

面向对象程序设计

(第 2 版)

刘正林 编著

华中科技大学出版社

内 容 简 介

本书以 ISO/ANSI C++ (ISO 14882) 标准为准则, 以美国 Microsoft 公司开发的 Visual C++ V6.0 为语言蓝本, 遵循全国计算机等级考试 C++ 科目的考试大纲, 全面、系统、完整地讲授面向对象程序设计方法的核心概念、主要语言特性、思维方式和面向对象程序设计技术。按照循序渐进, 突出重点, 深入浅出, 融会贯通的教学原则, 编写成自成体系的 C++ 教科书。每章都有小结, 归纳出必须掌握的重点内容, 并附有大量的习题, 以加深读者对重点内容的理解。在内容安排上有深有浅, 且侧重应用, 适用于各个层次的读者, 既适合以前从未接触过 C++ 的初学者, 也适合具有一定编程基础的读者作为学习面向对象程序设计方法, 提高编程能力的教材和参考书, 它也是广大电脑爱好者参加全国计算机等级考试, 备考 C++ 科目的教材和参考书。

图书在版编目(CIP)数据

面向对象程序设计(第2版)/刘正林 编著
武汉:华中科技大学出版社,2004年10月
ISBN 7-5609-

. 面
. 刘...
.
.

面向对象程序设计(第2版) 刘正林 编著

责任编辑:吴锐涛 封面设计:刘 卉
责任校对:刘 竣 责任监印:

出版发行:华中科技大学出版社 武昌喻家山 邮编:430074 电话:(027)87542624

经销:新华书店湖北发行所

录排:华中科技大学惠友科技文印中心 电话:(027)87543977
印刷:

开本:787×960 1/16	印张:	字数:612 000
版次:2004年10月第2版	印次:2004年10月第2次印刷	印数:
ISBN 7-5609- /		定价: 元

(本书若有印装质量问题,请向出版社发行部调换)

前 言

崭新的 21 世纪，以现代电子信息产业为龙头的全球经济一体化浪潮正席卷世界，这是当今人类所面临的巨大挑战，人们将认真面对挑战的内涵和挑战所带来的机遇。而以 IT (Information Technology) 技术为基础的信息产业正深入到人类社会生活的方方面面，无论是在生产制造、商业、国防和科技等领域，还是在第三产业，计算机软件都已成为担当重任的技术手段，互联网和软件已成为推动新经济发展的重要基础。因此，计算机软件技术将是各类专业的大专生、本科生和研究生所必备的基础知识和基本技能。面向对象的程序设计 (简称 OOP) 是软件技术的一场革命，必将成为 21 世纪普遍采用的软件开发方法。C++ 是面向对象程序设计的第一个大众化版本，是当前学习面向对象程序设计方法的首选语言。C++ 是著名 C 语言的面向对象扩展，最初设计 C 语言时，是将其作为一种面向系统软件 (操作系统和语言处理系统) 的开发语言，即用来代替汇编语言的，但是由于它具有强大的生命力，因而在事务处理、科学计算、工业控制和数据库技术等各个方面都得到了广泛应用。即便进入到以计算机网络为核心的信息时代，C 语言仍然是作为通用的汇编语言使用，用以开发软 (件) 硬 (件) 相结合的程序，如实时监控程序、控制程序和设备驱动程序等。而 C++ 是 C 语言的超集，它保留了 C 的所有组成部分而与其完全兼容，既可以进行传统的结构化程序设计，又能进行面向对象程序设计，是当今世界最为流行的面向对象程序设计语言。它的发展领导了程序设计语言变革的新潮流，大有取代以往程序设计语言之趋势。在系统软件的开发研究上，C++ 的运行效率与 C 相比毫不逊色，在大型应用软件开发上，以 Windows 开发环境为平台的 C++ 标准类库和组件正在迅猛发展，C++ 即将取代 C 已是不可抗拒的事实，它的触角几乎已触及到计算机研究和应用的各个领域。

1980 年 Bjarne Stroustrup 博士以 C 和 Simula 语言为基础创建了 C++，它是面向对象程序设计语言的第一个大众化版本，也是人类实施面向对象方法及其理论的第 1 个 (不是最好的) 成功案例。本书以 1998 年正式批准的 ISO/ANSI (American National Standard Institute) C++ 标准 (ISO 14882) 为准则，以美国 Microsoft 公司开发的 Visual C++ V6.0 为语言蓝本，目的是便于学生尽快掌握像 Visual C++ 和 C++ Builder 这些优秀可视化的应用程序开发工具，快速冲向计算机应用领域的前沿，成为各类专业从事计算机应用、具有相当的应用软件开发能力的技术人才。众所周知，面向对象程序设计是面向对象方法及其理论在程序空间的具体实现，本书首先概略介绍面向对象程序设计方法的核心概念，给学生一个概略的、且较为完整的系统概念，同时针对初学者，从实用的角度出发，重点讲解在 Windows 操作系统平台上用 Visual C++ V6.0 开发 C++

源程序所必须掌握的基本方法。接着系统介绍 C++ 所支持的面向对象程序设计方法的核心内容。使学生具有阅读和编写面向对象风格的 C++ 程序的能力。为进一步学习完全面向对象、面向计算机网络的 Java 语言打下牢固的基础。由于面向对象程序设计方法与传统结构化程序设计方法在思维方式、核心概念、编程技术等方面差别很大,新概念、新技术和新的专业术语较多。为了使读者便于学习,我们总结了多年教学和科研的实践经验,根据“循序渐进,突出重点,深入浅出,融会贯通”的教学原则,对内容进行了精选,强化如函数及其参数、指针和引用、链表和排序、类和对象、继承和虚函数等重要概念,舍去了如联合体和保护型继承之类的一些即将退化或应用很少的概念。重点介绍类的封装性、继承性和多态性等。

作者发扬“蜜蜂采蜜”的精神,广泛收集国内外最新发展动态信息和应用实例,并总结多年教学和科研的成果,经过消化和吸收,以“面向对象方法及其理论”为基础,酿造成“思路新颖,重点突出,逻辑清晰,易学易用”的知识奉献给读者,这是作者的创新之举。

本书以“实用、创新、深入浅出、通俗易懂”为特色,讲述思路清晰,层次分明,逻辑性强。为便于理解,决不生硬翻译国外的语言手册,避免使用晦涩难懂的语言,对于日新月异的计算机领域的许多新专业术语则采用通俗易懂的大众化语言讲述,对核心概念做到图文并茂,并举实例加以说明。书中的所有例程和习题中的程序都在目前国内最流行的 Visual C++ V6.0 上成功地运行过,在其他版本的 C++ 语言系统,如 C++ Builder V3.0 上都可以运行。第 6 章所述的模板类,是 C++ 语言系统目前发展的一种趋势,如 Microsoft 公司在 20 世纪末推出的 ATL (Active Template Library) 活动模板类库,业已成为开发应用程序的得力工具。它与微软标准基类库 Microsoft Foundation Classes (简称 MFC) 结合在一起,已成为目前绝大多数应用程序进行 Windows 开发的工业标准。活动模板库 ATL 和 WFC 是微软最新推出的应用程序框架。所谓“应用程序框架”是一个由配合完好的一组 class 类型构成的完整程序模型,具有标准应用软件所需的一切基本功能。

应用软件的开发是从事计算机应用的各类专业技术人员的大舞台,大都由从事计算机应用的各类专业人员来完成。这部分人员多数毕业于非计算机专业,不仅人数众多,而且分布在广阔的各类应用领域内。他们既具有扎实的本专业基础理论知识和丰富的实践经验,又熟悉计算机应用技术方面的知识,善于用计算机作为强有力的辅助工具去完成本应用领域的各种任务,在计算机应用领域的大舞台上发挥着其他人才无法替代的重要作用,是应用领域不可缺少的基本力量。因此,对非计算机专业的技术人员普及 OOP 的基础知识是当前刻不容缓的任务。本教材以此为编写宗旨,在内容安排上也有深有浅,因此,适合于大专院校理、工科各类专业学生作为“面向对象程序设计”课程的教科书。

2003 年 11 月国家教育部考试中心发出通知,从 2004 年开始将对“全国计算机等级

考试”作重大调整，新增了“C++”、“Java”和“Access”等3个科目，本书第2版按照2004版NCRE考试大纲“C++”科目的要求调整所编写的内容，因此，它也是广大电脑爱好者参加全国计算机等级考试，备考“C++”科目的教材和参考书。

当今世界，计算机硬件已走向标准化、规模化生产，性能价格比飞速提高，业已形成“6个月开发（计算机硬件），3个月上市，1个月清仓”的运营模式，应用软件的开发已成为担当重任的核心力量，互联网和软件已成为推动新经济发展的重要基础。开发任务基本上都是在开放系统平台上进行应用软件的二次开发工作。20世纪末风起云涌的Internet网和WWW万维网以及Java技术是当前IT界的三大热门话题，由美国Sun Micro System公司所创造的Java语言由C++发展而来的，它保留了C++大部分内容，并创造性地整合成完全面向对象、面向计算机网络的编程语言，它的发展已不仅仅只是一种计算机语言了，现已形成为Java技术。美国Microsoft公司为了维护自身的垄断地位将竞争对手从市场中排挤出去，推出了Visual Studio.NET（其内包括C#）企图打败Java。因此，我们面临的就这两种选择，在从事应用软件的二次开发工作时是选择Java技术，还是选择微软公司的.NET？目前Java技术的应用在国外暂时领先，而在国内暂时处于弱势。但是，任何一种技术要想主宰或垄断未来的计算机领域都是不可能的，不管是操作系统也好，语言处理系统也好，还是应用程序的开发软件也罢，都将是多种技术并驾齐驱、各显神通的格局，这是事物发展的必然趋势，是不以人们的意志为转移的客观规律。

在本书编写过程中，得到华中科技大学研究生院和出版社的大力支持，在此表示衷心的感谢！

作者的电子邮件地址：Cowherd_17@hotmail.com

通信地址：湖北省武汉市珞喻路1037号（邮编：430074）

华中科技大学主校区喻园小区39号402室 刘正林 收

编 者
2004年8月于武汉

目 录

第 1 章 概论	(1)
1.1 面向对象程序设计的有关概念和基本思想	(1)
1.1.1 面向对象程序设计的有关概念	(1)
1.1.2 面向对象程序设计的基本思想	(4)
1.2 面向对象程序设计的要点	(4)
1.2.1 抽象数据类型	(4)
1.2.2 消息传递机制	(7)
1.2.3 继承	(8)
1.3 C++ 程序结构的特点	(9)
1.3.1 标识符	(10)
1.3.2 预处理语句	(11)
1.3.3 输入/输出流操作语句	(11)
1.3.4 函数和语句	(14)
1.3.5 标准名空间和其他组成部分	(14)
1.4 Visual C++ V6.0 使用方法	(19)
1.4.1 源程序的编辑、存储和建立	(19)
1.4.2 编译、链接和运行源程序	(21)
1.4.3 关闭源程序	(25)
1.4.4 调试器的使用方法	(25)
1.4.5 查找信息	(26)
1.4.6 建立工程文件	(26)
小结	(29)
习题 1	(30)
第 2 章 从 C 快速过渡到 C++	(34)
2.1 数据类型	(34)
2.1.1 基本数据类型	(34)
2.1.2 复杂数据类型	(36)
2.2 C++ 的常量和变量	(37)

2.2.1	常量	(37)
2.2.2	变量	(41)
2.3	C++的指针	(44)
2.4	引用变量	(45)
2.4.1	“引用”的概念	(45)
2.4.2	引用的初始化	(45)
2.4.3	引用的使用	(46)
2.5	C++的运算符	(48)
2.5.1	表达式中的类型转换	(48)
2.5.2	new 和 delete 运算符	(50)
2.5.3	C++的运算符集	(54)
2.6	C++的函数	(55)
2.6.1	引用的应用	(56)
2.6.2	设置函数参数的默认值	(61)
2.6.3	内联函数	(63)
2.6.4	函数重载	(65)
	小结	(68)
	习题 2	(68)
第 3 章	类和对象	(84)
3.1	类的定义	(84)
3.1.1	类的定义格式	(84)
3.1.2	访问限制符	(87)
3.1.3	数据成员	(90)
3.1.4	成员函数	(92)
3.2	对象的定义	(93)
3.2.1	对象的定义格式	(93)
3.2.2	对象指针和对象引用的定义格式	(94)
3.2.3	访问类对象成员的方法	(96)
3.3	对象的初始化	(99)
3.3.1	构造函数和析构函数	(99)
3.3.2	构造函数的种类	(103)
3.3.3	C++的结构体	(119)
3.4	对象指针和对象引用的应用	(120)

3.4.1	对象和对象指针作为函数的参数	(120)
3.4.2	对象引用作函数参数	(122)
3.4.3	this 指针	(123)
3.4.4	递归类	(128)
3.5	静态成员	(132)
3.5.1	静态数据成员	(132)
3.5.2	静态成员函数	(136)
3.6	常量对象和常量成员	(139)
3.6.1	常量对象	(139)
3.6.2	常量成员函数	(141)
3.6.3	常量数据成员	(143)
3.7	友元	(145)
3.7.1	友元函数	(146)
3.7.2	友元类	(153)
3.8	标识符的作用域、可见性和名空间	(154)
3.8.1	标识符的作用域规则	(154)
3.8.2	作用域的种类	(154)
3.8.3	头文件	(157)
3.8.4	标识符的名空间	(159)
3.9	对象数组和成员对象	(166)
3.9.1	对象数组	(166)
3.9.2	对象成员和容器类	(170)
3.10	对象的存储类	(173)
3.10.1	对象的生存期和存储区域	(173)
3.10.2	各种存储类的对象	(174)
	小结	(181)
	习题 3	(183)
第 4 章	派生类、基类和继承性	(201)
4.1	继承的概念	(201)
4.1.1	什么是继承	(201)
4.1.2	两种继承类型	(202)
4.2	单继承的派生类	(203)
4.2.1	派生类的概念和定义	(203)

4.2.2	公有继承和私有继承	(205)
4.2.3	基类对象和派生类对象	(208)
4.2.4	基类和派生类的成员函数	(212)
4.2.5	C++ 结构体的继承	(213)
4.2.6	继承的传递性	(214)
4.3	派生类的构造函数和析构函数	(219)
4.3.1	派生类构造函数定义格式	(219)
4.3.2	派生类构造函数和析构函数的执行次序	(223)
4.4	基类和派生类的赋值规则	(223)
4.4.1	赋值兼容性规则	(223)
4.4.2	基类和派生类的对象指针	(225)
4.4.3	子类型和类型适应	(229)
4.4.4	不能继承的部分	(231)
4.5	多继承	(232)
4.5.1	多继承派生类	(232)
4.5.2	多继承派生类的构造函数	(234)
4.5.3	虚基类	(234)
小结		(240)
习题 4		(240)
第 5 章	多态性和虚函数	(252)
5.1	C++ 的多态性	(252)
5.2	运算符重载	(253)
5.2.1	运算符函数的定义	(253)
5.2.2	运算符重载规则	(258)
5.2.3	重载赋值运算符	(259)
5.2.4	典型应用	(264)
5.3	其他运算符的重载	(271)
5.3.1	重载增量、减量运算符	(271)
5.3.2	函数调用运算符()的重载	(273)
5.3.3	下标运算符[]的重载	(275)
5.4	同名成员函数	(279)
5.4.1	重载成员函数	(279)
5.4.2	基类和派生类的同名成员函数	(279)

5.4.3 基类指针和派生类对象	(281)
5.5 虚函数	(285)
5.5.1 静态联编	(285)
5.5.2 虚函数机制和动态联编技术	(286)
5.5.3 典型例程	(289)
5.5.4 虚函数表	(291)
5.6 纯虚函数和抽象类	(292)
5.6.1 纯虚函数	(292)
5.6.2 抽象类	(295)
5.7 虚析构造函数	(299)
小结	(303)
习题 5	(304)
第 6 章 模板	(316)
6.1 函数模板	(316)
6.1.1 引入函数模板	(316)
6.1.2 函数模板的定义	(319)
6.1.3 模板函数的调用和模板实参的缺省	(324)
6.2 类模板	(327)
6.2.1 类模板的定义	(327)
6.2.2 模板类的实例化和对象的定义	(330)
6.2.3 类模板的继承	(332)
小结	(335)
习题 6	(336)
第 7 章 C++ 的流库	(341)
7.1 流库的类层次结构	(341)
7.1.1 什么是流	(341)
7.1.2 流库的类层次结构	(343)
7.1.3 4 个预定义标准流对象	(352)
7.2 输出流	(353)
7.2.1 内部数据类型的输出	(353)
7.2.2 ostream 类中的主要成员函数	(354)
7.2.3 用户定义的 class 类型的输出	(355)
7.3 输入流	(357)

7.3.1	istream 类中的主要成员函数	(357)
7.3.2	用户定义类型的输入	(359)
7.4	格式控制	(363)
7.4.1	格式控制的输入/输出和无格式的输入/输出	(363)
7.4.2	C++新标准流库的格式控制	(364)
7.4.3	设置和清除格式控制标志的成员函数	(366)
7.4.4	操作符	(367)
7.4.5	读取格式控制标志的成员函数	(371)
7.4.6	设置域宽、填充字符和浮点精度	(373)
7.5	文件 I/O 流	(376)
7.5.1	文件的打开和关闭	(376)
7.5.2	ifstream、ofstream 和 fstream 类的构造函数和析构函数	(380)
7.5.3	文件的读/写	(381)
7.5.4	特殊的文件流	(388)
7.5.5	格式化的输入/输出文件	(389)
7.6	内存格式化 I/O 流	(394)
小结	(396)
习题 7	(398)
附录	ASCII 码表	(406)
参考文献	(407)

第 1 章 概 论

1.1 面向对象程序设计的有关概念和基本思想

1.1.1 面向对象程序设计的有关概念

近 10 年来所兴起的面向对象程序设计 OOP (Object-Oriented Programming) 方法是一种新的软件开发方法，它是计算机软件开发方法的一场革命。21 世纪的软件开发人才，不管从事何种专业，都必须具有 OOP 的基本知识，否则将难以适应时代发展的需要。

例如，计算机网络技术、多媒体技术、计算机集成制造系统 CIMS、人工智能等高科技的飞速发展，就迫切要求进一步改进系统的研究方法，特别是要求提高计算机软件的开发效率和质量。而过去所学的各种程序设计语言(包括 C 语言)都是面向功能的(Function Oriented)，称为传统的程序设计语言，使用这些语言来开发软件，效率很低。它首先要求编程者详细了解所研究对象的具体细节，包括它的功能、外观、内部结构、各种状态和基本原理等；然后再考虑在程序中如何用数据来描述它，还要花费很大的精力研究出一些算法，设计出一些函数来操作这些数据，改变它的状态，并实现各种功能。这就是面向功能的程序定律：

$$\text{程序} = (\text{算法}) + (\text{数据结构})$$

所谓“算法”是一个包含有限条指令的集合，这些指令确定了解决某一特定类型问题的运算序列，它是一个独立的整体。数据结构(包括数据类型和数据)也是一个整体。两者分开设计，以算法(函数或过程)为主，如图 1.1 所示。

随着实践经验的不断积累，软件工程师越来越注重把数据结构与算法看做一个独立的功能模块。程序定律重新被认定为：

$$\text{程序} = (\text{算法} + \text{数据结构})$$

即算法与数据结构是一个不可分割的整体,因为:

(1) 算法总离不开数据结构,算法是用来访问数据结构的,所以算法只能适用于特定的数据结构;

(2) 设计程序时,最好使算法与数据结构构成一对一(1:1)的关系,如图 1.2 所示,一种数据结构对应一个算法,对同一个数据结构的多个算法实现是多余的,也是没有必要的;

(3) 若数据结构改变了,则必须重新设计算法。

这种程序设计方法就类似于在硬件领域里,一个技术员安装一台电脑。若想给电脑添加一块“声卡”,是用最原始的方法,用集成电路芯片和材料去制作一块“声卡”,还是向电脑制造厂商选购一块“声卡”。前者要求技术员掌握“声卡”的基本原理、功能、内部结构、各种状态及调试技能等许多具体细节,显然这是一种效率很低的开发方法。软件开发如果走过去开发硬件的这条老路,矛盾更突出,不仅开发效率难于提高,而且程序代码的可重用性和软件的可维护性很差。尤其可维护性,它是软件产品质量的重要指标之一,因为软件产品不同于硬件产品,硬件产品一旦开发成功,只要达到所规定的技术指标就是成熟的产品,而软件产品不同,它总在不断修改、完善,从不成熟逐步走向成熟,因此软件要求“可维护性好”是必不可少的质量指标。传统的结构化语言,例如 C 语言是用函数实现对数据结构的操作,这就是用函数实现算法。如前所述,算法与数据结构应构成一对一的关系,特定的函数往往是操作特定的数据结构。若程序在维护时,数据结构改变了,则必须编写新的操作函数,这对程序的修改、完善极为不利。

1. 数据封装和信息隐藏(Data Encapsulation and Information Hiding)

在高科技时代的社会里,科技成果是人类共同拥有的财富。为了充分利用现代科技成果,必须减少处理的事情,根据所要达到的目的进行抽象,即抓住与目的有关的重要信息,忽略掉某些不重要的细节。例如,人类发明了电话,它已是人们进行通信的强有力工具之一。但绝大多数人只是以“使用它”为目的,只需知道它的功能和使用方法就可以了,而不必了解电话是如何实现通话的具体实现细节。因此,电话制造厂家将“细节”放在电话机壳内隐藏起来,封装成一个自成一体的电话机,而将一些操作按钮安装在机壳上,作为人-机操作的接口界面。人们只需操作电话机壳上的操作键就可以控制电话机,实现与外界的通话。又如,电脑制造厂商将“声卡”做成一块模板,自成一體。因为技术员一般关心的是“声卡”的功能,即只需知道它是做什么的,而并不关心“声卡”的工作原理和内部



图 1.1 算法与数据结构的关系
图 1.2 算法与数据结构应为 1:1 关系

结构，即不关心它是如何实现这些功能的。这种把具体实现细节的数据信息和操作方法在结构上隐藏起来，并将数据和处理数据的操作方法包装成自成一体的整体的做法，称为“数据封装”。外部的用户在严格的管理机制下只有通过有限条件的消息通路，即接口界面才能访问到隐藏在实体内的数据信息和操作方法，从而达到了“信息隐藏”的目的。无需知道封装单元内部是如何工作的就能使用它的思想称为“信息隐藏”。当然“信息隐藏”也具有保护的作用，可以防止外界的干扰破坏和误操作，如电话机壳对内部电路具有保护作用，但更重要的是隐藏了很多具体实现细节。

2. 对象 (Object)

对象是面向对象程序设计的核心概念。“对象”的概念并不神秘，它来源于生活。在现实生活中，其实我们每时每刻都在和对象打交道。例如，一本书、一台电脑、一部电话机，戴的手表、骑的自行车等。如果把问题抽象一下，会发现现实生活中的这些对象具有两个共同特点。第一，它们都具有自己的状态和外貌特征。例如，一部电话机有机型、大小、颜色等。第二，它们都具有自己的行为。例如，电话有通话、拨打、挂起等行为。

在面向对象程序设计中，抽象是最基本的原则思想之一。什么是抽象呢？抽象是对现实世界中的实体进行简化的描述，即使它模型化。从程序员的角度看，亦即是抓住编程者所关心的重要信息，而忽略掉一些不重要的细节部分。这是克服软件复杂性，将客观世界的模型在计算机中自然地表现出来的最好办法。在程序设计中，将对象的状态和外貌特征用数据来表示，称为对象的属性 (Properties)，而将对象的行为用对象中的程序代码来实现，称为对象的方法 (method)，并将这些数据和程序代码封装在一个程序实体内。例如：

```
struct Complex {
    double real , imag;    // C 语言中的结构体
};
// C++中的类，即 class 类型
```

```
class Complex {
private : (私有部分)
    double real , imag;
public : (公有部分)
    void set(double r , double i) // 成员函数
    {
        real = r;    imag = i;
        ...
    };
};
```

程序实体

```
void main( void )
```

```
{  
    Complex  a , b ;      // a、b 是 Complex 类的实例变量，即它的对象  
    a.set(4.0 , 6.0);     // 用成员函数的调用来实现消息发送  
    b.set(4.0 , 6.0);     // 同一条消息可发向多个对象  
    ...  
}
```

由此可知，C++ 引进了面向对象的思想，将 `struct` 类型引申成 `class` 类型，把数据和处理它的操作函数捆在一起，实现了数据封装和信息隐藏，因此，该类的对象是一个“状态”和“操作”的封装体。“状态”是由对象的数据结构的内容及其值定义的，方法是一系列的实现步骤，它由若干个“操作”组成。例如，一个人有很多信息，如姓名、年龄、身高、体重、文化程度、身份证号码，甚至还有体温、血压、脉搏次数等，但最重要的信息是姓名和身份证号码。这是人的重要属性。

1.1.2 面向对象程序设计的基本思想

面向对象程序设计的基本思想是在程序空间内，利用数据抽象的方法对客观世界中的实体进行描述，即将一种数据结构和操作该数据结构的方捆在一起，封装在一个程序实体内，从而实现数据封装和信息隐藏。亦即把数据结构隐藏在操作的后面，通过“操作”作为接口界面实现与外部的交流，即消息传递。对外部来讲，只知道“它是做什么的”，而不知道“它是如何做的”。

1.2 面向对象程序设计的要点

面向对象程序设计语言支持对象(Object)、类(Class)和继承(Inheritance)等要素，其要点分述如下。

1.2.1 抽象数据类型

客观世界是由各种对象组成的，任何事物都是对象，是某个类的实例(Instance)。在程序空间内，类是抽象数据类型 ADT(Abstract Data Type)的实现。

1. 归类

由于客观世界是复杂的，人们在研究它们时，为了克服复杂性，常根据不同的目的将客观事物加以区别，按照事物的状态和特性（即属性）、行为或用途分门别类，这就是归纳分类，简称“归类”。这种归类的结果便是逐步抽象的过程，最终可抽象出具有相同属性和相同行为的某一类对象，即找到这一类对象的抽象数据类型 ADT。于是在程序空间内，就可以编写一个类的程序代码。例如，对于各种各样的飞机，今天看见一架波音 747，它是一个对象，明天又看见一架图-154，它也是一个对象，如此等等。如果将这一类对象抽取它们的共同特性，便可构造一个类“Airplane(飞机)”。为了简化问题，假如编程的目的是进行科学普及，那么描述飞机的所有共有特性就非常简单，即：凡是飞机都能在空中飞行，具有改变飞行方向、控制飞行高度和速度的操作；飞机都具有机名、机型、飞行速度、高度、方向等数据，它们用来描述飞机的结构特性和状态等；而每架具体的飞机自然可作为此类“Airplane”的一个实例。所以该类的抽象数据类型 ADT 为：

飞机的属性

- 机名
- 机型
- 飞行高度
- 飞行方向
- 飞行速度

飞机的操作（方法）

- 改变飞行高度
- 改变飞行方向
- 改变飞行速度

2. ADT 的描述规范

ADT 是根据要达到的目的描述客观实体的抽象化模型, 它抽象出与目的相关的主要信息, 忽略掉一些次要信息, 构成一种数据结构, 它不仅给出了数据的组织形式, 还给出了处理这些数据的操作方法。面向对象程序设计首先对所研究的客观实体进行抽象, 并用 ADT 加以描述, ADT 规范化的格式为:

```
ADT     某种 ADT 的名称     is  
  
    Data  
        该 ADT 的数据组织形式  
  
    Operations  
        构造函数 (C++用构造函数实现初始化操作)  
            Initial Value : 用来初始化新创建对象的数据  
            Process : 初始化对象  
        操作 1  
            Input : 给定的输入值, 即用户传递给该 ADT 对象的输入值  
            Preconditions : 执行本操作时数据所必备的状态值  
            Process : 对数据的运算操作  
            Output : 返回给用户的数据  
            Postconditions : 执行本操作后数据的状态值  
        操作 2  
            ...  
        操作 n  
            ...  
  
end ADT     该 ADT 的名称
```

在对某种抽象数据类型进行描述时, 先要给这种 ADT 启用一个标识符作为它的名称, 其内包括 Data 和 Operations 两部分, 前者是对数据类型的描述; 后者是操作方法列表, 首当其冲的是初始化操作, 绝大多数 ADT 都具有初始化操作, 用于给该 ADT 新创建对象的数据赋初值。C++是用构造函数 (Constructor) 来实现初始化操作, 其后的操作方法因具体的应用目的而异。通常用 Input (输入) 来指定用户给定的输入值, 用 Preconditions (前提) 表示可执行本操作的前提条件是这些数据所必备的状态值, 用 Process (处理) 表示由该操作对数据所进行的运算操作, 用 Output (输出) 表示返回给用户的数据值, 用 Postconditions (结果) 表示执行本操作后数据内部状态值的改变。例如平面几何图形的一种——圆的抽象数据类型包括圆心和半径, 描述成规范化的 ADT 如下:

```
ADT     Circle     is
```

Data

用非负实数给出了圆心的 X、Y 坐标和半径

Operations

Constructor

Initial Value : 圆心的 X、Y 坐标和半径

Process : 给圆心的 X、Y 坐标和半径赋初值

Area

Input : 无

Preconditions : 无

Process : 计算圆的面积

Output : 返回圆面积的数值

Postconditions : 无

Circumference

Input : 无

Preconditions : 无

Process : 计算圆的周长

Output : 返回圆周长

Postconditions : 无

GetXCoordinate

Input : 无

Preconditions : 无

Process : 取圆心的 X 坐标值

Output : 返回圆心的 X 坐标值

Postconditions : 无

GetYCoordinate

Input : 无

Preconditions : 无

Process : 取圆心的 Y 坐标值

Output : 返回圆心的 Y 坐标值

Postconditions : 无

GetRadius

Input : 无

Preconditions : 无

Process : 取圆的半径值

Output : 返回圆的半径值

Postconditions : 无

...

end ADT Circle

今后我们在进行面向对象程序设计时,首先用这一规范化的 ADT 来描述研究对象,然后再进行程序设计,并且,为了简便起见省略掉那些无信息的项目,上例中对某些操作,如 Input、Preconditions 和 Postconditions 等项目均无信息,则不再写出,即只写出那些有信息的项目。

3. 类

C++用类(Class)来表示 ADT,在具体应用 Class 类型时,用该类的对象来存储和处理数据。那么,什么是类呢?类是面向对象程序设计封装的基本单元,它被当做一个样板,用来生产该类的所有对象,就像将“类”比做“饼干模具”用来生产“饼干”之类的“对象”一样。因此,一个类的所有对象都具有相同的数据结构,并且共享实现操作的程序代码。这就是说,在创建一个新对象时,该对象具有“类”定义中所描述的相同数据结构和操作它们的方法所对应的程序代码,而不必针对每个新创建对象的操作代码和数据结构重新编写一次程序,从而大大减轻了编程者的劳动强度。再加上无数软件技术人员对 OOP 语言系统完成了标准类库的设计工作,形成了“应用程序框架(Application Framework)”的运行机制,为面向对象程

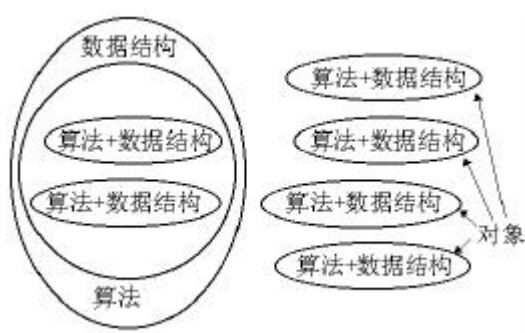


图 1.3 构成程序的对象

序设计提供了有力的支持。例如 Visual c++和 Visual Basic 已为 OOP 应用程序的编程者提供了 MFC(Microsoft Foundation Classes)标准类库和最新推出的 ATL(Active Template Library)活动模板库作为“应用程序框架”。已为编程者预先定义了许多如窗体、按钮、滚动条和对话框等对象的标准类。当编程者需要使用这些类的对象,例如要创建一个新的窗体时,只需用窗体类创建一个实例即可。既然现实世界是由各种对象组成,任何对象都具有一定的属性与操作,那么也就总能用数据结构和算法(操作该数据结构的方法)两者合一地对对象加以描述。这样,前述的程序定律就变成:

对象=(算法 + 数据结构)

程序=(对象 1 + 对象 2 +)

即程序就是许多对象在计算机中相继表现自己,而对象又是一个个程序实体。人们不再静止地看待数据结构,而把它看成是一个程序单位、一个程序分子或者一个对象的象征。它本身又包含算法和数据结构,即“对象”,正如图 1.3 所示的那样。

1.2.2 消息传递机制

OOP 采取消息(Message)传递机制作为对象之间相互通信和作用的惟一方式。

传统的结构化程序设计方法强调功能抽象和模块性，如图 1.4 所示，每个模块都是一个过程，其输入和输出对应着处理过程前、后的数据。因此，结构化程序设计方法将解题过程看做一系列的处理过程，由编程者编写调用“过程”的程序。对于相同的输入数据，经“过程”处理后，每次输出结果都相同，其编程繁琐，使用不灵活。

对象与传统的结构化程序设计方法中所说的数据有本质区别，对象本身具有很强的独立性，它不像结构化程序设计方法中的数据那样被动地等待对它执行某种操作，而是成为操作处理的主体，必须发消息（Message）请求对象执行它的某个（算法）操作，如图 1.5

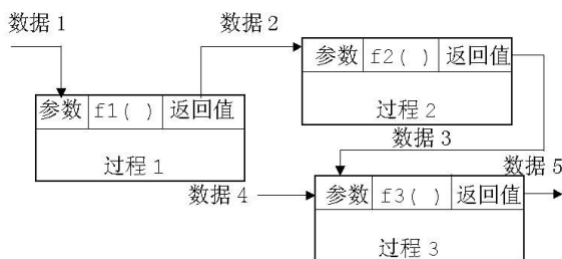


图 1.4 传统的结构化方法

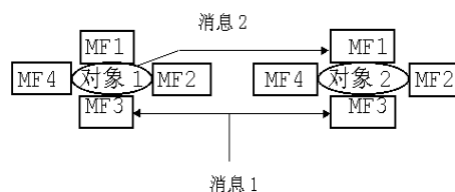


图 1.5 面向对象方法

所示，（算法）操作不是独立存在的实体，而是隶属于对象的，是对象的功能体现，其中 MF1（Method Function，方法函数）…… MF4 是隶属于对象的方法函数。每当需要改变对象的状态时，只能向对象发送消息来处理对象，消息带来了执行该操作的详细信息。C++ 是利用成员函数调用来实现消息发送的。在成员函数体内，处理对象的数据改变其状态，对象收到消息后，按消息提供的详细信息（包括目标对象、所请求的操作方法和参数）激活本对象内相匹配的操作即成员函数，并执行该操作，返回所需要的结果，以响应这条消息。并且，同一条消息可同时发至多个对象，并允许这些对象按照自身的状态加以响应。因此，与过程调用不同，同样的输入数据，可能因对象的状态不同而产生不同的输出结果。即对人弹琴，对牛弹琴，情况各异。

1.2.3 继承

继承是用来描述客观世界中实体间的一种关系，现实生活中到处都有继承的实例，如每个人从自己父母身上继承了人种、外貌和举止习惯等。在面向对象程序设计中 Class 类将按照“父类”（或称基类）、“子类”（或称派生类）的关系构成一个具有层次结构的系统。称为“类层次结构”。在这种“类层次结构”中，上层类所具有的性质和方法全部被下层类自动继承。因此，越是处在上层的类越具有普遍性和共性，越是处在下层的类则越细化、具体和专门化。

（1）继承将客观世界中一般和特殊的关系模型化成层次结构。如图 1.6 所示，小学生、中学生、大学生、研究生可归纳为“学生”；教授、讲师、助教可归纳为“教师”；学生、

工人、教师、农民……又可进一步用“人”加以概括,从而形成了类层次结构。一个类的上层可以有父类(或称基类),下层可以有子类(或称派生类)。例如,最上层的对象类“人”是“学生”类的基类,而“学生”又派生出“小学生”、“中学生”、“大学生”、“研究生”等派生类。

在一个类层次结构中,当每个类只允许有一个基类时,类的继承是单继承,即类的层次结构为树型结构。最上层的“人”为根结点,最下层的“小学生”、“中学生”、“大学生”等为叶结点。除根结点外,每个类都有它的基类,除了叶结点外,每个类都有它的派生类。

一个派生类可以从它的基类那里继承所有的数据和操作,并扩充自己的特殊数据和操作。基类抽象出共同的特征,而派生类表达差别。

当允许一个类有多个基类时,类的继承是多继承。图 1.6 中所示的“研究生”子类中,

既攻读研究生学位又从事教师工作的,就出现了既继承学生的属性和行为又继承教师的属性和行为。ISO/ANSI C++支持多继承,因此功能更强,使用更方便。

(2) 通过继承可增强程序代码的可重用性,代码的重用是利用继承基类的属性和方法来实现的。例如,由于研究生是学生的派生类,而学生又是人的派生类,因此,任一位研究生都应具有本类的属性:“专业”和“入学年月日”,还应当继承学生的属性:“学校名称”和“学号”,进而还应该继承人的属性:“身份证号码”和“姓名”。

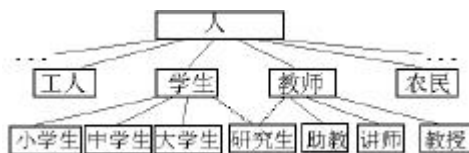


图 1.6 “人”的类层次结构图

(3) 在创建新的派生类时，只要指明新派生类是以哪一个已存在的类作为基类派生而来，便可自动继承基类的全部属性和方法。即新派生类只需定义新增加的属性和方法，不必再重复定义基类已有的属性和方法，这减少了程序的冗余信息，节省了存储空间。如图 1.7 所示，在创建新的派生类，例如“学生”时，只要指明新派生类是以“人”作为基类派生而来的，就可自动继承基类“人”的全部属性和方法。即新派生类“学生”不必再重写一遍基类“人”的“身份证号码”和“姓名”的程序代码，只需写出新增加的“学校名称”和“学号”等。同样只要指明研究生是“学生”的派生类，它自动继承了间接基类“人”的“身份证号码”和“姓名”，还自动继承了直接基类“学生”的“学校名称”和“学号”等，减少了程序的冗余信息，节省了存储空间。

(4) 修改和扩充程序时也不必修改原有的程序代码，只需增加一些新的代码，因而也无需知道原有的程序模块是如何实现的，从而极大地减少了软件的维护工作量，这是实现软件重用的重要机制。

顺便指出，软件产品的开发采用层次化结构进行模块化编程，将整个软件产品按功能划分成若干层 (layer)，每层实现一个确定的功能集，构成一层层程序模块，上一层模块可调用下层模块中的各种服务功能来完成本层所规定的功能。这种结构化、模块化过程，对于分析和综合软件产品，逐步由浅入深地学习和理解软件产品，在现有产品的基础上开发出新的、更强大功能的产品，具有积极的促进作用。面向对象方法及其理论的应用更能方便地完成这种层次化结构的实施。

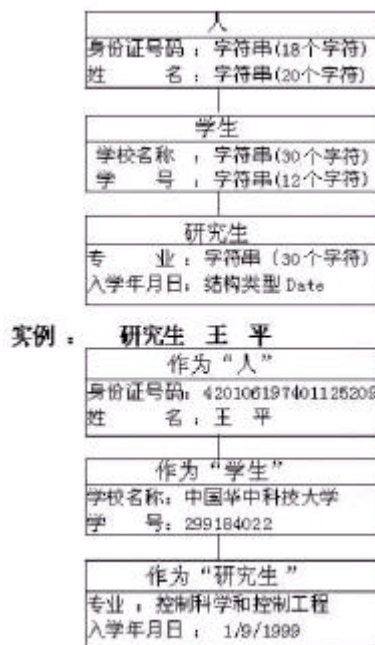


图 1.7 类的继承

1.3 C++程序结构的特点

为了理解 C++ 程序的特点，先看一个小的示例程序。

例 1.1 采用 C++ 流库的示例程序。

```

#include    < iostream >                // 使用 C++ 新标准的流库
using namespace std;                    // 将 std 标准名空间合并到当前名空间

void main( void )
{
    double x , y;
    cout << "请输入一个实数 : " ;      // 向 CRT 屏幕输出提示信息

```

```

cin >> x;                // 变量 x 接受键盘输入的实数
cout << "请输入另一个实数 : " ;    // 向 CRT 屏幕输出提示信息
cin >> y;                // 变量 y 接受键盘输入的另一个实数

double z = x + y ;
cout << "x + y = " << z << endl ;
}

```

下面结合示范程序分述 C++ 的主要组成部分。

1.3.1 标识符

标识符是由编程者定义的单词，由下划线_、英文大小写字母 (A ~ Z, a ~ z) 和数字 (0 ~ 9) 等字符有序排列组成的字符串，该单词不能以数字字符开头，即第一个字符必须是字母或下划线，中间不能有空格符。可用它来命名程序中一些实体，这些实体是函数名、常量名、变量名、类名、对象名、成员名、语句标号名等，在使用这些实体之前，必须用说明语句加以说明，通过说明将标识符引入到程序，标识符只能在称为“作用域”的程序区域内使用，例如程序中的下列语句都是说明语句：

```

double x , y;
double z = x + y;

```

定义标识符时应注意如下几点：

- (1) 标识符的长度（组成标识符的字符个数）是任意的。ISO/ANSI C 标准是 1~31 个字符，而 ISO/ANSI C++ 可到任意长度。
- (2) 标识符中的大小写字母是有区别的。例如：Max 和 max 是两个标识符。
- (3) 在实际应用中，建议尽量使用有意义的单词作标识符，做到“见名知义”。
- (4) 标识符的启用不要与语言系统的关键字相同，也不要与 C++ 标准类库和 C 语言标准函数库中的函数名、类名和对象名相同。例如示范程序中的 void、double 为语言系统的保留关键字，而 cout、cin 为 C++ 输入/输出流库的标准流对象名。

以下列举的为应避免使用的语言系统关键字。

asm	do	inline	short	typeid
auto	double	int	signed	typename
bool	dynamic_cast	long	sizeof	union
break	else	mutable	static	unsigned
case	enum	namespace	static_cast	using
catch	explicit	new	struct	virtual
char	extern	operator	switch	void
class	false	private	template	volatile
const	float	protected	this	wchar_t

const_cast	for	public	throw	while
continue	friend	register	true	
default	goto	reinterpret_cast	try	
delete	if	return	typedef	

1.3.2 预处理语句

C 和 C++ 的源程序开头经常出现以 “#” 开头的语句，它们是预处理语句，这种语句结尾没有分号，它是 C++ 源程序进入编译操作前，所预先进行的一种处理操作。例如，示范程序开头出现的文件包含命令是常用的一种，还有宏定义命令也被经常使用。详细情况请参阅文献[25]。

1.3.3 输入/输出流操作语句

C++ 源程序总少不了输入 (Input) / 输出 (Output) 操作语句，几乎每个源程序都要用到。扩展名为 .cpp 的源文件大多数都使用 C++ 的标准输入/输出流库，在第 7 章将作详细讨论，这里先从应用的角度作一简单介绍。

输入运算符 “>>” 和输出运算符 “<<”：在 C++ 运算符集合中，原来所包含的 “>>” 右移运算符和 “<<” 左移运算符，由于它们有箭头 “指向” 的含义，故适用于数据从源点向终点的流操作。所谓 “流操作 (stream)” 是指数据从源点搬运到终点的流动操作。在 C++ 新标准头文件 “iostream” 和老头文件 “iostream.h” 中给右移运算符 “>>” 和左移运算符 “<<” 赋予了新的含义和功能，用来作为输入/输出运算符。

左移运算符 <<：在 C++ 的标准输入/输出流库中给它赋予了新的操作功能，即重载成输出运算符，或者称为插入符 (Inserting、Putting)，它的书写格式为：

数据的流动		
终点	←	源点
cout (CRT 显示器) << 程序中的对象名;		

例如，示范程序中：

```
cout << "请输入一个实数：" ;
cin >> x;
cout << "请输入另一个实数：" ;
cin >> y;
double z = x + y ;
cout << "x + y =" << z << endl ;
```

表示将程序中对象的数据输出到显示器屏幕上。

若源程序中的对象是一个字符串常量,即用一对双引号包围的字符串,则输出运算符将按字符串常量的原样输出到显示器屏幕上,并能识别特殊字符的换码序列\n、\t、\v、\a等,这与C语言中printf()标准函数的使用一样。例如示范程序中执行完cin>>x>>y;希望显示一下x和y的值,校核键盘输入的值,可加入如下一条语句:

```
cout<< " x = " << x << "\t y = " << y << endl;
```

其中, 表示空格符,endl(最后一个字符是英文小写字母“l”,而不是数字字符“1”)与\n的作用类同,即回车、换新行。

允许多个输出操作组合成一条语句,输出运算符的结合规则是从左到右,即先输出“ x = ”,然后输出x的值,再输出“ \t y = ”……

输出运算符能自动识别其运算量的数据类型,不需要以%开头的转换说明符,使用方便,不易出错。例如,若传统的C语言风格编写的源程序中写有这样的语句:

```
double x = 12.6 , y = 16.8;
...    变量与转换字符不匹配
```

```
printf("x = %d , y = %f\n", x, y);
```

则变量x是double型,与%后的转换字符不匹配,而printf()函数又没有检查机制,因此编译、链接都能顺利通过,而输出的x值是随机值,不是正确结果。但C++的输出运算符能自动识别运算量的数据类型而不会发生这种出错情况,即:

```
cout<< " x = " << x << " , y = " << y << endl;
```

(3) 右移运算符>>:与输出运算符一样,标准输入/输出流库中将它重载成输入运算符,或称提取符(extracting、getting)。它的书写格式为:

数据的流动		
源点	→	终点
cin (键盘)	>>	程序中的变量名 (或对象名);

例如,示范程序中

```
double x , y;
cin >> x;
```

表示读取从键盘敲入的数据,传递给程序中的变量x。

由于从键盘进行输入操作时,应允许用户敲错,即要处理输入操作中的非法字符。为此把输入的数据类型分为整型、浮点型和字符串等3种。

对于整型和浮点型的输入,输入运算符跳过空白符(它包括空格符“Space”、换行符“CR”、水平制表符“Tab”等),然后读取对应于输入变量类型的值,操作者按压回车键则系统将敲入的数字字符序列转换成相应的数据类型后赋给作为右值的变量。读输入的过程一直进行到出现一个不合法部分时立即停止。

例 1.2 使用输入运算符>>的示范程序。

```

#include    < iostream >           // 使用 C++新标准的流库
using namespace std;              // 将 std 标准名空间合并到当前名空间

void main( void )
{   cout << "请输入一个整数 : ";
    int i;
    cin >> i;
    cout << "请再输入一个实数 : ";
    float f;
    cin >> f;
    cout << "输入的两个数是 " << i << " 和 " << f << endl;
}

```

• 正确的输入方法：

请输入一个整数 : 12(CR)
 请再输入一个实数 : 15.23(CR)
 输入的两个数是 12 和 15.23。

• 错误的输入之一：

请输入一个整数 : 12b33(CR)
 请再输入一个实数 : xxxxxx(CR)
 输入的两个数是 12 和 0。

由此可知，其结果是将值 12 和 0 分别赋给变量 i 和 f，这是因为 b 字符是非法的，因此第 1 次输入操作 (cin >> i;) 没有接收字符 b 就结束，输入操作在字符 b 的位置上停止处理。第 2 次输入操作 (cin >> f;) 遇到的第一个字符还是 b，因此输入操作立即结束，变量 f 所得的值为零。

• 错误的输入之二：

请输入一个整数 : 12.23(CR)
 请再输入一个实数 : xxxxxx(CR)
 输入的两个数是 12 和 0.23。

其结果是将值 12 和 0.23 分别赋给变量 i 和 f，这是因为对于整型变量 i，当读到小数点时为非法，第 1 次输入操作 (cin >> i;) 没有接收它就结束，输入操作在小数点位置上停止处理。第 2 次输入操作 (cin >> f;) 遇到的第一个字符是小数点，对变量 f 又是合法的，一直读到 .23 遇到回车键停止，f 读得 0.23。

与输出运算符一样，输入运算符也能自动识别运算量的数据类型，不需使用以%为开头的转换说明符。如：

```

int i;
printf( "Input i value : " );

```

```
scanf( "%d", &i );           // 必须在接收变量 i 前面加一个取地址运算符&
```

↓

```
int i;
cout << "Input i value : ";
cin >> i;           // 不需要在接收变量 i 前面加一个取地址运算符&
```

由此可知, C 语言中的 `printf()` 和 `scanf()` 这两个标准函数以及 `stdio.h` 中所声明的相关输入/输出操作的标准函数都还不够完善, 而在编写格式上又特别容易出错, 且有些错误又可以顺利通过编译和链接操作, 但在程序运行时却不能完成编程者所希望的输入/输出操作, 如上例中的第 3 行, 调用 `scanf()` 函数将键盘敲入的一个整数型数值由变量 `i` 保存, 但是若写成如下错误形式:

```
scanf("%d", i);           // 接收变量 i 前面缺少一个取地址运算符&
scanf("%f", &i);         // 接收变量 i 对应的转换字符应该是%d而不是%f
```

在编译和链接操作过程中系统检查不出任何错误而顺利通过, 但程序运行时执行该语句变量 `i` 将接收不到键盘敲入的数值, 这是令初学者十分头痛的程序隐患。因此, 建议编程者即使编写模块化程序 (不采用 OOP 的编程思想), 源文件的扩展名也不要使用 “.c” 而采用 “.cpp”, 即采用遵循 ISO/ANSI C++ 新标准的编程格式, 即便是以前开发的遵循 ISO/ANSI C 老标准的 C 语言程序, 也把扩展名 “.c” 改成 “.cpp” 再将源程序按新标准进行修改。例如, 尽量用 `<iostream>` 取代 `<stdio.h>`, 如尽量使用 `cout` 和 `cin` 语句代替 `printf()` 和 `scanf()` 这两个标准函数的调用语句, 以确保所编写程序的安全性和可扩充性 (`cout` 和 `cin` 语句可扩充用来对 `class` 类型的对象进行输入/输出操作), 详见文献 [26]。

允许多个输入操作组合成一条语句。输入运算符的结合规则也是从左至右, 例如在示范程序中, 也可以写成 “`cin >> x >> y;`”, 即先输入 `x` 的值, 再输入 `y` 的值。但实际应用时为了获得良好的人机界面, 通常是一条语句只输入处理一个变量, 并用输出提示字符串信息写成:

```
cout << "请输入一个实数 : ";
cin >> x;
cout << "请输入另一个实数 : ";
cin >> y;
double z = x + y ;
cout << "x + y = " << z << endl ;
```

1.3.4 函数和语句

C++ 的源程序是由若干个函数组成的, 函数之间是相互独立的, 在编程时应注意如下 3 个问题。

C++的控制台源程序 (console source program), 其输出结果即用 cout 语句输出在显示器上的结果, 显示在 MS-DOS 状态下的黑色底面窗口上, 该窗口称为“输出结果窗口”(详见[文献 25]的 1.2 节), 必须有一个并且只能有一个主函数 main(), 它是该程序的执行起点。

其他函数只能通过主函数调用, 或者由被主函数调用的函数来调用。函数在调用前, 首先要定义好, 或者事先用函数原型加以声明。使用语言系统提供的标准函数时, 需要将包含该函数的头文件利用 include 语句嵌入到该程序中。

C++语句以分号为结束。任何一个表达式加上一个分号可以组成一条语句, 只有分号而没有表达式的语句为空语句。

表达式; // 表达式语句
; // 空语句

C++语句可分为说明语句、表达式语句 (包括空语句)、流程控制语句 (条件、循环、开关) 和复合语句等。函数体内没有任何语句的被称为空函数。例如：

```
void func( ){ 空 }
```

复合语句：用一对大括号 { } 将多个相关联的语句括起来便构成复合语句，或称为块 (block) 语句。

1.3.5 标准名空间和其他组成部分

1. 标准名空间 (std namespace)

Visual C++遵循 ISO/ANSI C++标准的技术规范, 将 C++标准程序库中的所有实体名字即标识符名 (除不遵循作用域规则的宏指令名外) 都声明在名称为“std”的名空间内, 即这些标识符名都是该名空间的成员, std 名空间称为标准名空间 (standards namespace)。所谓名空间是指一个标识符在此程序范围内必须是惟一的区域, 从而确保了用户所启用的标识符名在任何时候不会与 C++标准程序库中的所有标识符名发生冲突, 详细情况请参阅 3.8.4 节。但是, 目前世界上还存在无数个仍然还采用已使用多年的 C++老标准程序库的程序代码, 如引入 <iostream.h>、<string.h>和<math.h>等头文件的程序, 为了解决与这些程序的兼容性问题, C++标准委员会设计出如下能快捷进行转换的新头文件名：

将原有的 C++系统的头文件扩展名 “.h” 去掉即可, 即 <iostream.h>转换成新头文件名就变成 <iostream>, <fstream.h>就变成 <fstream>, 而 <strstream.h>就变成 <strstream>..., 如此类推。Visual C++ V6.0 所包含的标准头文件如表 1.1 所示。

表 1.1 Visual C++ V6.0 所包含的标准头文件

序号	原文件名	新文件名	内 容
1	< algorithm.h >	< algorithm >	定义标准模板库 STL 中实现公用算法的许多函数模板
2	< bitset.h >	< bitset >	定义一个管理二进制位集合的类模板
3	< complex.h >	< complex >	定义一个支持复数运算的类模板
4	< deque.h >	< deque >	定义一个实现双端队列的类模板
5	< exception.h >	< exception >	定义管理异常处理的一些函数
6	< fstream.h >	< fstream >	定义一些管理和操作外部文件的输入/输出流的类模板
7	<functional.h>	<functional>	定义标准模板库 STL 的一些模板, 它们帮助创建在 < algorithm >和< numeric >头文件中定义的模板
8	< iomanip.h >	< iomanip >	声明一些读取一个参数的输入/输出流“操作符”函数
9	< ios.h >	< ios >	声明根类 ios_base 及其派生类模板, 它是许多输入/输出流类的基础
10	< iosfwd.h >	< iosfwd >	声明几个输入/输出流操作的类模板, 在它们定义之前
11	< iostream.h >	< iostream >	声明管理和操作标准流的输入/输出流类的对象
12	< istream.h >	< istream >	定义实现读取操作的类模板
13	< iterator.h >	< iterator >	定义标准模板库 STL 的一些模板, 它们帮助定义和操纵迭代器 (iterator)
14	< limits.h>	< limits >	测试数字类型的属性
15	< list.h >	< list >	定义标准模板库 STL 的一个实现列表容器的类模板
16	< locale.h >	< locale >	定义一些类和模板, 它们在输入/输出流类中控制局部的特殊行为
17	< map.h >	< map >	定义标准模板库 STL 用来实现联合容器的类模板
18	< memory.h >	< memory >	定义标准模板库 STL 的几个模板, 它们对各种容器类进行分配和释放存储空间的操作
19	< numeric.h >	< numeric >	定义标准模板库 STL 的几个实现公用数字函数的模板

(续表)

序号	原文件名	新文件名	内 容
20	< ostream.h >	< ostream >	定义实现写入操作的类模板
21	< queue.h >	< queue >	定义标准模板库 STL 的一个实现队列容器的类模板
22	< set.h >	< set >	定义标准模板库 STL 的用单个元素实现联合容器的类模板
23	< sstream.h >	< sstream >	定义一些管理和操作字符串容器类的输入/输出流操作的类模板, 它们支持存放在所分配的对象数组中的序列化对象, 且这些对象与类模板 basic_string 的对象都能方便地互相转换

序号	原文件名	新文件名	内 容
24	< stack.h >	< stack >	定义标准模板库 STL 的一个实现堆栈容器的类模板
25	< stdexcept.h >	< stdexcept >	定义一些公用的、报告异常信息的类
26	< streambuf.h >	< streambuf >	定义一些缓冲器流操作的类模板
27	< string.h >	< string >	定义一个实现字符串容器的类模板
28	<strstream.h>	<strstream>	定义一些管理和操作内存中 char 型字符串序列的输入/输出流类
29	< utility.h >	< utility >	定义标准模板库 STL 的一些常用模板
30	< valarray.h >	< valarray >	定义一些支持数值型数组的类和模板
31	< vector.h >	< vector >	定义标准模板库 STL 的一个实现向量容器的类模板

C++标准程序库还包含 C 语言标准函数库的 18 个头文件如表 1.2 所示。对于 C 语言标准函数库的头文件，去掉头文件中的扩展名“.h”后，再在头文件名前面加一个英文小写字母 c。如表 1.2 中“原头文件名”和“新头文件名”两栏所示，原头文件<ctype.h>

表 1.2 C 语言标准函数库的头文件

序号	原头文件名	新头文件名	内 容
1	< assert.h >	< cassert >	定义强制调试宏指令
2	< ctype.h >	< cctype >	字符分类和字符转换的宏指令
3	< errno.h >	< cerrno >	定义错误码的助记符号
4	< float.h >	< cfloat >	测试浮点类型的属性
5	< iso646.h >	< ciso646 >	以 ISO 646 标准字符集编程
6	< limits.h >	< climits >	包含环境变量、限制编译时间和整数型的范围
7	< locale.h >	< clocale >	提供国家和语言特殊信息的标准函数
8	< math.h >	< cmath >	常用数学标准函数
9	< setjmp.h >	< csetjmp >	定义用于执行非局部 goto 语句函数的数据类型

(续表)

序号	原头文件名	新头文件名	内 容
10	< signal.h >	< csignal >	管理各种异常情况的标准函数和常量
11	< stdarg.h >	< cstdarg >	定义用来读入函数实参的宏指令, 此类函数可接收可变数量的实参
12	< stddef.h >	< cstddef >	定义一些公用的数据类型和宏指令
13	< stdio.h >	< cstdio >	定义标准输入/输出软件包所需的数据类型和宏指令
14	< stdlib.h >	< cstdlib >	说明一些公用、转换、搜索和分类等操作的函数
15	< string.h >	< cstring >	说明字符串操作及其内存处理操作的函数
16	< time.h >	< ctime >	定义各种时间和日期格式间变换的函数
17	< wchar.h >	< cwchar >	定义广义流和几种字符串的操作函数
18	< wctype.h >	< cwctype >	定义大范围字符分类和字符转换的宏指令

就变成了< cctype >, < math.h >就变成了< cmath >, < stdio.h >就变成了< cstdio >..., 如此类推。

为了遵循国际标准委员会所制订的 C++新标准, 编程者应尽可能采用新标准的编写格式, 即应该使用“#include”预处理语句引入不带扩展名“.h”的新标准头文件, 有如下两种编程格式:

用“std::”指明标准名空间中的标识符。如:

```
#include    < iostream >
void main( void )
{
    std::cout << "请输入一个整数 : ";
    int i;    std::cin >> i;
    ...
}
```

由于 cout 和 cin 均在标准头文件 iostream 中声明为输入/输出流类的标准流对象, 而它们又位于 std 标准名空间内, 因此, 必须用“std::”加以修饰。

在用“#include”预处理语句引入了标准头文件 iostream 后, 接着写如下一条语句把 std 名空间内的所有成员都合并到当前名空间(显示在 Visual C++集成开发环境的编辑窗口上的程序所在的名空间)内:

```
using namespace std;
```

这样一来, std 名空间内的所有成员名都可以直接使用, 而不必对每个成员用前缀“std::”加以修饰, 请参看例 1.2 的程序代码。

2. 标准名空间的组织结构

C++的标准名空间的组织结构基本情况如下:

带扩展名“.h”的老标准头文件内容已不再属于 C++标准, 但考虑到与 C++老程

序兼容性问题在 C++ 语言系统中仍然存在，但这些头文件的内容已不处于 std 标准名空间内。

不带扩展名 “.h” 的新标准头文件如 < iostream > 包含对应的老头文件如 < iostream.h > 的功能，只是在标准程序库中的某些实现细节上做了一些小的修改。

C 语言标准函数库的头文件如 < stdio.h > 在 C++ 语言系统中仍然可以使用，它们并没有放在 std 标准名空间内。

C 语言标准函数库的头文件如 < string.h > 在 C++ 新标准中也有对应的头文件名如 < cstring >，它们提供与老头文件相同的内容，但全部放在 std 标准名空间内。

因此，目前在 C++ 编程领域同时存在着新的和老的两种引入头文件的编程格式，下面以例 1.2 的程序代码为例，其老的编程格式为：

```
#include    < iostream.h >
void  main( void )
{   cout << "请输入一个整数：" ;
    int i;   cin >> i;
        ...
}
```

如前所述，编程者应尽可能采用新标准的编写格式，但又要考虑与 C++ 老程序的兼容性问题，这只需要掌握如何把这些老编程格式修改成新标准的编程格式，即将所有 C++ 老标准头文件名中的扩展名 “.h” 去掉，对于 C 语言标准函数库的头文件名还要在其前面加上一个英文小写字母 c，接着在所有 #include 预处理语句后面加上一条 “using

```
namespace    std;”
```

语句即可。本书绝大部分例程均采用新标准的编写格式，但也留有少数的例程采用老编程格式以便读者能读懂一些 C++ 老程序，并提供给读者作为 “把 C++ 老程序修改成新标准的编写格式” 的练习题。

3. 其他组成部分

C++ 源程序除了以上部分外，还有其他一些组成部分，例如符号常量和注释信息也是程序的一部分。符号常量在讲解 C++ 基础部分的文献 [25] 中已作了详细介绍。

C++ 采用两种注释方法：

(1) ISO/ANSI C 标准使用 “/*” 和 “*/” 括起来进行注释，在 “/*” 和 “*/” 之间的所有字符都作为注释处理，在编译时被系统忽略。它可放在程序区域内的任何位置，并可占用多行。

(2) ISO/ANSI C++ 增加了单行注释，使用 “//”，从 “//” 开始，直到它所在的行尾，所有字符都作为注释处理。熟悉 ISO/ANSI C 老标准的编程者应特别注意，要尽量使用 C++ 风格的注释形式。例如，有这样一段源代码：

```
if( a > b ) {  
    // int x = a;      // 交换 a 和 b  
    // a = b;  
    // b = x;  
}
```

这种在一行内两次用 C++ 的注释形式将不会产生任何问题，但是，若采用 C 语言风格的注释形式，如：

```
if( a > b ) {  
    /* int x = a;      /* 交换 a 和 b */  
    a = b;  
    b = x;  
    */  
}
```

构成嵌套式注释形式，当碰到第 2 行的注释结束符“*/”时，则导致注释区段过早结束，这种嵌套式注释形式通常不能通过编译。

另外，在一些用 C 语言编写的老程序中，经常采用#define 预处理语句来定义符号常量，若采用 C++ 注释形式，例如：

```
#define ARRAY_SIZE      1000      // 数组的大小
```

预处理器会将尾端的 C++ 注释作为宏定义的一部分，显然会给程序带来隐患。当然，在 C++ 中，已不用#define 预处理语句来定义符号常量，这将在 2.2 节中详细讨论。

1.4 Visual C++ V6.0 使用方法

Visual C++ V6.0 是美国 MicroSoft 公司按 ISO/ANSI C++ (ISO 14882) 标准开发的，而且是目前国内外广为流行的一种 C++ 语言系统 (单机版)，它的功能较强，需要运行在 Windows 95 以上版本和 Windows NT V3.0 以上版本，它分为“标准版”、“专业版”和“企业版”，即便是“标准版”也需要占用 100 多 MB 的磁盘空间。因篇幅有限，本节针对初学者，从实用的角度出发，并结合 Windows 操作系统的有关部分，重点讲解开发 C++ 源程序所必须掌握的基本操作。

顺便指出，在 Visual C++ 的集成开发环境中也可以开发 C 语言的源程序，这种源程序的编辑文件名应为“文件名.C”，它是采用遵循 ISO/ANSI C 老标准的系统对其进行编译、链接操作。而对于 C++ 的源程序，其编辑文件名应为“文件名.CPP”，却采用遵循 ISO/ANSI C++ (ISO 14882) 标准的系统对其进行编译、链接操作。正如 1.1 节中所述，本书从实用的角度出发，书中所提到的“C 语言”的语法和编写规则 (特别声明的除外) 均是在 ISO/ANSI C++ 标准中继续保留的“C 语言”部分，即作为面向过程的模块化

基础部分来讲授，扬弃了 ISO/ANSI C 老标准及其 C 语言老版本中与 ISO/ANSI C++ 新标准不一致的内容。为此，书中的所有例程和练习题均采用“文件名.CPP”为编辑文件名。

1.4.1 源程序的编辑、存储和建立

首先必须特别指出，由于在 Visual C++ 的集成开发环境中开发程序，将产生容量较大的各种结果文件，即便是一个很简单的程序，其整个容量也将超过 1.44MB，而在编译、链接时由系统依照源程序的位置将其结果文件自动地存放在与源程序相同的位置上，因此，所有源程序不能放在 3.5 寸的软盘内而只能放在硬盘中，最好为本书所有源程序创建一个目录，如“VC6User”，然后，再以章节建立一系列的子目录，如“ch1”、“ch2”、“ch3”…。

启动系统后，在显示器屏幕上将出现“Microsoft Developer Studio”主窗口，如图 1.8 所示，按照“File(主菜单)/New(子菜单)”的操作路径进入到“New”对话框，如图 1.9 所示，在“New”对话框内，再选择多页面控件中的“File”页面，接着选择“C++ Source File”子菜单项，用鼠标箭头指在此项双击左键，则进入到“编辑窗口”。

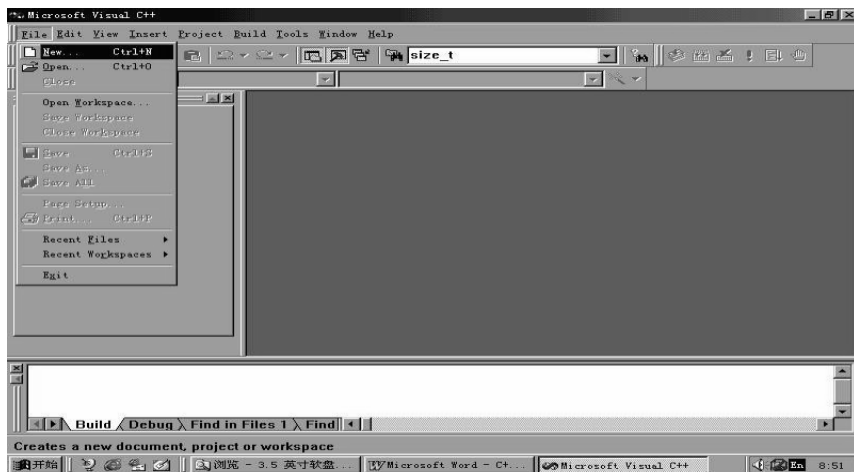


图 1.8 建立一个 C++ 源程序的操作

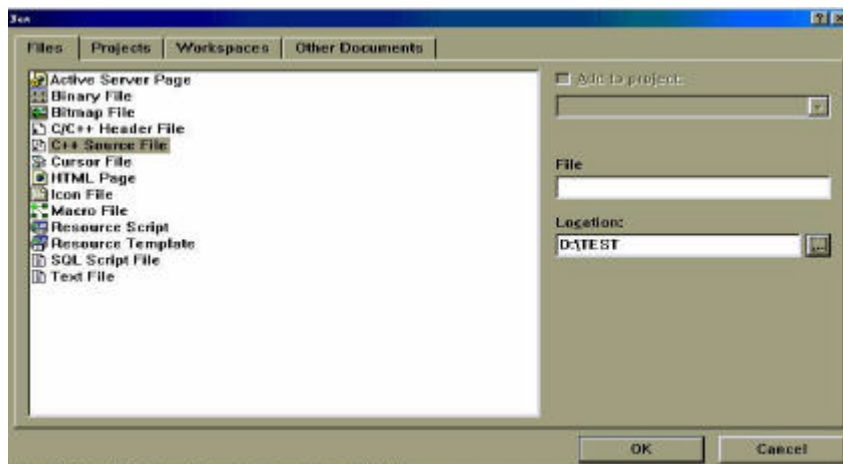


图 1.9 在“New”对话框内进入“编辑窗口”

1. 源程序的编辑

在“编辑窗口”内，键入用户编写的 FileName.cpp(或 FileName.c)的具体内容，完全采用 Windows 的编辑操作方法。选择如下几个特别有用的加以介绍：

在编辑窗口上设置一个块对象，其方法是用鼠标或用光标键将光标移动到设置块的起始处，接着一直按下鼠标左键拖动，移动鼠标箭头到这段块的结尾处，松开鼠标左键，如图 1.10 所示，鼠标箭头移动时所经过的部分都由白底黑字变成黑底白字，这段区域就是所设置的块。

将块对象的信息复制到“剪贴板”上，剪贴板是 Windows 操作系统开辟的一个存放信息的专用内存空间。最简便的操作方法是按下“Ctrl + C”(“+”的意思表示同时按下)，其中“C”是 Copy(复制)的首字母，它特别容易记忆，即将设置块内的信息复制到剪贴板上，例如，对于图 1.10 所示的设置块，实际上是把两条语句复制到剪贴板上。

把剪贴板上的信息粘贴到编辑窗口内的指定位置。将光标移动到指定位置的起始点处，接着按下“Ctrl + V”，即把存放在剪贴板上的信息粘贴到指定位置，由于剪贴板的桥梁作用，粘贴操作既可在窗口内，也可在窗口之间进行，显然，这可把整块程序裁剪下来复制到任何地方，特别是将“帮助”项内的例程整块复制到编程者所编写的源程序中时非常有用。正如图 1.11 所示，把剪贴板上的一条语句复制一次后得到了同样的一条语句，将其中的“请输入一个整数：”改成“请再输入一个实数：”，即在“请”和“输入”间插入一个“再”字，再将“整”字改成“实”字即可，则可以非常快地完成例 1.2 的源程序输入操作。



图 1.10 在编辑窗口上设置一个块对象



图 1.11 粘贴剪贴板的信息

删除整个设置块的信息。当块对象设置好后，只要按“Delete”或“Backspace”就把整个设置块的信息删除掉。若操作者希望恢复它，只需按“Alt + Backspace”即可。

裁剪整个设置块的信息。该操作是把设置块的信息从当前窗口删除掉，接着将它复制到剪贴板上保存起来，以便进一步向任何地方复制，这只要按“Shift + Delete”或“Ctrl + X”即可实现。显然，它与粘贴操作配合使用，可以实现整块的搬运、复制和删除等操作，且既可以在一个窗口内，也可以在窗口之间进行。

在主菜单下面的工具条内含有“Cut (Ctrl + X, 裁剪)”、“Copy (Ctrl + C, 复制)”、“Paste (Ctrl + V, 粘贴)”等 3 个选择按钮，当块对象设置好后，直接用鼠标箭头指向某选择按钮，然后按下鼠标左键即可代替键盘按键进行快捷方式的裁剪、复制和删除等操作。

2. 存储和建立源程序

其操作路径为“File(主菜单)/Save(子菜单)或 Ctrl+S(热键)”，在显示器屏幕上将出现“Save As(保存为:)”对话框，在该对话框中的“File Name 文件名(N 热键)”框内，键入磁盘文件名，例如：“p0101.cpp”，接着按回车键或点击 OK 按钮，则将键入的源程序以“p0101.cpp”为编辑文件名存入磁盘。

1.4.2 编译、链接和运行源程序

1. 检查或修改编译、链接路径

对源程序进行编译、链接前，有时最好先检查一下编译、链接路径，特别是为一台计算机刚建立了 Visual C++ 的集成开发环境，第 1 次进行编译、链接时更应该检查一下。其操作路径是“Tools(主菜单)/Options(子菜单)”，进入到 Options 对话框，在 Options 对话框内，再选择多页面控件中的 Directories 页面，用鼠标点开该页面的“Show Directories for:”选择控件，用户可根据需要选择“Executable files”、“Include files”、“Library files”和“Source files”，如图 1.12 所示。用户若选择“Include files”检查头文件，则在 Directories 框内，出现如下信息：

```
D:\Microsoft Visual Studio\VC98\INCLUDE
D:\Microsoft Visual Studio\MFC\INCLUDE
D:\Microsoft Visual Studio\ATL\INCLUDE
```

(由用户写入需要新增加的头文件)

用户还可以在 Directories 框内新增加一些头文件, 以满足开发实际程序的需要。



图 1.12 检查或修改编译、链接路径的操作

2. 编译、链接单个源文件的程序

16 位微型计算机的 C++ 语言处理系统如 Borland C++ 以及以前的 C++ 系统如 Turbo C++ 等, 在编译、链接操作时是由 CPU 直接去访问外部存储器——硬盘机进行必要的信息交换。由于外部设备硬盘机与 Memory (内部存储器, 简称“内存”) 相比执行速度要慢得多, 为了提高编译、链接操作的执行速度, Microsoft 公司提出了“创建工作区 (Building Workspace)”的新技术, 即开始编译、链接操作前, 为正要进行编译、链接的源文件在内存中创建一个“工作区”如图 1.13 所示, 将编译、链接操作时所需要的所有信息都读取到该“工作区”内, 这样一来, 由于“工作区”是内存的一个区段, 操作时 CPU 直接与内存交换信息, 而不必去访问硬盘机, 从而大幅度提高了编译、链接操作的执行速度, 目前此项技术已被广泛采用。

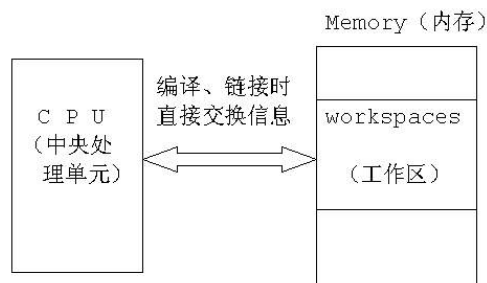


图 1.13 “创建工作区”技术

操作方法是沿着“Build (主菜单) / Compile filename.cpp (子菜单) 或者用 Ctrl+F7 (热键, 即把 Ctrl 键和 F7 键同时按下, 则直接一步选取到该子菜单)”的操作路径用鼠标左键点击, 系统将弹出一个如图 1.14 所示的选择型对话框, 询问操作者是否

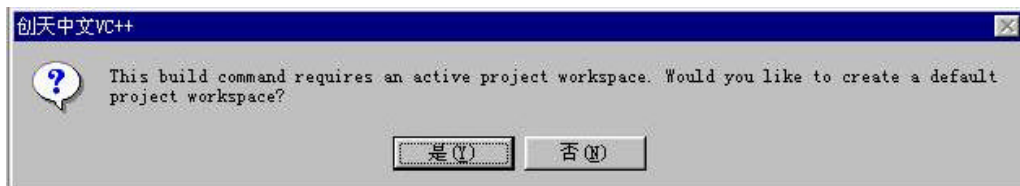


图 1.14 “询问创建工作区”的选择型对话框

要创建一个默认的“工程项目工作区”。如前所述，操作者理所当然地选择“是(Y)”，即用鼠标左键点击左边的“是(Y)”按钮，这时编译系统对当前源程序 filename.cpp (即放在编辑窗口上的源程序) 进行编译。在编译过程中，将所发现的编译错误 (Compiler Error) 显示在屏幕最下方的 Build 窗口上，并给出该错误所在的行号、编译错误码和错误性质的简洁提示。本系统的编译错误码采用“C5555”的格式表示，其中“5”代表一位十进制数码，在 Build 窗口的错误性质简洁提示行中将给出编译错误码，编程者可根据这些错误信息来修改源程序。若还需进一步了解错误信息和改错的对策，可沿着“Help (主菜单) / Contents (子菜单) 或 Search (子菜单)”操作路径用鼠标点击，将启动 MSDN (Microsoft Developer Network) 库，则弹出“MSDN Library Visual Studio 6.0”窗口，简称 MSDN 窗口，在该窗口的“MultiplePage”控件上点击“搜索”页面，在该页面的“输入要查找的单词 (F):”框内键入编译错误码，例如 C2660 (CR)，则在 MSDN 窗口上显示详细的信息，如图 1.15 所示，编程者仔细阅读后关掉或最小化 MSDN 窗口才能退回到原来的主窗口。往往一个错误会引起多行错误信息，因此一般是针对第 1

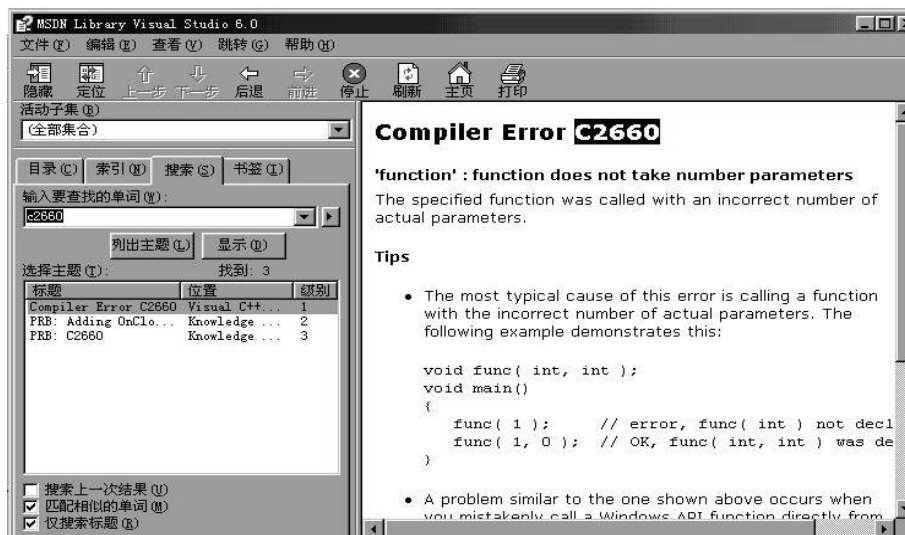


图 1.15 在 MSDN 窗口上由编译错误码查找错误信息和改错的对策

行错误来修改源程序再重新编译，如果原来的第 1 行错误消除了，再针对新的第 1 行错误来修改，再重新编译，直到没有编译错误为止，此时在 Build 窗口上，将出现如下信

息：

```
-----Configuration: Iosl0—Win32 Debug-----
```

```
Compiling...
```

```
filename.cpp
```

```
filename.obj - 0 error(s), 0 warning(s)
```

这说明编译成功，并生成以源程序名 filename 为名字的可重定位文件 filename.obj。在此还必须提醒初学者，在每次修改源程序后，一定要存盘一次以确保修改后的内容保存下来，然后再进行新一轮的编译操作，也避免了编译系统是用没有修改前的内容在进行编译。接着进行链接操作，操作方法是沿着“Build(主菜单)/Build filename.cpp(子菜单)或 F7(热键，即按压 F7 键则直接一步选取到该项子菜单)”的操作路径用鼠标左键点击，链接过程中的错误处理与编译时类同，仅链接错误码采用“LNK5555”的格式表示，例如，LNK2660。当链接成功后，在 Build 窗口上将出现如下信息：

```
-----Configuration: simpl—Win32 Debug-----
```

```
Linking...
```

```
filename.exe - 0 error(s), 0 warning(s)
```

则说明已生成了可执行文件 filename.exe。

3. 运行可执行文件

方法是沿“Build(主菜单)/Execute filename.exe(子菜单)”的操作路径用鼠标点击，则执行 filename.exe，并将弹出一个显示输出结果的窗口，如图 1.16 所示。源程序中凡是用“cout << 对象；”语句输出的结果都将显示在该窗口上。若按任意键则关闭输出结果窗口，屏幕将恢复原样，即回到“主屏幕”。

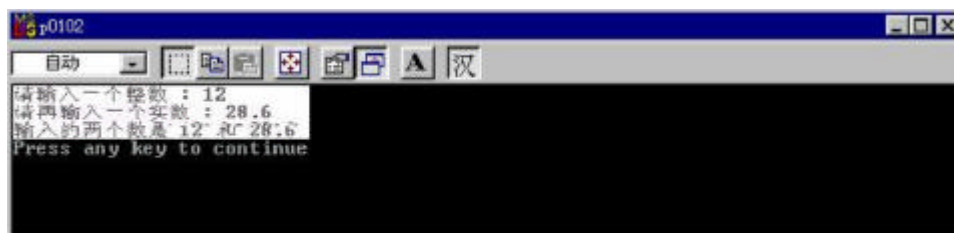


图 1.16 从输出结果的窗口上裁剪输出结果

4. 裁剪输出结果

在显示输出结果的窗口上，用鼠标点击工具栏上的“标记(Mark)”按钮，则在左上角将出现闪动的白色块光标，再将鼠标置于该光标上按下左键拖动，直到覆盖了所有输出结果为止，正如图 1.16 所示，则得到了一个待裁剪的白色块，再用鼠标点击工具栏上的“复制”按钮，则将它送到了 Windows 的“剪贴板”上，用户可把它粘贴到任何地方，

即 Windows 平台上任何形式的文件,如“.c”、“.cpp”和“.doc”等文件。

1.4.3 关闭源程序

对于含单个源文件的程序,每当调试完一个后,应该沿“File(主菜单)/Close Workspace(子菜单)”的操作路径用鼠标左键点击,关闭系统为该文件所建立的工作区后,才能调出另一个程序或者创建一个新的源文件进行调试。如果采用“File(主菜单)/Close(子菜单)”的操作路径用鼠标左键点击,那只关闭了编辑窗口而没有撤销“工作区”,若再“打开”或“新建”一个源文件时,将产生链接错误。这是因为在“工作区”内原来已经有一个 main()函数,再“打开”或“新建”一个源文件时又向“工作区”内添加了一个 main()函数,由于 main()函数对于“控制台程序(Console Program)”代表程序的执行起点,在“工作区”内存在两个 main()函数将无法进行链接操作。

1.4.4 调试器的使用方法

掌握调试器 Debugger 的使用方法是开发程序的最重要的基本技能。当编译、链接完成后,若发现某对象的结果有误,可启动调试器 Debugger,利用调试器“单步(热键 F10)”或“跟踪(热键 F11)”的功能,当按顺序逐条执行语句时,在“监视(Watch)”窗口上观察要监视对象的变化,以确定究竟在执行哪条语句时发生了错误。其操作步骤如下:

将光标移动到要设置的断点处,用鼠标沿“Build(主菜单)/Start Debug(1级子菜单)/Run to Cursor(2级子菜单,热键 Ctrl+F10)”的操作路径点击,启动调试器 Debugger,如图 1.17 所示,在屏幕右上方的编辑窗口内可见,程序执行到光标所

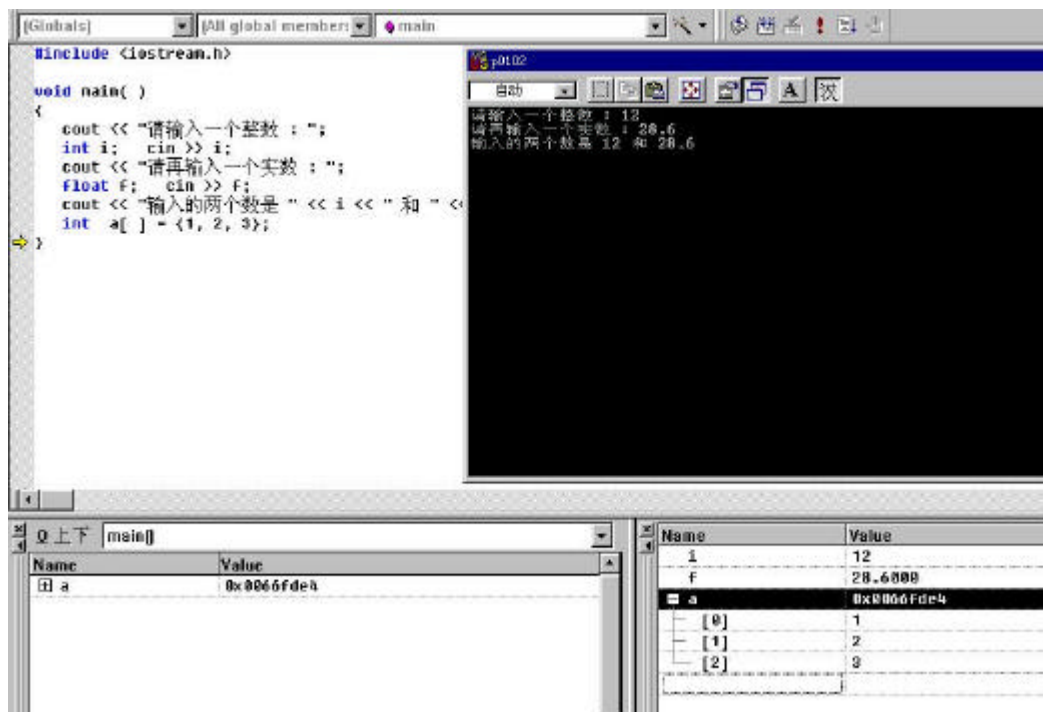


图 1.17 编辑窗口、监视窗口、工作区窗口和变量窗口

指定的断点处中断停止。屏幕左下方是变量的调试窗口，简称变量窗口，在 Context 框内填写的是 main()，此时，它自动地把 main() 函数内的变量在程序执行到断点处的值（断点处的这条语句还未执行）都显示出来，以便调试时进行监视。屏幕左上方是工作区窗口，若它还未打开，则用鼠标沿“View(主菜单)/Workspace(子菜单)”的操作路径打开它。

如果还需要监视复杂的数据结构，例如一个数组 a，可采用屏幕右下方的监视 (Watch) 窗口，若它还未打开，则用鼠标沿“View(主菜单)/Debug Windows(1 级子菜单)/Watch(2 级子菜单，热键 Alt+3)”的操作路径打开它，其内一般有 4 个监视窗口 Watch1 ... Watch4。操作者可选择某一监视窗口如 Watch1，在该窗口的“Name:”栏内键入所要监视的数组 a，按回车键后在“Value:”栏内将出现数组 a 的信息，用鼠标箭头指向 a 前面的 ☐，接着按下鼠标左键点开它，正如图 1.17 所示，则数组 a 的每个元素值都显示在“Value:”栏内。

每按一次 F10 键或 F11 键，则执行一条语句。F10 为“单步”操作，不跟踪到被调用函数体内，而 F11 为“跟踪”操作，则跟踪到被调用函数内的一条语句，调试者可根据情况选择使用。本例是在“监视 (Watch1)”窗口上观察每执行一条语句后被监视对象的变化，以确定究竟在执行哪条语句时发生了错误。

关闭调试器 Debugger 操作者应沿“Debug(主菜单)/Stop Debugging(子

菜单)”的操作路径用鼠标左键点击则关闭它。

1.4.5 查找信息

用鼠标沿“Help(主菜单)/Contents(子菜单)”的操作路径点击,启动 MSDN (Microsoft Developer Network) 库,将弹出 MSDN 窗口,在该窗口的“Multiple Page”控件上点击“目录”,则将出现“+ MSDN Library Visual Studio 6.0”,点击“+”打开目录,其内包含 Visual Basic、Visual C++、Visual Foxpro、Visual Visual J++等文档资料。再打开“Visual C++ Documentation”目录,查阅有关信息。例如,若要查找 ISO/ANSI C 中的标准函数 strcpy(),可在该窗口的“Multiple Page”控件上点击“搜索”页面,在该页面的“输入要查找的单词(F):”框内键入 strcpy(CR),则在 MSDN 窗口上将显示出 strcpy()的信息。

1.4.6 建立工程文件


对于由多个源文件组成的源程序,可采用建立工程文件的方法编辑、编译、链接和调试。其操作方法简述如下:

VisualC+++V6.0 按照“File(主菜单)/New(子菜单)”的操作路径进入到“New”对话框如图 1.18 所示,在 New 对话框内,再选择多页面控件中的 Project 页面,接着选择“Win32 Console Application”子菜单项(不使用任何可视化控件,如窗口、对话框、菜单等的 C++程序称为 Console 程序),这时项目(Project)的目标平台选择框内将会出现:“Win32”。



图 1.18 “New(新建)”对话框的多页面控件

在“Project name :”框内输入工程文件名,例如 P0103(项目名)。

在“Location:”框内输入或指定路径名,建立工程文件所需要的源文件均在该路径名下。例如,p0103main.cpp,volume.cpp等文件在D:\VC++User目录路径下,则应输入的路径名为:D:\VC++User,在“Location:”框内将会出现“D:\VC++User\p0103”。也可用鼠标左键点击“Location:”框右边的浏览键 ,在“Choose Directory”对话框内选取所需要的路径。随后用鼠标左键点击New对话框内的“OK”键,则将弹出“Win32 Console Application-Step 1 of 1”选择框,选择“An empty project”项后,点击finish(完成)按钮,则弹出“New Project Information”窗口,其内将出现如下信息:

```
+ Empty console application.
+ No files will be created or added to the project.
```

用鼠标点击该窗口内的“OK”按钮,则建立了名字为P0103的工程文件项目。

向工程文件项目中添加文件:既可以添加扩展名为“.cpp”的源文件,也可以是扩展名为“.h”的头文件,或者是filename.obj文件等。若所添加文件是过去已经开发好的已存在文件,则按照“Project(主菜单)/Add To Project(1级子菜单)/Files(2级子菜单)”的操作路径用鼠标左键点击,进入到“Insert Files into Project”对话框,其下方的“insert into:”应出现P0103。在该对话框中用鼠标左键点击的方法从指定的目录下选取所要添加到该工程文件项目的文件,选中后双击该文件名,则将所选中的文件添加到工程项目P0103中。若添加文件是一个新创建的,则按照“Project(主菜单)/Add To Project(1级子菜单)/New(2级子菜单)”的操作路径用鼠标左键点击,进入到编辑窗口,例如,p0103main.cpp就是一个新创建文件,将如下程序代码键入到编辑窗口:

例 1.3 由键盘敲入的长、宽和高值,计算该长方体体积的工程项目。它由两个源文件组成:

```
1: #include < iostream >
   // 用#include 预处理语句把输入/输出流库新标准头文件引入本程序
2: using namespace std;    // 将 std 名空间合并到当前名空间
3: unsigned volume(unsigned, unsigned, unsigned);
4: // 在 volume.cpp 文件中定义的外部函数的声明语句
5: void main( void )
6: {
7:     unsigned x, y, z, v; // 定义 4 个无符号整型变量 x、y、z 和 v
8:
9:     cout << " x = ";    // 对变量 x 的输入操作作用 cout 语句输出提示信息字符串 " x = "
10:    cin >> x;            // 用 cin 语句接收键盘敲入的无符号整型值保存在变量 x 中
11:    cout << " y = ";    // 对变量 y 的输入操作作用 cout 语句输出提示信息字符串 " y = "
```

```

12:  cin >> y;                // 用 cin 语句接收键盘敲入的无符号整型值保存在变量 y 中
13:  cout << " z = ";        // 对变量 z 的输入操作作用 cout 语句输出提示信息字符串 " z = "
14:  cin >> z;                // 用 cin 语句接收键盘敲入的无符号整型值保存在变量 z 中
15:  v = volume(x, y, z);
    // 调用 volume( )函数计算长、宽和高分别为 x、y 和 z 的长方体体积并保存在变量 v 中
16:  cout << " v = " << v << endl;
    // 输出该长方体体积值
17: }

```

程序代码敲完后，用鼠标左键点击工具条上的“保存”图标，将新建源文件保存在“D:\VC++User\P0103”目录下即可。接着，按同样的操作方法创建 volume.cpp 源文件，并添加到 P0103 工程项目中，其程序代码如下：

```

1: unsigned volume(unsigned l, unsigned w, unsigned h)
    // 该函数具有 3 个无符号整型的形式参数 l、w 和 h，接收长方体的长、宽和高值
2: {
3:     unsigned vk ;    // 定义一个无符号整型变量 vk，保存计算的体积值
4:
5:     vk = l * w * h; // 计算长、宽和高值分别为 l、w 和 h 的长方体体积
6:     return vk;      // 以所得的长方体体积值作为该函数的返回值
7: }

```

同样，当程序代码敲完后，用鼠标左键点击工具条上的“保存”图标，将新建源文件保存在“D:\VC++User\P0103”目录下即可。

编译、链接和运行工程文件项目：首先应在“编辑”窗口上，选中该工程文件项目中含 main ()函数的源文件，例如 p0103main.cpp，然后按照“Build (主菜单)/Rebuild All (子菜单)”的操作路径用鼠标左键点击，则对项目中所包含的所有文件进行编译、链接。若有错误，将显示在错误信息窗口上，便于编程者修改程序时阅读，直到没有错误信息为止。这时便生成了可执行文件 P0103.exe，其文件名就是项目名。运行工程文件的方法与运行单个源程序完全相同。

接着可使用调试器对整个工程项目的源文件进行调试，其操作方法与单个源文件的基本类同，只是在操作者使用 F11 跟踪到另一个源文件（如 volume.cpp）的被调用函数（如 volume()函数）内时，系统会自动地打开该源文件并显示在编辑窗口上，跟踪指示箭头也自动地停在该函数的第 1 条语句。

顺便指出,在 Visual C++ 集成开发环境上,编辑窗口的左边有一个“工作区窗口”如图 1.19 所示,它具有“Class View”和“File View”两个页面,前者是“分类观测”,后者为“文件观测”。本工程项目只有两个全局型的外部函数 `main()` 和 `volume()`,所以分类观测则在“工作区窗口”上显示出 Globals 目录下包含 `main()` 和 `volume()` 两个函数。如图 1.19 所示,“文件观测”分成“Source Files (源文件)”、“Header File (头文件)”和“Resource Files (资源文件)”等 3 个目录,本工程项目不包含头文件和资源文件,只有 `p0103main.cpp` 和 `volume.cpp` 两个源文件。操作者在“工作区窗口”上,让鼠标箭头指向其中某个文件名,再用左键双击它,则立即在编辑窗口上显示出该文件的全部源程序代码。

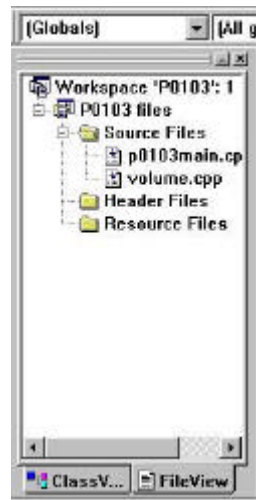


图 1.19 工作区窗口

再次打开工程文件项目的操作路径是“File (主菜单) / Open Workspace (子菜单)”,则将弹出“Open Workspace”对话框。在该框内选择工程文件所在的路径,如例程中应为“D:\VC++User\P0103”,然后选中 `P0103.dsw` 文件,用鼠标左键双击该文件名便能打开项目名为 `P0103` 的工程文件,即打开该工程文件所包含的所有文件,并为该项目分配了“工作区 (Workspace)”,用户随即可开始进行调试。

小 结

传统的模块化程序设计强调程序的功能,自顶向下分解任务,直到底层,以一个函数为一个功能单位,最顶层的主函数按顺序调用子函数来完成计算和操作并返回信息,它要求编程者详细了解所研究对象的具体细节,深入钻研各项业务和计算机知识,难以提高软件的开发效率。而采用面向对象程序设计方法编写应用程序,不要求编程者过于深入地了解计算机硬、软件知识和各项业务的基础理论知识,是高效率、高质量开发软件的有力工具。

面向对象程序设计以对象为核心,以抽象和归类为基础,强调程序分层分类的概念,这是克服软件复杂性,将客观世界的模型在计算机中自然地表现出来的最好方法。

面向对象程序设计把将一种数据结构和操作该数据结构的方 法捆在一起,封装在一个程序实体内,从而实现了数据封装和信息隐藏。通常,用一个数据成员描述某类的一个属性,就应该备有一些访问该数据属性操作方法的成员函数。

对象、消息传递机制和以继承为基础 的类层次结构是面向对象程序设计的三个要点。

对象是抽象数据类型 ADT 的实现,ADT 是根据要达到的目的描述客观实体的抽象化模型,抽象出与目的相关的主要信息,忽略掉一些次要信息,隐藏具体的实现细节。

消息传递机制是对象之间相互通信和作用的惟一方式。

开发 C++ 程序的操作步骤通常包括编辑、编译、链接、运行和调试等。

习 题 1

一、选择填空

- 关于 C++ 与 C 语言的关系的描述中, () 是错误的。
 - C 语言是 C++ 的一个子集
 - C 语言与 C++ 是兼容的
 - C++ 对 C 语言进行了一些改进
 - C++ 和 C 语言都是面向对象的
- 下面关于对象概念的描述中, () 是错误的。
 - 对象就是 C 语言中的结构变量
 - 对象代表着正在创建的系统中的—个实体
 - 对象是一个状态的操作 (或方法) 的封装体
 - 对象之间的信息传递是通过消息进行的
- 下面关于类概念的描述中, () 是错误的。
 - 类是抽象数据类型的实现
 - 类是具有共同行为的若干对象的统一描述体
 - 类是创建对象的样板
 - 类就是 C 语言中的结构类型
- 面向对象程序设计思想的主要特征不包括 ()。
 - 继承性
 - 功能分解, 逐步求精
 - 封装性和信息隐藏
 - 多态性
- 下列关于 C++ 关键字的说法中, () 是正确的。
 - 关键字是用户为程序中各种需要命名的“元素”所起的名字
 - 关键字是对程序中的数据进行操作的一类单词
 - 关键字是在程序中起分割内容和界定范围作用的一类单词
 - 关键字是 C++ 预先定义并能实现一定功能的一类单词
- 按照标识符的要求, () 符号不能组成标识符。
 - 连接符
 - 下划线
 - 大小写字母
 - 数字字符
- 下列符号中, () 不可作为分隔符。
 - ,
 - :
 - ?
 - ;
- 下列字符串中可以用作 C++ 标识符的是 ()。
 - _123
 - for~bar
 - case
 - 3var

二、判断下列描述的正确性, 对者划 , 错者划 ×

- C++ 中使用了新的注释符 “//”, C 语言中的注释符 “/*...*/” 不能在 C++ 中使用。

2. C++程序中, 每条语句结束时都加一个分号“;”。
3. C++中标识符内的大小写字母是没有区别的。
4. 在编写 C++程序时, 一定要注意采用人们习惯使用的书写格式, 否则将会降低其可读性。
5. C++是一种以编译方式实现的高级语言。
6. 在 C++编译过程中, 包含预处理过程、编译过程和链接过程, 并且这 3 个过程的顺序是不能改变的。
7. 预处理过程一般在编译过程之后、链接过程之前进行。
8. 源程序在编译过程中可能会出现一些错误信息, 但在链接过程中将不会出现错误信息。

三、问答题

1. 什么是数据封装和信息隐藏?
2. 如何理解类和对象?
3. 面向对象程序设计的要点是什么?

四、分析下列程序的输出结果

程序 1:

```
#include < iostream >
using namespace std;
void main( void )
{   cout << "Beijing" << " ";
    cout << "Shanghai" << "\n";
    cout << "Tianjin" << endl;
    cout << "Wuhan" << endl;
}
```

程序 2:

```
#include < iostream >
using namespace std;
void main( void )
{   int a, b;
    cout << "Please Input a : ";
    cin >> a;
    cout << "Please Input b : ";
    cin >> b;
    cout << "a = " << a << " , " << "b = " << b << endl;
    cout << "a - b = " << a - b << "\n";
}
```



```
    }
    假定输入 a = 9 , b = 3
```

程序 3 :

```
#include < iostream >
using namespace std;
void main( void )
{
    char c = 'm';
    int d = 5;
    cout << "d = " << d << " ; " ;
    cout << "c = " << c << "\n";
}
```

五、编译下列程序，改正所出现的各种错误信息，并分析输出结果

1.

```
main( void )
{
    cout << "This is a string!"    }
```

2.

```
#include < iostream >
void main( void )
{
    cin >> x;
    int p = x * x;
    cout << "p = " << p << "\n";
}
```

3.

```
#include < iostream >
using namespace std;
void main( void )
{
    int i, j;
    i = 5;
    int k = i + j;
    cout << "i + j = " << "\n";
}
```

六、通过对第五题 3 个程序中所出现问题的修改，回答下列问题

1. 从对第五题程序 1 的修改，总结出编程时应注意哪三个问题？
2. C++源程序中所出现的变量是否都必须先说明后使用？
3. 使用 cout 和运算符"<<"输出字符串时应注意什么问题？

4. 有些变量虽然说明了但是没有赋值, 这时能否使用?
5. 一个程序编译、链接通过了并且运行后得到了结果, 这个结果是否一定正确?
6. 请将下列程序补充完整, 其运行结果是在屏幕上输出字符串 "Hello word !"。

```
#include    < iostream >
int  main( void )
{
    _____
    return  0;
}
```

7. 请指出并改正下列程序中存在的 3 个错误。

```
main( void )
{
    std::cout <<  "This  is  a  string . \n"
    return  0;
}
```

错误 1 :	_____	改正 :	_____
错误 2 :	_____	改正 :	_____
错误 3 :	_____	改正 :	_____

第 2 章 从 C 快速过渡到 C++

C++是 C 的超集，C++保留了 C 的所有组成部分，与 C 具有优良的兼容性，它增添了“面向对象程序设计 OOP”部分，成为面向对象程序设计 OOP 的第一个大众化版本。本章着重介绍 C++具体吸收了 C 语言的哪些功能，并扩充、完善这些功能，针对数据类型、常量和变量、指针和引用、运算符及函数等重要概念加以简述，使学生了解 C 与 C++间的不同，快速过渡到 C++。

必须强调指出，熟悉 ISO/ANSI C 老标准的编程者，要想使自己快速成为编写 C++实用化程序的高手，必须尽快熟悉 ISO/ANSI C++新标准，在编写 C++实用化程序时，自觉地扬弃老标准的编程习惯，而采用新标准的编程风格。世界顶级 C++大师 Scott Meyers 在文献[26]中，总结自己长期积累的工程经验和对 C++专业编程技术人员进行教学的实践经验，归纳出了 50 条简短、明确、便于记忆的准则条款，作为这类编程者快速、熟练掌握 ISO/ANSI C++新标准的有力武器，自觉地使用新标准的编程格式以消除程序中的瑕疵和隐患。我们将在后续章节的相关内容中逐个地加以介绍。

2.1 数据类型

数据类型也简称为“类型”。C++中的每个变量和常量都具有数据类型的属性，数据类型分为基本数据类型和复杂数据类型，后者有时也称为构造数据类型和派生数据类型。

2.1.1 基本数据类型

C++的基本数据类型如表 2.1 所示，它保留了 C 语言的所有基本数据类型，只对实型数增加了长双精度类型(long double)。每个数据都要在计算机内存(Memory)中分配大小一定的空间来存放这些数据，各种数据类型所包含的二进制位数(bit，比特)各不相同，即数据类型的长度不同，例如字符型为 8 bits，短整型为 16 bits，长整型为 32 bits，单精度浮点型为 32 bits 和双精度浮点型为 64 bits 等。对于微型计算机，不管是 8 位机、16 位机还是 32 位机，其内存的一个存储单元长度(即所包含的二进制位数)都是 8bits，称为一个字节(byte)，即通常所说的一个字节等于 8 bits。因此，各种数据类型的长度通常都是用所占用的内存空间字节数(即内存空间的存储单元个数)来表示。各种数据类型的长度简称“数据长度”列于表 2.1 中。由此可知，对某种实例(如对象、变量和常量等)所进行的数据类型描述就确定了其在内存中所占有的空间大小，也确

定了其表示的数值范围，即数据在内存中存放，存放的格式取决于它的数据类型。下面对表 2.1 进行如下说明。

在表 2.1 中，出现的[*int*]可以省略，即在 *int* 之前有修饰符 *short*、*signed*、*unsigned*、*long* 出现时，可以省略关键字 *int*。

单精度类型(*float*)，双精度类型(*double*)和长精度类型(*long double*)统称

表 2.1 C++的基本数据类型

类型名	数据长度(BC3.1) (单位：字节)	数据长度(VC6.0) (单位：字节)	数值范围 (VC6.0)
<i>char</i>	1	1	- 128 ~ 127
<i>Signed char</i>	1	1	- 128 ~ 127
<i>unsigned char</i>	1	1	0 ~ 255
<i>short[<i>int</i>]</i>	2	2	- 32768 ~ 32767
<i>signed short[<i>int</i>]</i>	2	2	- 32768 ~ 32767
<i>unsigned short[<i>int</i>]</i>	2	2	0 ~ 65535
<i>int</i>	2	4	- 2147483648 ~ 2147483647
<i>signed[<i>int</i>]</i>	2	4	- 2147483648 ~ 2147483647
<i>unsigned[<i>int</i>]</i>	2	4	0 ~ 4294967295
<i>long[<i>int</i>]</i>	4	4	- 2147483648 ~ 2147483647
<i>signed long[<i>int</i>]</i>	4	4	- 2147483648 ~ 2147483647
<i>unsigned long[<i>int</i>]</i>	4	4	0 ~ 4294967295
<i>float</i>	4	4	约 6 位十进制有效数字
<i>double</i>	8	8	约 12 位十进制有效数字
<i>long double</i>	10	8	约 15 位十进制有效数字
<i>void</i>	0	0	无值型

为浮点类型。

char 型和各种 *int* 型有时又称为整数类型。因为这两种类型的变量（或称对象，一个基本数据类型的变量可以称为这种基本数据类型的对象）是很相似的。*char* 型变量在

内存中是以字符的 ASCII 码值的形式存储的。

各种数据类型的长度是以字节为单位,1 个字节(byte)等于 8 个二进制位(bit)。

void 型又称为无值型或空值型,其取值为空,数据长度为 0,它是一种人为理想化的数据类型,现实世界不存在一种数据是没有数值的,因此,C 和 C++中也没有 void 型的变量和常量。但定义它有两个用途:其一是用于定义无返回值的函数,即凡是没有返回值的函数都是把返回类型说明为 void 型;其二用来定义一种 void 型指针或称为无类型指针,如

```
void * pv;
```

这种无类型指针可以指向任何基本数据类型的目标变量,它是一种很有用的指针类型,将在 2.3 节详细讨论。

表 2.1 列举了两种典型的 C++语言处理系统的数据长度和数值范围,典型的 32 位机 C++语言系统为 Visual C++ V6.0(简称 VC6),数据长度和数值范围是指字长为 32 位微型计算机(简称 32 位机)的,而 16 位机典型的 C++语言系统为 Borland C++ V3.1(简称 BC31),BC31 以下的版本作了如下规定:

```
int          short [int]
signed [int]  signed short [int]
unsigned [int] unsigned short [int]
```

式中,“ ”表示“等价于”的意思,例如,整型(int)与短整型(short)的数据长度一样都是两个字节,而 VC6 版本却规定:

```
int          long [int]
signed [int]  signed long [int]
unsigned [int] unsigned long [int]
```

即将整型(int)的数据长度规定得与长整型(long)一样,都是 4 个字节。这两种语言系统都符合 ISO/ANSI C++新标准的技术规范,因为标准中只规定了基本数据类型长度的两个原则:int 型的数据长度不能比 short 型的短,而又不能比 long 型的长,可根据机器的体系结构来具体确定。这就导致了各计算机软件厂商推出的 C++产品存在着差异,这给程序的移植增加了麻烦。

2.1.2 复杂数据类型

C++中采用“*”、“&”、“[]”、“()”等运算符与标识符组合派生出一些复杂的数据类型,亦称构造数据类型或派生数据类型,其中“*”和“&”是单目运算符,放在标识符的前面,称为“前缀运算符”,而“[]”和“()”是双目运算符,放在标识符的后面,叫做“后缀运算符”。利用它们可以派生出如下类型。

1. 数组

常用格式：“数组名[下标]”为访问数组某元素表达式。

```
char s[ ] = "abcdefghi";
```

```
...
```

```
s[0] = "A";
```

2. 指针

指针实际上就是某存储单元的地址，包含“指针变量”和“常量指针”。常量指针将在下一节详细介绍。指针变量用来存放某普通变量地址，常称为“该指针变量指向了某目标变量”且指针变量被系统所分配的内存空间内存放的是另一个目标变量的地址值，它的数据类型是其所指目标变量的数据类型。若不特别声明通常把指针变量简称为指针。常用表达式为“*指针变量名”，它表示对该指针变量取内容运算，即用它来表示该指针变量所指的目标变量。

```
int * pi;
```

```
float * pf;
```

```
int (* pFun) (char *, int length);
```

第3个语句是一个说明语句，它声明了一个函数指针 pFun，前缀运算符“*”和后缀运算符“()”同时应用于一个派生类型。由于“()”运算符的优先级高于“*”运算符，因此再用一个圆括号包围“* pFun”，以表示 pFun 首先是一个指针，该指针指向具有两个形参的函数，叫做函数指针。详细内容请参阅文献[25]。

3. 枚举类型 (Enumerated types)

它是在 ISO/ANSI C 老标准中就已经定义了的复杂数据类型，ISO/ANSI C++新标准仍然保留。即将若干个整型常量按顺序排列集成 enum 类型。详细内容请参阅文献[25]。

```
enum color{red, yellow, green, ...};
```

4. 引用(reference)变量

引用变量常简称为“引用”，它才是 ISO/ANSI C++新标准增添的新数据类型，既不同于普通变量的类型，也不同于指针变量，常用的定义格式如下所述。

```
<数据类型> & 引用名 = 被引用对象;
```

引用名也是由编程者启用的一种标识符。

```
int val = 5;
```

```
int & refv = val;
```

引用是 C++新增加的一种很重要的类型，本章将详细地介绍它的概念和用法。

5. class 类型 (包括 struct 和 union 类型)

它是由用户定义的一种新的复杂数据类型，用来表示客观世界某类实体的抽象数据类型 ADT，在具体应用 class 类型时是用该类的对象来存储和处理数据。类是面向对象程序

设计的核心,是我们学习的重点内容。

6. 常量 (constants)

C++不仅保留 C 语言的常量,而且把它加以扩展。

```
const int tabsize = 8;
```

下一节将详细地讨论这部分常量。顺便指出,以后凡说到“类型”,则均指上述的基本数据类型和复杂数据类型。

2.2 C++的常量和变量

常量和变量是程序中处理的基本数据对象,它们通过运算符组合在一起构成了表达式。在任意表达式后面加上一个分号即“表达式;”,就构成了表达式语句。

2.2.1 常量

常量是在程序中其数值不能改变的量。在 ISO/ANSI C++新标准中仍保留着 ISO/ANSI C 老标准中的一类常量,它们在程序中不必进行说明就可以直接使用。请参阅文献[25]。

在 C 语言中采用宏定义 (#define) 语句定义符号常量,C++除保留这一功能外,还将符号常量这一名称加以扩展,即任何类型的对象(指针变量、引用变量、结构变量、数组、class 类型的对象)都可以定义成常量,亦即任何类型在定义时,启用一个标识符作为名字,并在它名字的前面加上关键字 const 就可作为常量使用,它的值在其作用域内不能被变更,即不能编写程序,如用赋值语句改变它的值。换句话说,常量不能放在赋值运算符的左边作为“左值表达式”,否则编译时系统将给出如下编译错误信息。

```
require a Lvalue !      (需要一个左值)
```

基本数据类型的变量可以定义为常量。

```
const int ArraySize = 100;      // 定义一个 int 型常量 ArraySize
int array[ArraySize];           // 用常量 ArraySize 指定数组 array[] 的大小
...
const float pi = 3.1415926;      // 定义圆周率 pi
```

对于熟悉 ISO/ANSI C 老标准的编程者应改变旧有的 C 语言编程习惯,尽量使用 const 取代#define 预处理语句来定义常量,即“尽量以编译器(compiler)取代预处理器(preprocessor)”,因预处理语句不属于语言本身的组成部分。C 语言编程习惯是使用#define 语句定义符号常量。

```
#define E 2.718281828459
```

语句中的大写字母 E 是自然对数的底。通常把定义符号常量的#define 语句放在一个头文件中,像这样包含常用数学常量的头文件可能是很早就由别人已经做成了的,编程者

有时就需要花费时间去查找它的原始定义语句。使用 `const` 取代 `#define` 预处理语句来定义常量，既简单且又有效地避免了这类麻烦，即：

```
const double E = 2.718281828459;
```

不仅基本数据类型的变量可定义为常量，数组也可定义为常量，这时数组的每个元素值在其作用域内不能被改变。其格式为：

```
<类型> const 数组名[元素个数] = {初值表};  
或 const <类型> 数组名[元素个数] = {初值表};
```

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

若程序中写有赋值语句

```
data[1] = 10; // 出错，不能修改常量值
```

则因赋值语句左边的 `data[1]` 是数组 `data[]` 的第 1 号元素为常量名，而常量名不能放在赋值运算符“=”的左边，所以必须在定义常量类型的同时进行初始化。

结构变量（即一个结构类型的对象）也可以定义为常量，此时其内的数据成员值在其作用域内都不能再改变。

```
struct complex {  
    double real;  
    double imag;  
};  
const complex x = {1.0,1.0};           // 定义成常量的结构变量 x  
x.real = 2.0 ;                          // 出错，不能修改常量值  
x.imag = 3.0 ;                          // 出错，不能修改常量值
```

对于指针要涉及的两个对象，即指针本身和指针所指的目标变量，则有：

若将指针所指的目标变量定义成常量，则将 `const` 直接加到指针说明语句之前，格式为：

```
const <类型> * 指针名 = 地址表达式;  
<类型> const * 指针名 = 地址表达式;
```

例如：

```
const char * pc = "asdf";
```

则 `pc` 所指的字符串不能更改，例如：

```
pc[3] = 'a'; // 出错，不能修改常量值
```

但指针本身不是常量类型。可以重新给它定向，让它指向另一个目标变量，写成：

```
pc = "hello";
```

在 C 语言中我们就知道，字符串常量是用一个字符串数组来处理，即实际存在一个匿名的字符串数组，我们启用“`x`”当做它的数组名，则上面的第 1 条语句可理解为：


```
char x[ ] = "asdf";      // 将字符串常量"asdf"存放在字符串数组 x[ ] 中
char * pc = x;           // 定义一个字符串指针 pc 并令它指向匿名字符串数组 x[ ]
```

或者写成：

```
char * pc;               // 先定义一个字符串指针 pc 令它指向匿名字符串数组 x[ ]
pc = x;                  // 再用赋值操作令它指向匿名字符串数组 x[ ]
```

这实质上是将字符串数组 `x[]` 的首地址赋值给指针 `pc`，使 `pc` 指向字符串“asdf”，根据一个数组与指向它的指针间有着密切的关系，即只要字符串指针 `pc` 指向了字符串数组 `x[]`，那么就有如下 4 种等效但不等价的访问该数组第 `i` 号元素的表达式：

`x[i]`，`pc[i]`，`*(x + i)`，`*(pc + i)`

因此，上面的第 2 条语句“`pc[3] = 'a';`”就是编程者想将匿名字符串数组 `x[]` 的第 3 号元素字符‘f’改成字符‘a’，正如图 2.1 所示，由于把字符串数组 `x[]` 定义成常量，故字符串“asdf”内的每个字符不能修改，但指针 `pc` 不是常量指针，可以用如下语句重新给它定向：

```
{ char s[ ] = "hello";
  pc = s;           // 等价于 pc = "hello";
```

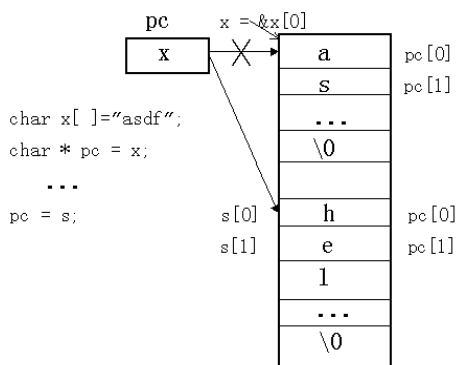


图 2.1 指向常量的指针 `pc` 可以重新定向

指针本身可以用运算符“*”加 `const` 定义为常量，即用“* `const`”把指针本身指定为常量，称为常量指针，即指针所具有的地址值是不可改变的，而指针所指目标对象的值是可以改变的。必须在定义常量指针的同时给它定向，其格式为：

```
<类型> * const 指针名 = 地址表达式;
```

例如：

```
char * const pc = "asdf"; // 指向字符串的常量指针
```

可以写做：

```
pc[3] = 'a';
```

但不能写做：

```
pc = "hello";           // 出错，不能给常量指针重新定向
```

指针和所指目标对象可同时定义为常量，成为指向常量的常量指针，这时两者都不能再改变了。其格式为：

`const <类型> * const 指针名 = 地址表达式;`

例如：

```
const char * const cpc = "asdf";
```

则有：

```
cpc[3] = 'a';           // 出错，不能修改常量值
cpc = "hello";          // 出错，不能给常量指针重新定向
```

对于初学者可以采用如下办法来记忆上述有关指针的3种常量的定义格式：

在“*”号的左边划一条垂直线，该垂直线右边的const使指针本身指定为常量，其左边的const使得指针所指的目标对象为常量；若const在垂直线的左、右边都出现，则就是定义了一个指向常量的常量指针。

顺便指出，常量指针是C++中一个非常有用的概念。除了用“*const”定义的常量指针外，还有变量的地址、数组名、结构数组名、对象数组名、函数名等也是常量指针。因此const与指针配合使用，可以定义指向常量的指针、常量指针和指向常量的常量指针，列于表2.1中。表中“类型”包括基本数据类型、结构体、类等。指向常量的指针和常量指针的区别在于：指向常量的指针只能指向常量，但编程者可随时修改指针的地址值，使它指向另一个常量；而常量指针只能通过初始化操作进行定向，一旦定向后，编程者就不能使它指向任何其他的对象。

表 2.1 指向常量的指针，常量指针和指向常量的常量指针

名 称	定 义 格 式	举 例
指向常量的指针	const<类型>*指针名;	const int *pc;
	<类型>const*指针名;	int const *pc;
常量指针	<类型>*const 指针名;	int *const cp;
指向常量的常量指针	const<类型> * const 指针名;	const int * const cpc;

可以把一个函数的返回值说明为常量。例如一个const型的指针函数，该函数的返回值是一个常量指针，其一般格式为：

<返回类型>

↓

<存储类> const <类型> * 指针函数名(参数表)

{ <函数体> }

其中，函数体中的“const <类型> *”应被看成一个整体，用来说明函数的返回类型，它表示函数返回值是一个指向常量的指针。

例 2.1 返回到一个指向常量的指针函数。

```
#include    < iostream >                // 使用 C++新标准的流库
using      namespace  std;              // 将 std 标准名空间合并到当前名空间
const char name[ ] = "BillGats";       // 定义一个常量型的字符串数组 name[ ]
// 返回到一个指向常量的指针函数 getName(void)
const char * getName(void)
{
    return  name;
    // 返回值是一个常量数组的首地址，该函数是返回到一个指向常量的指针
}
void main( void )
{
    const char * const cpc = getName( );
    //定义一个指向常量的常量指针 cpc，接受 GetName( ) 函数的返回地址值
    cout << "签 名：" << cpc << endl;
}
}
```

该程序的输出结果为：

签 名:BillGats

getName()函数的返回值只有用一个字符型且又是一个指向常量的常量指针才能接受，要想写程序修改字符串中的某个字符是不行的，这样就确保了所签名的字符串不被程序更改，如：

```
cpc[3] = "a";                // 出错，不能修改常量数组的元素
```

也不能换成其他人的名字，如：

```
strcpy(cpc, "Bush");        // 出错，常量指针 cpc 不能重新定向，指向另一个字符串
```

因此，该函数只能读取一个字符串，而不能读取其他字符串。

2.2.2 变量

变量是在程序执行时其值可以改变的量，在 C 语言中任何一个变量在使用前必须用说明语句定义。

1. C++ 保留 C 语言变量的基本要素

在 ISO/ANSI C++新标准中仍然保留 C 语言变量的基本要素。其定义格式为：

<存储类> <类型> 变量名 [= 初值];

其中[]所包围的部分可缺省，即有时在定义变量的说明语句在不给它赋初值，而在后面用赋值语句给它赋值。变量具有 4 个基本要素：作用域、名字、类型和值。

变量的名字即变量名，为标识符的一种，按标识符的命名规则给它启用名字。

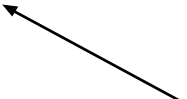
变量的类型包括基本数据类型和复杂数据类型。一个变量类型不仅决定了该变量所占内存空间的大小（字节数），而且也规定了该变量的合法操作。例如指针变量只能进行有限的几种具有实际操作含义的运算。

变量的值。任何变量必须具有数值才能参加运算和操作。可以在定义变量的同时给它赋初值，称为“初始化操作”。例如：

```
double price = 15.5;
```

也可在定义时不赋初值，而用赋值操作语句给它赋值：

```
char * p;  
p = "abcdefghi";  
int a;  
int * pa;  
pa = &a;           // 给指针变量赋地址值，称为“定向”
```



变量的作用域。变量在定义时由存储类来指定它的作用域。所谓“作用域(scope)”是指该变量在其上有定义的程序部分，通俗地说，即该变量名在某块程序区域是否有意义。从作用域角度出发，将变量分为全局变量(Global)和局部变量(Local)。在一个函数内或在一个复合语句（即块语句）内定义的变量，只在该函数体范围内或块的一对大括号的范围内才能使用，称为“局部变量”。详细内容请参阅文献[25]。

标识符的可见性。标识符的作用域与可见性密切相关。所谓“标识符的可见性”是指能否对该标识符进行访问操作，不能访问则称为“不可见”。我们仍以变量为例来进行说明。对变量进行访问操作，不外乎读取变量的值或对变量赋值（写入值）这两种操作。例如：

```
int i = 6;  
cout << "i = " << i << endl;  
printf("i = %d\n", i);  
可采用初始化操作和赋值操作对 i 变量赋值：  
int i = 6;           // 初始化操作
```

或

```
int i;      i = 6;      // 赋值操作
```

2. C++新增内容

除以上内容外，而C++中又增加了以下内容。

基本数据类型变量初始化操作的函数表示法。对于基本数据类型的单个变量，虽然保留了C语言的初始化操作，例如：

```
(auto) int i = 6;
```

但也可以采用函数表示法进行初始化操作, 格式为:

<存储类> <基本数据类型> 变量名 (初值);

例如:

```
void main( void )           或 static int i(6);
{ (auto) int i(6);
  ...
}
```

C++与C语言不同的是可以在程序中随时定义变量, 不必集中在作用域的开头和执行语句之前。例如: 把定义循环变量*i*放在for语句的初始化表达式(第一分量)中:

```
for( int i = 0; i < 6; i++ ) { ... }
```

或

```
for( int i(0); i < 6; i++ ) { ... }
```

请注意循环变量*i*的作用域不是在for语句循环体内, 而是在循环体外, 它等效于如下两条语句:

```
{ int i;
  for( i = 0; i < 6; i++ ) { ... }
```

因此, 不能在一个作用域中连续写多个这样的for语句, 造成*i*的重复定义, 而应采用另外的变量名*j*、*k*等, 写成:

```
for( int i = 0; i < 6; i++ ) { ... }
for( int j = 0; j < 6; i++ ) { ... }
```

或者写成:

```
for( int i = 0; i < 6; i++ ) { ... }
for( i = 0; i < 6; i++ ) { ... }
```

由于可随时定义变量, 因此在函数体内定义的局部变量(自动变量和内部静态变量), 其作用域应为从定义点(定义它的说明语句)开始, 到函数体结束。而在所有函数体外定义的变量(外部变量和外部静态变量), 其作用域为从定义点开始到文件结束。

作用域运算符“::”。若全局变量与局部变量同名, 则在局部变量的作用域内会将同名的全局变量隐藏起来。在C语言中在该作用域内无法再访问到同名的全局变量, 即它不可见, 但它仍然存在, 这便出现了标识符的可见性和存在性的“不一致问题”。特别是编程者若未注意到变量名被隐藏, 则由此产生的错误很难发现。因此, C++增添了作用域运算符“::”, 用以访问被隐藏的全局变量, 解决了全局变量和局部变量同名时所发生的冲突。

例 2.2 用作用域运算符访问被隐藏的全局变量。

```
#include <iostream>           // 使用 C++新标准的流库
using namespace std;         // 将 std 标准名空间合并到当前名空间
```

```

int    x;                                // 定义一个外部变量 x

void   main( void )
{
    auto int x = 1;                      // 在主函数体内定义一个自动变量 x
    cout << "(1) x = " << x << endl;    // 输出自动变量 x 的值
    ::x = 6;                             // 给外部变量 x 赋值
    cout << "(2) x = " << ::x << endl;  // 输出外部变量 x 的值
}

```

该程序的输出结果为：

```

(1) x = 1
(2) x = 6

```

2.3 C++的指针

C++的指针分为常量指针和指针变量。如前所述，变量的地址、数组名、结构数组名、对象数组名、函数名等，以及用“* const”说明的指针名均为常量指针。而指针变量是一种特殊的变量，它用来存放另一个变量（或对象）的地址值，但指针变量的数据类型是它所指目标变量的数据类型。

C++的指针变量与C语言一样，必须先给它赋地址值（或称“定向”）后才能参加运算和操作，否则使用未定向的指针变量将可能造成系统的故障。因为在定义一个指针时，编译系统虽然给指针变量分配了内存空间，但其内存放的内容（地址值）是随机的，若使用这种未定向的指针进行操作，可能会改变内存中其他部分的值，导致系统故障。例如：

```

float * px , * py;

* px = 12.6;           // 出错，使用了未定向的指针变量
* py = 16.8;           // 出错，使用了未定向的指针变量

```

正确的写法应该是：

```

float x , y , * px = &x , *py = &y;

* px = 12.6;
* py = 16.8;

```

在指针变量进行“定向”操作时，不管是用初始化操作还是地址赋值语句，应使指针变量和其目标变量的数据类型保持一致，否则，通常就需要采用强制类型转换把目标变量变成指针变量的数据类型，且这种转换也只能在基本数据类型之间进行。

C++也完全保留了C语言指针变量的一些特性，如数组和指向它的指针变量间的关系等式，还有扩展到指针数组和指向它的二级指针变量间的关系等式，以及结构数组和指向它的结构指针变量间的关系等式等，因为两者采用的是统一的地址计算公式，由此来访问这些数组的元素。详细内容请参阅文献[25]。

C++与C一样也可以定义void型指针（无类型指针），它可以指向任何基本数据类型

的变量。可以将已定向的各种类型指针直接赋给 void 型指针,反之,将 void 型指针赋给其他各种类型指针时,必须采用强制类型转换。

例 2.3 void 型指针变量的使用方法。

```
#include <iostream>          // 使用 C++ 新标准的流库
using namespace std;         // 将 std 标准名空间合并到当前名空间

void main( void )
{
    int a = 7, * pi;          // 定义 int 型的变量 a 和指针变量 pi
    double b = 5.6321, * pd;  // 定义 double 型的变量 b 和指针变量 pd
    void * pv;                // 定义一个无类型指针 pv
    pi = &a;                  // 令指针变量 pi 指向目标变量 a
    pv = pi;                  // int 型指针变量 pi 可直接赋给无类型指针 pv
    cout << "把无类型指针 pv 强制转换为 int 型指针后对其取内容运算的结果:" << * (int *)pv << endl;

    /* 表达式 “ * (int *)pv ” 实际上是要把无类型指针 pv 赋给一个 int 型指针变量, 必须使用强制类型转换 */

    pd = &b;                  // 令指针变量 pd 指向目标变量 b
    pv = pd;                  // double 型指针变量 pd 也可直接赋给无类型指针 pv
    cout << "把无类型指针 pv 强制转换为 double 型指针后对其取内容运算的结果:" << * (double *)pv << endl;

    /* 表达式 “ * (double *)pv ” 实际上是要把无类型指针 pv 赋给一个 int 型指针变量, 必须使用强制类型转换 */

}
```

该程序的输出结果为:

把无类型指针 pv 强制转换为 int 型指针后对其取内容运算的结果: 7

把无类型指针 pv 强制转换为 double 型指针后对其取内容运算的结果: 5.63

2.4 引用变量

2.4.1 “引用”的概念

引用变量 (reference variable) 简称为“引用”, 它是 IOS/ANSI C++ 标准中新增加的, 也是一种特殊类型的变量, 既不同于普通变量, 也不同于指针变量。引用相当于给变量赋予一个新的名字。其定义格式如下:

<类型> & 引用名 = 变量名;

这里所指的“类型”包括基本数据类型、结构体、联合体、类的实例 (对象) 等。用取地址运算符“&”就可以派生出它们的引用类型。引用必须初始化, 通过初始化引用, 使

它成为被引用变量的替换名。例如：

```
int i = 1;
int &j = i;          // 引用 j 的初始化操作
```

i 为 `int` 型变量，*j* 为被引用变量 *i* 的引用。特别强调指出，用 *i* 初始化引用 *j*，就是使引用变量 *j* 与被引用变量 *i* 的地址相联系，从而可把 *j* 当成 *i* 一样使用，*j* 是 *i* 的另一个名字。

2.4.2 引用的初始化

引用初始化和变量赋初值不同。因为引用初始化实质上是靠常量指针实现的，即用被引用变量 *i* 的地址 `&i` 初始化这一常量指针 `&j`。因此，初始化后引用的地址值 `&j`（它是一个常量指针）不会再改变，它总是指向被引用变量 *i*。其执行过程是这样的：当编译系统读到“`int i = 1;`”语句时，它给变量 *i* 分配一个内存空间，它的地址值为 `&i`。当读到“`int &j = i;`”语句时，对引用变量 *j* 进行初始化操作。编译系统并没有给引用 *j* 分配一个内存空间，而是产生一个常量指针（即常量地址）`&j`，引用的初始化操作就是将被引用变量 *i* 的地址值 `&i` 赋给常量指针 `&j`，使 `&j` 指向被引用变量 *i*，一旦经过初始化操作后，常量指针 `&j` 总是指向被引用变量 *i*。如图 2.2 所示，这就像有一面镜子那样，引用变量 *j* 就像是被引用变量 *i* 在镜子里的映像。显然，*j* 是一个虚拟的东西，本身不占有内存空间，而是 *i* 和 *j* 占用同一个内存空间。反映在程序上，*i* 和 *j* 代表同一变量，*j* 可以看成是变量 *i* 的替换名，即该变量具有两个名字。如同作家大都具有本名和笔名一样，用其中任何一个名字都是指同一作家，引用变量与被引用变量所指的都是同一个实体，该实体就是被引用变量。当用赋值语句改变 *i* 的值时，*j* 也随之改变，反之亦然。由于引用与被引用变量是通过地址相联系的，也可以说，引用是指针的另一种表示形式。



图 2.2 引用变量 *j* 的初始化操作

2.4.3 引用的使用

对引用的操作等价于对被引用变量的操作。由于经初始化的引用变量，其地址值是一个常量指针，总是指向被引用变量。因此，对引用 *j* 的操作就是对其所具有的常量指针 `&j` 所指向的内容，即被引用变量 *i* 的操作。例如：

```
j++;          // 等价于 i++;
```

反之，

```
i = 4;        // 等价于 j = 4;
```

对引用的操作就是对被引用变量的操作，反之，对被引用变量 *i* 的操作就是对引用

变量 *j* 的操作。反映在程序上, 引用 *j* 和被引用变量 *i* 代表同一个变量。

例 2.4 使用引用的示例程序。

```
#include <iostream>          // 使用 C++ 新标准的流库
using namespace std;        // 将 std 标准名空间合并到当前名空间

void main( void )
{
    int a(3);
    int &m = a ; // 定义引用 m 并初始化成对 int 型变量 a 的引用

    cout << "\n(1) a = " << a << " , m = " << m;
    // 引用 m 和被引用变量 a 的值是一样的

    m = m + 5;

    cout << "\n(2) a = " << a << " , m = " << m;
    // 对引用 m 的操作就是对被引用变量 a 的操作

    a ++;

    cout << "\n(3) a = " << a << " , m = " << m;
    // 对被引用变量 a 的操作就是对引用 m 的操作

    int n = m;
    cout << "\n(4) a = " << a << " , m = " << m << " , n = " << n;
    // 把引用 m 的值赋给变量 n , 即将被引用变量 a 的值赋给变量 n

    n = 6;

    cout << "\n(5) a = " << a << " , m = " << m << " , n = " << n;
    int * p = &m ;

    cout << "\n(6) &a = " << &a << " , p = " << p;
    // 对引用 m 取地址, 实际上就是取被引用变量 a 的地址

    int t = 120;
    int &d = t;

    cout << "\n(7) t = " << t << " , d = " << d;
    // 把常量赋给一个中间过渡变量 t, 再对 t 定义引用变量 d

    float f = 5.6f;
    int temp = (int)f;
    int &e = temp;
    cout << "\n(8) f = " << f << " , e = " << e << endl;
    /* 把浮点型的变量 f 强制转换成 int 型赋给中间过渡变量 temp, 再对 temp 定义 int 型
       引用变量 e */
}
```

该程序的输出结果为：

- (1) a = 3 , m = 3
- (2) a = 8 , m = 8
- (3) a = 9 , m = 9

```

(4) a = 9 , m = 9 , n = 9
(5) a = 9 , m = 9 , n = 6
(6) &a = 0x0066FDF4 , p = 0x0066FDF4
(7) t = 120 , d = 120
(8) f = 5.6 , e = 5

```

可以用某个引用 `m` 给一个变量 `n` 赋值，该变量 `n` 的值便是被引用变量 `a` 的值。但应注意：变量 `n` 与变量 `a` 的关系不同于引用 `m` 和被引用变量 `a` 的关系。“`int n = m;`”语句只是将变量 `a` 的值（也是 `m` 的值）赋给了 `n`，此外，`n` 与 `a` 没有任何联系。但引用 `m` 和被引用变量 `a` 却通过常量指针联系，代表着同一个变量。因此在执行“`n = 6;`”语句后，`n` 的值变为 6，而 `a` 和 `m` 都仍然为 9。

可将某个引用的地址值 `&m` 赋给一个指针 `p`，而指针 `p` 则指向被引用变量 `a`（如例 2.4 中的 (6)）。

通常，初始化引用时需要用一个类型相同的变量名。

若要用一个常量对一个引用初始化，应定义一个临时的中间变量 `t` 进行过渡（如例 2.4 中的 (7)）。若要用一个不同类型的变量来初始化引用，同样应定义一个临时的中间变量进行过渡（如例 2.4 中的 (8)）。

引用的用途是作为函数的参数或函数的返回值，将在以后的章节中讲述。

引用是一种特殊类型的变量，既不同于普通变量，也不同于指针变量。ISO/ANSI C++ 新标准对引用变量作了如下规定：

不能对引用变量取地址。指针变量可定义成二级指针，即指向指针的指针：

```
char ** p;
```

采用两个星号表示二级指针，然而对引用取地址，用两个“&”放在一个标识符前面，即进行二级引用是无意义的。例如：

```
&&j; // 出错，不能对一个变量名使用两个取地址号
```

不能建立引用数组。例如：

```
int a[10];
int & ra[10] = a; // 出错，没有引用数组
```

对 `void` 类型进行引用是不允许的。例如：

```
void & sum( ); // 出错，不能对 void 类型进行引用
```

`void` 只是在语法上被认为是一个类型，放在函数原型中作为返回值时，表示函数无返回值，但没有任何一个具体变量和对象，其类型是无值的。

引用不能用类型来初始化。因为引用是变量或对象的引用，而不是类型的引用。例如：

```
int &ra = int; // 出错，不能用类型来初始化引用
```

有空指针,但无空引用。空指针是代表指针变量的一种特殊状态,用户在编程时经常要设计一些检测和判断语句竭力避免这种状态,但它还是存在的、且有意义的,而空引用却是根本不存在的,毫无意义的东西。例如:

```
int * pi = NULL;
int & ri = NULL;    // 毫无意义
```

2.5 C++的运算符

C++保留了C语言中所有丰富的运算符,并增加了3个新的运算符,即new、delete和(作用域运算符)。运算量、功能(操作内容)、运算顺序等为运算符的3大要素。根据运算符所操作的运算量个数为一个、两个、三个,可将其分为单目运算符、双目运算符、三目运算符。例如,取负数运算符“-”为单目运算符,它只对一个运算量进行操作;加法运算符(+)和比较运算符(<, <=, !=, ==)为双目运算符,对两个运算量进行操作,且运算符夹在这两个运算量之间;C++中仅有一个三目运算符,即条件运算符(? :),它对3个运算量进行操作。

2.5.1 表达式中的类型转换

C++与C语言一样具有两种类型转换。

1. 隐式类型转换

这种类型转换是在执行运算符的操作时由系统自动进行的一种转换,有时也称自动类型转换。除了表达式中的隐式类型转换和赋值表达式右值自动转换成左值的类型(请参阅文献[25])外,在调用有返回值的函数时,总是将return后面的表达式类型自动转换为该函数返回值的类型(当两者类型不一致时),即以函数定义时函数返回值的类型说明为基准进行自动转换。

例 2.5 return 语句中表达式的隐式类型转换。

```
#include <iostream>           // 使用 C++ 新标准的流库
using namespace std;          // 将 std 标准名空间合并到当前名空间

int add( double x , double y )
{
    cout << "(2)x + y = " << x + y << endl;
    return x + y;
    /* 表达式 x + y 的结果值为 double 型, 将自动地转换成 int 型, 可能导致数据
       精度的降低*/
}

void main( void )
{
    int a;
    double b(3.56) , c(4.6);
```

```

a = b;
// 把 double 型变量 b 赋给 int 型变量 a, 将导致数据精度的降低
cout << "(1)a = " << a << " , b = " << b << endl;
cout << "(3)add( ) = " << add(b, c) << endl;
/* add( ) 函数的返回类型与 return 语句中的表达式类型不一致, 该表达式将自动地转换成
int 型, 可能导致数据精度的降低 */
}

```

该程序的输出结果为：

```

(1) a = 3 , b = 3.56
(2) x + y = 8.16
(3) d = 8

```

在表达式 “a = b;” 中, 将右值 b(double 型) 转换成为 int 型值 3, 然后赋给左值 a。而在调用 add() 函数时, 将 return 后面的表达式 (x + y) 的结果 (double 型值) 8.16 转换成函数的类型, 即将 int 型数 8 赋给 d。

2. 强制类型转换

此类转换因有明确的程序书写格式, 由编程者按格式书写程序来控制, 又称 “显式类型转换”。ANSI C++ 有两种书写格式。

强制转换表示法。与 C 语言相同, 其格式为：

(类型名) 表达式

编译时, 将表达式强制转换成 (类型名) 所指定的类型。例如：

```

double s;
int n = 2;
s = sqrt((double) n);

```

函数表示法。在 C++ 中, 基本数据类型名可当成一个函数名使用, 其格式为：

基本数据类型名 (表达式)

其作用是将 (表达式) 强制转换成指定的基本数据类型。例如：

```

double f = 6.86;
int h = int(f);           // 与 “int h =(int)f;” 等价

```

但是, 函数表示法不能用于非基本数据类型, 例如把一个正整数值转换为指针类型, 必须使用强制转换表示法：

```

char * p = (char *)0x00650666;

```

2.5.2 new 和 delete 运算符

C++ 为便于用户应用动态存储技术, 直接提供了 new 和 delete 运算符来实现动态存储管理功能。运算符 new 用于动态分配存储空间, 类似于 malloc() 函数, 而 delete 是它的配对物, 用于释放由 new 所开辟的内存空间, 类似于 free() 函数。

1. 运算符 new 的用法

new 运算符的使用格式如下：

```
new <类型>(初值表)
或 new <类型>
```

它表明在堆 (heap) 中动态建立了一个由 <类型> 所指定的变量或对象。第 一种形式由圆括号包围的“初值表”给出被创建变量或对象的初始值。也可采用不赋初始值的第 二种形式，但必须给它赋值后才能参加运算和操作。这里所说的 <类型> 包含所有的基本数据类型和派生 (数据) 类型以及用户定义的 class 类型。

new 和 delete 运算符的功能比 C 语言中的 malloc() 和 free() 标准库函数要强得多，不仅完全可取代后者，且更重要的是具有面向对象特征。new 为任意类型的对象在堆 (Heap) 中分配一块所需的存储空间，并返回该存储空间的首地址，有时也把该地址值称为 new 运算符的结果值。如果堆中没有足够的存储空间或分配出错时返回空 (NULL) 指针。因此，与使用 malloc() 函数一样，必须检测其返回值不为空指针。

例 2.6 运算符 new 和 delete 的用法。

```
#include    < iostream >           // 使用 C++新标准的流库
#include    < cstdlib >             // 将 std 标准名空间合并到当前名空间

using namespace std;

void main( void )                  // 采用 C 语言标准函数 malloc() 的典型格式
{
    int    x = 60 , * p;
    if((p = (int *)malloc(sizeof(int)))== NULL) {
        printf("Heap error !\n");
        exit(1);
    }                                // 异常终止路径

    if( ( p = new int ) == NULL ) {
        cout << "Heap error !\n";
        exit(1);
    }                                // 异常终止路径

    else * p = x; // 赋值操作，将 x 安全地保存在堆中
    cout << "Save a integer : " << * p << endl; // 在该程序段内
    // 通过指针 p 访问保存在堆中的变量 x      // 变量 x 被可靠保存

    delete p;
    cout << "Memory freed !\n";
}
```

正如例 2.6 那样，可采用赋值操作创建新的动态变量或动态对象，其格式为：

```
指针名 = new 类型;
```

用运算符 `new` 或 `malloc()` 函数创建的对象（或变量）称为“动态对象”（或动态变量），在例 2.6 中，写有：

```
p = new int;
```

/* 创建一个 `int` 型的动态变量，并将 `new` 运算符的结果值保存在指针变量 `p` 中，以后都是通过 `p` 指针来访问该动态变量 */

此时必须预先定义好同类型的指针，这种形式便于应用 `if` 语句，检测其返回值是否为空指针，一般格式为：

```
if((指针名 = new 类型) == NULL) { ... }
```

在例 2.6 中，写有：

```
if((p = new int) == NULL) { ... }
```

顺便指出，在以后的程序中该指针名用来代替所创建的动态对象名，该动态对象本身是匿名的。由于

	简化
<code>if(p == NULL) (即 if(p == 0))</code>	→ <code>if(!p)</code>
	简化
<code>if(p != NULL) (即 if(p != 0))</code>	→ <code>if(p)</code>

故一般格式可写为：

```
if (! (指针变量名 = new 类型)) { 出错处理操作; }
if ( 指针变量名 = new 类型 ) { 创建成功的处理操作; }
```

由于 C 和 C++ 都是以“非零值”和“零值”来界定“逻辑真”和“逻辑假”，因此，第 1 个条件语句圆括号内的表达式是简化形式，把它还原应该是“（指针名 = new 类型）== NULL”，即创建一个<类型>所指定的动态对象或动态变量，并将 `new` 运算符的结果值赋给左值指针变量保存起来，然后再判断该指针是否为空指针，若它确实是空指针，该条件表达式成立，则其结果值为“逻辑真”，它的值为整数 1，当然应该执行 `if` 分枝下的复合语句，该语句进行出错处理操作。而第 2 个条件语句还原后应该是“（指针名 = new 类型）!= NULL”，即判断该指针不是空指针，当它确实不是空指针时，则其结果值为“逻辑真”，它的值为整数 1，也就执行 `if` 分支下的复合语句，该语句当然进行创建成功的处理操作。有时，也可把定义同类型的指针变量和用 `new` 运算符创建新的动态对象同时进行，其格式为：

```
<类型> *指针变量名 = new <类型> (初值表);
```

即定义一个同类型指针变量保存 `new` 运算符的结果值与创建一个动态对象一气呵成，

但这就无法判断该指针是否为空指针。当编程者创建一个占用很小内存资源的动态对象或者有足够把握能成功创建时通常采用这种简单形式。

`new` 和 `delete` 优于 `malloc()` 和 `free()`，运算符 `new` 自动计算<类型>的大小而不需使用 `sizeof` 运算符，而且返回的指针能正确指向该类型，而不需作强制类型转换处理。更重要的是 `new` 和 `delete` 可用于用户定义的 `class` 类型，而 `malloc()` 和 `free()` 不能用来创建 `class` 类型的动态对象。

用 `new` 运算符也可创建一维动态数组并为它的元素在堆中分配存储空间，`new` 的结果值是数组第 0 号元素的地址，即该数组的首地址，其格式为：

<类型> * 指针变量名 = new <类型>[元素个数];

例如：

```
int * pa = new int[10];
```

即创建了具有 10 个元素的 `int` 型动态数组 `pa`，对单个变量也可以采用这种格式，只不过元素个数为 1。例如：

```
float * px = new float[1]; // 创建一个 float 型动态变量 px，但没有赋初值
float * py = new float[1]; // 再创建一个 float 型动态变量 py，但没有赋初值
```

用这种格式创建对象时，只是给对象分配了内存空间，并没有对它赋初值。因此在使用前，还必须用赋值操作对其赋值。例如：

```
* px = 12.6; // 给第 1 个动态变量 px 赋初值为 12.6
* py = 16.8; // 给第 2 个动态变量 px 赋初值为 16.8
```

采用地址赋值，把两个指针指向同一内存位置，将产生“别名 (Aliasing)”问题，用 `new` 运算符创建动态变量或对象时最容易产生。

例 2.7 自动变量的“别名”问题。

```
#include <iostream> // 使用 C++ 新标准的流库
using namespace std; // 将 std 标准名空间合并到当前名空间

void main( void )
{
    int a , * pa = &a; // 定义 int 型变量 a 和指针变量 pa，令 pa 指向 a
    int b , * pb = &b; // 定义 int 型变量 b 和指针变量 pb，令 pb 指向 b
    * pa = 28; // 通过指针 pa 给 a 赋值
    * pb = 64; // 通过指针 pb 给 b 赋值
    cout << " (1) a = " << * pa << " , b = " << * pb << endl;
    // 分别通过指针 pa 和 pb 读取变量 a 和 b 的值，输出到 CRT 上显示
    pa = pb; // 使指针 pa 和 pb 都指向了变量 b
    cout << " (2) a = " << * pa << " , b = " << * pb << endl;
    // 再通过指针 pa 和 pb 读取它们所指的内容，均是变量 b 的值
    * pa = 206; // 把 206 赋给 pa，所指的内容实际上是赋给变量 b
}
```

```
cout << " (3) a = " << * pa << " , b = " << * pb << endl;
// 通过指针 pa 和 pb 读取它们所指的内容,均是变量 b 的值
}
```

该程序的输出结果为：

```
(1) a = 28 , b = 64
(2) a = 64 , b = 64
(3) a = 206 , b = 206
```

如图 2.3 所示, 指针变量 `pa` 和 `pb` 分别指向两个 `int` 型变量 `a` 和 `b`, 并通过指针 `pa` 和 `pb` 分别给 `a` 和 `b` 赋值为 28 和 64。当执行了“`pa=pb;`”地址赋值语句后, `pa` 和 `pb` 指向了同一内存位置。因此, 再通过指针变量 `pa` 和 `pb` 访问的变量都是变量 `b`, 原来存放变量 `a` 的内存空间再也不可能通过指针变量 `pa` 和 `pb` 来访问了。但是, 不是用运算符 `new` 或 `malloc()` 函数创建的动态对象或动态变量自身还有对象名或变量名, 如自动变量还有变量名 `a`, 还可以用变量名 `a` 去访问这一内存空间。然而, 动态对象或动态变量的情况却大不相同, 我们用下面的例程加以说明。

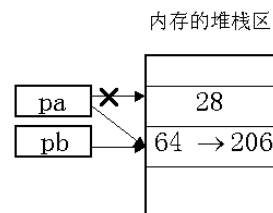


图 2.3 自动变量的别名问题

例 2.8 动态对象或动态变量的“别名”问题。

```
#include <iostream>
// 使用 C++ 新标准的流库

using namespace std;
// 将 std 标准名空间合并到当前名空间

void main( void )
{   int * pa = new int[1];
    // 定义一个 int 型的动态变量, 系统在堆中为它分配内存空间, 变量名就是 pa
    int * pb = new int[1];
    // 定义一个 int 型的动态变量, 系统在堆中为它分配内存空间, 变量名就是 pb
    * pa = 28;
    * pb = 64;
    cout << " (1) a = " << * pa << " , b = " << * pb << endl;

    pa = pb;
    // pa 和 pb 指向同一内存位置, 原来 pa 所指的内存空间被挂起, 再也无法访问
    cout << " (2) a = " << * pa << " , b = " << * pb << endl;
    * pa = 206;
    cout << " (3) a = " << * pa << " , b = " << * pb << endl;
    delete [ ] pa;
```



```
// delete [ ] pb; 对同一内存空间执行了两次 delete 操作, 产生非法操作
}
```

如图 2.4 所示, 由于动态对象或动态变量只有一个变量名, 即接受堆区内存空间首地址的指针变量名 `pa` 和 `pb`, 如果用地址赋值语句 “`pa = pb;`” 使它们指向同一内存位置, 将导致原来 `pa` 所指的内存空间被挂起, 再也无法访问该内存空间了, 即发生了内存泄漏 (memory leak, 详见文献 [26])。对于运行在图形用户界面的大型应用程序, 一个窗口或控件将占用不少容量的内存空间, 在堆中若 “内存挂起” 积累太多, 可能会导致堆中内存资源的耗尽, 编程者应避免这种情况的发生, 且用 `new` 创建对象时, 必须用前述方法测试 `new` 的返回值是否为空指针。

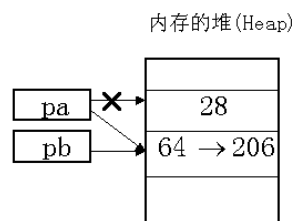


图 2.4 动态变量的别名问题

`new` 创建的对象, 其生存期是整个程序运行期间, 直到显式地用 `delete` 运算符撤销它。

2. 运算符 `delete` 的用法

`delete` 运算符只能作用于 `new` 返回的指针, 对于其他不是用 `new` 运算符获得的指针则没有意义。其格式为:

```
delete 指针名; // 该指针保存 new 分配的内存空间首地址
```

用 `delete` 运算符撤销由 `new` 创建的数组时采用如下格式:

```
delete [ ] 指向数组的指针名;
```

这里所说的数组可以是基本数据类型的数组, 或者是用户定义的 `class` 类型的对象数组。

对一个用 `new` 创建的动态对象只能使用一次 `delete` 操作。在例 2.8 中, 由于 `pa` 和 `pb` 指向了同一内存位置, 实质上它们所指的目标已变成了一个对象, 因此第 2 次用 “`delete [] pb;`” 再次撤销它时将产生非法操作。

`new` 和 `delete` 运算符在程序中是成对出现的, 即编程者用 `new` 运算符创建了一个动态对象或动态变量, 就必须要用 `delete` 运算符去撤销该动态对象或动态变量, 从而避免程序运行时发生内存泄漏问题。

2.5.3 C++ 的运算符集

表 2.2 列出了 C++ 运算符集的优先级和结合规则。说明如下:

C++ 保留了 C 语言丰富的运算符集, 并增加了 3 个新的运算符, 即 `new`、`delete` 和 `::` 作用域运算符。优先级和结合规则与 C 语言基本相同。

C++可以重新定义（或称“重载”）这些运算符，这将在以后章节中讲述。

在 ISO/ANSI C 老标准中，增量运算符“++”和减量运算符“--”只能用于整型量，而在 ISO/ANSI C++新标准中这两个运算符对整型和浮点型均可使用。

表 2.2 C++常用运算符的功能、优先级和结合规则

优先级	运算符	名 称	结合规则
1	()	函数参数表, 改变优先级	从左至右
		作用域运算符	
	[]	数组下标	
	· , ->	成员选择(访问)符	
	· * , -> *	成员指针选择符	
2	++ , --	增 1, 减 1 运算符	从右至左
	&	取地址	
	*	取内容	
	!	逻辑求反	
	~	按位求反	
	+ , -	取正数, 取负数	
	(数据类型)	强制类型转换	
	sizeof	计算数据类型长度	
	new , delete	动态存储分配	
3	* , / , %	乘法, 除法, 取余	从左至右
4	+ , -	加法, 减法	
5	<< , >>	左移位, 右移位	
6	< , <= , > , >=	小于, 小于等于, 大于, 大于等于	
7	== , !=	判断相等与不等	
8	&	按位求逻辑与	
9	^	按位求异或	
10		按位求逻辑或	
11	&&	逻辑与	
12		逻辑或	从右至左
13	? :	条件运算符	
14	= , += , -= , *= , /= , %= , &= , ^= = , <<= , >>=	赋值运算符	
15	,	顺序运算符	从左至右

2.6 C++的函数

函数是一个独立完成某种功能的程序块，它封装了一些程序代码和数据，实现了更高级的抽象和数据隐藏，使得编程者只关心函数的功能和使用方法，而不必关心函数功能的

具体实现细节。由于函数与函数之间通过输入参数和返回值（输出）来联系，因此可以把函数看做一个“黑盒（black box）”，除了输入输出外，其他的功能都封装隐藏在“黑盒”内，无需用户操心。

2.6.1 引用的应用

引用主要是作为函数的形参和函数的返回值。下面分别进行讨论。

1. 函数的引用调用

与C语言的函数一样，C++的函数也分为函数调用的值传递方式和地址传递方式，详细内容请参阅文献[25]。函数调用的地址传递方式可以通过改变形参指针所指向的变量值来影响实参，从而实现了数据的双向传递，这是函数间传递信息的一种重要手段。C++又引入了一种新的类型，即引用，使函数调用的地址传递方式更简单，使用更方便。

如前所述，引用是给一个已存在的变量启用一个替换名，对引用的操作等价于对被引用变量的操作。引用主要是作为函数的形参和函数的返回值。使用引用作为函数形参时，调用函数的实参要用变量名。函数调用时，将实参变量名赋给形参的引用，即对形参引用进行了初始化操作，实参就是被引用变量，相当于在被调用函数体内使用的形参就是实参的替换名，形参名和实参名代表的是同一个实体，于是在被调用函数体内，对形参引用进行操作改变其数值，实质上就是直接通过引用来改变实参的变量值。因此在函数调用时，就像接力赛跑一样，实参把接力棒传递给形参，在被调用函数体内形参就代表实参来完成规定的任务。由于引用与被引用变量是通过地址相联系的，所以引用调用实质上仍然是地址传递方式。所不同的是，实参变量对形参引用的初始化操作被当做值本身来传递，不必写取地址运算和取内容运算，因此更直接、更方便。所以，在C++中经常使用引用作为函数的参数，从而在被调用函数中改变调用函数的实参值，实际上是让函数返回了多个结果值。

例 2.9 函数的引用调用实现数据的双向传递。

```
#include <iostream>           // 使用 C++ 新标准的流库
using namespace std;          // 将 std 标准名空间合并到当前名空间
/* 把文献[25]中的 swap2(int * x, int * y) 的形参指针改成形参引用，写成
   swap3(int & x, int & y)，同样是函数调用的地址传递方式 */
void swap3( int & x, int & y )
{ // 形参引用 x 是实参 a 的替换名，形参引用 y 是实参 b 的替换名，不必采用取内容运算
  int temp = x;
  x = y;
  y = temp;
  cout << "In swap3( ) : x = " << x << " , y = " << y << endl;
  // 输出显示 x 和 y 的值，分别是实参 a 和 b 的值
```

```

}
void main( void )
{
    int a(5), b(9);
    cout << "Before called swap3( ): a = " << a << ", b = " << b << endl;
    swap3(a, b);    // 形参是引用，实参不必采用取地址运算
    cout << "After called swap3( ): a = " << a << ", b = " << b << endl;
}

```

该程序的输出结果为：

Before called swap3(): a = 5 , b = 9

In swap3() : x = 9 , y = 5

After called swap3(): a = 9 , b = 5

由此可见 (参看图 2.5):

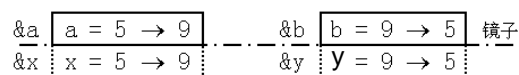


图 2.5 函数引用调用的参数传递

在引用调用中，实参用变量名，形参用引用。

调用时系统将实参的变量名赋给对应的形参引用，相当于执行了引用的初始化操作语句“int & x = a;”和“int & y = b;”。所以，在 swap3() 函数体内，对形参引用 x 和 y 所进行的交换操作，就等价于对实参变量 a 和 b 进行的操作，即在 main() 函数体内的 a 和 b 值也进行了交换。

凡是用指针作形参进行地址传递的函数，都可以将形参指针改为“形参引用”，函数调用语句中的实参就可以直接写变量名；被调用函数体内的形参可像变量一样使用，而不必使用取地址运算和取内容运算。今后，读者可以把原来 C 语言的老程序，先将编辑文件名中的扩展名“.C”改成“.CPP”，再按照这一修改办法修改成执行效率高且结构简单的程序格式。

2. 引用作为函数的返回值

返回引用的函数是非常有用的，该函数的调用表达式可以写在赋值运算符的左边，作为“左值表达式”，这类函数也可对它直接进行赋值、增量、减量等操作。因此，在 C++ 中，函数可返回一个引用，返回的引用作为左值直接进行增量和减量运算。下面举一个例子加以说明，若程序中写有一个 elem1() 函数：

```

char elem1( char * s , int n )
{
    return s[n];
}

```

则该函数执行一次带下标的操作，它读取一个字符数组，返回值为第 n 个字符，即读取第 n 个字符。这类函数不能写在赋值运算符的左边，成为“左值表达式”。也就是说，

elem1()函数只能读取第 n 个字符,而不能改写第 n 个字符。即

```
char *str = "VC++V5.0";
char ch = elem1(str, 5);
elem1(str, 10) = '6';    // 出错
```

如果将 elem1()函数改成 elem2(),利用它的返回值来改变数组某元素的值,即改写第 n 个字符,可使该函数返回一个指向第 n 个字符的指针,即把 elem1()函数改写成指针函数 elem2(),以解决这一问题。

例 2.10 指针函数 elem2()可作为“左值表达式”。

```
#include <iostream>    // 使用 C++新标准的流库
using namespace std;   // 将 std 标准名空间合并到当前名空间
// 指针函数 elem2( )的返回值是形参指针 s 所指字符串的第 n 个字符的地址
char * elem2(char * s, int n)
{    return &s[n];    }
void main( void )
{    char * c;
    // 为便于理解,定义一个字符串指针 c ,接受指针函数 elem2( )的返回值
    char str[ ] = "VC++V5.0";
    cout << "原来的字符串: " << str << endl;
    c = elem2(str, 5);    // 此两条语句可合并成一条
    * c = '6';            // 写成 "*elem2(str, 5) = '6';"
    cout << "修改后的字符串: " << str << endl;
}
```

该程序的输出结果为:

原来的字符串: VC++V5.0

修改后的字符串: VC++V6.0

正如图 2.6 所示,由于函数 elem2()的返回值是第 5 号元素的地址&s[5],把&s[5]赋给了字符指针 c,即指针 c 指向了字符串数组 str 的第 5 号元素'5',再用赋值语句“* c = '6';”把该字符改成'6'。其实可以把函数调用表达式 elem2(str, 5)直接写在赋值运算符的左边,作为“左值表达式”,这样不必通过字符指针 c 就可把它从程序中删掉,将两条赋值语句合并成一条语句:

```
*elem2(str, 5) = '6';
```

然而,在 C++中还有更好的方法。该函数不用传回一个指针,而返回一个引用,它的调用表达式也可写在赋值运算符的左边。

例 2.11 把返回引用的函数作为“左值表达式”。

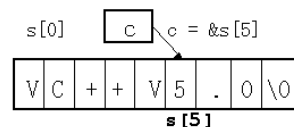


图 2.6 “c = elem2(str, 5);”的执行过程

```

#include    < iostream >           // 使用 C++新标准的流库
using namespace std;              // 将 std 标准名空间合并到当前名空间
// 返回引用的函数 elem( ), 其返回值就是形参指针 s 所指字符串的第 n 个字符
char & elem(char * s , int n)
{
    return s[n];
}
void main( void )
{
    char str[ ] = "VC++V5.0";
    cout << "原来的字符串 : " << str << endl;

    elem(str , 5) = '6';
    cout << "修改后的字符串 : " << str << endl;
}

```

返回引用的函数，其定义格式为：

<pre> 存储类 类型 & 函数名 (形参表) { ... return 表达式; } </pre>
--

与一般函数的定义格式类同，只是其中的“类型 &”指明函数的返回值为某种数据类型的引用，在函数体内与之对应有一个“return 表达式；”语句。当然，正如下面的例程那样，也可以有多个这样的 return 语句，它们大多出现在选择性的流程控制语句（如 if, switch）中。

这种返回引用的函数，只能用全局变量或静态变量以及全局对象或静态对象作返回值，而不能用该函数的自动变量或自动对象，因为自动变量或自动对象在该函数体外自动消失。也就是说，return 语句的表达式只能写全局变量名或静态变量名以及全局对象或静态对象，而不能写自动变量名。

例 2.12 返回引用的函数必须返回非自动型的变量或对象。

```

1: #include < iostream >           // 使用 C++新标准的流库
2: using namespace std;           // 将 std 标准名空间合并到当前名空间
3: int array[6] = { 66, 78, 86, 92, 88, 96 };
   // 定义外部型数组 array，存放 6 个学生某课程的成绩
4: int gradeA = 0 , gradeB = 0;
   // 定义外部型的变量 gradeA 和 gradeB，分别记录 A 类和 B 类学生的人数
   // level( )函数的返回值是 gradeA 或 gradeB，它们是外部变量
5: int & level( int k )
6: {   if( array[k] >= 80 )           // 对数组 array 的第 k 号元素进行判断
7:     return gradeA;                 // A 类学生的人数
8:     else
9:     return gradeB;                 // B 类学生的人数

```

```

10: }
11: void main( void )
12: {   for( int i = 0; i < 6; i++ )
13:     level( i ) ++;
    // level( )是返回引用的函数，可以直接进行增量运算
14:     cout << "number of gradeA is " << gradeA << endl;
15:     cout << "number of gradeB is " << gradeB << endl;
16: }

```

该程序是统计学生中 A 类学生和 B 类学生各占多少。A 类学生的标准是成绩在 80 分以上，其余都是 B 类学生。6 位学生的成绩存放在 array[]数组中。其输出结果为：

```

number of gradeA is 4
number of gradeB is 2

```

由于 level()函数返回的是引用，所以可以作为“左值表达式”直接进行增量操作。其返回语句中的 gradeA 和 gradeB 都是全局变量，这是方法 1。

level()是返回引用的函数，实际上是获得一个某数据类型变量的引用，而该变量的引用又是匿名的，为了便于解释，我们用 x 代表该变量的匿名引用名，而函数体内 return 语句后面表达式的结果值就是该函数的返回值。因此，当函数返回时就是用 return 语句后面表达式的结果值去初始化匿名的引用变量 x，即：

```

int & X = (   if(array[k] >= 80)
              return gradeA;
            else
              return gradeB; )

```

其右边用圆括号包围的是一个条件语句，由此可见，匿名的引用变量 x 初始化操作是受一个条件语句的控制，即当“array[k] >= 80”成立（成绩在 80 分以上）则 x 的被引用变量是 gradeA，若不成立（成绩低于 80 分）则匿名的引用变量 x 做加 1 运算，那么，也就是对它的被引用变量做增量运算。所以，“level(i) ++;”语句完成的功能是当成绩在 80 分以上则 gradeA 加 1，否则 gradeB 加 1。

全局变量会导致模块间产生强耦合是控制使用的变量类型，通常都是由一个软件开发组顶层负责人从软件工程项目的整体考虑来决定是否采用哪几个全局变量。因此，凡是能消除的全局变量都应该消掉。为此可采用下面的方法 2，即需要给 level()函数传递两个引用参数。为此，给它增设两个形参引用，即引用 int & gA、int & gB，分别作为两个实参 gradeA 和 gradeB 的替换名，将它们传递给被调用函数 level()。因此，这两个形参必须定义成引用，而不能定义为 int gA、int gB，即实参传递给形参不能采用值传递方式，而必须采用地址传递方式。

例 2.13 把 gradeA 和 gradeB 改为自动变量的方法 2。


```

#include    < iostream >           // 使用 C++新标准的流库
using namespace std;              // 将 std 标准名空间合并到当前名空间

int array[8] = { 66, 78, 86, 92, 78, 96, 70, 76 };
// 定义外部型数组 array, 存放 8 个学生某课程的成绩
// 返回引用的 level( )函数可以直接作为“左值表达式”

int & level( int k , int & gA , int & gB )
{
    if( array[k] >= 80 )
        return gA;    // 等价于 return gradeA;
    else
        return gB;    // 等价于 return gradeB;
}

void main( void )
{
    int gradeA = 0 , gradeB = 0;
    // 定义自动型变量 gradeA 和 gradeB , 分别记录 A 类和 B 类学生的人数
    for(int i = 0; i < 8; i++)
        level(i , gradeA , gradeB) = level(i , gradeA , gradeB) + 1;
    // 也可以写成“level( i , gradeA , gradeB ) ++;”

    cout << "number of gradeA is " << gradeA << endl;
    cout << "number of gradeB is " << gradeB << endl;
}

```

该程序的输出结果为：

```

number of gradeA is 3
number of gradeB is 5

```

由于函数的引用调用是地址传递方式的一种，当执行“level(i , gradeA , gradeB);”语句时，在实参传递给形参的过程中，相当于执行了“int k = i;”、“int & gA = gradeA;”和“int & gB = gradeB;”等形参初始化操作语句，使得形参引用 gA 和 gB 分别初始化成 gradeA 和 gradeB 的替换名，在 level()函数体内，对 gA 的运算和操作就等价于对 gradeA 的运算和操作，对 gB 的运算和操作也等价于对 gradeB 的运算和操作，因此对 level()进行增量操作，就等价于对 gradeA 或 gradeB 的增量操作。读者也可以思考如下两个问题：

如果把外部型数组 array 也改成自动型，应如何修改程序。

若使用内部静态型的数组和变量，应如何修改程序。

2.6.2 设置函数参数的默认值

在 C++中，允许在函数原型的声明语句中（最好是在声明语句中）或定义函数时，给一个或多个参数指定默认值（缺省值 Default）。以后在调用该函数时，若省略其中某些

实参则系统以它们的默认值作为这些参数的实参值。

例 2.14 设置函数参数的默认值。

```
#include <iostream> // 使用 C++新标准的流库
using namespace std; // 将 std 标准名空间合并到当前名空间
int m(8); // 定义外部变量 m , 并赋初值为 8
int & n = m; // n 为 m 的引用, 也可用来作为函数参数的默认值
int addInt(int x, int y = 7, int z = 2 * n);
void main( void )
{
    int a(5), b(15), c(20);
    int s = addInt( a, b );
    // a( = 5), b( = 15) 分别传递给形参 x 和 y, 形参 z 取用默认值 2 * n( = 16)
    cout << "SUN = " << s << endl;
}
int addInt( int x, int y, int z )
{
    return x + y + z;
}
```

函数参数的默认值是在形参表内用常量表达式、已经赋值的变量或任意表达式来指定的。在例 2.14 中, 参数 *z* 采用表达式指定默认值。

```
int m(8);
int & n = m;
int addInt(int x, int y = 7, int z = 2 * n);
```

在一个函数中, 可指定多个参数的默认值, 甚至指定全部参数的默认值。但必须是从右至左连续指定, 即指定和不指定不能交叉进行。例如:

```
int addInt(int x = 6, int y = 7, int z = m);
```

以下操作非法:

```

      左      ←      右
int addInt( int x, int y = 7, int z );
int addInt( int x = 6, int y, int z );
```

第 1 条语句的形参表中, 从右至左第 1 个形参“*int z*”没有指定默认值, 而第 2 个形参“*int y = 7*”又指定了默认值, 第 3 个形参“*int x*”又没有指定。第 2 条语句的形参表中, 从右至左第 1 个和第 2 个形参都没有指定默认值, 而第 3 个形参又指定了默认值, 所以它们都是非法操作。从第 2 条语句可得出一条推论: 若函数形参表中最左边的参数指定了默认值, 则所有的形参都必须指定默认值。

当进行函数调用, 实参数目少于形参时, 编译系统按从左至右的顺序用默认值来补足所缺少的实参。

例 2.15 使用函数参数的默认值。

```
#include <iostream> // 使用 C++新标准的流库
```

```

using namespace std;           // 将 std 标准名空间合并到当前名空间
void fun(int a = 1, int b = 3, int c = 5)
{   cout << "a = " << a << " , b = " << b << " , c = " << c << endl; }
void main( void )
{
    // 画有虚线的参数均采用默认值
    fun( );           // 函数参数选取为: a = 1, b = 3, c = 5
    fun(7);           // 函数参数选取为: a = 7, b = 3, c = 5
    fun(7, 9);        // 函数参数选取为: a = 7, b = 9, c = 5
    fun(7, 9, 11);    // 函数参数选取为: a = 7, b = 9, c = 11
    cout << "OK!\n";
}

```

该程序的输出结果为：

```

a = 1 , b = 3 , c = 5
a = 7 , b = 3 , c = 5
a = 7 , b = 9 , c = 5
a = 7 , b = 9 , c = 11
OK!

```

调用时，也必须是从右至左连续采用默认值，“采用”和“不采用”也不能交叉进行。若第 1 个参数采用默认值，则其后的参数必须都采用默认值，如“fun(7,9,11);”。也可以是第 1 个参数不采用默认值，从第 2 个参数开始采用默认值，如“fun(7);”。但不能写成：

```

fun( ,9);           // 出错，形参 a 采用默认值，而参数 b 不采用，参数 c 又采用

```

2.6.3 内联函数

1. 内联函数(Inline Function)引入的原因

函数调用时所做的准备工作要占用 CPU 资源，必然导致一些额外的时间开销而影响程序执行速度。例如，调用时要保护现场，将参数压入堆栈，并查到被调用函数的入口地址，同时把返回地址压入堆栈。调用结束时，要恢复现场并从堆栈中弹出返回值和返回地址，实现返回。显然，函数调用要有一定的时间和空间的开销，从而影响执行效率。对于一些函数体代码不大，但又频繁被调用的函数，C++的设计者为编程者提供了“内联函数”，以解决这类函数的执行效率问题。

2. 内联函数的定义方法

在学习“微型计算机原理”课程时，其内有关“汇编语言编程方法”中曾提到过“子程序(subroutine)”的概念，即当完成某个功能的一段(由多条汇编指令组成)汇编语言程序多处采用时，可将这段汇编语言程序设定成一个子程序，例如取名为 A，以后每写一条“CALL A”(调用子程序 A)指令就可代替这段汇编语言程序，从而大大简化了程序

结构，节省了所占用的内存空间。但这是有代价的，它是用子程序的调用过程占用 CPU 执行时间来换取所占用的内存空间减小，简称“以时间换空间”。然而，内联函数的思想方法是反其道而行之，即以空间换时间。当定义函数时，在函数头的前面加上关键字“inline”，则指明该函数为内联函数。那么，在编译时该函数不被编译为单独一段可调用的代码，而是将生成的函数体代码模块（这相当于把一条“CALL A”指令还原成子程序 A 所包含的段汇编语言程序）直接插入到该函数的每个调用处，从而避免了函数调用时的额外开销，但是程序代码的容量将增大，所占用的内存空间增多。例如：

```
#include    < iostream >           // 使用 C++新标准的流库
using      namespace  std;         // 将 std 标准名空间合并到当前名空间

inline int  add( int a, int b )
{   return a + b;   } ← 函数体代码块

void  main( void )
{   int  x = add( 2, 4 ); ← [调用处]
    cout << "x = " << x << endl;
}
```

3. 使用内联函数应注意的事项

在内联函数内不允许用循环语句和开关语句。因为执行循环语句和开关语句所消耗的时间较长，则调用函数准备工作所需要的时间占整个调用函数的执行时间的比例就很小，因此采用内联函数的作用就不大。

内联函数必须在使用之前定义好。因为内联函数必须预先编译生成了函数体代码块后，才能插入到程序的调用处。

内联函数将使程序代码容量增大。因此，内联函数只适合于有 1 ~ 5 行的小函数，对一个含有许多语句的大函数，函数的调用和返回的开销相对来说是微不足道的，所以也就没有必要用内联函数来实现。

递归函数需自己调用自己，编译时无法生成函数体代码模块，而不能定义成内联函数。

4. 使用内联函数取代#define 预处理语句定义的宏指令

C 语言中可以用#define 语句定义宏指令，特别是带参数的宏指令，例如：

```
#define  MAX(x, y)  ((x) > (y)) ? (x) : (y)
```

这里用#define 语句定义带参数的宏指令 MAX(x, y)，取出两者中的最大者。如前所述，由于预处理器（preprocessor）不是程序的组成部分，没有较好的检查机制，造成在宏指令内隐藏着程序瑕疵，下面用例程加以说明。

例 2.16 用#define 语句定义带参数的宏指令，求出两个整数中的最大值。

```
#include    < iostream >           // 使用 C++新标准的流库
using      namespace  std;         // 将 std 标准名空间合并到当前名空间
```

```

#define    MAX(x, y)    ((x) > (y)) ? (x) : (y)
void main()
{    int    a(7), b(0), c, d;        // 定义4个自动型的变量a、b、c和d
cout << " (1) a = " << a << " , b = " << b << endl;
// 输出显示变量a和b的值分别为7和0
c = MAX( ++a, b ); // 从后面输出结果可知, a 变量值被增量两次为9, 这显然是错误的
cout << " (2) a = " << a << " , b = " << b << " , max = " << c << endl;
/* 输出显示变量a、b和c(它是++a和b中的最大值)的值分别为9、0和9, 说明变量a被
   错误地增量两次 */
d = MAX( ++a, b + 12 );
/* 从后面输出结果可知, a 变量值增量一次为10, 与另一个实参b + 12( = 12) 比较,
   因后者大取出它作为宏指令的结果值赋给变量d */
cout << " (3) a = " << a << " , b = " << b << " , max = " << d << endl;
// 输出显示变量a、b和d(它是“++a”和“b + 12”中的最大值)的值分别为10、0和12
}

```

该程序的输出结果为：

```

(1) a = 7 , b = 0
(2) a = 9 , b = 0 , max = 9           (变量a被错误地增量两次)
(3) a = 10 , b = 0 , max = 12        (变量a增量一次)

```

显然, 在应用宏指令“MAX(++a, b)”时, 变量 a 被错误地增量了两次, 这是一种隐藏在预处理语句中、不易被发现的程序瑕疵, 充分说明预处理器 (preprocessor) 没有很好的检查机制。因此, 熟悉 ISO/ANSI C 老标准的编程者要想尽快掌握 ISO/ANSI C++新标准, 应牢记前述的“尽量以编译器 (compiler) 取代预处理器 (preprocessor)”的编程原则, 对于用#define 语句定义的带参数宏指令, 使用内联函数取代它, 例 2.17 就是把例 2.16 中, 用#define 语句定义的带参数宏指令用内联函数取代之。

例 2.17 用内联函数取代用#define 语句定义的带参数宏指令。

```

#include    <iostream>                // 使用 C++新标准的流库
using namespace std;                // 将 std 标准名空间合并到当前名空间
inline int max( int x, int y )
{    return x > y ? x : y; }
void main( void )
{    int    a(7), b(0), c, d;        // 定义4个自动型的变量a、b、c和d
        cout << " (1) a = " << a << " , b = " << b << endl;
// 输出显示变量a和b的值分别为7和0
        c = max( ++a, b );
// 调用内联函数 max(), 消除了隐藏在用#define 语句定义的带参数宏指令中的程序瑕疵
        cout << " (2) a = " << a << " , b = " << b << " , max = " << c << endl;

```

```

/* 输出显示变量 a、b 和 c (它是++a 和 b 中的最大值) 的值分别为 8、0 和 8, 变量 a 只被增量
一次 */

    d = max( ++a, b + 12 );
// 再次调用内联函数 max(), 求出 “++a” 和 “b + 12” 中的最大值

    cout << " (3) a = " << a << " , b = " << b << " , max = " << d << endl;
// 输出显示变量 a、b 和 d (它是 “++a” 和 “b + 12” 中的最大值) 的值分别为 9、0 和 12
}

```

该程序的输出结果为：

```

(1) a = 7 , b = 0
(2) a = 8 , b = 0 , max = 8
(3) a = 9 , b = 0 , max = 12

```

从程序的输出结果可知，采用内联函数 `max()` 取代用 `#define` 语句定义的带参数宏指令 `MAX(x, y)`，可以避免因预处理器没有很好的检查机制而产生的程序隐患，因此，尽量用内联函数取代用 `#define` 语句定义的带参数宏指令。

2.6.4 函数重载

所谓“函数重载”是指在同一作用域内多个函数体可以使用相同的函数名，这类函数称为“重载函数”。众所周知，一个函数所具有的功能都是靠顺序执行函数体内一条条语句来实现的，即一个函数的函数体就是该函数的实现版本。因此，一个重载函数的函数体通常称为它的一个实现版本。所以，“重载函数”是在同一作用域内具有多个实现版本的一系列同名函数。

例 2.18 普通函数的重载。

```

#include <iostream>           // 使用 C++ 新标准的流库
using namespace std;         // 将 std 标准名空间合并到当前名空间
// 参数为字符串的实现版本
void print( char * str )
{   cout << "print(char *) called !\n    str = \" ";
    cout << str << "\"\n";
}
// 参数为 int 型的实现版本
void print( int value )
{   cout << "print(int) called !\n    value = ";
    cout << value << endl;
}
// 参数为 long 型的实现版本
void print( long value )
{   cout << "print(long) called !\n    value = ";

```

```
        cout << value << endl;
    }
    // 参数为 double 型的实现版本
void    print( double  value )
{   cout << "print(double) called !\n    value = ";
    cout << value << endl;
}

void    main( void )
{   int    a(6);
    float  b(8.8f);
    cout << "(1)";
    print( "How are you ?" );    // 调用 void print(char * str)
    cout << "(2)";
    print( double(a) );          // 调用 void print(double value)
    cout << "(3)";
    print( long(a) );            // 调用 void print(long value)
    cout << "(4)";
    print( a );                  // 调用 void print(int value)
    cout << "(5)";
    print( 1.6 );                // 调用 void print(double value)
    cout << "(6)";
    print('a');                  // 调用 void print(int value)
    cout << "(7)";
    print(3.14159);              // 调用 void print(double value)
    cout << "(8)";
    print(b);                    // 调用 void print(double value)
}
```

该程序的输出结果为：

```
(1) print(char *) called !
    str = "How are you ?"
(2) print(double) called !
    value = 6
(3) print(long) called !
    value = 6
(4) print(int) called !
    value = 6
(5) print(double) called !
    value = 1.6
(6) print(int) called !
```

```

    value = 97
(7) print(double) called !
    value = 3.14159
(8) print(double) called !
    value = 8.8

```

通常对于完成不同任务的函数应该选择不同的函数名，但对于那些针对不同数据类型的对象完成相同任务的函数，若启用同样的函数名会给调用这些函数带来很大的方便。如例 2.18 中把分别输出字符串、整数和双精度浮点数的 3 个函数都取名为 `print()`，编译系统通过比较实参和形参的个数和类型，来决定调用正确的重载函数完成相应的操作。编程者还可以把 `print()` 函数扩充到输出的对象是向量和矩阵。

1. 重载函数的使用说明

重载函数间首先用函数参数的不同个数加以区分，然后再用参数类型加以区分，其中至少要有有一个参数类型不同，并且应确保参数类型确实不同。例如，用 `typedef` 语句定义的数据类型只是一个已有类型的替换名，并没有创建新的数据类型，所以如下的程序段是错误的：

```

typedef int INT;
void func(int x);
void func(INT x);          // 出错，func()函数重复定义

```

函数返回类型的不同并不能区分重载函数，如下声明是非法的：

```

int process(int i);
void process(int i);

```

普通重载函数（非成员函数）的作用域是一个源文件。

2. 重载函数的匹配规则

当调用一个重载函数如 `print()` 时，编译系统必须搞清函数名究竟是指哪一个函数，这是通过把实参个数和类型与所有被调用的 `print()` 函数的形参个数和类型一一比较来判定的，其查找顺序为：

首先寻找与实参的个数、类型完全相同的函数，如果找到了就调用这个函数。

寻找一个严格匹配的函数，如果找到了就调用这个函数。对于 `int` 型的形参，对应的实参为数值 0、`char` 型、`short int` 型，都是 `int` 型形参的严格匹配。对于 `double` 型的形参，对应的实参为 `float` 型也是严格匹配。如例 2.18 中的“`print('a');`”语句是调用“`void print(int value)`”函数。

按照 C++ 内部类型转换规则（参阅文献 [25]），寻找一个匹配的函数，如果找到了就调用这个函数。

小 结

C++的常量和变量都具有数据类型,包括基本数据类型和复杂数据类型,还有标准类库中和用户定义的 `class` 类型,统称为“类型”。实际编程时,应尽量使用 `const` 替代 `#define` 定义符号常量。

C++的指针分为常量指针和指针变量,C++也完全保留了 C 语言指针变量的一些特性。编程时应特别注意两个同类型指针指向内存中同一个地址位置的“别名”问题。

引用是 C++新增加的数据类型,它是被引用变量的一个替换名,主要用于调用函数时传递参数和返回值,由于它是采用地址传递方式,因此引用完全可以取代指针完成函数间信息的双向传递,而在书写格式上却更简洁方便。与传值调用方式相比较,它不需要为实参复制副本,节省了时间和空间的开销,而在书写上惟一的区别是把形参指定为引用,由于引用与被引用变量间是通过地址相联系的,也可以说,引用是存放变量或某类对象内存空间地址的另一种更高级的表示形式。

C++用新增加的一对运算符 `new` 和 `delete` 实现动态存储技术,可完全取代 C 语言标准库函数 `malloc()` 和 `free()`,且用法更灵活方便,更重要的是能用来创建和撤销 `class` 类型的动态对象,实际编程时,应尽量采用运算符 `new` 和 `delete` 取代 `malloc()` 和 `free()` 函数。

C++允许把频繁调用的小代码函数指定为内联函数,以提高执行速度,但这是用空间(程序容量)来换取时间(执行速度)。实际编程时,应尽量采用内联函数取代用 `#define` 语句定义的带参数宏指令。

C++允许在同一个作用域内,一个函数名可以有多个不同的函数体,即可以定义一个函数的多个不同实现版本。这是简化程序设计的一种多态性形式。

习 题 2

一、选择填空

1. 在 16 位机中, `int` 型字宽为()字节。
A. 2 B. 4 C. 6 D. 8
2. 类型修饰符 `unsigned` 修饰()类型是错误的。
A. `char` B. `int` C. `long int` D. `float`
3. 对于 `int * pa[5];` 的描述,()是正确的。
A. `pa` 是一个指向数组的指针,所指向的数组是 5 个 `int` 型元素
B. `pa` 是一个指向某数组中第 5 个元素的指针,该元素是 `int` 型变量
C. `pa[5]` 表示某个数组的第 5 个元素的值
D. `pa` 是一个具有 5 个元素的指针数组,每个元素是一个 `int` 型指针
4. 下列关于指针的运算中,()是非法的。

14. 下面程序的输出结果是()。

```
#include <iostream>
using namespace std;
void main( void )
{
    int a = 2, b = -1, c = 2;
    if( a < b )
        if( b < 0 ) c = 0;
    else c += 1;
    cout << " c = " << c << endl;
}
```

A. c = 0 B. c = 1 C. c = 2 D. c = 3

15. 下面 for 语句的循环次数为()。

```
for( int i = 0, x = 0; !x && i <= 5; i++ )
```

A. 5 次 B. 6 次 C. 7 次 D. 无穷次

16. 下列关于 C++运算符结合规则的描述中()是正确的。

A. 赋值运算符是从左至右结合
B. 迭代 (复合) 赋值运算符是从左至右结合
C. 单目运算符是从左至右结合
D. 双目运算符是从左至右结合

17. 下列关于左值 (LValue) 的描述中()是错误的。

A. 程序中的左值可以被访问和修改
B. 赋值或迭代 (复合) 赋值运算符的左边必须是左值
C. (字面) 常量的值不能改变, 因此 (字面) 常量是左值
D. 变量在内存中有对应的存储单元, 因此变量是左值

18. 下列关于类型转换的描述中()是错误的。

A. 若 a 为 int 型变量, b 为 char 型变量, 则 a + b 的值为 int 型
B. 若 a 为 float 型变量, b 为 int 型变量, 则 a - b 的值为 float 型
C. 若 a 为 double 型变量, b 为 float 型变量, 则 a * b 的值为 double 型
D. 若 a 为 int 型变量, b 为 int 型变量, 则 a / (double)b 的值为 int 型

19. 下面程序片段将输出字符 '*' 的个数为()。

```
#include <iostream>
using namespace std;
void main( void )
{
    int i = 100;
    while( 1 ) {
        i--;
```

```

        if( i == 0 )    break;
        cout << '*';
    }
}

```

- A. 98 个 B. 99 个 C. 100 个 D. 101 个

20. 下面程序的输出结果是()。

```

#include <iostream>
using namespace std;
void main( void )
{
    int a;
    for( int i = 2; i < 6; i += 2 ) {
        a = 1;
        for( int j = i; j < 6; j++ )
            a += j;
    }
    cout << a << endl;
}

```

- A. 9 B. 1 C. 11 D. 10

21. 下面程序的输出结果是()。

```

#include <iostream>
using namespace std;
void main( void )
{
    int a = 10;
    while( a > 7 ) {
        a--;
        cout << a << " , ";
    }
    cout << endl;
}

```

- A. 10 , 9 , 8 , B. 9 , 8 , 7 ,
 C. 10 , 9 , 8 , 7 , D. 9 , 8 , 7 , 6 ,

22. 下面程序的输出结果是()。

```

#include <iostream>
using namespace std;
void main( void )
{
    int a;
    for( int i = 1; i <= 100; i++ ) {
        a = i;
    }
}

```

```
        if( ++a % 2 == 0 )
            if( ++a % 3 == 0 )
                if( ++a % 7 == 0 )
                    cout << a << " , ";
    }
    cout << endl;
}
```

A. 39 , 81 , B. 42 , 84 , C. 26 , 68 , D. 28 , 70 ,

23. 试有如下程序：

```
#include <iostream>
using namespace std;
void main( void )
{   int  ** pp, * p, a = 10;
    p = & a;
    pp = & p;
    cout << ** pp + 1 << endl;
}
```

其输出结果是()。

A. p 的地址 B. pp 的地址 C. 11 D. 运行错误

24. 下面程序的输出结果是()。

```
#include <iostream>
using namespace std;
void main( void )
{   int  a[ ] = { 1, 3, 5, 7, 9, 11 }, * p, ** pp;
    p = a;
    pp = & p;
    cout << * (p++) << " , " << ** pp << endl;
}
```

A. 3 , 3 B. 1 , 1 C. 3 , 5 D. 1 , 3

25. 下面程序的输出结果是()。

```
#include <iostream>
using namespace std;
void fun( int i );
int main( void )
{   int  a = 2;
    fun( a );
    cout << endl;
    return 1;
}
```

```

}
void fun( int i )
{
    static int j = 1;
    int k = 1;
    if( i > 0 ) {
        ++ j;
        ++ k;
        cout << j << " " << k << " ";
        fun( i - 1 );
    }
}

```

- A. 语法错，程序无法通过编译 B. 2 2 2 2
C. 2 2 3 3 D. 2 2 3 2

26. 下面程序的输出结果是()。

```

#include <iostream>
using namespace std;
int fun( char * s );
int main( void )
{
    cout << fun( "Hello" );
    cout << endl;
    return 1;
}
int fun( char * s )
{
    char * t = s;
    while( * t != '\0' )
        ++ t;
    return (t - s);
}

```

- A. 语法错，程序无法通过编译 B. 5
C. 6 D. 0

27. 若一个函数无返回值，则定义它时函数的类型应是()。

- A. void B. 任意 C. int D. 无

28. 在函数说明时，下列()项是不必要的。

- A. 函数的类型 B. 函数参数类型和名字
C. 函数名字 D. 返回值表达式

29. 在函数的返回值类型与返回值表达式的类型的描述中，()是错误的。

- A. 函数返回值的类型是在定义函数时确定，在函数调用时是不能改变的

- B. 函数返回值的类型就是返回值表达式的类型
 - C. 函数返回值表达式类型与函数返回值类型不同时,表达式类型应转换成函数返回值类型
 - D. 函数返回值类型决定了返回值表达式的类型
30. 在一个被调用函数中,关于 return 语句使用的描述,()是错误的。
- A. 在被调用函数中可以不用 return 语句
 - B. 在被调用函数中可以使用多个 return 语句
 - C. 在被调用函数中,如果有返回值,就一定要有 return 语句
 - D. 在被调用函数中,一个 return 语句可返回多个值给调用函数
31. 下列的()是引用调用。
- A. 形参是指针,实参是地址值
 - B. 形参和实参都是变量
 - C. 形参是数组名,实参是数组名
 - D. 形参是引用,实参是变量
32. 在 C++ 中,关于下列设置参数默认值的描述中,()是正确的。
- A. 不允许设置参数的默认值
 - B. 设置参数默认值只能在定义函数时设置
 - C. 设置参数默认值时,应先设置右边的再设置左边的
 - D. 设置参数默认值时,应该全部参数都设置
33. 重载函数在调用时选择的依据中,()是错误的。
- A. 参数个数
 - B. 参数类型
 - C. 函数名字
 - D. 函数的类型
34. 在一个函数中,要求通过函数来实现一种不太复杂的功能,并且要求加快执行速度,选用()合适。
- A. 内联函数
 - B. 重载函数
 - C. 递归调用
 - D. 嵌套调用
35. 采用函数重载的目的在于()。
- A. 实现共享
 - B. 减少空间
 - C. 提高速度
 - D. 使用方便,提高可读性
36. 在将两个字符串连接起来组成一个字符串时,选用()函数。
- A. strlen()
 - B. strcpy()
 - C. strcat()
 - D. strcmp()
37. 说明语句“const char * ptr;”中,ptr 应该是()。
- A. 指向字符常量的指针
 - B. 指向字符的常量指针
 - C. 指向字符串常量的指针
 - D. 指向字符串的常量指针

二、判断下列描述的正确性,对者划 ,错者划

1. C++ 的程序中,对变量一定要先说明再使用,说明只要在使用之前就可以。

2. 数组赋初值时，初值表中的数据项的数目可以大于或等于数组元素的个数。
3. 指针是用来存放某种变量的地址值的变量。这种变量的地址值也可以存放在某个变量中，存放某个指针的地址值的变量称为指向指针的指针，即二级指针。
4. 引用是某个变量的别名，对引用的操作，实质上就是对被引用的变量的操作。
5. 在说明语句“`int a(5), &b = a, * p = &a;`”中，`b`的值和`* p`的值是相等的。
6. 已知：`int a(5);`表达式`(a = 7) + a`具有二义性。
7. 移位运算符在移位操作中，无论左移还是右移，所移出的空位一律补0。
8. 某个变量的类型高是指该变量被存放在内存中的高地址处。
9. 隐含的类型转换都是使数据的精度不会降低，而显式的类型转换是不安全的转换。
10. 类型定义是用来定义一些C++中所没有的新类型。
11. 在C++中，定义函数时必须给出函数的类型。
12. 在C++中，说明函数时要使用函数原型，即定义函数时的函数头部分。
13. 在C++中，所有的函数在调用前都要说明。
14. 在C++中，传址调用将被引用调用所替代。
15. 使用内联函数是以增大空间开销为代价的。
16. 返回值类型、参数个数和类型都相同的函数也可以重载。
17. 在设置了参数默认值后，调用函数的对应实参就必须省略。
18. 在for循环中，循环变量的作用域是该循环的循环体内。
19. 函数形参的作用域是该函数的函数体内。
20. 调用系统函数时，要先将该系统函数的原型说明所在的头文件包含进去。
21. 带返回值的函数只能有一个返回值。
22. 当在用户自行定义的函数体内使用return语句时，函数立即终止执行。
23. 函数的默认参数没有次序要求，可以随意定义。
24. 在程序中使用全局变量是良好的程序设计风格，它优于局部变量，因为可以避免定义额外的变量。
25. 内部静态变量的生存期贯穿函数定义过程的始终。

三、填空题

1. C++中用于控制流程的三种基本结构是_____、_____、_____。
2. 下面程序的输出结果是_____。

```
#include <iostream>
using namespace std;
void main( void )
{   int   a = 0, b = 2, c = 3;
```



```
switch( a ) {
case 0 : switch( b ) {
            case 1 : cout << " * ";    break;
            case 2 : cout << " % ";    break;
        }
case 1 : switch( c ) {
            case 1 : cout << " $ ";    break;
            case 2 : cout << " * ";    break;
            default : cout << " # ";
        }
}
}
```

3. 下面程序的输出结果是_____。

```
#include <iostream>
using namespace std;
void main( void )
{   int  i = 1, j = 10, k = 1;
    while( i <= j ) {
        k *= 2;
        j--;
    }
    cout << " k = " << k << endl;
}
```

4. 当执行完下面的程序片段后, i 的值____、j 的值____、k 的值_____。

```
int  a = 10, b, c, d, i, j, k;
b = c = d = 5;
i = j = k = 0;
for( ; a > b; ++b )
    i++;
while( a > ++c )
    j++;
do {
    k++;
} while( a > d++ );
```

5. 将下面程序片段(a)补充完整, 让它和程序片段(b)在功能上完全等价。

```
(a) double   s = 0.0;
    _____;
    int       k = 0;
```

```

do {
    s += d;
    _____;
    d = 1.0 / ( k * ( k + 1 ) );
} while( _____ );
(b) double s = 0.0;
for( int k = 1; k <= 10 ;k++ )
    s = 1.0 / ( k * ( k + 1 ) );

```

6. 下面程序的输出结果是_____。

```

#include <iostream>
using namespace std;
void main( void )
{
    int a[5], b[5], * pa, * pb;
    pa = a;
    pb = b;
    for( int i = 1; i < 4; i++, pa++, pb++ ) {
        * pa = i;
        * pb = 2 * i;
        cout << a[i - 1] << " " << b[i - 1] << endl;
    }
    pa = & a[1];
    pb = & b[1];
    for( i = 1; i < 3; i++ ) {
        * pa += i;
        * pb *= i;
        cout << * pa ++ << " " << * pb ++ << endl;
    }
}

```

7. 下面程序的输出结果是_____。

```

#include <iostream>
using namespace std;
void main( void )
{
    int a[3][2] = { 10, 20, 30, 40, 50, 60 };
    for( int i = 0; i < 2; i++ )
        cout << a[2 - i][i] << endl;
}

```

8. 下面程序的输出结果是_____。

```

#include <iostream>

```

```
using namespace std;
int a[ ] = { 5, 4, 3, 2, 1 };
main( void )
{   int i, * p, m= 0;
    for( p = a, i = 1; p + i <= a + 4; i++ ) {
        cout << * ( p + i );
        for( i = 0; i < 4; i ++ ) {
            m += p[i];
            cout << "\t" << m;
        }
    }
    cout << endl;
    return 0;
}
```

9. 下面程序的输出结果是_____。

```
#include <iostream>
using namespace std;
char * str[ ] = { "First", "Second", "Third" };
void func( char * s[ ] )
{   cout << * ++ s << endl; }
int main( void )
{   char ** ps;
    ps = str;
    func( ps );
    return 0;
}
```

10. 阅读下面的程序，当程序执行时，若

 若键入的值为 24 时，程序的输出结果是_____。

 若键入的值为 4 时，程序的输出结果是_____。

```
#include <iostream>
using namespace std;
int factor( int, int & , int & );
int main( void )
{   int number, squared, cubed, error;
    cout << "Enter a number(0 - 20): ";
    cin >> number;
    error = factor( number, squared, cubed );
    if( error )
```

```

        cout << "Error encountered ! \n";
    else {
        cout << "Number : " << number << endl;
        cout << "Squared : " << squared << endl;
        cout << "Cubed : " << cubed << endl;
    }
    return 1;
}

int factor( int n, int & rs, int & rc )
{
    if( n > 20 || n < 0 )
        return 1;
    rs = n * n;
    rc = n * n * n;
    return 0;
}

```

11. 下面定义的函数用来在字符串指针 `ps` 所指的字符串数组中插入字符型 `a` 值, `ps` 所指的数组中的数据已由小到大顺序存放, 而 `int` 型指针 `p` 所指内存空间内存放着该字符串数组的字符个数, 插入 `a` 值后数据的存放顺序仍然是由小到大顺序存放。请填写程序中的空白。

```

void func( char * ps, char a, int * p )
{
    int k, n = 0;
    ps[ * p ] = a;
    while( a > ps[n] )
        n++;
    for( k = * p; k > n; k-- )
        ps[k] = _____;
    ps[n] = a;
    ++ * p;
    ps[ * p ] = '\0';
}

```

12. 若给函数 `fun()` 的形参数组 `s[]` 传递字符串 " 4321cde" (代表空格), 则该函数的返回值是_____。

```

long fun( char s[ ] )
{
    long a;
    int m;
    while( * s == ' ' || * s == '\t' || * s == '\n' )
        s++;
    m = ( * s == '-' ? -1 : 1 );
}

```

```
    if( * s == '+' || * s == '-1' )
        s++;
    for( a = 0; * s >= '0' && * s <= '9'; s++ )
        a = 10 * a + ( * s - '0' );
    return m * a;
}
```

13. 函数 minElem() 计算并返回具有 n 个元素的 int 型数组中最小元素的下标, 该数组由一个指针变量 ps 指向它。请填写程序代码中的空白。

```
int minElem( int * ps, int n )
{
    int i = 0, j;
    for( j = 0; j < n; j++ )
        if( ps[j] < ps[i] )
            _____;
    return i;
}
```

14. 下面 decToBin() 函数的功能是将形参 d 的值转换成二进制数, 所得二进制数的每一位存放在一维数组 b[] 中, 且二进制数的最低位放在下标为 0 的元素内。请填写程序代码中的空白。

```
void decToBin( int d, int b[ ] )
{
    int i = 0, m;
    do {
        m = d % 2;
        b[i++] = m;
        d /= ____;
    } while( d );
}
```

15. 下面程序的输出结果是_____。

```
#include <iostream>
using namespace std;
void fun( int b )
{
    if( b ) {
        cout.put( '0' + b % 10 );
        fun( b / 10 );
    }
}

void main( void )
{
    fun( 11001 );
    cout << endl;
}
```

}

16. 下面程序的输出结果是_____。

```

#include    <iostream>
using namespace std;
int x;
void funA( int  & , int );
void funB( int  , int  & );
void main( void )
{   int  first, second = 5;
    x = 6;
    funA( first, second );
        cout << first << "  " << second << "  " << x << endl;
    funB( first, second );
        cout << first << "  " << second << "  " << x << endl;
}

void funA( int  & a, int  b )
{   int  first;
    first = b + 12;
    a = 2 * b;
    b = first + b;
}

void funB( int  u, int  & v )
{   int  second;
    second = x;
    v = second + 4;
    x = u + v;
}

```

四、分析下列程序的输出结果

程序 1：

```

#include    <iostream>
using namespace std;
int factorial(int a);
void main( void )
{   int s(0);
    for(int i(1); i <= 5; i++)
        s += factorial(i);
}

```

```
        cout << "5! + 4! + 3! + 2! + 1! = " << s << endl;
    }
    int factorial(int a)
    {
        static int b = 1;
        b *= a;
        return b;
    }
}
```

程序 2 :

```
#include <iostream>
using namespace std;
void swap(int & x, int & y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
void main( void )
{
    int a(5), b(9);
    cout << "Before called swap( ) : a = "<< a << ", b = " << b << endl;
    swap(a, b);
    cout << "After called swap( ) : a = "<< a << ", b = " << b << endl;
}
}
```

程序 3 :

```
#include <iostream>
using namespace std;
int & fun(int n, int a[ ])
{
    int & m = a[n];
    return m;
}
void main( void )
{
    int b[ ] = { 5, 4, 3, 2, 1 };
    fun(3, b) = 10;
    cout << "After called fun(3, b) : b[3] = "<< fun(3, b) << endl;
}
}
```

程序 4 :

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

```

void print(int), print(char), print(char *);
void main( void )
{   int u(2002);
    print('u');
    print(u);
    print("abcdef");
}
void print(char c)
{   cout << c << endl; }
void print(char * ps)
{   cout << ps << endl; }
void print(int a)
{   cout << a << endl; }

```

程序 5：

```

#include < iostream >
using namespace std;
void reloadfun(int), reloadfun(double);
void main( void )
{   float a(68.4f);
    reloadfun(a);
    char b('a');
    reloadfun(b);
}
void reloadfun(int x)
{   cout << "reloadfun(int) : " << x << endl; }
void reloadfun(double x)
{   cout << "reloadfun(double) : " << x << endl; }

```

五、按下述要求编程，并上机验证

1. 输入 5 个学生 4 门课程的成绩，然后求出：

- (1) 每个学生的总分；
- (2) 每门课程的平均分；
- (3) 输出总分最高的学生姓名和总分数。

2. 编程求下式的值：

$$n^1 + n^2 + n^3 + n^4 + \dots + n^{10}, \quad \text{其中：} n = 1, 2, 3.$$

编写函数时，设置参数 n 的默认值为 2。

3. 编写一个程序，实现输入一个整数并判断它能否被 3，5，7 整除，输出以下信息之一：

能同时被 3, 5, 7 整除;

能被其中两个数 (要输出是哪两个) 整除;

不能被 3, 5, 7 中的任一个数整除。

4. 一个含有 10 个整数元素的数组 (17, 85, 67, 83, 65, 49, 26, 92, 38, 42), 编写一个程序找出其中的最大数和其下标, 并在主函数体内输出显示最大数和对应的下标。
5. 编写一个程序, 提示用户输入一个字符串, 然后按大写字母格式输出该字符串, 要求使用字符串数组存放字符串。
6. 编写一个指针函数 `char * strFind(char * s, char * t)`, 用来查找字符串 `t` 在字符串 `s` 中最右边出现的位置, 若成功地找到则返回到该位置的地址, 若没有找到则返回 `NULL`, 并编写主函数来测试该函数的功能。程序运行时执行主函数体内语句, 从键盘按行键入两个字符串, 其中第 1 个字符串的长度大于第 2 个字符串并包含它, 并输出查找结果。
7. 编写一个判断质数的函数。
8. 编写一个函数 `reverseDigit(int num)`。该函数读入一个十进制整数, 并将该整数的每个位的数字逆序输出。
9. 用递归法求解: “一只母兔从 4 岁开始每年生一只小母兔, 按此规律第 n 年时有多少只母兔” 的问题。
10. 用递归法编写一个将整数转换成字符串的函数 `char * itoa(int d)`, 并编写一个主函数来测试该函数的功能。
11. 编写一个程序, 定义函数名为 `show()` 的两个重载函数, 第 1 个输出 `int` 型变量的值, 前面用字符串 “`int:`” 引导; 第 2 个输出一个字符, 前面用字符串 “`Achar:`” 引导。并编写主函数进行测试, 分别以 `int`、`float`、`char` 和 `short` 等数据类型的变量 (已被赋值) 和常量作为实参来调用 `show()` 函数。
12. 分析下面的宏定义有什么问题, 并用相应的内联函数取代它们。

```
#define MAX( a, b ) a > b ? a : b
#define FACTOR( a ) ( a ) * FACTOR( ( a ) - 1 )
```

13. 编写一个加密程序, 它通过 `cin` 语句从输入流中读取一段明文, 通过 `cout` 语句将密文写到输出流。采用下面简单的加密算法:

通过命令行参数读取密钥 `key`, 它是一个字符串;

明文中字符 `c` 的密文为 “`c ^ key[i]`”;

循环使用 `key` 中的各个字符, 直到把全部明文处理完毕。若 `key` 为空则不做加密;

解密时, 用同一个 `key` 重新加密密文, 于是就把密文还原成明文。

第 3 章 类 和 对 象

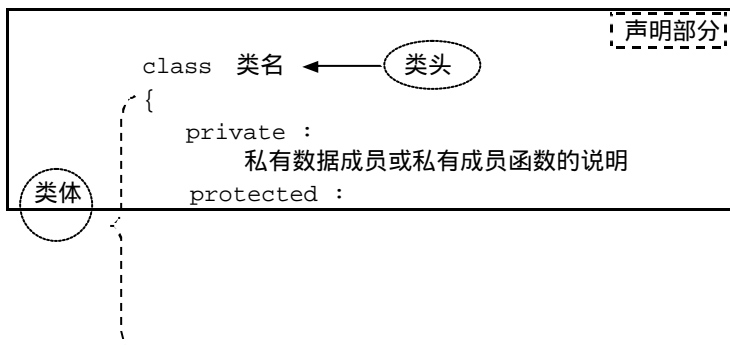
类是面向对象程序设计的核心概念,它实际上是由用户定义的一种新的复杂数据类型。类是通过抽象数据类型 ADT 方法来实现的一种 class 类型,它是对某一类对象的抽象,而对象是某种类的一个实例变量,因此类和对象是密切相关的。一个类是将不同类型的数据和与这些数据相关的操作封装在一起的集合体。因此,它具有更高的抽象性,实现了数据封装和信息隐藏。

在科学技术飞速发展的现代社会,人们需要了解和处理的事情太多、太复杂。为了便于达到某种目的,在某种程度上必须要隐藏一些与此目的无关紧要的具体实现细节。例如在现实生活中人们通常把电视看成一种娱乐和获取信息的工具来使用,不会考虑它的内部结构和工作原理,即将“制造电视机”和“使用电视机”区别开来。“制造电视机”需要由具备电视制造技术的专业技术人员来完成。在面向对象的计算机世界中,这种细节程度的数据隐藏就叫做“抽象(Abstract)”。在 OOP (Object-Oriented Programming) 中也有分工明确的两种编程:一种是标准类库的设计,这是由负责开发面向对象语言系统的软件专门技术人员来完成的;另一种是面向对象应用程序设计,它是由从事计算机应用的专业人员来完成的,这部分人员多数毕业于非计算机专业,不仅人数众多,而且分布在各个应用领域内,他们既具有扎实的本专业基础理论知识和丰富的实践经验,又熟悉计算机应用技术方面的知识,善于用计算机作为强有力的辅助工具去完成本应用领域内的各种任务,在计算机应用领域的大舞台上发挥着其他人才无法替代的重要作用,是应用领域不可缺少的基本力量。因此,对非计算机专业的技术人员普及 OOP 的基础知识是当前刻不容缓的任务。

3.1 类的定义

3.1.1 类的定义格式

类的定义具有如下标准化格式。



```

        保护数据成员或保护成员函数的说明
    public :
        公有数据成员或公有成员函数的说明
};
<各成员函数的实现（定义部分）>

```

类的定义格式分为说明部分和实现部分。说明部分用来描述该类中的成员，包括数据成员的说明和成员函数的说明，前者的说明是定义数据成员的数据类型，后者的说明有时是成员函数原型的声明语句（不带成员函数体，显然，C++与C语言一样，说明语句具有两种含义的功能：一种是起定义功能的说明语句，像定义各种变量的说明语句；另一种是起声明作用的说明语句，像函数原型的声明语句），有时是带成员函数体的整个成员函数的定义，即成员函数的实现部分放在类体内。成员函数是用来对数据成员进行操作的，又称为“方法（method）”。实现部分用来定义各种成员函数，包括成员函数头和成员函数体两个部分，与普通函数一样，成员函数所规定的功能就是靠顺序执行函数体内一条条语句来完成的，换句话说，以描述这些成员函数如何实现对数据成员的操作。总之，说明部分将告诉用户这个类是“干什么”的，而实现部分是告知用户“怎么干”的。

类的说明部分由类头和类体两部分组成，类头由关键字 `class` 和类名组成，类名是由编程者启用的一种标识符，有些软件开发商如 Microsoft 公司，用“C (Class)”开头的字符串作为类名，也有的用大写字母“T (Type)”开头的字符串作为类名，以便与对象名、函数名区别。类头用来向编译系统声明定义了一个新的 `class` 类型，而类体是对类的组织形式进行具体的描述，它由访问限制符（`private`，`protected`，`public`）数据成员和成员函数组成，整个类体用一对大括号包围起来，完整地表达对类的描述，并在后面加上一个分号以表示类定义的结束。例如，用类定义一个关于日期的抽象数据类型 ADT (Abstract Data Type)。日期类的类名为 `Date`，它能设置年、月、日的具体值，并能判断该年是否为闰年，还能显示年、月、日的具体值，ADT (抽象数据类型) 规范化的描述为：

```

ADT    Date is
    Data
        表示年份、月份和日的整数值
    Operations
        setDate
            Input : 表示年、月、日的整数值
            Process : 把表示年、月、日的整数值赋给年份、月份和日
        isLeapYear
            Process : 计算某年是否为闰年
            Output : 是闰年返回值为 1，不是闰年返回值为 0
        printDate

```

Process : 显示年、月、日的具体值

end ADT Date

日期类 Date 的说明部分为：

```
class Date {
public :
    // 3 个公有成员函数
    void setDate(int y = 2000, int m = 1, int d = 1);
    // 设置具体的年、月、日，其默认值为 2000 年 1 月 1 日
    int isLeapYear( void ); // 判断该年是否为闰年
    void printDate( void ); // 显示具体的年、月、日值
private :
    int year, month, day; // 3 个私有数据成员，保存具体的年、月、日值
};
```

日期类 Date 的实现部分，即各成员函数的定义部分为：

```
void Date::setDate( int y, int m, int d )
{   year = y;   month = m;   day = d; }
int Date::isLeapYear( void ) // 判断该年是否为闰年
{   return (   year % 4 == 0 // 能被 4 整除的年份基本上是闰年
            && year % 100 != 0 // 能被 100 整除的年份不是闰年
            || (year % 400 == 0); } // 能被 400 整除的年份又是闰年
void Date::printDate( void ) // 显示年、月、日的具体值
{   cout << year << "," << month << "," << day << endl; }
```

类由数据和函数组成，称为类的成员，因此，类有两种成员：数据成员和成员函数。

本例中 Date 类共有 6 个成员，其中，3 个为私有数据成员，另外 3 个为公有成员函数。这里的成员函数所采用的函数名命名方法是当前较流行的一种，即取用多个英文单词组成一个函数名，每个单词的第 1 个字母用大写（第 1 个单词例外），其余的用小写，各单词之间不加任何分隔符号，类名作为标识符的第 1 个字母用大写，如 Date 类名，其余的像数据成员名、成员函数名、对象名和变量名等的第 1 个字母用小写，如 year、month 和 day 以及 isLeapYear()、printDate() 和后面将用到的对象名 date1 等。

通常习惯上将类定义的说明部分或者整个定义部分（包含实现部分）放到一个头文件中。例如，可将前面定义的类 Date 放到一个名为 tdate.h 的头文件中。在需要引用 Date 类的源程序的开头处写上：

```
#include "tdate.h" /* 这里假定本源文件与 tdate.h 头文件具有相同的路径名，否则应写出详细的路径名 */
```

则 tdate.h 的全部内容将嵌入到这条语句的位置上。

例3.1 编写日期类 Date 的测试程序(即是测试所定义类的各项功能和特性的源程序，其内应包含主函数，能生成可执行程序)。

```

#include    < iostream.h >
/* 使用 C++老标准的流库，读者可将该程序改成使用 C++新标准的流库，修改时最好把 Date 类定
   义放在同一个源文件内，并去掉“#include    "tdate.h"”预处理语句 */
#include    "tdate.h"
void main( void )
{
    Date date1, date2;           // 定义日期类 Date 的两个对象 date1 和 date2
    date1.setDate(2000, 5, 4);   // 给对象 date1 设置年、月、日的具体值
    date2.setDate(2000, 4, 9);   // 给对象 date2 设置年、月、日的具体值
    int leap = date1.isLeapYear( );
    // 判断对象 date1 的年份是否为闰年，闰年 leap = 1，否则为 0
    cout << "LEAP = " << leap << endl;    // 输出“判断是否为闰年”的结果
    date1.printDate( void );      // 显示对象 date1 年、月、日的具体值
    date2.printDate( void );      // 显示对象 date2 年、月、日的具体值
}

```

该程序的输出结果为：

```

LEAP = 1
2000,5,4
2000,4,9

```

class 允许隐藏内部成员，它依靠类定义中的 3 个访问限制符 public、private、protected 来确定隐藏的程度，它们将类体划分成 3 大部分。图 3.1 形象地描绘了这 3 部分的区别。以“public：”开头的程序部分称为公有部分，以“private：”开头的程序部分称为私有部分，以“protected：”开头的程序部分称为保护部分。

类的定义只是定义了某种类的组织形式，即定义了一个新的 class 类型，相当于 C 语言中的结构体定义。编译系统并不给 class 类型的每个数据成员分配具体的内存空间。

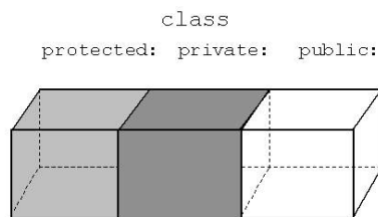


图 3.1 三种访问限制符的区别

3.1.2 访问限制符

访问限制符 (public、private、protected) 将类体分为 3 大部分，每一部分都可以有数据成员和成员函数，也可以只有数据成员或成员函数，但不同的访问限制符规定了该部分成员所具有的访问权限。

public 指定的公有部分是透明的，它的数据成员和成员函数是开放的，既可以由本类的成员函数直接访问，也可由程序的其他部分直接访问。例如允许该类的对象去直接访问它的公有部分，即以“对象名.公有成员名”的格式编写程序。

private 指定的私有部分像一个黑盒子，完全是隐藏的，它只能由本类的成员函

数直接访问,即在成员函数体内直呼其名地写出私有成员名,如在日期类 `Date` 的 `setDate()` 成员函数体内,可以直接写 “`year = y; month = m; day = d;`” 等语句。但是,不允许程序其他部分直接访问,例如不允许该类的对象去直接访问它的私有数据成员,即不允许编写成 “对象名.私有数据成员” 的形式。如在例 3.1 的 `main()` 函数体内,编写以下程序都是非法的。

```
date1.day = 6;
date1.month = 12;
date1.year = 2001;
cout << date1.day;
```

`protected` 指定的保护部分是半透明的,它可由本类成员函数或它的派生类成员函数直接访问,但也不允许程序其他部分直接访问它。保护部分主要用于类的继承,将在后面章节详细讨论。

通常总是将数据成员指定为私有的,以实现数据隐藏,这些数据成员用来描述该类对象的属性,因编程者无法直接访问它们而隐藏起来。一般将成员函数指定为公有的,作为该类对象访问私有数据成员的一个接口界面,即对象访问私有数据成员的一条消息通路提供给外界使用。因此,一个类的对象只能通过公有成员函数访问它的私有数据成员,从而隐藏了处理这些数据的具体实现细节,使得类对数据的描述和类提供给外界来处理数据的界面这两件事情互相独立,这就给出了面向对象的重要特性,使得一个类的用户惟一需要做的事情就是访问类的接口界面。正如图 3.2 所示,日期类 `Date` 封装在一个程序实体内(定义日期类 `Date` 的程序代码),将它的私有数据成员 `year`、`month` 和 `day` 等隐藏起来,不让对象随意访问,就像把电视机的零部件封装在机壳内一样隐藏和保护起来,用户只有操作机壳面板上的按键和开关才能收看电视。同样,对象要访问这些私有数据成员 `year`、`month` 和 `day` 等,也只有通过调用公有的成员函数 `setDate()`、`isLeapYear()` 和 `printDate()` 等才能实现,因为在本类成员函数体内可以直呼其名的访问本类的所有成员,不管它们是公有的、保护的还是私有的,也不管是数据成员还是成员函数。而成员函数就像电视机机壳面板上的按键和开关,是该类对象访问私有数据成员的一个接口界面。因此,在 C++ 中通常把公有成员函数称为一个类的接口界面,简称 “接口”。

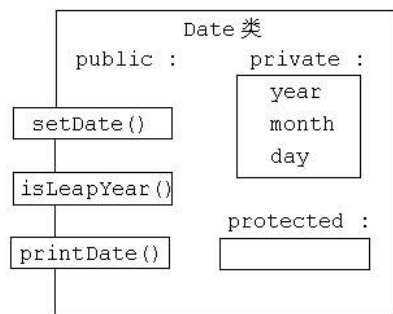


图 3.2 日期类 `Date` 的描述

`Counter` 类的定义。计数器是生产现场广为采用的测量和控制不可缺少的硬件器件,随着计算机技术的广泛普及和推广应用,用程序也可以完成它的功能。在 OOP 中,用类可以直接

定义一个计数器类 (Counter 类), 其抽象数据类型 ADT 为 : 计数器允许的取值范围为 0~4294967295 的正整数 , 可进行的操作是计数器加 1、减 1 和读计数器的值 , ADT 规范化的描述如下。

```

ADT      Counter  is
Data
    记录事件发生次数的正整数值
Operations
    Constructor
        Initial Value : 计数器的初值
        Process : 给计数器赋初值
    increment
        Process : 在计数器允许取值范围内对计数器加 1
    decrement
        Process : 当计数器的值不为 0 时减 1
    readValue
        Process : 读计数器的值
        Output : 返回计数器的值
    Destructor
        Process : 撤销 Counter 类的对象 ( 由系统自动完成 )
end ADT   Counter

```

例 3.2 Counter 类的定义部分、实现部分和测试程序。

```

#include    < iostream >                // 使用 C++ 新标准的流库
using namespace std;                    // 将 std 名空间合并到当前名空间
class Counter {
public :
    Counter( void )                      // Counter 类的构造函数 , 在 3.3 节介绍
    {   value = 0;                        // 初始化数据成员值
        cout << "Constructor called !\n";
        // 输出显示调用了本构造函数的信息
    }
    void increment( void )                // 在允许取值范围内计数器加 1
    {   if( value < 4294967295 ) value++;   }
    void decrement( void )                // 当计数器的数据成员值不为 0 时减 1
    {   if( value > 0 ) value--;   }
    unsigned readValue( void )            // 读计数器的值
    {   return value;   }
    ~ Counter( void )                     // Counter 类的析构函数

```

```

{   cout << "Destructor called !\n";   }
    // 只输出调用了本析构函数的信息, 其他什么也不做

private :
    unsigned value;           // Counter 类的私有数据成员, 保存计数值
};

void main( void )             // Counter 类的测试程序
{   Counter c1 ,c2;           // 定义了 Counter 类的两个对象 c1 和 c2

    for( int i = 1; i <= 6; i++ ) {
        c1.increment ( );
        // 在第 1 个 for 语句的循环体内 6 次向对象 c1 发送增量运算的消息

        cout << "value of c1 = " << c1.readValue( ) << endl;
                                // 每循环一次, 显示对象 c1 的计数值

        c2.increment( );       // 同样 6 次向对象 c2 发送增量运算的消息
    }

    cout << "After loop ,value of c2 = " << c2.readValue( ) << endl;
                                // 循环完成后, 显示对象 c2 的计数值。

    for( i = 1; i <= 5; i++ )
        c2.decrement( );
        // 在第 2 个 for 语句的循环体内 5 次向对象 c2 发送减量运算的消息

    cout << "Final value of c1 = " << c1.readValue( )
        << ",value of c2 = " << c2.readValue( ) << endl;
}    // 显示对象 c1 和 c2 最终的计数值

```

该程序的输出结果为：

```

Constructor called !   ( 创建对象 c1, 调用构造函数时输出的信息 )
Constructor called !   ( 创建对象 c2, 调用构造函数时输出的信息 )

value of c1 = 1
value of c1 = 2
value of c1 = 3
value of c1 = 4
value of c1 = 5
value of c1 = 6
After loop ,value of c2 = 6
Final value of c1 = 6,value of c2 = 1
Destructor called !   ( 撤销对象 c2, 调用析构函数时输出的信息 )
Destructor called !   ( 撤销对象 c1, 调用析构函数时输出的信息 )

```

3.1.3 数据成员

如前所述, 类的定义与结构体的定义一样, 仅仅定义了类的组织形式, 给出了一个新

的 `class` 类型，编译系统并不给类的每个数据成员（Data Members）分配具体的内存空间。数据成员既可以放在公有部分，称为公有数据成员，又可以放在私有或保护部分，称为私有或保护数据成员，其格式都是在类体内作如下定义：

<类型> 数据成员名;

数据成员只有<类型>说明,而无<存储类>说明,所以把类的数据成员说明为 `auto`、`register` 和 `extern` 型都是无意义的。这是因为类的定义只是定义了某种类的组织形式,即定义了一个新的 `class` 类型,编译系统并不给 `class` 类型的每个数据成员分配具体的内存空间,它们还是个虚的东西,因此<存储类>的说明是无意义的。

在类体中不允许对所定义的数据成员进行初始化。例如,下面的定义都是错误的：

```
class Counter {
public :
    ...
private :
    unsigned value = 0;    // 出错,在类体内定义的数据成员不能初始化
};

class Date {
public :
    ...
private :
    int year(2001),month(9),day(16);    // 出错,同上
};
```

数据成员的类型可以是基本数据类型（`char`, `int`, ..., `double` 等）和复杂数据类型（数组, 指针, 引用, ..., 结构变量等）,或已经定义好的类对象。当另一个类的对象作为这个类的成员时,称为“对象成员”,另一个类必须预先定义好,这称为“嵌套结构的类”。例如：

```
class N {                                // N类的定义部分
public :
    ...
};

class M {                                // M类的定义部分
public :
    ...
private :
    N n;                                // n是N类的对象
};
```

如果 N 类的对象 n 作为 M 类的成员,那么 N 类的定义部分必须写在 M 类定义之前,即便使用先声明、使用后再定义的办法都是不允许的,例如:

```
class N;                                // N类的声明
class M {                                // M类的定义部分
public:
    ...
private:
    N n;                                // 出错, n使用了未定义类 N
};
class N {                                // N类的定义部分
public:
    ...
};
```

例 3.3 M 类的一个对象作为 N 类的成员。

```
#include <iostream>                      // 使用 C++新标准的流库
using namespace std;                     // 将 std 名空间合并到当前名空间
class M {
public:
    int value;                           // M类的公有数据成员 value
    M( int v ) { value = v; }             // M类带参数的构造函数
    M( void ) { value = 0; }              // M类无参数的构造函数
    int getValue( void ) { return value; }
    // 读取 value 值作为返回值的成员函数
};
class N {
private:
    int id;                               // N类的私有数据成员 id
public:
    M obj;                                // M类的一个对象 obj 作为 N类公有对象成员
    N( void ) { id = 0; }                 // N类无参数的构造函数
    int getName( void ) { return id; }
    // 读取 id 值作为返回值的成员函数
};
void main( void )
{
    M m = M(6);                           // 定义 M类的对象 m, 调用带参数的构造函数 M(int v)
    N n = N( );                             // 定义 N类的对象 n, 调用无参数的构造函数 N( )
    cout << "value of m = " << m.getValue( ) << endl;
    // 输出 M类对象 m 的 value 值
```

```

    cout << "value of obj of n = " << n.obj.value << endl;
    // 输出N类对象 obj 的 value 值
}

```

该程序的输出结果为：

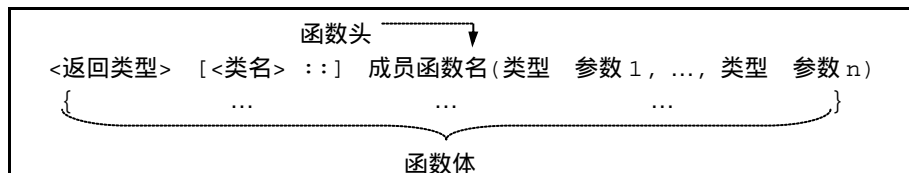
```

value of m = 6
value of obj of n = 0

```

3.1.4 成员函数

类体（即紧跟在类头后一对大括号所包围的程序部分）中不仅定义了该类的对象所具有的数据结构，还定义了对该类所具有的数据结构（其对象都按该类所具有的数据结构模式来创建）进行操作的成员函数（Member Function），其定义格式如下：



所有成员函数都必须在类体内用函数原型加以声明。而成员函数的定义（操作方法的实现部分）即由函数头和函数体组成的程序部分则较灵活，既可在类体外定义，也可在类体内定义。在类体外定义成员函数时，必须用类名加作用域运算符“::”来指明所定义的成员函数属于哪个类，它是加在成员函数名之前而不是加在函数返回类型说明前。然而，在类体内定义的成员函数，其函数头内可不写“<类名> ::”（格式中用方括号包围起来，以表示有时可能没有的部分）。

在类体内定义成员函数时，编译系统自动地将成员函数当做内联函数处理，其函数头前不必写关键字“inline”。如例3.2中的Counter类，其成员函数increment()、decrement()和readValue()都是内联函数。若是类体外定义的成员函数，必须用关键字“inline”指明它为内联函数编译系统才作为内联函数处理。

如前所述，C++是利用调用成员函数来向对象发送消息的，一个类的对象要表现它的行为，就必须调用该类的成员函数，因此在定义了类的对象后，该对象可用如下格式调用公有成员函数：

对象名.公有成员函数名(实参表);

例如：

```

Counter    c1, c2;
c1.increment ( );
c2.decrement ( );

```

在类的成员函数体内,可以直呼其名地访问该类的数据成员,不管它们是公有的、私有的还是保护的。由此可见,类的成员函数体这一块作用域是访问该类数据成员的广阔天地,是对该类数据成员,特别是私有数据成员进行操作的最重要场所。例如,在 Counter 类的 increment ()成员函数体内可以直接编写访问私有数据成员 value 的语句“value++;”。这也就是前面所说的“成员函数直接访问本类的数据成员”。若把该成员函数放在类体的公有部分,则就形成了“对象调用公有成员函数访问私有数据成员”这一重要的消息通路。

成员函数与普通函数一样,可设置参数的默认值(缺省值 Default)。如 Date 类的公有成员函数 setDate()设置了参数的默认值,即:

```
void    setDate( int y = 2000, int m = 1, int d = 1 );
```

若测试程序 main()函数体内写有:

```
date1.setDate( );  
date1.printDate( );
```

则在显示器上显示有:

```
2000,1,1
```

3.2 对象的定义

定义一个类就相当于创建了一个新的 class 类型。要使用类,还必须用已经定义的类去说明它的实例变量(即对象)。在 C++中, class 类型一旦被定义,它的实例变量(对象)就能被创建,并能初始化,且能定义指针变量指向它。在源程序中,该类名就可以像基本数据类型一样,用来定义具体的实例变量(对象)。例如:

```
Date    date1 , date2 , date[31] , * pdate;  
Counter  c1 , c2 , * pc;
```

3.2.1 对象的定义格式

在定义好称之为“类名”的一个 class 类型以后,就可以用如下格式定义它的对象:

<类名> <对象名表>;

<对象名表>中可以有一个或多个对象名,若为多个对象名,则用逗号分隔开。对象名可以是一般对象名(如上例中 Date、Counter 为类名, date1、date2 为 Date 类的两个一般对象名, c1、c2 为 Counter 类的一般对象),也可以是对象数组名,如 date[31]是由 31 个 Date 类的对象有序排列成 date[0], date[1], ..., date[30]等组成的集合,称该集合为“对象数组”,取对象数组名为 date, <对象名表>还可以定义指向该类对象的指针或对该类对象的引用。

3.2.2 对象指针和对象引用的定义格式

指向某类对象的指针简称“对象指针”，其定义格式为：

`<存储类> <类名> 指针名;`

例如：

```
Counter    c1 ,c2 ,* pc;
```

这里所说的<存储类>是指指针变量本身的存储类型，与 C 语言一样，有 `extern` 型、`static` 型、`auto` 型等。而 `auto` 的说明经常缺省，`extern` 在定义时不写，在声明时必须写。例如，若上面的定义语句写在一个块语句内，包括函数体的一对大括号包围的程序区域，则对象 `c1` 和 `c2` 以及指针变量 `pc` 都是自动型的变量。若这条说明语句写在文件名为 `file1.cpp` 中所有函数体以外，那它们的存储内是外部型的，如果编程者希望在另一个文件名为 `file2.cpp` 的源文件中使用的話，则必须在 `file2.cpp` 源文件内用如下说明语句声明：

```
extern      Counter c1 ,c2 ,* pc;
```

在这个声明点以后的程序区域才能使用。总之，上例的这条说明语句如果写在所有的函数以外，则对象 `c1` 和 `c2`、对象指针 `pc` 的存储类都是 `extern` 型的，若写在某函数体内则为 `auto` 型。这与定义指向基本数据类型的指针变量（如 `int * pi;`）相比，在方法、概念及其要求上都是类似的，只不过这里定义的指针变量 `pc` 是指向一种 `class` 类型的指针变量而已。顺便指出，上例中只定义了一个对象指针 `pc`，但 `pc` 并没有定向，即未指向该类某对象，还必须通过初始化操作或地址赋值操作将已存在的对象地址赋给它。即

```
pc = &c1;
```

与基本数据类型类似，也可以为某类的对象定义一个引用，称为“对象引用”，其格式为：

`<存储类> <类名> 对象名, 引用名 = & 对象名;`

例如：

```
Date    date1 , & rDate = date1;
```

```
Counter    c1 , &rc = c1;
```

引用 `rDate` 和 `rc` 分别是对象 `date1` 和 `c1` 的对象引用，`date1` 和 `c1` 称为“被引用对象”。通过引用的初始化操作，可把对象引用与被引用对象的地址相联系，使得对象引用与被引用对象指的是同一个实体，使对象引用成为被引用对象的替换名，其使用方法与基本数据类型是一样的。另外，在 C++ 中，对象指针或对象引用可作为另一个类的成员，或者作为自身类的成员，此时的对象指针或对象引用都是没有<存储类>说明的。即：

`<类名> * 指针名;`

或

`<类名> & 引用名;`

例如：

```
class M;           // 向编译系统声明，M 类在整个源程序内已经定义

class N {
public:
    M & rm;        // M 类的对象引用 rm 作为 N 类的公有数据成员
    ...
};

class M {
    ...
};
```

只要一个类已经声明（可以还没有定义），则指向该类的对象指针或该类对象的引用都可作为另一个类的数据成员。

例 3.4 当一个类的对象指针作为另一个类的成员时，可以先声明后定义。

```
#include <iostream>           // 使用 C++ 新标准的流库
using namespace std;         // 将 std 名空间合并到当前名空间

class M;                     // 向编译系统声明，M 类在整个源程序内已经定义

class N {
private:
    int id;                  // N 类的私有数据成员 id
public:
    M * pm;                  // M 类的对象指针 pm 作为 N 类的公有成员
    N( int i ) { id = i; }    // N 类的带参数的构造函数
    int getName( void ) { return id; } // 读取 id 值作为返回值的成员函数
};

class M {
public:
    int value;                // M 类的公有数据成员 value
    M( int v ) { value = v; } // M 类的有参构造函数
    M( void ) { value = 0; }  // M 类的无参构造函数
    int getValue( void ) { return value; }
    // 读取 value 值作为返回值的成员函数
};

void main( void )
{
    M    m1(6) , m2(8);      // 创建 M 类的两个对象 m1 和 m2
    N    n(4);                // 创建 N 类的一个对象 n

    n.pm = &m2;
}
```

/* 令对象 n 的对象指针成员 pm 指向对象 m2。请注意 n 是 N 类的对象，pm 是 M 类的对象指针

```

    作为 N 类的成员, m2 是 M 类的对象 */
    cout << "value of m1 = " << m1.getValue( ) << endl;
    // 输出对象 m1 的 value 值
    cout<<"value pointed by member pm of n = " <<[n.pm -> value]<< endl;
    // 通过对象 n 的对象指针成员 pm, 输出对象 m2 的 value 值
    cout << "id of n = " << n.getName( ) << endl; // 输出对象 n 的 id 值
}

```

该程序的输出结果为：

```

value of m1 = 6
value pointed by member pm of n = 8
id of n = 4

```

由此可见, 当 M 类的对象指针 pm 作为 N 类的成员时, 用先声明的办法, 可将 M 类的定义部分放在 N 类定义之后。请注意对象 n 访问数据成员 value 的表达式 “n.pm -> value” 的前提条件, 只有将对象 m2 的地址赋给了对象 n 的对象指针成员 pm 之后, 并且 value 是 M 类的公有数据成员时才能顺利地打通这条访问的消息通路, 其一般格式为:

对象名.对象指针成员名 -> 公有数据成员名

或

对象名.对象指针成员名 -> 公有成员函数名(参数表)

在嵌套结构的类中, 一个类的对象不能作为该类的成员, 但一个类的对象指针或者对象引用却可以作为自身类的成员, 从而构成了“递归类”, 它在队列、栈、链表、树和有向图等数据结构中得到了广泛的应用。这将在本章末尾作详细介绍。

3.2.3 访问类对象成员的方法

如前所述, 类有两种成员, 即数据成员和成员函数。每个对象都是类的一个实例, 都具有自己的数据成员值, 而该类所有对象的数据结构及其访问权限却是由它的类统一定义的。因此同一个类所创建的对象, 其数据结构都是相同的, 类中的成员函数是其所有对象共享的, 而不同对象的数据成员值却可以是不相同的。例如, 对已经定义的 Counter 类可用如下语句定义对象和对象指针:

```
Counter c1, c2, * pc;
```

对象 c1、c2 都具有各自的 value 值, 但 value 的数据类型 (unsigned) 和访问权限 (private) 则由 Counter 类统一定义。由于它是私有数据成员, 故程序的其他部分不能直接访问它, 必须通过公有部分的成员函数访问。

```

c1.value;           // 出错, 对象 c1 不能直接访问私有数据成员
c1.readValue( );    // 对象 c1 调用公有成员函数访问私有数据成员

```

1. 数据成员的访问方法

如上所述,对于私有数据成员,只有通过成员函数才能访问它,而公有数据成员可直接访问,其格式为:

对象名.公有数据成员名 或 对象指针名 -> 公有数据成员名
--

其中对象指针名应与对象属于相同的 class 类型,且必须通过初始化或赋值操作将已存在的对象地址赋给它。当然也可以通过成员函数访问它,再考查一个平面几何中的 Point 类,ADT 规范化的描述为:

```
ADT    Point    is
Data
    用整数表示点的 X、Y 坐标值
Operations
    setPoint
        Input : 表示点 X、Y 坐标的两个整数值
        Process : 把两个整数值分别赋给点的 X、Y 坐标
    readX
        Process : 读取点的 X 坐标值
        Output : 返回点的 X 坐标值
    readY
        Process : 读取点的 Y 坐标值
        Output : 返回点的 Y 坐标值
    move
        Input : 两个整数值,用来表示点在 X、Y 方向移动的分量值
        Process : 把两个分量值分别加到点的 X、Y 坐标值上,求得点移动后的
                  X、Y 坐标值
end ADT    Point
```

将 Point 类的定义写成如下程序代码,并把它存放在 tpoint.h 的头文件中:

```
class Point {
public :
    // 4 个公有成员函数
    void setPoint( int a , int b );    // 设定点的 X、Y 坐标值
    int readX( void ) { return x; }    // 读取点的 X 坐标值
    int readY( void ) { return y; }    // 读取点的 Y 坐标值
    void move(int xOffset ,int yOffset); // 计算点移动后的 X、Y 坐标值
    int x , y;                          // 两个公有数据成员 x 和 y
};

void Point::setPoint( int a , int b )
{
    x = a;          y = b;      }

void Point::move( int xOffset ,int yOffset )
{
    x += xOffset;    y += yOffset;    }
```


例 3.5 编写 Point 类的测试程序。

```

#include < iostream >                // 使用 C++新标准的流库
#include "tpoint.h"
using namespace std;                // 将 std 名空间合并到当前名空间

void main( void )
{
    Point p1 , p2;                  // 定义 Point 类的两个对象 p1 和 p2
    Point &ref = p1 , * ptr = &p2;
    // 还定义了一个对象 p1 的引用 ref 和指向对象 p2 的对象指针 ptr

    p1.x = 3;          p1.y = 5;
    // 可用对象名直接访问公有数据成员，给 p1 点设定 X、Y 坐标值

    p2.setPoint(8 , 10);
    // 也可用公有成员函数 setPoint( )给 p1 点设定 X、Y 坐标值

    p1.move(2 , 1);          // 调用公有成员函数 move( )，移动 p1 点
    p2.move(1 , -2);         // 调用公有成员函数 move( )，移动 p2 点

    cout << "x1 = " << ref.x << "," << "y1 = " << ref.y << endl;
    // 使用对象引用 ref 代替 p1 直接访问它的公有数据成员 x 和 y

    cout << "x2 = " << ptr -> x << "," << "y2 = " << ptr -> y << endl;
    // 使用对象指针 ptr 代替 p2 直接访问它的公有数据成员 x 和 y

}

```

该程序的输出结果为：

```

x1 = 5,y1 = 6
x2 = 9,y2 = 8

```

2. 成员函数的访问方法

访问成员函数的一般格式为：

对象名. 公有成员函数名 (参数表);

或

对象指针名 -> 公有成员函数名 (参数表);

其中对象指针名应与对象属于相同的 class 类型，且必须通过定向操作指向已存在的同类对象。

3. 通过对象引用访问成员的方法

对象引用可用来代替被引用的对象名访问成员，其格式为：

对象引用名. 公有数据成员名;

或

对象引用名. 公有成员函数名 (参数表);

由于对象引用名是被引用对象的一个替换名,因此,用它替代被引用对象名来访问公有数据成员和调用公有成员函数的格式与前面所写格式当然是一样的。

3.3 对象的初始化

3.3.1 构造函数和析构函数

构造函数和析构函数 (Constructor and Destructor) 是在类体中说明的两种特殊的成员函数。

1. 构造函数 (Constructor)

C++中的“类”只定义了一组对象的类型。要使用这个类还必须用“类”说明它的对象,每个对象中的数据成员还必须赋初值,这些功能都是由构造函数完成的。

构造函数是一种特殊的成员函数,其定义格式和其他成员函数相同,只是它以类名作为函数名,例如 Counter 类的构造函数为 Counter(),且不能指定返回类型和显式的返回值,连 void 都不能写,即根本就没有返回类型的概念。在 OOP 中,管理一个类对象的机制是每当该类的对象创建时,编译系统为它的对象分配内存空间,并自动调用构造函数初始化所创建的对象,从而确保每个对象自动地初始化。例如:

```
Counter c1;
```

编译系统为对象 c1 的数据成员 value 分配内存空间,并调用构造函数 Counter() 自动地初始化对象 c1 的 value 值设置为 0。

构造函数只完成新对象的初始化工作,不执行对象的主要任务。即在类创建新对象时,为对象的数据成员分配内存空间,给该对象的数据成员赋初值。

几乎每个类都定义了多个构造函数,以适应创建对象时,对象的参数具有不同个数和类型的情况,即构造函数可重载。

例 3.6 构造函数的重载,即多构造函数 (Multiple Constructors)。以复数为例,用 Complex 类来描述复数,在复数运算和复变函数中得到广泛应用,其 ADT 还应包含复数的各种运算和操作。为了简便起见,仅摘出多构造函数部分来讨论。

```
#include    < iostream >                // 使用 C++新标准的流库
using namespace std;                    // 将 std 名空间合并到当前名空间
class Complex {
public:
    Complex( double, double );           // Constructor(1), 一般构造函数
    Complex( void );                     // Constructor(2), 无参数构造函数
    Complex( const Complex & c );        // Constructor(3), 复制构造函数
    Complex( double real );              // Constructor(4), 类型转换构造函数
```

```

        void print( int i );
private :
    double real, imag;
};
// 一般构造函数，用两个实数分别给新创建对象的两个数据成员赋初值
Complex::Complex( double r , double i )
{
    real = r;      imag = i;
    cout << "Constructor(1) called : real = "
          << real << " ,imag = " << imag << endl;
    // 输出本构造函数被调用的提示信息和数据成员的值
}
// 无参数构造函数，创建新对象并将它的数据成员都设定成 0
Complex::Complex( void )
{
    real = 0.0;      imag = 0.0;
    cout << "Constructor(2) called : real = "
          << real << " ,imag = " << imag << endl;
    // 输出本构造函数被调用的提示信息和数据成员的值
}
/* 复制构造函数，把已存在对象的每个数据成员都对应赋值给新创建对象的数据成员，即对新创建
   对象进行“位模式拷贝” */
Complex::Complex( const Complex & c )
{
    real = c.real;      imag = c.imag;    // “同类对象的位模式拷贝”操作
    cout << "Constructor(3) called : real = "
          << real << " ,imag = " << imag << endl;
    // 输出本构造函数被调用的提示信息和数据成员的值
}
// 类型转换构造函数，把一个实数转换成虚数部分为零的复数
Complex::Complex( double r )
{
    real = r;      imag = 0.0;
    cout << "Constructor(4) called : real = "
          << real << " ,imag = " << imag << endl;
    // 输出本构造函数被调用的提示信息和数据成员的值
}
// 输出显示复数类对象的值，形参 i 用来表示对象的序号
void Complex::print( int i )
{
    if(imag < 0)      // 当虚数部分为负时，按“a - bi”格式输出
        cout << "c" << i << " = " << real << imag << "i" << endl;
    else              // 当虚数部分为正时，按“a + bi”格式输出
        cout << "c" << i << " = " << real << "+" << imag << "i" << endl;
}

```

```

}
void main( void )
{
    Complex    c1 ,c2( 6 ,8 ), c3( 5.6 ,7.9 ), c4 = c2;
    // 创建 Complex 类的 4 个对象, 它们调用不同的构造函数

    c1.print(1);           // 输出显示复数类对象 c1
    c2.print(2);           // 输出显示复数类对象 c2
    c3.print(3);           // 输出显示复数类对象 c3
    c4.print(4);           // 输出显示复数类对象 c4

    c1 = Complex(1.2 ,3.4);
    /* 赋值操作的右值表达式直接调用一般构造函数创建一个 Complex 类的对象, 执行赋值操作时
       通过调用默认重载赋值运算符函数把该新对象复制给 c1, 请参阅 6.2.3 节 */

    c3 = c2;               // 把复数类对象 c2 的每个数据成员值赋给 c3

    c2 = 5;
    /* 执行赋值操作将进行自动类型转换, 调用类型转换构造函数 Constructor(4)把常量 5 转
       换成实数部分为 5.0 虚数部分为零的复数 */

    c1.print(1);           // 输出显示复数类对象 c1
    c2.print(2);           // 输出显示复数类对象 c2
    c3.print(3);           // 输出显示复数类对象 c3
    c4.print(4);           // 输出显示复数类对象 c4
}

```

该程序的输出结果为:

```

Constructor(2) called : real = 0 ,imag = 0      ( 创建 c1, 调用无参数构造函数 )
Constructor(1) called : real = 6 ,imag = 8      ( 创建 c2, 调用一般构造函数 )
Constructor(1) called : real = 5.6 ,imag = 7.9  ( 创建 c3, 调用一般构造函数 )
Constructor(3) called : real = 6 ,imag = 8      ( 创建 c4, 调用复制构造函数 )
c1 = 0+0i                                       ( 执行 " c1.print(1); " 语句 )
c2 = 6+8i                                       ( 执行 " c2.print(2); " 语句 )
c3 = 5.6+7.9i                                  ( 执行 " c3.print(3); " 语句 )
c4 = 6+8i                                       ( 执行 " c4.print(4); " 语句 )
Constructor(1) : real = 1.2 ,imag = 3.4        ( 执行 " c1=Complex(1.2,3.4); " )
Constructor(4) called : real = 5 ,imag = 0     ( 执行 " c2 = 5; " )
c1 = 1.2+3.4i
c2 = 5+0i
c3 = 6+8i
c4 = 6+8i

```

一般在程序中不直接使用赋值操作调用构造函数,但可以在创建对象时采用初始化操作显式地调用构造函数(文献[3]p144 的第 11 行之 的讲法不确切),这是因为赋值操作有时会发生几次函数调用。如例 3.6 的 main()体内,写有:

```
c1 = Complex( 1.2 , 3.4 );
```

执行该赋值语句时，先完成右值表达式的操作，即直接调用构造函数创建一个新对象，然后要调用默认重载赋值运算符函数，把新创建的对象所有数据成员值赋值给对象 c1 的对应数据成员，这就进行了两次函数调用的操作。又如执行“c2 = 5;”语句时，要完成调用类型转换构造函数和默认重载赋值运算符函数的操作，显然增加了程序执行的额外开销，影响程序的执行速度，这是说明语句中的初始化操作与执行语句的赋值操作明显的不同之处（虽然，它们的操作符都是“=”，但操作内容是不同的，初学者应学会如何区分）。因此，通常都是在定义新对象时，显式地调用构造函数，即

```
Complex c1 = Complex( ) , c2 = Complex(6 , 8);
Complex c3 = Complex(5.6 , 7.9), c4 = Complex(c2);
还有
```

```
Date today = Date(2002 , 4 , 9);
Counter c1 = Counter( );
```

将 today 定义为 Date 类的对象，并调用构造函数 Date(int y, int m, int day); 初始化对象 today 的数据成员 year, month, day。其一般格式为：

类名 对象名 = 构造函数名(实参表);

由于“构造函数名”就是“类名”，所以可用更简单的形式：

类名 对象名(实参表);

从前面的例程可知，定义一个类的对象通常几乎都是采用这种简化形式，但初学者应知道它是由上面的简化过程得来的，因此我们必须牢牢记住，创建一个新对象必须调用构造函数对其初始化，这是 OOP 中创建对象的一种管理机制。

2. 析构函数(Destructor)

它是构造函数的配对物，与构造函数一样是与类同名的成员函数，并在函数名前加上一个“~”，以便与构造函数相区别。如例 3.2 中的析构函数~Counter()。

一个类只能有一个析构函数，且析构函数没有参数和返回值，它实现与构造函数相反的功能。在删除对象时执行一些清理任务，例如把该对象从注册表中删除掉，并释放对象所占用的内存空间等，这些操作通常由系统自动完成，不需要用户专门编写程序去完成。但是，有些对象如窗口对象可能还需做其他的事情，如当撤销一个屏幕窗口对象时，还需将它们从屏幕上清除掉，这些操作倒是需要用户编写具体程序完成。

对于用 new 运算符创建的动态对象，只有用 delete 运算符撤销时编译系统才调用析构函数，即 new 和 delete 运算符在程序中必须成对出现。

如果编程者没有给某个类定义析构函数，那么编译系统将生成一个什么也不做（对编程者来说不写任何语句）的缺省（默认）析构函数。例如，Complex 类的缺省析构函数

为：

```
Complex::~Complex( ) { 空 } // 函数体为空，即什么都不做的空函数
```

有人会问为什么要把缺省析构函数设计成什么都不做的空函数呢？因为只有这样，各种存储类的对象才能够按照它本来的创建和撤销管理机制运行。例如，自动对象在程序运行期间，每当遇到它的声明时就创建它，当离开它的作用域时就自动撤销它。这将在 3.10 节中详细介绍。

编译系统自动调用构造函数的次序和调用析构函数的次序是相反的。为了验证这一点，给 Counter 类新增加一个私有数据成员 “int who;”，用来记录所生成对象的序号，程序变为：

```
#include <iostream> // 使用 C++ 新标准的流库
using namespace std; // 将 std 名空间合并到当前名空间

class Counter {
public:
    Counter( int id )
    {
        value = 0; // 把私有数据成员 value 的初值设置为零
        who = id; // 用形参 id 初始化私有数据成员 who
        cout << "Object " << who << " initialized !\n";
        // 输出显示 “创建第几号对象时调用本构造函数” 的信息
    }
    void increment ( void )
    {
        if(value < 4294967295) value++;
    }
    void decrement( void )
    {
        if(value > 0) value--;
    }
    unsigned readValue( void ) { return value; }
    ~Counter( void )
    {
        cout << "Object " << who << " destroyed !\n";
    }
private:
    unsigned value; // 记录事件发生次数的私有数据成员
    int who; // 记录所创建对象序号的私有数据成员
};

void main( void )
{
    Counter c1(1) , c2(2); // 定义两个自动型对象 c1 和 c2，创建顺序为先 c1 后 c2
    cout << " OK !\n";
} // 在 main( ) 函数执行完时，撤销顺序是先 c2 后 c1
```

其输出结果为：

Object 1 initialized ! (创建对象 c1 时输出的信息)

Object 2 initialized ! (创建对象 c2 时输出的信息)

```
OK !
Object 2 destroyed !      ( 撤销对象 c2 时输出的信息 )
Object 1 destroyed !      ( 撤销对象 c1 时输出的信息 )
```

3.3.2 构造函数的种类

构造函数可带有各种类型的参数，根据参数所具有的类型可将它分为如下 4 种：

1. 无参数的构造函数

当程序中写有这样的语句，定义新对象时，如：

```
void main( void )
{
    Complex c1;
    ...
}
```

即 c1 创建时不带实参表，则在 Complex 类中必须定义这样一个构造函数：

```
Complex::Complex( void )
{
    real = 0.0; imag = 0.0;
}
```

显然它是不带任何参数的构造函数，称为无参数的构造函数，当它的函数体为空时称为默认构造函数或缺省构造函数(Default Constructor)，由系统自动提供的构造函数才是默认构造函数。

若某个类一个构造函数都未提供，则 C++ 提供一个什么也不做（函数体为空）的默认构造函数来创建对象，而不做任何初始化工作。

例 3.7 默认构造函数只创建新对象，不做任何初始化工作。

```
#include <iostream>          // 使用 C++ 新标准的流库
using namespace std;         // 将 std 名空间合并到当前名空间

class Point {
public:
    // 默认构造函数 Point( void ) { 空 }

    void setPoint(int a , int b);
    int readX( void ) { return x; }
    int readY( void ) { return y; }
    void move( int xOffset ,int yOffset );
}                               // 无构造函数

private:
    int x , y;                  // 两个私有数据成员 x 和 y 记录一个点对象的 x、y 坐标值
};

// 设定点的 x、y 坐标值

void Point::setPoint( int a , int b )
{
    x = a;      y = b;
}

// 计算点移动后的 x、y 坐标值分别在 x、y 方向偏移 xOffset 和 yOffset
```

```

void Point::move( int xOffset , int yOffset )
{
    x += xOffset; y += yOffset;
}

void main( void )
{
    Point p1 ,p2;           // 调用默认构造函数,只创建新对象,但对它们不赋初值

    cout << "x1 = " <<p1.readX( ) << " , " << "y1 = " << p1.readY( ) << endl;
    cout << "x2 = " <<p2.readX( ) << " , " << "y2 = " << p2.readY( ) << endl;
    p1.setPoint(3 ,5);      // 给对象 p1 设定点的 X、Y 坐标值
    p2.setPoint(8 ,10);     // 给对象 p2 设定点的 X、Y 坐标值
    p1.move(2 ,1);          // 移动 p1 点
    p2.move(1 ,-2);         // 移动 p2 点

    cout << "x1 = " << p1.readX( ) << " , " << "y1 = " << p1.readY( ) << endl;
    cout << "x2 = " << p2.readX( ) << " , " << "y2 = " << p2.readY( ) << endl;
}

```

该程序的输出结果为：

(划有虚线的数据均为随机数)

```

x1 = -858993460 , y1 = -858993460   (p1 和 p2 为自动型对象,未初始化时其
x2 = -858993460 , y2 = -858993460   数据成员为随机值)
x1 = 5 , y1 = 6
x2 = 9 , y2 = 8

```

只要某个类定义了一个构造函数(不一定是无参数的构造函数),则 C++不再提供默认构造函数。编程者需要何种类型的构造函数,都必须自行定义。

与变量定义类似,在用默认构造函数创建对象时,若创建的是全局对象或静态对象,则对象的所有数据成员都初始化为零(对于数值型)或空串(对于字符串)。若把例 3.7 测试程序作如下修改:

```

          修改
Point p1 , p2;   →   static Point p1 , p2;

```

则程序的输出结果变为:

```

x1 = 0 , y1 = 0      (main()函数体内第 1 个 cout 语句的输出)
x2 = 0 , y2 = 0      (main()函数体内第 2 个 cout 语句的输出)
x1 = 5 , y1 = 6      (main()函数体内第 3 个 cout 语句的输出)
x2 = 9 , y2 = 8      (main()函数体内第 4 个 cout 语句的输出)

```

现在再考查一个 Stack 类(堆栈类),堆栈是重要的数据结构之一,广泛地应用于各种场合。它被设计成只能在表的一端访问元素的顺序表,允许访问的一端叫做栈顶(top),不允许访问的另一端叫做栈底(bottom),这里所说的“访问”,其含义是插入和删除元素,或读取栈顶元素等操作。存放堆栈元素最简单的方法是采用数组,这种堆栈的大小不能超过数组的元素个数。往堆栈内插入一个元素的操作称为压入(push)操作,即从栈顶增加一个元素。如果令一个名字为 stklst 数组的第 0 号元素 stklst[0]存

放压入堆栈的第 1 个数据，接着依次压入的数据按顺序存放在 `stklist[1]`，`stklist[2]`，...，`stklist[n]` 内，则此时数组的第 0 号元素 `stklist[0]` 就是栈底，而 `stklist[n]` 元素则为栈顶。从栈顶删除一个元素的操作称为弹出（pop）操作，而堆栈内的数据弹出堆栈的顺序却是完全反过来，即 `stklist[n]`，`stklist[n-1]`，...，`stklist[0]`，这就是堆栈所遵循的“先进后出（LIFO, Last - In First Out）”原则，它就像一个自动手枪的子弹夹一样，最先压入的子弹总是最后发射出去。综上所述，Stack 类堆栈的抽象数据类型 ADT 为：

只需修改一条语句就能用于各种类型的堆栈；
 具有压栈（push）操作，将一个新的数据项压入堆栈；
 具有弹出（pop）操作，从堆栈中弹出一个数据项；
 堆栈总是遵循“先进后出”的原则；
 能检测堆栈是否为空；
 能检测堆栈是否已经装满；
 可把堆栈清空。

描述成规范化的 ADT 如下：

```

ADT    Stack    is

Data
    栈顶位置（top）和堆栈的数据项列表（stklist[ ]）

Operations

    Constructor
        Initial Value：栈顶位置
        Process：把栈顶位置设置为 -1，即将堆栈设置为空

    stkEmpty
        Process：检测堆栈是否为空
        Output：若堆栈为空，则返回 True，否则返回 False

    push
        Input：即将压入堆栈的一个数据项
        Process：把数据项压入堆栈，存放在栈顶位置
        Postconditions：栈顶位置增 1

    pop
        Preconditions：堆栈非空
        Process：删除栈顶的数据项
        Output：返回到栈顶的数据项值
        Postconditions：栈顶数据项被删除，栈顶位置减 1

    peek
        Preconditions：堆栈非空
  
```

```

        Process : 查找栈顶的数据项
        Output : 返回到栈顶数据项的值
        Postconditions : 栈顶位置不变

stkFull
    Process : 检测栈顶位置
    Output : 若栈顶位置达到了堆栈元素个数的最大值, 则返回 True, 否则
            返回 False

clearStk
    Output : 删除堆栈的所有数据项, 并把它设置为初始状态
    Postconditions : 堆栈被设置为初始状态

end ADT Stack

```

根据软件工程化规范要求, 我们引入了一个形式数据类型 `DataType`, 它是一种抽象的数据类型, 在编写程序时用它替代各种具体的数据类型, 而要使用该程序就必须将 `DataType` 实例化, 通常是用 `typedef` 语句把它指定成某种具体的数据类型。因此, 采用数组的堆栈可写成如下代码, 并存放在 `arystack.h` 头文件中:

```

#include < iostream >           // 使用 C++ 新标准的流库
#include < cstdlib >             // 标准函数 exit( ) 原型声明在其中
using namespace std;           // 将 std 名空间合并到当前名空间

typedef char DataType;
// 指定形式数据类型 DataType 为字符型, 若将数据类型改成其他类型, 只需修改该语句
const int maxStackSize = 80;    // 数组元素即堆栈数据项的最大个数

class Stack {
private:
    DataType stklist[MaxStackSize]; // 堆栈数组 stklist 用来存放数据项
    int top;                        // 栈顶
public:
    Stack( void );                 // 无参数构造函数
    void push(const DataType & item); // 将一个新的数据项压入堆栈的操作
    DataType pop(void);            // 从堆栈中弹出一个数据项
    DataType peek(void);           // 取栈顶元素的数据项值
    int stkEmpty(void);            // 检测堆栈是否为空
    int stkFull(void);             // 检测堆栈是否已满
    void clearStk(void);           // 设置堆栈为初始状态
};

Stack::Stack( void )
{
    top = -1;
}

void Stack::push( const DataType & item )
{
    if( top == maxStackSize - 1 ) {

```

```

    cerr << "堆栈已满，终止执行程序 !\n";
    exit(1);
    /* 检测到栈顶位置已达到堆栈最末一个元素的位置，堆栈已经装满，立即终止执行程序，
       退回到操作系统 */
}
top++; // 将栈顶位置增 1
stklist[top] = item; // 把数据项 item 复制到栈顶位置
}
DataType Stack::pop( void )
{
    DataType temp;
    // 定义一个 DataType 类型的中间对象 temp，保存原来栈顶位置的数据项值
    if( top == -1 ) {
        cerr << "堆栈已空，终止执行程序 !\n";
        exit(1);
        // 检测到栈顶位置是 -1，说明堆栈是空的，立即终止执行程序，退回到操作系统
    }
    temp = stklist[top]; // 将栈顶位置的数据项保存在对象 temp 中
    top--; // 栈顶位置减 1
    return temp; // 返回原来栈顶位置的数据项值
}
DataType Stack::peek( void )
{
    if( top == -1 ) {
        cerr << "堆栈已空，终止执行程序 !\n";
        exit(1);
        // 检测到栈顶位置是 -1，说明堆栈是空的，立即终止执行程序，退回到操作系统
    }
    return stklist[top]; // 返回栈顶位置的数据项值
}
int Stack::stkEmpty( void )
{
    return top == -1; } // 堆栈为空则返回 True 即 1，否则返回 False 即 0
int Stack::stkFull(void)
{
    return top == maxStackSize - 1; }
// 堆栈已满，则返回 True，即 1，否则返回 False，即 0
void Stack::clearStk( void )
{
    top = -1; }
// 将栈顶位置设置为 -1，即清除掉堆栈的所有数据项，把堆栈置为初始状态

```

例 3.8 利用堆栈遵循“先进后出”的原则，把键盘敲入的，并存放在字符数组 name[] 中的字符串逆序显示出来。

```

#include    "arystack.h"
void main( void )
{
    Stack s;

    char    name[81], reverse[81];
    cout << "请输入您的英文名字 : " ;    // 输出显示提示信息

    cin >> name;                          // 把键盘敲入的字符串保存在字符串数组 name[ ]中
    // 把存放在字符串数组 name[ ]中的输入字符串压入堆栈 s

    for( unsigned i = 0; i < strlen(name); i++ )
        s.push( name[i] );
    // 将堆栈 s 中的字符串按“先进后出”的原则弹出, 存放在字符串数组 reverse[ ]中

    for( i = 0; i < strlen(name); i++ )
        reverse[i] = s.pop( );
    /* 每循环一次通过堆栈对象 s 调用一次成员函数 pop( ), 从栈顶弹出一个字符存放在字符串
       数组 reverse[ ]中, 利用堆栈“先进后出”的操作原则得到一个逆序字符串 */
    reverse[strlen(name)] = '\0';    // 给逆序字符串的尾端添加一个结尾符
    cout << "很抱歉 ! 把您的英文名字逆序显示出来为 : " << reverse << endl;
}

```

该程序的输出结果为：

请输入您的英文名字 : JohnRodman (CR)

很抱歉 ! 把您的英文名字逆序显示出来为 : namdoRnhoJ

用键盘输入的字符串“JohnRodman”由字符串数组 name[]接受, 其数据结构图如图 3.3 所示, 它有 10 个字符, 占用 11 个字节的内存空间。在 main()函数的第 1 个 for 循环中, 每循环一次通过调用 Stack 类的成员函数 push(), 压入一个字符到堆栈内, 例如第 1 次循环时, 循环变量 i = 0, 即把 name[i] = “J”传递给形参 item, 栈顶 top 增 1 由 -1 变为 0, 然后把字符“J”存放在此位置, 并从 push()返回。接着进行第 2 次循环, 把字符“o”存放在栈顶 top 为 1 的位置。直到最后一个字符“n”存放在栈顶 top 为 9 的位置上为止, 循环变量 i 增 1 后为 10, 使得 for 语句第 2 分量的关系式“i < strlen(name)”不成立而退出循环。

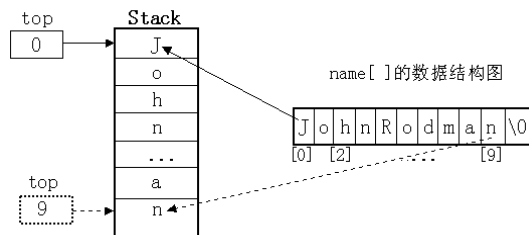


图 3.3 Stack 类的 push 操作

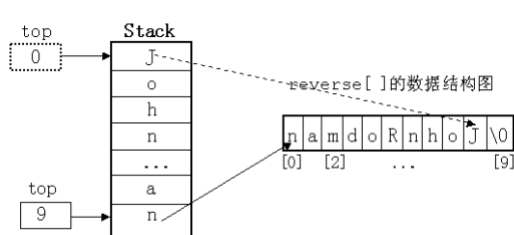


图 3.4 Stack 类的 pop 操作

在 `main()` 函数的第 2 个 `for` 循环中 如图 3.4 所示 每循环一次通过调用 `Stack` 类的成员函数 `pop()`，从堆栈中弹出一个字符放入数组 `reverse` 中，第 1 次循环栈顶 `top` 为 9，则是把栈顶位置的字符“n”作为成员函数 `pop()` 的返回值存入 `reverse[0]`，而栈顶 `top` 变为 8，指向堆栈中的字符“a”，第 2 次循环是把它存入 `reverse[1]` ...，如此类推，直到把栈顶 `top` 为 0 位置上的字符“J”存入 `reverse[9]`，则结束循环。接着把字符串数组的第 10 号元素放上字符串结尾符‘\0’。

例 3.9 用 `Stack` 类实现任意进制输出。

头文件 `arystack.h` 中的 `typedef` 语句应作如下修改：

```

                                改成
typedef char  DataType;  →  typedef long unsigned DataType;
#include "arystack.h"
// 以 R 进制输出正整数 num

void  multiRadixOutput( long unsigned num, int R )
{   Stack  s;                // 定义了一个整型堆栈 s
    // 把 R 进制的各位数由低至高依次压入堆栈 s，直到 num = 0 为止
    do { s.Push( num % R ); // 把余数压入堆栈
        num /= R;          // 用 num / R 替代 num
    } while( num != 0 );    // 直到 num = 0 时，结束循环
    // 按先进后出的原则，把 R 进制的各位数由高至低弹出堆栈 s，并输出到 CRT
    while( ! s.StkEmpty( ) )
        cout << s.Pop( );
}

void  main( void )
{   long  unsigned num;
    unsigned  R;
    for( int i = 0; i < 3; i++ ) {
        cout << "请输入计数制的基数 R (2 <= R <= 9) : " ;
        cin >> R;          // 将键盘敲入的计数制的基数保存在变量 R 中
        cout << "请输入一个正整数 : " ;
        cin >> num;        // 将键盘敲入的十进制正整数保存在变量 num 中
        cout << num << "以基数为" << R << "结果是";
        multiRadixOutput( num, R );
        // 调用 multiRadixOutput( )函数，把十进制正整数转换成 R 进制
        cout << endl;      // 输出一个换行符
    }    // for 语句循环体的结束
}

```

该程序的输出结果为：

请输入计数制的基数 R (2 <= R <= 9) : 8

请输入一个正整数 : 468

468 以基数为 8 的结果是 724

请输入计数制的基数 R (2 <= R <= 9) : 6

请输入一个正整数 : 65349

65349 以基数为 6 的结果是 1222313

请输入计数制的基数 R (2 <= R <= 9) : 2

请输入一个正整数 : 26

26 以基数为 2 的结果是 11010

用 Stack 类可以把一个正整数以任意进制 R (2 ≤ R ≤ 9) 的形式输出, 因为任何一个正整数 num, 若它以 R 为基数, 则有:

$$\text{num} = \text{stklist}[n] * R^n + \text{stklist}[n-1] * R^{n-1} + \dots + \text{stklist}[1] * R + \text{stklist}[0] \quad (3-1)$$

用 R 去除 3.1 式的两端, 则得:

$$\text{num}/R = \text{stklist}[n] * R^{n-1} + \text{stklist}[n-1] * R^{n-2} + \dots + \text{stklist}[1] \quad (3-2)$$

而

$$\text{num} \% R = \text{stklist}[0]$$

式中的 stklist[0] 就是 R 进制最右边位 (最低位) 的有效数字, 再用 R 去除 (3-2) 式的两端, 其余数就是 stklist[1].....如此类推, 直到求得所有位上的有效数字为止, 其算法归纳如下:

第 1 步: R 进制最低位的有效数字由计算式 $\text{num} \% R$ 求得, 把它压入堆栈 s;

第 2 步: 用 num / R 代替 num;

第 3 步: 重复执行上述第 1 步和第 2 步, 直到 $\text{num} = 0$ 为止, 此时所有位的有效数字都由低至高依次压入堆栈 s;

第 4 步: 把 R 进制的各位数由高至低弹出堆栈 s, 并输出到显示器上。

如图 3.5 所示为正整数 468 转换成 8 进制的过程, 堆栈的初始状态为 $\text{top} = -1$, 在 do~while 语句的第 1 次循环中, $\text{num} \% R$ 的结果为 4, 被压入堆栈 s, 接着把 num / R 的结果值 58 再赋给 num, 进入下一次循环, 重复上述过程, 直到 num 为零, 退出 do~while 循环语句。

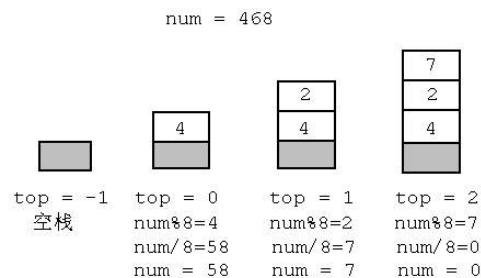


图 3.5 利用 Stack 类以 8 进制输出数据

堆栈总是遵循“先进后出”的原则，接着执行的后继语句是一个 while 语句，其循环体语句调用 Stack 类的成员函数 pop()，把 8 进制的各位数由高至低，即按 7→2→4 的顺序弹出堆栈 s，并输出到显示器上。

2. 复制构造函数（或称拷贝初始化构造函数）

功能。当创建一个新对象时，可用同“类”的一个已存在的对象去初始化新对象。

例如：

```
Complex a(3.2, 6.18);    // 先创建一个 Complex 类的对象 a
Complex b = a;
// 然后再创建一个同类的对象 b，并用已存在的对象 a 去初始化新建对象 b
```

上面两条说明语句是先创建一个 Complex 类的对象 a，然后再创建一个同类的对象 b，并用已存在的对象 a 去初始化新建对象 b，若把第 2 条语句按还原成后应该是：

```
Complex b = Complex( a );
```

该条说明语句中的“=”不是赋值操作而是初始化操作，它将已存在对象的每个数据成员（如 Complex 类中的 real、imag）的值都复制给新对象的对应数据成员，即按每个数据成员在类中说明的顺序，每次复制一个数据成员，称为实现“同类对象的位模式拷贝”，编程者应在类体中设计如下复制构造函数：

```
Complex::Complex( const Complex & c )
{   real = c.real;  imag = c.imag;   }
```

形参 c 是 Complex 类的一个常量类型引用（使用常量类型引用是为了确保已存在对象 a 的数据成员原始值在复制构造函数体内不被破坏），它由调用复制构造函数时对应的实参 a 进行初始化，即在复制构造函数体内，形参 c 就是对象 a 的一个替换名，而直呼其名访问的数据成员 real 和 imag 是新建对象 b 的数据成员，请参阅 3.4.3 节。

复制构造函数的特点。

该函数的第 1 个参数（一般只有这一个参数）是对同类对象的引用，且多数采用对同类常量类型的引用，这样可以确保被复制对象（上例中的对象 a）的所有数据成员值在复制构造函数体内不会被改变。上例中，新对象 b 通过复制构造函数初始化成与已存在对象 a 相同的复数值。

为了便于理解和记忆，设 x 为类名，形式为：

形参
↓

$X::X(\text{const } X \ \& \text{引用名}, \dots);$	或	$X::X(X \ \& \text{引用名}, \dots);$
---	---	-----------------------------------

称为复制构造函数（或称拷贝初始化构造函数），其中也包含只有这一个参数的构造函数。

若某个类没有定义复制构造函数, 而编程者又用同类的已存在对象去初始化新对象, 则编译系统将自动地生成一个如下形式缺省 (默认) 的复制构造函数:

```
X::X(const X & 引用名)
{
    数据成员名 1 = 引用名. 数据成员名 1;
    数据成员名 2 = 引用名. 数据成员名 2;
    ¼
}
```

作为该类的公有成员函数, 来实现“同类对象的位模式拷贝”, 它的拷贝策略是依次拷贝每个数据成员。但是由于这种缺省 (默认) 复制构造函数“X::X(const X & 引用名);”是一种通用格式, 故编程者应该谨慎使用, 在某些场合会带来程序隐患。为此应首先搞清楚编译系统在哪些情况下会自动调用缺省复制构造函数。

例 3.10 何时调用复制构造函数的测试。

```
#include <iostream> // 使用 C++ 新标准的流库
using namespace std; // 将 std 名空间合并到当前名空间

class Point {
public:
    Point( int i, int j ); // 一般构造函数的声明
    Point( Point & p ); // 复制构造函数的声明
    ~Point( void ); // 析构函数的声明
    int readX( void ) { return x; } // 读取点的 x 坐标值
    int readY( void ) { return y; } // 读取点的 y 坐标值
private:
    int x, y; // 两个私有数据成员分别保存点的 x 和 y 坐标值
};

Point::Point( int i, int j ) // 一般构造函数的定义
{
    static int k = 0; // 定义内部静态变量 k 用来记录调用一般构造函数的次数
    x = i; y = j;
    cout << "一般构造函数第" << ++k << "次被调用 !\n";
}

Point::Point( Point & p ) // 复制构造函数的定义
{
    static int i = 0; // 定义内部静态变量 i 用来记录调用复制构造函数的次数
    x = p.x; y = p.y;
    cout << "复制构造函数第" << ++i << "次被调用 !\n";
}

Point::~~Point( void ) // 析构函数的定义
{
    static int j = 0; // 定义内部静态变量 j 用来记录调用析构函数的次数
    cout << "析构函数第" << ++j << "次被调用 !\n";
}
```



```

}
Point func( Point Q )
{
    cout << "\n 已经进入到 func( )的函数体内  !\n";
    int      x, y;           // 定义自动变量 x 和 y 保存新对象 R 的 x 和 y 的坐标值

    x = Q.readX( ) + 10;
    y = Q.readY( ) + 20;
    cout << "定义对象 R , 则";

    Point    R( x, y );
    return  R;
}

void  main( void )
{
    Point    M(20, 35), P(0, 0);
    Point    N(M);
    cout << "下一步将调用 func( )函数, 请注意各种构造函数的调用次数  !\n\n";

    P = func(N);
    cout << "P = " << P.readX( ) << ", " << P.readY( ) << endl;
}

```

该程序的输出结果为：

一般构造函数第 1 次被调用 ! (创建对象 M 时输出的提示信息)

一般构造函数第 2 次被调用 ! (创建对象 P 时输出的提示信息)

复制构造函数第 1 次被调用 ! (创建对象 N 时输出的提示信息)

下一步将调用 func()函数, 请注意各种构造函数的调用次数 !

复制构造函数第 2 次被调用 ! (实参传递给形参时相当于执行了 “ Point Q=N; ” 输出的提示信息)

已经进入到 func()的函数体内 !

定义对象 R , 则一般构造函数第 3 次被调用 ! (创建对象 R 时输出的提示信息)

复制构造函数第 3 次被调用 ! (返回时, 创建匿名对象 x 时输出的提示信息)

析构函数第 1 次被调用 !

析构函数第 2 次被调用 ! } (撤销匿名对象 x, 对象 R 和形参 Q 输出的提示信息)

析构函数第 3 次被调用 ! }

P = 30, 55

析构函数第 4 次被调用 !

析构函数第 5 次被调用 ! } (撤销对象 N、P 和 M 输出的提示信息)

析构函数第 6 次被调用 ! }

说明：

用已存在的对象 (如 M) 显式地初始化新建对象 (如 N)。如例 3.10 , main() 函数内的语句 “ Point N(M); ”, 它完全等价于 “ Point N = M; ”, 这时需要调用复制

构造函数。如果该类未定义复制构造函数，则系统将调用默认复制构造函数。

在 C++ 中对象可作为函数的实参传递给形参，称为“传值调用”。如在例 3.10 中 main() 函数体内的语句 “P = func(N);” 的右值表达式便是属于这种情况，即：

(实参 N 传递给形参 Q)

函数原型： Point func(Point Q);

调用 func() 函数时，对象 N 作为实参，用来初始化被调用函数 func() 的形参 Q，相当于执行了 “Point Q = N;” 初始化语句，这时需要调用复制构造函数。如果该类未定义复制构造函数则系统将调用默认复制构造函数。

当对象作为函数返回值时，如上例中 func() 函数内的语句 “return R;” 便是属于这种情况，系统将用对象 R 来初始化一个临时的匿名对象 x，即相当于执行了 “Point x = R;” 初始化语句。这时也需要调用复制构造函数。如果该类未定义复制构造函数，则系统将调用缺省复制构造函数。

显然，在执行 “P = func(N);” 时，将两次调用复制构造函数，特别是当某类没有定义复制构造函数时，则系统将调用默认复制构造函数，从而还自动创建该类的临时对象。如果编程者不想利用该函数的返回值，例如只执行 “func(N);”，而防止第 2 次调用默认复制构造函数，可将该函数的返回类型定义成该类对象的引用，即：

```
类型名 & 函数名(参数表);
```

也可把函数的形参定义成类对象的引用，如：

```
类型名 & 函数名(类型名 & ,... ) ;
```

用以避免系统隐式地调用默认复制构造函数。对于例 3.10 中的 func() 函数，如果把它有形参和返回值都改成对象引用，同时还必须把对象 R 的存储类改成静态型的，即：

```
Point & func( Point & Q )
{
    ...
    static Point R( x , y );
    return R;
}
```

其输出结果变为：

一般构造函数第 1 次被调用！

一般构造函数第 2 次被调用！

复制构造函数第 1 次被调用！

下一步将调用 func() 函数，请注意各种构造函数的调用次数！

已经进入到了 func() 的函数体内！

定义对象 R，则一般构造函数第 3 次被调用！

P = 30, 55

析构函数第 1 次被调用 !

析构函数第 2 次被调用 !

析构函数第 3 次被调用 !

析构函数第 4 次被调用 !

显然,在调用 func()函数时,没有再调用复制构造函数。如前所述,这种返回引用的函数,只能用全局对象或静态对象作为返回值,而不能用自动对象作为返回值,因此必须将对象 R 定义成静态型。从上述输出结果可知,仅在执行“Point N(M);”语句时,调用了复制构造函数。因为只有当对象作为值传递时,编译系统才会调用复制构造函数(包括默认复制构造函数),而在传递对象的引用或者返回对象的引用时并不调用它,这是因为引用与被引用对象之间是通过地址相联系的,实参传递给形参是传送地址,而不像传值调用那样,需要在形参空间内做复制一个实参的副本,因此采用引用调用方式时,系统不会去调用复制构造函数。所以编程者有时把函数的参数和它的返回值定义成“类型名 &”的形式,以防止编译系统自动调用默认复制构造函数以减少程序执行时间和对内存资源的消耗。但是,必须强调指出,当函数需要返回一个对象时,若把它的返回值指定为该类的对象引用,特别不能是局部对象的引用,若编程者坚持这么做将会产生程序瑕疵,文献[26]中的条款 23 和 31 详细讨论了这个问题,其结论是把函数的参数定义成一个类的对象引用可以减少因调用复制构造函数而消耗的资源,但当函数需要返回一个对象时,最好的办法是采用传值方式,就让它传递回一个对象,即把函数的返回类型指定为该类的对象而不能是对象引用,更不能是该函数体内定义的局部对象的引用。

由于默认复制构造函数只完成一个数据成员的拷贝任务,即做“位模式拷贝”,所以当某类具有指针型数据成员,且构造函数含有用 new 为其对象的数据成员分配堆中的内存空间时,将使新建对象和被复制对象两者占有同一内存空间(资源),当对象撤销时该内存空间将经历两次资源回收操作,导致执行非法操作。在这种场合编程者就必须自行设计复制构造函数。

例 3.11 含有用 new 为其对象的数据成员分配堆中的内存空间的 Person 类。

```
#include <iostream> // 使用 C++ 新标准的流库
#include <cstring> // 标准函数 strlen( ) 的原型声明在其中
using namespace std; // 将 std 名空间合并到当前名空间

class Person {
public:
    Person( char * pN ); // 类型转换构造函数的声明

    Person(Person & p) // 默认复制构造函数, 只做位模式拷贝
    {
        pName = p.pName; // 使两个字符串指针指向同一地址位置
    }
};
```

```

    }

    void show( void )    // 输出显示指针 pName 所指的字符串
    {    cout << pName << " .\n";    }
    ~ Person( );        // 析构函数的声明

private :
    char * pName;        // 私有数据成员是一个字符串指针 pName
};

Person::Person( char * pN )        // 类型转换构造函数的定义
{    cout << "类型转换构造函数被调用 !\n";

    pName = new char[strlen( pN ) + 1];
    // 在堆中开辟一个内存块存放 pN 所指的字符串

    if( pName != NULL )    strcpy( pName ,pN );
    // 如果 pName 不是空指针 ( new 运算符成功执行 ), 则把形参指针 pN 所指的字符串复制给它
}

Person::~~ Person( void )        // 析构函数的定义
{    static int k = 0;    // 定义一个内部静态变量 k 记录调用析构函数的次数

    cout << "析构函数第" << ++k << "次被调用 !\n";

    pName[0] = '\0';
    // 把被撤销对象的数据成员, 即指针 pName 所指的字符串设置为空串

    delete pName;        // 用 delete 运算符撤销 pName 所指的目标字符串
}

void main( void )
{    Person p1( "WillianFord" );        // 创建 Person 类的对象 p1

    Person p2 = p1;
    /* 用已存在的对象 p1 去初始化新创建的对象 p2, 必将调用复制构造函数, 若所属类没有定义
       复制构造函数, 则调用默认的复制构造函数 */

    cout << "这本书作者的英文名字是";

    p1.show( );        // 输出显示对象 p1 的数据成员即指针 pName 所指的字符串
    cout << "另一本书作者的英文名字也是";

    p2.show( );        // 输出显示对象 p2 的数据成员即指针 pName 所指的字符串
    cout << "程序执行完 !\n";

}

```

这是一个不成功的程序, 虽然编译、链接都顺利通过, 但开始执行时, 先创建对象 p1, 调用构造函数 Person(char * pN), 用 new 从堆中为字符串 “ WillianFord ” 分配内存空间, 并将其返回的地址值赋给数据成员 pName。执行 “ Person p2 = p1; ” 时, 因为该类没有定义复制构造函数, 于是就调用默认复制构造函数, 由于它没有给新对象 p2 分配堆空间, 而只是完成了数据成员 pName 的拷贝任务, 故称为 “ 浅拷贝 ”。如

图 3.6 所示,这使得对象 p1 和 p2 的 pName 都指向了堆中的同一内存位置。当 main()

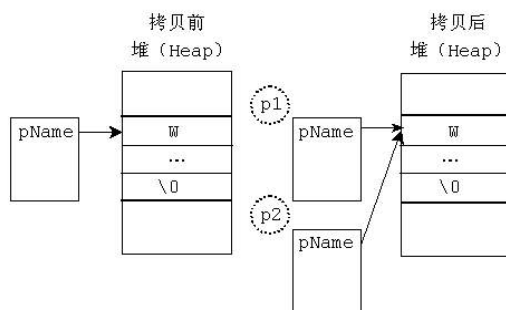


图 3.6 对象 p1 和 p2 的浅拷贝

函数结束时,调用析构函数逐个撤销对象,必然导致用 delete 对同一内存空间进行了两次资源回收操作,正如 2.5 节中所述为非法操作。为此必须自行设计复制构造函数,解决这个问题。所以,将例 3.11 的程序修改成:

```
#include    < iostream >                // 使用 C++新标准的流库
#include    < cstring >                  // 标准函数 strlen( )的原型声明在其中
using namespace std;                    // 将 std 名空间合并到当前名空间

class Person {
public:
    Person( char * pN );
    Person( Person & p );                // 自行设计复制构造函数
    void show( void )
    {   cout << pName << " .\n";   }
    ~ Person( void );
private:
    char * pName;
};

Person::Person( char * pN )
{   cout << "类型转换构造函数被调用 !\n";
    pName = new char[strlen( pN ) + 1];
    if( pName )   strcpy( pName , pN );
    // 如果 pName 不是空指针 ( new 运算符成功执行 ), 则把形参指针 pN 所指的字符串复制给它
}

Person::Person( Person & p )            // 自行设计复制构造函数
{   cout << "复制构造函数被调用 !\n";
    pName = new char[strlen( p.pName ) + 1];
    // 用运算符 new 为新对象的数据成员, 即指针 pName 所指的字符串在堆中分配内存空间
    if( pName )   strcpy( pName , p.pName );
```

```

    /* 如果 pName 不是空指针 ( new 运算符成功执行 ), 则把形参引用 p 对应的实参对象的数据
       成员 pName 所指的字符串复制给它 */
}

Person::~Person( void )
{
    static int k = 0;
    // 定义内部静态变量 k 用来记录调用析构函数的次数
    cout << "析构函数第" << ++k << "次被调用 !\n";

    pName[0] = '\0';
    // 把被撤销对象的数据成员, 即指针 pName 所指的字符串设置为空串
    delete pName;
}

void main( void )
{
    Person    p1( "WillianFord" );
    Person    p2 = p1;
    cout << "这本书作者的英文名字是";

    p1.show( );          // 输出显示对象 p1 的数据成员即指针 pName 所指的字符串
    cout << "另一本书作者的英文名字也是";

    p2.show( );          // 输出显示对象 p2 的数据成员即指针 pName 所指的字符串
    cout << "程序执行完 !\n";
}

```

该程序的输出结果为：

类型转换构造函数被调用！

复制构造函数被调用！

这本书作者的英文名字是 WillianFord。

另一本书作者的英文名字也是 WillianFord。

程序执行完！

析构函数第 1 次被调用！

析构函数第 2 次被调用！

因此，对于含有指针类型数据成员的类，当该类的构造函数含有用 new 为其对象的数据成员分配堆（Heap）中的内存空间时，自行编写复制构造函数是消除程序隐患的最好办法，它除了有“位模式拷贝”功能外，还应具有用 new 运算符为新对象分配内存空间的功能，这称为“深拷贝”。除此之外，都可以借助默认复制构造函数进行“浅拷贝”，使程序简化。

3. 类型转换构造函数

设 X 为类名，T 表示不同于 X 的类型，则形式为：

X(const T & ,...); 或 X(T & ,...);

的构造函数具有类型转换功能，称为类型转换构造函数，其中也包括只有一个形参 T 的构造函数。例如：

```
Complex::Complex(const double & r)
{
    real = r;    imag = 0.0;
}
```

`Complex` 类在定义中提供了将一个 `double` 型的单个参数 r 转换为用户所定义的 `Complex` 类的方法，把 `double` 型单个参数 r 赋给 `Complex` 类的 `real` 成员，而把它的 `imag` 置为 `0.0`。

4. 一般构造函数

设 X 为类名， T_1, T_2, \dots, T_n 为不同于 X 的类型，把形式为：

X(T₁, T₂, ..., T_n);

的构造函数，称为一般构造函数。其中也包括只含有一个或两个参数的构造函数，也允许 T_1, T_2, \dots, T_n 不同于 X 但彼此类型相同。如例 3.6 中 `Complex` 类的构造函数 “`Complex(double real, double imag);`” 就是属于这一类构造函数。

构造函数的分类如表 3.1 所示。

表 3.1 构造函数的分类

构造函数分类	原 型	备 注
一般构造函数	$X(T_1, T_2, \dots);$	X 为类名， $T_1、T_2$ 为不同于 X 的类型。
无参数构造函数	$X(void);$	
复制构造函数	$X(X \ \&, \dots);$ $X(const \ X \ \&, \dots);$	
类型转换 构造函数	$X(T, \dots);$ $X(T \ \&, \dots);$ $X(const \ T, \dots);$ $X(const \ T \ \&, \dots);$	T 为不同 X 的类型

3.3.3 C++的结构体

在 C++ 中结构体是 `class` 类型的一种特殊形式，它也可以有私有部分、公有部分、保护部分，也可有数据成员、成员函数，其中也包括构造函数和析构造函数。例如，在 C 语言中如下说明语句，它说明了一个结构名为 `Point` 的结构类型：

```
struct Point {
    float    x, y;
};
```

} // C 语言中的结构体

C 和 C++ 中的结构体具有如下不同点：

在 C 语言中用结构名 Point 说明结构变量时，必须使用关键字 struct，这适用于扩展名为 “.c” 的源文件，即符合 ISO/ANSI C 老标准的 C 语言源程序。如：

```
struct Point p;
```

而 C++ 中，Point 结构类型一经定义可像基本数据类型名一样使用，不必写关键字 struct，这适用于扩展名为 “.cpp” 的源文件。如：

```
Point p;
```

C++ 扩充了结构体的功能，即在扩展名为 “.cpp” 的源文件中，按 ISO/ANSI C++ 新标准的规定，结构体也可以有成员函数。例如：

```
struct Point {  
[public :]                // C++ 中，结构体的访问限制符 “public :” 可以缺省  
    float x, y;  
    void set( float xi, float yi )  
    { x = xi; y = yi; }  
};
```

C++ 的 struct 和 class 的区别：在 C++ 中结构体是一种特殊的 class 类型，其特殊性表现在对访问限制符的“缺省”使用上。例如上面的结构体 Point 也可以定义成一个类：

```
class Point {  
[private :]              // 类中的访问限制符 “private :” 可以缺省  
    float x, y;  
public :  
    Point( void ) { x = 0.0; y = 0.0; }  
    Point( float xi, float yi ) { x = xi; y = yi; }  
};
```

struct 中访问限制符 “public :” 可以缺省，而 class 中 “private :” 可缺省（用方括号包围的是可以省略不写的部分），换言之，类体内 “private :” 可以省略不写，其下面的所有成员都是私有的，结构体内 “public :” 可以省略不写，其下面的所有成员都是公有的。

struct 对其对象（结构变量）的初始化可采用大括号包围的初始化列表。例如：

```
Point p = {6.6, 8.6}; // 结构体 Point 的初始化
```

也可通过调用自身的成员函数或构造函数对结构变量进行初始化。而通常 class 对其对象的初始化，只能通过调用自身的成员函数或构造函数进行，不能使用初始化列表。

```
Point p; // Point 类对象 p 的初始化是调用其构造函数 Point(void)
```

只有当该类没有私有数据成员，没有构造函数，没有虚函数，且不是派生类时才可以用初

始化列表。

3.4 对象指针和对象引用的应用

3.4.1 对象和对象指针作为函数的参数

首先我们看一个例程。

例 3.12 函数传值调用和传址调用的不同。

```
#include <iostream> // 使用 C++ 新标准的流库
using namespace std; // 将 std 名空间合并到当前名空间

class M {
    int x, y; // 两个私有数据成员 x 和 y
public:
    M( void ) { x = y = 0; } // 无参数构造函数的定义
    M( int i, int j ) // 一般构造函数的定义
    { x = i; y = j; }
    void copy( M * m ); // 成员函数 copy( ) 的声明语句
    void setXY( int i, int j )
    { x = i; y = j; } // 为数据成员 x 和 y 重新设置值
    void print( void ) // 输出显示数据成员 x 和 y 的值
    { cout << "x = " << x << " , y = " << y << endl; }
};

void M::copy(M * m)
{
    x = m -> x; // 相当于执行了 "M * m = &p;" 的初始化操作
    y = m -> y;
}

void fun( M m1, M * m2 ) // 形参 m1 是传值方式, m2 是传地址方式
{
    m1.setXY( 12, 15 );
    /* (实参) 对象 p 传递给 (形参) 对象 m1 (得到一个对象 p 的副本), 但在 fun( ) 函数体内,
       通过 (形参) 对象 m1 调用成员函数 setXY( ) 给 m1 的数据成员 x 和 y 重新设置值分别为
       12 和 15, 但它不会影响 (实参) 对象 p 的数据成员 x 和 y 的值 (仍然是 5 和 7), 显然,
       传值方式的数据传递是 "单方向" 的 */
    m2 -> setXY(22, 25);
    /* (实参) 对象 q 的地址 &q 传递给 (形参) 对象指针 m2, 使得对象指针 m2 指向了 (实参)
       对象 q, 因此, 在 fun( ) 函数体内, 通过 (形参) 对象指针 m2 调用成员函数 setXY( ),
       实际上是对 (实参) 对象 q 的数据成员 x 和 y 重新设置值分别为 22 和 25 */
}

void main( void )
```

```

{
    M p(5, 7), q;           // 创建 M 类的两个对象 p 和 q

    q.copy(&p);              // 通过对象 q 调用成员函数 copy( ) 把 p 复制给对象 q

    fun(p, &q);              // 相当于执行了 “ M m1 = p; ” 初始化操作
                             // 相当于执行了 “ M * m2 = &q; ” 初始化操作

    p.print( );              // 输出显示对象 p 的数据成员 x 和 y 的值
    q.print( );              // 输出显示对象 q 的数据成员 x 和 y 的值
}

```

该程序的输出结果为：

```

x = 5 , y = 7              ( 传值方式调用 fun( ) 函数对象 p 的数据成员 x 和 y 值没有改变 )
x = 22 , y = 25           ( 传地址方式调用 fun( ) 函数对象 q 的数据成员 x 和 y 值发生改变 )

```

如前所述，函数调用时有“传值”和“传地址”两种参数传递方式，前者称为“传值调用”，后者称为“传址调用”。但“传址调用”比“传值调用”应用更普遍，它有如下优点：

采用“传址调用”，可在被调用函数体内改变作为函数实参的对象，即改变对象数据成员的值，实现函数间信息的双向传递。

“传值调用”需要将整个对象进行位模式拷贝，生成副本，而“传址调用”仅将对象的地址作为实参传递给形参对象指针，不仅执行速度快，且占用内存空间小，减少时空开销。

下面分析一下例 3.12 的程序代码并作如下说明：

该程序中有两个指向对象的指针，一个是成员函数 `copy()` 的形参 `M * m`，即以 `M` 类的对象指针 `m` 作为形参，另一个是用作一般函数 `fun()` 的形参 `M * m2`，即以 `M` 类的另一个对象指针 `m2` 作为形参。当形参是指向对象的指针时，调用函数所对应的实参应该是某对象的地址值，采用“&对象名”的格式。

`fun()` 函数有两个形参，一个是对象名 `m1`，进行值传递，另一个是指向对象的指针名 (`M * m2`)，采用地址传递方式。因此，前者的信息传递是单方向，即作为实参的对象 `p`，其数据成员 `x` 和 `y` 的值确实传递给作为形参的对象 `m1`，但是在 `fun()` 函数体内，对形参对象 `m1` 的操作，使其数据成员 `x`、`y` 的值发生变化却不能改变作为实参对象 `p` 的 `x` (仍等于 5)、`y` (仍为 7) 值。而采用地址传递方式的形参 `m2` 却能够实现数据的双向传递，当调用 `fun()` 函数时，在实参传递给形参的过程中使得对象指针 `m2` 指向了作为实参的对象 `q`，那么在该函数体内，对形参对象 `m2` 的操作实际上是对实参对象 `q` 进行操作，

所以对象 q 的数据成员值变成为 $x = 22, y = 25$ 。显然, 成员函数 `copy()` 的形参 $m * m$ 也是采用地址传递方式, 因此执行了 “`q.copy(&p);`” 语句后, 可将对象 p 拷贝给 q 。

从输出结果可知, 与基本数据类型一样, 以某类的对象作为形参传值方式调用函数, 其数据传递是单方向的, 即在函数调用时, 系统给每个形参分配内存空间, 再把实参传递给形参, 在形参的内存空间形成实参的一个 “副本”, 即复制了一个实参的拷贝件, 在被调用函数体内对形参的操作和运算改变的只是形参, 而不会影响实参对象, 这就像平时我们拷贝了一个文件, 将新复制的文件内容修改了, 但原来的源文件内容却不会改变一样。

3.4.2 对象引用作函数参数

若将 3.4 节中所述的规则从变量引申到对象则变成: 凡是用指针作形参进行地址传递的函数, 都可以将作为形参的对象指针改为 “对象引用”, 调用函数的实参可直接写对象名; 被调用函数体内的形参可像对象一样使用, 而不必采用取地址运算和取内容运算。显然, 对象引用作函数参数更简单、方便, 因而更普遍地被采用。例如, 若将例 3.12 的程序按上述规则作如下修改, 则得到采用引用调用的函数, 即:

```

...
void M::Copy( M & m )
{
    x = m.x;
    y = m.y;
}
// 相当于执行了“M & m = p;”初始化操作

void fun(M m1, M & m2);    // 因把 fun( )的定义放在后面, 必须声明

void main( )
{
    M p(5, 7), q;          // 创建 M 类的两个对象 p 和 q
    q.copy( p );           // 通过对象 q 采用传地址方式, 调用成员函数 copy( ) 把 p 复制给对象 q
    fun( p , q );          // 相当于执行了“M m1 = p;”初始化操作
                           // 相当于执行了“M & m2 = q;”初始化操作
    p.print( );            // 输出显示对象 p 的数据成员 x 和 y 的值
    q.print( );            // 输出显示对象 q 的数据成员 x 和 y 的值
}

void fun( M m1, M * m2 )
{
    m1.setxy( 12, 15 );
    m2 -> setxy( 22, 25 );
}

```

3.4.3 this 指针

C++为每个类的成员函数都隐含 (编程者不必写) 定义了一个特殊的指针, 称为 this 指针, 其格式为:

类名 * const this;

this 指针是作为隐含的参数自动传递给成员函数的。哪一个对象调用成员函数, this 指针就指向那个对象, 因此所有成员函数都拥有 this 指针。这是因为每当对象调用成员函数时, 系统就初始化 this 指针, 把对象的地址赋给它, 即给 this 指针定向, 令它指向调用该成员函数的对象。由于 this 指针说明为 * const 即常量指针, 因此在成员函数体内不能被修改而重新定向。使用它时应注意:

this 指针是成员函数的一个隐含参数, 在成员函数体内, 可以隐含使用 this 指针访问本类的数据成员和成员函数, 不管它们是公有的、私有的还是保护的, 都可以直呼其名地访问它们, 从而使得成员函数体内的这片程序区域成为访问本类所有成员 (数据成员和成员函数) 特别是私有成员的广阔天地。例如, Counter 类的成员函数 increment () 也可写成:

```
void Counter::increment ( Counter * const this )
{
    this->value++; // 直呼其名地访问的成员实际上是通过 this 指针
}

```

若程序中写有：

```
Counter c1;           // 定义 Counter 类的一个对象 c1 ( 当做现场的 “ 计数器 ” 使用 )
c1.increment( );
/* 调用成员函数 increment( )使对象 c1 的 value 值增 1，即 “ 计数器 ” c1 当某事件发生时，
   计数一次 */

```

则当执行到该语句时，调用成员函数 `increment()`，`this` 指针就指向了对象 `c1`，函数体内的 “`this->value++;`” 语句使 `c1` 的 `value` 值增 1。由于 `this` 指针是隐含参数，不必显式地写出来，同样可以达到 “用 `this` 指针代替它所指的对象 `c1` 来访问成员” 的目的，所以 “`this->value++;`” 语句与 “`value++;`” 语句完全等价。并且，在成员函数体内，访问本类成员有如下两种形式：

在成员函数体内直呼其名地访问本类的所有成员，包括数据成员和成员函数，特别是私有的；

在成员函数体内，通过本类的某个对象、或对象引用或对象指针调用本类的另一个成员函数，如：

```
#include <iostream>           // 使用 C++ 新标准的流库
using namespace std;         // 将 std 名空间合并到当前名空间

class M {
    int x, y;                 // 两个私有数据成员 x 和 y
public :
    M(void) { x = y = 0; }    // 无参数构造函数
    M(int i, int j)           // 一般构造函数
    { x = i; y = j; }
    void copy(M & m);          // 成员函数 copy( ) 的声明语句
    void setXY(int i, int j)
    { x = i; y = j; }         // 为数据成员 x 和 y 重新设置值
    void print( )              // 输出显示数据成员 x 和 y 的值
    { cout << "x = " << x << " , y = " << y << endl; }
};

void M::copy(M & m)
{
    x = m.x;
    y = m.y;
    /* 赋值运算的左值直呼其名地访问的数据成员 x 和 y 都是 this 指针所指对象 q ( 在主函数体内
       执行语句 “ q.copy(p); ” 时，调用该成员函数的对象 q ) 的成员，而右值是对象引用 m
    */
}

```

的被引用对象即实参对象 `p` 的数据成员。即把实参对象 `p` 的数据成员复制到 `this` 指针所指对象 `q` 对应的数据成员 */

```
print( );
/* 在一个成员函数体内直呼其名调用另一个成员函数 print( ), 显示 this 指针所指对象
   的数据成员值 */

m.print( );
/* 在一个成员函数体内通过形参引用 m 调用另一个成员函数 print( ), 显示被引用 ( 实参 )
   对象的数据成员值 */

}

void main( void )
{   M   p(5, 7), q;           // 创建 M 类的两个对象 p 和 q
    q.copy(p);                // 通过对象 q 调用成员函数 copy( ) 把 p 复制给对象 q
    ...
}
```

从整个程序来看, `this` 指针是一个局部型的指针变量, 每当程序通过某对象调用成员函数时, 系统就要对 `this` 指针重新“定向”, 让它指向调用该成员函数的对象, 如:

```
Counter    c1, c2;           // 定义 Counter 类的两个对象 c1 和 c2
c1.increment( );              // this 指针重新定向指向对象 c1
c2.increment( );              // this 指针重新定向指向对象 c2
```

当执行 “`c1.increment();`” 语句时, 系统给 `this` 指针重新“定向”, 令其指向调用该成员函数的对象 `c1`, 但在 `increment()` 成员函数体内它又是一个常量指针, 它在任何非静态成员函数体内都是可见的, 即都可以隐含使用 `this` 指针访问该对象的成员, 特别是私有数据成员。接着执行 “`c2.increment();`” 时, 系统又令 `this` 指针重新“定向”指向对象 `c2`, 进入到 `increment()` 函数体内后, `this` 指针又变成一个常量指针, 它总是指向对象 `c2`, 代替对象 `c2` (直呼其名地) 去访问其成员, 也不允许编程者写语句重新给它定向。

通常虽然不去显式地使用 `this` 指针, 但在某些场合需要显式地使用 `this` 指针来操作调用成员函数的对象。例如在被调用的成员函数体内用 “`* this`” 来标识该对象, 因为 `this` 指针指向了调用它的对象, 则对 `this` 指针进行取内容运算的结果也就是该对象。

总之, 当一个对象调用它所属类的成员函数时, 则 `this` 指针就定向指向该对象, 在成员函数体内, 就不能再修改它, 直到该成员函数执行完, `this` 指针才被释放; 若另一个对象再调用某成员函数时, 它又重新定向, 指向调用成员函数的对象, 所以它是常量指针和指针变量的统一体。

例 3.13 显式地使用 `this` 指针标识调用成员函数的对象。

```

#include    < iostream >          // 使用 C++新标准的流库
#include    < cstring >           // 标准函数 strcpy( )原型声明在其中
using namespace std;             // 将 std 名空间合并到当前名空间
class Personal {
public :
    Personal( const char * ps );
    /* 类型转换构造函数，把形参字符串指针 ps 所指的字符串常量转换成 Personal 类的对象，
       将该字符串常量存放在该对象的私有数据成员字符串数组 name[20]中 */
    void copy( Personal & rp );    // 复制 Personal 类对象的成员函数
    void print( void )             // 输出显示 Personal 类对象的数据成员 name[ ]
    { cout << "signatory : " << name << endl; }
private :
    char name[20];
    // 私有数据成员字符串数组 name[20]，记录个人姓名的字符串
};

Personal::Personal( const char * ps )
{
    if( ps )
        strcpy( name, ps );
    /* 也可写成 "if( ps != '\0' )"，即当 ps 所指的字符串不为空串时，把 ps 所指的（实
       参）字符串复制到该对象的私有数据成员、字符串数组 name[20]中保存起来 */
    else
        name[0] = '\0';
    // 若 ps 所指的字符串是空串，则把该对象的私有数据成员、字符串数组 name[20]置为空串
}

void Personal::copy( Personal & rp )
{
    if( this == & rp )
        return;
    /* 判断 this 指针所指的对象，即调用 copy() 成员函数的对象就是实参对象自己，即自己复制自己，换言之通过一个对象 p 执行 "p.copy(p);" 语句，则不做“位模式拷贝”操作直接返回 */
    strcpy( this-> name, rp.name );
    /* 若不是自己复制自己，则做“位模式拷贝”操作，将 Personal 类（实参）对象的惟一一个数据成员 name 复制到 this 指针所指的对象，即调用 copy() 成员函数对象的数据成员、字符串数组 name[20]中 */
}
// 相当于执行了 "Personal * const this = &a;" 初始化

void main( void )           // 相当于执行了 "Personal & rp = b;" 初始化操作
{
    Personal a( "" ), b( "Scott McNealy" );

```

化

```

/* 创建 Personal 类的两个对象 a 和 b, 对象 a 的的私有数据成员、字符串数组 name[20]
   中保存空串, 对象 b 的的私有数据成员、字符串数组 name[20] 中保存 "Scott McNealy"
   (史考特.麦克尼立) 字符串 */
cout << "Before copying : \n"; // 输出显示调用成员函数 copy( ) 前的提示信息
a.print( ); // 调用成员函数 print( ) 输出显示对象 a 所保存的字符串
b.print( ); // 调用成员函数 print( ) 输出显示对象 b 所保存的字符串

a.copy(b); // 调用成员函数 copy( ) 将 (实参) 对象 b 复制给对象 a
cout << "After Copying : \n"; // 输出显示调用成员函数 copy( ) 后的提示信息
a.print( ); // 调用成员函数 print( ) 输出显示对象 a 所保存的字符串
b.print( ); // 调用成员函数 print( ) 输出显示对象 b 所保存的字符串
}

```

该程序的输出结果为：

```

Before copying :
signatory :
signatory : Scott McNealy
After Copying :
signatory : Scott McNealy
signatory : Scott McNealy

```

this 指针只有在对象调用成员函数时才被初始化重新定向, 指向调用成员函数的对象, 并作为隐式参数传递给成员函数, 随后进入到成员函数体内就不能再修改了。

this 指针在某个类的成员函数体内是一个常量指针, 但也可以将它定义成指向常量的常量指针, 其方法是在定义成员函数, 或者在类体内声明成员函数时, 在函数头后面加上关键字 const, 例如:

```

class Counter{
public :
    Counter( void ); // Counter 类的无参数构造函数

    ...

    unsigned readValue( void ) const;
    { return value; }

    ~Counter( ); // Counter 类的析构函数

private :
    unsigned value; // Counter 类的私有数据成员
};

```

这实质上是把 this 指针在 readValue() 函数体内说明为如下类型:

```
const Counter * const this ;
```


为了与普通成员函数相区别，把成员函数 `readValue()` 称为“`const` (常量) 成员函数”，详见 3.6 节。那么，在该成员函数体内，不仅 `this` 指针不能重新定向，并且 `this` 指针所指对象的数据成员也不能再修改其值。例如：

```
unsigned readValue( void ) const
{
    return value ++ ;
    // 出错，在 const 成员函数体内，this 指针所指对象的数据成员也是常量不能再修改
}
```

3.4.4 递归类

1. 递归类的定义

与递归结构体类似,在类嵌套结构中,当类的数据成员是具有该类的对象指针和对象引用时,就形成了类的自我嵌套。这种类称为递归类,它在链表、树和有向图等数据结构中得到了广泛的应用。链表是一种最常用的动态数据结构,特别适合于处理数据个数预先无法确定且数据记录频繁变化的场合。链表由一个个的结点构成,即结点是组成链表的基本构件,一个结点由数据域和指向下一个结点的指针(指针域)组成,指针是把链表中单个结点链接起来的纽带,结点类是一个典型的递归类,因为指针域是一个指向自身类型的对象指针,其定义格式为:

```
class 结点类名 {
    <类型>      数据成员名; // 数据域
    <结点类名> * 对象指针名; // 指针域,指向下一个结点的对象指针
public :
    ...
};
```

2. 单向环形链表

如果把一个单向链表最后一个结点的指针域指向表头结点,则得到一个环形链表(Circular List)。由于它可以像双向链表那样,向前、后两个方向查找结点,而环形链表却比双向链表简单,只需一个指针域,且可以选择一条较近的路径查到目标结点。一个单向环形链表结点类的抽象数据类型 ADT 可描述如下:

```
ADT      CLNode    is
Data
    数据域用来保存数据值
    指针域是指向下一个结点的指针
Operations
    Constructor
        Initial Value : 数据值和下一个结点的指针
        Process : 初始化数据域和指针域
    insert
        Input : 当前结点(正在考察的结点称为当前结点)
        Process : 在当前结点后面插入一个新结点,若当前结点是链尾,令当前结点的指针域指向插入结点,而插入结点的指针域指向链头。若当前结点不是链尾,则令当前结点的指针域指向插入结点,而插入结点的指针域指向下一个结点
    get
        Process : 取下一个结点的地址值
```

Output : 返回值是下一个结点的地址值

print

Process : 输出显示结点数据域的字符串

end ADT CLNode

结点类 CLNode 的说明部分、实现部分如下所示，并存放在 CLNode.h 的头文件中：

```
#include < iostream >          // 使用 C++新标准的流库
#include < cstring >
// 标准函数 strcpy( )和 strlen( str )原型声明在其中
using namespace std;          // 将 std 名空间合并到当前名空间
class CLNode{
    char * ps;                  // 数据域，字符串指针
    CLNode * next;              // 指针域，指向下一个结点的指针
public:
    CLNode( char * str );       // 构造函数的声明
    void insert( CLNode & obj );
    CLNode * get(void);
    void print(void);
};
CLNode::CLNode( char * str )    // 构造函数的定义
{
    ps = new char[strlen( str ) + 1];
    // 在堆中为新对象的数据成员指针 ps 所指的字符串开辟内存空间
    strcpy (ps, str);           // 把作为参数的字符串保存在堆中
    next = NULL;                // 把 next 设置成空指针
}
// 插入结点的操作，把形参 obj 作为当前结点，调用本成员函数的对象为待插入结点
void CLNode::insert( CLNode & obj )
{
    /* 当前结点 obj 的 next 指针非空（环形链表已建立），则把它所指结点的地址值赋给待插入
       结点的 next 指针，即插入结点的 next 指针指向了它的下一个结点，或者在当前结点是链
       尾时，它的 next 指针原来指向链头，令它指向插入结点 */
    if ( obj.next != NULL )     // 也可简化写成“if ( obj.next )”
        next = obj.next;
    else
        next = & obj;
    /* 否则（环形链表尚未建立），把当前结点 obj 的地址值赋给插入结点的 next 指针，即令它
       指向当前结点 obj */
    obj.next = this;
    // 把插入结点的地址值赋给当前结点 obj 的 next 指针，令它指向插入结点
}
}
```

```

// 返回到下一个结点的地址值
CLNode * CLNode::get(void)
{   return next;   }
// 输出显示结点数据域的字符串
void CLNode::print(void)
{   cout << "第" << ps << "个结点的字符串为 : \"\" << ps << "\"\n";   }

```

例 3.14 环形链表的建立。

```

#include    "CLNode.h"        // 环形链表结点类 CLNode 定义在其中

void main( void )
{   CLNode  n1( "first" ), n2( "second" ),  n3( "third" ), n4( "fourth" );
    // 定义 4 个环形链表结点类的对象, 并分别用英文字符串标记每个结点的序号

    n2.insert(n1);
    /* 把 n1 作为表头, n2 插入到 n1 之后构成单向环形链表, 即 n1 的 next 指针指
       向 n2, n2 的 next 指针指向 n1 */

    n3.insert(n2);
    // 把 n3 插入到 n2 之后, 即 n2 的 next 指针指向 n3, n3 的 next 指针指向 n1

    n4.insert(n3);
    // 把 n4 插入到 n3 之后, 即 n3 的 next 指针指向 n4, n4 的 next 指针指向 n1

    cout << "单向环形链表已经构成, 它已链接成 : \n";

    CLNode * pnext;
    // 定义一个 CLNode 类的对象指针 pnext, 用来接受成员函数 Get( ) 的返回值

    pnext = n4.get( );
    /* 由于 n4 的 next 指针指向了 n1, 所以 n4.Get( ) 的返回值为结点 n1 的地址值, 即对象
       指针 pnext 指向了结点 n1 */

    pnext -> print( );           // 输出显示结点 n1 数据域的字符串
    pnext = pnext -> get( );    // 对象指针 pnext 向前移动一个结点指向了 n2
    pnext -> print( );           // 输出显示结点 n2 数据域的字符串
    pnext = pnext -> get( );    // 对象指针 pnext 向前移动一个结点指向了 n3
    pnext -> print( );           // 输出显示结点 n3 数据域的字符串
    pnext = pnext -> get( );    // 对象指针 pnext 向前移动一个结点指向了 n4
    pnext -> print( );           // 输出显示结点 n4 数据域的字符串

    cout << "\n 在结点 n2 后插入一个 NewNode 结点, 它链接成 : \n";

    CLNode nn( "NewNode" );     // 定义一个新结点 nn
    nn.insert(n2);              // 把 nn 插入到 n2 之后

    pnext = n4.get( );          // 令对象指针 pnext 指向 n1
    pnext -> print( );           // 输出显示结点 n1 数据域的字符串
    pnext = pnext -> get( );    // 对象指针 pnext 向前移动一个结点指向了 n2
    pnext -> print( );           // 输出显示结点 n2 数据域的字符串

```

```

pnext = pnext -> get( );    // 对象指针 pnext 向前移动一个结点指向了 nn
pnext -> print( );          // 输出显示新结点 nn 数据域的字符串
pnext = pnext -> get( );    // 对象指针 pnext 向前移动一个结点指向了 n3
pnext -> print( );          // 输出显示结点 n3 数据域的字符串
pnext = pnext -> get( );    // 对象指针 pnext 向前移动一个结点指向了 n4
pnext -> print( );          // 输出显示结点 n4 数据域的字符串
}

```

在 `main()` 函数体内,如图 3.7 所示,首先定义了 `CLNode` 类的 4 个对象 `n1~n4`,即创建了 4 个结点,每当创建对象时,都调用构造函数 `CLNode(char * str)` 初始化 4 个对象数据域的字符串指针 `ps`,使得 `n1 ~ n4` 的 `ps` 分别指向“first”、“second”、“third”和“fourth”等字符串,而它们的指针域 `next` 均为空指针。

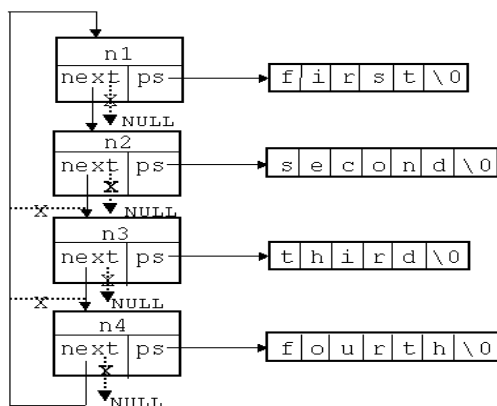


图 3.7 环形链表的建立和测试

接着连续 3 次调用成员函数 `insert()`,把这 4 个结点链接成一个单向环形链表,该成员函数显式地使用了 `this` 指针,完成链表的插入操作。如图 3.7 所示,在创建环形链表时,第 1 次调用 `insert()`,即执行“`n2.insert(n1);`”语句时,系统初始化 `this` 指针,使它指向结点 `n2`,同时用实参 `n1` 初始化形参引用 `obj`,即 `obj` 就是 `n1` 的替换名。由于当前结点 `n1` 的 `next` 指针为 `NULL`,则第一步令结点 `n2` 的 `next` 指针指向结点 `n1`,第二步令结点 `n1` 的 `next` 指针指向结点 `n2`,这样就形成了含两个结点的环形链表。

第 2 次调用 `insert()`,即执行“`n3.insert(n2);`”语句时,`this` 指针指向了结点 `n3`,由于当前结点 `n2` 的 `next` 指针不是空指针,因此,第一步把结点 `n1` 的地址值赋给 `n3` 的 `next` 指针,即令 `n3` 的 `next` 指针指向 `n1`。第二步令 `n2` 的 `next` 指针指向 `n3`,则形成了含 3 个结点的环形链表。接着第 3 次调用 `insert()`,即执行“`n4.insert(n3);`”语句,把 `n4` 插入到 `n3` 之后,最后构成了一个含 4 个结点的单向

环形链表。

该程序的输出结果为：

单向环形链表已经构成，它已链接成：

第一个结点的字符串为：“first”

第二个结点的字符串为：“second”

第三个结点的字符串为：“third”

第四个结点的字符串为：“fourth”

在结点 n2 后插入一个 NewNode 结点，它链接成：

第一个结点的字符串为：“first”

第二个结点的字符串为：“second”

第 NewNode 个结点的字符串为：“NewNode”

第三个结点的字符串为：“third”

第四个结点的字符串为：“fourth”

向单向环形链表的指定结点，如 n2 后面插入一个新结点 nn 的操作过程如图 3.8 所示，同样是调用成员函数 insert() 实现，即执行“nn.insert(n2);”语句，这类类似于上述从第 2 次调用 insert() 开始以后的操作，因为指定结点 n2 的 next 指针不是空指针，所以第一步令新结点 nn 的 next 指针指向 n3，第二步令指定结点 n2 的 next 指针指向新结点 nn。

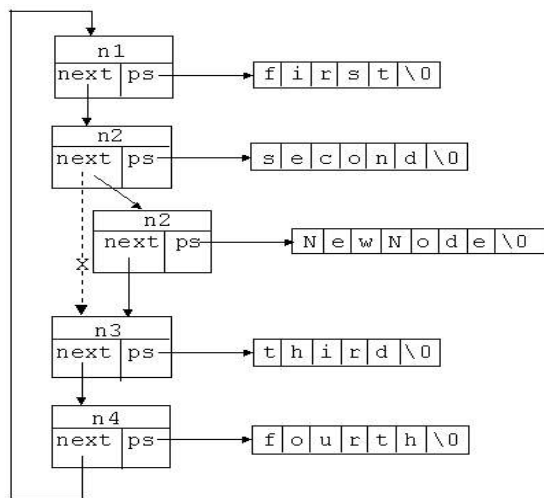


图 3.8 单向环形链表的插入操作

3.5 静态成员

静态成员 (Static Members) 包含静态数据成员和静态成员函数。

3.5.1 静态数据成员

在定义一个类时，若将一个数据成员指明为 `static` 型，则称该数据成员为静态数据成员 (Static Data Members)。不管该类创建了多少个对象，静态数据成员都只有一个静态数据值。因此，静态数据成员是该类所有对象共享的成员，也是连接该类中不同对象的桥梁。

例 3.15 Counter 类的静态数据成员。

```
#include    < iostream >           // 使用 C++新标准的流库
using namespace std;              // 将 std 名空间合并到当前名空间
class Counter {
public :
    Counter(char c)                // 构造函数，令静态数据成员 count 的值增 1
    { count ++;  ch = c; }
    static int home( void ) // 静态成员函数，读静态数据成员 count 的值
    { return count; }
    int readCh( void )          // 普通成员函数，读数据成员 ch 的值
    { return ch; }
    ~ Counter( void )           // 析构函数，令静态数据成员 count 的值减 1
    { count--; ch = 0; }
private :
    static int count;            // 静态数据成员 count
    char ch;                     // 普通数据成员 ch ，记录每个对象的序号
};
int Counter::count = 100;        // 静态数据成员 count 的初始化操作
void main( void )
{
    Counter    c1(1) ,c2(2) ,c3(3),c4(4);
    // 定义 4 个 Counter 类的对象 c1 ~ c4，编号分别为 1、2、3、4
    cout << "Value of Counter is currently " << Counter::home( ) << endl;
    // 输出显示静态数据成员 count 的值
    cout << "Object c1 NO : " << c1.readCh( ) << endl;
    // 输出显示对象 c1 的序号
    cout << "Object c2 NO : " << c2.readCh( ) << endl;
    // 输出显示对象 c2 的序号
    cout << "Object c3 NO : " << c3.readCh( ) << endl;
    // 输出显示对象 c3 的序号
    cout << "Object c4 NO : " << c4.readCh( ) << endl;
```

```
// 输出显示对象 c4 的序号
```

```
}
```

该程序的输出结果为：

Value of Counter is currently 104

Object c1 NO : 1

Object c2 NO : 2

Object c3 NO : 3

Object c4 NO : 4

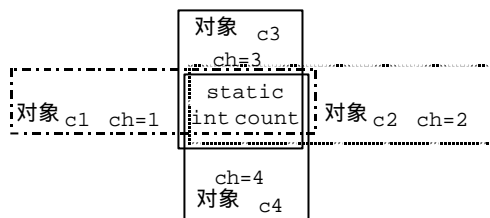


图 3.9 Counter 类的静态数据成员

如图 3.9 所示，每创建一个 Counter 类的对象，不管这些对象是 auto 型、extern 型还是 static 型，都调用一次构造函数 Counter()，静态数据成员 count 的值都增 1。因此，静态数据成员经常用来记录某个类所创建的对象个数。

静态数据成员可说明为公有的、私有的或保护的。说明为公有的可直接访问它，但必须用“类名::”加以指定，其格式为：

类名:: 静态数据成员名

例 3.16 访问公有静态数据成员的操作方法。

```
#include <iostream> // 使用 C++ 新标准的流库
using namespace std; // 将 std 名空间合并到当前名空间
class Counter {
public:
    static long count; // 公有静态数据成员 count
    Counter() // 构造函数，令静态数据成员 count 的值增 1
    { count++; }
    long getCount() // 公有成员函数，读取静态数据成员 count 的值
    { return count; }
    ~Counter() { count--; } // 析构函数，令静态数据成员 count 的值减 1
};
Counter c1, c2, c3; // 创建 Counter 类的 3 个外部对象
long Counter::count = 5; // 静态数据成员 count 的初始化操作
void main( void )
{
    cout << "(1)The object count is " << Counter::count << endl;
    // 用公有静态数据成员的直接访问法，输出显示 count 的值
    Counter c4;
    cout << "(2)The object count is " << Counter::count << endl;
}
```

该程序的输出结果为：

(1) The object count is 8 (count 初始值为 5，创建外部对象 c1、c2 和 c3 后为 8)

(2) The object count is 9 (创建自动对象 c4 后 count 值变成 9)

说明为私有的或保护的静态数据成员与普通私有的或保护的数据成员一样,只有通过调用公有成员函数才能访问它们。

例 3.17 访问私有静态数据成员的操作方法。

```
#include <iostream> // 使用 C++ 新标准的流库
using namespace std; // 将 std 名空间合并到当前名空间
class Counter {
    static long count; // 私有静态数据成员 count
public:
    Counter( void ) { count++; } // 构造函数, 令静态数据成员 count 的值增 1
    long getCount( ) { return count; }
    /* 公有成员函数, 读取静态数据成员 count 的值, 普通成员函数 getCount( ) 体内可以直接访问静态数据成员 count */
    ~Counter( ) { count--; } // 析构函数, 令静态数据成员 count 的值减 1
};
    0
    ↑
long Counter::count = 5; // 静态数据成员 count 的初始化操作
Counter c1, c2, c3; // 创建 Counter 类的 3 个对象
void main( void )
{
    cout << "(1)The object count is " << c3.getCount( ) << endl;
    /* 对象 c3 通过调用公有成员函数 getCount( ) 间接访问静态数据成员 count, 输出显示 count 的值 */
    Counter c4;
    cout << "(2)The object count is " << c3.getCount( ) << endl;
    // 通过外部对象 c3, 调用公有成员函数 getCount( ) 输出显示 count 的值
    cout << "(3)The object count is " << c4.getCount( ) << endl;
    // 通过对象 c4, 调用公有成员函数 getCount( ) 输出显示 count 的值
}
```

该程序的输出结果为：

(1) The object count is 8 (count 初始值为 5, 创建外部对象 c1、c2 和 c3 后为 8)
 (2) The object count is 9 (创建自动对象 c4 后 count 值变成 9, 通过对象 c3 访问)
 (3) The object count is 9 (通过自动对象 c4 访问 count 值仍然是 9)

静态数据成员是 class 类型中一种特殊的数据成员, 它存放在内存空间里的值始终保留, 只要程序一开始运行它就生成, 程序结束时才消失。其空间不会随着对象的创建而分配, 或随着对象的消失而回收, 所以它的空间分配不在类的构造函数里完成, 且空间回收也不在类的析构函数里完成。因此, 不管它是公有的、私有的还是保护的, 都必须要在所有函数体以外用如下格式对它进行初始化, 否则将产生链接错误。

```
<类型> 类名::静态数据成员名 = 初值;
```

显然,静态数据成员的初始化操作语句的前面不加关键字 `static`,这样可以与外部静态对象和外部静态变量区别开来。试比较例 3.16 和例 3.17,静态数据成员 `count` 的初始化操作语句和创建 3 个外部对象的说明语句是颠倒的,但静态数据成员 `count` 的初始值是相同的,这是因为当程序一开始执行时,系统便寻找每个类中的静态数据成员,并在内存中为它们分配内存空间,接着执行它们的初始化操作语句,将其初始值 5 存放在所分配的内存空间内,即 `count` 的初始值是 5,随后,执行全局对象的说明语句即创建 3 个外部对象 `c1`、`c2` 和 `c3`,每创建一个对象就调用一次构造函数使 `count` 值增 1,则 `count` 值最后变成 8。接着,才从 `main()` 主函数开始,按程序编写的顺序逐条执行主函数体内的每条语句直到函数体内的右大括号为止。所以,不管初始化操作语句和说明语句编写的先后顺序,程序执行顺序总是先执行静态数据成员 `count` 的初始化操作语句,随后再执行创建 3 个外部对象的说明语句,最后才从主函数开始按函数体内的编写顺序执行,所以,静态数据成员 `count` 的初始值在进入主函数体时是相同的。

显然,将例 3.17 中的 `count` 设置为 0,并在每个构造函数体内将 `count` 值增 1,而在析构函数体内把 `count` 值减 1,则静态数据成员 `count` 就可用来记录 `Counter` 类所创建的对象个数。但是日常生活中不以零为初值的例子很多,如出租车的里程费就不是以零为初始值。在编写这类程序时,应先将 `count` 初值(即基准值)设为 5,然后再创建 3 个对象,其结果为 8,明确地表达出编程者的意图。若将两语句对调,虽然结果一样,但编程者的意图含糊不清,程序可读性差。

由于普通成员函数都具有 `this` 指针,所以在普通成员函数体内都可以直呼其名地访问静态数据成员,如在 `getCount()` 成员函数体内,可直接写“`return count;`”语句。

静态成员(静态数据成员和静态成员函数)封装在类的作用域内,它不是全局性的。因此静态数据成员在安全性、消除模块间耦合方面比全局变量好得多,即利用静态数据成员可以消除掉所有的全局变量。

定义静态数据成员虽然使用了关键字 `static`,但这里不是指定该数据成员是静态存储类,而是指定它为该类所有对象共享的静态数据成员,它的作用域被封装在类体内,初学者要注意区分静态变量和静态数据成员这两个不同的概念。

3.5.2 静态成员函数

由于类的静态数据成员是所有对象共享的数据,因而若其中任一对象修改了静态数据成员的值,必然影响其他所有对象,导致对象间的强耦合,引起混乱,不好控制。为此,C++还提供了静态成员函数(Static Member Functions),用它来访问静态数据成员

则安全多了。这只需在定义一个类时将关键字“static”加到成员函数头前，即指定该成员函数为静态成员函数。下面用一个例程说明它的特点。

例 3.18 静态成员函数访问静态数据成员的两种方法。

```
#include <iostream>          // 使用 C++ 新标准的流库
using namespace std;         // 将 std 名空间合并到当前名空间

class Simple {
public:
    Simple( int x, int y ) { v1 = x; v2 = y; }
    // 构造函数，对两个静态数据成员 v1 和 v2 赋初值

    static void sum1( Simple * ps ) { ps -> v3 = v1 + v2; }
    /* 静态成员函数，通过形参指针 ps 才能访问普通数据成员 v3，但可直呼其名地访问静态数据
       成员 v1 和 v2 */

    int getV3( void ) { return v3; }
    // 普通成员函数，读普通数据成员 v3 的值

    static void sum2( Simple & r ) { r.v3 = v1 + v2; }
    /* 静态成员函数，通过形参引用 r，才能访问普通数据成员 v3，但可直呼其名地访问静态数
       据成员 v1 和 v2 */

private:
    static long v1, v2;       // 静态数据成员 v1 和 v2
    long v3;                  // 私有数据成员 v3
};

long Simple::v1 = 6;          // 静态数据成员 v1 的初始化操作
long Simple::v2 = 8;          // 静态数据成员 v2 的初始化操作

void main( )
{
    Simple obj1(3, 4);        // 定义 Simple 类的自动型对象 obj1
    obj1.sum1(&obj1);          // 通过对象 obj1 调用静态成员函数 sum1( )

    cout << "(1)The v3 of obj1 is currently " << obj1.getV3( ) << endl;
    Simple obj2(6, 8);
    Simple::sum2(obj2);        // 采用直接调用法，调用静态成员函数 sum2( )

    cout << "(2)The v3 of obj2 is currently " << obj2.getV3( ) << endl;
}
```

该程序的输出结果为：

```
(1) The v3 of obj1 is currently 7
(2) The v3 of obj2 is currently 14
```

静态成员函数在类体内定义，即把函数头和函数体都写在类体内，此时函数头前面一定要加关键字“static”。但它也可以在类体内先用函数原型声明，函数头前面加关键字“static”，然后，在类体外编写定义部分时，就不要在函数头前面加关键字“static”。

静态成员函数没有隐含的 `this` 指针, 这意味着它属于一个类而不属于某一个对象。因此, 在直接调用静态成员函数时, 必须明确指出该静态成员函数在哪个类的对象上操作, 通常采用如下格式:

类名::静态成员函数名(实参表);

例如:

```
Simple::sum2(obj2);
```

这是调用静态成员函数的一种通用方法, 称为“直接调用法”。不管是公有的、私有的还是保护的静态成员函数, 都可以采用“直接调用法”, 这也是调用私有静态成员函数的惟一方法, 如图 3.10 所示, 相当于在私有部分“黑匣子”上开了一个传递消息的孔, 这是打通隐藏在私有部分成员消息通路的方法之一, 若某些数据结构以及操作它们的方法需要隐藏的, 编程者可以把它们放在私有部分隐藏起来, 但为了便于访问可把操作方法指定为静态成员函数, 如例 3.18 中的“`Simple::sum2(obj2);`”语句。

调用公有静态成员函数的另一种方法是通过一个对象来调用静态成员函数, 则必须传递给它一个该类的对象指针来代替 `this` 指针。如例 3.18 中, 用指向 `Simple` 类的指针 `ps` 作为静态成员函数 `sum1(Simple *ps)` 的形参, 当执行“`obj1.sum1(&obj1);`”语句时, 进行实参传递给形参的操作, 使得对象指针 `ps` 指向了对象 `obj1`。在该函数体内, 用对象指针 `ps` 既可以访问非静态数据成员, 又可以访问静态数据成员。当然只有公有部分的静态成员函数才能通过对象来调用它。

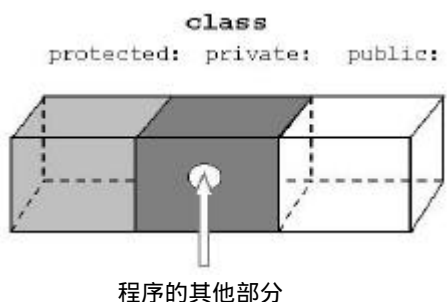


图 3.10 在封装墙上开了一个传递消息的孔

静态成员函数由于没有 `this` 指针, 因而只能访问类中的静态数据成员, 而不能访问类中的非静态数据成员, 这就是静态成员的安全机制。这里所说的访问是指在静态成员函数体内, 通过对象指针或对象引用直呼其名地进行访问, 例如在 `sum1()` 的函数体内, 可以直接写“`ps -> v3 = v1 + v2;`”语句。也就是说, 静态成员函数专门用来处理静态数据成员。如果实际中需要静态成员函数去访问类中的非静态数据成员, 方法之一是把该类的对象指针, 或者该类的对象引用作为静态成员函数的一个参数代替 `this` 指针, 传递给它, 则在该静态成员函数体内, 不管是私有的静态数据成员还是私有的非静态数据成员该参数都能访问。在例 3.18 中写有:

```
static void sum2( Simple & r )
{   r.v3 = v1 + v2;   }
```

如果把该静态成员函数写成：

```
static void sum2( void )
{ v3 = v1 + v2; }
```

则这时 v3、v1 和 v2 必须都是静态数据成员，否则不能访问，即若不给静态成员函数传递一个替代 this 指针的参数，则静态成员函数就专门负责访问静态数据成员。显然，这种方法仅是在静态成员函数的作用域内使得私有的静态数据成员和私有的非静态数据成员都能被访问。这样做虽然牺牲了本作用域内静态成员的安全机制，但不会影响其他程序部分，所以这种隐藏技巧是可取的。

例 3.19 静态成员函数专司访问静态数据成员。

```
#include <iostream> // 使用 C++ 新标准的流库
using namespace std; // 将 std 名空间合并到当前名空间

class Simple {
public :
    Simple(int x, int y) { v1 = x; v2 = y; }
    // 构造函数，对两个静态数据成员 v1 和 v2 赋初值

    int get( ) { return v3; }

    [ static void sum( ) { v3 = v1 + v2; } ]

    // 静态成员函数无 this 指针，在其函数体内只能直呼其名地访问静态数据成员

private :
    [ static long v1, v2, v3; ] // 一组静态数据成员 v1、v2 和 v3
};

long Simple::v1 = 0; // 静态数据成员 v1 的初始化操作
long Simple::v2 = 0; // 静态数据成员 v2 的初始化操作
long Simple::v3 = 0; // 静态数据成员 v3 的初始化操作

void main( void )
{ Simple obj( 36, 28 );
  Simple::sum( );
  cout << "The v3 of obj is currently " << obj.Simple::get( ) << endl;
}
```

当然，若 v3、v1 和 v2 都是静态数据成员，那么无论 Simple 类生成多少个对象，它们只有惟一的一组 v3、v1 和 v2 值。顺便指出，非静态成员函数，即普通成员函数由于具有 this 指针，是可以直接访问静态数据成员的。

静态成员函数与静态数据成员一样都封装在类的作用域内，即在类体的一对大括号包围的程序区域内。

3.6 常量对象和常量成员

如 2.2.1 节所述, 在 C++ 中任何数据类型的对象, 如基本数据类型、派生数据类型、class 类型的对象都可以定义成常量, 即在定义该数据类型的对象时, 启用一个标识符作为对象名, 并在它的名字前面加一个关键字 `const` 就把它定义成“常量对象”, 简称“常对象”。关键字 `const` 不仅可以用来定义“常量对象”, 还可以用来修饰一个类的成员函数和数据成员, 分别称为“常(量)对象”、“常(量)成员函数”和“常(量)数据成员”。众所周知, OOP 的数据封装和信息隐藏确保了数据的安全性, 但实际编程时又需要采用不同形式的数据共享, 例如一个类的所有对象共享该类的静态数据成员, 这又在一定程度上破坏了数据的安全性。对于需要共享的数据又要确保它们的安全性, 即不能让人随意地改变其数值的数据应采取必要的保护措施, 有一个办法就是把它们定义为常量, 那么, 在程序的整个运行期间它们的数值是固定不变的, 就不会受各种因素的影响而被破坏, 从而确保了数据的安全性。

3.6.1 常量对象

如前所述, 在定义某个类的对象时, 若在整个说明语句前或者在对象名前面加一个关键字 `const` 就把它定义成“常量对象”, 其定义格式如下:

<类名> <code>const</code> 对象名列表;

或

<code>const</code> <类名> 对象名列表;

其中对象名列表中的所有对象都定义成常量对象, 且常量对象在定义的同时必须进行初始化, 在程序的整个运行期间它们的数值也不能改变。

例 3.20 常量对象的定义和使用。

```
#include    < iostream >    // 使用 C++ 新标准的流库
using namespace std;    // 将 std 名空间合并到当前名空间

class Point {
    int    x , y;    // 两个私有数据成员 x 和 y 保存点对象的 x 和 y 的坐标值

public :
    Point( void )    { x = 0;    y = 0;    }    // 无参数构造函数的定义
    Point( int a , int b ) { x = a;    y = b;    }    // 一般构造函数的定义
    int readX( void ) const    {    return x;    }
    // 常量成员函数 readX( ) 的定义
    int readY( void ) const    { return y;    }
```

```

// 常量成员函数 readY( )的定义
void move( int xOffset ,int yOffset );
// 普通成员函数 move( )的声明
void show( void ) const
{   cout << "Constant member function called !" << endl;   }
void show( void )
{   cout << "Common member function called !" << endl;   }
};
void Point::move( int xOffset, int yOffset )    // 普通成员函数 move( )的定义
{   x += xOffset; y += yOffset;   }

void    main( void )
{   const    Point    origin, center( 60, 120 );
// 定义 Point 类的两个常量对象 origin( 坐标原点 )和 center( 圆心坐标 )
Point    p1( 30, 50 ), p2( 40, 80 );
// 再定义 Point 类的两个普通对象 p1 和 p2
cout << "(1) Constant object :" << endl;
cout << "origin( " << origin.readX( ) << " , " << origin.readY( ) << " )\n";
// 输出显示常量对象 origin的 X、 Y 坐标值
cout << "center( " << center.readX( ) << " , " << center.readY( ) << " )\n";
// 输出显示常量对象 center的 X、 Y 坐标值
cout << "(2) Before moving :" << endl;
cout << "p1( " << p1.readX( ) << " , " << p1.readY( ) << " )\n";
cout << "p2( " << p2.readX( ) << " , " << p2.readY( ) << " )\n";
// 分别显示普通对象 p1 和 p2 的 X、 Y 坐标值
cout << "(3) After moving :" << endl;
p1.move( 16, 28 );
p2.move( 36, 69 );
// 普通对象 p1 和 p2 可调用普通成员函数 move( ),但常量对象却不能调用它
cout << "p1( " << p1.readX( ) << " , " << p1.readY( ) << " )\n";
cout << "p2( " << p2.readX( ) << " , " << p2.readY( ) << " )\n";
// 普通对象 p1 和 p2 可调用常量成员函数 readX( )和 readY( )
    cout << "(4) Call overload function :" << endl;
center.show( );    // 常量对象 center 调用重载的同名常量成员函数 show( )
p1.show( );        // 普通对象 p1 调用重载的同名普通成员函数 show( )
}

```

该程序的输出结果为：

(1) Constant object :

```
origin( 0 , 0 )
center( 60 , 120 )
(2) Before moving :
p1( 30 , 50 )
p2( 40 , 80 )
(3) After moving :
p1( 46 , 78 )
p2( 76 , 149 )
(4) Call overload function :
Constant member function called !
Common member function called !
```

由此可知,在常量对象的定义语句中,如例 3.20 中的主函数体内第 1 条说明语句,对象名列表可以有多个对象名,它们之间用逗号分隔开。常量对象在定义的同时初始化后再也不能改变它们的数据成员值了。

3.6.2 常量成员函数

1. 常量成员函数的声明和定义

声明常量成员函数的格式如下:

< 返回类型 > 成员函数名(参数表) const;

由此可见,在类体内声明常量成员函数,就在函数头的后面加一个关键字 `const` 即可。常量成员函数与普通成员函数一样,其由函数头和函数体组成的定义部分既可以写在类体内,其格式为:

< 返回类型 > 成员函数名(参数表) const { < 函数体 > }
--

也可写在类体外,其格式为:

< 返回类型 > 类名::成员函数名(参数表) const { < 函数体 > }

在声明和定义常量成员函数时应注意如下两点。

在类体外定义常量成员函数时,切记不要把关键字 `const` 遗漏了,因为它也是函数类型说明的一个组成部分,否则将产生编译错误。如例 3.20 中的两个常量成员函数 `readY()` 和 `readX()` 都是在类体内定义的,若把它们放在类体外定义,则与普通成员函数一样先必须在类体内用函数原型加以声明,即:


```

class Point {
    int x, y;           // 两个私有数据成员 x 和 y 保存点对象的 x 和 y 的坐标值
public:
    ...
    int readX( void ) const;
    // 常量成员函数 readX( ) 的声明
    int readY( void ) const;
    // 常量成员函数 readY( ) 的声明
    ...
};

```

然后，在类体外按上述格式编写实现部分：

```

int Point::readX( void ) const    // 关键字 const 不能遗漏
{ return x; }                    // 常量成员函数 readX( ) 的定义

int Point::readY( void ) const    // 关键字 const 不能遗漏
{ return y; }                    // 常量成员函数 readY( ) 的定义

```

正如 3.4.3 节所述，定义一个常量成员函数实质上是把 this 指针在该成员函数体内定义成指向常量的常量指针，即：

```
const Counter * const this ;
```

因此，在常量成员函数体内不仅 this 指针不能重新定向，它总是指向调用该常量成员函数的对象，而且该对象的数据成员也不能被修改，即不能在常量成员函数体内有更新数据成员值的操作，否则也将产生编译错误。如：

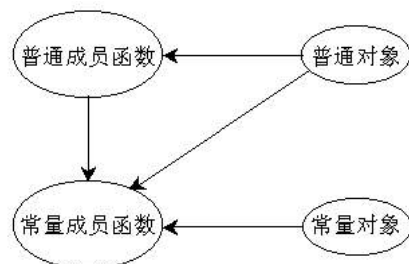
```

int Point::readX( void ) const
{ return x++; }    // 编译错误，不能更新 this 指针所指对象的数据成员值

```

2. 常量成员函数的使用

如图 3.11 所示，常量成员函数与普通成员函数在使用上有如下不同。



注：标有“→”符号的表示此方向可进行调用操作，如常量对象只能调用常量成员函数不能调用普通成员函数。

图 3.11 常量对象、普通对象、常量成员函数和普通成员函数的调用关系

由于常量对象不能被更新,所以常量对象只能调用它的常量成员函数,而不能调用普通的成员函数。如例 3.20 中的两个常量对象 `origin` 和 `center` 只能调用常量成员函数 `readX()` 和 `readY()`,而不能调用普通成员函数 `move()`。

普通对象既可以调用普通成员函数,也可以调用常量成员函数。如例 3.20 中的两个普通对象 `p1` 和 `p2` 既可以调用本类的普通成员函数 `move()`,也调用它的常量成员函数 `readX()` 和 `readY()`。

普通成员函数可以访问本类的常量成员函数,即在该普通成员函数体内,编写调用本类常量成员函数的语句。例如,若在普通成员函数 `move()` 体内增加一条包含调用常量成员函数 `readX()` 和 `readY()` 的语句,即:

```
void Point::move( int xOffset, int yOffset )
{   cout << "( " << readX() << " , " << readY() << " )" << endl;
    // OK! 增加一条包含调用常量成员函数 readX() 和 readY() 的语句
    x += xOffset; y += yOffset;
}
```

反之,常量成员函数却不能访问本类的普通成员函数,即在该常量成员函数体内,不能编写调用本类普通成员函数的语句。例如,若在常量成员函数 `readX()` 函数体内,添加一条调用普通成员函数 `move()` 的语句,即:

```
int readX( void ) const
{   move( 10, 30 );           // 出错,在常量成员函数体内不能调用普通成员函数
    return x;
}
```

当常量成员函数与普通成员函数同名时,就构成了重载成员函数,那么,常量对象则调用常量成员函数,而普通对象调用普通成员函数,即关键字 `const` 可以区分重载函数。如例 3.20 中,`Point` 类有两个重载成员函数 `show()`, 其中一个是普通成员函数,另一个是常量成员函数。因此,执行“`center.show();`”语句是通过常量对象 `center` 调用,所以选中调用常量成员函数,而执行“`p1.show();`”语句时,因 `p1` 是普通对象则选择调用普通成员函数,这从输出结果可以清楚地看出。

3.6.3 常量数据成员

在一个类体内,用关键字 `const` 可将某数据成员指定为常量数据成员,其格式为:

```
class 类名 {
    ...
```

<code>const</code>	<code>< 数据类型 ></code>	<code>数据成员名;</code>
--------------------	-----------------------------	---------------------

或

```
< 数据类型 >    const    数据成员名;
```

```
...
```

```
};
```

常量数据成员只不过是数据值不能再更新的数据成员，而一个类所创建的每个对象又都具有不同的常量数据成员值。当创建一个类的对象时，该类的常量数据成员就必须初始化，且它的值以后就不能再更改。因此，常量数据成员只有在该类构造函数的成员初始化列表中完成，其格式为：

成员初始化列表

```
构造函数名( 总参数表 ) : 常量数据成员名( 参数 )
{
    <函数体>
}
```

下面用一个具体例程来说明常量数据成员的定义和初始化。

例 3.21 定义一个 Airliner（客机）类，它包含常量数据成员 maxPassenger 保存最大载客量。

```
#include    < iostream >        // 使用 C++新标准的流库
#include    < cstring >          // 标准函数 strcpy( )原型声明在其中
using namespace std;           // 将 std 名空间合并到当前名空间
class Airliner {                // Airliner（客机）类的定义
    char name[20];              // 私有数据成员 name[20]存放机型名字符串
    const int maxPassenger;     // 常量数据成员 maxPassenger 保存最大载客量
    int passenger;              // 普通数据成员 passenger 保存实际载客量
public :
    Airliner( char * str, int a, int b ) : maxPassenger( a )
    {
        strcpy( name, str );
        // 把形参字符串指针 str 所指的字符串复制到对象的私有数据成员 name[20]中保存
        passenger = b;          // 形参 b 初始化普通数据成员 passenger
    }
    /* 常量数据成员 maxPassenger 只能在成员初始化列表中完成，普通数据成员既可以在构造
       函数体内进行，也可以在成员初始化列表中完成 */
    char * getName( void )      // 读取机型名
    {
        return name;
    }
    int getMaxPas( void )       // 读取最大载客量
    {
        return maxPassenger;
    }
    int getPassen( void )      // 读取实际的载客量
    {
        return passenger;
    }
};
```

```

void main( void )
{
    Airliner    airln1( "Boeing707", 102, 99 ),
               airln2( "Boeing747", 192, 169 );
    // 定义 Airliner 类的两个对象 airln1 和 airln2, 并调用构造函数进行初始化
    cout << "TYPE NAME \t MAX. PASSENGER\t PASSENGER \n";
    // 输出表头的标题字符串

    cout << airln1.getName( ) << " \t " << airln1.getMaxPas( )
         << " \t\t " << airln1.getPassen( ) << endl;
    cout << airln2.getName( ) << " \t " << airln2.getMaxPas( )
         << " \t\t " << airln2.getPassen( ) << endl;
    // 输出显示 airln1 和 airln2 两个对象的数据成员值
}

```

该程序的输出结果为：

TYPE NAME	MAX. PASSENGER	PASSENGER
Boeing707	102	99
Boeing747	192	169

由此可知：

每个对象的常量数据成员值与普通数据成员一样都是互相独立的，所不同的是常量数据成员在创建对象时初始化所得值不能再更新。例如 `Airliner` 类包含一个常量数据成员 `maxPassenger`，它的两个对象 `airln1` 和 `airln2` 初始化后的 `maxPassenger` 值是不同的，前者为 102 后者为 192，且该值不能再更新，只能读取它的值而不能更改它的值。

常量数据成员的初始化只能在该类构造函数头的成员初始化列表中完成，而不能在构造函数体内进行。并且，构造函数的总参数表应提供足够数目的参数以确保对该类对象的所有数据成员进行初始化，即初始化列表中所需参数应包含在构造函数的总参数表内。例如，`Airliner` 类的构造函数具有两个形参 `str` 和 `a`，其中包括成员初始化列表中所需的参数 `a`。

普通数据成员的初始化既可以在构造函数体内完成，又可以在成员初始化列表进行。例如，`Airliner` 类的构造函数也可以修改成如下形式：

```

Airliner( char * str, int a, int b ) : maxPassenger( a ), passenger( b )
{
    strcpy( name, str );
}

```

即让普通数据成员 `passenger` 在成员初始化列表中完成初始化操作。

创建对象调用这种带有成员初始化列表的构造函数时，其执行顺序是先执行成员初始化列表，接着再执行构造函数体内的语句。

3.7 友元

由于类的私有和保护部分对外是隐藏的，故从外部不能直接访问它们，只能通过调用公有成员函数才能访问。类的这种封装性和数据隐藏对提高软件的可靠性、可重用性和可维护性起着重要作用，但却增加了程序运行时的函数调用开销（即调用函数所需要的执行时间）。因为每次通过成员函数访问类的私有数据成员时，对非内联成员函数都需要做调用前的准备工作。如果访问非常频繁，则调用开销占整个函数调用过程的比率就变成一个不可忽视的问题，从而导致程序运行效率极低，即便是对内联成员函数也会导致程序代码容量的大幅度增加。如前所述，对于一个对象的私有数据成员，只能通过调用它的公有成员函数才能进行访问，这是一堵不透明的墙。当两个不同类的对象共享同一函数时，必然带来较大的开销。出于执行效率的考虑，而并非技术上必须这么做，C++提供了一些辅助手段，允许外面的类或函数去直接访问一个类的私有数据成员，如图 3.10 所示，相当于在这堵不透明的墙上开了一个传递消息的孔；第二种方法就是使用友元（Friend），友元可以是一个函数，称其为友元函数，友元也可以是一个类，称其为友元类。

3.7.1 友元函数

下面用一个例子来说明友元函数的作用。若定义了向量类 `Vector` 和矩阵类 `Matrix`，则各类都隐藏有私有数据成员，并提供了完整的操作函数。为简单起见，假设向量有 3 个元素，下标为 0、1、2。而矩阵包含 3 行 * 3 列 = 9 个元素，下标都是 0、1、2。若要定义一个矩阵乘向量的函数 `multiply()`，首先想到的办法是将它定义成外部函数，由 `Vector` 类和 `Matrix` 类共享它，此时在 `Vector` 类和 `Matrix` 类定义中，必须分别提供通过下标来访问各自元素的成员函数 `elem()`。

```
class Vector {
    double v[3];      // 一维 double 型数组 v[3] 作为该类的私有数据成员
public:
    ...
    double elem( int i ){ return v[i]; }    // 读向量 v 的第 i 号元素
};

class Matrix {
    double m[3][3]; // 二维 double 型数组 m[3][3] 作为该类的私有数据成员
public:
    ...
    double elem( int i, int j )
    { return m[i][j]; }                    // 读矩阵 m 的第 i 行第 j 列的元素
};
```

矩阵 $m[3][3]$ 乘向量 $v[3]$ 的运算法则如下：

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix} \begin{pmatrix} v_0 \\ v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} m_{00}v_0 + m_{01}v_1 + m_{02}v_2 \\ m_{10}v_0 + m_{11}v_1 + m_{12}v_2 \\ m_{20}v_0 + m_{21}v_1 + m_{22}v_2 \end{pmatrix}$$

其结果矩阵 r 的第 i 行应为：

$$r[i] = \sum_{j=0}^2 m[i][j] * v[j]; \quad i = 0, 1, 2$$

若将 `multiply()` 定义为外部函数，则可写为：

```
Vector multiply( const Matrix & m , const Vector & v )
{
    Vector r;
    for(int i = 0; i < 3; i++)
        for(int j = 0; j < 3; j++)

            // 调用 Matrix 类的 elem() n² = 3² 次
            ↓
            r.elem(i) += m.elem(i,j) * v.elem(j);
            ↑                               ↑
            //调用 Vector 类的 Elem() n² = 3² 次 //调用 Vector 类的 Elem() n² = 3² 次

    return r; ← // 调用 Vector 类的 elem() n = 3 次
}
```

由于 `multiply()` 不是成员函数，故不能直接访问 `Matrix` 类的私有数据成员 `m[][]` 和 `Vector` 类的私有数据成员 `v[]`，只有通过各自的成员函数 `Vector::elem(int i)` 和 `Matrix::elem(int i, int j)` 访问。每当调用一次 `Multiply()` 函数时，`Vector` 类的 `elem()` 要被调用 $2n^2 + n = 3*(2 * 3 + 1) = 21$ 次（本例 $n = 3$ ），式中 n 为向量的元素个数。而 `Matrix` 类的 `elem()` 要调用 $n^2 = 3^2 = 9$ 次，从而产生很大的调用开销，直接影响了程序的运行效率，特别是当 n 增加时，调用次数将大幅度增加。如果把 `multiply()` 定义成友元函数，则可以有限制地实现对类的私有部分和保护部分的直接访问，不必再定义各自的成员函数 `elem()`。

例 3.22 友元函数的应用实例。

```
#include <iostream> // 使用 C++ 新标准的流库
using namespace std; // 将 std 名空间合并到当前名空间
/* 设定行数和列数都为 3，只需修改 ROWNUM 和 VOLUMENUM 的设置个数，适用于行数小于和等于列数的各种矩阵 */
const int ROWNUM = 3;
```

```

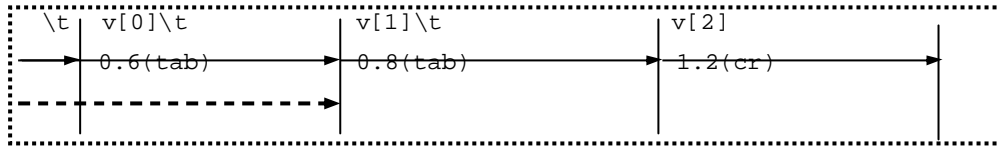
const int VOLUMENUM = 3;
class Matrix;
// 用 Vector 类描述一个行向量，设定它的行数与 Matrix 类矩阵的列数相等，否则不能相乘
class Vector {
    double v[VOLUMENUM];
public:
    Vector( void );                // 无参数构造函数
    Vector( double x, double y, double z ); // 一般构造函数
    void inputV( void );           // 用键盘给行向量的元素输入数值
    void dispV( void );           // 输出显示行向量各元素的值
    // ROWNUM 行 × VOLUMENUM 列的矩阵乘以 VOLUMENUM 行向量
    [...] friend Vector multiply(const Matrix &, const Vector &); [...]
};

class Matrix{
    double m[ROWNUM][VOLUMENUM];
public:
    Matrix( void );                // 无参数构造函数
    void inputM( void );           // 用键盘给矩阵的元素输入数值
    void dispM( void );           // 输出显示矩阵各元素的值
    [...] friend Vector multiply(const Matrix &, const Vector &); [...]
};

// 无参数构造函数，使行向量各元素的值设置为零
Vector::Vector( void )
{
    for( int i = 0; i < ROWNUM; i++ ) v[i] = 0;
}
// 一般构造函数，给行向量各元素赋初值
Vector::Vector( double x, double y, double z )
{
    v[0] = x; v[1] = y; v[2] = z;
}
void Vector::dispV( void )
// 为方便起见，显示格式采用如下向量形式： V( 5, 5, 5 )
{
    cout << " V(";                // 输出字母 v 和左圆括号
    for( int i = 0; i < VOLUMENUM; i++ ) {
        cout << v[i];              // 输出每个元素的值
        if(i != VOLUMENUM - 1) cout << " , "; // 各元素用逗号隔开
        else cout << ")\n\n";
        // 最后一个元素输出右圆括号和两个换行符
    }
}

// Vector 类的键盘输入格式：

```



```

void Vector::inputV( void )
{
    cout << "\n 每输入完一个元素按一次 TAB 键移动光标, ";
    cout << "\n 再输入下一个元素, 全部输入完后按 CR 键 !\n\n";
    // 输出每个行元素的提示信息。

    for( int i = 0; i < VOLUMENUM; i++ ) {
        cout << "\tv[" << i << "]" ;
        if( i == VOLUMENUM - 1 )      cout << "\n\t";
        /* 输出最后一个行元素的提示信息后要重新换行, 光标在新行还要移动一个 TAB 键的位置 */
        else                          cout << " "; // 否则输出空串。
    }

    for( i = 0; i < VOLUMENUM; i++ )
        cin >> v[i]; // 从键盘输入每个元素的值
}

Matrix::Matrix( void )
{
    int i, j;
    for(i = 0; i < ROWNUM; i++)
        for(j = 0; j < VOLUMENUM; j++)
            m[i][j] = 0; // 把矩阵的每个元素置为 0
}

// 显示格式:
void Matrix::dispM( void )
{
    int i, j;
    cout << " M(";
    

|            |
|------------|
| M( x, x, x |
| x, x, x    |
| x, x, x )  |


    // 输出字母'M和左圆括号

    for( i = 0; i < ROWNUM; i++ )
        for( j = 0; j < VOLUMENUM; j++ ) {
            cout << m[i][j]; // 输出每个元素的值
            if( j < VOLUMENUM - 1 )      cout << " , ";
            // 除最后一个列元素外, 各元素间用逗号隔开
            else if( j == VOLUMENUM - 1 && i == ROWNUM - 1 ) cout << ")";
            // 输出了最后一行的最后一个列元素后, 再输出右圆括号
            else cout << "\n ";
        }
    cout << "\n\n";
}

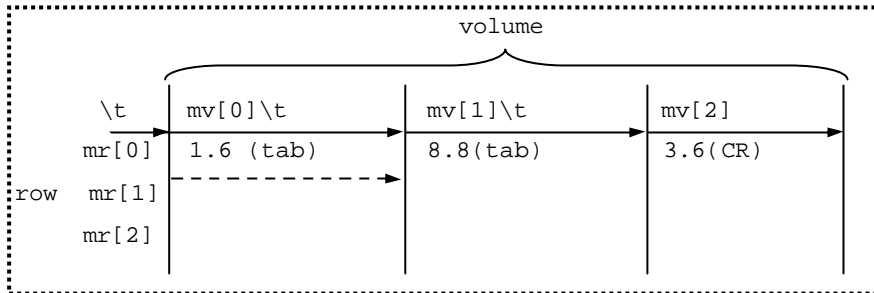
```



```

}
// Matrix 类的键盘输入格式：

```



```

void Matrix::inputM( void )
{
    cout << "\n 每输入完一个元素按一次 TAB 键移动光标，";
    cout << "\n 再输入下一个元素，一行输入完后按 CR 键 !\n\n";
    // 输出显示每列的提示信息
    for( int i = 0; i < VOLUMENUM; i++ ) {
        cout << "\tmv[" << i << "]" ;
        if( i == VOLUMENUM - 1 )    cout << "\nv";
        // 若是最后一个列元素的提示信息，则换行
        else                        cout << "";
        // 若不是最后一个列元素的提示信息，则输出空串
    }
    for( i = 0; i < ROWNUM; i++ ) {                // 输入所有行的元素
        cout << "mr[" << i << "]" \t";                // 输出显示每行的提示信息
        for( int j = 0; j < VOLUMENUM; j++ )        // 输入每行的列元素
            cin >> m[i][j];
    }
    cout << "\n";
}

/* ROWNUM 行 × VOLUMENUM 列的矩阵乘以 VOLUMENUM 行向量，其结果为一个 ROWNUM 行的行
   向量 */

Vector multiply( const Matrix & mt, const Vector & vc )
{
    Vector r;                // 存放结果的行向量 r
    for(int i = 0; i < ROWNUM; i++)
        for( int j = 0; j < VOLUMENUM; j++ )
            r.v[i] += mt.m[i][j] * vc.v[j];
        // 直接访问 Vector 类和 Matrix 类的私有数据成员
    return r;                // 以结果行向量 r 作为函数的返回值
}

```

```

void    main( void )
{
    Vector vt, re;           // 输出结果格式 :

    Matrix ma;
    vt.inputV( );
    ma.inputM( );
    ma.dispM( );
    cout << "          *\n\n";
    // *号前有 9 个空格。
    vt.dispV( );
    re = multiply(ma, vt);    // 调用友元函数的方法与普通函数一样
    cout << "          ||\n\n"; // *号前有 8 个空格
    re.dispV( );             // 输出显示结果
}

```

```

M( x, x, x
   x, x, x
   x, x, x )
  *
V (x, x, x )
  | |
V (x, x, x )

```

该程序的输出结果为：

每输入完一个元素按一次 TAB 键移动光标，
再输入下一个元素，全部输入完后按 CR 键！

```

v[0]    v[1]    v[2]
0.6     0.8     1.2(CR)

```

每输入完一个元素按一次 TAB 键移动光标，
再输入下一个元素，一行输入完后按‘CR’键！

```

      mv[0]    mv[1]    mv[2]
mr[0]  1.6     8.8     3.6(CR)
mr[1]  2.4     7.2     4.2(CR)
mr[2]  7.3     1.9     3.8(CR)
M(1.6,  8.8,  3.6
  2.4,  7.2,  4.2
  7.3,  1.9,  3.8)
  *
V(0.6 , 0.8 , 1.2)
  | |
V(12.32 , 12.24 , 10.46)

```

若在类体内声明一个普通函数，在原型前加上关键字 `friend` 就把它指定为该类的友元函数，它虽然不是成员函数，但可以访问该类的所有成员，特别是隐藏起来的私有成员。友元函数的定义则在类体外，因为它不是成员函数，不需要用“类名::”指定它属于哪个类。其作用是提高程序运行效率，相当于在类的封装墙上开了一个传递消息的孔。如例 3.22 中，把 `multiply()` 函数声明为 `Matrix` 和 `Vector` 两个类的友元函数，使得 `multiply()`

函数既能访问 `Matrix` 的私有数据成员，又可访问 `Vector` 类的私有数据成员，达到了“两个不同类的对象共享同一函数”的目的，这样一来，这两个类中不需要再设计读取数组元素的成员函数 `elem()` 了。

友元函数没有 `this` 指针，当它访问类的私有成员时，至少必须传递给它一个指向该类对象的指针或该类对象的引用，作为友元函数的一个参数替代 `this` 指针，在友元函数体内通过对象指针或对象引用才能访问私有成员。

例 3.23 传递一个指向该类对象的指针或该类对象的引用，作为友元函数的一个参数替代 `this` 指针。

```
#include <iostream> // 使用 C++ 新标准的流库
using namespace std; // 将 std 名空间合并到当前名空间

class Point {
    int x, y;
public:
    Point( int xi, int yi ) { x = xi; y = yi; }
    friend int compare( Point & a, Point & b );
};

// 替代 this 指针
↓
int compare( Point & a, Point & b )
{ return a.x * a.x + a.y * a.y - b.x * b.x - b.y * b.y; }
// 通过传递来的对象引用 a 和 b 访问私有数据成员 x 和 y

void main( void )
{ Point p(14, 17), q(25, 66);
  // 可像普通函数那样直接调用友元函数，不通过对象
  if( compare( p, q ) > 0 )
      cout << "q is closer to origin\n";
  else if( compare( p, q ) <= 0 )
      cout << "p is closer to origin\n";
}
```

友元函数可放在类的公有部分、保护或私有部分，但不管它放在哪个部分，它始终是开放的，可不通过对象直接调用。若不将 `compare()` 函数定义成友元，而定义为成员函数，可写为：

```
#include <iostream> // 使用 C++ 新标准的流库
using namespace std; // 将 std 名空间合并到当前名空间

class Point {
    int x, y;
public:
```

```
Point( int xi, int yi ) { x = xi;    y = yi; }
int compare( Point & b );
// 定义成 Point 类的成员函数, 只需一个参数 b, 另一参数由 this 指针传递
};

// 在类体外定义, 要用 "Point::" 指明该成员函数的所属类
int Point::compare( Point & b )
{   return x * x + y * y - b.x * b.x - b.y * b.y;   }
// 由于有隐含参数 this 指针, 在成员函数体内可直呼其名地访问 Point 类的成员

void main( void )
{   Point p( 14, 17 ), q( 25, 66 );
    // 通过对象 p 调用成员函数 Compare( )
    if( p.compare( q ) > 0 )
        cout << "q is closer to origin\n";
    else if( p.compare( q ) <= 0 )
        cout << "p is closer to origin\n";
}
```

3.7.2 友元类

C++允许将一个类的成员函数说明为另一个类的友元函数，该成员函数就可访问另一个类的所有成员，甚至可将整个类说明为另一个类的友元，称为“友元类”，该类的每个成员函数都可访问另一个类中的所有成员。

例 3.24 友元类例程。

```
#include    < iostream >           // 使用 C++新标准的流库
using namespace std;               // 将 std 名空间合并到当前名空间

class X {
    friend class Y;                 // Y 类是 X 类的友元类
public :
    void set( int I ) { x = i; }
    void display( void )
    {   cout << "x = " << x << " , " << "y = " << y << endl; }
private :
    int x;
    static int y;                  // 定义一个私有静态数据成员 y
};

class Y {
public :
    Y( int i, int j );             // Y 类的构造函数
    void display( void );
private :
    X a;                           // 对象成员，即 X 类的对象 a 作为 Y 类的成员
};

int X::y = 1;                      // X 类的私有静态数据成员 y 的初始化操作

Y::Y( int i, int j )
{   a.x = i;   X::y = j;   }      // 友类 Y 的成员函数体内可直接访问 X 类的私有成员

void Y::display( void )
{   cout << "x = " << a.x << " , " << "y = " << X::y << endl; }
    // 友类 Y 的成员函数可直接访问 X 类的私有成员 x 和 y (包括静态成员)

void main( void )
{   X    b;
    /* 定义 X 类的对象 b ，由默认构造函数使其私有数据成员 x 设置为零，静态数据成员 y 的初
       值为 1 */
    b.set(5);
    // 通过 X 类的对象 b 调用成员函数 set( ) 把数据成员 x 值设置为 5
    cout << "(1)";
```

```

b.display( );           // 输出显示 X 类的对象 b 的数据成员 x 和 y 值
Y    c( 6 , 9 );
/* 定义 Y 类的对象 c , 调用构造函数初始化数据成员 a (对象成员), 使得对象成员 a 的 x 和
   y 值分别为 6 和 9, 由于静态数据成员 y 的值被修改成 9, 它将影响 X 类的所有对象 */
cout << "(2)";
c.display( );           // 输出显示 Y 类的对象 c 的数据成员 a (对象成员)
cout << "(3)";
b.display( );
// 输出显示 X 类的对象 b 的数据成员 x 和 y 值, 此时静态数据成员 y 的值变成 9
}

```

该程序的输出结果为：

```

(1) x = 5 , y = 1
(2) x = 6 , y = 9
(3) x = 5 , y = 9

```

说明：

该程序把 Y 类说明为 X 类的“友类”，所以在 Y 类的成员函数中能直接访问 X 类的私有数据成员 x，即“a.x = i；”，a 是 X 类的对象，定义为 Y 类的私有对象成员。

在 X 类中又定义了一个静态数据成员 y，正如 3.5 节中所述，必须对它进行初始化。从例 3.24 中可知，在 Y 类的成员函数中也能直接访问 X 类的私有静态数据成员 y，因此友类 Y 可以直接访问 X 类的所有成员。

由于 X 类的数据成员 y 是静态的，通过 Y 类的对象 c 把它的值从原来值 5 变成 9 后，在 X 类的对象 b 中，y 成员的值也变成 9。由此可见，Y 类对象和 X 类对象共享静态数据成员 y。

3.8 标识符的作用域、可见性和名空间

3.8.1 标识符的作用域规则

ISO/ANSI C++ 新标准对此有如下规定：

“标识符”包括函数名、常量名、变量名、类名、对象名、成员名、语句标号名等。它们由编程者启用，是具有任意长度的字符数字序列，第一个字符必须是字母和下划线。标识符的作用域是能使用该标识符的程序部分，即一个程序段，它与标识符的可见性密切相关。

标识符的可见性是可以对该标识符进行访问（或称存取，Access）操作的为可见的，否则为不可见。

标识符作用域按其范围的大小可分为程序级、文件级、函数级、类级和块（Block，即复合语句）级等 5 种。程序级范围最大，块级最小。

3.8.2 作用域的种类

程序级作用域包含组成该程序的所有源文件,如外部函数和外部变量属于程序级作用域,它们在某个源文件中定义,经声明后在该程序的所有其他源文件中都是可见的,所以它是全局的,包含一个源程序中的所有其他作用域(文件级、函数级、类级和块级等)。

在所有块(block)函数体和类体以外定义的标识符具有文件级作用域,其作用域从定义点开始,到该源文件结束为止,例如内部静态函数和外部静态变量,以及用#define语句定义的宏指令和符号常量等。

在头文件中定义的标识符,其作用域可以扩展到包括头文件的任意源文件,但必须用#include语句纳入到各个源文件中,所以它也是全局的,包含一个源文件中的所有其他作用域(函数级、类级和块级等)。

类的作用域为类体,即类体的一对大括号所包围的程序部分,还包含该类的所有成员函数的定义范围,即函数头和函数体。在该范围内,一个类的所有成员函数都能访问同一类的任一其他成员,不管它们是公有的、私有的、还是保护的。

类的成员,即数据成员和成员函数为类作用域,而类名则是文件作用域。

友元函数名不是类作用域,而是文件作用域,它可以像普通函数那样在整个源文件中使用。

静态数据成员也是文件级作用域。

在一个复合语句即块语句内说明的标识符具有块级作用域,其作用域从说明点开始到复合语句的右大括号结束,如自动变量、内部静态变量和寄存器变量等。

如果块内还有一个嵌套块,那么外部块中的标识符作用域包括内部块。例如:

```

for(int i = 0; i < 6; i++){
    int x;
    // 内部块
    if(i) { x = 4; ... }
}
// 外部块

```

由于函数体在句法上可看成一个复合语句,所以函数中的绝大多数标识符是块级作用域,其中包括函数的形式参数。

在函数体内说明的语句标号是函数级作用域,语句标号是惟一具有函数级作用域的标识符。

只要语句标号在该函数体内作了说明,那么语句标号就可以在函数体内的任何地方使用,不必先说明后使用。例如:

```

...
for(int i = 0; i < 3; i++)

```

```

        for(int j = 0; j < 3; j++)
            if( num[i][j] < 0 )
                goto found;
        ...
found:    ... ;

```

语句标号在一个函数体内必须是惟一的，不管它放在哪个嵌套块中。例如：

```

void func( int a )
{
    x : cout << "At first label.\n";
    {
        // 一个嵌套块
        x : // 出错，重复定义的语句标号
        cout<<"At second label.\n";
    }
    if(--a > 0) goto x; // 究竟跳转到哪个 x : 所标识的语句
}

```

在不同的函数体内可使用相同的语句标号。在函数间不能使用 goto 语句转移。

在 C++ 中，通常标识符的可见性和存在性是基本一致的。只有在嵌套块的结构中，内层块的标识符覆盖了所有外层块的同名标识符，即在内层块的作用域范围内外层块的同名标识符不可见，但它们仍然存在，从而导致标识符的可见性和存在性不一致。如例 3.25 中，若将 main() 编为 Block #1，依此类推有 Block #2、Block #3。先在 Block #1 内定义了 3 个 int 型变量 a, b, c，又在 Block #2 中定义了同名变量 b 和 c，那么在 Block #2 的作用域内 int 型变量 a 仍然可见，但是在 Block #1 内原来定义的 int 型变量 b 和 c 却被隐藏起来变为不可见的，即分别被 Block #2 内定义的同名 int 型变量 b 和 float 型变量 c 所覆盖。进而在 Block #3 内又定义了同名 int 型变量 c，在 Block #2 内定义的同名 float 型变量 c 又被隐藏起来，且它覆盖了所有外层的同名变量。当退出 Block #3 时，其内的变量 c 消失了，Block #2 内定义的同名 float 型变量 c 恢复为可见的，且其值仍然为 8.8，依此类推……这里只有变量 a 在 Block #2 和 Block #3 中没有被同名变量覆盖，所以在这些块的作用域内都是可见的。变量 a 在 Block #2 中改变了数值(5 → 8)，并保持改变后的值，进入到 Block #3 以及后面的程序段。

例 3.25 标识符的可见性和存在期不一致。

```

#include    < iostream >           // 使用 C++新标准的流库
using namespace std;              // 将 std 名空间合并到当前名空间
void main( void )
{
    { int a(5), b(7), c(10);
      cout <<"Begin In Block #1 a = "<< a << ", b=" << b << ", c = " << c << endl;
      {
          { int b(8);
            float c(8.8f);
          }
      }
    }
}

```



```

        cout << "Begin In Block #2 a = "<<a<<" , b = "<<b<<" , c = " << c << endl;
#1    a = b;
        #2    {    int c;                                #3
                c = b;
                cout<<"In Block #3 a = "<<a<<" , b = " << b <<" , c = " <<c<<endl;
        }
        cout<<"End In Block #2 a = "<<a <" , b = " << b <<" , c = " << c << endl;
    }
    cout<<"End In Block #1 a = " <<a <<" , b = " << b <<" , c = " << c << endl;
}

```

该程序的输出结果为：

```

Begin In Block #1 a = 5, b = 7, c = 10
Begin In Block #2 a = 5, [b=8, c=8.8]
In Block #3 a = 8, b = 8,
c = 8
↓           ↓
End In Block #2 a = 8, [b=8, c=8.8]
End In Block #1 a = 8, b = 7, c = 10

```

如图 3.12 所示，标明了各标识符应属于哪一级作用域。例如，该类的类名 Buffer 为文件级作用域 (File scope)，数据成员名 str 和成员函数名 add 被封装在类体内为类级作用域 (Class scope)，而成员函数体内的自动变量 l 和形参 s 和 n 则是块级作用域 (Block scope)，外部 (全局) 对象名则是文件级作用域 (File scope)。

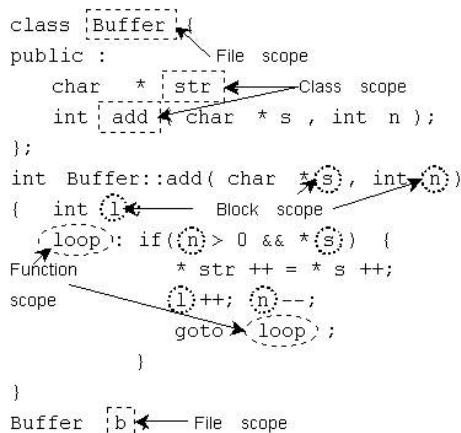


图 3.12 标识符的作用域

3.8.3 头文件

实用的程序经常采用工程文件，一个工程文件程序通常由多个源文件组成，每个源文件是一个可编译的程序单位。编程者在将程序分解成多个源文件时，必须规划好每个源文件中哪些信息在其他源文件内是可见的，哪些是不可见的。C++提供了在文件间开放和隐藏信息的方法，即指定变量或函数具有外部 (extern) 或静态 (static) 存储类型。

同一标识符的声明可使用多次，具有外部存储类型的声明语句如“extern int a;”等可以在多个源文件中分别使用多次，并将该标识符引入到这些源文件中。最好的办法是将其放在头文件中。例如，可把一个 Stack 类存放在 arystack.h 头文件中，头文件起着源文件与源文件之间的桥梁作用。在用到这些类、外部对象和外部函数的源文件内 (一

一般在文件的开头)写上:

```
#include "arystack.h"
```

若该源文件和 arystack.h 不在同一个子目录下,还必须包含路径名。

规划是否可以放在头文件中的经验规则(所谓“经验规则”是对#include 语句运行机制使用方法的一个合理建议,并非基本语法规定非要这么做不可)如下。

一般可放入头文件的程序部分有:

类型定义,如:

```
typedef unsigned char BYTE; 和 enum COLORS{BLACK, ..., WHITE};
```

函数声明,如:

```
[extern] int fun(int v); // 方括号包围的部分可缺省,即可省略不写的部分
```

内联函数定义,如:

```
inline int add(int a, int b) { return a + b; }
```

常量定义,如:

```
const int arraysize = 100;
```

全局变量和全局数组的声明,如:

```
extern int a;和extern char s[81];
```

预处理语句,如:

```
#include <iostream.h>
#define max(a, b) ((a > b) ? a : b)
```

注释,如:

```
/*check for End of File*/或//check for End of String
```

不适于放在头文件内的程序部分有:

普通函数定义,如:

```
int add(int a, int b) { return a + b; }
```

全局变量和数组的定义,如:

```
int k;和int a[6];
```

常量数组的定义,如:

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

例如,有一个编辑名为 area.prj 的工程文件,它由如下 3 个源文件组成:

```
#include "area.h" //circle.cpp 定义计算圆面积的函数
const double pi = 3.1415926;
double circle( double radius )
{ return pi * radius * radius; }
#include "area.h" //rect.cpp 定义计算矩形面积的函数
double rect(double width , double length)
```

```

{    return width * length;    }
-----
#include    < iostream >    //area.cpp 调用不同功能的函数计算面积

using    namespace    std;    // 将 std 名空间合并到当前名空间

#include "area.h"

void main( void )
{    double w , l;    // 定义两个 double 型变量 w 和 l 保存宽和高

    cout << "Please enter a width : ";
    cin >> w;    // 用键盘敲入一个实数保存在变量 w 中

    cout << "Please enter a length : ";
    cin >> l;    // 用键盘敲入一个实数保存在变量 l 中

    cout << "Area of the rectanle is " << rect(w, l) << endl;
    // 输出显示宽和高分别为 w 和 l 的矩形面积，其中包括调用 rect( )函数计算面积

    double r;    // 定义一个 double 型变量 r 保存半径值

    cout << "Please enter a radius : ";
    cin >> r;    // 用键盘敲入一个实数保存在变量 r 中

    cout << "Area of the circle is " << circle(r) << endl;
    // 输出显示半径为 r 的圆面积，其中包括调用 circle( )函数计算圆面积

}

```

这 3 个源文件中，都包含了自行定义的头文件。如：

```

double    circle( double radius );    // 在 area.h 文件内的函数声明语句
double    rect(double width , double length); // 在 area.h 文件内的函数声明语句

```

它只包含了各源文件内所定义的外部函数的声明语句，使这些函数可以在任意源文件中使用。凡是要调用这些外部函数的源文件，如 area.cpp，只需写上#include "area.h" 语句，就可达到信息共享的目的。

3.8.4 标识符的名空间

如前所述，编程者所启用的标识符，可用来作为函数名、常量名、变量名、类名、对象名、成员名、语句标号名等。当一个软件开发小组的许多人开发一个大型软件工程项目时，很容易引起所启用的标识符名发生冲突。例如，若在某个头文件中定义了这样一个常量：

```
const    int    MAXIMUM = 1024;
```

而在另一个头文件中又定义了一个同名的常量，即：

```
const    double    MAXIMUM = 99.99999;
```

那么，当某个工程项目同时包含这两个头文件时肯定会发生同名标识符的冲突。

为此，ISO/ANSI C++新标准引入了“标识符的名空间 (Namespace of Identifiers)”以及“名空间作用域 (Namespace scope)”。让编程者自行定义命名

不同的多个名空间，每个名空间的名称在所使用的程序区域内必须是惟一的，从而将 C++ 系统提供的惟一的、庞大的全局性名空间 (global namespace) 划分成多个名空间，处于不同名空间内的同名标识符不会发生冲突，即便它们出现在同一个编译单位的源文件中。例如：

```
namespace      sdm      {
    const double      MAXIMUM = 99.99999;
    class Buffer      { ... };
    Buffer getBuffer( void );
}
```

其中，sdm 为由编程者启用的该名空间的名称，紧跟其后的一对大括号所包围的区域为名空间体，这就构成了取名为“sdm”的名空间作用域，其内所列写的一个个实体，如 double 型常量名 MAXIMUM、类名 Buffer 和函数名 getBuffer 等就称为名空间的成员。这样一来，通过名空间声明就把 double 型常量名 MAXIMUM 和同名的 int 型常量名 MAXIMUM 区分开来。编程者可以采用如下 3 种方式访问该名空间的成员：

用 using 声明语句把一个名空间的所有成员都引入到当前作用域，即引入到编程者正在编写程序的作用域内，即：

```
void func( void )
{
    using namespace sdm;
    /* 用 using 语句把 sdm 名空间合并到当前名空间，在该函数体内的作用域内可直接使用 sdm
       名空间内的所有成员 */
    cout << MAXIMUM;           // 不用 “sdm::” 指明它所属的名空间
    Buffer getBuffer( void );   // 不用 “sdm::” 指明它所属的名空间
    ...
}
```

使用 using 声明语句只把名空间中的一个成员引入到当前作用域。例如：

```
void func( void )
{
    using sdm::MAXIMUM;
    // 用 using 语句只把 sdm::MAXIMUM 成员引入到当前名空间
    cout << MAXIMUM;           // OK，不用 “sdm::” 指明它所属的名空间
    Buffer getBuffer( void );   // 出错，必须用 “sdm::” 指明它所属的名空间
    ...
}
```

每次使用 sdm 名空间的成员用“sdm::”前缀明确地指明该成员所属的名空间。

例如：

```
void func( void )
{
    cout << sdm::MAXIMUM;      // OK，用 “sdm::” 指明它所属的名空间
}
```

```

Buffer buf = getBuffer( );
/* 出错，必须用“sdm::”指明它所属的名空间，即：
   “Buffer buf = sdm::getBuffer( )” */
...
}

```

1. 名空间的定义

名空间的定义格式如下：

<pre> namespace 名空间名 { < 名空间体 > } </pre>
--

其中，名空间名也是标识符的一种。一对大括号包围的区域称为“名空间体(namespace body)”，其内所声明的各种实体像变量、函数、结构类型和 class 类型等都称为名空间的成员，通过声明把这些成员引入到名空间所声明的区域中。

例 3.26 名空间的定义和使用。

```

1: namespace      M      {
2:     int      i;                      // 变量的说明
3:     int  g( int  a ) { return  a; }   // 函数 M::g( )的定义
4:     int  k( void );                  // 函数 M::k( )的声明
5:     void  f( void );                  // 函数 M::f( )的声明
6:     double h( double );              // 函数 M::h( )的声明
7: }
8: double M::h( double d ) { return  d; }
   // 在 M 名空间外编写函数 M::h( )的定义部分
9: namespace {                          // 非嵌套匿名的名空间
10:     int  r;                          // int 型变量 r 的说明语句
11: }
12: namespace      M      {
13: //     int      i;                      // 出错，变量 i 的重复定义
14:     int  k( void );                  // OK，函数 k( )的重复声明
15: //     int  f( void );
   // 出错，与 M::f( ) 函数的参数一样，仅返回类型不同
16:     namespace {                      // 内层匿名的名空间
17:         int  r = 1;
18:         int  s = 2;
19:     }
20:     int  g( char  c ) { return  r + c; }
   // M::g( int  a ) 的重载函数

```

```

21:         int k( void )
22:         { return g( i * s ); }           // OK, 函数 M::k( ) 的定义
23:     }
24: #include < iostream >                   // 使用 C++新标准的流库
25: using namespace std;                     // 将 std 名空间合并到当前名空间
26: void main( void )
27: { r = 6;
28:     cout << " Before M introduced r = " << r << endl;
    /* 在没有引入 M 名空间前, 非嵌套匿名的名空间内说明的 int 型变量 r 在这个源文件作
       用域内都是可见的 */
29:     cout << "(1) result of called g( int ) = " << M::g( r ) << endl;
30:     cout << "(2) result of called g( char ) = " << M::g( 'x' ) << endl;
    // 由于没有引入 M 名空间, 调用 M 名空间内的函数必须用 "M::g( r )" 格式
31:     using namespace M;                   // 将 M 名空间引入到当前名空间
32:     cout << " After M introduced r = " << r << endl;
33:     cout << "(3) result of called g( int ) = " << g( r ) << endl;
    /* 引入 M 名空间后, 非嵌套匿名的名空间内说明的 int 型变量 r 被 M 名空间内的同名变
       量 r 所覆盖, 访问的都是 M 名空间内的变量 r */
34:     cout << "(4) result of called g( char ) = " << g( 'x' ) << endl;
35:     i = 9;                               // 给 M 名空间的变量 i 赋初值为 9
36:     s = 6;
37:     cout << " i = " << i << endl;        // 输出显示变量 i 的值
38:     cout << "(5) result of called k( ) = " << k( ) << endl;
39:     cout << "(6) result of called h( ) = " << h( 3.14159 ) << endl;
40: }

```

该程序的输出结果：

Before M introduced r = 6	(第 28 行输出匿名的名空间内变量 r 的值)
(1) result of called g(int) = 6	(第 29 行输出)
(2) result of called g(char) = 121	(第 30 行输出)
After M introduced r = 1	(第 32 行输出 M 名空间内变量 r 的值)
(3) result of called g(int) = 1	(第 33 行输出, 实参取用 M 名空间内变量 r)
(4) result of called g(char) = 121	(第 34 行输出, 函数体内取用 M 名空间内变量 r)
i = 9	(第 37 行输出变量 i 的值)
(5) result of called k() = 54	(第 38 行输出调用函数 k() 的返回值)
(6) result of called h() = 3.14159	(第 39 行输出调用函数 h() 的返回值)

由此可知, 在定义一个新的名空间时应注意如下问题：

名空间的定义既可以编写在一个编译单位的源文件作用域内, 又可以写在另一个名空间体内, 例如,

```

namespace Outer {           // Outer 为外层名空间名
    int n = 6;
    int func( int num );
    namespace Inner {       // Inner 为内层名空间名
        double d = 9.996;
        int f( int j ) { return j + n; }
    }
}

```

这是一个嵌套结构的名空间定义，其外层名空间 Outer 作用域包括内层名空间 Inner 作用域。例如，外层名空间 Outer 作用域内定义的成员 int 型变量 n 在内层名空间 Inner 作用域内可以使用，但内层名空间 Inner 作用域内定义的成员 double 型变量 d 却不能使用。又如例 3.26 中，M 名空间内部又嵌套一个匿名的名空间（第 16 ~ 19 行）。

一个名空间的所有成员都必须在该名空间体内用原型声明，例如，函数原型声明、结构类型声明和 class 类型的声明等。而成员的定义部分既可以写在名空间作用域之内，也可以写在名空间作用域之外，但它必须是在成员声明点之后。例如，M 名空间定义的第 3 行函数 M::g() 的定义部分是在 M 名空间作用域之内，而第 6 行函数 M::h() 先用函数原型加以声明，然后在 M 名空间作用域之外，在该函数声明点之后第 8 行编写了函数 M::h() 的定义。

在一个编译单位的源文件作用域内，可以把名空间定义分割成几个部分，则第 1 次出现的定义部分称为“初始名空间的定义 (original namespace definition)”，其后续的定义部分则称为“扩展名空间的定义 (extension namespace definition)”。例如，M 名空间的定义被分成两个部分，第 1 部分从第 1 行至第 7 行，第 2 部分从第 12 行至第 23 行，这两部分应看成一个整体，其内的成员不能重复定义，如第 2 行和第 13 行出现变量 i 的重复定义；但成员的声明是可以重复的，如第 4 行和第 14 行重复声明函数 M::k() 是没有问题的。

编程者还可以在一个编译单位的源文件作用域内或者在另一个名空间体内，定义一个没有名字的名空间，称为“匿名的名空间 (Unnamed namespace)”，其格式为：

```
namespace { < 名空间体 > }
```

它实际上是取代如下两条语句：

```

namespace unique { < 名空间体 > }

using namespace unique;

```

为便于叙述，用 unique 作为这种无名空间的名字，以表示它不同于整个程序的其他标识符。不同之处就在于它在定义点后又用 using 声明语句把 unique 名空间引入到当前作用

域内,使得其内的成员在整个当前作用域内都是可见的,如第 9 ~ 11 行定义的一个匿名的名空间,其内的成员 `int` 型变量 `x` 在这个源文件作用域内都是可见的,如在主函数体内的第 27 行可用赋值语句给它赋初值为 6,从它的说明点(第 10 行)开始到它所在的源文件结尾,除掉发生同名变量冲突的区域(引入 `M` 名空间后的第 31 ~ 40 行)以外,它都是可见的。与静态变量相比,匿名的名空间具有更高级的功用,它使得像变量、对象和函数等实体在一个编译单位的整个源文件内都是可见的,虽然这些实体使用了外部链接(请参阅文献[25]),但在一个源程序所包含的其他源文件内却又是不可见的。

凡是在所有名空间和匿名的名空间以外说明的标识符名,以及函数名和 `class` 类名都具有“全局名空间作用域(global namespace scope)”,也称为“全局作用域”。从它们的说明点开始直到一个编译单位的整个源文件结尾处为它们的作用域。因此,在一个没有定义名空间的源程序中,正如我们以前所编写的各种例程,它们只有惟一的一个全局名空间,那些在所有函数体以外定义的外部对象、函数名和类名等就都具有“全局名空间作用域”,极易发生同名标识符名的冲突,特别在大型软件工程项目中,由多人同时进行开发工作的场合,这种矛盾就更为突出。

名空间还可以由编程者再启用一个名空间别名,所谓“名空间别名(namespace alias)”是一个已定义的名空间再声明一个替换名,其格式为:

<pre>namespace 名空间名 { < 名空间体 > } namespace 别名 = 名空间名;</pre>
--

当原来的名空间名很长时,可以采用一个简单的名空间别名代替之,例如:

```
namespace  UserControlBuffer  {    ...    }  
namespace  UCB = UserControlBuffer;
```

此后,UCB 就成为表示该名空间的标识符名。

2. using 声明语句

如前所述,使用 `using` 声明语句把另一个名空间的所有描述实体的标识符名引入到 `using` 声明语句所在的程序区域,其编写格式为:


```
using namespace 名空间名;
```

其中,“名空间名”所指的就是被引入的名空间,using 和 namespace 为关键字。这样一来,在 using 声明语句所在的程序区域内,就可以直接引用该名空间内的所有成员,而不需要使用前缀“名空间名::”指明它们所属的名空间。如例 3.26 中,在没有引入 M 名空间前的主函数体内(第 24~30 行),访问 M 名空间内的成员,如第 29 行的“M::g(r)”和第 30 行的“M::g('x')”都使用了前缀“M::”,当第 31 行用 using 声明语句引入了 M 名空间后,访问 M 名空间内的所有成员不需要使用前缀“名空间名::”直接引用,如第 33 行的“g(r)”和第 34 行的“M::g('x')”。

用 using 声明语句也可以引入一个嵌套结构的名空间,其格式为:

```
using namespace 外层名空间名::内层名空间名;
```

其中,用“外层名空间名::”作为前缀用来修饰内层名空间名。如前所述,using 声明语句还可以引入某名空间内的一个成员,即便它是在一个嵌套结构的内层名空间内,其格式为:

```
using namespace 外层名空间名::内层名空间名::成员名;
```

顺便指出,用前缀“::”表示“全局名空间(global namespace)”,有时需要把全局名空间内的某个成员引入到一个名空间。例如:

```
void fun( ); // 全局名空间内的函数 fun( )

...

namespace A {
void getVal( ); // A 名空间内的函数 getVal( )
}

namespace B {
using ::fun( ); // 把全局名空间内的函数“::fun( )”引入到 B 名空间
using A::getVal( ); // 把 A 名空间内的函数“A::fun( )”引入到 B 名空间
}

void home( )
{
    B::fun( ); // 实际上是调用全局名空间内的函数“::fun( )”
    B::getVal( );
    // 实际上是调用 A 名空间内的函数“A::getVal( )”,因为它已经成为 B 名空间的成员
}
}
```

对于初学者在使用 using 声明语句时应注意如下问题:

在大型软件工程项目中,由多人同时编写程序难免会发生同名标识符的冲突,再加上 C++ 系统本身的瑕疵,如多继承中常发生模棱两可的“二义性”问题,均是影响软件质量的重要因素,使用名空间和 using 声明语句是解决这类问题的最好办法。我们可以把同名的标识符放在不同的名空间内,再用 using 声明语句引入到所使用的程序区域,访问这些同名标识符的语句中,采用前缀“名空间名::”明确地指明该成员属于哪个名空间。例如:

```
namespace   UCB   {                // 定义一个 UCB 名空间
    ...
    typedef   int   Handle;        // Handle 作为 int 的新类型名
    ...
}
namespace   sdm   {                // 定义一个 sdm 名空间
    ...
    class   Handle { ... };        // Handle 是类名
    ...
}
void   func( )
{   using   namespace   sdm;        // 把 sdm 名空间的所有成员都引入到本函数体
    using   namespace   UCB;        // 把 UCB 名空间的所有成员都引入到本函数体
    ...
    Handle   hd;
    // 出错,出现“Handle 是类名还是新类型名?”的模棱两可的问题
    sdm::Handle   hd1;            // OK ! 定义 Handle 类的一个对象 hd1
    UCB:: Handle   hd2;            // OK ! 定义 int 型的一个变量 hd2
```

名空间名是标识符的一种,它必须在它所使用的程序区域内是惟一的。同一个名空间作用域内不要使用相同的标识符,也不要采用 using 声明语句把含有同名标识符的几个名空间引入到同一个程序区域,否则将产生同名标识符的冲突。如例 3.26 中的第 9 ~ 11 行定义了一个非嵌套结构的、匿名的名空间,其内说明了一个 int 型的变量 r,如前所述,匿名的名空间在定义时实际上立即被引入到它所在的程序区域,即在其定义以后的程序区域(第 12 ~ 40 行)内应该是可见的,但是由于在 M 名空间内的第 17 行又定义了,并在第 31 行用 using 声明语句把 M 名空间引入到主函数体内,从而导致第 31 ~ 40 行的程序区域内两个同名变量 r 的冲突,且 M 名空间内的变量 r 覆盖了另一个同名变量,即在该区域内访问的总是 M 名空间内的变量 r,而另一个变量 r 已无法再对它进行读取和写入的操作,这从程序的输出结果可清楚地看出。为此,在这种场合最好的办法是不要引入整个 M 名空间,而是用 using 声明语句有选择的引入本程序区域需要用的 M 名空间内的成员,

从而把同名的标识符排除在外。

正如 1.3 节所述，Visual C++ 遵循 ISO/ANSI C++ 标准的技术规范，将 C++ 原来的标准函数库中的所有实体名字即标识符（除了不遵循作用域规则的宏指令外）都声明在名称为“std”的名空间内，即这些标识符名都是该名空间的成员，std 名空间称为标准名空间（standards namespace）。编程者为了使用 C++ 新标准中的类库、函数库和流库等，通常使用 using 声明语句把 std 标准名空间引入到当前程序区域，详细情况请参阅 1.3 节。

3.9 对象数组和成员对象

3.9.1 对象数组

在 C 语言中，把具有相同结构类型的结构变量有序地集合起来便构成了结构数组。在 C++ 中，与此类似，将具有相同 class 类型的对象有序地集合在一起便构成了对象数组，对于一维对象数组也称为“对象向量”，因此，对象数组的每个元素都是同一种 class 类型的对象。

1. 对象数组的定义

对象数组的定义格式为：

```
<存储类> <类名> 对象数组名[元素个数] ... [= {初始化列表}];
```

其中，<存储类>是对象数组元素的存储类型，与变量一样有 extern 型、static 型和 auto 型等，该对象数组元素由<类名>指定它是哪一种 class 类型的对象。与普通数组类似，方括号内给出了某一维的元素个数即数组的大小。对象向量只有一个方括号，二维对象数组有两个方括号，如此类推。

例 3.27 对象数组的定义和使用。

```
#include <iostream> // 使用 C++ 新标准的流库
using namespace std; // 将 std 名空间合并到当前名空间

class Point {
    int x, y; // 两个私有数据成员 x 和 y 记录点坐标值
public:
    Point( void ) { x = y = 0; } // 无参数构造函数
    Point(int xi, int yi) // 具有两个参数的构造函数，或称一般构造函数
    { x = xi; y = yi; }
    Point(int c) { x = y = c; } // 具有单个参数的构造函数，或称类型转换构造函数
    void print( void ) // 输出显示点对象的坐标值
    { static int i = 0 ; // 用内部静态变量 i 记录点（对象）的序号
```

```
        cout << "P" << i++ << "(" << x << " , " << y << ")\n";
        // 以 "P(x , y)" 的格式输出显示点对象的坐标值
    }
};

void main( void )
{
    Point triangle[3] = {Point(0, 0), Point(5, 5), Point(10, 0)};

    int k = 0; // 用自动变量 k 记录两个三角形的序列号
    cout << "输出显示第" << ++k << "个三角形的 3 顶点 : \n";
    for(int i = 0; i < 3; i++) // 输出显示第 1 个三角形的 3 个顶点坐标值
        triangle[i].print( ); // 输出显示第 1 个三角形的 3 个顶点

    triangle[0] = Point(1);
    triangle[1] = 6; // 调用类型转换构造函数 Point(6)
    triangle[2] = Point(11, 1);

    cout << "输出显示第" << ++k << "个三角形的 3 顶点 : \n";
    for(i = 0; i < 3; i++) // 输出显示第 2 个三角形的 3 个顶点坐标值
        triangle[i].print( );

    Point rectangle[2][2] = { Point(0, 0), Point(0, 6), \
                               Point(16,6), Point(16,0)};

    cout << "输出显示一个矩形的 4 顶点 : \n";
    for(i = 0; i < 2 ; i++) // 输出显示一个矩形的 4 个顶点坐标值
        for(int j = 0; j< 2; j++)
            rectangle[i][j].print( );

    Point line45[3] = {0, 1, 2};

    cout << "输出显示 45 度直线上的 3 点 : \n";
    for(i = 0; i < 3; i++) // 输出显示 45 度直线上的 3 点坐标值
        Line45[i].print( );

    Point ptArray[3];

    cout << "输出显示对象向量 ptArray 的 3 元素 : \n";
    for(i = 0; i < 3; i++)
```

```

        ptArray[i].print( );
    // 输出显示对象向量 ptArray 的 3 元素 (点对象) 的坐标值
}

```

该程序的输出结果为：

输出显示第 1 个三角形的 3 顶点：

P0(0 , 0)

P1(5 , 5)

P2(10 , 0)

输出显示第 2 个三角形的 3 顶点：

P3(1 , 1)

P4(6 , 6)

P5(11 , 1)

输出显示一个矩形的 4 顶点：

P6(0 , 0)

P7(0 , 6)

P8(16 , 6)

P9(16 , 0)

输出显示 45 度直线上的 3 点：

P10(0 , 0)

P11(1 , 1)

P12(2 , 2)

输出显示对象向量 ptArray 的 3 元素：

P13(0 , 0)

P14(0 , 0)

P15(0 , 0)

2. 对象数组的初始化

当对象数组所属的类含有带参数的构造函数时,可用初始化列表按顺序调用构造函数初始化对象数组的每个元素。如例 3.27 中：

```

Point triangle[3] = {Point(0, 0), Point(5, 5), Point(10, 0)};
Point rectangle[2][2] = {Point(0, 0) , Point(0, 6),
                          Point(16,6) , Point(16,0)};

```

也可以先定义,然后采用赋值语句给每个元素赋值,其赋值格式为：

对象数组名[行下标][列下标] = 构造函数名(实参表);

例如：

```

rectangle[0][0] = Point(0, 0); 或   rectangle[0][0] = 0;
rectangle[0][1] = Point(0, 6);

```

```
rectangle[1][0] = Point(16, 6);
rectangle[1][1] = Point(16, 0);
```

与基本数据类型一样，在赋值操作时将进行“向左看齐”的自动类型转换，即右值表达式的数据类型将自动转换成左值表达式的数据类型，如“rectangle[0][0] = 0;”赋值语句，左值是 Point 类的一个对象，而右值却是常量 0，那么，在 Point 类体内必须编写一个类型转换的构造函数，它能将 int 型常量转换成 Point 类的一个（点）对象，且该点的两个坐标值相等，且都等于该 int 型常量值。

若对象数组所属类含有单个参数的构造函数，就得到一个把 int 型常量转换成 Point 类对象的类型转换构造函数，如例 3.27 中“Point(int c);”，则该构造函数置 x 和 y 为相同的值，在对象数组的初始化操作时也会自动调用这种类型转换构造函数，将 int 型常量自动转换成 Point 类的一个（点）对象，这样就可将说明语句简写成：

```
Point line45[3] = { 0, 1, 2 };
Point triangle[3] = { 0, // 调用类型转换构造函数 Point(0)
                    5, // 调用类型转换构造函数 Point(5)
                    Point(10, 0) };
```

创建对象数组时若没有初始化列表，则其所属类中必须定义一个无参数的构造函数，在创建对象数组的每个元素时自动调用它。如例 3.27 中，在执行“Point pt_array[3];”语句时，调用无参数构造函数 Point(void)，初始化对象数组 pt_array[] 的每个对象为 (0,0)。

如果对象数组所属类编写有多个重载构造函数，那么每当创建一个对象数组时，可以按每个元素的排列顺序（对于对象向量则按 0 号元素、1 号元素、2 号元素...排列顺序）调用与之相匹配的构造函数，每当撤销数组时，按相反的顺序调用析构函数。

例 3.28 对象数组中构造函数和析构函数的调用顺序。

```
#include <iostream> // 使用 C++ 新标准的流库
#include <string.h> // 标准函数 strcpy( ) 的原型声明在其中
using namespace std; // 将 std 名空间合并到当前名空间

class Personal {
    char name[20]; // Personal 类的私有数据成员，记录个人姓名信息
public :
    /* 类型转换构造函数，把形参字符串指针 n 所指的（姓名）字符串常量转换成 Personal 类
       的对象，其姓名信息存放在私有数据成员、字符串数组 name[20] 中保存 */
    Personal( char * n )
    {
        strcpy(name , n);
        // 把形参字符串指针 n 所指的字符串常量复制到 name[20] 中保存
        cout << name << " says hello !\n";
        // 输出显示创建哪一个对象（以姓名信息提示）时调用本构造函数
```

```

    }
    ~Personal( void )
    {   cout << name << " says goodbye !\n"; }
    /* 析构函数，撤销对象的操作由系统自动完成，输出显示撤销哪一个对象（以姓名信息提示）
       时调用本析构函数 */
};

void main( void )
{   cout << "创建对象数组，调用构造函数 : \n";

    Personal people[3] = {"Wang", "Li", "Zhang"};
    cout << "撤销对象数组，调用析构函数 : \n";

}

```

该程序的输出结果为：

创建对象数组，调用构造函数

Wang says hello !

Li says hello !

Zhang says hello !

撤销对象数组，调用析构函数

Zhang says goodbye !

Li says goodbye !

Wang says goodbye !

3.9.2 对象成员和容器类

当一个类的对象作为另一个类的成员时，该对象称为对象成员，另一个类称为“容器类(Container Class)”，这种方法称为“组合技术”。当创建容器类的对象时，作为对象成员所需的初始化参数应由容器类的构造函数提供，这是在它的构造函数头内，通过为对象成员指定一个初始化参数表来完成的，其格式为：

对象成员 1(参数表 1)，对象成员 2(参数表 2)，...

其中参数表为与对象成员所属类相对应的构造函数的参数表。

例 3.29 对象成员的初始化。

```

#include    < iostream >                // 使用 C++新标准的流库
using namespace std;                    // 将 std 名空间合并到当前名空间

class Foo {
    int i;                               // 定义 Foo 类的私有数据成员 i
public :
    Foo( void ) { i = 0; }               // 无参数构造函数
};

```

```

class Bar {
    int i;                                // 定义 Bar 类的私有数据成员 i

public :
    Bar(int x) { i = x; }
    // 类型转换构造函数, 将 int 型形参 x 转换成 Bar 的一个对象

    int get( void ) { return i; }
    // 读取 Bar 类的私有数据成员 i 的值
};

class Snafu {
    Foo f;                                // 对象成员 f 是 Foo 类的一个对象
    Bar b1;                                // 对象成员 b1 是 Bar 类的一个对象
    Bar b2;                                // 对象成员 b2 是 Bar 类的一个对象

public :
    Snafu( ) : b1(1), b2(2) { }
    // 容器类 Snafu 的构造函数, 含有对象成员的初始化参数表, 而函数体为空

    void read1( void )                    // 读对象成员 b1 的私有数据成员 i 的值
    { cout << "b1 of obj = " << b1.Bar::get( ) << endl; }
    void read2( void )                    // 读对象成员 b2 的私有数据成员 i 的值
    { cout << "b2 of obj = " << b2.Bar::get( ) << endl; }
};

void main( void )
{
    Snafu obj;                            // 定义容器类 Snafu 的对象 obj

    obj.read1( );
    /* 通过容器类 Snafu 的对象 obj 调用公有成员函数 read1( ), 读取对象成员 b1 的私有数
       据成员 i 的值*/

    obj.read2( );
    /* 通过容器类 Snafu 的对象 obj 调用公有成员函数 read2( ), 读取对象成员 b2 的私有数
       据成员 i 的值*/
}

```

该程序的输出结果为：

```

b1 of obj = 1
b2 of obj = 2

```

对象成员 b1 和 b2 的初始化参数表应放在所属类的构造函数 Snafu() 的头部, 并用一个冒号隔开, 各对象成员的初始化参数表 b1(1) 和 b2(2) 间用逗号隔开, 每个对象成员只能在初始化参数表中出现一次, 而对象成员 f 的构造函数 Foo() 没有参数, 所以对象成员 f 可以不写到初始化参数表中, 即可以缺省。

当创建容器类的对象, 例如执行 “Snafu obj;” 说明语句时, 编译系统自动地先

调用对象成员所属类相应的构造函数，执行初始化参数表 b1(1) 和 b2(2) 所规定的任务，然后再调用容器类的构造函数 Snafu()。而调用对象成员所属类构造函数的顺序取决于对象成员在容器类中定义的先后顺序，而非初始化参数表的排列顺序。为此，编程者应该使初始化参数表中对象成员的排列顺序与类体中对象成员的定义顺序保持一致，详见文献 [26] 条款 13。

撤销容器类对象时，先为容器类对象调用析构函数，再调用对象成员的析构函数。

对于容器类必须有一个构造函数，以便提供一个对象成员的初始化参数表 b1(1) 和 b2(2)，尽管该构造函数体为空函数也必须提供，因为编译系统不为任何容器类提供默认的构造函数。

必须在容器类中定义一些公有成员函数，用来访问对象成员中的私有数据成员，在容器类的这些成员函数体内，通过调用对象成员所属类的公有成员函数去访问对象成员的私有数据成员。在访问表达式中用对象成员所属的类名加作用运算符指明该成员函数，其格式为：

容器类对象成员名.对象成员所属的类名::成员函数名(实参表);

或

容器类的对象指针成员 -> 对象成员所属的类名::成员函数名(实参表);

式中的成员函数是对象成员所属类的，必须用对象成员所属类的类名加上作用域运算符指明。

例 3.30 访问容器类的成员。

```
#include    < iostream >                // 使用 C++ 新标准的流库
using namespace std;                    // 将 std 名空间合并到当前名空间

class Point {
    int x ,y;                            // Point 类的两个私有数据成员 x 和 y 记录坐标值
public :
    Point( int xi, int yi )
    {   x = xi;    y = yi;   }           // 一般构造函数
    Point( Point & p )                   // 复制构造函数
    {   x = p.x;    y = p.y;   }
    ~ Point( void ) {   }                // 析构函数，函数体为空
    int getX( void ) { return x; }       // 读 x 坐标值
    int getY( void ) { return y; }       // 读 y 坐标值
};

class Circle {                          // Circle 为容器类，center 是对象成员
```

```
Point center;
// 圆心用对象成员 center 表示, 它是 Point 类的一个对象。

int radius;

public :

    Circle(int x1, int y1, int r) : center(x1, y1)
    { radius = r; }

    int getRad( void ) { return radius; }
    // 读 Circle 类的私有数据成员 radius (半径) 的值
    int getCenterX( )          // 读圆心的 X 坐标值
    { return center.Point::getX( ); }
    int getCenterY( )          // 读圆心的 Y 坐标值
    { return center.Point::getY( ); }
};

void main( void )
{ Point pnt(160, 180), * bp;
  Circle spotlight(320, 240, 40), * dp;

  bp = & pnt;
  /* 令 Point 类的对象指针 bp 指向本类对象 pnt, 再通过 bp 调用本类的公有成员函数 */
  cout << "(1) pnt's X = " << bp -> getX( ) \
        << "\t Y = " << bp -> getY( ) << endl;

  dp = & spotlight;
  /* 令 Circle 类的对象指针 dp 指向本类对象 spotlight, 再通过 dp 调用本类公有成员函数,
     去访问对象成员 center 的私有数据成员 */
  cout << "(2) spotlight's X = " << dp -> getCenterX( ) \
        << "\t Y = " << spotlight.getCenterY( );
  cout << "\t radius = " << dp -> getRad( ) << endl;
}
```

该程序的输出结果为：

```
(1) pnt's X = 160          Y = 180
(2) spotlight's X = 320   Y = 240          radius = 40
```

3.10 对象的存储类

3.10.1 对象的生存期和存储区域

与基本数据类型的变量一样，在定义对象时也必须指明它的存储类。在 C++ 源程序中，可定义自动 (auto) 型对象、外部 (extern) 型对象、静态 (static) 型对象和动态对象 (又称堆对象，即用 new 和 delete 运算符创建和撤销的对象)。不同存储类的对象，其作用域和生存期也不相同。生存期与对象所在的存储区域密切相关，其存储区域如图 3.13 所示，有程序代码区 (CODE area)、数据区 (DATA area)、堆区 (HEAP area) 和堆栈区 (STACK area) 等，对应的生存期为静态生存期、动态生存期和局部生存期等。

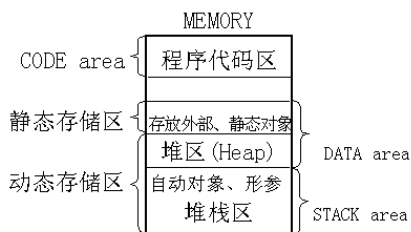


图 3.13 对象的存储区域

1. 静态生存期

具有静态生存期的对象，只要程序一开始运行或者运行到其定义点它们就存在，程序结束时才消失。它们存放在数据区中固定分配的内存空间，当程序没有进行初始化操作时，编译系统自动将其数据成员设置为零 (对数值型) 或空 (对字符串或指针)。如外部 (extern) 型对象、静态 (static) 型对象都具有静态生存期。

2. 动态生存期

使用 new 和 delete 运算符创建和撤销的对象 (包括变量)，以及调用 ISO/ANSI C 标准函数库中的 malloc() 和 free() 函数创建和撤销的变量具有动态生存期。当它们存放在内存的堆中，由 new 运算符为其分配 (或用 malloc() 函数为动态变量分配) 内存空间时，则生存期开始。当用 delete 运算符 (或用 free() 函数) 撤销它时，或者程序结束时生存期结束。

3. 局部生存期

自动对象和函数的形参具有局部生存期，它们存放在内存的堆栈区。若程序没有进行初始化操作，则其数据成员为随机值。

与基本数据类型的变量不同的是，每当创建该类的对象时，都将自动调用构造函数来实现每个新对象的初始化，每当撤销该类对象时都将自动调用析构函数 (若该类提供了析构函数) 或默认的析构函数，以完成对象存储空间的自动回收。

3.10.2 各种存储类的对象

1. 自动对象

与自动变量类似,在函数体或程序块(用一对大括号包围的程序片段)内定义的对象称为自动对象。定义它时就创建并自动调用构造函数来实现初始化,当程序退出定义它的作用域时,便自动调用析构函数或默认的析构函数撤销它。

2. 内部静态对象

与内部静态变量类似,在函数体或程序块内定义的静态对象称为内部静态对象。当程序执行到它的定义点时便创建它,并自动调用构造函数来实现初始化,当程序结束时便自动调用析构函数或默认的析构函数撤销它。

3. 外部对象和外部静态对象

它们都是全局型的对象,一旦程序启动,便按它们定义的先后次序创建,并依次调用构造函数进行初始化,当程序结束时按相反的次序调用析构函数撤销它们。

例 3.31 各种存储类对象的创建和撤销。

```
#include <iostream>           // 使用 C++新标准的流库
#include <cstring>             // 标准函数 strcpy( )的原型声明在其中
using namespace std;         // 将 std 名空间合并到当前名空间

class A {
    char string[50];
    // 字符串数组 string[50]作为 A 类的私有数据成员,存放对象名信息

public :
    A(char * st);              // 构造函数的声明语句
    ~A( void );               // 析构函数的声明语句
};

A::A( char * st )             // 构造函数的定义部分
{
    strcpy( string, st );
    /* 把形参字符串指针 st 所指的 ( 实参 ) 字符串常量复制到 A 类一个对象的数据成员
       string[50]中 */
    cout << string << "被创建时调用构造函数 !" << endl;
    // 输出显示创建某个对象时调用本构造函数的提示信息
}

A::~~A( void )
{
    cout << string << "被撤销时调用析构函数 !" << endl; }

void fun( void )
{
    cout << "在 fun( )函数体内 : \n" << endl;

    A    funObj( "fun( )函数体内的自动对象 funObj" ); // 定义一个自动对象 funObj
    static A    inStaObj( "内部静态对象 inStaObj" );
    // 定义一个内部静态对象 inStaObj
}
```

```

void main( void )
{
    A    mainObj("主函数体内的自动对象 mainObj");        // 主函数内的自动对象 mainObj
    cout << "主函数体内, 调用 fun( )函数前 : \n";        // 输出显示函数前的提示信息
    fun( );            // 调用普通外部函数
    cout << "\n 主函数体内, 调用 fun( )函数后 : \n";    // 输出显示函数后的提示信息
}

```

```

static A    exStaObj("外部静态对象 exStaObj");
A    gblObj("外部对象 gblObj");

```

该程序的输出结果为：

外部静态对象 exStaObj 被创建时调用构造函数 ！

外部对象 gblObj 被创建时调用构造函数 ！

主函数体内的自动对象 mainObj 被创建时调用构造函数 ！

主函数体内, 调用 fun()函数前 ：

在 fun()函数体内 ：

fun()函数体内的自动对象 funObj 被创建时调用构造函数 ！

内部静态对象 inStaObj 被创建时调用构造函数 ！

fun()函数体内的自动对象 funObj 被撤销时调用析构函数 ！

主函数体内, 调用 fun()函数后 ：

主函数体内的自动对象 mainObj 被撤销时调用析构函数 ！

内部静态对象 inStaObj 被撤销时调用析构函数 ！

外部对象 gblObj 被撤销时调用析构函数 ！

外部静态对象 exStaObj 被撤销时调用析构函数 ！

综上所述, 有：

自动对象的作用域仅在定义它的函数体或程序块内（一对大括号包围的程序片段），其作用域范围小，生存期也短。

内部静态对象的作用域虽然在定义它的函数体或程序块内，但其作用域范围小，生存期与作用域不一致，生存期却较长，从定义点开始一直到程序结束，它在作用域以外虽然存在但不可见。

外部静态对象的作用域与外部静态变量类似，是定义该对象所在的整个源文件，从定义点开始到文件结束。其生存期与作用域一致都比较长。

外部对象是在某个源文件中定义的，而它的作用域却是包含该源文件的整个源程序，其生存期与作用域一致，都是最长的。

总之，自动对象和内部静态对象（基本数据类型的变量也可看成基本数据类型的对象，

如 `int` 型的一个变量也可称为 `int` 型的一个对象, `double` 型也可称为 `double` 型的一个对象, 这样就把程序中的所有数据类型的实例都统一成对象) 统称为“局部对象”, 而外部对象和外部静态对象统称为“全局对象”。如前所述, 程序一开始执行就首先检查所有定义的 `class` 类型中的静态数据成员, 并给它们分配固定的内存空间。接着执行这些静态数据成员的初始化操作, 将初始值存放在对应的空间里。随后, 则寻找程序中定义的所有全局对象, 如例 3.31 中的外部静态对象 `exStaObj` 和外部对象 `gblObj`, 按照它们定义的先 (外部静态对象 `exStaObj`) 后 (外部对象 `gblObj`) 顺序创建并调用相匹配的构造函数进行初始化操作, 即给对象的所有数据成员赋初值。然后才从 `main()` 开始, 按顺序逐条执行主函数体内的每一条语句, 直到它的右大括号结束为止。在程序结束前, 先撤销主函数体内创建的自动对象, 按创建相反的顺序撤销, 每撤销一个对象调用一次析构函数 (如例 3.31 中的主函数体内的自动对象 `mainObj`), 接着调用析构函数撤销内部静态对象 (如例 3.31 中的内部静态对象 `inStaObj`), 最后, 按创建相反的顺序撤销全局对象 (如例 3.31 中的外部静态对象 `exStaObj` 和外部对象 `gblObj`)。

4. 动态对象

如图 3.13 所示, 全局变量、静态数据、常量存放在全局数据区 (Data Area), 所有类的成员函数和普通函数 (即非成员函数, 还包括友元函数) 的代码都存放在代码区 (Code Area), 为调用函数而分配的局部变量、函数参数、返回数据、返回地址等存放在堆栈区 (Stack Area), 余下的内存空间是自由存储区, 称之为堆区 (Heap Area)。

动态对象的创建。在 C++ 中, 可用 `new` 运算符动态地创建对象, 并为该对象在内存堆中分配存储空间, 取代了 ISO/ANSI C 老标准中的 `malloc()` 函数, 这类对象称为动态对象, 其定义格式为:

```
new <类型> (初值表)
```

或

```
new <类型>
```

它表明在堆 (Heap) 中动态建立了一个由 `<类型>` 所指定的对象。第 1 种形式由圆括号包围的“初值表”, 给出被创建对象的初始化值。也可采用不赋初始值的第 2 种形式, 但对象必须给它的数据成员赋初值后才能参加运算和操作, 第 1 种形式经常用来定义动态对象数组。这里所说的 `<类型>` 包含所有的基本数据类型和派生类型 (或称复杂的数据类型、构造类型), 以及 `class` 类型。New 运算符为任意类型的对象在堆 (Heap) 中分配一块所需的存储空间, 并返回该存储空间的首地址, 也可称为 New 运算符的运算结果值。当堆中没有足够的存储空间或分配出错时, 返回空 (NULL) 指针。因此, 与使用 `malloc()` 函数一样, 必须检测其返回值 (或称结果值) 不为空指针。在预先定义了一个同类型的指针后, 这种形

式便于采用 if 语句检测其返回值是否为空指针，可写为：

```
if ( ( 指针变量名 = new <类型> ) == NULL ) { ... }
```

例如：

```
if( ( p = new Point ) == NULL ) { ... }
```

顺便指出，与基本数据类型一样，在以后的程序中该指针名用来代替所创建的动态对象名，因该动态对象本身也是匿名的。

由于：

```

                                简化
if(p == NULL) (即 if(p == 0)) →  if( !p )
                                简化
if(p != NULL) (即 if(p != 0)) →  if( p )

```

一般格式可写为：

```
if( ! ( 指针变量名 = new <类型> ) ) { 出错处理操作; }
if( 指针变量名 = new <类型> ) { 创建成功的处理操作; }
```

由于 C 和 C++ 都是以“非零值”和“零值”来界定“逻辑真”和“逻辑假”，因此，第 1 个条件语句圆括号内的表达式是简化形式，把它还原后应该是“(指针名 = new 类型) == NULL”，即创建一个<类型>所指定的动态对象，并将 new 运算符的结果值赋给左值，即对象指针保存起来，然后再判断该指针是否为空指针，若它确实是空指针，该条件表达式成立，则其结果值为“逻辑真”，它的值为整数 1，当然应该执行 if 分支下的复合语句，该语句进行出错处理操作。而第 2 个条件语句还原后应该是“(指针名 = new 类型) != NULL”，即判断该指针不是空指针，当它确实不是空指针时，则其结果值为“逻辑真”，它的值为整数 1，执行 if 分枝下的复合语句，该语句当然进行创建成功的处理操作。有时，也可直接采用初始化操作创建新的动态对象，其格式为：

```
<类型> * 指针变量名 = new <类型> (初值表);
```

例如：

```
int * pi = new int;
Date * pd = new Date;
```

用 new 也可定义一个动态对象数组，new 的结果值是数组首地址，即第 0 号元素的地址，其格式为：

```
<类型> * 指针变量名 = new <类型> [元素个数];
```

例如：

```
Point * pt = new Point[3];
```

即创建了具有 3 个元素的 Point 类型的（动态）对象数组 pt，对单个对象也可以采用这

种格式, 只不过元素个数为 1。例如:

```
Point * px = new Point[1];
```

动态对象数组的初始化。用这种格式创建动态对象时, 只是给对象分配了内存空间, 不能对它赋初值。因此在使用前, 还必须用赋值操作对其赋值。其格式为:

```
类名 * 指针变量名;  
指针变量名[下标] = 构造函数名(参数表);
```

例 3.32 动态数组的创建、初始化和撤销。

```
#include <iostream> // 使用 C++ 新标准的流库
#include <cstring> // 标准函数 strcpy( ) 的原型声明在其中
using namespace std; // 将 std 名空间合并到当前名空间

class B {
    char name[80]; // 私有数据成员字符串数组 name[80] 记录某人姓名
    double b; // 私有数据成员 double 型变量 b 记录某人某类考试的得分值
public:
    static int count; // 静态数据成员 count 记录创建对象的个数
    B( char * s, double n ) // 一般构造函数
    {
        strcpy(name, s); b = n;
        /* 将形参字符串指针 s 所指的字符串常量复制给某对象的私有数据成员字符串数组
           name[80] (记录姓名), 并把第 2 个形参 n 赋值给另一个私有数据成员 b */
        cout << "调用一般构造函数, count : " << ++count << endl;
        // 将静态数据成员 count 值增 1, 即调用一次本构造函数对象个数加 1, 并输出显示
    }
    B( void ) { cout << "调用无参数构造函数, count : " << ++count << endl; }
    /* 无参数构造函数, 将静态数据成员 count 值增 1, 即调用一次本构造函数对象个数加 1,
       并输出显示 */
    ~ B( void )
    {
        cout << "调用析构函数撤销对象 " << name << "后, count : " \
            << --count << endl; // 调用一次本析构函数对象个数减 1, 并输出显示
    }
    void getb( char * s, double & n )
    {
        strcpy(s, name); n = b; }
    /* 把某对象的私有数据成员字符串数组 name[80] (记录姓名) 复制到形参字符串指针 s 所
       指的匿名字符串数组中, 再将某对象的另一个私有数据成员 b 赋值给第 2 个形参、引用 n
       的被引用变量中 */
};

int B::count = 0; // 将静态数据成员 count 值设置为零, 记录创建对象的个数

void main( void )
```



```

{   B *   p;           // 定义一个指向 B 类对象的指针变量

    double   d;
    charstr[80];
    /* 定义一个 double 型自动变量 d 和字符串数组 str[80]，以便在调用成员函数 getb( )
       时，保存两个数据成员的记录信息 */

    p = new B[3];
    // 调用无参构造函数，创建了 3 个 B 类的动态对象，所以 count 值为 3

    cout << "初始化对象数组 p[ ] !\n";
    p[0] = B( "ma", 4.8 );
    p[1] = B( "wang", 3.6 );
    p[2] = B( "li", 3.1 );
    /* 调用一般构造函数初始化每个元素，系统每初始化完一个元素就自动调用一次析构函数，
       撤销了因两次重复调用构造函数而产生的一个多余对象，使得 count 值仍保持为 3 */
    for( int i = 0; i < 3; i++ ) {
        p[i].getb( str, d );
        /* 在调用成员函数 getb( )，实参传递给形参的过程中，形参引用 n 是实参 d 的替换名，
           在函数体内对 n 的操作就是对 d 的操作，即把元素 p[i] 的私有数据成员 b 的值赋给了
           d */
        cout << str << " , " << d << endl;
    }
    cout << "\n";
    delete [ ] p;
    // 撤销动态数组 p[ ] 时，调用析构函数，按数组元素排列的相反顺序撤销各元素
}

```

该程序的输出结果为：

调用无参数构造函数，count : 1

调用无参数构造函数，count : 2

调用无参数构造函数，count : 3 （创建动态数组 p[] 的 3 个元素 count 为 3）

初始化对象数组 p[] ! （对同一个对象因两次重复调用构造函数使 count 多出一个）

调用一般构造函数，count : 4

调用析构函数撤销对象 ma 后，count : 3（系统自动调用一次析构函数使 count 为 3）

调用一般构造函数，count : 4

调用析构函数撤销对象 wang 后，count : 3

调用一般构造函数，count : 4

调用析构函数撤销对象 li 后，count : 3

ma , 4.8

wang , 3.6

li, 3.1

调用析构函数撤销对象 li 后, count : 2

调用析构函数撤销对象 wang 后, count : 1

调用析构函数撤销对象 ma 后, count : 0

说明:

使用“new <类型>[元素个数]”表达式创建动态对象时,在数组元素所属的类体中必须定义一个无参数的构造函数。

用上述动态数组的定义格式创建动态对象时,由于“[元素个数]”占据了“(初值表)”的位置而无法调用构造函数给对象(即对象数组的元素)赋初值,但是,在执行定义动态对象的说明语句,例如“p = new B[3];”时,已经调用了一次无参数的构造函数,静态数据成员 count 已增 1,接着用赋值操作对其赋值,例如“p[0] = B(“ma”, 4.8);”,右值表达式将对同一对象再次调用构造函数(只不过此时调用一般构造函数),势必产生一个多余的对象(即 count 值多出一个),系统会自动地调用一次析构函数,撤销该多余的对象,让 count 值恢复正确值。

成员函数 getb() 有两个形参,一个是形参字符串指针 s,另一个是形参引用 n,在调用该函数并将实参传递给形参的过程中,相当于执行了“char *s = str;”和“double &n = d;”语句,使得形参指针 s 指向了字符串数组 str,形参引用 n 成为实参 d 的替换名,在函数体内对 n 的操作就是对 d 的操作,即把元素 p[i] 的私有数据成员 b 的值赋给了 d。这种方法是利用形参指针和形参引用都具有的、在作用域切换时的接力作用,把对象数组每个元素的两个私有数据成员 b 和 name[] (为类作用域) 的值分别传递给主函数体内的普通变量及数组 n 和 str[] (为块作用域),然后通过后者输出显示。这是一种值得仿效的方法,可避免频繁调用公有成员函数去访问私有数据成员而造成程序执行的额外开销。

由输出结果可知,如果对象数组所属的类含有多个构造函数,那么每当建立动态对象数组时,按每个元素的排列顺序调用相匹配的构造函数,每当撤销数组时,按相反的顺序调用析构函数。

动态对象的撤销。

用 delete 运算符来撤销由 new 运算符创建的动态对象,即 delete 运算符只能作用于 new 返回的指针。其格式为:

delete	指针名;
--------	------

该指针保存的地址必须是 new 所分配的内存空间首地址。例如:

```
Date * pd = new Date;
```

```
...
delete pd;
```

顺便指出，delete 运算符可作用于 NULL 型指针，如上例可改写为：

```
pd = NULL;
delete pd;
```

用 delete 运算符撤销由 new 创建的数组时采用如下格式：

delete [] 指向数组的指针名;

这里所说的数组可以是基本数据类型的数组，或者是用户定义的、或者语言系统标准类库所定义的 class 类型的对象数组。指针名前只用一对空的方括号，可以忽略方括号内的数字(元素个数)。例如：

```
B * p;    double d;   char s[80];
p = new B[3];
...
delete [ ] p;
```

对一个用 new 创建的对象只能使用一次 delete 操作，否则将产生非法操作。

必须使用 new 和 delete 的原因在于：new 能自动调用类的构造函数初始化新创建的对象，delete 也能自动调用类的析构函数撤销对象，而 malloc() 和 free() 则不能，它们只能对基本数据类型的变量实现动态存储管理。并且 new 具有自动检测每种对象所需内存空间大小的功能，返回的 void 型指针会自动转换成与赋值号左边类型相同的指针，这比 malloc() 要优越。

小 结

C++中的 class 类型是 C 语言结构体的延伸，把结构类型引申成 class 类型，结构变量就引申为对象，并把成员函数纳入其中，这不是简单的重复，而是螺旋式上升到新的高度，发生了质的飞跃。初学者可参照表 3.2 用比较法加深理解。

表 3.2 C 语言的结构体和 C++类的对照表

数据类型	成员项	实 例	访问成员的方法
C 语言的结构体	数据成员	结构变量	结构变量名 . 成员名 结构指针名 -> 成员名

C++的类	public :	对象	对象名 . 公有数据成员名
	数据成员		对象名 . 公有成员函数名 (实参表)
	成员函数		对象指针名 -> 公有数据成员名
	protected :		对象指针名 -> 公有成员函数名 (实参表)
	数据成员		对象引用名 . 公有数据成员名
	成员函数		对象引用名 . 公有成员函数名 (实参表)
	private :		
	数据成员		
	成员函数		

类是抽象数据类型 ADT 的实现,类有两种成员:数据成员和成员函数,它们都可以放在公有部分、私有部分或保护部分。数据成员用来描述该类对象的属性,通常总是把它指定为私有的,以实现数据封装和信息隐藏;而将成员函数指定为公有的,作为该类对象访问私有数据成员的一条消息通路,提供给外界使用。初学者在设计一个类时,应遵循两个原则: 考虑程序的安全性把数据成员都放在私有部分; 公有成员函数是为用户提供服务的、完成该类所规定的各项任务和功能,也是访问本类所有私有数据成员的一条消息通路,常称为“接口 (interface)”所以,编程者应根据编程目的,努力使一个类的“接口”既完善且最小化。

成员函数由于隐含有 this 指针,其函数体内是访问本类成员的广阔天地,是对该类所有成员,特别是对私有数据成员进行操作的最重要场所,不管它们是公有的、私有的还是保护的。

用户定义了一个类也就创建了一个新的数据类型,即可把“类名”作为数据类型名直接去定义该类的对象。构造函数和析构函数是创建对象和撤销对象的特殊成员函数,构造函数可以重载,而析构函数只能有一个。系统自动地为无构造函数和无析构函数的类提供默认的构造函数和析构函数,以实现创建和撤销对象的管理运行机制,系统还为无复制构造函数的类提供默认的复制构造函数,但它只适用于浅拷贝,不能用于深拷贝。

静态数据成员可以取代所有的全局变量,但它不同于用 static 指定存储类的静态对象,而是 class 类型中特殊的数据成员,其特殊就表现在该类的所有对象只有一个静态数据成员值,起着连接该类所有对象的桥梁作用。而用 static 指定对象的存储类是论及对象所占内存空间的位置及其作用域、可见性和存在期等。应注意区分同样都是使用关键字 static,但所指的涵义却是两个不同的概念。静态成员函数无 this 指针专门负责访问静态数据成员,其直接调用法是打通被封装的私有数据成员消息通路的有力工具。若要使它能访问非静态数据成员,需传递一个该类的对象指针和对象引用或者对象本身以替代 this 指针。

友元不是成员函数而无 this 指针,其作用主要是提高程序运行效率和便于编程,在运算符重载中发挥着重要作用。

对象数组的每个元素都是同一 class 类型的对象,创建一个对象数组可用初始化列表按元素排列顺序调用元素所属类的构造函数,撤销时按相反的顺序调用析构函数。它

也具有与基本数据类型数组一样的属性。如数组名为该数组存储空间的首地址、数组和指向它的指针变量间的关系等式、访问每个元素都是使用统一的地址计算公式等。

容器类是应用广泛的实用化数据类型，是组合技术的具体实现。其构造函数应包含对象成员的初始化，应定义足够的公有成员函数访问对象成员的私有成员，特别是私有数据成员。

在大型软件工程项目中，使用名空间和 `using` 声明语句是避免同名标识符的冲突，消除模棱两可的“二义性”问题的最好办法。编程者可以自行定义命名不同的多个名空间，每个名空间的名字在所使用的程序区域内必须是惟一的，从而将 C++ 系统提供的惟一的、庞大的全局性名空间（`global namespace`）划分成多个名空间，处于不同名空间内的同名标识符不会发生冲突，即便它们出现在同一个编译单位的源文件中。采用 `using` 声明语句也能方便地把一个名空间或者它的一个成员引入到正在编写的程序区域。为了使用 C++ 新标准中的类库、函数库和流库等，初学者必须掌握如何使用 `using` 声明语句把 `std` 标准名空间引入到当前程序区域。

C++ 可定义自动对象、静态对象、外部对象和动态对象，它们具有不同的作用域、生存期和可见性。只有用 `new` 运算符创建的动态对象才能由编程者用 `delete` 运算符撤销，而其他存储类的对象（包括基本数据类型的变量）都是由系统自动地进行管理。

习 题 3

一、选择填空

1. 在下列关键字中,用以说明类中公有成员的是 ()。
A. public B. private C. protected D. friend
2. 下列的各类函数中,()不是类的成员函数。
A. 构造函数 B. 析构函数 C. 友元函数 D. 复制构造函数
3. 作用域运算符的功能是 ()。
A. 标识作用域的级别 B. 指出作用域的范围
C. 给定作用域的大小 D. 标识某个成员是属于哪个类
4. ()是不可以作为该类的成员的。
A. 自身类对象的指针 B. 自身类的对象
C. 自身类对象的引用 D. 另一个类的对象
5. ()不是构造函数的特征。
A. 构造函数的函数名与类名相同 B. 构造函数可以重载
C. 构造函数可以设置缺省参数 D. 构造函数必须指定类型说明
6. ()是析构函数的特征。
A. 一个类中只能定义一个析构函数 B. 析构函数名与类名不同
C. 析构函数的定义只能在类体内 D. 析构函数可以有一个或多个参数
7. 通常的复制构造函数的参数是 ()。
A. 某个对象名 B. 某个对象的成员名
C. 某个对象的引用名 D. 某个对象的指针名
8. 下列关于构造函数的描述中,()是错误的。
A. 构造函数的参数可以设置默认值
B. 在创建一个对象时构造函数自动被调用
C. 构造函数可以对静态数据成员进行初始化
D. 构造函数可以重载
9. 若有一个类 MyClass,执行“`MyClass a[3], * p[2];`”语句时会自动调用该类的构造函数 ()。
A. 2 次 B. 3 次 C. 4 次 D. 5 次
10. 若有一个类 MyClass,该类的复制(拷贝)构造函数的声明语句为 ()。
A. `MyClass & (MyClass agr)` B. `MyClass(MyClass agr)`
C. `MyClass(MyClass & agr)` D. `MyClass(MyClass * agr)`

11. 下面关于常量成员函数的描述中,()是正确的。
- A. 常量成员函数只能修改常量数据成员的值
 - B. 常量成员函数只能修改普通数据成员的值
 - C. 常量成员函数不能修改任何数据成员的值
 - D. 常量成员函数只能通过常量对象调用
12. 一个类的友元函数或友元类能够通过访问成员运算符“.”访问该类的()。
- A. 私有成员
 - B. 保护成员
 - C. 公有成员
 - D. 公有成员、保护成员和私有成员
13. 关于成员函数特征的下述描述中,()是错误的。
- A. 成员函数一定是内联函数
 - B. 成员函数可以重载
 - C. 成员函数可以设置缺省参数
 - D. 成员函数可以是静态的
14. 下面关于成员访问权限的描述中,()是错误的。
- A. 一个类的对象可以直接访问公有数据成员和公有成员函数
 - B. 类的私有数据成员只能被公有成员函数和该类的任何友元类或友元函数所访问
 - C. 只有类的成员函数或该类的派生类的成员函数和友元类或该友元函数可以访问保护成员
 - D. 在派生类的作用域内,可以直接访问基类的保护成员而不能访问它的私有成员
15. 在下面的类定义中,()是错误的语句。
- ```

Class Sample {
Public :
 Sample(int val); // A.
 ~ Sample(void); // B.
private :
 int a = 2.5; // C.
 Sample(void); // D.

```
16. 下面关于静态数据成员的描述中,( )是正确的。
- A. 静态数据成员是本类所有对象共享的数据
  - B. 类的每个对象都有自己的静态数据成员
  - C. 类的不同对象都有不同的静态数据成员
  - D. 静态数据成员不能通过本类的对象直接访问
17. 下述静态数据成员的特征中,( )是错误的。
- A. 说明静态数据成员时前边要加修饰符 static
  - B. 静态数据成员要在类体外初始化
  - C. 引用静态数据成员时,要在静态数据成员名前加<类名>和作用域运算符

- D. 静态数据成员不是所有对象共有的
18. 友元的作用是 ( )。
- A. 提高程序的运行效率                      B. 加强类的封装性
- C. 实现数据的隐藏                          D. 增加成员函数的种类
19. 已知类 A 中一个成员函数说明如下:
- ```
void set(A & a);
```
- 其中, A & a 的含义是 ()。
- A. 指向类 A 的指针为 a
- B. 将 a 的地址值赋给变量 Set
- C. a 是类 A 的对象引用, 用来作函数 set() 的形参
- D. 变量 A 与 a 按位逻辑与, 作为函数 set() 的参数
20. 下列关于对象数组的描述中, () 是错误的。
- A. 对象数组的下标是从 0 开始的
- B. 对象数组的数组名是一个常量指针
- C. 对象数组的每个元素是同一个类的对象
- D. 对象数组只能赋初值, 而不能被赋值
21. 下列定义中, () 是定义指向数组的指针 p。
- A. `int * p[5];` B. `int (* p)[5];`
- C. `(int *) p[5];` D. `int * p[];`
22. 已知: `print()` 函数是一个类的常成员函数, 它无返回值, 下列表示中, () 是正确的。
- A. `void print() const;` B. `const void print();`
- C. `void const print();` D. `void print (const);`
23. 关于 `new` 运算符的下列描述中, () 是错误的。
- A. 它可以用来动态创建对象和对象数组
- B. 使用它创建的对象或对象数组应使用运算符 `delete` 删除
- C. 使用它创建对象时要调用构造函数
- D. 使用它创建对象数组时必须指定初始值
24. 关于 `delete` 运算符的下列描述中, () 是错误的。
- A. 它必须用于 `new` 返回的指针
- B. 它也适用于空指针
- C. 对同一个动态对象可以使用多次该运算符
- D. 指针名前只用一对方括号符, 不管所删除数组的维数
25. 具有转换功能的构造函数, 应该是 ()。

- A. 不带参数的构造函数 B. 带有一个参数的构造函数
C. 带有两个以上参数的构造函数 D. 缺省构造函数

26. 试定义如下一个类 `MyClass`，在一个普通函数 `fun()` 体内把其对象的数据成员 `n` 的值修改为 50 的语句应该是 ()

```
MyClass {
    int n;
public:
    MyClass( int a ) { n = a; }
    void setNum( int b ) { n = b; }
}

int fun( )
{
    MyClass * ptr = new MyClass( 45 );
    _____;
}
```

- A. `MyClass(50)` B. `setNum(50)`
C. `ptr -> setNum(50)` D. `ptr -> n = 50`

27. 试定义如下一个类 `MyClass`，对定义部分中各语句描述正确的是 ()

```
class MyClass {
    int x, y, z;
public:
    void MyClass( int a ) { x = a; } //
    int fun( int a, int b )           //
    {
        x = a;
        y = b;
    }
    int fun( int a, int b, int c = 0 ) //
    {
        x = a;
        y = b;
        z = c;
    }
    static void setX( void ) { x = 10; } //
};
```

- A. 语句 `void MyClass(int a) { x = a; }` 是 `MyClass` 类的构造函数定义
B. 语句 `int fun(int a, int b)` 和语句 `int fun(int a, int b, int c = 0)` 实现类成员函数的重载
C. 语句 `static void setX(void) { x = 10; }` 实现本类数据成员 `x` 的更新操作
D. 语句 `int fun(int a, int b)`、`int fun(int a, int b, int c = 0)` 和 `static void setX(void) { x = 10; }` 都不正确

二、判断下列描述的正确性,对者划 , 错者划×

1. 使用关键字 `class` 定义的类中缺省的访问权限是私有 (`private`) 的。
2. 作用域运算符 “`::`” 只能用来限定成员函数所属的类。
3. 析构函数是一种函数体为空的成员函数。
4. 构造函数和析构函数都不能重载。
5. 说明或定义对象时,类名前面不需要加 `class` 关键字。
6. 访问对象的成员与访问结构变量的成员相同,使用运算符 “`.`” 或 “`->`”。
7. 所谓私有成员是指只有类中所提供的成员函数才能直接访问它们,任何类以外的函数对它们的访问都是非法的。
8. 某类中的友元类的所有成员函数可以存取或修改该类中的私有成员。
9. 可以在类的构造函数中对静态数据成员进行初始化。
10. 如果一个成员函数只需要存取一个类的静态数据成员,则可将该成员函数说明为静态成员函数。
11. 指向对象的指针与对象都可以作函数参数,但是使用前者比后者好些。
12. 对象引用作函数参数比用对象指针更方便些。
13. 对象数组的元素可以是不同类的对象。
14. 对象数组既可以在定义它的同时赋初值,又可以在以后赋值。
15. 指向对象数组的指针不一定必须指向数组的首元素。
16. “`const char * p;`” 语句说明了 `p` 是指向字符串的常量指针。
17. 一个类的构造函数中可以不包含对其对象成员的初始化。
18. 静态生存期的标识符的寿命是短的,而动态生存期标识符的寿命是长的。
19. 重新定义的标识符在定义它的区域内是可见的,而与其同名的原标识符在此区域内是不可见的。但是,它是存在的。

三、填空题

1. 在 C++ 中,定义一个 `class` 类型时,数据成员和成员函数的默认访问权限为_____。
2. 若含有对象成员的类定义中没有提供对该对象成员的初始化值,则调用该对象成员所属类的_____。
3. 在 C++ 中,每个类都隐含有一个指针称为_____,该指针指向_____。
4. 若一个类中含有对象成员,则在创建该类的一个对象时,首先调用的构造函数是_____。
5. 若已经定义了 `MyClass` 类,在主函数体内编写有 “`MyClass * p;`” 当执行 “`p=new`

MyClass;”语句时，将自动调用该类的_____。而在执行“delete p;”语句时，将自动调用该类的_____。

6. 非成员函数应声明为类的_____才能访问该类的私有成员。
7. 若把 FriendCalss 类定义成 MyClass 类的友元类，则应在 MyClass 类体内加入语句_____。
8. 下面的一个类定义了复制构造函数，请完成该类的定义和实现。

```
class MyClass {
    int x, y;
public:
    MyClass( int a = 0, int b = 0 )
    { x = a; y = b; }
    _____;
};
MyClass::_____
{ x = _____;
  _____;
}
```

9. 在下面程序代码中，一个类定义的构造函数和析构函数体内，申请和释放该类的私有数据成员 pa，请完成该类的实现部分。

```
class MyClass {
    int * pa;
public:
    MyClass( int a );
    ~ MyClass( void );
};
MyClass::MyClass( int a )
{ _____; }
MyClass::~ ~ MyClass( void )
{ _____; }
```

10. 下面程序通过把 Distance 类定义为 Point 类的友元类来实现计算两点间距离的功能，请完成该程序。

```
#include <iostream>
#include <cmath>
using namespace std;
class Point {
    float x, y;
public:
```

```

        _____;
Point( _____ )
{   x = a;   b = b;   }
void print( void )
{   cout << " X = " << x << endl;
    cout << " Y = " << y << endl;
}
};

class Distance {
public :
    float calDistance( Point & , Point & );
};

float Distance::calDistance( Point & a, Point & b )
{   float result;
    _____;

    cout << result << endl;
    return result;
}

main( void )
{   Point p( 10, 10 ), q( 20, 20 );
    Distance dis;
    dis.calDistance( p, q );
    return 1;
}
```

四、分析下列程序的输出结果

程序 1：

```
#include <iostream>
using namespace std;
class A {
    int a, b;
public :
    A( void );
    A(int i, int j);
    void print( void );
};

A::A( void )
{   a = b = 0;
```

```

        cout << "调用无参数的构造函数 !\n";
    }
    A::A(int i, int j)
    {
        a = i;    b = j;
        cout << "调用一般构造函数 !\n";
    }
    void A::print( void )
    {
        cout << "对象的a值：" << a << "，对象的b值：" << b << endl;
    }
    void main( void )
    {
        A    m , n(6, 8);
        m.print( );
        n.print( );
    }

```

程序 2：

```

#include    < iostream >
using namespace std;
class B {
    int a, b;
public :
    B( void );
    B(int i, int j);
    void printb( );
};
class A {
    B c;
public :
    A( void ) { }          // 函数体为空
    A(int i, int j);
    void printa( void );
};
B::B(int i, int j)
{
    a = i;    b = j;
    cout << "调用对象成员c所属类B的一般构造函数 !\n";
}
void B::printb( void )
{
    cout << "对象的a值：" << a << "，对象的b值：" << b << endl;
}
A::A(int i, int j) : c(i, j)

```

```
{    cout << "调用容器类 A 的一般构造函数 !\n";    }  
void A::printa( void )  
{    c.printb( );    }  
void main( void )  
{    A    m(16, 84);  
        m.printa( );  
}
```

程序 3:

```
#include    < iostream >  
using    namespace    std;  
class A {  
    double    total, rate;  
public :  
    A(double t,double r)        { total = t;    rate = r; }  
    friend double Count(A & a)  
    {    a.total += a.rate * a.total;  
        return a.total;  
    }  
};  
void main( void )  
{    A a1(160.6, 0.64), a2(76.8, 0.6);  
        cout << Count(a1) << "\t\t" << Count(a2) << endl;  
}
```

程序 4:

```
#include    < iostream >  
using    namespace    std;  
class A {  
    int a, b;  
public :  
    A( void );  
    A(int i, int j);  
    ~ A( void );  
    void set(int i, int j) { a = i;    b = j; }  
};  
A::A( void )  
{    a = b = 0;
```

```

        cout<<"调用无参数的构造函数，对象的 a = "<<a<<"，对象的 b = "<<b<<" ! \n";
    }
    A::A(int i, int j)
    {
        a = i;    b = j;
        cout<<"调用一般构造函数，对象的 a = "<<a<<"，对象的 b = " << b <<" ! \n";
    }
    A::~~ A( void )
    {
        cout <<"调用析构函数，对象的 a = "<< a <<"，对象的 b = " << b <<" ! \n";    }
    void main( void )
    {
        cout << "第 1 次测试开始 : \n";

        A    a[3];
        for(int i = 0; i < 3; i++)
            a[i].set(2 * i + 1, (i + 1) * 2);
        cout << "第 1 次测试完成 !\n\n";
        cout << "第 2 次测试开始 : \n";

        A    b[3] = { A(1, 2), A(3, 4), A(5, 6) };
        cout << "第 2 次测试完成 !\n\n";

    }

```

程序 5：

```

#include    < iostream >
using namespace    std;
class B {
    int x, y;
public :
    B( void );
    B(int i);
    B(int i, int j);
    ~ B( void );
    void print( void );
};
B::B( void )
{
    x = y = 0;
    cout << "调用无参数的构造函数 ! \n";
}
B::B(int i)
{
    x = i;    y = 0;
    cout << "调用类型转换构造函数 ! \n";
}

```

```
}
B::B(int i, int j)
{   x = i;   y = j;
    cout << "调用一般构造函数 ! \n";
}
B::~~ B( void )
{   cout << "调用析构函数撤销对象 ! \n"; }

void B::print( void )
{   cout << "对象的x值 = " << x << " , 对象的y值 = " << y << " ! \n"; }

void main( void )
{   B   * ptr;

    ptr = new B[3];
    ptr[0] = B( );
    ptr[1] = B(32);
    ptr[2] = B(64, 128);
    for( int i = 0; i < 3; i++ )
        ptr[i].print( );
    delete [ ] ptr;
}
```

程序 6 :

```
#include <iostream>
using namespace std;
class Complex{
    double real, imag;
public :
    Complex( void );           // Constructor(1)
    Complex( double real );    // Constructor(2)
    Complex( double real ,double imag ); // constructor(3)
    void print( void );
    void set( double r ,double i );
};

Complex::Complex( void )
{   set(0.0 ,0.0);
    cout << "Default Constructor(1) called.\n";
}

Complex::Complex( double real )
{   Set(real ,0.0);
```



```

        cout << "Constructor(2):real = " << real << " ,imag = " << imag << endl;
    }
Complex::Complex( double real ,double imag )
{
    set( real ,imag );
    cout << "Constructor(3):real = " << real << " ,imag = " << imag << endl;
}
void Complex::print( void )
{
    if( imag < 0 )
        cout << real << imag << "i" << endl;
    else
        cout << real << "+" << imag << "i" << endl;
}
void Complex::set( double r ,double i )
{
    real = r;    imag = i;    }
void main( void )
{
    Complex c1;
    Complex c2( 6 ,8 );
    Complex c3( 5.6 ,7.9 );
    c1.print( );
    c2.print( );
    c3.print( );
    c1 = Complex( 1.2 ,3.4 );
    c2 = 5;
    c1.print( );
    c2.print( );
    c3.print( );
}

```

程序 7 :

```

#include    < iostream >
using namespace std;
class Matrix {
    double m[3][3];
public :
    Matrix( void );
    double & elem(int i, int j) { return m[i][j]; }
    void dispm( void );
    friend Matrix invert(Matrix & );
}

```

```
};  
Matrix::Matrix( void )  
{   for(int i = 0; i < 3; i++)  
        for(int j = 0; j < 3; j++)  
            m[i][j] = 0;  
}  
Matrix invert(Matrix & t)  
{   Matrix r;  
    for( int i = 0; i < 3; i++ )  
        for( int j = 0; j < 3; j++ )  
            r.m[i][j] = t.m[j][i];  
    return r;  
}  
void Matrix::dispm( void )  
{   printf("\n");  
    for( int i = 0; i < 3; i++ )  
        for( int j = 0; j < 3; j++ )  
            printf("%.2f%s", m[i][j], (j < 2) ? ", " : "\n");  
}  
void main( void )  
{   Matrix    m, n ;  
    double x;  
    printf( "type 'TAB' key move curse !\n" );  
    for(int i = 0; i < 3; i++)  
        for(int j = 0; j < 3; j++) {  
            scanf("%Lf", &x);  
            m.elem(i, j) = x;  
        }  
    m.dispm( );  
    n = invert(m);  
    n.dispm( );  
}
```

程序 8 :

```
#include    < iostream >  
using namespace std;  
class Point {  
public :
```

```

    Point( int i, int j );
    Point(Point & p );
    ~ Point( void );
    int readX( void ) { return x; }
    int readY( void ) { return y; }
private :
    int x, y;
};

Point::Point( int i, int j )
{
    static int k = 0;
    x = i; y = j;
    cout << "一般构造函数第" << ++k << "次被调用 !\n";
}

Point::Point( Point & p )
{
    static int i = 0;
    x = p.x;    y = p.y;
    cout << "复制构造函数第" << ++i << "次被调用 !\n";
}

Point::~~ Point( void )
{
    static int j = 0;
    cout << "析构函数第" << ++j << "次被调用 !\n";
}

Point & func( Point & Q )
{
    cout << "\n已经进入到 func( )的函数体内 !\n";

    int x, y;
    x = Q.readX( ) + 10;
    y = Q.readY( ) + 20;
    cout << "定义对象 R, 则";

    static Point R( x, y );

    return R;
}

void main( void )
{
    Point M(20, 35), P(0, 0);
    Point N(M);
    cout << "下一步将调用 func( )函数, 请注意各种构造函数的调用次数 !\n\n";
    P = func(N);
    cout << "P = " << P.readX( ) << ", " << P.readY( ) << endl;
}

```

程序 9：若将程序 8 中的 func()函数改成：“Point func(Point & Q);”，其他都不变。

程序 10：若将程序 8 中的 func()函数改成：“Point & func(Point Q);”，其他都不变。

程序 11：

```
#include    < iostream >
using      namespace  std;
class      ClassOne    {
    int    i;
public :
    ClassOne( void );
    ClassOne( int val );
    ~ ClassOne( void );
    void print( void );
};

ClassOne::ClassOne( void )
{   cout << "Default  constructor  of  ClassOne !" << endl;
    i = 0;
}

ClassOne::ClassOne( int  val )
{   cout << "Constructor  of  ClassOne !" << endl;
    i = val;
}

ClassOne:: ~ ClassOne( void )
{   cout << "Destructor  of  ClassOne !" << endl;    }

void ClassOne::print( void )
{   cout << "The i of ClassOne' Object = " << i << endl;    }

    class ClassTwo    {
        ClassOne    myobj;
        int    i;
public :
        ClassTwo( void );
        ClassTwo( int  val );
        ~ ClassTwo( void );
        void print( void );
    };
};
```

```

ClassTwo::ClassTwo( void )
{   cout << "Default constructor  of  ClassTwo !" << endl;
    i = 0;
}
ClassTwo::ClassTwo( int  val )
{   cout << "Constructor  of  ClassTwo !" << endl;
    i = val;
}
ClassTwo::~ ~ ClassTwo( void )
{   cout << "Destructor  of  ClassTwo !" << endl;    }
void  ClassTwo::print( void )
{   cout << "The i of ClassTwo' Object = " << i << endl;    }
void main( void )
{   ClassTwo  objTwo( 16 );
    objTwo.print( );
}

```

程序 12：

```

#include    < iostream >           // 使用 C++新标准的流库
using namespace  std;             // 将 std 名空间合并到当前名空间
class   MyClass   {
    int  i;
public :
    MyClass( void );
    ~ MyClass( void );
    void  setValue( int  val );
};
MyClass::MyClass( void )
{   i = 0;
    cout << "Default  constructor  called !  i = " << i << endl;
}
MyClass::~ ~ MyClass( void )
{   cout << "Destructor  called !" << endl; }
void  MyClass::setValue( int  val )
{   i = val;
    cout << "i = " << i << endl;
}
void main( void )

```

```
{   MyClass   objArray[3], * p = objArray;
    for( int   j = 0; j < 3; j++ ) {
        p -> setValue( j + 1 );
        p ++;
    }
}
```

程序 13：把程序 12 的主函数改成下面的程序代码。

```
...
void main( void )
{   MyClass   * objArray[3];
    for( int   j = 0; j < 3; j++ )
        objArray[j] = new MyClass;
    for( j = 0; j < 3; j++ )
        delete objArray[j];
}
```

五、编程题

1. 假定有 UMPIRE 个裁判给体操比赛打分，参加比赛的运动员有 ATHLETE 名，现为比赛记分编写一个 Result 类，其程序代码如下，并把它存放在 result.h 的头文件中：

```
class Result {
    short   num;                // 运动员的编号
    char    name[10];           // 运动员姓名
    float   score[UMPIRE];
        /* 运动员得分，由 score[0] ~ score[UMPIRE - 2]记录各裁判的打分，
           score[UMPIRE - 1]记录平均分 */

public :
    Result( void );              // 无参数构造函数
    Result( short n, char * ps ); // 一般构造函数
    float maxRow( void );        // 求裁判打的最高分
    float minRow( void );        // 求裁判打的最低分
    float avg( void );           // 求运动员的平均分
    short readNO( void ) { return num; } // 读运动员编号
    void writeNO( int i ) { num = i; }   // 写运动员编号
    char * readName( void ){ return name; } // 读运动员姓名
    float readScore( int j ){ return score[j]; }
        // 读运动员得分

    void writeScore(int k, float av){ score[k] = av; }
```

```

        // 写入运动员得分
        friend void inSort( Result * pr, int n );
        // 按平均分由高到低排序
    };

```

要求：

去掉一个最高分和一个最低分，计算平均得分；

按运动员平均分从高到低排序输出显示；

在测试程序内（即 main() 函数体内）定义一个 Result 类的对象数组 r[ATHLETE]，它的每个元素记录着每个运动员的所有信息，各裁判的打分可用键盘敲入，但在屏幕上应有提示信息进行交互式操作；

排序算法采用“插入排序法”（参阅参考文献[25]）。

请编写一个完整的可运行源程序。

2. 现定义一个 Clock 类如下：

```

class Clock {
private :
    long    seconds;
public :
    Clock( void );
    Clock( int h, int m, int s );
    Clock( Clock & c );
};

...

void main( void )
{
    Clock    a( 9, 30 );        // (1)
    Clock    b = a;             // (2)
    Clock    c;                 // (3)
    c = b;                      // (4)
}

```

请指出(1) ~ (4)语句调用了哪个构造函数。

3. 将堆栈作为一种抽象数据类型，定义成如下类：

```

class Stack {                                // A stack of characters
    char data[100];
    int top;
public :
    void init( void ) { top = -1; }
    void push(char n);
    int pop( void );
}

```

```
int isEmpty(void) { return top == -1; }  
int isFull(void) { return top == 99; }  
};
```

它实现一个拥有 100 个字符的堆栈，试编写该类的成员函数程序代码，写成一个完整的可运行源程序。

4. 试编写 Time 类的类声明部分和成员函数的实现部分。具体要求如下：

包含小时 (hour)、分 (minute)、秒 (second) 等 3 个私有数据成员，
包含有设置 (成员函数名以 set 开头) 时、分和秒等数值的公有成员函数，
包含有读取 (成员函数名以 get 开头) 时、分和秒等数值的公有成员函数，
有按上午和下午各 12 小时或按 24 小时形式输出显示时间的公有成员函数，
有默认值的构造函数，默认值以 0 时 0 分 0 秒为准。

5. 定义整数集合类 IntegerSet，该类包含如下成员函数：

```
IntegerSet( );           // 类的构造函数，根据需要可定义多个重载的构造函数  
empty( );               // 清空该整数集合  
isEmpty( );             // 判断整数集合是否为空  
isMemberOf( );          // 判断某个整数是否在该整数集合中  
add( );                 // 把一个整数添加到整数集合中  
subset( );              // 从整数集合中删除一个整数元素  
isEqual( );             // 判断两个整数集合是否相等  
intersection( );        // 求两个整数集合的交集  
merge( );               // 求两个整数集合的并集  
print( );               // 按顺序输出显示该整数集合中的元素
```

该类还包含以下私有数据成员：

```
int element[100];       // 保存整数集合中的元素  
int endPosition;        // 指示整数集合的最后一个元素位置
```

注意：整数集合中不允许有相同元素存在。对于成员函数的参数和返回值可根据需要自行定义。

第 4 章 派生类、基类和继承性

继承性是面向对象程序设计的一个最重要的概念，理解“继承”是理解面向对象程序设计所有技术的关键。继承性允许在构成软件系统的层次结构中利用已存在的类并扩充它们，以支持新的功能，这使得编程者只需在新类中定义已存在类中所没有的成分来建立新类，从而大大提高了软件的可重用性（reusability）和可维护性。在面向对象程序设计中有一个公认的基本原则：设计一个类在尽可能提高代码可重用性的前提下应使程序代码尽可能简单化。对于客观世界中既有共性又有差别（个性）的两个类别以上的实体绝不是抽象成一个 class 类型来描述所有的实体，而是采用“继承（inheritance）”机制，先定义一个包含所有实体共性的 class 类型作为“基类”，然后，从该基类继承所有信息后再添加新信息（描述个性的部分），从而“扩展（extension）”出一些新的 class 类型，构成具有继承关系的类层次结构。

C++利用“继承机制（inheritance mechanism）”作为实现继承性的基础，它允许从现存的类中派生出新类，这些新类称为“派生类”或“子类”，前者则称为“派生类”的“基类”或“父类”。派生类继承了基类的全部成员（包括数据成员和成员函数），并且还可以增加基类所没有的数据成员和成员函数，以满足派生类特殊的应用要求。

4.1 继承的概念

4.1.1 什么是继承

继承是客观世界中实体间的一种关系。具有如下关键成分：

- 实体间共有的特征（共有的属性和状态）；
- 实体间的区别（个性）；
- 层次结构。

现实世界是分类分层的客观存在，物质可分为有机物和无机物，有机体又分为生命体和非生命体，生命体进而可分为动物、植物和微生物等。用继承来描述事物的层次关系，可帮助人们更准确地理解事物的本质。一旦搞清楚事物的层次结构，也就找到了解决问题的办法。在我们周围的世界里处处有继承，例如家庭成员。每个人从自己父（Dad）母（Mom）身上继承了相同的基本特征，如体魄、外形和举止习惯，而兄弟姐妹间又有不同之处。一个家庭

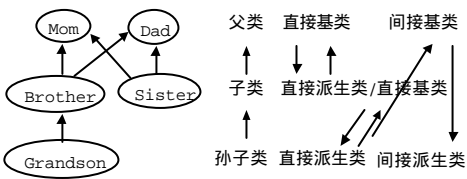


图 4.1 家庭内的继承关系

成员的关系是一种层次结构,如图 4.1 所示,由父类、子类和孙类构成了一个 3 层的类层次结构。父类是子类的直接基类,子类是父类的直接派生类,同时又是孙子类的直接基类,孙子类是子类的直接派生类。父类对孙子类,以及父类对曾孙类等都称为间接基类,反过来孙子类对父类及曾孙类等对父类都称为间接派生类。以后将直接基类和直接派生类分别简称为基类和派生类。

在编程方面,继承类似于家庭成员间的关系,只是它包含的是类与类之间的关系,一个类从另一个类中继承了所有属性。用这种技术是为了派生出一个新的类。用继承建立的类系统称为“类层次结构”,这与家庭内的继承关系完全类似。类的派生还可以无限继续下去,即派生类又可成为另一个派生类的基类。

如图 4.2 所示的类层次结构具有 5 层,第 1 层的 Point(点)类派生出第 2 层的 Rectangle(长方形)类和 Circle(圆)类,前者以长方形左上角顶点作为基准点确定它在屏幕上的位置,该基准点信息从 Point 类继承而来,后者以圆心作为基准点确定它在屏幕上的位置,它也是从 Point 类继承而来。而第 3 层的 Window(窗口)类是从 Rectangle(长方形)类中派生而得的,下面以窗口类作为基类,又可派生出第 4 层的 TextWindow(文本窗口)类和 GraphicsWindow(图形窗口)类。再类推下去,可派生出第 5 层的 TextEditor(文本编辑器)类和 IconEditor(图符编辑器)类。

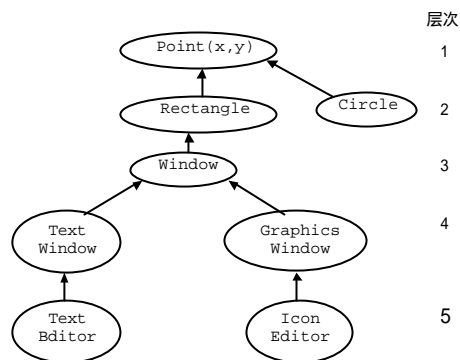


图 4.2 窗口的类层次结构

位置,该基准点信息从 Point 类继承而来,后者以圆心作为基准点确定它在屏幕上的位置,它也是从 Point 类继承而来。而第 3 层的 Window(窗口)类是从 Rectangle(长方形)类中派生而得的,下面以窗口类作为基类,又可派生出第 4 层的 TextWindow(文本窗口)类和 GraphicsWindow(图形窗口)类。再类推下去,可派生出第 5 层的 TextEditor(文本编辑器)类和 IconEditor(图符编辑器)类。

4.1.2 两种继承类型

抽象描述现实世界分类分层的类层次结构具有如下特点。

越在上层的类越具有普遍性和共性,越在下层的类越细化、专门化。

类层次的继承和多层类层次结构中继承的传递性。即派生类能自动地继承其基类的全部数据结构和操作方法。若类层次结构是多层的话,这种继承还具有传递性,即最下层的派生类能自动继承其上各层类的全部数据结构和操作方法。这是一种垂直的多层共享机制。

继承有两种类型,单继承和多继承。

1. 单继承

每个派生类仅只能有一个基类,称为“单继承”。如图 4.3 所示,基类和派生类构成了树形的类层次结构。

2. 多继承

允许派生类同时具有多个基类，称为“多继承”。这种基类和派生类的结构构成有向图的层次结构。在客观世界中确实也大量存在着这种关系，如图 4.1 所示的家庭关系就是一种多继承关系，可以抽象为图 4.4 所示的有向图结构。在自然界中，生物通常有父母双亲，这种安排使后代变异的可能性更大，从而具有更大的适应性和生存能力。在 OOP 中，多继承关系的描述却是一个十分复杂的问题，其复杂性就在于多继承关系会带来模棱两可的“二义性”问题。因此，在面向对象程序语言领域存在两大类别的语言处理系统：C++、Eiffel 和 Common LISP Object System (CLOS) 等，它们提供多继承，Java 只支持有限形式的多继承；Smalltalk、Object Pascal 等，不提供多继承。

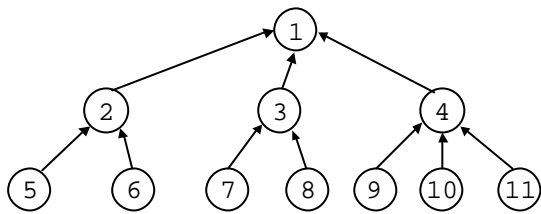


图 4.3 单继承构成了树形的类层次结构

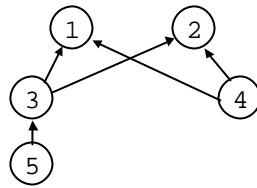


图 4.4 多继承构成的有向图结构

4.2 单继承的派生类

4.2.1 派生类的概念和定义

从任何已存在的类，无论是 C++ 标准类库提供的，还是用户自行定义的类中，都可以派生出新类，这些新类称为派生类 (Derived Class)；而已存在的用来派生新类的类称为基类 (Base Class)。派生类的定义形式和普通类定义基本一致，所不同的是派生类必须在类头部分指明它的基类，其形式如下：

```
class 派生类名 : <继承方式> 基类名
{
    <派生类新定义的成员>
}
```

其中，<继承方式>采用如下 3 个关键字指定：用 `public` 指定公有继承，其基类称为公有基类，该派生类称为公有派生类；用 `private` 指定私有继承，其基类称为私有基类，该派生类称为私有派生类；用 `protected` 指定保护继承，其基类称为保护基类，该派生类称为保护派生类。

例 4.1 派生类的定义格式。

```
#include    < iostream >           // 使用 C++ 新标准的流库
using namespace std;               // 将 std 标准名空间合并到当前名空间

class Base {
    float x, y;                     // 私有数据成员 x 和 y
public:
    Base( float x1, float y1 ) // 基类的构造函数
    {   x = x1; y = y1;   }
    void print( void )          // 输出显示私有数据成员 x 和 y 值
    {   cout << " x = " << x << "\n y = " << y << endl;   }
};
```

```

class Derived : public Base {
    float z; // 派生类 Derived 的新增数据成员

public:
    Derived( float x1, float y1, float z1 ) : Base( x1, y1 )
    {
        z = z1; // 派生类构造函数头中应包含基类的初始化列表
    }

    void print( void );
    /* 派生类重新定义的同名成员函数 Derived::print( ), 覆盖了从基类继承的
       Base::print( ), 即在派生类作用域内, 写"print( );"语句就是调用
       Derived::print( )成员函数 */
};

void Derived::print( void )
{
    Base::print( ); // 在派生类作用域内调用从基类继承的同名成员函数时, 需用“基类名::”指明

    cout << " z = " << z << endl ;
}

void main( void )
{
    Derived d1(3.0, 4.0, 5.0); // 定义派生类 Derived 的对象 d1
    d1.print( ); // 输出显示对象 d1 的数据成员 x、y 和 z 值
}

```

冒号把派生类与基类名分开, 并用它来建立派生类和基类之间的层次结构。

派生类在类体内也可以定义数据成员和成员函数。由于派生类继承了其基类的所有成员, 即基类的成员将自动地成为派生类的成员(共性, 可不再重复编写), 因此在派生类体中只列出新增的数据成员和成员函数(个性, 只编写新增加的)。例如, 可以用类 Base 来表示二维坐标中的点。

```

class Base {
    float x, y;
public:
    Base( float x1, float y1 )
    ...
};

```

Base 类用浮点数 x 和 y 表示点的坐标, 并定义了构造函数 Base(float x1, float y1) 设置点坐标以及显示点坐标的成员函数 print(), 那么从 Base 类可派生出描述三维坐标系点的 Derived 类如下所示:

```

class Derived : public Base {
    float z;

```

```
public:
    Derived( float x1, float y1, float z1 ) : Base( x1, y1 ){ z = z1; }
    void print( );
};
```

派生类的构造函数必须提供一个对基类数据成员进行初始化的参数表,写在派生类构造函数定义的第一行,即构造函数头所在行,其格式为:

构造函数名 (参数表) : 基类名 (参数名 1, 参数名 2, ...)
{ <函数体> }

派生类构造函数的参数表,必须包含从基类继承的数据成员和新增数据成员的初始化参数,这是在构造函数头后用冒号连接一个调用基类构造函数的初始化列表,显然,其中的“基类名 (参数名 1, 参数名 2, ...)”就是基类构造函数。如图 4.5 所示,对 Derived 类除了具有类体内自身定义的数据成员 `z` 和成员函数 `Derived::print()` 外,还继承了 Base 类中的所有数据成员 `x` 和 `y` 以及成员函数 `Base::print()`。

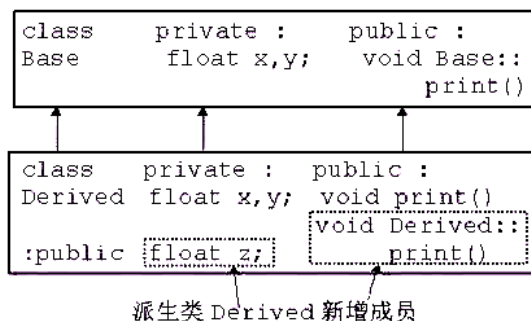


图 4.5 基类 Base 和派生类 Derived

在 `main()` 函数内, Derived 类的对象 `d1` 在生成时,调用派生类构造函数 `Derived()` 初始化所有数据成员 `x`、`y` 和 `z` 的值,可写为:

```
Derived    d1( 3.0, 4.0, 5.0 );
```

4.2.2 公有继承和私有继承

1. 继承方式的种类

前面讨论了派生类对基类的继承关系,在派生类的作用域内,从基类继承而来的基类成员作为派生类的成员,对于程序其他部分(例如对于所生成的对象)的访问权限是否发生了变化?派生类的对象是否可以直接访问基类的公有成员或私有成员呢?这是由在定义派生类、指定继承方式时,所用的关键字是 `public`、`private`,还是 `protected` 来决定的。如前所述,有公有继承、私有继承和保护继承,由它们继承而来的派生类分别称为公有、私有和保护派生类,具有直接关系的基类和派生类间访问权限如表 4.1 所

示。

从表 4.1 可知，对于公有继承，基类成员的访问权限在派生类中保持不变。换句话说，基类中的 `public` 成员在派生类中仍是 `public` 成员，`protected` 成员仍是 `protected` 成员，而基类的私有成员则变为不可见的，即在派生类作用域内，虽然从基类继承了私有成员，但却不能直接访问它们，而必须调用基类的公有成员函数才能访问到这些从基类继承的私有成员。

对于私有继承，基类成员的访问权限在派生类中都变成私有的。换句话说，基类中的

表 4.1 直接基类和直接派生类的访问权限

继承类型	基类中的访问权限	基类成员在派生类中的访问权限
public 公有继承	public private protected	public 不可见 protected
private 私有继承	public private protected	private 不可见 private
protected 保护继承	public private protected	protected 不可见 protected

`public` 成员和 `protected` 成员都是派生类的 `private` 成员。

对于保护继承，基类的所有公有成员和保护成员都作为派生类的保护成员，基类的私有成员仍然是私有的。由于保护继承用得很少，下面只讨论公有继承和私有继承。

2. 基类的私有成员在派生类的作用域内不可见

不管是哪种继承方式，派生类虽然继承了基类的所有成员，但派生类的成员函数并不一定能直接访问它们。具体来说，在派生类的成员函数体内，不能直接访问基类的私有成员。例如派生类 `Derived` 中的 `print()` 成员函数不能直接访问 `Base` 类中的私有数据成员 `x` 和 `y`。

```
void    Derived::print( void )
{
    cout << " x = " << x << "\n y = " << y << endl;
    // 出错，派生类的成员函数不能直接访问 Base 类中的私有数据成员 x 和 y
    cout << " z = " << z << endl ;
}
```

因此，派生类的成员函数也只能通过基类公有部分提供的成员函数访问基类的私有成员，特别是私有数据成员。所以正确处理 `Derived` 类中的 `print()` 函数应是调用基类 `Base` 中的公有成员函数 `Base::print()` 来访问基类的私有数据成员 `x` 和 `y`，其函数体

的定义如下：

```
void    Derived::print( void )
{
    Base::print( ); // 调用从基类继承的同名成员函数 print( ) 必须用 “基类名::” 指明
    cout << " z = " << z << endl ;
}
```

这里用 `Base::` 来标识 `print()` 是必须的，因为在派生类 `Derived` 作用域内，把基类的成员函数 `print()` 重新定义，即编写一个新的函数体（或称新的实现版本），也就是说，基类和派生类包含有同名的成员函数 `print()`，所以，必须用前缀“基类名::”指明究竟调用哪一个 `print()`。它告诉编译系统这里是调用基类 `Base` 的 `print()`，否则将引起 `Derived::print()` 的无约束条件限制的递归调用，形成无穷循环，导致死机。

不少初学者对此提出了这样的怀疑：既然派生类继承了基类的所有成员，但不管是哪种继承方式，从基类继承的私有成员在派生类作用域内又不可见，那不等于没有继承吗？其实，如果基类把这些成员放在公有部分（但这是很不安全的），则可明显看出这种继承关系，例如若 `x` 和 `y` 是基类的公有数据成员，那么，在派生类的成员函数体内就可以直接访问它们了，即便是基类的私有成员，派生类对象也可以通过它的公有成员函数去调用基类公有成员函数来访问从基类继承而来的私有成员，因此，这些成员在派生类作用域内被继承了是毫无疑问的，关键是如何打通消息通路，访问到它们。

3. 保护部分

如前所述，一个派生类从基类继承时，基类的所有私有成员不能被派生类成员函数直接访问，只有调用基类中的公有成员函数才能直接访问。例 4.1 中的派生类 `Derived` 的成员函数 `print()` 不能访问基类 `Base` 中的私有数据成员 `float x` 和 `float y`。如果希望派生类 `print()` 能够直接访问它们，也可将 `x, y` 定义在保护部分（`protected`）。

例 4.2 保护成员的使用。

```
#include    < iostream >           // 使用 C++ 新标准的流库
using namespace std;               // 将 std 标准名空间合并到当前名空间

class Base {
protected :
    float x, y;                    // 保护数据成员 x 和 y
public :
    Base( float x1, float y1 ) { x = x1; y = y1; }
    Void    print( void )
    {      cout << " x = " << x << "\n y = " << y << endl;    }
};

class Derived : public Base {
```



```

float z;                                // 派生类 Derived 的新增数据成员
public :
    Derived( float x1, float y1, float z1 ) : Base( x1, y1 ){ z = z1; }
    void    print( void );
};
void    Derived::print( void )
{   cout << " x = " << x << "\n y = " << y << "\n z = " << z << endl; }
// 在派生类 Derived 的成员函数体内, 可直接访问从基类继承而来的保护数据成员 x 和 y
void    main( void )
{   Derived    d1(3.0, 4.0, 5.0);
    d1.print( );
}

```

因此, 基类保护部分的成员 (数据成员和成员函数) 可以被派生类的成员函数直接访问, 但不能被程序的其他部分 (例如主函数) 或者其他类的成员函数直接访问。

4.2.3 基类对象和派生类对象

对象对其成员的访问操作是对象间最主要的传递消息手段, 因此, 应首先研究基类对象和派生类对象对其成员的访问操作。在不同的继承方式下基类对象和派生类对象对它们的各种成员的访问权限是有区别的。

1. 公有继承方式 (如图 4.6 所示)

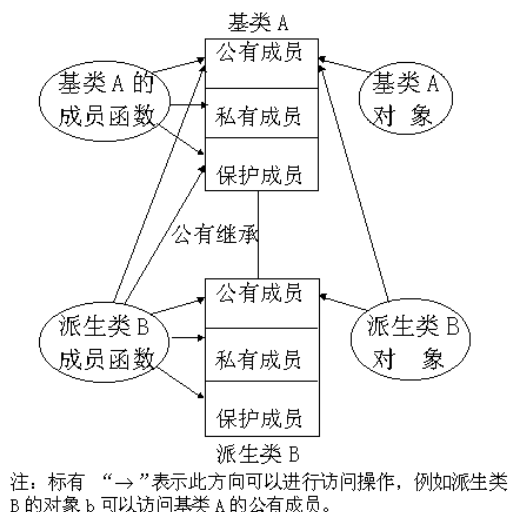


图 4.6 公有继承方式的访问限制

请读者特别注意, 我们先研究直接继承关系的变化规律, 然后再观察进入到间接继承关系的变化情况。由图 4.6 可知, 不论是基类还是派生类, 都是 class 类型。如前所述,

它们的非静态 (普通) 成员函数隐含着一个 `this` 指针, 即在成员函数体内可直接访问本类所有成员, 其中包括数据成员和成员函数, 这里是打通消息通路的重要场所。图中凡是标有箭头的都表示此方向可进行访问操作。显然, 基类对象和派生类对象要想访问它们各自的成员, 特别是隐藏起来的私有数据成员, 不管是派生类本身, 还是从基类继承的, 都必须在它们的成员函数体内去想办法, 寻找打通消息通路的途径。如基类对象访问本类的各种成员, 特别是隐藏起来的私有数据成员, 只有采用前述的办法, 通过基类对象去调用本类的公有成员函数, 在其成员函数体内可利用隐含的 `this` 指针直接访问这些私有数据成员。派生类对象访问本类新增成员也可利用本类的公有成员函数来达到目的, 但是, 那些从基类继承而来的成员, 特别是私有数据成员如何进行访问呢? 下面先研究公有继承方式下的直接继承关系的情况。

基类对象只能访问基类的公有成员 (包括数据成员和成员函数), 而不能访问基类的私有成员和保护成员。

例 4.3 在公有继承方式下, 基类对象和派生类对象对其成员的访问操作。

```
#include <iostream>          // 使用 C++ 新标准的流库
using namespace std;        // 将 std 标准名空间合并到当前名空间

class A {
    int a, b;                // 基类的私有数据成员 a 和 b
public:
    A( int i, int j ) { a = i; b = j; }      // 一般构造函数
    void move(int x, int y) { a += x; b += y; }
    void show( void )
    { cout << " A 类的对象 : (" << a << " , " << b << ")" << endl; }
};

class B : public A {         // 公有继承方式
    int x, y;                // 派生类新增的私有数据成员 x 和 y
public:
    B( int i, int j, int k, int l ) : A( i, j ), x( k ), y( l ) { }
    /* 函数体为空, 派生类新增数据成员 x 和 y 的初始化也可在其构造函数头上使用像对象成员
       那样的初始化列表来完成 */
    void show( void )
    { cout << " B 类的对象 : (" << x << " , " << y << ")" << endl; }
    void fun( void ) { move(3, 5); }
    //调用基类公有成员函数 move(), 因派生类中没有同名的成员函数 move(), 则可缺省 A::
};

void main( void )
{
    A e(1, 2);
```

```

/* 基类对象 e 不能直接访问本类的私有数据成员 a 和 b，只能通过本类的公有成员函数
   show( )来访问它们 */
e.show( );          // 输出显示基类对象 e 的数据成员值

B   d(3, 4, 5, 6);
/* 派生类对象 d 含有 4 个数据成员，其中 2 个从基类继承而来的私有数据成员 a (= 3) 和
   b (= 4)，另外 2 个是新增的私有数据成员 x (= 5) 和 y (= 6) */
d.fun( );           // 调用派生类的成员函数

d.A::show( );
// 在公有继承方式下，派生类对象 d 可直接调用从基类继承而来的成员函数 show( )

d.B::show( );
/* 在公有继承方式下，派生类 B 的对象 d 不仅可以直接访问本类的公有成员函数，即
   “d.B::Show( )；”，还可以直接访问基类 A 的公有成员函数，即“d.A::Show( )；” */
}

```

该程序的输出结果为：

A 类的对象：(1, 2)

A 类的对象：(6, 9)

B 类的对象：(5, 6)

派生类对象能访问本身类的公有成员，但不能访问本身类的私有成员和保护成员；可以访问基类的公有成员，但不能访问（从）基类（继承而来）的私有成员和保护成员。如例 4.3 中，由于在公有继承方式下，基类对象 e 不能直接访问本类的私有数据成员 a 和 b，故只有通过本类的公有成员函数 show()来访问它们。

在公有继承方式下，派生类 B 的对象 d 不仅可以直接访问本类的公有成员函数，如例 4.3 中的“d.B::show()；”语句，还可以直接访问（从）基类 A（继承而来）的公有成员函数，如例 4.3 中的“d.A::show()；”。因为，派生类继承了基类的所有成员，其中的公有成员函数在派生类作用域类仍然是公有成员函数，因此，派生类 B 的对象 d 当然可以直接调用它。在此特别指出，这是公有继承方式下一条非常重要、且极其有用的消息通路，由此可推出一系列很有应用价值的编程规则，使得公有继承方式获得广泛的采用，例如 Java 语言只有公有继承方式。

2. 私有继承方式（如图 4.7 所示）

与公有继承方式一样，基类对象只能访问基类的公有成员，而不能访问基类的私有成员和保护成员。

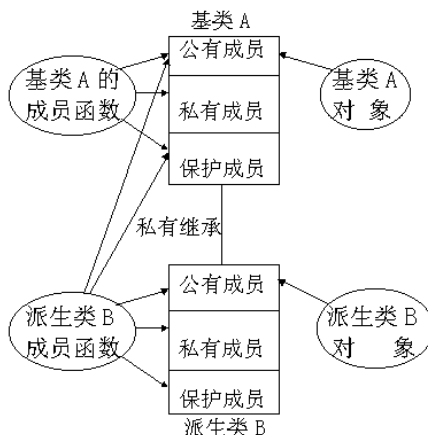
派生类对象只能访问本类的公有成员，不能访问本类的私有成员和保护成员。与公有继承方式不同，它不能访问基类的公有成员，因为基类的公有成员，特别是公有成员函数，已经变成私有成员函数了。同样更不能访问基类的私有成员和保护成员。

比较图 4.6 和图 4.7 可知，私有继承方式虽然比公有继承方式少了一条消息通路，即派生类对象不能访问基类的公有成员，但这是一条重要的消息通路，因为数据成员是用来描述对象的各种属性的，编程者随时希望掌握和控制这些属性，经常会读取或改变它们的属性值，因此派生类对象不仅要能方便地访问到本类的数据成员，也要能方便地访问到从基类继承而来的数据成员，特别是隐藏起来的私有数据成员。然而，由于在私有继承方式下，通过调用（从）基类（继承而来）的公有成员函数来访问（从）基类（继承而来）的私有数据成员这条消息通路被切断了，从而，造成了私有继承方式下消息通路的阻断，即图 4.7 右侧边的消息通路被全部阻断了，只有在图 4.7 左侧边的各消息通路上想办法打通消息通路。显然，派生类 B 成员函数是可以访问基类的公有成员和保护成员，编程者只有在派生类的公有部分再编写一些成员函数，在其成员函数体内调用基类的公有成员函数来访问这些私有成员，特别是私有数据成员，这样一来，“派生类对象访问（从）基类（继承而来）的私有数据成员”的消息通路就被打通了。

例 4.4 在私有继承方式下，基类对象和派生类对象对其成员的访问操作。

```
#include <iostream> // 使用 C++ 新标准的流库
using namespace std; // 将 std 标准名空间合并到当前名空间

class A {
    int a, b; // 基类的私有数据成员 a 和 b
public:
    A( int i, int j ) { a = i; b = j; } // 一般构造函数
    void move( int x, int y ) { a += x; b += y; }
    void show( void )
    { cout << " A 类的对象 : (" << a << " , " << b << ")" << endl; }
    class B : private A { // 私有继承方式
        int x, y; // 派生类新增的私有数据成员 x 和 y
    public:
        B( int i, int j, int k, int l ) : A( i, j ), x( k ), y( l ) { }
```



注：标有“→”表示此方向可以进行访问操作，例如派生类的成员函数可以访问基类 A 的公有成员。

图 4.7 私有继承方式的访问限制

```

/* 函数体为空，派生类新增数据成员 x 和 y 的初始化也可在其构造函数头上使用像对象成员
   那样的初始化列表来完成 */

void    show( void )
{    cout << " B类的对象 : (" << x << " , " << y << ")" << endl; }

void    fun( void ) { move(3, 5); }

void    f1( void ) { A::show( ); }

/* 在派生类 B 的公有部分内，编写公有成员函数 fun( )和 f1( )，在其函数体内，编写调
   用 (从) 基类 (继承而来的) 公有成员函数的语句。去访问从基类继承的私有数据成员 a 和
   b */

};

void    main( void )
{    A        e(1, 2);
/* 在私有继承方式下，基类对象 e 不能直接访问本类的私有数据成员 a 和 b，只有通过本类
   的公有成员函数 Show( )来访问它们，此时与公有继承方式一样 */
    e.show( );          // 输出显示基类对象 e 的数据成员值

    B    d(3, 4, 5, 6);
/* 派生类对象 d 含有 4 个数据成员，其中 2 个从基类继承而来的私有数据成员 a ( = 3 ) 和
   b ( = 4 )，另外 2 个是新增的私有数据成员 x ( = 5 ) 和 y ( = 6 ) */

    d.fun( );
    d.show( );
    d.f1( );
/* 派生类 B 的对象 d 也只能直接访问本类的公有成员函数，即"d.show( );"，既不能直接访
   问本类的私有数据成员 x 和 y，也不能直接访问基类 A 的公有成员函数 A::show( )和私有
   数据成员 a 和 b，只有通过本类的公有成员函数 B::show( )和 B::f1( )去访问 x、y 和
   a、b */

}

```

该程序的输出结果为：

A 类的对象 : (1 , 2)

B 类的对象 : (5 , 6)

A 类的对象 : (6 , 9)

如例 4.4 中，由于在私有继承方式下，基类对象 e 不能直接访问本类的私有数据成员 a 和 b，故只有通过本类的公有成员函数 show()来访问它们。派生类 B 的对象 d 也只能直接访问本类的公有成员函数，即“d.show();”，既不能直接访问本类的私有数据成员 x 和 y，也不能直接访问 (从) 基类 A (继承而来) 的公有成员函数 A::show()和私有数据成员 a 和 b，只有通过本类的公有成员函数 show()和 f1()去访问 x、y 和 a、b。

4.2.4 基类和派生类的成员函数

如图 4.6 和图 4.7 所示,根据类成员函数的基本特性,不管是公有还是私有继承方式,基类和派生类的成员函数都可以直接访问各自类的公有、私有和保护成员(包括数据成员和成员函数)。这里所说的“成员函数可直接访问”,是指在成员函数体内直呼其成员名地访问,而不必通过对象、对象指针和对象引用等来访问。

依此类推,不管是公有还是私有继承方式,派生类的成员函数都可以直接访问基类的公有和保护成员,而不能直接访问基类的私有成员(包括数据成员和成员函数)。

综合应用上述两个性质,参看图 4.7,可以在派生类的公有部分内设计成员函数去访问基类的私有成员,特别是私有数据成员,从而打通了派生类对象访问基类私有成员,特别是私有数据成员的消息通路。正如例 4.4 中,在派生类 B 的公有部分内编写公有成员函数 fun()和 fl()去访问基类的私有数据成员 a 和 b。

在私有继承方式下,基类的公有成员在派生类中都变成私有的,特别是成员函数变为私有后,切断了派生类对象访问基类的私有成员、特别是私有数据成员的消息通路。为此,C++还提供了一种“访问声明”的调整机制,其作用是在派生类作用域内恢复它们原来在基类中的访问权限,即原来在基类中是公有的,在派生类中重新恢复成公有的,其格式为:

```
class 派生类名 : private 基类名 {  
    ...  
public :  
    基类名::基类公有数据成员名;  
    基类名::基类公有成员函数名;  
    ...  
};
```

由于这是一种“访问声明”的调整机制,不是重载成员函数,所以成员函数名后面不能再写圆括号。

这种调整机制只能对直接派生类实现调整,而间接派生类可采用接力式调整,即逐级在每个直接派生类体内调整声明。

这种调整机制只能用于在派生类作用域内恢复所继承的成员原来在基类中的访问权限,而不能改变它们原来在基类中定义的访问权限,即原来是公有的可恢复成公有,但原来是私有的则不能改变成公有的。

例 4.5 “访问声明”调整机制的使用。

```
#include <iostream> // 使用 C++新标准的流库
```

```

using namespace std;           // 将 std 标准名空间合并到当前名空间

class A {
public :                         // 基类的公有成员函数
    void fun(int i) { cout << "Object's i = " << i << endl; }
    void go( void ) {      cout << "Call go( ).\n";      }
};

class B : private A {          // 私有继承方式
public :
    void hi( void ) {      cout << "Call hi( ).\n";      }
    // 新增公有成员函数
    A::fun;
    A::go;
    /* 把从基类继承的、因私有继承方式变成私有成员函数 fun( )和 go( )恢复成公有成员函数 */
};

void main( void )
{
    B d1;           // 创建一个派生类 B 的对象 d1
    d1.fun(6);      // 因 fun( )恢复成公有成员函数，则派生类 B 的对象 d1 可直接调用
    d1.go( );       // 因 go( )恢复成公有成员函数，则派生类 B 的对象 d1 可直接调用
    d1.hi( );       // 派生类 B 的对象 d1 可直接调用本类的公有成员函数 hi( )
}

```

4.2.5 C++结构体的继承

C++的 `struct` 是类的一种特殊形式，可以有成员函数（其中包含构造函数和析构函数），还可以有派生类。不管是 `class` 类型还是 `struct` 类型，在定义派生类时，类头部分冒号后的访问限制符 `public` 或 `private` 都可以缺省。如果派生类是 `class` 类型，`private` 可缺省；如果派生类是 `struct` 类型，缺省的是 `public`。这与类封装中的成员访问限制符缺省机制是一致的。例如，可写出两个 `struct` 分别表示点和圆，而表示圆的 `Circle` 结构类型是从表示点的 `Point` 结构类型公有继承而来的，两个类体内的成员都是公有的。

```

struct Point {
    int x , y;           // 缺省的是 "public :", 结构体内都是公有数据成员和成员函数
    void setLocn(int x1, int y1)
    {
        x = x1;    y = y1;    }
};

struct Circle : Point { // 缺省的是 "public :", 为公有继承方式
    int radius;

```

```
void    setSize(int r){ radius = r; }  
};
```

对于那些不需要考虑安全性、隐藏数据的场合，使用结构体是非常方便的，因为它具有与 `class` 一样的管理机制，可将所有成员都指定为公有的，以便于对象进行访问。

4.2.6 继承的传递性

如前所述，派生类对基类的继承具有传递性，即一种垂直的多层共享机制，最下层的派生类能自动继承其上各层类的全部数据结构和操作方法。例如：

```
class  A { ... };  
class  B : public A { ... };  
class  C : public B { ... };
```

如图 4.8 所示，A 类派生出 B 类，B 类又派生出 C 类，如前所述，A 类是 B 类的直接基类，B 类是 C 类的直接基类，而 A 类称为 C 类的间接基类，但底层的派生类 C 将继承所有上层基类 A 和 B 的成员。公有和私有继承的多层继承传递关系如表 4.2 所示。前面已讨论了直接基类和直接派生类间访问权限的变化情况，现从表 4.2 可知间接基类和派生类间访问权限的变化情况。

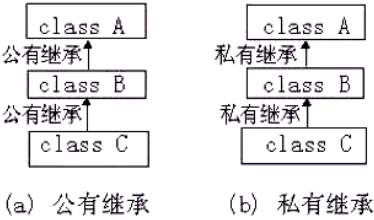


图 4.8 两种继承方式的传递

表 4.2 公有继承和私有继承的传递关系

继承方式	class A	class B: public A 的直接派生类	class C: public A 的间接派生类
公有继承	private protected public	不可见 protected public	不可见 protected public
私有继承	class A	class B: private	class C: private
	private protected public	不可见 private private	不可见 不可见 不可见

对公有继承方式，基类 A 和间接派生类 C 间的变化情况与基类 A 和直接派生类 B 间的变化完全一样，即基类 A 的私有成员在派生类 C 中不可见，而公有、保护成员没有变化，请参看例 4.6 和图 4.9。

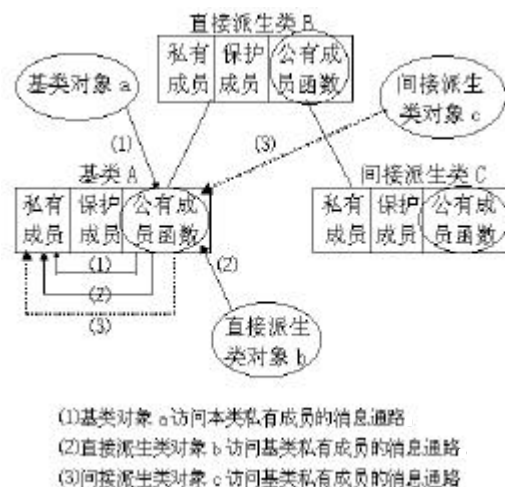


图 4.9 公有派生类对象访问基类成员

例 4.6 公有继承方式的传递性。

```

#include    < iostream >           // 使用 C++新标准的流库
using namespace std;              // 将 std 标准名空间合并到当前名空间

class A {
    int value;                     // 基类 A 的私有数据成员
public :
    A( int v ) { value = v; }      // 基类 A 的构造函数
    int readValue( void ) { return value; }
    // 读取基类的私有数据成员 value 值
};

class B : public A {               // 公有继承方式
    int total;
    // 继承基类 A 的私有数据成员 value , 直接派生类 B 新增了私有数据成员 total
public :
    B( int v, int t ) : A(v) { total = t; } // 直接派生类 B 的构造函数
    int readTotal( void ) { return total; }
    // 读取直接派生类 B 的私有数据成员 total 值
};

class C : public B {              // 公有继承方式
    int count;
    /* 不仅继承直接基类 B 的私有数据成员 total 还继承了间接基类 A 的私有数据成员 value ,
       间接派生类 C 新增了私有数据成员 count */
public :
    C( int v, int t, int c ) : B(v , t) { count = c; }
    /* 间接派生类 C 的构造函数, 其函数头只能在 “:” 后面写它的直接基类 B 的初始化列表, 通
  
```

```

        过接力方式来初始化从上层继承的所有数据成员 value 和 total, 即给它们赋初值 */
        int readCount( void ) { return count; }
        // 读取间接派生类 C 的私有数据成员 count 值
    };

    void main( void )
    {
        A a(2); B b(4, 6); C c(8, 10, 12);
        cout << " value of a = " << a.readValue( ) << endl; //
        cout << " value of b = " << b.readValue( ) << endl; //
        cout << " total of b = " << b.readTotal( ) << endl;
        cout << " value of c = " << c.readValue( ) << endl; //
        cout << " total of c = " << c.readTotal( ) << endl;
        cout << " count of c = " << c.readCount( ) << endl;
    }

```

该程序的输出结果为：

```

value of a = 2
value of b = 4
total of b = 6
value of c = 8
total of c = 10
count of c = 12

```

基类对象 a 通过调用基类 A 中的公有成员函数 readValue() 去访问它的私有数据成员 value, 这是公有继承方式下通过对象 a 访问基类 A 中的私有数据成员 value 的第一条消息通路。

直接派生类对象 b 可直接调用基类 A 中的公有成员函数 readValue() 去访问基类 A 中的私有数据成员 value, 这是公有继承方式下通过直接派生类对象 b 访问基类 A 中的私有数据成员 value 的第二条消息通路。

间接派生类对象 c 可直接调用基类 A 中的公有成员函数 readValue() 去访问基类 A 中的私有数据成员 value, 这是公有继承方式下通过间接派生类对象 c 访问基类 A 中的私有数据成员 value 的第三条消息通路。

读者也可根据前述原则, 找出对象 a、b、c 访问各自的 total 和 count 的消息通路。

对私有继承方式, 由表 4.2 可知, 基类 A 的公有、保护成员在直接派生类 B 中变成为私有的, 其私有成员则不可见, 继而在间接派生类 C 中基类 A 的所有成员都成为不可见的, 显然私有继承方式封装得很严实, 因此必须设法打通派生类对象, 访问基类 A 私有成员、特别是私有数据成员的消息通路。请参看例 4.7 和图 4.10。

例 4.7 私有继承方式的传递性。

```

#include <iostream> // 使用 C++ 新标准的流库

```

```

using namespace std;           // 将 std 标准名空间合并到当前名空间

class A {
    int value;                  // 基类的私有数据成员

public :
    A( int v ) { value = v; }   // 基类的构造函数

    int readValue( void ) { return value; }
    // 读取基类的私有数据成员 value 值

};

class B : private A {          // 私有继承方式

    int total;
    // 继承基类 A 的私有数据成员 value, 直接派生类 B 新增了私有数据成员 total

public :
    B( int v, int t ) : A( v ) { total = t; }
    // 直接派生类 B 的构造函数

    int readValue( void ) { return A::readValue( ); }
    /* 在私有继承方式下, 应在直接派生类 B 的公有部分内重新编写成员函数, 在该成员函数体内,
       编写调用基类 A 公有成员函数 readValue( ) 的语句 */

    int readTotal( void ) { return total; }
    // 读取直接派生类 B 的私有数据成员 total 值

};

class C : private B {

    int count;
    /* 不仅继承直接基类 B 的私有数据成员 total 还继承了间接基类 A 的私有数据成员 value,
       间接派生类 C 新增了私有数据成员 count */

public :
    C( int v, int t, int c ) : B(v , t) { count = c; }
    /* 间接派生类 C 的构造函数, 其函数头只能在 “:” 后面写它的直接基类 B 的初始化列表, 通过
       接力方式来初始化从上层继承的所有数据成员, 即给它们赋初值 */

    int readValue( void ) { return B::readValue( ); }
    /* 在私有继承方式下, 间接派生类对象 c 不能直接访问 B 类的公有成员, 更不能直接访问基
       类 A 中的公有成员, 应在 C 类的公有部分内, 设计成员函数进行接力式访问 */

    int readTotal( void ) { return B::readTotal( ); }
    /* 在私有继承方式下, 间接派生类对象 c 不能直接访问 B 类的公有成员函数 readTotal( ),
       应在 C 类的公有部分内, 设计同名公有成员函数, 在其函数体内才可以调用 B 类的公有成
       员函数 readTotal( ) */

    int readCount( void ) { return count; }
    // 读取间接派生类 C 的私有数据成员 count 值

};

```

```

void main( void )
{
    A a(2);    B b(4, 6);    C c(8, 10, 12);
    cout << " value of a = " << a.readValue( ) << endl;    //
    cout << " value of b = " << b.readValue( ) << endl;    //
    cout << " total of b = " << b.readTotal( ) << endl;
    cout << " value of c = " << c.readValue( ) << endl;    //
    cout << " total of c = " << c.readTotal( ) << endl;
    cout << " count of c = " << c.readCount( ) << endl;
}

```

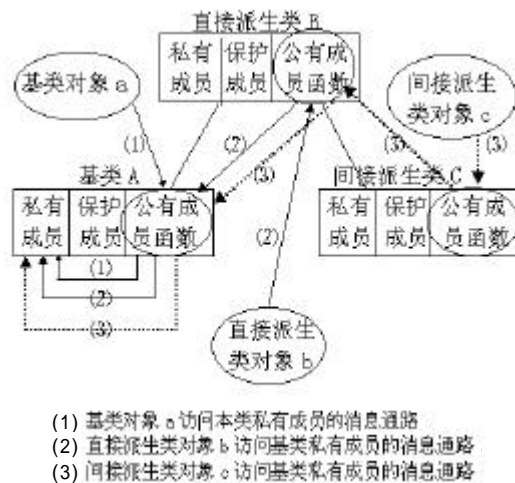


图 4.10 私有派生类对象访问基类成员

基类对象 a 通过调用基类 A 中的公有成员函数 readValue()，去访问它的私有数据成员 value，这是私有继承方式下，通过对象访问基类 A 中的私有数据成员 value 的第一条消息通路。

在私有继承方式下，直接派生类对象 b 不能直接调用基类 A 中的公有成员函数 readValue()，因为它已变成私有的（见表 4.2）。应在直接派生类 B 的公有部分内重新编写一个成员函数。在该成员函数体内编写调用基类 A 公有成员函数“A::readvalue()；”的语句，其调用格式为：

```
基类名::成员函数名( 参数表 );
```

例程中的“return A::readValue()；”语句用来访问基类 A 中的私有数据成员 value，这是在私有继承方式下通过对象访问基类 A 中的私有数据成员 value 的第二条消息通路。

在私有继承方式下，间接派生类对象 c 不能直接访问 B 类的公有成员，更不能直接访问基类 A 中的公有成员，应在 C 类的公有部分内设计成员函数进行接力式访问，间接

派生类对象 c 访问基类 A 中的私有数据成员 value 的消息通路如下：

```
(main( )函数体内) c.readValue( );    →    (C类体内) B::readValue( );
→ (B类体内) A::readValue( );    → (A类体内) readValue( ){return value;}
```

这样才能访问到间接派生类对象 c 中的 value 值。这是在私有继承方式下，通过对象访问基类 A 中的私有数据成员 value 的第三条消息通路。

从例 4.7 可知，C++ 允许派生类重新定义从基类继承的成员函数，即派生类定义了与基类同名的成员函数，称为派生类的成员函数覆盖了基类的同名成员函数。如果要在派生类中使用基类的同名成员，包括数据成员和成员函数，应使用如下格式：

```
基类名::数据成员名;
基类名::成员函数名(参数表);
```

在例 4.7 中，由于是私有继承，派生类 B 中由基类 A 继承的成员函数 readValue() 已变成私有成员函数，对象 b 不能直接访问它，只好在派生类 B 的公有部分内再定义一个同名的成员函数 B::readValue()，覆盖从基类继承的私有成员函数 A::readValue()。由于 B::readValue() 是公有的，故 B 的对象 b 可以访问它。然后在 B::readValue() 的函数体内调用基类 A 的同名成员函数 A::readValue() 访问（直接）基类 A 的私有数据成员 value。

4.3 派生类的构造函数和析构函数

4.3.1 派生类构造函数定义格式

当创建派生类对象时，由于派生类继承了基类的所有成员，派生类的每一个对象也包含了基类数据成员的值，就像派生类包含一个基类的对象作为无名成员一样，因此就必须提供一种机制，使得创建派生类的对象时能够自动初始化它所继承的基类数据成员，即给它们赋初值，在 C++ 中这是通过派生类的构造函数来实现的。

派生类构造函数的构成与包含对象成员的容器类 (Container Class) 的构造函数是类似的。将基类看成是派生类的一个无名成员，定义派生类构造函数时在函数头设置基类构造函数的初始化列表。

派生类构造函数应提供一种初始化机制，使得派生类对象在创建时，能自动地调用基类的构造函数来初始化从基类继承而来的数据成员，其格式为：

```
派生类名::派生类构造函数名(总参数表):基类构造函数名(参数表), 成员对象名(参数表)
{
    <函数体>
}
```

由此可知，基类构造函数由基类名来标识，派生类成员对象的构造函数由成员对象名

来标识, 派生类构造函数的总参数表应提供足够的参数对基类和成员对象的数据成员进行初始化, 而派生类新增数据成员的初始化既可以在其构造函数的初始化列表中进行, 也可以在其构造函数体内用赋值语句来完成, 此时甚至可以不从构造函数的总参数表中获取初值, 例如用一个计算表达式的结果作为右值赋给它 (详见习题 6 第三题之 5)。

例 4.8 基类和派生类的构造函数。

```
#include    < iostream >           // 使用 C++新标准的流库
using namespace std;               // 将 std 标准名空间合并到当前名空间

class Base {
    float x, y;                     // 基类的私有数据成员 x 和 y
public:
    Base( float x1, float y1 );     // 基类的构造函数
    ~ Base( void );                 // 基类的析构函数

    void setBase( float x1, float y1 ) { x = x1; y = y1; }
    // 重新设置基类的私有数据成员 x 和 y 值

    float getX( void ){ return x; } // 读取基类的私有数据成员 x 值
    float getY( void ){ return y; } // 读取基类的私有数据成员 y 值
    void print( void )              // 输出显示基类的私有数据成员 x 和 y 值
    { cout<<" 基类对象或派生类对象继承的 x 值 = "<<x<<"\t y 值 = "<<y<<endl; }
};

class Derived : public Base {       // 公有继承方式
    float z;                         // 派生类新增的私有数据成员 z
    Base a;                          // 派生类新增私有数据成员为基类的对象 a
public:
    Derived( float x1, float y1, float x2, float y2, float z1 );
    /* 派生类构造函数的原型声明, 共 5 个形参, 其中 x1 和 y1 用来初始化从基类继承而来的私有数据成员 x 和 y, x2 和 y2 初始化派生类新增数据成员、基类的对象 a, z1 赋值给派生类新增的私有数据成员 z, 详见本构造函数的实现部分 */

    ~ Derived( void );

    void setDerived( float x1, float y1, float x2, float y2, float z1 )
    { Base::setBase(x1, y1); a.setBase(x2, y2); z = z1; }
    // 重新设置派生类对象的数据成员值

    float getZ( void ) { return z; } // 读取派生类新增的私有数据成员 z 值
    void print( void )              // 输出显示派生类对象的数据成员值
    { Base::print( );
      // 调用从基类继承的成员函数 print( ), 输出显示从基类继承的数据成员值
      a.print( );
      /* 通过对象成员 a (其所属类为基类) 调用从基类继承的成员函数 print( ) (基类对象调用基类的成员函数 print( )), 输出显示对象成员 a 的数据成员值 */
    }
```

```

        cout << " 派生类对象的 z 值 = " << z << endl;
        // 输出显示派生类新增的私有数据成员 z 值
    }
};

Base::Base( float x1, float y1 )           // 基类的一般构造函数
{
    // 内部静态变量 i 记录基类构造函数被调用次数
    static int i = 0;
    x = x1; y = y1;                       // 给基类的私有数据成员 x 和 y 赋初值
    cout << "第" << ++i << "次调用基类的构造函数 !\n";
    // 输出显示本构造函数被调用次数的信息
}

Base::~Base( void )
{
    // 内部静态变量 i 记录基类的析构函数被调用次数
    static int i = 0;
    cout << "第" << ++i << "次调用基类的析构函数 !\n";
    // 输出显示本析构函数被调用次数的信息
}

Derived::Derived( float x1, float y1, float x2, float y2, float z1 )
    : Base( x1, y1 ), a( x2, y2 )
/* 派生类构造函数的初始化列表, 既包括对从基类所继承的成员初始化, 又包括成员对象 a 的初始化 */
{
    // 内部静态变量 j 记录派生类构造函数被调用次数
    static int j = 0;
    z = z1;
    cout << "第" << ++j << "次调用派生类的构造函数 !\n";
    // 输出显示派生类构造函数被调用次数的信息
}

Derived::~Derived( void )
{
    // 内部静态变量 j 记录派生类析构函数被调用次数
    static int j = 0;
    cout << "第" << ++j << "次调用派生类的析构函数 !\n";
    // 输出显示派生类的析构函数被调用次数的信息
}

void main( void )
{
    cout << "(1) 对基类 Base 的测试 : \n";           // 输出显示测试内容的提示信息
    Base b(10, 20);
    // 创建基类的一个对象 b, 调用构造函数给其数据成员 x 和 y 赋初值分别为 10 和 20
    b.print( );
}

```

```

/* 通过基类的对象 b 调用基类的成员函数 print( ), 输出显示基类对象的数据成员 x 和 y
   值 */
cout << "(2) 对派生类 Derived 的测试 : \n"; // 输出显示测试内容的提示信息
Derived d(1, 2, 3, 4, 5);
/* 创建派生类的一个对象 d, 并调用派生类的构造函数对其初始化, 先调用基类的构造函数给
   从基类继承而来的数据成员 x 和 y 赋初值分别为 1 和 2, 再调用基类的构造函数给新增对
   象成员 a 的数据成员 x 和 y 赋初值分别为 3 和 4, 最后执行派生类的构造函数体, 给新增
   数据成员 z 赋初值为 5 */
d.setDerived( 11, 12, 6, 7, 15 );
/* 通过派生类对象 d 调用派生类的成员函数 setDerived() 给派生类的所有数据成员重新赋
   值分别为 11、12、6、7 和 15 */
d.print( );
// 通过派生类对象 d 调用成员函数 print(), 输出显示它的所有数据成员值
cout << "(3) 其他测试 : \n"; // 输出显示测试内容的提示信息
float x1 = b.getX( );
/* 通过基类对象 b 调用基类的公有成员函数 getX(), 读取基类对象 b 的私有数据成员 x 值,
   将读取的值保存在自动型变量 x1 中便于输出显示 */
float y1 = b.getY( );
/* 通过基类对象 b 调用基类的公有成员函数 getY(), 读取基类对象 b 的私有数据成员 y 值,
   将读取的值保存在自动型变量 y1 中便于输出显示 */
float z1 = d.getZ( );
/* 通过派生类对象 d 调用派生类新增的公有成员函数 getZ(), 读取派生类对象 d 的私有数据
   成员 z 值, 将读取的值保存在自动型变量 z1 中便于输出显示 */
cout << " 读取基类对象 b 的 x 值 = " << x1 << " , y 值 = " << y1 << endl;
// 输出显示基类对象 b 的 x 和 y 值
cout << " 读取派生类对象 d 的 z 值 = " << z1 << endl;
// 输出显示派生类对象 d 的 z 值
cout << "(4) 现在开始撤销所有对象, 请注意撤销顺序 : \n";
}

```

该程序的输出结果为：

(1) 对基类 Base 的测试：

第 1 次调用基类的构造函数！（创建基类对象 b 时）

基类对象或派生类对象继承的 x 值 = 10 y 值 = 20

(2) 对派生类 Derived 的测试：

第 2 次调用基类的构造函数！（创建派生类对象 d 时，先初始化从基类继承的成员）

第 3 次调用基类的构造函数！（再初始化对象成员 a）

第 1 次调用派生类的构造函数！（最后初始化派生类成员）

基类对象或派生类对象继承的 x 值 = 11 y 值 = 12

基类对象或派生类对象继承的 x 值 = 6 y 值 = 7

派生类对象的 z 值 = 15

(3) 其他测试：

读取基类对象 b 的 x 值 = 10 , y 值 = 20

读取派生类对象 d 的 z 值 = 15

(4) 现在开始撤销所有对象，请注意撤销顺序：

第 1 次调用派生类的析构函数！（撤销派生类对象 d 时，先撤销派生类新增成员）

第 1 次调用基类的析构函数！（再撤销对象成员 a ）

第 2 次调用基类的析构函数！（最后撤销从基类继承的成员）

第 3 次调用基类的析构函数！（撤销基类对象 b ）

4.3.2 派生类构造函数和析构函数的执行次序

从例 4.8 可知：

在基类 Base 中定义了一个带有两个参数的一般构造函数，派生类 Derived 中还包含了一个基类的对象 a 作为私有数据成员，则派生类的构造函数共带有 5 个参数，前两个参数 x1、y1 用于基类数据成员的初始化，x2、y2 用于成员对象 a 的初始化，z1 用于初始化派生类新增的私有数据成员 z。

当创建派生类的对象 d 时，首先调用基类的构造函数 Base() 初始化派生类从 Base 基类继承而来的数据成员 x 和 y，然后调用对象成员 a 所属类（本例中它就是基类 Base）的构造函数对其进行初始化，最后才执行派生类的构造函数。因此，一言以蔽之，创建一个派生类对象调用构造函数的执行次序是：基类第一，内部的成员对象第二，派生类最后。这正符合人们的礼仪原则：“先兄长（基类），再客人（成员对象），后自己（派生类）”的顺序。如例 4.8 中：

```
void    main( void )
{
    Base    b(10,20);

    ...

    b.print( );

    Derived    d(1, 2, 3, 4, 5);
    /* 创建派生类对象调用构造函数的执行次序是：基类 Base 第一，内部的成员对象所属类即
       Base 类第二，派生类 Derived 最后 */

    ...

    d.print( );
}
```

在 main() 函数中创建派生类对象 d 时，首先调用基类的构造函数，初始化从基类继承的数据成员 x 和 y，然后为对象 b 调用基类的构造函数，初始化新增的对象成员 a 的数据成员，最后才初始化新增的数据成员 z，这从程序的输出结果中可清楚地看出。

从程序的输出结果可知，调用析构函数的操作顺序与构造函数相反。当派生类的一个对象离开作用域，自动地调用析构函数时，其执行顺序是派生类第一，内部成员对象第二，基类最后。源程序中不允许显式地调用析构函数。

4.4 基类和派生类的赋值规则

4.4.1 赋值兼容性规则

在公有继承方式下，可以把一个派生类对象赋值给一个基类对象。在此赋值中，只有

基类中的成员被复制。这与客观实际情况是完全吻合的，正如图 1.6 所示，以“人”为根结点的类层次结构图中，“研究生”类是“学生”类的派生类，那么，“研究生”类的一个具体实例（即对象）“王平”比“学生”类的一个具体实例要添加“专业”和“入学年月日”两个信息，换句话说，不是每个学生都具有研究生资格的，但研究生肯定是学生，只有具有本科学历并通过入学考试被某高等学校录入某专业的学生才具有研究生资格，所以把某个研究生当成学生是可以的，即可以把他的信息复制成“学生”类的一个具体对象，因为他具有“学生”类的具体对象所具备的“学校名称”和“学号”信息，但反过来，将一个“学生”类的具体对象复制成一个“研究生”类的对象是肯定行不通的。又如，Circle 类从 Point 类公有继承而来，则有：

```
Point p;      Circle c;
```

由于基类 \subset 派生类，即基类所包含的信息都蕴涵在派生类中，显然派生类所包含的信息比基类要多。因此，可以写做：

```
p = c;
```

一般格式为：

基类对象 = 派生类对象;

c 到 p 的赋值实际上是进行如下操作：

```
p.x = c.x;    p.y = c.y;
```

其中，只有 x 和 y 被复制，即复制了派生类对象 c 的圆心坐标值 x 和 y，而 radius 未被复制，因为它不在基类 Point 中。但是反过来因基类包含的信息要少，所以不能把基类 Point 的对象 p 赋值给派生类对象 c。例如：

```
c = p;      // 出错，基类对象的信息不够          赋值
```

因此，基类和派生类的赋值是单方向的，即“派生类对象 \rightarrow 基类对象”。

这种赋值兼容规则可扩展到对象指针和对象引用。在公有继承方式下，指向派生类对象的指针可以自动地转换为指向基类对象的指针。例如可以将 Circle 类的对象指针赋值给 Point 类的对象指针：

```
Point    p, * bp = &p;
Circle   c, * dp = &c;
```

则可以写成：

```
bp = dp;
// 隐式（自动）类型转换，指向派生类对象指针 dp 自动转换成基类的对象指针再赋给 bp
```

一般格式为：

基类的对象指针 = 派生类对象指针;

也可把 Circle 类的对象引用赋值给 Point 类的对象引用, 如:

```
Point p;  
point & p_ref = p;  
Circle c;  
Circle & c_ref = c;
```

则可以写成:

```
p_ref = c_ref;
```

一般格式为:

基类的对象引用 = 派生类的对象引用;

既然派生类的对象引用可以赋值给基类的对象引用, 而派生类的对象是可以用来初始化派生类的对象引用, 即:

派生类名 & 派生类的对象引用名 = 派生类的对象名;

那么, 派生类的对象也就可以用来初始化基类的对象引用, 即:

基类名 & 基类的对象引用名 = 派生类的对象名;

但反之不能进行, 即不能把基类 Point 的对象、对象指针和对象引用赋值给派生类 Circle 的对象、对象指针和对象引用, 例如:

```
c = p;           // 出错, 基类 Point 的对象 p 不能赋值给派生类 Circle 的对象 c  
dp = bp;         // 出错, 基类 Point 的对象指针 bp 不能赋值给派生类 Circle 对象指针 dp  
c_ref = p_ref;   // 出错, 基类 Point 的对象引用 p_ref 不能赋值给派生类 Circle 对象引用 c_ref
```

但 C++ 容许把基类的对象指针强制转换成派生类的对象指针, 换句话说, 基类的对象指针

转换成派生类的对象指针必须显式 (用强制类型转换表达式) 进行, 例如:

```
Point p, * bp = &p;  
Circle c, * dp = &c;  
dp = ( Circle * )bp; // 显式类型转换
```

显然, 基类对象指针可强制转换成它公有派生类的对象指针, 使得基类对象指针可用来指向它公有派生类的任何对象, 甚至显式地转换成派生类指针用来替代这些派生类对象

去访问它们的新增成员，这是公有继承方式下，基类对象指针很重要的性质。

综上所述，基类和派生类的赋值兼容规则可归纳为如下三点：

公有派生类的对象可以赋值给基类的对象，即将公有派生类对象中从基类继承而来的数据成员逐个赋值给基类对象的对应数据成员；

公有派生类对象的地址可以赋值给基类的对象指针；

公有派生类对象可以用来初始化基类的对象引用。

在私有继承方式下，由于基类公有成员在派生类中变成为私有的，所以基类的公有成员只能通过基类的对象指针访问，而不能通过派生类的对象指针访问，因此派生类的对象指针和基类的对象指针之间的所有转换都不能进行。

4.4.2 基类和派生类的对象指针

指向基类的指针和指向派生类的指针是密切相关的，其表现如下。

一个指向基类的指针可用来指向该基类公有派生类的任何对象，这是 C++ 实现程序运行时多态性的关键途径。

例 4.9 使用基类对象指针和派生类对象指针的例程。

```
#include    < iostream >           // 使用 C++ 新标准的流库
using      namespace  std;         // 将 std 标准名空间合并到当前名空间

class  Point  {
public :
    int  x , y;                    // 基类 Point 的公有数据成员 x 和 y
    Point( int xi, int yi ) { x = xi; y = yi; } // 一般构造函数
};

class  Circle : public Point {      // 基类 Point 公有派生出 Circle 类
public :
    int  radius;                  // 派生类 Circle 新增的公有数据成员 radius
    Circle( int x1, int y1, int r ) : Point( x1, y1 ) { radius = r; }
    /* 派生类 Circle 的构造函数，其构造函数头包含对从基类继承的数据成员 x 和 y( 圆心坐标 )
       进行初始化的列表 */
};

void  main( void )
{
    Point  pnt(160, 180), * bp;
    // 定义基类 Point 的对象 pnt 和 Point 类的对象指针 bp，但 bp 尚未定向
    Circle  spotlight(320, 240, 40), * dp;
    /* 定义公有派生类 Circle 的对象 spotlight 和 Circle 类的对象指针 dp，但 dp 尚未定向 */
    bp = & pnt;
```

```

// 令基类的对象指针 bp 指向基类对象 pnt 后, 可用 bp 代替 pnt 访问其成员
cout << "(1) pnt's X = " << bp -> x << "\t Y = " << bp -> y << endl;
// 用 bp 代替 pnt 访问对象 pnt 的公有数据成员 x 和 y

bp = & spotlight;
/* 该赋值语句使用了赋值规则, 左值为基类指针 bp, 右值是派生类对象的地址, 即对象指针
   &spotlight, 从而令基类的对象指针 bp 指向派生类对象 spotlight 后, 可以用基类指
   针 bp 代替 spotlight 去访问派生类对象 spotlight 中从基类继承的成员, 但不能访问
   其新增数据成员 radius */

cout << "(2) spotlight's X = " << bp -> x << "\t Y = " << bp -> y ;
/* 因基类的对象指针 bp 指向派生类对象 spotlight, 用基类指针 bp 代替 spotlight 去读
   取派生类对象 spotlight 中从基类继承的数据成员 x 和 y 值, 但不能读取其新增数据成员
   radius 的值 */

dp = & spotlight;
// 令派生类的对象指针 dp 指向派生类对象 spotlight

cout << "\t radius = " << dp -> radius << endl;
// 用派生类的对象指针 dp 代替派生类对象 spotlight 去访问其新增数据成员

cout << "(3) spotlight's X = " << bp -> x << "\t Y = " << bp -> y ;
cout << "\t radius = " << ((Circle *)bp) -> radius << endl;
// 把基类的对象指针 bp 强制转换成派生类的对象指针后才能访问其新增成员 radius

}

```

该程序的输出结果为：

```

(1) pnt's X = 160          Y = 180
(2) spotlight's X = 320   Y = 240      radius = 40
(3) spotlight's X = 320   Y = 240      radius = 40

```

在例 4.9 中, 由于 Circle 类从 Point 类公有继承而来, main() 函数内定义的基类指针 bp 不仅可以指向基类 point 的对象 pnt, 还可用地址赋值语句令它指向公有派生类 Circle 的对象 spotlight, 因此, 用它同样可以代替派生类对象名 spotlight 去访问公有派生类中从基类 Point 继承的公有成员 (包括数据成员和成员函数)。可是它不能访问派生类中的新增成员, 如例中的新增数据成员 radius。但是反过来却不行, 即不能用指向派生类的指针指向基类的对象。顺便指出, 赋值语句 “bp = & spotlight;” 实际上是应用了上述的 “派生类的对象指针可以赋值给基类的对象指针” 这条规则, 因为右值表达式 “& spotlight” 是对派生类的对象 spotlight 取地址, 它就可以看成为是派生类的对象指针, 所以, 这条规则也可以叙述成 “公有派生类对象的地址可以赋值给基类的对象指针”。

如果希望用基类指针访问其公有派生类的新增成员, 必须将基类指针用强制类型转换成派生类指针。格式为：

((派生类名*)基类指针名) -> 新增成员

例 4.9 中：

```
cout << "\t radius = " << ((Circle *)bp) -> radius << endl;
```

在公有继承方式下，基类指针是联系基类和派生类对象间消息传递的桥梁。例 4.9 中是采用地址赋值语句使基类的对象指针 bp 指向公有派生类对象 spotlight。但更多的情况是在调用函数时，用实参传递给形参来实现这种地址赋值。因此任何使用基类指针作为形参的函数，都同样可以用来处理派生类的对象，只需在调用该函数的语句中，以“&公有派生类对象名”作为实参即可，格式为：

函数原型：<返回类型> 函数名(基类名 * 指针名, ...);

调用语句： 函数名(& 公有派生类对象名 , ...);

那么，在该函数调用时、实参传递给形参的过程中，相当于执行了如下的初始化操作语句：

```
基类名 * 基类的对象指针名 = & 公有派生类对象名;
```

即令基类的对象指针指向公有派生类对象，则在函数体内凡是对基类对象的数据成员进行操作，即是用来对公有派生类中从基类继承的数据成员进行操作。

由于引用也是通过地址传递的，且采用对象引用传递函数的参数还可以避免取地址运算和取内容运算的麻烦。因此，凡是使用对基类的对象引用作为形参的函数，同样可以用来处理公有派生类的对象，其格式为：

函数原型： <返回类型> 函数名(基类名 & 引用名, ...); 调用语句： 函数名(公有派生类对象名, ...);
--

因此，在调用该函数时、实参传递给形参的过程中，相当于执行了如下形参引用的初始化操作语句：

基类名 & 基类的对象引用名 = 公有派生类对象名；

如前所述，公有派生类对象就成了形参引用的被引用对象，它们通过地址相联系，使得形参引用就是公有派生类对象（实参）的一个替换名，它们所指的实体都是被引用对象即作为实参的公有派生类对象，则在该函数体内凡是对形参引用的操作，实际上是对公有派生类对象进行操作。所以，该函数不仅可用来处理基类的对象，还可以处理从该基类派生出的子（该基类的直接派生类）孙（该基类的间接派生类）的公有派生类对象，使得该函数的使用范围大幅度扩大。众所周知，函数调用除了以上所述的传地址方式以外，还有传值方式调用，即函数的形参是基类的对象，而实参采用该基类的公有派生类对象，即：

函数原型： <返回类型> 函数名(基类名 对象名, ...); 调用语句： 函数名(公有派生类对象名, ...);

因此，在调用该函数时、实参传递给形参的过程中，相当于执行了如下形参对象的初始化操作语句：

基类名 基类的对象名 = 公有派生类对象名；

从而将公有派生类（实参）对象中从基类继承而来的数据成员逐个复制到（基类）形参对象的对应数据成员中，随后，进入到函数体内对形参（基类）对象的操作处理只改变它本身的数据成员值，而不会影响实参（公有派生类的）对象的数据成员值，即传值方式的数据传递是“单方向”的。

今后，编程者应记住这样一条编程规则：对于采用公有继承方式的类层次结构，若需要编写一个函数处理其中某一层对象，应将该函数的形参指定为该层次结构顶层基类的对象，或对象指针或对象引用，由上述“基类和派生类的赋值兼容规则”可知，该函数可用来处理这种类层次结构中任何一层的对象。

综上所述，在公有继承方式下，在一个类层次结构中，当形参作为该类层次结构顶层基类的对象时，实参则可以是该类层次结构中任何一层的对象，将“基类和派生类的赋值兼容规则”应用于函数调用时用实参初始化形参，具有如下三种情况：

用任何一层的公有派生类对象作为实参，对作为形参的顶层基类对象进行初始化；
用任何一层的公有派生类对象指针作为实参，对作为形参的顶层基类对象指针进行

初始化定向操作；

用任何一层的公有派生类的对象作为实参,对作为形参的顶层基类对象引用进行初始化。

4.4.3 子类型和类型适应

若类 B 是类 A 的公有派生类,则称类 B 是类 A 的一个子类型,当然类 A 还可以有其他的子类型。根据上述的赋值规则,凡是基类 A 的对象可使用的任何地方,都可以用公有派生类 B 的对象代替之,这正是文献[3](p227)所称的 B 类型适应 A 类型。所谓“类型适应”是指两种类型之间的关系,即类 A 中的操作可以用于操作类 B 的对象。例如,考察确定两点之间距离的函数 distance(), Circle 类是 Point 类的公有派生类, Circle 类就是 Point 类的子类型, Point 类还有一个子类型 Rectangle 类,则 distance() 函数不仅可以用于 Point 类的两个(点)对象,还可以用来计算(圆)对象的圆心间距离以及两个长方形基准点(长方形左三角顶点)间的距离。

例 4.10 若 Point 类有两个子类型 Circle 类和 Rectangle 类。计算两点之间距离的函数 distance() 是对 Point 类的两个(点)对象的操作,可以用于它的子类型 Circle 类的两个(圆)对象和 Rectangle 类的两个(长方形)对象。

```
#include <iostream>           // 使用 C++ 新标准的流库
#include <cmath>               // C 语言数学标准函数的原型声明在其中
using namespace std;          // 将 std 标准名空间合并到当前名空间
class Point {                 // Point 类作为基类
    int x, y;                 // 基类的私有数据成员 x 和 y
public:
    Point( int xi, int yi ) { x = xi; y = yi; } // 一般构造函数
    int readX( void ) { return x; }             // 读取 x 坐标值
    int readY( void ) { return y; }             // 读取 y 坐标值
    friend double distance(Point & a, Point & b); // 计算两点距离的友元函数
};

class Circle : public Point { // Circle 类是 Point 类的子类型
    int radius;               // 子类型 Circle 类的私有数据成员
public:
    Circle( int x1, int y1, int r ) : Point( x1, y1 ) { radius = r; }
    // 子类型 Circle 的构造函数
    int readR( void ) { return radius; }         // 读取圆的半径值
    friend double distance( Point & a, Point & b );
    // 把该函数声明为子类型 Circle 的友元函数
};
```

```

class Rectangle : public Point {                                     // Point 类还有一个子类型
    int width, height;
    // 子类型 Rectangle 类的新增数据成员 width (长方形的宽度) 和 height (高度)

public :
    Rectangle( int x0, int y0, int w, int h ) : Point( x0, y0 )
    {   width = w; height = h;   }                                     // 子类型 Rectangle 类的构造函数
    int readW( void ) { return width; }   // 读取长方形宽度值
    int readH( void ) { return height; }   // 读取长方形高度值
    friend double distance( Point & a, Point & b );
    // 把该函数声明为子类型 Rectangle 的友元函数

};

double distance( Point & a, Point & b )
{   double dx = a.x - b.x;           // 计算两点在 x 方向的差值
    double dy = a.y - b.y;           // 计算两点在 y 方向的差值

    return sqrt(dx * dx + dy * dy);
    /* 计算两点的距离作为该函数的返回值, sqrt( ) 是 C 语言运行库中的标准函数, 其原型声明
       明在新标准头文件 cmath 中 */
}

void main( void )
{   Circle k( 28, 36, 80 ), r( 72, 147, 160 );
    // 创建 Point 类的子类型 Circle 类的两个对象 k 和 r

    cout << " K's X = " << k.readX() << " , K's Y = " << k.readY() << endl;
    // 输出显示圆对象 k 圆心的坐标值

    cout << " R's X = " << r.readX() << " , R's Y = " << r.readY() << endl;
    // 输出显示圆对象 r 圆心的坐标值

    double dc = distance(k , r);
    /* 调用友元函数 distance( ) , 计算圆 k 和 r 的圆心距离将结果保存在自动变量 dc 中。在
       调用 distance( ) 函数, 将实参赋给形参的过程中, 相当于执行了 “ Point &a = k; ”
       和 “ Point & b = r; ” 两条形参引用初始化操作语句, 使形参引用 a 和 b 分别是实参 k
       和 r 的替换名, 对 a 的操作就是对 k 的操作, 对 b 的操作也就是对 r 的操作 */

    cout << " The distance of K and R center " << dc << endl;
    // 输出显示圆 k 和圆 r 的圆心间距离值

    Rectangle rec1(12, 22, 60, 88), rec2(42, 124, 80, 128);
    // 创建 Point 类的另一个子类型 Rectangle 类的两个对象 rec1 和 rec2

    double dr = distance(rec1 , rec2);
    /* 调用友元函数 distance( ) , 计算长方形 rec1 和 rec2 的基准点间距离, 并将结果保存在
       自动变量 dc 中。在调用 distance( ) 函数, 将实参赋给形参的过程中, 相当于执行了
       “ Point &a = rec1; ” 和 “ Point & b = rec2; ” 两条形参引用初始化操作语句, 使

```

形参引用 a 和 b 分别是实参 rec1 和 rec2 的替换名, 对 a 的操作就是对 rec1 的操作, 对 b 的操作也就是对 rec2 的操作 */

```
cout << " The distance of REC1 and REC2 datum mark " << dr << endl;
// 输出显示长方形 rec1 和 rec2 的基准点间的距离值
```

```
}
```

该程序的输出结果为：

```
K's X = 28 , K's Y = 36
R's X = 72 , R's Y = 147
The distance of K and R center 119.403
The distance of REC1 and REC2 datum mark 106.32
```

distance()函数的两个形参是基类 Point 的对象引用, 当公有派生类的对象 k 和 r 作为实参分别初始化形参 a 和 b 后, 其函数体内凡是对基类 Point 的对象引用 a 和 b 所进行的各种操作, 实际上都是用来对公有派生类的对象 k 和 r 进行操作, 由于“类型适应”而不会发生任何问题。应该注意的是当 Circle 类的 k 和 r 对象作为实参赋值给形参 a 和 b 时, 如前所述, 只有从基类继承的数据成员 x 和 y 被复制, 因此实际上是把每个对象的圆心点坐标值传递给 distance()函数, 然后由该函数计算出两圆心的距离。当然, 用基类 Point 的两个对象作为该函数的实参, 则用来计算基类两个点对象的距离, 而更重要的是用它公有派生类 Circle 的两个对象 k 和 r 作为实参代替之, 则是用来处理公有派生类的两个圆对象 k、r 圆心间的距离, Point 类和 Circle 类间的关系为“类型适应”, 使得 distance()函数不仅可以计算基类 Point 的两个点对象的距离, 还可以用来处理由它继承而来的子子(直接派生类)子孙(间接派生类)的公有派生类的所有对象。例如, Rectangle 类也是 Point 类的子类型, 再由 Rectangle 类公有派生出 Window 类, 即:

```
class Window : public Rectangle {
    int backgroundColor;          // 新增私有数据成员(背景颜色)
public :
    ...
    friend double distance(Point & a, Point & b);
};
```

则 distance()函数同样可以用来计算两个窗口对象的基准点间距离值。因此, 编程者应记住: 对于采用公有继承方式的类层次结构, 若需要编写一个函数处理其中某一层对象, 应将该函数的形参指定为该层次结构顶层基类的对象, 或对象指针, 或对象引用, 利用“类型适应”能将该函数的使用范围扩大到整个类层次结构的任何一层。

4.4.4 不能继承的部分

在继承传递过程中, 如下成分将不能被继承。

构造函数和析构函数。在创建派生类对象时,自动调用基类的构造函数,但它们只能在建立派生类对象时由编译系统自动调用,不能像其他被继承的成员函数那样由派生类显式调用,写在程序上的只能有初始化列表。当对象离开作用域时由编译系统自动调用析构函数,派生类也不允许显式地调用基类的析构函数。例如:

```
B(int v, int t) : A(v)
{
    A::A(v);
    /* 出错,基类构造函数不能像其他被继承的成员函数那样在派生类成员函数体内显式调用,只能放在初始化列表中 */
    total = t;
}
```

友元关系也是不可继承的。这与现实生活也是类似的,父母的朋友不一定是子女的朋友,继承树中类的友元联系并不向上或向下传递。

4.5 多 继 承

4.5.1 多继承派生类

可同时具有多个基类的派生类称为“多继承派生类”,它同时继承了这些基类的所有成员。因此每个派生类的对象都包含了所有基类的成员,如前所述,多继承派生类可以组成有向图层次结构。

定义一个多继承派生类与定义单继承派生类类似,不仅要指明该多继承派生类从哪几个基类继承而来,而且还要指明每个基类的继承方式,其格式如下:

```
class 派生类名 : <继承方式 1> 基类 1, <继承方式 2> 基类 2 , ...
{
    <函数体>
}
```

例 4.11 用多继承派生类描述一个白色圆桌。

```
#include <iostream> // 使用 C++新标准的流库
using namespace std; // 将 std 标准名空间合并到当前名空间
#define WHITE 15 // 白色,符号常量值为 15
const double PI = 3.14159;
// 圆周率 $\pi$ ,在 C++中不采用#define 预处理语句定义符号常量,而用 const
class Circle {
    double radius; // Circle 类的私有数据成员 radius
public :
    Circle( double r ) { radius = r; }
    // 构造函数给 radius 赋初值
    double area( void ) { return PI * radius * radius; }
```

```

        // 计算圆面积的成员函数 area( )
    };
    class Table {
        double height;          // Table 类的私有数据成员 height
    public :
        Table( double h ) { height = h; }          // 构造函数给 height 赋初值
        double getHeight( void ) { return height; } // 读取 height 的值
    };
    /* RoundTable 类既继承了 Circle 类的所有数据成员和成员函数，又继承了 Table 类的所有数
       据成员和成员函数 */
    class RoundTable : public Table , public Circle {
        int color;
    public :
        RoundTable( double h, double r, int c );
        int getColor( void ) { return color; }
    };
    RoundTable::RoundTable( double h, double r, int c ) : Table(h), Circle(r)
    {   color = c;   }
    void main( void )
    {   RoundTable table( 160.0, 180.0, WHITE );
        cout << "The table properties are : \n";
        // 输出显示“该桌子的属性特征:”提示信息
        cout << "\t Height = " << table.getHeight( ) << endl;
        // 输出显示“桌面高度 = ?”信息
        cout << "\t Area = " << table.area( ) << endl;
        // 输出显示“桌面面积 = ?”信息
        cout << "\t Color = " << table.getColor( ) << endl;
        // 输出显示“桌面颜色 = ?”信息
    }
}

```

该程序的输出结果为：

```

The table properties are :
    Height = 160
    Area = 101788
    Color = 15

```

如例 4.11 中，对一组指明的基类，每个基类都有各自的访问限制符，这些基类都是直接基类。例如要定义多继承派生类 RoundTable（白色圆桌），应具有桌子的特征和圆的几何特征，可用图 4.11 的层次结构图表示。

Table 类和 Circle 类是 RoundTable 类的基类，但 Table 类和 Circle 类的任一

基类都将是 RoundTable 类的间接基类。如前所述,Point 类(为了简单起见,在例 4.10 中未写出)是 Circle 类的直接基类,则又是 RoundTable 类的间接基类。如图 4.12 所示,用多继承派生建立的 RoundTable 类继承了所有上层基类的成员。

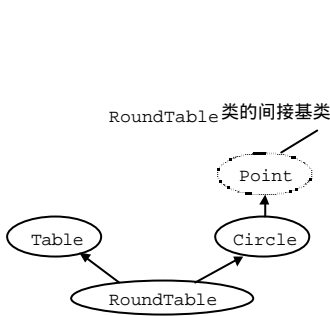


图 4.11 RoundTable 的类层次结构

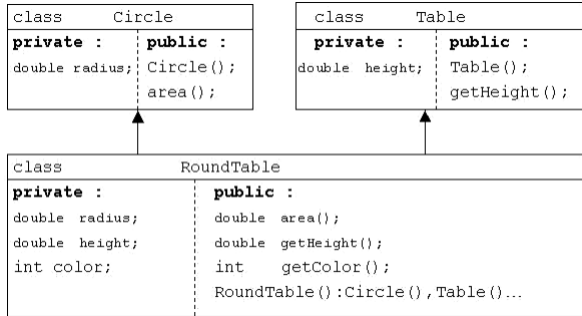


图 4.12 RoundTable 类继承了上层基类的成员

4.5.2 多继承派生类的构造函数

多继承派生类构造函数的定义格式为：

```
多继承派生类构造函数名( 总参数表 ) : 基类名 1(参数表 1), 基类名 2(参数表 2), ...  
                                          成员对象名(参数表 n + 1), ...  
{  
    <多继承派生类构造函数体>  
}
```

多继承派生类构造函数的总参数表所提供的参数应包含其后的各个分参数表。例如 RoundTable 类的构造函数参数表既包含了从 Circle 类继承的数据成员 radius(半径)赋初值所需要的参数 r, 也包含从 Table 类继承的数据成员 height(高度)所需的参数 h, 还有新增数据成员 color 所需的参数 c。顺便指出, 与单继承一样, 派生类新增数据成员的初始化可以在其构造函数的初始化列表中进行, 也可以在其构造函数体内用赋值语句来完成。

多继承派生类的构造函数必须承担调用该派生类所有直接基类的构造函数, 以完成这些基类的初始化任务, 同时派生类构造函数的参数个数必须包含完成所有基类初始化任务所需的参数个数。

多继承派生类构造函数的执行顺序是先调用所有基类的构造函数, 如例 4.11 中的 Table(h)和 Circle(r), 再执行派生类本身的构造函数。处于相同层次各基类构造函数的调用顺序取决于多继承派生类定义时, 在类头冒号后面指定各继承基类的先后顺序, 如例 4.11 的 RoundTable 类头可知, Table 类在前, Circle 类在后, 而与多继承派生类构造函数定义时函数头所列的成员初始化表的排列次序无关。但是, 编程者在编写实用化程序时, 应该使派生类构造函数定义时函数头所列的成员初始化表的排列次序与类头冒

号后面指定各继承基类的先后顺序一致，以避免产生额外的运行开销（详见文献[26]P58）。

4.5.3 虚基类

虚基类仅用于多继承，因为多继承建立的类层次结构中很可能有复杂的关系，有时编程者必须对基类继承的某个方向进行一定程度的控制。因为一个类不能多次作为某个派生类的直接基类，但可以多次作为一个派生类的间接基类。例如：

```
class B { ... };
class D : public B, public B { ... };
//出错，类B不能两次作为类D的直接基类
```

例 4.12 虚基类的使用。

```
#include <iostream> // 使用 C++ 新标准的流库
using namespace std; // 将 std 标准名空间合并到当前名空间

class A {
public:
    int value; // 基类的公有数据成员 value
    A(int v) { value = v; } // 基类的构造函数
};

class B : virtual public A {
// 在继承方式前加关键字 virtual 把 A 类指定为虚基类
public:
    int bValue; // 公有派生类 B 的新增公有数据成员 bValue
    B(int v, int b) : A(v) { bValue = b; } // 公有派生类 B 的构造函数
    int getbValue( void ) { return bValue; } // 读取新增公有数据成员 bValue 值
};

class C : virtual public A {
// 在另一条继承路径上也把 A 类指定为虚基类
public:
    int cValue; // 公有派生类 C 的新增公有数据成员 cValue
    C(int v, int c) : A(v) { cValue = c; } // 公有派生类 C 的构造函数
    int getcValue( void ) { return cValue; } // 读取新增公有数据成员 cValue 值
};

class D : public B, public C { // 最底层类 D 分别由 B 和 C 类公有继承而来
public:
    int dValue; // 最底层类 D 的新增公有数据成员 dValue
    int getValue( void ) { return value; }
    // 读取从虚基类 A 继承的公有数据成员 value 值
```

```
D(int v1, int v2, int b, int c, int d) : B(v1, b), C(v2, c), A(v1)
{ dValue = d; }
// 最底层类 D 的构造函数头上的初始化列表中, 还必须列出对虚基类构造函数 A(v1) 的调用
int getdValue( void ) { return dValue; }
// 读取最底层类 D 的新增公有数据成员 dValue 值
};

void main( void )
{
    D    obj(1, 2, 3, 4, 5);        // 创建最底层类 D 的一个对象 obj
    cout << " 最底层类 D 的对象 obj 的 value 值 = " << obj.getValue( ) << endl;
    cout << " 最底层类 D 的对象 obj 的 bValue 值 = " << obj.getbValue( ) << endl;
    cout << " 最底层类 D 的对象 obj 的 cValue 值 = " << obj.getcValue( ) << endl;
    cout << " 最底层类 D 的对象 obj 的 dValue 值 = " << obj.getdValue( ) << endl;
}
```

该程序的输出结果为：

```
最底层类 D 的对象 obj 的 value 值 = 1
最底层类 D 的对象 obj 的 bValue 值 = 3
最底层类 D 的对象 obj 的 cValue 值 = 4
最底层类 D 的对象 obj 的 dValue 值 = 5
```


1. 多继承中的二义性问题

如前所述，在 OOP 中，多继承关系的描述却是一个十分复杂的问题，其复杂性就是会带来模棱两可的“二义性”问题。在 C++ 中，有如下两种“二义性”问题：

若派生类从几个基类继承了同名的数据成员，或原型相同的成员函数时，则对它们的任何访问操作都会造成模棱两可的“二义性”问题。例如：

```
class   Base1   {
public :
void   draw( void );

    ...

};

class   Base2   {
public :
void   draw( void );

    ...

};

class   Derived : public Base1, public Base2 {
    ...           // 没有声明 draw( )

};

void   main( void )
{
    Derived   d;
    d.draw( );           // 出错，是调用 Base1::draw( ) 还是调用 Base2::draw( )

    ...

}
```

由于派生类 Derived 继承了两个实现版本（即具有不同的函数体）的同名成员函数 draw()，当派生类 Derived 的对象 d 访问 draw() 时，将产生模棱两可的“二义性”问题。有如下两种办法解决它：

用前缀“基类名::”写在同名成员的前面加以区别，称为“成员名限定法”即：

```
void   main( void )
{
    Derived   d;
    d.Base1::draw( );           // OK!调用 Base1::draw( )
    d.Base2::draw( );           // OK!调用 Base2::draw( )

    ...

}
```

其一般格式为：

对象名.基类::公有数据成员
对象名.基类::公有成员函数(< 参数表 >)

在派生类 `Derived` 作用域内, 重新定义一个同名的成员函数 `draw()`, 在其函数体内指明是调用 `Base1::draw()` 还是 `Base2::draw()`, 即:

```

...
class Derived : public Base1, public Base2 {
public :
    // 成员函数体, 内层作用域
    void draw( void ) { Base1::draw( ); }
    /* 重新定义一个同名的成员函数 draw( ),
       在其函数体内指明调用 Base1::draw( ) */
    ...
};
void main( void )
{
    Derived obj( 6 );
    obj.draw();
    /* 派生类对象 obj 调用自身成员函数 draw(), 执行函数体内的 “ Base1::draw( ); ” 语
       句 */
    ...
}

```

// 类体, 外层作用域

正如 3.8 节所述, 在一个嵌套的作用域范围内, 外层作用域和内层作用域如果具有同名的标识符时, 在内层作用域的标识符将掩盖外层的同名标识符。由于派生类 `Derived` 继承了 `Base1::draw()` 和 `Base2::draw()` 两个实现版本, 造成模棱两可的“二义性”问题。但是, 成员函数名是在外层作用域, 而 `Derived::draw()` 成员函数体却是内层作用域, 其成员函数名则掩盖外层的同名标识符。因此, 执行主函数体内的 “`obj.draw();`” 语句, 理所当然地调用的是 `Base1::draw()`。

当一个派生类从几个基类继承而来,但这几个基类又是同一个基类的直接派生类。参看例 4.12, 由于类 A 是类 D 的间接基类, 但是类 A 通过两个继承路径到最底层的派生类 D, 最底层类 D 从图 4.13 左边继承路径要保存一份顶层基类 A 的

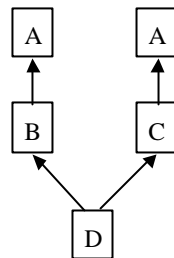


图 4.13 两条继承路径的类层次结构

数据成员 value 值, 即从基类 A 继承的部分, 把它称为基类 A 的一个子对象, 而从右边继承路径又要保存一份顶层基类 A 的数据成员 value, 也是从基类 A 继承的部分, 把它称为基类 A 的另一个子对象。因此派生类 D 的每个对象同时保存了两份 A 类数据成员 value 的值, 即产生了两个基类 A 的子对象, 从而产生了二义性。为了便于分析, 这是假定在没有采用虚基类机制时, 将例 4.12 简写成如下格式, 并画出类层次结构图, 如图 4.13 所示。

```
class A {
public:
    int value;
    ...
};
class B : public A { ... };
class C : public A { ... };
class D : public B, public C {
public:
    int getValue( void ) { return value; }
    ...
};
```

此时, 编译系统给派生类 D 的一个对象 obj 分配的内存空间布局如图 4.14(a) 所示, 它由一片连续的内存单元构成, 包含有从直接派生类 B 继承的基类 A 子对象(基类 A 的部分)和从直接派生类 C 继承的基类 A 子对象(基类 A 的部分)。这就相当于实际生活中有两个祖父, 他们不仅是孪生兄弟, 而且名字完全相同, 他们的孙子辈无法区分。如何解决这种二义性问题呢?

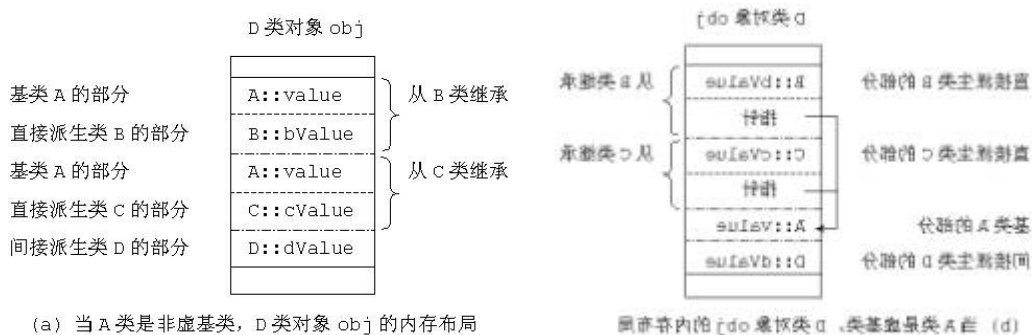


图 4.14 基类 A 为非虚基类和虚基类时 D 类对象的内存布局

2. 使用虚基类

将基类定义为虚基类，就可以确保该基类通过多条路径继承时，派生类仅仅只继承该基类一次，即本例中派生类 D 的对象 obj 只保存一份该基类 A 的数据成员 value 值，而不会产生二义性。定义虚基类的格式为：

```
class 派生类名 : virtual <继承方式> 基类
{
    <函数体>
}
```

其中 virtual 是指明虚基类的关键字，例 4.12 采用虚基类机制后，可简写为：

```
class A {
public :
    int value;
    ...
};
class B : virtual public A { ... };
class C : virtual public A { ... };
class D : public B, public C {
public :
    int getValue( ) { return value; }
    ...
};
```

其类层次结构图如图 4.15 所示，由此可看出，将基类 A 定义成虚基类后，派生类 D 的对象 obj 只保存一份虚基类 A 的数据成员 value 值，即只产生了一个虚基类 A 子对象（基类 A 的部分），此时给派生类 D 的对象 obj 分配的内存空间布局如图 4.14 (b) 所示，从直接派生类 B 继承的部分和从直接派生类 C 继承的部分都分别包含一个对象指针，指向虚基类 A 的数据成员 value 所存放的内存空间，通常不管这个类层次结构包含有多少层，最底层派生类对象创建时，由该派生类的构

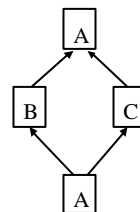


图 4.15 使用虚基类 A 的类层次结构

构造函数通过调用虚基类的构造函数对惟一的虚基类子对象（即虚基类部分，本例为虚基类 A 的数据成员 value）进行初始化，所以，最底层派生类构造函数的初始化列表中必须包含虚基类构造函数的调用。因此，在主函数体内执行“Derived obj(6);”语句，创建最底层派生类 Derived 的对象 obj 时，虚基类的构造函数只被调用一次，从而解决了多继承中的这种二义性问题。顺便指出，在定义虚基类时，关键字 virtual 和 public 的顺序可以互换而不产生编译错误。同时，为了确保虚基类在派生类中只继承一次，就必须在所有直接派生类中将它指定为虚基类，如例 4.12 中，B 类和 C 类都是 A 类的直接派生类，则在 B 类的类头部分要指定 A 类是虚基类，对 C 类定义时也要指定为虚基类，否则仍然会因多次继承而导致二义性。

3. 虚基类的构造函数

虚基类仅用于多继承，且它的构造函数必须只被调用一次。编写时注意如下几点：

若一个派生类有一个直接或间接的虚基类，那么由该虚基类直接或间接继承的派生类，在这些派生类构造函数的初始化列表中都应列出对该虚基类构造函数的调用，如例 4.12 中：

```
class B : virtual public A {
public :
    B( int v, int b ) : A( v ) { bValue = b; }
    // 初始化列表含有对该虚基类 A 构造函数的调用
    ...
};

class C : virtual public A {
public :
    C( int v, int c ) : A( v ) { cValue = c; }
    // 初始化列表含有对该虚基类 A 构造函数的调用
    ...
};
```

此外，在最底层派生类（如例 4.12 中的 D 类）构造函数的初始化列表中还必须列出对虚基类构造函数（如 A(v1)）的调用，如果没有列出则编译系统将调用该虚基类的缺省构造函数。如例 4.12 中：

```
class D : public B, public C {
public :
    D(int v1,int v2, int b, int c, int d) : B(v1, b), C(v2, c), A(v1)
    { dValue = d; }
    ...
};
```

若虚基类和非虚基类构造函数的调用同时出现在一个成员初始化列表中，先执行虚

基类构造函数的调用,再执行非虚基类构造函数的调用,即虚基类构造函数优先。

小 结

继承是面向对象程序设计中最重要、最关键性概念,有单继承和多继承两种类型。C++支持类的继承机制,从而进入实用化阶段。

在直接基类和直接派生类之间,常用的有公有继承和私有继承两种方式,由关键字 `public` 和 `private` 分别指定,客观世界中大量的关系可用公有继承来描述,在此继承方式下,基类的公有部分和保护部分的成员及其访问权限都毫无变化地被派生类继承下来,只有私有部分的成员才变成不可见的。而私有继承用在少数要求封装得很严密的场合,编程者应设法编写公有成员函数,从而打通派生类对象访问基类私有成员的消息通路。

在多层次的类层次结构中,继承的传递性取决于是公有继承还是私有继承。在公有继承方式下,基类的公有部分和保护部分的成员及其访问权限,都毫无变化地又传递给间接派生类,只有私有部分的成员在间接派生类作用域内也仍然不可见。在私有继承方式下,基类的所有成员在间接派生类作用域内都是不可见的。

“派生类对象访问从基类继承而来的私有成员”是一条重要的消息通路,初学者应掌握在各种情况下如何打通它的方法。

派生类构造函数在函数头内提供一个初始化列表,使得在创建派生类对象时,能自动地调用基类的构造函数来初始化从基类继承而来的数据成员,并能对派生类成员对象的数据成员进行初始化。

在公有继承方式下,基类指针和对象引用是联系基类和派生类对象间消息传递的桥梁。任何使用基类的对象、或对象指针或对象引用作为形参的函数,都同样可以用来处理派生类的对象,即利用子类型关系使用相同的函数统一处理基类对象和派生类对象。

C++的多继承机制会产生派生类对基类成员访问操作的“二义性”问题,解决办法是“成员名限定法”、在派生类体内重新定义同名的成员函数和使用虚基类等。

习 题 4

一、选择填空

- 下列对派生类的描述中,()是错误的。
 - 一个派生类可以作另一个派生类的基类
 - 派生类至少有一个基类
 - 派生类的成员除了它自己的成员外,还包含了它基类的成员
 - 派生类中从基类继承的成员,其访问权限保持不变
- 下列叙述中,()是错误的。
 - 公有继承方式下,基类的公有成员在派生类仍然是公有成员

- B. 公有继承方式下,基类的私有成员在派生类仍然是私有成员
C. 私有继承方式下,基类的公有成员在派生类变成私有成员
D. 保护继承方式下,基类的公有成员在派生类变成保护成员
3. 派生类对象对基类成员中的()是可以访问的。
A. 公有继承的公有成员 B. 公有继承的私有成员
C. 公有继承的保护成员 D. 私有继承的公有成员
4. 对基类和派生类的描述中,()是错误的。
A. 派生类是基类的具体化 B. 派生类是基类的子集
C. 派生类是基类定义的延续 D. 派生类是基类的组合
5. 派生类构造函数的成员初始化列表中不能包含()。
A. 基类的构造函数 B. 派生类中成员对象的初始化
C. 基类成员对象的初始化 D. 派生类中一般数据成员初始化
6. 关于多继承二义性的描述中,()是错误的。
A. 一个派生类的两个基类中都有某个同名成员,在派生类中对这个成员进行访问可能出现二义性
B. 解决二义性最常用的方法是用“类名::”指定成员名所属类
C. 基类和派生类中出现的同名成员函数也存在二义性问题
D. 一个派生类是从两个基类派生而来的,而这两个基类有一个共同的基类,对该基类成员进行访问时,也可能出现二义性
7. 设置虚基类的目的是()。
A. 简化程序 B. 消除二义性
C. 提高运行效率 D. 减少目标代码
8. 下列虚基类的声明中,()是正确的。
A. `class virtual B : public A`
B. `virtual class B : public A`
C. `class B : public A virtual`
D. `class B : virtual public A`
9. 在派生类对基类继承的传递性中,()是错误的。
A. 在公有继承方式下,直接派生类对象可以直接调用基类中的公有成员函数去访问基类的私有数据成员
B. 在公有继承方式下,间接派生类对象可以直接调用基类中的公有成员函数去访问基类的私有数据成员
C. 在私有继承方式下,直接派生类对象可以直接调用基类中的公有成员函数去访问基

类的私有数据成员

- D. 不管是私有继承还是公有继承, 基类的私有成员在派生类的作用域内都是不可见的
10. 带有虚基类的多层派生类构造函数的成员初始化列表中都要列出虚基类的构造函数, 这样将对虚基类的子对象初始化 ()。
- A. 与虚基类下面的派生类个数有关 B. 多次
C. 两次 D. 一次
11. 下面关于子类型的描述中, () 是错误的。
- A. 子类型关系是可逆的
B. 公有派生类的对象可以初始化基类的对象引用
C. 只有在公有继承方式下, 派生类是基类的子类型
D. 子类型关系是可以传递的
12. 下面 () 的叙述不符合赋值 (兼容) 规则。
- A. 派生类的对象可以赋值给基类的对象
B. 基类的对象可以赋值给派生类对象
C. 派生类的对象可以初始化基类的对象引用
D. 派生类对象的地址可以赋值给基类的对象指针
13. 下面的叙述中, () 是错误的。
- A. 派生类可以使用私有派生
B. 对基类成员的访问必须无二义性
C. 基类成员的访问能力在派生类中保持不变
D. 赋值 (兼容) 规则也适合多继承的组合
14. 下面程序代码中划线处, 正确的语句是 ()。

```
#include <iostream>
using namespace std;
class Base {
public:
    void fun( void )
    { cout << "Base::fun( ) called !" << endl; }
};
class Derived : public Base {
    void fun( void )
    { _____ // 显式调用基类的成员函数 fun( )
      cout << "Derived::fun( ) called !" << endl;
    }
public :
```


A. fun() B. Base.fun()
C. Base::fun() D. Base -> fun()

二、判断下列描述的正确性，对者划 ☐，错者划 ☒

1. C++语言中，既允许单继承，又允许多继承。
2. 派生类是从基类派生出来的，它不能再生成新的派生类。
3. 派生类的继承方式有两种：公有继承和私有继承。
4. 在公有继承中，基类中的公有成员和私有成员在派生类作用域内都是可见的。
5. 在公有继承中，派生类对象可以访问基类的公有成员。
6. 在私有继承中，派生类对象可以访问基类的公有成员。
7. 在私有继承中，基类中所有成员对间接派生类的对象都是不可见的。
8. 公有派生类的对象是可以访问基类的公有成员的。
9. 公有派生类的对象是可以访问基类的私有成员的。
10. 公有派生类的对象是可以访问基类的保护成员的。
11. 私有派生类的对象是可以访问基类的公有成员的。
12. 派生类是它的基类的组合。
13. 构造函数可以被继承。
14. 析构函数不能被继承。
15. 只要是类 M 继承了类 N，就可以说类 M 是类 N 的子类型。
16. 如果 A 类型是 B 类型的子类型，则 A 类型必然适应于 B 类型。
17. 在多继承情况下，派生类的构造函数的执行顺序取决于定义派生类时所指定的各基类的顺序。
18. 在单继承情况下，派生类中对基类成员的访问也会出现二义性。
19. 虚基类用来解决多继承中公共基类在派生类中只产生一个基类成员对象的问题。

三、指出下列程序中的错误，简洁说明错误原因，并加以改正

程序 1:

```
class    OneNum    {
    int n;
    OneNum(int v) {  n = v;  }
public :
    virtual void inc(void) {  n++;  }
    int get( void ) {  return n;  }
};

class    TwoNum : OneNum    {
    int m;
    OneNum(int v) {  n = v;  }
public :
    TwoNum(int v1, int v2) {  m = v2;  }
    void inc( void ) {  n++; m++  }
    int getSecond(void) {  return m;  }
};
```

程序 2:

```
#include    < iostream >
using    namespace    std;
class    Base    {
    float x , y ;
public :
    Base( float x1 , float y1 )  { x = x1 ;  y = y1 ; }
    void print( void ) { cout << " x = " << x << "\t y = " << y ; }
};

class    Derived : public Base    {
    float z ;
public :
    Derived(float z1) { z = z1 ; }
    void print( void )
    {  print( ) ;      cout << " z = " << z;    }
} ;

void    main( void )
{
    Derived    d(3, 4, 5);
    d.print( );
    cout << endl;
}
```

四、填空题

1. 对于 class 类型，缺省的继承方式是_____。
2. 若 Alpha 类继承了 Beta 类，则 Alpha 类称为_____类，Beta 类称为_____类。
3. 对基类数据成员的初始化必须在派生类构造函数中的_____处执行。
4. 设置虚基类的目的是_____，可通过_____标识虚基类。
5. 已知下面的程序框架，按注释中的提示补充细节。

```

#include    < iostream >
using namespace  std;
class Planet    {
protected :
    double  distance;          // 行星距太阳的距离
    int     revolve;           // 行星的公转周期
public :
    Planet( double  d , int  r )
    {
        distance = d;
        revolve = r ;
    }
};

class Earth : public Planet {
    double  circumference; // 地球绕太阳公转的轨道周长
public :
    // 定义构造函数 Earth(double d, int r), 并计算地球绕太阳公转的轨道周长
    // 假定 : circumference = 2 * d * 3.1416

    _____

    // 定义 show( )函数显示所有信息
    _____
};

main( void )
{
    Earth  obj( 9300000 , 365 );
    obj.show( );
    return 0;
}

```

五、写出源程序和程序输出结果

有下述程序，B 类是 A 类的公有派生类，C 类又是 B 类的公有派生类，若将它们都改成私有继承方式，为了确保 main() 函数体内的测试内容不变，请修改 B 类和 C 类，并写

出一个完整的源程序和程序的输出结果。

```
#include < iostream >
using namespace std;
class A {
    int value;
public :
    A( int v ) { value = v; }
    int readValue( void ) { return value; }
};
class B : private A {
    int total;
public :
    B( int v, int t ) : A( v )
    { total = t; }
    int readValue( void ) { return A::readValue( ); }
    int readTotal( void ) { return total; }
};
class C : private B {
    int count;
public :
    C( int v, int t, int c ) : B( v , t )
    { count = c; }
    int readValue( void ) { return B::readValue( ); }
    int readTotal( void ) { return B::readTotal( ); }
    int readCount( void ) { return count; }
};
void main( void )
{
    A a(2); B b(4, 6); C c(8, 10, 12);
    cout << " value of a = " << a.readValue( ) << endl;
    cout << " value of b = " << b.readValue( ) << endl;
    cout << " total of b = " << b.readTotal( ) << endl;
    cout << " value of c = " << c.readValue( ) << endl;
    cout << " total of c = " << c.readTotal( ) << endl;
    cout << " count of c = " << c.readCount( ) << endl;
}
```

六、分析下列程序的输出结果

程序 1 :

```

#include < iostream >
using namespace std;
class L {
    int X, Y;
public :
    void initL( int x, int y ) { X = x; Y = y; }
    void move( int x, int y ) { X += x; Y += y; }
    int getX( void ) { return X; }
    int getY( void ) { return Y; }
};
class R : public L {
    int W, H;
public :
    void initR( int x, int y, int w, int h )
{   initL( x, y ); W = w; H = h; }
    int getW( void ) { return W; }
    int getH( void ) { return H; }
};
class V : public R {
public :
    void fun( void ) { move(6, 2); }
};
void main( void )
{   V v;
    v.initR( 10, 20, 30, 40 );
    v.fun( );
    cout << "{" << v.getX( ) << " , " << v.getY( ) << " , "
        << v.getW( ) << " , " << v.getH( ) << "}" << endl;
}

```

程序 2 :

```

#include < iostream >
using namespace std;
class P {
    int pri1, pri2;
public :
    P( int p1, int p2 )
    {   pri1 = p1; pri2 = p2; }
}

```

```
int  incl( void )    {  return ++pri1;  }
int  inc2( void )    {  return ++pri2;  }
void  display( void )
{  cout << "pri1 = " << pri1 << "    pri2 = " << pri2 << endl;  }
};

class  D1 : private  P{
    int  pri3;
public :
    D1( int p1, int p2, int p3 ) : P( p1, p2 )
    {  pri3 = p3;  }
    int  incl( void ) {  return  P::incl( );}
    int  inc3( void ) {  return  ++pri3;  }
    void display( void )
    {  P::display( );  cout << "pri3 = " << pri3 << endl;  }
};

class  D2 : public  P    {
    int  pri4;
public :
    D2( int p1, int p2, int p4 ) : P( p1, p2 )
    {  pri4 = p4;  }
    int  incl( void )
    {  P::incl( );
      P::inc2( );
      return  P::incl( );
    }
    int  inc4( void )    { return  ++pri4;  }
    void display( void )
    {  P::display( );  cout << "pri4 = " << pri4 << endl;  }
};

class  D12 : private  D1, public  D2 {
    int  pri12;
public :
    D12( int p11, int p12, int p13, int p21, int p22, int p23, int p )
        : D1( p11, p12, p13 ), D2( p21, p22, p23 ) {  pri12 = p;  }
    int  incl( void )
    {  D2::incl( );
      return  D2::incl( );
    }
    int  inc5( void ) {  return ++pri12;  }
```

```

void display( void )
{   cout << "D2::display( )\n";       D2::display( );
    cout << "pri12 = " << pri12 << endl;
}
};

void main( void )
{   D12   d( 1, 2, 3, 4, 5, 6, 7 );
    d.display( );
    cout << endl;
    d.incl( );
    d.inc4( );
    d.inc5( );
    d.D12::incl( );
    d.display( );
}

```

程序 3：

```

#include    < iostream >
using namespace std;
class Person {
public:
    Person( void )
    {   cout << "Constructor of Person called !" << endl;   }
    ~ Person( void )
    {   cout << "Destructor of Person called !" << endl;   }
};

class Student : public Person {
public :
    Student( void )
    {   cout << "Constructor of Student called !" << endl;   }
    ~ Student( void )
    {   cout << "Destructor of Student called !" << endl;   }
};

class Teacher : public Person {
public :
    Teacher( void )
    {   cout << "Constructor of Teacher called !" << endl;   }
    ~ Teacher( void )

```

```
        {   cout << "Destructor of Teacher called !" << endl;   }
    };

int main( void )
    {   Student      s;
        Teacher      t;
        return 0;
    }
```

程序 4 :

```
#include    < iostream >
using namespace std;
class Data {
    int x;
public:
    Data( int x )
    {   cout << "Constructor of Data called !" << endl;   }
    ~ Data( void )
    {   cout << "Destructor of Data called !" << endl;   }
};

class Base {
    Data d1;
public:
    Base( int x ) : d1( x )
    {   cout << "Constructor of Base called !" << endl;   }
    ~ Base( void )
    {   cout << "Destructor of Base called !" << endl;   }
};

class Derived : public Base {
    Data d2;
public :
    Derived( int x ) : Base( x ) , d2( x )
    {   cout << "Constructor of Derived called !" << endl;   }
    ~ Derived( void )
    {   cout << "Destructor of Derived called !" << endl; }
};

main( void )
{   Derived      object( 6 );
    return 0;
}
```



```

}
```

程序 5 :

```

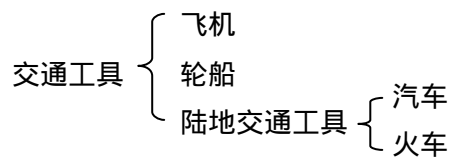
#include    < iostream >
using      namespace  std;
class      Base      {
    int     a, b;
public:
    Base( int  x, int  y )
    {      a = x;   b = y;   }
    void    show( void )
    {      cout << " Base : " << a << " , " << b << endl; }
};

class      Derived : public      Base {
    int     c;
public :
    Derived( int  x, int  y, int  z ) : Base( x , y ), c( z ) { }
    void    show( void )
    {      cout << " Derived : " << c << endl;      }
};

main( void )
{
    Base          b( 50, 50 ), * pb;
    Derived       d( 10, 20, 30 );
    pb = & b;
    pb -> show( );
    pb = & d;
    pb -> show( );
    return  0;
}
```

六、编程题

1. 用 C++ 中的派生类表示交通工具的如下分类 :



只描述类层次结构，类体内用省略号。

2. 建立一个基类 `Building`，用来存储一座楼房的层数，房间数以及它的总平方米数。
再定义一个派生类 `House` 从 `Building` 继承而来，并保存卧室和浴室的数量。同时由 `Building` 类派生出 `Office` 类，并保存灭火器和电话机的数目。
3. 大学一般有这几类人员：学生、教师、职员和在职读书的教师等。请编写描述这几类人员的 `class` 类型。

第 5 章 多态性和虚函数

面向对象编程语言 C++ 最重要的概念之一是多态性 (Polymorphism) 及其应用, 它来源于两个希腊词语——poly 和 morphism, poly 表示“多”的意思, morphos 的含义是“形态”。多态性是指 C++ 的代码可以根据运行情况的不同执行不同的操作。在实际程序中, 对象之间相互作用的方法可以任意组合, 即便是那些简单的类层次结构, 也会产生数量巨大的组合方式。因此, 常常必须依靠对象的多态性来开发中、大型软件。

5.1 C++ 的多态性

派生一个类的原因常常不仅是为了新添加一些描述新事例的变量、新的属性和操作它们的操作函数, 而是为了重新定义基类的成员函数, 以便满足派生类的新功能要求。当基类的成员函数在派生类中重新定义时, 其结果是使对象呈现多态性。因此不是整个类都具有多态性, 而是只有类的成员函数具有多态性。这种实现方式与自然语言中对动词的使用很类似, 此时动词类似于 C++ 的成员函数。一个对象若用“它”来表示, 则在现实生活中可以这样使用: “清洗它, 移动它, 分解它, 修理它。”这些动词仅代表了一般性动作, 因为不知道发生在哪种对象上。例如, 移动铅笔所需的操作完全不同于移动机床所需的操作, 尽管这两个概念是相似的, 只有知道“move(移动操作)”作用的对象, 才能将它与一系列的特殊操作联系起来。显然对于不同的对象, “move(移动操作)”的具体内容大不一样。

例如, 假设要把 draw() 函数添加到 Point 类和 Circle 类中, Point 类的 draw() 函数只是把一个点绘制在屏幕上, 而 Circle 类的 draw() 函数是以圆的形式画一连串的点, 如图 5.1 所示。

由于 Circle 类是 Point 类的派生类, 所以当 Point 和 Circle 类的对象 p 和 c 要进行相同的操作时, 需调用各自的成员函数 draw() 实现“画图”的操作, 其结果是画出不同形状的图, 这就是这类对象被称为多态性的原因。

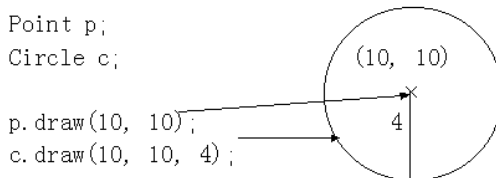


图 5.1 成员函数 draw() 的多态性

由此可知, 从基类 Point 继承而得派生类 Circle 类, 不仅能给派生类添加新的成员 (数据成员和成员函数), 而且更重要的

是可以重新定义基类中的成员函数 `draw()`，即在基类和派生类中有同名成员函数。其结果是使得它们的对象呈现出多态性。

C++的多态性表现在它为编程者提供了运算符重载、函数名重载和虚函数等运行机制。

5.2 运算符重载

5.2.1 运算符函数的定义

运算符重载并不是新概念，它在自然语言中已使用了几千年，例如一般的加法操作可以认为是“两数相加”、“将橘子和苹果加起来”和“给故事加上悬念”等。加法可用于抽象数据相加，也可用于可数的但相互没有关系的对象，如苹果和橘子相加，也可用于抽象事物，例如给故事增加悬念。因此加法的概念很直观，可表示数目增多或性质加强，但若不与上下文相联系，则加法的概念相当模糊。C++的运算符也类似于这种情况，C++语言原来定义的运算符是用来执行对基本数据类型的运算操作，但编程者定义了新的类型后，对已有的运算符必须赋予新的运算操作含义，使得它们适用于用户定义的新类型。即 C++语言本身提供的标准运算符可以重新在类中定义，使标准运算可作用于用户新定义的 class 类型的对象，从而使运算操作表现得自然而又合乎常规。

例 5.1 复数类加、减和赋值运算符的重载。

```
#include    < iostream >                // 使用 C++新标准的流库
using      namespace    std;            // 将 std 标准名空间合并到当前名空间

class      Complex {
    double  real, imag;
    // 私有数据成员 real 和 imag 分别保存复数对象的实数部分和虚数部分

public:
    Complex( void ) { real = imag = 0; }          // 无参数构造函数
    Complex( double r, double i )                // 一般构造函数
    { real = r; imag = i; }
    void operator = ( const Complex & c );        // 重载赋值运算符函数的声明
    Complex operator + ( const Complex & c );    // 重载加法运算符函数的声明
    Complex operator - ( const Complex & c );    // 重载减法运算符函数的声明
    friend void print( const Complex & c );      // 输出显示一个复数
};

// 重载赋值运算符函数的实现

void Complex::operator = ( const Complex & c )
{
    static int i = 0;
    // 定义一个内部静态变量 i，用来记录调用本重载赋值运算符函数的次数
    real = c.real ; imag = c.imag;              // 做“位模式拷贝”操作
    cout << "(" << ++i << ")" operator =( ) called ! ";
    // 输出显示调用本重载赋值运算符函数次数的信息
}
```

```

// 重载加法运算符函数的实现
inline Complex Complex::operator + ( const Complex & c )
{   return   Complex( real + c.real , imag + c.imag );   }
/* 创建一个匿名的自动型对象作为本重载加法运算符函数的返回值，调用构造函数利用其参数表实现两个复数的加法运算 */
// 重载减法运算符函数的实现
inline Complex Complex::operator - (const Complex & c )
{   return   Complex( real - c.real , imag - c.imag );   }
/* 创建一个匿名的自动型对象作为本重载减法运算符函数的返回值，调用构造函数利用其参数表实现两个复数的减法运算 */
void print( const Complex &c )
{   if( c.imag < 0 )           // 当复数的虚数部分为负数时直接输出
        cout << c.real << c.imag << 'i' << endl;
    else
        // 当复数的虚数部分为正数时，在前面加一个+号
        cout << c.real << " + " << c.imag << 'i' << endl;
}
void main( void )
{   Complex   a(2.0 , 3.0) , b(4.0 ,-2.0) , c;
    // 定义 3 个 Complex 类 a、b 和 c

    c = a + b;
    /* 由于表达式内含有 Complex 类的对象，将自动调用 Complex 类的重载加法运算符函数和重载赋值运算符函数，完成加法操作和赋值操作 */
    cout << " a + b = ";      // 输出显示" a + b = "的提示信息字符串
    print( c );              // 输出显示" a + b " 运算结果值

    c = a - b;
    /* 由于表达式内含有 Complex 类的对象，将自动调用 Complex 类的重载减法运算符函数和重载赋值运算符函数，完成减法操作和赋值操作 */
    cout << " a - b = ";      // 输出显示" a - b = "的提示信息字符串
    print( c );              // 输出显示" a - b " 运算结果值
}

```

该程序的输出结果为：

- ```

(1) operator =() called ! a + b = 6 + 1i
(2) operator =() called ! a - b = -2 + 5i

```

这时两个复数对象  $a$  和  $b$  相加就可以像整数加法一样写成“ $a + b$ ”的表达式形式，这与人们习惯的数学表达式完全一样，即赋予运算符“ $+$ ”号以新的运算操作含义，可实现复数的加法运算。显然要解决这一问题，必须在 `Complex` 类中对标准运算符“ $+$ ”重新定义，即定义一个重载加法运算符函数，使它可以作用于复数对象，实现复数的加法运算。同理，

编程者可重新定义“-”、“\*”、“/”等运算符,使它能作用于复数对象,按复数类所定义的规则完成复数的操作。

运算符重载是用定义重载运算符函数来实现的,重载运算符函数名由关键字 operator 和重载的运算符组成记为“operator <op>”,C++中的绝大多数标准运算符都能重载。如用 op 代表+, -, \*, / , %, ^ , & , | , ~ , ! , = , ... , ++ , - - , ... , [ ] , ( ) , new , delete 等。

定义运算符函数的关键是如何理解运算符表达式,并怎样转换成重载运算符函数的调用形式。这种转换取决于运算符是单目的还是双目的,且重载运算符函数可以定义为类的成员函数,也可以定义为友元(非成员)函数。所以,重载运算符函数有如下两种定义格式。

运算符函数重载为类的成员函数。由于成员函数隐含有 this 指针,作为隐含的参数传递给重载运算符函数,所以,对于单目运算符不带参数,对双目运算符仅带一个参数。其格式为:

```
<返回类型> <类名>::operator <op> (<类型> 参数)
{
 <函数体>
}
```

例如:

```
Complex operator +(const Complex & c);
```

转换

```
this -> operator +(b);
```

```
c = a + b; → a.operator +(b); // 重载加法运算符函数的调用格式
```

剖析上式可知,运算表达式的第一个运算量复数对象 a 为隐含对象,通过该对象 a 调用运算符函数 operator +(Complex & c)。表达式中的第二个运算量为显式对象 b,对形参 Complex & c 进行初始化,即在实参传递给形参过程中,相当于执行了“Complex & c = b;”形参引用 c 的初始化操作语句,使得形参引用 c 就是实参对象 b 的替换名,因此在该函数体内凡是对形参引用 c 的操作,实际上就是对实参对象 b 的操作。在 operator+( )函数体内将隐含对象 a 和形参对象 b 相加的结果作为该函数的返回值,在 main( )函数体内赋给 c。

运算符函数重载为友元函数。由于友元函数没有 this 指针,因此对于单目运算符友元函数必须带一个参数,对双目运算符应带两个参数,其格式为:

```
friend <返回类型> operator <op> (<类型> 参数 1, <类型> 参数 2)
{
 <函数体>
}
```

例如：

```
friend Complex operator + (const Complex & c1, const Complex & c2)
{ return Complex(c1.real + c2.real , c1.imag + c2.imag); }
```

若运算符表达式为“ $c = a + b;$ ”，则编译系统解释为调用重载加法运算符函数“ $\text{operator}+(a, b);$ ”的语句，求得复数对象  $a$  和  $b$  相加之和，然后再赋给对象  $c$ 。

综上所述，如表 5.1 和表 5.2 所示，将运算符表达式转换成运算符函数调用的形式，其中的  $x$  和  $y$  为对象名。



表 5.1 双目运算符函数

| 成员函数 | 表达式                 | 运算符函数调用                       |
|------|---------------------|-------------------------------|
| 是    | <code>X op Y</code> | <code>X.operator op(Y)</code> |
| 否    | <code>X op Y</code> | <code>operator op(X,Y)</code> |

表 5.2 单目运算符函数

| 成员函数 | 表达式               | 运算符函数调用                         |
|------|-------------------|---------------------------------|
| 是    | <code>op X</code> | <code>X.operator op( )</code>   |
| 是    | <code>X op</code> | <code>X.operator op(int)</code> |
| 否    | <code>op X</code> | <code>operator op(X)</code>     |
| 否    | <code>X op</code> | <code>operator op(X,int)</code> |

从表 5.1 可知,对于重载为成员函数的情况,运算符表达式的第一个运算量始终转换成对象 `x`,通过该对象调用运算符函数,则 `this` 指针指向该对象,并隐含地传递给运算符函数,在其函数体内,直呼其名地访问该对象的任何成员,特别是私有数据成员。而第二个运算量转换成重载运算符函数的实参对象 `y`,传递给重载运算符函数去初始化形参,在它的函数体内完成规定的运算操作。

表 5.2 列出了单目运算符的转换情况,由于单目运算符有“前置运算”(运算符放在运算量的前面)和“后置运算”(运算符放在运算量的后面)之分,参见表 2.2,有“++对象”、“--对象”等前置运算表达式,也有“对象++”、“对象--”等后置运算表达式。这些单目运算符在表 5.2 中仍然都用“`op`”表示,一个单目运算符的重载函数名是一样的,如“`operator ++( )`”,为了区分“前置运算”和“后置运算”只有将它们定义成两个重载函数用参数加以区分。对于“前置运算”,按单目运算符重载函数处理,即该运算只需一个运算量。若把它重载为成员函数,则把这惟一的运算量转换成调用该重载运算符函数的对象 `x`,由于成员函数隐含有 `this` 指针,该指针就指向调用该重载运算符函数的对象 `x`,并作为隐含的参数传递给该运算符函数,在这成员函数体内就可直呼其名地访问对象 `x` 的所有成员,特别是私有数据成员,用“`* this`”表达式就可以表示对象 `x`,所以,重载成员函数的参数表为空,即没有任何参数。如果把“前置运算”的单目运算符重载成非成员函数即友元函数(表 5.2 中的第 3 行),则这惟一的运算量就作为该友元函数的实参对象 `x` 传递给重载友元函数。对于“后置运算”,作为一种特殊的双目运算符处理,即将看成“对象 `x op` 运算量为零”的表达式。若重载为成员函数(表 5.2 中的第 2 行),对象 `x` 转换成调用该重载运算符函数的对象,而该函数具有一个特殊的参数,即一个 `int` 型的数值为零的虚拟参数传递给该重载运算符函数,该特殊参数在重载运算符函数定义时是匿名的,即只写数据类型 `int` 不写参数名,其函数体内完成该后置运算表达式所规定的操作。如果重载成非成员函数即友元函数(表 5.2 中的第 4 行),该友元函数将包含有对象 `x` 和 `int` 等两个参数,前者是单目运算符所需要的惟一运算量,后者是为了区分“前置运算”重载函数所添加的特殊参数。

两种重载形式的比较。通常单目运算符最好重载为成员函数,而双目运算符最好重载为友元函数,这是因为有些情况不能重载为成员函数,例如有一个复数对象 `c`,若程序

中写有表达式“ $6.8 + c$ ”，假如复数类中将“ $+$ ”重载特殊为成员函数，则编译系统解释为： $6.8.operator + (c)$ ，这显然是错误的。若将“ $+$ ”重载为友元函数，则编译系统解释为： $operator + (Complex(6.8), c)$ ，在 `Complex` 类中一定要有一个类型转换构造函数：

```
Complex::Complex(double r)
{ real = r; imag = 0.0; }
```

才能实现表达式“ $6.8 + c$ ”的运算。然而双目运算符中只有赋值运算符重载为成员函数，其他都重载为友元函数。

在例 5.1 中的重载加（减）法运算符函数体内，采用一个 `return` 语句，创建一个自动型的匿名对象作为该函数的返回值，并直接调用相匹配的一般构造函数初始化该匿名对象，而加（减）法运算操作直接在实参表中进行，即把两个复数对象的实数部分相加（或相减）作为结果的实数部分存放在匿名对象的数据成员 `real` 中，而将两个复数对象的虚数部分相加（或相减）作为结果的虚数部分存放在匿名对象的数据成员 `imag` 内。

下面再来确定这两个复数对象 `a` 和 `b` 与两个实参表达式“`real + c.real`”和“`imag + c.imag`”的关系。按表达式“`a + b`”转换成重载运算符成员函数的调用格式为“`a.operator + (b)`”，即第 1 运算量 `a` 应是调用本重载加（减）法运算符函数的对象，则 `this` 指针就指向了对象 `a`，因此在该成员函数体内直呼其名所访问的数据成员 `real` 应是对象 `a` 的数据成员，而第 2 运算量应为实参对象 `b`，它传递给形参引用 `&c` 时，就相当于执行了“`Complex & c = b;`”形参引用的初始化语句，因此在该重载运算符成员函数体内形参引用 `c` 就是实参对象 `b` 的替换名，即实参表达式“`real + c.real`”中的第 2 项应是实参对象 `b` 的实数部分。显然，该实参表达式就是把对象 `a` 的实数部分与对象 `b` 的实数部分相加起来，在一般构造函数体内赋给新创建的匿名对象的实数部分。而第 2 个实参表达式“`imag + c.imag`”是将对象 `a` 的虚数部分与对象 `b` 的虚数部分相加起来，在一般构造函数体内赋给新创建的匿名对象的虚数部分。

如前所述，由于匿名对象是自动型对象，因此这些函数的返回值类型必须是对象本身而不能是对象指针和对象引用，即返回方式只能采用传值方式而不能采用传地址方式。如果编程者希望采用传地址返回方式，可采用如下代码：

```
inline Complex & Complex::operator +(const Complex & c)
{
 static Complex result;
 // 定义一个 Complex 类的内部静态型对象 result，保存加法运算结果

 result.real = real + c.real;
 /* 将调用本重载运算符成员函数对象 a 的实数部分与对象 b 的实数部分相加的结果保存在静态型对象 result 的实数部分 */

 result.imag = imag + c.imag;
```

```

/* 将调用本重载运算符成员函数对象 a 的虚数部分与对象 b 的虚数部分相加的结果保存在静态对象 result 的虚数部分*/

return result;

/*由于函数返回到一个 (匿名的) Complex 类的对象引用 , return 语句后面的表达式只能采用静态型对象 result */

}

```

正如 3.3 节所述 , 采用传地址方式避免了调用复制构造函数 , 有利于提高程序的执行效率。

重载运算符函数的形参究竟是选用对象本身还是选用对象引用 , 即采用传地址方式还是传值方式。通常 , 以选用传值方式比较好 , 虽然 , 这会影响程序执行速度并增加内存消耗 , 但对于某些场合却能体现它的优越性。

例 5.2 重载运算符函数的形参以选用对象本身比较好。

```

#include < iostream.h > // 将 C++ 老标准流库的头文件 iostream.h 纳入本程序

class Complex {
 double real, imag;
 // 两个私有数据成员 real 和 imag 分别保存复数的实数部分和虚数部分

public:
 Complex(double r = 0 , double i = 0) : real(r), imag(i) { }
 // 由于采用形参默认值 , 该构造函数具有一般、无参数和类型转换等构造函数的功能

 void show(void) // 输出显示一个复数对象

 { cout << real << (imag < 0 ? " " : "+") << imag << "i\n" ; }

 friend Complex operator - (Complex , Complex);
 // 把减法运算符重载为友元函数

};

Complex operator - (Complex a, Complex b)
{ return Complex(a.real - b.real , a.imag - b.imag); }
// 选用 Complex 类的对象 a 和 b 作为重载减法运算符函数的形参

void main(void)
{
 Complex c1 , c2(6 , 8), c3(5.6 , 7.9), c4(c2);
 c3 = 8.9 - c2; // 双精度实数常量 8.9 减去复数对象 c2 再把结果赋给 c3

 cout << "c3 = ";
 c3.show(); // 输出显示复数对象 c3 的数据成员值

 c2 = c1 - 6.3; // 复数对象 c1 减去双精度实数常量 6.3 再把结果赋给 c2

 cout << "c2 = ";
 c2.show(); // 输出显示复数对象 c2 的数据成员值

}

```

该程序的输出结果为 :

```
c3 = 2.9 -8i
```

```
c2 = -6.3+0i
```

如果将重载减法运算符函数的形参改成 Complex 类的对象引用，即：

```
Complex operator - (Complex & a, Complex & b)
{ return Complex(a.real - b.real , a.imag - b.imag); }
```

则对于“8.9 - c2”和“c1 - 6.3”两个表达式将产生编译错误，若把第 2 个表达式转换成重载减法运算符函数的调用格式为：“operator - ( c1, 6.3 );”，从该调用语句可知，它显然是不合理的。因为对象引用是被引用对象的替换名，实参对象 c1 可以用来初始化形参引用 a，但常量 6.3 不是复数对象，不能用来初始化形参引用 b。

### 5.2.2 运算符重载规则

不能定义新的运算符，例如两个字符符号“\$”和“+”加在一起，在 C++ 中不是运算符，不能定义名为 operator \$( ) 的运算符函数。若没有合适的标准运算符，可直接编写函数，完成所需的操作。

不能改变 C++ 中运算符的优先级和运算量的个数，例如赋值号是双目运算符，因此不能将它重新定义成单目运算符函数，也不能改变优先级，即表达式“x = a + b;”永远表示求出“a + b”的和赋给 x，而不能定义成“(x = a) + b;”。

若运算符是一个非成员函数，则其参数中至少有一个必须是类的对象。例如，不能重新定义“+”运算符去连接两个字符串：

```
friend char * operator +(char * a , char * b)
{ ... }
```

为了防止出现编译混乱，该条规则禁止编程者去修改 C++ 中对基本数据类型进行的运算符操作。

不能将两个运算符合并去建立一个新的运算符。例如，可以直接重定义运算符函数 operator +=( )。但是有人认为，如果已经定义了 operator +( ) 运算符函数，再加上一个 operator =( ) 运算符函数把它们合并，以实现表达式“a += b;”的操作，这显然是错误的。

C++ 本身并不知道运算符重载表示什么操作含义，它是编程者自行定义的，因此也没有相应的检查措施，只有靠编程者自己检查，应用时应慎重选择合适的运算符。

下列标准运算符不能重载，以避免与 C++ 语法相矛盾：

|           |          |
|-----------|----------|
| ?: 三目运算符  | · 取成员运算符 |
| :: 作用域运算符 | * 取内容运算符 |

应保持运算符原来的操作含义，例如不要将“+”运算符定义成两值相减。将按位异或运算定义成指数运算等。

### 5.2.3 重载赋值运算符

可以重载的最重要的运算符是赋值运算符，它是双目运算符，如“`a = b;`”，编译系统解释为“`a.operator =(b);`”，其运算符函数(`operator =( )`)有如下限制。

如前所述，`operator =( )`必须重载为非静态（普通）成员函数，而不要重载成友元函数，这样处理可确保重载赋值运算符函数所获得的赋值目标运算量（赋值运算符左边的运算量即左值，也就是赋值运算操作的结果）始终是该类的一个对象，这与赋值表达式的语义是一致的，因为赋值运算操作的两个运算量应该是同一个 `class` 类型的两个对象，即把一个对象的每个数据成员按类定义中的顺序赋值给同类的另一个对象相对应的数据成员。

为了实现多重赋值，将赋值运算符函数返回值定义为目标对象的类型，即返回到目标对象。如例 5.1 中的复数类赋值运算符函数只能实现一次赋值：

```
void Complex::operator = (const Complex & c)
{
 ...
 real = c.real ; imag = c.imag;
 /* 把形参引用 c 的被引用（实参）对象的每个数据成员按类定义中的顺序赋值给同类的、
 调用本重载赋值运算符函数的另一个对象相对应的数据成员，即同类对象的“位模式拷贝” */
 ...
 // 由于本重载赋值运算符函数没有返回值，则不能实现多重赋值操作
}

void main(void)
{
 Complex a(4.0, 8.0) , b , c;
 // 定义 3 个 Complex 类的自动型对象 a、b 和 c

 b = a; c = b; // 两条赋值语句，一条语句只能进行一次赋值操作
 ↑ ↑
 目标对象 源对象

 ...
}
```

上面的两条赋值语句，实际上用一条多重赋值语句即可完成，例如：

```
c = b = a;
```

即完成两次或两次以上的赋值操作，它的操作是这样完成的，按照运算符的优先级和结合规则：...当具有相同优先级的运算符出现在一个表达式中时，根据结合规则来确定它们的运算顺序。因为赋值运算符“`=`”的结合规则是“从右至左”，上式实际的操作顺序如下：

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

这就要求“`b = a`”运算的结果值仍然是 `Complex` 类的一个对象，即重载赋值运算符函

数的返回值为 `Complex` 类的一个对象，但例 5.1 中的重载赋值运算符函数没有返回值而不能实现多重赋值。为了实现多重赋值，可将它定义为：

```
Complex Complex::operator = (const Complex & c)
{ real = c.real; imag = c.imag; // 同类对象的“位模式拷贝”
 return * this;
 // 返回到与调用本重载赋值运算符函数的对象同一 class 类型的另一个对象（目标对象）
}
```

该函数的最末端必须有“`return * this;`”语句，表示它返回到 `this` 指针所指的对象，按照上述表达式转换成调用运算符函数的规则应为：

|      |      |   |                 |             |
|------|------|---|-----------------|-------------|
| 目标对象 | 源对象  |   | 源对象             | 目标对象        |
| ↓    | ↓    |   | ↓               | ↓           |
| b    | = a; | → | b.operator =(a) | → Complex b |

在赋值表达式中，赋值运算符左边的运算量称为目标对象，赋值运算符右边的运算量称为源对象。由于是对象 `b` 调用重载运算符函数（非静态成员函数）`operator =(a)`，因此 `this` 指向对象 `b`，即 `*this` 就是对象 `b`。所以在多重赋值语句中，当将重载赋值运算符函数的返回值定义为目标对象的类型时，赋值运算符函数的返回值就是目标对象（赋值号左边的运算量），作为下一次赋值时的源对象（赋值运算符右边的运算量），因为“`b = a`”赋值操作的结果值为目标对象 `b`，因此“`c = b = a;`”语句执行完第一次赋值操作后，则变成“`c = b;`”再进行第二次赋值操作。

如前所述，由于引用是地址传递方式，书写简便，执行速度又快，因此通常将函数的参数和返回值定义成“类型 &”的形式，即函数的参数是源对象的引用，而函数返回值为目标对象的引用。即

```
Complex & Complex::operator = (const Complex & c)
{ real = c.real; imag = c.imag;
 return * this;
}

b = a; → b.operator =(a) → Complex &x = b;
```

这里是假想有个 `Complex` 类的对象引用 `x`（用 `x` 代表匿名的对象引用名）接受赋值运算符函数的返回值，`x` 对目标变量 `b` 引用，它就是目标变量 `b`，所以该运算符函数返回到目标变量 `b`。并且，由于调用赋值运算符函数和返回时均采用地址传递方式，所以执行速度快。

与复制构造函数一样，对任何类，当它没有编写重载赋值运算符函数时，C++ 自动提供一个默认的赋值运算符函数，但它只能完成“位模式拷贝”的任务，而不能为目标对象分配内存空间。对于含有指针类型数据成员类，且该类的构造函数又含有用 `new` 为其对象的数据成员分配堆（Heap）中的内存空间时，与复制构造函数一样，必须自行编写重载

赋值运算符函数，以避免程序隐患。

例 5.3 赋值运算符函数的深拷贝和浅拷贝问题。

```
#include <iostream> // 使用 C++ 新标准的流库
#include <cstring> // 标准函数 strcpy() 原型在其中
using namespace std; // 将 std 标准名空间合并到当前名空间

class Person {
 char * pName; // 字符串指针 pName 作为本类的私有数据成员
public:
 /* 类型转换构造函数，把形参字符串指针 pN 所指的 (实参) 字符串常量转换成 Person 类
 的对象，并将该字符串常量保存在该对象中 */
 Person(char * pN)
 {
 static int j = 0;
 // 定义一个内部静态变量 j，记录调用本构造函数的次数
 cout << "第" << ++j << "次调用类型转换构造函数 !\n";
 // 输出显示调用本构造函数次数的信息

 pName = new char[strlen(pN) + 1];
 /* 用 new 运算符在内存的堆 (heap) 中开辟一个存放形参字符串指针 pN 所指的 (实
 参) 字符串常量的空间，把 new 运算符结果值，即成功 (成功创建了一个 Person 类
 的动态对象) 开辟的内存空间首地址保存在本类的私有数据成员 pName 中 */

 if(pName) strcpy(pName, pN);
 /* "if(pName)" 也可写成 "if(pName != NULL)"，若 pName 不为空指针，即当
 new 运算符成功创建了一个新对象时，将形参字符串指针 pN 所指的 (实参) 字符串
 常量复制到新创建的 Person 类的对象中保存起来 */
 }

 Person(Person & p); // 复制构造函数的声明

 Person & operator = (Person & s);
 /* 重载赋值运算符函数的声明，将 Person 类的一个对象 (形参引用 s 的被引用对象，即实
 参对象) 赋值给调用本函数的另一个同类对象 */

 ~ Person(void) // 析构函数
 {
 static int k = 0;
 // 定义一个内部静态变量 k，记录调用本析构函数的次数
 cout << "第" << ++k << "次调用析构函数 !\n";
 // 输出显示调用本析构函数次数的信息

 pName[0] = '\0'; // 把要撤销对象的私有数据成员 pName 所指的字符串设置成空串
 delete pName; // 用运算符 delete 撤销该对象
 }

 void display(void) { cout << pName << endl; }
};
```

```

Person::Person(Person & p) // 复制构造函数
{
 cout << " 调用复制构造函数 !\n"; // 输出显示调用了本函数的信息

 pName = new char[strlen(p.pName) + 1];
 /* 用 new 运算符在内存的堆 (heap) 中开辟一个大小为形参对象引用 p 的数据成员 pName
 所指的字符串常量长度 (字符串个数) 加 1 的空间, 把 new 运算符结果值, 即成功开辟的
 内存空间首地址保存在同一 class 类型新创建对象的私有数据成员 pName 中 */

 if(pName) strcpy(pName, p.pName);
 /* "if(pName)" 也可写成 "if(pName != NULL)", 若 pName 不为空指针, 即当 new
 运算符成功地在内存的堆中开辟了一个空间, 再将形参对象引用 p(它的被引用对象是实参
 对象, 即一个已存在的对象) 的数据成员 pName 所指的字符串常量复制到新创建的对象中
 保存起来 */
}

Person & Person::operator = (Person & s)
{
 cout << " 调用赋值运算符函数 !\n"; // 输出显示调用了本函数的信息

 if(pName) {
 delete pName;
 pName = '\0';
 }
 /* "if(pName)" 也可写成 "if(pName != NULL)", 即若 pName 不为空指针, 则撤销目
 标对象已占有的内存空间, 并把要撤销对象的私有数据成员 pName 所指的字符串设置成空
 串 */

 pName = new char[strlen(s.pName) + 1];
 /* 给目标对象 (赋值运算符的左值) 重新分配新的内存空间, 即用 new 运算符在内存的堆
 (heap) 中开辟一个大小为形参对象引用 s 的数据成员 pName 所指的 (实参) 字符串常
 量长度 (字符串个数) 加 1 的空间, 把 new 运算符结果值, 即成功开辟的内存空间首地址
 保存在调用本重载赋值运算符函数对象 (赋值运算符的左值) 的私有数据成员 pName 中 */

 if(pName) strcpy(pName , s.pName);
 /* 若 pName 不为空指针, 即 new 运算符在内存的堆成功地开辟了一个内存空间, 则将右值
 对象所保存的字符串常量复制到左值对象中 */

 return * this; // 返回到调用本成员函数的对象
}

void main(void)
{
 Person p1("OldObject"); // 第 1 次调用类型转换构造函数
 Person p2 = p1; // 调用复制构造函数
 p1.display(); // 输出显示对象 p1 的字符串 "OldObject"
 p2.display(); // 输出显示对象 p2 的字符串 "OldObject"
 Person p3("NewObject"); // 第 2 次调用类型转换构造函数
 p3.display(); // 输出显示对象 p3 的字符串 "NewObject"
}

```



```

 p3 = p1; // 调用赋值运算符函数
 p3.display(); // 输出显示对象 p3 的字符串变成"OldObject"
}

```

该程序的输出结果为：

第 1 次调用类型转换构造函数！

调用复制构造函数！

OldObject (对象 p1 的字符串)

OldObject (对象 p2 的字符串)

第 2 次调用类型转换构造函数！

NewObject (对象 p3 赋值操作前的字符串)

调用赋值运算符函数！

OldObject (对象 p3 赋值操作后的字符串)

第 1 次调用析构函数！ (撤销对象 p3)

第 2 次调用析构函数！ (撤销对象 p2)

第 3 次调用析构函数！ (撤销对象 p1)

Person 类的私有数据成员 pName 为字符型指针变量，且构造函数又含有用 new 运算符为其对象的数据成员分配堆中的内存空间，同样存在深拷贝和浅拷贝问题(见 3.3 节)，对于这种场合，编程者必须自行定义赋值运算符函数和复制构造函数，再加上一个析构函数。这种自定义赋值运算符函数通常包含两部分，第一部分用来撤销目标对象（即作为左值的对象）已经占有的内存空间，第二部分再给目标对象重新分配新的内存空间。如例 5.3 中，在创建对象 p3 时，在堆中保存的字符串是“Newobject”，在执行“p3 = p1;”赋值语句的过程中，则调用重载赋值运算符函数，将表达式转换成运算符函数的调用格式为：

```
p3.operator = (Person & p1);
```

因此，首先撤销的对象应该是目标对象 p3，即将原先保存的字符串“Newobject”所占用的空间撤销掉由系统回收，然后再从堆中给对象 p3 分配新的空间，存储由源对象 p1“位模式拷贝”而得的字符串“OldObject”。

赋值运算符函数和复制构造函数的区别在于：C++会自动提供默认的赋值运算符函数和复制构造函数以方便用户，但是对于含有指针类型数据成员类，且该类的构造函数又含有用 new 运算符为其对象的数据成员分配堆(Heap)中的内存空间时，用户都必须自行编写赋值运算符函数、复制构造函数和析构函数，此时应注意两者的区别。

当用已存在的对象去初始化同类的新对象时，调用复制构造函数。如例 5.3 中，

```
Person p2 = p1;
```

在执行复制构造函数时，对象 p2 还不存在，由复制构造函数执行初始化操作。而对于“p3 = p1;”赋值语句，则调用赋值运算符函数将源对象 p1 赋值给目标对象 p3，此时 p1 和 p3 都是已存在的对象。

由于复制构造函数是用已存在的对象去初始化新对象，因此先编写用 `new` 运算符为新对象分配内存空间的语句，然后对新对象进行“位模式拷贝”即可。而赋值运算符函数的源对象和目标对象都已存在，因此应包含两部分，第一部分类似于析构函数的功能，撤销作为左值的对象，第二部分类似于复制构造函数的功能，即把作为右值的对象赋给作为左值的同一类型对象。

#### 5.2.4 典型应用

众所周知，经常用分数(Fraction)表示有理数(Rational)，若定义一个 Fraction (分数)类，它必然包含保存分子(numerator)值和分母(denominator)值的两个整数型数据成员，并且还必须重载 C++ 标准运算符集合中的很多运算符以适应分数的各种运算。由于实际应用的运算表达式有时很复杂，在编写重载运算符函数时应特别注意避免产生程序瑕疵。下面我们用例程来说明编程时应注意的技巧问题，因篇幅的限制仅以重载加法和乘法运算符函数为例加以说明。

例 5.4 Fraction (分数)类重载运算符函数的综合应用。

```
#include < iostream.h >
#include < math.h > // 标准函数 abs()的原型声明在其中
#include < stdlib.h > // 标准函数 exit()的原型声明在其中

class Fraction {
 int num, den;
 /* 用两个私有数据成员 num (numerator 的简写，分子) 和 den (denominator 的简写，
 分母) 分别记录分子值和分母值 */
 void standardize(void);
 /* 检测分数的分母是否为零，并除以分子和分母的最大公约数得到最简的标准形式，且分母
 总是正整数，分子的符号就是分数的符号 */
public :
 Fraction(int n = 0, int d = 1); // 带参数默认值的构造函数
 Fraction operator -(void) const; // 重载取负运算符函数
 void print(void); // 输出显示分数对象的值

 friend Fraction operator +(Fraction &summand, Fraction &addend);
 /* 重载加法运算符函数，返回到 Fraction 类的一个 (匿名) 对象，加法运算符两个运算量
 作为参数传递给该重载函数，即第 1 个形参 summand(被加数) 和第 2 个形参 addend(加
 数) */

 friend Fraction operator *(Fraction&faciend, Fraction&multiplier);
 /* 重载乘法运算符函数，返回到 Fraction 类的一个 (匿名) 对象，乘法运算符两个运算量
 作为参数传递给该重载函数，即第 1 个形参 faciend(被乘数) 和第 2 个形参 multiplier
 (乘数) */
```

```

};
int gstCDivisor(int m, int n) // 求最大公约数 (greatest common divisor)
{
 int temp; // 定义一个自动型变量 temp 保存中间计算结果
 while(n != 0) {
 temp = m;
 // 把第 1 个整数 m 暂时保存在变量 temp 中, 即把上一次循环的除数作为本次的被除数
 m = n;
 /* 再把第 2 个整数 n 赋给第 1 个整数 m, 即把上一次循环的余数 n 保存在 m 中, 该值将
 作为本次循环的除数 */
 n = temp % n;
 /* 此次循环若第 2 个整数 n 能整除第 1 个整数 m (表达式 “temp % n” 的结果值为零),
 则 n 将变成零, 随后将退出循环, 此时第 1 个整数 m 中保存的值就是最大公约数, 否则,
 把本次循环的除数 n 原来值 (上面赋值表达式已将它保存在第 1 个整数 m 中) 作为
 为下一次循环的被除数, 而把整除后的余数 n 作为下一次循环的除数再辗转相除, 直到
 余数 n 为零时退出循环 */
 }
 return abs(m);
 // 调用 C 语言的标准函数 abs() 把第 1 个整数 m 保存的最大公约数取绝对值
}

void Fraction::standardize(void)
{
 if(den == 0) { // 若除数为零, 也可写成 “if(! den)”
 cout << "发生除数为零的错误!!!" << endl;
 // 输出显示出错的提示字符串信息
 exit(1);
 // 异常终止路径, 调用 C 语言的标准函数 exit() 立即停止执行程序退回到操作系统
 }
 else if(num == 0) return;
 // 若分母不为零、分子为零则立即返回到调用侧, 不进行分数标准化处理
 else if(den < 0) {
 num = -num;
 den = -den;
 }
 // 当分母为负数时, 把分母 den 和分子 num 都变号, 分数值不变, 而分母 den 变正
 int gcd = gstCDivisor(num, den);
 // 调用 gstCDivisor() 函数求分子 num 和分母 den 的最大公约数保存在变量 gcd 中
 num /= gcd; // 用最大公约数 gcd 去除分子 num
 den /= gcd; // 用最大公约数 gcd 去除分母 den, 得到分数的最简形式
}

```



```

/* 调用构造函数对返回的一个 Fraction 类 (匿名) 对象进行初始化, 该分数对象的两个数
 据成员 num (分子) 和 den (分母), 其值分别等于两个形参对象引用, 即 summand (被
 加数) 和 addend (加数) 所指分数对象之积的分子和分母 */
}

void main(void)
{
 Fraction product, result, f1(1, 2), f2(-2, 3), f3(5, 6);
 // 创建 5 个 Fraction 类的自动对象, 其中 product 和 result 两个对象没有实参
 cout << "(1) 对象 result 的分数值 : "; // 输出显示第 1 次测试时结果的提示信息
 result.print(); // 输出显示分数对象 result 的分数值

 product = f1 * f2 * f3;
 // 先计算赋值语句的右值表达式, 即 3 个分数对象连乘 "f1 * f2 * f3"
 cout << "(2) 对象 product 的分数值 : ";
 product.print(); // 输出显示分数对象 product 的分数值

 result = -f1 + Fraction(2) * f2;
 cout << "(3) 对象 result 的分数值 : ";
 result.print();
}

```

该程序的输出结果为：

```

构造函数第 1 次被调用 ! (创建分数对象 product 的输出信息)
构造函数第 2 次被调用 ! (创建分数对象 result 的输出信息)
构造函数第 3 次被调用 ! (创建分数对象 f1(1, 2) 的输出信息)
构造函数第 4 次被调用 ! (创建分数对象 f2(-2, 3) 的输出信息)
构造函数第 5 次被调用 ! (创建分数对象 f3(5, 6) 的输出信息)

(1) 对象 result 的分数值 : 0 / 1

重载乘法运算符函数第 1 次被调用 ! (执行表达式 "f1 * f2" 调用重载函数的输出信息)
构造函数第 6 次被调用 ! (执行重载乘法运算符函数体内的 return 语句的输出信息)
重载乘法运算符函数第 2 次被调用 ! (执行表达式 "(f1*f2) * f3" 调用重载函数的输出信息)
构造函数第 7 次被调用 ! (执行重载乘法运算符函数体内的 return 语句的输出信息)

(2) 对象 product 的分数值 : -5 / 18

构造函数第 8 次被调用 ! (执行表达式 "Fraction(2)" 的输出信息)
重载乘法运算符函数第 3 次被调用 ! (执行表达式 "Fraction(2) * f2" 的输出信息)
构造函数第 9 次被调用 ! (执行重载乘法运算符函数体内的 return 语句的输出信息)
重载取负运算符函数第 1 次被调用 ! (执行表达式 "-f1" 调用重载取负运算符函数的输出信息)
构造函数第 10 次被调用 ! (执行重载取负运算符函数体内的 return 语句的输出信息)
重载加法运算符函数第 1 次被调用 ! (执行右值 "(-f1)+(Fraction(2)*f2)" 的输出信息)
构造函数第 11 次被调用 ! (执行重载加法运算符函数体内的 return 语句的输出信息)

(3) 对象 result 的分数值 : -11 / 6

```

## 1. Fraction 类的定义

在编写 Fraction 类的代码时,首先考虑的是把其对象的分子和分母作为私有数据成员 num(分子)和 den(分母),数据类型应定义成整型。为了使表示的有理数范围较大、精度较高,最好定义为 int 型或 long 型而不采用 short 型,又由于 Visual C++ 6.0 中 int 型与 long 型的数据长度是一样的都是 4 个字节(32bits,详见 2.1.1 节)。

Fraction 类还需要编写一个成员函数 standardize(),对分数进行标准化操作,它首先检测分数对象的分母不能为零,再找出分母不为零而分子为零的对象,对其不进行标准化操作立即返回到调用侧,即调用成员函数 standardize()的函数体内。接着,对分数对象进行如下标准化操作。

确保分母总是正整数,分数的正负由分子的符号确定。具体操作方法是:判断分母若为负,则对分子和分母都进行一次“取负”运算,分数值不变,但分母变成正整数。

成员函数 standardize()对于“检测分母是否为零、找出分母不为零而分子为零的对象和确保分母总是正整数”等 3 个操作是采用“else - if”形式的多分支结构条件语句实现的。

用分子 num 和分母 den 的最大公约数去除分数,以保证分子 num 和分母 den 总是互为质数的最简形式。

为此,例 5.4 专门编写了一个外部函数 gcdDivisor()求取两个整数的最大公约数,它用两个 int 型的形参 m 和 n 接收这两个整数,在该函数体内再定义一个 int 型自动变量 temp 保存中间计算结果,接着用一个 while 循环语句进行迭代运算,每循环一次把第 1 个整数 m 暂时保存在变量 temp 中,再把第 2 个整数 n 赋给第 1 个整数 m,即把本次循环的除数变成下一次循环的被除数,最后将暂时保存在变量 temp 中的第 1 个整数 m 与第 2 个整数 n 进行整除(两个整数相除所得的商只取整数部分),把整除所得的余数再保存在第 2 个整数 n 中,即把本次循环的余数变成下一次循环的除数。因此,每次循环结束时第 2 个整数 n 所保存的值实际上是上一次循环两个整数进行整除时的余数,如果在某次循环过程中,第 2 个整数 n 能整除第 1 个整数 m(表达式“temp % n”的结果值为零),则 n 将变成零,随后一旦执行“while( n != 0 )”即将退出循环,此时第 1 个整数 m 中保存的值就是最大公约数。换言之,在某次循环过程中,当第 1 个整数 m(它是被除数)能被第 2 个整数 n(它是除数)整除,即执行 while 循环体内的最后一条语句“n = temp % n;”,其赋值语句的右值表达式“temp % n”(变量 temp 的值就是第 1 个整数 m 的值)的结果值为零,使得左值 n 为零时将退出循环,而第 2 个整数 n 原来的值即为最大公约数,该 n 值在整除前已保存在第 1 个整数 m 中,对 m 取绝对值即可得到最大公约数。否则再进入下一次循环,把本次循环作为除数的 n 值(已将它保存在第 1 个整数 m 中)作为下一次循环的被除数,而把整除后的余数 n 作为下一次循环的除数再辗转相除,直到余数 n 为零时退出循环。

显然, 成员函数 `standardize()` 是专门对 `Fraction` 类的分数对象进行标准化操作的成员函数, 对其他 `class` 类型的对象是无意义的, 所以把它放在私有部分隐藏起来, 只有在 `Fraction` 类的成员函数体内才能调用它。

## 2. 慎重选择“是采用函数重载还是选用函数参数的默认值?”

在实际编程时, 经常会碰到这样的情况, 采用函数重载 (包括成员函数的重载) 方式还是选用函数参数的默认值, 例如: 若采用函数重载方式编写程序:

```
void fun(void); // 不带参数的重载函数 fun()
void fun(int a); // 带一个参数的重载函数 fun()
...
fun(); // fun()函数调用语句, 调用不带参数的重载函数 fun(void)
fun(12); // fun()函数调用语句, 调用带一个参数的重载函数 fun(int a)
```

那么, 选用函数参数的默认值编写程序为:

```
void fun(int a = 6); // fun()选用函数参数的默认值
...
fun(); // fun()函数调用语句, 调用 fun(6)
fun(12); // fun()函数调用语句, 调用 fun(12)
```

显然, 这两种编写方式其函数调用语句的格式完全一样, 若两种方式都采用则将产生软件领域最忌讳的、模棱两可的二义性问题, 如:

```
void fun(void); // 不带参数的重载函数 fun()
void fun(int a = 6); // 带参数默认值的重载函数 fun()
...
fun();
// fun()函数调用语句, 产生“是调用 fun(6)还是调用 fun(void)”的二义性问题
```

通常, 只要能为参数找到适当的默认值, 最好选用函数参数的默认值方式编写程序。另外, 成员函数是函数的一种同样存在这个问题, 例 5.4 中的 `Fraction` 类就是一个典型的例子。有人可能已想到对于 `main()` 体内的如下一条执行语句:

```
result = -f1 + Fraction(2) * f2;
```

若在 `Fraction` 类体内再定义一个类型转换构造函数, 即:

```
class Fraction {
 int num, den;
 void standardize(void);
public:
 Fraction(int n = 0, int d = 1); // 带参数默认值的一般构造函数
 Fraction(int n)
 { num = n; den = 1; }
 /* 类型转换构造函数, 把 int 型整数 n 转换成 Fraction 类的对象, 其分母为 1 而分子为
```

```

 n 的分数 */
 ...
};

```

这样一来，好像可以把上面的执行语句改写成：

```
result = -f1 + 2 * f2;
```

在执行“`2 * f2`”表达式时系统会自动调用类型转换构造函数，将整数常量 2 自动转换为分母为 1 而分子为 2 的分数。其实，若将“`2 * f2`”表达式转换成重载乘法运算符函数的调用格式即为：

```
operator *(2 , f2);
```

因重载乘法运算符函数是友元函数，其两个形参都是 Fraction 类的对象引用，它们的被引用对象即实参对象也都是 Fraction 类的对象，因此，系统应自动调用类型转换构造函数把整数 2 转换成 Fraction 类的一个分数对象，即：

```
operator *(Fraction(2) , f2);
```

但是，这种转换即执行实参表达式“`Fraction(2)`”时已经产生了“是调用带参数默认值的一般构造函数还是调用类型转换构造函数”的二义性问题。因此，遵照上述编程规则，选用只定义一个带参数默认值的构造函数，对于表达式中的整数 `n` 一律采用以该整数 `n` 作为惟一的实参，第 2 个实参缺省的调用格式“`Fraction(n)`”（详见文献[26]条款 24 和 38）。

如前所述，也可以将重载加法和乘法运算符函数的形参引用改成对象本身，即采用传值方式，则这些分数对象就可以直接与常量进行加法和乘法运算了，如：

```

class Fraction {
 int num, den;
 void standardize(void);
public :
 Fraction(int n = 0, int d = 1);
 Fraction operator -(void) const;
 void print(void);
 friend Fraction operator +(Fraction summand, Fraction addend);
 // 两个形参 summand 和 addend 都是 Fraction 类的对象
 friend Fraction operator *(Fraction faciend, Fraction multiplier);
 // 两个形参 faciend 和 multiplier 都是 Fraction 类的对象
};

...

void main(void)
{ Fraction product, result, f1(1, 2), f2(-2, 3), f3(5, 6);
 ...
}

```



```

 result = -f1 + 2 * f2; // 可直接书写与数学表达式“2 * f2”完全一样的形式
 result.print();
}

```

由于形参是 Fraction 类的对象本身, 若把表达式“2 \* f2”转换成重载乘法运算符函数的调用格式即为:

```
operator *(2 , f2);
```

那么, 在实参传递给形参的过程中, 相当于执行了如下两条初始化操作语句:

```

Fraction faciend = 2;
Fraction multiplier = f2;

```

其中, 当执行第 1 条语句时, 将自动地调用构造函数“Fraction(2)”把常量 2 转换为 Fraction 类的一个对象再与对象 f2 相乘, 从而使书写程序的格式更加合乎常规。

### 3. 返回一个 (匿名) 对象的函数, 千万不要返回函数体内的局部对象的引用

当函数需要返回一个对象时, 为避免程序隐患, 不要传递回函数体内定义的局部对象 (包括自动对象和内部静态对象) 的引用, 也不要传递回一个在函数体内用 new 运算符创建的动态对象, 可以在 return 语句的表达式中直接调用本类的构造函数, 用来初始化作为返回值的匿名对象, 即返回值传递采用传值方式。如例 5.4 中的重载加法运算符函数:

```

Fraction operator +(Fraction & summand, Fraction & addend)
{
 ...
 return Fraction(summand.num * addend.den + summand.den * addend.num,
 summand.den * addend.den);
 /* 调用构造函数对返回的一个 Fraction 类 (匿名) 对象进行初始化, 该分数对象的两个数
 据成员 num (分子) 和 den (分母), 其值分别等于两个形参对象引用, 即 summand (被
 加数) 和 addend (加数) 所指分数对象之和的分子和分母 */
}

```

这样处理虽然会引起调用默认复制构造函数 (因本例程的 Fraction 类没有自行编写复制构造函数, 所以调用系统自动提供的默认复制构造函数) 的额外开销 (详见 3.3.2 节), 但对于多重运算 (连续几个对象相乘或相加) 的表达式却是避免程序瑕疵、确保安全的最佳选择, 如例 5.4 中的 main() 函数体内有如下一条执行语句:

```
product = f1 * f2 * f3;
```

其右值表达式需要 2 次调用重载乘法运算符函数, 若把它转换成重载乘法运算符函数的调用格式则有:

```
product = operator * (operator *(f1 , f2) , f3);
```

因为乘法运算符的结合规则是从左至右, 所以上面右值表达式的运算顺序是先做“f1 \* f2”再做“(f1 \* f2) \* f3”运算, 当第 1 次调用完成时, 得到了(f1 \* f2)的结果值, 并把它赋给重载乘法运算符函数的第 1 个形参后, 传递回的匿名对象需要被删除掉, 编译

系统会自动完成这项操作任务，其最佳化机制能确保安全高效地完成，详细信息请参阅文献[26]的条款 23 和 31。

## 5.3 其他运算符的重载

### 5.3.1 重载增量、减量运算符

增量、减量运算符是单目运算符，但它们有前置运算和后置运算之分，为了区分前、后置运算，C++约定把前置运算符仍重载为单目运算符函数，即表达式“++a;”可解释为“a.operator ++( );”(重载为成员函数)，而把后置运算符看成双目的，重载为双目运算符函数，在其参数表内放上一个特殊的虚拟整数形参，该形参只是用来区分前、后置运算，此外无任何作用，所以在函数头中省略形参名，即重载为成员函数的格式为：

|                                                                                   |
|-----------------------------------------------------------------------------------|
| <pre>&lt;返回类型&gt; &lt;类名&gt;::operator &lt;op&gt; (int) {     &lt;函数体&gt; }</pre> |
|-----------------------------------------------------------------------------------|

表达式“a++;”可等价看做“a ++(0);”，这可解释为“a.operator ++(0);”(重载为成员函数)。该重载增(减)量运算符函数的操作内容应符合“后置运算”的相关规定。

对于前置增量运算符，仍将其重载为单目运算符函数，重载为成员函数的格式为：

|                                                                                       |
|---------------------------------------------------------------------------------------|
| <pre>&lt;返回类型 &amp;&gt; &lt;类名&gt;::operator &lt;op&gt; ( ) {     &lt;函数体&gt; }</pre> |
|---------------------------------------------------------------------------------------|

例 5.5 说明了根据前、后置增量操作的含义而决定其返回类型，前置增量运算符返回对象引用，后增量运算符返回对象的值。

例 5.5 重载增量运算符。

```
#include <iostream> // 使用 C++新标准的流库
using namespace std; // 将 std 标准名空间合并到当前名空间

class Counter {
 unsigned value; // 私有数据成员 value 用来对所发生的事件计数
public:
 Counter(void) { value = 0; }
 Counter & operator ++(void); // 前置增量运算符重载函数
 Counter operator ++(int); // 后置增量运算符重载函数
 void print(void); // 输出显示对象 value 值的成员函数
};

Counter & Counter::operator ++(void) // 前置增量运算符重载函数
{
 value++; // 直接把 Counter 类对象的 value 值增 1
 cout << " 调用前置增量运算符函数 !\n"; // 输出显示本成员函数被调用的信息
```

```
 return * this; // 该函数的返回值是 value 值增 1 后的对象
 }

Counter Counter::operator ++(int)
{
 Counter t; // 定义一个暂存对象 t, 保存增 1 前的对象
 t.value = value++; // 把增 1 前的对象 value 值保存在暂存对象 t 中
 cout << " 调用后置增量运算符函数 !\n";
 // 输出显示本后置增量运算符成员函数被调用的信息
 return t; // 该函数的返回值是 value 值增 1 前的对象
}

void Counter::print(void)
{
 static int i = 0; // 定义一个内部静态变量 i 记录显示的序号 (初值为 1)
 ++i; // 把显示的序号值增 1
 cout << "(" << i << ") 对象的 value 值 = " << value << endl;
 // 输出显示对象的 value 值
}

void main(void)
{
 Counter c , d; // 创建两个 Counter 类的对象 c 和 d
 cout << "一、检测后置增量运算符函数 : \n"; // 输出显示检测序号和内容
 for(int i = 0; i < 3; i++)
 c++; // 把对象 c 经 3 次增量操作, 即它的 value 值变成 3
 c.print(); // 输出显示对象 c 的 value 值

 d = c++;
 // 赋值表达式先计算右值表达式, 即对象 c
 c.print();
 d.print();
 cout << "二、检测前置增量运算符函数 : \n";
 for(i = 0; i < 3; i++)
 ++c;
 c.print();
 d = ++c;
 c.print();
 d.print();
}
}
```

该程序的输出结果为：

一、检测后置增量运算符函数：

调用后置增量运算符函数！

调用后置增量运算符函数！

调用后置增量运算符函数！

(1) 对象的 value 值 = 3                      (对象 c 的 value 值)

调用后置增量运算符函数 !

(2) 对象的 value 值 = 4                      (对象 c 的 value 值)

(3) 对象的 value 值 = 3                      (对象 d 的 value 值)

二、检测前置增量运算符函数 :

调用前置增量运算符函数 !

调用前置增量运算符函数 !

调用前置增量运算符函数 !

(4) 对象的 value 值 = 7                      (对象 c 的 value 值)

调用前置增量运算符函数 !

(5) 对象的 value 值 = 8                      (对象 c 的 value 值)

(6) 对象的 value 值 = 8                      (对象 d 的 value 值)

### 5.3.2 函数调用运算符( )的重载

函数调用运算符不是双目运算符,而是“多目”运算符,可适用于多个参数的情况,例如用于多维矩阵,作为接收多个参数的下标运算符使用,而下标运算符[ ]只能有一个参数。

例 5.6 函数调用运算符( )的重载实例。

```
#include <iostream> // 使用 C++新标准的流库
using namespace std; // 将 std 标准名空间合并到当前名空间
class A {
 int value[10][10];
 // 以 int 型的二维数组 value 作为本类的私有数据成员
public:
 int & operator()(int x, int y) { return value[x][y]; }
 // 重载函数调用运算符函数
};

void main(void)
{
 A a; // 创建 A 类的一个自动对象 a
 int i, j; // 定义两个 int 型的自动变量 i 和 j 作为数组下标
 for(i = 0; i < 3; i++)
 for(j = 0; j < 3; j++)
 a(i, j) = i * 10 + j;
 // 初始化 A 类的私有数据成员二维数组 value[10][10]的每个元素
 for(i = 0; i < 3; i++) {
 cout << "\n value[" << i << "] = ";
 for(j = 0; j < 3; j++) {
 cout << a(i, j); // 输出显示 value[10][10]的每个元素
```

```

 if(i == 0) cout << " "; // 第 0 行用 3 个空格分隔
 else cout << " "; // 其他行用 2 个空格分隔
 }
}
cout << "\n"; // 输出一个换行符

int x = 1, y = 1;
// 定义两个 int 型的自动变量 x 和 y, 并赋初值都是 1

a(x, y) = a(x + 1, y) + a(x, y + 1);
/* 求得自动对象 a 的数据成员、数组元素 value[2][1]和 value[1][2]之和, 再赋给它
 的 value[1][1]元素 */

cout << " a(" << x + 1 << ", " << y << ") = " << a(x + 1, y);
// 以 "a(2, 1) = xx" 格式输出显示 "a(2, 1)" 表达式的结果值

cout << "\n a(" << x << ", " << y + 1 << ") = " << a(x, y + 1);
// 以 "a(1, 2) = xx" 格式输出显示 "a(1, 2)" 表达式的结果值

cout << "\n a(" << x << ", " << y << ") = " << a(x, y) << endl;
// 以 "a(1, 1) = xx" 格式输出显示 "a(1, 1)" 表达式的结果值
}

```

该程序的输出结果为：

```

value[0] = 0 1 2
value[1] = 10 11 12
value[2] = 20 21 22
a(2, 1) = 21
a(1, 2) = 12
a(1, 1) = 33

```

说明：

通过重载函数调用运算符可实现如下数学函数的抽象：“ $a(i, j) = a * 10 + j$ ”，它用来对 `int` 型二维数组 `value[10][10]` 的每个元素进行初始化。因此，二维数组 `value[ ][ ]` 第 0 行的 3 个列元素的值分别为 0、1 和 2，第 1 行分别为 10、11 和 12，第 2 行分别为 20、21 和 22。

在执行 `main( )` 函数体内的 “ $a(x, y) = a(x + 1, y) + a(x, y + 1);$ ” 语句的过程中，函数调用运算符重载函数被调用了 3 次，即：

该赋值表达式首先计算右值的第 1 项 “ $a(x + 1, y)$ ”，因 `x` 和 `y` 的值都是 1，所以表达式 “ $a(x + 1, y)$ ” 即为 “ $a(2, 1)$ ”，若将函数调用运算符函数重载为成员函数，按前述的转换规则，第 1 运算量 `a` 应转换为调用该成员函数的对象，而后面的两个运算量 “2, 1” 则转换为该成员函数的两个实参传递给该函数，得到的成员函数调用格式为：“`a.operator( )(2, 1)`”，两个实参 2 和 1 分别传递给该成员函数的形参 `x` 和 `y`，在

成员函数体内，读取数组元素 `value[2][1]` 的值，所以计算右值表达式的第 1 项（即第 1 次调用“函数调用运算符重载函数”，即读取数组元素 `value[2][1]` 的值）的执行过程为：

`a(2, 1) → a.operator( )(2, 1) → 读 value[2][1] ( = 21) 的值`

接着计算右值表达式的第 2 项，即读取数组元素 `value[1][2]` 的值，则第 2 次调用过程为：

`a(1, 2) → a.operator( )(1, 2) → 读 value[1][2] ( = 12) 的值`

然后求得右值表达式的结果值，再赋给左值表达式，即赋给数组元素 `value[1][1]`（第 1 行第 1 列元素），此时第 3 次调用该成员函数读取数组元素 `value[1][2]` 的值，其执行过程为：

`a(1, 1) → a.operator( )(1, 1) → 读 value[1][1] ( = 33) 的值`

类 A 并没有重载“+”运算符。由于类 A 的函数调用运算符函数返回一个 `int` 类型的引用，所以编译系统可以采用为 `int` 类型定义的内部“+”操作，实现类 A 的加法运算。

函数调用运算符只能定义成非静态成员函数而不能是友元函数。

### 5.3.3 下标运算符[ ]的重载

下标运算符[ ]用于数组，在定义数组的说明语句中，方括号只是用来指定元素的个数，例如：

```
char ch, str[10] = "abcdefghi";
ch = str[6];
```

后一条语句才是使用了下标运算符访问第 6 号元素，由于编译系统不进行下标范围的检查，故无法确保访问数组元素不会越界。采用 `class` 类型可定义一种更安全、功能更强的数组类型，为此应对该 `class` 类型重载下标运算符[ ]。

例 5.7 下标运算符[ ]的重载实例。

```
#include <iostream> // 使用 C++ 新标准的流库
using namespace std; // 将 std 标准名空间合并到当前名空间

class CharArray {
 char * text; // 字符串指针 text 作为私有数据成员
 int size; // 私有数据成员 size 记录字符个数即数组大小
public:
 CharArray(int sz) { size = sz; text = new char[size]; }
 /* 构造函数，用 new 运算符在内存的堆 (heap) 中开辟一个大小由形参 sz 指定的内存空间，
 开辟成功时将该内存空间的首地址保存在私有数据成员 text 中 */
 ~ CharArray() { delete [] text; }
 // 析构函数，用 delete 运算符撤销数据成员 text (字符串指针) 所指的内存空间
```

```

char & operator[](int i); // 重载下标运算符函数
int getSize(void) { return size; } // 读取字符串数组的元素个数
};

char & CharArray::operator[](int i)
{
 static int k = 0; // 定义一个内部静态变量 k 记录调用本函数的次数
 if(i > size - 1) { // 检测下标超出范围,形参 i 作为下标
 i = size - 1; // 当下标超出范围时,取用最大下标[size - 1]
 cout << "\n(" << ++k << ")下标超出范围,返回到最后一个元素 !";
 // 输出显示“下标超出范围¼”的提示信息
 }
 return text[i];
 /* 返回一个字符变量的引用,其被引用变量是调用本成员函数对象的数据成员——字符串指针
 text 所指的字符串常量中的第 i 个字符,若下标没有超出范围则返回到第 i 个字符 */
}

void main(void)
{
 int cnt;
 CharArray str1(6); // 定义 CharArray 类的对象数组 str1
 char * str2 = "string"; // 定义一个字符串指针 str2 指向一个字符串常量
 for(cnt = 0; cnt < 8; cnt++)
 str1[cnt] = str2[cnt];
 /* 由于右值表达式 str2[cnt]是基本数据类型——字符串数组的元素,按系统规定的标准
 下标运算符进行操作,而左值表达式 str1[cnt]是 CharArray 类对象数组的元素,
 则自动调用重载下标运算符函数 */
 cout << "\n"; // 输出一个换行符
 for(cnt = 0; cnt < 8; cnt++)
 cout << str1[cnt];
 // 自动调用重载下标运算符函数,输出 str1 对象的字符串
 cout << "\n数组 str1 的元素个数 = " << str1.getSize() << endl;
 // 输出显示 CharArray 类对象数组 str1[cnt]的元素个数
}

```

该程序的输出结果为：

- (1)下标超出范围,返回到最后一个元素 !
- (2)下标超出范围,返回到最后一个元素 !

strin

- (3)下标超出范围,返回到最后一个元素 !
- (4)下标超出范围,返回到最后一个元素 !

数组 str1 的元素个数 = 6

下标运算符被认为是双目运算符,其第 1 个运算量是对象本身,第 2 个运算量是

重载下标运算符函数的参数。如例 5.7 中 CharArray 类对象数组的第 cnt 号元素下标表达式 “str1[cnt]”, 第 1 个运算量就是该对象数组元素本身, 第 2 个运算量 cnt 作为重载下标运算符函数的实参, 转换成重载下标运算符函数的调用格式为 “str1.operator[ ]( cnt)”。因此, 若将下标运算符函数重载为非静态 (普通) 成员函数, 则它只能有一个参数, 不可带多个参数, 即只能重载一维数组的下标运算符。例如:

```
CharArray a, b[40]; // 创建 CharArray 类的对象 a 和对象数组 b[]
a = b[20]; // 把对象数组 b[] 的第 20 号元素赋给对象 a
CharArray c[40][40]; // 创建一个 CharArray 类的二维对象数组 c[][]
a = c[20][10];
/* 出错, 编程者希望把二维对象数组 c 第 20 行第 10 列元素赋给对象 a, 因重载下标运算符只能带一个参数 */
```

在重载下标运算符中不会出现一个以上的方括号, 若确实需要这种表示时, 可采用函数调用运算符 “a = b(20, 10);”。

重载下标运算符函数只能声明为非静态 (普通) 成员函数, 不能重载为友元函数。因为重载下标运算符的目的就是使得下标运算符可以用于对象数组, 即直接用 “下标运算符[ ]” 去访问某个 class 类型对象数组的元素, 它的第 1 个运算量应该是对象数组元素本身, 第 2 个运算量是重载下标运算符函数的实参即一个已被赋值的下标变量。而访问基本数据类型数组元素仍然按系统规定好的运算内容进行, 不会去调用某个 class 类型中所重载的下标运算符函数。

可以将重载下标运算符函数的返回值类型指定为某个类的对象引用, 从而可以将该下标表达式写在赋值号的左边, 即作为左值表达式。如例 5.7 中:

```
char & CharArray::operator[](int i) // 重载成 CharArray 类的成员函数
{
 ...
 return text[i]; // ..., 若下标没有超出范围则返回到第 i 个字符
}
void main(void)
{
 ... 转换
 str1[4] = 'n'; → str1.operator[](4) = 'n';
 /* 将字符'n'赋给对象数组 str1[] 的第 4 号元素, 该赋值表达式把对象数组的下标表达式 str1[4]作为左值表达式 */
}
```

与所有双目运算符函数一样, 第 1 个运算量是对象 str1 本身, 第 2 个运算量是函数的参数, 即下标变量 int 型参数 i ( = 4 )。

C 和 C++ 都没有检查数组下标越界的机制,



```

char & CharArray::operator[](int i)
{
 static int k = 0;
 if(i > size - 1) {
 i = size - 1;
 cout << "\n(" << ++k << ")下标超出范围，返回到最后一个元素 !";
 }
 return text[i]; // 执行 (基本数据类型) 字符串数组的下标操作
}

```

下标运算符函数 `operator[ ]( )` 返回到第 `i` 个元素的引用。若 `i` 越界，则返回到最后一个元素，这种方法迫使下标在界内，这是由函数体内的条件语句来完成的。另一种方法是输出报错信息并立即终止程序，可写成：

```

#include <iostream>
using namespace std;
...
char & CharArray::operator[](int i)
{
 if(i > size - 1) {
 cout << "\n下标超出范围 !";
 exit(1);
 }
 return text[i]; // 执行 (基本数据类型) 字符串数组的下标操作
}

```

请注意 `CharArray` 类下标运算符函数的返回语句“`return text[i];`”，它虽然执行数组的下标操作，但不会发生递归调用，因为 `text[i]` 是字符型为基本数据类型，而不是 `class` 类型，编译系统将 `text[i]` 解释为 `*( text + i )`，不会再次调用 `CharArray` 类的下标运算符函数 `operator[ ]( )`，不会导致无穷递归调用。

当 `main( )` 函数体内的 `for` 语句循环变量 `cnt` 变成 6 以后，将一个越界下标 (`CharArray` 类对象 `str1` 的下标范围为 0~5) 传递给下标运算符函数 `operator[ ]( )`，它将检查出越界下标并迫使下标在界内 (最大下标为 5，使得第 2 个 `for` 语句只输出“`strin`” 5 个字符)，接着向显示器屏幕输出越界信息，静态变量 `k` 记录了越界次数，读者可自行分析程序的输出结果。

## 5.4 同名成员函数

C++ 中，同名的成员函数通常产生在如下两种情况。其一是在同一个类作用域内，多个成员函数体使用同一个成员函数名，这产生了成员函数的重载 (overloading)，其二是把 (从) 基类 (继承) 的成员函数在派生类作用域内重新加以定义以满足派生类的应用

要求，即在派生类作用域内，重新给该成员函数编写一个函数体实现新的功能，这样就在基类和派生类的两个类作用域类产生了同名成员函数，但因不在同一个作用域内将不发生重载而是覆盖（overriding），即派生类的成员函数掩盖了（从）基类（继承）的同名成员函数。由于成员函数的功能就是依靠顺序执行成员函数体内的一条条语句来实现的，因此，同名成员函数的多个不同的函数体称为不同的实现版本。下面就这两种情况的同名成员函数加以讨论。

#### 5.4.1 重载成员函数

与普通函数的重载类似，成员函数的重载是在同一个类作用域内，多个函数体可以使用相同的成员函数名，这类成员函数称为重载成员函数。这类成员函数或者参数的个数不同，或者至少有一个参数的类型不同。实际选择调用哪一个成员函数的匹配规则与普通重载函数类同。例如：

```
class Counter {
public:
 Counter(void) { v = 0; } // 无参数构造函数
 Counter & operator ++(); // 前置增量运算符函数
 Counter operator ++(int); // 后置增量运算符函数
 void print();
private:
 unsigned value;
};
```

如前所述，几乎每个类都定义了多个构造函数，以适应创建对象时对象的参数具有不同个数的情况，即定义了重载构造函数。

#### 5.4.2 基类和派生类的同名成员函数

在类层次结构中，基类的成员函数在派生类中重新定义，即在基类和派生类中有同名的成员函数。它们在编译时被加以区分，即编译系统确定调用哪一个类的成员函数，有如下区分方法。

根据类对象加以区分。即使基类和派生类的同名成员函数具有不同参数，也仍然是基类对象调用基类的成员函数，派生类对象调用派生类的成员函数。

例 5.8 区分基类和派生类的同名成员函数。

```
#include <iostream> // 使用 C++ 新标准的流库
using namespace std; // 将 std 标准名空间合并到当前名空间
class A {
```

```

public:
 int foo(int i) { return i + 1; }
 // 基类的公有成员函数 foo(), 带有一个 int 型形参 i

};

class B : public A {
public:
 float foo(float f) { return f + 10; }
 // 在派生类作用域类将 foo()重新定义, 带有一个 float 型形参 f 和新的函数体

};

void main(void)
{
 A a; B b;
 cout << "(1) 基类对象调用基类的成员函数 foo(), 其结果为 : "
 << a.foo(5) << endl;
 cout << "(2) 派生类对象调用派生类的成员函数 foo(), 其结果为 : "
 << b.foo(2) << endl;
 cout << "(3) 派生类对象调用基类的成员函数 foo(), 其结果为 : "
 << b.A::foo(6) << endl;

}

```

该程序的输出结果为：

- (1) 基类对象调用基类的成员函数 foo( ), 其结果为 : 6
- (2) 派生类对象调用派生类的成员函数 foo( ), 其结果为 : 12
- (3) 派生类对象调用基类的成员函数 foo( ), 其结果为 : 7

例 5.8 中, 由于派生类 B 中定义的同名成员函数 foo(float f) 与从基类 A 中继承的成员函数 foo(int i) 的作用域不同, 故虽然它们的形式参数不同, 但在派生类 B 的作用域内不会发生重载, 仍然是基类对象 a 调用基类的成员函数 A::foo(), 派生类对象 b 调用派生类的成员函数 B::foo()。

在派生类的成员函数体内需要访问基类中的同名成员函数, 则必须使用作用域分辨符加以区分, 否则将产生无穷循环的递归调用, 其格式为：

基类名::成员函数名( 参数表 );

例如：

```

void Derived::print(void)
{
 Base::print();
 // 在派生类作用域内调用基类同名的成员函数, 必须用基类名加作用域指明
 ...
}

```

在公有继承方式下, 派生类对象访问基类中公有部分的同名成员函数, 也必须使用作用域

运算符加以区分，其格式为：

派生类对象.基类名::成员函数名(参数表)；

在派生类中重定义的成员函数掩盖了基类中所有的同名成员函数，即使基类中的同名成员函数的参数表与派生类完全不同，在派生类作用域内基类同名成员函数也只有用“基类名::”的格式指明才能访问到它们。如例 5.8 中，基类和派生类具有同名成员函数 `foo()`，前者参数为 `int` 型，后者为 `float` 型，但“`b.foo(2)`”表达式中，实参为整数 2 仍然是调用派生类的成员函数 `B::foo(2)`，若派生类对象 `b` 调用基类的同名成员函数 `foo()` 必须写“`A::foo(6)`”。

### 5.4.3 基类指针和派生类对象

虽然在类层次结构中，基类指针可用来指向公有派生类的对象，但是当基类和派生类中具有同名的成员函数时，编程者即使让基类指针指向派生类对象，然而由于“静态联编”机制的特点，编译系统也只能根据基类指针定义时的类型说明，确定基类指针总是指向基类对象。这种情况又经常发生在呈现多态性的程序中，即在派生类作用域内重新定义从基类继承而来的成员函数，使得基类和派生类具有同名的成员函数。

例 5.9 测试基类和派生类具有同名的成员函数的调用情况。

```
#include <stdio.h> // 把 C 语言标准函数库的头文件 stdio.h 纳入本程序

class A {
public:
 void display(void) { puts("Class A"); }
 /* 基类的公有成员函数 display()，其函数体内调用标准运行库中的 puts() 函数，将字符串 "Class A" 输出在显示器上，以表示调用的是基类的 A::display() */
};

class B : public A {
public:
 void display(void) { puts("Class B"); }
 /* 在派生类作用域类重新定义公有成员函数 display()，其函数体内调用标准运行库中的 puts() 函数，将字符串 "Class B" 输出在显示器上，以表示调用的是派生类的同名成员函数 B::display() */
};

void show(A * p) { p -> display(); }
// show() 的形参是基类 A 的对象指针，在其函数体内用基类的对象指针 p 去调用 display()

void main(void)
{
 A * pa = new A; // 创建一个基类 A 的动态对象
 B * pb = new B; // 再创建一个派生类 B 的动态对象
```

```

pa -> display(); // 多接口界面, 基类动态对象 pa 调用 A::display()
pb -> display(); // 多接口界面, 派生类动态对象 pb 调用 B::display()

Show(pa);
// 通过 show() 的单接口界面, 用基类动态对象 pa 作实参, 调用的是 A::display()

Show(pb);
// 通过 show() 的单接口界面, 用派生类动态对象 pb 作实参, 仍然是调用 A::display()
}

```

该程序的输出结果为：

```

Class A
Class B
Class A
Class A (通过 show() 的单接口界面, 用派生类动态对象 pb 作实参)

```

说明：

puts( ) 是 C 语言标准函数库中的函数，其原型在 stdio.h 中，即

```
int puts(char * str);
```

puts( ) 函数将形参字符型指针 str 所指的实参字符串常量写到标准输出设备（显示器）中去。当执行调用语句“puts(“ClassA”);”时，则在显示器屏幕上出现 ClassA。因此，如果调用了基类 A 中的成员函数 A::display( )，则在显示器屏幕上出现 ClassA；而若调用了派生类 B 中的成员函数 B::display( )，则出现 Class B。编程者的意图是在 main( ) 函数内，第一次调用 show(pa)，实参采用基类 A 的指针 pa，第二次调用 show(pb)，实参采用派生类 B 的指针 pb，在显示器屏幕上似乎应该得到

```

Class A
Class B

```

的结果，但其实并非如此。

show( ) 函数对基类 A 和派生类 B 中的两个同名成员函数 display( ) 的访问都是通过指向基类 A 的指针 p 实现的。然而在 main( ) 函数内，当第二次调用 show(pb)，将实参传递给形参时，main( ) 函数内的 B 类指针 pb 自动转换成 A 类指针，再赋给 show( ) 函数的形参(A \* p)，因此“show(b);”语句理所当然地调用 A::display( )。由此可知，show( ) 函数总是只能访问基类 A 中的成员函数 A::display( )，而无法访问派生类 B 中的同名成员函数 B::display( )。屏幕上显示的结果为：

```

Class A
Class A

```

而达不到编程者的目的。

这种情况经常出现在编程者设计的“单界面，多实现版本”的程序中，正如例 5.9 中，设计一个函数 show( )，使用基类指针“A \* p”作为它的形式参数，形成了一个“单界

面”。再利用“一个指向基类的指针可用来指向从基类公有继承的任何对象”这一重要规则，在基类和派生类各层中，编写各种不同实现版本的同名成员函数 `A::display()`、`B::display()`...但是由于“静态联编”的弊病使得“在基类和派生类中有同名成员函数的情况下，编译时基类指针总是指向基类对象”而达不到目的。因此，要实现“单界面，多实现版本”的思想，必须依靠虚函数和“动态联编”机制。

从例程可知，编程者是想构造一个“单界面，多实现版本”的框架，这是面向对象编程方法的重要思想，这种方法可以让编程者控制很复杂的程序。然而，采用上述“静态联编”的方法，却不能实现“单界面，多实现版本”的框架。于是有人可能会提出在 `main()` 函数内编写如下语句：

```
pa -> display(); // 调用 A::display()
pb -> display(); // 调用 B::display()
```

上面是在 `main()` 函数体内定义了两个动态对象 `pa` 和 `pb`。对于自动型对象也可编写如下语句：

```
void main(void)
{
 A a; B b;
 a.display(); // 调用 A::display()
 b.display(); // 调用 B::display()
}
```

但是它们都是多接口界面。因为 `A::display()` 和 `B::display()` 都是普通的成员函数，则按“静态联编”机制来实现调用，即在编译时按调用成员函数的对象确定好调用哪个类的成员函数。如前所述，仍然是基类对象 `a` 调用基类的成员函数 `A::display()`，派生类对象 `b` 调用派生类的成员函数 `B::display()`，显然它们不是单接口界面而是多接口界面。

由于对象引用也是地址传递方式，与对象指针有类似的性质，即基类的对象引用可用于从基类公有继承的任何派生类对象”。因此，例 5.9 可改写为：

```
void show(A & r) { r.display(); }
void main(void)
{
 A a; B b; // 定义基类 A 的自动型对象 a 和派生类 B 的自动型对象 b
 a.display(); // 多接口界面，基类自动对象 a 调用 A::display()
 b.display(); // 多接口界面，派生类自动对象 b 调用 B::display()
 show(a);
 // 通过 show() 的单接口界面，用基类自动对象 a 作实参，调用 A::display()
 show(b);
 // 通过 show() 的单接口界面，用派生类自动对象 b 作实参，仍然调用 A::display()
}
```

顺便指出, 例 5.9 是“单接口界面, 多实现版本”构想的框架程序, 在计算机软件领域这种构想现在已得到广泛采用, 例如在 Windows 操作系统中, 对于各种不同类型的窗体, 如主窗口、对话框和按钮等, 操作者都是用鼠标左键去点击, 如响应“鼠标点击事件”的窗口函数就是一个“单接口界面”, 而这些不同类型的窗体所属类均在 Visual C++ 系统的标准类库 MFC (Microsoft Foundation Classes) 的类层次结构系统中, 如图 5.2 所示。

在 Windows Support 这条分支上, 由 CWnd 类公有派生出 CFrameWnd (帧窗口) 类、CButton (按钮) 类和 CDialog (对话框) 类等, 那么, 鼠标左键点击这些窗体对象, 如主 (帧) 窗口、按钮和对话框等时将作出不同的响应操作, 这就必须编写“多实现版本”的窗口函数。下面再举一个应用例程加以说明。

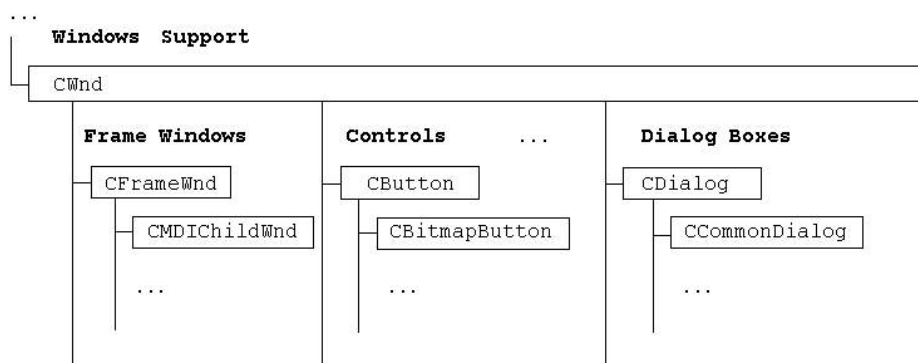


图 5.2 CWnd 类、Cbutton 类和 CDialog 类等的类层次结构图

例 5.10 又一个“单接口界面, 多实现版本”失败的例子。

```

#include <iostream> // 使用 C++ 新标准的流库
using namespace std; // 将 std 标准名空间合并到当前名空间

class Point {
public:
 Point(double i, double j) { x = i; y = j; }
 double area(void) const { return 0.0; } // 点的面积总是零

private:
 double x, y; // 基类 Point 的私有数据成员 x 和 y 记录点的 x 和 y 坐标值
};

class Rectangle : public Point { // 公有继承方式
public:
 Rectangle(double i, double j, double k, double l);
 // 形参 i 和 j 初始化长方形左上角基准点的 x、y 坐标, 形参 k 和 l 初始化它的宽和高
 double area(void) const { return w * h; } // 计算长方形面积的成员函数
};

```

```

private :
 double w , h ;
 // 公有派生类 Rectangle 的新增数据成员 w 和 h 分别记录长方形的宽和高
};
Rectangle::Rectangle(double i, double j, double k, double l) : Point(i, j)
{ w = k ; h = l ; }
// 公有派生类 Rectangle 的构造函数

void fun(Point & s)
{ cout << "The AREA = " << s.area() << endl; }
// 以基类 Point 的对象引用 s 作为形参，调用成员函数 area()

void main(void)
{ Rectangle rec(3.0 , 5.2 , 15.0 , 25.0);
 /* 定义一个派生类 Rectangle 的自动对象 rec，长方形左上角基准点的 x、y 坐标分别为
 3.0 和 5.2，宽和高分别为 15.0 和 25.0 */

 fun(rec);
 // 以派生类 Rectangle 的对象 rec 作为实参调用 fun() (单接口界面)
}

```

该程序的输出结果为：

```
The AREA = 0
```

程序的输出结果表明，由于采用静态联编，在 fun( ) 函数中，由形参 s 所引用的对象执行调用 area( ) 函数的操作，在程序编译阶段被选择到 Point::area( ) 函数体代码上，从而导致输出不期望有的结果，而编程者希望 s 所引用的对象执行调用 area( ) 函数的操作应选择到 Rectangle::area( ) 函数体代码上，完成计算长方形面积的任务，这是静态联编所无法做到的。

因此，“单接口界面，多实现版本”是一个十分有实用价值的程序框架。然而，由于静态联编机制的弊病而无法实现，人们必须寻求新的解决途径，动态联编技术就应运而生了。

## 5.5 虚函数

### 5.5.1 静态联编

如前所述，函数的功能是按顺序执行函数体内一条条语句来实现的，在编译时函数体这块程序片段被编译成一块由机器码构成的函数体代码模块（详见 2.6.3 节）。对于重载函数究竟调用同名函数的哪一个是在编译时按实参的个数和类型去寻找相匹配的那个函数（详见 2.6.4 节），将函数的调用语句链接上该函数体代码模块，一旦执行到函数调用语句时，当函数调用的准备操作完成后程序立即转向执行该函数体代码模块。所谓“联编”



是指调用函数时链接上相应函数体的代码模块,这一程序执行过程称为联编。例如:

```
...
print(3.1415926);

...
print(double value)
{ cout << value; }
```

C++有两种函数联编机制——静态联编和动态联编。静态联编 (Static Binding 或先期联编 Early Binding) 是指在编译时,编译系统根据传递给该函数的实参对象 (某种类型的实例) 就能决定调用固定 (即程序运行期间不能改变) 的函数体代码段。例如,编译系统根据浮点数 3.1415926 这一实例,调用 `print( double value ){ cout << value; }` 这一函数体代码段,生成向显示器屏幕输出浮点数的机器码代码段。在静态联编方式下,编译系统在编译该语句时就知道函数调用过程中将要使用哪些对象,并能决定调用哪一个函数,然后生成能完成语句功能的可执行代码。静态联编支持 C++ 中的运算符重载和函数名重载这两种形式的多态性。与动态联编相比,其优点是运行开销小 (仅传递参数,执行函数调用,清除堆栈等),程序执行速度快,缺点是灵活性差。

### 5.5.2 虚函数机制和动态联编技术

为了实现“单接口界面,多实现版本”的程序框架,C++提供了一种解决方法,引用虚函数机制,即采用动态联编技术。

#### 1. 虚函数的定义

虚函数是基类的公有部分或保护部分的某成员函数,在函数头前加上关键字“virtual”,其格式为:

```
class 基类名 {
public : (或 protected :)
 virtual <返回类型> 成员函数名(参数表);
 ...
};
```

如例 5.9 中,若将基类中的成员函数 `display()` 指定为虚函数,即:

```
#include <stdio.h> // 把 C 语言标准函数库的头文件 stdio.h 纳入本程序

class A {
public:
 virtual void display(void) { puts("Class A"); }
};

class B : public A {
public:
 virtual void display(void) { puts("Class B"); }
```

```

};
void show(A * p) { p -> display(); } // 多态函数
void main(void)
{
 A * pa = new A; // 创建一个基类 A 的动态对象
 B * pb = new B; // 再创建一个派生类 B 的动态对象
 pa -> display(); // 多接口界面, 基类动态对象 pa 调用 A::display()
 pb -> display(); // 多接口界面, 派生类动态对象 pb 调用 B::display()

 show(pa);
 /* 通过多态函数 show() 的单接口界面, 用基类动态对象 pa 作实参, 动态选中调用
 A::display() */
 show(pb);
 /* 通过多态函数 show() 的单接口界面, 用派生类动态对象 pb 作实参, 动态选中调用
 B::display() */
}

```

该程序的输出结果为：

```

Class A (多接口界面, 基类对象 A 调用基类的成员函数 A::display() 输出的信息)
Class B (多接口界面, 派生类对象 A 调用基类的成员函数 B::display() 输出的信息)
Class A (单接口界面, 用基类动态对象 pa 作实参, 动态选中调用 A::display() 输出的信息)
Class B (单接口界面, 用派生类动态对象 pb 作实参, 动态选中调用 B::display() 输出信息)

```

虚函数可以在一个或多个公有派生类中被重新定义（即可以有不同的实现版本），但在派生类中重新定义时，该同名虚函数的原型，其中包括返回类型、参数的个数和类型以及它们的排列顺序等，都必须完全相同，否则将不产生动态联编。

一旦在基类中说明了一个虚函数，那么对于所有后续派生类它仍然是虚函数，即使在派生类中没有用关键字“virtual”指明。

用虚函数实现程序运行时，多态性的关键之处在于必须用指向基类的指针访问虚函数。只有当同一个指向基类的指针访问虚函数时，多态性才能实现。如例 5.9 的改进版中，一个单界面的函数 show( )，其形参采用指向基类 A 的指针 (A \* p)。而在 show( ) 函数体内指明调用虚函数“p -> display( );”，即用指向基类 A 的指针 p 去访问虚函数。但究竟是调用 A::display( ) 还是 B::display( )，取决于程序运行时基类 A 的对象指针 p 的内容（地址值）。在执行调用语句“show(pa);”，把实参传递给形参时，将 pa 赋给了基类 A 的对象指针 p，则调用 A::display( )。而执行“show(pb);”时，则将 pb 赋给了基类 A 的对象指针 p，而调用 B::display( )。因此，是根据同一个指向基类的指针 p 传递消息“动态选中”，激活对象所属类的虚函数，完成该实现版本所规定的操作。这种“动态选中”的性质称为虚特性，也称虚函数机制。所谓动态联编（Dynamic Binding 或滞后联编 Late Binding）是指在程序运行时刻，将函数的界面（形式参数）

与函数的不同实现版本 (函数体) 动态地进行匹配的联编过程。顺便指出, 访问虚函数的 `show(A * p)` 函数, 在实现了动态联编机制后能呈现多态性, 故称为 “多态函数”。

如前所述, 引用也是通过地址方式传递消息, 对虚函数的访问也可通过对基类 A 的对象引用来进行。此时 `show( )` 可定义为:

```
void show(A & a) { a.display();}
void main(void)
{
 A a; B b; // 定义基类 A 和派生类 B 的自动型对象 a 和 b
 show(a); // 调用 A::display()
 show(b); // 调用 B::display()
}
```

虚函数只有在类层次结构中的对象才有意义。对于不是类层次结构中的基类成员函数, 虽然在语法上可以定义为虚函数, 但只会引起不必要的运行时间开销。如图 5.3 所示, 一个虚函数是属于基类 A 公有部分或保护部分的成员函数, 那么这个虚函数可以在 A 类的公有派生类 B 中重新定义, 即一个函数原型完全相同, 而函数体内的执行代码不同的同名虚函数, 得到了在派生类 B 中的新实现版本, 编程者可以用一个单接口界面 `show(A * p)` 函数调用多个实现版本 `A::display( )` 和 `B::display( )`。

虚函数必须是类的成员函数, 非成员函数即普通函数和友元函数不能说明为虚函数, 构造函数和静态成员函数也不能说明为虚函数, 但析构造函数可说明为虚函数, 称为 “虚析构造函数”。

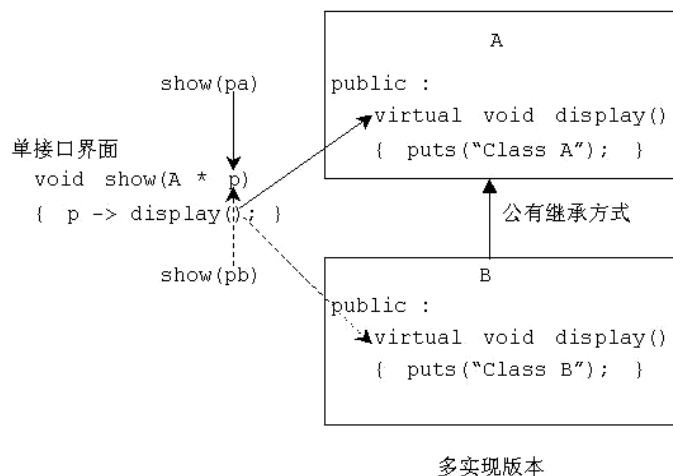


图 5.3 虚函数只能用于类层次结构中的对象

## 2. 虚函数和重载成员函数的区别

尽管虚函数和重载成员函数貌似相同, 但两者实际上有本质区别。

重载成员函数使用静态联编机制, 虚函数采用动态联编机制。

对于基类 A 中指定的虚函数 `f( )`, 编程者若在派生类 B 中重新定义 `f( )`, 必须

确保派生类 B 中的新函数与基类 A 中的虚函数  $f()$  具有完全相同的函数原型 ( 其中包括函数名、参数的个数和类型以及排列顺序 ), 才能覆盖原虚函数  $f()$  而产生虚特性, 执行动态联编机制。否则, 只要有一个参数不同, 编译系统就认为它是一个全新的函数, 而不实现动态联编机制。

#### 例 5.11 不实现动态联编机制的实例。

```
#include <iostream> // 使用 C++ 新标准的流库
using namespace std; // 将 std 标准名空间合并到当前名空间

class A {
public:
 virtual void print(int a, int b)
 // 基类 A 的虚函数具有两个 int 型的形参
 { cout << "a = " << a << " , b = " << b << endl; }
};

class B : public A {
public:
 virtual void print(int a, double d)
 { cout << "a = " << a << " , d = " << d << endl; }
 // 派生类 B 的虚函数 print() 有一个形参与基类同名虚函数不同
 /* 虽然把 print() 指定为虚函数, 但它与派生类中重新定义的函数原型不同, 将不产生动态联编, 仍在静态联编机制下运行 */
};

void show(A * p)
{ p -> print(3, 5.8); }

void main(void)
{ A * pa = new A;
 B * pb = new B;

 pa -> print(4, 8);
 pb -> print(6, 6.8);
 pb -> print(8, 16);
 show(pa); // 调用 A::print()
 show(pb); // 按静态联编机制仍调用 A::print()

 delete pa;
 delete pb;
}
```

本例中的多态函数 `show(A * p)` 总是调用基类的虚函数 `A::print(int a, int b)`, 因为派生类中的虚函数 `A::print(int a, double d)` 与基类的虚函数 `A::print( )` 原型不完全相同, 这样前者不能覆盖后者, 即不执行动态联编机制。

如前所述,重载成员函数必须在同一个作用域内,确保函数参数个数不同,或者至少有一个参数的类型不同。

### 5.5.3 典型例程

例 5.12 静态联编和动态联编混用的例程。

```
#include <stdio.h>

class A {
 int a;
public:
 virtual void f(void) { puts("Function A::f()"); }
 // 基类的虚函数 f(), 它被调用时输出显示"Function A::f()"字符串信息
 void g(void) { puts("Function A::g()"); }
 virtual void h(void) { puts("Function A::h()"); }
 // 基类的虚函数 h(), 它被调用时输出显示"Function A::h()"字符串信息
};

class B : public A {
 int b;
public:
 void g(void) { puts("Function B::g()"); }
 virtual void h(void) { puts("Function B::h()"); }
 // 公有派生类的同名虚函数 h() 被调用时输出显示"Function B::h()"字符串信息
};

void do(A & a)
{
 a.f(); a.g(); a.h();
}

void main(void)
{
 A aa; // 定义基类 A 的自动型对象 aa
 B bb; // 再定义派生类 B 的自动型对象 bb
 do(aa); // 以基类 A 的自动型对象 aa 作为实参调用 do() 函数
 do(bb); // 以派生类 B 的自动型对象 bb 作为实参调用 do() 函数
}
```

该程序的输出结果为:

```
Function A::f()
Function A::g()
Function A::h()
Function A::f()
Function A::g()
Function B::h()
```

A 为基类, 公有继承得 B 类。因此, 在派生类 B 的公有部分仍有一个从基类 A 继

承而来的虚函数  $f()$ ，由于它在派生类中没有重新定义，故其函数体代码仍然是：

```
virtual void B::f(void) { puts("Function A::f();") }
```

这也就是说，基类  $A$  继承的虚函数  $f()$  和派生类的虚函数  $B::f()$  具有完全相同的函数原型，则采用动态联编机制。

基类  $A$  中的成员函数  $g()$  在公有派生类  $B$  中又重新定义了一个新的实现版本。但因为  $g()$  不是虚函数，故仍采用静态联编。在  $main()$  函数体内，执行 “ $do(a);$ ” 语句将实参  $aa$  传递给形参  $A \& a$  时，相当于执行了一条引用初始化语句 “ $A \& a = aa;$ ”，因为  $aa$  是基类对象的引用，故理所当然调用基类  $A$  的成员函数  $A::g()$ ，向显示器屏幕输出 “Function A::g()” 信息。

基类  $A$  中的成员函数  $h()$  在公有派生类  $B$  中又重新定义了一个新实现版本。 $B::h()$  和  $A::h()$  的函数原型完全相同，所以具有虚特性，采用动态联编机制。

如前所述，实现 “单接口界面，多实现版本” 的单入口，且呈现多态性的  $do(A \& a)$  函数称之为 “多态函数”。在  $main()$  函数内执行 “ $do(a);$ ” 和 “ $do(b);$ ” 两语句的情况如图 5.4 所示。



图 5.4  $do(a)$  和  $do(b)$  两语句的执行情况

由于  $g()$  不是虚函数，仍实行静态联编。编译系统将实参  $bb$  转换成基类对象的引用，这样一来，在静态联编时  $go()$  函数总是只能访问基类  $A$  中的成员函数  $g()$ ，而不能访问它的派生类  $B$  中的同名成员函数。因为  $A::g()$  和  $B::g()$  作用域不同，不发生函数重载。

#### 5.5.4 虚函数表

使用虚函数引入动态联编技术，使得 `class` 类型注入了很大的灵活性。然而这一灵活性却是有代价的，它增加了执行时间的额外开销，因为系统建立动态联编管理机制增加了程序的执行时间。

C++为每个至少含有一个虚函数的类建立一个与之有关的虚函数表VFT(Virtual Function Table), 该表由一系列函数指针组成。现以例 5.12 为例, 对于基类 A 和公有派生类 B, 编译系统分别为它们建立了一个 VFT (Virtual Function Table, 虚函数表), 如图 5.5 所示。

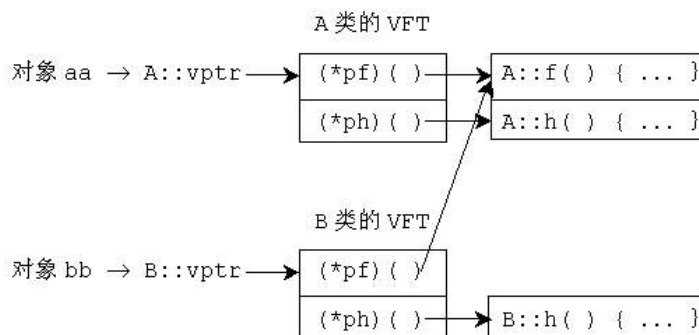


图 5.5 对象 aa 和 bb 的虚函数表与寻找路径

一个类只有一个虚函数表, 不管该类建立了多少个对象。每个类的 VFT 由它的所有对象共享, 如图 5.5 所示, 每个对象都有一个指针 `vp_ptr`, 该指针指向本对象所属类的虚函数表 VFT。

虚函数表是由系统自动生成的, 其内的函数指针也是隐含的, 即由系统自动地定向指向所属 class 类型的对应虚函数, 如图 5.5 所示 A 类 VFT 中的函数指针 `A::(*pf)()` 已由系统自动定向指向了虚函数 `A::f()`, ...。编程者不必编写操作它们的语句。

基类 A 的 VFT 包含两个函数指针 `A::(*pf)()` 和 `A::(*ph)()`, 分别指向两个虚函数 `A::f()` 和 `A::h()`。基类对象 a 含有一个指针 `A::vp_ptr`, 指向 A 类的虚函数表 VFT。

派生类 B 中除了包含有从 A 类继承的虚函数 `A::f()` 外, 还把基类 A 中的虚函数 `h()` 重新定义为 `B::h()`。因此, B 类的 VFT 也包含两个函数指针 `B::(*pf)()` 和 `B::(*ph)()`, 前者指向虚函数 `A::f()`, 另一个指向 `B::h()`。公有派生类 B 的对象 b 含有一个指针 `B::vp_ptr`, 指向 B 类的虚函数表 VFT。

从图 5.5 可知, 非虚函数 `A::g()` 和 `B::g()` 在虚函数表 VFT 中没有入口地址, 即没有函数指针指向它们。

调用虚函数时, C++是通过对象查找到对应 class 类型的虚函数表 VFT 的。上例中在调用多态函数 `do(A & a)` 时, 用基类对象 aa 作为实参 (调用语句为 “`do(aa);`”), 通过对象 aa 的指针 `A::vp_ptr` 寻找到 A 类的 VFT。首先执行 “`a.f();`” 语句, 从 A 类的 VFT 中查到对应的函数指针 `A::(*pf)()`, 从而引导到调用虚函数 `A::f() { ... }`, 如此类推。

用 B 类的对象 bb 调用多态函数 do(A & a) 时, 实际上是按顺序调用了 A::f( )、A::g( ) 和 B::h( ) 等函数, 这是因为:

在执行将实参 bb 传递给形参(A & a)的过程中, 由传递过来的对象 bb 的指针 B::vptr 寻找到 B 类的 VFT。在调用 a.f( ) 函数过程中, 从 B 类的 VFT 查到函数指针 (\*pf)( )。由于 B 类的 f( ) 只是继承 A 类的虚函数 f( ) 而没有重新定义, 因此 B 类的函数指针 B::(\*pf)( ) 是指向 A::f( ), 从而引导到执行调用 A::f( ){ ... } 函数。

由于 g( ) 不是虚函数, 仍采用静态联编机制, 故编译时将对象 bb 的引用隐式转换成 A 类的对象引用, 直接调用 A::g( ){ ... }。

在调用 a.h( ) 函数过程中, 经实参对象 bb 的指针 B::vptr 寻找到 B 类的 VFT, 其内函数指针 B::(\*ph)( ) 指向了 B::h( ), 因此执行 B::h( ){ ... }。

## 5.6 纯虚函数和抽象类

### 5.6.1 纯虚函数

纯虚函数是一种特殊的虚函数, 它在基类中声明为虚函数, 但却没有定义实现部分, 而是在它的各派生类中定义各自的实现版本, 其一般格式为:

```
class 基类名 {
 virtual <类型> 纯虚函数名(参数表) = 0;
 ...
};
```

其中, <类型> 是该纯虚函数返回值的数据类型。由此可见, 它与普通虚函数的声明基本一样, 仅在函数头后加上 “ = 0 ”, 即将该虚函数指明为纯虚函数。

在纯虚函数的所有派生类中都必须定义其实现版本, 否则, 将产生编译错误。例如, 对于一个在平面上表示具有一定形状物体的类 Shape, 把它作为基类可派生出 Circle (圆) 类和 Rectangle (矩形) 类, 如图 5.6、图 5.7 所示。显然, 基类 Shape 所体现的是一个抽象几何形状的概念, 若在它的类体内定义求面积和周长的成员函数 area( ) 和 perimeter( ), 实际上是对虚拟几何形状进行这些操作, 显然这是无意义的, 但是如果把这些成员函数用上述的格式指定为纯虚函数, 那就给 Shape 基类的所有派生类提供了一个公共的接口界面, 而在派生类 Circle 和 Rectangle 的类体内, 定义各自的求面积函数 area( ) 和求周长函数 perimeter( ) 的实现版本, 在基类 Shape 的类体内, 却没有定义这些函数的实现部分, 但要求各派生类必须重定义这些虚函数, 使得派生类中的这些虚函数变得有实际意义。例如, 将如下程序代码做成一个编辑文件名为 “ geometry.h ” 的头文件, 并存放在 d:\bc31\bcuser 目录下, 即 “ d:\bc31\bcuser\geometry.h ”。



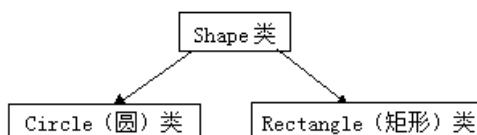


图 5.6 Shape 类及其派生类

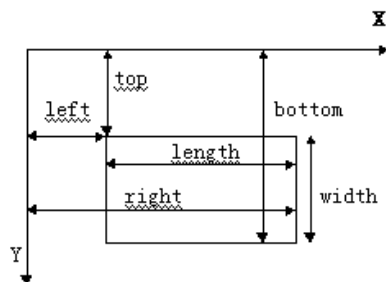


图 5.7 Rectangle 类的绘图参数

```

const float PI = 3.1415926; // 定义圆周率常量 PI
// 定义一个描述抽象几何形状类 Shape，即抽象类
class Shape {
protected:
 float x, y; // 平面几何形状的基点在屏幕上的 x 和 y 坐标的值
 int fillpattern; // 记录 draw() 函数的填充方式
 int color; // 记录 draw() 函数的填充颜色
public:
 Shape(float h = 0, float v = 0, int fill = 0);
 // 抽象类 Shape 构造函数的声明

 float getX(void) const { return x; } // 读取 x 坐标值
 float getY(void) const { return y; } // 读取 y 坐标值

 void setPoint(float h, float v);
 // 设定平面几何形状基点的 x 和 y 坐标值

 int getFill(void) const; // 返回填充方式
 void setFill(int fill) const; // 设置填充方式
 // 纯虚函数，其派生类必须定义各自的求面积和周长的成员函数

 virtual float area(void) const = 0; // 纯虚函数，计算面积
 virtual float perimeter(void) const = 0; // 纯虚函数，计算周长
 virtual void draw(void) const; // 纯虚函数，绘制图形
};

Shape::Shape(float h, float v, int fill) : x(h), y(v), fillpattern(fill)
{ } // 抽象类 Shape 构造函数的实现部分，而函数体为空

// 在 Shape 的派生类 draw() 成员函数进行初始化填充时调用的虚函数
void Shape::draw(void) const
{ setfillstyle(fillpattern, color); }

// 从 Shape 类公有继承的派生类 Circle
class Circle : public Shape {
protected:
 float radius; // 公有派生类 Circle 的新增私有数据成员 radius (圆半径)

```

```

public :
 Circle(float h = 0, float v = 0, float r = 0, int fill = 0);
 // 初始化圆心、半径和填充方式的构造函数

 float getRadius(void) const; // 读取圆半径值
 void setRadius(float r); // 设置圆的半径值
 virtual float area(void) const; // 求圆的面积
 virtual float perimeter(void) const; // 求圆周长
 virtual void draw(void) const;
 // 调用 Shape 类的 draw()初始化填充方式,再调用库函数 circle()画圆
};

// Circle 类的实现部分

Circle::Circle(float h, float v, float r, int fill) : Shape(h, v, fill)
{
 radius = r;
} // 公有派生类 Circle 的构造函数

float Circle::getRadius(void) const { return radius; }
void Circle::setRadius(float r) { radius = r; }
float Circle::perimeter(void) const { return 2 * PI * radius; }
float Circle::area(void) const { return PI * radius * radius; }
void Circle::draw(void) const
{
 Shape::draw();
 circle(x, y, radius);
}
// 调用 Shape 类的 draw()初始化填充方式,再调用 C 语言运行库标准函数 circle()画圆
// 从 Shape 类公有继承的派生类 Rectangle

class Rectangle : public Shape {
protected :
 float length, width;
 // 公有派生类 Rectangle 的新增数据成员 length 和 width, 记录长方形的长和宽
public :
 Rectangle(float h = 0, float v = 0, float l = 0, float w = 0,
 int fill = 0); // 初始化基点、长度和填充方式的构造函数声明

 float getLength(void) const; // 读矩形的长度
 void setLength(float l); // 设置矩形的长度
 float getWidth(void) const; // 读矩形的宽度
 void setWidth(float w); // 设置矩形的宽度
 virtual float area(void) const; // 求矩形的面积
 virtual float perimeter(void) const; // 求矩形的周长
 virtual void draw(void) const;
 /* 调用 Shape 类的 draw()初始化填充方式,再调用 C 语言运行库标准函数 rectangle()
 画矩形 */
};

```

```

// Rectangle 类的实现部分
Rectangle::Rectangle(float h, float v, float l, float w, int fill)
 : Shape(h, v, fill)
{
 length = l; width = w; } // 公有派生类 Rectangle 的构造函数
float Rectangle::getLength(void) const { return length; }
void Rectangle::setLength(float l) { length = l; }
float Rectangle::getWidth(void) const { return width; }
void Rectangle::setWidth(float w) { width = w; }
float Rectangle::area(void) const { return length * width; }
float Rectangle::perimeter(void) const
{
 return 2 * (length + width); }
void Rectangle::draw(void) const
{
 Shape::draw(); // 调用 Shape 类的 draw() 初始化填充方式
 rectangle(x, y, x + length, y + width);
 // 再调用 C 语言运行库标准函数 rectangle() 画矩形
}

```

像 Shape 这样含有纯虚函数的类，称为“抽象类”，即只要至少含有一个纯虚函数的类都叫抽象类，将在下一节详细讨论。

### 5.6.2 抽象类

C++ 提供抽象类机制来描述一般抽象概念性的东西，如上所述，描述平面几何形状的 Shape 类，以它作为基类派生出表示具体形状，如圆、矩形和椭圆等的变种 Circle 类、Rectangle 类和 ellipse 类等才是描述具体实体的类，但抽象类却为它的所有派生类提供了一个统一的公共接口界面，编程者可利用虚函数机制实现“单接口界面、多实现版本”这样灵活多变的功能。例如，Visual C++ 的标准类库 MFC (Microsoft Foundation Class) 中的 CObject 类就是抽象类，它是 MFC 所有标准类的根，即最上层的基类，也可作为用户所开发的面向对象程序系统的根。

由于抽象类是描述一般抽象概念性的东西，因此它只能作为基类，而不能为抽象类创建对象，正因为如此，它不能用来说明函数参数的类型和返回类型，也不能用于显式转换类型，但是，可以为抽象类定义指针变量和引用，以便让它们指向其“子子孙孙”的公有派生类对象。

#### 例 5.13 抽象类 Shape 的应用。

```

#include < graphics.h > // Borland C++ V3.1 的标准图形库的头文件
#include < stdlib.h > // 标准函数 exit() 原型在其中
#include < iostream.h >
#include < conio.h > // 标准函数 gotoxy() 原型在其中

```

```

#include "d:\bc31\bcuser\geometry.h"
// 抽象类 Shape 和其公有派生类 Rectangle、Circle 等均在其中
// 多态函数，形参是抽象类 Shape 的对象指针 p，用它访问虚函数 draw()
void grDraw(Shape * p) { p -> draw(); }

void main(void)
{ Circle cirObj(188, 165, 50); // 创建 Circle 类的一个自动对象 cirObj
 Shape * p1, * p2, * p3 = &cirObj;
 // 定义抽象类 Shape 的 3 个指针 p1、p2 和 p3，并令 p3 指向自动对象 cirObj
 char eol;
 // 存放键盘敲入的字符，变量名 eol 是 End of Line 的缩写
 int left, top, right, bottom; // 见图 5.8
 p1 = new Circle(360, 165, 60);
 // 创建 Circle 类的一个动态对象，并用抽象基类 Shape 的指针 p1 指向它
 p2 = new Rectangle(225, 250, 80, 60);
 // 创建 Rectangle 类的一个动态对象，并用抽象基类 Shape 的指针 p2 指向它
 cout << "Area / Perimeter of CirObj and p1 - p3 : \n";
 cout << "(1) cirObj : " << cirObj.area()
 << " / " << CirObj.Perimeter() << endl;
 // 计算 Circle 类自动对象 CirObj 的面积和圆周长，用对象直接调用成员函数
 cout << "(2) p1 (R60) : " << p1 -> Area()
 << " / " << p1 -> Perimeter() << endl;
 /* 计算 Circle 类动态对象的面积和圆周长，用指向它的抽象基类 Shape 的指针 p1 调用成
 员函数 */
 cout << "(3) p2 (80 * 60): " << p2 -> Area()
 << " / " << p2 -> Perimeter() << endl;
 /* 计算 Rectangle 类动态对象的面积和圆周长，用指向它的抽象基类 Shape 的指针 p2 调用
 成员函数 */
 cout << "(4) p3 (R50): " << p3 -> Area()
 << " / " << p3 -> Perimeter() << endl;
 /* 计算 Circle 类动态对象的面积和圆周长，用指向它的抽象基类 Shape 的指针 p3 调用成
 员函数 */
 cout << "(5) Type <CR> to view figures !\n"; // 敲 CR 键则显示图形
 cin.get(eol);
 // 读取键盘敲入的 CR 键
 int gdriver = DETECT, gmode, errorcode;
 /* 把图形驱动器 gdriver 设置为 DETECT，即自动检测当前系统屏幕显示器硬件的类型，并
 选用分辨率最大的显示模式 */
 initgraph(&gdriver, &gmode, "D:\\BC31\\BGI");

```

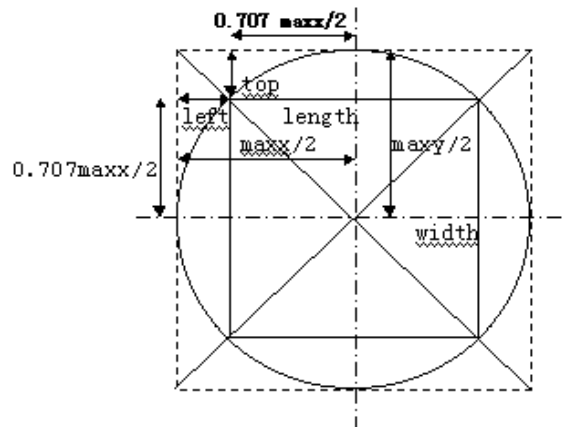
```
/* 调用标准图形库函数 initgraph()初始化图形系统,把图形驱动器代码装入内存,将屏
幕显示器变成图形状态,背景设为黑色 */
```

```
errorcode = graphresult(); //读取初始化的结果
```

```
//检测图形初始化的结果,常量 grOk 表示初始化操作成功
```

```
if(errorcode != grOk) {
 cout << "Graphics error: " << grapherrormsg(errorcode) << endl;
 cout << "Press any key to halt : ";
 getch();
 exit(1); //出错,终止程序,退回到操作系统
}
```

```
/* 在屏幕正中央,以屏幕最大的圆半径作一个圆,以该圆的内接正方形为画图范围,如图 5.8
所示,求出 left、top、right 和 bottom 等,其中标准库函数 getmaxx() 取当前图形
模式下 x 坐标的最大值, getmaxy() 取当前图形模式下 y 坐标最大值 */
```



注:粗线方框为图形区域

图 5.8 例 5.13 的绘图范围

```
left = (1 - 0.707) * getmaxx() / 2;
top = (1 - 0.707) * getmaxy() / 2;
right = 0.707 * getmaxx();
bottom = 0.707 * getmaxy();
setcolor(WHITE);
// 因为是在黑色背景上画图,设定画圆的颜色为白色
grDraw(pl);
// 调用多态函数 grDraw(),画抽象类 Shape 的指针 pl 所指的圆
setfillstyle(XHATCH_FILL, WHITE);
// 设置填充的式样和颜色,用斜网格线和白色填充
floodfill(360, 165, WHITE);
```

```

// 把圆心(360, 165)当种子, 用斜网格线和白色填充 p1 所指的圆
gotoxy(43, 6); // 把字符光标移到第 43 列第 6 行
cout << "p1 (R60)"; // 输出圆的标记字符串"p1 (R60)"

grDraw(p2);
// 调用多态函数 grDraw(), 画抽象类 Shape 的指针 p2 所指的矩形
setfillstyle(LTSLASH_FILL, WHITE);
// 用斜杠阴影线和白色填充

floodfill(240, 260, WHITE);
// 把矩形内的点(240, 260)当种子, 用斜网格线和白色填充 p2 所指的矩形
gotoxy(28, 15); // 把字符光标移到第 28 列第 15 行
cout << "p2 (80 * 60)";
// 输出矩形的标记字符串"p2 (80 * 60)"

grDraw(p3);
// 调用多态函数 grDraw(), 画抽象类 Shape 的指针 p3 所指的圆
setfillstyle(WIDE_DOT_FILL, WHITE); // 用稀疏点和白色填充
floodfill(188, 165, WHITE);
// 把圆心(360, 165)当种子, 用斜网格线和白色填充 p3 所指的圆。
gotoxy(21, 7); // 把字符光标移到第 21 列第 15 行
cout << "p3 (R50)";

long size = imagesize(left, top, right, bottom);
/* 标准库函数 imagesize() 返回值为存储一块屏幕图像所需的存储器字节数, 存放在长整
 型变量 size。若操作失败, 则返回值为-1 */
char * buf = new char[size];
// 在内存堆中开辟 size 个字节的空间作为缓冲器
if(size != -1) // 检测 imagesize() 函数执行是否成功
 if(buf != NULL) { // 判断缓冲器 buf 不为空
 getimage(left, top, right, bottom, buf);
 // 把上面屏幕图形复制到 buf 所指的内存区域
 putimage(left, top, buf, NOT_PUT);
 /* 把 buf 所指的内存区域存放的图形复制到起始位置为 (left, top) 的屏幕上,
 把原来“黑底白字”的图形改成“白底黑字”, 便于打印 */
 }
getch();
closegraph();
/* 撤销 initgraph() 函数所建立的图形状态, 释放用于保存图形驱动器和字体的内存, 退回
 到以前的文本 (Text) 状态 */
}

```

说明:

考虑到 Windows /Visual C++开发平台主要适用于办公自动化 OA ( Office Automation ) , 系统容量较大且执行速度较慢 , 其图形功能的技术思路与 MS-DOS/Borland C++开发平台完全不同 , 后者比较适用于某些工业控制场合的微型、小型系统 , 其图形功能简单、直观 , 且便于结合定量计算来绘制图形 , 所以 , 例 5.13 的程序用后者作为开发平台。第 1 次进入 Borland C++集成开发环境时 , 应首先修改链接环境的设置 , 其操作方法是沿着 “ Options (主菜单)/Linker (第 1 级子菜单)/Libraries (第 2 级子菜单)” 路径选择 , 则将弹出 Libraries 窗口 , 在该窗口内做如下选取即可 :

[X] Graphics library

在 main( ) 函数体内定义了 Circle 类的一个自动对象 cirObj 和一个动态对象 , 并用抽象基类 Shape 的指针 p3 和 p1 分别指向它们 , 还定义了 Rectangle 类的一个动态对象 , 并用抽象基类 Shape 的指针 p2 指向它。在以后的程序中 , 就可以用抽象基类 Shape 的这 3 个指针 p1、p2 和 p3 分别去调用其各派生类的公有成员函数 area( ) 和 perimeter( ) , 求出各图形的面积和周长。但是 , 这是 “ 多接口界面 , 多实现版本 ” 的操作机制。

绘制这 3 个对象的图形却是采用 “ 单接口界面 , 多实现版本 ” 的操作机制 , 为此 , 定义一个多态函数 grDraw( ) , 用抽象基类 Shape 的对象指针 p 作为形参 , 分别用前述的 3 个指针 p1、p2 和 p3 作为实参 , 调用多态函数 grDraw( ) , 按照动态联编操作机制 , 动态选中实参指针所指派生类对象的 draw( ) 成员函数。例如 , 执行 “ grDraw(p1); ” 语句 , 在实参传递给形参的过程中 , 使形参指针 p 指向 p1 所指的派生类 Circle 的圆对象 (360, 165, 60) , 按照动态联编操作机制 , 动态地选中 Circle 类的成员函数 draw( ) , 以点 (360, 165) 为圆心 , 以 60 为半径画圆。而执行 “ grDraw(p2); ” 语句时 , 则动态地选中 Rectangle 类的同名成员函数 draw( ) , 以点 (225, 250) 为左上方基点、80 为宽、60 为高画一个矩形。

由此可见 , 由于抽象类是描述一般抽象概念性的东西 , 因此只能作为其他类的基类 , 但不能创建抽象类的对象 , 也不能把一个函数的形参或者函数的返回值说明成抽象类型 , 但是 , 可以定义抽象类的对象指针和对象引用 , 以便让它们指向其 “ 子子孙孙 ” 的公有派生类对象。如例 5.13 中 , 定义了抽象类 Shape 的 3 个对象指针 p1、p2 和 p3 , 并令它们分别指向抽象类 Shape 的各派生类对象 , 且能利用动态联编机制实现 “ 单接口界面 , 多实现版本 ” 的操作机制 , 该程序的绘图范围和输出显示的图形如图 5.8 和图 5.9 所示。

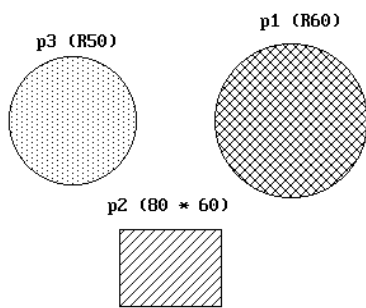


图 5.9 例 5.13 输出显示的图形

## 5.7 虚析构造函数

一个 class 类型所具有的惟一一个析构造函数也可以用关键字 “virtual” 指定为虚函数，称为“虚析构造函数”。给某些类指定虚析构造函数的目的是为了实现在动态联编方式，如在一个类层次结构中，当某层的公有派生类新增数据成员含有指针变量型（如字符串指针变量），且在构造函数体内又有用 new 运算符为其新创建对象的数据成员分配堆中的内存空间，那么在用 delete 运算符撤销该对象时，以确保程序运行时采用动态联编方式，正确选择析构造函数撤销该对象。下面用一个例程来说明虚析构造函数的特点和使用方法。

例 5.14 类层次结构中的虚析构造函数。

```
#include <iostream> // 使用 C++ 新标准的流库
#include <cstring> // 标准函数 strlen() 的声明在其中
using namespace std; // 将 std 标准名空间合并到当前名空间

class Base {
 int a;
 /* 基类 Base 的私有数据成员，记录包含在本基类的公有派生类 Derived 对象中字符串指针
 buf 所指字符串的字符个数（字符串长度） */
public:
 Base(int i) { a = i; } // 基类 Base 的构造函数
 virtual void print(void) // 输出显示基类 Base 的私有数据成员 a 的值
 { cout << "基类的数据成员 a = " << a << endl; }
 virtual ~Base(void) // 基类 Base 的虚析构造函数声明
 { cout << "基类的虚析构造函数 Base:: ~ Base() 被调用 ! \n"; }
 // 输出显示基类的虚析构造函数 Base:: ~ Base() 被调用的信息
};

class Derived : public Base {
 char * buf; // 公有派生类 Derived 的新增私有数据成员 buf
public:
 Derived(int n, char * s) : Base(n)
 { buf = s; // 令 buf 也指向形参字符串指针 s 所指的实参字符串
 if(! s)
 buf = NULL; // 若实参字符串为空串则把 buf 也设置为空串
 else {
 buf = new char[strlen(s) + 1];
 strcpy(buf, s);
 }
 }
 /* 若实参字符串不是空串，用 new 运算符在内存的堆中按其长度开辟空间，再将实参字
 符串复制到该空间内 */
};
```



```

 }
 void print(void)
 {
 Base::print();
 // 调用基类的虚函数 print(), 输出显示基类 Base 的私有数据成员 a 的值
 cout << " 派生类新增数据成员字符串指针 buf 所指的字符串 = " << buf << endl;
 /* 输出显示公有派生类 Derived 的新增私有数据成员 buf 所指的字符串, 即在创建该
 派生类对象时存放在用 new 运算符所开辟的内存空间中的字符串 */
 }
 virtual ~ Derived(void); // 公有派生类 Derived 的虚析构造函数声明
};

Derived::~ ~ Derived(void) // 公有派生类 Derived 的虚析构造函数实现部分
{
 delete [] buf; // 用 delete 运算符撤销 buf 所指的堆 (heap) 空间
 cout << "派生类的虚析构造函数 Derived::~ ~ Derived()被调用 ! \n";
 // 输出显示本虚析构造函数被调用信息
}

void show(Base * p) // 多态函数, 用基类 Base 的对象指针 p 作为形参
{
 p -> print();
}

void death(Base * p)
{
 delete p;
 /* 用 delete 运算符撤销基类 Base 的对象指针 p 所指对象 (可以是基类和公有派生类的对
 象) */
}

void main(void)
{
 Base * bp = new Derived(6, "Hello");
 /* 用 new 运算符创建公有派生类 Derived 的动态对象, 并用基类的自动型对象指针 bp (存
 放在堆栈区) 指向它, 以后可用 bp 去访问该动态对象的所有成员 */

 show(bp);
 /* 用基类 Base 的对象指针 bp 作为实参调用多态函数 show(), 输出显示它所指动态对象
 的数据成员值 */

 death(bp);
 // 用基类 Base 的对象指针 bp 作为实参调用 death(), 撤销 bp 所指动态对象
}

```

该程序的输出结果为：

|                                         |   |                      |
|-----------------------------------------|---|----------------------|
| 基类的数据成员 a = 6                           | } | ( 调用 show() 的输出信息 )  |
| 派生类新增数据成员字符串指针 buf 所指的字符串 = Hello       |   |                      |
| 派生类的虚析构造函数 Derived::~ ~ Derived() 被调用 ! | } | ( 调用 death() 的输出信息 ) |
| 基类的虚析构造函数 Base::~ ~ Base() 被调用 !        |   |                      |

### 虚析构函数的特点

虚析构函数是一种特殊的虚函数，其特殊性表现如下：

虚析构函数既具有虚函数的属性，还具有析构函数的属性，其函数名是在类名前面加一个“~”。因此，基类的虚析构函数在派生类作用域内重新定义（即编写一个派生类的虚析构函数体），当然它仍然是虚析构函数，但是派生类的虚析构函数名就与基类的虚析构函数名不同，换句话说，在类层次结构中各层的虚析构函数是不同的。如例 5.14 中，基类的虚析构函数名为“~ Base”而派生类的虚析构函数名为“~ Derived”。

若一个基类的析构函数被指定为虚析构函数，那么由它继承而来的各层派生类的析构函数就都是虚析构函数，不管是否使用了关键字“virtual”加以说明。如例 5.14 中，派生类 Derived 的虚析构函数头前面的“virtual”是可以省略不写的。

在一个类层次结构中，由于下层的派生类自动继承了所有上层类（包括直接基类和间接基类）的所有成员，若某层的虚析构函数被调用（删除本类新增的数据成员）并执行完成后，下一步将调用其直接基类的虚析构函数，删除从该直接基类继承的数据成员，接着再调用上一层间接基类的虚析构函数，……，依次类推，一直调用到最顶层基类的虚析构函数为止，这样才能完整地删除掉该对象的所有数据成员，以确保删除对象的操作正确地执行。文献[26]所提出的条款 14 明确指出：只要一个类含有至少一个虚函数，则其析构函数必须设置为虚析构函数。

### 2. 虚析构函数的使用方法

在 C++ 中，编程者若想采用动态联编机制，即在一个类层次结构中，基类说明有虚函数，而在派生类作用域内将基类的虚函数重新编写一个新的函数体，那么，基类的析构函数就必须指定为虚析构函数，否则就很可能产生程序隐患，特别在含有动态变量和动态对象的程序中，因使用 delete 运算符删除一个对象时不能确保正确调用析构函数。只有设置了虚析构函数，才能采用动态联编方式正确选中析构函数。

如例 5.14 中，基类 Base 有一个私有数据成员 a 和 3 个成员函数，其中一个构造函数 Derived()、虚析构函数~ Derived()和虚函数 print()。而其公有派生类 Derived 新增一个字符串指针 buf 作为私有数据成员，并将虚函数 print()在派生类作用域内重新编写了一个新的函数体以便于采用动态联编方式。同样，为了在使用 delete 运算符删除一个对象时正确调用析构函数，将基类 Base 的析构函数~ Base()设置为虚析构函数，则派生类的析构函数~ Derived()也为虚析构函数。

为了实现“单接口界面，多实现版本”的程序框架，编写了多态函数 show()，用基类 Base 的对象指针 p 作为形参，并用 p 访问虚函数 print()。当程序运行时根据形参对象指针 p 所具有的地址值 动态选中相应的虚函数 如例 5.14 中 执行‘show(bp);’语句时，实参 bp 虽然是基类 Base 的对象指针，但它指向了派生类 Derived 的一个动态

对象, 在实参传递给形参的过程中相当于执行了 “Base \* p = bp;” 初始化操作, 这样一来, 形参对象指针 p 所具有的地址值就是该派生类 Derived 的动态对象。因此, 在多态函数 show( ) 的函数体内, 执行 “p -> print( );” 语句时动态选中的是调用派生类的虚函数 “Derived::print( )”, 才能够完整地输出显示该动态对象的所有数据成员值, 这由程序的输出结果可以清楚地看出。若不把成员函数 print( ) 说明为虚函数, 则 “p -> print( );” 语句将按静态联编方式 (详见 5.4.3 节), 即在编译时就确定调用基类的成员函数 Base::print( ), 其输出结果如下:

基类的数据成员 a = 6

显然, 由于调用的是基类的成员函数 print( ), 只显示了该动态对象从基类继承而来的私有数据成员 a 的值, 而无法输出显示派生类新增的私有数据成员 — 字符串指针 buf 所指的字符串。

在 main( ) 函数体内, 首先用 new 运算符创建公有派生类 Derived 的动态对象, 并用基类的自动型对象指针 bp (存放在堆栈区) 指向它, 并调用构造函数初始化该动态对象, 在构造函数体内, 当实参字符串指针不是空串时, 又用 new 运算符按实参字符串的长度, 即字符个数加 1 在内存的堆中开辟一个存储空间, 并当成功时把 new 运算符的结果值, 即该内存空间的首地址保存在派生类新增的私有数据成员 — 字符串指针 buf 中, 即字符串指针 buf 指向了实参字符串 “Hello”, 其数据结构图如图 5.10 所示。

在程序运行期间执行到 “death( bp );” 语句时, 基类的对象指针 bp 指向派生

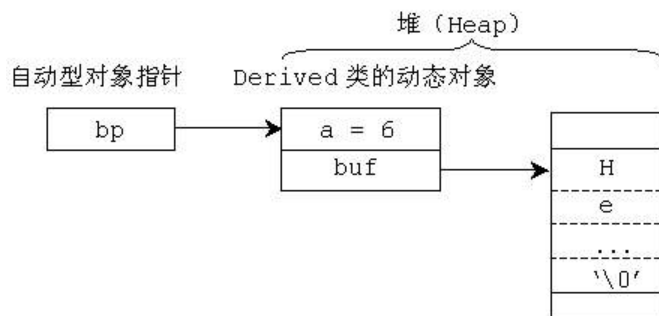


图 5.10 Derived 类动态对象的数据结构图

类 Derived 的动态对象, 当然动态选中调用派生类的虚析构造函数 ~ Derived( ), 在其函数体内, 用 delete 运算符删除了该对象的数据成员 buf 所指的字符串 “Hello”, 将堆中的这片空间释放掉由系统回收以备他用, 再输出显示如下信息:

派生类的析析构造函数 Derived:: ~ Derived() 被调用 !

当派生类的虚析析构造函数 ~ Derived( ) 执行完后, 接着就调用基类 Base 的虚析析构造函数, 从基类继承的数据成员 a 是由系统自动删除的, 不需要编程者写任何程序, 而调用基类 Base 的虚析析构造函数输出本成员函数已被调用的如下信息:

基类的虚析构函数 `Base::~Base()` 被调用！

这充分说明了上面虚析构函数特点中之所述的情况。

如果不把该类层次结构中基类 `Base` 的析构函数设置为虚函数即虚析构函数，则不可能采用动态联编方式而使用静态联编方式，此时不是在程序运行时而是在编译时就确定只调用基类 `Base` 的析构函数，自动地删除了本派生类 `Derived` 动态对象的从基类继承而来的私有数据成员 `a`，然后输出显示“基类的虚析构函数 `Base::~Base()` 被调用！”的字符串信息。但是，如图 5.10 所示，该动态对象新增的私有数据成员——字符串指针 `buf` 以及它所指的字符串“Hello”在堆中所占用的空间却没有被释放，这是一个关于内存管理上的、十分严重的程序瑕疵，因为它会导致不可预计的严重后果，如系统崩溃或将硬盘重新格式化。所以，在一个类层次结构中，总是把基类的析构函数设置成虚函数即虚析构函数。

## 小 结

C++ 的多态性表现在它为编程者提供了运算符重载，函数名重载和虚函数等运行机制。运算符重载和函数名重载采用静态联编机制，而虚函数采用动态联编机制。

运算符重载是把 C++ 本身提供的标准运算符重新在类中定义，使标准运算符可作用于用户新定义的类对象，关键是在理解运算符表达式操作含义的基础上，定义一个运算符重载函数，并将运算符表达式转换成运算符重载函数调用的形式。

在派生类作用域内允许重新定义基类的成员函数，即基类和派生类具有同名的成员函数，因类作用域不同，不会发生重载，而是在派生类作用域内派生类成员函数掩盖了基类的同名成员函数，调用它时必须采用“基类名::成员函数名(实参表)”的形式。

基类的对象指针和对象引用是联系其派生类消息通路的桥梁，但要实现“单接口界面，多实现版本”的操作机制，必须引用虚函数实现动态联编机制。

虚函数用于公有继承的类层次结构中，若把其基类的某成员函数指定为虚函数，则它可在一个或多个公有派生类中被重新定义，但在派生类中重新定义时，该同名虚函数的原型，其中包括返回类型、参数的个数和类型以及它们的排列顺序等，都必须完全相同才能实现动态联编。

纯虚函数是没有函数体（即没有定义函数的实现部分）的一个基类虚函数，在其函数头后面加上“= 0”标记，该基类的所有派生类都必须为每个纯虚函数提供一个它的实现版本。含有一个纯虚函数的类称为抽象类，用以描述抽象概念性的事物。

将一个类的析构函数设置为虚函数即虚析构函数以便采用动态联编机制。在一个类层次结构中，只要一个类含有至少一个虚函数，则其析构函数必须设置为虚析构函数。

## 习 题 5

### 一、选择填空

1. 对定义重载函数的下列要求中, ( ) 是错误的。
  - A. 要求参数的个数不同
  - B. 要求参数中至少有一个类型不同
  - C. 要求参数个数相同时, 参数类型不同
  - D. 要求函数的返回值不同
2. 下列函数中, ( ) 不能重载。
  - A. 成员函数
  - B. 非成员函数
  - C. 析构函数
  - D. 构造函数
3. 下列对重载函数的描述中, ( ) 是错误的。
  - A. 重载函数中不允许使用缺省参数
  - B. 重载函数中编译系统根据参数表进行选择
  - C. 不要使用重载函数来描述毫不相干的函数
  - D. 构造函数重载将会给初始化带来多种方式
4. 下列运算符中, ( ) 不能重载。
  - A. &&
  - B. [ ]
  - C. ::
  - D. new
5. 下列关于运算符重载的描述中, ( ) 是正确的。
  - A. 运算符重载可以改变运算量的个数
  - B. 运算符重载可以改变优先级
  - C. 运算符重载可以改变结合规则
  - D. 运算符重载不可以改变语法结构
6. 运算符重载函数是 ( )。
  - A. 成员函数
  - B. 友元函数
  - C. 内联函数
  - D. 带缺省参数的函数
7. 已知 Myclass 类成功地重载了 --、=、+ 和 [ ] 等几个运算符, 那么, 其中肯定重载为成员函数的运算符函数是 ( )。
  - A. + 和 =
  - B. [ ] 和后置 --
  - C. = 和 [ ]
  - D. 前置 -- 和 [ ]
8. 下列成对的表达式中, 运算符 “/” 的含义相同的一对是 ( )。
  - A. 8 / 3 和 8.0 / 3.0
  - B. 8 / 3.0 和 8 / 3
  - C. 8.0 / 3 和 8 / 3
  - D. 8.0 / 3.0 和 8.0 / 3
9. 在一个重载运算符函数的参数表中, 若没有任何参数, 则说明该重载运算符函数是 ( )。
  - A. 单目运算符非成员函数
  - B. 单目运算符成员函数
  - C. 双目运算符非成员函数
  - D. 双目运算符成员函数
10. 若将表达式 “--x / y” 中的 “--” 和 “/” 运算符都重载成友元函数, 则转换成重载运算符函数的调用格式可写成 ( )。
  - A. operator / ( x.operator --( ) , y )
  - B. operator / ( operator --( x ) , y )

- C. `x.operator --( ).operator / ( y )`  
 D. `y.operator / ( operator --( x ) )`
11. 若将表达式“`y * x ++`”中的“`*`”运算符重载为成员函数,“`++`”运算符重载成友元函数,则转换成重载运算符函数的调用格式可写成 (      )  
 A. `x.operator ++( 0 ).operator *( y )`  
 B. `operator *( x.operator ++( 0 ) , y )`  
 C. `y.operator *( operator ++ ( x , 0 ) )`  
 D. `operator * ( operator ++( x , 0 ) , y )`
12. 已知 `Fraction` 类有一个带单个 `double` 型参数的构造函数,且把“`-`”运算符重载成友元函数,使下列语句序列:
- ```
Fraction  x( 3.2 ) , y( 5.5 ) , z( 0.0 );
z = 8.9 - y;
y = x - 6.3;
```
- 能正常运行,重载运算符函数 `operator -()` 应在类体中声明为 ()
 A. `friend Fraction operator -(Fraction & , Fraction &);`
 B. `friend Fraction operator -(Fraction , Fraction);`
 C. `friend Fraction operator -(Fraction , Fraction &);`
 D. `friend Fraction operator -(Fraction & , Fraction);`
13. 下列关于动态联编的描述中, () 是错误的。
 A. 动态联编是以虚函数为基础的
 B. 动态联编在程序运行时确定所调用的函数代码块
 C. 动态联编调用多态函数时传递给它的是基类对象的指针或基类对象的引用
 D. 动态联编是在编译时确定调用某个函数的
14. 下列关于虚函数的描述中, () 是正确的。
 A. 虚函数是一个 `static` 类型的成员函数
 B. 虚函数是一个非成员函数
 C. 基类中说明了虚函数后,其派生类中的对应函数可不必重新再说明
 D. 基类中的虚函数和其派生类中的虚函数具有不同的参数个数和类型
15. 用虚函数只有在 () 和 () 时才能实现多态性。
 A. 基类和派生类具有同名虚函数,而它们的参数个数不同
 B. 用指向基类指针或基类的对象引用访问虚函数
 C. 基类和派生类具有同名虚函数,而它们的参数至少有一个类型不同
 D. 基类和派生类具有函数原型完全相同的同名虚函数,而函数体内的执行代码不同
16. 下列关于纯虚函数和抽象类的描述中, () 是错误的。
 A. 纯虚函数是一种特殊的虚函数,它没有具体的实现部分

- B. 抽象类是指具有纯虚函数的类
 - C. 一个基类中说明有纯虚函数, 该基类的派生类一定不再是抽象类
 - D. 抽象类只能作为基类来使用, 其纯虚函数的实现部分由派生类给出
17. 下列描述中, () 是抽象类的特性。
- A. 可以说明虚函数
 - B. 可以进行构造函数重载
 - C. 可以定义友元函数
 - D. 不能说明其对象
18. 虚函数必须是类的 ()。
- A. 成员函数
 - B. 友元函数
 - C. 构造函数
 - D. 析构函数
19. 关于虚函数, 下面的叙述 () 是正确的。
- A. 若在派生类作用域内重定义基类的虚函数时使用了关键字 `virtual`, 则该重定义函数仍然是虚函数
 - B. 虚函数不得声明为静态函数
 - C. 虚函数不得声明为另一个类的友元函数
 - D. 派生类必须重新定义基类的虚函数
20. 关于纯虚函数, 下面的叙述 () 是正确的。
- A. 纯虚函数是没有编写实现版本 (即无函数体的定义) 的虚函数
 - B. 纯虚函数的声明总是以 “`= 0;`” 结束
 - C. 派生类必须实现基类的纯虚函数
 - D. 含有纯虚函数的类不可能是派生类
21. 多态调用是指 ()。
- A. 以任何方式调用一个虚函数
 - B. 以任何方式调用一个纯虚函数
 - C. 借助于指向对象的基类指针或对象引调用一个虚函数
 - D. 借助于指向对象的基类指针或对象引调用一个纯虚函数
22. 关于抽象类, 下列叙述中 () 是正确的。
- A. 抽象类的成员函数中至少有一个是没有实现版本的函数 (即无函数体定义的函数)
 - B. 一个派生类从抽象类继承而来, 它必须实现该抽象类中的纯虚函数
 - C. 派生类不可能成为抽象类
 - D. 抽象类不能用来定义对象
23. 试有如下程序:

```
#include <iostream>
using namespace std;
class Base {
```

```

    char    c;
public:
    Base( char  a ) : c( a ) { }
    virtual ~ Base( void )
    {    cout << c;    }
};
class  Derived : public Base {
    char    c;
public :
    Derived( char  a ) : Base( a + 1 ), c( a ) { }
    ~ Derived( void )
    {    cout << c;    }
};
void main( void )
{    Derived    d( 'X' );    }

```

执行该程序的输出结果是 ()。

- A. XY B. YX C. X D. Y

24. 试有如下程序：

```

#include    < iostream >
using  namespace  std;
class  AA  {
public:
    virtual  void  fun( void )
    {    cout << "AA";    }
};
class  BB : public  AA    {
public :
    BB( void )
    {    cout << "BB";    }
};
class  CC : public  BB  {
public:
    virtual  void  fun( void )
    {    BB::fun( );
        cout << "CC";    }
};
void  main( void )
{    AA    aa,  * p;

```



```
BB      bb;
CC      cc;

p = & cc;

p -> fun( );

}
```

执行该程序的输出结果是 ()。

- A. BBAACC B. AABBC C. BBAABBC D. BBBBAACC

25. 试有如下程序：

```
#include <iostream>
using namespace std;
class XX {
protected:
    int k;
public:
    XX( int a = 5 ) : k( a ) { }
    ~XX( void ) { cout << "XX"; }
    virtual void fun( void ) const = 0;
};

inline void XX::fun( void ) const { cout << k + 3; }

class YY : public XX {
public:
    ~YY( void )
    { cout << "YY"; }
    void fun( void ) const
    { cout << k - 3;
      XX::fun( );
    }
};

int main( void )
{
    XX & ref = * new YY;
    ref.fun( );
    delete & ref;
    return 0;
}
```

执行该程序的输出结果是 ()。

- A. 28XX B. 28YYXX C. -33XX D. -33XXYY

二、判断下列描述的正确性，对者划 ☐ ，错者划 ☒

1. 函数的参数个数和类型都相同，只是返回值不同，这不是重载函数。
2. 重载函数可以带有缺省值参数，但是要注意二义性。
3. 多数运算符可以重载，个别运算符不能重载，运算符重载是通过函数定义实现的。
4. 对每个可重载的运算符来讲，它既可以重载为友元函数，又可以重载为成员函数，还可以重载为非成员函数。
5. 对单目运算符重载为友元函数，应说明一个形参。重载为成员函数时，不能显式说明形参。
6. 重载运算符保持原运算符的优先级和结合性不变。
7. 虚函数是用 `virtual` 关键字说明的成员函数。
8. 构造函数说明为纯虚函数是没有意义的。
9. 抽象类是指没有定义函数体的基类虚函数。
10. 动态联编是在运行时选定调用的成员函数。

三、填空题

1. 重载运算符仍然保持其原来的运算量（或称操作数）的个数、优先级和_____不变。
2. C++中构成重载运算符函数名的关键字是_____。
3. 若把表达式“`x = y * z`”中的“`*`”运算符重载为成员函数，则转换成重载运算符函数的调用格式可写成_____。
4. 表达式“`x = operator - (y , z)`”还可以写成_____。
5. 若表达式“`--x`”中的“`--`”运算符重载为成员函数，则转换成重载运算符函数的调用格式可写成_____。
6. 表达式“`operator ++ (x , int)`”还可以写成_____。
7. 一个 `Complex`（复数）类的定义如下，其中“`--`”运算符重载成友元函数，其功能是将形参对象的实数部分减 1 后该对象的引用，请把空白处补充完整。

```
class Complex {
    int real, imag;
public:
    Complex( int r = 0 , int i = 0 ) : real(r), imag(i) { }
    void show( void )
    {   cout << real << ( imag < 0 ? "-" : "+" ) << imag << "i\n" ;   }
    friend Complex & operator -- ( Complex & );
};

Complex & operator -- ( Complex & c )
{   c.real --;
```

```
    return  c;
}
```

8. Vector2D (二维向量) 类的定义如下, 其中把“+”运算符重载为成员函数, 其功能是将两个向量的分量 x 和 y 对应相加, 然后, 返回到作为相加结果的新对象, 请把空白处补充完整。

```
class Vector2D {
    double  x, y;
public:
    Vector2D( double  x0 = 0 , double  y0 = 0 ) : x(x0), y(y0) { }
    void  show( void )
    {   cout << "( " << x << " , " << y << " )" << endl;   }
    Vector2D  operator + ( Vector2D );
};

Vector2D  _____operator + ( Vector2D  v )
{   return  Vector2D( _____ );   }
```

9. 多态性可分为_____和_____两类。
10. 对虚函数调用有_____和_____两种方式。
11. 实现编译时的多态性机制称为_____, 实现程序运行时的多态性机制称为_____。
12. 在 C++ 中, 编译时的多态性是通过_____和模板体现的。
13. 在 C++ 中, 程序运行时的多态性是通过_____体现的。
14. 有一种特殊的虚函数, 重新定义时不要求同名, 这种虚函数是_____。
15. 含有纯虚函数的类称为_____。
16. 一个纯虚函数声明中最后 3 个字符是_____。

四、分析下列程序的输出结果

程序 1:

```
#include      < iostream >
using namespace  std;
class  B  {
    int  b;
public :
    B( int  i ) { b = i + 50;          show( ); }
    B( void ) { }
    virtual void show( void )
    { cout << "B::show( ) called. " << b << endl; }
```

```

};
class D : public B {
protected :
    int d;
public :
    D( int i ) : B(i) { d = i + 100; show( ); }
    D( void ) { }
    void show( void ) { cout << "D::show( ) called. " << d << endl; }
};
void main( void ) { D d1(108); }

```

程序 2 :

```

#include <iostream>
using namespace std;
class B {
    int b;
public :
    B( void ) { }
    B( int i ) { b = i; }
    virtual void virfun( void ) { cout << "B::virfun( ) called.\n "; }
};
class D : public B {
    int d;
    void virfun( void ) { cout << "D::virfun( ) called.\n "; }
public :
    D( void ) { }
    D( int i, int j ) : B(i) { d = j; }
    void show( void ) { cout << "D::show( ) called. " << d << endl; }
};
void fun( B * objp ) { objp -> virfun( ); }
void main( void ) { D * pd = new D; fun(pd); }

```

程序 3 :

```

#include <iostream>
using namespace std;
class A {
public :
    A( void ) { ver = 'A'; }

```

```
        void print( void ) { cout << "The A version " << ver << endl; }
protected :
    char  ver;
};
class  D1 : public  A  {
    int  info;
public :
    D1( int number ) { info = number;  ver = '1'; }
    void print( void )
    { cout << "The D1 info : " << info << " version " << ver << endl;  }
};
class  D2 : public  A  {
    int  info;
public :
    D2(int number)  { info = number; }
    void print( void )
    { cout << "The D2 info : " << info << " version " << ver << endl;  }
};
class  D3 : public  D1  {
    int  info;
public :
    D3( int  number ) : D1( number ) { info = number;  ver = '3'; }
    void print( void )
    { cout << "The D3 info : " << info << " version " << ver << endl;  }
};
void  print_info( A * p ) {  p -> print( );  }
void  main( void )
{  A  a;      D1  d1(4);   D2  d2(100);   D3  d3(-25);
    print_info(&a);        print_info(&d1);
    print_info(&d2);        print_info(&d3);
}
```

程序 4 :

```
#include      < iostream >
using namespace  std;
class  Matrix  {
    double * elem;
    int  row, col;
```

```

public :
    Matrix( int r, int c )
    {   row = r;   col = c;   elem = new double[row * col];   }

    double & operator( void ) ( int x, int y )
    {   return elem[col * (x - 1) + y -1];   }

    double operator( void ) (int x, int y) const
    {   return elem[col * (x - 1) + y -1];   }

    ~ Matrix( void ) { delete [ ] elem;   }

};

void main( void )
{   Matrix    m(5, 8);
    for( int i = 1; i < 6; i++ )
        m( i, 1 ) = i + 5;
    for( i = 1; i < 6; i++ )
        cout << m(i, 1) << " , " ;
    cout << endl;
}

```

五、编程题

1. 整理并扩充例 5.4 中的 Fraction (分数) 类，完善你认为尚缺的重载运算符函数并加以实现。
2. 整理并扩充习题 5 第三大题第 8 小题定义的 Vector2D (二维向量) 类，使它能完成向量加法、减法、内积、求向量长度和 C++ 流的 “<<” 输出运算符以及 “>>” 输入运算符等操作。
3. 定义如下两个类：

```

class Fruit {
public :
    virtual char * identify( ) { return "Fruit"; }
};

class Tree {
public :
    virtual char * identify( ) { return "Tree"; }
};

```

而 Apple 类既从 Fruit 类，又从 Tree 类派生而来，即

```
class Apple : public Fruit, public Tree { ... }
```

又有既从 Fruit 类，又从 Tree 类派生而来的 Pear 类，即

```
class Pear : public Fruit, public Tree { ... }
```

每个派生类至少有一个成员函数可实现在 CRT 上显示 “自己的类名及从哪些基类派生而来” 的信息，例如 Apple 类应该显示如下信息：(Apple : Fruit , Tree)。现有杂交品种 Apple_Pear(苹果梨)类，它从 Apple 类和 Pear 类继承而来，其类体内至少有一个成员函数可在 CRT 上实现显示如下信息的功能：

```
(Apple_Pear : (Apple : Fruit , Tree), (Pear : Fruit , Tree)).
```

试编写一个完整的可运行源程序完成上述功能，其中还应包括能测试这些功能的 main() 函数。

4. VeryLong 类的定义如下，它能表示任意大小整数：

```
class    VeryLong    {  
    ...  
public :  
    VeryLong( int      = 0 );  
    VeryLong( long     );  
    VeryLong( const char * );  
    // 形参是字符串指针，对应的实参为字符串，如：“98347712498863498”  
    VeryLong( VeryLong & );  
    ~ VeryLong( void );  
    ...  
};
```

请把该类的定义补充完整并编写所有成员函数的实现部分，使它能完成+、-、*、/、++、--、=、单目+、单目-和 C++流的“<<”输出运算符以及“>>”输入运算符等操作。

5. 编写 Date (日期) 类及其成员函数，使它至少具有如下特性：

创建能指定具体日期 (指定年、月、日) 的 Date 类对象，默认日期为 1949.10.1；
可以从一个输入流例如键盘读取一个日期保存在一个 Date 类对象，日期的输入格式为：年.月.日，其中年份可以是 4 位十进制数字，也可以是 2 位十进制数字 (即可以省略掉前两位)，月和日可以是 2 位十进制数字，也可以是 1 位十进制数字；
能通过一个输出流 (如终端显示器) 输出一个 Date (日期) 类对象，格式与输入相同；

能用==、!=、<、<=、>和>=等运算符对两个 Date (日期) 类对象进行关系运算；
能用前置或后置的增量和减量运算符，以及“+=”和“-=”运算符完成增加或减少一天，或者若干天的操作。

能用+和-把一个 Date (日期) 类对象与一个表示天数的整数相加减，其结果是一个表示若干天前或若干天后的日期，用一个 Date (日期) 类对象保存该日期。

Date (日期) 类必须正确表示日期，不会出现 13 月 2 日或 5 月 45 日超出月份和日期范围的错误情况。Date (日期) 类还能正确处理闰年问题，即所有能被 400 整除的年

份，以及能被 4 整除并同时不能被 100 整除的年份。

6. Clock 类以及 Hour (时) Minute (分) 和 Second (秒) 等结构类型的说明如下：

```
class Clock {
    int hh, mm, ss;
public:
    Clock( int h = 0, int m = 0, int s = 0 ) : hh(h), mm(m), ss(s) { }
    SetClock( int h = 0, int m = 0, int s = 0 );
    void show( void );
    ...
};

struct Hour {
    int hr;
    Hour( int n = 1 ) : hr(n) { }
};

struct Minute {
    int mt;
    Minute( int n = 1 ) : mt(n) { }
};

struct Second {
    int sd;
    Second( int n = 1 ) : sd(n) { }
};
```

其中 hh、mm 和 ss 分别表示小时 (最大值为 23), 分钟 (最大值为 59), 秒 (最大值为 59), 成员函数 SetClock() 把时钟调到指定的钟点, 成员函数 show() 的功能是按 “时:分:秒” 的格式显示当前时钟的时间。请重载 -, +, ++, --, += 和 -= 等运算符, 以实现时钟的运转。例如, 假定 clk 是 Clock 类的一个对象, 即它表示一个时钟, 那么, 执行 “clk += Minute(3);” 语句, 则使时钟 clk 前进 3 分钟。“clk - Hour(2) - Minute(30)” 表达式的结果值也是 Clock 类的一个对象即一个时钟的时间, 表示时钟 clk 两个半小时以前的时间。注意: 时间到 24 点自动复位为 0 点。此外, 还要求重载 ==, !=, >, >=, < 和 <= 等关系运算符以实现两个时钟的比较, 请编写一个满足上述要求的 Clock 类, 并用测试程序 (即主函数) 测试它。

7. 试将例 5.8 静态联编机制的源程序改写为动态联编机制, 可通过多态函数 void fun(Point&s) 这一 “单接口界面” 实现计算 Rectangle 类的对象 rec(3.0, 5.2, 15.0, 25.0) 和 Circle 类的对象 c(9.0, 15.6, 4.8) 面积, 写成一个完整的可运行源程序。

第 6 章 模 板

C++还实现了一种新的称之为“模板”的运行机制，它能重复利用已经开发好的数据结构和算法模块，经过挑选再组装到新的软件中，从而大幅度地节省了开发所需的人力和时间，大大地提高了开发效率。因此，模板特别适合于大型软件的开发。本章着重介绍有关模板的概念、定义方法和使用方法等，以便读者尽快地掌握模板这一强有力的工具，并为进一步学习流行的 C++语言系统中功能强大的标准模板类库打下牢固的基础，例如，Visual C++所配有的 ATL(Active Template Library，活动模板类库)。

6.1 函 数 模 板

6.1.1 引入函数模板

众所周知，C++的源程序由函数和类组成，所以模板也分为函数模板 (Function Template) 和类模板 (Class Template)。

1. 函数模板的有关概念

首先来考查在第 2 章讨论过的 swap() 函数，它的功能是对两个数据进行交换，如果这两个数据都是整型数，则该函数可写成：

```
void swap( int & x, int & y )
{
    int temp = x;
    x = y;
    y = temp;
}
```

若这两个数据都是实型数，则写成：

```
void swap( double & x, double & y )
{
    double temp = x;
    x = y;
    y = temp;
}
```

如果交换的是两个同一 class 类型 T 的对象，则有：

```
void swap( T & a, T & b )
{
    T temp = a;
    a = b;
    b = temp;
}
```

由此可见，为了把 `swap()` 函数用于不同的数据类型，就需要定义一系列的重载函数。如果把这一类函数抽象成一个模板，将数据类型作为它的一个参数，就构成了一个函数模板，则有：

```
template <class T> 或写    template <typename T>    // 模板的声明
void swap( T & a, T & b )
{
    T temp = a;
    a = b;
    b = temp;
}
```

其中，`class T` 叫做“模板参数”，它既可以是基本数据类型，也可以是标准类库中或者用户自行定义的 `class` 类型。必须指出，这里声明的 `swap()` 函数并不是一个信息完整、实际可用的函数，它代表的是一类函数，是这类函数的“样板”，称为“函数模板”，因为它描述了这类函数的基本情况，例如，它是一个无返回值的函数，具有两个形式参数，其基本操作内容是交换两个 `T` 类型的数据，并引入了一个类型参数 `T`，是需要实例化的，用以补充必要的传递信息。因此，要使用该函数模板，还必须把数据模板参数 `T` 实例化，才能完成具体的函数功能，这是用函数调用语句中的实参来实例化模板参数 `T`。如果实参是 `int` 型，即：

```
int i, j;
swap( i , j );
```

则模板参数 `T` 实例化为 `int` 型，函数模板 `swap` 实例化为仅用于 `int` 型数据进行交换操作的模板函数。若实参是 `double` 型，即：

```
double x , y;
swap( x , y );
```

则模板参数 `T` 实例化为 `double` 型，函数模板 `swap` 实例化为仅用于 `double` 型数据进行交换操作的模板函数。若实参是一个 `Point` 类的对象，则 `Point` 类的定义存放在头文件 `Point.h` 中，其源程序代码如下：

```
class Point {
    int x , y;        // 两个私有数据成员 x 和 y 保存 Point 类点对象的 x 和 y 坐标值
public :
    Point( void ) { x = 0; y = 0; }           // 无参数的构造函数
    Point(int i , int j) { x = i; y = j; }    // 一般构造函数
    int  readX(void) { return x; }           // 读取点对象的 x 坐标值
    int  readY(void) { return y; }           // 读取点对象的 y 坐标值
    void move(int  xOffset , int  yOffset);  // 移动点对象
    friend int  operator > ( Point & a, Point & b );
```

```

// 重载 “大于关系” 运算符函数
friend ostream & operator << ( ostream & os , Point & rp );
// 重载输出运算符<<函数

};

void Point::move( int xOffset , int yOffset )
{   x += xOffset; y += yOffset;   }

ostream & operator << ( ostream & os , Point & rp )
{   cout << "P(" << rp.x << " , " << rp.y << ")";
    // 以 “P(X , Y)” 的形式输出显示点对象的坐标值
    return os;
}

```

则把模板参数 T 实例化为 `Point` 类型, 即:

```

Point p1(40, 60), p2(128, 84);
swap( p1 , p2 );

```

如图 6.1 所示, 通过模板参数 T 的实例化也就把函数模板 `swap` 实例化成为能完成具体函数功能的模板函数 `swap(i, j)`、`swap(x, y)` 和 `swap(p1, p2)` 等。而 `swap(p1, p2)` 是将两个点 `p1` 和 `p2` 的 x 、 y 坐标值进行交换, 通常, 把对模板参数 T 进行实例化的参数称为“模板参数”。

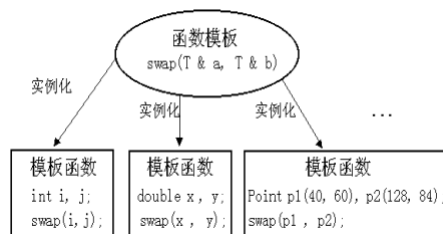


图 6.1 函数模板实例化成模板函数

2. 引入函数模板的原因

又如返回两个参数中较大者的函数 `max(x, y)`, C++ 的早期版本是采用宏定义语句实现的, 即

```
#define max(x, y) ((x > y) ? x : y)
```

然而, 由于 `#define` 语句只能进行字符串的替换, 而不能检查参数类型的合法性, 例如, 可能会将一个 `int` 型参数和一个 `Point` 类型的参数进行交换而导致程序故障。这就是为什么要引入函数模板的原因之一。即:

```

template <class T>
T & max(T & x, T & y)
{   return x > y ? x : y;   }

```

例 6.1 用模板技术把多个重载函数归纳为一个函数模板。

```

#include <iostream.h>
#include "Point.h"
// 函数模板 swap 的定义
template <class T>
void swap( T & a, T & b )

```

```

{   T temp = a;
    a = b;
    b = temp;
}

void main( void )
{   int      i = 24, j = 78;          // 定义两个 int 型自动变量 i 和 j
    double   x = 72.9, y = 48.6;      // 定义两个 double 型自动变量 x 和 y
    Point    p1(40, 60), p2(128, 84); // 创建两个 Point 类自动对象 p1 和 p2
    swap( i, j );                     // 实例化为用于 int 型数据的模板函数

    cout << "(1)After called swap( ) : i = " << i << " , j = " << j << endl;
    swap(x, y); // 实例化为用于 double 型数据的模板函数

    cout << "(2)After called swap( ) : x = " << x << " , y = " << y << endl;
    swap(p1, p2); // 实例化为用于 Point 类对象的模板函数

    cout << "(3)After called swap( ) : p1 = " << p1 << " , p2 = " << p2 << endl;
}

```

该程序的输出结果为：

```

(1) After called swap( ) : i = 78 , j = 24
(2) After called swap( ) : x = 48.6 , y = 72.9
(3) After called swap( ) : p1 = P(128 , 84) , p2 = P(40 , 60)

```

6.1.2 函数模板的定义

把那些针对不同数据类型而功能相同的一类函数抽象成一个函数模板,其定义格式为：

```

template <class T1 , class T2 , ... , class Tn>
// 即 template < 模板形参表 >, 下同
<返回类型> 函数名( 形式参数表 )
{
    ...                               // 函数体
}

```

其中 `template <class T1 , class T2 , ... , class Tn>` 是向编译系统声明“定义了一个模板”的声明语句。紧跟在关键字 `template` 后面,用尖括号包围的是模板形式参数表,简称“模板形参表”。它由一个或者多个用逗号隔开的模板形式参数组成,其内的每个参数前面都用关键字 `class` 来表示,标识符 `Ti` 是一个特定的 C++ 数据类型,它们可以是各种基本数据类型,也可以是标准类库或者用户自行定义的 `class` 类型。顺便指出,模板声明还有另外一种完全等价的形式,用“`typename`”取代关键字“`class`”,即：

```

template < typename T1 , ... , typename Tn>
// 即 template < 模板形参表 >
<返回类型> 函数名( 形式参数表 )
{
    ...                // 函数体
}

```

因此，模板形参具有如下三种形式：

```

typename <参数名>;
class <参数名>;
<数据类型说明> <参数名>。

```

其中，<参数名>是由编程者自行启用的标识符。第 种与第 种形式是完全等价的，有时可以互相替换。但是，必须强调指出，“typename”不能写在模板声明语句以外，否则将产生 C2899 的编译错误。因此，程序中所有的“typename”都可以替换成“class”，但反之程序中的所有“class”是不可能都替换成“typename”。这两种形式所声明的参数是虚拟类型参数，必须获取模板实参的信息才能确定自身的数据类型。这是通过函数模板的实例化过程才完成的，如前所述，函数模板只有实例化成模板函数，并生成了对应的函数调用版本的程序代码，这是由编译系统根据模板函数的调用语句中实际所使用的数据类型，更具体地说，即调用模板函数的实参表中实参所采用的数据类型。如例 6.1 中，编译系统根据函数模板的定义和如下两条语句：

```

int i = 24, j = 78;

...

swap( i, j );

```

对函数模板 swap(<T>) 进行实例化，生成如下模板函数 swap(int & a, int & b) 的程序代码：

```

void swap( int & a, int & b )
{
    int temp = a;
    a = b;
    b = temp;
}

```

而在处理这样两条语句时：

```

Point    p1(40, 60), p2(128, 84);

...

swap(p1, p2);

```

则生成如下的函数定义源程序代码：

```

void swap( Point & a, Point & b )

```

```

{   Point temp = a;
    a = b;
    b = temp;
}

```

接着，将函数定义的源程序代码翻译成本地机的机器码，并将其函数体代码转换成一个由本地机的机器码构成的函数体代码模块。因此，函数模板的每一次实例化就是一个函数定义，也就生成了对应的函数实现版本（即函数体代码模块），它也就是前述的模板函数。随后调用该模板函数的过程就与普通函数一样了。总之，在实例化过程中，模板实参的信息优先于函数实参的信息，即先根据模板实参来确定虚拟类型的数据类型，若信息不够再由函数实参来确定。然而，这些操作过程都是由系统自动完成而用户既看不见也感觉不到，理解了这一操作过程可帮助读者理解和记忆编写“函数模板”程序代码的规则和技巧。

第 一种形式所声明的参数是“常规参数”，即用像 `int`、`float` 和 `double` 等数据类型声明的参数。当函数模板实例化为一个模板函数后，这类“常规参数”也无法从其模板函数的实参表中获得具体的实参值，这与普通函数的形式参数类似，下面再用一个例程加以说明。

例 6.2 含“常规参数”的函数模板。

```

#include <iostream>      // 使用 C++ 新标准的流库
using namespace std;    // 将 std 名空间合并到当前名空间
const int SIZE = 6;     // 定义一个符号常量 SIZE
// 函数模板 compare( <T1 , T2> ) 的定义
template <typename T1, typename T2, int SIZE>
// 该函数模板带有两个虚拟类型参数 T1 和 T2 以及一个“常规参数”SIZE
void compare( T1 a[ ], T2 b[ ] )
{   for( int i = 0; i < SIZE; i++ ) {
        if( a[i] <= b[i] ) a[i] = b[i];
        // 若数组 a[ ] 的第 i 号元素不大于数组 b[ ] 的对应元素则 b[i] 赋给 a[i]
        else a[i] = 0;
        // 若数组 a[ ] 的第 i 号元素大于数组 b[ ] 的对应元素则将 a[i] 置零
    }
}

void main( void )
{   int maxToA[ ] = { 2, 24, 12, 42, 32, 9 };
    // 定义一个具有 6 个元素的、int 型数组 maxToA[ ] 并对其元素进行初始化
    double x[ ] = { 3.6, 2.8, 15.6, 24.2, 32.2, 9.9 };
    // 再定义一个 6 个元素的、double 型数组 x[ ] 并对其元素进行初始化
}

```

```
cout << "(1) 比较操作前 : " << endl; // 输出显示提示信息字符串
cout << "maxToA[ ] = { ";
// 输出显示“比较操作前”的 maxToA[ ]数组的元素值
for( int i = 0; i < SIZE; i++ ) {
    cout << maxToA[i];           // 输出显示该数组的第 i 号元素值
    if( i < SIZE - 1 ) cout << " , ";
    // 最后一个元素除外，每个元素后面输出一个逗号
    else cout << " };" << endl;
    // 最后一个元素后面输出空格、大括号和分号
}
// 输出显示“比较操作前”的 x[ ]数组的元素值
cout << "x[ ] = { ";
for( i = 0; i < SIZE; i++ ) {
    cout << x[i];               // 输出显示该数组的第 i 号元素值
    if( i < SIZE - 1 ) cout << " , ";
    // 最后一个元素除外，每个元素后面输出一个逗号
    else cout << " };" << endl;
    // 最后一个元素后面输出空格、大括号和分号
}
compare<int, double, SIZE>( maxToA, x );
// 调用模板函数 compare( T1, T2 )，因带有“常规参数”SIZE 模板实参必须写出
cout << "(2) 比较操作后 : " << endl; // 输出显示提示信息字符串
// 输出显示“比较操作后”的 maxToA[ ]数组的元素值
cout << "maxToA[ ] = { ";
for( i = 0; i < SIZE; i++ ) {
    cout << maxToA[i];           // 输出显示该数组的第 i 号元素值
    if( i < SIZE - 1 ) cout << " , ";
    // 最后一个元素除外，每个元素后面输出一个逗号
    else cout << " };" << endl;
    // 最后一个元素后面输出空格、大括号和分号
}
}
```

该程序的输出结果为：

(1) 比较操作前：

maxToA[] = { 2 , 24 , 12 , 42 , 32 , 9 };

x[] = { 3.6 , 2.8 , 15.6 , 24.2 , 32.2 , 9.9 };

(2) 比较操作后：

maxToA[] = { 3 , 0 , 15 , 0 , 32 , 9 };

由此可知,数组 `x[]` 的元素都是 `double` 型变量,其值都是双精度实型数,但赋值给 `int` 型数组 `maxToA[]` 的元素时都变成了整数,这是因为在执行模板函数 `compare()` 中, `if` 分支下的赋值语句 “`a[i] = b[i];`” 时,将产生 “向左看齐” 的自动类型转换 (详见文献[25]),即右值表达式 `b[i]` (`double` 型) 将自动转换成左值表达式 `a[i]` (`int` 型) 的数据类型再赋值给左值,所以,凡是大于 `maxToA[]` 数组对应元素值的 `x[]` 数组元素都是丢掉小数部分后再赋值给 `maxToA[]` 的对应元素。

函数模板的形参表中允许有几个不同类型的参数,既可以同时包含虚拟类型参数和常规参数。如例 6.2 中:

```
template <typename T1, typename T2, int SIZE>
```

其中, `T1` 和 `T2` 是不同数据类型的数组作为模板参数,在调用模板函数时 `T1` 实例化为 `int` 型,而 `T2` 实例化为 `double` 型,且这两个实参都是基本数据类型可以互相自动转换,完全可以采用这个函数模板进行比较操作。该函数模板包含的第 3 个参数 `SIZE` 却是常规参数,因此,必须在调用时给出模板实参表。

函数模板以任意类型 T_i ($1 \leq i \leq n$) 为模板参数,可适应任意数据类型并能完成功能相同的操作。但是,它的参数在没有实例化以前是一个虚拟的类型参数,使得函数模板的信息不完整而无法调用。只有当模板参数表内的 T_i 都实例化成具体的数据类型时,函数模板也就成为针对这一类特定类型的模板函数,它才是一个信息完整的、可以调用的函数。简言之,当编译系统读到用指定的数据类型作为模板实参调用函数模板时,就创建一个模板函数。

作为模板实参的 `class` 类型,必须在定义该类时,自行编写某些有关的重载运算符函数。例如,作为函数模板 `max` 实参的类,如 `Point` 类应有一个重载运算符函数 `operator>()`,即:

```
int operator > ( Point & a, Point & b )
{
    unsigned distance;
    /* 定义一个无符号整数变量 distance 记录比较两个点与原点距离的大小,它大于零则形参
       引用 a 所指的点对象与原点的距离比形参引用 b 所指的点对象要大 */
    distance = a.x + a.x + a.y + a.y - b.x - b.x - b.y - b.y;
    /* 把形参引用 a 所指的点对象与原点距离的平方减去形参引用 b 所指的点对象与原点距离的
       平方,并将其差值保存在无符号整数变量 distance 中 */
    return distance > 0 ? 1 : 0;
    // 当无符号整数变量 distance 大于零,则本函数返回值为 1,否则为 0
}

```

例 6.3 函数模板 `max(<T>)` 的应用实例。

```
#include <iostream.h>
#include <string.h>
```



```

#include "Point.h"          // Point 类的定义和实现部分

template <class T>
T max(T x, T y)
{   return (x > y) ? x : y;   }
// 重载模板函数用来比较两个字符串

char * max(char * s1, char * s2)
{   return strcmp(s1, s2) > 0 ? s1 : s2;   }
// 利用标准库函数 strcmp( ) 选择两个字符串中的较大者作为函数的返回值

void main( void )
{   int i = 24, j = 78;
    double x = 72.9, y = 48.6;
    Point p1(140, 260), p2(128, 84), p3;
    cout << "(1)max(i, j) is " << max(i, j) << endl;
    // "max(i, j)" 实例化显示两个 int 型数据之较大者

    cout << "(2)max(x, y) is " << max(x, y) << endl;
    // 显示两个 double 型数据之较大者

    cout << "(3)max(p1, p2) is " << max(p1, p2) << endl;
    // 显示两点中距离原点之较大者

    cout << "(4)max(s1, s2) is " << max("AMERICA", "CHINA") << endl;
    // 调用重载模板函数, 显示两个字符串中的较大者
}

```

该程序的输出结果为：

```

max(i, j) is 78
max(x, y) is 72.9
max(p1, p2) is P(140 , 260)
max(s1, s2) is CHINA

```

函数模板是一个虚拟的东西，它就像生产饼干的一套模具，虽然不是饼干产品，但用它可以大批量生产高、中、低档的饼干，档次的高低取决于做饼干的原材料和工艺。所以函数模板的定义格式中没有存储类的说明。

重载模板函数。C++还允许函数模板被一个或多个同名的非模板函数所重载。我们知道，编程者在定义一个模板前，必须把各种不同数据类型的处理操作基本统一成一种形式，以便于编写模板的程序代码，但是对于个别特殊的数据类型难于统一的，可通过重载模板函数来加以补充。通常，把原来的模板函数称为“主模板函数”，重载的模板称为“补充模板函数”。如例 6.3 中，主模板函数定义为：

```

template <class T>
T max(T x, T y)
{   return (x > y) ? x : y;   }

```

它对于 `int` 型、`float` 型和 `double` 型等基本数据类型，甚至 `class` 类型的对象都能成功地取出较大者。但是，若施加给两个字符串将产生错误的结果。例如：

```
char s1[ ] = "abe", s2[8] = "abk";
// 定义两个自动型字符串数组 s1 和 s2 分别保存两个 "abe" 和 "abk"

cout << max( s1 , s2 ) << endl;
// 取出较大者是按自动型字符串数组首地址的大小，在堆栈区数组 s1 的首地址大于 s2
```

取出的较大者为 "abe" 字符串这显然是不对的（前两个字符相同，第 3 个字符 'k' > 'e'），仔细分析可知，此时传递给该模板函数的是两个字符串数组的首地址进行比较，由于自动型数组存放在堆栈区先定义的数组首地址大于后定义的，所以取出较大者是 `s1` 内存放的字符串 "abe"。然而，取两个字符串中的较大者，应该是一个字符一个字符地进行比较，必须重载一个同名的非模板函数 "`char * max(char * s1, char * s2)`"，在其函数体内借助标准库函数 `strcmp()`，选择两个字符串中的较大者作为该函数的返回值。当模板参数是字符串时，调用该同名的非模板函数，实现取出两个字符串之较大者的功能。

6.1.3 模板函数的调用和模板实参的缺省

如前所述，在函数模板的定义格式中所声明的函数称为“模板函数”，调用模板函数的格式为：

模板函数名 < 模板实参表 > (实参表);

其中，<模板实参表>有时可以缺省，即可以省略不写。如例 6.1 中，模板函数的调用语句 "`swap(p1, p2);`" 还有例 6.3 中，"`max(p1, p2)`" 模板函数的调用表达式均省略掉了模板实参表，但是在调用模板函数时，模板实参表不是在任何情况下都可以缺省，只有在一定的条件下才能省略掉<模板实参表>。如前所述，系统在执行模板函数的调用语句时，必须要确定模板形参中每个虚拟类型参数实际所使用的数据类型，若从模板函数的实参表中能够得到足够的信息，确定该模板函数所有虚拟类型参数所使用的实际类型，正如例 6.1 中的 "`swap(p1, p2);`" 语句可省略掉<模板实参表>。而例 6.2 中的 "`compare<int, double, SIZE>(maxToA, x);`" 调用语句却是不能省略掉的，因为它的函数模板中含有“常规参数”。为此，我们把读者经常会遇到的几种不能缺省的情况列举如下：

含有“常规参数”的函数模板，在它的模板函数调用语句中，必须写出<模板实参表>。因为“常规参数”是用像 `int`、`float` 和 `double` 等声明的参数，不能再写入模板函数调用时的实参表，即“常规参数”不能再出现在模板函数调用时的实参表，否则将产生重复定义的错误，因此，必须在模板函数调用语句中写出<模板实参表>。当然，也可以把常规参数从函数模板声明语句中去掉，将这项信息放到模板函数的形参表中，如：

```
template <typename T1, typename T2>           // 把常规参数 SIZE 去掉
void compare( T1 a[ ], T2 b[ ], int sz ) // 模板函数新增一个形参 sz
```

```

{   for( int i = 0; i < sz; i++ ) {           // 形参 sz 确定数组大小
        if( a[i] <= b[i] ) a[i] = b[i];
        // 若数组 a[ ] 的第 i 号元素不大于数组 b[ ] 的对应元素则 b[i] 赋给 a[i]
        else    a[i] = 0;
        // 若数组 a[ ] 的第 i 号元素大于数组 b[ ] 的对应元素则将 a[i] 置零
    }
}

void main( void )
{   int maxToA[ ] = { 2, 24, 12, 42, 32, 9 };
    double x[ ] = { 3.6, 2.8, 15.6, 24.2, 32.2, 9.9 };
    ...
    compare( maxToA , x , SIZE );    // 模板函数调用语句中省略<模板实参表>
    ...
}    // 划有下划线的程序部分表示修改过的部分

```

这种编程方式可以避免每次书写模板函数调用语句时都要写出<模板实参表>的麻烦。

若模板函数调用语句的实参表中出现模棱两可 (ambiguity) 的数据类型时, 必须写出<模板实参表>明确地指明这些实参的数据类型。如例 6.3 中所定义的函数模板为:

```

template <class T>
T max( T x, T y )
{   return (x > y) ? x : y;   }

```

其模板函数的功能是获取两个形参中的较大者作为返回值。若有如下一条执行语句:

```
cout << "MAX =" << max( 12 , 22.8 ) << endl;
```

其中, “max(12 , 22.8)” 表达式将产生 C2782 编译错误, 因为在模板函数的调用语句中, 实参表的第 1 个参数是 int 型常量, 而第 2 个参数却是 double 型常量, 这与函数模板的定义不一致, 因此, 系统无法生成对应的函数调用版本的程序代码。显然必须在该调用语句中写出<模板实参表>明确地指明这些实参的数据类型, 即:

```
cout << "MAX =" << max < int >( 12 , 22.8 ) << endl;
```

编译系统生成对应的函数调用版本的程序代码为:

```

int max( int x, int y )
{   return (x > y) ? x : y;   }

```

当执行该函数调用程序代码, 实参传递给形参的过程中, 系统将实参 double 型常量 22.8 自动转换为 int 型常量 22 再赋给形参 y, 使得该函数返回的结果值为 34。当然, 也可以这样编写:

```
cout << "MAX =" << max < double >( 12 , 22.8 ) << endl;
```

此时将是把实参 int 型常量 12 自动转换为 double 型常量 12.0 再赋给形参 x, 使得该函数返回的结果值为 34.8。

若模板函数需获得由编程者所指定数据类型的返回值,在模板函数的调用语句中用<模板实参表>明确指定。如编程者希望通过模板函数 `max()` 获得一个 `int` 型返回结果,则写“`max < int > (12 , 22.8)`”,若需返回一个 `double` 返回值则写“`max < double > (12 , 22.8)`”,此时就不考虑参数的数据类型了。

若函数模板中的某些虚拟类型参数没有出现在模板函数的形参表中,且它们又不是模板形参表中最后连续的几个参数时,<模板实参表>中的这些参数不能省略。

例 6.4 函数模板的某个虚拟类型参数作为返回值。

```
#include    <iostream>           // 使用 C++新标准的流库
#include    <cstring>             // 纳入 C 语言标准函数库中的 string.h 头文件
using namespace std;            // 将 std 名空间合并到当前名空间
// 定义一个含有 3 个虚拟类型形参 T1、T2 和 T3 的函数模板,其中 T2 模板函数的返回值
template <typename T1, typename T2, typename T3>
T2 multiply( T1 x, T3 y )       // 返回到形参 x 和 y 的乘积
{   return      x * y;   }
void main( void )
{   int      i = 2, j = 8;           // 定义两个 int 型的自动变量 i 和 j
    double   a = 7.9, b = 4.6;      // 定义两个 double 型的自动变量 a 和 b

    cout << "(1) i * j = " << multiply < int, int > ( i, j ) << endl;
    /* 两个 int 型变量 i 和 j 相乘,由<模板实参表>的第 2 个实参指定返回的结果值也是 int
       型,最后一个实参可省略 */

    cout << "(2) a * b = " << multiply < double, double > ( a, b ) << endl;
    /* 两个 double 型变量 a * b 相乘,由<模板实参表>的第 2 个实参指定返回的结果值也是
       double 型,最后一个实参可省略 */

    cout << "(3) j * a = " << multiply < int, double > ( j, a ) << endl;
    /* int 型变量 j 乘以 double 型变量 a,由<模板实参表>的第 2 个实参指定返回的结果值也
       是 double 型,最后一个实参可省略 */

    cout << "(4) 6L * j * b = "
    << multiply < double, double, double > ( multiply< long, double > ( 6L, j ), b )
    << endl;
    /* 长整数常量 6L 乘以 int 型变量 j 再乘以 double 型变量 b,由<模板实参表>的第 2 个实
       参指定返回的结果值也是 double 型,最后一个实参可省略 */

}
```

该程序的输出结果为：

```
(1) i * j = 16
(2) a * b = 36.34
(3) j * a = 63.2
(4) 6L * j * b = 220.8
```

顺便指出,在模板函数的调用语句中,<模板实参表>所列写的实参必须与函数模板的声明语句中的<模板形参表>所列的形参要一一对应,即第1个实参对应第1个形参,第2个实参对应第2个形参...,如此类推。如前所述,在实例化过程中,模板实参的信息优先于函数实参的信息,虽然实参的个数可少于形参的个数,即有些模板实参可以省略不写,但只有当从函数实参表中所获得的信息足够确定模板形参表中全部虚拟类型或部分虚拟类型(它们是模板形参表中最后连续的几个参数)所对应实参的数据类型时,这些模板实参可以缺省。

6.2 类 模 板

6.2.1 类模板的定义

C++允许用户为某个类定义一种模板,称为“类模板”,当用户需要针对不同类型的数据编写几乎完全相同的程序代码时,可以使用类模板。例如,文献[25]定义的 Stack(堆栈)类,可使用 typedef 语句把形式数据类型 DataType 指定为某一特定的数据类型,使得 Stack 类可以构成字符型堆栈、整数型堆栈、实数型堆栈和类堆栈等,这只需修改 typedef 语句,把 DataType 指定为一种数据类型,如 char 型、int 型、double 型或 Point 类型等,使 Stack 类限定在一种数据类型上使用。但是,对于要在同一个程序中同时使用字符型堆栈、整数型堆栈、实数型堆栈和类堆栈等的场合,只有引入模板才能打破对形式数据类型 DataType 的限制,这就是为什么要引用类模板的原因。

在类模板定义的格式中,“template <class T₁, class T₂, ..., class T_n>”表示定义了一个模板。紧跟在关键字 template 后面,用尖括号包围的是模板参数表,T₁, T₂, ..., T_n称为模板的形式参数。因此,类模板的定义格式也可写成“template <模板形参表>”。模板形参表由一个或者多个用逗号隔开的模板参数组成,其内的每个参数前面都用关键字 class 或 typename 来表示,标识符 T_i是一个特定的 C++数据类型,它们可以是各种基本数据类型,也可以是标准类库中或用户定义的 class 类型。

类模板以任意类型 T_i (1 ≤ i ≤ n) 为模板参数,它是用于不同数据类型的一个类的样板,类模板与函数模板类似,是一个信息不完整、无法使用的类。要使用类模板,还必须把模板参数实例化成一种特定的数据类型,实例化模板参数的参数叫做模板实参。当 T_i 实例化为 char 型、int 型、float 型、double 型或用户定义的 class 类型时,类模板就实例化成一个模板类。例如,类模板 Stack 的模板参数 T 若实例化成字符型、整数型、实数型或 Complex 类型时,类模板 Stack 就实例化成字符堆栈、整数堆栈、实数堆栈和复数堆栈等。

类模板定义的一般格式为:

```

template <class T1 , class T2 ,... , class Tn>
// 即 template < 模板形参表 >,下同
class 类模板名 {
    ...          // 类体 , 类定义的声明部分
};

template <class T1 , class T2 ,... , class Tn>
<返回类型> 类模板名 <T1 , T2 ,... , Tn>::成员函数名 1 (形参表)
// 其中<T1 , T2 ,... , Tn>即为< 模板形参表 >, 下同
{
    ...          // 成员函数 1 的函数体
}

template <class T1 , class T2 , ... , class Tn>
<返回类型> 类模板名 <T1 , T2 ,... , Tn>::成员函数名 2 (形参表)
{
    ...          // 成员函数 2 的函数体
}

...

template <class T1 , class T2 ,... , class Tn>
<返回类型> 类模板名 <T1 , T2 ,... , Tn>::成员函数名 n (形参表)
{
    ...          // 成员函数 n 的函数体
}

```

与类的成员函数类似，类模板的所有成员函数都必须在类体内用函数原型加以声明，而成员函数的定义既可以在类体外，也可以在类体内定义。在类体外定义时，必须用“类模板 <模板参数名>::”指明该成员函数所属的类模板，它也可看成是模板类的类名。例如，对于类模板 Stack，类模板参数 T 替换了形式数据类型 DataType，可定义类模板 Stack 如下：

```

#include <iostream.h>
#include <stdlib.h>
const int MaxStackSize = 80;    // 数组元素，即堆栈数据项的最大个数
// 类模板 Stack 的定义
template <class T>              // 表示定义了一个模板
class Stack {
private :
    T stklist[MaxStackSize];    // 堆栈数组 stklist 用来存放数据项
    int top;
public :
    Stack(void);
    void push(const T & item); // 将一个新的数据项压入堆栈的操作
    T pop( void );             // 将一个新的数据项弹出堆栈的操作

```

```

    T peek( void );           // 取栈顶元素的数据项值
    int stkEmpty( void );     // 检测堆栈是否为空
    int stkFull( void );     // 检测堆栈是否已满
    void clearStk( void );    // 设置堆栈为初始状态
};

template <class T> // 类模板成员函数的实现部分必须有一个模板参数列表, 下同
// 用 Stack<T>::取代 Stack::指明成员函数所属的类模板, 下同
Stack<T>::Stack(void)       // 无参数的构造函数
{
    top = -1;
}

template <class T>
void Stack<T>::push(const T & item) // 把数据项 item 压入堆栈的操作
{
    if(top == MaxStackSize - 1) {
        cerr << "堆栈已满, 终止执行程序 !\n";
        exit(1);
    }
    top ++;
    stklist[top] = item;
}

template <class T>
T Stack<T>::pop(void)        // 把数据项 item 弹出堆栈的操作
{
    T temp;
    if(top == -1) {
        cerr << "堆栈已空, 终止执行程序 !\n";
        exit(1);
    }
    temp = stklist[top];
    top --;
    return temp;
}

template <class T>
T Stack<T>::peek(void)       // 取栈顶元素的数据项值
{
    if(top == -1) {
        cerr << "堆栈已空, 终止执行程序 !\n";
        exit(1);
    }
    return stklist[top];
}

template <class T>
int Stack<T>::stkEmpty(void) // 检测堆栈是否为空

```

```

{    return top == -1;    }
template <class T>
int Stack<T>::stkFull(void)    // 检测堆栈是否已满
{    return top == MaxStackSize - 1;    }
template <class T>
void Stack<T>::ClearStk(void)    // 设置堆栈为初始状态
{    top = -1;    }

```

并将它的定义存放在头文件 tmpstack.h 中。

在每个类模板成员函数的实现部分开头，都需要有一个模板参数列表，即

```
template <class T1 , class T2 , ... , class Tn>
```

表示这是一个类模板的成员函数，如类模板 Stack 的所有成员函数的开头处都以 template <class T> 开始，且类模板的类名为 Stack<T>。

6.2.2 模板类的实例化和对象的定义

由于类模板是一个抽象的、虚拟的东西，要使用它还必须实例化成模板类，其方法是在定义模板类的对象时，把模板参数用<模板实参表>指定为一个特定的数据类型，一般格式为：

类模板名 < 模板实参表 > 对象名列表；

其中，模板实参的数据类型可以是基本数据类型，如 char *、int 和 double 等，也可以是 class 类型。

其中，对象名列表可以是一个对象名，也可以是由逗号隔开的多个对象名，且每个对象还可以带有用一对圆括号包围的初值表，按类模板所定义的构造函数列写初值表。若有的对象不带初值表，则类模板中必须定义一个无参数的构造函数。

例 6.5 在一个源程序中，同时使用字符型堆栈、整数型堆栈。

```

#include <iostream.h>
#include <string.h>
#include "tmpstack.h"    //该头文件存放有类模板 Stack 的定义

void multiRadixOutput(long unsigned num, int R)
{
    Stack<int> s;
    /* 定义模板类的一个自动型对象 s，并把模板参数 T 指定为 int 型，则创建了一个自动型的
       整数堆栈 */
    do {
        s.push(num % R);
        num /= R;
    } while( num != 0 );
    while( ! s.stkEmpty( ) )
        cout << s.pop( );
}

```



```

    }
    void main( void )
    {
        long unsigned num;
        unsigned R;
        for(unsigned i = 0; i < 3; i++)    {
            cout << "请输入计数制的基数 R (2    R    9) : " ;
            cin >> R;
            cout << "请输入一个正整数 : " ;
            cin >> num;
            cout << num << "以基数为" << R << "的结果是";
            multiRadixOutput(num, R);
            cout << endl;
        }
        Stack<char> ss;
        /* 定义模板类的一个自动型对象 ss, 并把模板参数 T 指定为 char 型, 则创建了一个自动型
           的字符堆栈 */
        char name[81], reverse[81];
        cout << "请输入您的英文名字 : " ;
        cin >> name;
        for(i = 0; i < strlen(name); i++)
            ss.Push(name[i]);
        for(i = 0; i < strlen(name); i++)
            reverse[i] = ss.Pop( );
        reverse[strlen(name)] = '\0';
        cout << "很抱歉 ! 把您的英文名字逆序显示出来为 : " << reverse << endl;
    }

```

类模板、实例化的模板类及其对象之间的关系如图 6.2 所示, 类模板 Stack 经实例化后, 成为类名为 Stack<T>的模板类, 在 main() 函数体内, 用模板类 Stack<T>

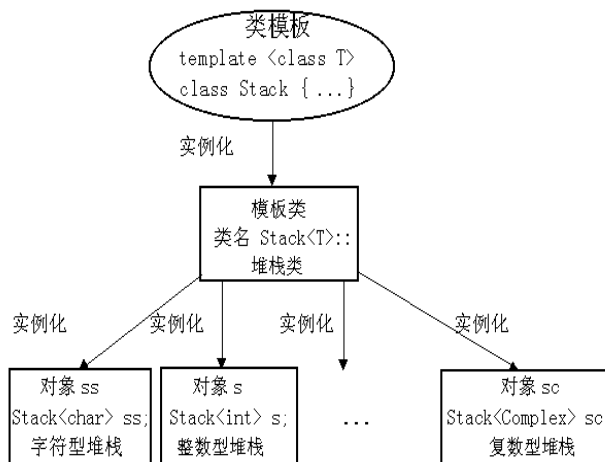


图 6.2 类模板、模板类和对象间的关系

定义了一个对象——字符型堆栈 `ss`，在 `multiRadixOutput()` 函数体内，又用模板类 `Stack<T>` 定义了另一个对象——整数型堆栈 `s`。不言而喻，还可以用它定义实数型堆栈和复数型堆栈等，从而实现了在同一个程序中同时使用字符型堆栈、整数型堆栈、实数型堆栈和类堆栈等。

6.2.3 类模板的继承

对于继承关系中的基类和派生类，它们中任何一个都可以是模板类或者模板类的实例即模板类。常见的情况有如下几种：

一个普通类从类模板的一个实例，即模板类继承而来。例如：

```
template    < typename  T > // 类模板的声明，模板形参 T 为虚拟类型参数
class      TmpBase  {      // 类模板的一个实例即模板类的定义
    T      value;          // 用模板形参 T 说明私有数据成员 value 的数据类型
public :
    ...
};
// 普通类 Derived 从模板类 TmpBase 继承而来，模板形参 T 对应的模板实参为 int
class      Derived : public TmpBase < int > {
    ...
};
```

由此可见，普通 `class` 类型 `Derived` 类是由一个模板类 `TmpBase` 继承而来，继承方式为公有继承方式，关键字 `public` 后面的类模板要跟有 <模板实参表>，定义格式为：

```
template < 模板形参表 >
```

```
class 类模板名 { // 类模板的一个实例即模板类作为基类
    ...          // 类体, 类定义的声明部分
};

class 派生类名 (普通类) : 类模板名 < 模板实参表 > {

    ...

};
```

一个模板类从普通类继承而来。在设计一个类层次结构时,最好用一个普通类作为基类,再由它派生出一个模板类,把模板类中与虚拟类型参数无关的数据成员分离出来放在基类(普通类)中,这是经常采用的一种编程方法。例如:

```
class Base { // 基类 Base 为普通类
    ...
};

template < typename T > // 类模板的声明, 模板形参 T 为虚拟类型参数
class TmpDerived : public Base { // 类模板的一个实例即模板类的定义
    T value;
    // 用模板形参 T 说明私有数据成员 value 的数据类型
public :
    ...
};
```

由模板类派生出模板类,即基类和派生类都是类模板的实例。例如:

```
template < typename T > // 类模板的声明, 模板形参 T 为虚拟类型参数
class TmpBase { // 基类 Base 为类模板的一个实例即模板类
    T value;
    // 用模板形参 T 说明私有数据成员 value 的数据类型
public :
    ...
};

template < typename T1, typename T2 >
// 类模板的声明, 模板形参 T1 和 T2 为虚拟类型参数
// 派生类也是类模板的一个实例即模板类
class TmpDerived : public TmpBase <T1> {
    T2 total;
    // 用模板形参 T2 说明私有数据成员 total 的数据类型
public :
    ...
};
```

```
};
```

一个模板类从一个基类继承而来，该基类由模板参数表给出。即一个模板类可以从一个尚未确定的基类继承而来，它究竟继承哪个基类由模板参数表决定。下面用一个例题说明之。

例 6.6 一个模板类 Derived 从一个尚未确定的基类继承而来。

```
#include    <iostream>          // 使用 C++新标准的流库
using namespace std;          // 将 std 名空间合并到当前名空间
class BaseA {                  // 普通基类 BaseA 的定义
public :
    BaseA( void )              // BaseA 类的构造函数
    {   cout << "BaseA 类的构造函数被调用 !" << endl;   }
};
class BaseB {                  // 普通基类 BaseB 的定义
public :
    BaseB( void )              // BaseB 类的构造函数
    {   cout << "BaseB 类的构造函数被调用 !" << endl;   }
};
// 类模板的说明，含有一个虚拟类型参数 T 和常规参数 cnt
template    < typename T, int cnt >
class BaseC {                  // 模板类 BaseC 的定义
    T count;
// 类模板的虚拟类型参数 T 说明私有数据成员 count 的数据类型
public :
    BaseC( T c = cnt )         // 模板类 BaseC 的构造函数
    {   count = c;
        cout << "BaseC 类的构造函数被调用 !" << endl;
    }
    int readCount( void ) { return count; }
    // 读取私有数据成员 count 的值
};
// 派生一个模板类，其基类由模板参数 T 实例化后决定
template    < typename T >
class Derived : public T {
public :
    Derived(void ) : T( )      // 派生类 Derived 的构造函数
    {   cout << "Derived 类的构造函数被调用 !" << endl; }
};
void main( void )
```

```

{   Derived < BaseA >   a;
    // 定义一个派生(模板)类 Derived 的对象 a, 其直接基类由模板实参指定为普通基类 BaseA

Derived < BaseB >   b;
/* 再定义一个派生(模板)类 Derived 的对象 b, 其直接基类由模板实参指定为普通基类
BaseB */

    Derived < BaseC < int, 6 > >   c;
/* 再定义一个派生(模板)类 Derived 的对象 c, 其直接基类由模板实参指定为 BaseC 类,
    由于 BaseC 类也是模板类, 用模板实参表指定其包含的虚拟类型参数 T 为 int 型, 常规参
    数为 6 */

    cout << "派生类 Derived 的对象 c 的 count 值 = " << c.readCount() << endl;
    // 输出显示派生(模板)类 Derived 的对象 c 的私有数据成员 count 值
}

```

该程序的输出结果为：

```

BaseA 类的构造函数被调用 !           (创建派生(模板)类 Derived 的对象 a 的输出信息)
Derived 类的构造函数被调用 !         (创建派生(模板)类 Derived 的对象 a 的输出信息)
BaseB 类的构造函数被调用 !           (创建派生(模板)类 Derived 的对象 b 的输出信息)
Derived 类的构造函数被调用 !         (创建派生(模板)类 Derived 的对象 b 的输出信息)
BaseC 类的构造函数被调用 !           (创建派生(模板)类 Derived 的对象 c 的输出信息)
Derived 类的构造函数被调用 !         (创建派生(模板)类 Derived 的对象 c 的输出信息)
派生类 Derived 的对象 c 的 count 值 = 6

```

由此可知：

若一个(派生)模板类从一个尚未确定的基类继承而来, 它究竟继承哪个基类由模板参数表决定, 创建该(派生)模板类的对象时, 必须用<模板实参表>指定它的基类, 其编写格式为：

派生(模板)类名 < 模板实参表 > 对象名列表;

如例 6.6 中, main() 函数体内创建 3 个 Derived 类的对象 a、b 和 c, 前两个对象 a 和 b 指定所属类的基类是普通类 BaseA 和 BaseB, 而对象 c 就比较复杂, 模板实参指定它的基类是 BaseC 类, 该基类也是一个模板类, 那么在<模板实参表>内又嵌套一个<模板实参表>, 内层的用来对基类 BaseC 进行实例化, 外层对派生(模板)类 Derived 进行实例化。

该派生(模板)类 Derived 的类体内必须编写一个构造函数, 与普通派生类定义一样, 在其函数头提供一个初始化列表, 以便调用基类的构造函数初始化从基类继承而来的数据成员, 所不同的是其基类是一个尚未确定的(模板)虚拟类型参数 T, 以 T 作为基类的构造函数名, 则派生(模板)类 Derived 的构造函数写成：

```

Derived(void) : T( )
{   cout << "Derived 类的构造函数被调用 !" << endl; }

```

小 结

模板是一种高效率重用程序代码的方式，C++有函数模板和类模板，它们都是虚拟的东西，必须随模板参数的实例化而成为模板函数和模板类。每个模板类的实例是一个具体对象，可以像普通对象一样使用，作为函数参数或返回值。

函数模板和类模板的定义通常放在头文件中，便于所有的源文件引用。

声明一个模板的编写格式为：

template	< 模板形参表 >
函数的定义	或 类的定义

其第1条语句为模板的声明语句。显然，一个函数模板包含模板形参表和函数形参表，它们都是函数模板实例化的信息源。而类模板中紧跟模板的声明语句之后的类定义部分，每个成员函数的实现部分前都必须加上一条模板的声明语句，并以“类模板名<模板形参表>”作为类名加上作用域运算符::用以指明该成员函数所属类。

模板形参分为虚拟类型参数和常规参数两种，虚拟类型参数需用关键字 `class` 或 `typename` 加以定义，对应的实参是基本数据类型和 `class` 类型。常规参数用具体的、像 `int`、`double` 和 `char *` 等数据类型定义，对应的实参通常是常量表达式。

函数模板实例化成模板函数，其调用语句的编写格式为：

模板函数名	< 模板实参表 >	(实参表);
-------	-----------	----------

编译系统根据调用语句所给出的模板实参去初始化模板形参，生成实例化模板函数的程序代码，即一个对应的、形参和返回值的数据类型均已确定的具体函数的实现部分程序代码。一个类模板实例化成模板类，创建对象的定义格式为：

类模板名	< 模板实参表 >	对象名列表;
------	-----------	--------

其中，模板实参的数据类型可以是基本数据类型，如 `char *`、`int` 和 `double` 等，也可以是 `class` 类型。

在调用模板函数时，对于模板参数表中最后几个参数数据类型的确定，若能从模板函数的实参表内获取足够的信息，则这些模板参数对应的模板实参可以省略。

习 题 6

一、选择填空

- 关于关键字“`typename`”和“`class`”，下列描述中（ ）是正确的。
 - 程序中所有的“`class`”都可以替换为“`typename`”
 - 程序中所有的“`typename`”都可以替换为“`class`”
 - A)和B)都不可以
 - A)和B)都可以
- 在调用模板函数时模板实参的使用，下列描述中（ ）是正确的。
 - 对于虚拟类型参数所对应的模板实参，若能从模板函数的实参中获得相同的信息，

则都可以省略

- B. 对于虚拟类型参数所对应的模板实参, 若它们是参数表中最后的若干个参数, 则都可以省略
- C. 对于虚拟类型参数所对应的模板实参, 若能够省略则必须省略
- D. 对于常规参数所对应的模板实参, 任何情况下都不能省略
3. 对于类模板, 下列描述中 () 是错误的。
- A. 用类模板定义一个对象时, 不能省略实参
- B. 类模板只能有虚拟类型参数
- C. 类模板本身在编译过程中不会生成任何代码
- D. 对于常规参数所对应的模板实参, 任何情况下都不能省略
4. 有如下函数模板定义:

```
template <class T>
T func( T x, T y ) { return x * x + y * y; }
```

在下列调用 func() 的语句中 () 是错误的。

- A. func(3, 5);
- B. func<>(3, 5);
- C. func(3, 5.5);
- D. func<int>(3, 5.5);

二、填空题

1. 已知:

```
int SQUARE( int n ) { return n + n; }
```

 和

```
double SQUARE( double n ) { return n + n; }
```

是一个函数模板的两个实例, 那么该函数模板的定义是_____。

2. 将适当的内容填写在下面程序横线处, 使该程序的执行结果为:

25 24 23 22 21

0 7.5 6.4 5.3 3.1

```
#include <iostream>
using namespace std;
template <typename T>
void func(_____)
{
    _____;
    for( int i = 0; i < n / 2; i++ )
        t = a[i], a[i] = a[n - 1 - i], a[n - 1 - i] = t;
}
void main( void )
{
    int a[5] = { 21, 22, 23, 24, 25};
```

```

double d[6] = { 3.1, 4.2, 5.3, 6.4, 7.5 };
func( a, 5 );
func( d, 6 );
for( int i = 0; i < 5; i++ )
    cout << a[i] << " ";
cout << endl;
for( i = 0; i < 6; i++ )
    cout << d[i] << " ";
cout << endl;
}

```

3. 将适当的内容填写在下面程序横线处，使该程序的执行结果为：

```

The pair is (22, 66)
The pair is (26.8, 89.6)

```

```

#include <iostream>
using namespace std;
template <typename T>
class Pair {
public :
    T x, y;
    void print( void )
    {   cout << "The pair is (" << x << " , " << y << ")" << endl; }
};
void main( void )
{
    _____;
    c1.print();
    _____;
    c2.print();
}

```

4. 试有如下程序：

```

#include <iostream>
using namespace std;
template <class T>
T summed( T * pi )
{
    T sum = 0;
    while( * pi )
        sum += * pi ++;
    return sum;
}

```



```
}  
void main( void )  
{   int  a[ ] = { 2, 4, 6, 8, 0, 12, 14, 16, 18 };  
    cout << summed( a ) << endl;  
}
```

该程序的输出结果是_____。

5. 将适当的内容填写在下面程序横线处, 使该程序的执行结果为: 4.4

```
#include <iostream>  
using namespace std;  
template <class T>  
T average( T * pd )  
{   T  sum = 0;  
    _____;  
    while( pd[i] )  
        sum += pd[i++];  
    return _____;  
}  
void main( void )  
{   double a[ ] = { 2.5, 4.5, 6.5, 8.5, 0.0, 12.5, 14.5, 16.5, 18.5 };  
    cout << average( a ) << endl;  
}
```

6. 将适当的内容填写在下面程序横线处, 使该程序的执行结果为: 9 8 7 6

```
#include <iostream>  
using namespace std;  
template <typename T>  
void func( T a[ ], int n )  
{   T  t;  
    for( int i = 0; i < n / 2; i++ )  
        t = a[i], a[i] = a[n - 1 - i], a[n - 1 - i] = t;  
}  
void main( void )  
{   int  a[ ] = { 6, 7, 8, 9 };  
    func( a, 4 );  
    for( int i = 0; i < 4; i++ )  
        cout << a[i] << " ";  
    cout << endl;  
}
```

7. 试有如下的类模板定义:

```

template <class TYPE>
class TempClass {
    TYPE    n;
public :
    TempClass( TYPE t );
    1/4
};

```

其中构造函数 TempClass(TYPE t)用 t 值初始化数据成员 n。因此，在模板类体外。

构造函数 TempClass(TYPE t)应定义为_____。

8. 将适当的内容填写在下面程序横线处，使该程序的执行结果为：5 5 5 9 9

```

#include <iostream>
using namespace std;
template <typename TYPE>
class MyClassTMP {
    TYPE data;
public :
    MyClassTMP( TYPE x, TYPE y, TYPE z );
    TYPE getData( void ) { return data; }
};
template <typename TYPE>
_____MyClassTMP( TYPE x, TYPE y, TYPE z )
{
    _____ ;
    else if( y > z )    data = y;
    else data = z;
}
void main( void )
{
    MyClassTMP <int> a1(3, 4, 5), a2(5, 3, 4), a3(4, 5, 3),
                    a4(7, 9, 8), a5(8, 7, 9);
    cout << a1.getData( ) << " " << a2.getData( ) << " " << a3.getData( )
        << " " << a4.getData( ) << " " << a5.getData( ) << endl;
}

```

三、编程题

1. 编写一个函数模板 SWAP，它可以用来交换任意两个类型相同的对象或基本数据类型变量的值。
2. 已知：

```

int square( int a[ ], int size );
double square( double a[ ], int size );

```

是一个函数模板的两个实例,其功能是计算并返回数组 `a[]` 的 `size` 个元素的平方和。试编写该函数模板的程序代码。

3. 已知一个模板的不完整声明如下:

```
template <class T, int r, int c>
void subtotal( T (* pa )[c]);
```

其中 `pa` 是 `r` 行 `c` 列二维数组的行数组指针, `subtotal()` 函数计算每一行从第 1 列到第 `c - 1` 列元素的合计,并将其合计值保存到该行的第 0 列元素中。试将该模板的代码补充完整。

4. 已知一个模板的不完整声明如下:

```
template <class T, int SIZE>
T valueOf( T determinant[ ][SIZE]);
```

其中 `valueOf()` 函数计算并返回 `SIZE × SIZE` 的行列式 `determinant[][SIZE]` 的值。试将该模板的代码补充完整。

5. 试有如下类声明:

```
class MyClass {
    int data1;
    double data2;
public:
    MyClass( int , double );
    int getData1( void ) const;
    double getData2( void ) const;
};

MyClass::MyClass( int d1, int d2 ) : data1(d1), data(d2) { }
int MyClass::getData( void ) const { return data1; }
double MyClass::getData2( void ) const { return data2; }
```

试将该类声明修改成类模板声明,使得数据成员 `data1` 和 `data2` 可以是任何数据类型。

第7章 C++的流库

C++的流库是一个较复杂的标准类库，它是用继承的方法建立起来的实用流库。它不仅提供了与C语言标准函数库类似的各种I/O操作功能，更重要的是使I/O操作具有面向对象的特征，功能更强、更灵活。

7.1 流库的类层次结构

7.1.1 什么是流

“流(Stream, 翻译成‘流操作’, 简称‘流’)”是一个抽象的概念, 通常“流”是与I/O操作紧密联系在一起的, 使I/O操作具有面向对象的特征。

在计算机内存(Memory)中, 任何一块数据经处理后, 从一个内存地址移动到另一个内存地址的数据流动称为流操作, 简称为流(Stream)。流实际上是通过引入缓冲器(Buffer)机制, 将一个对象数据传送到另一个对象的传递过程抽象成“数据从一个源点(Source)到一个终点(Sink)的流动操作”, 这些对象被称为“流对象”。所谓“缓冲器机制”是指编译系统自动在内存中为每一个正在使用的磁盘文件名(如图7.1所示的File1、File2)开辟一个缓冲器(如图7.1所示的Buffer1、Buffer2)。从磁盘向内存读取数据, 则一次从磁盘文件(如File1)中将一批数据输入到内存缓冲器(如Buffer1), 然后再从缓冲器逐个地将数据送到程序中对象所对应的存储空间内。如果从内存向磁盘文件(如File2)输出数据, 则必须将数据先送到内存的缓冲器(如Buffer2)集中, 待装满缓冲器后才将缓冲器的全部数据一次写入磁盘文件(如File2)中保存。从流中读取数据的操作称为“提取(Extracting)”操作或“获取(Getting)”操作, 向流中写入数据的操作称为“插入(Inserting)”操作或“压入(Putting)”操作。

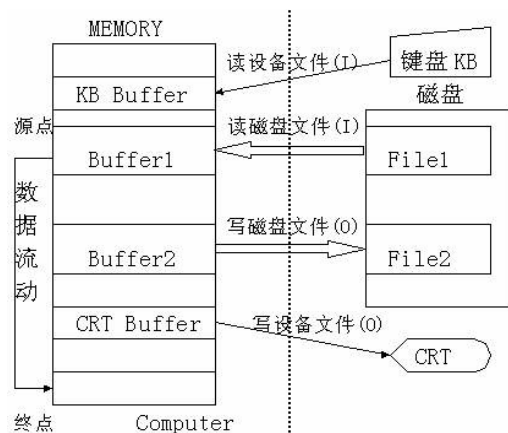


图7.1 用“缓冲器机制”抽象成流操作

顺便指出，正如图7.1所示，在计算机与外部设备之间用虚线画一条边界线，从磁盘读取数据对计算机来说应该是输入操作（Input，简写为I），而向磁盘写入数据对计算机而言应是输出操作（Output，简写为O）。

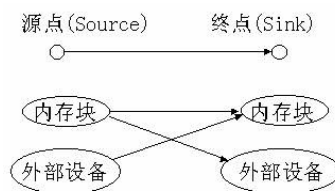


图7.2 广义的流

为了将外部设备与磁盘文件统一起来，C++也给外部设备引入了缓冲器机制，把实际的各种外部设备抽象化为“输入输出流的对象”，将流的概念进一步扩展成如图7.2所示的“广义的流”，其源点和终点既可以是内存块，也可以是外部设备。如图7.1所示，系统在内存中为CRT自动设置了一个专用缓冲器“CRT Buffer”，将CRT定义成标准输出流的一个对象cout，为键盘自动设置了一个专用缓冲器“KB Buffer”，将键盘定义成标准输入流的一个对象cin。这样一来，这里所说“流对象”既可以是一个外部设备，例如键盘、磁盘机、光盘机、显示器、打印机等，也可以是一个磁盘文件。这种逻辑上的统一为程序设计提供了很大的方便。C++和C一样，本身没有输入和输出操作的语句，其输入/输出操作是调用标准流库提供的I/O操作函数来完成的。缓冲器机制的引入为编程者提供了一个统一的接口界面，使得流操作与被访问对象无关。即流库的I/O操作函数既可以用来处理磁盘文件，又可以用来控制外部设备。这样一来，对外部设备的读/写与磁盘文件的读/写操作在编程方法上完全相同。即一个用于对磁盘文件读取数据的操作函数，也可用于键盘的读取操作。特别是C++将流与I/O操作紧密联系在一起，用

iostream 库取代了stdio 库，使I/O操作具有面向对象的特征，可以重新定义（通过重载运算符函数）已有的I/O操作运算符，以支持编程者自行定义的class类型的输入/输出操作。

在传统的ANSI C标准函数库中的printf(), scanf(), fopen(), fclose(), ... , strcpy(), sprintf()等都可看做是流操作。C++流库是用继承方法建立起来的输入/输出（I/O）类库，可分为如表7.1所示的3种类别。

表7.1 流类别与标准库函数

流类别	取代ANSI C库的标准函数
标准I/O流 文件	scanf(), printf(), fopen(),
I/O流 内存格式	fread(), fclose() sscanf(),
化流	sprintf()

顺便指出，尽管在C++源程序中可同时使用C++流库和ISO/ANSI C老标准函数库，但建议最好不要这样写程序，因为两者所建立的缓冲器刷新并不一定同步。

例7.1 C++流库和ANSI C标准函数库不要混用。

```
#include < iostream >           // 使用C++新标准的流库
#include < cstdio >              // C语言标准函数printf( )原型声明在其中
using namespace std;           // 将std标准名空间合并到当前名空间

void main( void )
{   int value = 3.0;
    cout << "\n There ";
    printf( "are %d ", value );
    cout << " bicycle left. ";
}
```

本例程希望产生如下输出结果：

```
There are 3 bicycle left.
```

虽然可以强制C++流库和ANSI C标准函数库的流操作同步，但需要相当多的执行时间，在C++中一般不使用ANSI C流操作函数。

7.1.2 流库的类层次结构

正如1.3.5节所述，Visual C++ V6.0中包含着两种流库，即C++老标准流库和C++新标准流库，前者所包含的流库头文件均带“.h”扩展名，以确保与过去老的C++程序兼容。后者所包含的流库头文件都不带“.h”扩展名，它是在前者的基础上进行了修改并将其模板化，使得流库具有更好的灵活性、兼容性和可扩充性。读者在熟悉了C++新

标准流库的类层次结构图以后，可查阅MSDN帮助系统了解更详细的信息。

1. C++老标准流库的类层次结构

Visual C++老标准流库是用继承方法建立起来的一个输入/输出(I/O)类库，它具有两个平行的基类：streambuf类和ios类，所有的stream类都由两者之一继承而来，它们的类层次结构图如图7.3和图7.4所示。streambuf类主要负责缓冲器管理的许多细节，如从源点到终点的所有数据移动，对缓冲器所需的基本操作机制都是由streambuf类提供支持的。缓冲器用来处理在两点间传送的数据，即数据从源点移到缓冲器，再从缓冲器移出，送入终点。对于输出缓冲器通常是一次写入一个字符到输出缓冲器，当缓冲器

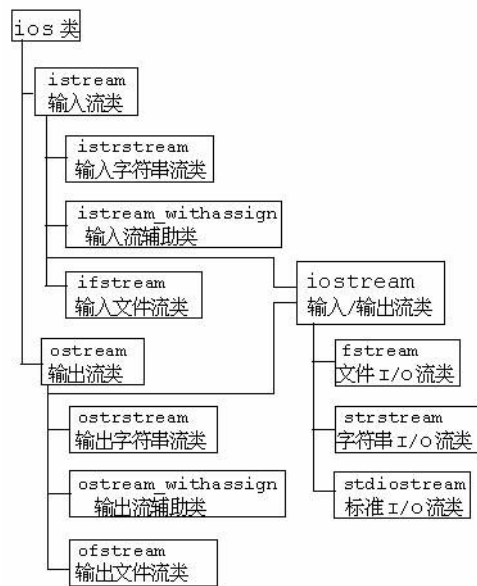


图7.3 输入/输出流类层次结构图



图7.4 streambuf类的层次结构

写满上溢时，才将缓冲器的全部数据一次写入磁盘文件中保存，并一次性刷新缓冲器，以备重新写入用。对于输入缓冲器也是从输入缓冲器一次读取一个字符，当缓冲器读空下溢时，才一次性地从磁盘将文件读入到缓冲器填满。

为了实现上述缓冲器机制并确保字符的读写顺序，streambuf类使用两个指针，即get指针（简称gp_{ptr}）和put指针（简称pp_{ptr}），如图7.5所示。

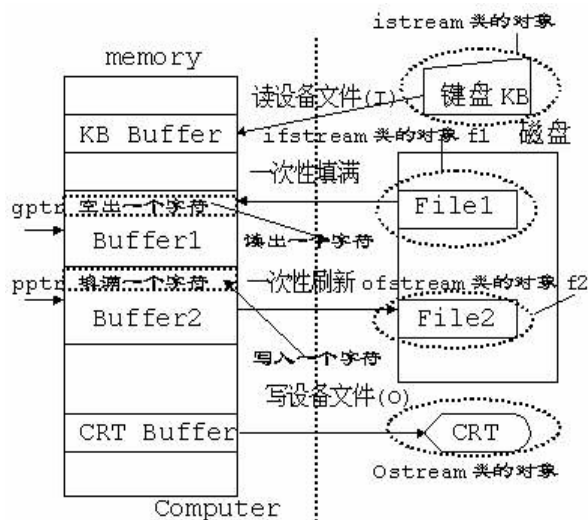


图7.5 get指针(gptr)和put指针(pptr)

对于输入缓冲器 (Input Buffer) 使用get指针, 每当从缓冲器读出一个字符 (或者说提取了一个字符) 时, get指针自动增1, 指向下一个字符, 或者说从缓冲器拿走一个字符, 则缓冲器内空出一个字符。如果get指针到达缓冲器尾部, 即指向缓冲器的最后一个可获得字符, 则说明缓冲器已空并溢出, 称为“下溢” (这就像井水干枯了要从外面补充地下水), 则一次性地将新数据从磁盘读入到缓冲器内填满, 这一物理过程称为“一次性填满”, 或抽象成逻辑过程, 称为从输入流对象f1一次性地读入到缓冲器。对于输出缓冲器 (output Buffer) 使用put指针, 每将一个字符放入到输出缓冲器 (或者说插入了一个字符), 则缓冲器内插入一个字符。当put指针到达缓冲器尾部时, 则缓冲器写满并溢出, 称为“上溢” (这就像井水已装满向外面溢出水), 则一次性地将缓冲器内容写入到磁盘, 并刷新缓冲器, 称为“一次性刷新”, 即将缓冲器内容一次性地取出, 写入磁盘, 缓冲器变空, 以备重新写入用。或抽象成逻辑过程, 称为“将缓冲器内容一次性地取出, 写入到输出流对象f2”。这与图7.6所示的井水“上溢”和“下溢”极其相似。

显然这种缓冲器是一个先进先出 (FIFO, Forward Input Forward Output) 的缓冲器, 适用于“字符队列”的数据结构。

流库中的一些类, 例如istream、ostream及其派生类ifstream、ofstream等都备有一类成员函数, 可随机移动get指针和put指针, 以实现随机存取。例如对磁盘文件添加内容, 可放在已存在文件的尾部, 也可插入到文件中的任何位置, 这是C语言中的I/O操作所望尘莫及的。

filebuf类用于文件I/O流的缓冲器操作, 通过文件描述符 (File

descriptor，或称文件句柄File Handle) 进行文件I/O操作，它是以低层次的I/O操作为基础，而高层次的I/O操作是在C语言中以文件指针为核心的缓冲型文件系统。filebuf类及其派生类备有打开、关闭和寻找文件的成员函数。

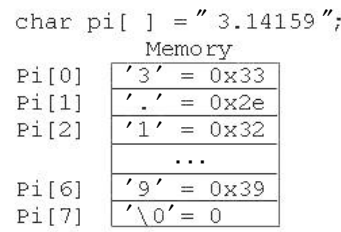


图7.7 浮点数在内存中的字符串格式化

strstreambuf类用来处理内存中存放字符串的缓冲器，允许编程者将字符写入到内存中的字符串数组或者从内存字符串数组中读出字符。由于一个字符占用内存一个字节，便于管理、传送、存储和转换，因此常常将内存中的各种数据实现字符形式的格式化，例如一个浮点数3.14159，以数字字符串的形式存放在内存中，如图7.7所示。

ios类提供了I/O流所需要的公共操作以及对流状态标志进行设置的功能，它管理全部I/O流操作和格式化I/O控制。streambuf类只有在与ios类组合后才能派生出各种实用的流类，为编程者提供使用流库的接口界面。通常编程者只与这些派生出的实用流类打交道。总之，iostream老标志流库是一个较复杂的类层次结构系统，共有18个类，用多继承的方法实现。但编程者经常使用、需掌握的只有ios类、istream类、ostream类和iostream类等。

顺便指出，Visual C++老标准流库主要是针对以字节为单位的单字符集合设计的，例如，读写的文件可以是按顺序的字节序列，也可以是能重复读写的字节集合如磁盘文件，也可以是由程序产生的字节流如管道（pipeline），也可以是由外部设备发送或接收的字节流等。其存储方式是一个字符占用内存的一个字节即一个存储单元，不仅便于管理、传送、存储和转换，也便于初学者理解和掌握。它是C++新标准流库的技术基础，掌握它对于重点学习新标准的内容是十分必要的。

流操作运算符的使用方法参见1.3节。

2. C++新标准流库

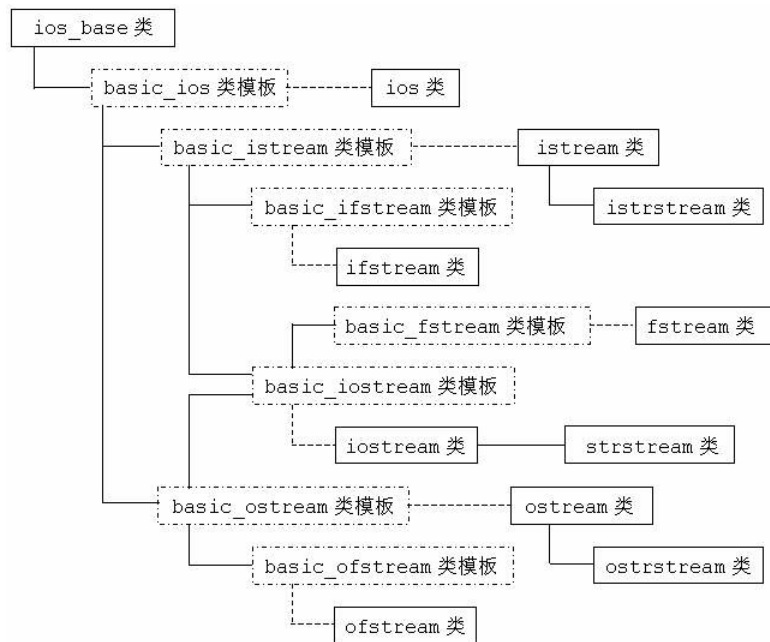
C++新标准流库所包含的流库头文件都不带“.h”扩展名，按照1.3节所述的办法可以快捷地转换成新头文件名，它虽然也是用继承方法建立起来的一个输入/输出（I/O）类库，但却是大量采用模板建立起来的一个类型安全性（type safety）和可扩充性都很好的流库系统。其内所用的类型几乎都定义成模板。例如，经常使用的新标准头文

件<iostream>可以用来处理字符流 (streams of characters), 这里字符不仅可以是单字节的char型字符, 也可以是wchar_t型字符, 或者是双字节的Unicode型字符, 这都是最终由类模板实例化时确定采用哪一种字符。若将类模板的形参指定为<char>, 即指定为C语言的char型字符则就可以实例化为与C++老标准流库基本相同的标准输入/输出流对象, 如cout、cin、cerr、clog等。由此可见, C++新标准流库具有灵活多变的可扩展性, 编程者应尽量采用C++新标准流库, 用#include语句把不带“.h”扩展名的新头文件如<iostream>纳入到当前程序中, 并用using语句把标准名空间std合并到当前名空间, 即:

```
#include <iostream> // 使用C++新标准的流库

...

using namespace std; // 将std标准名空间合并到当前名空间
```



框为class类型或类模板实例化生成的（模板）类。在此必须强调指出，C++新标准流库是在老标准流库的基础上进行模板化，并修改了继承体制，为增强诊断功能和软件的国际化扩充了其内容，定义了异常处理的exceptions体系以便在程序运行期间抛出异常信息。与此同时，还定义了facet类和locale类以解决多国文化的特殊字符集的处理。

理问题，facet类是描述某个文化背景所特有的信息，其中包括日期、时间、数值和货币等的表示法以及某个国家或民族字符集中的字符串排列顺序，信息代码与自然语言之间的映射关系等。例如一个为中国设计的locale类对象，其内包含一系列的中国文化特色的facet列表，它们用来描述中文字符串如何排序，如何用中国人习惯的形式读写日期、时间、数值和货币等，而一个针对美国人设计的locale类对象，就会以适合美国人的方式来完成这些工作，即C++允许同一个程序有多个locale类对象同时存在，使得同一程序内的不同部分采用不同的字符集处理方式。尽管如此，C++新标准流库仍然保留着老标准流库的所有功能，它仍然包含有流操作（stream）、流缓冲器（stream buffer）和操作符（manipulators）等机制以及cin、cout、cerr和clog等对象。因此，C++新标准流库提供了更灵活的弹性机制和更强大的功能，便于编程者忽略复杂的实现细节而出色、快捷地完成想做的工作。

图7.8中最顶层定义了一个ios_base流类，它保存格式控制信息，并包含用于输入/输出流操作的一些公用的成员函数，它们都与模板参数无关。由ios_base流类（普通类）派生出basic_ios类模板，该类模板定义了与模板参数有关的格式控制信息以及用于输入/输出流操作的一些公用的成员函数，这是典型的由普通类（ios_base流类）派生出一个类模板（basic_ios）的范例，其设计方法是把与虚拟类型无关的成员从basic_ios类模板中分离出来构成一个普通类ios_base，即：

```
class ios_base {
public:
    class failure;
    /* 为本类服务的成员类，所有异常信息都是由本类派生出的basic_ios类模板成员函数
clear()
    抛出 */
    typedef T1 fmtflag;          // 给格式控制标志类型T1定义一个新类型名fmtflag
    static const fmtflags boolalpha, dec, fixed, hex, internal, left,
        oct, right, scientific, showbase, showpoint, showpos,
skipws,
        unitbuf, uppercase, adjustfield, basefield, floatfield;
    // 定义一系列fmtflags类型的常量作为各种格式控制标志
    typedef T2 iostate;          // 给输入/输出流的状态T2定义一个新类型名iostate
    static const iostate badbit, eofbit, failbit, goodbit;
    // 定义一系列iostate类型的常量作为流的各种状态标志
    typedef T3 openmode;
    // T3为枚举型，描述这样一个对象，它能保存各种输入/输出流的打开模式
    static const openmode app, ate, binary, in, out, trunc;
```

```
// 定义一系列openmode类型的常量作为文件流的各种打开模式标志
typedef T4 seek_dir;
/* T4为枚举型，描述这样一个对象，它能保存定位模式，用来作为各种输入/输出流类成员函数
的
```

```
    参数 */

static const seek_dir beg, cur, end;
// 定义一系列seek_dir类型的常量作为各种定位模式标志

typedef T5 event;
/* T5为枚举型，描述这样一个对象，它能保存回调事件（callback event），用来作为用成
员
```

```
    函数ios_base::register_callback( )注册的回调函数参数 */

static const event copyfmt_event, erase_event, copyfmt_event;
// 定义一系列event类型的常量作为各种事件的值

class Init;
    ...
protected:
    ios_base( );           // 保护构造函数
};
```

而由ios_base类继承而来的basic_ios <E, T>类模板通过模板形参E和T的确定而实例化为一个模板类的对象，用来协助ios_base类控制流操作，即：

```
template < class E, class T = char_traits <E> >
    class basic_ios : public ios_base {
public:
    typedef E char_type;           // E为C语言的char型字符

    typedef T::int_type int_type;
    /* int_type为整数类型，它描述这样一个对象，能控制任意形式的数据序列，并通过成员函数
        eof()返回一个整型数值，该值仅类型转换成整型而数值保持不变 */

    typedef T::pos_type pos_type;
    /* 把类型名"T::pos_type"简化成pos_type，它是一个不透明类型，描述这样一个对象，能
        在一个流中保存任意文件位置指示器（即get指针和put指针）的信息 */

    typedef T::off_type off_type;
    /* 把类型名"T:: off_type"简化成off_type，它是一个带符号的整数类型，描述这样一个
        对
        象，能在各种流中保存定位操作的偏移量（一个字节） */
```

```
explicit basic_ios( basic_streambuf < E, T > * sb );
virtual ~basic_ios( );
void clear( iostate state = goodbit );
```

```

// 按形参state清除流状态标志，若流操作失败自动抛出failure类的对象
void setstate( iostate state );
// 按形参state设置流状态标志

bool good( ) const;
// 检测goodbit位为1即所进行的流操作成功时返回true，否则返回false

bool eof( ) const;
// 检测eofbit位为1即流操作进行到文件结尾时返回true，否则返回false

bool fail( ) const;
// 检测failbit位为1即流操作失败时返回true，否则返回false

bool bad( ) const;
// 检测badbit位为1即流缓冲器完整性被破坏时返回true，否则返回false

...
protected:
    basic_ios( );
    void init( basic_streambuf < E, T > * sb );
};

```

当basic_ios <E, T>类模板 中的形参 E 指定为 char 型，T 采用默认值为 char_traits<char>时，则产生了该类模板的一个实例，即编译系统自动地生成相应的类定义代码建立一个模板类，并用typedef语句把该模板类启用一个新类型名ios，即：

```
typedef basic_ios < char, char_traits <char> > ios;
```

其中，char型即为单字节字符，而char_traits型是具有各种字符特征的类型，例如单字节字符型、整数型和其他形式的数据类型等用一个名称为“char_traits”结构类型描述。顺便指出，这种类模板实例化为模板类的关系在图7.5中采用虚线表示。

basic_ios <E, T> 类模板继续派生出basic_istream类模板和basic_ostream类模板，前者实例化出一个istream（模板）类，它继续派生出istrstream类；后者实例化为ostream（模板）类，它继续派生出ostrstream类。这由如下程序代码可清楚看出：

```

class istrstream : public istream { ... };
class ostrstream : public ostream { ... };

```

然而，basic_istream类模板又派生出basic_ifstream类模板，并实例化生成出一个ifstream（模板）类；basic_ostream类模板又派生出basic_ofstream类模板，并实例化生成出一个ofstream（模板）类。

正如图7.5所示，basic_iostream类模板是一个多继承派生类模板，即：

```
template < class E, class T = char_traits <E> >
```

```

class basic_istream : public basic_istream < E, T >,
                      public basic_ostream < E, T > {
public:
    explicit basic_istream( basic_streambuf <E, T> * sb );
    virtual ~ basic_istream();
};

```

它既继承了basic_istream类模板,又继承了basic_ostream类模板。正如5.5节所述,在多继承中为了避免模棱两可的“二义性”问题,应将它的顶层类basic_ios设置为虚基类,即:

```

template < class E, class T = char_traits < E > >
class basic_istream : virtual public basic_ios < E, T > { ... };

```

和

```

template < class E, class T = char_traits < E > >
class basic_ostream : virtual public basic_ios < E, T > { ... };

```

basic_istream类模板实例化生成一个istream(模板)类。该类模板再派生出basic_fstream类模板,派生类模板实例化生成了一个fstream(模板)类。

C++新标准流库对缓冲器机制也进行了模板化,首先定义了一个流缓冲器类模板basic_streambuf,即:

```

template < class E, class T = char_traits <E> >
class basic_streambuf {
    ...
protected:
    basic_streambuf( ); // 无参数构造函数
    E * eback( ) const;
    // 返回到指向模板形参E所确定类型的一个对象即输入流缓冲器头部的指针
    E * gptr( ) const;
    // 返回到指向模板形参E所确定类型的一个对象即输入流缓冲器中下一个要读取元素的指针
    E * egptr( ) const;
    // 返回到指向模板形参E所确定类型的一个对象即输入流缓冲器尾端的指针
    E * pbase( ) const;
    // 返回到指向模板形参E所确定类型的一个对象即输出流缓冲器头部的指针
    E * pptr( ) const;
    // 返回到指向模板形参E所确定类型的一个对象即输出流缓冲器中下一个要写入元素的指针
    E * epptr( ) const;
    // 返回到指向模板形参E所确定类型的一个对象即输出流缓冲器尾端的指针
    ...
};

```

```
};
```

该模板描述一个基于流缓冲器 (stream buffer) 的抽象基类, 它控制一个流操作的特殊动作和流的元素序列, 当该类模板的形参E和T确定后实例化为一个模板类的对象, 用以实现缓冲器机制来完成对一个流的控制。每个流缓冲器都控制着两个独立的流, 一个是进行提取 (extractions, 也称读取) 操作的输入流, 另一个是做插入 (insertions, 或称写入) 操作的输出流。程序在运行时, 有时只进行输入流的读取操作, 或者只做输出流的写入操作, 有时两者都需要做, 那么, 流缓冲器就专司控制这两个流操作之间的协调配合。basic_streambuf类模板公有部分的成员函数提供了流缓冲器所需的所有公用操作, 其保护部分的成员函数则提供了一个流所做工作的一些特殊要求的操作。例如, 对于输入流最具代表性的几个保护成员函数有:

eback(): 该函数返回到指向该输入流缓冲器头部的指针, 简称“输入流头部指针”;

gptr(): 该函数返回到指向输入流缓冲器中下一个要读取元素的指针, 简称“输入流下一个指针”, 即get指针;

egptr(): 该函数返回到指向该输入流缓冲器尾端的指针, 简称“输入流尾部指针”。

对于输出流最具代表性的几个保护成员函数有:

pbase(): 该函数返回到指向该输出流缓冲器头部的指针, 简称“输出流头部指针”;

pptr(): 该函数返回到指向输出流缓冲器中下一个要写入元素的指针, 简称“输出流下一个指针”, 即put指针;

epptr(): 该函数返回到指向该输出流缓冲器尾部的指针, 简称“输出流尾部指针”。

所有缓冲器即使用户自行创建的缓冲器以及basic_streambuf< E, T>类模板的任意保护虚成员函数都应遵循如下协议:

若下一个指针, 如输入流的gptr指针或输出流的pptr指针不是空指针, 则输入流所有3个指针 (或输出流所有3个指针) 都应指向同一个元素序列, 否则该缓冲器不存在;

对于输出流, 若下一个指针pptr小于缓冲器尾部指针epptr, 则可在下一个指针pptr所指定的位置存入一个元素;

对于输入流, 若下一个指针gptr小于缓冲器尾部指针egptr, 则可在下一个指针gptr所指定的位置读取一个元素;

对于输入流，若缓冲器头部指针eback小于下一个指针gptr，则可将下一个指针gptr减1后所指定的位置回送一个元素（put back an element），即退回到原来的位置读取上一个元素。

basic_streambuf< E, T>类模板实例化成类模板的一个对象，它就是一个流缓冲器保存着上述这6个指针，它还保存有描述各种文化背景所特有信息的locale类对象以满足流缓冲器各种应用场合的操作要求。

basic_streambuf<E, T>类模板还派生出basic_filebuf类模板，即：

```
template < class E, class T = char_traits <E> >
class basic_filebuf : public basic_streambuf < E, T > {
    ...
protected :
    virtual int_type underflow( );
    // 即下溢操作，从输入流中读取当前元素c，成功时返回到该元素c，否则返回到文件结
尾

    virtual int_type overflow( int_type c = T::eof( ) );
    /* 即上溢操作，把作为形参的元素值c写入到一个输出流，成功时返回到非文件结尾，否
则

    返回到文件结尾 */
};
```

它实例化的模板类对象也是一个流缓冲器，其功能是控制一个存放在外部文件中的元素序列的读取和写入流操作。另外，它还保存着一个文件指针（file pointer），指向用open操作打开的一个与流相关联的文件对象。

C++新标准程序库为增强应用的灵活性大量地采用了模板化，例如针对各种不同字符类型，并考虑到一些特殊字符如“end-of-file”字符以及不同文化背景字符集合的处理细节上有所不同，引入了一个string（字符串）的概念。首先定义了basic_string类模板，即：

```
template < class E, class T = char_traits <E>, class A = allocator<T>
>
class basic_string { ... };
```

该类模板实例化的模板类对象，能控制模板形参E类型所指定的各种长度不同的元素序列，这些元素能作为basic_istream和basic_ostream类模板形参E对应的实参。在C++新标准程序库中提供两种不同字符类型的实例化模板类，即对char类型定义为string，对wchar_t类型则定义为wstring，即：

```
typedef basic_string < char > string;
```



```
typedef basic_string < wchar_t > wstring;
```

第1种是把该类模板的形参E指定为char，其他两个形参T和A取用其默认值，所得实例化的模板类名即为“basic_string<char,char_traits <char>, allocator<char>>”，然后用typedef语句将这个很长的模板类名启用一个新类型名string。所以若编程者处理像C语言那样的字符串，选用string对象即可，它不仅能完全替代字符串指针“char *”，还可以用面向对象程序设计技术高效、灵活地处理字符串，例如，自动实现动态内存分配和释放，重载加法运算符实现字符串的串接和能方便地转换成“char *”等。同时，有些string对象采用引用链接技术，这比采用字符串指针“char *”在时间和空间上具有更好的效率。第2种是把该类模板的形参E指定为wchar_t，该类型是广义字符类型，即包含上述的不同文化背景字符集合和很特殊的字符，其他两个形参T和A取用其默认值，所得实例化的模板类，因此，wstring对象是弹性更大、功能更强的模板类对象。

为了在处理各种字符串的输入/输出流操作中引入缓冲器机制并进行模板化，C++新标准流库还定义了一个basic_stringbuf类模板，即：

```
template < class E, class T = char_traits <E>, class A = allocator<E> >
class basic_stringbuf { ... };
```

它仍然是一个流缓冲器，用来控制存放在对象数组中元素序列的读取/写入流操作，缓冲器内存空间的分配、扩展和释放都必须按照该元素序列的存放格式。

对于初学者在了解了C++新标准流库的类层次结构后，接着应掌握经常使用的如下相关头文件：

< iostream >：它是使用最多的标准头文件，几乎所有的C++源程序都必须包含它，因为只要用到cout和cin预定义流对象进行标准设备的输入/输出流操作，就必须用#include语句把该头文件引入到当前源程序中。

< fstream >：其内定义了一些类模板，它们支持对存放在外部文件例如磁盘文件中的数据序列进行输入/输出流操作，当使用文件流对象进行外部文件的读取和写入操作时，就必须用#include语句把该头文件引入到当前源程序中。

< stringstream >：该标准头文件中定义了一些class类型，它们支持存放在char型字符串数组中数据序列的输入/输出流操作。这些数据序列能很方便地转换成C语言的字符串。当使用字符串流对象针对存放在内存空间内的char型字符串进行输入/输出流操作时，就必须用#include语句把该头文件引入到当前源程序中。

< sstream >：其内定义了一些管理和操作字符串容器类的输入/输出流操作

的类模板，支持存放在所分配的对象数组中的序列化对象，这些对象与类模板 `basic_string` 的对象都能方便地互相转换。当使用字符串流对象针对存放在对象数组中的序列化对象进行输入/输出流操作时，就必须用 `#include` 语句把该头文件引入到当前源程序中。

< `iomanip` >: 其内定义了一组称为“操作符 (manipulators)”的函数，例如，设置输入/输出字符宽度的 `setw()` 函数，设置浮点数精度的 `precision()` 函数。当进行输入/输出流格式控制时，若采用与输入/输出运算符 “>>” 和 “<<” 配合使用的操作符函数，就必须用 `#include` 语句把该头文件引入到当前源程序中。

7.1.3 4个预定义标准流对象

C++新标准流库中，由系统预先定义了 `cin`、`cout`、`cerr` 和 `clog` 等4个流对象，称为“预定义标准流对象 (Predefined standards stream object)”。

过去经常使用的 `cout` 语句，其左运算量 `cout` 的类型是 `ostream`，它不是类名，而是 `basic_ostream` 类模板实例化的一个模板类（模板形参 `E` 指定为 `char` 型）用 `typedef` 语句所启用的新类型名，在C++新标准流库中该类型的定义为：

```
typedef basic_ostream < char, char_traits<char> > ostream;
```

而这4个预定义标准流对象的定义以及与之相关联的标准外部设备如下：

```
extern istream cin; // 与标准输入设备（键盘）相关联的流对象，类似于C语言的stdin
```

```
extern ostream cout;
// 与标准输出设备（显示器）相关联的流对象，类似于C语言的stdout
```

```
extern ostream cerr;
/* 与标准出错信息输出设备（显示器）相关联的流对象，类似于C语言的stderr，发送给它的内
```

```
容立即输出，无缓冲器机制 */
```

```
extern ostream clog;
// 与标准出错信息输出设备（显示器）相关联的流对象，具有缓冲器机制
```

如前所述，C++新标准流库对外部设备也引入了缓冲器机制并进行模板化，把实际的外部设备抽象成为“输入/输出流类的对象”，且将经常使用的外部设备定义为I/O流类的标准化流对象，简称标准流对象。例如，把输入标准流对象 `cin` 定义为 `basic_istream` 类模板实例化（模板形参 `E` 指定为 `char` 型）的一个模板类 `istream` 的对象，作为键盘输入数据的源点，而把输出标准流对象 `cout`、`cerr` 和 `clog` 定义成 `basic_ostream` 类模板实例化（模板形参 `E` 指定为 `char` 型）的一个模板类 `ostream` 的对象，作为输出到显示器的数据（包括错误信息）的终点，显然，它们处理的数据都是

char型字符即字节流。今后，为了叙述方便，我们就把istream和ostream新类型名分别称为istream流类和ostream流类，但读者必须知道它们是由相应的类模板实例化而得到的模板类。

程序启动时，系统自动建立这4个标准流对象，并把cin连接到键盘，cout连接到CRT，cerr和clog经不同的缓冲方式也连接到显示器，使得输出到cerr和clog的错误信息总是显示在显示器的屏幕上。当程序执行完成时，系统自动撤销这4个标准流对象。这是由系统预先自动设定的，编程者不需编写任何操作语句。

7.2 输 出 流

左移运算符“<<”在ostream类中被重载为一个输出运算符，也称插入(Inserting)运算符，流的输出就是用插入的方法实现的。它有两个运算量，在它左边的运算量为ostream类，例如预定义标准流对象cout，右运算量可以是任何类型的对象，如基本数据类型char、int、float和double等，也可以是用户自行定义，并重载了该运算符“<<”的class类型的一个对象。

7.2.1 内部数据类型的输出

顺便指出，通过查看iostream库有关文件可知，输入运算符“>>”在istream流类中重载，输出运算符“<<”在ostream流类中重载。在使用C++新标准iostream库的源程序开头处必须写有：

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

详细内容可参看1.3节。

7.2.2 ostream类中的主要成员函数

在<ostream>新标准头文件（包含在<iostream>头文件中，很少直接使用#include语句包含到当前程序中）中，定义了一个类模板basic_ostream，它从basic_ios类模板继承而来，即：

```
template < class E, class T = char_traits<E> >
class basic_ostream : virtual public basic_ios < E, T > {
public:
    ...
    basic_ostream & put( E c );
    /* 向输出流中输出一个字符，其返回值是ostream类的对象引用，模板形参E实例化为
```

char

```
    型 */  
    basic_ostream & write( E * s , streamsize n );  
    /* 向输出流中输出一个字符串指针s所指的、字符个数为n的字符串，通常streamsize类  
       型实际上是int型，模板形参E实例化为char型，其返回值是ostream类的对象引用*/  
    basic_ostream & flush();  
    pos_type tellp();  
    // 返回到put指针的当前位置  
    basic_ostream & seekp( pos_type pos );  
    // 把put指针移动到形参pos指定的绝对位置上  
    basic_ostream & seekp( off_type off, ios_base::seek_dir way );  
    // 把put指针移动到偏移值为形参off所指定的相对位置上  
};
```

若将它的模板形参E指定为char型，而T采用默认值char_traits<char>时，该类模板实例化成“basic_ostream < char, char_traits<char> >”形式的模板类，为了书写方便再用一个typedef语句给这样长的模板类名启用一个新类型名ostream，即：

```
typedef basic_ostream < char, char_traits<char> > ostream;
```

因此，以后经常就把这个实例化的模板类简称为ostream类，但读者应特别明确，在C++新标准流库中的ostream类是一个模板形参为char型的实例化模板类名的简称。由该模板类的定义可知，put()和write()这两个成员函数是ostream(模板)类在流库中所提供的主要输出操作，是编程者使用流库的主要界面之一。

成员函数put()。要输出二进制数据（以字节为单位）或单个字符，可使用basic_ostream< char, char_traits<char> >类模板的成员函数put()。实参char替代形参E后系统生成的模板类成员函数的原型声明为：

```
ostream & ostream::put( char c );  
// 向输出流中输出一个字符，成员函数put( )的返回值是ostream类的对象引用  
若有“int ch = 'x';”，则调用语句可写为“cout.put(ch);”或“cout <<  
(char)ch;”，也可以直接使用字符常量作为put( )的实参，写成：  
cout.put( 'x' );
```

成员函数write()。它可用来输出一个字符个数为n的字符串，其原型为：

```
ostream & ostream::write( char * ptr , int n );
```

由字符指针ptr所指的字符串，向流中输出n个字符，它经常用来输出二进制数据块。

7.2.3 用户定义的class类型的输出

C语言的I/O操作中没有提供对用户自行定义类型的支持，例如，可以在C语言中定

义一个结构体Point和结构变量p：

```
struct Point {
    float x, y;
} p;
```

但是没有办法扩展C的I/O操作功能，得到直接输出结构变量p的表达式，不能写成：

```
printf( "struct Point p is % ? \n", p );
```

因此，输出操作的标准函数printf()只能识别系统内部的数据类型，而没有办法把其功能扩展到对用户自行定义类型的支持。在C++中允许编程者重载（重新定义）“<<”运算符，使它能完成对用户自行定义类型的输出，且使用非常灵活方便。

例7.2 在用户自行定义的class类型中，重载输出运算符。

```
#include <iostream> // 使用C++新标准的流库
using namespace std; // 将std标准名空间合并到当前名空间

class Point {
    float x, y, z;
    // 3个私有数据成员x、y和z分别记录三维空间点的坐标值

public :
    Point( float i, float j, float k ) // 构造函数
    {   x = i;   y = j;   z = k;   }
    // 重载输出运算符函数

    friend ostream & operator << ( ostream & os, Point p )
    {   os << "(" << p.x << " , " << p.y << " , " << p.z << " )\n";
        // 以"(x , y , z)"格式输出显示点对象的坐标值

        return os;
        // 返回到标准输出流类ostream的一个对象引用os，即cout
    }
};

void main( void )
{   Point a(1, 2, 3), b(4, 5, 6), c(7, 8, 9);
    // 创建Point类的3个点对象a、b和c

    cout << a << b << c; // 输出显示3个点对象a、b和c的坐标值
}
```

该程序的输出结果为：

```
(1 , 2 , 3)
(4 , 5 , 6)
(7 , 8 , 9)
```

在Point 类中重载的输出运算符“<<”仍然是一个双目运算符。有两个运算量，对

应于operator<<()重载运算符函数的第一个形式参数os是标准输出流类ostream的一个对象引用,它出现在输出运算符的左边。第二个形式参数出现在输出运算符的右边,为程序中待输出的对象。例如:"cout << a;" ,按照重载运算符的转换规则,上式可改写为:

```
operator << ( cout, a );           // 重载运算符函数实际的调用语句
```

编译系统将它解释为:将Point类的一个对象a输出到标准输出流对象cout(即显示器)中去。

重载输出运算符"<<"的格式为:

<pre>ostream & operator << (ostream & os , Point p) { 相 对于该类对象的输出操作; return os; }</pre>
--

在main()函数体内的"cout<<a;"语句中,引起流操作的对象是a,而a为Point类的对象,因此,编译系统会自动调用Point类中的重载输出运算符函数operator<<()。它的第一个形式参数必须定义为输出流对象的引用ostream & os,而把要输出的对象作为第二个形式参数,使用值传递"Point p"或引用传递"Point & pr"都可以。

重载输出运算符函数的返回值必须定义为所用输出流类对象的引用ostream &,其目的是当几个输出运算符连在一起使用时,该重载输出运算符函数按从左至右的结合规则顺利执行。例如:

```
cout << a << b << c;           (*)
```

按结合规则先执行"cout << a;" ,则可写成实际调用重载输出运算符函数的语句形式:

```
operator << ( cout, a );
```

而该函数的原型为:

相当于执行了"ostream & os=cout;"

```
ostream & operator << (ostream & os , Point p);
```

在实参传递给形参的过程中,相当于执行了对输出流类对象的引用变量os的初始化操作,使得在该函数体内的os 就是cout。因此,该函数的结尾处必须写有一条返回语句"return os;" ,这就使得该函数返回到cout,这说明表达式 (cout << a) 的结果应为cout,则式(*)变为:"cout << b << c;" ,接着执行"cout << b;"后变为"cout << c;"。顺便指出,Point类的重载输出运算符函数体也可写成如下简洁形式:

```
ostream & operator << ( ostream & os , Point p )  
{ return os << "(" << p.x << " , "<< p.y << " , "<< p.z << " )\n"; }
```

对于Point这种class类型(流的终点为cout),必须将重载输出运算符函数

operator <<() 定义成Point类的友元函数，而不能定义为成员函数。如果定义为成员函数，则写成：

```
ostream & operator << ( Point p ) { ... }
```

则“cout << a;”可改写为：

```
cout.operator << ( a );
```

该语句表示引起“<<”流操作的对象是cout，这显然是不合理的。实际上引起“<<”流操作的对象是a，而不是cout。因此，应将重载输出运算符函数定义为友元函数。

7.3 输 入 流

输入流与输出流类似，右移运算符“>>”在istream类中被重载为一个输入运算符，也称提取(Extracting)或读取运算符，流的输入就是用提取的方法实现的。它也有两个运算量，在它左边的运算量为istream类，例如预定义标准流对象cin，其右边的运算量可以是任何类型的对象，如基本数据类型char、int、float和double等，也可以是用户自行定义，并重载了该运算符“>>”的class类型的一个对象。

内部数据类型输入操作的编程方法请参看1.3节。

7.3.1 istream类中的主要成员函数

与输出流类同，在<istream>新标准头文件（包含在<iostream>头文件中，很少直接使用#include语句包含到当前程序中）中，定义了一个类模板basic_istream，它从basic_ios类模板继承而来，即：

```
template <class E, class T = char_traits<E> >
class basic_istream : virtual public basic_ios<E, T> {
public:
    class sentry;

    explicit basic_istream( basic_streambuf < E, T > * sb );
    virtual ~ istream();    // 虚析构函数
    // 一组重载的输入运算符函数

    basic_istream & operator >> ( int & n );
    basic_istream & operator >> ( double & n );
    ...

    streamsize gcount( ) const;
    // 一组重载的get( )成员函数

    int_type get( );
    basic_istream & get( E & c );
```

```

/* 当模板形参E实例化为char，该成员函数是从输入流中读取单个字符存放在形参c被引
用的字符变量中 */

basic_istream & get( E * s, streamsize n, E delim );
/* 当模板形参E实例化为char，streamsize类型即为int型，该成员函数是从输入流
istream(例如键盘)中读取字符个数为形参n的一个字符串存放在形参s所指的流缓冲
器内 */

...

// 两个重载的getline( )成员函数

basic_istream & getline( E * s, streamsize n );
/* 当模板形参E实例化为char，streamsize类型即为int型，该成员函数是从输入流中
读取一行文本字符，存放在形参s所指的流缓冲器内 */

...

basic_istream & ignore( streamsize n = 1,
                        int_type delim = T::eof() );
// 读取并丢弃从当前位置开始的n个字符，返回到调用本成员函数的对象

int_type peek( );
// 返回当前位置上的字符，但get指针所指的输入位置不变

basic_istream & read( E * s, streamsize n );
// 读取形参n所指定字节数的数据块存放在形参s所指定的字符空间中，

basic_istream & putback( E c );
// 将形参c所指定的字符回送给输入流

pos_type tellg( );
// 返回到get指针的当前位置

basic_istream & seekg( pos_type pos );
// 把get指针移动到形参pos指定的绝对位置上

basic_istream & seekg( off_type off, ios_base::seek_dir
way );
// 把get指针移动到偏移值为形参off所指定的相对位置上

...

};

```

若将它的模板形参E指定为char型，而形参T采用默认值char_traits<char>时，该类模板实例化成“basic_istream < char, char_traits<char> >”形式的模板类，为了书写方便再用一个typedef语句给这样长的模板类名启用一个新类型名istream，即：

```
typedef basic_istream < char, char_traits<char> > istream;
```

因此，以后经常就把这个实例化的模板类简称为istream类，但读者应特别明确，在C++

新标准流库中的`istream`类是一个模板形参为`char`型的实例化模板类名的简称。由该模板类的定义可知，`istream`类在流库中执行的主要是输入操作，首先要用到与输入流操作相对应的两个成员函数`get()`和`read()`。

在`istream`类中重载了一系列适用于不同情况的`get()`，它是`basic_istream<char, char_traits<char> >`模板类的成员函数。当把实参`char`替代形参`E`后系统自动生成的模板类成员函数的原型声明如下，其中包括：

```
istream & get( char * s , int n, char delim );  
...
```

从输入流`istream`（例如键盘）中读取一个字符串存放到形参`s`所指的缓冲器内，直到遇到EOF（End of File，文件结尾）或读入了`n - 1`个字符，或者遇到由`delim`指定的终止字符为止。顺便指出，`basic_istream<E, T>`类模板定义中的`streamsize`类型就是`int`型，下同。

在`istream`类中还有另一种与 类似的两个成员函数，它们经常用来读取文本文件中的一行字符，当把实参`char`替代形参`E`后系统自动生成的模板类成员函数的原型声明为：

```
istream & getline( char * s , int n, char delim );  
...
```

它们的功能都是从输入流中读取一行文本字符的成员函数，存放到形参`s`所指的缓冲器内，直到遇到EOF（End of File，文件结尾）或读入了`n - 1`个字符，或者遇到由`delim`指定的终止字符为止，返回到“*this”即调用本成员函数的对象。它们与`get()`的区别是，`get()`不读取每行的终止字符，而`getline()`读取每行的终止字符。

输入流中还重载了读取单个字符并存放在形参`c`被引用的字符变量内的成员函数，实例化后的原型声明是：

```
istream & get( char & c );  
...
```

由此可知，该形参`c`被指定为引用，实际调用时可写为：

```
char ch;  
...  
cin.get( ch );
```

输入流操作中另一类重要的成员函数`read()`对应于实例化后的`ostream`（模板）类中的`write()`，实例化后的原型声明是：

```
istream & read( char * s , int n );  
...
```

其功能是从输入流中读取一个字符串以数组的形式存放到s所指的缓冲器内，返回到“*this”即返回到调用本成员函数的对象。若读取操作在文件结尾前停止，说明操作出错，该成员函数自动调用“setstate(failbit)”把读取操作失败标志failbit设置为1。它常用来读二进制数据块，该数据块的长度作为它的第2个参数，由计算类型长度表达式sizeof(type)求得，type则为所读二进制数据块的类型。因此，它可读取任何类型的数据。当用该成员函数将字符输入到字符缓冲器中时，必须由第2个参数指明缓冲器的大小。

7.3.2 用户定义类型的输入

与输出流一样，必须重载输入运算符“>>”，其格式为：

```
istream & operator >> ( istream & is, Point & p )
{
    相对于该类对象的输入操作;
    return is;
}
```

在例7.2中，若考虑用键盘敲入Point类的x、y和z的数值，则应在类定义中重载输入运算符“>>”。

例7.3 在用户自行定义的class类型中，重载输入运算符。

```
#include < iostream > // 使用C++新标准的流库
using namespace std; // 将std标准名空间合并到当前名空间
class Point {
    float x, y, z;
    // 3个私有数据成员x、y和z分别记录三维空间点的坐标值
public :
    Point( float i, float j, float k ) // 构造函数
    {
        x = i; y = j; z = k;
    }
    // 重载输出运算符函数
    friend ostream & operator << ( ostream & os , Point p )
    {
        os << "(" << p.x << " , " << p.y << " , " << p.z << " )\n";
        // 以“(x , y , z)”格式输出显示点对象的坐标值
        return os;
        // 返回到标准输出流类ostream的一个对象引用os，即cout
    }
    // 重载输入运算符函数
    friend istream & operator >> ( istream & is , Point & p )
    {
        cout << " X Coordinates = "; is >> p.x;
        /* 先输出提示信息，再将键盘敲入的点对象的x坐标值存放在点对象引用p所指对象
```

的

```

        私有数据成员x中 */
        cout << " Y Coordinates = ";    is >> p.y;
        /* 先输出提示信息，再将键盘敲入的点对象的y坐标值存放在点对象引用p所指对象
        的
        私有数据成员y中 */
        cout << " Z Coordinates = ";    is >> p.z;
        /* 先输出提示信息，再将键盘敲入的点对象的z坐标值存放在点对象引用p所指对象
        的
        私有数据成员z中 */
        return is;
        // 返回到标准输入流类istream的一个对象引用is，即cin
    }
};

void main( void )
{
    Point    a( 1, 2, 3 );    // 定义Point类的自动对象a
    cout << a;                // 输出显示点对象a的坐标值
    cin >> a;                  // 用键盘给点对象a重新敲入坐标值
    cout << a;                // 再次输出显示点对象a的坐标值
}

```

该程序的输出结果为：

```

(1 , 2 , 3)
X Coordinates = 4(CR)
Y Coordinates = 5(CR)
Z Coordinates = 6(CR)
(4 , 5 , 6)

```

与输出运算符函数类似，重载输入运算符函数必须定义为Point类的友元函数，而不能是成员函数。它的返回值必须定义为所用输入流类对象的引用istream&。它应具有两个参数，第1个形参必须定义为输入流对象的引用istream & is。

重载输入运算符函数的第2个形参必须是Point类对象的引用“Point & p”，不能是“Point p”。即实参传递给形参的方式是引用传递（实质上是地址传递方式），不能是值传递方式。如果将第2个形参定义为Point p，则实参传递给形参的方式是赋值方式（即单方向的值传递）。这是因为重载输入运算符函数的原型为：

```
istream & operator >> (istream & is, Point p);
```

在main()函数内的调用语句：

```
cin >> a;                                相当于执行了Point p = a;
```

按运算符函数的转换规则可改写为：

```
operator >> ( cin , a );
```

由于形参对象`p`和实参对象`a`不在同一个作用域内，`a`的作用域是`main()`函数体，`p`的作用域是重载输入运算符函数体内，它们的存储空间是两个实体。如前所述，函数参数的“值传递方式”是单方向传递数据，这与C语言中基本数据类型作为函数参数进行“值传递方式”的数据单方向传递是一致的，只能由实参对象`a`传给形参对象`p`，即如3.3节所述，此时将调用复制构造函数（本例程中的`Point`类没有编写复制构造函数则调用系统提供的默认复制构造函数），将实参对象`a`“位模式拷贝”给形参对象`p`，即形参对象`p`是实参对象`a`的一个副本。接着在进入到该重载输入运算符函数体内，对形参对象`p`所进行的各种操作，改变形参对象`p`的数据成员值，但它不能传回给实参对象`a`。这就相当于我们把一个文件用复印机复制了一份“拷贝件”，而我们在“拷贝件”上把内容修改了，但原件还是原来的内容。因此在调用重载输入运算符函数，进行参数传递的过程中，实参`a`可以传递给形参`p`，但在该函数体内键盘敲入的新数据改变了形参`p(x, y, z)`内的值，却无法传回给实参`a`中的`x`、`y`和`z`，所以`main()`函数内的第3条语句的输出结果仍然是`a(1, 2, 3)`。

若把第2个形参定义为“`Point & p`”，则当调用重载输入运算符函数时，实参传给形参采用地址传递方式，相当于执行了一条“`Point & p = a;`”语句，使得形参对象`p`是实参对象`a`的引用，即形参对象`p`就是实参对象`a`的一个替换名，它们的存储空间是同一个实体。虽然形参对象`p`和实参对象`a`不在同一个作用域内，`a`的作用域是`main()`函数体，`p`的作用域是重载输入运算符函数体内，但是这就像接力赛跑一样把接力棒由实参对象`a`传递给了形参对象`p`，那么，在重载输入运算符函数体内，凡是对形参对象`p`的输入流操作实际上就是对实参对象`a`的输入流操作。因此，从键盘敲入的数据才能修改实参对象`a`中的数据成员`x`、`y`和`z`的值。

与C语言中的`scanf()`函数使用方法类似，可输出“提示字符串”改善人机界面。例如，可写为：

```
cout << " x Coordinates = ";
is >> p.x;
cout << " y Coordinates = ";
is >> p.y;
...
```

甚至还可控制键盘敲入的字符必须是数字字符，以下程序能使读入的数据跳过非数字字符。

例7.4 例7.3的改进版本。

```
#include < iostream > // 使用C++新标准的流库
#include < cctype > // 宏指令isdigit(c)声明在其中
```

```

using namespace std;           // 将std标准名空间合并到当前名空间
// 跳过非数字字符的处理函数

void skipNonDigit( istream & is )
{   char c;           // 定义一个字符型变量c接收键盘敲入的字符
    while( 1 ) {      // 无穷循环语句，但接收到数字字符后在返回
        is >> c;
        if( isdigit(c) ) {
            is.putback(c);
            return;
        }
        /* 当c为数字字符时，putback( )将字符c送回到输入流对象is的缓冲区后返回，
           否则继续接收键盘输入字符 */
    }
}

class Point {
    float x, y, z;
    // 3个私有数据成员x、y和z分别记录三维空间点的坐标值
public :
    Point( float i, float j, float k ) // 构造函数
    {   x = i;   y = j;   z = k;   }
    // 重载输出运算符函数

    friend ostream & operator <<( ostream & os , Point p )
    {   os << "(" << p.x << " , " << p.y << " , " << p.z << " )\n";
        // 以"(X , Y , Z)"格式输出显示点对象的坐标值

        return os;
        // 返回到标准输出流类ostream的一个对象引用os，即cout
    }
    // 重载输入运算符函数

    friend istream & operator >> ( istream & is , Point & p )
    {   cout << " X Coordinates = ";
        skipNonDigit(is);      is >> p.x;
        /* 先输出提示信息，再将键盘敲入的点对象的x坐标值存放在点对象引用p所指对象
        的
        私有数据成员x中，键盘每敲入一个字符经跳过非数字字符处理函数过滤掉非数字
        字
        符 */
        cout << " Y Coordinates = ";

```

```

        skipNonDigit(is);          is >> p.y;
/* 先输出提示信息，再将键盘敲入的点对象的y坐标值存放在点对象引用p所指对象
的

        私有数据成员y中，键盘每敲入一个字符经跳过非数字字符处理函数过滤掉非数字
        字
        符 */
        cout << " Z Coordinates = ";
        skipNonDigit(is);          is >> p.z;
/* 先输出提示信息，再将键盘敲入的点对象的z坐标值存放在点对象引用p所指对象
的

        私有数据成员z中，键盘每敲入一个字符经跳过非数字字符处理函数过滤掉非数字
        字
        符 */
        return is;
// 返回到标准输入流类istream的一个对象引用is，即cin
    }
};

void main( void )
{
    Point a(1, 2, 3);           // 定义Point类的自动对象a
    cout << a;                  // 输出显示点对象a的坐标值
    cin >> a;                   // 用键盘给点对象a重新敲入坐标值
    cout << a;                  // 再次输出显示点对象a的坐标值
}

```

该程序编写了一个skipNonDigit()函数，它采用宏指令isdigit(c)判断敲入的字符是否为数字字符，当c为数字字符时，则调用istream类中的成员函数putback()将字符c送回到输入流的缓冲器内，并返回到调用函数。否则，继续等待键盘敲入新的字符。putback()的原型是：

```
istream & istream::putback( char ch );
```

其功能是把字符ch送回到输入流，即放回到istream类的缓冲器内，如果该字符不能被送回，则将流状态置为“失败”。

7.4 格式控制

7.4.1 格式控制的输入/输出和无格式的输入/输出

众所周知，把要处理的数据存放在计算机内部的存储格式与这些数据在外部进行显示、传送或存储的表示形式是不同的，前者应便于存储而后者应便于操作者观察和阅读。

因此，进行输入/输出流操作时就必须要做相应的转换工作，因此对计算机的I/O格式进行控制就是完成这项工作。它针对的是键盘、显示器和打印机等字符型的外部设备以及磁盘中的文本文件。所以，对于格式控制的输入/输出，数据流不管它们是哪种数据类型，从控制的角度看，一律当做每个元素都是字符的序列。例如，一个数值为3.14159的double型变量，若要传送到显示器中显示，一定先转换成为数字字符串变成一个字符序列再进行传送，因此该数据也可以输出到一个文本文件保存，用文本编辑器可随时打开查看。

无格式的输入/输出就与此不同，数据在计算机内部的存储格式与在外部进行传送或存储的表示格式是完全一样的，因此，它只能用于磁盘、光盘或磁带等外存储媒介体中的二进制数据文件。通常，这类文件无法用文本编辑器打开查看（参阅文献[25]）。

对于针对键盘、显示器和打印机等字符型的外部设备以及磁盘中的文本文件，编程者如果没有编写格式控制的输入/输出语句，则系统在进行输入/输出流操作时将采用默认的输入/输出格式。由于C++流库重载的输入运算符">>"和输出运算符"<<"都能自动识别作为运算量的字符序列的数据类型，因此，对于整型和浮点型的基本数据类型（如short、int、long、float和double等）的输入流操作和默认的格式控制，输入流跳过空白符（它包括空格符"Space"、换行符"CR"、水平制表符"Tab"等），从第1个非空白字符开始读取对应于输入变量类型的值，操作者按回车键则系统自动地将敲入的数字字符序列转换成相应的数据类型后赋给作为右值的变量。读输入的过程一直进行到出现一个不合法部分时立即停止，详见1.3.3节。而对于字符串的输入是从第一个空白字符开始读入到下一个空白字符结束。对于输出流操作，整数按一般整数形式，负数前面带“-”号；实型数有定点形式和科学表示法的指数形式，选取两种中字符个数较少者；字符串输出无双引号，以字符常数零'\0'为结尾符，也有单个字符输出的格式。

顺便指出，C++提供了比较灵活的方式，用于控制输入输出的格式。在C语言中通过标准函数printf()和scanf()进行的格式化的I/O操作，在C++中也可以使用，且C++有控制更灵活、功能更强的操作符格式控制。

7.4.2 C++新标准流库的格式控制

在C++新标准流库中，作为流库类模板最顶层的根类ios_base，其内含有一组数据类型为fmtflag的公有静态数据成员，即：

```
class ios_base {
public:
    class failure;
    /* 为本类服务的成员类，所有异常信息都是由本类派生出的basic_ios类模板成员函数
```

```
clear()
    抛出 */
    typedef T1  fmtflag;          // 给格式控制标志类型T1定义一个新类型名fmtflag
    static  const  fmtflags  boolalpha, dec, fixed, hex, internal, left,
                                oct, right, scientific, showbase, showpoint, showpos,
skipws,
                                unitbuf, uppercase, adjustfield, basefield, floatfield;
    // 定义一系列fmtflags类型的常量作为各种格式控制标志
    ...
    fmtflags  flag( ) const;          // 读取所保存的格式控制标志
    fmtflags  flags( fmtflags  fmtfl );
    // 读取形参mask指定的格式控制标志fmtfl, 返回值为以前保存的标志
    fmtflags  setf( fmtflags  fmtfl );    // 设置格式控制标志fmtfl
    fmtflags  setf( fmtflags  fmtfl, fmtflags  mask );
    // 设置形参mask指定的某个字段中的格式控制标志fmtfl, 返回值为设置前保存的标志
    void  unsetf( fmtflags  mask );      // 清除由形参mask指定的某字段或某标
志
    streamsize  precision( ) const;
    // 设置显示精度, 缺省值为小数点后6位, 返回所保存的显示精度值
    streamsize  precision( streamsize  prec );
    // 设置显示精度为prec, 缺省值为小数点后6位, 返回值为设置前的显示精度值
    streamsize  width( ) const;          // 设置域宽为以前保存的值, 返回该域宽
值
    streamsize  width( streamsize  wide );
    // 设置输出数据的域宽为wide, 返回设置前的域宽值
    ...
protected:
    ios_base( );          // 保护部分的无参数构造函数
};
```

各种格式标志名和具体含义如表7.2所示。其中,fmtflags为枚举类型,这些格式

表7.2 格式控制状态标志位

状态标志位	含 义	所 在 字 段
Skipws	跳过输入中的空白符	
Left	输出数据按输出域左对齐	adjustfield
Right	输出数据按输出域右对齐	adjustfield
internal	输出数据在指定域宽内部对齐,即	adjustfield

	符号位左对齐，数值右对齐，中间可填充字符（如空格字符）	
Dec	转换基数为十进制形式	basefield
Oct	转换基数为八进制形式	basefield
Hex	转换基数为十六进制形式	basefield
showbase	输出的数值数据前带有基数符（0表示8进制或0x表示16进制）	
showpoint	浮点数输出显示小数点	
uppercase	用大写字母输出十六进制数	
showpos	输出正数显示‘+’号	
scientific	浮点数输出采用科学表示法	floatfield
fixed	用定点数形式显示浮点数	floatfield
unitbuf	完成输出操作后立即刷新流	
boolalpha	以true和false值读取和写入到逻辑类型的对象	

标志都定义成一组符号常量，并由一个long型整数保存（有时说成int型，但在Visual C++中，int型和long型整数的数据长度是一样的都是4个字节），正如图7.9所示，每一个标志占用一个二进制位（bit），其值为1时表示该标志被设置，为0则表示没有设置。由于这些格式标志符号常量都是ios_base类的公有静态数据成员，因此，编程者使用时必须在标志前面加上前缀“ios_base::”。另外，系统把关系密切的、处于相邻位置的标志规定为一个字段（field），其中共有adjustfield（对齐方式字段）、basefield（进制方式字段）和floatfield（浮点方式字段）等3个字段，adjustfield字段包含dec、oct和hex等标志位，basefield字段包含left、right和internal等标志位，floatfield字段包含scientific和fixed等标志位。由于这3个字段也是ios_base类的公有静态数据成员，因此，编程者使用时必须在标志前面加上前缀“ios_base::”。

7.4.3 设置和清除格式控制标志的成员函数

1. 设置格式控制标志位的成员函数setf()

ios_base类模板中还提供了一系列成员函数，对流的输出/输入操作进行格式控制。

首先，若要设置一个格式控制标志，例如把scientific设置为1，即把实型数输出设置为科学表示法的指数显示，可使用setf()函数。ios_base根类提供如下两个重载

的成员函数设置格式控制标志：

```
fmtflags    setf( fmtflags fmtfl );  
fmtflags    setf( fmtflags fmtfl , fmtflags mask );
```

其中fmtflags类型是一个枚举类型，即：

```
enum    {  
    skipws = 0x0001,  
    left   = 0x0002,  
    right  = 0x0004,  
    ...  
};
```

形参fmtfl指定格式控制标志，对应的实参为格式控制标志常量，如ios_base::scientific，即将浮点数输出采用科学表示法。第1个成员函数只带有一个形参，它是将指定的标志位设置为1，其他标志位保持原来状态不变，函数返回到设置前的格式控制标志值。例如，若把输出到显示器的数值设置为输出正数时显示正号“+”，则调用的格式为：

```
cout.setf( ios_base::showpos );
```

它经常用于上述3个字段以外的格式控制标志设置。而第2个成员函数含有fmtfl和mask两个形参，mask用来指定字段，对应的实参为ios_base::adjustfield、ios_base::basefield和ios_base::floatfield，该函数用来设置某个字段中的标志，它先把整个字段的所有标志都清除掉，然后将指定标志位设置为1，函数仍然返回到设置前的该格式控制标志值。

由于每个标志位是long型整数中的一个二进制位（bit），因此setf（）中的参数可用位逻辑运算符进行组合，使得一次可设置多个状态位。

例7.5 setf（）函数的使用。

```
#include    < iostream >                // 使用C++新标准的流库  
using namespace std;                    // 将std标准名空间合并到当前名空间  
void main( void )  
{    cout.setf( ios_base::showpos | ios_base::scientific );  
    cout << 123 << "    " << 123.45 << "\n";  
}
```

设置showpos标志，使得每个正数前面都加上“+”号，同时设置scientific，使浮点数按科学表示法进行显示。上述程序的输出结果为：

```
+123      +1.2345e+02    (    为空格符)
```

2. 清除格式控制标志位的成员函数unsetf（）

`ios_base`根类中还提供了清除格式控制标志位的成员函数，它的原型是：

```
void unsetf( fmtflags mask );
```

其中，形参`mask`既可以指定一个字段，也可指定某个格式控制标志位。更具体地说，要清除某个字段的所有标志位或把某个标志位设置为0，例如要改变输出到显示器的浮点数显示形式，若把`floatfield` 字段的所有标志位或者`scientific`标志位设置为0，可通过调用成员函数`unsetf()`，即：

```
cout.unsetf( ios_base::floatfield );
```

或

```
cout.unsetf( ios_base::scientific );
```

它的使用方法与`setf()`函数类似。

7.4.4 操作符

如前所述，C++流库把标准运算符集合中的左移运算符“<<”和右移运算符“>>”分别重载成输入运算符和输出运算符，使得输入流/输出流的表达式更简洁，更能形象直观地体现面向对象风格。C++流库提供了许多输入流/输出流操作和格式控制的成员函数，虽然可以通过流对象使用访问成员运算符“.”和指向成员运算符“->”调用这些成员函数，但是这些调用表达式却不能与输入运算符“<<”和输出运算符“>>”直接配合使用。例如，`ios_base`类模板中为格式控制提供了如下两个设置输出数据域宽的成员函数，它的原型是：

```
streamsize width() const;
```

```
streamsize width( streamsize wide );
```

其中，第1个是无参数成员函数，它的返回值是已保存的域宽值，第2个成员函数由形参`wide`指定设置的域宽值，返回值是设置前的域宽值。调用它们时必须通过流对象，例如，若需设置输出显示的数据域宽，并输出一个数据值需编写如下3条语句：

```
cout << " value of object = ";
```

```
cout.width( 12 );
```

```
cout << object.value;
```

显然，这会给编程增添很多麻烦，特别在执行多个输入/输出流操作的场合，显得十分烦琐并容易出错。为此，C++流库又为编程者提供了一整套能与输入运算符“<<”和输出运算符“>>”直接配合使用的特殊操作函数，称为“操作符（manipulator）”，每个操作符都与一个具体的、能完成输入/输出流操作功能和格式控制的函数相联系，有时也称为操作符函数，并放在标准名空间`std`内，做成一个标准头文件`<iomanip>`，即：

```
namespace std { // MANIPULATORS
    T1 resetiosflag( ios_base::fmtflags mask );
```

```

/* 当对一个流对象str进行读取和写入操作时，该函数返回到str对象，它是通过调用
   str.setf( ios_base::fmtflag(), mask)完成的 */
T2  setiosflag( ios_base::fmtflags mask );
/* 当对一个流对象str进行读取和写入操作时，该函数返回到str对象，它是通过调用
   ios_base::str.setf( mask )完成的 */
T3  setbase( int base );
/* 当对一个流对象str进行读取和写入操作时，该函数返回到str对象，它是通过调用
   str.setf( mask, ios_base::basefield )完成的 */
template < class E >
T4  setfill( E c );
/* 当对一个流对象str进行读取和写入操作时，该模板操作符函数返回到str对象，它
   是通过调用str.fill( filch )完成的，E必须与str的类型相同 */
T5  setprecision( int n );
/* 当对一个流对象str进行读取和写入操作时，该函数返回到str对象，这是通过调用
   str.precision( prec )完成的 */
T6  setw( int n );
/* 当对一个流对象str进行读取和写入操作时，该函数返回到str对象，这是通过调用
   str.width( wide )完成的 */
};

```

由此可见，所有操作符分别返回到一个未指定的数据类型T1 ~ T6，这些类型都编写有重载输入运算符和输出运算符的成员函数“basic_istream::operator>>()”和“basic_ostream::operator<<()”，因此，流对象的各种输入/输出流操作可以直接与输入运算符“<<”和输出运算符“>>”配合使用。同时，每个操作符都对应着ios_base类中的一个成员函数与之相关联，当它们与输入运算符“<<”和输出运算符“>>”一起配合使用时，该操作符会自动地调用与之相关联的成员函数，完成相应的输入/输出流操作和指定的格式控制。例如，7.4.1节所述的两个成员函数setf()和unsetf()，前者对应的操作符为setiosflag(ios_base::fmtflags mask)，后者对应的是resetiosflag(ios_base::fmtflags mask)。实现与上述成员函数同样的操作功能，若使用操作符则非常简洁、明快和方便，可把多个操作用一条语句一气呵成。如例7.5可改写成：

```

#include    < iostream >          // 使用C++新标准的流库
#include    < iomanip >            // 它包含有操作符函数的原型声明
using namespace std;             // 将std标准名空间合并到当前名空间
void main( void )
{   cout << setiosflags( ios_base::showpos | ios_base::scientific )
    << 123 << "    " << 123.45 << "\n";
}

```

```
}
```

当然，在使用操作符的源程序开头必须用#include语句把<iomanip>头文件引入到当前源程序中。另外，包含在<iostream>头文件中的<ios>头文件（它很少直接使用#include语句包含到当前程序中）内又定义了一组操作符函数，把它们也放在std标准名空间中，即：

```
namespace std {
    typedef T1 streamoff;
    typedef T2 streamsize;
    class ios_base; // 最顶层根类的声明语句
    // TEMPLATE CLASSES, 几个类模板的声明
    template < class E, class T = char_traits<E> >
    class basic_ios;
    typedef basic_ios< char, char_traits<char> > ios;
    // 用typedef语句给实例化模板类basic_ios<char>启用一个新类型名ios
    typedef basic_ios< wchar_t, char_traits<wchar_t> > wios;
    // 用typedef语句给实例化模板类basic_ios<wchar>启用一个新类型名wios
    template < class St >
    class fpos;
    typedef fpos< mbstate_t > streampos;
    typedef fpos< mbstate_t > wstreampos;
    // MANIPULATORS, 一组操作符函数的声明
    ios_base & boolalpha( ios_base & str );
    /* 它调用ios_base类的成员函数str.setf( ios_base::boolalpha ), 用来改变
作
    为参数的ios_base类str对象的值，返回到对象str */
    ios_base & noboolalpha( ios_base & str );
    /* 它调用ios_base类的成员函数str.unsetf( ios_base::boolalpha ), 用来改
变
    作为参数的ios_base类str对象的值，返回到对象str */
    ios_base & showbase( ios_base & str );
    /* 它调用ios_base类的成员函数str.setf( ios_base::showbase ), 用来改变
为
    参数的ios_base类str对象的值，返回到对象str */
    ios_base & noshowbase( ios_base & str );
    /* 它调用ios_base类的成员函数str.unsetf( ios_base::showbase ), 用来改变
    作为参数的ios_base类str对象的值，返回到对象str */
    ios_base & showpoint( ios_base & str );
```

```

/* 它调用ios_base类的成员函数str.setf( ios_base:: showpoint ), 用来改变
   作为参数的ios_base类str对象的值, 返回到对象str */
ios_base & noshowpoint( ios_base & str );
/* 它调用ios_base类的成员函数str.unsetf( ios_base:: showpoint ), 用来改
   变作为参数的ios_base类str对象的值, 返回到对象str */
ios_base & showpos( ios_base & str );
/* 它调用ios_base类的成员函数str.setf( ios_base::showpos ), 用来改变作为
   参数的ios_base类str对象的值, 返回到对象str */
ios_base & noshowpos( ios_base & str );
/* 它调用ios_base类的成员函数str.unsetf( ios_base::showpos ), 用来改变
   为参数的ios_base类str对象的值, 返回到对象str */
ios_base & skipws( ios_base & str );
/* 它调用ios_base类的成员函数str.setf( ios_base::skipws ), 用来改变作为
   参数的ios_base类str对象的值, 返回到对象str */
ios_base & noskipws( ios_base & str );
/* 它调用ios_base类的成员函数str.unsetf( ios_base::skipws ), 用来改变作
   为参数的ios_base类str对象的值, 返回到对象str */
ios_base & unitbuf( ios_base & str );
/* 它调用ios_base类的成员函数str.setf( ios_base::unitbuf ), 用来改变作
   为参数的ios_base类str对象的值, 返回到对象str */
ios_base & nounitbuf( ios_base & str );
/* 它调用ios_base类的成员函数str.unsetf( ios_base::unitbuf ), 用来改变
   作参数的ios_base类str对象的值, 返回到对象str */
ios_base & uppercase( ios_base & str );
/* 它调用ios_base类的成员函数str.setf( ios_base::uppercase ), 用来改变
   作参数的ios_base类str对象的值, 返回到对象str */
ios_base & nouppercase( ios_base & str );
/* 它调用ios_base类的成员函数str.unsetf( ios_base::uppercase ), 用来改
   变参数的ios_base类str对象的值, 返回到对象str */
ios_base & internal( ios_base & str );
/* 它调用ios_base类的成员函数str.setf( ios_base::internal,
   ios_base::adjustfield ), 用来改变作为参数的ios_base类str

```

```

        对象的值，返回到对象str */
ios_base & left( ios_base & str );
/* 它调用ios_base类的成员函数str.setf( ios_base::left,
    ios_base::adjustfield ), 用来改变作为参数的ios_base类
    str对象的值，返回到对象str */
ios_base & right( ios_base & str );
/* 它调用ios_base类的成员函数str.setf( ios_base::right,
    ios_base::adjustfield ), 用来改变作为参数的ios_base类
    str对象的值，返回到对象str */
ios_base & dec( ios_base & str );
/* 它调用ios_base类的成员函数str.setf( ios_base::dec,
    ios_base::basefield ), 用来改变作为参数的ios_base类
    str对象的值，返回到对象str */
ios_base & hex( ios_base & str );
/* 它调用ios_base类的成员函数str.setf( ios_base::hex,
    ios_base::basefield ), 用来改变作为参数的ios_base类
    str对象的值，返回到对象str */
ios_base & oct( ios_base & str );
/* 它调用ios_base类的成员函数str.setf( ios_base::oct,
    ios_base::basefield ), 用来改变作为参数的ios_base类
    str对象的值，返回到对象str */
ios_base & fixed( ios_base & str );
/* 它调用ios_base类的成员函数str.setf( ios_base::fixed,
    ios_base::floatfield ), 用来改变作为参数的ios_base类
    str对象的值，返回到对象str */
ios_base & scientific( ios_base & str );
/* 它调用ios_base类的成员函数str.setf( ios_base::scientific,
    ios_base::floatfield ), 用来改变作为参数的ios_base类str
    对象的值，返回到对象str */
};

```

这样一来，使用这些操作符函数就可以方便地与输入运算符">>"和输出运算符"<<"配合直接设置各种格式控制标志。例如，若把输出到显示器的实型数设置为显示精度为小数点后面3位的浮点数形式则可写成：

```
cout << fixed << setprecision(3);
```

如果是输出到一个流对象，即basic_ostream类模板实例化的一个模板类对象ostr，则有：

```
ostr << fixed << setprecision(3);
```

7.4.5 读取格式控制标志的成员函数

编程时，常常需要将格式控制标志读出以便观察，可采用在ios_base类中定义的flags()成员函数，它有两种形式：

```
fmtflags  flag( ) const;           // 读取所保存的格式控制标志

fmtflags  flags( fmtflags  fmtfl );
// 读取形参mask指定的格式控制标志fmtfl，返回值为以前保存的标志值
```

第1种形式的返回值为与流相关的格式控制标志的当前值；第2种形式将流的格式控制标志值设置为参数fmtfl的值，返回值为以前保存的标志值。

例7.6 成员函数flags()的使用。

```
#include <iostream>           // 使用C++新标准的流库
using namespace std;          // 将std标准名空间合并到当前名空间

void  showflags( long  f );

void  main( void )
{  unsigned f;
   f = cout.flags( );          // 读入状态标志位到变量f
   showflags(f);               // 以二进制形式输出显示状态标志位

   cout.setf( ios_base::hex | ios_base::right );
   // 设置输出数据为十六进制和右对齐

   f = cout.flags( );          // 再把状态标志位读入到变量f
   showflags(f);               // 以二进制形式输出显示状态标志位

   cout.unsetf( ios_base::hex | ios_base::right );
   // 将hex位和right位复位

   f = cout.flags( );          // 再把状态标志位读入到变量f
   showflags(f);               // 以二进制形式输出显示状态标志位

   cout << "\n";
}

void  showflags( long  f )
{  unsigned i;                 // 定义一个无符号整型变量i作为移位屏蔽码
   int  j;                     // 定义一个int型变量j记录二进制位从左至右的位序号
   for(i = 0x8000, j = 1; i; i >>= 1, j++) {
       if( i & f ) cout << "1";
       // 也可写成"if( (i & f) != 0 )"，若某个二进制位为1输出数字字符串"1"
       else      cout << "0";
       // 若某个二进制位为0输出数字字符串"0"

       if( j % 4 == 0 && j < 15 ) cout << ",";
       // 每隔4位二进制位输出一个",",直到第14位
```



```

    }
    // 把形参变量f以二进制形式读出，每4位（bit）为一组，组间用逗号隔开
    cout << "\n";
}

```

如果直接输出存放状态标志值的变量f（可写成cout << f;），则输出结果将是一个十进制数值。编写一个showfrags（）函数，将其按二进制位展开，便于观测，例如该程序的输出结果为：

```

0010,0000,0000,0001    ( f = 2001 )
0010,0000,0100,0101    ( f = 2045 )
0010,0000,0000,0001    ( f = 2001 )

```

7.4.6 设置域宽、填充字符和浮点精度

1. 设置域宽、填充字符和浮点精度的成员函数

除了格式标志外，ios_base类还提供了设置域宽、填充字符、设置浮点新精度的成员函数：

```

streamsize  ios_base::width( streamsize  wide );
// 设置域宽为以前保存的域宽值，返回该域宽值

streamsize  ios_base::fill( streamsize  ch );    // 填充字符，缺省值为空格

streamsize  ios_base::precision( streamsize  prec );
// 设置显示精度为prec，缺省值为小数点后6位

```

首先看一个例程，再说明它们的用法。

例7.7 设置域宽、填充字符和浮点数精度。

```

#include    < iostream >                // 使用C++新标准的流库
using namespace std;                    // 将std标准名空间合并到当前名空间

void main( void )
{
    int i;                               // 定义一个自动型变量i作为下标变量和循环变量

    double v[3] = {1200.235, 320.923, 54430.00};
    // 定义一个自动型一维数组v[ ]并初始化3个元素

    cout.precision(2);                   // 设置显示精度为小数点后2位

    cout.setf( ios_base::right | ios_base::showpoint | ios_base::fixed );
    // 设置右对齐，对浮点数要显示小数点，按固定格式显示浮点数

    cout.fill( '*' );                    // 当数据宽度小于设置宽度时，空闲位置上填充字符
    '*'

    for( i = 0; i < 3; i++ ) {
        cout << "Check Value : $";
    }
}

```

```

        cout.width(10);           // 设置10个字符的宽度
        cout << v[i] << endl;    // 输出显示数组v[ ]的所有元素
    }
}

```

该程序的输出结果为：

```

Check Value : $***1200.23
Check Value : $****320.92
Check Value : $**54430.00

```

`width()`、`fill()`和`precision()`均为`ios_base`类的成员函数，调用格式一般为：

对象名.成员函数名(实参表);

例如：`cout.width(10);`

`fill()`为输出域左右对齐填充所需的字符，其参数缺省时填充的字符为空格。C语言中的`printf()`仅能填充空格，而`fill()`可填充任意字符。

`precision()`设置显示浮点数精度为小数点后面的位数或者有效位数，缺省值为小数点后面6位。

`width()`设置域宽，只有当设置的域宽大于输出的字符串长度（字符串包含的字符个数）时，才需要填充字符。当输出字符串的个数大于域宽时，则突破域宽，输出全部字符，`width()`的缺省值由系统设置为0，此时输出不需要填充字符，即数据将按实际的宽度输出全部字符。`width()`所设置的宽度仅对下一个流的输出操作有效，在一次输出操作完成后，宽度又回到了0，如例7.7中，把`width(10)`放在for语句的循环体内，就是为了在每次输出操作前设置宽度为10，以确保每个`v[i]`的显示域宽都是10。该例程为账本上的金额数据，需排列整齐，如果不设定显示精度（即去掉“`cout.precision(2);`”语句），则输出小数点后6位，输出字符数大于设置的域宽10，不需要填充字符，也不可能右对齐，输出结果为：

```

Check Value : $1200.235000
Check Value : $320.923000
Check Value : $54430.000000

```

2. 操作符函数的使用

如前所述，若使用操作符函数（Manipulator）可以直接与输入运算符“<<”和输出运算符“>>”配合使用，可把多个操作作用一条语句一气呵成。如例7.7可改写成：

```

#include < iostream >           // 使用C++新标准的流库
#include < iomanip >             // 它包含有操作符函数的原型声明

```

```

using namespace std;           // 将std标准名空间合并到当前名空间
void main( void )
{   int i;
    double v[3] = { 1200.235, 320.923, 54430.00 };
    cout << setprecision(2)
           << setiosflags( ios_base::right |
                           ios_base::showpoint |
                           ios_base::fixed )
           << setfill( '*' );
    for( i = 0; i < 3; i++ )
        cout << "Check Value : $" << setw(10) << v[i] << endl;
}

```

由此可知，操作符函数`setw()`对应`ios_base`类的成员函数`width()`，而操作符`setprecision()`对应`ios_base`类的成员函数`precision()`，操作符`setfill()`对应`basic_ios`类模板的成员函数`fill()`等。

3. 其他操作符

插入换行符`endl`：它定义在`<ostream>`新标准头文件（包含在`<iostream>`头文件中，很少直接使用`#include`语句包含到当前程序中）中，它是一个操作符模板函数，实现与格式控制无关的回车换行操作，与`'\n'`的功能相同。它在`<ostream>`中定义为：

```

namespace std
{
    ...
    template<class E, class T>
        basic_ostream < E, T > & operator << ( basic_ostream < E, T > os,
                                                const E * s );

    template class < E, T >
        basic_ostream < E, T > & endl( basic_ostream < E, T > os );
    template class<E, T>
        basic_ostream < E, T > & ends( basic_ostream < E, T > os );
    template class<E, T>
        basic_ostream < E, T > & flush( basic_ostream < E, T > os );
};

```

该操作符通过一个输出流对象`os`调用`basic_ostream <char, char_traits<char> >`（模板形参`E`实例化为`char`型的模板类即为`ostream`模板类，以下简称`ostream`类）模板类的成员函数，即`"os.put(os.widen('\n'))"`，向输出流对象`os`写入一个换行符`'\n'`，然后再调用该模板类的成员函数`os.flush()`完

成流缓冲器的清理工作以确保读写的同步。

插入字符串结尾符ends :它也是在<ostream>新标准头文件中的一个操作符模板函数，该操作符通过一个输出流对象os调用ostream类的成员函数，即“os.put(os.widen ('\\0'))”，向输出流对象os写入一个字符串结尾符'\\0'。

设置整数进制的操作符(函数):使用操作符可以设置10进制、8进制和16进制。系统的初始状态设置为10进制，设置的进制状态一直有效，直到下一次设置完成为止。相应的操作符有：

- dec : 将整数的输入/输出设置为10进制，它是默认设置，
- oct : 将整数的输入/输出设置为8进制，
- hex : 将整数的输入/输出设置为16进制，
- setbase : 设置整数进制为8进制、10进制和16进制，
- showbase : 给输出整数添加一个表示进制的前缀，8进制添加一个数字0，16进制添加一个0x或0x，
- noshowbase : 取消showbase操作符的设置，它是默认设置，即在系统的初始状态下添加的进制前缀中的英文字母为小写，可通过如下操作符把设置改为大写字母，
- uppercase : 将添加的进制前缀中的英文字母改为大写字母，
- nouppercase : 取消uppercase操作符的设置，把进制前缀中的英文字母恢复成小写字母。

这些操作符所调用的成员函数请参阅7.4.2。

设置逻辑常量输出方式 :在系统的初始状态下，逻辑常量输出1(逻辑真)和0(逻辑假)的整数，通过使用如下操作符设置，可以使之输出true(逻辑真)和false(逻辑假)，设置一直保持到下一次设置完成为止。

- boolalpha : 使逻辑常量输出true(逻辑真)和false(逻辑假)，
- noboolalpha : 取消boolalpha操作符的设置，使逻辑常量恢复输出1(逻辑真)和0(逻辑假)，它是默认设置。

设置前导空白字符处理方式 :在系统的初始状态下，输入某个数据(例如用键盘敲入一个数据)时，跳过前面的空白符才读取第1个非空白字符。可通过使用如下操作符设置，使得输入时把空白符作为数据的一部分接收，设置一直保持到下一次设置完成为止。

- skipws : 输入时跳过前面的空白符，它是默认设置，
- noskipws : 取消skipws操作符的设置，

- `ws` :在用`noskipws`操作符设置的状态下输入时,用它使得输入时跳过前面的空白符,使输入流指针`gptr`指向第1个非空白符。

设置流缓冲器工作方式 :如前所述,通常当流缓冲器的数据写满时才发送给外部设备。可通过使用如下操作符设置改变这种工作状态,使得每次写入到缓冲器的数据立即发送给外部设备,设置一直保持到下一次设置完成为止。

- `unitbuf` :每次写入到缓冲器的数据立即发送给外部设备,
- `nounitbuf` :取消`unitbuf`操作符的设置,它是默认设置,
- `flush` :强制性地完成流缓冲器的清理工作以确保操作的同步。

设置正数符号的表示方式 :在系统的初始状态下,输出的正数前不显示`+`号,但可以通过使用如下操作符设置,使得在正数前显示`+`号,设置一直保持到下一次设置完成为止。

- `showpos` :正数前显示`+`号,
- `noshowpos` :取消`showpos`操作符的设置,它是默认设置。

7.5 文件I/O流

利用流操作同样可以处理文件,不管是文本方式还是二进制方式的文件。在C++中,要对文件进行读写操作,即进行文件I/O操作,首先必须创建一个流,然后将这个流与文件相关联(称为打开文件),这时才能进行读取和写入操作。文件使用完后,需关闭文件。这与C语言中的3个过程是一样的,即:打开文件——>文件的读/写——>关闭文件。

7.5.1 文件的打开和关闭

正如图7.8所示,C++新标准流库的类层次结构中,分别定义了`basic_ifstream`和`basic_ofstream`两个类模板,前者从`basic_istream`类模板继承而来,用于处理输入文件流,后者从`basic_ostream`类模板继承而来,用于处理输出文件流,即:

```
template < class E, class T = char_traits<E> >
class basic_ifstream : public basic_istream < E, T > {
public:
    explicit basic_ifstream( );           // 无参数构造函数

    explicit basic_ifstream ( const char * s,
                               ios_base::openmode mode = ios_base::in );
    // 一般构造函数,创建输入文件流对象并打开相关联的文件,默认打开方式为只读

    basic_filebuf < E, T > * rdbuf() const;
```

```

        // 返回到保存在流缓冲器内的指针，该指针指向basic_filebuf类模板实例化的对象
bool    is_open( )    const;
void    open( const    char    *    s,
                ios_base::openmode    mode = ios_base::in );
/* 打开与文件流对象相关联的文件进行读取操作，形参字符串指针s指向对应的实参文件
名
        字符串，形参指定文件打开方式，默认值为只读 */
void    close( );          // 关闭输入文件流
};
和
template    < class E, class T = char_traits<E> >
class    basic_ofstream : public    basic_ostream < E, T >    {
public:
    explicit    basic_ofstream( );
    explicit    basic_ofstream( const    char    *    s,
                ios_base::openmode    mode = ios_base::out |
ios_base::trunc );
    /* 一般构造函数，创建输入文件流对象并打开相关联的文件，默认打开方式为清除原来文
件
        内容变成空文件只写 */
    basic_filebuf < E, T >    *    rdbuf( )    const;
    // 返回到保存在流缓冲器内的指针，该指针指向basic_filebuf类模板实例化的对象
    bool    is_open( )    const;
    void    open( const char    *    s,
                ios_base::openmode    mode = ios_base::out |
ios_base::trunc );
    /* 打开与文件流对象相关联的文件进行只写操作，形参字符串指针s指向对应的实参文件
名
        字符串，形参指定文件打开方式，默认值为只写 */
    void    close( );          // 关闭输出文件流
};

```

当模板形参E确定为char型，模板形参T采用默认值char_traits<char>时，则实例化为如下两个模板类：

```

template    < char, char_traits<char> >
class    basic_ifstream;
和
template    < char, char_traits<char> >

```

```
class basic_istream;
```

为了书写方便再用两个typedef语句分别给它们启用新的类型名来替代这两个很长的模板类名，即：

```
typedef basic_ifstream<char> ifstream;
typedef basic_ofstream<char> ofstream;
```

因此，今后就把basic_ifstream <char>和basic_ofstream <char>这两个模板类名分别简称为ifstream类和ofstream类，并将它们封装在标准名空间std内，然后放在<fstream>新标准头文件中。因此，在处理文件的源程序的开头处，必须包含如下两条语句：

```
#include < fstream >
// 使用C++新标准的文件流
using namespace std;
```

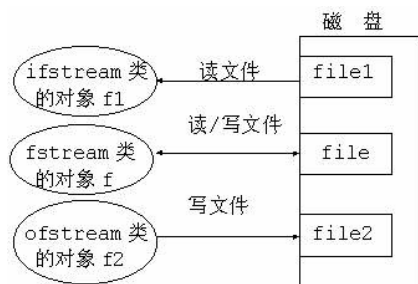


图7.10 各种文件打开方式
所对应的流类对象

```
// 将std标准名空间合并到当前名空间
```

ifstream类支持为读取操作打开的文件，而ofstream类支持为写入操作打开的文件。如图7.10所示，要打开一个文件进行读操作，必须创建其类型为ifstream类的对象；要打开一个文件进行写操作，必须创建一个类型为ofstream类的对象；若要打开一个既要读又要写的文件，则必须创建其类型为fstream类的对象，即：

```
ifstream in; // object of input stream
ofstream out; // object of output stream
fstream both; // object of input and output stream
```

因此，要打开一个文件，首先必须建立一个相应的流，然后将它与要打开的文件相关联，通常使用ifstream、ofstream和fstream类的成员函数open()，其实例化后的原型声明为：

```
void open( const char * s,
           ios_base::openmode mode = ios_base::out |
           ios_base::trunc );
```

第1个参数指向相关联文件名的字符串指针s，文件名中可以包含驱动器号及多层子目录的完全路径名，在路径名中要用双反斜杠(\\)将目录隔开，通常可以用一对双引号包括的字符串常量作为实参，也可以用主函数main()的参数argv[]传递命令行(详见文献[25])。

打开文件有如下两种编写方式：

- 首先创建一个文件流对象f1，然后在需要的时候用成员函数open()打开文件，即：

```
ifstream    f1;
f1.open( "e:\\user\\filename.cpp" );
// 输入文件流f1与文件filename.cpp相关联，即打开一个文本文件进行读操作
```

第 条语句自动调用了ifstream类（用于处理输入文件流的类）的构造函数，以创建并初始化一个称之为f1的流对象。

第 条语句是在使用f1之前，必须先建立一个文件缓冲器，并将这个流对象和文件缓冲器以及一个实际的物理文件（"e: user\\filename.cpp"）连接起来。这两项任务由ifsteram中的成员函数open()来完成。

- 打开文件的另一种编写方法是在创建文件流对象f1的同时打开文件，即：

```
ifstream    f1( "e:\\user\\filename.cpp" );
```

这时成员函数名open可以省略不写，在创建文件流对象f1时由其构造函数自动调用成员函数open()打开文件。

如果打开一个文件，一切正常，则为该磁盘文件建立了一个输入缓冲器，且get指针指向缓冲器的起始位置。第2个参数mode表示文件打开的模式，其类型是在ios_base类中用typedef语句为模板形参T3启用的新类型名openmode。表7.3给出了ios_base类中所提供的文件打开模式标志"ios_base::openmode"，若不写第2个实参则默认打开模式为"out"即以只写模式打开文件。

输入文件流ifstream类、输出文件流ofsteram类和输入/输出文件流fstream类都定义了各自的成员函数open()，其原型基本相同，仅第2个参数mode的设置不同。对于ifstream类的对象， mode默认设置打开模式为ios_base::in，通常使用open()函数时仅写第1个参数即可，即打开一个文本文件只进行读取操作，简称为“只读”，如：

表7.3 iso_base类中提供的文件打开模式标志位

方式名	用 途
app	以输出添加数据方式打开文件

ate	文件打开后定位在文件尾
binary	以二进制方式打开文件，缺省时以文本方式
in	以只读方式打开文件
out	以只写方式打开文件
trunc	若打开已存在的文件进行写操作，则将清除文件原有内容变成空文件；若文件不存在则创建新文件。

```
ifstream f1;
f1.open( "a:\\user\\file1.cpp" );
```

对于ofstream类的对象，mode默认设置打开模式为ios_base::out，打开一个文本文件只进行写入操作，简称为“只写”，即：

```
ofstream f2;           可缺省
f1.open( "file2.cpp", ios_base::out );
```

若打开一个用于读和写的文本文件，应首先创建一个输入/输出文件流fstream类的对象，将读和写的操作通过“位或”操作组合起来。

```
fstream f;
f.open( "file.cpp", ios_base::in | ios_base::out );
```

文本方式和二进制方式：因文件分为文本文件和二进制文件，详见文献[25]。若要打开一个二进制文件，除了需要向打开文件的成员函数open()传递ios_base::binary标志外，打开和关闭文件的方法与文本方式相同，例如：

```
ifstream f2;
f2.open( "a:pnt.doc", ios_base::binary );
```

当文件不指明打开方式，即缺省时，则以文本方式打开。在这种情况下，输入时回车/换行符要转换为字符‘\n’，在输出时，字符‘\n’转换为回车/换行符，这些转换在二进制方式下是没有的。这是文本方式和二进制方式的主要区别。

7.5.2 ifstream、ofstream和fstream类的构造函数和析构函数

由于C++中的类所带的构造函数和析构函数具有创建该类对象和初始化对象的机制，且ifstream、ofstream和fstream类的构造函数的参数和缺省值与其成员函数open()完全相同，即

```
ifstream f1;
f1.open( "a:file1.cpp" );
```

可以写成：

ifstream f1.open("file1.cpp"); 或 ifstream f1("file1.cpp");

如果要以二进制方式打开文件，并在文件尾追加写入二进制数据，也可写成：

```
ofstream f2( "E:\\user\\file2.obj" , ios_base::binary |
ios_base::app );
```

流操作过程可能会出错，例如打开一个不存在的文件进行读操作，或读一个不允许读（或写）的文件，或用不正确格式写入数据等都将产生错误，必须进行检测并加以处理，检测流错误的办法很多，通常采用如下两种办法：

采用条件语句判断文件流的状态标志位failbit是否为true，即：

```
if( ! f1 ) { <异常处理路径> }
if( f1 ) { <正常处理路径> }
```

在调用成员函数open()打开文件时，若它得到一个空指针则会自动调用成员函数“basic_ios::setstate(failbit)”，将文件流的状态标志设置为true，则说明流操作出错，那么，表达式“! f1”则为非零值（也可写成“f1 == NULL”关系表达式的形式，它表示文件流f1获得了空指针），说明文件流操作失败。也可以同时测试多个流错误的办法是采用while循环，在任何一个流出错时就自动退出循环。例如：

```
while( f1 && cout ) {
    <copy file1.cpp to cout> // 正常处理路径，将file1.cpp文件输出到显示器
}
```

程序也可以通过调用成员函数basic_ios::fail()检测文件打开是否失败，该函数会自动调用basic_ios::rdstate()读出状态标志位的值，若文件打开时出错，即“rdstate() & failbit”结果值为true，把它作为fail()函数的返回值。在异常处理路径内判断是哪一类错误。如前所述，错误标志位在ios_base类中定义成枚举类型，即：

```
enum T2 {
    goodbit = 0x00, // 未出错
    eofbit = 0x01, // 到达文件结尾
    failbit = 0x02, // 操作失败
    badbit = 0x04 // 非法操作
};
```

一旦检测出错误位后，对该流的读出和写入操作都将停止，basic_ios(模板)类的成员函数basic_ios::clear(int)可以清除流的错误标志位。

C++的一个优点是流对象在退出作用域时，会调用析构函数自动关闭所有文件，编程者不必编写关闭文件的语句。如果程序在中途要关闭某文件，则必须使用输入文件流(basic_ifstream)或输出文件流(basic_ofstream)或输入/输出文件流等(模板)类(basic_fstream)的成员函数close()。

7.5.3 文件的读/写

一旦文件打开后，接着需要编写操作语句来读写文件内容，二进制文件和文本文件所采用的读写操作是不同的。

1. 文本文件的读/写

文本文件的读操作通常采用 `istream` 类中的成员函数 `basic_istream::get()` (前者是后者的实例化类)，写成：

```
ifstream    f1( "file1.cpp" );

    ...

while( f1 ) {
    char    c;           // 定义一个字符型自动变量c，保存读取的单个字符
    f1.get(c);           // 调用成员函数get( )从f1中读取单个字符保存在字符变量c中
    cout << c;           // 把字符变量c输出到显示器
}
```

由于 `ifstream(模板)` 类由 `istream(模板)` 类派生而来，根据面向对象继承的规则，它也会自动地调用成员函数 `get()`，即：

```
istream &    get( char & c );
```

每次读取 `f1` 中的单个字符传给 `c`，然后传送给显示器，文件在 `get` 指针的控制下按顺序读取，直到 `EOF`，编程者可不必关心具体实现细节。如果不是写到输出标准流对象 `cout`，而是写到另一个文件中，则在进行写操作前，必须打开另一个输出流文件，写成：

```
ofstream    f2( "file2.cpp" );

if( ! f2 ) {
    cerr << "\a Can't open file2.cpp.\n";
    exit(1);
}

/* 异常终止路径，也可写成 "if( f2 == NULL )"，若与f2相关联的文本文件file2.cpp不能打开，调用标准函数exit( )立即终止执行程序，退回到操作系统 */

char    c;           // 定义一个字符型自动变量c，保存读取的单个字符
while( f2 && f1.get( c ) )
    f2.put( c );

/* 每循环一次调用成员函数get( )从f1中读取一个字符，再调用put( )写到f2中，直到与f2相关联的文本文件的末尾或者读写过程中出错为止 */
```

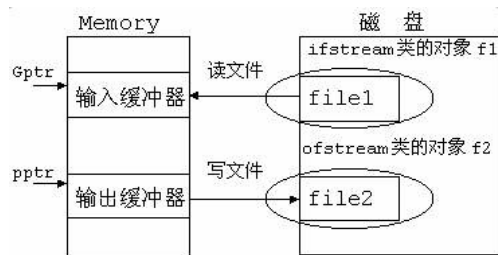


图7.11 文本文件的读/写过程图

如图7.11所示，在while循环语句中，程序每次从f1中读取一个字符，然后由basic_ostream::put()写到f2中，实际上是从与f1相关联的输入缓冲器中读取一个字符，写到与f2相关联的输出缓冲器中，由于get指针和put指针能自动增1，故确保了字符的读写顺序。当get指针指到输入缓冲器的终止位置时，说明整个文件已读完，此时输出缓冲器已复制了整个文件的内容，f1.get(c)读到文件尾将测试到该表达式的结果值为假(false)，则程序跳出循环，执行后继语句。顺便指出，与文本文件相关联的缓冲器以NULL结尾。如前所述，对于文本文件经常采用读取一行字符串的重要成员函数getline()，写成：

```
ifstream    f1( "file1.cpp" );
if( ! f1 )  {
    cerr << "\a Can't open file1.cpp.\n";
    exit(1);
}
/* 异常终止路径，也可写成"if( f1 == NULL )"，若与f1相关联的文本文件file1.cpp不能打开，调用标准函数立即终止执行程序，退回到操作系统 */
while( f1 )  {
    char    buf[100];    // 定义一个字符型数组buf[100]，保存读取的一行字符串
    f1.getline( buf, sizeof(buf) );
    // 调用成员函数getline()，从f1中读取100个字符存放到字符串数组buf[100]中
    cout << buf;        // 字符型数组buf[100]中的字符输出到显示器
}
```

2. 二进制文件的读/写

二进制文件的读写就是无格式的输入/输出流操作。二进制文件不含空白字符（空格字符"space"、换行符"cr"和水平制表符"tab"），数据没有通过换行符组织成一行行字符串的形式，任何8位(bit)值都可能出现。因此，将二进制文件调到屏幕上显示出的信息很难看懂。通常是利用ostream类的成员函数write()将数据写入二进制文件。写二进制数据有两种方法：一是一次写入一个字符，使用成员函数put(char c)；二是一次写一组字符（但不是以终止符结束），则使用如下成员函数：

```
ostream & write( const char* pch , int nCount );
...
```

为了便于调试，一般采用按字符形式格式化的方法，通过强制类型转换，将程序中要写入文件的对象或对象数组转换成字符串形式，写入二进制文件，其调用语句的一般格式为：

```
输出文件流对象名.write( (char *) & 对象名或&对象数组名[下标] ,
sizeof(<对象名或对象所属类名>) );
```

同样读二进制文件的方法通常是将文件按字符形式格式化的方法读入到一个与写入时同类的对象或对象数组中，其调用语句的一般格式为：

```
输入文件流对象名.read( (char *) & 对象名或&对象数组名[下标] ,
sizeof(<对象名或对象所属类名>) );
```

这样一来，可对二进制文件读写任何类型的数据。

例7.8 二进制文件的读/写。

```
#include < iostream > // 使用C++新标准的流库
#include < fstream > // 使用C++新标准的文件流
#include < cstdlib > // 标准函数exit( )的原型在其中
using namespace std; // 将std标准名空间合并到当前名空间

struct Person {
    char name[20]; // 私有数据成员字符串数组name[20]保存“姓名”
    double height; // 私有数据成员double型变量height保存身高，单位为cm
    unsigned short age; // 私有数据成员无符号整数age保存年龄
};

struct Person people1[4];
struct Person people2[4] = { "Wang", 1.65, 25, "Zhang", 1.72, 24,
                             "Li", 1.89, 21, "Hung", 1.70, 22 };

void main( void )
{
    ofstream f2("d:\\test\\Personal.dat", ios::binary);
    // 打开二进制流文件只写，输出文件流ofstream类的对象为f2
    if( ! f2 ) { // 也可以写成"if( f2 == NULL )"
        cout << "\a Can't open Personal.dat.";
        exit(1);
    }
    /* 当f2所关联的文件没有成功地打开时，输出出错信息并调用标准函数exit( )，立即终止执行程序，退回到操作系统 */
```

```

short    cnt = 0;
while( f2 && cnt < 4 )    {
    f2.write( (char *) &people2[cnt], sizeof(Person) );
    cnt++;
}
// 取出结构数组people2的数据，写到输出文件流ofstream类的对象f2，直到全部写完
f2.close( );           // 关闭文件流f2
ifstream f1("d:\\test\\Personal.dat", ios::binary);
// 打开二进制流文件只读，输入文件流ifstream类的对象为f1
if( ! f1 )    {
    cout << "\a Can't open Personal.dat.";
    exit(1);
}
/* 当f1所关联的文件没有成功地打开时，输出出错信息并调用标准函数exit( )，立即终止执行程序，退回到操作系统 */

cout << "NAME\t" << "HEIGHT\t" << "AGE\n";
for(int i = 0; i < 4; i++) {
    f1.read((char * ) &people1[i], sizeof(Person));
    /* 从输入文件流ifstream类的对象f1 中读取内容，并把它们存放在结构数组
       people1中 */
    cout << people1[i].name << "\t" << people1[i].height
         << "\t" << people1[i].age << endl;
    // 输出显示结构数组people1各元素的内容
}
f1.close( );           // 关闭文件流f1
}

```

该程序的输出结果为：

NAME	HEIGHT	AGE
Wang	1.65	25
Zhang	1.72	24
Li	1.69	21
Huang	1.7	22

Person类的people1[4]、people2[4]是两个结构数组，people2[4]是用初始化列表给它的每个元素数据成员赋初值。

在以二进制方式打开的文件中，采用basic_istream::read()和basic_ostream::write()成员函数可以读写任何类型的对象。例中是读写结构体

point的结构数组p[]。只需向打开文件的成员函数传送ios_base::binary 标志即可。

如前所述basic_istream::read()与basic_ostream::write()的原型为：

```
istream & read( char * s , int n );
...
ostream & write( const char * s , int n );
...
```

read()成员函数从输入流中读取n个字节，并把它们存放在形参字符串指针s所指的缓冲器中。write()成员函数是将n个字节从形参字符串指针s所指的缓冲器取出，写到输出文件流。

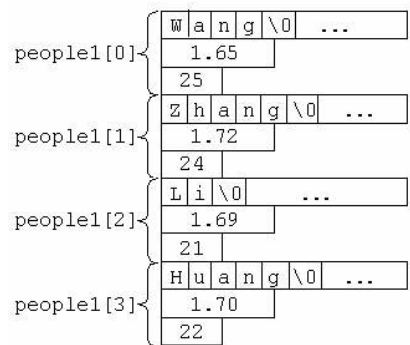


图7.12 结构数组people1[]的

数据结构图

本程序中write()是对结构数组people2[]进行操作，由于它不是字符型数组，故必须进行强制类型转换。结构数组people1[]的数据结构图如图7.12所示，结构数组 people2[] 的数据结构与 people1[] 一样，因此写入到 d:\\test\\Personal.dat文件中共有26个字节的二进制数据。

此例中写数据是否结束由自动变量cnt控制，不能只依靠f1表达式，它只能检测写操作是否出错而不能判终，若取消cnt < 2，表达式将变成无穷循环。如前所述，以二进制方式打开的文件，其关联的缓冲器不是以NULL结尾。

3. ios_base类的输入/输出流状态标志 iostate

由7.1.2节可知，在ios_base类中用typedef语句给T1 ~ T5类型分别定义了新的类型名，其中T2定义为：

```
typedef T2 iostate;
static const iostate badbit, eofbit, failbit, goodbit;
```

它是用来记录输入流状态和输出流状态的标志，实际上是一个枚举类型，记录如下4种

流状态：

- badbit ：流缓冲器完整性被破坏的状态标志
- eofbit ：读取到文件结尾的状态标志
- failbit ：读取流中无效字段的失败状态标志
- goodbit ：没有设置任何状态标志

参看7.1.2节可知，由ios_base类继承而来的basic_ios <E, T>类模板内又定义了如下几个成员函数对这些状态标志进行控制：

```
template < class E, class T = char_traits<E> >
class basic_ios : public ios_base {
public :
    ...
    void clear( iostate state = goodbit );
    // 按形参state清除流状态标志，若流操作失败自动抛出一个failure类的对象
    void setstate( iostate state );
    // 按形参state设置流状态标志
    bool good( ) const;
    // 检测goodbit位为1即所进行的流操作成功时返回true，否则返回false
    bool eof( ) const;
    // 检测eofbit位为1即流操作进行到文件结尾时返回true，否则返回false
    bool fail( ) const;
    // 检测failbit位为1即流操作失败时返回true，否则返回false
    bool bad( ) const;
    // 检测badbit位为1即流缓冲器完整性被破坏时返回true，否则返回false
    ...
protected :
    basic_ios( ); // 保护无参数构造函数
    void init( basic_streambuf< E, T > * sb );
};
```

4. 文件流的定位

C++流的定位操作主要用于文件流，所以我们以文件流为例介绍流的定位。它使得C++的文件流操作非常灵活，可以在任意位置向文件添加内容。正如7.1.2节所述，C++新标准流库是在老标准流库的基础上加以修改并进行模板化，它们都采用缓冲器机制，流的定位实际上是对缓冲器定位，缓冲器的位置以字节为单位，用basic_ios类模板中的pos_type类型即4个字节的int型保存，并使用两个指针来指示缓冲器的位置，输入

流采用get指针、编程时用名gptr称为输入流指针，输出流采用put指针、编程时用名pptr称为输出流指针，输入/输出流同时具有输入流指针gptr和输出流指针pptr，且它们是互不干扰、互相独立的。

对文件的读写操作都是分别从输入流指针或输出流指针所指定的位置开始，为了确保文件的读写顺序，每读写一个字符输入流指针或输出流指针都自动增1指向下一个位置，即指针总是向文件结尾方向移动。但是，也可以通过定位操作控制指针随机地向前或向后移动。C++流的定位即指针的移动方式有3种，由ios_base类中定义的如下3个seek_dir枚举类型的变量指定：

- ios_base::beg：缓冲器的开始位置即相当于文件的开头，
- ios_base::cur：缓冲器的当前位置，
- ios_base::end：缓冲器的终止位置即相当于文件的结尾，

而seek_dir枚举类型定义如下：

```
ios_base::enum seek_dir { beg = 0; cur = 1; end = 2; }
```

另外，在文件定位控制中，还用到了如下一些新类型名：

- basic_ios::pos_type：用以表示某种绝对位置的类型，通常就是long型，
- basic_ios::off_type：用以表示某种相对位置的类型，通常就是long型，
- basic_ios::int_type：用以表示某种整数类型，如int或unsigned型，
- basic_ios::char_type：用以表示某种字符类型，如char型，

在输入流对象中，随机移动get指针的成员函数有如下几个：

```
basic_istream & seekg( pos_type pos );  
basic_istream & seekg( off_type off, ios_base::seek_dir dir );
```

它们在basic_istream类模板中定义，第1个成员函数将输入流中的get指针移动到从缓冲器的开始位置计算起偏移量为形参pos的绝对位置上，pos值以字节为单位，如图7.13所示。例如：

```
istream input; // input是类似于cin的输入流对象，例如像扫描仪的外部设备  
...  
seekg(0); // 把get指针重新移到缓冲器的开始位置
```

第2个也是用于移动get指针而使用更灵活的成员函数。如图7.13所示，该成员函数把get指针移动到从用第2个参数dir指定的起始位置计算起，偏移量为形参off的相对位置上，dir的类型说明为ios_base::seek_dir。

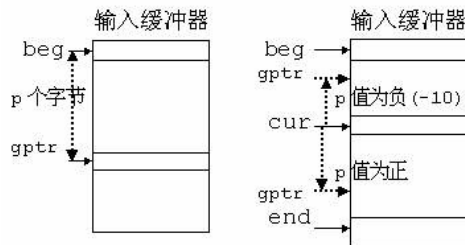


图7.13 移动get指针的成员函数seekg()

例如：

```
input.seekg( -10, ios_base::cur );
```

该seekg()的调用语句把起始点设定在get指针的当前位置，p值为-10，get指针向前即向缓冲器的起始位置推进10个字节，若p值为正，则向后即向缓冲器的终止位置推进。

在输入流中，如basic_istream类模板还备有一个成员函数，它返回到get指针的当前位置，即其返回值是从缓冲器起始位置算起到get指针当前位置的偏移量（参看图7.13）。其原型为：

```
pos_type tellg( void );
```

因此，如果想以字节为单位给出缓冲器的大小，可编写如下语句：

```
input.seekg( 0, ios_base::end );
```

```
long size = input.tellg( );
```

第1条语句将get指针移动到缓冲器的终止位置，第2条语句由tellg()的返回值是从缓冲器的起始位置算起一直到缓冲器终止位置的偏移量，也就是缓冲器的大小。若该缓冲器是为某磁盘文件而设置，则变量size所获得的值就是文件的大小。

顺便指出，对输入流只使用get指针，对输出流只使用put指针，在输出流ostream类中同样也重载了一系列处理put指针的成员函数，它们的原型是：

```
basic_ostream & seekp( pos_type pos );
```

```
basic_ostream & seekp( off_type off, ios_base::seek_dir dir );
```

```
pos_type tellp( );
```

其使用方法与istream类中的成员函数完全一样。

7.5.4 特殊的文件流

正如7.1.2所述，为了将外部设备与磁盘文件统一起来，C++也给外部设备引入了缓冲器机制，把实际的各种外部设备抽象化为“输入输出流的对象”，将流的概念进一步扩展成如图7.2所示的“广义的流”，其源点和终点既可以是内存块，也可以是外部设备。缓冲器机制的引入为编程者提供了一个统一的接口界面，使得流操作与被访问对象无关。即流库的I/O操作函数既可以用来处理磁盘文件，又可以用来控制外部设备（把外部设备称为“设备文件”）。这样一来，对外部设备的读/写与磁盘文件的读/写操作在编程方法上完全相同。因此，所谓“特殊的文件流”就是对计算机标准外部设备（即计算机控制台的外部设备包含键盘和显示器）进行输入/输出文件流操作。用"CON"（控制台的英文Console缩写）为文件名创建的输入流所关联的外部设备是计算机键盘，可用于键盘的输入控制。用"CON"为文件名创建的输出流所关联的外部设备是计算机控制台的显示器，可用来控制输出显示。

例7.9 控制台的文件流操作。

```
#include < iostream >          // 使用C++新标准的流库
#include < fstream >            // 使用C++新标准的文件流
using namespace std;           // 将std标准名空间合并到当前名空间

void main( void )
{   ifstream   fin( "CON" );    // 创建一个输入文件流对象fin与控制台键盘相关
    ofstream   fout( "CON" );   // 创建一个输出文件流对象fout与控制台显示器相关联
    ofstream   fprn( "PRN" );  // 创建一个输出文件流对象fprn与打印机相关联

    fin.tie( & fout );
    fout << "OK !" << endl << "Type a string : ";
    char str[20];
    fin >> str;
    fout << "String stored in str[20] : " << str << endl;
    fprn << "String stored in str[20] : " << str << endl;
}
```

该程序的输出结果为：

OK !

Type a string : abcdefghi

String stored in str[20] : abcdefghi

其中,“`fin.tie(&fout);`”语句调用了`basic_ios`类模板的成员函数`tie()`,其原型声明为:

```
basic_ostream < E, T > * tie( basic_ostream <E, T> * str );
```

实例化后的原型声明为:

```
ostream * tie( ostream * str );
```

它的功能是通过控制输入流指针和输出流指针将输入流对象`fin`和输出流对象`fout`或`fprn`进行同步控制,使得在每次进行输入操作前,强制性地命令输出流将已输出到流缓冲器的数据立即发送给像显示器或磁盘的输出设备。

7.5.5 格式化的输入/输出文件

格式化的输入/输出文件都是文本文件,即整个文件是由一个个`char`型单字符组成的字符序列。因此,对于输入文件流操作都可以在各种文本编辑器上,事先把要输入的数据文件制作好以作为应用程序的操作处理所备用的文件;同样,应用程序所生成的格式化数据文件也可以用像记事本这样的文本编辑器来阅读和查看。

作为输入源流对象的格式化文件,像`cin`(键盘)或某个输入文件流都具有一个流的结束标记,可以用7.5.3节所介绍的`basic_ios`类模板的成员函数`eof()`来检测流操作是否读到该结束标记,并将它保存在`ios_base::eofbit`状态标志中。对于与键盘相关联的输入流对象,操作者可同时按“`Ctrl + Z`”来插入该结束标记,但在编写程序中,对于每个`cin`语句(不是多个输入操作的组合,而只是进行一次输入操作),其后都应编写一条“`cin.clear();`”语句,以恢复`ios_base::eofbit`状态标志位,确保后面的`cin`语句能够正确执行。

例7.10 保存应用程序所生成的格式化数据文件。

```
#include < fstream >           // 使用C++新标准的文件流
#include < iomanip >           // 操作符函数原型声明在其中
#include < cmath >             // 标准数学函数的原型声明在其中
using namespace std;          // 将std标准名空间合并到当前名空间

void main( void )
{
    ofstream log10Table( "LOG10TABLE.TXT", ios_base::trunc );
    // 创建一个输出文件流类ofstream的对象log10Table并打开相关联的空文件
    log10Table << "TEN ";
    // 向log10Table对象输出第0列(纵列)表头为十位数(TEN)

    for( int i = 0; i < 10; i++ )
        log10Table << setw( 5 ) << i << " ";
    // 向log10Table对象输出各纵列表头数字1 ~ 9(表示个位数),设置输出字符宽度为5
```



```

7      1.845  1.851  1.857  1.863  1.869  1.875  1.881  1.886  1.892
1.898
8      1.903  1.908  1.914  1.919  1.924  1.929  1.934  1.940  1.944
1.949
9      1.954  1.959  1.964  1.968  1.973  1.978  1.982  1.987  1.991
1.996

```

由此可见，对于格式化控制的输入/输出流，如这种格式化输入输出文件的流操作，采用操作符函数可以大大简化程序结构，创建一个输出文件流类的对象log10Table后，设置域宽、设置输出精度和输出数据等操作，一气呵成地完成一系列流操作。

例7.11 编写一个BuildData类模板从BuildData_base（普通）类继承而来，该类模板能把从键盘敲入的一组指定数据类型的数据保存在程序指定的文件（DATA1.TXT和DATA2.TXT）中，并将文件所保存的数据输出到显示器上。为了满足输入/输出字符串的特殊要求，需要定义一个补充模板类。

```

#include    < iostream >        // 使用C++新标准的流库
#include    < fstream >         // 使用C++新标准的文件流
#include    < cstring >         // 处理char型字符串的原型声明在其中
#define    BUFSIZE    81        // 定义符号常量BUFSIZE保存缓冲器数值大小
using namespace std;           // 将std标准名空间合并到当前名空间
// 定义一个普通类BuildData_base作为BuildData类模板和其补充模板类的基类
class BuildData_base {
protected :
    char *   fileName;          // 保护数据成员fileName为字符串指针保存文件名字符串

    BuildData_base( const char * fname )
    {   fileName = new char[strlen( fname ) + 1];
        // 用new运算符在内存的堆中开辟一个空间保存文件名字符串
        strcpy( fileName, fname );
        // 将形参字符串指针fname所指的的文件名字符串拷贝到保护数据成员fileName保存

    }
    ~ BuildData_base( void )// 析构函数
    {   delete [ ] fileName;   }
    // 用delete运算符释放fileName所指的字符串堆空间
};

template < typename T >        // 定义BuildData类模板，模板形参T为虚拟类型

```

```

class BuildData : public BuildData_base {
    T buffer;           // 私有数据成员为虚拟类型T的对象
public :
    BuildData( const char * fnm ) : BuildData_base( fnm ) { }
    // 派生类模板BuildData的构造函数，其函数体为空
    int keyInData( void ) // 读取从键盘输入数据的成员函数
    {
        ofstream fout( fileName, ios_base::trunc );
        // 创建输出文件流类ofstream的对象fout，并打开相关联的空文件fileName
        if( fout.fail( ) ) return 1;
        /* 也可以写成"if( fout.fail( ) != NULL )"，调用成员函数fail( )检测文
            件打开是否失败？当文件打开失败该成员函数的返回值为1 */
        while( cin >> buffer )
            fout << buffer << endl;
        /* 从键盘读取一组字符序列写入到输出文件流类对象fout中，直到遇到流结束标记
            或
            流操作出错为止 */
        cin.clear( );
        fout.close( ); // 关闭输出文件流类对象fout
        return 0;      // 当成功完成数据输入，则该成员函数的返回值为0
    }
    int showData( void )
    {
        ifstream fin( fileName );
        // 创建输入文件流类ifstream的对象fin，并打开相关联的已存在文件fileName
        if( fin.fail( ) ) return 1;
        /* 也可以写成"if( fin.fail( ) != NULL )"，调用成员函数fail( )检测文
            件
            打开是否失败？当文件打开失败时该成员函数的返回值为1 */
        while( fin >> buffer )
            cout << buffer << endl;
        /* 从对象fin相关联的文件中读取一组字符序列发送给显示器，直到遇到流结束标记
            或
            流操作出错为止 */
        fin.close( ); // 关闭输入文件流类对象fin
        return 0;     // 当成功完成数据显示，则该成员函数的返回值为0
    }
};

template < >           // 针对字符串的输入/输出流操作定义一个补充模板

```

```

class BuildData < char * > : public BuildData_base {
    char buffer[BUFSIZE]; // 私有数据成员buffer指定为字符串数组
public :
    BuildData( const char * fnm ) : BuildData_base( fnm ) { }
    int keyInData( void )
    {
        ofstream fout( fileName, ios_base::trunc );
        // 创建输出文件流类ofstream的对象fout，并打开相关联的空文件fileName
        if( fout.fail( ) ) return 1;
        /* 也可以写成"if( fout.fail( ) != NULL )"，调用成员函数fail( )检测文件打开是否失败？当文件打开失败时该成员函数的返回值为1 */
        while( cin.getline( buffer, BUFSIZE ) )
            fout << buffer << endl;
        /* 从键盘读取一行字符串写入到输出文件流类对象fout中，直到遇到流结束标记或
流
        操作出错为止 */
        cin.clear( );
        fout.close( ); // 关闭输出文件流类对象fout
        return 0; // 当成功完成数据输入，则该成员函数的返回值为0
    }
    int showData( void )
    {
        ifstream fin( fileName );
        // 创建输入文件流类ifstream的对象fin，并打开相关联的已存在文件fileName
        if( fin.fail( ) ) return 1;
        /* 也可以写成"if( fin.fail( ) != NULL )"，调用成员函数fail( )检测文件
件
        打开是否失败？当文件打开失败时该成员函数的返回值为1 */
        while( fin.getline( buffer, BUFSIZE ) )
            cout << buffer << endl;
        /* 从对象fin相关联的文件中读取一行字符串发送给显示器，直到遇到流结束标记或
流
        操作出错为止 */
        fin.close( ); // 关闭输入文件流类对象fin
        return 0; // 当成功完成数据显示，则该成员函数的返回值为0
    }
};

void main( void )
{
    BuildData < int > intData( "DATA1.TXT" );

```



```

/* 创建一个BuildData < int >模板类的对象intData，并打开相关联的空文件
   DATA1.TXT */
cout << "After enter a integer, type Ctrl + Z : " << endl;
// 输出“从键盘敲入一个整数和回车键后，再按压‘Ctrl + Z’”的提示信息字符串
intData.keyInData( );
// 调用成员函数keyInData( )将键盘输入的整数写入到DATA1.TXT文件
cout << endl << "The enter integer : " << endl;
// 输出显示提示信息字符串
intData.showData( );
// 调用成员函数showData ( )读取DATA1.TXT文件的整数发送到显示器
BuildData < char * > strData( "DATA2.TXT" );
/* 创建一个BuildData < char * >模板类的对象strData，并打开相关联的空文件
   DATA2.TXT */
cout << "After enter a string, type Ctrl + Z : " << endl;
// 输出“从键盘敲入一个字符串和回车键后，再按压‘Ctrl + Z’”的提示信息字符串
strData.keyInData( );
// 调用成员函数keyInData( )将键盘输入的字符串写入到DATA2.TXT文件
cout << endl << "The enter a string : " << endl;
// 输出显示提示信息字符串
strData.showData( );
// 调用成员函数showData ( )读取DATA2.TXT文件的整数发送到显示器
}

```

该程序的输出结果为：

```

After enter a integer, type Ctrl + Z :
31415926
The enter integer :
31415926
After enter a string, type Ctrl + Z :
Welcom to WUHAN !
The enter a string :
Welcom to WUHAN !

```

由此可见：

对于含有虚拟类型的类模板，若需要处理字符串的输入/输出流操作时，必须针对字符串定义一个补充模板，而含有虚拟类型的类模板称为“主模板”，补充模板就是该类模板特例化的一个实例。如例7.11中，先定义的BuildData类模板为主模板，后定义的BuildData < char * >类模板为主模板的形参虚拟类型T指定为字符串指针的

补充模板，其定义格式为：

```
template      < >      class  主模板名 < 非虚拟  
类型 >  { ... };
```

即template后面的模板形参为空，它的类模板名与主模板名相同，然后，在其类体内针对某种特殊的数据类型编写具体的处理程序。读者试比较例7.11中的主模板和补充模板的具体程序代码可找出它们的区别进而总结编程要点。

本程序在操作时应注意，当输入了一个整数或char型字符串再键入回车键后，此时程序还没有退出while循环，操作者应同时按压Ctrl键和Z键，即敲入“^Z”向输入流插入流结束标记，程序才会执行接下来的“cin.clear();”语句，使得标准输入流对象cin恢复使用状态。

7.6 内存格式化I/O流

在实际编程时，经常要处理各种类型的数据，有字符型、整数型、浮点型、结构体、类等，这些数据若以文件方式存放在内存中以备使用，则显得太冗长，特别是在某些场合不太实用，例如使用Windows图形用户接口在图形屏幕上显示各种数据，但是若采用二进制方式，在调试时又很难辨认。编程者必须格式化内存中的数据，即把数据转换成便于管理、存储、传送和显示等的单字节序列形式，将其保存在内存缓冲器中，以便在使用时随时调用这些格式化的数据。

在C++新标准流库的头文件<strstream>中，定义了istrstream、ostrstream和strstream等类，其中ostrstream类从ostream类继承而来，它用于将不同类型的数据进行内存格式化，变成为C语言中的char型字符串，并放到一个作为缓冲器的字符数组中。istrstream类是从istream类继承而来的，它用来将文本方式的字符串转换成变量所需要的内部格式，而strstream类从iostrstream类继承而来，可用来进行两种转换。在使用它们的源程序中必须写上：

```
#include      < strstream >  
...  
using      namespace      std;
```

使用上述的类可以方便地实现如下转换：

将一个字符串内的数字字符转换成二进制形式，存放在某种类型的对象中。

将一个二进制数转换成字符串，存放在一个作为缓冲器的字符数组中。

例7.12 内存格式化流操作。

```
#include < iostream >
#include < fstream >
#include < stringstream >
#include < iomanip >
#include < cstdlib >
using namespace std;
void main( void )
{
    char    buf[80];    // 定义一个char型字符串数组
    stringstream    buffer( buf, sizeof( buf ), ios_base::out );
    /* 创建一个字符串流类stringstream的对象buffer，它将字符串数组buf[80]当做
```

字符串

流缓冲器stringstreambuf，即向对象buffer写入数据也就是向字符串数组buf[80]写入数据 */

```
int    number = 30;
char    * color = "RED";
double weight = 135.96;
buffer << "There are " << number;
buffer << " chair that are " << color;
buffer << ",\nwith a total weight of ";
// 向对象buffer即向字符串数组buf[80]写入一系列数据
buffer << fixed << setprecision(2) << weight << " kilos." << '\0';
// 采用操作符函数设置输出数据为浮点形式，设置实型数输出精度为小数点后两位
ofstream    report("REPORT.TXT");
// 创建一个输出文件流类ofstream的对象report，并打开相关联的空文件
```

REPORT.TXT

```
if( ! report ) {
    cout << "\a Can't open report.doc.";
    exit(1);
}
// 当空文件REPORT.TXT不能打开时，输出出错信息并立即终止执行程序退回到操作
```

系统

```
report << buf ;
/* 当空文件REPORT.TXT成功地被打开，将字符串流缓冲器stringstreambuf即字符串数组
buf[80]的所有元素写入REPORT.TXT文件 */
report.close( );
buffer.seekp(0);
```

```

        // 将缓冲器的put指针复归，指向缓冲器的首地址，以备下次使用
        cout << buf << endl;
    }

```

该程序的输出结果为：

```

There are 30 chair that are RED,
with a total weight of 135.96 kilos.

```

`stringstream`类的构造函数：创建一个`stringstream`类的对象`buffer`来建立格式化字符串，`stringstream`类具有两个重载的构造函数。其原型是：

```

stringstream( );
stringstream( char * s, streamsize n,
             ios_base::openmode mode = ios_base::in | ios_base::out);

```

第一个构造函数没有参数，用于创建`stringstream`类的对象，并管理它的动态缓冲器。当所有插入字符大于缓冲器时，会自动扩大。第二个构造函数的第1个参数是指向缓冲器的指针`s`，第2个参数`n`用来设置缓冲器的大小，第3个形参`mode`用于设置`put`指针的初始方式，而不影响`get`指针，可设置如下值：`ios_base::out`或`ios_base::ate`或`ios_base::app`。第1个值是正常操作，`put`指针初始化为指向缓冲器的开始位置，`ios::ate`和`ios::app`使`put`指针指向缓冲器终止位置，此时，缓冲器以`NULL`结尾，如果其中存储的是二进制数据，则采用`ios_base::ate`和`ios_base::app`打开模式标志的函数就不能正确操作，因此不能确定数据的正确性。

所以，例7.12中自定义一个字符数组`buf[80]`作为缓冲器来格式化数据，即存放在其内的是一个个字符格式的数据，然后将该字符串存入文本文件`REPORT.TXT`中。

字符串流写入文本文件：打开一个文本文件`REPORT.TXT`的方法与前述完全相同，而写入文本文件的方法非常简单，只需写“`report << buff;`”语句，即可将缓冲器格式化的数据写入文件`REPORT.TXT`中。

输入输出字符串流：用自定义的简单输入输出字符串流`buffer`能支持多种内存格式化操作，本例中先将`buf`缓冲器的格式化数据写入到文本文件`REPORT.TXT`中，然后再将这些数据送到显示器显示。在重新使用`buffer`前，应将`put`指针卷回到0值，使其指向缓冲器的首地址，以备下次使用。这种在同一函数中执行多个格式化操作，例如在使用Windows的图形用户界面时，必须先格式化内存中的数据，然后调用界面函数显示这些数据，这是经常采用的方法。

小 结

Visual C++V6.0系统中包含有两个I/O流库，即C++新标准流库和老标准流库。前者是在后者的基础上进行了修改并模板化，从而为用户提供了功能更强大、使用更灵活的流操作系统，前者是本章学习的重点内容。读者在编写实用化程序时应尽量采用C++新标准流库。

使用C++新标准流库编写程序时，应该在文件的开头处用#include预处理语句把相关的头文件纳入到当前程序：当使用cin、cout、cerr和clog等预定义标准流对象进行标准外部设备的I/O操作时，必须包含<iostream>头文件；若使用文件流对象进行文件的I/O操作时，必须包含<fstream>头文件；当使用setw、fixed和setprecision等操作符函数时，必须包含<iomanip>头文件。与此同时，还必须用如下声明语句把标准名空间std引入到当前程序：

```
using namespace std;
```

在熟悉C++新标准流库类层次结构图的基础上，应用面向对象程序设计的3个要点（参阅1.2节），理顺各个层次结构的类模板和类之间的继承关系。在理解的基础上，掌握和记忆它们所包含的数据成员和成员函数及其使用方法。C++新标准流库类层次结构图中，最顶层的“根”类是ios_base类，由它派生出一个类模板basic_ios< E , T >，当模板形参E指定为char型（即C语言中的字符类型）、形参T采用默认值char_traits<char>时，则实例化为“basic_ios< char , char_traits<char> >”模板类，再用typedef语句给这个较长模板类名启用一个新类型名ios即得到ios流类。其他流类如法炮制，例如，basic_istream< E , T >和basic_ostream< E , T >类模板都是从类模板basic_ios< E , T >继承而来，实例化得到后的新类型名分别为istream和ostream，如此类推……

C++流库把标准运算符集合中的“>>”和“<<”运算符分别重载成输入运算符（在basic_istream< E , T >类模板体内）和输出运算符（在basic_ostream< E , T >类模板体内）。编程者也可以在自行定义的class类型的类体内重载这两个运算符。

输入输出流操作分为无格式控制和格式控制两种。前者主要针对磁盘、磁带和光盘上的二进制文件，后者用于磁盘上的文本文件和键盘、显示器以及打印机等字符设备。格式控制的编程方法之一是调用ios_base类的成员函数setf()设置指定的格式标志，而用它的另一个成员函数unsetf()可清除指定的格式标志。

格式控制的编程方法之二是使用操作符函数，它们可以直接与输入运算符“<<”和输出运算符“>>”配合使用，可把多个操作用一条语句一气呵成。很多操作符启用的标识符与对应的格式标志相同，常用的操作符有：

- endl ：插入换行符，与'\n'的功能相同，
- left ：在设定的域宽内左对齐，右端空白处填充设定的字符，
- right ：在设定的域宽内右对齐，左端空白处填充设定的字符，
- fixed ：浮点数按定点格式输出，

- `setw` : 设置输入和输出的域宽,
- `scientific` : 浮点数按指数格式(科学表示法)输出,
- `setprecision` : 设置浮点数的精度(有效位数或小数位数),
- `showpoint` : 无条件地显示小数部分,
- `noshownpoint` : 清除`showpoint`操作符的设置,不显示小数部分,
- `setfill` : 设置填充字符。

要打开一个文件进行读操作,必须创建其类型为`ifstream`类的流对象;要打开一个文件进行写操作,必须创建一个类型为`ofstream`类的流对象;若要打开一个既要读又要写的文件,则必须创建其类型为`fstream`类的流对象。可以在创建一个文件流对象的同时,通过文件流类的构造函数自动调用其成员函数`open()`打开相关联的文件,由该函数的第2个形参指定打开模式,它是由表7.3所示的模式标志常量指定的,这些模式标志常量都是`ios_base`类的公有静态数据成员。

通过文件流对象调用其成员函数可以检测文件流的状态标志,即:

- `basic_ios::clear()` : 若发生异常则抛出一个`failure`类(为`ios_base`类服务的成员类)的对象,
- `basic_ios::good()` : 所进行的流操作成功时返回`true`,否则返回`false`,
- `basic_ios::fail()` : 流操作失败时返回`true`,否则返回`false`,
- `basic_ios::bad()` : 流缓冲器完整性被破坏时返回`true`,否则返回`false`,
- `basic_ios::eof()` : 流操作进行到文件结尾时返回`true`,否则返回`false`,
- `is_open()` : 检测流对象是否与一个打开的文件相关联。

C++流的定位操作主要用于文件流,采用缓冲器机制,流的位置以字节为单位,使用两个指针来指示缓冲器的位置,即输入流指针`gptr`和输出流指针`pptr`。C++流的定位即指针的移动方式有3种,由`ios_base`类中定义的如下3个`seek_dir`枚举类型的变量指定:

- `ios_base::beg` : 缓冲器的开始位置即相当于文件的开头,
- `ios_base::cur` : 缓冲器的当前位置,
- `ios_base::end` : 缓冲器的终止位置即相当于文件的结尾,

另外,在文件定位控制中,还用到了如下一些新类型名:

- `basic_ios::pos_type` : 用以表示某种绝对位置的类型,通常就是`long`型,
- `basic_ios::off_type` : 用以表示某种相对位置的类型,通常就是`long`型,
- `basic_ios::int_type` : 用以表示某种整数类型,如`int`或`unsigned`型,
- `basic_ios::char_type` : 用以表示某种字符类型,如`char`型。

与输入文件流定位有关的成员函数有:

- `basic_istream::seekg()` : 把`get`指针移动到指定的绝对或相对位置上,
- `basic_istream::tellg()` : 返回到`get`指针的当前位置,

与输出文件流定位有关的成员函数有:

- `basic_ostream::seekp()` : 把put指针移动到指定的绝对或相对位置上,
- `basic_ostream::tellp()` : 返回到put指针的当前位置。

以CON为文件名的输入文件流对象对应于键盘的输入控制,以CON为文件名的输出文件流对象对应于显示器的输出控制,以PRN为文件名的输出文件流对象对应于打印机的输出控制,统称为特殊的文件流。

无格式控制的输入/输出流只用于像磁盘、光盘和磁带等设备的二进制文件,有关的成员函数有:

- `basic_istream::get()` : 读取并返回当前输入位置(即get指针所指的位置)的字符代码或将读取的字符存入指定的变量中,
- `basic_istream::getline()` : 读取一行字符串存入到指定的字符缓冲器中,
- `basic_istream::ignore()` : 读取并丢弃从当前位置开始的若干字符,
- `basic_istream::peek()` : 返回当前位置上的字符,但get指针所指的输入位置不变,
- `basic_istream::putback()` : 将指定字符回送给输入流,
- `basic_istream::read()` : 读取指定字节数的数据块存放在指定的字符空间中,
- `basic_ostream::put()` : 输出指定的单个字符,
- `basic_ostream::write()` : 输出指定字符空间中的指定字节数的数据块。

习 题 7

一、选择填空

1. 进行文件操作时需要包含()文件。
A. `iostream.h` B. `fstream.h` C. `stdio.h` D. `stdlib.h`
2. C++流库中将其标准运算符集合的“<<”运算符重载了,它是一个_____。
A. 用于输出操作的成员函数 B. 用于输入操作的成员函数
C. 用于输入操作的非成员函数 D. 用于输出操作的非成员函数
3. 已知:`int a, * pa = &a;` 输出指针pa的地址值(十进制)的方法是()。
A. `cout << pa;` B. `cout << * pa;`
C. `cout << &pa;` D. `cout << long(pa);`
4. 下列输出字符A的方法中,()是错误的。
A. `cout << put('A');` B. `cout << 'A';`
C. `cout.put('A');` D. `char A = 'A'; cout << A;`
5. 关于`getline()`函数的下列描述中,()是错误的。
A. 该函数是用来从键盘上读取字符串的

- B. 该函数读取的字符串长度是受限制的
C. 该函数读取字符串时遇到终止符就停止
D. 该函数中所使用的终止符只能是换行符
6. 关于read()函数的下列描述中,()是正确的。
A. 该函数只能从键盘中读取字符串
B. 该函数读取的字符串长度是不受限制的
C. 该函数只能用于文本文件的操作
D. 该函数只能按规定读取所指定的字符数
7. ios类提供控制格式标志位中,()是指定转换十六进制形式的标志位。
A. hex B. oct ; C. dec D. left
8. "ofstream f (Personal.dat", ios_base::app |
 ios_base::binary);"语句的功能是建立流对象f,并试图打开文件与之相链接,
 且()。
A. 若文件存在,将文件指针定位在文件开头,若文件不存在则建立一个新文件
B. 若文件存在,将它变成空文件,若文件不存在则打开失败
C. 若文件存在,将文件指针定位在文件尾,若文件不存在则建立一个新文件
D. 若文件存在则打开失败,若文件不存在则建立一个新文件
9. 在打开磁盘文件的访问方式中,()是以追加方式打开文件。
A. in B. out C. app D. ate
10. 当进行任何C++流操作后,都可以用C++流类的有关成员函数检测流的状态,其中
 只能用于检测输入操作的成员函数的函数名是()。
A. fail B. eof C. bad D. good
11. 执行如下两条语句:

```
ofstream   outfile( "Personal.dat" );  
if( ... )   cout << "OK";  
else        cout << "FAIL";
```

后,若文件成功地打开则显示OK字符串,否则显示FAIL字符串。由此可知,上面if语句圆括号内"..."处的表达式为()。

- A. outfile.fail()或outfile B. outfile.good()或 !outfile
C. outfile.good()或outfile D. outfile.fail()或 !outfile
12. 下列函数中,()是对文件进行写操作。
A. get() B. read() C. seekg() D. put()

二、判断下列描述的正确性，对者划 ☐ ，错者划 ×

1. 使用输出运算符"<<"可以输出各种类型的变量的值，也可以输出指针值。 (☐)
2. 预定义的插入符从键盘上接收数据是不带缓冲区的。
(☐)
3. 预定义的提取符和插入符是可以重载的。
(☐)
4. get()函数不能从流中读取终止字符，终止字符仍留在流中。getline()函数从流中读取终止字符，但终止字符被丢弃。
5. 使用打开文件函数open()之前，要定义一个流类对象，可使用open()函数来操作该对象。
6. 使用关闭文件函数close()关闭一个文件时，其流对象仍存在。
7. 以app方式打开文件时，当前的读指针和写指针都定位于文件尾。
8. 打开ASCII码流文件和二进制流文件时，打开方式是相同的。
9. read()和write()函数可以读写文本文件，也可以读写二进制文件。
10. seekg()函数和seekp()函数分别用来定位gp指针和pp指针。

三、填空题

1. 在ios_base类中定义的、用于控制输入输出格式的枚举常量中，控制对齐方式的3个常量名是_____。
2. 表达式"cout << hex"还可以表示为_____。
3. 在ios_base类中定义的、用于控制输入输出格式的枚举常量中，控制浮点数表示形式（科学表示法，定点表示法）的2个常量名是_____。
4. 表达式"cout << \"\\n\""还可以表示为_____。
5. 试有如下程序：

```
#include <iostream>
#include <fstream>
using namespace std;
void main( void )
{
    char s[25] = "Programming language";
    ofstream f1( "DATA.TXT" );
    f1 << "C++ Programming";
    f1.close( );
    ifstream f2( "DATA.TXT" );
    if( f2.good( ) ) f2 >> s;
```

```

        f2.close( );
        cout << s;
    }

```

该程序的输出结果为_____。

6. 如下程序的输出结果是

```

5.00000
5
+5

```

请将程序中的空白处补充完整：

```

#include    < iostream >
#include    < iomanip >
using namespace std;
void main( void )
{
    double    x = 5;
    cout << _____ << x;
    cout << endl << _____ << x;
    cout << endl << _____ << x;
}

```

7. 如下程序的输出结果是

```

TTTTTTTTTT5.23
5.23TTTTTTTTTT

```

请将程序中的空白处补充完整：

```

#include    < iostream >
#include    < iomanip >
using namespace std;
void main( void )
{
    double    x = 5.23;
    cout << _____ << setw( 14 ) << x;
    cout << endl << left << setw(_____) << x;
}

```

8. 试有如下程序：

```

#include    < iostream.h >
class AT {
    friend ostream & operator << ( ostream & , AT );
} at;
ostream & operator << ( ostream & os, AT )

```

```

{   return    os << '@';   }
int  main( void )
{   cout << "MyHome" << at << "isHere.com" << endl;
    return  0;
}

```

该程序的输出结果为_____。

9. 试有如下程序：

```

#include< iostream >
#include< fstream >
using namespace std;
int  main( void )
{   ofstream  outf( "D:\\temp", ios_base::trunc );
    outf << "World Wide Web";
    outf.close( );
    ifstream  inf( "D:\\temp" );
    char  str[20];
    inf >> str;
    inf.close( );
    cout << str << endl;
    return  0;
}

```

该程序的输出结果为_____。

四、分析下列程序的输出结果

程序1：

```

#include    < iostream >
using namespace std;
void  main( void )
{   cout.width(20);
    cout.fill( '*' );
    cout.setf(ios::hex | ios::right);
    cout << "abcdefghi\n";
    cout.unsetf(ios::hex | ios::right);
    cout.width(20);
    cout.fill( '*' );
    cout.setf(ios::hex | ios::left);
    cout << "abcdefghi";
}

```

```

        cout << "\n";
    }

```

程序2：

```

#include    < iostream >
#include    < fstream >
#include    < cstdlib >
using namespace std;
void main( void )
{
    ifstream    fl( "D:\\file1.cpp" );
    if( ! fl ){
        cout << "\a Can't open file1.cpp.\n";
        exit( 1 );
    }
    while( fl && cout ) {
        char buff[100];
        fl.getline( buff, sizeof( buff ) );
        cout << "\n" << buff;
    }
}

```

程序3：

```

#include    < iostream >
#include    < fstream >
#include    < cstdlib >
using namespace std;
void main( void )
{
    fstream    outfile, infile;
    outfile.open( "text.dat", ios::out );
    if( ! outfile ) {
        cout << "text.dat can't open.\n";
        abort( );
    }
    outfile << "123456789\n";
    outfile << "aaabbbbbbbccc\n" << "ddddfffeeeegggghhh\n";
    outfile << "OK !\n";
    outfile.close( );
}

```

```

infile.open( "text.dat", ios::in );
if( ! infile ) {
    cout << "file can't open.\n";
    abort( );
}
char    textline[80];
while( ! infile.eof( ) ) {
    infile.getline( textline, sizeof( textline ) );
    cout << textline << endl;
}
}

```

程序4：

```

#include    < iostream >
#include    < fstream >
#include    < cstdlib >
using namespace std;
void main( void )
{
    fstream    file1;
    file1.open( "text1.dat", ios::out | ios::in );
    if( ! file1 ) {
        cout << "text1.dat can't open.\n";
        abort( );
    }
    char    textline[ ] = "123456789abcdefghijkl.\n\0";
    for( int i = 0; i < sizeof( textline ); i++)
        file1.put(textline[i]);
    file1.seekg(0);
    char    ch;
    while( file1.get( ch ) )
        cout << ch;
    file1.close( );
}

```

程序5：

```

#include    < strstrea.h >
void main( void )

```

```

{   ostream    ss;
    ss << "Hi, Good morning !\n\0";
    char *   buf = ss.str( );
    cout << buf << endl;
}

```

程序6：

```

#include    < strstrea.h >
#include    < iostream.h >
char    a[ ] = "1000";
void    main( void )
{   int    dval, oval, hval;
    istream    iss( a, sizeof( a ) );
    iss >> dec >> dval;
    iss.seekg( ios::beg );
    iss >> oct >> oval;
    iss.seekg( ios::beg );
    iss >> hex >> hval;
    cout << "decVal : " << dval << endl;
    cout << "octVal : " << oval << endl;
    cout << "hexVal : " << hval << endl;
}

```

五、编程题

1. 编写一个程序，从键盘输入一个表示生日的8位十进制数，它能分年、月、日显示这个生日。例如，若输入“19690809”，则输出为：

出生年： 1969

出生月： 8

出生日： 9

2. 函数parentheses()的原型为：

```
const char * parentheses( const char * filename );
```

它接收一个C++源程序文件名，检查该文件源程序中的括弧是否配对，括弧包括()、[]和{ }3种。不同括弧互相嵌套时必须完整嵌套，即像“{ ... [...] ... }”这样的嵌套虽然是配对的但却是错误的。函数在发现第1个错误或整个文件扫描完成时结束，并返回一个描述扫描结果的字符串。例如：

“文件...括弧嵌套正确！”

“文件...第...行的‘（’没有配对的‘）’！”

“文件...第...行的括弧未能正确嵌套！”

试编写并调试这个函数。

附录 ASCII 码表

ASCII 码是 American Standard Code for Information Interchange (美国国家标准信息交换码) 的缩写, 由美国国家标准化协会 (American National Standard Institute, 缩写 ANSI) 制定, 它给出了 128 个字符的 3 种进制的 ASCII 代码值。

字符	十进制	八进制	十六进制	字符	十进制	八进制	十六进制	字符	十进制	八进制	十六进制	字符	十进制	八进制	十六进制
NULL	0	000	00	SP	32	040	20	@	64	100	40	'	96	140	60
SOH	1	001	01	!	33	041	21	A	65	101	41	a	97	141	61
STX	2	002	02	"	34	042	22	B	66	102	42	b	98	142	62
ETX	3	003	03	#	35	043	23	C	67	103	43	c	99	143	63
EOT	4	004	04	\$	36	044	24	D	68	104	44	d	100	144	64
END	5	005	05	%	37	045	25	E	69	105	45	e	101	145	65
ACK	6	006	06	&	38	046	26	F	70	106	46	f	102	146	66
BEL	7	007	07	,	39	047	27	G	71	107	47	g	103	147	67
BS	8	010	08	(40	050	28	H	72	110	48	h	104	150	68
HT	9	011	09)	41	051	29	I	73	111	49	i	105	151	69
LF	10	012	0A	*	42	052	2A	J	74	112	4A	j	106	152	6A
VT	11	013	0B	+	43	053	2B	K	75	113	4B	k	107	153	6B
FF	12	014	0C	,	44	054	2C	L	76	114	4C	l	108	154	6C
CR	13	015	0D	-	45	055	2D	M	77	115	4D	m	109	155	6D
SO	14	016	0E	.	46	056	2E	N	78	116	4E	n	110	156	6E
SI	15	017	0F	/	47	057	2F	O	79	117	4F	o	111	157	6F
DLE	16	020	10	0	48	060	30	P	80	120	50	p	112	160	70
DC1	17	021	11	1	49	061	31	Q	81	121	51	q	113	161	71
DC2	18	022	12	2	50	062	32	R	82	122	52	r	114	162	72
DC3	19	023	13	3	51	063	33	S	83	123	53	s	115	163	73
DC4	20	024	14	4	52	064	34	T	84	124	54	t	116	164	74
NAK	21	025	15	5	53	065	35	U	85	125	55	u	117	165	75
SYN	22	026	16	6	54	066	36	V	86	126	56	v	118	166	76
ETB	23	027	17	7	55	067	37	W	87	127	57	w	119	167	77
CAN	24	030	18	8	56	070	38	X	88	130	58	x	120	170	78
EM	25	031	19	9	57	071	39	Y	89	131	59	y	121	171	79
SUB	26	032	1A	:	58	072	3A	Z	90	132	5A	z	122	172	7A
ESC	27	033	1B	;	59	073	3B	[91	133	5B	{	123	173	7B
FS	28	034	1C	<	60	074	3C	\	92	134	5C		124	174	7C
GS	29	035	1D	=	61	075	3D]	93	135	5D	}	125	175	7D
RS	30	036	1E	>	62	076	3E	^	94	136	5E	~	126	176	7E

参 考 文 献

[1] William Ford & William Topp. DATA STRUCTURES WITH C++. Prentice Hall, Inc. 北京：清华大学出版社，1997.

[2] 殷人昆，陶永雷，谢若阳. 数据结构(用面向对象方法与 C++描述). 北京：清华大学出版社，1999.

[3] 吕凤翥著. C++语言基础教程. 北京：清华大学出版社，1999.

[4] 钱能. C++程序设计教程. 北京：清华大学出版社，1999.

[5] [美]Michael J.Young. 邱仲潘等译. Visual C++ 6 从入门到精通. 北京：电子工业出版社，1999.

[6] Bjarne Stroustrup. The C++ Programming Language 3rd edition. Addison Wesley long man, 1997.

[7] Brian W. Kernighan & Dennis M. Ritchie. The C++ Programming Language. 1988.

[8] 张松梅. C++语言教程. 成都：电子科技大学出版社，1993.

[9] 宜晨. Visual C++ 5.0 实用培训教程. 北京：电子工业出版社，1998.

[10] Herbert Schildt. 杨长虹等译. 最新 C++语言精华(第 2 版). 北京：电子工业出版社，1997.

[11] Steve Oualine. 辛运韩等译. 使用 C++编程大全. 北京：电子工业出版社，1997.

[12] 王培杰等. 面向对象的 Windows 编程技术. 大连：大连理工大学出版社，1994.

[13] 牛允鹏. 全国计算机等级考试教程(二级)——C 程序设计. 北京：电子工业出版社，1996.

[14] 武延军，赵彬. 精通 ASP 网络编程. 北京：人民邮电出版社，2000.

[15] 王燕. 面向对象的理论与 C++实践. 北京：清华大学出版社，1997.

[16] 张国峰. C++程序设计实用教程. 北京：清华大学出版社，1996.

[17] Borland C++ 3.0 技术丛书. 北京：希望电脑公司，1993.

[18] C 言语研究会. C サンプル & ツール集, 日本：技术评论社，1986.

[19] C 言语研究会. C ツールライブラリ, 日本：技术评论社，1986.

[20] 严蔚敏，吴伟民. 数据结构(C 语言版). 北京：清华大学出版社，1997.

[21] Rowe Glenn W.. Introduction to Data Structures and Algorithms with C++. Prentice Hall Europe, 1997.

- [22] Rick Decker, Stuart HirshField. Working Classes : Data Structures and Algorithms Using C++. Boston. International Thomson Publishing Inc, PWS Publishing Co., 1997.
- [23] 边莫英. 软件技术基础. 天津: 天津大学出版社, 1993.
- [24] 章烨, 郑茜译. C++技术详解. 北京: 希望电脑公司, 1991.
- [25] 刘正林, 周纯杰编著, 最新C语言程序设计教程. 武汉: 华中科技大学出版社, 2003.
- [26] Scott Meyers 著, 侯捷译, Effective C++中文版 (2nd Edition). 武汉: 华中科技大学出版社, 2001.
- [27] 中国教育部考试中心, 全国计算机等级考试二级教程——C++语言程序设计. 北京: 高等教育出版社, 2004.