

引用变量 refs，refc 相当于整型变量 size 和 co1or 的别名，即：

```
refc＝3;
```

其效果为变量 color 和 refc 同时被赋值为 3。

```
refs+=2;
```

其效果是变量 size 和 refs 同时在原来值 5 的基础上加 2，即改变为 7。

2．引用和指针的比较

C＋＋语言中增加引用这种数据形式，主要是用于在一定的范围内，代替或改进指针的作用。虽然在说明方式上十分相似，但在概念上却有着明显的不同。其区别最主要有两点：

（1）指针表示的是一个对象变量的地址，而引用则表示一个对象变量的别名。因此在程序中表示其对象变量时，前者要通过取内容运算符*，而后者可直接代表：

```
int   m，n;
int   *p =  ＆m；
int   ＆r = m；
```

而在对 m 赋值时，下面三者是等价的：

```
m = 5;
*p = 5；
```

```
r = 5;
```

引用类型变量与其他类型变量不同，它没有自己的值和地址空间，只是作为另一变量的别名，在它的生存期期间两个名字绑定在一起，因此，引用类型的使用是有限制的：

引用类型变量不能被引用；

引用类型不能组成数组；

引用类型不能定义指针。

正是这些限制，保证了它的安全性，反而成为人们选择它取代方便灵活的指针的原因。

(2) 指针是可变的，它可以指向变量 m，也可以指向变量 n，而引用变量只能在定义时一次确定，不可改变。

```
int  m，n；
int * p =  &m；
int  &r = m：
```

以后执行

```
p =  &n；
```

是合法的，而引用变量 r 只可与 m 相联。

因此，有人说可以把引用看成指针常量，即：

```
const int * p =  &m；
int  &  r = m；
```

二者更相近。不过这里 p 对应的是 m 的地址，而引用变量 r 不关心 m 的地址，它直接与变量 m 本身相关。

3．引用型参数

把函数的参数说明为某一类型（或类）的引用类型，意味着这个参数为“变量参数”，称为引用调用。

引用型参数在函数被调用时，相应的实参必须是对应类型的变量或对象；在调用函数体运行前，生成该实参的引用变量；在整个函数体运行过程中，这个引用变量相当于作为实参的变量或对象的别名，直到函数调用结束返回。

引用型参数的优点是：

(1) 它可以把函数外的变量以别名的形式引入到函数体内参加运算，非常方便，这种方式比用指针解决这个问题更合理。

(2) 它不必在调用时创建与实参变量或对象对应的值参数变量，当实参变量或对象占用内存较多时，这可以节省内存。

(3) 用指针也可以实现类似于引用调用的效果，但由于指针可以改变内容，

任意赋值，因此它不如引用型参数安全。

关于引用调用，在后面的章节有更多的实例介绍。

4．引用型的函数返回值

一般的函数要返回一个值，例如：

```
int  max（int a，int b）{return（a＞b）？a : b；};
```

这个函数返回二者之中较大的值，可令：

```
int c＝max（3，5）；
```

当把函数的返回类型说明为引用型时，这个函数返回的不仅仅是某一变量或对象的值，而且返回了它的“别名”，——该函数的调用也可以被赋值。让我们举例说明：

```
int  a = 2，b = 3，c = 4；
int * p ＝ &a；
int & r = b;
```

下面的一些表达式：

```
c，a＋b，*p，r＋c*b，r
```

其中 a＋b，r＋c*b 是表达式，它们仅可计算出一个值。另外 c，*p，r 也是表达式，而且它们又可被赋值：c=5，*p=6，r=7，等等，因此这类表达式被称为“左值表达式”。

引用型返回类型的函数调用就是一种左值表达式，可以作为值，也可被赋值：

```
int a=3，b=5，c;
int & maxr（int & m，int & n）{
if（m>n）return m;
return n;
}
```

函数 maxr（ ）可以有两个用法：

```
c = maxr（a，b）；
```

它把变量 a，b 中较大者的值赋给了变量 c，返回的是较大者的值。

```
maxr（a，b）=10；
```

它把变量 a，b 中较大者的值改变为 10，返回的是较大者本身。

```
maxr（a，b）++；
```

它把变量 a,b 中当前较大者的值加 1，返回的也是较大者本身。

函数的调用本身也可以作为变量和对象来使用，这是引用概念在 C＋＋程序设计中非常重要的应用，它所提供的编程灵活性，在运算符的重载，在 C＋

＋语言提供的 I/O 系统中有十分重要的体现。

在有些场合下，以引用代替指针，除了可以实现相同的目标外，同时增加程序的安全性，引用型函数可以作为“左值”，它的这种功能可以说是程序设计语言的一个重要发展，C＋＋程序员在程序设计中应充分使用引用概念，必然有利于提高程序的水平。

6.5 程序实例

6.5.1 按人名字典序排列电话簿

本节的程序涉及应用编程经常碰到的字符串和文本的处理，已知有 n 个人的姓名，以及每个人有一个电话号码，然后按这些人名的字典序排列且输出其电话号码表。

```
//program6_2.cpp
# include <iostream.h>
# include <iomanip.h>
void swap (char* & ,char* & );
int compword (char*,char*);
void main(void){
    const int n=5,t=8;
    int i,j,index;
   char  *tele[n],*sele;
   char  *name[n]={"Zhaolin","Mazhigang","Liguoping",
                          "Sunyingmin","Mazilan"};
   for (i=0; i<n; i++){
       cout<<"Input"<<name[i]<<"\'s telephone number:" ;
       tele[i]=new(char[t]);
       cin>>tele[i];
       cout<<endl;
       }
  cout<<setw(15)<<"NAME"<<setw(15)<<"TELE NO" ;
  for  (i=0;i<n-1;i++){
       sele=name[i];
```

```
        index=i;
        for (j=i+1;j<n;j++)
          if (compword(name[j],sele)){
             sele=name[j];
             index=j;
           }
        swap (name[index], name[i]);
        swap(tele[index], tele[i]);
        }
        for(i=0;i<n;i++){
           cout<<endl<<"              "<<setw(14);
           cout<<setiosflags(ios::left);
           cout<<name[i]<<tele[i];
       }
}
void swap(char* & a,char* & b){
   char* temp;
  temp = a;
  a = b;
  b = temp;
}
int compword (char * a, char * b){
  while (*a != '\0' && *b != '\0')
     if (*a != *b)
         return(*a<*b);
      else{a++;b++;}；
   return(*a=='\0');
}
```

此程序运行将在屏幕显示：

Input Zhaolin’s telephone number:24173721
Input Mazhigang’s telephone number:25724717
Input Liguoping’s telephone number:29144415
Input Sunyingmin’s telephone number:21219789
Input Mazilan’s telephone number:29722773

NAME	TELE NO
Ligoping	29144415
Mazhigang	25724717
Mazilan	29722773
Sunyingmin	21219789
Zhaolin	24173721

此程序体现了 C++语言为字符串处理提供的方便。

在上面的程序中，虽然人名和电话号码都是字符串，但前者是作为字符串来处理的。name[i]表示一个字符型指针，它指向的是一个字符串(人名)，而电话号码则是作为字符数组处理的，tele[i]表示一个长度 t=8 的一维字符数组。

字符串也可以直接由一个字符型指针来表示，方法是赋给它一个字符串常量：用双引号括起来的字符串（用单引号括起来的是单个字符常量)，第 10 行说明了由 n 个字符型指针变量组成的数组，同时用 n 个名字字符串常量作为初值赋给它们。这时每个指针，例如 name[0]实际上是指向了一个字符数组，该字符数组长度为 8，其字符内容为：'Z'，'h'，'a'，'o'，'l'，'i'，'n'和'＼0'，最后一个是尾符。

二维数组 tele[n]可以视为由 n 个一维数组组成的数组，因此，tele[i]是其中第 i＋1 个长度为 t 的一维数组的名，或其首地址，tele[i]实为一个指针常量。

指针 name[i]直接用来输出字符串。

指针 tele[i]直接用来输入一串字符，这正是 C++语言为字符串处理提供的方便，不然的话，应按下面的方式处理：

```
char  *pc=name[i];
While(*pc!='\0')cout<<pc++;
```

可代替 cout<<name[i];

```
for(j=0;j<t;j++)cin>>tele[i][j];
```

可代替 cin>>tele[i]

19～29 行使人名表按字典序排列，其方法与上一节的排序原理相同，对于 n 个量进行排序有许多更好的算法，我们选择的“选择排序算法”虽然效率不高，但程序简单易懂。

函数 compword( )是一个按字典序比较两个字符串大小的函数，令两个串自左至右比较，发现有第一个不等字符，即按该对字符决定两个串的先后，如全部相等，则短者为先。

注意其中两个 return 语句中的表达式(*a＜*b＝和(*a=='\0')为关系表达式，

其值依表达式为真或假分别取值 true(1)或 false(0)。即如有不等的对应字符则如*a＜*b,返回 1，否则返回 0。

例如“ Mazhigang ”和“ Mazilan ”比到第 4 个字符，分别为‘ h ’和‘ i ’，这时因‘ h ’＜‘ i ’，故说明：“ Mazhigang ”＜“ Mazilan ”,即前者应排在前。

函数 swap()采用了指针类型的引用参数。

可以有许多不同的算法实现同样的任务，读者不妨用自己的方法编程，比较其优劣，加深对语言功能的理解。

6.5.2 构建人员档案链表

为了管理人员（职员，学生，居民等）或物资、文献的档案资料，用链表的形式很方便。一般采用数组要事先确定数组的项数，有时在编程时无法估计数量的大小，如果硬设定一个值，此值过大造成浪费空间，值小了又会发生溢出。链表的形式就非常灵活，它可长可短，其功能主要是通过指针、动态创建和撤消数据对象的运算符 new 和 delete 来完成的。

```
//program6_3.cpp
#include<iostream.h>
void  main(void)
{
      int  i;
     struct  person  {
          char  name[12];
          int  age;
          char  sex;
         person  *  next;
      };
     person  *  head, * tail, * temp;
     temp  =  new  person;
     head  =  temp;
     tail  =  head;
      for (i=1; ;i++){
          cout<<endl<<i<<" "<<"name:";
          cin>>temp->name;
```

```
            if (temp->name[0] != '*'){
                cin>>temp->age>>temp->sex;
                temp->next = NULL;
                tail = temp;
            }
            else{
                delete temp;
                tail->next = NULL;
                break;
            }
            temp->next = new person;
            temp = temp->next;
        }
        temp = head;
        while (temp!=NULL){
        cout<<endl<<temp->name<<" "<<temp->age;
        cout<<" "<<temp->sex;
        temp = temp->next;
    }
}
```

下面就这个程序作几点说明：

(1) 7～12 **行说明了一种用户定义的结构数据类型，一个** struct **类型可以由若干个不同类型的成员组合而成，使用结构以后就可以更为灵活的描述大千世界，有关** struct **的具体的语法说明和使用方法可以参看第四、七章。在课后的习题中有相关内容的习题，有兴趣的读者可以作一作。本例中的** stuct **类型名为** person,**它由四个数值成员：字符数组** name[12],**整数** age,**字符型数据** sex **和** person **类型的指针** next **组成。**

结构类型 person **的一个变量，由上述四个成员变量组成，例如，可说明：** person a; **则** a **有四个分量**: a.name[12], a.age, a.sex, a.next**。不过在本程序中未说明** person **类型的变量,而是利用动态内存分配运算符** new **生成的动态结构变量，它们由结构类型指针指向。。**

(2) 第 12 **行说明了三个** person **类型的指针变量** temp, head, tail, **它们指向的变量是** person **类型的，其分量可有两种方法表示：**

(*temp).name[2]**或** temp->name[2]

(*temp).age 或 temp->age

(*temp).next 或 temp->next

在程序中采用了后一种表示方法。

⑶ 16～31 行通过一个 for 语句的运行完成人员档案资料表的输入。此程序的特点是，程序设计者不规定输入数据的数量，可以只输入一二项，也可以输入几百、几千项。

for 语句中控制变量 i 不被用来决定循环是否中止，没有常见的“i＜n”“i＜=n”出现。

该循环的出口在第 28 行的 break 语句，当录入员完成了全部数据的输入之后，他只需键入字符‘*’——作为人名，即可令程序停止输入，转到 32 行开始输出。

⑷ 18，20 行完成一项数据的输入，而 next 域的内容由程序自动填入。

⑸ 假设利用本程序输入三个人员的数据后结束，除打印出三人的情况之外，系统中保留的数据形式，可由图 6.1 示意。

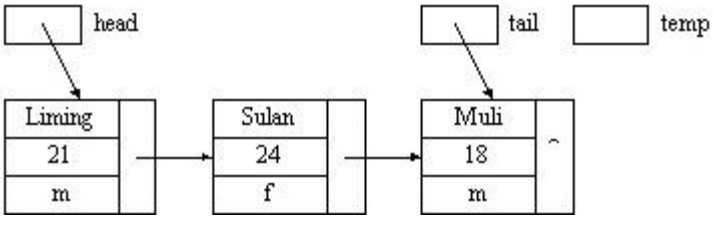


图 6.1　系统中保留的数据形式

⑹ 在 13，28 行使用了动态创建数据对象的运算符 new,其使用的形式是：

〈指针变量〉= new〈类型名〉

其具体功能是：创建一个该类型的变量（这个变量本身没有名字），然后把这个变量的地址送给该类型的指针变量。

temp, temp->next 都是 person 类型的指针变量，执行完 13 行后，*temp 就是刚创建的一个 person 型变量，其效果与说明语句：

person per;

是一样的，不过用 new 生成的变量可在程序执行中创建，又可在任何时候撤消，故这类数据形式称为动态变量。程序中没有利用运算符 delete 释放这些动态结构变量，而是在程序结束时一同释放。，

采用此法，录入员想录入多少项数据，程序就生成多少个 person 变量，没

有数组的长度限制。例如，录入了三项，程序实际上是用 new 运算符创建了四个 person 型数据块，最后一个因 name[12]一栏输入为‘ * ’，又通过 delete 在 27 行被释放掉，然后跳出循环。因此，保留了如图 6.1 的三项，指针 head 指向链的首项，指针 tail 指向链的尾项，而 temp 因刚刚把它指向的第四块数据释放掉，故 temp 指针无值。

这个程序是示意性的，实际的链表可以做很多操作，在“数据结构”课程中会有详细的讨论。

思考题

1. 什么是地址，什么是地址中的内容，两者的区别是什么？
2. 尝试建立一个二维数组并画出该数组在内存中的存储状态图。
3. 什么是指针？地址和指针有什么样的关系？
4. 指针的值和类型是怎样规定的？它有哪些运算？
5. 用指针可有几种表示数组元素的方法，试简述三种方法的使用方法。
6. 简述指针和数组的关系。
7. 指针有哪些运算？和普通的数据类型的运算有什么不同？
8. &和*的用法各有几种？它们的使用方法是什么？
9. 试述函数指针的用法。在计算机上实现 simpson 函数。
10. 试述指针在函数的参数传递中的作用及其使用方法。
11. 简述指针和函数的关系。
12. 怎样使用动态分配运算符对指针变量进行动态分配？
13. 什么是引用？引用和指针的区别是什么？
14. 引用性参数具有哪些优点？
15. 简述指针在 C++语言中的重要性和它在程序安全方面的负面影响。
16. 检验自己是否已经看懂了 6.1 节中的选择排序的程序，尝试将这个例子改造为从小到大的排列顺序。

练习题

1. 说明下面定义的指针的含义

```
float *pf;
double *pd[2][3];
int (*pi) [2];
char *ch = 'abcde';
char *name[] = {'Tom','John','Lennis'};
```

2．读下面的代码并完成要求的问题

已知程序的第一行输出数据是 65fdf4，请判断下面的四行输出哪些可以确定其输入值，请写出那些可以判断的值。

```
#include "stdio.h"
main(){
  int tmp = 10;
  int *p = &tmp;
  int *q = p;
  printf ("%x\n",q) ;
  printf ("%d\n",*p) ;
  printf ("%x\n",&tmp) ;
  printf ("%x\n",&p) ;
  printf ("%x\n",p) ;
}
```

3．下面语句声明的是什么?

double*ara[4][6]；

a．一个双精度浮点型数组

b．一个双精度浮点型指针数组

c．一个非法的声明

4．设有以下的语句:

```
int a[10] = {0,1,2,3,4,};
int *p = a;
```

请指出以下的对 a 数组元素的正确引用是哪些？并指出其值是什么。

a[p-a], p[2], * (* (a+1)), * (&a[3])

5．编制函数 char * FindPlace（char *str, char c）；该函数返回字符串 str 中第一次出现字符 c 的位置以后的字符串。如果没有 c 字符则返回一个空字符串。

6．完成下面的一段程序，使该程序能够输出指定的二维数组任意行任意列元素的值

```
main () {
  static int a[3][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  int  (*p) [4], i, j ;
  p = a ;
  scanf ("i = %d, j = %d", &i, &j) ;
  printf ("a[%d, %d] = %d\n", i, j, _______) ;
}
```

请在空格处填入代码，使得运行情况可以达到如下的效果：

```
i = 0, j = 0
a[0, 0] = 1
```

7. 已知如下的结构：`static char *name[]={"Tom","John","Follow me"}`

试编写一个程序来输出这个数组中的数据，输出效果如下：

```
Tom
John
Follow me
```

8. 仔细检查以下的代码，完成题目。

```
#include <stdio.h>
void main (void) {
  int *p;
  p = new int;
  *p = 10;
  cout << "int value is:" << *p;
  delete p;
}
```

请指出程序的输出，并详细描述这段程序在内存中执行时的变化情况。并回答下列问题：在程序的第三行执行后，p 中的数据是什么？*p 的值是什么？在程序的第四行执行后，p 中的数据是什么？*p 的值是什么？如果在 delete p ；语句之后还有输出&p 的语句 printf（"%x\n",&p），这条语句有意义吗？

9. 编写求 3 * 4 阶矩阵和其自身转置矩阵的乘积的程序，自定义结构。要求具有比较好的模块化性能。完成上述题目后，尝试编制可以允许用户自己输入矩阵的阶数的矩阵乘积程序。

10. 自己设计一种数据结构，编写一个函数 IMalloc（n），当用户输入一个 n 以后，可以在内存中为用户分配 n 个字节的空间，并将内存空间的首地址的指针返回给用户。在编制一个函数 IMFree（）来释放这些内存空间。（提示：可参考 11 题的结构）

11. 有如下定义的结构：

```
typedef struct {
  char m_char;
  StringNode *Next;
}StringNode
```

试利用这种结构存储字符串,并编制函数 SearchMax 在这个字符串中检索出出现次数最多的字符。

第七章 类与对象

类（class）的概念是面向对象程序设计的核心概念。把数据和对象的操作和处理封装在一个程序模块中的方法，可以说是人们积几十年程序设计实践的经验总结。把程序以类的形式组织成若干模块，使其获得了最佳的结构特性，类的概念的引入使程序设计技术发生了革命性的转变。从结构程序设计（SP）中以函数作为程序的基本模块转变为面向对象程序设计（OOP）中，以类作为程序的基本模块，这一变化使程序设计技术出现了质的飞跃。

类是对现实世界中客观事物的抽象，通常将众多的具有相同属性的事物归纳、划分成为某个类。面向对象方法中的类，是对具有相同属性和行为的同一类对象的抽象描述，其内部包括属性（本类的数据成员）和行为（本类的成员函数）两个主要部分，即是说，类以数据为中心，把相关的一批函数组成为一体。

类的概念抓住了程序的本质。程序的基本元素是数据，而函数是围绕数据进行的处理和操作。抓住了数据这个“纲”，程序中关系复杂的各种函数就变得脉络清楚，可以随着相应的数据组合成类。类的使用使得：

- 程序设计本身更有条理了；
- 程序的可读性更好了；
- 程序设计的过程真正像是机器部件的组装；
- 程序由多个程序员设计变得方便和自然；
- 由于程序的零部件化，使得程序的可重用性变成切实可行的事情，等等。

引入了类（class）和对象（object）的概念，就使得 C++语言与 C 语言相比发生了本质的变化。

class 和 object 的概念在 C++编程中所起的作用，可以从两个角度来分析。

第一，从程序的组织角度，C++通过 class 把数据以及对这些数据进行处理和运算的函数封装为互相关联的程序模块，通过对象（也即类实例，或自定义类类型的变量）及相关方法与手段对所定义的类进行使用，这与 C 和 Pascal 等语言把程序划分为具有互相调用关系的函数或过程是不同的。

第二，从数据类型的角度，C++通过 class 引入了抽象数据类型的概念，一个由数据成员和函数成员组成的类就是一种新的数据类型。C++语言为用户提供了设计反映不同应用背景特征的千变万化的数据类型的可能性。

在 C++程序中，程序员可以根据需要定义多种多样的数据类型：stack（栈），queue（队列），set（集合），vector（向量），matrix（矩阵）等。

其它通用语言可能设置的类型，C++程序员都可以方便地定义。同时与具体问题密切结合的类型也出现在C++程序中，如：windows（窗口），menu（菜单），student（学生），employee（雇员），car（小汽车），elevator（电梯）等。

通过类与对象而实现的面向对象程序设计的三大特征是：封装性、继承性、多态性。通过抽象对所要处理的问题进行划分、进行归类，通过类（class）类型对所要处理的问题进行抽象描述，从而将逻辑上相关的数据与函数进行封装。封装所进行的“信息隐蔽”为的是减少对象间的联系，提高软件的模块化程度，增强代码的重用性。

通过类的继承，使新生成的所谓派生类可从其基类那里得到已有的属性（数据）和行为特征（方法），从而可简化人们对事物的认识和描述。面向对象语言通过继承机制很方便地实现了程序代码的可重用问题。

多态性是通过对函数和运算符进行重载以及通过在基类及其派生类间对虚函数进行使用来具体体现的。多态性可使程序易于编制、易于阅读理解与维护。

本章介绍类与对象的说明及其相关使用，下一章将重点介绍继承性与多态性方面的内容。

7.1 设计一个栈类

为了学会OOP方法，首先让我们看看C++程序中类及其对象是怎样工作的。

栈（stack）是程序设计过程中经常碰到的一种数据结构形式，它对于数据的存放和操作有下面这样的特点：

（1）它只有一个对数据进行存入和取出的端口；

（2）后进者先出，即最后被存入的数据将首先被取出。其形式很像一种存储硬币的小容器，每次只可以从顶端压入一个硬币，而取出也只可从顶端进行，即后进先出。

这样的数据存储和管理形式在一些实际的程序设计中很有用。例如，编译系统中（这是一类比较复杂的程序），对于函数调用的处理，对于表达式计算的处理，都是利用了栈这样的数据结构。

当要设计一个处理某类问题的自定义类（类型）时，首先要对问题进行分析，抽象出该类所要处理的数据信息，进而设立相呼应的各数据成员；而后考虑准备对这些数据信息要施加哪些操作，从而进一步抽象出该类的各成员函数。

本例要考虑自定义栈（stack）类型，而栈中需要保存并处理一批同类型的数据，可用一个数组来实现此功能。对数据的操作总从栈顶进行，需要设一个

变量来记录栈顶的当前位置。从而抽象出栈类所需要的如下两个数据成员：

```
float data [maxsize];   //data 中存放栈的实际数据
int top;                //top 为栈顶位置
```

而后考虑准备对栈中数据要施加哪些操作（作为示例，此处只考虑如下三种操作）：

push — 将一个数据“压入”栈顶；
pop — 将栈顶那一数据“弹出”并返回；
empty — 判断当前栈是否为空（栈）。

从上述三种操作而抽象出栈类所需的如下三个成员函数：

```
void push (float a);    //将数据 a“压入”栈顶
float pop (void);       //将栈顶数据“弹出”并返回
bool empty(void);       //判断栈是否为空
```

为了显示对象的“诞生”与“死亡”信息，还可对所设计的栈类加入显式的构造与析构函数。从而可设计出如下的自定义栈类型 stack（它含有两个数据成员以及五个成员函数）。

下述程序还通过在 main 函数中说明 stack 类型的对象 s1、s2 来表示两个具体的栈，进而通过类成员函数对那两个具体的栈进行不同的操作，并显示栈中数据来证明操作的正确性。

```
//program 7_1.cpp
# include<iostream.h>
const int maxsize=6;
class stack{
   float  data[maxsize];
   int  top;
public:
    stack(void);
   ～stack(void);
   bool  empty(void);
   void  push(float  a);
   float  pop(void);
};
stack::stack(void){
    top=0;
   cout<<"stack  initialized."<<endl;
```

```
}
stack:: ~stack(void){
   cout<<"stack destroyed"<<endl;
}
bool stack::empty(void){
   return top==0;
}
void stack::push(float a){
    if(top==maxsize){
       cout<<"Stack overflow!"<<endl;
        return;
    }
    data[top]=a;
    top++;
}
float stack::pop(void){
    if(top==0){
        cout<<"An empty stack!"<<endl;
         return 0;
    }
    top--;
   return data[top];
}
void main( ) {
   stack s1,s2;
   for(int i=1;i<=maxsize;i++)
       s1.push(2*i);
    for(i=1;i<=maxsize;i++)
      cout<<s1.pop( )<<" ";
    for(i=1;i<=maxsize;i++)
       s1.push(2.5*i);
    for(i=1;i<=maxsize;i++)
      s2.push(s1.pop( ));
    cout<<endl;
```

```
    do
      cout<<s2.pop( )<<" ";
    while(!(s2.empty( )));
}
```

该程序执行后的屏幕显示结果如下：

```
stack initialized.
stack initialized.
12  10  8  6  4  2
2.5  5  7.5  10  12.5  15
stack destroyed.
stack destroyed.
```

程序的开始处给出了类 stack 的完整说明，而在主函数中则介绍了如何利用类 stack 来进行操作。注意：在主函数中，必须通过类对象如 s1 及 s2 来对该 stack 类的公有成员函数进行调用，这一点与前面章节中对全局函数的调用方式是不相同的（那时并不需要类对象，而仅通过使用函数名附带上其实参即可）。

栈（stack）又称堆栈，是一种程序设计中常用的数据存储和操作形式，其对数据的“后进先出的操作方式可以用人们常见的一种存放硬币的工具来描述（见图 7.1）。在以后的数据结构课程中还会有详细的讲解。

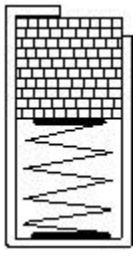


图 7.1 一种保存硬币的工具的示意图

7．2 类和对象的说明

一个类说明的常用格式为：

```
class <自定义类类型名> {
 private:
     <各私有成员说明>;
 public:
     <各公有成员说明>;
};
```

有以下几点需要注意：

(1) private 关键字后列出本类的私有成员；public 关键字后列出本类的公有成员。

成员又区分为数据成员与函数成员（也称成员函数）两种。数据成员代表该类对象含有的数据，描述的是该类对象的属性；而函数成员则代表如何对该类对象所含数据进行操作的代码，描述的是对该类对象的处理方法。类所含有的成员函数既可放于类定义体的花括号之中，也可按类体外定义方式放于花括号之外进行说明（放在类体外说明时，类内必须有其函数原型，且类外函数说明的前面必须用“ <类名>:: ”来限定)。

(2) private 与 public“ 说明段 ”的顺序可以任意；每一类型的“ 说明段 ”均可以出现多次（如可出现多个 public“ 说明段 ”等)，或者一次也不出现（若不需要这种类型的成员的话）;若紧随左花括号后的第一个“ 说明段 ”为 private 的话，则该 private 关键字可以省略。

(3) 类的 private 私有成员：只可在本类中对这些成员进行访问，在别处是“ 不可见的 ”。类的 public 公有成员：这些成员不仅在类中可以访问，而且在建立类对象的其他模块中，也可以通过对象来访问它们。

(4) 按如下方式来说明对象（即某种类类型的变量，或称该类的一个实例（instance))：

<自定义类类型名> <对象名 1>， … ，<对象名 n>；

按如下方式来使用对象分量（也称类成员函数及类的数据成员)：

<对象名>.<分量名>

注意，若在该类的说明中含有带参数的构造函数的话，则在说明对象的同时，要给出具体实参去初始化该对象（参看下一节内容)。此时说明对象的方式应改为：

<自定义类类型名><对象名 1> (<实参表 1>)， … ，<对象名 n> (<实参表 n>)；

(5) 与其他“ 小级别 ”的变量以及结构体变量相类似，也可以说明类对象的数组（数组分量为对象）以及指向对象的指针。另外，对对象还可进行如下一些操作与使用：同类型的对象间可以相互赋值；对象可作为函数参数（如，对象作形参，对象指针作函数参数等)；函数的返回值可以是对象（或指向对象的指针)；可以在一个类中说明具有类类型的数据成员等。

下面看一个使用自定义类(类型)的程序示例。

```
//program 7_2.cpp
#include <iostream.h>
```

```
class point {
    private:
        float xcoord,ycoord;
    public:
        float getx() {return xcoord;}
            //注意：函数体中的 return 语句也可改写为
            // “ return (*this).xcoord; ”或“ return this->xcoord; ”
        float  gety();
        void setx(float x) {xcoord=x;}
        void  sety(float  y);
        void  displayx()  {cout<<"xcoord="<<xcoord<<endl;}
        void  displayy()  {cout<<"ycoord="<<ycoord<<endl;}
};
float  point::gety()  {return  ycoord;}
void point::sety(float y)  {ycoord=y;}
void main() {
    point obj1, *p, objArr[6];
    obj1.setx(123.5);
    //cout<<obj1.xcoord<<endl;
        //出错！在建立类对象的 main 函数中，不可通过对象来访问类的 private 私有成员
    cout<<obj1.getx()<<endl;      obj1.displayx();
    p=new point;                  (*p).setx(5.6);              p->sety(7.8);
    float k=(*p).getx();          float m=p->gety();
    cout<<"k="<<k<<endl;          cout<<"m="<<m<<endl;
    for (int i=0; i<3; i++){
        objArr[i].setx(i+10.1);
        objArr[i].sety(i+20.2);
    }
    p=&objArr[5];
    while (p>=&objArr[3]){
        p->setx(88.8);    p->sety(99.9);
        p--;
    }
    for (i=0; i<6; i++)
```

```
        cout<<objArr[i].getx()<<"        "<<objArr[i].gety()<<endl;
}
```

程序执行后的显示结果如下：

```
123.5
xcoord=123.5
k=5.6
m=7.8
10.1        20.2
11.1        21.2
12.1        22.2
88.8        99.9
88.8        99.9
88.8        99.9
```

这个类型 point 表示的是平面上一个点，它由两个坐标值组成，其“值集”应该是平面上的所有点集，每一值占用相当于两个浮点数的 8 个字节。为了说明两个 point 类型的“变量”p1 和 p2，可以使用如下形式的对象说明语句：

```
point  p1,p2;
```

对于具体的两个“点”p1 与 p2，可以根据类 point 的说明，对其施以所需的某些操作，如：

```
p1.setx(3.5);              p1.sety(4.7);
p2.sety(p1.getx());        p2.setx(p1.gety());
```

第 1 行的两个函数调用语句，为点 p1 设置了两个坐标值，使得 p1.xcoord=3.5,p1.ycoord=4.7。由于 class 说明中指明类 point 的数据成员 xcoord 和 ycoord 是私有的，因此变量 xcoord，ycoord 除了在类内可以使用之外，在类外不能出现，因此若在 main 中直接用

```
p1.xcoord=3.5; p1.ycoord=4.7;
```

是非法的。由此也可以看出私有成员与公有成员的区别。设定成员的访问控制可以实现信息隐藏，通过类和对象的封装：

（1）在类中指定一些数据成员和函数成员为私有的，这就保证了程序运行的安全性。私有成员只有本类的函数成员或友元可以访问，其安全性可以保证，出了错也容易发现。

（2）把函数与它所处理的数据联系到一起，使得程序中大量的操作、运算、处理等得到了最合理的划分，这种划分的合理性，其直接结果就是程序模块的可重用性加强了。

第 2 行是为点 p2 赋值，但这里所赋的两个值是从 p1 点已赋的值中得到的，其执行结果应为：

```
p2.xcoord=4.7,  p2.ycoord=3.5;
```

另外也容易看出，对象说明与变量说明没有什么区别，当把类视为一种由用户定义的新的数据类型时，二者就完全一致了。另外，下节还要介绍对象的初始化问题，也与变量相似。

程序中还使用了对象的数组（数组分量为对象）以及指向对象的指针。对它们的使用方式也与先前相类似。

还有一点要注意的是：在类的函数成员（定义）代码中，凡没显式给出对象名（类变量名）而对类成员进行的访问或使用，均隐含解释为是对“当前调用者对象（变量）”的成员进行访问或使用。另外，也可使用另一种方式来访问或使用“当前调用者对象（变量）”的成员，那就是通过使用 this 指针。

this 为 C++保留字，它为一个指针，在任一个类成员函数的运行过程中，指针 this 总指向当前调用者对象（或说 this 总代表当前调用者对象的地址）。所以，类成员函数中总可以用“*this”来表示当前调用者对象，也可用“this-><成员名>”或“(*this).<成员名>”来表示当前调用者对象的某成员；还可以通过“return this”来返回该当前调用者对象的指针（即地址），用“return *this”返回该当前调用者对象（本身）。

但注意，在类成员函数中访问或使用当前调用者对象的某成员时，通常不使用上述这种显式的“this-><成员名>”或“(*this).<成员名>”的方式，而使用缺省了“this->”及“(*this).”修饰符号的即仅使用“<成员名>”的这种所谓的隐含方式。例如，上述示例中，在 getx 内使用数据成员 xcoord 时，并不使用(*this).xcoord 或 this->xcoord，虽然也可那样来使用。

this 指针的概念较抽象，下面给出对它的进一步说明以加强理解。

在类的定义中，类的成员函数总是要对该类的数据成员进行处理。但类是“虚”的，它只是一种抽象，一种设计，要处理的实际上是该类的对象。

从另一方面说，类的函数成员的定义是说明性的，它的执行必须与该类的某一对象相联系。例如：

```
cobj.print ();
```

这就产生了一个问题：在定义函数成员时还没有具体的对象出现，但函数成员的定义中却往往要对它所将要依附的对象进行处理。为了解决这个问题，C++语言为类的定义设置了一个抽象的指针常量：this 指针。它无须用户的定义，相当于在类 C 的每一成员函数中隐含定义了：

```
const C* this;
```

其特点是：

(1) this 是类 C 的指针，它指向类 C 的对象。

(2) this 是在该类的对象 cobj 被创建后，其成员函数 f()被调用时，this 也就同时被说明和创建，即

```
C cobj；
cobj.f()
```

相当于在 f()中有一说明

```
const C* this=&cobj;
```

在函数的运行过程中，指针 this 总代表着对象 cobj 的地址。即是说，可以用*this 表示该对象，也可用类似于 this->n 以及 this->print()来表示该对象的成员。有时也使用 return this；表示返回该对象的指针。

7.3 对象的初始化，构造与析构函数

7.3.1 基本概念及定义

对象也被称为类变量，一个类的对象是这个类的一个实例。和变量一样，它也可以为其数据成员赋初值。不过对象的初始化情况比较复杂，可以有下列多种不同的方式，其中最重要的方式是构造函数。

1. 公有数据成员

如果一个类的数据成员是公有的，那么其对象的初始化与一般变量，结构变量或变量数组的初始化没有什么区别。例如：

```
class address{
  public:
    long  telenum;
    char  addr[30];
};
class person{
  public:
    char  name[15];
    int  age;
    address  paddr;
};
```

```
person p1={"Zhang Hua",23,{2475096,"NanKai University"}};
```

大多数类的数据成员是私有的或保护的,不能采用这种方式。

2．公有的初始化函数

在类中设置公有的初始化函数完成此项任务，在上节中的类 point 中，函数 setx(float x)和 setY(float y)可以完成初始化的任务，也可以专门设计一个初始化函数：

```
public:
  void initpoint( ){
     xcoord=0; ycoord=0;
   };
```

3．构造函数

为了更好地解决对象的初始化问题，C++规定在类的说明中可以包含一个或多个特殊的公有函数成员——构造函数。构造函数具有下列特征：

（1）函数名与类名相同。

（2）无函数（返回）类型说明。

（3）构造函数在一个新的对象被建立时，该对象所隶属类的构造函数自动地被调用，对这个对象完成初始化工作。

（4）在上一条中提到的新对象的建立包括两种情况,①在对象说明语句中；②用 new 函数建立新的动态对象时。

（5）如果一个类说明中没有给出显式的构造函数，系统将自动给出一个缺省的（隐式的）构造函数：

<类名>(void){}

这个函数什么也不做。

（6）如果说明中包括多个构造函数，一般它们有不同的参数表和函数体。系统在（自动）调用构造函数时按照一般函数重载的规则选择其中之一。

例如在 class point 中可有显式的构造函数。

```
point(void){
   xcoord=0.0;
   ycoord=0.0;
}
point(float ix,float iy){
   xcoord=ix;
   ycoord=iy;
}
```

前一个无参的构造函数把每个新对象——一个平面上的点初始化为（0.0,0.0），而第二个构造函数则为新的对象点赋予由用户指定的初值。如果这两个构造函数同时存在，程序员在说明新的对象时可任意选择：

```
point p(2.0,1.5), q;
```

上面的对象说明语句建立了两个对象 p 和 q，对于点 p，用参数 2.0 和 1.5 通过第二个构造函数进行初始化，而点 q 没有提供初始化的值，它将自动选择第一个构造函数，把 xcoord，ycoord 初始化为 0.0。

4. 析构函数

与构造函数在对象生成时自动执行相对应，析构函数是专门用来在对象的生存期结束时做善后工作的。它也是类的特殊函数成员：

（1）析构函数名一律为～〈类名〉，如～point。

（2）无函数返回类型。

（3）无参数。

（4）一个类只可有一个析构函数，也可以缺省。

（5）在对象注销时，包括用 delete 函数释放动态对象时，系统自动调用析构函数。

（6）若某个类定义中没有给出显式的析构函数的话，则系统自动给出一个缺省的（隐式的）如下形式的析构函数：

```
~<类名>(void){}
```

此函数什么事情也不做。

这里介绍的构造函数和析构函数是类的要素，它们在一个对象的生存期的开始阶段和结束阶段起着举足轻重的作用。承担着对象的初始化和收尾工作。

形象地说，每当对象“诞生”时（通过对象说明语句或通过 new 来建立或生成新对象时），系统都将自动调用对象所属类的构造函数，以完成对象的初始化工作；而每当对象“死亡”前（当对象退出其说明区域即作用域，或使用 delete 释放动态对象时），系统都将自动调用对象所属类的析构函数，以完成对象撤销前的善后工作。

7.3.2 构造与析构函数使用示例

下述的自定义类 String 具有一个显式的构造函数与一个显式的析构函数。对象“诞生”时，在构造函数中通过 new 分配了动态空间（系统资源），对象“死亡”时，在析构函数中应通过 delete 来释放所申请到的动态空间（系统资源）。

请分析下述程序执行后会显示出什么样的结果?

```
//program 7_3.cpp
#include <iostream.h>
#include <string.h>
class String {
      char * text;
    public:
      String( char * str );
       ~String();
       void printStr(){cout<<text<<endl;}
};
String::String( char * str ){
       cout<<"enter 'String::String', str=>"<<str<<endl;
      text = new char[strlen(str)+1];
      strcpy( text, str );
}
String::~String(){
       cout<<"enter 'String::~String', text=>"<<text<<endl;
       delete[]text;
}
void main(){
       String str1("a1d11");
       String str2("s22g22");
       str1.printStr();
       str2.printStr();
      cout<<"ending main!"<<endl;
}
```

程序执行后的显示结果如下:

```
enter 'String::String', str=>a1d11
enter 'String::String', str=>s22g22
a1d11
s22g22
ending main!
enter 'String::~String', text=>s22g22
```

enter 'String::~String', text=>a1d11

7.4　自定义类及其使用

本节首先自定义一个集合类型 set，并利用它来实现与集合有关的各种运算。而后使用与 7.1 节相类似的自定义栈类型 stack 来求解迷宫问题。

7.4.1　创建一个集合类型 set

任何一个类的定义都相当于一个用户定义的新数据类型。特别是对于一些通用类型，比如，复数，矩阵，集合，向量等在数学上已有比较规范的那些运算。

集合是某一类数据的聚集，其中的数据为同一类型，数量不定，数据间没有顺序关系。集合类型的变量应可以进行若干种特殊的运算，例如，把一个元素数据加入到集合中，判定集合是否为空，把两个集合中的公共元素组成一个新的集合等等。下面的程序是通过定义一个 set 类，实现上面的目标。

首先设计（抽象）出集合类型 set 中的数据成员：用于存放集合元素的私有数据成员 elems（为一个数组），以及另一个表示当前集合中实际具有了多少元素的私有数据成员 card。

另外提供 10 个公有函数成员（其中包括一个构造函数），用于实现所考虑到的集合运算，它们构成使用 set 类来处理问题的“对外接口”。

提供的自定义集合运算为以下 9 种：

1）判断元素 elem 是否为某集合的成员（由 Member 函数来完成）；
2）将 elem 元素加入到某集合之中（AddElem 函数）；
3）将 elem 元素从某集合中删去（RmvElem 函数）；
4）将某集合的所有元素，拷贝到另一个集合中（Copy 函数）；
5）判断二集合包含的元素是否完全相同（Equal 函数）；
6）将一个集合中的所有元素显示出来（Print 函数）；
7）求二集合的交集=>第三个集合（Intersect 函数）；
8）求二集合的并集=>第三个集合（Union 函数）；
9）判断一个集合是否包含于另一个集合之中（Contain 函数）。

```
//program 7_4.cpp
#include <iostream.h>
```

```
const int maxcard=20;                    //集合元素最大数
enum ErrCode {noErr,overflow};
class set {
      int elems [maxcard];               //存放集合元素
      int card;                          //记录元素个数
 public:
      set (void){card=0;};               //构造函数
      bool Member(int);
       ErrCode AddElem(int);
      void RmvElem (int);
      void Copy (set*);
      bool Equal (set*);
       void print();
       void Intersect(set*,set*);
       ErrCode Union(set*,set*);
      bool Contain (set*);
};
//判断元素 elem 是否为某集合的成员 -- 判断形参 elem 是否等于当前调用者
//对象的 elems[0]->elems[card-1]中的某一元素，是则返 true，否则返 false。
bool set::Member(int elem) {
       for(int i=0;i<card;i++)
             if (elems[i]==elem)
                   return true;
        return false;
}
//将 elem 元素加入到某集合之中
ErrCode set::AddElem(int elem) {
       for(int i=0;i<card;i++)
            if (elems[i]==elem)    //集合中已有 elem 时，不重新加入而立即返回
             return(noErr);
       if (card<maxcard) {
             elems[card++]=elem;    //将 elem 加入到对象的私有成员 elems 数组中
             return(noErr);
       }
```

```
        else
            return  overflow;   //数组已满时，返回一个出错标志
    }
    /*   将 elem 元素从某集合中删去  --  当从数组 elems 中找到形参 elem 后，将该数组元素的所有“ 后继 ”元素统统“ 前移 ”一个位置，即数组中不可出现“ 空洞 ”。删除一元素后，记录集合元素数的 card 变量值减 1。  */
    void set::RmvElem (int elem) {
        for(int  i=0;i<card;i++)
         if  (elems[i]==elem){
              for(;i<card-1;i++)
               elems[i]=elems[i+1];
              card--;
         }
    }
    //将某集合的所有元素，拷贝到另一个集合中  --  将当前调用者对象的集合元素，
    //拷入由指针参数 sp 所指向的对象之集合中。
    void set::Copy (set* sp) {
        for(int i=0;i<card;i++)
             sp->elems[i]=elems[i];
        sp->card=card;
    }
    //判断二集合包含的元素是否完全相同  --  当前调用者对象之集合
    //与指针参数 sp 所指对象之集合是否相同。
    bool set::Equal(set* sp){
        if(card!=sp->card)
            return  false;   //元素个数不同，则二集合必不相同
        for(int i=0;i<card;i++)
            if(!sp->Member(elems[i]))
                 return false;
            // (作为实参的)当前调用者对象的 elems[i]分量不为指针 sp 所指对象
            //之集合的元素  --  也即，当发现有一个元素不相等时，则返 false
        return true;
    }
    //将一个集合中的所有元素以某种格式显示出来
```

```
void set::print () {
    cout << "{";        //最初显示一个"{"
    for(int i=0; i<card-1; i++)
        cout <<elems[i]<< ",";
        //显示出当前调用者对象的 elems 数组中各元素，元素间以逗号分割
    cout <<elems[card-1];
    cout << "}"<<endl;              //最后显示一个"}"
}
```

/* 求二集合的交集 => 第三个集合 -- 当前调用者对象之集合与第 1 指针参 sp1 所指对象之集合的交集 => 第 2 指针参 sp2 所指对象之集合。

实现方法为：用调用者对象的诸 elems[i]，与 sp1 所指对象的诸 elems[j]一一作比较，将其中的相同者(共有元素)送入 sp2 所指对象的 elems 数组的从 0 下标开始的各单元中。*/

```
void set::Intersect(set* sp1,set* sp2){
    sp2->card=0;
    for(int i=0;i<card;i++)
        for(int j=0;j<sp1->card;j++)
            if(elems[i]==sp1->elems[j]){
                sp2->elems[sp2->card++]=elems[i];
                break;
            }
}
```

/* 求二集合的并集 => 第三个集合 -- 当前调用者对象之集合与第 1 指针参 sp1 所指对象之集合的并集 => 第 2 指针参 sp2 所指对象之集合。

注意：实现本函数时，用到本 set 类的成员函数 Copy 以及 AddElem。它先将 sp1 所指对象之集合，通过 Copy 拷贝到 sp2 所指对象之集合（目的集合）中，而后再将调用者对象之集合的诸元素 elems[i]，通过 AddElem 加入到目的集合中。 */

```
ErrCode set::Union(set* sp1,set* sp2){
    sp1->Copy(sp2);
    for(int i=0;i<card;i++)
        if(sp2->AddElem(elems[i])==overflow)
            return(overflow);
    return(noErr);
}
```

//判断一个集合是否包含于另一个集合之中 -- 当前调用者对象之集合，

```
//是否包含指针参 sp 所指对象之集合，是返 true，否则返 false。
bool set::Contain (set* sp) {
        for(int i=0;i<sp->card;i++)
              if(!Member(sp->elems[i]))
               return false;
        return true;
}
void main() {
  int i,b;
  set s,s1,s2,s3,s4;                  //说明 5 个 set 类型的对象（变量）
 for (i=0;i<10;i++){                  //为对象 s,s1,s2 设置初值
        b=s.AddElem(i);
        b=s1.AddElem(2*i);
        b=s2.AddElem(3*i);
   }
 //显示出对象 s,s1,s2 的当前值
  cout<<"s=";       s.print();
  cout<<"s1=";      s1.print();
  cout<<"s2=";      s2.print();
 //将元素 0->4 从集合 s,s1,s2 中删去（使用 RmvElem 函数）
  for (i=0;i<5;i++){
        s.RmvElem(i);
        s1.RmvElem(i);
        s2.RmvElem(i);
    }
  cout<<"After RmvElem(0->4), s=";       s.print();
              //又一次显示出对象 s,s1,s2 的当前值
  cout<<"After RmvElem(0->4), s1=";      s1.print();
  cout<<"After RmvElem(0->4), s2=";      s2.print();
  s.Intersect(&s1,&s3);          //s 与 s1 的交送 s3
  b=s.Union(&s2,&s4);            //s 与 s2 的并送 s4
  cout<<"s3=s*s1=";      s3.print();
  cout<<"s4=s+s2=";      s4.print();
  if (s3.Equal(&s4)) cout<<"SET s3=s4 "<<endl;
```

```
        else    cout<<"SET s3!=s4 "<<endl;
    if (s3.Equal(&s3)) cout<<"SET s3=s3 "<<endl;
        else    cout<<"SET s3!=s3 "<<endl;
    if (s3.Contain(&s4))          cout<<"SET s3 contains s4 "<<endl;
        else    cout<<"SET s3 do not contains s4 "<<endl;
    if (s4.Contain(&s2)) cout<<"SET s4 contains s2 "<<endl;
        else    cout<<"SET s4 do not contains s2 "<<endl;
}
```

程序执行后的显示结果如下：

```
s={0,1,2,3,4,5,6,7,8,9}
s1={0,2,4,6,8,10,12,14,16,18}
s2={0,3,6,9,12,15,18,21,24,27}
After RmvElem(0->4), s={5,6,7,8,9}
After RmvElem(0->4), s1={6,8,10,12,14,16,18}
After RmvElem(0->4), s2={6,9,12,15,18,21,24,27}
s3=s*s1={6,8}
s4=s+s2={6,9,12,15,18,21,24,27,5,7,8}
SET s3!=s4
SET s3=s3
SET s3 do not contains s4
SET s4 contains s2
```

7.4.2 利用 stack 类型解迷宫问题

下面用 stack 来解迷宫问题。

栈 stack 是一种很有用的数据结构，现在我们把 stack **类作为一种用户定义的数据类型，进而把它应用在解有趣的迷宫问题。**

如下是求解迷宫问题的要点：%

1．所谓迷宫问题（Mazing Problem**）就是在迷宫中找一条从入口到出口的可通行的路径。在具体的编程问题中把迷宫定义为元素值为 0 或 1 的二值矩阵：**

maze［m］［p］；

其中 maze［0］［0］**为迷宫的入口点，而** maze［m－1］［p－1］**为迷宫的出口点。并规定：**

当迷宫矩阵中的(i,j)点为“可达”点时，maze[i,j]=0；

当迷宫矩阵中的(i,j)点为“不可达”点时，maze[i,j]=1。

2．在迷宫中的每一个位置（i，j），可以向 8 个方向移动一步，这 8 个方向可以记为 N，NE，E，SE，S，SW，W，NW。见图 7.2。%

例如从位置（i，j）沿 NE 方向移动一步，移到位置（i－1，j＋1）。

3．可按如下方式来表示所找到的那条“通路”:

step1: (0,0),SE -- 从(0，0)点朝“东南”迈一步，到达(1，1)点；

step2: (1,1),NE -- 从(1，1)点朝“东北”迈一步，到达(0，2)点；

step3: (0,2)，E -- 从(0，2)点朝“东”迈一步，到达(0，3)点；

.....

4.实现时,程序中将 8 个方向: N(北),NE(东北),E(东),SE(东南),S(南),SW(西南)， W(西)，NW(西北)，分别用 0 到 7 这 8 个数码来表示（即，程序中的 dir 之值)。

5．程序中使用了一个自定义 stack 类（类型），而后通过使用 stack 类对象来保存当前已在 maze 迷宫矩阵中找到的“半截路”(由从 maze[0，0]到 maze[m-1，p-1]的那条“通路”之“前半截”的若干个“迈步”构成，而每一“迈步”均以类似于“(0,0),SE”的这种格式来表达)。

6．程序中使用的 items 结构用于保存上述的“迈步”信息:

```
struct items{
  int x,y;
  int dir;
};       //从(x,y)点，沿 dir 方向，往下“迈步”
```

7．程序中自定义的 stack 类(类型),包含两项私有数据成员 data 与 top，data 中始终保存着当前已在 maze 迷宫矩阵中找到的“半截路”，而 top 则记录当前“半截路”的长度。

```
class stack {
    items data [maxsize];
        //data 中保存“半截路”，其“迈步”信息用 items 结构体来保存
    int top;    //top 记录“半截路”长度
public:
    ......
    void push (items it);
    items pop (void);
    ......
};
```

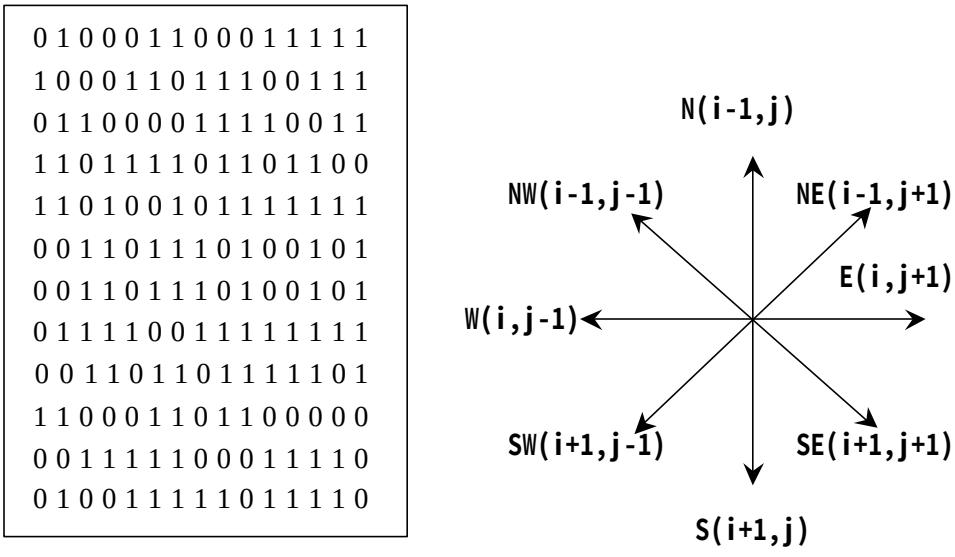


图 7.2 迷宫的（0，1)矩阵和方向说明

下面是为此设计的 C＋＋**程序,读者应注意用户定义的** stack **类作为一种新的数据类型在程序中的作用。**%

```
//program7_5.cpp %
#include <iomanip.h>
#include <process.h>
const int m=12,p=15;            //处理 m 行 p 列的迷宫矩阵
const int maxsize=m*p;          //栈的最大长度为 m*n
struct offsets{
  int a,b;
};
const offsets move[8] = {   //从“当前点”迈向 8 个不同方向时的“偏移量”
      {-1,0},{-1,1},{0,1},{1,1},
      {1,0},{1,-1},{0,-1},{-1,-1}
};
struct items {                  //用于保存“迈步”信息的结构
  int x,y;   int dir;
};
void printDir(int dir){         //根据方向数码值 dir，显示出对应的方向符号串
```

```
        char dirName[8][3]={"N", "NE", "E", "SE", "S", "SW", "W", "NW"};
        cout<<setw(2)<<dirName[dir];
}
class stack {                   //自定义 stack 类（类型）
        items data [maxsize];   //data 中保存“半截路”，其“迈步”信息用 items 结构体保存
        int top;                //top 记录“半截路”长度
  public:
        stack (void){top=0;};                   //构造函数
        bool empty(void){                       //当前栈是否为“空”
                return (top==0)?true:false;
        };
        void push (items it);                   //“压栈”
        items pop (void);                       //“弹出”栈顶元素
        void print(void);                       //显示“迈步”序列
        int gettop(void){return top;}           //获取当前栈长度
};
void stack::push (items it) {
    //在类外定义 stack 类的 push 成员函数，实现“压栈”操作
   if (top==maxsize){
      cout << "stack is full!"<< endl;
        return;
    }
    data[top].x=it.x;               data[top].y=it.y;
    data[top].dir=it.dir;           top ++;
}
items stack::pop (void) {    //pop 成员函数，“弹出”栈顶元素并返回
   if (top==0) {                //注：也可用 if( empty())来进行判断
      cout << "stack is underflow!"<< endl;
        exit(0);                //栈溢出时，退出
    }
    top --;
   return data[top];            //“弹出”栈顶元素并返回
}
void stack::print (void) {   //print 成员函数，显示当前栈中保存的“半截路”
```

```
   cout<<"top="<<top<<endl;
  for(int i=0;i<top;i++) {
       cout<<"step"<<i+1<<":    ("<<data[i].x<<","<<data[i].y<<"),";
       printDir(data[i].dir);          cout<<"\t";
       if( (i+1)%3==0) cout<<endl;     //每显示 3 个“迈步信息”换一行
   }
}
bool mark[m][p];
      /*   mark 矩阵（全局性的二维数组），与 maze 大小相同，标记 maze 中各对应点是否已经“走过”(“试探过”) -- mark 的(g,h)点为 0，意味着 maze 中的(g,h)点为“尚未走过”之点。  */
int maze[m][p] = {    //迷宫矩阵 maze 及其初始值（全局性的二维数组）
                      //注，也可在 main 中通过 cin 来输入各元素的值
      {0,1,0,0,0,1,1,0,0,0,1,1,1,1,1},          {1,0,0,0,1,1,0,1,1,1,0,0,1,1,1},
      {0,1,1,0,0,0,0,1,1,1,1,0,0,1,1},          {1,1,0,1,1,1,1,0,1,1,0,1,1,0,0},
      {1,1,0,1,0,0,1,0,1,1,1,1,1,1,1},          {0,0,1,1,0,1,1,1,0,1,0,0,1,0,1},
      {0,0,1,1,0,1,1,1,0,1,0,0,1,0,1},          {0,1,1,1,1,0,0,1,1,1,1,1,1,1,1},
      {0,0,1,1,0,1,1,0,1,1,1,1,1,0,1},          {1,1,0,0,0,1,1,0,1,1,0,0,0,0,0},
      {0,0,1,1,1,1,1,0,0,0,1,1,1,1,0},          {0,1,0,0,1,1,1,1,1,0,1,1,1,1,0}
};
void path(int m, int p)                 //在 m 行 p 列的 maze 迷宫矩阵中，找一条“通路”
{
      mark[0][0]=true;                  //由(0,0)点“起步”
     items temp={0,0,2};                //将第 1“迈步”定为(0，0，2)
      stack  mazepath;                  //在 stack 类对象 mazepath 中保存已找到的“半截路”
     mazepath.push(temp);               //将第 1“迈步”压入栈中
      while(!mazepath.empty())
      //栈中保存的当前“半截路”不为“空”时方可接着往下“迈步”;
      //否则，意味着找尽了各种可能性还是“无路可走”，则应无解结束。
      {
         temp=mazepath.pop();           //从当前“半截路”的“顶点”处，接着“往前迈”
        int  i=temp.x;     int  j=temp.y;
         int  d=temp.dir;               //沿 dir 方向“往前迈”
         while(d<8)                     //共 8 个方向可迈，要逐一试探该 8 个方向
```

```
        {
        int g=i+move[d].a;
        int h=j+move[d].b;
        if((g<0)||(h<0)||(g>=m)||(h>=p))
              d++;                    //迈步“出界”，试下一方向
        else
        {
          if((g==m-1)&&(h==p-1))      //到达出口点，显示找到的“通路”后返回
          {
                mazepath.print();
                int ind=mazepath.gettop();
                cout<<"step"<<ind+1<<":   ("<<i<<","<<j<<"),";
                printDir(d);          cout<<endl;
                return;               //找到“通路”并显示出后而成功返回
          }
         if( (!maze[g][h]) && (!mark[g][h]) )
           //迷宫矩阵的(g,h)点为可达之点，而且(g,h)点为“尚未走过”之点
          {
                mark[g][h]=true;      //标记(g,h)点为“已走过”之点
                temp.x=i;    temp.y=j;    temp.dir=d;
                mazepath.push(temp);
                     //将迈向(g,h)的上一点(i,j)以及方向值 d 作为一个三元组(一个“迈步”)
                     //存入栈中，意味着，从(i,j)沿 d 方向迈到了(g,h)。
                i=g;   j=h;   d=0;     //而后，将(g,h)当作新起点，从 0 方向接着往下迈
          }
          else    //换下一个方向（重新由(i,j)计算另一个(g,h)）
                d++;
          }       //else ...ending
        }         //while (d<8) ...ending
          //注：若 d 的 8 个方向均告失败的话，则“回退”一步，
          //从栈顶取出“父点”，并从“父点”的原 d 值接着往下“试探”
      }   //while (! mazepath.empty()) ...ending
    cout<<"no path in the maze"<<endl;    //找尽了各种可能性仍无解，失败返回
}
```

```
void main() {
  for (int i=1;i<m;i++)
    for (int j=0;j<p;j++)
          mark[i][j]=false;
          //将 mark 数组各元素均置为 0，认为当前的 maze 迷阵各点全“没走过”
  path(m,p);      //在 m 行 p 列的 maze 迷宫矩阵中，找一条“通路”
}
```

程序执行后的显示结果如下：

```
top=28
step1:  (0,0),SE        step2:  (1,1),NE        step3:  (0,2),  E
step4:  (0,3),  E       step5:  (0,4),SW        step6:  (1,3),SE
step7:  (2,4),  W       step8:  (2,3),SW        step9:  (3,2),  S
step10:  (4,2),SW       step11:  (5,1),  S      step12:  (6,1),SW
step13:  (7,0),SE       step14:  (8,1),SE       step15:  (9,2),  E
step16:  (9,3),NE       step17:  (8,4),NE       step18:  (7,5),  E
step19:  (7,6),SE       step20:  (8,7),  S      step21:  (9,7),SE
step22:  (10,8),  E     step23:  (10,9),NE      step24:  (9,10),  E
step25:  (9,11),  E     step26:  (9,12),NE      step27:  (8,13),SE
step28:  (9,14),  S     step29:  (10,14),  S    %
```

程序说明：%

（1）此程序对于任意给定的 m×p 阶迷宫 maze［m］［p］，可以找到一条从（0，0）起始到（m－1，p－1）结束的可通过路径，并印出路径上各点之坐标。如果迷宫不存在这样的道路，那么函数 path（）将在搜索了所有的可能路径而失败后（这时栈中为空），输出“无路”的信息。%

（2）path（）函数采用回溯式的搜索方法，即搜索到某一位置时，如果 8 个方向都不可能向前进展的话，它将退回一步，改向下一方向继续搜索。在整个搜索过程中 stack 的对象，实际的栈 mazepath 负责记录搜索的过程，向前一步要压进一新的节点，在需回溯时又要退出一项。最终当搜索到终点（m－1，p－1）时，栈中正好保存整个一条成功的路径。

读者可以看到，采用 stack 类型后，通过对其成员函数 push（）和 pop（）的使用，给程序设计提供了极大的方便。%

（3）我们在这个程序中，还可以看到，结构类型的变量，如变量 temp，data［i］等都是 items 类型的。items 是一种结构类型，也可以视为一个类，因此，items 类型的变量 temp，data［i］也是该类的对象，其使用方法如：%

temp.x，temp.y，temp.dir，

data［i］.x，data［i］.y，

等等，与一般从对象引用其成员的方式是一样的。

7.5　类的静态成员及常量成员

由关键字 static 修饰的类成员说明称为类的静态成员。由关键字 const 修饰的类成员说明称为类的常量成员（注意成员又细分为数据成员与函数成员两种）。

7.5.1　类的静态成员

类的静态成员为其所有对象所共享，不管有多少对象，静态成员只有一份存于公用内存中。

1．静态数据成员

该类所有对象的该静态数据成员实际上是同一个变量，其数据将为该类的所有对象所共享（否则的话，同一个类的不同类对象之相同分量名字的各数据成员所占据的存储空间是相互独立的）。

注意，在类中说明的静态数据成员属于引用性说明，即是说，还必须在类外文件作用域中的某个地方对静态数据成员按如下格式进行定义性说明（且仅能说明一次）：

<类型> <类名>::<静态数据成员> = <初值>；

由于类的静态数据成员在一个类中只有一份拷贝，它并不为某一个类对象所特有，所以，对类的静态数据成员的访问通常使用“ <类名>::<静态数据成员名> ”的方式。当然也可通过“ <对象名>.<静态数据成员名> ”的方式来进行指定，但由于只有一份拷贝，其中的<对象名>也只是起到一个类名的作用而已。

例如，设 float xcoord 是类 point 的静态数据成员，p 、q 是 point 的对象，则在任何情况下：

p.getx()和 q.getx()

将永远同时取相同的值。如果 xcoord 是公有数据成员的话，那么

p.xcoord，q.xcoord，point::xcoord

实质上是相同的。

2．静态函数成员

函数成员被说明成静态的，同样将与该类的不同对象无关。对类的静态函数成员的引用通常使用“<类名>::<静态函数成员调用>”的方式（当然也可通过“<对象名>.<静态函数成员调用>”的方式）。

类的静态函数成员与类的非静态函数成员的最大区别在于：类的静态函数成员没有 this 指针，从而无法处理不同调用者对象的各自数据成员值。通常情况下，类的静态函数只处理类的静态数据成员值（它只隶属于类而不属于任一个特定对象）。若要访问类中的非静态成员时，必须借助对象名或指向对象的指针那样的函数参数。

例如在类 point 中，函数成员

```
gety(void)
```

在一般情况下引用形式为：

```
cout<<p.gety( )<<" "<<q.gety( );
```

如果把 gety(void)说明为静态的，

```
static float gety(void){return ycoord;};
```

上述的引用就可以改为：

```
cout<<point::gety( );
```

从这里可以看出，类中的静态函数成员如果只涉及其他静态成员，例如其中的数据成员 ycoord 也是静态成员的话，则 point::gety( )被调用时，函数的执行及返回值都是确定的。但若函数所涉及的数据成员 ycoord 不是静态的，这时 point::gety( )将无确切含义，因为其中的变量 ycoord 对不同的对象来说是不同的。因此，在这种情况下，函数 gety( )的定义应改写为：

```
static float gety(point &ob){return ob.ycoord;};
```

在说明了对象 p，q 之后，函数 gety 的调用形式为：

```
cout<<point::gety(p)<<"       "<<point::gety(q);
```

利用静态函数 gety( )分别得到对象（点）p 和 q 的 Y 轴坐标值。

综合“变量与函数的存储类别”处对 static 的使用，到目前为止，使用 static 可进行的五种说明有：局部静态变量、全局静态变量、具有静态存储类别的函数、类的静态数据成员以及类的静态函数成员。下面对它们稍做区别：

（1）局部静态变量是指在函数或块的内部说明的静态变量，它的作用域仅局部于函数或块（出函数或块后则不可见）；但它的“生命期”却与整个程序的执行期相同。

（2）全局静态变量指的是在所有函数的外部说明的具有单文件级全局性的静态变量（相对多文件程序来说，其作用域也是局部的，仅局部于说明它的那一文件）。它的作用域仅局部于单文件（在其他文件中不可见，从而不可在其他文件中被使用）；但它的“生命期”却与整个程序的执行期相同。

（3）具有静态（static）存储类别的函数称为静态（static）函数（有时也称为内部函数）。这种函数只具有文件级作用域（也称单文件级作用域），即是说，这样的函数只能在本文件的内部被调用，在其他文件中均不可见，从而在其他文件中不能调用这种函数。

（4）类的静态数据成员是指由关键字 static 修饰的类中的数据成员。 类的静态数据成员为该类的所有对象所共享。 对类的静态数据成员的访问通常使用“<类名>::<静态数据成员名>”的方式。

（5）类的静态函数成员是指由关键字 static 修饰的类中的函数成员。对类的静态函数成员的引用通常使用“<类名>::<静态函数成员调用>”的方式。类的静态函数成员没有 this 指针，通常只在其中处理类的静态数据成员值。

3．静态成员使用示例

读下面的程序，其中的自定义类 claA 中含有静态数据成员以及静态函数成员，请给出程序执行后的屏幕显示结果。

```
//program 7_6.cpp
#include <iostream.h>
class claA {
  public:
    double x,y;
    static int num;     //公有的静态数据成员，用于记录通过构造函数生成对象的个数
    claA() {
        x=0;    y=0;
        num++;      //每生成一个对象，静态数据成员 num 增加 1
    }
    claA(double x0, double y0) {
        x=x0;       y=y0;       num++;
    }
    static void staFun() {          //静态函数成员，输出静态数据成员 num 的当前值
        cout<<"current_num="<<num<<endl;
    }
};
int claA::num=0;            //必须在类外初始化静态数据成员
void main() {
    claA obj(1.2, 3.4), *p;
    cout<<"claA::num="<<claA::num<<"\t";
    claA::staFun();
```

```
    cout<<"obj.num="<<obj.num<<"\t";
    obj.staFun();
    claA A[3];          //说明具有 3 个对象的数组 A，将三次调用其构造函数
    cout<<"claA::num="<<claA::num<<"\t";
    claA::staFun();
    p=new claA(5.6, 7.8);          //生成动态对象*p，又一次调用构造函数
    cout<<"claA::num="<<claA::num<<"\t";
    claA::staFun();
    cout<<"p->num="<<p->num<<"\t";
    p->staFun();
}
```

程序执行后，屏幕显示结果为：

```
claA::num=1        current_num=1
obj.num=1          current_num=1
claA::num=4        current_num=4
claA::num=5        current_num=5
p->num=5           current_num=5
```

注意：将 claA 类中的数据成员 x、y 以及 num 都说明为 public 公有型的，是为了在主调函数 main 中可以直接存取它们而使程序简单化，否则（对非公有型的数据成员），在类外存取它们时还要设立类似于 getx()那样的公有成员函数。

7.5.2 类的常量成员

可以把类的成员说明为常量成员。

1．常量数据成员

通过关键字 const 修饰的类中的数据成员。它不同于一般的符号常量，在成员说明时不能被赋值，而只能在对象被说明时通过构造函数的成员初始化列表的方式来赋初值。一旦对象被创建，其常量数据成员的值就不允许被修改，任何类内外函数只可读取其值，但不可改变它。例如：

```
class CC{
  int i;
  const int c1;          //私有的常量数据成员 c1
```

```
public:
  const int c2;          //公有的常量数据成员 c2
  CC(int a,int b):c1(a),c2(b){
     //成员初始化列表 c1(a)、c2(b)将实参 a 与 b 的值赋给 c1 和 c2
     i=c1;
   };
…
};
```

在类外说明 CC **类对象时：**

```
CC cobj(4,7);
```

这时对象 cobj **的三个数据成员** cobj.c1，cobj.c2，cobj.i **分别有值：**

4，7，4

其中 cobj.c1 **为常量 4，不能改变；**

cobj.c2 **为常量 7，不能改变，它是公有成员，可以在类外被引用；**

cobj.i **的值为 4，它可以被类** CC **的成员函数所改变。**

注意，cobj **对象的常量数据成员** c1 **与** c2 **的初值 4 和 7，正是靠说明** CC **类对象时“** CC cobj(4,7); **”转去调用构造函数，进而通过构造函数的成员初始化列表的方式去进行赋值的。**

2．常量函数成员

类的函数成员也可以被说明为常量类型。常量类型的函数成员只有权读取相应对象（即调用者对象*this）的内容，但无权修改它们。

类的常量函数成员的说明格式如下：

<类型说明符> <函数名>（ <参数表> ）const;

要注意的是，修饰符 const 要加在函数说明的尾部（若放在首部的话，则是对函数值的修饰），它是函数类型的一部分。另外，在该函数的实现部分也要加 const 关键字。

例如：

```
class CC{
  int me;
public:
   int readme()const{return me;}      //常量成员函数
   void writeme(int i){me=i;}
   …
};
```

当一个函数成员被 const 说明后，其函数中出现的对该对象（即*this）的任何写入或修改都将被系统检查为出错。如果把函数 writeme()说明为：

```
void writeme(int i)const{me=i;}
```

也将会由系统检查出错。

常量型函数成员的说明有两个作用：

（1）当函数体较大较复杂时，由系统帮助避免对对象内容的修改。

（2）当对象被说明为常量对象时，可由系统限制，不让非常量函数成员被调用。例如：

```
void sw(const CC& con, CC& fix);
```

在这个函数中，参数 con 为常量对象，参数 fix 为一般对象，因此在函数体中：

```
fix.readme();
fix.writeme(2);
con.readme();
```

都是合法的，但

```
con.writeme(4);
```

却是非法的，因为 con 是常量对象，而函数 writeme()是非常量型的。

系统实现这一常量型函数成员禁止写入和修改对象内容的功能，是通过把常量型函数成员中隐含的 this 指针说明为：

```
const CC *const this;
```

型的，其中前面的 const 指出 this 的内容不变（总是指向该对象），后面的 const 则指出*this 不可改变。

因此，当程序员需要对对象做小的修改时，可以采用非正规的方法：

```
int readme()const{return me++;}
```

是非法的，因为 me++不被允许，但

```
int readme()const{return((CC*)this)->me++}
```

就不会出错。因为把 this 指针的类型强迫改为 CC*型，于是 me++也被允许了。

这是利用系统机制实现的小技巧，只可偶尔一用。

7.6 友元

用关键字 friend 说明友元的概念为 C++所特有，其作用是，在类的说明语句中出现：

1．位于一个函数说明语句之前，指出该函数为这个类的友元函数。

2．位于一个类名之前，指出该类是这个类的友元类。

例如：

```
class A{
    …
    friend int f(int a);
    …
    friend class B;
    …
};
```

在类 A 中说明的友元函数 f：

（1）它不是 A 的函数成员。

（2）f 的定义可以在类 A 的说明内，也可以在类外。

（3）函数 f 虽不是 A 的成员，但有权访问和调用 A 的所有私有及保护成员。

在类 A 中说明的友元类 B：

（1）它可能是与 A 无关的另外一个类。

（2）要在类外说明。

（3）B 的任一函数都有权访问和调用类 A 的所有成员，包括私有及保护成员。

面向对象程序设计主张程序的封装，数据的隐藏，不过任何事物都不是绝对的，友元的概念是 C++语言为用户提供的在局部打破这种封装和隐藏的手段，好像一个家庭的财务，总是要通过防盗门，门锁，保险柜等措施不让外人接触。但在特殊情况下，例如全家出游，又需检查煤气，水，电情况，就不得不把钥匙交给可依赖的邻居。这位邻居就是友元。不过友元的概念有点类似于结构程序设计语言中的 goto 语句，虽有必要设置，但不宜多用。

注意，在类 A 的友元函数和类 A 的成员函数中（定义处）都可处理与使用类 A 的私有成员，但两种函数的最大使用区别是：对友元函数来说，参加运算的所有运算分量（如类对象）必须显式地列在友元函数的参数表中（由于友元函数中没有 this 指针，从而没有“当前调用者对象”的概念）；另外，调用友元函数时根本不通过类对象（因为它并非类的成员）。但对成员函数来说，它总以当前调用者对象（*this）作为该成员函数的隐式第一运算分量。若所定义的运算多于一个运算对象时，才将其余运算对象显式地列在该成员函数的参数表中。另外，调用类成员函数时必须通过类对象。

关于友元说明还应注意的是，它有如下三特点：“单方向”、“不继承”、“不

传递”。

“单方向” — 若 ClaA 具有友元类 ClaB，并不意味着 ClaB 也具有友元类 ClaA（“非相互”）。

“不继承” — 若 ClaA 具有友元类 ClaB（即，ClaB 是 ClaA 的友元类），又 ClaB 具有子代类 ClaC（即，ClaC 是 ClaB 的派生类），并不意味从 ClaC 可以直接存取 ClaA 的私有成员。关于继承与派生请参看后面章节的内容。

“不传递” — 若 ClaA 具有友元类 ClaB（即，ClaB 是 ClaA 的友元类），又 ClaB 具有友元类 ClaC（即，ClaC 是 ClaB 的友元类），并不意味从 ClaC 可以直接存取 ClaA 的私有成员。

下述程序自定义一个示意性的复数类型 complex，其中说明了两个友元函数，使用该类可以完成复数的加法以及对复数的输出。并编制了主函数，说明 complex 类对象，对定义的各函数进行调用。

```
//program 7_7.cpp
#include<iostream.h>
class complex {                          //自定义的复数类型 complex
      double  real;                      //复数实部
      double  imag;                      //复数虚部
public:
      complex();                         //无参构造函数
      complex(double r, double i);       //2 参构造函数
      friend complex addCom(complex c1, complex c2);
            //友元函数，实现复数加法 c1+c2（二参数对象相加），返回 complex 类对象
            //注意友元函数中没有 this 指针，从而没有“当前调用者对象”的概念
      friend void outCom (complex c);
            //友元函数，输出 complex 类对象 c 的有关数据（各分量）
};
complex::complex(){              //无参构造函数，将 real 及 imag 置为 0
      real=0;       imag=0;
}
complex::complex(double r, double i) {              //2 参构造函数
      real=r;       imag=i;
}
complex addCom(complex c1, complex c2) {    //友元函数，实现复数加法 c1+c2
      complex  c;                        //局部于函数的临时对象
```

```
    c.real=c1.real+c2.real;
    c.imag=c1.imag+c2.imag;
    return  c;                          //返回 complex 类对象
}
void outCom (complex com) {         //友元函数，输出 complex 类对象 c
    cout<<"("<<com.real<<", "<<com.imag<<")";
}
void main(void){
    complex  c1(1,2), c2(3,4), res;   //说明类对象 c1，c2，res
    res=addCom(c1,  c2);
        //调用友元函数 addCom 实现复数加法，实参为类对象 c1 与 c2，且返回类对象
        //注意，调用友元函数时根本不通过类对象（因为它并非类的成员）
    outCom(c1);    //调用友元函数 outCom 输出类对象 c1
    cout<<" + ";
    outCom(c2);    //调用 outCom 输出类对象 c2
    cout<<" = ";
    outCom(res);    //输出类对象 res
    cout<<endl;
}
```

程序执行后，屏幕显示结果为:

```
(1, 2) + (3, 4) = (4, 6)
```

实际上，实现上述示意性复数类型 complex 时，最常用的当然还是类成员函数，在类成员函数内（定义处）可处理与使用本 complex 类的私有成员，而且总以当前调用者对象（*this）作为该成员函数的隐式第一运算分量。下面仅给出此种实现方式的类定义“ 构架 ”, 具体各成员函数的实现以及对它们的使用（调用）请读者作为一个练习来完成。

```
class complex{
    double  real;                       //复数实部
    double  imag;                       //复数虚部
public:
    complex();                          //无参构造函数
    complex(double r, double  i);       //2 参构造函数
    complex addCom(complex c2);         //调用者对象与对象 c2 相加，返回 complex 类对象
    void  outCom  ();                   //输出调用者对象的有关数据(各分量)
```

```
};
```

7.7 类之间的关系

封装的实现主要是通过把若干数据和函数成员组织到一个类中完成的，在一个类或一个对象中，成员之间的一系列关系已在前面讨论过。

问题的另一方面就是被封装起来的各个类之间是如何发生联系的，C++语言为类和对象之间的联系提供了许多方式，主要有：

1. 一个类的对象作为另一个类的成员；
2. 一个类的成员函数作为另一个类的友元；
3. 一个类定义在另一个类的说明中，即类的嵌套；
4. 一个类作为另一个类的派生类。

有关友元的概念与使用已经在上一节讨论过。至于一个类作为另一个类的派生类，将在下一章进行专门的讲解与讨论。下面说明另外两种情况。

1. 类的对象成员

自定义类中的数据成员可以是另一个类的类对象 — 靠类定义中的对象说明来指定这样的数据成员，意味着在一个“大对象”中包含着属于另外一个类的“小对象”。注意如下类 line 的定义，其数据成员 start 与 end 就为另一个称为 pixel 的类对象。

类定义中的对象说明与居于在类外的对象说明语句不同，后者意味着对象的创建；类中的对象成员说明并不直接与对象的创建和初始化相联系，它要等所在的类的对象被创建时（通过构造函数）一同被创建。这就是下面要说的为什么负责创建类 line 对象的构造函数还要负责一同创建出类 line 所包含的对象成员 start 与 end 的真正原因。

```
class pixel {
        int x,y;
public:
        pixel(){x=0;y=0;}
        pixel(int x0, int y0) {
                x=x0;    y=y0;
        }
};
class line{
        pixel start,end;
```

```
        int color;
public:
        line(int sx,int sy,int ex,int ey,int col):
          start(sx,sy),end(ex,ey){
              color=col;
          };
};
void main(){
        line line1(20,20,100,20,2),    line2(20,20,20,100,1);
}
```

在这个例中，类 line 有两个对象成员 start 和 end，它们是 pixel 类的对象。

当在程序中创建 line 类的对象时，必须同时创建 pixel 类的两个对象：

line line1（20，20，100，20，2），line2（20，20，20，100，1）；

首先用实参 20，20 对 line1.start 进行初始化，用实参 100，20 对 line1.end 初始化，再进行 line1.color=2，完成 line1 的初始化，其次进行 line2 的初始化。

请注意，类 line 的构造函数的定义，需在函数体之前通过成员初始化符表把参数传到对象成员的构造函数中去。

成员初始化符表：其格式为：

：〈成员初始化符〉，…，〈成员初始化符〉

成员初始化符：〈对象成员〉（〈初值表〉）

例如：start（sx，sy），end（ex，ey）。

即是说，若某个类中含有对象成员，则该类的构造函数就应包含一个初始化符表，用于负责对类中所含各对象成员进行初始化。

在定义（生成）一个含有对象成员的类对象时，它的构造函数被系统调用，这时将首先按照初始化符表来依次执行各对象成员的构造函数，完成各对象成员的初始化工作，而后执行本类的构造函数体。析构函数的调用顺序恰好与之相反。

如果对象成员的类未定义构造函数，成员初始化符可缺省，这时系统调用缺省构造函数创建成员对象。

类的对象成员意味着一个类的对象包含了另一个类的对象。如类 line 的一个对象，包含了两个 pixel 类的对象。这是类之间最简单的对象包含关系。

实现不同类对象的包含，还可以利用指向类对象的指针。

不直接以对象作为类的成员，而是以指向对象的指针作为类的成员，也是实现包含关系的常用方法。在这种情形下，由于对象指针是类的一般数据成员，

所以不必如对象成员那样，在构造函数说明中加入成员初始化符表，而是直接在构造函数内进行初始化即可。

2．类的嵌套

既然类中可以定义用户定义的数据类型，那么，允许在类说明中定义类也是自然的。

一个类的说明包含在另一个类说明中，即为类的嵌套。多数版本的 C++语言允许类的嵌套。读者注意，它与类的对象成员不同。例如：

```
class CC{
  class C1{…};
public:
  class C2{…};
  C1 f1(C2);
  …
};
```

其中类 C1 是类 CC 的私有嵌套类，类 CC 外的函数中不能使用它；类 C2 是类 CC 的公有嵌套类，可以在 CC 类外使用，但类名应为 CC::C2。即是说，在类 CC 之外，如在 main 主函数中：

```
C1 a,b;
C2 a,b;
CC::C1 a,b;
```

都是非法的，但

```
CC::C2 a,b;
```

是合法的，它说明了两个嵌套类 C2 的对象。

嵌套类的使用并不方便，不宜多用。

7.8 自定义类中的运算符重载

C++允许重新定义某些运算符的作用及其实现（功能），可以给运算符扩充添加新用法（“老用法”依然可行）。即是说，可以“借用”老算符，对自定义类对象实现自定义的新功能。

7.8.1 以两种方式对运算符进行重载

在第五章对运算符重载进行过介绍，但那时还无法讲解与自定义类相关的运算符重载的概念及其使用方法。在自定义类中可以通过两种方式对运算符进行重载：按照类成员方式或按照友元方式。

运算符重载的定义是一个函数定义过程（其函数名处为：operator <运算符>）。例如，在自定义类 set 中，通过友元（函数）方式定义“&”运算符，用于处理集合元素的包含关系；并通过类成员（函数）方式定义“+”运算符，用于实现二集合的求并运算：

```
class set {
        int elems [maxcard];            //集合各元素放于私有数据成员 elems 中
        int card;                       //集合中的实际元素个数 card
public:
        ...
        friend bool operator & (int, set);
        set operator + (set S2);
        ...
};
```

上述定义的友元函数为“operator &”，用来重载运算符&，扩充其功能，使之能处理“<int> & <Set>”式样的运算，用于判断第一分量的<int>集合元素是否包含于第二分量的<Set>集合之中。如，3 & s1，用来判断集合元素 3 是否包含于集合 s1 之中。

还以类成员函数方式重载了运算符“+”:求出当前调用者对象也即集合 *this 与 S2 的并集合后返回。函数名为“operator +”，用来重载运算符“+”，扩充其功能，使之能处理“<Set> + <Set>”式样的运算，用于求出两个集合的并。如，s1+s2。

当以类的友元函数方式来重载运算符(也称为友元运算符)时，具有如下特征：

（1）友元函数内（定义处）可处理与使用本类的私有成员；

（2）所有运算分量必须显式地列在本友元函数的参数表中（由于友元函数中没有 this 指针），而且这些参数类型中至少要有一个应该是说明该友元的类类型或是对该类的引用；

当以类的公有成员函数方式来重载运算符（也称为类运算符）时，具有如下特征：

（1）类成员函数内（定义处）可处理与使用本类的私有成员；

（2）总以当前调用者对象（*this）作为该成员函数的隐式第一运算分量，

若所定义的运算多于一个运算对象时，才将其余运算对象显式地列在该成员函数的参数表中；

一般地说，单目运算符重载常选用成员函数方式，而双目运算符重载常选用友元函数方式。但不管选用哪种重载方式，对重载算符的使用方法则是相同的。

注意，被用户重定义（重载）的运算符，其优先级、运算顺序（结合性）以及运算分量个数都必须与系统中的本原运算符相一致，而且不可自创新的运算符。

1．友员方式重载示例

自定义一个称为 point 的类，其对象表示平面上的一个点（x，y），并通过友员方式对该类重载二目运算符“+”和“^”，用来求出两个对象的和以及两个对象（平面点）的距离。各运算符的使用含义（运算结果）如下所示：

(1.2，-3.5) + (-1.5，6) = (-0.3，2.5)；

(1.2，-3.5) ^ (-1.5，6) = 9.87623。

并编制主函数，说明类对象，而后通过类对象实现所定义的相关运算（以验证其正确性）。

```
//program 7_8.cpp
#include <iostream.h>
#include <math.h>
class point {
    double x,y;
public:
    point (double x0=0, double y0=0) {x=x0;    y=y0;}
    friend point operator + (point pt1, point pt2);
    friend double operator ^ (point pt1, point pt2);
    void display();
};
point operator + (point pt1, point pt2) {
    //二目运算符“+”，求出两个对象的和
    point temp;
    temp.x=pt1.x+pt2.x;
    temp.y=pt1.y+pt2.y;
    return temp;
}
```

```
double operator ^ (point pt1, point pt2) {
    //二目运算符“^”，求出两个对象（平面点）的距离
    double d1, d2, d;
    d1=pt1.x-pt2.x;
    d2=pt1.y-pt2.y;
    d=sqrt(d1*d1+d2*d2);
    return (d);
}
void point::display () {
    cout <<"( "<<x<<", "<<y<<" )"<<endl;
}
void main() {
    point s0, s1(1.2, -3.5), s2(-1.5, 6);
    cout<<"s0=";      s0.display();
    cout<<"s1=";      s1.display();
    cout<<"s2=";      s2.display();
    s0=s1+s2;              //二对象的“+”运算，将调用“operator +”函数
    cout<<"s0=s1+s2=";   s0.display();
    cout<<"s1^s2="<<(s1^s2)<<endl;   //二对象的“^”运算，将调用“operator ^”函数
}
```

程序执行后，屏幕显示结果为：

```
s0=( 0, 0 )
s1=( 1.2, -3.5 )
s2=( -1.5, 6 )
s0=s1+s2=( -0.3, 2.5 )
s1^s2=9.87623
```

2．类成员方式重载示例

可以把上一程序中通过友员方式重载的运算符，改写为以类的公有成员函数方式进行重载，以实现（解决）相同的问题。下面仅给出此种实现方式的类定义“构架”，具体各成员函数的实现以及对它们的使用（调用）请读者作为一个练习来完成。

```
class point {
    double x,y;
public:
```

```
    point (double x0=0, double y0=0){x=x0; y=y0;}
    point operator + (point pt2);
    double operator ^ (point pt2);
    void display();
};
```

7.8.2 利用运算符重载实现集合 set 类型

虽然我们在前面已经定义了一个 set 类，但使用这个类的函数成员不如使用集合运算符方便，例如：

```
s.Intersect(&s1,&s3)
```

不如表示为：

```
s3=s*s1
```

而 s.Member(i) （确定整数 i 是否在 s 中）

不如表示为：

```
i&s
```

下面我们利用运算符重载（overloading）改造原来的集合类型定义，达到使集合的运算符合人们习惯的目的。

实现时通过“借用”一批老运算符(如，*、+、-、&、>=、>等)，来表示集合 set 类对象的交、并、差、元素属于、包含、真包含等运算，各算符的具体运算含义必须首先在 set 类中通过运算符重载函数由用户进行说明或自定义（即是说，系统并不自动为每一个自定义类预定义这些算符功能），而后就可按照所定义的含义对它们进行使用。

本程序使用友元函数方式对各运算符进行重载。作为练习，读者可试着将它们改造为类成员函数的重载方式。

```
//program 7_12.cpp
#include <iostream.h>
const int maxcard=20;
class set {
    int elems [maxcard];
    int card;
public:
    //除 set 与 print 仍说明为本类的公有成员函数外，其余 9 个运算符重载函数均被说明
    //为本 set 类的友元函数(它们并非 set 类的成员函数，而是在类外说明的全局性函数)
```

```
    set (void){card=0;}                    //构造函数
    void  print();                         //成员函数，在类外说明
    friend bool operator & (int, set);     //  &：判断某元素是否为某集合的成员
    friend bool operator == (set, set);    //  ==：判断两个集合是否相同
    friend bool operator != (set, set);    //  !=：判断两个集合是否不相同
    friend set operator + (set, int);      //  +：将某元素加入到某集合中
    friend set operator + (set, set);      //  +：求两个集合的并集合
    friend set operator - (set, int);      //  -：将某元素从某集合中删去
    friend set operator * (set, set);      //  *：求两个集合的交集合
    friend bool operator < (set, set);     //  <：集合 1 是否真包含于集合 2 之中
    friend bool operator <= (set, set);    //  <=：集合 1 是否包含于集合 2 之中
};
/* 重载运算符 &：判断元素 elem 是否为集合 set 的成员。注意：友元函数并非类成员函数，
函数名前不加'set::'类限定符；由于不是类成员函数，也就不存在当前调用者对象，所以集
合对象 s 必须作为参数列出来！判断形参 elem 是否等于形参对象 s 的 elems[0]->elems[card-1]
中的某一元素，是则返 true，否则返 false。*/
bool operator & (int elem, set s)  {
    for(int i=0;i<s.card;i++)
        if (s.elems[i]==elem)
            return true;
     return  false;
}
bool operator == (set s1, set s2) {        //判断集合 S1 是否与集合 S2 相同
    if(s1.card!=s2.card)
        return  false;                     //元素个数不同，则二集合不相同
    for(int i=0;i<s1.card;i++)
        if(!(s1.elems[i]&s2))
            return false;                  //发现有一个元素不相等时，则返 false。
    return  true;                          //全相等时，返 true
}
bool operator != (set s1, set s2) {        //判断两个集合 S1 与 S2 是否不相同
    return !(s1==s2);                      //借用'=='来处理'!='
}
set operator + (set s, int elem){          //将元素 elem 加入到集合 s 中
```

```
    set res=s;
    if (s.card<maxcard) {                    //数组不超界时，方可加入
        if(!(elem&s))       //若 elem 不为 s 集合的元素时，加入 elem(已有时，不再加)
            res.elems[res.card++]=elem;
    }
    return res;
}
set operator + (set s1, set s2) {            //求两个集合 S1 与 S2 的并集合
    set  res=s1;                             //先将 s1 拷贝到 res 集合
    for(int i=0;i<s2.card;i++)
        res=res+s2.elems[i];                 //再将 S2 的诸元素加入到 res 集合
    return res;                              //返回 res 集合
}
set operator - (set s, int elem) {           //将元素 elem 从集合 s 中删去
    set res=s;
    if(!(elem&s)) return res;                //若元素 elem 不在集合 s 中，则仍返回 s
    for(int i=0;i<s.card;i++)
        if  (s.elems[i]==elem)
          //当从 s 集合的数组 elems 中找到形参 elem 后，将该数组元素的
          //所有"后继"元素统统"前移"一个位置，即不可出现"空洞"。
    for(;i<s.card-1;i++)
        res.elems[i]=res.elems[i+1];
    --res.card;                  //删除一元素后，记录集合元素数的 card 变量值减 1
    return res;
}
//重载运算符 *：求两个集合 S1 与 S2 的交集合。
//实现方法为：用 S1 集合的诸 elems[i]，与 s2 集合的诸 elems[j]一一作比较，将其中的
//相同者(共有元素)送入 res 对象的 elems 数组的从 0 下标开始的各单元中，最后返回 res。
set operator * (set s1, set s2){
    set res;
    for(int i=0;i<s1.card;i++)
        for(int  j=0;j<s2.card;j++)
          if(s1.elems[i]==s2.elems[j]){
              res.elems[res.card++]=s1.elems[i];
```

```
                    break;
                }
        return res;
}
//重载运算符 < :  集合 s1 是否真包含于集合 s2 之中。
//当集合 s1 比 s2 的元素个数少，而且 s1 包含于 s2 之中时，返回 true。
bool operator < (set s1, set s2) {
        return ((s1.card<s2.card) &&(s1<=s2));
}
bool operator <= (set s1, set s2){          //集合 s1 是否包含于集合 s2 之中
        if(s1.card>s2.card) return false;   //当集合 s1 比 s2 的元素个数多时，返回 false
        for(int i=0;i<s1.card;i++)
            if(!(s1.elems[i]&s2))           //发现某一个 s1 的元素不属于 s2 时，返回 false
                return false;
        return  true;                       //不出现上述二情况时，返回 true
}
void set::print () {                        //成员函数 print，在类外说明时要加'set::'类限定符
        cout << "{";
        for(int i=0;i<card-1;i++)           //显示出当前调用者对象的 elems 数组中各元素
           cout <<elems[i]<< ",";
        cout <<elems[card-1];
        cout << "}"<<endl;
}
void main() {
        set s,s1,s2,s3,s4;
        for (int i=0;i<10;i++){
            s=s+i;      s1=s1+2*i;      s2=s2+3*i;
        }
        cout<<"s=";     s.print();
        cout<<"s1=";    s1.print();
        cout<<"s2=";    s2.print();
        for (i=0;i<5;i++){
            s=s-i;      s1=s1-i;        s2=s2-i;
        }
```

```
    cout<<"After RmvElem(0->4), s=";          s.print();
    cout<<"After RmvElem(0->4), s1=";         s1.print();
    cout<<"After RmvElem(0->4), s2=";         s2.print();
    s3=s*s1;      s4=s+s2;
    cout<<"s3=s*s1=";         s3.print();
    cout<<"s4=s+s2=";         s4.print();
    if(s3==s4)   cout<<"SET s3=s4 "<<endl;
        else cout<<"SET s3!=s4 "<<endl;
    if(s3==s3) cout<<"SET s3=s3 "<<endl;
        else cout<<"SET s3!=s3 "<<endl;
    if(s4<=s3)   cout<<"SET s3 contains s4 "<<endl;
        else cout<<"SET s3 do not contains s4 "<<endl;
    if(s2<=s4) cout<<"SET s4 contains s2 "<<endl;
        else cout<<"SET s4 do not contains s2 "<<endl;
}
```

程序执行后的显示结果如下：

```
s={0,1,2,3,4,5,6,7,8,9}
s1={0,2,4,6,8,10,12,14,16,18}
s2={0,3,6,9,12,15,18,21,24,27}
After RmvElem(0->4), s={5,6,7,8,9}
After RmvElem(0->4), s1={6,8,10,12,14,16,18}
After RmvElem(0->4), s2={6,9,12,15,18,21,24,27}
s3=s*s1={6,8}
s4=s+s2={5,6,7,8,9,12,15,18,21,24,27}
SET s3!=s4
SET s3=s3
SET s3 do not contains s4
SET s4 contains s2
```

在 main( )**中对于** set **类型的变量（对象）作了初步的简明操作。**

在我们的程序中，为运算符&**定义一种新的意义：作为集合的“属于”运算的运算符。其使用方法可从程序中找到。**

这里，elem & s **是一个关系表达式，左运算分量是** int **型变量或常量，右运算分量** s **是集合类型的对象，运算结果为** bool **类型，即指出真或假。**

当然，程序员也可以选择其它运算符，例如用%**、**@**、**^**等来代替**&**，一般应**

符合人们的习惯。例如后面集合的并用“+”，集合的交用“*”，集合的包含关系用“＞”或“＞＝”，在没有符号“∪”,“∩”,“⊇”,“⊆”可供选择的条件下，用“+”,“*”,“＞”,“＞＝”来表示它们是最恰当的。

可以看出，如果把“operator&”视为函数名，就正好与原来定义一般函数时它们的原型以及函数定义的方式相吻合。事实上，C++语言正是通过这种形式，把运算符的概念与函数的概念相联系的。所以，运算符重载函数的使用，可以通过两种方式：

```
elem & s
```

这是一个 bool 型的关系表达式；

```
operator & (elem, s)
```

是与上面的关系表达式等价的函数调用形式的表达式。在做了上面的定义之后，这两种形式是等价的。当然，人们在使用中通常会选择方便简明的第一种形式。

7.9 拷贝构造函数

7.9.1 概述

拷贝构造函数是一种特殊的构造函数，具有一般构造函数的所有特性。拷贝构造函数只含有一个形参，而且其形参为本类对象的引用。即是说，拷贝构造函数的原型为：“<类名>（<类名>&）;”。

拷贝构造函数的作用是使用一个已存在对象 — 由拷贝构造函数的参数指定的对象（的各成员当前值）去初始化另一个相同的新对象（尚不存在，正在创建）。

某些特殊情况下，用户必须给出显式的拷贝构造函数以实现用户指定的‘深拷贝”功能。如果用户没有声明类的拷贝构造函数（即没给出显式的拷贝构造函数）时，系统会自动生成一个隐含的（缺省的）拷贝构造函数，它只进行对象间的“原样拷贝”(即位对位的拷贝，也称为“浅拷贝”)。

在下述 3 种情况下，系统都将自动地去调用对象所属类的拷贝构造函数(若无显式的，则调用系统隐含的，从而只进行“浅拷贝”处理)：

（1）当使用如下二方式之一的说明语句，用已存在对象（的各成员当前值）来创建一个相同的新对象时：

```
<类名> <对象名 2>（<对象名 1>）;
<类名> <对象名 2> = <对象名 1>;
```

即是说，在说明新对象<对象名 2>时，准备用已存在对象<对象名 1>来对它进行初始化。

（2）若对象作为函数的赋值参数，在调用函数时，当刚进入被调函数处首

先要进行实参和形参的结合，此时会自动调用拷贝构造函数，以完成从已存在的实参对象（的各成员当前值）来创建一个相同的形参新对象（尚不存在，正在创建）。

（3）若函数的返回值是类的对象，在执行被调函数的返回语句后（也即在函数调用完成返回时），系统会自动创建一个与返回值相同的临时新对象（放入调用者的栈空间处），此时，系统也要去调用拷贝构造函数（用当前的已存在对象即函数返回值，去创建出那一临时新对象）。

一般规定，所创建的临时对象，仅在创建它们的外部表达式范围内有效，表达式结束时，系统将调用析构函数去“销毁”该临时对象。

调用拷贝构造函数的 3 种情况之程序“构架”示例：

```
#include<iostream.h>
class  point  {                          //自定义类 point
       int x,y;
   public:
       point(point& pt) {                //拷贝构造函数
            cout<<"Enter  copy-constructor!"<<endl;
            …
       }
       …
};
void func1(point pt) {                   //point 型的类对象作为函数的赋值参数（情况 2）
       …
}
point func2() {                          //函数返回结果为 point 型的类对象（情况 3）
       point p1(606,808);
       …
       return  p1;                       //返回 point 型的类对象
}
void main() {
       point  pt1(123,456);
            //说明对象 pt1，将调用普通构造函数（假设类中具有相应的二参普通构造函数）
       point  pt2=pt1;                   //与使用“ point pt2(pt1); ”效果相同（情况 1）
       func1(pt2);                       //情况 2，调用 func1 进行实参对象 pt2 与形参 pt 结合时
       pt2=func2();                      //情况 3，当调用 func2 结束返回一个对象时
}
```

注意，对本“构架”程序来说，若不提供显式拷贝构造函数的话，系统将调用隐含的拷贝构造函数进行“浅拷贝”处理。

7.9.2 显式拷贝构造函数的使用

在某些情况下，必须给出显式的拷贝构造函数。

假设在某类的普通构造函数中分配并使用了某些系统资源（例如通过 new 分配并使用了系统的堆空间），而且在该类的析构函数中释放了这些资源（例如通过 delete 来释放所分配的堆空间）。比如像下面的 person 类：

```
class person {
```

```
    char * pName;
  public:
    person (char * pN){                          //普通构造函数
        pName=new char[strlen(pN)+1];            //动态申请了系统资源（堆空间）
        strcpy(pName,pN);                        //将实参 pN 串拷贝到新分配的动态空间中
    }
    ~person (){                                  //析构函数
        delete pName;                            //释放（归还）系统资源（堆空间）
    }
};
```

对上述假设的类，当遇到需要执行拷贝构造函数的那 3 种情况时，如果用户没有提供类的显式拷贝构造函数，而让系统去执行缺省的拷贝构造函数只进行对象间的“原样拷贝”（也称“浅拷贝”）的话，会出现两个对象拥有同一个资源即拥有同一块系统堆空间的情况（因为进行了“原样拷贝”，从而使得两个不同对象的 pName 指针成员值即地址值完全相同，意味着两者指向了同一块系统堆空间）。当对象析构时（两个对象各要被析构一次），则会遇到同一资源被释放两次（两次 delete 释放同一堆空间）的错误。

为了避免出现上述的错误，必须给出显式的拷贝构造函数，以实现用户指定的“深拷贝”功能。即是说，在拷贝构造函数中，首先要动态申请它自己的 pName 空间（具有了两份不相冲突的资源），并向自己的 pName 空间拷入原对象的内容（所谓的“深拷贝”）。这样在释放两份不相冲突的堆空间时将不再出错。注意，“浅拷贝”即位对位的拷贝处理时，只拷贝了 pName 指针值（使两个对象的 pName 指向同一块内存动态空间），但不再自动申请另一块 pName 空间，更谈不上向新的 pName 空间拷入内容了。即是说，此时需要“添加”类似于如下样式的拷贝构造函数：

```
person (person & p) {                                    //拷贝构造函数
    cout<<"Copying "<<p.pName<<" into its owm block"<<endl;
    pName=new char[strlen(p.pName)+1];                   //动态申请它自己的 pName 空间
    strcpy(pName, p.pName);                  //向自己的 pName 空间拷入内容（“深拷贝”）
}
```

7.10 结构与联合

已在前面的章节中简单地介绍过结构的说明及其使用。当时只是将它作为一种存放数据的实体来使用来看待的。实际上，C++中的结构同样能够像类那样地实现封装（而在 C 中则不能）。即是说，除数据成员外，也可在结构中定义函数成员；结构也可以有它自己的构造函数、析构函数以及 this 指针等等。可以看出，类和结构在一定意义上是等价的。

结构与类的区别是：在缺省情况下，结构的成员是公有的（隐含为 public

属性），而类的成员是私有的（隐含为 private 属性）。大多数有经验的程序员通过使用结构来“封装”若干数据的集合，例如，把有关窗口的主要数据组成一个结构，或把有关图形文件的数据组成一个结构等等。而当既描述数据成员又刻画其操作（函数成员）时使用类 class。另外，在讨论结构的时候，我们往往用结构变量这个词；但在讨论类时，我们用对象这个词。

联合在 C++语言中除了类似于类和结构之外，受到了更多的限制，主要是其所有数据成员重叠，都从同一位置开始存储。通常也只在要描述一批可重叠存放的数据成员时才使用联合 union，而且也大多用在只有数据成员的情形。

定义一个联合 union（类型）的最常用格式如下：

```
union <自定义联合类型名>{
    <联合成员说明>;
};
```

可按如下方式来说明联合类型的变量：

```
<自定义联合类型名> <联合变量名 1>, ..., <联合变量名 n>;
```

按如下方式来使用联合变量的变量分量：

```
<联合变量名>.<成员名>
```

例如：

```
union myUnionType {     //定义一个联合类型 myUnionType, 它具有五个数据成员
        char    ch;
        int     i;
        long    l;
        float   f;
        double  d;
};
myUnionType u1,u2,u3;       //说明了 3 个联合类型的变量
u1.i = 6;                   //为联合类型的变量 u1 的数据成员 i 赋值
u2.d = 5.327;               //为联合类型的变量 u2 的数据成员 d 赋值
```

对联合的使用有如下几点注意：

（1）由于所有数据成员重叠，都从同一位置开始存储，所以联合 u1 的起始地址&u1 与其各分量的起始地址如&u1.i 及&u1.d 等都是相同的。

（2）联合变量所占空间的大小等同于其中的最大（最长）分量所占空间的大小。如本例中的 sizeof(u1)或 sizeof(myUnionType)、以及 sizeof(double)和 sizeof(u1.d)的值都相同，均为 8。

（3）由于所有数据成员重叠，因此在任一特定时刻，联合中存储的只是某

一数据成员的值，此时其他成员一般是无意义的。

总之，C++语言中的类和对象概念使得结构和联合在程序设计中的重要性变轻，在作为 C++语言的衍生语言 Java 语言中，干脆取消了结构和联合这两种数据类型。

思考题：

1. 在 C 十十中如何进行数据封装和数据隐藏？
2. 什么是类？为什么会有类的概念？
3. 类的构成元素有哪些？各自的作用是什么？
4. 类是通过怎样的机制完成信息的封装的？
5. 什么是对象？对象和类在概念上的区别是什么？
6. 什么是构造函数？在类的初始化过程中有什么作用？怎样定义？
7. 什么是析构函数？在类的结束过程中有什么作用？怎样定义？
8. 构造函数和析构函数是必须的吗？
9. 什么是类的共有成员和私有成员？说明公有成员和私有成员的意义。
10. 怎样访问类的私有成员和公有成员？
11. 你认为类将数据分为公有成员和私有成员的好处是什么？
12. 什么是静态成员？它有什么性质？在类中的作用是什么？
13. 什么是友元？它有什么作用？怎样定义？
14. this 指针的含义是什么？它在 C++中的作用是什么？
15. 什么是结构和联合？比较它们和类的异同点。
16. 什么是类的嵌套？它的性质是什么？

练习题：

1. 队列是一种程序设计中常用的数据结构。它的结构和栈类似。但队列中的数据是先进先出，仿照栈的定义，定义一个新的队列类，其函数成员如下：

   queue（void）构造函数

   ～queue（void）析构函数

   bool Empty（void）清空队列

   bool full（void）判断队列是否已经满

   void add（int a）向队列中加入一个数据

   float delete（void）从队列中取出一个数据

2. 仔细检查下列代码，指出错误的地方。

```
#include＜iostream.h＞
classA      {
      public：
                  void A(int I=0){m=i;};
                  void Show(){cout<<m;};
                  void ~A(){};
      private:
                  int     m;
};
int     main ()
{
      A a(5);
      a.m += 10;
      a.Show();
      Return0;
}
```

3．指出下面 this 指针哪些使用有错误。

```
class X{
  int i;
  public：
    X () { i = 0;} ;
    void func1 () {
      X x;
      this = &x;
      *this.i = 2;
      X * This;
      This = this;
      This = &x;
    };
void main () {
  X x;
  this->x = 10;
}
```

4．设计一个学生类（CStudent），其数据成员是学生的注册号、姓名、数学、外语、计算

机课程的成绩。然后对于学生对象的数组进行输入，在设计一个函数，以注册号 n 为参数，在数组中搜索注册号是 n 的学生，并返回该生的全部信息。

5．构造一个复数类，将复数的表示和复数的几种基本运算包含进去。要求包含的复数运算至少有复数的加、减、乘、除和取模。创建完这个类之后，请编一个程序来进行各项的验证。

6．设计一个简单的计算器，要求：

（1） 从用户处读入算式。

（2） 可以进行加、减、乘、除运算。

（3） 运算要有优先级。

（4） 用户可以按任何顺序输入。

（5） 不限定用户输入的计算式的长度

（6） 有排错功能，当用户输入错误的计算式时提示用户。

例如如果用户输入：3 + 4 * 5 – 7，计算结果应为 16。

（提示：可以使用本节所定义的栈类来帮助进行处理）

7．通过上述的习题我们可以看到类是一个强有力的封装工具，试将第 6 题的计算器按类的要求进行封装，形成一个完整的能进行四则运算的实用类。使它能被分发给他人使用。（自行定义类结构）不同的同学完成之后互相交换各自设计的类，相互进行验证和使用。

（使一个人完成的功能对其它人是透明的，这一点是 C++的一个重要的目标，体会在交换过程中 C++是怎样保证这一点的。）

第八章　继承与派生

C++程序用不同的类定义来表示一组数据及对这些数据的操作，往往在不同的类之间有某种关系，除了上一章介绍的包含关系和友元关系之外，更多的是继承与派生关系。

例如，一个公司或工厂的计算机管理系统，与设备有关的数据，如设备登记号、设备购入时间、设备价值等数据及若干操作等可以构成一个类。

交通工具是设备中的一类，它除了作为设备具有一般性之外，又会有一些作为交通工具的特定数据及操作，如它应有由公安机关发给的牌照号等。

又如汽车，它是交通工具的一种。司机姓名、牌号，可能是它特有的数据。而货车、轿车、大客车又是汽车中的不同集合，与它们分别相关的又会有不少数据，如轿车的使用人姓名、货车的吨位、客车的载人数等等，都是它们特有的数据项。

把与设备，交通工具，汽车，轿车，货车，客车相关的数据及操作定义为互不相关的独立的类是不科学的，同时，也会造成许多重复内容，例如，所有这些类，都包括同一个数据成员：设备登记号、购入时间等等。

不同类的定义应反映出类之间的相关关系，反映出上面例子中的层次关系。C++语言中提供了类定义的派生和继承的功能，很好地解决了上面提出的问题。

两个类之间的继承关系，若类 A 是类 B 的基类，则类 B 是类 A 的派生类。我们首先从下面的实例中学习如何建立类与类之间的继承关系。

8.1　公司雇员档案的管理

公司中的雇员档案的管理是整个公司的管理系统的一个组成部分，其雇员的档案数据内容一般根据雇员在公司中的不同职责和职位而有所区别。除了一般雇员之外，可能还有管理人员、工程师和高级主管，这些人员也是雇员，但他们又有其特殊性，在数据库中又必须保存某些特定的信息，例如管理人员有级别信息，工程师有学位、专业信息等等。因此，在管理软件中所设计的类应反映其层次关系和特殊性。

当要设计一个公司雇员档案管理的应用程序时，首先要对问题进行分析，整理抽象出所要处理的数据信息以及这些数据间的相互关系。例如，由于对不同的公司雇员其档案保存的内容也即数据信息是不同的，所以可考虑自定义多个类（类型），使每一种类（类型）对应于一类公司雇员。但注意到，这些类（类型）之间存在许多“共性”的数据，从而可利用 C++对类的派生与继承关系，来有效地处理这些既有“共性”又有各自“特性”数据的类（类型）。

假设公司雇员分为：雇员（employee）、经理（manager）、工程师（engineer）、高级主管（director）。而且假定只关心这几类雇员各自的如下一些数据：

employee（雇员）类：　　姓名、年龄、工资；
manager（经理）类：　　姓名、年龄、工资、行政级别；
engineer（工程师）类：　姓名、年龄、工资、专业、学位；
director（高级主管）类：姓名、年龄、工资、专业、学位、职务。

可看出，若利用 C++对类的派生与继承关系，把 employee（雇员）类定义为所有类的基类，由 employee（雇员）类出发（作为基类），派生出 manager 类（经理类，仅比基类多出一个“特性成员”— 行政级别）以及 engineer 类（工程师类，要比基类多出两个“特性成员”— 专业、学位）；而后又由 engineer 类（工程师类）出发（作为基类），派生出 director 类（高级主管类，仅比它自己的基类即 engineer 类多出一个“特性成员”— 职务）。根据上述的分析归纳与整理，进而可形成本示例中的如下几个相互有关的类（类型）。而后本示例还编制出了主函数，对上述这些类（类型）进行了使用和操作。

```
// program 8_1.cpp
#include <iostream.h>
#include <string.h>
class employee {          //自定义的 employee 类，它将作为其它几个类的基类
      short age;
      float salary;
  protected:
     char * name;
  public:
     employee (short ag, float sa, char * na){   //基类构造函数
          age=ag;
          salary=sa;
          name=new char[strlen(na)+1];
          strcpy(name,na);
      }
     void print () const{
          cout<<"      "<<name<<":  ";
          cout<<age<<" : ";
          cout<<salary<<endl;
      }
     ~employee() {delete[]name;}
};
class manager:public employee {              //派生类 manager
    int level;                               //行政级别
  public:
    manager(short ag, float sa, char* na, int lev)
          :employee (ag,sa,na){              //派生类构造函数
          level=lev;
    }
    void print () const {
          employee::print();                 //调用基类的 print 显示“共性”数据
               //调用基类（父类）的公有函数成员 print 时要通过类名限定
          cout <<"      level:"<<level<<endl;
     }
};
    /* 注意：允许派生类 manager（子类）中的 print 与基类（父类）的 print 重名，按如下
```

规定进行处理：对子类而言（在子类定义范围内以及通过子类对象访问重名成员时），不加类名限定时默认为是处理子类成员，而要访问父类重名成员时，则要通过类名限定。 */

```
class engineer:public employee {                //派生类 engineer
    char speciality,adegree;
    /* 分别表示专业：'E'--电子， 'M'--机械， 'C'--计算机， 'A'--自动化专业；
    学位：'O'--博士， 'M'--硕士， 'B'--学士， 'N'--无学位 */
public:
    engineer(short ag, float sa, char* na, char sp, char ad)
          :employee (ag, sa, na) {          //派生类构造函数
        speciality=sp;
        adegree=ad;
    }
    void print () const{
        employee::print();                  //调用基类 print
        cout << "      speciality:"<<speciality<<endl;
        cout << "     academic degree:"<<adegree<<endl;
    }
};
enum ptitle {PS,GM,VPS,VGM};
class director:public manager {             //派生类 director
     ptitle post;
  public:
     director(short ag, float sa, char* na, int lev, ptitle po)
          :manager(ag, sa, na, lev) {       //派生类构造函数
        post=po;
     }
     void print () const{
        manager::print();
        cout << "      post:"<<post<<endl;
     }
};
void main() {                               //主函数，对所定义的类进行使用
    employee emp1(23,610.5,"zhang"), emp2(27,824.75,"zhao");
    manager man1(32,812.45,"li",11), man2(34,1200.5,"cui",7);
    engineer eng(26,1420.10,"meng",'E','M');
    director dir(38,1800.2,"zhou",2,GM);
    emp1.print();                           //输出雇员 emp1 的有关全部信息
    emp2.print();                           //输出雇员 emp2 的有关全部信息
    man1.print();                           //输出管理人员 man1 的有关全部信息
    man2.employee::print();                 //调用基类的公有函数成员 print
                                            //显示 man2 的有关雇员信息
    eng.print();                            //输出工程师 eng 的有关全部信息
    dir.print();                            //输出高级主管 dir 的有关全部信息
```

}

程序执行后的显示结果如下：

zhang: 23 : 610.5
zhao: 27 : 824.75
li: 32 : 812.45
level:11
cui: 34 : 1200.5
meng: 26 : 1420.1
speciality:E
academic degree:M
zhou: 38 : 1800.2
level:2
post:1

在上面的程序中，定义了四个类，employee，manager，engineer **和**director**，它们是相关的，其关系就是继承和派生关系。其继承关系如右图所示，关于有关的语法规定在下节介绍。**

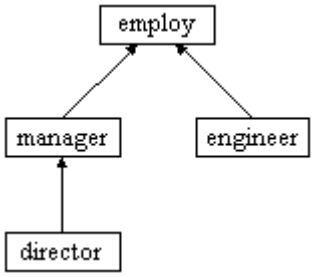


图 8.1 四个类之间的继承关系

8.2 派生类说明及其构造和析构函数

8.2.1 派生类说明

派生类说明也是类说明，不过需要指明它所继承的基类，同时在类的成员中可增加一类保护成员。其一般格式为：

class<**类名**>:<**基类说明表**> {
private:
<**私有成员表**>
protected:
<**保护成员表**>
public:
<**公有成员表**>
};

其中：

基类说明表：列出所给类的基类；

每个基类说明格式为：<派生方式><基类名>

派生方式：公有派生，保护派生或私有派生。

Public 表示公有派生，protected **表示保护派生，**private **表示私有派生。**

派生方式决定了从基类继承过来的成员在派生类中的封装属性。

派生类可以有 1 **至多个基类，或者说，派生类是由** 1 **至多个基类而派生出来**

的类（类型）。通过派生方式来指定各基类成员的被继承方式。

基类成员在各自派生类中的存取权限由表 8.1 给出。

表 8.1 基类成员在各自派生类中的存取权限

派生方式(基类的被继承方式)	在基类中的存取权限	在派生类中的存取权限
public	public	public
public	potected	protected
public	private	(inaccessible)
potected	public	potected
potected	potected	protected
potected	private	(inaccessible)
private	public	private
private	potected	private
private	private	(inaccessible)

public 派生方式：使基类的公有成员和保护成员在派生类中仍然是公有成员和保护成员，而基类的私有成员不可在派生类中被存取。

protected 派生方式：使基类的公有成员和保护成员在派生类中都变为保护成员，而基类的私有成员不可在派生类中被存取。

private 派生方式：使基类的公有成员和保护成员在派生类中都变为私有成员，而基类的私有成员不可在派生类中被存取。

C++语言中类的继承与派生的有关规则是相当灵活的，在有关构造函数，对象初始化，对象指针，以及虚函数等等问题上，需要把有关的概念和规则搞清楚。

单讲继承与派生关系，有下面几点应注意：

（1）一个类可以派生出多个派生类。

（2）一个类可有一个或多个基类，称为单一继承和多重继承。

（3）派生类又可有派生类，称为多级继承。

（4）继承关系不可循环。例如，类 A 继承类 B，类 B 继承类 C，类 C 又继承类 A，这是非法的。

（5）基类的友元关系和基类的构造函数和析构函数都不能被派生类所继承。

从上节的 program 8_1 中可以看到，类 manager 和类 engineer 是类 employee 的派生类，类 director 是类 manager 的派生类。反之，后者是前者的基类。这里的派生方式都是公有派生。从程序中的派生类说明和 main() 函数中可以看出各个成员的引用权限。例如：

class employee 中的 age 是私有数据成员，因此，仅在 employee 内部引用，而 name 是保护成员,因此它可以在类 employee 和派生类 manager 和类 engineer 中引用。其中语句

```
name = new char[strlen(na)+1];
strcpy(name,na);
cout<<name<<":"<<"level."<<level<<endl;
```

都是合法的。

但如果在 main()中出现

cout<<empl.name;

就是非法的了。

* 8.2.2 有关成员存取权限问题的进一步讨论

下面给出一个程序，请仔细分析其中的类继承与派生关系，找出所有违反存取权限的使用情况（语句），并上机进行测试验证，以加深如何对它们进行正确使用的理解。

```
// program 8_2.cpp
#include <iostream.h>
class B {
    int priDat;
protected:
    int proDat;
public:
    int pubDat;
};
class D11 : public B {
    void f11() {
      pubDat=11;          //OK!   pubDat 仍具有 public 存取权限（公有继承）
      proDat=12;          //OK!   proDat 仍具有 protected 存取权限（公有继承）
      priDat=13;          //ERROR!   不可存取基类的 private 成员
    }
};
class D21 : public D11 {
    void f21() {
      pubDat=121;         //OK!   仍为 public
      proDat=122;         //OK!   仍为 protected
      priDat=123;         //ERROR!   仍无法访问基类的 private 成员
    }
};
class D12 : protected B {
    void f12() {
      pubDat=21;          //OK!   pubDat 变为 protected 存取权限（保护继承）
      proDat=22;          //OK!   proDat 仍为 protected 存取权限（保护继承）
      priDat=23;          //ERROR!   不可存取基类的 private 成员
    }
};
class D22 : public D12 {
    void f22() {
      pubDat=221;         //OK!   仍为 protected
      proDat=222;         //OK!   仍为 protected
      priDat=223;         //ERROR!   在继续派生的新类中仍无法访问基类的 private 成员
    }
};
```

```
class D13 : private B {
    void f13() {
        pubDat=31;          //OK!   pubDat 变为 private 存取权限（私有继承）
        proDat=32;          //OK!   proDat 变为 private 存取权限（私有继承）
        priDat=33;          //ERROR!  不可存取基类的 private 成员
    }
};
class D23 : public D13 {
    void f23() {
        pubDat=321;         //ERROR!  不可存取基类的 private 成员
        proDat=322;         //ERROR!  不可存取基类的 private 成员
        priDat=323;         //ERROR!  在继续派生的新类中仍无法访问基类的 private 成员
    }
};
void main() {
    B ob0;
    ob0.pubDat=1;           //OK!   可以存取公有成员
    ob0.proDat=2;           //ERROR!  不可存取 protected 成员
    ob0.priDat=3;           //ERROR!  不可存取 private 成员
    D11 d11;    D21 d21;    D22 d22;    D23 d23;
    d11.pubDat=4;           //OK!   可以存取公有成员
    d11.proDat=5;           //ERROR!  不可存取 protected 成员
    d11.priDat=6;           //ERROR!  不可存取 private 成员
    d21.pubDat=7;           //OK!   可以存取公有成员
    d21.proDat=8;           //ERROR!  不可存取 protected 成员
    d22.pubDat=9;           //ERROR!  不可存取 protected 成员
    d23.pubDat=10;          //ERROR!  不可存取 private 成员
}
```

在上述程序中，凡出现违反存取权限的使用情况（语句），其后的注解处均给出了“ERROR!”标识，并简单指出了出错原因。请查看上面所给出的存取权限表，以加深如何对它们进行正确使用的理解。

由于程序中具有许多处语法错误，无法通过编译，根本谈不上产生运行结果。即是说，只有将所有注有“ERROR!”的语句全部改正后，方可运行该程序得到某种结果。

注意，虽然派生类继承了（“全盘”接收了）基类的所有成员（确切地说是除掉构造函数和析构函数之外的所有成员），但基类的 private 成员在派生类中根本不可被访问（即是说，要想让某成员在派生类中能够被访问的话，就必须在基类中将其说明为 potected 或 public 属性的）。派生方式的不同（由程序员来选择、来控制），也使得被继承过来的那些成员在派生类中具有了新的（可能与基类不同的）存取权限。可以看出，若以 private 派生方式派生出的类（作为基类）再去派生它的派生类（“子类”）时，则在其派生类（“子类”）中，根本无法存取其基类（“父类”）中的任一个成员！

下面再进一步讨论派生类中出现的四种成员以及程序中对它们的使用。

派生类中“全盘”接收了基类的所有成员，通过派生方式的选择，进而又指定了被继承过来的每一成员的新存取权限。除上述成员外，通常还要在派生类中添加本派生类的“特有”成员，正是靠这种添加才使得派生类有区别于其基类，才可使派生类的功能得到进一步扩展

与提高。具体地说，派生类中可出现如下四种成员：

（1）不可访问的成员 — 基类的 private 私有成员被继承过来后，这些成员在派生类中是不可访问的。另外，在建立派生类对象的模块中，也无法通过对象来访问基类的私有成员。如果从派生类继续派生新类，在新类中仍然无法访问基类的这些私有成员。

（2） 私有成员 — 包括在派生类中新增加的 private 私有成员以及从基类私有继承过来的某些成员（如，私有继承过来的基类的 public 以及 potected 成员）。这些成员在派生类中是可以访问的。但是，在建立派生类对象的模块中，是无法通过对象来访问它们的。如果从派生类继续派生新类，在新类中则变成了不可访问成员。

（3）保护成员 — 包括在派生类中新增加的 potected 保护成员以及从基类继承过来的某些成员(如,公有继承过来的基类的potected成员,以及保护继承过来的基类的public 和 potected 成员)。这些成员在派生类中是可以访问的。但是，在建立派生类对象的模块中，是无法通过对象来访问它们的。如果从派生类继续派生新类，在新类中则可能成为私有成员或者保护成员。

（4） 公有成员 — 包括在派生类中新增加的 public 公有成员以及从基类公有继承过来的基类的 public 成员。这些成员不仅在派生类中可以访问，而且在建立派生类对象的模块中，也可以通过对象来访问它们。如果从派生类继续派生新类，则依派生方式的不同，在新类中它们可能成为私有成员，也可能成为保护成员或者公有成员。

8.2.3 派生类的构造函数和析构函数

派生类对象的创建和初始化与基类对象的创建和初始化有关。

在这里必须指出：虽然类 B 的的派生类 A 只可以存取类 B 的保护成员和公有成员，但是，派生类 A 的对象的创建，则必须包含着基类 B 的对象的创建。换句话说，派生类 A 的对象包含了基类 B 的对象，这里面基类 B 的私有成员也在其中！

构造函数和析构函数是用来创建和释放该类的对象的，当这个类是派生类时，其对象的创建和释放应与其基类对象及成员对象相联系。

即是说，构造派生类对象时，要对其基类数据成员、对其所含对象成员的数据成员以及其它的新增数据成员一块进行初始化。这种初始化工作是由派生类的构造函数来完成的（通过初始化符表）。

派生类的构造函数的一般格式如下：

<派生类名>（<参数总表>）：<初始化符表>{

<构造函数体>

}

而 <初始化符表> 按如下格式构成：

<基类名 1>（<基类参数表 1>），…，<基类名 n>（<基类参数表 n>），<对象成员名 1>（<对象成员参数表 1>），…，<对象成员名 m>（<对象成员参数表 m>）

其中的<基类参数表 i>是与<基类名 i>所指基类的某个构造函数相呼应的参数表；而<对象成员参数表 i>是与<对象成员名 i>所属类的某个构造函数相呼应的参数表（注：若无对象成员时，则不出现此后半部分）。这里的基类名与对象成员名的次序无关紧要，各自出现的顺序可以任意。

创建派生类对象时系统按下列步骤工作：

（1）调用各基类的构造函数，调用顺序按照它们被继承时声明的顺序（从左到右）；

（2）再调用各对象成员的构造函数（如果该类有对象成员的话），调用顺序按照它们在派生类中声明的顺序（从左到右）。注意，在派生类中声明对象成员的顺序可以与派生类构造函数处所列对象成员的顺序不相同，它们之间没有必然联系；

（3）最后调用派生类自己的构造函数（执行其函数体）。

释放派生类对象时系统的工作步骤则相反：

（1）先调用派生类自己的析构函数；

（2）再调用对象成员的析构函数，调用顺序按照它们在派生类中声明的相反顺序（从右到左）；

（3）最后调用各基类的析构函数，调用顺序按照它们被继承时声明的相反顺序（从右到左）。

请看下面的简单例子：

```
// program 8_3.cpp
# include<iostream.h>
class CA{
   int  a;
public:
   CA(int  n){  a=n;   cout<<"CA::a="<<a<<endl;  };
    ～CA(){cout<<"CAobj is destructing."<<endl;};
};
class CB{
   int b;
public:
   CB(int  n){  b=n;   cout<<"CB::b="<<b<<endl;  };
   ～CB(){cout<<"CBobj is destrcting."<<endl;};
};
class CC:public CA{
   int c;
public:
   CC(int n1,int n2):CA(n2){  c=n1;   cout<<"CC::c="<<c<<endl;  };
   ～CC(){cout<<"CCobj is destructing"<<endl;};
};
class CD:public CB,public CC{
   int d;
public:
   CD(int n1,int n2,int n3,int n4):CC(n3,n4),CB(n2){
      d=n1;        cout<<"CD::d="<<d<<endl;
   };
   ～CD(){cout<<"CDobj is destructing"<<endl;};
};
void main(void){
   CD CDobj(2,4,6,8);
}
```

这个程序只有一个对象说明语句，但却要运行四个构造函数和四个析构函

数。从下面的运行结果，可以看出调用这些函数的先后次序。输出为：

CB::b=4
CA::a=8
CC::c=6
CD::d=2
CDobj is destructing
CCobj is destructing
CAobj is destructing.
CBobj is destrcting.

从这个例子可知：

（1）派生类构造函数的参数不仅要为自己的数据成员提供初始化数据，还要为基类，以及基类的基类提供初始化数据。

（2）由初始化符表指明哪些参数用于本类，哪些参数用于基类。

（3）在多数多重继承的情况下，初始化工作先基类（多个基类则按基类说明表处的自左至右顺序，而并不按初始化符表处的顺序），再对象成员，最后是自身。如果基类又是一个派生类，那么它的初始化又同样按本条指出的顺序，这是个递归过程。如类 CD 的对象 CDobj 的初始化顺序为：

CD 的基类 CB 的初始化；
CD 的基类 CC 的初始化；
CD 的对象成员的初始化（无）；
CDobj 自身的初始化。

其中：

CD 的基类 CB 的初始化——执行 CB（）；
CD 的基类 CC 的初始化又分为三步：
CC 的基类 CA 的初始化；
CC 的对象成员初始化（无）；
CCobj 自身的初始化。

所以上述初始化过程为：

CB 初始化；
CA 初始化；
CC 初始化；
CD 初始化。

故在设计派生类构造函数时应注意参数的分配。

读者可能已经注意到了如下事实：通过继承可使派生类中“拥有”一个基类的对象；通过将类中的数据成员说明成是另一个类的对象时，也使得在该类中“拥有”了那一个类的对象。但两者在概念和使用上既有关联又有较大的区别。

当类的数据成员为另一个类的对象时，意味着包含，是指 A 类对象中总含有一个 B 类对象（对象成员）。它们属于整体与部分的关系（“has a”关系），如汽车和马达，马达是汽车的一部分。不妨称包含类对象的类为“组装类”。

当使用类的继承产生派生类后，派生类对象中也总“拥有”基类的对象成员，这意味着派生类的对象必然是一个基类对象（“is a”关系），如汽车和轿车，首先轿车就是汽车（具有汽车的所有特征），另外它又比汽车有所特殊（还具有另外一些特殊属性）。

在使用它们时有两点需要注意。一是注意构造函数和析构函数的执行次序以

及“组装类”或派生类构造函数应负有的“责任” — 既要对所包含的每一个对象成员的初始化负责（若含有对象成员的话），又要对其直接基类的初始化负责（若又为派生类的话）。二是注意由于“组装”关系与继承关系的不同，决定了对其对象成员（或基类成员）的访问方式以及对其对象可施加操作的某些不相同。例如，由于派生类的对象必然是一个基类对象，通过派生类对象，也就可以直接调用（或存取）其基类的公有或保护成员函数（或公有及保护数据成员）。如最常用的调用方式为：<派生类对象>.<基类的公有或保护成员>。但通过“组装类”的类对象调用其对象成员的公有成员函数（或公有数据成员）时，则必须使用另外的调用方式：<组装类对象>.<对象成员>.<对象成员所属类的公有成员>。

又比如，如下的赋值操作是允许的：

<基类对象> = <派生类对象>；

因为派生类对象必然是一个基类对象，它包含着基类对象所需要的一切数据（另外还有“富余”，但“富余”部分被“甩掉”不进行赋值）。

但反方向的赋值则不被允许（ <派生类对象> = <基类对象>；）。可这样来理解：基类对象不具有派生类对象所需的一切数据，反方向不具有“is a”关系！不可进行赋值！

另外，如下的两种赋值操作都是不允许的，因为它们的类型不匹配，属于整体与部分的关系，不可相互赋值：

<对象成员所属类的对象> = <组装类对象>；

<组装类对象> = <对象成员所属类的对象>；

8.3 其他特征的继承关系

派生类对于基类，除了继承基类的公有和保护成员之外，在其它方面还有继承关系。

8.3.1 友元关系以及静态成员的继承

1．友元关系

前文已经指出，基类的友元不继承。即，如果基类有友元类或友元函数，则其派生类不因继承关系也有此友元类或友元函数。

另一方面，如果基类是某类的友元，则这种友元关系是被继承的。即，被派生类继承过来的成员，如果原来是某类的友元，那么它作为派生类的成员仍然是某类的友元。总之：

（1）基类的友元不一定是派生类的友元；

（2）基类的成员是某类的友元，则其作为派生类继承的成员仍是某类的友元。

2．静态成员的继承

如果基类中被派生类继承的成员是静态成员，则其静态属性也随静态成员被继承过来。

具体地说，如果基类的静态成员是公有的或是保护的，则它们被其派生类继承为派生类的静态成员。即：

（1）这些成员通常用“<类名>::<成员名>”方式引用或调用。

（2）这些成员无论有多少个对象被创建，都只有一个拷贝。它为基类和派生类的所有对象所共享。

8.3.2 与基类对象和派生类对象相关的赋值兼容性问题

派生类对象间的赋值操作依据下面的原则：

（1）如果派生类有自己的赋值运算符的重载定义，即按该重载函数处理。

（2）派生类未定义自己的赋值操作，而基类定义了赋值操作，则系统自动定义派生类赋值操作，其中基类成员的赋值按基类的赋值操作进行。

（3）二者都未定义专门的赋值操作，系统自动定义缺省赋值操作（按位进行拷贝）。

另一方面，基类对象和派生类对象之间允许有下述的赋值关系（允许将派生类对象“当作”基类对象来使用）：

（1）基类对象 = 派生类对象；

允许这种赋值（在上一节中已经提到了这一特性），但只赋“共性成员”部分（注意，派生类对象除含有基类对象的成员外，通常还具有自己特有的成员以区别于其基类）。

但反方向的下述赋值不被允许：派生类对象 = 基类对象。

（2）指向基类对象的指针 = 派生类对象的地址；

允许该形式的赋值，是函数超载及虚函数（下一节介绍）用法的基础。

注：通过指向基类型的指针可以直接访问基类成员部分，但访问非基类成员部分时，要经过指针类型的强制转换 。

注意，下述赋值不允许：指向派生类类型的指针 = 基类对象的地址。

系统可以作上述规定的根据是，在派生类的对象如 dobj 创建后，其在内存中的存储形式是先存其基类如 CB 的实例各成员，然后是派生类如 CD 自己的各成员。因此，当用 CB 类的指针如 pb 指向其派生类对象如 dobj 时，与指向一个 CB 类对象的效果是一样的。当然，这时 CD 类自有成员的实例部分就不能保证了。

（3）基类的引用 = 派生类对象；

允许进行！即派生类对象可以初始化基类的引用。

注：通过引用只可以访问基类成员部分（而不可访问非基类成员部分，因为不可将基类的引用强制转换为其派生类类型）。

注意，下述赋值不允许：派生类的引用 = 基类对象。

下面给出一个示意性的简单使用例子。

```
// program 8_4.cpp
#include<iostream.h>
class base{                          //基类 base
    int a;
public:
    base (int sa) {a=sa;}
    int geta(){return a;}
};
class derived:public base {          //派生类 derived
    int b;
public:
```

```
    derived(int sa, int sb):base(sa) {b=sb;}
    int getb(){return b;}
};
void main () {
    base bs1(123);                  // base 类对象 bs1
    cout<<"bs1.geta()="<<bs1.geta()<<endl;
    derived der(246,468);           // derived 类对象 der
    bs1=der;                        //OK! “基类对象 = 派生类对象;”
    cout<<"bs1.geta()="<<bs1.geta()<<endl;
    //der=bs1;                      //ERROR! “派生类对象 = 基类对象;”
    base *pb = &der;
            //“指向基类型的指针 = 派生类对象的地址;”
    cout<<"pb->geta()="<<pb->geta()<<endl;          //访问基类成员部分
    //cout<<pb->getb()<<endl;   //ERROR! 直接访问非基类成员部分
    cout<<"((derived *)pb)->getb()="<<((derived *)pb)->getb()<<endl;
            //访问非基类成员部分时，要经过指针类型的强制转换
    //derived *pd = &bs1;
            //ERROR! “指向派生类类型的指针 = 基类对象的地址;”
}
```

程序执行后的显示结果如下：

```
bs1.geta()=123
bs1.geta()=246
pb->geta()=246
((derived *)pb)->getb()=468
```

8.4 派生关系中的二义性处理

继承和派生把不同的类联系到一起，这里就产生了一个同名成员的处理问题，下面我们分几种情形来介绍。

1．单一继承时基类与派生类间重名成员的处理

单一继承（只有一个基类）时，若基类与派生类的成员重名，则按如下规定进行处理：对派生类而言（在派生类定义范围内以及通过派生类对象访问重名成员时），不加类名限定时默认为是处理派生类成员，而要访问基类重名成员时，则要通过类名限定。

例如基类 CB **有保护数据成员** int a **;有公有函数成员** void show（void）{…}。**而** CB **的派生类** CD **也有自己的数据成员** int a**,和函数成员** void show（void）{…}**；这时通过** CD **的对象** CDobj **来引用它们：**

CDobj.a，CDobj.show()

这里用的是派生类 CD **自己的成员** a **和** show（）。**如果需要引用从** CB **中继承过来的成员** a **和** show（），**则还需加上类限定符：**

CDobj.CB::a，CDobj.CB::show()

从而避免了二义性。

下面看一个简单例子。

```
// program 8_5.cpp
#include <iostream.h>
class CB {
public:
    int a;
    CB(int x){a=x;}
    void showa(){cout<<"Class CB -- a="<<a<<endl;}
};
class CD:public CB {
public:
    int  a;                                          //与基类 a 同名
    CD(int x, int y):CB(x){a=y;}
    void showa(){cout<<"Class CD -- a="<<a<<endl;}   //与基类 showa 同名
    void print2a() {
        cout<<"a="<<a<<endl;                         //派生类数据成员 a
        cout<<"CB::a="<<CB::a<<endl;                 //访问基类重名成员 a
    }
};
void main() {
    CB CBobj(12);
    CBobj.showa();                                   //调用基类的 showa()
    CD CDobj(48, 999);
    CDobj.showa();                                   //派生类 CD 的 showa
    CDobj.CB::showa();                               //访问基类 showa
    cout<<"CDobj.a="<<CDobj.a<<endl;                 //派生类 CD 的 a
    cout<<"CDobj.CB::a="<<CDobj.CB::a<<endl;         //访问基类 a
}
```

程序执行后的显示结果如下：

```
Class CB -- a=12
Class CD -- a=999
Class CB -- a=48
CDobj.a=999
CDobj.CB::a=48
```

2．多重继承情况下二基类间重名成员的处理

在多重继承时，也可能出现类似的问题。设类 CD 以类 CB1，类 CB2 为基类，这是多重继承。类 CB1 和类 CB2 可能有同名或同原型的成员，在它们都被 CD 所继承时，即可能产生二义性，如 show（）同为 CB1 和 CB2 的公有成员，则类 CD 从 CB1 和 CB2 继承了两个可能函数体不同的函数成员 show（）。显然通过 CD 的对象 CDobj 直接引用函数 show（）即 CDobj.show（）可能产生二义性。这里的解决方法和上文一样，也是要加上基类限定符：

CDobj.CB1::show( )**和** CDobj.CB2::show( )

如此二义性问题得到解决。

概括地说，多重继承（具有多个基类）情况下二基类间成员重名时，要按如下方式进行处理：对派生类而言（在派生类定义范围内以及通过派生类对象访问

重名成员时），不加类名限定时默认为是处理派生类成员，而要访问基类重名成员时，则要通过类名限定。

3．多级混合继承（非虚拟继承）包含两个基类实例情况的处理

更复杂的情况出现在多级多重混合继承的情形，现在以一个比较简单的情况为例：类 CD 以类 CB1 和类 CB2 为基类，同时，类 CB1 和 CB2 又同以类 CA 为基类，这是一个最简单的混合继承的情形。这种情形会产生更为有趣的现象，即使在上面的四个定义中没有同名或同原型的成员，也会产生二义性的问题。这是因为当创建了类 CD 的一个对象 CDobj 时，也就是同时首先创建了其基类 CB1 和 CB2 的对象，而在创建 CB1 和 CB2 的对象过程中，又必须首先分别创建基类 CA 的对象，请注意这里的特征：

① CD 的对象 CDobj 中首先应包含 CB1 的对象和 CB2 的对象来作为 CDobj 的组成部分；

② CB1 的对象则首先应以一个 CA 的对象作为其组成部分；

③ CB2 的对象亦首先应以一个 CA 的对象作为其组成部分。

由此可以看出，在 CDobj 中包含着两个 CA 的实例。

例如 CA 有数据成员 a 和函数成员 show()，那么，通过 CD 对象引用 CDobj.a，CDobj.show（），或在 CD 类内的函数成员中引用 a 和 show（），都会产生二义性问题。因为这时 CDobj.a 和 CDobj.show（）都有两个不同的实体与其相对应。

这时的二义性问题从表面上难于理解：出自同一个类 CA 的定义，引自同一对象 CDobj，似乎不应产生问题。而深入地分析指出，这里的二义性产生自对象 CDobj 包含着两个 CA 的对象。

C++语法解决这个问题的方法有两个：

① 仍用类限定符：

CDobj.CB1::a 和 CDobj.CB2::a，

CDobj.CB1::show( )和 CDobj.CB2::show( )

如此指出 CDobj.CB1::a 是 CD 通过 CB1 从 CA 继承过来的成员 a，CDobj.CB2::a 是 CD 通过 CB2 从 CA 继承过来的成员 a，以示区别。

② 通过虚基类。虚基类的概念及其使用方法，将在下一节进行介绍。

上述所讨论的类间继承关系示例如下：

```
class A{…};
class B : public A{…};
class C : public A{…};
class D : public B, public C{…};
```

D 类对象的存储结构示意：

（ ((A) B) ((A) C) D ）

上述多级混合继承关系应用例举：

例 1. 类 A--人员类； 类 B--学生类； 类 C--助教类； 类 D--学生助教类。

例 2. 类 A--人员类； 类 B--学生类； 类 C--工人类； 类 D--工人学生类。

例 3. 类 A--家具类； 类 B--沙发类； 类 C--床类； 类 D--沙发床类。

* 8.5 虚基类

为解决由混合（多重多级）继承造成的二义性问题，所采用的另一方法是，利用 C++语

言中的虚基类的概念：即类 B 作为类 D1，D2，……，Dn 的基类，当把类 B 定义为派生类 D1，D2，……，Dn 的虚基类时，各派生类的对象共享其基类 B 的一个拷贝。这种继承称为共享继承。

说明方式为：在派生类 D1,D2,……，Dn 定义的基类表中，关于基类 B 的说明中，增加关键字“virtual”。例如：

```
class A{…};
class B：virtual public A{…};
class C：virtual public A{…};
class D：public B，public C{…};
```

如此，当说明类 D 的对象 Dobj 时，它将只包含类 A 的一个拷贝，上一节提到的二义性问题将不再发生。

注意，虚基类的说明是在定义派生类时靠增加关键字“virtual”来指出的。在使用了虚基类后，系统将进行“干预”，使各派生类的对象共享其基类的同一个拷贝。

采用虚拟继承后，D 类对象的存储结构改变为：

```
(  (  (A) B C  )  D  )
```

* 8.5.1 虚基类一般应用示例

```
// program 8_6.cpp
#include <iostream.h>
class A {
public:
    int i;
    void showa(){cout<<"i="<<i<<endl;}
};
class B: virtual public A {          //对类 A 进行了虚拟继承
public:
    int j;
};
class C: virtual public A {          //对类 A 进行了虚拟继承
public:
    int k;
};
class D:public B,public C {
    //派生类 D 的二基类 B、C 具有共同的基类 A，但采用了虚拟继承
    //从而使类 D 的对象中只包含着类 A 的 1 个实例
public:
    int n;
    void showall(){
        cout<<"i,j,k,n="<<i<<","<<j<<","<<k<<","<<n<<endl;
                        //若非虚拟继承时会出错！因为“D::i”具有二义性
    }
};
void main() {
```

```
    D Dobj;                 //说明 D 类对象
    Dobj.i=11;              //若非虚拟继承时会出错！因为“ D::i ”具有二义性
    Dobj.j=22;
    Dobj.k=33;
    Dobj.n=44;
    Dobj.showa();           //若非虚拟继承时会出错！因为“ D::showa ”具有二义性
    Dobj.showall();
}
```

程序执行后的显示结果如下：

```
i=11
i,j,k,n=11,22,33,44
```

* 8.5.2 具有显式有参构造函数的虚基类的初始化问题

若虚基类的构造函数具有参数的话，则对其任一个直接或间接派生类的构造函数来说，它们的成员初始化列表中都必须包含有对该虚基类构造函数的直接调用。为了保证虚基类子对象只被初始化一次，规定只在创建对象的那一派生类的构造函数中调用虚基类的构造函数，而忽略该派生类的各基类构造函数中对虚基类构造函数的调用。

另外，C++**又规定，在派生类构造函数的成员初始化列表中，若有虚基类构造函数调用的话，则对它们的调用将优先于非虚基类构造函数。**

下一示例的类间虚拟继承关系如下：

```
class ClaA{…}；
class ClaB：virtual public ClaA{…}；
class ClaC：virtual public ClaA{…}；
class ClaD：public ClaB, public ClaC{…}；
```

ClaD **类对象存储结构示意：**

```
(  (  (ClaA)  ClaB  ClaC)  ClaD  )
```

但注意虚基类 ClaA **的构造函数具有参数，在它的间接派生类** ClaD **的构造函数处，其成员初始化列表中还必须包含对该虚基类构造函数的直接调用（若** ClaA **非虚基类时，则不需要也不可以这样做 !)。**

```
// program 8_7.cpp
#include <iostream.h>
class ClaA {                            //ClaA 为虚基类
public:
    ClaA(int a0){                       //虚基类 ClaA 的构造函数有参数
        cout<<"ClaA =>a0="<<a0<<endl;
    }
};
class ClaB: virtual public ClaA {       //派生类 ClaB，虚拟继承了 ClaA
public:
    ClaB(int a0, int b0):ClaA(a0) {     //要对其直接基类（虚基类）ClaA 的初始化负责
        cout<<"ClaB =>a0,b0="<<a0<<", "<<b0<<endl;
    }
};
```

```
class ClaC: virtual public ClaA       {       //派生类 ClaC，虚拟继承了 ClaA
public:
        ClaC(int a0, int c0):ClaA(a0){        //要对其直接基类（虚基类）ClaA 的初始化负责
                cout<<"ClaC =>a0,c0="<<a0<<", "<<c0<<endl;
        }
};
class ClaD:public ClaB, public ClaC {
        //二重继承，派生类 ClaD 的二基类 ClaB、ClaC 又具有共同的虚基类 ClaA
public:
        ClaD(int a0, int b0, int c0, int d0)
                :ClaB(a0,b0), ClaC(a0,c0), ClaA(8086)
        {
                cout<<"ClaD =>a0,b0,c0,d0="<<a0<<", "<<b0<<", "<<c0<<", "<<d0<<endl;
        }
};
void main() {
        ClaA Aobj(11);
        cout<<"------------------------"<<endl;
        ClaB Bobj(22,33);
        cout<<"------------------------"<<endl;
        ClaC Cobj(44,55);
        cout<<"------------------------"<<endl;
        ClaD Dobj(404, 505, 606, 707);
}
```

程序执行后的显示结果如下：

```
ClaA =>a0=11
------------------------
ClaA =>a0=22
ClaB =>a0,b0=22, 33
------------------------
ClaA =>a0=44
ClaC =>a0,c0=44, 55
------------------------
ClaA =>a0=8086
ClaB =>a0,b0=404, 505
ClaC =>a0,c0=404, 606
ClaD =>a0,b0,c0,d0=404, 505, 606, 707
```

注意，虚基类 ClaA 的构造函数具有参数，ClaD 是虚基类 ClaA 的间接派生类。类 ClaD 的构造函数除去要对其直接基类 ClaB 和 ClaC 的初始化负责外，还必须负责对虚基类 ClaA 的初始化（必须包含有对虚基类 ClaA 之构造函数的直接调用，如本例的“ ClaA(8086) ”）。

在该派生类构造函数的成员初始化列表中，对虚基类构造函数的调用“ ClaA(8086) ”将优先于对其它两个非虚基类构造函数的调用。

另外注意，虚基类子对象只是通过“ ClaA(8086) ”被初始化了一次，该派生类的两个基类构造函数中，通过“ ClaB(a0,b0) ”及“ ClaC(a0,c0) ”对虚基类构造函数的调用都将被忽略。

但是，如果虚基类的构造函数没有参数、或者根本就没提供显式的虚基类构造函数的话（表示用该虚基类的无参构造函数或系统默认构造函数来完成有关的初始化工作），此时则不需要按上述方式做 -- 对其任一个直接或间接派生类的构造函数来说，它们的成员初始化列表中都不再需要包含有对该虚基类构造函数的直接调用。读者可作为练习去上机进行具体测试。

注意，本程序中只出现了一个虚基类 ClaA 的间接派生类 ClaD，如若还出现别的间接派生类 – 如 ClaD 又进一步派生出 ClaE 时，则在 ClaE 的构造函数处也必须包含对虚基类 ClaA 之构造函数的直接调用（如下面的“ ClaA(e0) ”）。

```
class ClaE:public ClaD {
public:
    ClaE(int a0, int b0, int c0, int d0, int e0)
        :ClaD(a0,b0,c0,d0), ClaA(e0)
    {
        …
    }
};
```

8.6 多态性与虚函数

多态性是面向对象程序设计的重要特征，它与封装性和继承性一起并称 OOP 的三大特征。类和对象把相关的数据与程序组合到一起，谓之封装；类与对象的派生关系是类之间最主要的关系，谓之继承；对于同样的消息——指对于类的成员函数的调用命令，当不同类型的对象接收时可以导致完全不同的行为，谓之多态性。封装使 C++程序组织严密，继承使其结构科学，多态则使其生动而富有魅力。

结构程序设计强调名字的区分，如类型名，常量名，变量名，函数名等等，各有其明确的说明而不相混淆。在这基础上面向对象程序设计则支持多态性，从现实世界的实际情况和人们的习惯出发，C++程序中允许大量的同名函数出现。

第一类多态性体现在 C++语言允许函数重载和运算符重载（overloading），使用同样的函数名和同样的运算符来完成不同的数据处理与操作。另一类多态性则体现在 C++语言程序中允许存在有若干函数，有完全相同的函数原型，却可以有多种多样的相异的函数体。这种现象称为函数的超载或过载（overriding）。它的实现与类的继承与派生相联系，虚函数（virtual function）概念是实现的关键。这种超载功能，使用户能够编出更好的程序。

8.6.1 函数重载与静态联编

函数重载指的是，允许多个不同函数使用同一个函数名，但要求这些同名函数具有不同的参数表（当然，函数体的实现代码通常也不同）。函数重载的概念与简单使用已经在第五章介绍过，重新提及它是为了与下面介绍的函数超载及动态联编进行比较与区别。

类定义处允许给出多个同名的构造函数，但要求它们的参数表必须不相同，这也是一种函数重载的概念。

允许函数重载和运算符重载，则可通过使用同样的函数名和同样的运算符来

完成不同的数据处理与操作。例如，同一个函数名 abs（）可以计算不同类型数据的绝对值；同一运算符“＜＜”可有完全不同的运算与之对应（可以是整数的移位，也可以是数据的输出）。因为重载函数的形参表（进而调用时的实参表）是不同的，因此，系统对这种多态性，对同名函数的处理比较简单。在编译过程中就可以确定该函数与程序中的哪一段代码相联系，即在编译时就已确定函数调用语句对应的函数体代码，故称为静态联编（static banding）处理方式。

函数重载可以是相对同一个类的类内或类外函数，也可出现于完全无关的多个类中。通常这些函数可以是相互语义无关的。

8.6.2 函数超载、虚函数及动态联编

多态性的另一种体现在 C++语言程序中允许存在有若干函数，有完全相同的函数原型，却可以有多种多样的相异的函数体。例如前文曾讨论过，如不同类中的 print（）函数，show（）函数，draw（）函数，这种现象称为函数的超载或过载。它的实现与类的继承与派生相联系，虚函数概念是实现的关键。例如 show（）可以是一个对外的公共的调用函数方式，但在程序中可以隐藏起多种完全不同的处理，虽然它们可能都是要向屏幕显示某些内容。由于超载函数可以允许不同的函数具有相同的函数原型，因此，在编译阶段，系统是无法判断此次调用应执行哪一段函数代码。只有到了运行过程中执行到此处时，才能临时判断应执行哪一函数代码，这种处理方式称为动态联编（dynamic banding）。

1．函数超载

．仅在基类与其派生类的范围内实现（与使用）；

．允许（支持）多个不同函数（的实现）使用完全相同的函数名、函数参数表以及函数返回类型；

．通常这些函数是语义相近的或完全相同的（仅具体实现方法即实现代码不同而已!）。

2．虚函数

虚函数是 C++语言中的重要概念。虚函数在编程中的灵活使用，可使程序具有更好的结构和可重用性。

．在定义某一基类（或其派生类）时，若将其中的某一非静态成员函数的属性说明为 virtual，则称该函数为虚函数。其一般说明形式为：

virtual <返回类型><函数名>（<参数表>）{…}；

．虚函数的使用与函数超载密切相关。若基类中某函数被说明为虚函数，则意味着其派生类中也要用到与该函数同名、同参数表、同返回类型、但函数（实现）体不同的这同一个所谓的超载函数。例如：

```
class graphelem {              //自定义类 graphelem，将作为其它图元类的基类
  protected:
    int color;                 //颜色 color
  public:
    graphelem(int col){
        color=col;
    }
    virtual void draw(){ … }；//基类中含有一个虚函数
};
```

上述的基类 graphelem 中之所以自定义了一个虚函数 draw，是因为它的每一个派生类都要“ draw ”出属于那一派生类的类对象图形（虽然都叫做“ draw ”，但各函数的实现可以完全不同），所以可利用函数超载的手段，在基类 graphelem 及其派生类中，共用同一个虚函数 draw。下面是其各派生类定义的“ 构架 ”。

```
class line:public  graphelem{        //自定义类 line，为基类 graphelem 的派生类
  public：
    virtual void draw(){ ... }；   //虚函数 draw，负责画出“ line ”
    ...
}；
class circle:public graphelem{       //自定义类 circle，为基类 graphelem 的派生类
  public：
    virtual void draw(){ ... }；   //虚函数 draw，负责画出“ circle ”
    ...
}；
class triangle:public graphelem{     //类 triangle，为基类 graphelem 的派生类
  public：
    virtual void draw(){ ... }；   //虚函数 draw，负责画出“ triangle ”
    ...
}；
```

把基类中的函数 draw()说明为虚函数，从而其所有派生类中的具有相同原型的函数成员 draw()也就都成为虚函数。虚函数的机制要点为：

（1）在基类 CB 中说明某一函数成员 f（）为虚函数，方法是在说明前加关键字“ virtual ”。如 virtual void draw（）；

（2）在 CB 的各个派生类 CD1，CD2，…，CDn 中定义与 f（）的原型完全相同（但函数体可以各异）的函数成员 f（）。无论是否用关键字 virtual 来说明它们，它们自动地定义为虚函数（即是说，派生类中虚函数处的关键字 virtual 可以省略，但基类处的不可省）。

（3）当在程序中采用以 CB 类指针 pb 的间接形式调用函数 f（），即使用 pb->f（）时，系统对它将采用动态联编的方式进行处理。

虚函数和重载函数既有相似之处，又相互有区别。它们都是在程序中设置一组同名函数，都反映了面向对象程序中多态性特征，但虚函数有自己的特点：

（1）虚函数不仅同名，而且同原型。

（2）虚函数仅用于基类和派生类之中，不同于函数重载可以是类内类外函数。

（3）虚函数需在程序运行时动态联编以确定具体函数代码，而重载函数在编译时即可确定。

（4）虚函数一般是一组语义相近的函数，而函数的重载，可能相互是语义无关的。

另一点要指出的是：构造函数不能说明为虚函数。这是显然的，因为构造函数的调用一般出现在对象创建的同时或之前，这时无法用指向其对象（尚未创建）的指针来引用它。但析构函数可以说明为虚函数，此时这一组虚函数的函数名是不同的。当在析构函数中采用基类指针释放对象时，应注意把析构函数说明为虚函数，以确定释放的对象。

3．动态联编

与虚函数以及程序中使用指向基类的指针（变量）密切相关。

注意：C++规定，基类指针可以指向其派生类的对象（也即，可将派生类对象的地址赋给其基类指针变量），但反过来不可以。这一点正是函数超载及虚函数用法的基础。

例 1. 建立上述类 line、类 circle 以及类 triangle 的类对象，而后调用它们各自的 draw 函数“画出”它们。

方法 1：直接通过类对象调用它们各自的 draw 函数（由类对象可以唯一确定要调用哪一个类的 draw 函数）。

```
line ln1;          circle cir1;          triangle tri1;
ln1.draw();        cir1.draw();          tri1.draw();
```

方法 2：使用指向基类的指针（变量），而后通过指针间接调用它们各自的 draw 函数（动态联编方式，要靠执行程序时指针的“动态”取值来确定调用哪一个类的 draw 函数）。

```
graphelem   *pObj;
line ln1;          circle cir1;          triangle tri1;
pObj=&lin1;        pObj->draw();
pObj=&cir1;        pObj->draw();
pObj=&tri1;        pObj->draw();
```

例 2. 假设 inte_algo 为基类（准备实现定积分的计算），其中说明了一个虚函数 integrate，并在其三个派生类（rectangle、ladder、simpson -- 表示矩形计算方法、梯形计算方法、simpson 计算方法）中，也说明了该超载函数 integrate （必然也为虚函数）。

此时，程序中可设一个指向基类 inte_algo 的指针变量 p，如，inte_algo * p；那么，可通过使用如下形式的一个通用函数 integrateFunc 来实现调用不同派生类的超载函数（即虚函数）integrate 的目的（从而最终实现三种不同的计算定积分的方法）：

```
void integrateFunc(inte_algo * p) {    //基类指针 p 可指向任一个派生类的对象
    p->integrate();          //根据 p 的指向，调用不同派生类的 integrate 函数
}
```

而后，只需在主调函数处使用如下形式的语句来实现三种不同定积分的计算：

```
integrateFunc( <某派生类的对象地址> );
```

该函数调用的功能等同于：

```
p =  某派生类的对象地址;   //使用基类指针指向其派生类的对象
p->integrate();                    //将调用 p 所指向类的 integrate 虚函数
```

在编译阶段，系统发现要调用的是一个虚函数 integrate，而此时它又无法确定，程序究竟要调用哪一个派生类的超载函数 integrate（因为依赖于运行时 p 指针的动态取值!）。此种情况下，系统将采用动态联编方式来处理：在运行阶段，通过 p 指针的当前值，去动态地确定出它所指向的那一个对象所属的类（基类或某一派生类），而后，再去找到该类的那一个函数（的函数体代码，进而去执行）。

由于虚函数以及动态联编的概念与使用较难理解，下面给出进一步的有关说明。假设 pb 为指向基类的指针，而在基类及其派生类中使用了虚函数 f（）。

所谓动态联编的处理方式，就是指到了程序执行时才决定把表示式 pb->f（）到底和哪一个 f（）的执行代码相联系，其执行步骤是：

（1）在编译过程中，扫描到表达式 pb->f（）时，首先检查 f（）是否为虚

函数（若 f（）不是虚函数，则按静态联编处理，在编译时必须为 f（）确定对应的函数体代码）。

（2）若 f（）为虚函数，则仅把与 f（）同原型的虚函数的地址信息等列表待查。

（3）在程序运行阶段，当程序执行到表达式 pb->f（）时，根据指针当前所指向的对象类属，来决定这时的 f（）应执行哪个类中的哪个 f（），从而决定执行哪个函数体。

这里，虚函数的动态联编的实现，主要依赖于下面一些条件：

（1）基类 CB 可有多个派生类 CD1，CD2，…，CDn。

（2）CB 类指针 pb 可以指向 CB 的对象，也可以指向 CD1，CD2，…，CDn 的对象。

（3）CB 类和 CD1，CD2，…，CDn 类可以有同样原型（而函数体各异）的函数成员 f（）（虚函数）。

（4）表达式 pb-> f（）虽然可以代表不同的函数 CB::f（），CD1::f（），…，CDn::f（），但却不必由程序员在编程时用类属限定符“ CB:: ”等来指定这里的 f（）是哪一个——这就是动态联编的关键。运行着的程序根据查对当前指针 pb 所指向的对象是属于哪个类的，再决定到底执行哪个 f（）。

在程序设计中利用虚函数和动态联编的方式，可以提高程序的水平和质量。是否能够在程序中充分地，正确地使用虚函数，是衡量一个 C++程序员编程水平的标志之一。采用虚函数对于程序有益之处在于：

（1）可使程序简单易读。例如，如果不如此处理上面讨论的实例，“ pb->f（）”肯定就要复杂的多；首先，在 f（）之前须增加类属限定符，显式地指明这里的 f（）是指哪个类的函数成员；其次，由于当前 pb 指针指向的对象是属于哪个类的，不一定可以简单确定，很可能需要若干条件判断语句来完成。

（2）它使得程序模块间的独立性加强。例如有另一个类需要对各种图形（firgure）进行屏幕显示，通过虚函数的处理，它可以只与基类 figure 及其指针 pb，函数成员 draw（）直接打交道，而与 figure 的派生类 rectangle，circle，triangle，square 等没有直接的联系。

（3）增加了程序的易维护性。例如，在 figure 类的派生类中再增加一个派生类 ladder，除了增加类 ladder 的定义之外，有关调用虚函数 draw（）的程序不需要修改！

（4）提高了程序中“ 信息隐藏 ”的等级。类的封装本身是私有成员的隐藏，而在基类与派生类之间虚函数的设置，实际上是以各派生类的基类为对外的接口，被隐藏的实际上是在各派生类中、其内容各异的实际处理。

读者应从这里体会面向对象程序设计所带来的程序内在质量的提高。

8.6.3 纯虚函数与抽象基类

抽象基类的概念是虚函数概念的自然引申，它是虚函数使用的一个更理想的形式。

如果不准备在基类的虚函数中做任何事情，则可使用如下的格式将该虚函数说明成纯虚函数：

virtual <函数原型>=0;

即是说，若在虚函数的原型后加上“ =0 ”字样而替掉函数定义体（没有具体的实

现），则这样的虚函数称为纯虚函数。例如：

```
virtual void print()=0;
```

纯虚函数只为其派生类的各虚函数规定了一个一致的“原型规格”（该虚函数的实现将在它的派生类中给出）。

含有纯虚函数的基类称为抽象基类。注意，不可使用抽象基类来说明并创建它自己的对象，只有在创建其派生类对象时，才有抽象基类自身的实例伴随而生。

抽象基类的引入体现了上文中虚函数使基类作为这一组类（基类和它的若干派生类）的抽象对外接口的思想。通过抽象基类，再“加上”各派生类的特有成员以及对基类中那一纯虚函数的具体实现，方可构成一个具体的实用类型。

许多引入虚函数的程序，把基类的虚函数说明为纯虚函数，从而使基类成为一种抽象基类，可以更自然的反映实际应用问题中对象之间的关系。

注意，如果一个抽象基类的派生类中没有定义基类中的那一纯虚函数、而只是继承了基类之纯虚函数的话，则这个派生类还是一个抽象基类（其中仍包含着继承而来的那一个纯虚函数）。

关于虚函数和纯虚函数的应用，在下面一节的程序中体现。

8.7 虚函数使用实例

本节将给出两个使用虚函数的程序实例，第一个程序用来计算函数的定积分，而另一个则利用图元类来进行屏幕画图。

8.7.1 计算函数的定积分

假设函数 f（x）=4/（1+x*x），求函数 f（x）的定积分：$\int_a^b$ f（x）dx 的值。

设计一个程序，计算定积分的近似值，可以有三种近似算法供选择，它们都须首先把区间[a，b]分成 n 个等份，于是每一小段 h=（b-a）/n，函数 f（x）的自变量将顺序取为：a，a+h，a+2h，……，a+（n-1）h，a+nh=b，共有 n+1 个点。

矩形法积分近似计算公式为：

sum=（f（a）+f（a+h）+f（a+2h）+ …… +f（a+（n-1）h））h

梯形积分计算公式为：

sum=（f（a）+2f（a+h）+2f（a+2h）+……+2f（a+（n-1）h）+f（b））h/2

simpson 法积分公式为：

sum=（f（a）+4f（a+h）+2f（a+2h）+4f（a+3h）+2f（a+4h）+……+2f（a+（n-2）h + 4f（a+（n-1）h）+f（b））h/3

在下面的程序 program8_8 中，以类的形式定义了定积分的不同近似计算方法，使得用户可以方便地选择它们。

从程序实例中可以看到超载和重载的区别，前者定义了多个原型完全相同但函数体代码不同的函数。后者则是两个以上函数名相同但参数表（和函数体）不同的函数。

程序实现时，首先定义一个基类 inte_algo，并在其中说明一个虚函数 integrate；而后分别定义基类的三个派生类 -- rectangle、ladder、simpson，且在此三个派生类中也说明同一个虚函数 integrate（函数超载概念），但它们各自的

函数体不同（对应着不同的积分方法）。而后通过使用指向基类 inte_algo 的指针变量 p，并使 p 指向不同的派生类对象，最终使系统通过动态联编的方式去实现调用不同派生类的超载函数（即虚函数）integrate 的目的，进而实现三种不同的计算定积分的方法。

```
// program 8_8.cpp
#include <iostream.h>

float function(float x){   //欲积分的函数
      return  4.0/(1+x*x);
}

class inte_algo {          //基类 inte_algo
protected:
    float  a,b;            //a，b 为积分区间的左右边界
     int  n;               //n 表示把[a，b]区间划分成多少个小区段进行积分
     float  h,sum;         //h 表示步长，sum 表示积分结果值
public:
      inte_algo (float left, float right, int steps) {
           //基类构造函数，由参数带来的 left，right 将赋给表示积分区间
           //左右边界的 a 与 b，而将 steps 赋给表示划分多少个小区段的 n
           a=left;        b=right;
           n=steps;       h=(b-a)/n;          //算出步长 h
           sum=0.0;                           //sum 初值为 0
       }
       virtual  void integrate(void);         //基类中说明了一个虚函数 integrate
};

class rectangle:public inte_algo {             //派生类 rectangle
public:
        rectangle(float left,float right,int steps)
              :inte_algo  (left,right,steps){
         }
       virtual  void  integrate(void);         //虚函数 integrate
};

class ladder:public inte_algo {                //派生类 ladder
public:
        ladder(float left,float right,int steps)
              :inte_algo  (left,right,steps){
         }
       virtual void integrate(void);
};

class simpson:public inte_algo {               //派生类 simpson
```

```
public:
        simpson(float  left,float right,int steps)
              :inte_algo  (left,right,steps){
         }
       virtual void integrate(void);
};

void  inte_algo::integrate(){                    //基类虚函数定义
      cout<<sum<<endl;
}

void  rectangle::integrate(){                    //派生类 rectangle 之虚函数定义
      //计算定积分的公式为：sum=( f(a)+f(a+h)+f(a+2h)+...+f(a+(n-1)h) )h
      float  al=a;      // al 为调用 f 函数时的实参值，依次取值 a，a+h，a+2h，...
      for(int i=0; i<n; i++) {                   //共在 n 个点处计算函数值
            sum+=function(al);
            al+=h;                               //al 每次增加一个步长 h
      };
      sum*=h;
      cout<<sum<<endl;                           //显示积分结果 sum
}

void  ladder::integrate(){                       //派生类 ladder 之虚函数定义
      //计算公式：sum=( f(a)+2f(a+h)+2f(a+2h)+...+2f(a+(n-1)h)+f(b) )h/2
      float  al=a;      //al 为调用 f 函数时的实参值，依次取值 a，a+h，a+2h，...
      sum=(function(a)+function(b)/2.0);         //对边界点 a，b 特殊处理
      for(int  i=1;i<n;i++)   {                  //还要在 n-1 个点处计算函数值
            al+=h;
            sum+=function(al);
      };
      sum*=h;
      cout<<sum<<endl;
}

void  simpson::integrate(){                      //派生类 simpson 之虚函数定义
            //计算公式：sum = ( f(a) + 4f(a+h) + 2f(a+2h) + 4f(a+3h)
            //+ 2f(a+4h) + ... + 2f(a+(n-2)h) + 4f(a+(n-1)h)+h(b) )h/3
      sum=function(a)+function(b);               //对边界点 a，b 特殊处理
      float  s=1.0,   al=a;
            // s 依次取值 1,-1；1,-1；...。从而可使(3.0+s)依次成为公式中
            //的系数 4,2，  4,2，  4,2，...；  al 依次取值  a，a+h，a+2h，...
      for(int  i=1;i<n;i++)   {                  //还要在 n-1 个点处计算函数值
            al+=h;
            sum+=(3.0+s)*function(al);           //(3.0+s)依次为系数  4,2，4,2，4,2，...
```

```
            s=-s;
        };
        sum*=h/3.0;
        cout<<sum<<endl;
}

void integrateFunc(inte_algo * p) {
        p->integrate();
            //根据 p 所指向的对象的不同，调用不同派生类的 integrate
}

void main() {
    rectangle rec(0.0, 1.0, 10);          //rectangle 类对象 rec
    ladder lad(0.0, 1.0, 10);             //ladder 类对象 lad
     simpson sim(0.0, 1.0, 10);           //simpson 类对象 sim
     integrateFunc(&rec);
                //实参指向了派生类 rectangle 的对象，使 integrateFunc 函数
                //去调用 rectangle 类的 integrate 虚函数
     integrateFunc(&lad);          //调用的将是 ladder 类的 integrate 虚函数
     integrateFunc(&sim);          //调用的将是 simpson 类的 integrate 虚函数
}
```

程序说明：

（1）这个程序运行后的输出结果为：

```
3.23993
3.33993
3.14159
```

它们实际上是对于定积分 $\int_0^1 4/(1+\chi^2)dx$ 的近似计算结果。采用三种不同的算法，但都是把区间[0，1]划分为 10 小份来计算，从而以三种不同的方式求出各小份对应的小面积 s[i]，并将它们累加到一起（程序中的 sum 累加值）来作为积分的近似结果。事先我们已经知道，上面定积分的精确值正好是圆周率 π。三种算法结果不同，通过比较可以知道 simpson 算法的精度最高。

（2）程序中定义了 4 个类，其中 inte_algo 是基类，其它三个类是它的派生类。基类中的五个数据成员 a，b，n，h，sun 是它的派生类的共有的成员。在这个例中，派生类没有自己特有的数据成员。

基类 inte_algo 除构造函数外，还有一个公有函数成员 integrate()，它被说明为 virtual（虚）函数。inte_algo 的三个派生类 rectangle，ladder 和 simpson 都有一个同名的公有函数 integrate()；这三个函数也是虚函数。

从程序的这种安排，可以了解到设计者的思路：基类并不是完整的类，它在程序中只起到一种框架的作用，它所包含的是与定积分计算相关的数据成员，和计算定积分的函数成员的格式：函数名，参数表，返回类型等等。换句话说，基类把三个派生类的公共一致的成分包括到它自己的内部，而把三种算法的差异部分留给各个派生类。

在这种情况下，可以利用 C++语言提供的“纯虚函数”的概念，干脆把基类

变为“抽象基类”，其办法是把基类中对虚函数的说明改为：

virtual void integrate（void）= 0；

同时删去类体外对该虚函数的那三行定义。从而

① 基类 inte_algo 的函数成员 integrate（）就是一个纯虚函数；

② 由于基类 inte_algo 有纯虚函数成员，从而它就是一个抽象基类，抽象基类不允许创建它自己的对象。只有在创建其派生对象时，才有它自身的实例出现。

下一个实例中，就是采用纯虚函数来实现的。

8.7.2 利用图元类画图的程序

本程序的实现要点如下：

1. 本程序设立并处理以下图元：直线（line 类），矩形（rectangle 类），三角形（triangle 类），圆（circle 类），正方形（square 类）。

2. 将每一种图元设计成一个类（class），在每一个类的定义中，除含有其构造函数外，还包含一个可将本类的图元画出来（显示出来）的公有函数 draw。

3. 由于每一个图元都要用到颜色（color）数据成员，所以设立一个基类 graphelem，它含有 protected 型数据成员 color，以及一个虚函数 draw（由于每一个类都要“draw”出属于它的类对象的图形，所以可利用函数超载的手段，在基类 graphelem 及其派生类中，共用同一个虚函数 draw）。

4. 由于不准备在基类 graphelem 的虚函数 draw 中做任何事情，所以在其原型后加上“=0”字样（而替掉函数定义体），这样的虚函数称为纯虚函数。纯虚函数不能被直接调用，它只为其派生类的各虚函数规定了一个一致的“原型规格”。

5. 含有纯虚函数的基类称为抽象基类，即是说，本程序定义的基类 graphelem 即为抽象基类。注意，不可用抽象基类 graphelem 来说明并创建它自己的对象，只有在创建其派生类对象时，才有其自身的实例伴随而生。

本例中的抽象基类 graphelem：它是各种具体图元（line，rectangle，triangle，circle 等）之共同点的一个抽象综合，通过它，再“加上”各派生类的特有成员，方可构成一个具体的图元类型。

6. 程序中至少要设立具有以下关系的六个类：

抽象基类 graphelem（含有数据成员 color，以及一个纯虚函数 draw）；

由抽象基类 graphelem 直接派生出的四个类：line，rectangle，triangle，circle；

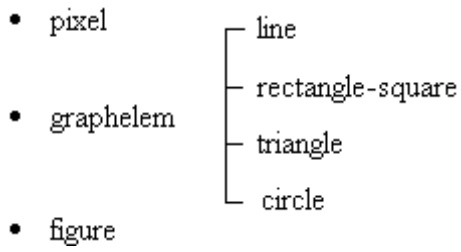
由 rectangle 直接派生出的一个类：square。

7. 由于“画”以上图元时，都要用到“点”的概念与位置，所以设立的第七个类为：pixel，它表示的是一个像素点。

8. 为了通过虚函数进行动态联编处理，需说明并使用指向基类 graphelem 的指针（注，该程序中说明了 10 个这种指针，放在一个称为 pg 的数组中，即是说，pg[0]，pg[1]，... ，pg[9]均为这种指向基类 graphelem 的指针）。而后通过这些指针的动态取值（使它们指向不同的派生类），进而利用函数调用“pg[i]->draw( )”（i=0，1，... ，9）来“画”出组成一个图形的不同图元来。

本程序中的第八个类 figure（中的 paint 成员函数）正是用来完成上述“画”图元功能的。

概括地说，本程序要自定义 pixel、graphelem、line、rectangle、triangle、circle、square 和 figure 这 8 个类（类型）。这个程序是一个压缩到最小且具有面向对象风格的 C++程序。程序中所定义的 8 个类间的相互关联关系如下：



虽然 pixel，figure **这两个类与其它六个类没有继承和派生的关系，但却有成员关系。类** pixel **的对象要作为** graphelem **及其派生类的成员和构造函数成员的参数。而类** figure **则以** graphelem **类的对象指针数组作为其数据成员。由于基类的对象指针不但可以指向基类的对象，同时也可以指向该基类的派生类对象，因此，类** figure **就间接地与五个派生（图元）类发生了关系，它的数据成员是一个由十个指针组成的数组，这十个指针可用来指向任何一类的图元对象。实际上，** figure **类以至多十个图元作为其组成元素，** figure **的一个对象就是一个至多包括十个图元的组合图形。**

```
// program 8_9.cpp
#include <iostream.h>

class pixel {                          //类 pixel，表示屏幕像素点
    int x,y;
public:
    pixel(){                           //构造函数一，无参
      x=0;      y=0;
    }
    pixel(int a, int b){               //构造函数二，参数 a、b 表示点的位置
      x=a;      y=b;
    }
    pixel(const pixel& p){             //构造函数三，参数 p 为某个已存在对象
      x=p.x;    y=p.y;
    }
    int getx(){return x;}              //获取对象的 x 值
    int gety(){return y;}              //获取对象的 y 值
};

enum colort{                           //枚举类型，用于定义颜色常量（名字）
    black,blue,green,cyan,red,magenta,brown,
    lightgray,darkgray,lightblue,lightgreen,
    lightcyan,lightred,lightmagenta,yellow,white,blink
};

class graphelem{                       //基类 graphelem（实际为抽象基类）
protected:
    colort color;                      //颜色 color
public:
```

```
    graphelem(colort col){
        color=col;
    }
    virtual void draw( )=0;              //纯虚函数 draw
};

class line:public graphelem{             //派生类 line
    pixel start, end;                    //成员为 pixel 类对象
  public:
    line(pixel sta, pixel en, colort col)
    :graphelem(col),start(sta),end(en)
    { };
    virtual void draw( );                //虚函数 draw
};

class rectangle:public graphelem{        //派生类 rectangle
     pixel ulcorner,lrcorner;
  public:
    rectangle(pixel ul,pixel el,colort col)
        :graphelem(col),ulcorner(ul),lrcorner(el)
    {};
    virtual void draw( );                //虚函数 draw
};

class circle:public graphelem{           //派生类 circle
    pixel center;
    int radius;
  public:
    circle(pixel cen,int rad,colort col) :graphelem(col),center(cen){
        radius=rad;
    };
    virtual void draw( );                //虚函数 draw
};

class triangle:public graphelem{         //派生类 triangle
    pixel pointa,pointb,pointc;
  public:
    triangle(pixel pa,pixel pb,pixel pc,colort col)
        :graphelem(col),pointa(pa),pointb(pb),pointc(pc)
    {};
    virtual void draw( );                //虚函数 draw
};

class square:public rectangle{           //派生类 square
```

```
  public:
    square(pixel ul,int lh,colort col)
        :rectangle(ul,pixel(ul.getx()+lh,ul.gety()+lh),col)
    {};
    virtual void draw( );              //虚函数 draw
};

class figure{
    //类 figure，通过它的 paint 函数可“画”出组成一个图形的各图元
    graphelem *pg[10];                 //pg 数组含有 10 个指向基类的指针
  public:
    figure (   graphelem *pg1=0,   graphelem *pg2=0,
               graphelem *pg3=0,   graphelem *pg4=0,
               graphelem *pg5=0,   graphelem *pg6=0,
               graphelem *pg7=0,   graphelem *pg8=0,
               graphelem *pg9=0,   graphelem *pg10=0   )
    {
        pg[0]=pg1;          pg[1]=pg2;
        pg[2]=pg3;          pg[3]=pg4;
        pg[4]=pg5;          pg[5]=pg6;
        pg[6]=pg7;          pg[7]=pg8;
        pg[8]=pg9;          pg[9]=pg10;
    }
    void paint( ){                     //由 paint 成员函数“画”出图形的各图元
        for(int i=0;i<10;i++)
          if(pg[i]!=0)
              pg[i]->draw( );
    };
};

void main( ){
    //说明 9 个类对象（图元），它们是构成一个图形的九个“部件”
    square  sq(pixel(40,40),120,black);                    //一个正方形
    circle  ce1(pixel(100,100),50,green);                  //两个圆
    circle ce2(pixel(100,100),2,blue);
    triangle   tr1(pixel(100,62),pixel(98,97),pixel(102,97),blue);   // 两 个 三 角 形
    triangle tr2(pixel(98,103),pixel(102,103),pixel(100,130),blue);
    rectangle re1(pixel(98,54),pixel(102,62),red);         //四个长方形
    rectangle re2(pixel(98,138),pixel(102,146),red);
    rectangle re3(pixel(54,98),pixel(62,102),red);
    rectangle re4(pixel(138,98),pixel(146,102),red);
    figure fig(&sq,&ce1,&ce2,&re1,&re2,&re3,&re4,&tr1,&tr2);
    fig.paint( );                  //调用 paint 函数，“画”出 fig 对象的 9 个图元
}
```

程序说明：

（1）figure 类的构造函数，含有 10 个参数，且该 10 个参数均为可缺省参数（被赋了缺省值的参数，调用时，相应实参可缺省）。说明 figure 类对象时，其实参可为 10 个（或少于 10 个）指向 graphelem 不同派生类对象的具体指针（注意，C++允许基类指针指向其派生类对象；而此处的各派生类对象正是准备“画”出的那些不同图元）。注意如下语法：函数定义处，若有可缺省参数的话，必须放于参数表的“最右边”，且要连续出现。

（2）paint 成员函数中，由于 pg[i]为各派生类对象的地址（由构造函数的各实参带来），从而使随后的 i 循环可“画”出所设计图形的各图元（图元数不多于 10）。

（3）main 函数中，说明了一个由九个图元构成的 figure 类对象 fig，这九个图元由调用构造函数时带去的那九个实参所确定，各实参均为指向 graphelem 不同派生类对象的具体指针（也即，对象地址）。

（4）读者可能已经发现这个程序不完整，它缺少五个派生类中虚函数 draw 的具体实现代码。有关在图形模式下作图的说明已在前面章节中提及，需要根据实际采用的 C++编译版本来具体编制 draw 函数的实现代码，而这方面的内容并不属于 C++语言的基本规则范围之内。

* 8.7.3 在 VC6 下实现利用图元类画图的程序

由于本书的所有例子都是在 VC6.0 环境下调试运行通过的，下面简要给出在 VC6.0 下实现上节“画图”程序的一种方式，以供尚有富裕学习时间与能力的学生参考，此部分并不作为本课程的教学要求，其中所涉及的许多与 Windows 编程技术相关的知识与内容，需通过今后进一步的学习来逐步掌握。

VC6.0 的“绘图”功能，是在“Windows 窗口模式”下来实现的。可使用 VC6.0 的 AppWizard，通过用户进行的一些框架程序类型以及其它一些具体指定，而不必亲自编写任何一条程序代码，首先获得一个由系统自动生成的、执行后将能在“Windows 窗口模式”下显示出字符串“Hello World!”的可执行程序。而后要在这一框架式“底层支撑程序”的基础上，编写并添加进行“画图”逻辑处理的所谓“上层程序部分”。

可按照如下四个步骤来进行。

步 1：使用 VC6 的 AppWizard 自动生成一个典型的“Hello World”框架程序（程序框架）。

生成这一种框架程序的操作步骤为：File => New => Projects => Win32 Application =>（右上处）“Project name”域中填入如“9_7 VCDraw”，“Location”框内，添入你自用的文件夹名 => OK => 选第 3 种类型：“A Typical "Hello World" application” => Finish => OK。

步 2：在所生成的框架程序基础上，增加相关的“绘图”处理类、以及处理 WM_PAINT 消息的“重画”处理函数 MyPaintProFunc，并将它们组织在同一个头文件 -- 如“MyDraw3.h”之中。

步 3：在所生成的那一主要文件“9_7 VCDraw.cpp”的开头处（最初的两个#include 行之下），加入如下的一行：

```
#include "MyDraw3.h"   //新增头文件
  //内容为本例相关的“绘图”处理类、以及处理 WM_PAINT 消息的“重画”处理函数 MyPaintProFunc
```

步 4：在所生成的那一主要文件“9_7 VCDraw.cpp”的 WndProc 函数的 switch 语句中，将“case WM_PAINT:”处理分支内的“GetClientRect(hWnd, &rt);”与“EndPaint(hWnd, &ps);”

之间的“ DrawText(hdc, szHello, strlen(szHello), &rt, DT_CENTER); ”语句替换为如下的一个函数调用句：

```
MyPaintProFunc(hWnd, hdc, rt);
```

//调用本例的“绘图”处理函数 MyPaintProFunc（处于“ MyDraw3.h ”文件中）

通过上述的步 1 将自动生成一个“ Hello World ”框架程序（也称程序框架），它含有十多个文件，这些文件包括“ Source Files ”、“ Header Files ”以及“ Resource Files ”等，该程序已经是一个可执行程序，执行它将在“ Windows 窗口模式 ”下显示出（确切地说是绘制或画出）字符串“ Hello World! ”。

注意，执行所生成程序后进入的是“ Windows 窗口模式 ”。与“ DOS 窗口模式 ”不同，该模式支持鼠标操作，是以“消息”进行驱动执行的，而且还自动为用户提供了可用的“ File ”和“ Help ”菜单以及一些简单的菜单命令项，以及可用于进行窗口“极大化”、“极小化”和“关闭”的鼠标按钮。

步 2 到步 4 是要在自动生成的那一框架程序的基础上，主要通过与其中的“ .cpp ”文件进行“衔接交融”，进而把那些与绘图逻辑有关的程序代码添加进去。至于添加代码中涉及到的 Windows 编程技术与所使用的相关语句（或函数等）则还需要通过学习一些专门的有关书籍与资料方可掌握与应用。

还有一点需要注意，通过上述步 1 所生成的框架程序的“ .cpp ”文件主要由以下 5 个函数来构成：WinMain、MyRegisterClass、InitInstance、WndProc、About。其中的 WinMain 等同于“ DOS 窗口模式 ”下的 main 函数，它首先通过调用 InitInstance 来进行本应用的初始化工作，而后进入一个消息循环，循环检查、发送并处理本应用的各种消息，直至用户对此次的应用进行了“关闭”为止。由于已经有了主函数 WinMain，原来所编程序中的 main 函数就要“降级”成为一个被调用的用户自定义函数，我们命名该函数为“ MyPaintProFunc ”，并在其中说明 9 个类对象（图元）以构成所画图形的九个“部件”，而后通过调用成员函数完成所需的“画图”功能。

经过上述的步 1 至步 4 后，“ 9_7 VCDraw.cpp ”文件的大致“骨架”内容如下：

```
// 9_7 VCDraw.cpp : Defines the entry point for the application.
//
#include "stdafx.h"
#include "resource.h"
#include "MyDraw3.h"    //新增头文件
    //内容为本例相关的“绘图”处理类、以及处理 WM_PAINT 消息的“重画”处理函数
MyPaintProFunc
    …
int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE  hPrevInstance,
                     LPSTR      þCmdLine,
                     int        nCmdShow)
{
    …
}
ATOM MyRegisterClass(HINSTANCE hInstance)
{
    …
}
```

```
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    ...
}
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
    ...
    switch (message)
    {
        case WM_COMMAND:
            ...
            break;

        case WM_PAINT:
            hdc = BeginPaint(hWnd, &ps);
            // TODO: Add any drawing code here...
            RECT rt;
            GetClientRect(hWnd, &rt);
            MyPaintProcFunc(hWnd, hdc, rt);
          //调用本例的“绘图”处理函数 MyPaintProcFunc （处于“MyDraw3.h”文件中）
            EndPaint(hWnd, &ps);
            break;
        case WM_DESTROY:
            ...
            break;
        default:
            ...
    }
   return 0;
}
LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM
lParam)
{
    ...
}
```

以下为新增头文件“MyDraw3.h”的内容。

```
//MyDraw3.h    -- 新增头文件，与“9_7 VCDraw.cpp”相呼应。内容为本例相关的
//“绘图”处理类、以及处理 WM_PAINT 消息的“重画”处理函数 MyPaintProcFunc
//*********************************************************************
class pixel {
    …                    //省略号“…”意味着此处的代码与原来的完全相同
};
enum colort{
```

```
    …
};
class graphelem{
    …
public:
    …
    virtual void draw(HWND hWnd)=0;          //注意，VC6 下的 draw 函数需要参数
};
COLORREF get_rgb(int col){
    //函数 get_rgb，将颜色常量 col 转换成 VC6 绘图时所需的 RGB 颜色值返回
    COLORREF rgb;
    switch (col){
    case red:
        rgb   =RGB(255,0,0);          break;    //红色 red 对应于 RGB(255,0,0)
    case  green:
        rgb   =RGB(0,255,0);          break;    //绿色 green 对应于 RGB(0,255,0)
    case blue:
        rgb   =RGB(0,0,255);          break;
    case cyan:
        rgb   =RGB(0,255,255);        break;
    case white:
        rgb   =RGB(255,255,255);      break;
    case magenta:
        rgb   =RGB(255,0,255);        break;
    case yellow:
        rgb   =RGB(255,255,0);        break;
    case lightgray:
        rgb   =RGB(192,192,192);      break;
    case darkgray:
        rgb   =RGB(128,128,128);      break;
    default:
        rgb   =RGB(0,0,0);            break;
    }
    return (rgb);
}
class line:public graphelem{
    …
  public:
    …
    virtual void draw(HWND hWnd);            //虚函数 draw，也需要参数
};
void line::draw( HWND hWnd){                 //类 line 之虚函数 draw 的类体外定义
    HDC hdc=GetWindowDC(hWnd);               //获取设备描述表 DC
    HPEN myPen;                              //说明画笔对象
```

```
        myPen=CreatePen( PS_SOLID, 2, get_rgb(graphelem::color)  );  //设置画笔属性
        HPEN pOldPen = (HPEN)SelectObject( hdc,myPen );      //添加画笔对象（到 DC 中）
        MoveToEx(hdc,start.getx(),start.gety(),NULL);              //移至线始点处
        LineTo(hdc,end.getx(),end.gety());                 //画至线终点处（使用新设置的画笔属性）
        (HPEN)SelectObject( hdc,pOldPen );                 //恢复原画笔（各属性）
        ReleaseDC(NULL,hdc);                               //释放所获取到的那一设备描述表
}
class rectangle:public graphelem{
        …
public:
        …
        virtual void draw(HWND hWnd);                      //虚函数 draw
};
void rectangle::draw(HWND hWnd){                           //类 rectangle 之虚函数 draw 的类体外定义
        HDC hdc=GetWindowDC(hWnd);
        HPEN myPen;
        myPen=CreatePen( PS_SOLID, 1, get_rgb(graphelem::color)  );  //设置画笔属性
        HPEN pOldPen = (HPEN)SelectObject( hdc,myPen );
        Rectangle(hdc, ulcorner.getx(),ulcorner.gety(),
        lrcorner.getx(),lrcorner.gety());                  //“画”出所需矩形来
        (HPEN)SelectObject( hdc,pOldPen );
        ReleaseDC(NULL,hdc);
}
class circle:public graphelem{
        …
    public:
        …
        virtual void draw(HWND hWnd);                      //虚函数 draw
};
void circle::draw(HWND hWnd){                              //类 circle 之虚函数 draw 的类体外定义
       HDC  hdc=GetWindowDC(hWnd);
        HPEN myPen;
        myPen=CreatePen( PS_SOLID, 2, get_rgb(graphelem::color)  );  //设置画笔属性
        HPEN pOldPen = (HPEN)SelectObject( hdc,myPen );
        Ellipse(hdc,center.getx()-radius,center.gety()-radius,
                    center.getx()+radius,center.gety()+radius);          //“画”出所需圆来
        (HPEN)SelectObject( hdc,pOldPen );
        ReleaseDC(NULL,hdc);
}
class triangle:public graphelem{
        …
public:
        …
        virtual void draw(HWND hWnd);                      //虚函数 draw
```

```
};
void triangle::draw(HWND hWnd){                    //类 triangle 之虚函数 draw 的类体外定义
    HDC hdc=GetWindowDC(hWnd);
    HPEN myPen;
    myPen=CreatePen( PS_SOLID, 2, get_rgb(graphelem::color)  );   //设置画笔属性
    HPEN pOldPen = (HPEN)SelectObject( hdc,myPen );
    MoveToEx(hdc,pointa.getx(),pointa.gety(),NULL);
    LineTo(hdc,pointb.getx(),pointb.gety());
    LineTo(hdc,pointc.getx(),pointc.gety());
    LineTo(hdc,pointa.getx(),pointa.gety());       //“画”出所需三角形来
    // Restore the old pen to the device context
    (HPEN)SelectObject( hdc,pOldPen );
    ReleaseDC(NULL,hdc);
}
class square:public rectangle{
    …
  public:
    …
    virtual void draw(HWND hWnd);                  //虚函数 draw
};
void square::draw(HWND hWnd){                      //类 square 之虚函数 draw 的类体外定义
    HDC hdc=GetWindowDC(hWnd);
    HPEN myPen;
    myPen=CreatePen( PS_SOLID, 2, get_rgb(graphelem::color) );   //设置画笔属性
    HPEN pOldPen = (HPEN)SelectObject( hdc,myPen );
    rectangle::draw(hWnd);                         //“画”出所需正方形来
    // Restore the old pen to the device context
    (HPEN)SelectObject( hdc,pOldPen );
    ReleaseDC(NULL,hdc);
}
class figure{
    …
};
//本例处理 WM_PAINT 消息的“重画”处理函数 MyPaintProFunc（实现原 main 函数的功能）
void MyPaintProFunc(HWND hWnd, HDC hdc, RECT rt)
{                  //说明 9 个类对象（图元），它们是构成一个图形的九个“部件”
    …              //与原来 main 中的说明语句完全相同，从略。
    figure fig(&sq,&ce1,&ce2,&re1,&re2,&re3,&re4,&tr1,&tr2);
                   //一个由九个图元构成的 figure 类对象
    fig.paint(hWnd);
                   //调用 paint 成员函数，“画”出 fig 对象的 9 个图元
}
```

思 考 题

1． 什么是继承关系和派生关系？在现实社会中是怎样表现的？

2． 什么是派生类？什么是基类？两者的关系是什么？

3． 派生的方式有几种？它们之间的异同点有那些（如，不同的派生方式使基类成员在各自派生类中的存取权限有何不同）？

4． 简叙派生类中可能出现的四种成员（不可访问的成员，私有成员，保护成员，公有成员），程序中如何对它们进行使用？

5． 简述在一个派生类的创建过程中所经历的步骤和系统所作的工作。派生类的构造函数是否要对其基类数据成员、对其所含对象成员的数据成员以及其它的新增数据成员一块进行初始化？

6． 通过继承可使派生类中“拥有”一个基类的类对象；通过将类中的数据成员说明成是另一个类的类对象时，也使得在该类中“拥有”了那一个类的类对象。两者在概念和使用上的关联与区别是什么？

7． 友元关系、赋值和静态成员在继承中所遵循的原则有哪些？

8． 什么是派生过程中的二义性问题？C++是怎样处理它们的？

9． 什么是虚基类？它有什么特点和作用？

10． 什么是多态性？多态性在 C++语言中表现为什么？

11． 什么是函数重载？什么是静态联编？

12． 什么是函数超载？什么是动态联编？为什么会有动态联编？

13． 什么是虚函数？它有什么特点？使用方法是什么？

14． 什么是纯虚函数和抽象基类？使用它们的目的是什么？

15． 虚函数给程序设计带来了什么好处？

练 习 题

1． 参照“8.1 公司雇员档案的管理”一节中使用的类定义及其程序实现方法，利用继承性与派生类来管理学生教师档案：由 person（人员）类出发（作为基类），派生出 student（学生）及 teacher（教师）类；而后又由 student（学生）类出发（作为基类），派生出 graduateStudent（研究生）类。可假定这几个类各自具有的数据成员为：

person（人员）类： 姓名、性别、年龄；

student（学生）类：姓名、性别、年龄、学号、系别；

teacher（教师）类：姓名、性别、年龄、职称、担任课程；

graduateStudent（研究生）类：姓名、性别、年龄、学号、系别、导师。

为简化起见，每个类可只设立构造函数以及显示类对象数据的成员函数 print。而后编制简单的主函数，说明上述有关的类对象，并对其类成员函数进行简单使用（调用）。

2． 请用类的派生来重新组织下面的数据。

动物，哺乳动物，鸟，人，鸭子，鸟目，灵长目，猴子，节肢动物，蚯蚓，蜈蚣。

（自定义数据结构，要求能将上述动物的特性进行合理的表示）

3． 对 8.2.2 节的 program 8_2.cpp 的程序进行测试验证，以加深对基类成员在各自派生类中的存取权限及其使用方式的理解；而后对程序中所出现的违反存取权限的使用情况（语句）进行修改，并在类中添加用来显示数据成员的函数如 print，并通过调用 print 将派生类中的数据成员的当前值显示出来（而得到你所设计的某种结果）。

4． 将 8.2.3 节的 program 8_3.cpp 中的派生类 CD 改写为如下形式后（类中添加了两个对象成员 obcc 与 obcb 并通过派生类构造函数对它们进行初始化，其余部分保持不变），请问

程序执行后的输出结果是否也会改变？请解释原因。

```
class CD:public CB,public CC{
        int d;
        CC obcc;
        CB obcb;
public:
        CD(int n1,int n2,int n3,int n4)
            :CC(n3,n4),CB(n2), obcb(100+n2),obcc(100+n3,100+n4){
            d=n1;
            cout<<"CD::d="<<d<<endl;
        };
        ~CD(){cout<<"CDobj is destructing"<<endl;};
};
```

5． 自定义一个如下的日期时间类 DateTimeType**，它含有类** DateType**（参看第七章的练习题** 6**）与类** TimeType**（参看第七章的练习题** 5**）的类对象作为其数据成员，并具有所列的其他几个成员函数。而后编制主函数，说明** DateTimeType **的类对象，并对其成员函数以及二对象成员所属类的公有成员函数进行使用。**

```
class DateTimeType {              //自定义的日期时间类 DateTimeType
    DateType date;                //类 DateType 的类对象 date 作为其数据成员
    TimeType time;                //类 TimeType 的类对象 time 作为其另一个数据成员
public:
    DateTimeType(int y0=1, int m0=1, int d0=1, int hr0=0, int mi0=0, int se0=0);
        //构造函数，设定 DateTimeType 类对象的日期时间，并为各参数设置了默认值
    DateType& getDate(){ return date; }        //返回本类的私有数据对象 data
    TimeType& getTime(){ return time; }        //返回本类的私有数据对象 time
    void incrementSecond(int s);               //增加若干秒，注意“进位”问题
    void printDateTime();                      //屏幕输出日期时间对象的有关数据
};
```

注意，每一个 DateTimeType **类对象中总包含有一个** DateType **类对象（对象成员）以及一个** TimeType **类对象（对象成员），编制与实现本程序时，必须包含第七章练习题** 6 **和练习题** 5 **所定义的** DateType **与** TimeType **自定义类（类型）。**

之所以设置了公有的类成员函数 getDate **与** getTime**，是为类外如主函数处使用该类的私有数据成员** date **与** time **提供方便（否则的话，类外无法直接访问该类的私有数据成员）。另外，两成员函数返回的都为引用，为的是可将返回对象当作一个独立变量来使用（如可以继续做左值等）。例如，假设编制了如下形式的主函数：**

```
void main() {
        DateTimeType dttm1(1999,12,31,23,59,59), dttm2;
        (dttm1.getDate()).printDate();          //调用对象成员所属类的公有成员函数
        dttm1.printDateTime();
        dttm2.printDateTime();
        dttm1.incrementSecond(30);
        dttm1.printDateTime();
}
```

程序执行后，应该在屏幕上显示出类似于如下形式的结果：

```
1999-12-31      1999-12-31      23:59:59
1-1-1   0:0:0
2000-1-1        0:0:29
```

6．自定义一个如下的日期时间类 DateTimeType**，它由基类** DateType**（参看第七章的练习题** 6**）与** TimeType**（参看第七章的练习题** 5**）二者公有派生。并编制主函数，说明派生类对象，并对派生类的成员函数以及二基类的公有成员函数进行使用。**

```
class DateTimeType : public DateType, public TimeType {
public:
    DateTimeType(int y0=1, int m0=1, int d0=1, int hr0=0, int mi0=0, int se0=0);
                        //构造函数，设定派生类对象的日期时间，并为各参数设置默认值
    void incrementSecond(int s);                    //增加若干秒，注意“进位”问题
    void printDateTime();                           //屏幕输出日期时间对象的有关数据
};
```

注意，虽然没在派生类 DateTimeType **中说明数据成员，但由于公有继承了基类** DateType **与** TimeType**，所以本派生类中包含了二基类的所有数据成员，意味着派生类类对象必然是一个基类类对象（“**is a**”关系）。编制与实现本程序时，也必须包含第七章练习题** 6 **和练习题** 5 **所定义的** DateType **与** TimeType **自定义类（类型）。**

对派生类对象进行使用时，假设编制了如下形式的主函数：

```
void main() {
        DateTimeType dttm1(1999,12,31,23,59,59), dttm2;
        dttm1.printDate();              //直接调用基类的公有成员函数 printDate
        dttm1.printDateTime();
        dttm2.printDateTime();
        dttm1.incrementSecond(30);
        dttm1.printDateTime();
}
```

程序执行后，应该在屏幕上显示出类似于如下形式的结果：

```
1999-12-31      1999-12-31      23:59:59
1-1-1   0:0:0
2000-1-1        0:0:29
```

7．某商场有如下的几种货品：衬衣、鞋子、帽子、裤子、冰箱、电视、立柜、壁橱、沙发。每一种货物都有详细的说明信息。

衬衣：布料、尺寸、单价、产地、库存量、所属货柜；

鞋子：皮料、尺寸、单价、产地、库存量、所属货柜；

帽子：布料、样式（平顶或尖顶）、尺寸、单价、产地、库存量、所属货柜；

裤子：布料、尺寸、单价、产地、库存量、所属货柜；

冰箱：制冷类型、样式（二门或三门）、颜色、尺寸、单价、产地、库存量、重量、所属货柜；

电视：样式（彩色或黑白）、颜色、尺寸、单价、产地、库存量、重量、所属货柜；

立柜：木料、颜色、尺寸、单价、产地、库存量、所属货柜；

壁橱：木料、颜色、尺寸、单价、产地、库存量、所属货柜；

沙发：木料、皮料、颜色、尺寸、单价、产地、库存量、所属货柜；

对这些商品的操作有：

新商品的录入，商品的进库，商品的出库，商品的调价，所属货柜的管理，库存的统计，

总价格的计算，产地的统计。

要求自行设计数据结构，用类结构将上述的货品表示出来。在上一步的基础上，将上述的商品管理计算机化，完成操作要求的功能。

8． 将 8.5.1 节的 program 8_6.cpp 中的派生类 B 和 C 的虚拟继承改为如下形式的一般继承（其他部分保持不变）：

```
class B: public A {…};
class C: public A {…};
```

此时程序中对基类 A 中的数据成员 i 以及成员函数 showa 的使用将产生二义性，请使用“类限定”的方式对出现二义性的地方进行修改（如通过使用“B::i”以及“B::showa()”等），仍使该程序保持原功能，且执行后的显示结果仍为：

```
i=11
i,j,k,n=11,22,33,44
```

9． 利用虚函数手段，按照 3 种不同的计算方法来求出 Fibonacci 数列的第 n 项（的具体项值）并输出。具体地说，可通过在基类 baseCla 及其派生类 fib1Cla、fib2Cla 和 fib3Cla 中说明如下的同一个虚函数“virtual double fib(int n);”，来实现求 Fibonacci 数列第 n 项值并返回的 3 种不同求解方法。例如，可设计并使用已经在第 4 和第 5 章的练习中所实现的求解方法：简单变量“数据平移”法、使用数组的实现法以及使用递归函数的实现法。

下面给出具体的“实现程序骨架”。

```
class baseCla {                    //自定义的基类 baseCla
public:
    virtual double fib(int n)=0;   //基类 baseCla 中说明了一个虚函数 fib，且为纯虚函数
};
class fib1Cla:public baseCla {     //由基类 baseCla 派生出的 fib1Cla 类
public:
    virtual double fib(int n);     //派生类中说明同一个虚函数 fib（简单变量“数据平移”法）
};
class fib2Cla:public baseCla {     //派生类 fib2Cla
public:
    virtual double fib(int n);     //派生类中说明同一个虚函数 fib（使用数组的求解法）
};
…
void fun(baseCla *p, int n) {
        //自定义函数 fun，形参 p 为指向基类的指针，其对应实参
        //可为不同派生类对象的地址；n 指明要求出数列的第 n 项
    double d = p->fib(n);          //根据 p 指针值的不同，将调用不同派生类的虚函数 fib
    cout.flags(ios::scientific);
    cout.precision(15);
    cout<<"fib("<<n<<")="<<d<<endl;
}
void main() {
    fib1Cla obj1;                  //fib1Cla 类对象 obj1
    fib2Cla obj2;                  //fib2Cla 类对象 obj2
    fib3Cla obj3;                  //fib3Cla 类对象 obj3
    cout<<"------------ fib1Cla ------------"<<endl;
```

```
    fun(&obj1, 1476);              //简单变量“数据平移”求解方法，求 fib(1476)
    cout<<"------------ fib2Cla ------------"<<endl;
    fun(&obj2, 888);               //数组求解方法，求 fib(888)
    cout<<"------------ fib3Cla ------------"<<endl;
    fun(&obj3, 35);                //递归求解方法，求 fib(35)
}
```

程序执行后，屏幕显示结果为：

```
------------ fib1Cla ------------
fib(1476)=1.306989223763399e+308
------------ fib2Cla ------------
fib(888)=1.704274475850073e+185
------------ fib3Cla ------------
fib(35)=9.227465000000000e+006
```

10． 对“8.7.2 利用图元类画图的程序”增加其他你所关心与需要的图元类（如，梯形类，菱形类，椭圆类等），且在每一个类中，仍通过使用虚函数 draw，来“画”出属于它的类对象的有关图形。而后改造主函数，对新增加的图元类进行使用，“画”出你所设计的某种图形。

第九章 模板

本章介绍函数模板与类模板的定义及其使用方法。通过使用模板，可使所编程序更加紧凑，增加程序的通用性及可重用性。

模板的概念是仅在 C++语言的高版本中才引进的。模板（template）是一种参数化的类型，在有模板概念之前，C++语言的程序员热衷于涉及“类属类”（Generic class），其目标是为了实现程序代码的可重用性，通过“类属编程”使得同一结构的不同实例共用同样的代码。这样的类属数据结构包括：栈，队列，数组，矩阵，链表，二叉树，散列表和图等等。这项工作的意义是明显的，这样的类库应用很广，可以节省代码，易于维护；当然，类属类的编程也十分繁琐。

C++（高版本）的模板支持类属编程，大大简化了类属数据结构的编程工作。熟练地使用模板概念，是高级的 C++程序员的必要条件之一。

类模板是一种带参类，或说是具有共性的一组类。模板增加了以类为特征的程序模块的通用性。

例如，集合、矩阵、链表都可以设计成类。但是如果采用模板的形式，则编程的效率更高。例如，集合类，根据集合类元素的不同类型，如 int 型，char 型，float 型，point 型，complex 型等等，可以是系统类型，也可以是用户定义类型。总之对于每一种元素类型，可以定义一个集合类。利用模板的形式，以类型为参数，可以定义一个集合（set）模板，每指定一种类型作为“实参”，这个模板就成为一个特定的集合类。

函数模板也有类似的特征。

9.1 函数模板

9.1.1 函数模板的概念及说明

1．函数模板概念

通常我们设计的算法（如某一个函数体中的那些处理语句）是可以处理多种数据类型的。但目前来说，处理这样的问题，即使设计为重载函数也只是使用相同的函数名，函数体仍然要分别定义。例如下述对两个 int 型数据、double 型数据、char 型数据，...，求出其中最大值的函数例子：

```
int max (int a, int b){                    //函数 max，求两个 int 型数据的最大值
        if(a>b)    return a;
          else     return b;
}
double max (double a, double b){ //重载函数 max，求两个 double 型数据的最大值
        if(a>b)    return a;
          else     return b;
}
char max (char a, char b){                 //重载函数 max，求两个 char 型数据的最大值
        if(a>b)    return a;
          else     return b;
}
...
```

实际上，若“提取”出一个可变化的类型参数 T，上述那些函数则可以“综合”成为如下的同一个函数（模板）：

```
T max (T a, T b){
        if(a>b)    return a;
          else     return b;
}
```

这正是函数模板的概念，它实际上代表着一组函数。只是该组函数的参数类型以及返回值类型可以变化而已。

在 C++中定义一个函数模板时，只是简单地在上述那一“综合”出的具有类型参数 T 的函数之前，增加“template <class T>”，以表示下面要定义一个模板，它具有的可变化类型参数名字叫做 T（当然也可叫做别的，是一个标识符）。C++中其完整定义格式如下：

```
template <class T> T max (T a, T b) {
        if(a>b)    return a;
          else     return b;
}
```

注意，类型形参名 T 同时被用在了函数模板 max 的“返回类型”以及“形参表”之中，它们代表同一个（可变化的）类型。

如此，我们定义了一个简单的函数模板，当我们为类型参数 T 指定为某一具体的类型时，函数模板 max 就是一个具体求较大元的 max 函数了。当然，类型参数 T 的“实参”,应该是一个可以进行大于运算“ > ”的系统定义的类型（如

int，float，char 等），或已重载了运算符“ > ”的用户自定义类型。

概括地说，利用函数模板（带类型参数的函数），一次就可定义出具有共性（除类型参数外，其余全相同）的一组函数（可以处理多种不同类型数据的函数）。

可以使用函数模板的形式来定义的例子很多，例如求绝对值的函数，求最大元的函数，求中值的函数，排序的函数等等，它们都可以针对不同类型的数据，对应出同名的多个函数的重载。这些同名的重载函数，更方便的处理方法当然还是定义成一个以参加操作的数据之类型为模板参数的函数模板。

2．函数模板说明

函数模板定义的一般格式为：

template < <模板参数表> > < 函数定义 >；

其中：

template：关键字。指明为函数模板或类模板。

模板参数表：用尖括号括起来，一个或多个模板参数，用“ , ”分开。

模板参数：其格式为 class<类名>，其中的类名是一个标识符，该类名对应的实参类型可以是系统预定义类型如 int、char 等，也可以是用户自定义类型。

函数定义：与一般函数定义一样：

<返回类型><函数名>(<参数表>){<函数体>}；

应注意的是，在模板参数表中的类型参数应出现在上述的“ <返回类型> ”或“ <参数表> ”或“ <函数体> ”之中（否则将没有可变性，只能定义出一个具体的函数）。

关于函数模板的使用有下面一些说明：

（1）函数模板可以像一般函数那样来直接使用，用户只需给出具体的实参，而系统则根据所提供的实参信息，分析确定出函数模板的各“ 类型形参名 ”所对应的具体类型，从而将其实例化为一个具体的函数（也称模板函数），而后再去调用执行。

（2）函数模板在被调用时与同名的函数调用没有什么区别，那么系统是如何处理的呢？例如编译器扫描到函数调用表达式 max(c1,c2)时：

① 首先搜索程序说明中是否有参数表恰与 max(c1,c2)之参数表完全相同的同名函数，如果有，就调用此函数代码付诸执行。否则执行下一步。

② 检查是否有函数模板，经适当实例化成为参数匹配的同名函数。如果有，调用此实例化的模板函数代码付诸执行。否则执行下一步。

③ 检查是否有同名函数，可经参数的自动转换后实现参数匹配。如果有，调用该函数代码付诸执行。

如果三种情况都未找到匹配函数，则按出错处理。

值得注意的是，模板函数调用时，与一般函数不同之处在于，它不允许类型的转换。也就是说，调用函数的实参表在类型上必须与某一实例化了的模板函数的函数形参表完全匹配。相反，对于一般的函数定义，系统将进行实参到形参类型的自动转换。

（3）模板函数也可以重载。同样，重载的条件是二同名模板函数必须有不同的参数表。

9.1.2 函数模板应用举例

1．函数模板例 1

本例定义一个函数模板 max，而后对它进行不同的调用。

```
// program 9_1.cpp
#include <iostream.h>
template <class T> T max (T a, T b){     //函数模板 max 的定义
   if(a>b)    return a;
      else    return b;
}
void main() {
   int i1=-11, i2=0;
  double  d1,  d2;
   cout<<max(i1,i2)<<endl;            //由实参 i1，i2，系统可确定“类型形参 T”对应于 int
   cout<<max(23,-56)<<endl;           //由实参 23，-56，系统可确定“类型形参 T”对应于 int
   cout<<max('f',  'k')<<endl;        //由实参'f'，'k'，系统可确定“类型形参 T”对应于 char
  cout<<"d1,d2=?  ";
   cin>>d1>>d2;
   cout<<max(d1,d2)<<endl;            //由实参 d1，d2，系统可确定“类型形参 T”对应于 double
  //cout<<"max(23,-5.6) = "<<max(23,-5.6)<<endl;
            //出错!  参数具有二义性，“double”?  “int”?
            //模板函数调用时，不进行实参到形参类型的自动转换！
}
```

程序执行后的显示结果如下：

```
0
23
```

```
k
d1,d2=? 1.23 5.68
5.68
```

2．函数模板例 2

本例定义一个函数模板与一个函数，它们都叫做 min，C++允许这种函数模板与函数同名的所谓重载使用方法。但注意，在这种情况下，每当遇见函数调用时，C++编译器都将首先检查是否存在重载函数，若匹配成功则调用该函数，否则再去匹配函数模板。

```
// program 9_2.cpp
#include <iostream.h>
#include <string.h>
//函数模板 min，具有类型参数 type，min 的功能为求出 a 与 b 中的小者返回
//（注意 type 型的 a 与 b 要能够进行“<”比较运算!）
template <class type> type min (type a, type b){
    return (a<b?a:b);
}
//函数 min，它有两个字符串型的参数，不能直接使用“<”来进行比较，
//要通过函数 strcmp 来实现
char* min (char* a, char* b){
    return (strcmp(a,b)<0?a:b);                    //返回 a，b 串中的小者
}
void main() {
    cout<<min(3,-10)<<endl;                        //两个 int 数据求最小
        //（编译器首先匹配重载函数 min，不成功后又去匹配并使用函数模板 min）
    cout<<min(2.5,99.5)<<endl;                     //两个 double 数据求最小（将使用函数模板 min）
    cout<<min('m','f')<<endl;                      //两个 char 数据求最小（将使用函数模板 min）
    char* str1="The C program", * str2="The C++ program";
        //两个字符串求最小（与重载函数 min 匹配成功，使用重载函数!）
    cout<<min(str1, str2)<<endl;
}
```

程序执行后的显示结果如下：

```
-10
2.5
f
```

The C program

3．函数模板例 3

本例定义两个函数模板，它们都叫做 sum，都使用了一个类型参数 Type，但两者的形参个数不同，C++允许使用这种函数模板重载的方法。另外注意，函数（模板）的参数表中允许出现与类型形参 Type 无关的其它类型的参数，如此例中的“ int size ”。

```
// program 9_3.cpp
#include <iostream.h>
//第一个函数模板 sum，使用了一个类型参数 Type，具有两个形参
//本函数模板 sum 的功能为：求出 array 数组的前 size 个元素之和并返回
template <class Type> Type sum (Type * array, int size ){
        Type total=0;
        for (int i=0;i<size;i++)            //累加前 size 个数到 total
                total+=*(array+i);
        return total;
}
//第二个函数模板 sum，使用了一个类型参数 Type，但具有三个形参
//本函数模板 sum 的功能为：求出 a1 数组与 a2 数组的前 size 个元素之和并返回
template <class Type> Type sum (Type * a1, Type * a2, int size ){
        Type total=0;
        for (int i=0;i<size;i++)            //累加 a1 数组与 a2 数组的前 size 个数到 total
                total+=a1[i]+a2[i];
        return total;
}
void main() {
  int a1[10]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int a2[8]={2, 4, 6, 8, 10, 12, 14, 16};
  double af[10]={1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9, 11};
    cout<<sum(a1,10)<<endl;           //求出 a1 数组前 10 个元素之和并输出
    cout<<sum(af,10)<<endl;           //求出 af 数组前 10 个元素之和并输出
   cout<<sum(a1,a2,8)<<endl;          //求出 a1 数组与 a2 数组的前 8 个元素之和并输出
}
```

程序执行后的显示结果如下：

```
55
```

60.5

108

9.2 类模板

9.2.1 一个队列类模板

队列（queue）和栈（stack）都是特殊的线性数据结构，栈是只允许在表的一端对表中的数据进行存取（“先进后出”），而队列则是允许在表的一端存入而在另一端取出，即所谓“先进先出”。这种数据结构在程序设计中经常被采用，队列中项的类型可以有所不同，但队列的基本操作是一致的，因此可以把它设计为带有类型参数的类模板。

本例定义一个处理队列的类模板 Queue，它含有一个类型形参 keytype，以指出队列中的数据项均为这种 keytype 可变化类型的。

队列数据放于作为类成员的动态数组 queue 之中，在构造函数中，将通过 new 来生成该动态数组，动态数组 queue 的大小由类的私有数据成员 Maxsize 之值来确定。

```
// program 9_4.cpp
#include <iostream.h>
#include <process.h>
template <class keytype> class Queue {          //类模板 Queue，使用类型形参 keytype
      int  Maxsize;                             //队列的大小
      int front,rear;                           //元素放在 queue[front+1]到 queue[rear]之中
      keytype *queue;                           //动态数组 queue，用来存放队列数据
           //大小由 Maxsize 之值来确定，但此例并不循环使用数组空间
public:
      Queue (int size) {                        //构造函数，生成动态数组来存放队列数据
           Maxsize=size;
          queue=new  keytype[Maxsize];
           front=rear=-1;                       //意味着队列为空
        };
      int IsFull () {                           //是否队列已满
           if  (rear==Maxsize-1) return 1;
               else return 0;
```

```
    };
    int IsEmpty () {                          //是否队列为空
        if (front==rear) return 1;
            else return 0;
    };
    void Add(const keytype);                  //往队列尾增添数据
    keytype Delete(void);                     //从队列首删取数据
};
//函数成员 Delete 在类体外定义，函数名前要加类限定符“ Queue<keytype>:: ”
template <class keytype> keytype Queue<keytype>::Delete(void){
    if (IsEmpty()){
        cout << "the queue is empty"<<endl;
        exit (0);
    }
    return queue [++front];
}
//函数成员 Add 在类体外定义，实际上相当于一个函数模板
template <class keytype> void Queue<keytype>::Add(const keytype item){
   if (IsFull()) cout << "the queue is full"<<endl;
       else queue[++rear]=item;
};
void main() {                                 //创建类模板 Queue 的实例对象并对它们进行使用
    int i=0;
    Queue<int> Qi(10);                        //以 int 取代类型形参 keytype 后成为一个具体类
    Queue<double> Qf1(10),Qf2(10);
    while (!Qi.IsFull()) {                    //Qi 中能盛放 10 个数，while 循环体将被执行 10 遍
        Qi.Add(2*i++);                        //往 Qi 中加入 10 个数: 0,2,4,6,8,10,12,14,16,18
        Qf1.Add(3.0*i);                       //往 Qf1 中加入 10 个数: 3,6,9,12,15,18,21,24,27,30
    }
    for (i=0; i<4; i++) {                     //四次循环，每次总先往 Qf2 的队列尾部加入两个数
                                              //而后又从首部删取一个数并输出
        Qf2.Add(4.5*Qi.Delete());
            //从 Qi 队列首删取一元素，并乘以 4.5，而后将其加入到 Qf2 的队列尾部
            //四次循环每次往 Qf2 的队列尾部加入 1 个数：0*4.5, 2*4.5, 4*4.5, 6*4.5
```

```
        Qf2.Add(Qf1.Delete()/2.0);
            //从 Qf1 队列首删取一元素，并除以 2.0，而后将其加入到 Qf2 的队列尾部
            //四次循环每次往 Qf2 的队列尾部加入 1 个数：3/2.0, 6/2.0, 9/2.0, 12/2.0
        cout <<Qf2.Delete()<<endl;
            //从 Qf2 队列首删取一元素，并将其显示出来：0*4.5, 3/2.0, 2*4.5, 6/2.0
    }
}
```

程序执行后的显示结果如下：

```
0
1.5
9
3
```

这里给出了一个队列类模板，在 main（）函数中指出了它的使用方法，关于类模板的说明格式，在下一节介绍。

一般模板的设计受到重视是因为其通用性和可重用性强。严格地说，这里给出的 Queue 类模板有一个明显的缺点：就是它不仅存储空间是事先限定的，而且这一有限的空间的利用也不充分。例如 Qi(10)，这个队列的容量为 10，它至多存入 10 项，而且即使取走若干项有了空闲空间后也不可重新进行使用。若稍加改造，使存取数据时首先通过对下标进行模 Maxsize 的运算，则可实现循环使用动态数组 queue 空间的功能，我们把它作为一个练习题。

当然，如果用链表代替数组实现队列，就没有溢出问题了。

9.2.2 类模板说明

利用类模板（带类型参数或普通参数的类），一次就可定义出具有共性（除类型参数或普通参数外，其余全相同）的一组类。即是说，与使用函数模板的优越性相似，通过使用类模板，可使得所定义类中的某些数据成员、某些成员函数的参数、某些成员函数的返回值都可以是任意类型的（包括系统预定义类型以及用户自定义类型）。也可以这样说，通过类模板可将程序所处理对象（数据）的类型参数化，从而使得同一段程序可用于处理多种不同类型的对象（数据），提高了程序的抽象层次与可重用性。

类模板的说明就是一个带有模板参数的类定义，其格式为：

```
template < <模板参数表> > class < 类模板名 >
{ <类模板定义体> };
```

说明：

template：关键字，指明本说明为类模板说明或函数模板说明。

模板参数表：用尖括号<,>括起来，用来说明若干个类型形参或普通形参。说明类型形参时，使用“ class <类型形参名> ”的方式，说明普通形参时，使用“ <类型> <普通形参名> ”的方式（注意函数模板不许使用普通形参 !）。

class：关键字 class 指出定义的是类模板。

类模板名：标识符。

类模板定义体：它实际上是一个类的定义体，在定义体中，要以类模板参数作为某一类或类型名来使用(否则将没有可变性,只能定义出一个具体的类)。

例如：

```
template <class T, int i> class TestClass {
        //具有类型形参 T 与普通形参 i 的类模板 TestClass
    T  buffer[i];
        //T 类型的数组 buffer，数组大小随普通形参 i 的大小而变化
    ...
}
```

下面是与类模板有关的几个问题：

（1）类模板的实例化。

不能使用类模板来直接生成对象，因为其类型参数是不确定的，故须首先对模板参数指定“ 实参 ”，实例化的形式为：

<类模板名> < <具体实参表> >

例如：

```
TestClass<char, 10>
    //以 char 取代类型形参 T 并以 10 取代普通形参 i 后形成的具体类
TestClass<double, 8>
    //以 double 取代类型形参 T 并以 8 取代普通形参 i 后形成的具体类
```

利用类模板说明类对象时，要随类模板名同时给出对应于类型形参或普通形参的具体实参，从而将类模板实例化为某个具体的类（也称为模板类，而后通过该具体类来说明对象）。

其中类型形参的相应实参要为类型名，而普通形参的相应实参必须为一个常量。

TestClass<double, 8> dobj1, dobj2；

即是说明了两个经实例化后的那一个类的对象 dobj1 与 dobj2。

由此可以看出，每个类模板的实例就是一个具体的类，由类模板的实例可

以说明或创建该类的对象。

也可以用由用户定义的类（类型）来进行实例化。例如，假设我们已经定义了类 complex，类 point，于是可以用 complex 和 point 作为“实参”来对于类模板 TestClass 进行实例化：

```
TestClass <complex, 15>
TestClass <point, 20>
```

这两个实例化类的元素是复数——类 complex 的对象，点——类 point 的对象。

（2）类模板的成员函数既可以在类体内进行说明（自动按内联函数处理），也可以在类体外进行说明（定义）。若在类体外定义类模板的成员函数时（注意，类模板的成员函数实际上是一个函数模板），其定义格式类似于函数模板的定义。另外要记住应在函数（模板）名前加上类限定符（在类体外定义成员函数的要求）。在类体外定义类模板成员函数的一般格式如下：

```
template <模板参数表> 函数类型 类模板名 <模板参数名字表>::成员函数名 ( 函数参数表 ) {
    ...    //成员函数的函数体
};
```

上述的“模板参数名字表”来自于“模板参数表”，由“甩掉”说明部分的“类型”而得，是对类型形参或普通形参的使用。而“类模板名<模板参数名字表>::”所起的作用正是在类体外定义成员函数时在函数名前所加的类限定符！

例如，对具有一个类型参数 T 和一个普通参数 i 的类模板 TestClass，在类体外定义其成员函数 getData 时的大致样式如下：

```
template <class T, int i> T TestClass<T,i>::getData(int j) {
    ...    //成员函数的函数体
};
```

其中的“TestClass<T,i>::”所起的作用正是在类体外定义成员函数时在函数名前所加的类限定符！

（3）类模板的参数。类模板是对于类的进一步抽象，被称为类属类，可以说是类的类。模板的定义体现了这个集合中的类的共性，而模板参数则标识出集合中类的区别。例如矩阵可以定义为一个类，而一个反映矩阵特征的类模板，可以一次定义出分别由不同类型的元素组成的矩阵类。

但实际的类模板可能是依赖于两个类型参数的。例如，可以定义一种向量或数组，它可以有不同类型的分量，同时也可以有不同类型的下标（如同 Pascal 语言中的数组），于是这样的类模板可以有两个类型参数。

可以定义一种类模板，表示不同维数，不同元素类型的矩阵类。于是这样

的一个类模板的模板参数可能除了一个类型之外，还有两个整型参数，等等。

总之，设计 C++语言类模板时，首先设计出所需要的若干个模板参数，并规定各参数的使用含义就成为至关重要的了。

9.2.3 使用类型参数和普通参数的类模板

本例的类模板定义中使用了类型参数以及普通参数（非类型参数）。类型形参的相应实参为类型名，而普通形参的相应实参必须为一个常量。

```
// program 9_5.cpp
#include "iostream.h"
#include "string.h"
template <class T, int i> class TestClass {
      //类模板 TestClass，使用类型形参 T 以及普通形参 i
public:
        T buffer[i];              //T 类型的 buffer 数组，数组大小由普通形参 i 的值指定
        T getData(int j);         //getData 函数，负责返回 T 类型的 buffer（数组）的第 j 个分量
};
template <class T, int i>T TestClass<T,i>::getData(int j) {    //在类体外定义成员函数 getData
    return  *(buffer+j);          //返回 buffer（数组）的第 j 个分量
};
void main() {                     //创建类模板 TestClass 的实例对象并对它们进行使用
      TestClass<char, 5> ClassInstA;
      char cArr[6]="abcde";
      strcpy(ClassInstA.buffer, cArr);            //为类对象 ClassInstA 的 buffer 数组赋值
      for(int i=0; i<5; i++) {
      //调用类成员函数 getData 取出 buffer 数组各元素值并输出：a  b  c  d  e
            char res=ClassInstA.getData(i);
            cout<<res<<"  ";
      }
      cout<<endl;
      TestClass<double, 6> ClassInstF;
      double fArr[6]={12.1, 23.2, 34.3, 45.4, 56.5, 67.6};
      for(i=0; i<6; i++)
           ClassInstF.buffer[i]=fArr[i]-10;       //为类对象 ClassInstF 的 buffer 数组赋值
```

```
    for(i=0; i<6; i++) {
    //调用 getData 取出 buffer 数组各元素值并输出：2.1   13.2   24.3   35.4   46.5   57.6
        double res=ClassInstF.getData(i);
        cout<<res<<"   ";
    }
    cout<<endl;
}
```

程序执行后的显示结果如下：

```
a   b   c   d   e
2.1   13.2   24.3   35.4   46.5   57.6
```

通过该示例看到了类模板定义中所使用的类型参数以及普通参数两者的使用含义。若只使用其中的一种参数，所定义的类模板的通用性就会降低。读者可按照如下形式的类模板定义，作为一个练习，编程并上机进行调试使用，而后对这三个相关类模板的性能进行评价与比较。

```
template <class T> class TestClass {       //类模板 TestClass，仅使用类型形参 T
public:
    T  buffer[10];            //T 类型的数据成员 buffer 数组，大小固定为 10
    T  getData(int  j);       // getData 函数，返回 T 类型的 buffer 数组的第 j 个分量
};
template <int i> class TestClass {          //类模板 TestClass，仅使用普通形参 i
  public:
    int  buffer[i];           //使 buffer 的大小可变化，但其类型则固定为 int
    int  getData(int  j);     //int 型的 getData，返回 buffer 数组的第 j 个分量
};
```

9.3 关于类模板的若干问题说明

模板是 C++语言中一个非常重要的高级概念，它相当于 Ada 语言中的类属类（generic class）。不具备模板功能的低版本 C++语言的用户，曾花费大量的努力实现类属类的功能。现在有了模板，为 C++语言的高级编程提供了巨大的方便。另一方面，类模板概念是类的概念的又一层抽象，自然会产生一系列的相关问题需要解决，本节将涉及其中比较重要的一些方面。

9.3.1 静态成员及友元

1．类模板的静态成员

类模板也允许有静态成员。实际上，它们是类模板之实例化类的静态成员。也就是说，对于一个类模板的每一个实例化类，其所有的对象共享其静态成员。例如：

```
template<class T>class C{
  static T t；     //类模板的静态成员 t
};
```

类模板的静态成员在模板定义时是不会被创建的，其创建是在类的实例化之后。如：

```
CA<int>aiobj1, aiobj2；
CA<char>acobj1, acobj2；
```

对象 aiobj1 和 aiobj2 将共享实例化类 CA<int>的静态成员 int t，而对象 acobj1，acobj2 将共享实例化类 CA<char>的静态成员 char t。

2．类模板的友元

类模板定义中允许包含友元。我们讨论类模板中的友元函数，因为说明一个友元类，实际上相当于说明该类的成员函数都是友元函数。

① 该友元函数为一般函数，则它将是该类模板的所有实例化类的友元函数。

② 该友元函数为一函数模板，但其类型参数与类模板的类型参数无关。则该函数模板的所有实例化（函数）都是类模板的所有实例化类的友元。

③ 更复杂的情形是，该友元函数为一函数模板，且它与类模板的类型参数有关。例如，函数模板可以用该类模板作为其函数参数的类型。在友元函数模板定义与相应类模板（的类型参数）有关时，该友元函数模板的实例有可能只是该类模板的某些特定实例化（而不是所有实例化）类的友元。

9.3.2 特例版本

类模板的使用十分方便，但大多数类模板不能任意进行实例化。也就是说类模板的类型参数往往在实例化时不允许用任意的类（类型）作为“实参”。

C++语言中没有对模板的“实参”类型进行检查的机制，它仅仅是通过实际操作中发生语法错误时，才能指出实例化的错误。

模板的“实参”不当，主要会在实例化后的函数成员调用中体现出来，例

如：

```
template <class T> class stack {                    //栈中元素类型为 T 的 stack 类模板
    T num [MAX];                                    //num 中存放栈的实际数据
    int top;                                        //top 为栈顶位置
public:
    stack () { top=0; }                             //构造函数
    void push (T a) { num[top++]=a; }               //将数据 a “压入”栈顶
    void showtop(){                                 //  显示栈顶数据
        //模板中通用的 showtop，显示栈顶的那一个 T 类型的数据
        //（必须为可直接通过运算符“<<”来显示的数据）
        if (top==0) cout << "stack is empty!"<< endl;
            else cout<<"Top_Member:"<<num[top-1]<<endl;
    }
};
```

在上面的类模板 stack 中，以下的实例化都是可行的：stack<int>i1,i2; stack<char>c1,c2; stack<float>f1,f2; 等等。但如果采用用户定义类型而又未在该类中对运算符“<<”进行重载时，就会产生问题，例如：

```
stack<complex>com1,com2;
```

由于在执行 com1.showtop()函数时，将需要对 complex 类型的数据 num[top-1]通过使用运算符“<<”来进行输出，而系统和用户都没有定义过这种操作，因此，类模板 stack 的实例化 stack<complex>就是不可行的了。

如果用户在上述情况下，需要使 stack<complex>可行，可有两个办法。

① 对于类 complex 追加插入运算符“<<”的重载定义；

② 也可在类模板 stack 的定义中增加一个“特例版本”（也称“特殊版本”）的定义。例如在上例中，可以在类模板定义之后给出如下形式的特例版本：

```
void stack<complex>::showtop(){
    //专用于 complex 类型的 showtop（专门补充的“特例版本”），显示栈顶的
    //那一个 complex 型数据。其中的 stack<complex>为一个实例化后的模板类。
    if (top==0)    cout << "stack is empty!"<< endl;
    else
        cout<<"Top_Member:"<<num[top-1].get_r()<<", "<<num[top-1].get_i()<<endl;
}
```

此处假设自定义的复数类型 complex 中具有公有的成员函数 get_r()以及 get_i()，用于获取复数的实部和虚部。如此，当实例化 stack<complex>时将按

该特例版本的定义进行。

概括地说，当处理某一类模板中的可变类型 T 型数据时，如果处理算法并不能对所有的 T 类型取值做统一的处理，此时可通过使用专门补充的所谓特例版本来对具有特殊性的那些 T 类型取值做特殊处理。

③ 也可以对函数模板，或类模板的个别函数成员补充其“ 特例版本 ”定义。例如，可将该例的 showtop 功能进一步划分，让 showtop 调用另一个新增加的 show 函数，而由 show 函数具体考虑对两种情况的处理：一种处理可直接通过运算符 “ << ” 来显示的数据，另一种 “ 特例版本 ” 专用于处理 complex 类型的数据。此时的 show 函数的编码如下：

```
template<class T> void stack<T>::show(T t){
        //通用的 show，可直接通过 “ << ” 来显示的 T 型数据
    cout<<"t="<<t<<endl;
}
void stack<complex>::show (complex com){
        //专用于 complex 类型的 show （“ 特例版本 ”）
    cout<<"com=("<<com. get_r()<<','<<com. get_i()<<')'<<endl;
}
```

读者可编制完整的程序，通过调用与上述栈类模板 stack 相关的特例版本 showtop 以及 show，来实现对 complex 类型数据的具体处理，我们把它留作一个练习。

9.3.3 按不同方法来派生类模板

通过继承可以产生派生类。通过继承同样可产生派生的类模板。可以使用多种不同的方式来派生出类模板，如，可使用一般类作基类来派生出类模板，也可使用类模板作基类而派生出新的类模板。下面列举几种常用的派生类模板的 “ 构架 ” 方式：

（1）一般类（其中不使用类型参数的类）作基类，派生出类模板（其中要使用类型参数）。

```
class CB {          //CB 为一般类（其中不使用类型参数），它将作为类模板 CA 的基类
    ...
};
template <class T> class CA:public CB {
                    //被派生出的 CA 为类模板，使用了类型参数 T，其基类 CB 为一般类
```

```
    T t;          //私有数据为 T 类型的
  public:
    ...
};
```

（2）类模板作基类，派生出新的类模板。但仅基类中用到类型参数 T（而派生的类模板中不使用 T）。

```
template <class T> class CB {
                  //CB 为类模板（其中使用了类型参数 T），它将作为类模板 CA 的基类
    T t;          //私有数据为 T 类型的
  public:
    T gett(){   //用到类型参数 T
        return t;
    }
    ...
};
template <class T> class CA:public CB<T> {
                  //CA 为类模板，其基类 CB 也为类模板。注意，类型参数 T
                  //将被“传递”给基类 CB，本派生类中并不使用该类型参数 T
    double t1; //私有数据成员
  public:
    ...
};
```

注意，若使用类模板作基类而派生出新的类模板时，基类的名字应为实例化后的“CB<T>”而并非仅使用“CB”。例如，在本例的派生类说明中，要对基类进行指定时必须使用“CB<T>”而不可只使用“CB”：

```
        template <class T> class CA:public CB<T> {…}
```

（3）类模板作基类，派生出新的类模板，且基类与派生类中均使用同一个类型参数 T。

```
template <class T> class CB {
                  //CB 为类模板（其中使用了类型参数 T），它将作为类模板 CA 的基类
    T t;          //数据成员为 T 类型的
  public:
    T gett(){   //用到类型参数 T
        return t;
```

```
    }
    …
};
template <class T> class CA:public CB<T> {
                    //CA 为类模板，其基类 CB 也为类模板。注意，类型参数 T 将被
                    //“传递”给基类 CB；本派生类中也将使用这同一个类型参数 T
    T t1;           //数据为 T 类型的
  public:
    …
};
```

（4）类模板作基类，派生出新的类模板，但基类中使用类型参数 T2，而派生类中使用另一个类型参数 T1（而不使用 T2）。

```
template <class T2> class CB {
                    //CB 为类模板（其中使用了类型参数 T2），它将作为类模板 CA 的基类
    T2 t2;          //数据为 T2 类型的
  public:
    …
};
template <class T1, class T2> class CA:public CB<T2> {
                    //CA 为类模板，其基类 CB 也为类模板。注意，类型参数 T2 将被
                    //“传递”给基类 CB；本派生类中还将使用另一个类型参数 T1
    T1 t1;          //数据为 T1 类型的
  public:
    …
};
```

下面举两个小例子。

例 1：一般类作基类，派生出类模板。

```
// program 9_6.cpp
#include <iostream.h>
class CB {                                  //基类 CB
    int i;          char c;
public:
    CB(int i0, char c0) { i=i0; c=c0; }     //基类构造函数
    int geti() { return i; }
```

```
    char getc() { return c; }
};
template<class T> class CA:public CB{          //派生出的类模板 CA
    T t;
public:
    CA(T t0, int i0, char c0):CB(i0, c0){
        //派生类（模板）构造函数，要对基类的初始化负责
        t=t0;
    }
    void putt(T tt){t=tt;}
    T gett(){return t;}
    void print(){       //注意其中调用了基类的公有成员函数 geti 与 getc
        cout<<"t="<<t<<", i="<<geti()<<", c="<<getc()<<endl;
    }
};
void main(){
    CA<double> objd(123.5, 68, 'A');          //创建派生类“CA<double>”的类对象
    cout<<"objd.geti()="<<objd.geti()<<",   objd.getc()="<<objd.getc()<<endl;
                    //通过派生类对象调用基类的公有成员函数 geti 与 getc
    objd.print();                             //调用派生类的公有成员函数 print
    objd.putt(objd.gett()+39.6);
    cout<<"objd.gett()="<<objd.gett()<<endl;
    objd.print();
}
```

程序执行后的输出结果为：

```
objd.geti()=68,    objd.getc()=A
t=123.5, i=68, c=A
objd.gett()=163.1
t=163.1, i=68, c=A
```

例 2：类模板作基类，派生出类模板。

```
// program 9_7.cpp
#include <iostream.h>
template<class T>class CB{                    //类模板 CB 作为基类
public:
```

```
    void fb(T t){
        cout<<"t="<<t<<endl;
    }
};
template<class T1, class T2> class CA:public CB<T2>{  //派生出的类模板 CA
    //注意，基类的名字应为实例化后的“ CB<T2> ”而并不能仅使用“ CB ”
    //类型参数 T2 将被“ 传递 ”给基类 CB
public:
    void fa(T1 t1){
        cout<<"t1="<<t1<<endl;
    }
};
void main(){
    CA<double,char> obj;   //使用两个类型参数将模板 CA 实例化，而后说明其对象 obj
    obj.fb('E');           //通过派生类对象调用基类的公有成员函数 fb
    obj.fa(35.8);          //调用派生类的公有成员函数 fa
}
```

程序执行后的输出结果为：

```
t=E
t1=35.8
```

注意：由于函数成员 fb 和 fa 都是公有的，且为公有派生，故可通过对象 obj 来直接调用它们。

类模板之间亦可有多继承。设还有另一个已定义的类 CD：

```
class CD{…};
```

则可由类模板 CB 和类 CD 派生一个类模板 CA：

```
template<class T1, class T2>
class CA:public CB<T2>,public CD{
    T1 ca;
    …
};
```

9.4 程序实例：链表类模板的设计

所谓链表，就是以指针作为“链”，把若干相同类型的数据串接起来，它与数组相比较：都是存储若干相同类型的数据的线性序列；数组在存储器中占用一片连续的空间，因此其大小是固定的；对于表中元素的存取是根据数组下标，即数据的顺序号来实现的。

链表在存储器中是分布式存储的，其每个数据项附加一个指针，是通过指针把这些数据串接成一个线性表的。每项只有一个指针的链表称为单向链表，由于每一项可以放在任何地方，因此，表的长度没有限制，这是链表的优越之处。

本例定义一个处理链表的类模板 list，它含有一个类型形参 T，以指出每一链表项中所存放数据（data）的数据类型。

```
// program 9_8.cpp
#include <iostream.h>
#include <process.h>
template <class T> class list {          //类模板 list, 使用类型形参 T
  struct node {
      T data;                            //每一链表项中存放一个 T 类型的数据 data
      node * next;
  } *head, *tail;
    /*  两个私有数据成员 head 与 tail，它们均为指向链表项（结构体）的指针。其中的 head 总指向链表的首项，而 tail 总指向链表的尾项。每当准备往链表中加入一个表项（及其表项 data 数据）时，程序中首先使用“new node”来动态生成一个新的表项空间，并“填入”该表项的 data 数据，而后通过指针的改变与关联，将该表项加入到以 head 为首以 tail 为尾的当前链表结构中（以形成一个更新后的链表）。 */
public:
  list() {                               //构造函数，创建一个“空链表”
      head=tail=NULL;
  };
  void Insert (T * item) {
    /*  动态生成一块链表项空间，并将 item 所指向的 T 类型的数据放入该链表项的 data 域，而后将新生成的该链表项插入到原链的链首（链表的“栈”式用法）。 */
      node * pn;
      pn=new node;
      pn->next=head;                     //插入到原链的链首
      pn->data=*item;                    //将数据放入链表项的 data 域
```

```
        head=pn;                    //新项作为新的链首
        if (tail==NULL)             //若处理链表第一项，使 tail 也指向它
            tail=pn;
    };
    void Append (T * item) {
        /*  Append 的参数 item 为指向 T 类型数据的指针。动态生成一块链表项空间，并将 item
所指向的 T 类型的数据放入该链表项的 data 域，而后将新生成的该链表项附加到原链的链
尾（链表的“队列”式用法）。 */
        node * pn;
        pn=new node;
        pn->next=NULL;              //该项将作为链表末项
        pn->data=*item;             //将数据放入链表项的 data 域
        if (tail==NULL)             //若处理链表第一项，使 head 与 tail 都指向该项
            head=tail=pn;
        else {                      //处理链表非首项时，将新生成项“链接”到末项之后
            tail->next=pn;
            tail=pn;
        }
    };
    T Get (){                       //取出链表首项数据（data 域值），并将该首项从链表中删去
        if (head==NULL)             //链表为空，退出程序
            exit (0);
        T temp=head->data;          //temp 即为要返回的链表首项数据（data 域值）
        node * pn=head;
        if (head->next==NULL)
            head=tail=NULL;         //若链表中仅此一项，删除后链表应变为空链
        else
            head=head->next;        //链表项多于 1 时，“下推”首指针 head
        delete (pn);                //删除链表首项
        return temp;                //返回链表首项数据（data 域值）
    }
};                                  //类模板 list 定义结束
class person{                       //自定义 person 类（类型），仅含有三个公有的数据成员
  public:
```

```
    char name[20];              //姓名
    int age;                    //年龄
    float hight;                //身高
};
void main() {                   //创建类模板 list 的实例对象并对它们进行使用
    person ps;
    list <int> link1;           //int 型空链表 link1
    list <person> link2;        // person 型空链表 link2
    cout<<"  --- Input 5 person's information ---"<<endl;
    for (int i=0;i<5;i++){      //输入 5 个人的有关信息并进行处理
        cout<<"input "<<i<<" inf(name,age,hight):";
        cin>>ps.name>>ps.age>>ps.hight;
        link2.Insert(&ps);      //将 ps 对象（当前人的信息）插入到 link2 链表的链首
        link1.Append(&i);       //将当前对象的顺序号附加到 link1 链表的链尾
    }
    /* 上述循环结束后，link1 和 link2 链表的逻辑结构及内容概示如下：
     link1 链表：head -> 0 -> 1 -> 2 -> 3 -> 4
     link2 链表： head -> 4 号人员信息 -> 3 号人员信息 -> 2 号人员信息
      -> 1 号人员信息 -> 0 号人员信息  */
    cout<<"  ----- The result ----"<<endl;
    for (i=0;i<5;i++) {
        ps=link2.Get();         //取出 link2 链表首项的人员信息
        link2.Append(&ps);      //将刚从 link2 首取来的人员信息，
                                //再一次附加到 link2 链表的链尾
        cout<<ps.name<<" "<<link1.Get()<<endl;   //输出 link2 链表人员信息的 name，
                                //以及 link1 链表表项中的对象的顺序号
    }
}
```

程序执行后的显示结果如下：

```
  --- Input 5 person's information ---
input 0 inf(name,age,hight):zhangLi 20 1.68
input 1 inf(name,age,hight):wangyue 21 1.72
input 2 inf(name,age,hight):liming 19 1.75
input 3 inf(name,age,hight):zhaoyi 19 1.78
```

```
input 4 inf(name,age,hight):chenjin 20 1.8
   ----- The result ----
chenjin 0
zhaoyi 1
liming 2
wangyue 3
zhangLi 4
```

程序说明：

（1）这个程序以类模板的形式定义了一个链表，这个链表的数据项可以是不同类型的。例如在 main()**中，说明了两个对象链表分别为整型和** person **型。**

（2）类成员由两个私有数据成员和四个公有函数成员组成。数据成员为类内定义的结构 node **类型的指针，用于存放数据的变量由动态分配（**new**）方式生成。**

除构造函数之外，为链表设计了三种操作，即：

Insert()**：在链前插入一项。**

Append()**：在链尾插入一项。**

Get()**：从链首取出数据，并删去该项。**

还可以根据实际需要为链表设计其它操作，例如在链表中进行搜索，在中间插入等等。

（3）这样的链表也可以作为栈和队列来使用，只允许 Insert()**和** Get()**时，它基本上是一个栈；而只允许** Append()**和** Get()**时，它起到队列的作用，这样的栈和队列将没有长度的限制。**

（4）在主函数中，说明了两个空的链表（对象），然后对它们进行了一系列的插入、附加和删除（取值）操作。

思考题

1. **什么是模板？为什么要使用模板？模板可以分为几种类型？**
2. **怎样定义一个模板？**
3. **什么是模板的静态成员？如何定义和使用一个模板的静态成员？**
4. **什么是模板的友元？如何定义和使用一个模板的友元？**
5. **什么是模板的参数？定义模板的参数时应该注意什么？**

6. 什么是模板的实例化？

7. 为什么说类模板和函数模板的使用有利于加强程序的可重用性？

练习题

1. 对于 program9_2 中的类模板 list 补充成员函数：

   int count():返回当前链表的项数。

   void htot():把链首项移到链尾。

2. 编写一个对 n 元数组进行排序的函数模板。

3. 编写一个求 m*n 阶矩阵中最大元和最小元的函数模板。

4. 设计一个先进先出的队列类模板，可以定义多种数据类型的队列，且能够实现队列空间的循环使用。

第十章 输入输出流

C++语句虽然从 C 语言接收下一套以 printf 函数库形式工作的 I/O 机制，但它并不满足于此，又开发了一套自己的具有安全、简洁、可扩展的高效 I/O 系统。以流类库形式工作的这个 I/O 系统显然是成功的。在前面的各章中，几乎所有的程序实例中都使用了系统提供的 I/O 流，以实现必要的 I/O 操作。本章将对于 C++的 I/O 流系统做出全面的阐述，其目的有二：

（1）介绍 I/O 流系统的工作原理和有关概念，特别是 I/O 操作和格式控制的方法；

（2）把以实现 I/O 操作为基本功能的通过类的继承与多继承关系构造起来的流类库，作为一个按 OOP 框架构造的实际的 C++程序系统范例，帮助读者对以类为核心设计软件的方法与技术有一个较深入的认识。

10.1 文件、流及 C++的流类库

10.1.1 流类库的优点

用 C++语言自己的支持 I/O 操作的流类库代替 printf 函数族，是一个明显的进步。虽然不少 C 程序员满足于 C 系统提供的 I/O 函数库，认为它是有效和方便的，但与 C++的 I/O 系统相比，就显示出明显的缺点，因此没有人否认这种取代是必然的。

（1）简明与可读性

从直观上来看，这种改变使得 I/O 语句更为简明，增加了可读性。用 I/O 运算符（提取运算符“>>”和插入运算符“<<”）代替不同的输入输出函数名（如 printf，scanf 等）是一个大的改进。例如，从下面的两个输出语句可以反映出二者之间的差别：

```
printf("n=%d,a=%f\n", n, a);
cout<<"n="<<n<<",a="<<a<<endl;
```

虽然两个语言的输出结果是一样的，但在编写程序语句和阅读它们时，感觉却是不同的。后者简明，直观，易写，易读，几乎所有的 C 程序员转而使用 C++语言后，都很自然地接受了新的形式，再让他返回到旧的方式反而会感到不习惯了。

（2）类型安全（type safe）

所谓类型安全，是指在进行 I/O 操作时不应对于参加输入输出的数据在类型上发生不应有的变化。仍以最简单的输出语句为例，下面是一个显示颜色值 color 和尺寸 size 的一个简单函数：

```
show(int color, float size){
   cout<<"color="<<color<<",size="<<size<<endl;
}
```

在这个函数的调用过程中，系统（编译器）将自动按参数的类型定义检查实参的表达式，显示的结果中，第一个自然是整数值，第二个 size 必然是浮点类型值。

如果采用 printf（）函数，由于其参数中的数据类型必须由程序员以参数格式%d，%f，%c，%s 的形式给出，同样实现上述函数 show（），就可能产生编译器无法解决的问题：

```
show(int color,float size){
   printf("color=%f,size=%d\n", color, size);
}
```

程序员在确认输出数据类型时发生错误是可能的，这时输出数据的类型：color 是 int 型，size 是 float 型，与 printf（）中给出的参数格式符%f 对应 color，%d 对应于 size，两者发生了矛盾。因此说，它是类型不安全的。而 C++的 I/O 系统不会出现这种情形。

（3）易于扩充

在 C++语言所附的 I/O 系统，在其流类的定义中，把原来 C++语言中的左、右移位运算符“ << ”和“ >> ”，通过运算符重载的方法，定义为插入（输出）和提取（输入）运算符。这就为输入输出功能对于各种用户定义的类型数据的扩充，创造了方便的条件。而在 stdio.h 文件中说明的 printf（）函数却很难做到这一点。例如：

在 C++语言提供的 I/O 系统中，它是把运算符“ << ”的重载函数作为输出流类 ostream 的成员函数来定义的，分别对字符串 char，short，int，long，float，double，const void *（指针）等类型作了说明。在此基础上，用户不难对于新的类型数据的输出来重载运算符“ << ”。它可以作为用户定义的类型（例如类 complex）的友元函数来定义：

```
friend ostream & operator<<(ostream & s, complex c){
   s<<'('<<c.re<<','<<c.im<<')';          return s;
}
```

10.1.2 文件与流的概念

文件（File）是计算机的基本概念，一般指存储于外部介质上的信息集合。每个文件应有一个包括设备及路径信息的文件名。其中外部介质主要指硬盘，也可包括光盘、软盘或磁带等。信息是数据和程序代码的总称。

文件分为文本文件和二进制文件，前者以字节（byte）为单位，每字节对应一 ASCII 码，表示一个字符，故又称字符文件。文本文件保存的是一串 ASCII 字符，可用文字处理器对其进行编辑，输入输出过程中系统要对内外存的数据格式进行相应转换。二进制文件以字位（bit）为单位，实际上是由 0 和 1 组成的序列，输入输出过程中，系统不对相应数据进行任何转换。例如整数 12345 以文本形式存储占用五个字节，以二进制形式存储则可能只占用两个字节（16 bits）。

在程序中，文件的概念不单是狭义地指硬盘上的文件，所有的有输入输出功能的设备，例如键盘，控制台，显示器，打印机都被视为文件。这就是广义的文件的概念。就输入输出操作来说，这些外设和硬盘上的文件是一致的，对于程序员来说文件只与信息的输入输出相关，而且这种输入输出是串行序列形式的。于是，人们把文件的概念抽象为“流”（stream）。

流是程序设计对 I/O 系统中文件的抽象。一个输入文件，它可能是一个只读磁盘文件，也可能是键盘，一律把它视为流的“源”。而一个输出文件则称为流的“汇”。

C++的 I/O 系统，定义了一系列由某种继承派生关系的流类，并为这些抽象的流类定义一系列的 I/O 操作函数，当需要进行实际的 I/O 操作时，只需创建这些类的对象（称为流），并令其与相应的物理文件（硬盘或软盘文件名或者外设名）相联系。

因此，文件可以说是个物理概念，而流则是一个逻辑概念。所有流（类对象）的行为都是相同的，而不同的文件则可能具有不同的行为。如，磁盘文件可进行写也可进行读操作；显示器文件则只可进行写操作；而键盘文件只可进行读操作。

I/O 操作是针对抽象的流来定义的，对文件的 I/O 操作，其前提是把该文件与一个（对象）流联系起来，这是 C++的 I/O 系统的基本原理。具体地说，当程序与一个文件交换信息时，必须通过“打开文件”的操作将一个文件与一个流（类对象）联系起来。一旦建立了这种联系，以后对该流（类对象）的访问就是对该文件的访问，也就是对一个具体设备的访问。可通过“关闭文件”的

操作将一个文件与流（类对象）的联系断开。

在内存中开辟一片区域作为输入输出操作的缓冲区，可以提高运行效率，因此 I/O 操作可以区分为缓冲 I/O 和非缓冲 I/O。

10.1.3 C++的流类库

严格地说，C++的 I/O 系统并不是 C++语言的一部分，它是系统为用户提供的专用于 I/O 的标准类（及函数，对象）等。

作为基本类的主要几个类在头文件 iostream.h 中被说明。下面将对其中几个主要类的内容作一简介。

· ios 类：在其中以枚举定义方式给出一系列与 I/O 有关的状态标志，工作方式等常量，定义了一系列涉及输入输出格式的成员函数（包括设置域宽，数据精度等），它的一个数据成员是流的缓冲区指针。同时，类 ios 作为虚基类派生了输入流类 istream 和输出流类 ostream。

· streambuf 类，负责管理流的缓冲区。包括负责设置缓冲区和在缓冲区与输入流和输出流之间存取字符的操作的成员函数。

· istream 类和 ostream 类除继承了类 ios 的成员之外，主要为 C++的系统数据类型分别对于运算符“>>”和“<<”进行重载。

· iostream 类以 istream 和 ostream 为基类，它同时继承二者，以便创建可以同时进行 I/O 操作即进行输入和输出双向操作的流。

· istream-withassign 类是 istream 的派生类，主要增加了输入流（对象）之间的赋值（“=”）运算。

· ostream-withassign 类是 ostream 的派生类，主要增加了输出流（对象）之间的赋值（“=”）运算。

· iostream-withassign 类是 iostream 的派生类。

这八个类的继承关系如图 10.1：其中 streambuf 类与 ios 类之间没有继承关系，当 I/O 操作需要使用 I/O 缓冲区时，可以创建缓冲区对象，通过流的缓冲区指针，来完成有关缓冲区的操作。

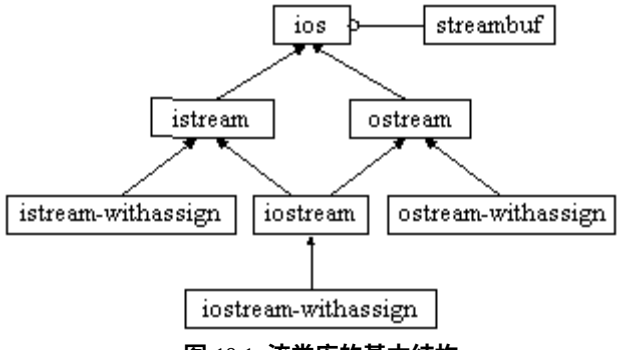


图 10.1 流类库的基本结构

大多数 I/O 操作的函数包括在 ios 类和 istream 类，ostream 类，iostream 类中。名字带有“ withassign ”的三个类，实际上是补充了流对象的赋值操作（注，某些编译系统，如 VC6，没提供 iostream-withassign 类）。由此也可以看到，具有层次关系的类说明的灵活性。

在 10.4 节中，我们还可以看到，在这个基本 I/O 类库结构基础上，如何通过派生类的说明，进一步把文件（指磁盘文件）I/O 功能加进来。

在头文件 iostream.h 中，除了类的定义之外，还包括四个对象的说明，它们被称为标准流，或预定义流。

· cin 是类 istream 的对象（在 Borland C++中，为 istream-withassign 类的对象），为标准输入流，在不作其它说明条件下，与标准输入设备（一般指键盘）相关联。

· cout 是类 ostream 的对象（在 Borland C++中，为 ostream-withassign 类的对象），为标准输出流，在不作其它说明条件下，与标准输出设备（显示器）相关联。

· cerr，clog 也是类 ostream 的对象（在 Borland C++中，为 ostream-withassign 类的对象），与标准（错误）输出设备（显示器）相关联，前者为非缓冲方式，后者为缓冲方式。令 cerr 为非缓冲的流，可以保证出现错误后立即把出错信息在显示器上输出。

正是因为 cin 和 cout 是预定义的流，所以在本书的大量程序中，可以未经说明直接使用它们。其条件是包含头文件 iostream.h，则有关的类说明和对象流的说明，已经写好并存在其中了。另一点要注意的是，程序中使用上述 4 个预定义流类对象进行读写时，不必先进行“ 打开文件 ”的操作，使用完后也不需要进行“ 关闭文件 ”的操作（因为这些流类对象与文件之间的联系是预定义好的，可认为系统已为每一程序都隐含进行了对它们的打开与关闭操作）。

在类 ios 中，还提供了一些控制输入输出格式的函数成员（在 10.3 节介绍）。

10.2 插入与提取运算符的重载

在 10.1.1 节“ 流类库的优点 ”处，简单提到过对插入与提取运算符进行重载的事，本节将对它们进行较为细致的讨论。

1．系统预先进行的有关重载

对于 ostream 类以及 istream 类来说，针对某些最常用的基本数据类型（如对 int、char、float、double、char*等），C++预先对“ 插入 ”算符“ << ”（也

称输出算符）以及“提取”算符“ >> ”（也称输入算符）进行了运算符重载定义，从而使得对上述基本数据类型的某表达式 x，使用“ cout<<x; ”的运算符调用方式（注意，cout 为预定义的 ostream 类对象），完全等同于“ cout.operator<<(x); ”的函数调用方式。它们的功能都为：调用 cout 对象（属于 ostream 类）的“ operator<< ”函数，带去实参 x，用于实现将实参 x（的值），通过“ operator<< ”函数，输出到 cout（屏幕）上。

由于算符重载函数“ operator << ”返回的是引用“ ostream& ”，可达到作为左值的目的，起到一个 ostream 类型的独立对象（变量）的作用，从而可知如下两种使用方式的合法性与等同性：

```
(cout.operator<<(x)).operator<<(y);      //函数调用方式
cout<<x<<y;                              //运算符调用方式
```

同理对“ cin>>x; ”的运算符调用方式（注意，cin 为预定义的 istream 类对象）完全等同于“ cin.operator>>(x); ”的函数调用方式。它们的功能都为：调用 cin 对象（属于 istream 类）的“ operator>> ”函数，带去实参 x（变量），用于实现：把从 cin（键盘）输入的值放入（赋值给）实参 x（变量）。

由于算符重载函数“ operator >> ”返回的是引用“ istream& ”，可达到作为左值的目的，起到一个 istream 类型的独立对象（变量）的作用，从而也可知如下两种使用方式的合法性与等同性：

```
(cin.operator>>(x)).operator>>(y);       //函数调用方式
cin>>x>>y;                               //运算符调用方式
```

另外注意，用户可对任一个自定义类（类型）重载插入与提取运算符“ << ”与“ >> ”，以扩大输入输出运算符的使用范围（否则的话，不可使用算符“ << ”与“ >> ”直接对该自定义类的对象进行输入与输出操作，因为系统只对预定义的某些最常用基本数据类型提供了输入输出运算符的具体实现）。

2．对自定义类重载插入与提取运算符

对自定义类重载插入与提取运算符“ << ”与“ >> ”时，通常总以友元方式来重载，而且大都使用类似于如下的重载格式：

```
friend istream& operator>>(istream& in, complex& com);
friend ostream& operator<<(ostream& out, complex com);
```

其中的“ operator>> ”用于完成从 istream 类的流类对象 in 上（如对应实参可为 cin，即指定从键盘上）输入一个复数的有关数据放入 complex 型引用对象 com 中；而“ operator<< ”则用于实现往 ostream 类的流类对象 out 上（如对应实参可为 cout，即指定往屏幕上）输出 complex 类对象 com 的有关数据。

另外注意：上面重载的输入输出运算符的返回类型均为引用，为的是可使用返回结果继续作左值，也即使返回结果能起到一个独立对象（变量）的作用，从而可使用像“ cout<<c1<<c2; ”以及“ cin>>c1>>c2; ”这样的调用语句。还有，“ operator >> ”的第二形参 com 也必须被说明成引用“ complex& com ”，目的则是要将输入数据直接赋值给对应实参变量（所拥有的存储空间中）。

下面看一个具体使用例。

自定义如下形式的一个简单的复数类 complex，其中除去重载完成复数加法与乘法的运算符“ + ”与“ * ”之外，还要重载用于完成复数输入输出的运算符“ << ”与“ >> ”，以实现直接对该自定义类的对象进行输入与输出的操作。而后编制主函数，说明 complex 类对象，并对各重载运算符进行使用，以验证它们的正确性。

```
// program 10_1.cpp
#include<fstream.h>
class complex {                    //自定义的简单复数类 complex
      double  r;                   //复数实部
      double  i;                   //复数虚部
public:
      complex(double r0=0, double i0=0){        //构造函数，并设置参数默认值
            r=r0;          i=i0;
       }
      complex operator +(complex c2){           //重载运算符“ + ”，实现二复数相加
            complex c;          c.r=r+c2.r;
            c.i=i+c2.i;         return c;
      }
      complex operator *(complex c2){           //重载运算符“ * ”，实现二复数相乘
            complex temp;                       temp.r=(r*c2.r)-(i*c2.i);
            temp.i=(r*c2.i)+(i*c2.r);           return temp;
      }
     friend istream& operator >> (istream& in, complex& com){
                                  //以友元方式重载输入运算符“ >> ”，输入复数
            in>>com.r>>com.i;
            return in;            //该 return 语句不可缺少（因为函数返回类型为“ istream& ”）
       }
     friend ostream& operator << (ostream& out, complex com){
```

```
                                    //以友元方式重载输出运算符“ << ”，输出复数
        out<<"("<<com.r<<", "<<com.i<<")"<<endl;
        return out;          //该 return 语句不可缺少（因为函数返回类型为“ ostream& ”）
    }
};
void main() {                //主函数，说明 complex 类对象，并对重载的运算符进行使用
    complex c1(1,1), c2(2,3), c3, res;
    cout<<"c1="<<c1<<"c2="<<c2;          //要调用“ operator << ”运算符重载函数
    res = c1+c2;
    cout<<"c1+c2="<<res;
    cout<<"c1*c2="<<c1*c2;
    cout<<"Input c3:";
    cin>>c3;                             //要调用“ operator >> ”运算符重载函数
    cout<<"c3+c3="<<c3+c3;
}
```

程序执行后，屏幕显示结果为：

```
c1=(1, 1)
c2=(2, 3)
c1+c2=(3, 4)
c1*c2=(-1, 5)
Input c3:3 -5
c3+c3=(6, -10)
```

有以下两点注意：

① 主函数的输入输出语句中出现的类对象 cout 以及 cin 正是输入输出重载函数中引用型形参 out 以及 in 的对应实参。即是说，若使用“ cout<<c1; ”它将等同于“ operator<<(cout, c1); ”，而使用“ cin>>c3; ”则等同于使用“ operator>>(cin, c3); ”。

② 本程序调用自定义的“ operator >> ”和“ operator << ”运算符重载函数时，其第一实参使用的是类对象 cout 和 cin。实际上，该第一实参也可使用 fstream 类的类对象（对应于一个磁盘文件），那时的输入输出将与磁盘文件相关联（而不再是键盘与屏幕），fstream 类及其类对象的使用方法请参看随后的 10.4 节，那时读者可作为练习来具体实现。

10.3 I/O 的格式控制

C++语言的 I/O 系统有完善的格式控制功能，除了以类 ios 的成员函数形式给出了一套控制函数之外，还另外设计了使用方便的格式控制符和控制函数。此外，用户还可以定义自己的格式控制函数。下面分别介绍。

10.3.1 用于格式控制的类 ios 成员函数

在类 ios 的说明中，定义了一批公有的格式控制标志位以及一些用于格式控制的公有成员函数，通常先用某些成员函数来设置标志位，然后再使用另一些成员函数来进行格式输出。另外，ios 类中还设置了一个 long 类型的数据成员用来记录当前被设置的格式状态，该数据成员被称为格式控制标志字（或标志状态字）。标志字是由格式控制标志位来“合成”的。

注意，ios 类作为诸多 I/O 流类的基类，其公有成员函数当然可被各派生类的对象所直接调用。

类 ios 中用于格式控制的公有成员函数有：

```
long  flags();              //返回当前标志字
long flags(long);           //设置标志字并返回
long setf(long);            //设置指定的标志位
long unsetf(long);          //清除指定的标志位
long setf(long,long);       //设置指定的标志位的值
int width();                //返回当前显示数据的域宽
int width(int);             //设置当前显示数据域宽并返回原域宽
char fill();                //返回当前填充字符
char fill(char);            //设置填充字符并返回原填充字符
int precision();            //返回当前浮点数精度
int precision(int);         //设置浮点数精度并返回原精度
```

所涉及的标志字（状态字）的各位都控制一定的 I/O 特征，例如标志字的右第一位为 1，则表示在输入时跳过空白符号。标志字的各位以枚举类型形式定义于 ios 说明中：

```
enum{
  skipws=0x0001        //输入时跳过前导的空白符号（如，空格、Tab 键、换行等）
  left=0x0002          //左对齐输出，右边填充“填充字符”(在域宽范围内)
```

```
    right=0x0004        //右对齐输出，左边“填充字符”，此为默认对齐方式
    internal=0x0008     //在符号位和基指示符之后、但在数值之前填充
    dec=0x0010          //十进制格式（默认数制）；
    oct=0x0020          //八进制格式
    hex=0x0040          //十六进制格式
    showbase=0x0080     //输出带有基指示符（8 进时带 0，16 进时带 0x）
    showpoint=0x0100    //输出浮点数带小数点
    uppercase=0x0200    //十六进制大写输出 A 到 F，对科学表示法显示大写的 E
    showpos=0x0400      //输出正整数带正符号（+）
    scientific=0x0800   //输出浮点数用科学表示法（有指数部分）
    fixed=0x1000        //输出浮点数以定点形式（无指数部分）
    unitbuf=0x2000      //插入后立即刷新流缓冲区
    stdio=0x4000        //插入后立即刷新标准流 stdout 和 stderr
};
```

这个枚举定义指出标志状态字的各个不同的位所控制的不同功能。例如 dec=0x0010 说明其右第 5 位控制数据是否为十进制格式。

下面进一步介绍 ios 中用于格式控制的 6 个公有成员函数的功能及其使用。

(1) ios::flags

a. 格式一： long flags(long lFlags);

通过参数 lFlags 来重新设置（更新）标志字，并返回更新前的标志字。

参数 lFlags 是由 ios 类中预定义的用来表示各格式控制标志位的上述枚举常量值来“合成”的。每一枚举常量值都代表着格式控制标志字中的某一个二进制位（bit），当设置了某个标志位属性时，该位将取值“1”，否则该位取值“0”。另外注意，通过使用位运算符“|”可将多个格式控制标志位属性进行“合成”。但从使用角度看，所设置的标志位属性不能产生互斥。例如，格式控制标志字中设立了三个平行的标志位（ios::dec、ios::oct 和 ios::hex）用于表示数制，程序员应保障任何时刻只设置其中的某一个标志位。还有表示对齐标志位的 os::left、ios::right 和 ios::internal，以及表示实数格式标志位的 ios::scientific 和 ios::fixed，这些互斥属性也不能同时设置。

b. 格式二： long flags();

无参的 flags 函数用来返回当前的标志字（值）。

(2) ios::setf

a. 格式一： long setf (long lFlags) ;

通过参数 lFlags 来设置指定的格式控制标志位（使那些位的值为“1”。

注意，与具有一个参数的 flags 函数的“替换”方式不同，此处为“添加”方式，即是说，它并不更改其它 lFlags 不涉及到的那些标志位的当前值)。

参数 lFlags 的可取值及使用含义与上述 flags 函数中的 lFlags 参数相同。

b. 格式二： long setf(long lFlags, long lMask);

设置指定的格式控制标志位的值。首先将第二参数 lMask 所指定的那些位清零，而后用第一参数 lFlags 所给定的值来重置这些标志位，函数的返回值为设置前的标志字。

参数 lFlags 的可取值及使用含义与上述 flags 函数中的 lFlags 参数相同。

为了保障所设置的数制标志位（ios::dec、ios::oct 和 ios::hex）不产生互斥，通常要使用这种具有两个参数的如下形式的 setf 函数，如要设置 16 进制时使用：

```
setf(ios::hex, ios::basefield);
```

其中的 ios::basefield 为一个在 ios 类中定义的公有静态常量，它的取值为 ios::dec|ios::oct|ios::hex。

同理，为了保障所设置的对齐标志位（ios::left、ios::right 和 ios::internal）以及实数格式标志位（ios::scientific 和 ios::fixed）不相冲突，也要使用这种具有两个参数的 setf 函数，而且要用到在 ios 类中定义的另外两个公有静态常量 -- ios::adjustfield 和 ios::floatfield。ios::adjustfield 的取值为 ios::left|ios::right|ios::internal，而 ios::floatfield 的取值为 ios::scientific|ios::fixed。

例如，要设置对齐标志位为 ios::right 以及实数格式标志位为 ios::fixed，可使用：

```
setf(ios::right, ios::adjustfield);
setf(ios::fixed, ios::floatfield);
```

(3) ios::unsetf

```
long unsetf( long lFlags );
```

通过参数 lFlags 来清除指定的格式控制标志位（使那些位的值为“0”）。

参数 lFlags 的可取值及使用含义与上述 flags 函数中的 lFlags 参数相同。

(4) ios::fill

```
char fill( char cFill );
```

将“填充字符”设置为 cFill，并返回原“填充字符”。注意，缺省的填充字符为空格。

另外，调用无参的“fill()”将返回当前的“填充字符”。

(5) ios::precision

int precision(int np);

设置浮点数精度为 np 并返回原精度。当格式为 ios::scientific 或 ios::fixed 时，精度 np 指小数点后的位数，否则指有效数字。

另外，调用无参的“precision()”将返回当前浮点数精度。

(6) ios::width

int width(int nw);

设置当前被显示数据的域宽 nw 并返回原域宽。默认值为 0，将按实际需要的域宽进行输出。此设置只对随后的一个数据有效，而后系统立刻恢复域宽为系统默认值 0。

另外，调用无参的“width()”将返回当前显示数据的域宽。

下面是一些格式函数的使用例子：

cin.flags()返回当前的（键盘）输入流中的标志状态字，但它一般不单独使用。

```
cin.flags(cin.flags()|ios::skipws);
```

这个函数的调用，其实参为两个状态字的“或”运算的结果。其功能是在原有的状态不变条件下，输入时跳过前导空白。

```
cout.flags(cout.flags()|ios::showbase);
```

此函数调用结果为要求输出时标明数制基数符，其它设置不变。如果要求不标明基数符，则需执行：

```
cout.flags(cout.flags()&～ios::showbase);
```

用 setf 函数有时可能比较方便：

```
cin.setf(ios::skipws);
```

同样是要求输入时跳过空白。为了改变这一设置，令其输入时不跳过空白可用下面的两种方式之一：

```
cin.unsetf(ios::skipws);
```

或

```
cin.setf(0, ios::skipws);
```

再如：

```
cout.unsetf(ios::uppercase);
```

可令十六进制数以小写字符输出，而

```
cout.setf(1, ios::uppercase);
```

则可令其大写输出。

下面的程序中使用了格式控制函数，将“展现”flags 和 setf 等函数来设置（操作）格式控制标志位的功能。

```
// program 10_2.cpp
```

```
#include <iostream.h>
void main() {
//显示出 ios 类中定义的三个公有静态常量的值
      cout<<ios::basefield;                                    //输出：112
      cout<<"\t"<<(ios::dec|ios::oct|ios::hex)<<endl;          //输出：112
      cout<<ios::adjustfield;                                  //输出：14
      cout<<"\t"<<(ios::left|ios::right|ios::internal)<<endl;  //输出：14
      cout<<ios::floatfield;                                   //输出：6144
      cout<<"\t"<<(ios::scientific|ios::fixed)<<endl;          //输出：6144
//注意 flags 将重新设置（更新）原标志字，为“替换”方式
     cout.flags(ios::showbase);
      cout<<cout.flags();                                      //输出：128
      cout.flags(ios::showpoint);
      cout<<"\t"<<cout.flags()<<endl;                          //输出：256
     cout.unsetf(ios::showpoint);
      cout<<cout.flags();                                      //输出：0
//注意 setf 并不更改其实参 lFlags 不涉及到的那些标志位的当前值（“添加”方式）
      cout.setf(ios::showbase);
      cout<<"\t"<<cout.flags()<<endl;                          //输出：128
      cout.setf(ios::showpoint);
      cout<<cout.flags();                                      //输出：384
      cout.unsetf(ios::showpoint);
      cout<<"\t"<<cout.flags()<<endl;                          //输出：128
}
```

程序输出结果为：

```
112         112
14          14
6144        6144
128         256
0           128
384         128
```

再看一个示例程序：

```
// program 10_3.cpp
# include<iostream.h>
```

```
void main( ){
        cout.setf(ios::scientific);                 //科学表示法
        cout.setf(ios::showpos);                    //显示正号
        cout<<4785<<27.4272<<endl;
        cout.unsetf(ios::showpos);                  //不用显示正号
        cout.precision(2);                          //小数点后取 2 位
        cout.width(5);                              //打印宽度为 5
        cout<<4785<<","<<27.4272<<endl;
        cout.fill('#');                             //用"#"填充空格
        cout.width(8);                              //宽度为 8
        cout<<4785<<endl;
}
```

程序执行后的输出结果为：

```
+4785+2.742720e+001
   4785,2.74e+001
####4785
```

10.3.2 格式控制符

为了进行格式控制，上一节以 ios 的成员函数定义的格式函数已经完全够用。但是，它们的使用有些不够方便。例如，在使用中必须加上流类对象名和“.”进行限定，而且它们必须以单独的语句调用。因此，C++的 I/O 系统又定义了一些用来管理 I/O 格式的控制函数。它们不在类的封装之内，表面上也不以函数调用的形式出现（而直接用于提取和插入算符之后），因此它们被称为格式控制符，格式控制符包括有参和无参的控制符，分别在两个头文件中说明。

定义在 iostream.h 文件中的无参 I/O 控制符有：

endl	输出时插入换行符并刷新流
ends	输出时在字符串后插入 NULL 作为结束符
flush	刷新流，将缓冲区中的当前信息立即输出到目标设备
ws	输入时略去前导的空白字符（空格、Tab 键、换行）
dec	令 I/O 数据按十进制格式（默认数制）
hex	令 I/O 数据按十六进制格式
oct	令 I/O 数据按八进制格式

定义在 iomanip.h 文件中的带参控制符有：

setbase (int base)	设置数制转换基数为 base
resetiosflags (long lFlags)	清除参数 lFlags 所指定的标志位
setiosflags (long lFlags)	设置参数 lFlags 所指定的标志位
setfill (char cFill)	将“填充字符”设置为 cFill
setprecision (int np)	设置浮点数精度为 np
setw (int nw)	设置当前显示数据域宽为 nw

注意 setprecision 的确切含义：当格式为 ios::scientific 或 ios::fixed 时，精度 np 指小数点后的位数，否则指有效数字。

这些格式控制符大致可以代替 ios 的格式函数成员的功能，且使用比较方便。例如，为了把整数 457 按 16 进制输入，可有两种方式：

```
int i=457;
cout.setf(ios::hex, ios::basefield);
cout<<i<<endl;
```

或者：

```
int i=457;
cout<<hex<<i<<endl;
```

由例中可以看出采用格式控制符比较方便，二者的区别主要为：用成员函数须增加限定前缀：“ cout ”，且要单独成一语句，而控制符是类外定义的，使用时无此要求。

下面的示例程序中使用格式控制函数以及有参和无参的格式控制符，对输出数据的宽度、精度等方面进行设置与使用。

```
// program 10_4.cpp
#include <iomanip.h>
void main() {
    cout.width(6);          //格式控制函数 width(6)，只管随后那一个数 4785 的域宽
    cout<<4785<<27.4272<<endl;                          //  478527.4272
    cout<<setw(6)<<4785<<setw(8)<<27.4272<<endl;        //  4785  27.4272
                            //有参格式控制符 setw(6)及 setw(8)也只管随后一个数的域宽
    cout.width(6);          //只管随后那一个数 4785 的域宽
    cout.precision(3);      //此时的格式不为 ios::scientific 也不为 ios::fixed，
                            //precision(3)设置浮点数的有效数字
    cout<<4785<<setw(8)<<27.4272<<endl;                 //  4785     27.4
    cout<<setw(6)<<4785<<setw(8)<<setprecision(2)<<27.4272<<endl;
        //  4785         27。  注意 setprecision(2)设置浮点数的有效数字
```

```
    cout.setf(ios::fixed, ios::floatfield);        //今后以定点格式显示浮点数（无指数部分）
    cout.width(6);
    cout.precision(3);       //当格式为 ios::fixed 时，precision(3)设置小数点后的位数
    cout<<4785<<setw(8)<<27.4272<<endl;                          //   4785    27.427
}
```

程序执行后的输出结果：

```
478527.4272
4785 27.4272
4785      27.4
4785         27
```

10.3.3 用户定义格式控制符

C++的 I/O 系统还为用户提供了自定义格式控制符的功能。一般在下面两种情形下，由用户定义自己的格式控制符是有必要的。

（1）用于输出空格的格式控制符：

```
ostream&    sp(ostream& outs){
        return outs<<'   ';
}
```

经过对于用户定义格式控制符 SP 的定义，它就可以在格式输出过程中使用。例如：

```
void main(){
    int a=123;
    cout<<a<<sp<<"Group:"<<sp<<43.75<<endl;
}
```

程序执行后的显示结果如下：

```
123 Group: 43.75
```

其中的自定义格式控制符 SP，它简单地表示输出一个空格，这种方式比用' '或" "更可靠。

注意，用户定义的格式控制符 SP 实质上仍是一个函数，该函数要与 ostream 类的参数 outs 相联系（如，其实参可为 cout），函数中要依靠那一个 return 语句返回 ostream 类的一个引用（以使返回结果仍可以起到一个独立对象的作用）。请参看 10.2 节的有关解释以加深理解。

（2）也可以在 I/O 格式控制中定义一些包含复合动作的控制符：

```
istream& hexin(istream& in){
    in>>hex;
    cout<<"Enter number using hexadecimal format:";
    return in;
};
ostream& Bankout(ostream& out){
    out.setf(ios::left,ios::adjustfield);
    out<<'$'<<setw(12)<<setfill('#');
    return out;
};
```

这里定义的格式控制符有复合功能：

hexin **用于输入流，它完成两项工作：首先把数据输入设置为按十六进制格式，然后显示出按十六进制输入的提示。**

Bankout **可用于金额数的输出，它可同时规定一个金额数在输出时，按左对齐，域宽为** 12，**空用“**#**”来填充。**

通过在如下 main **函数中的使用，可看出它们的具体功能：**

```
void main(){
    int n;
    cin>>hexin>>n;                          //其中对自定义的 hexin 进行了调用
    cout<<"The number in decimal is:"<<n<<endl;
    double balance=1275.48;
    cout<<Bankout<<balance<<endl;           //对自定义的 Bankout 进行了调用
}
```

程序执行后的显示结果如下：

```
Enter number using hexadecimal format:1234
The number in decimal is:4660
$1275.48#####
```

10.4 磁盘文件 I/O

现在我们介绍磁盘（或光盘、磁带）文件的 I/O **操作。如前所述，从逻辑概念上说，磁盘文件与前面讨论的标准设备（键盘、显示器）文件没有本质的区别，标准流** cin，cout **等与文件流大致相当。不过，从具体细节上，还是有些**

区别。因此，C++的 I/O 系统在基本类 ios，istream，ostream 等等的定义基础上，又为磁盘文件的 I/O 派生出一个专用的 I/O 流类系统：

类 filebuf 从 streambuf 派生

类 fstreambase 从 ios 派生

类 ifstream 从 istream 和 fstreambase 派生

类 ofstream 从 ostream 和 fstreambase 派生

类 fstream 从 iostream 和 fstreambase 派生

从而形成了支持文件 I/O 操作的、类似于基本流类族（以 ios 为中心）的一个流类族，其说明全部包含在头文件 fstream.h 中。

这个流类族实际上是基本流类族的扩充，它是在原来已提供的 I/O 操作的基础上再补充进与用户说明的流类对象（磁盘文件流）有关的一些特别的功能，其类的说明都是作为基本流类的派生类出现的，在进行文件 I/O 操作时，包含了头文件 fstream.h，同时也就包含了头文件 iostream.h 中的内容。

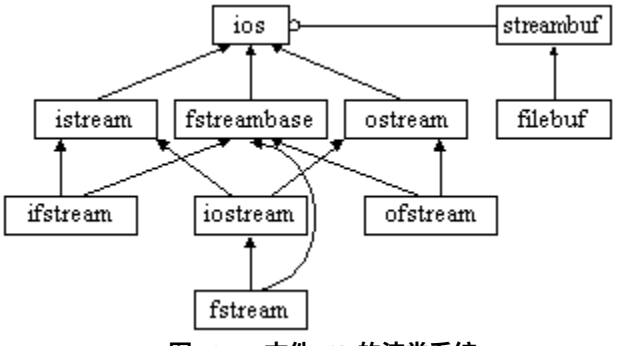


图 10.2　文件 I/O 的流类系统

上述专用于磁盘文件 IO 的流类（类型）所支持的功能如下：

ifstream：支持（提供）从本流类（对象）所对应的磁盘文件中输入（读入）数据；

ofstream：支持（提供）往本流类（对象）所对应的磁盘文件中输出（写出）数据；

fstream：支持（提供）对本流类（对象）所对应的磁盘文件进行输入和输出数据的双向操作。

注意，C++中没有预定义的文件流（类对象），即是说，程序中用到的所有文件流类对象都要进行自定义（规定对象名及打开方式，并将该流类对象与一个具体的磁盘文件联系起来等）。

10.4.1　文件的打开与关闭

为了对一个磁盘文件进行 I/O 即读写操作，必须首先打开文件，I/O 操作完成后再关闭。

对于 C++的 I/O 系统来说，打开工作包括在流（对象）的创建工作之中。流的创建通常是由对应流类的构造函数完成的，其中包括把创建的流与要进行读写操作的文件名联系起来，并打开这个文件（另外也可通过成员函数 open 来显式完成文件的打开工作）。例如：

```
ofstream outfile1("myfile1.txt");
```

将创建 ofstream 类的对象 outfile1；使流类对象 outfile1 与磁盘文件“myfile1.txt”相联系；并打开用于“写”的磁盘文件“myfile1.txt”。

也可按照如下方式来打开文件：

```
ofstream outfile1;                  //创建 ofstream 类的对象 outfile1
outfile1.open("myfile1.txt");       //通过成员函数 open 来打开文件
```

文件流可分别对于 ifstream 类、ofstream 类和 fstream 类说明其对象的方式创建。三个类的构造函数为：

```
ifstream::ifstream(char * name, int mode=ios::in,
                   int file_attrb=filebuf::openprot);
ofstream::ofstream(char * name, int mode=ios::out,
                   int file_attrb=filebuf::openprot);
fstream::fstream(char * name, int mode,
                 int file_attrb=filebuf::openprot);
```

第一个参数为文件名字符串（包括路径）。

第二个参数为对文件进行的 I/O 模式（访问模式），其值已在 ios 中进行了定义，使用含义如下：

```
ios::in                 //用于读入
ios::out                //用于写出
ios::ate                //打开并指向文件尾
ios::app                //用于附加数据打开并指向文件尾
ios::trunc              //如文件存在则清除其内容
ios::nocreate           //如文件不存在，则操作失败
ios::noreplace          //如文件存在，则操作失败
ios::binary             //二进制文件（缺省时为文本文件）
```

参数 mode 可缺省，当文件流为输入文件流时，其缺省值为 in；为输出文件流时，缺省值为 out。如果需要，可用上述枚举常量的一个组合来表示所需的访问模式（通过位运算符“|”来进行组合）。如：

```
ios::in|ios::out        -- 以读和写（可读可写）方式打开文件；
ios::out|ios::binary    -- 以二进制写方式打开文件；
```

ios::in|ios::binary　　-- 以二进制读方式打开文件。

第三个参数指定所打开文件的保护方式。该参数与具体的操作系统有关，一般只用它的缺省值“filebuf::openprot”。

例如，创建一个输出文件流（文本文件）并输出一串字符的程序：

```
// program 10_5.cpp
# include<fstream.h>
void  main(){
   ofstream  output("hello.txt");     //缺省打开模式 mode 时，隐含为文本文件
   output<<"Hello  world!"<<endl;
 }
```

执行完该程序后,可在当前目录(文件夹)下,看到新创建的文件“hello.txt”,并可通过任何一个编辑软件来看到该文件中的内容：“Hello world!”字符串。

对于文件的 I/O 操作可分为按文本（text）方式和按二进制（binary）方式两种。虽然文件和流都是对于信息的一种抽象，无论是数据还是程序代码，都把它们看成是字符或位的序列；不过，文本方式与二进制方式之间，除了下面两点区别：

（1）文本方式以字符（byte）为单位；

（2）文本方式没有压缩功能；

之外，还有第三个区别：

（3）文本方式有格式，它不仅以字符为单位，还以常量数字，单词（字符串）和行为单位。即它不仅要区分一个整数，一个浮点数，一个字符，一个字符串，同时它还要分行。文本（text）方式的名称就来源于此。

后面的 10.5 节还要进一步对这两种文件处理方式进行讨论。

用于输入和同时读写的文件 I/O 操作是类似的,只须对 ifstream 类和 fstream 类创建对象即可。

C++的 I/O 系统为了适应用户的习惯，也提供 open 函数和 close 函数来完成上述工作，其方式是：用 open()和 close()来代替构造函数和析构函数。其方法是：

```
// program 10_6.cpp
# include<fstream.h>
void  main(){
   ofstream  output;
    output.open ("hello.txt");
   output<<"Hello  world!"<<endl;
```

```
    output.close ();
}
```

注意，open 函数的参数与上面关于构造函数的说明一致。

10.4.2 使用插入与提取算符对磁盘文件进行读写

对文件的“读写操作”通常使用预定义的类成员函数来实现（随后介绍），但也可使用继承而来的插入和提取运算符“>>”和“<<”来进行，这基于如下的事实：

ifstream 类由 istream 类所派生，而 istream 类中预定义了公有的运算符重载函数“operator >>”，所以，ifstream 流（类对象）可以使用预定义的算符“>>”来对自定义磁盘文件进行“读”操作（允许通过派生类对象直接调用其基类的公有成员函数）；

ofstream 类由 ostream 类所派生，而 ostream 类中预定义了公有的运算符重载函数“operator <<”，所以，ofstream 流（类对象）可以使用预定义的算符“<<”来对自定义磁盘文件进行“写”操作；

fstream 类由 iostream 所派生，iostream 类由 istream 与 ostream 类共同派生，所以，fstream 流（类对象）可以使用预定义的算符“>>”和“<<”来对自定义磁盘文件进行“读” 与“写”操作。

还有一点需要注意：使用预定义的算符“<<”来进行“写”操作时，为了今后能正确读出，数据间要人为地添加分隔符（比如空格），这与用算符“>>”来进行“读”操作时遇空格或换行均结束一个数据相呼应。

实际上，插入“<<”和抽取“>>”运算符，以及标准流 cin、cout 等都是按文本方式来组织与定义的。

下述示例程序对同一个文本文件做了如下的 3 件事：1）往文本文件写数据；2）往文本文件尾部追加数据；3）从文本文件读出数据并显示在屏幕上。

```
// program 10_7.cpp
#include <fstream.h>
void main() {
    //1）往文本文件写数据
    ofstream outfile1("myfile1.txt");
        //创建流类对象 outfile1，并使它与文本型磁盘文件“myfile1.txt”相联系
    outfile1<<"Hello!...CHINA!    Nankai_University"<<endl;
        //使用算符“<<”对 outfile1 对象（所对应的文件）进行“写”操作
```

```
    outfile1.close();                    //关闭文件
    //2)  往文本文件尾部追加数据
    outfile1.open("myfile1.txt", ios::app);
        //打开用于“追加”的文本型文件“myfile1.txt”(第二参数 ios::app 的作用)
    int x=1212, y=6868;
    outfile1<<x<<" "<<y<<endl;
        //使用“<<”进行写(‘追加’)。注意,数据间要人为添加分割符
        //对 text 文件处理时,系统要对内外存的数据格式进行相应转换
    outfile1.close();                    //关闭文件
    //3)  从文本文件读出数据并显示在屏幕上。
    char str1[80], str2[80];
    int x2,y2;
    ifstream infile1("myfile1.txt");
        //创建流类对象 infile1;使它与文本型文件“myfile1.txt”相联系;并将其打开
    infile1>>str1>>str2;                 //使用“>>”来进行读操作(遇空格、换行均结束)
    infile1>>x2>>y2;
    infile1.close();
    cout<<"str1="<<str1<<endl;           //将读出的数据显示在屏幕上
    cout<<"str2="<<str2<<endl;
    cout<<"x2="<<x2<<endl;
    cout<<"y2="<<y2<<endl;
}
```

程序执行后的显示结果如下:

```
str1=Hello!...CHINA!
str2=Nankai_University
x2=1212
y2=6868
```

10.4.3 使用类成员函数对文件流(类对象)进行操作

本节介绍以下几个常用的对文件流(类对象)进行操作的类成员函数:get、put;read、write;以及 getline。

1. 类成员函数 get 与 put

使用类成员函数 get 与 put 可以对自定义磁盘文件进行读与写操作。

ostream::put 与 istream::get 的最常用格式为：

ostream& put(char ch);

功能：将字符 ch 写到自定义文件（ostream 流对象所联系的文件）中。

istream& get(char& rch);

功能：从自定义文件（istream 流对象所联系的文件）中读出 1 个字符放入引用 rch 中。

注意，put 实际上只是 ostream 类中定义的公有成员函数，但通常是通过其派生类 ofstream 的类对象来对它进行调用。同理，通常通过 ifstream 的类对象来直接调用 get。

下面看两个简单程序：程序一从键盘输入任一个字符串，通过 put 将其写到自定义磁盘文件“ft.txt”中，并统计且显示出输出符号的个数；而程序二通过使用 get 从那一文件中读出所写字符串，并统计所读符号的个数，一并显示在屏幕上。

```
// program 10_8.cpp
#include <fstream.h>
#include <stdio.h>
void main() {
    char str[80];
    cout<<"Input string:"<<endl;
    gets(str);                          //从键盘输入字符串（以“换行”结束输入）
    ofstream  fout("ft.txt");
    int i=0;
    while(str[i])    fout.put(str[i++]);    //通过 put 将 str 中各字符写到文件中
    cout<<"len="<<i<<endl;                  //显示输出符号的个数
    fout.close();
}
```

程序执行后的显示结果如下：

```
Input string:
12345 abcdef! ok!!
len=18
```

这个程序把字符串（包括空格和标点全部）送到了文件 ft.txt 中，并不必考虑其文本格式。

```
// program 10_9.cpp
#include <fstream.h>
```

```
void main() {                    //使用 get 从文件中读出符号串并显示在屏幕上
    ifstream fin("ft.txt");
    char ch;
    int i=0;
    fin.get(ch);                 //先读一个符号 ch，若文件为空（结束）时，fin.eof()将取真值
    while(!fin.eof()) {          //当读入的符号 ch 为有效符号（非文件结束）时继续
        cout<<ch;                //将读出的 ch 显示在屏幕上（对所读符号的“处理”）
        i++;                     //统计字符个数
        fin.get(ch);             //读下一个符号 ch
    }
    cout<<endl<<"len="<<i<<endl;
    fin.close();
}
```

程序执行后的显示结果如下：

```
12345 abcdef! ok!!
len=18
```

从例中可以看出，使用 get 与 put 函数对文件进行读、写十分简单，以每次一字节（8 位），顺序进行，文件中的各种特殊字符如空格等，它也一律按普通字符处理。实际上，利用 get 与 put 函数可以完成文件的拷贝工作，其大致程序“构架”如下所示（假设被拷贝的“源文件”以及拷贝到的“目的文件”的名字与路径均由命令行参数来提供）：

```
void main(int argc, char* argv[ ]) {
    …
    ifstream  fin(argv[1]);          //命令行参数 1 提供被拷贝的“源文件”
    ofstream fout(argv[2]);          //命令行参数 2 提供拷贝到的“目的文件”
    char ch;
    fin.get(ch);
    while(!fin.eof()){               //从头读到文件结束（当前符号非文件结束符时继续）
        fout.put(ch);                //对流 fout 进行写
        fin.get(ch);                 //对流 fin 进行读
    }
    …
}
```

请读者完成该程序并上机进行调试运行，我们把它留作一个练习。

2. 类成员函数 read 与 write

使用类成员函数 read 与 write 可以对文件进行读写操作。通常使用 read 与 write 对二进制文件（binary file）进行读写。

一般在处理大批量数据当需要提高 I/O 操作速度、简化 I/O 编程时，以二进制方式进行读写可显示出它的优越性。所谓二进制方式，就是简单地把文件视为一个 0、1 串，以字位（bit）为单位，不考虑文本格式，输入输出过程中，系统不对相应数据进行任何转换。

程序中用到的 read 与 write 类成员函数的常用格式及功能如下：

ostream::write

```
ostream& write( const char* pch, int nCount );
```

功能：将 pch 缓冲区中的前 nCount 个字符写出到某个文件（ostream 流对象）中。

istream::read

```
istream& read( char* pch, int nCount );
```

功能：从某个文件（istream 流对象）中读入 nCount 个字符放入 pch 缓冲区中（若读至文件结束尚不足 nCount 个字符时，也将立即结束本次读取过程）。

使用 read()，write()函数代替 get()和 put()，可以一次完成读写操作，例如在前面的例子中我们使用 write()代替 put()时，其中的语句：

```
int i=0;
while(str[i]) fout.put(str[i++]);   //通过 put 将 str 中各字符写到文件中
```

可用下面的语句取代：

```
fout.write(str, sizeof(str));
```

以下的示例程序先使用 write 往自定义二进制磁盘文件中写出如下 3 个“值”：字符串 str 的长度值 Len（一个正整数）、字符串 str 本身、以及一个结构体的数据，而后再使用 read 读出这些“值”并将它们显示在屏幕上。

```
// program 10_10.cpp
#include <fstream.h>
#include <string.h>
void main() {
    char str[20]="Hello world!";                    //准备写出的字符串 str
    struct stu{
        char name[20];
        int age;
        double score;
```

```
    } ss={"wu jun", 22, 91.5};                  //说明 ss 结构体变量，并赋了初值
    cout<<"WRITE to 'fb.bin'"<<endl;
    ofstream fout("fb.bin", ios::binary);       //打开用于“写”的二进制磁盘文件
    int Len=strlen(str);                        //求出字符串 str 的长度值 Len
    fout.write( (char*)(&Len), sizeof(int) );   //使用 write 函数将字符串长度值 Len 写出
    fout.write(str, Len);                       //使用 write 一次将 str 内容全部写出
    fout.write((char*)(&ss),  sizeof(ss));      //使用 write 将 ss 结构体的内容全部写出
    fout.close();                               //关闭文件
    cout<<"----------------------------"<<endl;
    //而后再使用 read 读出这些“值”并将它们显示在屏幕上
    cout<<"-- READ it from 'fb.bin' --"<<endl;
    char str2[80];
    ifstream fin("fb.bin", ios::binary);        //以读方式打开二进制文件“fb.bin”
    fin.read( (char*)(&Len), sizeof(int) );     //使用 read 将字符串长度值 Len 读入
    fin.read(str2, Len);                        //读入字符串本身放入 str2
    str2[Len]='\0';                             //增加结束符
    fin.read( (char*)(&ss), sizeof(ss) );       //读入数据放入 ss 结构体之中
    cout<<"Len="<<Len<<endl;
    cout<<"str2="<<str2<<endl;
    cout<<"ss=>"<<ss.name<<","<<ss.age<<","<<ss.score<<endl;
    fin.close();                                //关闭文件
    cout<<"----------------------------"<<endl;
}
```

程序执行后的显示结果如下：

```
WRITE to 'fb.bin'
----------------------------
-- READ it from 'fb.bin' --
Len=12
str2=Hello world!
ss=>wu jun,22,91.5
----------------------------
```

注意，与 text 文本文件不同，通过 write 写出到 binary 二进制文件中的各数据间并不需要（再写出一个）分割符，这是因为 write 与 read 函数的第二参数指定了读写长度。

3. 类成员函数 getline

使用类成员函数 getline 可以对文件进行“读”操作。

istream::getline 的最常用格式为：

istream& getline(char* pch, int nCount, char delim = '\n');

功能：从某个文件（istream 流对象）中读出一行（至多 nCount 个字符）放入 pch 缓冲区中，缺省行结束符为“\n”（也即第 3 参数的 delim 可用于显式指定别的行结束符）。

注意，getline 函数所操作的文件通常为 text 文本文件。

下述程序功能：读出本源程序文件“program 10_11.cpp”的各行并显示在屏幕上（假设本源程序已存放在“program 10_11.cpp”文件中）。

```
// program 10_11.cpp
#include <fstream.h>
void main() {
    char line[81];
    ifstream infile("program 10_11.cpp ");
        //打开用于“读”的文件“program 10_11.cpp”(即本源程序文件)
    infile.getline(line, 80);          //读出一行（至多 80 个字符）放入 line 中
    while(!infile.eof()) {             //尚未读到文件结束时，继续循环（处理）
        cout<<line<<endl;              //显示在屏幕上
        infile.getline(line,80);       //再读一行
    }
    infile.close();
}
```

程序执行后，将在屏幕上显示出上述那一源程序本身。

实际上，经常使用 getline 来读出某个 text 文本文件中的“一篇文章”（如可以是一个 C++源程序，或某一文本内容等），而后对读入的各行进行处理。

另外，也经常从键盘（即 cin 文件），通过 getline 来读入并处理行（符号串）信息。请分析并叙述如下程序的实现功能，并上机进行验证。

```
// program 10_12.cpp
#include <fstream.h>
#include <string.h>
void main() {
    char line[3][81];
    cout<<"-- input 3 lines --"<<endl;
```

```
    for(int i=0; i<3; i++)
        cin.getline(line[i], 80);
    cout<<"-- reverse output --"<<endl;
    for(i=0; i<3; i++){
        for(int j=strlen(line[i])-1; j>=0; j--)
            cout<<line[i][j];
        cout<<endl;
    }
}
```

10.5 text 文件与 binary 文件

前面已经多次谈及并使用与处理过 text 文件与 binary 文件。本节将通过实例进一步对它们各自的使用特点进行区分。

以 text 文件形式存储数据，优点是具有较高的兼容性，可利用任何一个文字处理程序进行阅读或编辑修改等。缺点是存储一批纯数值信息时，要在数据之间人为地添加分割符（否则将导致数据无法正确读出）；另外 text 文件通常比 binary 文件所占的磁盘空间大，且不便于对数据实行随机访问（因为每一数据所占磁盘空间的大小通常不相同）。另外，对 text 文件的输入输出过程中，系统要对内外存的数据格式进行相应转换。

以 binary 文件形式存储数据，优点是便于对数据实行随机访问（每一同类型数据所占磁盘空间的大小均相同，不必在数据之间人为地添加分割符），而且所占的磁盘空间通常比 text 文件要小。缺点是兼容性低，不可利用文字处理程序进行阅读或编辑修改等。另外注意，对 binary 文件的输入输出过程中，系统不对数据进行任何转换。

注意，是由程序员来决定将数据存储为 text 文件或者 binary 文件两种形式之一：通过将打开文件方式（访问模式）的 nMode 设置为“ios::binary”，可使打开的文件为 binary 文件形式；而在缺省情况下，打开的文件为 text 文件形式。通常将纯文本信息（如字符串）以 text 文件形式存储，而将数值信息以 binary 文件形式存储。

10.5.1 按用户设置的文件形式进行读写

1.对同一批数据按照两种文件形式进行处理

下述示例性程序的功能为：将 a 数组中准备好的 8 个 int 型数据，分别通过算符“ << ”依次写出到 text 文件 ft.txt 之中（注意各数据在文件中“长短”不一 -- 与每一数据所具有的‘位数’有关，另外注意数据间必须加入分割符 -- 空格）。而且还通过使用类成员函数 write 将这相同的 8 个 int 型数据依次写出到 binary 文件 fb.bin 之中（注意各数据在文件中“长短”相同 -- 仅与数据具有的类型 int 有关，而且数据间不需要加入分割符）。

另外，程序中通过使用无参的成员函数“ tellp() ”来获取当前已写出到各文件的位置信息，以确认每一数据在文件中所占的“位数”即字节数。ostream::tellp 之功能为：获取并返回“输出指针”的当前位置值（从文件首到当前位置的字节数）。

```
// program 10_13.cpp
#include  <fstream.h>
void main() {
        int a[8]={0,1,-1,1234567890};
        for(int i=4; i<8; i++)
                a[i]=876543210+i-4;
        //后 4 个 a[i]均由 9 位数字组成，在 text 文件中所占位数即字节数也为 9
        ofstream  ft("ft.txt");                          //打开 text 文件
        ofstream fb("fb.bin", ios::binary);              //打开 binary 文件
        for(i=0; i<8; i++) {                             //将 a 数组数据写到 ft.txt 和 fb.bin 之中
                ft<<a[i]<<" ";                           //数据间人为地添加了分割符 -- 空格
                fb.write((char*)(&a[i]), sizeof(a[i]));  //数据间不需要分割符
                cout<<"ft.tellp()="<<ft.tellp()<<",      ";  //显示 ft 文件输出指针当前位置
                cout<<"fb.tellp()="<<fb.tellp()<<endl;       //显示 fb 文件输出指针当前位置
        }
        ft.close();
        fb.close();
}
```

程序执行后的输出结果如下：

```
ft.tellp()=2,       fb.tellp()=4
ft.tellp()=4,       fb.tellp()=8
ft.tellp()=7,       fb.tellp()=12
ft.tellp()=18,      fb.tellp()=16
```

ft.tellp()=28,　　fb.tellp()=20

ft.tellp()=38,　　fb.tellp()=24

ft.tellp()=48,　　fb.tellp()=28

ft.tellp()=58,　　fb.tellp()=32

2.使用 read 与 write 对 text 文件进行操作时可能出错

前面提到过，通常使用 read 与 write 对 binary 型二进制文件进行操作，这是因为使用它们对 text 文件进行操作时可能出错。请分析下面的程序示例，将发现其中的出错原因：系统试图将一批连续整数（从 first 到 last）先写到文件中而后再读进来。但若使用 text 文件时，在 write 时有可能多写出了一些东西（如，回车换行符号等），这样将导致 read 时产生错误。

```
// program 10_14.cpp
#include <fstream.h>
void main(){
    int first,last;                    //往文件中写出一批整数 -- 从 first 到 last
    cout<<"first,last=?";
    cin>>first>>last;                  //键盘输入 first 与 last
    int f0=first;
    ofstream f1("file1.txt");          //按文本方式打开文件，导致 read 与 write 时可能出错！
    while ( f0 <= last ) {             //写出一批整数（从 first 到 last）
        f1.write( (char*)(&f0), sizeof(int) );
        f0++;
    }
    cout<<"共写出了 last-first+1="<<last-first+1<<"个数据"<<endl;
    cout<<"文件末位置：tellp()="<<f1.tellp()<<endl;
    f1.close();
    //将刚写出到文件中的数据通过 read 读入
    f0=first;
    ifstream f2("file1.txt");          //按文本方式打开文件，导致 read 与 write 时可能出错！
    while(f0<=last){                   //将保存在文件中的数据（从 first 到 last）逐一读入
        int tmp;
        f2.read((char*)(&tmp), sizeof(int));
        if(tmp!=f0)                    //若读入时产生数据错误，输出提示信息
            cout<<"ERR: f0,tmp="<<f0<<", "<<tmp<<endl;
        f0++;
```

```
    }
    f2.close();
}
```

程序执行结果为（注意输出时多写出了一个字节！读入数据时就出现了错误）：

```
first,last=?1 30
共写出了 last-first+1=30 个数据
文件末位置：tellp()=121
ERR: f0,tmp=26, 25
ERR: f0,tmp=27, 25
ERR: f0,tmp=28, 25
ERR: f0,tmp=29, 25
ERR: f0,tmp=30, 25
```

若对程序中按文本方式打开文件的两句进行修改：

```
    ofstream f1("file1.txt");                    //文本文件
    ifstream f2("file1.txt");                    //文本文件
```

修改为：

```
    ofstream  f1("file1.txt",   ios::binary);    //二进制文件
    ifstream  f2("file1.txt",   ios::binary);    //二进制文件
```

则执行后的结果将是正确的：

```
first,last=?1 30
共写出了 last-first+1=30 个数据
文件末位置：tellp()=120
```

3.采用 binary 文件形式对结构体数据进行读写处理

从键盘读入 n 个结构体数据（个数 n 由用户通过键盘输入来指定），使用 write 将这些结构体数据写出到某个自定义二进制磁盘文件中，而后再使用 read 读出这些结构体数据并进行处理（如，求出 n 个 score 的平均值 ave）。

```
// program 10_15.cpp
#include <fstream.h>
void main() {
    struct person {
        char name [20];
        int age;
        float score;
    } ss;                                        //ss 结构体
```

```
    int n;
    cout<<"n=? ";
    cin>>n;                                  //个数 n 由用户指定
    ofstream fout("f01.bin", ios::binary);   //打开二进制文件
    for(int i=0; i<n; i++) {                 //处理 n 个结构体（数据）
        cout<<"name, age, score=? ";
        cin>>ss.name>>ss.age>>ss.score;      //从键盘输入 ss 结构体数据
        fout.write( (char *)(&ss), sizeof(ss));  //将 ss 之数据写到文件中
    }
    fout.close();
    //使用 read 读出这些结构体数据并进行处理（如，求出 n 个 score 的平均值 ave）
    ifstream fin("f01.bin", ios::binary);    //打开二进制文件“ f01.bin ”
    float ave=0;
    fin.read( (char *)(&ss), sizeof(ss));    //读入 1 个结构体数据放入 ss
    while (!fin.eof()) {                     //若尚未到达文件末时，继续循环处理
        ave+=ss.score;                       //累加
        fin.read( (char *)(&ss), sizeof(ss));  //再读数据到 ss
    }
    fin.close();
    cout<<"n="<<n<<"   ave="<<ave/n<<endl;   //输出结果（平均值）
}
```

程序执行后的显示结果如下：

```
n=? 3
name, age, score=? zhang 22 85
name, age, score=? wang 23 92
name, age, score=? chen 22 87.5
n=3   ave=88.1667
```

10.5.2 对数据文件进行随机访问

使用类成员函数 write 与 read，并配合使用类成员函数 seekg、tellg 以及 seekp 和 tellp，就可以对文件进行“随机性”（非顺序性）的读写操作。

为了方便二进制方式的读写，系统提供了设置和读取文件的读写指针位置的函数。其中由类 istream 提供函数成员：

long tellg();

istream& seekg(long offset, int dir=ios::beg);

由类 ostream 提供了函数成员：

long tellp();

ostream& seekp(long offset, int dir=ios::beg);

其中参数 offset 给出一整数，表示相对偏移字节数，参数 dir 有三个取值（在 ios 类中说明）：

ios::beg //相对于文件开始位置

ios::cut //相对于指针当前位置

ios::end //相对于文件尾的位置

由第二个参数决定 offset 的数是从哪里开始计算的，参数 dir 有缺省值为 ios::beg。

它们的使用方法：

long pos=fout.tellp(); 把文件流 fout 的写指针的当前位置取出送到变量 pos。

fin.seekg(0, ios::beg); 把文件流 fin 的读指针定位到文件开头。

即是说，seekp 的具体功能为：将“输出指针”的值置到一个新位置，使以后的输出从该新位置开始。新位置由参数 offset 与 dir 之值确定。

seekg 功能：将“读入指针”的值置到一个新位置，使以后的读入从该新位置开始。新位置由参数 offset 与 dir 之值确定。

tellg 功能：获取“读入指针”的当前位置值（从文件首到当前位置的字节数）。

tellp 功能：获取“输出指针”的当前位置值（从文件首到当前位置的字节数）。

下面给出两个程序示例。

1．程序示例 1

从键盘输入 10 个 int 型数，而后按输入的相反顺序输出它们。要求使用如下的实现方法：使用 binary 文件，将键盘输入的数据依次存放在该文件中，而后使用随机访问方式（从后往前）逐一读入这些数据并显示在屏幕上。

```
// program 10_16.cpp
#include <fstream.h>
void main() {
    const int n=10;
    int x, i;
    ofstream fout("fdat.bin", ios::binary);
```

```
    cout<<"Input "<<n<<" integers:"<<endl;
    for(i=1; i<=n; i++) {                    //输入 10 个数据，并写出到二进制文件"fdat.bin"中
        cin>>x;
        fout.write((char*)(&x),sizeof(int));
    }
    fout.close();
    cout<<"---- The result ----"<<endl;
    ifstream fin("fdat.bin", ios::binary);
    for(i=n-1; i>=0; i--) {                  //从文件中（从后往前）读出数据显示在屏幕上
        fin.seekg(i*sizeof(int));            //读指针位置与 i 的大小有关（从后往前读）
        fin.read((char*)(&x),  sizeof(int));
        cout<<x<<"  ";                      //将所读数据显示在屏幕上
    }
    fin.close();
    cout<<endl;
}
```

程序执行后的输出结果为:

```
Input 10 integers:
1 2 3 4 5 6 7 8 9 10
---- The result ----
10 9 8 7 6 5 4 3 2 1
```

2．程序示例 2

使用 write 将多个 person 类型的结构体数据,写出到某个二进制磁盘文件的指定位置处,而后再使用 read 从另外指定的位置处读入某些结构体数据并显示在屏幕上。

具体处理程序中，有以下几点需要注意：

（1）person 型结构体数组 stu 的初值中，各数组分量（结构体）的 num（编号）相互不同，且从 1 至 10 连续分布。而后进行下述处理：使用 write 将 stu 数组中的 10 个结构体数据写出到二进制磁盘文件的指定位置处(以编号 num 为“序号”进行随机写)；再使用 read 从用户指定的位置处（由 num 值即“序号”来指定）随机读入某些结构体数据并显示在屏幕上。

（2）按编号 num 为“序”进行随机读写处理时，总是通过类似于下述的语句先求出一个读写位置偏移量 offs：“long offs = sizeof(person) * (recnum-1);”,而后再使用 seekg 或 seekp 将读写指针置于 offs 位置处（从文

件首“后推”offs 字节处），之后才使用 read 或 write 成员函数来进行具体对象数据的读写操作。

（3）程序中按如下形式的语句将 stu[i]结构体中的数据写出到二进制文件中：

```
fout.write( (char *)(&stu[i]), sizeof(person) );
```

该语句负责将结构体数据一次性地写出到与流对象 fout 相关联的 f01.bin 文件中。但还应注意：

① write 函数的第一参数（被写缓冲区地址）的类型必须强制转换为“char*”类型的（是 write 函数所要求的）；

② 一次就写出第二参数所指定的“sizeof(person)”字节大小的数据块；

③ 数据间不必人为地添加分隔符。

有关 read 函数的使用也有与此相类似的注意点。

```
// program 10_17.cpp
#include <fstream.h>
void main() {
    struct person {                 //person 结构体类型
        int num;                    //编号
        char name [20];             //姓名
        float score;                //成绩
    };
    //说明一个结构体数组 stu 并赋初值，注意，各结构体的分量 num（编号）相互不同
    person stu[10]={ {5, "zhou", 88.5}, {3, "sun", 89}, {7, "zheng", 91.5},
        {1, "zhao", 90.5},      {6, "wu", 94}, {2, "qian", 91},{9, "feng", 87.5},
        {4, "li", 84}, {8, "wang", 79}, {10, "chen", 90}   };
    int  recnum;
    ofstream fout("f01.bin", ios::binary);
    for(int i=0; i<10; i++) {       //使用 write 将 stu 数组的 10 个结构体数据
            //写到文件的指定位置处（以 num 为“序号”进行随机写）
        recnum=stu[i].num;
        long offs=sizeof(person)*(recnum-1);
            //计算出第 recnum 个结构体在磁盘文件中的位置 offs
        fout.seekp(offs);
        fout.write( (char *)(&stu[i]), sizeof(person) );      //将结构体数据写出到文件中
    }
```

```
    fout.close();
    //使用 read 从另外指定的位置处（num“序号”处）读入某些结构体数据并显示
    ifstream fin("f01.bin", ios::binary);
    cout<<"-- random reading, input num --\n";
    cout<<"num=? (1--10/otherwise exit):";
    cin>>recnum;                    //键盘输入 num“序号”
    person tmp;
    while (recnum>0 && recnum<11) {
            //限制编号在 1～10 之间（超出范围自动结束程序）
        long offs=sizeof(person)*(recnum-1);
            //计算出编号为 recnum 的结构体在文件中的位置 offs
        fin.seekg(offs);
        fin.read( (char *)(&tmp), sizeof(person) );          //从文件读结构体数据到 tmp 中
        cout<<tmp.num<<"    "<<tmp.name<<"    "<<tmp.score<<endl;    //显示在屏幕上
       cout<<"num=?  (1--10/otherwise exit):";
        cin>>recnum;                //键盘输入新的 num“序号”
    }
    fin.close();
}
```

程序执行后，屏幕显示结果为：

```
-- random reading, input num --
num=? (1--10/otherwise exit):4
4    li    84
num=? (1--10/otherwise exit):9
9    feng    87.5
num=? (1--10/otherwise exit):2
2    qian    91
num=? (1--10/otherwise exit):6
6    wu    94
num=? (1--10/otherwise exit):11
```

10.6　字符串流

本节简单介绍 C++流类库中预定义的两个字符串流类：ostrstream 和 istrstream。其中的 ostrstream 由 ostream 所派生，而 istrstream 则由 istream 派生而来。

通过 ostrstream 类的使用，可将不同类型的信息转换为字符串，并存放在（输出到）一个用户设定的字符数组中；而通过 istrstream 类的使用，则可将用户字符数组中的字符串取出（读入），而后反向转换为各种变量的内部形式。既是说，字符串流类对象并不对应于一个具体的物理设备，而是将内存中的字符数组看成是一个逻辑设备，并通过“借用”对文件进行操作的各种运算符和函数，最终完成上述所谓的信息转换工作。这类似于 C 语言的库函数 sprintf 以及 sscanf 所要实现的功能。

使用字符串流类时，必须包含头文件 strstrea.h。

1．ostrstream 类的使用

只介绍该类中的两个成员函数：构造函数 ostrstream::ostrstream 和另外一个函数 ostrstream::pcount。

（1）构造函数 ostrstream::ostrstream

该类最常用的构造函数的一般格式为：

```
ostrstream( char* str, int n, int mode = ios::out );
```

其中的 str 为字符数组，它将作为输出的“目的地”（输出数据将存放在 str 中）。n 用来指出 str 中最多能够存放多少个字符（注意，一旦长度达到了 n，则不可再接收其他任何字符了）。第三参数 mode 指明流的打开方式，默认为 out（注，还可显式指定 ios::ate 或 ios::app，两者功能相同，都将把输出数据“附加”到尾部）。

注意，由于 ostrstream 是 ostream 的派生类，所以，凡可以用于 ostream 类的运算符或成员函数，都可以使用到该 ostrstream 派生类的对象上（用来往与该派生类对象关联的作为输出“目的地”的 str 字符数组中填充数据）。再一点是，也可使用 ostream::seekp 函数来指定写出位置（以实现随机写功能）。

（2）ostrstream::pcount

使用格式：int pcount() const;

功能：返回一个数值，表示目前已经输出到字符串流即字符数组中的字符个数（字节数）。

2．istrstream 类的使用

只介绍该类中最常用的两个构造函数 istrstream::istrstream。

（1）一参构造函数

```
istrstream( char* str );
```

由参数 str 指定了一个以'\0'为结束符的字符串（字符数组），它的“整体字符”将作为“输入源”。

（2）二参构造函数

istrstream(char* str, int n);

由参数 str 指定字符数组，它将作为“输入源”，由第二参数 n 指出仅使用 str 的前 n 个字符（而不是“整体字符”）。

注意：

① 二参构造函数时，并不要求 str 中必须具有'\0'结束符号；

② 若 n=0，则假定 str 为一个以'\0'为结束符号的字符串（字符数组）。

由于 istrstream 是 istream 的派生类，所以，凡可以用于 istream 类的运算符或成员函数，都可以使用到该 istrstream 派生类的对象上（用来从与该派生类对象关联的“输入源”即 str 字符数组中读入数据）。再一点是，也可使用 istream::seekg 函数来指定读入位置（以实现随机读功能）。

下面给出一个程序示例。

```
// program 10_18.cpp
#include <strstrea.h>
#include <iomanip.h>
void main() {
    char str[122];
    ostrstream outstr(str, sizeof(str));          //说明“输出型”字符串流类对象 outstr，
                //并使它与字符数组 str 相联系（输出数据将存放在 str 中）
    cout<<"outstr.pcount()="<<outstr.pcount()<<endl;
                //pcount 函数返回目前已经输出到字符串流中的字符个数（字节数）
    for(char c='A'; c<='G'; c++)
        outstr<<c;          //通过运算符“<<”输出数据到与 outstr 相关联的 str 数组中
    cout<<"outstr.pcount()="<<outstr.pcount()<<endl;
    for(int i=1; i<=15; i+=2)
        outstr<<setw(4)<<oct<<i;              //对 outstr 对象使用格式控制符
    cout<<"outstr.pcount()="<<outstr.pcount()<<endl;
    outstr.setf(ios::fixed);                  //使用格式控制函数
    outstr.precision(2);
    for(i=1; i<=5; i++)
        outstr<<"    "<<2.222*i;
    outstr<<ends;                             //添加串尾结束符，等同于：outstr<<'\0';
```

```
    cout<<str<<endl;
    cout<<"outstr.pcount()="<<outstr.pcount()<<endl;
    istrstream instr(str);                    //说明“输入型”字符串流类对象 instr，
                    //并使它与字符数组 str 相联系（输入数据将来自 str 数组）
    char s1[30];
    instr>>s1;          //通过运算符“>>”从与 instr 相关联的 str 数组中输入数据到 s1
    cout<<"s1="<<s1<<endl;
    instr.seekg(20);                          //对类对象 instr 使用 seekg 函数（随机读功能）
    instr.setf(ios::oct);                     //使用格式控制函数
    for(i=1; i<=4; i++){
        int d;
        instr>>d;
        cout<<d<<"    ";
    }
    instr.seekg(40);                          //重新设置读取位置（随机读）
    double x,y,z;
    instr>>x>>y>>z;
    cout<<"x="<<x<<", y-z="<<y-z<<endl;
}
```

程序执行结果为：

```
outstr.pcount()=0
outstr.pcount()=7
outstr.pcount()=39
ABCDEFG     1     3     5     7    11    13    15    17    2.22    4.44    6.67    8.89    11.11
outstr.pcount()=71
s1=ABCDEFG
7    9    11    13    x=2.22,  y-z=-2.23
```

10.7 其它输入输出控制函数

在头文件 iostream.h 中定义的几个基本流类 ios，istream，ostream 中，还定义了一些与输入输出操作有关的状态量和控制用的成员函数，有的在 I/O 操作中常常用到。

10.7.1 I/O 操作状态控制

对每一个 istream 流或 ostream 流来说，系统都提供一个与其关联的 I/O 操作状态标志字（也称 I/O 操作状态字）。通过设置或测试该状态字可使用户对流的 I/O 操作状态进行控制，以便及时发现有关错误并进行有效的干预与处理。

I/O 操作状态字是在类 ios 中定义的，它的各位的状态由如下的标志位（常量）来描述：

```
ios::goodbit=0x00          //流处于正常状态（没设置任何的状态标志位）
ios::eofbit=0x01           //输入流结束（到达文件末尾）
ios::failbit=0x02          //I/O 操作失败（会使随后的操作也失败）
ios::badbit=0x04           //失去了流缓冲区的完整性（流被破坏）
```

注：某些编译系统还提供操作状态标志位 ios::hardfail=0x80，表示 I/O 出现了致命错误（注：VC6 没提供该标志位）。

I/O 操作状态字的状态可以通过 ios 类中提供的检测函数来查阅（注，系统总根据最近一次 I/O 操作的结果来自动对状态字的相应位进行置位），也可通过状态设置函数来人为地对状态字进行设置。相关函数有：

```
int  good();               //I/O 流正常（没设置任何的状态标志位）返回非 0，
                           //否则返回 0
int  eof();                //到达了文件末尾（状态字的 eofbit 位被置 1）则返回非 0，
                           //否则返回 0
int  fail();               //流状态字的 failbit、badbit 或 hardfail 中任一个位被置 1，
                           //则返回非 0（意味着随后的操作将失败），否则返回 0
int  bad();                //流状态字的 badbit 或 hardfail 位中任一个被置 1，
                           //则返回非 0（严重错误，流被破坏），否则返回 0
int  rdstate();            //返回当前 I/O 操作状态字
int  operator!();          //与函数 fail()功能相同
void  clear(int ef=0);     //无参调用可清除全部出错信息（将状态字的各位均
                           //清为 0）；带参，可人工将某些状态标志位设置为 1。
```

注意，clear 函数的参数 ef 值可以通过使用状态标志位常量或者它们的组合（通过使用位运算符“|”）来设定，如，

```
fin.clear(fin.rdstate()|ios::failbit);
```

它用来在流 fin 的 I/O 操作状态标志字中设置 failbit 位（且不改变其他位的值）。

另外注意，fail 函数与 bad 函数的使用区别是：当 fail 函数返回值为真（非 0），但 bad 函数的返回值为假（0 值），此时的流尚未被破坏，该错误是可以被恢复的（如可通过使用 clear 函数清除出错标志位后仍可继续处理）;但若 bad 函数的返回值为真（非 0），表示发生了严重错误，此时的流已被破坏，该错误不可被恢复，此时应停止 I/O 操作。

下面列举这些函数的一些使用方式：

（1）把 good()；eof()；fail()；bad()作为布尔函数，用 if 语句实现情况处理程序块。

例如：

```
ifstream fin("f1.txt");
if(fin.good())       //表示打开文件成功，可继续进行后续处理
    ...
```

又例如：

```
while(!fin.eof())   //表示文件尚未结束，可进行随后的操作处理
    ...
```

（2）用函数 rdstate()作为开关语句的分情形表达式，如：

```
switch(fin.rdstate()){
    //按照输入流 fin 的 I/O 操作状态字的当前值来分支
      case ios::goodbit:
            //对正常情况的处理
      case ios::eofbit:
            //对输入流结束的处理
      case ios::failbit:
            //读操作失败的处理（流尚未破坏，该错误可被恢复）
      case ios::badbit:
            //对严重错误的处理（流被破坏，不可恢复）
      default:
            //其他情况的处理
   };
```

（3）在“ios::failbit”被置位，即发生了一次操作失败的错误后，可输出某些提示信息，报告可能丢失操作数据，然后调用清除函数 clear()恢复 I/O 状态字为 0，即正常状态，继续进行 I/O 操作。

但在“ios::badbit”被置位，即发生了一次严重错误时，首先也应输出某些提示信息进行报告，但不可以继续进行后续的 I/O 操作。

10.7.2 其它 I/O 控制

本节介绍另外几个有用的 I/O 控制函数。

1．流的刷新

C++系统提供的流多为缓冲流，缓冲 I/O 操作可提高效率，但有时也会产生问题，例如：

```
char  pstr[100];
cout<<"Enter Password:";
cin>>pstr;
```

是常用的编程形式，但由于用来显示字符串“ Enter Password: ”的输出语句执行之后，输出串被送入到一个可能未满的缓冲区中，则有可能不会立即显示到屏幕上。这时执行输入语句时，未能按预想的那样在屏幕上看到提示“ Enter Password: ”，它可能在输入完成之后的某个时候才被输出。为此 C++的 I/O 系统提供了刷新函数：

```
ostream& ostream::flush();
```

于是，上面的接收口令的程序段可写为：

```
char pstr[100];
cout<<"Enter Password:";
cout.flush();
cin>>pstr;
```

如此，可保证提示及时地显示在屏幕上。

如上节所介绍的，刷新函数可被系统提供的（类外）控制符 flush 所代替，上面的处理可以改写为更方便的形式：

```
cout>>"Enter Password:"<<flush;
cin>>pstr;
```

2．流的连接

流的连接（tie：系到一起）也与刷新有关，在 ios 类中提供成员函数：

```
ostream *  ios::tie(ostream *);
```

该函数通常由一输入流（对象）调用，以一个输出流为参数。调用结果是把这个输出流连接到输入流上,同时返回原来与该输入流连接的输出流的地址。例如：

```
ostream * old=cin.tie(&cout);
```

它把 cout 连接到 cin，且把原来与 cin 相连的输出流地址传给 old。

输出流 outst 与输入流 inst 连接之后，在流 inst 的每次抽取（输入）操作之前都要对流 outst 进行刷新。因此，在调用 cin.tie(&cout);之后，上文中 flush();和 flush 的使用就没有必要了。

C++的 I/O 系统，已在缺省情况下把 cout 连接到 cin 上，因此，直接使用

```
cout<<"Enter Password:";
cin>>pstr;
```

也不会出现不显示的问题，因为已作了函数 cin.tie(&cout);的缺省调用。

3．输入流的简单操作

对于输入流除了采用抽取“ >> ”操作顺序读入之外，类 istream 中还给出几个简单的特别处理函数：peek(); ignore(int); 和 putback(char)。

函数 peek()用来在“ 不输入 ”的条件下读出输入流的当前字符，这里所谓“ 不输入 ’就是用 peek()函数读了当前字符后，该字符仍为输入流的当前字符，下次抽取（“ >> ”）的字符仍是它。其用法简单：

```
char ch=cin.peek();
```

函数 ignore(int)用来在输入流中跳过若干字符，其用法为：

```
cin.ignore(5);
```

表示其后的五个字符将被忽略。

函数 putback(char)用来把刚从输入流中抽取的字符再返退回去，其用法是：

```
char c;
cin>>c;
cin.putback(c);
```

相当于一次 undo 操作。

思考题：

1. C++语言的流类库和 C 语言的 I/O 输出语句相比有什么优点？
2. 流类库是通过什么机制使得 I/O 变得简单明了？
3. 什么是类型安全？类型安全是怎样实现的？
4. 流类库是怎样实现易于扩充性的？
5. 详述文件的概念，文件的种类有几种？程序中的文件概念和普通文件的概念有什么不同？
6. 什么是流？流的概念和文件的概念有什么异同？

7. C++中为流定义了哪些类？它们之间的继承关系是什么？
8. C++为用户预定义了那几个标准流？分别代表什么含义？
9. 简述 C++的 I/O 格式控制。
10. 试述文件 I/O 流类的类结构。
11. 简述文件的打开和关闭的过程和步骤。
12. 文件的读写方式有哪几种？分别完成什么功能？

练习题：

1. 输出下述的数据：

   4343**fdjf，0x11，3534.34343，2.34334E+03，

2. 先建立一个文本文件，然后利用本章所述的 C++的流类库对这个文件进行读写操作。
3. 定义一个 3*3 的矩阵类，并在这个矩阵类上实现矩阵的加法、减法和乘法运算。计算任意从键盘上输入的两个 3*3 阶矩阵的各种运算的结果，将结果在屏幕上输出。
4. 按文本方式和二进制方式分别对一个文件进行读写，并比较两者的不同。
5. 已知一个类 CStudent，类中包含一个学生的基本数据如下：

   编号，姓名，班级，性别，年龄，数学成绩，语文成绩，外语成绩，奖惩纪录。

   请设计一个简单的数据文件，能够存储相应学生的情况，当用户从屏幕上输入一个学生的相应的信息后，将该信息存入到这个数据文件中。
6. 为 CStudent 类提供输出运算符“<<”，使得该运算符能够完成将一个学生的信息按如下格式输出到屏幕上：

   D001 李平 1 男 16 89 98 94 三好学生

7. 在上两题的基础上编制一个简单的学籍管理软件。要求完成以下的功能：
   (1) 能够从屏幕上读取一个学生的信息并将信息存入到数据文件中。
   (2) 能够将指定的学生的信息从数据文件中删除。
   (3) 能够按编号、姓名对学生的信息进行检索并将检索结果输入到屏幕上。
   (4) 可以统计全部学生的成绩。
   (5) 要求有错误提示功能，例如性别只能输入男女，输入错误应提示重新输入。
   (6) 如果检索不到相应的信息应提示用户。
8. 请为操作系统编写一个扩展命令 CompFile，用户输入两个文件名，该命令比较两个文件是否相同，并将结果输出到屏幕上。

第十一章 用 C++语言设计面向对象程序

由于 C++语言支持抽象数据类型、类的继承机制以及以虚函数为代表的多态性，使得 C++语言成为面向对象程序设计（OOP）的有力工具，虽然我们已经在各章（特别是第七章以后）分别介绍了其各种功能和机制的编程技术，但仍然不足以清楚地展示一个完整的面向对象程序设计的总体过程。本章拟通过自顶向下地介绍电梯仿真系统的用 C++语言描述的 OOP 程序，向读者展示初步的面向对象的编程方法，并从中体现 OOP 的风格和特征。

面对一个实际问题，面向对象编程的过程一般分为三个阶段，即面向对象分析（OOA），面向对象设计（OOD）和面向对象实现（OOI）。前两个阶段与语言无关。

OOA 阶段的任务是从问题陈述中把涉及问题领域和系统行为的对象，类及类之间的联系抽取出来。例如在办公大楼电梯仿真系统中，搭乘电梯的人，在楼间运行的电梯和各个楼层，它们是实际问题中的对象和类。

OOD 阶段则需进一步对系统进行总体结构设计，把数据设计与过程设计结合为一体，封装为模块，同时为实现问题需求可能还需要引入其它一些对象和类。

OOI 阶段才是用 C++语言进行编程实现的过程。关于 OOA，OOD 的内容本书不可能详细讨论，在全部本科的学习过程中，学生将有机会在有关 OOP 的后续课程（如数据结构、数据库系统原理、面向对象编程技术、软件工程等）中得到系统的训练。

本章介绍两个 OOP 编程实例，目的是脱离开语法讲解的范围，让读者直接面对问题，学习如何编写面向对象的 C++程序。这里要说明的是，在实际问题编程时，可能与我们介绍的情形有区别，即由于 OOP 允许充分利用程序的可重用性，依靠一个越来越完备的标准模板库（STL），程序员在开发一个新的软件时，新编写的类及对象可能只占程序的一小部分，充分利用系统提供的类模板（容器、迭代器、算法类）或用户过去编写的类或模板，可以大大缩短代码开发时间，把主要精力用在 OOA 和 OOD 上面，这样的编程既可靠又快速，是面向对象编程的一大优点。

11.1 一个 Palindrome 的识别程序设计

这是一个简单问题。“ Palindrome ”译为“ 回文 ”，指顺读和反读都一样的单词或字符串。识别回文就是判断输入的字符串（不包括空格）是否是一个迴文，即正向和逆向阅读都是同一个串，如 abcba，dstt sd，就是“ 迴文 ”,而 abcd，AOP 不是。

问题的关键是采用一种数据结构可以方便地反读字符串,我们选用具有‘ 后进先出 ”特点的栈（stack）结构，为此设计一个 Stack 类。其次，为了从输入的字符串中去掉空格，设计一个用户定义的函数 dblank ()。

Stack 类的设计如下：

```
//astack.h
#include <iostream.h>
#include <stdlib.h>
const int MaxSize=50;
class Stack{
  private:
    char slist[MaxSize];
    int top;
  public:
    Stack (void){top = 0;};
    ~Stack (void){};
    void Push (const char &item);
    char Pop (void);
    bool StackEmpty (void) { return top == 0;};
    bool StackFull (void) { return top == MaxSize;};
};
void Stack::Push (const char &item){
  if (StackFull ())
    cout<< "Stack overflow! "<<endl;
  else{
    slist[top] = item;
    top++;
  }
```

```
}
char Stack::Pop (void){
   if (StackEmpty ()){
        cout<< " an empty stack! "<<endl;
       return ;
   }
   return slist[--top];
}
```

函数 dblank()设计为：

```
void dblank (const char * str, const char * dblankstr){
  char * pstr = str;
  char * pdblankstr = dblankstr;
  while(*pstr != '\0'){
       if (*pstr !=' '){
            * pdblankstr = * pstr;
              pdblankstr++;
        }
        pstr++;
   }
   *pdblankstr='\0';
}
```

回文识别程序可以写为：

```
//program2_2.cpp
 #include <iostream.h>
 #include "astack.h"
 void dblank(char *,char * );
 void  main(){                                          //主函数部分
     Stack    S；
     Char    palstring[80], dblankstring[80], ch；
     int    i = 0；
     bool    ispalindrome = true；
    cin  get(palstring,80);                             //输入一个字符串,以'\0'结尾
    /*cin>>会把空格，制表键，回车都做为结束符，所以就不能用它输入含空格的字符串，而 cin.get 仅以回车做结束符。*/
```

```
    dblank(palstring, dblankstring)；                    //删去串中的空格
    while (dblankstring[i] != '\0'){
        S.Push(dblankstring[i])；
        i++;
    }
    i = 0;
    while(!S.StackEmpty()){
        ch = S.Pop();
        if (ch != dblankstring[i]){
            ispalindrome =false;
            break;
        }
        i++;
    }
   if  (ispalindrome)
        cout<< "\'"<<palstring<< "\'"<< "is a palindrome"<<endl;
    else
        cout<< "\'"<<palstring<< "\'"<< "is not a palindrome"<<endl;
}

void dblank(char * str,char * dblankstr){
   ……
}
```

这个程序虽然不大，但却包含了用户定义的类 Stack 和用户定义的函数 dblank(),它既不属于 SP 框架，也不属于 OOP 框架，而是一种混合框架，为了使得 C++程序成为严格的 OOP 框架程序，可以有两种解决办法：

1．一个比较大的 C++程序往往有许多全局变量和一些用户定义的类外函数（在下一个实例中可以看到），可以把它们组成一个由若干公有成员走出的类。

2．在这个小型实例中，唯一的类外函数 blank(),其功能也可以在主函数中实现：

```
  #include <iostream.h>
#include "astack.h"
void dblank(char *,char * );
 void  main(){                                       //主函数部分
```

```
    Stack    S；
    Char    palstring[80], dblankstring[80], ch；
    int    i = 0；
    Bool    ispalindrome = true；
    cin  >>  palstring；                                    //输入一个字符串，以'\0'
结尾
    while (palstring[ i] != '\0'){                          //进栈，同时删去串中的空
格
        if (palkstring[i] != ' ' ) S.Push(palkstring[i])；
        i++;
    }
    i = 0;
    while(!S.StackEmpty()){
       ch = S.Pop();
       if (palstring[i] == ' ') i++;
        if (ch != dblankstring[i]){
            ispalindrome =false;
            break;
        }
        i++;
    }
   if  (ispalindrome)
        cout<< \ ' '<<palstring<< \ ' '<< "is a palindrome"<<endl;
    else
        cout<< \ ' '<<palstring<< \ ' '<< "is not a palindrome"<<endl;
}
```

11．2 楼宇电梯系统仿真程序的设计

在实际工作中，往往需要由计算机来模拟一个系统的运行过程。例如工厂的生产中，由原材料、工人、设备、产品等多因素构成的生产过程仿真；一个电厂的电力的生产、供应、管理的全过程的仿真；一个城市的道路交通系统，对车流量、道路进行管理，其中包括各种道路、行人、红绿灯系统、警察、各

种车辆等等，它们在运行中全处在动态变化之中。这样的一个仿真系统的开发，有利于有关部门安排警力，改进红绿灯系统的设置，在最紧张的地段拓宽路面，改行单行线，建设新道路、立交桥、地铁等等。

当然，设计这样的仿真程序不是一件很容易的事。C++语言所支持的面向对象程序设计技术正是为规模较大、比较复杂的问题编制高质量、可读性好、可维护性好、可重用性好的软件最强有力的方法。

我们的问题是一个规模稍小一点的仿真问题，它是要编写一个程序，模拟办公大楼中全部电梯的工作过程。这个仿真程序可以用来监测系统运行情况，改善大楼管理，它也可以看成是一种游戏程序。

下面让我们对这个系统作一个初步描述：

·办公大楼有若干层（例如，十层），每层有电梯，同时有步行楼梯；

·全楼有若干部（例如，不多于 10 部）电梯同时供使用，电梯容量为 24 人，速度每上下一层需 5 秒，在某一层停下至少 15 秒。其运行状态可分：向上、向下、停止，当前乘客数，当前所在层数。它设有一个“按钮数组”，例如第五层的按钮按下，意味着有乘客在第 5 层到达目标层，等等。

·在楼的每一层，有电梯数，有按钮表示有人等待向上或向下，由若干人在等待，有若干电梯在本层停下，等等。

·在大楼中（包括进出）的总人数不超过 500 人，每个人站在电梯前有个目标层，他有一个最大的忍受等待时间，因为他可以选择电梯或是步行走楼梯，等等。

·还有下面若干假设：

在每个时间段要进大楼的人数在 0～199 之间随机取值；

用电梯的每个人的目标层在 1～10 之间取值；

一个人在进电梯或改走楼梯之前的等待时间在 180～360 秒范围内随机发生；

一个人到达目标层后第二次再乘电梯中间的工作时间在 400～6600 秒间随机取值。

以及其它假设。

总之，在我们设计仿真程序之前应对整个系统有个比较量化的了解，同时应对整个系统的几方面因素有一定的划分。

例如，可以把整个大楼视为系统的整体，在这个系统中运行的有不同类的对象：

电梯集合，按自己规律不断运行的若干电梯，仿真系统应随时显示各个电梯的当前状态，同时在一定时刻应提供某些仿真统计结果。

楼层集合，各个楼层的状态也是在不停地变化，停下的电梯，等电梯的人群、各按钮的状态，各层状态虽各不相同但有共同的属性。

人员集合，所有大楼中的人，都处在不断变化着的各种状态之中，等待，搭乘（在电梯中），工作，步行（楼梯）等。在等待的人分布在各个楼层，以不同的目标层为目的，他们分乘不同的电梯，在不同的楼层工作，经过不同的时间间隔后再次使用电梯进入等待状态。

整个仿真过程按一定的规则和随机性假设来运行。

系统的基本要求是：设定和控制仿真时间，管理整个系统的正常运行，随时显示系统的当前状态，输出一段仿真运行后的结果。

这样的一个仿真系统的运行过程大致可以用下面的程序来描述：

```
//simulation.cpp
#include"simulation.h"
#include<…>
#include<…>

simulation    simone;                  // 一次仿真

void main (void){
   simone.display ();                  // 显示初始画面
   while (simone.continues()){         // 仿真过程
      simone.perform();                // 整个系统动作一次
      simone.display();                // 动态显示
   }
   simone.results();                   // 输出仿真结果
}
```

这是一个抽象的仿真程序的运行过程，类 simulation 的内容，包括这里已经引用的成员函数 display(),continues(),perform(),result(),可以分别逐步进行编程，让我们就以此作为自顶向下(Top-Down)的 oop 设计过程的开始。

11.2.1 电梯仿真程序的初步框架

根据仿真程序的基本功能,设计一个 simulation 类,作为系统的初步框架，它只涉及系统与用户相关的一些属性，例如，设定仿真时间，控制仿真过程结束等等。至于仿真过程的细节，还要经过逐步求精的过程一步步地完成。在这

个过程的每一步都是可以上机调试的。

现在目标已经明确，就是要写出上节末程序中要求的函数，主要包括 display ( ) ,continues ( ) ,perform ( ) 和 results ( ) 作为函数成员的类的说明和定义。

我们按照写大程序的习惯，把程序分为两部分：

simulation.h：类的说明部分

simulation.cpp：类的实现部分

当然，主函数 main ( ) 也可以包含在里面。

```
//simulation.h        这就是所谓的头文件
#ifndef   _SIMULATION_H
#define   _SIMULATION_H
class simulation{
  long timeSimulate;          //仿真时间（秒数）
  long timeRemaining;         //剩余时间（秒数）
public:
   simulation () { timeSimulate = 3600; timeRemaining = 3600;}
   simulation ( long secs) { timeSimulate = secs; timeRemaining = secs;}
   bool   continues (void);
   void   SetTime (long secs);
   long   getTim (void);
   void   reduceTime (long secs);
   void   perform (void);
   void   display (void);
   void   results (void);
};
#endif  //_SIMULATION_H
```

程序说明：

⑴ 第 2～3 行和 18 行的编译预处理命令的功能为：

第 2 行：如果未对_SIMULATION_H 作出定义，则向下一行进行，否则跳过这段程序直到第 18 行以后。

第 3 行：把_SIMULATION_H 定义为空串。

第 19 行：与第 2 行呼应，如条件不成立，跳过这段程序。

它们的作用是避免这个头文件的内容在多次被包含的情况下重复运行。在这个实例的许多模块中都有这样的处理，不再说明。

⑵ 类 simulation 有两个私有数据成员：timeSimulate 是一个长整数，纪录设定的仿真时间；另一个 timeRemaining 则纪录剩余的仿真时间长度，当这个值变为 0 时仿真停止。

⑶ 类 simulation 说明了 9 个公有函数成员，除了构造函数和上文中要求的 continues ()，perform ()，display () 和 results () 之外，另说明了 SetTime ()，getTime ()，reduceTime () 三个可以在运行中改变和读取时间设置的操作。

⑷ 从这段程序可以看出，类 simulation 的任何一个实例或对象都是一次实际的仿真过程。

下面是类 simulation 的定义或称实现。

```
//simulation.cpp
#include "simulation.h"
#include <iostream.h>
#include <conio.h>

bool    simulation::continues (void){
  if (timeRemaining>0)
     return true;
  else   return false;
} //根据剩余时间判定是否继续仿真

void    simulation::SetTime(long secs){
   timeSimulation=secs;
   timeRemaining=timeSimulate;
} //运行中设置仿真时间

long    simulation::getTime (void){
  return timeRemaining;
} //读出剩余时间

void   simulation::reduceTime (long secs){
  if (secs>timeRemaining)
       timeRemaining=0;
   else
```

```
        timeRemaining-=secs;
} //减少仿真时间

void   simulation::perform(void){
   cout<<"Simulation! Press <space> to continume";
   while (getch()!=' ');
   reduceTime (600);
} //这是一次虚拟的仿真运行动作

void   simulation::display (void){
   cout<<endl<<"Time remaining";
   cout<<timeRemaining<<"secs";
} //显示信息

void   simulation::results (void){
   cout<<endl<<"Simulation results:";
   cout<<endl<<"Time at start:"<<timeSimulate<<"secs";
   cout<<endl<<"Time at end:"<<timeRemaining<<"secs";
} //显示仿真结果
```

⑴ 这是一个抽象的框架程序，是自顶向下设计过程的开始，但它又是可运行的，读者可以按上节的 main()来执行它。

⑵ 设置两个构造函数，当创建一个 simulation 类型的对象时，第一个构造函数自动地为其设置缺省的模拟时间为 1 小时即 3600 秒,也可以用第二个构造函数通过参数设定仿真时间。

⑶ 成员函数 continue()是判断仿真是否继续进行的布尔函数，我们把它的返回类型说明为 bool 类型，如果把返回类型改为 int 型也是可以的，只需把返回语句改为：

```
return timeRemaining>0;
```

即可，二者效果相同，但采用 bool 类型能增加可读性。

⑷ SetTime()函数与重载的构造函数：

```
simulation::simulation (long secs){
  timeSimulate = secs;
  timeRemaining = timeSimulate;
```

```
}
```

的功能基本相同，在对象说明时可以直接赋初值：

```
simulation simone,simtwo(1800);
```

这里，对象 simone 的创建调用无参构造函数进行初始化，设置仿真时间为 3600 秒，而另一个对象 simtwo 的创建，则因有实参 1800，它会自动调用有参构造函数，从而设置仿真时间为 1800 秒。而函数 SetTime()可以在运行中任何时间调用。

⑸ perform()函数是系统运行函数，它一般是十分复杂的，不过在这里仅做两件事：显示，并要求用户按空格键；等待，每按一次空格键减少仿真时间 600。在任何实际运行的系统运行中如我们的电梯运行系统，它将完成许多工作。不过这将是自顶向下设计的下一步任务了。

⑹ 后面的 display()和 results()都是简化了的虚拟操作，在具体的仿真问题中，它们都应有更多的操作，在我们的实例中，限于篇幅，不可能面面俱到，重点是仿真运行过程的实现，而对于过程显示和结果输出就只能从简了。

11.2.2 电梯运行系统 building

为了把初步的抽象仿真程序具体化，我们把整个办公大楼视为一个电梯运行的系统，它应该是类 simulation 的派生类，把它称为类 building，这个类与其基类的主要区别是：

1）其数据成员不是简单的两个时间值：timeSimulate 和 timeRemaining，而是三个对象：Persons, Floors, Elevators.它们分别是人员集合，楼层集合，电梯集合，这是整个仿真系统的动作实体；

2) 其函数成员，除了继承基类的函数成员之外，列出的函数成员虽然名称与基类的函数相同，但执行的任务的内容是不同的。

类 building 的定义也分为两部分：

```
1  //building.h
2  #ifndef  _BUILDING_H
3  #define  _BUILDING_H
4
5  #include "simulation.h"
6  #include "person.h"
7  #include "floor.h"
8  #include "elevator.h"
```

```
9
10  class  building : public simulation{
11     persSet   Persons;
12     floorSet   Floors;
13     elevSet   Elevators;
14  public :
15     bool   continues (void);
16     void   perform (void);
17     void   display (void);
18  }
19
20  #endif     //_BUILDING_H
```

程序说明:

(1) 类 building 是 simulation 的派生类，是它的具体化。当然，基类的数据成员和函数成员也是 building 类的成员，因此是基类的扩展。

(2) 最明显的扩展是抽象的仿真类中只有两个整数作为它的数据成员。而 building 类中增加了三个数据成员：参与到系统中的人的集合，大楼中的若干电梯的集合及大楼的各个楼层组成的集合。仿真过程中他们的状态不断地在变化。时间、人群、电梯、楼层的各种状态值组成了整个大楼类的数据成员。

(3) 由上节的说明可知。类 simulation 的主要操作：display ()，perform () 和 results () 是虚拟化的，当然应该具体化，不过由于 results () 与其他方面涉及较少，这里就不再求精了。而原来的 continues () 基本功能是正常的，但还要作一点小小的改进，所以在类 building 中重新说明了这三个函数成员，以取代基类中的简单定义。

它们的定义在 building.cpp 中给出:

```
1   //building.cpp
2   #include "building.h"
3   #include <conio.h>
4   #include <……>        //  与函数 display ()的需求有关
5
6   bool   building::continues (void){
7     if (kbhit ())
8        if (getch()==ESCKEY)
```

```
9            SetTime (0);
10    return simulation::continues();
11  }// 增加了按 ESC 键可以终止仿真过程的功能
12
13  void   building::perform (void){
14     Persons.action ();
15     Elevators.action (Floors, Persons);
16     reduceTime (1);
17     totalTime++;
18  }// 整个大楼的一次动作，即每个人和每个电梯的一次动作
19
20  void building::display (void){
21     Floors.showFloor (Persons);
22     Elevators.showElevators ();
23     avgWait = Floors.avgWaiting ();
24     avgRide = Elevators.avgRiding ();
25     dis (……)；
26  }   // 显示功能被简化
27  // 程序中的行号，不是程序的一部分，是为了说明的方便，下同。
```

程序说明：

(1) 第 6～11 行定义了新的 continue（）函数，仅仅是增加了一项终端仿真过程的手段，在仿真程序运行中，按一下 Esc 键，就会引起强迫剩余时间量为 0（SetTime（0）），从而终止仿真过程。

(2) 第 13～18 行是新的 perform（）函数的定义，这里有较大的变动：主要的动作发生在 14～15 行；系统的每个人应有一次选择和动作；每个电梯有一次动作，该动作将与人群和各楼层的状态有关。第 16 行减少剩余时间 1 秒，即每一步动作占用 1 秒钟时间。这和（尚未说明的）状态变量 totalTime 加 1 是同一含义。虽然这个函数不长，但读者不难发现，第 14～15 行的两次函数调用可能包含比较多的人或电梯的动作。同时也涉及楼层状态的变化，可以说这将是下一步编程的主要任务。

第 20～32 行定义了新的 display（），从 11.1 节中的抽象主程序可以看出，主循环每隔一秒循环一次，执行一次 perform（），然后就通过执行 display（）来修改屏幕上的显示内容。

11.2.3 电梯仿真程序的总体框架

完成了类 simulation 及其派生类 building 的设计，并没有结束总体框架的设计，还必须设定一系列在整个电梯仿真过程中的常量、变量及随机函数值，其中，常量是指为系统运行设定的值，例如，大楼的楼层数 MAXFLOORS，电梯数 MAXELEVS，电梯的容量 CAPACITY 等等。全局变量是指系统运行中要统计或记录的量，例如，当前搂内总人数 inBuilding，离开人数 leftBuilding 等等，这些变量的设定主要用于系统状态显示和统计结果,设计者可以根据需要增减。随机函数的设定是仿真系统运行所需要的，例如，一个在楼外的人要根据随机函数 wantsToEnter()来决定是否进楼，他在等待电梯时根据随机函数 EnterDest()来决定其乘电梯的目标层数等等,这些随机函数值的生成细节在代码中忽略。为了说明这些全局量，可以定义成一个更大的类 elvesim，但这里仅仅把它们以全局说明的形式组成一个头文件。

```
1  //elevsim.h
2  #ifndef _ELEVSIM_H
3  #define _ELEVSIM_H
4  #include <stdlib.h>
5  #include "simulation.h"
6  enum direction{DOWN = -1, NODIRECTION = 0, UP = 1};
7                  //direction 的值表示电梯的运行状态
8  const int ESCKEY = 27;
9                  //常量 ESCKEY = 27 是<Esc>键的 ASCII 码
10 const int MAXELEVS = 10;                    //电梯数
11 const int MAXFLOORS = 10;                   //楼层数
12 const int MAXPERSONS = 500;                 //最多人数
13 const int ELEVWAIT = 15;              //在一层电梯停下的等待时间(秒)
14 const int CAPACITY = 24;                    //电梯容量
15 const int TRAVELTIME = 5;                   //层间运行时间
16 static bool  wantsToEnter (void){
17   return rand ()<200;
18 };            //决定是否进入大楼
19 static int  EnterDest (void){
20   return 1+ (rand() % (MAXFLOORS-1));
21 };            //决定转到第几层，目标楼层选择
```

```
22  static int MaxWait(void){
23    return 180+ (rand() % 180);
24  };              //决定最大等待时间，等待超时改走楼梯
25  static int Business(void){
26    return 400 + (rand() % 180);
27  };              //决定到达目标层后的连续工作时间
28  static bool   leaving (void){
29    return rand() < 22000 == 0 ? false: true;
30  };              //决定该人是否离开大楼
31  static int totalPeople;                    //总人数
32  static  int  inBuilding;                   //正在楼内人数
33  static  int  leftBuilding;                 //离开人数
34  static  int  avgWait;                      //平均等待人数
35  static  int  avgRide;                      //平均乘客
36  static  int  tookStair;                    //步行楼梯人数
37  static  int  totalTime;                    //仿真运行时间
38
39  #endif //_ELEVSIM_H
```

电梯仿真系统的主程序与前述有所不同：

```
// elevsim.cpp
 1   #include<stdlib.h>
 2   #include<time.h>
 3  #include  "elevsim.h"
 4   #include  "building.h"
 5  #include  <...>
 6   //与显示相关的标准函数库
 7
 8   void initdisplay (void) ;
 9
10   building  buildsim;
11
12  void main ( ) {
13     srand( (unsigned)time( NULL ) );        //使每次产生的随机数都不同
```

```
 14    initdisplay( );
 15    buildsim.display( ) ;
 16   while(buildsim. continues ( ) ) {
 17       buildsim.perform( ) ;
 18       buildsim.display( ) ;
 19    }
 20   buildsim.results ( ) ;
 21  }
 22
 23  void initdisplay(void) {
 24      // ……
 25    //设置屏幕显示的初始状态和数值，
 26    //其主要部分为：
 27    //屏幕底行：显示主要统计项目的名称和
 28    //数值，右下角为仿真时钟
 29    //屏幕左边列：显示楼层状态，自下向上
 30    //分列 0 层，1 层…，9 层，对每层列出
 31    //层号，等待人数，按钮状态(一，U，D)
 32    //屏幕顶行和中央区：显示各电梯状态：
 33    //顶行自左至右列出 0 一 9 号电梯的当
 34    //前所在楼层号，而在第 i 号电梯所
 35    //在的列，和 i 号电梯所在的楼层对
 36    //应的行处，列出该电梯的乘客人
 37    //和运行状态(U，D，N)
 38    //显示的状况可见下面的示意图(图 11．1)，显
 39    //示的内容和方法，完全可以由设计者
40  //创造出不同的布局
41  }
```

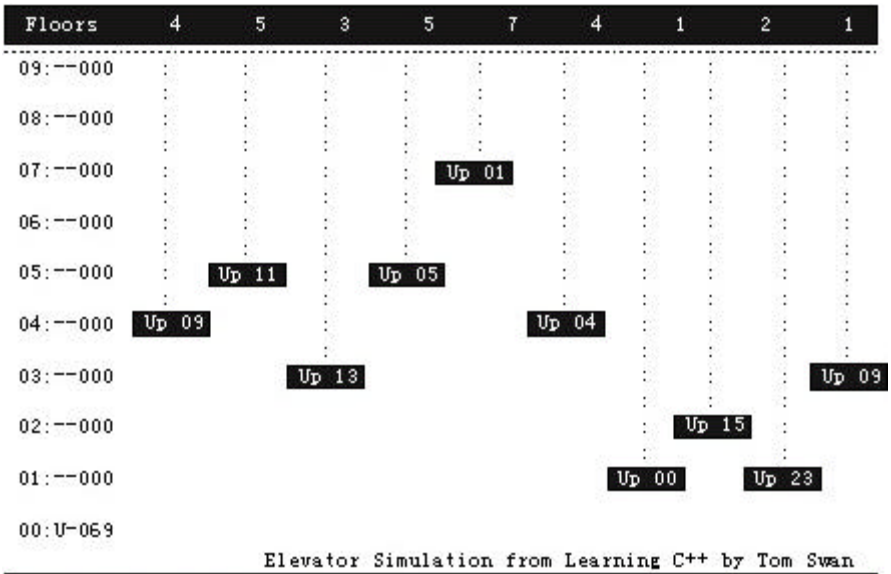


图 11.1 屏幕显示的布局设计

图 11.1 是一个屏幕显示的布局设计方案，图的中间部分是移动着的电梯，电梯中的符号表示上行或下行，数字显示电梯中的人数，图的下部给出系统运行的若干状态值，从定义来看，要变动的显示内容包括：

与楼层状态有关的信息显示；
与电梯有关的信息显示；
重新计算各层的平均等待人数；
重新计算各电梯的平均乘客数；
重新显示若干状态变量值（在最下行）；
totalPeople：总人数；
inBuilding：大楼内人数；
leftBuilding：离开大楼人数；
avgWait：各层平均等待人数；
avgRide：各电梯平均乘客数；

tookStair：步行走楼梯的人数；
MAXELEVS：电梯数；
MAXFLOORS：楼层数；

在屏幕的右下角重新显示从仿真开始时计数的时间，用*小时*分*秒的形式显示。

显示的任务虽然列了许多，但实际编程却比较简单，因此程序中忽略这些细节,读者可以根据需要和个人的兴趣采用不同的设计方案。

现在让我们来讨论一下整个设计过程，从中分析一下各模块之间的关系。全局仿真程序 elevsim 的主要核心是类 building 的设计，类 building 的设计有四个相关类：它的基类 simulation；它的成员类：persSet，floorSet 和 elevSet。而后三个类又有它们的成员类，分别是：Person，floor，elevator。所以各程序模块的关系可以如图 11．2 所示。

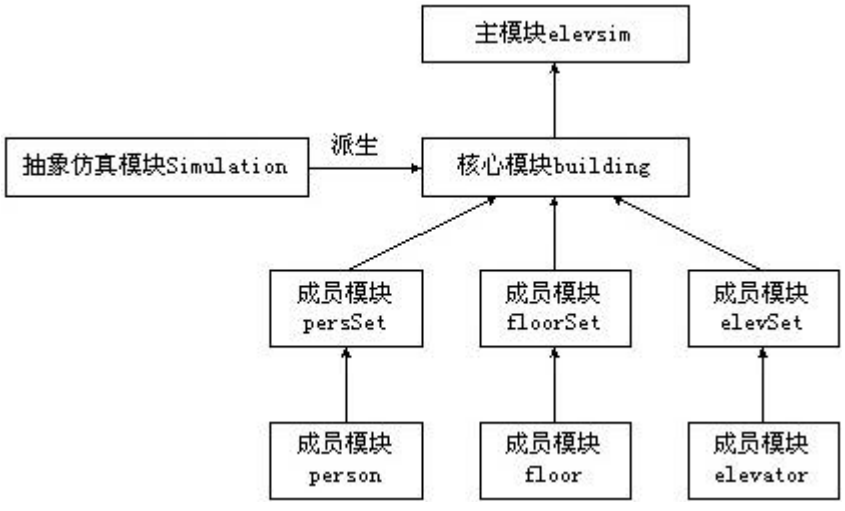


图 11.2 各程序模块的关系图

各程序模块之间既有联系，又相对封闭，形成各自的独立结构，这样的程序可读性较好，易于编写，易于修改，易于调试。

在这个总体框架下，下一步开始程序设计的主要任务，设计大楼中的活动实体：人，电梯和大楼的各个楼层在仿真过程中的动作。“自顶向下”地设计人员类 person 与人员集合类 persSet；楼层类 floor 和楼层集合类 floorSet；电梯类 elevator 和电梯集合类 elevSet。

11.2.4 人员类 person 与人员集合类 persSet

从上节的程序中已经可以看到 person 类和 persSet 类的设计是很重要的，它规定了系统中的动作实体之一 person（人）的行为，由于作为人员集合的 persSet 类是人员类 person 的对象数组，其成员函数较为简单，所以，两个类的说明包括在同一文件之中。

下面是 person 类和 persSet 类的定义：

```
1   //person.h
2   #ifndef _PERSON_H
3   #define _PERSON_H
4
5   #include "elevsim.h"
6
7   class   person{
8         int   floorNowOn;                  //所在楼层（在楼外为-1）
9         int   desfination;                 //目标楼层
10        int   maxWaitTime;                 //最大的容忍等待时间
11        bool   waitingForElev;             //是否正在等待
12        bool   takingStairs;               //是否走楼梯
13        int   elevNowIn;                   //所在电梯号（不在电梯内为-1）
14    public:
15        person ();
16        void   action (void);                          //人的一次动作
17        bool   upwaiting (int floorNumber);           //是否等上行电梯
18        bool   dnwaiting (int floorNumber);           //是否等下行电梯
19        bool   loadIfWaiting (int elevNumber,
20                              int floorNumber,
21                              int &pdest);            //该层是否有电梯搭乘
22        bool   loadIfGoing (int elevNumber,
23                            int floorNumber,
24                            direction dir,
25                            int&pdest);
    //是否在该楼层有同一方向运行的电梯可以搭乘
26        bool   discharge (int elevNumber,
```

```
27                          int floorNumber);     // 是否到达目标层，走出电梯
28  };
29
30  class   persSet{
31      person   pa[MAXPERSONS];
32    public:
33      void   action  (void);                              //每个人动作一次
34      void   numWaiting (int floorNumber,
35                          int &nup,int &ndn); //统计该层等待上行下行人数
36      bool   loadAny (int elevNumber,
37                          int floorNumber,
38                          int &pdest);       //是否有人在该层搭乘该电梯
39      bool    loadOne (int elevNumber,
40                          int floorNumber,
41                          direction dir,
42                          int &pdest);      //是否有人在该层搭乘该方向电梯
43      int discharge(int elevNumber,
44                          int floorNumber);   //该电梯在该层走出的人数
45  };
46
47  #endif //_PERSON_H
```

程序说明：

（1）类 person 是在大楼中的电梯运行系统中的人，包括一个人与电梯运行有关的数据和操作，另一个类 persSet 实际上是 person 的对象数组，不过它定义了几个特殊的操作。

（2）类 person 的数据成员为：

floorNowOn：当前所在楼层，在楼外为-1

destination：要到的目标层

maxWaitTime：当前还能再等待的时间

waitingForElev：是否在等电梯

takingStairs：是否在走楼梯

elevNowIn：所在电梯号，不在电梯中为-1

（3）共七个公有函数成员，其中 action()是最基本的也最复杂。它们的功能可从下面的定义中了解。

```
1   //person.cpp
2   #include "person.h"
3   #include <stdlib.h>
4
5   person::person (){
6     floorNowOn = -1;                    //不在楼内
7     waitingForElev = false;             //未等电梯
8     destination = -1;                   //无目标层
9     maxWaitTime = 0;                    //不等待
10    takingStairs = false;               //未步行楼梯
11    elevNowIn = -1;                     //不在电梯中
12  }// 一个人员对象的初始设置
13
14  void   person::action (void){
15                                        //如在楼外，决定是否进大楼
16    if (floorNowOn<0){                  //不在楼内
17      if (wantsToEnter()){              //决定进入
18        destination = EnterDest();      //决定目标层
19        floorNowOn = 0;                 //当前楼层：在底层
20         maxWaitTime = MaxWait();       //设置最大等待时间
21         waitingForElev = true;         //设置为等待电梯状态
22         takingStairs = false;          //不走楼梯
23         elevNowIn = -1;                //不在电梯中
24         totalPeople++;                 //总人数+1
25         inBuilding++;                  //楼内人数+1
26       }
27    }
28    else{                               //已在楼中
29      if (waitingForElev){              //正在等待
30        if ((maxWaitTime--) <= 0){      //等待时间超限
31          waitingForElev = false;       //停止等待
32          takingStairs = true;          //改走楼梯
33          maxWaitTime = 30*abs(destination-floorNowOn);
34                          //每走一层用 30 秒，作为下次等待上限
```

```
35              tookStair++;                    //步行者人数计数
36            }
37          }
38          else                                //未等待
39            if (takingStairs)  {              //正在走楼梯
40              if ((maxWaitTime--) <= 0){      //已到达目标层
41               floorNowOn = destination;      //在目标层
42               takingStairs = false;          //停止走楼梯
43               maxWaitTime = Business();      //设置新的工作时间
44              }
45            }
46            else                              //也未走楼梯
47              if (elevNowIn<0)                //未在电梯中，值为－1
48                if ((maxWaitTime--) <= 0){    //等待时间超限
49                  if (leaving())              //决定离开
50                    destination = 0;          //设置目标层为最底层
51                  else {
52                    destination = EnterDest();
53                                              //选某一层为目标层
54                    if (destination == floorNowOn)
55                      destination = 0;
56                  }                           //不允许到同一层
57                  maxWaitTime = MaxWait();
58                                              //设置最大等待时间
59                  waitingForElev = true;      //设置为等待状态
60                } //  开始等待电梯到新的目标层
61              if ((floorNowOn == 0)&&(destination == 0)){
62                floorNowOn = -1;                  //送出楼外
63                 leftBuilding++;                  //离去人数+1
64                 inBuilding--;                    //楼内人数-1
65              }
66      }
67  }/* action() */
68
```

```
69  bool  person::upwaiting (int floorNumber){
70    return waitingForElev &&
71               (floorNowOn==floorNumber) &&
72               (destination>floorNowOn);
73  }                                    //判定是否在该层等向上的电梯
74
75  bool  person::dnwaiting (int floorNumber){
76    return waitingForElev &&
77               (floorNowOn==floorNumber) &&
78               (destination<floorNowOn);
79  }                                    //判定是否在该层等向下的电梯
80
80  bool  person::loadIfWaiting (int elevNumber,
81                                  int floorNumber,
82                                  int &pdest){
83    if (waitingForElev &&
84       (floorNowOn ==floorNumber)){        //等到电梯
85         waitingForElev = false;           //进入电梯
86         elevNowIn = elevNumber;           //电梯号
87         pdest = destination;              //取目标层
88         return true;
89       }
90       return false;
91  }                                    //判定在该层是否进入该电梯，
92
94  bool  person::loadIfGoing (int elevNumber,
95                                  int floorNumber,
96                                  direction dir,
97                                  int &pdest){
98    direction  pdir;
99    if (destination > floorNumber)
100     pdir = UP;
101   else
102     pdir = DOWN;
```

```
103   if (waitingForElev &&
104       (floorNowOn == floorNumber) &&
105      (dir == pdir)){
106         waitingForElev = false;                    //停止等待
107         elevNowIn = elevNumber;                    //进入该电梯
108         pdest = destination;                       //取目标层
109         return true;
110    }
111   return false;
112 }                                                  //判定是否进入该上(下)行电梯
113
114 bool   person::discharge (int elevNumber,
115                           int floorNumber){
116   if ((elevNowIn == elevNumber)&&
117      (destination == floorNumber)){
118         elevNowIn = -1;                            //走出电梯
119         floorNowOn = destination;                  //到达目标
120         if (floorNowOn != 0)
121            maxWaitTime = Business();               //开始工作
122         return true;
123       }
124       else
125         return false;
126 }                                  //判断乘客是否到达目标层而走出电梯
127
128 /* 以上为类 person 的实现，下面介绍类 persSet 的函数定义 */
129
130 void   persSet::numWaiting (int floorNumber,
131                             int &nup,int &ndn){
132   nup = ndn = 0;
133   for (int i = 0; i<MAXPERSONS; i++){
134      nup += pa[i].upwaiting (floorNumber);
135      ndn += pa[i].dnwaiting (floorNumber);
136    }
```

```
137 }                                                    //统计在等电梯的人中要向
138                                                      //上和向下行的人数
139
140 bool   persSet::loadAny (int elevNumber,
141                           int  floorNumber,
142                           int  &pdest){
143   for (int i=0; i<MAXPERSONS; i++)
144       if (pa[i].loadIfWaiting (elevNumber,
145                             floorNumber,pdest))
146          return true;                               //进了电梯
147    return false;                                     //无人进电梯
148 }     //该电梯在这一层是否接收了乘客
149
150 bool   persSet::loadOne (int elevNumber,
151                          int floorNumber,
152                          direction dir,
153                          int &pdest){
154   for (int i=0; i<MAXPERSONS; i++)
155      if (pa[i].loadIfGoing (elevNumber,
156                             floorNumber,dir,pdest))
157          return true;                               //进了电梯
158    return false;                                    //无人进去
159 }     //该上行或下行的电梯在这层是否接收了乘客
160
161 int persSet::discharge (int elevNumber,
162                         int floorNumber){
163   int n = 0;
164   for (int i=0; i<MAXPERSONS; i++)
165      n += pa[i].discharge (elevNumber, floorNumber);
166   return n;
167 }     //统计走出电梯的人数
168
169 void persSet::action(){
170 for(int i=0;i<MAXPERSONS;i++ )
```

```
171  pa[i].action();
172 }       //每个人各动作一次
```

程序说明：

(1)本节所提供的程序已经深入到仿真过程的细节。由于把仿真过程的细节划分成若干具体的处理，因此，尽管整个系统错综复杂，头绪繁多，但涉及到一个操作，它可能就是简单的了。即使在有关人的类 person 中，除了 action()函数之外，各函数成员都是很简单的。action()程序略长，建议读者一定要看明白。

(2)首先宏观地看一下它们的功能：

person 类

①构造函数：5~12 行，为 6 个数据成员赋值，这 6 个初始值的设定，都是根据整个系统的一个假定决定的：即所有的人在一开始时是在办公大楼之外的。

②action()：14~67 行，这是一个最长，也最重要的函数。它的功能是根据该人当前所处的状态来决定他做一个什么动作：

```
A．现在楼外：
决定是否进楼➔进楼后开始等电梯(底层没有办公室)，状态修正。
决定不进➔仍在楼外，状态不变。
B．现在楼内：
正在等电梯：
    等待超时➔转为走楼梯，状态修正。
    等待未超时➔继续等，时间-1。
未等电梯：
    正在走楼梯：
        已到达目标层➔转为工作，状态修正。
        未到达目标层➔继续走，时间-1。
    正在工作中：
        工作完成：
            决定离开➔到底层
            决定不离开➔到另一层，开始等电梯，状态修正。
        工作未完成➔继续工作，时间-1。
    正在电梯中➔另做处理。
    来到底层➔离开大楼，状态修正。
```

函数 action()就是完成了上面的由判定决定动作的逻辑的。细心的读者可

能会发现一个问题：person 的动作（action）中为什么不包括进入电梯，乘电梯和走出电梯？实际上这部分任务是在电梯（elevator）的动作（action）中完成的，那么，在这期间，person 的状态值是怎样处理的呢？

(3)这里要说明的几处随机判断：

①第 17 行，随机函数 WantstoEnter（），根据随机函数 rand（）产生的整数是否<200 来决定是否决定进入。如此，大约每一秒钟有千分之六的人要进入大楼。当然这个常数 200 是可以调整的，增大则会增加楼内的人数。

②第 20，57 行，随机函数 MaxWait（），为进楼后等待电梯选择一个最大等待时间，这时间值在 180~360 秒之间（即 3~6 分钟）之间随机选择。

③第 18,52 行,随机函数 EnterDest(),为进楼的人选定目标楼层,它在 1~9 层之中随机选择，按系统要求，底层即第 0 层不是工作场所，只是进出通道。

④第 43 行，随机函数 leaving()，为到达目标楼层后要开始工作的人确定工作时间，其值在 400~6600 秒之间随机选择。

⑤第 49 行，随机函数 leaving()，为完成一段工作的人决定是离开大楼，还是选择到另一层去办另一件事。其值由 rand()<22000 来决定，大约有 2/3 的完成一段工作的人决定离开大楼。

(4)函数成员 upwaiting()和 dnwaiting()用来判断决定某人在某楼层应等电梯向上还是向下。

(5)函数成员 loadIfWaiting()和 loadIfGoing()用来判断决定某人在某楼层可否进入电梯。

(6)discharge()是 person 的最后一个函数成员,利用它可以判定某人是否应在某层从某电梯里出来。

下面是几个 persSet 类的函数成员：

(7)类 persSet 的 action()函数，就是要该集合中所有的人动作一次。

(8)函数成员 numWaiting()用来统计等电梯的人中，要向上和要向下乘电梯的人数。

(9)函数成员 loadAny()和 loadOne()分别检查某电梯在某层是否有人被接受进入，和某上行或下行电梯在某层是否有人被接受进入。

(10)函数成员 discharge()用来统计某电梯在某层走出电梯的人数。

当把人在整个电梯运行系统的规律搞清之后，类似的把楼层（floor）和楼层集合（floorSet），电梯（elevator）和电梯集合（elevSet）以类的形式设计出来也就是不难的事情了。

11.2.5 楼层类 floor 与楼层集合类 floorSet

楼层是大楼的组成部分，对于我们的具体问题来说，一个楼层有多少房间，有什么照明设备等等，是不相关的事情。对于办公楼的电梯系统来说，一个楼层就意味着有电梯出入口，可能有人等待，有的要求向上，有的要求向下。

下面是类 floor 的说明：

```
//floor.h
#ifndef _FLOOR_H
#define _FLOOR_H

#include "person.h"

class floor{
   int floorNumber;            //楼层号 0~9
   Bool up,down;               //按钮
   int np;                     //等电梯人数
public:
    floor();
   void SetFloorNumber(int n){
       floorNumber=n;};
   Bool downButton(void){return down;};
   Bool upButton(void){return up;};
   void reSetUpButton(void){up=false;};
   void reSetDownButton(void){down=false;};
   int getNumWaiting(void){return np;};
   void showFloor(persSet &persons);
   void SetUpButton(void);
   void SetDownButton(void);
};

class floorSet{
   floor fa[MAXFLOORS];
public:
    floorSet();
   void showFloors(persSet &persons);
```

```
  void reSetButton(direction dir,int floorNumber);
  Bool signalUp(int floorNumber);
  Bool signalDown(int floorNumber);
  Bool signalSameDir(direction dir,int floorNumber);
   int avgWaiting(void);
};

#endif //_FLOOR_H
```

程序说明：

（1）类 floor 的一个对象，有四项数据，即楼层号，等电梯人数，两个（向上和向下）按钮。由此可知，这是一个简化了的“楼层”，它只有两个按钮，可以同时通知所有的电梯。等待电梯的人也不区分它们在等待哪部电梯。

（2）类 floor 的操作大都比较简单，十个函数中有六个因过于简单而在类内以内联函数方式定义。

（3）类 floorSet 实际上是 floor 的一维数组，仅仅为了赋予一定的操作才以类的形式出现，这样显得更有层次，更清晰。

下面是 floor 类和 floorSet 类的实现：

```
1   //floor.cpp
2   #include "floor.h"
3   #include <...>
4
5   floor::floor(){
6      up=down=false; np=0;
7   }
8                                          //设置初始状态
9   void floor::showFloor(persSet &persons){
10     char uc='-',dc='-';
11     int nup,ndn;
12     persons.numWaiting(floorNumber,nup,ndn);
13                          //统计在 floorNumber 层的(向上向下)等待人数
14     np=nup+ndn;
15     if(nup>0) SetUpButton();
16     if(ndn>0) SetDownButton();
```

```
17    if(up) uc='U';
18    if(down) dc='D';
19     disp(...);
20                                        //把楼层号 FLOORnUMBER 和 uc,
    dc,np
21                                        //显示在屏幕左侧的适当位置，这里
22                                        // 的“适当”位置是指：要使各楼
23                                        //层的数据显示从下到上按层号列出
24  }
25  void floor::SetUpButton(void){
26     if(floorNumber<MAXFLOORS-1)
27        up=true;
28  }                                       //最顶层不可置 up 为 true
29
30  void floor::SetDownButton(void){
31     if(floorNumber>0)
32        down=TURE;
33  }                                       //最底层不可置 down 为 true
34  /* 下面是类 floorSet 的函数成员定义 */
35
36  floorSet::floorSet(){
37     for(int i=0;i<MAXFLOORS;i++)
38        fa[i].SetFloorNumber(i);
39  }                                       //每层的楼层号是一起赋值的
40
41  void floorSet::showFloors(persSet &persons){
42     for(int i=0;i<MAXFLOORS;i++)
43        fa[i].showFloor(persons);
44  }                                       //在屏幕上的左侧显示每一楼层
45                                          //的数据——它们从下到上列出
46
47  void floorSet::reSetButton(direction dir,
48                             int floorNumber){
49     if(dir==UP)
```

```
50          fa[floorNumber].reSetUpButton();
51       else
52          if(dir==DOWN)
53             fa[floorNumber].reSetDownButton();
54    }                                          //对指定楼层按一个按钮
55
56    Bool floorSet::signalUp(int floorNumber){
57       for(int i=MAXFLOORS-1;i>floorNumber;i--)
58          if(fa[i].upButton() || fa[i].downButton())
59             return true;
60        return false;
61    }                                          //查以上各层是否有等待信号
62
63    Bool floorSet::signalDown(int floorNumber){
64       for(int i=0;i>floorNumber;i++)
65          if(fa[i].upButton() || fa[i].downButton())
66             return true;
67        return false;
68    }                                          //查以下各层是否有等待信号
69
70    Bool floorSet::signalSameDir(direction dir,int floorNumber)
71    {
72       if(dir==UP)
73          return fa[floorNumber].upButton();
74       else
75          if(dir==DOWN)
76             return fa[floorNumber].downButton();
77          else
78             return false;
79    }                                          //查某层是否有 dir 方向的等待
80
81    int floorSet::avgWaiting(void){
82       int total=0;
83       for(int i=0;i<MAXFLOORS;i++)
```

```
84          total +=fa[i].getNumWaiting();
85      return(total/ MAXFLOORS);
86  }                                          //计算各层平均等待人数
87
88                                             //完成了 floorSet 类的定义
```

程序说明：

(1)第 5~7 行为给 floor 类的数据成员 up，down，np 赋初值 false 和 0。

那么 floor 类的另一数据成员 floorNumber 的初值是如何获得的呢，请看第 36~39 行，这里是类 floorSet 的构造函数，这里的关系可以描述为：

在整个仿真程序中出现在办公大楼 Building 系统中的是类 floorSet 的对象 floors。floors 实际上是类 floor 的对象数组，大楼共有十层，数组就由 10 个 floor 对象组成。

在说明 floorSet 类的对象 floors 时，应自动进行有关的初始化：

①如果 floorSet 有基类的话,先自动调用基类的构造函数。这里没有基类，故跳过此步。

②如果 floorSet 有成员为类的对象,则先自动调用成员类的构造函数为其成员初始化，现 floorSet 有 floor 类的 MAXFLOORS 个对象组成的数组作成员，故须对每一 floor 类的对象 fa[i]调用构造函数 floor()，从而为每个 fa[i]中的三个数据成员：up，down，np 赋了初值。

③最后自动调用 floorSet 类自己的构造函数，如 36~39 行的程序所示，它为每一个 fa[i]的数据成员 floorNumber 赋初值 i。

因此，当程序中出现说明语句：floorSet floors；时，这个简单的语句除了为 floors 分配内存之外还自动隐含地执行上述的一系列初始化工作。

(2)我们再看第 9~24 行的函数 showFloor()和第 41~45 行的 showFloors()。后者实际上是对于楼层集合 floors 的 10 个元素 fa[o]，fa[1]，…，fa[9]分别调用类 floor 的函数 showFloor()，换句话说，后者由十次执行前者组成。

显然由于 floors——类 floorSets 的一个实例，是整个电梯运行系统——Building 的一个组成部分,所以 floorset::showFloors()就必然是总仿真程序的主要操作之一：display()的一个组成部分，每运行一秒钟，都要把各楼层的状态显示一次，或说把显示修正一次。

至于 floor::showFloor()的具体任务，从 9~23 行可以看出，它主要是把该楼层的层号，两个按钮的状态，等待人数，显示在屏幕的左侧。由于 floor::showFloor()的执行，则把十个楼层的上述数据从下向上地列在屏幕的左侧。

由于显示的问题涉及具体的库函数操作，各种不同版本的 C++语言的库函数可能有差别，因此在本程序中采用功能描述的方法。有关显示的具体实现可以互不相同，无论在字符模式下实现还是在图形模式下实现都是可以的。

(3)类 floor 中,除了为创建对象进行初始化的函数和显示其内容的函数之外，用来查看其内容的函数 downButton()，upButton()，getNumWaiting()时必要的；用来设置数据值的函数 setFloorNumber()，reSetUpButton()，reSetDownButton()，setUpButton()，setDownButton()当然也是十分必要的。

还有一个必要的函数成员：setNumWaiting(int)，在这里是有意省略了，事实上它是十分必要的，没有它，楼层的等待人数值 np 就无法修改和更新，永远保持初始化时的 0。其定义为

```
void floor::setNumWaiting(int n){
  np=n;
}
```

它的作用应加入到 person::action()函数中。例如,当该人一段工作结束，就需调用该所在楼层的 getNumWaiting()；把得到的 np 值加 1 后通过 setNumWaiting(getNumWaiting()+1)；使该层等待人数加 1。类似地当某人在某层等待超时改走楼梯，或进入到一个电梯中时则应执行：

```
setNumWaiting(getNumWaiting()-1);
```

(4)为类 floorSet 设计的操作，大多具有综合统计性质：

reSetButton()用来关闭指定层的 Up 或 Down 按钮,当某电梯在某层装进所有等待（上或下）的人员时，应执行此操作。

signalUp(),signalDown(),signalSameDir()是用来检查某层或某些层的 Up 和 Down 按钮的。

avgWaiting()用来统计各层的平均等待人数，它也是仿真过程中要显示的一个内容，从这里也可以看到 SetNumWaiting(int)；的必要性。

11.2.6 电梯类 elevator 与电梯集合类 elevSet

现在我们考察整个办公大楼作为一个仿真运行系统的第三个实体类 :电梯。相对于人员和楼层来说，电梯可以说是三者中的主角，因为在人员和楼层的定义设计中,数据和操作的设置却是以是否与电梯运行有关或无关而决定取舍的。同样，先设置类 elevator 与类 elevSet 的成员，然后再完成它们的定义实现：

```
1  //elevator.h
2  #ifndef _ELEVATOR_H
```

```
3  #define _ELEVATOR_H
5  #include "elevsim.h"
6  #include "floor.h"
7  #include "person.h"
8
9  class elevator{
10     int elevNumber;                    //本电梯号
11     int timeToAction;                  //再启动前时间
12     int floorNumber;                   //当前楼层号
13     bool stopped;                      //是否正在停止
14     direction dir;
15     bool buttons[MAXFLOORS];           //梯内按钮
16     int passengers;                    //乘员数
17     bool buttonUp(void);               //查有无上行
18     bool buttonDown(void);             //查有无下行
19  public:
20     elevator();
21     int getPassengers(void){return passengers;}
22     void SetelevNumber(int n) {elevNumber=n;}
23     void showElevator(void);
24     void SetDirection(floorSet & floors);
25     bool elevStopping(floorSet & floors);
26     void action(floorSet & floors, persSet & persons);
27 };
28
29 class elevSet {
30    elevator ea[MAXFLOORS];
31 public:
32    elevSet( ) ;
33    void showElevators (void);
34    void action(floorSet & floors ,
35    persSet & persons) ;
36    int avgRiding(void) ;
37 };
```

```
38
39 # endif // _ELEVATOR _H
```

程序说明：

(1)第 5—7 行指出，电梯的运行与使用电梯的人员——电梯的乘客有关，与各楼层的状态(它的等待人数，它的按钮)有关。

(2)第 10 一 16 行是电梯的状态数据：

电梯号：0 — MAXELEVS-1。

停止时间：直到再启动前的停止时间，当 timeToAction 的值为 0 时应开始启动(上或下)。

当前所在楼层号：0 一 MAXFLOORS-l。

是否处于停止状态：stopped 是一布尔量。

运行状态：UP，DOWN 或 NODIRECTION。

梯内按钮：布尔数组，下标 0 .. MAXFLOORS-1，指示在某层是否有人要到达目标层。

乘客人数：passengers＜＝CAPACITY。

(3)第 17—18 行是两个私有函数成员：用来检查电梯内是否有人需要继续上行或下行。

(4)类 elevator 设有七个公有函数成员，其中设有一个 showElevator()函数，这是因为它与楼层一样，也需要在仿真过程中随时显示各个电梯的当前状态。而人员 person 类则无类似函数，这是因为不可能对办公大楼中的所有人员状态在屏幕上予以显示。

另一引人注目的函数成员是 action()，这是因为电梯和人员一样要有“动作”，每隔一个时间间隔(例如 1 秒)，电梯应有一个新的动作。请注意，电梯和人员有动作 action()，而楼层没有动作。其它关于 elevator 的操作在定义它们时讨论。

(5)和 Person 类，floor 类一样，elevator 类也定义了相应的 elevSet 类，它是 MAXELEVS 个电梯的集合(数组)。它有自己的一些操作；

showElevators()：显示全体电梯状态。

action()：全体电梯动作一次。

avgRiding：全体电梯的当前平均载客人数。

下面是各函数成员的定义：

```
1  //elevator.cpp
2  #include "elevator.h"
3  # include <...>
```

```
4  //与显示有关的标准函数库
5
6  elevator::elevator( ) {
7      elevNumber  = -1;                    //未正式赋值
8      timeToAction = ELEVWAIT;             //暂停 15 秒
9      floorNumber  = 0;                    //在底层
10     stopped=true;                        //停止状态
11     dir  = NODIRECTION;                  //无运行方向
12     passengers  = 0;                     //电梯空
13     for(int i=0;i<=MAXFLOORS;i++)
14     buttons[i]= false ;                  //置梯内按钮
15 }
16
17 bool elevator::buttonUp(void) {
18    for (int i=floorNumber+1 ;i<MAXFLOORS ;i++)
19    if(buttons[i]) return true ;
20       return false ;
21 }                                         //查电梯内是否有人上行
22
23 bool elevator::buttonDown(void) {
24    for (int i=0;i<floorNumber;i++)
25    if (buttons[i]) return true;
26       return false;
27 }                                         //查电梯内是否有人下行
28
29 void elevator::showElevator(void) {
30 ……
31            //在屏幕的最上面一行列出 10 个电梯
32            //（自左至右)的当前层数
33            //然后在每个电梯的相应位置，显示该
34            //电梯的当前乘客人数和状态(上行或下
35            //行，或停止)
36            //所谓“相应位置”是指：例如 3#电梯目前
37            //处在第四层，则相应位置应在 4#楼层
```

```
38              //所在的行和 3#电梯所在的列
39 }
40
41 void elevator::SetDirection(floorSet & floors){
42    if(buttonUp( )) dir=UP;
43     else
44       if (buttonDown( )) dir=DOWN;
45        else
46          if (floors.signalUp (floorNumber))
47              dir=UP;
48           else
49              if(floors.signalDown (floorNumber))
50                dir = DOWN ;
51              else
52                dir=NODIRECTION ;
53 }                                          //确定电梯运行方向
54
55 bool elevator::elevStopping(floorSet & floors) {
56    if(buttons[floorNumber])
57      return true;                          //本层有人要下
58    else
59       if(floors.signalSameDir(dir,floorNumber))
60         return true;                       //本层有人等且与
61                                            //电梯同方向
62       else
63          if(floorNumber == 0)
64             return true;                   //已到底层
65           else
66              if(floorNumber ==   MAXFLOORS - 1)
67                 return true;               //已到顶层
68              else
69                if(dir == NODIRECTION )
70                   return true;             //电梯在无向状态
71                else
```

```
72                  return  false;                  //不停
73 }                                                //决定电梯是否应停止
74
75 void elevator::action(floorSet & floors ,persSet & persons)
76 {
77  int  pdest;
78
79  if(stopped){
80     passengers -= persons.discharge (elevNumber,
81                                          floorNumber);
82                                    //如果电梯停下,有若干人从该
83                                    //楼层走出电梯到达目的地。这是电梯
84                                    //停下后第一项要做的事
85    if(dir== 0 && passengers<CAPACITY) {
86                                    //电梯未定向且内有空位
87       if  (persons.loadAny(elevNumber,floorNumber,pdest)){
88                                    //该层有人等待上电梯
89          passengers++;             //乘客数加 1
90          timeToAction++ ;          //停时间加 1
91         buttons[pdest]  =  true;   //按目标层按钮
92          if  (pdest>floorNumber)
93            dir  =  UP;
94          else
95            dir  =  DOWN;
96                                    //按第一个乘客要求决定上下
97       }
98    }
99   if(dir != 0 && passengers < CAPACITY) {
100                                   //电梯有空位，但是已经决定上行或下行
101      if(persons.loadOne(elevNumber,floorNumber,
102          dir,pdest))  {
103                                   //该层有人等待且方向与电梯相同
104         passengers++;                 //乘客数加 1
105         timeToAction++;               //停时间加 1
```

```
106         buttons[pdest] = true;              //按目标层按钮
107         if (passengers>= CAPACITY)
108             timeToAction= 0;                //如果乘客满开始运行
109         }
110     }
111     if(timeToAction -- <= 0) {
112                         // 已经到启动时间
113       if(dir==0)   //NODIRECTION = 0
114           SetDirection (floors );
115       if(dir == 0 && floorNumber>0)
116           dir = DOWN;
117       if(dir == 0)
118           timeToAction = ELEVWAIT;              //停在底层
119       else{
120           floors.reSetButton (dir,floorNumber);
121                                                 //重新设置该层的 dir 方向按钮
122           stopped = false;                      //启动
123           timeToAction = TRAVELTIME;            //到下一层的运行时间
124       };
125     }
126     }                                 //上面代码处理电梯停止状态
127     else                              //下面代码是电梯运行状态
128     if (timeToAction -- <=0) {                  //运行到另一层
129       if(dir == UP)
130           floorNumber ++;                       //楼层号加 1
131       else
132         floorNumber -- ;                        //楼层号减 1
133     SetDirection (floors);                      //决定方向
134     if(elevStopping(floors) ){                  //电梯应该停止
135       floors.reSetButton (dir,floorNumber);
136                                                 //修正该层按钮
137       stopped=true ;                            //停
138       timeToAction=ELEVWAIT;                    //置停止时间
139       buttons[floorNumber] = false;             //修正电梯内按钮
```

```
140     }else
141       timeToAction=TRAVELTIME ;              //继续再行一层
142   }
143 }                       //电梯的一步动作
144
145 elevSet::elevSet( ){
146    for (int i=0;i<MAXELEVS;i++ )
147     ea[i].SetelevNumber(i);
148 }                       //创建电梯数组并赋全部初值
149
150 void elevSet::showElevators(void){
151    for(int i= 0 ;i<MAXELEVS;i++ )
152     ea[i].showElevator( ) ;
153 }                       //显示全部电梯状态
154
155 void elevSet::action(floorSet & floors ,
156                              persSet & persons) {
157    for(int i=0;i<MAXELEVS;i++ )
158    ea[i].action(floors ,persons);
159 }                       //每个电梯各动作一次
160
161 int elevSet::avgRiding(void) {
162    int total=0;
163    for(int i=0;i<MAXELEVS;i++)
164       total += ea[i].getPassengers();
165    return total/MAXELEVS ;
166 }                        //计算当前平均乘客数
```

程序说明：

(1)第 6—15 行构造函数 elevator()为电梯设置初始值，它的根据是，整个系统在开始时，所有电梯都是在大楼的底层，电梯内无人，停止状态按钮组全为弹起状态，至少停止 15 秒钟。

(2)第 41—53 行 setDirection()的功能是根据在各楼层等待电梯的人群要求决定电梯开始运行的方向，其判定策略是：

电梯内有人要求上行则上行：dir＝UP；
电梯内无人要求上行则再查：
电梯内有人要求下行则下行：dir＝DOWN；
电梯内也无人要求下行(空)则再查：
以下各层有人等电梯则下行；以下各层无人等电梯，这时说明电梯内无人要求向上或向下，且在该层以上或以下各层无人等电梯，故置 dir＝NODIRECTION

(3)第 55—73 行，函数成员 elevStopping()用来决定电梯是否应停下来，其判定策略是：

本电梯中有人要在本层下，则停止；
电梯中无人要下，但本层有人等电梯且与电梯同方向，应停止；
无人要下，又无人等且同向但电梯已到底层或顶层，也停止；
电梯在 1～8 层，但电梯处在“无向”状态，停止；
电梯处在 UP 或 DOWN 状态——继续运行。

(4)第 75 一 143 是本程序中又一个大而复杂的函数定义，它是根据下面的逻辑策略编写的：

根据有关状态决定电梯的动作：
电梯处在停止状态：
下载以该层为目标的乘客；
从乘客人数中减去下载乘客数；
电梯无定向并有空位且
该层有人等电梯
上一个乘客，修改相应参数；
电梯已定向但有空位
该层有人等待且方向一致
上一个乘客，修改相应参数；
如电梯满，准备启动；
若暂停时间到，转向启动；
确定运行方向(除非电梯内没有要求上行的乘客，各层又无人等待电梯而电梯又在底层，则继续停运)；
修正有关参数；
若暂停时间不到，时间减 1。
电梯处在运行状态：

运行时间为 0(到达下一层)：
改变层号，决定方向；
电梯应该停止：转向停止，修正参数；
电梯不该停止：再运行一层。
运行时间大于 0：该时间 timeToAction--。

其中比较复杂的是到达一状态时，如何决定电梯的运行方向。因为它与电梯内乘客的要求，本层等待电梯人员的要求和其它各层等待电梯人员情况以及所在楼层都有关系。

(5)第 17—27 的两个函数用于检查电梯内是否有人要求上行或下行。其依据就是电梯内的按钮数组。

(6)第 29—39 行的 showElevator(),负责在屏幕上显示该电梯的若干状态，这里从略。

(7)电梯组 elevSet 类实际上是 elevator 类的对象数组类，其函数成员自然包括：构造函数 elevSet()，当执行：elevSet elevators；时，自动地为每一电梯 ea[i]进行了初始化，同时为每一电梯赋予了电梯号。

Showelevators()显然是对组中每一电梯的状态进行显示。

action()；就是要求组中每个电梯完成一步动作。

另一个函数成员 avgRiding()用来计算当前各电梯中乘客人数的平均值。为显示统计数字作准备。

11.2.6

在前面几节，我们已经基本上完成了整个仿真程序的设计，从这个例子，我们应该学习到以下各点：

(1) 以类为核心的面向对象的程序设计方法。程序的大部分都组织到以类为核心的程序模块之中，如此划分的程序显得自然合理，模块内部的数据与方法(函数)之间联系密切，而类与类之间的联系相对较少，且清楚明了。一个好的 C 十十程序(指规模较大时)就应该是由彼此有一定关系的若干个类模块的组合。

(2) 在我们的例中，所采用的设计方法，大致是一种自顶向下(top—Down)方法，就是先抽象后具体，先整体后局部，先总体联系后局部细节。严格地说，我们的设计并没有完成。原因是：在抽象设计阶段提出的仿真过程中的三个主要操作

perform ()；

```
    display();      '
    results();
```

中只完成了 perform()(后面还要指出它的不足),而 display()虽然只把要求讲了，但它的细节实现并未给出，有待于进一步实现。至于 results()的设计，尚未涉及，它可以由读者按照自顶向下的方法来完成，相对于 perform () 的实现，display () 和 results () 的实现要容易得多。

(3) 一般的自顶向下的设计过程，大多数不是简单的单向设计行为，总要辅以反向的调整和修改。例如，整个系统的运行与动作，应该说是相当完整细致的，但也不能说没有问题。例如在 floor 类的设计中，我们漏掉了对于楼层等待人数的设置与修改功能，例如可以增加成员函数 SetNumWaiting()函数。另一方面在 person 类的 action()函数设计中，当一个人从工作状态进入等待(电梯)状态时，当一个人走进大楼进入等待状态时，应对该楼层的等待人数做加 1 处理；而当一个人进入电梯，或改走步行楼梯时，也应对该楼层的等待人数做减 1 处理。这种反向的程序修正是所有程序中都会碰到的。以类为核心的模块化设计使这种修改比较容易进行，不容易造成错误。

(4) 电梯系统的仿真应该说是一个比较复杂的编程问题，特别是把它作为教材的内容，其规模显然是大了一些。从这个例子可以看到，面向对象程序设计方法的思想实质是以数据为基点来组织程序。抓到了数据的特征，就抓到了程序的纲，整个程序的脉络就清楚了。电梯运行系统的数据包括：

① 时间：时间是贯穿整个仿真系统的最基本的数据。这是任何一种仿真程序(无论什么过程的仿真)最基本的数据，围绕时间的操作，就是 simulation 类中所设定的有关操作：时间初始化，设置，修改，判定是否到时，以及由时间控制的 perform()，display()，results()。

② 与电梯运行系统有关的三个实体：电梯组，人员集合和楼层集合，整个仿真过程就是这三个实体在相互关联中动作的过程。应该说整个程序设计的成功首先应归功于这一点。正确地把那个错综复杂的动态系统中的千头万绪，分成了三个运动中的实体，问题才逐渐清晰起来。从程序结构来看，最基本的，最有实质内容的程序模块，集中在类 person，floor，elevator 的定义中，围绕这三者设计的操作，例如 person∷action()，elevator∷action()，showFloor()，showElevator()等等，才是整个程序的关键。

(5) 面向对象的程序设计方法提供了由多个程序员合作编程的良好条件，在楼宇电梯仿真系统的设计过程中也充分体现了这一特点。例如，系统的总体设计，person 类，elevator 类，floor 类的设计是可以由不同的程序员完成的，另外，系统的显示和结果输出部分可由专人设计，由于 display ()，results

（）的运行与 perform（）的运行之间主要由一些全局变量相联系，因此，把显示和结果输出的设计作为类和对象独立出来，也是完全可行的。

思考题

1．简述本章提供的办公大楼电梯运行系统仿真程序的各个类模块划分、功能和相互关系。

2．通过对本章的学习，谈谈你对于面向对象程序设计过程中，围绕不同类型的数据及对这类数据的操作和运算来划分程序模块的思想的理解。设想这样的系统，如果按以函数为中心的 SP 设计方法完成，将是什么样子。对两种情形进行一次比较。

3．在本章电梯运行仿真程序的设计过程中自顶向下设计方法是如何体现的?它有什么优点?

练习题

1．为系统的显示(display())部分，重新进行你的设计：

(1)确定哪些类和成员与其有关需要修改；

(2)完成这些模块的程序设计。

2．文中提到，在设计中漏掉了对于各楼层等待人数(np)的处理，现在请你对系统的这一不足进行完善：

(1)确定它应涉及程序系统的哪些部分；

(2)完成有关模块的程序设计。

* 第十二章　异常处理

12.1　异常处理的基本思想以及 C++实现

程序运行中的某些错误（或意外情况）不可避免但可以预料。如，做除法或模运算时使用的分母 y 为 0，程序中可通过添加如下形式的测试语句来判断是否出现了这种意外情况（即异常），“ if （y==0）{ cout<<"Error occuring -- Divided by 0!"；exit（1）；}”，若出现的话，则用户程序将进行干预，比如先在屏幕上给出适当的错误提示信息，而后退出程序停止运行等。这实际上已经是在做异常处理的工作了，读者从前面章节的某些例题中可能早就注意到了这种对异常（或错误）的程序处理方式。实际上，C++还提供更方便的对异常进行处理的机制，那就是在程序中使用 try、catch 和 throw。

请看如下程序，其中使用了 try、catch 和 throw 来对除以 0 或模 0 所产生的异常进行处理。程序要求输入两个整数 i1、i2 以及一个运算符 op（“ / ”或者“ % ”），而后进行相应运算并对所出现的异常进行处理。

```
// program 12_1.cpp
#include <iostream.h>
void main() {
    try   {     //try 程序块为“ 受监控 ”的块，块中所抛掷的异常
                //将被本 try 块后的某一个 catch 块所捕获并处理
        int i1, i2;
        char op;
        cout<<"Input I1 I2 OP:";
        cin>>i1>>i2>>op;                    //输入两个整数 i1、i2 以及一个运算符 op
        if (op=='/'){
            if (i2==0)                      //分母为 0 时，抛掷一个字符串 — char*类型的异常
                throw "Divided by 0!";                  //所抛掷的异常将被随后的 catch 块捕获并处理
            cout<<i1<<" / "<<i2<<'='<<i1/i2<<endl;      //正常情况（分母非 0）的处理
        }
        else
            if (op=='%'){
                if (i2==0)                  //分母为 0 时，抛掷整数 i1 — int 型异常
                    throw i1;               //所抛掷的异常将被随后的 catch 块捕获并处理
                cout<<i1<<" % "<<i2<<'='<<i1%i2<<endl;          //正常情况处理
            }
            else
                cout<<"OP error -- must be '/' or '%'!"<<endl;  //限制 op 只能为“ / ”或者“ % ”
        cout<<"22 / 5="<<22/5<<endl;                    //再进行一些其他的处理
    }
    catch (int i) {                                     //捕获“ int ”类型的异常并进行处理
        cout<<"Error occuring -- "<<i<<" % 0 "<<endl;   //输出相关的异常信息后结束本 catch 块
    }
    catch (char * str) {                                //捕获“ char * ”类型的异常并进行处理
```

```
        cout<<"Error occuring -- "<<str<<endl;                    //输出相关的异常信息后结束本 catch 块
    }
    cout<<"End main function!"<<endl;                             //异常处理结束后，均转到此处执行该语句
}
```

程序执行后的输出结果为：

```
Input I1 I2 OP:33 5 %
33 % 5=3
22 / 5=4
End main function!
```

第二次执行：

```
Input I1 I2 OP:33 0 /
Error occuring -- Divided by 0!
End main function!
```

注意，通过 throw 抛掷异常后，系统将跳转到与所抛掷的实参（表达式）类型完全匹配的那一个 catch 块去执行，而执行完 catch 块后将不再返回，继而转到 catch 块序列的最后一个 catch 块的“下一语句”处去执行。

对实际问题进行处理的程序中，还会碰到多种多样的产生异常（或错误）的情况，如，磁盘中的文件被移动或缺少所需文件时将导致无法打开文件，内存不足致使通过 new 无法获取到所需要的动态空间，用户提供了不恰当的输入数据等。为使程序具有更好的容错能力并体现出程序的健壮性，设计程序时，应该充分考虑程序执行过程中可能发生的各种意外情况（即异常），并对它们进行恰当的处理。即是说，当异常出现后，要尽可能地让用户程序来进行干预，排除错误（至少给出适当的错误提示信息），而后继续运行程序。

下面对 try、catch 和 throw 的功能及其使用注意点做进一步的说明。

通过 try 可构成所谓的 try 块，用于标记运行时可能出现异常的程序代码。即是说，try 程序块在运行时将成为“受监控”的代码块，其中所抛掷的任何异常（包括在 try 块中直接抛掷的异常、以及通过所调用的“下层”函数而间接抛掷的各种异常）都将被捕获并处理（注意，抛掷异常是通过 throw 语句来完成的）。try 块的使用格式如下：

```
try {
  < “受监控”的语句序列（通常包含抛掷异常的 throw 语句） >
}
```

通过 catch 可构成所谓的 catch 块，它紧跟在 try 块的后面，用于监视并捕获运行时可能出现的某一种类（类型）的异常并对此种异常进行具体的处理（处理程序代码包含在 catch 块之中）。即是说，catch 程序块将监视并捕获处理程序异常，若没有异常出现的话，catch 程序块将不被执行。若准备处理多种类型的异常的话，要同时给出一个连续的 catch 块序列，届时系统将通过按序逐一“比对”所抛掷的异常类型（既可是系统预定义的基本类型，也可是用户自定义的某种类型）来确定执行该 catch 块序列中的哪一块。catch 块的使用格式如下：

```
catch ( <欲捕获的异常类型及形参名字> ) {
  < 对该类型异常的具体处理语句序列 >
}
```

注意，如果 catch 块的具体处理语句序列中根本不使用形参（对应值）的话，则 catch 块首的括号中可以只给出一个异常类型而不必给出形参名字。另外，还允许在 catch 块首括号中仅写上 3 个点符号（即省略号“...”），其使用含义为，该 catch 块将捕获任何类型的异常。即是说，它与任一种异常类型都可以“匹配”成功。若使用这种 catch 块的话，它应该

是 catch 块序列的最后一块，否则的话，其后的所有 catch 块都将起不到任何作用（永远不会被“匹配”成功。因为系统是按序与 catch 块序列中的每一块逐一进行“比对”的）。

抛掷异常的 throw 语句的使用格式为： throw < 表达式 >

通常使用“<表达式>”的特殊情况，如，一个变量或对象、一个常量或字符串等。系统将根据表达式的值类型来与各 catch 块首括号中形参的异常类型进行“比对”，若“匹配”成功（要求类型完全一致，系统并不自动进行类型转换），则跳转到那一 catch 块去执行，即进行相应的异常处理；若所有匹配都不成功，则将自动去执行一个隐含的系统函数“abort()”来终止整个程序的运行。

更确切地说，throw 语句将把运行时遇到的异常事件信息通知给相匹配的异常处理 catch 块代码（throw 后的“<表达式>”实际上起着实参的作用，它代表的正是异常事件信息，而 catch 块首括号中设立的正是所对应的形参）。与函数调用类似，通过 throw 抛掷异常后，也同样有一个实参与形参相结合的过程，形实参结合后，catch 块中处理的实际上是 throw 语句所提供的实参信息。但与函数调用不同的是，函数调用完成后仍要返回到调用处继续执行，而 throw 语句所进行的“调用”实际上是带有实参的跳转，跳转到的目标是形实参完全匹配的那一个 catch 块，而执行完 catch 块后将不再返回，继而转到 catch 块序列的最后一个 catch 块的“下一语句”处去执行。另外注意，catch 块必须有且仅允许有一个形参的说明（可以缺少形参名字，但不可以缺少其类型），throw 语句后携带的也只是一个相对应的实参（表达式）。

12.2 多级多层次捕获与处理

C++允许使用如下所述的多级多层次的捕获与处理方式。一是允许抛掷异常的语句处于捕获与处理异常的 catch 块所在函数的“下属层次”，例如，可以在主调函数中处理异常，而在被调函数中抛掷异常。即是说，若在本层次的函数中无法处理所抛掷的异常的话，系统总会自动沿着“调用链”一直向上传递那一异常。系统将首先找到异常抛掷句 throw 所处函数的“父辈”函数，进一步可能又要去找其“祖辈”函数，如此等等。

二是允许在同一程序中使用多个 try 块，而且这些 try 块可以处于不同的位置并具有不同的层次级别。抛掷异常后，系统将首先在相应的低层次级别的 try 后的 catch 块序列中寻找匹配，若不成功，将自动沿着“调用链”向上传递该异常，并在上一层次的 try 后的 catch 块序列中继续寻找匹配。直到最终匹配成功，或者已到达了最高层次的 main 函数后仍不成功时，则自动调用系统隐含函数“abort()”来终止整个程序的运行。

下述程序将对用户提供不恰当输入数据之异常进行处理：输入一个限定范围内的整数，若输入数据不处于限定范围之内时，则视为程序运行异常，而后抛掷并捕获处理这些异常。若输入正确，则在屏幕上显示出所输入的整数。另外，在程序中自定义一个异常类 myExcepCla，以方便对异常的处理。

对输入数据所限定的范围如下：只允许输入 -50 到 50 之间除去 0、11 与 22 之外的那些整数之一。具体对异常的处理设定为：

若输入值为 0，则抛掷一个“char* ”类异常

— 显示“main-catch::Exception : value equal 0! ”;

若输入值小于 -50，则抛掷一个自定义类 myExcepCla 异常

— 显示“main-catch::Exception : value less than -50! ”;

若输入值大于 50，则抛掷一个“char* ”类异常

— 显示“InData-catch::Exception : value greater than 50! ”;

若输入值为 11，则利用输入值抛掷一个 int 型异常

— 显示“ InData-catch::Caught INT exception! i=11 ”;

若输入值为 22，则抛掷一个 double 型异常

— 显示“ main-catch::Caught unknown exception! ”。

要求程序使用两个 try 块,一个处于 main 函数中,而另一个则处于被 main 调用的 InData 函数之中。 它们具有不同的位置并具有不同的层次级别。main 函数中 try 后的 catch 块序列捕获并处理“ char* ”、 myExcepCla 以及“ 其他 ”任何类型的异常；而 InData 函数中 try 后的 catch 块序列则捕获并处理 int 与“ char* ”类型的异常。

```
// program 12_2.cpp
#include <iostream.h>
#include <string.h>
class myExcepCla {                          //自定义的 myExcepCla 类（类型）
    char message[100];
public:
    myExcepCla(char* msg) {                 //构造函数，实参带来私有字符串数据 message
        strcpy(message, msg);
    }
    char* GetErrMsg() {                     //成员函数，获取私有数据 message 字符串
        return message;
    }
};
void InData();                              //函数原型
void main() {
    try    {                                //处于 main 中的 try 程序块，它为“ 受监控 ”的程序块
        InData();                           //调用自定义函数 InData，其中可能抛掷异常
        cout<<"End try block!"<<endl;
    }
    catch (char * str) {                    //捕获“ char * ”类型的异常并进行处理
        cout<<"main-catch::"<<str<<endl;
    }
    catch (myExcepCla ob) {                 //捕获“ myExcepCla ”类型的异常并进行处理
        cout<<"main-catch::"<<ob.GetErrMsg()<<endl;
    }
    catch ( ... ) {                         //捕获“ 其他 ”任何异常并进行处理
        cout<<"main-catch::"<<"Caught unknown exception!"<<endl;
    }
    cout<<"End main function!"<<endl;       //本层次的某个异常处理结束后，均转到此处执行该语句
}
void f(int val) {                           //自定义函数 f，负责判断 val 是否为 11 或 22，
    if(val==11)                             //输入值为 11，抛掷一个 int 型异常
        throw val;
    if(val==22)                             //输入值为 22，抛掷一个 double 型异常
        throw 123.86;
}
```

/* 注意， f 中所抛掷的 int 型异常，将首先被自动沿着“调用链”向上传递到其“父辈”函数 InData，而在 InData 中的 catch 块序列中，该异常恰好被匹配成功而得到处理。f 中所抛掷的 double 型异常，首先也被自动沿着“调用链”向上传递到其“父辈”函数 InData，而在 InData 中的 catch 块序列中无法匹配成功又进一步被传递到其“祖辈”函数 main，在 main 中与“catch (...)”匹配成功而得到处理。 */

```
void InData() {
    int val;
    cout<<"Input a INTEGER(from -50 -- 50, excpt 0 、11、22):";
    cin>>val;
    if (val==0)                    //若输入值等于 0，抛掷“char*”类异常，在 main 中被捕获处理。
        throw "Exception : value equal 0!";
    try    {                       //处于 InData 中的 try 程序块，它为“受监控”的另一个程序块
        if (val<-50) {
            myExcepCla exc("Exception : value less than -50!");
            throw exc;         //该异常将被 main 中 try 后的“catch (myExcepCla ob)”所捕获处理
        }
        if (val>50)
            throw "Exception : value greater than 50!";     //在本 try 后的 catch 块序列中被捕获处理
        else
            f(val);                                         //被调函数 f 中可能抛掷异常
        cout<<"OK! value="<<val<<endl;                      //不发生异常时，屏幕显示出输入值 val
    }
    catch (int i) {                                         //捕获“int”类型的异常并进行处理
        cout<<"InData-catch::"<<"Caught INT exception! i="<<i<<endl;
    }
    catch (char * str) {                                    //捕获“char *”类型的异常并进行处理
        cout<<"InData-catch::"<<str<<endl;
    }
}
```

注意，InData 函数中 try 后的某一个 catch 块被执行后（异常被处理结束后），将结束 InData 函数，返回到其主调函数 main 的 try 程序块的 InData 函数调用语句的下一语句处，即是说，将接着去执行 main 中的“cout<<"End try block!"<<endl;”语句。

程序执行后的输出结果为：

```
Input a INTEGER(from -50 -- 50, excpt 0 、11、22):0
main-catch::Exception : value equal 0!
End main function!
```

第二次执行：

```
Input a INTEGER(from -50 -- 50, excpt 0 、11、22):88
InData-catch::Exception : value greater than 50!
End try block!
End main function!
```

第三次执行：

```
Input a INTEGER(from -50 -- 50, excpt 0 、11、22):50
OK! value=50
End try block!
```

End main function!

本实例程序中使用了两个 try 块，一个处于 main 函数中，而另一个则处于被 main 调用的 InData 函数之中。它们具有不同的位置并具有不同的层次级别。抛掷异常后，系统总是首先在相应的低层次级别（如本例的 InData 函数）的 try 后的 catch 块序列中寻找匹配，若不成功，再自动沿着“调用链”向上传递该异常，并在上一层次（如本例的 main 函数）的 try 块后的 catch 块序列中继续寻找匹配。

另一个注意点是：若程序中含有多个不同位置并具有不同层次级别的 try 块时，一旦异常被某一个层次的 catch 块所捕获，在执行完那一 catch 块后，系统的继续运行“点”将是本层次 catch 块序列之最后一个 catch 块的下一语句处（若那里不再有其他语句时，则结束那一函数而返回到上一层次的主调函数中）。

本程序还自定义了一个异常类 myExcepCla，通过对它的使用，可建立起一种对异常进行处理的统一模式。

12.3 系统自动进行“堆栈展开”过程

C++进行异常处理时，总要自动地进行如下所述的一个“堆栈展开”过程：每当抛掷一个异常后，系统首先找到能捕获处理该异常的 catch 块（注意，其中可能自动沿着“调用链”多次向上传递该异常，并假定在某一层次的 try 块后的 catch 块序列中找到了相匹配的 catch 块），紧接着系统要利用所抛掷的对象（throw 句中的“实参”）对相应 catch 块的“形参”进行“初始化”，而后将立即检查从抛掷异常的那一 try 块的块首到该异常被抛掷之间已构造但尚未析构的那些处于堆栈中的局部对象（与变量）。若有的话，系统将自动对这些局部对象进行相应的退栈与析构处理（通过自动调用各对象的相应析构函数来完成，析构对象的顺序与构造它们的顺序恰好相反），再往后才去执行相应 catch 块中的代码而完成对异常的处理。

系统之所以要这样做，是因为异常处理机制中，通过 throw 语句抛掷异常后，系统实际上是要找到并“跳转”到捕获异常的 catch 块去执行，而且在执行完那一 catch 块后将不再返回（到 throw 语句之后），继而转到 catch 块序列的最后一个 catch 块的“下一语句”处去执行。正是由于这种“跳转”后的不再返回，若抛掷异常的 throw 语句处于某一下属层次的局部作用域（如某个局部块作用域或某个函数作用域）之中时，throw 的“跳转”实际上相当于跳出了那些作用域，所以系统将自动检查在那些作用域中已经构造但尚未析构的处于堆栈中的局部对象（与变量），并自动进行相应的退栈与析构处理。

请分析以下程序执行后会显示出什么结果，注意系统对局部块作用域中的局部对象 ob1 和 ob2 所进行的自动析构处理。

程序中说明了一个具有显式构造函数和析构函数的自定义类 ClaA，并在其中显示目前已经进入该构造函数或析构函数的提示信息（因为要通过输出信息来观察系统提供的自动析构处理功能）。

编制 main 主函数，并在其中设置局部块作用域，且说明 ClaA 自定义类的局部对象 ob1 和 ob2，之后通过抛掷“char*”类型的异常“跳转”出该局部块作用域（此时系统将对局部对象进行自动析构处理）。

由于要抛掷并处理异常，将 main 中的程序段置于 try 块的块体之中，并添加捕获“char*”类型异常的 catch 块。

本实例的重点在于理解异常处理过程中的“堆栈展开”以及对象的自动析构处理功能。

```
// program 12_3.cpp
#include <iostream.h>
```

```
#include <string.h>
class ClaA {                                    //自定义类 ClaA
    char s[100];
public:
    ClaA (char * str) {             //构造函数，带来私有数据成员 s 字符串的值，并显示构造信息
        cout<<"Constructing ClaA obj -- "<<str<<endl;
        strcpy(s, str);
    }
    ~ClaA() {                                   //析构函数，显示析构信息
        cout<<"Destructing ClaA obj -- "<<s<<endl;
    }
};
void main() {
    cout<<"Begin main function!"<<endl;
    try    {                                    //try 程序块为“受监控”的程序块
        cout<<"Enter try-block!"<<endl;
        {                                       //出现了一个局部块作用域
            cout<<"Begin a block-statement!"<<endl;
            ClaA ob1("ob1"), ob2("ob2");        //局部于块的对象 ob1 和 ob2 均被分配在系统的堆栈中
            cout<<"Throwing exception, in block-statement!"<<endl;
            throw "throwing 'char*' exception in block-statement!";      //抛掷一个“char*”异常
            cout<<"After throw, in block-statement! End block-statement!"<<endl;
        }
    }
    catch (char * str) {                        //捕获“char *”类型的异常并进行处理
        cout<<"In catch-block, deal with: "<<str<<endl;
    }
    cout<<"Execution resumes here. End main function!"<<endl;            //异常处理结束后，转到此处执行
}
```

程序执行后，屏幕显示结果为：

```
Begin main function!
Enter try-block!
Begin a block-statement!
Constructing ClaA obj -- ob1
Constructing ClaA obj -- ob2
Throwing exception, in block-statement!
Destructing ClaA obj -- ob2
Destructing ClaA obj -- ob1
In catch-block, deal with: throwing 'char*' exception in block-statement!
Execution resumes here. End main function!
```

注意，上一程序中，抛掷异常的 throw 语句处于一个局部块作用域内，throw 的“跳转”实际上相当于跳出了那个局部块作用域（“跳转”到相呼应的那一 catch 块而不再返回），所以系统将自动检查在那一局部块作用域中已经构造但尚未析构的处于堆栈中的局部对象 —如本例的 ob1 和 ob2，并自动对它们进行相应的退栈与析构处理（析构顺序为 ob2 而后 ob1）。

实际上，当抛掷异常的 throw 语句处于下属层次的某个函数作用域之中时，throw 的“跳转”就相当于跳出了那个函数作用域（而不再返回），那时系统同样也将自动检查在那一函数作用域中已经构造但尚未析构的处于堆栈中的局部对象,并自动进行相应的退栈与析构处理。

例如，若将本例程序 main 函数中的 try 块改写为：

```
try {
    cout<<"Enter try-block, calling fun!"<<endl;
    fun();
}
```

而又在程序中添加如下形式的一个 fun 函数的话，则系统会自动对 ob1 与 ob2 进行相应的退栈与析构处理，读者可作为一个练习去完成，并上机进行调试验证。

```
void fun() {
    cout<<"Begin fun!"<<endl;
    ClaA ob1("obj1"), obj2("ob2");
    cout<<"Throwing exception, in fun!"<<endl;
    throw "throwing 'char*' exception in fun!";
    cout<<"After throw, in fun! End fun!"<<endl;
}
```

注意，fun 函数中所抛掷的异常，是由其父辈函数 main 处的 catch 块所捕获并处理的。

思 考 题

1．什么叫做异常？ 什么叫做异常处理？

2．C++提供了什么样的异常处理机制？该机制有何优点？

3．如何配合使用 C++的 try、catch 和 throw 来完成异常处理工作？

4．catch 块序列中各块的顺序是否重要？

5．在 catch 块首括号中仅写上 3 个点符号（即省略号“...”），其使用含义是什么？

6．如何理解 throw 语句所进行的“调用”实际上是带有实参的跳转？

7．系统如何来确定由 throw 跳转到的目标 catch 块？

8．执行完某个相匹配的 catch 块后，将转到何处去接着执行？

9．C++是否允许使用多级多层次的捕获与处理方式？

10．若在本层次的函数中无法处理所抛掷的异常的话，系统是否要自动沿着“调用链”一直向上传递那一异常？

11．是否允许在同一程序中使用多个 try 块，而且这些 try 块可以处于不同的位置并具有不同的层次级别？

12．若程序具有多个 try 块时，抛掷异常后，系统将怎样去寻找匹配的目标 catch 块？

13．进行异常处理时，系统是否要自动进行一个所谓的“堆栈展开”过程？它要做些什么事情？

14．系统为什么要自动进行“堆栈展开”过程？若系统不帮着这样做的话，设想由用户程序来处理相同工作是否将遇到很大的麻烦？

练 习 题

1．无法打开文件的异常处理。编程序，打开用户指定的 text 文件，读出其中的各行内容并显示在屏幕上。但要求进行如下的异常处理：当文件不存在而打不开时，认为是一种错误，此时通过抛掷与捕获异常来处理该错误。要求程序抛掷、捕获“char*”类型的异常并进行处理 — 输出相关的提示警告信息后，不

再进行相关文件的后继处理，通过 exit 退出程序而结束。若打开文件成功，则将文件中的各行内容读出并显示在屏幕上。

2．内存不足获取动态空间失败的异常处理。编程序，按照用户指定的次数 times，通过使用 new 来进行 times 次数的动态内存分配，每次分配一个固定大小的非常大的结构体变量空间。但要求进行如下的异常处理：通过 new 进行内存动态分配失败时（没有足够的空间可供分配时）将抛掷一个异常而后由程序捕获并进行处理。要求程序抛掷、捕获“char*”类型的异常并进行处理 — 输出相关的警告信息，而后结束那一 catch 块，继而转到 catch 块的下一语句处去执行。

注意，在具有不同配置不同软、硬件环境的计算机上运行该题的所编程序时，可能会显示出不同的结果。这是因为：只有所在计算机在当时的运行环境下再也没有可供编译系统进行分配的动态存储空间时，才产生并处理这种异常。

3．使用 throw 抛掷不同类型的异常并进行处理。按照用户进行的输入指定（1—6，或其他），使用 throw 抛掷不同类型的异常，而后由程序捕获这些异常并进行处理。要求程序抛掷、捕获下述不同类型的异常：1—int、2—double、3—char、4—myCla、5—student、6—char*、其他—...（其中的“...”表示任何类型的其他异常）。其中的 myCla 为用户自定义的类类型，而 student 则为用户自定义的结构体类型。每一 catch 块都是在捕获某种类型的异常后，首先输出相关的提示信息，而后结束那一 catch 块，继而转到最后一个 catch 块的下一语句处去执行。

4．对 throw 所跳过的函数作用域之中的局部对象进行自动析构。模仿 12.3 节的“program 12_3.cpp”，并参照那一程序之后所给提示，将 main 函数中的 try 块改写为：

```
try {
    cout<<"Enter try-block, calling fun!"<<endl;
    fun();
}
```

而后又在程序中添加一个具有 ClaA 类局部对象说明 ob1 和 ob2 的 fun 函数，并在该函数中通过使用 throw 来抛掷异常，该异常将被其父辈函数 main 处的 catch 块所捕获并处理。

进一步，可设计在 main 函数中调用自定义函数 fun（fun 中说明 ClaA 类的局部对象 ob1 和 ob2），而在 fun 中又进一步调用自定义函数 fun2（fun2 中说明 ClaA 类局部对象 ob21 和 ob22），在 fun2 中抛掷异常，并在 main 的 try 块后的 catch 块中捕获并处理异常。此时将会涉及到两个层次的函数作用域中 4 个局部对象的自动析构处理过程 （抛掷异常的 throw 语句与捕获处理该异常的 catch 块之间相隔了两个作用域层次 — 因为要在 fun2 函数的父辈的父辈函数 main 中捕获并处理异常，从而系统要负责对 throw 语句所跳过的那两个作用域层次中处于堆栈中的局部对象进行自动析构处理）。