

新工科建设之路·计算机学科系列教材



C++ 面向对象程序设计

第 4 版

杜茂康 刘友军 主编 袁浩 李昌兵 王永 武建军 副主编



中国工信出版集团



电子工业出版社
Publishing House of Electronics Industry
http://www.phei.com.cn



C++ 面向对象 程序设计

第 4 版

杜茂康 刘友军 主编 袁浩 李昌兵 王永 武建军 副主编

电子工业出版社
Publishing House of Electronics Industry
北京 • BEIJING

内 容 简 介

本书以 C++ 14/17/20 标准为指引,深入浅出地介绍了标准 C++面向对象程序设计的相关知识,包括 C++对 C 语言的扩展以及类、对象、友元、继承、多态、虚函数、重载、I/O 流类库、文件、模板与 STL、异常、多线程等内容。全书本着易于理解、实用性强的原则设计其内容和案例,并以一个规模较大的综合性程序贯穿于 C++面向对象编程的全过程,引领读者理解和掌握面向对象程序设计的思想、方法和技术,以及运用 C++设计自定义类进行软件开发的方法。

本书取材新颖,内容全面,通俗易懂,可作为高等院校计算机、电子信息类专业及其他理工类相关专业和信息管理与信息系统等专业的教材,也可作为 C++语言自学者或程序设计人员的参考用书。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有,侵权必究。

图书在版编目(CIP)数据

C++面向对象程序设计 / 杜茂康, 刘友军主编. —4 版. —北京: 电子工业出版社, 2024.3

ISBN 978-7-121-47490-3

I. ① C… II. ① 杜… ② 刘… III. ① C++语言—程序设计 IV. ① TP312.8

中国国家版本馆 CIP 数据核字(2024)第 053967 号

责任编辑: 章海涛

印 刷: 三河市双峰印刷装订有限公司

装 订: 三河市双峰印刷装订有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本: 787×1 092 1/16 印张: 27.5 字数: 704 千字

版 次: 2007 年 5 月第 1 版

2024 年 3 月第 4 版

印 次: 2024 年 3 月第 1 次印刷

定 价: 68.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888, 88258888。

质量投诉请发邮件至 zlts@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式: 192910558 (QQ 群)。

前 言

与传统的面向过程程序设计相比较,面向对象程序设计降低了软件开发的复杂度,提高了软件开发的效率,更容易开发出具有高可靠性、可重用性和易维护性的软件,是当前软件开发的主流技术,是每个软件开发人员必须具备的基本能力。C++是在 C 语言基础上扩展了面向对象机制而发展起来的一门程序设计语言,程序结构清晰,代码简洁,可移植性强,支持数据抽象、面向过程和面向对象程序设计,具有面向过程和面向对象双重程序语言特性。

C++语言因其稳定性、高效性、兼容性和可扩展性而被广泛应用于各种专业领域和系统级的程序设计中,常被用来设计操作系统(如 UNIX、Windows、macOS)、设备驱动程序或者其他需要在实时约束下直接操作硬件的软件。计算机图形学、用户界面设计、游戏、虚拟现实、网络软件、嵌入式程序是使用 C++语言最深入的专业领域,银行、贸易、保险业、远程通信、军事等领域的应用程序,常采用 C++语言设计其核心代码,以保证软件性能和开发效率。

无论从编程思想、代码效率、程序的稳定性和跨平台性,还是从语言本身的实用性来讲,C++语言都是面向对象程序设计语言的典范。学好 C++语言,读者不但能够用于实际的程序开发,而且有助于理解面向对象程序设计技术的精髓,再学习诸如 Java、C#、Python、Go、Rust 之类的面向对象程序设计语言就简单了。

多年教学经验和编程实践给我们的真切体会是“[读教科书明其理,看技术书知其用](#)”。教科书的原理剖析和技术书的案例分析相结合,更利于读者深刻地理解和掌握 C++语言程序设计的基本原理和技术,更利于读者将学到的技术应用于实际的软件开发中。

本书正是基于这样的认知而编写的,兼具面向对象程序设计教材和技术图书的特点,既比较深入地介绍 C++面向对象程序设计的技术和原理,又清晰地介绍 C++程序的开发方法,且通过程序实例将两者较好地结合在一起。书中精心设计了一个[贯穿于全书大部分章节的规模较大的专业课程管理程序 comFinal](#),并不断地利用面向对象的 C++程序扩展其功能,使之成为一个较为完整的、综合了抽象、封装、继承、多态和运算符重载技术的程序,让读者掌握 C++面向对象程序设计的方法。

本书自 2007 年出版至今,受到了广大师生和软件开发人员的好评,得到了多所高校的认可并被一直选为教材,重印多次。许多读者发来求解书中疑问或索取习题参考答案的邮件,有的与作者探讨了 C++继承与多态的几个特殊问题、将 C++类移植到 Windows 程序中的方法,有的指出了书中的错误和不当之处。这些内容和 C++新标准的不断发布都是本书持续改进的源泉。本次修订对原书第 3 版的整体结构进行了较大调整,删除了过时及部分较难且不实用的内容,并用 C++新标准对一些程序案例进行了重新设计,主要体现在以下几方面:

(1) 删除了原书第 3 版第 10~12 章关于 C++ MFC 程序设计的全部内容,增加了多线程程序设计的内容。

(2) 以 C++ 11/14/17/20 标准为蓝本修订了各章内容,如智能指针、自动类型推断、lambda 函数、移动对象、构造函数的继承和委托、对象初始化列表、override 和 final、pair 和 tuple 容器、运算符重载、仿函数、元编程、noexcept 异常、多线程等内容,并在书中用“C++11”“C++14”“C++17”字样对首次出现的内容进行了标识,这些内容不能在 VC++ 6.0 这样的早

期编译器中进行编译，在支持 C++ 11/14/17/20 标准的编译环境中才能够正常运行。

(3) 注重面向对象程序设计和分析能力的培养。在介绍类、继承和多态设计时，更加重视对类设计的分析，并用 UML 方法进行建模，更利于面向对象程序设计思路的培养和形成。

(4) 按照 C++ 的新标准修订了全书例程，并在 Visual C++ 2022 环境中进行了运行测试。

本次修订后的教材分为 10 章。

第 1 章介绍面向对象程序设计的特征、C++ 程序的结构、数据的输入和输出等内容。

第 2 章介绍 C++ 11/14/17/20 标准对 C 语言非面向对象方面的扩展，包括智能指针、左值引用和右值引用、const 和 constexpr 常量、自动类型推断、范围 for、数据类型转换、lambda 表达式、重载、内联函数、命名空间、作用域等内容。

第 3~8 章介绍 C++ 面向对象程序设计的思想、特征和方法，包括类和对象、继承与复用、多态与虚函数、运算符重载与仿函数、模板与元编程、STL 程序设计、异常等内容。

第 9 章介绍 C++ 多线程程序设计，包括线程的基本概念、进程与线程的关系、线程运行原理和设计方法，以及简单的线程同步的实现。

第 10 章介绍 C++ 的流和文件的内容，包括 C++ 流类的成员函数，用输入流、输出流处理程序数据，创建、读写和格式化二进制文件、随机文件的方法等。

本书内容全面、析理深透、注重实用，精心设计了易于理解和示范性强的图形和案例程序，深入浅出地介绍了 C++ 面向对象程序设计的原理和各种技术标准，并对面向对象编程过程中容易发生的误解和错误进行重点分析，颇具启发性。

本书由杜茂康、刘友军、袁浩、李昌兵、王永、武建军编写。杜茂康编写了第 1、2、3 章，刘友军编写了第 4、5 章，袁浩编写了第 6、7 章，王永编写了第 8 章，李昌兵编写了第 9 章，武建军编写了第 10 章。全书由杜茂康审校和统稿。

本书在编写过程中得到了不少专家、学者、老师和同事的指导、支持和帮助，2004 级信息管理 with 信息系统专业两位热爱程序设计的学生李明闯和王晓润仔细阅读了本书第 1 版初稿中的全部内容，校正了初稿中的许多错误，广大读者也指正了本书前 3 版的错误和不当之处，并提出了许多有用的建议。在此谨向他们表示诚挚的感谢！

本书在编写过程中参考了国内外大量相关文献，大部分已被列入书后的参考文献，在此谨向这些文献的作者表示衷心感谢！

面向对象程序设计是一项不断发展创新的程序技术，C++ 语言更是博大精深，其标准和规范基本上每三年就更新一次，发展变化快，新技术层出不穷。鉴于作者水平有限，加之经验不足，书中一定存在不少错误和不当之处，恳请专家、同行和读者批评指正。

为了便于读者学习和教师教学，本书配有以下教学资源：全部例题的程序代码、部分习题的程序代码、配套的电子课件。有需要者可从华信教育资源网 (<http://www.hxedu.com.cn>) 上进行下载，或者扫描封底的二维码获取。

作 者

目 录

第 1 章 C++与面向对象程序设计	1
1.1 面向对象程序设计概述	2
1.1.1 面向过程程序设计	2
1.1.2 面向对象程序设计	3
1.1.3 面向对象程序设计语言的特征	4
1.2 C++语言概述	6
1.2.1 C++语言简史	7
1.2.2 C++的特点	8
1.2.3 C++程序的结构	8
1.2.4 标准 C++程序设计	11
1.3 数据的输入和输出	13
1.3.1 数据类型	13
1.3.2 流的概念	14
1.3.3 cin 和提取运算符>>	15
1.3.4 cout 和插入运算符<<	17
1.3.5 输出格式控制符	19
1.3.6 数制基数	21
1.3.7 string 和字符串的输入、输出	22
1.3.8 数据输入的典型问题	24
1.4 编程实作：Visual C++ 2022 编程简介	28
习题 1	32
第 2 章 C++程序设计基础	34
2.1 C++语言对 C 语言的类型扩展和类型定义	35
2.2 C++程序变量设计的基本思想	36
2.3 左值、右值和断言	38
2.4 指针	39
2.4.1 指针概述	39
2.4.2 void*指针和获取数组首、尾元素位置的指针	41
2.4.3 内存的分配和释放	42
2.4.4 智能指针	44
2.5 引用	49
2.5.1 左值引用	49
2.5.2 右值引用、移动及其语义	52
2.6 const 和 constexpr 常量	54
2.6.1 常量的定义	54
2.6.2 const、constexpr 与指针	55

2.6.3	const 与引用	56
2.6.4	顶层 const 和底层 const	57
2.7	auto、decltype 和 decltype(auto)类型	58
2.8	C++新式 for 循环和数组	60
2.8.1	begin、end 和基于范围的 for 循环	60
2.8.2	vector 和 valarray	61
2.9	数据类型转换	63
2.10	函数	66
2.10.1	函数原型	66
2.10.2	函数参数传递的方式	67
2.10.3	函数默认参数	71
2.10.4	函数返回值	72
2.10.5	函数重载	75
2.10.6	函数与 const 和 constexpr	78
2.10.7	内联函数	81
2.11	匿名函数	82
2.12	命名空间	88
2.13	变量	90
2.13.1	变量定义	90
2.13.2	作用域	90
2.13.3	变量的类型和生命期	92
2.13.4	变量初始化	93
2.13.5	局部变量与函数返回地址	97
2.14	预处理器	97
2.15	文件的输入和输出	100
2.15.1	文件操作的基本流程	100
2.15.2	输入流、输出流的泛化思想	101
2.16	编程实作：C++程序设计初步	102
习题 2	104
第 3 章	类和对象	110
3.1	类的抽象和封装	111
3.1.1	抽象	111
3.1.2	封装	113
3.2	结构	115
3.2.1	C++对结构的扩展	116
3.2.2	类	118
3.3	数据成员	119
3.4	成员函数	120
3.4.1	成员函数定义方式和内联函数	120
3.4.2	常量成员函数	122
3.4.3	成员函数重载和默认参数值	123

3.5	对象	123
3.6	构造函数设计	126
3.6.1	编译器添加的默认成员函数	127
3.6.2	构造函数和类内初始值	128
3.6.3	默认构造函数	130
3.6.4	重载构造函数	133
3.6.5	构造函数与初始化列表	135
3.6.6	委托构造函数	137
3.7	析构函数	138
3.7.1	析构函数的设计思想和定义	138
3.7.2	弱指针与析构函数	140
3.8	赋值运算符函数、复制构造函数和移动函数设计	142
3.8.1	赋值运算符函数	142
3.8.2	复制构造函数	146
3.8.3	移动函数	149
3.9	静态成员	154
3.10	this 指针	157
3.11	对象应用	161
3.11.1	成员访问操作符	161
3.11.2	对象数组与对象指针	164
3.11.3	向函数传递对象	165
3.11.4	对象成员	166
3.12	类的作用域和对象的生命期	169
3.13	友元	172
3.14	编程实作：类的接口与实现的分离	173
3.14.1	头文件	174
3.14.2	源文件	175
3.14.3	对类的应用	176
习题 3	180
第 4 章	继承	185
4.1	继承的概念	186
4.2	protected 与继承	187
4.3	继承方式	188
4.4	派生类对基类的扩展	191
4.4.1	成员函数的重定义和名字隐藏	191
4.4.2	基类成员访问	193
4.4.3	using 和隐藏函数重现	194
4.4.4	派生类修改基类成员的访问权限	195
4.4.5	友元与继承	196
4.4.6	静态成员与继承	197
4.4.7	继承和类作用域	198

4.5	构造函数和析构函数	199
4.5.1	派生类构造函数的建立规则	200
4.5.2	派生类构造函数和析构函数的调用次序	205
4.5.3	派生类的赋值、复制和移动操作	207
4.6	基类与派生类对象的关系	208
4.6.1	派生类对象对基类对象的赋值和初始化	209
4.6.2	派生类对象与基类对象的类型转换	211
4.7	多继承	213
4.7.1	多继承的概念和应用	213
4.7.2	多继承方式下的成员二义性	215
4.7.3	多继承的构造函数和析构函数	216
4.8	虚拟继承	217
4.9	继承和组合	222
4.10	编程实作：继承编程应用	226
习题 4	231
第 5 章	多态	237
5.1	多态概述	238
5.1.1	多态的概念	238
5.1.2	多态的意义	240
5.1.3	多态和绑定	241
5.2	虚函数	241
5.2.1	虚函数的意义	241
5.2.2	override 和 final	244
5.2.3	虚函数的特性	246
5.3	虚析构函数	251
5.4	纯虚函数和抽象类	252
5.4.1	纯虚函数和抽象类	252
5.4.2	抽象类的应用	254
5.5	运行时类型信息	262
5.5.1	dynamic_cast	263
5.5.2	typeid	266
5.6	编程实作：多态编程应用	268
习题 5	269
第 6 章	运算符重载	274
6.1	运算符重载基础	275
6.2	重载二元运算符	277
6.2.1	类与二元运算符重载	277
6.2.2	非类成员方式重载二元运算符的特殊用途	280
6.3	重载一元运算符	282
6.3.1	作为成员函数重载	282
6.3.2	作为友元函数重载	284

6.4	特殊运算符重载	285
6.4.1	重载++和--	285
6.4.2	下标[]和赋值运算符=	287
6.4.3	类型转换运算符	289
6.4.4	仿函数	292
6.5	输入/输出运算符重载	293
6.6	编程实作：运算符重载编程应用	295
习题 6	300
第 7 章	模板和 STL	303
7.1	模板的概念	304
7.2	函数模板和模板函数	305
7.2.1	函数模板的定义	305
7.2.2	函数模板的实例化	306
7.2.3	模板参数	307
7.3	类模板	310
7.3.1	类模板的概念	310
7.3.2	类模板的定义	311
7.3.3	类模板实例化	313
7.3.4	类模板的应用	315
7.4	模板设计中的独特问题	316
7.4.1	模板参数类型推导	316
7.4.2	内联与常量函数模板	320
7.4.3	默认模板实参	320
7.4.4	仿函数应用	321
7.4.5	成员模板	322
7.4.6	可变参数函数模板	323
7.4.7	元编程的基本概念	324
7.4.8	模板重载、特化、非模板函数及调用次序	327
7.5	STL 程序设计	330
7.5.1	函数对象	330
7.5.2	顺序容器	331
7.5.3	迭代器	339
7.5.4	pair 和 tuple 容器	342
7.5.5	关联式容器	345
7.5.6	算法	352
7.5.7	STL 容器和算法处理自定义类的常见问题	355
7.6	编程实作：模板和 STL 编程应用	357
习题 7	358
第 8 章	异常	362
8.1	异常处理概述	363
8.2	C++异常处理基础	364

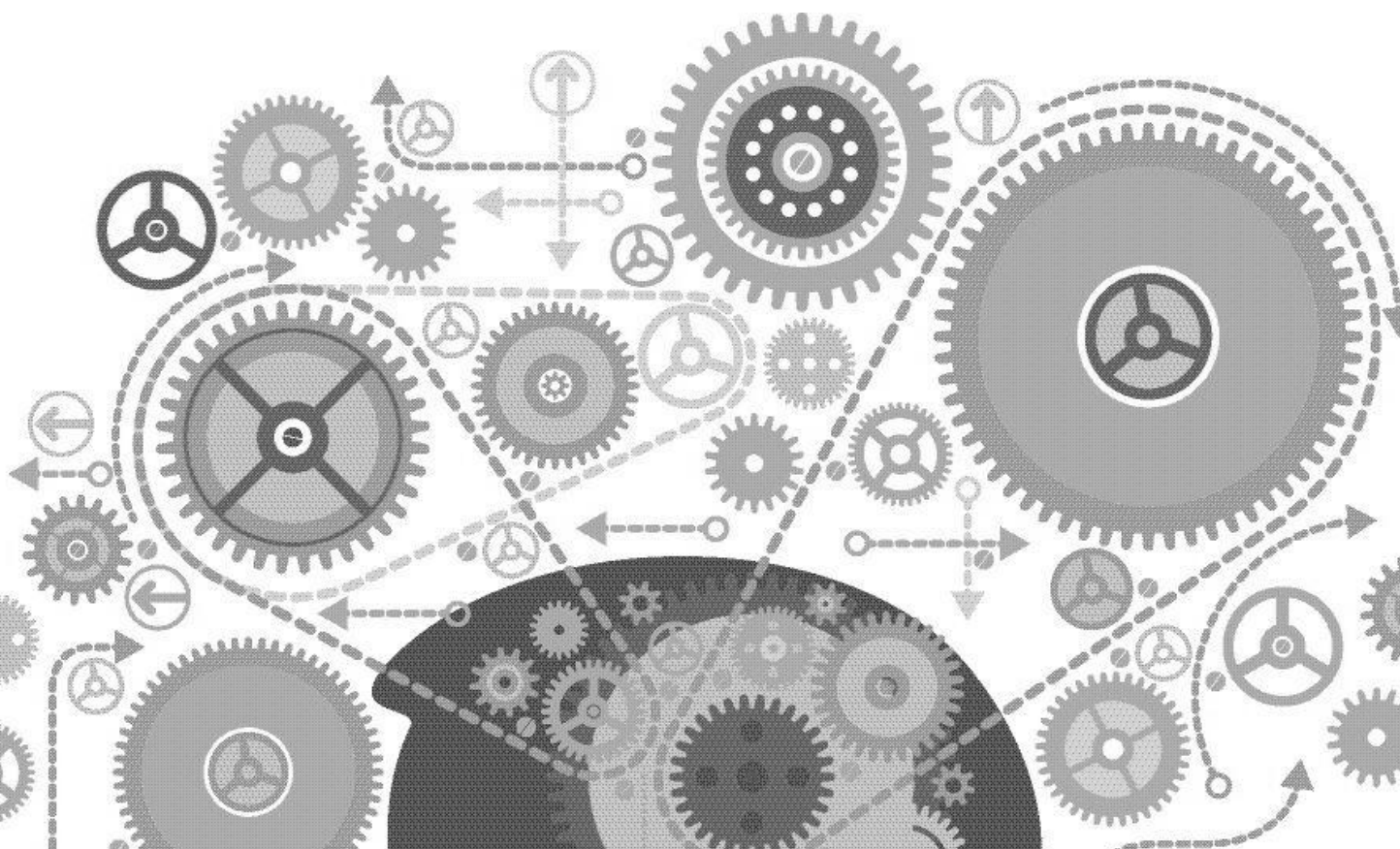
8.2.1	异常处理的结构	364
8.2.2	异常捕获	365
8.3	异常和函数	367
8.4	异常处理的特殊情况	368
8.5	异常和类	373
8.5.1	构造函数和异常	373
8.5.2	异常类	375
8.5.3	派生异常类的处理	378
习题 8	381
第 9 章	线程	383
9.1	程序、进程和线程	384
9.2	线程等待和线程 ID 获取	388
9.2.1	线程等待	388
9.2.2	获取线程 ID	390
9.3	类和线程	391
9.4	线程同步	393
9.4.1	互斥锁	393
9.4.2	读写锁	396
9.4.3	信号量	397
9.4.4	条件变量	402
习题 9	405
第 10 章	流和文件	410
10.1	C++ I/O 流及流类库	411
10.2	I/O 流类的成员函数	412
10.2.1	类 istream 的常用成员函数	412
10.2.2	类 ostream 的常用成员函数	414
10.2.3	数据输入、输出的格式控制	415
10.3	文件操作	418
10.3.1	文件和流	418
10.3.2	二进制文件	420
10.3.3	随机文件	423
习题 10	425
参考文献	428

第 1 章

C++ 与面向对象程序设计

面向对象程序设计是用对象模拟客观世界中的事物及其行为，用消息传递模拟对象之间的相互作用，使程序与实际问题的相似性，从而降低软件开发的难度，提高软件开发的效率，适合大型的、复杂的应用程序开发。

本章介绍面向对象程序设计语言的特征，以及 C++ 语言的数据类型、程序结构、数据输入和输出等内容。



1.1 面向对象程序设计概述

早期计算机程序的规模较小，主要开发方式为个人设计、个人使用，没有统一的开发原则，只需将相应的程序代码组织在一起，再让计算机执行它，就可以完成相应的程序功能。随着计算机的普及和技术发展，程序的规模和复杂度越来越大，到了 20 世纪 60 年代初期，个人软件开发方式已不能满足需求，面临许多困境，如软件开发费用超出预算，不能按期完成软件开发，质量达不到要求，软件维护困难，如此等等。这就是所谓的“软件危机”，因此迫切需要改变软件的生产方式，提高软件开发效率。

1.1.1 面向过程程序设计

20 世纪 60 年代末出现了影响深远的结构化程序设计 (Structure Programming, SP) 思想。结构化程序设计采用“自顶向下、逐步求精、模块化”的方法进行程序设计，即采用功能抽象、模块分解、自顶向下、分而治之的方法，将一个复杂的、庞大的软件分解成为许多易于控制、处理、可独立编程的模块。各模块可由结构化程序设计语言的子程序（函数）实现，子程序则由顺序、分支、循环三种基本结构组成。其基本特点是：① 按层次组织模块；② 每个模块只有一个入口和一个出口；③ 代码与数据分离，即“程序 = 数据结构 + 算法”。

结构化程序设计是一种以功能为中心的面向过程程序设计方法，先将要解决的问题分解成若干模块，再根据模块功能设计一系列用于存储数据的数据结构，并编写一些函数（或过程）对这些数据进行操作，最终的程序是由许多函数（或过程）组成的。

在结构化程序设计中，数据与过程分离。数据代表问题空间的客体，表达实际问题中的信息。代码则是用于体现和加工处理这些数据的算法。在设计软件时，必须时时考虑要处理的数据的结构和类型，若要对不同格式的数据做相同的处理，或者对相同格式的数据做不同的处理，都必须编写不同的代码，代码的重用性较差。

此外，数据与过程的分离还会导致代码的可维护性比较差，因为当数据结构改变时，所有与之相关的处理过程都要进行修改，增大了代码维护的难度。

例如，实现一个通讯录管理程序，程序员编写了 4 个函数：inputData(), printData(), searchPhone(), searchAddr(), 分别实现通讯录数据的输入、输出和查询功能。许多技术都可以实现通讯录的数据存取，如数组、链表、队列等。采用数组存取数据的程序结构大致如下：

```
struct Person{                                // 用于存放个人信息的数据结构
    char name[10];
    char addr[20];
    char phone[11];
}
Person p[100];                                // 保存所有个人信息的全局数组
int n = 0;                                    // 用于保存实际人数的全局变量
void inputData(){...}                         // 初始化全局数组 P，读入每个人的姓名、地址和电话
void searchAddr(char *name){...}              // 根据姓名查找地址
void searchPhone(char *name){...}             // 根据姓名查找电话号码
void printData(){...}                         // 打印输出每个人的姓名、地址和电话
```

这 4 个函数通过全局数组 `p[]` 共享数据，并且相互影响。如果将这些函数提供给其他程序员使用，就必须让该程序员知道他不能定义和修改全局数组 `p[]`，只能通过这 4 个函数存取全局数据。

结构化程序设计代表了面向过程程序设计的基本编程方法：先定义一些全局性的数据结构，再编写一些函数对这些数据结构进行操作，如图 1-1 所示。

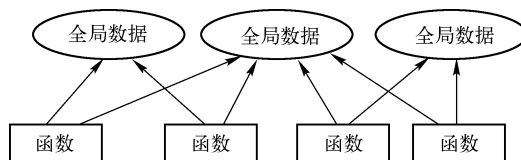


图 1-1 面向过程程序设计的模型

由图 1-1 可知，数据与函数之间存在潜在的连接关系。某全局数据的修改可能引起大量操作该全局数据的函数的修改。此外，若某函数意外修改了某全局数据，很可能引起程序数据的混乱。例如，在个人通讯录管理程序中，**Person** 的变化会引起操作它的所有函数（如 `inputData()`、`searchAddr()` 等）的修改。此外，谁也没有办法限制其他程序员定义与全局数据同名的变量，也不能限制他修改全局数组的值。当程序规模较大时，这个问题尤其突出，代码维护困难。

支持结构化程序设计的高级语言称为结构化程序设计语言，提供了顺序、分支和循环三种基本结构，支持面向过程的程序设计，是面向过程的程序设计语言。**FORTRAN**、**C** 等都是广泛使用的面向过程的程序设计语言。

1.1.2 面向对象程序设计

随着计算机和网络技术的发展，软件应用的领域不断扩大，需求越来越大。同时，软件的规模和复杂度也在不断增加，升级改版的时间要求却在缩短，面向过程程序设计技术已不能满足软件开发在效率、代码共享和更新维护等方面的需求了，取而代之的是面向对象程序设计技术。

面向对象是指以对象为中心进行分析、设计和构造应用程序的机制。其基本观点是：计算机求解的都是现实世界中的问题，它们由一些相互联系且处于不断运动变化的事物（对象）组成，如果能够用对象描述问题域中的各客观事物，用对象之间的关系描述客观事物之间的联系，用对象之间的作用描述事物之间的交流和驱动，就能将客观世界中的问题直接映射到计算机中，实现对现实问题的真实模拟。

这里涉及三方面的问题：一是如何把客观事物表示为计算机中的对象；二是如何用对象之间的关系反映客观事物之间的联系；三是如何用对象之间的作用反映客观事物之间的交流和驱动。

对于第一个问题，面向对象技术的解决方法是：对于任何一个客观事物，用数据表示它的特征，用函数描述它的行为，并把两者结合成一个整体，称为对象，代表一个客观事物。由此可知，一个对象由数据和函数两部分构成。数据常被称为数据成员，函数则被称为成员函数。一个对象的数据成员通常只能通过自身的成员函数修改。

对象真实地描述了客观事物，将数据和操作数据的过程（函数）绑在一起，形成一个相

互依存、不可分离的整体（对象）。从同类对象中抽象出共性，形成类。同类对象中的数据原则上只能用本类提供的方法（成员函数）进行处理。

对于第二个问题，面向对象技术提供了继承、对象成员、对象依赖等机制来描述客观事物之间诸如父子关系、汽车及其组成部件的包含关系、某人与他的宠物狗之间的依赖关系等。

对于第三个问题，对象之间的消息传递机制用于表示客观事物之间的关系。禁止一个对象以任何未经允许的方式修改另一个对象的数据，如果它需要向另一个对象传递数据，或者得到它的服务，可以向该对象发送消息，对方会响应消息，执行特定函数来完成消息发送者的操作要求。

面向对象程序技术能够实现对客观事物的自然描述，反映客观世界的本来面目，使程序模块化设计更简单、自然，如图 1-2 所示。

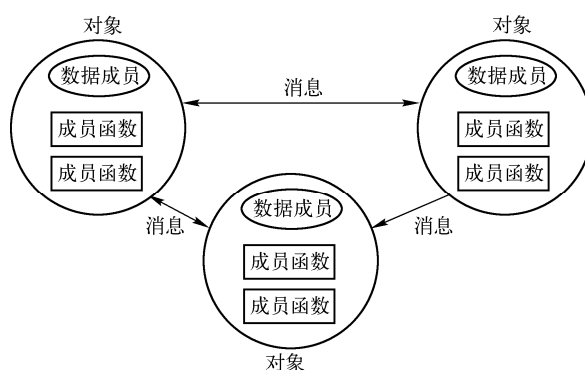


图 1-2 面向对象程序设计的模型

面向对象程序技术提高了软件的可靠性、可重用性、可扩展性和可维护性。因为某类对象数据的改变只会引起该类对象程序代码的改变，而与其他类型的对象无关，这就把程序代码的修改维护局限在了一个很小的范围内。由于数据和操作它的函数是一个整体，因此易被重用。在扩展某对象的功能时，不用考虑它对其他对象的影响，软件功能的扩展更容易。

1.1.3 面向对象程序设计语言的特征

面向对象程序设计语言经历了一个较长的演变过程，20 世纪 50 年代的 LISP 就引入了信息隐藏和封装机制，60 年代的 Simula 语言提出了抽象和封装，引入了数据抽象和类的概念，被认为是第一个面向对象语言（但不具备面向对象语言的全部特征）；70 年代的 Smalltalk 则是第一个真正面向对象的程序设计语言。现在广泛使用的 C++、C#、Object-C、Java、Python、Go 等都是面向对象的程序设计语言。

面向对象程序设计语言具有以下特征。

1. 抽象 (Abstract)

抽象是指有意忽略问题的某些细节和与当前求解问题无关的方面，抽取事物的主要特征，并用来描绘客观事物。抽象的结果是形成对应客观事物的抽象数据类型，简称 ADT (Abstract Data Type)。

在现实生活中，人们常从特征和行为两方面描绘客观对象。抽象也是如此，它从现实中的客观事物出发，分析同类事物应当具备的共同特征和行为，将这些特征抽取出来并形成描绘

该类事物的属性，将行为抽象为描绘事物的函数（也称为方法）。抽象是一个从个性到共性的过程。例如，要形成学生的抽象概念，就要观察现实中的各位同学，忽略各自的独特爱好和特长，抽取他们作为学生的共有特征和行为，就形成了学生的抽象数据类型，如图 1-3 所示。

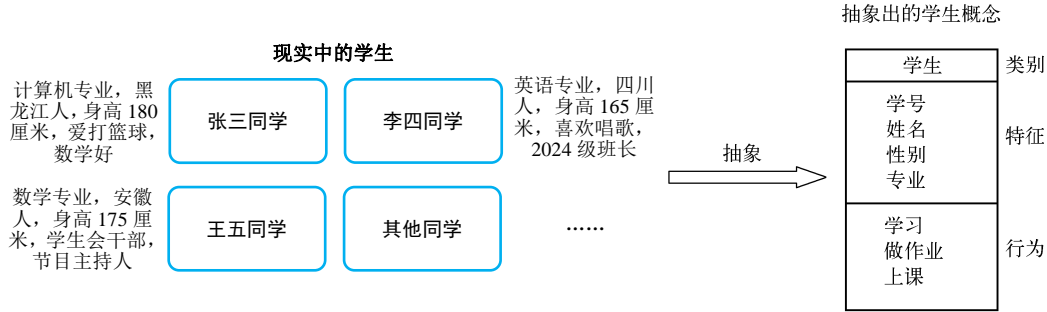


图 1-3 学生概念的抽象过程

2. 封装 (Encapsulation)

抽象只是给出了对形成的抽象数据类型的功能描述，设计出了该类型应提供的各项功能，以及使用这些功能的方法，相当于为用户提供了使用该数据类型功能的接口，但没有实现任何具体的特征和功能。实现这些特征和功能的过程被称为封装。

好比建大楼分为图纸设计和按图建楼两个阶段。抽象相当于图纸设计阶段，设计图确定了大楼的基本结构和功能，人们可以据此掌握大楼的基本情况，规划它的应用。但只有按图建楼后，才能够住进大楼。这个建楼的过程相当于封装。因此，人们可以通过抽象形成的数据类型了解其基本情况，规划它的用途，但在其封装后才能够在程序中真正应用。

鉴于上述原因，人们就说抽象导致了接口与实现的分离。抽象只是完成了 ADT 接口的设计，而具体实现由封装完成。封装就是包装并实现抽象出的数据类型，使之成为可用于程序设计的抽象数据类型的过程。

封装是面向对象程序技术的重要特征，将抽象出的特征（用数据表示）和行为（用函数表示）捆绑成一个整体，并且编码实现抽象所设计接口的功能。封装后的 ADT 由接口和实现两部分组成。接口在外，描述了抽象类型显示给用户的外部视图，而抽象数据类型的结构和接口功能的实现细节的程序被封装隐藏，用户对此一无所知，也不需知道，因为这并不影响对该功能的使用。反之，封装后的抽象数据类型更易于使用，因为用户不会被复杂的内部结构和实现细节所干扰，只需向接口传递正确的参数，就能够使用所需的功能。

抽象和封装是认知客观事物并把它表示成可用于程序设计的抽象数据类型的两个阶段，抽象完成抽象数据类型的整体设计，封装则实现设计所需的功能。比如要做手电筒，抽象阶段的任务是了解现实，设计出手电筒的大小、形状、颜色、灯泡的规格，以及便于用户更换灯泡、电池和开关的三个接口，并描述好接口的外形和功能。封装的任务就是按照抽象的设计结果，通过具体的电路和电子开关实现各接口的功能，并把具体的实现细节包装进手电筒内部，不让别人知道，只留操控开关（接口）在外，如图 1-4 所示。人们只能够通过允许的接口使用手电筒，电不足了或灯泡坏了，只能通过指定的接口更换，要用光的时候打开开关，不用的时候断开。而封装在手电筒内部的电路和连接各部件的电子线路不让人们知道和操控。

面向对象程序用类（class）实现封装，封装后的抽象数据类型称为类类型。面向对象程序设计的主要任务是对问题空间的各类客观事物进行抽象，构造出代表问题空间各类事物的类。

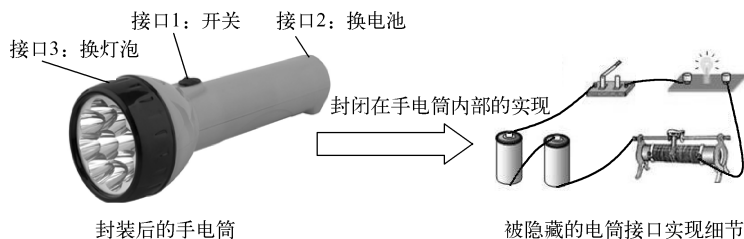


图 1-4 手电筒封装

3. 继承 (Inherit)

继承源于生物界。通过继承，后代能够获得与其祖先相同或相似的特征和能力。面向对象程序设计语言也提供了类似生物继承的机制，允许一个新类由现有类派生，能够继承现有类的属性和行为，并且修改或增加新的属性和行为，成为一个功能更强大、满足更多应用需求的类。

继承是面向对象程序设计语言的一个重要特征，是实现软件复用的一个重要手段。在面向对象程序设计中，如果一个类 **B** 继承了另一个类 **A**，就称类 **B** 为**子类** (subclass)，称类 **A** 为**超类** (superclass)。

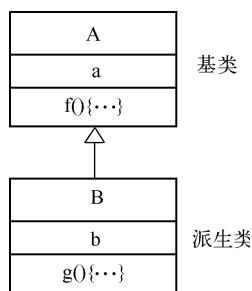


图 1-5 派生类 B 继承

C++的发明者 **Stroustrup** 认为，超类和子类这两个概念容易让人产生误解，他提出了**基类**和**派生类**这两个术语，分别用来表示超类和子类的概念。本书将引用 **Stroustrup** 的术语来表示继承关系。图 1-5 是表示两个类继承关系的一个简图，表示类 **A** 是类 **B** 的基类 (超类，也称为父类)，类 **B** 是类 **A** 的派生类 (子类)。

派生类 **B** 继承了基类 **A** 的所有特征和行为，尽管类 **B** 只定义了数据成员 **b** 和成员函数 **g()**。但它实际上具有 **a**、**b** 两个数据成员，具有两个成员函数 **f()** 和 **g()**，其中的 **a** 和 **f()** 是从基类 **A** 继承得到的。

继承分为单继承和多重继承。单继承规定每个子类只能有一个父类，图 1-5 就是一个单继承。多重继承允许每个子类有多个父类。继承为软件设计提供了一种功能强大的扩展机制，允许程序员基于已经设计好的基类创建派生类，并且可以为派生类添加基类不具有的属性和行为，极大地提高了软件复用的效率。

4. 多态 (Polymorphism)

多态是面向对象程序设计语言的另一重要特征，它的意思是“一个接口，多种形态”。也就是说，不同对象针对同一种操作会表现出不同的行为。多态与继承密切相关，通过继承产生不同的类，而这些类分别对某成员函数进行了定义，当这些类的对象调用该成员函数时会做出不同的响应，执行不同的操作，实现不同的功能。

1.2 C++语言概述

C++语言是从 C 语言发展演变而来的，在 C 语言的基础上引入了类 (class) 的概念，并增加了封装、继承、多态等面向对象的语言处理机制。C++语言向前兼容 C 语言程序设计，使得绝大部分 C 程序可以不加修改就能在 C++环境下编译运行，同时提供了面向对象的程序

设计机制，支持面向对象程序设计，是一种面向过程与面向对象的混合式编程语言。

1.2.1 C++语言简史

在计算机发展的早期，操作系统之类的系统软件主要是用汇编语言编写的。由于汇编语言依赖于计算机硬件系统，用它编写的软件系统的可移植性和可读性都比较差。

UNIX 系统最初也是用汇编语言编写的。为了提高 UNIX 系统的可移植性和可读性，1970 年，美国 AT&T 贝尔实验室的 Ken Thompson 以 BCPL(Basic Combined Programming Language) 为基础，设计了非常简洁且与硬件结合紧密的 B 语言，且用该语言改写了 UNIX，并在 PDP-7 上实现了它。

B 语言是一种无类型语言，直接对机器字进行操作，过于简单，且功能不强。1972 年到 1973 年间，贝尔实验室的 Dennis Ritchie 对 B 语言进行了改造，添加了数据类型的概念，设计出了 C 语言，并在 1973 年与 Thompson 用 C 语言重写了 UNIX 的 90%以上的代码，这就是 UNIX 5。在此之后，C 语言又进行了多次改进，1975 年 UNIX 6 发布后，C 语言突出的优点引起了世人的普遍关注。1977 年，不依赖于具体机器指令的、可移植的 C 语言出现了。

C 语言简洁、灵活，具有丰富的数据类型和运算符，具有结构化的程序控制语句，支持程序直接访问计算机的物理地址，具有高级语言和汇编语言的双重特点。1978 年以后，C 语言先后被移植到大、中、小及微型计算机上。随着 UNIX 系统在各种类型的计算机上的实现和普及，C 语言逐渐成了最受欢迎的程序设计语言之一。

但是 C 语言本身也存在一些缺陷：类型检查机制较弱，缺乏支持代码重用的语言结构，当程序规模大到一定程度时，就很难控制程序的复杂性了，不适合大型软件系统的开发设计。

1979 年，贝尔实验室的 Bjarne Stroustrup 借鉴了 Simula 中类的概念，对 C 语言进行了扩展和创新，将 Simula 的数据抽象和面向对象等思想引入 C 语言，称为“带类的 C”(C with class)，这就是 C++ 的早期版本。1983 年，“带类的 C” 正式改名为 C++。此后，C++ 语言逐渐发展成了主流编程语言，其标准经历了若干次修订，每次修订都增加和修改了一些内容，以适应计算机及程序技术发展的需要，如图 1-6 所示。

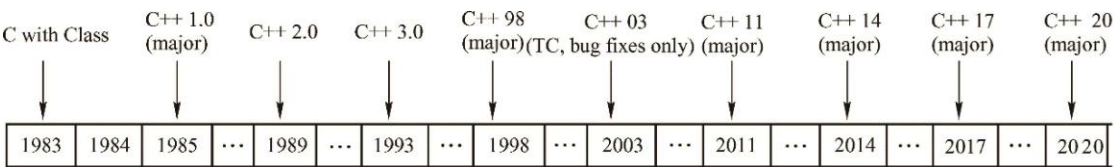


图 1-6 C++标准的发展

到目前，C++ 已完成了六次变化较大的标准修订。

第一次修订发生在 1985 年，其主要变化是引入虚函数、函数和运算符的重载、引用、常量等概念。

第二次修订是 1998 年。为了实现 C++ 的标准化，成立了由 ANSI (American National Standards Institute, 美国国家标准化协会) 和 ISO (International Standards Organization, 国际标准化组织) 参加的联合标准化委员会。每 5 年视实际需要更新一次标准。委员会于 1998 年提出了 C++ 的 ANAI/ISO 标准，引入了命名空间的概念，增加了标准模板库(Standard Template Library, STL) 中的标准容器类、通用算法类和字符串类型等内容，使得 C++ 语言更为实用。

此版本后的 C++是具有国际标准的编程语言，通常简称 ANSI C++或 ISO C++ 98 标准，比 Stroustrup 最初定义的 C++要大得多，也复杂得多，人们称此为标准 C++。为了与标准 C++相区别，将之前的版本称为传统 C++。

第四次修订是 2011 年，称为 C++ 11，该标准包含了核心语言的新功能，并且拓展了 C++ 标准程序库，加入了大部分 C++ TR1（Technical Report 1，C++ Library Extensions（函式库扩充）的一般名称），包括正则表达式、智能指针、哈希表、随机数生成器等程序库。

第五次较大修订在 2017 年完成，对 C++核心库、并发技术、并行技术、网络规范、大规模软件系统支持等方面进行修订或扩展。

第六次修订在 2020 年完成，引用了概念、协程、范围、视图等新功能，并对标准库的功能进行了扩充。

1.2.2 C++语言的特点

C++语言保留了 C 语言的原有特征和优点，支持 C 语言程序设计，同时对 C 语言进行了扩展，增加了面向对象的新特征和语言处理机制，支持面向对象的程序设计，是 C 语言的超集。概括而言，C++语言具有以下特点。

① 高效性。C++语言允许直接访问物理地址，支持直接对硬件编程和位（bit）操作，能够实现汇编语言的大部分功能，生成的目标代码质量高，程序运行效率高。C++语言虽然是一种高级语言，但是具有机器语言的许多功能，适用于编写系统软件。

② 灵活性。C++程序中几乎可以不受限制地使用各种程序设计技术，开发出各种特殊类型的程序，能够灵活适应诸多不同领域的程序开发。

③ 丰富的运算符和数据类型。C++语言不仅提供了 int、char、bool、double、float 等内置数据类型，还允许通过结构、类、枚举定义用户数据类型，并通过对+、-、\、*、%、||、&、<<、>>、>、<、>>等运算符的重载增加对自定义数据类型的运算功能，支持算术运算、逻辑运算、位操作等运算。

④ 可移植性。C++语言具有平台无关性，开发出的程序能够比较容易地从一种类型的计算机系统中移植到另一种类型的计算机系统中。

⑤ 支持面向对象程序设计。C++语言对 C 语言的最大改进就是融入了面向对象程序设计的思想，提供了把数据和数据操作封装在一起的抽象机制，支持类、继承、重载和多态等面向对象的程序设计，使 C++程序在软件复用和大型软件的构造和维护等方面变得容易、高效，提高了软件开发的效率和质量。

总之，C++语言保留了 C 语言简洁、高效和接近汇编语言等特点，对 C 语言的类型系统进行了改进和扩展，比 C 语言更安全、可靠。但 C++语言最重要、最有意义的特征是支持面向对象的程序设计。

1.2.3 C++程序的结构

C++兼容 C 语言程序设计，二者的程序结构相同，常由以下 3 部分构成。

1. 声明部分

声明部分包括头文件包含、全局变量或全局常量的声明、函数声明等内容。

C++编译系统（或其他软件提供者）提供了许多具有不同功能的函数，这些函数常被分为声明（函数头，包括函数返回类型、函数名、形参表）和实现（实现函数功能的程序代码）两部分。函数声明常以源代码的方式被集中放置在头文件中，函数实现则被编译成二进制代码，存放在各种库文件中。在 C++程序中，用“`#include 头文件名`”的形式将其引入程序，并按照头文件中的声明向函数提供参数，就能够引用该函数的功能。这就是在 C++程序的声明部分包含头文件的原因。

为了提高程序的可读性，常将函数定义放在主函数后面。在 C++程序中，如果函数的调用先于其定义，就必须在声明部分对该函数进行声明，告诉 C++编译系统此函数的定义在后面，这样它才能被调用。后续将会发现，声明部分常常还包括类的声明。

此外，声明部分通常还被用来定义本程序要用到的全局变量和符号常量。

2. 主函数部分

同 C 程序一样，C++程序的主函数也是 `main()`，它是程序执行的起点和主体。C++程序从函数 `main()` 的第一条语句开始，顺序执行其中的程序代码，执行完函数 `main()` 的全部语句后，程序就结束了。一段代码若想被执行，只有被函数 `main()` 直接或间接调用才行。

3. 函数定义部分

函数定义部分用来定义函数的功能，所有在前面只做了声明的函数都必须在此进行定义，即编写相关函数的程序代码。

现在来看一个简单的 C++程序，借此了解 C++程序的基本结构。

【例 1-1】 从键盘输入 10 个整数，并按从大到小顺序输出。

说明：程序代码前面的行号是为了分析问题而添加的。

```
0 // Eg1-1.cpp
1     #include <iostream>
2     #define      N      10
3     void sort(int a[], int n);
4     void print(int a[], int);
5     using namespace std;
6     void main() {
7         int a[N];
8         cout<<"input 10 numbers:\n";
9         for(int i = 0; i < N; i++)
10             cin>>a[i];
11         sort(a, N);
12         print(a, N);
13     }
14
15     void sort(int a[], int n) {
16         for(int i = 0; i < n-1; i++) {
17             for(int j = i+1; j < n; j++) {
18                 if(a[i] < a[j]) {
19                     int t = a[i];
20                     a[i] = a[j];
21                     a[j] = t;
                }
            }
        }
    }
```

声明部分

主函数部分

函数定义部分

```

22         }
23     }
24 }
25 }
26 void print(int a[], int n) {
27     for(int i = 0; i<n; i++) {
28         cout<<a[i]<<" ";
29     }
30     cout<<endl;
31 }

```

} 函数定义部分

① 源文件类型名。C 语言程序文件的类型名是 .c，C++程序文件的类型名是 .cpp。

② 注释语句和语句结束符。C++语言支持 C 语言的块注释语句，即写在/* 和 */之间的语句块被视为注释。此外，C++语言增加了一个行注释符“//”，可以出现在一个语句行的任何位置，其有效范围是从它开始到该行结束。

同 C 程序一样，C++程序中也用“;”表示一条语句的结束。

③ 数据的输入和输出。C++语言常用 cin 输入数据，用 cout 输出数据，它们是在 iostream 中定义的。

第 1 行是一条预编译命令，其作用是将头文件 iostream 的内容包含（添加）到本程序中。当调用 cin 和 cout 命令时，C++语言就知道在 iostream 中寻找它们的函数定义了。

第 5 行是命名空间引用语句，其中的 std 称为标准命名空间，程序中用到的 cout 和 cin 函数就是在此命名空间中定义的函数，关于 std 的介绍详见 1.2.4 节。

第 8 行的 cout 表示输出。语句“cout<<"input 10 numbers: \n";”用来把“<<”后面的“input 10 numbers: \n”输出到显示器屏幕上，提示用户输入 10 个数字。该字符串最后的“\n”与 C 语言中的含义一样，是个转义符，表示换行。最后的分号是语句结束符。

第 10 行中的 cin 用于接收从键盘输入的数据。语句“cin>>a[i];”用来把键盘输入的数据存入数组元素 a[i]。第 9~10 行构成的 for 循环用于从键盘输入 10 个整数到数组 a 中。

第 27~28 行构成一个 for 循环，用于连续输出 a 数组的 10 个元素，各数组元素之间用空白间隔。

第 29 行“cout<<endl;”语句中的“endl”相当于“\n”，用于在屏幕上输出回车符。

④ 函数声明和函数定义。第 3 行是函数 sort()的前向引用声明。本程序在函数 main()中调用 sort()时（第 11 行）还没有定义 sort()，所以在第 3 行中进行了声明，void 表示该函数不返回任何值。同理理解第 4 行。第 15~25 行是冒泡法排序函数 sort()的定义，第 26~30 行是函数 print()的定义。

⑤ 主函数。第 6~13 行是函数 main()。同 C 程序一样，每个 C++程序必须有一个名为 main()的主函数，它是程序执行的起点。main()后的一对“{ }”中的所有程序代码构成了 main()的函数体。

运行该程序，当屏幕上显示“input 10 numbers: ”时，从键盘输入 10 个整数，输入一个数据后按空格键（也可按 Enter 键），10 个数据输入完成后，程序会把它们按从大到小顺序输出。运行结果如下：

```

input 10 numbers:
21  3  4  5  12  3  5  4  78  9
78  21  12  9  5  5  4  4  3  3

```

说明：上述第 2 行是从键盘输入的数据，输完按 **Enter** 键，第 3 行是输出的结果。

1.2.4 标准 C++程序设计

如 1.2.1 节所述，标准 C++增加了传统 C++中没有的一些特征，并对原来的库函数进行了修订，是传统 C++的超集。两种版本的 C++有大量相同的库和函数（标准 C++更多），如两种版本中都有 `scanf()`、`printf()`、`cin()`、`cout()`等函数，它们的用法完全相同。另一方面，许多 C++编译器（如 **Visual C++**、**C++ Builder**）同时提供了对标准 C++及传统 C++的支持，而且允许在程序中同时调用两种标准的库函数。为了区分程序所调用的库函数来源，C++采用了以下解决方案。

1. 头文件区别

传统 C++保留之前与 C 语言同样风格的头文件和库函数调用方式，标准 C++则采用没有 .h 扩展名的新式头文件，若是 C 函数库的头文件，则将“c”放在文件名前面，如：传统 C++的头文件为 `iostream.h`、`fstream.h`、`string.h`、`stdio.h`、`ctype.h`、`math.h`，标准 C++对应的头文件为 `iostream`、`fstream`、`string`、`cstdio`、`cctype`、`cmath`。

2. 命名空间限定

传统 C++的库函数调用同 C 语言的一样，直接调用函数就行了。标准 C++中的任何内容（不包括来源于 C 库文件中的函数）则用“`std::`”前缀限定，其全名是“`std::x`”，x 可以是函数、常量、数据结构、系统变量等内容。这样，`std`把标准 C++中的内容统一管理了，能够有效地区别于传统 C++中的同名标识符。

【例 1-2】 从键盘输入一个整数，判断它是否为素数。

其传统 C++和标准 C++的程序如下：

```
// Eg1-2A.cpp (传统 C++)
#include <iostream.h>
#include <stdio.h>
#include <math.h>

void main() {
    int x;
    cout<<"输入数字: ";
    scanf("%d", &x);
    bool prime = true;
    for(int i = 2; (i <= x-1) & prime; i++) {
        if(x%i == 0)
            prime = false;
    }
    if(prime)
        cout<<x<<"是素数! "<<endl;
    else
        cout<<x<<"不是素数! "<<endl;
}
```

```
// Eg1-2B.cpp (标准 C++)
#include <iostream>
#include <cstdio>
#include <cmath>

void main() {
    int x;
    std::cout<<"输入数字: ";
    scanf("%d", &x);
    bool prime = true;
    for(int i = 2; (i <= x-1) & prime; i++) {
        if(x%i == 0)
            prime = false;
    }
    if(prime)
        std::cout<<x<<"是素数! "<<std::endl;
    else
        std::cout<<x<<"不是素数! "<<std::endl;
}
```

两个程序的功能完全相同，但它们调用的函数来源于不同的函数库，这一点通过头文件和 `cin`、`cout` 和 `endl` 的引用方式就可以看出来。在标准 C++ 程序中，每次调用来源于标准库的函数或符号都需要加上 “`std::`” 限定，否则程序是无法通过编译的。但是，如果调用较多就会显得烦琐，可以用 “`using namespace std;`” 一次性引入 `std` 命名空间中的全部名称，然后就可以直接调用标准库中的函数了。

【例 1-3】 修改例 1-2 的标准 C++ 程序，用 “`using namespace std`” 引入标准库的命名空间。

```
// Eg1-3.cpp
#include <iostream>
#include <cstdio>
#include <cmath>
using namespace std;

void main() {
    int x;
    cout<<"输入数字: ";
    scanf("%d", &x);
    bool prime = true;
    for(int i = 2; (i <= x-1) & prime; i++) {
        if(x%i == 0)
            prime = false;
    }
    if(prime)
        cout<<x<<"是素数! "<<endl;
    else
        cout<<x<<"不是素数! "<<endl;
}
```

本程序与例 1-2 的程序完全相同，但用 “`using namespace std`” 一次性引入了 `std` 命名空间

的全部标识符，因此在调用 `cout`、`endl` 等标准库中的函数或标识符时，就不必再用“`std::`”进行限定了。

许多 C++ 编译器同时支持传统和标准 C++ 程序，甚至允许在一个程序中同时调用来源于传统库和标准库的函数。但是，当前 C++ 标准具有的库函数或许更优化，而且新标准具有传统标准 C++ 所没有的新特性，功能更强大。因此，应该多用新标准中的库函数进行程序设计，这样的程序设计常被称为标准 C++ 程序设计。

微软早期的 C++ 编译环境（如 Visual C++ 6.0，简称 VC 6.0）同时支持传统 C++ 和标准 C++ 程序设计，但不支持 C++ 11 及之后的标准。而近年的编译器版本，如 Visual Studio 2019/2022 等，则只支持标准 C++ 程序设计，不支持传统 C++ 编程，如果有“`#include <iostream.h>`”或“`#include <string.h>`”之类的语句，程序不能正确编译。

1.3 数据的输入和输出

程序执行的基本逻辑是“输入数据 → 处理数据 → 输出结果”，数据的输入、输出是程序设计的基本问题，方法是用数据类型定义内存变量，再将数据输入内存变量进行运算，最后将运算结果输出。

1.3.1 数据类型

数据是程序运算处理的对象，被设计成了不同的类型。数据类型是程序分配和使用存储单元的基本技术，是程序设计中最基本、最重要的概念。数据类型决定了能够对它进行的运算规则，相同类型的数据占据相同大小的存储空间，具有同样的运算方法，而不同类型的数据具有不同的表示方法和运算规则。有些语言（如 Smalltalk 和 Python）在程序运行时检查数据类型，称为动态数据类型语言。但 C 语言与之相反，它的类型检查发生在编译时期，在程序被编译时就必须确定每个变量的数据类型，是一种静态数据类型语言。

程序设计语言通常包括三种基本数据类型：字符型、数值型和逻辑型。C 语言的基本数据类型也是这三种，对应的类型符为 `char`、`int`、`bool`。`char` 类型数据储存字符的 ASCII 值，数值类型保存数据的补码，`bool` 类型数据比较特殊，用 0 表示逻辑假值，用非 0 值表示逻辑真值。例如：

```
char x = '2';
int y = 2;
```

虽然 `x` 和 `y` 的值都是 2，但在内存中的存储差异很大。`x` 占 1 字节内存大小，其中保存的值为 50（2 的 ASCII 值），而 `y` 占 4 字节，其中保存的是数字 2。

C++ 语言是从 C 语言发展而来的，保留了 C 语言的类型系统和程序结构，以兼容 C 语言程序的设计和运行。C++ 语言对 C 语言的最大改进是引入了类，提供了程序开发中常用的类型和算法，并允许程序员自定义数据类型。因此，C++ 语言的数据类型非常丰富，如表 1-1 所示。其中，除了 `class`、`string` 和 STL 中的类型，其余类型都与 C 语言相同。表 1-1 中的类型前缀限定符（`short`、`signed` 等）是为了更精确地表示数据而制定的，数据范围表示用它限定对应类型和数据的大小范围，空白表示不能用它限定对应的类型。例如，`signed` 列中的 -128~127 表示 `signed char` 类型的取值范围。

表 1-1 C++语言的数据类型

大类	类型	标识符	长度	范围	类型前缀限定符				
					short	signed	unsigned	long	unsigned long
基本类型	整型	int	4	$-2^{31} \sim 2^{31}-1$	-32768~32767	$-2^{31} \sim 2^{31}-1$	$0 \sim 2^{32}-1$	$-2^{31} \sim 2^{31}-1$	$0 \sim 2^{32}-1$
		long	4	$-2^{31} \sim 2^{31}-1$			$0 \sim 2^{32}-1$	C++ 11 增加	
	字符型	char	1	-128~127		-128~127	0~255		
		wchar_t	2		宽字符				
		char16_t	2					Unicode 字符	
		char32_t	4					Unicode 字符	
	实型	float	4	$-3.4^{38} \sim 3.4^{38}$					
	双精度	double	8	$-1.7^{308} \sim 1.7^{308}$				$-1.7^{308} \sim 1.7^{308}$	
	逻辑型	bool	1	true/false					
空	void								
自定义数据类型	数组	T []	T 可以是上述基本类型，以及结构、联合、类等自定义类型						
	指针	T *							
	引用	T &							
自定义数据类型	结构	struct	除了枚举类型，各种自定义类型都可以包括其他类型定义的若干字段						
	联合	union							
	枚举	enum							
	类	class							
STL	字符串	string	C++标准模板库中的字符串类型，具有强大的字符串存取、运算能力，建议编程中多用它进行字符串的处理，C 语言中没有此类型						
	其他	vector、deque、list、set、multiset、map、multimap、tuple							

short 只能限定 int（即 short int，可省略 int），称为短整数，占 2 字节。

signed/unsigned 可以限定 char、short 和 int 三种类型，分别表示有符号数、无符号数。在默认情况下，这些类型被系统设置为 signed。例如，语句“signed int x;”和“int x;”具有完全相同的作用，定义了有符号整数 x。

long 只能限定 int 和 double，表示长整数和长双精度数。C++ 11 标准中还定义了 long long 类型。

同 C 语言一样，C++语言可以用类型编码字符（大写、小写均可）指定数字常量的类型，示例如表 1-2 所示。

表 1-2 用类型符指定常量的类型

常量值	3	3u	3l	3ul	3.0	3.0f	3.0l	3LL
类型	int	unsigned	long	unsigned long	double	float	long double	long long

注意：在不同编译环境中，long double 和 int 占用的内存大小不尽相同。例如，int 在 16 位机器中的长度为 2 字节，而在 32 位机器中的长度为 4 字节。但是，用 short 和 long 限定的 int 具有固定的长度，在任何支持 C++语言的编译器中，它们的长度都分别为 2 和 4。因此，为了编写可移植的程序，应将整型变量声明为 short 或 long。

1.3.2 流

在 C++中，I/O（Input/Output，输入/输出）数据是一些从源设备到目标设备的字节序列，

称为字节流（简称“流”，**stream**）。除了图像、声音数据，字节流通常代表的都是字符。因此，多数情况的流都是从源设备传输到目标设备的字符序列。

流分为输入流和输出流两类。输入流（**input stream**）是指从输入设备流向内存的字节序列。例如，在图 1-7 中，若源设备是键盘，目标设备是内存，则表示的是输入流，表示通过键盘输入数据到内存变量中。输出流（**output stream**）是指从内存流向输出设备的字节序列。在图 1-7 中，若源设备是内存，目标设备是显示器，就是输出流，表示把内存中的字符逐个输出到显示器。

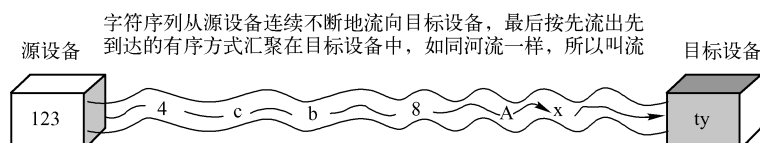


图 1-7 流示意

在 C++ 语言中，标准输入设备通常是指键盘，标准输出设备通常是指显示器。为了从键盘输入数据，或者将数据输出到显示器上，程序中需要包含头文件 **iostream**。该头文件中定义了输入流 **istream** 和输出流 **ostream**，而且用它们定义了 **cin** 和 **cout** 两个对象，近似如下语句：

```
istream cin;  
ostream cout;
```

其中，**cin**（读作 **see-in**）用于从键盘输入数据，**cout**（读作 **see-out**）将内存数据输出到显示器。

1.3.3 cin 和提取运算符 >>

在 C++ 程序中，常用 **cin** 输入数据。其用法如下：

```
cin >> x;
```

程序执行到 **cin** 语句时，就需要从键盘输入数据，输入的数据被插入输入流，数据输完后，按 **Enter** 键结束。当遇到“>>x”时，就从输入流中提取一个数据，存入内存变量 **x**。

① **cin** 一般代表键盘，“>>”是提取运算符，用于从输入流中提取数据，并存储在其后的变量 **x** 中。**x** 是程序中定义的变量名，原则上，**x** 应该是系统内置的简单数据类型，如 **int**、**char**、**float** 等。

② 在一条 **cin** 语句中可以同时为多个变量输入数据。通常，输入数据的个数应当与 **cin** 语句中的变量个数一致，各输入数据之间用一个或多个空白（包括空格、回车、占位符）作为间隔符，全部数据输入完成后，按 **Enter** 键结束。例如：

```
int x1;  
double x2;  
char x3;  
cin >> x1 >> x2 >> x3;
```

假设 **x1** 为 5，**x2** 为 3.4，**x3** 为 'A'，则下面的两种输入方式等效：

5 3.4 A

或（每输入一个数据后按 **Enter** 键）：

```
5  
3.4  
A
```

当一条 `cin` 语句中有多个提取运算符“>>”时，就需要从键盘输入多个数据到输入流，每当遇到一个“>>”，就从输入流中提取一个数据存入其后的变量。

可以把一条 `cin` 语句分解为多条 `cin` 语句，也可以把多条 `cin` 语句合并为一条语句。上面的输入语句与下面的语句组等效：

```
cin>>x1;
cin>>x2;
cin>>x3;
```

③ 在“>>”后面只能出现变量名，这些变量应该用系统预定义的简单类型定义的，否则将出现错误。下面的语句是错误的：

```
cin>>"x" = ">>x;           // 错误，>>后面含有字符串"x = "
cin>>12>>x;               // 错误，>>后面含有常量12
cin>>'x'>>x;              // 错误，>>后面含有字符常量'x'
```

④ `cin` 具有自动识别数据类型的能力，“>>”将根据其后变量的数据类型从输入流中为它们提取对应的数据。例如：

```
cin>>x;
```

假设输入数据 2，“>>”将根据 `x` 的类型决定 2 到底是数字还是字符。若 `x` 是 `char` 类型，则 2 就是字符'2'，会将其 ASCII 值 50 存入 `x`；若 `x` 是 `int`、`float` 之类的类型，则 2 就是一个数字。

再如，若输入 34 且 `x` 是 `char` 类型，则只有字符 3 被存储到 `x`（存入'3'的 ASCII 值 51）中，4 将继续保存在输入流中；若 `x` 是 `int` 或 `float`，则 34 就会被存储在 `x` 中。

⑤ 数值型数据的输入。在读取数值型数据时，“>>”将首先忽略数字前面的所有空白符号，如果遇到正、负号或数字，就开始读入，包括浮点型数据的小数点，当遇到空白或其他非数字字符时，停止提取数据。例如：

```
int x1;
double x2;
char x3;
cin>>x1>>x2>>x3;
```

假如输入“35.4A”并按 Enter 键，第 1 个“>>”根据 `x1` 的类型 `int`，从输入流中提取一个整数存储在 `x1` 中，这个整数只能是 35。因为接下来的“.”不是整数的有效数字，所以提取 35 后，输入流中的数据是“.4A”；第 2 个“>>”将从输入流中为 `x2` 提取数据，`x2` 是 `double` 型，只能把“.4”提取到 `x2` 中，因为接在 4 后面的 A 不是 `double` 类型的有效值，所以 `x2` 的结果为 0.4（0 由系统产生）；第 3 个“>>”为 `x3` 提取数据，`x3` 是 `char` 类型，所以字符'A'就被输入 `x3`。

这个结果或许不正确，却从另一方面说明了，在输入数据时，一定要注意数据之间间隔符的正确输入。结合上述各种情况，来看一个数据输入的综合性例子。

【例 1-4】 假设有变量定义语句如下：

```
int a,b;
double z;
char ch;
```

下面的语句说明了数据输入的含义。

语句

输入

内存变量的值

1	cin>>ch;	A	ch='A'
2	cin>>ch;	AB	ch='A', 而 'B' 被保留在输入流中等待被读取
3	cin>>a;	32	a=32
4	cin>>a;	32.23	a=32, 后面的 .23 被保留在输入流中等待被读取
5	cin>>z;	76.21	z=76.21
6	cin>>z;	65	z=65.0
7	cin>>a>>ch>>z	23 B 3.2	a=23, ch='B', Z=3.2
8	cin>>a>>ch>>z	23B3.2	a=23, ch='B', Z=3.2
9	cin>>a>>b>>z	23 32	a=23, b=32, 计算机等待输入下一个数据存入 z
10	cin>>a>>z	2 3.2 24	a=2, z=3.2, 而 24 被保留在输入流中等待被读取
11	cin>>a>>ch	132	a=132, 计算机等待输入 ch 的值
12	cin>>ch>>a	132	ch='1', a=32

1.3.4 cout 和插入运算符<<

在 C++ 程序中，一般用 `cout` 输出数据，其用法如下：

```
cout<<x;
```

程序执行到 `cout` 语句时，将在屏幕上显示 `x` 的值。`x` 可以是字符串、变量或常量。

`cout` 默认代表显示器，“<<”是插入运算符，用来将其右边的 `x` 的值插入输出流（`cout` 是流的目的地址，所以最终把 `x` 显示在显示器上）。

1. 输出字符类型的数据

字符类型数据包括字符常量、字符串常量、字符变量和字符串变量。对于字符常量和字符串常量，`cout` 把它们原样输出；对于字符变量和字符串变量，`cout` 把变量的值输出。例 1-5 是一个字符输出示例程序。

【例 1-5】 用 `cout` 输出字符数据。

```
// Eg1-5.cpp
#include <iostream>
using namespace std;

void main(){
    char ch1 = 'c';
    char ch2[] = "Hello C++!";
    cout<<ch1;
    cout<<ch2;
    cout<<"C";
    cout<<"Hello everyone!";
}
```

程序的运行结果如下（这个结果是由程序中的 4 条 `cout` 语句共同输出的）：

```
cHello C++!CHello everyone!
```

2. 连续输出

`cout` 语句能够同时输出多个数据，其用法如下：

```
cout<<x1<<x2<<x3<<...;
```

其中，`x1`、`x2` 和 `x3` 可以是相同或不同类型的数据，此命令将依次输出 `x1`、`x2` 和 `x3` 的值。`cout`

的这种语法格式表明，可以把多条 `cout` 语句合并成一条语句。当然，也可以把一条 `cout` 语句分解为多条语句。将 `Eg1-5.cpp` 程序中的 4 条 `cout` 语句合并成如下一条语句，不会影响程序的功能，其运行结果完全相同：

```
cout<<ch1<<ch2<<"C"<<"Hello everyone!";
```

与 C 语言一样，C++ 语言也可以将一条语句写在多行上。例如，上面的语句也可以写成下面的形式：

```
cout<<ch1
    <<ch2
    <<"C"
    <<"Hello everyone!";
```

3. 输出换行

例 1-5 的输出结果并不清晰，如果输出在多行上，效果更好。在 `cout` 语句中，可以通过输出换行符 “\n” 或 `endl` 操纵符，将输出光标移动到下一行的开头处。两者的区别是，`endl` 会在输入流中插入一个换行符并且立即刷新输出缓冲区，而 “\n” 只是插入换行符，但不会刷新输出缓冲区。

【例 1-6】 在例 1-5 的输出语句中增加换行符，使输出结果更清晰。

```
// Eg1-6.cpp
#include <iostream>
using namespace std;

void main(){
    char ch1 = 'c';
    char ch2[] = "Hello C++!";
    cout<<ch1<<endl;                // L1
    cout<<ch2<<"\n";                // L2
    cout<<"C"<<endl;                // L3
    cout<<"Hello everyone!\n";      // L4
}
```

本程序的输出如下：

```
c
Hello C++!
C
Hello everyone!
```

`endl` 与 “\n” 具有相同的功能，它们可以出现在 `cout` 语句中 “<<” 后的任何位置。“\n” 还可以直接放在字符串常数的后面，如语句 L4 中的 “\n”。

4. 输出数值类型的数据

数值类型常量可以利用 `cout` 直接输出，例如：

```
cout<<1<<2<<3<<endl;
```

将输出

```
123
```

数值变量的输出也是如此，如下面的程序段：

```
int x1 = 23;
float x2 = 34.1;
double x3 = 67.12;
cout<<x1<<x2<<x3<<900;
```

其中的 `cout` 语句将输出

```
2334.167.12900
```

从上面两条输出语句的结果可以看出：`cout` 在输出多个数据时，不会在数据之间插入任何间隔符，其结果是使输出数据变得含混不清，如数值 1、2、3 被输出成了“123”。

针对这种情况，需要在 `cout` 输出语句中添加间隔符。例如，可将上面的语句改写为

```
cout<<1<<" "<<2<<" "<<3<<endl;
cout<<"x1 = "<<x1<<" "<<"x2 = "<<x2<<" "<<"x3 = "<<x3<<endl<<900<<endl;
```

输出结果如下，显然它比前面的输出结果更清晰。

```
1  2  3
x1 = 23 x2 = 34.1 x3 = 67.12
900
```

1.3.5 输出格式控制符

在程序运行过程中，常常需要按照一定的格式输出其运行结果，如设置数值精度、设置小数点的位置、设置输出数据宽度或对齐方式……数据输出格式的设置是程序设计的一个重要内容，影响到程序结果的清晰性。

C++ 语言提供了许多控制数据输入输出格式的函数和操纵符（也称为操纵函数或操纵算子），如 `setprecision()`、`setw()`、`right()` 等，它们都是在头文件 `iomanip` 中定义的，应用时要包含该头文件。

1. 设置浮点数的精度

在需要设置输出数据的精度时，可以采用操纵函数 `setprecision()`，用法如下：

```
setprecision(n)
```

其中，`n` 代表有效数位，包括整数的位数和小数的位数。如 `setprecision(3)` 将所有数值的输出精度都指定为 3 位有效数字，直到再次用 `setprecision()` 函数改变精度为止。例如，语句

```
cout<<setprecision(3)<<3.1415926<<" "<<2.4536<<endl;
```

将输出

```
3.14 2.45
```

2. 设置输出域宽和对齐方式

操纵函数 `setw()` 用于设置输出数据占用的列数（域宽，即占用的字符个数），用法如下：

```
setw(n)
```

其中，`n` 是输出数据占用屏幕宽度的字符数，默认输出数据按右对齐。若输出数据的位数比 `n` 小，则左边留空；若输出数据的实际位数比 `n` 大，则输出数据将自动扩展到所需占用的列数。例如：

```
cout<<"1234567812345678"<<endl; // L1
```



```
cout<<setw(8)<<23.27<<setw(8)<<78<<endl;           // L2
cout<<setw(8)<<"Abc"<<78<<endl;                       // L3
```

上述语句的输出结果如下：

```
1234567812345678
23.27      78
Abc78
```

setw()只对紧随其后的一个输出数据有效，语句 L3 中的 setw(8)只对跟在其后的字符串 "Abc"有效，所以最后的 "78" 按默认方式输出，紧接在"Abc"的后边。

3. 设置对齐方式

操纵函数 setiosflags()和 resetiosflags()可用于设置或取消输入、输出数据的各种格式，包括改变数制基数、设置浮点数的精度、转换字母大小写、设置对齐方式等。其用法如下：

```
setiosflags(long f);
resetiosflags(long f);
```

iostream 头文件还定义了两个表示对齐方式的常数，表示左对齐的常数值是 ios::left，表示右对齐的常数值是 ios::right，它们可作为 setiosflags 和 resetiosflags 操纵符函数的参数，用于设置输出数据的对齐方式。在默认方式下，C++语言按右对齐方式输出数据。用 setiosflags()设置输出对齐方式成功后，将一直有效，直到用 resetiosflags()取消它。

4. 设置正号显示和布尔值

默认情况下，输出正数时，不会输出正号，布尔类型在输出时用 0 表示 false，用 1 表示 true。在有些情况下，需要输出正号，将逻辑结果输出为 true 或 false，可以在输入流中插入相应的操作符，showpos（设置正号）、noshowpos（取消正号）、boolalpha（用 true 和 false 表示布尔值）。此外，在输出流中插入 scientific 可以将数据设置为科学记数法显示，设置后将一直有效，直到再次设置为其他数据显示方式才会被取消。

```
cout<<showpos<<p<<"\t"<<4.5<<2<<noshowpos<<"\t"<<4.8<<endl;
cout<<scientific<<p<<"\t"<<123.01<<"\t"<<fixed<<3.40009<<endl;
cout<<(3>2)<<"\t"<<(3>5)<<"\t"<<boolalpha<<(3>2)<<"\t"<<(3>5)<<endl;
```

这三条语句的输出结果如下：

```
+3.141593    +4.5+2    4.8
3.141593e+00  1.230100e+02  3.400090
1    0    true    false
```

其中，p=3.141593。

【例 1-7】 用 setiosflags()和 resetiosflags()设置和取消输出数据的对齐方式。

```
// Eg1-7.cpp
#include <iostream>                                // L1
#include <iomanip>                                  // L2
using namespace std;

void main() {                                       // L3
    cout<<"123456781234567812345678"<<endl;         // L4
    cout<<setiosflags(ios::left)<<setw(8)<<456<<setw(8)<<123<<endl; // L5
    cout<<resetiosflags(ios::left)<<setw(8)<<123<<endl; // L6
```

```
}
```

这个程序的输出结果如下：

```
123456781234567812345678
456      123
123
```

输出结果的第 1 行是语句 L4 输出的；第 2 行是语句 L5 输出的，输出的两个数据各占 8 位，且设置了左对齐方式；第 3 行是语句 L6 的输出，输出数据占 8 位，由于在输出之前用 `resetiosflags(ios::left)` 操纵符取消了左对齐，使数据输出又变成了默认的右对齐方式，因此输出数据的左边留了 5 个空白。

1.3.6 数制基数

同 C 语言一样，C++ 语言可用 `0x/0X` 表十六制常量、`0` 表示八进制常量，C++ 14 增加了用 `0b/0B` 表示二进制，同时支持用单引号对数字进行分位表示，以便于大数据阅读。

【例 1-8】 数字常量的进制控制和千分位间隔表示。

```
// Eg1-8
#include <iostream>
using namespace std;

void main() {
    int x1 = 23;
    int x2 = 023;
    int x3 = 0x23;
    int x4 = 0b11011101; // C++ 14
    int x5 = 6'123'456'789; // C++ 14
    int x6 = 0b101'111'001'111; // C++ 14
    cout << "x1 = " << x1 << "\tx2 = " << x2 << "\tx3 = " << x3 << "\n"
         << "x4 = " << x4 << "\tx5 = " << x5 << "\tx6 = " << x6 << endl;
    getchar();
}
```

在支持 C++ 14 以上标准的编译环境中，程序运行的结果如下（x5 的结果是因溢出了 int 表示范围而产生的数据）：

```
x1 = 23   x2 = 19   x3 = 35
x4 = 221  x5 = 1828489383  x6 = 3023
```

当程序需要在运行过程中通过 `cin` 和 `cout` 输入、输出不同进制的数字时，可在 `cin` 或 `cout` 语句中应用 `iostream` 头文件中预定义的 `hex`、`oct`、`dec` 操纵符，分别表示十六进制数、八进制数和十进制数。在用键盘输入数据时：

- ❖ 十进制数：直接输入数据本身，如 78。
- ❖ 十六进制数：在要输入的数据前加 `0x` 或 `0X`，如 `0x1A`（对应的十进制数是 26）。
- ❖ 八进制数：在输入的数据前加 `0`，如 `043`（代表十进制数 35）。

【例 1-9】 输入、输出不同进制的数。

```
// Eg1-9.cpp
#include <iostream>
```

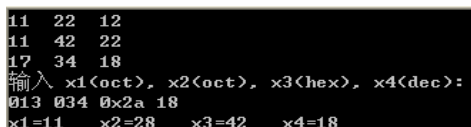
```

using namespace std;

void main(){
    int x = 34;
    cout<<hex<<17 <<" "<<x<<" "<<18<<endl;
    cout<<17 <<" "<<oct <<x<<" "<<18<<endl;
    cout<<dec<<17 <<" "<<x<<" "<<18<<endl;
    int x1, x2, x3, x4;
    cout<<"输入 x1(oct), x2(oct), x3(hex), x4(dec):"<<endl;
    cin>>oct>>x1; // 八进制数
    cin>>x2; // 八进制数
    cin>>hex>>x3; // 输入十六进制数
    cin>>dec>>x4; // 输入十进制数
    cout<<"x1="<<x1<<"\tx2="<<x2<<"\tx3="<<x3<<"\tx4="<<x4<<endl;
}

```

设置数制基数后，它将一直有效，直到遇到下一个基数设置。本程序运行结果如图 1-8 所示。其中，第 1 行和第 2 行的 11 是十六进制数，第 2 行的 42 和 22 是八进制数，第 3 行都是十进制数。第 5 行是从键盘输入的数据，013、034 是八进制数，0x2a 是十六进制数，18 是十进制数。最后一行是按十进制输出的数据。



```

11 22 12
11 42 22
17 34 18
输入 x1(oct), x2(oct), x3(hex), x4(dec):
013 034 0x2a 18
x1=11 x2=28 x3=42 x4=18

```

图 1-8 程序运行结果

1.3.7 string 和字符串的输入、输出

字符串是计算机中应用最多的一类数据，如文字的屏幕显示或打印输出，Word 程序的文档编排、文字查找与替换，对程序而言，都是字符串处理。然而，C 语言中并没有字符串数据类型，熟悉 C 程序设计的人们往往习惯性地采用 char 类型的指针或数组进行字符串处理，并采用一套独立的函数实现字符串的比较、查找和替换等操作，比较麻烦。

C++语言的基本类型系统中也没有字符串类型，但在其标准模板库（STL）中提供了字符串类型——string，可以像 int、double 等基本数据类型一样定义 string 类型的对象，并可用>、<、>=、<=、<>、=、+=等运算符进行字符串的各种运算。

此外，string 具有字符串的查找、替换、取子串、插入子串等处理能力（见本书 7.5.2 节），进行程序中的字符串处理非常便捷。

1. string 对象的定义和初始化

string 是 STL 中定义的字符串处理类，存放在头文件 string 中。因此，若要引用 string 类，应当在程序中“#include <string>”。用 string 定义对象有以下几种形式：

```

string c; // 定义字符串 c，不含任何字符
string c1("this is a string"); // 定义字符串 c1，并用指定字符串初始化其内容
string c2 = c1; // 定义字符串 c2，并用 c1 初始化它
string s[10]; // 定义字符串数组，能够保存 10 个字符串，相当于 char[][];
string s(5, 'c'); // 定义 s，用 5 个 'c'，即“ccccc”初始化

```

2. string 类型的赋值

string 类型的赋值操作与 int 等基本类型的赋值操作相同，不必用 strcpy() 函数。例如：

```
string s1, s2, s3[3];           // 定义 string 对象及数组
string name[3] = {"tom", "jerry", "duck"}; // string 对象数组定义与初始化
s1 = "this is a string!";      // string 赋值
s2 = s1;
s3[0] = s1;                     // string 数组元素访问
s3[1] = "string arr";
```

3. string 类型的连接

用 “+” 和 “+=” 可以对两个 string 类型对象进行连接运算。例如：

```
string s1("I am a boy"), s3;
string s2 = "I come from China!";
s3 = s1 + "," + s2;             // s3: I am a boy, I come from China!
s1 += "," + s2;                 // s1: I am a boy, I come from China!
```

4. string 类型的输入、输出和大小比较

string 类型的输入、输出与 int 等基本类型相同，可以用 cin 和 cout 直接输入或输出，比较运算符 >、>=、==、<、<=、!= 可以对 string 类型的变量进行比较，同 C 语言的字符串比较运算一样，实际上比较的是两个 string 对象对应位置字符的 ASCII 值。

【例 1-10】 输入两个字符串，并比较其大小。

```
// Eg1-10.cpp
#include <iostream>
#include <string>
using namespace std;

int main(){
    string s1, s2, big;
    cout<<"输入两个字符串: "<<endl;
    cin>>s1 >> s2;
    cout<<"参加比较的两个字符串是: "<<s1<<" ", "<<s2<<endl;
    if(s1 > s2)
        big = s1;
    else if (s1 == s2)
        big = "same";
    else
        big = s2;
    cout<<"大字符串是: "<<big<<endl;
    return 0;
}
```

程序运行结果如下：

```
输入两个字符串：
tom jerry           // 键盘输入
参加比较的两个字符串是: tom, jerry      // 屏幕输出
大字符串是: tom
```

1.3.8 数据输入的典型问题

在为程序变量输入数据时，如果类型不匹配，或者对输入流中的数据提取控制不恰当，都有可能导致程序运行错误，或者产生难以理解的运行结果，常见问题有以下 3 种。

1. 输入数据类型不匹配引发的问题

按照 `cin` 语句的先后次序，依次为各语句中指定的变量，从键盘输入正确类型的数据，程序才能够正确运行。否则，即使程序完全正确，但输入数据有问题，程序也可能出现运行错误，甚至无法正常运行。

【例 1-11】 从键盘为不同类型的变量 `a`、`b`、`z` 等输入数据，分析输入数据类型不当引发的错误。

```
// Eg1-11.cpp
#include <iostream>
using namespace std;

int main() {
    int a, b;
    double z;
    char ch;
    cin>>ch;
    cin>>a>>b;
    cin>>z;
    cout<<"ch = "<<ch<<"\ta = "<<a<<"\tb = "<<b<<"\tz = "<<z<<endl;
    return 0;
}
```

当正确输入数据时，本程序没有任何问题。例如，按下面的方式输入数据，各数据之间由空格间隔：

```
A 32 49 8.7 // 键盘输入
ch = A a = 32 b = 49 z = 8.7 // cout 产生的屏幕输出
```

但是，如果不小心多输入了一个字符 `B`，则会产生难以理解的输出结果：

```
AB 32 49 8.7 // 键盘输入
ch = A a = -858993460 b = -858993460 z = -9.25596e+61 // cout 产生的屏幕输出
```

产生这个结果的原因是：当“>>”从输入流中提取字符 `A` 并存入变量 `ch` 后，接下来应当为 `a` 提取数据，由于“>>”从输入流中提取数据是依次进行的，这次提取的数据只能是字符“`B`”，它与 `a` 的类型不符合（`a` 为 `int`）。遇到这种情况，就无法把当前提取的数据保存在变量 `a` 中，C++ 程序并不报告错误，而是设置输入失效位，并关闭输入流，此后的所有 `cin` 语句会被忽略而不被执行（直到用 `cin.clear()` 清除该失效位）。但是，程序不会终止，其他语句照样会被执行。

因此，变量 `a` 的值其实是执行 `cin` 语句之前的旧值。由于 `a` 是 `main()` 中的局部变量，且其值未被初始化，是一个未知值，Visual C++ 编译器将未初始化的 `int` 数据处理成“-858993460”（在其他 C++ 编译环境中则不一定是此数）。同理，为 `a`、`b`、`z` 输入数据的 `cin` 语句也不会被执行，它们的值实际上是建立这些局部变量时保留的未知值。从中还可以了解到，Visual C++ 将 `double` 类型的未初始化局部变量处理成“-9.25596e+61”。

2. 为变量输入空白字符的问题

提取运算符“>>”从输入流中提取一个数据项时，将略掉以任何方式产生的空白（如按空格键、Enter 键、Tab 键产生的空白）。假设有变量 `c1`、`c2`、`n`，其定义和输入语句如下：

```
char c1, c2;
int n;
cin>>c1>>c2>>n;
```

若需要将字符 `A` 输入 `c1`、空白输入 `c2`、`3` 输入 `n`，则只能从键盘依次输入数据“`A 3`”，但是，这样的输入流只能够将 `A` 存入 `c1`、`3` 存入 `c2`，`n` 得不到任何输入值。

1) 用函数 `get()` 输入空白字符

实际上，`cin` 是一个功能强大的对象，具有许多成员函数，如 `get()`、`ignore()`、`putback()`、`getline()`，具备读取空白字符，以及包含空白的字符串的能力。利用 `cin` 的函数 `get()` 可以提取输入流中的任何字符，包括空格键、Enter 键、Tab 键等的输入，用法如下：

```
cin.get(char c);
```

其中，`c` 是 `char` 类型的字符变量。函数 `get()` 将从输入流中提取当前字符并存入字符变量 `c`，不会略过任何符号（包括用空格键、Enter 键、Tab 键等方式输入的空白字符）。

为了将 `A` 存入 `c1`，将空白存入 `c2`，将 `3` 存入 `n`，可修改上面的 `cin` 语句为如下语句组：

```
cin.get(c1);
cin.get(c2);
cin>>n;
```

这组语句与下面的语句组是等价的：

```
cin>>c1;
cin.get(c2);
cin>>n;
```

函数 `get()` 有多种用法，如不带参数的函数 `get()` 可用于略过输入流中的当前字符。

2) 用函数 `getline()` 输入包含空白的长字符串

函数 `getline()` 可以一次读取一行字符，其用法如下：

```
cin.getline(char *c, int n, char = '\n');
```

其中，`c` 是保存输入数据的数组，`n` 为要提取的字符个数，指示从输入流中读取 `n-1` 个字符到数组 `c` 中（系统会在第 `n` 个位置填写结束符“`\0`”），第 3 个参数用于指定停止从输入流中提取数据的结束符（默认结束分隔符是 `\n`，可以在此指定其他结束字符）。

函数 `getline()` 有两种结束方法：① 输入流中的字符个数多于指定的个数，已从输入流读够了 `n-1` 个字符；② 虽然没有读够 `n-1` 个字符，但遇到了指定的结束符号。

【例 1-12】 用函数 `getline()` 读取一行键盘输入。

```
// Eg1-12.cpp
#include<iostream>
using namespace std;

void main() {
    char s1[100];
    cout << "use getline input char: ";
    cin.getline(s1, 11);
```



```
    cout<<s1<<endl;
}
```

下面是程序执行时的一组输入数据和输出结果：

```
use getline input char: Hello C++, I am Tom
Hello C++
```

输出表明，s1 字符串得到的输入为“Hello C++”。这说明当输入流中的字符多于 getline() 指定的字符个数时，getline() 只提取指定个数的字符，多余的字符被忽略。

再次执行程序，输入和输出情况如下：

```
use getline input char: Hello
Hello
```

表明 s1 得到的字符串为“Hello”，说明 getline() 从输入流中提取的字符数虽然少于指定个数，但遇到了指定的结束字符（默认为回车符）也会结束数据读取。

3. 函数 getline() 没有读取输入数据就结束了的问题

1) 输入流中的字符多于 getline() 需要的字符数

当函数 getline() 以上面提到的第一种方式结束时，执行后续 cin 语句就会发生“还没有从键盘为之输入数据，该语句就结束了”之类的问题。

其原因是：当输入流中的字符多于函数 getline() 指定接收的字符个数时，会将把余下的字符留在输入流中，同时会设置输入失效位，并关闭输入流。也就是说，此后的所有 cin 语句都失效，不会再被执行了。

【例 1-13】 从键盘为两个字符串输入数据，字符串中可能包括空白字符。

```
// Eg1-13.cpp
#include<iostream>
using namespace std;

void main() {
    char s1[100];
    char s2[10];
    cout<<"use getline input s1: ";           // L1
    cin.getline(s1, 10);                      // L2
    cout<<"input s2: "<<endl;                 // L3
    // cin.clear();                           // L4*
    // cin.ignore(1024, '\n');                 // L5*
    cin.getline(s2, 6);                       // L6
    cout<<"s1 = "<<s1<<endl;                  // L7
    cout<<"s2 = "<<s2<<endl;                  // L8
}
```

执行该程序，若为 L2 的 getline() 输入的字符个数小于 10 个，程序运行情况正常。但是，如果输入的字符串多于指定的接收个数 11，结果就不对了，如下所示：

```
use getline input char: Hello C++,I am Tom
input s2:
s1 = Hello C++
s2 =
```

// L1、L2 执行情况
// L3 执行情况
// L7 执行情况
// L8 执行情况

此结果表明，程序并未在语句 L6 处停下来等待键盘输入，输出结果的最后一行表明它好像确实没有获取什么字符。如果清楚程序建立的输入流，并且知道 `cin`、`getline()` 的数据提取方法，就容易理解 `getline()` 没有提取数据的原因，找到解决此问题的办法。

当执行语句 L2 时，从键盘输入 “Hello C++, I am Tom\n”，则建立了图 1-9 的输入流，由于函数 `getline()` 的第 2 个参数为 10，因此它从输入流中为 `s1` 提取 9 个字符（第 10 个字符为结束符 “\0”，由系统提供），在 `p1` 位置结束数据提取，并设置输入失效位，同时关闭输入流，忽略语句 L6 的执行。事实上，即使语句 L6 后面还有其他 `cin` 语句，也不会再被执行。

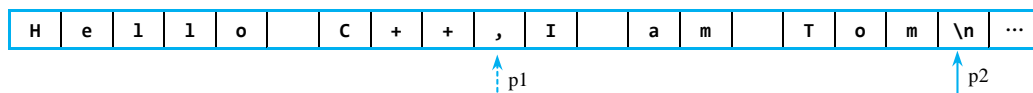


图 1-9 例 1-13 的输入流

解决这类问题的方法很简单，只需将函数 `getline()` 的第 2 个参数设置为一个更大的数字，使函数 `getline()` 提取数据时到达指定结束符处 `p2` 就行了。此外，可按下述方法解决。

解决此问题的方法需要两个步骤：<1> 调用 `cin` 对象的函数 `clear()`，将设置为失效的输入位重新设置为有效；<2> 调用 `cin` 对象的函数 `ignore()`，忽略残留在输入流中的多余字符。

函数 `clear()` 只是恢复输入位有效，使后续的 `cin` 语句能够正常接收输入数据，但它不删除当前输入流中的剩余数据，若不删除，下一条读数据的语句会接着读取这些数据。例如，在图 1-9 中，使用 `clear()` 恢复输入位后，下一条读数据的语句（L6）将从 `p1` 位置开始为 `s2` 读取数据。但这不是程序需要的，需要用函数 `ignore()` 忽略余下的字符，从键盘为 `s2` 输入数据。

函数 `ignore()` 的用法如下：

```
cin.ignore(int nCount = 1, char delim = EOF);
```

其中，`nCount` 是忽略掉的字符个数，默认为 1，`delim` 可以指定结束符位置，默认值为 `EOF`。

函数 `ignore()` 有两种结束方式：一是到达了指定忽略个数的字符位置，二是遇到了指定的结束字符。若指定忽略的个数太少，下一条读数据的语句会接着读取剩下的输入字符。为了避免这种情况，通常把忽略个数设置为一个足够大的数字。以下是 `ignore()` 的两种典型用法：

```
cin.ignore(); // 忽略一个字符
cin.ignore(1024, '\n'); // 忽略 1024 个字符，或遇回车字符就结束
```

在例 1-12 中，取消 L4 和 L5 语句前面的注释符后，程序就能够正常运行了。

2) 函数 `getline()` 提取了上一条 `cin` 语句遗留在输入流中的 “\n” 结束符而不读数据的问题

【例 1-14】 设计一个程序，从键盘输入学生的学号和姓名，其中外国学生的姓名由 `first name` 和 `second name` 组成，两者之间用空白作间隔。

程序设计思路：用 `sno`、`name` 分别表示学号和姓名，由于 `name` 中可能包括空白，因此用函数 `getline()` 为它输入数据。

```
// Eg1-14.cpp
#include <iostream>
#include <string>
using namespace std;

void main(){
    int sno;
    char name[10];
```

```

cout<<"input Sno: ";
cin>>sno;                                // L1
cout<<"input Name: ";
cin.getline(name, 10);                    // L2
cout<<"Sno: "<<sno<<endl;
cout<<"Name: "<<name<<endl;
}

```

程序运行结果如下：

```

input Sno: 12345
input Name:
Sno: 12345
Name:

```

此结果表明，当执行语句 L1，从键盘输入 12345 并按 Enter 键后，并未等待语句 L2 的键盘数据输入，程序就结束了。分析如下：执行语句 L1，从键盘输入 12345 并按 Enter 键后，建立了如图 1-10 所示的输入流。cin 为 sno 读取有效数字“12345”后遇到 p 处的“\n”，因而结束数据提取，“\n”则成为输入流中下一次提取数据的第 1 个字符。

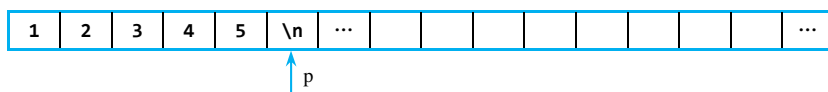


图 1-10 例 1-14 的输入流

语句“cin.getline(name, 10)”执行时，首先遇到“\n”，于是为 name 提取了该符号。由于函数 getline() 的默认结束符是“\n”，因此将结束执行。本程序如果还有后续 cin 语句，将被正常执行。

解决这类错误的方法是“吃掉”上一条 cin 语句保留在输入流 p 位置的“\n”，使语句 L2 正常执行。字符读取函数 getchar()、cin 的成员函数 get() 和 ignore() 都能够从输入流的当前位置提取一个字符，利用它们提走 p 位置的“\n”，就解决了程序的错误。

函数 getchar() 并非 cin 的成员，而是字符读取函数，直接调用即可。而 get() 和 ignore() 是 cin 的成员函数，因此调用时要用“cin.”进行限定。不带参数的 get() 函数会提取输入流当前位置的符号（包括“\n”在内的任何字符），ignore() 则会略过输入流中指定个数的字符。

因此，解决本例程错误的方法是：在语句 L1 后调用函数 getchar()、get() 或 ignore() “吃掉”输入流中的字符“\n”。

```

cin >> Sno;                                // L1
getchar();                                // 或 cin.get();    或 cin.ignore(1);

```

1.4 编程实作：Visual C++ 2022 编程简介

支持 C++ 程序设计的编译程序很多，常见的有 Clang、Dev C++ 和 Visual C++ 等。Visual C++ 是 Microsoft 公司推出的基于 Windows 的集成开发环境，简称 VC++，提供了编写程序源代码的编辑器，创建各类资源文件（如对话框、图标、菜单等）的资源编辑器，具有编辑、编译、链接等功能。利用它可以输入、编辑源程序，进行程序的编译、调试、链接，最后生成可执行的命令程序。

VC++ 6.0 是一个常用的 C++ 编译器版本，但不支持 C++ 11 及之后的 C++ 标准。本书的某些例程需要在支持 C++ 11、C++ 14、C++ 17、C++ 21 标准的程序环境中运行，所以选择了 VC++ 2022 开发环境。下面以一个简单的例子介绍在此环境中编写 C++ 程序的过程。

【例 1-15】 某次考试成绩如下，编写程序计算每位同学的平均分。要求成绩从键盘输入，输出结果的形式与下面相同，但要输出每位同学的平均分。

	语文	数学	政治	化学	英语	平均分
学生 1	67	76	87	89	76	
学生 2	78	87	78	90	87	
.....						

程序设计思路：设计一个二维数组 s，保存学生的成绩和平均分；设计一个读入学生成绩表的函数 ReadData()，将学生成绩读入数组 s 的前 5 列中；设计一个计算平均成绩的函数 AveScore()，计算每位同学的平均成绩，并将计算结果放入 s 数组的第 6 列；设计一个输出数据的函数 OutData()，将 s 数组的数据按指定格式输出。

1. 在 VC++ 中编辑源程序

在 Windows 10 的 VC++ 2022 环境中实现例 1-15 程序的过程如下。

<1> 选择“开始 | 所有程序 | Visual Studio 2022”菜单命令，启动 VC++ 2022。

<2> 选择“文件 | 新建 | 项目”菜单命令，弹出“创建新项目”对话框，如图 1-11 所示。



图 1-11 Visual C++ 2022 的“创建新项目”对话框

<3> 选择“C++”和“Windows”，并从右边的列表中选择“空项目”，然后单击“下一步”按钮。

<4> 弹出“配置新项目”对话框，在“项目名称”文本框中输入项目名称“Eg1-15”，在“位置”文本框中指定本项目保存的磁盘位置（如 D:\temp）。然后单击“创新”按钮，在 D:\temp 目录下创建了 Eg1-15 项目文件夹，并进入 Visual C++ 的编程环境，如图 1-12 所示。

<5> 选择“项目 | 添加新项目”，在弹出的“添加新项目”对话框中选择“C++ 文件(.cpp)”，并在“名称”文本框中输入源文件名“Eg1-15.cpp”，然后单击“添加”按钮。

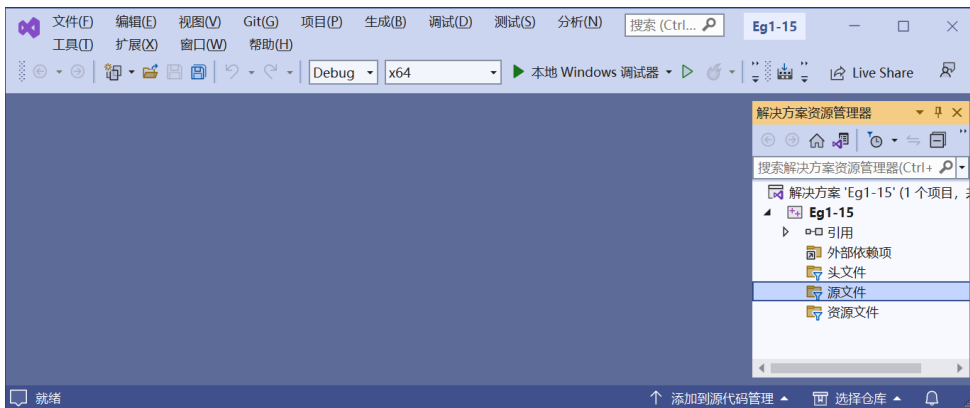


图 1-12 Visual C++ 2022 编程环境

<6> 展开“解决方案管理器”中的项目“Eg1-15”，双击“源文件”中的“Eg1-15.cpp”，输入如下代码。

```
// Eg1-15.cpp
#include <iostream>
#include <iomanip> // setw 在此头文件中定义
using namespace std;
#define StuNum 5 // StuNum 代表学生人数

void ReadData(double s[][6],int n); // 这 3 行是函数声明
void AveScore(double s[][6],int n);
void OutData(double s[][6],int n);
void main() {
    double s[StuNum][6]; // 定义保存学生成绩的数组
    ReadData(s,2); // 读入学生成绩
    AveScore(s,2); // 计算各学生的平均分
    OutData(s,2); // 输出学生成绩表
}

void ReadData(double s[][6], int n) {
    for(int i = 0; i < n; i++) {
        cout<<"输入学生 "<<i+1<<" 的 5 科成绩: "; // 在屏幕上提示输入学生成绩
        for(int j = 0; j < 5; j++) // 输入学生的 5 科成绩
            cin>>s[i][j];
    }
}

void AveScore(double s[ ][6],int n) {
    for(int i = 0; i < n; i++){
        double sum = 0;
        for(int j = 0; j < 5; j++)
            sum = sum+s[i][j];
        s[i][5] = sum/5.0;
    }
}

void OutData(double s[][6], int n) { // 下面的 cout 语句在屏幕上输出科目名称
    cout<<setw(17)<<"语文"<<setw(8)<<"数学"<<setw(8)<<"政治"
```

```

        <<setw(8)<<"化学"<<setw(8)<<"英语"<<setw(8)<<"平均分"<<endl;
    for(int i = 0; i < n; i++){
        cout<<setw(8)<<"学生 "<<i+1;
        for(int j = 0; j < 6; j++)
            cout<<setw(8)<<s[i][j];
        cout<<endl;
    }
}

```

2. 编译和调试程序

选择“编译 | 重新生成解决方案”菜单命令，编译源程序，并在输出窗口中指出编译的结果。如果在编译过程中发现了错误，就会显示在输出窗口中。若有错误，则修改后再次编译程序，直到改正了全部错误。

编译成功后，选择“调试 | 开始执行”菜单命令，或按快捷键 **Ctrl+F5**，执行该程序。结果如图 1-13 所示。其中，第 1、2 行后面的数字是程序执行函数 `ReadData()` 时从键盘输入的成绩，第 3、4、5 行是函数 `OutData()` 的输出。

```

输入学生 1 的5科成绩: 67 76 87 89 76
输入学生 2 的5科成绩: 78 87 78 90 87
      语文    数学    政治    化学    英语    平均分
学生 1      67      76      87      89      76      79
学生 2      78      87      78      90      87      84
Press any key to continue_

```

图 1-13 计算学生成绩平均分程序的运行结果

3. Visual Studio 运行结束后闪退问题

Visual Studio 2022 等版本在默认安装情况下，可能设置“调试停止时自动关闭控制台”选项，会导致在 Visual Studio 环境中执行程序（包括按组合键 **Ctrl+F5**）后，没有看到程序输出结果就“闪退”了。如遇到这样的情况，可用如下方法解决。

<1> 选择“调试 | 选项”菜单命令，弹出“选项”对话框，如图 1-14 所示。

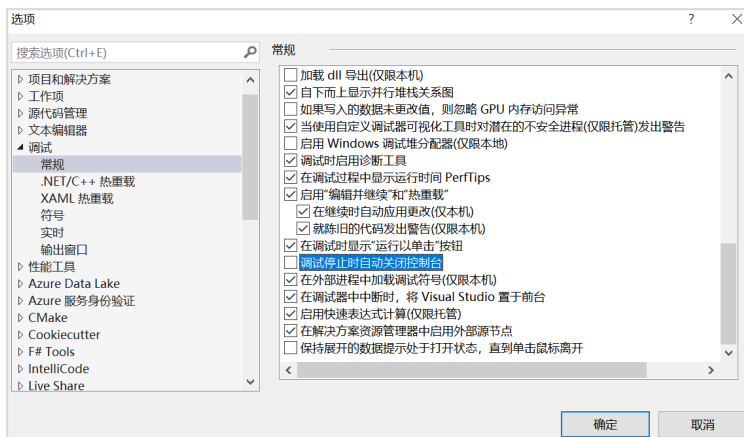


图 1-14 取消“调试停止时自动关闭控制台”

<2> 选中“调试 | 常规”列表项，并在右边的“常规”列表中取消“调试停止时自动关闭控制台”复选框的勾选，然后单击“确定”按钮。

习题 1

- 1.1 什么是抽象和封装？
- 1.2 什么是类、对象、继承和多态？试举例说明。
- 1.3 面向对象程序设计与面向过程程序设计有什么区别？
- 1.4 C++语言有什么特点？
- 1.5 理解流的概念。
- 1.6 阅读下面的程序，写出程序的输出结果。

(1)

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

void main() {
    int a;
    char b;
    char c[4];
    double d;
    cin>>a>>b>>c>>d;
    cout<<"a"<<a<<endl;
    cout<<"b"<<b<<endl;
    cout<<"c"<<c<<endl;
    cout<<"d"<<d<<endl;
}
```

输入数据“12 345 634 3214”并按 Enter 键后。

(2)

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

void main() {
    int a = 20, b = 18, c = 24;
    cout<<"123456789012345678901234567890"<<"\n";
    cout<<setiosflags(ios::left);
    cout<<hex<<setw(10)<<a<<setw(10)<<b<<setw(10)<<c<<endl;
    cout<<oct<<setw(10)<<a<<setw(10)<<b<<setw(10)<<c<<endl;
    cout<<resetiosflags(ios::left);
    cout<<dec<<setw(10)<<a<<setw(10)<<b<<setw(10)<<c<<endl;
}
```

(3)

```
#include <iostream>
#include <bitset>
using namespace std;
int main() {
    int x1 = 56;
    int x2 = 070;
    int x3 = 0x38;
```

```

int x4 = 0b000'011'000;
cout<<"x1 = "<<x1<<"\tx2 = "<<x2<<"\tx3 = "<<x3<<"\tx4 = "<<x4<<endl;
cout<<"x1 = "<<bitset<8>(x1)<<"\tx2 = "<<bitset<10>(x2)
    <<"\tx3 = "<<0x38<<"\tx4 = "<<070<<endl;
}

```

bitset<n>(x)用于将 x 转换成 n 位二进制数。

1.7 用函数 setw()和 cout、for 语句循环，编写输出下面图形的程序。

```

*
***
*****
*****

```

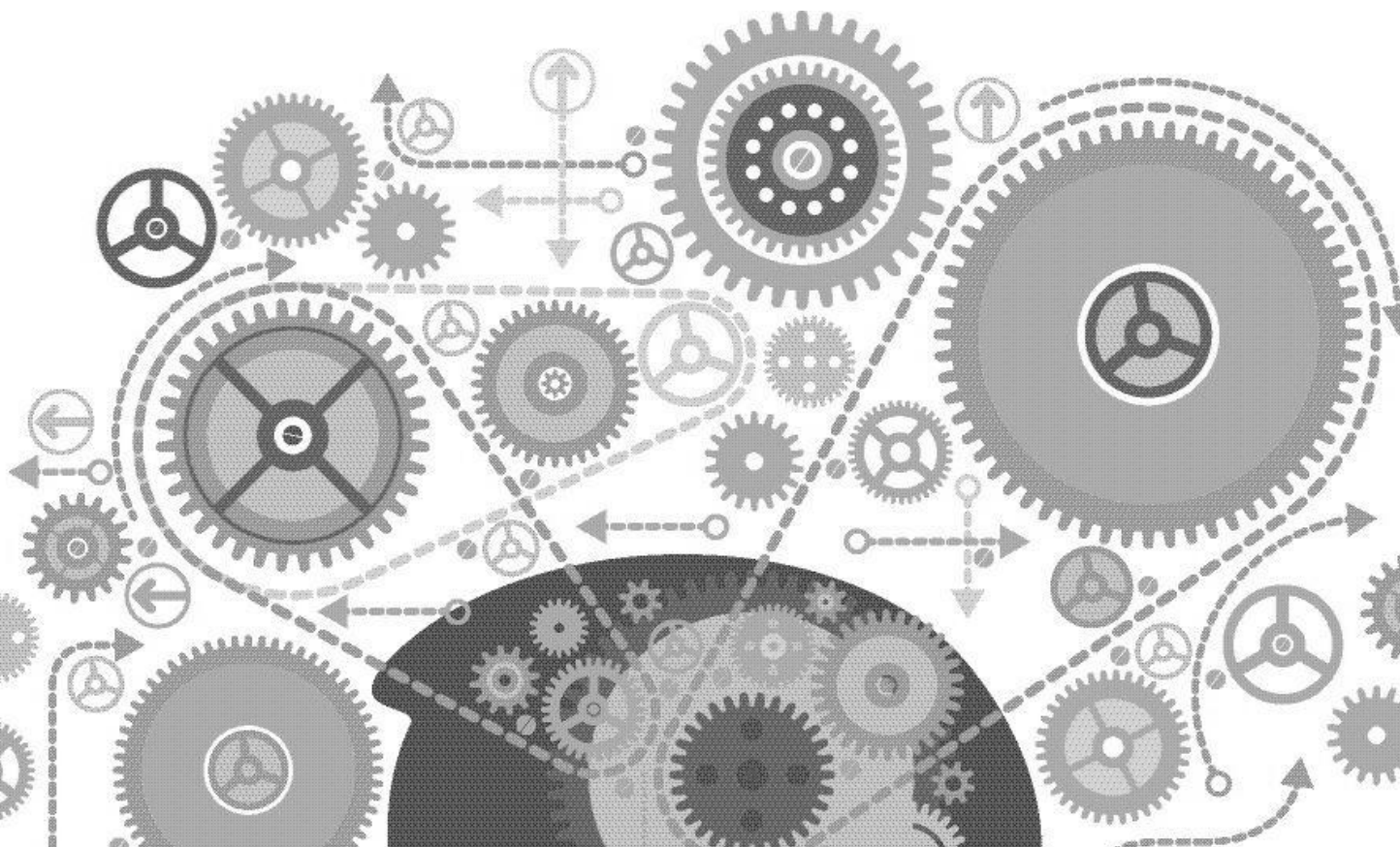
1.8 某校教师的课酬计算方法是：教授 100 元/小时，副教授 80 元/小时，讲师 60 元/小时，助教 40 元/小时。编写计算教师课酬的程序，从键盘输入教师的姓名、职称、授课时数，然后输出该教师应得的课酬。

第 2 章

C++ 程序设计基础

C++语言是从C语言发展而来的，保留了C语言的特性，是C语言的超集。C语言原有的数据类型、表达式、语句命令、函数及程序在C++语言中仍然可用。C++语言对C语言的最大改变就是引入了面向对象程序设计的机制，并对C语言的某些特性进行了扩展。

本章介绍C++语言对C语言非面向对象方面的扩展，包括智能指针、左值引用和右值引用、`const` 和 `constexpr` 常量、范围 `for` 和初始化列表、自动类型推断、函数重载和内联函数、Lambda 表达式、命名空间、预处理器、文件处理等内容。



2.1 C++语言对C语言的类型扩展和类型定义

1. C++语言对 C 语言的类型扩展

C++语言保留了 C 语言的基本数据类型，并对 C 语言的结构、联合、枚举等自定义数据类型进行了扩展。C++定义的结构名、联合名、枚举名等都是类型名称，可以直接用于变量的声明或定义，不必再加上 `struct`、`union`、`enum` 这样的前缀。此外，在结构和联合中还可以定义函数（见第 3 章）。例如，有下述类型声明：

```
enum color{black, white, red, blue, yellow};           // C, C++ 98/03
enum class color1{black1, white1, red1, blue1, yellow1}; // C++ 11
struct student {
    char Name[6];
    int age;
    int getAge() { return age; }
};
union xy {
    int x;
    char y;
    int f() { return x+y; }
};
bool black = false;                                     // L1, 错误, 不能通过编译
bool black1 = false;                                    // L2, 正确
```

其中，`enum` 是 C 语言中不限作用域的**枚举**（enumeration），其作用域不会受到定义它的“{ }”限制，在上一级作用域内仍然有效，因此语句 L1 是错误的。因为枚举 `color` 定义的 `black` 标识在语句 L1 处仍然有效，再定义 `black` 就属于重定义错误了。

C++ 11 标准对此进行了补充，可用 `enum class` 定义枚举，称为 `enumerator`，将其作用域限制在定义它的“{ }”中，因此语句 L2 定义 `black1` 时并不会与 `color1` 中的枚举常量 `black1` 发生冲突，后者的作用域被限定在定义它的“{ }”中，不会延伸到语句 L2 处。如果在程序中要引用 `enumerator` 中的枚举常量，就需要用枚举名进行限定，引用形式如下：

```
color1 c1 = color1::red1;
switch (c1) {
    case color1::black1:
        cout<<"black"<<endl;
        break;
    case color1::red1:
        .....
}
```

有了上面结构、枚举等的定义后，在 C++中可以用如下形式定义相关类型的变量：

```
student s1;
xy x1;
color col;
```

但在 C 语言中，结构和联合中是不允许定义函数的，并且必须在相关变量的定义前面加上对应的关键字，形式如下：

```
struct student s1;
union xy x1;
enum color col;
```

2. C++语言的类型定义

同 C 语言一样, C++语言也可以继续用 `typedef` 为已有类型定义一个容易阅读的描述性名称, 以提高代码的可读性, 同时 C++ 11 标准提出了用 `using` 定义数据类型。区别在于, `typedef` 具有“向下兼容”性, 支持早期的编译器, 但 `using` 不支持“向下兼容”, 只能在 C++ 11 标准以上的编译器中运行。两者的定义方法如下:

```
typedef type newname;
using newname = type;
```

其中, `type` 是已经存在的数据类型, `newname` 是为 `type` 指定的新类型名。新类型名 `newname` 并未取代原来的类型 `type`, 即 `type` 和 `newname` 在程序中都是可用的。例如:

```
typedef float house_price;           // L1
using house_price = float;          // L2, 本语句实现了与语句 L1 相同的功能
house_price x, y;                   // L3
float x, y;                         // L4, 本质上, 语句 L3 定义的 x、y 都是 float 类型
```

2.2 C++程序变量设计的基本思想

如果只学习过 C 语言之类的面向过程程序设计语言, 在学习 C++面向对象程序设计技术时, 首先应当转换设计程序变量的思维。先看一个具有代表性的简单例子。

【例 2-1】 如下程序在 C 中存在编译问题, 但在 C++中是正确的。

```
// Eg2-1.c
void main() {
    int x;           // L1
    x = 9;           // L2
    int y;           // L3
    y = x+1;         // L4
}
```

在 VC 6.0 中创建一个 C 程序项目, 并添加上述名为 `Eg2-1.c` 的代码。在编译该程序时, 产生了如下编译错误信息:

```
Compiling ...
Eg2-1.c
F:\cprogram\ui\ Eg2-1.c(7) : error C2143: syntax error : missing ';' before 'type'
F:\cprogram\ui\ Eg2-1.c(8) : error C2065: 'y' : undeclared identifier
```

如果在 VC 6.0 中将上面的程序名改为 `Eg2-1.cpp` (`.cpp` 是 C++程序), 再添加到项目中, 程序就能够正常编译执行。同样的代码为什么会出现两种编译结果呢? 这个问题实际上代表了面向过程和面向对象两种程序设计技术对待程序全局变量和局部变量的不同理念。

面向过程程序设计的基本思想是“程序=数据结构+算法”, 因此程序设计的基本方法是:

① 定义程序要用到的数据结构和全局变量; ② 定义操作数据和全局变量的若干函数, 定义函数时, 也是先定义好要使用的所有局部变量才开始编写函数执行代码; ③ 在主程序(主函

数)中组织执行流程,按次序调用各函数,进行全局变量的运算和修改,实现程序功能。

形式上,这种程序设计的逻辑结构如下:

```
struct A{ ... };           // 定义数据结构
int x, y, z;               // 定义全局变量
int f1(...) {
    int i, j, k;           // 定义局部变量
    for(i = 0; i < n; i++) { // 第 1 条执行语句开始后,就不允许再定义变量
        x += 10;
        .....
    }
}
int f1(...) { ... }
void main() {              // 主函数:组织程序执行流程,实现程序功能
    int a, b, c;           // 定义主函数中的局部变量
    struct A s;
    f1(a, b, c);           // 执行语句开始,不允许再定义变量
    .....
    f2(s);
}
```

C 语言就是按照这样的程序结构和变量设计思想进行程序设计的。在这种方式下,局部变量应该在函数的可执行语句之前定义,如果在第一条可执行语句之后再定义变量,就会产生编译错误。在例 2-1 中,语句 L3 是一条变量定义语句,但它前面的“x=9;”是一条执行语句,违背了“执行语句开始之后就不再定义变量的规则”,因此产生了编译错误。

面向对象程序的变量设计则完全不同,主张“**尽量减小变量的作用域范围,少用(甚至不用)全局变量,变量应就近定义,就近使用**”。主要原因是,面向对象程序中的变量通常是用结构复杂的类定义的对象,一个对象可能具有非常多的数据成员,会占用较多的内存空间。一个对象在完成了它的功能后,如果留存在内存中,只会浪费存储空间,应该尽早从内存中被清除出去。然而,全局变量在完成其功能后,即使再无任何用途,也会在程序运行期间一直保留在内存中,浪费着存储资源。同样,具有较长生存期的局部变量也存在类似问题。例如,在上述形式的逻辑结构中,如果函数 main()中的函数 f1()需要较长运行时间(如 5 分钟),那么在 f1()中并不会被用到的结构变量 s 在这 5 分钟对内存空间的占用就是浪费,所以在函数 f1()被调用后再定义 s 是更合理的做法。此外,“**变量就近定义,就近使用**”可以减少程序设计人员的负担,在编写代码行数较多的程序时,不用记住或查找前面定义的变量名,也不用担心变量重定义问题。

因此,面向对象程序设计语言允许在任何语句位置定义变量。C++也是如此,在程序中的任何语句位置,包括 for、while、do-while 循环语句内部以及 switch 和 if 等复合句中都可以定义变量。

【例 2-2】 在 C++中,在 for 循环内部定义局部变量。

```
// Eg2-2.cpp
#include <iostream>
using namespace std;

void main() {
    int n = 1;
```



```

    for(int i = 1; i <= 10; i++) {
        int k;
        n = n*i;
    }
    cout<<n<<i<<endl;
}

```

// i^[1]、k 的作用域至此结束
 // i 在此的值是 11
 // n、i 的作用域到此结束

在 C++ 中，变量在包含定义它的最近一对“{}”内有效，称为块作用域。因此，n 在整个函数 main() 内有效，而 i 和 k 仅在 for 循环体内有效，在 for 循环体外就无效了。

2.3 左值、右值和断言

左值 (lvalue) 是指可被寻址的数据（能找到该数据存放的内存地址），**右值 (rvalue)** 是指临时值或常量。简单地讲，左值是指放置在赋值语句左边的变量，右值则是放置在赋值语句右边的变量或表达式。例如：

```

int n = 10, x = 3, y;           // L1
y = 10 + x;                     // L2
n = n + y;                      // L3

```

在语句 L1 中，n、x、y 是左值，10 和 3 是右值；在语句 L2 中，y 是左值，10+x 是右值；在语句 L3 中，n 是左值，n+y 是右值。可以看出，左值都是变量，右值则是表达式、常量或变量。那么，在一条赋值语句的左、右两边都出现同一变量名时（如语句 L3 中的 n），该如何区别呢？

实际上，任何一个变量都包括两个要素：变量对应的内存区域和在此内存区域中存储的数据。变量对应的内存区域称为它的左值，内存区域中的内容则称为它的右值。当变量名出现在赋值语句左边时，使用它的左值，表示将右边表达式的计算结果写入该内存区域；当变量名出现在赋值句的右边时，使用它的右值，即读取对应内存区域中的数据。

概言之，左值是指内存区域，用变量名进行操作，凡是对变量的修改都是通过它的左值进行的，如给变量赋值、对变量进行自增、自减操作等。

断言 (assert) 是一种检测错误的宏，可以用来对表达式的结果进行判断，如果为假，就会退出程序。在软件开发阶段运用断言进行测试非常有效，能够快速找到错误并进行修改。

【例 2-3】 在 C++ 中，用断言检查平方根函数的参数必须大于 0。

```

// Eg2-3.cpp
1  #include <iostream>
2  #include <cassert>
3  #include <cmath>
4  using namespace std;
5
6  double sqrtdd(double x) {
7      assert(x >= 0.0);

```

[1] 在 C++ 语言规范中，i 的作用域到此就结束了。但某些早期编译器扩大了 for 循环中定义变量的作用域，将其有效范围扩展到了定义该变量的 for 循环后的“}”，如 Visual C++ 6.0。

```

8     return sqrt(x);
9 }
10 int main() {
11     cout<<sqrtd(-6.0)<<endl;
12     return 0;
13 }

```

编译并运行程序，将产生下面的输出。

```

Assertion failed: x >= 0.0, file F:\cprogram\ui\abc.cpp, line 8
abnormal program termination

```

这个结果表示程序出现了错误，根据提示信息可以快速定位到有错误的代码处，发现并修改错误，将“-6.0”改为“6.0”就对了。应用程序一般需要应用大量的测试代码来检测程序可能发生的错误，在测试（Debug）版中使用断言查找错误不失为一种好方法，当各种错误都得以正确处理后，在正式版中再删除或注释断言。或者用如下方式处理，在 `assert` 头文件包含的前面定义 `NDEBUG` 宏，让所有断言失效，以减少断言的系统开销。在将测试版软件转换成发行（Release）版时可以采用这种方法，从而提高转换效率。

```

#define    NDEBUG
#include <cassert>
.....

```

2.4 指针

2.4.1 指针概述

1. 指针的基本概念

指针用于存放一个对象在内存中的地址，从而间接地操作这个对象。指针的典型用法是建立链接的数据结构，如树（tree）和链表（list），或管理在程序运行过程中动态分配的内存空间，或用作函数参数以便传递数组或大型的类对象。指针的通用定义形式如下：

```
T *p;
```

其中，`T` 代表任意数据类型。这条形式语句实际确定了两处相关的内存位置：`p` 和 `*p`。

`p` 是指针变量本身，只能存储内存地址，其分配方法与普通变量的相同：都存储在栈（也叫堆栈）中，遵守同样的作用域和生存期规则，都可以进行赋值或复制。例如，如下语句

<code>double d, *pd;</code>	<code>// L1</code>	0x0018FF40	3.2	d
<code>int n = 0, *p;</code>	<code>// L2</code>	
<code>d = 3.2;</code>	<code>// L3</code>	
<code>p = &n;</code>	<code>// L4</code>	0x0018FF3C	0x0018FF40	pd
<code>pd = &d;</code>	<code>// L5</code>	
<code>*p = 10;</code>	<code>// L6</code>	0x0018FF38	0	n
		
		0x0018FF34	0x0018FF38	p
		

被复制到 VC 6.0 的函数 `main()` 中执行，可以调试观测到编译器为语句 L1、L2 定义的变量分配图 2-1 所示的内存区域，即堆栈（stack）。

图 2-1 VC 6.0 中建立的堆栈

在应用方面，指针变量与普通变量有以下区别：

- ① 指针变量 `p` 保存的内容是某内存单元的地址（内存单元的编号，具有相同的长度）。

由于每个内存单元地址的长度相同，因此指针变量需要的内存大小相同（取决于系统字长），与数据类型无关。在 VC 6.0 中，int 型指针 p 和 double 型指针 pd 都是 4 字节。但是，普通变量所占内存空间大小与类型相关。例如，双精度数 d 占 8 字节内存，而整型 n 占 4 字节内存，见图 2-1。

指针变量可用“&”运算符取某变量的地址，或用 malloc() 分配的内存地址为其赋值。例如，执行语句 L4、L5 后，p 和 pd 的值（n 和 d 的内存地址）见图 2-1。

② p 保存的内存地址，称为**指针所指的变量**，用“*p”表示对它的访问，也称为**指针解引用**。其实质是某内存区域的首地址，从而操控从该单元开始的一片连续内存区域，大小为定义指针时使用的数据类型的空间。因此，指针“p”与定义指针的数据类型无关，“*p”才有关。

在图 2-1 中，通过 p 的内容（&n，实际值是 0x0018FF38）可以找到内存单元“0x0018FF38”，并读写包括此单元在内的连续 4 字节的内容，这 4 字节正好是变量 n 的内存区域。因此，执行语句 L6 后，n 的值将改变为 10，如图 2-2 所示。

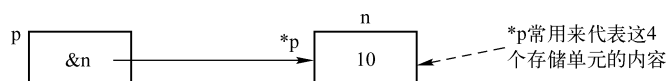


图 2-2 指针与其所指对象之间的关系

指针是一个复杂的概念，不同类型的指针能够指向（保存）不同类型变量的内存地址。

```

double d;
int *pi;           // pi 是指向 int 的指针
int **pc;          // pc 是指向 int 指针的指针
int *pA[10];       // pA 是指针数组，具有 10 个 int 类型的指针元素
int (*pa)[10];     // pa 是一个指针，指向具有 10 个 int 类型的元素的数组
int (*f)(int, char); // f 是指向具有两个参数的函数的指针
int *f(int);       // f 是一个函数，返回一个指向 int 的指针
  
```

定义指针时注意，指针的类型必须与它指向的对象类型一致，否则可能引发错误。例如，对上面定义的 pi 和 d，下面的赋值是错误的。

```

pi = &d;           // 错误，类型不匹配，只能把 int 类型的变量地址赋给 pi
  
```

2. 空指针与指针初始化

空指针是没有指向任何内存单元的指针，可以用 NULL、0、nullptr 赋值。

```

T *ptr = 0;
ptr = NULL;
ptr = nullptr;           // C++ 11
  
```

T 可以是任何一种数据类型，如 int、double 等。由于没有任何变量会被分配到地址 0，因此 0 可以作为一个指针常量，表明指针当时没有指向任何变量。NULL 是系统提供的一个预定义宏，定义为“#define NULL 0”。nullptr 是 C++ 11 引入的新值，可以被转换成任意类型的指针。

虽然并未要求指针在定义时必须初始化，但最好养成良好的指针初始化习惯。因为未初始化的指针的值是随机的（具体是什么值取决于所对应的内存单元），使用未初始化的指针可能导致各种程序问题，这也是 C++ 程序产生错误的主要根源。在旧版 C++ 中常用如下方式初始化指针：

```
int* p5 = 0;
int* p3 = NULL;
```

在 C++ 11 中，虽然仍然可用上述指针初始化方式，但并不提倡，而是用 `nullptr` 进行初始化。其原因是，虽然地址 0 一定不会被任何应用程序使用，代表空指针是安全的，但含义并不清晰（0 也可以代表整数 0），`NULL` 还有可能导致二义性。而作为指针的字面量，`nullptr` 可以赋值给任何类型的指针，也能够与任意的指针相比较，且不会与其他数据类型的 0 值相混淆。

在 C++ 11 标准中，建议用如下方式初始化指针：

```
int* p1 = nullptr;
int* p2{}; // 等价于 p2 = nullptr;
int* p3 = new int(0);
int* p4 = new int;
```

其中，`p1`、`p2` 的功能完全相同，将对应的指针初始化为空指针，`p3` 和 `p4` 则为指针分配了内存空间，`p3` 还将分配到的内存空间初始化为 0。

2.4.2 void*指针和获取数组首、尾元素位置的指针

1. void*

鉴于任何类型的指针变量保存的都是某内存单元的地址，所需存储空间的大小完全相同，C++ 提供了一种无类型指针 “`void*`” 来表示这个概念。用 “`void*`” 定义的变量，只表示它能够保存一个内存地址，与数据类型没有关系，可以接收任何数据类型（除了函数指针）的指针。而且，两个 `void*` 指针之间可以相互赋值、进行比较。但是，在使用 `void*` 指针前，必须显式地将它转换成某种数据类型的指针后，才能访问其所指内存区域的数据，其他操作都是不允许的。

【例 2-4】 `void*` 指针的应用。

```
// Eg2-4.cpp
#include <iostream>
using namespace std;

void main() {
    int i = 4, *pi = &i;
    void* pv;
    double d = 9, *pd = &d;
    pv = &i; // L1: 正确
    pv = pi; // L2: 正确
    // cout<<*pv<<endl; // L3: 错误
    pv = pd; // L4: 正确
    cout<<*(double*)pv; // L5: 正确，输出 9
}
```

因为 `pv` 是 `void*` 指针，无法确定 `*pv` 所指内存区域的大小和类型，所以无法访问。必须像语句 L5 那样经过强制类型转换后才能访问。

`void*` 最重要的用途是作为函数的形式参数或返回类型，以便向函数传递或返回类型可变的对象，使用时再将它显式转换成适当的类型，从而设计出功能强大的函数。

2. 指针、数组及位置获取函数 `begin()`和 `end()`^{C++ 11}

数组是在编译期就必须确定元素个数的批量存储空间分配技术，如

```
int a[10], b[] = {1,2,3,4,5,6,7,8,9,0};
int n;
cin>>n;
int c[n]; // 错误
```

数组 `c` 的定义是错误的，因为数组大小需要在编译时确定，但 `n` 只有在程序运行时才能够通过键盘输入获取，所以是错误的。解决办法是，按最大需求设置数组大小，这就可能产生内存空间的浪费。如运行程序 10 次，只有 1 次需要 10000 元素的 `c` 数组，其余 9 次都只需要小于 100 个元素的 `c` 数组，但为了满足那 1 次的需求，必须将 `c` 数组的大小定义为 10000。有没有更合理的解决方案呢？那就是用指针创建动态数组，每次都按实际需要分配数组大小。

```
int n;
cin>>n;
int *c = new int[n];
for(int i = 0; i < n; i++)
    c[i] = i*10; // c 本质上与数组相同，可用与数组相同的方法访问
```

指针和数组在多数情况下是可以互操作的，可以用指针处理数组，也可以用数组方式处理指针。在向函数传递参数时，指针可以传递给数组参数，数组也可以传递给指针参数。其原因是当用数组作为函数参数时，编译器实际上是用指针实现数组参数的。例如，对于

```
int f(int a[], int size);
```

C++（包括 C 语言）在编译时将函数 `f()` 转换成如下形式，

```
int f(int *, int size);
```

这也是为什么无法直接定义函数参数数组大小的原因，即“`int f(int a[10])`”这样的函数定义将被编译器转换为“`int f(int *a)`”。因此，必须将数组大小也定义成函数的参数才能传递。

为了方便以指针方式读写数组的内容，C++ 11 标准在 `iterator` 头文件中提供了两个标准库函数 `begin()`和 `end()`，用于确定指向数组第一个元素和最后元素后一位置的指针，为遍历数组提供方便。其用法如下：

```
begin(a)
end(a)
```

其中，`a` 是数组名，`begin(a)`返回指向 `a[0]`的指针，`end(a)`返回指向数组的最后元素后一位置的指针。

如下代码用 `begin()`和 `end()`遍历数组，输出数组元素，非常方便。

```
int a[] = {1,2,3,4,5,6,7,8,9,10};
for(int *p = begin(a); p != end(a); p++)
    cout<<*p <<" ";
cout << endl;
```

2.4.3 内存的分配和释放

指针常与堆空间的分配有关。所谓堆（`heap`），就是一块内存区域，允许程序在运行期间以指针的方式从中申请一定数量的存储空间（其他存储空间的分配是在编译时完成的，称为

栈或堆栈)，用于程序数据的处理。堆内存也称为动态内存。

堆内存的管理由程序员完成。在 C 语言中，如果需要使用堆内存，程序员可以用函数 `malloc()` 从堆中分配指定大小的存储区域，用完之后必须用函数 `free()` 将之释放。如果用完之后没有用函数 `free()` 将它释放，就会造成内存泄漏（[memory leak](#)，自己不用了，其他程序也无法使用）。因此，函数 `malloc()` 和 `free()` 在 C 程序中总是成对出现的。例如：

```
#include <stdlib.h>                                // malloc()和 free()定义于此头文件中

void main() {
    int *p;
    p = (int*)malloc(sizeof(int));    // 从堆中分配 4 字节 (int 的大小) 并转换为 int 类型
    *p = 23;
    free(p);                            // 释放堆内存
}
```

`malloc()` 的使用比较麻烦，除了要计算需要的内存大小，还必须对获得的内存区域进行类型转换才能使用。为此，C++ 提供了 `new` 和 `delete` 两个运算符进行堆内存的分配和释放，它们分别与 `malloc()` 和 `free()` 相对应，但使用起来更简单。

1. new 的用法

`new` 的功能类似于 `malloc()`，用于从堆中分配指定大小的内存区域，并返回获得的内存区域的首地址。对于 “`type *p;`” 定义的指针 `p`，下面 3 种赋值方法都是正确的：

```
p = new type;                // 用法 1
p = new type(x);              // 用法 2
p = new type[n];              // 用法 3，分配数组空间
```

其中，`type` 代表任意数据类型。用法 1 只分配堆内存，用法 2 将分配到的堆内存初始化为 `x`，用法 3 分配具有 `n` 个元素的数组空间。

`new` 能够根据 `type` 自动计算分配的内存大小，不需要用函数 `sizeof()` 计算。若分配成功，将得到的堆内存的首地址存放在指针变量 `p` 中，否则返回空指针（`0`）。在程序中，可以用 `0` 作为判断内存分配成功与否的依据。

2. delete 的用法

`delete` 的功能类似于函数 `free()`，用于释放 `new` 分配的内存，以便它被其他程序使用。

```
delete p;                    // 用法 1
delete []p;                  // 用法 2，释放数组空间
```

其中，`p` 是用 `new` 分配的堆空间指针变量。用法 1 用于释放动态分配的单个指针变量，用法 2 用于释放动态分配的数组存储区域。

`new` 和 `delete` 必须成对使用，否则就会产生内存泄漏。例如：

```
int *a = new int[100], b = 10;
a = &b;
```

执行 “`a = &b;`” 语句后，就会产生内存泄漏，分配给指针 `a` 的 `100×4B` 空间无法访问，也无法释放，在程序运行过程中相当于这部分内存丢失了，直到程序结束后，操作系统才会释放它。

【例 2-5】 用 `new` 和 `delete` 分配、释放堆内存。

```

// Eg2-5.cpp
#include <iostream>
using namespace std;

int main() {
    int *p1, *p2, *p3;
    p1 = new int; // 分配一个能够存放 int 类型数据的内存区域
    p2 = new int(10); // 分配一个 int 类型大小的内存区域, 并将 10 存入其中
    p3 = new int[10]; // 分配能够存放 10 个整数的数组区域
    if(!p3) { // 程序中常会见到这样的判定
        cout<<"allocation failure"<<endl; // 分配不成功, 就显示错误信息
        return 1; // 终止程序, 并返回错误代码
    }
    *p1 = 5;
    *p3 = 1;
    p3[1] = 2; // 访问指向数组的数组元素
    p3[2] = 3;
    cout<<"p1 address: "<<p1<<" value: "<<*p1<<endl;
    cout<<"p2 address: "<<p2<<" value: "<<*p2<<endl;
    cout<<"p3[0] address: "<<p3<<" value: "<<*p3<<endl;
    cout<<"p3[1] address: "<<&p3[1]<<" value: "<<p3[1]<<endl;
    delete p1; // 释放 p1 指向的内存
    delete p2;
    delete p3; // 错误, 只释放了 p3 指向数组的第 1 个元素
    // delete []p3; // 正确, 释放 p3 指向的数组
    return 0;
}

```

delete p3 与 delete []p3 是有区别的, 前者只释放了第一个数组元素 (p[0]) 的内存, 而没有释放其余数组元素 (p[1]~p[9]) 所占的内存空间, 会造成内存泄漏; 后者则将 p3 指向的数组所占用的全部内存空间均归还系统。

3. new、delete 与函数 malloc()、free()的区别

在 C++ 程序中, 仍然可以使用函数 malloc()、free() 进行动态存储空间的管理, 但它们没有 new、delete 方便。以下是函数 malloc() 和 free() 不具备的功能: ① new 能够自动计算要分配的内存大小, 不必用 sizeof() 计算所要分配的内存字节数, 减少了出错的可能性; ② new 不需要进行类型转换, 能够自动返回正确的指针类型; ③ new 可以对分配的内存进行初始化; ④ new 和 delete 可以被重载, 程序员可以借此扩展 new 和 delete 的功能, 建立自定义的存储分配系统。

2.4.4 智能指针 ^{C++ 11}

动态内存分配 (堆内存) 是 C++ 程序最容易出错的地方, 有时会忘记使用 delete 或调用函数 free() 释放为指针分配的堆内存, 造成内存泄漏。C++ 新标准的理念之一是, 不再使用 delete 或调用函数 free() 手动释放堆内存, 而是用智能指针对象管理堆内存。用智能指针进行堆内存分配和使用的方法与普通指针差不多, 主要区别是智能指针会自动释放所引用的堆内存, 不需像使用 delete 或调用函数 free() 那样进行显式的内存回收。

C++ 03 标准引入的智能指针是 `auto_ptr`，但是存在一些问题，现已基本不用。C++ 11 标准引入了 `unique_ptr`、`shared_ptr` 和 `weak_ptr` 三种智能指针取代 `auto_ptr`^[2]。智能指针是用模板设计（见第 7 章）的，定义在 `memory` 头文件中，包括 `auto_ptr`、`unique_ptr` 和 `shared_ptr` 等，其定义形式如下：

```
x_ptr<type> p; // L1
x_ptr<type> p2(p); // L2, 适用于 auto_ptr, share_ptr
x_ptr<type> p3(new type(x)) // L3
```

`x_ptr` 代表智能指针，可以是 `auto_ptr`、`shared_ptr` 或 `unique_ptr`；`type` 可以是任何数据类型，包括 `class` 类型（见第 3 章）。

语句 L1 定义了可以指向 `type` 类型对象的空智能指针 `p`；语句 L2 定义了指向 `type` 类型的指针 `p2`，并用已定义的 `p` 对它进行初始化，即 `p2` 复制了 `p` 的内容；`p3` 定义了指向 `type` 类型的智能指针，并用 `new` 为它分配了动态内存，且用 `x` 初始化了该内存（不是必须的）。

可以像普通指针一样操作智能指针。例如，在上述定义中，`p` 表示指针本身，`*p` 表示指向的对象，`p->member` 和 `(*p).member` 表示指向对象的成员 `member`。智能指针还有两个常用成员函数 `get()` 和 `swap()`。`p.get()` 能够返回 `p` 中保存的指针，需要小心使用，若智能指针释放了所指向的对象，则指针指向的对象也销毁了；`p.swap(p2)` 则交换 `p` 与 `p2` 指针的内容。

智能指针的赋值方式与普通指针略有差异，可以在定义时就为它分配动态存储空间，但不允许先定义智能指针，再为它分配动态存储空间。类似如下形式：

```
x_ptr<type> p1, p2(new type); // 正确
p1 = new type; // 错误
p1 = p2;
```

同类型的 `auto_ptr`、`shared_ptr` 智能指针之间可以相互赋值，`unique_ptr` 指针之间则不允许相互赋值。比如，对于“`p1 = p2;`”，若定义 `p1`、`p2` 的 `x_ptr` 是 `auto_ptr` 或 `shared_ptr` 类型，则该语句就是正确的；若 `x_ptr` 是 `unique_ptr`，则是错误的。

智能指针与普通指针之间不能够随意赋值，不能把智能指针指向普通内存变量，或者把非智能指针赋值给智能指针。直接把智能指针赋值给普通指针是错误的，要通过智能指针的成员函数 `get()` 获取智能指针中的指针后，再赋值给普通指针。例如：

```
int x = 9;
int *ip = new int(1);
shared_ptr<int> sp(new int(8));
// sp = &x; // 错误
// sp = ip; // 错误
// ip = sp; // 错误
ip = sp.get(); // 正确
```

把定义 `sp` 的 `shared_ptr` 换成 `auto_ptr`，情况完全相同。

1. 独占指针（`unique_ptr`）

独占指针 独占它所分配到的内存空间使用权，指针之间不允许相互赋值，也不允许用一个独占指针初始化另一个独占指针，否则在编译时会发生错误。例如：

[2] `auto_ptr` 采用移动语义（move）实现其复制和赋值操作，易被误用。C++ 17 标准已取消了 `auto_ptr` 指针。

```

unique_ptr<string> p1(new string("auto"));           // 初始化方式 1
unique_ptr<string> p2;
unique_ptr<int> p3 = make_unique<int>(123);         // 初始化方式 2, C++ 14
unique_ptr<string> p4(p1);                          // 错误
p2 = p1;                                             // 错误

```

不允许为 p2 赋值，如何才能为它赋值呢？可以通过函数 `reset()` 或 `move()`，即

```

p2.reset(new string("hello"));
p2 = std::move(p1);                                // move() 是 std 命名空间中的移动函数

```

函数 `make_unique()` 是 C++ 14 才为独占智能指针提供的创建函数，但 C++ 11 就为共享指针提出了创建函数 `make_share()`。在程序开发过程中，应当尽量多用这两个函数初始化智能指针，因为它们的效率更高，原因在下节的“共享指针”中介绍。

“移动”是 C++ 11 提出的资源移动概念。所谓**移动** (`move`)，即转移内存空间的使用权，如同现实生活中房屋产权的转换一样，张三将自己的房屋产权转交给李四后，该房屋就只能被李四支配，张三再无这套房。这里的概念是一样的，p1 将所指内存区域转移给 p2 后，就变成了 `nullptr` 指针，p2 则全权拥有“auto”所在的内存空间，会负责该内存空间的使用和释放。

移动操作通常用于从函数中返回智能指针。对于 `unique_ptr` 而言，移动操作只能在同类型的 `unique_ptr` 之间进行，如下例所示。

【例 2-6】 用 `unique_ptr` 指针返回输入函数 `InputStudent()` 创建的学生对象。

```

// Eg2-6.cpp
#include<iostream>
#include<memory>
using namespace std;

struct Student {
    string name;
    int age;
};

unique_ptr<Student> InputStudent() {
    unique_ptr<Student> stu(new Student);
    cout<<"input name, age"<<endl;
    cin>>stu->name;
    cin>>stu->age;
    return stu;
}

int main() {
    unique_ptr<Student> ptr_s;           // L1
    ptr_s = move(InputStudent());        // L2, 可不写 move
    cout<<"name: "<<ptr_s->name<<"\nAge: "<<ptr_s->age<<endl;
    unique_ptr<int []> pi(new int[10]); // L3, 分配一组堆对象
    for (int i = 0; i < 10; i++)         // L4
        pi[i] = i*10;
    for (int i = 0; i < 10; i++)         // L5
        cout<<pi[i]<<"\t";
    cout<<endl;
}

```

语句 L1 定义了独占指针 `ptr_s`，语句 L3 在堆空间中建立了具有 10 个元素的数组并用 `pi` 指针指向该数组，语句 L4 和 L5 展示了 `unique_ptr` 读写批量数组元素的方法。

语句 L2 中的 `move()` 可以省略，其语句等价于“`ptr_s = InputStudent();`”，因为 `InputStudent()` 函数返回的 `stu` 本身是个临时变量，符合“资源移动”的语义，所以 C++ 编译器会自动对它执行移动操作。

函数 `InputStudent()` 和函数 `main()` 都用 `new` 分配了堆内存，但没有在程序的任何语句位置用 `delete` 回收空间。这是没有问题的，智能指针 `ptr_s` 和 `pi` 会在程序结束时自动释放它所占用的内存空间。相反，如果定义 `stu` 和 `pi` 用的是裸指针，即“`Student *stu = new Student;`”，或者“`int *p = new int[10];`”，本程序就会产生内存泄漏。

2. 共享指针 (shared_ptr)

共享指针即多个指针可以指向同一个对象，主要用来管理多个用户程序共同访问的内存区域，适用于网络中的并发程序设计。同类型的 `shared_ptr` 指针之间可以相互赋值，也可以用一个 `shared_ptr` 指针去初始化另一个 `shared_ptr` 指针。

`shared_ptr` 是一种采用了引用计数的智能指针，关联了一个计数器，其中保存着指向同一个内存对象的指针（包括非智能的指针）数。当有指针要使用该对象时，并不会进行对象复制或重新分配内存的操作，而是让该指针直接获取该对象的地址，并让计数器增 1。比如，如果对指针进行复制，或进行指针之间的赋值，或用该指针初始化另一个指针，或向函数传递指针参数，等等，都会使计数器加 1。反之，当一个指针离开所指对象时，就会减少该对象的引用计数，当计数器为 0 时，就会销毁该对象。`shared_ptr` 有一个成员函数 `use_count()`，可以查看共享指针的计数器。

【例 2-7】 用智能指针 `shared_ptr` 实现两数交换，并查看共享指针的数量。

```
// Eg2-7.cpp
#include <iostream>
#include <memory>
using namespace std;

void swap(shared_ptr<int> a, shared_ptr<int> b) {
    int t;
    t = *a;
    *a = *b;
    *b = t;
}

void main() {
    shared_ptr<int> p1(new int(9));
    cout<<"1: p1.count = "<<p1.use_count()<<endl;
    shared_ptr<int> p2(p1);
    cout<<"2: p1.count = "<<p1.use_count()<<endl;
    shared_ptr<int> p3(new(int)), p4(new int(8));
    {
        shared_ptr<int> p5{p1};
        cout<<"3: p1.count = "<<p1.use_count()<<endl;
    }
    cout<<"4: p1.count = "<<p1.use_count()<<endl;
```

```

cout<<"p1 = "<<*p1<<"\tp4 = "<<*p4<<endl;
swap(p1, p4);
cout<<"p1 = " <<*p1<<"\tp4 = "<<*p4<<endl;
p3 = p4 = p1;
cout<<"5: p1.count = "<<p1.use_count()<<endl;
}

```

运行结果如下：

```

1: p1.count = 1
2: p1.count = 2
3: p1.count = 3
4: p1.count = 2
p1 = 9    p4 = 8
p1 = 8    p4 = 9
5: p1.count = 4

```

函数 `swap()` 接受两个 `shared_ptr` 类型的智能指针参数，实现了两数的交换。请读者结合程序分析语句 `p1.count` 的 5 个输出计数值。本例用 `new` 多次分配了动态内存空间，但并未用 `delete` 回收这些内存区域，智能指针 `shared_ptr` 会自动回收它们，不会造成内存泄漏。

C++ 11 提供了函数 `make_shared()` 来创建 `share_ptr` 指针，将指针管理和数据放置在同一内存区域，具有更高的访问效率，应尽量用它来创建共享指针。例如：

```

shared_ptr<int>p1{ new int }, p2 = p1;
shared_ptr<int>p3 = make_shared<int>(), p4 = make_shared<int>();

```

共享指针 `p1` 和 `p2`、`p3` 和 `p4` 的共享内存结构分别如图 2-3 和图 2-4 所示，前者将指针管理数和业务数据分散在两处存放，后者则将两类数据合存一处，具有更高的访问效率。

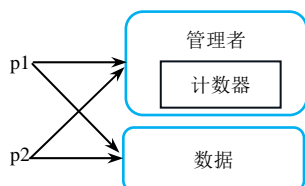


图 2-3 p1 和 p2 指针的内存共享结构

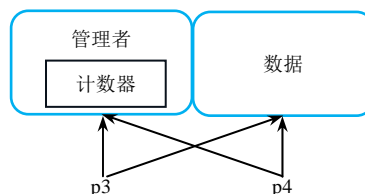


图 2-4 p3 和 p4 指针的内存共享结构

3. 弱指针 (weak_ptr)

弱指针是一个不控制资源生命周期的智能指针，是对对象的一种弱引用，目的是协助共享指针工作。弱指针并不占有内存，因此没有指针的解引用操作，但是能够提供对一个或多个共享指针拥有的对象的访问，且不参与引用计数，可以用来解决两个共享指针相互引用产生的死锁问题（见 3.7.2 节）。

`weak_ptr` 可以由一个 `shared_ptr` 或另一个 `weak_ptr` 构造，也可以直接把 `shared_ptr` 赋值给 `weak_ptr`。还可以通过 `weak_ptr` 的 `lock()` 成员函数获取对应的 `shared_ptr`，作用是把 `weak_ptr` 绑定到对应 `share_ptr` 的内存。构造和析构 `weak_ptr` 指针时不会引起对应 `shared_ptr` 计数器的增加和减少。

【例 2-8】 弱指针示例。

```

// Eg2-8.cpp
#include <iostream>

```

```

#include <memory>
using namespace std;

void swap(shared_ptr<int> a, shared_ptr<int> b) {
    int t = *a;
    *a = *b;
    *b = t;
}

int main() {
    shared_ptr<int> sp1(new int(9)), sp2(new int(7)), sp3(sp1);
    cout<<"1: sp1.count = "<<sp1.use_count()<<endl;           // L1
    weak_ptr<int> w1(sp1), w2(sp2);
    cout<<"2: w1.count = "<<w1.use_count()<<"\tsp1.count = "<<sp1.use_count()<<endl; // L2
    cout<<"sp1 = "<<*sp1<<endl;
    // cout<<"w1 = "<<*w1<<endl;                               // L3, 错误, weak_ptr 没有解引用
    // swap(w1, w2);                                           // L4
    swap(w1.lock(), w2.lock());                                // L5
    cout<<"*sp1 = "<<*sp1<<"\t*sp2 = "<<*sp2<<endl;           // L6
    return 0;
}

```

程序运行结果如下：

```

1: sp1.count = 2
2: w1.count = 2   sp1.count = 2
sp1 = 9
*sp1 = 7  *sp2 = 9

```

用 `new` 创建 `sp1` 时，`sp1` 的共享计数器为 1，当用 `sp1` 初始化 `sp3` 时，`sp3` 指向了 `sp1` 相同的堆内存，因此语句 L1 输出 `sp1` 的计数器为 2；`w1`、`w2` 分别与 `sp1`、`sp2` 共享相同的内存，但它们是弱指针，因此不会增加引起 `sp1`、`sp2` 共享内存的计数器增加，语句 L2 输出的计数器仍然是 2。

弱指针只能辅助共享指针工作，没有资源控制权，没有解引用，不能够操作内存，所以语句 L3 和 L4 是错误的。但是，弱指针有一个成员函数 `lock()`，可以绑定到对应共享指针的内存，进而操作该内存。语句 L5 利用 `lock()` 进行了 `w1`、`w2` 与 `sp1`、`sp2` 所指共享内存的绑定，实现了对共享内存的数据交换。

2.5 引用

任何变量都具有左值和右值两个要素，左值对应变量的内存区域，右值对应保存在变量内存区域中的值。以前只允许给变量的左值定义别名，称为引用。但在 C++ 11 标准中，除了可以为左值指定别名，也可以给表达式、常量和变量的右值定义别名，称为右值引用。自然，左值的别名也就称为左值引用了，由于习惯原因，仍称之为引用。

2.5.1 左值引用

左值引用是某个对象（变量）的别名，即某对象的第二名称，俗称引用，是 C++ 引入的

新概念，在 C 语言中并没有。引用由符号 “&” 引导定义，形式如下：

类型 &引用名 = 变量名；

例如：

```
int i = 9; // L1
int &ir = i; // L2
```

语句 L2 定义 ir 为 i 的别名，相当于 i 还有一个名称叫 ir，对 ir 的操作就是对 i 的操作。

【例 2-9】 引用的简单例子，ir 是 i 的引用，它们是同一内存区域的两个名称。

```
// Eg2-9.cpp
#include <iostream>
using namespace std;

void main() {
    int i = 9;
    int &ir = i;
    cout<<"i = "<<i<<"    "<<"ir = "<<ir<<endl;
    ir = 20;
    cout<<"i = "<<i<<"    "<<"ir = "<<ir<<endl;
    i = 12;
    cout<<"i = "<<i<<"    "<<"ir = "<<ir<<endl;
    cout<<"i 的地址是: "<<&i<<endl;
    cout<<"ir 的地址是: "<<&ir<<endl;
}
```

本程序的运行结果如下：

```
i = 9    ir = 9
i = 20    ir = 20
i = 12    ir = 12
i 的地址是: 0029FDB0
ir 的地址是: 0029FDB0
```

从结果可以看出，ir 和 i 其实是同一内存变量，对 ir 的操作实际就是对 i 的操作。

使用引用时需要注意以下问题。

① 在定义引用时，“&” 在类型和引用名之间的位置是灵活的，以下定义完全相同。

```
int& ir = i;
int & ir = i;
int &ir = i;
```

② 在变量声明时出现的 “&” 才是引用运算符（包括函数参数声明和函数返回类型的声明），其他地方出现的 “&” 则是地址操作符。例如：

```
int i;
int &r = i; // 引用
int& f(int &i1, int &); // 引用参数，函数返回引用
int *p = &i; // &取 i 的地址
cout<<&p; // &取 p 的地址
```

③ 引用必须在定义时初始化，不能在定义完成后再给它赋值；为引用提供的初始值可以是一个变量名，也可以是另一个引用名；可以为同一个变量定义多个引用。例如：

```
float f; // L1
```



```

float &fr;                                // L2, 错误, fr 未初始化
float &r1 = f;                             // L3
float &r2 = f;                             // L4
float &r3 = r1;                           // L5

```

r1、r2、r3 都是 f 的别名，对它们的任何运算都是对 f 的运算。

④ 引用对应变量的左值，代表变量的内存区域，是一种隐式指针，但与指针存在区别。

【例 2-10】 引用与指针的区别。

```

// Eg2-10.cpp
void main() {
    int i = 9;                                // L1
    int *pi = &i;                             // L2, &为取地址运算
    int &ir = i;                              // L3, &定义引用
    *pi = 2;                                  // L4
    ir = 8;                                   // L5
}

```

语句 L1、L2、L3 定义的变量如图 2-5 所示。pi 是指针，ir 是引用。语句 L4 是指针的解引用形式，把 i 对应的内存值改为 2。语句 L5 是引用的使用形式，把 i 对应的内存值改为 8。

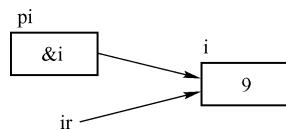


图 2-5 引用及其所引对象之间的关系

虽然引用实质上也是一种指针，但与指针至少存在两点区别：

- ❖ 指针必须通过解引用运算符“*”才能访问它所指向的内存单元，而引用与普通变量的访问方法相同。
- ❖ 指针是一个变量，有自己独立的内存区域，可以重新被赋值，以指向其他地址。但引用只是某变量的别名，甚至没有自己独立的内存区域，必须在定义时进行初始化，并且一经定义就再也不能作为其他变量的引用了。

⑤ 用运算符“&”获取一个引用的地址时，实际取出的是引用对应的变量的地址。例如：

```

int i = 9;
int &ir = i;
int *pi = &ir;

```

pi 实际指向的是 i，因为 ir 是 i 的别名，所以 &ir 将获得 i 的内存地址。

⑥ 建立引用时，引用应当类型匹配。例如：

```

double d;
int &rd = d;                                // 错误，引用与它对应的变量类型不一致

```

⑦ 引用与数组。可以建立数组或数组元素的引用，但不能建立引用数组。例如：

```

int i = 0, a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, *b[10];
int (&ra)[10] = a;                          // L1: 正确，ra 是具有 10 元素的整型数组的引用
int &aa = a[0];                             // L2: 正确，数组元素的引用
int *(&rpa)[10] = b;                         // L3: 正确，rpa 是具有 10 个整型指针的数组的引用
int &ia[10] = a;                            // L4: 错误，ia 是引用数组，每个数组元素都是引用
ra[3] = 0;                                  // L5: 正确，数组引用的用法
rpa[3] = &i;                                // L6: 正确

```

在鉴别引用与数组的关系时，记住一个原则：引用不分配内存，一次只能为一个已有变量定义一个别名，而数组一次需要定义多个元素，所以不能定义引用数组。

语句 L4 的含义是：定义 ia 为具有 10 个元素的数组，其中每个元素都是一个引用，这就要求一次性定义 10 个引用名，是不允许的。

语句 L1 的含义是：“(&ra)”指定 ra 是一个引用，“(&ra) [10]”指定 ra 是具有 10 元素的数组的引用，“int (&ra)[10]”指定数组的 10 个元素都应该是 int 类型。最后确定 ra 是 b 的引用，b 正好是具有 10 个 int 元素的数组，符合 ra 的要求。语句 L3 定义的 rpa 与此相同，只不过要求数组的元素是指针罢了。注意语句 L1、L3 和 L4 的区别，其含义是不同的。

数组引用的使用方法与普通数组相同，可以通过下标变量访问元素值，如 L5 和 L6 所示。

⑧ 引用与指针。可以建立指针的引用，但不能创建指向引用的指针。例如：

```
int i = 0, a[10];
int &*ip = i;           // L1: 错误，ip 是指向引用的指针
int *pi = &i;
int *&pr = pi;         // L2: 正确，pr 是指针的引用
```

语句 L1 的含义是：“*ip”定义了 ip 是一个指针，“&*ip”定义 ip 指向的是一个引用，这就不对了，因为引用是已定义变量的别名，它不分配内存，指向引用就无法确定指到哪里了，因此不允许这种用法。语句 L2 定义 pr 是引用，是一个指向整型数据的指针 pi 的引用。

在 C++ 中，引用主要用来定义函数的参数和返回类型。因为引用只需要传递一个对象的地址，在传递大型对象的函数参数或从函数返回大型对象时，可以提高效率。

2.5.2 右值引用、移动及其语义 C++ 11

复制大量数据是一件高成本的操作，复制临时值更是会带来不必要的开销，一些软件常采用“浅拷贝”技术（只复制数据的内存地址而不是复制数据本身）来避免批量数据复制，提高程序效率。C++ 11 标准增加了右值引用，以“资源移动”而不是复制的方式解决临时值的复制问题，基本思想是：所有变量（或说具名项）都会被深复制，所有临时值（或说不具名项）则只需要转移它的数据。

右值引用就是绑定到右值上的引用，用“&&”进行定义，形式如下：

类型 &&引用名 = 表达式；

例如：

```
double r = 10;
double &lr1 = r;           // L1, 正确，变量名代表左值
double &lr2 = r+10;        // L2, 错误，引用只能是变量
double &&rr = r;           // L3, 错误，变量名代表左值，而&&需要右值
double &&rr = r+10;        // L4, 正确，rr 为表“r+10”计算结果，即 20
```

需要注意的是，任何具名对象的变量名都是左值，只能够绑定在左值引用上，不能够绑定到右值引用上。例如，在上述代码中，r 是一个具名对象，是一个左值，可以绑定到左值引用上，因此语句 L1 是正确的；语句 L3 企图将 r 绑定到右值引用 rr 上，因此是错误的。

如果需要将具有名称的变量（左值）绑定到右值引用，可以用 std 命名空间中的移动函数 move() 实现。实际上，函数 move() 本身并没有“移动”数据，它只是将左值转换成一个右值。也就是说，它将移动的变量认定为临时量，然后将它与右值绑定。例如，语句 L3 可用如下形

式将 `r` 对应内存中的值绑定到右值引用 `rr`。

```
double &&rr = std::move(r);
```

右值引用是 C++ 11 为了支持移动操作而引入的新型引用类型，主要用来将引用绑定到即将销毁的临时对象上（如常量或表达式），相当于为没有名称的临时对象定义了一个名称，并延长了它的生命期。这种方式是直接使用临时变量，而不是复制它，可以节省数据复制的系统开销。

【例 2-11】 右值引用的定义和使用。

```
// Eg2-11.cpp
#include <iostream>
using namespace std;

void main() {
    int x = 10;
    int &r = x;
    // int &&ar = x; // L1: 错误，变量名只能被绑定到左值
    int &&rx = x + 10 * 3; // L2: 正确，rx 为表达式结果值的内存
    cout<<"x = "<<x<<"\trx = "<<rx<<endl; // L3
    x = 20;
    cout<<"x = "<<x<<"\trx = "<<rx<<endl; // L4
    int y = rx; // L5
    cout<<"y = "<<y<<endl; // L6
}
```

程序运行结果如下：

```
X = 10  rx = 40
X = 20  rx = 40 // 修改 x 对右值引用 rx 无影响
Y = 40
```

语句 L1 错误的原因是企图定义 `ar` 为变量 `x` 的右值引用，但变量名对应变量的左值，只能绑定为左值引用。“`int &&xx=std::move(x)`”语句可以实现对 `x` 右值引用的定义。

语句 L2 指定 `rx` 为表达式“`x + 10 * 3`”的右值引用。那么，`rx` 到底对应哪个对象呢？上面说的“右值引用只能绑定到常量或即将被销毁的对象上”，又是哪个对象呢？答案是系统为了保存表达式“`x + 10 * 3`”的计算结果而创建的无名临时对象。该对象只有短暂的生存期，在它的值被取用后（该表达式对应的语句执行完毕，下一条语句执行之前），马上会被销毁。右值引用 `rx` 绑定的就是这个对象，过程如下。

<1> 计算表达式“`x + 10 * 3`”的值，结果为 40，系统将创建 `int` 类型的无名对象，并将 40 保存在该对象中。

<2> 指定 `rx` 为该无名对象的右值引用，即无名对象的别名。假设没有右值引用，该操作应该是“使用无名对象的值 40，用完后马上销毁该无名对象，收回它的内存区域”。如果以后要再次使用“`x + 10 * 3`”，每次必须重新计算。

<3> `rx` 遵守变量的作用域和生存期规则，在定义它的函数 `main()` 作用域内有效，相当于延长了无名对象的生存期。

当要再次应用表达式“`x + 10 * 3`”时，直接用 `rx` 即可，不必再次计算。这样既可以节省运算该表达式使用的计算资源，也可以把右值引用传递给其他对象，或者作为函数的参数传

递，就像本例将 `rx` 传递给 `y` 一样，提高了程序效率。

注意：右值引用和左值引用都是引用，所绑定的都是变量的内存地址。可用“左值持久、右值短暂”进行分辨，即左值绑定的是具有较长生命期的变量名（对应变量的内存区域）；右值只能绑定到常量，或者表达式求值过程中创建的无名临时对象上（右值实际绑定到了临时对象的内存地址），本来这类对象用完就会被销毁，其生命期是短暂的，而右值引用“接管”了该临时对象的内存区域，延长了它的生命期，使它可再次被使用。

2.6 `const`和`constexpr`常量

2.6.1 常量的定义

变量实质上是在程序运行过程中其值可以改变的内存单元的名称。常量是在程序执行过程中其值固定不变的内存单元的名称，在 C++ 中常用 `const` 或 `constexpr` 定义，方法如下：

```
const  常量类型  常量名 = 常量表达式;
constexpr 常量类型  常量名 = 常量表达式;           // C++ 11
```

例如：

```
const int i = 10;           // L1
const char c = 'A';        // L2
const char s[] = "C++ const !"; // L3
constexpr char c = 'A';    // L4
constexpr char s[] = "C++ const "; // L5
```

语句 L1 定义了常量 `i`，语句 L2 定义了字符常量 `c`，语句 L3 定义了字符常量数组 `s`。语句 L4 与 L2、L5 和 L3 的功能相同。

① `const` 和 `constexpr` 常量必须在定义时初始化，且常量一经定义就不能修改（常量名不能够出现在赋值符“=”的左边）。例如：

```
const int n;           // 错误，常量 n 未被初始化
const int i = 5;       // 定义常量 i
i = 10;                // 错误，修改常量
i++;                   // 错误，修改常量
```

② 在 C++ 中，表达式可以出现在常量定义语句中。例如：

```
int j, k = 9;           // L1
const int i1 = 10+k+6;  // L2, i1 为 25
const int i2 = j+10;    // L3, i2 未知，因为 j 不确定
```

当常量定义语句中出现表达式时，C++ 将首先计算表达式的值，然后把计算结果指定给常量。语句 L3 尽管在有的编译环境下不会出错误，但没有意义，因为 `j` 的取值是未知的。

③ `constexpr` 与 `const` 的功能基本相同，都用于定义常量，但存在区别。

`constexpr` 常量称为**编译期常量**，必须在编译期对其定义的常量进行初始化。因此，只接受在编译器就能够明确确定的值，用于初始化 `constexpr` 常量的表达式中的每部分值都是程序运行前就可以确定的字面值常量。但是，返回 `constexpr` 类型的函数是可以接受程序执行期才能够确定其值的变量的。

`const` 常量称为运行期常量，所定义常量的初始化可以延迟到程序运行期。具体而言，`const` 只限制了定义的常量在程序运行期间不可被修改，但初始值即使在运行时才取得也是可以的。例如，若函数 `size()` 的功能是计算类型数据的长度，则

```
const int n = size();           // L1: 正确，但 n 值的取得是在执行函数时
constexpr int m = size();      // L2: 错误，程序编译时不知道 size() 的值
const int i = 10;
int j = 21;
const int i1 = i + 10;         // L3: 正确
const int j1 = j + 10;         // L4: 正确
constexpr int i2 = i + 10;     // L5: 正确，编译时可确定 i 值为 10
constexpr int j2 = j + 10;     // L6: 错误，j 是变量
```

语句 L2 错误的原因是：在编译时不能得知函数 `size()` 的值，除非函数 `size()` 的返回值类型也是 `constexpr`。语句 L6 的错误原因与此相似，`j` 是变量，不允许作为 `constexpr` 的初始化值。但是 `const` 无此限制，所以语句 L4 是正确的。

可以理解为，所有 `constexpr` 对象都是 `const` 对象，但不是所有 `const` 对象都是 `constexpr` 对象。如果需要根据常量拥有的值能够在编译期确定，就应该使用 `constexpr` 而不是 `const`。

2.6.2 `const`、`constexpr` 与指针

`const` 可以与指针结合，由于指针涉及“指针本身和指针所指的对象”，因此它与常量的结合也比较复杂，可分为三种情况，如图 2-6 所示。

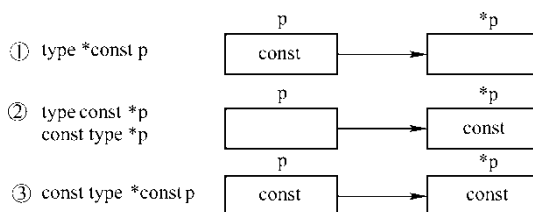


图 2-6 `const` 与指针的三种关系

`type` 代表 C++ 语言的任意数据类型，`p` 是一个指针。其中，第①种是常量指针，即指针是常量，不能被修改，但其所指内存区域是变量，可被修改；第②种是指向常量的指针，即指针是变量，可再指向其他内存单元，但其所指单元是常量，不能被修改；第③种是指向常量的常指针，指针及其所指内存单元均为常量，都不能被修改。

【例 2-12】 `const` 与指针的关系。

```
// Eg2-12.cpp
#include <iostream>
using namespace std;

int main(){
    char *const p0;           // L1 错误，p0 是常量，必须初始化
    char *const p1 = "dukang"; // L2 错误
    char const *p2;           // L3 正确
    const char *p3 = "dukang"; // L4 正确
    const char *const p4 = "dukang"; // L5 正确
    const char *const p5;     // L6 错误，p5 是常量，必须初始化
```

```

p1 = "wankang";           // L7 错误, p1 是常量, 不可改
p2 = "wankang";           // L8 正确, p2 是变量, 可改
p3 = "wankang";           // L9 正确, p3 是变量, 可改
p4 = "wankang";           // L10 错误, p4 是常量, 不可改
p1[0] = 'w';              // L11 正确
p2[0] = 'w';              // L12 错误, *p2 是常量, 不可改
p3[0] = 'w';              // L13 错误, *p3 是常量, 不可改
p4[0] = 'w';              // L14 错误, *p4 是常量, 不可改
return 0;
}

```

例中, *p2 和 *p3 的定义形式等价, 只是 p2 没有被初始化, p3 被初始化了。请结合图 2-6 和本例中的注释理解各语句正确和错误的原因。

const 对指针和变量之间的相互赋值具有一定影响: const 对象的地址只能赋给指向 const 对象的指针, 否则引起编译错误。语句 L2 错误的原因就是常量对象 dukang 只能赋给指向常量的指针 (如语句 L4、L5), 却赋给了指向变量的常量指针 p1。但是, 指向 const 对象的指针可以指向 const 对象, 也可以指向非 const 对象。例如:

```

int x = 9;                 // L1
const int y = 9;          // L2
int *p1;                  // L3
const int *p2;            // L4
p1 = &y;                  // L5, 错误
p2 = &x;                  // L6, 正确
p2 = &y;                  // L7, 正确

```

语句 L5 将引起编译错误, 若改为 “p1 = &x;”, 则是正确的。请结合上述说法, 理解语句 L7 正确的原因。

用 constexpr 限定指针比 const 简单得多, 它只限制指针变量本身是常量, 与所指的变量没有关系。例如:

```

constexpr int* p = nullptr;
static int x = 10;
int* p1 = &x;
constexpr int* p2 = &x;    // C++ 17

```

p1 是普通指针; 而 p 和 p2 是常量指针, 指向的对象不是常量。

2.6.3 const 与引用

在定义引用时, 可以用 const 进行限制, 使它成为不允许被修改的常量引用。例如:

```

int i = 9;
int &rr = i;
const int &ir = i;
rr = 8;
ir = 7;                    // 错误

```

最后一条语句是错误的, 因为 ir 是 const 引用, 不允许通过它修改对应的变量 i。

const 引用可以用常量初始化, 但非 const 引用不能用常量初始化。例如:

```

int i = 2;                // L1

```

```
const double &ff = 10.0;           // L2
const int &ir = i+10;              // L3
int &i = 3;                        // L4, 错误
```

语句 L4 错误和语句 L2、L3 正确的原因与编译器的处理方式有关。编译器在实现常量引用时生成了一个临时对象，然后让引用指向该对象。但该对象对用户而言是隐藏不可知的，不能访问。例如，对于“`const double &ff = 10.0;`”，编译器将其转换为类似如下形式：

```
double temp = 10.0;
const double &ff = temp;
```

编译器先创建临时变量 `temp`，然后将引用 `ff` 绑定到它，`temp` 将保持到引用 `ff` 的生命期结束。

2.6.4 顶层 const 和底层 const

指针实际上定义了两个对象：指针本身和它所指的对象。这两个对象都可以用 `const` 进行限定，当指针本身被限定为常量时，称指针为**顶层 const**；当所指的对象被限定为常量，而指针本身未被限定时，称指针为**底层 const**；当指针和所指对象两者都被限定为常量时，则指针为顶层 `const`，所指对象为底层 `const`。

```
int i = 0;
const int ic = 32;
int *const p1 = &i;           // p1 为顶层 const
const int *p2;                // P2 为底层 const
const int *const p3 = &ic;     // p3 为顶层 const, (*p3)为底层 const
```

更一般地，顶层 `const` 是不可被修改的常量对象，此概念可以推广到任意的数据类型，它们定义的常量对象都是顶层 `const`。底层 `const` 则与指针和引用这样的复合类型^[3]定义有关，其中指针比较特殊，既可以是顶层 `const`，也可能是底层 `const`。所有声明为 `const` 的引用都是底层 `const`。例如：

```
int i = 3;
const double d = 9.0;         // ic 为顶层 const
const int ic = 32;            // ic 为顶层 const
const int &ri = i;             // ri 为底层 const
const int &ric = ic;           // ric 为底层 const
```

在进行复制操作时，复制顶层 `const` 对象与底层 `const` 对象存在以下区别。

① 复制顶层 `const` 不受影响。由于执行复制时不影响被复制对象的值，因此它是否为常量对复制没有影响。例如，对于上述语句组，执行如下复制操作。

```
i = ic;                        // 正确：ic 是一个顶层 const，对此操作无影响
p2 = p3;                      // 正确：p2 和 p3 指向的对象类型相同，p3 顶层 const 部分不影响
```

② 底层 `const` 的复制是受限制的。要求粘贴和复制的对象有相同的底层 `const` 或者能够转换为相同的数据类型。一般，非常量能够转换成常量，反之则不行。例如，针对前述语句组，执行如下复制操作。

```
p2 = p3;                      // 正确：p2 为底层 const，p3 是顶层也是底层 const 且类型同
p2 = &i;                      // 正确：p2 为底层 const，&i 为 int*，且能转换成 const int*
```

[3] 指针和引用都包含两部分内容。指针包括指针和指针所指对象，引用包括引用本身和引用的对象。


```

p2= &ic;                // 正确: p2为底层 const, &ic 为 const int*
p2 = &ri;                // 正确: p2 和 ri 为相同类型的底层 const
int *p = p3;            // 错误: p3 包括底层 const 定义, 而 p 没有
const int &r2 = i;       // 正确: const int&可以绑定到一个普通 int 上
int &r = ic;             // 错误: 普通的 int&不能绑定到 int 常量上

```

“p2 = p3;”是正确的。因为 p3 既是底层 const 又是顶层 const, 要求粘贴的对象拥有相同的底层 const 资格, 而 p2 是一个同类型的底层 const, 符合复制条件, 所以正确。而 “int *p = p3;”是错误的, 原因是 p 不是底层 const, 不符合复制条件, 所以错误。

“p2 = &i;”是正确的。因为 p2 是底层 const, 虽然对 i 取地址得到是 int*是非常量, 但可以转换常量 “const int*”, 符合 p2 底层 const 复制的要求。

其余语句正误的原因分析, 请参考语句中的注释。

2.7 auto、decltype和decltype(auto)类型^{C++ 11/14}

1. auto (Automatic Variable Type, 自动变量类型)

auto 是 C++ 11 开始引入的类型推断定义符, 用法如下:

```
auto 变量名 1 = 表达式 1, 变量名 2 = 表达式 2, ...;
```

auto 运用从表达式结果推断出的类型定义变量, 并用表达式的值初始化该变量。auto 会忽略表达式的顶层 const 和引用的 const, 而保留指针底层 const。例如:

```

int i;
const int *const p = &i;
const int ic = i, &rc = ic;
auto x = 3 + 8;           // int x=3+8;
auto c = 's';            // char c='s'
auto s = "abcde";        // const char *s="abcde"
auto z = x + 3.8;         // double z=x+y
auto pi = &i;            // int *pi=&i
auto pc = &ic;           // const int *pc=&ic, 忽略顶层 const 保留底层 const
auto rrc = rc;            // int rrc, 忽略引用的 const
auto ric = ic;           // int ric, 忽略顶层 const
auto pp = p;             // const int *p, 忽略顶层 const

```

用 auto 设置一个引用类型时, 初始值中的顶层 const 会被保留, 而引用字面常量时需要指定为 const 引用。例如, 对上述 i 和 ic, 用 auto 定义如下引用:

```

auto &ri = i;             // int &ri = i
auto &rc = ic;            // const int &rc = ic, 顶层 const
auto &r0 = 4.3;           // 错误, 不能够将非常绑定到常数
const auto &r1 = 4.3;     // 正确, const double &r1 = 4.3

```

由于 auto 需要根据表达式的值推断数据类型, 因此要求在用 auto 定义变量时, 必须提供变量的初始化表达式, 且表达式的类型是清楚而明确的。另外, auto 是一种变量定义语句, 可以在一条语句中同时定义多个变量, 但数据类型只能有一种。例如:

```

auto x = 3, y = 12, z = 30; // 正确, x、y、z 为 int 类型
auto a = 3, b = 3.2;       // 错误, a 和 b 的类型不同

```

```
auto a = 3, j // 错误, j 没有初始化表达式
```

2. decltype(表达式类型)

decltype 也是 C++ 11 引入的类型推断定义符。如果只需定义变量，但不想用表达式值初始化它，就可用 decltype 定义，用法如下：

```
decltype(表达式) 变量 = 表达式;  
decltype((表达式)) 变量 = 表达式; // 定义引用
```

decltype 用于从表达式推断出类型并定义变量，但不用表达式的值初始化定义的变量。与 auto 不同，当表达式是变量时，decltype 的处理方式是不忽略顶层 const，其结果是定义与变量相同类型的变量（包括顶层 const 和引用在内）。

当用“(())”把表达式括起来时，定义的一定是引用。而用“()”时，只有当变量本身是引用时，定义的才是引用。

【例 2-13】 用 auto 和 decltype 定义变量。

```
// Eg2-13.cpp  
#include <iostream>  
using namespace std;  
  
int n;  
double f(int n) {  
    int s = 0;  
    for(int i = 1; i <= n; i++)  
        s += i;  
    return s;  
}  
  
void main() {  
    int i = 10, j, *p = &i, &r = i;  
    const int ic = i, &cj = ic;  
    decltype(f(5)) s; // double s  
    decltype(i+3.4) x = 9; // double x;  
    decltype(ic + 3) y1; // int y1;  
    decltype(ic) y2 = 4; // const int y2 = 4;  
    // decltype(ic) y3; // 错误, const int y3  
    decltype(p) p1; // int *p1  
    decltype((i)) ri = j; // int &ri = j  
    decltype(*p) rp = i; // int &rp = i  
    auto x1 = ic; // int x1 = ic  
    decltype(cj) x2 = ic; // const int &x2 = ic  
}
```

当表达式是解引用时，decltype 将得到引用类型，如上述 rp 所示。

从本例最后两行可以看出 auto 和 decltype 在处理推断表达式是变量时的区别，即表达式为变量 ic 时，auto 根据其值推断数据类型，而 decltype 用与 ic 相同的类型定义 x2。

此外，在对数组的处理上，auto 和 decltype 也存在差别。例如：

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
auto p1 = a; // 等价于: int *p1  
decltype(a) p2;  
// 等价于: int p2[10]
```

在处理数组的问题上，`auto` 将对象定义为指向数组第一个元素类型的指针，相当于“`auto p1(&a[0])`”，即 `p1` 为指向 `a[0]` 数据类型的指针，`a[0]` 的类型为 `int`，因此等价于定义语句“`int *p1`”；`p2` 则不同，`decltype` 采用与 `a[]` 完全相同的类型定义 `p2`，定义语句为“`int p2[10]`”。

3. `decltype(auto)` ^{C++ 14}

当用 `decltype` 从表达式中定义并初始化变量时，就会采用如下定义形式：

```
decltype(expr) var = expr;
```

如果表达式比较长，这个定义就会显行冗余和烦琐，如

```
decltype(g(x,y,z) + f(a,b,c) + 4.0*x + y^2+3) var = g(x,y,z) + f(a,b,c) + 4.0*x + y^2+3);
```

用从表达式 `(g(x, y, z)+f(a, b, c)+4.0*x+y^2+3)` 推断出的数据类型定义了变量 `var`，再用该表达式的结果初始化 `var`。简单的一个定义看起来让人费尽思考。但是，该语句与某些时候用 `STL` 或命名空间中的某些数据类型和函数组成的表达式相比，算是短的了。

为此，C++ 14 增加了表达式推断类型 `decltype(auto)`，用 `auto` 替换 `decltype(expr)` 中的 `expr`。上述 `var` 定义可用如下语句取代，这样就去掉了冗余，简化了语句，看起来简洁明了。

```
decltype(auto) var = (g(x,y,z) + f(a,b,c) + 4.0*x + y^2+3); // C++ 14
```

2.8 C++新式for循环和数组

下面简要介绍基于范围的 `for` 循环和 C++ 标准库中的向量 `vector` 和 `valarray`（见第 7 章），以方便学习过程中的批量数据处理。

2.8.1 `begin`、`end` 和基于范围的 `for` 循环 ^{C++ 11}

为了使指针、数组之类的连续数据列表操作更加简单和安全，C++ 11 引入了用于获取数组、列表、链表等序列数据首、尾地址的通用函数 `begin()`、`end()` 和范围 `for` 循环。函数 `begin()` 返回指向序列首元素的指针，函数 `end()` 返回指向序列尾元素后一位置的指针，如图 2-7 所示。

范围 `for` 语句用于遍历数组、`STL` 容器（见第 7 章）或其他序列。它们的用法如下：



图 2-7 `begin()` 和 `end()` 确定的位置

```
begin(序列)
end(序列)
for(变量声明:序列)
    循环体
```

其中的序列必须是一组同类型的连续数据，如数组、用“`{}`”括起来的值列表、`string` 字符串或 `STL` 的容器（如 `list`、`stack`、`vector` 等）。

范围 `for` 循环的工作流程如下：<1> 定义变量；<2> 将序列的第 1 个元素赋值给变量，执行循环体；<3> 将序列的第 2 个元素赋值给变量，执行循环体；然后是第 3 个、第 4 个……<4> 将序列的最后 1 个元素赋值给变量，执行循环体，结束。

【例 2-14】 利用函数 `begin()`、`end()` 和范围 `for` 循环，计算数组元素的平方，统计字符串的字符数，并将所有的字符改为大写字母。

```
// Eg2-14.cpp
```

```

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

void main() {
    int a[10] = {1,2, 3, 4, 5, 6, 7, 8, 9, 10};
    string s("hello, this is s string!");
    int n = 0;
    for(int i : a) // L1: 定义 i, 从 a[0]一直取到 a[9]
        cout << i << "\t";
    cout<<endl;
    for(auto& i : a) // L2: i 为依次为 a[0]~a[9]的引用
        i *= i; // L3: 通过引用修改 a[0]~a[9]
    for(int *p = begin(a); p != end(a); p++) // L4: 指针 p 访问 a 数组
        cout << *p << "\t";
    cout<<endl;
    for(auto &c : s) { // L5: c 依次为 s[0]~s[24]的引用
        n++; // L6: 计算字符个数
        c = toupper(c); // L7: 通过引用把字符改为大写
    }
    cout<<"s 共有: "<<n<<"个字符"<<endl;
    for(auto p = begin(s); p != end(s); p++) // L8: 通过指针访问字符串
        cout<<*p;
}

```

程序运行结果如下:

```

1   2   3   4   5   6   7   8   9   10
1   4   9   16  25  36  49  64  81  100
s 共有: 23 个字符
HELLO, THIS IS S STRING!

```

在范围 for 循环中, 用 auto 自动推算数据类型是最方便的, 将语句 L1、L4 的 for 语句的 int 改为 auto, 具有完全相同的语义和功能。注意语句 L1 和 L2 的区别: 语句 L1 中的 i 为普通变量, 通过 for 循环依次读取 a 数组的每一个下标变量; 而语句 L2 中的 i 是引用, 通过 for 循环依次作为数组 a[] 每个下标变量的引用, 并通过引用实现了对每个下标变量的平方运算。

注意: string 本身也有获取字符串首尾位置的函数 begin() 和 end(), 功能与这里的 begin() 和 end() 相同, 但用法有区别。用它实现语句 L8 等价功能的语句如下:

```

for(auto p = s.begin(); p != s.end(); p++)
    cout<<*p;

```

2.8.2 vector 和 valarray

vector 和 valarray 是 STL 中的数据结构, 它们的定义分别包含在 <vector> 和 <valarray> 头文件中, 两者的用法与数组非常相似, 但功能更强大, 具有自动识别数组元素个数的成员函数 size(), 能够动态扩展数组大小, 并可对数组排序等。其用法如下:

```

vector <数据类型> 数组名{初始列表};
vector<数据类型> 数组名 = {初始值列表};
vector<数据类型> 数组名(数组元素个数);

```

也可以一次性定义多个数组。`valarray` 的用法与 `vector` 完全相同，可以参照应用。

【例 2-15】 `vector` 应用示例。

```
// Eg2-15.cpp
#include<iostream>
#include<vector>
using namespace std;

void main() {
    vector<int> v1 = {0,1,2,3,4,5}, v2{1,2,3,4,5,6,7,8}, v3(11);           // L1
    v1.resize(v2.size());                                                // L2
    cout<<v1.size()<<endl;                                              // L3
    for(int i = 0; i < v1.size(); i++)                                   // L4
        v3[i] = v1[i] + v2[i];                                          // L5
    v3[8] = 10;                                                          // L6
    v3[9] = v3[10] = 10;                                                // L7
    for (int x : v3)                                                      // L8
        cout<<x<<"\t";
    cout<<endl;
}
```

程序运行结果如下：

```
      8
    1   3   5   7   9   11   7   8   10   10   10
```

语句 L1 定义了 `v1`、`v2` 和 `v3` 三个数组，`v1` 和 `v2` 在定义时利用初始化列表将数 `v1` 的大小定义为 6，`v2` 的大小为 8，并进行了初始化，`v3` 的大小为 11。

注意：`vector` 数组大小定义用的不是 `[]`，而是 `()`。

语句 L2 利用 `vector` 的成员函数 `size()` 获取数组 `v2` 的大小（8），并用成员函数 `resize()` 将数组 `v1` 的大小重置为这个值。`v1` 本身只有 6 个元素，现在扩展为 8 个元素，最后的两个元素自动设置为 0，即 `v1[6]`、`v1[7]` 的值都是 0。语句 L3 输出了第 1 行的 8，即 `v1` 的数组大小。

`valarray` 与 `vector` 的用法相同，将上例中的 `vector` 改为 `valarray`，其余程序代码不进行任何修改，将得到完全相同的运行结果。但是，`valarray` 是具有逐元素依次操作功能的一维数组，当操作数是标量时，会与 `valarray` 数组中的每个元素进行逐个运算。此外，两个同类型的 `valarray` 数组能够进行算术运算，如果两个数组的元数个数不同，以个数少的数组为准，长数组多出的数据元素将被舍弃，运算结果中的元素个数与短数组的元素个数相同。

下面举一个简单的 `valarray` 数组应用案例，介绍 `valarray` 在进行向量运算的高效性。

【例 2-16】 `valarray` 应用示例。

```
// Eg2-16.cpp
#include<iostream>
#include<valarray>
using namespace std;

int main() {
    valarray<float> v1 = {1,2,3,4}, v2{1,2,3,4,5}, v3(10);           // L1
    v3 = v1 + v2;                                                      // L2, v3 的元素个数与 v1 相同
    for (int x : v3)                                                    // L3
        cout << x << "\t";
}
```

```

    cout<<endl;
    v1.resize(v2.size());
    v3 = (v1 + 2.0f) + v2 * 2.0f;
    for (int x : v3)
        cout<<x<<"\t";
    cout<<endl;
    v3 = sin(v1 * 2.0f + 1.0f);
    for (int i = 0; i < v3.size(); i++)
        cout<<v3[i]<<"\t";
    cout<<endl;
}

```

// L4, v1 扩展为 5 个元素
// L5, v3 具有 5 个元素
// L6
// L7, v3、v1 有 5 个元素
// L8

程序运行结果如下：

```

2      4      6      8
4      6      8      10     12
0.841471    0.841471    0.841471    0.841471    0.841471

```

借助 `valarray` 的向量运算特性，可以写出语句 L2、L5 和 L7 这样简洁的向量运算语句，体现了 `valarray` 数组的强大运算功能。请读者结合运行结果，分析各语句的功能。

2.9 数据类型转换

数据类型转换就是将一种数据类型转换为另一种数据类型。在同一个算术表达式中，若出现了两种以上的数据类型，则会先进行数据类型转换，再计算表达式的值。例如：

```
cout<<34+21.45+'a'<<endl;
```

其中出现了 3 种数据类型：34 是 `int` 类型，21.45 是 `double` 类型，'a' 是 `char` 类型。运算的过程如下：先将 34 转换成 `double` 类型的 34.00，再完成 34.00+21.45 的运算，得到 `double` 类型的结果 55.45，然后将 `char` 型的 'a' 转换成 `double` 类型的 97.00，再计算 55.45+97.00，最后的结果是 `double` 类型的 152.45。

C++ 语言的数据类型转换分为隐式转换和显式转换两类，经常发生在算术表达式计算、函数的参数传递、函数返回值及赋值语句中。

1. 隐式类型转换

以下 4 种情况，C++ 语言会自动对参与运算的数据类型进行转换，称为**隐式类型转换**。

① 同一算术表达式中出现了多种数据类型。转换的总原则是尽可能避免损失精度，因此窄数据类型（占用存储空间少的类型）向宽数据类型转换（占用存储空间多的类型），具体情况如图 2-8 所示。

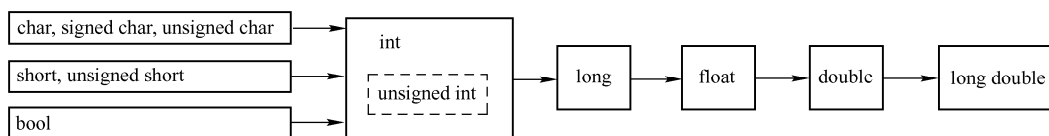


图 2-8 C++ 类型转换方法

整数转换的方法是把小整数类型转换成较大的整数类型，即对于 `bool`、`char`、`signed char`、

unsigned char、short 和 unsigned short 等类型，如果在同一表达式中出现了，只要对应变量的值能够用 int 类型存储，它们就会提升成 int 类型；反之，如果超出了整数的表示范围，就将其提升成 unsigned int 类型。

② 将一种类型的数据赋值给另一种类型的变量，会发生隐式类型转换，把赋值句右边的表达式结果转换成赋值句左边变量的类型。例如：

```
int a = 2;
float b = 3.4;
double c = 2.2;
b = a; // 将 a 的值 2 转换成 float 型的 2.0 再赋给 b
a = c; // 将 c 的值 2.2 转换成 int 型的 2 赋给 a
```

由于宽类型数据所占用的存储空间比窄类型多，因此窄类型向宽类型转换不会有什么问题，而宽类型数据转换成窄类型常会发生精度损失，是不安全的。C++常采用截取方法进行宽类型向窄类型的转换，即从宽类型中截取与窄类型大小相同的存储区域作为转换的结果，而宽类型中多出的字节就丢掉了。例如，对于“a=c”，C++将截取 c 的整数部分并赋值给 a，至于 c 的小数部分就丢掉了，所以 a 的最终结果是 2。

③ 在函数调用中，若实参与形参的类型不相符合，则把实参的类型转换成形参的类型。

④ 在函数返回时，若函数返回表达式的值与函数声明中的返回类型不相同，则把表达式结果转换成函数返回类型。例如：

```
float min(int a, int b) {
    return a < b ? a : b;
}
```

return 语句中的表达式结果为 int 类型，与函数 min() 的返回类型 float 不同，因此会发生类型转换，将“a < b ? a : b”的结果转换 float 类型后再返回给函数 min()。

假设对上面的函数 min() 存在如下函数调用：

```
int a = 2;
float b = 3.4;
int x = min(b, a+3.5);
```

由于 min() 的形参是 int 类型，因此在“min(b, a+3.5)”调用中，将把 b 的值 3.4 从 float 类型转换成 int 类型的 3，“a+3.5”的结果 5.5 为 double 类型，也被转换成 int 类型的 5，再传给相应的形参。

2. 显式类型转换

把一种数据类型强制转换为另一种类型就称为显式类型转换，也称为强制类型转换。形式如下：

```
(type) exp // C 的强制类型转换
或 type (exp) // C++ 的强制类型转换
```

其中，type 是目标类型，exp 是要进行类型转换的表达式，显式类型转换是把 exp 转换成 type 型。第一种是 C 语言支持的类型转换方式，在 C++ 语言中同样可用；第二种是 C++ 语言才允许使用的类型转换方式。例如：

```
int a = 4;
float c = (float) a; // C 语言中使用的类型转换方式在 C++ 中仍可用，结果为 4.0
```



```
a = int(8.8); // 只能用于 C++语言而不能用于 C 语言的类型转换方式，结果为 8
```

C++语言还有 4 个强制类型转换符，即 `static_cast`、`dynamic_cast`、`const_cast` 和 `reinterpret_cast`。其用法如下：

```
x_cast <type> (exp)
```

其中，`x_cast` 代表强制类型转换符，可以是 `static_cast`、`dynamic_cast`、`const_cast` 或 `reinterpret_cast` 之一，`type` 是强制转换后的类型，`exp` 是要转换类型的表达式。

`static_cast` 是静态强制转换，能够实现任何标准类型之间的转换，如从整型到枚举类型，从浮点型到整型之间的转换等。事实上，凡是隐式转换能够实现的类型转换，`static_cast` 都能够实现。例如：

```
char p = 'd';
int x = static_cast<int>(p); // 将 p 转换成 int，x=100
double y = static_cast<double>(54); // 将 54 从 int 类型转换成 double 类型
```

`const_cast` 是常量强制转换，用于强制转换 `const` 或 `volatile`（可变）的数据，它转换前后的数据类型必须相同，可以用来在运算时暂时删除数据的 `const` 限制。

【例 2-17】 利用 `const_cast` 转换去掉引用的 `const` 限制。

```
// Eg2-17.cpp
#include<iostream>
using namespace std;

void sqr(const int &x) {
    const_cast<int &>(x) = x*x; // L1, 去掉了 x 的 const 限制，否则不能修改 x
    // x = x*x; // L2, 错误，x 为 const 变量，不能被修改
}

void main() {
    int a = 5;
    const int b = 5;
    sqr(a); // L3, 通过引用将 a 改为 25
    cout<<a<<endl; // L4, 输出 25
    sqr(b); // L5, 由于 b 为 const 变量，sqr 对其修改无效
    cout<<b<<endl; // L6, 输出 5
}
```

函数 `sqr()` 的参数 `x` 是常量引用，不能修改实参的值，但是语句 “`const_cast<int &>(x);`” 在执行时暂时去掉了 `x` 的 `const` 限制，将其结果改为 “`x*x`”，本语句执行后，`x` 即恢复为 `const`，因此语句 `L2` 是错误的。语句 `L5` 通过 `sqr` 的形参 `x` 修改了实参 `b` 单元中的值为 25，但 `b` 是 `const` 变量，在函数结束时，此修改无效，`b` 恢复原来的值 5。

`reinterpret_cast` 是重解释强制转换，接受变量在内存中的地址，并把该地址对应内存单元中的二进制位数据直接当作另一个类型来解释，能够实现两个互不相关的数据类型之间的转换。例如，将整型转换成指针，或把一个指针转换成与之不相关的另一种类型的指针。

```
int i;
char *c = "try fly";
i = reinterpret_cast<int>(c);
```

`reinterpret_cast` 其实是按强制转换所指定的类型对要转换数据对应的内存区域进行重新定义。在本例中，`reinterpret_cast` 将 `c` 对应的内存区域（一个内存地址，因为 `c` 是指针）重新

定义为一个整数，这种转换在这里并没有实际意义。

`dynamic_cast` 是动态强制类型转换，主要用于基类与派生类对象之间的指针转换，以实现多态。与其他几个强制类型转换运算符不同的是，`dynamic_cast` 完成的类型转换是在程序运行期间实现的，其他类型的强制转换在编译时就完成了。

3. 类型转换和精度损失

无论是显式还是隐式类型转换，都有可能产生精度损失，特别是整数与实数之间的转换，如 `float`、`double` 与 `int`、`long` 之间的转换通常会产生精度损失。此外，表达式运算过程中的类型转换也可能引起精度损失，有时可以通过合理调整运算次序避免精度损失。

【例 2-18】 类型转换和精度损失示例。

```
// Eg2-18.cpp
#include <iostream>
using namespace std;

int main(int argc, char** argv) {
    long x = 123456789;
    long x2 = x + 1.0f - 1.0;
    long x3 = x + (1.0f - 1.0);
    cout<<"x1 = " <<x<<"\nx2 = "<<x2<<"\nx3 = "<<x3<<endl;
    return 0;
}
```

在 VS 2022 中，此程序的输出结果如下：

```
x1 = 123456789
x2 = 123456791
x3 = 123456789
```

结果表明，`x2` 发生了精度损失，`x3` 通过运算次序调整得到了正确结果。在设计程序时，需要谨慎对待类型转换产生的精度问题，并采取必要的措施减少精度损失。

2.10 函数

函数是 C++ 程序的基本构件，定义方法和规则与 C 语言的基本相同。这里仅就 C++ 语言对函数扩充的几方面进行介绍。

2.10.1 函数原型

C++ 语言是一种强类型检查语言，每个函数调用的实参在编译期间都要经过类型检查。如果实参类型与对应的形参类型不匹配，C++ 语言就会尝试可能的类型转换，若没有类型转换行得通，或实参个数与函数的参数个数不符，就会产生编译错误。要实现这样的检查，就要求所有的函数必须在调用之前进行声明或定义。

函数原型就是常说的函数声明，由函数返回类型、函数名和形参表三部分构成。形参表中包括所有形参的类型和参数名，参数之间用“,” 隔开。形式如下：

```
rtype f_name(type1 p1, type2 p2, ...);
```

其中, `rtype` 是函数的返回类型, `f_name` 是函数名, `type1`、`type2`……是形参的类型, `p1`、`p2`……是形参名。形参名可以省略, 即 `p1`、`p2` 是可省略的。

虽然函数原型只有一条语句, 但是它描述了函数的接口, 说明了调用该函数的全部信息。

【例 2-19】 函数原型的一个简单例子: 计算数字平方的函数。

```
// Eg2-19.cpp
#include <iostream>
using namespace std;

double sqrt(double f);                                // L1, 函数原型
void main() {
    for(int i = 0; i < 10; i++)
        cout<<i<<"*"<<i<<" = "<<sqrt(i)<<endl;
}

double sqrt(double f) {                                // L2, 函数定义
    return f*f;
}
```

`main()`之前的函数声明 `double sqrt(double f)`就是函数原型, 其中的形参名 `f` 是可省略的。在 C 语言中没有这个声明也可以, 但在 C++语言中没有这个函数原型就会引起编译错误。

说明:

① 函数定义时的返回类型、函数名以及参数的个数、次序和类型必须与函数原型相符, 但参数名可以不同。如下函数与例 2-19 中的 `sqrt()`是同一函数, 尽管它们的参数名不同。

```
double sqrt(double d){ return d*d; }
```

② 若一个函数没有返回类型, 则必须指明它的返回类型为 `void` (包括主函数 `main()`)。

```
int f(int, int);
f(int, int);                                // L1, 在 C++ 98 中可行, 但在 C++ 11 中错误
void f1(int i, int j);
void main()
```

上面两个 `f()`函数在早期 C++中完全等价, 其返回类型为 `int`。但在标准 C++中, 函数都应该有返回类型, 因此语句 L1 在某些编译环境中会产生编译错误。

2.10.2 函数参数传递的方式

当函数被调用执行时, C++首先分配一片存储区域供它使用, 函数执行完成后就收回, 称为堆栈 (栈)。函数执行期间, 会在堆栈中为形参和函数定义的变量分配内存单元。

在调用函数时, C++会先把实参传递给形参, 称为参数传递。传递的内容可以是实参的值、地址或引用。据此常将 C++的参数传递分为 3 类: 值传递, 指针传递, 引用传递。

1. 值传递

值传递是用实参的值来初始化形参, 方法是把实参的值复制到对应的形式参数在堆栈内分配到的存储单元中, 复制完成后, 实参与形参之间就没有关系了, 这种参数传递方式也称为按值传递。

按值传递参数时, 函数处理的是实参的复制值, 这些复制值在堆栈中, 其修改不会引起

实参值的变化。当函数执行完后，该函数用来保存数据的堆栈被系统收回，函数中定义的变量、常数以及函数调用时传递的参数都会因存储区域的释放而销毁。

例如，在如下代码段中，函数 `swap1()` 无法实现两数 `x` 和 `y` 的交换。

```
void swap1(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
x = 10;
y = 5;
swap1(x, y);
```

当 `swap1(x, y)` 调用发生时，将 `x` 的值 10 复制给形参 `a`，将 `y` 的值复制给形参 `b`，之后 `x` 与 `a`、`y` 和 `b` 就无联系了，`swap1()` 函数实际交换的是形参 `a` 和 `b` 的值。因此，执行 `swap1(x, y)` 函数调用后，`x` 的值仍然为 10，`y` 仍然为 5。

2. 指针传递

指针作为参数时，C++ 把实参的地址复制到指针形参在堆栈内分配到的存储单元中，使指针形参指向实参的内存区域，因此能够实现对实参的操作。例如，用指针参数实现两数交换的函数如下：

```
void swap2(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
x = 10;
y = 5;
swap2(&x, &y);
```

在调用函数 `swap2()` 时，C++ 把实参 `x` 的地址复制到参数 `a` 在堆栈内的存储单元中，把 `y` 的地址复制到参数 `b` 在堆栈内的存储单元中。参数 `a`、`b` 实际指向了 `x`、`y` 的内存单元，因此函数 `swap2()` 通过指针参数 `a`、`b`，就能完成实参 `x` 和 `y` 对应的内存数据的交换。

在应用指针处理函数的参数时，必须把数组参数结合在一起考虑。数组的两个特性对其作为函数参数有较大影响：① 不允许复制数组；② 数组通常被转换成指针。

【例 2-20】 设计对具有 6 个元素的整数数组进行冒泡法排序的函数。

```
// Eg2-20.cpp
void sortArr(int a[6]) {
    for(int i = 0; i < 6-1; i++)
        for(int j = 0; j < 6-i-1; j++) {
            if(a[j] > a[j+1]) {
                int t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
            }
        }
}
```

假设有如下数组和函数调用：

```
void main() {  
    int b[] = {21, 13, 4, 1, 7, 5};  
    sortArr(b);  
}
```

因为上面的两个原因，不能复制数组，意味着无法按值传递方式把数组 `b` 的每个元素值复制到形参数组 `a` 的每个下标元素中。数组参数会被转换成指针（`int *a`），当向它传递数组时，实际上传给它的是指向实参数组首元素的指针。因此，`sortArr(b)`等价于 `sortArr(&b[0])`。更一般，若仅从调用合法性上讲，只要向函数 `sortArr()`传递一个整型变量的地址都是允许的。例如：

```
int x = 9;  
sortArr(&x); // 语法正确，但执行函数的 for 循环会出问题
```

虽然不能按值传递的方式传递数组，但 C/C++语言允许把形参写成数组的形式：

```
void sortArr(int a[]) {...}  
void sortArr(int a[6]) {...} // 希望有 6 个元素，实际上并不一定
```

这两个函数本质相同，它们会被转换成如下函数。

```
void sortArr(int *a) {...}
```

这 3 个函数完全是一个函数。

因此，希望采用类似于“`void sortArr(int a[6])`”的参数形式指定数组的大小不可行，常规方法是显式传递一个表示数组大小的参数。类似于如下形式，两个函数完全相同。

```
void sortArr(int a[], int n) {...}  
void sortArr(int *a, int n) {...}
```

另一种方法是传递数组引用参数，因为 C++语言允许定义数组的引用，也就可以将它作为函数的参数或返回类型。例如，修改函数 `sortArr()`的参数表如下，代码不做任何修改。

```
void sortArr(int (&a)[6]){...}
```

这种方法是有缺陷的，使用起来并不方便。因为引用参数要求实参完全匹配，而数组大小也是数组类型的一部分，因此函数 `sortArr()`仅能对具有 6 个元素的整数数组进行排序，多于或少于 6 个元素都不行。如针对函数 `sortArr(int (&a)[6]){...}`的下列调用：

```
int b[] = {21,13,4,1,7,5};  
int c[] = {1,2,3,4,5}, k = 0;  
sortArr(b); // 正确  
sortArr(&k); // 错误，k 不是具有 6 个元素的数组  
sortArr(c); // 错误，c 不是具有 6 个元素的数组
```

3. 引用参数

引用传递能够达到与指针参数同样的效果，但使用方式与按值传递参数相同，比使用指针参数简单。引用作为参数传递的是实参变量本身（引用是变量的左值，因此传递的是实参的地址），而不是将实参的值复制到函数参数在运行栈中的存储区域中。此外，引用参数可被认为是变量的别名，在传递引用参数时，引用参数对应实参的别名。因此，对于引用参数，函数操作的是实参本身，而不是实参的副本，这意味着函数能够改变实参的值。

【例 2-21】 用引用参数实现两数交换的函数 `swap()`。

```
// Eg2-21.cpp
#include<iostream>
using namespace std;

void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

void main(){
    int x = 5, y = 10;
    swap(x, y);
    cout<<"x = "<<x<<"\ty = "<<y<<endl;
}
```

程序的运行结果如下:

```
x = 10 y = 5
```

由于引用参数传递的是实参地址，因此在调用函数时，不能向引用参数传递常数。如对于例 2-21 的函数 swap()，如下调用是错误的：

```
int x = 5;
swap(3, 4);           // 错误，3, 4 是常数
swap(x, 9);           // 错误，9 是常数
swap(6, x);           // 错误，6 是常数
```

除了像指针一样用于改变实参的值时需要引用参数，C++引入引用的另一原因是传递大型的类对象或数据结构。在按值传递参数的情况下，传递小型类对象和结构变量不存在效率问题，但在传递大型结构变量或类对象时，需要进行大量的数据复制（把实参对象或结构变量的值复制到函数参数在运行栈分配的存储区域中），效率就太低了。

【例 2-22】 按值传递参数与引用传递参数的效率对比。

```
// Eg2-22.cpp
#include <iostream>
#include <string>
using namespace std;

struct student {
    char name[12] = "";           // 学生姓名，初始化为空字符串
    char Id[8] = "";              // 学号，初始化为空字符串
    int age = 0;                  // 年龄，初始化为 0
    double score[10] = {0};       // 10 科成绩，初始化为 0
};

void print(student a) {
    cout<<a.name<<"\t"<<a.Id <<"\t"<<a.age<<endl;
    for(int i = 0; i < 10; i++)
        cout<<a.score[i]<<endl;
}

void main() {
    student x;
```

```

// .....
print(x);
cout<<sizeof(x)<<endl;
}
// 对 x 进行赋值的语句省掉了
// 计算 x 的内存块大小

```

在调用函数 `print(x)` 打印学生的各项数据时，将把 `x` 的各项数据复制到 `print()` 的堆栈内为参数 `a` 分配的存储块中。最后一条语句计算出学生结构的大小是 104 字节，计算过程如下：

$$12 \text{ (name)} + 8 \text{ (Id)} + 4 \text{ (age)} + 8 \times 10 \text{ (score)} = 104$$

即向函数 `print()` 传递 `student` 类型的参数数据时，将完成 104 字节的数据复制。如果频繁调用此函数，进行参数复制的开销是相当可观的。若将函数 `print()` 的参数改为引用形式：

```
void print(student &a){...}
```

则每次调用该函数打印 `student` 类型的学生数据时，只需要复制 4 字节的地址数据（对于 32 位计算机），较之于 104 字节的参数复制，减少了大量的数据复制。

2.10.3 函数默认参数

C++ 允许为函数提供默认参数，也称为缺省参数。在调用具有默认参数的函数时，如果没有提供调用参数，将自动把默认参数作为相应参数的值。

【例 2-23】 设计函数 `sqrt()`，计算给定数字的平方，默认计算 1.0 的平方。

```

// Eg2-23.cpp
#include <iostream>
using namespace std;

double sqrt(double f = 1.0); // sqrt()具有默认参数值，默认时 f = 1.0
void main() {
    cout<<sqrt()<<endl; // 调用 sqrt()时没有提供实参，按默认值 f = 1.0 调用函数
    cout<<sqrt(5)<<endl; // 调用时提供了实参，默认值就无效了，f = 5
}
// 定义时，不能指定默认参数。如下语句若改为 double sqrt(double f = 1.0)，则错
double sqrt(double f) {
    return f*f;
}

```

说明：

① 在指定函数的默认值时，如果它有函数原型，就只能在函数原型中指定对应参数的默认值，不能在定义函数时再重复指定参数默认值。当然，若没有函数原型，在定义时指定参数默认值即可。

② 在具有多个参数的函数中指定默认值时，所有默认参数都必须出现在无默认值参数的右边。即，一旦某参数开始指定默认值，它右边的所有参数都必须指定默认值。

```

int f(int i1, int i2 = 2, int i3 = 0); // 正确
int g(int i1, int i2 = 0, int i3); // 错误，i3 没有默认值
int h(int i1 = 0, int i2, int i3 = 0); // 错误，i1 默认后，其右边的 i2 没有默认值

```

③ 可以用表达式作为默认参数，只要表达式可以转换成形参所需的类型即可。全局变量可以作为默认参数值，但是局部变量不能。

【例 2-24】 设计函数 `dog()`，输出狗名、高和长。默认狗名为 `tom`，0.8 米高，1.1 米长。

```
// Eg2-24.cpp
#include <iostream>
#include <string>
using namespace std;

string name = "tom";
double h = 0.8, len = 1.1;
void dog(string dogname = name, double high = h, double lenh = len) {
    cout<<"Dogname:"<<dogname<<"\tHigh:"<<h<<"\tLenth:"<<len<<endl;
}

int main() {
    name = "Jake"; // L1, 修改全局变量, 改变默认实参值
    double h = 2.1; // L2, h 隐藏了全局变量 h, 对 dog 参数 h 的默认值无影响
    dog();
    return 0;
}
```

运行结果如下:

```
Dogname:Jake    High:0.8    Lenth:1.1
```

运行结果分析: `dog()` 函数应用全局变量 `name`、`h`、`len` 作为参数的默认值; 调用函数 `dog()` 时, 系统会将调用点上 `name`、`h`、`len` 的值作为对应参数的默认值。调用函数 `dog()` 前, 语句 L1 修改了全局变量 `name` 的值, 也就改变了 `dogname` 参数的默认值。语句 L2 定义了局部变量 `h`, 隐藏了全局变量 `h`, 但并不影响全局变量 `h` 的值 (仍为 0.8), 而函数参数的默认值与局部变量无关, 所以有上面的输出结果。

④ 在调用具有默认参数值的函数时, 若某实参采用默认值, 则其右边的所有实参都应采用默认值。例如:

```
int f(int i1 = 1, int i2 = 2, int i3 = 0){ return i1+i2+i3; }
```

针对此函数, 有如下调用:

```
f(); // 正确, i1 = 1, i2 = 2, i3 = 0
f(3); // 正确, i1 = 3, i2 = 2, i3 = 0
f(2, 3); // 正确, i1 = 2, i2 = 3, i3 = 0
f(4, 5, 6); // 正确, i1 = 4, i2 = 5, i3 = 6
f(, 2, 3); // 错误, i1 默认了, 而右边的 i2、i3 没有默认
```

2.10.4 函数返回值

C++ 函数返回值的基本规则和方法与 C 语言的相同, 在此仅对增、改的内容进行简介。

1. 默认返回值和返回 void

在函数没有指定返回类型时, C 语言和早期的 C++ 默认其返回值为 `int` 类型, C++ 11 不再支持这种默认返回值。除了类的构造函数、析构函数和类型转换运算符重载函数可以没有返回类型, 所有函数都必须有返回类型, 如果确实不返回函数值, 就应当指定其返回值为 `void`。

指定返回值为 `void` 的函数可以没有 `return` 语句, 其余函数 (除了类的构造函数和析构函数) 都必须用 `return` 返回函数计算结果。`return` 语句的功能是结束当前正在执行的函数, 将执行程序的控制权返回调用该函数的语句位置。它有两种调用形式:

```
return;  
return 表达式;
```

事实上，每个函数都是通过 `return` 语句结束函数调用的。虽然返回值为 `void` 的函数没有 `return` 语句，但系统会在该函数最后一条语句的后面隐式地执行 `return` 语句。

【例 2-25】 设计求数组最大值的函数 `maxArr()`、实现两数交换的函数 `swap()`。

```
// Eg2-25.cpp  
int maxArr(int a[], int n) {                               // L1  
    int max = a[0];  
    for(int i = 1; i < n; i++)  
        if(max < a[i])  
            max = a[i];  
    return max;  
}  
  
void swap(int &a, int &b) {  
    if(a == b)  
        return;                                           // L2  
    else{  
        int t = a;  
        a = b;  
        b = t;  
    }  
}
```

在早期 C 语言程序中，如果语句 L1 的前面没有 `int`，也是正确的，系统会默认为 `int`，但在 C++ 11 中是错误的，必须为该函数指定返回类型。

在函数 `swap()` 中，如果要交换的两个变量的值相同，就直接结束程序，没有执行交换的必要了。

2. 返回引用

函数可以返回一个引用，原型如下：

```
rtype &f_name(type1 p1, type2 p2, ...);
```

其中，`rtype` 是返回类型，`type1` 和 `type2` 分别是形参 `p1`、`p2` 的数据类型。

当一个函数返回引用时，实际返回了一个变量的内存地址。既然是内存地址，就能够读写该地址所对应的内存区域中的值，因此函数调用能够出现在赋值语句的左边。

【例 2-26】 返回引用的两数相加函数。

```
// Eg2-26.cpp  
#include <iostream>  
using namespace std;  
  
int s;  
int& f(int i1, int i2) {  
    s = i1+i2;  
    return s;  
}  
void main(){
```

```

    int t = f(1, 3);                                // L1
    cout<<s<<" ";                                  // L2
    f(2, 8)++;                                       // L3
    cout<<s<<" ";                                  // L4
    f(2, 3) = 9;                                    // L5
    cout<<s<<endl;                                  // L6
}

```

运行结果如下：

```
4 11 9
```

由于函数 `f()` 的返回类型为引用，因此它返回的是全局变量 `s` 的地址。语句 `L1` 中的函数调用 `f(1, 3)` 将把 1 和 3 相加的结果 4 存入 `s`，并返回 `s` 的地址，最后把 `s` 中的值复制到 `t` 中。语句 `L1` 执行后，`s` 的值成为 4，所以语句 `L2` 输出的 `s` 值为 4。

语句 `L3` 中的 `f(2, 8)` 将使 `s` 的值更改为 10，然后返回 `s` 的地址，之后对 `s` 执行了自增运算，所以语句 `L4` 输出的 `s` 值为 11。

语句 `L5` 中的 `f(2, 3)` 将修改 `s` 的值为 5，然后返回 `s` 的地址，再将该地址中的值改为 9。这次函数调用实际上等效于如下两条语句，因为 `f(2, 3)` 返回的是 `s` 变量的地址：

```

f(2, 3);
s = 9;

```

所以语句 `L6` 输出的 `s` 值为 9。

当一个函数返回引用时，`return` 语句必须返回一个变量，返回值类型的函数中的 `return` 则可以返回一个表达式。作为返回值类型的函数，如下函数 `g()` 是正确的：

```

int g(int i1, int i2) {
    return i1+i2;
}

```

但是，如下将 `g()` 函数定义成返回引用的函数则是错误的：

```

int &g(int i1, int i2) {
    return i1+i2;
}

```

原因是返回引用的函数需要返回一个变量。若将函数 `g()` 改为返回常量的引用函数，在 C++ 中是允许的。例如：

```

const int &g(int i1, int i2) {
    return i1+i2;
}

```

C++ 在返回一个表达式时，将首先计算表达式的值，然后生成一个临时变量，并将表达式的值存入生成的临时变量。“`return i1+i2`” 的处理过程大致如下：

```

int temp = i1+i2;
return temp;

```

临时变量 `temp` 对程序员并不可见，当函数调用完成时，`temp` 占用的内存空间就被回收了，所以一个返回类型为引用的函数不能返回一个表达式。但是当将函数定义成返回 `const` 引用类型的函数时，C++ 将把 `temp` 的地址作为函数的返回值，并且保留 `temp`，直到引用函数结果的变量的生命期结束。

3. 返回 auto C++ 14

自 C++ 14 起, 可以用 auto 自动推断函数的返回类型, 这项特性对于需要处理多种数据类型的函数设计非常有用, 能够以精简的代码设计出同时处理多种数据类型的通用程序。

【例 2-27】 返回 auto 的求两数最大值函数。

```
// Eg2-27.cpp
#include<iostream>
using namespace std;

auto max(int a, double b) {
    return a > b ? a : b;
}

int main() {
    cout<<"max(4, 3.4) = "<<max(4, 3.4)<<"\tmax(5, 7.8) = "<<max(5, 7.8)<<endl;
}
```

程序运行结果如下,

max(4, 3.4) = 4 max(5, 7.8) = 7.8

2.10.5 函数重载

1. 函数重载的概念

函数重载就是允许在同一程序中定义多个同名函数, C++规定, 重载函数必须具有不同的形参表 (如不同的参数类型或参数个数等)。

【例 2-28】 重载计算 int、float、double 三种类型数据绝对值的函数。

```
// Eg2-28.cpp
#include<iostream>
using namespace std;

int Abs(int x) { return x>0?x:-x; }
float Abs(float x) { return x>0?x:-x; }
double Abs(double x) { return x>0?x:-x; }

void main() {
    cout<<Abs(-9)<<endl;
    cout<<Abs(-9.9f)<<endl;
    cout<<Abs(-9.8)<<endl;
}
```

调用时, C++会调用与实参类型最相符合的那个重载函数。Abs(-9)的实参是 int 类型, 所以将调用函数 Abs(int x), Abs(-9.9f)调用函数 Abs(float x), Abs(-9.8)调用函数 Abs(double x)。

2. 函数重载解析过程

把函数调用与多个同名重载函数中的某函数相关联的过程称为[函数重载解析](#)。在具有多个同名函数的情况下, 需要找到形式参数与实参类型匹配最好的那个函数, 匹配的原则和次序如下。

① 精确匹配。精确匹配是指函数的实参与形式参数类型完全相同, 不需要做任何转换或

只需进行平凡转换（如从数组名到指针、函数名到函数指针或 $T^{[4]}$ 到 `const T` 等）的参数匹配。

② 提升匹配。提升主要是指从窄类型到宽类型的转换，这种转换没有精度损失，包括整数提升和 `float` 到 `double` 的提升。整数提升包括从 `bool` 到 `int`、`char` 到 `int`、`short` 到 `int`，以及它们的无符号版本，如 `unsigned short` 到 `unsigned int` 的提升。

③ 标准转换匹配。如 `int` 到 `double`、`double` 到 `int`、`double` 到 `long double`，派生类指针到基类指针的转换（见第 4 章），如 `T*` 到 `void *`、`int` 到 `unsigned int` 的转换。

④ 用户定义的类型转换。在 C++ 语言中，程序员可以定义类型转换函数（见 6.4.3 节）。如果在程序中定义了这样的转换函数，这些转换函数也会用于重载函数的匹配。

【例 2-29】 函数重载解析的例子。

```
// Eg2-29.cpp
#include <iostream>
using namespace std;

void f(int i){ cout<<i<<endl; }
void f(const char*s){ cout<<s<<endl; }

void main() {
    char c = 'A';
    int i = 1;
    short s = 2;
    double ff = 3.4;
    char a[10] = "123456789";
    f(c); // f(int i) 提升
    f(i); // f(int i) 精确匹配
    f(s); // f(int i) 提升
    f(ff); // f(int i) 转换
    f('a'); // f(int i) 提升
    f(3); // f(int i) 精确匹配
    f("string"); // f(const char*s) 精确匹配
    f(a); // f(const char*s) 精确匹配
}
```

在进行重载函数解析时，将按照“**精确匹配**→**提升**→**转换**→**用户定义的类型转换**”的次序寻找一个恰当的函数调用。

例如，对于 `f(i)` 调用，由于 `i` 是 `int`，正好与 `f(int)` 的形参类型相匹配，因此 `f(i)` 会调用函数 `f(int)`；对于 `f(c)` 调用，由于 `c` 是 `char` 类型，在重载函数中没有 `f(char)` 这样的函数，因此精确匹配失败，接着 C++ 会尝试能否进行参数提升，恰好 `char` 能够提升为 `int`，所以调用 `f(int)`。

对于 `f(ff)` 调用，首先是参数匹配失败，接着是参数提升失败，然后尝试参数类型转换，正好 `double` 能够转换成 `int`（要损失数值精度），所以调用函数 `f(int)`。

3. 重载函数的注意事项

① 重载函数必须具有不同的参数表（即参数类型，或参数个数，或参数顺序方面有所不同）才是正确的。如果两个函数只有返回类型不同，而函数名、参数表都完全相同，就不能称为重载函数，而是属于函数的重复定义，是错误的。如下 3 个 `f()` 函数是正确的函数重载：

[4] T 代表任意数据类型。

```
int f(int, int);
double f(int);
int f(char);
```

如下两个 f() 函数只有返回类型不同，是错误的：

```
int f(int);
double f(int);
```

② 在定义和调用重载函数时，要注意它们的二义性。例如：

```
int f(int x) { ... }
double f(int& x) { ... }
int g(unsigned int x) { return x; }
double g(double x) { return x; }
```

函数 f() 和 g() 都是正确的重载函数，但是如何调用它们呢？例如：

```
int a = 1;
f(a); // 错误，产生二义性
g(a); // 错误，产生二义性
```

对于 f(a) 函数调用，C++ 无法确定调用 f(int& x) 还是 f(int x)，因为两种调用的函数参数都能够精确匹配，所以会产生二义性。因此，需要谨慎对待值形参和引用参数混用的重载函数，如果一个函数使用了引用参数，其他重载函数最好也使用引用参数，否则容易出现类似 f() 函数重载的二义性问题。

对于函数调用 g(a)，由于精确匹配和提升对于 g(a) 的调用都会失败，因此会使用转换的原则调用 g(a)，但 int 既可以转换成 unsigned int，也可以转换成 double，即 g(a) 调用 g(unsigned int x) 或 g(double x) 都是正确的，因此也会产生二义性。

注意：C++ 所有的标准转换都是等价的，没有哪个转换存在优先权，从 int 到 unsigned int 的转换并不比从 int 到 double 的转换高一个优先级。

③ 重载函数和 const 形参。顶层 const 参数不影响实参的传入，也就是说，一个拥有顶层 const 参数的函数无法与另一个没有顶层 const 的同名同参数类型的函数相区别，属于重定义错误。例如，同一程序中若出现如下两个函数，则会产生重定义编译错误。

```
int f(int x, int y) { cout<<"fa"<<endl; }
int f(const int x, const int y) { cout<<"fb"<<endl; }
```

底层 const 则是可区分的，拥有指针或引用参数的函数和拥有底层 const 指针或引用的同名函数属于重载函数。

【例 2-30】 设计通过底层 const 引用区分重载函数 f()，通过底层 const 指针区分的函数 g()。

```
// Eg2-30.cpp
#include <iostream>
using namespace std;

void f(int &x) { cout<<"f(int &)"<<endl; }
void f(const int &x) { cout<<"f(const int&)"<<endl; }
void g(const int * x) { cout<<"g(const int *)"<<endl; }
void g(int * x) { cout<<"g(int *)"<<endl; }

void main() {
    int x = 10;
    const int y = 9;
```

```

f(x); // 调用 f(int &x)
f(y); // 调用 f(const int &)
g(&x); // 调用 g(int *)
}

```

程序运行结果如下：

```

f(int &)
f(const int& )
g(int *)

```

因为 `const` 对象不能转换成非 `const` 对象，所以用 `const` 对象调用函数 `f()` 和 `g()` 时只能调用 `const` 参数的函数。反之，非常量可以转换为 `const` 常量，因此 4 个函数都能够接收非常量参数。但当传递的实参为非常量对象的引用或指向常量的指针时，编译器会优先调用非常量参数的函数。

2.10.6 函数与 `const` 和 `constexpr`

在 C++ 中，函数参数可能是大型对象，用值传递方式进行参数传递时需要进行大量的数据复制，存储空间和运行时间的开销较大，效率较低。而用 `const` 或 `constexpr` 限定的指针或引用传递参数则可以避免函数对参数对象进行修改，既高效又安全。

用 `const` 限制函数的参数能够保证函数不对参数做任何修改，但向形参传递实参的过程就是对对象的复制过程，同样要遵守 2.6.4 节中关于顶层 `const` 和底层 `const` 复制的规则。

1. 形参是顶层 `const`

一方面，`const` 限定的参数不可修改；另一方面，实参传递忽略顶层 `const`。例如：

```

int f(int i1, const int i2) {
    i1++;
    // i2++; // 错误，i2 是 const，不可修改
    return i1+i2;
}

```

本函数中的 `i1++` 没有问题，而 `i2++` 是错误的。原因是，`i2` 是 `const` 参数，而 `const` 变量不允许重新赋值，也不允许修改。`i2++` 将使 `i2` 增加 1，是不允许的。

在调用函数时，可以忽略参数的顶层 `const` 限制，即向顶层 `const` 参数传递的实参既可以是常量对象，也可以是非常量对象。由于参数 `i2` 就是顶层 `const`，可以接收常量和非常量实参，因此下述调用都正确。

```

const int x = 9;
int y = 100;
f(100, x); // x 是常量实参
f(x, y);   // y 是非常量实参

```

由于参数的顶层 `const` 被忽略，因此不能通过顶层 `const` 区分函数形参，在重载函数时当注意这个问题。例如，若还存在如下函数 `f()`，则不是重载，而是重定义错误。

```

int f(int i1, int i2) {...}

```

2. 形参是底层 `const`

简言之，常见的底层 `const` 包括用 `const` 限定的引用和指针两种情况，它们的复制规则同

样适用于函数的底层 `const` 参数，即同类型的底层 `const` 或者能够被转换为相同的数据类型才能够被复制，而且非常量能够被转换成常量，但常量不能被转换为非常量。例如：

```
int i = 10, const j = 10;
const int *p1 = &i;           // 正确
const int *p2 = &j;           // 正确
const int &r1 = i;             // 正确
const int &r2 = 10;            // 正确
int *p3 = p1;                 // 错误
int &r3 = r1;                  // 错误
int &r4 = r2;                  // 错误
```

`p1`、`p2`、`r1`、`r2` 都是底层 `const`，能够接收常量和非常量。`p2` 和 `r2` 接收的类型是类型相同的常量对象，类型完全匹配，所以正确。`p1` 和 `r1` 接收的是类型相同的非常量对象，非常量可以向常量转换，因此也正确。

`p1` 是底层 `const`，要求复制对象也是底层 `const`，但 `p3` 是普通对象而非底层 `const`，所以出错。`r3` 和 `r4` 是普通引用，普通引用只能用相同类型初始化，但 `r1` 和 `r2` 都常量，类型不符。在函数参数是底层 `const` 时，上面的复制规则适用于底层 `const` 参数传递。

【例 2-31】 `fp1()` 和 `fr1()` 分别是有底层 `const` 指针和引用的函数，`fp2()` 和 `fr2()` 具有无 `const` 限定普通指针和引用的函数，运用上述底层 `const` 参数的复制规则，分析程序中 `fp1()`、`fp2()`、`fr1()` 和 `fr2()` 调用正确或错误的原因。

```
// Eg2-31.cpp
#include <iostream>
using namespace std;

void fp1(const int *ap1){ }
void fp2(int *ap2){ }
void fr1(const int & ar1){ cout << "fr1" << endl; }
void fr2(int &ar2){ cout << "fr2" << endl; }

void main() {
    int i = 10;
    const int j = 10;
    int *p1;
    const int *p2;
    const int *const p3=&i;
    fp1(p1);    fp1(p2);    fp1(p3);           // 正确
    fp1(&i);    fp1(&j);    fp2(p1);           // 正确
    fp2(p2);           // 错误
    fp2(p3);           // 错误
    fp2(&i);           // 正确
    fp2(&j);           // 错误
    fr1(i);    fr1(j);           // 正确
    fr2(i);           // 正确
    fr2(j);           // 错误，非 const 引用参数只能接收同类非 const 实参
}
```

对于按值传递的函数参数而言，将参数限定为 `const` 型意义不大，因为它们不会引起函数调用时实参的变化。但指针和引用参数存在实参被意外修改的危险，将相关参数限制为 `const`

类型则可以避免这种风险。此外，对于返回指针或引用的函数，也可以用 `const` 限制其返回值。

【例 2-32】 返回 `const` 引用的函数。

```
// Eg2-32.cpp
#include<iostream>
using namespace std;

const int& index(int x[], int n) {
    return x[n];
}

void main() {
    int a[] = {0,1,2,3,4,5,6,7,8,9};
    cout<<index(a, 6)<<endl;
    index(a, 2) = 90; // 错误
    cout<<a[2]<<endl;
}
```

函数调用 `index(a, 2)` 返回 `a[2]` 的地址，由于函数 `index()` 返回的是 `const` 引用，因此不能对它进行修改。如果 `index()` 的返回值没有 `const` 限制，如

```
int& index(int x[], int n) {
    return x[n];
}
```

那么主函数没有错误。“`index(a, 2) = 90;`”将先返回数组元素 `a[2]`，再将它改为 90，则函数 `main()` 中的最后一条语句将输出 90。

用 `constexpr` 限定函数与 `const` 有些区别，主要有以下两点：[C++ 11](#)

① 如果需要在编译期间就确定常量值（如数组下标），最好使用 `constexpr` 而非 `const` 函数。只要传入的参数值都能够在编译期确定，`constexpr` 函数就能够在编译期计算出函数值。但是，如果有任何一个参数的值不能在编译期间确定，就会产生错误。

② 使用了不能够在编译期间确定值的参数调用 `constexpr` 函数时，该函数可以像普通函数一样被调用，C++ 会在运行期计算它的值。

【例 2-33】 返回 `constexpr` 常量的函数。

```
// Eg2-33.cpp
#include <iostream>
using namespace std;

constexpr int inc(int i) { return i+1; }

int main() {
    int x;
    cin>>x;
    double stu1[inc(x)]; // L1, 错误
    double stu2[inc(9)]; // L2, 正确
    cout<<inc(x)<<endl // L3, 正确
    return 0;
}
```

语句 L1 错误的原因是，定义数组 `stu1` 需要在编译期间确定下标维度大小，但 `inc(x)` 调用的参数 `x` 不能够在编译期确定，所以错误。语句 L2 传递给 `inc` 的参数 9 是已知值，在编译期

就能够确定函数 `inc()` 的最终结果，即 10。因此，语句 L2 等效于 “`double stu2[10]`”。语句 L3 中，虽然传递给函数 `inc()` 的参数 `x` 的值不能够在编译期确定，但并未要求在编译期间就确定函数值，因此就会像普通函数一样工作，即在执行到该语句时才调用函数 `inc(x)`。

2.10.7 内联函数

函数调用是件开销很大的事情，需要完成诸如保存寄存器中的值、复制参数到堆栈等一系列操作，为了避免函数调用的开销，可以将函数设置为内联函数。在函数声明或定义时，将 `inline` 关键字加在函数返回类型前面，就将它指定成了内联函数（`inline function`）。

【例 2-34】 求两个数最大值的内联函数。

```
// Eg2-34.cpp
#include<iostream>
using namespace std;

inline int max(int a, int b) {
    return a>b ? a : b;
}

void main() {
    int x1 = max(3, 4);
    int x2 = max(7, 2);
    int x3 = max(x1, x2);
    cout<<"x1 = "<<x1<<"\tx2 = "<<x2<<"\tx3 = "<<x3<<endl;
}
```

内联函数的声明、定义和调用方法与普通函数相同，但两者的编译执行方式不同。在编译时，C++ 将用内联函数的代码替换对它的每次调用。如上函数 `main()` 多次调用了内联函数 `max()`，在编译此程序时会将 `main()` 中的函数调用替换成如下形式：

```
void main() {
    int x1 = 3>4 ? 3 : 4;
    int x2 = 7>2 ? 7 : 2;
    int x3 = x1>x2 ? x1 : x2;
}
```

由此可以看出，编译程序不需为内联函数建立函数调用时的运行环境，不需要进行参数传递，所以执行时间更快。当然，由于每次调用内联函数时，都会插入它的函数代码，因此会使程序的代码增加，占用更多的存储空间。

说明：

① 内联函数的声明或定义必须在函数被调用之前完成。

② 一般而言，只有几行程序代码的、经常被调用的简单函数才适宜作为内联函数。

③ `inline` 关键字仅是对编译器的一种建议，并非写上 `inline` 的函数就一定是内联函数，将复杂函数指定为内联函数是无效的，编译器还是会把它处理成普通函数。例如，以下 3 类函数就不能作为内联函数：递归函数，函数体内含有循环、`switch` 语句之类复杂结构的函数，或者具有较多行代码的大函数。

2.11 匿名函数^{C++ 11}

1. 匿名函数的基本概念和定义方法

C++ 11 标准提出了一种称为 **Lambda** 的表达式，实质上是一种基于模板（见第 7 章）的匿名函数（[anonymous function](#)），通常以简短的、类似于表达式的形式出现在程序语句中。匿名函数同普通函数一样具有形参表、返回类型和函数体，但它没有函数名，并且在一条语句的内部就能够被定义和调用，定义形式如下：

```
[capture](parameters) [mutable][->return_type] {statement} // L1
```

其中，`capture` 是匿名函数的捕获列表，`parameters` 是函数的形参表，`->return_type` 是函数返回类型，`statement` 是函数体，可选项 `mutable` 用于指示需要修改“捕获的值变量”。例如：

```
int a = 1, b = 2;
cout<<[a](int c) ->int {return a + c; }(2)<<endl; // L2
```

在 `cout` 语句中定义并调用了匿名函数，该函数的返回类型为 `int`，它接受 1 个 `int` 参数 `c`，并将 `c` 与捕获的 `a` 的值相加。语句中“(2)”之前是匿名函数的定义，有了“(2)”就会调用执行该函数，它将实参 2 传递给形参 `c`。因此，`cout` 语句将输出函数执行结果值 3。

语句 L2 中如果没有 `endl` 前的“(2)”，就只定义了匿名函数，这个定义没有问题。但是，`cout` 语句就有错误了，毕竟它只能输出函数执行的结果而不是一个函数定义。

2. 匿名函数的返回类型、形式参数表、泛型和命名

匿名函数的返回类型采用了置尾设置方式，即用“->”在函数形参表和函数体之间指明返回类型，上面语句 L1 中的“->return_type”就是匿名函数返回类型的定义形式。如果没有指定匿名函数的返回值类型，编译器也能够根据 `return` 表达式的结果推导出来。如果没有 `return` 语句，编译器就会推导其返回类型为 `void`。因此，语句 L2 与如下语句等价：

```
cout<<[a](int c) { return a + c; }(2)<<endl; // L3
```

语句 L1 中的“(parameters)”是匿名函数的形式参数表，其用法与普通函数的形参表相同，如果不需要参数传递，就可以连同“()”一起省略。

C++ 11 标准可以自动推导匿名函数的返回类型，但参数类型必须明确指定。C++ 14 标准修改了这个限制，匿名函数的返回类型、参数类型都可以是 `auto` 类型，由编译器自动推导出来。例如，在支持 C++ 14 及以上标准的编译器中，求两数最大值的匿名函数可以设计为：

```
[](auto a, auto b) { return a > b ? a : b; } // C++ 14 及以上标准
```

这实质上是一个求两数最大值的泛型匿名函数，凡是能够用“>”运算符比较出大小的两个数，不管其类型是否相同，都可以用这个匿名函数求出最大值。例如：

```
cout<<[](auto a, auto b){ return a>b?a:b; }(184.9,'a') << endl; // 输出 184.9
cout<<[](auto a, auto b){ return a>b?a:b; }(84.9, 'a') << endl; // 输出 97
cout<<[](auto a, auto b){ return a>b?a:b; }(4.9, 3.2) << endl; // 输出 4.9
cout<<[](auto a, auto b){ return a>b?a:b; }(9, 10) << endl; // 输出 10
```

由此可知，由于可以用 `auto` 自动推导参数的类型，一个匿名函数相当于针对不同参数类型（包括两种不同类型的组合）的许多匿名函数，这样的函数就被称为**泛型函数**。

对于需要多次使用的匿名函数，可以为它定义一个变量名（相当于函数名）。例如，可以

为求两数最大值的匿名函数定义一个名称 `lmax` (相当于函数名), 则上面的代码可以写成如下形式。

```
auto lmax = [](auto a, auto b) { return a > b ? a : b; };
cout<<lmax(184.9, 'a')<<endl;           // 输出 184.9
cout<<lmax(84.9, 'a')<<endl;           // 输出 97
cout<<lmax(4.9, 3.2)<<endl;             // 输出 4.9
cout<<lmax(9, 10)<<endl;                 // 输出 10
```

3. 捕获外部变量

匿名函数的定义从一对 “[]” 开始, 称为 **闭包**, 用于捕获要用到的外部变量 (也称父变量, 即匿名函数定义语句所在作用域中的变量)。如果匿名函数要使用外部变量, 就将它们写在 “[]” 中; 如果不需要任何变量, 就保留空 “[]”。总之, 匿名函数必须以 “[]” 开头, 并且在它的函数体中只能使用被捕获的外部变量, 那些未被捕获的外部变量则不能够被使用。

1) 按值捕获外部变量

匿名函数捕获外部变量的方式类似于函数参数传递, 也有值捕获和引用捕获两种。值捕获就是将外部变量名直接写在匿名函数开头的 “[]” 中, 变量之间用逗号间隔。C++ 会将按值捕获的变量值复制给匿名函数使用, 与普通函数的值形参传递方式的区别在于, 被捕获的值参数具有 `const` 特征, 在匿名函数体内不能够修改它的值。例如:

```
int a = 1, b = 1;
cout<<[a]{ return a + 8; }()<<endl;    // L1, 正确
cout<<[b]{ return ++b + 8; }()<<endl;    // L2, 错误
```

语句 L1 定义了无参匿名函数, 以值方式捕获了外部变量 `a`, 并在匿名函数体中执行了 `a+8` 的运算。匿名函数体右边的 “()” 则是调用该函数, 因为 `a` 被捕获时的值为 1, 所以语句 L1 将输出 9。

语句 L2 是错误的, 原因是外部变量 `b` 是按值捕获的, 在匿名函数中 `b` 具有 `const` 性质, 不能够在匿名函数体中执行自增运算: `++b`。如果确实要在匿名函数中修改按值捕获的变量, 可以在定义匿名函数时使用 `mutable` 选项, 其功能是取消值捕获变量的 `const` 特性, 这样就可以在匿名函数体中修改按值捕获的变量了。如果使用了 `mutable` 选项, 那么匿名函数的参数表不可以省略, 如果不需要参数, 也要写上 “()”。

注意, C++ 是通过复制方式向匿名函数传递按值捕获变量值的, 相当于在匿名函数内部建立了一个同名变量, 并将外部变量的值复制给该同名变量, 复制完成后, 两者就没有关系了。因此, 即使匿名函数通过 `mutable` 选项修改了捕获的值变量, 也不会影响对应的外部变量值。例如, 用 `mutable` 设置上面的语句 L2, 在匿名函数中实现 `b+1` 运算, 该自增运算对外部变量 `b` 没有任何影响。

```
int a = 1, b = 1;
cout<<[b]() mutable { return ++b + 8; }()<<endl;    // L3, 正确, 输出 10
cout<<"b="<<b<<endl;                                // L4, 输出: b=1
```

还要注意的, 匿名函数在定义时就复制了外部变量 (按值捕获) 的值。因此, 若在匿名函数定义语句之后修改外部变量的值, 是不会对匿名函数产生影响的。例如:

```
int a = 1, b = 2;
auto fac = [a](int c) { return a + c; };              // L4, fac 捕获的 a 值固定为 1
```

```

a = 10; // L5, 对 fac 捕获的 a 无影响
cout<<fac(2); // L6, 输出结果: 3

```

语句 L4 定义了匿名函数，并为它定义了名称 fac。该匿名函数捕获了外部变量 a，接受一个整型参数 c，返回 int 类型数据，其功能是计算出 a+c 的和。语句 L6 通过名称 fac 调用匿名函数，传递 2 给实参 c，它将与定义时捕获的 a 的值 1 相加，因此 cout 语句将输出 3。由此可以看到，虽然语句 L5 修改了变量 a 的值，但它对 fac 定义时捕获的 a 值没有任何影响。

2) 按引用捕获外部变量

按引用捕获就是将外部变量的引用写在匿名函数开头的 “[]” 中，C++ 会在匿名函数中建立对应外部变量的引用。因此，当匿名函数修改引用捕获变量时，实际修改的是外部变量。如果匿名函数要修改外部变量的值，就需要用引用捕获方式。

【例 2-35】 匿名函数引用捕获对外部变量的修改影响。

```

// Eg2-35.cpp
#include<iostream>

void main() {
    int a = 1, b = 1;
    std::cout << [a, &b](int c) {
        std::cout<<"in lambda:\ta = "<<a<<"\tb = "<<b<<std::endl;
        return b += a + c;
    }(5)<<"\nin main:b = "<<b<<std::endl;
}

```

程序运行结果如下：

```

in lambda:   a = 1   b = 1 // 匿名函数内部输出结果
7           // return 表达式结果
in main:b = 7           // 匿名函数执行结束后的输出结果

```

在 cout 语句中定义了匿名函数，按值捕获了 a，按引用捕获了 b，并在 return 语句中将 b 的修改为 b+a+c 的结果，由于 a、b 的值都是 1，而传递参数 c 的是 5，因此 b 的值被修改为 7。程序运行结果表明，匿名函数对引用捕获变量的修改确实修改了对应的外部变量。

在同一个匿名函数中可以按值和按引用捕获多个外部变量，并且有一些默认的捕获描述标识方法，表 2-1 列出了匿名函数捕获外部变量的常用方法。

表 2-1 匿名函数捕获外部变量的常用方法

捕获列表	说 明	捕获列表	说 明
[]	不捕获任何外部变量	[=]	传值捕获所有外部变量
[&]	传引用捕获所有外部变量	[x, &y]	传值捕获 x，引用捕获 y
[&, x]	传值捕获 x，引用捕获其余变量	[=, &x]	引用捕获 x，传值捕获其余变量

注意：捕获列表中不允许重复捕获变量。例如，[=, a]是重复捕获，因为“=”指明了用值传递方式捕捉了所有变量，其中包括捕获 a，再次列出 a 就是重复捕获了；同样，[&, &this] 中的“&this”也是重复捕获。

3) 广义捕获外部变量

C++ 14 标准提出了“**初始捕获**”，即在匿名函数的闭包（开头的 []）中允许为捕获的外部变量指定一个新名称。初始捕获可以为外部变量或表达式指定一个新名称（其作用域限于匿

名函数体中)，这样就能够将表达式的计算结果绑定到新名称。这种通过闭包中定义的新名称捕获外部变量或表达式的方式称为**广义捕获**。

【例 2-36】 匿名函数广义捕获外部变量和表达式的示例。

```
// Eg2-36.cpp
#include<iostream>

void main() {
    int x = 4; // L1
    auto y = [&r = x, xx = x + 1] { // L2
        r += 2; // L3
        std::cout<<"r = "<<r<<"\txx = "<<xx<<std::endl; // L4
        return xx + 2; // L5
    }; // L6
    std::cout<<y(); // L7
    std::cout<<"\nx = "<<x<<std::endl; // L8
}
```

程序运行结果如下：

```
r = 6    xx = 5
7
x = 6
```

语句 L2~L6 定义了一个匿名函数，并用变量名 y 代表它。语句 L2 的闭包中定义了两个只能在匿名函数中应用的局部变量 r 和 xx。其中，r 是外部变量 x 的引用名称 r，因此语句 L3 对 r 加 2 的修改实际上就是对外部变量 x 的修改，所以语句 L4 输出的 r（实际上就是 x 的值）和 L8 输出的 x 均为 6。xx 是表达式“x+1”的名称，由于 x=4，因此 xx 的值为 5，语句 L4 的输出结果“xx=5”就是由此得来的。

语句 L5 通过 xx 引用了表达式“x+1”的值 5，所以“return xx+2”返回计算结果 7 作为匿名函数的计算结果。因此，语句 L7 输出的匿名函数调用结果值为 7。

注意：在初始捕获的闭包中定义表达式的名称时只能引用外部变量名，引用闭包中前面定义的同域变量是错误的。例如：

```
auto y = [&r = x, xx = r + 1]{ r += 2; return xx + 2; };
```

这个匿名函数中 xx 的定义是错误的，虽然 r 是外部变量 x 的引用，但它和 xx 是闭包内的同域变量。

以前的 C++ 标准并没有广义捕获方式，是 C++ 14 标准才提出的。广义捕获能够提高匿名函数与外部程序的交互能力，实现更加灵活、更加强大的功能。

4. 匿名函数的应用

匿名函数可以出现在程序语句的各位置，如运算表达式、函数参数表、循环语句、输出语句中，也可以是独立的函数调用形式，使用灵活，可以用简洁代码设计出功能强大的程序。

C++ 标准库（STL）中的许多数据结构和算法，如集合（set）、映射（map）、链表（list）、排序算法（sort）、合并算法（merge）等，初学者通常能够运用它们对系统的内置数据类型（如 double、char、int 等）进行存储或排序，但在用它们处理自定义的 struct、class 却遇到这样那样的问题。原因是这些数据结构和算法需要对存取的数据进行大小比较后按顺序存储，内置

数据类型的大小比较方法是明确的，STL 算法会自动处理，但并不知道用户自定义数据类型的大小比较方法，需要用户提供。运用匿名函数向上述数据结构或算法提供自定义数据类型的大小比较方法，可使问题轻松解决，下面举一个简单的例子。

C++的 `algorithm` 头文件中有一个具有三个形参的通用排序算法函数 `sort()`，其用法如下：

```
sort(begin, end [, cmp])
```

`sort()`对 `begin` 和 `end` 指定的数据区间（如数组、链表中任意两个位置之间的数据等）进行排序，第 3 个参数是一个指定排序方式的函数名称，该函数返回 `bool` 类型。对于内置数据类型，省略第 3 个参数时按升序（执行小于比较函数），如按降序排列，就需要提供比较函数。

【例 2-37】 用 STL 的函数 `sort()`对 `double` 数组排序，并通过函数和匿名函数指定排序方式。

```
// Eg2-37.cpp
#include <iostream>
#include <algorithm>
using namespace std;

bool doublecmpare(double x1, double x2) { return x1 < x2; }           // L1

void main() {
    double a[] = {23, 10, -4, 9, 34, 50};
    sort(a, a + 6);                                                  // L2
    for (auto x : a)
        cout<<x<<"\t";
    cout<<endl;
    sort(a, a + 6, [](double x, double y) { return x > y; });       // L3
    for (auto x : a)
        cout<<x<<"\t";
    cout<<endl;
    sort(a, a + 6, doublecmpare);                                    // L4
    for (auto x : a)
        cout<<x<<"\t";
    cout << endl;
}
```

程序运行结果如下：

-4	9	10	23	34	50
50	34	23	10	9	-4
-4	9	10	23	34	50

第 1 行输出是语句 L1 通过 `sort()`的默认方式（实施小于比较，即升序）排序后的结果，第 2 行输出是语句 L3 通匿名函数设置 `sort()`实施大于比较，按照从大到小的方式排序后的输出结果。第 3 行输出是语句 L4 通过普通比较函数设置了 `sort()`实施小于比较、按升序排序后的输出结果。

【例 2-38】 设有工人类的结构 `Worker` 包括姓名、年龄和工资属性，用 STL 的函数 `sort()`对 `Worker` 的数组排序，实现按姓名升序排序、按工资降序排序。

```
// Eg2-38.cpp
#include<iostream>
#include<string.h>
#include<algorithm >
```

```

using namespace std;

struct Worker {
    char name[10];
    int age;
    double salary;
    void outdata() { cout << name << "\t" << age << "\t" << salary << endl; } // L1
};

bool nameCmp(Worker a, Worker b) { return strcmp(a.name, b.name) < 0; } // L2

void main() {
    Worker w[] = {"tom",34,3500.0}, {"jack",40,5321.8}, {"anje",28,4898.1};
    cout<<"-----排序前-----" << endl;
    for(Worker x : w)
        x.outdata();
    cout<<endl<< "-----按姓名升序-----" << endl ;
    sort(w, w + 3, [](Worker a, Worker b){ return strcmp(a.name, b.name) < 0; }); // L3
    // sort(w, w+3, nameCmp); // L4
    for (Worker x : w)
        x.outdata();
    cout<<endl<< "-----按工资降序-----" << endl;
    sort(w, w + 3, [](Worker a, Worker b) { return a.salary > b.salary; }); // L5
    for(Worker x : w)
        x.outdata();
    cout<<endl;
}

```

程序运行结果如下：

```

-----排序前-----
tom    34    3500
jack    40    5321.8
anje    28    4898.1

-----按姓名升序-----
anje    28    4898.1
jack    40    5321.8
tom     34    3500

-----按工资降序-----
jack    40    5321.8
anje    28    4898.1
tom     34    3500

```

语句 L1 利用 C++语言对 C 语言的结构体的扩展特性，在结构体内部定义了输出函数 outdata()（详细讨论见第 3 章），以便精简程序代码，语句 L2 定义了两个 Worker 对象进行小于比较的方法，若用被注释掉的语句 L4 替换语句 L3 对 w 数组进行按姓名的排序，则程序结果完全相同。

语句 L3 和 L5 通过匿名函数为函数 sort() 设置了进行两个 Worker 对象进行大小比较的方法，前者按 name 从小到大排列，后者按照 salary 从大到小排列，程序运行结果确实是这样排列的。

本例示范了匿名函数的灵活性，在应用 STL 处理自定义数据类型时，通过匿名函数向算法或容器提供必要的参数，往往能够简化问题，提高编程效率。

2.12 命名空间

1. 命名空间的概念

在程序设计时，要求同一程序在全局作用域中声明的每个变量、函数、类型、常量等都必须具有唯一的名称，如有重复，就会产生命名冲突。程序员不一定对系统的全部库函数名称和全局变量符号都熟悉，容易定义与系统已有名称重复的变量名。另外，如果一个程序由许多程序员共同编写，彼此并不知道对方定义的标识符名称，同名在所难免。诸如此类原因还有很多，如在程序中引入另一个系统或第三方软件商提供的库文件，它们定义的全局名称（如全局变量、函数、类型等的名称）也容易与当前程序的已有名称相同。上述情况引发的名字冲突问题称为[全局命名空间污染问题](#)，处理起来并不容易，在大型程序中尤其困难，C++ 引入了命名空间来解决此问题。

[命名空间](#)就是每个程序员或每个不同的函数库各自独立地定义的一个名称，将自己设计的全部对象（包括变量、函数、类型、类等）都包含在此名称之下。这样，每个变量的全名就是“命名空间::对象名称”，只要命名空间不同名，就能够有效地区分程序中的同名变量。

2. 命名空间的定义

定义命名空间的关键字是 `namespace`，其语法格式如下：

```
namespace name {  
    members;  
}
```

其中，`name` 是命名空间的名称，只要是一个合法的 C++ 标识符都可以；`members` 是命名空间中包括的成员，可以是变量、函数声明、函数定义、结构声明、类的声明等。

【例 2-39】 设计命名空间 ABC，其中包括计数器、学生结构、类型定义和简单函数。

```
// Eg2-39.cpp  
namespace ABC {  
    int count;  
    typedef float house_price;  
    struct student {  
        char *name;  
        int age;  
    };  
    double add(int a, int b) { return (double)a+b; }  
    inline int min(int a, int b);  
};  
int ABC::min(int a, int b) { return a>b?a:b; }
```

这里定义了一个命名空间 ABC，它有 5 个成员：`count`、`house_price`、`student`、`add()`和 `min()`，其中包括变量、结构、类型以及函数的声明或定义。

函数的定义有两种方式：在命名空间内部定义，如 `add()`；也可以在命名空间外部定义，

如 `min()`。当在命名空间外定义时，要用“命名空间::”作为函数名的前缀，表示该函数是属于某个命名空间的成员，它的有效范围仅在此命名空间内。

3. 命名空间的应用

命名空间内的成员的作用域局限于命名空间内部，程序中可以通过作用域限定符“::”访问它，语法格式如下：

`namespace_name::identifier`

`identifier` 就是命名空间的成员名。例如，访问上面的 `ABC` 命名空间中的成员：

```
void main() {
    ABC::count = 1;           // 访问 ABC 空间中的 count
    int count = 9;           // main() 中定义的 count，与 ABC 中的 count 无关
    ABC::student s;          // 用 ABC 空间中的 student 结构定义 s
    s.age = 9;
    int x = ABC::min(4,5);    // 调用 ABC 中的 min() 函数计算两数最小值
}
```

另外，可以用 `using` 命令将命名空间中的成员引入当前程序，以简化命名空间的使用。`using` 有下面两种使用方式。

① 引用命名空间的单个成员。语法格式如下：

`using namespace_name::identifier`

例如，用 `using` 简化 `ABC` 命名空间中 `count` 的使用：

```
void main() {
    using ABC::count;         // L1
    count = 2;                // L2
    // int count = 9;         // L3
    .....
    count = count+2;          // L4
}
```

语句 `L1` 用 `using` 将 `ABC` 命名空间中的 `count` 变量引入了 `main()` 函数体，因此语句 `L3` 会产生 `count` 重复定义错误。若没有语句 `L1`，则语句 `L3` 是正确的，但语句 `L2` 和 `L4` 应该写成：

```
ABC::count = 2               // L2
ABC::count = ABC::count+2    // L4
```

② 引入命名空间的全部成员。语法格式如下：

`using namespace 命名空间名称`

例如，用 `using` 将 `ABC` 命名空间的全部成员引入 `main()` 函数体：

```
void main() {
    using namespace ABC;      // L1
    int count = 9;            // 错误，已有来源于 ABC 中的 count，重复定义
    student s;
    count = 5;
    s.age = min(43, 32);
}
```

语句 `L1` 用 `using` 语句将 `ABC` 命名空间中的全部成员引入 `main()` 函数体，因此可以直接使用来源于 `ABC` 中的任何成员。

命名空间为全局名称的冲突问题提供了一种解决方案。如果一个系统由多个文件组成，而每个文件由不同的开发商或程序员提供，那么开发商和程序员可以将他们定义的名称局限在自定义的命名空间中，即使大家都定义了相同的名称，也不会产生冲突。

2.13 变量

变量代表了一段可操作的内存空间，也可以认为变量是内存单元的符号化表示，是程序设计的重要元素，程序只有通过变量才能进行数据的存储和运算，实现各类业务数据的加工处理。

2.13.1 变量定义

变量代表一个有名称的、具有特定属性的存储单元，在其中可以存储数据，称为变量的值，变量的值是可以被修改的。想要在程序中获得变量，就必须用程序语言提供或定义的数据类型定义变量。数据类型决定了变量所占用存储单元的大小和变量值的范围，以及可以进行的运算。

在 C++ 中，可用基本数据类型（称为内置类型，如表 2-2 所示）、STL 的 `list`、`vector`、`set`、`map`、`stack` 等结构或类类型和用户自定义数据类型（主要是类）定义变量。在通常情况下，将用基本类型定义的称为变量，用 STL 的数据结构和类类型定义的称为对象。

表 2-2 C++ 的基本数据类型

类型	char	short	int	long	long long	float	double	long double	bool
名称	字符	短整数	整数	长整数	长长整数	浮点数	双精度	长双精度数	布尔

表 2-2 中的前 5 种类型都属于整数类型，都可以用 `signed` 和 `unsigned` 限定，其表示的数据范围从左至右逐步增大。例如，如下语句分别定义了变量 `x`、`y`、`d` 和对象 `s`。

```
int x;
unsigned int y;
double d;
stack<int> s;
```

2.13.2 作用域

作用域是指标识符在程序中可用的有效范围。这里的标识符包括变量名、函数名、常量名、对象名、语句标号、宏名等。在 C++ 中，作用域通常分为全局作用域、局部作用域和文件作用域；还有一种更细的划分方法，按照作用域范围，从大到小分为程序作用域、文件作用域、类作用域、函数作用域和块作用域。

① 程序作用域：指一个标识符在整个程序范围内有效。若一个程序由多个文件组成，具有这种作用域的标识符可以在该程序的各文件中应用。具有程序作用域的标识符只能在某文件中定义一次，其他文件若要用它，必须在文件中用 `extern` 声明。例如，若有 10 个文件都要用到变量 `x`，则 `x` 只需在一个文件中定义，在其他 9 个文件中用 `extern` 声明后就能使用了。

② 文件作用域：在一个文件中所有函数之外定义的标识符（包括函数名）和静态变量（包

括函数内的局部静态变量)具有文件作用域,其有效范围为从定义它的语句位置开始,直到文件结束。具有文件作用域的标识符只能够在定义它的文件中使用,不能在组成同一程序的其他文件中使用,也不会与其他文件中的同名标识符发生命名冲突。

③ 函数作用域:在函数范围内有效的标识符,包括函数的形参和函数内部定义的变量。

④ 块作用域。写在“{}”中的一条或多条语句就构成了一个语句块,在其中定义的标识符就只能在这对“{}”中使用,而且只在定义它的语句位置到离它最近的“}”之间有效,即只能在这段代码区域内引用它,这就是块作用域。

在函数中,一旦在当前作用域中找到了需要的名称,编译器就会忽略外层作用域中的同名变量。也就是说,若局部变量和某个全局变量同名,局部变量名会隐藏全局变量名。在这种情况下,可以通过作用域限定符“::”存取全局变量的值。

【例 2-40】 块作用域及作用域限定符的应用。

```
// Eg2-40.cpp
int i;                                // L1
int f() {
    int i;                            // L2
    i = 1;                            // 修改 L2 定义的 i
    ::i = 0;                          // 修改 L1 定义的 i
    {
        int j = 0;                   // L3
        static int k;                // L4
        i = 2;                       // 修改 L2 定义的 i
        ::i = 3;                     // 修改 L1 定义的 i
    }                                // j、k 的作用域到此结束
    j = 2;                           // 错误, j 已失去无定义
    return k;                         // 错误, k 已失去作用域
}
```

在本例中,最外层的 *i* 和函数名 *f* 具有文件作用域,可在整个文件中应用。函数 *f()* 内层的 *i*、*j*、*k* 具有块作用域,只能在包含它的最近一对“{}”内有效。

if、*switch*、*for* 和 *while* 之类的复合语句也是一种块语句,在其中(包括在其条件测试语句中)定义的变量名具有块作用域,其有效范围是该语句本身。

【例 2-41】 在 *if* 语句中定义的变量的作用域示例。

假设在 *if* 之前没有 *i* 和 *p* 的任何说明和定义。

```
// Eg2-41.cpp
if(int i = 5) {                       // i 作用域自此开始
    int p = 0;                        // p 的作用域自此开始
}                                     // p 的作用域到此结束
else {
    i = 1;
    p = 2;                            // 错误, p 无定义
}                                     // i 的作用域到此结束
```

在 *switch* 中定义的变量的作用域示例如下:

```
void f(int i) {
    switch(int j = i) {                // j 的作用域开始于此
```

```

        case 1:    j = j+1;
        case 2:
        .....
        case 3:    cout<<j;
    }
    cout<<j<<endl;
}

```

// j 的作用域到此结束
// 错误, j 已无定义

对于 **for** 和 **while** 循环语句, 标准 C++ 规定在其循环测试条件中定义的变量, 其作用域也限于循环本身, 即结束于循环体结束的 “}”。按此标准, 如下程序存在错误。

```

void f1(int z) {
    for(int i = 0; i < z; i++) {
        int j = i;
        cout<<i*j<<endl;
    }
    cout<<i<<endl;
}

```

// i 的作用域到此结束
// 错误, i 已无定义

但在某些 C++ 编译器中 (如 VC 6.0), 这段代码能够正确编译和运行, 原因是在标准 C++ 之前, 上述 **for** 循环是按如下方式处理的:

```

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

```

2.13.3 变量的类型和生命期

堆区
栈区
全局数据区
程序代码区

根据变量的作用域范围, 变量可以分为全局变量和局部变量两大类型。在函数内部定义的变量是**局部变量** (包括函数参数), 它们只能在定义它的函数中使用; 在函数之外且不在任何一对 “{}” 中定义的变量 (不属于任何函数) 是**全局变量**, 其有效范围从其在文件中的定义位置开始到文件结束。

变量的生命期是指变量在内存中存在的时间, 与变量所在的内存区域有关。为了更清楚地理解这个问题, 先看看运行程序对内存的使用情况。

程序在其运行期间内, 程序代码和数据被分别存储在 4 个内存区域中,

图 2-9 内存区域 如图 2-9 所示。

- ❖ 程序代码区: 程序代码 (即程序的各函数代码) 的存放区域。
- ❖ 全局数据区: 程序的全局数据 (如全局变量) 和静态数据的存放区域。
- ❖ 栈区: 程序的局部数据 (在函数中定义的变量) 的存放区域。
- ❖ 堆区: 程序的动态数据 (**new**、**malloc** 均在此区域中分配存储空间) 的存放区域。

全局数据区也称为静态存储区, 其特点是: 在编译时为其中的变量分配内存空间并进行初始化。对于定义时没有提供初始值的变量, 系统会自动将其初始化为 0。在程序运行期间, 这个区域中的变量一直存在, 直到程序结束才由系统负责回收变量对应的内存空间。

堆区的数据由程序员管理, 程序员可以用 **new** 或 **malloc** 分配其中的存储单元给指针变量, 用完之后, 由程序员用 **delete** 或 **free** 将其归还系统后, 其他程序才能再次使用。如果用完后, 没有用 **delete** 或 **free** 回收该内存空间, 它将一直被占用, 其他程序无法再次使用, 类似于 “堆

满了垃圾的屋子，别人无法进入”，这是 C 和 C++ 程序中最常发生的问题，称为“内存泄漏”。

在函数中定义的局部变量（除了静态类型的局部变量，这类变量在全局数据区中），只有当函数被调用时，系统才会为函数建立堆栈，并在其中为函数定义的局部变量分配存储空间，但不会对分配的存储单元做初始化工作。一旦函数调用结束，系统就会回收局部变量在栈区中的存储单元。

全局变量和静态变量存储在全局数据区中，具有较长的生命期，在程序运行的整个期间都有效。非静态的局部变量存储在栈区，其生命期较短，只在函数调用期间有效。

静态变量可分为静态全局变量和静态局部变量，二者都保存在全局数据区，具有相同的生命期。但二者的作用域不同，前者的作用域是整个程序范围，后者的作用域局限于定义它的语句块。静态局部变量和普通局部变量相比，二者具有相同的作用域和不同的生命期。静态局部变量与全局变量有着同样长的生命期，即程序结束时它才会被释放。普通局部变量的生命期只有函数调用期间才存在，函数调用完成后就结束了。

【例 2-42】 静态变量的生存期长于其作用域的示例。

```
// Eg2-42.cpp
#include <iostream>
using namespace std;

static int n;                                // n 被初始化为 0
void f() {
    static int i;                             // i 被初始化为 0
    int j = 0;
    i += 2;
    j += 2;
    cout<<"i = "<<i<<" ";
    cout<<"j = "<<j<<endl;
}
void main() {
    n += 5;
    f();
    // i = 2;
    f();
}
```

// 输出 i = 2, j = 2;
// 错误, i 虽为 static, 但其作用域为函数 f() 内部
// 输出 i = 4, j = 2;
// i、n 的生命期到此才结束

第 1 次调用函数 f() 后，因为 i 为静态变量，虽然失去了作用域（这就是 i=2 错误的原因），但未失去其生存期（占据的内存未被系统回收），第 2 次调用函数 f() 时，将直接在 i 对应的存储器中加 2，所以结果是 4。而 j 是普通局部变量，第 1 次调用函数 f() 后，其作用域和生存期都结束了，第 2 次调用又重新开始定义它，所以两次调用函数 f()，j 的输出都是 2。

2.13.4 变量初始化

1. 变量初始化及 C++ 89/03 标准的初始化列表

变量初始化是指当变量在被创建时就为它指定一个初值。初值可以是一个简单的常量或任意复杂的表达式。同时定义多个变量时可用前面定义的变量初始化后面定义的变量。例如：

```
int i = 10, j = i*10;
```

初始化列表是指在一对“{}”中放入初始值并赋予变量的方式，有如下两种常见形式：

```
int a={1}; // 初始化列表形式 1, C++ 98/01, C++ 11
int x{0}; // 初始化列表形式 2, C++11
```

对于 C++ 89/03 标准，只支持形式 1 的初始化式，且始化列表只适用于 POD (Plain Old Data) 类型的数据结构。其中，Plain 代表它是一个普通类型，Old 代表它与 C 语言兼容的旧数据类型。比如，int、float，以及能用 C 语言的 memcpy() 等函数进行操作的类、结构体，就是 POD 类型的数据。在传统的 C 及 C++ 程序设计中，初始化列表通常用于数组和结构体变量的初始化。

【例 2-43】 C++ 中的老式初始化列表，在 VC 6.0 中可以编译执行，但在 VC 2022 中有编译错误。

```
// Eg2-43.cpp
#include <iostream>
using namespace std;

struct A {
    int x;
    struct B
    {
        int i;
        int j;
    } b;
};

int main(int argc, char** argv) {
    int arr[3] = {1, 2, 3}; // L1, 普通数组
    A a = {1, {2, 3}}; // L2, POD 结构
    int i = {9.6}; // L3, 在 C++ 11 中是错的，原因见后
    cout<<arr[0]<<"\t"<<arr[1]<<"\t"<<arr[2]<<endl;
    cout<<a.x<<"\t"<<a.b.i<<"\t"<<a.b.j<<endl;
    cout<<"i = "<<i<<endl;
    return 0;
}
```

语句 L1、L2、L3 采用了初始化列表方式分别对数组、结构和 int 类型的变量进行了初始化。注意，语句 L3 中初始化值 9.6 的类型为 double，与其初化的 int 型变量 i 的类型是不匹配的。在 VC 6.0 中，编译运行的结果如下：

```
1      2      3
1      2      3
i = 9
```

2. C++ 11 的统一初始化列表

变量的初始化方法很多，如下方法都是正确的：

```
int x = 0;
int x(0);
int x = {0.5}; // L4, 在 C++ 98/03 中正确，在 C++ 11 中错误
```

不要把“`int x = 0;`”等同于如下语句组，它们是不同的。

```
int x;  
x = 0;
```

虽然 `x` 的最终值都是 10，但“`x = 0`”是赋值语句，可以理解为：先除掉 `x` 对应内存单元中的值，再写入 0；而“`int x = 0`”没有这个过程，是在为 `x` 分配内存单元的同时就写入 0。

虽然方法不少，但是每种方法只适用于某些类型变量的初始化，并没有某种初始化方法能够适用于所有类型的变量。C++ 11 标准对变量的初始化列表进行了改进，称为统一初始化列表（uniform initialization），与 C++ 98/03 标准中的变量初始化列表相比，至少有以下区别。

① 在 C++98/03 标准中，仅部分类型的变量才允许使用初始化列表（如不能初始化 `int`、`float` 和 `double` 等），而 C++ 11 中的初始化列表适用于所有变量类型的初始化。

② 在 C++ 11 标准中，初始化列表中的 `{ }` 除了用于变量初始化，还可以用于对变量赋值。例如，下面两条语句在 VC 6.0 中都是错误的（老标准），但在 VC 2022 中是正确的。

```
int x{0}; // C++ 11  
y = {45}; // C++ 11
```

③ 对初始化列表的精度要求不同。C++ 98/03 标准中的初始化列表可以是窄化（narrowing initialization）的，即当初始化列表中的值与变量类型不匹配时，会将列表值转换为变量的类型。但是，C++ 11 标准中的初始化列表则是为非窄化的初始化（non-narrowing initialization），要求列表值的数据类型与初始化变量的类型严格匹配，如果两者的类型不相同，就不会进行有损精度（窄化）的类型转换，而会产生类型不匹配的编译错误。

```
int p1 = 3.14; // 正确，编译器执行了窄化操作，有数据损失  
int p2 = {3.14}; // 错误，列表指示编译器需要阻止窄化操作  
unsigned u1 = -1; // 正确，编译器执行了窄化操作，有数据损失  
unsigned u2 = {-1}; // 窄化错误，不允许负值
```

在使用字面常量值作为初始化列表值时，编译器会检查该常量值能否被目标类型准确表示，如果能够准确表示就允许通过编译，如果不能准确表示（存在数据损失），就产生编译错误。

```
double d = {3.159}; // L1, 正确  
float f1 = {3.159}; // L2, 正确  
float f2 = {d}; // L3, 窄化错误
```

`f1` 和 `f2` 的初始化列表似乎完全相同的，为什么 `f1` 的初始化正确，而 `f2` 的初始化是错误的呢？这就是上面所讲的道理。`double` 类型的字面常量 3.159 能够在无数据损失（或损失在允许范围内）的情况下转换成 `float` 类型，因此是允许的。但是，`f2` 中的 `d` 则不一样，虽然 `d` 的当前值也为 3.159，但也可能是其他值，因此必须考虑 `double` 类型（`d` 的类型）中的所有值，是否都能够无损失地转换为 `float`，显然无法保证这一点，因此 `f2` 的初始化列表是错误的。

无符号整数与普通整数之间的转换也会产生窄化错误，因为 `signed int`、`int` 和 `unsigned int` 虽然表示的数字个数大致相同，但其所表达的值域并不相同。

```
int i1 = {3}; // L4, 正确  
unsigned u1 = {4}; // L5, 正确  
unsigned u2 = {i1}; // L6, 窄化错误，不支持负值（i1 可以取负值）  
int i2 = {u1}; // L7, 窄化错误，会超出最大数
```

总之，C++ 11 初始化列表通过禁止下列转换（对隐式转换加以限制），以实现无损失的变量初始化：① 从浮点类型到整数类型的转换；② 从 long double 到 double、float 的转换，以及从 double 到 float 的转换；③ 从整数类型到浮点类型的转换；④ 从整数或无作用域枚举类型到不能表示原类型所有值的整数类型的转换（如语句 L6、L7）。但是，对于②～④而言，如果来源是常量表达式并且其值能完全存储于目标类型变量中，就是允许的，上面的语句 L2 和 L5 就属于这种情况。

3. 变量初始化的基本原则

变量初始化的默认规则是：如果定义变量时提供了初始值表达式，系统就用这个表达式的值作为变量的初值；如果定义变量时没有为它提供初值，那么全局数据区中的变量将被系统自动初始化为 0，栈和堆中的变量不被初始化（其值可能是该区域分配给变量之前就有的值，也可能是其他未知值）。

全局变量、命名空间的变量、静态变量会被保存在全局数据区中，所以它们会被系统自动初始化为 0；局部变量（也叫自动变量）被存储在栈区中，动态分配的变量（用 malloc 和 new 建立）被存储在堆区中，它们都不会被系统用默认值初始化。

【例 2-44】 全局变量、静态变量、局部变量的初始化。

```
// Eg2-44.cpp
#include <iostream>
using namespace std;

int n;                                     // 初始化为 0
void f(){
    static int i;                         // 初始化为 0
    int j;                                // 不被初始化, j 值未知
    cout<<"i = "<<i<<" ";
    cout<<"j = "<<j<<endl;
}

int *p1;                                  // p1 被初始为 0

void main() {
    int *p2;                              // p2 不被初始化, 值未知
    int m;                                // m 不被初始化, 值未知
    f();                                  // 输出 i=0, j=?, ?表示不确定值
    cout<<"n = "<<n<<endl;               // 输出 n=0
    cout<<"m = "<<m<<endl;               // 输出 m=?, ?表示不确定值
    if(p1)
        cout<<"p1 = "<<p1<<endl;         // p1=0, 无输出
    if(p2)
        cout<<"p2 = "<<p2<<endl;         // 输出 p2=?, ?表示不确定地址
}
```

对变量初始化应当引起足够的重视，未被初始化的局部变量拥有一个不确定的值，它是引起程序运行错误的重要原因之一，并且不易查错。有些编译器会对此提出警告，有些编译器则会出现运行错误。比如，本例在 VC 6.0 中会提出编译警告但程序可正常运行，在 VS 2015 中会出现运行时错误，而在 VS 2022 中会出现变量未初始化的编译错误。

2.13.5 局部变量与函数返回地址

弄清楚局部变量的存储方式和生命期后，用指针或引用从函数中返回一个地址时要小心，一定不要返回函数内部定义的局部变量的指针或引用，这样的指针被称为“悬挂指针”或“失效引用”。因为当该函数调用结束后，返回的指针或引用仍然可能会被主调函数使用，但其对应的局部变量已经在函数调用结束时就失去了作用域和生命期，其对应的内存单元已经被系统回收了，有引发程序错误的极大可能，或者导致程序运行结果不正确。

【例 2-45】 函数 `f1()` 返回局部对象的引用，会产生不可预知的错误运行值。

```
// Eg2-45.cpp
#include<iostream>
using namespace std;

int &f1(int x) {
    int temp = x;
    return temp;
}

void main(){
    int &i = f1(3);
    cout<<i<<endl;
    cout<<i<<endl;
}
```

虽然在两条输出 `i` 的语句间没有其他语句，但两次输出的结果仍然可能不一致。下面是在 VC 6.0 中的输出结果：

```
3
4200045 // VS 2015 中的结果：1581570872
```

第 2 次输出的 4200045 只是一个随机值而已，就算是其他值也是可以理解的。原因很简单，`f1()` 函数返回了局部变量的 `temp` 的地址，函数调用结束后，这个地址就无效了，会再次把这个存储区域分配给谁，这个存储区域中的内容会被如何改写，将不得而知。

同样，如果函数的返回类型是指针，也不要返回局部变量的地址，否则会引发与本例同样的错误。如下函数 `f1()` 就存在这样的问题。

```
int *f1() {
    int temp = 1;
    return &temp;
}
```

2.14 预处理器

C++ 预处理器（也叫**预编译器**）提供了一些预处理命令，如 `#define`、`#else`、`#elif`、`#endif`、`#error`、`#if`、`#ifdef`、`#ifndef`、`#include`、`#line`、`#pragma`、`#undef` 等。这些命令在正式编译之前执行，所有的预处理命令都以“`#`”开头，独占一行，语句结束时不需要“`;`”。

1. `#define` 和 `#undef`

`#define` 常用于定义一个标识符常量或带参数的宏。例如：

```
#define pi          3.14159
#define MAX(a,b)    ((a)>(b) ? (a) : (b))
```

在对程序进行预编译时，C++会用“**#define**”定义的标识符常量的值替代常量名，用宏的代码替代宏名。例如，若对上述定义存在如下引用：

```
x = pi+5;
int y = MAX(9, 3);
```

在编译相应程序之前，如上两条语句将被预处理为如下形式：

```
x = 3.14159+5;
int y = ((9)>(3)?(9):(3));
```

#undef 用于删除由**#define** 定义的宏，使之不再起作用。例如：

```
#undef MAX
```

此命令之后，**MAX** 就不再有意义了。

需要注意的是，宏是一种应当尽量少用或者不用的老技术，通过将带参数的宏名替换成定义的文本，实现了代码复用功能。但宏缺乏类型概念，并且将自身与命名空间、作用域和其他程序语言要素独立开，可能带来各种诡异的程序错误。C++中的 **const** 常量、内联函数或 **constexpr** 常量表达式都是取代宏的更好方案。

2. 条件编译

条件编译指示编译器只对满足条件的语句或语句块进行编译，使同一程序在不同的编译条件下，能够得到不同的目标代码。常用的条件编译有以下两种形式。

1) 第 1 种形式

```
#ifdef 标识符
    语句组 1
[#else
    语句组 2
]
#endif
```

[]中的内容是可选项，既可以有**#else** 部分，也可以没有**#else** 部分。意思是，如果已经用**#define** 定义了某标识符，就编译语句组 1；否则编译语句组 2，当然前提是存在**#else** 部分。

2) 第 2 种形式

```
#ifndef 标识符
    语句组 1
[#else
    语句组 2
]
#endif
```

如果没有用**#define** 定义某标识符，就编译语句组 1，否则编译语句组 2。

#ifndef 有一种用于避免多次包含（**#include**）多次展开同一个头文件引发的重复代码错误技术，称为包含守卫（**include guard**），广泛应用于头文件设计中。其结构如下：

```
#ifndef xx_h
#define xx_h
    // 头文件代码
```

```
#endif
```

如果在同一个程序中多次包含（`#include`）了这种形式的头文件，只会展开 1 次（第 1 条 `#include` 语句），其余 `#include` 语句都不会展开了。

【例 2-46】 条件守卫头文件示例。

先创建一个空的 C++ 项目，从中添加一维数组相加、相乘的头文件 `varray.h`。

```
// varray.h
#ifndef varray_h
#define varray_h
#include <valarray>
#include <iostream>
using namespace std;

auto vecSum(valarray<int> v1, valarray<int> v2) {
    valarray<int> v3 = v1 + v2;
    return v3;
}

valarray<int> vecMul(valarray<int>v1, valarray<int>v2) {
    auto v3 = v1 * v2;
    return v3;
}

void displayVec(valarray<int>v) {
    for (int x : v)
        cout<<x<<"\t";
    cout<<endl;
}
```

然后，在该项目中添加一个 `useVarray.cpp` 文件。

```
// useVarray.cpp
#include"varray.h"
#include"varray.h"

int main() {
    valarray<int> v1{1,2,3,4 }, v2{5,6,7,8}, v3;
    v3 = vecSum(v1, v2);
    displayVec(v3);
    v3 = vecMul(v1, v2);
    displayVec(v3);
    v3 = v2 / v1+10;
    displayVec(v3);
}
```

程序中两次 “`#include"varray.h"`”，如果 `varray.h` 头文件没有用 `#ifndef` 定义守卫包含，将产生重定义编译错误。运行该程序，结果如下：

6	8	10	12
5	12	21	32
15	13	12	12

2.15 文件的输入和输出

程序与文件的数据交换方法同它与标准输入/输出设备的数据交换方法相同，从文件中读取数据与从键盘输入数据的方法相似，将数据写入文件与将数据输出到显示器的方法相似。但 `iostream` 中定义的数据类型和函数只能用于标准输入/输出设备的数据处理。

2.15.1 文件操作的基本流程

在头文件 `fstream` 中定义了两个类 `ifstream` 和 `ofstream`，类 `ifstream` 用于处理输入文件流，类 `ofstream` 用于处理输出文件流，可以进行文件操作，包括以下 5 个步骤。

<1> 在程序中包含头文件 `fstream`

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

<2> 定义文件流变量

```
ifstream inData;           // 定义输入文件流变量
ofstream outData;          // 定义输出文件流变量
```

<3> 将文件流变量与磁盘文件关联起来

```
inData.open(filename, mode)      // outData.open(filename, mode)
```

`filename` 是磁盘文件名，`mode` 是打开或建立文件的方式，可以是：

```
ios::in           // 打开输入文件，ifstream 类型变量的默认方式
ios::out          // 建立输出文件，ofstream 类型变量的默认方式
ios::app          // 增加方式，若文件存在，则在尾增加数据，否则建立文件
ios::trunc        // 若文件存在，则文件中已有内容将被清除
ios::nocreate     // 若文件不存在，则打开操作失败
ios::noreplace   // 若文件存在，则打开操作失败
```

例如，要打开目录 `C:\dk` 下的文件 `ab.txt`，若该文件存在，则打开，否则建立该文件。可以用如下命令建立：

```
ofstream outData;
outData.open("C:\\dk\\ab.txt", ios::app);
```

说明：由于“\”被 C++ 语言用于转义符，因此在指定文件路径时用“\\”作为文件路径中目录之间的间隔符，其中第一个“\”是转义符，与回车换行符“\n”中的“\”意义相同。

第<2>、<3>步也可以合并为一步。下面的命令与上面两条命令等价：

```
ofstream outData("C:\\dk\\ab.txt", ios::app);
```

<4> 用文件流（<<或>>）操作文件，读、写文件数据。将输入文件流变量与>>连接，就能够从文件中读入数据，与 `cin` 用法相同。将输出文件流变量与<<连接，就能够将数据输出到文件中，与 `cout` 用法相同。

<5> 关闭文件。文件操作完成后，应该关闭文件。关闭文件时，系统会立即将文件缓冲区中的数据写回磁盘文件，并断开文件流变量与磁盘文件之间的联系。关闭文件的方法如下：

```
inData.close();           // inData 是输入文件流变量
outData.close();          // outData 是输出文件流变量
```

【例 2-47】 建立文件 D:\data.txt，从键盘输入数据（23、34、56、78、98、23、32、89、12）到文件中，然后从该磁盘文件中将这些数据读出到数组 a 中，并计算其和。

```
// Eg2-47.cpp
#include<iostream>
#include<fstream>
using namespace std;

void main() {
    ofstream outData("D:\\data.txt");           // 在 D 盘根目录下建立文件 data.txt
    ifstream inData;                             // 定义 inData 为输入数据的文件
    int x, a[10];
    for(int i = 0; i < 10; i++) {
        cin>>x;
        outData<<x<<" ";                       // outData 将 x 写入文件 data.txt，数据间用空白间隔
    }
    outData.close();                             // 关闭文件 data.txt
    inData.open("D:\\data.txt");                 // 以输入方式打开 D:\data.txt 文件，以便从中读数据
    int j = 0;
    while(!inData.eof())                         // 从文件中读数据，直到遇到文件结束符
        inData>>a[j++];                         // 从文件中将数据读入到数组 a 中
    inData.close();                             // 关闭文件
    int s = 0;
    for(int i = 0; i < 10; i++) {
        s += a[i];
        cout<<a[i]<<" ";                       // 输出数组 a，该数组中的数据来源于文件
    }
    cout<<endl;
    cout<<"the sum is: "<<s<<endl;
}
```

2.15.2 输入流、输出流的泛化思想

C++的输入流、输出流并不局限于键盘、显示器和文件，而是一种泛型程序设计方法，任何继承于类 `istream`、`ostream` 或 `iostream` 并提供了相应实现的类都是一个流，都可以用来输入、输出数据或者建立文件，也可以应用 `istream`、`ostream` 或 `iostream` 的对象作为函数参数，写出具有通用功能的输入、输出函数。此外，C++标准库中还有一个字符串流类 `stringstream`，可以用任意可打印的字符创建一个字符串对象，该类的成员函数 `str()`能够返回流中的字符串。

【例 2-48】 设计一个输出函数 `writeString()`，用 `ostream` 对象作为参数，能够接受任意类型数据的输出流。

```
// Eg2-48.cpp
#include<iostream>
#include<sstream>
#include<fstream>
using namespace std;

void writeString(ostream &os) {
    os << "hello! this is a generalize function, can write screem, file ..... "<<endl;
```

```

}
int main() {
    ofstream myfile("D:\\gene.txt");
    stringstream sstring;
    writeString(cout);           // L1
    writeString(myfile);        // L2
    writeString(sstring);       // L3
    cout<<"From sstring: "<<sstring.str(); // L4
    return 0;
}

```

函数 `writeString()` 能够接受任何从 `ostream` 类型及其派生出的类对象为实参，如 `main()` 中的 `cout`、`myfile`、`sstring` 等，运行程序，输出的结果如下：

```
hello! this is a generalize function,can write screem, file .....
```

```
From sstring: hello! this is a generalize function, can write screem, file .....
```

语句 L1 中的函数 `writeString()` 将字符串输出到 `cout` 设备，所以在屏幕上输出了第 1 行字符串；语句 L2 在 D 盘创建了 `gene.txt` 文件，并将字符串“hello! ……”写入了文件；语句 L3 将字符串写入了字符串变量 `sstring`；语句 L4 将字符串变量的内容输出到屏幕，也就是运行结果的第 2 行。

2.16 编程实作：C++程序设计初步

【例 2-49】 有三名学生的姓名、学号以及数学、英语、计算机成绩如下：

李大海, s1601,87,56,97

王明志, s1602,87,89,78

张致新, s1603,98,76,88

将学生成绩保存到磁盘文件，然后从磁盘上读出学生成绩并计算每位同学的总分。

程序设计思路：分两步进行。① 在 D 盘建立输出数据文件 `student.dat`，然后通过循环从键盘输入学生数据；② 建立输入文件，从 `student.dat` 文件中读取每位同学的数据并计算总分，同时输出。

1. 编写读入学生成绩到文件中的源程序

启动 Visual C++ 2022，选择“文件 | 新建 | 项目 | 控制台项目 | C++”菜单命令，在弹出的“名称”对话框中输入项目文件名 `student`，指定文件目录；单击“解决方案资源管理器 | `student`”项目下的“源文件”列表，打开 `student` 源文件，输入如下代码。

```

// Eg2-49-1.cpp
#include <iostream>
#include <string>
#include <fstream>
using namespace std;

void main() {
    ofstream outfile("D:\\student.dat");
    string name, id;

```

```

int math, eng, computer;
for(int i = 0; i < 3; i++) {
    cout<<"输入姓名: ";
    cin>>name;
    cout<<"输入学号: ";
    cin>>id;
    cout<<"输入数学成绩: ";
    cin>>math;
    cout<<"输入英语成绩: ";
    cin>>eng;
    cout<<"输入计算机成绩: ";
    cin>>computer;
    outfile<<name<<" "<<id<<" "<<math<<" "<<eng<<" "<<computer<<endl;
}
outfile.close();
}

```

2. 运行程序，查看建立的文件

编译程序，修改程序中的错误，然后运行程序。根据屏幕提示，输入每个学生的各项数据，每输入一个数据后按一次 Enter 键。结束后，用 Windows 的记事本或 VC 编辑器打开建立的文件 “D:\student.dat”，观察文件内容，如图 2-10 所示，然后按相同格式增加 5 个学生的成绩在文件后面。

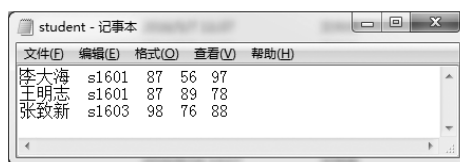


图 2-10 用文件建立的数据文件

3. 读取建立的文件数据

文件一经建立，就能多次使用，可以对它进行读取、修改、查找等操作。现在编写一程序，读出文件 student.dat 中的数据并显示。在 C++编辑器中输入、编译并运行如下程序，然后检查显示的数据是否与文件 D:\student.dat 中的数据一致。

```

// Eg2-49-2.cpp
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

void main() {
    ifstream infile("D:\\student.dat");
    string name, id;
    int math, eng, computer, sum = 0;
    cout<<"姓名\t" << "学号\t" << "数学\t" << "英语\t" << "计算机\t" <<"总分"<< endl;
    infile>>name;
    while(!infile.eof()) {
        infile>>id>>math>>eng>>computer;
        sum = math + eng + computer;
        cout<<name << "\t"<<id<<"\t"<<math<<"\t"<<eng<<"\t"<<computer<<"\t"<<sum<<endl;
        infile>>name;
        sum = 0;
    }
}

```

```
infile.close();  
}
```

为了对文件中的数据进行各种处理，常常将文件中的数据读取到数组、树、链表或栈之类的数据结构中。

习 题 2

- 2.1 C++语言的 struct、enum、union 与 C 语言的有何区别？
- 2.2 const、constexpr 与#define 有什么区别？如何区分顶层 const 和底层 const？
- 2.3 什么是数据类型转换？C++的数据类型转换有哪几种？在哪些情况下会发生隐式类型转换？
- 2.4 什么是函数重载？在函数调用时，C++是如何匹配重载函数的？
- 2.5 什么是引用？左值引用和右值引用有什么区别？使用它们时要注意哪些问题？
- 2.6 什么是内联函数？它有什么特点？哪些类型的函数不能被定义为内联函数？
- 2.7 Lambda 表达式是什么？它由哪几部分组成？值捕获、引用捕获和广义捕获有什么不同？
- 2.8 简述命名空间的概念。如何访问特定命名空间的成员？STD 命名空间是怎么回事？
- 2.9 C++的作用域有哪些类型？变量有哪些类型？变量类型与变量的初始化有什么关系？
- 2.10 指出下面代码的错误。

(1)

```
int &f1(int x = 0, int y) {  
    return x*y;  
}  
int *f2(int a, int b = 1) {  
    int t = a*b;  
    return &t;  
}  
int main() {  
    int i = 10;  
    const int r;  
    int &lr = i;  
    int &&rr = i;  
    rr = lr;  
    int &a, *p;  
    r = 10;  
    a = r;  
    char *c = "nonconst";  
    const char *pc1 = "dukang";  
    char *const pc2 = "dukang";  
    char const *pc3 = "dukang";  
    const char const *pc4 = "dukang";  
    pc1[2] = 't';  
    pc2[2] = 't';  
    pc3[2] = 't';  
    pc4[2] = 't';  
    pc1 = c;
```

```

    pc1 = pc2;
    pc1 = pc3;
    pc2 = pc4;
    pc3 = c;    cout<<f1(3);
    cout<<f2(2,3);
    return 0;
}

```

(2)

```

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

int main() {
    int i = 10;
    unique_ptr<int> ap1(new int(4)), ap2;
    ap2 = ap1;
    cout<<*ap2;
    cout<<*ap1<<endl;
    char *c;
    shared_ptr<char> sc;
    sc = c;
    sc = new char(10);
    return 0;
}

```

(3)

```

int main(){
    double d = 92.221, d1{92.221}, d3{d};
    int x = d;
    x = {32};
    d = {32};
    d = x;
    d = {x};
    int y = {d};
    return 0;
}

```

(4)

```

#include <iostream>
using namespace std;

void main() {
    int a = 1, b = 2;
    auto f1 = [](int c) mutable->int { return b += ++a + c; };
    auto f2 = [a, &b] (int c)->int { return b += ++a + c; };
    a = 4;
    b = 4;
    cout<<f1(4)<<"\ta = "<<f2(6)<<endl;
    cout<<"in main:\ta = "<<a<<"\tb = "<<b<<endl;
}

```

}

2.11 读程序，写出程序的执行结果。

(1)

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

int print(int i){ return i*i; }
double print(double d){ return 2*d; }
int main() {
    int a = 25;
    float b = 9.2;
    double d = 3.3;
    char c = 'a';
    short i = 3;
    long l = 9;
    cout<<print(a)<<endl<<print(b)<<endl<<print(d)<<endl;
    cout<<print(c)<<endl<<print(i)<<endl<<print(l)<<endl;
    return 0;
}
```

(2)

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

int n;
int *p1;
void fun(){
    static int a;
    int b;
    cout<<"a = "<<a<<" ";
    cout<<"b = "<<b<<endl;
}

int main() {
    int *p2;
    int m;
    fun();
    {
        int n(10), m(20);
        cout<<"n = "<<n<<endl<<"m = "<<m<<endl;
    }
    cout<<"n = "<<n<<endl<<"m = "<<m<<endl;
    if(p1)
        cout<<"p1 = "<<p1<<endl;
    if(p2)
        cout<<"p2 = "<<p2<<endl;
    return 0;
}
```

(3)

```

#include <iostream>
using namespace std;

double f(int a = 10, int b = 20, int c = 5){ return a*b*c; }
int main() {
    cout<<f()<<endl<<f(20)<<endl<<f(10, 10)<<endl<<f(10, 10, 10)<<endl;
    return 0;
}

```

(4)

```

#include <iostream>
using std::endl;
using std::cout;

int &f(int& a, int b = 20) {
    a = a*b;
    return a;
}

int main() {
    int j = 10;
    int &m = f(j);
    int *p = &m;
    cout<<j<<endl;
    m = 20;
    cout<<j<<endl;
    f(j, 5);
    cout<<j<<endl;
    *p = 300;
    cout<<j<<endl;
    return 0;
}

```

(5)

```

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

struct Node {
    int data;
    shared_ptr<Node> next;
};

int main() {
    int a[] = {3, 4, 1, 8, 9, 2, 7};
    shared_ptr<Node> list(new Node), p;
    list->data = 0;
    p = list;
    for(auto v : a) {
        shared_ptr<Node> q(new Node);
        q->data = v;
        p->next = q;
    }
}

```

```

        p = p-> next;
    }
    p->next = NULL;
    p = list->next;
    int s = 0;
    while(p) {
        cout<<p->data<<"\t";
        s += p->data;
        p = p->next;
    }
    cout<<"\ns = "<<s<<endl;
    return 0;
}

```

(6)

```

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

int main() {
    int s = 0;
    int a[] = {-1, -3, 0, 5, 8, -11};
    std::for_each(begin(a), end(a), [&a, &s](int &x) { x = abs(x); s += x; });
    auto Max = [](int a, int b) { return a > b ? a : b; };
    sort(a, a + 6, [](int a, double b) { return a>b; });
    int m = a[0];
    for(auto v : a) {
        cout << v << "\t";
        m = Max(m, v);
    }
    cout<<"m = "<<m<<"\ts = "<<s<<endl;
    return 0;
}

```

注：for_each 是 STL 的通用遍历算法，基本结构如下。

```

Function for_each(first, last, Function fn) {
    while (first != last) {
        fn (*first);
        ++first;
    }
}

```

其中，first、last 是一段数据区间的首、尾地址，Function 是一个函数名称。

(7)

```

#include <iostream>
using namespace std;

void main() {
    int x = 4;
    auto y = [&r = x, x = x + 1](int a, int b)

```

```

{
    r += 2*a+b;
    cout<<"r = "<<r<<"\tx = "<<x<<endl;
    return x+2;
};
y(3, 5);
cout<<"x = "<<x<<endl;
}

```

2.12 重载函数 `min()`，分别计算 `int`、`double`、`float`、`long` 类型数组中的最小值。

2.13 有学生的结构 `Student` 包括姓名、学号、年龄和班级，用 STL 的函数 `sort()` 对 `Student` 的数组排序，实现按班级降序排序、按年龄升序排序的输出。

2.14 某单位职工的基本工资数据如下：

职工编号	姓名	基本工资	加班工资	奖金	扣除	实发工资
K01	Tom	1200	500	1000	134	
K02	John	2000	120	500	300	
K03	White	1400	200	400	120	

编写程序，实现如下功能：从键盘输入各位职工的工资数据，存入磁盘文件 `Salary.dat`；然后从中读出职工的工资数据，并计算每位职工的实发工资；输出格式与上面相同，但要输出已被计算出来的实发工资。

实发工资的计算方法如下：

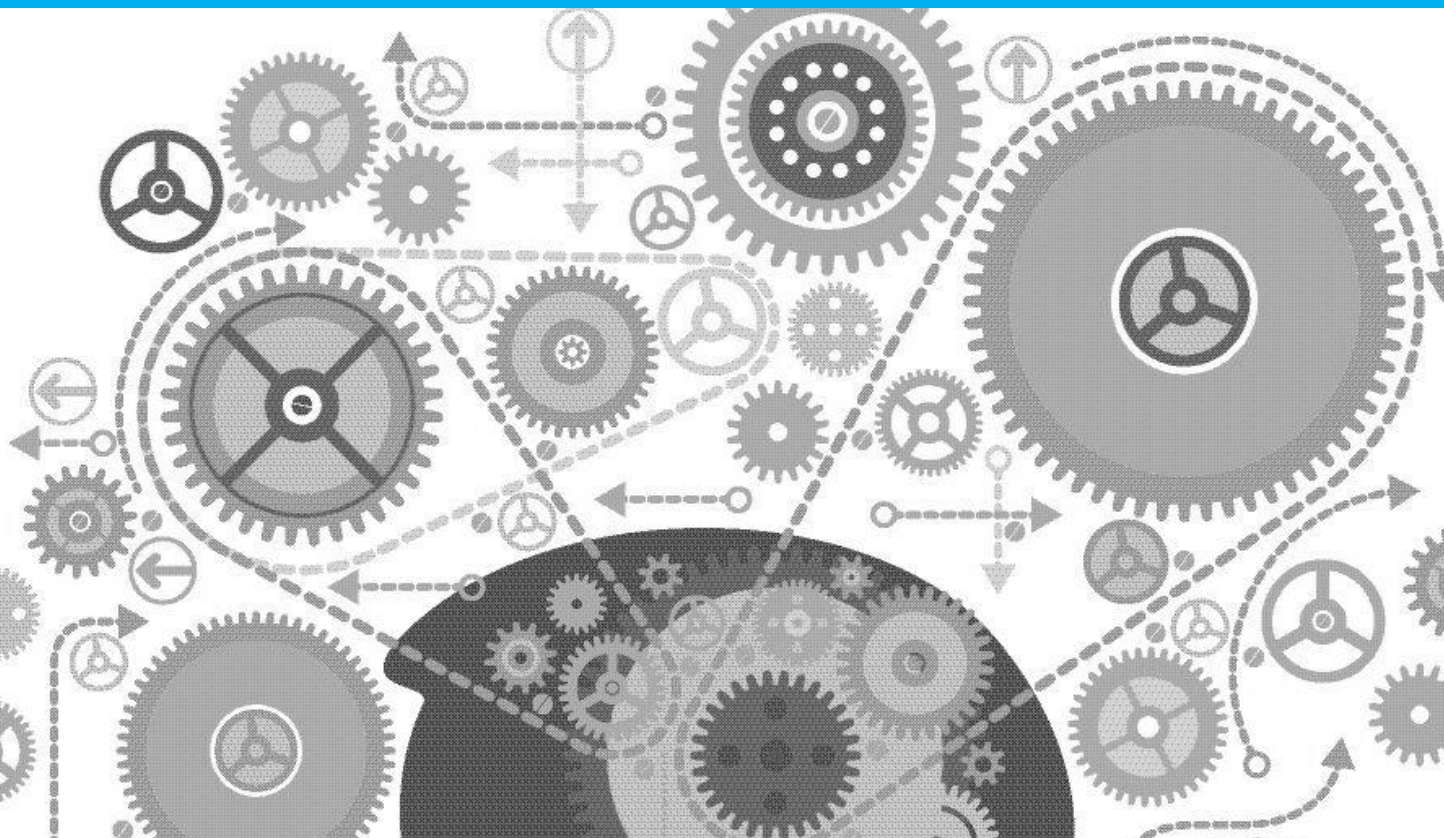
实发工资 = 基本工资 + 加班工资 + 奖金 - 扣除

第 3 章

类和对象

类（class）是面向对象程序设计的核心，是实现数据封装和信息隐藏的工具，是继承和多态的基础。类是一种有别于普通数据类型的自定义数据类型。普通数据类型只能包括数据定义，类却可以同时包括数据和函数的定义，并把它们组合成一个整体。对象其实是由类定义出来的变量。广义地讲，类与对象就是数据类型与变量的关系，凡是用数据类型定义的变量都是对象。

本章介绍应用抽象和封装的方法进行类设计的基础知识，包括类的结构、定义、访问权限、构造函数和析构函数、静态成员和类对象、this 指针、对象复制和对象移动等内容。



3.1 类的抽象和封装

通过数据抽象和封装,面向对象程序设计出表示问题的抽象数据类型(Abstract Data Type, ADT)。抽象数据类型是用于表示应用问题的数据结构,通常由基本数据类型(如 int、char、double 等)组成,并包括一组服务(实现特定功能的函数,常被称为抽象数据类型的接口)。面向对象程序设计的主要任务就是对求解问题域中的各类事物进行数据抽象,然后把它们封装成对应的抽象数据类型——类。

3.1.1 抽象

抽象是人们认识客观事物的一种常用方法,指在描述客观事物时,有意去掉被考察对象的次要部分和具体细节,只抽取与当前问题相关的重要特征进行考察,形成可以代表对应事物的概念。计算机软件开发中采用的抽象方法主要有过程抽象和数据抽象两种。

过程抽象是面向过程程序设计采用的以“功能为中心”的抽象方法,将整个系统的功能划分为若干部分,每部分由若干过程(函数)完成。过程抽象强调过程的功能设计,只需准确地描述每个过程要完成的功能,而忽略实现功能的详细细节(只设计函数应该提供的功能,不涉及具体的编码实现)。例如,若要实现两数的加、减、乘、除运算,采用过程抽象方法会得到类似于如下抽象结果:

```
x add(a, b);           // 功能: 完成 a+b
x sub(a, b);           // 功能: 完成 a-b
x mul(a, b);           // 功能: 完成 a×b
x div(a, b);           // 功能: 完成 a÷b
```

过程抽象的结果给出了函数的名称、接收的参数和能够提供的功能,至于这些函数如何编码实现,则不是过程抽象关心的事情(函数的内部实现可以采用不同方式,但并不会影响函数的使用)。因此,过程抽象提供了信息隐藏和重用性,函数使用者只需知道函数的名称、功能和参数形式,不需了解其实现细节,就可以进行函数调用,应用它的功能。

数据抽象是面向对象程序设计方法,采用以“数据为中心”的抽象方法,忽略事物与当前问题域无关的、不重要的部分和具体细节,抽取同类事物与当前所研究问题相关联的、公有的基本特征和行为,形成关于该事物的抽象数据类型。在抽象数据类型中,常用数据表示事物的基本特征,称为数据成员;用函数表示其行为,称为成员函数。

【例 3-1】 某社区要对小区内的宠物狗实行信息化管理,设计表示宠物狗的抽象数据类型。

(1) 问题分析

现实生活的各种宠物狗差别很大:有的高大,有的矮小;有的嘴长,有的嘴短;有的毛红,有的毛白;有的跑得快,有的跑得慢;有的叫声大,有的叫声小……要完完全全地把各种狗的全部特征和行为描述出来非常困难,哪怕只把狗嘴说清楚也不容易,因为每个狗嘴的形状、大小、色彩各有不同。但是,这里的问题域是小区对宠物狗的管理,不需要把狗的所有特征和行为都描述出来。比如,狗喜欢什么饮食、食量大小、睡眠习惯、狗尾长短等特征和行为与本问题域没有太大关系,可以不予考虑。反之,狗的名字和主人是谁等特征对本问题域而言则是不可忽略的主要特征。

（2）数据抽象

忽略与本问题域无关的特征和行为：宠物狗的叫声大小，狗尾大小和长短，狗耳形状和大小，狗的听力好坏，饮食习惯^[1]……

忽略不重要的、次要的宠物狗特征和行为：狗出生地在哪里，狗父狗母是谁，有无狗兄狗弟，狗毛的长短如何、粗细怎样，见到人后如何摇头摆尾，如此等等。

对于感兴趣的、与本问题研究有关的宠物狗共性特征进行抽取和描述。在对特征进行抽象时，忽略每只宠物狗的具体特征，把所有宠物狗的共性特征描述出来即可。假设对宠物狗的品种、毛色、高低、长短感兴趣，以便通过这些特征识别是哪家的宠物狗，那么抽象方法如下：在抽象狗毛颜色时，忽略狗毛的长短、粗细等，只关注狗毛是有颜色的，用 **color** 表示；在抽象狗的高低时，忽略藏獒比吉娃娃要高大许多，只关注狗是有高度的，用 **high** 表示；忽略每条狗的具体尺寸、狗名字和主人姓名，**len** 表示长短，**name** 表示狗名，**owner** 表示它的主人，**breed** 表示品种。抽象过程如图 3-1 所示。

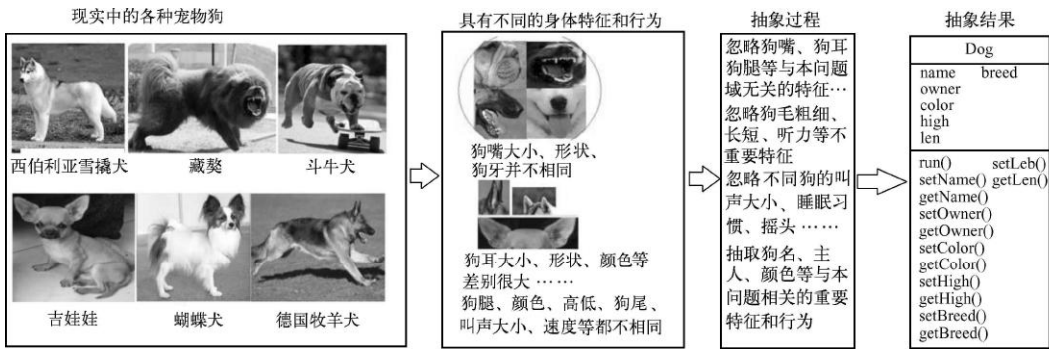


图 3-1 从各种具体的狗类中抽取共同的主要特征进而抽象出宠物狗类的过程

按照相同方法对宠物狗的行为进行抽象，假设对宠物狗的奔跑速度比较关注，要对其进行抽象：忽略吉娃娃的奔跑快慢、跑步状态、头尾摆动状况，忽略牧羊犬、藏獒以及所有宠物狗奔跑的具体姿势和区别，用函数 **run()** 表示宠物狗跑这一行为。

表 3-1 宠物狗的初次抽象

抽象类型	Dog
重要特征	owner, name, color, high, len, breed
重要行为	run()

同样，忽略所有对识别狗时影响较小的、次要的特征和行为，只抽取与本问题域相关的主要特征和行为，形成本问题域的关于宠物狗的初次抽象，如表 3-1 所示。

然而，数据抽象到此并未结束，以数据为中心，并非只有数据，还包括对数据的操作。其原因是在面向对象程序设计中，数据通常被视为对象的“内部机密”，不允许直接访问，只有通过对象提供的授权函数才能操作访问。也就是说，类 **Dog** 的 **name**、**color** 等特征数据会被隐藏起来，在程序中不能够直接读写它们，只有通过类 **Dog** 提供的成员函数才能够修改和访问这些数据。好比向某人借钱，你不能直接去拿他的钱包并从中拿钱，只能由他自己从钱包中拿钱给你，他有多少钱只有他告诉你才知道，称为

[1] 这些被忽略掉的特征并非不重要，有些甚至是宠物狗的重要特征，但数据抽象应首先立足于问题域对客观事物的信息需求，从小区宠物狗信息管理问题域的角度看，这些信息确实没有什么用处，所以被忽略。但是，如果建立的是宠物狗医疗信息管理或宠物狗保健管理之类的信息系统，同样是设计宠物狗的抽象数据类型，这些特征或许是不可忽略的。

信息隐藏，需要通过封装才能够实现。

也就是说，抽象会将提取出的特征数据隐藏起来，不让用户知道和操作，但会提供操作这些数据的功能函数，并向用户公布。用户可以通过这些功能函数操作隐藏的数据，这些对用户公布的功能函数被称为接口。因此，除了设计出描述事物的特征数据，抽象的任务还包括为抽象数据类型设计出清晰而足够的接口，使用户可以通过这些接口访问隐藏在内部的特征数据，但抽象并不关心这些接口的实现细节。

计算机中对数据的操作不外乎读出和写入两类，因此设计接口的一种简单方法是针对抽象出的每个特征数据 x ，设计 `getx()`、`setx()` 两个读写该数据的函数^[2]，形式如下：

```
T x;
T getx() { return x; }
void setx(T y) { x = y; }
```

其中， T 代表 x 的数据类型，函数 `getx()` 用于读取 x 的值，`setx()` 用于设置 x 的值。

随着狗的长大，高度会发生变化，可以设计函数 `setHigh()` 实现对 `high` 的修改，函数 `getHigh()` 获取 `high` 的值。同样，需要设计其他属性的修改和访问函数，为了方便起见，设计函数 `display()` 一次性输出狗的各项数据，最终完成的抽象数据类型 `Dog`，如表 3-2 所示。

表 3-2 宠物狗的最终抽象

类 型	Dog	
重要特征 (数据成员)	<code>string name</code>	<code>name</code> 为宠物狗名，字符串类型
	<code>string owner</code>	<code>owner</code> 为狗的主人名字，字符串类型
	<code>string color</code>	<code>color</code> 为狗的颜色，字符串类型，如“黑色”
	<code>double high</code>	<code>high</code> 为狗的身高，双精度数类型
	<code>double len</code>	<code>len</code> 为狗的长短，双精度数类型
	<code>string breed</code>	<code>breed</code> 为狗的品种，字符串类型，如“贵宾犬”
接口 (成员函数)	<code>void run()</code>	按照狗的品种输出狗跑的大致状态、速度等
	<code>void setName(string) / string getName()</code>	设置/获取狗的名字
	<code>void setOwner(string) / string getOwner()</code>	设置/获取狗的主人名字
	<code>void setHigh(double) / double getHigh()</code>	设置/获取狗的身高
	<code>void setLen(double) / double getLen()</code>	设置/获取狗的长短
	<code>void setBreed(string) / string getBreed()</code>	设置/获取狗的品种
	<code>void setColor(string) / string getColor()</code>	设置/获取狗的颜色
	<code>void display()</code>	输出狗的全部数据

数据抽象是面向对象程序设计中非常重要的概念，面向对象程序设计的核心任务就是设计出求解问题域中各相关事物的抽象数据类型。

3.1.2 封装

抽象是将对象可以被观察到的行为设计成对应抽象数据类型的一组接口访问函数。通过这组函数，用户可以了解到该抽象类型的全部功能和调用方法。但是，抽象并没有实现每个函数，也不关心如何实现它。这就说明，抽象导致了接口与实现的分离，定义了接口但并未实现接口，这样的抽象数据类型还不能用来进行程序设计。要想使用它，必须首先实现各接口的功能函数。这个任务由封装实现。

[2] 此方法称为 `setter` 和 `getter` 方法，当数据较多时，需要为每个数据成员都设计成对的读写函数，显得很烦琐，受到不少争议。

封装与抽象是互补的概念。抽象关注对象的外部视图，封装关注对象的内部实现，用来完成数据抽象设计的目标：用户只能通过接口访问抽象数据类型的功能，只需向接口函数传递正确的参数，就能够使用该接口的功能，不必知道这些功能的实现细节和内部数据的状态。

为了实现数据抽象设计的目标，封装对抽象形成的数据类型进行了包装，将数据和对数据的操作捆绑成一个整体，并且编码实现抽象所设计的接口功能；同时，采用信息隐藏技术，只将接口显露给用户，允许用户通过接口访问该抽象类型的功能，但将接口之外的其余部分（通常包括数据成员、接口的实现细节和抽象类型的结构）都隐藏起来，不让用户知道和访问。

封装后的抽象数据类型才是可以用来进行程序设计的数据类型，由两部分构成：接口和实现。接口显露在外，描述了抽象类型显露给使用者的外部视图；实现则封装了细节，用户对此一无所知，也不需知道，因为这并不影响用户对该数据类型功能的使用。反之，封装后的抽象数据类型更便于使用，因为用户不会被复杂的内部结构和实现细节所干扰，只需向接口传递正确的参数就能够使用到需要的功能。

面向对象程序设计语言通过类（**class**）来实现封装，也可以说，封装后的抽象数据类型被称为类。类具有封装能力，能够将抽象类型的数据和操作函数包装成一个整体，并将数据的内部结构和接口的实现细节隐藏起来，只向外界提供接口。除了能够通过接口访问类的功能，外部对象对类的内部构造和实现细节是一无所知的。类的基本结构如下：

```
class 类名 {  
    public:  
        公共成员;  
    private:  
        私有成员;  
};
```

类的封装机制是：类名是抽象数据类型的名称，整个“{ }；”内部的全体成员是一个整体，成员之间可以相互访问，不受限制。**private** 区域是对外部用户隐藏的区域，用于实现信息隐藏，用户不知道并且无权操作设置在此区域的数据和函数；**public** 区域是对用户公开的区域，用于实现接口功能，此区域的数据和函数可以被用户调用，用户只能通过它们操作 **private** 区域的隐藏数据。

注意：无论是 **private** 还是 **public** 区域都可以放置数据和函数，但出于信息隐藏的目的，常常将数据成员放置到 **private** 区域，而将接口函数放置在 **public** 区域。

【例 3-2】 用类对宠物狗的抽象结果进行封装，完成宠物狗抽象数据类型 **Dog** 的最后设计。

（1）问题分析

例 3-1 已经完成了对宠物狗的抽象，需要隐藏的数据成员包括 **name**、**owner**、**high**、**len**、**breed**，只需用 C++ 的数据类型定义这些数据成员，并把它们放入 **class** 的 **private** 区域。为了便于字符串的输入/输出，用 **string** 类型定义 **name**、**owner**、**breed**；可以用 **int**（以厘米为单位）或 **double** 类型（以米为单位）定义 **high** 和 **len**，这里将其定义为 **double** 类型。

接口函数是围绕数据成员设置的，因此其参数类型与其对应数据成员的类型相同。例如，接口函数 **setHigh()** 用于设置 **high**，而 **high** 是 **double** 类型，只需传递 **double** 类型的参数，并用它设置 **high**，就完成了对狗高的修改。同时，接口函数 **getHigh()** 用于返回 **high**，因此需要返回 **double** 类型的数据，然后返回 **high**，就能够获取狗的高度。按同样的思路和方法，完成所有接口函数的设计，并把它们放置在 **public** 区域。

(2) 抽象数据类型

宠物狗封装好的抽象数据类型 **Dog** 如下。

```
class Dog {
public:
    void run() { cout<<"I am "<<name<<" , my speed is "<<rand()<<endl; }
    void setName(string Dname) { name = Dname; };
    void setOwner(string D0name) { owner = D0name; }
    void setHigh(double Dhigh) { high = Dhigh; }
    void setLen(double Dlen) { len = Dlen; }
    void setBreed(string type) { breed = type; }
    void setColor(string Dcolor) { color = Dcolor; }
    string getName() { return name; }
    string getOwner() { return owner; }
    double getHigh() { return high; }
    double getLen() { return len; }
    string getBreed() { return breed; }
    string getColor() { return color; }
    void display() {
        cout<<"dog's name:"<<name<<"\tbread:"<<breed
            <<"\tcolor:"<<color<<"\thight:"<<high
            <<"\tlength:"<<len<<"\tdog Owner:"<<owner<<endl;
    }
private:
    string name;
    string owner;
    string color;
    double high;
    double len;
    string breed;
};
```

经过封装后的类 **Dog** 才是可以用于程序设计的抽象数据类型，在程序中可以用来定义变量，如同用 **int** 定义变量一样。例如：

```
int x;
Dog dog1;
```

从某种意义上，这两条语句并没有什么本质的区别。它们分别申请到了一块内存区域，一块名称为 **x**，而另一块名称为 **dog1**。**x** 是用系统内置数据类型定义的，而 **dog1** 是用自定义数据类型定义的。区别在于，**x** 只有 4 字节，可以直接读写 **x** 的值，而 **dog1** 是更大的一片内存区域，其中的数据成员只能通过 **public** 区域的接口函数才能被读写。

用类封装后的抽象数据类型称为类，面向对象程序设计的主要任务就是对问题空间的各类客观事物进行抽象，构造出代表各类事物的类。

3.2 结构

面向对象设计语言通常采用类实现对抽象数据类型的封装，但在 **C++** 中，结构（**struct**）具有与类完全相同的功能，也可以用来设计类。

3.2.1 C++对结构的扩展

最初的 C++被称为“带类的 C”，扩展了 C 语言的功能。C++的结构不仅可以包含数据，还可以包含函数，同时引入了 `private`、`public` 和 `protected` 三个访问权限限定符，其目的是实现封装和信息隐藏。C++中，结构的定义形式如下：

```
struct 类名 {  
    [public:]  
    成员;  
    private:  
    成员;  
    protected:  
    成员;  
};
```

其中的成员包括数据成员和成员函数，`public`、`protected` 和 `private` 用于设置成员访问权限，这些访问控制符可以按任意次序出现任意多次。

`private` 用于实现信息隐藏。被设置为 `private` 权限的成员（包括数据成员和成员函数）称为类的私有成员，只能被结构内部的成员访问。结构之外的对象只能通过 `public` 区域中的公有接口才能访问这个区域中的成员。

`public` 用于实现接口功能。被设置为 `public` 权限的成员称为类的公共成员，可被任何函数访问（包括结构内和结构外的函数）。结构的成员的默认访问权限就是 `public`，即一个没有被任何访问权限限制的成员，实际上具有 `public` 权限，可以被任何对象访问。

`protected` 与继承有关，以后介绍。

C++结构的成员默认都具有 `public` 权限，可被直接访问，通过访问控制权限的设置，可以改变成员的访问权限。

【例 3-3】 用结构对圆进行抽象，构造出计算圆周长和面积的抽象数据类型。

(1) 问题分析

圆是一种常见的几何图形，具有圆心和半径，但本问题只需计算圆的周长和面积，与圆心没有太大关系，可以被忽略，只需考虑圆的半径（`radius`）、周长（`perimeter`）、面积（`area`）。

(2) 数据抽象

忽略圆心后，半径就是唯一的数据成员了，用 `r` 表示，按照抽象的原则，设置为私有数据成员，以实现信息隐藏。接口函数 `setR()`、`getR()` 用于设置、读取 `r` 的信息，`perimeter()` 计算圆的周长，`area()` 计算圆的面积。

(3) UML 类图

UML（**Unified Modeling Language**，**统一建模语言**）定义了用例图、类图、对象图、状态图、活动图、序列图、协作图、构件图、部署图等 9 种标准图形，用于从不同侧面对问题进行描述，能够表达软件设计中的动态和静态信息，便于系统的分析和构造，是面向对象软件的标准化建模语言。本书不打算详细介绍 UML 的全部内容，但会引用其中的部分图形来描述与之对应的内容。这里首先介绍 UML 类图，以使用它表示数据抽象设计的类。

类图（**Class Diagram**）是 UML 中最重要也是最常见的一种图形，用于描述类的成员组成和关系。类图用一个矩形表示，其中包括类名、数据成员和成员函数三部分，如图 3-2 所示。

在类图中，用“+”表示 public 访问权限，用“-”表示 private 访问权限，用“#”表示 protected（保护）访问权限，表示方法是在类成员的前面写上与其访问权限相对应的符号。

将 Circle 数据抽象的结果用类图表示出来，如图 3-3 所示，其含义是：类的名称是 Circle，只有一个数据成员 r，其类型为 double，访问权限为 private，即需要进行信息隐藏。该类有 4 个公有接口函数，其中函数 getR()、perimeter()、area()不需要参数，返回类型是双精度，函数 setR()接收一个 double 类型的参数，无返回结果。

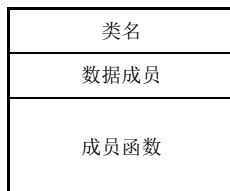


图 3-2 UML 类图的表示方法

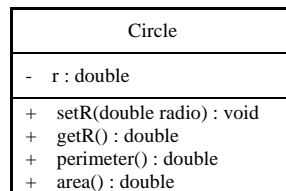


图 3-3 Circle 的类型

用结构对图 3-3 的类 Circle 进行封装，完成抽象数据类型 Circle 的设计，并用它定义半径为 4 的圆。完整的代码如下所示。

```
// Eg3-3.cpp
#include<iostream>
#include<string>
using namespace std;

struct Circle {
    public:                                     // L1
        void setR(double radio) { r = radio; }
        double getR() { return r; }
        double perimeter() { return 2*3.14*r; }
        double area() { return 3.14*r*r; }
    private:                                   // L2
        double r;
};

void main() {
    Circle c;
    // c.r = 4;                                // L3 错误，r 为 private，不可访问
    c.setR(4);

    cout<<"r = "<<c.getR()<<"\tperimeter = "<<c.perimeter()<<"\tarea = "<<c.area()<<endl;
}
```

访问权限限定符的有效范围是从其出现开始直到下一个权限设置，所以本例中的成员函数 setR()、getR()、perimeter()、area()都被设置为 public 权限，它们可被类 Circle 之外的对象访问，因此在函数 main()中对它们进行函数调用是正确的。但是 r 不一样，它具有 private 权限，只能被类 Circle 内部的 setR()等 4 个函数访问，类 Circle 之外的任何对象访问 r 都是禁止的，这就是结构对 r 进行信息隐藏的效果，也是语句 L3 错误的原因。

在结构封装的类中，如果没有设置成员的访问权限，就默认为 public 权限。比如，删掉语句 L1、L2，那么所有成员都具有 public 权限，语句 L3 也是合法的了。

3.2.2 类

类是具有信息隐藏能力，能够完成接口与实现的分离，把数据抽象的结果封装成可以用于程序设计的抽象数据类型，是面向对象程序设计语言中通用的数据封装工具。Java、Python、Rube、Scale 等都用关键字 `class` 来定义类。在 C++ 中，类具有与结构完全相同的功能，用法一致，形式如下：

```
class class_name {
    [private:]                // 可以省略
    成员;
public:
    成员;
protected:
    成员;
};                             // 分号必不可少
```

其中，`class_name` 是类名，常用首字符大写的标识符表示；`private`、`public`、`protected` 用于指定成员的访问权限，与其在结构中的含义和用法都相同；成员可以是数据成员或成员函数；一对“{}”界定了类的范围，“}”后面的“;”必不可少，表示类声明的结束。

【例 3-4】 设计复数类 `Complex`，提供复数的修改、输入和显示功能。

(1) 问题分析

复数是数学和工程中常用的一种数据类型，由实部和虚部组成，能够进行加、减、乘、除等数学运算，但本问题并未要求实现这些功能，因此忽略这些运算。

(2) 数据抽象

出于信息隐藏目的，将复数的实部、虚部设置为 `double` 类型的私有成员，并且设置输入、修改、显示的接口函数 `inputData()`、`setReal()`、`setImage()`、`display()`，如图 3-4 所示。

用类封装后的类 `Complex` 及其使用的完整代码如下。

```
// Eg3-4.cpp
#include<iostream>
using namespace std;

class Complex {
public:
    void display() {
        if(image < 0)    cout<<real<<image<<"i"<<endl;
        else    cout<<real<<"+ "<<image<<"i"<<endl;
    }
    void inputData() {
        cout<<"input real: ";
        cin>>real;
        cout<<endl<<"input image: ";
        cin>>image;
    }
    void setImage(double i) { image = i; }
    void setReal(double r) { real = r; }
private:
    double image;
    double real;
};
```

Complex
- image : double - real : double
+ display() : void + inputData() : void + setImage(double) : void + setReal(double) : void

图 3-4 Complex 类图

```

};
void main() {
    Complex c1;
    // c1.image = 9.2;                // L1, 错误
    c1.inputData();
    c1.display();
    c1.setImage(9.2);                // L2
    c1.setReal(5.3);
    c1.display();
}

```

说明：

① 类或结构后的“{…};”包围的区域是一种独立的作用域，称为类域。类域之中的数据和函数通称成员，其中数据称为数据成员，函数则常被称为成员函数。

② 类声明中的访问限定符 `private`、`public`、`protected` 没有先后次序之分，哪个在前面，哪个在后面没有区别。通常，`public` 成员放在前面，`private` 成员放在后面，以方便用户了解类的可访问接口。

③ 在同一个类中，访问限定符 `private`、`public`、`protected` 的出现次数没有限制。例如，可以将一个 `public` 区域中的成员拆分为多个 `public` 区域，也可以将多个 `public` 区域中的成员合并在一个 `public` 区域中。

④ 数据成员和成员函数都可以设置为 `public`、`private` 或 `protected` 属性。出于信息隐藏的目的，常将数据成员和只能让类内部访问的成员函数设置为 `private` 权限，将需要让类的外部函数（非本类定义的函数）访问的成员函数设置为 `public` 权限。

⑤ 同一类域的成员不受 `public`、`private` 和 `protected` 访问权限的限制，相互之间可以直接访问，并且不受声明先后次序的影响，前面的成员函数可以直接访问在它后面定义的成员。例如，本例的函数 `display()` 和 `inputData()` 并未通过参数传递形式，就直接访问了私有成员 `image` 和 `real`。事实上，在一个成员函数内部还可以直接调用另一个成员函数。

⑥ 结构也是一种类，与类具有相同功能，用法也相同。把例 3-3 的 `struct` 换成 `class`，或把本例的 `class` 换成 `struct`，设计的类 `Complex` 或 `Circle` 是完全相同的。唯一的区别是，在没有指定成员的访问权限时，结构的成员具有 `public` 权限，类的成员具有 `private` 权限。例如：

<pre> struct Complex { double r; double i; public: }; </pre>	<pre> class Complex { double r; double i; public: }; </pre>
--	---

`struct Complex` 中的 `r` 和 `i` 是 `public` 成员，而 `class Complex` 中的 `r` 和 `i` 是 `private` 成员。

虽然结构和类具有同样的功能，但在实际工作中，常用类设计具有成员函数的类，结构则保留 C 语言中的用法，常用来设计只包含 `public` 数据成员的结构。

3.3 数据成员

数据成员可以是任何数据类型，如整型、浮点型、字符型、数组、指针、引用，以及 STL 库中的 `vector`、`list`、`map` 等数据类型，也可以是另一个类的对象或指向对象的指针，还可以

是指向自身类的指针或引用，但不能是自身类的对象；可以是运行期常量 `const`，但不能是编译器常量 `constexpr`，可以用 `decltype` 推断定义，但不能用 `auto` 推断定义。此外，数据成员不能被指定为寄存器（`register`）和外部（`extern`）存储类型。例如：

```
class A{...};
class B{
private:
    int r;
    A obja1, *obja2;           // 正确
    B *objb, &objr;           // 正确
    B b1;                     // 错误
    auto b=a+1;               // 错误
    decltype(r) a;            // 正确
    extern int c;              // 错误
    const int x;               // 正确
    constexpr int y;          // 错误
public:
    .....
};
```

C++ 11 之前的规范中不允许在声明（或定义）类时为数据成员赋初值，但自 C++ 11 标准开始规定，可以为数据成员提供一个类内初始值，用于创建类对象时初始化数据成员。例如：

```
class A {
private:
    int a = 0;                 // C++ 11 之前错误，C++ 11 之后正确
    int y = {0};              // C++ 11 之前错误，C++ 11 之后正确
    int b[3] = {1,2,3};       // C++ 11 之前错误，C++ 11 之后正确
    const int c = a;           // C++ 11 之前错误，C++ 11 之后正确
public:
    // .....
};
```

类 A 在类似于 VC 2013 后的编译环境中是正确的，在 VC 6.0（不支持 C++ 11 标准）环境中则会产生编译错误。

实际上，类的声明（或定义）只是增加了一种自定义数据类型，此时类内部的数据成员并没有获取到相应的内存空间。只有在用类定义对象（变量）时，数据成员才会被分配空间，在这个时间点上才会用相应的初始值初始化数据成员。

3.4 成员函数

3.4.1 成员函数定义方式和内联函数

在面向对象程序设计中，类的成员函数也称为方法或服务，有两种定义方式。

1. 类内定义成员函数

在声明类时，直接在类的内部就给出成员函数的定义，以这种方式定义的成员函数若符合内联函数的条件，就会被系统处理为内联（`inline`）函数。例如，一个日期类的定义如下：


```

class Date {
    int day, month, year;
public:
    void init(int d, int m, int y) {
        day = d;
        month = m;
        year=y;
    }
    int getDay() {
        return day;
    }
    .....
};

```

在类 `Date` 中，直接在类的内部进行了成员函数 `init()` 和 `getDay()` 的定义，而且这两个函数比较简单，符合内联函数的要求，会被编译器自动设置为内联函数。

2. 类外定义成员函数

在声明类时，若只声明了成员函数的原型，则需在类的外部定义该成员函数，方法如下：

```
r_type class_name::f_Name(T1 p1, T2 p2, ...);
```

其中，`r_type` 是成员函数的返回类型，`class_name` 是类名，`::` 是域限定符，用于说明函数 `f_Name()` 是 `class_name` 的成员函数，`f_Name` 是成员函数名，`T1`、`T2` 是参数类型；`p1`、`p2` 是形式参数，在类声明的函数原型中并无任何意义，可以省略。但在定义成员函数时，形参是不可省略的。

例如，`Date` 类的成员函数 `init()` 和 `getDay()` 也可以用如下方法定义。

```

class Date {
    int day, month, year;
public:
    void init(int, int, int);           // 省略了形式参数
    int getDay();
    inline int getMonth()
};

int Date::getDay() { return day; }
int Date::getMonth() { return month; }
inline void Date::init(int d, int m, int y) {
    day = d;
    month = m;
    year = y;
}

```

`init()` 和 `getMonth()` 是内联函数，而 `getDay()` 不是。在 C++ 中，若在类外成员函数的声明或定义前面加上关键字 `inline`，则该成员函数也能被定义为内联函数。

说明：

- ① 若采用类外方式定义成员函数，则类声明时成员函数原型中的形参名可以省略，只声明各形参的类型。
- ② 在类外定义成员函数时，成员函数的返回类型、函数名称、参数表必须与成员函数原型的声明完全相同，而且必须指出每个形参的名称。

③ 在类外定义成员函数时，必须在成员函数名前面加上类名，并且在类名与成员函数之间用 “::” 间隔。

3.4.2 常量成员函数

在 C++ 中，为了禁止成员函数修改数据成员的值，可以将它设置为常量成员函数，方法是在函数原型的后面加上 `const`。形式如下：

```
class x {
    .....
    T f(T1, T2, ...) const;
    .....
};
```

其中，`T` 是函数返回类型，`f` 是函数名，`T1`、`T2`... 是各参数的类型。将成员函数设置为 `const` 类型后，表明该成员函数不会修改任何数据成员的值。例如：

```
class Employee {
    char *name;
    double salary;
public:
    void init(const char *Name, const double y);
    double getSalary() const;           // 常量函数，不能修改 name 和 salary
    char *getName() const;             // 常量函数，不能修改 name 和 salary
    void addSalary(double x) const;     // 常量函数，不能修改 name 和 salary
};
// 本函数的参数是常量，但不是常量成员函数
void Employee::init(const char *Name, const double y) {
    name = new char[strlen(Name) + 1];
    strcpy(name, Name);
    salary = y;
}
double Employee::getSalary() const {   // 正确
    return salary;
}
void Employee::addSalary(double x) const {
    salary += x;                       // 错误，常量成员函数不能修改数据成员
}
char *Employee::getName() {           // 错误，缺少 const，与类中声明的原型不符
    return name;
}
```

说明：

① 只有类的成员函数才能定义为常量函数，普通函数不能定义为常量函数。如下函数定义是错误的：

```
int f(int x) const {                  // 错误，普通函数不能被指定为 const
    int b = x*x;
    return b;
}
```

② 常量参数与常量成员函数是有区别的，常量参数限制函数对参数的修改，但与数据成员是否被修改无关。

3.4.3 成员函数重载和默认参数值

类的成员函数也可以像普通函数一样被重载和设置参数的默认值。例如：

```
class Date {
    int day, month, year;
public:
    void init(int d, int m = 8, int y = 2016) {           // m、y 设置了默认值
        day = d;
        month = m;
        year = y;
    }
    void init(int d, int m) {
        day = d;
        month = m;
        year = 2016;
    }
    void init(int d) {
        day = d;
        month = 8;
        year = d;
    }
};
```

成员函数重载和默认参数值设置的规则与普通函数相同，即重载成员函数必须具有不同的参数表，如果为某个参数指定了默认值，就要求它右边的全部参数都必须为默认值。

3.5 对象

类描述了同类事物共有的属性和行为，类的对象是具有该类所定义的属性和行为的实体。类是抽象的、概念性的范畴，对象是内存中实际存在的个体。类与对象的关系实质上就是数据类型与变量的关系。类是一种自定义数据类型，用它定义的变量就是对象，遵守变量的作用域和生命期规则。广义上，在面向对象程序设计中，用任何数据类型定义的变量都可以称为对象。

1. 对象的定义

定义对象与定义一个普通变量的方法没有区别，形式如下：

```
类名 对象1, 对象2;
```

【例 3-5】 设计时钟类，要求能够完成时间的设置和显示，并创建时钟类的对象，演示对象的概念和用法。

(1) 问题分析

任何时钟都是一个独立存在的有形实体，在钟面设置有时针、分针、秒针，人们通过这

些指针查看时间，如果时间不准确，还可以通过这些指针调整时间。这些是时钟提供给人们使用的接口，可以设置对应的 `public` 函数来实现该功能。

时钟的内部结构和内部运行机制则被封装在内部，人们不知道这些内部结构，也不知道时钟的内部运行机制。（指针怎样移动？是电流驱动还是机械驱动呢？）事实上，这些都是时钟的事情，人们没必要知道。可以设置 `private` 成员来实现对它们的隐藏。

(2) 数据抽象

用类 `Clock` 来抽象和封装时钟类，用 `hour`、`minute`、`second` 表示时、分、秒，用私有成员函数 `run()` 仿真时钟的内部运行机制，相关操作（如设置时、分、秒）分别用公共成员函数 `setTime()`、`setMinute()` 和 `setSecond()` 模仿，通过它们调整时间；用 `dispTime()` 模仿时间的显示。类 `Clock` 的抽象过程如图 3-5 所示。

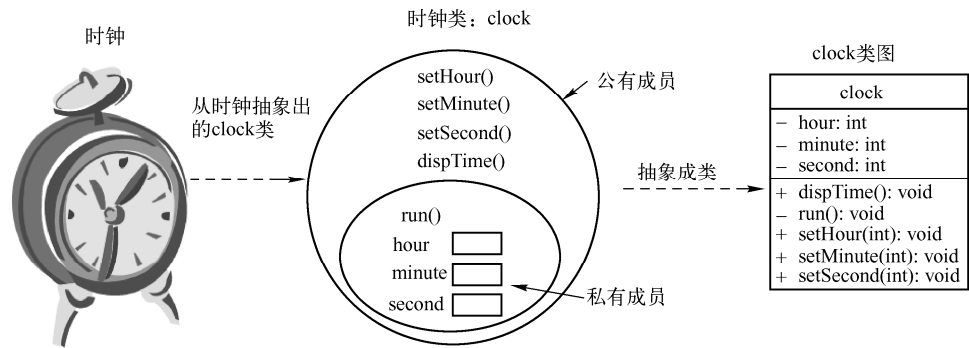


图 3-5 时钟类 `Clock` 的抽象过程

封装好的类 `Clock` 就是可以使用的抽象数据类型，可以像内置的 `int`、`double` 等类型一样定义变量，即类 `Clock` 的对象。下面语句定义了 `myClock`、`yourClock` 两个对象：

```
Clock myClock, yourClock;
```

每个对象都有 8 个成员，其中包括 3 个数据成员 `hour`、`minute`、`second` 和 5 个成员函数 `run()`、`dispTime()`、`setHour()`、`setMinute()` 和 `setSecond()`。图 3-6 是用类 `Clock` 定义的两个对象 `myClock` 和 `yourClock` 的对象内存示意。

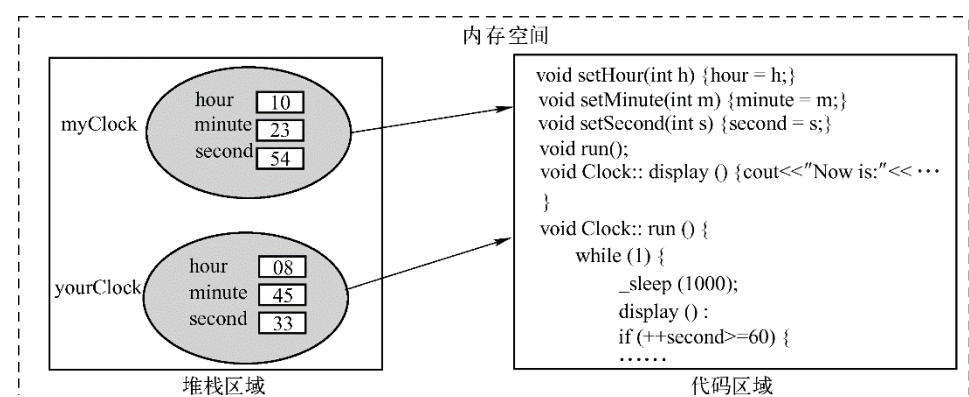


图 3-6 `myClock` 和 `yourClock` 对象示意

C++ 会在堆栈中为每个对象独立地分配存储空间，有多少个对象就要分配多少次存储空间。但是，只为每个对象的数据成员分配独立的存储空间，而同一类的成员函数在内存中只有一份代码，供该类的所有对象公用。这样做的原因是，同一个类的所有对象的成员函数都

相同，通过共享可以节约内存资源；但所有对象的数据成员是不相同的（静态数据成员例外），必须由各对象单独存储。

2. 对象应用

对象应用是指调用对象的接口函数获取对象的功能，方法是用成员访问限定符“.”作为对象名和对象成员之间的间隔符，形式如下：

```
对象名.数据成员名  
对象名.成员函数名(实参表)
```

例如，访问 `myClock` 的成员：

```
myClock.setHour(12);  
myClock.dispTime();
```

说明：

- ① 在类外只能访问对象的公共成员，不能访问对象的私有成员和受保护成员。
- ② 如果定义了对象指针，在通过指针访问对象的公共成员时，要用“->”作为指针对象和对象成员之间的间隔符。例如：

```
Clock *pClock;  
pClock = new Clock;  
pClock->setHour(10);  
pClock->dispTime();
```

3. 对象赋值

同一个类的不同对象之间、对象指针之间可以相互赋值。方法如下：

```
对象名1 = 对象名2;
```

例如，对于前面的 `Clock` 类，如下用法是正确的：

```
Clock *pa, *pb, aClock, bClock;  
.....  
bClock = aClock;  
pa = new Clock;  
.....  
pb = pa;
```

说明：

- ① 进行赋值的两个对象必须类型相同。
- ② 对象赋值就是进行数据成员的值的复制，赋值之后，两个对象互不相干。在上面的语句中，经过赋值后，`bClock` 的数据成员与 `aClock` 的数据成员的值是相同的，但赋值完成后，它们就没有关系了。
- ③ 若对象有指针数据成员，赋值操作可能产生“指针悬挂”问题。这个问题留待介绍析构函数时再进行分析。

类 `Clock` 及其对象引用的完整代码如下。

```
// Eg3-5.cpp  
#include <iostream>  
#include <string>  
using namespace std;
```

```

class Clock {
public:
    void setHour(int h) { hour = h; }
    void setMinute(int m) { minute = m; }
    void setSecond(int s) { second = s; }
    void dispTime() { cout<<"Now is: "<<hour<<":"<<minute<<":"<<second<<endl; }
private:
    int hour, minute, second;
};

void main() {
    Clock *pa, *pb, aClock, bClock;
    aClock.setMinute(12);
    aClock.setHour(16);
    aClock.setSecond(27);
    bClock = aClock;
    pa=new Clock;
    pa->setHour(10);
    pa->setMinute(23);
    pa->setSecond(34);
    pb=pa;
    pa->dispTime();
    pb->dispTime();
    aClock.dispTime();
    bClock.dispTime();
}

```

程序运行结果如下：

```

Now is: 10:23:34
Now is: 10:23:34
Now is: 16:12:27
Now is: 16:12:27

```

3.6 构造函数设计

在进行类的设计时，只把客观事物与问题域相关的主要特征和行为抽象并封装成类是不够的，这样的类虽然描述清楚了问题域中的事物及其关系，能够反映和解决问题本身，但是存在缺陷，在用它们定义对象并编程时，可能还会出现某些问题。在用类定义对对象时必须调用构造函数，当类对象失去作用域和生命期时必须调用析构函数，在用已经定义好的旧对象初始化新定义对象（包括向函数传递对象参数）时必须调用类的复制构造函数，在同类对象之间进行相互赋值时必须调用类的赋值运算符函数，因此在设计类时必须考虑这些成员函数的设计。

本节将介绍对象的构造函数，解决对象的构造问题；3.7 节将介绍析构函数，解决对象失去生命期时的资源回收问题；3.8 节将介绍复制构造函数、赋值运算符函数以及它们的移动函数版本，解决对象的复制问题。

3.6.1 编译器默认添加的成员函数

从对象程序设计的角度出发，在设计类时应当考虑对象定义时数据成员的初始化，对象之间的复制、赋值、移动和销毁等问题，这些是通过构造函数、赋值运算符函数和析构函数来实现的。也就是说，在设计类时，除了抽象和封装对象本身的数据成员和成员函数，还应当根据需要，考虑上述特殊成员函数的设计。如果程序员没有提供它们的设计，C++编译器会在需要的时候自动在类中添加以下 6 个成员函数：① 默认构造函数；② 复制构造函数；③ 赋值运算符函数；④ 移动构造函数；⑤ 移动赋值运算符函数；⑥ 析构函数。假设类 `MyClass` 有 n 个各种类型的数据成员，编译器为它默认生成的成员函数形式如下：

```
class MyClass {
    type1 x1;
    type2 x2;
    .....
    typen xn;
public:
    MyClass():x1(), x2(), ..., xn() { }           // L1, 默认构造函数
    MyClass(const MyClass& other) :x1(other.x1),   // L2, 复制构造函数
                                   x2(other.x2),
                                   .....
                                   xn(other.xn) { }

    MyClass& operator = (const MyClass& other) {    // L3, 赋值运算符函数
        x1 = other.x1;
        x2 = other.x2;
        .....
        xn = other.xn;
        return *this;
    }

    MyClass(const MyClass&& other) :x1(std::move(other.x1)), // L4, 移动构造函数
                                   x2(std::move(other.x2)),
                                   .....
                                   xn(std::move(other.xn)) { }

    MyClass& operator = (const MyClass&& other) {    // L5, 移动赋值运算符函数
        x1 = std::move(other.x1);
        x2 = std::move(other.x2);
        .....
        xn = std::move(other.xn);
        return *this;
    }

    ~MyClass() {                                   // L6, 析构函数
        xn.~typen();
        .....
        x2.~type2();
        x1.~type1();
    }
};
```

其中，移动构造函数和移动赋值函数是 C++ 11 标准后才有的。从上述形式代码可以看出，生成的默认复制构造函数和赋值运算符函数在进行两个对象之间的复制时，方法是将对对应数

据成员的值从一个对象复制或移动给另一个对象。在大多数情况下，C++自动为类生成的这些成员函数能够满足应用需求，但当有指针类型的数据成员时，复制的是指针本身（浅拷贝），会导致两个对象的指针指向同一内存地址，在对象析构时产生“悬挂指针”，引起程序运行的错误。

要解决这样的问题，就需要显式设计对应的构造函数、赋值运算符函数和析构函数，有“5 法则”和“0 法则”可以参考。所谓“5 法则”，是指复制构造函数、移动构造函数、赋值运算符函数、移动赋值运算符函数和析构函数 5 个操作要么全部设计，要么一个也不设计。其原因是，使用传统指针管理对象的内存资源时，这 5 个操作通常都会同时用到，设计不全就会产生“悬挂指针”。所谓“0 法则”，是指用 C++ 11 中的智能指针 `unique_ptr` 或 `shared_ptr` 来代替传统指针，把资源管理工作交由智能指针管理，就不用实现上述 5 个操作函数中的任何一个，由编译器生成的函数就能够完成需要的工作。

3.6.2 构造函数和类内初始值

构造函数与类内初始值具有相似的功能，都用于为对象的数据成员提供初始值。类内初始值是指在类声明时为数据成员指定的初值。**构造函数**（**constructor function**）是与类同名的特殊成员函数，其主要任务是初始化对象的数据成员，并为成员函数创建“工作环境”（如分配内存、打开资源文件、连接网络或数据库等）。只要有类的对象被创建，就一定会执行构造函数。其定义形式如下：

```
class X {
    .....
    X(...);                      // 构造函数
    T m = a;                     // 类内初始值, C++ 11
    .....
}
```

其中，X 是类名，X(...)是构造函数，可以有参数表。构造函数的声明和定义方法与类的其他成员函数相同，可以在类的内部定义构造函数，也可以先在类中声明构造函数，然后在类外进行定义。在类外定义构造函数的形式如下：

```
X::X(...) {
    .....
}
```

构造函数的特点如下：① 构造函数与类同名，并且没有返回类型；② 构造函数可以被重载；③ 构造函数由系统自动调用，不允许在程序中显式调用；④ 构造函数不能被声明为 `const` 函数。

在用类定义对象时，类内初始值和构造函数将按以下次序执行：<1> 编译器建立对象，为数据成员分配内存空间；<2> 若指定了数据成员的类内初始值，则用类内初始值初始化数据成员；<3> 根据定义对象时提供的参数，匹配正确的构造函数，执行构造函数。

【例 3-6】 某桌子类 `Desk` 具有长、宽、高、重 4 个数据成员，为它设计构造函数，并为宽和高设置类内初始值，通过类内初始值和构造函数的参数对数据成员进行初始化。

```
// Eg3-6.cpp
#include <iostream>
```

```

using namespace std;
class Desk {
public:
    Desk(int, int); // 构造函数声明
    void outData() {
        cout<<"Weight = "<<weight<<"\tHeight = "<<height<<endl;
        cout<<"Length = "<<length<<"\tWidth = "<<width<<endl;
    }
private:
    int width, length, weight = 2, height = 3;
};
Desk::Desk(int w, int h) { // 构造函数定义
    width = w;
    length = h;
    cout<<"call constructor!"<<endl;
}

void main() {
    Desk d(3, 5);
    d.outData();
}

```

程序运行结果如下：

```

call constructor! // 构造函数输出
Weight = 2   Height = 3
Length = 5   Width = 3

```

此结果表明，虽然程序中没有“`d.Desk(...)`”之类的语句，但构造函数确实被调用了，而且类内初始值发挥了作用。构造函数调用的时机是函数 `main()` 中用类 `Desk` 定义对象 `d` 时。对于语句“`Desk d(3, 5);`”，编译器可能将其扩展成如下语句组：

```

Desk d;
执行类内初始化;
d.Desk::Desk(3, 5);

```

编译器首先为对象 `d` 分配内存空间，完成后执行类内初始化“`weight = 2, height = 3`”，然后立即自动调用构造函数初始化对象 `d` 的数据成员 `width` 和 `length`。

在定义构造函数时，必须注意以下问题。

- ① 构造函数不能有任何返回类型，即使 `void` 也不行。
- ② 构造函数由系统自动调用，不能在程序中显式调用。
- ③ 定义对象数组或用 `new` 创建动态对象时，也要调用构造函数，但定义数组对象时，会自动调用不需要参数的构造函数（包括无参数构造函数和所有参数有默认值的构造函数）。
- ④ 构造函数通常应定义为公共成员，因为在程序中定义对象时，要调用构造函数，尽管是由编译系统进行的隐式调用，但也是在类外进行的成员函数访问。

请参考上述说明，理解下面代码的注释中所指出的错误原因。

```

class Desk {
    Desk(){ weight = height = width = length = 0; } // 无参构造函数为 private
public:
    void Desk::Desk(int ww, int l, int w, int h) { // 错误，不能有返回类型

```

```

        weight = ww;
        height = l;
        width = w;
        length = h;
    }
private:
    int weight, length, width, height;
};

void main() {
    Desk d(2,3,3,5);           // 构造函数在定义对象时被系统自动调用
    d.Desk(1,2,3,4);           // 错误，构造函数不能被显式调用
    Desk a[10];                 // 错误，调用 Desk::Desk(), 但它是 private
    Desk *pd;
    Desk d;                     // 错误，调用 Desk::Desk(), 但它是 private
    pd = new Desk(1,1,1,1);     // 调用构造函数 Desk::Desk(int,int,int,int)
}

```

3.6.3 默认构造函数

创建类对象时必须调用构造函数，如果定义对象时提供了初始化值，就会调用参数与初始化值匹配的构造函数；如果没有提供初始化值，编译器就会自动调用默认构造函数，包括不带参数的构造函数，或者为所有的参数都提供了默认值的构造函数。

1. 无参数构造函数

C++规定，每个类必须有构造函数，如果一个类没有定义任何构造函数，在需要时（定义对象时）编译器会为它生成一个默认构造函数。类似于如下形式：

```

class X {
    X():x1(), ..., xn() {}
    .....
}

```

默认构造函数是一个无参数构造函数，负责对象的初始化。如果创建的是全局对象或静态对象，就将对象的位模式全部设置为 0（可以理解为将所有数据成员初始化为 0）；如果创建的是局部对象，就不会对对象的数据成员进行初始化。

在某些时候，程序员必须显式地定义无参数构造函数，以解决对象的初始化问题。比较典型的有如下情况。

① 只有在没有定义任何构造函数时，编译器才会为类生成默认构造函数；一旦为类定义了任何形式的构造函数，编译器就不再生成默认构造函数。在这种情况下，若需要创建无参数对象，则必须显式定义无参数构造函数。在 C++ 11 中，可以用如下方式要求编译器创建默认构造函数。

```

class X {
    X() = default;           // 要求编译器生成默认构造函数，C++ 11
    X(...) {}                 // 定义了需要参数的构造函数
    .....
}

```

前面的例 3-2 至例 3-5 都没有定义任何构造函数，C++编译器会自动为这些例中的类创建

默认构造函数，但是不会为例 3-6 创建默认构造函数。

② 在某些情况下，编译器生成的默认构造函数会执行错误操作。比如，当类具有数组或指针成员时，用默认构造函数执行对象初始化，很有可能产生“指针悬挂”问题。

③ 在某些情况下，编译器无法为类创建默认构造函数。比如，不能为有引用或 `const` 类型数据成员类创建默认构造函数；再如，类 A 的一个数据成员是用类 B 创建的，但类 B 有其他构造函数，却没有默认构造函数，在这种情况下，类 A 必须定义构造函数，并负责为对象成员提供构造函数初值。

【例 3-7】 设计表示平面坐标位置的点类，可以修改和获取点的 `x`、`y` 坐标值，设置构造函数对点的数据成员进行初始化，并且能够用数组保存一系列的点。

问题分析与数据抽象：将点抽象成 `Point` 类，将它的坐标值 `x`、`y` 设置为私有数据成员，并设置接口函数 `setPoint()` 修改 `x`、`y` 的坐标值，设置接口函数 `getX()`、`getY()` 获取坐标点的 `x`、`y` 值，设置构造函数 `Point(int xx, int yy)` 初始化点的坐标值。由于要定义数组，而且已定义了有参数的构造函数，编译器就不会再创建默认构造函数了，必须显式定义默认构造函数，将坐标点初始化为 0。所以，`Point` 类图如图 3-7 所示。

完整的程序代码如下。

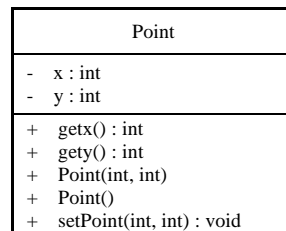


图 3-7 `Point` 类图

```
// Eg3-7.cpp
#include <iostream>
using namespace std;

class Point {
private:
    int x, y;
public:
    Point(int a, int b) { setPoint(a, b); }           // L1
    int getX() { return x; }
    int getY() { return y; }
    Point() { x = 0; y = 0; }                         // L2 显式定义无参构造函数
    void setPoint(int a, int b) { x = a; y = b; }
};

Point p0;                                           // L3, 调用构造函数 Point()
Point p1(1, 1);                                    // L4, 调用构造函数 Point(int, int)

void main() {
    static Point p2;                                // L5
    Point p3;                                        // L6, 调用构造函数 Point()
    Point a[10];                                    // L7, 调用构造函数 Point()
    Point *p4;                                       // L8, 不调用任何构造函数
    p4 = new Point;                                 // L9, 调用构造函数 Point()
    p4->setPoint(8, 9);
    cout<<"p0: "<<p0.getX()<<" "<<p0.getY()<<endl;
    cout<<"p1: "<<p1.getX()<<" "<<p1.getY()<<endl;    // L10
    cout<<"p2: "<<p2.getX()<<" "<<p2.getY()<<endl;
    cout<<"p3: "<<p3.getX()<<" "<<p3.getY()<<endl;
    cout<<"p4: "<<p4->getX()<<" "<<p4->getY()<<endl;
```

```

        cout<<"a[0]: "<<a[0].getx()<<" "<<a[0].gety()<<endl;
    }

```

程序运行结果如下。

```

p0: 0,0
p1: 1,1
p2: 0,0
p3: 0,0
p4: 8,9
a[0]: 0,0

```

在本程序中，语句 L3、L5、L6、L7 和 L9 调用无参数构造函数 Point()完成对象的构造，若将语句 L1、L2 和 L4 注释掉，则类 Point 没有任何构造函数，系统会为它生成一个默认构造函数：Point::Point(){}，以完成无参数对象 p0、p2、p3、*p4 和数组对象 a 的构造。

程序执行的结果如下：

```

P0: 0,0
P2: 0,0
P3: ?,?
P4: 8,9
a[0]: ?,?

```

其中的“?”表示值未知。

如果将语句 L2 注释掉，由于语句 L1 已经定义了带参数的构造函数，因此编译器不会再为类 Point 生成默认构造函数，语句 L3、L5、L6、L7 和 L9 就无法调用无参构造函数创建 p0、p2、p3、*p4 等对象，程序编译时将产生“找不到合适的构造函数……”之类的错误信息。

2. 默认参数构造函数

在实际程序中，有些构造函数的参数在多数情况下都比较固定，只是有时候会发生变化。可以为这样的类设置默认参数，将固定值设置为对应参数的默认值，称为**默认参数构造函数**。

【例 3-8】 在多数情况下，新建点坐标都是(0, 0)，修改例 3-7 设计的类 Point，设置构造函数的默认参数值为坐标(0, 0)。

```

// Eg3-8.cpp
#include <iostream>
using namespace std;

class Point {
private:
    int x, y;
public:
    Point(int a = 0, int b = 0) { setPoint(a, b); }
    // Point() { x = 0; y = 0; } // L1, 显式定义无参构造函数
    void setPoint(int a, int b) { x = a; y = b; }
    .....
};

Point p1(1, 1); // L2, 调用 point(int, int)构造函数

void main () {
    static Point p2; // L3, 调用 point(a, b), a、b 默认为 0
    Point p3, a[10]; // L4, 调用 point(a, b), a、b 默认为 0
}

```

```

    Point *p4;
    p4 = new Point;                                     // L5, 调用 point(a, b), a、b 默认为 0
    .....
}

```

如果显式定义了无参数构造函数，又定义了全部参数都有默认值的构造函数，就容易在定义对象时产生二义性。在本程序中，如果去掉语句 L1 的注释，语句 L3、L4、L5 将产生编译错误。因为 `Point()` 和 `Point(int a = 0, int b = 0)` 都可以定义对象 `p2`、`p3` 和 `*p4` 所指向的对象，系统不能确定应该调用哪个构造函数，因此会产生二义性冲突错误。

3.6.4 重载构造函数

在一个类中，构造函数可以重载。与普通函数的重载一样，重载的构造函数必须具有不同的函数原型（参数个数、参数类型或参数次序不能完全相同）。

【例 3-9】 设计一个日期类，能够接收年、月、日 3 个参数，或者月份和日期 2 个参数，或者日期 1 个参数，或没有参数，若未提供年、月、日，则设置为 2008 年 8 月 8 日。

问题分析与数据抽象：日期类的年、月、日可以用 `year`、`month`、`day` 三个数据成员表示，出于信息隐藏目的，将它们设置为 `private` 成员。围绕 3 个数据成员，可以设置 `setDay()`、`getDay()` 等读写数据的公有接口，同时设置接口函数 `dispDate()` 显示对象的年、月、日信息，题目还要求 `Tdate()` 能够通过 4 种方式建立对象，由于建立对象时需要调用构造函数。因此，可以用 4 个具有不同参数的构造函数来满足这些要求。因为要在其他函数中调用这些构造函数创建对象，所以必须将构造函数设置为 `public` 属性。`Tdate` 类图如图 3-8 所示。

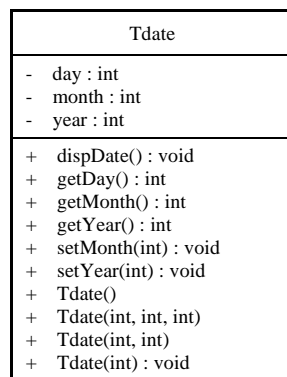


图 3-8 Tdate 类图

测试日期类构造函数重载的完整程序如下，其中省略了 `setDay()`、`setYear()`、`setMonth()`、`getMonth()` 等。

```

// Eg3-9.cpp
#include <iostream>
using namespace std;

class Tdate {
public:
    Tdate();
    Tdate(int d);
    Tdate(int m, int d);
    Tdate(int m, int d, int y);
    // .....                                     // 省略了设置和读取数据成员值的接口函数
    void display(){ cout<<"month<<"/"<<day<<"/"<<year<<endl; }
private:
    int year = 2008, month = 8, day = 8;          // L0, 类内初始值, C++ 11
};
Tdate::Tdate() { display(); }
Tdate::Tdate(int d) {
    day = d;
}

```

```

        display();
    }

    Tdate::Tdate(int m, int d) {
        month = m;
        day = d;
        display();
    }

    Tdate::Tdate(int m, int d, int y) {
        month = m;    day = d;    year = y;
        display();
    }

    void main() {
        Tdate oneday;                                // L1
        Tdate aday();                                // L2, 可以吗?
        Tdate bday1(10);                             // L3
        Tdate bday2 = 10;                             // L4
        Tdate cday(2, 12);                             // L5
        Tdate dday(1, 2, 1998);                       // L6
    }

```

语句 L0 为数据成员指定了类内初始值，将先于构造函数为数据成员指定初值。显然，如果构造函数修改了数据成员的值，就会覆盖类内初始值；如果构造函数没有为数据成员执行初始化，该数据成员就会保留类内初始值指定的值。

本程序运行结果如下，请读者结合上面的说明，分析各输出行数据的来源。

```

8/8/2008                                // L1 的输出
8/10/2008                               // L3 的输出
8/10/2008                               // L4 的输出
2/12/2008                               // L5 的输出
1/2/1998                                // L6 的输出

```

本例在各构造函数内调用成员函数 `display()` 并无任何意义，并不恰当，这里调用它只是为了测试各构造函数的调用情况。语句 L1 将调用构造函数 `Tdate()`，L3、L4 将调用构造函数 `Tdate(int)`，L5 将调用构造函数 `Tdate(int, int)`，L6 将调用构造函数 `Tdate(int, int, int)`。

语句 L2 不会调用任何构造函数，也不会定义任何对象。事实上，它声明了一个名为 `aday()` 的无参数函数，该函数返回一个 `Tdate` 类型的对象。

注意：L4 形式的对象定义语句“`Tdate bday2 = 10;`”调用的是构造函数 `Tdate::Tdate(int)`，把一个 `int` 类型的整数转换成一个 `Tdate` 类型的对象，等价于“`Tdate bday2(10);`”。仅当类提供了只需要一个参数的构造函数的情况下，才能使用 L4 这样的定义形式。

在一些情况下，可以用带默认参数的构造函数来替代重载构造函数，达到相同的效果。如上面的类 `Tdate` 可以用一个带默认参数的构造函数来替代所有的重载构造函数，如下所示：

```

class Tdate {
public:
    Tdate(int m = 8, int d = 8, int y = 2008) {
        month = m;
        day = d;
    }

```



```

        year = y;
        display();
    }
    .....
};
// 其他成员

```

虽然类 `Tdate` 看上去很简洁，但它几乎具有例 3-9 中类 `Tdate` 全部构造函数的功能。

3.6.5 构造函数与初始化列表

除了在函数体中通过赋值语句为数据成员赋初值，构造函数还可以采用成员初始化列表的方式对数据成员进行初始化。而且在某些情况下，必须采用初始化列表的方式才能够完成成员的初始化。成员初始化列表类似于如下形式：

```

构造函数名(参数表): 成员 1(初始值), 成员 2(初始值), ... {
    .....
}

```

构造函数参数表后面的 “:” 与函数体 { ... } 之间的内容就是成员初始化列表。其含义是将 “()” 中的初始值赋给它前面的成员。

【例 3-10】 用构造函数初始化列表对例 3-9 设计的类 `Tdate` 的成员 `month` 进行初始化。

```

// Eg3-10.cpp
#include <iostream>
using namespace std;

class Tdate {
public:
    Tdate(int m, int d, int y);
    .....
protected:
    int month, day = 30, year;
};
Tdate::Tdate(int m, int d, int y):month(m) {
    year = y;
    cout<<month<<"/"<<day<<"/"<<year<<endl;
}

void main() {
    Tdate bday2(2, 1, 2024);
}
// 其他公共成员
// d 并没有被采用
// 输出: 2/30/2024 (错误)

```

`Tdate` 类的数据成员 `month`、`day` 与 `year` 的初始化方式是不同的，`month` 采用初始化列表方式进行初始化，`day` 采用类内初始值，`year` 采用的是普通函数的初始化方式。

说明：

① 构造函数初始化列表中的成员初始化次序与它们在类中的声明次序相同，与初始化列表中的次序无关。如对例 3-10 中的类而言，下面三个构造函数是完全相同的。

```

Tdate::Tdate(int m,int d,int y):month(m),day(d),year(y){ }
Tdate::Tdate(int m,int d,int y):year(y),month(m),day(d){ }
Tdate::Tdate(int m,int d,int y):day(d),year(y),month(m){ }

```

尽管三个构造函数初始化列表中的 `month`、`day` 和 `year` 的次序不同,但它们都是按照 `month` → `day` → `year` 的次序初始化的,这个次序是其在类 `Tdate` 中的声明次序。它们在功能上与如下构造函数等效:

```
Tdate::Tdate(int m, int d, int y) {  
    month = m;  
    year = y;  
    day = d;  
}
```

② 构造函数初始化列表的执行次序。如果数据成员有类内初始值,那么执行次序为:类内初始值 → 构造函数初始化列表 → 构造函数体。

在一个类中,下列成员必须采用类内初始值或构造函数初始化列表进行初始化:常量成员、引用成员、类对象成员,以及派生类构造函数对基类构造函数的调用等。**注意:**类内初始值是从 C++ 11 标准才开始有的,此前这些成员必须以构造函数初始化列表的方式初始化。

【例 3-11】 常量和引用成员必须通过类内初始值或构造函数初始化列表进行初始化。

```
// Eg3-11.cpp  
#include <iostream>  
using namespace std;  
class A {  
    int x, y;  
    const int i = 4, j; // C++ 11 之前不允许, C++ 11  
    int &k;  
public:  
    A(int a, int b, int c) : j(b), k(c), x(y) {  
        y = a;  
        cout<<"x = "<<x<<"\t"<<"y = "<<y<<endl;  
        cout<<"i = "<<i<<"\t"<<"j = "<<j<<"\t"<<"k = "<<k<<endl;  
    }  
};  
void main() {  
    int m = 6;  
    A x(4, 5, m);  
}
```

本程序的运行结果如下:

```
x = ?    y = 4  
i = 4    j = 5    k = 6
```

“?”表示值未知。构造函数初始列表中的 `x(y)`表示用 `y` 的值初始化 `x`,由于初始化列表先于构造函数体执行且 `y` 未执行类内初始化,因为此时还没有执行构造函数体中的“`y = a;`”语句, `y` 的值未知,致使 `x` 未知。当构造函数初始列表执行完后,再执行函数体,才将参数值 4 赋给 `y`。

本类的 `i`、`j`、`k` 都是引用或 `const` 成员,必须采用类内初始值或构造函数初始化列表的方式进行初始化,其他方式都是错误的。例如,若将 `A` 的构造函数改写为如下初始化方式,程序将出现编译错误。

```
A(int a, int b, int x){ j = b;    k = x; }
```

作为构造函数初始列表与函数体执行次序的验证，将例 3-11 的构造函数改为如下形式，其余代码不做任何修改，则 x 和 y 都将为 4，从而表明 x(a)的确先于“y = x;”执行。

```
A(int a, int b, int c):j(b), k(c), x(a) {
    y = x;
    .....
}
```

采用构造函数初始化列表方式与在构造函数体内赋值的方式进行数据成员的初始化虽然结果相同，但列表方式是在定义时直接初始化数据成员，赋值方式是先初始化再赋值，效率更低。

3.6.6 委托构造函数 C++11

一些类具有多个构造函数且各构造函数中可能具有一些共同的程序代码，C++ 11 标准之前的常用技术方法是用这些代码构建一个成员函数，然后在需要的构造函数中调用它。C++ 11 标准提出了委托构造函数，能够更好地解决这个问题。其思想是，用被重复使用的代码设计一个构造函数，并在其他构造函数中调用它。一个构造函数使用其他构造函数实现其需要的功能，或者说一个构造函数把它自己的一些（或全部）职责委托给其他构造函数完成，就称为委托构造函数（delegating constructor function）。

委托构造函数只能够在初始化列表中调用它要委托的构造函数，而且初始化列表中不允许再有其他的成员初始化列表，但委托构造函数的函数体中可以有程序代码。

【例 3-12】 改造例 3-9 设计的类 Tdate 的无参数、一个参数和两个参数的构造函数，它们都委托具有三个参数的构造函数实现自己的功能。改造后的程序代码如下：

```
// Eg3-12.cpp
#include <iostream>
using namespace std;

class Tdate {
public:
    Tdate();
    Tdate(int d);
    Tdate(int m, int d);
    Tdate(int m, int d, int y);
    // ..... // 省略了设置和读取数据成员值的接口函数
    void display() { cout<<"month<<"/"<<"day<<"/"<<"year<<endl; }
private:
    int year = 2008, month = 8, day = 8; // C++ 11
};

Tdate::Tdate() :Tdate(8, 1, 2008) { // L1, 委托构造函数
    cout<<"delegating constructor Tdate()"<<endl;
}

Tdate::Tdate(int d):Tdate(8,d,2008), month(2) { } // L2, 错误
Tdate::Tdate(int m, int d):Tdate(m, d, 2008) { } // L3, 委托构造函数
Tdate::Tdate(int m, int d, int y) { // L4, 普通构造函数
    month = m;    day = d;    year = y;
    display();
}
```

```

void main() {
    Tdate oneday;
    Tdate bday1(10);
    Tdate bday2 = 10;
    Tdate cday(2, 12);
    Tdate dday(1, 2, 1998);
}

```

这个程序具有与例 3-9 完全相同的功能。语句 L1、L2、L3 位置的都是委托构造函数，它们委托了语句 L4 的构造函数完成自己的职责。语句 L1 的委托构造函数体有程序代码，它在被委托构造函数 Tdate(8, 1, 2008) 执行后才会被执行。语句 L2 错误的原因是委托构造函数初始化列表不允许有成员初始化列表，应该删掉“month(2)”。

3.7 析构函数

析构函数与构造函数是一对特殊的函数，都由系统自动调用，成对使用，功能相反。在创建对象时，系统自动调用构造函数进行对象所需资源的分配。在对象结束生命期时，系统自动调用析构函数，释放对象所占用的资源。每个类必须有构造函数和析构函数，如果没有为该提供构造函数或析构函数，C++ 编译器就会自动为该添加一个默认的构造函数或析构函数。

3.7.1 析构函数的设计思想和定义

析构函数（**destructor function**）用于在对象生命期结束时完成对象的清理工作，作用与构造函数相反，所以析构函数的名称就是在默认构造函数名的前面加符号“~”，以强调它与构造函数是一对互补的函数。例如，在构造函数中用 **new** 为对象分配了堆内存，就应该在析构函数中用 **delete** 释放对象分配的堆内存；如果在构造函数中为对象分配了网络资源，建立了远程连接，就应该在析构造数中释放对象占用的网络资源，断开网络连接，如此等等。

析构函数的名称由“~”+“类名”构成，形式如下：

```

class X {
    .....
public:
    ~X();                      // 析构函数
    .....
};

```

在类外定义析构函数的形式如下：

```

X::~~X() {
    .....
}

```

析构函数的特点如下：① 析构函数的名称是在类名前加上“~”，不能是其他名称；② 析构函数没有返回类型（**void** 也不行），没有参数表；③ 析构函数不能重载，一个类只能有一个析构函数；④ 析构函数只能由系统自动调用，不能在程序中显式调用析构函数。

当创建一个对象时，C++ 将首先为数据成员分配存储空间，接着调用构造函数对成员进

行初始化工作；当对象生命期结束时，C++将自动调用析构函数清理对象所占据的存储空间，然后销毁对象。

【例 3-13】 析构函数的应用。

```
// Eg3-13.cpp
#include <iostream>
using namespace std;

class A {
private:
    int i;
public:
    A(int x) {
        i = x;
        cout<<"constructor: "<<i<<endl;
    }
    ~A(){ cout<<"destructor : "<<i<<endl; }
};

void main() {
    A a1(1);
    A a2(2);
    A a3(3);
} // L1
```

本程序的运行结果如下：

```
constructor: 1
constructor: 2
constructor: 3
destructor : 3
destructor : 2
destructor : 1
```

当程序执行到语句 L1 位置时，所有对象的生命期到此结束，将按照与构造相反的次序调用析构函数完成对象销毁前的清理工作，程序的运行结果也证实了这一情况。

说明：

- ① 若有多个对象同时结束生命期，则将按照与调用构造函数相反的次序调用析构函数。
- ② 构造函数和析构函数都可以是内联函数。
- ③ 虽然析构函数和构造函数都只能被系统自动调用，但这些调用都是在类的外部进行的，应该将它们设置为类的公共成员。
- ④ 每个类都应该有一个析构函数。

如果没有显式定义析构函数，C++编译器将产生一个最小化的默认析构函数（见 3.6.1 节），类似如下情况：

```
X::~~X(){ }
```

通常，默认析构函数能够满足对象析构的要求。但有些情况下，必须编写析构函数才能够完成对象销毁前的资源回收工作，如必须用它来释放由构造函数分配的自由存储空间。

【例 3-14】 类 B 有指针成员，且构造函数为之分配了自由存储空间，就应该用析构函数

回收构造函数分配的自由存储空间。

```
// Eg3-14.cpp
#include <iostream>
using namespace std;

class B {
private:
    int *a;
    char *pc;
public:
    inline B(int x) {
        a = new int[10];
        pc = new char;
    }
    inline ~B() {
        delete []a;
        delete pc;
    }
};

void main() {
    B x(1);
}
```

对于例 3-14 而言，如果没有析构函数，程序也能正常运行。但是，当用类 **B** 建立的对象结束生命期时，系统不会回收由 **B** 的构造函数分配的自由存储空间，会产生内存泄漏。本例是应该为类提供析构函数的典型情况。即，若在构造函数中用 **new** 或 **malloc()** 分配了存储空间，就应该在析构函数中用 **delete** 或 **free()** 释放这些存储空间。

最后，简要介绍设计构造函数时应牢记的两条原则和 **RAII** 范式。

原则 1：不要在析构函数中抛出异常（异常见第 8 章）。析构函数是在对象被销毁时才被执行的函数，在其中抛出的异常将无法被捕获，因此会导致程序崩溃。在 C++ 11 及以上的标准中，析构函数中的异常会导致程序产生运行时错误并中断程序的执行，而在 C++ 03 标准中则取决于编译器的具体实现，最常见的行为是程序直接退出运行。

原则 2：若类中有虚函数（见第 5 章），则析构函数也必须是虚函数。

RAII（**Resource Acquisition Is Initialization**，资源获取即初始化）范式是 C++ 发明者 Bjarne Stroustrup 提出的一种设计范式，基本思想是绑定资源到对象，并利用构造和析构机制自动处理资源。每当想要获取一个资源（内存、文件、数据库、网络连接……）时，就创建一个对象（通过构造函数获取这些资源），当对象退出作用域时，通过析构函数自动释放这些资源。这样就能够保障资源管理的正确性和便捷性。

3.7.2 弱指针与析构函数

在现代编程中，程序员更愿意用智能指针管理对象的内存资源，这不仅可以减少指针处理的麻烦，还能够避免不小心所造成的内存泄漏。但是，在用共享指针管理对象的堆内存时，共享指针（**shared_ptr**）可能产生循环引用，导致死锁或对象资源不能回收的问题，用弱指针

(weak_ptr) 可以协助共享指针解决此问题 (见 2.3.4 节)。

【例 3-15】 用弱指针解决共享指针循环引用产生的内存泄漏问题。

```
// Eg3-15.cpp
#include <iostream>
#include <memory>
using namespace std;

class B; // L1, 前向声明
class A {
public:
    A() { cout<<"A Constructor ..."<<endl; }
    ~A() { cout<<"A Destructor ..."<<endl; }
    shared_ptr<B> pb; // L2, 在 A 中引用 B
    // weak_ptr<B>pb; // L3, 弱指针
};
class B{
public:
    B() { cout<<"B Constructor ..."<<endl; }
    ~B() { cout<<"B Destructor ..."<<endl; }
    shared_ptr<A> pa; // L4, 在 B 中引用 A
};
int main() {
    shared_ptr<A> spa{ new A() }; // L5
    shared_ptr<B> spb{ new B() }; // L6
    spa->pb = spb; // L7, 循环引用了 A
    spb->pa = spa; // L8, 循环引用了 B
    cout<<"spa.count = "<<spa.use_count()<<"\tspb.count = "<<spb.use_count()<<endl;
}
```

执行程序, 结果如下:

```
A Constructor ...
B Constructor ...
spa.count = 2   spb.count = 2
```

这个结果表明析构函数有被执行, 如果需要在析构函数中释放对象占用的资源, 如关闭文件, 断开数据库连接, 向服务器传送文件并断开网络连接等操作, 就不会被执行了, 显然是一种会浪费系统资源的问题程序。根源就是共享指针的循环引用, 如图 3-9 所示。

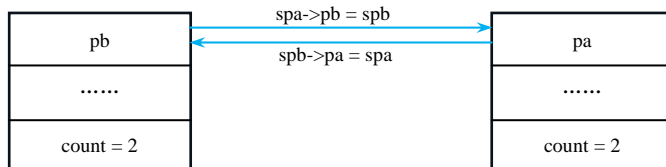


图 3-9 share_ptr 指针的循环引用问题

共享指针回收其占用堆内存的时机是对象的引用计数为 0。语句 L5 用 new 为 spa 分配了堆内存, 并用 spa 指向该内存, 所以 spa 的引用计数 count 就是 1; 语句 L8 再次引用了 spa, 所以 spa 的引用计数 count 变为 2。同理可知, spb 的引用计数 count 也为 2。当函数 main() 结束时, spa 和 spb 结束生命期, 它们的引用计数减 1, 都变成了 1, 但是都不是 0, 所以不会

调用它们的析构函数，导致 `new A()` 和 `new B()` 分配的堆内容未被回收，产生了内存泄漏。

弱指针主要是为解决共享指针循环引用而设计的，可以用共享指针构造，也可以把弱指针变量赋值给它，但不会改变共享内存的引用计数。在本例中，用注释掉的语句 L3 取代语句 L2，即用弱指针定义类 A 的 pb，其余程序代码不进行任何修改。程序运行结果如下：

```
A Constructor ...
B Constructor ...
spa.count = 2    spb.count = 1
B Destructor ...
A Destructor ...
```

当将类 A 的 pb 定义为弱指针后，对于 “`spa->pb = spb`” (L7)，由于 pb 是弱指针，当它引用 spb 时不会增 spb 的引用计数，因此 spb 的 count 仍然是 1。当函数 `main()` 结束时，spb 失去作用域和生命期，其引用计数减 1，count = 0，因此要调用它的析构函数。由于 spb 引用了 spa，现在 spb 被析构了，因此引用 spa 的引用计数也要减 1，再加上 spa 也失去作用域和生命期，引用计数再减 1，因此也变成 0，所以它的析构函数也会被调用。

本例中，将 B 类的 pa 设置为弱指针，或者 pa 和 pb 都设置为弱指针，都能够解决问题。

3.8 赋值运算符函数、复制构造函数和移动函数设计

同类对象之间的赋值和复制操作需要通过类的赋值运算符函数和复制构造函数完成。C++ 11 标准还提出了这两个函数的移动函数版本，以便在某些情况下执行对象移动操作。在面向对象程序设计过程中，对象的赋值、复制和移动很普遍，以致每个类都应该具有这些成员函数，如果在设计类时没有显式地定义它们，编译器就会自动为该类生成赋值运算符函数、复制构造函数、移动函数（移动复制构造函数、移动赋值运算符函数），从而定义各函数的默认操作。

在大多数情况下，由系统自动为类添加的默认成员函数能够胜任其工作，完成对象的赋值、复制和移动操作。但在某些情况下，默认函数会产生问题。典型情况是当类具有指针类型数据成员时，依赖编译器自动生成的赋值运算符函数进行对象赋值，或复制构造函数进行对象复制，都会产生“指针悬挂”问题，必须显式定义类的赋值运算符函数和复制构造函数。

一般，如果一个类需要显式地定义析构函数，就需要为它显式地定义赋值运算符函数和复制构造函数。

3.8.1 赋值运算符函数

赋值运算符用于实现同类对象间的相互赋值。当把类的一个对象赋值给另一个对象时，就会调用类的赋值运算符成员函数来完成对象间的赋值。类似如下形式：

```
class A{...};
A a, b;
a = b;                                     // 调用赋值运算符函数
```

这里的 “=” 即赋值运算符，它是所有类都拥有的一个成员函数，称为赋值运算符成员函数，功能是把 “=” 右边对象的数据成员复制给左边对象。

1. 默认赋值运算符函数

一个类如果没有显式定义赋值运算符函数，C++编译器就会为该类生成一个默认赋值运算符成员函数，以按位复制（bit-by-bit）的方式实现对象非静态数据成员的复制，即把赋值号右边对象的数据成员值原样复制到赋值号左边对象的对应数据成员中。按位复制就是把一个对象各非静态数据成员的值原样复制到目标对象中。

当把一个对象赋值给同类的另一个对象时，C++将按照如下步骤完成赋值操作：

<1> 查找该类是否定义了赋值运算符函数，如果有且是可访问的（**public** 成员），就调用此赋值运算符成员函数进行对象赋值；如果定义了但不是可访问的（**private** 或 **protected** 成员），就产生编译错误。

<2> 如果该类没有显式定义赋值运算符函数，编译器就自动为该类生成一个赋值运算符函数，执行默认的赋值操作，称为**默认赋值运算符函数**。在通常情况下，默认赋值运算符函数足以解决对象之间的赋值问题。但是，当类包含指针类型数据成员时，默认赋值运算符函数通常会引发“指针悬挂”问题。

<3> 具有非静态的引用、**const** 和 **unique_ptr** 指针成员类不能够被复制构造或复制赋值，编译器不会为这样的类生成默认的复制构造函数和赋值运算符函数，如果要复制，就需要用其他类的指针成员，如 **shared_ptr** 或普通指针。

<4> 编译器不会为定义了移动复制构造函数的类生成默认的复制构造函数和赋值运算符函数。

```
class MyClass {
    int x1;
public:
    MyClass() {}
    MyClass(MyClass&&) = default;           // L1, 移动赋值
};
int main() {
    MyClass m1, m2;
    m1 = m2;                               // L2, 错误
}
```

语句 L2 是错误的，因为这个操作需要通过赋值运算符函数实现，但语句 L1 定义了移动赋值构造函数，所以编译器就不再为该类生成默认的赋值运算符函数，即不能执行“=”了。要执行就需要重载赋值运算符函数：**operator =()**，或者去掉语句 L1。

【例 3-16】 字符串类 **String** 具有指针数据成员 **ptr** 用于存放字符串内容，**n** 存放字符串编号，但没有重载赋值运算符函数，编译器生成的默认赋值运算符函数产生了“悬挂指针”。

```
// Eg3-16.cpp
#include <iostream>
#include <string>
using namespace std;

class String {
    char *ptr;
    int n;
public:
    String(char *s, int a) {
```

```

        ptr = new char[strlen(s)+1];
        strcpy(ptr, s);
        n = a;
    }
    ~String(){ delete []ptr; }
    void print(){ cout<<ptr<<endl; }
};

void main() {
    String p1((char*)"Hello", 8);           // L1
    {
        String p2((char*)"chong qing", 10); // L2
        p2 = p1;                             // L3
        cout<<"p2: ";                         // L4
        p2.print();                           // L5
    }                                         // L6
    cout<<"p1: ";                             // L7
    p1.print();                             // L8, 错误
}                                           // L9

```

本程序在运行时会产生错误信息，错误发生在语句 L9 处。下面是运行结果：

```

p2: Hello
p1: 茸茸茸茸茸

```

输出第 1 行是语句 L4 和 L5 产生的，第 2 行是语句 L7 和 L8 产生。“p1:”后面的输出是没有意义的乱码数据，程序执行到语句 L9 时，将产生指针悬挂的错误信息。错误原因是：当执行“p2 = p1;”时，String 类没有提供显式的赋值运算符函数，C++ 将为它生成默认赋值运算符函数，并调用它进行对象赋值。编译器为 String 类生成的赋值运算符函数类似如下形式：

```

String& String::operator = (const String &s) {
    ptr = s.ptr;
    n = s.n;
    return *this;
}

```

“p2 = p1;”操作相当于执行如下两条语句：

```

p2.n = p1.n;           // L10
p2.ptr = p1.ptr;       // L11

```

语句 L10 没有问题，但语句 L11 将使 p2 和 p1 对象的成员指针 ptr 指向同一动态存储区域，如图 3-10(b) 所示。可以看出，“p2 = p1;”执行后，p2 和 p1 的 ptr 成员指针都指向了相同的内存单元，当程序执行到 L6 指示的位置时，p2 对象的生命期结束，调用 p2 的析构函数回收 p2.ptr 指向的动态存储区域。当 p2.ptr 指向的动态存储单元被回收后，p1.ptr 仍然指向该内存区域，但该区域已不可用，这就是所谓的“**指针悬挂**”问题。

程序运行到 L9 位置时，p1 对象的生命期结束，将调用 p1 的析构函数。执行 p1 析构函数中的“delete []ptr;”语句时将产生错误，因为 p1.ptr 指向的内存已被 p2 的析构函数回收了。

2. 重载赋值运算符函数

当类没有指针成员时，编译器生成的默认赋值运算符函数能够正确地实现对象之间的赋值操作。但是，在设计类似 String 这样的包含指针数据成员的类时，编译器生成的默认赋值

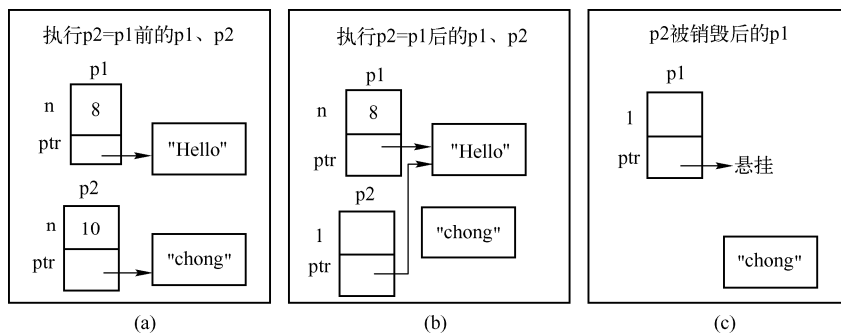


图 3-10 默认赋值操作运算符产生的指针悬挂问题

运算符函数不能够正确地实施对象之间的赋值，必须为它显式提供赋值运算符成员函数，才能实现对象之间的复制赋值操作。运算符函数是类的特殊成员函数，有独特的定义语法，所有运算符函数的名称都是 `operatorXX`。`XX` 代指运算符本身，如 `operator+`、`operator-` 分别是 “+” 和 “-” 运算符函数的名称。

赋值运算符是一个二元运算符，常返回本类对象的引用，其定义形式如下：

```
class X {
    .....
    X& operator = (const X &source, ...);
};
```

`operator = ()` 可以有多个参数。若有多个参数，则要求除第一个参数外的其余参数都要有默认值。第一个参数必须是自身类类型的引用，这个参数通常是 `const` 类型（但不是必须的，设为 `const` 只是为了避免函数中的误操作修改了 “=” 右边的对象）。

【例 3-17】 定义类 `String` 的赋值运算符函数，解决赋值操作引起的指针悬挂问题。

为类 `String` 增加赋值运算符的重载函数，省略的代码与例 3-16 中的完全相同。

```
// Eg3-17.cpp
#include <iostream>
#include <string>
using namespace std;

class String {
    .....
public:
    String& operator = (const String& s);           // 重载赋值运算符函数
    .....
};
.....
String& String::operator = (const String& s) {
    if(this == &s)
        return *this;
    delete []ptr;
    ptr = new char[strlen(s.ptr) + 1];
    strcpy(ptr, s.ptr);
    return *this;
}

void main() {
```

```
.....  
}
```

这次程序就没有错误了，运行程序将产生如下输出结果：

```
p2: Hello  
p1: Hello
```

3.8.2 复制构造函数

在用已经存在的对象初始化新建对象时，会调用复制构造函数完成对象数据成员的复制，该操作在面向对象程序设计中非常普遍。因此，在设计类时必须考虑复制构造函数的设计问题。比如，以下几种情况都会调用复制构造函数。

```
class X{ };  
X obj1;  
X obj2 = obj1;           // 情况 1: 调用复制构造函数  
X obj3(obj1);            // 情况 2: 调用复制构造函数  
f(X o);                  // 情况 3: 以对象作函数参数时，调用复制构造函数  
X f() {  
    X t;  
    .....  
    return t;             // 情况 4: 返回类对象时会调用复制构造函数  
}  
X a[4] = {obj1, obj2}     // 情况 5: a[0]、a[1]调用复制构造函数，a[2]、a[3]调用默认构造函数
```

1. 默认复制构造函数及指针悬挂问题

每个类都应该有一个复制构造函数，如果没有定义类的复制构造函数，在需要时，编译器将为其创建一个具有最小功能的复制构造函数，称为**默认复制构造函数**。形式如下：

```
X::X(const X&, ...) { }
```

如果有多个参数，要求第一个参数必须是自身类类型的引用，其余参数必须有默认值。

默认复制构造函数以成员按位复制的方式实现成员的复制。在没有指针类型的数据成员时，默认复制构造函数能够很好地工作，但当一个类有指针类型的数据成员时，会产生“指针悬挂”问题。

【例 3-18】 Person 是处理人员姓名和年龄的类，其中的姓名用字符串指针类型的数据成员 name 处理，由于没有考虑对象复制时指针成员的特殊性而重定义复制构造函数，默认复制构造函数引起了“指针悬挂”问题。

```
// Eg3-18.cpp  
#include <iostream>  
#include<string>  
using namespace std;  
  
class Person {  
private:  
    char *name;  
    int age;  
public:  
    Person(char *Name, int Age);
```

```

~Person();
void setAge(int x) { age = x; }
void print();
};
Person::Person(char *Name, int Age) {
    name = new char[strlen(Name) + 1];
    strcpy(name, Name);
    age = Age;
    cout<<"constructor ..."<<endl;
}
Person::~Person() {
    cout<<"destructor ..."<<age<<endl;
    delete []name;
}
void Person::print(){
    cout<<name<< "\t The Address of name: "<<&*name<<endl;
}
void main() {
    Person p1((char*)"张勇", 21); // L1
    Person p2 = p1; // L2
    p1.setAge(1);
    p2.setAge(2);
    p1.print();
    p2.print();
}

```

这个程序存在问题，在不同编译环境下会产生类似的错误。在 VC 6.0 环境下运行时，在输出如下内容之后会弹出一个错误信息对话框，显示程序试图删除一个空指针。

```

constructor ....
张勇 The Address of name: 张勇
张勇 The Address of name: 张勇
destructor...2
destructor...1

```

输出结果表明程序只调用了一次构造函数，但调用了两次析构函数。输出结果的第 2、3 行分别是 p1.print 和 p2.print 产生的，表明 p1 和 p2 的 name 成员指向了同一个内存地址。

构造函数的调用是执行 “Person p1(“张勇”, 21);” 发生的，语句 L2 将调用复制构造函数进行 p2 的初始化。因为类 Person 没有定义复制构造函数，所以 C++编译器将为它生成一个默认复制构造函数，以成员按位复制的方式将 p1 各数据成员的值复制到 p2 的对应成员。对于非指针类型的数据成员 age 而言，这样的复制并没有什么问题。但在复制指针成员 name 时就出问题了，它会将 p1.name 的值复制到 p2.name，导致 p2 和 p1 的 name 成员指向了同一个内存地址，如图 3-11 左图所示。

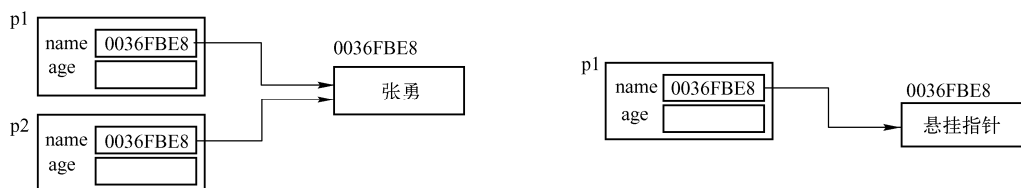


图 3-11 复制构造函数引起的指针悬挂问题

当遇到 main 最后的 “}” 时，将首先调用 p2 的析构函数，其中的语句 “delete []name;” 将把 p2.name 所指向的自由存储单元归还系统，但是 p1.name 此时仍指向此存储区域，即 “指针悬挂” 问题，如图 3-10 右图所示。

接下来系统将调用 p1 的析构函数，这次语句 “delete []name;” 就出问题了，原因是 p1.name 所指向的存储区域已经被 p2 的析构函数释放了，不能再次释放。

2. 定义复制构造函数

如果类没有指针成员，默认复制构造函数能够胜任其工作。但是如果类存在指针类型的数据成员，通常需要显式定义复制构造函数。

【例 3-19】 为例 3-18 的类 Person 定义复制构造函数，解决对象复制时产生的指针悬挂问题。

在例 3-18 的类 Person 中增加复制构造函数的定义。省略部分与例 3-18 中的代码相同。

```
// Eg3-19.cpp
.....
class Person {
    .....
public:
    Person(const Person &p);           // 复制构造函数
    .....
};
Person:: Person(const Person &p) {
    if(this == &p)
        return;
    name = new char[strlen(p.name) + 1];
    strcpy(name, p.name);
    age = p.age;
    cout<<"Copy constructor ... "<<endl;
}
.....
void main(){ ... }
```

编译并运行该程序，这次不会有错误，将产生如下输出结果：

```
constructor ...
Copy constructor ...
张勇      The Address of name: 张勇
张勇      The Address of name: 张勇
destructor ... 2
destructor ... 1
```

程序运行结果的第 2 行输出表明，在定义 p2 时调用了类 Person 的复制构造函数。

3. 复制函数的应用说明

- ① 复制构造函数与一般构造函数相同，与类同名，没有返回类型，可以重载。
- ② 复制构造函数的参数常常是 const 类型的本类对象的引用。
- ③ 要注意区分复制构造函数与赋值运算符成员函数的调用时机：在把一个对象赋值给已定义对象好的对象时，调用赋值运算符函数；当时正在定义新对象，但要用已建好的对象初

始化该新对象时调用复制构造函数。容易把如下语句 L2 误会为调用赋值运算符函数，请注意区分。

```
class X{...};
X obj1;
X obj2;
obj2 = obj1;           // L1, obj2 已定义了，把 obj1 赋值给 obj2，调用赋值运算符函数
X obj3 = obj1;         // L2, 新建 obj3，并用 obj1 初始化 obj3，调用复制构造函数
```

④ 当类具有指针类型的数据成员时，默认复制构造函数就可能产生指针悬挂问题，需要提供显式的复制构造函数。在其他情况下，默认复制构造函数就能完成对象的创建工作了。

对复制构造函数的调用常在类的外部进行，应该将它指定为类的公共成员。

3.8.3 移动函数 C++ 11

1. 对象移动的概念

程序设计中的对象复制是一件普通而常见操作，但在某些情况下，对象复制后就立即被销毁了，这样的对象就是临时对象，如以下代码中的函数 f() 所示。

```
class A{...};
A f() {
    A t;
    .....
    return t;
}
A b;
b = f();
```

语句“b = f();”的执行过程是：<1> 调用函数 f()，执行其函数体的代码；<2> 执行“return t;”，将创建无名临时对象并用 t 初始化，然后析构 t，返回调用语句；<3> 将无名对象复制赋值给 b；<4> 销毁无名临时对象。

显然，无名临时对象会占用系统资源（资源的多少与其数据成员相关，如有大容量的数组成员就会占用大量的存储空间），分配和回收这些资源都会占用系统时间和资源，复制赋值给另一个对象同样会耗占系统资源。

C++ 11 标准提出了采用对象移动而非复制的新技术来解决临时对象的复制问题，可以极大地提高程序性能。对象移动与对象复制操作的过程基本相同，区别在于第<3>步。

采用对象移动技术执行“b = f();”语句的第<3>步是将临时对象的资源“转移”给对象 b，其余操作相同。这个过程节省了把临时对象的资源复制给对象 b 的系统开销，如图 3-12 所示。临时对象的资源被转移后，就不要对它曾经持有的资源有任何期望，如同现实中的资产转移后，其所有权归新拥有者所有，原来的拥有者不应该再处理它。

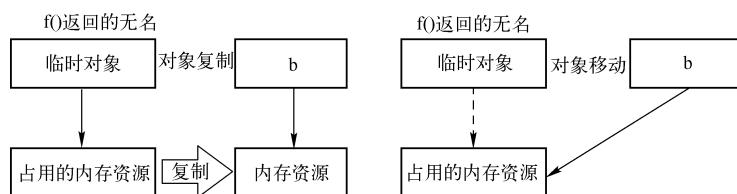


图 3-12 对象复制和对象移动的对比

2. C++标准库中的 move()函数

对象移动相当于把某对象拥有的内存资源“转让”给另一对象使用，实质是把对象的内存资源（右值）绑定到要转移给的对象。由于变量名对应内存的左值，不能直接绑定到右值。C++ 11 标准库提供了移动函数实现对象的右值绑定，此函数定义在 `utility` 头文件中。

```
int x = 0;
int &lrx = x;           // 正确，左值引用
int &&rrx = x;          // 错误，变量名是左值，不能绑定右值
int &&rrx = std::move(x); // 正确，rrx 绑定到 x 的右值
```

函数 `move()` 不仅可以绑定内置数据类型的右值，也可以绑定用户自定义类型的右值。

【例 3-20】 用 C++ 标准库的移动函数移动对象的右值资源。

```
// Eg3-20.cpp
#include <iostream>
#include <string>
#include <utility>
using namespace std;

class A {
    int a;
public:
    void setA(int x) { a = x; }
    int getA() { return a; }
};

void main() {
    A b;
    // A &&r = b;           // L1 错误
    A &&r = move(b);        // L2 正确
    r.setA(8);
    cout<<b.getA()<<"\t"<<r.getA()<<endl; // L3
    int x = 9;
    int &&rx = std::move(x);
    cout<<"rx = "<<rx<<"\tx = "<<x<<endl; // L4
    cout<<"rx Addr: "<<&rx<<"\t\tx Addr: "<<&x<<endl; // L5
}
```

程序运行结果如下：

```
8      8           // L3 语句输出
rx = 9   x = 9      // L4 语句输出
rx Addr: 002EFBA4    x Addr: 002EFBA4 // L5 语句输出
```

输出结果表明，用函数 `move()` 从源对象移动内存资源给新对象后，并不会销毁源对象，新对象“接管”了源对象的内存资源，但仍然可以通过源对象操作对应的内存资源，可以访问，也可以赋值，如语句 L3 的“`b.getA()`”。

但是，用函数 `move()` 移动资源实际上是对源对象的一种承诺：将 A 对象的资源移动给另一个对象后，A 就不再使用原对象。所以，用它来移动临时对象的资源是非常恰当的用法，因为临时对象在其资源被转移后，即刻失去生命期而被销毁。

3. 移动赋值运算符函数和移动复制构造函数

在对象赋值和新对象初始化时，都可以执行对象移动操作，用转移对象资源的方式取代复制资源的方式，将一个对象的内存右值转移给另一个对象操控。如果来实现对象移动，就需要为类定义移动赋值运算符函数和移动复制构造函数。形式如下：

```
class A {
    .....
    A(A&& o) {...}                // 移动复制构造函数
    A &operator = (A&& o) {...}    // 移动赋值运算符函数
};
```

与赋值运算符函数和复制构造函数类似，如果一个类没有定义这些函数，编译器就会尝试为该类定义默认移动复制构造函数和默认移动赋值运算符函数。但是，如果一个类定义了赋值运算符函数、复制构造函数或者析构函数，编译器就不会为它生成默认的移动构造函数和移动赋值运算符函数。只有当一个类没有定义这些函数且每个非静态数据成员都可以移动（内置数据类型是可移动的，如果数据成员是自定义类类型，只有当它也定义了移动函数时，才是可移动的）时，编译器才会生成默认的移动构造函数和移动赋值运算符函数。例如：

```
struct A {
    int x;                        // 内置类型可以移动
    std::string s;                // string 定义了移动操作
};
class B {
    A a;                          // A 有默认移动函数
};
class C {
    A a;
public:
    C() {}
    C(C&o) {}                      // 定义了复制构造函数，不会有默认移动函数
};
B a1, a2 = std::move(a1);         // a2 使用默认移动构造函数
C c1, c2 = std::move(c1);         // c2 使用复制构造函数
```

在上面的代码段中，类 A、B 的数据成员都是可移动的，它们没有定义赋值运算符函数、复制构造函数和析构函数，编译器会为它们生成移动赋值运算符和移动复制构造函数。因此，定义 a2 时，函数 move() 调用默认移动函数，将 a1 的内容“转移”给 a1。

【例 3-21】 类 Book 具有书名（bookName）和书价（price）数据成员，为它设计移动赋值运算符函数和移动复制构造函数，采用对象移动方式处理临时对象复制，以提高效率。

```
// Eg3-21.cpp
#include<iostream>
#include<string>
using namespace std;

class Book {
public:
    Book(char* name = (char*)"", double x = 0):price(x) {           // 默认构造函数
        newbkName(name);
    }
```

```

        cout<<"constructor ..."<<endl;
    }
    Book(const Book& bk):price(bk.price) { // 复制构造函数
        newbkName(bk.bookName);
        cout<<"Copy constructor ..."<<endl;
    }
    ~Book() { delete []bookName; }
    Book(Book&& bk):bookName(bk.bookName) { // 移动复制构造函数
        price = bk.price;
        bk.bookName = nullptr;
        cout<<"Move constructor ..."<<endl;
    }
    Book& setData(char* name, double p) {
        newbkName(name);
        price = p;
        return *this;
    }
    Book &operator = (Book& bk) { // 赋值运算符函数
        if (this == &bk)
            return *this;
        delete []bookName;
        newbkName(bk.bookName);
        cout<<"operator = "<<endl;
        return *this;
    }
    Book &operator = (Book &&bk) { // 移动赋值运算符函数
        if (this == &bk)
            return *this;
        delete []bookName;
        bookName = bk.bookName;
        bk.bookName = nullptr;
        cout<<"move operator = (&&)"<<endl;
        return *this;
    }
    char* getName() { return bookName; }
    double getPrice() { return price; }
private:
    void newbkName(char *name) { // 类内私用函数
        bookName = new char[strlen(name) + 1];
        strcpy(bookName, name);
    }
    char* bookName;
    double price;
};

Book getBook(Book a) { // 普通函数，调用复制构造函数传递 a
    Book b = a; // 调用复制构造函数
    return a; // 返回右值，调用移动复制构造函数
}

void main() {
    Book b, book; // L1

```

```

book.setData((char*)"数据库原理", 32.4);
Book a = getBook(book); // L2
cout<<a.getName()<<"\t"<<a.getPrice()<<endl;
Book c = std::move(a); // L3
b = std::move(c); // L4
a = b; // L5
}

```

程序运行结果如下：

```

constructor ... // L1 语句输出，构造 b
constructor ... // L1 语句输出，构造 book
Copy constructor ... // L2 语句输出，调用复制构造函数传递参数
Copy constructor ... // L2 语句输出，函数局部对象 b 构造
Move constructor... // L2 调用移动构造函数把函数返回临时对象内存资源转换给 a
数据库原理 32.4
Move constructor ... // L3 语句，移动复制构造函数把 a 的资源转换给新建对象 c
move operator = (&&) // L4 语句输出，move()调用移动赋值把 c 的内存资源移动给 b
operator = // L5 语句输出，调用赋值运算符进行对象复制赋值

```

说明：

① 对象资源被移动后，它应该是可析构和有效的。

“可析构”是说对象资源被移给另一个对象后，马上销毁它也不会影响其他对象。比如，执行语句 L2 后，就立即析构 `getBook()` 函数创建的临时对象，不会对 `a` 产生影响。在为类设计移动函数时，必须保障对象移动后就进入可析构状态，即销毁该对象不会影响“窃取”其资源的另一对象，对于被移动的指针成员，可以设置它为 `nullptr`，如本例的移动复制构造函数和移动赋值运算符函数将 `bookName` 设置为 `nullptr`，否则会产生“指针悬挂”问题。

“有效的”是说该对象的内存资源被移动后，并不会立即销毁对象，它仍处于可用状态，可以被赋新值（但是，通过移动而“窃取”了该内存资源的对象可能会修改其中的值）。比如，上面的语句 L4 执行后，对象 `a` 的内存资源被转移给了 `c`，但 `a` 仍是有效的。

② 复制左值，移动右值。一个类同时设置了复制构造函数、赋值运算符函数、移动复制构造函数和移动赋值运算符函数时，编译器会使用与普通函数相同的参数匹配规则进行函数调用。对于左值使用复制构造函数或赋值运算符函数，如语句 L2 的参数名 `book`、语句 L5 的 `b` 只能是左值，所以调用复制构造函数和赋值运算符函数。而语句 L2 的函数 `getBook()` 返回的是对象右值，所以调用移动复制构造函数。

③ 若类没有移动函数，则右值会被复制。如果一个类定义了复制构造函数和赋值运算符函数，但没有移动构造函数和移动赋值运算符函数，编译器不会生成默认的移动构造函数和移动赋值运算符函数，就不能执行移动操作，即使调用函数 `move()` 也只能执行对象复制操作。

```

class A {
public:
    A() = default;
    A(A& o) :x(o.x) { cout<<"1"<<endl; }
    A& operator = (A &o) {
        x = o.x;
        cout<<"2"<<endl;
        return *this;
    }
};

```

```

    }
private:
    int x;
};
A a1, a2;
A a3(a1);
A a4 = std::move(a2);           // 调用复制构造函数，因无移动复制构造函数
a3 = std::move(a1);             // 调用赋值函数，因无移动赋值运算符函数

```

3.9 静态成员

1. 静态成员的声明及意义

在类中，若在数据成员或成员函数的声明或定义前面加上关键字 **static**，就将它定义成了静态数据成员或静态成员函数，统称为**静态成员**。静态成员同样遵守 **public**、**private**、**protected** 访问权限的限定规则。其定义形式如下：

```

class X {
    .....
    static type dataName;
    static type funName(...);
    .....
}

```

其中，**dataName** 是静态数据成员的名称，**funName()** 是静态成员函数的名称，**type** 代表数据类型。

静态数据成员是属于类的，整个类只有一份副本，相当于类的全局变量，供该类所有对象公用，能够被该类的所有对象访问；非静态数据成员是属于对象的，每个对象都有非静态数据成员的一份副本，为该对象专用。

静态成员函数也是属于整个类的，只能访问属于该类的静态成员（包括静态数据成员和静态成员函数），不能访问非静态成员（包括非静态的数据成员和成员函数）。

2. 静态成员的定义

在类的声明中，将数据成员指定为静态成员只是一种声明，并不会为该数据成员分配内存空间，在使用前应该对它进行定义。静态数据成员常常在类外进行定义，形式如下：

```

类型 类名::静态成员名;
类型 类名::静态成员名 = 初始值;

```

但是，对静态成员函数而言，除了在类声明中的成员函数前面加上关键字 **static**，其定义与普通函数没有区别。

注意：① 在类外定义数据成员时，不能加 **static** 限定词；② 在定义静态数据成员时可以指定它的初始值（第 2 种定义形式），若定义时没有指定初值，系统默认其初值为 0。

原则上，类的静态数据成员必须在类外定义，否则会出错。但在一些编译器中（如 VC 6.0），若没有在类外进行静态数据成员的定义，它会在定义该类的第一个对象时定义相关的静态数据成员（为所有的静态数据成员分配内存空间），并将这些静态数据成员初始化为 0。

3. 静态成员的访问

静态成员属于整个类，如果将它定义为类的公共成员，在类外可用下面两种方式访问。

① 通过类名访问（这种访问方式是非静态成员不具有的）：

类名::静态数据成员名；

类名::静态成员函数名(参数表)；

② 通过对象访问：

对象名.静态成员名；

对象名.静态成员函数名(参数表)；

【例 3-22】 设计一个书类，能够保存书名、定价以及所有书的册数和总价。

问题分析与数据抽象：用 Book 表示书类，每本书都有书名和定价，用数据成员 bkName 和 price 表示。但是，书的册数和总价则不是每本书都有的数据，整个书类用一个变量统计就可以了，静态数据成员是全类对象共用的数据成员，可以用静态成员 number、totalPrice 表示书的册数和总价。

为了访问数据成员，以数据成员为中心，分别为每个成员设置修改成员值的函数 setXX() 和读取成员值的函数 getXX()，以及显示书本信息和统计结果的函数 display()。

书的册数和总价的统计可以在构造函数和析构函数中进行，每定义一本新书就增加书的册数和总价，每析构一本书就减少册数和总价。另外，修改书价也会引起总价的变化。

Book 类图如图 3-13 所示，下画线标注的数据成员是静态成员，实现后的程序代码如下。

// Eg3-22.cpp

```
#include <iostream>
```

```
#include <string>
```

```
using namespace std;
```

```
class Book {
```

```
private:
```

```
    string bkName;
```

```
    double price;
```

```
    static int number;
```

```
    static double totalPrice;
```

```
public:
```

```
    Book() {
```

```
        bkName = "";
```

```
        price = 0;
```

```
        number++;
```

```
    };
```

```
    Book(string, double);
```

```
    ~Book();
```

```
    void setName(string bname) { bkName = bname; }
```

```
    void setPrice(double bprice) {
```

```
        totalPrice -= price;
```

```
        price = bprice;
```

```
        totalPrice += price;
```

```
    }
```

```
    double getPrice() { return price; }
```

```
    string getName() { return bkName; }
```

```
    static int getNumber() { return number; }
```

```
    static double getTotalPrice() { return totalPrice; }
```

```
    void display();
```

Book
- bkName : string
- <u>number : int</u>
- price : double
- <u>totalPrice : double</u>
+ Book()
+ Book(string, double)
+ display() : void
+ getName() : string
+ getPrice() : double
+ setName(string) : void
+ setPrice() : void

图 3-13 Book 类图


```

};
Book::Book(string name, double Price) { // 构造函数，可访问静态和非静态成员
    bkName = name;
    price = Price;
    number++;
    totalPrice += price;
}
Book::~Book() {
    number--; // 析构一本书就减少书的册数
    totalPrice -= price; // 析构一本书就减少书的总价
}
// 此函数仅是一个验证，表示非静态成员函数可以访问静态的数据和函数成员
void Book::display() {
    cout<<"book name: "<<bkName<<" "<<"price: "<<price<<endl;
    cout<<"number: "<<number<<" "<<"totalPrice: "<<totalPrice<<endl;
    cout<<"call static function"<<getNumber()<<endl;
}
int Book::number = 0; // 定义并初始化静态数据成员
double Book::totalPrice = 0;
void main() {
    Book b1("C++程序设计", 32.5), b2;
    b2.setName("数据库系统原理");
    b2.setPrice(23);
    cout<<b1.getName()<<"\t"<<b1.getPrice()<<endl; // L1
    cout<<b2.getName()<<"\t"<<b2.getPrice()<<endl; // L2
    cout<<"总共: "<<b1.getNumber()<<"\t 册书" // L3
        <<"\t 总价: "<<b1.getTotalPrice()<<"\t 元"<<endl;
    {
        Book b3("数据库系统原理", 23);
        cout<<"总共: "<<b1.getNumber()<<"\t 册书" // L4
            <<"\t 总价: "<<b1.getTotalPrice()<<"\t 元"<<endl;
    } // b3 析构
    cout<<"总共: "<<Book::getNumber()<<"\t 册书" // L5
        <<"\t 总价: "<<Book::getTotalPrice()<<"\t 元"<<endl;
    b2.display();
}

```

本程序的运行结果如下：

```

C++程序设计    32.5 // L1 的输出
数据库系统原理    23 // L2 的输出
总共: 2    册书    总价: 55.5    元 // L3 的输出
总共: 3    册书    总价: 78.5    元 // L4 的输出
总共: 2    册书    总价: 55.5    元 // L5 的输出
book name: 数据库系统原理 price: 23 // 之后是函数 display() 的输出
number: 2 totalPrice: 55.5
call static function 2

```

对比语句 L3 和 L4 的输出会发现，每增加一册书，书的册数和总价都能够正确增加；每减少一册书，书的册数和总价就会减少，这些都是静态成员完成的功能。如果没有静态数据成员，只有通过全局变量才实现这样的功能。但全局变量会破坏类的封装性，给程序维护带

来负担（程序中的其他函数可能误改全局变量）。

语句 L4 通过对象 b1 调用类 Book 的静态成员函数，但程序运行结果表明它输出的是已经定义了对象 b3 后的总价和总册数。事实上，若将语句 L4 中的 b1.getNumber() 改写成如下调用，也会得到完全相同的结果。

```
b2.getNumber()
或者 b3.getNumber()
或者 Book::getNumber()
```

因为静态成员函数属于整个类，不论通过哪个对象调用到的静态成员函数都是相同的，所以建议通过类调用静态成员函数（见语句 L5），以区别于普通成员函数的调用。成员函数 b2.display() 只是一种验证，表明非静态成员函数可以调用静态成员和普通成员（非静态）。

说明：

① 同普通成员函数一样，静态成员函数也可以在类内部或类外定义，还可以定义成内联函数。

② 静态函数只能访问静态成员（包括静态的数据成员和成员函数），不能访问非静态成员。例如，如下函数定义是错误的。

```
int Book::getNumber() {
    price = 20;           // 错误，getNumber() 是静态函数，不能访问非静态成员
    return number;
}
```

③ 在类外定义静态成员函数时，不能加上 static 限定词。如下函数定义是错误的。

```
static int Book::getNumber() { return number; }
```

④ 静态成员函数可以在定义类的任何对象之前被调用，非静态成员只有在定义对象后，通过对象才能访问。例如，对于类 Book，有如下语句。

```
int x = Book::getNumber();    // 正确
int y = Book::getPrice();     // 错误，getPrice() 不是静态成员函数
void main() { cout<<x<<endl; }
```

3.10 this 指针

1. this 指针的概念

类的每个对象都有自己的数据成员，有多少个对象，就有多少份数据成员的副本。然而类的成员函数只有一份副本，不论多少个对象，都公用这个成员函数。那么，不同对象是怎样公用这个成员函数的呢？换句话说，在程序运行过程中，成员函数怎样知道哪个对象在调用它，它应该处理哪个对象的数据成员呢？答案就是 this 指针。

this 是用于标识一个对象自引用的隐式指针，代表对象自身的地址，并且不允许修改，所以被指定为 const 指针，相当于如下定义：

```
class X{};
X *const this;
```

在编译类成员函数时，C++ 编译器会自动将 this 指针添加到成员函数的参数表中。在调用类的成员函数时，调用对象会把自己的地址通过 this 指针传递给成员函数。

【例 3-23】 一个处理坐标点的简单类 Point。

```
// Eg3-23.cpp
#include <iostream>
using namespace std;

class Point {
private:
    int x, y;
public:
    Point(int a = 0, int b = 0) { x = a; y = b; }
    void move(int a, int b) { x = a; y = b; }
    int getx() { return x; }
    int gety() { return y; }
};

void main(){
    Point p1, p2;
    p1.move(10, 20);
    p2.move(3, 4);
}
```

在编译类 Point 时，编译器会将 Point 类型的 this 指针参数添加到它的成员函数的参数表中。因为类 Point 的成员函数都是在类内定义的，所以它们还将被设置为内联函数。经过编译器的处理后，类 Point 的成员函数类似于如下形式：

```
inline point(Point *const this, int a, int b) { this->x = a; this->y = b; }
inline getx(Point *const this) { return this->x; }
inline gety(Point *const this) { return this->y; }
inline void move(Point *const this, int a, int b) { this->x = a; this->y = b; }
```

当 Point 类型的对象调用某个成员函数时，C++会把该对象的地址作为传递给 this 指针的实参，这样成员函数就知道调用它的对象是谁了。

例如，p1.move(10, 20)会被编译器转换成如下形式的函数调用：

```
move(&p1, 10, 20);
```

而 p2.move(3, 4)被编译器转换成如下形式的函数调用：

```
move(&p2, 3, 4);
```

因为 this 指针是在程序员不知晓的情况下，由编译器添加到成员函数参数表中的隐含参数，所以称它为隐式指针。

说明：

① 尽管 this 是一个隐式指针，但在类的成员函数中可以显式地使用它。比如，类 Point 也可以定义如下：

```
class Point {
private:
    int x, y;
public:
    Point(int x = 0, int y = 0) {
        this->x = x;
        this->y = y;
    }
}; // 用 this 区别数据成员与形参，典型用法
```

```

    }
    void move(int a, int b) { (*this).x = a;    (*this).y = b; }
    int getx() { return this->x; }
    int gety() { return this->y; }
};

```

this 是一个指针，必须按指针的用法引用它，如 “**this->x**” 或 “**(*this).x**”。

② 在类 **X** 的非 **const** 成员函数中，**this** 的类型就是 **X ***。**this** 并不是一个常规变量，不能给它赋值，但是可以通过它修改数据成员的值。在类 **X** 的 **const** 成员函数中，**this** 被设置成 **const X *** 类型，不能通过它修改对象的数据成员值。

③ 静态成员函数没有 **this** 指针，在静态成员函数中不能访问对象的非静态数据成员，因为非静态数据成员是通过 **this** 指针传递给成员函数的。没有 **this** 指针，就意味着不能将对象的地址传递给静态成员函数。这也是静态成员函数只能访问静态数据成员的原因（静态数据成员是类范围内的全局变量，本类的所有成员函数都可以访问）。

2. 通过 **this** 返回对象地址或自引用的成员函数

在类成员函数中，可以通过 **this** 指针返回对象的地址或引用，这也是 **this** 的常用方式。引用是一个地址，允许函数返回引用就意味着函数调用可以被再次赋值，即允许函数调用出现在赋值语句的左边。下面是一个具有返回本类对象的指针、引用及普通对象的类 **Tdate**，借此理解 **this** 指针的一些典型应用方法。

【例 3-24】对于日期类 **Tdate**，设计修改其年、月、日的成员函数，测试通过 **this** 指针返回对象的指针和引用的各种情况。

```

// Eg3-24.cpp
#include <iostream>
using namespace std;

class Tdate {
private:
    int yy, mm, dd;
public:
    Tdate(int y = 2006, int m = 01, int d = 01);
    Tdate &setYear(int year);
    Tdate &setMonth(int month);
    Tdate *setDay(int day);
    Tdate setDate(int y, int m, int d);
    void display();
};

Tdate::Tdate(int y, int m, int d) { yy = y;    mm = m;    dd = d; }
Tdate& Tdate::setYear(int year) { yy = year;    return *this; }
Tdate& Tdate::setMonth(int month) { mm = month;    return *this; }
Tdate* Tdate::setDay(int day) { dd = day;    return this; }
Tdate Tdate::setDate(int y, int m, int d) {
    yy = y;
    mm = m;
    dd = d;
    return *this;
}

```

```

void Tdate::display() {
    cout<<"address is: "<<this<<"\t"<<yy<<":"<<mm<<":"<<dd<<endl;
}

void main() {
    Tdate d1, d2; // L1
    cout<<"d1 "; d1.display(); // L2
    cout<<"d2 "; d2.display(); // L3
    d1.setYear(2007).setMonth(03).setDay(30); // L4
    cout<<"d1 "; d1.display(); // L5
    d1.setDate(2000,01,10).setDay(30); // L6
    cout<<"d1 "; d1.display(); // L7
    Tdate *p; // L8
    p = d1.setDay(21); // L9
    cout<<" p "; // L10
    p->display(); // L11
    Tdate d3 = d2.setYear(2006).setMonth(4); // L12
    cout<<"d3 "; d3.display(); // L13
    d1.setYear(2007).setMonth(03) = d3; // L14
    cout<<"d1 "; d1.display(); // L15
}

```

本程序的运行结果如下：

```

d1 address is: 002DFC60 2006:1:1
d2 address is: 002DFC4C 2006:1:1
d1 address is: 002DFC60 2007:3:30
d1 address is: 002DFC60 2000:1:10
p address is: 002DFC60 2000:1:21
d3 address is: 002DFC2C 2006:4:1
d1 address is: 002DFC60 2006:4:1

```

第 1、2 行是语句 L2、L3 的输出，对照函数 `Tdate::display()` 不难理解此结果。从中可以看出，d1、d2 的 `this` 指针的值分别是 002DFC60 和 002DFC4C，这就是 d1 和 d2 对象的地址。

语句 L4 看上去有些怪异，但结合函数定义也不难理解。由于成员函数 `setYear()` 返回的是对象的引用，因此 `d1.setYear(2007)` 的结果仍然是 d1，但 d1 的 `year` 已被设置成了 2007。同理可知，`d1.setYear(2007).setMonth(03)` 的结果仍然是 d1，但月份已被设置为 3。由于结果仍是 d1，也就不难理解 `d1.setYear(2007).setMonth(03).setDay(30)` 了，它等价于如下语句组：

```

d1.setYear(2007);
d1.setMonth(03);
d1.setDay(30);

```

这就不难理解语句 L5 的输出，即运行结果的第 3 行。

运行结果的第 4 行为什么不是 “d1 address is: 002DFC60 2000:1:30” 呢？它是程序中语句 L7 的输出。但在 L7 输出之前，L6 似乎已经将 d1 的日期设置为了 30。语句 L6 如下：

```

d1.setDate(2000,01,10).setDay(30);

```

原因是函数 `setDate()` 返回的是一个普通 `Tdate` 对象，不是指针，也不是引用。编译器对函数 `setDate()` 的处理方式类似于如下情况。

```

Tdate Tdate::setDate(int y, int m, int d) {

```

```

        yy = y;    mm = m;    dd = d;
        Tdate tmp = *this;
        return tmp;
    }

```

所以，函数 `setDate()` 返回的不是对象 `d1` 本身，而是一个临时对象 `tmp`。因此，语句 L6 中的 `setDay(30)` 实际等价于 `tmp.setDay(30)`，这就是 `d1` 的成员 `dd` 没有被修改的原因。

请读者根据成员函数的返回类型分析语句 L9、L12 和 L14 的赋值为什么是可行的，并进一步分析程序的运行结果。

3.11 对象应用

3.11.1 成员访问操作符

C++语言（包括 C 语言）提供了一些用于结构、类、联合等复合类型的对象成员的访问操作符，如表 3-3 所示。其中，`x` 是对象，`p` 是对象指针。在类对象的应用程序设计中，经常会用到这些访问操作符来访问对象成员，因此很有必要了解和掌握它们。

表 3-3 对象成员的访问操作符

访问操作符	.	->	*	.*	->*
作用	成员选择	指针成员选择	解引用	成员解引用	指针解引用成员选择
案例	<code>x.m</code>	<code>p->m</code>	<code>*x</code>	<code>x.*m</code>	<code>p->*m</code>

下面设计一个简单的类 `Person`，简要介绍表 3-3 中所列访问操作符在类对象访问中的应用方法。由于本例的主要目的是说明访问操作符应用，因此将类 `Person` 的所有成员都设计为 `public` 属性，以便简化问题。

【例 3-25】 设计具有姓名、编号、年龄的简单类 `Person`，能够输出和修改 `Person` 的编号和年龄。

```

class Person {
public:
    char* name = nullptr;
    int id;
    int age;
    void outData() {
        cout<<"id : "<<id<<"\tname : "<<name<<"\tage : "<<age<<endl;
    }
    int modifyId(int Id) {
        id = Id;
        return age;
    }
    int modifyAge(int Age) {
        age = Age;
        return age;
    }
};

```

1. 圆点 “.”、指针 “->” 成员选择符和解引用

圆点 “.” 和 “->” 成员访问运算符与 C 语言中 struct 成员的用法相同，其用法如下：

```
x.m;           // “.” 成员选择
p->m;          // “->” 成员选择
(*p).m        // 解引用成员选择
```

x 是要访问的对象，m 是该对象的 public 成员。访问过程是：根据 x 找到其所在的内存区域，然后访问该区域中的 m 成员。

p 是一个指向要访问对象的指针，m 是 p 所指对象的成员。访问过程是：<1> 先通指针 p 找到要访问对象的地址；<2> 根据 p 中找到的地址找到要访问对象的内存区域；<3> 访问找到对象区域中的成员 m。由此可见，“->” 实际上是一个间接访问成员的方法，因此也称为[间接成员引用](#)或“延迟成员选择操作符”。

*p 称为[解引用](#)，p 是指向要引用对象的指针，操作过程与 “p->m” 相同。在具有多级子结构的对象访问中，解引用可能需要通过多重括号确定对象，用 “->” 进行成员访问就会使程序代码更为简洁。

当结合应用 “.” 和 “*” 进行成员访问时，须知 “.” 的优先级高于解引用运算 “*”。如下语句示范了上面三种成员访问方法的用法。

```
Person p1;
Person *p2 = new Person();
p1.age = 21;
p2->age = 21;
(*p2).age = 21;
```

2. 成员解引用 “.*” 和间接成员解引用 “->*”

“*” “.” “->” 成员访问运算可以在对象或指针对象中直接使用，而成员解引用运算符 “.*” 和 “->*” 是指向类成员的指针，可以保存成员在类中的相对地址，通过 “.*” 和 “->*” 可以灵活访问类成员，编写出功能强大且代码精炼的程序。

“.*” 和 “->*” 并不是类的成员，而是在程序中可以用来指向类成员的指针变量。因此，同任何变量一样，需要定义后才能够使用。如下语句定义了指向 Person 类成员的指针：

```
int (Person::* P_int) = &Person::age;           // L1
char* (Person::* Pname) = &Person::name;        // L2
int (Person::* pf1)(int) = &Person::modifyAge;   // L3
void (Person::* pf2)() = &Person::outData;       // L4
P_int = &Person::id;                             // L5
pf1 = &Person::modifyAge;                         // L6
```

定义了 4 个指向 Person 类的成员指针。

语句 L1 定义了成员指针 P_int，并指向了成员 age。实际上，P_int 的类型是 int Person::*，能够指向 Person 类中所有 int 类型的数据成员。比如，语句 L5 就让 P_int 指针指向了类 Person 的数据成员 id。

语句 L2 定义的 Pname 的类型是 char* Person::*，能够操作类 Person 中所有 char* 类型的数据成员。

语句 L3 定义的 pf1 的类型是 int Person::*(int)，是一个函数成员指针，能够访问类 Person 中所有返回类型为 int 并且具有一个 int 类型参数的成员函数，如语句 L6。

语句 L4 定义了 `Person` 类的函数成员指针 `pf2`, 其类型为 `void Person::*()`, 能够调用类 `Person` 中所有类型为 `void` 的无参数成员函数。

由于解引用成员指针是针对类定义的, 因此能够与类 `Person` 的任何对象中的类型相符成员进行绑定, 并以解引用的方式访问绑定的数据成员或成员函数。例如:

```
Person p3;
p3.*Pname = new char(10);           // 用字符串解引用成员 Pname 访问 p3 的 name
strcpy(p3.*Pname, "Mike");           // 访问 p3 的 name
(p3.*pf2)();                          // 用函数解引用 pf2 访问 p3 的 outData() 函数
```

定义类 `Person` 的指针对象 `p4`, 通过解引用成员访问 `p4` 所指对象,

```
Person* p4 = new Person();
p4->*P_int = 10007;                   // 间接解引用成员访问整数指针 id
cout<<(p4->*pf1)(21);
```

上面分析的完整函数 `main()` 如下。注意, 程序中并没有定义 `p3`、`p4`, 而直接引用了 `p1`、`p2`, 由此掌握可用不同的成员解引用指针操作符多次访问同一对象的成员。

```
int main() {
    // 1: 圆点 “.” 成员选择符的应用方法
    Person p1;
    p1.name = new char(10);
    p1.id = 10001;
    p1.age = 10;
    strcpy(p1.name, "Tom");
    p1.outData();
    cout << "id: " << p1.id << "\tname: " << p1.name << "\tage: " << p1.age << endl;

    // 2: 指针 “->” 成员选择操作符的应用
    Person* p2;
    p2 = new Person();
    p2->age = 20;
    p2->id = 1002;
    p2->name = new char(10);
    strcpy(p2->name, "Jack");

    // 3: 解引用 “*” 操作符的应用
    (*p2).age = 21;
    cout<<"id: "<<(*p2).id<<"\tname: "<<(*p2).name<<"\tage: "<<(*p2).age<<endl;

    // 4: 成员解引用 “.*” 的应用
    int(Person:: * P_int) = &Person::age;           // L1
    char* (Person:: * Pname) = &Person::name;        // L2
    int(Person:: * pf1)(int) = &Person::modifyAge;    // L3
    void(Person:: * pf2)() = &Person::outData;        // L4

    p1.*P_int = 23;
    P_int = &Person::id;
    p1.*P_int= 10004;
    (p1.*pf2)();
    (p1.*pf1)(30);
    p1.outData();
}
```

```

// 5: 指针成员解引用 “->*” 的应用
p2->*P_int = 40;
P_int = &Person::id;
p2->*P_int = 10005;
(p2->*pf2)();
(p2->*pf1)(32);
p2->outData();
return 0;
}

```

在 VS 2022 中编译运行本程序，输出结果如下：

```

id: 10001      name: Tom      age: 10
id: 10001      name: Tom      age: 10
id: 10002      name: Jack     age: 21
id: 10004      name: Tom      age: 23
id: 10004      name: Tom      age: 30
id: 10005      name: Jack     age: 21
id: 10005      name: Jack     age: 32

```

3.11.2 对象数组与对象指针

类实际是一种自定义数据类型，既然是一种数据类型，就可以用来定义各种变量（即对象）。对象数组就是用类定义的数组，它的每个元素都是对象。对象指针是指用来指向类对象的指针。对象指针与结构指针的访问方法相同，用“->”或“(*指针).”两种操作符访问其所指对象的成员。

【例 3-26】 对象数组和对象指针的应用。

```

// Eg3-26.cpp
#include <iostream>
using namespace std;

class point {
private:
    int x = 0, y = 0; // L1
public:
    point() { x = 1; y = 1; } // L2
    point(int a, int b) { x = a; y = b; } // L3
    int getx() { return x; }
    int gety() { return y; }
};

void main() {
    point p1(3, 3); // 定义单个对象
    point p[3] = {{2,2}, {3,3}, {4,4}}; // L4
    point p2[3]; // L5
    point* pt; // L6
    for(int i = 0; i < 2; i++) {
        cout<<"p["<<i<<"] .x = "<<p[i].getx()<<"\t"; // 对象数组元素的访问
        cout<<"p["<<i<<"] .y = "<<p[i].gety()<<endl;
    }
}

```

```

    pt = &p1;                                // 指向单个对象的指针
    cout<<"Point pt->x: "<<pt->getx()<<endl;    // 指针对象访问方法 1
    pt = p2;                                // 指向对象数组的指针
    cout<<"Point Array pt->x: "<<pt->getx()<<endl;
    pt++;                                    // 指向对象数组下一元素
    cout<<"Point Array pt->x: "<<pt->getx()<<endl;
    cout<<"Point (*pt).x: "<<(*pt).getx()<<endl;    // 指针对象访问方法 2
}

```

对象数组 `p` 有 3 个元素，每个元素都是一个 `point` 对象。程序的运行结果如下：

```

p[0].x = 2    p[0].y = 2
p[1].x = 3    p[1].y = 3
Point pt->x: 3
Point Array pt->x: 1
Point Array pt->x: 1
Point (*pt).x: 1

```

说明：

- ① 如果在定义对象数组时没有进行元素的初始化，要求定义数组的类必须有默认构造函数。在本例中，如果没有定义语句 L2 处的默认构造函数，那么语句 L5 是错误的。当然，不定义语句 L2 的默认构造函数，但为语句 L3 处的构造函数参数 `a`、`b` 指定默认值也可以。
- ② 语句 L4 是用需要参数的构造函数创建数组时必需的初始化方式。

3.11.3 向函数传递对象

类类型也可以作为函数的参数类型，向函数传递对象。除了必须按照对象的访问控制权限访问类对象的成员，作为参数传递的类对象，类类型与普通变量的传递规则和方法是一致的，也可以分为三种参数传递方式：传递值，传引用，传指针。

传递值时，将通过复制构造函数以按位复制的方式，将实参对象的每个数据成员的值按位复制到形参对象的各数据成员中。参数传递完成后，形参与实参就没有关系了，所以按值传递对象的方式不能修改实参对象的值。引用和指针参数则不会调用任何构造函数，它们会将实参对象的地址传递给函数，能够在函数中修改实参对象的值。

【例 3-27】 按传递值、传引用、传指针的方式向函数传递参数对象。

```

// Eg3-27.cpp
#include <iostream>
using namespace std;

class MyClass {
    int val;
public:
    MyClass(int i) { val = i; }
    int getval() { return val; }
    void setval(int i) { val = i; }
};

void display(MyClass ob) { cout<<ob.getval()<<endl; }
void change1(MyClass ob) { ob.setval(50); }

```

```

void change2(MyClass &ob) { ob.setval(50); }
void change3(MyClass *ob) { ob->setval(100); }

void main() {
    MyClass a(10);
    cout<<"Value of a before calling change  -----";
    display(a);
    change1(a);
    cout<<"Value of a after calling change1()-----";
    display(a);
    change2(a);
    cout<<"Value of a after calling change2()-----";
    display(a);
    change3(&a);
    cout<<"Value of a after calling change3()-----";
    display(a);
}

```

本程序的运行结果如下：

```

Value of a before calling change  -----10
Value of a after calling change1()-----10
Value of a after calling change2()-----50
Value of a after calling change3()-----100

```

函数 `change1()` 按值方式传递对象，不能修改对象 `a` 的成员值；函数 `change2()` 和 `change3()` 分别按引用和指针方式传递对象，都修改了对象 `a` 的成员值。

说明：

① 函数接收参数对象后，在函数体内必须按照访问权限访问对象成员，即只能访问对象的公有成员。例如，对上述类 `MyClass` 而言，如下语句是错误的，它访问了对象 `ob` 的私有成员 `val`。

```
void change(MyClass ob) { ob.val = 90; };
```

② 类成员函数可以访问本类参数对象的私有、保护、公有成员，而普通函数（非类成员）只能访问参数对象的公有成员。例如：

```

class A {
    private:
        int x;
    public:
        void f(A b){ b.x = 10; }           // 正确，类的成员函数可以访问同类参数对象的私有成员
};
void g(A b){ b.x = 10; }                 // 错误，普通函数不能访问类的私有成员

```

3.11.4 对象成员

类的数据成员一般都是基本数据类型，也可以是结构、联合、枚举之类的自定义数据类型，还可以是其他类的对象。用其他类的对象作为类的成员被称为**对象成员**。

对象成员的定义形式如下：

```
class X {
```

```

        类名 1  成员名 1;
        .....
        类名 n  成员名 n;
    };

```

对象成员必须用类内初始值或构造函数初始化列表进行初始化。只有当对象成员有默认构造函数时才可以省略，此时编译器会自动调用对象成员的默认构造函数，其他情况必须使用类内初始化或者在构造函数初始化列表中显式初始化对象成员，否则产生错误。

【例 3-28】 类 **Sid** 完成学生学号的管理，类 **Student** 完成学生学号和姓名的管理。

问题分析与数据抽象：本例主要探讨对象成员的初始化和应用问题。类 **Sid** 只用于管理学号的输入和修改，数据成员 **id** 表示学号，**setSid()**和 **getSid()**修改和返回学号；类 **Student** 有学号和姓名，用数据成员 **m_sid** 和 **name** 表示。其中，**m_sid** 已由类 **Sid** 实现了，可以通过类成员引用其功能。因此，类 **Student** 只需实现对 **name** 的管理即可，但要考虑对象成员 **m_sid** 的初始化问题，必须在类 **Student** 构造函数初始化列表或类内初始值中对 **m_sid** 进行初始化。

Sid 和 **Student** 的类图如图 3-14 所示，菱形连接了 **Sid** 和 **Student**，表示两类之间是聚合关系，菱形所在指向整体，包含另一方的一个或多个对象。

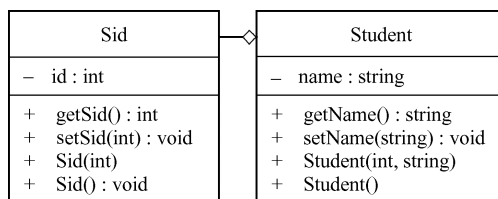


图 3-14 **Sid** 和 **Student** 的类图

实现类图关系的程序代码如下：

```

// Eg3-28.cpp
#include <iostream>
#include<string>
using namespace std;

class Sid {
public:
    ~Sid() { cout<<"Sid des ... "<<id<<endl; };
    Sid(int sid) :id(sid) { cout<<"Sid cons ... "<<id<<endl; }
    int getSid() { return id; }
    void setSid(int sid) { id = sid; }
private:
    int id;
};

class Student {
public:
    Sid m_sid; // L1
    // Sid m_sid = 9818; // L2, C++ 11
    // Sid m_sid = Sid(9818); // L3, C++ 11
    // Sid m_sid{ 9818 }; // L4, C++ 11
    ~Student() {
        cout<<"Stu des ... "<<name<<"\t"<<m_sid.getSid()<<endl;
    }
    Student(string sname, int stuid) : m_sid(stuid), name(sname) { // L5
        cout<<"Stu con ... "<<name<<"\t"<<m_sid.getSid()<<endl;
    }
    string getName() { return name; }
    void setName(string sname) { name = sname; }
}

```

```

private:
    string name;
};

void main() {
    Student s("Randy", 9818);
    s.setName("Tom"); // L6
    cout<<s.getName()<<"\t"<<s.m_sid.getSid()<<endl; // L7
}

```

本程序的运行结果如下：

```

Sid con ... 9818
Stu con ... Randy      9818
Tom 9818
Stu des ... Tom        9818
Sid des ... 9818

```

这个结果表明类 **Student** 和 **Sid** 的构造函数和析构函数都被调用了，并且通过 L6 语句把 **s** 对象的 **name** 修改成了 **Tom**。

说明：

① 两类之间聚合关系的实现。语句 L1 的“**Sid m_sid;**”定义了类 **Student** 的对象成员 **m_sid**，实现了类 **Sid** 和 **Student** 之间的聚合关系。许多 UML 工具生成的“**Sid *m_sid;**”用对象指针建立两个类之间的联系，也是实现对象之间聚合关系的常用技术。

② 对象成员的初始化。

一是通过构造函数初始化列表初始化对象成员。语句 L5 处的“**:m_sid(stuid)**”用于实现对象成员 **m_sid** 的初始化，调用类 **Sid** 的构造函数“**Sid::Sid(int sid)**”对 **m_sid** 的数据成员 **id** 进行初始化。

当一个类有对象成员且该成员没有进行类内初始化时，必须在其构造函数的初始化列表中，对对象成员进行初始化。只有当对象成员所在类有默认构造函数时，可以不在类的构造函数初始化列表中写出初始化对象成员的列表，但编译器会在此位置自动调用对象成员的默认构造函数。

如本例中，若将 L5 处的“**:m_sid(stuid)**”删掉，其余代码不做任何修改，程序在编译时将出现错误，原因是无法初始化类 **Student** 的对象成员 **id**。若为类 **Sid** 的构造函数指定默认值，类似“**Sid(int sid=0){...}**”，则删除 L5 处的“**:m_sid(stuid)**”就不会有问题。

二是通过类内初始值初始化对象成员。C++ 11 标准后，可以在声明类对象成员时就对它进行初始化，即为其指定类内初始值。语句 L2、L3、L4 是为对象成员 **m_sid** 指定类内初始值的不同方法，这三条语句形式不同，含义完全相同。

在本例中，删掉 L5 处的“**:m_sid(stuid)**”和 L2、L3 或 L4 某一处的注释，程序也不会有问题。

③ 对象成员的访问。对象成员同样遵守 **public**、**private**、**protected** 访问权限的约束限定。L7 中的“**s.m_sid.getSid()**”通过 **s** 的 **public** 对象成员 **m_sid** 访问了 **id**，这是访问对象成员的成员函数的典型语法。例如：

```

class Student {
private:
    Sid m_sid;

```

```

.....
}
Student s1("Tom", 1811);
s.m_sid.getSid();           // 错误, m_sid 是 Student 类的私有成员, 不能通过它访问任何数据

```

④ 对象成员初始化次序问题。对象成员的构造次序与它们在类中的声明次序相同，与它们在构造函数初始化列表中的次序无关。

【例 3-29】 对象成员的构造次序。

```

// Eg3-29.cpp
#include <iostream>
using namespace std;

class A {
    int a;
public:
    A(int i = 1) :a(i) { cout<<"constructing A: "<<a<<endl; }
};
class B {
    int b;
public:
    B(int i) :b(i) { cout<<"constructing B: "<<b<<endl; }
};
class C {
    A a1, a2;
    B b1, b2;
public:
    C(int i2, int i3, int i4) : b1(i3), b2(i4), a2(i2) {}
};

void main() {
    C x(2, 3, 4);
}

```

本程序的输出结果如下，请读者分析其原因。

```

constructing A: 1
constructing A: 2
constructing B: 3
constructing B: 4

```

3.12 类的作用域和对象的生命期

1. 类的作用域

类构成了一种特殊的作用域，称为**类域**。类域是指类定义时的一对“{ }”括起来的范围，其形式如下：

```

class X {                               // 类域开始
    .....
};                                       // 类域结束

```

类域范围内的成员可以互相访问，不受成员访问控制权限及定义的先后次序的影响，这

与类外的函数只能访问类的公共成员是不同的。例如：

```
class X { // X的类域开始了，最外层{ }所框定的范围就是X的类域
    int a, b;
    float c;
public:
    int f1(int i) {
        int a, y;
        a = i;
        X::a = 9; // 同一类域中的函数和数据可以相互访问
        return f3(a); // 同一作用域内，成员函数调用不受先后次序影响
    }
    void f2(int j) {
        // y = 1; // 错误，y未定义，y只在f1内有效
        b = f1(j);
        a = j+b;
    }
    int f3(int n) {
        return n*n;
    }
}; // X的类域结束了，在后面就只能访问X的公有成员了
X n, *p;
n.f1(2); // 正确，在类域外访问类的公有成员
n.a = 2; // 错误，在类域外不能访问类的私有成员
p->f1(3);
```

在成员函数内部定义的变量，其作用域限于定义它的成员函数。如果类的数据成员与某个成员函数内部定义的变量同名，可用“类名::数据成员名”的方式访问数据成员。例如，成员函数 f1() 中的 “X::a = 9” 就是对类 X 的数据成员 a 的访问。

说明：

① 同一类中的成员拥有相同的作用域，它们可以相互访问而不受访问权限的限定，而且不受成员声明先后次序的影响。如在上面的类 X 中，f1() 访问了定义于其后的成员函数 f3()，f2() 直接访问了私有数据成员 a、b 和公有成员函数 f1()。

② 在类域之外只能通过类的对象访问类的公有成员，而且必须用对象名和成员限定符“.”进行限定；如果通过对象指针访问类成员，就必须使用对象指针名和“->”加以限定。

2. 对象的生命期

对象的生命期是指对象从它被创建开始到被销毁前在内存中存在的时间，分为静态生命期和动态生命期。**静态生命期**是指对象具有与程序运行期相同的生命期，这类对象一旦被建立后，它将一直存在，直到程序运行结束时才被销毁。全局对象和静态对象具有静态生命期。

动态生命期是指局部对象的生命期，局部对象具有块作用域，它的生命期是从它的定义位置开始，遇到离它最近的“}”就结束了。

【例 3-30】 对象的生命期分析。

```
// Eg3-30.cpp
#include <iostream>
using namespace std;

class X {
```

```

public:
    X(int ii = 1) { i = ii;    cout<<"X ("<<ii<<") created"<<endl; }
    ~X() { cout<<"X ("<<i<<") destroyed"<<endl; }
private:
    int i;
};
class Z {
public:
    Z():x3(3), x2(2) { cout<<"Z created"<<endl; }
    ~Z() { cout<<"Z destroyed"<<endl; };
private:
    X x1, x2, x3;
};

X a(200);                                // a 的生命期开始了

void main (void) {
    Z z;                                // z 的生命期开始了，且其成员对象 x1、x2、x3 的生命期也开始了，且先于 z
    {
        X c(100);                        // c 的生命期开始了
        static X b(50);                  // b 的生命期开始了
    }                                    // c 的生命期结束了
}                                       // z、x3、x2、x1、b 的生命期依次结束
                                       // 函数 main() 结束后，a 的生命期才结束

```

说明：生命期与对象的构造次序和销毁次序密切相关。

① 局部对象和静态对象的构造次序与它们在块中的声明次序相同，即在块中先声明的就先构造，块即对象定义所在的“{ }”限定的代码区域。全局对象在函数 `main()` 之前构造，在函数 `main()` 结束之后销毁。

② 对象数据成员（包括对象成员）的构造次序与其在类中的声明次序相同，而与它们在构造函数的初始化列表中的次序无关。如在上面的类 `Z` 中，尽管 `x3` 在类 `Z` 的构造函数的初始化列中先于 `x2` 和 `x1`（`x1` 用默认值构造，未在初始化列表中），但在建立类 `Z` 的对象时，C++ 仍会按照声明次序依次建立 `x1`、`x2`、`x3` 对象。

③ 在对象生命期结束时，具有相同生命期的对象将按照与其构造的相反次序销毁。

④ 非静态对象的生命期与其作用域是一致的，而静态对象的生命期长于其作用域，程序结束时静态对象的生命期才结束。

例 3-30 的运行结果如下，请读者结合上面的说明，分析此结果的产生过程。

```

X (200) created
X (1) created
X (2) created
X (3) created
Z created
X (100) created
X (50) created
X (100) destroyed
Z destroyed
X (3) destroyed
X (2) destroyed

```

```

X (1) destroyed
X (50) destroyed
X (200) destroyed           // 在某些环境中运行时（如 VC 6.0），没有这行输出，试析其原因

```

3.13 友元

类的封装性具有信息隐藏的能力，使外部函数只能通过类的公有成员函数才能访问类的私有成员。如果要多次访问类的私有成员，就要多次访问类的公有成员函数，需要进行频繁的参数传递、参数类型检查、函数调用等操作，不但操作麻烦，而且会占用较多的存储空间和时间，降低程序的运行效率。

能否给某些函数特权，让它们可以直接访问类的私有成员呢？C++给出的答案是友元（friend）。友元机制允许一个类授权其他函数直接访问类的 `private` 和 `protected` 成员。

友元包括友元函数、友元类和友元成员函数。最常用的是友元函数，定义形式如下：

```

class X {
    .....
    friend T f(X ...);           // 声明 f 为 X 类的友元
    .....
};
.....
T f(X...) { ... }               // 友元不是类成员函数，定义时不能用 “X::f” 限定函数名

```

其中，`T` 代表函数返回类型，友元不是类的成员函数，在定义时不能把“类名::”放在它的函数名前面。在上述形式的友元函数中，常常需要把类 `X` 作为它的参数类型，这样才能更好地体现友元的意义。

【例 3-31】 `Point` 是处理屏幕坐标点的类，为它设计计算两点之间距离的友元函数。

```

// Eg3-31.cpp
#include <iostream>
#include <cmath>
using namespace std;

class Point{
private:
    int x, y;
    friend int dist1(Point p1, Point p2);           // 声明 dist1 为 point 类的友元
public:
    Point(int a = 10, int b = 10) { x = a; y = b; }
    int getx() { return x; }
    int gety() { return y; }
};

int dist1(Point p1, Point p2) {
    double x = (p2.x-p1.x);
    double y = (p2.y-p1.y);
    return sqrt(x*x + y*y);
}

int dist2(Point p1, Point p2){
    // dist2()是普通函数

```

```

double x = p2.getx()-p1.getx();           // 普通函数只能访问对象的公有成员
double y = p2.gety()-p1.gety();
return sqrt(x*x + y*y);
}

void main(){
    Point p1(2, 5), p2(4, 20);
    cout<<dist1(p1, p2)<<endl;
    cout<<dist2(p1, p2)<<endl;
}

```

函数 `dist1()` 和 `dist2()` 都能计算出两点之间的距离。但 `dist1()` 是类 `Point` 的友元，可以直接访问参数对象的私有数据成员；`dist2()` 不是类 `Point` 的友元，只能通过参数对象的公共成员函数访问其私有数据成员。

友元使编程更简洁，程序运行效率也更高，但可以直接访问类的私有成员，破坏了类的封装性和信息隐藏。

说明：

① 在类域中的函数原型前加上关键字 `friend`，就将该函数指定为类的友元了。类的友元函数是一种特殊的普通函数，可以直接访问该类的私有成员。关键字 `friend` 用于声明友元，它只能出现在类的声明中。

② 友元函数并非类的成员函数，所以它不受 `public`、`protected`、`private` 的限定，无论将它放在哪个区域，都是完全相同的。

③ 友元不具逆向性和传递性。即，若 `A` 是 `B` 的友元，并不表示 `B` 是 `A` 的友元（除非特别声明）；若 `A` 是 `B` 的友元，`B` 是 `C` 的友元，也不能代表 `A` 是 `C` 的友元（除非特别声明）。

一个类还可以是另一个类的友元，称为友元类。友元类的所有成员函数都是另一个类的友元函数，能够直接访问另一个类的所有成员（包括 `public`、`private` 和 `protected` 属性的）。友元类的定义形式如下：

```

class A {
    .....
    friend class B;           // 声明类 B 是类 A 的友元类
};
class B {
    .....
};

```

类 `B` 是类 `A` 的友元类，它的任何成员函数都能直接访问类 `A` 的私有成员。

注意：友元类不是双向的，类 `B` 是类 `A` 的友元并不意味着类 `A` 是类 `B` 的友元，如果想让类 `A` 成为类 `B` 的友元，必须在类 `B` 中加上 `friend class A` 的声明。

3.14 编程实作：类的接口与实现的分离

在实际的软件开发过程中，一个应用程序的代码量常常很大，可能由多个程序员分别编写不同的程序模块。将一个完整应用程序的所有程序代码放在一个文件中既不现实也不方便。实际情况是一个应用程序可能由多个文件组成。每个文件强调其逻辑结构，完成一定的功能，可以由不同的程序员编写，并且能够被编译器分别编译，最后通过一定的方式组装成一个应

用程序。

类也常按这样的方式组织，分为接口和实现两部分。接口是指类的声明，实现是指类的成员函数的定义。在 C++ 程序中，常把接口放在一个与类同名的头文件中（扩展名为 .h 的文件），而把类的实现放在一个与类同名的源程序中（扩展名为 .cpp 的文件）。

【例 3-32】 建立一个整数堆栈类 **Stack**，栈的默认大小为 10 元素，能够完成数据的入栈和出栈处理。将类的声明（接口）存放在单独的头文件 **stack.h** 中。

(1) 问题分析

堆栈是计算机领域中广泛应用的一种数据存储技术，是一种按顺序存取的数据结构，类似生活中按层次存放衣服的箱子，后放入的衣服压在上次入箱的衣服上面，称为入栈（**push**）；取出衣服时每次都只能取最上层的衣服，称为出栈（**pop**）。最先放入的衣服在箱子底层，最后才能取出，因此堆栈是先进后出（**First-In/Last-Out, FILO**）的数据结构，如图 3-15 所示。

(2) 数据抽象

可以用数组、链表之类的数据存取技术实现堆栈，通过限定只能在数组或链表的一端进行数据读写，就能够实现。本例将堆栈抽象成类 **Stack**，用数组 **data** 保存堆栈的数据，为了实现只在数组一端进行读写数据的操作，设置 **top** 指针指示栈顶元素，每次只能够读出它指向的元素，每读出一个数据，**top** 就向下移动一个元素位置；同样，每次保存数据，只能保存在 **top** 指向的位置，每存入一个数据，**top** 就向上移动一个位置，再设置 **maxSize** 表示数组的最大下标，代表堆栈容量，如图 3-16 所示。

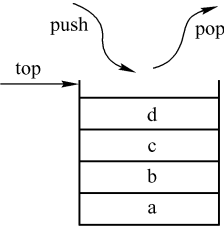


图 3-15 堆栈示意

Stack	
-	data : int*
-	maxSize : int
-	top : int
<hr/>	
+	howMany() : int
+	pop : int
+	push(int) : void
+	Stack(int)

图 3-16 堆栈类

3.14.1 头文件

头文件常作为软件的接口，以源码的方式提供给用户，用户可用 **#include** 宏把头文件包含到自己的程序中。头文件中的信息对用户是可见的，常包含以下内容：常量定义、类型定义、变量声明、枚举、宏定义、条件编译指令、函数声明、内置函数定义、模板声明或模板定义，以及注释等内容。

对类而言，类的声明常放在一个与类同名的头文件中，最终以源码形式提供给类的用户。用户通过头文件就能了解类的全部成员（包括数据成员和成员函数），而且可以将类的头文件包含到应用程序中，定义类对象，并根据头文件提供的信息，向类的成员函数传递参数，使用类的功能。因此，将堆栈类的声明单独保存在名为“**stack.h**”的头文件中，代码如下。

```
// 堆栈 stack 的头文件：stack.h
#ifndef Stack_h
#define Stack_h
```

```

class Stack {
private:
    int *data;           // 存放栈数据
    int top;             // 存放栈顶指针
    int maxSize;         // 栈的容量
public:
    Stack(int stacksize = 10); // 构造函数建立具有 10 元素的默认栈
    ~Stack();
    void push(int x);         // 元素入栈
    int pop();                // 元素出栈
    int howMany();            // 判定栈中有多少个元素
};
#endif

```

说明：

① 如下语句是为 `stack.h` 增加的条件编译

```

#ifndef Stack_h
#define Stack_h
.....
#endif

```

检查程序中是否有 `stack_h` 标识符，若没有，则声明 `Stack` 类，否则不再对 `Stack` 类进行声明。

② 同一头文件可能被多个不同的文件多次采用 `#include` 包含到文件中，如果这些文件最终被放进了同一个应用程序，就相当于在同一程序中多次引入了相同头文件中的内容，会产生重复定义的错误。因此，常在头文件中加上条件编译，这样当同一程序多次引入相同头文件时，只有第一次才会把头文件包含到程序中。

3.14.2 源文件

函数或类的声明常被放在头文件中，它们的实现代码则常被存放在源文件中，称为接口与实现的分离。这样做的好处是，可以把头文件以源代码的方式提供给用户，而源文件以编译后的目标文件的方式（如 C++ 的各种库文件）提供给用户，能够达到信息和技术保密的目的，也为多个程序员同时进行软件开发提供了技术支持。在类设计时，常把类成员函数的实现代码放在与类同名的 `CPP` 源文件中。

本例中，将堆栈成员函数的实现代码放入 `stack.cpp` 源文件，如下所示：

```

// 堆栈 stack 的源文件: stack.cpp
#include "stack.h"           // 包含头文件
#include <iostream>           // push 和 pop 都用到了 cout, 所以包含此头文件
using namespace std;

Stack::Stack(int stacksize) {
    if (stacksize > 0) {
        maxSize = stacksize;
        data = new int[stacksize];
        for (int i = 0; i < maxSize; i++)
            data[i] = 0;
    }
    else {

```

```

        data = 0;
        maxSize = 0;
    }
    top = 0;
}
Stack::~Stack() {
    delete[] data;
}
void Stack::push(int x) {
    if (top<maxSize) {
        data[top] = x;
        top++;
    }
    else{
        cout<<"堆栈已满，不能再压入数据："<<x<<endl;
    }
}
int Stack::pop() {
    if(top <= 0) {
        cout<<"堆栈已空！"<<endl;
        exit(1);                // 堆栈操作失败，退出程序！
    }
    top--;
    return data[top];
}
int Stack::howMany() {
    return top;
}

```

3.14.3 对类的应用

把类的声明与实现分别保存在头文件和源文件中,可以通过**#include** 将头文件包含到要使用它们的程序中。但只包含类的头文件是不够的,还需要指明源文件所在的位置。

1. 直接引用类源文件

现在建立 **Stack** 类的应用程序。按如下步骤建立引用 **Stack** 类的测试项目：**stackuse**。在 VS 2022 中,选择“文件 | 新建 | 项目 | 空项目 | C++”菜单命令;在弹出的对话框的“位置”中指定保存项目的文件夹,在“名称”中输入项目名称 **stackuse**,然后单击“确定”按钮,编译器会创建项目 **stackuse**;添加主文件 **stackuse.cpp**,从中输入如下代码。

```

// 应用栈类的主程序: stackuse.cpp
#include "stack.h"
#include <iostream>
using namespace std;
void main() {
    Stack s1;
    s1.push(1);
    s1.push(12);
}

```



```

s1.push(32);
int x1 = s1.pop();
int x2 = s1.pop();
int x3 = s1.pop();
cout<<x1<<"\t"<<x2<<"\t"<<x3<<endl;
}

```

然后，把 `stack.h` 和 `stack.cpp` 复制到 `stackuse.cpp` 所在的目录中。

编译该项目，会出现多个类似于如下链接错误：

LNK2019 无法解析的外部符号 "public: void __thiscall Stack::push(int)"

此错误是指找不到 `Stack::push` 函数的实现代码。因为 `stackuse.cpp` 程序只包含 `stack.h`，并没有告诉 C++ 编译器在哪里可以找出 `stack.h` 中声明的成员函数 `Stack::push` 的实现代码。可以用以下方法解决此问题。

方法 1：在应用程序中添加 “`#include stack.cpp`”，即把 `stackuse.cpp` 中的 `#include "stack.h"` 替换为 `#include "stack.cpp"`，即：

```

#include "stack.cpp"
#include <iostream>
using namespace std;
void main() {...}

```

方法 2：在应用程序中添加 “`#include "stack.h"`”，然后把 `stack.cpp` 添加到应用程序的工程项目中。在本例中，保持 `stackuse.cpp` 的原样，即：

```

#include"stack.h"
#include<iostream>
using namespace std;
void main() {...}

```

将 `stack.cpp` 源文件添加到应用程序的工程项目的步骤为：选择“项目 | 添加现有项”菜单命令，弹出选择文件的对话框，找到 `stack.cpp` 源文件，然后添加到当前应用程序的工作项目中。编译并运行程序，可得到如下输出结果：

```

32  12  1

```

2. 引用类的静态库

方法 1 和方法 2 把类的实现代码暴露给用户了，为了向用户隐藏实现代码，可以把类的实现文件编译成静态链接库（扩展名为 `.lib` 的文件），静态链接库由目标代码组成（二进制代码文件），然后把它与类的头文件一起提供给用户。

1) 静态库的制作

现以制作 `stack.cpp` 的静态链接库 `stack.lib` 为例，介绍静态链接库的制作方法。

<1> 在 VS 2022 中，选择“文件 | 新建 | 项目 | 静态库 | C++”菜单命令，如图 3-17 所示，在弹出的对话框的“位置”中指定保存项目的文件夹，在“名称”中输入项目名称：`stack`。然后单击“确定”按钮。

上述操作会建立 `stack` 项目，并在项目中添加头文件 `pch.h`、`framework.h` 和源文件 `pch.cpp`、`stack.cpp`，可以将它们全部删掉，并进行如下第<3>步操作。

说明：若不删除这些文件，即不必执行第<5>步操作，但需要在 `stack.cpp` 修改时，保留其中对 `pch.h`，`framework.h` 头文件的包含。



图 3-17 建立静态库的项目设置

<2> 把前面设计的 `stack.h` 和 `stack.cpp` 复制到本项目建立的文件夹“`stack\stack`”中。

<3> 按照方法 2，把 `stack.h` 和 `stack.cpp` 添加到此工程中。本项目只有类 `Stack` 的代码，没有主函数 `main()`。

<4> 在 VS 2022 中选择“项目 | 属性”菜单命令，弹出“`stack` 属性页”对话框，在“配置属性”列表中单击“C/C++ | 预编译头”，然后在右边“预编译头”的下拉列表中选择“不使用预编译头”，并删除“预编译头文件”中的 `pch.h`，如图 3-18 所示。单击“确定”按钮。

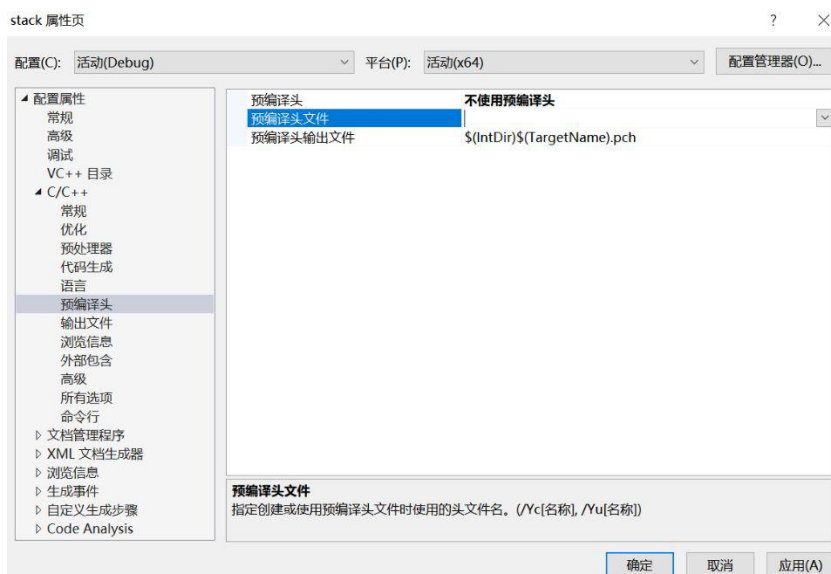


图 3-18 取消静态链接库的预编译头文件

<5> 编译此工程。编译成功后，会在目录 `C:\stack\x64\debug` 中（该目录由编译器建立）生成 `stack.lib` 静态库文件。

2) 静态库的应用

建立了 `stack.lib` 静态库后，就可以在程序中应用它了。不同 C++ 编译环境引用静态库的

方法大同小异。在 VS 2022 中，可以采用多种方法引用静态库中的函数。其中较便捷的一种方法是把头文件和静态库复制到要引用它的项目的文件夹中，然后直接把它们添加到该项目中。下面以在新建项目 `stackUse` 应用 `stack.lib` 为例，说明静态库的应用方法。

<1> 在 VS 2022 中，选择“文件 | 新建 | 空项目 | C++”菜单命令，在弹出的对话框的“位置”中指定保存项目的文件夹，在“名称”中输入项目名称：`stackUse`。

<2> 把 `stack.h` 和 `stack.lib` 复制到 `stackUse` 项目的源代码文件夹 `xx\stackUse\stackUse` 中。“xx”是<1>中指定的保存项目的文件夹。

<3> 选择“项目 | 添加现有项”菜单命令，把 `stack.h` 添加到 `stackUse` 项目中。

<4> 选择“项目 | 添加新项 | C++ 文件 (*.cpp)”菜单命令，输入 `stackUse.cpp`，创建并打开 `stackUse.cpp`，输入如下代码。

```
#include "stack.h"
#include <iostream>
#pragma comment(lib, "stack.lib")           // 加载静态库到项目中
using namespace std;

void main() {
    Stack s1;
    s1.push(1);
    s1.push(12);
    s1.push(32);
    int x1 = s1.pop();
    int x2 = s1.pop();
    int x3 = s1.pop();
    cout<<x1<<"\t"<<x2<<"\t"<<x3<<endl;
}
```

编译并运行程序，程序的输出为：

```
32    12    1
```

如果是在 VC 6.0 中引用自定义静态库，就需要设置 Visual C++ 的集成环境，把 `stack.lib` 在磁盘上的位置告知编译器。

<1> 选择“新建”，在弹出的对话框中选择“Win32 Static Library”，如图 3-19 所示。在该项目中添加 `stack.h` 和 `stack.cpp` 文件，编译后即可生成 `stack.lib` 静态库。

<2> 启动 VC 6.0，在其中建立一个如下简单应用程序，并把 `stack.h` 复制到如下应用程序所在的磁盘目录中。

```
#include "stack.h"
#include <iostream>
using namespace std;

void main() {
    Stack s;
    s.push(10);
    cout<<s.pop()<<endl;
}
```

编译此程序，将产生多个链接错误。

<3> 选择“工程 | 设置”菜单命令，弹出如图 3-20 所示的对话框，选择其中的“Link”标签。

<4> 在“L 对象/库模块”编辑框的最后输入静态库文件 `stack.lib` 所在的目录“`C:\lib\stack\debug\stack.lib`”。

<5> 编译程序，这次不会有错误了。运行此程序，将在屏幕上输出 10。

在实际编程中，常把 `stack.h` 和 `stack.lib` 复制到与主程序相同的目录中。

不管用哪种方式，在 VC 6.0 中需通过图 3-20 所示的对话框指明静态库所在的磁盘目录。



图 3-19 新建“lib”工程

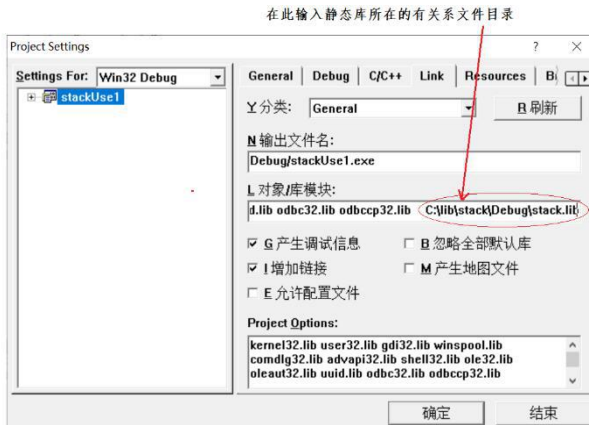


图 3-20 编译环境设置

静态库真正实现了类的接口与实现的分离。类的设计者将类的头文件和实现代码的静态库提供给用户，用户只能通过头文件了解类的接口和类的各成员函数的功能，但无法知道各成员函数的实现代码，也就无法修改类成员的实现代码，这就使类的封装和信息隐藏更彻底。

习 题 3

3.1 结构与类有什么区别？

3.2 什么是构造函数和析构函数？其作用是什么？有哪些类型的构造函数，分别会在什么时候被调用？

3.3 类对象的访问权限有哪几种？各有何特点？如果将构造函数的访问权限指定为 `private`，会出现什么情况？

3.4 举例说明什么是构造函数的初始化列表和类内初始值，它有什么作用。

3.5 什么是 `this` 指针？它有什么作用？

3.6 什么是友元？它有什么作用？

3.7 什么是类的静态成员？它与普通成员有什么区别？

3.8 分析下面代码中的错误。

```
class X {
private:
    int a = 0, &b;
    const int c;
    void setA(int i) { a = i; }
```

```

    X(int i) { a = i; }
public:
    int X() { a = b = c = 0; }
    X(int i, int j, int k) { a = i;    b = j;    c = k; }
    static void setB(int k) { b = k; }
    setC(int k) const { c = c+k; }
};

void main() {
    X x1;
    X x2(3);
    X x3(1, 2, 3);
    x1.setA(3);
}

```

3.9 读程序，写出代码运行结果。

(1)

```

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

class X {
    int a;
    char *b;
    float c;
public:
    X(int x1, char *x2, float x3):a(x1), c(x3) {
        b = new char[sizeof(x2)+1];
        strcpy(b, x2);
    }
    X():a(0), b("X::X()"), c(10){ }
    X(int x1, char *x2 = "X::X(...)", int x3 = 10):a(x1), b(x2), c(x3) { }
    X(const X&other) {
        a = other.a;
        b = "X::X(const X &other)";
        c = other.c;
    }
    void print() { cout<<"a = "<<a<<"\t"<<"b = "<<b<<"\t"<<"c = "<<c<<endl; }
};

void main() {
    X *A = new X(4, "X::X(int, char, float)", 32);
    X B, C(10), D(B);
    A->print();
    B.print();
    C.print();
    D.print();
}

```

(2)

```

#include <iostream>

```

```

using namespace std;

class Implementation {
public:
    Implementation(int v) { value = v; }
    void setValue(int v) { value = v; }
    int getValue() const { return value; }
private:
    int value;
};

class Interface {
public:
    Interface(int);
    void setValue(int);
    int getValue() const;
private:
    Implementation *ptr;
};

Interface::Interface(int v):ptr(new Implementation(v)) { }
void Interface::setValue(int v) { ptr->setValue(v); }
int Interface::getValue() const { return ptr->getValue(); }

void main() {
    Interface i(5);
    cout<<i.getValue()<<endl;
    i.setValue(10);
    cout<<i.getValue()<<endl;
}

```

(3)

```

#include <iostream>
using namespace std;

class A {
    int x;
public:
    A():x(0) { cout<<"constructor A() called ..."<<endl; }
    A(int i):x(i) { cout<<"X"<<x<<"\tconstructor ..."<<endl; }
    ~A() { cout<<"X"<<x<<"\tdestructor ..."<<endl; }
};

class B {
    int y;
    A x1, x2[3];
public:
    B(int j):x1(j), y(j) { cout<<"B"<<j<<"\tconstructor ..."<<endl; }
    ~B() { cout<<"B"<<y<<"\tdestructor ..."<<endl; }
};

void main() {
    A x1(1), x2(2);
}

```

```
    B B1(3);  
}
```

(4)

```
#include <iostream>  
#include <assert.h>  
using namespace std;  
  
class Ctor {  
public:  
    Ctor(char* str = nullptr);  
    Ctor(Ctor&& t);  
    Ctor& operator = (Ctor&& t);  
    Ctor(Ctor& t);  
    Ctor& operator = (Ctor& t);  
    ~Ctor();  
private:  
    char *p = nullptr;  
};  
  
Ctor::Ctor(char* str) {  
    if(str) {  
        this->p = new char[strlen(str) + 1];  
        strcpy(this->p, str);  
    }  
    cout<<"1:Ctor(Char *)"<<endl;  
}  
  
Ctor::Ctor(Ctor&& t):p(move(t.p)) {  
    t.p = nullptr;  
    cout<<"2:Ctor(Ctor&&)"<<endl;  
}  
  
Ctor& Ctor::operator = (Ctor&& t) {  
    this->p = move(t.p);  
    t.p = nullptr;  
    cout<<"3:=(Ctor&&)"<<endl;  
    return *this;  
}  
  
Ctor::Ctor( Ctor& t) {  
    this->p = new char[strlen(t.p) + 1];  
    strcpy(this->p, t.p);  
    cout<<"4:Ctor(Ctor&)"<<endl;  
}  
  
Ctor& Ctor::operator = (Ctor &t) {  
    if(this != &t) {  
        delete[] this->p;  
        if(t.p) {  
            this->p = new char[strlen(t.p) + 1];  
            strcpy(this->p, t.p);  
        }  
    }  
}
```



```

    }
}
cout<<"5:=(Ctor &)"<<endl;
return *this;
}

Ctor::~Ctor() {
    if(this->p) {
        delete[] this->p;
        this->p = nullptr;
    }
    cout<<"~Ctor"<<endl;
}

void main() {
    Ctor c1("OK!"), c2("Hello");
    Ctor c3(c1);
    c3 = c2;
    c3 = move(c2);
    Ctor c4(move(c1));
}

```

3.10 某单位的职工收入包括基本工资 Wage、岗位津贴 Subsidy、房租 Rent、水费 WaterFee、电费 ElecFee。设计实现工资管理的类 Salary，其形式如下：

```

class Salary {
private:
    double Wage, Subsidy, Rent, WaterFee, ElecFee;
public:
    Salary() { 初始化工资数据的各分项 };
    Salary() { 初始化工资的各分项数据为 0 };
    void setXX(double f) { xx = f; };
    double getXX() { return xx; };
    double RealSalary(); // 计算实发工资
    .....
};

```

其中，成员函数 setXX()用于设置工资的各分项数据，成员函数 getXX()用于获取工资的各分项数据，XX 代表 Wage、Subsidy 等数据成员，如 Wage 对应的成员函数为 setWage()和 getWage()。

实发工资 = Wage + Subsidy - Rent - WaterFee - ElecFee

编写程序，完善该类的设计，建立具有 5 个工人的数组，写出测试该类各成员函数的主函数 main()，并用 STL 的函数 sort()对工人数组按照实发工资降序输出。

3.11 设计工人类 Worker，具有姓名 name、年龄 age、工作部门 dept、工资 salary 等数据成员。其中，salary 即第 10 题中设计的 Salary 类型的数据。按照第 10 题的形式完成类 Worker 的程序设计，并统计工人的人数（用静态成员统计人数）。

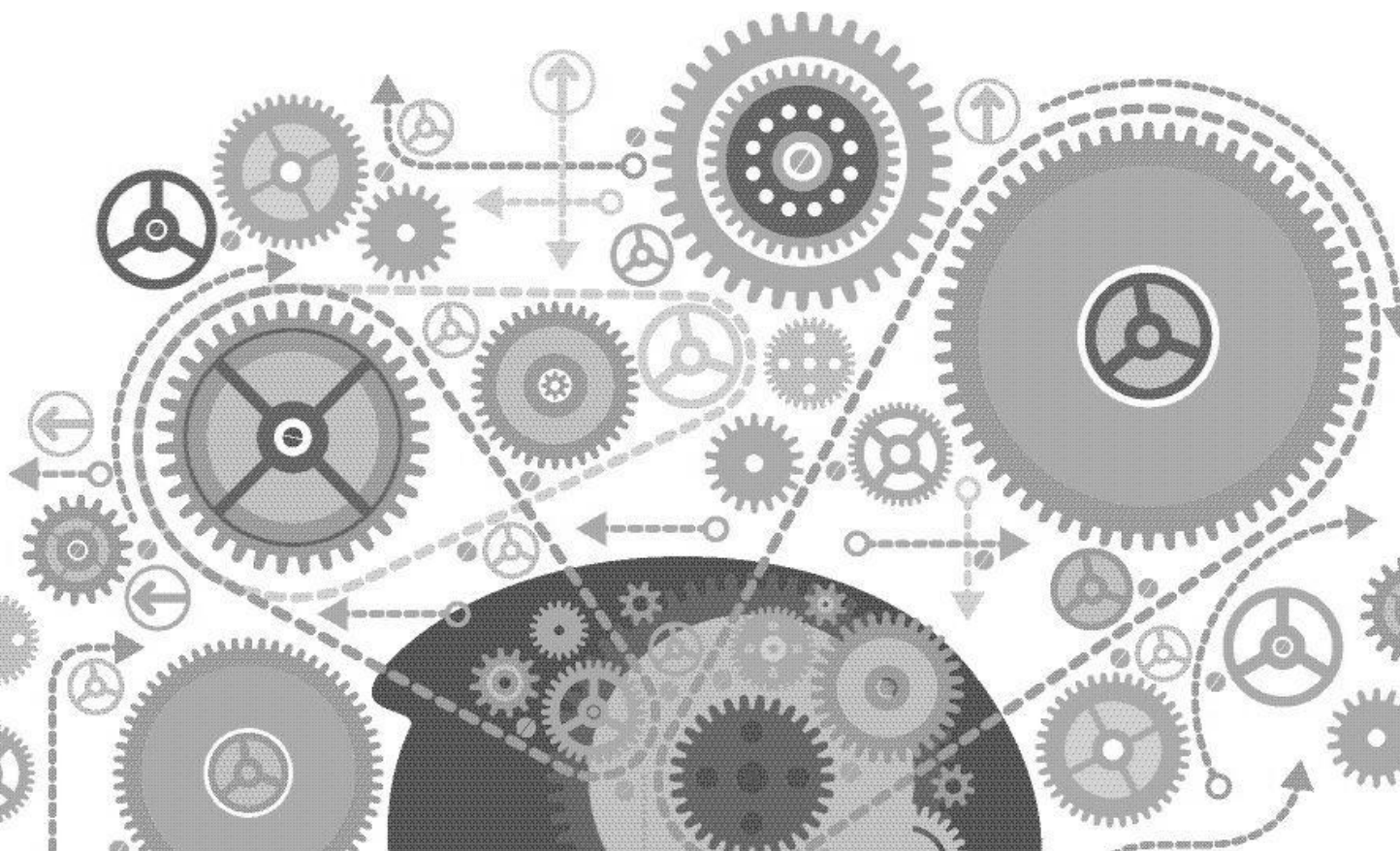
3.12 设计一个整型链表类 List，能够实现链表节点的插入（insert）、删除（delete），以及链表数据的输出操作（print）。

第 4 章

继 承

继承是代码重用的基本技术，是面向对象程序设计的重要特征之一。继承克服了面向过程程序设计语言没有软件复用机制的缺点，使软件复用变得简单、易行，可以复用已有的程序资源，缩短软件开发的周期。

本章介绍 C++ 继承的基本知识，包括：继承的方式、类型、派生类对基类成员的重载、覆盖和访问，派生类和基类构造函数的关系，以及多继承的二义性和虚拟继承等内容。



4.1 继承的概念

继承源于生物界，是指后代能够传承前代的特征和行为。面向对象程序设计语言提供了与此概念相近的语言处理机制，可以基于已有的类创建新类，使新类自然获得已有类的全部功能。已有类被称为**基类**，新类被称为**派生类**。在一些面向对象程序设计语言（如 Java）中，派生类也被称为**子类**，基类则被称为**父类**或**超类**。

事实上，派生类复制了基类的全体数据成员和成员函数，具有基类全体成员的一份复制品。派生类不仅能够继承基类的功能，还能够对其进行扩充、修改或重定义。

只能从一个基类派生的继承称为单继承，可以从多个基类派生的继承称为多继承。在单继承方式下，派生类只有一个基类；而在多继承方式下，派生类可以有多个基类。许多面向对象程序设计语言只支持单继承，而 C++ 语言既支持单继承，又支持多继承。

同一个类可以作为多个类的基类，一个派生类也可以是另一个类的基类，通过这种方式，可以形成同类事物的一种继承层次结构。例如，大学中有教师和学生，学生可以分为研究生和本科生，教师又可以分为任课教师和教辅人员。将教师 and 学生的共有特征和行为抽象出来，形成基类——“人”，将研究生和学生的共有特征和行为抽象为“学生”类，将任课教师和教辅人员的共有特征和行为抽象为“教师”类，就形成了图 4-1 所示的继承层次结构的 UML 图。

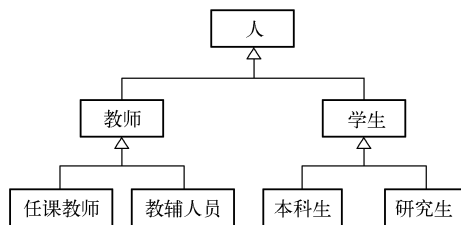


图 4-1 大学人员的继承层次

在 UML 图中，用空心三角形箭头表示继承关系，箭头端是基类，另一端是派生类。在图 4-1 中，人是教师和学生的基类，教师和学生则是人的派生类。派生类也可以是其他类的基类，如学生是本科生和研究生的基类，而本科生和研究生是学生的派生类。最顶层的人也是第三层的任课教师等类的基类，继承图中离派生类最近的基类称为直接基类，较远的基类称为间接基类。例如，人就是教师的直接基类，是任课教师的间接基类。

在图 4-1 中，可以把各类人员公有的数据成员和成员函数放在最上层的基类人中，其他各类人员则从基类继承这些数据成员和成员函数。例如，把每类人都有的姓名、身份证号、性别、身高等属性，以及获取姓名、修改身份证号、修改身高等行为，分别设计为基类的数据成员和成员函数。教师和学生类只需定义各自特有的数据成员（如教师编号、职称、学号、学习专业等）和成员函数（如修改教师编号、获取学生的专业等成员函数），至于姓名、身份证之类的数据成员，以及修改姓名、获取身份证号等成员函数，则从基类人中继承。

继承使得一个类可以复用其他类的程序代码，提高了软件复用的效率，缩短了软件开发的周期。概括而言，继承具有以下优点：

- ❖ 继承是一种在普通类的基础上构造、建立和扩展新类的最有效手段。
- ❖ 继承是自动传播代码的有力工具。

- ❖ 继承能够降低代码和数据的冗余度，增强程序的可重用性。
- ❖ 继承能够清晰地体现相似类之间的层次结构关系。
- ❖ 继承能够通过增强一致性来减少模块间的接口和界面，提高程序的易维护性。

4.2 protected与继承

`protected` 可以用来设置类成员的访问权限，具有 `protected` 访问权限的成员称为保护成员。`protected` 主要用于继承，对于一个不被任何派生类继承的类而言，`protected` 访问属性与 `private` 完全相同。然而在继承结构中，基类的 `protected` 成员虽然不能被派生类的外部函数访问，却能够被派生类的内部成员直接访问。

【例 4-1】 类 B 有数据成员 `i`、`j`、`k`，希望 `j` 可以被派生类和自身访问，但不希望除此之外的其他函数访问。`protected` 权限正好具有这样的访问控制能力。

```
// Eg4-1.cpp
#include <iostream>
using namespace std;

class B {
private:
    int i;
protected:
    int j;
public:
    int k;
};
class D: public B {
public:
    void f() {
        i = 1;
        j = 2;
        k = 3;
    }
};

void main() {
    B b;
    b.i = 1; // L5, 错误
    b.j = 2; // L6, 错误
    b.k = 3;
}
```

// L1, 表示 D 从 B 派生

// L2, 错误

// L3, 正确

// L4, 正确

派生类 D 继承了类 B，L1 位置的 `public` 表示公有继承，其意义是：类 D 中从基类 B 继承而来的成员 `i`、`j`、`k` 保持了与类 B 中相同的访问权限，如图 4-2 所示。

在派生类中可以直接访问基类的 `public` 和 `protected` 成员，但不能直接访问基类的 `private` 成员。因此，D 的成员函数 `f()` 可以直接访问基类 B 的 `public`

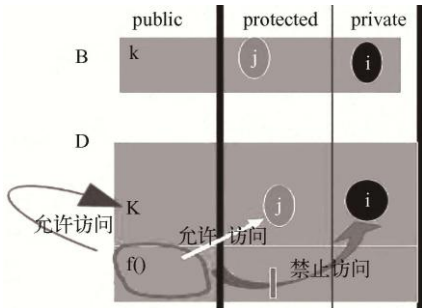


图 4-2 基类成员在派生类中的访问属性

成员 `k` 和 `protected` 成员 `j`，但不能直接访问 `B` 的 `private` 成员 `i`，这就是语句 `L2`、`L3`、`L4` 正确或错误的原因。

对于类的外部函数，`protected` 和 `private` 成员都是不可访问的，语句 `L5` 访问类 `B` 的 `private` 成员，语句 `L6` 访问类 `B` 的 `protected` 成员，都是错误的。

说明：

① 一个类如果不被其他类继承，则其 `protected` 和 `private` 成员具有相同的访问属性，它们都只能被本类成员访问，不能被类的外部函数访问。

② 一个类如果被其他类继承，派生类不能直接访问它的 `private` 成员，但能够直接访问它的 `protected` 成员，这就是 `protected` 成员与 `private` 成员的区别。

③ 尽管基类的 `public` 和 `protected` 成员都能被派生类直接访问，但是 `public` 成员能够被类的外部函数直接访问，`protected` 成员则不能。

4.3 继承方式

C++的继承分为公有继承、保护继承和私有继承，也称为公有派生、保护派生和私有派生。不同继承方式会不同程度地改变基类成员在派生类中的访问权限。

1. C++继承的形式

在 C++中，继承的语法形式如下：

```
class 派生类名:[继承方式] 基类名 {  
    派生类成员声明或定义;  
};
```

```
struct 派生类名:[继承方式] 基类名 {  
    派生类成员声明或定义;  
};
```

其中，继承方式可以是 `public`、`protected`、`private`，分别对应公有继承、保护继承、私有继承。类和结构具有相同的功能，区别是，如果省略继承方式，C++默认类为 `private` 继承，结构为 `public` 继承。例如：

```
class B {...}           // 用 struct 或 class 定义的类都可以作为基类  
struct D1:B {...}       // D1 public 继承于 B  
class D2:B {...}        // D2 private 继承于 B
```

派生类成员的定义与普通类的成员定义方式相同，但可以访问基类的 `public` 和 `protected` 成员。派生类通过继承获得了基类全体数据成员和成员函数的一份副本，不需要编程就能够拥有与其基类相同的功能。

2. 公有继承

继承方式为 `public` 的继承称为**公有继承**，在这种继承方式下，基类成员的访问权限在派生类中保持不变。

【例 4-2】 公有继承的例子。

```
// Eg4-2.cpp  
#include <iostream>  
using namespace std;  
class Base {
```



```

    int x;
public:
    void setx(int n) { x = n; }
    int getx() { return x; }
    void showx() { cout<<x<<endl; }
};
class Derived:public Base {           // L1
    int y;
public:
    void sety(int n) { y = n; }
    void sety() { y = getx(); }       // L2
    void showy() { cout<<y<<endl; }
};

void main() {
    Derived obj;                      // L3
    obj.setx(10);                     // L4, 从 base 继承
    obj.showx();                      // L5, 从 base 继承
    obj.sety(20);                     // L6
    obj.showy();                      // L7
    obj.sety();                       // L8
    obj.showx();                      // L9, 从 base 继承
    obj.showy();                      // L10
}

```

程序运行结果如下：

```

10
20
10
10

```

Base 类的 public 成员函数 setx()、getx()、showx() 形成了它的接口，外部函数可以通过这些接口函数访问 Base 类的功能，并通过这些接口函数访问其私有成员 x，如图 4-3 的上半部分所示。

类 Derived 从类 Base 公有派生，拥有与类 Base 相同的一份数据成员和成员函数，而且在类 Derived 中的这些成员具有与它们在类 Base 中相同的访问权限。在图 4-3

中，类 Derived 成员的 setx()、getx()、showx() 和 x 就是从基类 Base 继承而来的。由于 setx()、getx()、showx() 是基类 Base 的公有成员，在公有继承方式下，它们在派生类 Derived 中也保持了与在基类 Base 中相同的访问权限，即这些成员函数在类 Derived 中也是公有成员。

Derived 具有 x、y 两个私有数据成员，y 是它自己定义的，x 则继承于 Base。它们虽然同为类 Derived 的私有成员，都不能被 Derived 的外部函数直接访问，但有区别：在 Derived 中定义的成员函数 sety()、showy() 不能直接访问 x，只能通过 Base 的公有成员函数 setx()、getx() 和 showx() 访问 x。语句 L2 中的 getx() 访问基类 Base 的 x 就是一个例子，其定义为

```
void Derived::sety() { y = getx(); }
```

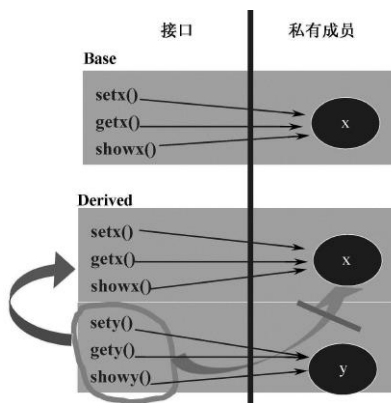


图 4-3 public 继承 Base 和 Derived 类的成员示意

但不能定义成

```
void Derived::setx() { y = x; }
```

函数 `main()` 定义了一个派生类对象 `obj`，并调用了其成员函数 `setx()`、`showx()`、`sety()` 和 `showy()`，尽管 `Derived` 没有定义成员函数 `setx()`、`showx()`，但从基类 `Base` 继承了它们，因此也是可用的。

说明：

① 公有继承不改变基类成员在派生类中的访问权限。在公有继承方式下，基类中的 `public` 成员、`private` 成员、`protected` 成员在派生类中保持与其在基类中相同的访问权限。

② 派生类自己定义的成员函数不能直接访问基类的私有成员，只能通过基类的 `public` 成员或 `protected` 成员访问它们。

3. 私有继承

继承方式为 `private` 的继承称为**私有继承**。在私有继承方式下，基类的 `private` 成员在派生类中仍是 `private` 成员，但基类的 `public` 和 `protected` 成员在派生类中会变成 `private` 成员。

在例 4-2 中，若将语句 L1 处的 `public` 改为 `private`，则 `Derived` 就从 `Base` 私有派生，即

```
class Derived:private Base {  
    ...  
}
```

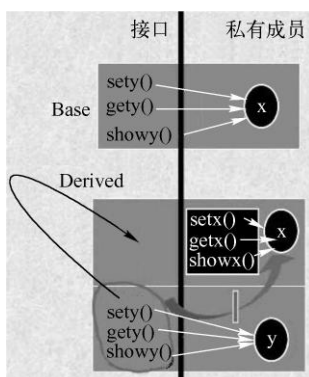


图 4-4 `private` 继承 `Base` 和 `Derived` 的成员示意

在其余代码不做任何修改的情况下，`Base` 和 `Derived` 类的成员结构如图 4-4 所示。可以看出，基类 `Base` 中的公有成员函数 `setx()`、`getx()` 和 `showx()` 在派生类 `Derived` 中都被改变成了 `private` 成员，这一改变是由 `private` 继承方式引起的。成员函数 `setx()`、`getx()` 和 `showx()` 不再是类 `Derived` 的公有接口，`Derived` 的外部函数不能够直接访问它们。因此，在 `private` 继承方式下，例 4-2 的 `main()` 中的语句 L4、L5 和 L9 就是错误的。

说明：

① 在私有继承方式下，基类的公有成员和保护成员在派生类中都变成了 `private` 成员，不再是派生类的公有接口函数，不能被派生类的外部函数访问。

② 在私有继承方式下，虽然基类的 `public` 和 `protected` 成员在派生类中都变成了 `private` 成员，但它们仍然有区别。派生类的成员函数不能直接访问基类的 `private` 成员，但可以直接访问基类的 `public` 和 `protected` 成员，并且通过它们访问基类本身的 `private` 成员。

例如，在私有继承方式下，`Derived` 中的函数 `sety()` 要访问类 `Base` 的 `x` 成员，只能通过类 `Base` 的成员函数 `getx()` 进行，不能直接访问，如下所示：

```
void Derived::sety() { y = getx(); } // 正确  
void Derived::sety() { y = x; } // 错误
```

对比图 4-3 和图 4-4 可以发现，在公有继承方式下，`Derived` 的外部函数能够直接访问从 `Base` 继承来的成员函数 `setx()`、`getx()` 和 `showx()`；但在私有继承方式下，`Derived` 的外部函数则不能访问这些成员函数。

4. 保护继承

继承方式为 `protected` 的继承称为保护继承。在保护继承方式下，基类的 `public` 成员在派生类中的访问权限将被修改为 `protected` 权限，基类的 `protected` 成员在派生类中仍为 `protected` 成员，基类的 `private` 成员在派生类中仍为 `private` 成员。

表 4-1 是在各种继承方式下，基类成员在派生类中的权限情况的汇总表。

表 4-1 基类成员在派生类中的访问权限

基类	派生类								
	public 继承			protected 继承			private 继承		
	public	protected	private	public	protected	private	public	protected	private
public	√				√				√
protected		√			√				√
private			√			√			√

5. 阻止继承 C++11

如果不想让一个类作为其他类的基类，就可以用 `final` 关键字阻止它被继承。

```
class Base{...} // 可以被继承
class NoDeri final {...} // 不能被继承
class D final:Base {...} // 正确，D 不能被继承
class D1:NoDeri {...} // 错误，NoDeri 不能被继承
class D2:D {...} // 错误，D 不能被继承
```

4.4 派生类对基类的扩展

通过继承，派生类复制了基类数据成员和成员函数的一份副本，不用编程就具备了基类的程序功能。在此基础上，派生类可以再定义新成员或对基类继承来的成员函数进行重定义，实现需要的新功能。它们之间的关系可以概括如下：派生类可以增加新的数据成员和成员函数，重载从基类继承到的成员函数，覆盖（重定义）从基类继承到的成员函数，改变基类成员在派生类中的访问属性。

但是，派生类不能继承基类的以下内容：析构函数、基类的友元函数、静态数据成员和静态成员函数。在 C++ 11 之前，派生类不能继承基类的构造函数。自 C++ 11 标准起，基类的构造函数也可以被继承。

4.4.1 成员函数的重定义和名字隐藏

基类的数据成员和成员函数在派生类中都有一份复制品，派生类能直接访问从基类继承而来的 `public` 和 `protected` 成员，且只能通过这两类成员访问从基类继承而来的 `private` 成员。

派生类不仅可以添加基类没有的新成员，还可以对从基类继承得到的成员函数进行覆盖或重载。**覆盖**也称为重定义，是指派生类可以定义与基类具有相同函数原型的成员函数（具有相同的返回类型、函数名及参数表）；而**重载**要求成员函数具有不同的函数原型。

注意：派生类对继承得到的基类成员函数的重定义或重载会影响它们在派生类中的可见性，基类的同名成员函数会被派生类重载的同名函数所隐藏，称为**名字隐藏**。

【例 4-3】 设计计算矩形与立方体面积和体积的类。

(1) 问题分析

矩形具有长和宽，面积=长×宽，没有体积，可以设置为 0。具有高的矩形就是立方体，面积=2×底面积+2×侧面积 1+2×侧面积 2，体积=底面积×高。其中的底面积就是矩形的面积。立方体是对矩形的扩展，矩形完成了长和宽的处理，在此基础上完成高的处理就能够实现其功能。这一关系可以通过继承实现。

(2) 数据抽象

将矩形抽象成类 **Rectangle**，用数据成员 **width** 表示宽，**length** 表示长，为了方便立方体派生类访问数据成员，将它们设置为 **protected** 访问权限；成员函数 **setWidth()**、**setLength()**、**getWidth()**、**getLength()** 用于设置和获取矩形的宽和长，**area()** 和 **volume()** 用于计算矩形的面积和体积，**outData()** 用于输出矩形的长和宽。

将立方体抽象成类 **Cube**，并从类 **Rectangle** 派生，类 **Rectangle** 已经完成了矩形长和宽的处理功能，所以只需增加数据成员 **height** 表示高，成员函数 **setHeight()** 和 **getHeight()** 完成对高的读、写功能。

虽然 **Rectangle** 类已经设置了成员函数 **area()**、**volume()** 和 **outData()** 分别计算面积和体积、输出数据成员的值，但立方体的面积、体积计算方法是不同的，需要重新定义它们。数据抽象结果的类图及之间的继承关系如图 4-5 所示。

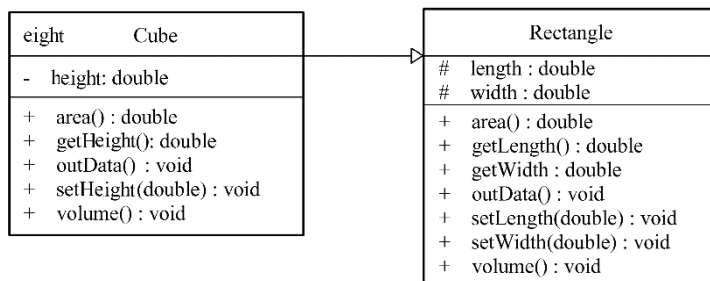


图 4-5 矩形和立方体的类图和继承关系

实现该继承关系的代码如下：

```
// Eg4-3.cpp
#include <iostream>
using namespace std;

class Rectangle {
public:
    void setLength(double h) { length = h; }
    void setWidth(double w) { width = w; }
    double getLength() { return length; }
    double getWidth() { return width; }
    double area() { return length*width; }
    double volume() { return 0; }
    void outData(){ cout<<"length = "<<length<<"\t"<<"width = "<<width<<endl; }
protected:
    double width;
    double length;
};

class Cube : public Rectangle {
protected:
    double height;
public:
    void setHeight(double h) { height = h; }
    double getHeight() { return height; }
    double area() { return 2*length*width+2*length*height+2*width*height; }
    double volume() { return length*width*height; }
    void outData(){ cout<<"length = "<<length<<"\t"<<"width = "<<width<<"\n"<<"height = "<<height<<endl; }
};
```

```

};
class Cube :public Rectangle{
public:
    void setHeight(double h) { height = h; }
    double getHeight() { return height; }
    double area() { return width*length*2+width*height*2+length*height*2; } // L1
    double volume() { return Rectangle::area()*height; }
    void outData() {
        Rectangle::outData(); // L2
        cout<<"height = "<<height<<endl;
    }
private:
    double height;
};

void main() {
    Cube cub1;
    cub1.setLength(4); // L3
    cub1.setWidth(5); // L4
    cub1.setHeight(3); // L5
    cub1.Rectangle::outData(); // L6
    cub1.outData(); // L7
    cout<<"立方体面积 = "<<cub1.area()<<endl; // L8
    cout<<"立方体底面积 = "<<cub1.Rectangle::area()<<endl; // L9
    cout<<"立方体体积 = "<<cub1.volume()<<endl; // L10
}

```

程序运行结果如下：

```

length = 4    width = 5
length = 4    width = 5
height = 3
立方体面积 = 94
立方体底面积 = 20
立方体体积 = 60

```

派生类 **Cube** 通过继承获得了基类 **Rectangle** 全体成员的一份复制品，具有矩形长、宽和面积的处理能力。但从 **Rectangle** 继承来的成员函数 **outData()**、**area()**、**volume()** 不能完成立方体的数据输出、表面积和体积计算的功能，因而被进行了重定义。也就是说，类 **Cube** 中的成员函数 **outData()**、**area()** 和 **volume()** 都有两个版本，一个是从 **Rectangle** 继承得到的，另一个是它自己定义的，并且重定义的函数名会隐藏它从基类继承到的成员函数名。

4.4.2 基类成员访问

在派生类中，可以直接访问通过继承得到的基类的 **public** 和 **protected** 成员，就好像这些成员是它自己定义的一样。派生类对基类成员的访问大致有以下 3 种方式。

① 通过派生类对象直接访问基类成员。在 **public** 继承方式下，基类的 **public** 成员在派生类中也是 **public**，可以被派生类对象的外部函数直接访问。在例 4-3 中，语句 L3 就属于这种访问方式。

② 在派生类成员函数中直接访问基类成员。在 `public`、`protected`、`private` 三种继承方式下，基类的 `public` 成员和 `protected` 成员可以被派生类的成员函数直接访问。在例 4-3 中，语句 L1 直接访问了基类的 `protected` 成员 `width` 和 `length`。

③ 通过基类名称访问被派生类重定义所隐藏的成员。`Cube` 类的成员函数 `outData()`、`area()`、`volume()` 都有两个版本，而且是可用的。派生类对象 `cub1` 直接调用到的是派生类重定义的成员函数，如例 4-3 中的语句 L7、L8 和 L10；如果要调用从基类继承到的同名成员函数，就需用“`基类::函数名(…)`”形式指明是基类的成员函数，如语句 L2、L6 和 L9。

4.4.3 using 和隐藏函数重现 C++11

如果基类的某个成员函数具有多个重载的函数版本，而派生类需要覆盖（重定义）其中某个重载函数，定义自己的新功能，就会隐藏基类同名的全部重载函数。在这种情况下，派生类有两种方式来引用被隐藏的基类成员函数：一是使用基类名称限定要访问的成员函数，如例 4-3 的语句 L6 和 L9；二是重载基类的所有同名函数，其代码与基类完全相同。

这两种方法都很烦琐，C++ 11 标准允许用 `using` 声明使基类重载函数在派生类中可见。在派生类中用 `using` 声明基类的函数名，不需提供函数参数。一条 `using` 语句就可以让所有从基类继承来的同名函数在派生类中可见，这些函数的访问权限与 `using` 语句所在区域的访问权限相同。

【例 4-4】 基类 B 的成员函数 `f1()` 具有 3 个重载函数，派生类 D 新增了函数 `f1()` 的功能，此函数会隐藏基类 B 的 `f1()` 在派生类的可见性，用 `using` 将基类的 `f1()` 引入派生类作用域。

```
// Eg4-4.cpp
#include <iostream>
using namespace std;

class B {
public:
    void f1(int a) { cout<<a<<endl; }
    void f1(int a,int b) { cout<<a + b<<endl; }
    void f1() { cout<<"B::f1"<<endl; }
};

class D : public B {
public:
    using B::f1; // L1, 使基类的 3 个 f1() 函数在此区域可见
    void f1(char * d) { cout << d << endl; }
};

void main() {
    D d;
    d.f1(); // L2, 正确
    d.f1(3); // L3, 正确
    d.f1(3, 5); // L4, 正确
    d.f1((char*)"Hello c++!");
}
```

如果派生类 D 中没有 L1 语句位置的 `using` 声明，基类 B 的 3 个 `f1()` 成员函数在派生类中会被类 D 定义的 `f1()` 隐藏，那么语句 L2、L3、L4 对 `f1()` 函数的调用就会出错。

4.4.4 派生类修改基类成员的访问权限

基类的 `protected` 和 `public` 成员是允许派生成员直接访问的，但在不同继承方式下，可以改变它们在派生类中的访问权限。比如，在 `private` 继承方式下，基类的 `public` 和 `protected` 成员在派生类中的访问权限都会被更改为 `private` 访问权限。

然而某些时候，从类的整体设计上考虑，需要改变个别基类成员在派生类中的访问权限，这可以通过在派生类中使用 `using` 声明实现该目的。

在派生类的 `public`、`protected` 或 `private` 权限区域内，用 `using` 再次声明基类的非 `private` 成员，就可重新设置它们在派生类中的权限为 `using` 语句所在区域的权限，即 `using` 语句在 `public` 区域内为 `public` 权限，在 `protected` 内为 `protected` 权限，在 `private` 内为 `private` 权限。

【例 4-5】 类 D 私有继承了类 Base，修改基类 Base 成员在派生类中的访问权限，设置基类成员 y 在派生类中的权限为 `private`，其余成员在派生类中保持与其在基类中相同的权限。

```
// Eg4-5.cpp
#include <iostream>
using namespace std;

class Base {
public:
    int x = 0;
    void setxyz(int a, double b, float c) {
        x = a;
        y = b;
        z = c;
    }
protected:
    double y = 0;
    float getZ() { return z; }
private:
    float z = 0;
};

class D :private Base {
protected:
    using Base::getZ;           // 指定 getZ() 为 protected 权限
    // using Base::z;          // 错误，不允许修改基类的 private 成员
public:
    using Base::x;              // 指定 x 为 public 权限
    using Base::setxyz;         // 指定 setxyz() 为 public 权限
    void display() {
        cout<<"x = "<<x<<"\ty = "<<y<<"\tz = "<<getZ()<<endl;
    }
private:
    using Base::y;              // 指定 y 为 private 权限
};

void main() {
    D d;
    d.setxyz(8, 9, 10);
    d.display();
}
```

```
}
```

类 D 私有继承类 Base，因此 D 从 Base 继承来的全体成员都是 **private** 权限，但派生类 D 使用 **using** 声明改变了基类成员在 D 中的访问权限，具体情况见程序中的注释。

注意：类的私有成员永远只能被本类内部的成员所访问！因此，不允许派生类使用 **using** 声明改变基类 **private** 成员在派生类中的访问权限，否则派生类就可以轻易修改基类 **private** 成员的访问权限，从而间接访问类的私有成员，这与类的封装思想是相违背的。

4.4.5 友元与继承

每个类只能控制本类成员的访问权限。因此，如果一个类 A 继承了其他类，那么它声明的友元也只能访问类 A 自己的全体成员，包括从基类继承到的 **public** 和 **protected** 成员。类 A 的基类和派生类并不认可这种友元关系，按照规则只能访问公有成员。

【例 4-6】 类 Deri 是基类 Base 的友元，函数 f1() 和 f2() 是类 Deri 的友元，分析下面程序中语句 L4、L5、L7 正确的原因，以及 L8 和 L6 错误的原因。

```
// Eg4-6.cpp
#include <iostream>
using namespace std;

class Base {
public:
    int x = 0;
protected:
    double y = 0;
private:
    float z = 0;
    friend class Deri; // L1
};

class Deri :public Base {
protected:
    int dx = 1;
public:
    friend void f1(Deri d); // L2
    friend void f2(Base b); // L3
    void f3(Base b) { cout<<b.x<<b.y<<b.z<<endl; } // L4, 正确
};

void f1(Deri d) {
    cout << d.x << d.y <<d.dx<<endl; // L5, 正确
    // cout<<d.z<<endl; // L6, 错误
}

void f2(Base b) {
    cout << b.x << endl; // L7, 正确
    // cout << b.y << endl; // L8, 错误
}

void main() {
    Base b;
    Deri d;
```

```

        f1(d);
        f2(b);
    }

```

语句 L1 声明了类 Deri 为类 Base 的友元，因此 Deri 的所有成员函数都可以访问 Base 的全体成员，语句 L4 处的 f3() 是 Deri 的成员函数，可以直接访问 Base 类型对象的公有成员 x，保护成员 y 和私有成员 z。

由于 f1() 函数是 Deri 类的友元，因此语句 L5 通过 Deri 类的对象 d 访问派生类自定义的成员 dx，以及从基类继承到的 public 成员 x 和 protected 成员 y，是允许的。语句 L6 访问基类 Base 对象中的私有成员 z 是错误的，因为 f1() 并非 Base 的友元。

函数 f2() 虽然接收 Base 类型的参数，但它只是 Deri 类的友元，并不是 Base 类的友元，因此只能访问 Base 类对象的 public 成员，这是语句 L7 正确和 L8 错误的原因。

4.4.6 静态成员与继承

在继承结构中，如果基类定义了静态成员，那么在整个继承结构中只有该成员的唯一定义，不论从该基类派生出了多少个或多少层次的派生类，静态成员都只有一个实例，为整个继承结构中的全体对象所共用。

静态成员的这种特征，对于管理继承结构中的共享数据或统计对象个数很有用，将共享数据或计数器设置为基类的静态成员，就能够实现这样的目的。

【例 4-7】 假设父亲生了儿子、女儿，儿子又生有孙子，构成了家族继承结构，统计家族成员的人数。

问题分析与数据抽象：用 Father、Son、Daughter、Grandson 分别表示父亲类、儿子类、女儿类和孙子类，它们通过继承形成了层次结构的继承体系。在 Father 类中设计静态成员 personNum 统计家族的人数，每构造一个对象人数就增 1，每析构一个对象就减 1；由于每个人都有姓名，因此在基类 Father 中设置数据成员 name 代表人名。为了便于派生类访问数据成员 personNum 和 name，把它们设置为 protected 权限。

```

// Eg4-7.cpp
#include <iostream>
#include<string>
using namespace std;

class Father {
protected:
    string name;
    static int personNum;
public:
    Father(string Name = ""):name(Name) { personNum++; }
    ~Father() { personNum--; }
    static int getPersonNumber() { return personNum; }
};
int Father::personNum = 0;
class Son:public Father {
public:
    Son(string name):Father(name) { }
}

```



```

};
class Daughter:Father {
public:
    Daughter(string name):Father(name) { }
};
class Grandson:public Son {
public:
    Grandson(string name):Son(name) { }
};

void main() {
    Father son("tom");
    Son sson("jack");
    Daughter dson("mike");
    {
        Grandson gson("s.jack");
        cout<<son.getPersonNumber()<<endl;           // L1, 输出 4
    }
    cout<<son.getPersonNumber()<<endl;               // L2, 输出 3
}

```

运行该程序，输出结果为：

```

4
3

```

这个结果是保存在静态成员 `personNum` 中的值。每创建一个继承结构上的对象，该值会增 1。当程序执行到 L1 语句时，已分别用 `Father`、`Son`、`Daughter`、`Grandson` 四个类各创建了一个对象，每创建一个对象，静态成员 `personNum` 就增 1，因此为 4；到语句 L2 时，`gson` 失去作用域，将调用 `Grandson` 的析构函数，`personNum` 的值减 1，因此为 3。

4.4.7 继承和类作用域

每个类都建立了属于自己的作用域，本类的全体成员都位于此作用域内，相互之间可以直接访问，不受定义先后次序的影响。例如，一个成员函数可以调用在它后面定义的另一成员函数。

当存在继承关系时，派生类的作用域嵌套在基类作用域的内层。因此，在解析类成员名称时，如果在本类的作用域内没有找到，编译器就会接着在外层的基类作用域内继续寻找该成员名称的定义，类似于如下形式：

```

class A {
    int g();
    .....
};
class B:public A {
    int h(int);
    .....
};
class C:public B {
    int c;   int h();   int f(int);
    .....
}

```

```
};
```

经编译器处理后，形成了类似于如下块作用域：

```
A {
    int g() { }
    .....
    B {
        int h(int) {...};
        .....
        C {
            int c;
            int h() {...};
            int f(int i) { ..... return B::h(i); }    // L1
            .....
        }
    }
}
.....
C xa;
xa.g();
xa.h(3);                                           // L2, 错误
xa.B::h(3);                                       // L3, 正确
```

当执行“xa.g()”时，就会从块作用域 C 逐层向外寻找 g() 的定义。由于类 C 没有定义 g()，接着在直接基类 B 所在的外层作用域内寻找 g()，仍然没有找到，接下来在最外层基类 A 的作用域内查找，找到了函数 g() 并调用它。

在匹配寻找派生类对象成员名称的过程中，一旦在某个作用域内找到了，就停止查找。即使外层作用域内还有同名成员，也不找了。接下来进行调用函数的参数匹配。语句 L2 处的“xa.h(3)”错误的原因就在这里，在类 C 的作用域内找到了成员函数 h()，但不需要参数，虽然基类 B 的外层作用域内有符合要求的成员函数 h()，但编译器不会向外层继续查询，也不会调用它。

因此，如果派生类和基类有同名成员，那么派生类中的同名成员会隐藏外层基类作用域中的同名成员，如果需要在派生类中访问基类的同名成员，可用“类名::成员名”进行限定，如语句 L3 所示。

类作用域的这种嵌套方法，使内层的派生类能够像使用自己的成员一样使用外层基类作用域内的成员，但外层基类不能操作内层派生类作用域内的成员，正好满足了继承技术的实际需求。

4.5 构造函数和析构函数

在继承结构中，派生类不但继承了基类的数据成员，而且可能定义了新数据成员，这些成员都需要通过构造函数进行初始化。与第 3 章介绍的类的设计原则相同，位于继承结构中的类（基类、派生类）也需要设计构造函数、复制构造函数、赋值运算符函数、移动复制构造函数、移动赋值运算符函数和析构函数控制它们的对象在执行相应操作时的行为。如果一个类（派生类）没有定义它们，编译器就会在符合条件的情况下，自动为它们生成相应的默认函数。

例如，本章前面例子中的基类和派生类都没有定义构造函数，但是它们都符合默认构造函数的生成规则，C++编译器会在必要时自动为它们创建默认的构造函数，并用它们对基类和派生类的数据成员进行初始化。

4.5.1 派生类构造函数的建立规则

1. 派生类只能通过构造函数初始化列表对基类进行初始化，但可以通过构造函数初始化列表或类内初始值对对象成员进行初始化

派生类可能有多个基类，也可能包括多个对象成员。在创建派生类对象时，派生类的构造函数除了要负责本类成员的初始化外，还要调用基类和对象成员的构造函数，并向它们传递参数，以完成基类子对象和对象成员的建立和初始化。

派生类只能采用构造函数初始化列表的方式向基类构造函数传递参数，也可以在列表中对对象成员传递参数（对象成员还可以用类内初始值初始化）。形如：

```
派生类构造函数名(参数表):基类构造函数名(参数表), 对象成员名1(参数表), ... {  
    .....  
}
```

2. 派生类必须定义构造函数的情况

当基类只含有带参数的构造函数时，即使派生类本身没有数据成员要初始化，也必须定义构造函数，并以构造函数初始化列表的方式向基类或对象成员的构造函数传递参数，以实现基类子对象和对象成员的初始化。而且，构造函数初始化列表中的命令执行完成后，才会执行构造函数体中的程序代码。当然，对象成员还可以采用类内初始值方式进行初始化。

【例 4-8】 为例 4-3 的类 **Rectangle** 添加构造函数，并通过构造函数初始化矩形的长和宽。不修改例 4-3 的程序代码，只在类 **Rectangle** 中添加构造函数，如下所示。

```
#include <iostream>  
using namespace std;  
  
class Rectangle {  
public:  
    .....  
    Rectangle(double w = 0, double len = 0):width(w), length(len) { }  
    .....  
};  
.....
```

编译本程序,出现“无法引用 Cube 的默认构造函数”错误。原因是类 **Cube** 的基类 **Rectangle** 有了构造函数，编译器不会再为它生成默认构造函数。在这种情况下，编译器也不会为派生类 **Cube** 生成默认构造函数。因此，类 **Cube** 必须定义构造函数，并通过它为基类 **Rectangle** 的构造函数提供初始值。

在类 **Cube** 中添加构造函数，并在其初始化列表中调用基类构造函数，程序就正确了。

```
class Cube :public Rectangle {  
public:  
    .....  
    Cube(double w = 0, double len = 0, double h = 0):Rectangle(w, len), height(h) { }
```

```
private:
    double height;
};
.....
```

本例表明：在基类有其他构造函数而没有默认构造函数的情况下，编译器不会为派生类生成默认构造函数，派生类必须定义构造函数，并通过它为基类构造函数提供初始化值，以完成对基类对象的初始化。

3. 派生类可以不定义构造函数的情况

当具有下述情况之一时，派生类可以不定义构造函数：基类没有定义任何构造函数，基类有默认构造函数（包括无参构造函数和全部参数都有默认值的构造函数）。

在这些情况下，派生类可以不向基类传递构造函数的参数，甚至不需要构造函数（如果派生类没有成员需要初始化）。如果派生类没有定义构造函数，编译器就会自动为派生类生成默认构造函数，并通过它调用基类的默认构造函数，实现基类成员的初始化。

【例 4-9】 类 A 具有默认构造函数，其派生类 B 没有成员要初始化，不必定义构造函数。

```
// Eg4-9.cpp
#include <iostream>
using namespace std;

class A {
public:
    A() { cout<<"Constructing A"<<endl; }
    ~A() { cout<<"Destructing A"<<endl; }
};

class B:public A {
public:
    ~B() { cout<<"Destructing B"<<endl; }
};

void main() {
    B b;
}
```

程序运行结果如下：

```
Constructing A
Destructing B
Destructing A
```

本例虽然派生类 B 没有定义构造函数，但它符合生成默认构造函数的条件：① 基类 A 有默认构造函数；② 派生类 B 没有构造函数。编译器会为它生成一个默认构造函数，并通过它调用基类的默认构造函数，创建它的基类子对象。类似如下形式：

```
B::B():A() { }
```

在定义类 B 的对象时，这个构造函数会被自动调用，因此有了上面的程序运行结果。

4. 派生类构造函数只负责直接基类的初始化

C++语言标准有一条规则：若派生类的基类也是另一个类的派生类，则每个派生类只负责它的直接基类的构造函数调用。这条规则表明，当派生类的直接基类只有带参数的构造函

数但没有默认构造函数（包括缺省参数和无参构造函数）时，它必须在构造函数的初始化列表中调用其直接基类的构造函数，并向基类的构造函数传递参数，以实现派生类对象中的基类子对象的初始化。这条规则有一个例外，当派生类存在虚基类时，它必须为虚基类（直接虚基类或间接虚基类）的构造函数提供初始化值。

【例 4-10】 类 C 具有直接基类 B 和间接基类 A，每个派生类只负责其直接基类的构造。

```
// Eg4-10.cpp
#include <iostream>
using namespace std;

class A {
    int x;
public:
    A(int aa):x(aa) {
        cout<<"Constructing A"<<endl;
    }
    ~A(){ cout<<"Destructing A"<<endl; }
};

class B:public A {
public:
    B(int x):A(x){ cout<<"Constructing B"<<endl; } // 只负责直接基类 A 的构造
};

class C :public B{
public:
    C(int y):B(y){ cout<<"Constructing C"<<endl; } // 只负责直接基类 B 的构造
};

void main(){
    C c(1);
}
```

运行结果如下：

```
Constructing A
Constructing B
Constructing C
Destructing A
```

可以看出，第一个被调用的构造函数属于最早的基类。其过程如下：在 `main()` 中定义 `c` 对象时，将导致类 C 的基类 B 的构造函数调用；在调用 B 的构造函数时，由于 B 又是从类 A 派生的，所以先调用 A 的构造函数；然后返回到 A 的派生类 B，最后返回到 B 的派生类 C。类 B 和 C 的构造函数是在返回的过程中被调用的。

5. 派生类继承基类的构造函数 C++11

C++ 11 标准允许派生类继承直接基类的构造函数，这在之前是禁止的。这个新规则为派生类构造函数的设计带来了一定程度上的方便，特别是基类构造函数具有较多参数，而派生类没有数据成员需要初始化，但它必须提供构造函数，其唯一目的是为基类构造函数提供初始化值。在这种情况下，它只需要继承直接基类的构造函数就可以了。

继承构造函数的方法是用 `using` 在派生类中声明基类构造函数名，形式如下：

```

class Base:{ ... }
class Derived: [public] Base {           // 继承方式也可以是 private 或 protected
    .....
    using Base::Base;                   // 继承基类构造函数
    .....
}

```

“Base::Base”即基类构造函数的名称，using 语句说明了派生类要继承基类的构造函数，它不受访问权限控制，放在 public、protected 或 private 区域中没有任何区别。

用 using 在派生类中声明基类的构造函数和其他成员有所不同，声明其他成员只是使该成员在指定的派生类权限区域可见，并不生成代码。而用 using 继承基类构造函数，则会使编译器在派生类中生成类似于如下形式的派生类构造函数代码：

```
Derived(参数表):Base(形参表) { }
```

其中的参数表与基类 Base 构造函数的参数表一致，这些参数是提供给基类构造函数使用的。如果基类有多个构造函数，那么 using 语句会在派生类中为基类的每个构造函数生成一个与之对应的构造函数，并具有与基类构造函数相同的访问权限。

【例 4-11】 类 A 具有数据成员 x、y，并且定义了初始化它们的构造函数；类 B 从 A 派生，没有任何成员需要初始化；类 C 从类 B 派生，具有新定义数据成员 c。设计类 A、B、C 的构造函数。

问题分析：按照规则，类 B 虽然没有数据成员要初始化，但是它必须为基类 A 的构造函数提供初值（除非 A 具有默认构造函数），现在可以通过继承 A 的构造函数使问题简化，类 C 要定义构造函数以便初始化其成员 c，同时必须为直接基类 B 提供构造初值。

程序代码如下：

```

// Eg4-11.cpp
#include <iostream>
using namespace std;

class A {
    int x, y;
public:
    A(int aa) :x(aa) { cout<<"Constructing A:x = \t"<<x<<endl; }
    A(int a, int b):x(a), y(b) {
        cout<<"Constructing A:x = \t"<<x<<endl;
    }
};

class B :public A {
public:
    using A::A; // L1
    /* B(int x):A(x) { // L2
        cout<<"Constructing B\t"<<endl;
    } */
};

class C :public B {
    using B::B; // L3
    int c;
}

```

```

public:
    C(int x, int y, int z):B(x,y), c(z) {                // L4
        cout<<"Constructing C:\t"<<c<<endl;
    }
};

void main() {
    B b1(1), b2(8, 9);                                // L5
    C c1(1), c2(3, 4);                                // L6
}

```

程序运行结果如下:

```

Constructing A:x = 1
Constructing A:x = 8
Constructing A:x = 1
Constructing A:x = 3

```

语句 L1 使类 B 继承了基类 A 的构造函数, 类 A 有两个构造函数, 所以编译器会为类 B 生成两个构造函数, 类似于如下形式:

```

B::B(int a) : A(a) { }
B::B(int a, int b) : A(a, b) { }

```

因此, 语句 L5 定义 b1、b2 时分别调用了类 B 通过继承得到的构造函数 B(int)和 B(int, int), 并将参数传递给了基类 A 的构造函数, 如运行结果的第 1、2 行所示。如果 B 没有继承 A 的构造函数, 它必须编写类似语句 L2 处的构造函数, 否则会产生编译错误。

语句 L3 继承了基类 B 的构造函数, 编译器会为类 C 生成如下两个构造函数:

```

C::C(int a) : B(a) { }
C::C(int a, int b) : B(a, b) { }

```

语句 L6 定义类 C 的对象 c1 和 c2 时调用的正是这两个构造函数。

语句 L4 定义了类 C 的构造函数, 通过它初始化数据成员 c。但是, 该构造函数的初始化列表中还调用了基类构造函数 B(a, b), 这个构造函数是由语句 L1 产生的。

说明:

- ① 派生类用 using 声明它继承基类的构造函数时, using 语句的位置不受访问权限影响, 继承来的构造函数的访问权限与基类构造函数原有访问权限相同。
- ② 如果基类有多个构造函数, 就都会被继承, 例外情况如④所列。
- ③ 如果基类构造函数具有参数默认值, 这些默认值不会被继承, 继承将为派生类生成多个构造函数, 每个构造函数的参数依次少一个。例如, 有类 A 和 B 如下:

```

class A {
    int x, y;
public:
    A(int a, int b = 2) :x(a), y(b) { cout<<"a = "<<a<<"\tb = "<<b<<endl; }
};
class B :public A {
public:
    using A::A;
};

```

继承将为类 B 生成构造函数 B(int a) : A(a, 2)和 B(int a, int b) : A(a, b)。

④ 基类的默认构造函数、复制构造函数和移动复制构造函数不能够被继承。派生类如果没有定义这些构造函数，在符合对应默认函数生成规则的情况下，编译器将按规则自动为派生类生成对应的默认函数版本。在确定是否应生成这些构造函数时，将忽略继承构造函数的存在。比如，当一个类只有继承来的构造函数但没有定义其他构造函数时，编译器就认为该类没有定义构造函数，会自动为它生成默认构造函数、复制构造函数和赋值运算符函数。

⑤ 若派生类在继承基类构造函数的同时还需要定义其他构造函数，则必须按照前面的规则定义，即必须在构造函数初始化列表中为基类构造函数提供初始化值（除非基类有默认构造函数）。

4.5.2 派生类构造函数和析构函数的调用次序

如果派生类具有多个基类和对象成员，在创建派生类对象时，它们的构造函数调用次序如下：类内初始值 → 基类构造函数 → 对象成员构造函数 → 派生类构造函数。

① 当有多个基类时，将按照它们在继承方式中的声明次序调用，与它们在构造函数初始化列表中的次序无关。当基类 **B** 本身又是另一个类 **A** 的派生类时，则先调用基类 **A** 的构造函数，再调用基类 **B** 的构造函数。

② 当有多个对象成员时，将按照它们在派生类中的声明次序调用，与它们在构造函数初始化列表中的次序无关。

③ 当构造函数初始化列表中的基类和对象成员的构造函数调用完成后，才执行派生类构造函数体中的程序代码。

对象析构的次序与其构造次序正好相反，即按照与构造函数相反的次序调用析构函数，最先构造的对象最后销毁。

【例 4-12】 类 **D** 从类 **B** 派生，并具有用类 **A** 和 **C** 建立的对象成员。分析创建 **D** 的对象时，基类、对象成员、派生类构造函数和析构函数的调用次序。

```
// Eg4-12.cpp
#include <iostream>
using namespace std;

class A {
    int x;
public:
    A(int i = 0):x(i) { cout<<"Construct A----"<<x<<endl; }
    ~A() { cout<<"Des A----"<<x<<endl; }
};

class B {
    int y;
public:
    B(int i):y(i) { cout<<"Construct B----"<<y<<endl; }
    ~B() { cout<<"Des B----"<<y<<endl; }
};

class C {
    int z;
public:
    C(int i):z(i) { cout<<"Construct C----"<<z<<endl; }
```

```

    ~C() { cout<<"Des C----"<<z<<endl; }
};
class D : public B {
public:
    C c1, c2;
    A a0, a4;
    D():a4(4), c2(2), c1(1), B(1) {
        cout<<"Construct D----5"<<endl;
    }
    ~D() { cout<<"Des D----5"<<endl; }
};

void main() {
    D d;
}

```

派生类 D 具有基类 B 和对象成员 c1、c2、a0、a4，根据前述调用次序，当在 main() 函数中建立对象 d 时，将按照 B→c1→c2→a0→a4→d 的次序调用基类和对象成员的构造函数。

尽管类 D 的构造函数初始化列表是按 a4→c2→c1→B 次序排列的，但这个排列并不影响各构造函数和析构函数的调用次序，所以本程序的运行结果如下：

```

Construct B----1
Construct C----1
Construct C----2
Construct A----0
Construct A----4
Construct D----5
Des D----5
Des A----4
Des A----0
Des C----2
Des C----1
Des B----1

```

该结果也证实了前面所讨论的构造函数和析构函数的调用次序。

在本程序中还需要注意成员对象 a0 的构造。在类 D 的构造函数初始化列表中并没有对 a0 进行初始化，但从输出结果可以看出，a0 仍然按其在类 D 中的声明次序进行了初始化。其原因是类 A 具有默认构造函数，编译器会按 a0 在类 D 中的声明次序自动调用此构造函数。在程序编译时，编译器会改写 D 的构造函数为如下类似形式：

```

D::D():a0(0), a4(4), c2(2), c1(1), B(1) { ... }

```

其中，对 a0 默认构造函数的调用是编译器加上去的。

说明：

① 构造函数与类对象的建立有密切关系，继承使类的构造函数变得更加复杂，派生类构造函数除了初始化本类成员，还要负责为基类和对象成员的构造函数传递初始化参数。

② 第 3 章讨论的非继承的构造函数的原则同样适用于派生类。例如，若定义派生类的对象数组就需要派生类具有默认构造函数，就可能要求它的基类和对象成员也要有默认构造函数，等等。

4.5.3 派生类的赋值、复制和移动操作

派生类对象的构造函数不但要初始化自身的成员，而且肩负着初始化基类成员的责任。同样，派生类的赋值函数和复制构造函数以及移动赋值运算符函数和移动复制构造函数不但要执行派生类成员的复制和移动，还要负责基类部分数据成员的复制和移动。

如果一个类没有定义赋值运算符函数、复制构造函数、移动赋值运算符函数和移动复制构造函数，编译器会为它们自动生成一个默认的函数版本。当然，生成默认函数是有条件的，当一个类有虚析构函数时，即使没有定义这些函数，编译器也不会生成它们。另外，如果类定义了赋值运算符或复制构造函数，编译器也不会为它生成移动赋值运算符和移动构造函数。

派生类在定义赋值运算符函数、复制构造函数和它们的移动函数版本时，要负责对基类成员进行相应的处理，它们应当调用基类与之对应的赋值运算符函数、复制构造函数和移动复制构造函数，来完成基类成员的相应处理。下面的例子介绍派生类定义这些函数的方法。

【例 4-13】 类 A 具有数据成员 x，并定义了赋值运算符函数、复制构造函数和它们的移动函数版本，以实现对象间的赋值、复制或移动操作，类 B 从类 A 派生，并有数据成员 y。设计类 B 的赋值运算符函数、复制构造函数和移动复制构造函数，实现派生类 B 的对象间的赋值、复制和移动操作。

```
// Eg4-13.cpp
#include <iostream>
using namespace std;

class A {
    int x;
public:
    A(int a = 0, int b = 2) :x(a) { }
    A &operator = (A& o) {
        x = o.x;
        cout<<"In A =(A&)"<<endl;
        return *this;
    }
    A& operator = (A &o) = default; // 生成默认的移动赋值函数
    A(A &o):x(o.x) { cout<<"In A(&)"<<endl; }
    A(A &o):x(std::move(o.x)) { cout<<"In A(&&)"<<endl; }
};

class B :public A {
    int y;
public:
    B(int a = 0, int b = 0) : A(a), y(b){ }
    B& operator = (B& o) {
        A::operator = (o);
        cout<<"In B = (B&)"<<endl;
        return *this;
    }
    B& operator = (B &o) {
        A::operator = (std::move(o));
        cout<<"In B = (B&&)" << endl;
        return *this;
    }
};
```

```

    }
    B(B &o):A(o) { cout << "In B(&)" << endl; }
    B(B &&o):A(std::move(o)) { cout<<"In B(&&)"<< endl; }
};

void main() {
    B b, b1(1, 2);
    b = b1; // L1
    B b2(b); // L2
    B b3 = std::move(B(8, 9)); // L3
    b1 = std::move(b3); // L4
}

```

程序运行结果如下：

```

In A = (A&) // L1 的输出
In B = (B&) // L1 的输出
In A(&) // L2 的输出
In B(&) // L2 的输出
In A(&&) // L3 的输出
In B(&&) // L3 的输出
In B = (B&&) // L4 的输出

```

程序运行结果的前 6 行表明，在执行派生类对象的赋值、复制、移动时都正确调用了基类和派生类的对应函数。最后一行是执行语句 L4 时的输出，用于实现派生类对象的移动赋值，它实际上也正确执行了对基类部分成员 x 的移动，之所以只输出了 “In B = (B&&)”，是因为它调用了基类 A 的默认移动赋值运算符函数，而该函数没有输出。

说明：派生类在定义赋值运算符函数、复制构造函数及对应的移动函数版本时，无论基类的对应函数是自定义的还是系统生成的默认函数版本，都应当执行它们，以实现对象的正确复制或移动。因此，应当将基类的自定义赋值运算符函数、复制构造函数和它们的移动构造函数版本都设置为 public 权限，以便派生类能够访问它们。

4.6 基类与派生类对象的关系

通过继承，派生类获得了基类成员的一份副本，构成了派生类对象内部的一个基类子对象。因此，基类对象与派生类对象之间存在赋值相容性。**赋值相容**是指在公有派生方式下，凡是需要基类对象的地方都可以使用派生类对象。基类对象能够解决的问题用派生类对象也能够解决，包括如下 3 种情况：① 把派生类对象赋值给基类对象；② 把派生类对象的地址赋值给基类指针；③ 用派生类对象初始化基类对象的引用。原因是，任何一个派生类对象的内部都包含一个基类子对象，在进行派生类对象向基类对象的赋值时，C++采用截取的方法从派生类对象中复制其基类子对象并将之赋值给基类对象，如图 4-6 所示。

在进行了上面的赋值后，可以通过基类对象访问派生类对象，但只能访问派生类从基类继承而来的成员，不能访问派生类定义的新成员。反之则不行，即不能进行从基类对象到派生类对象的赋值转换，因为基类对象中并不包含派生类子对象，无法从中找到派生类自身定义的成员。因此，不能把基类对象赋值给派生类对象，不能把基类对象的地址赋值给派生类对象的指针，也不能把基类对象作为派生对象的引用。

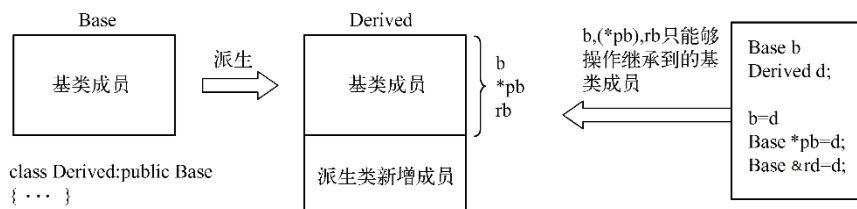


图 4-6 派生类和基类之间的赋值相容关系

4.6.1 派生类对象对基类对象的赋值和初始化

在把派生类对象赋值给基类对象，或者用派生类对象初始化基类对象时，并不存在从派生类向基类的类型转换。本质上是执行基类对象的复制构造函数，赋值运算符函数及其移动函数版本，通过它们把派生类对象中从基类继承到的数据成员复制给基类对象。

【例 4-14】 类 B 从类 A 派生，设计类 B 的复制构造函数和赋值运算符函数，并验证把派生对象赋值给基类对象或通过它初始化基类对象时，相关函数的调用情况。

```
// Eg4-14.cpp
#include <iostream>
using namespace std;

class A {
    int a;
public:
    void setA(int x) { a = x; }
    int getA() { return a; }
    A() :a(0) { cout<<"A::A()"<<endl; }
    A(A& o):a(o.a) { cout<<"A::A(&o)"<<endl; }
    A& operator = (A o) {
        a = o.a;
        cout<< "A::operator = "<<endl;
        return *this;
    }
};

class B :public A {
    int b;
public:
    void setB(int x) { b = x; }
    int getB() { return b; }
    B():b(0) { cout<<"B::B()"<<endl; }
    B(B& o):b(o.b) { cout<<"B::B(&o)"<<endl; }
    B& operator = (B o) {
        b = o.b;
        cout<<"B::operator = "<<endl;
        return *this;
    }
};

void main() {
    A a1, *pA;
    B b1, *pB;
```

```

    b1.setA(2);
    a1 = b1;
    b1.setA(10);
    A a2 = b1;
    a2.setA(1);
    cout<<a1.getA()<<endl;           // L1, 输出 2
    cout<<b1.getA()<<endl;           // L2, 输出 10
    cout<<a2.getA()<<endl;           // L3, 输出 1
    // a2.setB(5);                     // L4, 错误
    // b1 = a1;                         // L5, 错误
}

```

程序运行结果如下：

```

A::A()                               // 1
A::A()                               // 2
B::B()                               // 3
A::A(&o)                             // 4
A::operator=                         // 5
A::A(&o)                             // 6
2                                     // 7
10                                   // 8
1                                   // 9

```

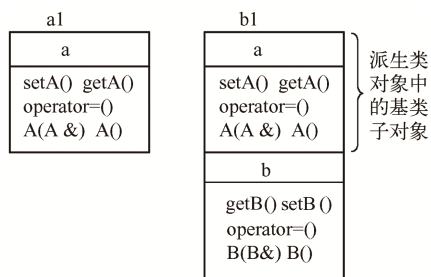


图 4-7 A 和 B 的对象示意

基类对象 **a1** 和派生类对象 **b1** 的基本结构如图 4-7 所示。这里只是为了说明基类 **A** 与派生类 **B** 之间的关系，内存中的 **a1**、**b1** 对象是不包括成员函数的。现结合此图和程序中的注释，理解程序中的各赋值语句，分析程序中各输出语句的结论。

程序运行结果第 1 行是命令 “A a1” 的输出，第 2、3 行是 “B b1” 定义 **b1** 时的输出。

第 4、5 行都是执行 “a1 = b1;” 语句时的输出。这条语句等价于 “a1.operator=(b1);”，在向 **operator** 传递参数 **b1** 时调用基类复制构造函数，输出了第 4 行，执行本函数时输出了第 5 行。

第 6 行是执行 “A a2 = b1;” 语句时调用基类 **A** 的复制构造函数的输出，等价于 “A a2(b1);”。

最后 3 行输出表明，当完成派生类对象向基类对象的赋值或初始化之后，两者之间的数据成员是独立存在的，不再有任何联系，一个对象数据成员的修改不会引起另一对象数据成员的变化。

上面的分析表明：在把派生类对象赋值给基类对象时会调用基类的赋值运算符函数；在用派生类对象初始化基类对象时，会调用基类的复制构造函数，将派生类对象中的基类子对象的数据成员复制给被赋值或被初始化的基类对象。

main() 函数中语句 **L4** 和 **L5** 的错误表明，只能把派生类对象赋值给基类对象，或者用它初始化基类对象；并且，通过基类对象只能够访问基类成员，不能访问派生类新增加的成员。

当初始化或赋值一个类的对象时，实际上是调用某函数。基类和派生类对象之间只允许从派生类对象到基类对象的赋值或初始化。把派生类对象赋值给基类对象时，实际运行的只是基类定义的赋值运算符函数，该函数只能处理基类自身的数据成员。当用派生类对象初始

化基类对象时，实际上也只调用基类的复制构造函数，该函数只能复制基类定义的成员函数。

4.6.2 派生类对象与基类对象的类型转换

每个派生类对象中都包含有一个基类子对象，编译器可以由此实现派生类对象向基类对象的类型转换。反过来，基类对象中没有派生类新增加的成员，不能将它转换成派生类对象，因为它无法补充派生类定义的成员。因此，不存在从基类对象到派生类对象的类型转换。

1. 派生类对象到基类对象的隐式类型转换

由 4.6.1 节可知，用派生类对象赋值或初始化基类对象时，实际是通过赋值运算符函数或复制构造函数复制派生类对象中的基类子对象，并没有执行类型转换。但是，当把基类对象的指针或引用绑定到派生类对象时，编译器会自动执行从派生类对象到基类对象的隐式类型转换。

例如，对于例 4-14 的基类 A 和派生类 B，下面的代码会发生类型转换。

```
B b, b1, b2;
A *pa = &b1;           // 正确，执行派生类向基类的转换
A &ra = b2              // 正确，执行派生类向基类的转换
A a = b;                // 正确，没有类型转换，通过基类复制构造函数初始化 a
```

在 pa 和 ra 的定义过程中，派生类对象 b1 和 b2 的类型为 B，会被转换成基类类型 A，转换的方法是截取派生类对象中的基类子对象部分，然后将 pa 的 ra 分别绑定到转换后的对象上。

不论以哪种方式把派生类对象赋值给基类对象，都只能访问到派生类对象中的基类子对象的成员，不能访问派生类的自定义成员。例如：

```
a.setB(10)              // 错误，不能通过基类对象访问派生类定义的成员
pa->getB();              // 错误，不能通过基类对象指针访问派生类定义的成员
ra.setB(1);             // 错误，不能通过基类对象的引用访问派生类定义的成员
a.setA(10);             // 正确
pa->getA();              // 正确
ra.setA(1);             // 正确
```

2. 基类对象到派生类对象的类型转换

只能把派生类对象赋值给基类对象（包括对象、引用、指针），不能把基类对象赋值给派生类对象。即使一个基类对象的指针或引用实际绑定到了一个派生类对象，编译器也不会执行从基类到派生类的隐式类型转换。例如，针对前面对 b、pa、pb 和 a 定义和赋值的代码段，下面的赋值方式是错误的：

```
b = a;                  // 错误，不允许从基类向派生类的转换
B *pb = pa;             // 错误，不能把基类对象的地址赋值给指向派生类对象的指针
B &rB = ra;              // 错误，不能把基类对象作为派生类对象的引用
```

虽然基类的指针 pa 和引用 ra 实际绑定到了派生类对象，它们也只对 B 从基类 A 继承而来的对象部分具有操作权限，对于 B 新增加的成员是无从知晓的。但是，pa 和 ra 又确实绑定的是一个派生类对象，其对应内存区域中是一个完整的派生类对象，把它们转换成派生类类型是安全的。在这种情况下，可以用如下方式指示编译器实施强制类型转换：


```
B *pb = static_cast<B*> (pa);
B &rB = static_cast<B&> (rA);
```

3. 对象、指针和引用的区别

把派生类对象赋值给基类对象或用派生类对象初始化基类对象，与把基类对象的指针或引用绑定到派生类对象存在一定的区别。前者在完成赋值或初始化操作后，基类对象与派生类对象就没有关系了，而指针或引用从来就没有生成新对象，它们操作的是派生类对象内部的基类子对象。

```
void main() {
    B b, b1;
    A a = b, *pa = &b1, &rA = b1;           // L1
    b.setA(10);                             // L2
    a.setA(9);                              // L3
    pa->setA(20);                             // L4
    rA.setA(1);                             // L5
    cout<<b.getA( );                        // L6, 输出 10
    cout<<b1.getA( );                      // L7, 输出 1
}
```

执行语句 L1 后，对象 a 和 b 是没有任何联系的两个对象，语句 L2 将对象 b 的数据成员 a 设置为 10，语句 L3 将对象 a 的数据成员 a 设置为 9，但它与对象 b 的数据成员 a 没有任何关系。因此，语句 L6 输出的是数据 10。

执行语句 L1 后，pa 和 rA 都绑定到了对象 b1 内部的基类子对象上，它们操作的都是对象 b1。语句 L4 通过指针 pa 将对象 b1 从基类继承到的数据成员 a 修改为 20，随后语句 L5 通过引用将它修改为 1，所以语句 L7 输出的是引用 rA 修改后的 1。

4. 派生类对象作为函数参数传递给基类对象

在函数中，如果形式参数是基类对象，也可以用派生类对象作为实参。对于例 4-14，下面的程序段说明了把派生类对象作为实参传递给函数的基类对象参数的三种情况。

```
void f1(A a, int x) { a.setA(x); }
void f2(A *pA, int x) { pA->setA(x); }
void f3(A &rA, int x) { rA.setA(x); }
void main() {
    B b;
    b.setA(1);
    f1(b, 10);                             // b.a 未被 f1()修改，仍然为 1
    f2(&b, 10);                             // b.a 被 f2()修改为 10
    f3(b, 15);                             // b.a 被 f3()修改为 15
}
```

1) 形参是基类对象

如果形参是基类对象，在调用该函数时，可以用派生类对象作为实参调用该函数。这种参数传递方式将调用基类对象（形参对象）的复制构造函数，把派生类中的基类子对象的对应数据成员复制给函数的形参对象，参数传递完成后，形参对象与实参对象就没有任何关系了，所以不能修改实参（派生类对象）的数据成员值。

函数调用 f1(b, 10)就是这种类型的参数传递方式。通过形参对象 a 的复制构造函数，C++

将 **b** 内部的基类子对象复制给形参 **a** 后，实参 **b** 与形参 **a** 就没有关系了。**f1()** 在其函数体内修改形参 **a** 的数据成员值为 10，但此修改与实参 **b** 没有关系。所以，函数调用 **f1(b, 10)** 执行后，**b** 内的基类成员 **a** 的值仍然为 1，并未被函数 **f1()** 所修改。

2) 形参是基类对象的引用或指针

如果函数的形参是基类对象的引用或指针，在函数调用时，实参可以是派生类类型的对象或其地址。编译器将进行隐式类型转换，基类类型的形参引用或指针被绑定到派生类实参对象内部的基类子对象上，形参操作的实际上是实参本身。因此，这两种参数传递方式都能够修改实参的值。

函数 **f2()** 和 **f3()** 的第一个参数分别是基类 **A** 的指针和引用，所以当 “**f2(&b, 10);**” 执行时，**f2()** 的指针形参 **pA** 会被绑定到派生类对象 **b** 内部的基类子对象上，即指向派生类对象 **b** 内部的基类子对象，函数执行后，会将 **b** 的数据成员 **a** 修改为 10。同样，当 “**f3(b, 15);**” 语句执行时，**f3()** 的引用形参 **rA** 会被绑定到 **b** 内部的基类子对象上，即 **rA** 是 **b** 继承到的基类子对象的别名，函数执行后会将对象 **b** 的数据成员 **a** 修改为 15。

4.7 多继承

4.7.1 多继承的概念和应用

C++ 允许一个类从一个或多个基类继承。如果一个类只有一个基类，就称为单继承。如果一个类具有两个或两个以上的基类，就称为多继承。多继承的形式如下：

```
class 派生类名:[继承方式] 基类名 1, [继承方式] 基类名 2, ... {  
    派生类成员声明或定义;  
};
```

其中，继承方式可以是 **public**、**protected**、**private**，分别对应公有、保护和私有继承。通过多继承，派生类就获得了多个基类的数据成员和成员函数的一份复制品，具有多个基类的复合功能。

例如，假设有 4 个类 **Base1**、**Base2**、**Base3** 和 **Derived**，它们的继承关系如图 4-8 所示。其中，**Base1** 有公有成员函数 **setx()**、保护成员函数 **getx()**、私有数据成员 **a**；**Base2** 有公有成员函数 **sety()** 和 **gety()**、私有数据成员 **y**；**Base3** 有公有成员函数 **setz()** 和 **getz()**、私有成员 **z**；类 **Derived** 虽然只定义了公有成员函数 **setd()**、**display()** 和私有数据成员 **d**，但它通过继承复制了 **Base1**、**Base2**、**Base3** 三个基类的全部数据成员和成员函数，具有三个基类的复合功能。

【例 4-15】 实现图 4-8 继承关系的简单程序。

```
// Eg4-15.cpp  
#include <iostream>  
using namespace std;  
  
class Base1{  
private:  
    int x;  
protected:  
    int getx(){ return x; }  
};
```

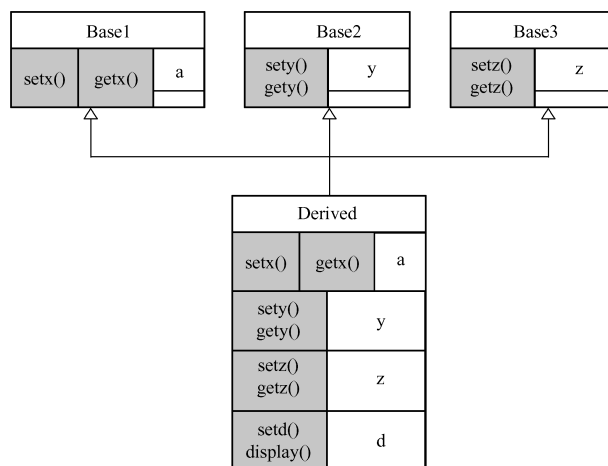


图 4-8 多重继承的示意

```

public:
    void setx(int a = 1){ x = a; }
};
class Base2{
private:
    int y;
public:
    void sety(int a){ y = a; }
    int gety(){ return y; }
};
class Base3{
private:
    int z;
public:
    void setz(int a) { z = a; }
    int getz() { return z; }
};
class Derived:public Base1, public Base2, public Base3 {
private:
    int d;
public:
    void setd(int a){ d = a; }
    void display();
};
void Derived::display() {
    cout<<"Base1 ...x = "<<getx()<<endl;
    cout<<"Base2 ...y = "<<gety()<<endl;
    cout<<"Base3 ...z = "<<getz()<<endl;
    cout<<"Derived ...d = "<<d<<endl;
}

void main() {
    Derived obj;
    obj.setx(1);
    obj.sety(2);
}
  
```

```
    obj.setz(3);
    obj.setd(4);
    obj.display();
}
```

运行结果如下：

```
Base1 ...x = 1
Base2 ...y = 2
Base3 ...z = 3
Derived ...d = 4
```

虽然 **Derived** 类只定义了两个成员函数 `setd()`、`display()`和数据成员 `d`，但它拥有从基类继承来的公有成员函数 `setx()`、`getx()`、`sety()`、`gety()`、`setz()`、`getz()`，并且允许 **Derived** 的外部函数直接调用这些成员函数。

4.7.2 多继承方式下的成员二义性

在多继承方式下，派生类继承了多个基类的成员，当两个不同基类拥有同名成员时，容易产生命名冲突问题。

【例 4-16】 类 **A** 和类 **B** 是 **MI** 的基类，它们都有成员函数 `f()`，因此在类 **MI** 中就有通过继承而来的两个同名成员函数 `f()`，在调用时易产生二义性冲突。

```
// Eg4-16.cpp
class A {
public:
    void f(){ cout<<"From A"<<endl; }
};

class B {
public:
    void f() { cout<<"From B"<<endl; }
};

class MI:public A, public B {
public:
    void g() { cout<<"From MI"<<endl; }
};

void main() {
    MI mi;
    mi.f(); // 错误
    mi.A::f(); // 正确
}
```

`mi.f()`调用会产生二义性的命名冲突。因为 **MI** 有两个名为 `f` 的成员函数，一个来源于类 **A**，另一个来源于类 **B**。编译器无法确定 `mi.f()`是调用 `A::f()`还是 `B::f()`，所以产生编译错误。

在这种情况下，应当用类域限定符明确指出调用函数所属的基类。如要调用来源于基类 **A** 中的函数 `f()`，就应明确地写成 `mi.A::f()`，要调用来源于 **B** 的函数 `f()`，就应写成 `mi.B::f()`。

4.7.3 多继承的构造函数和析构函数

当一个类具有多个基类时，派生类必须负责为每个基类的构造函数提供初始化参数，构造的方法和原则与单继承相同。构造函数的调用次序仍然是先基类再对象成员，然后是派生类的构造函数。基类构造函数的调用次序与它们在被继承时的声明次序相同，与它们在派生类构造函数初始化列表中的次序没有关系。同样，析构函数的调用次序与构造函数的调用次序相反。

【例 4-17】 类 Base1、Base2、Base3、Derived 的继承关系见图 4-8，验证其构造函数和析构函数的调用次序。

```
// Eg4-17.cpp
#include <iostream>
using namespace std;

class Base1 {
private:
    int x;
public:
    Base1(int a = 1) {
        x = a;
        cout<<"Base1 constructor x = "<<x<<endl;
    }
    ~Base1() { cout<<"Base1 destructor ..."<<endl; }
};

class Base2 {
private:
    int y;
public:
    Base2(int a) {
        y = a;
        cout<<"Base2 constructor y = "<<y<<endl;
    }
    ~Base2() { cout<<"Base2 destructor ..."<<endl; }
};

class Base3 {
private:
    int z;
public:
    Base3(int a){
        z = a;
        cout<<"Base3 constructor z = "<<z<<endl;
    }
    ~Base3(){ cout<<"Base3 destructor ..."<<endl; }
};

class Derived:public Base1, protected Base2, private Base3 {
private:
    int y;
public:
    Derived(int a, int b, int c):Base3(b), Base2(a) {
```

```

        y = c;
        cout<<"Derived constructor y = "<<y<<endl;
    }
    ~Derived(){ cout<<"Derived destructor ..."<<endl; }
};

void main() {
    Derived d(2, 3, 4);
}

```

本程序的运行结果如下：

```

Base1 constructor x = 1
Base2 constructor y = 2
Base3 constructor z = 3
Derived constructor y = 4
Derived destructor ...
Base3 destructor ...
Base2 destructor ...
Base1 destructor ...

```

在派生类 **Derived** 构造函数的初始化列表中并没有对基类 **Base1** 的构造函数进行初始化，但这里是允许的，因为 **Base1** 具有默认构造函数，编译系统会自动调用它。

另外，**Derived** 在其构造函数的初始化列表中，对基类构造函数的调用次序与继承的次序并不相同，C++按继承的次序调用基类的构造函数。

4.8 虚拟继承

C++在解析派生类的成员函数时，按照以下次序查找成员函数所属的类：在派生类中查找该函数，如果找到，就确定该函数是派生类的成员函数，否则在基类中查找该成员函数。

在单继承方式下，这种解析方式能够正确地找到成员函数所属的类对象。但在多继承方式下，当一个类从多个基类派生，而这些基类从同一个类派生时，这种解析方式就会产生成员名称的二义性。例如，在一个学校的人员管理系统中，有学生、雇员、学生雇员等不同类型的人，学生雇员就是勤工俭学的学生，具有学生和雇员两类人的共同特征。经过抽象之后，将这几类人抽象成 **Student**、**Employee**、**StuEmployee** 类，并且将他们共同的特征和行为抽象形成基类 **Person**，它们之间的继承关系如下所示：

```

class Person {
    char *name;
public:
    void SetName();
    char* GetName();
    .....
}
class Student:public Person{ ... }
class Employee:public person{ ... }
class StuEmployee:public Student, public Employee{ ... }

```

图 4-9 是各类的继承结构示意图，**Person** 类有两个成员函数 **SetName()**和 **GetName()**，并且

是 **Student** 和 **Employee** 的基类。因此，**Student** 和 **Employee** 两个类中都有 **Person** 的数据成员和成员函数的一份副本。类 **stuEmployee** 从 **Student** 和 **Employee** 多重派生，具有这两个类的数据成员和成员函数的一份副本，其结构如图 4-10 所示。

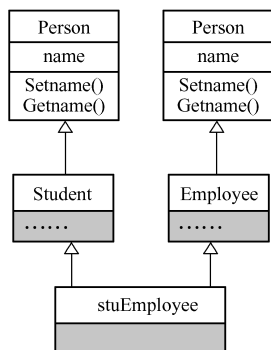


图 4-9 学校人员多重继承示意

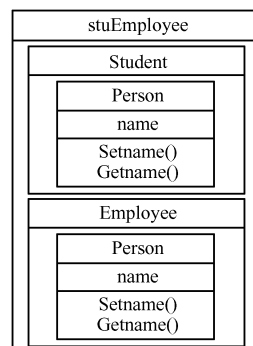


图 4-10 **stuEmployee** 类结构示意

从图 4-10 可以看出，类 **stuEmployee** 中拥有类 **Person** 的两份成员，这种形式的继承容易使派生类对象的成员解析产生二义性。例如，假设有下面的成员引用：

```
stuEmployee s;
s.SetName();           // 错误，二义性冲突
```

对于 **s.SetName()** 函数调用，编译器先在类 **stuEmployee** 自定义的成员中查找成员函数 **SetName()**，结果没有找到；然后，编译器会在 **stuEmployee** 的基类中查找成员函数 **SetName()**，但在 **Student** 和 **Employee** 两个基类中都找到了成员函数 **SetName()**，编译器不能确定应该调用哪个基类的 **SetName()**，因此产生二义性的命名冲突。出现这种问题时，可以指明调用成员函数所属的类来解决二义性的命名冲突。例如：

```
s.Student::SetName();
s.Employee::SetName();
```

但是，这样的调用方式并未解决本质问题：在同一个对象 **s** 中存在 **Person** 的两份不同数据成员，容易产生数据的不一致性。为了解决这类问题，C++ 引入了虚拟继承的概念。

1. 虚拟继承的定义方式

利用关键字 **virtual** 限定继承方式，将公共基类指定为虚基类，就可以使该基类的成员在派生类中只有一份副本。虚基类的定义形式如下：

```
class 派生类名:virtual [继承方式] 基类名 1, virtual [继承方式] 基类名 2, ... {
    派生类成员声明与定义;
};
```

对于前面的类 **stuEmployee**，若采用如下虚拟继承方式，则将基类 **Person** 声明为虚基类。

```
class Student: virtual public Person{...}           // Person 为虚基类
class Employee: virtual public Person{...}          // Person 为虚基类
class StuEmployee:public Student,public Employee{...}
```

图 4-11 是类 **StuEmployee** 虚拟继承的示意。通过对公共基类的虚拟继承，派生类只保留了虚基类的一份成员副本，如图 4-12 所示。

现在通过派生对象引用虚基类中的成员就不会产生二义性的命名冲突了。例如：

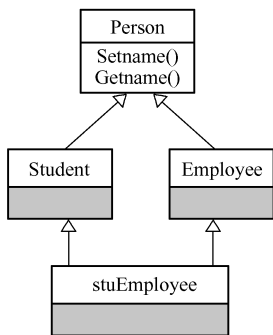


图 4-11 虚拟继承的示意

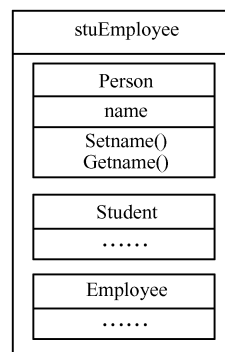


图 4-12 派生类中的虚基类 Person

```
stuEmployee s;
s.SetName();
```

// 正确

2. 虚拟继承的构造次序

在虚拟继承方式下，派生类需要在其构造函数的初始化列表中对虚基类（包括直接虚基类和间接虚基类）进行初始化，以实现虚基类对象的构造。但构造函数的调用次序与非虚拟继承不同，将按以下次序进行：① 先调用虚基类的构造函数，再调用非虚基类的构造函数；② 若同一继承层次中包含多个虚基类，就按照它们被继承的先后次序调用；若某个虚基类的构造函数已被调用，就不再被调用；③ 若虚基类由其他基类派生而来，则先调用虚基类的基类构造函数，再调用虚基类的构造函数；④ 调用派生类的构造函数。

【例 4-18】 虚基类的执行次序分析。

// Eg4-18.cpp

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

class A {
    int a;
public:
    A(){ cout<<"Constructing A"<<endl; }
};

class B {
public:
    B(){ cout<<"Constructing B"<<endl; }
};

class B1:virtual public B, virtual public A {
public:
    B1(int i) { cout<<"Constructing B1"<<endl; }
};

class B2:public A, virtual public B {
public:
    B2(int j) { cout<<"Constructing B2"<<endl; }
};

class D: public B1, public B2 {
public:
    D(int m, int n): B1(m), B2(n) { cout<<"Constructing D"<<endl; }
    A a;
};
```

```
};

void main(){
    D d(1, 2);
}
```

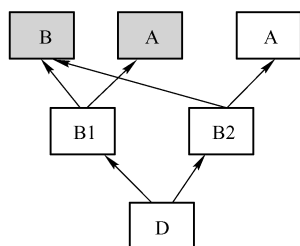


图 4-13 D 的继承结构示意图

程序的运行结果如下：

```
Constructing B
Constructing A
Constructing B1
Constructing A
Constructing B2
Constructing A
Constructing D
```

图 4-13 是派生类 D 的继承结构示意图，左上部分有阴影的 B 和 A 表示虚基类。可以看出，派生类 D 从 B1、B2 派生，由于都不是虚拟派生，所以按继承次序应先构造 B1，再构造 B2。由于 B1 从 B、A 虚拟派生，将依次调用 B 和 A 的构造函数，完成后才能调用 B1 的构造函数，即按 B→A→B1 的次序构造。程序运行结果的前 3 行就是这样得来的。

当 D 的基类 B1 构造完成后，按次序应该构造基类 B2。因为 B2 是从 A、B 派生的，而 B 是虚基类，所以先调用 B 的构造函数。但是，作为虚基类的 B 在构造 D 的虚基类 B1 时就已经被构造了，所以不再构造。接下来构造 B2 的基类 A。因为 A 不是 B2 的虚基类，所以直接调用它的构造函数，与 A 以前是否被构造无关，输出结果的第 4 行就来源于此。如果 A 是 B2 的虚基类，就不会调用 A 的构造函数，也不会有第 4 行输出结果，因为作为虚基类的 A 在构造 B1 时已经被构造了。

B2 的基类构造完成后，就应该调用 B2 的构造函数了，这就是输出结果的第 5 行。

D 的基类构造完成后，接下来应该构造它的成员对象 a，最后构造 D 自己。这就是输出结果的第 6 行和第 7 行。

3. 虚基类由最终派生类初始化

在没有虚拟继承的情况下，每个派生类的构造函数只负责其直接基类的初始化。但在虚拟继承方式下，虚基类由最终派生类的构造函数负责初始化。最终派生类是指在多层次的继承结构中，创建对象时所用的类。由于继承层次结构上的每个类都可能创建对象，因此每个派生类都应该在它的构造函数初始化列表中为虚基类构造函数提供初始化值（不管虚基类是它的直接基类，还是间接基类）。例如，对于例 4-18 而言，如果有如下对象定义：

```
B1 b(1);
D d(2, 3);
```

对象 b 由类 B1 定义，所以 B1 是其虚基类 A、B 的最终派生类，它应该负责 A、B 的构造。对象 d 由派生类 D 创建，所以派生类 D 是虚基类 A、B 的最终派生类，它应该负责 A、B 的构造。

在虚拟继承方式下，若最终派生类的构造函数没有明确调用虚基类的构造函数，编译器就会尝试调用虚基类不需要参数的构造函数（包括无参和缺省参数的构造函数），如果没找到，就会产生编译错误。

在例 4-18 中，最后的派生类 D 的构造函数并没有为虚基类 B 和 A 提供初始化值（没有在类 D 的构造函数初始化列表中调用 B 和 A 的构造函数），程序也没有产生错误，原因是 B 和 A 都有无参构造函数。

【例 4-19】 类 A 是类 B、C 的虚基类，类 ABC 从 B、C 派生，是继承结构中的最终派生类，它必须负责虚基类 A 的初始化。

```
// Eg4-19.cpp
#include <iostream>
using namespace std;

class A {
    int a;
public:
    void f() { cout<<"A"<<endl; }
    A(int x) {
        a = x;
        cout<<"Virtual Bass A ..."<<endl;
    }
};

class B :virtual public A {
public:
    void f() { cout<<"B"<<endl; }
    B(int i):A(i) { cout<<"Virtual Bass B ..."<<endl; }
};

class C :virtual public A {
    int x;
public:
    C(int i):A(i) {
        cout<<"Constructing C ..."<<endl;
        x = i;
    }
};

class ABC :public C, public B {
public:
    ABC(int i, int j, int k):C(i), B(j), A(k) { // L1, 这里必须对 A 进行初始化
        cout<<"Constructing ABC ..."<<endl;
    }
};

void main() {
    ABC obj(1, 2, 3);
    obj.f(); // 调用 B::f()
}
```

程序的运行结果如下：

```
Virtual Bass A ...
Constructing C ...
Virtual Bass B ...
Constructing ABC ...
B
```

虽然 A 是 ABC 的间接基类，但它是虚基类，而且没有缺省构造函数，所以 ABC 必须采用语句 L1 的方式对 A 进行初始化。如果 ABC 还有派生类，则该派生类也必须为 A 提供构造函数初始化值。假如 B 和 C 从 A 采用普通而非虚拟继承的方式派生，则在 L1 中对 A 的构造函数调用是不允许的，会产生编译错误。

4. 成员函数冲突与优先级

与非虚拟的多重继承一样，如果虚基类和派生类中都有同名成员函数，仍然有可能产生命名冲突。如类 A 具有函数 f()，类 B 和 C 都从 A 虚拟派生，类 ABC 继承了类 B 和类 C，见例 4-19，对于类 ABC 的成员函数 f()调用，存在以下情况：

① 如果类 B、C 和 ABC 都没有定义函数 f()，在类 ABC 的对象中只有一个来源于虚基类 A 中的函数 f()，没有冲突。但是，若 B、C 都不是虚拟继承于 A，或者其中一个虚拟继承于 A、另一个非虚拟继承于 A，则在 ABC 对象中的函数 f()有多个，会产生冲突。

② 如果类 B、C 中有一个定义了函数 f()，则在调用 ABC 对象的函数 f()时，B 或 C 中的函数 f()具有优先权。例 4-19 中“obj.f()”的输出证明它调用的是类 B 中的函数 f()。

③ 不论上面哪种情况，如果 ABC 类定义了函数 f()，当 ABC 的对象调用函数 f()时，不会有冲突，ABC 中的函数 f()具有优先权。

4.9 继承和组合

继承和组合（也称为合成）是面向对象程序实现代码重用的两种主要方法。通过继承，派生类可以获得基类程序代码的一份副本，从而达到代码重用的目的。组合体现了类之间的另一种关系，是指一个类可以包含用其他类定义的对象成员。

继承关系常被称为“Is-a”关系，即两个类之间若存在 Is-a 关系，就可以用继承来实现它。例如，水果和梨，水果和苹果，它们就具有 Is-a 关系。因为梨是水果，苹果也是水果，所以梨和苹果都可以从水果继承，获得所有水果都具有的通用特征。

组合常用于描述类之间的“Has-a”关系，即一个类拥有其他类。例如，汽车有发动机、车轮胎、座位等，计算机有 CPU、硬盘、内存条、显示器等，这些都可以用类的组合关系来实现。

【例 4-20】 设计学生选课程序，要求能够管理指定课程的选课学生名单。

(1) 问题分析

可以设计课程类、学生类，分别管理课程信息和学生信息。学生选修课程则与课程和学生都有关系，可以设计选修课程类来管理它。选修时需要知道选修的课程信息和学生名单，这一关系可以用类的包含关系处理，即选修课程类的内部包含课程类和学生类的对象。

(2) 数据抽象

用 Course 表示课程类，包括课程编号、学分和课程名称，分别用数据成员 courseName、cno 和 credit 表示，并设计对应的 setter 写入器和 getter 读取器来设置和读取数据成员的值，设计成员函数 setCourse()和 display()一次性设置和显示全体数据成员，并设计默认构造函数。

用 Student 表示学生类，它有学号和姓名，分别用数据成员 sno 和 stuName 表示，并为各数据成员设置对应的写入器和读取器，以及显示成员值的成员函数 display()。

选修课程类是本题的核心类,用 `SelectCourse` 表示,用数据成员 `course`、`maxNum`、`curNum` 分别表示选学的课程、最多允许选课的人数、实际选课人数,用指针成员 `stu` 存取选课学生名单,它是指向 `Student` 类型的动态数组,以便更好地实现对选课学生人数的动态管理。为这些数据成员设计对应的 `setter` 写入器和 `getter` 读取器,以及能够随时添加和打印选课学生信息的成员函数 `appendStudent()` 和 `display()`。

`SelectCourse` 通过对象成员 `course` 和指针成员 `stu` 将课程类和学生类组织成了一个整体,形成了图 4-14 所示的 UML 关系图。

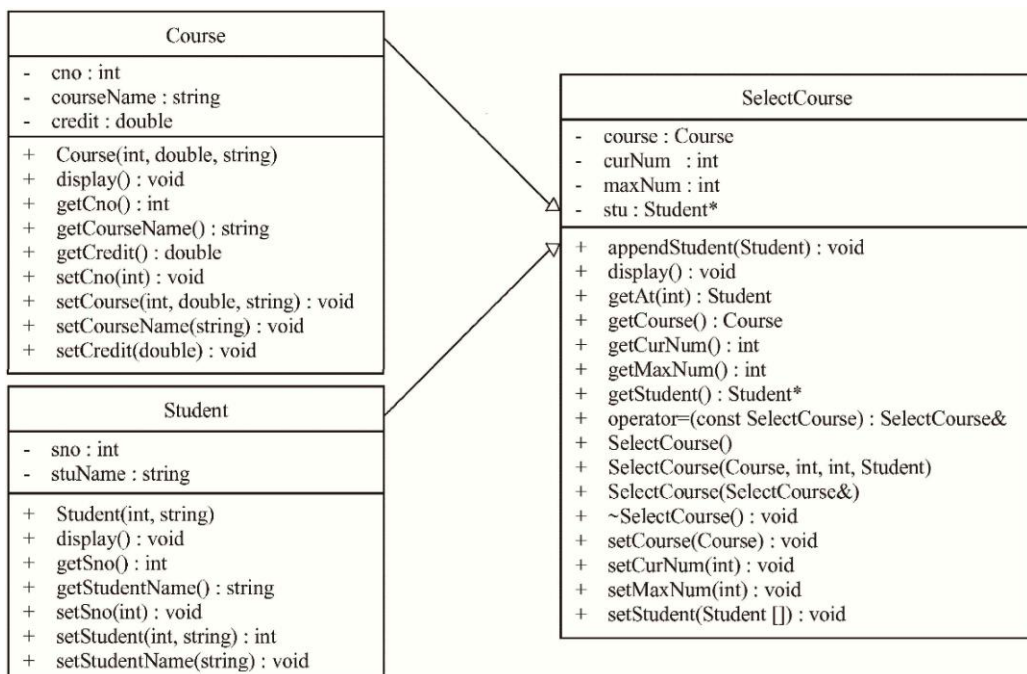


图 4-14 选课系统的 UML 关系图

在各类的主要功能设计完成后,必须对各类的构造函数、析构函数、复制赋值函数、移动复制函数、移动赋值运算符函数进行评估与设计(见 3.6.1 节)。在本例中,由于 `Course` 和 `Student` 都没有特殊的处理要求,也没有指针类型的数据成员,所以不必设计这 6 个特殊函数,由系统默认生成就行了。但类 `SelectCourse` 不同,因为它有 `stu` 指针成员,由编译器生成的默认复制构造函数、赋值运算符函数已不能够正确处理 `stu` 的复制和赋值了,它们会引发“指针悬挂”问题,因此必须重新定义它们。按照 `RAII`(见 3.7.1 节)方法,即在类 `SelectCourse` 的构造函数中为 `stu` 分配内存空间,在析构函数中回收 `stu` 的内存空间。

```

// Eg4-20.cpp
#include <iostream>
#include <string>
using namespace std;

// 课程类 Course 的代码如下,编译器会为该类生成合成的复制构造函数和赋值运算符函数
class Course {
public:
    void setCno(int cNumber) { cno = cNumber; }
    void setCredit(double crd) { credit = cno; }

```

```

void setCourseName(string cname) { courseName = cname; }
int getCno() { return cno; }
double getCredit() { return credit; }
string getCourseName(){ return courseName; }
Course(int Cno = 0, double cre = 0, string cName = "") { setCourse(Cno, cre, cName); }
void display() {
    cout<<"课程号: "<<cno<<"\t 课程名称: "<<courseName<<"\t 学分: " <<credit<<endl;
}
void setCourse(int Cno = 0, double cre = 0, string cName = "") {
    cno = Cno;
    credit = cre;
    courseName = cName;
}
private:
    int cno;
    double credit;
    string courseName;
};

```

// 下面是学生类 **student** 的程序代码，编译器会为该类生成默认的复制构造函数和赋值运算符函数

```

class Student {
public:
    void setSno(int Snumber) { sno = Snumber; }
    void setStudentName(string Sname) { stuName = Sname; }
    int getSno() { return sno; }
    string getStudentName() { return stuName; }
    Student(int Sno = 0, string SName = "") { setStudent(Sno, SName); }
    void display() { cout<<"学号: "<<sno <<"\t 姓名: "<<stuName<<endl; }
    void setStudent(int Sno = 0, string Sname = "") { sno = Sno; stuName = Sname; }
private:
    int sno;
    string stuName;
};

```

/* 下面是选修课程类 **SelectCourse** 的类代码，由于默认复制构造函数和默认赋值运算符函数不能够正确完成指针成员 **stu** 的复制，因此必须显式定义复制构造函数和赋值运算符函数，否则程序运行时会产生“指针悬挂”错误！

*/

```

class SelectCourse {
public:
    SelectCourse() { stu = new Student[maxNum]; }
    SelectCourse(Course c, int mNum, int cNum, Student s[]):course(c), maxNum(mNum),
        curNum(cNum), stu(new Student[maxNum]) {
        for (int i = 0; i < cNum; i++)
            stu[i] = s[i];
    }
    ~SelectCourse() { delete []stu; }
    SelectCourse(const SelectCourse &o):course(o.course), maxNum(o.maxNum), curNum(o.curNum) {
        stu = new Student[o.maxNum];
        for (int i = 0; i < o.curNum; i++)
            stu[i] = o.stu[i];
    }
    SelectCourse& operator = (const SelectCourse o) {

```

```

        course = o.course;
        maxNum = o.maxNum;
        curNum = o.curNum;
        delete []stu;
        stu = new Student[maxNum];
        for (int i = 0; i < o.curNum; i++)
            stu[i] = o.stu[i];
        return *this;
    }

    void setCourse(Course c) { course = c; }
    void setMaxNum(int n) { maxNum = n; }
    void setCurNum(int n) { curNum = n; }
    int getMaxNum() { return maxNum; }
    int getCurNum() { return curNum; }
    Course getCourse(){ return course; }
    Student* getStudent() { return stu; }
    void setStudent(Student s[]) { stu = s; }
    Student getAt(int n) { return stu[n]; }
    void appenStudent(Student s) {
        if(curNum < maxNum)
            stu[curNum++] = s;
    }
    void display() {
        course.display();
        cout<<"实多选课人数: "<<maxNum<<"\t 实选人数: "<<curNum<<endl;
        cout<<"选课学生名单: "<<endl;
        for(int i = 0; i < curNum; i++)
            stu[i].display();
    }
private:
    int maxNum = 10, curNum = 0;
    Course course;
    Student *stu = nullptr;
};

// 测试各类设计效果的 main()函数, 因篇幅所限, 并未对各类的每个函数进行测试
void main() {
    // 测试 SelectCourse 类构造函数和显示函数的运行情况
    Course course;
    course.setCourse(101, 3.5, "C++面向对象程序设计");
    Student s[2], s1;
    s[0].setStudent(10, "高大山");
    s[1].setStudent(11, "李明育");
    SelectCourse sc(course, 10, 2, s);
    cout<<"-----sc-----"<<endl;
    sc.display();
    // 测试 SelectCourse 类的复制构造函数和添加选课学生函数的运行情况
    SelectCourse sc2, sc1 = sc;
    s1.setStudent(14, "黄始仁");
    sc1.appenStudent(s1);
    cout<<"-----sc1(sc)-----"<<endl;
    sc1.display();
}

```



```

// 测试 SelectCourse 类的赋值运算符函数的运行情况
sc2 = sc1;
cout<<"-----sc2 = sc1-----"<<endl;
sc2.display();
// 测试 SelectCourse 类中获取学生名单和人数的成员函数的运行情况
Student *sname = sc2.getStudent();
cout <<"-----sc2.getStudent()-----"<<endl;
for(int i = 0; i < sc2.getCurNum(); i++)
    (sname++)->display();
}

```

程序运行结果如下：

```

-----sc-----
课程号: 101    课程名称: C++面向对象程序设计    学分: 3.5
实多选课人数: 10    实选人数: 2
选课学生名单:
学号: 10    姓名: 高大山
学号: 11    姓名: 李明育

-----sc1(sc)-----
课程号: 101    课程名称: C++面向对象程序设计    学分: 3.5
实多选课人数: 10    实选人数: 3
选课学生名单:
学号: 10    姓名: 高大山
学号: 11    姓名: 李明育
学号: 14    姓名: 黄始仁

-----sc2 = sc1-----
课程号: 101    课程名称: C++面向对象程序设计    学分: 3.5
实多选课人数: 10    实选人数: 3
选课学生名单:
学号: 10    姓名: 高大山
学号: 11    姓名: 李明育
学号: 14    姓名: 黄始仁

-----sc2.getStudent()-----
学号: 10    姓名: 高大山
学号: 11    姓名: 李明育
学号: 14    姓名: 黄始仁

```

组合和继承是面向对象程序进行类设计和构建大型复杂应用程序的两种基本方法，本例的类 `SelectCourse` 的设计示范了使用组合方法把多个类组装成功能强大的复杂类的方法，具有典型代表意义。请结合程序中的注释，理解掌握组合类的设计方法，特别是什么情况下必须设计类的构造函数、复制构造函数和赋值运算符函数。

4.10 节将以一个具体的编程实例介绍使用继承技术设计功能强大的复杂类的方法。

4.10 编程实作：继承编程应用

继承是面向对象程序设计的重要特性，通过继承能够设计出可重用性高、可扩展性好的应用程序，它在现代软件开发中的应用极其广泛。下面以一个简易的专业课程体系管理的继

承类设计为例，介绍使用继承技术进行软件开发的方法和过程。

【例 4-21】 某校每位学生都要学习英语、语文、数学三门公共课程以及不同的专业课程。会计学专业要学习会计学 and 经济学两门课程，化学专业要学习有机化学和化学分析两门课程。编程序管理学生成绩，计算公共课的总分和平均分，以及所有课程的总成绩。

(1) 问题分析

在这个问题中，由于英语、语文、数学三门公共课程是所有学生都要学习的，可以将它们抽象成一个基类 **comFinal**，管理这三门基础课程的成绩。两个专业的课程则分别抽象成类 **Account** 和 **Chemistry**，分别管理会计学和化学两个专业的课程成绩。由于在整个问题中还涉及学生，因此应该抽象出学生类 **Student** 来管理学生的档案。为简化问题，此处忽略了学生类的设计，仅用一个姓名代表学生，并将此名字作为类 **comFinal** 的一个数据成员。

(2) 数据抽象

用 **comFinal** 表示公共基类，用字符数组 **name** 表示参与课程学习的学生姓名，用 **english**、**chinese**、**math** 分别表示英语、语文和数学成绩，并为每个数据成员设计成员函数 **set()**、**get()**，以修改、读取其值，以及成员函数 **getTotal()**、**getAverage()**、**show()**，分别计算总分和平均分、输出学生的各科成绩。设计构造函数 **ComFinal()** 为各数据成员提供初始化值。由于设计了具有参数的构造函数，编译器不会再为本类生成默认构造函数，为了便于类 **ComFinal** 的继承及构造，以及本类无参对象和数组的定义，重定义了它的默认构造函数。

用 **Account** 表示会计学专业类，从 **ComFinal** 派生，具有会计学 and 经济学两门课程，分别用数据成员 **account** 和 **econ** 表示，设计设置/读取数据值的成员函数 **set()**、**get()**，以及成员函数 **getMajTotal()**、**getMajAvg()** 和 **show()**，分别计算两门课程的总分、平均分，以及显示输出学生的各项成绩数据。

用 **Chemistry** 表示化学专业类，有化学分析和化学两门课程，分别用数据成员 **analy** 和 **chemistry** 表示，按照与类 **Account** 相同的方法设计类 **Chemistry**。类继承结构如图 4-15 所示。

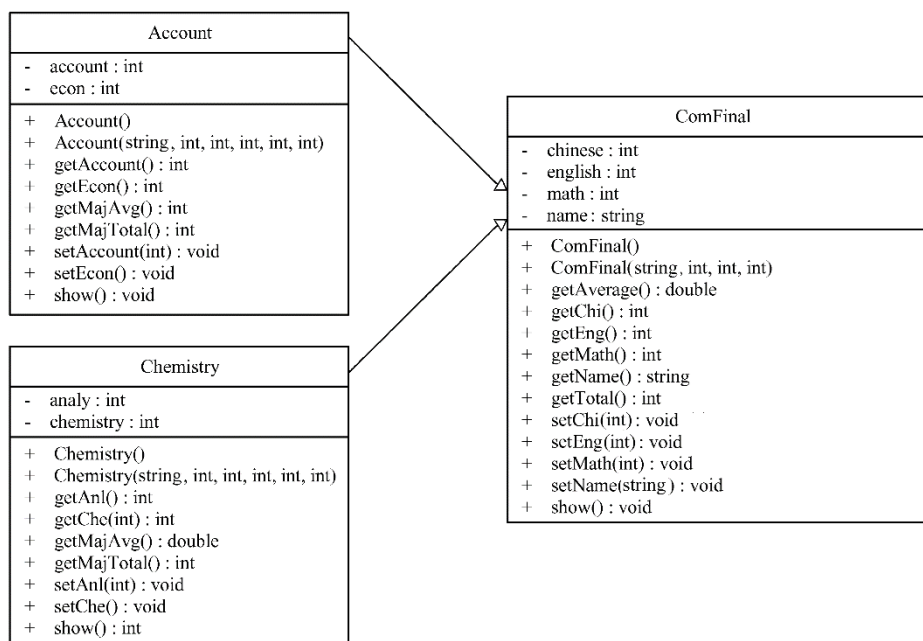


图 4-15 类继承结构

三个类都没有指针成员，也没有在构造函数中为任何数据成员分配动态存储空间，不需要在析构函数中进行动态存储空间的回收。因此，在上面的数据抽象过程中没有为各类定义复制构造函数、赋值运算符函数和析构函数。在没有定义这些函数的情况下，编译器会为它们生成对应的默认函数，这些默认函数能够正确完成对象的构造、复制、赋值和析构。

为了实现类的重用，将三个类的声明和实现分别保存在不同的头文件和源程序文件中。为此，可以先建立各类的头文件和源程序，将它们保存在 C:\course 目录下的项目子文件中。操作过程如下：

1. 在项目中添加各类的头文件和源码文件

<1> 建立目录 C:\course，用于保存本程序中的所有文件。然后启动 Visual C++ 2020，选择“新建 | 项目 | 空项目 | C++”菜单命令，在弹出的“新建项目”对话框中指定“位置”为“C:\course”，在“名称”文本框中输入“com_main”，单击“确定”按钮。

<2> 在“解决方案资源管理器”中单击右键，然后在弹出的快捷菜单中选择“头文件 | 添加 | 新建项”，在弹出的对话框中选中“头文件”，并在“名称”文本框中输入“ComFinal.h”。按照同样的方法，将 Account.h、Chemistry.h 添加到项目中。

<3> 按照与添加头文件相同的方法，将各类的源文件 ComFinal.cpp、Account.cpp、Chemistry.cpp 添加到项目中。到目前为止，这些头文件和源文件都是空文件。

2. 编写各类头文件和源文件的程序代码

1) 建立 ComFinal 类

(1) 在 comFinal.h 头文件中输入如下内容：

```
// comFinal.h
#ifndef comFinal_h
#define comFinal_h
#include<string>
using std::string;

class comFinal {
protected:
    string name; // 学生姓名
    int english, chinese, math; // 公共课成绩及总分
public:
    comFinal(string n, int Eng, int Chi, int Mat);
    comFinal() {};
    string getName() { return name; }
    int getEng() { return english; }
    int getChi() { return chinese; }
    int getMat() { return math; }
    void setEng(int x) { english = x; }
    void setChi(int x) { chinese = x; }
    void setMat(int x) { math = x; }
    int getTotal() { return english + chinese + math; }
    double getAverage() { return (english + chinese + math) / 3; }
    void show(); // 显示学生各公共课的成绩、平均分和总分
};
```

```
#endif
```

(2) 在 ComFinal.cpp 源文件中输入如下内容:

```
// ComFinal.cpp
#include <iostream>
#include "comFinal.h"
using namespace std;

comFinal::comFinal(string n, int Eng, int Chi, int Mat) {
    name = n;    english = Eng;    chinese = Chi;    math = Mat;
}

void comFinal::show() {
    cout<<"学生姓名: "<<getName()<<endl;
    cout<<"英语成绩: "<<getEng()<<endl;
    cout<<"语文成绩: "<<getChi()<<endl;
    cout<<"数学成绩: "<<getMatt()<<endl;
    cout<<"基础课总分: "<<getTotal()<<endl;
    cout<<"基础课平均成绩: "<<getAverage()<<endl<<endl;
}
```

2) 建立 Account 类

(1) 在 Account.h 头文件中输入如下内容:

```
// Account.h
#include "comFinal.h"
#ifndef Account_h
#define Account_h

class Account :public comFinal {
protected:
    int account;                // 会计学成绩
    int econ;                   // 经济学成绩
public:
    Account(string n, int Eng, int Chi, int Mat, int Acc, int Eco);
    Account() {}
    int getMajTotal() { return econ + account; }
    float getMajAve() { return float((account + econ) / 2); }
    int getAccount() { return account; };
    int getEcon() { return account; }
    void setAccount(int x) { account = x; }
    void setEcon(int x) { econ = x; }
    void show();
};
#endif
```

(2) 在 Account.cpp 源文件中输入如下内容:

```
// Account.cpp
#include "account.h"
#include<iostream>
using namespace std;
```

```

Account::Account(string n, int Eng, int Chi, int Mat, int Acc,
                int Eco) : comFinal(n, Eng, Chi, Mat) {
    econ = Eco;
    account = Acc;
}
void Account::show() {
    comFinal::show();
    cout<<"会计学成绩: "<<account<<endl;
    cout<<"经济学成绩: "<<econ<<endl;
    cout<<"总分: "<<getTotal() + account + econ<<endl;
}

```

3) 建立 Chemistry 类

(1) 在 Chemistry.h 头文件中输入如下内容:

```

// Chemistry.h
#include "comFinal.h"
#ifndef chemistry_h
#define chemistry_h

class Chemistry :public comFinal {
protected:
    int chemistry;           // 化学成绩
    int analy;               // 化学分析成绩
public:
    Chemistry(string n, int Eng, int Chi, int Mat, int Chem, int Anal);
    Chemistry() { };
    int getMajTotal() { return analy + chemistry; }
    float getMajAve() { return float((chemistry + analy) / 2); }
    int getChe() { return chemistry; }
    int getAnl() { return analy; }
    void setChe(int x) { chemistry = x; }
    void setAnl(int x) { analy = x; }
    void show();
};
#endif

```

(2) 在 Chemistry.cpp 源文件中输入如下内容:

```

// Chemistry.cpp
#include<iostream>
#include"Chemistry.h"
using namespace std;

Chemistry::Chemistry(string n, int Eng, int Chi, int Mat, int Chem, int Anal)
                                                    :comFinal(n, Eng, Chi, Mat) {

    chemistry = Chem;
    analy = Anal;
}
void Chemistry::show() {
    comFinal::show();
    cout<<"有机化学: "<<chemistry<<endl;
}

```

```
    cout<<"化学分析: "<<analy<<endl;
    cout<<"总分: "<<getTotal() + chemistry + analy<<endl;
}
```

4) 建立主程序并运行程序

在 com_main.cpp 中输入下面的程序代码:

```
// com_main.cpp
#include "Chemistry.h"
#include "Account.h"
#include <iostream>
using namespace std;

void main() {
    Account a1("张三星", 98, 78, 97, 67, 87);
    Chemistry c1("光红顺", 89, 76, 34, 56, 78);
    a1.show();
    cout<<"-----"<<endl;
    c1.setAnl(100);
    c1.show();
}
```

编译并运行该程序, 输出结果如下:

```
学生姓名: 张三星
英语成绩: 98
语文成绩: 78
数学成绩: 97
基础课总分: 273
基础课平均成绩: 91
```

```
会计学成绩: 67
经济学成绩: 87
总分: 427
```

```
-----
学生姓名: 光红顺
英语成绩: 89
语文成绩: 76
数学成绩: 34
基础课总分: 199
基础课平均成绩: 66
```

```
有机化学: 56
化学分析: 100
总分: 355
```

习 题 4

- 4.1 简述继承、基类、派生类的概念, 以及基类与派生类的关系。
- 4.2 简述分别在 private、public 和 protected 继承方式下, 基类与派生类成员之间的关系。

4.3 在哪些情况下，派生类即使没有成员需要初始化也必须定义构造函数？

4.4 简述虚拟继承的概念，C++引入虚拟继承的原因。

4.5 假设有类 A、B、C、D，类 A 是类 B 的基类，类 B 是类 D 的基类，类 B 有用类 C 创建的一个对象成员，若定义类 D 的对象，分析各类的构造函数和析构函数的调用次序。

4.6 对于题 4.5，假设类 C 也从类 A 派生，即类 A 同为 B 和 C 的基类，其余题意同题 4.5，分析定义类 D 的对象时，各类的构造函数和析构函数的调用次序。

4.7 对于题 4.6，假设类 B 和类 C 都虚拟继承自类 A，分析定义类 D 的对象时，各类构造函数和析构函数的调用次序。

4.8 指出下面程序中的错误。

```
#include <iostream>

class A {
    int x;
    A(int a) { x = a; }
public:
    setA(int y) { x = y; }
};

class B final:private A {
public:
    B(){ cout<<"B"<<endl; }
};

class D:public B {
public:
    void setA(int a){ A::setA(a); }
}

void main() {
    A a1(2), a2;
    A a3 = a1;
    B b;
    b.setA(3);
}
```

4.9 读程序，分析程序的运行结果。

(1)

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

class A {
public:
    A(int a, int b):x(a), y(b) { cout<<"A constructor ..."<<endl; }
    void Add(int a, int b) { x += a; y += b; }
    void display() { cout<<"("<<x<<", "<<y<<")"; }
    ~A() { cout<<"destructor A ..."<<endl; }
private:
    int x, y;
};

class B:private A {
private:
```



```

    int i, j;
    A Aobj;
public:
    B(int a, int b, int c, int d):A(a, b), i(c), j(d), Aobj(1, 1) {
        cout<<"B constructor ..."<<endl;
    }
    void Add(int x1, int y1, int x2, int y2) {
        A::Add(x1, y1);
        i += x2;
        j += y2;
    }
    void display() {
        A::display();
        Aobj.display();
        cout<<"("<<i<<", "<<j<<")"<<endl;
    }
    ~B() { cout<<"destructor B ..."<<endl; }
};

void main() {
    B b(1, 2, 3, 4);
    b.display();
    b.Add(1, 3, 5, 7);
    b.display();
}

```

(2)

```

#include <iostream>
using namespace std;

class B {
protected:
    void f1(int a, int b) { cout<<a + b<<endl; }
    void f2(int a) { cout<<a<<endl; }
};

class D : public B {
public:
    using B::f1;
    void f1(char * d) { cout<<d<<endl; }
};

void main() {
    D d;
    d.f1(3, 5);
    d.f1((char*)"Hello using!");
}

```

(3)

```

#include <iostream>
using namespace std;

class A {
    int x, y;

```

```

    public:
        A(int a = 0, int b = 0) :x(a), y(b) { cout<<"a = "<<a<<"\tb = "<<b<<endl; }
};
class B :public A {
    public:
        using A::A;
};

void main() {
    B b, b1(10), b2(20), b3(30, 30);
}

```

(4)

```

#include <iostream>
using namespace std;

class A {
    int x;
    public:
        A(int a = 0, int b = 2) :x(a) { }
        A &operator = (A& o) {
            x = o.x;
            cout<<"In A = (A&), x = "<<x<<endl;
            return *this;
        }
        int getX() { return x; }
        A& operator = (A &&o) = default;
};

class B :public A {
    int y;
    public:
        B(int a = 0, int b = 0):A(a), y(b) { }
        B& operator = (B& o) {
            A::operator = (o);
            cout<<"In B = (B&), x = "<<getX()<<"\ty = "<<y<<endl;
            return *this;
        }
        B& operator = (B&& o) {
            A::operator = (std::move(o));
            cout<<"In B = (B&&), x = "<<getX()<<"\ty = "<<y<<endl;
            return *this;
        }
};

void main() {
    B b, b1(1, 2);
    b = b1;
    b1 = std::move(b);
}

```

(5)

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

class A {
public:
    A(int a):x(a) { cout<<"A constructor ..."<<x<<endl; }
    int f() { return ++x; }
    ~A() { cout<<"destructor A ..."<<endl; }
private:
    int x;
};

class B:public virtual A {
private:
    int y;
    A Aobj;
public:
    B(int a, int b, int c):A(a), y(c), Aobj(c) { cout<<"B constructor ..."<<y<<endl; }
    int f() {
        A::f();
        Aobj.f();
        return ++y;
    }
    void display() { cout<<A::f()<<"\t"<<Aobj.f()<<"\t"<<f()<<endl; }
    ~B() { cout<<"destructor B ..."<<endl; }
};

class C:public B {
public:
    C(int a, int b, int c):B(a, b, c), A(0) { cout<<"C constructor ..."<<endl; }
};

class D:public C, public virtual A {
public:
    D(int a, int b, int c):C(a, b, c), A(c){ cout<<"D constructor ..."<<endl; }
    ~D(){ cout<<"destructor D ...."<<endl; }
};

void main() {
    D d(7, 8, 9);
    d.f();
    d.display();
}
```

4.10 某出版社发行图书和光盘，利用继承设计管理出版物的类。

要求如下：建立一个基类 **Publication** 存储出版物的标题 **title**、出版物名称 **name**、单价 **price** 及出版日期 **date**；用类 **Book** 和 **CD** 分别管理图书和光盘，它们都从类 **Publication** 派生；类 **Book** 具有保存图书页数的数据成员 **page**，类 **CD** 具有保存播放时间的数据成员 **playtime**；每个类都有构造函数、析构函数，且都有用于从键盘获取数据的成员函数 **inputData()**和用于显示数据的成员函数 **display()**。

4.11 一个教学系统至少有学生和教师两类人员，假设教师的数据有教师编号、姓名、年龄、

性别、职称和系别，学生的数据有学号、姓名、年龄、性别、班级和语文、数学、英语三门课程的成绩。编程完成学生和教师数据的输入和显示。

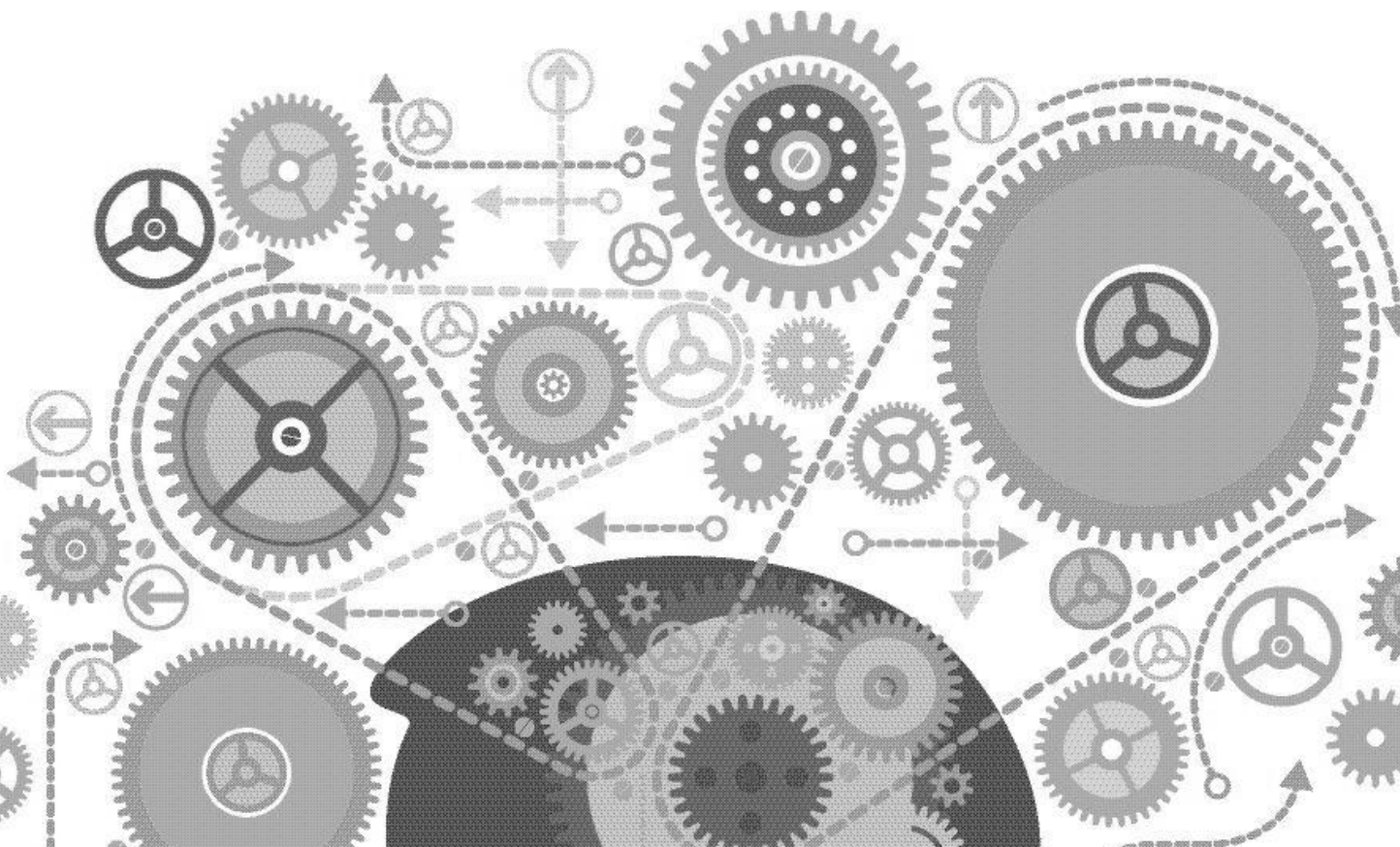
要求如下：设计三个类 **Person**、**Teacher**、**Student**；类 **Person** 是 **Teacher** 和 **Student** 的基类，具有此二类共有的数据成员姓名、年龄、性别，并具有输入和显示这些数据的成员函数；类 **Teacher** 继承了类 **Person** 的功能，并增加对教师编号、职称、系别等数据成员进行输入和显示的成员函数；按同样的方法完善类 **Student** 的设计。

第 5 章

多 态

多态是面向对象程序设计语言的又一重要特征，是指不同对象收到同一消息时会产生不同的行为。继承处理的是类与类之间的层次关系问题，而多态处理的是类的层次结构之间以及同一个类内部同名函数之间的关系问题。简单地说，多态就是在同一个类内部，或者继承体系结构的基类与派生类中，用同名函数来实现不同的功能。

本章介绍多态的原理与实现方法，包括虚函数、纯虚函数和抽象类等知识。



5.1 多态概述

5.1.1 多态的概念

多态 (polymorphism) 是指不同对象收到相同消息时会执行不同的操作, 通俗地讲, 就是用一个相同的名称定义多个不同的函数, 这些函数可以针对不同数据类型实现相同或相似的功能, 即所谓的“一个接口、多种实现”。例如:

```
int add(int x, int y) { return x+y; }
double add(double x, double y) { return x+y; }
float add(float x, float y) { return x+y; }
.....
void main() {
    cout<<add(3, 4);
    cout<<add(4.5, 8);
}
```

这里的“add(x, y)”是一个接口, 定义了计算两数和的功能, 但它有多种实现, 可以分别实现计算整数、双精度数、浮点数等的两数和的功能。对于使用者而言, 它只需知道 add() 函数的功能是计算两数和, 把要求求的两个参数传给 add() 即可, 并不需要知道 add() 有多少个函数版本, 也不需要了解这些函数是如何实现的。因此, 从这个意义上, 多态简化了程序设计的复杂性, 减轻了程序员的负担。

广义上, 面向对象程序设计语言的多态有 3 种表现形式: ① 重载多态, 包括函数重载多态和运算符重载多态, 上面的 add() 函数就是函数重载多态的例子; ② 模板多态, 通过一个模板生成不同的函数或类 (第 7 章介绍); ③ 继承多态, 通过基类对象的指针 (引用), 调用不同派生类对象的同名成员函数, 实现不同的程序功能。

若无特别说明, 面向对象程序设计的多态通常是指第 3 种情况。实现这种多态, 要具备以下 3 个必要条件: ① 有继承; ② 派生类要覆盖 (重定义) 基类的虚函数, 即派生类具有与基类同名虚函数原型完全相同的虚成员函数; ③ 把基类的指针或引用绑定到派生类对象上。下面的例子说明多态的基本概念。

【例 5-1】 设计一个管理动物声音的软件。

问题分析与数据抽象: 所有的动物都会发声, 但是当没有说明是猫、狗或鸟等具体动物时, 则无法说清楚它发出的是什么声音。虽然无法确定是什么声音, 但确实知道动物有声音, 面向对象程序设计语言提出了虚函数来表达这类确实存在但无法通过代码实现的抽象概念, 等到了可知的具体动物时, 它会发出什么声音就是明确的了, 此时再对相应的虚函数进行编码实现。

为此, 可以用 Animal 表示动物类, 用虚成员函数 sound() 表示动物会发声这一行为。Dog、Cat、Wolf、Bird 是具体的动物, 它们可以继承 Animal 的所有特征和行为。但是, 每类动物能够发出什么声音是明确的, 而且各不相同, 需要覆盖 (重定义) 从 Animal 继承来的成员函数 sound()。Animal 和 Dog 等动物的继承关系形成了图 5-1 所示的继承结构。

据此继承结构, 可以设计出如下简易类:

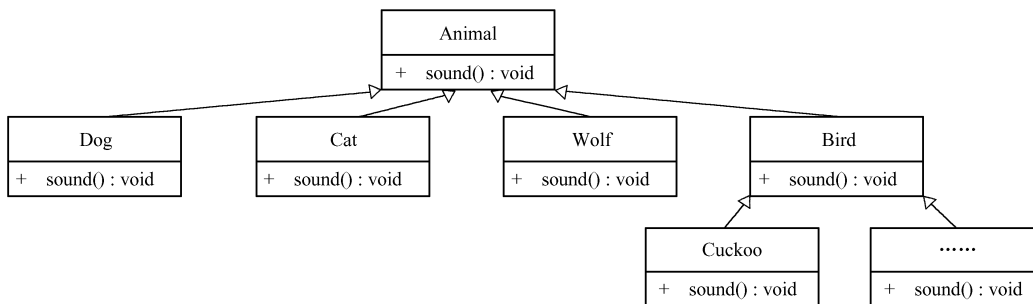


图 5-1 Animal 和 Dog 等类的继承结构

```

class Animal {
public:
    virtual void sound() { cout<<"unknow!"<<endl; }
};
class Dog :public Animal {
public:
    void sound() { cout<<"wang, wang, wang!"<<endl; }
};
class Cat :public Animal {
public:
    void sound() { cout<<"miao, miao, miao!"<<endl; }
};
class Wolf :public Animal {
public:
    void sound() { cout<<"wu, wu, wu!"<<endl; }
};

```

多态是指当基类的指针（或引用）绑定到派生类对象上，通过此指针（引用）调用基类的成员函数时，实际上调用到的是该函数在派生类中定义的覆盖函数版本。例如，对于上面的继承结构，下面的 pA 指针实现的就是多态。

```

void main() {
    Animal *pA;
    Dog dog;
    Cat cat;
    Wolf wolf;
    pA = &dog;    pA->sound();    // pA 绑定到 Dog 的对象，调用 Dog 的 sound() 函数
    pA = &cat;    pA->sound();    // pA 绑定到 Cat 的对象，调用 Cat 的 sound() 函数
    pA = &wolf;   pA->sound();    // pA 绑定到 Wolf 的对象，调用 Wolf 的 sound() 函数
}

```

pA 是基类 Animal 的指针，当它指向派生类对象来调用基类的成员函数 sound() 时，将调用实际所指对象对应的类类型中的成员函数 sound()。例如，pA 指向 Cat 类的对象时，pA->sound() 调用 Cat 中的成员函数 sound()；当 pA 指向 Wolf 类的对象时，pA->sound() 调用 Wolf 类中定义的成员函数 sound()。

除了指针，把基类的引用绑定到派生类对象上时，也能够实现多态。更一般地，在面向对象程序设计中，多态更多地体现在用基类对象的指针或引用作为函数的参数，通过它调用派生类对象中的覆盖函数版本。例如，针对 Animal 继承结构，要设计管理动物声音的函数

`animalSound()`，以便管理每种动物的声音，多态能够很好地实现此需求。

```
void animalSound(Animal &animal) { animal.sound(); }
```

函数 `animalSound()` 体现了“一个接口，多种实现”，即以基类 `Animal` 的引用为接口，可以访问图 5-1 所示继承结构中 `Animal` 类的多个派生类对象的成员函数 `sound()`。

```
animalSound(dog);           // 调用 Dog::sound()
animalSound(cat);           // 调用 Cat::sound()
animalSound(Wolf);          // 调用 Wolf::sound()
```

`animalSound()` 函数通过基类 `Animal` 的引用或指针，能够访问从 `Animal` 类派生出的每种具体动物类实现的 `sound()` 函数版本，这就是多态。

5.1.2 多态的意义

多态是继数据抽象与封装、继承后，面向对象程序设计语言的第三个基本特征。通过多态，基类可以表达“做什么”的设计思想，派生类则体现“怎么做”，从另一角度将接口与实现分离开来，基类体现了接口，派生类则体现了实现。多态对于软件开发和维护而言意义重大，使开发者在没有确定某些具体功能如何实现的情况下，能够站在高层（基类）设计并完成系统开发，待功能明确并实现后，通过多态可以很容易地融入系统。

概括而言，多态具有以下优点。

① 可替换性。多态对已存在代码具有可替换性。例如，在 `Animal` 继承结构中，如果现有的类 `Dog` 重新编写了成员函数 `sound()`，需要更新，只要该函数的原型保持不变，不用修改原系统中函数 `animalSound()` 的任何代码，就能够调用到类 `Dog` 新编写的函数成员 `sound()`。也就是说，用新编写的类 `Dog` 更换以前的版本，原系统不受影响就能够调用新类的功能。软件升级变得简单易行。

② 可扩展性。多态对代码具有可扩展性。增加新的子类不影响已存在类的多态性、继承性，以及其他特性的运行和操作。也就是说，在不影响原系统功能的情况下，很容易派生新类，扩展系统新功能。

例如，上面的动物声音管理软件中并没有涉及鸟类的声音管理，现在要扩展系统的功能，使它能够管理鸟类的声音。多态容易实现类似于此的系统功能扩展，只需从类 `Animal` 派生类 `Bird`，再由类 `Bird` 派生各种鸟类，如布谷鸟（`Cuckoo`），并覆盖类 `Animal` 的成员函数 `sound()`，见图 5-1。现在，只需将类 `Bird` 或 `Cuckoo` 的对象传递给函数 `animalSound()`，就能够自动调用这些类的成员函数 `sound()`，管理鸟类的声音。例如：

```
Bird bird;
Cuckoo cuckoo;
animalSound(bird);           // 调用 Bird::sound()
animalSound(cuckoo);         // 调用 Cuckoo::sound()
```

函数 `ainmalSound()` 不作任何修改，就扩展了管理鸟类声音的功能。由此可见，通过多态扩展软件功能非常方便。

③ 灵活性。在多态程序结构中，基类通过虚函数，向派生类提供了一个共同接口，派生类只要覆盖了基类的虚函数，基类指针（引用）就能容易地调用派生类实现的虚函数版本。从这种意义上，基类提供接口，派生类提供实现，两者分离，使软件功能的整体设计和功能的逐步实现、扩展更加灵活。

5.1.3 多态和绑定

多态与绑定密切相关。源程序需要经过编译、连接后才能够形成可执行文件，在这个过程中必须把调用函数名与对应函数关联在一起，这个过程就是**绑定 (binding)**，又称为**联编**。

绑定分为静态绑定和动态绑定。

静态绑定，即静态联编，是指在编译程序时根据调用函数提供的信息，把它对应的具体函数确定下来，即在编译时就把调用函数名与具体函数绑定在一起。

动态绑定，又称为动态联编，是指在编译程序时还没有足够的信息能确定函数调用所对应的具体函数，只有在程序运行过程中，执行函数调用时，才能取得对应的类型信息，把调用函数名与具体函数绑定在一起。

静态绑定和动态绑定都能够实现多态。采用静态绑定实现的多态称为**静态多态**，前面介绍的函数重载和第 6 章将介绍的运算符重载都具有静态多态。采用动态绑定实现的多态称为**动态多态**，是通过继承，在程序运行时才确定虚函数的实现代码。通常，面向对象程序设计的多态是指运行时的多态。

静态多态在编译时就确定了函数调用的具体函数，不需要在执行程序时从多个同名函数中匹配调用函数，所以执行速度快。动态多态需要在执行程序时从多个同名函数中匹配调用函数，所以比静态多态的执行效率低，但提供了更大的灵活性、更高的问题抽象性和程序的可维护性。

5.2 虚函数

5.2.1 虚函数的意义

虚函数是运行时多态的基础，是通过动态绑定实现其函数调用的，允许函数调用与函数体之间的绑定关系在程序运行时才确定，即在程序运行时才确定调用函数的功能。

第 4 章曾经介绍过基类与派生类的对象之间具有如下赋值相容关系：派生类对象可以赋值给基类对象，派生类对象的地址可以赋值给用基类定义的指针，派生类对象可以作为基类对象的引用。但不论哪种赋值方式，都只能通过基类对象（或基类的指针与引用）访问到派生类对象从基类中继承到的成员，可这并不总是需要的。

【例 5-2】 某公司有经理、销售员、小时工等多类人员。经理按周计算工资；销售员每月底薪 800 元，然后加销售提成，每销售一件产品提取销售利润的 5%；小时工按小时计算工资。每类人员都有姓名和身份证号等信息，设计管理员工工资的程序。

问题分析和数据抽象：经理、销售员、小时工等各类工作人员都是公司的雇员，每类人员都有姓名和身份证号等信息，可以将它们抽象为雇员类 **Employee**，用 **name** 和 **id** 分别表示姓名和身份证号。表示其他类型人员的类则从类 **Employee** 派生。

将经理抽象成类 **Manager**，用 **weeklySalary** 表示他的周薪，并设计成员函数 **setSalary()/getSalary()** 修改和访问周薪。将销售员抽象成类 **SalesPerson**，用 **basePay** 表示底薪，**salesValue** 表示销售额，用成员函数 **setBasePay()/getBasePay()**、**setSalesValue()/getSalesValue()** 修改和读取底薪及销售额。将小时工抽象成类 **HourPerson**，用 **hprice** 表示小时工资，用 **hour** 表示工作

时间，并用 `setHprice()/getHprice()`、`setHour()/getHour()` 设置/读取小时工资及工作时间。

由于要计算每类人员的工资并打印输出，因此可以将它抽象成成员函数 `getSalary()` 和 `print()`，并设置为基类 `Empolyee` 的虚成员函数。由于每类人员的工资计算和输出信息都不同，因此各派生类需要覆盖（重定义）这两个成员函数。它们的继承关系如图 5-2 所示。

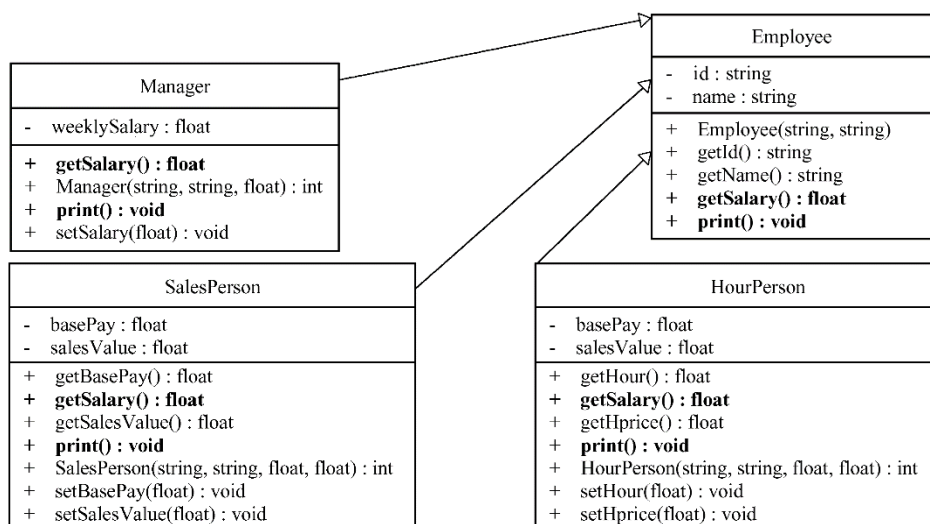


图 5-2 经理、销售员、小时工和雇员类的继承关系

采用多态方式计算人员工资、打印信息可以简化程序设计。以基类 `Employee` 类为接口，通过该类的指针或引用调用派生类对象的成员函数 `getSalary()` 和 `print()`，就可以有效地解决问题。这里只给出类 `Employee` 和 `Manager` 的设计和部分代码，类 `SalesPerson` 和 `HourPerson` 与类 `Manager` 的设计大同小异，在此略掉。先看一个没有用虚函数的程序版本。

```

// Eg5-2.cpp
#include <iostream>
#include <string>
using namespace std;

class Employee {
public:
    Employee(string Name, string Id) { name = Name; id = Id; }
    string getName() { return name; } // 返回姓名
    string getID() { return id; } // 返回身份证号
    float getSalary() { return 0.0; } // 返回工资
    void print() { cout<<"姓名: "<<name<<"\t\t 编号: "<<id<<endl; } // 输出姓名和身份证号
private:
    string name;
    string id;
};

class Manager:public Employee {
public:
    Manager(string Name, string Id, float s = 0.0):Employee(Name, Id){
        weeklySalary = s;
    }

    void setSalary(float s) { weeklySalary = s; } // 设置经理的周薪
  
```

```

float getSalary() { return weeklySalary; }           // 获取经理的周薪
void print() {                                       // 打印经理的姓名、身份证号、周薪
    cout<<"经理: "<<getName()<<"\t\t 编号: "<<getID()<<"\t\t 周工资: "<<getSalary()<<endl;
}
private:
    float weeklySalary;                             // 周薪
};

void main() {
    Employee e("黄春秀", "N00009"), *pM;
    Manager m("刘大海", "N00001", 128);
    m.print();
    pM = &m;
    pM->print();
    Employee &rM = m;
    rM.print();
}

```

程序的运行结果如下:

```

经理: 刘大海      编号: N00001      周工资: 128
姓名: 刘大海      编号: N00001
姓名: 刘大海      编号: N00001

```

显然, 输出结果的第 2 行和第 3 行并不是我们期望的。因为指针 `pM` 和引用 `rM` 所操作的都是派生类的对象 `m`, 所以希望 `pM->print()` 和 `rM.print()` 的输出结果与 `m.print()` 相同。也就是说, 三个输出结果都应 与第 1 行相同。

产生这种输出的原因是: 当基类对象的指针指向派生类对象时, 只能通过它访问派生类对象中的基类子对象。因此, 尽管基类 `Employee` 的指针 `pM` 指向了派生对象 `m`, 但它只能访问到 `m` 中属于基类 `Employee` 子对象的那部分成员, 所以 `pM->print()` 只能访问在 `Employee` 中定义的成员函数 `print()`, 不能访问在 `Manager` 中定义的成员函数 `print()`。同理, 引用 `rM.print()` 也只能访问基类 `Employee` 中的成员函数 `print()`。

实际上, 这里需要通过基类指针 `pM` 访问派生类 `Manager` 中的成员函数 `print()`, 因为 `pM` 实际指向的是一个 `Manager` 对象。在 C++ 中, 当基类对象的指针实际指向了一个派生类对象时, 可以通过强制类型转换, 将基类指针转换成派生类的指针, 这样就能实现对派生类成员函数的访问。如在例 5-2 中, 把 `pM->print()` 函数调用转换成下面的函数调用形式:

```
((Manager*)pM)->print();
```

则通过 `pM` 访问到的就是在派生类 `Manager` 中定义的成员函数 `print()`, 输出结果是:

```

经理: 刘大海      编号: N00001      周工资: 128

```

但是, 这种类型转换方法并不灵活, C++ 给出了一种更好的解决方案——[虚函数](#)。虚函数只能在类中定义, 即只能把类的成员函数声明为虚函数, 不属于任何类的普通函数不能被定义成虚函数。虚函数的定义是在成员函数的声明前面加上关键字 `virtual`, 其他方面与普通成员函数的定义和使用方法完全相同。例如:

```

class x {
    .....
    virtual f(参数表);
}

```

构造函数之外的任何非静态函数都可以是虚函数。需要注意的是，关键字 `virtual` 只能出现在类内部的函数声明之前，不能用于修饰类外部的成员函数定义。另外，如果基类把一个函数声明为虚函数，那么该函数在其派生类中也是虚函数。

`virtual` 的意义在于指示编译器，对该函数采取延后联编（动态绑定）的方法，在程序运行过程中才确定与之对应的调用函数，具有虚函数的类也被称为多态类。没有用 `virtual` 限定的函数采用早期联编（静态绑定）的方式，在编译过程中就会确定与之对应的调用函数。

虚函数的运行机制可以概括如下：如果基类中的非静态成员函数被定义为虚函数，且当派生类覆盖了（指在派生类中定义的成员函数，其函数原型与其基类中的某成员函数完全相同，也称为重定义）基类的虚函数，当通过基类的指针或引用调用虚函数时，编译器将执行动态绑定，调用该指针（或引用）实际所指对象所在类中的虚函数版本。

在例 5-2 中，若将基类 `Employee` 的成员函数 `print()` 声明为虚函数，就能够让 `pM->print()` 和 `rM.print()` 访问到派生类 `Manager` 中定义的 `print()`。做法很简单，只需在 `Employee` 类的成员函数 `print()` 前面加上 `virtual`，其余程序代码不做任何修改，如下所示。

```
class Employee {
    .....
    virtual void print(){ cout<<"姓名: "<<name<<"\t\t 编号: "<<id<<endl; }
    .....
};
```

下面是将基类 `Employee` 中的 `print()` 设置为虚函数后的运行结果。

经理: 刘大海	编号: N00001	周薪: 128
经理: 刘大海	编号: N00001	周薪: 128
经理: 刘大海	编号: N00001	周薪: 128

此结果表明，`pM->print()` 和 `rM.print()` 调用的都是派生类 `Manager` 定义的函数 `print()`。

有了虚函数后，在设计继承结构中的基类时必须考虑两类成员函数的设计：一类是基类希望派生类继承而不要改变的函数；另一类是基类希望派生类进行覆盖的函数，要求派生类各自定义适合自身的函数版本。对于前者，基类应将其设置为普通成员函数；对于后者，基类应将其定义为虚函数。

5.2.2 `override` 和 `final` C++11

`override` 和 `final` 是 C++ 11 标准才提出来的用于限定虚函数的关键字，不能用它们限定非虚函数。如果派生类本意是想覆盖基类的某个虚函数，定义自己的虚函数版本，却因疏忽而提供了一个与基类的虚函数同名但形参表不同的函数，仍然是合法的，编译器会将该函数作为派生类新增的成员函数处理，它与从基类继承到的虚函数是两个相互独立的函数。例如：

```
class B {
public:
    virtual void outData(int a) { cout<<a; };
};
class D : public B {
public:
    void outData(double b) { cout<<b; }
};
```

类 D 的本意是用自己定义的 `outData()` 覆盖从基类继承来的虚函数 `outData()`，但不小心将 `int` 类型的参数定义成了 `double` 类型，编译器会将它们处理成两个不同版本的函数。类似于下面的形式：

```
class D {
    .....
    virtual void outData(int a) { cout<<a; };           // 从基类继承，虚函数
    void outData(double b) { cout<<b; };               // 类 D 自定义，非虚函数
}
```

类 D 虽然并不会出现编译错误，但是无法正确实现多态，这与设计该类之本意并不相符，实际上也是一种程序错误，要发现这样的错误非常困难。在 C++ 11 新标准中，可以用 `override` 关键字对派生类提供的覆盖虚函数进行标识，如果用 `override` 标记了派生类的某个函数，而该函数却没有覆盖其基类的某个虚函数，将出现编译错误，程序员就能够及时发现错误，解决问题。

```
class D:public B {
    public:
        void outData(double b) override { cout << b; }    // 覆盖基类的虚函数
};
```

`override` 声明类 D 将用 `outData()` 覆盖从基类继承到的虚函数 `outData()`，但这两个函数的参数类型并不相同，将出现编译错误。类设计者因而能够及时发现错误，从而将派生类 D 的 `outData()` 成员函数的形参类型改为 `int`，实现类 D 对虚函数 `outData()` 的覆盖。

`override` 只能出现在派生类的成员函数声明中，用来标记从基类继承到的虚函数。如果用它标记基类的非虚函数，或者派生类新定义的函数，就会产生编译错误。例如：

```
class B {
    public:
        virtual void g1(int a) { cout<<a; };
        void g2(int b) { cout<<b; }
};
class D : public B {
    public:
        void g1(int b) override { cout<<b; }           // 正确，g1()与基类的虚函数 g1()匹配
        void g2(int b) override { cout<<b; }           // 错误，基类 B 的 g2()不是虚函数
        void g3(int b) override { cout<<b; }           // 错误，基类 B 没有 g3()函数
        void g1(char b) override { cout<<b; }           // 错误，与基类的虚函数 g1()不匹配
};
```

注意：用 `override` 可以明确表达类设计者用某函数覆盖从基类继承到的虚函数的意图，而不是说必须用 `override` 才能定义覆盖函数。

实际上，在 C++ 中，只要派生类中的函数声明与基类的某个虚函数原型相同，无论是否用 `override` 关键字声明，它都是基类虚函数的覆盖版本。

例如，下面对 D1 类的成员函数 `g1()` 的声明添加了 `override` 关键字，而 D2 的 `g1()` 没有用 `override` 声明，但它们与上面的类 B、D 中 `g1()` 的函数原型相同，因此都是虚函数。

```
class D1:public D {
    public:
        void g1(int x) override { cout<<x; }           // 添加了 override 声明
};
```



```
};
class D2:public D1 {
    void g1(int a){ cout<<a; }           // 没用 override 声明
};
```

基类可以将只想让派生类继承而不允许覆盖的虚函数指定为 **final**。虚函数一旦被限定为 **final**，则任何派生类对该函数的覆盖定义都是错误的。同 **override** 一样，**final** 也只能用来限定虚函数。例如，对前面的类 **B** 和 **D**，有如下继承关系。

```
class D1:public D {
public:
    void g1(int x) final { cout<<x; }      // 正确，不允许 D1 的派生类覆盖 g1()
    void f(int y) final { cout<<y; }      // 错误，f() 不是虚函数
};
class D2:public D1 {
    void g1(int a) override { cout<<a; }  // 错误，D1 已声明 g1() 为 final
};
```

派生类 **D1** 覆盖了从基类 **D** 继承到的虚函数 **g1()**，并用 **final** 声明它的 **g1()** 为最终版本，不允许 **D1** 的任何派生类覆盖 **g1()**，只能够继承它。因此，派生类 **D2** 企图覆盖 **g1()** 是错误的。**f()** 并不是虚函数，不允许用 **final** 限定，所以也是错误的。

5.2.3 虚函数的特性

派生类继承了基类的全部成员函数，但是对于从基类继承来的虚函数，派生类通常需要定义自己的覆盖函数版本，以实现派生类需要的新功能。

此外，在通常情况下，如果在程序中声明了某个函数而没有调用它，就可以不提供该函数的定义，但是类的虚函数不一样，无论程序中是否调用了虚函数，都必须为每个虚函数提供定义。这是因为编译器无法确定到底哪个虚函数会被调用。为了避免虚函数被调用时还没有定义的事情发生，因此要求所有的虚函数在程序执行之前都有定义。

除了上面两点不同于普通成员函数之外，虚函数还具有以下几个特性：① 一旦将某成员函数声明为虚函数后，它在类的继承结构中就永远为虚函数了；② 将基类的成员函数定义为虚函数后，虚特性在定义它的类和之后继承它的派生类中有效，即使派生类在重定义该函数时并没有将它声明为虚函数，它仍然是虚函数；③ 如果定义虚函数的类是从其他类派生，这些虚函数不会影响基类中的同名成员函数，基类中的同名函数保持它的原有特性。

【例 5-3】 虚函数与派生类的关系。

```
// Eg5-3.cpp
#include <iostream>
using namespace std;

class A {
public:
    void f(int i){ cout<<"...A"<<endl; };
};
class B: public A {
public:
    virtual void f(int i){ cout<<"...B"<<endl; }
```



```

};
class C: public B {
public:
    void f(int i){ cout<<"...C"<<endl; }
};
class D: public C {
public:
    void f(int){ cout<<"...D"<<endl; }
};

void main() {
    A *pA, a;
    B *pB, b;
    C c;
    D d;
    pA=&a;    pA->f(1);           // 调用 A::f()
    pA=&b;    pA->f(1);           // 调用 A::f()
    pA=&c;    pA->f(1);           // 调用 A::f()
    pA=&d;    pA->f(1);           // 调用 A::f()
}

```

程序的运行结果如下：

```

...A
...A
...A
...A

```

这个结果并没有体现成员函数 f() 的虚特性。怎么回事呢？

图 5-3 是 A、B、C、D 四个类中的成员函数 f() 是否为虚函数的示意。在类 B 中将 f() 声明为虚函数，所以在 B 的派生类 C 及 C 的派生类 D 中，f() 都是虚函数。

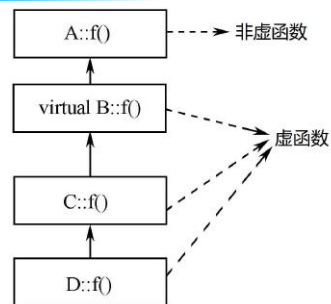


图 5-3 虚函数与继承

① 虚函数特性只对自定义它之后的派生类有效，而对之前的基类则没有任何影响，因此在 B 的基类 A 中，f() 不是虚函数。因为 A::f() 不是虚函数，例 5-3 的函数 main() 中的 4 个函数调用 pA->f(1) 都只能调用到基类 A 中定义的函数 f()，所以就得到了上面的程序执行结果。

② 如果基类定义了虚函数，当通过基类指针（智能指针）或引用调用派生类对象时，会执行动态绑定，将访问到它们实际所指对象中的虚函数版本。

例如，若把例 5-3 中 main() 的 pA 指针修改为 pB，则将体现虚函数的特征。

```

void main() {
    A *pA, a;
    B *pB, b;
    C c;
    D d;
    // pB = &a;    pB->f(1);           // 错误，派生类不能访问基类对象
    pB = &b;    pB->f(1);           // 调用 B::f()
    pB = &c;    pB->f(1);           // 调用 C::f()
    pB = &d;    pB->f(1);           // 调用 D::f()
}

```

“pB = &a;”是错误的，因为 pB 是用派生类 B 定义的指针，将它指向基类 A 的对象是错

误的。

但是对于 c 和 d 而言，pB 是它们的基类 B 的指针，而 f() 是在 B 中定义的虚函数，因此通过 pB 访问 c 和 d 是正确的，而且会体现虚函数 f() 的多态特征：即当 pB 指向 c 时，将调用 C::f()；当 pB 指向 d 时，将调用 D::f()。因此，程序的运行结果将是：

```
...B
...C
...D
```

③ 只有通过基类对象的指针（包括智能指针）和引用访问派生类对象的虚函数时，才能体现虚函数的特征。当通过普通的基类对象访问派生类对象时，不能实现虚函数的特征，只能访问到派生类从基类继承到的成员函数。

【例 5-4】 只能通过基类对象的指针和引用才能实现虚函数的特征。

```
// Eg5-4.cpp
#include<iostream>
#include<memory>
using namespace std;

class B {
public:
    virtual void f() { cout<<"B::f"<<endl; };
};

class D : public B {
public:
    void f() { cout<<"D::f"<<endl; };
};

void main() {
    D d;
    B* pB = &d, & rB = d, b;
    unique_ptr<B> pb{ new D }; // 基本类智能指针指向了派生对象
    b = d;
    b.f();
    pB->f();
    rB.f();
    pb->f();
}
```

本程序的运行结果如下：

```
B::f
D::f
D::f
D::f
```

输出结果的第 1 行是 b.f() 产生的，表明通过基类对象 b 调用派生类对象 d 时，只能访问到基类 B 的成员函数 f()。第 2 行是 pB->f() 产生的，第 3 行是 rB.f() 产生的，第 4 行则是用基类定义的独占型智能指针（也可用 share_ptr 指针）实现的多态输出。程序运行结果表明，通过基类 B 的指针 pB 和引用 rB 都能实现虚函数的多态性，访问到派生类中定义的虚函数 f()。

④ 派生类中的虚函数要保持其虚特征，必须与基类虚函数的函数原型完全相同（要求每

个对应形参的类型相同，函数返回类型也相同)，否则就是普通的重载函数，与基类的虚函数无关。

【例 5-5】 基类 B 和派生类 D 都具有成员函数 f()，但它们的参数类型不同，因此不能体现虚函数特征。

```
// Eg5-5.cpp
#include <iostream>
using namespace std;

class B {
public:
    virtual void f(int i) { cout<<"B::f"<<endl; };           // L1
};
class D:public B {
public:
    int f(char c) { cout<<"D::f..."<<c<<endl; }           // L2
    void f(int i) const { cout<<"D::f" <<endl; };           // L3
};

void main() {
    D d;
    B *pB = &d, &rB = d, b;
    pB->f('1');
    rB.f('1');
}
```

本程序的运行结果如下：

```
B::f
B::f
```

这个结果表明，基类成员函数 f() 虽然是虚函数，而且基类指针 pB 和引用 rB 绑定到了派生类 D 的对象上，但通过它们并没有调用到派生类 D 定义的成员函数 f()。

原因是派生类 D 中虽然有两个与基类 B 中同名的成员函数 f()，但它们都与基类的虚函数原型不同，所以它们是不同的成员函数。要让 D 中的 f() 成为虚函数，必须让它与 B 中的 f() 具有相同的函数原型，即 D 中成员函数 f() 的原型必须是 void f(int i)。

⑤ 派生类对象通过从基类继承的成员函数调用虚函数时，将访问到派生类中的版本。

【例 5-6】 派生类 D 的对象通过从基类 B 继承的成员函数 f() 调用派生类 D 中的虚函数 g()。

```
// Eg5-6.cpp
#include <iostream>
using namespace std;

class B {
public:
    void f(){ g(); }
    virtual void g() { cout<<"B::g"; }
};

class D:public B {
public:
    void g() { cout<<"D::g"; }
```

```
};

void main() {
    D d;
    d.f();
}
```

程序运行结果如下：

```
D::g
```

由于 D 没有定义函数 f(), 因此 d.f() 将调用 B::f(), 而 B::f() 又调用了类 B 的虚函数 g()。归根结底, 实质上是派生类 D 的对象 d 在调用虚函数 g(), 因此将调用 D 中的虚函数 g()。如果 B::g() 不是虚函数, 本程序的输出结果将是 “B::g”。

【例 5-7】 分析下面程序的输出结果, 理解虚函数的调用过程。

```
// Eg5-7.cpp
#include<iostream>
using namespace std;

class B {
public:
    void f() { cout<<"bf "; }
    virtual void vf() { cout<<"bvf "; }
    void ff() { vf(); f(); };
    virtual void vff() { vf(); f(); }
};

class D: public B {
public:
    void f() { cout<<"df "; }
    void ff() { f(); vf(); }
    void vf() { cout<<"dvf"; }
};

void main() {
    D d;
    B *pB = &d;
    pB->f();
    pB->ff();
    pB->vf();
    pB->vff();
}
```

程序的运行结果如下：

```
bf    dvf    bf    dvf    dvf    bf
```

请读者结合前面介绍的虚函数特征, 理解这个结果的产生过程。

⑥ 只有类的非静态成员函数才能被定义为虚函数, 类的构造函数和静态成员函数不能被定义为虚函数。

⑦ 内联函数也不能是虚函数。因为内联函数采用的是静态绑定方式, 而虚函数是在程序运行时才与具体函数动态绑定, 采用的是动态绑定方式, 即使虚函数在类体内被定义, C++ 编译器也将它视为非内联函数处理。

5.3 虚析构函数

析构函数可以定义为虚函数（构造函数不能是虚函数）。在继承体系结构中，如果基类的析构函数是虚函数，则其所有派生类的析构函数都是虚函数。

在销毁通过基类指针（或引用）调用的派生类对象时，虚析构函数能够使继承结构中各层次的类对象的析构函数都被调用。在销毁自由存储空间中用 `new` 建立的对象时，虚析构函数可以确保在用 `delete` 销毁动态分配的派生类对象时调用到正确的析构函数，完成对象所占内存空间的回收。典型情况是当用基类对象的指针（或引用）调用派生类对象时，如果基类析构函数不是虚函数，则通过基类指针（或引用）对派生类的析构很可能是不彻底的。

【例 5-8】 在非虚析构函数的情况下，通过基类指针对派生对象的析构是不彻底的。

```
// Eg5-8.cpp
#include <iostream>
using namespace std;

class A {
public:
    ~A(){ cout<<"call A::~~A()"<<endl; }
};

class B:public A {
    char *buf;
public:
    B(int i) { buf = new char[i]; }
    ~B() {
        delete [] buf;
        cout<<"call B::~~B()"<<endl;
    }
};

void main() {
    A* a = new B(10);
    delete a;
}
```

程序运行结果如下：

```
call A::~~A()
```

这表明，通过基类 A 的指针 a 对派生类对象的销毁是不彻底的，因为派生类对象的析构函数没有被调用，分配给派生对象成员 `buf` 的动态存储空间没有被回收，造成了内存泄露。

如果将类 A 和 B 的析构函数设置为虚函数，即在析构函数 `~A()` 和 `~B()` 前面加上限定词 `virtual`，类似于如下形式：

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

class B:public A {
    .....
    virtual ~B() {...}
```

```
};
```

当然，即使 `virtual ~B()` 前面没有 `virtual`，它仍然是虚函数。将 `A`、`B` 两类的析构函数设置为虚函数后，本程序的运行结果如下：

```
call B::~~B()  
call A::~~A()
```

这表明，如果析构函数是虚函数，在通过基类的指针（或引用）销毁派生类对象时，同时调用了基类和派生类的析构函数，派生类对象的 `buf` 成员所占用的动态存储空间被回收了。

5.4 纯虚函数和抽象类

在通常情况下，定义一个类的目的是用它来建立对象，并利用对象来解决实际问题。但在有些情况下，定义类的时候并不知道如何实现它的某些成员函数，定义该类的目的也不是为了建立它的对象，而是为了表达某种概念，并作为继承结构顶层的基类，然后以它为接口访问派生类对象。那些在基类中无法实现的成员函数，在派生类中却有具体的实现方法。在面向对象程序设计语言中，用纯虚函数来表达这类函数。具有纯虚函数的类被称为**抽象类**。

例如，本章前面对动物类 `Animal` 的抽象就是一个很好的例子。只知道所有的动物都会发声，却无法说出发出什么声音，可以用纯虚函数 `sound()` 来表示动物会发声这一行为，等到从它派生出 `Cat`、`Dog` 等具体动物类型时就知道该如何发声了，这时再重定义从类 `Animal` 继承来的成员函数 `sound()`，实现各自的 `sound()` 函数版本。定义类 `Animal` 的目的不是要定义它的对象，而是用来表达动物这一概念，希望通过它的指针或引用访问由它派生出的具体动物类的虚函数 `sound()`，把它定义为抽象类，就能够实现这一设计目标。

5.4.1 纯虚函数和抽象类

纯虚函数是指在声明时被初始化为 0 的类成员函数，形式如下：

```
class X {  
    .....  
    virtual return_type func_name (param) = 0;  
}
```

纯虚函数在基类中声明，但它在基类中没有具体的函数实现代码，要求继承它的派生类为纯虚函数提供实现代码。

一个类可以声明一个或多个纯虚函数，只要有纯虚函数的类就是抽象类。抽象类只能作为其他类的基类，不能用来建立对象，所以又称为**抽象基类**。

C++对抽象类有以下限定：① 抽象类中含有纯虚函数，由于纯虚函数没有实现代码，因此不能建立抽象类的对象；② 抽象类只能作为其他类的基类，可以通过它的指针或引用访问到它的派生类对象，实现运行时的多态性；③ 如果派生类只是简单地继承了抽象类的纯虚函数，并没有重定义它们，那么它也是一个抽象类。

【例 5-9】 在一个图形系统中，设计计算各种图形面积的程序。

问题分析：所有图形都有面积，但只有落实到三角形、矩形等具体图形时才能够计算出它的面积。可以设计一个抽象类 `Figure` 来表示图形的概念，并为它设置纯虚函数 `area()` 计算

图形的面积。而圆、三角形、矩形等具体图形从类 **Figure** 派生，由它们提供纯虚函数 **area()** 的实现版本。借助虚函数，可以通过基类 **Figure** 的指针或引用访问到三角形、矩形等派生类实现的面积函数。

```
// Eg5-9.cpp
#include <iostream>
using namespace std;

class Figure {
protected:
    double x, y;
public:
    void set(double i, double j) { x = i; y = j; }
    virtual void area() = 0; // 纯虚函数
};

class Triangle:public Figure {
public:
    void area(){ cout<<"三角形面积: "<<x*y*0.5<<endl; } // 覆盖基类的纯虚函数
};

class Rectangle:public Figure {
public:
    void area(int i) { cout<<"这是矩形，它的面积是: "<<x*y<<endl; }
};

void main() {
    Figure *pF;
    // Figure f1; // L1, 错误
    // Rectangle r; // L2, 错误
    Triangle t; // L3
    t.set(10, 20);
    pF = &t;
    pF->area(); // L4
    Figure &rF = t;
    rF.set(20, 20);
    rF.area(); // L5
}
```

程序运行结果如下：

三角形面积: 100

三角形面积: 200

在本程序中，基类 **Figure** 的成员函数 **area()** 是纯虚函数，因此 **Figure** 是一个抽象类，不能用来建立对象，这是语句 **L1** 错误的原因。

尽管 **Figure** 的派生类 **Rectangle** 重新定义了成员函数 **area()**，但它有一个 **int** 类型的参数，根据虚函数的特征，只有派生类与基类的虚函数具有完全相同的函数名、返回类型和参数表时，才能够体现虚函数的特征，所以 **Rectangle** 中的函数 **area()** 并不是其基类 **Figure** 的纯虚函数 **area()** 的覆盖函数版本。因此，**Rectangle** 仍然是一个抽象类，不能建立它的对象，这就是语句 **L2** 错误的原因。

派生类 **Triangle** 为基类 **Figure** 的纯虚函数 **area()** 提供了覆盖函数版本，不再是抽象类了，

可以用它来建立对象，所以语句 L3 是正确的。

语句 L4 和 L5 中的 `pF`、`rF` 分别是用抽象类 `Figure` 定义的指针和引用，可以实现对派生类 `Triangle` 对象 `t` 的虚函数 `area()` 的访问。

5.4.2 抽象类的应用

在设计类的继承结构时，可以把各派生类都需要的功能设计成抽象基类的虚函数，各派生类根据自己的情况重新定义虚函数的功能，以便描述各类特有的行为。由于抽象基类具有各派生类成员函数的虚函数版本，可以把它作为访问整个继承结构的接口，即通过抽象基类的指针或引用，访问在各派生类中实现的虚函数版本，这种方式也称为[接口重用](#)，即不同的派生类都可以把抽象基类作为接口，让其他程序通过此接口访问各派生类的功能。

通过接口重用的方式能够设计出功能强大的类继承结构，在设计处理大量类型不同但在高层又具有统一接口的类时，这种方式非常有效。设计人员可以通过继承抽象类的接口，为继承增加新类，也可以重新定义各派生类中虚函数的实现代码，而这些改动不会引起抽象类接口的变化，也就不会导致以抽象类为接口的其他程序代码的变化。

图 5-4 是抽象类接口重用的一个简单示意图。抽象基类 `Base` 以虚函数的方式定义了继承结构中各派生类共有的函数接口，各派生类再根据自己的情况继承或重定义各自的虚函数版本。外部函数 `pf()` 通过基类 `Base` 的对象指针（引用）就能够访问到在 `Base` 中声明的所有虚函数。事实上，`pf()` 访问的并非 `Base` 中定义的虚函数，而是以它为接口访问各派生类对象中的虚函数。比如，在如下函数调用中，`pf()` 实际访问的是派生类 `Derived1` 中定义的成员函数 `vf1()`、`vf2()` 等：

```
Derived1 d1;  
pf(&d1);
```

而在如下函数调用中，`pf()` 实际访问的是派生类 `Derived21` 中定义的成员函数 `vf1()`、`vf2()` 等：

```
Derived21 d2;  
pf(&d2);
```

以基类 `Base` 作为类继承结构的接口，更强大的功能在于它能自动适应继承结构的扩展。假设在图 5-4 的继承结构中增加了 `Base` 的直接派生类 `Derived4`，以 `Base` 为接口访问继承结构的 `pf()` 函数不需做任何修改，就能访问派生类 `Derived4` 定义的 `vf1()`、`vf2()` 等虚函数。此外，当重定义任何派生类中的 `vf1()`、`vf2()` 等虚函数时，`pf()` 也不需要做任何修改就能访问最新的虚函数版本。抽象类的这种能力为软件的升级和维护带来了极大方便。现在来看一个以抽象类作为继承结构接口的完整例子。

【例 5-10】 扩展例 5-9 图形面积和体积的程序功能，假设有点、圆、圆柱体几种图形，要计算每种图形的面积和体积，并且要求输出各种图形的类名称和各个类对象的数据成员。用接口与实现分离的方式，实现点、圆、圆柱体等几何图形的面积和体积计算功能。

问题分析：点、圆、圆柱体、三角形、四边形等都是几何图形，它们具有一些共性，如都有类型名、面积、体积和周长等。但在没有具体到圆、三角形等具体形状时，又无法进行面积、体积和周长的计算，虽然它们只是个概念，但是确实存在，适合用纯虚函数和抽象类来描述它们。

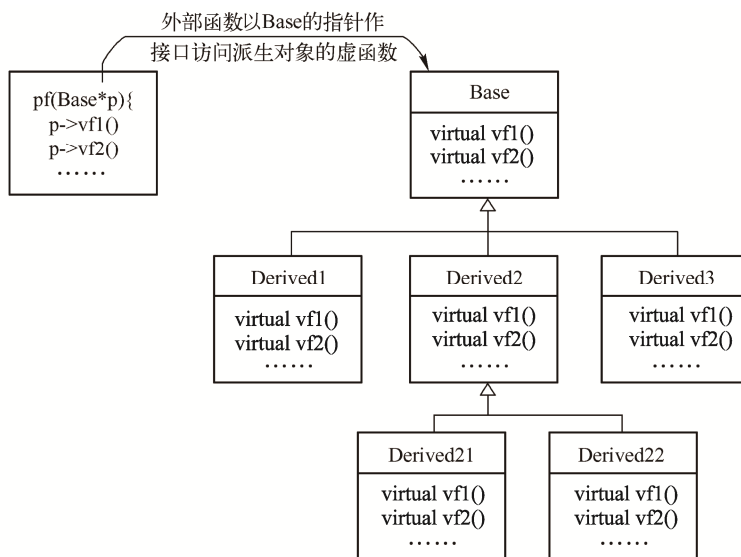


图 5-4 抽象类作为继承结构的访问接口

因此，用类 **Shape** 表示几何图形这一概念，把各类图形计算面积、体积的函数设置成纯虚成员函数 **area()** 和 **volume()**。为了便于图形数据的打印输出，在类 **Shape** 中设置输出图形类型、图形数据（如面积、体积、圆半径等）的纯虚函数 **printShapeName()** 和 **print()**。将点、圆、圆柱体等具体图形抽象成类 **Point**、**Circle**、**Cylinder**，它们从类 **Shape** 派生，每个类根据自己的实情，重定义从类 **Shape** 继承到的纯虚函数 **area()**、**volume()** 等。

各类和抽象基类 **Shape** 形成了图 5-5 所示的继承结构，图中只列出了各类需要重定义的虚函数，省略了它们的数据成员和其他成员函数。这样，以类 **Shape** 为接口，通过指针或引用能够访问到 **Point**、**Circle** 等派生类实现的纯虚函数 **area()**、**volume()** 等的覆盖函数版本，实现计算各类图形的面积和体积，以及数据输出等功能。

(1) 基类 shape

Shape 是一个抽象类，头文件的内容如下。

```
// Shape.h
#ifndef SHAPE_H
#define SHAPE_H
class Shape {
public:
    virtual double area() const = 0;
    virtual double volume() const = 0;
    virtual void printShapeName() const = 0;
    virtual void print() const = 0;
};
#endif
```

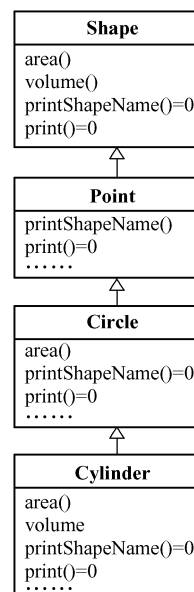


图 5-5 Shape 继承

Shape 是一个抽象类，没有数据成员，但定义了继承结构中各类都需要的公有成员函数。定义类 **Shape** 的目的只是为了供 **Point**、**Circle** 等类继承，并提供被外部函数访问的统一接口。**Shape.h** 对类 **Shape** 的定义是完整的，不需要源代码文件。

(2) 类 Point 的头文件

类 Point 从类 Shape 派生, 为了定义它的对象, 它必须覆盖基类 Shape 中的全部纯虚函数。

```
// Point.h
#include<iostream>
using namespace std;

#include "shape.h"
#ifndef POINT_H
#define POINT_H
class Point : public Shape {
public:
    Point(int = 0, int = 0);           // 构造函数, 将坐标值初始化为(0, 0)
    void setPoint(int, int);          // 设置点的坐标值
    int getX() const { return x; }    // 返回横坐标值
    int getY() const { return y; }    // 返回纵坐标值
    // 覆盖 Shape 类中的全部纯虚函数
    virtual double area() const;
    virtual double volume() const;
    virtual void printShapeName() const { cout<<"Point: "; }
    virtual void print() const;
private:
    int x, y;                         // Point 的坐标值
};
#endif
```

(3) Point 类的源文件

```
// Point.cpp
#include "point.h"

double Point::area()const { return 0; }
double Point::volume()const { return 0; }
Point::Point(int a, int b) { setPoint(a, b); }
void Point::setPoint(int a, int b) { x = a; y = b; }
// 按[x, y]形式输出 point 对象的数据
void Point::print() const{ cout << '[' << x << ", " << y << ']'<< " "; }
```

(4) 类 Circle 的头文件

类 Circle 从类 Point 派生, 继承了类 Point 的全部数据成员, 但圆是有半径的, 因此新增了数据成员 radius 表示半径。圆的面积、类型名称和输出数据不同于点, 因此覆盖了从 Point 继承到的虚函数 area()、printShapeName()、print(), 实现了类 Circle 需要的功能。

类 Circle 没有覆盖从类 Point 继承的体积计算函数 volume(), 因为类 Point 已覆盖了它从类 Shape 继承的这个纯虚函数, 返回体积 0。而圆没有体积, 可以直接复用函数 Point::volume() 的功能, 没有必要覆盖它。

```
// Circle.h
#ifndef CIRCLE_H
#define CIRCLE_H
#include "point.h"
class Circle : public Point {
```

```

public:
    Circle(double r = 0.0, int x = 0, int y = 0);
    void setRadius(double);
    double getRadius() const;
    virtual double area() const;
    virtual void printShapeName() const { cout<<"Circle: "; }
    virtual void print() const;
private:
    double radius; // 圆的半径
};
#endif

```

(5) Circle 类的源文件

```

// Circle.cpp
#include "circle.h"

Circle::Circle(double r, int a, int b):Point(a, b) { setRadius(r); }
void Circle::setRadius(double r) { radius = r > 0 ? r : 0; }
double Circle::getRadius() const { return radius; }
double Circle::area() const { return 3.14159 * radius * radius; }
void Circle::print() const {
    Point::print();
    cout<<" Radius = "<<radius;
}

```

(6) Cylinder 类

在圆的基础上增加高就变成了圆柱体，因此从 Circle 类派生出 Cylinder 类是很合理的设计。但类 Cylinder 的面积、体积、类型和输出数据函数都不同于 Circle，因此需要覆盖虚函数 area()、printShapeName()、print()，实现自己需要的功能。

```

// Cylinder.h
#ifndef CYLINDER_H
#define CYLINDER_H
#include "circle.h"
class Cylinder : public Circle {
public:
    Cylinder(double h = 0.0, double r = 0.0, int x = 0, int y = 0 );
    void setHeight(double);
    double getHeight();
    virtual double area() const;
    virtual double volume() const;
    virtual void printShapeName() const { cout<<"Cylinder: "; }
    virtual void print() const;
private:
    double height;
};
#endif

```

(7) Cylinder 类的源文件

```

// Cylinder.cpp

```

```

#include "cylinder.h"

Cylinder::Cylinder(double h, double r, int x, int y):Circle(r, x, y) { setHeight(h); }
void Cylinder::setHeight(double h) { height = h > 0 ? h : 0; }
double Cylinder::getHeight() { return height; }
double Cylinder::area() const { return 2*Circle::area() + 2*3.14159*getRadius()*height; }
double Cylinder::volume() const { return Circle::area()*height; }
void Cylinder::print() const {
    Circle::print();
    cout<<"    Height = "<<height;
}

```

(8) 函数 main()

```

// Main.cpp
#include <iostream>
#include <iomanip>
#include "shape.h"
#include "point.h"
#include "circle.h"
#include "cylinder.h"
using namespace std;

void vpf(const Shape *bptr) {                                     // 利用基类 Shape 的指针作接口访问派生类
    bptr->printShapeName();                                       // 打印对象所在的类名
    bptr->print();                                                 // 打印对象的数据成员
    cout<<"\nArea = "<<bptr->area()                                // 输出对象的面积
        <<"\nVolume = "<<bptr->volume()<<"\n\n";                // 输出对象的体积
}

void vrf(const Shape &bref) {                                     // 利用基类 Shape 的引用作接口访问派生类
    bref.printShapeName();                                       // 打印对象所在的类名
    bref.print();                                                 // 打印对象的数据成员
    cout<<"\nArea = "<<bref.area()                                // 输出对象的面积
        <<"\nVolume = "<<bref.volume()<<"\n\n";                // 输出对象的体积
}

int main() {
    // 设置数据输出格式，保留小数点后 2 位有效数字
    cout<<setiosflags(ios::fixed | ios::showpoint)<<setprecision(2);
    Point point(7, 11);
    Circle circle(3.5, 22, 8);
    Cylinder cylinder(10, 3.3, 10, 10);
    Shape *arrayOfShapes[3];                                     // 定义基类对象的指针数组
    arrayOfShapes[0] = &point;
    arrayOfShapes[1] = &circle;
    arrayOfShapes[2] = &cylinder;
    cout<<"----- 通过基类指针访问虚函数 -----\n";
    for(int i = 0; i < 3; i++)
        vpf(arrayOfShapes[i]);
    cout<<"----- 通过基类引用访问虚函数 -----\n";
    for (int j = 0; j < 3; j++)
        vrf(*arrayOfShapes[j]);
}

```

```

    return 0;
}

```

编译并运行本程序，运行结果如下。

```

----- 通过基类指针访问虚函数 -----
Point: [7, 11]
Area = 0.00
Volume = 0.00

Circle: [22, 8]; Radius = 3.50
Area = 38.48
Volume = 0.00

Cylinder: [10, 10]; Radius = 3.30; Height = 10.00
Area = 275.77
Volume = 342.12

----- 通过基类引用访问虚函数 -----
Point: [7, 11]
Area = 0.00
Volume = 0.00

Circle: [22, 8]; Radius = 3.50
Area = 38.48
Volume = 0.00

Cylinder: [10, 10]; Radius = 3.30; Height = 10.00
Area = 275.77
Volume = 342.12

```

本程序把 `Shape` 设计成抽象类，并从中设计了 4 个纯虚函数 `area()`、`volume()`、`printShapeName()`和 `print()`，它们构成了访问派生类 `Point`、`Circle`、`Cylinder` 中同名虚函数的接口。每个类都根据自己的实际情况覆盖了各虚函数，以实现本类需要的功能。程序中体现类 `Shape` 作为接口的是如下 2 个函数：

```

void vpf(const Shape *bptr)
void vrf(const Shape &bref)

```

函数 `vpf()`和 `vrf()`具有完全相同的功能，`vpf()`通过抽象基类 `Shape` 的指针 `bptr` 访问各派生类对象中定义的虚函数版本。下面的 `for` 循环展示了 `vpf` 通过指针访问 `point`、`circle`、`cylinder` 对象中的虚函数的过程，程序运行结果中的第 2~9 行就是这个 `for` 循环输出的。

```

for(int i = 0; i < 3; i++)
    vpf(arrayOfShapes[i]);

```

函数 `vrf()`通过 `Shape` 的引用 `bref` 访问各派生类对象的虚函数。下面的 `for` 循环演示了函数 `vrf()`通过 `Shape` 的引用访问派生类 `Point`、`Circle`、`Cylinder` 对象中的虚函数的过程，程序运行结果中的“-----通过基类引用访问虚函数-----”后的全部输出结果都是该 `for` 循环输出的。

```

for(int j = 0; j < 3; j++)
    vrf(*arrayOfShapes[j]);

```

当一个类有虚函数时，C++编译器会为该类创建一个虚函数表 `vtable`，在其中保存该类每个虚函数的地址，同时还会在该类的对象中自动添加一个虚函数指针成员 `vptr`，指向该类虚函数表的首地址。因此，如果一个类有虚函数，那么其对象所占存储空间的大小要比无虚函

数时多 4 字节或 8 字节（32 位系统的地址编码占 4 字节，若是 64 位系统，则为 8 字节）。

图 5-6 是本程序建立的对象与虚函数表的结构。arrayOfShape 数组保存了 3 个对象的地址，分别是 point、circle 和 cylinder，这 3 个对象的虚函数指针分别指向 Point、Circle、Cylinder 类的虚函数表 vtable()。

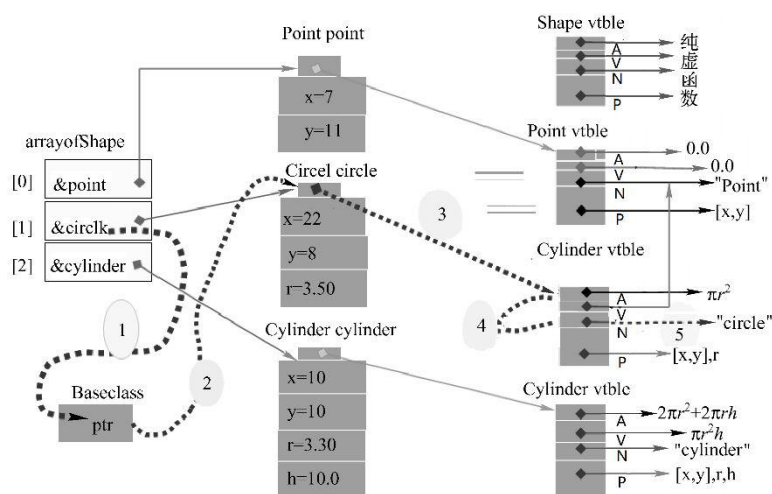


图 5-6 对象与虚函数的结构

每个类的虚函数表中保存了 4 个虚函数的地址：A 表示 `area()` 的地址，V 表示 `volume()` 的地址，N 表示 `printShapeName()` 的地址，P 表示 `print()` 的地址。图中各类的虚函数地址指向了对应的函数运行结果。Point vtable 中的 A 和 V 指向 0.0，表示它算出的点的面积和体积为 0。再如，Circle vtable 中的 A 指向了 πr^2 ，表示 Circle 的面积函数 `area()` 将按 πr^2 计算圆的面积。

图 5-6 中的虚线表示 `vpf(arrayOfShapes[1])` 调用时函数 `baseclassPtr->printShapeName()` 的执行过程，虚线上的编号表示虚函数的执行顺序。

- ① 将 `arrayOfShape` 数组的第 `i` 个元素的地址传递给将函数 `vpf()` 的形式参数，图 5-6 中所示的是 `i = 1` 时的情况，即将 `circle` 对象的地址传递给函数形参 `bptr`。
- ② 通过指针 `bptr` 找到 `circle` 对象。
- ③ 通过 `circle` 对象的虚函数指针 `vptr` 找到 `Circle` 类的虚函数表 `vtable`。
- ④ 从 `Circle` 的虚函数表向下偏移 8 字节（4 字节为一个地址，前面有 `area()` 和 `volume()` 两个虚函数地址，共 8 字节），找到类 `Circle` 的虚函数 `printShapeName()` 的指针。
- ⑤ 根据 `printShapeName()` 虚函数指针，找到 `Circle` 的 `printShapeName()` 虚函数版本，执行该虚函数，将在屏幕打印输出字符串 "Circle"。

本程序代表了抽象类的典型用法，在设计类继承层次结构时，把各类都具有的通用功能抽象成虚函数或纯虚函数，放在最上层的基类中，派生类再继承或覆盖这些虚函数，完成本类需要的功能。然后以抽象基类为接口，就能够访问整个继承结构中各派生类定义的虚函数，不再需要针对每个具体类编写独立的应用程序，简化了软件设计的复杂度，也给软件的功能扩展和软件维护带来了极大的方便。

【例 5-11】设计简单工厂模式计算几何图形的面积。

工厂模式是一种常用的软件开发模式，其优点是对外隐藏了系统内部的创建过程，在创建对象时不会对客户端暴露创建逻辑，而是使用一个共同的接口来访问新创建的对象，有利

于软件结构优化和功能扩展。借助面向对象语言中的抽象类、继承等技术可以方便地实现各种复杂程度的工厂模式。从原理上，实现最简单的工厂模式需要设计三种位于不同层面的类来模仿工厂产品的生产方式：① 抽象产品类（作为具体产品的基类，包含共有的可显示自身产品信息的纯虚函数，其析构函数也需要定义为虚函数）；② 具体产品类（实现从抽象产品类继承来的虚函数，生产实际的“产品”）；③ 抽象工厂类（提供生产具体产品的方法）。

在本例中，用 **Shape** 代表作接口的抽象产品类，具体的产品类则为 **Circle**、**Square**……（可以有任意多个产品类），用类 **Factory** 表示生产各种产品（圆、正方形、长方形……）的工厂。函数 **main()** 则代表应用 **Factory** 工厂生产的各种几何图形并计算其面积的应用方。

在工厂模式中，工厂类的基本结构通常是用一系列的选择结构（**switch** 或 **if**）确定要生产的产品，用抽象基类的指针指向实际产品类的对象。

```
// Eg5-11.cpp
#include<iostream>
#include<memory>
using namespace std;

using PRODUCT = enum {
    CIRCLE,
    SQUARE
    // .....
};

class Shape {
public:
    virtual double area(double) = 0;
    virtual ~Shape() {}
};

class Circle :public Shape {
public:
    double r = 0;
    virtual double area(double) { return 3.14 * r * r; }
};

class Square :public Shape {
public:
    double l = 0;
    virtual double area(double l) { return l * l; }
};

class Factory {
public:
    unique_ptr<Shape> MakeProduct(PRODUCT product) {
        unique_ptr<Shape> ptr = nullptr;
        switch (product) {
            case CIRCLE:
                ptr = unique_ptr<Circle>(new Circle); break;
            case SQUARE:
                ptr = unique_ptr<Square>(new Square); break;
        }
        return ptr;
    }
};
```

```

};

int main() {
    Factory factory; // L11, 建立工厂
    unique_ptr<Shape> produceptr = nullptr;
    int i;
    cout<<"input product, 1:circle, 2: squqre, ....."<<endl;
    cin>>i;

    switch (i) { // L12, 根据输入生产产品
        case 1:
            produceptr = factory.MakeProduct(CIRCLE); // L13, 生产 Circle
            cout<<"circlce area = "<<produceptr->area(3.0)<<endl;;
            break;
        case 2:
            produceptr = factory.MakeProduct(SQUARE); // L14, 生产 Square
            cout<<"square area = "<<produceptr->area(4.0)<<endl;;
            break;
        default:
            break;
    }
}

```

程序运行结果如下：

```

input product,1:circle, 2: squqre, ..... // 产品生产提示信息
2 // 输入 2
square area=16 // 创建正方形

```

本程序虽然简单，却有三方面的内容需要学习理解。一是抽象类作接口的基本方法，二是“工厂模式”的基本程序框架，三是多应用智能指针（`unique_ptr`），可以简化堆内存的管理，至少不用考虑内存是否产生泄漏的问题。

5.5 运行时类型信息

C++是一种强类型语言，每个对象（包括常量和表达式）的数据类型都必须在编译期确定，并且在程序运行期内保持类型不变。但是，为了实现多态，C++指针或引用的类型可能与它们实际代表的类型不一致（如基类指针可以指向派生类对象），当将一个多态指针转换为其实所指对象的类型时，就需要知道对象的类型信息，这些信息只有在程序运行时才能够确定。

运行时类型信息（Run-Time Type Information, **RTTI**）提供了在程序运行时刻获取对象类型的方法，是面向对象程序语言为解决多态问题而引入的一种语言特征。早期的面向过程程序设计语言（如 C 和 Pascal 等）本质上是一种静态类型语言，程序的数据类型必须在编译期确定，且在运行期内不能改变，不支持 RTTI 机制。

在 C++中，用于支持 RTTI 的运算符有 `dynamic_cast`、`typeid`。在一些编译环境的默认设置中，RTTI 是关闭的，如 Visual C++ 6.0。因此，要在 Visual C++ 6.0 中编译运行本节后面的例程，需要在程序编译运行之前，按下述方法启用 Visual C++ 6.0 的 RTTI 机制。

选择 Visual C++ 6.0 的“工程 | 设置…”菜单命令，弹出如图 5-7 所示的对话框；从“Y

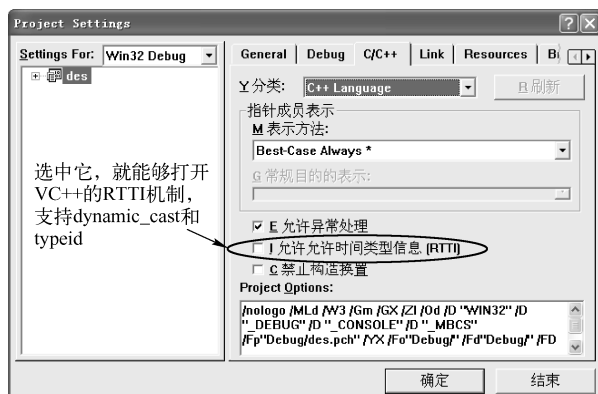


图 5-7 设置 Microsoft Visual C++ 的 RTTI

分类”的列表中选择“C++ Language”，然后选中“允许时间类型信息[RTTI]”，见图 5-7 中的圈释。也就是说，如果一个程序中包含 `dynamic_cast` 或 `typeid` 关键字，在 Visual C++ 6.0 环境中会产生错误，需要用上述方法启动 Visual C++ 的 RTTI 机制后，程序才能正常运行。

更多编译器在默认设置中已经启动了 RTTI 机制，如 Borland C++ Builder 编译器。Visual C++ 2010 之后的版本，RTTI 机制是默认为可用的。

5.5.1 `dynamic_cast`

`dynamic_cast` 是一个强制类型转换操作符，用于实现多态基类的指针（引用）与派生类指针（引用）之间的转换，分为向上转换和向下转换两种。向上转换是指在类的继承层次结构中，从派生类向基类方向的转换，即把派生类对象的指针或引用转换成基类对象的指针或引用，这种转换常常通过 C++ 的默认方式完成。

`dynamic_cast` 用于程序运行期执行类型转换，与之对应的 `const_cast`、`static_cast` 和 `reinterpret_cast` 强制类型转换则是在编译期完成的。`dynamic_cast` 的用法如下：

```
dynamic_cast<type *>(e)           // 指针转换，e 是指针
dynamic_cast<type &>(e)          // 引用转换，e 必须是左值
dynamic_cast<type &&>(e)         // 右值转换（右值，临时变量），e 不能是左值
```

其中，`type` 必须是类类型，通常情况下 `type` 类型中应该有虚函数。

`dynamic_cast` 把表达式 `e` 转换成 `type` 类型的数据。当 `e` 的类型符合下面三种情况之一时，转换能够成功：
① `e` 的类型是 `type` 的公有派生类；
② `e` 的类型是 `type` 的公有基类；
③ `e` 与 `type` 是同一类类型。
如果指针不能转换成目标类型，转换失败，`dynamic_cast` 将返回 `null` (0) 指针值；如果引用转换失败，`dynamic_cast` 将引发异常 `std::bad_cast`，可以在 `try-catch`（见第 8 章）中处理。

【例 5-12】 用 `dynamic_cast` 实现向上强制转换和向下强制转换。

```
// Eg5-12.cpp
#include<iostream>
using namespace std;

class Base {
public:
    virtual ~Base() {}
```

```

};
class Derived :public Base {
    void f() { cout<<"f in Derived!\n"<<endl; }
};
void main() {
    Base *pb, b;
    Derived d, *pd = &d;
    pb = &d; // 默认转换, 编译时完成, 是常用方式
    pb = dynamic_cast<Base *>(&d); // 向上转换, 运行时完成
    pb = dynamic_cast<Base *>(pd); // 向上转换, 运行时完成
    pb = &b;
    pd = dynamic_cast<Derived *>(pb); // L1, 向下强制转换
    if (pd)
        cout<<"ok";
    else
        cout<<"error!\t";
    pd = dynamic_cast<Derived *>(&b); // L2, 向下强制转换
    if (pd)
        cout<<"ok";
    else
        cout<<"error!\t";
    pb = &d;
    pd = dynamic_cast<Derived *>(pb); // L3, 向下强制转换
    if (pd)
        cout<<"ok!";
    else
        cout<<"error!\t\n";
}

```

程序的运行结果如下:

```
error!   error!   ok!
```

这个结果表明语句 L1、L2 的转换是失败的，L3 的转换才是成功的。按照 C++ 的规则，语句 L1 和 L2 的 `dynamic_cast` 向下强制转换是不安全的，L3 是安全的。原因是，语句 L1 和 L2 转换的基类指针 `pb` 指向的 `b` 都是基类 `Base` 的一个对象，而基类对象 `b` 对派生类中增加的成员不知道，不能通过它转换出那些在派生类中新增的成员。换句话说，语句 L1 中的 `dynamic_cast` 根本不能从 `pb` 指向的对象 `b` 中转换出在派生类 `Derived` 中新增的成员函数 `f()`，因为 `b` 只是一个 `Base` 对象。语句 L3 则不一样，`pb` 虽然是一个基类 `Base` 的指针，但实际指向的是派生类 `Derived` 的一个对象，所以 `pb` 能够被转换成派生类类型对象的指针。

说明：

- ① 在用 `dynamic_cast` 进行基类与派生类指针或引用之间的转换时，基类必须是多态的，即基类至少有一个虚函数。
- ② 只有在支持 RTTI 的程序环境中，才能使用 `dynamic_cast` 进行类型转换。
- ③ 在向下强制类型转换时，只有当基类类型的指针或引用实际指向了一个派生类对象时，`dynamic_cast` 才能将它们转换成派生类对象的指针或引用，否则转换不会成功。

在类的继承结构中，在默认情况下，当用基类类型的指针（引用）操作派生类对象时，只能通过该指针（引用）访问派生类中对基类虚函数的覆盖函数版本。那些在基类中没有被

定义为虚函数或派生类新增加的函数，通过基类指针是无法访问的。

【例 5-13】 有 3 个类，B 是 D1 和 D2 的基类，通过 B 的指针能够访问派生类的虚函数 f()。

```
// Eg5-13.cpp
#include <iostream>
#include <typeinfo>
using namespace std;

class B {
    int x;
public:
    B(int i){ x = i; }
    int getx(){ return x; }
    virtual void f(){ cout<<"1: 基类 B 中的 f(), x = "<<x<<endl; }
};

class D1:public B {
    int x;
public:
    D1(int i):B(i) { }
    virtual void f() { cout<<"2: 派生类 D1 中的 f(), x = "<<getx()<<endl; }
};

class D2:public B {
    int x;
public:
    D2(int i):B(i) { }
    virtual void f() { cout<<"3: 派生类 D2 中的 f(), x = "<<getx()<<endl; }
    void g() { cout<<"4: 这是派生类 D2 特有的函数，其他类都没有! \n"; }
};

class D3 :public D2,public D1 {
    int z;
public:
    D3(int i):D2(i), D1(i) { z = i; }
};

void AccessB(B *pb){
    pb->f();
    // pb->g(); // L1, B 中没有定义 g()为虚函数，不能访问
}

void main() {
    B b(1);
    D1 d1(2);
    D2 d2(3);
    D3 d3(4);
    // B b1(d3); // L2, 转换会失败，B 不明确
    AccessB(&b);
    AccessB(&d1);
    AccessB(&d2);
}
```

被注释的 L2 语句会导致 d3 向 B 的转换失败，原因是 d3 中存在来源于 D1、D2 的两个基类 B，编译器无法确定用哪个 B 进行转换。本程序的运行结果如下：

- 1: 基类 B 中的 f, x = 1
- 2: 派生类 D1 中的 f, x = 2
- 3: 派生类 D2 中的 f, x = 3

这表明,函数 `AccessB(B *)`通过基类类型的指针访问到了派生类对象 `d1` 和 `d2` 中的虚函数 `f()`。

函数 `AccessB()`中的语句 “`pb->g();`” 若不被注释, 将引发一条编译错误。因为 `pb` 是指向基类对象 `B` 的指针, 通过它只能访问那些在类 `B` 中已经定义的函数, 但 `g()`在基类 `B` 中没有被定义, 它是派生类 `D2` 定义的成员函数, 通过 `pb` 无法找到该函数, 所以出错。

由本例可知, 通过基类指针不能访问派生类新增加的成员函数, 因为这些函数在基类中并没有定义。但在某些情况下, 为了完成一些特定的程序任务, 或者出于某种目的, 确实需要通过基类指针访问派生类的新增成员函数, 而且这种向下强制转换也是安全的, 就可以通过 `dynamic_cast` 来实现这样的转换需求。

例如, 在本程序的函数 `AccessB()`中, 当 `pb` 指向派生类 `D2` 的对象时, 需要访问 `D2` 类的成员函数 `g()`; 当 `pb` 指向其他类的对象时, 访问虚函数 `f()`。像这样的情况可以利用 `dynamic_cast`, 将函数 `AccessB()`改写为如下形式, 其余程序代码不做任何修改, 就能够通过基类类型 `B` 的指针 `pb` 访问到派生类 `D2` 新增加的成员函数 `g()`。

```
void AccessB(B *pb) {
    D2 *p = dynamic_cast<D2 *>(pb);
    if(p)                                     // 如果转换成功, 就调用 p->g()
        p->g();
    else                                       // 如果转换不成功, 就调用 pb->f()
        pb->f();
}
```

当将 `AccessB()`改写为上述形式后, 例 5-12 的运行结果如下。

- 1: 基类 B 中的 f, x=1
- 2: 派生类 D1 中的 f, x=2
- 4: 这是派生类 D2 特有的函数, 其他类都没有!

这表明, 当传递派生类 `D2` 的对象给基类指针 `pb` 时, `dynamic_cast` 将 `pb` 转换成了一个 `D2` 类型的指针, 并把它赋值给了 `D2` 类型的指针 `p`。

5.5.2 typeid

在具有多态的 C++程序中, 基类对象的指针或引用可以绑定到继承结构中的任何一个派生类对象上, 因而引发了基类指针或引用的不确定性问题。即, 并非任何时候都能够确定基类指针或引用实际指代的数据类型, 它可能绑定到了某个基类对象, 也可能绑定到了某个派生类对象。在这种情况下, 可以用 `typeid` 操作符在程序运行时判定一个对象的真实数据类型, `typeid` 定义于头文件 `typeinfo` 中, 它的用法如下:

```
typeid(exp)
```

其中, `exp` 可以是任何表达式, 也可以是类对象、指针或引用, `typeid` 操作符返回一个 `type_info` 类的对象引用, `type_info` 类也是在头文件 `typeinfo` 中定义的, 它包含了一个数据类型的许多信息, 该类有一个成员函数 `name()`, 可以用它来获得表达式 `exp` 的类型名称。

【例 5-14】 用 `typeid` 判定数据的类型。

```
// Eg5-14.cpp
#include <iostream>
#include <typeinfo>
using namespace std;

class A {};

void main() {
    A a, *p;
    A &rA = a;
    cout<<"1: "<<typeid(a).name()<<endl;
    cout<<"2: "<<typeid(p).name()<<endl;
    cout<<"3: "<<typeid(rA).name()<<endl;
    cout<<"4: "<<typeid(3).name()<<endl;
    cout<<"5: "<<typeid("this is string").name()<<endl;
    cout<<"6: "<<typeid(4+9.8).name()<<endl;
}
```

本程序的运行结果如下：

```
1: class A
2: class A * __ptr64
3: class A
4: int
5: char const [15]
6: double
```

在类继承结构中，当把派生类对象赋值给基类对象，或把基类对象的指针或引用绑定到派生类对象时，如果基类没有虚函数，`typeid` 操作符返回的是基类类型，而不是它们实际所指的派生类类型。但是，如果基类有虚函数，`typeid` 操作符返回的是指针或引用实际所指的类类型。在实际编程时，可以利用 `typeid` 的这个特点，在程序运行时对变量或对象的实际类型进行识别，并针对识别出的类型进行一些特殊处理。此外，`typeid` 在多态中的一个重要用途就是识别多态运行过程中基类指针或引用实际指向的对象类型，并针对识别出的类型做出不同的处理。

【例 5-15】 利用 `typeid` 获取基类指针所指实际对象的类类型，调用不同类中的成员函数。

```
// Eg5-15.cpp
#include <iostream>
#include <typeinfo>
using namespace std;

class B {
    int x;
public:
    virtual void f() { cout<<"1: B::f()"<<endl; }
};

class D1:public B {
public:
    virtual void g() { cout<<"2: D1::g()"<<endl; }
};

class D2:public B {
    int x;
```



```

public:
    virtual void f() { cout<<"3: D2::f() "<<endl; }
    void h() { cout<<"4: D2::h()\n"; }
};

void AccessB(B *pb) {
    if (typeid(*pb) == typeid(B))
        pb->f();
    else if (typeid(*pb) == typeid(D1)) {
        D1 *pd1 = dynamic_cast<D1 *>(pb);
        pd1->g();
    }
    else if (typeid(*pb) == typeid(D2)) {
        D2 *pd2 = dynamic_cast<D2 *>(pb);
        pd2->h();
    }
}

void main() {
    B b;
    D1 d1;
    D2 d2;
    AccessB(&b);           // 输出:      1: B::f()
    AccessB(&d1);          // 输出:      2: D1::g()
    AccessB(&d2);          // 输出:      4: D2::h()
}

```

5.6 编程实作：多态编程应用

在本书 4.10 节设计的课程体系继承结构中，设计了 comFinal、Account、Chemistry 三个类，这些类的相关头文件 comFinal.h、account.h、chemistry.h，以及类的实现文件 comFinal.cpp、account.cpp、chemistry.cpp，都保存在目录 C:\course 中。

【例 5-16】 现对 4.10 节的编程实例进行完善，将 comFinal、Account、Chemistry 中的成员函数 show() 设置成虚函数，并设计一个访问该类继承结构的接口函数 display()，通过基类 comFinal 类型的指针，访问派生类 Account 和 Chemistry 重定义的虚函数 show()。

图 5-8 是类继承层次中的虚函数设置情况，实现该继承结构的多态编程过程如下。

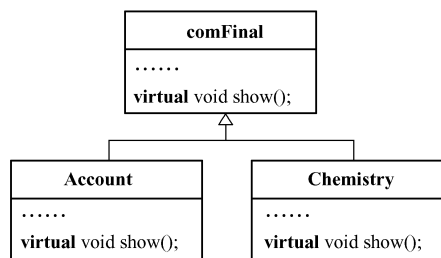


图 5-8 类继承层次中的虚函数

<1> 打开 4.10 节建立在目录 C:\course 中的工程项目文件 com_main.dsw。

<2> 在类 comFinal 的成员函数 show() 声明前面加上限定词 virtual：

```

class comFinal {
    .....
    virtual void show();
};

```

除此之外，comFinal、Account、Chemistry 三个类的其他程序代码可不做任何修改。当然，

也可以在类 Account、Chemistry 的函数 show() 声明前面加上限定词 virtual。由于 Account、Chemistry 是 comFinal 的派生类，即使它们的函数 show() 前面没有 virtual，也是虚函数。

<3> 改写主程序。编写访问本课程类继承结构的接口函数 display() 和主函数 main()，为此可以改写原来的主文件 com_main.cpp，内容如下：

```
// com_main.cpp
#include "comFinal.h"
#include "Chemistry.h"
#include "Account.h"
#include <iostream>
using namespace std;

void display(comFinal* p) { p->show(); }

void main() {
    comFinal *a[3]; // a 为基类对象指针的数组
    comFinal c("王十", 78, 78, 76);
    Account a1("张三星", 98, 78, 97, 67, 87);
    Chemistry c1("光红顺", 89, 99, 34, 56, 78);
    a[0] = &c;
    a[1] = &a1;
    a[2] = &c1;
    for(int i = 0; i < 3; i++) {
        cout<<"-----a["<<i<<" ]-----\n";
        display(a[i]);
    }
}
```

编译并运行本程序，将得到如图 5-9 所示的运行结果，这些运行结果是函数 display() 通过基类 comFinal 定义的指针 p 分别调用 comFinal 对象 (a[0])、Account 对象 (a[1]) 和 Chemistry 对象 (a[2]) 得到的。此外，可以通过基类 comFinal 对象的引用实现对派生类对象的访问，为此可将函数 display() 改为如下形式：

```
void display(comFinal &p){ p.show(); }
```

并将 main() 中对函数 display() 的调用改为如下形式：

```
// a[i] 为指针，*a[i] 即所指对象
display(*a[i]);
```

将得到与图 5-9 完全相同的运行结果。

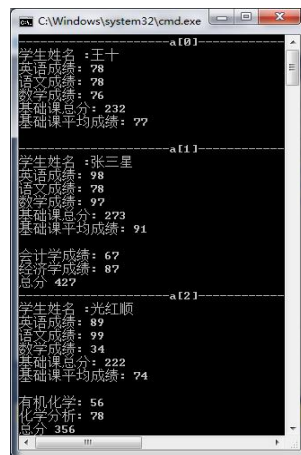


图 5-9 程序运行结果

习 题 5

5.1 解释下述概念：

多态	联编	静态联编	动态联编	虚函数
纯虚函数	抽象类	RTTI	工厂模式	

5.2 虚函数有什么特点？C++ 是如何实现虚函数的动态绑定的？

5.3 抽象类有何特点和作用？

5.4 指出下面程序段中的错误。

```
class B {
public:
    B(int a) { cout<<a<<endl; }
    virtual void f() = 0;
    virtual void f1(int a) { cout<<a; };
    virtual void f2(int a) final { cout<<a; };
    void f3(int c) final { cout<<c; }
    void f4(int b) { cout << b; }
};
class D:public B {
public:
    void f1(int b) override { cout<<b; }
    void f2(int b) override { cout<<b; }
    void f3(int b) override { cout<<b; }
    void g1(char b) override { cout<<b; }
};
void main() {
    D d1;
}
```

5.5 读程序，写出程序运行的结果。

(1)

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

class Base {
protected:
    int n;
public:
    Base(int m) { n = m++; }
    virtual void g1() {
        cout<<"Base::g1() ..."<<n<<endl;
        g4();
    }
    virtual void g2() {
        cout<<"Base::g2() ..."<<++n<<endl;
        g3();
    }
    virtual void g3() {
        cout<<"Base::g3() ..."<<++n<<endl;
        g4();
    }
    virtual void g4() { cout<<"Base::g4() ..."<<++n<<endl; }
};

class Derive:public Base {
    int j;
public:
```

```

    Derive(int n1, int n2):Base(n1){ j = n2; }
    void g1(){
        cout<<"Deri::g1() ..."<<"+n<<endl;
        g2();
    }
    void g3(){
        cout<<"Deri::g2() ..."<<"+n<<endl;
        g4();
    }
};

void main() {
    Derive Dobj(1, 0);
    Base Bobj = Dobj;
    Bobj.g1();
    cout<<"-----"<<endl;
    Base *bp = &Dobj;
    bp->g1();
    cout<<"-----"<<endl;
    Base &bobj2 = Dobj;
    bobj2.g1();
    cout<<"-----"<<endl;
    Dobj.g1();
}

```

(2)

```

#include <iostream>
Using namespace std;

class Shape {
public:
    virtual double area() { return 0; }
    virtual void print()=0;
};

class Circle:public Shape {
protected:
    double r;
public:
    Circle(double x):r(x) {}
    double area() { return 3.14*r*r; }
    void print() { cout<<"Circle : r = "<<r<<"\t area = "<<area()<<endl; }
};

class Cylinder:public Circle {
    double h;
public:
    Cylinder(double r, double x):Circle(r), h(x){ }
    double area(){ return 2*3.14*r*r+2*3.14*h; }
};

void shapeArea(Shape &s) { cout<<s.area()<<endl; }
void shapePrint(Shape *p) { p->print(); }

```

```

void main() {
    Shape *s[3];
    Circle circle(10);
    Cylinder cylinder(20,100);
    s[0] = &circle;
    s[1] = &cylinder;
    for(int i = 0; i < 2; i++) {
        shapeArea(*s[i]);
        shapePrint(s[i]);
    }
}

```

注意：本例有意不在 Cylinder 类中重载纯虚函数 print()，因此需要仔细分析 shapePrint(s[1]) 的输出结论。

(3)

```

#include <iostream>
using namespace std;

class A {
public:
    void virtual f() { cout<<"f() in class A"<<endl; }
};
class B:public A {
public:
    void f() { cout<<"f() in class B"<<endl; }
    void fb() { cout<<"normal function fb \n"; }
};
class C:public A {
public:
    void f() { cout<<"f() in class C"<<endl; }
    void fc() { cout<<"normal function fc"<<endl; }
};

void f(A *p) {
    p->f();
    if(typeid(*p) == typeid(B)) {
        B *bp = dynamic_cast<B*>(p);
        bp->fb();
    }
    if (typeid(*p) == typeid(C)){
        C *bc = dynamic_cast<C*>(p);
        bc->fc();
    }
}

void main() {
    A *pa;
    B b;
    C c;
    pa = &b;
    f(pa);
    pa = &c;
    f(pa);
}

```

5.6 用抽象类设计计算二维平面图形面积的程序，在基类 TDshape 中设计纯虚函数 area()和 printName()。area()用于计算几何图形的面积，printName()用于打印输出几何图形的类名，如类 Triangle 的对象就打印输出“Triangle”。每个具体形状类则从抽象类 TDshape 派生，各自需要定义其独有的数据成员和成员函数，并定义 area()和 printName()的具体实现代码，如图 5-10 所示。

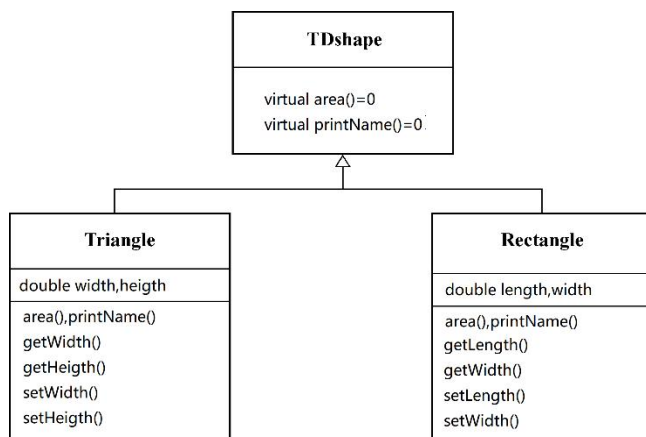


图 5-10 题 5.6 图

要求：编写以 TDshape 为接口的函数，借以访问具体类，如 Triangle 类和 Rectangle 类的成员函数 area()和 printName()。

5.7 某公司有总经理 CEO、雇员 Employee、小时工 HourlyWorker 和营销人员 CommWorker，他们的薪酬计算方法如下：总经理实行年薪制，一年 15 万元；雇员按月计酬，方法是基本工资+奖金；小时工按工作时间计酬，方法是工作小时×每小时单价；营销人员按月计酬，方法是基本工资+销售利润×5%。

每类人员都有姓名、职工编号、年龄、性别、工资等数据。设计计算各类人员报酬的程序，用虚函数 getWage()计算各类人员的应得报酬，用虚函数 print()打印输出各位工作人员的基本数据。

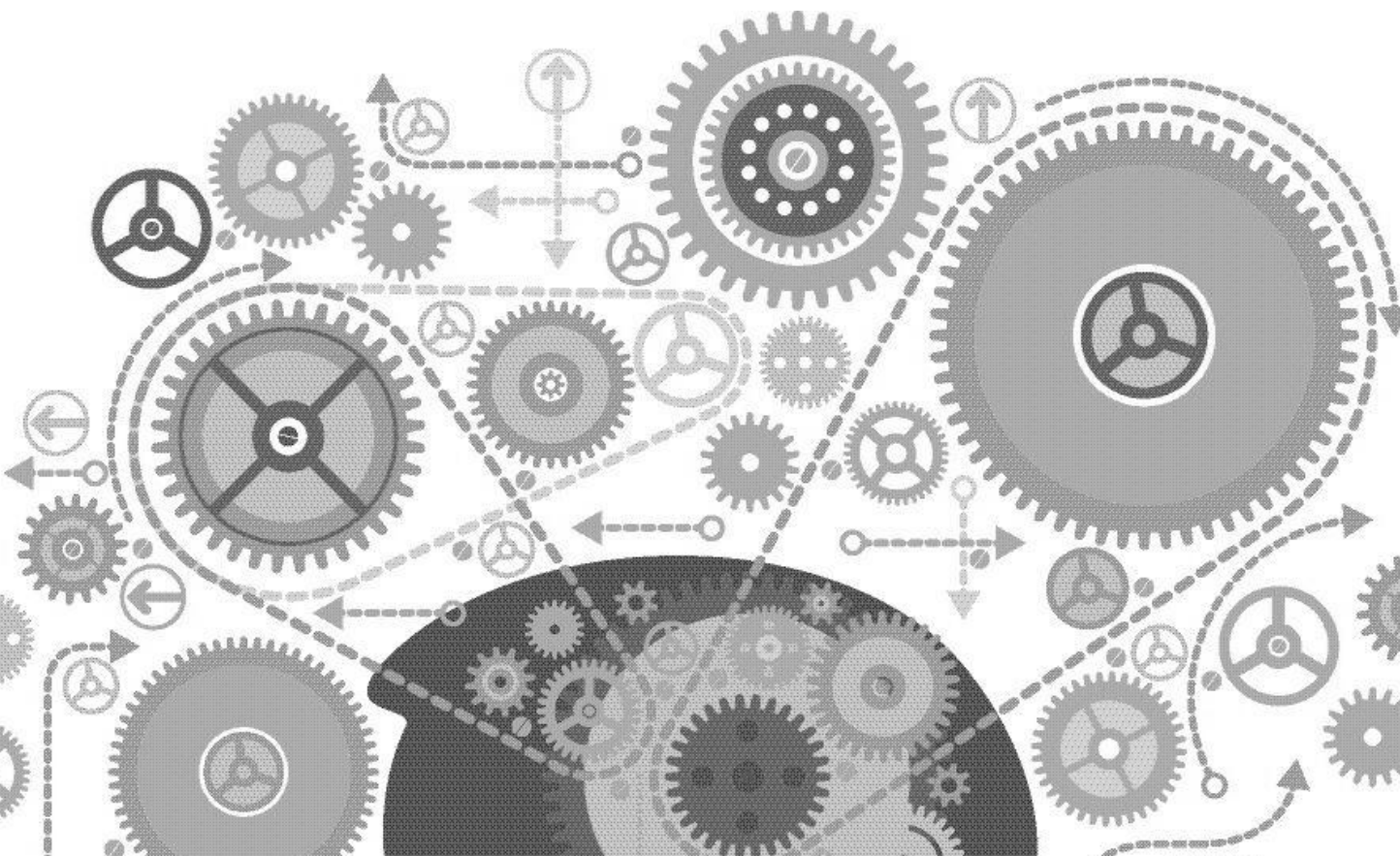
提示：将各类人员都共有的属性和行为抽象在类 Person 中，包括姓名、职工编号、年龄、性别等，以及成员函数 getWage()和 print()。将 getWage()设计为纯虚函数，将 print()设计为一般虚函数，其余类从类 Person 派生，各类再定义 getWage()的实现方法，并重载函数 print()输出具体数据。此外，每个类需根据实际情况定义相应的成员函数，获取诸如工作时间、基本工资、销售利润之类的基础数据。

第 6 章

运算符重载

运算符重载是 C++ 语言的一项强大功能。通过重载，可以扩展 C++ 运算符的功能，使它们能够操作用户自定义的数据类型，用运算符写出简洁的代码，增强程序代码的直观性和可读性。

本章介绍 C++ 运算符重载的相关内容，包括：以类成员函数、友元和普通函数方式进行运算符重载的方法，输入、输出运算符及一些特殊运算符（如++、--、[]、()等）的重载方法。



6.1 运算符重载基础

C++语言有丰富的运算符，每个运算符都能够操作多种数据类型。例如，如下三条语句

```
int i = 2+3;
double j = 2+4.8;
float f = float(3.1) + float(2.0);
```

都用到了加法运算符“+”，但每次运算的数据类型并不相同，因而不能用一个完全相同的函数实现这三次运算。实际上，C++语言是通过运算符重载来实现上述功能的。

运算符实际是一种特殊的函数，称为**运算符函数**。运算符重载只不过是一种特殊的函数重载，它的命名规则和参数确定不同于普通的函数重载，有特殊的函数命名方式和固定不变的参数个数。就加法运算符而言，C++语言重载了多个加法运算符函数，每个重载的“+”都能够实现不同类型数据的加法运算。例如，对于上面的三个加法表达式，C++语言提供了类似以下形式的运算符重载函数：

```
int operator+(int, int);
double operator+(int, double);
float operator+(float, float);
```

其中，`operator` 是 C++语言规定的每个运算符函数必有的限定词，指明它是一个运算符函数。`operator+` 就是加法运算符“+”的函数名，紧接在“+”后面的就是函数的参数表。

C++语言为每个运算符都提供了多个重载函数，它们的功能相同，但用于操作不同的内置数据类型（如 `int`、`float`、`char`、`double` 等）。例如，“+”能够完成 `int`、`float`、`char`、`double` 等数据类型的加法运算，甚至允许在同一个加法表达式中出现不同的数据类型，它们都是通过加法运算符的重载函数实现的。

应用运算符能够编写出简洁的表达式、清晰而高效的程序代码。为了方便自定义数据类型（如自定义 `class` 或 `struct`）的各种运算，通过重载的方式，C++允许扩展运算符的功能，使重载后的运算符能够对自定义数据类型进行运算。

例如，设有自定义的复数类 `Complex`，其形式如下：

```
class Complex {
    double real, image;
public:
    .....
};
```

定义了如下复数对象，并且需要实现两个复数相加的运算：

```
Complex c1, c2, c3;
.....
c1 = c2+c3;
```

当 C++编译器遇到“`c2+c3`”表达式时，将产生编译错误。因为 `Complex` 是程序员自定义的数据类型，C++并未定义它的加法运算，所以“+”不能实现两个 `Complex` 数据相加的功能。

但是，C++允许程序员重载加法运算符“+”，扩展它的功能。即，在不改变“+”对系统内置数据类型相加的既有功能基础之上，增加对 `Complex` 类型数据相加的功能。

重载方法如下：

```
Complex operator+(Complex c1, Complex c2) {...}
```

1. 运算符重载限制

为了使运算符重载后不影响其原有功能的正常运行，C++对重载运算符进行了一些限制。

1) 可以重载的运算符

只有 C++预定义操作符集合中的运算符才能够被重载，这些运算符如下：

+	-	*	/	%	^	&		~
!	,	=	<	>	<=	>=	++	--
<<	>>	==	!=	&&		+=	-=	/=
%=	^=	&=	=	*=	<<=	>>=	[]	()
->	->*	new	new[]	delete	delete[]			

2) 不能被重载的运算符

. .* :: ?: sizeof typeid alignof(类型的内存对齐)

3) 只能被重载为类成员函数的运算符

运算符函数可以被重载为类的非静态成员函数、类的友元函数，或普通函数。但 C++规定，以下运算符只能被重载为类的非静态成员函数：

= [] () ->

4) 运算符重载的基本原则

- ① 不能改变运算符的优先级。
- ② 不能改变运算符的结合顺序（如+、-、*、/ 按照从左到右结合运算，这个顺序不能改变）。
- ③ 重载运算符不能使用默认参数。
- ④ 不能改变运算符所需要的参数个数。
- ⑤ 不能创造新运算符，只能重载系统已有的运算符。
- ⑥ 不要改变运算符的原有含义。
- ⑦ 若运算符被重载为类的成员函数，则只能是非静态的成员函数。

2. 运算符重载的语法

运算符的运算结果通常为值类型，所以其重载函数常返回值类型的数据，语法形式如下：

```
返回类型 operator@(参数表)
```

其中，operator 是 C++的保留关键字，表示运算符函数；@代表重载运算符，可以是前面列举的可重载运算符中的任何一个。

在重载运算符时，参数表中的参数个数必须与运算符实际需要的操作数个数相符合。例如，两个整数相减必须有且只有两个参数，其对应的运算符函数 operator-()可表示为

```
int operator-(int a, int b) { return a-b; }
```

而整数取反的运算符函数 operator-()只能有一个参数，类似于如下形式：

```
int operator-(int a){ return -a; }
```

3. 与类相关的运算符重载方式

类是使用最多的自定义数据类型，C++为类提供了以下默认运算符：赋值运算 (=)，取

类对象地址的运算符 (&), 成员访问运算符 (如 “.” 和 “->”)。这些运算符不需要重载就可以使用。但是, 如果要在类中使用其他运算符, 就必须明确地重载它们。

1) 非静态成员函数重载运算符

如果运算符作为类的成员函数重载, 其参数个数要比该运算符实际的参数个数少一个。原因是类成员函数的第一个参数是由系统自动添加的 **this** 指针, 由 C++ 编译系统自动处理, 不需要显式指定。而静态成员函数没有 **this** 指针, 不能传递第一个运算数, 所以 C++ 标准不允许将运算符重载为类的静态成员函数。

例如, 在复数类 **Complex** 中重载加法运算符, 以成员函数重载的形式如下:

```
class Complex {
    double real, image;
public:
    Complex operator+(Complex b){...}
    .....
};
```

2) 友元或普通函数重载运算符

如果将运算符重载普通函数或类的友元函数, 其参数个数与运算符实际需要的参数个数相同。例如, 用友元和普通函数重载复数类 **Complex** 的加法运算符的形式如下:

```
class Complex {
    .....
    friend Complex operator+(Complex a, Complex b);    // 友元声明
    .....
};
Complex operator+(Complex a, Complex b) {...}        // 友元定义
Complex operator-(Complex a, Complex b) {...}        // 普通函数
```

友元与普通函数的区别在于, 友元可以直接访问类的私有成员, 而普通函数只能通过类的公有成员函数访问其私有成员。

3) 运算符重载为成员函数与非成员函数的选择

如上所述, 除了=、[]等运算符只能重载为类的成员函数, 有些运算符既可以用类成员函数方式重载, 也可以用普通函数方式重载, 到底用哪种方式重载更好呢?

一般而言, 复合赋值运算符 (如+=、-=、*=、/=等) 通常应该重载为类成员函数, 但不是必须这样做 (与 “=” 不同); 另外, 对于要改变对象状态的运算符, 或者与运算对象密切相关的运算符, 如++ (自增)、-- (自减)、解引用运算符, 也适合重载为类成员函数。

算术运算 (+、*、/、-等)、相等与否的比较、关系运算、位运算等运算符具有对称性, 通常允许运算符左、右两边的对象进行交换或类型转换, 则适宜重载为非成员函数。

6.2 重载二元运算符

6.2.1 类与二元运算符重载

二元运算符是指需要两个操作数的运算符, 又称为双目运算符, 如+、-等。其常用形式如下:

a @ b

其中，@代表任一可重载的二元运算符，a和b代表两个参与运算的操作数，如1+3、3*x等。上面的调用形式将被C++解释为以下两种运算符函数调用形式之一：

```
a.operator@(b)
operator@(a, b)
```

第一种形式是@被重载为类的非静态成员函数的解释方式，这种方式要求运算符@左边的参数（第一个参数）必须是一个对象，operator@()是该对象的成员函数。第二种形式是@作为类的友元或普通重载函数时的解释方式。

二元运算符作为类的非静态成员函数、友元、普通函数重载时的区别如下：

① 以非静态成员函数方式重载二元运算符时，只能有一个参数，该参数实际上是函数的第二个参数（运算符右边的操作数），其第一个参数（运算符左边的操作数）由C++通过this指针隐式传递，而作为普通函数和类的友元函数重载时需要两个参数。

② 调用类的重载运算符时，作为类成员函数重载的运算符，要求左操作数必须是一个类对象，而作为友元或普通函数重载的运算符则无此限制。

重载二元运算符为非静态成员函数的形式如下：

```
class X {
    .....
    T1 operator@(T2 b) {                // 实际为 T1 operator@(X *this, T2 b)
        .....                          // 其中 X *this 形参由编译器自动添加
    }
}
```

其中，T1是运算符函数的返回类型，T2是参数的类型，原则上T1、T2可以是任何数据类型，但实际上它们常与X相同。

重载二元运算符为类的友元函数或普通函数时需要两个参数，其形式如下：

```
class X {
    .....
    friend T1 operator(T2 a, T3 b);
}
T1 operator(T2 a, T3 b) {...}          // 友元函数定义
T1 operator(T2 a, T3 b) {...}          // 普通函数
```

T1、T2、T3可以是不同的数据类型（通常都是X）。友元与一般函数重载的区别是：友元可以访问类的任何数据成员，而普通函数只能访问类的public成员。

【例6-1】设计复数类Complex，利用成员运算符函数重载实现复数的加、减运算，用友元运算符函数重载实现其乘、除等复数运算。

```
// Eg6-1.cpp
#include<iostream>
using namespace std;

class Complex {
private:
    double r, i;
public:
    Complex (double R = 0, double I = 0):r(R), i(I){};
```

```

Complex operator+(Complex b);           // L1, 复数加法
Complex operator-(Complex b);           // L2, 复数减法
friend Complex operator*(Complex a, Complex b); // L3, 复数乘法
friend Complex operator/(Complex a, Complex b); // L4, 复数除法
void display();
};
Complex Complex::operator +(Complex b){ return Complex(r+b.r, i+b.i); }
Complex Complex::operator -(Complex b) { return Complex(r-b.r, i-b.i); }
Complex operator *(Complex a, Complex b) {
    Complex t;
    t.r = a.r*b.r-a.i*b.i;
    t.i = b.r*b.i+b.i*b.r;
    return t;
}
Complex operator/(Complex a, Complex b) {
    Complex t;
    double x;
    x = 1/(b.r*b.r+b.i*b.i);
    t.r = x*(a.r*b.r+a.i*b.i);
    t.i = x*(a.i*b.r-a.r*b.i);
    return t;
}
void Complex::display() {
    cout<<r;
    if (i > 0)
        cout<<"+";
    if (i != 0)
        cout<<i<<"i"<<endl;
}
void main(void) {
    Complex c1(1, 2), c2(3, 4), c3, c4, c5, c6;
    c3=c1+c2;
    c4=c1-c2;
    c5=c1*c2;
    c6=c1/c2;
    c1.display();
    c2.display();
    c3.display();
    c4.display();
    c5.display();
    c6.display();
}

```

程序的运行结果如下:

```

1+2i
3+4i
4+6i
-2-2i
-5+24i
0.44+0.08i

```

对于程序中的运算符调用：

```
c3 = c1+c2;
c4 = c1-c2;
```

C++会将它们转换成如下形式的调用语句：

```
c3 = c1.operator+(c2);
c4 = c1.operator-(c2);
```

虽然这两次函数调用只提供了一个参数 `c2`，但实际上是两个参数，其左参数虽然没有出现在参数表中，但编译器会通过 `c1` 对象的 `this` 指针传递该参数（`c1`）。

而 `c5` 和 `c6` 的计算是通过友元 “*” 和 “/” 重载实现的：

```
c5 = c1*c2;
c6 = c1/c2;
```

C++编译器会将它们转换成如下函数调用形式：

```
c5 = operator*(c1, c2);
c6 = operator/(c1, c2);
```

概括而言，可以用下面两种方式调用以类成员函数方式重载的二元运算符：

```
a @ b;           // 隐式调用二元运算符@
a.operator@(b)    // 显式调用二元运算符@
```

友元运算符函数的调用也有下面两种形式：

```
a@b;           // 隐式调用二元运算符@
operator@(a, b) // 显式调用二元运算符@
```

在上面的程序中，若将主函数 `main()` 中对 `c3`、`c4`、`c5`、`c6` 的计算语句改为下列显式调用方式，将得到完全相同的运行结果：

```
c3 = c1.operator+(c2);
c4 = c1.operator-(c2);
c5 = operator*(c1, c2);
c6 = operator/(c1, c2);
```

6.2.2 非类成员方式重载二元运算符的特殊用途

对于不要求返回左值且可以交换操作数左右次序的运算符（如+、-、*、/等），最好用非类成员函数方式（包括友元和普通函数）重载它，其中的原因如下。

在调用重载的二元运算符函数时，如果实参与形参的类型不匹配，对于以类成员方式重载的运算符函数，编译器只对第二个参数进行所有可能的隐式类型转换，不对第一个参数进行类型转换；对于以非类成员函数（包括友元和普通函数）方式重载的运算符函数，编译器对两个参数都会进行可能的隐式类型转换。鉴于此，最好将二元运算符重载为普通函数（包括友元），以实现二元运算的对称性。例如，下面对例 6-1 设计的类 `Complex` 的应用中，语句 L2 是错误的。

```
void main() {
    Complex c1, c2(1, 2);
    c1 = c2+2;           // L1
    c1.display();
}
```

```

        c1 = 2+c2;                                // L2
        c1.display();
    }

```

表达式“c2+2”是正确的。虽然例 6-1 中 `Complex` 的成员函数 `operator+()` 要求“+”的两个参数都是 `Complex` 类型的对象，而“c2+2”的第二个参数是 `int` 类型，但 C++ 会调用默认构造函数

```
Complex(double R = 0, double I = 0):r(R), i(I) { }
```

将“2”转换成一个 `Complex` 对象。所以，“c2+2”等效于如下表达式：

```
c2 + Complex(2, 0)
```

转换之后，就符合函数 `Complex::operator+(Complex)` 的参数需求了，因此是正确的。

而表达式“2+c2”对于例 6-1 而言是错误的。因为 `Complex` 的 `operator+()` 运算符函数要求参加加法运算的两个参数是 `Complex` 对象，而“2+c2”中的 2 不是 `Complex` 对象。由于 2 是“2+c2”表达式的第一个参数，而在例 6-1 中，`operator+()` 是以成员函数方式重载的，C++ 不会对它进行任何形式的隐式类型转换，因此是错误的。当然，可以采用如下形式进行显式转换。

```
Complex(2, 0) + c2
```

但是，如果例 6-1 中的 `operator+()` 运算符函数是 `Complex` 类的友元或普通重载函数，当提供给它的第一个参数类型不匹配时，C++ 就会对它进行可能的隐式转换。由于“2+c2”的第一个参数不符合 `operator+()` 的第一个参数要求，C++ 会调用 `Complex` 的构造函数对它进行隐式类型转换，转换结果如下：

```
Complex(2, 0) + c2
```

因此，当 `operator+()` 被重载为 `Complex` 类的友元运算符函数或普通运算符函数时，“2+c2”表达式是正确的。

在设计 C++ 程序时，像“2+c2”和“c2+2”之类的对称运算表达式最好通过友元或普通函数重载运算符来实现，如下例所示。

【例 6-2】 用友元运算符重载实现复数与实数的加法运算。方法是：实数与复数的实部相加，复数的虚部保持不变。

```

// Eg6-2.cpp
#include <iostream>
using namespace std;

class Complex {
private:
    double r, i;
public:
    Complex(double R = 0, double I = 0) :r(R), i(I) { }
    friend Complex operator+(Complex a, double b) { return Complex(a.r + b, a.i); }
    friend Complex operator+(double a, Complex b) { return Complex(a + b.r, b.i); }
    void display();
};

void Complex::display() {
    cout<<r;
    if (i > 0)

```



```

        cout<<" ";
    if (i != 0)
        cout<<i<<"i"<<endl;
}

void main(void) {
    Complex c1(1, 2), c2;
    c2 = c1 + 5;
    c2.display();           // 输出: 6+2i
    c2 = 5 + c1;
    c2.display();           // 输出: 6+2i
}

```

6.3 重载一元运算符

一元运算符只需要一个运算参数，如求相反（-）、自增加（++）等。其常见调用形式为

或

```

@a
a@

```

其中，@代表一元运算符，a代表操作数。@a代表前缀一元运算，如“++a”；a@表示后缀运算，如“a++”等。@a调用将被C++解释为如下形式之一：

```

a.operator@()
operator@(a)

```

前者是一元运算符作为类成员函数重载时的函数形式，后者是作为非类成员（如友元或普通函数）重载时的函数形式。以类成员函数方式重载的一元运算符不需要参数，而以非类成员函数方式重载时则需要一个参数。

6.3.1 作为成员函数重载

一元运算符作为类成员函数重载时不需要参数，其形式如下：

```

class X {
    .....
    T operator@() {...}           // 等价于 T operator@(X *this), this 指针参数由系统自动添加
}

```

T是运算符@的返回类型。形式上，作为类成员函数重载的一元运算符没有参数，但实际上它包含了一个隐含参数，即由编译器自动添加的this指针。

【例 6-3】 设计一个时间类 Time，能够完成秒钟的自增运算。

```

// Eg6-3.cpp
#include<iostream>
using namespace std;

class Time {
private:
    int hour, minute, second;
public:

```

```

    Time(int h, int m, int s);
    Time& operator++();
    void display();
};
Time::Time(int h, int m, int s) {
    hour = h;
    minute = m;
    second = s;
    if(hour >= 24)                                     // 若初始小时大于等于 24, 重置为 0
        hour = 0;
    if(minute >= 60)                                   // 若初始分钟大于等于 60, 重置为 0
        minute = 0;
    if(second >= 60)                                  // 若初始秒钟大于等于 60, 重置为 0
        second = 0;
}
Time& Time::operator ++() {
    ++second;
    if(second >= 60) {
        second = 0;
        ++minute;
        if(minute >= 60) {
            minute = 0;
            ++hour;
            if(hour >= 24)
                hour = 0;
        }
    }
    return *this;
}
void Time::display() {
    cout<<hour<<":"<<minute<<":"<<second<<endl;
}

void main(){
    Time t1(23, 59, 59);
    t1.display();
    ++ ++t1;                                           // 连续自增, 隐式调用方式
    t1.display();
    t1.operator ++();                                  // 显式调用方式
    t1.display();
}

```

本程序的运行结果如下:

```

23:59:59
0:0:1
0:0:2

```

以类成员方式重载的一元运算符函数, 也有下面两种调用方式:

```

@a;                                                    // 隐式调用一元运算符@
a.operator@()                                          // 显式调用一元运算符@

```

@代表所有重载为类成员函数的一元运算符。像++、--这样能够实现连续自增、自减的运算符，其重载函数应该返回对象的引用，否则不能实现对象的连续运算。

6.3.2 作为友元函数重载

用友元函数重载一元运算符时需要一个参数。如在例 6-4 中，将++运算符重载为类 Time 的友元函数的情况如下。

【例 6-4】 用友元重载类 Time 的自增运算符++。

```
// Eg6-4.cpp
class Time {
    .....
    friend Time& operator++(Time &t);
};
Time& operator ++(Time &t) {
    ++t.second;
    if(t.second >= 60) {
        t.second = 0;
        ++t.minute;
        if(t.minute >= 60) {
            t.minute = 0;
            ++t.hour;
            if(t.hour >= 24)
                t.hour = 0;
        }
    }
    return t;
}

void main() {
    Time t1(23, 59, 59);
    t1.display();
    ++++t1; // 隐式调用方式
    t1.display();
    operator++(t1); // 显式调用方式
    t1.display();
}
```

本程序中略掉的程序代码与例 6-3 完全相同，程序的运行结果也与例 6-3 完全相同。用友元重载一元运算符时，同样有两种调用运算符函数的方式。

```
@a; // 隐式调用一元运算符@
operator@(a) // 显式调用一元运算符@
```

对于像++、--这样的特殊一元运算符，运算结果会影响操作数自身，当以友元或普通函数方式重载这类运算符时，应该将参数设置为引用类型。此外，为了实现连续的自增、自减运算，还需要像例 6-3 和例 6-4 那样，让重载运算符函数返回对象的引用。如果传递值参数，就不能修改实参对象的值；如果返回值，就不能实现连续运算。比如，将例 6-4 中的运算符函数 operator++()改为如下重载形式：

```

class Time {
    .....
    friend Time operator++(Time t);
};
Time operator++(Time t) {
    .....
    return t;
}
.....

void main() {
    Time t1(23, 59, 59);
    t1.display();
    ++ ++ t1;
    t1.display();
    operator++(t1);
    t1.display();
}

```

经此修改后，本程序的运行结果如下：

```

23:59:59
23:59:59
23:59:59

```

此结果表明，向运算符函数传递值形参不能够修改对象的值，返回值类型的对象不能实现对象的连续运算。

6.4 特殊运算符重载

前面介绍了运算符重载的整体情况和一般规则，适用于 C++ 的绝大多数运算符重载。本节再介绍几个特殊运算符的重载方法。

6.4.1 重载++和--

++和--都有前置和后置两种情况，而且通常需要改变运算对象自身的状态，适合重载为类的成员函数（但并非必须，允许重载为非成员函数）。由于它们都是一元运算符，无论将其重载为类成员运算符函数还是普通函数（友元和一般函数），其前置和后置的重载函数原型都相同，无法区分。例如，对于自增运算符++，重载为类的成员函数时，前自增和后自增都会是如下形式：

```

class X {
    .....
    X& operator++() {...};           // 前自增
    X operator++() {...};           // 后自增
}

```

若将它们重载为友元运算符，都是如下形式：

```

class X {

```

```

        friend X& operator++(X& o);           // 前自增的友元声明
        friend X operator++(X& o);           // 后自增的友元声明
    }

```

C++内置的“++”和“--”前置运算符函数返回的是运算对象的引用，后置运算返回对象的原值（递增、递减之前的值）而非引用。为了保持与它们的一致性，重载的自增、自减前置运算符函数也通常返回对象的引用，后置运算则返回值而非引用（并非必须）。

因此，无论重载为成员函数还是友元函数，前自增和后自增运算的函数原型只有返回类型略有差异，C++将它们视为同一函数，在编译上面的X类时，会产生operator++()函数重定义的编译错误。自减运算符也存在同样的问题。

为了区分类似于++、--这类既可以前置又可以后置的运算符，C++采用了如下方法：如果重载为前置运算符，就采用常规的重载方法；如果重载为后置运算符，就在运算符函数的参数表中增加一个无用的形式参数，其作用是向编译器指明该函数是后置运算。下面是重载为类成员的一元运算符的前置和后置形式：

```

class X {
    .....
    X& operator@() {...};           // 前置
    X operator@(int) {...};         // 后置
}

```

下面是重载为友元和普通函数的形式：

```

class X {
    friend X& operator@(X& o);       // 前置的友元声明
    friend X operator@(X& o, int);    // 后置的友元声明
}

X& operator@(X& o) {...};           // 前置的普通函数重载
X operator@(X& o, int) {...};        // 后置的普通函数重载

```

其中，@代表++或--运算符。可以看出，不管是被重载为类的成员函数，还是被重载为类的友元或普通函数，后置形式的一元运算符都比前置形式的一元运算符多一个形式参数。这个形式参数唯一的作用是告知C++编译器该运算符是后置运算，参数的值没有任何实际意义，所以它常常只有一个类型名，连形式参数的名称也不需要。

【例 6-5】 设计一个计数器 counter，用数据成员 n 保存计数器的值，用类成员函数重载自增运算符实现计数器的自增，用友元重载实现计数器的自减。

```

// Eg6-5.cpp
#include<iostream>
using namespace std;

class Counter {
private:
    int n;
public:
    Counter(int i = 0) { n = i; }
    Counter& operator++();
    Counter operator++(int);
    friend Counter& operator--(Counter &c);
    friend Counter operator--(Counter &c, int);
    void display();
}

```

```

};
Counter& Counter::operator++() {
    ++n;
    return *this;
}
Counter Counter::operator++(int) {
    Counter t(*this);
    n++;
    return t;
}
Counter& operator--(Counter &c) {
    --c.n;
    return c;
}
Counter operator--(Counter &c, int) {
    Counter temp(c);
    c.n--;
    return temp;
}
void Counter::display() {
    cout<<"counter number = "<<n<<endl;
}

void main() {
    Counter a;
    ++a; // 调用 Counter::operator++()
    a.display();
    a++; // 调用 Counter::operator++(int)
    a.display();
    --a; // 调用 operator--(Counter &c)
    a.display();
    a-- ; // 调用 operator--(Counter &c, int)
    a.display();
}

```

程序运行结果如下：

```

counter number = 1
counter number = 2
counter number = 1
counter number = 0

```

6.4.2 下标[]和赋值运算符=

1. 重载下标运算符

C++将数组下标定义为一种运算，用运算符[]表示。在 C/C++中，数组不具有检测下标值范围的功能，在存取数组元素时，不小心就可能造成数组元素的越界访问，产生不正确的程序结果。

在 C++中，可以重载下标运算符[]。重载时可以检查数组的大小，并可在访问数组元素

时检查下标值是否越界，禁止对数组的越界访问，以建立安全的数组。其重载形式如下：

```
class X {  
    .....  
    X& operator[](type n);  
};
```

说明：

① []是一个二元运算符，具有数组名和下标变量两个参数。数组名由编译器通过对象的 **this** 指针隐式传递，在函数参数表中不须提供。因此，在参数表中只需提供一个参数，代表数组的下标，原则上 **type** 可以是任何一种顺序数据类型，如枚举、字符、整型等。

② 因为[]可以同时出现在赋值符“=”的左边和右边，所以重载运算符[]时常返回引用，返回引用的函数可以在赋值符“=”的左边调用。

③ []只能被重载为类的非静态成员函数，不能被重载为友元和普通函数。

【例 6-6】 设计一个工资管理类，能够根据职工的姓名录入和查询职工的工资。

```
// Eg6-6.cpp  
#include <iostream>  
#include <string>  
using namespace std;  
  
struct Person {                                     // 职工信息的基本结构  
    double salary;  
    char* name;  
};  
class SalaryManage {  
    Person* employ;                                // 存放职工信息的数组  
    int max;                                       // 数组下标上界  
    int n;                                       // 数组中的实际职工人数  
public:  
    SalaryManage(int Max = 0) {  
        max = Max;  
        n = 0;  
        employ = new Person[max];  
    }  
    ~SalaryManage() { delete[]employ; }  
    double& operator[](const char* Name) {         // 重载[], 返回引用  
        Person* p;  
        // 如下代码段查找有无姓名与 Name 参数代表的同名职工，如有，就返回其工资，否则建立该职工，存  
        // 入工资，然后返回该职工的工资  
        for (p = employ; p < employ + n; p++) {  
            if (strcmp(p->name, Name) == 0)  
                return p->salary;  
        }  
        p = employ + n++;  
        p->name = new char[strlen(Name) + 1];  
        strcpy(p->name, Name);  
        p->salary = 0;  
        return p->salary;  
    }  
}
```



```

void display() {
    for (int i = 0; i < n; i++)
        cout<<employ[i].name<<" " <<employ[i].salary<<endl;
}
};

void main() {
    SalaryManage s(3);
    s["杜一为"] = 2188.88;
    s["李海山"] = 1230.07;
    s["张军民"] = 3200.97;
    cout<<"杜一为\t"<<s["杜一为"]<<endl;
    cout<<"李海山\t"<<s["李海山"]<<endl;
    cout<<"张军民\t"<<s["张军民"]<<endl;
    cout<<"-----下为 display 的输出-----\n\n";
    s.display();
}

```

本程序的运行结果如下：

```

杜一为      2188.88
李海山      1230.07
张军民      3200.97

```

以下为 display 的输出-----

```

杜一为      2188.88
李海山      1230.07
张军民      3200.97

```

2. 重载赋值运算符=

程序中的赋值语句是通过赋值运算符函数实现的，使用它的场合较多。在设计类时，如果没有为它提供赋值运算符成员函数，编译器会自动为它生成一个默认的赋值运算符函数。如果该类对象不需要分配动态存储空间，默认赋值运算符函数能够正确完成对象的赋值复制。

如果对象构造时分配了动态存储空间，默认赋值运算符函数通常不能正确地进行对象的赋值复制，需要为类重载赋值运算符函数。此外，有时需要通过赋值运算符实现特殊的对象赋值复制操作，也需要重载赋值运算符函数。关于重载赋值运算符函数的详细内容，请参考本书 3.8.1 节。

6.4.3 类型转换运算符

本书 2.9 节介绍了与类无关的数据类型之间的转换方法，这里再介绍 C++ 中类与其他数据类型之间的转换方法。

1. 用构造函数实现类的类型转换

构造函数具有类型转换功能，能够将其他类型的数据转换成类类型的对象。如果类 Y 有一个构造函数接受一个 X 类型的参数，该构造函数就能够把 X 类型的对象转换成 Y 类型的对象。形式如下：

```
X x;
```

```

Y y(x); // 或 Y y=x;
Y y{x}; // C++ 11

```

【例 6-7】 有日期类 `Date`，设计其构造函数，能够将整型数据转换成 `Date` 类的对象。

```

// Eg6-7.cpp
#include <iostream>
using namespace std;

class Date {
private:
    int year, month, day;
public:
    Date(int yy = 1900, int mm = 1, int dd = 1) {
        year = yy;
        month = mm;
        day = dd;
    }
    void Show(){ cout<<year<<"-"<<month<<"-"<<day<<endl; }
};

void main() {
    Date d(2000, 10, 11);
    d.Show();
    d = 2006;
    d.Show();
}

```

程序的运行结果如下：

```

2000-10-11
2006-1-1

```

语句“`d = 2006;`”调用类 `Date` 的构造函数，将整数 2006 转换成类 `Date` 的临时对象，然后将此对象赋值给 `Date` 类型的对象 `d`。此语句等效于语句“`d = Date(2006);`”。

因为只提供了一个参数给 `Date` 的构造函数，所以 `month` 和 `day` 都默认为 1。程序运行结果的第 2 行就是对象 `d` 的成员函数 `show()` 输出的结果。

2. 类型转换运算符

构造函数只能实现其他类型（如 `int`、`float`、`char` 等）向类类型的转换，不能实现类类型向其他类型的转换（如把一个类类型转换成 `int`、`float`、`char` 等类型），如果需要这样的功能，就要重载类型转换运算符。

类型转换运算符是类的一个特殊成员函数，它负责将一个当前类类型的对象转换成基本类型或其他类类型的对象，定义形式如下：

```

class X {
    .....
public:
    operator type() {
        .....
        return type 类型的数据;
    }
}

```

```
};
```

类型转换运算符的功能是将类 X 转换成 type 类型，type 通常是 C++ 的基本数据类型。当然，type 也可以是另一个类类型。定义类型转换运算符时，需要注意以下 3 点：① 必须定义为类的成员函数；② 函数原型中不能指定返回类型，必须返回目标类型 type 定义的数据；③ 能够为一个类定义多个类型转换函数。

【例 6-8】 为圆类 Circle 设计类型转换函数，当将 Circle 对象转换成 int 类型时，返回圆的半径；当将它转换成 double 类型时，返回圆的周长；当将它转换成 float 类型时，返回圆的面积。

```
// Eg6-8.cpp
#include <iostream>
using namespace std;

class Circle {
private:
    double x, y, r;
public:
    Circle(double x1, double y1, double r1) { x = x1; y = y1; r = r1; }
    operator int() { return int(r); }
    operator double() { return 2*3.14*r; }
    operator float() { return (float)3.14*r*r; }
};

void main() {
    Circle c(2.3, 3.4, 2.5);
    int r = c; // 调用 operator int(), 将 Circle 类型转换成 int
    double length = c; // 调用 operator double(), 将 Circle 类型转换成 double
    float area = c; // 调用 operator float(), 将 Circle 类型转换成 float
    double len = (double)c; // 将 Circle 类型对象强制转换成 double
    cout<<r<<endl;
    cout<<length<<endl;
    cout<<len<<endl;
    cout<<area<<endl;
}
```

本程序的运行结果如下，请结合此结果理解每次转换函数的调用情况。

```
2
15.7
15.7
19.625
```

3. 类型转换的二义性问题

无论是定义把其他类型转换成类类型的构造函数，还是定义把类类型转换成其他类型的类型转换运算符函数，都要注意避免转换函数的二义性问题，特别是在定义多个参数都是数值类型的构造函数，或者多个目标类型都是数值类型的类型转换函数时，这个问题尤为突出。

【例 6-9】 类 B 同时设置了 int 和 double 类型参数的构造函数，以及将类 B 转换成 int 和 float 的类型转换函数，容易引发二义性问题。

```
// Eg6-9.cpp
#include <iostream>
using namespace std;

class B {
    double x;
public:
    B(float a = 0) :x(a) { }
    B(double b = 0.0) :x(b) { }
    operator int() { return x; }
    operator float() { return x; }
};

void f(long l) { cout << l << endl; }

void main() {
    B b(4); // L1, 无法确定调用 B::B(float)还是 B::B(double)
    f(b); // L2, 无法确定调用 operator int()还是 operator float()
}
```

语句 L1 和 L2 都会产生二义性问题。在执行语句 L1 时，并没有精确匹配调用参数的构造函数，只有把 int 类型的 4 转换成 float 或 double 才能定义对象 b。但两个转换都是可行的，并无优劣之分^[1]，编译器无法确定调用哪个构造函数，因此产生二义性冲突。同样，L2 处的函数 f()需要 long 类型的参数，实参对象 b 可以转换成 int 或 float 类型，两个转换都可行，编译器也无法确定调用哪个转换函数，因而也会产生二义性问题。

综上所述，在进行类设计时最好不要像类 B 那样，同时为类设置多个内置数值类型的构造函数或类型转换函数，容易产生二义性问题。

6.4.4 仿函数

在设计 C++ 的类时，可以重载函数调用运算符，重载后就可以像调用函数一样使用类对象，因此也被称为**仿函数**。其定义形式如下：

```
class X {
    operator type() {...} // 类型转换运算符
    返回类型 operator( ) (形参表); // 函数调用运算符
    .....
}
```

容易把函数调用运算符与类类型转换运算符混为一谈，因此特意列出，注意区分。

【例 6-10】 设计点类 Point，包含表示坐标位置的数据成员 x、y，重载函数调用运算符，其功能是可以指定参数移动坐标 x、y，或者输出点的坐标值。

```
// Eg6-10.cpp
#include<iostream>
using namespace std;
```

[1] 在 C++ 中，只要是类型转换，编译器就按相同的规则处理，彼此之间并无优先级可言。例如，把 int 转换成 float 与转换成 double 都是同等级别，把 int 转换成 float 并不比转换成 double 有优先权。其他类型之间的转换也是如此。

```

class Point {
public:
    Point(int a = 0, int b = 0):x(a), y(b) { }
    Point &operator()(int dx, int dy) { // 函数调用运算符
        x += dx;
        y += dy;
        return *this;
    }
    Point operator()(int dxy) { // 函数调用运算符
        x += dxy;
        y += dxy;
        return *this;
    }
    void operator()() { // 函数调用运算符
        cout<<"["<<x<<"<<y<<"]"<<endl;
    }
private:
    int x, y;
};

int main(){
    Point pt, pt2(2, 2); // L1, 调用 Point::Point()和 Point::Point(int, int)
    pt = pt2(3, 6); // L2, 调用 Point::operator(int,int)
    pt(); // L3, 调用 Point::operator(), 输出: [5, 8]
    pt2(6); // L4, 调用 Point::operator(int)
    pt2(); // L5, 调用 Point::operator(), 输出: [11, 14]
}

```

程序运行结果如下：

```

[5, 8]
[11, 14]

```

语句 L2、L3、L4、L5 与函数调用的形式没有什么区别，但它们并非函数调用，而是分别调用了重载的函数调用运算符。此外，要注意区分语句 L1 的 `pt2(2, 2)` 和语句 L2 的 `pt2(3, 6)`，虽然它们形式相同，但前者是调用构造函数定义对象，后者使用的是对象的函数调用运算符函数。

说明：① 函数调用运算符只能用非静态成员函数重载；② 函数调用运算符函数可以有任意多个参数，可以重载，但不能有参数默认值。

6.5 输入/输出运算符重载

在默认情况下，输入运算符 `>>` 和输出运算符 `<<` 只能实现 C++ 内置数据类型（如 `int`、`float`、`char` 等）的输入和输出，要实现对自定义数据类型的输入和输出，就需要对它们进行重载。

1. 重载输出运算符 `<<`

输出运算符 `<<` 也称为插入运算符，通过重载，可以实现自定义数据类型的输出。其形式如下：

```
ostream &operator<<(ostream &os, classType object) {
    .....
    os<< ...                                // 输出对象的实际成员数据
    return os;                               // 返回 ostream 对象
}
```

`ostream` 是定义于 `iostream` 头文件中的输出流类，其主要功能是实现数据的输出。重载输出运算符<<就是扩展 `ostream` 类的功能，使它能够输出自定义类型 `classType` 的数据，如类与结构类型的对象等。

<<是二元运算符，其第一个参数是 `ostream` 对象的引用，第二个参数是要输出数据类型的一个对象，返回一个 `ostream` 对象的引用。由于输出运算符不会改变输出对象的值，因此在重载<<时，通常将第二个参数设置为 `const` 类型的引用，可以避免复制实参，提高效率。其形式如下：

```
ostream &operator<<(ostream &os, const classType& object) {
    .....
}
```

由于运算符函数 `operator<<()` 的第一个参数必须是 `ostream` 类对象的引用，因此它不能够被重载为类的成员函数，只能被重载为普通函数（通常为类的友元函数）。因为当 `operator<<()` 被重载为类的成员函数时，其第一个参数必然是通过 `this` 指针传递的当前对象，而不是 `ostream` 类对象的引用。因此，输出运算符<<常被重载为类的友元函数。

2. 重载输入运算符>>

输入运算符>>也称为提取（析取）运算符，用于输入数据。通过输入运算符>>的重载，就能够用它输入用户自定义的数据类型，其重载形式如下：

```
istream &operator>>(istream &is, class_name &object) {
    .....
    is>> ...                                // 输入对象 object 的实际成员数据
    return is;                               // 返回 istream 对象
}
```

输入运算符也是一个二元运算符，其重载运算符函数 `operator>>()` 的第一个参数必须是 `istream` 类对象的引用，所以它不能被重载为类的成员函数，只能被重载为普通函数（通常为类的友元函数）。第二个参数通常是用户自定义类型数据的引用，函数返回 `istream` 类型的引用。

`istream` 也是 `iostream` 头文件中的一个类，其主要功能是实现数据的输入。

3. >>和<<重载的应用

在进行类设计时，如果该类有庞大的数据成员需要输入或输出，可以重载输入输出运算符，以简化该类对象数据成员的输入和输出。

【例 6-11】 有一销售人员类 `Sales`，其数据成员有姓名 `name`、身份证号 `id`、年龄 `age`。重载输入输出运算符实现对类 `Sales` 的数据成员的输入和输出。

```
// Eg6-11.cpp
#include<iostream>
#include<string>
using namespace std;
```

```

class Sales {
private:
    char name[10];
    char id[18];
    int age;
public:
    Sales(char *Name, char *ID, int Age);
    friend ostream &operator<<(ostream &os, Sales &s);           // 重载输出运算符
    friend istream &operator>>(istream &is, Sales &s);           // 重载输入运算符
};

Sales::Sales(char *Name, char *ID, int Age) {
    strcpy(name, Name);
    strcpy(id, ID);
    age = Age;
}

ostream& operator<<(ostream &os, Sales &s) {
    os<<s.name<<"\t";           // 输出姓名
    os<<s.id<<"\t";             // 输出身份证号
    os<<s.age<<endl;             // 输出年龄
    return os;
}

istream &operator>>(istream &is, Sales &s) {
    cout<<"输入雇员的姓名、身份证号、年龄"<<endl;           // 显示输入提示信息
    is>>s.name>>s.id>>s.age;                                     // 数据成员数据输入
    return is;
}

void main() {
    Sales s1((char*)"杜康", (char*)"214198012111711", 40);     // L1
    cout<<s1;                                                     // L2
    cout<<endl;                                                   // L3
    cin>>s1;                                                       // L4
    cout<<s1;                                                     // L5
}

```

程序运行结果如下：

```

杜康    214198012111711    40
输入雇员的姓名、身份证号、年龄
Tom 100 23
Tom    100    23

```

运行结果的第 1 行是语句 L2 的输出结果；第 2、3 行是语句 L4 的结果，其中第 3 行是从键盘输入的数据，第 4 行是语句 L5 产生的输出结果。

6.6 编程实作：运算符重载编程应用

字符串是程序设计时经常用到的数据类型，在传统的 C/C++ 程序设计中，字符类型数据的处理常常通过字符指针或字符串函数来完成，但这些函数在进行字符串的赋值、大小比较等操作时并不方便。标准 C++ 提供了一个 **String** 类，它重载了 +、+=、==、>、>=、<、<= 等运算符函数，使字符串的赋值、连接与大小比较等操作与 C++ 内置的 **int** 和 **float** 等类型的数

据一样简便。要使用标准 C++ 的 String 类，需在程序中“#include <string>”。

下面的例子模拟实现了标准 C++ 中 String 类的部分功能，从中可以了解运算符重载的方法和用途，以及 C++ 标准库中 String 类的强大功能。

1. 编程实例一

【例 6-12】 设计一个字符串类 String，通过运算符重载实现字符串的输入、输出以及 +=、==、!=、<、>、>=、[] 等运算。

```
// Eg6-12.cpp
#include <iostream>
using namespace std;

class String {
private:
    int length;                // 字符串长度
    char *sPtr;                // 存放字符串的指针
    void setString( const char *s2);    // 仅供内部成员函数调用的私有设置函数
    // 重载输入、输出运算符
    friend ostream &operator<<(ostream &os, const String &s);
    friend istream &operator>>(istream &is, String &s);
public:
    String(const char * = "");
    ~String();
    const String &operator = (const String &R);    // 重载赋值运算符 =
    const String &operator += (const String &R);    // 字符串的连接 +=
    bool operator == (const String &R);            // 字符串的相等比较 ==
    bool operator != (const String &R);            // 字符串的不等比较 !=
    bool operator!();                               // 判定字符串是否为空
    bool operator<(const String &R) const;         // 字符串的小于比较 <
    bool operator>(const String &R);               // 字符串的大于比较 >
    bool operator>=(const String &R);              // 字符串的大于等于比较
    char &operator[](int);                          // 字符串的下标运算
};

void String::setString(const char *s2) {
    sPtr = new char[length+1];
    strcpy(sPtr,s2);
}

String::String(const char *s):length(strlen(s)) { setString(s); }
String::~String(){ delete []sPtr; }
const String &String::operator=(const String &R) {
    if(&R != this) {                // 若=左右两边的串不相同，才将=右边的串赋值给左边的串
        delete [] sPtr;
        length = R.length;
        setString(R.sPtr);
    }
    return *this;
}

const String &String::operator+=(const String &R) {
    char *temp = sPtr;
```

```

    length += R.length;
    sPtr = new char[length+1];
    strcpy(sPtr, temp);
    strcat(sPtr, R.sPtr);
    delete []temp;
    return *this;
}

bool String::operator==(const String &R){ return strcmp(sPtr, R.sPtr) == 0; }
bool String::operator!=(const String &R){ return !(*this == R); }
bool String::operator!(){ return length == 0; }
bool String::operator<(const String &R)const{ return strcmp(sPtr, R.sPtr)<0; }
bool String::operator>(const String &R){ return R<*this; }
bool String::operator>=(const String &R){ return !(*this<R); }
char &String::operator[](int subscript){ return sPtr[subscript]; }
ostream &operator<<(ostream &os, const String &s) {
    os<<s.sPtr;
    return os;
}

istream &operator>>(istream &is, String &s) {
    char temp[100];
    is>>temp;
    s = temp;
    return is;
}

int main() {
    String s1("happy"), s2("new year"), s3;
    cout<<"s1 is "<<s1<<"\ns2 is "<<s2<<"\ns3 is "<<s3 // L1
        <<"\n 比较 s2 和 s1: "
        <<"\ns2 ==s1 的结果是 "<<(s2 == s1 ? "true" : "false")
        <<"\ns2 != s1 的结果是 "<<(s2 != s1 ? "true" : "false")
        <<"\ns2 > s1 的结果是 "<<(s2 > s1 ? "true" : "false")
        <<"\ns2 < s1 的结果是 "<<(s2 < s1 ? "true" : "false")
        <<"\ns2 >= s1 的结果是 "<<(s2 >= s1 ? "true" : "false");
    cout<<"\n\n 测试 s3 是否为空: "; // L2
    if(!s3) {
        cout<<"s3 是空串"<<endl; // L3
        cout<<"把 s1 赋给 s3 的结果是: "; // L4
        s3 = s1;
        cout<<"s3 = "<<s3<<"\n"; // L5
    }
    cout<<"s1 += s2 的结果是: s1 = "; // L6
    s1 += s2;
    cout<<s1; // L7
    cout<<"\ns1 += to you 的结果是: "; // L8
    s1 += " to you";
    cout<<"s1 = "<<s1<<endl; // L9
    s1[0] = 'H';
    s1[6] = 'N';
    s1[10] = 'Y';
    cout<<"s1 = "<<s1<<"\n"; // L10
    return 0;
}

```

```
}
```

本程序的运行结果如下：

```
s1 is happy // L1 语句输出下面连续的 9 行
s2 is new year
s3 is
比较 s2 和 s1:
s2 == s1 的结果是 false
s2 != s1 的结果是 true
s2 > s1 的结果是 false
s2 < s1 的结果是 true
s2 >= s1 的结果是 false
测试 s3 是否为空: s3 是空串 // L2 和 L3 语句输出
把 s1 赋给 s3 的结果是: s3 = happy // L4 和 L5 语句输出
s1 += s2 的结果是: s1 = happy new year // L6 和 L7 语句输出
s1 += to you 的结果是: s1 = happy new year to you // L8 和 L9 语句输出
s1 = Happy New Year to you // L10 语句输出
```

请结合主函数 `main()` 中注释的语句编号, 以及运行结果中的注释逐一理解每次输出结果, 并借此理解运算符重载的方法和用途。

2. 编程实例二

【例 6-13】 改写 5.6 节的课程结构类, 为 `comFinal`、`Account`、`Chemistry` 类重载输出运算符函数 `operator<<()`, 使程序能够直接利用 `cout` 输出各类的对象。

1) 重载 `comFinal` 类的输出运算符函数 `operator<<()`

启动 VC 2022, 打开目录 `C:\course` 中的 `com_main.sln` 工程文件; 打开 `comFinal.h` 头文件, 并在 `comFinal` 类的声明中添加输出运算符函数 `operator<<()` 的重载声明, 如下所示:

```
// comFinal.h
class comFinal{
    friend ostream &operator<<(ostream &os, comFinal &s);
    ..... // 其余代码不作任何修改
}
```

打开 `comFinal.cpp` 源文件, 并在其中添加输出运算符函数 `operator<<()` 的程序代码。

```
// comFinal.cpp
.....
ostream& operator<<(ostream &out, comFinal &o) {
    cout<<"姓名\t"<<"汉语\t"<<"数学\t"<<"英语\t"<<"总分\t"<<"平均分"<<endl;
    out<<o.name<<"\t"<<o.chinese<<"\t"<<o.math<<"\t"<<o.english<<"\t"
        <<o.getTotal()<<"\t"<<o.getAverage()<<endl<<endl;
    return out;
}
.....
```

2) 重载类 `Account` 的输出运算符函数 `operator<<()`

在 `Account` 头文件的类中添加如下函数声明:

```
// Account.h
```

```
class Account {
    friend ostream &operator<<(ostream &os, Account &s);
    .....
}
.....
```

// 其余代码不作任何修改

添加在 Account.cpp 中的实现代码如下:

```
// Account.cpp
ostream& operator<<(ostream &out, Account &o) {
    cout<<"姓名\t"<<"汉语\t"<<"数学\t"<<"英语\t"
        <<"会计学\t"<<"经济学\t"<<"总分\t"<<"平均分"<<endl;
    out<<o.name<<"\t"<<o.chinese<<"\t"<<o.math<<"\t"<<o.english<<"\t"
        <<o.account<<"\t"<<o.econ<<"\t"<<o.getTotal()+o.account+o.econ
        <<"\t"<<(o.getTotal()+o.account+o.econ)/5<<endl<<endl;
    return out;
}
.....
```

3) 重载类 Chemistry 的输出运算符函数 operator<<()

在类 Chemistry 的头文件和源代码中分别添加如下代码:

```
// Chemistry.h
class Chemistry {
    friend ostream &operator<<(ostream &os, Chemistry &s);
    .....
}
.....
```

// 其余代码不作任何修改

添加在 Chemistry.cpp 文件中的函数代码如下:

```
// Chemistry.cpp
ostream& operator<<(ostream &out, Chemistry &o) {
    cout<<"姓名\t"<<"汉语\t"<<"数学\t"<<"英语\t"<<"化学\t"
        <<"化学分析\t"<<"总分\t"<<"平均分"<<endl;
    out<<o.name<<"\t"<<o.chinese<<"\t"<<o.math<<"\t"
        <<o.english<<"\t"<<o.chemistry<<"\t"<<o.analy<<"\t\t"
        <<o.getTotal() + o.chemistry + o.analy<<"\t"
        <<(o.getTotal() + o.chemistry + o.analy)/5<<endl<<endl;
    return out;
}
.....
```

4) 验证重载结果

改写应用程序的主函数 main(), 如下所示:

```
// com_main.cpp
#include "Chemistry.h"
#include "Account.h"
#include <iostream.h>

void main() {
    comFinal com("刘科学", 78, 76, 89);
```

```

Account a1("张三星", 98, 78, 97, 67, 87);
Chemistry c1("光红顺", 89, 76, 34, 56, 78);
cout<<com; // L1
cout<<"-----"<<endl;
cout<<a1; // L2
cout<<"-----"<<endl;
cout<<c1; // L3
}

```

语句 L1、L2、L3 分别调用了 comFinal、Account 和 Chemistry 的输出运算符函数。程序运行结果如下所示。

姓名	汉语	数学	英语	总分	平均分		
刘科学	76	89	78	243	81		

姓名	汉语	数学	英语	会计学	经济学	总分	平均分
张三星	78	97	98	67	87	427	85

姓名	汉语	数学	英语	化学	化学分析	总分	平均分
光红顺	76	34	89	56	78	333	66

习 题 6

- 6.1 C++为什么要允许运算符重载？
- 6.2 在程序中进行运算符重载时要注意哪些限制条件？
- 6.3 将运算符作为类的友元重载和类成员函数重载有什么区别？
- 6.4 类的类型转换方法有哪些？
- 6.5 读程序，写出程序的运行结果。

(1)

```

#include <iostream>
using namespace std;

class ABC {
    int a, b, c;
public:
    ABC(int x, int y, int z):a(x), b(y), c(z){ }
    friend ostream &operator<<(ostream &out, ABC& f);
};

ostream &operator<<(ostream &out, ABC& f) {
    out<<"a = "<<f.a<<endl<<"b = "<<f.b<<endl<<"c = "<<f.c<<endl;
    return out;
}

void main() {
    ABC obj(10,20,30);
    cout<<obj;
}

```

(2)

```

#include <iostream>
#include <string.h>
using namespace std;

class X {
private:
    char *s;
public:
    X(const char *b) {
        s = new char[sizeof(b)+1];
        strcpy(s, b);
    }
    ~X() { delete []s; }
    void display(){ cout<<"s = "<<s<<endl; }
};

void main() {
    X x1("ok");
    X x2(x1);
    X x3 = x1;
    x2.display();
    x3.display();
}

```

写出本程序的结果，并指出本程序存在的错误。

(3)

```

#include <iostream>
using namespace std;

class Number {
    int n;
public:
    Number(int x):n(x) { }
    Number& operator++() { ++n; return *this; }
    Number& operator++(int) { n++; return *this; }
    friend Number &operator--(Number &o);
    friend Number &operator--(Number o, int);
    void display() { cout<<"This Number is: "<<n<<endl; }
};

Number &operator--(Number &o) { --o.n; return o; }
Number &operator--(Number o, int) { o.n--; return o; }

void main() {
    Number N1(10);
    +++++N1;
    N1.display();
    N1++;
    N1.display();
    --N1;
    N1.display();
    N1-----;
    N1.display();
}

```

}

重载运算符--时，前置自减运算采用了引用参数，后置自减运算采用的是普通参数，注意它们对程序结果的影响。

(4)

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

class Student {
private:
    char *name;
    int age;
    double money;
public:
    Student(const char *n = "NoKnow", int Age = 17, double Mey = 1000.998):age(Age), money(Mey) {
        name = new char[sizeof(n) + 1];
        strcpy(name, n);
    }
    operator char*() { return name; }
    operator int() { return age; }
    operator double() { return money; }
};

void main() {
    Student s1("阿瓦尔古丽", 19, 280000.998);
    char *Name = s1;
    int Age = s1;
    double Money = s1;
    cout<<Name<<"\t"<<Age<<"\t"<<Money<<endl;
    Student s2("武昌鱼");
    Name = s2;
    Age = s2;
    Money = s2;
    cout<<Name<<"\t"<<Age<<"\t"<<Money<<endl;
}
```

6.6 设计一个计数器类 Calculator，只有一个用于计数的数据成员 count。该计数器的有效计数范围是 0~65535，实现计数器的前自增、后自增、前自减、后自减、两个计数器相加减等运算。

6.7 建立一个二维坐标系的类 TwoCoord，用 x、y 表示坐标值，实现两坐标点的加、减运算，计算两坐标点间的距离：

(1) 重载输入、输出运算符，使之能够直接输入、输出坐标点的坐标值。

(2) 重载仿函数（即函数调用运算符），具有一个 TwoCoord 参数，通过仿函数将 TwoCoord 的 x、y 坐标与参数对象的 x、y 分别相加，输出计算后的 x、y 坐标值，并返回坐标值相加后的二维坐标对象。

第 7 章

模板和 STL

模板 (Template) 是 C++ 实现代码重用机制的重要工具，是泛型技术 (与数据类型无关的通用程序设计技术) 的基础。模板是概念级的通用程序设计方法，把算法设计从具体数据类型中分离出来，能够设计出独立于具体数据类型的通用模板程序，包括函数模板和类模板两种类型。在计算机编程领域中被广泛应用的 ANSI C++ STL (Standard Template Library, 标准模板库) 就是用模板技术实现的。

本章主要介绍如下内容：函数模板、类模板及其实例化，模板的类型与非类型参数，STL 常见模板类、算法、元组、迭代器，以及模板设计和元编程等的基本概念。



7.1 模板的概念

某些程序除了所处理的数据类型，程序代码和功能完全相同，但为了实现它们，却不得不编写多个与具体数据类型紧密结合的程序。例如，为了求两个 `int`、`float`、`double`、`char` 类型数值中的最小数，需要编写下列函数：

```
int min(int a, int b) { return (a<b) ? a : b; }
float min(float a, float b) { return (a<b) ? a : b; }
double min(double a, double b) { return (a<b) ? a : b; }
char min(char a, char b) { return (a<b) ? a : b; }
```

有没有办法能够简化这个编程过程，只写一段相同的代码，却能计算出不同类型数据的最小值呢？在 C 语言中，可以通过下面的宏来实现这个想法：

```
#define min(x, y) ((x)<(y) ? (x) : (y);)
```

在 C++ 中，虽然也可以利用宏来进行类似的程序设计，但宏避开了 C++ 的类型检查机制，在某些情况下可能引发错误，是不安全的。更好的方法是用模板来实现这样的程序。

模板相当于某些工艺制造中的模具、母板，通过它们能够制作出形状和功能相同的产品来。只不过 C++ 中的模板所“生产”的产品是函数或类。例如，只需要编写如下函数模板就能够生成上面所有的 `min()` 函数。

```
template <typename T>
T min(T a, T b) {
    return (a<b) ? a : b;
}
```

其中，关键字 `template` 表示定义模板，`min` 模板没有涉及任何具体的数据类型，“<>”中的 `typename` 表示 `T` 可以是任何数据类型，称为类型参数，也可以使用 `class` 替换它，下面的定义是完全等价的。

```
template <class T>
T min(T a, T b) { return (a<b) ? a : b; }
```

但是，建议在 `template` 中用 `typename` 而不是 `class` 来定义类型参数。其一，`typename` 更能体现模板的泛型编程含义；其二，避免模板参数中的 `class` 与类定义中的 `class` 含义混淆。

模板 `min` 代表了求两数最小值的通用算法，与具体数据类型无关，但能够生成计算各种具体类型数据的最小值函数。编译器的做法是用具体数据类型替换模板中的 `T`，生成某实际类型的函数 `min()`。例如，用 `int` 替换模板中所有 `T`，就能生成求两个 `int` 类型数据最小值的函数 `min()`。

模板是一种忽略具体数据类型，只考虑程序操作逻辑的通用程序设计方法，它操作的是参数化的数据类型（类型参数，也称类属参数）而非实际数据类型。

在调用模板时，必须为它的类型参数提供实际数据类型，编译器将利用该数据类型替换模板中的全部类型参数，自动生成与具体数据类型相关的可以运行的程序代码，这个过程称为模板的实例化。由函数模板实例化生成的函数称为模板函数，由类模板实例化生成的类称为模板类。

图 7-1 是模板、模板函数、模板类和对象的关系简图。

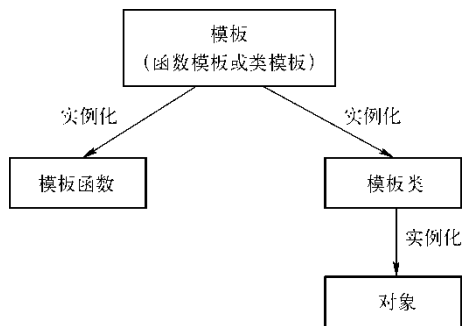


图 7-1 模板、模板函数、模板类和对象之间的关系

7.2 函数模板和模板函数

在计算机中，许多算法操作的数据类型虽然不同，但在程序逻辑上是相同的。例如，对 `int`、`float`、`char`、`double`、`short` 等不同数据类型进行排序、查找、求最大值、最小值等，完成这些任务的同类函数除了数据类型不同，函数的程序代码完全相同。在 C++ 中，函数模板是设计这类函数的高效工具。

函数模板提供了一种通用的函数行为，该函数行为可以用不同的数据类型进行调用，编译器会根据调用的数据类型，自动将它实例化为对应数据类型的函数代码。也就是说，函数模板代表了一个函数家族。与普通函数相比，函数模板中某些函数元素的数据类型是未确定的，这些元素的类型将在使用时根据调用函数的实参类型才能确定；与重载函数相比，函数模板不需要程序员重复编写函数代码，可以由模板自动生成许多功能相同但参数和返回值类型不同的函数。

7.2.1 函数模板的定义

函数模板的定义形式如下：

```
template <typename T1, typename T2, ...>
    返回类型 函数名(参数表) {
    .....
}
```

// 函数模板定义体

其中，`template` 是定义模板的关键字，写在“<>”中的 `T1`、`T2`、... 是模板参数，表示类或函数在定义时要用到的类型或值，如用来定义函数的形参、函数返回值，或定义函数的局部变量。其中的 `typename`（可用 `class` 替换）表示其后的参数可以是任意类型。

模板参数的作用域局限于函数模板范围内，并且要求每个模板参数要在函数的形参列表中至少出现一次。

【例 7-1】 求两数最小值的函数模板。

```
// Eg7-1.cpp
#include <iostream>
using namespace std;

template < typename T>
T Min(T a, T b) {
```

```

    return (a < b) ? a : b;
}

void main() {
    double a = 2, b = 3.4;
    float c = 2.3, d = 3.2;
    cout<<"2、3    的最小值是: "<<Min(2, 3)<<endl;
    cout<<"2、3.4  的最小值是: "<<Min(a, b)<<endl;
    cout<<"'a'、'b' 的最小值是: "<<Min('a', 'b')<<endl;
    cout<<"2.3、3.2 的最小值是: "<<Min(c, d)<<endl;
}

```

程序运行结果如下：

```

2、3    的最小值是: 2
2、3.4  的最小值是: 2
'a'、'b' 的最小值是: a
2.3、3.2 的最小值是: 2.3

```

说明：

① 在定义模板时，不允许在 `template` 语句与函数模板定义之间有任何其他语句。下面的模板定义是错误的：

```

template <typename T>
int x;                                     // 错误，不允许在此位置有任何语句
T Min(T a, T b) {...}

```

② 函数模板可以有多个类型参数，每个参数必须用关键字 `class` 或 `typename` 限定。例如：

```

template <typename T1, typename T2, typename T3>
T1 fx(T1 a, T2 b, T3 c) {...}

```

在这种情况下，函数模板的返回类型通常是 `T1`、`T2`、`...` 中的一种，但每个类型参数（`T1`、`T2`、`...`）必须出现在函数的形参表中。

③ 普通函数或类通常把函数或类的声明放在 `.h` 头文件中，把实现代码放在另一个同名的 `.cpp` 源文件中，以达到接口与实现分离的目的。模板（包括函数模板和类模板）则不一样，由于在用模板创建（实例化）模板函数或模板类时，编译器必须掌握函数模板或类成员函数模板的确切定义，因此必须把模板的声明和定义保存在同一文件中，通常保存在同一头文件中。

7.2.2 函数模板的实例化

当编译器遇到普通函数时会读取函数定义、检查错误并生成可执行代码，但当遇到关键字 `template` 和跟随其后的函数定义时，它只是简单地知道：这个函数模板在后面的程序代码中可能用到。在这个阶段，编译器只会检测和模板参数无关的语法错误（如变量名是否符合规定，是否缺少“}”或“;”等），并不会根据函数模板生成任何代码，因为此时它并不知道函数模板要处理的具体数据类型，无法生成任何函数代码。

也就是说，对函数模板进行编译时并不会生成可执行的二进制命令代码，只有当编译器遇到程序中对函数模板的调用时，才会根据调用语句中实参的具体类型，推断并确定模板参数的实际数据类型，并用此类型替换函数模板中的模板参数，生成能够处理该数据类型的函

数代码，这个过程称为**模板的实例化**，生成的函数称为**模板的实例**，也称为模板函数。

例如，在例 7-1 中，当编译器遇到 `template <typename T> T min(T a, T b) {...}` 模板定义时，并不会生成任何函数代码，但当它遇到函数调用 `min(2, 3)` 时，会推断出调用参数 2、3 的类型都是 `int`，就会用 `int` 替换函数模板中的所有类型参数 `T`，生成求两个整数最小值的模板函数，如下所示：

```
int Min(int a, int b) {  
    return (a<b) ? a : b;  
}
```

当编译器遇到 `Min('a', 'b')` 调用时，就会根据参数 'a'、'b' 的类型 `char`，将模板中的 `T` 全部替换成 `char` 类型，生成 `char` 类型的模板函数 `Min(char, char)`，再调用此模板函数计算对应参数的最小值。图 7-2 是例程 7-1 中函数模板 `Min` 实例化为不同模板函数的示意。

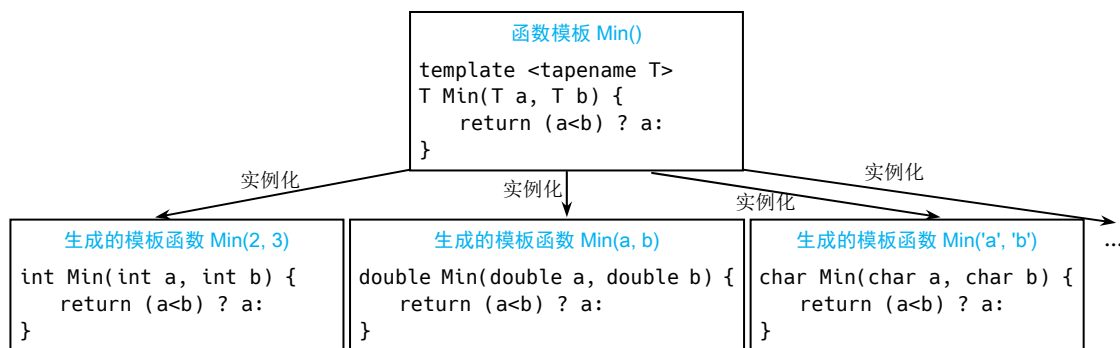


图 7-2 函数模板 `Min()` 实例化的模板函数

当多次使用具有相同类型的参数调用模板时，编译器只在第一次调用时生成模板函数，此后再遇到相同类型的参数调用时，它将调用第一次实例化生成的模板函数。例如，若在例 7-1 中有如下函数调用：

```
int x = Min(2, 3);  
int y = Min(3, 9);
```

编译器只在计算 `x` 时生成模板函数 `int Min(int, int)`，并用此函数计算出 `y` 值。

7.2.3 模板参数

1. 模板参数的匹配问题

C++ 在实例化函数模板的过程中，只是简单地将模板参数替换成调用实参的类型，并以此生成模板函数，不会进行参数类型的任何转换。这种方式与普通函数的参数处理有着极大的区别。在普通函数的调用过程中，C++ 会对类型不匹配的参数进行隐式类型转换。

例如，对于例 7-1 中的函数模板 `Min()`，在函数 `main()` 中进行不同的实例化：

```
void main() {  
    double a = 2, b = 3.4;  
    float c = 2.3, d = 3.2;  
    cout<<"2, 3.2    的最小值是: "<<Min(2, 3.2)<<endl;  
    cout<<"a, c      的最小值是: "<<Min(a, c)<<endl;  
    cout<<"'a', 3     的最小值是: "<<Min('a', 3)<<endl;  
}
```


编译该程序，会产生类似如下形式的多个编译错误：

```
C2782, "T min(T,T) " : 模板参数“T”不明确
.....
```

错误来源于函数 `main()` 中的 3 次函数调用 `Min()`，是同一类型的错误，即模板参数不匹配。例如，在遇到函数调用 `Min(2, 3.2)` 时，编译器将先用调用实参的类型实例化函数模板，生成模板函数。由于 `Min(2, 3.2)` 的调用参数类型分别为 `int` 和 `double` 类型，而函数模板 `Min()` 中只有一个类型参数 `T`，不能让 `T` 同时取 `int` 和 `double` 两种类型。在模板实例化的过程中，C++ 不会进行任何形式的隐式类型转换，于是产生了上述编译错误信息。解决的办法有以下几种。

1) 在模板调用时进行参数类型的强制转换

在模板函数调用过程中，强制转换调用实参的类型，使其类型与模板参数的要求相符，这样可以避免模板实例化过程中的类型匹配问题。例如，将上面的模板调用改写成如下形式的语句，程序就能够编译运行，并得到正确的运行结果。

```
cout<<"2、3.2    的最小值是："<<Min(double(2), 3.2)<<endl;
cout<<"a、c      的最小值是："<<Min(a, double(c))<<endl;
cout<<"'a'、3    的最小值是："<<Min(int('a'), 3)<<endl;
```

2) 显式指定函数模板实例化的类型参数

前面所有实例化都是当遇到 `Min` 模板的调用时，C++ 编译器才根据调用参数自动推断出模板的参数类型，从而生成模板函数，称为隐式实例化。此外，可由用户显式指定模板参数的类型：在调用函数模板时，将参数的实际类型写在调用函数名后面的“<>”中，编译器将以“<>”中指定的类型实例化函数模板，生成模板函数。

```
cout<<"2、3.2    的最小值是："<<Min<double>(2, 3.2)<<endl;
cout<<"a、c      的最小值是："<<Min<double>(a, c)<<endl;
cout<<"'a'、3    的最小值是："<<Min<int>('a', 3)<<endl;
```

当有多个模板参数时，就要在“<>”中以“,”为分隔符，分别指定各模板参数的类型。

3) 指定多个模板参数

为了避免一个模板参数与多个不同调用实参的类型冲突问题，可以为函数模板指定多个不同的类型参数。

【例 7-2】 用两个模板参数实现求最大值的函数模板。

```
// Eg7-2.cpp
#include <iostream>
using namespace std;

template <typename T1, typename T2>
T1 Max(T1 a, T2 b) {
    return (a>b) ? a : b;
}

void main(){
    double a = 2, b = 3.4;
    float c = 5.1, d = 3.2;
    cout<<"2、3.2    的最大值是："<<Max(2, 3.2)<<endl;
    cout<<"a、c      的最大值是："<<Max(a, c)<<endl;
    cout<<"'a'、3    的最大值是："<<Max('a', 3)<<endl;
}
```

函数的运行结果如下：

```
2、3.2    的最大值是：3
a、c      的最大值是：5.1
'a'、3    的最大值是：a
```

这个运行结果并不精确，甚至存在较大的误差，但不代表程序有什么错误。如果注意到第一个调用参数 **T1** 的类型就不会有问题，因为函数模板 **max()** 的返回值依赖于模板参数 **T1**，如果在调用时将精度较高的数据类型作为第一参数，即让 **T1** 取它的类型，结果将是准确的。

例如，将例 7-2 中的函数调用依次改为 **Max(3.2, 2)**、**Max(c, a)**、**Max(3, 'a')**，将得到正确的程序运行结果。但是，这毕竟不方便，更好的解决方法是用 **auto** 推断函数返回类型，如下：

```
template <typename T1, typename T2>
auto max(T1 a, T2 b) { return (a>b) ? a : b; }           // C++ 14
```

2. 类型与非类型模板参数

在函数模板的“<>”中可以包括两种类型的模板参数：类型模板参数和非类型模板参数。类型模板参数是指用 **typename**（或 **class**）限定的参数，可以看成类型说明符，如同 **int**、**double**、**char** 等系统内置数据类型一样使用。例如，可以用来指定函数模板的返回类型，定义模板形式参数或模板内的局部变量等。非类型模板参数采用明确的数据类型指定的参数，定义方法与普通函数的形参定义差不多，如同样可以为它指定默认值等。但是，并非任何数据类型都可以作为模板的非类型参数，只有整型（包括 **int**、**bool** 和指针类型）才能够作为非类型参数。

【例 7-3】 用函数模板实现数组的选择法排序，数组可以是任意类型，用类型参数设定；数组大小是整数类型，用非类型参数指定。

```
// Eg7-3.cpp
#include <iostream>
using namespace std;

template <typename T, int n = 6>                // a 为类型参数，n 为非类型参数，默认值 6
void sort(T a[n]) {
    for(int i = 0; i < n; i++) {
        int p = i;
        for(int j = i; j < n; j++) {
            if(a[p] < a[j])
                p = j;
        }
        T t = a[i];
        a[i] = a[p];
        a[p] = t;
    }
}

template <typename T>
void display(T& a, int n) {
    for(int i = 0; i < n; i++)
        cout<<a[i]<<"\t";
    cout<<endl;
}

void main() {
```



```

int m = 7;
int a[] = {1, 41, 2, 5, 8, 21, 23};
char b[] = {'a', 'x', 'y', 'e', 'q', 'g', 'o', 'u'};
// sort<int, m>(a);           // L1, 错误, 只能向非类型参数传值, 不能传变量 m
sort<int, 7>(a);              // L2, 正确
sort<char, 8>(b);
display(a, m);                // L3, 正确, n 对应模板函数的普通形参
display(b, 8);
}

```

程序运行结果如下:

```

41  23  21  8   5   2   1
y   x   u   q   o   g   e   a

```

本程序定义了两个函数模板：一个是数组排序函数模板 `sort()`，另一个是通用的显示数组内容的函数模板 `display()`。它们能够对任意内置数据类型的数组进行排序和输出，只需把数组的名称和大小传递给它们，函数模板 `sort()` 就能对数组进行从大到小的排序，函数模板 `display()` 就能够输出数组的各元素。如果传递的实参是用户自定义的数据类型，如类、结构、枚举等，只要重载了它们的比较运算符函数 `operator<()` 和输出运算符函数 `operator<<()`，也能够用 `sort()` 和 `display()` 对它们进行排序和输出。

注意：在调用函数模板时，只能向非类型模板参数传递常数，不能传递变量。

语句 L1 向模板函数 `sort()` 的非类型参数传递变量 `m`，因此是错误的；语句 L3 向模板函数 `display()` 也传递了变量 `m`，但没有出错，原因是 `m` 传递的对象是 `display()` 的普通形参 `n`（没有在模板的 `<>` 中指定，就不是模板参数），既不是模板的类型参数，也不是模板的非类型参数，按照普通函数参数传递的方式进行处理。

3. 模板参数的作用域

模板参数遵循普通函数参数的作用域范围规则：模板参数的可用范围在其声明之后，直到模板声明或定义结束之前；同普通局部变量一样，模板参数也会隐藏外层作用域中声明的同名标识符。但与普通函数不同的是，在函数模板中不能重用模板参数名称。

```

struct A{ };
template<typename A, typename B>
void f(A a, B b) {
    A t = a;           // t 是 typename A 指定的类型
    int B;              // 错误, 重用模板参数定义变量名称
}

```

在函数模板 `f()` 内，模板参数 `A` 隐藏了外层作用域定义的结构类型 `A`。

7.3 类模板

7.3.1 类模板的概念

函数模板用于设计程序代码相同但所处理数据类型不同的通用函数。与此相似，类模板用来设计结构和成员函数完全相同但所处理的数据类型不同的通用类。例如，对于堆栈类而

言（见 3.14 节），可能存在整数栈、双精度数栈、字符栈等不同数据类型的栈，每个栈类除了所处理的数据类型不同，类的结构和成员函数完全相同。但是，为了在非模板的类设计中实现这些栈，不得不重复编写各栈类的相同代码，如下所示。

整数栈：

```
class intStack {
private:
    int data[size];           // 用整型数组存放栈数据
    int top;                  // 栈顶指针
    int size = 10;            // 栈的默认大小
public:
    void init() {...}         // 初始化栈
    void Push(int x) {...};    // 入栈操作
    int Pop() {...};           // 出栈操作
};
```

双精度栈（其中省略的内容与 intStack 相同）：

```
class DoubleStack {
private:
    double data[size];        // 用双精度数组存放栈数据
    .....
};
```

字符栈（其中省略的内容与 intStack 相同）：

```
class CharStack {
private:
    char data[size];          // 用字符数组存放栈数据
    .....
};
```

上述各堆栈类，除了所处理的数据类型不同，程序代码都相同。在 C++ 中，用类模板来设计这样的类簇最高效，一个类模板就能够实例化生成所有需要的堆栈类。

类模板也称为类属类，可以接收数据类型作为参数，设计出与具体数据类型无关的通用类。在设计类模板时，可以用与具体数据类型无关的通用类型参数定义其中的某些数据成员、成员函数的参数或返回值。

7.3.2 类模板的定义

类模板与函数模板的定义形式相似，必须以关键字 **template** 开始，后面是用 “<>” 括起来的模板参数，然后是类名。形式如下：

```
template<typename T1, typename T2, ...>
class 类名 {
    .....
}
```

其中，**typename** 表示其后的参数可以是任何类型，同一个类模板中可以定义多个不同的模板参数。**typename** 也可以用 **class** 代替，但它与“类名”前面的 **class** 具有不同的含义：一个用于声明类型参数，一个用于声明类。

类模板的模板参数也有类型参数和非类型参数两种。在类模板的“<>”中，用 `typename` 或 `class` 指定的就是类型参数，用某种实际数据类型定义参数就是非类型参数。与函数模板一样，非类型参数也是受限制的，只能是整型（包括 `int`、`bool`、指针类型等），在调用类模板时只能为非类型参数提供常数值。在如下模板参数表中，`T1`、`T2` 是类型参数，`T3` 是非类型参数：

```
template<typename T1, typename T2, int T3>
```

在实例化时，必须为 `T1`、`T2` 提供一种数据类型，为 `T3` 指定一个整常数（如 10），该模板才能被正确地实例化。

现在以类模板 `Stack` 的设计为例，说明类模板的定义方法、类型参数和非类型参数的用法。

【例 7-4】 设计一个堆栈的类模板 `Stack`，在模板中用类型参数 `T` 表示栈中存放的数据，用非类型参数 `MAXSIZE` 代表栈的大小。栈模板头文件 `Stack.h` 的代码清单如下：

```
// Eg7-4a.cpp / Stack.h
template<typename T, int MAXSIZE>                // MAXSIZE 是非类型参数，代表栈的容量大小
class Stack {
private:
    T elems[MAXSIZE];                          // elems 数组用于存储栈的数据元素
    int top;                                    // 栈顶指针
public:
    Stack() { top = 0; };
    void push(T e);                             // 入栈操作
    T pop();                                     // 出栈操作
    bool empty(){ return top == 0; }            // 判断栈是否为空
    bool full(){ return top == MAXSIZE; }       // 判断栈是否满
};
template<typename T, int MAXSIZE>                // push 成员函数的类外定义
void Stack< T, MAXSIZE>::push(T e) {
    if(top == MAXSIZE) {
        cout<<"栈已满，不能再加入元素了！";
        return;
    }
    elems[top++] = e;
}
template<typename T, int MAXSIZE>                // pop 成员函数的类外定义，指定为内联函数
inline T Stack<T, MAXSIZE>::pop() {
    if(top <= 0) {
        cout<<"栈已空，不能再弹出元素了！"<<endl;
        return 0;
    }
    top--;
    return elems[top];
}
```

说明：

① 在 `Stack` 的模板参数中，`T` 是类型参数，`MAXSIZE` 是非类型参数。在实例化 `Stack` 栈时，必须为 `MAXSIZE` 提供一个 `int` 类型的常量值，不能用数据类型作为调用参数。

② 类模板的数据成员、成员函数的参数可以是类型参数，也可以是普通数据类型。如成员函数 `push()` 的参数、数据成员 `elems` 的类型是类型参数 `T`，而 `top` 是 `int` 类型。

③ 在类模板内部定义成员函数时，其定义方法与普通类成员函数的定义方法相同，如 `Stack` 的构造函数和栈空判定函数 `empty()` 的定义。

④ 在类模板外部定义成员函数时，必须将模板声明加在成员函数的名称前面，而且需要用类模板的完整类型限定符“`类名<模板参数名>::`”进行限定。例如，`Stack` 的 `push()` 成员函数的定义如下：

```
template<typename T, int MAXSIZE>                // 类模板声明
void Stack<T, MAXSIZE>::push(T e) {
    .....
}
```

函数名 `push` 前面的“`Stack<T, MAXSIZE>::`”就是 `Stack` “类模板的完整类型限定符”，在类外定义的所有 `Stack` 成员函数都必须将此限定符加在函数名称前面。

⑤ 如果要在类模板外将成员函数定义为 `inline` 函数，就应该将 `inline` 关键字加在类模板的声明后。如 `Stack` 的成员函数 `pop()` 就被定义成了 `inline` 函数，其定义如下：

```
template<typename T, int MAXSIZE>                // pop 成员函数的类外定义
inline T Stack<T, MAXSIZE>::pop() { ... }         // 指定为内联函数
```

⑥ 不能像普通类那样简单地把类的声明放在头文件（如“`类名.h`”）中，而把类的实现放在源文件（如“`类名.cpp`”）中，这样做可能为类模板的实例化带来问题。因此，常将类模板的声明和定义都放在同一个文件中，就像将 `Stack` 模板的声明和定义都放在 `Stack.h` 中一样。

7.3.3 类模板实例化

类模板实例化包括模板实例化和成员函数实例化。当用类模板定义对象时，将引起类模板的实例化。在实例化类模板时，如果模板参数是类型参数，就必须为它指定具体的数据类型；如果模板参数是非类型参数，就必须为它指定一个常量值。如对前面的 `Stack` 类模板而言，下面是它的一条实例化语句：

```
Stack<int, 10> iStack;
```

其中，`<int, 10>` 就是为实例化 `Stack` 指定的模板实参，`iStack` 是用类模板 `Stack` 定义的一个对象。由于 `Stack` 的第一个模板参数是类型参数，因此必须为它指定一种具体的数据类型（这里是 `int`）；第二个模板参数是非类型参数，必须为它指定一个常量值（这里是 `10`）。这条语句将用类模板 `Stack` 实例化生成容量为 `10` 的整数栈类。

编译器实例化 `Stack` 的方法是：将 `Stack` 模板声明中的所有类型参数 `T` 替换成 `int`，将所有非类型参数 `MAXSIZE` 替换成 `10`，这样就用类模板 `Stack` 生成了一个 `int` 类型的模板类。为了区别于普通类，暂且将该模板类记为 `Stack<int, 10>`，即在类模板名后的“`<>`”中写上模板实参表。该类的代码如下：

```
class Stack {
private:
    int elems[10];                // elems 数组用于存取栈的数据元素
    int top;                      // 栈顶指针
public:
    Stack() { top = 0; };
    void push(int e);             // 入栈操作
```

```

int pop(); // 出栈操作
bool empty() { return top == 0; } // 判断栈是否为空
bool full(); // 判断栈是否满
};

```

最后，C++将用这个模板类定义一个对象 iStack。当用语句 “Stack<char, 20> cStack;” 实例化 Stack 类时，C++将用 char 替换 Stack 模板中所有的 T，用 20 替换模板中的 MAXSIZE，生成能够处理字符类型的模板类 Stack<char, 20>，再用生成的模板类定义对象 cStack。图 7-3 是类模板、模板类及模板对象之间的关系。

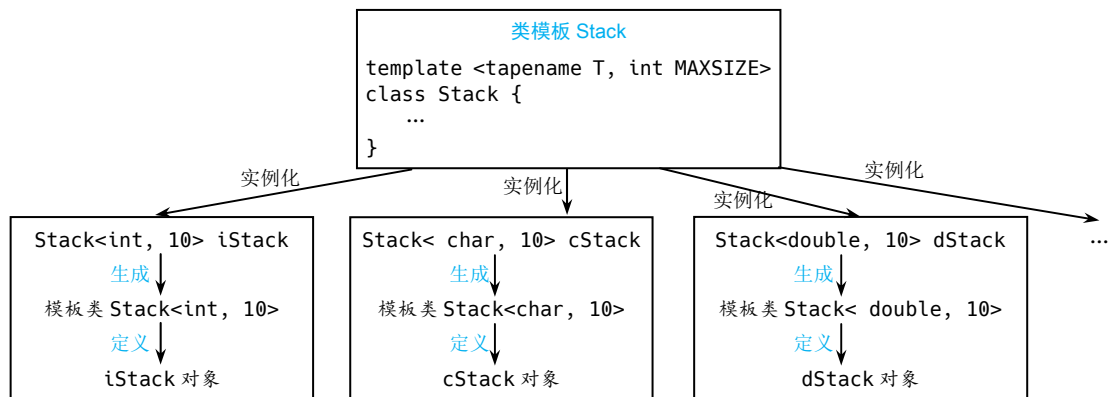


图 7-3 类模板、模板类及模板对象之间的关系

由图 7-3 可知，类模板、模板类及模板对象之间的关系为：由类模板实例化生成针对具体数据类型的模板类，再由模板类定义模板对象。

注意：在上面的实例化过程中，只有被调用的构造函数才会被实例化，并不会实例化类模板的其他成员函数。也就是说，在用类模板定义对象时并不会生成构造函数之外的类成员函数的代码。

类模板成员函数的实例化发生在该成员函数被调用时，这意味着只有那些被调用的成员函数才会被实例化。或者说，只有当成员函数被调用了，编译器才会为它生成真正的函数代码。例如，对于例 7-4 的类模板 Stack，假设有如下函数：

```

void main() {
    Stack<int, 10> iStack;
    iStack.push(i);
}

```

在函数 main() 中并没有调用 Stack 的成员函数 empty()、full() 和 pop()，所以 C++ 在 Stack<int, 10> iStack 的实例化过程中不会生成函数 empty()、full() 和 pop() 的代码。作为验证，可以删掉 Stack.h 中成员函数 pop() 的类外定义，同时将函数 empty() 和 full() 在 Stack 中的定义改为如下声明，再编译运行该程序，可以发现程序同样能够正确地执行。

```

class Stack {
public:
    ....
    bool empty(); // 判断栈是否为空
    bool full(); // 判断栈是否满
};

```

7.3.4 类模板的应用

为了使用类模板对象，必须显式地指定模板实参。

下面的例子展示了如何应用类模板 `Stack<>`。

```
// Eg7-4b.cpp
#include"stack.h" // 该头文件的内容见例 7-4 所示的程序清单
#include<iostream>
using std::cout; // 只使用 std 域名空间中的 cout
using std::endl; // 只使用 std 域名空间中的 endl

void main() {
    Stack<int, 10> iStack; // 实例化 Stack 为 int 类型的栈类，栈容量为 10
    Stack<char, 10> cStack; // 实例化 Stack 为 char 类型的栈类，栈容量为 10
    cout<<"-----intStack----\n";
    int i;
    for(i = 1; i < 10; i++)
        iStack.push(i);
    for(i = 1; i < 10; i++)
        cout<<iStack.pop()<<"\t";
    cout<<"\n\n-----charStack----\n";
    cStack.push('A');
    cStack.push('B');
    cStack.push('C');
    cStack.push('D');
    cStack.push('E');
    for(i = 1; i < 6; i++)
        cout<<cStack.pop()<<"\t";
    cout<<endl;
}
```

本程序运行结果如下：

```
-----intStack----
 9   8   7   6   5   4   3   2   1
-----charStack----
 E   D   C   B   A
```

与普通类的对象一样，类模板的对象或引用也可以作为函数的参数，只不过这类函数通常是模板函数，且其调用实参常常是模板类对象。

【例 7-5】 为例 7-4 建立的类模板 `Stack` 编写一个函数 `display()`，能够读取并显示用模板类 `Stack` 建立的栈中的所有元素。

```
// Eg7-5.cpp
#include"stack.h"
#include<iostream>
using namespace std;

template<class T>
void display(Stack<T, 10> s) {
    while(!s.empty())
        cout<<s.pop()<<"\t";
}
```

```

        cout<<endl;
    }

    void main() {
        Stack<int, 10> iStack;
        cout<<"-----intStack----\n";
        for(int i = 1; i < 10; i++)
            iStack.push(i);
        display(iStack);
    }

```

运行该程序，将输出如下结果：

```

-----intStack----
9 8 7 6 5 4 3 2 1

```

本程序中的类模板及函数模板的实例化过程如下。

<1> 在本例中，通过#include "stack.h"直接引用了类模板 Stack，并定义了一个函数模板 display()来显示 Stack 的模板类对象。

```

template<typename T>
void display(Stack<T, 10> s) {
    .....
}

```

该函数模板的形参是类模板 Stack 的对象 s。

<2> 在函数 main()中，语句“Stack<int, 10> iStack;”将引起类模板 Stack 的实例化，生成了一个 int 类型的模板类 Stack<int, 10>，并建立了该模板类的一个对象 iStack。

<3> 调用语句“display(iStack);”，以模板类 Stack<int, 10>的对象 iStack 为实参，编译系统将利用函数模板 display()实例化生成如下模板函数。

```
void display(Stack<int, 10> s){...}
```

<4> 调用模板函数 display()，输出栈对象中的内容。

7.4 模板设计中的独特问题

7.4.1 模板参数类型推导

前面的模板参数只涉及值类型，实际编程中还会涉及左值引用、右值引用等，简述如下。

1. 传值模板参数

值类型模板参数的形式如下：

```

template<typename param>
r_type function(param p, ...) {...}

```

当调用函数模板 function 时，编译器将根据传递给形参 p 的实参类型推导出模板参数 param 的实际类型，并以此实例化生成模板函数。为了体现函数的值形参语义，编译器在推导模板参数 param 类型的过程中，会去除所有加在实参上的类型限定符（包括 const、&和&&），因为这样才能够保证模板函数值形参可复制的语义（函数调用时将实参的值复制到值形参变

量中)。

【例 7-6】 模板传值形参的类型检测。

```
// Eg7-6.cpp
#include<iostream>
#include<memory>
#include<typeinfo>
using namespace std;

template<typename T>
void f(T p) {
    cout<<typeid(p).name()<<endl;
}

int main() {
    int i1 = 0;
    int& i2 = i1;
    const int& i3 = i1+8;
    int&& i4 = 5 + 7;
    int* i5 = new int(1);
    unique_ptr<int> i6 {new int};
    f(2);
    f(i1);
    f(i2);
    f(i3);
    f(i4);
    f(i5);
    // f(i6); // L1, unique_ptr 指针不能够被复制
    f(move(i6)); // L2
}
```

typeid 是 C++ 的类型运算符函数，可以在程序执行过程中动态识别表达式的类型，它的成员函数 name() 能够获取类型名称。该程序的运行结果如下：

```
int // 从 2 推导的 p 类型
int // 从 i1 推导的 p 类型
int // 从 i2 推导的 p 类型
int // 从 i3 推导的 p 类型
int // 从 i4 推导的 p 类型
int * __ptr64 // 从 i5 推导的 p 类型
class std::unique_ptr<int,struct std::default_delete<int>>
```

由运行结果可知，加在 i2、i3、i4 变量的 const、左值引用和右值引用都去掉了，从它们推导出的模板参数 p 都是 int 值类型，这样才能够保证调用模板函数 f() 时把实参的值复制到形参 p 中。

智能指针 unique_ptr 是不能够被复制的 (C++ 标准禁用了它的复制语义)，因此 L1 语句的调用是错误的，但 L2 通过移动函数将智能指针 i6 的右值传递给模板参数 p 是允许的。

2. 左值引用模板参数

当类型参数为可变引用（左值引用）时，可以向它传递能够推断出地址的实参，如果是

const 类型的实参，将保留它的 const 限定。但是，不能够接受字面常量和临时变量，这是因为引用参数是可以修改实参值的，但将常量和临时变量传给引用参数，就与该语义相违背了。修改上面的例子，将 f() 的类型参数改为引用参数。

【例 7-7】 模板传引用形参的类型检测。

```
// Eg7-7.cpp
#include<iostream>
#include<memory>
using namespace std;

template<typename T>
void f(T& p) {
    cout<<typeid(p).name()<<endl;
}

int main() {
    int i1 = 0;
    int& i2 = i1;
    const int& i3 = i1 + 8;
    int&& i4 = 5 + 7;
    int* i5 = new int(1);
    unique_ptr<int> i6 { new int };
    // f(2); // 错误,
    f(i1); // p 为 int&
    f(i2); // p 为 int &
    f(i3); // p 为 const int&
    f(i4); // p 为 const int&
    f(i5); // p 为 int *&
    f(i6); // p 为 unique_ptr<int>*p
    // f(move(i6)); // unique_ptr 指针不能够被复制
}
```

f(2)向模板 f(T& p)的引用模板参数 p 传递字面常量，f(move(i6))则向 p 传递临时变量，这与 p 是引用，可以通过它修改 p 的语义相违背，因此都是错误的。

事实上，f(i3)、f(i4)、f(i5)、f(i6)都与引用参数的语义有所违背，因为从这些参数中推断出的模板参数 p 要么是 const 引用，要么是指针的引用，都与 p 本身作为引用参数其值可被修改的语义相违背。这里的程序之所以能够编译运行，是因为 f 模板并没有修改 p，如果将 f 改变如下形式，那么 f(i3)~f(i6)的定义都是错误的。

```
template<typename T>
void f( T& p) {
    T t = 0;
    p = t;
    cout<<typeid(p).name()<<endl;
}
```

因此，如果设计模板 f()的本意是不需要通过它修改参数 p，就可以把该参数设置为 const 引用，类似于如下形式，它能够接受任何实参。

```
void f(const T& p) { ... // 模板中不能有修改 p 的语句 }
```

3. 右值引用模板参数

形如 `T&&` 这样的右值引用参数，既可以接受左值引用实参，也可以接受右值引用实参，通常被称为转发引用。

【例 7-8】 模板右值引用参数的类型检查。

```
// Eg7-8.cpp
#include<iostream>
#include<memory>
using namespace std;

template<typename T>
void f(T&& p) {
    cout<<typeid(p).name()<<endl;
}

int main() {
    int i1 = 0;
    int& i2 = i1;
    const int& i3 = i1+8;
    int&& i4 = 5 + 7;
    int* i5 = new int(1);
    unique_ptr<int> i6 { new int };
    f(2);
    f(i1);
    f(i2);
    f(i3);
    f(i4);
    f(i5);
    f(i6);
    f(move(i6));
}
```

程序的运行输出如下：

```
int
int
int
int
int
int * __ptr64
class std::unique_ptr<int, struct std::default_delete<int>>
class std::unique_ptr<int, struct std::default_delete<int>>
```

函数 `main()` 中对于 `f()` 的所有调用都是正确的。其中，`f(2)`、`f(i4)`、`f(move(i6))` 传递给模板参数 `p` 的参数都是右值引用，编译系统会去掉这些右值的限定符后，推导出的类型分别为 `int` 和 `unique_ptr<int>`，模板参数 `p` 就是对应类型的右值引用。`f(i2)`、`f(i3)` 传递给函数模板 `f()` 的是 `int` 的左值引用，编译器将用右值模板参数 `p` 来接收 `int` 左值引用的参数，此时模板参数 `p` 的类型就是 `int &`。为什么会这样呢？原来，当右值不是模板参数时，它是不能够接受左值的，因为没有模板，类型替换就不会参与到左值和右值的相互转换中，没有类型替换发生，左值引用就不会变换为右值引用了。C++ 编译器对待当模板形参和调用实参都是运用的同一种方

表 7-1 引用折叠规则

形 参	实 参	
	&	&&
T&	T&	T&
T&&	T&	T&&

法,称为引用折叠,如表 7-1 所示。由表 7-1 可知,只要形参和实参两者中有一个引用是左值时,就会折叠到左值引用,只有当两者都是右值时,才会折叠到右值(严谨地说,结果是&出现最少的那个)。由此可见,当类型参数为左值引用时,无论实参是左值还是右值,推断出的模板参数都是左值引用。当类型参数是右值引用时,只有当实参也是右值时,推断出的模板参数才是右值引用。

当一个函数模板要调用另一个模板函数时,有时不需要进行引用折叠,即主调函数在向被调函数传参数时,不改变实参数的引用类型。例如,函数模板 f()的形式定义如下:

```
template<typename T> void f(T&& p) {...}
```

函数模板 g()需要调用模板函数 f(),但不希望 f()改变模板参数 p 的引用类型,即:若 p 是左值引用,则传递给 f()的也是左值引用;若 p 是右值引用,则传给 f 的也是右值引用。这时就可以用 forward 使用非限定的类型参数显式实例化 f()的调用参数,并称这种模板参数转发方式为完美转发,也就是调用模板,将模板参数完美转发给被调用函数 f(),其形式定义如下:

```
template<typename T> void g(T && p) {
    f(forward<T>(p));           // 完美转发
    .....
};
```

7.4.2 内联与常量函数模板

无论是函数模板还是类模板,如同普通函数和类成员函数一样,都可以定义为 inline 和 constexpr 函数。inline 和 constexpr 关键字需要放在模板参数列表之后、函数返回类型之前。

```
template <typename T>
inline T Min(T a, T b) { return (a<b) ? a : b; }
template <class T>
constexpr T Min(T a, T b) { return (a<b) ? a : b; } // C++ 11
```

下面的 Min()函数模板声明则是错误的,inline 关键字的位置不对。

```
inline template <typename T>
T Min(T a, T b) { return (a<b) ? a : b; }
```

7.4.3 默认模板实参 C++ 11

与普通函数的参数可以有默认值类似,模板参数也可以有默认值(包括函数模板和类模板),遵守同样的规则:一旦为某个模板参数指定了默认值,则它右边的模板参数都应该有默认值。

【例 7-9】设计比较两个不同类型数字大小的函数模板 compare(),第二个模板参数的类型默认为 double。当第一个参数大于第 2 个参数时,返回 1;小于第二个参数时,返回-1;相等时,返回 0。

```
// Eg7-9.cpp
#include<iostream>
```

```

using namespace std;

template <typename T, typename D = double>           // 默认模板参数
int compare(T t = 0, D u = 0) {
    if(t > u)
        return 1;
    else if(t < u)
        return -1;
    else
        return 0;
}

void main() {
    cout<<compare(10, 'a')<<"\t";                // compare<int, char>(10, 'a')
    cout<<compare<int, char>()<<"\t";              // compare<int, char>(0, 0)
    // 下面两次 compare 调用都使用了默认模板参数 double
    cout<<compare(20)<<"\t";                        // compare<int, double>(20, 0)
    cout<<compare<int>()<<"\t";                    // compare<int, double>(0, 0)
    // compare();                                  // 错误：不能确定模板参数 T 的类型
}

```

程序运行的结果如下：

```
-1      0      1      0
```

为类模板指定默认参数的方法与函数模板相同，下面是为例 7-4 设计的堆栈类指定默认值的例子，其中省略的代码与例 7-4 完全相同。

```

template<class T = int, int MAXSIZE = 10>           // 模板参数默认值
class Stack {
    .....
};
Stack<> iStack;                                     // 默认实例化 iStack 为 int 类型的栈类，栈容量为 10
Stack<char, 20> cStack;                             // 实例化 cStack 为 char 类型的栈类，栈容量为 20

```

7.4.4 仿函数应用

仿函数本质是对类的函数调用运算符函数 `operator()` 的重载。其形式定义如下：

```

class A {
    A operator()(参数表) {...}           // 仿函数
}

```

仿函数，又叫函数对象，可以像调用函数一样使用对象，非常方便。与普通函数最大的区别在于，仿函数可以灵活应用于自身或其他函数。例如：

```

A obj;
obj(...);                                     // 调用仿函数

```

其中的参数表与 `operator()` 的参数表相匹配。

【例 7-10】 设计一个通用求和、积的函数模板 `accumulate()`，调用类模板 `Add` 和 `Mul` 的仿函数计算数据区间的和或积。

```

// Eg7-10.cpp
#include<iostream>

```

```

#include<vector>
using namespace std;

template<typename T>
class Add {
public:
    T operator()(const T& x, const T& y) { return x + y; }
};

template<typename T>
class Mul {
public:
    T operator()(const T& x, const T& y) { return x * y; }
};

template<typename iter,typename T, typename function>
T accumulate(iter begin, iter end, T init, function f) {
    for(iter it = begin; it != end; ++it)
        init = f(init, *it);
    return init;
}

int main() {
    vector v1 = {2.1, 3.1, 4.1, 5.1, 7.1}; // L1, C++ 17 以上标准
    vector<double> v2 = {2.0, 3.0, 4.0, 5.0, 7.0}; // L2, C++ 17 之前标准
    double sum = accumulate(v1.begin(), v1.end(), 0.0, Add<int>{}); // L3
    double mul = accumulate(v2.begin(), v2.end(), 1.0, Mul<double>{}); // L4
    cout<<"sum = "<<sum<<"\tmul = "<<mul<<endl;
}

```

程序运行结果如下：

```
sum = 21    mul = 840
```

注意，语句 L1 和 L2 是有区别的。语句 L1 必须在 C++ 17 及以上的标准中执行，可以根据列表中的内容自动推导向量 v1 的大小和数据类型。但是，在 C++ 17 标准之前，编译器只能根据初始化列表推导出数组元数的个数，无法推导数组的类型，必须像语句 L2 那样显式确定向量 v1 的类型。

7.4.5 成员模板

可以把类（包括普通类和类模板）的某个或某几个成员函数设置为模板，称为**成员模板**。成员模板的定义方法与普通函数模板相同，但它是类的成员，可以访问类的所有成员，使用与类成员访问权限和作用域限定的相同规则。此外，成员模板不能是虚函数。

【例 7-11】 OutArray 是一个数组输出的代理类，为它设计一个成员模板，用于输出指定大小的不同类型数组值。

为简化问题，可以重载 OutArray 类的仿函数 operator()（类的函数调用运算符）为模板成员函数，接收模板数组参数，并输出该数组中的数据元素。

```

// Eg7-11.cpp
#include<iostream>
using namespace std;

```

```

class OutArray {
public:
    OutArray(ostream& o = cout):os(o){ }
    template<typename T> void operator()(T *a, int n) {
        for(int i = 0; i < n; i++)
            os<<a[i]<<"\t";
        os<<endl;
    }
private:
    ostream &os;
};

void main() {
    double d[] = {1.2, 3.4, 5.6, 8, 9, 21};
    const char *c[] = {"abc", "efg", "der", "aa"};
    OutArray out; // 定义 OutArray 类对象
    out(d, 6); // 实例化 OutArray::operator(double *, int)
    out(c, 4); // 实例化 OutArray::operator(char *, int)
}

```

程序运行结果如下：

```

1.2  3.4  5.6  8  9  21
abc  efg  der  aa

```

7.4.6 可变参数函数模板 C++ 11

C++ 11 标准支持参数类型和个数都不确定的函数模板，即可变参数函数模板，为功能需求明确但数据类型和参数个数不确定的函数设计提供了更大的灵活性。可变参数函数模板的定义形式如下：

```

template<typename T1, typename ... T2>
r_type f(T1 p, T2 ... arg) { ... }

```

其中，T2 是可变模板参数，称为参数包，可以是零个或多个类型不同的模板参数；“...” 则是一个操作符，当 “...” 在参数左侧时表示打包，在参数右侧时则表示解包。

【例 7-12】 设计函数模板 mymax()，能够从任意多个数字中计算出最大值。

```

// Eg7-12.cpp
#include<iostream>
using namespace std;

template<typename T >
T mymax(T t) { // 结束条件
    return t;
}

template<typename T1, typename ...T2> // 打包，将多个类型参数打包到 T2
double mymax(T1 p, T2 ... arg) { // 打包，将多个函数参数打包到 arg
    double ret = mymax(arg ...); // 包扩展，在实例化时解包 arg
    if(p > ret)
        return p;
    else

```



```

        return ret;
    }

    void main() {
        cout<<mymax(1, 12, 3, 4, 20)<<"\t";           // 输出: 20
        cout<<mymax('5', 32, '2', 23.0)<<"\t";         // 输出: 53 ('5'的 ASCII 值)
        cout<<mymax('a', 'z', 2)<<"\t";               // 输出: 122 ('z'的 ASCII 值)
        cout<<mymax(2, 3.2)<<endl;                     // 输出: 3.2
    }

```

同函数模板一样，编译器根据模板函数调用的实参推断模板参数类型，对于可变参数模板，编译器将从实参中推断出参数的个数以及每个参数的类型。对于 `main()` 中的函数调用，编译器为 `mymax` 实例化生成下面的函数版本：

```

double mymax(int p, int arg, int arg, int arg, int arg);
double mymax(char p, int arg, char arg, double arg)
double mymax(char p, char arg, int arg)
double mymax(int p, double arg)

```

各模板函数中的粗体就是通过调用实参从 `mymax` 函数模板中的参数包 `T2` 推断出来的。

可变参数函数通常都是递归调用的：函数实现时往往先处理包中的第一个实参，再用包中的剩余实参调用函数，称为包扩展，类似于如下形式：

```

return_type f(T1 p, T2 ... arg) {
    处理 arg 中的第一个参数;
    f(...arg 中除第一个参数之外的其余参数);           // 递归调用
    .....
}

```

为了终止递归，应该为可变参数模板定义一个非可变参数的函数。如例 7-12 中的 `double mymax()` 就是用于结束递归的函数，它会在可变参模板函数的参数处理完毕时被调用，用于结束函数 `mymax()` 的递归调用。例如，`mymax('a', 'z', 2)` 调用的包扩展过程如下：

```

mymax('a', 'z', 2);           // <1> 包扩展函数
mymax('z', 2)                 // <2> 包扩展函数
mymax(2)                     // <3> 包扩展函数
mymax()                      // <4> 非模板函数，结束递归调用

```

7.4.7 元编程的基本概念

1994 年，Erwin Unruh 写了一个求解小于 N 的全部素数的 C++ 程序，该程序并不能通过编译，但编译器在错误信息中显示出了求解结果。这个程序证明了编译器具有计算能力，程序的功能并非必须在运行期才能够实现，在编译期也是可以实现的。

元编程 (meta-programming) 正是基于编译器具有计算能力这个基础演化出来的一种编程技术，其思想是：利用模板技术，编写出能够在编译期间计算出运行时需要的常数、类型、代码的元模板，该模板能够读取、分析、转换或者生成其他程序，甚至在运行时修改程序自身（反射），有人形象地称之为“编程序的程序”。

在设计数学统计之类运算（如求和、积、均值、最大值、最小值等）的可变参数函数模板时，要注意函数返回类型的设计。通常，可变参数函数模板是以第一个参数作为返回类型

的，这可能存在问题，产生错误运算结果。但是，利用元编程则可以解决该问题。

【例 7-13】 设计可变参数的函数模板 `mysum()` 计算任意多个类型不确定数字的总和。

```
// Eg7-13.cpp
#include<iostream>

template<typename T>
T mysum(T s) { return s; }
template<typename T, typename ... P>
T mysum(T t, P ... x) {                                     // C++ 14, 可用 auto mysum(T t, P ... x)
    return t + mysum(x ...);
}

int main() {
    auto s1 = mysum(2.0f, 4.3f, 10.1f, -3.6f);
    auto s2 = mysum(2, 4.3f, 10u, -45.2);
    std::cout<<"s1 = "<<s1<<std::endl;
    std::cout<<"s2 = "<<s2<<std::endl;
}
```

程序运行结果如下：

```
s1 = 12.8
s2 = -2147483648
```

`s1` 是正确的，`s2` 则显然不对。原因是计算 `s1` 中的各参数都是同一种数据类型，运算过程中不会产生因类型转换而损失的数据精度。但是，计算 `s2` 中的各参数类型都不相同，在运算过程中会发生多次类型转换，在转换过程中就会产生精度损失，因此会出现错误。按照递归函数的调用规则，最先被运算的是 `10u-45.2=-35.1`，这个计算没有问题。但是这次计算的返回类型是 `u`，因此 `-35.1` 将被转换在 `unsigned` 类型，会是一个很大的数字，然后对这个结果进行累加求和，以致产生错误结果。错误的根源是模板 `mysum` 的返回值类型存在问题。

```
T mysum(T t, P ...x) {
    return t + mysum(x ...);
}
```

这个模板始终以第一个参数的类型 `T` 作为返回值类型，问题就在这里。如果能够根据实际调用参数计算出表达式 “`t+mysum(x...)`” 结果值的数据类型，再将其作为模板 `mysum()` 的返回值类型，那么 `mysum()` 能够返回正确的结果。

但是，`t+mysum(x...)` 到底是什么数据类型呢？可能是 `t` 的类型 `T`，也可能是 `mysum(x...)` 的返回类型 `P`，但不论是哪一种，肯定都是可以计算出来的。在此，假设 `t+mysum(x...)` 的类型是结构 `SumType`（也可以是 `class`，用 `struct` 是因为其成员的默认访问权限是 `public`，可以简化问题），让编译器根据调用实参自动推导 `SumType` 的实际类型。由于模板 `mysum(x...)` 是递归定义的，因此只需要针对其递归定义，分别定义出 `mysum()` 只有一个参数和多个参数时的数据类型就行了，这可以通过模板重载实现。

【例 7-14】 基于元模板的可变参数函数模板计算多类型数字总和的函数 `mysum()`。

```
// Eg7-14.cpp
#include<iostream>
using namespace std;
```

```

template<typename ...P> struct SumType; // L1

template<typename T> struct SumType<T>{ // L2
    using type = T;
};

template<typename T, typename ...P> // L3
struct SumType<T,P...> {
    using type = decltype(T() + typename SumType<P...>::type()); // L4
};

template<typename ...P>
using sumRetType = typename SumType<P...>::type; // L5

template<typename T>
T mysum(T s) { return s; }

template<typename T, typename ...P>
sumRetType<T, P ...> mysum(T t, P ...x) { // L6
    return t + mysum(x ...);
}

int main() {
    auto s1 = mysum(2.0f, 4.3f, 10.1f, -3.6f);
    auto s2 = mysum(2, 4.3f, 10u, -45.2);
    std::cout<<"s1 = "<<s1<<std::endl;
    std::cout<<"s2 = "<<s2<<std::endl;
}

```

程序运行结果如下：

```

s1 = 12.8
s2 = -28.9

```

语句 L1 前向声明了结构 `SumType`，语句 L2、L3 通过重载方式定义了结构 `SumType`，该结构主要用于确定 `mysum(P...p)` 的返回类型。根据 `mysum()` 的递归递义，L2 定义了递归结束条件函数（`mysum()` 具有 0 或 1 个参数）的返回类型，即 `mysum(T t)` 的返回类型，当然就是模板参数类型 `T` 了。

语句 L3 以重载方式定义了表达式 “`t+mysum(...p)`” 的返回类型，实际上是 `mysum()` 递归运算过程中的函数返回类型。其中，`T()` 是 `t` 的类型，而 “`typename SumType<P...>::type()`” 中的 `typename` 是表达式类型识别运算符，可计算出 `SumType<P...>::type()` 的数据类型，这是一个递归定义。这样，“`t+mysum(...p)`” 中 “+” 左、右的类型就能够确定了，然后可用 `decltype` 推导出整个表达式的类型。

语句 L5 对元模板中的结构 `SumType` 内部的 `type` 成员进行了类型定义，并在语句 L6 中用作了函数板的返回类型。

在 C++ 11 的标准头文件 `<type_traits>` 中提供了类似于 `SumType` 的通用类型 `comm_type`（在 C++ 14 及以上标准中的名称为 `comm_type_t`），运用它和可变参数函数模板，可以方便实现泛型版的通用求和函数模板，改写后的 `mysum()` 版本如下，两者具有完全相同的功能。

【例 7-15】 运用 C++ 14 标准的 `common_type_t` 通用数据类型，设计泛型可变参数求和函数模板。

```
// Eg7-15.cpp
```

```

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

template<typename T>
T mysum(T s) { return s; }

template<typename T, typename ... P>
common_type_t<T, P ... > mysum(T t, P ... x) {           // C++ 14
    return t + mysum(x ...);
}

int main() {
    auto s1 = mysum(2.0f, 4.3f, 10.1f, -3.6f);
    auto s2 = mysum(2, 4.3f, 10u, -45.2);
    cout<<"s1 = "<<s1<<endl;
    cout<<"s2 = "<<s2<<endl;
}

```

程序运行结果如下，

```

s1 = 12.8
s2 = -28.9

```

7.4.8 模板重载、特化、非模板函数及调用次序

1. 模板重载

模板可以与另一个模板或普通函数同名，即**重载**。与函数重载的规则相同，要求重载的同名函数模板必须具有不同的形参表。

【例 7-16】 设计从两个数中找出最大值的函数模板，并重载该函数模板，实现从任意三个数中找出最大值的函数模板。

```

// Eg7-16.cpp
#include<iostream>
#include<string>
using namespace std;

template <typename T>
inline T const& Max(T const& a, T const& b) {
    return a < b ? b : a;
}

template <typename T>
inline T const & Max(T const& a, T const& b, T const& c) {
    return Max(Max(a, b), c);
}

int main(){
    int a = 5, b = 12;
    string s1 = "aString1", s2 = "aString2";
    const char* c1 = "hello template override!";
    const char* c2 = "hello C++ 11!";
    const char* c3 = "hello everyone!";
}

```

```

    cout<<Max(7, 42, 32)<<endl;           // L1
    cout<<Max(a, b)<<endl;                 // L2
    cout<<Max(s1, s2)<<endl;               // L3
    cout<<Max(c1, c2, c3)<<endl;           // L4
    cout<<Max(c1, c3)<<endl;               // L5
}

```

程序运行结果如下：

```

42                                     // L1 的输出
12                                     // L2 的输出
aString2                             // L3 的输出
hello everyone!                       // L4 的输出，错误
hello everyone!                       // L5 的输出，错误

```

从输出结果可以看出，重载的函数模板 `Max()` 能够正确地从 2 个或 3 个数值型和 `string` 类型的数据中找出最大值，但不能从 `char*` 类型的字符串中找出正确的最大值。

2. 模板特化

针对不同的数据类型，模板（包括函数模板和类模板）可以实例化出适用于该数据类型的可用函数或类，但要让一个模板实现对全部数据类型的正确处理，却不一定做得到。例如，在例 7-16 中，模板函数 `Max()` 不能从 `char*` 类型的 2 个或 3 个字符串中找出正确的最大值。原因很简单，`char*` 类型的字符串需要用函数 `strcmp()` 而不是 “<” 运算符比较其大小。为了解决这个问题，C++ 允许为模板定义针对某种数据类型的替代版本，称为 **模板特化**。特化是模板对指定数据类型的特殊实现，其语法形式如下：

```

template <>
用具体类型替换模板参数的函数模板或类模板；

```

“<>” 中不需要任何内容，表示模板特化。特化模板必须与原模板具有相同的结构，在设计某种数据类型的特化模板时，只需把原模板中的类型参数替换成指定数据类型后，重新编写函数模板（或类模板成员函数）的实现代码就行了。

【例 7-17】 为例 7-16 的函数模板 `Max()` 提供特化版本，找出两个 `char*` 类型字符串中的最大值。

在例 7-16 中添加处理 `const char*` 最大值计算的模板特化函数 `Max()`，用 `const char*` 替换模板 `Max()` 中的 `T`，其余代码不做任何修改。

```

// Eg7-17.cpp
.....
template<>
const char* const& Max(const char* const& a, const char* const& b) {
    return strcmp(a, b) < 0 ? b : a;
}

```

不修改程序的代码（包括函数 `main()`），程序运行结果如下：

```

42
12
aString2
hello template override!               // 正确

```

```
hello template override! // 正确
```

最后两行分别是 `Max(c1, c2, c3)`和 `Max(c1, c3)`的输出结果，它们是 `Max` 模板的特化版本从 `char*`类型的字符串中找出的最大值，这次是正确的。

3. 为模板补充普通函数

针对模板不能正确处理的特定数据类型，除了提供处理该类型的特化模板，还可以提供处理该类型的普通函数版本。

【例 7-18】 在例 7-17 中添加从两个 `const char*`类型的字符串中找出最大值的普通函数 `Max()`。

在例 7-17 中添加普通函数 `Max()`，其余代码不作任何修改，完整的程序代码如下：

```
// Eg7-18.cpp
#include<iostream>
#include<string>
using namespace std;

template <typename T>
inline T const& Max(T const& a, T const& b){ return a < b ? b : a; }

template <typename T>
inline T const& Max(T const& a, T const& b, T const& c){ return Max(Max(a, b), c); }

template<>
const char* const& Max(const char* const& a, const char* const& b){
    return strcmp(a, b) < 0 ? b : a;
}

inline char const* Max(char const* a, char const* b){
    return strcmp(a, b) < 0 ? b : a;
}

int main(){
    int a = 5, b = 12;
    string s1 = "aString1", s2 = "aString2";
    const char* c1 = "hello template override!";
    const char* c2 = "hello C++ 11!";
    const char* c3 = "hello everyone!";
    cout<<Max(7, 42, 32)<<endl; // L1 输出: 42
    cout<<Max(a, b)<<endl; // L2 输出: 12
    cout<<Max(s1, s2)<<endl; // L3 输出: aString2
    cout<<Max(c1, c2, c3)<<endl; // L4 输出: hello template override!
    cout<<Max(c1, c3)<<endl; // L5 输出: hello template override!
}
```

程序运行结果与例 7-17 完全相同，但是这次计算两个 `char*`类型字符串中的最大值时，调用的是普通函数 `Max()`，而不是函数模板的特化版本 `Max()`。

4. 函数参数匹配与调用次序

一个程序中可以同时拥有重载函数模板、特化模板和普通重载函数，对于每次函数调用，将从重载函数、普通函数以及能够通过调用实参推断出的模板函数中进行选择。当几者都与

调用函数相匹配时，编译器的选择次序如下。

① 在众多符合调用条件的函数中选择最佳匹配的函数，在匹配非模板函数时会进行参数类型转换。在匹配模板函数时，除了下面两种转换，不会进行其他的数据类型转换。

- ① **const 转换**：可以将一个非 **const** 对象的引用或指针传递给 **const** 类型的引用或指针形参；
- ② **数组或函数指针转换**：如果函数形参不是引用类型，就可以对数组或函数类型的实参应用正常的指针转换规则。即，数组实参可以转换为一个指向其首元素的指针，函数实参可以转换成该函数类型的指针。

② 如果模板函数、模板特化函数和普通函数都符合函数调用要求，就优先调用普通函数；如果没有普通函数，就优先调用模板特化函数；如果没有普通函数和特化模板函数，才调用模板函数。如果有多个重载模板函数都符合要求，就选择精确匹配的模板函数；如果多个模板函数与调用实参都能精确匹配，就会产生二义性错误。

在例 7-17 中，同时具有符合 **char*** 类型字符串最大值计算的模板函数 **Max()** 和模板特化函数 **Max()**，因此选择模板特化函数 **Max()**；在例 7-18 中，具有符合调用条件的模板函数 **Max()**、模板特化函数 **Max()** 和普通函数 **Max()**，因此选择普通函数 **Max()**。

7.5 STL 程序设计

STL (Standard Template Library) 即标准模板库，是 C++ 较晚引入的基于模板技术的程序库，提供了模板化的通用类和通用函数，包括容器、迭代器、算法三大核心内容，为各种编程问题提供高效的解决方案。

STL 提供了许多可以直接用于程序设计的通用数据结构和功能强大的类与算法，这些数据结构和算法是准确而高效的，其设计与实现经过了严格的标准化评估，能够保证其中的类和函数在每个符合 C++ 标准的编译器中正常运行，用来解决编程中的各种问题，可以减少程序测试时间，编写高质量的代码，提高编程效率。

7.5.1 函数对象

对于程序设计中经常使用的 +、-、*、/、% (取模) 等算术运算符，>、<、>=、<=、==、!= 等关系运算符，以及 !、||、&& 等逻辑运算符，STL 标准库都为它们定义了对应的运算符模板类，能够实现对应的运算符操作，称为函数对象。所有函数对象的定义都在 **functional** 头文件中，具体名称如表 7-2 所示，在程序中可以用这些模板定义对象，实现相应的运算。

【例 7-19】 函数对象应用示例。

```
// Eg7-19.cpp
#include<iostream>
#include<functional>
#include<algorithm>
using namespace std;

void main() {
    int a[] = {3, 1, 7, 0, -3, 2, 8, -5};
    plus<int> iadd;
```



```

        minus<double> dm;
        less<int> les;
        int s = iadd(5, 6);
        double d = dm(25, 5);
        cout<<"s = "<<s<<"\td = "<<d<<"\t";

        if(les(5, 7))
            cout<<"5<7"<<endl;
        else
            cout<<"5>7"<<endl;
        // L1, 从小到大排序 a 数组
        sort(a, a + 8, less<int>());
        for(int i = 0; i < 8; i++)
            cout<<a[i]<<"\t";
        cout<<endl;
        sort(a, a + 8, greater<int>());
        for(int i = 0; i < 8; i++)
            cout<<a[i]<<"\t";
        cout<<endl;
    }

```

表 7-2 STL 中的函数对象

算术对象	关系对象	逻辑运算对象
plus<T>	equal_to<T>	logical_and<T>
minus<T>	not_equal_to<T>	logical_or<T>
multiplies<T>	greater<T>	logical_not<T>
divides<T>	greater_equal<T>	
modulus<T>	less<T>	
megate<T>	less_equal<T>	

// L2, 从大到小排序 a 数组

程序运算结果如下:

```

s = 11      d = 20      5<7
-5  -3   0   1   2   3   7   8
8   7   3   2   1   0  -3  -5

```

程序中的“s = iadd(5, 6)”与“s = 5+6”等效，“d = dm(25, 5)”与“d=25-5”等效。从形式上，不如直接用+、-运算符更简单，但在后面即将介绍的 STL 容器、算法中，常会使用 functional 头文件中定义的函数对象进行模板设计。比如，上例语句 L1 和 L2 中的 less 和 greater 就是“<”和“>”比较运算符的函数对象。

sort()是在 algorithm 头文件中定义的 STL 排序函数模板，可以对由第 1、2 个参数指定的数组区间，采用由第 3 个参数（返回逻辑真值或假值的函数名称）指定的比较方法进行排序，less 是从小到大的排序，greater 则表示从大到小的排序方式。在没有指定第 3 个参数时，默认采用 less 方式排序。下面的两条语句是等效的：

```

sort(a, a + 8);
sort(a, a + 8, less<int>());

```

7.5.2 顺序容器

容器（container）是用类模板实现的、用来存储其他对象的对象。STL 的容器包括顺序容器、关联容器和容器适配器三类。

顺序容器是将相同类型对象的有限集按顺序组织在一起的容器，用来表示线性数据结构，常被称为序列容器，包括向量（vector）、链表（list）和双端队列（deque）。

关联容器是非线性容器，是用来根据键（key，又称为查询键）进行快速存储、检索数据的容器，这类容器可以存储值的集合或键-值对，主要包括集合（set）、多重集合（multiset）、映射（map）和多重映射（multimap）。键是关联容器中存储在有序对中的特定类型的值。集

合和多重集合存储与操作的只是键，其元素由单个数据构成；而映射与多重映射存储的键-值对，其元素是由<key-value>构造的复合数据。

容器适配器主要指堆栈（stack）和队列（queue），它们实际是受限访问的顺序容器类型。表 7-3 列出了 STL 库中大部分常用容器的名称及其所在的头文件名称。

表 7-3 STL 中的容器及头文件名

STL 容器名	头文件名	说 明
vector	<vector>	向量，从后面快速插入和删除，直接访问任何元素
list	<list>	双向链表
deque	<deque>	双端队列
set	<set>	元素不重复的集合
multiset	<set>	元素可重复的集合
stack	<stack>	堆栈，后进先出（LIFO）
map	<map>	一个键只对一个值的映射
multimap	<map>	一个键可对多个值的映射
queue	<queue>	队列，先进先出（FIFO）
priority_queue	<queue>	优先级队列

STL 是经过精心设计的，为了减小操作使用容器的难度，大多数容器提供了相同的成员函数。其中，以下成员函数是所有容器都支持的：

```
empty()           // 判断容器是否为空，若为空，返回 true，否则返回 false
max_size()        // 返回容器最大容量，即容器能够保存的最多元素个数
size()            // 返回容器中当前元素的个数
swap              // 交换两个容器中的元素
```

此外，各容器类还具有默认构造函数、复制构造函数、析构函数，并重载了=、<、<=、>、>=等运算符函数，可以直接进行两个容器之间的比较运算。

表 7-4 是只有 vector、deque、list、set、multiset、map、multimap 才支持的成员函数。

表 7-4 顺序和关联容器共同支持的成员函数

成员函数名	说 明	成员函数名	说 明
begin()	指向第一个元素	rbegin()	指向按反顺序的第一个元素
end()	指向最后一个元素之后的位置	rend()	指向按反顺序的末端位置
erase()	删除容器中的一个或多个元素	clear()	删除容器中的所有元素

1. vector

vector 是 STL 中最简单的向量容器，类似于 C 语言中的数组，能够将批量数据存储在连续的内存区域中，并可以像数组一样通过运算符[]访问其元素。但它与数组是不同的，数组必须在定义时确定大小，存储在栈上；vector 存储在堆中，在插入或删除数据时，能够自动扩展和压缩其大小，比数组更灵活。

如图 7-4 所示，v 是整型向量，begin()、end()是向量的头尾查找函数，rbegin()、rend()是反向查找向量头尾的函数，iterator 代表指向某个元素的迭代器，用于遍历向量。

1) vector 的构造（模板参数 T 是数组类型）

```
vector<T> c           // 产生一个空 vector，其中没有任何元素
```

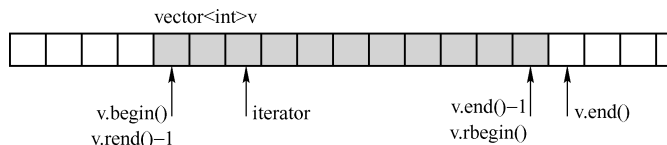


图 7-4 vector 示意

```
vector<T> c1(c2)           // 产生同型 c2 向量的一个副本 (c2 所有元素被复制给 c1)
vector<T> c(n, elem)       // 产生大小为 n 的向量 c, 且每个元素都是 elem
vector<T> c(beg, end)      // 产生一个向量 c, 并用区间 [beg, end] 作为元素的初值
```

2) 赋值操作

```
c1 = c2                   // 将向量 c2 的元素全部赋值给向量 c1
c.assign(n, e)            // 复制 n 个元素 e, 赋值给向量 c
c.assign(beg, end)        // 将区间 [beg, end] 内的元素赋值给 c
c1.swap(c2)              // 将 c1 和 c2 向量互换
```

3) 直接访问向量元素

```
c.at[n]                  // 返回下标 n 所标识的元素, 若下标越界, 返回 "out_of_range"
c[n]                     // 返回下标 n 所标识的元素, 不进行范围检查
c.front()                // 返回第一个元素
c.back()                 // 返回最后一个元素
```

4) vector 的常用操作

```
c.insert(pos, e)         // 在 pos 位置插入元素 e 的副本, 并返回新元素的位置
c.insert(pos, n, e)      // 在 pos 位置插入元素 e 的 n 个副本, 不返回值
c.insert(pos, beg, end)  // 在 pos 位置插入区间 [beg, end] 内的所有元素
c.push_back(e)           // 在尾部插入元素 e
c.pop_back()             // 删除最后一个元素
c.erase(pos)             // 删除 pos 位置的元素
c.erase(beg, end)        // 删除区间 [beg, end] 内的所有元素
c.clear()                // 删除所有元素, 清空容器
c.size()                 // 返回向量 c 中的元素个数
c.resize(n)              // 将 c 重新设置为大小为 n 个元素的向量, 如果 n 比原来的元素
                        // 多, 那么多出的元素常被初始化为 0
```

上述成员函数参数中涉及的位置 `pos` 都与 `vector` 的迭代器有关, 要操作这些成员函数, 必须定义对应向量的迭代器, 并通过迭代器访问 `pos` 指向的向量元素。

【例 7-20】 vector 的应用举例。

```
// Eg7-20.cpp
#include<iostream>
#include<vector>           // 向量头文件
using namespace std;

// 输出并删除整数向量的全部元素。注意, 本函数将倒序输出向量中的元素
void display(vector<int> &v) {
    while(!v.empty()) {
        cout<<v.back()<<"\t";           // 输出向量的尾部元素
        v.pop_back();                     // 删除向量尾部元素
    }
    cout<<endl;
}
```

```

void main() {
    int a[] = {1, 2, 3, 4, 5, 6};
    vector<int> v1, v2; // 定义只有 0 个元素的向量 v1、v2
    vector<int> v3(a, a+6); // 定义向量 v3, 并用 a 数组初始化该向量
    vector<int> v4(6); // 定义具有 6 个元素的向量 v4
    v1.push_back(10); // 在 v1 向量的尾部加入元素 10
    v1.push_back(11);
    v1.push_back(12);
    v1.insert(v1.begin(), 30); // 将 30 插到 v1 向量的最前面
    v2 = v1; // 将 v1 赋值给 v2, v2 与 v1 具有相同的元素
    v3.assign(3, 10); // 重置 v3, 且只有 3 个元素, 均为 10
    cout<<"v1: "; display(v1);
    cout<<"v2: "; display(v2);
    cout<<"v3: "; display(v3);
    v4[0] = 10; v4[1] = 20; // 用数组方式访问向量元素
    v4[2] = 30; v4[3] = 40;
    cout<<"v4: ";
    for(int i = 0; i < 6; i++)
        cout<<v4[i]<<"\t";
    cout<<endl;
    v4.resize(10); // 重置向量 v4 的大小, 已有元素不受影响
    cout<<"v4: ";
    display(v4);
}

```

程序运行结果如下:

```

v1:  12  11  10  30
v2:  12  11  10  30
v3:  10  10  10
v4:  10  20  30  40  0  0
v4:  0  0  0  0  0  0  40  30  20  10

```

本例用到了 `vector` 的多种常用操作方法, 包括定义、赋值、插入、删除、显示等内容, 请参见程序注释理解此运行结果, 借此理解 `vector` 的操作方法。

2. list

STL 中的 `list` 是一个双向链表, 可以从头到尾或从尾到头访问链表中的节点, 节点可以是任意数据类型, 如图 7-5 所示。链表中节点的访问常常通过迭代器进行。在图 7-5 中, `L` 是 `int` 类型的链表, 可用成员函数 `front()` 找到 `list` 的第一个元素, 用 `back()` 找到 `list` 的最后一个元素。迭代器 `iterator` 用于指向链表的节点, 通过它可以遍历整个链表。

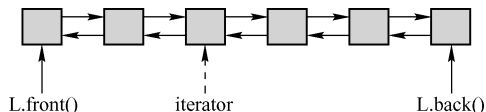


图 7-5 链表示意

1) 链表的构造 (模板参数 `T` 是链表的数据类型)

```

list<T> c // 建立一个空链表 c
list<T> c1(c2) // 建立与 c2 同型的链表 c1 (c2 的每个元素都被复制)

```

```
list<T> c(n) // 建立具有 n 个元素的链表 c，元素值由默认构造函数产生
list<T> c(n, e) // 建立 n 个元素的链表 c，每个元素的值都是 e
list<T> c(beg, end) // 建立一个链表 c，并用 [beg, end] 区间内的元素作初始化
c.~list<e>() // 销毁链表 c，释放内存
```

2) 链表赋值

```
c1=c2 // 将 c2 链表的全部元素赋值给 c1 链表
c1.assign(n, e) // 将元素 e 复制 n 次到 c1 链表
c.assign(beg, end) // 将区间 [beg, end] 的元素赋值给 c
c1.swap(c2) // 将链表 c1 和 c2 的全部元素互换
```

3) 链表存取

```
c.front() // 返回第一个元素，不检查元素存在与否
c.back() // 返回最后一个元素，不检查元素存在与否
```

4) 链表插入和删除

```
c.insert(pos, e) // 在 pos 位置插入元素 e 的副本，并返回新元素的位置
c.insert(pos, n, e) // 在 pos 位置插入元素 e 的 n 个副本，没有返回值
c.insert(pos, beg, end) // 在 pos 位置插入区间 [beg, end] 内的全部元素
c.push_back(e) // 在尾部追加一个元素 e 的副本
c.pop_back(e) // 删除最后一个元素
c.push_front(e) // 在表头插入元素 e 的一个副本
c.pop_front() // 删除第一个元素
c.remove(val) // 删除值为 val 的元素
c.remove_if(op) // 删除所有“造成 op(e) 结果为 true”的元素
c.erase(pos) // 删除 pos 指向的元素，返回下一元素的位置
c.erase(beg, end) // 删除区间 [beg, end] 内的所有元素，返回下一元素位置
c.resize(n) // 将链表 c 的大小重新设置为 n
c.clear() // 删除链表所有元素，将整个容器置空
```

5) 链表的特殊操作

```
c.unique() // 若存在多个相邻且值相等的元素，则删除重复元素，只留一个
c.unique(op) // 若存在若干相邻且使 op() 操作为 true 的元素，则删除重复，只留一个
c1.splice(pos, c2) // 将 c2 内的所有元素转移到 c1 的 pos 之前
c1.splice(pos, c2, c2pos) // 将 c2 链表的 c2pos 所指元素移到 c1 的 pos 指向的位置
c1.splice(pos, c2, c2beg, c2end) // 将 c2 的 [c2beg, c2end] 区间的所有元素转移到 c1 的 pos 前
c.sort() // 以 operator< 为准则，对所有元素排序
c.sort(op) // 以 op() 为准则，对所有元素排序
c1.merge(c2) // c2 中的全部元素合并到 c1，若 c1、c2 都已排序，则合并后 list 仍有序
c.reverse() // 将所有元素反序
```

说明：

① 上面涉及的 c、c1、c2 都是链表。

② op 可以是 less<> 或 greater<> 之一，应用时须在“<>”中写上类型，如 greater<int>。

less 指定排序方式为从小到大，greater 指定排序方式为从大到小，默认排序方式为 less。

【例 7-21】 list 应用的一个例子。

```
// Eg7-21.cpp
#include<iostream>
#include<list> // 链表头文件
using namespace std;
```

```

void main() {
    int i;
    list<int> L1, L2; // 定义两个没有元素的整数链表 L1 和 L2
    int a1[] = {100,90,80,70,60};
    int a2[] = {30,40,50,60,60,60,80};
    for(i = 0; i < 5; i++)
        L1.push_back(a1[i]); // 将 a1 数组加入 L1 链表
    for(i = 0; i < 7; i++)
        L2.push_back(a2[i]); // 将 a2 数组加入 L2 链表
    L1.reverse(); // 将 L1 链表倒序
    L1.merge(L2); // 将 L2 合并到 L1 链表中
    cout<<"L1 的元素个数为: "<<L1.size()<<endl; // 输出 L1 的元素个数
    L1.unique(); // 删除 L1 中相邻位置的相同元素, 只留 1 个
    while(!L1.empty()) {
        cout<<L1.front()<<"\t"; // 输出 L1 链表的链首元素
        L1.pop_front(); // 删除 L1 的链首元素
    }
    cout<<endl;
}

```

本程序的运行结果如下:

```

L1 的元素个数为: 12
30  40  50  60  70  80  90  100

```

在合并语句 L2 到 L1 前, L1 和 L2 都是有序的, 所以合并后的 L1 也是有序的。合并前的 L1 有 5 个元素, L2 有 7 个元素, 则合并后的 L1 有 12 个元素。执行 L1.unique()后, 删除了 L1 相邻的相同元素, 最后只有 8 个元素了。

3. stack

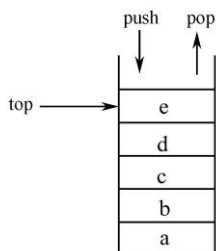


图 7-6 堆栈示意

堆栈 (stack) 又称为栈, 是一种受限制的向量, 只允许在向量的一端存取元素, 后入栈的元素先出栈, 即 LIFO (Last In First Out), 如图 7-6 所示。STL 中的堆栈提供的主要操作如下:

```

push() // 将一个元素加入 stack, 加入的元素放在栈顶
top() // 返回栈顶元素的值
pop() // 删除栈顶元素

```

top()与 pop()不同, top()只返回栈顶元素的值, 不删除元素, 而 pop()只删除栈顶元素, 不返回值。

【例 7-22】 STL stack 应用的例子。

```

// Eg7-22.cpp
#include<iostream>
#include<stack>
using namespace std;

void main() {
    stack<int> s;
    s.push(10);    s.push(20);    s.push(30);
    cout<<s.top()<<"\t";
    s.pop();    s.top() = 100;
}

```

```

s.push(50);    s.push(60);
s.pop();
while(!s.empty()) {
    cout<<s.top()<<"\t";
    s.pop();
}
cout<<endl;
}

```

程序运行结果如下（请读者分析，结果中为何没有 20 和 60）：

```

30    50    100    10

```

4. string

STL 中的 `string` 是一种特殊类型的容器，除了可以作为字符类型的容器，更多的是作为一种数据类型——字符串，可以像 `int`、`double` 之类的基本数据类型一样定义 `string` 类型的变量，并进行各种运算。本书 1.4.7 节已经对 `string` 的定义、赋值和比较进行了介绍，这里补充介绍 `string` 类的常用成员函数，以及它与 `char*` 字符串的区别。

1) `string` 的常用成员函数

假设 `s1`、`s2` 的定义如下：

```

string s1 = "ABCDEFGH";
string s2 = "0123456123";
string s;

```

`string` 成员函数如下：

```

substr(n1, n)    // 取子串函数，从当前字符串的 n1 下标开始，取出 n 个字符。如 s=s1.substr(2, 3)
                  // 的结果为：s="CDE"

swap(s)          // 交换字符串。如 s1.swap(s2)的结果为：s1="0123456123", s2="ABCDEFGH"

size()/length()  // 计算字符串中当前存放的字符个数。如 s1.length()的结果为：7

capacity()        // 计算字符串的容量（可容纳的字符个数）。如 s1.capacity()的结果为：31

max_size()        // 计算 string 类型数据的最大容量。如 s1.max_size()的结果为：4294967293

find(s)           // 在当前字符串中查找子串 s，若找到，则返回 s 在当前串中的起始位置，否则返回常数
                  // string::npos。如 s1.find("EF")的结果为：4

rfind(s)          // 同 find，但从后向前进行查找。如 s1.rfind("BCD")的结果为：1

find_first_of(s)  // 在当前串中查找子串 s 第一次出现的位置。如 s2.find_first_of("123")的结果
                  // 为：1

find_last_of(s)   // 在当前串中查找子串 s 最后一次出现的位置。如 s2.find_last_of("123")的结果
                  // 为：9

replace(n1, n, s) // 替换当前字符串中的字符，n1 是替换的起始下标，n 是要替换的字符个数，s 是用来
                  // 替换的字符串。如 s1.replace(2, 3, s2)的结果为：s1="AB0123456123FH"

replace(n1, n, s, n2, m) // n1 是替换的起始下标，n 是替换掉的字符个数，s 是用来替换的字符串
                  // n2 是 s 中用来替换的起始下标，m 是 s 中用于替换的字符个数。如
                  // s1.replace(2, 3, s2, 2, 3)的结果为：s1="AB234FH"

insert(n, s)       // 在当前串的下标位置 n 之前，插入 s 串。如 s1.insert(2, "88888")的结果为：
                  // s1="AB88888CDEFH"

insert(n1, n, s, n2, m) // 在当前串的下标位置 n1 下标后插入 s 串，n2 是 s 串中要插入的起始下标，m 是 s
                  // 串中要插入的字符个数。如 s1.insert(2, s2, 3, 2)的结果为：s1="AB34CDEFH"

```

2) `string` 与 C 语言形式的 `char*` 字符串的转换

尽管 `string` 与 `char*` 本质上都是字符串数据类型，但它们是有区别的：① `string` 是复杂的

模板类，而 `char*` 是简单类型 `char` 的指针；② `string` 类型的字符串不需要 `null` (`'\0'`) 结束符，而 `char*` 需要，因此不能将 `string` 类型的串直接赋值给 `char*` 类型的字符串。

如果需要将 `string` 类型的串转换成 `const char*` 类型的串，就可用 `string` 的成员函数 `data()` 实现。如果需要将一般类型的 `char*` 串赋值给 `string` 类型的串，就可用 `string` 类的成员函数 `copy()` 完成。下面的代码段说明了它们的用法：

```
string s1 = "ABCDEFH";
const char* cs1;
cs1 = s1.data();           // 函数 data() 只适用于赋值给 const char* 类型的串
char *cs2;
int len = s1.length();    // 计算 string 类型串的长度
cs2 = new char[len+1];    // 分配 char* 串的存储空间
s1.copy(cs2, len, 0);     // 复制 string 串的内容到 char* 串
cs2[len] = 0;             // 因 string 串中没有 null，将它加在 char* 串的最后，上述代
                           // 码将 cs1 和 cs2 字符串赋值为 "ABCDEFH"
```

【例 7-23】 `string` 应用的例子。

```
// Eg7-23.cpp
#include<iostream>
#include<string>
using namespace std;

void main() {
    string s1 = "中华人民共和国成立了";
    string s2 = "中国人民从此站起来了！";
    string s3, s4, s5;
    s3=s1+", "+s2;
    int n = s1.find_first_of("人民");
    if(n != string::npos)
        cout<<"人民在 s1 中的位置: "<<n<<endl;
    else
        cout<<"在 s1 中没有该子串！";           // npos 是没有找到时的函数返回值
    s4 = s1.substr(4, 10);
    cout<<"s1 = "<<s1<<endl;
    cout<<"s2 = "<<s2<<endl;
    cout<<"s3 = "<<s3<<endl;
    cout<<"s4 = "<<s4<<endl;
    if(s1 > s2)
        cout<<"s1 > s2 = true"<<endl;
    else
        cout<<"s1 > s2 = false"<<endl;
    s3.replace(s3.find("从此"), 4, "从 1949 年");
    cout<<"s3 after replace = "<<s3<<endl;
    s3.insert(s3.find("站"), "10 月");
    cout<<"s3 after insert = "<<s3<<endl;
}
```

程序运行结果如下：

```
人民在 s1 中的位置: 4
s1 = 中华人民共和国成立了
```

```

s2 = 中国人民从此站起来了！
s3 = 中华人民共和国成立了，中国人民从此站起来了！
s4 = 人民共和国
s1 > s2 = true
s3 after replace = 中华人民共和国成立了，中国人民从 1949 年站起来了！
s3 after insert = 中华人民共和国成立了，中国人民从 1949 年 10 月站起来了！

```

7.5.3 迭代器

迭代器（iterator）是 STL 的核心抽象，是解耦数据结构和算法的基础方法，是基于模板的一种可以解引用、比较和改变引用位置的通用型指针，提供了适用于多种容器类型的通用操作，能够把算法和容器有效地组织起来协同工作，实现强大的程序功能。其中的主要操作如下：

```

operator *                // 返回当前位置上的元素值
operator++/operator--    // 将迭代器前进/后退一个元素位置
operator==/operator!=    // 判定两个迭代器是否指向同一个位置
operator=                 // 为迭代器赋值
begin()/rbegin()         // 指向容器起点（第一个/最后元素）位置
end()/rend()             // 指向容器的结束点，结束点在最后（第一个）元素之后（之前）

```

begin()和 end()返回的迭代器构成了“左闭右开”的开区间，rbegin()和 rend()用于反向顺序操作容器，构成的是“左开右闭”的区间。

1. 用 iterator 定义迭代器

STL 中的每种容器都提供了遍历它的迭代器，所有算法都是通过迭代器实现的。若某容器要使用迭代器，就必须先定义迭代器，然后才能够使用。方法是，用对应容器类型限定的 iterator 定义迭代器对象，其形式定义如下：

```
X<type>::iterator iter;
```

其中，X 代表 STL 中的容器，type 代表数据类型。例如，定义 int 类型的链表和迭代器：

```
list<int> L1;
list<int>::iterator iter;
```

其中，iterator 是定义迭代器的关键字，list<int>是迭代器能够访问的容器类型。完成此定义后，迭代器 iter 可用于访问所有 int 型的 list。

【例 7-24】 链表迭代器应用举例。

```

// Eg7-24.cpp
#include<iostream>
#include<list>
using namespace std;

int main() {
    int i;
    list<int> L1, L2, L3(10);                // L1、L2 是空链表，L3 是有 10 个元素的链表
    list<int>::iterator iter;                // 定义迭代器 iter
    int a1[] = {100, 90, 80, 70, 60};
    int a2[] = {30, 40, 50, 60, 60, 60, 80};
    for(i = 0; i < 5; i++)

```

```

        L1.push_back(a1[i]); // 插入 L1 链表元素，在表尾插入
    for(i = 0; i < 7; i++)
        L2.push_front(a2[i]); // 插入 L2 链表元素，在表头插入
    // 通过迭代器 iter 顺序输出 L1 的所有元素。迭代器是指针，访问 iter 指向的元素
    for(iter = L1.begin(); iter != L1.end(); iter++)
        cout<<*iter<<"\t" ;
    cout<<endl;
    int sum = 0;
    for(iter = --L2.end(); iter != L2.begin(); iter--){ // 通过迭代器反向输出 L2 的所有元素
        cout<<*iter<<"\t";
        sum+=*iter; // 计算 L2 所有链表节点的总和
    }
    cout<<"\nL2: sum = "<<sum<<endl;
    int data = 0;
    for(iter = L3.begin(); iter != L3.end(); iter++) // 通过迭代器修改 L3 链表的内容
        *iter = data += 10; // 修改迭代器所指节点的数据
    for(iter = L3.begin(); iter != L3.end(); iter++)
        cout<<*iter<<"\t";
    cout<<endl;
    return 0;
}

```

程序运行结果如下：

```

100 90 80 70 60
30 40 50 60 60 60
L2: sum = 300
10 20 30 40 50 60 70 80 90 100

```

本程序为整数类型的 list 定义了一个迭代器 iter，该迭代器可用于遍历任何 int 类型的 list 容器；定义了三个链表 L1、L2、L3，通过迭代器 iter 顺序遍历 L1，并输出 L1 的所有元素；利用迭代器 iter 反向访问 L2 的元素，反向输出各元素的值，并计算 L2 所有元素的总和；同时，通过迭代器 iter 修改了 L3 链表所有元素的值。

2. 用 auto 定义迭代器

自 C++ 11 标准开始，可以用 auto 自动推导迭代器的类型，编译器通过容器的 begin() 和 end() 函数的返回值可以自动推断出迭代器的类型，如果在遍历容器的循环控制程序中用它来定义循环变量，就可以使程序代码更加简洁。此外，STL 提供了适用于各种容器类型的自由函数 begin() 和 end()，它们也能够返回容器参数的类型，用法如下：

```

begin(x); // 返回 x 容器的首元素位置
end(x); // 返回 x 容器最后元素的后一个位置

```

其中，x 是用 STL 中的容器定义的对象，函数返回迭代器。

【例 7-25】 设计一个包括多名学生的链表，每位学生有姓名和年龄，输出每名学生的姓名和年龄，以及所有学生的平均年龄。

设计思路：设计结构 Student，存储和输出学生的姓名和年龄信息；在 list 中保存多名学生信息；分别用学生链表的容器类型和 auto 自动推导类型两种方法定义迭代器并遍历链表。为了减少重复代码；为 Student 定义成员函数 oudData()，输出学生信息。

```
// Eg7-25.cpp
#include<iostream>
#include<list>
#include<string>
using namespace std;

struct Student {
    string name;
    int age;
    void outData() { cout<<name<<"\t"<<age<<"\t\t"; }
};

int main() {
    list<Student> L = {"张三", 21}, {"李四", 18}, {"黄明", 27};
    double avg = 0;
    int n = 0;
    for (auto iter = L.begin(); iter != L.end(); iter++) {           // L1
        iter->outData();                                           // L2
        avg += iter->age;
        n++;
    }
    cout<<"avg = "<<avg/n<<endl;
    n = 0;    avg = 0;
    for(auto iter = begin(L); iter != end(L); iter++) {           // L3
        iter->outData();                                           // L4
        avg += iter->age;
        n++;
    }
    cout<<"avg = "<<avg /n<< endl;
    for(auto x:L)                                                  // L5
        x.outData();                                              // L6
}
```

程序运行结果如下:

张三	21	李四	18	黄明	27	avg=22
张三	21	李四	18	黄明	27	avg=22
张三	21	李四	18	黄明	27	

这 3 行数据分别由语句 L2、L4、L6 的 outData() 语句输出(平均年龄由相近的 cout 输出)。语句 L1 通过 auto 从 list 容器的成员函数 begin() 中自动推导出迭代器 iter 的类型, 语句 L3 则通过 auto 从 STL 库中的自由函数 begin() 中推导出 iter 的类型。

语句 L5 的范围 for 用 auto 自动推断 x 的类型并遍历输出了链表 L 中的学生信息。一般情况下, 只要 begin() 和 end() 返回的是迭代器, 就可以用范围 for 遍历它。由于 STL 中的所有容器都提供了成员函数 begin() 和 end(), 因此都可以用范围 for 进行遍历。为了减少范围 for 遍历时复制数据的开销, 可以采用引用类型的自动推断, 如用下面的方式推断语句 L5 中的 x:

```
for (auto& x:L) {
    x.outData();
}
```

说明: STL 中的迭代器可分为双向迭代器、前向迭代器、后向迭代器、输入迭代器和输

出迭代器几种。前面介绍的迭代器属于双向迭代器，读者若要了解其他迭代器的用法，可参考 C++ 的帮助文档或 STL 的相关书籍。

7.5.4 pair 和 tuple 容器

pair 和 tuple 也是一种顺序容器，pair 是由<键, 值>构成的值对数据类型，tuple 则可以是任意个不同数据类型组合成的一种复杂结构的顺序容器，它们为复杂数据类型的构造和多功能函数的设计提供了便利。在过去，如果函数要返回多个计算结果，就需要通过指针或引用参数实现，代码复杂，过程烦琐。现在使用元组实现这样的功能，问题就简单了。

1. pair 值对

pair 是在头文件 `utility` 中定义的一个值对模板类型，主要用来把两个有关联的数据组合成一个数据结构，两个数据可以是同一类型或者不同类型。例如，`pair<int, float>` 把一个 `int` 和一个 `float` 组合成一种数据结构。

1) pair 对象构造

模板参数中的 T1、T2 可以是任意数据类型

```
pair<T1, T2> p1; // 使用默认构造函数
pair<T1, T2> p2(v1, v2); // 用给定值 v1、v2 初始化
pair<T1, T2> p3(p2); // 用复制构造函数初始化
pair<T1, T2> p4{v1, v2}; // 用 v1、v2 列表初始化, C++ 11
```

例如：

```
pair<int, double> p1, p2(1, 5000); // p1 使用默认构造函数
pair<string, string> student{"张大明", "信息管理与信息系统专业"};
pair<int, double> p3(p2);
```

2) pair 元素访问

pair 实质上是一个结构体，提供了 `first` 和 `second` 两个公有成员，用于访问 pair 的两个数据成员。例如：

```
pair<char*, double> p2("讲师", 3000);
p2.first = "教授"; // 将“讲师”修改为“教授”
p2.second = 3500; // 将 3000 修改为 3500
cout<<p2.first<<"\t"<<p2.second<<endl; // 输出结果：教授 3500
```

3) 赋值操作

两个 pair 对象，如果它们的 `first` 和 `second` 成员类型相同，就可以相互赋值；此外，可以用函数模板 `make_pair()` 创建 pair 对象，然后把它赋给 pair 对象。

```
pair<string, double> s1, s2;
s1 = make_pair("李明海", 90); // 创建 pair 对象再赋值
s2 = s1; // 同类型 pair 对象直接赋值
```

pair 类型通常用来构造其他容器中的元素，如 `map` 中的映射值对，或者 `vector`、`list`、`set` 中的元素；也常用于函数的返回类型，可以一次返回一个值对，如学生的姓名和成绩、职工的姓名和工资，使函数代码更为精练和简洁。

2. tuple 元组 ^{C++11}

tuple 是 tuple 头文件中定义的一种容器模板，称为元组。pair 只能有两个元素，但 tuple 可以有多个元素。STL 的许多容器虽然可以存取多个元素，但每种容器只能存取用一种数据类型，而 tuple 可以有任意多个不同类型的元素（当然，定义完成后的 tuple 元组中，构成元素的成员数目是固定的），并且可以用 STL 中的其他容器，如 list、vector、map、set 等，作为元组的构成成员，建立随意而功能强大的数据结构。

1) tuple 对象构造

定义一个 tuple 时，需要指出每个成员的数据类型，可以使用 tuple 的默认构造函数或值列表方式对每个成员进行初始化，形式如下：

```
tuple<T1, T2, ... Tn> t;           // 使用默认构造函数
tuple<T1, T2, ... Tn> t(v1, v2 ... vn); // 使用指定值初始化
tuple<T1, T2, ... Tn> t{v1, v2 ... vn}; // 用值列表初始化
```

例如，定义时，通过元组构造函数初始化元组对象。

```
tuple<int, string, const char*> t1, t2(1, "数据结构", "3.5 学分");
tuple<string, vector<double>, int, list<int>> someVal("constants", {3.12, 2.34, 32}, 42, {0, 1, 1, 1});
```

在定义 tuple 时，可以用列表直接初始化。例如：

```
tuple<int, int, int> t = {1, 2, 3};
tuple<int, int, int> t{1, 2, 3};
```

2) tuple 元素访问

tuple 中的每个数据成员都是 public 属性，且每个成员都可以是对象、数组之类的复杂数据类型。C++ 标准模板中提供了一个函数模板 get()，它以类似于数组下标索引的方式访问元组中指定位置的成员，用法如下：

```
get<i>(t)           // t 是元组，i 是元组中的元素位置，第 1 个元素位置为 0
```

其中，t 是一个 tuple 对象，“<>”中的值必须是常量表达式，表示访问第几个成员，返回指定成员的引用。例如，表示某考生各科成绩的元组如下：

```
tuple<const char *, int, vector<string>, list<int>> >
tue("tom", 101, {"语文", "数学", "英语"}, {76, 87, 91});
get<0>(tue)           // 返回考生姓名: "tom"
get<1>(tue)           // 返回考生考号: 101
get<2>(tue)           // 返回考试科目: {"语文", "数学", "英语"}
get<3>(tue)           // 返回科目成绩: {76, 87, 91}
```

3) tuple 操作

两个同类型的 tuple 可以进行小于 (<)、相等 (==) 和赋值 (=) 运算。此外，可以用函数 make_tuple() 构造 tuple 对象，用 auto 通过调用参数的类型推断 tuple 类型等。例如：

```
auto t=make_tuple("string", 3, 2.1);
```

编译器会根据实参推断元组 t 的类型为：tuple<const char*, int, double>，等价于如下定义：

```
tuple<const char *, int, double> t("string", 3, 2.1);
```

当希望将一些类型不同但具有联系的数据组合成单一对象而又不想定义类或结构时，用元组把这些数据组合起来（快而随意的数据结构）就显得非常有用。此外，用元组作为函数的返回类型可以一次返回多个数据。

【例 7-26】 元组应用方法案例。在一个成绩系统中，要求函数返回的成绩数据包括：学生姓名、学号、专业和 5 门课程的成绩，用元组处理数据可以简化程序设计。其中，函数 `inputData()` 用于说明为元组中的成员输入数据，并通过函数返回元组的方法；函数 `display()` 用于说明向函数传递元组参数的方法。

```
// Eg7-26.cpp
#include <tuple>
#include<string>
#include<list>
#include <vector>
#include<iostream>
using namespace std;

struct Grade {                                     // 表示学生科目、成绩的数据结构
    string  courseName;
    double grade;
    Grade(string s, double g):courseName(s), grade(g) {}
};
typedef tuple<string, int, string, vector<Grade>> Student;
// 函数返回的 Student 是一种复杂的元组数据类型，该元组保存学生的姓名、学号、专业，以及任意多门课程的成绩
Student inputData() {
    Student stu;
    cout<<"输入学生数据: 姓名, 学号, 专业"<<endl;
    cin>>get<0>(stu)>>get<1>(stu)>>get<2>(stu);      // 元组简单元素的访问方法
    string cName;
    double score = 0;
    int i = 1;
    while(cName != "exit") {
        cout<<"输入第"<<i++<<"科目名称, 输入 exit 结束:\t";
        cin>>cName;
        if(cName == "exit")
            break;
        cout<<"成绩:\t";
        cin>>score;
        get<3>(stu).push_back(Grade(cName, score)); // 向元组向量元素中添加对象
    }
    return stu;
}

void display(Student student) {
    cout<<get<0>(student)<<"\t"<<get<1>(student)<<"\t"<<get<2>(student)<<"\t"<<endl;
    // for 循环示范了访问元组中具有不确定个数的向量元素的访问方法
    for(int i = 0; i < get<3>(student).size(); i++)
        cout<<get<3>(student)[i].courseName<<"\t"<<get<3>(student)[i].grade<<endl;
}

void main() {
    auto t = make_tuple("string", 3, 20.01);        // 用 auto 和 make_tuple 定义元组的方法
    tuple<const char*, int, double> tt("string", 3, 20.01);
    tuple<int, int, int > t3 = {1,2,3};             // 列表初始化
    tuple<int, int, int > t5{1,2,3};
```



```

tuple<int, string, const char*> t1, t2{1, "数据结构", "3.5 学分"};
t1 = t2; // 同类型元组可以赋值
cout<<get<0>(t1)<<"\t"<<get<1>(t1)<<"\t"<<get<2>(t1)<<endl; // 元组访问的常规方法

// 下面的代码段示例了具有链表、向量元素的元组定义方法，以及元组中的链表访问方法
tuple<string, vector<double>, int, list<int>> vtable("tomoto", {3.12, 2.34}, 42, {10, 8, 9});
list<int>::iterator iter; // 访问链表的迭代器
for(iter = get<3>(vtable).begin(); iter != get<3>(vtable).end(); iter++)
    cout<<*iter<<"\t";
cout<<endl;

// student 对象示例了向元组中的对象数组赋值的方法
Student student{"李四", 1011, "计算机专业", {{{"英语", 76.4}, {"高数", 87}, {"C++语言", 89}}};
display(student);
student = inputData();
cout<<endl;
display(student);
}

```

程序运行结果如图 7-7 所示，该程序通过键盘输入“张大明”同学的 3 科成绩，请读者结合程序中的注释分析程序运行结果的输出数据。

```

1      数据结构      3.5学分
10     8          1      9
李四   1011      计算机专业
英语   76.4
高数   87
C++语言 89
输入学生数据: 姓名, 学号, 专业
张大明 1201 计算机软件
输入第 1 科目名称, 输入Exit结束:      数据库原理
成绩: 89
输入第 2 科目名称, 输入Exit结束:      数据结构
成绩: 76
输入第 3 科目名称, 输入Exit结束:      C++程序语言
成绩: 97
输入第 4 科目名称, 输入Exit结束:      exit
张大明 1201 计算机软件
数据库原理 89
数据结构 76
C++程序语言 97
请按任意键继续. . .

```

图 7-7 程序运行结果

7.5.5 关联式容器

STL 关联式容器包括集合和映射两大类，分别是 `set` 和 `multiset`、`map` 和 `multimap`，它们通过关键字（也称为查找关键字）存储和查找元素。在每种关联容器中，元素是按关键字有序存储的，容器遍历就以此顺序进行。因此，关联容器不支持 `push_front()` 和 `push_back()` 之类的操作，因为这些操作会与排序规则相冲突，没有支持的必要。

1. `set` 和 `multiset`

集合类 `multiset` 和 `set` 提供集合的操作，集合中的元素称为**关键字**，两者提供的操作方法基本相同，只是 `multiset` 允许元素重复而 `set` 不允许重复。如果向 `set` 集合中插入相同的元素，`set` 会忽略它；向 `multiset` 集合插入相同元素时，则不会有问题，如图 7-8 所示。

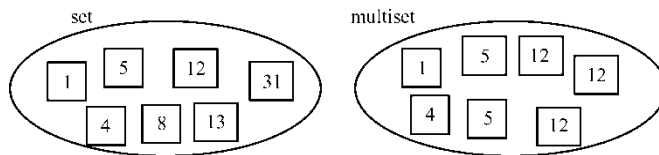


图 7-8 set 和 multiset

set 和 multiset 中的元素是有序存储的（排序树），在插入或删除元素时会自动进行元素排序。因此，如果要创建自定义类型（如 struct 或 class）的集合，就必须提供它们比较大小的成员函数，并将该成员函数设置为 const 函数。因为集合中的元素在默认情况下是升序排列的，用的是小于运算符函数（operator<）进行大小比较，该函数的返回结果是一个布尔值。所以最简单的方法是重载自定义类型的成员函数 operator<()。当然，也可以用普通函数方式重载运算符 operator<。形式如下：

```
class A {
    T x;
    .....
    bool operator<(A o) const { return x<o.x; }
}
```

以普通函数方式重载 operator<()运算符函数的形式：

```
bool operator<(A a, A b) { return a.getx() < b.getx(); }
```

其中，getx()获取对象的私有数据成员 x。当然，将 operator<()重载为类的友元可直接读 x。

1) set 和 multiset 的定义

```
set/multiset<T, [op]> c           // 建立一个空的 set/multiset 集合
set/multiset<T, [op]> c(op)       // 以 op 为排序准则建立一个空集
set/multiset<T, [op]> c1(c2)      // 建立一个集合 c1，并用集合 c2 初始化 c1
set/multiset<T, [op]> c(beg, end) // 用区间[beg, end]建立一个集合 c
```

其中，op 是排序运算符函数，可以是 less<>或 greater<>之一，应用时须在<>中写上类型，如 greater<int>。less 指定排序方式为从小到大，greater 指定排序方式为从大到小。op 可以省略，省略时的默认排序方式为 less。

2) set 和 multiset 的赋值比较运算

set 和 multiset 支持>、>=、<、<=、!=、==比较运算。

例如，若有集合 c1、c2，则可以用 c1==c2、c1>c2 对它们进行相等或大于判断。另外，可以用赋值运算符“=”进行集合赋值，如 c1=c2。

3) set 和 multiset 容量计算

```
size()           // 计算容器的大小
empty()          // 判断容器是否为空，若为空，则返回 0
max_size()       // 返回容器能够保存的最大元素个数
```

4) set 和 multiset 常用操作

```
count(e)         // 计算集合中元素 e 的个数
find(e)          // 查找集合中第一次出现元素 e 的位置
lower_bound(e)   // 查找集合中第一个“元素值>=e”的位置
upper_bound(e)   // 查找集合中第一个“元素值<e”的位置
insert(e)        // 在当前集合中插入元素 e
```

insert(pos, e)	// 将 e 插入 pos 位置
insert(beg, end)	// 将 [beg, end] 区间内的所有元素插入当前集合
erase(e)	// 删除集合中的元素 e
erase(pos)	// 删除集合中指定位置 pos 的元素
erase(beg, end)	// 删除区间 [beg, end] 的所有元素
clear()	// 清空集合
begin()	// 指向第一个元素位置, 常与迭代器结合应用
end()	// 指向最后元素的下一位置, 常与迭代器结合应用

【例 7-27】 集合应用的例子。

```
// Eg7-27.cpp
#include<iostream>
#include<string>
#include<set>
#include <functional>
using namespace std;

void main() {
    int a1[] = {-2, 0, 30, 12, 6, 7, 12, 10, 9, 10};
    set<int, greater<int>> set1(a1, a1+7);           // 定义从大到小排序的整数集合
    set<int, greater<int>>::iterator p1;             // 迭代器的定义要与集合排序相符
    set1.insert(12);    set1.insert(12);           // 向集合插入元素
    set1.insert(4);
    for(p1 = set1.begin(); p1 != set1.end(); p1++)
        cout<<*p1<<" ";                          // 输出集合中的内容, 它是从大到小的
    cout<<endl;
    string a2[] = {"杜明", "王为", "张清山", "李大海", "黄明浩", "刘一", "张三", "林浦海", "王小二", "张清山"};
    multiset<string>set2(a2, a2+10);                // 字符串 multiset 集合, 从小到大排序
    multiset<string>::iterator p2;
    set2.insert("杜明");    set2.insert("李则");
    for(p2 = set2.begin(); p2 != set2.end(); p2++)
        cout<<*p2<<" ";                          // 输出集合内容
    cout<<endl;
    string sname;
    cout<<"输入要查找的姓名: ";
    cin>>sname;                                     // 输入要在集合中查找的姓名
    p2 = set2.begin();
    bool s = false;                                 // s 用于判定找到姓名与否
    while(p2 != set2.end()) {
        if(sname == *p2) {                          // 如果找到, 就输出姓名
            cout<<*p2<<endl;
            s = true;
        }
        p2++;
    }
    if(!s)
        cout<<sname<<"不在集合中!"<<endl;         // 如果没有找到, 就给出提示
}
}
```

程序运行结果如下:

```
30 12 7 6 4 0 -2
```

```
杜明 杜明 黄明浩 李大海 李则 林浦海 刘一 王为 王小二 张清山 张清山 张三
```

```
输入要查找的姓名: 杜明
```

```
杜明
```

```
杜明
```

本程序定义了两个集合 `set1` 和 `set2`。`set1` 是 `set` 类型的从大到小排序的集合，因 `set` 类型是不允许元素重复的，程序输出结果表明确实没有重复元素。`set2` 是 `multiset` 类型的 `string` 集合，没有指定排序方式，默认为从小到大排序。程序还具有姓名查询功能，从键盘输入一个姓名后，将从集合 `set2` 中查找姓名，并列出的名字。

【例 7-28】 设计包括多名学生的集合对比 `set` 和 `multiset` 的区别，每位学生有姓名和年龄，输出各位学生的姓名和年龄，以及所有学生的平均年龄。

设计思路：设计 `Student` 结构存取和输出学生姓名和年龄信息。由于集合（`set/multiset`）中的元素是有序的，因此必须定义 `Student` 对象大小比较的方法，可以通过重载运算符函数 `Student::operator<()` 来实现。因为 `set` 模板的默认比较函数 `operator<()` 是 `const` 函数，所以 `Student` 的重载运算符函数也必须设置为 `const` 函数。

```
// Eg7-28.cpp
#include<iostream>
#include<set>
#include<string>
using namespace std;

struct Student {
    string name;
    int age;
    bool operator<(const Student& o) const { return age < o.age; }           // L1
    void outData()const { cout<<name<<" , \t"<<age<<"\t\t"; }
};

int main() {
    set<Student> s1 = {{"张三",21}, {"李四",18}, {"黄明",27}};                // L2
    multiset<Student> s2 = {{"张三",21}, {"李四",18}, {"黄明",27}};          // L3
    s1.insert(Student({"张三", 21}));                                         // L4
    s2.insert(Student({"张三", 21}));                                         // L5
    for(auto iter = s1.begin(); iter != s1.end(); iter++) {
        iter->outData();
    }
    cout<<endl;
    for (auto iter = s2.begin(); iter != s2.end(); iter++) {
        iter->outData();
    }
}
```

程序运行结果如下：

```
李四, 18      张三, 21      黄明, 27
李四, 18      张三, 21      张三, 21      黄明, 27
```

语句 L1 定义了类 `Student` 的小于比较运算符函数，在将 `Student` 对象插入 `set/multiset` 集合时，系统会自动调用它，以实现集合排序必须进行的大小比较。对比语句 L2、L3 集合中对

象的初始化顺序和程序运行结果的输出顺序可知，集合中的元素是按照小于比较运算符函数（语句 L1）定义的 `age` 从小到大排了序的。本程序如果没有定义 L1 语句处的小于比较运算符函数，将无法通过编译。

2. map 和 multimap

`map` 和 `multimap` 提供了操作<键, 值>的方法（其中的值也称为映射值），存储一对对象，即键对象和值对象。键对象是用于查找过程中的键，值是与键对应的附加数据。例如，若键是单词，对应的值是表示该单词在文档中出现次数的数字，这样的 `map` 就成了统计单词在文档中出现次数的频数表；再如，键是单词，值是单词出现的页号链表（同一单词可在不同页多次出现），用 `multimap` 实现这样的键值对象就可以构造单词索引表。`map` 中的元素不允许重复，而 `multimap` 中的元素是可以重复的，如图 7-9 所示。

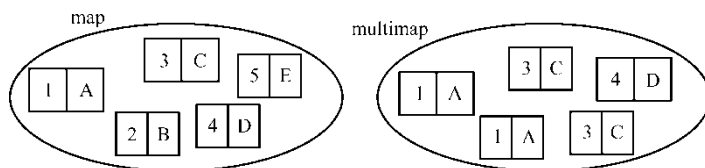


图 7-9 map 和 multimap 映射示意

虽然 `set/multiset` 集合中的元素只包括键，而 `map` 映射中的元素则由<键, 值>对构成，但 `map/multimap` 提供的操作也是针对各元素中的键进行的。同理，如果将 `struct` 或 `class` 等自定义对象作为<键, 值>对中的键存储在 `map` 或 `multimap` 中，也必须定义该结构或类的比较运算符重载函数，或者定义能够比较对象大小的普通函数。

前面介绍的 `set/multiset` 集合的操作方法同样适用于 `map/multimap`，包括集合的建立方法、成员函数、比较运算、排序规则和方法等，只需将其中的 `set` 更改为 `map`，将 `multiset` 更改为 `multimap`，所以在此不再介绍。但是，需要对 `map/multimap` 集合的成员函数 `insert()` 及元素的访问方法作两点补充说明。

1) 关于成员函数 `insert()`

形式上，`map/multimap` 和 `set/multiset` 的成员函数 `insert()` 都具有如下相同的形式：

```
insert(e)
```

```
// 将元素 e 插入 map、multimap、set、multiset
```

虽然语法形式相同，但是插入 `map/multimap` 和 `set/multiset` 的元素是有区别的。插入 `set/multiset` 的元素是单独的键，而插入 `map/multimap` 的元素是<键, 值>构成的一对数据，这对数据是一个不可分割的整体，需要用函数 `make_pair()` 构造：

```
make_pair(k, v)
```

其中，`k` 代表键，`v` 代表值，`make_pair` 将用<`k`, `v`>构造映射的元素。

2) 关于 `map/multimap` 元素访问

`map/multimap` 的元素是由<键, 值>构成的，同一个键可以对应多个不同的值，并且通过相关映射的迭代器访问它们。`map/multimap` 类型的迭代器提供了两个数据成员：一个是 `first`，用于访问键；另一个是 `second`，用于访问值。

此外，`map` 类型的映射可以用键作为数组下标，访问该键所对应的值，但 `multimap` 类型的映射不允许用数组下标的方式访问其中的元素。

【例 7-29】 用 multimap 构造汉英对照字典。

```
// Eg7-29.cpp
#include<iostream>
#include<string>
#include<map>
using namespace std;

void main() {
    multimap<string, string> dict; // dict 是用于存放字典的 multimap
    multimap<string, string>::iterator p; // 定义访问字典的迭代器
    string eng[] = {"cliff", "berg", "precipice", "tract"};
    string che[] = {"悬崖", "冰山", "悬崖", "一片, 区域"};
    for(int i = 0; i < 4; i++)
        dict.insert(make_pair(eng[i], che[i])); // 批量插入字典元素
    // 插入单个元素
    dict.insert(make_pair(string("tract"), string("地带")));
    dict.insert(make_pair(string("precipice"), string("危险的处境")));
    dict.insert(make_pair("day", "一天")); // L1, 正确
    // dict["precipice"] = "悬崖, 峭壁"; // L2, 错误
    for(p = dict.begin(); p != dict.end(); p++) // 输出字典内容
        cout<<p->first<<"\t" <<p->second<<endl; // first 是键, second 是值
    string word;
    cout<<"请输入要查找的英文单词: ";
    cin>>word;
    for(p = dict.begin(); p != dict.end(); p++) // 遍历字典, 查找单词
        if(p->first == word)
            cout<<p->second<<endl; // 输出单词的中文解释
    cout<<"请输入要查找的中文单词: ";
    cin>>word;
    for(p = dict.begin(); p != dict.end(); p++) // 遍历字典, 查找汉词
        if(p->second == word)
            cout<<p->first<<endl; // 输出汉词对应的英语单词
}
```

程序运行结果如下:

```
berg        冰山
cliff        悬崖
precipice    悬崖
precipice    危险的处境
tract        一片, 区域
tract        地带
请输入要查找的英文单词: precipice
悬崖
危险的处境
请输入要查找的中文单词: 悬崖
cliff
precipice
```

说明:

① 程序中的 make_pair 用于建立<键, 值>, 这样的键-值对是 map/multimap 映射中的元素

结构。

② 语句 L1 在支持 C++ 11 标准的编译环境（如 VC++ 2015 以上版本）中是正确的，而在早期的 VC 编译环境中是错误的。错误的原因在于"day"、"一天"是 char* 类型的串，make_pair 以此建立的将是<char*, char*>类型的键/值对，但 dict 需要的是<string, string>类型的键-值对。在 C++ 11 标准中，可以将 const char* 类型的字符串传递给 string 类型的变量，系统进行自动转换，但之前并不支持。

③ map 映射支持数组下标操作，但 multimap 映射不支持下标操作，这是语句 L2 错误的原因。

【例 7-30】 用 map 查询学生的专业，学生包括学号和姓名。

设计思路：设计结构或类 Student 存储学生信息，设计 map 映射学生和他的专业，用迭代器遍历 map 来查找学生的专业。

此外，由于 Student 对象作为 map 的<键,值>中的键，因此还需要设计比较 Student 对象大小的函数，这里通过普通函数重载 operator<()实现。

```
// Eg7-30.cpp
#include<iostream>
#include<map>
#include<string>
using namespace std;

class Student {
    string name;
    int age;
public:
    Student(string Name, int Age):name(Name), age(Age) {}
    void outData()const{ std::cout<<name<<","<<age<<"\n"; }
    string getName()const { return name; }
    int getAge()const { return age; }
};

bool operator<(Student a, Student b) { return a.getAge()<b.getAge(); } // L1

int main() {
    map<Student, string> m = {{Student("张三", 21), "信管专业"},
                             {Student("黄明", 27), "会计专业" } }; // L2
    m[Student("王五", 22)] = "会计学"; // L3
    m.insert(make_pair(Student("张三", 24), "经济学")); // L4
    for (auto iter = m.begin(); iter != m.end(); iter++) {
        (iter->first).outData(); // L5
        cout<<iter->second<<endl; // L6
    }
    string name;
    bool find = false;
    cout<<endl<<"输入想查询的学生姓名: ";
    cin>>name;
    for (auto iter = m.begin(); iter != m.end(); iter++) {
        if ((iter->first).getName() == name) { // L7
            (iter->first).outData();
            cout<<iter->second<<endl; // L8
        }
    }
}
```



```

        find = true;
    }
}
if(!find)
    cout<<"没有找到该学生!"<< endl;
return 0;
}

```

程序运行结果如下：

```

张三,   21   信管专业
王五,   22   会计学
张三,   24   经济学
黄明,   27   会计专业

```

输入学生查询姓名：张三

```

张三,   21   信管专业
张三,   24   经济学

```

语句 L1 定义了对 `Student` 对象进行小于比较的函数 `operator<()`，在向 `map` 中插入 `Student` 对象时，系统会自动调用该函数按年龄对学生对象排序，对比输出数据和语句 L2、L3、L4 中的 `Student` 对象顺序，可以发现，输出数据是按学生的 `age` 排序的。

语句 L2、L3、L4 分别展示了 `map` 的列表初始化方法、数组下标访问方法，以及用 `make_pair` 构造 `map` 元素然后用 `insert()` 插入的方法。

语句 L5、L6、L7 展示了如何在 `map` 容器中通过迭代器访问 `class` 对象成员的方法。

`multimap` 与 `map` 的用法基本相同，区别在于，`map` 映射中的键不允许重复，而 `multimap` 中的键是允许重复的。此外，`map` 允许用数组的下标访问映射中的值，而 `multimap` 是不允许的，`multimap` 在构造一键对多值的查询时非常有用。

7.5.6 算法

算法（algorithm）是用模板技术实现的适用于各种容器的通用程序，常常通过迭代器间接地操作容器元素，并且返回迭代器作为算法运算的结果。`STL` 提供了 100 多个算法，每个算法都是一个函数模板或者一组函数模板，能够对许多类型的容器进行操作，各种容器可能包含不同类型的数据元素。`STL` 的算法覆盖了在容器上实施的各种常见操作，如遍历、排序、检索、插入及删除元素等。许多算法不但适用于系统提供的容器类型，而且适用于普通的 C++ 数组或自定义类型的容器。下面介绍 `STL` 库中的几个常用算法。

1. `find` 和 `count` 算法

`find` 算法从一个容器中查找指定的值，`search` 算法则是从一个容器查找由另一个容器所指定的序列值，`count` 算法用于统计某个值在指定数据区间中出现的次数。其用法如下：

```

find(beg, end, value)
search(beg1, end1, beg2, end2)
count(beg, end, value)

```

在 `find()` 中，`[beg, end]` 是指定的区间，常用迭代器位置描述该区间，`value` 是要查找或统计的值。如果在 `[beg, end]` 区间中找到 `value` 值，`find()` 就返回区间中值为 `value` 的第一个元素

位置，否则返回最后元素位置。

`search()`在`[beg1, end1]`区间内查找有无与`[beg2, end2]`相同的子区间，如果找到，就返回`[beg1, end1]`中第一个相同元素的位置，否则返回 `end1`。`count()`统计 `value` 在区间`[beg, end]`中出现的次数。

【例 7-31】 find 和 count 算法举例。

```
// Eg7-31.cpp
#include<iostream>
#include<list>
#include<algorithm>
using namespace std;

void main() {
    int arr[] = {100, 200, 300, 400, 500, 500, 600, 700, 800, 900, 1000};
    int *ptr;
    ptr = find(arr, arr+9, 400);                // 查找 400 在 arr 数组中的地址
    cout<<"400 在数组中的下标是: "<<ptr-arr<<endl;    // find 返回地址，计算出数组元素位置
    list<int> L1;                                // 定义链表 L1
    int a1[] = {30,40,50,60,60,60,80};
    for(int i = 0; i < 7; i++)
        L1.push_back(a1[i]);                    // 将 a1 数组加入 L1 链表
    list<int>::iterator pos;
    pos = find(L1.begin(), L1.end(), 80);        // 在 L1 中查找 80，结果放于 pos 中
    if(pos != L1.end())
        cout<<"L1 链表中存在数据元素: "<<*pos;    // 输出找到的数据
    cout<<"，它是链表中的第: "<<distance(L1.begin(), pos)+1
        <<distance(L1.begin(), pos)+1           // distance 计算迭代器与链首元素间隔的元素个数
        <<"个节点! "<<endl;
    int n1 = count(arr, arr+10, 500);           // 统计 arr 数组中 500 的个数
    int n2 = count(L1.begin(), L1.end(),60);    // 统计 L1 链表中 60 的个数
    cout<<"arr 数组中有: "<<n1<<"个"<<500<<endl;
    cout<<"L1 链表中有: "<<n2<<"个"<<60<<endl;
}
```

程序运行结果如下：

```
400 在数组中的下标是: 3
L1 链表中存在数据元素: 80，它是链表中的第: 7 个节点!
arr 数组中有: 2 个 500
L1 链表中有: 3 个 60
```

2. Merge 算法

Merge 算法能够对两个容器进行合并，并将结果存放在第三个容器中，其用法如下：

```
merge(beg1, end1, beg2, end2, dest)
```

`merge()`将`[beg1, end1]`与`[beg2, end2]`区间合并，把结果存放在容器 `dest` 中。如果参与合并的两个容器中的元素是有序的，那么合并的结果也是有序的。

说明：`list` 链表也提供了一个成员函数 `merge()`，能够把两个 `list` 类型的链表合并在一起。同样，如果合并前的链表是有序的，那么合并后的链表仍然有序。

【例 7-32】 merge 算法与 list 的成员函数 merge()的应用。

```
// Eg7-32.cpp
#include<iostream>
#include<list>
#include<algorithm>
using namespace std;

void main() {
    int a1[] = {10, 20, 30, 40, 50, 60, 70};
    int a2[] = {40, 50, 60};
    int a[10];
    merge(a1, a1+7, a2, a2+3, a);           // 将 a1、a2 合并，结果放在 a 数组中
    cout<<"a[]:\t";
    for(int i = 0; i < 10; i++)
        cout<<a[i]<<"\t";
    cout<<endl;
    list<int> L1, L2;
    list<int>::iterator pos;                // pos 迭代器用于输出链表元素
    for(int i = 0; i < 7; i++)
        L1.push_back(a1[i]);               // 插入 L1 的链表元素
    for(int j = 0; j < 3; j++)
        L2.push_back(a2[j]);               // 插入 L2 的链表元素
    L1.merge(L2);                           // 用 list 的 merge 成员合并 L1、L2
    cout<<"L1:\t";
    for(pos = L1.begin(); pos != L1.end(); pos++) // 用迭代器 pos 输出合并后的 L1
        cout<<*pos<<"\t";
    cout<<endl;
}
```

程序运行结果如下：

```
a[]: 10  20  30  40  40  50  50  60  60  70
L1: 10  20  30  40  40  50  50  60  60  70
```

3. sort 算法

sort 算法可以对指定容器区间内的元素进行排序，默认排序方式是升序，其用法如下：

```
sort(beg, end [, cmp])
```

其中，[beg, end]是要排序的区间，cmp 是设置排序方式的大小比较函数，省略时默认为小于比较，即按从小到大的顺序对该区间的元素进行排序。

【例 7-33】 利用 sort 算法对数组和向量进行排序。

```
// Eg7-33.cpp
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
void main() {
    int a1[] = {10, -20, 30, 4, 50, 13, 7};
    int a2[] = {-2, 0, 30, 12, 6, 7, -9, 56, 32, 78};
```

```

    sort(a1, a1+7); // 排序 a1 数组
    cout<<"a1[]: ";
    for(int i = 0; i < 7; i++)
        cout<<a1[i]<<"\t";
    cout<<endl;
    vector<int> L1;
    vector<int>::iterator pos;
    for(int i = 0; i < 10; i++)
        L1.push_back(a2[i]); // 将 a2 数组插入 L1 链表
    sort(L1.begin(), L1.end()); // 排序 L1
    cout<<"L1: ";
    for(pos = L1.begin(); pos != L1.end(); pos++)
        cout<<*pos<<"\t"; // 输出 L1 链表中的值
    cout<<endl;
    sort(L1.begin(), L1.end(), greater<int>()); // 降序排序 L1
    cout<<"Desc L1: ";
    for(pos = L1.begin(); pos != L1.end(); pos++)
        cout<<*pos<<"\t"; // 输出 L1 链表中的值
    cout<<endl;
}

```

程序运行结果如下:

a1[]:	-20	4	7	10	13	30	50			
Asc L1:	-9	-2	0	6	7	12	30	32	56	78
Desc L1:	78	56	32	30	12	7	6	0	-2	-9

7.5.7 STL 容器和算法处理自定义类的常见问题

初学者在应用 STL 库处理自定义类对象时,有时可能会遇到系列冗长的编译错误信息。遇到这样的错误时不要慌张,多数时候就那么一两个问题,但编译器报出了上百个错误。可以先排除程序本身的语法错误,再根据具体的错误信息检查是否是模板参数不匹配,或者某项内容与 `const` 参数不符合,以及未定义运算符函数 `operator<()`或 `operator>()`等。

最常见的错误是应用 STL 中的 `set`、`map`、`sort`、`min`、`max` 等容器和算法处理自定义类对象时,没有重载小于或大于运算符函数,无法进行自定义类对象排序所产生的错误。原因是 `set` 和 `map` 容器是有序的(包括 `multiset` 和 `multimap` 等),在将元素插入这类容器时就要进行对象的大小比较,如果类对象没有提供大小比较函数就无法排序,程序就会产生错误。

同理,如果自定义类没有定义大小比较方法,在用 `sort`、`min`、`max` 等算法处理这样的类对象时也会产生错误。解决的方法是为类提供比较大小的函数,最简单的方法是重载小于比较运算符函数,形式如下:

```

class A {
    .....
    // set、map 要求存入其中的对象的比较函数为 const 函数
    bool operator<(const A& o) const {...}
    .....
};

```

【例 7-34】 定义简单的复数类,建立该复数的数组和 `set` 集合,并用 STL 中的 `sort` 算法

对数组进行排序输出。

```
// Eg7-34.cpp
#include<iostream>
#include<set>
#include<algorithm>
using namespace std;

class Complex {
private:
    double i, r;
public:
    Complex(double ir, double rr):i(ir), r(rr) {}
    void set_r(double pr) { r = pr; }
    void set_i(double pi) { i = pi; }
    const double& real()const { return r; }
    double& image() { return i; }
    // bool operator<(const Complex &o)const { return r < o.r; } // L1
    void outdata() {
        if (i >= 0)
            cout<<r<<"+"<<i<<"i"<<endl;
        else
            cout<<r<<i<<"i"<<endl;
    }
};

int main() {
    Complex c1[] = {{1.0,1.0}, {6.0,2.0}, {3.0,3.0}};
    cout<<"-----sort array output-----"<<endl;
    sort(c1, c1+2);
    for(Complex c:c1)
        c.outdata();
    cout<<"-----sort set output-----"<<endl;
    set<Complex>cset;
    for(Complex c:c1)
        cset.insert(c);
    Complex c2{4.0, 4.0}, c3{1.0, 1.0};
    cset.insert(c2);
    cset.insert(c3);
    for(Complex c:cset)
        c.outdata();
}
```

这个程序将产生编译错误，因为 `set()`、`sort()` 无法对 `Complex` 的对象进行大小比较。取消语句 L1 的注释，为 `Complex` 类重载运算符函数 `operator<()` 后，程序运行结果如下：

```
-----sort array output-----
1+1i
2+6i
3+3i
-----sort set output-----
1+1i
```

2+6i
3+3i
4+4i

7.6 编程实作：模板和STL编程应用

【例 7-35】 STL 的容器和算法不但适用于内置数据类型，而且适用于用户自定义的数据类型。继续 6.6 节的编程实例二，下面用 STL 容器继续完成 comFinal、Account 和 Chemistry 课程结构的程序设计。

STL 的各种容器，如 vector、list、stack、deque、set/multiset、map/multimap 等，都可以用来存取类 comFinal、Account 及 Chemistry 的对象。这里用 list 容器存取各类的对象。

1. 编写主程序

启动 Visual C++ 2022，打开 C:\course 中的 com_main.sln 工程文件，将其中主函数 com_main.cpp 的源代码修改如下：

```
// comList.cpp
#include<iostream>
#include<list>
#include"Chemistry.h"
#include"Account.h"
using namespace std;

void main() {
    list<comFinal*> comList;           // 定义基类 comFinal 对象的指针链表
    list<comFinal*>::iterator pos;
    comFinal com1("阿曼", 76, 87, 90);
    Account a1("张三星", 98, 90, 97, 90, 90);
    Chemistry c1("光红顺", 89, 80, 80, 80, 80);
    comList.push_back(&com1);         // 将基类 comFinal 对象的指针加入链表
    comList.push_back(&a1);           // 将派生类 Account 的对象指针加入链表
    comList.push_back(&c1);           // 将派生类 Chemistry 的对象指针加入链表
    for(pos = comList.begin(); pos != comList.end(); pos++)
        (*pos)->show();              // 遍历链表，输出各对象的数据成员
}
```

函数 main() 定义了基类 comFinal 的指针链表 comList 和迭代器 pos，三条 comList.push_back 语句分别将指向基类 comFinal 的对象 com1 的指针、派生类 Account 对象 a1 的指针、派生类 Chemistry 对象 c1 的指针添加到 comList 链表中，构成了如图 7-10 所示的 comList 链表。

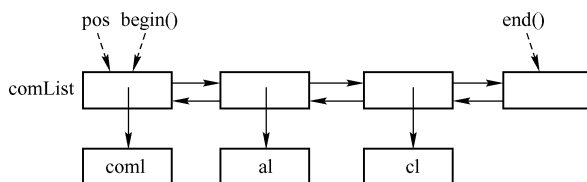


图 7-10 comList 链表的结构

由于 pos 迭代器本身是指向 comList 链表节点的指针, 而 comList 链表节点是指向对象的指针, 因此通过 pos 迭代器访问对象成员函数 show() 的语法如下:

```
(*pos)->show();
```

2. 验证程序运行结果

由于 comList 链表是用 STL 中的 list 容器建立的基于基类 comFinal 的链表, 而 pos 是基于该链表的迭代器, 实际是指向基类 comFinal 的指针, 通过它能够访问 comList 中的全部对象 (包括基类 comFinal 以及派生类 Account、Chemistry 的对象), 因此下面的循环将访问到 comList 链表中的所有元素对象。

```
for(pos = comList.begin(); pos != comList.end(); pos++)
    (*pos)->show();
```

由于 show() 是定义于基类 comFinal 中的虚函数, 且派生类 Account、Chemistry 提供了各自的 show() 函数实现版本, 因此这个访问具有多态性, 能够调用到迭代器 pos 所指向的对象成员的 show() 函数。编译运行 comList 程序, 将得到如图 7-11 所示的结果。

```
学生姓名:阿曼
英语成绩:76
语文成绩:87
数学成绩:90
基础课总分:253
基础课平均成绩:84

学生姓名:张三星
英语成绩:98
语文成绩:90
数学成绩:97
基础课总分:285
基础课平均成绩:95

会计学成绩:90
经济学成绩:90
总分:465
学生姓名:光红顺
英语成绩:89
语文成绩:80
数学成绩:80
基础课总分:249
基础课平均成绩:83

有机化学:80
化学分析:80
总分:409
请按任意键继续. . .
```

图 7-11 程序运行结果

习 题 7

7.1 解释下列概念:

模板 函数模板 模板函数 类模板 模板类

7.2 什么是模板的类型参数与非类型参数? 有什么区别?

7.3 举例说明 C++ 在匹配模板参数的过程中可能遇到的问题, 有哪些解决方法。

7.4 简述类模板的实例化过程, 说明函数模板、特化模板和普通函数都存在时的调用次序。

7.5 读程序, 写结果。

(1)

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

template<typename T>
T mymin(T t) { return t; }
template<typename T1, typename ... T2>
double mymin(T1 p, T2 ... arg) {
    double ret = mymin(arg ...);
    if (p < ret)
        return p;
    else
        return ret;
}

void main() {
    cout<<mymin(100, 12, 30, 4, 20)<<"\t";
    cout<<mymin('a', 'z', 2)<<"\t";
    cout<<mymin(2, 7.8)<<endl;
```



```
}
```

(2)

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

template <typename T>
inline T const& mymin(T const& a, T const& b) {
    cout<<"1:\t";
    return a < b ? a : b;
}

template<>
const char* const& mymin(const char* const& a, const char* const& b) {
    cout<<"2:\t";
    return strcmp(a, b) < 0 ? a : b;
}

inline char const* mymin(char const* a, char const* b) {
    cout<<"3:\t";
    return std::strcmp(a, b) < 0 ? a : b;
}

int main() {
    int a = 5, b = 12;
    string s1 = "aString1", s2 = "aString2";
    cout<<mymin(a, b)<<endl;
    cout<<mymin(s1, s2)<<endl;
    cout<<mymin("How are you!", "Hello Template!")<<endl;
}
```

(3)

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

template<typename T>
typename common_type<T>::type Max(T t) {
    return t;
}

template<typename T, typename ... P>
typename common_type<T, P ...>::type Max(T t, P ... p) {
    if(t > Max(p ...))
        return t;
    else
        return Max(p ...);
}

int main() {
    auto t1= Max(-4, 50.2f, 9.9, -2.6L);
    auto t2 = Max(74, 50.2f, 9.9, -2.6L);
    std::cout<<"t1 = "<<t1<<std::endl;
    std::cout<<"t2 = "<<t2<<std::endl;
}
```

(4) 分析下面程序的功能。

```
#include <tuple>
#include<string>
#include <vector>
#include<iostream>
using namespace std;

typedef tuple<string, string, int, string> Student;
vector<Student> inputData() {
    Student stu;
    vector<Student> sv;
    for(int i = 0; i < 3; i++) {
        cout<<"输入学生数据, 包括: 姓名、学号、年龄、专业"<<endl;
        cin>>get<0>(stu)>>get<1>(stu)>>get<2>(stu)>>get<3>(stu);
        sv.push_back(stu);
    }
    return sv;
}

void display(vector<Student> sv) {
    for(int j = 0; j < 3; j++) {
        cout<<get<0>(sv[j])<<"\t"<<get<1>(sv[j])<<"\t"<<get<2>(sv[j])<<"\t"<<get<3>(sv[j])<<endl;
    }
}

void main() {
    vector<Student> s;
    s = inputData();
    display(s);
}
```

(5)

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

void main() {
    string name[] = {"张大年", "刘明海", "李煜"};           // 雇员姓名
    double salary[] = {1200, 2000, 1450};                  // 雇员工资
    map<string, double> sal;                                // 用映射存储姓名和工资
    map<string, double>::iterator p;                        // 定义映射的迭代器
    for(int i = 0; i < 3; i++)
        sal.insert(make_pair(name[i], salary[i]));         // 将姓名、工资加入映射
    sal["tom"] = 3400;                                       // 通过下标运算加入 map 元素
    sal["bob"] = 2000;
    for(p = sal.begin(); p != sal.end(); p++)               // 输出映射中的全部元素
        cout<<p->first<<"\t"<<p->second<<endl;              // 输出元素的键和值
    string person;
    cout<<"输入查找人员的姓名: ";
    cin>>person;
    for(p = sal.begin(); p != sal.end(); p++)
```

```
        if(p->first == person)
            cout<<p->second<<endl;
    }
```

注：在程序提示“输入查找人员姓名”时输入：tom。

7.6 设计一个函数模板，实现两数的交换，并用 int、char 等类型的数据进行测试。

7.7 设计一个函数模板，从 int、char、double、char* 等类型的数组找出最大值元素。提示：可用类型参数传递数组、用非类型参数传递数组大小，为了找出 char* 类型数组中的最大值元素，需要对该类型进行函数重载，即以普通函数的方式重载对 char* 类型数组求取最大值元素的函数。

7.8 用模板设计一个通用的单向链表类 List，实现链表节点的增加、删除、查找以及链表数据的输出操作。

7.9 假设有一个类 Worker，形式如下：

```
class Worker {
    char name[10];
    int age;
    double salary;
public:
    Worker(...);
    void SetData(char *Name, int Age, double wage);
    void Display();
    .....
}
```

其中，name 表示姓名、age 表示年龄、salary 表示薪酬；构造函数 Worker() 实现各数据成员的初始化，成员函数 SetData() 用于重置各数据成员的值，Display() 用于显示输出各数据成员的值。

完成该类的设计，并用 STL 中的链表 list（或向量 vector、堆栈 stack、队列 queue 等数据结构）管理该类的对象，要求：至少建立两个链表，每个链表中至少存入 Worker 类的 3 个对象，通过迭代器访问输出各节点对象的数据成员，并利用链表的 merge 算法将两链表合并在一起，然后输出合并后的链表节点对象。

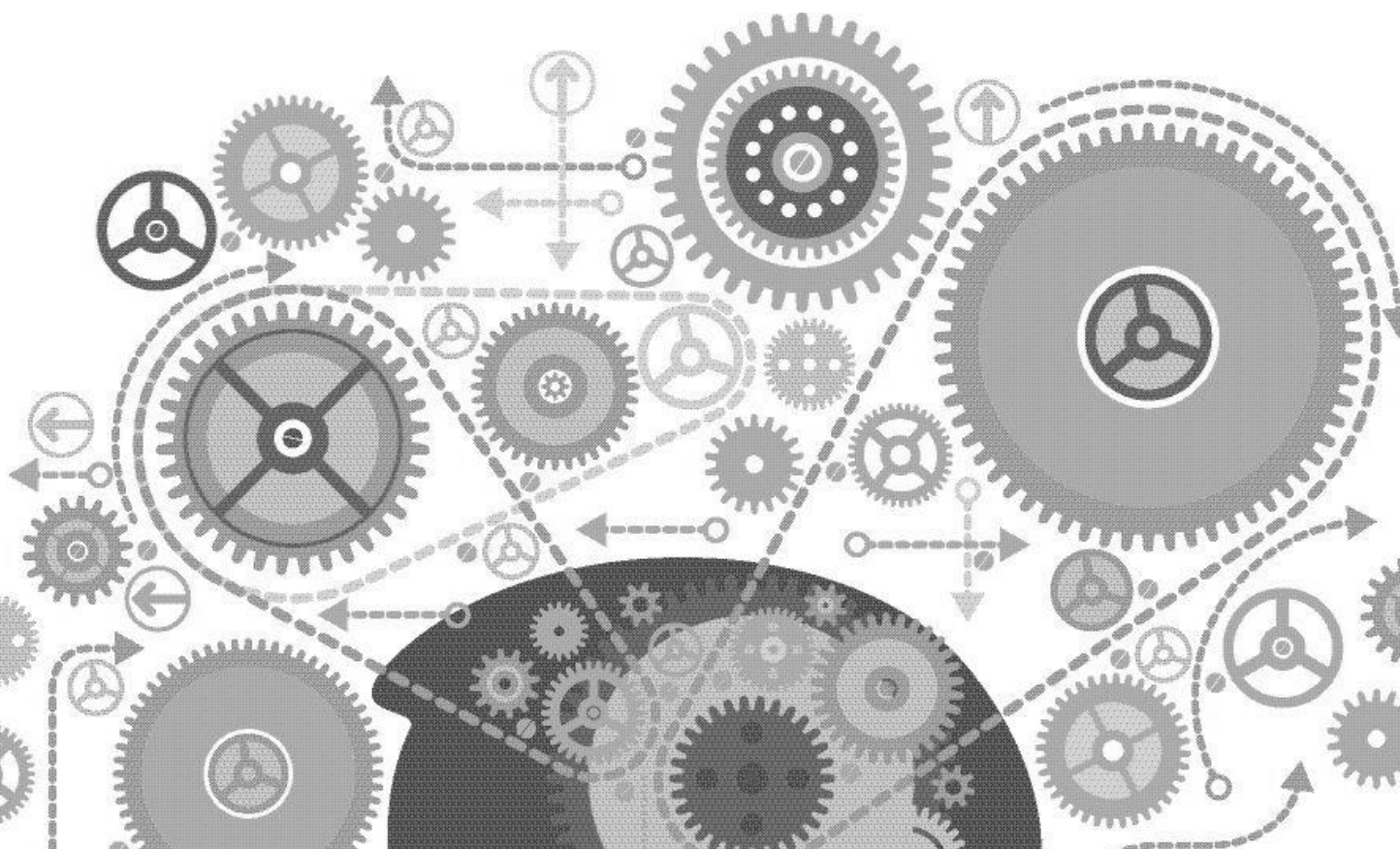
7.10 建立两个 int 类型的向量 vector，利用 merge 算法将其合并，再用 sort 算法对合并后的向量排序。

第 8 章

异 常

C++语言的异常处理机制能够将异常检测与异常处理分离，一部分程序负责检测问题，另一部分程序负责问题的处理，提高了程序的清晰性和可读性，有助于程序员编写出清晰、健壮、容错能力更强的程序，适用于大型软件开发。

本章介绍 C++语言的异常处理机制，包括异常处理结构、异常函数、异常类及其继承结构等。



8.1 异常处理概述

异常是指程序运行期间发生的不正常情况，如 `new` 无法获得所需内存、数组下标越界、运算溢出、除数为 0、无效的函数参数、打开不存在的文件，等等。异常处理就是指对程序执行过程中发生的异常进行适当处理，避免程序出现丢失数据或破坏系统运行等灾难性的后果。

在软件开发中要彻底避免异常是不可能的。软件开发人员必须充分考虑程序运行过程中可能出现的各种异常情况，以保证应用程序的正确性和更强的容错能力。传统程序处理异常的典型方法是不断测试程序的执行情况，并对测试结果进行处理。如以下伪代码所示：

```
执行任务 1
    if 任务 1 未能被正确执行
        执行错误处理代码
执行任务 2
    if 任务 2 未能正确执行
        执行错误处理代码
执行任务 3
.....
```

上述伪代码表示的程序逻辑是：程序每执行一个任务后，就检查执行是否成功，如果不成功，就执行错误处理代码，如果成功，就执行下一任务……如此反复，直到所有任务执行完毕，或者因某个错误而终止应用程序。

在这种程序设计方法中，错误处理代码分布在整个程序的各个部分，凡是代码可能出错的地方都要进行错误处理。其优点是程序的执行过程和错误处理情况非常清晰；缺点是随着程序复杂性的增加，过多的错误处理代码使程序本身的代码受到“污染”，变得晦涩难懂，加深了代码理解和维护的困难。

此外，如果设计的函数或类是提供给其他程序员重用的，虽然设计人员可以检测到异常条件的存在，但他无法确定其他应用该函数或类的程序员会如何处理这些异常；另一方面，如果应用这些函数或类的程序员想要按照自己的意愿处理异常，但是他无法检测到异常条件是否存在，因为他无法看到或修改程序员设计的异常检测代码。

C++的异常处理机制较好地解决了上述两个问题。其基本思想是将异常发生和异常处理分别放在不同的函数中，产生异常的函数不需要具备处理异常的能力。当一个函数出现异常时，可以抛出一个异常，然后由该函数的调用者捕获并处理这个异常，如果调用者不能处理，可以将该异常抛给其上一级的调用者处理。

大概而言，利用 C++的异常处理机制能够完成以下事情：

① 改善程序的可读性和可维护性，将异常处理代码与主干程序代码分离，适合团队开发大型项目。

② 提供了有力的异常检测和可能的异常恢复手段，以统一方式处理异常。

③ 在异常引起系统错误之前处理异常。

④ 处理由库函数或第三方提供的函数引起的异常。

⑤ 在出现无法处理的异常时执行清理工作，并以适当的方式退出程序。

8.2 C++异常处理基础

8.2.1 异常处理的结构

C++引入了三个用于异常处理的关键字 **try**、**throw**、**catch**。**try** 用于监测可能发生的异常，**throw** 用于抛出异常，**catch** 用于捕获并处理由 **throw** 抛出的异常。

try-throw-catch 构造了 C++异常处理的基本结构，形式如下：

```
try {  
    .....  
    if err1  
        throw xx1  
    .....  
    if err2  
        throw xx2  
    .....  
    if errn  
        throw xxn  
}  
catch(type1 arg) {...}           // 异常类型 1 错误处理  
catch(type2 arg) {...}           // 异常类型 2 错误处理  
catch(typem arg) {...}           // 异常类型 m 错误处理  
.....                           // 其他语句
```

其中，**catch** 后的 “()” 中只能有单个类型或单个对象声明，称为异常声明；**type1**、**type2**、**typem** 是数据类型关键字，可以是系统内置的数据类型（如 **char**、**double** 等），也可以是自定义的数据类型，如类或结构。在 **catch** 的参数表中，可以只有类型名而没有形参，如果不需要捕获由 **throw** 语句抛出的异常值，就可以不提供形参名称。

在设计具有异常处理能力的程序时，必须将要检测其错误的程序代码放在 **try** 块中，且对那些可能出现异常的语句进行测试，并根据测试结果决定是否抛出（**throw**）异常。

throw 语句用于抛出异常，用法如下：

```
throw exception;
```

其中，**exception** 就是异常，可以是任何数据类型的表达式，包括类对象。如果是类对象，就要求相应的类具有析构函数和复制构造函数（或移动构造函数），**throw** 将利用 **exception** 生成一个临时的异常对象，然后将其抛出，该异常对象能够被 **catch** 捕获并处理。

catch 必须紧跟在 **try** 块后，用于捕获由 **throw** 抛出的异常。同一 **try** 块可以抛出多个不同类型的异常，每个 **catch** 块只能处理一种类型的异常。因此，当一个 **try** 块抛出了多种不同类型的异常时，就应该有多个 **catch** 异常处理块与之对应。

try-throw-catch 异常处理的执行逻辑如下：当程序执行过程中遇到 **try** 块时，将进入 **try** 块并按正常的程序逻辑顺序执行其中的语句；如果 **try** 块的所有语句都被正常执行，没有发生任何异常，那么 **try** 块中不会有异常被 **throw**。在这种情况下，程序将忽略所有的 **catch** 块，顺序执行那些不属于任何 **catch** 块的程序语句，并按正常逻辑完成程序的执行，就像 **catch** 块不存在一样。

如果在执行 **try** 块的过程中，某条语句产生错误并用 **throw** 抛出了异常，那么程序控制流

程将从该 `throw` 子句转移到 `catch` 块, `try` 块中自该 `throw` 语句后的所有语句都不会再被执行了。C++ 将按 `catch` 块出现的次序, 用异常的数据类型与每个 `catch` 参数表中指定的数据类型相比较, 如果两者类型相同, 就执行该 `catch` 块, 同时把异常的值传递给 `catch` 块中的形参 `arg` (如果该块有 `arg` 形参)。只要有一个 `catch` 块捕获了异常, 其余 `catch` 块都将被忽略。如果没有任何 `catch` 能够匹配该异常, C++ 将调用系统默认异常处理程序处理该异常, 其通常做法是直接终止该程序的运行。

【例 8-1】异常处理的简单例程。

```
// Eg8-1.cpp
#include<iostream>
using namespace std;

void main() {
    cout<<"1--befroe try block ... "<<endl;
    try {
        cout<<"2--Inside try block ... "<<endl;
        throw 10;
        cout<<"3--After throw ... "<<endl;
    }
    catch(int i) {
        cout<<"4--In catch block1 ... errcode is ... "<<i<<endl;
    }
    catch(char * s) {
        cout<<"5--In catch block2 ... errcode is ... "<<s<<endl;
    }
    cout<<"6--After Catch ... ";
}
```

程序运行结果如下:

```
1--befroe try block ...
2--Inside try block ...
4--In catch block1 ... errcode is ... 10
6--After Catch ...
```

这个结果表明 `try` 块之前的语句被正常执行 (输出 “1--……”), `try` 块中第一次执行 `throw` 之前的语句被正常执行 (输出 “2--……”), `try` 中 `throw` 之后的语句不被执行 (没有输出 “3--……”)。当有异常抛出时, 捕获了异常的 `catch` 块将被执行 (输出 “4--……”), 其他 `catch` 块将被略过 (没有输出 “5--……”), `catch` 块后 (不属于 `catch` 块) 的语句也会被执行 (输出 “6--……”)。

8.2.2 异常捕获

异常捕获由 `catch` 完成, `catch` 必须紧跟在与之对应的 `try` 块后, 目的是捕获并处理该 `try` 块抛出的某种异常。如果异常被某个 `catch` 捕获, 程序将执行该 `catch` 块中的代码, 之后将继续执行 `catch` 块后的语句; 如果异常不能被任何 `catch` 块捕获, 它将被传递给系统的异常处理模块, 程序将被系统异常处理模块终止。

`catch` 根据异常声明的数据类型捕获异常, 异常声明是指紧接在 `catch` 后 “()” 中的参数表,

与函数的形参表相似,但只允许有一个参数。如果 `catch` 参数表中异常声明的数据类型与 `throw` 抛出的异常的数据类型相同,该 `catch` 块将捕获异常,程序流程将进入该 `catch` 块执行。当进入一个 `catch` 块后,将用 `throw` 抛出的异常对象初始化异常声明中的参数。与函数的参数传递相似,如果 `catch` 的参数是非引用类型,采用值传递方式将异常对象复制给 `catch` 参数,如果在 `catch` 块内对参数进行修改,修改的是复制到的副本而不是异常对象本身。如果参数是引用类型,该参数传递的就是异常对象的一个别名,修改参数也就是修改异常对象本身。

注意: `catch` 在进行异常数据类型的匹配时,除如下 3 种情况的类型转换,不会进行其他数据类型的默认转换,只有与异常的数据类型精确匹配的 `catch` 块才会被执行。① 允许非常量向常量的类型转换,即 `throw` 语句抛出的非常量对象可以匹配一个接受常量对象的 `catch` 语句;② 允许派生类向基类的类型转换;③ 数组被转换成指向数组元素类型的指针,函数被转换成指向该函数类型的指针。

例 8-2 是对例 8-1 的简化,但该例的 `catch` 块不会被调用。

【例 8-2】 `catch` 捕获异常时,不会进行数据类型的默认转换。

```
// Eg8-2.cpp
#include<iostream>
using namespace std;

void main() {
    cout<<"1--Before try block ..."<<endl;

    try {
        cout<<"2--Inside try block ..."<<endl;
        throw 10;
        cout<<"3--After throw ..."<<endl;
    }
    catch(double i) {                                // 仅此与例 8-1 不同
        cout<<"4--In catch block1 ... errcode is ... "<<i<<endl;
    }

    cout<<"5--After Catch ...";
}
```

程序运行结果如下:

```
1--Before try block ...
2--Inside try block ...
abnormal program termination                                // 程序因异常而结束
```

其中的 `abnormal program termination` 是由系统异常处理模块给出的,该信息可能会因编译器不同而存在差异。

此结果表明,程序执行了 `try` 块中的“`throw 10;`”语句后就被中止了,并没有执行 `catch` 块及 `catch` 块后的语句。其原因是,“`throw 10;`”抛出的是一个 `int` 类型的异常,而 `catch(double i)` 只能捕获 `double` 类型的异常。虽然 `int` 可以转换成 `double` 类型的数据,但 `catch` 并不会进行这样的转换,导致程序中没有适当的 `catch` 块能够处理 `try` 块中抛出的异常。因此,该异常最后只能由系统的异常处理模块处理,系统异常处理模块中止了该程序的执行。

8.3 异常和函数

1. 在函数中处理异常

异常处理可以局部化为一个函数，就是将处理异常的 try-throw-catch 结构置于函数中，每次进行该函数的调用时，异常将被重置。

【例 8-3】 temperature 是一个检测温度异常的函数，当温度达到冰点或沸点时产生异常。

```
// Eg8-3.cpp
#include<iostream>
using namespace std;

void temperature(int t) {
    try{
        if(t == 100)
            throw "沸点! ";
        else if(t == 0)
            throw "冰点! ";
        else
            cout<<"the temperature is OK ..."<<endl;
    }
    catch(int x) { cout<<"temperature = "<<x<<endl; }
    catch(const char *s) { cout<<s<<endl; }
}

void main() {
    temperature(0);           // L1
    temperature(10);          // L2
    temperature(100);         // L3
}
```

程序的运行结果如下：

```
冰点!
the temperature is OK ...
沸点!
```

temperature()是一个具有异常处理能力的函数，当调用参数为 0 或 100 时，将产生字符串类型的异常，catch 将捕获该异常并进行处理。

注意：在函数内部进行异常处理时，针对 try 块抛出的所有异常都应该提供对应的 catch 块对之进行处理；若在调用函数时发生了不能够处理的异常，程序将会被终止。

例如在例 8-3 中，若将 catch(const char *s)修改为 catch(double s)，则调用语句 L1 时会产生“abnormal program termination”而被终止。因为 temperature(0)将产生 const char*类型的异常，由语句“throw “冰点! ”;”产生，但函数没有能够捕获 const char*类型异常的 catch 块，最后会被系统异常处理模块终止。

2. 在函数调用中完成异常处理

在前面的例子中，异常的检测和处理只能由函数设计人员完成，函数的调用者并不能对相关异常进行任何处理。因为异常的检测和处理都是在同一个函数中完成的，函数调用者无

法进行异常的检测。C++的异常处理机制允许将异常发生与异常处理分开，即将产生异常的代码放在一个函数中，将检测处理异常的函数代码放在另一个函数中，这种方式能够让异常处理更具灵活性和实用性，编写出容错性更强的程序。

【例 8-4】 修改例 8-3，将异常处理从函数中独立出来，由调用函数完成。

```
// Eg8-4.cpp
#include<iostream>
using namespace std;

void temperature(int t) {
    if(t == 100)
        throw "沸点! ";
    else if(t == 0)
        throw "冰点! ";
    else
        cout<<"the temperature is "<<t<<endl;
}

void main() {
    try {
        temperature(10);
        temperature(50);
        temperature(100);
    }
    catch(const char *s){ cout<<s<<endl; }
}
```

程序运行结果如下：

```
the temperature is 10
the temperature is 50
沸点!
```

在本例中，函数 `temperature()` 只抛出了异常，把异常的检测和处理留给了函数的调用者，调用者可以根据函数的实际调用情况进行异常的检测和处理，使异常处理更具合理性。比如，当温度降到 0℃ 时，启动加热器；当温度升到 100℃ 时，关闭加热器。

8.4 异常处理的特殊情况

1. `noexcept` 异常声明 ^{C++11}

如果确定某函数能够正常运行，不会产生任何异常，可以用 `noexcept` 声明它不会产生异常，形式如下：

```
rtype f(...) noexcept {                // 不会抛出异常
    .....
}

rtype g(...) {                          // 可能抛出异常
    .....
}
```

`noexcept` 声明是 C++ 11 新标准提出的，与早期版本的空 `throw` 语句等价，形式如下：

```
rtype f(...) throw() {           // 不会抛出异常
    .....
}
```

如果一个声明了 `noexcept` 的函数，又在其函数体类使用 `throw` 抛出了异常，多数编译器不会在编译时对 `noexcept` 进行检测，因此程序能够通过编译。但是，在执行这样的函数时，一些 C++ 编译器会调用 `terminate` 终止该程序的执行。

关于 `noexcept`，补充说明以下两点：① `noexcept` 说明符实际上可以接收一个 `bool` 类型实参，形式如下：

```
rtype f(...) noexcept(e){ ... }
```

其中，`e` 是一个逻辑表达式，若其结果为 `true`，表示函数 `f()` 不会抛出异常；若结果是 `false`，则表明 `f()` 可能抛出异常。例如：

```
void f(int x) noexcept(true) { ... }           // f()函数不抛出异常
void g(int x) noexcept(false) { ... }          // g()函数可能抛出异常
```

② `noexcept` 除了是一个说明函数是否会抛出异常的声明符，也是一个可以用来判断函数是否会产生异常的运算符。这为程序设计带来了方便，因为事先知道函数不会产生异常可以简化对该函数的调用编码(至少可以不用考虑 `try-catch` 的调用方式)，在调用前可以用 `noexcept` 判断函数是否会抛出异常以达到这一目的，形式为：

```
noexcept(e);
```

`e` 是一个表达式，可以包括多个函数调用。例如，对于上面的函数 `f()` 和 `g()`，有

```
noexcept(f(4)+g(6));
```

函数 `f()` 本身做了不抛出异常的声明，但是函数 `g()` 可以抛出异常，因此应该用如下形式调用此表达式才是正确的：

```
try { f(4)+g(6); }
catch(...) { ... }
```

2. 捕获所有异常

在多数情况下，`catch` 都只用于捕获某种特定类型的异常，但它也具有捕获全部异常的能力。其形式如下：

```
catch(...) {
    .....
}
```

`catch` 参数表中的省略号可以匹配任何异常类型。

【例 8-5】 改写前面的函数 `Errhandler()`，使之能够捕获所有异常。

```
// Eg8-5.cpp
#include<iostream>
using namespace std;

void Errhandler(int n) throw() {
    try {
        if(n == 1)
            throw n;
```

```

        if(n == 2)
            throw "dx";
        if(n == 3)
            throw 1.1;
    }
    catch(...) { cout<<"catch an exception ..."<<endl; }
}

void main() {
    Errhandler(1);
    Errhandler(2);
    Errhandler(3);
}

```

程序运行结果如下：

```

    catch an exception ...
    catch an exception ...
    catch an exception ...

```

Errhandler()抛出了 int、char 和 double 三种类型的异常，但只有一个 catch 块，程序执行结果表明该 catch 块捕获了所有的异常。

当一个 try 块后有多块 catch 块与之相匹配时，如果其中有能捕获全部异常的 catch 块，应该将它放在最后，作为没有被前面捕获的异常的默认处理方案，如果将它放在前面，那么其后的所有 catch 块都将毫无意义，因为无论什么异常都被它捕获了。

3. 再次抛出异常

如果 catch 块无法处理捕获的异常，或者只能处理异常的一部分，其余异常需要由它的外层调用函数继续处理，就可以将异常再次抛出，形式如下：

```
throw;
```

这样的空 throw 语句只能出现在 catch 块中，或者 catch 块内的调用函数中。虽然 throw 后面没有抛出的异常对象表达式，但它实际上是将当前 catch 块捕获的异常对象再次抛出。从一个 catch 块中再次抛出的异常不会再被同一个 catch 块所捕获，它将被传递给外部的 catch 块处理。

catch 块再次抛出异常时，如果需要，可以修改异常对象的值，表示它已作了处理。要达到这一目的，就需要将 catch 的异常声明为引用类型，其修改才会有效。

【例 8-6】 在异常处理块中再次抛出同一异常。

```

// Eg8-6.cpp
#include<iostream>
using namespace std;

void Errhandler(int n) {
    try{
        if(n == 1)
            throw n;
        if(n == 4)
            throw 'a';
        cout<<"all is OK ..."<<endl;
    }
}

```

```

    catch (int n) {
        n = 100;
        cout<<"exception inside is:\t"<<n<<endl;
        throw; // 再次抛出 catch 捕获的异常
    }
    catch(char &c) {
        c = 'B';
        cout<<"exception inside is:\t"<<c<<endl;
        throw;
    }
}

void main() {
    try{ Errhandler(1); }
    catch(int x) { cout<<"exception in main ...\t"<<x<<endl; }
    try{ Errhandler(4); }
    catch(char x) { cout<<"exception in main ...\t"<<x<<endl; }
    cout<<"... End ..."<<endl;
}

```

程序运行结果如下：

```

exception inside is: 100
exception in main... 1
exception inside is: B
exception in main ... B
... End ...

```

内层异常处理结构将它所捕获的异常再次抛出，它抛出的异常被位于 `main()` 中的外层 `catch` 块捕获并处理。从输出结果前两行可以看出，虽然 `catch(int n)` 块内对异常对象 `n` 进行了修改，但它再次抛出的异常对象值却是修改之前的值；而 `catch(char &c)` 块内对异常对象的修改是有效的。由此可知，如果希望 `catch` 块对它再次抛出的异常对象进行修改，应该使用引用参数捕获异常对象。

注意：只能从 `catch` 块中而不是 `try` 块中再次抛出异常，这种方式有利于构成对同一异常的多层处理机制，增强了异常处理的能力，因为一个内层不能处理的异常，外层是有可能处理的。

4. 异常的嵌套调用

在出现异常的情况下，`try` 块用于指示编译器到哪里查找 `catch` 块，没有紧跟在 `try` 后面的 `catch` 是没有用途的，即 `try` 和 `catch` 块之间不应该有其他语句。`try` 块可以嵌套，即一个 `try` 块中可以包括另一个 `try` 块，这种嵌套可能形成一个异常处理的调用链。

【例 8-7】 嵌套异常处理示例。

```

// Eg8-7.cpp
#include<iostream>
using namespace std;

void fc() {
    try { throw "help ..."; }
    catch(int x) { cout<<"in fc..int handler"<<endl; }
}

```

```

        try { cout<<"no error handle ..."<<endl; }
        catch(const char *px) { cout<<"in fc..char* handler"<<endl; }
    }
    void fb() {
        int *q = new int[10];
        try {
            fc();
            cout<<"return form fc()"<<endl;
        }
        catch(...) {
            delete []q;
            throw;
        }
    }
    void fa() {
        char *p = new char[10];
        try{
            fb();
            cout<<"return from fb()"<<endl;
        }
        catch(...) {
            delete []p;
            throw;
        }
    }
    void main() {
        try {
            fa();
            cout<<"return from fa"<<endl;
        }
        catch(...) { cout<<"in main"<<endl; }
        cout<<"End"<<endl;
    }
}

```

程序运行结果如下：

```

    in main
    End

```

为什么会是这样的结果呢？在程序执行过程中，调用了哪些处理异常的 `catch` 块呢？

图 8-1 是本例异常处理过程的示意图，图中实线箭头是函数及异常的调用过程，虚线是函数调用及异常处理的返回过程。

在函数调用 `fa()`→`fb()`→`fc()`的过程中，当调用到 `fc()`时，`fc()`中第一个 `try` 块中的 `throw "help ..."`抛出了一个字符串类型的异常，与该 `try` 相对应的 `catch` 块将被用来捕获该异常，即图 8-1 中的⑤。但是该 `catch` 块只能捕获 `int` 类型的异常，而“`throw "help ..."`”抛出的是字符串类型的异常，所以 `fc()`中的第 1 个 `catch` 块不能处理该异常。这时，该异常将被返回到调用 `fc()` 的上一级 `try` 块对应的 `catch` 中被处理，如图 8-1 中的⑥所示。

注意：尽管 `fc()`的第 2 个 `catch` 块能够捕获 `const char*`类型的异常，但异常是由 `fc()`的第 1 个 `try` 块抛出的，与它没有关系（第 2 个 `catch` 块属于第 2 个 `try` 块），所以根本不会调用它。

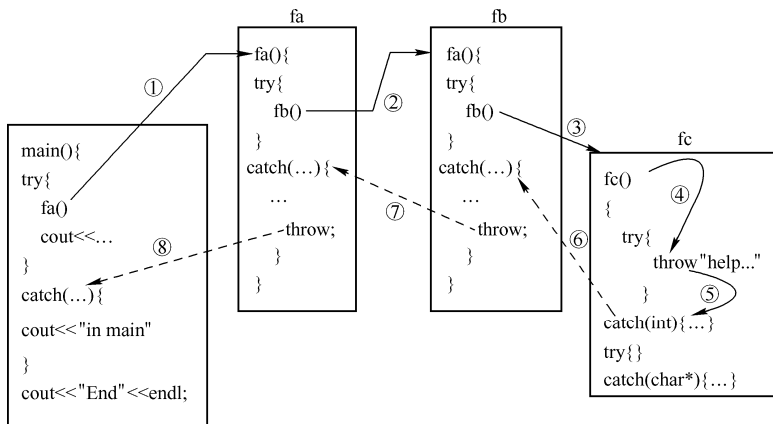


图 8-1 异常动态调用过程

当异常沿图中⑥所示的虚线返回到 fb() 的 catch 块中后，由于 fb() 的 catch 块能够捕获任何异常，所以捕获并处理了从 fc() 传递来的异常。在完成对指针 q 的清理后，fb() 的 catch 块再次抛出了该异常。该异常将返回到上级 fa() 的 catch 模块，如图 8-1 中的⑦所示。

fa() 的 catch 捕获了该异常，在完成对指针 p 的清理后，再次抛出该异常。fa() 的上层异常处理模块是 main() 中的 catch 块，它将捕获该异常，如图 8-1 中的⑧所示。

main() 中的 catch 块将输出字符串 "in main"。最后，程序将执行 main() 中 catch 后的语句，所以输出了字符串 "End"。

例 8-7 展示了异常处理的一般逻辑，从上面的分析可以看出，这样的异常处理方法比较完善，它将每个函数分配的动态存储空间都回收了。

8.5 异常和类

8.5.1 构造函数和异常

函数（包括普通函数和类成员函数）可以通过返回值将其执行状态返回给调用者，调用者可以借此判断函数运行是否正确。但是，如果在构造函数中发生了错误，外部函数如何知道该对象有没有被正确构造呢？因为构造函数没有返回值，无法通过返回值将对象构造失败的原因告知外部函数。传统程序方法可能采用如下策略处理发生在构造函数中的异常：① 返回一个处于错误状态的对象，外部函数可以检查该对象状态，以便判定该对象是否被成功构造；② 设置一个全局变量保存对象构造的状态，外部函数可以通过该变量值判断对象构造的情况；③ 在构造函数中不做对象的初始化工作，而是专门设计一个成员函数负责对象的初始化。

现在，利用异常处理机制能够很好地处理构造函数中的异常问题，当构造函数出现错误时就抛出异常，外部函数可以在构造函数之外捕获并处理该异常。

对构造函数的异常处理体现了 C++ 异常处理机制的真正能力，能够自动调用异常发生前已构造的所有局部对象的析构函数。如果正被构造的对象还有对象成员，且在发生异常之前这些对象成员已经被构造了，那么 C++ 异常机制将调用这些对象成员的析构函数；如果该对象还有对象成员数组，且异常发生时对象数组已被部分构造，那么已被构造了的数组对象的析构函数也将被调用。

【例 8-8】类 B 有一个类 A 的对象成员数组 obj，类 B 的构造函数进行了自由存储空间的过量申请，最后造成内存资源耗尽，产生异常，则异常将调用对象成员数组 obj 的析构函数，回收 obj 占用的存储空间。

```
// Eg8-8.cpp
#include<iostream>
using namespace std;

class A {
    int a;
public:
    A(int i = 0):a(i) { }
    ~A(){ cout<<"in A destructor ..."<<endl; }
};

class B{
    A obj[3];
    double *pb[10];
public:
    B(int k) {
        cout<<"int B constructor ..."<<endl;
        for(int i = 0; i < 10; i++) {
            pb[i] = new double[20000000];
            if(pb[i] == 0)
                throw i;
            else
                cout<<"Allocated 20000000 doubles in pb["<<i<<"]"<<endl;
        }
    }
};

void main() {
    try{ B b(2) ; }
    catch(int e){ cout<<"catch an exception when allocated pb["<<e<<"]"<<endl; }
}
```

程序运行结果如下：^[1]

```
int B constructor ...
Allocated 20000000 doubles in pb[0]
Allocated 20000000 doubles in pb[1]
Allocated 20000000 doubles in pb[2]
Allocated 20000000 doubles in pb[3]
in A destructor ...
in A destructor ...
in A destructor ...
catch an exception when allocated pb[4]
```

由运行结果可知，当程序执行进入类 B 的构造函数时，循环正常运行了 4 次，pb[0]~pb[3] 都分配到了相应的数组空间。当进行第 5 次循环为 pb[4]分配数组空间时，内存资源已不能满

[1] 本测试是早期在 VC 6.0 且内存有限的计算机环境下测试的，在内存较充足的计算机环境下未必能够重现相同的测试结果。

足申请要求，所以产生异常。产生异常时，C++异常处理机制将调用异常发生前已构造的 obj 对象成员数组中每个对象的析构函数，这就是运行结果中有 3 行 “in A destructor...” 的由来。

8.5.2 异常类

1. 简单的异常类

异常可以是任何类型，包括自定义类。用来传递异常信息的类都可以称为**异常类**。异常类可以非常简单，甚至没有任何成员；也可以与普通类一样复杂，有自己的成员函数、数据成员、构造函数、析构函数、虚函数等，还可以通过继承方式构成异常类的继承层次结构。

在实际程序设计过程中，许多异常都是类而不是内置数据类型。使用异常类的优点是可以通过它创建传递错误信息的对象，异常处理程序可以利用这个对象获取错误信息，以便进行有针对性的异常处理。

【例 8-9】 设计一个堆栈，当入栈元素超出了堆栈容量时，就抛出一个栈满的异常；如果栈已空，还要从栈中弹出元素，就抛出一个栈空的异常。

```
// Eg8-9.cpp
#include <iostream>
using namespace std;

const int MAX = 3;
class Full { }; // L1 堆栈满时抛出的异常类
class Empty { }; // L2 堆栈空时抛出的异常类
class Stack {
private:
    int s[MAX];
    int top;
public:
    void push(int a);
    int pop();
    Stack(){ top = -1; }
};
void Stack::push(int a) {
    if(top >= MAX-1)
        throw Full(); // L3
    s[++top] = a;
}
int Stack::pop() {
    if(top < 0)
        throw Empty(); // L4
    return s[top--];
}

void main() {
    Stack s;
    try {
        s.push(10);
        s.push(20);
        s.push(30);
```

```

        // s.push(40);                                // L5, 将产生栈满异常
        cout<<"stack(0) = "<<s.pop()<<endl;
        cout<<"stack(1) = "<<s.pop()<<endl;
        cout<<"stack(2) = "<<s.pop()<<endl;
        cout<<"stack(3) = "<<s.pop()<<endl;            // L6
    }
    catch(Full) { cout<<"Exception: Stack Full"<<endl; } // L7
    catch(Empty) { cout<<"Exception: Stack Empty"<<endl; } // L8
}

```

程序运行结果如下：

```

stack(0) = 30
stack(1) = 20
stack(2) = 10
Exception: Stack Empty

```

语句 L1、L2 分别定义了两个简单的异常类 **Full**、**Empty**。如果堆栈已满还要继续加入元素，将抛出一个 **Full** 异常，在语句 L3 处；如果在栈已空的情况下，继续从栈中弹出元素，将抛出一个 **Empty** 异常，在语句 L4 处。

语句 L5 被注释了，否则将引发栈满异常。栈中总共只有 3 个元素，语句 L6 试图从空栈中继续弹出元素，将引起栈空异常，抛出类 **Empty** 的一个对象，该对象被语句 L8 捕获，从程序运行结果的最后一行可以看到这个结论。

如果异常类只是为某个单独的类提供异常处理，就可以在应用它的类中进行定义，形成嵌套类。例如，若例 8-9 的 **Full**、**Empty** 是只用于类 **Stack** 的异常类，它也可以定义如下。其中标注省略号的地方与例 8-9 中的代码完全相同。

```

class Stack {
    .....
public:
    class Full { };                // 异常类
    class Empty { };              // 异常类
    .....
};
void main() {
    .....
    catch(Stack::Full) { ... }
    catch(Stack::Empty) { ... }
}

```

main()函数中的 **catch** 捕获异常时要加限定符 **Stack::**，因异常类 **Full** 和 **Empty** 是类 **Stack** 的成员类。

2. 异常对象

由异常类建立的对象称为异常对象，异常的处理过程实际上是异常对象在 **throw** 和 **catch** 之间的传递过程，需要调用复制构造函数和析构函数。在没有显式定义它们的情况下，C++ 调用由编译器为异常类自动生成的默认构造函数、默认复制构造函数和析构函数完成对象的建立与传递过程，如图 8-2 所示。该过程可以概括为：① 当 **try** 块中的 **throw** 抛出异常表达式时，将调用异常类的适当构造函数创建异常类的一个临时对象 **t**，**throw** 将抛出 **t**；② 与 **try**

相匹配的 **catch** 块将调用复制构造函数用 **t** 初始化生成适用于 **catch** 块的临时对象 **x**，传递完成后调用 **t** 的析构函数销毁对象 **t**；③ 进入该 **catch** 块进行异常处理。

异常类并非总像前面的 **Full** 和 **Empty** 那样简单，也可以有数据成员和成员函数，成为复杂的类。在实际编程中，常通过异常类的成员传递异常信息，以便进行适当的异常处理。

【例 8-10】 修改例 8-9 的异常类 **Full**，修改后的 **Full** 具有构造函数和成员函数，还有一个数据成员。利用这些成员，可以获取异常发生时没有入栈的元素信息。

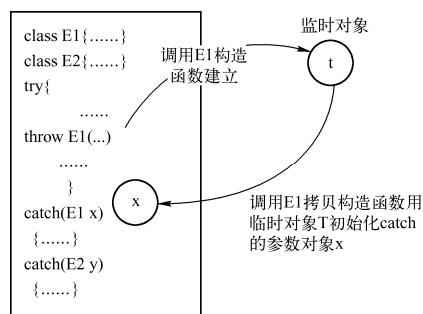


图 8-2 异常对象的处理过程

```

// Eg8-10.cpp
#include <iostream>
using namespace std;

const int MAX = 3;
class Full {
    int a;
public:
    Full(int i):a(i) { }
    int getValue() { return a; }
};

class Empty { };

class Stack {
private:
    int s[MAX];
    int top;
public:
    Stack(){ top = -1; }
    void push(int a) {
        if(top >= MAX-1)
            throw Full(a);
        s[++top] = a;
    }
    int pop() {
        if(top < 0)
            throw Empty();
        return s[top--];
    }
};

void main() {
    Stack s;
    try {
        s.push(10);
        s.push(20);
        s.push(30);
        s.push(40);
    }
}

```

// L1

// L2

```

catch(Full e) {                                     // L3
    cout<<"Exception: Stack Full ..."<<endl;
    cout<<"The value not push in stack: "<<e.getValue()<<endl; // L4
}
}

```

程序运行结果如下：

```

Exception: Stack Full ...
The value not push in stack: 40

```

由于 s 栈只能存放 3 个元素，因此语句 L2 将产生 s 栈满的异常，致使语句 L1 的 throw 抛出异常。语句 L1 的 throw Full(a) 将调用构造函数 Full::Full(int i) 建立一个临时对象，其中 i 的值为 40，throw 语句随后会抛出该临时对象。语句 L3 的 catch 将捕获该对象，并调用 Full 的默认复制构造函数用该临时对象初始化异常对象 e，e 的数据成员 a 因此就获得了来源于 throw 语句抛出的临时对象的数据成员 a 的值 40，所以语句 L4 的 e.getValue() 获得的值为 40。

3. 捕获异常对象的引用

同函数参数的传递可以传值和传引用一样，catch 块捕获异常的参数传递也有按值传递和按引用两种方式。如果 catch 块的异常声明是一个值参数，C++ 将以按值传递的方式把 throw 语句抛出的异常对象传递给 catch 参数表中的参数对象。前面例子中的 catch 块都是按值传递异常对象的，这种方式将调用异常类的复制构造函数用 throw 抛出的临时对象初始化 catch 声明的异常对象。

如果 catch 块的异常声明是一个引用参数，C++ 将把该引用参数绑定到由 throw 抛出的临时异常对象上，即引用参数是临时对象的别名，这种情况不会调用异常类的复制构造函数来初始化 catch 参数声明中的异常对象。当异常对象具有较多数据成员时，传引用方式能避免大量的数据复制，提高程序效率。

例如，将例 8-10 语句 L3 中的 catch 改为如下形式，其余程序代码不做任何修改，就将该程序的 catch 修改成了传引用的方式。

```

catch(Full &e) { ... }
.....

```

修改后的程序将得到与例 8-10 完全相同的结果，但二者对异常对象的处理方法完全不同。传引用不会调用 Full 类的复制构造函数来初始化对象 e，而是直接将 e 绑定到了 throw 语句抛出的临时对象上，e 即该临时对象的别名。

8.5.3 派生异常类的处理

在设计软件的异常处理系统时，可以将各种异常汇集起来，根据异常的性质，将其分属到不同的类中，形成异常类的继承体系结构。在处理异常时，可以根据程序的实际情况捕获那些可能发生的相关异常，并做出正确的处理。利用多态，还可以将异常类设计为具有多态特性的继承结构，利用多态的强大功能处理异常。例如，一个远程登录程序的可能异常类层次结构如图 8-3 所示。其中，基类 BasicException 代表基本异常，包括文件系统异常 (FileSysException)、操作系统异常 (OsException)、安全异常 (SecurityException) 三类；文件系统异常又包括文件没有找到 (FileNotFound)、访问的磁盘不存在 (DiskNotFound)；安全异

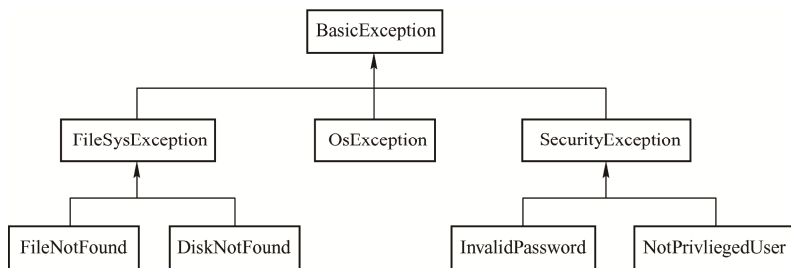


图 8-3 远程访问系统的异常类层次结构

常又包括口令错误（InvalidPassword）、未授权用户（NotPrivilegedUser）。

图 8-3 只列出了远程登录系统的部分异常，事实上还有大量其他类型的异常，如执行的文件不存在、用户标志错误、修改只读文件等。

【例 8-11】 设计图 8-3 所示异常继承结构中从 BasicException 到 FileSysException 部分的异常类。

```

// Eg8-11.cpp
#include<iostream>
using namespace std;

class BasicException {
public:
    const char* Where() { return "BasicException ..."; }
};

class FileSysException:public BasicException{
public:
    const char *Where() { return "FileSysException ..."; }
};

class FileNotFound:public FileSysException{
public:
    const char *Where() { return "FileNotFound ..."; }
};

class DiskNotFound:public FileSysException {
public:
    const char *Where() { return "DiskNotFound ..."; }
};

void main() {
    try{
        // .....
        throw FileSysException();
    }
    catch(DiskNotFound p) { cout<<p.Where()<<endl; }
    catch(FileNotFound p) { cout<<p.Where()<<endl; }
    catch(FileSysException p) { cout<<p.Where()<<endl; }
    catch(BasicException p) { cout<<p.Where()<<endl; }
    try {
        // .....
        throw DiskNotFound();
    }
    catch(BasicException p) { cout<<p.Where()<<endl; }
}
  
```

// 程序代码

// 程序代码


```

        catch(FileSysException p) { cout<<p.Where()<<endl; }
        catch(DiskNotFound p) { cout<<p.Where()<<endl; }
        catch(FileNotFound p) { cout<<p.Where()<<endl; }
    }

```

程序运行结果如下。

```

    FileSysException ...
    BasicException ...

```

运行结果第 2 行是错误的。结合程序的第 2 个 try 块可知，它应该是“DiskNotFound ...”才正确。产生该错误的原因是第 2 个 try 块后的 4 个 catch 块的排列次序有问题。在处理异常派生类时，要特别注意各 catch 块捕获异常的次序。捕获基类异常的 catch 块应该放在最后，捕获派生类异常的 catch 块应该放在前面。因为能够捕获基类对象的 catch 块也能捕获派生类对象，若将它放在最前面，它将提前捕获派生类对象。

异常类继承结构也可以用多态实现，多态可以简化异常的捕获。

【例 8-12】 设计图 8-3 所示异常继承体系中从 BasicException 到 FileSysException 部分的多态异常类。

多态实现程序如下，其中省略的代码与例 8-11 完全相同。

```

// Eg8-12.cpp
#include <iostream>
using namespace std;

class BasicException {
public:
    virtual char* Where(){ return "BasicException ..."; }
};
.....
void main() {
    try {
        ..... // 程序代码
        throw FileSysException();
    }
    catch(BasicException &p) { cout<<p.Where()<<endl; }
    try {
        ..... // 程序代码
        throw DiskNotFound();
    }
    catch(BasicException &p) { cout<<p.Where()<<endl; }
}

```

该程序将得到如下运行结果，结合程序代码可以知道该结果是正确的。

```

    FileSysException ...
    DiskNotFound ...

```

在用多态实现的异常继承体系结构中，catch 块的异常处理过程非常简单，一条具有多态异常捕获能力的语句：

```

    catch(BasicException &p) { cout<<p.Where()<<endl; }

```

等效于如下 4 条异常捕获语句：

```

catch(BasicException p) { cout<<p.Where()<<endl; }
catch(FileSysException p) { cout<<p.Where()<<endl; }
catch(DiskNotFound p) { cout<<p.Where()<<endl; }
catch(FileNotFound p) { cout<<p.Where()<<endl; }

```

实际上，只要是位于图 8-3 继承层次结构中的所有异常类，`catch(BasicException &p)`都能捕获，并能调用到异常对象正确的 `Where()`成员函数。

习 题 8

8.1 什么是异常？C++为什么要引入异常处理机制？

8.2 简述 `try-throw-catch` 异常处理的过程。

8.3 什么是异常类？

8.4 阅读下面的程序，写出程序运行结果。

(1)

```

#include <iostream>
using namespace std;

void main() {
    int a[] = {8, 5, 5, 0, 6, 0, 8, 5, 5, 0, 7, 8};
    for(int i = 0; i < 5; i++) {
        try {
            cout<<"in for loop ..."<<i<<"\t";
            if(a[i+1] == 0)
                throw 1;
            cout<<a[i]<<"/"<<a[i+1]<<" = "<<a[i]/a[i+1]<<endl;
        }
        catch(int) { cout<<"end"<<endl; }
    }
}

```

(2)

```

#include <iostream>
using namespace std;

void err(int t) {
    try {
        if(t > 100)
            throw "bigger than 100";
        else if(t < -100)
            throw t;
        else
            cout<<"t in right range ..."<<endl;
    }
    catch(int x) { cout<<"error---"<<x<<endl; }
    catch(const char *s) { cout<<"error---"<<s<<endl; }
    catch(float f) { cout<<"error---"<<f<<endl; }
}

```

```

void main() {
    err(200);
    err(99);
    err(-1210);
}

```

(3)

```

#include <iostream>
using namespace std;

class excep {
private:
    const char* ch;
public:
    excep(const char* m = "exception class ...") { ch = m; }
    void print() { cerr<<ch<<endl; }
};

void err1() {
    cout<<"enter err1\n";
    throw excep("exception");
}

void err2() {
    try {
        cout<<"enter err2\n";
        err1();
    }
    catch (int) {
        cerr<<"err2:catch\n";
        throw;
    }
}

void err3() {
    try {
        cout<<"enter err3\n";
        err2();
    }
    catch (...) {
        cerr<<"err3:catch\n";
        throw;
    }
}

void main() {
    try { err3(); }
    catch (...) { cerr<<"main:catch\n"; }
}

```

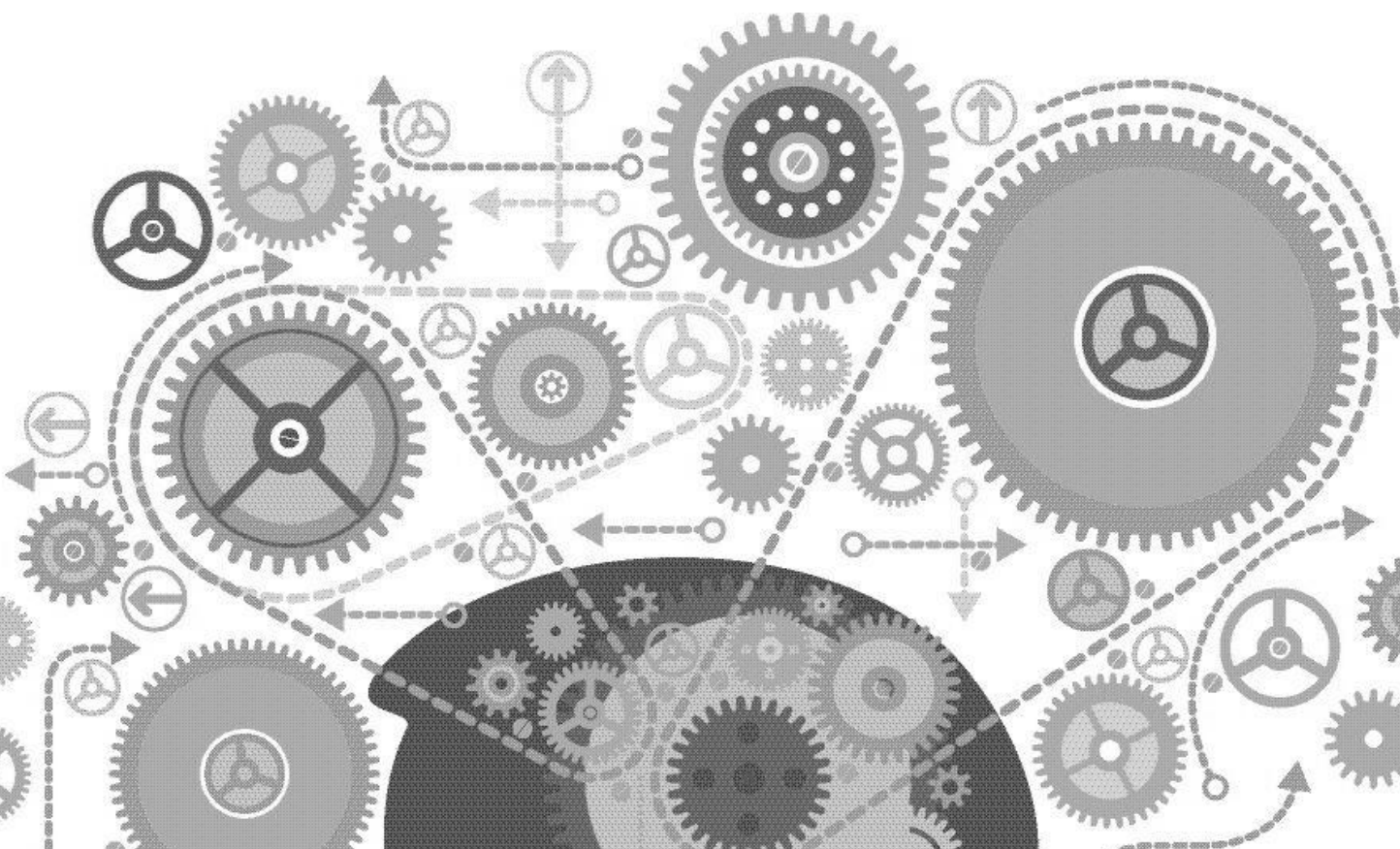
8.5 设计一个只能容纳 10 个元素的队列，如果入队元素超出了队列容量，就抛出一个队列已满的异常；如果队列已空还要从中取出元素，就抛出一个队列已空的异常。

第 9 章

线 程

C++ 11 标准引入了线程，以便为多任务并发的高性能程序设计提供支持。

本章简要介绍线程的基本概念、运行原理和 C++ 线程的设计方法，主要内容包括：进程和线程的关系，简单的 C++ 线程、多线程和线程同步等内容。



9.1 程序、进程和线程

程序（program）是用程序设计语言编写的用来完成特定任务的一组命令集合，以文件形式保存在各种存储设备中，是静态的。**进程**（process）是被装载到内存中处于运行状态的程序，是动态的，要经历从外存储入内存，在内存中运行，运行完成后从内存中清除的完整过程，这个过程称为进程的生命周期。

线程（thread）是进程中执行运算的最小单位，是从一个进程中划分出来的更小指令集合，该指令集合能够被 CPU 作为一个独立单元进行调度和执行。如果一个进程内可以划分出多个线程，并允许在同一时间并行执行它们，就称为多线程。在多线程系统中，单个线程并不拥有系统资源，而是只拥有少量在运行过程中必不可少的资源（程序计数器，一组寄存器和栈），系统资源由同一个进程中的各线程共享，因此线程之间的通信简便，调度切换效率高。

在早期计算机系统中，操作系统进行资源分配和独立调度执行的基本单位是进程。在单核 CPU 时代，这种程序执行方式并无大碍。但是，随着**对称处理机**（Symmetric Multi-Processing, SMP，在一台计算机中汇集了一组处理器）和多核心 CPU 的出现，以进程为调度执行单位出现了许多弊端。一是进程作为资源拥有者，在创建、撤销与切换时存在较大的时空开销；二是由于多处理机可以同时满足多个运行单位，而多进程并行执行的开销过大，进程间切换的效率较低，于是在 20 世纪 80 年代出现了线程。

线程是以时间片为单位进行轮转执行的，如以 1 秒钟为单位时间片，则在有 4 个线程的单 CPU 系统中，每个线程运行 1 秒钟后，就马上切换至另一个线程执行。由于线程间的切换也存在少量的系统开销，因此在只有一个 CPU 的系统中，多线程并不比单进程的执行存在什么优势。但是，在具有多 CPU 的系统中，就可以让每个 CPU 单独执行一个线程，线程的效率优势就显示出来了，图 9-1 是 4 个线程在单核 CPU 和 4 核 CPU 中的调度执行过程示意，在 4 核 CPU 中，4 个线程可以同时执行，而在单核 CPU 中只能线性执行，效率显然要高得多。

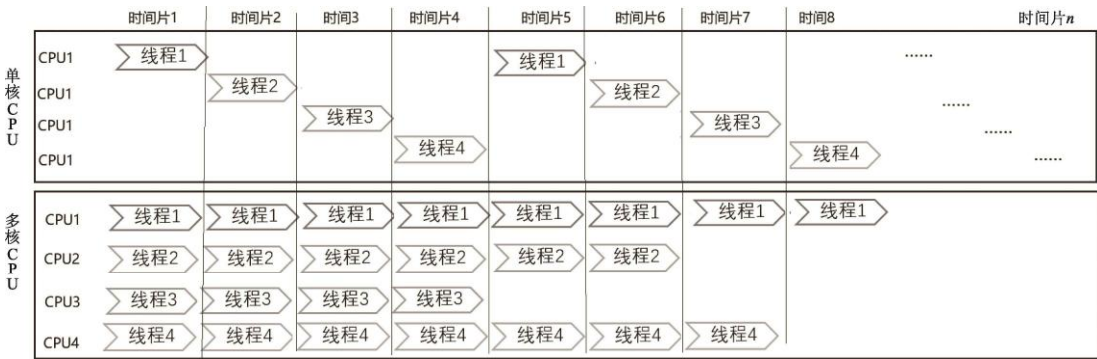


图 9-1 单核 CPU 与多核 CPU 的线程执行方式与效率对比

在现代操作系统中，进程是操作系统分配资源的基本单位，线程则是系统独立运行和独立调度的基本单位。由于线程比进程更小，基本上不拥有系统资源，并且能够共享同一进程内的资源，线程间的调度切换比进程间的调度所付出的代价要小得多，因此能够提高系统内多个程序并发执行的程度，提高系统资源的利用率和吞吐量。图 9-2 概述了程序、进程和线程的关系。

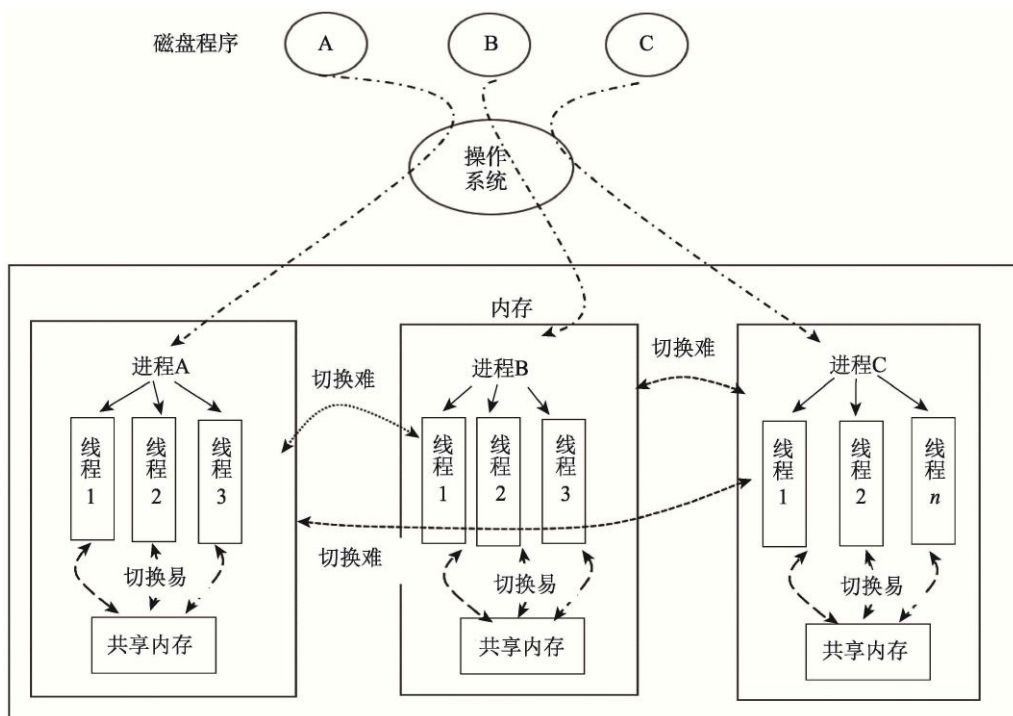


图 9-2 程序、进程和线程的关系

之前的 C++ 标准并不支持线程，如果要在 C++ 中进行线程设计，需要通过操作系统提供的线程 API (Application Programming Interface) 函数才能够实现。主流操作系统又分为 Linux 和 Windows 两大系列，两者的线程 API 并不相同，因此需要针对实际操作系统进行开发。

在 Linux 的 pthread.h 头文件中，提供了创建线程的 API 函数，如下：

```
int pthread_create(pthread_t* thread,           // 返回创建的线程 ID
                  const pthread_attr_t* attr, // 设置线程属性，通常为 NULL（用系统默认属性）
                  void* (start_routine)(void*), // 设置线程函数
                  void* arg);                 // 传入所设置的线程函数的参数
```

同样，Windows 系统在 windows.h 头文件中提供了创建线程的 API 函数，如下：

```
HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes, // 系统安全属性，常取 NULL
                   SIZE_T dwStackSize, // 线程栈大小，常设为 0（表示用系统默认值）
                   LPTHREAD_START_ROUTINE lpstartAddress, // 设置线程函数
                   LPVOID lpParameter, // 传入设置的线程函数的参数
                   DWORD dwCreationFlags, // 设置启动方式，常设为 0（创建后立即启动）
                   LPDWORD lpThreadId // 返回创建的线程 ID
);
```

操作系统 Linux 和 Windows 的线程 API 函数具有各自固定的格式，但它们的主要参数和创建线程的方法基本相同。现在，以操作系统 Windows 中的线程设计为例，简要介绍线程 API 函数的应用方法，以及线程的结构和运行原理。

【例 9-1】简单 Windows API 线程设计。

```
// Eg9-1.cpp
#include<windows.h> // 线程 API 函数定义在此头文件中
#include<stdio.h>
```



```

DWORD CALLBACK WinThread(LPVOID lpParameters) {           // L1, Windows 线程
    while (true) {
        static int i = 0;
        Sleep(1000);                                     // L2
        printf("%d 秒过去了\n", ++i);
    }
    return 0;
}
int main() {
    DWORD threadID;
    HANDLE hthread = CreateThread(NULL, 0, WinThread, NULL, 0, &threadID); // L3
    if (hthread == NULL) {                                // L4
        printf("线程创建失败.\n");
        return -1;
    }
    printf("thread: %d\n", threadID);                      // L5
    Sleep(5000);                                           // L6
    return 0;
}

```

运行本程序，结果如下：

```

thread: 8456
1 秒过去了
2 秒过去了
3 秒过去了
4 秒过去了
5 秒过去了

```

语句 L1 建立了一个 Windows 的线程函数 WinThread（也可以是其函数名），要使该函数成为合法的线程，其调用方式必须是 `_stdcall`，而 Windows 中两个宏 `WINAPI` 和 `CALLBACK` 的值都是 `_stdcall`，因此语句 L1 中将 WinThread 的调用方式指定为 `CALLBACK` 是符合 Windows 系统线程规范的。语句 L2 调用 windows.h 头文件中的函数 Sleep() 停顿 1 秒钟，函数 Sleep() 的参数以微秒为单位，Sleep(1000) 即休眠 1 秒钟。

在 C++ 中，当程序被执行时，会首先创建函数 main() 的线程并执行它。main 线程是主线程，在此之后创建的线程属于它的子线程。主线程优先获得 CPU 资源，且主线程执行完毕，程序（进程）也就执行完了，这时子线程也会被迫结束。

语句 L3 调用 Windows 的线程创建函数 CreateThread()，将语句 L1 定义的函数 WinThread() 创建为主线程 main 的子线程（设置函数 CreateThread() 的第 3 个参数为 WinThread），设置该线程的执行方式为创建后立即执行（将函数 CreateThread() 的第 5 个参数值设置为 0），并将该线程的 ID 保存在 threadID 变量中（设置为 CreateThread() 函数的第 6 个参数）。

语句 L5 输出了 CreateThread() 创建的子线程 ID，即 8456。语句 L6 让主线程延时 5 秒钟，其间子线程 WinThread 是一直处于运行状态的，每过 1 秒钟输出一个结果。当等待 5 秒后，主线程 main 执行结束，子线程 WinThread 也会被强制销毁，整个程序就结束了。

例 9-1 中的线程不具有跨平台移植能力，只能在 Windows 操作系统中运行，不能在 Linux 操作系统中运行。此外，API 线程的设计也不方便，对线程的函数原型要求较为严格（参数个数，函数返回类型都需要按要求设置）。

为此，C++ 11 提出了线程标准，解决了这样的问题。图 9-3 是 C++ 11 标准中常用的线程类及其成员函数。

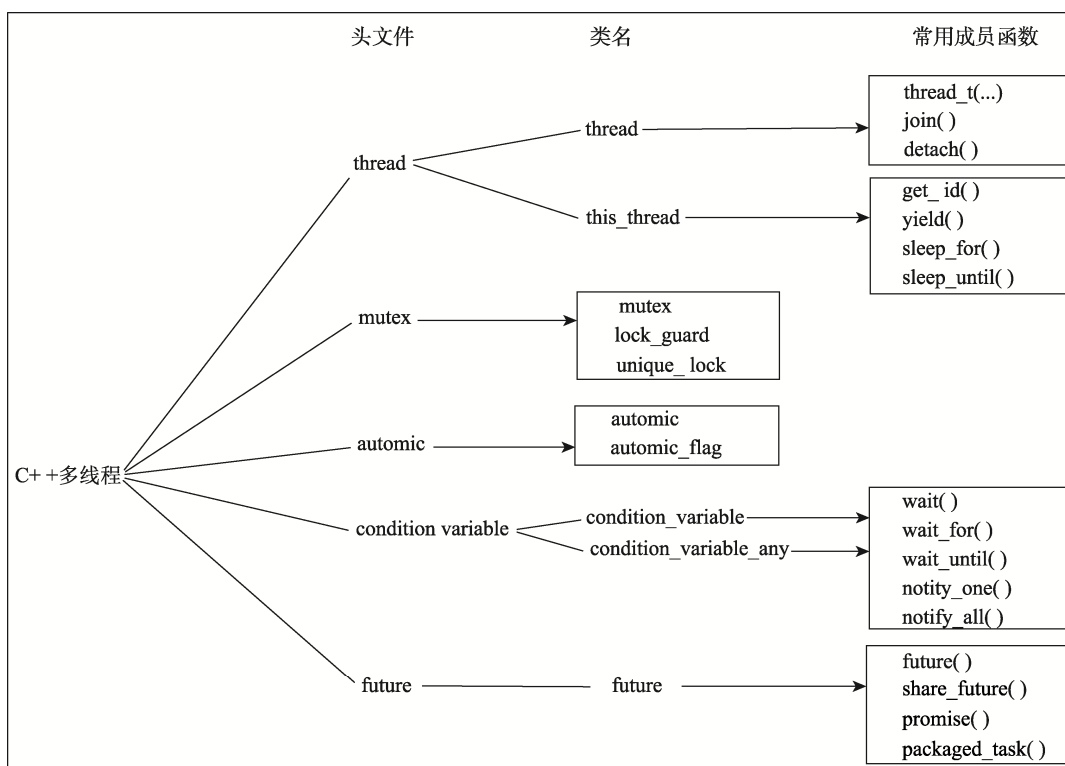


图 9-3 C++ 11 常用线程的类及其成员函数

C++ 11 标准中的线程类 `thread` 具有跨平台运行能力，同一个线程程序可以在 Linux 和 Windows 操作系统中运行。此外，对作为线程的函数也没有特别的要求，可以将任何函数作为线程函数。下面用 C++ 11 中的 `thread` 线程类创建两个简单的线程，一个有参数，一个没有参数，借以了解线程的创建和参数传递方法。用 `thread` 创建线程的方法如下：

```
thread t(funcname, para1, ..., paran);
```

其中，`thread` 是头文件 `thread` 中定义的线程类，`para1`、`...`、`panrn` 是传递给线程函数 `funcname` 的调用实参。

【例 9-2】 用 C++ 11 的线程类 `thread` 创建无参线程函数 `thread1()` 和有参线程函数 `thread2()`。

// Eg9-2.cpp

```
#include<iostream>
#include<thread>
#include<Windows.h>
using namespace std;

void thread1() {
    while (true) {
        Sleep(1000);
        cout<<"thread 1 "<<endl;
    }
}
```

// L1, 普通无参函数

```

void thread2(int a, int b) {                                     // L2, 普通有参函数
    while (true) {
        Sleep(500);
        cout<<"thread 2: "<<a<<"+"<<b<<" = "<<a+b<<endl;
    }
}
int main() {
    std::thread t1(thread1);                                     // L3, 创建无参线程
    std::thread t2(thread2, 1, 2);                               // L4, 创建有参线程
    Sleep(2000);
    // t1.detach();                                             // L5, 让线程与对象 t1 分离
    // t2.detach();                                             // L6, 让线程与对象 t2 分离
    return 0;
}

```

运行这个程序，将产生如下输出：

```

thread 2: 1+2 = 3
thread 1
thread 2: 1+2 = 3
thread 2: 1+2 = 3
End main                                                    // L7, 主线程 main 结束
thread 1
thread 2: 1+2 = 3
thread 2: 1+2 = 3
.....                                                       // L8, 不断重复上面的输出

```

但是，在产生上面输出的同时，这个程序会因异常而崩溃！其原因是，主函数 `main()` 在休眠 2 秒钟后会结束运行，其中的线程对象 `t1` 和 `t2` 会因程序结束而被销毁，但绑定到线程对象 `t1` 和 `t2` 上的线程还会继续运行，因此产生错误。L7 位置的输出就是 `main` 线程结束时输出的，其后的输出表明，在 `main()` 结束后，子线程仍然在运行状态中。

可以取消语句 L5、L6 的注释，让线程与其绑定的对象分离，程序就能够正常运行了。但这并不是一种好的解决方法，因为通常情况下都需要通过对象去控制线程的运行状态（如获取线程 ID、查看运行状态、控制它的生命周期等），当线程脱离对象后，就不便进行这些控制了。最好的办法是让主线程等待子线程执行结束后，再结束主线程，具体办法见 9.2 节。

9.2 线程等待和线程ID获取

9.2.1 线程等待

为了处理好主线程与子线程之间的控制流程，类 `thread` 提供了成员函数 `detach()` 和 `join()` 来解决主线程结束了而子线程仍然在运行的问题。

`detach()` 的作用是将子线程与创建它的线程对象分离开来，以此避免线程对象与它拥有的线程具有不同生存期的矛盾，其用法如例 9-2 所示（语句 L5 和 L6），在此不再介绍。

`join()` 的作用是将子线程加入让主线程等待的队列，即：让子线程执行完成后主线程再继续执行的线程队列。也就是说，`join()` 的作用是阻塞主线程的执行，让对应的子线程先执行。

在这种情况下，子线程可以安全地访问主线程中的资源，主线程则会等待子线程结束，回收子线程占用的系统资源后，再继续执行。

一个子线程只能调用 `join()` 和 `detach()` 中的一个，而且只允许调用一次，同时要求调用这两个函数的子线程处于运行状态，可以通过 `joinable()` 来判断线程是否可以成功执行 `join` 或 `detach` 操作，其基本用法如下：

```
thread t(treadfunc, para1 ...);           // 创建线程对象 t
t.joinable();                             // 判断 t 是否可以加入 join 队列，返回 true 或 false
t.join();                                 // 阻塞主线程，让主线程等待 t 优先执行
t.detach();                               // 分离 t 和它控制的线程，让线程独立
```

【例 9-3】 修改例 9-2，让主线程等待子程结束后再退出。

```
// Eg9-3.cpp
#include<iostream>
#include<thread>
#include<Windows.h>
using namespace std;

void thread1() {
    while(true) {
        Sleep(1000);
        cout<<"thread 1"<<endl;
    }
}

void thread2(int a, int b) {
    while(true) {
        Sleep(500);
        cout<<"thread 2 : "<<a<<"+"<<b<<" = "<<a+b<<endl;
    }
}

int main() {
    std::thread t1(thread1);
    std::thread t2(thread2, 1, 2);
    if(t1.joinable())                               // L1
        t1.join();
    if(t2.joinable())                               // L2
        t2.join();
    Sleep(2000);
    return 0;
}
```

程序运行结果如下，

```
thread 2 : 1+2 = 3
thread 1
thread 2 : 1+2 = 3
thread 2 : 1+2 = 3
thread 1
.....
```

// 重复上面的输出

语句 L1、L2 检测 t1、t2 是否符合阻塞条件，如果符合条件，就将它们加入 main 线程的

等待队列中。这样，main 线程就会等待 t1、t2 对象的子线程执行完成后才退出系统。由于 t1、t2 分别拥有的子线程 thread1、thread2 都是死循环，因此 main 线程会永远等待，程序不会正常结束。

9.2.2 获取线程 ID

线程被创建后，系统会为它分配一个线程 ID。这个 ID 在整个操作系统范围内是唯一的，在程序中可以用线程 ID 来识别不同的线程。C++ 11 提供了两种获取线程 ID 的方法，分别是类 this_thread 中的成员函数 get_id() 和类 thread 中的成员函数 get_id()，如下所示：

```
std::this_thread::get_id();           // static 成员函数
std::thread::get_id();                 // 非 static 成员函数
```

类 this_thread 中的 get_id() 是一个静态成员函数，用于获取当前线程的 ID，不需要创建对象就可以通过类 this_thread 直接引用它。而类 thread 的 get_id() 是一个非静态的成员函数，必须创建类 thread 的对象后，通过对象才能够引用它。

注意：成员函数 get_id() 获取的线程 ID 是一个封装好的类类型 thread::id，可以用 cout 直接输出，但不能直接作为整数使用，也不能够直接转换成整数。如果需要将 id 作为整数使用，可以先将其转换成一个 ostringstream 类型的字符串输出流对象，再将此对象转换成字符串类型，最后才能够将该字符串转换成整数。下面的例子就采用了这样的方法，将主线程 main 的线程 ID 转换成了整数。

【例 9-4】设计线程 thread1 创建一个磁盘文件，并将自己的线程 ID 和一串字符写入文件，创建线程 thread2 读取 thread1 创建的文件内容，用静态和非静态的 get_id() 获取 thread1 的线程 ID，并将主线程 main 的线程 ID 转换为整数。

```
// Eg9-4.cpp
#include<iostream>
#include<thread>
#include<fstream>
#include<sstream>
using namespace std;

void thread1(string filename) {
    ofstream outfile(filename);           // L1, 创建磁盘文件
    outfile<<this_thread::get_id()<<"\t"  // L2, 在文件中写入线程 ID 和字符串
        <<"thread1 write this string!"<<endl;
    cout<<"in thread 1, ID: "<<this_thread::get_id()<<endl; // L3, 输出当前线程的 ID
}

int thread2(string filename) {
    ifstream infile(filename);           // L4, 可打开 thread1 创建的文件
    char s[100];
    cout<<"I am thread 2, thread1 write the following string: "<<endl; // L5
    while(!infile.eof()) {               // L6, 读出 thread1 建立的文件数据
        infile.getline(s, 100);
        cout << s << endl;              // L7, 输出文件中的内容
    }
    return 1;
}
```

```

}

int main() {
    thread t1(thread1, "D:\\\\abc.txt");           // L8, 创建 t1 线程对象, 建立 abc.txt 文件
    thread t2(thread2, "D:\\\\abc.txt");           // L9, 创建 t2 线程对象, 读取 abc.txt 文件
    cout<<"thread1 ID: "<<t1.get_id()<<endl;       // L10, 获取 t1 线程对象的线程 ID
    if (t1.joinable())
        t1.join();
    if (t2.joinable())
        t2.join();
    cout<<"main thread ID: "<<this_thread::get_id()<<endl; // L11, 输出主线程 main 的 ID
    thread::id mid = std::this_thread::get_id();     // L12, 获取主线程 main 的 ID
    ostringstream oss;
    oss<<mid;                                         // L13, 转换主线程 ID 为字符串流对象
    std::string str = oss.str();                     // L14, 将 ID 对象转换为字符串 ID
    std::cout<<"main thread ID: "<<str<<std::endl;
    unsigned long long threadid = std::stoull(str); // L15, 将字符串 ID 转换为数值型
    std::cout<<"main thread ID: "<<threadid<<std::endl;
    return 0;
}

```

执行程序, 其中的一个输出结果如下:

```

thread1 ID: 17608                                // 语句 L10 的输出
I am thread 2, thread1 write the following string: // 语句 L5 的输出
17608 thread1 write this string!                  // 语句 L6、L7 的输出
in thread 1, ID: 17608                            // 语句 L3 的输出
main thread ID: 18232
main thread ID: 18232
main thread ID: 18232

```

从前 4 行输出结果可以看出, 程序并非按照 `main` 中的代码顺序执行, 而是同时执行线程对象 `t2` 和 `t1` 中的线程, 如果先执行 `t2` 后再执行 `t1` 的线程, 就不会有第 3 行的输出, 因为第 3 行输出内容是从 `t1` 的线程创建的文件中读出的内容 (如果不执行 `t1` 的线程, 文件中就不会有内容)。

多次运行这个程序, 每次输出的结果可能都不相同, 这就是多线程并发。本程序共有 3 个线程同时执行, 即主线程 `main` 和 `t1`、`t2` 对象拥有的两个子线程。每个线程都可能最先输出, 也可能最后输出, 这就是多线程并发运行的实际情况。

语句 L3 和 L10 分别用静态成员函数 `this_thread::get_id()` 和非静态成员函数 `thread::get_id()` 获取 `t1` 对象的线程 ID, 结果是相同的, 都是 17608。语句 L12~L15 则展示了将线程 ID 对象转换成整数类型的过程。

9.3 类和线程

前面介绍的线程都是非类成员的普通函数, 是否能够将类的成员函数设置为线程函数呢? 因为成员函数作线程能够更方便地应用对象自身的功能函数, 便捷地处理对象内部的数据, 灵活高效地实现线程的功能。

由于 Linux 和 Windows 的线程创建函数对参数表和返回类型都有严格的要求，线程函数的参数必须与其显式声明一致。因此，当用 Linux 或 Windows 的线程 API 设计类的某成员函数为线程时，只能将它设置为静态类型的成员函数，不能够设置为非静态成员函数。原因是非静态成员函数的第 1 个参数是系统隐式传递的 `this` 指针，类似于如下形式：

```
class A {
    .....
    void f1(int a,int b) {...}
    static void f2(int a, int b) {...}
}
```

`f1()`和 `f2()`被编译器处理后，其原型变成了如下形式：

```
void f1(A *this, int a, int b);
void f2(int a, int b);
```

`f1()`的函数原型发生了变化，不符合作为线程函数的参数限定要求，所以不能够作为类的线程成员函数，`f2()`才符合要求。

但是，运用 C++ 11 的类 `thread` 创建类的线程成员函数就不受 `static` 的限制了，静态和非静态成员函数都可以设置为类的线程成员函数。用类 `thread` 创建线程成员函数的原型如下：

```
thread t(func, p1, p2, ...);           // 创建线程对象 t
```

其中，`func` 是线程函数名，`p1`、`p2` 则是函数 `func` 的形式参数。如果 `func` 是类的非静态成员函数名，它的第一个参数是类的 `this` 指针，只需在 `p1` 前面传递类的 `this` 指针就符合 `thread` 创建线程函数的要求了。

【例 9-5】 为类 `myThread` 设计线程成员函数 `Write()`，将它的线程 ID 和一些字符串写入磁盘文件，设计 `static` 线程成员函数 `Read()`读取并输出 `Write()`创建的磁盘文件内容。

```
// Eg9-5.cpp
#include<fstream>
#include<sstream>
#include<thread>
#include<iostream>
using namespace std;

class myThread {
    shared_ptr<thread> t1, t2;
public:
    myThread() { t1 = t2 = nullptr; }
    ~myThread(){}
    void Write(string filename) {
        t1.reset(new thread(&myThread::thread1, this, filename)); // L1
        if (t1->joinable())
            t1->join();
    }
    void Read(string filename) {
        t2.reset(new thread(&myThread::thread2, filename)); // L2
        if(t2->joinable())
            t2->join();
    }
    void thread1(string filename) {
```

```

        ofstream outfile(filename);
        outfile<<this_thread::get_id()<<"\t"<<"string1"<<endl;
        outfile<<this_thread::get_id()<<"\t"<<"string2"<<endl;
        outfile.close();
    }
    static void thread2(string filename) {
        ifstream infile(filename);
        char s[200];
        while (!infile.eof()) {
            infile.getline(s, 100);
            cout<<s<<endl;
        }
    }
};

int main() {
    myThread t;
    t.Write("D:\\abc.txt");
    t.Read("D:\\abc.txt");
    return 0;
}

```

本例用 `shared_str` 指针管理线程对象，因此不必回收用 `new` 分配的动态内存空间。当然，也可以用非智能指针或 `thread` 成员对象管理 `myThread` 类的读写线程函数。执行程序后，结果如下：

```

10760  string1
10760  string2

```

语句 L1 是将非静态成员函数设置线程的方法，L2 是将静态成员函数设置为线程的方法。通过对比可以发现，当以非静态成员函数创建线程时，需要向线程函数多传递一个 `this` 指针。

9.4 线程同步

假设多个线程需要操作同一资源，如读写同一个内存变量，修改同一个磁盘文件，使用同一台打印机，如果不加控制就会产生错误，这种控制技术称为**线程同步**。其基本思想是，当一个线程在对某内存区域进行写操作时，其他线程都不可以对这个内存区域进行操作，需要等到该线程完成对该内存区域的写操作并释放对它的控制后，其他线程才能够对该内存区域进行操作；如果所有线程执行的都是读操作，就可以同时执行。线程同步的方法则是用互斥锁、信号量、条件变量、读写锁等技术对多线程共用的内存区域加以保护和使用控制，以避免多线程访问时所产生的冲突问题。

9.4.1 互斥锁

互斥锁即 `mutex`，是 C++ 11 提出的最基本的锁类型，作用是对多线程共同访问的资源进行保护。其主要成员如下：

```
mutex::lock();
```



```
mutex::unlock();
mutex::try_lock();
```

一个 `mutex` 在同一时刻最多只能属于一个线程，获取 `mutex` 的线程就成为它的拥有者，可以对 `mutex` 实施 `lock` 操作。等到该线程执行 `unlock` 操作后，其他线程才能获得该 `mutex`。互斥锁类似公园的长椅，如果有人占用了，其他人需要等占用者走了才能够用，如果是空闲的，就先到者先用。C++标准中的互斥锁类型如表 9-1 所示。

表 9-1 C++标准中的互斥锁类型

互斥锁类型	标准	说 明
<code>mutex</code>	C++ 11	基本互斥锁
<code>timed_mutex</code>	C++ 11	有限时机制的互斥锁
<code>recursive_mutex</code>	C++ 11	能被同一线程递归锁定的互斥锁
<code>recursive_timed_mutex</code>	C++ 11	<code>timed_mutex</code> 和 <code>recursive_mutex</code> 双重特点的互斥锁
<code>shared_mutex</code>	C++ 17	共享互斥锁
<code>shared_timed_mutex</code>	C++ 14	有限时机制的共享互斥锁

【例 9-6】 设计一个抢占教室座位号的程序，假设在 3 秒内，每名学生每次只可以占 1 个座位，但可以占座 3 次。

线程可以轻松模仿这个过程，如下面的函数 `occuSeat()` 所示，它以学生姓名为参数，每调用一次函数就表示某学生的一次抢占座位行动。

```
// Eg9-6.cpp
#include<iostream>
#include<windows.h>
#include<thread>
using namespace std;

int seatnum = 0;
void occuSeat(string name) {
    for (int i = 0; i < 3; i++) {
        ++seatnum;
        cout<<name<<"抢占了座位号: "<<seatnum<<endl;
        Sleep(3000);
    }
}

int main() {
    thread t1(occuSeat, "张三");
    thread t2(occuSeat, "李四");
    t1.join();
    t2.join();
    return 0;
}
```

程序运行结果如下：

```
李四抢占了座位号: 张三抢占了座位号: 2
2
张三抢占了座位号: 4
```

李四抢占了座位号：4
李四张三抢占了座位号：6
抢占了座位号：6

这个结果表明，同一个座位被两个学生同时抢占了，显然是不对的。因为线程对象 t1 和 t2 同时读写了同一内存区域 seatnum。如果采用 mutex 互斥锁对 seatnum 进行保护，使其在同一时间内只能被一个线程读写，就不会出现这样的问题了。

【例 9-7】 设计一个抢占教室座位号的程序，假设在 3 秒内，每名学生每次只可以占 1 个座位，可以占座 3 次，但同一次座位不允许被多次抢占。

```
// Eg9-7.cpp
#include<mutex>
#include<thread>
#include<iostream>
#include<windows.h>
using namespace std;

int seatnum = 0;
mutex seatmux; // L1, 定义互斥锁
void occuSeat(string name) {
    for (int i = 0; i < 3; i++) {
        seatmux.lock(); // L2, 锁住互斥锁
        ++seatnum;
        cout<<name<<"抢占了座位号："<<seatnum<<endl;
        seatmux.unlock(); // L3, 释放互斥锁
        Sleep(3000); // L4, 等待 3 秒钟
    }
}

int main() {
    thread t1(occuSeat, "张三");
    thread t2(occuSeat, "李四");
    t1.join();
    t2.join();
    return 0;
}
```

程序运行结果：

张三抢占了座位号：1
李四抢占了座位号：2
李四抢占了座位号：3
张三抢占了座位号：4
张三抢占了座位号：5
李四抢占了座位号：6

程序运行结果是正确的。其原因是，假设 t1 和 t2 两个线程对象同时执行到语句 L2 处，则只有一个线程（当前是 t1）能够执行“seatmux.lock();”语句，获得 seatmux 互斥锁，另一个线程只能等待已经获得 seatmux 互斥锁的线程执行完语句 L3 处的“seatmux.unlock();”后，才能执行 L2 处的“seatmux.lock();”并获得 seatmux 互斥锁，此后才得以执行后续程序语句。由此可知，多线程可以利用互斥锁控制程序代码在同一时间内只能被一个线程执行，从而保证多线程对内存变量的正确读写。

9.4.2 读写锁 C++ 17

`shared_mutex` 是 C++ 17 标准才引入的读写锁，与 `mutex` 互斥锁相比较，`shared_mutex` 允许线程具有更高的并发性。因为 `mutex` 只有加锁或者不加锁两种状态，而且一次只允许一个线程加锁；`shared_mutex` 也称为“共享 - 独占锁”，允许多个线程同时以读模式加锁，但只允许一个线程以写模式加锁，并且读时不允许写、写时不允许读。即读是共享的，写是独占的。比如，有 10 个线程要读取同一个内存区域的值，用 `mutex` 锁只允许一个线程同时读数据，用 `shared_mutex` 锁允许 10 个线程同时读数据，显然更加高效。

`shared_mutex` 锁有读模式加锁、写模式加锁和不加锁三种状态。C++ 标准引入了 `unique_lock`（独占锁）和 `shared_lock`（共享锁）两个对象来配合 `shared_mutex` 的使用，以简化对线程共享数据的读写操作。如果某线程已经通过 `unique_lock` 获得了 `shared_mutex` 锁，那么其他线程就不能获得该锁，尝试获得此锁（无论是读模式还是写模式）的线程会被阻塞；当没有任何线程获得独占锁时，其他线程才能用 `shared_lock` 获得 `shared_mutex` 锁。此外，每个线程在同一时刻只能获得 `shared_mutex` 的一个锁（共享锁或独占锁）。

`unique_lock` 和 `shared_lock` 对象在被定义时会自动调用构造函数对 `shared_mutex` 加锁，在对象失去作用域时会自动调用析构函数对其锁住的 `shared_mutex` 对象解锁，所以不需要在程序中使用 `unlock` 操作对其锁住的 `shared_mutex` 对象解锁。

【例 9-8】 用 `shared_mutex` 设计读写数据的线程，实现对同一内存数据的写入和同时读取功能。

```
// Eg9-8.cpp
#include<iostream>
#include<thread>
#include<shared_mutex>
using namespace std;

shared_mutex rwlock;                                // L1, 定义读写锁
int sharedata = 0;                                   // L2, 共享内存区域
void readData(string tname) {                        // L3, 读线程函数
    for(int i = 0; i < 12; i++) {
        shared_lock<shared_mutex> rlock(rwlock);    // L4, 对 rwlock 申请读锁
        cout<<tname<<"\tdata = "<<sharedata<<endl;
    }                                                 // L5, 自动释放 rwlock 的读锁
}

void writeData(string tname) {                       // L6, 写线程函数
    for (int i = 0; i < 5; i++) {
        unique_lock<shared_mutex> wlock(rwlock);    // L7, 对 rwlock 申请写锁
        sharedata++;
        cout<<tname<<"\tdata = "<<sharedata<<endl;
    }                                                 // L8, 自动释放 rwlock 的写锁
}

int main() {
    thread w1(writeData, "w1");
    thread r1(readData, "r1");
    thread r2(readData, "r2");
```

```

        thread r3(readData, "r3");
    r1.join();
    r2.join();
    r3.join();
    w1.join();
    return 0;
}

```

程序定义了 1 个写线程 w1 和 3 个读线程 r1、r2 和 r3，多次运行程序，结果不尽相同，其中某次运行的部分输出如下：

```

w1      data=1
r1      data=1
r2      data=1
r3      data=1
r2      data=1
r1      data=1
r1      data=1
r1r3r2  data =  data =  data = 111          // L9

r3      data = 1
r1      data = r21      data = 1          // L10

w1      data = 2
r3      data = 2
r1r2    data =  data = 22          // L11
.....

```

由输出结果可知，三个读线程 r1、r2、r3 都读到了相同的数据，表明它们同时对 `rwlock` 加读锁（`shared_lock`）是成功的（语句 L4）。在输出结果的同一行出现了 r1、r2、r3（语句 L9、L10、L11）表明三个线程是同时运行的，并且对 I/O 设备进行了抢占，一个线程还没有输完数据，输出设备就被另一个线程抢占了。

三个读线程 r1、r2、r3 与写线程 w1 是冲突不共存的，它们不能够同时执行。因此，输出结果中不会有 r1、r2 或 r3 的输出行中出现 w1 的情况。即，在执行读线程的同时，不允许执行写线程；也不会有输出 w1 的行中出现 r1、r2 或 r3 的情况，即执行写线程的同时不允许执行读线程。

9.4.3 信号量 C++ 20

信号量本质上是一个非负的整数计数器，具有 P、V 两种操作，一次 P 操作使信号量减 1，一次 V 操作使信号量加 1，主要用来控制多线程（进程）对公共资源的访问，限制并发访问共享资源的线程数量，被广泛应用于线程（进程）之间的同步和互斥控制中。其控制方法是，当信号量值大于 0 时，线程被执行，否则被阻塞（线程被挂起，直到信号量大于 0）。

信号量只能在定义时被赋值，或者通过 P、V 操作修改。C++ 的信号量是 C++ 20 标准提出的，在头文件 `semaphore` 中提供了两种类型的信号量，如表 9-2 所示。

`counting_semaphore` 和 `binary_semaphore` 都是模板类类型，具有相同的成员，在程序中的应用方法相同。所谓信号量，就是该类内部的一个计数器。`acquire()` 和 `release()` 则是能够修改该计数器的成员函数。其中，`counting_semaphore` 是实现了非负资源计数的信号量，即信号

表 9-2 信号量 C++20

类 型	主要操作	说 明
counting_semaphore binary_semaphore	acquire()	执行 p 操作。若信号量大于 0，则减少 1，线程继续；否则阻塞线程，直到信号量大于 0 再唤醒继续执行
	release(n)	执行 v 操作，信号量加 n（省略 n，则加 1）

量可以是大于 0 的整数值；binary_semaphore 是只拥有二个状态的信号量，即信号量的值只能够是 0 或 1，相当于相互锁（mutex）。

acquire()称为资源获取函数，其操作流程是，若内部计数器大于 0，则将它减少 1，并让线程继续执行；否则，就阻塞线程，直至内部计数器大于 0 且成功减少 1，才能唤醒并继续执行线程。

release(n)称为资源释放函数，其功能是执行内部计数器的加 n 操作（如果省略 n，就执行加 1 操作），如果执行 release 操作后信号量大于 0，就会唤醒 acquire 阻塞的线程。

C++并不要求 acquire()和 release()配对执行，也不要求在同一个线程内部同时执行这两个操作。因此，可以在同一线程内部，也可以在不同线程之间分开执行 acquire 和 release 操作。信号量的定义方法如下：

```

counting_semaphore<N> csem(信号量初始数量);           // N是信号量上限值
counting_semaphore csem(信号量初始数量);               // 信号量无上限值
binary_semaphore bsem(初值);                           // 初值只能够是 0 或 1

```

其中，csem 和 bsem 是信号量对象名，信号量初始数量是一个整数值。

【例 9-9】 设计 1 个线程函数，可以通过信号量同时启动 5 个线程，最多 10 线程。

```

// Eg9-9.cpp
#include <iostream>
#include <semaphore>
#include <thread>
using namespace std;

counting_semaphore<10> csem(5);                          // L1
int cakeNumber = 0;
binary_semaphore bsem(0);                                // 示例二值信号量的定义方法，程序中并没有用它
void makeCake(string threadname) {
    csem.acquire();                                       // L2
    cout<<threadname<<"\tmake Cake " << ++cakeNumber << endl;
    // csem.release();                                   // L3
}

int main() {
    cout<<"main:ready to signal :release\n";
    thread t1 = thread(makeCake, "bake A:");             // L4
    thread t2 = thread(makeCake, "bake B:");             // L5
    thread t3 = thread(makeCake, "bake C:");             // L6
    cout<<"main: signal end\n";
    t1.join();
    t2.join();
    t3.join();
    return 0;
}

```

程序运行结果如下：

```
main:ready to signal :release
main: signal end
bake C: make Cake bake A:      make Cake 2bake B:      make Cake 3
```

1

语句 L1 定义了信号量 `csem`，其初始值为 5，最大值为 10。线程函数 `makeCake()` 只执行了资源获取函数 `acquire()`，但并未释放资源。语句 L4、L5、L6 分别创建了三个线程对象 `t1`、`t2`、`t3`（对应三个线程），每执行一个线程，`csem` 的信号量减 1，因为初始值为 5，减 3 之后仍然大于 0，所以程序正常运行。从输出结果的第三行可以看到，在同一输出行中有 3 个线程的交叉输出，表明它们是并发执行的。

【例 9-10】 信号量小于 0 时，线程被阻塞。在例 9-9 中，将 L1 语句中 `csem` 信号量的初值改为 2，即修改成如下语句：

```
counting_semaphore<10> csem(2); // L1
```

其余代码不作任何修改，再次执行程序，运行结果如图 9-4 所示。

```
main:ready to signal :release
main: signal end
bake C: make Cake bake A:      make Cake 21
```

图 9-4 信号量为 0，线程被阻塞

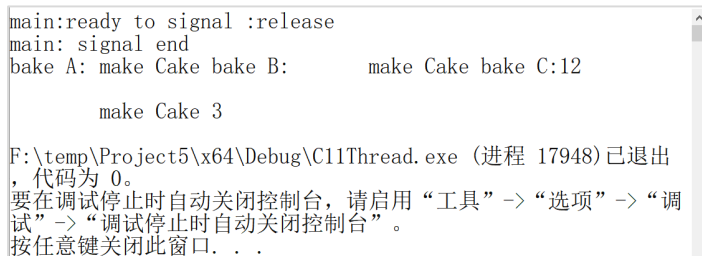
图 9-4 中只有 `bake A` 和 `bake C`，没有 `bake B` 的输出，并且程序一直处于停滞状态，表明线程 `t2` 被阻塞了。原因很简单，`csem` 的初始信号量是 2，执行语句 L4、L5、L6 成功创建了线程对象 `t1`、`t2`、`t3`。输出结果第 3 行表明，线程 `t3`（线程对象 `t3` 控制的线程，下同）首先被调度，则线程中的 `csem.acquire()` 语句会被执行，`csem` 信号量会减少为 1；接着线程 `t1` 被调度，再次执行 `csem.acquire()`，`csem` 信号量减少为 0；最后，线程 `t2` 被调度执行，当该线程中的 `csem.acquire()` 被执行时，由于 `csem` 信号量为 0，所以 `t2` 线程被阻塞，等待 `csem` 信号量大于 0，但程序并没有任何语句增加 `csem` 信号量，所以 `t2` 线程会一直等待，直到非正常结束。

【例 9-11】 信号量充足，阻塞线程被激活。在例 9-10 中，在主线程 `main` 中增加 `csem` 信号量，保障线程 `t1`、`t2`、`t3` 有充足的资源得以执行完成。

修改后的 `main()` 函数如下，其余程序代码不作修改。

```
// Eg9-11.cpp
.....
int main() {
    cout<<"main:ready to signal:release\n";
    thread t1 = thread(makeCake, "bake A:");
    thread t2 = thread(makeCake, "bake B:");
    thread t3 = thread(makeCake, "bake C:");
    cout<<"main: signal end\n";
    csem.release(3); // 释放资源，csem 信号量加 3
    t1.join();
    t2.join();
    t3.join();
    return 0;
}
```

程序运行结果如图 9-5 所示。



```
main:ready to signal :release
main: signal end
bake A: make Cake bake B: make Cake bake C:12
make Cake 3
F:\temp\Project5\x64\Debug\C11Thread.exe (进程 17948) 已退出
, 代码为 0。
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调
试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口...
```

图 9-5 信号量充足, 线程被执行

【例 9-12】 设计线程函数, 通过信号量控制多个线程依次执行 (线性执行, 无并发性)。

修改前面的程序, 只需将信号量 `csem` 的初值设置为 1, 同时当线程获得资源并使用后, 就立即释放它使信号量加 1。完整的程序如下:

```
// Eg9-12.cpp
#include <iostream>
#include <semaphore>
#include <thread>
using namespace std;

counting_semaphore csem(1); // L1
int cakeNumber = 0;
void makeCake(string threadname) {
    csem.acquire(); // L2
    cout<<threadname<<"\tmake Cake "<<++cakeNumber<<endl;
    csem.release(); // L3
}

int main() {
    cout<<"main:ready to signal :release\n";
    thread t1 = thread(makeCake, "bake A:"); // L4
    thread t2 = thread(makeCake, "bake B:"); // L5
    thread t3 = thread(makeCake, "bake C:"); // L6
    cout<<"main: signal end\n";
    t1.join();
    t2.join();
    t3.join();
    return 0;
}
```

程序运行结果如下:

```
main:ready to signal :release
main: signal end
bake B: make Cake 1
bake A: make Cake 2
bake C: make Cake 3
```

虽然本程序的代码同前面的程序并没有什么区别, 但从输出结果可以看到, 三个线程并未在同一行中输出各自的运行结果, 而是各自输出在独立的行中, 表明线程是线性执行的, 即一个线程执行完成后, 再执行第另一个线程。这是由于在信号量 `csem` 始终不会超过 1 的控

制下，线程 t1、t2、t3 无法并发执行。同时，这个输出结果表明，线程的调度执行并非按照它们的创建次序进行，而是具有一定的随机性。比如在本例中，线程 t1 首先被创建，但输出结果表明线程 t2 首先被调度。

可以用信号量控制多线程模仿于生产者—消费者模型。信号量代表产品数量，当生产者线程（增加信号量的线程）完成产品生产后，调用 `release()` 操作增加信号量，表示资源数量增加，可以根据当前资源的数量按需要唤醒指定数量的资源消费者线程（执行 `acquire` 减少信号量的线程）。资源消费者线程一旦获得信号量，就会减少资源数量，如果资源数量减少到 0，那么消费者线程将全部处于阻塞状态；当有新资源到来时，消费者线程将继续被唤醒。

【例 9-13】 3 个面包师，每次烤 1 个面包；2 个顾客，每次消费 1 个面包。设计线程，模仿这个过程。

设计思路：设计代表面包编号的全局变量 `cakeNumber`，每烤 1 个面包就让 `cakeNumber` 加 1，每消费 1 个面包就减 1；设计一个 `counting_semaphore` 类型的信号量、一个代表生产面包的线程函数 `makeCake()` 和一个代表消费面包的线程函数 `consumerCake()`；每生产 1 个面包就调用 `release` 操作，让信号量加 1，每消费 1 个面包就调用 `acquire` 操作，让信号量减 1。

```
// Eg9-13.cpp
#include <iostream>
#include <semaphore>
#include <thread>
using namespace std;

counting_semaphore csem(1); // 信号量初始为 1，也可为 0
int cakeNumber = 0;
void makeCake(string threadname) {
    cout<<threadname<<": make Cake "<<++cakeNumber<<"\t"<<endl;
    csem.release();
}
void consumerCake(string threadname) {
    csem.acquire();
    cout<<threadname<<"consumer Cake "<<cakeNumber--<<"\t"<<endl;
}

int main() {
    cout<<"main: ready to signal:release\n";
    thread t1 = thread(makeCake, "bake A:"); // L2, 面包师 1
    thread t2 = thread(makeCake, "bake B:"); // L3, 面包师 2
    thread t3 = thread(makeCake, "bake C:"); // L4, 面包师 3
    thread t4 = thread(consumerCake, "customer 1:"); // L5, 顾客 1
    thread t5 = thread(consumerCake, "customer 2:"); // L6, 顾客 2
    cout<<"main: signal end\n";
    t1.join();
    t2.join();
    t3.join();
    t4.join();
    t5.join();
}
```

多次执行程序，每次结果不尽相同，其中某次的运行结果如下：

```

main:ready to signal: release
main: signal end
bake A::make Cake 1    bake C::make Cake 2
bake B::make Cake 3
customer 1:consumer Cake 3
customer 2:consumer Cake 2

```

9.4.4 条件变量

在线程设计中存在这样一种业务逻辑,就是线程 A 通过无限次循环检测某个条件的成立,如果条件不成立就一直检测,直到条件成立时才能够执行其他业务逻辑,而条件是由另一个线程 B 修改的。两个线程共同应用的条件可以用信号量来抽象,如图 9-6 所示。

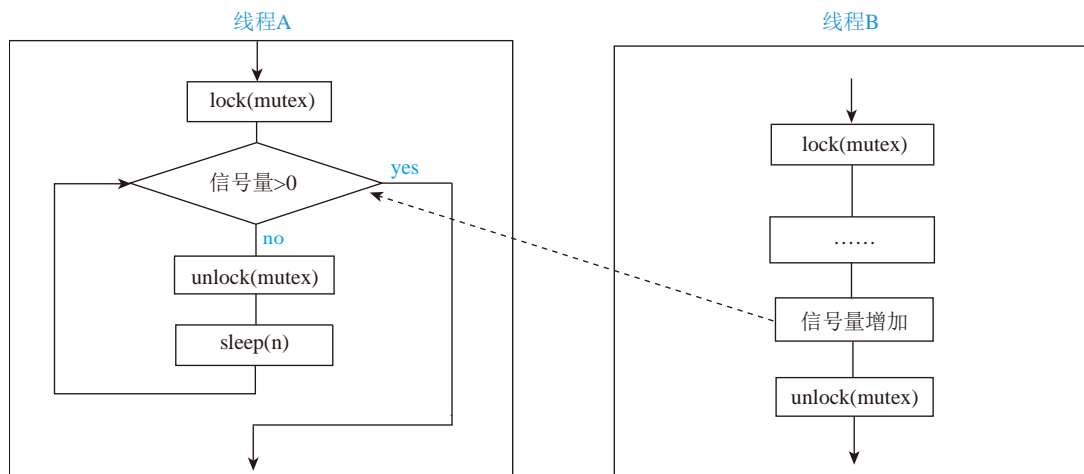


图 9-6 具有无限次循环检测线程的并发执行

线程 A 存在较大的效率问题,可能存在这样一种情况:信号不大于 0,但在它刚好释放互斥锁之后,线程 B 增加了信号量,它也会睡眠 n 秒钟,等到下次进行条件检测时才会退出循环。

如果线程 A 在检查到条件不满足(信号量不大于 0)时,就立即释放互斥锁并处于等待状态,线程 B 在增加了信号量后就主动通知线程 A,并让出互斥锁, A 就不用每次条件不符合时都等待 n 秒钟,而是条件一旦满足就马上执行,最大限度地提高运行效率。运用 C++ 条件变量和互斥锁,就能够实现这样的线程功能。条件变量的使用流程为: `lock(mutex) → wait() → unlock(mutex)`。其中, `wait()` 是条件变量的一个成员函数,会释放互斥锁并等待条件变量另一个成员函数 `notify()` 的通知,当 `wait()` 接到 `notify()` 的通知后,将执行 `try_lock()`。在多线程环境中, `try_lock()` 操作可能失败(其他线程可能趁机获得了互斥锁),产生虚假唤醒的情况。为了防止虚假唤醒,应当使用 `while` 循环来实现等待(反复尝试获得具备执行条件的互斥锁,总会成功),如图 9-7 所示。

C++ 的 `<condition_variable>` 头文件中定义了 `condition_variable` 和 `condition_variable_any` 两种类型的条件变量,它们都需要与互斥锁配合。条件变量的主要成员函数如下:

```

wait(unique_lock <mutex>&lck)
wait(unique_lock <mutex>&lck, Predicate pred)

```

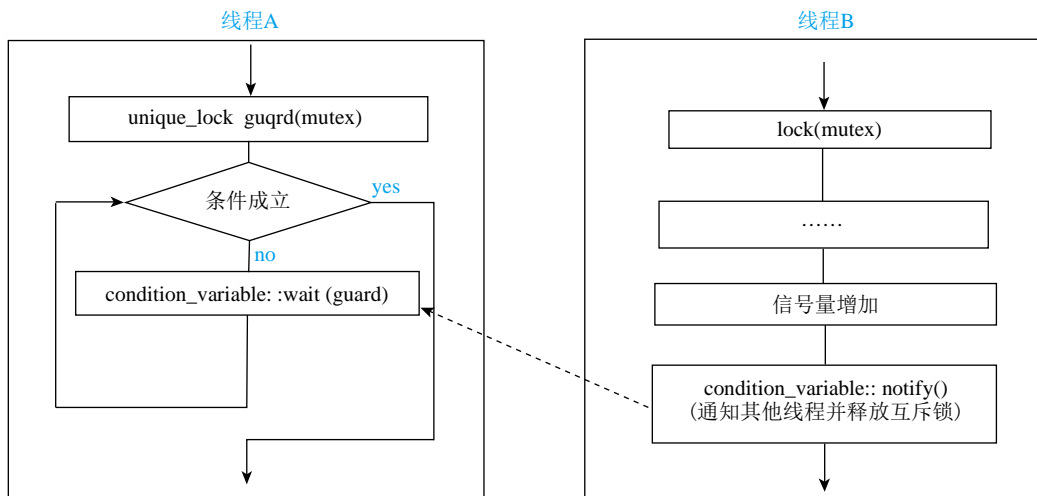


图 9-7 条件变量并发程序结构

函数 `wait()` 的功能是阻塞当前线程，直到收到 `notify()` 的通知为止。在第二种用法中，当 `pred = false` 时，阻塞线程；当 `pred = true` 时，不阻塞线程。`wait()` 可依次拆分为三个操作步骤：① 释放互斥锁 (`lck`)；② 等待在条件变量上；③ 再次获取互斥锁（当条件变量接到 `notify()` 的通知时，将再次获取 `lck`）。

```

notify_one()
notify_all()

```

`notify_one()` 只唤醒一个线程，不存在锁争用问题，所以被唤醒的线程能够立即获得锁。其余线程则会继续被阻塞，等待再次被 `notify_one()` 或者 `notify_all()` 唤醒。

`notify_all()` 会唤醒所有被阻塞的线程，因此存在锁的争用，只有一个线程能够获得锁，其余未获得锁的线程会继续被阻塞，当持有锁的线程释放锁时，这些线程会继续尝试获得锁。

条件变量最常见的一种应用场景是，多个线程同时执行任务队列中的多个任务，在刚开始时没有任务，任务队列为空，所有线程因为“任务队列为空”这个条件处于被阻塞状态。一旦有任务进来，就会以信号量的方式唤醒某些线程来处理这个任务。

【例 9-14】 设计一个面包销售的生产者 - 消费者程序。生产者不断地生产面包并放入销售队列中，消费者从队列中取出面包，并执行面包销售任务。

设计思路：在生产者 - 消费者模型中，通常由生产者产生任务后将其放入任务队列，然后通知消费者从任务队列中取出一个任务并予以执行。利用条件变量和互斥锁，可以便捷地实现这类程序模型。

方法是，让生产者线程和消费者线程通过互斥锁共同维护任务队列，实现两个线程对任务队列的异步访问，即：生产者线程获取互斥锁后将创建的任务放入任务队列，然后释放互斥锁并通知消费者线程到队列中领取任务；消费者线程在接到生产者线程的通知后从等待中激活，获取互斥锁并从队列中取出一个任务，然后释放互斥锁并执行任务。这样，生产者线程就能够再次获取互斥锁，并将新任务放入队列……

在本程序中，设计面包类 `Cake` 表示面包销售任务，其成员函数 `saleCake()` 代表面包销售，该函数输出卖面包的线程编号和被卖面包的编号。设计线程函数 `producerThread()` 代表生产者程序、`consumerThread()` 代表消费者程序，它们通过条件变量 `condivar` 运用互斥锁 `Mutex` 共同

维护和使用任务队列 `task`。

卖面包的任务队列 `task` 用 STL 库中的队列 `deque`（头文件 `deque` 中定义）建立，`deque` 中的主要成员函数如下：

```
deque::push(elem);           // 将 elem 元素插入队尾
deque::front();              // 获取队首元素，该操作不删除元素
deque::pop();                // 删除队首元素
```

完整的程序代码如下：

```
// Eg9-14.cpp
#include<thread>
#include<mutex>
#include<condition_variable>
#include<iostream>
#include<queue>
using namespace std;

class Cake {
public:
    Cake(int Id = 0) {
        cakeID = Id;
    }
    void saleCake() {
        cout<<"threadID: "<<this_thread::get_id()<<"sale a cake, cakeID: "<<cakeID<<endl;
    }
private:
    int cakeID;
};

mutex Mutex;           // L1
queue<Cake> tasks;
condition_variable condivar; // L2
void consumerThread() {
    Cake cake;
    while (true) {
        unique_lock<mutex> guard(Mutex); // L3, 锁住 Mutex
        while (tasks.empty()) {
            // 若获得互斥锁 guard 但任务队列为空，则条件变量 condivar 的 wait() 会释放 guard，不向下执行
            // 若接到 notify() 通知（由 producerThread 线程的 condivar.notify_one() 发出），则 wait()
            // 将直接获得锁，并继续执行
            condivar.wait(guard); // L4
        }
        cake = tasks.front(); // L5, 获取队列中的任务
        tasks.pop();          // L6, 删除已获取的任务
        cake.saleCake();      // L7, 执行卖面包的任务
    }
}

void producerThread() {
    int cakeID = 0;
    while(true) {
        lock_guard<mutex> guard(Mutex); // L8, 锁住 Mutex
```

```

        tasks.push(Cake(cakeID)); // L9, 建立一个入队任务
        cout<<"threadID: "<<this_thread::get_id()<<"Produce a cake, cakeID:"<<cakeID<<endl;
        condivar.notify_one(); // L10, 通知 L4 处的 wait(), 并释放 Mutex
        cakeID++;
        this_thread::sleep_for(chrono::seconds(1));
    }
}

int main() {
    thread consumer1(consumerThread);
    thread consumer2(consumerThread);
    thread consumer3(consumerThread);
    thread producer(producerThread);
    producer.join();
    consumer1.join();
    consumer2.join();
    consumer3.join();
    return 0;
}

```

程序运行的部分结果如下：

```

threadID: 13372 Produce a cake, cakeID: 0
threadID: 13064 sale a cake, cakeID: 0
threadID: 13372 Produce a cake, cakeID: 1
threadID: 13064 sale a cake, cakeID: 1
threadID: 13372 Produce a cake, cakeID: 2
threadID: 13372 Produce a cake, cakeID: 3
threadID: 13372 Produce a cake, cakeID: 4
threadID: 14388 sale a cake, cakeID: 2
threadID: 14388 sale a cake, cakeID: 3
.....

```

本线程程序的运行原理如下：生产者线程 `producer()` 首先获得互斥锁 `Mutex`，生产 1 个面包放入任务队列 `tasks`，其间 3 个消费者线程 `consumer1`、`consumer2` 和 `consumer3` 由于无法对 `Mutex` 加锁（语句 L3），因而不能够操作 `task` 队列。当生产者线程 `producer()` 完成对 `task` 队列的操作后（语句 L9）后，就通过条件变量 `condivar` 的 `notify_one()` 函数通知消费者线程，并同时释放 `Mutex`（语句 L10）。

消费者线程 `consumer1`、`consumer2` 和 `consumer3` 都采用无限循环等待条件变量 `condivar` 的通知（语句 L4），其中的某个消费者线程会获取到生产者线程 `producer()` 释放的互斥锁 `Mutex`，然后从 `tasks` 队列中取出队首的面包，执行函数 `saleCake()`，完成面包销售任务。

习 题 9

- 9.1 什么是线程，简述程序、进程和线程的关系。
 - 9.2 简述互斥锁，信号量和条件变量的执行过程。
 - 9.3 阅读分析程序，写出运行结果（某些程序运行结果不唯一，写出某种可能的结果）。
- (1)

```

#include<iostream>
#include<thread>
#include<array>
using namespace std;

void show(int i) {
    cout<<"hello threadShow"<<i<<endl;
}

int main() {
    array<thread, 3> threads = { thread(show, 1), thread(show, 2), thread(show, 3) };
    for (int i = 0; i < 3; i++) {
        if(threads[i].joinable())
            threads[i].join();
    }
    return 0;
}

```

(2)

```

#include<thread>
#include<memory>
#include<windows.h>
#include<iostream>
using namespace std;

class MyThread {
public:
    MyThread() {}
    ~MyThread() {}
    void Start() {
        stopped = false;
        pthread.reset(new std::thread(&MyThread::func, this, 1, 2));
    }
    void Stop() {
        stopped = true;
        if(pthread) {
            if(pthread->joinable())
                pthread->join();
        }
    }
private:
    void func(int a, int b) {
        while(!stopped) {
            cout<<"线程成员函数 func("<<a<<" , "<<b<<")"<<endl;
            if (a + b < 8) {
                a++;
                b++;
            }
            else
                stopped = true;
        }
    }
}

```

```

    }
private:
    std::shared_ptr<std::thread> pthread;
    bool stopped;
};

int main() {
    MyThread t;
    t.Start();
    Sleep(2000);
    t.Stop();
}

```

(3)

```

#include<iostream>
#include<windows.h>
#include<thread>
#include<mutex>
using namespace std;

int  stuID = 1000;
mutex stuIDmux;
void occuSeat(string name) {
    stuIDmux.lock();
    cout<<name<<"学号: "<<stuID<<endl;
    stuIDmux.unlock();
    Sleep(1000);
}
void makeID(int i) {
    stuIDmux.lock();
    stuID += i;
    stuIDmux.unlock();
    Sleep(1000);
}

int main() {
    thread t1(makeID, 1);
    thread t2(makeID, 2);
    thread t3(occuSeat, "张三");
    thread t4(occuSeat, "李四");
    t1.join();
    t2.join();
    t3.join();
    t4.join();
    return 0;
}

```

(4)

```

#include <iostream>
#include <semaphore>
#include <thread>

```



```

using namespace std;

counting_semaphore sema(1);
counting_semaphore semb(0);
void A() {
    for(int i = 0; i < 5; i++) {
        sema.acquire();
        cout<<"A";
        semb.release();
    }
}
void B() {
    for (int i = 0; i < 5; i++) {
        semb.acquire();
        cout<<"B";
        sema.release();
    }
}

int main() {
    thread t1(A);
    thread t2(B);
    t1.join();
    t2.join();
    return 0;
}

```

(5)

```

#include<thread>
#include<mutex>
#include<condition_variable>
#include<iostream>
using namespace std;

mutex mutx;
condition_variable cvar;
int ID = 10;
bool success = false;
void thread1() {
    {
        unique_lock<mutex>lock(mutx);
        ID = 1000;
        cout<<"the id in thread1 is: "<<ID<<endl;
        success = true;
        cvar.notify_all();
        this_thread::sleep_for(chrono::seconds(1)); // 睡眠 1 秒钟
    }
}

int main() {
    thread t(thread1);
    unique_lock<mutex>lock(mutx);
}

```

```

while(!success) {
    cvar.wait(lock);
    cout<<"the ID in main:"<<ID++<<endl;
}
cout<<"start thread successfully."<<endl;
t.join();
return 0;
}

```

9.4 设计两个线程，一个线程将“Hello C++ thread!”写入文件 A1.dat，另一个线程读出文件 A1.dat 的内容并显示在屏幕上。

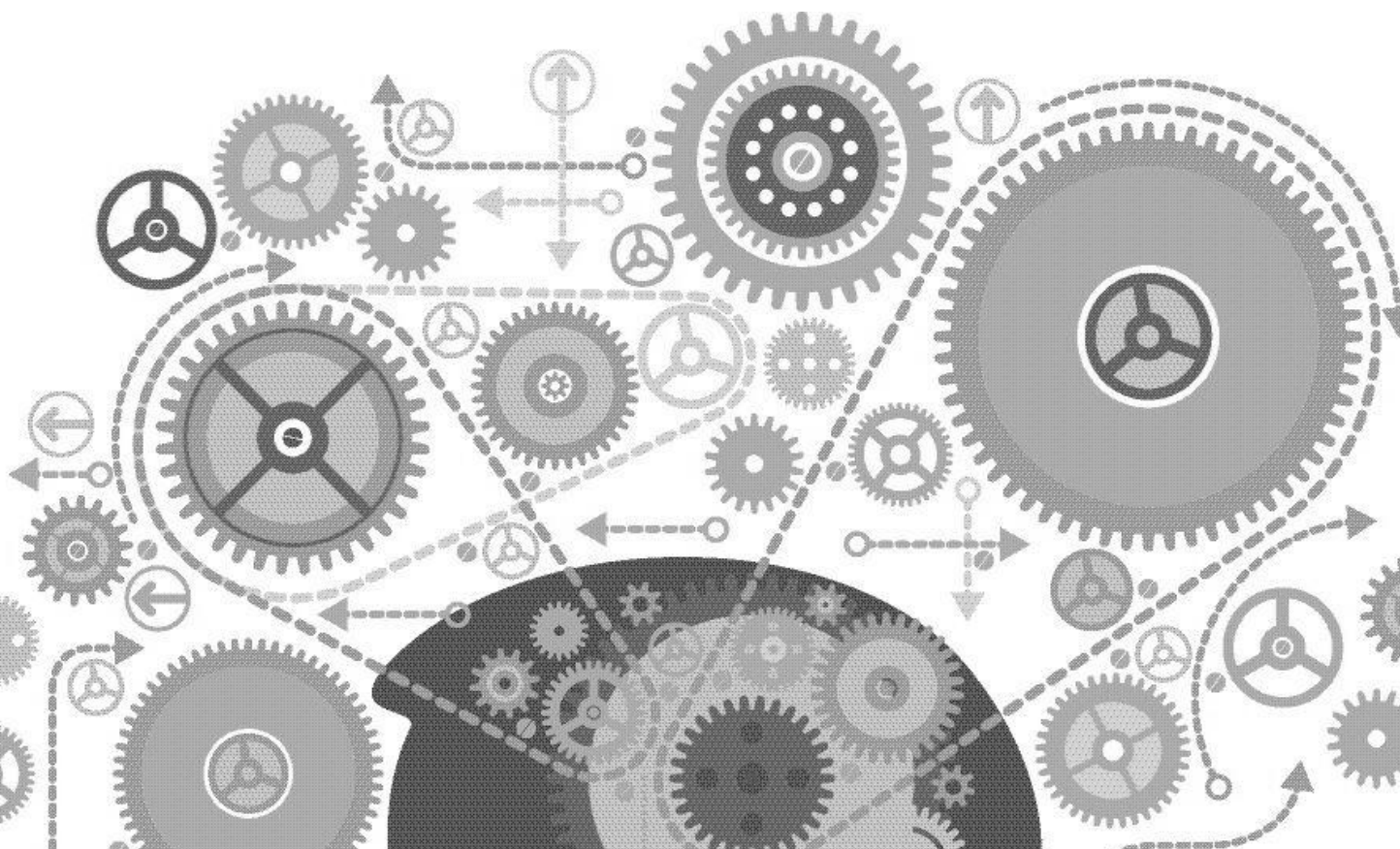
9.5 设计一个文件读写的线程程序，有一个写者和多个读者，多个读者可以同时读文件，但写者在写文件时不允许有读者在读文件，在读者读文件时也不允许有写者在写文件。

9.6 设计生产和销售盒饭的线程，一个生产者线程不断地生产盒饭并放入任务队列，每个盒饭有编号，品种和价格，品种和价格通过键盘输入，盒饭编号从 1 开始连续编号，由系统自动生成；设计 5 个消费者线程从任务队列中取出盒饭并完成任务，任务很简单，如输出盒饭的编号、品种和价格。

第 10 章

流和文件

C++具有一个功能强大的 I/O 类继承结构用于处理数据的输入/输出问题。第 1、2 章对该结构中的基本输入、输出操作及简单的文件处理进行了粗略介绍。本章在此基础上介绍 C++流类的继承结构和文件管理，包括用流类的成员函数 `get()`、`getline()`、`read()`、`put()`、`write()` 输入输出数据，以及二进制文件和随机文件的存取方法和数据格式化等内容。



10.1 C++ I/O流及流类库

计算机数据的输入、输出本质上是字符序列在主机与外设之间的有向流动。在 C++ 程序中，输入、输出操作是通过“流”处理的。所谓流，是指数据的有向流动，即数据从一个设备传递到另一个设备，分为输入流和输出流两类。输入流是指从外设流向内存的数据流，如从键盘输入数据，从磁盘文件读取数据到内存数组中的数据流都是输入流；输出流是指从内存流向外设的数据流，如数据存盘或数据打印就是数据从内存流向磁盘或打印机的输出流。

流实际上是一种对象，在使用前被建立，使用后被删除。数据的输入、输出操作就是从流中提取数据或者向流中添加数据。通常，把从流中提取数据的操作称为析取（提取），即读操作；向流中添加数据的操作称为插入，即写操作。

C++ 建立了一个十分庞大的流类库来实现数据的输入、输出操作，其中的每个流类实现不同的功能，通过继承组织在一起。图 10-1 是 C++ 流类库中一些主要流类的继承关系。

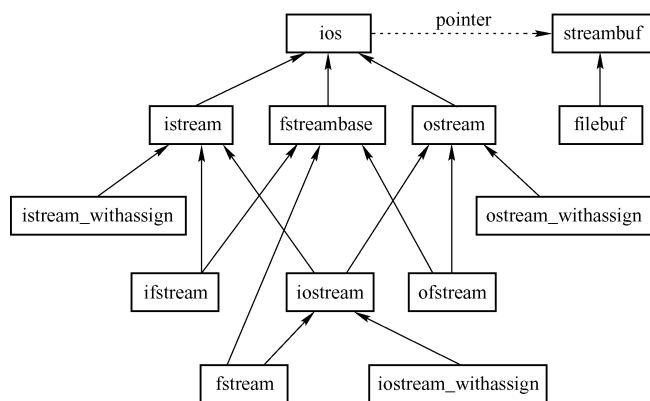


图 10-1 C++ 流类的继承关系

`ios` 是所有流类的基类，提供对流状态进行设置的主要功能。它有一个指向类 `streambuf` 的指针，`ios` 及其派生类都通过该指针，利用 `streambuf` 实现数据的输入、输出。

类 `streambuf` 主要作为其他类实现数据存取操作的支撑。它定义了对缓冲区的通用操作，如设置缓冲区，从缓冲区中读取数据，或向缓冲区写入数据等。数据存取缓冲区由一个字符序列、一个输入缓冲区指针及一个输出缓冲区指针组成，这两个指针分别指向数据存取缓冲区中要插入或析取的字符位置。类 `streambuf` 提供了与物理设备的接口，实现了缓冲或处理流的通用方法。

`istream` 是输入流类，提供从流中提取数据的操作，实现数据输入功能；`ostream` 是输出流类，提供将数据插入流的操作，实现数据输出功能；`iostream` 是类 `istream` 和类 `ostream` 的派生类，继承了类 `istream` 和类 `ostream` 的功能，支持数据输入、输出的双向操作，常用来实现数据的输入和输出功能。

类 `fstreambase` 由类 `ios` 派生，提供了文件操作的许多功能，但不会被用来进行实际的文件操作，而是作为其他文件操作类的公共基类。类 `ifstream` 用来实现文件读取操作，类 `ofstream` 用来实现文件写入操作。类 `fstream` 继承了类 `fstreambase` 和类 `iostream` 的功能，实现了文件读、写的双向操作。

类 `filebuf` 由类 `streambuf` 派生, 使用文件来保存缓冲区中的字符序列, 主要提供对上述各类的文件缓冲支持。文件有读、写、打开、关闭等常用操作。读文件就是将指定文件中的内容读到缓冲区中; 写文件就是将缓冲区中的字符写到指定的文件中; 打开文件就是将 `filebuf` 同某个磁盘文件相链接; 关闭文件就是断开 `filebuf` 与文件的链接。

为了便于程序数据的输入、输出, C++在 `iostream` 头文件中预定义了以下几个标准输入、输出流对象, 程序中包含此头文件后, 就可直接应用这些对象进行数据的输入、输出。例如:

```
ostream cout;                // cout 与标准输出设备相关联
ostream cerr;                // cerr 与标准错误输出设备相关联 (非缓冲方式)
ostream clog;                // clog 与标准错误输出设备相关联 (缓冲方式)
istream cin;                 // 与标准输入设备相关联
```

`cout`、`cerr`、`clog` 都是输出流对象。`cerr` 和 `clog` 用于输出错误信息, `clog` 常与打印机关联, 以缓冲方式输出流中的数据, 即流中的数据被先送到缓冲区, 当缓冲区满时, 才从缓冲区中被送到输出设备。`cerr` 常与显示器关联, 以非缓冲方式输出错误信息, 即输出流中的数据不通过缓冲区直接送到输出设备, 其输出速度比缓冲方式快。

在默认情况下, 标准输入设备已被关联到键盘, 标准输出设备则被关联到了显示器, 这就是在程序中可以用 `cin` 从键盘输入数据, 用 `cout` 将数据输出到显示器的原因。

10.2 I/O流类的成员函数

10.2.1 类 `istream` 的常用成员函数

类 `istream` 定义了许多用于从流中提取数据和操作文件的成员函数, 并对流的输入运算符 `>>` 进行了重载, 实现了对内置数据类型的输入功能。其中常用的成员函数是 `get()`、`getline()`、`read()`, 它们在类 `istream` 中的函数原型如下:

```
class istream:virtual public ios {
public:
    istream& operator>>(double &);                // 具有许多 operator>>重载成员函数
    .....
    int get();
    istream& get(char *, int, char = '\n');
    istream& get(char &);
    istream& getline(char *, int, char = '\n');
    istream& read(char *, int);
    istream& ignore(int = 1, int = EOF);
    int peek();
    istream& putback(char);
    .....
};
```

1. 成员函数 `get()`

从类 `istream` 的声明中可以看出, 成员函数 `get()` 大致有 3 种用法。

1) `int get()`

不带参数的成员函数 `get()` 从输入流中提取一个字符 (包括空白字符), 并且返回该字符

作为函数的调用值。当遇到文件结束符时，返回文件结束常量 EOF。

2) istream& get(char * c, int n, char = '\n')

该函数从输入流中提取 $n-1$ 个字符（包括空白字符），把它们放在字符数组 **c** 中，并在字符串结束处添加 '\0'。函数中的第 3 个参数用于指定字符结束的分隔符，其默认值是 '\n'。该函数在以下情况会结束读取字符的操作：① 读取了 $n-1$ 个字符；② 遇到了指定的结束分隔符；③ 遇到了文件结束符。**注意：**该函数不会将输入流中的结束分隔符放入数组 **c**，数据读取完成后，结束分隔符仍然保留在输入流中。

3) istream& get(char &c)

该函数从输入流中提取一个字符（包括空白字符），并且把它放在字符引用 **c** 中。当遇到文件结束符时返回 0，否则返回对 **istream** 对象的引用。

2. 成员函数 read()

成员函数 **read()** 从流中读取 **n** 字节，并将其存放到字符数组 **c** 中。**read()** 执行非格式化的操作，即对读出的字节不做任何处理，直接送到指定内存单元后由程序的类型定义进行解释。其用法如下：

```
istream& read(char *c, int n);
```

3. 成员函数 ignore() 和 getline()

成员函数 **ignore()** 从流中读取指定个数的字符但不保存，**getline()** 从流中一次读取一行字符，其用法可参见 1.4.8 节。

【例 10-1】 用成员函数 **get()** 和 **getline()** 输入数据。

```
// Eg10-1.cpp
#include <iostream>
using namespace std;


void main(){
    char c, a[50], s1[100];
    cout<<"use get() input char: ";
    while((c = cin.get()) != '\n')           // L1
        cout<<c;
    cout<<endl;
    cout<<"use get(a, 10) input char: ";
    cin.get(a, 10);                          // L2
    cout<<a<<endl;
    cin.ignore(1);                           // L3
    cout<<"use getline(s1, 10) input char: ";
    cin.getline(s1, 10);                     // L4
    cout<<s1<<endl;
}
```

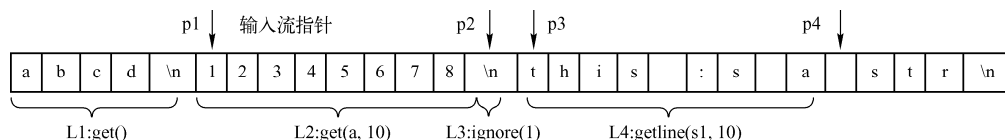
由于 **get()** 是类 **istream** 的公有成员函数，**cin** 是类 **istream** 的对象，因此 **cin.get()** 调用是合法的。下面是程序执行时的一组输入数据和输出结果：

```
use get() input char: abcd
ab cd
use get(a, 10) input char: 12345678
```



```
12345678
use getline(s1,10) input char: this is a str
this is a
```

其中，代表换行。上面的输入数据将建立如下输入流：



语句 L1 的函数 `get()` 遇到 “`\n`” 将停止从流中提取字符的操作，输入流的指针将停留在 `p1` 位置，下次将从 `p1` 位置开始读取流中的数据。

语句 L2 将从 `p1` 位置开始读取流中的数据，没读够 9 个字符就提前结束了。因为它提前遇到了 `p2` 位置的结束分隔符 “`\n`”，所以停止了读操作。因此，`a` 数组的值是 “12345678\0”，最后的 “\0” 是函数 `get()` 添加的。**注意：**函数 `get()` 不会清除输入流中的分隔符 “`\n`”，当语句 L2 结束时，输入流指针指向 `p2` 位置。

语句 L3 的 `ignore(1)` 将略过输入流中的 1 个字符，执行后，输入流指针将指向 `p3` 位置。

语句 L4 的 `getline(s1, 10)` 将从 `p3` 位置开始读取流中的连续 9 个字符到 `s1` 数组中，因此 `s1` 中的内容将是 “this is a\0”。语句 L4 结束后，输入流指针在 `p4` 位置，但 `p4` 到最后的结束分隔符 “`\n`” 之间的字符已不可用。

说明：`getline()` 与有 3 个参数的成员函数 `get()` 的用法很接近，但函数 `get()` 在遇到结束分隔符时不会清除它，仍将其保留在输入流中。而 `getline()` 在遇到结束分隔符时，要将其从输入流中清除。函数 `get()` 处理结束分隔符的这种方式是程序错误的原因之一。例如，若将程序中的语句 L3 删除，则语句 L4 的函数 `getline()` 不会再接收从键盘输入的任何数据，因为它一开始就读取了输入流中的结束分隔符 “`\n`”，这个结束符是上一次函数 `get()` 遗留下来的。

10.2.2 ostream 的常用成员函数

类 `ostream` 提供了许多用于数据输出的成员函数，并通过对流的输出运算符 `<<` 的重载，实现了对内置数据类型的输出功能。其中常用的成员函数是 `put()`、`write()`、`flush()`，它们在类 `ostream` 中的函数原型如下：

```
class _CRTIMP ostream : virtual public ios {
public:
    ostream& operator<<(...);
    ostream& flush();
    ostream& operator<<(long);
    ostream& put(char c);
    ostream& write(const char *c, int n);
    ostream& seekp(streampos);
    .....
};
```

成员函数 `put()` 插入一个无格式的字节到输出流中，参数 `c` 是要输出的字符。

成员函数 `write()` 是一个无格式的输出成员函数，插入一个字符序列到输出流。参数 `c` 是要输出的字符数组，`n` 表示要输出的字符个数，该函数返回对象的引用。

在输出数据时，C++ 首先将输出数据保存在输出缓冲区中，当输出缓冲区满时，才将数

据送到输出设备。显然，当缓冲区未滿时，数据输出就可能有延迟。成员函数 `flush()` 用于刷新输出流，即不管输出缓冲区滿与不满，它都会立即将缓冲区中的数据送到输出设备。

【例 10-2】 用 `get()` 读取数据，用 `put()` 和 `write()` 输出数据。

```
// Eg10-2.cpp
#include<iostream>
#include<cstring>
using namespace std;
void main() {
    char c;
    char a[50] = "this is a string...";
    cout<<"use get() input char: ";
    while((c = cin.get()) != '\n') // L1, 用 get() 读取字符，遇回车结束
        cout.put(c); // L2, 将 c 中的字符输出
    cout.put('\n'); // L3, 输出一个回车换行符
    cout.put('t').put('h').put('i').put('s').put('\n'); // L4, 输出 this
    cout.write(a, strlen(a)-1).put('\n'); // L5, write() 一次输出多个字符
    cout<<"look"<<"\t here!"<<endl;
}
```

运行程序，其输入和输出结果如下：

```
use get() input char: how are you!
how are you
this
this is a string...
look    here!
```

`put()` 和 `write()` 返回的都是输出流对象的引用，所以允许语句 L4、L5 中的连续书写形式。

10.2.3 数据输入、输出的格式控制

1. 类 `ios` 提供的格式控制标志符

类 `ios` 是 C++ 所有流类的基类，包含 C++ 流的主要特性，其中最重要的三个特性是数据格式化标志、错误状态位标志和文件操作模式。

1) 类 `ios` 的格式化标志符

格式化标志符用于指定数据输入、输出的格式化方式，必须结合格式化设置函数使用。类 `ios` 的格式化标志符如表 10-1 所示。

表 10-1 类 `ios` 的格式化标志符

格式化标志符	作 用	格式化标志符	作 用
<code>ios::skipws</code>	跳过输入流中的空白	<code>ios::showpoint</code>	输出带小数点的数据
<code>ios::left</code>	输出数据按左对齐	<code>ios::uppercase</code>	用大写字母输出十六进制数
<code>ios::right</code>	输出数据按右对齐	<code>ios::showpos</code>	在正数前加 “+”
<code>ios::dec</code>	按十进制输入、输出	<code>ios::scientific</code>	用科学记数法输出浮点数
<code>ios::oct</code>	按八进制输入、输出	<code>ios::fixed</code>	用定点数形式输出浮点数
<code>ios::hex</code>	按十六进制输入、输出	<code>ios::unitbuf</code>	输出完成后立即刷新缓冲区
<code>ios::showbase</code>	在数据前面显示基数（八进制是 0，十六进制是 0x）		

2) 类 ios 的格式化成员函数

类 ios 提供了一些格式化数据的成员函数, 这些成员函数利用上述格式化标志符对输入或输出数据进行格式化。类 ios 及其派生类的对象都可以用它们来设置或取消数据的输入、输出格式。

```
// 设置、取消指定的格式化标志 flags, flags 可以是表 10-1 的格式化标志符
setf(flags)/unsetf(flags)
setf(flags, filed)           // 先清除、然后设置标志
```

3) 设置域宽、精度、填充字符的成员函数

```
ch = fill() / fill(ch)       // 获取当前填充字符/设置填充字符
p = precision() / precision(p) // 获取/设置当前浮点数的精度, p 是一个整型变量, 代表精度位数
w = width() / width(w)       // 获取/设置字段宽度 (以字符个数计算), w 是整型变量, 代表输出宽度
```

说明:

① 成员函数 fill()、precision()、width() 都有无参数和有参数两种形式, 无参数形式用于获取当前输出格式, 有参数形式用于设置数据的输出格式。

② 在用 width(w) 设置了输出域宽之后, 当输出的字符个数小于 w 时, 将用当前的填充字符 (默认填充字符为空白) 填充不足数位, 以达到设置的宽度。用 width 设置的输出宽度只对紧随其后的一个输出数据有效。例如:

```
cout.width(20);           // 设置输出宽度为 20 个字符
cout<<10<<30<<40;
```

第 2 条 cout 语句中的 “10” 将按 20 个字符宽度输出, 而 30 和 40 按默认宽度输出, 实际输出结果是 “[103040]”。10 前面有 18 字符宽度的空白 (输出时没有[])。

【例 10-3】 用类 ios 的成员函数及格式化标志符设置输出数据的格式。

```
// Eg10-3.cpp
#include<iostream>
using namespace std;

void main() {
    char c[30] = "this is string";
    double d = -1234.8976;
    cout.width(30);
    cout.fill('*');
    cout.setf(ios::left);
    cout<<c<<"----L1"<<endl;
    cout.width(30);
    cout.setf(ios::right);
    cout<<c<<"----L2"<<endl;
    cout.width(30);
    cout.setf(ios::internal);
    cout<<d<<"----L3"<<endl;
    cout.setf(ios::dec|ios::showbase|ios::showpoint);
    cout.width(30);
    cout<<d<<"----L4"<<"\n";
    cout.setf(ios::showpoint);
    cout.precision(10);
    cout.width(30);
```

```

cout<<d<<"----L5"<<"\n";
cout.width(30);
cout.setf(ios::oct,ios::basefield);
cout<<100<<"----L6"<<"\n";
}

```

程序运行结果如图 10-2 所示。其中，L1~L5 的输出宽度都被设置为 30，输出结果表明当输出数据的位数不足设置的输出宽度时，将用填充字符填充不足数位。L1 和 L2 是左、右对齐设置的输出情况，L3 是浮点数默认格式的输出情况，L4 是将输出有效位数设置为 10 位的输出情况，L5 是将输出数据精确度设置为 10 位有效数字（未包括小数点和数据的符号位），L6 是将输出数据设置为八进制的输出情况。

```

this is string-----L1
*****this is string---L2
*****-----1234.9---L3
*****-----1234.90---L4
*****-----1234.897600---L5
*****-----0144---L6
Press any key to continue

```

图 10-2 例 10-3 程序运行结果

2. 利用操纵符格式化数据

C++流类库中的每个流对象都维护着一个格式状态值，控制着数据格式化操作的细节，如输出数据的基数（默认为十进制数据）、对齐方式、精度等。除了类 `ios` 中提供的格式化成员函数和格式化标志符，C++还提供了一组可以对数据进行格式化的预定义操纵符。

操纵符（也称为操纵算子）在流对象中的应用方式与数据一样，但不会引起输入、输出数据的操作，而是改变流对象的内部状态值，修改数据的输入、输出格式。

<code>showbase(noshowbase)</code>	// 显示（不显示）数值的基数前缀
<code>showpoint(noshowpoint)</code>	// 显示（不显示）小数点（只有当小数部分存在时才显示小数点）
<code>showpos(noshowpos)</code>	// 在非负数中显示（不显示）+
<code>skipws(noskipws)</code>	// 输入数据时，跳过（不跳过）空白字符
<code>uppercase(nouppercase)</code>	// 十六进制显示（不显示）为 0X (0x)，科学记数法显示 E (e)
<code>dec /oct / hex</code>	// 十进制/八进制/十六进制
<code>left/right</code>	// 设置数据输出对齐方式为：左/右 对齐
<code>fixed</code>	// 以小数形式显示浮点数
<code>scientific</code>	// 用科学记数法显示浮点数
<code>flush</code>	// 刷新输出缓冲区
<code>ends</code>	// 插入空白字符，然后刷新 ostream 缓冲区
<code>endl</code>	// 插入换行字符，然后刷新 ostream 缓冲区
<code>ws</code>	// 跳过开始的空白

C++还提供了下面几个操纵符函数，它们可以直接应用在输出流“<<”中。在应用这些操纵符函数时，必须在程序中包含头文件 `<iomanip>`。

<code>setfill(ch)</code>	// 设置 ch 为填充字符
<code>setprecision(n)</code>	// 设置浮点数的精度为 n 位有效数字
<code>setw(n)</code>	// 设置数据的输出宽度为 n 个字符
<code>setbase(b)</code>	// 将此后的数值型数据的输出基数设置为 b (b=8、10、16) 进制

【例 10-4】 修改例 10-3，用操纵符格式化输出数据，实现同样的功能。

```
// Eg10-4.cpp
```

```

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

void main() {
    char c[30] = "this is string";
    double d = -1234.8976;
    cout<<setw(30)<<left<<setfill('*')<<c<<"----L1"<<endl;
    cout<<setw(30)<<right<<setfill('*')<<c<<"----L2"<<endl;
    cout<<dec<<showbase<<showpoint<<setw(30)<<d<<"----L3"<<"\n";
    cout<<setw(30)<<showpoint<<setprecision(10)<<d<<"----L4"<<"\n";
    cout<<setw(30)<<setbase(8)<<100<<"----L5"<<"\n";
    cout<<100<<"----L6\n";
}

```

执行本程序，将得到与例 10-3 几乎相同的运行结果（除 L6 之外）。

10.3 文件操作

文件是存储在存储介质上（如磁盘、磁带、光盘）的数据集合。按存储格式，文件可以分为文本文件和二进制文件。文本文件在磁盘上存放相关字符的 ASCII 值，所以又称为 ASCII 文件。二进制文件在磁盘上存储相关数据的二进制编码，是把内存中的数据，按其在内存中的存储形式原样写到磁盘上而形成的文件。

文本文件与二进制文件的一个较大区别是对待回车换行符的处理方式。在文本方式下，输入流中的回车符和换行符都会被处理成字符'\n'，输出流中的字符'\n'则会被转换成回车符或换行符。但在二进制方式下，不会进行回车符、换行符与'\n'之间的转换。

10.3.1 文件和流

C++将文件视为一个个字符在磁盘上的有序集合，用流来实现文件的读写操作，用来建立文件流对象的类有 ifstream、ofstream、fstream。它们都是由类 ios 派生的，能够直接访问类 ios 定义的各种操作。其中，ifstream 是输入文件流类，用于建立输入文件流对象，进行文件的读操作；ofstream 是输出文件流类，用于建立输出文件流对象，进行文件的写操作；fstream 则能够建立输入输出文件流对象，可对文件实施读与写的双向操作。

用流操作文件，大致需要经过下述过程。

1. 建立文件流

为了通过流对文件进行操作，应先建立文件流对象，如下所示：

```

ifstream iFile;
ofstream oFile;
fstream ioFile;

```

这里定义了 iFile、oFile、ioFile 三个文件流对象。iFile 是输入文件流对象，能够读取磁盘文件中的数据；oFile 是输出文件流对象，能够将数据写入磁盘文件；ioFile 是输入、输出文件流对象，既能从磁盘文件读取数据，又能将数据写入磁盘文件。

2. 打开文件

文件流对象被创建后，必须与文件关联起来才有意义。每个文件流类都提供了一个成员函数 `open()`，应用它可以将文件流对象与文件关联起来，称为打开文件。

```
void open(const char *filename, int mode, int access);
```

第 1 个参数 `filename` 用于指定关联的文件名，可以包含完整的磁盘路径。第 2 个参数 `mode` 用于指定打开文件的模式，其值是在类 `ios` 中定义的，如表 10-2 所示。第 3 个参数 `access` 用于指定打开文件时的保护方式，该参数的取值由文件缓冲区类 `filebuf` 提供，可以是下面的取值之一：

```
filebuf::openport           // 共享方式
filebuf::sh_none            // 独占方式，不允许共享
filebuf::sh_read            // 允许读共享
filebuf::sh_write           // 允许写共享
```

表 10-2 C++打开文件的模式

文件打开方式	说 明
<code>ios::in</code>	以输入方式打开文件，即读文件（ <code>ifstream</code> 类对象默认方式）
<code>ios::out</code>	以输出方式打开文件，即写文件（ <code>ofstream</code> 类对象默认方式）
<code>ios::app</code>	以添加方式打开文件，新增加的内容添加在文件尾
<code>ios::ate</code>	以添加方式打开文件，新增加的内容添加在文件尾，但下次添加时则添加在当前位置
<code>ios::trunc</code>	文件存在就打开并清除其内容，如不存在，就建立新文件
<code>ios::binary</code>	以二进制方式打开文件（默认为文本文件）
<code>ios::nocreate</code>	打开已有文件，若文件不存在，则打开失败
<code>ios::noreplace</code>	若打开的文件已经存在，则打开失败

在实际应用过程中，可以根据需要将某些方式组合起来使用：

```
ios::in|ios::out           // 以读/写方式打开文件，适用于 fstream 流对象
ios::in|ios::binary        // 以二进制读方式打开文件
ios::out|ios::binary       // 以二进制写方式打开文件
ios::out|ios::in|ios::binary // 以二进制读/写方式打开文件，适用于 fstream 流对象
```

例如，对于上面建立的 `oFile` 流对象，要用二进制写方式打开 `C:\dk` 下的 `a.dat` 文件，命令如下：

```
oFile.open("C:\\dk\\a.dat", ios::in|ios::binary);
```

用 `ioFile` 流对象以文本读/写方式打开当前文件夹下的 `data.txt` 文件的命令如下：

```
ioFile.open("data.txt", ios::in|ios::out);
```

说明：

- ① 如果在文件打开方式中未指明 `ios::binary`，默认以文本方式打开文件。
- ② 文件标识符字符串中的路径间隔符需要用两个“\”表示，第一个反斜杠是转义符，第二个才代表目录间隔符。
- ③ 可以把文件流的定义、文件流与磁盘文件的关联过程合并在一起。例如：

```
ifstream iFile("C:\\dk\\a.dat", ios::in|ios::binary);
fstream ioFile("data.txt", ios::in|ios::out);
```

在定义文件流对象时就将它们与磁盘文件关联在一起，实际上是通过流类的构造函数完

成的，括号中的文件名称和文件读写方式是提供给构造函数的参数。它与先定义文件流对象，再调用流对象的成员函数 `open()` 打开磁盘文件的效果完全相同。

3. 读写文件内容

在定义了文件流对象，并通过它打开磁盘文件后，就可以对文件进行读、写操作了。文件流是从 `ios`、`istream`、`ostream` 及 `iostream` 类继承而来的（见图 10-1），这些流类的输入、输出成员函数同样适用于文件操作。例如，可以用文件流类的输出运算符 `<<` 以及 `put()`、`write()` 等成员函数向文件写入数据；也可以用文件流类的输入运算符 `>>` 以及 `get()`、`getline()`、`read()` 等成员函数读取文件数据。

4. 关闭文件

文件操作结束后，应该调用文件流对象的成员函数 `close()` 关闭文件，如下所示：

```
iFile.close();
oFile.close();
```

关闭文件的实质是断开文件流对象与文件的连接。如果是写文件，在关闭文件时还会将文件缓冲区中的数据写入磁盘。关闭文件后，文件流对象仍然存在，可以用它再次打开其他文件。

本书 2.14 节介绍了用文件流 `ifstream` 和 `ofstream` 处理文本文件的方法，在此不再介绍。

10.3.2 二进制文件

任何文件，无论它包含的是格式化的文本，还是未格式化的原始数据，都可以用文本文件或二进制文件形式打开。文本文件操作的是字符流，二进制文件操作的则是字节流。

之前介绍的都是文本文件，二进制文件的操作过程与它大致相同。但在读文件时，两者判定文件结束标志的方法存在区别。在读文本文件的过程中，当 `get()` 之类的成员函数遇到文件结束符时，返回常量 `EOF` 作为文件结束标志，而二进制文件不能用 `EOF` 作为文件结束的判定值。因为 `EOF` 的值是 `-1`，若文件中某个字节的值为 `-1`，就会被误认为是文件结束符。C++ 提供了一个成员函数 `eof()` 来解决这个问题，它的用法如下：

```
int xx::eof()
```

其中，`xx` 代表输入流对象，到达文件末尾时，返回一个非 0 值，否则返回 0。

C++ 提供了两种操作二进制文件的方式：**一是**用文件流的成员函数 `get()` 和 `put()` 按字节方式读写文件数据；**二是**用文件流的成员函数 `read()` 和 `write()` 按数据块的方式读、写文件数据。

1. 用函数 `get()` 和 `put()` 操作二进制文件

函数 `get()` 和 `put()` 分别定义于 `istream` 和 `ostream` 类，它们在类中的原型如下：

```
istream &get(char &ch);
ostream &put(char ch);
```

由于文件流类是从 `istream`、`ostream`、`iostream` 类继承来的（见图 10-1），这些类的成员函数同样可用于文件流类。因此，在文件操作中可以用 `get()` 从输入文件流中读取字符，用 `put()` 函数将字符插入到输出文件流中。

【例 10-5】 用二进制方式建立一个磁盘文件，将 ASCII 值为 0~90 之间的字符写入文件

C:\dk\a.dat，然后用二进制文件方式读出并在屏幕上显示 a.dat 的内容。

```
// Eg10-5.cpp
#include <iostream>
#include <fstream>
using namespace std;

void main() {
    char ch;
    ofstream out("C:\\dk\\a.dat", ios::out|ios::binary);           // L1
    for(int i = 0; i < 90; i++) {
        if(!(i % 30))
            out.put('\n');                                         // L2
        out.put(char(i));                                         // L3
        out.put(' ');                                             // L4
    }
    out.close();                                                  // L5
    ifstream in("C:\\dk\\a.dat", ios::in|ios::binary);           // L6
    while(in.get(ch))
        cout<<ch;                                                // L7
    in.close();                                                   // L8
}
```

语句 L1~L4 建立文件 C:\dk\a.dat。语句 L2 每隔 30 个字符就向文件中插入一个换行符，目的是便于语句 L7 每行最多输出 30 个字符。

语句 L5 关闭了以写方式打开的 a.dat 文件，便于语句 L6 以读方式打开该文件。语句 L7 从文件流中一次读出一个字符，并将它显示出来。文件操作完成后，通过语句 L8 将其关闭。

程序执行时，先建立磁盘文件 a.dat，再将文件中的数据读出并显示，如图 10-3 所示。



图 10-3 ASCII 值为 0~90 的字符

2. 用函数 read()和 write()操作二进制文件

成员函数 read()用于读取输入流中的数据块，成员函数 write()用于向输出流中写入数据块，它们分别定义于 istream 和 ostream 类中，在相关类中的原型如下：

```
istream& read(char *buf, int n);
ostream& write(const char *buf, int n);
```

read()一次从输入流中读取 n 字节的内容放入输入缓冲区域 buf 中，write()一次插入 n 字节到输出流，即一次写入 n 字节内容到文件。

【例 10-6】 设计一个 person 类，从键盘输入每个人的姓名、身份证号、年龄、地址等数据，并将每个人的信息保存在 C:\dk 下的二进制文件 person.dat 中，然后将文件中的个人信息读出来，保存在 vector 类型的向量中并显示出来。

```
// Eg10-6.cpp
#include <iostream>
#include <vector>
```



```

#include <string>
#include <fstream>
using namespace std;

class Person {
private:
    char name[20];
    char id[18];
    int age;
    char addr[20];
public:
    Person() {}
    Person(char* n, char* PerId, int Age, char* Address) {
        strcpy(name, n);
        strcpy(id, PerId);
        strcpy(addr, Address);
        age = Age;
    }
    void display() {
        cout<<name<<"\t"<<id<<"\t"<<age<<"\t"<<addr<<endl;
    }
};

void main() {
    vector<Person> p; // L1
    vector<Person>::iterator pos; // L2
    char ch;
    ofstream out("C:\\dk\\person.dat", ios::out|ios::app|ios::binary); // L3
    char Name[20], ID[18], Addr[20];
    int Age;
    cout<<"-----输入个人档案-----"<<endl<<endl;
    do{ // L4
        cout<<"姓名: "; cin>>Name;
        cout<<"身份证号: "; cin>>ID;
        cout<<"年龄: "; cin>>Age;
        cout<<"地址: "; cin>>Addr;
        Person s1(Name, ID, Age, Addr); // L5
        out.write((char*)&s1, sizeof(s1)); // L6
        cout<<"Enter another person (y/n)?";
        cin>>ch;
    } while(ch == 'y'); // L7
    out.close(); // L8
    ifstream in("C:\\dk\\person.dat", ios::in|ios::binary); // L9
    Person s1;
    in.read((char*)&s1, sizeof(s1)); // L10
    while(!in.eof()) { // L11
        p.push_back(s1); // L12
        in.read((char*)&s1, sizeof(s1)); // L13
    } // L14
    cout<<"\n-----从文件中读出的数据-----"<<endl<<endl; // L15
    pos = p.begin(); // L16
}

```

```

        for(pos = p.begin(); pos != p.end(); pos++)                // L17
            (*pos).display();                                     // L18
    }

```

语句 L1 定义了 `Person` 类的一个向量 `p`，用于保存 `Person` 类的对象。语句 L2 定义 `Person` 类型向量的迭代器 `pos`，通过它可以遍历 `Person` 类型的向量。语句 L3 定义了一个输出文件流对象 `out`，它以二进制写方式打开数据文件 `C:\dk\person.dat`，如果该文件不存在，就建立它，如果存在，就打开且在该文件末尾添加数据。

语句 L4~L7 是一段循环程序，用于从键盘输入每个人的数据。每输入一个人的数据，通过语句 L5 调用 `Person` 的构造函数创建一个对象 `s1`，再通过语句 L6 调用输出流的成员函数 `write()`，将对象 `s1` 的数据块写入磁盘文件 `person.dat` 中。语句 L6 的形式如下：

```
out.write((char*)&s1, sizeof(s1));
```

从对象 `s1` 对应内存区域的数据块中读出 `sizeof(s1)` 字节放到输出流中（写入 `Person.dat` 文件），`sizeof(s1)` 字节是作为一个整体写入文件的，所以当从该文件读出数据时，也应该把这些字节作为一个整体读出才有意义，即一次从文件中读取 `sizeof(s1)` 字节。

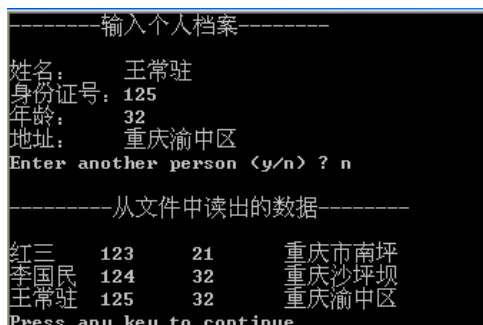
在完成写文件的操作后，语句 L8 关闭了文件，以便文件被其他程序使用。语句 L9 定义了输入文件流对象 `in`，并以二进制读方式打开前面建立的 `person.dat` 文件。

语句 L10~L14 从 `person.dat` 文件中读取每个 `Person` 对象的数据，读出的数据被放在对象 `s1` 中（见语句 L10、L13），并将该对象插入到向量 `p` 的末尾。读出数据的语句如下：

```
in.read((char*)&s1, sizeof(s1));
```

从输入流中读取 `sizeof(s1)` 字节，并用读出的数据块初始化 `s1` 对象的对应数据成员。每次读出的数据块与建立文件时写入的数据块大小相同。

语句 L16~L17 通过向量迭代器 `pos` 遍历向量 `p`，并通过迭代器，将每个 `Person` 对象的数据成员显示出来。程序执行的结果如图 10-4 所示。



```

-----输入个人档案-----
姓名:      王常驻
身份证号:  125
年龄:      32
地址:      重庆渝中区
Enter another person (y/n) ? n
-----从文件中读出的数据-----
红三      123      21      重庆市南坪
李国民    124      32      重庆沙坪坝
王常驻    125      32      重庆渝中区
Press any key to continue

```

图 10-4 例 10-6 程序运行结果

10.3.3 随机文件

文件具有顺序访问和随机访问两种方式。[顺序访问](#)是指按照从前到后的顺序依次对文件进行读写操作，有些存储设备只支持顺序访问，如磁带。[随机访问](#)也称为直接访问，可以按任意次序对文件进行读写操作。支持顺序访问的文件称为顺序文件。支持随机访问的文件称为随机文件。

随机访问利用 C++ 流类提供的指针操作函数在文件中移动指针，指向要读写的字节位置，

然后从该位置读取或写入指定字节的数据块，这样就实现了文件数据的随机访问。

由于随机访问不需要从文件开始位置顺序读写前面的数据，可以直接把文件指针定位到指定位置并进行文件数据的读、写操作，因此能够快捷地查询、修改、删除文件中的数据。

类 `istream` 定义了 3 个操作输入文件读指针的成员函数，它们的函数原型如下：

```
istream& seekg(long pos);           // ①
istream& seekg(long off, dir);      // ②
long tellg();                       // 返回文件读指针的当前位置
```

`seekg()` 有两种用法。第①种是绝对定位形式，直接将文件读指针定位到距离文件开始位置的第 `pos` 字节处。第②种是相对定位形式，将文件读指针定位到偏移 `dir` 位置 `off` 字节的位置，`dir` 可取下面的值：`ios::cur`，当前文件指针位置；`ios::beg`，文件开始位置；`ios::end`，文件结束位置。

类 `ostream` 也定义了 3 个操作输出文件写指针的成员函数，它们的函数原型如下：

```
ostream& seekp(long pos);
ostream& seekp(long off, dir);
long tellp();                       // 返回文件写指针的当前位置
```

这些函数的参数及用法与前面 3 个函数的基本相同，只不过它们用于写文件操作，而前面 3 个函数用于读文件操作。

【例 10-7】 某雇员类 `Employee` 有编号、姓名、年龄、工资等数据成员。设计一个随机文件保存各雇员的各项数据。

```
// Eg10-7.cpp
#include <iostream>
#include <string>
#include <fstream>
using namespace std;

class Employee {
private:
    int number, age;
    char name[20];
    double sal;
public:
    Employee() { }
    Employee(int num, char* Name, int Age, double Salary) {
        number = num;
        strcpy(name, Name);
        age = Age;
        sal = Salary;
    }
    void display() {
        cout<<number<<"\t"<<name<<"\t"<<age<<"\t"<<sal<<endl;
    }
};

void main(){
    ofstream out("Employee.dat", ios::out|ios::binary); // 定义随机输出文件
    Employee e1(1, "张三", 23, 2320);
```

```

Employee e2(2, "李四", 32, 3210);
Employee e3(3, "王五", 34, 2220);
Employee e4(4, "刘六", 27, 1220);
out.write((char*)&e1, sizeof(e1));           // 按 e1、e2、e3、e4 顺序写入文件
out.write((char*)&e2, sizeof(e2));
out.write((char*)&e3, sizeof(e3));
out.write((char*)&e4, sizeof(e4));

// 如下代码将 e3 (王五) 的年龄改为 40 岁
Employee e5(3, "王五", 40, 2220);
out.seekp(2*sizeof(e1));                     // 将文件指针定位到第 3 (起始为 0) 个数据块
out.write((char*)&e5, sizeof(e5));           // 将 e5 写到第 3 个数据块位置, 覆盖 e3
out.close();                                 // 关闭文件

ifstream in("Employee.dat", ios::in|ios::binary); // 建立二进制输入文件
Employee s1;                                // s1 用于保存从文件中读出的数据
cout<<"\n-----从文件中读出第 3 个人的数据-----\n\n";
in.seekg(2*(sizeof(s1)), ios::beg);          // 移动读文件指针, 定位到第 3 个数据块
in.read((char*)&s1, sizeof(s1));             // 读出第 3 个雇员的数据块
s1.display();
cout<<"\n-----从文件中读出全部的数据-----\n\n";
in.seekg(0,ios::beg);                        // 移动文件指针, 指向文件开头
in.read((char*) &s1, sizeof(s1));            // 读出第 1 个数据块
while(!in.eof()) {                           // 如果没有读完文件, 就继续读
    s1.display();                             // 显示读出的雇员数据
    in.read((char*)&s1, sizeof(s1));          // 读当前文件指针处的数据
}
}

```

程序运行结果如下:

```

-----从文件中读出第 3 个人的数据-----
3      王五      40      2220           // 可见王五的年龄已被修改
-----从文件中读出全部的数据-----
1      张三      23      2320
2      李四      32      3210
3      王五      40      2220
4      刘六      27      1220

```

本程序应用 `seekp()` 移动输出文件的写指针, 修改文件中指定位置的数据; 应用 `seekg()` 移动输入文件的读指针, 读取指定文件的指定位置的数据。请读者结合程序代码中的注释分析, 理解随机读、写二进制文件的方法。

习 题 10

- 10.1 C++ 预定义了哪几个输入、输出流对象? 简述其作用。
- 10.2 什么是顺序文件和随机文件? 简述在 C++ 程序中建立文件的过程。
- 10.3 读程序, 写出程序运行结果。

(1)

```
#include <iostream>
```

```

#include <fstream>
using namespace std;

void main(){
    fstream ou, in;
    ou.open("a.dat", ios::out);
    ou<<"on fact\n";
    ou<<"operating file \n";
    ou<<"is the same as inputing/outputing data on screen ...\n";
    ou.close();
    char buffer[80];
    in.open("a.dat", ios::in);
    while(!in.eof()) {
        in.getline(buffer, 80);
        cout<<buffer<<endl;
    }
}

```

(2)

```

#include <iostream>
#include <string>
#include <fstream>
using namespace std;

class Worker {
private:
    int number, age;
    char name[20];
    double sal;
public:
    Worker(){ }
    Worker(int num, const char* Name, int Age, double Salary):number(num), age(Age), sal(Salary){
        strcpy(name, Name);
    }
    void display() {
        cout<<number<<"\t"<<name<<"\t"<<age<<"\t"<<sal<<endl;
    }
};

void main(){
    ofstream out("Worker.dat", ios::out|ios::binary);
    Worker man[]={Worker(1,"张三",23,2320), Worker(2,"李四",32,2321),
        Worker(3,"王五",34,2322), Worker(4,"刘六",27,2324),
        Worker(5,"晓红",23,2325), Worker(6,"黄明",50,2326)};
    for(int i = 0; i < 6; i++)
        out.write((char*)&man[i], sizeof(man[i]));
    out.close();
    Worker s1;
    ifstream in("Worker.dat", ios::in|ios::binary);
    in.seekg(2*(sizeof(s1)), ios::beg);
    in.read((char*)&s1, sizeof(s1));
}

```

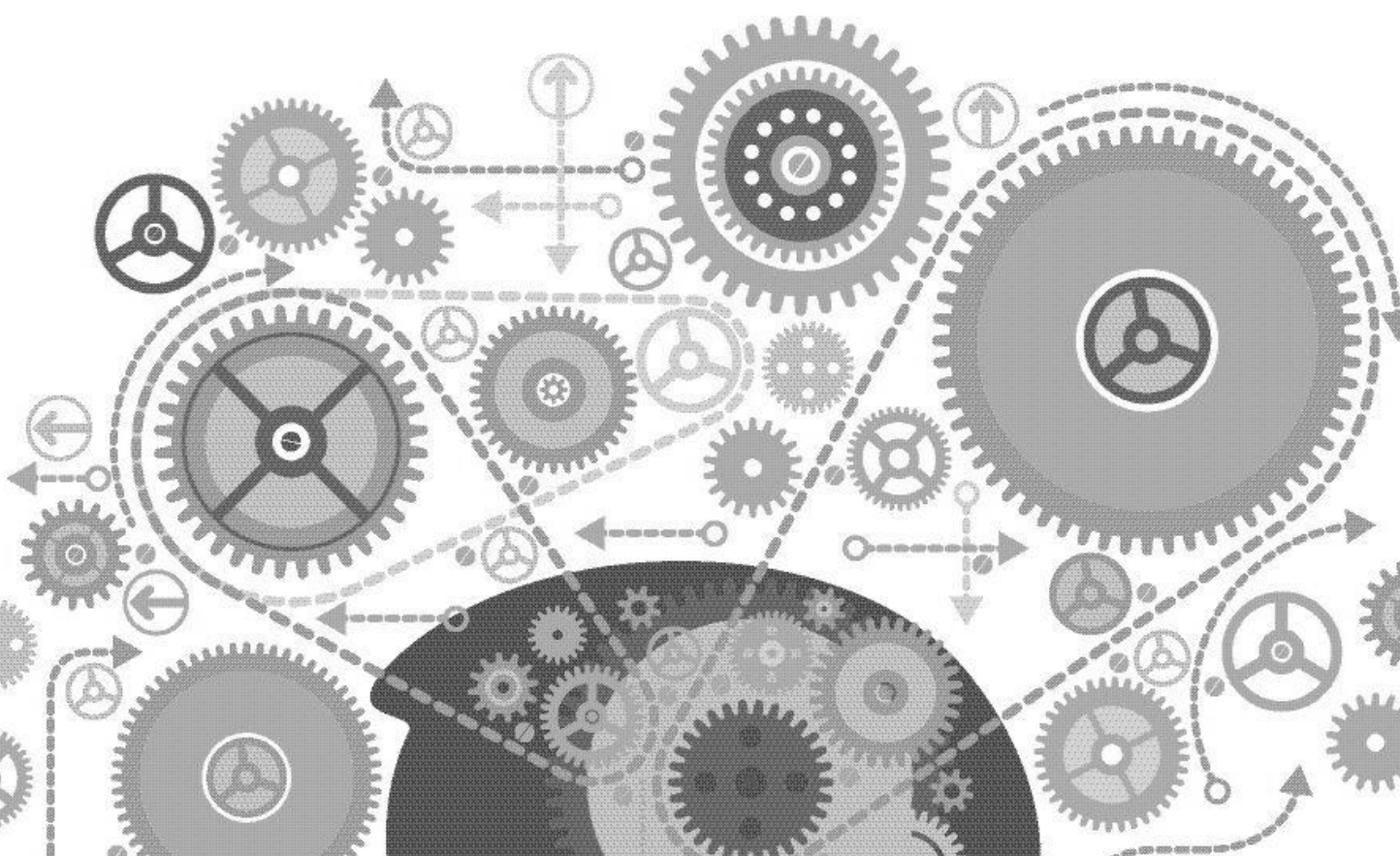
```
s1.display();
in.seekg(0, ios::beg);
in.read((char*)&s1, sizeof(s1));
s1.display();
in.close();
}
```

10.4 编写程序，将文件 A1.dat 的内容复制到文件 A2.dat 中，并在屏幕上显示文件内容。

10.5 设计一个建立同学通讯录文件的程序，文件中的每条记录包括各同学的姓名、学校、专业、班级、电话号码、通信地址、邮政编码等数据。

10.6 图书类 Book，包括书名、出版社名称、作者姓名、图书定价等数据成员。编程序完成 Book 类的设计，从键盘读入 10 本图书的各项数据，并将这 10 本图书的相关数据写入磁盘文件 book.dat，然后从 book.dat 中读出图书数据，计算所有图书的总价，显示每本图书的详细信息，每本图书的信息显示在一行上。

参考文献



- [1] [德]PERTER G. 深度探索 C++ 14. 吴野, 译. 北京: 电子工业出版社, 2020.
- [2] 张运龙. C++服务器开发精髓. 北京: 电子工业出版社, 2021.
- [3] [美]BRUCE E. Java 编程思想. 陈昊鹏, 译. 北京: 机械工业出版社, 2011.
- [4] [美]B LIPPMAN 等. C++ Primer. 王刚, 杨巨峰, 译. 北京: 电子工业出版社, 2013.
- [5] 苏小红, 王宇颖, 孙志岗. C 语言程序设计. 3 版. 北京: 高等教育出版社, 2020.
- [6] [美]WALTER S. C++入门经典. 9 版. 周靖, 译. 北京: 清华大学出版社, 2015.
- [7] 谭浩强. C++面向对象程序设计. 2 版. 北京: 清华大学出版社, 2014.
- [8] 陈维兴, 林小茶. C++面向对象程序设计教程. 北京: 清华大学出版社, 2000.
- [9] 陈志泊, 王春玲. 面向对象的程序设计语言——C++. 北京: 人民邮电出版社, 2002.
- [10] 周霁如, 林伟健. C++程序设计基础. 北京: 电子工业出版社, 2006.
- [11] 钱能. C++程序设计教程. 北京: 清华大学出版社, 1994.
- [12] 郑莉, 董渊. C++语言程序设计. 2 版. 北京: 清华大学出版社, 2001.
- [13] 杨庚, 王汝传. 面向对象程序设计与 C++语言. 北京: 人民邮电出版社, 2002.
- [14] 吕凤翥. C++语言程序设计. 北京: 电子工业出版社, 2004.
- [15] 甘玲, 邱劲. 面向对象技术与 Visual C++. 北京: 清华大学出版社, 2004.
- [16] 梁普选. C++程序设计与软件技术基础. 北京: 电子工业出版社, 2004.
- [17] 钱能. C++程序设计教程. 2 版. 北京: 清华大学出版社, 2005.
- [18] 田志良. 面向对象程序设计循序渐近. 北京: 学苑出版社, 1994.
- [19] 蔡明志. Borland C++ 4.0 使用与编程指南. 北京: 清华大学出版社, 1994.
- [20] 侯俊杰. 深入浅出 MFC. 2 版. 武汉: 华中科技大学出版社, 2001.
- [21] DAVID V, NICOLAI M.J. C++ Templates 中文版. 陈伟柱, 译. 北京: 人民邮电出版社, 2004.
- [22] NICOLAI M.J. C++标准程序库. 侯捷, 孟岩, 译. 武汉: 华中科技大学出版社, 2002.
- [23] [美]HERBERT S. C++参考大全. 4 版. 周志荣, 朱德芳等, 译. 北京: 电子工业出版社, 2003.
- [24] [美]H.M. DEITEL, P.J. DEITEL. C++程序设计教程. 4 版. 施平安, 译. 北京: 清华大学出版社, 2004.
- [25] [美]ERIC N. C++大学教程. 3 版. 侯普秀, 曹振新, 译. 北京: 清华大学出版社, 2005.
- [26] [美] H.M. DEITEL, P.J. DEITEL. C++大学教程. 2 版. 邱仲潘, 等译. 北京: 电子工业出版社, 2001.
- [27] [加]GORAN S. 面向对象编程: 工程和技术人员的 C++语言. 马海军, 段晓勇等, 译. 北京: 清华大学出版社, 2003.
- [28] [美]HERBERT S. C++基础教程. 张林娣, 译. 北京: 清华大学出版社, 2002.
- [29] [美]BJARNE S. C++程序设计语言. 4 版. 裘宗燕, 译. 北京: 机械工业出版社, 2002.
- [30] [美]STANLEY B.L., [加]JOSÉE L. C++ Primer. 3 版. 潘爱民, 张丽, 译. 北京: 中国电力出版社, 2002.
- [31] [美]STEPHEN C.D. C++程序设计陷阱. 陈君等, 译. 北京: 中国青年出版社, 2003.
- [32] [美]TIMOTHY A.B. 面向对象编程导论. 3 版. 黄明军, 李桂杰, 译. 北京: 机械工业出版社, 2003.