

# 目录

<b>第1章 概述.....7</b>	<b>2.6 练 习 题..... 34</b>
1.1 C++语言发展历史.....7	<b>第3章 运算符与表达式..... 36</b>
1.2 一个简单的C++程序.....8	3.1 基本运算符..... 36
1.3 编程的基本要求.....10	3.1.1 算术运算符..... 38
1.4 C++程序的开发步骤.....10	3.1.2 关系运算符..... 39
1.5 VC++开发环境简介.....11	3.1.3 逻辑运算符..... 40
1.6 C++标准及开发工具.....15	3.1.4 位运算符..... 41
1.7 本书组织结构.....16	3.1.5 条件运算符..... 44
1.8 练 习 题.....18	3.1.6 赋值运算符..... 45
<b>第2章 数据类型与变量.....19</b>	3.1.7 逗号运算符..... 45
2.1 关键字和标识符.....19	3.1.8 自增、自减运算符..... 46
2.1.1 关键字.....19	3.1.9 sizeof 运算符..... 47
2.1.2 标识符.....21	3.1.10 typeid 运算符..... 48
2.1.3 标点符号.....21	3.2 表达式..... 49
2.1.4 分隔符.....21	3.2.1 左值表达式和右值表达式..... 49
2.2 基本数据类型.....22	3.2.2 算术表达式..... 49
2.2.1 布尔型.....23	3.2.3 赋值表达式..... 50
2.2.2 字符型.....23	3.2.4 关系表达式..... 50
2.2.3 整型.....24	3.2.5 逻辑表达式..... 50
2.2.4 浮点型.....25	3.2.6 逗号表达式..... 51
2.2.5 空类型void.....25	3.2.7 表达式语句..... 52
2.3 字面值.....25	3.3 类型转换..... 52
2.3.1 逻辑值.....25	3.3.1 自动类型转换..... 52
2.3.2 整型值.....26	3.3.2 赋值转换..... 54
2.3.3 浮点值.....26	3.3.3 强制类型转换..... 55
2.3.4 字符值.....27	3.4 小 结..... 56
2.3.5 字符串值.....28	3.5 练 习 题..... 56
2.4 变量.....28	<b>第4章 基本语句..... 60</b>
2.4.1 变量的说明.....29	4.1 语句分类..... 60
2.4.2 变量的初始化.....29	4.1.1 说明语句..... 60
2.4.3 auto 初始化.....30	4.1.2 表达式语句..... 60
2.4.4 变量的赋值.....30	4.1.3 选择语句..... 61
2.4.5 变量的输入输出.....30	4.1.4 循环语句..... 61
2.4.6 命名常量.....33	4.1.5 跳转语句..... 61
2.5 小 结.....33	4.1.6 空语句..... 61

4.1.7 复合语句.....	61	5.4.2 重载函数的调用.....	95
4.1.8 异常处理语句.....	61	5.5 嵌套调用和递归调用 .....	96
4.1.9 标号语句.....	62	5.5.1 函数的嵌套调用.....	96
4.2 程序的基本结构 .....	62	5.5.2 函数的递归调用.....	97
4.2.1 顺序结构.....	62	5.6 函数原型 .....	103
4.2.2 选择结构.....	63	5.7 AUTO 函数.....	104
4.2.3 循环结构.....	63	5.8 特殊参数 .....	105
4.3 选择语句.....	64	5.8.1 带缺省值的形参.....	105
4.3.1 条件语句.....	64	5.8.2 可变参数.....	107
4.3.2 开关语句.....	68	5.9 内联函数 INLINE.....	109
4.4 循环语句.....	71	5.10 作用域 .....	109
4.4.1 while 语句.....	71	5.10.1 块作用域.....	109
4.4.2 do-while 语句.....	73	5.10.2 文件作用域.....	111
4.4.3 for 语句.....	74	5.10.3 函数原型作用域.....	112
4.4.4 循环语句的比较.....	75	5.10.4 函数作用域.....	112
4.4.5 循环的嵌套 .....	76	5.11 程序运行期的存储区域.....	112
4.5 跳转语句.....	77	5.11.1 静态存储区.....	113
4.5.1 break 语句.....	77	5.11.2 全局存储区.....	113
4.5.2 continue 语句.....	78	5.11.3 动态存储区.....	113
4.5.3 goto 语句与标号语句 .....	79	5.12 存储类 .....	113
4.6 例子.....	79	5.12.1 auto 变量.....	113
4.7 小 结.....	83	5.12.2 register 变量 .....	114
4.8 练 习 题.....	84	5.12.3 static 变量与多文件项目 .....	114
<b>第 5 章 函数和编译预处理.....</b>	<b>87</b>	5.12.4 extern 变量.....	116
5.1 函数的基本概念 .....	87	5.12.5 存储类小结.....	118
5.1.1 库函数和用户定义函数.....	87	5.13 编译预处理 .....	118
5.1.2 无参函数和有参函数.....	87	5.13.1 包含文件.....	119
5.1.3 有返回函数和无返回函数.....	88	5.13.2 无参宏.....	120
5.2 函数的定义 .....	88	5.13.3 有参宏.....	123
5.2.1 无参函数的定义.....	89	5.13.4 条件编译.....	125
5.2.2 有参函数的定义.....	89	5.13.5 条件编译示例.....	129
5.2.3 函数定义的要点.....	90	5.13.6 其它预处理指令.....	130
5.3 函数的调用 .....	91	5.14 小 结.....	130
5.3.1 函数调用的形式.....	91	5.15 练 习 题.....	132
5.3.2 函数调用的方式.....	91	<b>第 6 章 数组.....</b>	<b>138</b>
5.3.3 函数调用的要点.....	93	6.1 一维数组 .....	138
5.4 函数的重载.....	94	6.1.1 一维数组的定义.....	138
5.4.1 重载函数的定义.....	94	6.1.2 一维数组的初始化.....	139

6.1.3 一维数组的使用.....	140	7.4.1 共同体类型的定义.....	191
6.1.4 基于范围的for 语句.....	141	7.4.2 共同体变量的说明及使用.....	191
6.1.5 一维数组的应用.....	141	7.5 类型定义 TYPEDEF.....	193
6.1.6 调用算法来简化编程.....	148	7.6 小 结.....	194
6.2 二维数组.....	150	7.7 练 习 题.....	195
6.2.1 二维数组的定义.....	150	<b>第 8 章 指针和引用.....</b>	<b>198</b>
6.2.2 二维数组的初始化.....	151	8.1 指针及指针变量.....	198
6.2.3 二维数组的应用.....	151	8.1.1 地址的概念.....	198
6.3 数组与函数.....	154	8.1.2 指针变量.....	200
6.4 字符数组与字符串.....	157	8.1.3 指针的运算.....	201
6.4.1 字符数组的定义.....	157	8.1.4 关键字 nullptr.....	205
6.4.2 字符数组的初始化.....	158	8.2 指针与结构.....	205
6.4.3 字符数组的输入输出.....	160	8.2.1 结构的指针.....	205
6.4.4 字符数组的操作.....	162	8.2.2 指针作为结构成员.....	207
6.5 字符串处理函数.....	162	8.3 指针与数组.....	209
6.5.1 字符串处理函数.....	162	8.3.1 用指针访问数组.....	209
6.5.2 字符数组的应用.....	165	8.3.2 指针与字符串.....	214
6.6 小 结.....	167	8.3.3 指针的数组.....	216
6.7 练 习 题.....	167	8.4 指针与函数.....	224
<b>第 7 章 结构、枚举、共同体.....</b>	<b>171</b>	8.4.1 指针作为形参.....	224
7.1 结构.....	171	8.4.2 函数返回指针.....	225
7.1.1 结构类型的定义.....	171	8.4.3 函数的指针.....	226
7.1.2 定义结构变量.....	174	8.5 VOID 指针与 CONST 指针.....	231
7.1.3 结构变量的初始化.....	175	8.5.1 void 指针.....	231
7.1.4 结构变量的使用.....	175	8.5.2 const 指针.....	233
7.1.5 结构的数组.....	177	8.6 动态使用内存.....	233
7.1.6 结构中的静态成员.....	180	8.6.1 new 和 delete 运算符.....	234
7.1.7 结构的嵌套定义.....	182	8.6.2 动态变量管理.....	238
7.1.8 C++ 结构中的构造函数与析构函数.....	182	8.6.3 注意要点.....	240
7.2 位域.....	184	8.7 引用.....	241
7.2.1 位域的定义.....	184	8.7.1 左值引用变量.....	241
7.2.2 位域的使用.....	185	8.7.2 左值引用与数组、指针的关系..	242
7.3 枚举.....	187	8.7.3 左值引用与函数.....	243
7.3.1 枚举类型及枚举变量.....	187	8.7.4 右值引用&&.....	246
7.3.2 枚举的使用.....	188	8.8 LAMBDA 表达式.....	248
7.3.3 C11 强类型枚举.....	190	8.8.1 语法构造.....	248
7.4 共同体.....	191	8.8.2 简单用法.....	250
		8.8.3 嵌套 L 式与高阶函数.....	251

8.8.4 调用 STL 算法.....	251	10.4.2 复合对象设计实例.....	313
8.9 单向链表及其应用*.....	254	10.4.3 复合对象设计要点.....	316
8.9.1 链表概述.....	254	10.5 对象数组.....	317
8.9.2 建立无序链表、遍历和撤销.....	257	10.5.1 定义和使用.....	318
8.9.3 添加、查找与删除元素.....	259	10.5.2 对象数组作为成员.....	319
8.9.4 建立有序链表.....	263	10.6 静态成员.....	321
8.10 小结.....	264	10.6.1 静态数据成员.....	321
8.11 练习题.....	266	10.6.2 静态成员函数.....	323
<b>第 9 章 类和对象.....</b>	<b>272</b>	10.7 CONST 和 VOLATILE 修饰符.....	324
9.1 类.....	272	10.7.1 修饰符 const.....	324
9.1.1 类的定义.....	272	10.7.2 修饰符 volatile*.....	325
9.1.2 类成员的可见性.....	275	10.8 类成员的指针*.....	326
9.1.3 类的数据成员.....	276	10.8.1 数据成员的指针.....	326
9.1.4 类的成员函数.....	278	10.8.2 成员函数的指针.....	327
9.1.5 类与结构的区别.....	281	10.9 小结.....	330
9.2 对象.....	282	10.10 练习题.....	332
9.2.1 对象的创建.....	282	<b>第 11 章 类的继承.....</b>	<b>336</b>
9.2.2 访问对象的成员.....	283	11.1 继承与派生.....	336
9.2.3 类与对象的关系.....	285	11.1.1 基类与派生类.....	336
9.3 THIS 指针.....	285	11.1.2 派生类的定义与构成.....	337
9.4 类中还有什么?.....	286	11.1.3 继承方式与访问控制.....	338
9.5 小结.....	287	11.2 派生类的构造和析构.....	340
9.6 练习题.....	288	11.2.1 派生类的构造函数.....	340
<b>第 10 章 构造函数与析构函数.....</b>	<b>290</b>	11.2.2 派生类继承构造函数.....	342
10.1 构造函数.....	290	11.2.3 派生类的析构函数.....	343
10.1.1 构造函数的定义.....	290	11.3 二义性问题.....	346
10.1.2 缺省构造函数.....	292	11.3.1 多继承造成的二义性.....	346
10.1.3 委托构造函数.....	292	11.3.2 支配规则.....	347
10.2 析构函数.....	293	11.4 虚基类.....	350
10.3 拷贝构造函数与单参构造函数.....	295	11.4.1 共同基类造成的二义性.....	350
10.3.1 拷贝构造函数.....	295	11.4.2 虚基类的说明.....	351
10.3.2 赋值操作函数.....	298	11.4.3 虚基类的例子.....	353
10.3.3 用 string 替代 char*.....	302	11.5 子类型关系.....	355
10.3.4 单参构造函数.....	304	11.6 虚函数.....	358
10.3.5 移动语义.....	306	11.6.1 定义和使用.....	358
10.4 复合对象与成员对象.....	310	11.6.2 成员函数中调用虚函数.....	359
10.4.1 构造过程.....	310	11.6.3 构造函数中调用虚函数.....	360
		11.6.4 虚析构函数.....	361

11.6.5 纯虚函数与抽象类.....	362	13.3.2 模板实例化.....	412
11.6.6 关键字 <i>final</i> .....	364	13.3.3 类模板的继承.....	414
11.6.7 显式虚函数改写 <i>override</i> .....	364	13.3.4 可变参量的类模板.....	416
11.6.8 抽象类设计实例.....	364	13.4 标准模板库 STL .....	417
11.7 继承性设计要点 .....	371	13.4.1 容器.....	417
11.8 小结 .....	372	13.4.2 迭代器.....	419
11.9 练习题.....	376	13.4.3 容器的共同成员类型和操作....	421
<b>第 12 章 运算符重载.....</b>	<b>381</b>	13.4.4 算法< <i>algorithm</i> >.....	422
12.1 一般运算符重载 .....	381	13.4.5 基于 C11 简化编程.....	424
12.1.1 运算符重载函数.....	381	13.4.6 函数对象.....	426
12.1.2 双目运算符的重载.....	382	13.4.7 <i>vector</i> 、 <i>deque</i> 和 <i>list</i> .....	427
12.1.3 单目运算符的重载.....	383	13.4.8 <i>set</i> 和 <i>multiset</i> .....	434
12.2 用友元函数实现运算符重载 .....	385	13.4.9 <i>map</i> 和 <i>multimap</i> .....	436
12.2.1 友元函数 .....	385	13.4.10 <i>stack</i> 、 <i>queue</i> 和 <i>priority_queue</i> .....	443
12.2.2 定义重载函数.....	386	13.5 小结.....	445
12.2.3 用户自定义字面值 UDL.....	389	13.6 练习题.....	446
12.3 特殊运算符的重载 .....	391	<b>第 14 章 输入输出流.....</b>	<b>448</b>
12.3.1 赋值操作函数.....	391	14.1 概述 .....	448
12.3.2 类型转换函数.....	391	14.1.1 流 <i>Stream</i> .....	448
12.3.3 下标运算符 .....	393	14.1.2 文件.....	449
12.3.4 函数调用运算符.....	393	14.1.3 缓冲.....	449
12.4 小结 .....	395	14.2 基本流类 .....	449
12.5 练习题.....	396	14.2.1 基本流类体系.....	450
<b>第 13 章 模板.....</b>	<b>399</b>	14.2.2 预定义标准对象.....	451
13.1 模板的概念 .....	399	14.2.3 流的格式控制.....	452
13.2 函数模板 .....	400	14.2.4 流的错误处理.....	454
13.2.1 定义 .....	400	14.3 标准输入/输出 .....	455
13.2.2 模板实例化 .....	401	14.3.1 <i>cin</i> 输入要点.....	456
13.2.3 函数模板与有参宏的区别....	402	14.3.2 输入操作的成员函数.....	457
13.2.4 模板函数的重载.....	402	14.3.3 <i>cout</i> 输出要点.....	458
13.2.5 可变参量的函数模板.....	404	14.3.4 输出操作的成员函数.....	459
13.2.6 函数模板例子.....	405	14.3.5 重载提取和插入运算符.....	459
13.2.7 完美转发 .....	406	14.4 文件流 .....	461
13.2.8 自动类型推导 <i>decltype</i> .....	408	14.4.1 文件概述.....	461
13.2.9 引用塌缩规则.....	411	14.4.2 文件处理的一般过程.....	462
13.3 类模板 .....	411	14.4.3 文件的打开与关闭.....	462
13.3.1 定义 .....	411	14.4.4 文本文件的使用.....	465
		14.4.5 二进制文件的使用.....	467

14.4.6 文件的随机访问.....	470	16.4 修饰符 EXPLICIT .....	510
14.5 小 结 .....	472	16.5 修饰符 MUTABLE .....	511
14.6 练 习 题.....	473	16.6 RTTI 与 TYPEID .....	512
<b>第 15 章 异常.....</b>	<b>475</b>	16.7 强制类型转换 .....	514
15.1 异常的概念 .....	475	16.7.1 static_cast 运算符.....	515
15.2 异常类型的架构 .....	477	16.7.2 dynamic_cast 运算符.....	516
15.3 异常处理语句 .....	479	16.7.3 const_cast 运算符.....	517
15.3.1 throw 语句.....	479	16.7.4 reinterpret_cast 运算符 .....	518
15.3.2 try-catch 语句.....	481	16.8 小 结.....	519
15.3.3 异常处理的例子.....	483	<b>附录 A ASCII 码表 .....</b>	<b>520</b>
15.3.4 C11 修饰符 noexcept.....	488	<b>ASCII 控制字符 .....</b>	<b>521</b>
15.4 终止处理器 .....	489	<b>附录 B 常用库函数 .....</b>	<b>522</b>
15.5 扩展新的异常类型 .....	490	表 B-1 运行期库的功能分类.....	522
15.6 异常类型的应用 .....	492	表 B-2 运行期库头文件.....	523
15.7 函数设计中的异常处理 .....	494	表 B-3 标准 C++ 库头文件.....	524
15.8 异常可能导致内存泄露 .....	497	表 B-5 数学函数<math.h>.....	525
15.9 小 结 .....	498	表 B-6 C 标准库<stdlib.h> .....	526
15.10 练 习 题.....	499	表 B-7 字符串函数<string.h> .....	528
<b>第 16 章 C++标准语法补充 .....</b>	<b>502</b>	表 B-8 时间函数<time.h> 与<sys/timeb.h> .....	530
16.1 静态断言 STATIC_ASSERT .....	502	表 B-9 可变参数<stdarg.h>.....	531
16.2 修饰符 CONSTEXPR .....	503	表 B-10 断言<assert.h> 与<crtdbg.h> .....	532
16.3 命名空间 NAMESPACE .....	505	参考文献.....	533
16.3.1 命名空间的定义.....	506		
16.3.2 空间中成员的访问.....	507		
16.3.3 C11 inline 空间.....	510		

# 第1章 概述

本章介绍 C++语言的起源、发展概况及其特点, C++程序的基本结构, 面向对象程序设计的基本概念, 简单的上机操作过程。

## 1.1 C++语言发展历史

C++语言是在 C 语言的基础上逐步发展和完善的, C 语句是吸收了其它高级语言的优点逐步成为实用性很强的语言。

二十世纪六十年代, Martin Richards 开发了 BCPL 语言(Basic Combined Programming Language)。1970 年, Ken Thompson 在 BCPL 语言基础上发明了实用的 B 语言。1972 年, 贝尔实验室的 Brian W. Kernighan 和 Dennis M. Ritchie 在 B 语言的基础上, 进一步充实和完善, 设计了 C 语言, 在 1978、1979 年出版了著名的《C 编程语言》一书, 史称“K&R”。此后 C 语言经多次改进, 并得到广泛应用。Unix 操作系统就是基于 C 语言开发的。目前 C 语言的国际标准是 87ANSI C 规范。常见的有 Microsoft C, Turbo C, Quick C 等, 每一种版本略有不同, 但基本类型、运算和语句是基本兼容的。

C 语言具有以下特点:

- 1.结构化编程语言。以函数作为基本模块, 语法简洁、使用灵活方便。
- 2.具有一般高级语言的特点, 又具有汇编语言的特点。除了提供对数据进行算术运算、逻辑运算、关系运算之外, 还提供了二进制整数的位运算。用 C 语言开发的应用程序, 不仅结构性较好, 且程序执行效率高。
- 3.程序的可移植性比较好。在某一种计算机上用 C 语言开发的应用程序, 源程序经少许更改或不更改, 就可以在其它型号和不同档次的计算机上重新构建运行。
- 4.编程自由度大, 运行错误比较多而且较难解决。指针是 C 语言的灵魂, 精通指针的程序员可以编写非常简洁、高效的程序, 但却不易理解, 而且出错难以排除。初学者掌握 C 语言指针并不容易。

随着 C 语言的不断推广, C 语言存在的一些不足也开始显露出来。例如, 数据类型检查机制比较弱; 缺少代码重用机制; 以函数为模块的编程不能适应大型复杂软件的开发与维护。

1980 年, 贝尔实验室的 Bjarne Stroustrup 博士及其同事对 C 语言进行了改进和扩充, 把 Simula 67 语言中的类引入到 C 中, 称为“带类的 C”。1983、1984 年间进一步扩充, 由 Rick Maseitti 提议将改进后的语言命名为 C++语言(称为 C Plus Plus), 这两个加号应书写为上标, 分别表示**虚函数**和**运算符重载**。之后 C++语言又扩充了模板、异常等新概念, 使 C++功能日趋完善。

C++语言继承了 C 语言的特点, 还具有以下特点:

1.C++是 C 的一个超集,它基本上具备了 C 语言的所有功能。一般情况下 C 语言源代码不作修改或略作修改,就可在 C++环境下构建运行。

2.C++是一种面向对象编程语言。面向对象编程的特性是封装性、继承性和多态性。类作为程序的基本模块。封装性隐藏模块内部的实现细节,而使外部使用更方便更安全。继承性提高了模块的可重用性,而且使程序结构更贴近现实概念的描述。多态性使行为的抽象规范与具体实现相互协调,使行为的一致性和灵活性得到统一。抽象编程和模板提供更好的可重用性,而异常处理则增强了编程的可靠性。这些特征都非常适合大型复杂软件的编程实现。

3.C++语言程序可理解性、可维护性更好。对于大型软件的开发维护,这一特点非常重要。

## 1.2 一个简单的 C++程序

当前常用的 C++语言各有不同,但基本内容都是相同的。本书以学习 ANSI ISO 标准 C/C++为主,主要采用 Visual Studio 2015(VS2015,其中包含 VC14 版)作为开发环境,同时兼容 GevC++5.11 中的 GCC4.9.2。

从 VS98(VC6)到 VS2015(VC14)都是集成开发环境,都可编辑编译和调试用 C/C++语言编写的程序。VC6 能运行在 Win7 上,但不能在 Win8 以上运行。VS2015 可运行在 Win7 以上。

集成环境为了区分是 C 语言和 C++语言程序,约定当源程序文件的扩展名为“.c”时,为 C 语言程序;文件的扩展名为“.cpp”时,则为 C++程序。在文件扩展名中不区分大小写。本书中除作特殊说明外,所有源程序文件的扩展名均为“.cpp”。

下面通过一个简单的实例,说明 C++程序的基本结构及其特点。

例 1-1 根据输入的半径,求出一个圆的面积,并输出计算结果。假设源程序文件名为 ex0101.cpp。VC6 版本的源文件如下:

```
#include <iostream.h>
void main(void){
    float r, area;                //说明两个变量:半径 r 和圆面积 area
    cout << "输入半径 r=";        //显示提示信息
    cin>>r;                        //从键盘上输入半径变量 r 的值
    area = 3.1415926 * r * r;      //计算圆面积
    /*输出半径和圆面积*/
    cout << "半径=" <<r<<'\n';    //输出变量 r 的值
    cout << "圆面积="<<area<<'\n'; //输出圆面积 area 的值
}
```

VS2015 版本的源文件如下:

```
#include <iostream>
using namespace std;
int main(void){
    float r,area;                //说明两个变量:半径 r 和圆面积 area
    cout << "输入半径 r=";        //显示提示符
    cin>>r;                        //从键盘上输入半径变量 r 的值
    area = 3.1415926 * r * r;      //计算圆面积
    /*输出半径和圆面积*/
    cout << "半径=" <<r<<'\n';    // 输出变量 r 的值
```



```
cout << "圆面积="<<area<<'\n';  
system("pause");  
return 0;  
}
```

VC6 能支持非标准和部分标准的 C++ 编码。上面 VC6 源文件不能在 VS2015 上运行，而 VS2015 版本可以在 VC6 上运行。如果你安装了 VS2015 就只能运行后一个版本。

使用 C++ 集成环境，将以上内容输入到文件中，然后用菜单或按钮“build”构建可执行程序，再执行该程序。

该程序在执行时出现一个命令窗口，并显示提示信息，假设从键盘上输入 3.5，显示结果如下：

```
输入半径 r=3.5  
半径=3.5  
圆面积=38.4845
```

程序先输入半径  $r$ ，根据输入值求出圆面积  $area$ ，最后输出半径值和圆面积。对该程序的基本结构及各语句说明以下几点。

### 1. 注释

注释是一段文本信息，用来说明程序的功能或方法。在 C++ 程序的任何位置都可插入注释。注释对程序执行不起作用。注解可增加程序的可读性和可理解性。在 C++ 语言中有两种注解。

第一种是传统 C 语言风格的注释，用“/\*”和“\*/”把一行或多行文本信息括起来作为注释。这种注解可出现在程序中的任何位置，注释文本可以有多行，但不能嵌套。

第二种是 C++ 行注释，用两个连续的“/”字符开头，到本行结束为注释内容。

上面例子中包含了这两种注释。

### 2. 包含文件或编译预处理

用 # 开头的行称为编译预处理指令，`#include` 称为包含指令，指定一个文件，`<iostream.h>` 是标准输入输出流的头文件。如果程序中要从键盘上输入数据，或要将在显示器上输出结果，就应包含该文件。一般程序都需要这个包含指令。有关编译预处理在第 5 章介绍。

### 3. 主函数 main

程序中的 `main` 函数称为主函数。每个 C++ 程序都有一个且仅有一个主函数，程序总是从 `main` 函数开始执行的。`main` 前面的 `void` 表示该函数不返回任何东西，而圆括号中的 `void` 表示该函数启动执行不需要提供任何数据，这个 `void` 可以省略。

VC6 允许 `void main`，表示 `main` 不返回任何值。但标准的 C 都要求 `int main` 要求返回整数。另外，VC6 在启动控制台程序时自动添加 `system("pause")`，用户能看到控制台输出，而 VS2015 从 IDE 中用 F5 或 Ctrl+F5 启动程序时不再添加，就需要在 `return 0` 之前添加 `system("pause")`，否则就执行完成后自动关闭。DevC++ 也是自动添加的。

### 4. 花括号对“{}”

每个函数体都是以“{”开始，并以“}”结束，分别称为开花括号和闭花括号。函数中会出现嵌套的花括号，表示复合语句或作用域，花括号一定要配对使用。

### 5. 语句

一个函数体中包含有多条语句，每条语句以分号“;”结束。各种语句在第 4 章介绍，函

数在第5章介绍。

### 6.程序的书写规则

按照 C++的语法规则，程序的书写形式是自由的，可以将一个语句写成若干行（不能在一个基本语法单位之间换行），也可将若干个语句写在一行内。但为了方便阅读和交流，程序的书写格式及源程序文件中的格式应符合以下基本规则。

(1)对齐规则。同一层次的语句必须从同一列开始，闭花括号与对应的开花括号在垂直方向对齐，或者闭花括号与对应的开花括号所在的语句头字符在垂直方向对齐。这样在垂直方向对齐的一系列语句表示按自上向下顺序执行序列。

(2)缩进规则。程序中除了顺序结构之外，还有选择结构和循环结构(第4章介绍)，为了方便程序的阅读和理解，体现程序的层次结构，属于内一层次的语句，应缩进若干个字符，通常是一个 Tab 键。

(3)函数定义应该从一行开头书写。

7.C++语言没有专门的输入输出语句，数据的输入输出是通过函数调用来实现的。用 `cin>>` 表示输入，用 `cout<<` 表示输出。

8.程序中严格区分大小写字母。C++程序严格区分大小写字母。如 `B` 与 `b` 表示两个不同的标识符。

## 1.3 编程的基本要求

什么是“好”程序的标准？并没有一致的标准的答案，但我们认为下面几方面可作为编程的基本要求。

1.正确性。首先要求程序正确无误，包括语法和语义正确，算法描述正确。这是对程序的最基本的要求。

2.可读性和可理解性。对于复杂程序，如果难以阅读难以理解，其正确性也很难保证。第一，书写格式应规范。第二，程序结构性好，应采用软件工程的程序设计方法来设计程序。第三，程序中增加足够的注释，说明程序设计的目的和方法。

3.可维护性。这要求程序易于扩展新功能，而且在扩展时尽可能不改变已有编程。

4.简洁，执行速度快。最后再改进程序，使编程更简洁更高效。

必须指出，要想设计出高质量的程序，仅学习本课程的知识是不够的，还要掌握数据结构、算法设计与分析、软件工程及程序设计方法学等知识。本课程特别强调实践，只有把理论学习与大量的上机实践相结合，才能学好本课程，才可能设计出高质量的程序。

## 1.4 C++程序的开发步骤

针对一个实际问题，用 C++语言设计一个程序时，通常要经过如下五个基本步骤，如图 1.1 所示。

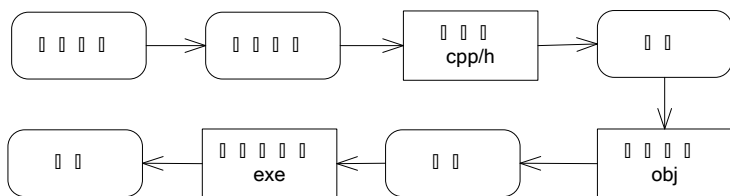


图 1-1 程序开发的基本步骤

1.需求分析。根据要解决的实际问题，分析所有需求，并用合适的方法、工具进行详细描述。如果需求分析错误就可能导致下面的编程完全失去意义。

2.根据需求设计编程。先设计解决方案，然后将解决方案实现为 C++程序，再利用 C++集成环境或某种文本编辑器将设计好的源程序输入到计算机文件中。源文件的扩展名为 `cpp`，也可能是扩展名为 `h` 的头文件。如果设计方案错误将导致程序不能满足需求。源程序编辑器往往能区分 C++语言的关键字、运算符和标识符，而用不同颜色显示，以方便查看。

3.编译源程序生成目标程序。如果有语法或语义错误，要根据提示信息返回到第 2 步修改源程序文件，直到消除所有的编译错误为止。编译后为源程序产生目标文件。在 PC 机上，目标程序文件的扩展名为 `obj`。完成编译工作的工具统称为编译器 `compiler`。

4.将目标文件连接为可执行文件。将一个或多个目标程序与程序所调用的库函数连接后，生成一个可执行文件。在 PC 机上，可执行文件的扩展名为 `exe`。如果在多个文件之间函数调用有错误，此时将给出连接错误信息。完成连接工作的工具称为连接器 `linker`。

5.执行程序。运行测试可执行程序文件，输入测试数据，并分析输出结果。如果结果不满足需求或者与预期不同，就要返回到第 1 步或第 2 步重复以上过程，直到得到正确结果。实际上，可执行文件有两个版本，一个是用于调试程序的 `Debug` 版本，文件比较大，其中包含了源代码调试信息，另一个版本是用来发布软件产品的发布版本，文件比较小，没有调试信息。

## 1.5 VC++开发环境简介

先介绍 VC++6 开发环境，再介绍 VS2015。

VC++6 提供了一个集成开发环境，如图 1.2 所示。集成环境的功能包括源程序的输入、编辑和修改，源程序文件的保存与打开，程序的编译、连接、执行，也包括调试与跟踪、项目的自动管理，提供应用程序的开发工具，多窗口管理，提供联机帮助等。该集成环境功能强大，配置复杂，只有经过较长时间的上机实践、逐步理解和体会，才能熟练运用。

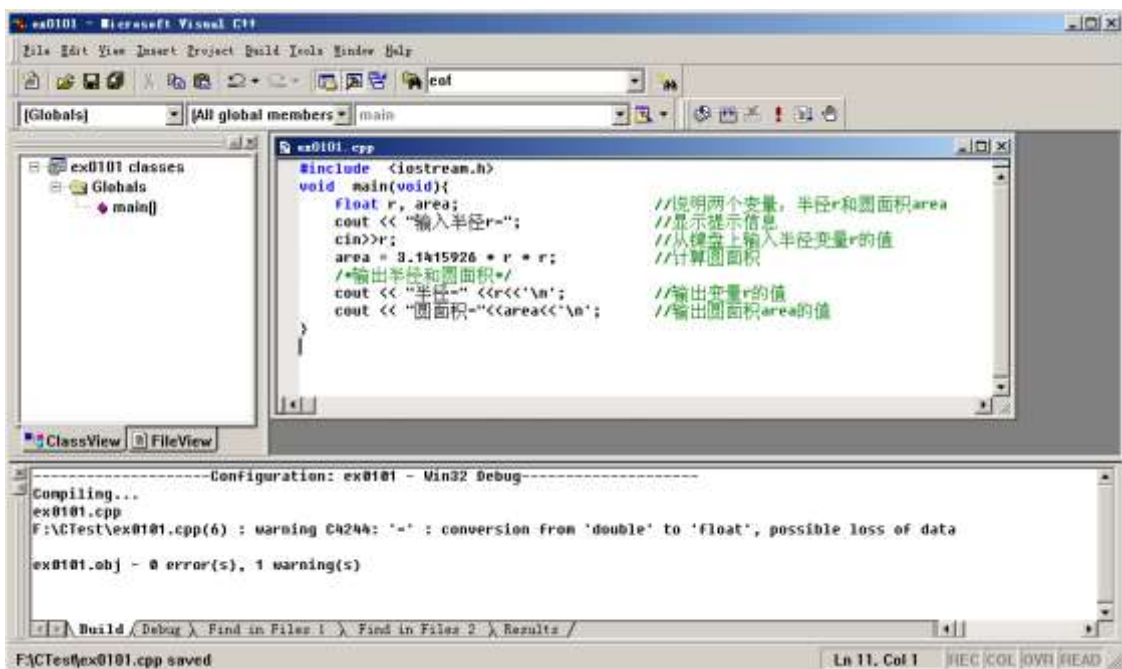


图 1-2 VC++集成环境

在 Windows 操作系统下启动 VC++集成环境, 则产生所图 1.2 的一个组合窗口。最上面部分为窗口标题。第二部分为菜单条, 菜单有“文件 (File)”、“编辑 (Edit)”等菜单。第三部分为工具条, 有“打开”文件、“剪切”、“复制”、“粘贴”等。主体部分由三个子窗口组成, 左边的子窗口为程序结构窗, 有两个视图。类视图 ClassView 用于查看定位程序中的类、全局函数、全局变量等。文件视图 FileView 中列出项目中的所有文件和资源。右边的子窗口为源程序编辑窗, 用于查看编辑源程序文件, 可同时打开多个源程序窗口。下边的子窗口提示程序构建信息, 用来显示出错信息或调试程序的有关信息。

每个程序的构建都需要一个项目工作区 project workspace, 保存为一个 xxx.dsw 文件。集成开发环境只能打开一个项目工作区。因此运行完一个程序之后, 应关闭该程序的项目工作区。file 菜单 Close workspace 就是用来关闭当前工作区, 否则新打开的源文件不能正确构建。

一个项目 project 用来构建一个程序, 一个程序可包含多个源文件, 但其中只能有一个 main 函数, 作为该程序的执行起点。

一般不需要用菜单创建项目。当打开一个 cpp 源文件后开始编译或构建时, 就自动建立一个缺省项目。如果需要特殊配置, 可以在 Project 菜单中修改缺省配置。

建立 cpp 源文件有以下两种方式:

- 1、利用 VC 在资源管理器中对 cpp 文件的关联。在系统中安装 VC 环境之后, 资源管理器对 cpp 后缀文件就关联到 VC 集成环境, 这样只要在资源管理器中建立一个 cpp 后缀文件, 就可以双击打开启动集成环境。用右键菜单新建一个文本文件, 缺省后缀为 txt, 改为 cpp 后缀既可。

注意, 资源管理器可能配置为隐藏文件后缀, 隐藏后缀就不能改变文件后缀。去掉隐藏

后缀的方法如下，在资源管理器菜单：“工具”-->“文件夹选项”-->“查看”-->“隐藏已知文件类型的扩展名”，去掉该选项。

## 2、利用 VC 集成环境来建立 cpp 源文件。

在集成开发环境中，选择“File”菜单中的“New”命令，这时弹出一个子窗口，如图 1.2 所示，有四个标签，分别是文件 Files、项目 Projects、工作区 Workspaces 和其它文档 Other Documents。用鼠标单击 Files 标签，选择“C++ Source File”，确定文件目录 Location 和文件名 File(只需文件名，不要指定后缀)，最后点击“OK”既可。

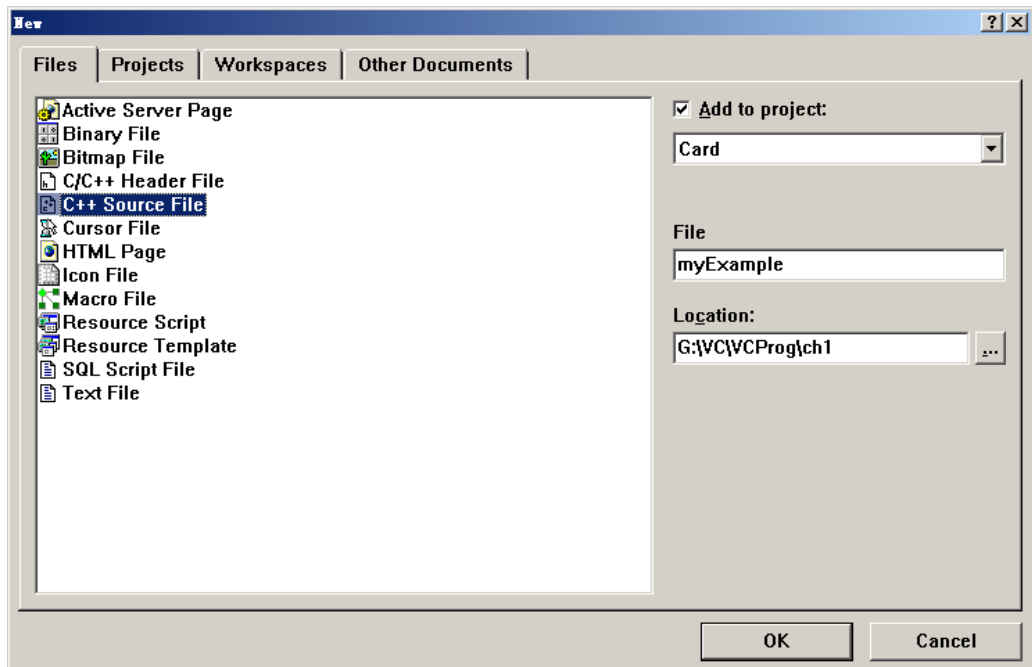


图 1-3 用 VC6 工具建立源文件

要打开一个已存在的源程序文件时，可选择 File 菜单中的 Open 命令，然后根据提示信息选择相应的源程序文件名。由系统将指定的源程序文件打开进入编辑子窗口，就可以对源程序文件进行编辑处理。

输入源程序或编辑结束后，应该先保存源文件。可选择 File 菜单中的 Save 命令来实现。源程序文件存盘后，可选择 Build 菜单中的 Build 命令来编译和连接程序。当编译和连接成功后，可选择 Build 菜单中的 Execute 命令来执行程序。

VC6 可运行在 WinXP、Win7 系统上，但不支持 Win8、Win10 系统。

VS2015 相对于 VC6，摒弃了其中非标准的语言和库，更多地支持 C 语言和 C++标准。

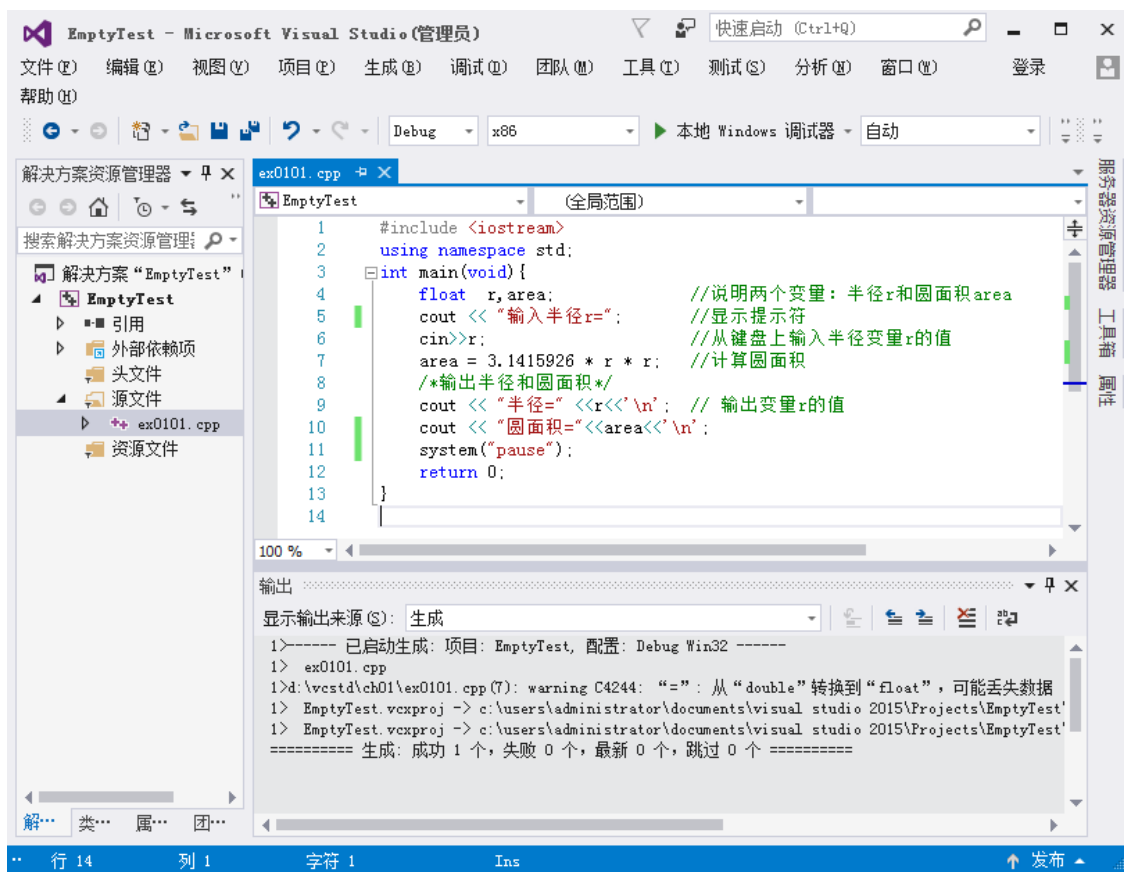


图 1-4 用 VS2015 开发 C++ 程序

VS2015 界面左边是一个解决方案资源管理器，相当于一个项目管理器。页面左上角所显示的名称就是当前项目的名称。注意最多打开一个项目，不能打开多个项目。

一个项目的演化过程如下：

1. 创建项目。“文件”、“新建”、“项目”，或打开已有项目。
2. 配置项目。项目中对源文件，新建或加入已有源文件；
3. 编码。修改源文件编码。
4. 构建项目。“生成”，“生成解决方案 F7”，就是编译和连接；
5. 运行。“调试”，“开始执行 Ctrl+F5”，或者“开始调试 F5”，控制台窗口交互。
6. 返回第 2 步或第 3 步，直到项目完成且运行无错。
7. 关闭项目。“文件”，“关闭解决方案”，即关闭所有源代码窗口。

与 VC6 不同，不能先打开一个源文件之后再自动创建项目。VS2015 要求先创建或打开一个项目，才能加入已有源文件。另外，第 4 步时应确保上一轮控制台窗口已关闭。

1.6 C++标准及开发工具

从 1990 年第 1 个 C 标准发布至今，已陆续发布了 4 个版本，如表 1.1 所示。

表 1-1 C/C++标准概览

简称	名称及发布时间	语言关键特征
C89	ISO/IEC 9899:1990	新的关键字，如 const、volatile、signed 等；宽字符 wchar_t 及宽串、多字节字符。函数可变形参。
C99	ISO/IEC 9899:1999	不定长数组；for 内部变量作用域；inline，long long，取消函数返回类型默认 int
C11	ISO/IEC 9899:2011， 也称为 C1X	unicode 字符类型 char16_t、char32_t，可变宏参量，右值引用和移动语义；类型推导(auto)；基于范围的 for 循环；Lambda 函数；另一种函数语法；创建对象的改进(减少临时对象)；显式虚函数改写 override；空指针 nullptr；强类型枚举 enum class/struct；成员初始化；委托构造函数；继承构造函数；可变参数模板；用户自定义字面值 UDL；静态断言；常量表达式 constexpr
C14	ISO/IEC 14882:2014 也称为 C1Y	泛型 Lambda 函数；函数返回类型推导；另一种类型推导 decltype；成员初始化；二进制字面值

本文将介绍标准中的关键概念。同时也提醒读者，不存在完全符合标准的 C 开发工具，而且所有开发工具或多或少都有自己的非标准方言。

当前流行许多 C++开发工具。下表列出目前常见的部分工具。

表 1-2 常见 C++工具

名称	说明
Visual C++系列	Microsoft Visual Studio 中的一个组件，主要用于开发 Windows 系统程序和应用程序。从 VS98 的 v6 到 VS2015 的 v14 多个版本。其中 v6 对标准兼容仅达 83.43%，而 v7.1 标准兼容性达到 98.22%。
Borland 家族 Turbo C/C++ Borland C++	TC 是最早的、经典的 C/C++集成开发环境，主要支持 DOS 应用程序开发。BC ++系统，其中 5.5 版标准兼容性达 92.73%。
GCC 家族 gcc g++ Mingw	gcc 原先是 gnu c 的编译器，g++是 gnu c++编译器，EGCS(Enhanced GNU Compiler Suite)是 gcc 改进版。目前 gcc 改名并扩展为 GCC(GNU Compiler Collection)。GCC3.3 标准兼容性达 96.15%。 DevC++只是 GCC 的一个外壳，也包含了 Mingw。 Mingw 或 Cgywin,是在 windows 平台上的 gnu c/c++编译器、库文件及运行环境。
Clang	C、C++、Objective-C 轻量级编译器。源代码发布于 BSD 协议下。 编译快速，占用内存较少，产生文件小，出错提示友好且完备。可替代 GCC。

本文所有示例代码都在 VS2015 上运行，同时兼容 DevC++5.11(GCC4.9.2)，如果两者有区别并尽可能指明。

一个 C++ 系统由以下三个主要部分组成：

1、语言规范。确定了语言构造的语法和语义，教我们如何书写源程序。在各种 C++ 语言之间大部分构造符合 ANSI ISO 的 C/C++ 标准，但每一种具体 C++ 系统都有自己的扩展方言和习惯用法。

2、工具。最基本工具是编译器、连接器，在集成开发环境中还包括源程序编辑器、项目配置管理、调试器 debugger 等。使用这些工具能将源程序转换为可执行程序。每一种 C++ 系统在用户界面功能都有所不同。

3、标准库。一个 C++ 系统往往要提供了一组标准函数库和类库，称为应用编程接口 API。源程序可以调用这些库来扩展程序的功能，如输入输出。API 是语言的扩展，调用 API 也可简化编程，提高编程效率。注意 Windows 上大量的库不一定能在 Unix/Linux 系统上运行，反之亦然。只有少量的标准库是通用的。

## 1.7 本书组织结构

本书可分为以下 3 部分：

1、结构化编程。从第 2 章到第 8 章，介绍 C/C++ 语言的基础概念，其中主要是 C 语言构造，以函数为基本模块，主要介绍结构化编程的方法要点。

2、面向对象编程。从第 9 章到第 15 章，介绍 C++ 高级语言概念，主要是 C++ 类的构造，以类作为基本模块，主要介绍面向对象编程的方法要点，也包括运算符重载和模板。

3、补充与附录。第 16 章依据 ISO C++ 标准对前面各章进行补充。附录部分给出了 ASCII 编码规范和常用函数库、类库。

下面简单介绍各章主要内容。

第 2 章先介绍 C++ 语言的关键字、标识符等概念，主要介绍 5 种基本数据类型和 5 种字面值，最后介绍了变量的说明、初始化、赋值等，以及命名常量。这些是 C/C++ 语言中最基础概念。

第 3 章在前一章基础上介绍 10 种基本运算符和 7 种表达式，也详细介绍了基本数据类型之间的类型转换。

第 4 章先介绍了 9 种语句，然后介绍了结构化编程的 3 种基本结构，主要介绍了 C 语言的流程控制语句，包括 2 种选择语句、3 种循环语句、3 种跳转语句，另两个跳转语句，return 在下一章介绍，throw 语句在第 15 章介绍。

第 5 章先介绍函数的定义和调用，介绍了函数重载、递归调用、函数原型，也介绍了形参缺省值、内联函数，作用域与存储类，最后介绍了编译预处理。

第 6 章介绍一维数组和二维数组，也介绍了用函数处理数组的要点，最后介绍了字符数组与字符串处理函数。

第 7 章介绍自定义类型，包括结构、位域、枚举和共同体，其中比较常用的是结构和枚举。



第 8 章介绍指针和引用，难点是指针。详细介绍了指针概念和运算，介绍了指针与结构，指针与数组、指针与函数之间的关系，介绍了 `void` 与 `const` 指针，介绍了 `new/delete` 动态使用内存的方法。对于引用，介绍了引用概念、引用与数组，引用与指针、引用与函数之间的关系。最后作为一个综合性实例介绍了一个单向链表的例子。

第 9 章进入面向对象编程，简单介绍了类的定义、类成员的可见性、数据成员和成员函数，也简单比较了结构和类之间的区别。介绍了对象概念、创建对象、操作对象。最后介绍了 `this` 指针。

第 10 章在上一章基础上详细介绍对象如何创建和撤销。先介绍了构造函数与析构函数，然后详述拷贝构造函数和单参构造函数。介绍了复合对象和成员对象的构造和析构、对象数组，对类中的静态成员进行了详细阐述。介绍了 `const` 修饰符，特别对指针成员和引用成员的设计进行了举例说明。最后介绍了一种特殊指针：指向类成员的指针。

第 11 章介绍类的继承性和多态性。介绍了基类与派生类，派生类的构造与析构，针对多继承带来的二义性问题给出解决方法，也介绍了多继承导致的语义二义性，以及虚基类。介绍了如何用虚函数实现行为多态性。将对象链表作为一个例子，介绍了抽象编程。最后总结了继承性设计的要点。

第 12 章介绍运算符重载，作为一种多态性。除了成员函数实现重载，还介绍了友元函数如何实现重载，最后介绍了 4 种特殊的重载函数。

第 13 章介绍模板，作为一种通用性设计技术。介绍了函数模板和类模板，也详细介绍了标准模板库 STL 中的容器和迭代器的用法。

第 14 章介绍输入输出流，使程序能控制外部设备。主要介绍标准输入输出和文件流。这一章介绍流类库，没有介绍新的语法概念。

第 15 章介绍异常，以改进程序可靠性。先介绍异常概念，结合 STL 类库介绍了异常类型架构，介绍了 `throw` 引发异常，`try-catch` 捕获异常，讨论了如何扩展新的异常类型，如何应用异常类型。还讨论了函数设计中的异常处理方法，在构造函数、析构函数中的异常。

第 16 章依据 ANSI C++ 标准对前面内容补充，介绍了 10 个关键字，其中一些关键字在前面使用过，但没有详细解释，例如逻辑型 `bool`，命名空间 `using namespace`，类型标识 `typeid` 等。

附录 A 给出了 ASCII 编码表。

附录 B 给出了部分常用函数库和类库。

对于各章的一些共同性质，说明以下几点：

(1) 本书中所有例子都在 VC++2015 环境中得到确认，但要移植到其它 C++ 环境中可能需要少许更改。

(2) 本书例子中的输入输出调用了 `<iostream>` 中的 IO 流类库来实现，大多 C 语言编程要调用 `<stdio.h>` 中的 `scanf`、`printf` 等函数来完成输入输出。

(3) 为了方便读者适应纯 C 语言环境，在前 8 章中对 C++ 语法构造进行特别说明。如逻辑型 `bool`、`typeid` 运算符、函数重载、形参缺省值、引用、`new` 与 `delete` 等，这些都属于 C++ 概念，而不是 C 语言概念。第 9 章以后都属于 C++ 语言概念。

## 1.8 练 习 题

- 1.在 VC++集成环境下运行本小程序例子。
- 2.编程的基本要求有哪些？
- 3.在 VC++集成环境下，从输入源程序到得到正确结果，要经过哪些步骤？
- 4.在 VC++中，有哪两种注释方式？每一种注解方法适用于什么场合？

## 第2章 数据类型与变量

计算机 CPU 所处理的数据都有自己的类型。CPU 指令能直接处理的数据类型就是基本数据类型，构成了 C++ 程序设计的最基本单元。任何数据都要先定义其所属数据类型，然后才能使用。一种数据类型决定了一组数据的值的范围和可执行的计算。本章将介绍基本数据类型，以及如何说明这些类型的常量和变量。

### 2.1 关键字和标识符

C++ 语言提供了丰富的数据类型，包括基本(primitive)数据类型、用户自定义类型。基本数据类型是系统预先定义的，是可直接使用的数据类型。用户自定义类型是由基本类型定义而来的新类型，包括结构体、类等，在后面章节介绍。本节介绍组成 C++ 程序的基本单位：关键字、标识符、标点符号、分隔符。

#### 2.1.1 关键字

在 C++ 语言中，关键字(keyword)或者保留字是指系统预先定义的、具有特殊含义和用途的英文单词，因此不允许作为标识符。下面按字母顺序列出 C++ 语言中的部分关键字。

auto	bool	break	case	catch
char	char16_t	char32_t	class	const
constexpr	const_cast	continue	decltype	default
delete	do	double	dynamic_cast	else
enum	enum class	enum struct	explicit	extern
false	float	for	friend	goto
if	inline	int	long	mutable
namespace	new	nullptr	operator	override
private	protected	public	register	
reinterpret_cast	return	short	signed	sizeof
static	static_assert	static_cast	struct	switch
template	this	throw	true	try
typedef	typeid	typename	union	unsigned
using	virtual	void	volatile	wchar_t
while				

上面按关键字的用途分类，如下表所示。

表 2-1 关键字的用途分类

关键字	用途	章节
bool, true, false, char, wchar_t, char16_t, char32_t, int, long, short, float, double, signed, unsigned, void	基本数据类型	2.2 节
const	命名常量	2.4.5 节
sizeof, typeid	运算符	3.1 节
if-else, switch, case, default, while, do-while, for, break, continue, return, goto	流程控制语句	第 4 章
void	函数无形参, 或函数无返回值	5.2 节
inline	内联函数	5.8 节
auto, register, static, extern	存储类	5.11 节
struct, enum, enum class, union	结构、共同体、枚举	第 7 章
typedef, using	类型同义词定义	7.5 节
void	void 指针可指向任何类型的数据	8.5.1 节
const	const 指针, 常量指针	8.5.2 节
new, delete	动态使用内存	8.6 节
class, private, public, protected, this	类, 访问控制修饰符, 当前对象指针	第 9 章, 第 10 章
const	不变性修饰符	10.6.1 节
volatile	易变性修饰符	10.6.2 节
virtual	虚基类和虚函数	第 11 章
operator, friend	运算符重载函数, 友元函数	第 12 章
template, class, typename	模板, 用于函数模板或类模板定义	第 13 章
throw, try, catch	引发异常与捕获异常	第 15 章
static_assert	静态断言	16.1 节
constexpr	常量表达式	15.2 节
namespace, using	命名空间的定义与使用	16.3 节
explicit, mutable	C++标准补充修饰符	16.4 节, 16.5 节
const_cast static_cast dynamic_cast reinterpret_cast	C++类型转换运算符	16.6 节

在 VC 集成开发环境中的源文件编辑器中, 所有关键字都显示为一种特别的颜色, 如蓝

色，以区别于其它文字。

### 2.1.2 标识符

程序中经常要对变量、函数、自定义类型进行命名。标识符(identifier)就是对变量、函数、自定义类型等进行命名的字符串。每个标识符都以字母或下划线开始、由字母、数字及下划线组成的一个字符序列。C++语言中构成标识符的语法规则如下：

(1)由字母(a~z, A~Z)、数字(0~9)或下划线(\_)组成。

(2)第一个字符必须是字母或下划线。例如，example1、Birthday、My\_Message、Mychar、Myfriend 及 thistime 是合法的标识符；8key、b-milk 及-home 是非法的标识符。

(3) VC++中标识符最多由 247 个字符组成。一个有效的标识符的长度为 1~247 个字符，当标识符的长度超过 247 个字符时，其前面 247 个字符有效，后面的字符无效。

(4) 在标识符中，严格区分大小写字母。例如，book 和 Book 是两个不同的标识符。

(5) 关键字不能作为标识符使用。

下面的符号均不符合标识符的定义，不能用作合法的标识符：

```
5ab           //不能以数字开头
$cd           //不能用符号$开头
b1.5          //不能使用小数点
this          //关键字不能用作标识符
```

### 2.1.3 标点符号

程序中需要一些标点符号来作为语法约束。C++语言中的标点符号有以下 10 个：

- , 逗号用做数据之间的分隔符，也可作为运算符。
- ; 分号作为语句结束符。
- : 冒号用作语句标号。
- ' 单引号作为字符常量标记符。
- " 双引号作为字符串常量标记符。
- { 左花括号和 } 右花括号可表示复合语句的开始和结束，也用于表示自定义类型的成员范围。
- ( 左圆括号和 ) 右圆括号改变表达式的运算次序，也用于表示函数形参表和函数调用。
- ... 省略号(3 个连续小数点)在形参表中表示可变参数。

注意，以上标点符号都是英文单字节符号，不能错误输入中文的双字节符号。

### 2.1.4 分隔符

分隔符用来分隔 C++语言中的语法单位，表示前一个语法实体的结束和下一个语法实体的开始。C++中的分隔符有空格符(space)、制表符(tab)、换行符(enter)、注释符(/ \* \*/ 和 //)、运算符和标点符号。其中前 3 种分隔符仅起分隔作用，而后 3 种起双重作用。例如，注释符起注释说明的作用，另一方面也起分隔符作用。一个运算符既表示一种运算，又起分隔符的作用。

2.2 基本数据类型

基本数据类型是 C++中预定义的数据类型，有以下五种：布尔型(bool)、字符型(char)、整数型(int, long)、浮点型(单精度型 float、双精度型 double)和空类型(void)。如下表所示。

表 2-2 基本数据类型

类型名	类型名	字节	数值范围
bool	布尔型	1	true 或 false
[signed] char	有符号字符型	1	-128 ~ 127, $-2^7 \sim 2^7 - 1$
unsigned char	无符号字符型	1	0 ~ 255, $0 \sim 2^8 - 1$
wchar_t	宽字符型	2	0 ~ 65535
char16_t	UTF-16 字符	2	0 ~ 65535
char32_t	UTF-32 字符	4	0 ~ 4294967295
[signed] short [int]	有符号短整型	2	-32768 ~ 32767, $-2^{15} \sim 2^{15} - 1$
unsigned short [int]	无符号短整型	2	0 ~ 65535, $0 \sim 2^{16} - 1$
[signed] int	有符号整型	4	-2147483648 ~ 2147483647 $-2^{31} \sim 2^{31} - 1$
unsigned int	无符号整型	4	0 ~ 4294967295, $0 \sim 2^{32} - 1$
[signed] long [int]	有符号长整型	4	-2147483648 ~ 2147483647 $-2^{31} \sim 2^{31} - 1$
unsigned long [int]	有符号长整型	4	0 ~ 4294967295, $0 \sim 2^{32} - 1$
[signed]long long[int]	有符号长整型	8	-9,223,372,036,854,775,808~ 9,223,372,036,854,775,807 $-2^{63} \sim 2^{63} - 1$
unsigned long long[int]	有符号长整型	8	0~18,446,744,073,709,551,615 $0 \sim 2^{64} - 1$
float	单精度浮点型	4	$\pm 3.4 \times 10^{\pm 38}$ (7 位数)
double	双精度浮点型	8	$\pm 1.7 \times 10^{\pm 308}$ (15 位数)
long double	长双精度型	8	$\pm 1.7 \times 10^{\pm 308}$ (15 位数)

- 注1， 类型名中的方括号[]中的内容可省略。
- 注2， 类型 int 与 long 对于当前 VC6 到 VS2015 编译器是一样的。早期 16 位系统上 int 为 16 位，long 为 32 位。在 32 位系统上，int 提升为 32 位，long 仍为 32 位。在 64 位系统上，int 仍为 32 位，而 long 大多提升为 64 位，与 long long 一样。
- 注3， 上表依据 vs2015 规范。早期的 vc6 不支持 long long，用\_\_int64 代替。
- 注4， 按 C++标准 long double 应为 12 字节(例如 GCC 是 12 字节)，但 VS2015 仍为 8 字节。用 cout<<sizeof(类型名) 可看到该类型的字节大小。
- 注5， 对于浮点数，应理解为两个值(尾数和指数)和 1 个符号构成，即科学表示法。主

要区别是有效的尾数个数和指数范围，float 指数范围是从-38 到+38，double 是从-308 到+308。看似范围很大，但尾数并非连续值。

### 2.2.1 布尔型

布尔型是 C++引入的类型，也称为逻辑型，用关键字 `bool` 表示。`bool` 值只有两个：`true` 和 `false`，分别表示逻辑“真”和逻辑“假”。一个布尔值可能来自一个关系表达式的计算结果。两个布尔值之间能进行逻辑与、或操作。单个布尔值可进行逻辑非操作。一个 `bool` 值占用 1 个字节。

在使用逻辑类型时，应注意以下几点：

(1)`bool` 类型是 C++的基本类型，而不是 C 语言的基本类型。传统 C 语言用整型 0 表示逻辑“假”，非 0 值表示逻辑“真”。

(2)使用 `<iostream>` 中 `cout` 输出一个 `bool` 值，只能看到 1 或 0 值。下面输出才能显示 `true` 或 `false`：`cout<<boolalpha<<boolean`，`boolean` 是一个 `bool` 变量。

(3)用 `cin` 来输入一个 `bool` 值，还是用 1 表示 `true`，0 表示 `false`。

### 2.2.2 字符型

#### 1. 窄字符 `char`

单字节字符称为窄字符字符型。用关键字 `char` 表示。一个字符占用 1 个字节，按 ASCII 字符集，可以表示一个字母、数字、标点符号、分隔符等。附录 A 给出了 ASCII 字符集。ASCII 是英文 American Standard Code for Information Interchange 的缩写。ASCII 字符集是最通用的字符编码标准，称为 ISO-8859 字符集。此外字符 `char` 可用于存储 UTF-8 字符集中的字符。

为了表示中文字符，多个 `char` 形成多字节编码来表示 GB2312 中文字符。用 2 个 `char` 表示一个中文。这种多字节编码支持中英文混合的字符串，称为 ANSI 格式。

窄字符所支持的字符串为 `std::string`，输出用 `cout`。

一个字符 `char` 也能表示单字节整数，取值范围是-128~127。8 位中最高位表示符号，1 表示为负数，0 表示为正数，其余 7 位表示数据(负数用补码表示)。无符号字符型 `unsigned char` 的取值范围是 0~255。字符作为整数，可进行算术运算，如加、减、乘、除、求模，也能进行按位运算等。

#### 2. 宽字符类型 `wchar_t`

`wchar_t` 全称是 wide character type，称为宽字符类型，用于兼容多种语言的字符，如中文 Window 系统中的 GBK 字符集中的中日韩字符。一个 `wchar_t` 字符表示一个中文字符，同时兼容英文字符，一个英文字符也占 2 字节。

宽字符 `wchar_t` 所支持的字符串为 `std::wstring`，输出用 `wcout`。

关于宽字符类型的表示与输出，下面一段代码只能运行在 VS2015 上：

```
wchar_t c1 = L'中'; // wchar_t
wcout.imbue(locale("chs")); // 设置简体中文
wcout << c1 << endl; // 输出'中'
```

注意，Linux 上的 GCC 将 `wchar_t` 设置为 4 字节。

### 3.char16\_t 与 char32\_t

C11 增加这两种类型分别支持 UTF-16 与 UTF-32，分别是 2 字节和 4 字节。最常见的多语言编码就是 unicode。其中 UTF-16 和 UTF-32 都是 unicode 编码标准。UTF-16 存储方式为 Big Endian(简称 BE)和 Little Endian(简称 LE)，两者区别在于存储顺序，比如字符'A'用 UTF-16BE 方式表示是 0x0041，用 UTF-16LE 方式表示则为 0x4100。VS2015 编译器与 DevC++(gcc)多采用 UTF-16BE 编码方式。

unicode 兼容 ASCII。即 ASCII 所能表示的字符，用 unicode 编码可得出一样的值。

unicode 不兼容 GBK(中文 Windows 默认编码)，在需要时进行转换。

## 2.2.3 整型

整数包括 `char`(1 字节)，`short`(2 字节)，`int`(4 字节)，`long`(4 字节)和 `long long`(8 字节)。

整数类型简称整型，用于定义整数值，如 1、10、1024、-34 等。典型整数为 `int`。

整型可分为有符号整型和无符号整型，分别在 `int` 前加 `signed` 和 `unsigned` 来区别。有符号整型数据的最高位表示符号，1 表示负数，0 表示正数，其余位数表示数据。

短整数是用 `short` 修饰的 `int`，2 个字节，共 16 位。有符号的短整数如图 2.1(a)所示，最高位表示符号，其余 15 位用补码来表示数据。无符号的整数不需要符号位，16 位都用来表示数据，如图 2.1(b)所示。从图中可看出，对于内存中同样的二进制位“1111111111111111”，如果把它看成是无符号整数，值为 65535；如果看做为带符号数，则最高位为符号，其值则为 -1。原因是“1111111111111111”的补码为 1。

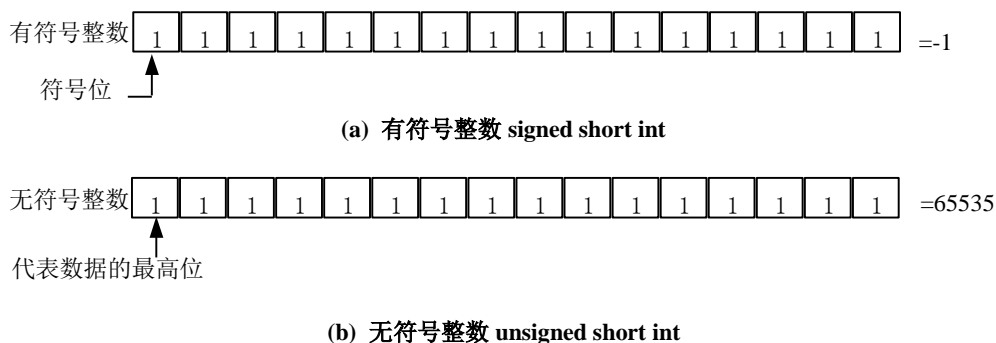


图 2-1 16 位整数的二进制编码



整数采用补码表示。正数的补码就是它自己，而负数的补码是“反码加1”，即先按位求反码(0变为1，1变为0)，再在最低位加1。对于任何一个值 *a*，其补码的补码仍为 *a*。

例如，-3 如何表示为二进制？

原码     00...0011，中间...都是0

反码     11...1100，中间...都是1

加1     11...1101，中间...都是1

这样，-3 的二进制表示就是 11...1101。读者应掌握整数的二进制表示，整数的按位运算需要知道内部二进制表示。

整数可参与多种运算，如算术运算、关系运算、按位运算等，下一章介绍。

## 2.2.4 浮点型

浮点型也称实型，用于定义带小数点的实数，如 3.14、9.8、-1.23 等。计算机对浮点数的存储采用 IEEE754 标准，在内部将一个浮点数转换成二进制形式的科学表示法，分别存储符号、指数和尾数部分。一般不要求读者掌握浮点数的二进制表示法。

float 是单精度浮点型，占用 4 字节。有 7 位准确数字。

double 是双精度浮点型，占用 8 字节，可以表示更大的指数(可表示更大或更小的数值)，以及小数点后更多的位数，有 15 位准确数字。

值得注意的是，long double 仍然是 8 字节浮点数，与 double 一样。

所有的浮点数都是带符号的，因此不能用 unsigned 来修饰 float 或 double。

浮点型数据可参与多种运算，如算术运算、关系运算等，但不能参与求余数%运算、按位运算，也不适合参与逻辑运算(尽管不会导致语法错误)。

## 2.2.5 空类型 void

空类型也称空值型，用关键字 void 表示，void 类型表示“未知”类型，不能说明其变量，但具有特殊的作用如下：

- void 用于函数形参说明一个函数没有形参。第 5 章介绍。
- void 用于函数返回值说明该函数没有任何返回值。第 5 章介绍。
- void 指针可指向任意类型的数据。void 指针常作为通用性函数的形参。第 8 章介绍。

## 2.3 字面值

在程序中直接指定的、不变的数值被称为字面值 literal。在表达式中，一个字面值表示一个确定的值，可对变量进行初始化，或对变量赋值，或直接参与运算。在 C++ 语言中，根据数据类型可将字面值分为逻辑值、整型值、浮点值、字符值、字符串值。

### 2.3.1 逻辑值

逻辑值是 bool 型的值，只有 true 和 false 两个值，内部分别用整数 1 和 0 来表示。

### 2.3.2 整型值

整型值即整数，包括长整数(long)、短整数(short)、有符号整数(int)和无符号整数(unsigned int)。在 C++ 中整型常量可用十进制、八进制和十六进制表示。

整数加后缀表示数据类型：

- u 或 U 表示无符号整数。
- l 或 L 表示 long 整数。
- ll 或 LL 表示 long long 整数
- l 或 L 与 u 或 U 的组合表示无符号长整数。

(1) 十进制整数。除了表示正、负数的符号外，以 1~9 开头，由 0~9 组成的数。例如：

117 +225 -28 0 1289 5000L 2008u 345ul 345lu

表示正数的“+”号可以省略。

(2) 八进制整数。以 0 开头，由 0~7 组成的数。例如：

0156 0556700L 061102u

(3) 十六进制整数。以 0X 或 0x 开头，由 0~9、A~F 或 a~f 组成的数。例如：

0x12A 0x5a0BL 0xFFFF2u

(4) 二进制整数。C14 允许二进制表示整数，以 0B 或 0b 开头，由 0 或 1 组成的数。例如，0b1011 表示十进制 11。

### 2.3.3 浮点值

浮点值由整数和小数两部分组成。实型常量包括单精度(float)数、双精度(double)数和长双精度(long double)数。实型常量可采用定点表示法或科学表示法。

(1) 定点表示法(也称为小数表示法)。由整数和小数两部分组成(中间用小数点隔开)或在小数点的左边或右边有数字。

- 不带后缀表示双精度数 double。
- 后缀字符 f 或 F 表示单精度数 float。
- 后缀字符 l 或 L 表示长双精度数 long double。

例如：

1.7148f 0.5 .25 18. 2.8 3.69L

(2) 科学表示法(也称为指数表示法)。由尾数和指数两部分组成，中间用 E 或 e 隔开。例如：

-3.62E2 //表示  $-3.62 \times 10^2$

1e-10 //表示  $10^{-10}$

科学表示法必须有尾数和指数两部分，并且指数只能是整数。

注意，1.65 和 1.65F 在数值大小相同，但它们占用的存储空间不同，1.65 是 double 型，占 8 字节，1.65F 是 float 型，占 4 字节。

例 2-1 各种类型常量的输出显示。

```
#include <iostream>
using namespace std;
int main(void) {
```

```

cout << 12 << ' ' << 012 << ' ' << 0x12 << '\n';
cout << 100 << ' ' << 0144 << ' ' << 0x64 << ' ' << '\12';
cout << 256 << ' ' << 123u << ' ' << 1024ul << ' ' << 128L << '\012';
cout << 1.2 << ' ' << -.34 << ' ' << -3.4E12 << ' ' << 87.4L << endl;
system("pause");
return 0;
}

```

运行程序，输出结果如下：

```

12 10 18
100 100 100
256 123 1024 128
1.2 -0.34 -3.4e+12 87.4

```

### 2.3.4 字符值

用单引号括起来的字符值，包括普通字符值和转义字符值。

(1) 普通字符常量。用一对单引号将单个字符括起来构成一个普通字符常量。例如：

'b' 'B' '5' '&'

都是合法的普通字符常量。这种字符常量用来表示可直接在键盘上输入的字母、数字和符号。

一个字符常量实际存储的是其在 ASCII 表中对应的编码数值。例如，字符'5'的 ASCII 编码为十进制 53。在存储字符'5'时，实际上存储的是整数值 53(二进制为 00110101)，而不是整数 5。字符常量的存储形式与整数的存储形式相同。一个字符常量可以赋给一个整型变量。反之，一个整型常量也可赋给一个字符变量。在整数可以参与运算的地方，字符数据也可参与。此时相当于对该字符的编码参与运算。

(2) 转义字符常量。转义字符常量是一种特殊表示形式的字符常量，它以“\”开头，后跟一些表示特殊含义的字符序列组成。通常，对于不可显示的字符或不能从键盘上输入的字符，只能采用转义序列来表示。表 2.3 列出了常用的转义字符、十进制值及含义。

表 2-3 常用转义字符及其含义

符号	值	含义	符号	值	含义
\a	7	响铃	\v	11	纵向制表
\b	8	退格 Backspace	\'	39	单引号'
\f	12	换页	\"	34	双引号"
\n	10	换行 Enter	\\	92	反斜杠\
\r	13	回车	\0	0	空 null 字符，串结束符
\t	9	横向制表 Tab	\ooo		1~3 位八进制数
\xhhh		1~3 位十六进制数	\uxxxx		utf-16, 4 个十六进制数
			\Uxxxxxxxx		utf-32, 8 个十六进制数

例如，换行符的 ASCII 编码值为 10，要表示该字符'有多种方式：'\n'、'\12'(八进制)、'\012'(八进制)、'\xa'(十六进制)、'\x0a'(十六进制)、'\x00a'(十六进制)。

采用转义序列可以表示键盘上不能直接输入的字符，例如图形符号“▼”，其 ASCII 编码为 31，则可以表示为'\x1f'或'\037'，下面语句在屏幕上输出该字符：

```
cout << '\x1f';
```

采用转义序列表示单个字符时，虽然单引号内包含多个字符，但是它们整体上只表示一个字符，编译器在见到字符“\”时，会对后面字符作特别处理。如果是一个整型常量，则必须是一个八进制或十六进制数。八进制数可用 0 开头，也可不用 0 开头，最多读取 3 个 0-7 的数字组成一个八进制数；而十六进制数必须以 X 或 x 开头，最多读取 2 个 0-f 字符组成十六进制数。

上面所说字符指的是单字节 char，对于宽字符 wchar\_t 类型、char16\_t 与 wchar32\_t，在这些字符值前加适当的前缀。例如：

```
wchar_t ch2 = 'a';
wchar_t ch3 = L'B';           //英文字符可加也可不加 L
wchar_t ch4 = L'中';          //中文字符加 L
char16_t ch5 = u'文';         //UTF-16加 u
char32_t ch6 = U'字';         //UTF-32 加 U
```

2.3.5 字符串值

用双引号括起来的若干个字符被称为字符串值(简称串值)。串值存放在连续多个字符型数据中。例如：

```
"This is a dog."   "bb"      "2008-9-10"      "C++程序设计"
```

字符串中可包含空格符、转义字符、中文字符等。字符串值不同于字符值，两者之间区别如下：

- (1) 字符值用单引号，而字符串值必须用双引号。
- (2) 存储方式不同。例如，串值"Program"、"m"、字符值'm'的存储方式如下图所示。每个串都在尾部添加结束标志\0。比如串"m"占 2 个字节，一个字节存放字符 m，下一个字节存放串结束标志\0。而字符值'm'仅占一个字节，用来存放字符 m。

P	r	o	g	r	a	m	\0
m	\0						
m							

图 2-2 字符串与字符的存储方式

- (3) 串值和字符值所能进行的运算是不同的。例如，"m"+"n"是非法运算，而'm'+'n'是合法的(其结果是这两个字符对应的 ASCII 码值相加：109+110)。
- (4) 串值的存放需要一个字符数组，在第 6 章将详细介绍。
- (5) 一个字符可表示最多一个转义符，但一个字符串中可能包含多个转义符。例如："b\xabHc\12345\\"包含了 8 个字符。

2.4 变量

变量(variable)是值可变的命名的实例。变量的用途就是保存参与计算的值或计算结果。

一个变量有 3 个基本要素：变量名、类型和值。一个变量名标识了一个存储数据的空间；每个变量都有一个确定的数据类型，其类型定义了变量的存储方式、取值范围、可执行的运算或操作。变量是通过说明(声明)来定义的；每个变量必须有一个确定的值时才能参与运算。不管什么类型的变量，说明在前，使用在后。

### 2.4.1 变量的说明

程序中的每个变量在使用之前必须先说明。变量名用一个标识符来命名。所谓变量说明(declare, 也称为声明)就是指变量的名称及其类型。变量说明语句的语法规则为：

<数据类型> <变量表>;

在说明变量时，应注意以下几点：

(1) 相同类型的变量可以放在一个说明语句中说明，变量名之间用逗号隔开。例如：

```
int number1, number2, number3; //说明整型变量 number1, number2, number3
float total, sum;              //说明浮点型变量 total, sum
```

(2) 变量可以在使用之前的任何地方说明。例如：

```
int count;                    //说明整型变量 count
count = 1;
float sum;                    //说明浮点型变量 sum
sum = 0;
```

(3) <数据类型>可以是基本数据类型或自定义类型，以及这些类型的数组、指针或引用类型。在后面章节将介绍数组、指针和引用。

(4) 变量说明后，系统会根据类型为其分配相应大小的内存空间，引用除外。

(5) 局部变量在说明后如果没有初始化，其值是一个随机值。

### 2.4.2 变量的初始化

在变量说明的同时可指定一个初始值，称为变量的初始化，就是在说明的变量名之后用一个等号后跟一个字面值或者一个表达式。例如：

```
bool b = false;
float x = 2.5f;
char ch1 = 'm';
int number1 = 1, number2 = 2, number3 = 3;
```

或者用圆括号将初始值放在变量右边：

```
bool b(false);
float x(2.5f);
```

如果只说明变量而未初始化，局部变量的值是不确定的。建议对每个说明的变量都加上初始化，可以避免一些潜在的错误。例如，对逻辑型变量用 false、对字符、整数和浮点数用 0 来初始化。

如果程序中直接使用未初始化的变量，VC6 编译器将给出警告，而 VS2015 编译错误。变量说明及初始化的语句就被称为变量说明语句。

2.4.3 auto 初始化

C11 标准提供一种类型推理机制，在变量初始化时，如果右边字面值的类型可知，那么左边变量的类型就可以自动推导，从而避免相同的语义重复。例如：

```
auto b = true;           //b 类型为 bool
auto c = 'A';            //c 类型为 char
auto i = 22;             //i 类型为 int
auto k = 33LL;           //k 类型为 long long 即 __int64
auto m = 44u;            //m 类型为 unsigned int
auto f = 3.14f;          //f 类型为 float
auto d = 3.1415926;      //d 类型为 double
```

关键字 auto 在后面还有多种用法。

2.4.4 变量的赋值

变量的赋值就是给已说明的变量赋予一个新的值，变量的赋值要用赋值语句来完成。在赋值时应按该变量的数据类型来赋予相应类型的值，可以是一个字面值，也可以是一个表达式的值。如果没有按相应类型进行赋值，编译器就按照一套强制转换规则来进行转换。如果不符合强制转换规则，则编译器会报错。例如：

```
float x, y;              //说明 2 个 float 变量
x = 2.5f;                //使变量 x 的值为 2.5f
y = 1.426355;            //使变量 y 的值为 1.426355，但会产生一个警告
```

上面第 1 条语句是变量说明语句，而后面 2 条语句是赋值语句。

赋值语句中可将一个变量赋给另一个变量，例如：

```
x = y;                   //将变量 y 的值赋给变量 x
```

此时 x 的值改变，与 y 相同。

对整型变量的初始化或者赋值时，如果等号右边是字面值，此时编译器的语法检查往往比较宽松，比如，unsigned int 应该是无符号的正整数，但可用负值来初始化。例如：

```
unsigned int j = -1;
cout<<j<<endl;
```

上面第一条语句用-1 来初始化无符号整型变量 j，没有任何错误，而且很有用。第二条语句将 j 的值显示出来，可以看到 4294967295，这正是 unsigned int 型的最大值，其 32 位都是 1。因为-1 就是 0 减去 1 的结果，0 意味着 32 位全 0，减去 1 就是全 1。

2.4.5 变量的输入输出

变量的值可以来自初始化，或者赋值语句，也可以来自键盘输入 cin。为了观察变量的值，也需要将变量的值显示出来，要用 cout。表 2.4 给出了采用<iostream>对基本类型进行简单输入输出的情况。

表 2-4 基本类型变量的简单输入输出

变量类型	cin 输入	cout 输出
bool	不能输入	输出 1 表示 true，0 表示 false。

char	可输入字母、数字、符号、组合键。 不能输入空格符 (Space)、制表符 (Tab)、换行符 (Enter)。	转换为 ASCII 可见字符或控制符。
int, short	缺省为十进制输入, 前面可加+或-。 dec 表示十进制输入输出; hex 表示十六进制; oct 表示八进制。	缺省为十进制输出。如果是 unsigned, 显示正值, 如果是 signed, 可能有负值, 负值前加-。
float、double	能自动识别定点表示法和科学表示法。	自动选择输出方式, 尾数部分保留 6 位有效数字, 四舍五入。如果有效数字超过 6 位数字, 或者有效数字在小数点右边第四位之后, 就以科学表示法输出。

表中列出了最简单的输入输出, 更复杂的控制可参见 14.2.3 节。

在进行输入输出时, 注意以下几点。

(1) 如果要将 bool 类型变量名按 true 和 false 进行输出, 模仿下面例子:

```
cout<<boolalpha<<boolvar;
```

(2) 如果要输出一个 char 变量的整数值, 而不是字符, 就要先转换为 int 值再输出, 例如:

```
cout<<(int)c.
```

(3) 对于整数 int 和 short 类型变量的输入输出, 缺省为十进制 dec, 也可改变为八进制 oct 或十六进制 hex。例如:

```
cout<<oct<<i; //将 i 输出为八进制;
cout<<hex<<i; //将 i 输出为十六进制;
cout<<dec<<i; //将 i 输出为十进制, 注意前面并不添加 0x 或 0X
```

注意, 对 X 进制一次设置之后, 后面的 cout 都有效, 除非改变设置。

(4) 如果要输出一个换行符, 可用 cout<<endl;

(5) 尽管 double 比 float 具有更高精度, 但用 cout 输出时显示相同的有效数字。

例 2-2 变量的说明、赋值和输出。

```
#include <iostream>
using namespace std;
int main(void){
    bool b = false;
    short s1, s2 = 100;
    int len;
    long second;
    float f = 12345678.9f;
    double d = 0.0000123456789;
    cout << "b=" << b<< endl;
    cout << "f=" << f << endl;
    cout << "d=" << d << endl;
    len = 300;
    second = 128;
    b = true;
    s1 = s2 = 30;
    cout << "len=" << len << endl;
    cout << "second=" << second << endl;
    cout << "b=" << b << endl;
    cout << "s1=" << s1<<'\t'<<"s2="<<s2<< endl;
    //D
```

```

    cout << "s1=(hex)" << hex << s1 << endl;           //E
    cout << "s1=(oct)" << oct << s1 << endl;           //F
    system("pause");
    return 0;
}

```

运行程序，输出结果如下：

```

b=0
f=1.23457e+07
d=1.23457e-05
len=300
second=128
b=1
s1=30    s2=30
s1=(hex)1e
s1=(oct)36

```

D 行表示将 30 赋给 s2，然后 s2 再赋给 s1，因此 s1 等于 s2。.

E 行显示为十六进制值，F 行显示为八进制值。

例 2-3 整数和浮点数的输入输出。

```

#include <iostream>
using namespace std;
int main(void){
    int i1, i2, i3;
    cout<<"input 3 int values:";
    cin>>i1;                                     //A
    cout<<"i1="<<i1<<endl;
    cin>>hex>>i2;                                 //B
    cout<<"i2=(dec)"<<dec<<i2<<endl;
    cin>>oct>>i3;                                 //C
    cout<<"i3=(dec)"<<dec<<i3<<endl;

    double d1, d2, d3;
    cout<<"input 3 double values:";
    cin>>d1;
    cout<<"d1="<<d1<<endl;
    cin>>d2;
    cout<<"d2="<<d2<<endl;
    cin>>d3;
    cout<<"d3="<<d3<<endl;
    system("pause");
    return 0;
}

```

运行程序，输入输出结果如下：

```

input 3 int values:1234 1c 24
i1=1234
i2=(dec)28
i3=(dec)20
input 3 double values:1.23e-2
d1=0.0123
123456
d2=123456
-1234567
d3=-1.23457e+006

```

当执行 A 行 cin 输入一个整数时，实际上输入了 3 个整数，使 B 行和 C 行也完成了输入。



第1个整数是十进制输入1234，第2个是十六进制输入1c，十进制输出为28；第3个是八进制输入24，十进制输出为20。

输入的第1个double值为科学表示法1.23e-2，显示为定点表示0.0123，这是因为没有超过6位有效数字。第2个输入6位有效数，输出显示为6位。第3个输入7位有效数字，输出就转换为科学表示法。

当输入整数或浮点数时可能出错，对于错误处理，可参见14.2.4节。

#### 2.4.6 命名常量

命名常量就是用关键字const来修饰的变量。const是英文constant的缩写，表示不可变的意思，它作为修饰符放在类型名的前面。命名常量只能在说明时指定其值，一旦初始化后就不能用赋值语句再修改其值。例如：

```
const double pi = 3.1415;      //承诺下面代码不会改变pi的值
...
pi = 3.1415926;               //编译错误
```

在进行大型程序设计时，命名常量非常有用。例如，在一个程序中需要反复使用pi的值，如果在多个地方要改用一个新值，那么程序的可维护性将非常差。而如果定义一个命名常量pi，那么程序的可读性将增强。当要修改新值时，只需修改常量pi一个地方。

基本类型是构建程序的原材料，用这些基本类型可以做下面事情：

- 定义变量表示该类型的一个值，而且一个变量需要一个命名。
- 定义某种类型的数组，以管理多个元素，共用一个名字。在第6章介绍。
- 定义结构类型和共同体类型，它们都是自定义类型。在第7章介绍。
- 定义基本类型、自定义类型的指针和引用。在第8章介绍。
- 定义类，在第9章介绍。

## 2.5 小结

- 关键字是系统预先定义的、已经具有特殊含义和用途的英文单词，不允许用户重新定义，即不能作为新的标识符出现在程序中。
- 标识符是用来对变量、函数、自定义类型进行命名的字符串，以字母或下划线开始的字母、数字以及下划线组成的字符序列。
- 数据类型包括基本数据类型和自定义数据类型。基本数据类型是预定义的数据类型，包括布尔型bool、字符型char、整数型short和int、浮点型float和double，空类型void。
- 字符型char同时也是一种单字节整数型，同样可用signed和unsigned来确实是否带符号。
- 对每一种类型应掌握其字节数和数值范围。
- long用来修饰int和double在VC6中不起作用。
- 每一种基本数据类型都对应有特定的字面值。

- 字符型字面值有一组转义字符，应掌握。
- 变量用于保存参与计算的值及计算结果。每一个变量都应先说明、后使用。说明应确定类型和变量名，也可进行初始化。说明语句用于说明一组变量的类型、变量名及初始值。变量在使用时可读取参与计算，也能用赋值语句来保存计算结果。
- 可利用<iostream>中的 cin 从键盘读入特定类型的值，用 cout 将值输出到显示器。但对不同类型的值有不同的处理方式。
- 命名常量就是用 const 修饰的变量，必须加以初始化，而且以后不可再改变。

## 2.6 练 习 题

1. 下面哪一个可作为合法的标识符？  
A default      B register      C extern      D Void
2. 下面哪一个不能作为合法的标识符？  
A integer      B default      C VAR      D cher
3. 下面哪一个是非法的字面值？  
A 0xEF      B 1.2e0.6      C 5L      D '\56'
4. 下面哪一个是非法的数据类型？  
A signed short int      B long short  
C unsigned long int      D unsigned int
5. 下列十六进制的整型常数中，哪一个是非法的？  
A 0xbe      B 0x2c      C xef      D 0xEF
6. 下面哪一个合法的字符常量？  
A "A"      B 72      C '\326'      D D
7. 字符串"Ug\ 'f\"028'"中有多少个字符？  
A 7      B 8      C 9      D 字符串非法
- 8 给出下面程序输出结果。

```
int main(){
    int i = 036;
    cout<<i<<endl;
    cout<<hex<<i<<endl;
    cout<<oct<<i<<endl;
    return 0;
}
```

9. 对于下面两个浮点数：  
double d = 1234567;  
float f = 1234567;  
用 cout 输出 d 和 f，输出结果一样吗？
10. 给出下面程序输出结果。  
int main(void){  
 cout<<"Zhao:Hello!";      cout<<"How are you?\n";  
 cout<<"Liu:I am fine,\t";      cout<<"Thank you\n";  
 cout<<6+'012'<<'\'t';      cout<<6+'x12'<<endl;  
 return 0;  
}

11. 给出下面程序输出结果。

```
int main(void) {
    char ch1='a',ch2='b',ch3='c';
    int i=9,j=8,k=7;
    double x=3.6,y=5.8,z=6.9;
    ch1=ch2;ch2=ch3;ch3=ch1;
    cout<<"ch1="<<ch1<<"  ch2="<<ch2<<"  ch3="<<ch3<<endl;
    j = k; k = i; j = k;
    cout<<"i="<<i<<"  j="<<j<<"  k="<<k<<endl;
    x = y; y = x; x = z; z = y;
    cout<<"x="<<x<<"  y="<<y<<"  z="<<z<<endl;
    return 0;
}
```

## 第3章 运算符与表达式

运算符(operator)也称为操作符,对程序中的数据进行运算。参与运算的数据称为操作数(operand)。变量、字面值等通过运算符组合成表达式,一个表达式也能作为操作数来构成更复杂的表达式。表达式(expression)是构成程序语句的基本要素。本章将介绍基本运算符,以及如何由基本运算符组成的各种表达式。

### 3.1 基本运算符

对于运算符,应注意以下几方面。

(1) 运算符的功能和语义。每个运算符都具有特定的操作语义。例如,3+9 中运算符“+”完成算术加法运算。

(2) 运算符的操作数。每个运算符对其操作数的个数、类型和值都有一定限制。运算符分为单目运算符(unary operator)、双目运算符(binary operator)和三目运算符(ternary operator),它们分别要求有 1 个、2 个和 3 个操作数。类型的限制是指特定的运算符只能作用于特定的数据类型。例如,“%”求余数运算符要求操作数的数据类型为整型。有些运算符对操作数的值有一定的限制。例如,“/”除法运算符要求除数不能为 0。

(3) 运算符的优先级(precedence)。每个运算符都有确定的优先级。当在一个表达式中出现多个运算符时,优先级决定了运算的先后顺序。优先级高的运算符先运算,优先级低的后运算。例如,乘法“\*”优先级高于“+”,所以在表达式 5+6\*2 中,要先做“\*”运算,后做“+”运算。圆括号能改变优先级。例如表达式(5+6)\*2 就要先计算括号内的表达式,再计算括号外的表达式。圆括号能嵌套使用,内层优先。

(4) 运算符的结合性(associativity)。一个单目运算符可能要与其左边的表达式结合(称**右向左**),例如 count++, count 后置自增 1;也可能要与其右边的表达式结合(称**左向右**),例如!isEmpty, !是求逻辑非。一个双目运算符可能从**左向右**结合,例如 i+j, 先计算 i 再计算 j;也可能从**右向左**结合,例如 i=j+2, 先计算右边的 j+2, 再赋给 i。赋值=运算是从右向左计算的。一个运算符的结合性确定了该运算符特定的运算次序,但结合性往往不够可靠,文献规范与实际情况不一定相符。

表 3.1 给出了 C++中的主要运算符的功能、优先级、目数、结合性。表中按优先级从高到低分为 16 个级别,其中一些级别中有多个运算符,例如第 1 级和第 2 级,排在上端的运算符拥有较高级别。本章下面将介绍其中一些基本运算符,包括算术运算符、关系运算符、逻辑运算符、位运算符、条件运算符、赋值运算符、逗号运算符、自增自减运算符、sizeof 和 typeid 运算符等。其余运算符(表中有阴影的部分)将在后面章节介绍。

表 3-1C++运算符

优先级	运 算 符	功能及说明	目数	结合性
1	::	作用域解析	单双	无
2	()	函数调用	单双	无
	[]	数组下标		
	. 或 ->	访问成员运算符		
	typeid(...)	求表达式的类型标识		
	const_cast	常量类型转换		
	dynamic_cast	动态类型转换		
	reinterpret_cast	重新解释的类型转换		
	static_cast	静态类型转换		
	++ 或 --	后缀自增、自减		
3	new 或 delete	创建(分配内存)或撤销(释放内存)	单目	左向右
	++ --	前缀自增、自减		
	*	间接寻址		
	&	取地址		
	+ 或 -	正、负号		
	!	逻辑非		
	~	按位求反码		
	sizeof(...)	求对象/值或类型的字节长度		
	(type)expr 或 type(expr)	强制类型转换		
4	.* ->*	成员指针运算符	双目	左向右
5	* / %	乘、除、取余	双目	左向右
6	+(加) -(减)	加、减	双目	左向右
7	<< >>	左移位、右移位	双目	左向右
8	< <= > >=	小于、小于等于、大于、大于等于	双目	左向右
9	== !=	等于、不等于	双目	左向右
10	&	按位与	双目	左向右
11	^	按位异或	双目	左向右
12		按位或	双目	左向右
13	&&	逻辑与	双目	左向右
14		逻辑或	双目	左向右
15	?:	条件运算符	三目	左向右
16	= += -= *= /= %=	赋值运算符	双目	右向左
	<<= >>= &= ^=  =			
17	,	逗号运算符	双目	左向右

### 3.1.1 算术运算符

进行算术运算(如加、减、乘、除)的运算符被称为算术运算符。

#### 1. 单目算术运算符

- 负数运算符, 将操作数变成原值的相反数。
- + 正数运算符, 与操作数符号相同, 经常省略。

例如:

```
int a = -3;
int c = -a;           //将 a 存储单元的值的相反数放到 c 的存储单元中, a 中的值没有改变
cout << c << ' ' << a;    //输出: 3 -3
```

#### 2. 双目算术运算符

包括四则运算和求余数运算。

- + 加法运算符
- 减法运算符
- \* 乘法运算符
- / 除法运算符
- % 求模运算符(或求余数运算符)

+, -和\*运算符的功能分别与数学中的加法、减法和乘法的功能相同, 分别计算两个操作数的和、差、积。这三个运算符可作用于全部基本类型。当作用于 bool 类型时, 虽然没有什么合理的语义, 但语法没有错误。

对于乘法运算符 “\*”, 若两边操作数中有一个是实型数时, 则作实数的乘法运算, 否则作整数乘法运算(当操作数有一个整型数和一个实型数时, 编译器会自动把整型数强制转换成实型数进行运算)。

对于除法运算符 “/”, 若两边的操作数均为整型数, 则作整除运算, 结果只取运算结果的整数部分, 去掉小数部分, 并不进行四舍五入。如果有一个操作数是浮点数, 结果就是浮点数。例如:

```
int a = 33;
cout << a/2;           //输出 16
cout << 5/2;           //输出 2
cout << 5.0/2;         //输出 2.5
```

在除法运算中, 除数不能为 0。如果被除数是整数, 除数为 0 将导致严重错误而终止程序。如果被除数是浮点数, 除数为 0 将导致数据溢出, 得到一个无效浮点数(NaN, Not a Number 的缩写), 用 cout 显示为 “1.#INF”, 表示正数溢出, “-1.#INF” 表示负数溢出。

对于求模运算符 “%”, 其两边的操作数必须是整型数, 其计算结果是两数相除后所得到的

的余数。如：

```
cout << 11%4;      //输出 3
cout << 35%3;      //输出 2
cout << 5.0%3;     //编译错误，%两边都必须为整型操作数
```

求模运算符“%”右面的值不能是0，否则会导致严重错误而终止程序。

例如：

```
int i = 0;
cout << 32 % i;    //运行出错
```

对于两个整数  $x$  和  $y$ ，除法和求模运算应满足以下恒等式：

$x / y * y + x \% y == x$

当  $x$  为负值时， $x \% y$  也为负值。例如  $-7 \% 2$ 、 $-7 \% -2$  的结果都是  $-1$ 。

实际上对浮点数也能求模，只是要调用函数 `fmod` 来实现。

算术运算符的优先级为(括弧中运算符的优先级相同)：(单目+、-)高于(\*、/、%)高于(双目+、-)。如：

```
int a = 5;
int b = 3;
float c = 3.1;
cout << -a*b+c;    //输出-11.9
```

两个整数做加法、减法或乘法运算时，即便结果溢出也不是错误。例如：

```
short s1 = 32765;
s1 = s1 + 3;
cout<<s1<<endl;
```

输出-32768，而不是 32768。实际上，观察二进制数据，这两个值是一样的。

### 3.1.2 关系运算符

对两个操作数进行比较运算的运算符就是关系运算符。C++中提供了6个关系运算符：< (小于)、<= (小于等于)、> (大于)、>= (大于等于)、== (等于)、!= (不等于)。它们都是双目运算符。关系运算的结果是一个表示“真”或“假”的逻辑值，即一个 `bool` 值。当关系成立时，其运算结果为真；当关系不成立时，结果为假。但C语言中没有逻辑型，关系运算的结果要用一个 `int` 值表示，0 即为假，1 即为真。

例如：

```
3 > 5      //3 不大于 5，结果为 false, 0
3 != 5     //3 不等于 5，结果为 true, 1
```

关系运算符的优先级为(括弧中运算符的优先级相同)：(>、>=、<、<=) 高于 (==、!=)。关系运算符的优先级比算术运算符低，但比赋值运算符(=)高。如：

```
int a = 5, b = 3, c = 6, d;
d = a > b == c;    //等价于 d = ((a > b) == c); d 的值为 0
d = a == b < c;    //等价于 d = (a == (b < c)); d 的值为 0
d = a > b <= c;    //等价于 d = ((a > b) <= c); d 的值为 1
```

可以使用()来改变运算符的计算次序。

既然关系运算的结果是一个逻辑值，那么就可以保存在一个逻辑型 `bool` 变量中。例如：

```
bool b = a > 5;
```

由于浮点数在计算机内进行运算和存储时会产生误差，因此在比较两个浮点数时，建议不要直接比较两数是否相等。例如，执行下面语句：

```
double d1 = 3.3333, d2 = 4.4444;
if(d1 + d2 == 7.7777)
    cout<<"相等"<<endl;
else{
    cout<<"不等"<<endl;
    cout<<d1 + d2<<endl;
}
```

条件语句中用“==”来判断浮点数是否相等，结果是不等，但 d1+d2 输出结果却是 7.7777。两个实型数即便输出结果完全一样，其内部值也可能不一样。判断两个实数是否相等的正确方法是：判断两个实数之差的绝对值是否小于一个给定的允许误差数，如判断 d1 是否等于 d2 时，应改为：

```
fabs(d1 - d2) <= 1e-6
```

其中，fabs() 是计算绝对值的一个库函数，使用时要包含头文件 math.h。

### 3.1.3 逻辑运算符

对逻辑值进行运算的运算符就是逻辑运算符。C++语言提供了 3 个逻辑运算符，用于表示操作数之间的逻辑关系，它们是!(逻辑非)、&&(逻辑与)、||(逻辑或)。逻辑运算的结果仍然是逻辑值。

逻辑非(!)是单目运算符，它对操作数进行取反运算。当操作数为非 0(逻辑真)时，! 运算后结果为 0(逻辑假)。反之，若操作数为 0(逻辑假)，!运算后结果为 1(逻辑真)。

注意，所有非 0 的值在逻辑上都作为“真”。例如：

```
cout<<!4 <<endl;      //输出 0, 假
cout<<!-4 <<endl;     //输出 0, 假
cout<<!4.1<<endl;     //输出 0, 假
cout<<!-4.1<<endl;    //输出 0, 假
```

&&和||运算符是双目运算符。运算规则如下：

- &&(逻辑与)表示“而且”的语义。当两个操作数都是 1(逻辑真)时，&&运算后的结果才为 1(逻辑真)；否则就为 0(逻辑假)。
- ||(逻辑或)表示“或者”的语义。当两个操作数都是 0(逻辑假)时，||运算后的结果为 0(逻辑假)；否则为 1(逻辑真)。

例如：

```
!(2 > 6)           //结果为 1, 真
7 > 3 && 10 > 6     //结果为 1, 真
5 > 2 || 4 > 8      //结果为 1, 真
4 && 5 < 6          //结果为 1, 真
```

逻辑运算符的运算优先级为：! 高于 && 高于||。注意，! 的优先级具有较高优先级，甚至高于算术运算符。而&&和||的优先级则比算术运算符和关系运算符低。

对于逻辑值应该只能允许执行逻辑运算(与、或、非)，其它运算都没有合理的语义。另一方面，参与逻辑运算的应该只能是逻辑值，而不应该允许其它类型的值参与。但因 C++将逻辑值保存为整数值，这样使得逻辑值可参与所有的运算，而且逻辑运算符可作用于所有类型



的值，而没有错误提示。这是 C/C++ 语法不严密之处。读者应注意避免。

例 3.1 逻辑运算符的例子。

```
#include <iostream>
using namespace std;
int main(void){
    int a, b;
    bool res1, res2, res3, res4, res5;
    a = 3; b = 5;
    res1 = a > 3 && b >= 5;
    res2 = a > 3 || b >= 5;
    res3 = ! (b >= 5);
    res4 = ! (a == b);
    res5 = a > b && b < 6;
    cout << boolalpha;           //C
    cout << "res1=" << res1 << '\t';
    cout << "res2=" << res2 << '\t';
    cout << "res3=" << res3 << '\t';
    cout << "res4=" << res4 << '\t';
    cout << "res5=" << res5 << endl;
    system("pause");
    return 0;
}
```

执行程序，输出结果如下：

```
res1=false    res2=true    res3=false    res4=true    res5=false
```

上面代码中采用 `bool` 变量来保存逻辑运算的结果，而且能输出 `true` 或 `false`。

### 3.1.4 位运算符

位运算是对于字节内部二进制位进行移位或逻辑运算。位运算应属于一种算术运算。位运算的操作数必须是 `char`、`short`、`int`、`long` 或 `long long` 类型，而且结果也是一样类型。除了按位求反`~`是单目运算符，其余位运算都是双目运算符。C++ 提供了两类位运算：移位运算和按位逻辑运算。

#### 1. 移位运算符

移位运算符的格式为：

`operand << n` 将操作数 `operand` 向左移动 `n` 个二进制位，右边补 0，可能改变符号。

`operand >> n` 将操作数 `operand` 向右移动 `n` 个二进制位，保持符号不变。

其中，`n` 为整数。注意移位运算并不改变 `operand` 本身的值。例如：

```
short operand = 0x8, n = 3;
short a = operand << n;           //结果为 0x40
```

operand	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
a	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0

操作数左移 `n` 个二进制位后，右边移出的空位用 0 补齐。

当左移时，对于带符号数，最高位表示符号，可能会因为低位的 1 或 0 移到最高位，最终改变操作数的符号。如：

```
short s = 4567;
short result = s << 3;
cout << result ; //输出: -29000
```

如果操作数是正数，而且左移没有导致改变符号，那么左移  $n$  位，相当于乘以  $2^n$ 。

我们知道，int 型的数值范围是  $-2^{31}$  到  $2^{31}-1$ 。下面代码利用左移位来得到 int 型的最小值和最大值。

```
int min = 1 << 31;
cout<<"min = "<<min<<endl;
int max = (1 << 31) - 1;
cout<<"max = "<<max<<endl;
```

显示结果为：

```
min = -2147483648
max = 2147483647
```

对带符号数右移，操作数右移  $n$  位后，左边移出的空位用符号位补齐，最高位始终补与原来最高位相同的数，这样右移不会改变符号。如：

```
short operand = -23116, m = 3;    //移动前为 0xa5b4
short b = operand >> m;          //结果为 0xf4b6, 即-2890
```

operand	1	0	1	0	0	1	0	1	1	0	1	1	0	1	0	0
b	1	1	1	1	0	1	0	0	1	0	1	1	0	1	1	0

对不带符号数右移，操作数右移  $n$  位后，左边移出的空位用 0 补齐。

对一个 unsigned 数或者一个正值右移  $n$  位，相当于除以  $2^n$ 。

移位运算的执行效率很高，这是因为 CPU 能直接执行移位指令。

对于移位运算，注意以下几点：

(1)不要尝试对 float 或 double 数据进行移位运算，编译会出错。

(2)移动位数  $n$  应不大于左操作数的位数，如 int 移位应不大于 32。如果  $n$  大于左操作数位数，实际移动位数要自动按字长取模： $n\%(\text{sizeof}(\text{int}))$ 。例如， $i<<33$  就是  $i$  左移 1 位。

(3)当移动位数  $n$  为负值，C 语言规范没有确定是什么结果。在 VC 中，左移  $<<$  负位数，相当于循环右移，即移出的位补回到最高位。右移  $>>$  负值位数，结果总为 0。

(4)左移位  $<<$  与  $\text{cout}<<$  可能混淆，右移位  $>>$  与  $\text{cin}>>$  可能混淆，可用括号消除这些错误，例如  $\text{cout}<<(\text{k}<<3)$ 。实际上， $\text{cout}<<$  是对左移位运算符  $<<$  的重载定义形式， $\text{cin}>>$  是对右移位运算符  $>>$  的重载定义形式。在第 12 章详细介绍。

## 2. 按位逻辑运算符

按位逻辑运算有 4 个：求反  $\sim$ 、与  $\&$ 、或  $\mid$ 、异或  $\wedge$ 。

(1)按位求反“~”运算符是一个单目运算符，对操作数逐位取反，得到反码。若二进制位为0，则取反后为1；若二进制位为1，则取反后为0。例如：

```
short int m = ~0xc3 //结果为 0xff3c
```

0xc3	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	1
~0xc3	1	1	1	1	1	1	1	1	0	0	1	1	1	1	0	0

按位求反运算符具有比较高的优先级，高于一般的算术运算符。而其它按位逻辑运算符的优先级则比较低。

对于一个整数 a，其反码的反码还是 a。例如：

0 的反码是-1，-1 的反码是 0

1 的反码是-2，-2 的反码是 1

2 的反码是-3，-3 的反码是 2

(2)按位逻辑与“&”对两个操作数逐位进行运算。若对应位都为1，则该位结果为1，否则为0。例如：

```
short int a=0xc3 & 0x6e //结果为 0x42
```

0xc3	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	1
0x6e	0	0	0	0	0	0	0	0	0	1	1	0	1	1	1	0
a	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0

按位与常用于复位(置0)或屏蔽运算。例如：

a = a & 0xff00 的结果就是将低8位都置0，其它位不变。

(3)按位逻辑或“|”对两个操作数逐位进行运算。若对应位都为0，则该位结果为0，否则为1。例如：

```
short int b = 0x12 | 0x3d //结果为 0x3f
```

0x12	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0
0x3d	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	1
b	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1

按位或常用于置位运算。例如：

b = b | 0x00ff 就是将低8位都置1，其它位不变。

(4)按位逻辑异或“^”也是对两个操作数逐位进行运算。异或运算的规则是，若对应位不同，则该位结果为1，否则为0。例如：

```
short int c = 0x5a ^ 0x26 //结果为 0x7c
```

0x5a	0	0	0	0	0	0	0	0	0	1	0	1	1	0	1	0
~0x26	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0
c	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0

按位逻辑异或有一个特点，如果  $a \wedge b = c$ ，那么  $c \wedge b = a$ 。b 将 a 转换为 c，也能将 c 再复原为 a。

显然，两个相等的值异或运算，结果为0。不相等的两个值异或运算结果不为0。

### 3.1.5 条件运算符

C++中唯一的三目运算符就是条件运算符，由两个符号“?”和“:”组成，要求有3个操作数，形式如下：

<表达式 1>?<表达式 2>:<表达式 3>

根据<表达式 1>为真或假，选择<表达式 2>或<表达式 3>作为运算结果。

(1) 条件表达式的执行顺序为：先求解表达式 1，若值为非 0，表示条件为真，则求解表达式 2，此时表达式 2 的值就作为整个条件表达式的结果；若表达式 1 的值为 0，表示条件为假，则求解表达式 3，表达式 3 的值就是整个条件表达式的值。例如：

```
a > b ? a : b
```

结果是 a 和 b 的较大值。

(2) 通常情况下，表达式 1 应该是一个关系表达式或逻辑表达式，表达式 2 和表达式 3 可以是常量、变量或表达式，而且应该是相同类型。例如：

```
x == y ? 'Y' : 'N'
(d = b*b-4*a*c) >= 0 ? sqrt(d) : sqrt(-d)    //sqrt 是开平方函数
ch = ch >= 'A' && ch <= 'Z' ? ch+'a'-'A': ch;  //大写字母转小写
```

(3) 条件表达式的优先级比较低，仅高于赋值和逗号运算符。因此，`min=((a<b)?a:b)`中的括号可以去掉，等效于 `min=a<b?a:b`。

(4) 虽然 C++ 文档说明条件运算符的结合性为“从右向左”，但实际上并非如此。

例 3-2 三目运算符的使用。

```
#include <iostream>
using namespace std;
int main(void){
    int i = 1, j = 2, k = 3;
    cout << "The larger value of : "          //求出 i 和 j 中的最大值
         << i << " and " << j << " is "
         << (i > j ? i : j) << endl;
    int max = (i >= j ? i : j) < k ? k : (i >= j ? i : j); //A
    cout << "The largest value of : "          //求出 i, j 和 k 中的最大值
         << i << ", " << j << " and " << k
         << " is " << max << endl;
    system("pause");
    return 0;
}
```

程序运行后，在屏幕上输出的结果为：

```
The larger value of : 1 and 2 is 2
The largest value of : 1, 2 and 3 is 3
```

上面代码中 A 行使用了多次条件运算符，来计算三个值中的最大值。

### 3.1.6 赋值运算符

赋值运算就是将一个表达式的值赋给一个变量。C++语言提供了两类赋值运算符：基本赋值运算符和复合赋值运算符。基本赋值运算符为“=”，复合赋值运算符有多种形式：+=、-=、\*=、/=、%=、<<=、>>=、&=、^=、|=。例如，`a=3`表示将3赋给变量a，即经过这个赋值后，a的取值变为3，并一直保持，直到下一次将一个新值赋给变量a。

(1) 赋值运算符都是双目运算符，从右向左进行。例如，`sum1=sum2=0`相当于`sum1=(sum2=0)`，先执行`sum2=0`，后执行`sum1=sum2`。

(2) 要求赋值运算符左操作数必须是左值，左值能存储值。例如：

```
x = 3 + 5          //正确，x 是左值
x - 3 = 5          //语法错误，x-3 不是左值
```

(3) 复合赋值运算符是将算术运算或位运算与赋值相结合，同一个变量即参加运算，也是被赋值的变量，出现在赋值运算符的两边。复合赋值运算符是一个整体，中间不能用空格隔开。例如：

```
a *= 6             //相当于 a = a*6
a %= 6             //相当于 a = a%6
a += 3 + 6         //相当于 a = a+(3+6)
```

初学者容易犯的一个错误就是混淆“=”运算符和“==”运算符。分析下面代码：

```
int a = 5, b = 3;
int d = a == b;    //d的值为0
```

### 3.1.7 逗号运算符

逗号“,”既是标点符号(用做分隔符)，又是运算符。逗号运算符用来将两个表达式连接起来，逗号表达式的一般表达形式为：

<表达式 1>,<表达式 2>,<表达式 3>,...,<表达式 n>

逗号运算符是双目运算符，取其右边操作数的值作为整个表达式的结果。逗号运算符的优先级最低。例如：

```
int i,j,k;          //作标点符号用
funx(x,y,z);        //作标点符号用
a+b, a+c            //逗号运算符
1+4, 3.4+10, 3&4    //逗号运算符
x = 3+6, x*3, x+6    //第一个逗号运算结果为 27，第二个逗号运算结果为 15
x = (y=3, y+1)      //y 的值为 3，x 的值为 4
```

例 3-3 逗号运算符的使用。

```
#include <iostream>
using namespace std;
int main(void){
    int a=3, b=7, c=4, t, x;
    t = (a+=1, b-=2, c+3);
    x=3+6,x*3,x+6;
    cout << "a=" << a << '\t';
```

```

cout << "b=" << t << '\t';
cout << "c=" << c << '\t';
cout << "t=" << t << endl;
cout << "x=" << x << endl;
    x=(x=3+6,x*3),x+6;
cout << "x=" << x << endl;
x=(x=3+6,x*3,x+6);
cout << "x=" << x << endl;
x=(x=3+6,x=x*3,x+6);
cout << "x=" << x << endl;
system("pause");
return 0;

}

```

运行程序，输出的结果为：

```

a=4      b=7      c=4      t=7
x=9
x=27
x=15
x=33

```

### 3.1.8 自增、自减运算符

自增、自减运算就是对一个变量加 1 或减 1。自增用“++”运算符，而自减用“--”运算符，它们都是单目运算符。运算符“++”和“--”是一个整体，中间不能用空格隔开。++是使操作数按其类型增加 1 个单位，--是使操作数按其类型减少 1 个单位。

自增(或自减)运算符要区分前缀和后缀两种形式。

- 前缀形式，运算符放在操作数左边，例如++count。先使操作数自增(自减)1 个单位后，然后取其值作为运算结果。
- 后缀形式，运算符放在操作数右边，例如 count++，先取操作数的值参与当前表达式的运算，然后再使操作数自增(自减)1 个单位。

例如：

```

int count = 15, digit = 16, number, amount;
number = ++count;           // count 和 number 都为 16
amount = digit++;           // amount 的值为 16, digit 的值为 17

```

自增、自减运算符要求操作数必须是左值，往往是一个变量。前缀式的计算结果是操作数修改后的值，因此结果仍然是一个左值；而后缀式的值是原先操作数的值，所以它不是左值。如：

```

int amt=63, nut=96;
++ ++amt;           //相当于++(++amt)，结果为 65
++nut--;           //相当于++(nut--)，nut--不是左值，所以语法错误
(++nut)--;         //++nut 是左值，所以语法正确

```

自增、自减运算符具有较高的优先级，高于所有的算术运算符。

例 3-4 自增、自减运算符的使用。

```

#include <iostream>
using namespace std;
int main(void){

```

```

int a = 2, b = 3;
int c, d, e;
c = ++a + b-- ;           //c = 3 + 3
cout<<"a="<<a<<"\t"<<"b="<<b<<"\t"<<"c="<<c<<endl;
d = ++ ++a + ++b;         //d = 5 + 3
cout<<"a="<<a<<"\t"<<"b="<<b<<"\t"<<"d="<<d<<endl;
e = a++ - -- --b;         //e = 5 - 1
cout<<"a="<<a<<"\t"<<"b="<<b<<"\t"<<"e="<<e<<endl;
system("pause");
return 0;
}

```

程序运行后，在屏幕上输出的结果为：

a=3	b=2	c=6
a=5	b=3	d=8
a=6	b=1	e=4

在上面这些运算符中，只有赋值运算符和自增自减运算符可以更改左值，常称为副作用 (side effect)。

应避免在同一个表达式中对同一个变量进行多次自增自减运算，否则结果难以合理解释。

例如：

```

int a = 2;
int b = ++a + ++a;

```

最后 **b** 的值为 8，**a** 为 4。实际上 **b** = 4 + 4，而不是 **b** = 3 + 4。

再如：

```

a = 2;
int c = a++ + ++a;

```

最后 **c** 的值为 6，**a** 为 4。实际上 **c** = 3 + 3，而不是 **c** = 2 + 3。一个比较合理的解释是这里的 + 运算是从右向左计算的(但这违背 + 的结合性，从左向右)，真正执行的是：

```

int c = ++a + a++;

```

注意，尽管后缀自增自减运算符具有比较高的优先级，也不能先计算。例如：

```

int a = 2;
int b = 3 + (a++); //等同于 b = 3 + a++

```

此时变量 **b** 的值是 3+2，而不是 3+3。

应避免在同一表达式中对同一变量多次自增或自减。不同编译器会得到不同结果。例如

```

char c='C';
++c = c++;
cout<<c<<endl;

```

在 VC 序列中结果是 E，而 DevC++ (GCC) 的结果是 D。

### 3.1.9 sizeof 运算符

每种类型的变量都占用一定大小的存储单元。存储单元的大小与变量类型、CPU 及操作系统有关。如 **int** 型变量在 16 位 CPU 中占用 2 字节内存，在 32 位 CPU 中占用 4 字节内存。在程序中，若要获取变量占用的存储单元大小，应使用 **sizeof** 运算符来获取。**sizeof** 运算符用于测试某种数据类型或表达式的类型在内存中所占的字节数，它是一个一元运算符。其语法格式为：

sizeof(<类型名>) 或 sizeof(<表达式>)

例如:

```
sizeof (int)           //整数类型占 4 个字节, 结果为 4
sizeof (3+3.6)         //3+3.6 的结果为 double 实数, 结果为 8
```

例 3-5 sizeof 运算符的使用, 输出各种类型占用的字节数。

```
#include <iostream>
using namespace std;
int main(void){
    cout << "size of bool:" << sizeof(bool) << endl;
    cout << "size of char:" << sizeof(char) << endl;
    cout << "size of wchar_t:" << sizeof(wchar_t) << endl;
    cout << "size of short:" << sizeof(short) << endl;
    cout << "size of int:" << sizeof(int) << endl;
    cout << "size of long:" << sizeof(long) << endl;
    cout << "size of long long:" << sizeof(long long) << endl;
    cout << "size of float:" << sizeof(float) << endl;
    cout << "size of double:" << sizeof(double) << endl;
    cout << "size of long double:" << sizeof(long double) << endl;
    system("pause");
    return 0;
}
```

程序运行后, 在屏幕上输出的结果为:

```
size of bool:1
size of char:1
size of wchar_t:2
size of short:2
size of int:4
size of long:4
size of long long:8
size of float:4
size of double:8
size of long double:8
```

### 3.1.10 typeid 运算符

对于任何一个表达式, 应该知道它的类型究竟是什么。例如:

```
int i = -2;
unsigned j = 3;
cout<<i*j<<endl;           //输出 4294967290
```

表达式 `i*j` 的类型是什么? C++提供了 `typeid` 运算符(称为类型标识 `type identity`), 能获取一个表达式在运行时刻的类型信息。VC6 和 DevC++需要包含 `<typeinfo.h>` 或 `<typeid>`, VS2015 可以不包含。

例 3-6 typeid 运算符的例子。

```
#include <iostream>
using namespace std;
int main(){
    int i = -2;
    unsigned j = 3;
    cout<<i*j<<endl;
    cout<<typeid(i*j).name()<<endl;
```



```
    system("pause");  
    return 0;  
}
```

程序运行后，在屏幕上输出的结果为：

```
4294967290  
unsigned int
```

用“typeid(表达式).name()”可以获取任何一个表达式的类型名称。DevC(GCC)上得到相同结果但以编码形式表示。如果表达式类型为 long long，则显示为\_\_int64。

从上面例子可以知道，一个带符号的 int(可能为负值)与一个无符号的 int 的算术运算的结果是一个无符号 int 值。这种类型转换参见 3.4.1 节。

关于 typeid 运算符还有更多内容，在最后一章详细介绍。

## 3.2 表达式

表达式(expression)是由运算符、括号和操作数(operand)构成的序列，在运行时能计算出一个值的结果。其中的操作数可以是字面值、变量等，也可以是表达式。

表达式的种类很多，分类方法也很多。按运算符的不同，可将表达式分为算术表达式、赋值表达式、关系表达式、逻辑表达式、逗号表达式等等。按表达式能否放在赋值号的左边还是右边，可将表达式分为左值表达式和右值表达式。

表达式按照其中运算符的优先级和结合性来求值。每个表达式都有确定的运算结果(表达式的值)和所属类型(即结果值的类型)，其类型取决于表达式中的运算符和操作数的类型。

### 3.2.1 左值表达式和右值表达式

能放在赋值号左边的表达式称为左值表达式，简称左值(l-value)。左值表达式必须能指定一个存放数据的空间，一般是变量。

右值表达式就是能放在赋值号右边的表达式，简称为右值(r-value)。例如：

```
int bottom, midx;           //定义变量 bottom 和 midx  
bottom                     //bottom 是左值表达式  
bottom + 1                 //bottom+1 不是左值表达式，是右值表达式  
6                           //6 是常量，不是左值表达式，是右值表达式  
int const top = 10;        //说明 const 变量 top  
top                         //top 不能修改，不是左值表达式，是右值表达式  
midx = bottom              //bottom 是右值表达式  
++bottom = 3;              //左值加上前置自增自减，可做左值
```

左值表达式可作为右值表达式，但右值表达式不一定能作为左值表达式。

左值加上前置自增或自减还可做左值，但左值加上后置自增或自减却不能做左值。例如 bottom++不能做左值。

### 3.2.2 算术表达式

由算术运算符、位运算符和操作数构成的表达式称为算术表达式。一个算术表达式利用四则运算和位运算来计算一个值，结果值的类型符合自动类型转换规则。自动类型转换在

### 3.4.1 节介绍。

### 3.2.3 赋值表达式

由赋值运算符和操作数构成的表达式称为赋值表达式。赋值表达式要求赋值运算符左边必须是左值，其功能就是用右值表达式的值来更改左值。赋值表达式的计算顺序是从右向左进行的，运算结果取左值表达式的值。例如：

```
int high, low, midx=20;    //说明 3 个整型变量
high = 7*6                //表达式的值为 high 的值 42
high = low = 0            //相当于 low=0, high=low, 整个表达式的值为 high 的值 0
midx += 3*9               //相当于 midx = midx + 3*9
```

使用复合赋值表达式可使语句表达更简练。对赋值表达式说明以下几点。

(1) 赋值表达式的结果是左值，因此可出现在赋值运算符“=”的左边。例如：

```
(x=5)=23+6                //x=5 是左值，被改为 29，即 x 和整个表达式的值为 29
x=y=z=0                  //先使 z=0，然后再将 z 的值赋给 y，最后将 y 的值赋给 x
```

(2) 说明语句中的符号“=”用来初始化，尽管在书写上与赋值运算符“=”一样，但含义却不同。

```
float radius1 = 5.63f;    //说明变量 radius1，并初始化成 5.36
int sum1 = total1 = 0;    //语法错误，要求分别初始化 sum1 和 total1
float radius;             //说明变量 radius
radius = 5.63f;           //给变量 radius 赋值
```

### 3.2.4 关系表达式

由关系运算符和操作数构成的表达式称为关系表达式。其中的操作数可以是任何类型的表达式。关系表达式的结果为逻辑值，即表达式中的关系成立时，其值为逻辑真(用 1 表示)，否则其值为逻辑假(用 0 表示)。关系表达式通常用来构造简单条件表达式，用在程序流程控制语句中。例如：

```
if(x > 0) y = x;          //如果 x 大于 0，y 取 x 的值
else y = -x;              //如果 x 不大于 0，y 取 -x 的值，最后 y 是 x 的绝对值
```

注意，“=”和“==”含义完全不同，注意不要误写。试比较下面两个程序段：

```
if(x==168) ...            //判断 x 是否等于 168，条件可能成立，也可能不成立
if(x=168) ...             //将值 168 赋给 x，计算结果为逻辑 1，条件永远成立
```

从语法上看，上面两个程序段都没有语法错误。因此，一旦将“==”误写成“=”，编译器不会指出语法错误，但会产生错误结果。

关系表达式中可以包含算术表达式，如  $j < i * i$ 。先计算算术表达式的值，然后再计算关系表达式。

### 3.2.5 逻辑表达式

由逻辑运算符和操作数构成的表达式称为逻辑表达式。其中的操作数应该是逻辑表达式、

关系表达式、算术表达式等。逻辑表达式的运算结果为逻辑值，一般用来构造比较复杂的条件表达式。

逻辑表达式中往往包含多个关系表达式，要先计算关系表达式的值，再计算逻辑表达式。但逻辑与(&&)和逻辑或(||)运算符在运算时，从左向右计算，根据规则一旦能确定整个表达式的值时，就结束计算，后面的表达式就不再计算。

(1) 对于<条件 1> && <条件 2>逻辑表达式，先计算<条件 1>的值，若其值为逻辑真(非 0)，就再计算<条件 2>的值，若其值为非 0，则结果为 1，否则为 0。若<条件 1>的值为逻辑假(值为 0)，就已经知道整个表达式为假，就不再计算<条件 2>的值，结束整个表达式的计算。

例如：

```
year % 4 == 0 && year % 100 != 0
```

该表达式判断闰年的一种情形。先计算 year 是否能被 4 整除，如果不能被整除，整个表达式的值就是假，&&后面表达式不再计算，例如 year=2001 就不是闰年。如果 year 能被 4 整除，再计算 year 是否能被 100 整除，如果能被 100 整除，表达式的值为假，例如 year=1900 就不是闰年。如果 year 不能被 100 整除，整个表达式的值为真，例如 year=2004 就是闰年。

用&&来连接两个条件，前一个条件满足是后一个条件计算的前提。

(2) 对于<条件 1> || <条件 2>逻辑表达式，先计算<条件 1>的值，若其值为假，再计算<条件 2>的值，并将<条件 2>的值作为整个表达式的值。若<条件 1>的值为真，就已经知道整个表达式的值为真，就结束整个表达式计算。例如：

```
year % 400 == 0 || year % 4 == 0 && year % 100 != 0
```

该表达式判断 year 是否为闰年。先计算 year%400 是否为 0。如果 year 能被 400 整除，整个表达式的值就是真，||后面的表达式不用再计算，例如 year=2000 就是闰年。若 year 不能被 400 整除，再计算后面表达式。

用||来连接两个条件时，前一个条件不满足应该是后一个条件计算的前提。

在一个逻辑表达式中，应避免出现有副作用的表达式，如赋值表达式或自增、自减运算符。因为这些表达式并不一定执行。例如，

```
int a=1,b=2,c=3,d=4;
bool x = a>b && b-->c && c < ++d; //因 a 不大于 b，所以不执行后面的 b--和++d
bool y = c>a || (d += 5);          //因 c 大于 1，所以不执行后面的 d+=5
```

### 3.2.6 逗号表达式

由逗号运算符和操作数组成的表达式称为逗号表达式，就是用逗号将多个表达式分隔起来，从左向右逐个计算各个表达式，并将最右边的表达式作为逗号表达式的值。例如：

```
int p,w,x=8,y=10,z=12;
w = (x++, y, z+3)-5;      //w 的值为 (z+3)-5，即 10
p = w+5, y+x, z;          //p 的值为 w+5，即 15，z 的值 12 作为逗号表达式的值
```

如果逗号运算符的右操作数是一个左值，该逗号运算的结果也是左值。例如：

```
(y++, z) = 2              //z 是左值，(y++, z)的结果也是左值，z 的值被改为 2
```

一个逗号表示式中所包含的各个表达式都应具有副作用，应包含赋值表达式或自增自减

表达式。如果没有副作用，这个表达式就没有实际用途。如上面  $p = x+5, y+x, z;$ ，有用的只是第一个表达式  $p = x+5$ 。没有副作用的表达式应该在逗号表达式中删去。

### 3.2.7 表达式语句

任何一个表达式后加上一个分号“;”就构成一条表达式语句。表达式语句的一般格式为：

<表达式>;

注意，分号是构成语句的组成部分，而不是作为语句之间的分隔符。例如：

```
x = 20;
3 * 8 - 9;
y = a + b * c;
i = 0, j = 0;
i++;
```

这些都是合法的表达式语句，第二个表达式语句只做了算术运算，并没有保存结果，即没有副作用，在程序中应消除这些没用的表达式语句。

## 3.3 类型转换

C++是强类型语言。每个变量或表达式都具有明确的类型。在一个表达式中，运算符的某个操作数如果不符合类型要求，就要对操作数进行类型转换。C++中的类型转换有3种：自动类型转换、赋值转换和强制类型转换。前两种属于隐式类型转换，后一种属于显式类型转换。

### 3.3.1 自动类型转换

自动类型转换是在表达式计算中自动地将操作数转换成特定类型。这种转换的规则是从范围较小的类型向范围较大的类型转换。如果在一个表达式中出现不同数据类型(类型必须兼容)的数据进行混合运算时，系统将利用该转换规则先转换，然后再进行运算。

**规则 1。**两个不同类型的操作数进行算术运算时，先将较小范围的数值转换为另一个较大范围的数值，然后再进行计算。各种基本数据类型的数值范围从小到大排列次序如下：

`char, bool → short → int → long long → float → double`

`unsigned char → unsigned short → unsigned int → unsigned long long → float → double`

例如定义两个变量 `a` 和 `f`：

```
int a = 100;
float f = 32.2f;
```

则计算以下表达式：

`a / f`

处理过程为：先将 `a` 的值转换成 `float` 型，然后再进行浮点数的除法运算，结果为一个 `float` 值 3.10559。在这个过程中 `a` 变量的值不改变。

分析下面两行输出结果是否一样。

```
short s1 = 32765;
cout<<s1 + 3<<endl;
s1 = s1 + 3;           //A
cout<<s1<<endl;
```

第1行输出为32768, 由于3是int型面值, short加上int结果为一个int, 32765+3=32768, 它是int型的合法值。

第2行输出的是short型值, A行中将一个int值赋给了一个short, 就可能丢失数据, 但编译器可能仅给出警告。实际上32768对于short是超界了, 变成一个负值-32768。

按规则1可知, 两个有符号的值之间进行算术运算, 其结果是有符号的。一个无符号的值与一个浮点数(如float)进行算术运算, 其结果是浮点数(如float)。但两个无符号的值之间进行算术运算(两个值中没有unsigned int), 其结果是有符号的int。例如:

```
unsigned char c1 = 2;
unsigned short s1 = 3;
cout<<(c1 * s1)<<endl;           //乘法, 输出6
cout<<typeid(c1 * s1).name()<<endl; //输出 int
```

**规则2.** 对于bool、char、short、int类型, 任意两个值之间进行算术运算、位运算, 其结果都是一个int值。任意两个值之间进行逻辑运算, 其结果都是一个bool值。例如:

```
char c1 = -2;
short s1 = 3;
cout<<(c1 * s1)<<endl;           //乘法, 输出-6
cout<<typeid(c1 * s1).name()<<endl; //输出 int
cout<<(c1 & s1)<<endl;           //按位与, 输出2
cout<<typeid(c1 & s1).name()<<endl; //输出 int
cout<<(c1 && s1)<<endl;          //逻辑与, 输出1, 表示true
cout<<typeid(c1 && s1).name()<<endl; //输出 bool
```

**规则3.** 对于bool、char、short、int类型, 任一个类型值(无论是否带符号)与unsigned int之间进行算术运算, 其结果都是unsigned int类型。例如:

```
char c1 = -2;
unsigned short s1 = 2;
unsigned int j = 3;
cout<<(c1 * j)<<endl;           //乘法, 输出4294967290, 而不是-6
cout<<typeid(c1 * j).name()<<endl; //输出 unsigned int
cout<<(s1 * j)<<endl;           //乘法, 输出6
cout<<typeid(s1 * j).name()<<endl; //输出 unsigned int
```

由上面例子可知, unsigned int被系统认为是整型数值中的最大范围的类型, 因此其它类型与之计算时, 都要转换到unsigned int。此时如果另一个整数恰好是负值, 而结果是不带符号的正值, 就不能得到预期结果, 但二进制结果是正确的, 例如4294967290与-6是一样的。

在处理表达式的过程中, 并不是将变量直接转换成最大范围的类型, 而是在表达式处理过程中, 按照需要逐步进行转换。例如:

```
int i=1; char ch=2;
float f= 3.0f; double df = 4.0;
cout<<(ch*i+f*2.0-df)<<endl;
```

表达式ch\*i+f\*2.0-df的计算过程为:

(1) 将ch转换为int型, 计算ch\*i, 即2\*1, 结果为2, 类型为int。

- (2) 将 `f` 转换为 `double` 型, 计算 `f*2.0`, 即 `3.0*2.0`, 结果为 `6.0`, 类型为 `double`。
- (3) 将 `ch*i` 的结果 `2` 转换为 `double` 型, 计算 `2.0+6.0`, 结果为 `8.0`, 类型为 `double`。
- (4) 计算 `8.0-df`, 即 `4.0`, 整个表达式的结果为 `4.0`, 类型为 `double`。

自动类型转换的基本规则是“宽化”或者“提升”, 即将较小范围的数值类型转换到较大范围的数据类型。大多数自动类型转换是安全的。

### 3.3.2 赋值转换

赋值类型转换出现在初始化表达式或者赋值表达式中。当初始化或赋值运算符的左值表达式的类型与右值表达式类型不同, 且类型兼容时, 将进行类型转换到左值类型。先计算出右值表达式的值, 然后将其转换为左值类型后再赋给左值。例如:

```
char ch = 'b';
short s = ch;           //char 转换到 short
cout << s << endl;      //输出: 98

int i = 4000000;
s = i;                  //int 转换到 short
cout << s << endl;      //输出: 2304, 因为 i 的值超出 s 的存储范围, 发生截断

float f = i;            //int 转换到 float
cout << f << endl;      //输出: 4e+006

double d = 23.56;
i = d;                  //double 转换到 int
cout << i << endl;      //输出: 23, 因为整型变量不能存放小数部分
float t = 34;
i = t;                  //float 转换到 int
cout << i << endl;      //输出: 34
```

赋值转换也发生在函数调用时。当函数调用的实参类型与函数定义的形参类型不同且类型兼容时, 将转换到形参类型。赋值转换也发生在函数返回时。当用 `return` 语句返回的表达式类型与函数定义的返回类型不同且类型兼容时, 将转换到返回类型。将在第 5 章介绍。

对于基本数据类型, 任意两种类型之间都可以进行赋值转换, 但应注意下面情形:

- 将一个 `double` 数值转换为 `float` 时, 其指数部分和小数部分将缩小到 `float` 可表示的范围。如果原先 `double` 值超过了 `float` 类型可表示的范围, 将不能得到正确结果。
- 将一个浮点数转换为整数时, 将去掉小数部分。如果原先浮点值大于整数可表示的值的范围, 将不能得到正确结果。
- 将一个较大范围的整数(例如 `int`)转换为较小范围的整数(如 `short`), 将截断高位字节, 仅保留低位字节的值。如果原先数值大于小范围整数类型可表示的范围, 将不能得到正确结果。
- 将一个较小范围的整数(如 `char`)或浮点数(如 `float`)转换为较大范围的整数(如 `int`)或 `double` 型, 将保持原值不改变。

一般情况下, 编译器对于可能导致数据丢失的情形会给出警告, 但不完全。不经意之间就可能产生意料不到的结果。例如:

```
int i = 2, j = 4;
double df;
df = i/j*100;           //i/j 的值为 0, 而不是 0.5
```

```
cout << "df=" << df << '\t';    //输出 0, 而不是 50
i = 4.6, j = 5.7;                //编译时给出警告
float x = i + j;                  //x 的值并不是 10.3, 而是 9 = 4 + 5
cout << "x=" << x << '\n';      //输出 9
```

### 3.3.3 强制类型转换

强制类型转换(也称显式类型转换)是用类型转换运算符明确指明的一种转换操作, 将一个表达式强制转换到某个指定类型。强制类型转换的一般形式为:

<目标类型名>(表达式)

或者

(目标类型名)<表达式>

例如:

```
int a = 7, b = 2;
double y1 = a / b
```

此时 y1 的值是 3.0。如果希望得到 3.5, 就要对除法的操作数进行强制类型转换如下:

```
y1 = double(a)/b 或者 y1 = (double)a/b;
```

该语句的计算过程为:

(1) 先将 a 的值强制转换成 double 型, 由于除(/)的两个操作数类型要一致, 因此系统将 b 自动转换为 double 型, 最后进行浮点数的除法, 结果为 3.5。

(2) 将 3.5 赋值给 y2。

注意, y1 = double(a/b) 将得到 3.0, 这是因为先做整数除法得到 3 之后再转换到 double 类型。

对于基本数据类型之间的强制类型转换, 将按照前面赋值转换的规则进行。

关于强制类型转换, 说明以下两点。

(1) 一个强制类型转换是否正确取决于所处理的值的范围, 一般来说, 强制转换是不安全的。

(2) 类型强制转换作用于一个表达式, 并非作用于数据存储单元, 即不改变变量的类型和值。例如:

```
double width = 2.36, height = 5.5, areal;
int area2 = int(width)*int(height); //areal 值为 10, width 和 height 仍为 double
型
areal = width*height;                //area2 值为 12.98
```

何时需要进行强制类型转换? 只有当自动类型转换和赋值转换都不能达到目的时, 才使用强制类型转换。当要从整数除法得到浮点数结果时, 就应先将整数转换为浮点数。一般来说, 强制类型转换是不安全的, 只有在特定条件下执行才比较安全, 所以应谨慎使用。

上面介绍的强制类型转换是 C 语句的传统方式, C++ 标准扩充了 4 种新的类型转换: xxx\_cast, 以取代传统方式, 在最后一章介绍。

### 3.4 小 结

- 运算符对操作数进行运算。操作数可以是变量、字面值，也可以是一个表达式。
- 表达式由操作符和操作数按一定规则组成。表达式根据某些约定、求值次序、结合性和优先级来计算。每个表达式在执行时能得到一个确定的值，该值具有确定的类型。
- 算术运算应注意溢出问题。
- 关系运算的结果应该是逻辑值。对浮点数的相等或不等的判断应注意不是绝对相同或不同。
- 逻辑运算应作用于逻辑值。在逻辑与&&计算时，注意第一个条件满足是第二个条件执行的前提，而在逻辑或||计算时，第一个条件不满足是第二个条件执行的前提。
- 移位运算中，左移位可能改变数值的符号。右移位时最高位用符号位补齐，即保持符号不改变。按位逻辑运算不涉及符号位。
- 条件运算符是唯一的三目运算符。
- 赋值运算符是最常用的运算符。从右向左计算。复合赋值运算是将算术运算或位运算与赋值相结合。
- 逗号运算符是优先级最低的运算符，最后一个表达式的值作为整个表达式的值。
- 自增、自减运算要注意区分前缀和后缀两种形式。前缀形式先使操作数自增(自减)1个单位后，然后取其值作为运算结果，而后缀形式则是先取操作数的值参与当前表达式的运算，然后再使操作数自增(自减)1个单位。注意不要在同一个表达式中对同一个变量进行多次自增、自减运算。
- sizeof 运算符用来获取一个表达式或类型的存储字节大小。
- typeid 运算符用来获取一个表达式的类型名称。应包含#include <typeinfo.h>
- 左值表达式是能放在赋值号左边的表达式；能放在赋值号右边的表达式称为右值表达式。左值表达式可作为右值表达式，但右值表达式不一定能作为左值表达式。
- 一个表达式的数据类型可转换为其它类型。类型转换包括自动类型转换、赋值类型转换及强制类型转换。
- 自动类型转换(即隐式转换)的基本规则是“宽化”或者“提升”，即将较小范围的数值类型转换到较大范围的数据类型。自动转换一般是安全的。
- 赋值类型转换出现在初始化表达式或者赋值表达式中。先计算出右值表达式的值，然后将其转换为左值类型后再赋给左值。
- 强制类型转换(即显式转换)是编程控制的一种转换，基本规则是“窄化”，即将较大范围的数值类型转换到较小范围的数据类型。只有当自动转换和赋值转换不能达到目的时，才使用强制转换。

### 3.5 练 习 题

1. 下面哪一个运算符要求操作数都是整型？



A /      B <=      C %=      D =

2. 设有说明语句: double x, y; 则表达式  $x=3, y=x+5/3$  的值是\_\_\_\_\_。

A 4.66667      B 4      C 4.0      D 3

3. 假设变量 a、i 已正确定义, 且 i 已正确赋值, 下列哪一个合法的赋值表达式?

A a==1      B a=++i++      C a=a+=5      D a=int(i)

4. 设有语句: int a=13, b=9, c; 执行  $c = a / b + 0.8$  后, c 的值为\_\_\_\_\_。

A 1.8      B 1      C 2.24444      D 2

5. 若变量 a 是 int 类型, 并执行了语句  $a = 'A' + 1.6$ ; 下列哪一个叙述是正确的?

A a 的值是字符 A      B a 的值是浮点型  
C 不允许字符型与浮点型相加      D a 的值是字符 'B'

6. 变量 x、y 和 z 均为 double 型且已正确赋值, 下面哪一个表达式不能正确表示数学式  $\frac{x}{y \times z}$  ?

A  $x * (1 / (y * z))$       B  $x / y * z$       C  $x / y * 1 / z$       D  $x / y / z$

7. 设有语句 int a=5; 则执行表达式  $a-=a+=a*a$  后, a 的值是\_\_\_\_\_。

A -5      B 25      C 0      D -20

8. 表达式  $16/4*\text{float}(4)+2.0$  的数据类型是\_\_\_\_\_。

A int      B float      C double      D 不确定

9. 设有语句: int m=13, n=3, 则执行  $m \% = n + 2$  后, n 的值是\_\_\_\_\_。

A 5      B 1      C 3      D 0

10. 设有语句 int a=5, b=6, c=7, d=8, m=2, n=2, 则逻辑表达式  $(m=a>b) \&\& (n=c>d)$  运算后, n 的值为\_\_\_\_\_。

A 3      B 2      C 1      D 0

11. 设有语句: int x=8, float y=8.8; 下列表达式中错误的是\_\_\_\_\_。

A  $x\%3+y$       B  $y*y\&\&++x$       C  $(x>y) + (\text{int}(y)\%3)$       D  $---x+y$

12. 整型变量 m 和 n 的值相等, 且为非 0 值, 下面哪一个表达式的值为零?

A  $m | n$       B  $m \wedge n$       C  $m || n$       D  $m \& n$

13. 下面哪一个表达式能正确表示逻辑关系: "age $\geq 18$  或 age $\leq 60$ "?

A  $\text{age} \geq 18 \text{ or } \text{age} \leq 60$       B  $\text{age} \geq 18 | \text{age} \leq 60$   
C  $\text{age} \geq 18 \&\& \text{age} \leq 60$       D  $\text{age} \geq 18 || \text{age} \leq 60$

14. 下列程序的运行结果是\_\_\_\_\_。

```
int main(void){
    int a = 9, b = 2 ;
    float x = 6.6f, y = 1.1f, z;
    z = a/2 + b*x/y + 1/2;
    cout<<z<<endl;
    return 0;
}
```

A 17      B 15      C 16      D 18

15. 下列程序的运行结果是\_\_\_\_\_。

```
int main(void)
{ int a=5,b=4,c=3,d; d = (a > b > c); cout<<d<<endl; return 0;}
```

A 5      B 3      C 1      D 0

16. 下列关于类型转换, 下面哪一种说法是错误的?

- A 赋值表达式的类型是左值的类型
- B 逗号表达式的类型是最后一个表达式的类型
- C 在算术表达式中不同类型数据参与运算将自动转换类型
- D 在不同类型数据组成的表达式中, 其表达式类型是 double 型

17. 在算术表达式中, 下面哪一种类型转换是错误的?

- A 一个 int 值加上一个 float 值的类型为 float。
- B 两个 unsigned char 值相加的类型为 int。
- C 一个 char 值加上一个 short 值的类型为 int。
- D 一个 unsigned int 值加上一个 int 值的类型为 int。

18. 给出下面程序的输出结果。

```
int main(){
    int a = 10, b = 9, c = 8;
    c = a -= ( b - 6 );
    c = a % 8 + (b = 5);
    cout<<"a"<<a<<endl;
    cout<<"b"<<b<<endl;
    cout<<"c"<<c<<endl;
    return 0;
}
```

19. 给出下面程序的输出结果。

```
int main(){
    int x,y,z;
    x = y = 5;
    z = ++x || ++y;
    cout<<"x"<<x<<endl;
    cout<<"y"<<y<<endl;
    cout<<"z"<<z<<endl;
    return 0;
}
```

20. 给出下列程序的运行结果\_\_\_\_\_。

```
int main(void){
    int a=1,b=2,c;
    cout<<"a=1,b=2"<<endl;
    cout<<"-(c=a==1):"<<-(c=a==1)<<endl;
    cout<<"c=a>=b:"<<(c=a>=b)<<endl;
    return 0;
}
```

21. 设 x 是 int 型变量, 请写出判断 x 为大于 2 的偶数的表达式。

22. 设 x 是 int 型变量, 请写出判断 x 的绝对值大于 8 的表达式。

23. 分别给出下列表达式的值。

- A 1/3                      B 1/3.0                      C 1%3                      D 21/3

24. 设有语句: int a=9,b=9,c=9; 分别给出下列表达式的值。

- A a/=2+b++-c++      B a+=b+c++      C a-=++b-c--      D a\*=b+c--

25. 分别给出下列表达式的值。

- A !('5'>'8')||3<9;      B 6>3+2-('0'-8)

C  $3*5|6<<2$

D  $'a'=='b'<=3\&5$

26. 根据题目要求, 编写完整的程序。

- (1)从键盘上输入一些组合键, 如 Ctrl+B、Ctrl+D、Ctrl+Y 等, 如何能知道这些键的值。
- (2)从键盘上输入一个整数  $x$ , 将其低六位全部置 1, 其余各位不变。然后用十进制和十六制输出  $x$  的值。
- (3)从键盘上输入一个整数  $x$ , 将其高六位全部置 0, 其余各位不变。然后用十进制和十六制输出  $x$  的值。
- (4)从键盘上输入一个摄氏温度  $C$ , 求该温度对应的华氏温度  $F(F=5/9C+32)$  及绝对温度  $K(K=C+273.15)$ 。
- (5)从键盘上输入任意两个实数, 求它们的和、差、积、商, 并输出结果。
- (6)输入任意一个实数, 将其整数部分与小数部分分开, 分别输出在两行上。例如, 输入 23.45, 输出:

23

0.45

## 第4章 基本语句

一个 C++ 程序可以由若干个源程序文件组成，一个源程序文件由编译预处理指令、自定义类型说明和函数组成，一个函数由若干条语句组成。语句是组成程序的基本单位。本章介绍 C++ 基本语句的分类、程序结构，详细介绍其中的流程控制语句，包括选择语句、循环语句和跳转语句等。

### 4.1 语句分类

程序是由语句构成的。每一条语句都用一个分号结尾。C++ 语句可分为以下 9 大类。

#### 4.1.1 说明语句

程序中往往要引入新的名称，来表示某个类型的变量或命名常量，也可能表示用户自定义的某种类型，如结构类型、枚举类型、类等。所有这些引入新名字的语句统称为说明语句（也称为声明语句）。也可以将说明语句再详细划分为类型说明语句、变量或对象说明语句。说明语句在程序执行过程中并没有对数据进行任何操作，仅是向编译器提供一些说明性信息。说明语句可出现在函数中，也可以出现在函数之外。

变量说明语句就是最常见的说明语句。例如：

```
int i = 3;
```

这是一条说明语句。变量 *i* 是前面不曾出现的变量。该语句说明了一个新变量 *i*，后面的代码就能对这个变量 *i* 进行操作。

在后面章节还将看到，说明语句可用于说明函数原型、结构、类等。

注意编程中的说明与定义的区别。说明可以是笼统的，比如说明一个函数原型、一个类原型，说明可以不包含具体内容，比如不包含函数体，不包含类中的成员。说明也可以是具体的，而定义必须是具体的，当我们说定义一个函数时，不仅要确定函数原型，也要确定函数体。

#### 4.1.2 表达式语句

在任一表达式后面加上一个分号就构成一条表达式语句。表达式语句的作用是执行计算。变量说明语句和表达式语句相结合，完成计算过程。例如：

```
int i = 3;    //这是说明语句，而不是赋值语句
```

```
i = 4;       //这是赋值语句，是一种表达式语句
```

赋值语句、逗号语句、自增自减语句都是常见的表达式语句。

一个函数调用可作为一个操作数，是表达式的一部分，所以函数调用语句也是一种表达

式语句。后面章节将详细介绍。

### 4.1.3 选择语句

程序中往往要根据某个条件来执行不同的代码。选择语句就是先计算一个条件，如果为真就执行特定的一段代码，如果条件为假，还可以执行另一段代码。有两种选择语句：`if-else` 和 `switch` 语句。

### 4.1.4 循环语句

编程中还经常要重复执行一段代码，直到满足特定的停止条件为止。有三种循环语句：`while`、`do-while` 和 `for` 语句。

### 4.1.5 跳转语句

一般情况下，程序自上向下逐条语句执行，形成一条控制流。但有时需要在一个函数中将控制流跳转到另一个地方开始执行，或者从一个函数中将控制流返回到调用方。跳转语句有 `break`、`continue`、`return`、`goto` 语句。`C++` 扩充的 `throw` 语句用来引发异常，导致控制流跳转到异常处理语句，也属于一种跳转语句，将在后面章节详细介绍。

选择语句、循环语句和跳转语句也被统称为流程控制语句。本章后面将详细介绍这三类语句。

### 4.1.6 空语句

只由一个分号“`;`”构成的语句称为空语句，它不执行任何动作，主要用于满足特殊语法要求。例如，循环语句中需要一条语句作为循环体，但你又不想做任何事情，就可用一个空语句。本章后面将会用到空语句。

### 4.1.7 复合语句

复合语句(也称块语句 `block` 或者分程序)，是用一对花括号 `{ }` 把一条或多条语句括起来构成的一条语句。一条复合语句可包含多条语句，但在逻辑上被看作一条语句，可以出现在任何需要一条语句的地方。

花括号是一种标点符号，左花括号表示复合语句的开始，右花括号表示复合语句的结束。复合语句不需要分号来结束。复合语句主要用在流程控制语句中。

一个函数体的开始和结束也需要一对花括号 `{ }`，但一般不能把一个函数体作为一条复合语句。

复合语句的花括号 `{ }` 区间构成一个作用域，其中可以定义变量，将在下一章介绍。

显然，一条复合语句内部可嵌套多条复合语句，嵌套层次的数目没有明确限制。

### 4.1.8 异常处理语句

在执行代码过程中，可能会引发某些类型的异常(用 `throw` 语句引发异常，或者因某种错误引发异常)，就需要捕获这些异常类型并加以适当处理，这样的语句就是异常处理语句。这是 `C++` 扩充的一种新语句，主要是 `try-catch` 语句，将在第 15 章详细介绍。

### 4.1.9 标号语句

前面介绍的 `goto` 跳转语句需要对函数中的某一条语句添加标号，这样使 `goto` 语句能根据标号的名称跳转到该语句开始执行。

## 4.2 程序的基本结构

结构化编程有 3 种基本结构：顺序结构、选择结构和循环结构。每一种基本结构都由若干模块组成，一个模块是一条复合语句的抽象。一个函数就是一个模块，下一章介绍函数。每一种结构都是一个模块，而且每个模块都能在内部嵌套更小的多个模块。

每个模块的设计应遵循以下原则：

- 每个模块都应该是单入口、单出口。
- 每个模块都有机会执行，即不能让一些模块永远都不能执行。
- 模块中不能有死循环，即不能只进不出。

### 4.2.1 顺序结构

顺序结构的程序是按从上到下的顺序依次执行各个模块。如图 4.1 所示，一个模块中嵌套了两个模块，先执行 A 模块，再执行 B 模块。

单入口要求，若 A 模块执行，B 模块就一定执行。即不允许在两个模块之间插入另一个入口，只执行 B 模块而不执行 A 模块。这样保证 B 模块执行之前，A 模块一定执行完毕。

单出口要求，当退出顺序结构时，A 模块和 B 模块都执行完毕。

如果在 A 模块中有一条 `goto` 语句，就可能导致 B 模块不能执行。如果在 B 模块中有标号语句，就可能使外部 `goto` 语句能跳转进入 B 模块执行，而 A 模块不能执行。这样就违背了单入口、单出口的原则。

在 A 模块和 B 模块中，并非禁止所有的跳转。在任何模块中都允许执行以下两种跳转语句：

- `return` 语句：将当前函数的执行返回到调用方。
- `throw` 语句：引发异常到外层的异常捕获处理语句。

以上两种跳转语句都有确定的目标，而且一般情况下只有在满足特定条件时，才使用这两种跳转语句，而不是无条件跳转，所以经常会用到这两种跳转语句。

顺序结构在逻辑上比较简单，容易理解，往往用来描述高层的执行顺序。例如，输入数据、计算结果、输出结果，三个模块按次序执行。每个模块都可以递归地嵌套内部子模块，而每个模块都应该是三种基本结构之一。

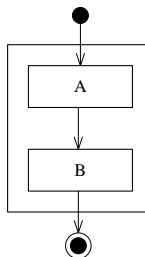


图 4-1 顺序结构

### 4.2.2 选择结构

选择结构是根据条件的不同结果做出不同的选择,从而执行不同的语句。如图 4.2 所示。先计算一个条件,如果为真就执行 A 模块,否则就执行 B 模块(或者什么都不执行,图右面一种情形)。

单入口要求,如果 A 模块能执行,条件就一定为真。不允许出现这样的情形,执行了 A 模块,而没有执行条件判断,或者条件判断不为真。单出口要求,在退出选择结构时,要么 A 模块执行,要么 B 模块执行。

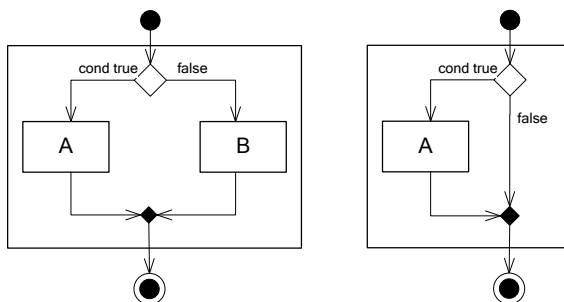


图 4-2 选择结构

如果 A 模块中有一条 goto 语句跳转到 B 模块中,或者跳转到其它地方,就会破坏单入口单出口的原则。

两个模块都应有机会执行,即该条件不应恒为真或恒为假。

实现选择结构的语句有 if-else 语句和 switch 语句。

### 4.2.3 循环结构

循环结构就是按照一个条件对某些语句重复执行多次。一个循环结构由一个条件和一个

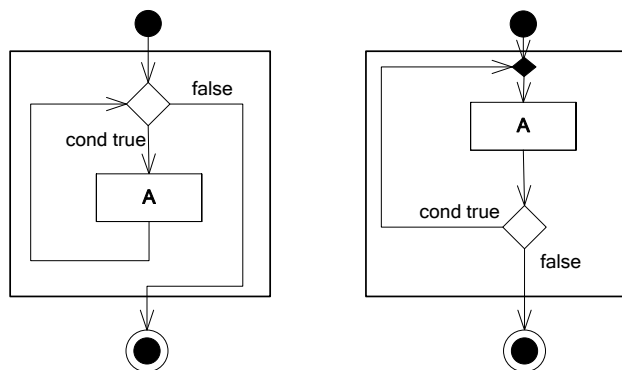


图 4-3 循环结构

循环体(表示为模块 A)构成。如图 4.3 所示。有左右两种情形:

情形 1。先判断条件,若为真就执行模块 A,然后再转去判断条件,直到条件为假,退出循环结构。这种情形对应 while 语句和 for 语句。

单入口要求,模块 A 执行,条件必须满足。即不允许外部直接跳转到模块 A 开始执行。单出口要求,当退出循环结构时,条件一定为假。即不允许在退出循环结构时,条件没有判

断或者条件为真的情形。

条件如果恒为真，而且模块 A 中没有跳转语句，就形成了死循环，不能退出循环结构。反之，条件如果恒为假，模块 A 将没有机会执行。这样都将违背模块设计的原则。

情形 2。先执行模块 A，然后再判断条件，如果为真，就再执行模块 A，直到条件为假，退出循环结构。这种情形对应 do-while 语句。

模块 A 至少执行一次。单入口要求，模块 A 再次执行，条件必须满足。单出口要求，当退出循环结构时，条件一定为假。

条件如果恒为真，而且模块 A 中没有跳转语句，此时形成死循环，不能退出循环结构。反之，条件如果恒为假，模块 A 将执行一次。

对于以上两种情形，如果在模块 A 中有 goto 语句跳转到结构之外，或者模块 A 中有标号语句使外部 goto 语句能跳转到模块 A 中执行，都会破坏单入口单出口原则。

循环结构是最强有力的结构，可以让程序按条件自动执行，以得到预期结果。同时循环结构也是最复杂的结构，它可能导致死循环、甚至程序崩溃。

循环结构可以嵌套，即 A 模块中可包含更小的循环结构或者其它结构。

循环结构中，模块 A 中允许执行除 goto 之外的其它跳转语句。例如：

- break 语句：结束循环，退出当前循环结构。
- continue 语句：结束本轮循环，跳转到当前循环结构的条件判断，尝试下一轮循环。
- return 语句：当前函数的执行返回到调用方。
- throw 语句：引发异常到外层的异常处理语句。

顺序结构、选择结构和循环结构都是基本结构。一个程序应该仅由这三种基本结构嵌套形成，不允许出现 goto 语句。这是结构化编程的最基本要求。结构化方法还强调“自顶向下，逐步求精”的设计过程。就是先从小处着眼，根据需求先描述一个抽象的过程框架，然后再对其中每一步进行分解、细化，逐步得到编码。另外，结构化设计还有“高内聚、低耦合”的设计原则，在后面介绍。

## 4.3 选择语句

选择语句又称为分支语句，如果在程序中需要根据不同的条件来决定执行何种操作，就要使用选择结构。C++语言提供了两种选择结构语句：条件语句(if 语句)和开关语句(switch 语句)。

### 4.3.1 条件语句

条件语句又称为 if 语句，根据一个条件来决定是否执行某条语句，或者从两条语句中选择一个执行。

#### 1. if 语句

if 语句的语法格式为：



```
if(<表达式>
    <语句>
```

其中，<表达式>的值作为条件，如果计算值为非 0，即为逻辑真，就执行<语句>，否则就执行下一条语句。执行过程如图 4.4 所示。条件表达式可以是任何表达式，但常用的是关系表达式或逻辑表达式；作为循环体的语句可以是单条语句，也可以是一条复合语句，即用花括号括起来的多条语句。

例如：

```
int a = 1, b = 2, c = 0;
if (a > b)
    c = a - b;
cout << c << endl;
```

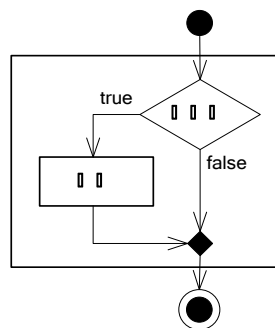


图 4-4 if 语句的执行过程

在该程序中，if 语句会判断  $a > b$  这个表达式的运算结果是否为逻辑真，若是，则执行  $c = a - b$  语句。如果结果为逻辑假，则会跳过语句  $c = a - b$ ，转而执行下面的 `cout << c << endl` 语句。所以输出的  $c$  值是 1，而不是 0。

当条件满足时所执行的一条语句可以是复合语句。例如：

```
float score = 0;
cout << "Please Enter Score: ";
cin >> score;
if (score > 100 || score < 0){
    cout<<score<<" is invalid. 0-100 is valid!"<<endl;
    return;
}
```

上面代码检查输入的分数，合理的分数假设为 0 到 100 之间。如果输入的分数 `score` 不在合理范围之内，就要执行下面的复合语句中的 2 条语句。

为了使程序方便阅读，在书写条件语句时，满足条件所执行的语句都缩进一个 `tab` 位置，只有在语句很短时才与 if 条件写在同一行上。

## 2. if-else 语句

if-else 语句的语法格式为：

```
if(<表达式>
    <语句 1>
else
    <语句 2>
```

其中,<表达式>的值作为条件,如果计算为非0,即为逻辑真,就执行<语句1>,否则就执行<语句2>。它们可以是一条语句,也可以是一条复合语句。执行过程如图4.5所示。

例如:

```
int x;
cout << "input x=";
cin >> x;
if (x % 2 == 0)
    cout<<x<<" is even!"<<endl;
    //能被2整除,则为偶数
else
    cout<<x<<" is odd!"<<endl;
    //否则,为奇数
}
```

程序中条件语句判断x的值是奇数还是偶数。若x的值能被2整除,输出x是偶数(even);否则输出x是奇数(odd)。

在进行条件判断时,往往使用C语句的一个潜规则:“非0为真”。例如上面if语句等价于下面写法:

```
if (x % 2)                //等价于 x % 2 != 0
    cout<<x<<" is odd!"<<endl;    //不能被2整除, x为奇数
else
    cout<<x<<" is even!"<<endl;    //能被2整除, x为偶数
```

上面直接把算术表达式作为条件,就是将所有非0值都看作逻辑真。我们建议还是用关系表达式或逻辑表达式作为条件比较直接、清晰可读。

if-else是最常用的条件语句,用一个条件形成了两个分支。为了使程序方便阅读,一般将“else”与“if”书写在同一列上,在垂直方向对齐,而且将语句1和语句2缩进一个tab位置。

### 3. if 语句的嵌套

对于if-else语句,语句1或语句2又可以是一条if-else语句,这就形成嵌套的if语句。例如:

```
if (表达式1)
    <语句1>
else
    if(表达式2)
        <语句2>
    else
        <语句3>
```

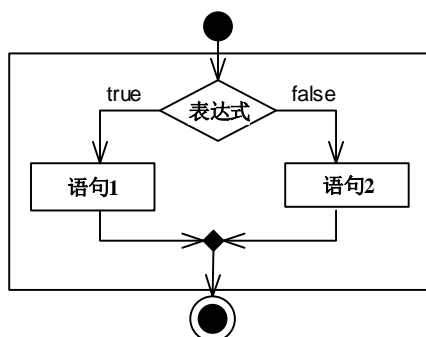


图 4-5 if-else 语句的执行过程

上面是一条 if-else 语句的结构，只是其中 else 部分嵌套了一条 if-else 语句。这种结构的执行过程如图 4.6 所示。这种结构用两个条件形成了三个分支。多次使用这样的嵌套方式可用  $n$  个条件形成  $n+1$  个分支。

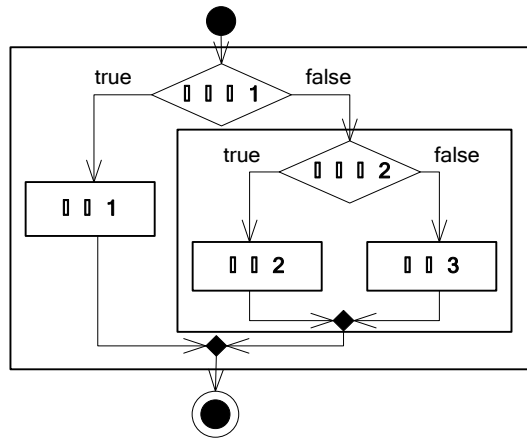


图 4-6 嵌套的 if-else 语句形式

例 4-1 实现一个函数  $y = \begin{cases} 1 & x > 0 \\ 0 & x = 0, \text{ 任意给定自变量 } x \text{ 的值, 求函数 } y \text{ 的值.} \\ -1 & x < 0 \end{cases}$

```
#include <iostream>
using namespace std;
int main(void){
    float x, y;
    cout << "input x=";
    cin >> x;
    if (x > 0)           //条件 1
        y = 1;          //语句 1
    else
        if(x == 0)       //条件 2
            y = 0;       //语句 2
        else
            y = -1;       //语句 3, 此时, x < 0
    cout<<"x="<<x<<" y="<<y<<endl;
    system("pause");
    return 0;
}
```

上面嵌套的 if-else 语句能实现函数  $y$  的功能。下面语句也能实现相同的函数  $y$  功能：

```
y = 1;           //先假设 x > 0 的情形
if (x <= 0)
    if (x == 0)
```

```

        y = 0;                //x == 0 的情形
    else
        y = -1;              //此时, x < 0

```

在书写 if 语句嵌套形式时要特别注意 else 与 if 的配对问题。一定要确保 else 与 if 的对应关系不存在歧义性。对于上面代码,可能将“else”与第一个 if 垂直对齐,但并没有改变程序的功能,如下所示:

```

    if (x <= 0)
        if (x == 0)
            y = 0;            //x == 0 的情形
    else
        y = -1;              //此时, x < 0

```

C++中规定 else 与其前边最近的未配对的 if 相配。因此上面代码仍然完成原先功能。如果真的要使 else 与第一个 if 配对,就要用复合语句花括号{}来改变配对关系,如下所示:

```

    y = 1;                    //
    if (x <= 0){
        if (x == 0)
            y = 0;            //x == 0 的情形
    }else                     //else 与第一个 if 配对
        y = -1;              //此时, x > 0

```

这样仅添加了一对花括号就改变所实现的函数。当  $x > 0$  时,  $y = -1$ ; 当  $x < 0$  时,  $y = 1$ 。

例 4-2 输入一个分数,输出相应的五分制成绩。设 90 分以上为“A”,80~89 分为“B”,70~79 分为“C”,60~69 分为“D”,60 分以下为“E”。

```

#include <iostream>
using namespace std;
int main(void){
    int iScore = 0;
    cout << "Please Enter Score: ";
    cin >> iScore;
    if (iScore >= 90)
        cout << "A";                // >=90
    else if (iScore >= 80)
        cout << "B";                //80-89
    else if (iScore >= 70)
        cout << "C";                //70-79
    else if (iScore >= 60)
        cout << "D";                //60-69
    else
        cout << "E";                //<60
    system("pause");
    return 0;
}

```

上面程序用 4 个条件划分了 5 个分支。注意,在嵌套的 if 语句中,后一个 if 条件判断以前一个 if 条件不满足为前提,因此 if 条件的次序很重要。

### 4.3.2 开关语句

开关 switch 语句也称为多选择语句、或者多分支语句。它可以根据给定的表达式,从多个分支语句序列中选择执行一个分支。该语句的一般格式为:

```

switch(<表达式>){
case <常量表达式 1>: <语句序列 1>
                    [break; ]
case <常量表达式 2>: <语句序列 2>
                    [break; ]
    ...
case <常量表达式 n>: <语句序列 n>
                    [break; ]

[default:          <语句序列 n+1>]

}

```

其中，<表达式>的类型只能是整型，即 char、short 或 int 型。<常量表达式>的类型也只能 char、short 或 int 型，通常是字面值。每个<语句序列>可由一条或多条语句组成，也可以是空。每个 case 常量及其后的语句序列构成一个 case 子句。一条 switch 语句可包含一个以上的 case 子句。多个 case 子句的常量之间不应重复。如果有 break 语句的话，它应该是语句序列中的最后一条语句。其中 break 语句和最后的 default 子句都是任选的。一条完整的 switch 语句涉及到 4 个关键字：switch、case、break、default。

假设一条 switch 语句包含 default 子句，它的执行过程如图 4.7 所示。

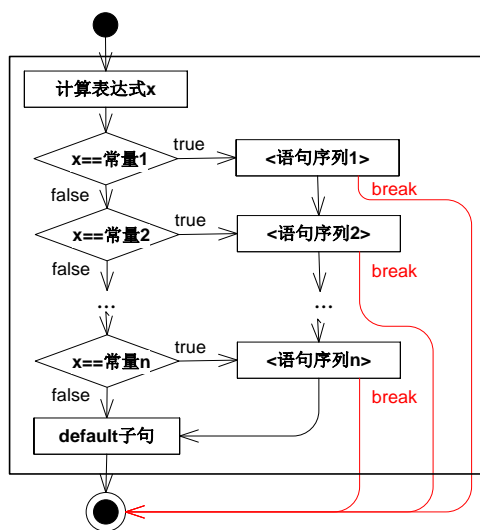


图 4-7 switch 语句的执行过程

先计算表达式的值，设为  $x$ ，再依次与下面 case 常量进行比较。若  $x$  与某个 case 常量相等，则以此为入口，转去执行该 case 子句的语句序列，一直执行下去，直到遇到 break 语句或 switch 语句结束的右花括号为止。如果  $x$  与所有的 case 常量都不相等，而且有 default 子句，就执行 default 子句，然后结束 switch 语句的执行。

应该注意的是，每个 case 常量只是一个入口标号，而不能确定前一个 case 子句的终点。所以如果要使每个 case 子句作为一个条件分支，就应将 break 语句作为每个 case 子句的最后

一条语句，这样每次执行只能执行其中一个分支。这是 `switch` 语句的习惯用法。

`default` 子句可以放在 `switch` 语句中的任何位置，但习惯上作为最后一个子句，此时不需要再将 `break` 作为它的最后一条语句。

例 4-3 输入一个分数，输出相应的五分制成绩。设 90 分以上为“A”，80~89 分为“B”，70~79 分为“C”，60~69 分为“D”，60 分以下为“E”。

```
#include <iostream>
using namespace std;
int main(){
    float score = 0;
    cout << "Please Enter Score: ";
    cin >> score;
    if (score > 100 || score < 0){
        cout<<score<<" is invalid. 0-100 is valid!"<<endl;
        return;
    }
    switch (int(score/10)){
        case 10:
        case 9: cout<<'A';
                break;
        case 8: cout<<'B';
                break;
        case 7: cout<<'C';
                break;
        case 6: cout<<'D';
                break;
        default:cout<<'E';
    }
    system("pause");
    return 0;
}
```

上面程序允许输入浮点数成绩，而且假定合理的分数范围应该在 0 到 100 之间，并用 `if` 语句检查。对表达式 `score/10` 进行强制类型转换为 `int`，这是必要的。第一个 `case` 子句的语句序列为空，这样就与第二个 `case` 子句公用一个入口。当省略 `case` 后面的语句序列时，则可实现多个入口，执行同一个语句序列。

例 4-4 任意给定一个月份数，输出它属于哪个季节(12 月、1 月、2 月是冬季；3 月、4 月、5 月是春季；6 月、7 月、8 月是夏季；9 月、10 月、11 月是秋季)。

```
#include <iostream>
using namespace std;
int main(void){
    int month;
    cout<<"input month=";
    cin>>month;
    switch(month){
        case 12:
        case 1:
        case 2:
            cout<<"winter"<<endl;           //12、1、2 月是冬季
            break;
        case 3:
        case 4:
        case 5:
            cout<<"spring"<<endl;           //3、4、5 月是春季
    }
```

```

        break;
    case 6:
    case 7:
    case 8:
        cout<<"summer"<<endl;           //6、7、8 月是夏季
        break;
    case 9:
    case 10:
    case 11:
        cout<<"autumn"<<endl;           //9、10、11 月是秋季
        break;
    default:
        cout<<"input month error!"<<endl;
    }
    system("pause");
    return 0;
}

```

switch 语句能实现的功能，用嵌套的 if 语句同样能实现。在一些情况下用 switch 语句可能具有较好的可读性。但 switch 语句在使用中有许多限制和注意点。例如，条件表达式不能是浮点数，而且只能判断是否相等来确定入口，break 容易忘记等。

## 4.4 循环语句

编程中经常要反复执行同一语句序列，直至满足某个停止条件为止，这种重复执行过程称为循环。C++提供了 3 种循环语句：while、do-while()和 for 语句。每一种循环都具有三要素：

- 1、初始条件，主要用于设置启动循环的一些变量；
- 2、终止条件，用于设置循环的出口点；
- 3、循环变量的修改，在循环过程中修改循环变量以达到终止条件，避免死循环。

### 4.4.1 while 语句

while 语句的语法格式为：

```

while(<表达式>)
    <循环体语句>

```

其中，<表达式>确定了执行循环的条件，可以是任意合法表达式，但通常是一个关系表达式或逻辑表达式。如果表达式的值为非 0(即逻辑真)，就执行<循环体语句>，然后再计算<表达式>再判断，直到<表达式>的值为 0(即逻辑假)。<循环体语句>是一条语句，也可以是一条复合语句。图 4.8 描述了 while 语句的执行过程。

如果在一开始表达式的值为 0，即为逻辑假，则直接退出循环，而不执行循环体语句。

下面代码计算  $\text{sum}=1+2+3+\dots+100$ 。当然你可以不用循环，只用一个简单公式就能得到

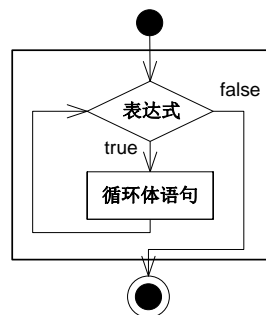


图 4-8 while 语句的执行过程

结果，但我们的目的是掌握循环语句的用法。

```
int sum = 0, i = 1;           //初始条件
while (i <= 100){             //循环条件，当 i=101 时停止循环
    sum += i;
    i++;                       //循环变量 i 修改，以控制循环次数
}
```

上面 **while** 语句中的循环体是一条复合语句，也可简化为一条语句如下：

```
while (i <= 100)
    sum += i++;               //变量 i 后缀自增，以控制循环次数
```

如果在执行循环过程中循环无法终止，就形成死循环或无限循环。通常在循环三要素中若有一个考虑不当，就有可能造成死循环。死循环在语法上是没有错误的。在程序设计时，应该避免产生死循环。例如下面代码：

```
while (x = 3){
    ++x;
    y = x;
}
```

由于表示循环条件的表达式的值永远为 3，即永远为逻辑真，循环无法终止，因此这是一个死循环。

一条 **while** 语句的条件即便恒为真，也不一定是死循环，函数体中可用 **break** 语句来停止循环。

例 4-5 计算  $1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n}$  的和刚好大于等于 3 时的项数  $n$ 。

```
#include <iostream>
using namespace std;
int main(void){
    int n = 1;                //对循环变量 n 初始化 1
    double sum = 0;           //对累加求和变量 sum 初始化 0
    while(sum < 3){           //循环条件为 sum 小于 3
        sum += 1.0/n;         //A 修改循环变量 sum
        n++;                  //修改循环变量 n
    }
    cout<<"n="<<n-1<<"    sum="<<sum<<endl;
    system("pause");
    return 0;
}
```

程序运行后，在屏幕上输出的结果为：

```
n=11 sum=3.01988
```

该程序中比较容易出错的是 A 行，是  $1.0/n$  而不是  $1/n$ 。因为  $1/n$  的值是 0，sum 永远达不到 3，就形成了死循环。

在书写循环语句时，一般将循环体语句缩进一个 **tab** 位置，这样方便程序阅读理解。

在 **while** 表达式中可直接用 **cin>>n** 循环输入整数，如下面代码：

```
int n;
while (cin >> n) {
    cout << n*n << endl;
}
```

要停止循环并非输入 0，而是输入 **Ctrl+Z**。



### 4.4.2 do-while 语句

do-while 语句也是一种循环语句，先执行循环体，再判断条件，语法格式为：

```
do
    <循环体语句>
while(<表达式>);
```

其中，<循环体语句>是一条语句；<表达式>可为任意表达式，但通常是一个关系表达式或逻辑表达式。执行过程如图 4.9 所示。先执行<循环体语句>，然后再计算<表达式>的值，如果表达式的值为真，再次执行<循环体语句>，否则结束循环。

do-while 语句与 while 语句的区别在于，do-while 语句的循环体至少会被执行一次，而 while 语句的循环体有可能一次都不执行。如果一个 while 语句的循环体至少要执行一次的话，就可以用 do-while 来实现。例如下面代码计算  $\text{sum}=1+2+3+\dots+100$ 。

```
int sum = 0, i = 1;           //初始条件
do
    sum += i++;               //循环变量 i 修改，以
    控制循环次数
while (i <= 100);             //循环条件，当 i=101 时停止循环
```

例 4-6 从键盘输入若干字符，直至按下换行键结束，统计输入的字母的个数。

```
#include <iostream>
using namespace std;
int main(void){
    int count=0;
    char ch = 0;
    do{
        ch = cin.get();           //A
        if(ch >= 'A' && ch <= 'Z' || ch >= 'a' && ch <= 'z')
            count++;
    }while(ch != '\n');           //用回车键作为循环终止条件
    cout<<"count="<<count<<endl;
    system("pause");
    return 0;
}
```

A 行中的 `cin.get()` 函数用于从键盘读入任一字符，包括空格、Tab、换行符等，并将读入的字符赋给变量 `ch`。如果该字符是大写或小写字母，则 `count` 加 1，并继续循环，直至 A 行读入的字符为换行符时，结束循环。

例 4-7 输入多个温度值，计算这些温度的平均值。

```
#include <iostream>
using namespace std;
int main(void){
    char ch=0;                //为循环控制变量
    int count=0;              //温度的个数
    double temperature = 0.0, tempsum = 0; //温度总和
    double average = 0.0;
```

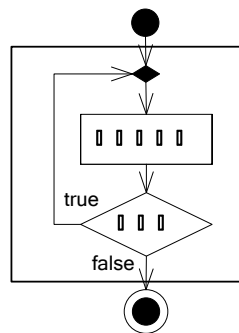


图 4-9 do-while 语句的执行过程

```

do{
    cout<<"输入一个温度: ";
    cin>> temperature;
    tempsum += temperature; //建议多用一个变量, 看起来正规点
    count++;
    cout<<"是否需要输入下一个温度?(y/n) : ";
    cin>>ch;
}while(ch == 'y');
average = tempsum /count;
cout<< "平均温度为:" << average <<endl;
system("pause");
return 0;
}

```

上面两个例子中, 都要求循环体至少要执行一次, 这也是选择 **do-while** 循环语句的原因, 当然用 **while** 语句也同样能实现。

当 **do-while** 语句执行完成, **while<条件>** 不一定为假, 这是因为函数体中可用 **break** 语句来停止循环。

#### 4.4.3 for 语句

**for** 语句的一般语法格式为:

```

for(<表达式 1>; <表达式 2>; <表达式 3>)
    <循环体语句>

```

其中, 表达式 1、表达式 2 和表达式 3 可以是任意表达式。表达式 1 常用于设置循环变量的初值; 表达式 2 设置循环终止的条件; 表达式 3 对循环变量进行修改; 循环体语句是一条语句, 可以是一条复合语句。

图 4.10 表示了 **for** 语句的执行过程如下:

- (1) 计算表达式 1 的值, 对循环变量进行初始化。
- (2) 计算表达式 2 的值, 作为循环条件。
- (3) 若表达式 2 的值为非 0(真), 则执行循环体语句; 否则就结束 **for** 语句的执行。
- (4) 计算表达式 3 的值, 对循环变量进行修改, 以控制循环次数。
- (5) 重复(2)、(3)、(4)步。

下面代码计算  $\text{sum}=1+2+3+\dots+100$ 。

```

for(int sum = 0, i = 1; i <= 100; i++)
    sum += i;

```

上面 **for** 语句中, 表达式 1 是一个逗号表达式, 说明了两个循环变量并初始化。表达式 2 是循环条件。表达式 3 对循环变量加 1 来控制循环次数。循环体语句是一条赋值语句。上面 **for** 语句可改变如下:

```

for(int sum = 0, i = 1; i <= 100; sum += i++);

```

将原先的循环体语句放在表达式 3 中, 而使得循环体改变为一条空语句, 即一个分号。这里我们看到了空语句的一种用法。在这里空语句不可缺少。如果不小心少写一个分号, 下

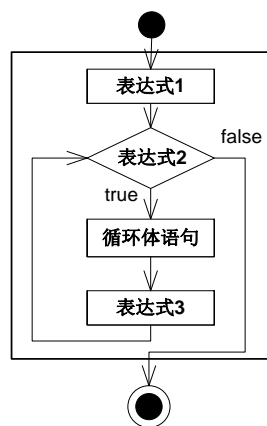


图 4-10 **for** 语句的执行过程

一条语句就作为这个 for 语句的循环体。

在 for 语句中,表达式 1、表达式 2 和表达式 3 都可以省略,但圆括号中的两个分号不能省略。如果省略了表达式 1,那么循环变量初始化将放在 for 语句之前完成;如果省略了表达式 2,意味着循环条件永远为真,那么在循环体中必须有跳转语句(例如 break)来终止循环;如果省略了表达式 3,就要将循环变量修改代码放在循环体内完成。

下列 3 段代码都是计算  $\text{sum}=1+2+3+\dots+100$ 。

```
(1) i = 1; //设置循环变量 i 的初值为 1
    for(; i <= 100; i++) //省略了表达式 1
        sum += i;
(2) for(i = 1; ; i++) //省略了表达式 2
        if(i > 100) //当 n 大于 100 时控制循环终止
            break;
        else
            sum += i;
(3) i = 1; //设置循环变量 i 的初值为 1
    for(; i <= 100;){ //省略了表达式 1 和表达式 3
        sum += i;
        i++; //修改循环变量 i 的值,使其加 1
    }
```

对 for 语句,有以下几点说明。

(1)for 语句的循环体可能一次也不执行,也可能执行多次。这与 while 语句相似。for 语句与 while 语句可以互相转换。

(2)表达式 1 中可说明变量并进行初始化,在 for 语句后面的代码不可再访问这些变量,除非有特殊的编译选项或者 VC6 可用。

(3)还有一种新式的 for 语句,称为基于范围的 for 语句,作用于一个数组或一个 STL 容器(如 vector)中的每个元素的遍历,第 6 章介绍。

#### 4.4.4 循环语句的比较

下面从循环三要素的角度来分析 3 种循环语句的异同。

##### while 循环:

```
<循环变量>=<初值>;
while(<条件>){
    <循环体语句>
    <改变循环变量>
}
```

##### do-while 循环:

```
<循环变量>=<初值>;
do{
    <循环体语句>
    <改变循环变量>
}while(<条件>);
```

##### for 循环:

```
for(<循环变量>=<初值>;
    <条件>; <改变循环变量>){
    <循环体语句>
}
```

(1) while 和 for 语句在每次执行循环体语句之前测试循环条件,然后决定是否执行循环体语句;而 do-while 语句在每次执行循环体语句之后测试循环条件,然后再决定是否再次执行循环体语句。如果一开始循环条件不成立,while 和 for 语句不执行循环体语句;而 do-while 语句要执行一次循环体语句。

(2) for 语句在<表达式 3>中不仅能控制循环,而且还可实现循环体中的操作。

(3) 对于 while 和 do-while 语句,在循环体中应包含循环变量修改来控制循环结束。循环

变量的初始化应在 `while` 和 `do-while` 语句之前完成。而 `for` 语句可以在 <表达式 1> 中实现循环变量的初始化。

(4) `while` 语句能实现的, 用 `for` 语句都能实现。所以 `for` 语句用得多一些, `while` 语句次之, 用得最少的是 `do-while` 语句。

#### 4.4.5 循环的嵌套

如果在循环体内又包含了循环语句, 就称为循环的嵌套。3 种循环语句中循环体语句都可以是任意语句, 当然也可以是循环语句, 而且嵌套的层次没有数量限制。

当一个循环体内嵌套内层循环时, 内层循环往往要使用外层循环的循环变量, 但内层循环不应改变外层循环的循环变量。

例 4-8 按下面格式打印九九表。

```
1*1= 1
1*2= 2 2*2= 4
1*3= 3 2*3= 6 3*3= 9
1*4= 4 2*4= 8 3*4=12 4*4=16
1*5= 5 2*5=10 3*5=15 4*5=20 5*5=25
1*6= 6 2*6=12 3*6=18 4*6=24 5*6=30 6*6=36
1*7= 7 2*7=14 3*7=21 4*7=28 5*7=35 6*7=42 7*7=49
1*8= 8 2*8=16 3*8=24 4*8=32 5*8=40 6*8=48 7*8=56 8*8=64
1*9= 9 2*9=18 3*9=27 4*9=36 5*9=45 6*9=54 7*9=63 8*9=72 9*9=81
```

第 1 行到最后 1 行, 总共有 9 行, 分别标记从 1 到 9。这是外层循环, 用一个变量 `i` 控制。对于每一行, 从 1 到 `i`, 打印 `i` 个乘积。这是内层循环, 用一个变量 `j` 控制。

另一个问题就是控制打印位置, 主要是使表格在垂直方向对齐。有多种办法来达到这个目的。可采用 `iomanip` 头文件中提供的输出控制函数。用 `setw(n)` 函数来控制打印每个整数将占用 `n` 个字符位。编程如下:

```
#include <iostream>
#include <iomanip>
using namespace std;
int main(){
    for (int i = 1; i <= 9; i++){
        for (int j = 1; j <= i; j++){
            cout<<' '<<j<<'* '<<i<<'='<<setw(2)<<j*i;
            cout<<endl;
        }
        system("pause");
        return 0;
    }
}
```

注意 `setw(2)` 函数仅对下面一个数据输出有效, 没有持续性, 因此每次输出数据项时都要重新设置所占位数。有关输出格式控制函数在第 14 章详细介绍。

## 4.5 跳转语句

编程中往往要根据某些条件来改变程序执行的流程。跳转语句就是控制执行流程的语句。当执行到该类语句时，它要改变程序的执行顺序，即不执行下面语句，而跳到另一个语句处接着执行。跳转语句包括 `break`、`continue`、`goto`、`throw` 语句。其中 `throw` 语句将在后面章节介绍。

### 4.5.1 `break` 语句

`break` 语句用于结束 `switch` 或循环语句的执行。其语法格式为：

```
break;
```

当程序执行到该语句时，将终止 `switch` 或循环语句的执行，并将控制转移到该 `switch` 或循环语句之后的第一条语句开始执行。前面介绍过，`switch` 语句中的每个 `case` 分支只起一个入口标号的作用，而不具备终止 `switch` 语句的功能。因此在 `switch` 语句内部，要终止该语句的执行，必须使用 `break` 语句。对于循环语句，在其循环体内，当某一条件满足要终止该循环语句的执行时，也可使用 `break` 语句。图 4.11 表示了 3 种循环语句中的 `break` 和 `continue` 语句的执行流程。

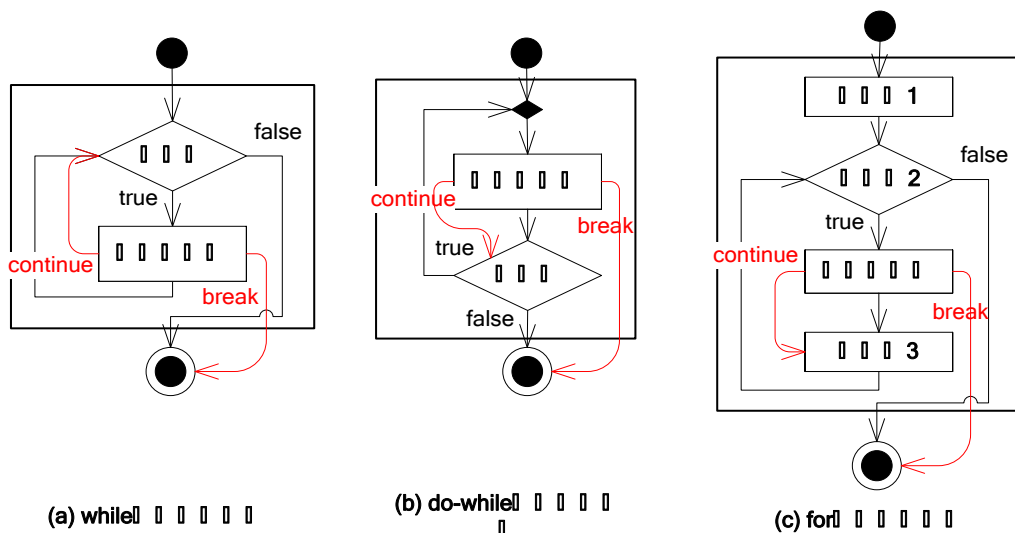


图 4-11 循环语句中的 `break` 和 `continue` 语句的执行流程

例 4-9 输出 2~100 之间的所有素数，每行输出 10 个，最后打印素数的个数。

什么是素数？例如 2、3、5、7、11、13...，只能被 1 和自己整除的自然数就是素数，也称为质数。2 是第一个素数。除 2 之外的偶数都肯定不是素数，所以只要检查从 3 到 100 的每个奇数是否满足整除要求，就能确定是否是素数。一个外层循环从 3 到 100 的每个奇数，

用一个循环变量  $i$  控制。

对于每个奇数  $i$ ，检查是否能被某个值  $j$  整除， $j$  的范围是从 3 到  $i$  的平方根，不需要更大的范围。这是一个内层循环，当发现  $i$  被  $j$  整除，那么  $i$  就不是素数，就应立即停止内层循环，此时就用到 `break` 语句来停止内层循环。当内层循环停止后，再判断是被整除了，还是未被整除而循环到头了。如果是循环到头了， $i$  就是一个素数，计数、打印。否则  $i$  就不是素数。内层循环结束，外层循环到下一个奇数。最后再打印计数值。

编程如下：

```
#include <iostream>
using namespace std;
int main(void){
    int count=1;                //2 是已知的第一个素数
    cout<<2<<" ";              //先输出 2
    for(int i = 3; i <= 100; i += 2){    //外层循环，100 以内奇数
        int j;
        for(j = 3; j * j <= i; j += 2)    //内层循环
            if( i % j == 0)                //如果被整除，i 不是素数
                break;                    //停止内层循环
        if(j * j > i){                    //如果循环到头了，i 就是素数
            count++;
            cout<<i<<" ";
            if(count % 10 == 0)
                cout<<endl;                //控制每行输出 10 个
        }
    }
    cout<<"\ntotal " <<count<<" primes"<<endl;
    system("pause");
    return 0;
}
```

程序运行后，在屏幕上输出的结果为：

```
2 3 5 7 11 13 17 19 23 29
31 37 41 43 47 53 59 61 67 71
73 79 83 89 97
total 25 primes
```

在多重循环中，`break` 语句只终止其所在层次的循环，而不会终止所有循环。

#### 4.5.2 continue 语句

在循环结构中往往需要在满足特定条件时停止本轮循环，跳转到当前循环语句的条件表达式，判断是否进行下一轮执行。这就需要 `continue` 语句。`continue` 语句的语法格式为：

```
continue;
```

`continue` 语句仅能用于循环语句中，其作用是终止本轮循环，将控制流跳转到当前循环的条件测试。对于 `for` 语句，则跳转到<表达式 3>开始执行，如图 4.11 所示。

对于 3 种循环语句，如果将 `continue` 作为循环体中的最后一条语句，将没有任何意义。这是因为循环体执行完最后一条语句，将自动进行下一轮条件测试。所以一般来说，`continue` 语句不会用在循环体的最后一条语句。例如：

```
for(int i = 1; i <= 100; i++){
```

```
    if (i % 3 == 0)
        continue;
    cout<<i<<" ";
}
```

上面 for 语句中 i 从 1 到 100 循环, 但输出的数据中去掉了 3 的倍数。这是因为当 i 为 3 的倍数时, 执行 continue 语句, 结束本轮循环, 调去执行 i++, 开始执行下一轮循环。

注意, break 语句与 continue 语句的区别:

(1) break 语句是终止本层循环, continue 语句是终止本轮循环。

(2) break 语句可用于循环和 switch 语句中, continue 语句只能用在循环语句中。

### 4.5.3 goto 语句与标号语句

在编写汇编语言程序时经常要用到无条件跳转语句, 这是因为机器语言支持无条件跳转语句。在 C/C++ 语言中也提供了无条件跳转语句 goto。其语法格式为:

```
goto <语句标号>;
```

其中, 语句标号是用户命名的标识符, 用于标识某条语句的开始位置。语句标号放在该语句的最前边, 用冒号 “:” 与语句隔开。

当程序执行到 goto 语句时, 将控制无条件转移到所指定的语句标号处, 即从语句标号为新的入口点继续执行下面的语句。带有该标号的语句可以在 goto 语句前面, 也可以在 goto 语句的后面。

goto 语句能从多重循环体中直接跳到循环的最外面, 能省却多个 break 语句。例如:

```
int i, j;
for ( i = 0; i < 10; i++ ){
    cout<<"Outer loop executing. i = "<<i<<endl;
    for ( j = 0; j < 3; j++){
        cout<<"Inner loop executing. j = "<<j<<endl;
        if ( i == 5 )
            goto stop;                //goto 到 stop 语句
    }
}
/* This message does not print: */
cout<<"Loop exited. i = "<<i;        //该语句没有机会执行
stop:                                //标号语句
cout<<"Jumped to stop. i = "<<i<<endl;
```

上面代码中一条 goto 语句从两层循环中直接跳出来, 看起来很灵活, 但也造成一条语句永远没有机会执行。

所有的循环语句能实现的功能, 采用条件语句加 goto 语句都能实现。但由于 goto 语句会降低程序的可读性和可理解性, 应该少用或不用 goto 语句。结构化编程严格禁止使用 goto。实践表明, 不存在哪一种编程功能非要用 goto 不可。如果在程序中出现了 goto, 你总能找到消除的办法。

## 4.6 例子

利用本章学过的语句可设计实现更复杂的编程。我们按照结构化设计的方法来进行一些实际问题的求解。

例 4-10 输入任意 3 个数，按从大到小的顺序输出。

假设输入的是 1、2、3，输出应为 3、2、1。整个过程需要以下 3 步：

- 1、输入 3 个值，分别保存到 3 个变量中，假设分别为 x、y、z。
- 2、按从大到小排列这 3 个值，就是使 x 最大、y 次之、z 最小。
- 3、按 x、y、z 的次序输出。

比较复杂的是第 2 步，需要进行 3 轮的比较和交换：

第 1 轮：处理 x 和 y，经过比较和交换，使 x 大于等于 y。结果为(2,1,3)

第 2 轮：处理 x 和 z，经过比较和交换，使 x 大于等于 z。结果为(3,1,2)

第 3 轮：处理 y 和 z，经过比较和交换，使 y 大于等于 z。结果为(3,2,1)

这样就得到计算结果。编程实现如下：

```
#include <iostream>
using namespace std;
int main(void){
    float x,y,z,temp;           //temp 是用于交换的临时变量
    cout<<"input x,y,z=";      //提示输入 3 个数
    cin>>x>>y>>z;             //输入 3 个数 x,y,z
    if(x < y)
        {temp = x; x = y; y = temp;} //交换 x 与 y 的值，将大数放在 x 中
    if(x < z)
        {temp = x; x = z; z = temp;} //交换 x 与 z 的值，将最大数放在 x 中
    if(y < z)
        {temp = y; y = z; z = temp;} //交换 y 与 z 的值，将最小数放在 z 中
    cout<<"large to small: "<<x<<"\t"<<y<<"\t"<<z<<endl;
    system("pause");
}
```

这个例子是最简单的排序问题。排序的关键是比较和交换。一般来说，处理 2 个数需要 1 次比较和交换，处理 3 个数需要 3 次，处理 4 个数需要 6 次等。一般来说，要交换 2 个变量的值，需要执行 3 步，要利用一个临时变量。在后面章节学过数组之后，就能将 n 个数据存放在一个数组中，然后再用循环来实现排序。

例 4-11 已知  $\frac{\pi}{4} \approx 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$ ，求  $\pi$  的近似值，要求第 n 项小于或等于  $10^{-6}$ 。

计算过程有如下 2 步：

1、求  $\pi/4$  的多项式的值。用一个循环进行逐项累加，需要一个循环变量 i 从 1 开始奇数，作为每一项的分母。循环停止条件是  $1.0/i$  的值小于或等于  $10^{-6}$ 。注意每次循环都要改变符号，使正负交替累加。

2、计算  $\pi$  并输出。

编程实现如下：

```
#include <iostream>
using namespace std;
int main(void){
    int n = 1;                //n=1,3,5,7..
    double t = 1, pi = 0;
    const double eps = 1e-6; //eps 用于设置误差
    while (1.0/n >= eps){
        pi += t/n;
        n += 2;               //下一个奇数
    }
```



```

        t = -t;                      //使各项以正负数交替出现
    }
    system("pause");
    cout<<"pi="<<4*pi<<" "; n="<<n<<endl;
}

```

执行程序,得到 $\pi$ 的近似值为 3.14159,此时  $n=1000001$ ,刚好大于  $10^6$ ,满足精度要求。注意 while 条件是  $1.0/n \geq \text{eps}$ 。如果你不小心书写为  $1/n \geq \text{eps}$ ,将会是什么结果?你是否可以把 while 条件改为  $n < 1000000$ ?是否可改用 for 语句来实现?

例 4-12 Fibonacci 数列的前 6 项为 1、1、2、3、5、8。按此规律输出该数列的前 30 项,每行输出 5 项。

首先要找出各项值的规律。第 1 项和第 2 项为 1,后面每一项都是其前两项之和。例如,  $2=1+1$ 、 $3=2+1$ 、 $5=3+2$ 、 $8=5+3$ 。过程需要以下两步:

- 1、先确定并输出第 1 项(变量 a)和第 2 项(变量 b);
- 2、从第 3 项循环到第 30 项,用一个循环变量 i 控制。先计算第 i 项的值,  $c=a+b$ ,再控制输出 c,然后推移变量  $a=b$ ,  $b=c$ ,为计算下一项做准备。

编程实现如下:

```

#include <iostream>
#include <iomanip>
using namespace std;
int main(void){
    int a=1,b=1,count=2,i,c;
    cout<<setw(10)<<a<<setw(10)<<b; //输出第 1 项和第 2 项
    for(i=3;i<=30;i++){
        c=a+b;                      //当前项为前两项之和
        cout<<setw(10)<<c;           //输出的每个数占 10 个字符
        if(++count % 5 == 0)
            cout<<endl;              //每行输出 5 个数
        a=b; b=c;                    //推移数据项,为求下一项作准备
    }
    system("pause");
    cout<<"\n";
}

```

以上程序用 for 语句实现。循环体中并没有用到循环变量 i,变量 i 仅用于控制循环次数。Fibonacci 序列是一种有用的序列,在后面学到函数递归调用时,还可用递归来实现。

例 4-13 按十进制输入一个正整数,按逆序逐位输出。例如,输入 345,则输出 543。反复输入、计算、输出,直到输入 0 结束。如果输入了负值,提示错误并重新输入。

整个过程是一个循环,对于每一次循环执行以下步骤:

- 1、输入一个正整数,如 345,并保存到一个变量 m 中。
- 2、如果 m 是 0 就停止循环。
- 3、如果 m 是负数,给出提示并重新输入。
- 4、计算每一位并输出。需要一个循环过程,每一轮循环处理一位数。有以下 3 步:
  - 4.1 对于 m,先求其个位数(除以 10 的余数)并输出(如 5)。
  - 4.2 再将 m 除以 10 的商赋给 m; 如 34。

4.3 重复 4.1 和 4.2，直到  $m$  的值为 0。可用一个循环语句实现。编程如下：

```
#include <iostream>
using namespace std;
int main(void){
    int m, n;
    while(1){
        cout<<"input an int(0 to exit):";
        cin>>m;                //输入正整数 m
        if (m == 0)
            break;
        if (m < 0){
            cout<<"input m error"<<endl;
            continue;
        }
        while(m != 0){
            n = m % 10;          //取 m 的最后一位数
            cout << n;           //输出 m 的最后一位数
            m /= 10;             //去掉当前 m 的最后一位数
        }
        cout<<"\n";
    }
    system("pause");
    return 0;
}
```

这个程序使用了嵌套循环，在外层循环中使用了一个恒为真的条件，但用了带条件的 `break` 语句来停止外层循环，所以这不是一个死循环。当输入了负数时，用 `continue` 来执行下一轮执行。当我们要反复输入执行某些程序时，这时一个可供借鉴的例子。内层的 `while` 条件是  $m \neq 0$ ，虽然可以简化为 `while (m)`，但我们建议用条件表达式更清楚。

例 4-14 百钱买百鸡。用 100 元钱要买 100 只鸡。公鸡每只 5 元，母鸡每只 3 元，小鸡每 3 只 1 元，要求每种鸡至少买 1 只，问公鸡、母鸡和小鸡各买多少只？

首先我们判断，解决这个问题不容易找到一个现成的公式来直接计算。

可采用穷举法来解决这一问题。

1、最多能买到多少只公鸡？。最少要用 3 元买 1 只母鸡，最少要用 1 元买 3 支小鸡，那么  $(100-3-1)/5 = 19$ ，至多能买 19 只公鸡，用 `cock` 表示买的公鸡数，从 1 到 19 循环。

2、最多能买到多少母鸡？  $(100-5-1)/3=31$ ，最多 31 只，用 `hen` 表示买的母鸡数，从 1 到 31 循环。

3、用 `chicken` 表示小鸡数，其值只能是  $100-\text{cock}-\text{hen}$ ，而且应该是 3 的倍数。三者所用钱数如果正好是 100 元。可用两层循环实现，编程如下：

```
#include <iostream>
using namespace std;
int main(void){
    int cock, hen, chicken;
    for(cock=1; cock<=15; cock++){          //买的公鸡数为 1~15 只
        for(hen=1; hen<=31; hen++){        //买的母鸡数为 1~31 只
            chicken = 100 - cock - hen;
            if(5*cock + 3*hen + chicken/3 == 100 &&
                chicken % 3 == 0){          //A, 小鸡数为 3 的倍数
                cout<<"cock="<<cock<<" ";
                cout<<"hen="<<hen<<" ";
            }
        }
    }
}
```

```

        cout<<"chicken="<<chicken<<endl;
    }
}
system("pause");
return 0;
}

```

执行程序，输出结果如下：

```

cock=4  hen=18  chicken=78
cock=8  hen=11  chicken=81
cock=12 hen=4   chicken=84

```

注意，因1元钱买3只小鸡，所以小鸡数肯定是3的倍数。如果A行程序改写为：

```
if(5*cock+3*hen+chicken/3==100)
```

想一想能否得到一样的结果？

**例4-15 猴子吃桃问题。**猴子第一天摘下若干桃子，当即吃了一半，又多吃了一个。第二天又将剩下的桃子吃掉一半，又多吃了一个。以后每天都吃了前一天剩下的一半再多一个。到第10天想吃的时候就剩一个桃子了。求第一天共摘下多少个桃子。

首先我们判断，不太容易找到一个现成的公式来直接计算。我们尝试从后往前推断。第10天开始时的桃子为1，那么加1后的2倍就是第9天开始时的桃子数。第9天桃子数加1的2倍就是第8天的桃子数，以此推算共9次，就可以计算出第1天桃子的总数。

```

#include <iostream>
using namespace std;
int main(void){
    int a = 1;
    for(int day = 10; day > 1; day--)        //循环9次
        a = (a + 1) * 2;
    cout << "第1天桃子的总数为: " << a << endl;
    system("pause");
    return 0;
}

```

执行程序，输出结果如下：

```
第1天桃子的总数为: 1534
```

程序代码非常简单，关键是从后往前推的思路。

## 4.7 小 结

- C++语句可分为9类：说明语句、表达式语句、选择语句、循环语句、跳转语句、空语句、复合语句、异常处理语句(后面介绍)、标号语句。
- 说明语句用来引入新的名称，可能是说明新的变量，也可能是新的类型。
- 一条表达式语句就是一个表达式加上一个分号所形成一条语句。赋值语句、自增自减语句都是常见的表达式语句。
- 空语句就是一个分号，主要用于特定的语法需要一条语句，但又不想做任何事。
- 一条复合语句包含一组语句，但逻辑上看作是一条语句。在结构化编程中具有重要

作用。

- 结构化编程的基本结构只有 3 种结构：顺序结构、选择结构和循环结构。任意复杂的程序都是由这三种基本结构嵌套形成。结构化编程拒绝 goto 语句。
- 流程控制语句是一大类语句的统称，包括选择语句、循环语句和跳转语句。
- 选择语句用来根据特定条件来选择执行特定语句，包括 if-else 语句和 switch 语句。
- 循环语句用来根据特定条件来重复执行特定语句，包括 while 语句、do-while 语句和 for 语句。
- 跳转语句包括 break、continue、goto、throw 语句。其中 throw 语句将在后面介绍。

## 4.8 练 习 题

1. C++的语句分类中不包含下面哪一类语句？  
A 复合语句                  B 说明语句                  C 赋值语句                  D 循环语句
2. 结构化编程的三种基本结构是\_\_\_\_\_。  
A 顺序结构、选择结构、循环结构                  B 循环结构、转移结构、顺序结构  
C 递归结构、循环结构、转移结构                  D 嵌套结构、递归结构、顺序结构
3. 为了避免嵌套的 if-else 语句的二义性，C++语言规定 else 与\_\_\_\_\_配对。  
A 垂直对齐的 if                  B 在其之前未配对的最近的 if  
C 在其之后最近的 if                  D 同一行上的 if
4. 关于 if 语句，下面哪一种说法是错误的？  
A 一个 if 只能有一个 else 与之配对                  B if 语句中可以包含循环语句  
C if 语句中可以有多于一个 else if 子句                  D if 语句中不能包含 switch 语句
5. 关于 switch 语句，下面哪一种说法是错误的？  
A default 子句可以放在任何地方，而不一定放在最后。  
B case <表达式>的值必须是一个整型值。  
C 每个子句序列中必须要有 break 语句。  
D 可以有 default 子句，也可以没有。
6. 关于 while 语句，下面哪一种说法是错误的？  
A 在执行 while 语句时，其循环体语句可能一次都不执行。  
B 循环体可以是一条语句，也可以是一条复合语句。  
C 循环体中可以改变控制变量，使 while<条件>不满足而停止循环。  
D 当 while 语句执行完成时，while<条件>一定不为真。
7. 关于 do-while 语句，下面哪一种说法是错误的？  
A 在执行 do-while 语句时，其循环体语句可能一次都不执行。  
B 循环体可以是一条语句，也可以是一条复合语句。  
C 循环体中可以改变控制变量，使 while<条件>不满足而停止循环。  
D 当 do-while 语句执行完成时，while<条件>一定不为假。
8. 关于 for 语句，下面哪一种说法是错误的？

- A 3个表达式都可以为空。  
B 如果第2个表达式为空,表示循环条件恒为真。  
C while语句能实现的循环都能用for语句实现。  
D 第1个表达式在每次循环中都要执行一次。
9. 关于break语句,下面哪一种说法是错误的?  
A break语句可用于循环体内,终止循环语句的执行  
B break语句可用于if体内,终止if语句的执行  
C break语句可用于switch体内,终止switch语句的执行  
D 在一个循环体内,break语句可以出现多次。
10. 下列for语句的循环次数是\_\_\_\_\_次。  
for (int i = 1; i <= 5; sum++) sum += i;  
A 5                      B 4                      C 0                      D 无限
11. 下列for语句的循环次数是\_\_\_\_\_次。  
for(int k=0; ;k++)  
A 无限                      B 0                      C 有语法错,不能执行                      D 1
12. 以下代码段的输出结果是\_\_\_\_\_。  
int x = 3;  
do{  
    cout<<(x-=2)<<" ";  
}while (--x);  
A 1                      B 3 0                      C 1 -2                      D 死循环
13. 下列程序的输出结果是\_\_\_\_\_。  
int main(void){  
    int x=1,i=1;  
    for (; x < 50; i++){  
        if(x >= 10) break;  
        if(x % 2 != 0){  
            x += 3; continue;  
        }  
        x--=-1;  
    }  
    cout<<x<<' '<<i<<endl;  
}
- A 12 7                      B 11 6                      C 12 6                      D 11 7
14. 下列程序的输出结果是\_\_\_\_\_。  
int main(void){  
    int n = 'm';  
    switch(n++){  
        default: cout<<"error";break;  
    }

```

        case 'k':case 'K':case 'l':case 'L':cout<<"good"<<endl;break;
        case 'm':case 'M':cout<<"pass"<<"\t";
        case 'n':case 'N':cout<<"warn"<<endl;
    }
}

```

A pass

B warn

C pass warn

D error

15. 按要求完成编程。

(1)任意输入一个 int 整数，计算其中有多少位是 1。

(2)任意输入一个 int 整数，显示为十六进制形式，不能用 hex 控制符，而且要求输出 8 个字符。

(3)任意输入一个浮点数，小数点后最多有两位，作为一笔金额，显示为中文金额格式，银行里经常要用这种格式。例如，输入 1234.56，输出“壹仟贰佰叁拾肆元伍角陆分”。0 为零，7 为柒，8 为捌，9 为玖，10 为拾，还有万、亿。注意，2004 应输出“贰仟零肆元整”，而不是“贰仟零佰零拾肆元整”。尝试你能准确处理的金额是多少。

(4)输入若干个字符，统计输入的数字字符的个数。

(5)从键盘上输入一个正整数 n，按下式求出 y 的值：

$$y = 1! + 2! + 3! + \dots + n!$$

再编程分析，结果 y 应该是一个正整数，在整型范围之内，可正确计算的最大的 n 和 y 分别是多少。

(6)求一元二次方程  $ax^2+bx+c$  的根。任意输入系数 a、b、c 的值，显示方程的根。注意什么条件下无根，什么条件下有一个根、两个根。

(7)某个数列的前 5 项为： $\frac{1}{2}, \frac{3}{2}, \frac{5}{3}, \frac{8}{5}, \frac{13}{8}$ ，按此规律求出该数列的前 20 项，显示每一项的分子和分母。

(8)输入一个正整数 x，显示其所有的因子。在此基础上，求出 1~100 之间的完全数。所谓完全数是指该数刚好等于它的因子之和（自己本身除外）。例如，6 的因子为 1，2，3，且  $6=1+2+3$ ，因此 6 是一个完全数。

(9)编程验证一个数学定理，任意一个正偶数都能分解为两个素数之和，例如， $6=3+3$ 、 $8=3+5$ 。

## 第5章 函数和编译预处理

程序设计中常常要重复使用一部分相同功能，就需要定义函数、调用函数。函数(function)是结构化编程的基本模块，也是C程序的主要构造单位。本章将介绍函数如何定义、如何调用、函数的原型、递归调用、函数的重载、以及作用域和存储类等。

编译预处理是C/C++语句的特色之一。包含#include 就是最常用的编译预处理指令。本章将介绍其它指令。

### 5.1 函数的基本概念

前面程序中都只有一个主函数 main()，但实际上一个程序往往由多个函数组成。一个函数具有确定的名称、形式参量和返回值，完成一种特定的功能。函数的好处是对于调用方而言的。调用方无需知道被调用函数内部如何实现的诸多细节，从而简化了调用方的编程。

所有的函数定义都是平行的，包括主函数 main。也就是说，在一个函数的函数体内，不能再定义另一个函数，即函数不能嵌套定义。但函数体内可以调用其它函数，这样形成嵌套调用的结构。习惯上把调用方称为主调函数。一个函数还可以调用自己，称为递归调用。

当启动一个C++程序时，是从调用其中的 main 函数开始执行的。main 函数可以调用其它函数，被调用函数执行完成之后再返回 main 函数，最后由 main 函数返回以结束程序。一般情况下 main 函数不允许被其它函数调用。一个C++程序必须有且仅有一个主函数 main。一个C++程序可能由多个源文件组成，但只能有一个 main 函数。

在C++语言中，从不同的角度对函数分类如下。

#### 5.1.1 库函数和用户定义函数

从函数定义的角度看，函数可分为库函数和用户自定义函数两种。

(1) 库函数：C++编译系统、操作系统或其它系统为方便用户编程而预定义的函数。这些函数都有原型说明在特定的头文件中。例如<iostream>文件包含了一组处理输入输出的对象和函数。<math.h>包含了一组数学计算的函数，如平方根 sqrt。程序员只需在程序前包含这些头文件，就可以在下面代码中直接调用这些函数。程序员可参考 MSDN 或相关文献来获知有哪些头文件，以及这些头文件中所包含的函数说明及用法。

(2) 用户自定义函数：程序员根据自己的需要而定义的函数。这类函数往往功能独特，使用范围比较有限。自定义函数是程序设计最常见的现象。算法实现主要就是函数设计。

#### 5.1.2 无参函数和有参函数

从函数调用时数据传送的角度来看，函数可分为无参函数和有参函数两种。

(1) 无参函数：函数定义时没有定义形式参量(formal parameter 简称**形参 parameter**)，那么函数调用也无需提供实际参量(actual parameter 简称**实参 argument**)。

(2) 有参函数：也称为带参函数。在函数定义时说明有一定数量的形参，并按次序排列。每个形参都有确定的类型。这就要求该函数调用时要提供相应数量和类型的实参，以启动函数执行。有的形参只作为输入数据，而有的形参既能作为输入数据也能输出结果(例如数组、指针或引用类型)。

### 5.1.3 有返回函数和无返回函数

函数返回时是否有返回值，从这个角度可把函数分为有返回值函数和无返回值函数两种。

(1) 有返回值函数：如果函数定义时确定了一个返回值的类型，而不是 `void`，那么函数调用执行完后要向调用方返回一个结果，就是函数返回值。返回值的类型就是该函数的类型。例如 `sqrt(2)` 是一个函数调用，它将返回一个 `double` 值作为结果(2 的平方根)，那么 `sqrt(2)` 的类型就是 `double`，可直接参与表达式计算。如果函数体中有多条路径返回，那么所有的路径都要返回相同类型的值。

(2) 无返回值函数：如果函数定义时确定返回 `void`，那么该函数的调用执行完成后将不会返回任何值。例如 `setw(5)` 是一个函数调用，它的执行不会返回值，所以只能进行单独调用，而不能直接参与表达式计算。注意，无返回值并不意味着函数执行没有结果。一个函数的计算结果可能作用在函数之外的数据上，而不一定要返回。

一个函数可看作一个封装的模块。如图 5.1 所示。它有一个名字，说明它的功能和用途。还可能有一组有序的形参，说明调用时应输入哪些类型的实参及多少个实参。一个函数还可能有一个返回值，来说明调用时能返回什么结果。

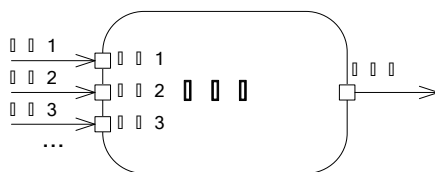


图 5-1 一个函数作为一个模块

一个函数的名称、形参和返回值是该函数对内、对外联系的接口。对于函数的设计者来说，要关注该函数内部如何实现，按照输入的形参如何得到计算结果，以及如何返回结果。对于函数的调用方来说，可将一个函数看作是一个“黑盒”，除了接口外，不必关心内部如何实现。例如我们使用 `cin` 和 `cout`，而无需关心“黑盒”内部如何实现。

## 5.2 函数的定义

一个函数定义由两个部分组成：函数头和函数体。函数头包括返回值类型、函数名以及形参表。函数头也称为函数原型(prototype)，确定了一个函数的调用形式，也确定了一个函数的语义功能。函数体为函数提供一种实现方式，用一对花括号括起来，由一组语句组成，确定了该函数执行时的具体操作。定义函数在先，调用函数在后。下面分别说明定义有参函数和无参函数的定义格式。



### 5.2.1 无参函数的定义

定义无参函数的一般格式为：

```
<type> <函数名>(void)           //函数头
{ <函数体> }
```

其中，<type>为函数返回值的类型，它可以任一类型，可以是基本数据类型，也可以是自定义类型，也可以是这些类型的指针或引用类型。返回值的类型就是函数的类型。<函数名>是一个标识符，应说明函数的功能和用途，通常是一个动词短语，首字母小写。圆括号中的void可以省略。<函数体>用一对花括号封装，一组语句实现函数的功能。

如果一个函数不需要输入任何数据就能完成特定功能，就可定义为无参函数。在很多情况下，无参函数也没有返回值，此时函数返回值的类型为void。例如：

```
void print_f(){
    cout << "ok\n";
}
```

无参函数的调用比较简单，只要确定函数名就能调用。例如：

```
void main(){
    print_f();
}
```

### 5.2.2 有参函数的定义

定义有参函数的一般格式为：

```
<type> <函数名>(<形参表>)
{ <函数体> }
```

有参函数比无参函数多了一个<形参表>。<形参表>一个或多个形参说明构成，每个形参说明的一般格式为：

```
<类型> <形参名>
```

其中，<类型>是形参的类型。<形参名>是形参的名字，用一个标识符表示，一般首字母小写。这个格式与变量定义相似，如：int x。多个形参之间用逗号分隔，而且不能重名，如：int x, float f。多个相同类型的形参必须单独指定类型，而不能书写为int x, y。

一个函数的名字与其形参表作为一个整体，被称为该函数的基调 signature。

如果一个函数的返回类型不是void，就有一个确定的返回类型。有返回值的函数的函数体中就一定有return语句来返回某个结果。return语句是一种跳转语句，当执行到return语句时，立即返回到调用方。return语句的格式为：

```
return <表达式>;
```

或

```
return;
```

第一种格式用于有返回值的函数。其中<表达式>的值将作为返回值。如果return语句中<表达式>类型与函数定义的类型不同，系统就试图进行赋值类型转换(参见3.4.2节)。对于有

返回值的函数，函数体中的每一条路径执行结束都要以 `return` 结尾，而且返回的表达式类型一定要与函数定义的返回值类型兼容。

第二种格式用于无返回值的函数。函数体中可以没有 `return` 语句，语句执行完后，自动返回到调用者。

例如，设计一个函数求二个整数中的大数，将两个输入的整数作为形参，而将结果作为返回值。函数定义如下：

```
int max(int x, int y)
{ return ( x > y ? x : y); }
```

第一行 `int max(int x, int y)` 是 `max` 函数的函数头，`max` 是函数名，返回值是一个整数。形参表中有两个整数：`x` 和 `y`，用逗号分隔。在函数体内，`return` 语句把 `x`(或 `y`) 的值返回给主调函数。

### 5.2.3 函数定义的要点

在定义一个函数时要注意以下方面：

(1) 先确定该函数所要实现的功能，可用自然语言或数学方式来描述实现该功能的算法和步骤。注意一个函数的功能应单一化简单化，而不宜多功能化复杂化，否则就不方便调用。

(2) 为函数确定一个名字，让程序员看到函数名就能明白函数的功能。

(3) 确定函数在执行过程中是否需要调用方提供某些数据。如果需要，就要确定函数的形参，注意形参的名字应该能体现它的用途。如果不需要形参，形参表可为空或用 `void` 说明。即使参数表为空，一对圆括号 “`()`” 也不可省。

(4) 确定函数在执行完成后是否有结果要返回给调用方。若有，要确定返回值的类型。若没有，则函数的返回值类型为 `void`。

(5) 在函数体中允许出现多个 `return` 语句，但每次调用只能有一个 `return` 语句被执行，因此只能返回一个函数值。如果函数体中有多个分支，应保证每个分支均有确定类型的返回值，否则可能出现逻辑错误。例如，把大写字母转换为小写字母的函数 `toLower` 如下：

```
char toLower(char c){
    if(c >= 'A' && c <= 'Z')
        return c + 'a' - 'A';
}
```

编译该函数时将出现一个警告，指出未使每个执行分支均有要求的返回值。如果形参 `c` 是小写字母，该函数就没有确定返回值。应修改如下：

```
char toLower(char c){
    if(c >= 'A' && c <= 'Z')
        return c + 'a' - 'A';
    return c;
}
```

(6) VC6 中如果函数返回类型为 `int`，可以省略不写。如上面 `int max(int x, int y)` 可以简写为 `max(int x, int y)`。但 C99 之后不允许这种“潜规则”，要明确说明返回类型。

(7) 有时函数体中需要知道自己函数的名称，C99 提供了一个内置局部变量，可获取当

前函数的名称：\_\_func\_\_。

(8) 无返回值的函数体可以是空的，但花括号{}不可省。空函数没有什么意义，只在特定场合使用，或者临时使用。

(9) 如果 main 函数返回 int，且函数体中返回 0，VS2015 中 return 0; 语句可省。但 DevC++ 不可省。本文建议不要省略。

(10) C14 标准允许用 auto 自动推导函数返回类型，从 return 表达式类型推出返回类型，要求函数内各个 return 表达式应具有相同类型。例如：

```
auto f() {  
    int i = 3;  
    return i + 33;  
}
```

## 5.3 函数的调用

对于一个函数，只有被调用时，其函数体才能执行。一个函数调用(function call)就是确定一个函数名并提供相应的实参，然后开始执行该函数体中的语句。当函数体执行结束时，函数调用就得到了一个值，返回给调用方。在 C++ 程序中，除 main 函数外，任一函数均不能单独构成一个完整的程序。main 函数直接或间接地调用其它函数来实现程序的功能。

### 5.3.1 函数调用的形式

调用函数的形式为：

<函数名> (<实参表>)

其中，<函数名>是已定义的一个函数的名字；<实参表>由零个、一个或多个实参(用逗号分隔)构成。如果是调用无参函数，<实参表>为空，但括号不能省。

<实参表>中，每个实参都是一个表达式。<实参表>中的实参与被调用函数的<形参表>中的形参，个数及对应位置的类型应该相一致。首先，类型应相同，至少要能兼容。如果类型不同，将采用赋值类型转换，将实参的值转换成形参类型，然后再赋给形参。其次，实参与形参的个数应该相同，但也有例外。如果形参说明带缺省值，实参个数可能少于形参个数。

在函数调用过程中，先计算各实参的值，再赋给对应的形参，然后再启动执行函数体。

一个函数调用是一个表达式，其类型就是该函数的返回值的类型。无返回值的函数调用的类型就是 void。

### 5.3.2 函数调用的方式

按函数调用在语句中的作用来划分，有 3 种函数调用方式。

#### 1. 函数调用语句

函数调用语句就是一个函数调用加上一个分号构成的一条语句。

例如：

```
printf();  
max(a, 3);
```

都是函数调用语句。这种调用方式忽略了函数的返回值。

对于无返回值的函数调用，就只能用这种调用方式，例如上面 `printf()` 调用。

对于有返回值的函数调用，也能如此调用，只是忽略了返回值。但如果返回值比较重要的话，如此调用就失去意义了。例如上面 `max(a, 3)` 调用语句将不能得到 `a` 和 `3` 的较大值。

## 2. 函数调用作为表达式

既然一个函数调用本身就是一个表达式，也具有特定返回类型，因此函数调用可出现在表达式中，以函数返回值参与表达式的计算。这种方式要求函数有返回值，而不是 `void`。

例如：

```
z = max(a, b);
```

是一个赋值语句，把 `max` 的返回值赋予变量 `z`。

## 3. 函数调用作为实参

一个函数调用也能作为另一个函数调用的实参。这种情况是把该函数的返回值作为实参，因此要求该函数必须是有返回值的。

例如：

```
m = max(max(a, b), c);
```

就是把 `max` 调用的返回值又作为 `max` 函数的实参，求出 `a`、`b`、`c` 的最大值。此时作为实参的函数调用要先执行，返回的结果作为外层 `max` 函数调用的实参值，再次调用执行。

例 5-1 用实例验证一个数学定理：任何一个大于 2 的偶数都能分解为两个素数之和。例如：`4=2+2`，`6=3+3`，`8=3+5`，等等。

一种验证途径是，输入任意一个大于 2 的偶数，都能给出所有的分解。程序设计应该能反复输入，而且能检查输入值的合法性。编程如下：

```
#include <iostream>  
using namespace std;  
bool isPrime(unsigned n){ //函数定义，如果 n 是素数，返回 true，否则 false  
    if (n < 2)  
        return false;  
    if (n == 2 || n == 3 || n == 5 || n == 7) //10 以内的素数  
        return true;  
    for (unsigned i = 2; i * i <= n; i++)  
        if (n % i == 0)  
            return false;  
    return true;  
}  
int main(){  
    int n;
```

```

for(;;){
    cout<<"Input an even integer (-1 to break):";
    cin >> n;
    if (n == -1)
        break;
    if (!(n > 2 && n % 2 == 0)){
        cout<<"Input error!"<<endl;
        continue;
    }
    for (int i = 2; i <= n / 2; i++)
        if (isPrime(i) && isPrime(n - i)) //函数调用
            cout<<i<<"+"<<n-i<<"="<<n<<endl;
}
system("pause");
return 0;
}

```

首先定义并实现了一个函数 `bool isPrime(unsigned n)`。这个函数的功能是检查形参 `n` 是否为素数。如果是，返回 `true`，否则就返回 `false`。因此该函数返回一个 `bool` 类型。形参的类型为 `unsigned` 无符号整数，原因是它能持有最大的正整数，就能最大程度地满足素数计算要求。函数体的实现比较简单易懂。这个函数具有通用性，例如你用一条 `for` 语句就能列出 100 以内的所有素数。

下面是主函数设计。用一个 `for(;;)` 设计了一个“死循环”。目的是能反复输入验证。但如果输入 -1，就用 `break` 语句退出循环。如果输入的 `n` 不是大于 2 的偶数，将提示错误并重新输入，使用了 `continue` 语句。真正的验证是用一个 `for` 语句从 2 循环到输入值的一半，用变量 `i` 控制。if 语句的条件中调用了 `isPrime` 函数两次，计算 `isPrime(i) && isPrime(n - i)`，也就是 `i` 和 `n-i` 都是素数时，就找到了一种分解，打印结果，再循环尝试下一种分解。所以程序对于输入的 `n` 能找到所有的分解。例如：当输入 14 时，输出为

```

3+11=14
7+7=14

```

### 5.3.3 函数调用的要点

- 函数说明在前，该函数调用在后。
- 函数调用执行时，每个形参都分配了独立的存储单元，用于接收实参传递来的数据。函数调用执行期间，形参和实参各自拥有独立的存储单元。函数调用执行结束时，系统将回收分配给形参的存储单元。
- 如果函数的形参为普通变量(不是数组、指针或引用)，那么在调用该函数时，实参将给形参赋值，这种参数传递方式称为以值传递(`call by value`)，简称值传递或传值。值传递是单向输入的，即只由实参传给形参，而不能由形参传回来。也就是说，实参值赋给形参后，形参值在函数体中的变化对实参值无任何影响。分析下面例子。

```

#include <iostream>
using namespace std;
int (float x, float y){
    float t;
    t=x; x=y; y=t;
    cout << "函数 swap: x=" << x << "  y=" << y << '\t';
}
int main(void){

```

```
float a=4,b=7;
cout << "主函数: a=" << a <<" b=" << b << '\t';
swap(a, b);
cout << "主函数: a=" << a <<" b=" << b << '\n';
system("pause");
return 0;
}
```

执行上面程序，输出结果为：

```
主函数: a=4 b=7      函数 swap: x=7 y=4      主函数: a=4 b=7
```

可以看到，调用 `swap` 函数后，`main` 函数中的变量 `a` 和 `b` 的值不变。尽管函数 `swap` 中交换了两个形参的值，但并不能改变实参的值。

在第 8 章将介绍指针和引用，它们可作为函数形参，函数中就能改变实参。

- 函数调用时，多个实参的求值顺序因编译系统而异，可能从右向左，也可能从左向右。例如下列代码，在不同的编译系统中，其运行结果可能有差异。

```
int x = 5;
swap(x, x++);
```

## 5.4 函数的重载

函数的重载(`overload`)就是多个函数具有相同的名称，但有不同的形参(个数不同或者类型不同，以及是否有 `const` 修饰)。函数的名称及形参作为一个整体，被称为函数的基调(`signature`)。重载的多个函数虽然名称相同，但基调仍不同，函数调用就是根据基调来区分的。函数重载是 C++ 提供的一种新特征。

为什么需要函数重载？经常要设计这样一些函数，它们的功能语义相同或相似，但处理的形参的类型或个数不同。例如，下面几个输出函数分别输出 `int` 型、`double` 型、`char` 型：

```
void print_int(int i){ .....}
void print_double(double d){ .....}
void print_char(char c){ .....}
```

在调用上述函数时，不仅要记住这些函数的不同名称，而且还要根据所处理数据类型来选择相应的形参，这将给函数调用带来麻烦。利用函数重载就能给这些函数取相同的名称，而用不同的形参来进行区分。例如，我们可以把上述函数定义为：

```
void print(int i){ .....}
void print(double d){ .....}
void print(char c){ .....}
```

在调用这些函数时，根据函数调用所提供的实参的类型来决定实际执行的是哪个函数。例如，`print(1.0)` 将调用上面第二个函数 `void print(double d)`，这是因为 `1.0` 的类型为 `double`。

### 5.4.1 重载函数的定义

先分析下面例子。

例 5-2 分别用函数实现求两个 `int` 数、`float` 和 `double` 数中的大数。

```

#include <iostream>
using namespace std;
int max(int x, int y)                                //A 求两整数中的大数
{ return (x > y ? x : y); }
float max( float a, float b )                       //B 求两单精度实数中的大数
{ return (a > b ? a : b); }
double max(double m, double n)                     //C 求两双精度实数中的大数
{ return (m > n ? m : n); }
int main(void){
    int a1,a2;
    float b1,b2;
    double c1,c2;
    cout << "输入两个整数:\n";
    cin >> a1 >> a2;
    cout<< "输入两个单精度数:\n";
    cin>>b1>>b2;
    cout<< "输入两个双精度数:\n";
    cin>>c1>>c2;
    cout<<"max("<<a1<<','<<a2<<")="<<max(a1,a2)<< "\n";    //D
    cout<<"max("<<b1<<','<<b2<<")="<<max(b1,b2)<< "\n";    //E
    cout<<"max("<<c1<<','<<c2<<")="<<max(c1,c2)<< "\n";    //F
    system("pause");
    return 0;
}

```

上例中定义了三个 `max` 函数，分别求出两个 `int`、两个 `float` 和两个 `double` 的大数。这三个函数的函数名相同，形参的个数也相同，只是形参的类型不同，所以它们的基调不同。

函数调用时，编译器根据实参的个数和类型来确定应该调用哪一个函数。如上例中，当实参为 `int` 型时，D 行调用由 A 行定义的函数 `max`；当实参为 `double` 型时，F 行调用由 C 行定义的函数 `max`。

定义重载函数时应注意以下几点。

(1) 一般来说，一组重载函数具有相同的功能语义，函数的名称应该能反映函数的功能语义。所以一组重载函数之间应该具有相同的功能，只是以不同的方式来处理不同的数据。

(2) 两个函数的基调相同，仅返回值类型不同，不能定义为重载函数。例如：

```

float fun(float x){ ... }
void fun(float x){ ... }

```

编译上面定义的两个函数将报错。

#### 5.4.2 重载函数的调用

每个合法的函数调用将唯一关联一个被调用的函数。编译器将完成这个过程，这个过程被称为绑定(binding，又称为联编)。

重载函数的调用绑定是在编译时刻由编译器根据实参与形参的匹配来决定。过程如下：先构造一个函数候选集，包含了一组函数，这些函数与被调用函数同名，而且形参与调用实参的个数相同，或者个数不同，但后面的形参均带缺省值(后一种情况比较复杂，在后面详细分析)。然后尝试按照下面过程从候选集中选择一个最佳匹配：

(1) 先尝试寻找唯一严格匹配(即每个实参类型都与形参类型完全相同)。如果找到一个函

数，则绑定该函数。例如：

```
print(1);          //绑定到 void print(int)
print(2.3);        //绑定到 void print(double);
print('c');        //绑定到 void print(char);
```

(2) 再通过赋值类型转换寻求一个匹配。具体来说，如果严格匹配不成功，就尝试对实参进行赋值类型转换(参见 3.4.2 节)，然后用转换后的实参类型与候选集中的函数形参进行严格匹配。：如：

```
print(1.0f); // 绑定 void print(double);
```

因为没有形参为 float 类型的 print 函数，此时严格匹配失败，1.0f 将转换成 double 类型，然后再试图进行匹配，于是匹配到 void print(double)函数。

如果有多个形参，情况就比较复杂了。例如对于前面 3 个 max 重载函数，有下面函数调用：

```
cout<<max('a', 'b')<<endl;    //正确，绑定到 max(int, int)
cout<<max('a', 99)<<endl;      //正确，绑定到 max(int, int)
cout<<max(100, 'a')<<endl;     //正确，绑定到 max(int, int)
cout<<max('a', 12.3)<<endl;    //错误
cout<<max(12, 12.3)<<endl;     //错误
cout<<max(3.4f, 12.3)<<endl;   //错误
```

前 3 个调用是合法的，max(char, char)调用、max(char, int)调用和 max(int, char)调用都能唯一转换匹配到 max(int, int)函数。而后 3 个调用却是非法的。max(char, double)调用、max(int, double)调用和 max(float, double)调用都不能唯一绑定到某一个函数。你可能认为应能绑定到 max(double, double)函数，而实际上却有二义性。

(3) 再通过用户自定义转换寻求一个匹配。具体来说，如果(1)和(2)都失败，则对实参进行自定义转换(有一种运算符重载函数称为类型转换函数，定义在类中，第 12 章介绍)。然后用转换后的实参类型与重载函数的形参进行严格匹配。

(4) 如果上述(1)、(2)、(3)匹配都失败，则该重载函数的调用失败。

所谓多态性就是一个名称具有多种具体形态。函数重载就是 C++提供的一种多态性。即同一个函数名称，但具有多种具体的基调。函数重载是静态的、在编译时刻处理的。在后面章节将介绍动态的、在运行时刻处理的多态性。

## 5.5 嵌套调用和递归调用

函数调用允许嵌套。一个函数直接或间接地调用自己就是递归调用。

### 5.5.1 函数的嵌套调用

在 C++中，任意一个函数的定义都是独立的，不允许在一个函数的定义中再定义另一个函数。函数之间都是平等的、平行的，即不允许函数的嵌套定义。但允许在一个函数的定义中调用另一个函数，即允许函数的嵌套调用。嵌套的函数调用返回时将根据嵌套层次逐层返回。如图 5.2 所示，函数 main()中调用函数 g()，函数 g()中又调用 h()，然后逐层返回：

```
int main(){
```



```

.....
f();
.....
}
void f(){
.....
g();
.....
}
void g(){
.....
}

```

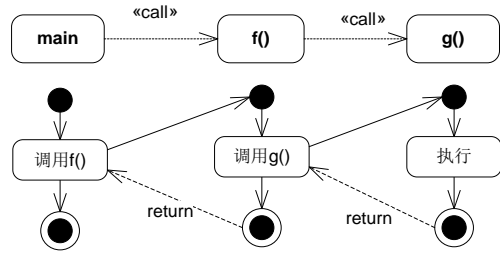


图 5-2 嵌套调用及返回

例 5-3 输入正整数  $n$  和  $k$ ，计算  $1^k+2^k+3^k+\dots+n^k$  的值。

```

#include <iostream>
using namespace std;
int powers(int n, int k){
    int product = 1;
    for(int i = 1; i <= k; i++)
        product *= i;
    return product;
}
int sump(int k, int n){
    int sum = 0;
    for(int i = 1; i <= n; i++)
        sum += powers(i, k);
    return sum;
}
int main(void){
    int n, k;
    cout<<"input n and k:";
    cin>>n>>k;
    if (n >= 1 && k >= 1)
        cout<<sump(k, n)<<endl;
    else
        cout<<"input error";
    system("pause");
    return 0;
}

```

函数 `powers` 用来计算  $n^k$ ，函数 `sump` 通过调用函数 `powers` 来求  $1^k+2^k+3^k+\dots+n^k$  的值，函数 `main` 调用 `sump` 来输出计算结果。

读者可自行运行测试上面程序。

### 5.5.2 函数的递归调用

递归调用(recursive call)是一种特殊的函数调用，如果一个函数在其函数体中直接或者间接地调用了自己，这个函数就是一个递归调用函数。如果是直接调用自己为直接递归；间接调用自己为间接递归。

为什么需要递归调用？一个函数何时需要调用自己？在解决一个复杂问题时，常常分解

成若干个子问题。如果每个子问题的性质与原来问题相同，只是处理的数据比原问题更小，这时就可通过对各个子问题的求解和综合来解决整个问题，而每个子问题的求解过程也可采用与原问题相同的分解与综合的方式来解决。递归函数为上述设计方法提供了一种自然、简洁的实现机制。比如，数学归纳法所表示的计算往往适用递归函数来实现。

例 5-4，求阶乘  $n! = 1*2*3*...*n$ 。用数学归纳法表示为  $n! = n * (n-1)!$ ， $n \geq 2$ 。

如果  $n$  大于 1 时，求  $n!$  的问题就能分解为求  $(n-1)!$ ，然后再乘以  $n$ 。那么这个分解过程可以持续进行，一直到求  $1!$ 。此时整个问题就解决了。编程如下：

```
#include <iostream>
using namespace std;
unsigned factorial(unsigned n){
    if(n == 1)
        return 1;
    else
        return n * factorial(n-1);    //递归调用
}
int main(){
    unsigned n;
    cin>>n;
    cout<<n<<"!="<<factorial(n)<<endl;
    system("pause");
    return 0;
}
```

上面 `factorial` 函数设计简单易懂。读者可自行测试以上程序。

在定义递归函数时，应注意两个关键：

(1) 结束条件。结束条件指出何时不需要递归调用，而能直接得到结果。例如在上述例子中  $n==1$  即为结束条件，返回 1 作为结果。

(2) 降低  $n$  或趋向结束条件。这确定了问题的分解和综合过程，属于问题求解的一般情形。例如在上例中， $n$  不为 1 时就计算  $n-1$ 。

递归调用本质上是一种嵌套调用，只不过是嵌套调用自身。递归函数的一次调用执行包括递推和回归两个过程。对于上面例子，如果输入  $n=4$ ，那么递归函数的递推和回归过程如图 5.3 所示。

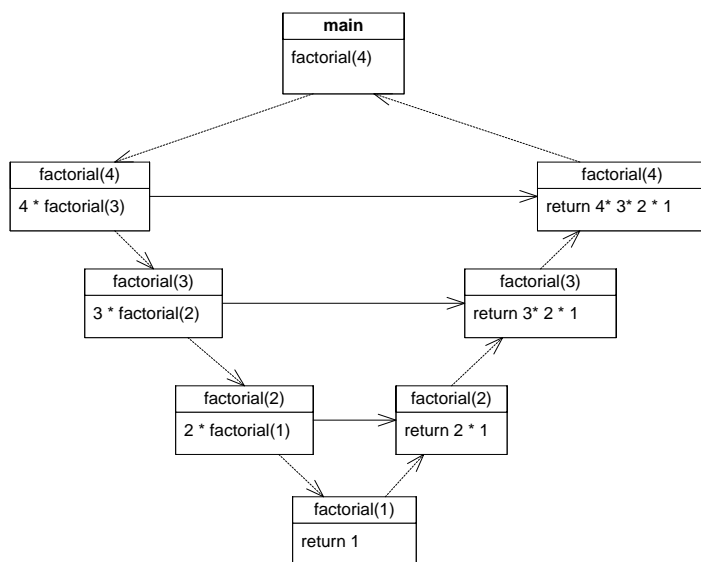


图 5-3 递归调用的递推和回归过程

图中沿虚线箭头向下的方向表示了递推的过程，一直持续到嵌套调用停止。沿虚线箭头向上的方向表示了回归的过程，每一个回归都对应水平方向上返回的一个表达式结果，最终到达主函数。

不采用递归调用，实现上面 **factorial** 函数的一种形式如下：

```

unsigned factorial2(unsigned n){
    unsigned s = 1;
    for (unsigned i = 1; i <= n; i++)
        s *= i;
    return s;
}
  
```

比较这两种实现方式，递归函数更接近数学归纳法的描述，可读性更好。而非递归的实现则具有更高的计算效率，因为函数的调用和返回需要一些额外的开销，这些开销比简单的加法和乘法要大得多。

在递归函数中一般是先判断递归结束条件，然后进行递归调用；否则在执行程序时，就可能产生无穷尽的递归调用。

例 5-5 求 Fibonacci 数列的前 30 个数，要求每行输出五个数。Fibonacci 数列的递归公式为：

$$f_n = \begin{cases} 1 & n = 1 \\ 1 & n = 2 \\ f_{n-1} + f_{n-2} & n > 2 \end{cases}$$

显然，递归调用条件是  $n > 2$ 。编码如下：

```

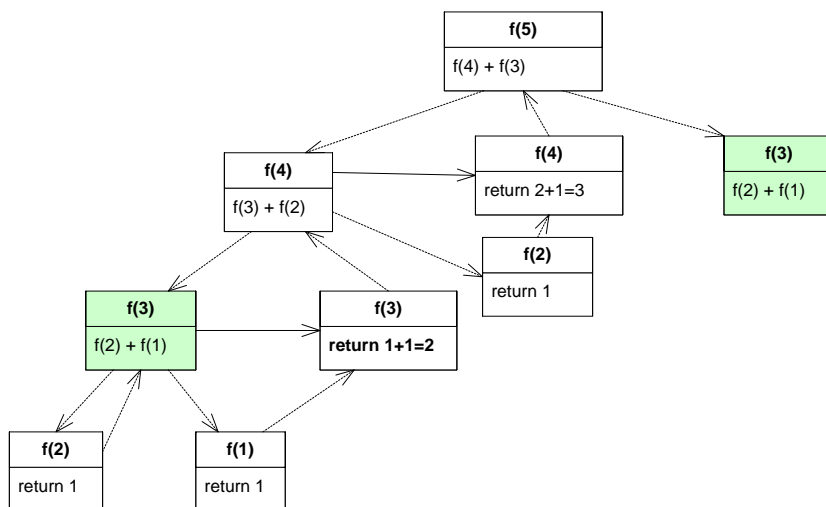
#include <iostream>
#include <iomanip>
using namespace std;
int f(int n){
    if (n == 1 || n == 2)           //递归结束条件
        return 1;
    else
        return f(n - 1) + f(n - 2); //进行递归调用
}

int main(void){
    int i ;
    for (i = 1 ; i <= 30 ; i ++){
        cout <<setw(10) << f(i);
        if ( i % 5 == 0 )
            cout << "\n";
    }
    cout << "\n";
    system("pause");
    return 0;
}

```

对上面例子,  $f$  函数的设计非常接近数学归纳法所表示的递归公式。但仔细分析其递推过程就能发现重复计算的问题。例如, 用  $n=5$  来调用递归函数  $f(5)$ , 递推过程如图 5.4 所示。我们就能发现  $f(3)$  重复计算了 2 次。当计算  $f(n)$  时,  $f(n-2)$  就要重复计算两次, 而且应加以递归。所以当  $n$  的值比较大时, 重复计算将导致很大的开销。

另一方面, 在 `main` 函数中调用 30 次  $f$  函数, 实际上做了大量无效的计算。例如当  $i=30$  来调用  $f(i)$ , 那么在  $f$  函数中就要计算  $f(29)$  和  $f(28)$ 。实际上, 在前面 `main` 函数中已经计算过这两个值了, 因为  $i$  是从 1 循环到 30 的。



对于同一个问题，既可用递推又可用递归来设计时，哪一种更好呢？通常，使用递归，程序简洁易懂；而使用递推，程序的执行速度要快得多。但有些问题并不容易用递推来解决。最典型的例子是河内(Hanoi)塔问题。

例 5-6 河内塔问题。如图 5.5 所示，有三根柱子 A、B、C。设 A 柱上有  $n$  个盘子，每个盘子大小不等，大盘子在下，小盘子在上。要求将 A 柱上的  $n$  个盘子移到 C 柱上，每一次只能移动一个盘子，在移动的过程中，可以借助于任一根柱子作为过渡，但必须保证三根柱上的盘子都是大盘在下，小盘在上。要求输入任意  $n \geq 1$ ，编程输出移动盘子的步骤。

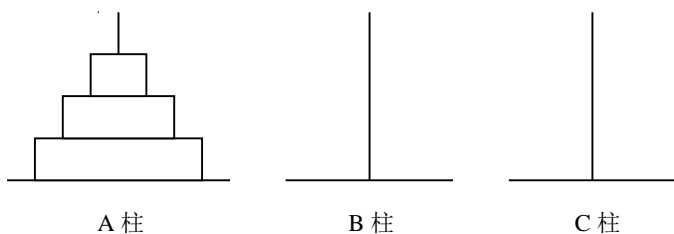


图 5-5 Hanoi 塔问题

先从实例分析入手。当  $n=1$  时，只需 1 步就行。当  $n=2$  时，需要以下 3 步：

1. A 到 B，把上面的小盘子移动到过渡柱子上；
2. A 到 C，把下面的大盘子移动到目标柱子上；
3. B 到 C，把过渡柱子上的小盘子移动到目标柱子上，结束。

当  $n=3$  时，需要以下 7 步：

1. A 到 C
2. A 到 B
3. C 到 B，此时 A 柱上面两个小盘子被移到 B 柱上过渡。
4. A 到 C，把下面最大盘子移动到柱子 C 上。此时 A 柱子为空，可作为过渡柱子使用。
5. B 到 A
6. B 到 C
7. A 到 C，此时 B 柱子上的 2 个盘子移动到目标柱子上，结束。

我们对上面求解过程进行归纳，就能发现可将移动  $n$  个盘子的问题简化为移动  $n-1$  个盘子的问题。将  $n$  个盘子从 A 柱经 B 柱移到 C 柱可分解为下面 3 步：

(1) 将 A 柱上的  $n-1$  个盘子经 C 柱移到 B 柱上。此时 C 柱为空。当  $n=2$  时，对应第 1 步。当  $n=3$  时，对应前 3 步。

(2) 将 A 柱上的最大一个盘子移到 C 柱上。此时 A 柱为空。当  $n=2$  时，对应第 2 步。当  $n=3$  时，对应第 4 步。

(3) 将 B 柱上的  $n-1$  个盘子(B 柱上的所有盘子)经 A 柱移到 C 柱上。当  $n=2$  时，对应第 3 步。当  $n=3$  时，对应后 3 步。

假设把  $n$  个盘子从 A 柱移到 C 柱的移动次数为  $f(n)$ ，那么  $f(n) = 2 * f(n-1) + 1$  且  $f(1)=1$ 。显然这是一个递归公式，非常容易实现。

上面第(1)步和第(3)步是移动  $n-1$  个盘子，只是柱子改变了。这种分解可一直递推下去，直到移动一个盘子才结束。其实以上 3 个步骤只包含两种操作：

(1) 将  $n$  个盘子从  $x$  柱经  $y$  柱移到  $z$  柱上, 是一个递归函数, 用 `hanoi(int n, char x, char y, char z)` 函数实现。

(2) 将一个盘子从  $a$  柱移到  $b$  柱, 用函数 `move(char a, char b)` 实现, 其实就是输出移动盘子的提示信息。编程如下:

```
#include <iostream>
using namespace std;
void move(char a, char b){
    cout<<a<<" to "<<b<<endl;
}
void hanoi(int n, char x, char y, char z){
    if ( n == 1 )
        move(x, z);                //移动一个盘子
    else{
        hanoi (n - 1, x, z, y);    //递归调用
        move(x, z);                //移动一个盘子
        hanoi (n - 1, y, x, z);    //递归调用
    }
}
unsigned hanoiCount(int n){          //计算 n 个盘子需要移动的次数
    if (n == 1)
        return 1;
    else
        return hanoiCount(n - 1) * 2 + 1;
}
int main(void){
    int n;
    while(1){
        cout << "Input number of plates(0 to exit):";
        cin >> n;
        if (n == 0)
            break;
        if (n < 0){
            cout<<"input error, retry"<<endl;
            continue;
        }
        hanoi( n, 'A' , 'B' , 'C');
        cout<<hanoiCount(n)<<" steps\n";
    }
    system("pause");
    return 0;
}
```

注意, 执行以上程序时, 输入的盘子数不能大(不要大于 16), 否则无法完成计算(可能会导致堆栈溢出)。

上面程序中也设计了一个函数 `hanoiCount(int n)` 来计算  $n$  个盘子的移动次数, 直接使用了递归公式, 用递归函数来实现。看起来很简单, 但能否更简单? 不用递归, 也不用循环, 是否能解决此问题。经过分析可知, 移动  $n$  个盘子的次数就是  $2^n - 1$ , 从二进制位来看, 就是  $n$  个 1 位。那么我们就设计一个更高效率的函数如下:

```
unsigned hanoiCount2(int n){
    if (n < 32)
        return (1 << n) - 1;    //n 个 1 位
    if (n == 32)
```

```

        return -1;                //unsigned 的最大值为全 1 位
    return 0;
}

```

由于 C++ 整数 `int` 最大值为 32 位，所以用整数 `int` 最多能计算 32 个盘子的移动次数。如果要移动 64 个盘子，假设一秒钟移动 1000 次，需要多少年。读者可以自行设计计算。

## 5.6 函数原型

一般来说，函数先定义后调用。但 C++ 程序也允许先说明函数原型，再调用函数，在别的地方定义函数。一个完整的函数定义包含一个函数头和一个函数体。其中函数头就是函数原型(**prototype**)，包括了函数的名称、形参及返回值，确定了函数调用的合法形式。就可将函数的原型说明与定义分离，先说明函数原型，使下面代码能调用该函数，就能检查函数调用是否符合语法规则。而函数的完整定义就可以放在后面，或者另一个源文件中，或者编译后的库文件中(如 `lib` 文件)。一个函数只能定义一次，但原型说明(**declare**，也称为声明)可以多次出现。

函数原型说明由函数返回类型、函数名和形参表组成。形参表必须包括形参类型，但不必对形参命名。这三个元素被称为函数原型。

函数原型说明的一般形式为：

<返回类型> <函数名>(<形参表>);

函数原型说明与函数定义在返回类型、函数名和形参类型上必须一致。一个函数原型说明是一条说明语句，必须以分号“;”结束。以下是几点说明。

(1) 一个函数的原型说明应该出现在该函数调用之前。通常在源文件中的调用方函数定义之前。

例 5-7 列出 100 以内的所有素数。

```

#include<math.h>
#include<iostream>
using namespace std;
bool isPrime(unsigned n);           //A 函数原型说明
int main(){
    for(int i = 2; i <100; i++)
        if (isPrime(i))           //B 函数调用
            cout<<i<<'\\t';
    system("pause");
    return 0;
}
bool isPrime(unsigned n){          //C 函数定义
    if (n < 2)
        return false;
    if (n == 2 || n == 3 || n == 5 || n == 7)
        return true;
    for (int i = 2; i <= sqrt(n); i++)
        if (n % i == 0)
            return false;
    return true;
}

```

程序中的 A 行是一个函数的原型说明。如果去掉 A 行代码，编译器就会给出编译错误信息，指出 B 行中的函数调用是错误的。C 行开始时函数的完整定义。

(2) 在函数原型说明中，<形参表>可以只列出形参的类型而不写名称。例如，在上面例子中，A 行的函数说明可以写成：

```
bool isPrime(unsigned);
```

对原型说明中的形参，也可以任意给出一个标识符，不一定要与函数定义的形参名相同。如：

```
bool isPrime(unsigned x);
```

现在我们就知道为什么需要函数原型说明了。对于一组通用设计的函数，希望能被更多的人在更多的编程中调用，但又不能公开源代码而被人篡改或滥用，也不想反复编译而浪费时间。将这些函数原型说明在一个头文件中，函数定义放在其它源文件中，再将这些源文件编译为库文件(如 lib 或 dll 文件)。然后将头文件和库文件公开给他人使用。就像我们使用 math.h 中的 sqrt 函数一样，先用#include<math.h>来说明函数原型，后面代码就能调用其中说明的函数，编译器能检查调用合法性，同时能避免对被调用函数重复编译。

附录 B 列出很多头文件及其函数，读者可尝试调用更多函数来简化自己的编程。

## 5.7 auto 函数

C14 能根据函数的 return 表达式来推断函数的返回类型。如果函数中含有多个 return 表达式，这些表达式必须可推断为相同类型。

auto 函数可以有递归，但递归调用必须在函数定义中的至少一个 return 语句之后。例如：

```
auto func(int i) {  
    if (i == 1)  
        return 1; // return type deduced as int  
    else  
        return (func(i - 1) + i);  
}
```

下面这种情形不能推导：

```
auto wrong(int i) {  
    if (i != 1)  
        return (wrong(i - 1) + i); //编译错误  
    else  
        return i; // return type deduced as int  
}
```

auto 函数的调用可以与 auto 变量初始化相结合，例如：

```
auto v = func(5);  
cout << func(v) << endl;
```

函数形参不能为 auto，但 lambda 函数的形参可以为 auto，以后介绍。

注意 auto 函数在调用之前必须定义，而不能在说明 auto 原型之后就调用，然后别处再定义该函数。这意味着在头文件中不允许仅说明 auto 函数的原型，而应包含其定义。



## 5.8 特殊参数

特殊参数包括带缺省值的形参和可变参数。在定义函数时可确定形参的缺省值，那么在调用时就能缺省该实参。可变参数是指函数定义中说明了形参个数不固定，在调用时可提供任意多个实参。

### 5.8.1 带缺省值的形参

一般来说，一个函数调用所提供的实参数量应该与被调用函数的形参数量是一样的。但如果一部分形参带缺省值，那么函数调用的实参数量就可以减少，缺省值就可以起作用了。在函数定义时为一些形参说明缺省值，好处就是能减少函数调用的实参数量。当然，如果函数调用提供了实参，那么对应形参的缺省值就不起作用。形参的缺省值只有当实参缺省时才起作用，这是 C++ 语言的特色之一。

何时需要为形参说明缺省值？对于一个函数的一个形参，如果多数的函数调用的实参都用某个值，而且这个形参是最后一个，就可以为此形参说明这个值作为缺省值，这样多数函数调用就可用这个缺省值来进行调用。

例如下面一个函数 `print` 以某种进制形式输出一个整数值，形参 `base` 表示用什么进制，`value` 是要输出的值。

```
void print(int value, int base);
```

在多数情况下要以十进制来输出整数值，例如：

```
print(x, 10);
```

偶尔也会以其它进制来输出，如 “`print(7, 2);`”。

在 C++ 中，允许在定义或说明函数时，为某些形参指定缺省值。当调用这些函数时，如果没有提供相应的实参，那么相应的形参就用指定的缺省值。例如，上面 `print` 函数可以说明如下：

```
void print(int value, int base=10);
```

这样，如果要调用该函数以十进制输出某个整数值，则可以省略第 2 个实参，而采用下面的调用形式：

```
print(x);
```

编译程序会根据相应说明中的缺省值，把上述函数调用编译为 `print(x, 10)`。当然，如果要以其它进制输出，调用时仍要提供函数的第二个实参。例如，以二进制输出 `x` 值的函数调用如下：

```
print(x, 2);
```

形参的缺省值能简化函数调用，但应注意下面几点：

(1) 带缺省值的形参应处于形参表的最后面。看下面例子。

例 5-8 函数形参的缺省值示例。

```

#include <iostream>
using namespace std;
void f(int a, int b=1, int c=0){           //A
    cout << "a=" << a << '\t';
    cout << "b=" << b << '\t';
    cout << "c=" << c << endl;
}
int main(void){
    int x=0, y=1, z=2;
    f(x, y, z);           //A 正确
    f(x, y);              //B 相当于 f(x,y,0)
    f(x);                 //C 相当于 f(x,1,0);
    f();                  //D 错误, 第一个形参没有缺省值
    system("pause");
    return 0;
}

```

如果把 A 行修改成如下代码都是错误的:

```
void f(int a, int b=1, int c)
```

或

```
void f(int a=2, int b, int c)
```

或

```
void f(int a=2, int b=1, int c)
```

或

```
void f(int a=2, int b, int c=3)
```

也就是说, 从第一个带缺省值的形参开始, 其右边的所有形参都应指定缺省值。

(2) 必须在函数调用前就指定缺省值。在函数原型说明中可以指定形参的缺省值, 而在其后定义函数时, 就不必再指定缺省值。这是因为形参缺省值是提供给编译器在检查函数调用时使用的, 当发现调用函数中没有提供相应实参时, 编译器就使用默认值。例如:

```

void f(int a, int b=1, int c=0);    //A 函数原型中给出缺省值

int main(void){
    int x=0, y=1, z=2;
    f(x, y, z);
    f(x, y);
    f(x);
    return 0;
}

void f(int a, int b, int c){        //B 函数定义中没有缺省值
    cout << "a=" << a << '\t';
    cout << "b=" << b << '\t';
    cout << "c=" << c << endl;
}

```

A 行是函数 f 的原型说明, 指定了后两个形参的缺省值。如果 B 行函数定义语句中再次指定形参默认值, 不论两处指定的形参默认值是否相同, 编译器都认为是重复指定, 会报错。

(3) 注意重载函数时, 同名函数的形参缺省值可能发生二义性错误。例如:

```
void f(int a, int b=1, int c=0);    //A
```

```

void f(int a, int b);                //B

int main(void){
    int x=0, y=1, z=2;
    f(x, y, z);
    f(x, y);                        //C 出错
    f(x);
    return 0;
}

```

A 行和 B 行的两个函数是重载函数，编译时没有错误。但 C 行报错，这是因为函数调用  $f(x, y)$  具有二义性，不能确定调用哪一个函数。

### 5.8.2 可变参数

根据 ANSI C89 标准，函数的形参定义可包括若干固定形参和一个可变形参(...)。在形参表中，固定形参在前，可变形参在后。在调用时，先提供固定实参，对于可变形参，可提供 0 个、1 个或多个实参，在函数体中要调用 `<stdarg.h>` 中定义的 3 个函数(或宏)和 1 个类型来获取实参值，再完成计算。

例 5-9 用可变参数求若干个正整数的平均值，用 -1 表示结尾。

```

#include<iostream>
#include<stdarg.h>
using namespace std;
int average(int first, ...);        //A
int main(){
    int i = 3, j = 4;
    cout<<"Average is: "<<average( 2, i, j, -1 )<<endl;    //B
    cout<<"Average is: "<<average( 5, 7, 9, 11, -1)<<endl;    //C
    cout<<"Average is: "<<average( -1 )<<endl;            //D
    system("pause");
    return 0;
}
int average( int first, ... ){
    int count = 0, sum = 0, i = first;
    va_list marker;                //E
    va_start( marker, first );      //F
    while( i != -1 ){
        sum += i;
        count++;
        i = va_arg( marker, int);    //G
    }
    va_end( marker );              //H
    return( sum ? (sum / count) : 0 );
}

```

执行程序，输出结果为：

```

Average is: 3
Average is: 8
Average is: 0

```

A 行定义函数原型 `average` 中包含一个 `int` 形参和一个可变形参，用 3 个连续的点来表示可变形参，这要求调用函数时至少提供一个 `int` 实参作为固定实参。该函数约定对 0 个到多个 `int` 值计算平均数，而且以 -1 结尾。

B 行调用提供了 4 个实参，而最后 -1 作为结尾标记，因此有 3 个实参与计算。

C 行调用有 5 个实参，对 4 个实参计算平均值。

D 行仅有 1 个实参-1，表示没有实参与计算，约定返回 0。

E 行 `va_list` 表示实参表类型，`marker` 就是当前实参表的名字。

F 行调用 `va_start` 函数来对实参表中的可变实参初始化，确定可变实参的位置，从 `first` 之后的实参都是可变实参。

G 行调用 `va_arg` 函数来获取下一个实参值，第一个实参是当前实参表，第二个实参确定实参的类型，该函数返回下一个实参值。一般在循环语句中调用该函数，就能循环读出所有的实参。函数中要约定实参结束的条件，例子中用-1 表示结束，也可用第一个实参来确定后面实参的个数，例如 `int average(int num,...)`。注意，没有其它办法能知道实参的个数。

最后，H 行调用 `va_end` 函数对实参表 `marker` 复位。

在定义可变参数函数时，必定要用到 1 个类型 `va_list` 和 3 个函数 `va_start`、`va_arg` 和 `va_end`，必须包含头文件 `<stdarg.h>`。通常有如下 4 个步骤。

- 1、说明一个 `va_list` 类型的变量。如上例中的 `marker` 变量。`va_list` 是头文件中定义的一种数据类型，只有通过这种类型的变量才能从实参表中取出可变实参。

- 2、初始化可变实参。调用 `va_start` 函数，提供两个实参，一个是 `va_list` 类型的变量，另一个是函数的形参表中最后一个固定形参的变量名，即省略号“...”前的一个变量名(如 `first`)。该函数使 `va_start` 中的第一个实参指向实参表中的第一个可变实参，并为读取第一个可变实参作好准备。

- 3、循环读取实参。调用 `va_arg` 函数，提供两个实参，第一个实参与 `va_start` 的第一个实参相同(`marker`)，表示当前可变实参，第二个实参应该是一个 C++ 中预定义的数据类型名(如例中的 `int`)。该函数的作用就是将当前可变实参转换为指定类型的数据，并返回；同时使 `va_arg` 的第一个实参(`marker`)指向下一个可变实参，为读取下一个可变实参作好准备。

- 4、实参表复位。调用函数 `va_end`，提供一个实参，与 `va_start` 函数的第一个实参相同。该函数的作用是完成可变实参的收尾工作，以便函数能正常返回。注意，在定义可变参数的函数体中，在 `return` 语句之前必须调用 `va_end` 一次，否则就可能会出现不可预测的错误。

在实际编程中，用户自定义的可变参数函数比较少见。较多见的是 `<stdio.h>` 中定义的 `scanf`、`printf` 等函数。这些函数的形参表中，前面形参决定了后面形参的个数和格式，作为 C 语句最主要的格式化 IO 方式。本文后面没有再用到可变参数。

使用可变参数时，应注意以下几点。

- (1) 定义函数时，形参表中的固定形参必须放在可变形参前面，用省略号“...”表示有可变形参。函数调用时可以只提供固定实参，如例中的 D 行。

- (2) 在读取第一个可变参数前必须调用函数 `va_start` 一次，完成初始化工作；然后使用 `va_arg` 依次取各个可变的参数值；最后调用函数 `va_end` 做好收尾工作。

- (3) 函数体内必须能够知道实际调用的实参个数及类型。这需要对函数调用的实参进行明确的约定。

- (4) 建议不要对可变参数的函数定义重载函数。

## 5.9 内联函数 inline

所谓内联函数就是用关键字 `inline` 修饰的函数，该函数的所有调用都被替换为该函数的函数体。一个函数是否内联仅对性能有作用，不影响其功能。

何时需要内联函数？由于函数的调用和返回都有一定的时间和内存的开销，如果你定义了一个函数，但又想避免调用返回的开销，就可说明该函数为内联 `inline` 函数。

例如，下面给出了 `max` 的内联函数实现：

```
inline int max(int a, int b){  
    return (a > b ? a : b);  
}
```

对于一个内联函数，编译器尝试用其函数体来替代其调用，这样就能避免函数调用和返回的开销。不过内联函数的代价是使代码变得更长。也就是说，内联函数可能会用更长的可执行代码来换取更高的执行效率，就是用空间换时间。内联函数对程序设计没有直接影响，也难以验证是否真正起作用。注意不要对递归函数使用内联。

## 5.10 作用域

每个标识符(如变量名)都有确定的说明位置，决定了它将在什么范围内有效(能被访问)。所谓作用域(scope)就是标识符的有效区域。每个作用域也可看做是一个命名空间，其中不允许出现重名定义。

在 C++ 中，作用域共分为五类：块作用域、文件作用域、函数原型作用域、函数作用域和类作用域。下面介绍前四种作用域，类作用域在后面章中介绍。

### 5.10.1 块作用域

一个块(block)就是一条复合语句，用一对花括号“{ }”括起来的部分程序。块可以嵌套，即一个块中可嵌套内层块，而且嵌套层数没有限制。块之间不能有交叉，也就是说，没有一条语句能同时位于两个非嵌套的块中。

块作用域也被称为局部作用域(local scope)，是指在一个块内说明的标识符(如局部变量)，其作用域始于标识符的说明处，止于该块的结尾处，可以被其内层块访问。具有块作用域的变量只能在该变量说明处到块尾之间使用，而不能在其它区域使用。

例 5-10 块作用域的例子。

```
#include <iostream>  
using namespace std;  
int main(void){  
    int i, j;  
    i = 1; j = 2;  
    if (i < j){  
        int a, b;  
        a = i++;  
        b = j;  
        cout<<a<<'\\t'<<b<<endl;  
    }  
}
```

```

    cout<<i<<'\\t'<<j<<endl;
    system("pause");
    return 0;
}

```

上面程序中有两个块。第一个是 `main` 函数块，其中说明了变量 `i`、`j`，其作用域是从定义到函数结束。第二个块是嵌套块，其中定义了变量 `a`、`b`，其作用域仅限于 `if` 语句块，而不能在块外使用。嵌套块中可使用其外层块中先前说明的变量 `i` 和 `j`。

在内层嵌套块中可说明与外层块中同名的标识符，而且在内层嵌套块中按名使用优先。

例 5-11 同名变量的作用域示例。

```

#include <iostream>
using namespace std;
int main(void){
    int i=1,j=2,k=3;
    cout<<i<<'\\t'<<j<<'\\t'<<k<<endl;
    if(i < k){
        int i=5, j=6;           //A
        k = i + j;               //B k = 5 + 6
        cout<<i<<'\\t'<<j<<'\\t'<<k<<endl;
    }
    cout<<i<<'\\t'<<j<<'\\t'<<k<<endl;    //C
    system("pause");
    return 0;
}

```

上面程序中嵌套块中 A 行说明的变量 `i` 和 `j` 与外层说明的变量同名。B 行使用 `i`、`j` 时，访问的是内层说明的变量，而不是外层的同名变量，因为内层标识符优先，就隐藏了其外层块中的同名标识符，就不能访问外层块中被隐藏的同名变量 `i` 和 `j`。

当程序执行到 C 行，退出了内层块，那么内层块中说明的变量就不存在了。

执行程序，输出结果为：

1	2	3
5	6	11
1	2	11

在函数内说明的一个函数原型也具有块作用域，从说明处开始到函数体结束都是该原型的有效范围，即能被有效调用。

函数的形参也具有块作用域，从说明处开始到函数体结束都是形参标识符的有效范围。函数体内定义的局部变量不能与形参重名。

`for` 语句中的第一个表达式中说明的循环控制变量具有块作用域，其作用域为 `for` 语句之内。在 VC2015 或 Dev-C++(GCC)中，其作用域仅作用于 `for` 语句之内。但在 VC6 中，其作用域被扩张到 `for` 语句之后到块尾，并非仅作用于 `for` 语句之内。例如：

```

int main(){
    for(int i=0;i<10;i++){
        cout<<i*i<<'\\t';    //for 语句中说明的变量 i 具有块作用域
    }
    cout<<i;                  //VC++允许，标准 C++不允许
    ...
}

```

具有块作用域的变量就是局部变量。局部变量是存储在系统堆栈中的。堆栈是一种“先进后出”的数据结构。外层块中说明的局部变量要先入栈(开辟空间)，内层块中说明的局部变

量后入栈。在当前堆栈中的变量能被访问。当内层块执行结束时，内层块的局部变量先出栈(回收空间)，就不能再访问了，但能访问外层块中的局部变量。局部变量随着程序执行动态开辟和回收空间，所以不同块中用同一个名字能访问不同的变量。如上面例子中的B行和C行。

### 5.10.2 文件作用域

文件作用域(file scope)是说明在所有的块和类之外的标识符的作用范围，从说明开始到当前文件结束。

在函数外定义的变量具有文件作用域。具有文件作用域的变量可在当前文件中被访问，如果它不是静态的(static)，就被称为全局(global)变量。在多文件组成的程序中，非静态全局变量可被其它文件使用，而静态全局变量局限于当前文件中使用。

例 5-12 文件作用域变量示例。

```
#include <iostream>
using namespace std;
int i = 11, j = 22, k = 33;           //A
int main(void){
    cout<<i<<'\\t'<<j<<'\\t'<<k<<endl;    //B
    int i=1,j=2,k=3;                 //C
    cout<<i<<'\\t'<<j<<'\\t'<<k<<endl;    //D
    if(i < k){
        int i=5, j=6;
        k = i + j;                   //k = 5 + 6
        ::i += 2;                    //E
        cout<<i<<'\\t'<<j<<'\\t'<<k<<endl;
    }
    cout<<i<<'\\t'<<j<<'\\t'<<k<<endl;
    cout<<::i<<'\\t'<<::j<<'\\t'<<::k<<endl; //F
    system("pause");
    return 0;
}
```

上面程序中A行说明的3个变量是文件作用域的有效范围，即它们可被同一个文件中下面所有的函数访问。例如main函数中的B行。内层块中如果说明了相同名称的标识符，将隐藏外层的同名标识符，所以C行说明了内层作用域中的3个变量，使D行能访问，因为内层优先。

在内层块中，如何访问文件作用域中的定义的同名全局变量？要使用一个**全局作用域运算符“::”**。该运算符全称为“作用域解析运算符”，全局作用域只是一种用法，后面还有其它用法。

例如，E行中用“::i”来确定被访问的是全局变量i，而不是局部变量i。F行中也使用了这个运算符来访问全局变量。

执行上面程序，输出结果为：

11	22	33
1	2	3
5	6	11
1	2	11
13	22	33

关于全局变量，后面还将介绍全局变量的存储。

5.10.3 函数原型作用域

函数原型作用域(简称为原型作用域 `prototype scope`)是在函数原型形参表中说明的标识符所具有的作用范围, 它从形参变量(标识符)定义开始到函数原型说明结束。由于函数原型中说明的形参标识符与该函数的定义和调用无关, 仅有形式上的意义, 只要求多个形参不重名即可。

(1) 函数原型的形参标识符可与其定义不同, 例如:

```
float function (int x,float y);           //函数 function 的原型说明
...
float function (int a,float b)           //函数 function 的定义
{ 函数体 }
```

(2) 函数原型的形参标识符可省略, 例如:

```
float function (int,float);
```

5.10.4 函数作用域

函数作用域(`function scope`)专门针对标号语句中的标识符, 从函数开始到函数结束的整个范围内均有效。注意, 函数体内说明的局部变量具有块作用域(从说明处到块结束), 并非函数作用域。函数体内说明的标号在整个函数中都能被 `goto` 语句使用, 所以 `goto` 语句能往后跳, 也能往前跳, 但不允许 `goto` 语句跳转到另一个函数。但由于结构化设计不提倡使用 `goto` 语句, 所以函数作用域也没有多大用处。

5.11 程序运行期的存储区域

一个 C++程序在启动执行前, 系统为它分配内存空间。一个程序的存储空间在逻辑上可分为四个区域, 如图 5.6 所示。

- (1) 代码区: 存放可执行代码, 在执行过程中该区域不伸缩。这个区域也被称为代码段。
- (2) 静态存储区: 存放程序中定义的静态变量, 在执行过程中该区域不伸缩。
- (3) 全局存储区: 存放程序中定义的全局变量。静态存储区和全局存储区合起来被称为数据段。数据段的大小在执行期间不会动态伸缩。
- (4) 动态存储区: 以堆栈存放程序中定义的局部变量, 在程序执行过程中动态伸缩。这个区域也被称为堆栈段。

代码区: 存放可执行代码
静态存储区: 存放静态变量
全局存储区: 存放全局变量



动态存储区：存放局部变量

图 5-6 一个 C++程序的存储区域划分

### 5.11.1 静态存储区

在程序设计中往往需要定义一些静态变量。静态变量就是用 `static` 说明的变量。静态变量可以说明在块作用域中(就像局部变量),也可以说明在文件作用域中(就像全局变量)。但在程序运行时,所有静态变量都存放在静态存储区中,在程序加载时就确定了静态存储空间,而且在执行过程中不会动态伸缩。当程序结束时才回收静态存储区。这样我们就知道,静态变量的生存期就是整个程序的运行期。下面将详细介绍静态变量。

### 5.11.2 全局存储区

全局存储区就是用来存储所有的全局变量。与静态变量相似,在程序运行时,所有全局变量都存放在全局存储区中,在程序加载时就确定了全局存储空间,而且在执行过程中不会动态伸缩。当程序结束时才回收全局存储区。全局变量的生存期就是整个程序的运行期。

### 5.11.3 动态存储区

动态存储区是以堆栈形式来存储局部变量的区域。在程序加载进入内存时,由操作系统动态确定一个堆栈区域。当程序执行从一个外层块进入一个内层块时,内层块中的局部变量就被“压入 `push`”堆栈,然后内层块中的代码就能访问这些局部变量。当程序执行退出一个内层块返回到外层块时,这些内层块中局部变量将从堆栈中“弹出 `pop`”,空间被回收,就不能再访问了。随着函数的调用和返回,随着块的进入和退出,堆栈的使用区域是动态伸缩的。这样我们就能知道,局部变量的生存期就是其块作用域执行期间。

## 5.12 存储类

存储类(storage-class)确定了变量占用内存的期限、可见性以及存储区域。在 C++中,变量的存储类分为四种:自动(auto)、寄存器(register)、静态(static)、外部类型(extern)。每个变量只能选择其中之一。

### 5.12.1 auto 变量

在 C11/VC2010 之前,关键字 `auto` 修饰的局部变量称为自动变量。这里自动意味着它是动态存储的。自动变量一定是局部的,只能定义在块内,其作用域为块作用域。局部变量缺省为自动变量,因此在局部变量说明时通常都不用 `auto` 修饰。全局变量一定不是自动变量。C11/VC2010 之后, `auto` 变量的含义改变为编译器自动推导变量的类型。

### 5.12.2 register 变量

用 `register` 修饰的局部变量称为寄存器变量。`register` 指示编译器尽可能利用 CPU 寄存器来存储变量，以提高变量的存取速度。例如：

```
for(register int a = 1; a < n; a++){  
    ...  
}
```

局部变量可定义为寄存器变量。静态变量和全局变量不能定义为寄存器变量。

因 CPU 寄存器数量有限，且有特定用途，可用作变量存储的寄存器很少，故尽管所有 C++ 编译系统在语法上都支持 `register` 关键字，但其实大多数编译系统并未兑现。C11 标记 `register` 为过时弃用 `deprecated`。

### 5.12.3 static 变量与多文件项目

静态变量就是用关键字 `static` 修饰的变量。静态变量都存储在静态存储区中。在说明静态变量时，若没有指定初值，编译器将其初值置为 0(`bool` 类型将为 `false`)。例如下面代码：

```
static float y=4.5f;           //静态全局变量 y，初值 4.5  
static char s;                 //静态全局变量 s，初值 0  
void f(void){  
    static int x;              //静态局部变量 x，初值 0  
    cout<<x<<','<<y<<','<<(int)s<<endl;  
}
```

说明在文件作用域中的静态变量称为静态全局变量。例如上面的 `y` 和 `s`。说明在块作用域中的静态变量被称为静态局部变量。例如上面的 `x`。它们都存储在静态存储区中，生存期就是整个程序的运行期。

#### (1) 静态局部变量

静态局部变量的特点：在程序开始执行时就获得所分配的内存，其生存期是全程的，而作用域是局部的。在程序执行退出其作用域时，系统并不收回其所占内存，当下次执行该函数时，该变量仍保持在内存中，仍保留原来的值。

静态局部变量的作用：保存函数每次执行后的结果，以便下次调用函数时，继续使用上次计算的结果。其它函数不能访问该变量。

例 5-13 静态局部变量示例。

```
#include <iostream>  
using namespace std;  
int f(int i){  
    static int r = 1;          //说明静态局部变量  
    r *= i;                    //改变静态局部变量  
    return r;                  //返回静态局部变量  
}  
int main(void){  
    for(int i = 1; i < 5; i++)  
        cout<<i<<"!="<<f(i)<<'\n';  
    system("pause");  
    return 0;  
}
```

执行上面程序，输出结果为：

```
1!=1
2!=2
3!=6
4!=24
```

上面程序中 `f(int i)` 函数被调用了 4 次，分别计算 1 到 4 的阶乘，而且 `f(int i)` 函数内没有循环，只是说明了一个静态局部变量来保存每次调用后的计算结果。

## (2) 静态全局变量

静态全局变量区别于非静态全局变量，静态全局变量仅限于本文件内使用，不能被其它文件使用，而非静态全局变量能被其它文件使用。当我们说“全局变量”时，一般是指非静态全局变量。

如果一个程序仅由一个文件组成，在说明全局变量时，有无 `static` 修饰没有什么区别。

如果一个程序由多文件构成，那么在一个文件中就能说明一些静态全局变量仅供本文件中的多个函数或类使用，其它文件中可说明同名的静态全局变量，而不会导致重名。因此合理使用静态全局变量，有助于削弱全局变量过多而导致的命名冲突问题。

例 5-14 静态全局变量示例。一个程序由 3 个源文件构成：`main.cpp`、`c1.cpp` 和 `c2.cpp`。结构如图 5.7 所示。

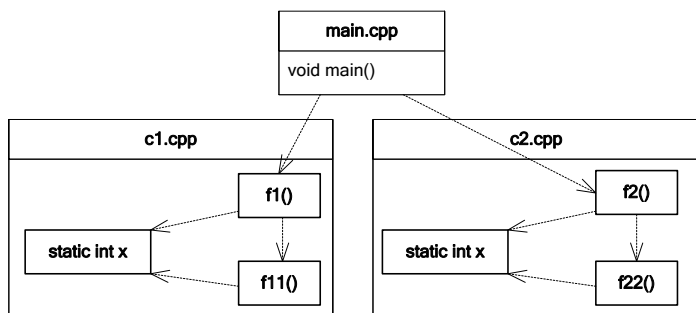


图 5-7 静态全局变量示例

这个程序包含了 3 个文件。文件 `main.cpp` 中只有 `main()` 函数，其中调用了另外两个文件中的 `f1()` 和 `f2()` 函数。文件 `c1.cpp` 和 `c2.cpp` 文件中分别定义了一个静态全局变量 `x`，分别供同一文件中的 2 个函数使用。编程如下：

```
//main.cpp 文件
#include <iostream>
void f1();    //函数原型说明
void f2();    //函数原型说明
int main(){
    f1();      //函数调用
    f2();      //函数调用
    system("pause");
    return 0;
}
//c1.cpp 文件
#include <iostream>
```

```
using namespace std;
static int x = 11;
void f11() {
    cout<<"c1.cpp: x="<<x<<endl;
}
void f1() {
    x += 2;
    f11();
}
//c2.cpp 文件
#include <iostream>
using namespace std;
static int x = 22;
void f22() {
    cout<<"c2.cpp: x="<<x<<endl;
}
void f2() {
    x += 2;
    f22();
}
```

在 VS2015 环境中，一个项目如果需要多个源文件，就在该项目的“源文件”中右键“添加”“新建项”或“已有项”。可一次性添加多个已有项。

执行该程序，输出结果为：

```
c1.cpp: x=13
c2.cpp: x=24
```

可以看到，不同文件中可定义同名的静态全局变量。此时 `main()` 函数不能访问定义在其他文件中的静态全局变量 `x`。此时如果将 `c1.cpp` 或者 `c2.cpp` 中的 `static` 修饰去掉一个，将会是什么结果？仍然正确。这是因为一个变量 `x` 为静态，只能在本文件中访问，而另一个变量 `x` 为全局变量，但被静态变量屏蔽，并没有发生命名冲突。但如果将两个 `static` 都去掉，就会发生重复命名，将给出连接错误。

#### 5.12.4 extern 变量

关键字 `extern` 修饰的全局变量称为外部变量。全局变量缺省为外部变量，所以外部变量的定义性说明可省略 `extern`。一个外部变量定义在一个文件中，可以被其它文件使用，只要先做一个引用性说明。外部变量是 C++ 程序中最大范围的变量。在运行时刻所有外部变量都存储在全局存储区，外部变量的生存期就是整个程序的运行期。

(1) 定义性说明：说明全局变量并分配内存、初始化。一个外部变量只能做一次定义性说明。其格式为：

```
[extern] <类型名> <变量名>[ = <初始值>;]
```

注意，如果使用 `extern` 修饰就一定要赋予初始值。例如：

```
extern int global = 100;
```

如果不用 `extern` 修饰，初始值可缺省取 0，也能赋予初始值。建议对每个全局变量都做显式初始化。

前面介绍的全局变量都是外部变量。显然，在一个多文件项目中不能出现同名的多个外部变量。

(2) 引用性说明：对于一个在其它文件中已定义的全局变量，在当前文件中说明开始引用，就可使下面代码能使用该变量。一个外部变量可做多次引用性说明。

外部变量的引用性说明只有一种形式，例如：

```
extern int global;           //不能指定初始值
```

引用性说明不能指定初始值。它告诉编译器下面代码要使用一个外部变量，而这个外部变量可能定义在后面，也可能在其它文件中。

在以下两种情况会用到外部变量的引用性说明：一是在同一文件中，全局变量使用在前，定义在后。二是在多文件程序中，一个文件 **fa** 要使用在另一个文件 **fb** 中定义的全局变量 **x**，在文件 **fa** 中要对 **x** 做外部说明。

例 5-15 外部变量使用示例。该程序由 2 个文件构成，文件 **main.cpp** 中定义了一个全局变量 **x** 供其中 **main()** 函数和另一个文件 **f1.cpp** 中的函数 **f1()** 使用。文件结构如图 5.8 所示。

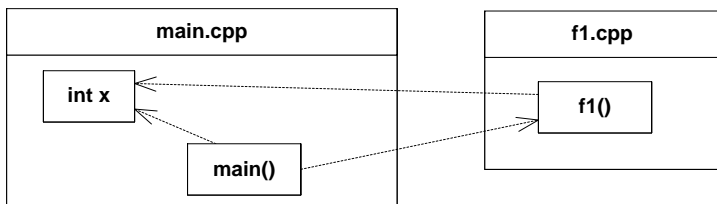


图 5-8 外部变量使用示例

编程如下：

```
//main.cpp
#include <iostream>
using namespace std;
void f1();           //函数原型说明，extern 缺省
int x = 11;          //定义全局变量 x
int main(){
    cout<<"main.cpp: x="<<x<<endl;
    f1();
    cout<<"after call f1: x="<<x<<endl;
    system("pause");
    return 0;
}

//f1.cpp
#include <iostream>
using namespace std;
extern int x;         //外部变量 x 的引用性说明
void f1(){
    x++;              //使用外部变量 x
    cout<<"f1.cpp:x="<<x<<endl;
}
```

上面程序的项目需要多文件配置(详见上一节)。执行程序后，输出结果为：

```
main.cpp: x=11
```

```
f1.cpp:x=12  
after call f1: x=12
```

可以看到, 每个函数原型说明都隐含着外部 `extern` 引用。这是因为每个函数(不包括类中的成员函数)都是全局性的, 都可以被其它文件中的函数调用, 只是调用前需要一个原型说明。

### 5.12.5 存储类小结

一般来说, 一个程序由多个源文件组成。如果多个文件之间需要共享某些数据, 就需要定义非静态全局变量。

一个文件中可包含一组函数或类。如果同一文件中的多个函数或类之间需要共享某些数据, 就要定义静态全局变量。

无论是非静态全局变量还是静态全局变量, 都会造成“数据耦合”现象, 即多个函数或类依赖于外部定义的某个数据。数据耦合违背了结构化设计的“低耦合”原则, 在复杂程序设计中会导致可读性、可维护性降低。例如, 全局变量难以修改, 难以移动, 导致依赖它的模块也难以修改和移动。使用全局变量与 `goto` 语句一样有害, 应尽可能避免定义使用全局变量。消除全局变量并非难事, 经过更细致的分解, 在函数设计时添加合适的形参和返回值就能解决。

结构化设计提倡“高内聚”设计原则, 即有关联的或者被依赖的数据和代码都封装在一起。这样的程序可理解性强、易于维护。显然“高内聚”提倡使用局部变量。

如果一个函数执行中需要某些数据来存储每次执行的结果, 以供下次执行时使用, 就需要定义块作用域中的局部静态变量。在后面的类中, 在一个类的内部也往往需要定义静态数据成员作为类的成员(而不是对象的成员), 它们属于类作用域中的局部静态变量。局部静态变量放在静态存储区, 具有整个程序的生存期, 这与全局变量一样, 但它们的定义和使用都是局部的。使用静态局部变量是比较合理的。

非静态局部变量是使用最广泛的, 例如函数的形参、函数体内的非静态局部变量。

总之, 当我们要定义使用一个变量时, 应按照下面次序来优先选择:

1. 非静态局部变量: 函数调用或函数体内使用, 动态存储, 首选。
2. 静态局部变量: 函数或类内使用, 静态存储、全程生存期。
3. 静态全局变量: 单个文件内共享。应慎重使用。
4. 非静态全局变量: 多文件共享。应尽量避免。

如果真的要使用全局变量, 也应该定义为命名常量, 使其不能更改。即便可以更改, 也只允许一个函数来更改, 其它函数读取。如果一个全局变量能被多个函数更改, 程序的理解和维护就麻烦了。

## 5.13 编译预处理

所谓编译预处理(preprocessor)就是在编译 C/C++源文件之前所进行的前期处理工作, 简称为预处理。预处理是 C/C++语言的一个重要特色。预处理可以完成以下工作:

- 插入指定的文件, 例如 `#include` 指令。
- 定义宏(macro)、宏扩展, 可进行字符串替换, 以及消除宏定义。

- 条件编译, 按特定条件(宏定义或常量表达式)来选择编译指定的程序段。
- 其它功能, 如编译错误信息提示等。

预处理需要使用一组指令, 都是以“#”开头。每条预处理指令单独占一行, 末尾没有分号。预处理指令可出现在程序中的任何位置, 但常位于开始位置。通过预处理指令先将源文件处理为一个临时文件, 其中只有 C/C++ 指令, 然后再对临时文件启动编译。

### 5.13.1 包含文件

包含文件就是将另一个源程序文件的全部内容插入到当前位置, 用 `include` 指令实现。该指令有两种格式:

```
#include <文件名>
```

或

```
#include "文件名"
```

这两种格式在查找文件路径上有区别。

`#include <文件名>`: 从系统目录中查找文件, 如果找不到, 预处理器就报错。主要用于包含编译系统预定义的头文件。所谓系统目录是指存放 C++ 系统文件的目录。所以这种格式常用于包含系统提供的头文件, 如 `iostream`、`math.h` 等。例如, 如果你的 VC 安装在 `C:\Program Files` 目录中, 那么系统提供的头文件就位于 `C:\Program Files\Microsoft Visual Studio 14.0\VC\include` 目录中。

`#include "文件名"`: 从用户当前工作目录中开始查找文件。若未找到, 再到系统目录中查找。若都未找到, 则预处理器报错。用户当前工作目录是指项目(project)所在的目录。这种格式常用于包含用户自己编写的文件。双引号括起来的文件名可以是文件的全路径名, 但目录名与目录名之间, 或目录名与文件名之间必须用两个“\”或一个“/”隔开。例如: `#include "c:\my\my.h"`。

对包含文件指令的几点说明。

(1)被包含的文件一般是“.h”(head 的缩写)后缀文件, 称为头文件。头文件中一般是一组函数或类的原型说明。当然也可包含其它文件, 如“.cpp”文件。

(2)`#include` 指令可出现在程序中的任何位置, 但通常放在程序的开头。

(3)一条 `include` 指令只能指定一个被包含文件。

(4)在一个被包含的文件中, 也可用 `#include` 指令来包含另一个文件。即文件可以嵌套包含。例如:

`file1.cpp` 中有命令:

```
#include "file2.h"
```

`file2.h` 中有命令:

```
#include "file3.h"
```

(5)应避免包含关系中出现“环路”。上面例子中, 如果 `file3.h` 文件中包含了 `file1.cpp` 或 `file2.cpp` 或自己, 都将形成环路, 导致预处理失败。一般情况下会给出警告或错误提示。

(6)小心同一个文件被插入多次而导致重复定义。如果一个项目中的两个文件都包含了同

一个文件，就可能出现这种情形。例如，一个程序有两个文件 `file1.cpp` 和 `file2.cpp`，它们各自都有下面包含命令：

```
#include "common.h"
```

此时 `common.h` 文件中如果没有条件编译，将导致该文件中说明的全局变量、函数或类重复定义而致使编译失败。后面将介绍如何用条件编译来解决此问题。

### 5.13.2 无参宏

一个宏(macro)是一个有命名的字符串。一个宏在定义之后可用于宏调用、宏展开，也能用于控制条件编译。

无参宏就是没有形参的宏。定义无参宏的格式为：

```
#define <标识符> [<字符串>]
```

其中，<标识符>为宏的名字。<字符串>就是宏的值。宏的值不用双引号，如果用双引号，双引号就是值的一部分。定义一个宏可以没有值，只用于控制条件编译。

例如：

```
#define PI 3.1415926
#define NULL 0
#define DEBUG
```

定义了一个宏，名为 `PI`，其值为 `3.1415926`。在此行之后的代码中就可以用 `PI` 来代替 `3.1415926`，这种使用宏的行为被称为“宏调用”。在编译预处理时，将用 `3.1415926` 替代每一个出现的 `PI`(字符串除外)。这种替换过程称为“宏展开”。又如：

```
#define PROMPT "面积为:"
```

定义了一个宏，名为 `PROMPT`，值为一个字符串“面积为:”。在编译预处理时，将用“面积为:”来代替 `PROMPT`。

对于一个已定义的宏，也可以解除宏定义，以终止其作用域。解除一个宏的定义，可用 `#undef <宏名>` 指令。例如：

```
#define PI 3.1415926 //A
...
#undef PI //B
...
```

在 `A` 行与 `B` 行之间的代码段是宏 `PI` 的作用域。如果没有 `#undef` 指令，宏的作用域将延伸到文件末尾。

例 5-16 用无参宏求圆面积。

```
#include <iostream>
using namespace std;
#define PI 3.1415926
#define R 2.8 //A
#define AREA PI*R*R
#define PROMPT "面积为:"
#define CHAR '!'

int main(void){
```



```

    cout << PROMPT<<AREA<<CHAR<<'\\n';
    system("pause");
    return 0;

}

```

执行程序后，输出结果为：

```
面积为:24.6301!
```

关于宏定义、宏调用和宏展开，注意以下几点。

(1) 宏名应符合标识符的语法规则。通常宏名用大写字母，以区别程序中的变量名或函数名。

(2) 宏定义可出现在程序中的任何一行的开始位置，但通常将宏定义放在源程序文件的开头部分。

(3) 一个宏的作用域是从宏定义开始到本文件结束，或用`#undef`指令终止宏。如果在此区间有包含文件`#include`指令，那么这个宏的作用域就延伸到被包含的整个文件。同样，被包含文件中定义的宏在包含文件中也起作用。即包含`#include`指令先起作用，然后再确定宏的作用域。

(4) 一个宏定义中可以使用已定义的宏。如上例 A 行定义宏 AREA 时，用到已定义的宏 PI 和 R。在编译预处理时，先对 PI 和 R 作宏展开如下：

```
#define AREA 3.1415926*2.8*2.8
```

宏展开后，实际计算的内容为：

```

void main(void)
{   cout << "面积为:"<< 3.1415926*2.8*2.8<<'!'<<'\\n'; }

```

(5) 宏展开时，不作任何计算，也不作语法检查，仅对宏名作简单的字符串替换。例如：

```

#define A 3+3
#define B A*A
int main(void)
{   cout<<B<<'\\n'; return 0;}           //A

```

执行程序，输出结果为 15，而不是  $6*6=36$ 。编译预处理将 A 行处理为：

```
{   cout<<3+3*3+3<<'\\n'; }
```

预处理指令无需分号结尾，但有分号也不会报错，例如：

```

#define PI 3.1415;                               //A
int main(void){
    double r, area;
    cout<<"输入半径:";
    cin>>r;
    area = PI*r*r;                                //B
    cout<<"面积为:"<<area<<'\\n';
    return 0;
}

```

A 行没有错，而 B 行语法错，这是因为经宏展开后为：

```
area=3.1415;*r*r;
```

显然, “area=3.1415;” 是一条语句, 而后面的 “\*r\*r;” 不符合语法规则。

(6) 对于一个已定义的宏, 应适当限制其作用域, 让它在合理范围内起作用, 以避免宏被误调用, 也防止引入不需要的宏。对于一个宏, 如果在其作用域中出现了包含指令#include, 该宏的作用域将延伸到被包含文件中, 而如果被包含文件中恰好用此标识符作其它用途, 就会发生被误调用的问题。如果定义宏的文件被其它文件所包含, 那么就可能引入了不需要的宏而导致重定义的错误。所以限制一个宏的作用域很重要。程序员往往会忘记用#undef 及时消除宏定义。

(7) 如果宏名出现在字符串中, 将不进行宏展开。

```
#define A "We"
#define B "A are students."           //字符串中出现了宏名 A
int main(void){
    cout<<"A "<<'\\t';                //字符串中的 A 不进行宏替换
    cout<<B<<'\\n';
    return 0;
}
```

执行程序后, 输出结果为:

```
A      A are students.
```

(8) 在同一个作用域内, 同一个宏名不允许重复定义, 以避免宏调用和宏展开的两义性。

如果一个无参宏带有一个值, 看起来就像一个命名常量, 例如上面的PI。但无参宏与命名常量之间具有本质差别。宏是无类型的, 只是字符串替换, 而且值是不计算的。而命名常量是有类型约束的, 它的值是可以计算的。在定义全局常量时, 经常用无参宏。例如:

```
#define NULL 0
```

无参宏有什么实际用途? 无参宏一般用于常量定义、控制条件编译。后面将介绍条件编译。

一个C/C++系统往往提供一些预定义宏(predefined macros), 一般用1个或2个下划线开头。例如宏“\_\_FILE\_\_”的值就是当前文件名, 宏“\_\_LINE\_\_”的值是当前行的行号。预定义宏可直接调用。一部分是ANSI标准兼容的, 另一部分是本地方言, 而且方言远多于标准。

例 5-17 预定义宏示例。

```
#include <iostream>
using namespace std;
int main(){
    cout<< __DATE__ <<endl;           //当前日期
    cout<< __TIME__ <<endl;           //当前时间
    cout<< __TIMESTAMP__ <<endl;       //当前文件的最后修改日期及时间
    cout<< __FILE__ <<endl;           //当前文件名
    cout<< __LINE__ <<endl;           //当前行号
    cout<< __WIN32 <<endl;             //当前平台是否为 Win32, 1 表示是
    system("pause");
    return 0;
}
```

执行程序, 输出结果如下:

```
Aug  2 2016
09:59:51
```

```
Fri Jul 8 18:05:19 2016
d:\vcstd\ch05\ex0517.cpp
8
1
```

关于预定义宏的细节，请参看相关文档。

### 5.13.3 有参宏

有参宏就是带若干形参的宏。有参宏在宏展开时，先用实参替换字符串中出现的对应的形参，再进行宏替换。定义有参宏的一般格式为：

```
#define <宏名>(<形参表>) <含形参的字符串>
```

当形参表中有多个形参时，相互之间用逗号隔开。每个形参仅给出名称。例如：

```
#define V(a, b, c)  a * b * c                //A
...
volumn = V(3.5, 8.3, 1.6);                  //B
```

A 行定义了求长方体体积的宏 V，三个形参 a、b、c 分别表示长方体的长、宽、高。调用有参宏称为宏调用，在宏调用中给出实参。编译预处理在对宏调用进行替换时，先用实参替代宏定义中对应的形参，并将替代后的字符串替代宏调用。如 B 行经宏替换后为：

```
volumn = 3.5 * 8.3 * 1.6;
```

注意实参替换形参时，对实参不作任何计算，只是简单的字符串替换。

对有参宏说明以下几点。

(1) 在定义有参宏时，宏名与左圆括号之间不能有空格。否则，编译预处理将其作为无参宏的定义，而不是有参宏。例如：

```
#define AREA (a,b) (a)*(b)
```

则编译预处理认为是将无参宏 AREA 定义为“(a,b) (a)\*(b)”，而不将“(a,b)”作为形参。

(2) 宏调用中给出的实参如果是一个表达式，最好把每个实参都用圆括号括起来，以免出错。例如：

```
#define AREA(a,b)  a*b                //A
...
c= AREA(2+3, 2+4);                    //B
```

B 行经宏展开后，c 的值为 2+3\*2+4=12，而不是你预料的 5\*6=30。

如果将 B 行改为：

```
c= AREA((2+3), (2+4));
```

或者，将 A 行的宏定义改为：

```
#define AREA(a,b)  (a)*(b)
```

都能得到你想要的结果。下面定义 max 有参宏，计算两个值中的最大值：

```
#define max(a,b)  (((a) > (b)) ? (a) : (b))
```

(3) 一个宏定义通常在一行内定义。当一个宏定义多于一行时可用“\”续行。例如：

```
#define AREA(a, b) (a)*\
(b)
```

行尾的转义符“\”表示要跳过其后的换行符。该转义符“\”的作用相当于续行符。

## 1. 有参宏与有参函数的区别

有参宏有什么实际用途？有参宏希望用来取代功能简单、代码短小、运行时间极短、调用较频繁的程序代码。有参宏与有参函数看起来相似，但有本质上的不同。

(1) 定义形式不同。宏定义只给出形参名，而不指明类型；而函数定义必须指定每个形参的类型。

(2) 调用处理不同。宏由编译预处理程序处理，而函数由编译器处理。宏调用仅作字符串替换，不做任何计算；而函数调用要先依次求出各个实参的值，然后才执行函数调用。

(3) 函数调用要求实参类型必须与对应的形参类型一致，即做类型检查；而宏调用没有类型检查。例如上面 `max` 在调用时，2 个实参可以是除 `bool` 之外的任意类型，就可避免编写多个重载函数。

(4) 函数可用 `return` 语句返回一个值，而宏不返回值。

(5) 多次调用同一个有参宏，经宏展开后，要增加代码长度；而对同一个函数的多次调用，不会明显增加代码长度。有参宏有点像内联 `inline` 函数。

## 2. C99 可变宏参数

C99 标准提出可变宏参数。可变宏参数与函数的可变形参类似。看下面例子：

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

#define CHECK1(x, ...) if (!(x)) { printf(__VA_ARGS__); } //A
#define CHECK2(x, ...) if ((x)) { printf(__VA_ARGS__); } //B
#define CHECK3(...) { printf(__VA_ARGS__); } //C

int main() {
    CHECK1(0, "here %s %s %s", "are", "some", "varargs1(1)\n"); //D
    CHECK1(1, "here %s %s %s", "are", "some", "varargs1(2)\n"); //E
    CHECK2(0, "here %s %s %s", "are", "some", "varargs2(3)\n"); //F
    CHECK2(1, "here %s %s %s", "are", "some", "varargs2(4)\n"); //G
    // always invokes printf in the macro
    CHECK3("here %s %s %s", "are", "some", "varargs3(5)\n");
    system("pause");
    return 0;
}
```

上面 A、B、C 行所定义的可参宏的最后参数是省略符(3 个小数点)，表示可变宏参数。宏体中用 `__VA_ARGS__` 来对应宏调用时的 1 个或多个实参，也包括实参之间的逗号。

执行上面程序，显示如下：

```
here are some varargs1(1)
here are some varargs2(4)
here are some varargs3(5)
```

对于可变宏参数，VS2015 允许宏调用是 0 个实参，而 DevC++(GCC)则要求至少 1 个实参。

### 3. 断言 assert

断言 `assert` 是 ANSI C 语言标准中的函数, 在运行时检查特定条件是否满足。大多 C/C++ 语言系统都采用有参宏的形式来实现断言。这种断言相对于静态断言 `static_assert`, 被称为动态断言。

要用断言就应包含 `<assert.h>` 或 `<cassert>`。断言语法形式如下:

`assert(expr 表达式);`

其中, `expr` 表达式的计算结果应该是一个整数或逻辑值, 如果非 0 则为真, 继续执行。如果为 0 则为假, 断言失败, 显示出错位置(包括文件名、行号及表达式)并终止程序。

断言是最简单的逻辑判断, 是进一步支持单元测试(unit test)的基础。假设你编写了一个函数 `int f(int a, int b)` 计算返回 `a` 和 `b` 的最小公倍数。你就需要编写输入输出语句来测试该函数是否正确。这样需要多次的手工输入, 人工观察判断结果是否正确。你还需要多个测试数据来验证。利用断言可建立一个简单测试函数, 包含一组断言来实现自动测试。

```
int f(int a, int b);
void testF(){
    assert(1==f(1,1));
    assert(2==f(1,2));
    assert(2==f(2,2));
    assert(6==f(2,3));
    assert(6==f(2,-3));
    //...
}
int main(){ testF();...}
```

启动测试函数, 如果断言都正确则不显示任何东西。如果任何一个断言失败, 就给出提示并终止程序, 这样你就能分析源代码中的缺陷, 修正后再启动, 直到全部断言正确。这种测试方法减少了很多重复键盘操作和人工判断的负担。

在 `assert` 断言表达式中注意不要引入副作用, 可以读取变量, 但不要改变变量。断言的执行与否应独立于被测试的程序逻辑。如果在包含 `<assert.h>` 语句之前定义宏 `NDEBUG`, 那么下面程序中的断言就不参与编译。

#### 5.13.4 条件编译

在编译源程序时, 往往要根据某个条件来选择编译某一段代码, 这就需要条件编译。条件编译有两种条件: 一种是判断某个宏名是否已定义, 另一种是计算某个常量表达式的值是否为真(非 0)。

##### 1. 宏名作为条件

根据一个宏名是否定义, 以及是否有 `#else` 分支, 有以下四种格式:

第一种格式为:

```
#ifdef <宏名>
```

```
<程序段>
#endif
```

如果指定的<宏名>已定义, 则编译<程序段>, 否则就不编译。  
程序中往往要根据是否调试执行来选择编译一段代码。例如:

```
#ifdef DEBUG
    cout<<"x="<<x<<endl;
    ...
#endif
```

如果前面定义了宏 `DEBUG`(确定了本次编译的目的是调试程序, 而不是构造最终可执行程序), 就编译这些输出语句, 否则就不编译。

根据系统提供的预定义宏, 可以知道执行环境的配置情况, 如 `CPU`、操作系统、开发系统版本等, 利用条件编译, 就可以编写适合多种平台执行的通用性程序。

第二种格式为:

```
#ifdef <宏名>
    <程序段 1>
#else
    <程序段 2>
#endif
```

如果<宏名>已定义, 就编译<程序段 1>, 否则就编译<程序段 2>。

第三种格式为:

```
#ifndef <宏名>
    <程序段>
#endif
```

如果未定义<宏名>, 就编译该<程序段>; 否则不要编译。

第四种格式为:

```
#ifndef <宏名>
    <程序段 1>
#else
    <程序段 2>
#endif
```

如果未定义<宏名>, 就编译<程序段 1>; 否则就编译<程序段 2>。

## 2. 常量表达式作为条件

在编译预处理过程中, 可以计算一个常量表达式作为编译条件, 有三种格式。

第一种格式为:

```
#if <常量表达式>
    <程序段>
#endif
```

如果<常量表达式>的值为真(非 0), 就编译<程序段>, 否则就不编译。<常量表达式>中只能包含 `int` 常量、`char` 常量和 `defined` 运算符, 可以形成关系表达式或逻辑表达式。

运算符 **defined** 的格式为:

`defined(<标识符>)` 或 `defined <标识符>`

如果<标识符>已定义为宏名, 运算符的值就为 1(真), 否则为 0(假)。这样,

```
#if defined(DEBUG)
```

就等价于:

```
#ifdef DEBUG
```

第二种格式为:

```
#if <常量表达式>
```

```
<程序段 1>
```

```
#else
```

```
<程序段 2>
```

```
#endif
```

如果<常量表达式>的值为真(非 0), 就编译<程序段 1>, 否则编译<程序段 2>。

一般来说, 宏展开时不进行计算, 但在条件编译指令的<常量表达式>中出现宏调用时, 对宏展开的整数值进行算术计算。

例 5-18 条件编译中的表达式计算示例。

```
#define HLEVEL 2
#define DLEVEL 4 + HLEVEL
#if DLEVEL > 5                                //A
    #define SIGNAL 1
    #if STACKUSE == 1
        #define STACK 200
    #else
        #define STACK 100
    #endif
#endif

#include <iostream>
using namespace std;
int main(){
    cout<< DLEVEL <<endl;                      //B
    cout<< DLEVEL * DLEVEL <<endl;              //C
    cout<< SIGNAL <<endl;                      //D
    cout<< STACK <<endl;                      //E
    system("pause");
    return 0;
}
```

执行程序, 输出结果如下:

```
6
14
1
100
```

宏 **DLEVEL** 的值经展开后为 “4+2”, 但在 A 行进行表达式计算时以 6 作为结果。显然这是算术计算的结果。在下面的 B 行和 C 行展开仍为 “4+2”。

上面例子中, 如果 A 行条件不满足, 就不会定义 **SIGNAL** 和 **STACK** 这两个宏, 就会导致 C++代码中 D 行和 E 行编译出错。实际上, 我们应尽量避免在 C++代码中出现宏调用。另一方面, A 行的条件编译有缺陷, 可以修改它, 使一个条件编译有统一的结果。不管条件是否满足, 都能使特定的宏得到定义, 只是具有不同的值。例如:

```
#if DLEVEL > 5
```

```

#define SIGNAL 1
#if STACKUSE == 1
    #define STACK 200
#else
    #define STACK 100
#endif
#else
    #define SIGNAL 0
    #if STACKUSE == 1
        #define STACK 100
    #else
        #define STACK 50
    #endif
#endif

```

第三种格式是一种嵌套方式，就好像嵌套的 if-else 语句：

```

#if <常量表达式 1>
    <程序段 1>
#elif <常量表达式 2>
    <程序段 2>
#elif <常量表达式 3>
    <程序段 3>
...
#else
    <程序段 n>
#endif

```

上面 **#elif** 指令可看作是 **#else** 和 **#if** 的结合体。例如：

```

#if defined(CREDIT)
    credit();
#elif defined(DEBIT)
    debit();
#else
    printererror();
#endif

```

上面例子中，两个宏 **CREDIT** 和 **DEBIT** 的定义是互斥的。如果定义了第一个宏，就编译 **credit()** 语句。如果定义了第二个宏，就编译 **debit()** 语句，否则就编译 **printererror()** 语句。

在条件编译中可以定义其它的宏。下面例子需要事前定义一个宏 **DLEVEL**，值为一个整数。

```

#if DLEVEL == 0
    #define STACK 0
#elif DLEVEL == 1
    #define STACK 100
#elif DLEVEL > 5
    display( debugptr );           //函数调用语句
#else
    #define STACK 200
#endif

```

上面例子中根据宏 **DLEVEL** 的不同的值，再定义其它的宏，如 **SIGNAL** 和 **STACK**，也能控制编译 C++ 指令。



## 5.13.5 条件编译示例

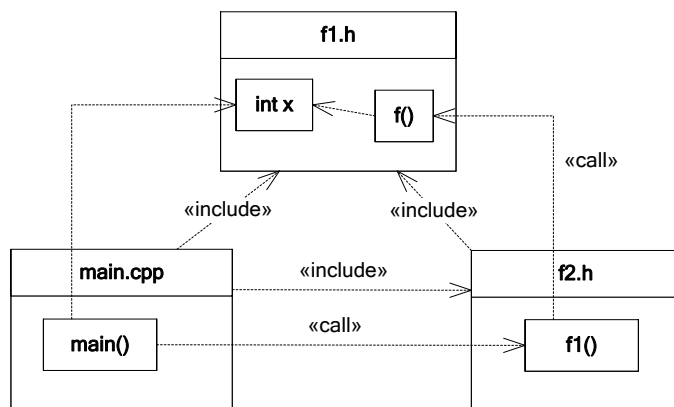


图 5-9 条件编译示例

条件编译的一种常见用法是对同一个头文件避免多次包含。

例 5-19 条件编译示例。如图 5.9 所示，在 f1.h 文件中说明了一个全局变量 x 和一个函数 f()。main.cpp 文件中 main() 函数要调用 f2.h 文件中的 f1() 函数，就要包含了 f2.h 文件。main() 函数还要访问 f1.h 中的全局变量 x，也要包含 f1.h 文件。但因 f1() 函数也要访问全局变量 x，也包含了 f1.h 文件。这样 f1.h 文件就被 main 函数包含两次。编程如下：

```
//f1.h
int x = 11;
void f(){
    x += 2;
}
//f2.h
#include "f1.h"
void f1(){
    f();
}
//main.cpp
#include <iostream>
using namespace std;
#include "f1.h"
#include "f2.h"
int main(){
    cout<<"x="<<x<<endl;
    f1();
    cout<<"x="<<x<<endl;
    system("pause");
    return 0;
}
```

这个例子并不需要多文件项目配置，通过包含文件将 3 个文件组成一个程序。对文件 main.cpp 进行编译，会发生全局变量 x 重定义错误。这是因为 f1.h 文件被包含了两次。每个头文件都可能出现被多次包含的问题。利用编译预处理能解决此类问题。我们修改 f1.h 文件如下：

```
#ifndef F1_h
#define F1_h
```

```
int x = 11;
void f(){
    x += 2;
}

#endif
```

第一条语句与最后一条语句构成一个条件编译控制。如果未定义宏 `F1_h`，那么就是第一次插入。先定义此宏，然后再插入下面代码。当再次要插入该文件时，由于已定义了此宏，所以就不会再次插入下面代码，从而避免了重复定义。

这种模式具有通用性。对于每个头文件都可以如此处理，来避免可能的重定义问题。

### 5.13.6 其它预处理指令

除了上面介绍的预处理指令之外，还有下面一些指令。

#### (1) `#error <字符串>`

该指令用于产生编译错误信息，将<字符串>作为错误信息并停止编译。该指令经常用于检查一致性，以避免预处理过程中出现冲突。例如：

```
#if !defined(__cplusplus)
    #error C++ compiler required.
#endif
```

#### (2) `#line <正整数> "文件名"`

该指令更改内部存储的当前文件名和当前行，也就是更改预定义宏 `__FILE__` 和 `__LINE__` 的值。该指令经常用于一个程序生成器中用来指出错误信息，使编译错误信息能指出原始文件中发生错误的地方，而不是生成文件中出错的地方。

#### (3) `#pragma <指令>`

每一个 C/C++ 系统都支持特定的硬件或操作系统的一些特性，该指令就是为当前编译器提供一种控制编译的方式，以保持与 C/C++ 语言的兼容性。其中<指令>是每一种编译器独有的指令。当编译器发现一个<指令>不能识别时，只是给出警告而继续编译。例如：

```
#pragma message(字符串)
用来在编译时刻给出提示信息，指定的信息将出现在编译命令显示的地方。例如：
#if _M_IX86 == 500
    #pragma message( "Pentium processor build" )
#endif
```

#### (4) `#import "文件名" [属性表]` 或 `#import <文件名> [属性表]`

该指令从一个类型库文件(如 TLB、ODL、EXE、DLL 等后缀文件)中导入信息，转换为 C++ 类，并生成两个头文件，用 C++ 源码重新构建类型库的内容。

以上预处理指令都有较多细节，请参看 MSDN 等相关文档。

## 5.14 小 结

- 一个函数完成一项功能，是结构化编程的基本模块。一个结构化程序由多个函数构成。
- 一个函数作为一个封装体，具有自己的名称和形参，也能返回一个值作为计算结果。

- 所有的函数定义都是平行的，包括主函数 `main`。在一个函数的函数体内，不能再定义另一个函数，即不能嵌套定义。但函数体内可以调用其它函数，这样形成嵌套调用。习惯上把调用方称为主调函数。
- 一个函数调用就是确定一个函数名并提供相应的实参，然后开始执行该函数体中的语句。实参的个数与类型应该与函数的形参对应。当函数体执行结束时，函数调用就得到了一个值，返回给调用方。函数定义在前，函数调用在后。
- 函数的重载就是多个函数具有相同的名称，但有不同的形参(个数不同或者类型不同)。函数的名称及形参作为一个整体，被称为函数的基调(signature)。重载函数的调用是在编译时刻由编译器根据实参与形参的匹配来决定。这是 C++ 提供的一种静态的多态性。
- 函数调用允许嵌套。一个函数直接或间接地调用自己就是递归调用。用数学归纳法能表示的计算往往都能用递归函数实现，但执行效率不高。如果一个问题确实不容易用非递归方式解决，那么递归函数就是最佳方案，例如河内塔问题。
- 函数原型就是函数的名称、形参及返回值说明，没有函数体的实现过程。说明一个函数原型之后，就能调用该函数，而函数的实现可以放在后面或者其它文件中。
- 函数的形参可以带有缺省值，主要是为了简化函数调用。但必须是形参表中最后几个形参，而且注意与函数重载发生冲突。
- 可变参数是函数定义中说明形参个数不固定，在调用函数时可提供任意多个实参。要调用 `<stdarg.h>` 中的 3 个函数和 1 个类型来处理。
- 内联函数就是用 `inline` 修饰的函数，希望编译器能将函数体替代函数调用，以提高执行效率。
- 作用域就是一个标识符的有效区域。作用域共分为五类：块作用域、文件作用域、函数原型作用域、函数作用域和类作用域。定义在块作用域中的变量就是局部变量，而定义在文件作用域中的变量就是全局变量。
- 一个 C++ 程序在内存中具有特定的存储区域。代码区用来存储程序代码，数据区用来存储静态数据和全局数据，而堆栈区用来存储局部变量数据。只有堆栈区是动态伸缩的。
- 存储类确定了变量占用内存的期限、可见性以及存储区域。变量的存储类分为四种：自动(auto)、寄存器(register)、静态(static)、外部类型(extern)。每个变量只能选择其中之一。定义在块作用域中的静态变量就是静态局部变量，能保存函数每次执行后的结果，只能被当前函数访问。定义在文件作用域中的静态变量就是静态全局变量，能被本文件中的所有函数访问。
- 全局变量会造成“数据耦合”，即多个函数或类依赖于外部定义的某个数据，违背了结构化设计的“低耦合”原则，在复杂程序设计中会导致可读性、可维护性降低。所以应尽可能避免定义使用全局变量。
- 一个程序可以由多个文件组成。本章介绍了如何配置一个多文件的项目。
- 外部变量就是用 `extern` 修饰的变量，就是非静态的全局变量，能被一个项目中的所有文件访问。定义性说明就是说明一个新的全局变量，而引用性说明就是要引用另一个文件中定义的全局变量。

- 编译预处理就是在编译 C/C++源文件之前所进行的前期处理工作。包含文件就是将另一个文件的内容插入到当前位置,常包含头文件用于函数原型说明。
- 一个宏是一个命名的字符串。无参宏就是没有形参的宏,一个宏也可以没有值。无参宏的主要用途是控制条件编译。有参宏就是有形参的宏。
- 条件编译就是在编译源程序时,根据某个条件来选择编译某一段代码。条件可以是某个宏名是否已定义,或者某个常量表达式的值是否为真(非0)。条件编译的用途主要是控制编译执行的方式,如调试程序、通用性编程,也能解决包含文件时可能发生的多次插入的问题。

### 5.15 练 习 题

1. 对于C++的函数,下面哪一种说法是错误的?  
A 函数的定义不能嵌套,但函数的调用可以嵌套。  
B 函数的定义可以嵌套,但函数的调用不能嵌套。  
C 一个程序执行从 main 函数开始。  
D main 函数可调用其它函数,而其它函数一般不调用 main 函数。
2. 对于函数 void f(int x),下面哪一个调用是正确的?  
A int y = f(9);      B f(9);      C f(f(9));      D x = f();
3. 对于函数重载,下面哪一种说法是错误的?  
A 函数名不同,但形参的个数与类型相同      B 函数名相同,形参的个数或类型不同  
C 函数名相同,形参的个数和类型也相同      D 函数名相同,返回值不同,与形参无关
4. 下面哪一对函数满足函数重载规则?  
A float fun(float x); void fun(float y);  
B float funa(float x); void fun(float x, float y);  
C float fun(float y); void fun(float x);  
D float fun(float x, float y); void fun(float y);
5. 下列程序执行结果是\_\_\_\_\_。  

```
int f(int Int){  
    if (Int==0) return 1;  
    return (Int + f(Int-1));  
};  
  
int main(void){  
    int inT=9;  
    cout<< "result=" << f(inT) <<'\n';  
    return 0;  
}
```

  
A result=1      B result=37      C result=46      D 编译错
6. 下列哪一个递归函数能正确执行?

- A int f(int n) { if (n < 1) return 1; else return n\*f(n+1); }
- B int f(int n) { if (n > 1) return 1; else return n\*f(n-1); }
- C int f(int n) { if (abs(n) < 1) return 1; else return n\*f(n/2); }
- D int f(int n) { if (n > 1) return 1; else return n\*f(n\*2); }

7. 下面程序的运行结果是\_\_\_\_\_。

```
int f(int x){
    static int u=1;
    x += x;
    return u *= x;
}

int main(void)
{ int x=10; cout<<f(x)<< '\t'; cout<<f(x)<<endl; return 0; }
```

- A 10 20                      B 20 800                      C 20 400                      D 20 20

8. 下面程序的运行结果是\_\_\_\_\_。

```
int t(void)
{ static int i = 1; i += 2; return i; }

int t1(void)
{ int j =1; j += 2; return j; }

int main(void){
    int j=2;
    t(); cout << "I="<<t()<<" ";
    t1(); cout << "J="<<t1()<< '\n';
    return 0;
}
```

- A I=5 J=3                      B I=5 J=5                      C I=3 J=5                      D I=3 J=-3

9. 下面程序的运行结果是\_\_\_\_\_。

```
float p(float x, int i)
{ x = x + 2.5; i = i + x; return x; }

int main(void){
    int i = 10; float x = 3.25;
    x = p(x, i)-1; cout<<"x="<<x <<" ,i="<<i;
    return 0;
}
```

- A) x=4.75,i=10    B) x=5.75,i=12    C) x=5.75,i=10    D) x=4.75,i=12.5

10. 下面程序的运行结果是\_\_\_\_\_。

```
int loop( int n){
    if(n == 1) return 10;
    else
        if(n % 2 == 0) return loop(n - 1) + 2;
```

```

        else return loop(n-1) + 3;
    }

    int main(void)
    { cout<<loop(3)<<endl;    return 0;}

    A 14          B 15          C 16          D 10

```

11. 下面程序的编译运行结果是\_\_\_\_\_。

```

#define AA      10
#define D (x)  x * x                                //A
int main(void){
    int x = 1, y = 2, t;
    t = D(x+y) * AA;                                //B
    cout<<t;
    return 0;
}

```

A B 行中的表达式有错      B 50      C 30      D A 行中的宏定义有错

12. 设有以下宏定义和语句，则 i 的值为\_\_\_\_\_。

```

#define ONE      1
#define TWO      (ONE+ONE)
#define THREE    ONE+TWO
i=THREE*3+TWO*2;

```

A 13      B 11      C 9      D 8

13. 执行以下程序输出是\_\_\_\_\_。

```

#define P      5
#define R      2+P
int main(void)
{ float a1; a1 = P* R * R; cout << "a1="<<a1<<'\n'; return 0;}

A a1=75      B a1=245      C a1=49      D a1=25

```

13. 执行以下程序输出是\_\_\_\_\_。

```

static int c;
int main(void){
    #if c*3
        int i=10.88; cout<<"i="<<i<<"\n";
    #else
        int j=10000.99;cout<<"j="<<j<<"\n";
    #endif
    return 0;
}

```

A j=10000      B j=10.88      C 语法错，无输出      D j=10

14. 关于编译预处理命令，下面哪一个说法是正确的？

- A 以"#"开头, 必放在在程序开头      B 以#开头的行, 后面不可加分号  
C 以"#"开头, 可出现在一行中的任何位置      D 以"#"开头的行, 可出现程序中的任一位置

15. 读程序, 写出下面程序输出结果。

```
int c_multiple(int a, int b){
    int i;
    for (i = (a>b?a:b); i <= a*b; i++)
        if (i % a == 0 && i % b == 0)
            return i;
}

int main(void){
    cout <<c_multiple(2,5)<<"\n";
    cout <<c_multiple(6,8)<<"\n";
    return 0;
}
```

16. 读程序, 写出下面程序的输出结果。

```
void f(int n){
    if(n/10){ cout<<n%10<<'\\n'; f(n/10); }
    else    cout<<n;
}

int main(void)
{ f(579); cout<<endl; return 0; }
```

17. 读程序, 写出下面程序的输出结果。

```
void f(int n){
    cout<<n/10<<'\\n';
    if(n/10) f(n/10);
}

int main(void)
{ f(345); cout<<endl; return 0; }
```

18. 读程序, 写出下面程序的输出结果。

```
int x = 100;

int main(void){
    int x=10, k=20;{
        int x=20;
        k=::x;
        cout<<x<<'\\t'<<k<<endl;
    }
    cout<<x<<'\\t'<<k<<endl;
    return 0;
}
```

19. 读程序, 写出下面程序的输出结果。

```
void swap(int p1, int p2)
{   int p;  p = p1; p1 = p2; p2 = p; }

int main(void) {
    int x=20,y=40;
    cout<<"x="<<x<<"\n"<<"y="<<y<<"\n";
    swap(x, y);
    cout<<"x="<<x<<"\n"<<"y="<<y<<"\n";
    return 0;
}
```

20. 根据题目要求, 编写完整的程序, 可所用已有函数, 也可自行添加函数。

(1)编写一个函数 `int digitSum(int x)`, 返回形参 `x` 的十进制数的各位数之和, 例如 123 的各位数为  $1+2+3=6$ 。

(2)编写一个函数, 返回三个整数中的最大数并输出。要求先编写函数实现两个数中求最大数。

(3)编写一个函数, 计算组合数:  $C(m, r) = m! / (r! \times (m-r)!)$ , 其中  $m, r$  为正整数, 且  $m > r$ 。分别求出  $C(4, 2)$ ,  $C(6, 4)$ ,  $C(8, 7)$  的组合数。求阶乘和组合数分别用函数来实现。

(4)设计一个函数, 将输入的十进制整数转换为相应的十六进制数并输出, 要求输出 8 个十六进制数。

(5)设计一个程序, 求出 5~500 之间的所有素数, 要求每行输出 5 个素数。设计一个函数来判断一个整数是否为素数。

(6)设计一个程序, 对于输入的一个正整数  $x$ , 分解质因数并按从小到大的次序输出所有质因数。例如,  $12=2*2*3$ ,  $13=13$ ,  $14=2*7$ 。

(7)设计一个程序, 输入两个正整数, 求出这两个整数的最小公倍数。编写一个函数来求两个数的最小公倍数。

(8)设计一个程序, 输入两个正整数, 求出这两个整数的最大公约数。编写一个函数来求两个数的最大公约数。

(9)对于 Hanoi 塔问题, 如果要移动 64 个盘子, 假设一秒钟移动 1000 次, 大约需要多少年又多少月又多少天。

(10)编写一个函数, 用非递归方式求 Fibonacci 数列的前  $n$  项,  $n$  作为函数形参。

(11)当  $x > 1$  时, Hermite 多项式定义为:

$$H_n(x) = \begin{cases} 1 & n = 0 \\ 2x & n = 1 \\ 2xH_{n-1}(x) - 2(n-1)H_{n-2}(x) & n > 1 \end{cases}$$

当输入实数  $x$  和整数  $n$  后, 求出 Hermite 多项式的前  $n$  项的值。分别用递归函数和非递归函数实现。

(12)阿克曼函数定义为:

$$Acm(m, n) = \begin{cases} n+1 & m = 0 \\ Acm(m-1, 1) & n = 0 \\ Acm(m-1, Acm(m, n-1)) & n > 0, m > 0 \end{cases}$$

其中  $m, n$  为正整数。设计一个程序, 分别求出  $Acm(2, 3)$ ,  $Acm(2, 6)$  的值。要求用一个函数求  $Acm(m,$



n)的值。

(13)设计一个程序,把输入的整数实现逐位正序和反序输出。如输入一个整数 3456,则输出 3456 和 6543。分别设计两个函数,一个实现正序输出;另一个实现反序输出。

(14)设计一个程序,由 3 个文件组成,一个文件 `main.cpp` 包含 `main` 函数,第 2 个文件 `max.cpp` 包含一个函数 `max(int, int)`,第 3 个文件 `hex.cpp` 包含一个函数 `toHex(int x)` 将形参 `x` 显示为十六进制。建立各文件,然后建立一个多文件项目,输入任意两个整数,求出最大值并以十六进制形式显示。

如果不配置多文件项目,如何更改程序使之能构建运行?

(15)已知三角形的三条边长为 `a`、`b`、`c`,则三角形的面积为:

$$area = \sqrt{s(s-a)(s-b)(s-c)}$$

其中  $s=(a+b+c)/2$ 。设计一个函数求三角形的面积。

## 第6章 数组

单个变量只能存放一个数据值。程序往往要处理一组相同类型、彼此相关的一组数据，就需要一种特殊的数据结构来处理，这就是数组。数组(array)是一种派生类型。一个数组能同时存放多个数据值，并能对每个数据进行访问。本章将介绍一维数组、二维数组和字符数组(包括字符串)的定义及使用。

### 6.1 一维数组

一个数组(array)是由相同类型的一组变量组成的一个有序集合。数组中的每个变量称为一个元素(element)，所有元素共用一个变量名，就是数组的名字。数组中的每个元素都有一个序号，称为下标(index)。访问一个元素就可以用数组名加下标来实现。数组必须先定义后使用。

#### 6.1.1 一维数组的定义

一维数组就是具有一个下标的数组。定义一个数组有 3 个要素：类型、名称与大小。语法格式为：

<数据类型> <数组名> [<常量表达式>]

其中，<数据类型>确定了该数组的元素的类型，可以是一种基本数据类型，也可以是已定义的某种数据类型。<数组名>是一个标识符，作为数组变量的名字。方括号中的<常量表达式>必须是一个正整型数据，其值为元素的个数，即数组的大小或长度。注意这里的方括号[]表示数组，而不是表示可缺省内容。例如，下面定义了三个不同类型的数组：

```
int a[5];           //定义了一个 int 数组 a
float b[20];        //定义了一个 float 数组 b
double c[5];        //定义了一个 double 数组 c
```

对于上面数组 a，元素类型为 int，a 是数组名，方括号中的 5 表示数组的长度，即该数组包含了 5 个元素，分别是 a[0]、a[1]、a[2]、a[3]、a[4]。

如果一个数组有 n 个元素，那么数组中元素的下标从 0 开始到 n-1。

具有相同类型的数组可以在一条说明语句中定义。例如：

```
int a1[5], a2[4];    //同时定义两个整型数组
```

具有相同类型的单个变量和数组也可以在一条语句中定义。例如：

```
int x, y[20];        //同时定义整型变量和整型数组
```

系统为一个数组分配一块连续的存储空间。该空间的字节大小为  $n \times \text{sizeof}(\text{元素类型})$ ，其中 n 为数组的长度。

一个数组的大小必须在定义时确定，可以用一个正整数常量，也可以用命名常量构成的

一个表达式。例如：

```
const int m = 10;
int a[m + 10];          //相当于 int a[20]
```

一个数组的大小不允许用变量来指定。注意，VC 系列到 VS2015 都不支持可变长数组，尽管它是 C99 标准之一。DevC++(GCC)支持可变长数组。

一个数组定义之后，其下标的有效范围就确定了。在定义一个数组时，各下标可以用来表示不同的语义。例如定义一个数组：float score[8]来表示某学生在某学期的 8 门课程的成绩，那么 score[0]就可以表示高等数学成绩，score[1]可表示英语成绩，score[2]表示 C++程序设计课程成绩，等等。

### 6.1.2 一维数组的初始化

在定义一个数组的同时可以给各元素赋予初值。语法格式为：

<类型> <数组名>[<常量表达式>] {<初值表>};

其中<初值表>是用逗号隔开的一组元素值。值的类型必须与数组元素的类型相兼容。

例如，下面都是合法的初始化。

```
int a[10]={1,2,3,4,5,6,7,8,9,10};          //初始化数组的所有元素
float x[5]={2.1,2.2,2.3,2.4,2.5};          //初始化数组的全部元素
int b[10]={1,3,5,7,9};                      //初始化前 5 个元素，其余元素值为 0
int c[]={2,4,6,8,10};                      //由给定的元素个数确定数组 c 的大小
```

花括号之前的等号可省略，例如：

```
int a[10]{ 1,2,3,4,5,6,7,8,9,10 };
```

表 6.1 列出了上述 4 个数组初始化后各元素对应的数据值。

表 6-1 数组初始化后各元素的值

数组 a	元素值	数组 x	元素值	数组 b	元素值	数组 c	元素值
a[0]	1	x[0]	2.1	b[0]	1	c[0]	2
a[1]	2	x[1]	2.2	b[1]	3	c[1]	4
a[2]	3	x[2]	2.3	b[2]	5	c[2]	6
a[3]	4	x[3]	2.4	b[3]	7	c[3]	8
a[4]	5	x[4]	2.5	b[4]	9	c[4]	10
a[5]	6			b[5]	0		
a[6]	7			b[6]	0		
a[7]	8			b[7]	0		
a[8]	9			b[8]	0		
a[9]	10			b[9]	0		

对数组元素的初始化，说明以下几点。

(1) 初始化时，可以对全部元素赋初值，也可以对部分元素赋初值。若对数组的部分元素赋初值时，未赋值的元素值缺省为 0。

(2) 若对所有元素赋初值，可以不指定数组的长度，编译器会根据初值表中数据的个数自动确定数组的长度。

(3) 在定义数组时, 编译器必须知道数组的大小。因此只有在初始化的数组定义中才能省略大小。

(4) VC 系列到 VS2015 都不支持 C99 标准的指定初始化 `designated initializer`, 可对指定下标元素初始化。

### 6.1.3 一维数组的使用

在定义一个数组之后, 就可以使用该数组。主要用法是通过下标访问各元素。数组中的每个元素由唯一下标来确定, 通过数组名及下标就可以唯一确定数组中的一个元素。下标可以是表达式, 其值必须是整数。例如下面代码:

```
int a[5], b[2], i, j;
a[0] = b[0] = 2;           //下标为常量
i = 1; j = 3;
a[i] = j;                  //下标为变量,
a[j+1] = 8;                //下标为表达式
a[j] = 3 * a[1];
a[b[0]] = a[i] + a[0];     //下标是数组元素的值
b[1] = a[2];
int h[5];
for(i = 0; i < 5; i++)
    h[i] = i * i;          //利用循环为数组元素赋值
```

表 6.2 给出了执行以上程序段后, 三个数组中各元素的值。

表 6-2 三个数组元素的值

数组元素	数据值	数组元素	数据值	数组元素	数据值
a[0]	2	b[0]	2	h[0]	0
a[1]	3	b[1]	5	h[1]	1
a[2]	5			h[2]	4
a[3]	9			h[3]	9
a[4]	8			h[4]	16

关于数组的使用, 说明以下几点:

(1) 通过下标访问数组元素时, 注意下标越界, 尤其是使用表达式计算下标。下标越界访问不会有错误提示, 只是访问邻近的内存单元。如果仅读取元素, 结果无法预料。如果修改了越界元素, 就可能导致严重错误。

(2) 两个数组之间不能直接赋值, 即使类型和长度都相同, 也不能直接赋值。例如:

```
int x[5], y[5] = {2, 4, 6, 10, 100};
x = y;           //语法错误, 赋值号左边不是左值
```

要将数组 `y` 中的各元素值依次拷贝到数组 `x` 中, 可用循环语句来实现。例如:

```
for(int k = 0; k < 5; k++)
    x[k] = y[k];
```

(3) 两个数组名之间不宜使用关系运算符。即使两者大小内容都相同, 用关系运算符来判等也不同。例如:

```
if (x == y)
```

结果恒为假。这是因为这个表达式在判断两个数组的存储地址是否一样。数组名的本质是数组元素的首地址, 详见第 8 章。

### 6.1.4 基于范围的 for 语句

前面所使用的 for 语句是传统 C 语言形式。C11 提供一种新式 for 语句，称为基于范围 (range-based) 的 for 语句，对一个数组或 STL 容器(如 vector)中每个元素从头到尾遍历一次。

基于范围的 for 语句的一种简单形式如下：

**for(<类型> <变量名> : <数组名/容器名>){ <变量名>可读 }**

假设有 `int a[]={1,2,3};`

```
for(int e : a){cout<<e<<" ";
```

将数组 `a` 中每个元素依次传给变量 `e`，然后在后面循环体中处理该变量 `e`。

下面是一个简单例子：

```
#include <iostream>
using namespace std;
int main() {
    int x[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    for( int y : x ) {           //A
        cout << y << " ";       //B
    }
    cout << endl;
    for( auto y : x ) {          //C
        cout << y << " ";
    }
    cout << endl;
    system("pause");
    return 0;
}
```

上面 A 行中，变量 `y` 作为数组元素的一个拷贝，在 for 循环体中可读，如 B 行所示。循环体中改变 `y` 并不能改变数组元素。C 行采用 `auto` 关键字自动确定变量 `y` 的类型，简化类型说明，推荐使用。

如果要在循环体中改变元素，将 `y` 类型改为引用类型(引用在第 8 章介绍)。例如：

```
for( int &y : x ) {              //A
    cout << y++ << " ";         //B
}
```

上面 A 行将 `y` 类型改为 `int &`，即 `int` 引用类型，这样 `y` 就作为元素的引用。下面 B 行对 `y` 的改变就作用于数组元素之上。执行修改后的程序，显示如下：

```
1 2 3 4 5 6 7 8 9 10
2 3 4 5 6 7 8 9 10 11
```

for 循环体中可用 `break` 停止循环，也可用 `continue` 停止本元素的循环，开始下一个元素的循环。

基于范围 for 循环只能在数组的作用域范围内起作用。如果将一个数组通过形参传递给函数，该函数内就不能用这种 for 循环，这是因为 for 不能自动计算循环的终点。此时可调用 `<algorithm>` 中的 `for_each` 函数来替代，后面介绍。

### 6.1.5 一维数组的应用

利用数组可存储大量数据，也能用循环方式来访问，就能实现多种问题的求解，下面通过几个实例来说明一维数组的应用。

## (1) 多项式计算

例 6-1 有函数  $y = 5x^5 - 3.2x^3 + 2x^2 + 6.2x - 8$ , 任意输入一个  $x$  值, 求  $y$  的值。要求  $x$  和  $y$  为 `float` 型。

这个函数是一个一元  $n$  次多项式。一般结构为  $y = a_n x^n + a_{n-1} x^{n-1} + \dots + a_2 x^2 + a_1 x + a_0$ , 如果按多项式直接计算, 会有多次重复计算。例如, 计算  $x^5 = x * x * x * x * x$ , 就重复计算了  $x^4 = x * x * x * x$ 。浮点数的乘法计算开销很大, 我们应寻求更效率的解决办法。

对于  $n$  次多项式的计算可分解为  $n$  个一次式来计算, 可避免重复计算。例如, 可将多项式  $5x^5 - 3.2x^3 + 2x^2 + 6.2x - 8$  分解为  $(((((5x+0)x-3.2)x+2)x+6.2)x-8)$ 。虽然可用一个算术表达式直接实现函数  $y$ , 但我们希望程序能方便处理任意一个一元  $n$  次多项式。

一种办法就是将多项式的各个系数依次存放在一个数组中,  $n$  次多项式就有  $n+1$  个元素, 这样只要重复计算  $n$  个一次式就可以得到该多项式的值。如果多项式中缺某一项, 其系数要用 0 补齐。编程如下:

```
#include <iostream>
using namespace std;
int main(void) {
    const int n = 5; // n 次多项式
    double a[n+1] = {5, 0, -3.2, 2, 6.2, -8}, y = a[0], x;
    cout<<"input x=";
    cin>>x;
    for(int i = 0; i < n; i++)
        y = y * x + a[i+1];
    cout<<"y="<<y<<endl;
    system("pause");
    return 0;
}
```

如果要计算另一个一元  $n$  次多项式, 只需改变  $n$  的值和数组  $a$  的初始化部分即可。但是更好的设计是分离出来一个函数来专门计算一元  $n$  次多项式的值。编程如下:

```
#include <iostream>
using namespace std;
double multinomial(int n, double a[], double x) {
    double y = a[0];
    for(int i = 0; i < n; i++)
        y = y*x + a[i+1];
    return y;
}

int main(void) {
    const int n = 5;
    double a[n+1] = {5, 0, -3.2, 2, 6.2, -8}, x;
    cout<<"input x=";
    cin>>x;
    cout<<"y="<<multinomial(n, a, x)<<endl;
    system("pause");
    return 0;
}
```

上面例子中设计了一个函数来计算一元  $n$  次多项式的值。第 1 个形参表示  $n$  次, 第 2 个形参是一个一维数组, 存放  $n+1$  个系数。第 3 个形参是  $x$  的值。该函数返回一个值作为多项

式的值。

例 6-2 对一个数组中的  $n$  个元素按由小到大的顺序排列，即升序排序。

例如原先数据为  $\text{int } a[6] = \{6, 4, 2, 1, 3, 5\}$ ，排序之后的结果是  $\{1, 2, 3, 4, 5, 6\}$ ；

数据排序方法是常用算法。排序算法很多，下面介绍选择排序算法和冒泡排序算法。

## (2) 选择排序

以升序为例，对  $n$  个元素进行选择排序，可分为  $n-1$  轮。第 1 轮，在所有  $n$  个元素中选择一个最小元素，放在头一个元素位置，这个过程需要  $n-1$  次比较和 1 次交换。此时头一个元素  $a[0]$  就是最小值，而剩余的  $n-1$  个元素(从  $a[1]$  到  $a[n-1]$ )仍然是无序的。第 2 轮就对剩余的元素再次选择一个最小的放在次排头位置  $a[1]$ 。如此循环  $n-1$  轮，就能对  $n$  个元素实现升序排列。每一轮中主要操作就是比较和交换。

下面以  $\text{int } a[6] = \{6, 4, 2, 1, 3, 5\}$  为例，说明每一轮的过程。设外层循环变量  $i$  从 0 循环到 4。在每一轮过程中，设  $\text{minValue}$  为最小值， $\text{minIndex}$  为最小值的下标。在每一轮开始时，设头一个元素为最小值，然后依次与后面元素比较。在每次比较之后，如果发现更小的元素，就更新这两个值。当比较完成之后，如果最小值不是头一个元素，就将最小值与头一个元素进行交换，使头一个元素成为参与排序的元素的最小值。

第 1 轮:  $i = 0$ ;

$\text{minValue} = 6, \text{minIndex} = 0,$        $\boxed{6}, \boxed{4}, 2, 1, 3, 5$

$\text{minValue} = 4, \text{minIndex} = 1,$        $6, 4, \boxed{2}, 1, 3, 5$

$\text{minValue} = 2, \text{minIndex} = 2,$        $6, 4, 2, \boxed{1}, 3, 5$

$\text{minValue} = 1, \text{minIndex} = 3,$        $6, 4, 2, 1, \boxed{3}, 5$

$\text{minValue} = 1, \text{minIndex} = 3,$        $6, 4, 2, 1, 3, \boxed{5}$

$a[0]$  与  $a[\text{minIndex}]$  交换:       $\boxed{1}, 4, 2, \boxed{6}, 3, 5$

$a[0]$  为最小值，不参与下面循环

第 2 轮:  $i = 1$ ;

$\text{minValue} = 4, \text{minIndex} = 1,$        $1, \boxed{4}, \boxed{2}, 6, 3, 5$

$\text{minValue} = 2, \text{minIndex} = 2,$        $1, 4, 2, \boxed{6}, 3, 5$

$\text{minValue} = 2, \text{minIndex} = 2,$        $1, 4, 2, 6, \boxed{3}, 5$

$\text{minValue} = 2, \text{minIndex} = 2,$        $1, 4, 2, 6, 3, \boxed{5}$

$a[1]$  与  $a[\text{minIndex}]$  交换:       $1, \boxed{2}, \boxed{4}, 6, 3, 5$

前两个有序，不参与下面循环

第 3 轮:  $i = 2$ ;

$\text{minValue} = 4, \text{minIndex} = 2,$        $1, 2, \boxed{4}, \boxed{6}, 3, 5$

$\text{minValue} = 4, \text{minIndex} = 2,$        $1, 2, 4, 6, \boxed{3}, 5$

$\text{minValue} = 3, \text{minIndex} = 4,$        $1, 2, 4, 6, 3, \boxed{5}$

$a[2]$  与  $a[\text{minIndex}]$  交换:       $1, 2, \boxed{3}, 6, \boxed{4}, 5$

前三个有序，不参与下面循环

第 4 轮:  $i = 3$ ;

$\text{minValue} = 6, \text{minIndex} = 3,$        $1, 2, 3, \boxed{6}, \boxed{4}, 5$

$\text{minValue} = 4, \text{minIndex} = 4,$        $1, 2, 3, 6, 4, \boxed{5}$

a[3]与 a[minIndex]交换:      1, 2, 3, 4, 6, 5      前四个有序, 不参与下面循环  
 第 5 轮: i = 4;  
 minValue = 6, minIndex = 4      1, 2, 3, 4, 6, 5  
 a[4]与 a[minIndex]交换:      1, 2, 3, 4, 5, 6      前五个有序, 全部升序排列, 结束

按上面过程编程如下:

```
#include <iostream>
using namespace std;
int main(void){
    const int n = 6;
    int a[n] = {6, 4, 2, 1, 3, 5};
    int i, j, minValue, minIndex, temp;
    for(i = 0; i < n - 1; i++){           //外层循环控制轮次
        minValue = a[i];                 //假设排头数据最小
        minIndex = i;
        for(j = i + 1; j < n; j++){       //内层循环控制每轮比较次数
            if(minValue > a[j]){           //是否找到更小的数
                minValue = a[j];
                minIndex = j;
            }
        }
        if(i != minIndex){                //是否需要交换
            temp = a[i];
            a[i] = a[minIndex];
            a[minIndex] = temp;
        }
    }
    for(auto y : a)                       //输出排序后的数据
        cout<<y<<'\\t';
    cout<<endl;
    system("pause");
    return 0;
}
```

选择排序算法中关键是两层嵌套循环和两个条件语句。选择排序算法的特点是每一轮先用内层循环选择一个最小元素, 然后再看是否需要交换。对  $n$  个元素排序最多需要  $n-1$  次交换。比较次数是  $n*(n-1)/2$ 。

对于上面的编程, 如果要适应其它不同长度和内容的数组排序, 最好还是用函数来实现选择排序和数组输出。程序修改如下:

```
#include <iostream>
using namespace std;
void selectSort(int data[], int n){
    int i, j, minValue, minIndex, temp;
    for(i = 0; i < n-1; i++){
        minValue = data[i];
        minIndex = i;
        for(j = i + 1; j < n; j++){
            if(minValue > data[j]){
                minValue = data[j];
                minIndex = j;
            }
        }
        if(i != minIndex){
```



```
        temp = data[i];
        data[i] = data[minIndex];
        data[minIndex] = temp;
    }
}

int main(){
    int a[6] = {6, 4, 2, 1, 3, 5};
    selectSort(a, 6);
    for (auto y : a)
        cout << y << "\t";
    system("pause");
    return 0;
}
```

上面函数 `selectSort` 对一个 `int` 数组按选择排序算法进行升序排列。第一个形参 `data` 是一个 `int` 数组，不需要确定大小，第 2 个形参 `n` 确定该数组的大小。函数 `printArray` 用来将一个 `int` 数组打印出来。这两个函数仅能处理 `int` 数组，读者可自行修改使其能处理 `double` 数组和 `float` 数组。

数组作为函数形参，如果形参前有 `const` 修饰，那么该数组的元素在函数中就不能被改变，只能读取元素。反之，如果没有 `const` 修饰，函数中就可以改变数组元素作为结果。例如，`void selectSort(int data[], int n)` 函数要改变 `data` 数组元素，就不能添加 `const` 修饰。而 `void printArray(const int data[], int n)` 函数只能读取 `data` 数组元素，而不能改变。

上面实现的选择排序算法具有较少的交换次数，但需要两个变量来记录每一轮的最小值 `minValue` 及下标 `minIndex`。还有一种比较简单的实现，编程如下：

```
void selectSort2(int data[], int n){
    int i, j, temp;
    for(i = 0; i < n-1; i++){
        for(j = i + 1; j < n; j++){
            if(data[i] > data[j]){
                temp = data[i];
                data[i] = data[j];
                data[j] = temp;
            }
        }
    }
}
```

这个排序算法让外层控制的 `data[i]` 与内层控制的 `data[j]` 之间比较并直接交换，在每一轮之后，最小的元素放在最前面。与前一种方式相比较，最多交换的次数与比较次数相同，都是  $n*(n-1)/2$ 。虽然交换次数可能多，但编程简单，在性能要求不高的情况下可用。

### (3) 冒泡排序

冒泡排序的特点是比较和交换连续进行，就像水中的气泡，比较轻的(较小的)不断向上冒，比较重的(较大的)就沉在下面，上冒一次就需要一次交换。以升序为例，对 `n` 个元素进行冒泡排序，也分为 `n-1` 轮。每一轮进行相邻比较和交换，结果是使最大元素排在最后。在下一轮冒泡中，最后元素不参与比较和交换。

下面以 `int a[6] = {6, 4, 2, 1, 3, 5}` 为例，说明每一轮的过程。设外层循环变量 `i` 从 0 循环到

4, 共 5 轮外层循环, 使得最后位置存放最大值。在每一轮中, 需要一个内层循环, 设循环变量  $j$  从 0 到  $4-i$ , 比较次数是  $5-i$  次, 最多交换  $5-i$  次。比较和交换在相邻元素  $a[j]$  和  $a[j+1]$  之间进行。

第 1 轮, $i = 0$	<u>6</u> , <u>4</u> , 2, 1, 3, 5	
$j = 0$ , 比较, 交换	<u>4</u> , <u>6</u> , 2, 1, 3, 5	
$j = 1$ , 比较, 交换	4, <u>2</u> , <u>6</u> , 1, 3, 5	
$j = 2$ , 比较, 交换	4, 2, <u>1</u> , <u>6</u> , 3, 5	
$j = 3$ , 比较, 交换	4, 2, 1, <u>3</u> , <u>6</u> , 5	
$j = 4$ , 比较, 交换	4, 2, 1, 3, <u>5</u> , <u>6</u>	//最后一个最大, 不参与下一轮
第 2 轮, $i = 1$	<u>4</u> , <u>2</u> , 1, 3, 5, 6	
$j = 0$ , 比较, 交换	<u>2</u> , <u>4</u> , 1, 3, 5, 6	
$j = 1$ , 比较, 交换	2, <u>1</u> , <u>4</u> , 3, 5, 6	
$j = 2$ , 比较, 交换	2, 1, <u>3</u> , <u>4</u> , 5, 6	
$j = 3$ , 比较	2, 1, 3, <u>4</u> , <u>5</u> , 6	//最后两个有序, 不参与下一轮
第 3 轮, $i = 2$	<u>2</u> , <u>1</u> , 3, 4, 5, 6	
$j = 0$ , 比较, 交换	<u>1</u> , <u>2</u> , 3, 4, 5, 6	//至此排序完成, 但比较过程仍继续
$j = 1$ , 比较	1, <u>2</u> , <u>3</u> , 4, 5, 6	
$j = 2$ , 比较	1, 2, <u>3</u> , <u>4</u> , 5, 6	//最后三个有序, 不参与下一轮
第 4 轮, $i = 3$	<u>1</u> , <u>2</u> , 3, 4, 5, 6	
$j = 0$ , 比较	<u>1</u> , <u>2</u> , 3, 4, 5, 6	
$j = 1$ , 比较	1, <u>2</u> , <u>3</u> , 4, 5, 6	//最后四个有序, 不参与下一轮
第 5 轮, $i = 4$	<u>1</u> , <u>2</u> , 3, 4, 5, 6	
$j = 0$ , 比较	<u>1</u> , <u>2</u> , 3, 4, 5, 6	//最后五个有序, 全部升序排列, 结束

根据上面过程, 设计一个函数完成冒泡排序, 编程如下:

```
void bubbleSort(int data[], int n){
    int i, j, temp;
    for(i = 0; i < n-1; i++)                //外层循环控制总轮次
        for(j = 0; j < n-1-i; j++)          //内层循环控制比较次数
            if(data[j] > data[j+1]){         //是否需要相邻交换
                temp = data[j];
                data[j] = data[j+1];
                data[j+1] = temp;
            }
}
```

冒泡排序算法的特点是比较和交换在两个相邻元素之间进行。对  $n$  个元素排序, 比较次数是  $n*(n-1)/2$ , 这与选择排序一样, 但最多交换次数也等于比较次数, 大于选择排序的  $n-1$  次交换。理论上说, 冒泡排序算法的复杂度更高, 但代码实现简单。

冒泡排序算法的一种改进算法称为快速排序, C 语言标准库 `<stdlib.h>` 中给出了一个快速排序库函数 `qsort`, 可对任意类型数组进行排序。其中涉及到函数指针, 到第 8 章介绍。

为了方便测试排序算法, 下面一个函数产生伪随机数序列来填充一个数组:

```
#include <stdlib.h>
```

```

#include <time.h>
#include <iostream>
using namespace std;

void makeRandom(int data[], int n){
    srand(time(NULL));           //取当前时间秒数作为随机数发生器种子
    for(int i = 0; i < n; i++){
        data[i] = rand() % 100;  //填充 100 以内的随机数
    }
}

```

按如下方式来测试各种排序算法:

```

int main(){
    int a4[10];
    makeRandom(a4, 10);          //填充随机数
    for (auto y : a4)             //打印排序前的数据
        cout << y << '\t';
    bubbleSort(a4, 10);          //调用某一种排序算法
    for (auto y : a4)             //打印排序后的数据
        cout << y << '\t';
    system("pause");
    return 0;
}

```

#### (4) 两分查找

例 6-3 已知一个数组中的元素是有序的, 如升序。给定一个值, 在该数组中查找是否存在, 如果存在就返回其下标。例如在升数序列{-56, -23, 0, 8, 10, 12, 26, 38, 65, 98}中, 如果查找数据为 38, 应返回下标 7。如果查找数据为 39, 应返回-1, 表示未找到。

在一个数组中查找一个值, 最简单的就是顺序查找, 也称为线性查找。这样最多比较次数就是  $n$  次。但如果该数组元素是有序的, 就有一种快速查找方法, 称为两分查找法, 能使最多比较次数减半, 即  $n/2$  次。

两分查找也称为折半查找、或字典查找(英文字典按字母顺序升序排列, 中文字典按中文拼音字母顺序升序排列)。过程如下:

- 1、将  $n$  个元素确定为一个查找范围, 求中间位置  $\text{binary}=n/2$ 。
- 2、如果  $x$  等于  $a[\text{binary}]$ , 就找到了, 返回  $\text{binary}$  下标, 结束。

如果  $x$  小于  $a[\text{binary}]$ , 就将前面  $\text{binary}$  个元素作为下一轮查找的范围, 到第 1 步。

如果  $x$  大于  $a[\text{binary}]$ , 就将后面  $\text{binary}-1$  个元素作为下一轮查找的范围, 到第 1 步。

如此递推, 每一轮查找时, 查找范围折半。直到找到, 或未找到。

关键之一是如何确定一个查找范围。设置 3 个变量:  $\text{low}$  指向查找范围的底部,  $\text{high}$  指向查找范围的顶部,  $\text{binary}$  指向折半的位置, 即查找范围的中部  $\text{binary} = (\text{low}+\text{high})/2$ 。

关键之二是如何确定未找到而停止循环。在前半区查找时, 使  $\text{high}=\text{binary}-1$ ; 在后半区查找时, 使  $\text{low}=\text{binary}+1$ 。如果  $\text{low} > \text{high}$ , 未找到, 停止循环。编程实现如下:

```

#include <iostream.h>
int searchBy(int data[], int n, int x){
    int low = 0;                      //标识查找区间
    int high = n-1;
    int binary = (low + high) / 2;     //确定折半位置
}

```

```

while(x != data[binary] && low <= high){    //循环条件控制
    if(x < data[binary])
        high = binary - 1;                //在前半区间查找
    else
        low = binary + 1;                  //在后半区间查找
    binary = (low + high) / 2;
}
if (low <= high)
    return binary;
else
    return -1;
}

int main(void){
    int s[10]={-56,-23,0,8,10,12,26,38,65,98 };
    int x;
    while(1){
        cout<<"input (-100 to exit)x=";
        cin>>x;                            //输入待查找数据
        if (x == -100)
            break;
        int i = searchBy(s, 10, x);
        if (i == -1)
            cout<<"Not found"<<endl;
        else
            cout<<"Found at "<<i<<endl;
    }
}

```

上面例子中设计了一个 `searchBy` 函数来完成两分查找。第 1 个形参确定一个待查数组，该数组的元素应该按升序排列。第 2 个形参确定了该数组的长度。第 3 个形参确定了要查找的数据。返回值如果为-1，表示未找到。如果不是-1，就是找到的元素的下标。如果要查找的值有多个同值元素，就不能确定究竟是哪一个下标被返回。

C 标准库 `<stdlib.h>` 中提供了一个两分查找库函数 `bsearch`。其中涉及函数指针，到第 8 章介绍。C++ 标准库 `<algorithm>` 提供的两份查找函数 `binary_search` 容易使用。

### 6.1.6 调用算法来简化编程

前面介绍的排序和查找算法只是原理性说明，真正的软件工程中应尽可能调用系统提供的现成的算法来简化编码，避免下标越界，提高编码质量。

C++ 提供了一组算法 `<algorithm>`，可对一维数组进行各种计算，如排序、二分查找、遍历、线性查找、随机生成等等。对算法 `<algorithm>` 的详细介绍见第 13 章。下面将数组作为一种简单容器来调用算法中的函数，以简化编程。下面是一个例子。

```

#include<iostream>
#include<algorithm>                //A
using namespace std;
int main() {
    int a[10];                    //B 定义 10 个 int 元素的数组

```

```

generate(a, a + 10, rand);           //调用 generate 生成伪随机数
for (auto x : a)                     //输出显示
    cout << x << " ";
cout << endl;
sort(a, a + 10, [](auto x, auto y) {return x<y; }); //调用 sort 升序排序
for (auto x : a)                     //输出显示
    cout << x << " ";
cout << endl;
if (binary_search(a, a + 10, 15724)) //调用 binary_search 二分查找
    cout << "15724 found" << endl;
else
    cout << "15724 no found" << endl;
auto it = find(a, a + 10, 15724);    //调用 find 做线性查找
if (it == a + 10)
    cout << "15724 no found" << endl;
else
    cout << *it << " found by find" << endl;
system("pause");
return 0;
}

```

执行程序，输出结果如下：

```

41 18467 6334 26500 19169 15724 11478 29358 26962 24464
41 6334 11478 15724 18467 19169 24464 26500 26962 29358
15724 found
15724 found by find

```

A 行包含头文件<algorithm>，然后就可调用其中大量的有用的函数。

B 行定义 10 个 int 元素的数组，作为被调用函数的操作对象。

第 1 行输出是产生随机数之后。

第 2 行输出是升序排序之后。

上面调用了 4 个函数，每个函数的前 2 个实参都是确定数据范围：a, a+10, a 是头一个元素，a+10 是最后元素的下一个空位。

函数 generate 的第 3 个实参是 rand，就是<stdlib.h>的 rand 函数，取随机数。

函数 sort 的第 3 个实参是一个 Lambda 表达式，确定升序排序的比较规则。

函数 binary\_search 执行二分查找，第 3 个实参就是要找的值。返回 bool 值表示是否找到。注意该函数仅对升序排序的数组查找管用。

函数 find 执行线性查找(从头到尾查找)，第 3 个实参就是要找的值，返回一个称作迭代器的变量，相当下标，如果下标到达 a+10 尾端就是未找到，如果未到达尾端，就是找到了，而且\*it 就是要找的值，可理解为按下标找出值。

如此看来，这 4 个函数都可简单使用，读者可模仿改变，最后完成自己的编程。

上面代码中输出显示的 3 行代码出现了两次，你可能希望编写一个简单函数来简化。

```
void print(int a[], int n) {
    for_each(a, a + n, [](auto x) {cout << x << " "; });
    cout << endl;
}
```

上面 `print` 函数中你不能使用基于范围的 `for` 语句: `for(auto x:a){...}`, 编译错误。但可用 `for_each` 函数, 称做遍历函数, 第 3 个实参是一个 Lambda 表达式, 说明要对遍历到的每个元素 `x` 要做什么。这样就可以调用该函数来简化输出显示: `print(a,10);`

如何简单计算这 10 个元素的平均值? 编写一个函数来实现:

```
double getAverage(int a[], int n) {
    double sum = 0;
    for_each(a, a + n, [&sum](auto x) {sum += x; });
    return sum / n;
}
```

函数中第 2 行调用 `for_each` 函数, 第 3 个实参是一个 Lambda 表达式, 此时注意其中 `[&sum]` 表示它要捕获外边的 `sum` 变量, 而且要改变其值, 称作“引用捕获”。最简单形式是 `[&]`, 说明该 Lambda 表达式用引用方式访问外边所有变量。引用和 Lambda 表达式在第 8 章介绍。

上面例子中调用了 `<algorithm>` 的 5 个函数, 作用于一个 `int a[10]` 数组, 编码简单实用, 没有使用任何下标循环, 因此也不存在下标越界的危险。读者可参照相关文档尝试其它函数, 也可改变代码用做其它目的。

## 6.2 二维数组

二维数组就是具有两个下标的数组, 确定一个元素需要两个下标值。二维数组经常用来表示矩阵或者二维表格。习惯上将二维数组的第一个下标称为行下标, 第二个下标称为列下标。

### 6.2.1 二维数组的定义

二维数组的定义与一维数组类似, 其语法格式为:

`<数据类型> <数组名>[<常量表达式 1>][<常量表达式 2>]`

例如:

```
int a[3][5], b[2][3];           // 数组 a 是 3 行 4 列, 数组 b 是 2 行 3 列
float score[30][6];           // 数组 score 是 float 元素, 30 行 6 列
```

`a` 数组是一个二维数组, 包含 15 ( $3 \times 5$ ) 个数组元素。可以将二维数组 `a[3][5]` 看成是由 3 个元素组成的一维数组: `a[0]`、`a[1]`、`a[2]`。而 `a[0]`、`a[1]`、`a[2]` 又是包含 5 个元素的一维数组:

```
a[0]:  a[0][0] a[0][1] a[0][2] a[0][3] a[0][4]
a[1]:  a[1][0] a[1][1] a[1][2] a[1][3] a[1][4]
a[2]:  a[2][0] a[2][1] a[2][2] a[2][3] a[2][4]
```

`a` 数组中的各个元素在内存中按先行后列的顺序存储。

由于二维数组可被看作一维数组，故此二维数组也称为数组的数组。

### 6.2.2 二维数组的初始化

二维数组的初始化与一维数组类似。例如，下面都是正确的初始化数组元素的格式。

```
int a[3][4]={{1,2,3,4},{3,4,5,6},{5,6,7,8}}; //按行初始化数组的全部元素
int b[3][4]={1,2,3,4,3,4,5,6,5,6,7,8};      //按数据顺序初始化数组的全部元素
int c[][3]={{1,3,5},{5,7,9}};                //初始化全部数组元素，隐含行数为 2
int d[][3]={{1},{0,1},{0,0,1}};              //初始化部分数组元素，其余值为 0
```

当初始化数组的所有元素时，数组 a 与数组 b 的初始化是等价的，可以省略内层的花括号。这里隐含着一维数组与二维数组之间的等同性，在第 8 章可实现两者之间的转换。

通常，用矩阵形式表示二维数组中各元素的值。下面用 3 个矩阵形式表示上述 3 个数组初始化后各元素对应的数据值。

a 数组:	c 数组:	d 数组:
1   2   3   4	1   3   5	1   0   0
3   4   5   6	5   7   9	0   1   0
5   6   7   8		0   0   1

定义二维数组时，如果对所有元素赋初值，可以不指定该数组的行数，系统会根据初始表中的数据个数自动计算数组的行数，但一定要指定列数。

### 6.2.3 二维数组的应用

在二维数组中，每个元素是通过数组名及行、列下标来确定的。

遍历一个二维数组，一般需要两层循环：行循环和列循环。但不能简单采用基于范围的 for 的两层循环。对行循环可采用范围 for，对于列循环就要用下标，例如：

```
int b[3][4] = { {1,2,3,4},{5,6,7,8},{9,10,11,12} };
for (auto m : b){
    for (int i = 0; i < 4; i++)
        cout << m[i]<< " ";
    cout << endl;
}
```

上面编码可验证，二维数组是一维数组的数组。

例 6-4 已知一个 3\*4 的矩阵 a，将其转置后输出。

矩阵转置就是将矩阵的行列互换。生成一个新的矩阵 b，将 a 矩阵的 a[i][j] 元素变成 b 矩阵的 b[j][i] 元素。算法为：b[i][j]=a[j][i]。例如：

a 矩阵:	b 矩阵:
1   2   3   4	1   3   5
3   4   5   6	2   4   6
5   6   7   8	3   5   7
	4   6   8

编程如下：

```
#include <iostream>
using namespace std;
int main(void){
```

```

int a[][4]={    {1,2,3,4},
                 {3,4,5,6},
                 {5,6,7,8}};

int b[4][3], i, j;
for(i=0;i<3;i++){                                //输出 a 矩阵
    for(j=0;j<4;j++)
        cout<<a[i][j]<<'\\t';
    cout<<endl;
}
for(i=0;i<4;i++){                                //实现矩阵转置
    for(j=0;j<3;j++)
        b[i][j]=a[j][i];
for(i=0;i<4;i++){                                //输出转置后的 b 矩阵
    for(j=0;j<3;j++)
        cout<<b[i][j]<<'\\t';
    cout<<endl;
}
system("pause");
return 0;
}

```

二维数组作为函数形参时有限制,即高维不能确定大小(即便确定了也没用),而低维一定要确定大小。这样当要传递不同列的数组时就要提供不同的函数,而且还要再添加一个形参来表示行数。对上面例子添加两个打印矩阵函数,用重载函数实现,编程如下:

```

#include <iostream>
using namespace std;

void print2D(int a[][4], int row){                //打印 row 行 4 列的矩阵
    for(int i = 0; i < row; i++){
        for(int j = 0; j < 4; j++){
            cout<<a[i][j]<<'\\t';
        }
        cout<<endl;
    }
}

void print2D(int a[][3], int row){                //打印 row 行 3 列的矩阵
    for(int i = 0; i < row; i++){
        for(int j = 0; j < 3; j++){
            cout<<a[i][j]<<'\\t';
        }
        cout<<endl;
    }
}

int main(void){
    int a[][4]={1,2,3,4},
               {3,4,5,6},
               {5,6,7,8}};

    int b[4][3],i,j;
    print2D(a, 3);                                //打印 a[3][4]
    for(i=0;i<4;i++){                              //实现矩阵转置
        for(j=0;j<3;j++)
            b[i][j] = a[j][i];
    print2D(b, 4);                                //打印 b[4][3]
    system("pause");
    return 0;
}

```

上面代码中打印不同列的二维数组需要不同的函数,到第 8 章基于指针能实现任意行任意列的二维数据打印。



例 6-5 编程实现两个矩阵的乘法运算。

根据矩阵乘法的运算规律： $M[m1][n] \times N[n][n2] = Q[m1][n2]$ ，矩阵 M 的列数必须等于矩阵 N 的行数，两个矩阵才能相乘，并且矩阵 Q 的行数等于矩阵 M 的行数 m1，Q 的列数等于

矩阵 N 的列数 n2。二维数组 m、n、q 中，则  $q[i][j] = \sum_{k=1}^n m[i][k] \times n[k][j]$ 。

下面是求  $M[3][4] \times N[4][3] = Q[3][3]$  的程序：

```
#include <iostream>
using namespace std;
void print2D(int a[][3], int row){           //打印 row 行 3 列的矩阵
    for(int i = 0; i < row; i++){
        for(int j = 0; j < 3; j++){
            cout<<a[i][j]<<'\\t';
        }
        cout<<endl;
    }
}
int main(void){
    int m[3][4]={    {1,2,3,4},
                      {2,2,3,1},
                      {5,4,2,3}};

    int n[4][3]={    {6,3,2},
                      {2,8,1},
                      {6,9,5},
                      {2,4,6}};

    int i,j,k,q[3][3] = {0};                //A

    for(i = 0; i < 3; i++){
        for(j = 0; j < 3; j++){
            for(k = 0; k < 4; k++){          //B
                q[i][j] += m[i][k] * n[k][j];
            }
        }
    }

    print2D(q, 3);                          //输出矩阵 q
    system("pause");
    return 0;
}
```

A 行将数组 q 中的所有元素赋初值为 0，B 行循环求出数组 q 中的一个元素。

执行程序，输出结果如下：

36	62	43
36	53	27
56	77	42

例 6-6 按下列格式打印杨辉三角(也称为 Pascal 三角)。要求打印前 10 行。

```

      1
    1  1
  1  2  1
1  3  3  1
1  4  6  4  1
1  5 10 10  5  1
1  6 15 20 15  6  1
1  7 21 35 35 21  7  1
```

```

      1   8  28  56  70  56  28   8   1
    1   9  36  84 126 126  84  36   9   1

```

杨辉三角形类似于一个表格形式。第 2 行开始，头尾都是 1，其它数字等于其上方相邻两个数字之和。可用一个二维数组来存放。这样需要两步计算。

第 1 步，生成二维数组 `a[10][10]`。

第 2 步，控制打印输出。

关键是第 1 步。把前 5 行按二维数组来分析：

```

1
1  1
1  2  1
1  3  3  1
1  4  6  4  1

```

可以看出数据元素的分布规律。数组中头一列 `a[i][0]` 和最后一列 `a[i][i]` 元素都为 1。其它每个元素都是其上一行同列元素与上一行前一列元素之和： $a[i][j] = a[i-1][j-1] + a[i-1][j]$ 。编程如下：

```

#include <iostream>
#include <iomanip>
using namespace std;
int main(void){
    const int m = 10;
    int a[m][m], i, j;

    for(i = 0; i < m; i++){                //生成数据
        a[i][0] = 1;                        //将数组中第 0 列元素 a[i][0] 置 1
        a[i][i] = 1;                        //将数组中最后一列元素 a[i][i] 置 1
        for(j = 1; j < i; j++){
            a[i][j] = a[i-1][j-1] + a[i-1][j];    //计算其它元素的值
        }

        for(i = 0; i < m; i++){              //按要求格式输出
            for(int k = 0; k < 30-2*i; k++){
                cout<<" ";
            }
            for(j = 0; j <= i; j++){
                cout<<setw(5)<<a[i][j];
            }
            cout<<endl;
        }
    }
    system("pause");
    return 0;
}

```

### 6.3 数组与函数

在定义一个函数时，数组可作为函数的形参。前面我们已用到一维数组作为函数形参的例子。一个函数不能返回一个数组，往往返回一个指针来指向一个数组(指针在第 8 章介绍)。数组的元素具有特定类型和值，在调用一个函数时，数组元素可作为函数调用的实参，条件是类型相符。

一个数组的名称实际上是该数组元素存储区域的首地址。一个数组作为一个函数形参，

函数调用时用一个数组作为实参。实参数组传递给形参数组时，并没有传递实参数组中的各个元素，而是把实参数组的首地址传递给形参，这样函数体中就能利用形参数组来访问实参数组中的元素，如果没有 `const` 修饰该形参，函数体中还能改变数组中的元素。

由于数组实参传递给形参只是数组的地址，而没有包含实参数组的长度，所以往往需要再增加一个形参来表示数组的长度。

如果一个函数产生一个数组作为计算输出结果，可采用数组形参，但应返回结果元素的个数。调用方先定义一个空数组，作为实参来调用函数，函数返回后不仅填入元素，而且返回元素的个数，调用方就能使用该计算结果。

例 6-7 设计一个函数，对任一个正整数分解质因数，并且将各因数存放在一个 `int` 数组中。例如： $15 = 3 \times 5$ ； $16 = 2 \times 2 \times 2 \times 2$ ； $17 = 17$ ； $18 = 2 \times 3 \times 3$ ，等。

该函数原型设计为：`int primeFactors(int n, int a[]);`

形参 `n` 就是要分解的整数，分解的因子将存放到数组 `a` 中，返回一个整数表示因子的个数。同时约定，如果 `n < 2`，就返回 0。

前面曾介绍一个函数 `bool isPrime(int n)` 判断整数 `n` 是否为素数(即质数)。在此基础上再设计一个函数 `int nextPrime(int n)`，返回大于 `n` 的下一个素数。编程如下：

```
#include <iostream>
#include <assert.h>
using namespace std;

bool isPrime(int n){
    if (n < 2)
        return false;
    if (n == 2 || n == 3 || n == 5 || n == 7)
        return true;
    for (int i = 2; i * i <= n; i++)
        if (n % i == 0)
            return false;
    return true;
}
//返回大于 n 的最小素数
int nextPrime(int n){
    int i = n + 1;
    while (!isPrime(i))
        i++;
    return i;
}
//对 n 分解质因数，因数存放到数组 a 中，返回因数个数
int primeFactors(int n, int a[]){
    if (n < 2) return 0;
    if (isPrime(n)){
        a[0] = n;
        return 1;
    }
    int k = 2, i = 0;
    while(1){
        while (n > 1 && n % k == 0){
            a[i++] = k;
            n = n / k;
        }
        if (n == 1) break;
    }
    //k 作为最小的因数，i 作为数组下标
    //如果 n>1 且被 k 整除
    //得到一个因子 k
    //求商 n，再循环
    //当商等于 1 时，所有因数分解完成
```

```

        k = nextPrime(k);                //否则，求下一个素数
    }
    return i;
}
//测试程序，测试大于 1 的整数
void test1(int n){
    if (n < 2) return;
    int fact[32];                        //A 最多 31 个因数
    int num = primeFactors(n, fact);
    int m = 1;
    for (int i = 0; i < num; i++){
        cout<<fact[i]<<"*";
        m *= fact[i];
    }
    cout<<"\b"<<"="<<m<<endl;        //' \b'表示回退一个字符
    assert(m == n);                    //断言
}
int main(){
    for(int i = 2; i < 21; i++)          //测试 2-20 之间的整数
        test1(i);
    system("pause");
    return 0;
}

```

执行程序，输出结果如下：

```

2=2
3=3
2*2=4
5=5
2*3=6
7=7
2*2*2=8
3*3=9
2*5=10
11=11
2*2*3=12
13=13
2*7=14
3*5=15
2*2*2*2=16
17=17
2*3*3=18
19=19
2*2*5=20

```

函数 `int primeFactors(int n, int a[])` 在调用前要先说明一个 `int` 数组，这个数组的大小应该能容纳最大 `int` 值分解的所有因子，因为最大的 `int` 值不超过  $2^{31}$ ，因此因子数组大小应不小于 31。A 行说明的数组大小为 32，是符合要求的。

在测试函数 `test1` 中使用了 `<assert.h>` 中的一个有参宏 `assert` 断言。

一维数组作为形参时，可以不指定数组的大小，需要另加一个形参来指定大小。

二维数组作为形参时，至少要确定列的大小，这样就限制了实参数组选择的灵活性。

例 6-8 设计一个程序，求两个矩阵的和。

矩阵加法运算规则： $A[m][n] + B[m][n] = C[m][n]$ 。只有矩阵 A 的行数和列数分别等于矩阵 B 的行数和列数时，这两个矩阵才能相加。矩阵 C 的行数和列数等于矩阵 A 的行数和列

数。 $c_{ij}=a_{ij}+b_{ij}$ 。以  $n$  行 4 列数组为例，编程如下：

```
#include <iostream>
using namespace std;
void plus4(int x[][4], const int y[][4], int n){
    for(int i = 0; i < n; i++)
        for(int j = 0; j < 4; j++)
            x[i][j] += y[i][j];
}

void print2D(const int c[][4], int n){    //输出矩阵各元素的值
    for(int i = 0; i < n; i++){
        for(int j = 0; j < 4; j++)
            cout<<c[i][j]<<"\t";
        cout<<endl;
    }
}

int main(void){
    int a[3][4]={    1,3,4,2,
                    2,3,1,2,
                    3,5,4,2};
    int b[3][4]={    3,2,6,5,
                    4,3,7,2,
                    3,3,1,4};

    plus4(a, b, 3);    //数组作为实参
    print2D(a, 3);    //数组作为实参
    system("pause");
    return 0;
}
```

多维数组作形参时，可以不指定最高维的大小，但其它各维的大小必须指定。上面函数只能处理  $n$  行 4 列的数组。在第 8 章将介绍如何用指针来定义任意行任意列的二维数组作为形参。

## 6.4 字符数组与字符串

在程序中经常要处理各种字符文本数据，例如人的姓名、住址、身份证号等。传统 C 语言没有提供语言级字符串 `string` 类型。字符串往往要通过 `char` 数组或 `char` 指针来实现。下面介绍字符数组，字符数组所支持的字符串。

字符数组就是字符型元素组成的数组。一个字符串是由 0 值结尾的字符数组。字符数组也分为一维数组和多维数组，一维数组可存放一个字符串，多维数组可存放多个字符串。下面主要介绍一维字符数组的定义、初始化和操作。上节介绍的一般性数组的定义及初始化方法同样适用于字符数组，但字符串自有其特点。

本文所涉及的字符串有三种：基于字符数组的字符串；基于字符指针的字符串(第 8 章介绍)；基于 `<string>` 的字符串类型(第 10 章介绍)。

### 6.4.1 字符数组的定义

字符数组定义的语法格式与一般性数组的定义一样。

```
char <数组名>[<常量表达式>]
```

其中, <数组名>是标识符, <常量表达式>的值必须是正整数, 说明数组的大小, 即字符的个数。例如:

```
char s1[8], s2[10];           //定义两个字符型数组 s1, s2
char s3[5][10];              //定义一个二维字符数组 s3
```

其中, s1 数组包含 5 个字符元素。s1 数组中的每个元素都是 char 变量。s3 是一个二维数组, 可以存放 5 个字符串, 每个字符串最多可以存放 10 个字符。

字符数组还有宽字符 wchar\_t 的数组。例如:

```
wchar_t a[5];                 //5 个 wchar_t 占用 10 个字节。
```

Unicode 编码的字符类型 char16\_t 与 char32\_t 也可定义数组。

## 6.4.2 字符数组的初始化

对字符数组可用字符或字符串进行初始化。

### 1.用字符常量进行初始化

语法格式为:

```
char <数组名>[<常量表达式 1>] = {<字符常量初值表>};
```

这种方法就是一般数组的初始化方法, 要逐个列出各个元素的值, 比较麻烦。例如:

```
char s1[8] = {'C', 'o', 'm', 'p', 'u', 't', 'e', 'r'};
char s2[10] = {'m', 'o', 'u', 's', 'e'};
char s3[][5] = {{'b', 'o', 'o', 'k'}, {'b', 'o', 'o', 'k', '2'}};
```

字符数组 s1 中的 8 个字符元素都进行初始化, 而且最后一个元素不是 0, 那么 s1 不能作为字符串来处理, 只能作为字符数组。这是因为每个字符串都要用 0 值来结尾(注意不是字符 '0')。字符数组 s2 就被初始化为一个字符串。同理, s3[0]是一个字符串, 而 s3[1]就不是。一个字符数组的当前值是否字符串, 决定了对它的处理方式。

### 2.用字符串进行初始化

采用字符串字面值来初始化字符数组, 其语法格式为:

```
char <数组名>[<常量表达式>] = <字符串常量>;
```

例如:

```
char s11[] = "Computer", s22[] = "mouse";
char s33[][5] = {"one", ""};
```

表 6.3 列出了上述 6 个数组初始化后各元素对应的数据值。

表 6-3 数组初始化后各元素的值

数组元素	数据值	数组元素	数据值	数组元素	数据值	数组元素	数据值	数组元素	数据值	数组元素	数据值
s1[0]	C	s2[0]	m	s3[0][0]	b	s11[0]	C	s22[0]	m	s33[0][0]	o
s1[1]	o	s2[1]	o	s3[0][1]	o	s11[1]	o	s22[1]	o	s33[0][1]	n

s1[2]	m	s2[2]	u	s3[0][2]	o	s11[2]	m	s22[2]	u	s33[0][2]	e
s1[3]	p	s2[3]	s	s3[0][3]	k	s11[3]	p	s22[3]	s	s33[0][3]	\0
s1[4]	u	s2[4]	e	s3[0][4]	\0	s11[4]	u	s22[4]	e	s33[0][4]	\0
s1[5]	t	s2[5]	\0	s3[1][0]	b	s11[5]	t	s22[5]	\0	s33[1][0]	\0
s1[6]	e	s2[6]	\0	s3[1][1]	o	s11[6]	e			s33[1][1]	\0
s1[7]	r	s2[7]	\0	s3[1][2]	o	s11[7]	r			s33[1][2]	\0
		s2[8]	\0	s3[1][3]	k	s11[8]	\0			s33[1][3]	\0
		s2[9]	\0	s3[1][4]	2					s33[1][4]	\0

字符 `char` 数组可表示中英文混合串, 按 GBK 编码每个中文字符占 2 个字节, 例如:

```
char name[] = "28 研究所"; // 占用 9 字节
```

UTF-8 编码的字符串也可作为 `char` 窄串, 需要加前缀 `u8`:

```
char str1[] = u8"Hello World";
char str2[] = u8"☺" = \U0001F607 is O:-)";
```

对 `wchar_t` 数组的宽串初始化应添加前缀 `L`, 例如:

```
wchar_t b[] = L"size"; // 占用 10 字节
wchar_t c[] = L"28 研究所"; // 占用 12 字节
wchar_t d[] = L""; // 占用 2 字节
```

对 `char16_t` 和 `char32_t` 数组的宽串的初始化应分别添加 `u` 和 `U` 前缀:

```
char16_t strr1[] = u"hello";
char32_t strr2[] = U"hello";
```

C11 开始支持 `raw` 串, 也称为毛串或原始串, 其中可包含任意字符, 包括双引号"、反斜杠\、换行符, 其中不需要转义符。毛串常用于 HTML、XML 串。毛串用 `R` 前缀:

```
char raw_narrow [] = R"(An unescaped \ character)";
wchar_t raw_wide[] = LR"(An unescaped \ character)";
char raw_utf8[] = u8R"(An unescaped \ character)";
char16_t raw_utf16[] = uR"(An unescaped \ character)";
char32_t raw_utf32[] = UR"(An unescaped \ character)";
```

两个字面值可拼接起来, 有 2 种等价写法, 例如:

```
char strp1[] = "12" "34";
char strp2[] = "12"\
    "34"; // 反斜杠也称为续行符
```

串中若含有转义符总应特别小心, 例如:

```
char strp3[] = "\x05five";
```

并非你所期望的 5 个字符, 而是 4 个, 原因是 `'\x05f'` 被当做 1 个字符。因此下面拼接书写形式可能更安全:

```
char strp4[] = "\x05" "five"
```

关于字符串初始化, 说明以下几点。

(1) 用字符串初始化时, 系统在字符串末尾自动加上一个 0 值, 作为字符串的结尾标志。因此字符数组长度至少要比字符串中的字符个数大 1。

(2) 系统能根据字符串的长度(即字符个数)来确定数组的大小。

(3) 用字符串初始化一维字符数组时, 大多省略花括号。

(4) 除了毛串, 其它字符串中都可带有转义字符(见 2.3.4 节)。

(5) 不含任何字符的字符串""称为空串, 尾 0 也要占 1 个字符。

(6) 除了用字符数组定义字符串, 字符串还可用字符指针来定义, 或类型(如 `std::string`)来定义。后面详细介绍。

### 6.4.3 字符数组的输入输出

字符数组的输入输出可以逐个元素处理, 也可以按字符串的方式处理。

#### 1. 逐个元素处理

**方式 1。**用 `cin>>`和 `cout<<`。例如下面代码:

```
char s1[5];
for(int i = 0; i < 5; i++)
    cin >> s1[i];           //A 设执行时输入:ab c d efgh
for(int i = 0; i < 5; i++)
    cout << s1[i];         //输出:abcde
```

A 行用 `cin>>`在输入单个字符时, 将跳过空格、制表符和换行符, 因此用 `cout>>`输出时就没有这些分割符。A 行程序在执行时可以多输入字符, 用回车结束, 超过的字符被暂存在输入缓冲区中, 可能被下面其它输入 `cin` 指令误读。

这种方式可以输入输出中英文混合串, 但实际上比较少用。

**方式 2。**用 `cin.get` 和 `cout.put` 函数。例如:

```
char s2[5];
for(int i = 0; i < 5; i++)
    cin.get(s2[i]);         //A 设执行时输入:a b cdef
for(int i = 0; i < 5; i++)
    cout.put(s2[i]);        //则执行时输出:a b c
```

A 行用 `cin.get` 输入时, 可以读取空格符、制表符和换行符, 因此下面调用 `cout.put` 函数输出时能看到输出这些分割符。A 行在执行时也可以多输入字符。

这种方式也可输入输出中英文混合串。

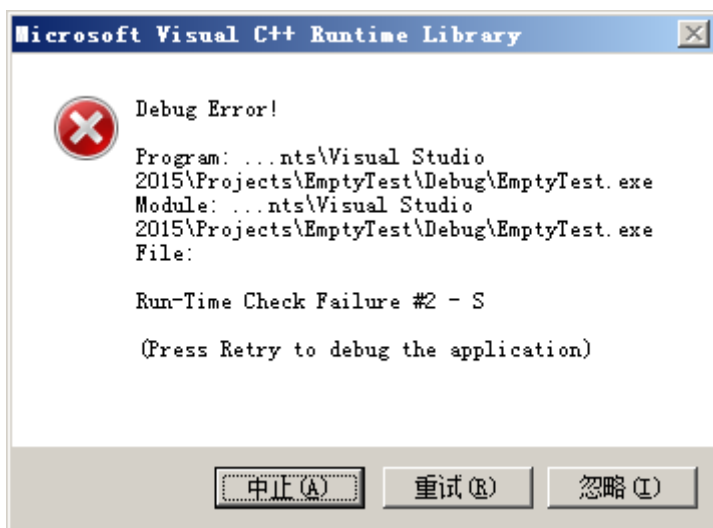
#### 2. 按字符串处理

将一个字符数组作为一个串来输入输出。例如:

```
char s3[5];
cin>>s3;                     //A 设输入 ab cdefgh
cout<<s3;                    //输出:ab
```

A 行的 `cin>>`从输入流中连续读取字符, 直到遇见空格、制表符或换行符时结束。如果输入字符数超过了数组大小, 就会侵占相邻内存, 可能导致严重错误而被迫退出。所以输入分割符之前的字符数应小于长度。例如, 输入“abcd”是安全的, 但如果输入超过 4 个字符要出错, 往往程序中止, 并给出下面提示。





虽然将数组长度加大可以提高安全性，但也不可靠。

比较安全的串输入是调用 `cin.getline` 函数。该函数也能输入含空格、制表符的字符串。例如下面代码：

```
char s4[5];
cin.getline(s4, 5);           //A 设输入 a bcdef
cout<<s4<<endl;             //输出: a bc
```

A 行的 `cin.getline` 函数从输入流中读取一行字符作为一个字符串，直到读到换行符为止。该函数的第一个形参为字符数组，第二个形参确定了至多可读取的字符个数。读取的一行字符中可包括空格、制表符。在第 14 章中将详细介绍字符串的其它输入输出方法。

用 `cout` 输出字符串时，并不关心字符数组的大小，而是连续输出字符，直到遇见尾 0 为止。

### 3. 宽字符串处理

对于 `wchar_t` 数组的输出应采用 `wcout`，如果包含中文，还需要添加本地化控制，例如：

```
wchar_t c[] = L"28 研究所";
wcout.imbue(locale("chs"));    //说明按中文简体字符集输出
wcout << c << endl;
```

其中 `imbue` 函数用于设置本地区域，以确定 `cout` 输出的字符集。如果没有设置或设置错误都会导致仅能输出英文字符，而其它字符不能输出。

对于 `wchar_t` 数组的输入应采用 `wcin`，如果包括中文，还需要添加本地化控制，例如：

```
wchar_t ee[20];
wcin.imbue(locale("chs"));    //说明按中文简体字符集输入
wcin >> ee;
```

以这种方式输入如果超过 19 个字符就会导致溢出错误，建议采用 `wcin.getline` 输入：

```
wchar_t s4[5];
wcin.imbue(locale("chs"));
wcin.getline(s4, 5);
```

宽串输入输出仅适用于 VS2015，DevC++(GCC)不适用。

#### 6.4.4 字符数组的操作

字符数组的操作与前面整数数组的操作，基本形式是一样的，都是用下标指定元素进行读取或写入，以实现各种转换或计算。

应注意区别被处理的字符数组是否为字符串。如果是字符串，就有尾 0，就能用函数计算其元素个数，而不需要另外指定元素个数。

应注意窄串与宽串在处理中英文混合串的共性与区别。窄串中一个英文字符占 1 字节，一个中文字符占相邻 2 字节(称为多字节编码)。因此窄串 `str[i]` 指的是第 `i` 个字节，也是第 `i` 个元素。而宽串(如 `wchar_t`)中任何一个中英文字符都占 2 字节，包括尾 0。因此宽串 `wstr[i]` 指的是第 `i` 个元素或第 `i` 个字符，而不是第 `i` 个字节。

下面代码段将一个表示星期几的从 0 到 6 的整数 `week`，转换为符合中文习惯的“星期日”到“星期六”的输出，采用了宽串处理，只能在 VS2015 上运行：

```
wchar_t weekday[] = L"日一二三四五六";  
wcout.imbue(locale("chs"));  
wcout << L"星期"<<weekday[week] << endl;
```

访问任何数组的元素，下标越界不仅导致溢出问题，还可能导致更严重的安全性问题。

C11 提出带边界检查(bound-checking)的一组安全函数，每个安全函数名都有 `s` 后缀，如 `strcpy_s`，`strcat_s` 等。VS2015 在编译传统调用(如 `strcpy`)时报错，同时建议调用安全拷贝函数 `strcpy_s`，如果你坚持用老版本，就要先定义宏 `_CRT_SECURE_NO_WARNINGS`。对传统的 `scanf` 函数调用也报错，建议改为 `scanf_s` 调用。新的函数虽不能阻止溢出，但能及时中止程序以避免更严重的安全性问题。

### 6.5 字符串处理函数

C++ 系统提供了许多处理字符串的函数，在头文件 `<string>` 中给出了这些函数的原型说明。本节只介绍几个常用的字符串处理函数，其它函数读者可参看相关文档。

#### 6.5.1 字符串处理函数

一部分函数执行没有副作用，例如求字符串的长度、比较两个字符串。这些函数调用是安全的。但也有一部分函数具有副作用，会改变字符串的长度或者元素内容，例如字符串拷贝、拼接。调用这些函数时应注意实参数组的大小是否足够大。

一部分函数要返回一个生成的字符串作为结果，但因函数不能返回数组，所以往往返回一个指针，指向结果数组的首地址。你可以直接用 `cout` 输出。指针将在第 8 章介绍。

##### 1. 字符串长度函数

函数原型为：`int strlen(const char s[]);`

其中，形参 `s` 是字符数组(或字符串常量)，该函数返回 `s` 中字符串中的字符个数，即字符串的长度。字符串的尾 0 并不计算在长度内。例如：

```
char s1[10] = "flower";  
cout<<strlen("watch")<<endl;           //输出值为 5
```

```
cout<<strlen(s1)<<endl;           //输出值为 6
cout<<sizeof(s1)<<endl;           //输出值为 10
```

VS2015 对于 `strlen` 提供了安全函数 `strnlen_s`。

## 2. 字符串拷贝函数

函数原型为: `char * strcpy(char to[], const char from[]);`

其中,形参 `to` 为接收字符串的数组。如果该数组原先有内容的话,就会被覆盖。形参 `from` 可以是字符数组或字符串常量。该函数将 `from` 中存放的字符串或字符串常量拷贝到字符数组 `to` 中。该函数返回一个指针指向 `to` 字符串。注意,`to` 数组长度不小于 `from` 的长度。例如:

```
char str1[10]="flower", str2[100], str3[20], str4[6];
strcpy(str2, str1);           //将字符串"flower"拷贝到 str2 中
cout<<str2<<endl;           //输出为:flower
cout<<strcpy(str3, "南京");   //将"南京"拷贝到 str3 中,再输出
```

数组之间不能赋值,如 `str2=str1` 是错误的。可用 `strcpy` 函数来实现字符串复制。但如果 `to` 数组的长度不够的话,下标就会越界而导致缓冲区溢出,从而导致更严重的安全问题。例如, `strcpy(str4, str1)` 就是危险的,因 `str1` 中的字符串 "flower" 占用了 7 个字节空间,而 `str4` 只有 6 个字节。

如果要限制最多拷贝的字符个数,有函数原型如下:

```
char * strncpy(char to[], const char from[], int size);
```

此函数最后一个形参 `size` 限制最多拷贝多少字符,应不大于 `to` 数组的大小。

VS2015 对于 `strcpy` 提供安全函数 `strcpy_s`。

## 3. 字符串拼接函数

函数原型为:

```
char * strcat(char s1[], const char s2[]);
```

其中,形参 `s1` 是一个字符数组,形参 `s2` 可以是一个字符数组或一个字符串常量。该函数将 `s2` 字符串拷贝到 `s1` 字符串之后,使 `s1` 成为一个拼接串,并返回指针指向 `s1` 串。例如:

```
char s2[10] = "day",s1[20] = "week";
strcat(s1, s2);           //构成新串"weekday"
cout<<s1<<endl;           //输出为: weekday
```

显然,应确保数组 `s1` 长度足够大,能存放拼接后的字符串,否则会导致错误。

可以利用函数的返回值进行连续计算,例如:

```
char str3[40] = "南京";
cout<<strcat(strcat(str3, "理工大学"), "计算机学院");
```

如果要限制最多拷贝的字符个数,有函数原型如下:

```
char * strncat(char s1[], const char s2[], int size);
```

此函数最后一个形参 `size` 限制最多拷贝的字符个数。

处于安全性考虑,VS2015 建议调用 `strcat_s` 和 `strncat_s`。

## 4. 字符串比较函数

函数原型为:

```
int strcmp(const char s1[], const char s2[]);
```

其中,形参 `s1` 和 `s2` 都可以是字符数组或字符串常量。该函数比较 `s1` 和 `s2` 的这两个字符串的大小,即按从前到后的顺序逐个比较对应字符的 ASCII 码值。若 `s1` 中的字符串大于 `s2`

中的字符串，则返回值大于 0(如 1)；若两字符串相等，则返回 0；否则返回值小于 0(如-1)。例如：

```
char s1[10] = "week", s2[20] = "day";
cout<<strcmp(s1, s2)<<endl;           //输出结果为 1
cout<<strcmp(s2, s1)<<endl;           //输出结果为-1
cout<<strcmp(s1, "week")<<endl;       //输出结果为 0
```

以下函数功能与库函数 `strcmp` 完全相同：

```
int strcmp(char s1[], char s2[]){
    int i = 0;
    while(s1[i] == s2[i] && s1[i] != 0)
        i++;
    if(s1[i] < s2[i])
        return -1;
    if(s1[i] > s2[i])
        return 1;
    return 0;
}
```

如果在比较时要忽略字母大小写，可以使用下面函数：

```
int _stricmp(const char s1[], const char s2[]);
```

如果要限制最多比较的字符个数，可使用下面函数：

```
int strncmp(const char s1[], const char s2[], int size);
```

此函数最后一个形参 `size` 确定了仅比较前面 `size` 个字符。

宽字符串的处理与窄串处理相对应，如下表所示。

表 6-4 窄串与宽串分别调用不同处理函数

串处理功能	窄串处理函数	宽串处理函数
求串长，字符数	<code>strlen</code> , <code>strnlen_s</code>	<code>wcslen</code>
串拷贝	<code>strcpy_s</code>	<code>wcscpy_s</code>
串拼接	<code>strcat_s</code>	<code>wcscat_s</code>
串比较	<code>strcmp</code>	<code>wcscmp</code>
串逆序	<code>strrev</code>	<code>wcsrev</code>

除了上面所介绍的几个函数之外，系统还提供了很多关于字符和字符串处理的函数。附录 B 列出了一部分常用窄串处理库函数，读者也可参看 MSDN 文档。

在使用这些库函数时，应注意以下几点：

(1)在库函数中字符数组作为形参，一般用字符指针来表示字符数组。例如：

```
int strlen(const char *string);
```

(2)在调用这些库函数时，要保证数组应具有足够大小，如果要纳入的串长度超过了数组的大小，将导致溢出中止。

(3)在调用这些库函数时，要确保调用实参不是空指针 `NULL`，否则结果难料。

C++编程应尽可能采用 C++标准库，而不是 C 函数库，虽然后者运行效率高，但内存安全需要太多用心，使程序员不能全心去解决实际问题。如果真的需要字符串，而且 C++标准库可用，就应避免采用字符数组来表示字符串。C++标准模板库<string>中提供了 `string` 和

wstring 类型, 采用动态内存实现任意长的串, 而且提供了一组安全的操作函数, 常作为 C++ 编程的标准字符串类型。

### 6.5.2 字符数组的应用

例 6-9 对于一个中英文字符串, 计算各个字符出现的次数, 再按出现次数降序排序。

宽字符串 wchar\_t 的数组非常适合描述中英文串, 每个元素都是一个字符,

设计方法是先计算各字符出现次数, 并保存字符与次数; 再采用冒泡排序算法按次数降序排序; 最后输出结果。编程如下:

```
#include<iostream>
using namespace std;
#define LEN 20
int main(){
    wchar_t str[LEN] = L"知之为知之,不知为不知,是知也";
    wchar_t set[LEN];
    int n = wcslen(str), count = 0, counts[LEN];
    //先计算各字符出现次数
    for(int i = 0; i < n; i++){
        wchar_t a = str[i];
        int j;
        for(j = 0; j < count; j++){
            if(a == set[j]){
                counts[j]++;
                break;
            }
        }
        if(j >= count){
            set[count] = a;
            counts[count++] = 1;
        }
    }
    //按出现次数对各个字符降序排序, 冒泡排序
    for(int i = 0; i < count - 1; i++){
        for(int j = 0; j < count - 1 - i; j++){
            if(counts[j+1] > counts[j]){
                wchar_t t = set[j];
                set[j] = set[j+1];
                set[j+1] = t;
                int tc = counts[j];
                counts[j] = counts[j+1];
                counts[j+1] = tc;
            }
        }
    }
    //输出结果
    cout << "串长:" << n << ", 共有" << count << "个中英文字符" << endl;
    wcout.imbue(locale("chs"));
    for(int k = 0; k < count; k++){
        wcout<<set[k]<<":"<<counts[k]<<endl;
    }
    system("pause");
    return 0;
}
```

执行程序, 显示结果如下:

```
串长: 15, 共有 7 个中英文字符
知:5
之:2
为:2
```

```
,:2
不:2
是:1
也:1
```

例 6-10 在一组中国人姓名中查找属于某个姓氏的全部姓名。

一个姓名作为一个字符数组，那么一组姓名就可以用字符的二维数组来存放。由于中国人的姓名结构是姓在前名在后，所以这种查找本质上就是判断姓名串中前几个字符。设计一个函数专门判断一个字符串开头是一个特定子串，原型如下。

```
bool strStartWith(wchar_t s1[], char_t s2[]);
```

第一个形参 s1 是全串，第二个形参 s2 是要判断的子串。如果 s1 开头子串是 s2，就返回 true，否则就返回 false。编程如下：

```
#include <iostream>
#include <string.h>
using namespace std;
bool strStartWith(const wchar_t s1[], const wchar_t s2[]){
    if (wcslen(s1) == 0 || wcslen(s2) == 0)
        return false;
    if (wcsncmp(s1, s2, wcslen(s2)) == 0)
        return true;
    return false;
}
int main(){
    wchar_t names[][20] = {                                //存放若干姓名
        L"张三",L"李四",L"王五",L"赵六",L"张小三"
    };

    wchar_t surname[20];
    cout<<"输入一个姓氏:";
    wcin.imbue(locale("chs"));
    wcout.imbue(locale("chs"));
    wcin.getline(surname, sizeof(surname));
    int num = sizeof(names) / 20;                          //计算已有姓名个数
    int count = 0;
    for (int i = 0; i < num; i++){                          //循环每一个姓名
        if (strStartWith(names[i], surname)){
            wcout<<"i="<<i<<"："<<names[i]<<endl;
            count++;
        }
    }
    if (count == 0)
        cout<<"没找到"<<endl;
    system("pause");
    return 0;
}
```

如果有 50 万个姓名，最后 for 就要循环 50 万次，才能完成查找。但如果姓名是按字典顺序存放，那么查找效率就可能提高，前面介绍的折半查找就能起作用。

## 6.6 小 结

- 一个数组是由相同类型的一组变量组成的一个有序集合。
- 数组中的每个变量称为一个元素，所有元素共用一个变量名，就是数组的名字。
- 数组中的每个元素都有一个序号，称为下标。访问一个元素可用数组名加下标来实现。
- 定义一个数组有 3 个要素：元素类型、名称与长度(即大小)。一个数组占用一块连续存储空间，字节大小为  $n * \text{sizeof}(\text{元素类型})$ ，其中  $n$  为数组的长度。一个数组的长度必须在定义时确定，而且在执行过程中不能改变。
- 通过表达式下标访问数组元素时，注意下标越界。
- 两个数组之间不能直接赋值，也不宜使用关系运算符。
- 数组具有多种实际用途，如多项式计算、排序(如选择排序、冒泡排序等)、查找(如二分查找)等。
- 二维数组就是具有两个下标的数组，确定一个元素需要两个下标值。二维数组经常用来表示矩阵或者二维表，用行下标和列下标。
- 由于二维数组可被看作一维数组，故此二维数组也称为数组的数组。定义二维数组时，如果对所有元素初始化，可以不指定该数组的行数，由系统自动计算。
- 二维数组可表示矩阵，可完成矩阵的转置、加法、乘法等计算。
- 数组可作为函数的形参，一维数组做形参往往需要另一个形参确定数组的大小，二维数组做形参必须确定列下标的长度。
- C++语言没有直接提供语言级的字符串类型。要通过字符数组或字符指针来定义字符串。字符数组就是 `char` 类型的元素组成的数组。字符串是用 0 值结尾的字符数组。一个字符数组可用一个字符串字面值进行初始化。
- 键盘输入一个字符串最好使用 `cin.getline` 函数，需要确定最多输入的字符个数，比较安全。
- 两个字符串之间不能用赋值语句进行赋值，但可用 `strcpy` 函数进行拷贝。
- 两个字符串之间不能用关系运算符进行比较，但可用 `strcmp` 函数进行比较。
- 函数 `strlen` 可以求长度，函数 `strcat` 可以拼接两个串。
- 数组作为函数形参，如果形参前有 `const` 修饰，那么该数组在函数中不能被改变该数组的元素，只能读取元素。反之，如果没有 `const` 修饰，函数中就可以改变数组元素作为结果。
- 充分利用这些字符串处理函数可以提高编程求解能力。

## 6.7 练 习 题

1. 下面哪一个数组说明语句是错误的？

A) `int a[]={1,2};`

B) `char a[3];`

- C) `char s[10]="test";`     D) `int n = 5, a[n];`
2. 假设在函数中有说明语句: `int a[10]={9,1,6,0,8};` 数组元素 `a[5]` 的值是\_\_\_\_\_。
- A 8                      B 0                      C 6                      D 随机值
3. 下面哪一个数组说明语句是错误的?
- A `int b[][3]={0,1,2,3};`     B `int d[3][ ]={{1,2},{1,2,3},{1,2,3,4}};`  
C `int c[100][100]={0};`     D `int a[2][3];`
4. 有语句: `int b[][3]={{9},{1,6},{0,8},{1,2,3}};` 数组元素 `b[3][2]` 的值是\_\_\_\_\_。
- A 3                      B 8                      C 6                      D 9
5. 下面哪一个数组说明语句是正确的?
- A `char s4[2][3]={"xyz","abc"};`     B `char s1[]="xyz";`  
C `char s3[][]={'x','y','z'};`     D `char s2[3]="xyz";`
6. 下面哪一个数组说明语句是错误的?
- A `char s4[]="Ctest\n";`     B `char s3[20]="Ctest";`  
C `char s2[]={'C', 't', 'e', 's', 't'};`     D `char s1[10];s1="Ctest";`
7. 设有说明语句: `char s[5][5]={"abc","efgh"};` 值为字符 `g` 的数组元素是\_\_\_\_\_。
- A `s[4][4]`                      B `s[1][4]`                      C `s[1][3]`                      D `s[1][2]`
8. 设有说明语句: `char s[10];` 对 `s` 的数组元素正确引用是\_\_\_\_\_。
- A `s(6)`                      B `s[10]`                      C `s[6+1]`                      D `s[1.5]`
9. 执行以下代码段后, `t` 的值为\_\_\_\_\_。
- ```
int b[3][3]={0,1,2,0,1,2,0,1,2},i,j,t=1;
for(i=0;i<3;i++)
    for(j=i;j<=i;j++)
        t=t + b[i][i] + b[j][j];
```
- A 7                      B 9                      C 4                      D 3
10. 对于一个函数 `void sort(int a[], int n=10);`  
设有语句: `int b[7];` 下面哪一个函数调用能正确执行?
- A `f(b);`                      B `f(b, 9);`  
C `f(b, sizeof(b)/sizeof(int))`                      D `b = f(b, 7);`
11. 下列程序的输出结果是\_\_\_\_\_。
- ```
#include <iostream.h>
void main(void){
    int i, k, a[10], p[3];
    k=5;
    for (i=0; i<10; i++) a[i]=i;
    for (i=0; i<3; i++) p[i] = a[i *(i+1)];
    for (i=0; i<3; i++) k+=p[i] * 2;
    cout<<k<<endl;
}
```
- A 21                      B 22                      C 23                      D 24



12. 下列程序的输出结果是\_\_\_\_\_。

```
#include <iostream.h>

void main(void){
    char w[][10]={"ABCD","EFGH","IJKL","MNOP"}, k;
    for(k=1; k<3; k++) cout<<w[k];
    cout<<endl;
}
```

A) BCD                      B) EFGH                      C) IJKL                      D) EFGH IJKL

13. 根据题目要求, 编写完整程序加以验证, 可调用已有函数, 也可添加其它函数。

(1)编写一个函数: `bool isSorted(const int a[], int n)`, 判断数组 `a` 中的元素是否按升序排列。

(2)编写一个函数: `void getRandom(int a[], int n)`, 生成 0 到 100 之间的随机整数作为数组元素。再编写一个函数 `int getMax(const int a[], int n)`, 在数组 `a` 中找出最大值并返回下标。再编写一个排序函数, 将 `int` 数组进行升序排序, 运行程序并验证正确性。

(3)编写一个函数: `bool insertSort(int a[], int maxnum, int n, int x)`, 形参 `a` 是一个数组, 形参 `maxnum` 是数组 `a` 的大小, 即能容纳最多元素个数, `n` 是当前已插入的元素个数, 形参 `x` 是要插入的一个值。该函数尝试将 `x` 插入到数组 `a` 中的合适位置, 使所有元素保持升序排列。如果成功插入, 返回 `true`。如果没有位置插入(即已满), 就返回 `false`。验证该函数的正确性。

(4)编写一个函数: `void trim(char a[])`, 将字符串 `a` 的前后的所有空格过滤掉, 注意中间的空格要保留。例如, 字符串 " ab c " 的过滤结果是 "abc"。

(5)编写一个函数: `void replace(char rs[], const char a[], const char s1[], const char s2[])`, 在字符串 `a` 中查找是否有 `s1` 子串, 如果有, 就将 `s1` 子串替换为 `s2` 串, 最后将结果串写入到形参 `rs` 串中。

(6)编写一个函数: `bool isSmith(int n)`, 判断一个正整数 `n` 是否为 Smith 数。

Smith 数的概念: 一个非素数, 其各位数之和等于其所有质因数的各位数之和。例如:

$4 = 2 * 2$ ,  $4 = 2 + 2$ , 所以 4 就是一个 Smith 数。

$22 = 2 * 11$ ,  $2 + 2 = 2 + 1 + 1$ , 22 也是一个 Smith 数。

$27 = 3 * 3 * 3$ ,  $2 + 7 = 3 + 3 + 3$ , 27 也是一个 Smith 数。

尝试计算大于等于 4937774 的下一个 Smith 数。

(7)先编写一个函数: `int getRev(char a[])`, 计算并返回字符串 `a` 的逆序。

逆序的概念: 在一个字符串中, 如果存在  $i < j$ , 且  $a[i] > a[j]$ , 则称  $a[i]$  和  $a[j]$  构成一个逆序。

例如 "DAABEC" 的逆序是 5, 其中 D 与 A、A、B、C 构成 4 个逆序, E 与 C 构成 1 个逆序。

要求任意输入 MAX 个字符串 (MAX 是一个宏, 值为 >2 的正整数), 每个串不多于 20 个字符, 先计算各串的逆序数, 再按逆序数升序输出各串及其逆序数。

(8) 一个集合 `set` 中的各个元素相互之间不相等。先编写一个函数: `int getSet(int rs[], const int a[], int n)`, 从数组 `a` 中取出相互不等的元素, 放入数组 `rs` 中, 并返回 `rs` 中元素的个数。此时数组 `rs` 中各元素都不相等, 就构成一个集合 `set`。例如: `a={3, 1, 2, 3, 1, 5, 2, 1}`, 那么结果 `rs={3, 1, 2, 5}`, 返回 4。先测试验证该函数的正确性。

然后再尝试设计一个函数, 不仅能得到集合数组, 而且得到集合中各元素出现的次数。例如上面例子中

集合 `rs={3, 1, 2, 5}`，其中各元素出现次数分别为{2, 3, 2, 1}。

再设计一个函数，将集合中的各元素按出现次数降序排序，最后输出各个元素及其出现次数。例如，上面例子输出结果如下：显示格式为“元素值：出现次数”。

```
1:3
3:2
2:2
5:1
```

(9) 对一个数组 `float a[5][6]`，求出每行中的最大值及其行、列下标。

(10) 编写一个函数：`int getWordCount(char a[])`，统计字符串 `a` 中的单词个数，单词之间用一个或多个空格或 `tab` 符隔开。

(11) 做一个小游戏，21 个人围成一个圈，编号依次为 1~21。从第 1 号开始报数，报到 5 的倍数的人离开，一直报下去直到最后只剩下 1 人，求出此人的编号。

## 第7章 结构、枚举、共同体

基本数据类型（布尔型、字符型、整数型、浮点型）以及这些类型的数组类型，都是 C/C++ 语言中预定义的数据类型，在编程中可直接定义这些类型的变量。但是这些类型仍不足以描述更加复杂的变量。本章介绍的结构、枚举、共同体都是用户自定义类型。用户要先定义这些类型，再定义该类型的变量，然后再操作这些变量来完成计算。

### 7.1 结构

在编程中往往要将一组数据聚集起来表示一个实例。例如要表示一名学生就要表示其学号、姓名、性别、成绩等属性。表示一个日期就需要描述年、月、日这 3 个属性。同样表示时间需要时、分、秒。表示一个分数，就需要一个整数分子和一个非零整数的分母。如果把这些数据都作为彼此独立的变量，就难以反映它们之间的关系。因此需要将这些数据聚集起来组成一个整体，这样就需要结构类型。

所谓结构(structure，简称为 struct，也称为结构体)就是一种自定义类型。一个结构有一个类型名字，包含一组成员。结构的一个变量中，每个成员都持有自己的值。结构类型可定义数组，结构中也能包含静态成员。虽然 C++ 扩展结构成为与类 class 差不多的类型，如继承性、多态性、模板等，但本文先介绍传统 C 语言的结构。

#### 7.1.1 结构类型的定义

定义一个结构类型要用一条结构说明语句，格式如下：

```
struct <结构类型名>{  
    <成员类型 1> <成员名 1>;  
    <成员类型 2> <成员名 2>;  
    ...  
    <成员类型 n> <成员名 n>;  
};
```

其中，struct 是定义结构类型的关键字。<结构体类型名>是用户命名的标识符。花括号内的部分称为结构体，由若干成员说明组成。每个成员有自己的数据类型和名称，成员名是用户自己定义的标识符。成员的数据类型可以是基本数据类型以及基本类型的数组，也可以是已用户自定义类型，以及这些类型的数组。若几个成员具有相同数据类型，各成员名之间用逗号隔开，就像定义同一类型的多个变量一样。

## 1. 成员定义与初始化

C14 允许对非静态成员添加初始化。例如：

```
struct mystruct {
    int a = 7;
    bool b = false;
    char name[20] = "";
    decltype(3.4) c = 3.4; //double
}
```

成员初始化与变量初始化形式相同，只是不能用 `auto` 来自动推导成员类型，但可用 `decltype` 来推导成员类型。注意，如果采用这种初始化，将不能用花括号对其结构变量做初始化，除非提供构造函数。

在一个结构类型中各个成员的名字不能重复，因为结构为其各成员标识符提供了一个作用域。

从逻辑上看，各个成员之间是无序的，这是因为访问各个成员是按名访问的。但从数据存储来看，一个结构变量的各成员是按说明的次序来存储的。

定义一个结构类型是一条完整的说明语句，所有成员要用一对花括号括起来，最后用分号结束。

下面定义一个结构类型来表示学生的各种信息：

```
struct Student{
    char num[10];           //学号
    char name[20];         //姓名
    char sex;               //性别，用'f'表示女性，'m'表示男性
    float score[5];        //5 门课程的成绩，应确定各下标所对应的课程
};
```

上面定义了一个结构类型 `Student`，其中包含 4 个成员，每个成员都说明了它的类型和名字。

一些结构类型具有通用性。例如日期包含年、月、日三个成员，就可以用一个结构类型来表示日期，例如：

```
struct Date{
    short year, month, day; //占 6 字节
};
```

定义一个结构类型时，可使用前面已定义的其它结构类型。比如，要表示学生的出生日期和入学日期，就可以在上面 `Student` 结构类型中添加两个新成员：

```
struct Student{
    ...
    Date birthdate;           //出生日期
    Date enrolldate;         //入学日期
};
```

例如，下面 `Employee` 是描述职员的结构类型：

```
struct Employee{
    char num[5], name[20];    //职员编号和姓名，占 25 字节
    char sex;                 //性别，占 1 字节
};
```

```
Date birthday;           //出生日期, 占 6 字节
Date employdate;         //受雇日期, 占 6 字节
};
```

结构中的成员的类型能否是自己结构类型? 例如, 职员经理也是一名职员:

```
struct Employee{
    ...
    Employee manager;
};
```

编译上面的结构类型会报错, 不允许结构类型作为其成员的类型, 但允许结构的指针作为其成员的类型。在下一章将介绍指针。

一个结构类型应该定义在什么地方? 多数结构类型定义在函数的外边, 使所有函数能用它作为形参或返回类型。一个结构类型也可定义在某个函数体中, 只能在此函数内使用, 这种情形不多见。

## 2. 结构大小与成员对齐

如何计算一种结构类型的存储字节大小? 每一种结构类型都具有确定的存储大小。使用 `sizeof(类型名)` 可以知道该类型的字节大小。例如 `sizeof(Date)` 为 6, `sizeof(Employee)` 为 38, 即各个成员字节大小之和。

如果一个结构中含有一个 `int` 或 `float` 类型的成员, 该类型的字节大小就会扩展到 4 的整数倍。例如, 在 `Employee` 结构类型中添加一个 `int` 成员:

```
struct Employee{
    char num[5], name[20];           //职员编号和姓名, 占 25 字节
    char sex;                         //性别, 占 1 字节
    Date birthday;                   //出生日期, 占 6 字节
    Date employdate;                 //受雇日期, 占 6 字节
    int salary;                       //工资, 占 4 字节, 新加成员
};
```

此时 `sizeof(Employee)` 为 44, 而不是  $38+4=42$ 。这时为了适应 32 位计算中, 以 4 字节为单位(称为字对齐)来访问内存能提高效率。

C11 提供一个运算符 `alignof(类型)` 得到一个整数, 说明该类型按多少字节对齐。比如 `alignof(Employee)` 的值为 4, 此时 `sizeof(Employee)` 应该是 4 字节的倍数。

如果一个结构中含有一个 `double` 类型的成员, 该类型的字节大小就会扩展到 8 的整数倍。例如, 在 `Employee` 结构类型中再添加一个 `double` 成员:

```
struct Employee{
    char num[5], name[20];           //职员编号和姓名, 占 25 字节
    char sex;                         //性别, 占 1 字节
    Date birthday;                   //出生日期, 占 6 字节
    Date employdate;                 //受雇日期, 占 6 字节
    int salary;                       //工资, 占 4 字节
    double highestSalary;             //最高工资, 占 8 字节, 新加成员
};
```

此时 `alignof(Employee)` 的值为 8, 此时 `sizeof(Employee)` 应该是 8 字节的倍数, 因此 `sizeof(Employee)` 为 56, 并非  $44+8=52$ 。56 是不小于 52 且最接近 52 的 8 的整数倍。结构的大小并不一定是成员大小之和。如果最大成员大小为 4 字节, 就自动向 4 字节对齐。如果最大

成员大小为 8 字节，就自动向 8 字节对齐。以上对齐存储方式是 VC 缺省编译配置。

C11 也提出一种人工设置对齐的修饰符 `alignas(整数)`，这个整数应该是 2 的幂，如 2、4、8 等，说明一个成员或一个结构按指定整数字节对齐。据称对于 64 位计算，按 8 字节对齐，当处理大批量结构数据时能极大提高计算效率，当然内存消耗也比较大。

### 7.1.2 定义结构变量

结构类型定义之后，就可创建该类型的变量（也称为实例），就像基本类型一样。格式为：

`[struct] <结构类型名> <变量名表>;`

其中，关键字 `struct` 可有可无；<结构类型名>必须在该说明语句之前定义。如果有多个变量，就用逗号隔开。例如，对于前面已定义的结构体类型 `Date`：

```
struct Student stu1;           //定义 Student 类型变量 stu1
Employee emp1, emp2;          //定义 Employee 类型变量 emp1 和 emp2
```

上面这种格式是最常见的，还有下面两种不太常用的格式也能定义结构变量。

(1) 定义结构类型的同时定义变量。格式为：

```
struct <结构体类型名> {
    <结构体成员表>
}<变量名表>;
```

其中，变量名表所列举的变量之间用逗号分开。例如：

```
struct Student{                //A
    char num[10], name[20];
    char sex;
    float score[5];
    Date birthdate;
    Date enrolldate;
}stu1, stu2;
```

这条说明语句既定义了结构类型 `Student`，也定义了该类型的两个变量。这个结构类型与这两个变量都属于相同的作用域中。如果出现在文件作用域中，这两个变量就是全局变量。如果出现在块作用域中，例如出现在一个函数体中，那么这两个变量就是局部变量。该类型只能在本函数的下面使用，而不能被其它函数使用。

(2) 匿名结构类型定义变量。定义结构类型的同时也定义了若干变量，只是结构类型没有命名，这样使下面程序不能再定义其变量。其格式为：

```
struct {
    <结构体成员表>
}<变量名表>;
```

这种方式只能一次性定义变量，以后不能再定义该结构类型的变量。例如：

```
struct{                          //B
    char num[10], name[20];
    char sex;
    float score[5];
    Date birthdate;
    Date enrolldate;
}stu3, students[10];
```

该语句定义了一个结构变量和一个结构数组变量。尽管 A 行和 B 行定义的结构体成员相

同，但系统认为变量 `stu1` 与 `stu3` 是不同类型的。用 `typeid(stu1).name()` 可看到 `stu1` 的类型名称为“`struct Student`”。用 `typeid(stu3).name()` 可看到 `stu1` 的类型名称为“`struct __unnamed`”。

### 7.1.3 结构变量的初始化

在定义一个结构变量时可初始化其各成员的值。初始化方式有两种：一是用花括号括起来若干字面值或变量，按成员说明的次序，对各成员进行初始化。二是用同类型的另一个变量来进行初始化，就是复制所有成员给新变量。例如下面代码：

```
struct Date{
    short year, month, day;
};
...
Date d1 = {2009, 3, 17};    //A
Date d2 = d1;
```

A 行列举成员值对 `d1` 中的各个成员变量赋初值，这种方式与给数组变量赋初值类同。列举的初始化数据值的个数可以小于结构成员个数，其余成员的缺省值取 0。B 行用同类型的变量 `d1` 对 `d2` 初始化，就是将变量 `d1` 的各成员值拷贝给 `d2`。

如果结构中的成员是数组或结构变量，就可以用嵌套的花括号。例如：

```
struct Student{
    char num[12], name[20];
    char sex;
    Date birthdate;
    Date enrolldate;
    float score[5];
};
...
Student s1 = {"0806559919", "张三", 'm',
              {1990, 5, 3}, {2008, 9, 3},
              {87, 83, 86, 92, 91}};
```

注意在列举各成员值时，可以仅对前面部分成员初始化，但不能用逗号跳过某个成员。

注意，从 VC6 到 VS2015 都不支持对结构成员的指定初始化(指定某成员赋予初始值)，尽管它是 C99 标准之一。

### 7.1.4 结构变量的使用

#### 1. 结构体变量的操作

同类型的两个结构变量之间可以赋值，就是复制所有成员值。

两个结构变量之间不能进行关系运算，即不能判等、大于、小于等计算。

不能用 `cin` 和 `cout` 对一个结构变量直接输入或输出，只能对其基本类型的成员进行输入和输出。

#### 2. 结构体成员的访问

要访问某个结构变量的某个成员，格式为：

<结构变量>.<成员名>

其中，“.”称为成员运算符，其作用是指定结构变量中的某个成员。成员运算符的优先级比较高(参见第2章)。例如下面代码段：

```
Student s1;
```

```
strcpy(s1.num, "0806559919");
strcpy(s1.name, "张三");
s1.sex = 'm';
s1.birthdate.year = 1990;           //嵌套的成员访问
s1.birthdate.month = 5;
s1.birthdate.day = 3;
```

如果一个成员是结构类型或数组，就不能像初始化那样为它赋值。例如下面赋值语句是错误的：

```
s1.enrolldate = {2008, 9, 3};       //错误
s1.score = {87, 83, 86, 92, 91};    //错误
```

如果一个成员是基本类型，就可按该类型要求进行操作。

结构类型可作为函数的形参或返回值。实参变量将赋值给形参，此时要复制所有成员，然后开始执行函数。当函数返回一个结构变量时，也要通过一次赋值来得到结果。这种形式计算性能比较低，很少用。结构类型的指针或引用作为函数的形参和返回值是比较常见的。下一章将介绍指针和引用。

例 7-1 输入一名职员的编号、姓名、性别、出生日期和工资，然后输出。编程如下：

```
#include <iostream>
#include <iomanip>
#include <string.h>
using namespace std;
struct Date{
    short year,month,day;
};
struct Employee{
    char num[5],name[20];
    char sex;
    Date birthday;
    float salary;
};
Employee inputAnEmp(){                //结构类型作为函数的返回值
    Employee emp;
    char sex[5];
    cout<<"输入编号 姓名 性别 出生年 月 日 工资(空格分隔)"<<endl;
    cin>>emp.num>>emp.name;
    cin>>sex;
    if (strcmp(sex, "男") == 0 || strcmp(sex, "m") == 0)
        emp.sex = 'm';
    else
        if (strcmp(sex, "女") == 0 || strcmp(sex, "f") == 0)
            emp.sex = 'f';
        else
            emp.sex = 0;
    cin>>emp.birthday.year>>emp.birthday.month;
    cin>>emp.birthday.day>>emp.salary;
    return emp;
}
void printAnDate(Date d){              //结构类型作为函数形参
    cout<<d.year<<". "<<d.month<<". "<<d.day;
}
void printTitle(){
    cout<<setw(5)<<"编号"<<setw(10)<<"姓名"<<setw(5)<<"性别";
    cout<<setw(13)<<"出生日期"<<setw(8)<<"工资"<<endl;
}
void printAnEmp(Employee emp){         //结构类型作为函数形参
```



```

    cout<<setw(5)<<emp.num<<setw(10)<<emp.name;
    cout<<setw(5)<<(emp.sex == 'f'? "女":"男");
    cout<<setw(5)<<' ';
    printAnDate(emp.birthday);
    cout<<setw(8)<<emp.salary<<endl;
}
int main(void){
    Employee e1 = {"024","李四",'f',{1977,8,3},4567};
    Employee e2 = inputAnEmp();          //A
    printTitle();
    printAnEmp(e1);
    printAnEmp(e2);
    system("pause");
    return 0;
}

```

执行程序，输入输出如下：

输入编号	姓名	性别	出生年	月	日	工资(空格分隔)
023	张三	男	1983	2	8	3456
编号	姓名	性别	出生日期		工资	
024	李四	女	1977.8.3		4567	
023	张三	男	1983.2.8		3456	

上面第2行是输入的数据。

函数 `inputAnEmp` 中定义了一个 `Employee` 变量，并用 `cin` 输入各成员的值，最后返回该变量。A 行用返回的变量来初始化变量 `e2`。函数 `printAnEmp` 的一个形参为 `Employee` 类型，接收一个 `Employee` 变量并用 `cout` 打印输出。

用函数形参来处理结构变量，最好是用指针或引用，这样能避免复制成员，效率更高，下一章将介绍。

### 7.1.5 结构的数组

一个结构类型不但可以定义单个变量，也能定义数组。结构类型的数组与基本类型的数组基本相同。

#### 1. 结构数组的定义及初始化

结构数组的定义与基本类型数组的定义格式一样。结构数组中的每个元素作为一个结构变量。初始化需要一个花括号，其中再嵌套各元素的值。例如：

```

struct Student{
    char num[12],name[20];
    char sex;
    float mathscore;
};
...
Student st[2] = {"001","Wangping",'f',84}, {"002","Zhaomin",'m',64}};

```

定义一个结构数组时，就为多个结构元素分配了存储空间。如果未列出后面元素的初始化，就用 0 来初始化，与普通数组一样。

#### 2. 结构数组的使用

先用数组的下标指定一个元素，再用成员运算符指定结构的成员。下面通过例子说明结构数组的用法。

例 7-2 对一组学生的一门课程成绩进行统计并输出。

已知一组学生的学号、姓名、性别、年龄和数学成绩，统计这些学生的成绩，不仅包括平均分，还要统计各分数段的人数及百分比。还要输出不及格学生名单。编程如下：

```
#include <iostream>
#include <iomanip>
using namespace std;
struct Student{
    char num[12],name[20];
    char sex;
    float mathscore;
};
struct GradeLevel{    //成绩统计结构类型
    float average;    //平均分
    int ac;            //90-100 分人数
    int bc;            //80-89 分人数
    int cc;            //70-79 分人数
    int dc;            //60-69 分人数
    int ec;            //0-59 分人数
};
void printGradeLevel(GradeLevel gl){
    int number = gl.ac + gl.bc + gl.cc + gl.dc + gl.ec;
    cout<<"统计数据: "<<endl;
    cout<<"考生数量:"<<number<<" 平均成绩: "<<gl.average<<endl;
    cout<<"90-100: " <<gl.ac<<"; 占"<<(float)gl.ac/number*100<<"%"<<endl;
    cout<<"80-89: " <<gl.bc<<"; 占"<<(float)gl.bc/number*100<<"%"<<endl;
    cout<<"70-79: " <<gl.cc<<"; 占"<<(float)gl.cc/number*100<<"%"<<endl;
    cout<<"60-69: " <<gl.dc<<"; 占"<<(float)gl.dc/number*100<<"%"<<endl;
    cout<<"0-59: " <<gl.ec<<"; 占"<<(float)gl.ec/number*100<<"%"<<endl;
}
void printTitle(){
    cout<<setw(6)<<"学号"<<setw(10)<<"姓名";
    cout<<setw(5)<<"性别"<<setw(6)<<"成绩"<<endl;
}
void printAStud(Student s){
    cout<<setw(6)<<s.num<<setw(10)<<s.name;
    cout<<setw(5)<<s.sex<<setw(6)<<s.mathscore<<endl;
}
//统计分数，并打印不及格名单
GradeLevel stat_outE(Student s[], int n){
    GradeLevel level = {0};
    float sum = 0;
    for(int i = 0; i < n; i++){
        sum += s[i].mathscore;
        if (s[i].mathscore >= 90)
            level.ac++;
        else if (s[i].mathscore >= 80)
            level.bc++;
        else if (s[i].mathscore >= 70)
            level.cc++;
        else if (s[i].mathscore >= 60)
            level.dc++;
        else{
            level.ec++;
            printAStud(s[i]);
        }
    }
    level.average = sum / n;
```

```

    return level;
}

int main(void){
    Student st[10]={{"001","Wangping",'f',84},{"002","Zhaomin",'m',64},
                    {"003","Wanghong",'f',54},{"004","Lilei",'m',92},
                    {"005","Liumin",'m',75}, {"006","Meilin",'m',74},
                    {"007","Yetong",'f',89}, {"008","Maomao",'m',78},
                    {"009","Zhangjie",'m',66},{"010","Wangmei",'f',39}};

    cout<<"不及格学生名单: "<<endl;
    printTitle();
    GradeLevel gl = stat_outE(st, 10);
    printGradeLevel(gl);
    system("pause");
    return 0;
}

```

执行程序，输入输出如下：

不及格学生名单：

学号	姓名	性别	成绩
003	Wanghong	f	54
010	Wangmei	f	39

统计数据：

考生数量:10 平均成绩: 71.5

90-100:1;占 10%

80-89: 2;占 20%

70-79: 3;占 30%

60-69: 2;占 20%

0-59: 2;占 20%

上面函数 `stat_outE` 对 `n` 个学生进行统计，返回一个统计结果，并打印不及格的学生名单。

例 7-3 对一组学生按其数学成绩由高到低排序输出。

在上面例子基础上再添加 2 个函数，就能解决问题。编程如下：

```

#include <iostream>
#include <iomanip>
using namespace std;
struct Student{
    char num[12], name[20];
    char sex;
    float mathscore;
};
void printTitle(){
    cout<<setw(6)<<"学号"<<setw(10)<<"姓名";
    cout<<setw(5)<<"性别"<<setw(6)<<"成绩"<<endl;
}
void printAStud(Student s){
    cout<<setw(6)<<s.num<<setw(10)<<s.name;
    cout<<setw(5)<<s.sex<<setw(6)<<s.mathscore<<endl;
}
void bubbleSort(Student s[], int n){           //冒泡降序算法
    Student temp;
    for(int i = 0; i < n-1; i++){
        for(int j = 0; j < n-i-1; j++){
            if(s[j].mathscore < s[j+1].mathscore){
                temp = s[j];                     //A 交换结构元素
                s[j] = s[j+1];
                s[j+1] = temp;
            }
        }
    }
}

```

```

        s[j] = s[j+1];
        s[j+1] = temp;
    }
}

void printStuds(Student s[], int n){           //打印一组学生
    printTitle();
    for(int i = 0; i < n ; i++)
        printAStud(s[i]);
}

int main(void){
    Student st[10]={{"001","Wangping",'f',84},{"002","Zhaomin",'m',64},
                    {"003","Wanghong",'f',54},{"004","Lilei",'m',92},
                    {"005","Liumin",'m',75}, {"006","Meilin",'m',74},
                    {"007","Yetong",'f',89}, {"008","Maomao",'m',78},
                    {"009","Zhangjie",'m',66},{"010","Wangmei",'f',39}};

    printStuds(st, 10);           //先打印排序前的名单
    bubbleSort(st, 10);           //排序
    cout<<"按成绩从高到低排序:"<<endl;
    printStuds(st, 10);           //打印排序后的名单
    system("pause");
    return 0;
}

```

上面 `bubbleSort` 函数采用了冒泡降序算法。注意其中 A 行及下面两行，为了交换两个结构元素，需要执行 3 次赋值，而每次赋值都要复制 40 个字节(因为 `sizeof(Student)=40`)。如此交换显然效率低下。如果学生数量更大，这将是很大的计算开销。在后面我们还将用指针来改进排序，使每次赋值只要复制 4 个字节。

### 7.1.6 结构中的静态成员

对于结构类型中的成员，不能用关键词 `register`，`auto`，`extern` 来修饰，但可用 `static` 来修饰。用 `static` 修饰的结构成员就是一个静态成员。

静态成员不同于非静态成员，特点如下：

(1)静态成员的值为该结构类型的所有变量所共享。静态成员的值是结构类型的值，而不是各结构变量的成员值。虽然通过一个结构变量能修改静态成员值，但它只是一个值，所以其它结构变量也能看到这个变化。

(2)在运行时静态成员的值将存储在静态存储区中，其生存期是从程序开始执行到程序结束退出为止。因此即便没有定义该类型的任何变量，该类型的静态成员的值也存在而且可访问。

除了在结构类型定义中要说明静态成员之外，C++语言还要求在全局空间中定义这个静态成员，分配空间并设置初值，格式为：

<数据类型> <结构类型名>::**静态成员名** [= <初值>];

其中，::`是作用域解析运算符。<结构类型名>是静态成员所在的结构类型。<静态成员名>已经在该结构类型定义中进行了说明，并且已确定其<数据类型>。初值缺省为 0。静态成员也可以是数组。`

例如，对于前面定义的 `Employee` 职员类型。如果要描述所有职员的一个最低工资标准，就可以在 `Employee` 结构类型中添加一个成员 `lowestSalary`。由于这个值并非某个职员自身

的数据，而是所有职员共享的一个值，所以该成员应该为静态成员。

```
struct Employee{
    char num[5],name[20];
    char sex;
    Date birthday;
    float salary;
    static float lowestSalary;           //静态成员的说明
};
float Employee::lowestSalary = 800;     //静态成员的定义及初始化
```

访问一个结构类型中的一个静态成员应该使用下面格式：

<结构类型>::<静态成员名>

例如：

```
if (emp.salary < Employee::lowestSalary)
    emp.salary = Employee::lowestSalary;
```

虽然也能通过某个结构变量来访问其静态成员，但容易误解为访问该变量自己的成员，因此不推荐使用。

例 7-4 检查所有职员工资，如果某职员工资低于最低工资标准，就将其工资更改为最低工资。

编程如下：

```
#include <iostream>
using namespace std;
struct Date{
    short year,month,day;
};
struct Employee{
    char num[5],name[20];
    char sex;
    Date birthday;
    float salary;
    static float lowestSalary;           //静态成员的说明
};
float Employee::lowestSalary = 800;     //静态成员的定义及初始化
void checkSalary(Employee emps[], int n){
    cout<<"当前最低工资标准为"<<Employee::lowestSalary<<endl;
    for(int i = 0; i < n; i++){
        if (emps[i].salary < Employee::lowestSalary)
            emps[i].salary = Employee::lowestSalary;
    }
}
int main(){
    Employee::lowestSalary = 900;       //更改静态成员的值
    Employee emplist[3];
    emplist[0].salary = 780;
    emplist[1].salary = 867;
    emplist[2].salary = 1030;
    checkSalary(emplist, 3);
    cout<<emplist[0].salary<<endl;
    cout<<emplist[1].salary<<endl;
    cout<<emplist[2].salary<<endl;

    Employee::lowestSalary = 1000;     //更改静态成员的值
    checkSalary(emplist, 3);
    cout<<emplist[0].salary<<endl;
    cout<<emplist[1].salary<<endl;
```

```

    cout<<emplist[2].salary<<endl;
    system("pause");
    return 0;
}

```

上面例子中 `checkSalary` 函数用来检查并调整一组职员当前工资。在 `main` 函数中，当第一次调整最低工资标准时还没有定义任何职员。在定义了一组职员之后，就可以调用 `checkSalary` 函数来调整职员工资。虽然也能通过某个职员来访问最低工资标准，例如 `emplist[2].lowestSalary`，但这个值是所有职工共享的一个值，因此应通过类型名来访问静态成员。

注意，当计算一个结构的字节大小时，其中的静态成员不包括在内。例如 `Employee` 结构的大小是  $5+20+1+6+4=36$ ，其中并不包括静态成员 `lowestSalary`。也就是说，当定义一个结构 `Employee` 局部变量时，只占用 36 字节用于存储非静态成员。这是因为静态成员在定义结构变量之前就已经存在了。

### 7.1.7 结构的嵌套定义

一个结构 A 可以定义在另一个结构 B 内部，结构 A 称为内嵌类型 `nested type`，结构 B 称为 A 的围类型 `enclosing type`。例如：

```

struct Person {
    char name[20];
    struct Address {    //说明 Address 是 Person 的一个内嵌结构
        char city[20];
        char street[20];
    }resideAddr;        //A 用内嵌结构说明第 1 个变量
    Address jobAddr;    //B 用内嵌结构说明第 2 个变量
};

void myfunc(){
    Person p1 = { "Zhang",{"Nanjing","ZhongShan"},{"ShangHai","ZhangYu"} };
    cout << p1.resideAddr.city << ";" << p1.resideAddr.street<<endl;
    cout << p1.jobAddr.city << ";" << p1.jobAddr.street << endl;
    Person::Address myAddr = { "Beijing","WangFuJing" };
}

```

内嵌类型说明之后就可在其围类型中直接用其名称来说明该类型的变量，例如 B 行。

在围类型变量初始化时，可对其内嵌类型成员也做初始化，但应按说明的次序进行。

访问内嵌类型成员时，应说明完整路径：<围类型变量>.<成员变量>.<内嵌成员变量>

例如：`p1.resideAddr.street`。

在围类型的作用域中，比如一个函数中，其内嵌类型可用于说明变量。内嵌类型名称应包含其围类型：<围类型名称>::<内嵌类型名称> <变量名>，其中要使用作用域解析运算符。

例如 `Person::Address myAddr`;

### 7.1.8 C++结构中的构造函数与析构函数

C++扩展了结构，可添加构造函数、析构函数等，与类 `class` 一样，区别只是缺省为公有 `public` 访问。

如果你使用 C++结构，不管是否编写构造函数，当说明该结构的变量时，就要执行编译器自动生成的缺省构造函数。

结构的构造函数 **constructor** 是在创建该结构的实例或变量时自动执行的一种特殊函数，其作用是对成员定制初始化，以简化结构变量的使用。

构造函数的名称与结构名称一致，可无参也可多参，不说明其返回值，函数体中无需 **return** 语句。例如：

```
struct Date {
    short year, month, day;           //3 个成员
    Date() {}                         //第 1 个构造函数，无参，空体
    Date(int y, int m, int d) {       //第 2 个构造函数，有参
        year = y;
        month = m;
        day = d;
    }
};
```

上面第 1 个构造函数，就是不显式定义任何构造函数时，编译器自动生成的无参构造函数。这样你就可以用 **Date d2;**形式来说明结构的变量。

第 2 个构造函数有 3 个形参，函数体中分别对 3 个成员初始化。这样可以定制初始化过程，而且支持多种灵活方式。比如有一个函数 **void print(Date d)**，你可以临时构造一个日期来调用，例如，**print(Date(2016,4,5))**。

下表列出不同的构造函数所支持的创建实例的不同方式。

表 7-1 构造函数与创建实例方式

构造函数	实例变量说明方式	说明
无构造函数	<b>Date d1;</b> //OK <b>Date d2={2016,4,5};</b> //OK <b>print(Date(2016,4,5));</b> //错误	第 1 行调用缺省的无参构造函数。第 2 行先调用缺省构造函数，然后利用结构成员的公有和有序特性来完成初始化，也兼容传统初始化方式。
仅有 <b>Date()</b>	<b>Date d1;</b> //OK <b>Date d2={2016,4,5};</b> //错误 <b>print(Date(2016,4,5));</b> //错误	只支持无带参形式创建实例。
仅有 <b>Date(y,m,d)</b>	<b>Date d1;</b> //错误 <b>Date d2={2016,4,5};</b> //OK <b>print(Date(2016,4,5));</b> //OK	只支持带参形式创建实例。
<b>Date()</b> <b>Date(y,m,d)</b>	<b>Date d1;</b> //OK <b>Date d2={2016,4,5};</b> //OK <b>print(Date(2016,4,5));</b> //OK	推荐方式。即可满足传统方式，又有定制的灵活性。

构造函数用于支持创建实例，当一个实例撤销时，也要调用析构函数。

结构中的析构函数 **destructor** 是在撤销一个实例变量时自动执行的一种特殊函数，其作用是清理指针成员所指向的动态内存的回收清理。第 8 章详细介绍。

析构函数的语法形式是结构名称之前加~。例如：

```
~Date() {
    cout << "Date::~Date() " <<year<< endl;
}
```

析构函数是无参的，因此是唯一的。析构函数不返回任何东西。

与构造函数一样，如果你不显式定义析构函数，编译器就自动建立一个空体的析构函数。

C++结构中也可定义成员函数，就像类一样。参见第 9 章以后内容。

## 7.2 位域

前面介绍的命名变量的最小单位是一个字节。典型类型就是 `bool` 类型，只有两个值，实际上只需要 1 位就可表示。我们经常遇到大量的小范围值的情形。例如，描述一名职员性别、是否已婚、是否为部门经理、是否休假、工资级别等信息。如果每个数据都用 1 个字节或 4 个字节来表示，在内存紧张的情况下就显得浪费空间，在嵌入式编程环境中尤显突出。有时我们还要在程序中控制特定的设备接口，如串行接口、定时器接口、磁带机驱动器等等，就需要按位进行操作。此时就要用到位域。

位域(bit field)是一种特殊的结构成员类型，也称为位段。一个位域有一个名称，也描述了在结构中所占位数，能直接按名操作(如初始化、赋值、比较等)。

尽管位运算(如移位运算、按位逻辑运算)能解决这些问题，但不能按位命名计算，会降低程序的可读性和可理解性。位域计算能节省内存，也能提高程序可读性，但位域计算的执行效率可能较低，而且往往依赖特定处理器和系统。下面介绍的例子都运行在 32 位 x86 处理器和 VS2015 之上。

### 7.2.1 位域的定义

一个位域就是结构类型中的一个成员，一个结构中可定义多个位域，位域能与普通成员混合定义。定义一个位域的格式为：

<类型> <位域名>:<位数>;

其中，<类型>只能是 `unsigned int`、`signed int` 或 `int` 类型，这是标准 C++ 的要求。VC6.0 系统扩展了 `char`、`short` 和 `long` 类型，带不带符号都可以。<位域名>是一个标识符，作为位域的名称。<位域名>缺省表示这几位跳过不用。<位数>是一个正整数，要求不大于<类型>的位长。例如：

```
struct CELL {
    unsigned short character : 8;    // 00000000 ????????
    unsigned short foreground : 3;  // 00000??? 00000000
    unsigned short intensity : 1;   // 0000?000 00000000
    unsigned short background : 3;  // 0???0000 00000000
    unsigned short blink : 1;       // ?0000000 00000000
};
```

各个位域成员按二进制从最低位开始占用自己的位数。总共占用 16 位，即 `unsigned short` 的 16 位数，用 2 字节描述了 5 个成员。位域成员可与其它成员混合，但多个位域成员往往集中描述。例如：

```
struct Employee{
    char num[5], name[20];
    unsigned gender : 1;           //性别 0 = 女, 1 = 男
    unsigned isMarried : 1;        //是否已婚 0 = false, 1 = true
    unsigned isManager : 1;        //是否为经理 0 = false, 1 = true
    unsigned lay_out : 1;          //是否休假 1 = 休假, 0 = 在岗
};
```



```
    unsigned salaryLevel : 4;    //工资级别 0-15
};
```

在结构中定义了 5 个位域。位域的类型都是 `unsigned int`，每个位域有名字，而且确定了位数。这 5 个位域总共有 8 位，但占用了一个字长，即 32 位，这是因为在一个结构中按多个位域的类型取最长位数，`unsigned` 位长为 32。还要求按字对齐，即该结构的字节长度应该是 4 的整数倍。所以  $5+20+4=29$ ，再对齐到 32，因此 `sizeof(Employee)` 的值为 32 字节。这样看来并没有节省多少内存。

将上面的 5 个位域类型由 `unsigned` 改变为 `unsigned char`:

```
struct Employee{
    char num[5], name[20];
    unsigned char gender : 1;        //0 = 女, 1 = 男
    unsigned char isMarriaged : 1;  //0 = false, 1 = true
    unsigned char isManager : 1;    //0 = false, 1 = true
    unsigned char lay_out : 1;      //1 = 休假, 0 = 在岗
    unsigned char salaryLevel : 4;  //工资级别 0-15
};
```

这样 5 个位域按 `char` 类型取位长并对齐，应该只占 1 个字节，`sizeof(Employee)` 的值就是 26 字节，即  $5+20+1=26$ 。

对于含位域的一个结构，这些位域按什么次序排列是与特定系统相关的。在 VC 系列中按从最低位到最高位排列。例如：

```
struct Mybitfields{
    unsigned short a : 4;        // 00000000 0000????
    unsigned short b : 5;        // 0000000? ????0000
    unsigned short c : 7;        // ????????0 00000000
};
void main(void){
    Mybitfields test = {2, 31, 6};
    cout<<sizeof(test)<<endl;
}
```

结构变量 `test` 的长度是 2 字节，3 个位域按下面逻辑次序排列：

```
00001101    11110010
cccccccb    bbbbaaaa
```

我们看到位域 `b` 跨越了两个字节。这两个字节的 16 进制值为 `0x0DF2`。处理器将 `test` 值作为一个 `unsigned short` 类型数据。在 x86 处理器系统中存储这两个字节时，低字节在前，高字节在后。因此在物理内存中 `0xF2` 字节在前，`0x0D` 字节在后。在后面我们将采用共同体类型来观察此结构的细节。

注意一个位域不能是一个数组。不能定义位域的指针，也不能对位域求地址。

### 7.2.2 位域的使用

位域作为结构的成员，其使用方式与普通成员一样。

在给一个位域赋值时，如果超过它的数值范围，就会自动丢弃高位数据，这相当于按此位域长度求模。例如对于上面 `Mybitfields` 结构，有下面代码段：

```
Mybitfields test;
test.a = 19;                //A
cout<<test.a<<endl;        //输出 3
if (test.a == 19%16)
    cout<<"19%16==3"<<endl;    //输出 19%16==3
```

位域 a 的位数是 4, 值的范围是 0-15。A 行赋值 19, 按 16 求模, 实际存储的是  $19\%16=3$ 。

例 7-5 位域使用示例。管理一组职员的信息。编程如下:

```
#include <iostream>
#include <iomanip>
using namespace std;
struct Employee{
    char num[5], name[20];
    unsigned char gender : 1;        //0 = 女, 1 = 男
    unsigned char isMarriaged : 1;   //0 = false, 1 = true
    unsigned char isManager : 1;     //0 = false, 1 = true
    unsigned char lay_out : 1;       //1 = 休假, 0 = 在岗
    unsigned char salaryLevel : 4;   //工资级别 0-15
};
void printAnEmp(Employee emp){
    cout<<setw(5)<<emp.num<<setw(10)<<emp.name;
    cout<<setw(5)<<(emp.gender == 0 ? "女":"男");
    cout<<setw(5)<<(emp.isMarriaged == 1 ? "已婚":"未婚");
    cout<<setw(5)<<(emp.isManager == 1 ? "经理":"员工");
    cout<<setw(5)<<(emp.lay_out == 1 ? "休假":"在岗");
    cout<<" 工资级别:"<<setw(2)<<(int)emp.salaryLevel<<endl;
}
void printEmps(Employee emps[], int n){
    for(int i = 0; i < n; i++)
        printAnEmp(emps[i]);
}
void f1(Employee emps[], int n){
    for(int i = 0; i < n; i++)
        if (emps[i].gender == 0 && emps[i].lay_out == 0)
            emps[i].salaryLevel++;
}
int main(){
    Employee emplist[4] = { {"001", "张三", 1, 0, 0, 0, 7},
                             {"002", "李四", 0, 1, 0, 1, 8},
                             {"003", "王五", 1, 1, 1, 1, 12},
                             {"004", "赵六", 0, 0, 0, 0, 3}};

    cout<<"调整工资之前"<<endl;
    printEmps(emplist, 4);
    f1(emplist, 4);
    cout<<"调整工资之后"<<endl;
    printEmps(emplist, 4);
    system("pause");
    return 0;
}
```

结构类型 **Employee** 中定义了 5 个位域。例子中描述了如何对位域进行初始化、条件判断、组成逻辑表达式、赋值等操作。其中 **f1** 函数是对所有在岗的女职员加一级工资。执行程序, 输出如下:

调整工资之前

001	张三	男	未婚	员工	在岗	工资级别: 7
002	李四	女	已婚	员工	休假	工资级别: 8
003	王五	男	已婚	经理	休假	工资级别: 12
004	赵六	女	未婚	员工	在岗	工资级别: 3

调整工资之后

001	张三	男	未婚	员工	在岗	工资级别: 7
002	李四	女	已婚	员工	休假	工资级别: 8
003	王五	男	已婚	经理	休假	工资级别: 12
004	赵六	女	未婚	员工	在岗	工资级别: 4

何时需要使用位域？(1)内存紧张时，如嵌入式开发环境中节省内存就很重要。(2)需要控制硬件设备。(3)为了满足特殊需要，例如必须使用他人定义的含位域的结构类型。

## 7.3 枚举

在编程中经常用到这样一些类型，这些类型只有有限的几个值或实例。例如，性别 **Gender** 作为一种类型，只有男和女 2 个值。季节 **Season** 作为类型只有 4 个值。一个星期只有 7 天，一年只有 12 个月，扑克牌只有 4 种花色、每种花色只有 13 个牌面值，一副牌只有 54 张，等等。这样的类型都具有有限的实例，而且不允许再说明或创建新的实例，描述这样的类型就可以使用枚举。

枚举(enumeration，简称 **enum**)是一种用户自定义类型。一种枚举类型就是由若干语义相关的命名常量组成的一个有限集合。枚举类型的一个变量本质上就是一个 **int** 值，只是值的范围受到限制，只能取指定的几个常量，而且通过命名能增强程序可读性。

### 7.3.1 枚举类型及枚举变量

#### 1. 枚举类型

定义一个枚举类型的格式为：

```
enum <枚举类型名> {  
    <枚举元素 1> [= <整型常量 1>],  
    <枚举元素 2> [= <整型常量 2>],  
    ...  
    <枚举元素 n> [= <整型常量 n>]  
};
```

其中，**enum** 是定义枚举类型的关键字；<枚举类型名>是用户定义的标识符。枚举元素也称为枚举常量，也是用户定义的标识符。一个枚举类型应包含多个枚举常量，并用逗号分割。每个枚举常量可以指定一个整数常量值。如果缺省，就会从 0 开始或从前一个指定值开始递增给枚举常量指定一个值。例如：

```
enum Gender{female, male};  
enum Season{ spring=1, summer, autumn, winter};
```

枚举类型 **Gender** 说明了 2 个常量：**female** 和 **male**，分别对应整数 0 和 1。

枚举类型 **season** 有 4 个常量：**spring**、**summer**、**autumn** 和 **winter**。**spring** 的值被指定为 1，那么剩余各元素的值依次为：2, 3, 4。

枚举常量的值也能用 16 进制数来初始化。

注意，在同一作用域中定义多个枚举类型时，类型之间的所有成员之间都不能重名。例如：

```
enum E{e1, e2};  
enum E2{e2, e3};
```

此时 E 中的 e2 与 E2 中的 e2 重名，编译器指出这是重定义错误，原因是枚举类型不能为其成员提供独立的作用域，这一点不同于结构。

枚举类型可以定义为全局类型，可以定义在函数中作为局部类型。全局枚举类型中的各个枚举常量本质上都是全局命名常量，程序中直接读取成员，而不能添加类型名。例如：

```
enum E{e1, e2};
enum E2{ee2, e3};
void main(){
    //cout<<E::e1<<endl;    //编译出错
    cout<<e1<<endl;        //输出 0
    cout<<e2<<endl;        //输出 1
    cout<<ee2<<endl;       //输出 0
    cout<<e3<<endl;        //输出 1
    //...
}
```

上面例子说明，枚举常量隐式转换为 int 再输出。

## 2. 枚举类型的变量

定义枚举类型的变量及初始化与结构类同。例如：

```
enum Season{spring, summer, autumn, winter}s=winter; //定义枚举类型及变量 s
enum Weekday{Sun, Mon, Tues, Wed, Thurs, Fri, Sat}; //定义枚举类型
Weekday day1 = Fri, day2 = Wed; //先定义类型，后定义变量
enum{red, green, blue}a1,a2; //不定义类型名只定义变量
```

枚举类型的一个变量本质上就是一个 int 值，可用一个枚举成员来初始化。

### 7.3.2 枚举的使用

枚举变量与整数类型变量一样，可以进行赋值运算、判等运算、逻辑运算等。

因为枚举变量可隐式转换为 int，因此可将一个枚举变量赋值给一个 int 变量。例如：

```
Weekday day1 = Mon, day2 = Tues;
int day = day1;//OK
```

然而不能将一个 int 隐式转换为一个枚举类型，因此不能对一个枚举变量赋值为一个 int 值，即便该值是有效的成员值。例如：

```
day2 = day;//error
```

可采用强制类型转换运算符(参见 3.4.3 节)或 static\_cast 将 int 转换为枚举类型，例如：

```
day2 = Weekday(day);
day2 = static_cast<Weekday>(day + 1);
```

虽然枚举变量可参与算术运算，但要作为 int 型参与算术运算，而且结果不再是枚举类型。例如，day1+day2 的类型是 int 型，而不是 Weekday 类型。虽然可用强制类型转换将一个 int 值转换为一个枚举值，但结果不一定是合法的枚举常量。例如：

```
Weekday day1 = Fri, day2 = Wed;
Weekday day3 = Weekday(day1 + day2); //day3 的值为 8
cout<<day3<<endl; //输出 8
```

枚举量之间的有效运算实际上非常有限。例如，以二进制位定义的枚举量之间可以进行按位或运算，如前面 open\_mode 类型中，in|out|binary 就表示了一个 int 值，用来控制打开文件的操作模式，详见第 14 章。

在输入输出时，如果用 `cin` 输入一个枚举变量，只能作为 `int` 值输入。同样，如果用 `cout` 输出一个枚举变量，也只能输出其 `int` 值，而不是命名常量的名字。

枚举类型可作为函数的形参或返回值的类型。例如：

```
float getAverageSalary(Employee emps[], int n, Gender sex);
```

枚举类型 `Gender` 作为此函数的形参，按性别计算一组职员平均工资。

枚举类型可作为结构类型中成员的类型。例如：

```
struct Employee{
    char num[5],name[20];
    Gender sex;           //sex 只能取 female 或 male
    ...
};
```

例 7-6 枚举类型的应用示例。编程如下：

```
#include<iostream>
using namespace std;
enum Gender{female, male};    //定义枚举类型
struct Employee{
    char num[5],name[20];
    Gender sex;                //枚举作为结构的成员
    float salary;
};

float getAverageSalary(Employee emps[], int n, Gender sex){//枚举做形参
    float sum = 0;
    int count = 0;
    for(int i = 0; i < n; i++){
        if (emps[i].sex == sex){    //判等运算
            sum += emps[i].salary;
            count++;
        }
    }
    if (count > 0)
        return sum / count;
    return 0;
}

int main(){
    Employee emps[4] ={{"001","张三",female, 1200},
                        {"002","李四",male, 1400},
                        {"003","王五",female, 1600},
                        {"004","赵六",male, 1500}};

    float f1 = getAverageSalary(emps, 4, female);//枚举成员做实参
    float f2 = getAverageSalary(emps, 4, male);
    cout<<"Female avg salary is "<<f1<<endl;
    cout<<"Male avg salary is "<<f2<<endl;
    system("pause");
    return 0;
}
```

上面例子中枚举类型 `Gender` 作为一个结构成员的类型，作为一个函数的形参类型。在函数中枚举变量之间进行关系运算，枚举常量用来初始化结构成员。

传统枚举类型的大小都是 4 字节，即一个 `int` 值的大小。

枚举可定义在结构或类中，作为结构或类的内嵌类型。要访问枚举成员就要添加结构名

或类名在作用域运算符之前，即<结构名>::<成员名>。

建议将内嵌枚举类型名加入，如<结构名>::<枚举类型名>::<成员名>，此时明确要求访问指定枚举类型中的成员。

内嵌的枚举类型可说明变量或作为函数形参，但应说明结构名和作用域：

<结构名>::<枚举类型名> <变量名>

例如：

```
struct S{
    int x;
    enum E{e1, e2} e;           //内嵌枚举类型 1
    enum E2{ee2, e3};          //内嵌枚举类型 2，这两个枚举中的成员之间不重名
};
int myFunc(){
    cout<<sizeof(S)<<endl;      //输出 8，内嵌枚举类型 2 并不占内存
    cout<<S::E::e1<<endl;      //输出 0，访问内嵌枚举的成员，有枚举作用域
    cout<<S::e1<<endl;         //输出 0，访问内嵌枚举的成员，无枚举作用域
    cout<<S::E2::e3<<endl;      //输出 1，访问内嵌枚举的成员，有枚举作用域
    cout<<S::e3<<endl;         //输出 1，访问内嵌枚举的成员，无枚举作用域
    S::E2 e11 = S::E2::e3;      //用内嵌枚举说明变量并初始化
    S s1={1, S::E::e1};         //结构变量说明及初始化
    S s2={2, S::e2};           //结构变量说明及初始化
}
```

枚举类型有两种用法：一种是定义枚举类型的实例，然后操作该实例。就像上面例 7-6。另一种是使用枚举类型中的成员，经常在结构或类中定义枚举类型，而后通过作用域运算符来使用其中的枚举成员。

使用枚举类型有两个好处：一是约束整数常量的有效范围，使用枚举常量，编译器能检查正确性；二是用一组命名常量代替整数表示特定语义，可读性好，易于理解。

### 7.3.3 C11 强类型枚举

前面介绍的都是传统 C 语言中的枚举，C11 扩展了“作用域限定 *scoped*”枚举类型，也称为强类型枚举，将传统枚举作为“无作用域限定 *unscoped*”枚举。

作用域限定的枚举的说明语法：

enum [class|struct] [标识符] [:类型]{枚举常量表};

在 **enum** 之后添加 **class** 或 **struct** 说明就称为强类型枚举。对强类型枚举，访问其成员就必需用“标识符::成员名”。例如：

```
enum class Suit { Diamonds, Hearts, Clubs, Spades };
void PlayCard(Suit suit){
    if (suit == Suit::Clubs){
        /*...*/
    }
}
```

相同作用域中的不同强类型枚举之间可定义同名成员。例如：

```
enum class E { e1, e2 };
enum class E2 { e2, e3 }; //e2 是同名成员
```

强类型枚举变量不能隐式转换为整数 **int**，例如：

```
Suit hand = Suit::Clubs;
int intHand = hand;
```

强类型枚举变量需要显式强制类型转换才能转换为 `int`，例如：

```
int intHand = (int)hand;
int intHand2 = static_cast<int>(hand);
```

同理，`cout` 不能直接输出强类型枚举成员或其变量，也要先强制转换为整数后才能 `cout` 输出。例如：

```
cout << int(E::e2)<< endl;
cout << static_cast<int>(E2::e2) << endl;
```

强类型枚举变量不能参与算术运算，如加减，原因是强类型枚举变量作为对象，而不是整数，因此不能隐式转换为整数。

从无域限定的枚举添加一个 `class` 成为强类型枚举，增加很多限制，但强类型编程确实应该如此。

## 7.4 共同体

共同体(`union`，也称为联合)是一种用户自定义类型，一个共同体由若干成员组成，这些成员共享同一块存储空间。对于一个共同体变量，在某一时刻只能使用其中一个成员，而且读出数据的成员应与写入数据的成员是同一个，否则结果难料。共同体的用处很有限，使用共同体的唯一好处可能就是节省内存，但易出错。

### 7.4.1 共同体类型的定义

定义一个共同体类型的格式为：

```
union <共同体类型名>{
    <成员类型 1> <成员名 1>;
    <成员类型 2> <成员名 2>;
    ...
    <成员类型 n> <成员名 n>;
};
```

其中<成员类型>可以是已定义的任何类型，也包括结构类型、枚举类型、共同体类型，以及这些类型的数组。其实定义共同体类型与定义结构类型的格式完全相同，只要用关键字 `union` 代替 `struct`。例如：

```
union Number{
    unsigned char c[4];
    int x;
    float y;
};
```

定义了一个共同体类型 `Number`，其中包含 3 个成员。每个成员都是 4 字节，所以 `Number` 类型变量的大小就是 4 字节。

### 7.4.2 共同体变量的说明及使用

#### 1. 共同体变量的说明及初始化

说明共同体变量的方法也与结构相同，区别是初始化只能有一个成员的值。例如：

```

union Number{
    unsigned char c[4];
    int x;
    float y;
}n1={0,1,2,3};           //对第1个成员初始化

```

## 2. 共同体变量的使用

共同体变量及成员的使用与结构变量相似，区别在于多个成员共享同一空间，共同体变量的大小就是其中最大成员的大小。

例 7-7 利用共同体的特点，查看整数和浮点数的内存实际存储情况。

```

#include <iostream>
using namespace std;
union Number{
    unsigned char c[4];
    int x;
    float y;
};

void printBytes(Number n){ //共同体作为形参
    cout<<"bytes:";
    cout<<hex<<(int)n.c[0]<<"-"<<(int)n.c[1]\
        <<"-"<<(int)n.c[2]<<"-"<<(int)n.c[3]<<endl;
}

int main(){
    Number a;           //说明共同体变量
    a.x = 1;            //对其中 int x 成员赋值 1
    cout<<"int = 1"<<endl;
    cout<<"float = "<<a.y<<endl;    //以其中 float y 成员输出
    printBytes(a);       //以其中 char c[]成员输出
    a.y = 1;
    cout<<"\nfloat = 1.0"<<endl;
    cout<<"dec int = "<<dec<<a.x<<endl;
    cout<<"hex int = "<<hex<<a.x<<endl;
    printBytes(a);
    system("pause");
    return 0;
}

```

例子中 `printBytes` 函数按字节次序以 16 进制打印各字节内容。

执行程序，输出如下：

```

int = 1
float = 1.4013e-045
bytes:1-0-0-0

float = 1.0
dec int = 1065353216
hex int = 3f800000
bytes:0-0-80-3f

```

先说明了一个共同体变量 `a`，然后把 `int` 值 1 赋给它，然后能看到对应的 `float` 值，以及各字节的排列。这个例子验证了 x86 系统按低位字节在前、高位字节在后的顺序存储。

例 7-8 利用共同体，验证结构中的位域成员的排列规律。

```

#include <iostream>
using namespace std;
struct Mybitfields{
    unsigned short a : 4;
    unsigned short b : 5;
}

```



```

    unsigned short c : 7;
};
union BitBytes{
    unsigned char c[2];
    Mybitfields bitb;
};
int main(void){
    BitBytes a;
    a.bitb.a = 2;
    a.bitb.b = 31;
    a.bitb.c = 6;
    cout<<"bytes:"<<hex<<(int)a.c[0]<<"-"<<(int)a.c[1]<<endl;
    system("pause");
    return 0;
}

```

含位域的结构类型 **Mybitfields** 在前面介绍过。我们要验证在物理内存中各位域的排列规律。

执行程序，输出如下：

```
bytes:f2-d
```

3 个位域按下面**逻辑次序**排列：

```

00001101 11110010
cccccccb bbbbaaaa

```

这两个字节的 16 进制值为 0x0DF2。在存储这两个字节时，低字节在前，高字节在后。因此我们看到在物理内存中 0xF2 字节在前，0x0D 字节在后。

在使用共同体时，应注意以下几点：

- (1) 对于一个共同体变量，用哪个成员写入就应该用它来读出，否则结果难料。
- (2) 当写入一个“小”成员时，仅改变部分字节，而不改变其它字节。

以前的共同体成员受到限制。C11 标准支持任意类型的成员(所谓无限制的共同体 **Unrestricted Union**)，也支持匿名共同体，可说明构造函数和成员函数。但如果任何一个成员类型带有自定义构造函数，就意味着必须添加更多编码用于成员的创建和撤销。读者可阅读相关文档。

## 7.5 类型定义 typedef 与 using

用关键字 **typedef** 给一个已有的数据类型起一个同义词名称，目的是简化编程，增强可读性。语法格式如下：

```
typedef <类型> <类型同义词>;
```

其中，<类型>可以是基本类型名，也可以是用户自定义类型名。<类型同义词>是一个标识符。这是一条说明语句，但不能出现在函数体之中。类型同义词具有文件作用域。

在定义之后就可以用同义词来说明变量或函数形参。例如：

```

typedef float REAL;           //将 float 定义为 REAL
typedef unsigned int size_t;   //将 unsigned int 定义为 size_t
REAL x,y;                     //等同：float x,y;
typedef struct B{
    char name[20];

```

```
char author[10];  
float price;  
}BOOK;                                //定义结构类型 B, 定义了一个别名 BOOK  
BOOK book[10];                        //等同: B book[10];  
typedef char AUTHOR[10];              //给 char[10]起别名 AUTHOR  
AUTHOR author;                        //等同: char author[10];
```

C11 采用 using 也能为已有数据类型起一个别名, 语法如下:

```
using <类型同义词> = <类型>;
```

与上面 typedef 具有相同效果, 例如:

```
using size_t = unsigned int;
```

当为函数指针(下一章介绍)起别名时, using 更简单易懂:

```
using func = void(*) (int); 等价于
```

```
typedef void (*func) (int);
```

typedef 不支持模板别名, 而 using 可支持(模板在第13章介绍)。

注意, typedef 与 using 并未产生新的数据类型, 只是给出已有类型的同义词或别名。

## 7.6 小 结

- 一个结构(也称为结构体)就是一种聚集数据类型, 有一个名字, 包含了一组成员。当定义该结构的一个变量时, 每个成员都持有自己的值。
- 一个结构类型的存储大小是其各成员所占字节之和, 但要根据最大成员的字节大小对齐。
- 同类型的两个结构变量之间可以赋值, 就是复制所有成员值。
- 同类型的两个结构变量之间不能进行关系运算, 即不能判等、大于、小于等计算。
- 用成员运算符来访问成员: <结构变量>.<成员名>。
- 结构类型可作为函数的形参或返回值。实参变量将赋值给形参(复制所有成员), 然后开始执行函数。当函数返回一个结构变量时, 也要通过一次赋值来得到结果。
- 结构类型可定义数组。结构数组常作为函数的形参。
- 结构类型中能包含静态成员。静态成员的值为该结构类型的所有变量所共享。
- 位域是一种特殊的结构成员类型, 也称为位段。一个位域有一个名称, 也描述了在结构中所占位数, 能直接按名操作(如初始化、赋值、比较等)。位域计算能节省内存, 也能提高程序可读性, 但位域计算的执行效率可能较低, 而且往往依赖特定处理器和系统。
- 一个枚举是由若干语义相关的命名常量组成的一个有限集合。枚举类型的一个变量本质上就是一个 int 值, 只是值的范围受到限制, 只能取指定的几个常量。虽然枚举变量可参与算术运算, 但要作为 int 型参与算术运算, 而且结果不再是枚举类型。
- 枚举类型可作为函数的形参或返回值的类型, 可作为结构类型中的成员。
- 使用枚举类型的好处是能约束整数常量值的有效范围, 用一组命名常量代替整数参与运算, 使程序更易理解。
- 一个共同体(也称为联合)由若干成员组成, 这些成员共享同一块存储空间。

- 对于一个共同体变量，在某一时刻只能使用其中一个成员。
- 共同体的大小就是其中最大的成员的大小。
- 使用共同体的好处是节省内存。另一个用途就是能在物理内存中查看不同类型数据的实际存储情况。
- 关键字 `typedef` 可以给一个已定义的数据类型起一个别名。

## 7.7 练 习 题

1. 关于结构类型，下面哪一种说法是错误的？
  - A 每个结构类型都有一个名称。
  - B 结构中每个成员都用一个名称。
  - C 结构中各成员的名称不重复。
  - D 每个成员都有自己的类型。
2. 设有语句：`struct xy{ int x ; float y; char z ;} example;` 下面哪一个叙述错误？
  - A `struct` 是结构类型的关键字。
  - B `example` 是结构类型的名称。
  - C `x, y, z` 都是结构的成员名称。
  - D `struct xy` 或 `xy` 是结构类型的名称。
3. 关于结构类型，下面哪一种说法是错误的？
  - A 结构类型不可作为其成员的类型。
  - B 结构变量的大小就是其各成员的大小之和。
  - C 结构类型可以定义在函数之外。
  - D 结构类型可以定义在函数之中。
4. 分析下面语句：

```
struct Property{
    char name[20];
    char value[40];
}p1 = {"name", "ZhangSan"}, p2 = {"age"}, p3 = {"blue"}, p4 = p1;
```

  - A `p1` 出错
  - B `p2` 出错
  - C `p3` 出错
  - D `p4` 出错
5. 对于结构变量的操作，下面哪一种说法是错误的？
  - A 两个相同类型的结构变量之间可以赋值，就是复制全部成员的值。
  - B 两个结构变量之间不能进行关系运算。
  - C 不能用 `cin` 输入一个结构变量的全部成员。
  - D 可以用一个 `cout` 输出一个结构变量的全部成员的值。
6. 有以下语句：`struct Point{int x, y;}ps[3] = {{1, 2}, {3, 4}};`  
那么 `ps[1].x` 和 `ps[2].y` 的值分别是\_\_\_\_\_。
  - A 1 2
  - B 1 4
  - C 3 4
  - D 3 0
7. 对于结构中的静态成员，下面哪一种说法是错误的？
  - A 用 `static` 修饰的成员一定要在全局空间中定义并初始化。
  - B 该结构的所有变量将共享静态成员的一个值。

C 无论是否说明该结构的变量，该结构的静态成员都已经存在。

D 不能用某个结构变量::静态成员来改变静态成员的值。

8. 设有以下枚举类型说明语句：

```
enum weekday{Mon=1,Tues,Wed,Thurs,Fri,Sat,Sun=0}week;
```

下面那一条赋值语句是错误的？

A week=weekday(1);    B week=1;    C week=Mon;    D week=(weekday)1;

9. 下列程序的输出结果是\_\_\_\_\_。

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

```
void main(void){
```

```
    enum tag{Up=1, Down, Left, Right}x = Up, y;
```

```
    enum tag z = Left; y = Down;
```

```
    cout<<x<<" "<<y<<" "<<z<<endl;
```

```
}
```

A Up Down Left    B 1 2 3    C 0 1 2    D Up Left Down

10. 下面哪一条语句是错误的？

A union work{ char ch[10]; int i; float f;}x={"teacher",10,6.3};

B union work{ char ch[10]; int i; float f;}x={"teacher"};

C union work{ char ch[10]; int i; float f;}x={10};

D union work{ char ch[10]; int i; float f;}x={6.3};

11. 设有以下语句：

```
union Numeric{int i; float f; double d;}u;
```

变量u所占存储单元的字节数为\_\_\_\_\_。

A 16

B 4

C 8

D 12

12. 设有以下说明语句：

```
struct test{
```

```
    int i; char ch; float f;
```

```
    union uu {char s[5]; int m[2];} ua;
```

```
} ex;
```

下列对成员m[1]的正确引用是\_\_\_\_\_。

A ex.m[1]

B ex.uu.m[1]

C ex.ua.m[1]

D ex.test.m[1]

缺省配置下，sizeof(test)的值是\_\_\_\_\_。

A 12

B 9

C 22

D 20

13. 下面程序的输出结果是\_\_\_\_\_。

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

```
void main(void){
```

```
    union baby {
```

```
        char name[10];
```

```
        int number;
```

```
    }b={"YangYang"};
```

```

        cout<<b.name<<" ";
        b.number=65;
        cout<<b.name<<" "<<b.number<<endl;
    }

```

A YangYang YangYang 65

B YangYang 65 65

C YangYang A 65

D YangYang 65

14. 读程序，给出下面程序输出结果。

```

#include <iostream.h>
struct A{
    int x, y;
    static int dx;
};
int A::dx = 3;
void main(){
    cout<<A::dx<<endl;
    A a1 = {1, 2}, a2 = {3, 4};
    cout<<a2.dx<<endl;
    a1.dx = 2;
    cout<<a2.dx<<endl;
    cout<<A::dx<<endl;
}

```

15. 根据要求编写完整的程序。

(1)输入一个班的若干同学的通讯录。该通讯录包括：学号、姓名、住址、电话、E-mail 等。从键盘输入某人的姓名，然后从已有通讯录中查找该人是否存在，并给出相应信息。打印通讯录。

(2)输入  $n$  个人的编号、姓名、身高，然后按身高从小到大的顺序排列输出。

(3)建立一个结构类型 **Course** 表示课程，包括课程编号、课程名称、考核方式。建立一个枚举类型表示两种考核方式：一种是百分制，一种是等级制，即 A、B、C、D、E。一门课程只能选其一种考核方式。建立一个结构类型，表示学生成绩，包括学号、姓名、课程编号、考核成绩。注意，考核成绩的类型应该是一个共同体，它的一个值要么是一个百分制成绩，要么是一个等级成绩，要根据该课程的考核方式来定。编写程序并测试。

## 第8章 指针和引用

在程序运行时变量和函数都存放在内存中，通过变量名来访问数据、通过函数名来调用函数都是直接访问方式。还有另一种间接访问方式就是用指针。指针的本质是内存地址。指针往往用于说明函数的形参，使实参能通过指针传递，以提高函数调用的效率。利用指针能动态地使用内存，以提高内存使用效率。指针也能用来表示数据关联，以构成复杂的数据结构。指针是 C 程序中最常见的类型。引用是 C++ 扩展的新概念，主要用于函数形参和返回类型。本章将详细介绍指针和引用的概念及应用。

### 8.1 指针及指针变量

指针(pointer)的本质是内存地址。指针变量就是专门存储地址的一种变量。通过指针变量所存储的地址来访问数据是一种间接寻址方式。由于处理器的机器语言能支持间接寻址，所以使用指针可以达到较高的计算性能。

#### 8.1.1 地址的概念

C++ 编译器对不同对象或变量按其数据类型分配合适大小的存储空间。例如为 `char` 或 `bool` 型变量分配 1 个字节(bytes)的存储空间，`short` 分配 2 字节，`int` 和 `float` 分配 4 个字节，为 `double` 型变量分配 8 个字节的存储空间。当程序执行时，代码和变量都加载到内存中。计算机内存被分成若干个存储单元，存储单元以字节为单位。每个存储单元都有一个固定的编号，这个编号就是内存地址。尽管一个变量可能占用多个字节空间，但都通过第一个字节的地址来访问。存放某个变量的第一个字节的地址就是该数据的首地址。

指针即内存单元的地址，而数据是内存单元中的内容(或值)。

假设在程序中说明了 1 个 `int` 型的变量 `a`，其值为 68。系统为变量 `a` 分配 4 字节的存储空间，设首地址为 `0X0065FDF4`。通过地址 `0X0065FDF4` 就能找到变量 `a` 在内存中的存储单元，从而对变量 `a` 进行访问。`0X0065FDF4` 就是变量 `a` 的指针。知道一个变量的地址和变量的类型就能对变量进行访问，就如同知道房间号就能找到房间，从而找到房间里的主人。

指针是一种特殊的数据类型。所有类型的变量，无论是基本类型、用户定义类型、还是这些类型的数组，在一次运行时都有确定的地址，因此它们都有指针。对于 32 位系统，地址长度就是 32 位，因此一个指针需要 4 个字节，与整型 `int`、浮点型 `float` 具有相同大小的长度。一个指针不仅有值，而且还要确定其类型，表示它能指向什么类型的数据，决定了通过它要取用多少字节作为该变量的值。

同一个变量在不同机器上执行或在不同时刻执行，其地址都不一样。这是因为在加载一个程序时，系统根据当前可用内存来确定使用哪一块内存。在编程中一个具体的地址值没有

多大意义，不应该直接用一个地址常量来为一个指针赋值，这是因为在运行时你不知道这个地址中当前存放的是什么。如果你通过这个地址读取它的内容，将得到不可预知的结果。如果你改变了它的内容，就会导致不可预知的后果，甚至导致严重错误而退出。所以对指针的操作应小心谨慎。

怎样能知道一个变量在运行时刻的内存地址？把取地址运算符&放在变量前面就得到它的首地址。例如 b 是一个变量，那么&b 就表示它的地址。下面例子能看到一组局部变量的首地址。

例 8-1 显示一组局部变量的首地址。

```
#include <iostream>
using namespace std;
int main(){
    bool b = true;
    char c = 'c';
    short s = 3;
    int i = 4;
    float f = 1.0f;
    double d = 1.0;
    cout<<"&b="<<&b<<endl;
    cout<<"&c="<<hex<<(int)&c<<endl;
    cout<<"&s="<<&s<<endl;
    cout<<"&i="<<&i<<endl;
    cout<<"&f="<<&f<<endl;
    cout<<"&d="<<&d<<endl;
    system("pause");
    return 0;
}
```

执行程序，输出如下：

```
&b=0025F8CA
&c=25f8cb
&s=0025F8CC
&i=0025F8D0
&f=0025F8D4
&d=0025F8D8
```

注意在不同机器上或不同时刻运行，显示的地址都不一样。上面是 VS2015 选择 x86/32 位，Release 编译，得到的结果。能看到局部变量在内存中的一些排列规律。如图 8.1 所示。

每个变量在运行时刻都具有明确的内存地址。每个地址都是 32 位二进制值。其中变量 s 类型为 short 应占 2 字节，但实际上却分配了 4 字节空间。每个编译系统或运行系统都具有自己的分配内存的特殊方法。以 Debug 编译或在 VC6 上运行，地址顺序将颠倒过来，这是因为局部变量将以堆栈方式存储。

如果一个变量存放的是某个对象或值的地址，就说该变量是一个指针变量，且指向某个对

0025F8CA	b : 1	占1字节
0025F8CB	c : 'c'	占1字节
0025F8CC	s : 3	占4字节
0025F8D0	i : 4	占4字节
0025F8D4	f : 1.0f	占4字节
0025F8D8	d : 1.0	占8字节

图 8-1 一组局部变量的存储地址

象或值。在 C++ 程序设计中，指针变量只有确定了指向才有意义。

### 8.1.2 指针变量

指针变量就是专门存放地址的一种特殊变量。指针变量中存放的是地址值。一个指针的值就是一个地址。

指针变量与其它变量一样，在使用之前必须先说明。说明指针变量的格式为：

<类型名> \* <变量名> [= &<变量>];

其中，<类型名>是这个指针变量所指向的对象的类型，简称指针类型，它可以是任何一种类型。\*表示这个变量是一个指针变量。这个变量的类型就是“<类型名> \*”。<变量名>是一个标识符。指针变量可以进行初始化，等号之后给出一个变量的地址，要求这个变量的类型与指针类型相符。

假设程序中说明了一个变量 `int i = 4`，而且在运行时该变量 `i` 的地址为 `0025F8D0`。

说明一个指针变量：

```
int * pa = &i;
```

此时指针变量 `pa` 中就存放了变量 `i` 的地址，即 `pa` 中存放的值为 `0025F8D0`。我们称 `pa` 指针指向了变量 `i`。如图 8.2 所示。

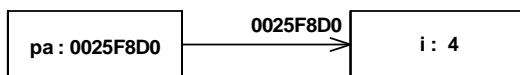


图 8-2 指针变量的值与其所指内容

现在访问变量 `i` 就有两种方式：一是按变量名 `i` 来访问。将变量名 `i` 转换为一个相对地址，在运行时经动态定位得到 `i` 的地址，再找到 `i` 的存储单元。二是通过指针变量 `pa` 来访问。按 `pa` 变量的地址先找到 `pa`，然后根据 `pa` 的值再找到变量 `i` 的存储单元，从而对变量 `i` 进行访问。前一种访问方式称为直接寻址，后一种称为间接寻址。间接方式的好处是一个指针 `pa` 在不同时刻可指向不同的整数变量，这样通过一个指针变量就能访问多个数据。

例如：

```
int *ip1, i2;           //说明一个指针变量 ip1(其类型为 int *)和一个 int 变量 i2
float *fp;              //说明一个指针变量 fp(其类型为 float *)
```

指针变量 `ip1` 所指向的变量的类型为 `int` 型，指针变量 `fp` 所指向的变量的类型为 `float` 型。

下面说明指针变量的几种写法都是合法的。

```
int *p;                 // *与类型名之间有空格，与变量名 p 之间没有空格
int* p1;                // *与类型名之间没有空格，与变量名 p1 之间有空格
int*p2;                 // *与类型名和变量名 p2 之间都没有空格
int * p3;               // *与类型名和变量名 p3 之间都有空格
```

在说明一个指针变量后，无论该指针变量指向何种类型的对象，系统都为其分配 4 个字节大小的存储空间，因为指针变量存放内存单元的地址，在 32 位计算机中内存地址的取值范围是相同的，都是 4 个字节。

在一个变量说明语句中，或者一个函数形参表中，一个变量或形参之前加一个“\*”，就说明了该变量或该形参是一个指针类型。注意这里的“\*”是一个说明符号，并非一个运算符，



区别于“\*”作为双目乘法运算符。下面还将介绍“\*”作为单目运算符的第三种用法。

### 8.1.3 指针的运算

既然指针就是地址,那么指针的运算实际上就是地址的运算。但由于地址是特殊的数据,使指针所能进行的运算受到一定限制。对于指针只能进行赋值运算、间接引用运算、算术运算、两个指针间的减运算和关系运算。

#### 1. 赋值运算

一个指针变量如果既没有初始化,也没有赋值,那它的值就是随机值,指向就不确定。如果此时使用指针变量,就会访问不确定的存储单元,可能有很大危害。因此指针变量在使用之前必须有确定的指向,通过给指针赋值就可以使之指向确定的数据。

下面例子说明如何给指针赋值,以及应注意的一些问题。

```
int a = 16, b = 28;           //说明整型变量 a,b
float x = 32.6f, y = 69.1f;   //说明浮点型变量 x,y
int *pa, *pb = &b;           //说明两个指向 int 对象的指针变量 pa,pb, 并使 pb 指向变量 b
float *px, *py = NULL;        //说明两个指向 float 对象的指针变量 px,py, 使 py 为空指针
px = &x;                      //使指针 px 指向变量 x
*pa = &b;                     //非法, 左值与右值的类型不同, 左值是 int 型, 右值是 int*型
pa = pb;                      //pa 和 pb 都指向同一个变量
pa = &x;                      //非法, pa 指向对象的类型只能是 int 型, 而 x 是 float 型
pb = 0x3000;                  //非法, 不能用字面值给指针变量赋值
```

在一个变量名之前加一个“&”,就是取得该变量的地址,称为求址运算(address-of)。例如,“&b”是一个表达式,就是求变量 b 的地址。如果 b 的类型是 int,那么表达式&b 的类型就是“int\*”,以此类推。这里“&”是一个单目运算符。前面第3章介绍的“&”是一个双目运算符,就是按位逻辑与运算符。本章后面还将介绍“&”的第三种用法,用来说明引用变量。

NULL 是一个宏,其值为 0,说明在多个头文件中,例如<iostream>中包含的<iios>。值为 NULL 的指针称为空指针。当一个指针变量暂时无法确定其指向或暂时不用时,可将它赋予 NULL。在使用一个指针前应判定不能为空,以保证程序正常执行。

在为指针赋值时应保证指针类型的一致性。例如,你不能把一个 int 变量的地址赋给一个 float 指针,反之也不行。即便你使用强制类型转换能通过编译,也会在运行时出现错误。

注意,赋值运算还隐含在函数调用时,实参值传递给形参也是一种赋值。

#### 2. 间接引用运算

对于一个指针变量,就能通过它的值来访问对应地址的值。这种由地址求内容的过程就是间接引用运算。“\*”作为一种单目运算符,作用于一个指针变量,就是按该指针的地址求值(英文称 dereference,含义是去引用,习惯称为间接引用)。例如:

```
int a = 16, b = 28;
int *pa, *pb = &b;
cout<<*pb<<endl;           //输出 pb 所指对象 b 的值 28
```

```
pa = &a;
cout<<a<<endl;           //输出 a 的值 16
*pa = 32;                 //修改 pa 所指对象 a 的值, 使 a 的值变成 32
cout<<a<<endl;           //输出 a 的值 32
```

从上面的例子可以看出, “\*指针变量” 既可作为左值, 也可作为右值。作为左值可以修改指针所指对象的值, 作为右值只能读取指针所指对象的值。

如果指针 `pa` 的类型是 “`int *`”, 那么 “`*pa`” 的类型就是 `int`, 这就是去引用的含义。

“`&`” 所表示的求址运算与 “`*`” 所表示的间接引用运算是一对互逆的运算:

- 对于任一个变量 `v`, “`*&v`” 表达式等价于 `v`, 即 `*&v == v`。
- 对于任一个指针变量 `p`, “`&*p`” 表达式等价于 `p`, 即 `&*p == p`。

这是两个重要的等价式, 将用于后面的数组访问。

至此, “`*`” 有三种不同的含义, 随其所作用的对象及位置的不同而不同。例如:

```
int a = 16, b = 28;
int *pa = &a, *pb;    // *表示 pa、pb 是指针变量, 指针说明
a * = a * b;          // *表示乘法运算, 双目运算符
*pa = 123;            // *表示间接引用 pa 所指对象 a, 单目运算符
```

指针的一个重要用途是定义函数形参。在调用函数时所提供的实参(即主调方某变量的地址)必须符合形参的指针类型, 才能将实参赋值给形参。函数内通过形参指针就能访问主调方的变量。

#### 例 8-2 指针作为函数形参。

```
#include <iostream>
using namespace std;
void square(double *d){           //A
    if (d != NULL)                //B
        *d = *d * *d;            //C 等价于 *d = (*d) * (*d);
}
int main(){
    double d1 = 2.2;              //D
    square(&d1);                  //E
    cout<<d1<<endl;              //F
    system("pause");
    return 0;
}
```

上面 A 行定义了一个函数 `square`, 其形参为一个 `double` 型指针。此函数对指针所指对象的某个 `double` 变量求平方。此函数没有返回值。B 行先判断形参指针不为空, 这是指针作为形参的通常处理方法, 目的是防止通过空指针进行有害操作。C 行包含了间接引用运算符、乘法运算符和赋值运算符, 完成平方计算。注意 C 行并没有改变指针 `d`, 而是改变了指针 `d` 所指对象的调用方实参的值。

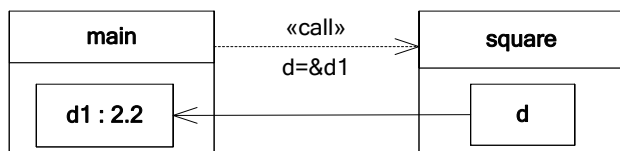


图 8-3 函数形参指向主调方的变量

主函数 `main` 中在 D 行定义了一个局部变量 `d1`。E 行调用 `square` 函数，实参为 `&d1`，即变量 `d1` 的地址被赋值给函数形参 `d`，使形参 `d` 指向变量 `d1`。如图 8.3 所示。在执行函数 `square` 时就能读取并改变该变量的值。当函数返回后，F 行输出的 `d1` 就是原值 2.2 的平方，4.84。

用指针作为函数形参的好处就是用传递地址来代替传递值。不管多大对象，传地址只需 4 字节，所以适合传递“大”对象给函数来处理，能提高计算效率。

### 3. 与整数的算术运算

指针与整数之间的算术运算有两种：与整数的加减运算和自增自减运算。这些运算只在访问某个数组时才有用。

#### (1) 与整数的加减运算

既然指针变量存储的是数据的内存地址，就可将指针视为类似整型的变量。指针加上或减去一个整数的结果应该是一个新的地址值。

指针的加减运算与普遍变量的加减运算不同。一个指针加上或减去一个整数 `n`，表示指针从当前位置向后或向前移动 `n` 个 `sizeof(<数据类型>)` 字节的存储单元。指针的数据类型决定了加减运算的单位尺度。

一个数组是一组相同类型的元素的集合，每个元素的字节大小相同。指针的加减运算就是元素指针的前移或后移，这样就可用来访问数组元素。例如：

```
int a[] = {11,22,33,44,55,66,77,88}; //定义一个 int 数组 a
cout<<"a="<<a<<endl;           //输出 a 是头一个元素的地址
int * pb = &a[4];                 //pb 指向 a[4] 元素
cout<<"pb="<<pb<<" *pb="<<*pb<<endl; //输出 a[4] 元素的地址和值
cout<<"pb+3="<<pb+3<<" * (pb+3)="<<*(pb+3)<<endl; //输出 a[7] 元素的地址和值
cout<<"pb-2="<<pb-2<<" * (pb-2)="<<*(pb-2)<<endl; //输出 a[2] 元素的地址和值
```

假设数组变量 `a` 的首地址是 `0x0012FF60`，则 `a[4]` 元素的地址计算如下：

`pb = &a + 4*sizeof(int) = 0x0012FF60 + 4*4 = 0x0012FF70。`

`pb+3 = &pb + 3*sizeof(int) = 0x0012FF70 + 3*4 = 0x0012FF7C，指向 a[7] 元素。`

`pb-2 = &pb - 2*sizeof(int) = 0x0012FF70 - 2*4 = 0x0012FF68，指向 a[2] 元素。`

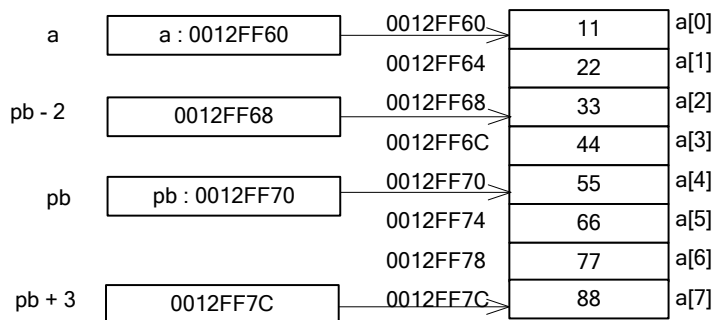


图 8-4 指针的加法和减法运算

图 8.4 给出了指针移动示意图。注意这种计算只能用于访问数组，而且访问地址不能越界，就如同下标不能越界一样。

通过上面例子我们知道，一个指针加减一个整数常量就是指针后移或前移。但要注意不能对两个指针变量相加。我们也看到，一个数组的名字本质上就是头一个元素的地址，但这个指针的值是一个常量，不能改变它的值。

### (2) 自增、自减运算

指针的自增、自减运算是指针加减运算的一种特例，就是指针从当前位置向后或向前移动 `sizeof(数据类型)` 字节大小的 1 个存储单元。例如：

```
int a[] = {11,22,33,44,55,66,77,88};
int * pb = &a[4];
int * p1 = pb--;
```

此时，指针 `p1` 指向 `a[4]`，而 `pb` 指向 `a[3]` 元素。

注意，指针的自增、自减往往与间接引用混合运算，此时应注意计算顺序及语义。例如：

```
int a[] = {11,22,33,44,55,66,77,88};
int b, * p = &a[4];
```

下面分别计算各个表达式语句(以自增为例)：

```
b = *p++; //先取 p 指向的 a[4] 的值 55，然后使指针 p 后移指向 a[5]
b = *(p++); //等同于 *p++
b = (*p)++; //读取 p 指向的 a[4] 的值 55，然后对其自增 1，成为 56
b = *++p; //先使指针 p 后移指向 a[5]，然后读取 p 所指向的值 66
b = ++*p; //先读取 p 指向的变量 a[4]，然后对 a[4] 自增 1，成为 56
b = ++*p++; //a[4] 自增 1，成为 56，然后指针 p 后移指向 a[5]
```

## 4. 两指针相减

当两个指针都指向同一数组元素时，这两个指针的相减才有意义。两个指针相减的结果为一整数，表示两个指针之间数组元素的个数。例如：

```
int a[10] = {11,22,33,44,55,66,77,88};
p1 = &a[1]; //p1 指向 a[1]
p2 = &a[6]; //p2 指向 a[6]
cout<<p2-p1; //输出 5
```

对于两个指针，不适合进行其它算术运算，如加法、乘法、除法、求余等，也不适合进

行按位运算。这些计算即便能执行，但因结果难以控制，也是危险的。

## 5. 关系运算

指针关系运算一般有下列两种情况：

- 比较两个指针所指向的对象在内存中的位置，如相等还是不相等，判断两指针是否指向同一个对象。
- 判断一个指针是否为常量 `NULL`，即判断空指针。`NULL` 是一个宏，值为 0。

例如下面代码段：

```
int a[10], *p1, *p2;
p1 = &a[1];           //p1 指向 a[1]
p2 = &a[6];           //p2 指向 a[6]
if (p1 == p2)
    cout<<"equal"<<endl;
else
    cout<<"not equal"<<endl;    //执行
if (p1 != NULL)
    cout<<"p1 is not null"<<endl;    //执行
```

### 8.1.4 关键字 `nullptr`

传统 C 语言编程中用 `NULL` 或 0 来判断空指针。C++ 引入函数重载之后，如果还用 `NULL` 就可能导致运行错误，原因是宏 `NULL` 的值是 0，其类型为 `int`，并非指针类型。

下面定义 2 个重载函数：

```
void f1(char * str) {
    if (str != NULL) //应改为 nullptr
        cout << "string is " << str << endl;
    else
        cout << "str is null"<<endl;
}
void f1(int a) { cout << "int a is " << a << endl; }
```

函数调用 `f1(NULL)` 将执行哪一个函数？程序员原意是用空指针调用第 1 个函数，但实际上执行了第 2 个函数。

C11 引入 `nullptr` 作为空指针常量，以替代传统的宏 `NULL`，尤其在作为函数实参时，应采用 `f1(nullptr)` 来防止错误执行。

## 8.2 指针与结构

结构作为一种自定义类型，也可说明指针变量。指针变量也可作为结构的成员。

### 8.2.1 结构的指针

说明一个结构类型的指针与基本类型一样。假设已定义一个结构 `Employee`，就能定义该类型的指针，例如：

```
Employee emp, * ep1 = &emp;    //说明一个指针 ep1 指向一个结构变量 emp
```

通过一个结构指针变量，就能访问该结构变量内的各个成员。格式为：

<结构指针变量> -> 成员名

其中，“->”是一种成员访问运算符(中间不能有空格)，其左边必须是一个结构或类的指针变量。成员访问运算符的优先级高于间接引用运算符\*和求址运算符&。如果用间接引用运算符，有一种等价形式如下：

(\*<结构指针变量>).成员名

注意上面表达式中不能缺少括号，这是因为成员访问运算符“.”的优先级高于间接引用运算符“\*”。这种用法比较少见。例如：

```
ep1 -> sex = 'f'      //通过指针来访问结构的成员
(*ep1).sex = 'f'     //等价方式
```

### 例 8-3 结构体指针的说明与使用

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

struct Date{                                //定义生日类型的结构体
    short year,month,day;
};

struct Employee{                            //定义职工信息结构类型
    char num[5],name[20];
    char sex;
    Date birthday;
    float salary;
};

void printAnDate(Date *d){                  //结构指针作为函数形参
    if (d != NULL)
        cout<<d->year<<'. '<<d->month<<'. '<<d->day;
}

void printTitle(){
    cout<<setw(5)<<"编号"<<setw(10)<<"姓名"<<setw(5)<<"性别";
    cout<<setw(13)<<"出生日期"<<setw(8)<<"工资"<<endl;
}

void printAnEmp(Employee * emp){           //结构指针作为函数形参
    if (emp == NULL) return;
    cout<<setw(5)<<emp->num<<setw(10)<<emp->name;
    cout<<setw(5)<<(emp->sex == 'f'? "女":"男");
    cout<<setw(5)<<' ';
    printAnDate(&emp->birthday);           //A
    cout<<setw(8)<<emp->salary<<endl;
}

int main(void){
    Employee e1 = {"024","李四",'f',{1977,8,3},4567};
    Employee *ep = &e1;
    printTitle();
    printAnEmp(ep);                        //B
    system("pause");
    return 0;
}
```

执行程序，输出如下：

编号	姓名	性别	出生日期	工资
024	李四	女	1977.8.3	4567

上面 printAnDate 函数有一个指针形参 Date \*d，表示该函数将接受一个 Date 变量的地

址。在 A 行调用时,用`&emp->birthday`表达式来计算 `emp` 所指结构变量的 `birthday` 成员的地址。

函数 `printAnEmp` 有一个指针形参 `Employee *emp`,表示该函数将接受一个 `Employee` 变量的地址。在 B 行调用时,用一个 `ep` 指针作为实参,调用前已指向一个 `Employee` 结构变量 `emp`。这是一个 36 字节的“大”对象,而传递给形参的只是 4 字节的地址。这里的函数与前一章相比,具有更高计算效率。

通过结构指针访问其成员,在使用函数库时也有用,下面程序显示当前日期、时间和星期。

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <time.h>
using namespace std;
int main(){
    time_t ltime = time(NULL);           //取得当前时间, time_t 是当前秒单位计时
    tm * now = localtime(&ltime);        //转换为本地时间, tm 是一个结构类型
    int year = now->tm_year + 1900;       //取得当前年份
    int month = now->tm_mon + 1;          //取得当前月份, 0-11
    int day = now->tm_mday;               //取得当前日, 1-31
    int hour = now->tm_hour;
    int min = now->tm_min;
    int sec = now->tm_sec;
    int week = now->tm_wday;              //0-6, 0 是周日, 1 是周一
    cout<<year<<"-"<<month<<"-"<<day<<" "<<hour<<":"<<min<<":"<<sec<<" 星期
"<<week<<endl;
    system("pause");
    return 0;
}
```

注意,第 1 行宏定义不可缺少,因为 VS2015 对 `localtime` 函数调用报错,出于安全性考虑。

### 8.2.2 指针作为结构成员

指针变量可以作为结构类型的成员,使结构变量能通过指针来关联一个变量。

修改上面 `Employee` 结构类型,让每个职员都能知道自己的部门经理 `manager` 是哪一名职员。为该结构添加一个成员 `manager`,但不是 `Employee` 类型,因为结构不能嵌套定义,但能定义 `Employee` 的指针。

```
struct Employee{
    ...
    Employee *manager; //添加一个指针变量,指向一个 Employee 变量
};
```

下面代码可确定一名雇员如何作为另一名雇员的经理。

```
Employee emp1;
Employee emp2;
emp1.manager = &emp2;
```

设计一个函数,能为一组职员设置一名经理,也能为一组职员撤销经理。这里将用到结构的数组。完整编程如下。

例 8-4 指针作为结构的成员。

```
#include <iostream>
#include <iomanip>
```

```

using namespace std;
struct Date{
    short year,month,day;
};
struct Employee{
    char num[5],name[20];
    char sex;
    Date birthday;
    float salary;
    Employee *manager;           //一个指针成员
};
void printAnDate(Date *d){
    if (d != NULL)
        cout<<d->year<<'. '<<d->month<<'. '<<d->day;
}
void printTitle(){
    cout<<setw(5)<<"编号"<<setw(10)<<"姓名"<<setw(5)<<"性别";
    cout<<setw(13)<<"出生日期"<<setw(8)<<"工资"<<setw(12)<<"经理姓名"<<endl;
}
void printAnEmp(Employee * emp){
    if (emp == NULL) return;
    cout<<setw(5)<<emp->num<<setw(10)<<emp->name;
    cout<<setw(5)<<(emp->sex == 'f'? "女":"男");
    cout<<setw(5)<<' ';
    printAnDate(&emp->birthday);
    cout<<setw(8)<<emp->salary;
    cout<<setw(12)<<(emp->manager==NULL?"空":emp->manager->name)<<endl;
}
void printEmps(Employee emps[], int n){           //打印一组职员
    printTitle();
    for(int i = 0; i < n; i++)
        printAnEmp(&emps[i]);                     //打印一名职员
}
void setManager(Employee emp[], int n, Employee *mgr){//为一组职员设置经理
    for(int i = 0; i < n ; i++)
        emp[i].manager = mgr;
}
int main(void){
    Employee emps[] = { {"021","张三",'m',{1983,3,4},3456},
                        {"024","李四",'f',{1977,8,3},4567},
                        {"027","王五",'m',{1970,9,2},5432}};

    printEmps(emps, 3);           //显示初始记录
    setManager(emps, 3, &emps[2]); //设置经理
    printEmps(emps, 3);
    setManager(emps, 3, NULL);     //撤销经理
    printEmps(emps, 3);
    system("pause");
    return 0;
}

```

执行程序，输出如下：

编号	姓名	性别	出生日期	工资	经理姓名
021	张三	男	1983.3.4	3456	空
024	李四	女	1977.8.3	4567	空
027	王五	男	1970.9.2	5432	空
编号	姓名	性别	出生日期	工资	经理姓名
021	张三	男	1983.3.4	3456	王五



024	李四	女	1977.8.3	4567	王五
027	王五	男	1970.9.2	5432	王五
编号	姓名	性别	出生日期	工资	经理姓名
021	张三	男	1983.3.4	3456	空
024	李四	女	1977.8.3	4567	空
027	王五	男	1970.9.2	5432	空

上面 `setManager` 函数为一组职员设置一名经理，第 3 个形参是一个指针。调用时如果实参是一名职员的地址，就设置其为经理，如果实参是一个空指针，就表示撤销这些职员的经理。在结构类型中通过指针成员 `manager` 表示实例之间的语义关联，如下图所示。

上面 `printEmps` 函数和 `setManager` 函数都有结构数组作为形参。在调用函数时，并非将整个数组的内容传递给形参，而是仅传递了数组的指针。

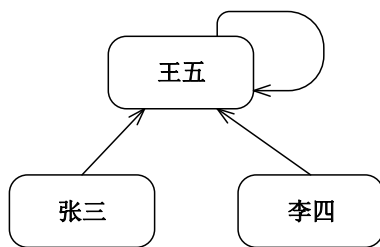


图 8-5 指针表示实例之间关联

### 8.3 指针与数组

指针与数组之间有密切关系。一个数组的名字本身就是一个指针，可用指针来访问数组元素。可以定义指针的数组，每个元素都是一个指针。指针与字符串密切相关，指针可用于定义并处理字符串。也能定义一个指针指向一个数组，即数组的指针。还能定义一个指针指向另一个指针，即二级指针，二级指针能用来访问二维数组。

#### 8.3.1 用指针访问数组

一般情况下，使用数组名加下标来访问数组元素，数组名本身就是头一个元素的地址，下标要换算为偏移地址来确定元素的位置，因此可直接用指针来访问数组。

##### 1. 一维数组

一个数组名就是一个指针，其值就是该数组在内存中的首地址，这是一个常量，不许改。

例如：`int a[5] = {1,2,3,4,5};`

定义了一个数组 `a`，那么 `a` 的类型是什么？

执行 `cout<<typeid(a).name();`语句可以看到 `a` 的类型为“`int *`”。在系统内部，`a` 只是一个 `int` 指针，它自己并不知道是 5 个元素组成的数组。

数组名 `a` 表示了该数组的首地址，即第 0 个元素的地址。由指针的加法运算规则可知，`a+1` 就表示了第 1 个元素的地址。同理，`a+i` 表示第 `i` 个元素的地址 ( $0 \leq i \leq 4$ )。

进一步，由间接引用运算符的语义可知，`*(a+i)` 就表示了第 `i` 个元素，即 `a[i]`，因此有下面等价式：

`a[i] == *(a+i)`                      //两者都表示了第 `i` 个元素

在上面等价式两侧都进行求址运算&, 就能得到下面等价式:

```
&a[i] == &*(a+i) == a+i    //都表示第 i 个元素的地址
```

上面使用了前面介绍的一个等价式: 对于任何一个指针 p:  $\&*p == p$

上面这些等价式还将用于二维数组的访问。

**例 8-5** 用指针常量来访问一维数组元素。

```
#include <iostream.h>
void main(void){
    int a[] = {1,2,3,4,5};
    for(int i = 0; i < 5; i++){
        cout<<"a["<<i<<"]="<<*(a + i)<<" at "<< a + i <<endl;
    }
}
```

执行程序, 输出如下:

```
a[0]=1 at 0x0012FF6C
a[1]=2 at 0x0012FF70
a[2]=3 at 0x0012FF74
a[3]=4 at 0x0012FF78
a[4]=5 at 0x0012FF7C
```

注意, 输出的地址值在不同机器上或者在不同时刻执行, 输出值是不同的。

使用指针变量也能访问一维数组元素。例如:

```
#include <iostream>
using namespace std;
int main(void){
    int a[] = {1,2,3,4,5};
    int *p = a;           //指针变量 p 指向数组 a 的首地址
    for(int i = 0; i < 5; i++, p++){
        cout<<"a["<<i<<"]="<<*p<<" at "<<p<<endl;
        system("pause");
        return 0;
    }
}
```

执行程序, 输出结果与上面类同。

**例 8-6** 用指针对一维数组进行降序排序。

```
#include <iostream.h>
void swap(int *p, int *q){           //用指针来交换两个值
    int t = *p;
    *p = *q;
    *q = t;
}

void bubbleSort(int *a, int n){      //冒泡降序
    for(int i = 0; i < n-1; i++){
        for(int j = 0; j < n-i-1; j++){
            int *p = a + j;
            if(*p < *(p+1))
                swap(p, p+1);
        }
    }
}

void print(int *a, int n){           //打印各元素
    for(int i = 0; i < n; i++){
        cout<<*a++<<" ";
        cout<<'\n';
    }
}

void main(void){
    int b[10] = {94,-35,56,48,80,-12,38,0,-9,58};
```

```

    bubbleSort(b, 10);
    print(b, 10);
}

```

执行程序，输出如下：

```
94 80 58 56 48 38 0 -9 -12 -35
```

## 2. 二维数组

二维数组也是一个指针。例如：

```

int b[3][4]={ {11,12,13,14},
               {21,22,23,24},
               {31,32,33,34}};

```

定义了一个二维数组 b，那么 b 的类型是什么？

执行 `cout<<typeid(b).name();` 可以看到 b 的类型为 `int (*)[4]`，这是指向一维数组 `int[4]` 的一个指针，称为一维数组的指针。它自己不知道有多少行，只知道每行有 4 列。这意味着 `b+1` 表示移动指针到下一行，即跳过 `sizeof(int)*4` 个字节。这种一维数组指针也被称为行指针。

二维数组 b 可看成是一个一维数组，它有 3 个元素：`b[0]`、`b[1]`、`b[2]`。每个元素 `b[0]`、`b[1]`、`b[2]` 又分别是一个一维数组，分别有 4 个 `int` 元素，形成下面 3 行 4 列的形式：

```

b[0]   b[0][0] b[0][1] b[0][2] b[0][3]
b[1]   b[1][0] b[1][1] b[1][2] b[1][3]
b[2]   b[2][0] b[2][1] b[2][2] b[2][3]

```

`b[0]`、`b[1]` 和 `b[2]` 既然表示一维数组，那么 `b[0]`、`b[1]` 和 `b[2]` 就表示对应一维数组的首地址，因此，

`b+0` 与 `b[0]` 的值相同，都表示数组 b 的首地址，即第 0 行第 0 列元素的地址；

`b+1` 与 `b[1]` 的值相同，都表示第 1 行第 0 列元素的地址；

...

`b+i` 与 `b[i]` 的值相同，都表示第 i 行第 0 列元素的地址 ( $0 \leq i \leq 2$ )。

虽然 `b+i` 与 `b[i]` 的值相同，但两者含义不同。

- `b+i` 和 `&b[i]` 是行指针，其类型是 `int (*)[4]`，加 1 就是下移 1 行。
- `b[i]` 和 `*(b+i)` 是列指针，其类型为 `int *`，加 1 就是后移 1 列，指向下一个 `int`。

下面导出 `b[i][j]` 元素的等价式：

```
b[i][j] == *(b[i] + j) == (*(b + i) + j) == (*(b + i))[j]
```

反复使用前面介绍的等价式 `a[i] == *(a+i)`，容易推出上面等价式。

这样访问二维数组元素就有 4 种形式。

例 8-7 用指针访问二维数组元素。

```

#include<iostream>
#include<typeinfo>
using namespace std;
void main(){

```

```

int b[3][4] = { {11,12,13,14},
                {21,22,23,24},
                {31,32,33,34}};
cout<<typeid(b).name()<<endl;
cout<<typeid(b[1]).name()<<endl;
for(int i = 0; i < 3; i++){
    cout<<b[i][0]<<" ";
    cout<<*(b[i] + 1)<<" ";
    cout<<*(b + i) + 2)<<" ";
    cout<<*(b + i)[3]<<endl;
}
system("pause");
return 0;
}

```

执行程序，输出如下：

```

int [3][4]
int [4]
11 12 13 14
21 22 23 24
31 32 33 34

```

上面例子中先输出了 **b** 和 **b[1]** 的类型名称，然后分别用 4 种形式来输出各元素，每一列用一种形式。虽然有多种形式来访问二维数组元素，但最简单仍是下标 **b[i][j]** 形式。

### 3. 数组的指针

从前面例子可以看到，二维数组 **b** 的类型是指向一维数组的指针。那么数组的指针是什么？它是单个指针的一种类型，用来指向一维数组。在定义一维数组的指针变量时，必须确定一维数组的类型和大小。例如，`int (*bp)[4]` 就是定义了一个数组的指针，指针 **bp** 指向 `int[4]` 的数组，**bp** 的类型就是 `int (*)[4]`。

一维数组的指针常作为矩阵的行指针来访问二维数组。例如：

```
int b[3][4];
```

```
int (*bp)[4] = b; //定义 bp 指针，可指向 int[4]，初始化指向 b
```

接下来 **bp** 的用法就与 **b** 一样，可用 4 种形式来访问二维数组 **b**。完整编程如下：

```

#include<iostream>
#include<typeinfo>
using namespace std;
int main(){
    int b[3][4] = { {11,12,13,14},
                    {21,22,23,24},
                    {31,32,33,34}};

    int (*pb)[4] = b; //定义一个数组的指针
    cout<<typeid(pb).name()<<endl; //输出 int (*)[4]
    for(int i = 0; i < 3; i++){
        cout<<pb[i][0]<<" ";
        cout<<*(pb[i] + 1)<<" ";
        cout<<*(pb + i) + 2)<<" ";
        cout<<*(pb + i)[3]<<endl;
    }
    system("pause");
    return 0;
}

```

```
}
```

**bp** 是一个变量，可指向另一个 **n** 行 4 列的 **int** 数组。

数组的指针可以指向一个二维数组，在访问元素时也与二维数组名一样。但这两种形式仍有区别。二维数组形式可定义数组的实体，并能初始化各元素，例如 `int b[][4]={...};`。而数组的指针只是单个变量，它不能用来定义数组的实体，也不能初始化各元素，只能指向已定义的某个数组。

数组的指针常用来作为函数的形参，来代替二维数组的形式。我们在第 6 章介绍过，二维数组作为函数形参的例子，下面函数打印 **row** 行 4 列的矩阵。

```
void print2D(int a[][4], int row){
    for(int i = 0; i < row; i++){
        for(int j = 0; j < 4; j++){
            cout<<a[i][j]<<'\\t';
        }
        cout<<endl;
    }
}
```

第 1 个形参可以改为数组的指针形式：

```
void print2D(int (*a)[4], int row)
```

函数体不用改变，调用方式也不用改变。

#### 4. 传递任意行任意列的矩阵

如果你想打印一个 3 行 3 列矩阵，就不能调用上面 **print2D** 函数，因为它只能处理 **n** 行 4 列的矩阵。如果你想设计一个函数来打印任意行、任意列的矩阵，有一种简单办法，就是将二维数组转换为一维数组，传递给函数形参。函数中再将一维数组当做二维数组来处理。

例 8-8 用函数处理任意行任意列的矩阵

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

void print2D(int *a, int row, int col){    //打印 row 行 col 列的 int 矩阵
    for(int i = 0; i < row; i++){
        for(int j = 0; j < col; j++){
            cout<<a[i * col + j]<<'\\t';    //A
        }
        cout<<endl;
    }
}

int main(void){
    int a[][4]={{1,2,3,4},
                {3,4,5,6},
                {5,6,7,8}};

    int b[4][3],i,j;
    print2D((int *)a, 3, 4);                //B 打印 3 行 4 列的矩阵
    for(i = 0; i < 4; i++)                    //实现矩阵转置
        for(j = 0; j < 3; j++)
            b[i][j] = a[j][i];
    print2D((int *)b, 4, 3);                //C 打印 4 行 3 列的矩阵
    system("pause");
    return 0;
}
```

上面定义了一个函数 **print2D** 来打印 **row** 行 **col** 列的 **int** 矩阵，第一个形参是一个 **int** 指

针,也可等价改变为 `int a[]` 形式。函数体中 A 行将 i 行 j 列的二维下标 `a[i][j]` 映射到一维下标 `a[i * col + j]`,这是将一维数组转换为二维数组的关键,它利用了二维数组元素按行按列连续存储的特点。

在 `main` 函数中两次调用了该函数,调用前要将二维数组强制转换为 `(int *)`,即一维数组。

要传递任意行任意列的矩阵,还有一种办法是采用二级指针,即指针的数组,本章后面介绍。

### 8.3.2 指针与字符串

在第6章介绍了用字符数组来存储字符串,本节介绍用指针来定义和处理字符串的方法。

#### 1. 用指针定义字符串

利用字符的指针变量可定义字符串常量。例如:

```
char *s1 = "C++ Programming";
```

注意区别于下面用字符数组定义字符串的形式:

```
char s2[] = "C++ Programming";
```

上面两条语句分别定义了两个字符串。这两种定义的差别如下:

**1、存储方式不同。**`s1` 串的含义是,先在内存中申请一块空间并存入串值,然后将头元素地址返回给 `s1`。如果另一个字符指针(如 `s3`)也说明相同串值,就不会重新申请空间,只是返回相同地址。这种存储方式称为“串池 `string pooling`”,其中串值不能更改。

而 `s2` 是一个字符数组,其串值放入一个独立的不共享的存储空间中。

**2、名字的可变性不同。**指针 `s1` 是一个变量,可以改变。例如:

```
s1 = s2; //s1 指向 s2, s1 原先的字符串就不能再访问
```

```
s1 = "Java Programming"; //s1 又指向新的常量串
```

而 `s2` 是一个常量,不能改变。

**3、内容的可变性不同。**`s1` 串的内容不许更改,无论是否用 `const` 说明。用字符数组定义的 `s2` 串内容可改变,例如:

```
strcpy(s1, "abc"); //编译不报错,执行出错
```

```
strcpy(s2, "Java Programming"); //可以执行,但可能越界了
```

如果需要一个字符串常量,可用 `const` 来修饰。这样若修改该串值编译时就提前报错。

```
const char * s1 = "C++ Programming";
```

C11 建议用 `auto` 来说明更简单,也能起到相同作用:

```
auto s1 = "C++ Programming"; // const char *
```

C11 也支持宽串和 Unicode 串的 `auto` 定义方式:

```
auto s2 = L"C++程序设计"; // const wchar_t*
```

```
auto s3 = u"hello"; // const char16_t*
```

```
auto s4 = U"hello"; // const char32_t*
```

```
auto s5 = R("Hello"); // raw const char*
```

## 2. 用指针处理字符串

用指针来处理字符串，比数组下标处理方式更灵活。

### 例 8-9 用指针处理字符串

```
#include<iostream>
using namespace std;
int strLength(const char * s){
    if (s == NULL) return -1;
    const char *s1 = s;
    while(*s != 0)
        s++;
    return s - s1;
}
int strcpy(char *to, const char *from, int n){
    if (to == NULL || from == NULL || n <= 0)
        return -1;
    int i = 0;
    while (i < n - 1 && *from != 0){
        *to++ = *from++;
        i++;
    }
    *to = 0;
    return i;
}
int main(){
    char *s1 = "Java Programming";
    char s2[] = "C++ Programming";
    s1 = s2;
    s1 = "C/C++ Programming";
    cout<<strLength(s1)<<": "<<s1<<endl;
    s1 += 6;
    cout<<strLength(s1)<<": "<<s1<<endl;
    cout<<strLength(s2+4)<<": "<<s2+4<<endl;
    cout<<strLength("")<<endl;
    char *s3= NULL;
    cout<<strLength(s3)<<endl;
    char s4[8] = "No Use";
    cout<<strcpy(s4, s1, 8)<<endl;
    cout<<s4<<endl;
    system("pause");
    return 0;
}
```

设计了一个函数 `strLength` 来计算一个字符串的长度，相当于库函数 `strlen`。函数形参 `s` 是一个常量字符指针，返回一个 `int` 表示长度。如果调用实参是一个空指针，就返回-1。函数体中第一条语句检查形参是否为空指针，即指针值是否为 0，这是指针作为形参的一般处理方式。然后移动 `s` 指针直到尾 0 处，然后返回相对移动位置作为串的长度。注意 `const` 修饰符确保实参字符串不会在函数中改变其内容，但形参 `s` 仍然可以改变。

第二个函数是一个字符串拷贝，相当于库函数 `strcpy`，不同在于返回实际拷贝的字符个数。在 6.4.4 节用字符数组实现了该函数，现在提供一个字符指针的实现。差别在于要检查字符指针是否为空，以及移动形参指针来读写字符。

在 `main` 函数中验证了字符指针定义字符串的特点，可以看到字符指针可改变。测试了前

面定义的两个函数。

执行程序，输出如下：

```
17:C/C++ Programming
11:Programming
11:Programming
0
-1
7
Program
```

读者可自行尝试其它测试。

注意，如果一个函数的一个形参是 `char*`，它往往要求函数调用提供一个字符串实参，而不是提供指向单个字符的地址。字符指针 `char*`就意味着字符串，这是字符指针的习惯用法。

8.3.3 指针的数组

指针也可以定义数组。如果一个数组中的元素都是同一类型的指针，则称这个数组为“指针的数组”。定义指针的数组的一般格式如下：

<类型> \* <数组名> [大小] [= {...}];

其中“<类型>\*”就是数组中各元素的类型。<数组名>是数组的标识符。[大小]确定了数组中元素的数量。大小可以为空，如果后面有初始化，按初始化元素个数来确定大小。例如：

```
int b[3][4];
int *p[] = {b[0], b[1], b[2]};
```

定义了一个指针数组 `p`，其中 `p[]`表示 `p` 是一个数组，有 3 个元素，然后与“`int*`”结合，表示每个元素都是一个 `int` 指针。此时 `p` 的类型为 `int*[3]`或 `int**`。

注意区别指针的数组与数组的指针。

表 8-1 指针的数组 vs 数组的指针

区别	指针的数组	数组的指针
语义	说明一个数组，其中每个元素都是一个指针，这些指针具有相同类型。	说明了单个指针，可指向一种数组，这种数组具有确定的类型和大小。
定义类型	<类型名> *[n]	<类型名> (*) [n]
兼容类型	n 可缺省 <类型名> * * 不兼容二维数组：<类型名> [m] [n]	n 不可缺省 兼容二维数组：<类型名> [m] [n]
用法例子	1.访问二维数组，例如： int b[3][4]; int * p[3] = {b[0], b[1], b[2]}; p 与 b 一样能访问各元素	1.访问二维数组，例如： int b[3][4]; int (*p)[4] = b; p 与 b 一样能访问各元素 2.作为函数形参来传递固定列的矩



	2. 作为函数形参来传递任意列的矩阵 3. 处理字符串的数组 4. 处理结构的数组	阵
--	---	---

## 1. 访问二维数组

对于二维数组，指针的数组也能用来访问各元素。

例 8-10 用指针的数组来访问二维数组元素。

```
#include<iostream.h>
#include<typeinfo.h>
void main(){
    int b[3][4] = { {11,12,13,14},
                    {21,22,23,24},
                    {31,32,33,34}};

    int *p[] = {b[0], b[1], b[2]};           //A
    cout<<typeid(p).name()<<endl;           //B
    for(int i = 0; i < 3; i++){
        cout<<p[i][0]<<" ";
        cout<<*(p[i] + 1)<<" ";
        cout<<*(*(p + i) + 2)<<" ";
        cout<<*(p + i)[3]<<endl;
    }
}
```

执行程序，输出如下：

```
int * [3]
11 12 13 14
21 22 23 24
31 32 33 34
```

例子中 A 行定义了一个指针的数组 `p`，并进行初始化，使其 3 个元素(即 3 个 `int` 指针)分别指向二维数组 `b` 的 3 行的头一个元素。如图 8.5 所示。

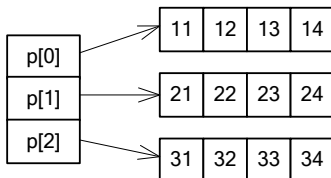


图 8-6 用数组的指针来访问二维数组

B 行显示 `p` 的类型为“`int **`”，即 `int` 的二级指针。

指针的数组 `p` 在初始化之后，就与二维数组 `b` 一样能用 4 种形式来访问各元素。

## 2. 二级指针

一个指针变量拥有自己的存储区间，因此也有自己的地址。如果一个指针变量中存放了另一个指针的地址，就称该指针变量为“指针的指针”，即二级指针。说明二级指针变量的语法格式为：

<数据类型> \*\*<变量名>

其中，\*\*指明其后的变量名为二级指针。例如：

```
int x = 32;
int *p = &x;
int **pp = &p;
```

则 x、p 和 pp 三者之间的关系如图 8.6 所示。

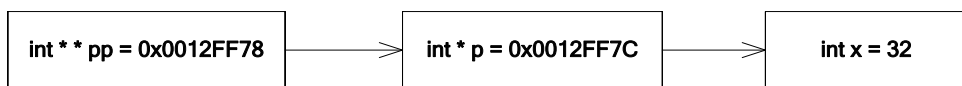


图 8-7 一级指针和二级指针

指针变量 p 是一级指针，它指向 x；指针变量 pp 是二级指针，它指向 p。通过 p 和 pp 都可以访问 x。利用间接引用运算符，\*pp 就是它所指向变量 p 的值，即 x 的地址；\*\*pp 就是它所指向的变量 p 所指向的变量的值，即 x 的值。

一级指针可用于访问一维数组。当然，二级指针可用于访问二维数组。例如：

```
#include<iostream>
using namespace std;
int main(){
    int b[][4] = { {11,12,13,14},
                  {21,22,23,24},
                  {31,32,33,34}};

    int *p[] = {b[0], b[1], b[2]};           //A 定义指针的数组，并指向二维数组
    int **pp = p;                           //B 定义一个二级指针，指向指针的数组
    for(int i = 0; i < 3; i++){
        cout<<pp[i][0]<<" ";
        cout<<*(pp[i] + 1)<<" ";
        cout<<*(pp + i) + 2)<<" ";
        cout<<*(pp + i)[3]<<endl;
    }
    system("pause");
    return 0;
}
```

二级指针可以用来访问二维数组，但不能直接指向一个二维数组。上面例子中 A 行先定一个指针的数组 p，类型为“int \* [3]”，初始化使其各元素指向二维数组 b[3][4] 的 3 行。然后 B 行将 p 赋给一个二级指针 pp，使 pp 指向指针的数组 p，这样才能通过二级指针 pp 来访问二维数组。

前面 8.3.1 节介绍了如何把一个二维数组传递给函数的一种办法。就是将二维数组转换为一维数组，即一级指针作为形参。在函数中再将二维下标映射到一维下标。主调方要用强制类型转换把二维数组转换为一维指针。下面我们采用二级指针作为函数形参，就可以免去下标映射，而能直接访问二维数组。编程例子如下：

例 8-11 用二级指针形参来传递二维数组。

```

#include <iostream>
using namespace std;
void print2D(int **a, int row, int col){
    for(int i = 0; i < row; i++){
        for(int j = 0; j < col; j++){
            cout<<a[i][j]<<'\t';
        }
        cout<<endl;
    }
}
int main(void){
    int a[][4]={{1,2,3,4},
                {3,4,5,6},
                {5,6,7,8}};

    int b[4][3],i,j;
    int *p1[] = {a[0], a[1], a[2]};          //A
    print2D(p1, 3, 4);
    for(i=0;i<4;i++)                          //实现矩阵转置
        for(j=0;j<3;j++)
            b[i][j] = a[j][i];
    int *p2[] = {b[0], b[1], b[2], b[3]};    //B
    print2D(p2, 4, 3);
    system("pause");
    return 0;
}

```

采用二级指针作为形参虽然能简化函数的设计，但主调方在调用之前要先定义一个指针的数组，并使各元素指向二维数组的各行(例子中 A 行和 B 行)，然后才能作为实参来调用函数。这样主调方编程就比较麻烦了。

上面我们提供了两种办法来传递二维数组给函数。无论哪一种办法，传递一个二维数组都需要 3 个形参。假如要设计一个函数计算两个矩阵相乘，就至少需要 6 个形参。过多的形参导致调用不方便、易出错，这不能满足高级语言进行复杂计算的要求。在后面我们将介绍采用面向对象的封装性设计，一个矩阵应作为一个对象来处理。

如果你要处理 `double` 型的二维数组，那么前面处理 `int` 数组的函数都不适用，就要再添加一组函数。在后面章节我们将采用模板(template)来处理多种类型的数据，而不用重复设计。

### 3. 字符串的数组

在实际编程中往往要处理一组相关字符串。既然一个字符指针 `char *` 就能定义一个字符串，那么字符指针的一个数组就能定义一组字符串。例如：

```
char *c1[] = {"Red", "Green", "Blue"};
```

定义了一个字符串数组，有 3 个元素。每个元素就是一个 `char*`，即一个字符串。

第 6 章中使用字符的二维数组也能定义一组字符串，例如：

```
char c2[][6] = {"Red", "Green", "Blue"};
```

这两种定义在内存存储方面有很大差异。下面通过一个例子来分析。

例 8-12 用指针的数组和二维数组两种方式来定义一组字符串。

```

#include<iostream.h>
void main(){
    char *c1[] = {"Red", "Green", "Blue"};
}

```

```

char c2[][6] = {"Red", "Green", "Blue"};
cout<<&c1[0]<<": "<<hex<<(int)c1[0]<<": "<<c1[0]<<endl;
cout<<&c1[1]<<": "<<hex<<(int)c1[1]<<": "<<c1[1]<<endl;
cout<<&c1[2]<<": "<<hex<<(int)c1[2]<<": "<<c1[2]<<endl;
cout<<hex<<(int)c2[0]<<": "<<c2[0]<<endl;
cout<<hex<<(int)c2[1]<<": "<<c2[1]<<endl;
cout<<hex<<(int)c2[2]<<": "<<c2[2]<<endl;
}

```

执行程序，输出如下：

```

0x0012FF74:425020:Red
0x0012FF78:4260c8:Green
0x0012FF7C:425024:Blue
12ff60:Red
12ff66:Green
12ff6c:Blue

```

这两种定义形式之间的差别如下图所示。

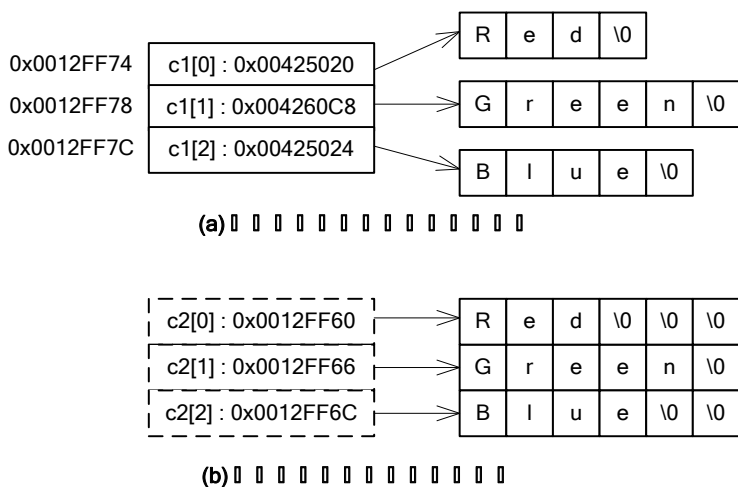


图 8-8 两种定义形式具有不同的存储结构

用指针的数组来定义一组字符串，如 `c1`。首先 `c1` 是一个数组，有 3 个元素，每个元素都指向一个字符串，而且这 3 个元素也有自己的存储空间，大小为  $4 \times 3 = 12$  字节。其次，我们注意到，这三个字符串并非连续的内存空间，而且仅占用自身所需大小， $4 + 6 + 5 = 15$  字节。定义这些串总共占用了  $12 + 15 = 27$  字节空间。注意，串的内容不可改变。

用二维数组来定义一组字符串，如 `c2`。每个串所占字节长度一样。第二维的大小起码是最长串的长度加 1。最长串“Green”长度为 5，所以定义了 6 个字节。这些长度相同的串连续存放，这样总共占用  $6 \times 3 = 18$  字节。如果串比较短，就浪费了一些空间。短串越多，浪费就越多。但如此定义的串内容可变。

用指针的数组来定义一组字符串，每个串都有自己独立的空间，其地址保存在数组的各元素之中，这些地址的排列就形成了一种“索引”，每一个索引关联一个实体，利用这种索引就能实现更灵活的计算。

我们在 6.5.2 节介绍了如何对二维数组定义的字符串进行排序。其中一次交换两个字符串就要交换两个串的实体，需要复制 3 次，计算开销比较大。现在我们用指针的数组来对一组字符串进行排序，就能避免交换串的实体，而仅交换“索引”。

例 8-13 对一个字符串数组按升序排序，并输出结果。

```
#include <iostream>
#include <string.h>
using namespace std;
void bubbleSort(char * strs[], int n){
    char *temp;
    for(int i = 0; i < n-1; i++){
        for(int j = 0; j < n-i-1; j++){
            if(strcmp(strs[j], strs[j+1]) > 0){
                temp = strs[i];           //交换指针
                strs[i] = strs[j];
                strs[j] = temp;
            }
        }
    }
}
int main(void){
    char * week[] = { "Monday",
                      "Tuesday",
                      "Wednesday",
                      "Thursday",
                      "Friday",
                      "Saturday",
                      "Sunday"};

    const int num = sizeof(week)/4;      //计算串的个数
    bubbleSort(week, num);               //调用函数完成排序
    for (auto s : week)                  //输出排序后的结果
        cout << s << endl;
    system("pause");
    return 0;}
```

执行程序，输出如下：

```
Friday
Monday
Saturday
Sunday
Thursday
Tuesday
Wednesday
```

在排序函数调用之前的 `week` 数组与调用之后的结果如下图所示。

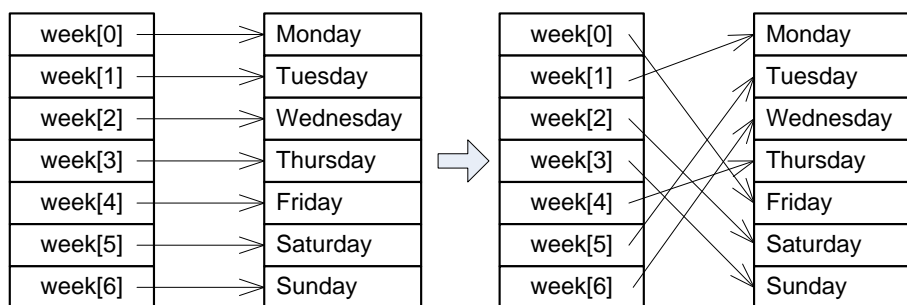


图 8-9 字符串数组的排序

排序并未改变各字符串的存储内容，而只是改变了指针数组 `week` 中的指针的值，即改变了各个串的索引，避免了对字符串内容的交换所需要的串复制，提高了执行效率。这是指针运用的一个重要例子。

字符串的数组经常作为函数的形参。例如，`main` 函数有一种重载形式如下：

```
int main(int argc, char *argv[]);
```

其中，`argc` 表示命令行中参数（字符串）的个数；`argv` 是一个字符串数组，包含了命令行输入的各个字符串。也可将 `argv` 的类型写为 “`char ** argv`”。这种 `main` 函数在命令行启动程序时能带一组参数。例如 “`copy sourcefile destinfile`” 就是一个命令行指令，将源文件 `sourcefile` 复制到目标文件 `destinfile` 中。其中 `copy` 是启动的程序或命令，`sourcefile` 和 `destinfile` 是命令参数。该命令行由三个字符串 `copy`、`sourcefile` 和 `destinfile` 组成。这些字符串都存放在 `argv` 数组中。

例 8-14 命令行参数。

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[]){
    cout<<"argc = "<<argc<<endl;
    cout<<"The program name is "<<argv[0]<<endl;
    for (int i = 1; i < argc; i++)
        cout<<"argv["<<i<<"]="<<argv[i]<<" ";
    cout<<endl;
    return 0;
}
```

执行此程序需要在 DOS 命令行输入如下命令行：`ex0814 abc xyz`

执行程序，输出结果如下：

```
F:\CTest\Debug>ex0814 abc xyz
argc = 3
The program name is ex0814
argv[1]=abc argv[2]=xyz
```

这样就能在命令行启动程序时，通过命令参数来控制程序的执行。

#### 4. 处理结构的数组

采用指针的数组所建立的“索引”尤其适合处理“大”对象的数组，例如结构的数组。我们在前面 7.1.4 节介绍了如何对一组 **Student** 结构按数学成绩进行排序。其中一次交换就要执行 3 次结构赋值，每次赋值都要复制 40 个字节(每个 **Student** 结构大小为 40 字节)，导致很大的计算开销。下面我们采用指针的数组来建立这些结构的索引，然后进行降序排序。

例 8-15 对结构数组进行排序，然后输出结果。

```
#include <iostream>
#include <iomanip>
using namespace std;
struct Student{
    char num[12], name[20];
    char sex;
    float mathscore;
};
void printTitle(){
    cout<<setw(6)<<"学号"<<setw(10)<<"姓名";
    cout<<setw(5)<<"性别"<<setw(6)<<"成绩"<<endl;
}
void printAStud(Student *s){           //指针作为形参
    cout<<setw(6)<<s->num<<setw(10)<<s->name;
    cout<<setw(5)<<s->sex<<setw(6)<<s->mathscore<<endl;
}
void bubbleSort(Student *s[], int n){   //冒泡降序算法
    Student *temp;
    for(int i = 0; i < n-1; i++)
        for(int j = 0; j < n-i-1; j++)
            if(s[j]->mathscore < s[j+1]->mathscore){
                temp = s[j];           //交换结构的指针
                s[j] = s[j+1];
                s[j+1] = temp;
            }
}
void printStuds(Student *s[], int n){   //打印一组学生
    printTitle();
    for(int i = 0; i < n ; i++)
        printAStud(s[i]);
}
int main(void){
    Student st[] = {"001","Wangping",'f',84},{ "002","Zhaomin",'m',64},
                   {"003","Wanghong",'f',54},{ "004","Lilei",'m',92},
                   {"005","Liumin",'m',75},  {"006","Meilin",'m',74},
                   {"007","Yetong",'f',89},  {"008","Maomao",'m',78},
                   {"009","Zhangjie",'m',66},{ "010","Wangmei",'f',39}};
    const int num = sizeof(st) / sizeof(Student); //计算元素的个数
    Student *stp[num];           //A 定义一个指针的数组
    for (int i = 0; i < num; i++) //B 使数组中各指针元素指向对应的结构变量
        stp[i] = &st[i];
    printStuds(stp, num);        //先打印排序前的名单
    bubbleSort(stp, num);        //排序
    cout<<"按成绩从高到低排序:"<<endl;
    printStuds(stp, num);        //打印排序后的名单
    system("pause");
}
```

```
    return 0;  
}
```

执行程序，输出如下：

学号	姓名	性别	成绩
001	Wangping	f	84
002	Zhaomin	m	64
003	Wanghong	f	54
004	Lilei	m	92
005	Liumin	m	75
006	Meilin	m	74
007	Yetong	f	89
008	Maomao	m	78
009	Zhangjie	m	66
010	Wangmei	f	39

按成绩从高到低排序：

学号	姓名	性别	成绩
004	Lilei	m	92
007	Yetong	f	89
001	Wangping	f	84
008	Maomao	m	78
005	Liumin	m	75
006	Meilin	m	74
009	Zhangjie	m	66
002	Zhaomin	m	64
003	Wanghong	f	54
010	Wangmei	f	39

上面排序函数和打印函数都有结构指针的数组作为形参，每次函数调用只传递 4 字节的地址。在排序函数中交换的是指针，而不是结构实体。处理结构的数组要先建立各结构变量的“索引”。A 行先定义了一个结构指针的数组，B 行使其中各指针元素指向各结构变量，使索引关联实体。然后再调用排序和打印函数。程序中没有改变原先结构数组 st 实体，只是改变了索引。运用指针能提高计算效率，主要是用指针传递大对象和指针数组作为索引来实现。

读者可尝试添加函数对学号、姓名、性别进行排序。

## 8.4 指针与函数

指针的一个重要作用就是给函数传递“大”对象。指针不仅可作为函数的形参，也能作为函数的返回值。函数也有指向自己的指针，通过函数的指针也能调用函数。

### 8.4.1 指针作为形参

指针作为函数形参，本章前面已出现多次。指针作为函数形参的传值方式称为地址传递，适用于传递“大”对象(如结构变量)或者一个数组。下面总结指针形参的各种用法和要点。

#### 1. 用数组名作为形参和实参

第 6 章介绍过。用数组名作为形参和实参时，实际上是形参和实参共用内存中的一个数组。如果在被调用函数中修改了某个数组元素的值，那么在调用方函数中也能反映出来。

#### 2. 数组和指针交替作为形参和实参

由于数组的本质就是指针，数组名既可作实参也可作形参。同样指针变量既可作实参也



可作形参。其功能都是使指针变量指向数组的首地址。

如果一个指针作为函数形参,就要明确调用方实参应提供一个数组还是一个对象的地址。例如一个函数原型为 `void printAStud(const Student *s)`,其功能是打印 `s` 所指向的一个 `Student` 结构变量,而不是处理 `Student` 结构的一个数组。如果调用实参是一个数组,编译不会出错,但只处理头一个元素。这并不是该函数的正确用法。

字符指针作为形参,一般是要求实参提供一个字符串,而不是单个字符的地址。例如:  
`int strlen(const char * str)`。如果提供了单个字符的地址,编译通过而运行可能出错。

再例如一个函数原型为 `int sum(const int *p, int n)`,其功能是对 `n` 个连续 `int` 值求和并返回,它希望第 1 个实参能提供一个 `int` 数组,第 2 个实参说明该数组的大小。如果函数调用第 1 个实参是单个整数的地址,而且第 2 个实参不是 1,编译不会出错,但结果无法预料。

数组和指针交替作为形参和实参时,注意有以下 3 个等价关系:

- 一级指针(如 `int *`)等价于一维数组(如 `int[n]`)。
- 数组的指针(如 `int (*)[n]`)等价于二维数组(如 `int[][n]`)。
- 二级指针(如 `int **`)等价于指针的数组(`int *[n]`)。

注意,数组与指针的区别如下:

(1)一般来说,用数组可定义实体,而用指针来指向实体、传递给函数来操作实体。只有字符指针能定义字符串实体,而其它指针都不能用来定义实体。

(2)所有数组名都是常量,不能作为赋值的左值。而非 `const` 指针都可变,可作为赋值的左值。

#### 8.4.2 函数返回指针

函数可以返回指针,指向特定实体作为计算结果,但不能指向函数体内定义的非静态局部变量。例如,库函数:

```
char * strcpy(char * to, const char * from)
```

把 `from` 串的内容拷贝到 `to` 串,并返回 `to` 串的首地址。

如果一个函数的返回值为一个指针类型,函数体中就要用 `return` 语句返回一个地址值给调用方。

例 8-16 编写一个函数将一个字符串前后空格去掉,并返回结果串的首地址。

例如,对" `ab c` "处理后的结果应为"`ab c`"。

设计思路:分 3 个步骤。先收缩串尾的空格,再收缩串头的空格,最后各字符移动到串头。编程如下:

```
#include<iostream>
#include<string.h>
char * trim(char * s){
    if (s == NULL) return NULL;
    if (strlen(s) == 0) return s;
    char *tail = s + strlen(s) - 1;    //1, 收缩串尾的空格
    while (*tail == ' ') tail--;
    if(tail < s + strlen(s) - 1)
        *(tail + 1) = '\0';
```

```
char * head = s; //2, 收缩串头的空格
while (*head == ' ') head++;
char *h = s; //3, 各字符移动到串头
if(head > h)
    while (*h++ = *head++);
return s;
}
int main(){
    char s[] = " ab c ";
    cout<<strlen(s); //输出原串长
    char * s1 = trim(s);
    cout<<": "<<s1<<": "<<strlen(s1)<<endl; //输出处理之后的串及长度
    system("pause");
    return 0;
}
```

函数可以返回指针,但不能返回函数内定义的非静态局部变量的地址。这是因为函数内部非静态局部变量都存放在堆栈中。当函数返回时堆栈弹出,局部变量都被撤销,此时再通过返回的指针来访问已被撤销的数据,就会导致严重错误。

用函数返回指针,有如下几种情形:

- 返回的指针就是某个形参指针,如库函数 `strcpy` 就是返回第一个形参指针。
- 指向某个局部静态变量,比较少见。
- 指向某个全局变量,几乎不用,因为全局变量可直接访问,用不着通过指针来访问。
- 指向动态创建的对象,后面将介绍。

指针作为形参可返回计算结果。引用变量作为形参也能起到相同作用。

### 8.4.3 函数的指针

在程序运行时,每个函数都有自己的存储地址,因此函数也有自己的指针。所谓函数指针就是函数在运行时刻的内存地址。本质上,一个函数名就是一个指针常量,它指向该函数代码的首地址。一个函数调用就是找到该函数名所表示的代码地址,将实参按一定次序压入堆栈,再启动执行代码。

函数指针变量能指向不同名称的函数(但形参与返回值都确定),然后就能用该指针来调用函数。函数指针提供编程灵活性。

#### 1. 函数指针的定义与操作

定义函数指针的语法格式为:

<返回类型> (\*<指针变量名>)(<形参表>) [ = <函数名>];

其中,<返回类型>为这类函数返回值的类型;<指针变量名>就是指针名,注意指针名和\*要用圆括号括起来;<形参表>也要用圆括号括起来。<形参表>和<返回类型>决定了该指针变量可指向的一类函数。例如:

```
int (*fp1)(int *, int);
void (*fp2)(void);
```

`fp1` 是指向一类函数的一个指针变量,此类函数有两个形参,且返回 `int`。`fp2` 也是一个函

数的指针变量，它所指向的函数没有形参，也没有返回值。

用 `typeid(fp1).name()` 可看到 `fp1` 的类型为 “`int (__cdecl*)(int *,int)`”。其中 “`__cdecl*`” 是 VC 扩展的一个关键字，说明对 C/C++ 函数的一种缺省调用方式。

实际上，用 `typeid(函数名).name()` 可以看到任何一个已定义函数的类型。

如果程序中多次使用同一类函数指针，而且函数类型又比较长，为了方便使用，常用 `typedef` 为一类函数起一个同义词，然后就可以多次使用它来说明函数指针变量。例如：

```
typedef int (*FP1)(int *, int);    //FP1 就是一类函数的类型同义词
FP1 fp1;                          //用类型同义词来说明函数指针变量
```

在说明函数指针变量的同时，可以进行初始化，也可用赋值语句使指针指向某个函数。

如何对函数指针变量赋值？可以把具有相同形参表和返回类型的函数名，或者用取地址运算符作用于一个函数名，赋给一个函数指针变量。例如：

```
int fun(int * a, int n){ ... }
void fun1(void) { ... }
fp1 = fun;
fp2 = &fun1;
```

注意，只能将具有相同的函数形参表和相同返回类型的函数名赋给函数指针变量。例如，`fp2 = fun` 是不允许的，因形参不同。

用函数指针变量如何来调用函数？用函数指针变量名，或者间接引用运算符作用于函数指针变量，再提供实参来调用被指向的函数。下面用指针 `fp1` 调用来函数 `fun`，用 `fp2` 调用 `fun1`：

```
fp1(a, 5);                        //调用 fp1 所指向的函数 fun
(*fp1)(a, 5);                    //调用 fp1 所指向的函数 fun
fp2();                            //调用 fp1 所指向的函数 fun1
```

注意，如果没有对函数指针正确赋值就调用函数，结果不可预料。

## 2. 函数指针的应用

函数指针主要用于一组具有相同形参和返回值的多个函数的选择和调用。

例 8-17 任意输入两个操作数和一个四则运算符+/\*，完成相应运算，并给出结果。

例如，输入 “`3 * 5`”，输出 “`3 * 5 = 15`”。

```
#include <iostream>
using namespace std;
float add(float x, float y){
    cout<<x<<" + "<<y<<" = ";
    return x+y;
}
float sub(float x, float y){
    cout<<x<<" - "<<y<<" = ";
    return x-y;
}
float mul(float x, float y){
    cout<<x<<" * "<<y<<" = ";
    return x*y;
```

```

    }
    float dev(float x, float y){
        cout<<x<<" / "<<y<<" = ";
        if (y == 0){
            cout<<" 除数为 0，结果应为异常";
            return 0;
        }
        return x/y;
    }
    int main(void){
        float a, b;
        char c;
        float (*p)(float, float);           //A
        while(1){
            cout<<"输入格式: 操作数 1 运算符 操作数 2\n";
            cin>>a>>c>>b;
            switch(c){                       //B
                case '+': p = add; break;
                case '-': p = sub; break;
                case '*': p = mul; break;
                case '/': p = dev; break;
                default:  cout<<"输入出错!\n"; continue;
            }
            cout<<p(a, b)<<'\n';             //C
            cout<<"继续吗?(y/n):";
            cin>>c;
            if(c != 'y' && c != 'Y') break;
        }
        system("pause");
        return 0;
    }

```

程序 A 行说明了一个函数指针 p，能指向具有两个 float 形参并返回 float 的函数。前面就定义了 4 个这样的函数，分别完成加、减、乘、除运算。

B 行开始的语句根据输入的运算符，把完成不同运算的函数名赋给指针变量 p，使指针 p 指向对应的函数。然后 C 行通过指针变量 p 来调用执行函数，得到结果。

定义一个函数指针，不仅确定了一组目标函数的形参和返回值，也确定了这些函数的统一的语义。函数指针往往作为函数的形参，这样函数体中就可以通过函数指针来调用某一个函数。

例 8-18 对于一组学生，根据每个属性都能进行排序。前面介绍了使用冒泡排序算法对每个学生的数学成绩进行降序排序。现在要求根据输入情况来选择排序方式，例如：

- 选择 0，退出
- 选择 1，按学号升序排序
- 选择 2，按姓名升序排序
- 选择 3，按性别排序
- 选择 4，按数学成绩降序排序

不同的排序方式要求不同的比较，而冒泡排序算法要适应多种比较方式。可以采用函数指针来指向不同的比较函数，而排序算法不需要改变。编程如下：

```

#include <iostream.h>
#include <iomanip.h>
#include <string.h>

```

```

struct Student{
    char num[12], name[20];
    char sex;
    float mathscore;
};
typedef int (*compFP)(Student *s1, Student *s2); //A 函数指针的类型定义 compFP

void printTitle(){
    cout<<setw(6)<<"学号"<<setw(10)<<"姓名";
    cout<<setw(5)<<"性别"<<setw(6)<<"成绩"<<endl;
}
void printAStud(Student *s){
    cout<<setw(6)<<s->num<<setw(10)<<s->name;
    cout<<setw(5)<<s->sex<<setw(6)<<s->mathscore<<endl;
}
void bubbleSort(Student *s[], int n, compFP cfp){ //冒泡升序, 函数指针做形参
    if (cfp == NULL) return;
    Student *temp;
    for(int i = 0; i < n-1; i++){
        for(int j = 0; j < n-i-1; j++){
            if(cfp(s[j], s[j+1]) > 0){ //若 s[j]"大于"s[j+1]就交换
                temp = s[j];
                s[j] = s[j+1];
                s[j+1] = temp;
            }
        }
    }
}
void printStuds(Student *s[], int n){ //打印一组学生
    printTitle();
    for(int i = 0; i < n ; i++){
        printAStud(s[i]);
    }
}
int compNum(Student *s1, Student *s2){
    return strcmp(s1->num, s2->num);
}
int compName(Student *s1, Student *s2){
    return strcmp(s1->name, s2->name);
}
int compSex(Student *s1, Student *s2){
    return s1->sex - s2->sex;
}
int compMathscore(Student *s1, Student *s2){
    return int(s2->mathscore - s1->mathscore);
}

void printChoice(){
    cout<<"选择 0, 退出\n";
    cout<<"选择 1, 按学号升序排序\n";
    cout<<"选择 2, 按姓名升序排序\n";
    cout<<"选择 3, 按性别排序\n";
    cout<<"选择 4, 按数学成绩降序排序\n";
}
void main(void){
    Student st[] = {{ "002", "Wangping", 'f', 84 }, { "001", "Zhaomin", 'm', 64 },
                    { "004", "Wanghong", 'f', 54 }, { "003", "Lilei", 'm', 92 },
                    { "006", "Liumin", 'm', 75 }, { "005", "Meilin", 'm', 74 },
                    { "008", "Yetong", 'f', 89 }, { "007", "Maomao", 'm', 78 },
                    { "010", "Zhangjie", 'm', 66 }, { "009", "Wangmei", 'f', 39 } };
    const int num = sizeof(st) / sizeof(Student);
}

```

```

Student *stp[num];
for (int i = 0; i < num; i++)
    stp[i] = &st[i];
printStuds(stp, num);          //先打印排序前的名单
while(1){
    printChoice();
    int choice = 0;
    compFP fp = NULL;          //定义函数指针变量 fp
    cin>>choice;
    if (choice == 0)
        break;
    if (choice >=1 && choice <= 4){
        switch (choice){
            case 1: fp = compNum; break;          //确定具体的比较函数
            case 2: fp = compName; break;
            case 3: fp = compSex; break;
            case 4: fp = compMathscore; break;
        }
        bubbleSort(stp, num, fp);                //调用排序函数
        printStuds(stp, num);
    }
}
}

```

A 行用 `typedef` 定义了函数指针类型的一个同义词 `compFP`。

```
typedef int (*compFP)(Student *s1, Student *s2);
```

也可以用 `using` 更简单直观:

```
using compFP = int (*)(Student *s1, Student *s2);
```

这种函数就是计算两个学生之间进行某种比较的结果, 约定如下: 如果 `s1` “大于” `s2`, 返回值就大于 0; 如果 `s1` “等于” `s2`, 返回值就等于 0; 如果 `s1` “小于” `s2`, 返回值就小于 0。这里的“大于”“小于”具有抽象的语义, 为排序算法提供比较结果, 而排序算法本身无需知道“大于”“小于”的具体语义。

冒泡排序算法的最后一个形参就使用了这种函数指针类型:

```
void bubbleSort(Student *s[], int n, compFP cfp)
```

排序函数中直接调用形参 `cfp` 就能得到比较结果, 实现“升序”排序, 但自身并没有确定两个学生之间如何比较大小。

之后定义了 4 个函数 `compXxx`, 按照 `compFP` 的要求来确定形参和返回值。

在主函数中用 `switch` 语句来根据输入来动态确定具体的比较函数, 然后再调用排序函数。

这样设计体现了一般性的抽象编程与具体化编程之间的关系。排序算法实现的是升序排序, 但没有确定具体的比较依据, 只确定了调用某个比较函数来取得结果。这就是一般性的抽象的编程, 具有通用性。通用性设计一般不需要频繁改变。具体按什么属性来排序, 由各种比较函数来给出具体实现。例如 `compName` 函数按姓名升序比较, 而 `compMathscore` 函数就根据数学成绩降序比较。还可以扩展其它比较函数, 而不需要改变排序算法。

函数的指针还有其它用途, 例如可以定义函数指针的数组, 存放一组函数的地址, 就能根据需要自动运行一组函数。

## 8.5 void 指针与 const 指针

关键字 `void` 除了用于说明函数无参和无返回值之外, `void` 指针还具有通用性编程的作用。关键字 `const` 用来说明常量, 但对于指针具有更丰富的形式和语义。

### 8.5.1 void 指针

说明一个指针变量时应该说明其类型, 但如果暂时不知道具体类型, 就可以说明它是 `void` 型。`void` 指针是指向不确定类型的指针, 可指向任何类型的变量或函数, 即任何类型指针都可赋给 `void` 指针。

对于一个 `void` 指针, 在对其进行间接引用或算术运算之前, 必须将其强制转换为某一种具体类型的指针。在未转换前, `void` 指针除了可以被赋值之外, 不能执行任何与类型相关的计算, 否则结果难料。

例 8-19 `void` 指针的简单用法。

```
#include<iostream.h>
void main(){
    void *p;                                //说明一个 void 指针 p
    int i = 1, *ip;
    float f = 1.0f, *fp = &f;
    p = &i;                                //p 指向整型变量 i
    // cout<<"i="<<*p<<endl;              //A 语法错误
    cout<<"i="<<*(int *)p<<endl;           //B 输出 p 指向对象 i 的值
    cout<<"as float, i = "<<*(float *)p<<endl; //C int 1 作为 float 输出
    ip = (int *)p;                          //使 ip 指向 p 指向的变量 i
    cout<<"i="<<*ip<<endl;                //输出 ip 指向对象 i 的值
    p = &f;                                //p 指向变量 f
    cout<<"f="<<*(float *)p<<endl;         //输出 p 指向对象 f 的值
    cout<<"as int, f = "<<*(int *)p<<endl;   //D float 1 作为 int 输出
}
```

执行程序, 输出如下:

```
i=1
as float, i = 1.4013e-045
i=1
f=1
as int, f = 1065353216
```

程序 A 行中对 `void` 指针 `p` 在转换之前尝试进行间接引用, 就会导致语法错误。B 行将其转换为 `int` 指针, 能正确访问。C 行将其转换为 `float` 指针, 就是将 `int 1` 的 4 字节作为一个 `float` 进行访问, 得到 `1.4013e-045` 值。下面 D 行将 `float 1.0f` 作为一个 `int`, 输出整型值 `1065353216`。我们曾在 6.4 节用共同体来观察 4 字节的 `int` 和 `float` 的内部存储, 得到了相同结果。

对于一个 `void` 指针, 用强制类型转换可将其转换为任一种类型的指针。如果转换类型与实际类型不符, 就会出现运行错误, 因此这种转换具有潜在危险性。第 16 章将介绍更安全的强制类型转换。

例 8-20 用函数实现任意类型的两个数据交换。

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

```

void swap(void *p1, void *p2, int elemsize){           //void 指针作为形参
    char *c1 = (char *)p1, *c2 = (char *)p2;         //转换为 char*指针
    for(int i = 0; i < elemsize; i++){
        char c = c1[i];
        c1[i] = c2[i];
        c2[i] = c;
    }
}
int main(void){
    int a = 3, b = 4;
    swap(&a, &b, sizeof(int));
    cout<<a<<" "<<b<<endl;
    double x = 5.5, y = 6.6;
    swap(&x, &y, sizeof(double));
    cout<<x<<" "<<y<<endl;
    system("pause");
    return 0;
}

```

`void` 指针作为函数形参意味着通用性设计,因为它能接收任意类型的指针实参。上面 `swap` 函数实现任一种类型的两个变量之间的交换,但要求这两个变量是同一类型。第 3 个形参确定了该类型的字节数,即长度 `sizeof(类型)`。调用方实参一定要保证满足以上条件,否则结果不可预料。函数内将 `void` 指针强制转换为 `char` 指针,然后按字节数逐个字节处理。

在主函数中分别交换了 `int` 和 `double`。读者可自行测试其它类型。

C 函数库中经常用 `void` 指针来实现通用性编程。例如,在 `<stdlib.h>` 中提供了一个通用的快速排序算法 `qsort` 函数,可对任意类型数组元素进行排序。该函数就包含了 `void` 指针来指向被排序的数组,函数原型如下:

```

void qsort(void *base, size_t num, size_t width,
    int (__cdecl *compare)(const void *elem1, const void *elem2));

```

`size_t` 是用 `typedef` 定义的 `unsigned` 类型。第 1 个形参确定了被排序的数组,第 2 个形参确定了元素个数,第 3 个形参确定了元素的字节大小。最后一个形参是一个函数指针,要求提供一个比较函数,此函数的形参也是 `void` 指针。

在 `<stdlib.h>` 中还有一个两分查找函数 `bsearch`,函数原型如下:

```

void *bsearch(const void *key, const void *base, size_t num, size_t width, int
    (__cdecl *compare ) (const void *key, const void *datum) );

```

第 1 个形参 `key` 指向要查找的一个值,后面形参与上面 `qsort` 一样。

下面例子先对一个 `int` 数组升序排序,然后再用二分查找方法查找某个元素。

```

#include<iostream>
#include<stdlib.h>
using namespace std;
int complnt(const void *p1, const void *p2) { //A 比较函数
    return *(int *)p1 - *(int *)p2;
}
int main() {
    int a[] = { 5,2,7,3,6,1 };
    qsort(a, sizeof(a) / sizeof(int), sizeof(int), complnt); //B 快速排序
    for (auto y : a)
        cout << y << " ";
}

```



```
cout << endl;
int b = 3;    //指定要查找的一个值 3 或 4
int *r=(int *) bsearch(&b, a, sizeof(a) / sizeof(int), sizeof(int), complnt); //二分查找
if (r)        //若返回指针不为空，则找到
    cout << *r << " found" << endl;
else
    cout << *r <<" no found" << endl;
system("pause");
return 0;
}
```

利用 void 指针实现通用性编程，执行效率高且节省内存，但指针类型转换比较容易出错，不易纠错。

对于通用性设计，另一种方案是函数模板，易于理解、易于编程，但效率相对降低且内存开销较大，将在第 13 章介绍。

### 8.5.2 const 指针

关键字 const 是一种修饰符，表示不可改变。在变量说明语句中，用 const 修饰的变量称为命名常量。下面我们介绍用 const 修饰的指针变量，主要用于函数形参或返回值。

有三种不同形式来说明 const 型指针变量或形参，如下表所示。

表 8-2 const 修饰指针变量的不同方式

const 修饰方式	约束	说明
const char *s1="abc"; char const *s1="abc";	不能通过指针 s1 改变所指内容	typeid 可见
char * const s2 = "abc"	指针 s2 不能被改变	typeid 不可见
const char * const s3="abc"	指针及其所指内容都不能改变	typeid 忽略第 2 个 const

引用类型的变量类同于指针类型的变量，所以这三种形式完全适用于引用类型变量。

const 指针主要用作函数形参，限制在函数体内不能修改指针变量的值，或不能修改指针所指向的数据值。这种修饰对于函数调用方来说是重要约定。例如 `int strlen(const char * str)`。

函数也可以返回一个 const 修饰的指针。例如：`const int * f1()`，返回值只能赋给一个 const 修饰的 int 指针，如 `const int * p1 = f1()`；通过返回的指针不能改变指针所指向的内容。

const 还将用于面向对象编程，在后面介绍。

## 8.6 动态使用内存

在实际编程中，往往要根据需要动态申请内存、使用内存，用完之后再回收内存。例如一个程序要多次打开对话框与用户交互。每次创建一个对话框时，都动态申请使用一块内存。当关闭一个对话框时，就回收这块内存。下次再创建对话框就可重复使用这块内存。这样就提高了内存使用效率。在此过程中，申请、使用和回收内存都需要指针。

传统 C 语言编程调用 `<stdlib.h>` 中的 `malloc` 和 `free` 函数来申请和回收内存，而 C++ 编程可使用 `new` 和 `delete` 来实现，下面介绍这两个运算符。

### 8.6.1 new 和 delete 运算符

在程序执行过程中，可能希望根据输入或计算而来的一个整数值来说明数组的大小。但使用数组说明语句是无法实现的。尽管可变长数组是 C99 标准之一，但 VC 系列不支持。DevC++(GCC) 可支持可变长数组。例如：

```
int n; cin>>n;
```

```
float a[n];
```

VC 编译器指出数组 `a` 说明无效，因为 `n` 不是常量。

此时可用 `new` 运算符来申请分配内存空间，例如：

```
int n; cin>>n;
```

```
float *p = new float [n]; //申请 n 个 float 空间
```

```
p[0] = 1.1; //使用动态空间
```

```
...
```

```
delete []p; //回收动态内存
```

## 1.new 运算符

`new` 运算符动态分配 `n` 个 `float` 元素的数组，并使 `p` 指向这个数组。该数组大小由 `n` 的值来确定的，所以它是动态可变的。动态分配的内存来自操作系统管理的“堆”，而不是来自函数当前栈。当程序不再使用这块内存时，要用 `delete` 命令来回收。

`new` 和 `delete` 运算符分别用于动态分配内存和动态收回。用 `new` 运算符分配内存空间，格式如下：

格式 1: `<指针变量> = new <类型>;`

格式 2: `<指针变量> = new <类型> (初始化值);`

格式 3: `<指针变量> = new <类型> [<整数表达式>]`

格式 4: `<指针变量> = new <类型> [<整数表达式 1>] [<整数表达式 2>];`

其中，`<指针变量>` 确定了一个指针。`<类型>` 就是要创建的变量的类型，通常与指针变量的类型一致。

格式 1 的功能是：请求系统分配由 `<类型>` 确定大小的一块连续内存空间，并把内存起始地址赋给指针变量。如果请求分配不成功，指针的值为 `NULL` (即 0)。`new` 表达式的类型为 “`<类型> *`”。

格式 2 除了完成第一种格式功能之外，还可将一组值作为动态变量的初始化值。这种格式不仅能用于基本数据类型，也能用于用户定义类型(结构，类)，需要定义构造函数(第 10 章介绍)。

格式 3 分配指定类型和大小的一维数组。对于动态创建的数组元素不能初始化。注意，`new` 表达式的类型仍为 “`<类型> *`”。

格式4 分配指定类型和大小的二维数组。注意，`new` 表达式的类型为“<类型>(\*)[<整数表达式 2>]”，即数组的指针，等价于二维数组。

如果分配内存不成功，则指针为空。因此在内存比较紧张或者请求一块“大”内存后应立即判断返回指针是否为空。

如果 `new` 结构类型，在分配内存之后就自动执行该结构的构造函数。

分配内存成功后，内存中仍存放随机值，因此在使用之前应初始化。如果未初始化而直接访问，就会出错而且编译器不会报错或警告。

## 2.delete 运算符

分配内存成功后就可通过该指针来使用这块空间。用完之后用 `delete` 来回收所分配的内存。`delete` 运算符用来将动态分配到的内存空间归还给系统，其格式为：

格式1: `delete <指针变量>;`

格式2: `delete [] <指针变量>;`

其中<指针变量>的值是用 `new` 分配内存空间返回的地址。

第1种格式把<指针变量>所指向的内存空间归还给系统，归还的大小就是指针变量类型的大小。第2种格式是把<指针变量>所指向的数组元素的内存空间归还给系统，可以将回收内存的大小放在方括号中，但一般不起作用。

如果 `delete` 结构的实例变量，那么就要自动执行该结构的析构函数。

如果用 `new` 分配内存而忘记用 `delete` 回收，那么可用内存就减少，称为内存泄露。在循环语句中内存泄露可能耗光内存而导致程序终止。注意，编译器无法检查是否存在内存泄露。

用 `delete` 回收之后该指针就是“挂空指针”，就不能再使用该指针来访问已回收的内存，除非再次用 `new` 来分配内存。

如果在分配内存之后该指针赋给其它指针变量，就会使多个指针都指向同一块空间。此时如果通过其中一个指针执行了 `delete`，那么所有其它指针都被“挂空”。通过任何一个挂空指针来访问已回收的内存或者再次回收都会造成不可预测的错误。

用 `delete p` 回收内存时，即便指针 `p` 为空指针，也不会导致错误。

总之，在使用动态内存时应注意防泄漏、防挂空访问或回收。

例 8-21 `new` 和 `delete` 的简单用法。

```
#include <iostream>
using namespace std;
int main(){
    int *ip1 = new int;
    *ip1 = 25;
    cout<<&ip1<<": "<<hex<<ip1<<"; *p1="<<dec<<*ip1<<endl;
    delete ip1;
    float * fp1 = new float(2.5);
    cout<<&fp1<<": "<<hex<<fp1<<"; *fp1="<<*fp1<<endl;
    delete fp1;
    float * fp2 = new float[10];
    cout<<&fp2<<": "<<hex<<fp2<<endl;
    for(int i = 0; i < 10; i++)
        fp2[i] = (float)i;
```

```

    for (i = 0; i < 10; i++){
        cout << "fp2[" <<i<<"]=" << fp2[i]<<" ";
        if((i+1) % 5 == 0 ) cout<< '\n';
    }
    delete []fp2;
    system("pause");
    return 0;
}

```

执行程序，输出如下：

```

0x0012FF7C:0x00440050;*p1=25
0x0012FF78:0x00440050;*fp1=2.5
0x0012FF74:0x00441D20
fp2[0]=0 fp2[1]=1 fp2[2]=2 fp2[3]=3 fp2[4]=4
fp2[5]=5 fp2[6]=6 fp2[7]=7 fp2[8]=8 fp2[9]=9

```

主函数中前后申请使用了 1 个 int、1 个 float 和 10 个 float 空间，用完立即回收。通过 3 个局部指针变量来访问动态空间。这 3 个指针变量都存在于函数堆栈中，我们能看到连续存放的地址。但动态申请的空间则可能重复使用，例如第 1 次申请的 int 空间回收之后，又被第 2 次申请的 float 变量使用。

何时要用动态内存？编译时无法确定，运行时才能确定，就要考虑使用动态内存。

例 8-22 实现一个动态的任意行、不等列的二维表，假定元素为 int。运行时确定行数以及每行的列数，再输入数据，打印数据并计算每行的平均值。例如输入：

```

4
3 11 12 13
2 21 22
4 31 32 33 34
1 41

```

第 1 行输入 4 表示下面有 4 行。然后每一行的第一个值表示该行的元素个数。

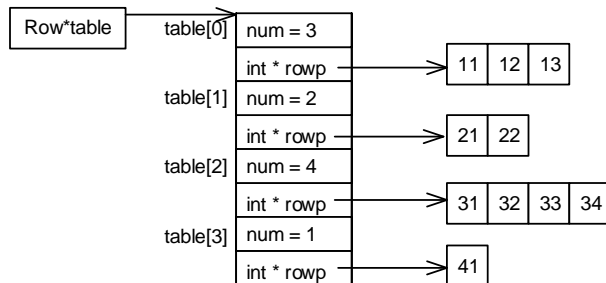


图 8-10 任意行、不等列的二维表结构

设计要点是一个结构类型 Row，表示每行元素的个数 num 和指针 rowp，使指针 rowp 指向一行数据，并用 Row 的一个数组来表示各个行。如图 8.9 所示。在建立这样一个二维表时，

要先确定行数，即确定 **Row** 数组 **table** 的元素的个数，再确定每一行的列数，即 **rowp** 所指向的 **int** 数组中元素的个数。

编程如下：

```
#include<iostream>
using namespace std;
struct Row{
    int num;
    int * rowp;
};
void print(Row *tb, int row){
    cout<<row<<endl;
    for(int i = 0; i < row; i++){
        int sum = 0;
        cout<<tb[i].num<<":";
        for(int j = 0; j < tb[i].num; j++){
            sum += tb[i].rowp[j];
            cout<<tb[i].rowp[j]<<" ";
        }
        cout<<":avg="<<(float)sum/tb[i].num<<endl;
    }
}
void del(Row *tb, int row){
    for(int i = 0; i < row; i++)           //先撤销每一行的元素
        delete [] tb[i].rowp;
    delete []tb;                          //最后撤销 Row 行
}
int main(){
    int i, j, row = 0;
    cin>>row;
    Row * table = new Row[row];           //创建 Row 行
    for (i = 0; i < row; i++){            //循环每一行
        int num = 0;
        cin>>num;
        table[i].rowp = new int[num];     //创建一行，下面根据输入填写一行数据
        table[i].num = num;
        for (j = 0; j < num ; j++){
            int k = 0;
            cin >> k;
            table[i].rowp[j] = k;
        }
    }
    print(table, row);                    //打印与计算
    del(table, row);                      //撤销内存
    system("pause");
    return 0;
}
```

执行程序，输入输出如下：

```
4
3 11 12 13
2 21 22
4 31 32 33 34
1 41
4
3:11 12 13 :avg=12
2:21 22 :avg=21.5
4:31 32 33 34 :avg=32.5
```

```
1:41 :avg=41
```

上面程序展示了利用指针表示多个数据之间的关联。

### 8.6.2 动态变量管理

编程中经常要动态管理一组变量。新元素随时会加入,已有元素随时会被撤销。关键问题有二个。一是确保新加入元素的指针不能覆盖已有元素的指针,而且能存储,以供访问或撤销。二是撤销一个元素之后应能让新元素加入。下面通过一个例子说明动态变量管理。

例 8-23 动态管理一组职员信息。建立一个定额职员表,可随时添加新职员、删除已有职员、用编号来查找某职员、打印职员信息等。

```
#include<iostream.h>
#include<string.h>
#include <iomanip.h>
struct Employee{
    char num[5],name[20];
    char sex;
    float salary;
};
int addEmp(Employee *eps[], int n, char * num, char * name,
           char sex, float salary){
    for (int i = 0; i < n; i++){
        if (eps[i] == NULL)
            break;
    }
    if (i >= n) return -1;
    eps[i] = new Employee;
    strcpy(eps[i]->num, num);
    strcpy(eps[i]->name, name);
    eps[i]->sex = sex;
    eps[i]->salary = salary;
    return i;
}
int searchEmp(Employee *eps[], int n, char * num){
    for(int i = 0; i < n; i++){
        if (eps[i] != NULL)
            if (strcmp(eps[i]->num, num) == 0)
                return i;
    }
    return -1;
}
bool removeEmp(Employee *eps[], int n, char * num){
    for(int i = 0; i < n; i++){
        if (eps[i] != NULL){
            if (strcmp(eps[i]->num, num) == 0){
                delete eps[i];
                eps[i] = NULL;
                return true;
            }
        }
    }
    return false;
}
void removeAll(Employee *eps[], int n){
    for (int i = 0; i < n; i++){
        if (eps[i] != NULL){
            delete eps[i];
            eps[i] = NULL;
        }
    }
}
```

```

    }
}

void printEmps(Employee *eps[], int n){
    cout<<setw(6)<<"编号"<<setw(10)<<"姓名";
    cout<<setw(5)<<"性别"<<setw(6)<<"工资"<<endl;
    for (int i = 0; i < n; i++){
        if (eps[i] == NULL) continue;
        cout<<setw(6)<<eps[i]->num<<setw(10)<<eps[i]->name;
        cout<<setw(5)<<eps[i]->sex<<setw(6)<<eps[i]->salary<<endl;
    }
}

void main(){
    const int count = 20;           //职员定额
    Employee *eps[count];          //定义职员表
    for (int i = 0; i < count; i++) //初始化,置空位
        eps[i] = NULL;
    addEmp(eps, count, "001", "Wangping", 'f', 2184);
    addEmp(eps, count, "002", "Zhaomin", 'm', 2164);
    addEmp(eps, count, "003", "Wanghong", 'f', 2154);
    addEmp(eps, count, "004", "Lilei", 'm', 2292);
    printEmps(eps, count);
    if (removeEmp(eps, count, "003"))
        cout<<"003 号职员被删除"<<endl;
    printEmps(eps, count);
    int k = 0;
    if ((k = searchEmp(eps, count, "004")) >= 0)
        cout<<"004 号职员被找到:"<<k<<endl;
    else
        cout<<"004 号职员未找到"<<endl;
    if (addEmp(eps, count, "005", "Liumin", 'm', 2375) >= 0)
        cout<<"005 号职员加入"<<endl;
    else
        cout<<"名额已满,不能加入"<<endl;
    printEmps(eps, count);
    removeAll(eps, count);
}

```

执行程序,输出如下:

```

编号      姓名      性别      工资
001  Wangping    f      2184
002  Zhaomin     m      2164
003  Wanghong    f      2154
004  Lilei       m      2292
003 号职员被删除
编号      姓名      性别      工资
001  Wangping    f      2184
002  Zhaomin     m      2164
004  Lilei       m      2292
004 号职员被找到:3
005 号职员加入
编号      姓名      性别      工资
001  Wangping    f      2184
002  Zhaomin     m      2164
005  Liumin      m      2375
004  Lilei       m      2292

```

主函数中前3条语句用一个 **Employee** 指针建立了一个数组,作为职员表,而且对每个指

针元素初始化为空 NULL，表示空位，可加入职员。下面一组函数都作用于这个职员表：

- 函数 `addEmp` 向职员表中添加人员，寻找空位，请求内存，加入指针，填写数据，返回下标；如果找不到空位，返回-1 表示职员表已满。
- 函数 `removeEmp` 按指定的职员编号查找，如找到就删除，置空位，并返回 `true`；如找不到就返回 `false`。
- 函数 `searchEmp` 按指定的职员编号查找所有非空位，如找到就返回下标，如找不到就返回-1。
- 函数 `printEmps` 打印所有非空位的职员信息。
- 函数 `removeAll` 删除表中所有非空位的职员。最后调用此函数来回收所有动态分配的内存。

动态变量管理的关键是建立一个指针的数组，而且贯彻一个约定：一个元素占一个位子，如果元素被撤销，那么指针元素就被置空，以便新元素能加入。每个函数都以此约定进行设计。

### 8.6.3 注意要点

在使用 `new` 和 `delete` 运算符时，应注意以下几点：

(1)如果请求内存数量较大，或者内存有限，每次用运算符 `new` 请求分配空间后，应立即判断其指针值是否为 0。若为 0，表示动态分配内存失败，此时应终止程序执行，或者进行出错处理。例如：

```
float *p = new float[7000];
if(p == 0){
    cout<<"动态分配内存不成功，终止程序执行!\n";
    exit(-1);
}
```

(2)用运算符 `new` 分配内存空间的指针应该保存起来，以便用 `delete` 运算符回收被分配使用的内存空间。如果在程序结束之前没有用 `delete` 回收动态请求的内存空间，这部分内存空间可能无法再利用。这就是“内存泄漏”问题。没有简单办法能检查程序中是否存在内存泄露。往往需要重启程序或操作系统才能恢复。这样的程序如果反复执行，将导致系统内存越来越少。这是一种隐藏的程序缺陷，不易发觉，而且系统也难以自动检查。程序员有责任管理维护内存的一致性。

(3)动态分配数组的存储空间时，不能在分配空间时进行初始化。对于结构类型的动态变量，如果要初始化，也要求编写构造函数。我们将在类中介绍如何编写构造函数。

(4)用 `new` 可动态分配多维数组空间，但要求 `new` 运算符返回的指针类型与赋值运算符左操作数类型一致，否则就必须进行强制类型转换。例如：

```
float (*pa)[20] = new float [10][20];           //A
float * pc = (float *) new float [10][20];       //B
float (*pb)[20] = new float [1][20];             //C
```

A 行 `new` 运算符分配 10 行 20 列的二维数组空间，其返回指针类型与 `pa` 一致，不用类型转换。B 行 `new` 运算符返回的指针类型是 `float(*)[20]`，而左值 `pc` 的类型是 `float*`，就要作



类型转换。C 行 `new` 运算符分配 1 行 20 列的二维数组，虽然与 `new float[20]` 分配的存储空间大小相同，但 `new` 返回的指针类型不同。

(5)用“`delete` 指针;”在释放该指针所指向的动态存储空间后，这个指针就被“挂空”，就不能再用该指针来访问这个空间，对该指针也不能再次执行 `delete` 回收，否则就会导致不可预测的错误。没有简单办法能判断一个指针是否被挂空。如果程序中有多个指针指向同一块动态存储空间时，就可能发生这种错误。

## 8.7 引用

引用(reference)是对已有变量的一种别名机制，是 C++ 提供的一种高级传递方式，主要用于函数的形参和返回值。引用类型分为左值(lvalue)引用和 C11 提出的右值(rvalue)引用两种。本节先介绍左值引用，再简单介绍右值引用。

### 8.7.1 左值引用变量

左值引用类型的变量是其它变量的别名，因此对引用变量的操作实际上就是对被引用变量的操作。当说明一个引用变量时，必须要绑定另一个变量对其初始化，除非该引用是函数形参，调用时再绑定另一个变量。说明左值引用变量的语法格式如下：

<类型> &<引用变量名> [=<变量名>];

其中，<类型>必须与<变量名>的类型相同或兼容，&指明该变量为左值引用类型。例如：

```
int x;
int &refx = x;
```

变量 `refx` 就是一个左值引用类型变量，它给 `int` 变量 `x` 起了一个别名 `refx`，即 `refx` 与 `x` 这两个名字指的是同一个内容。`refx` 称为对 `x` 的引用，`x` 称为 `refx` 的引用对象。在说明引用类型变量 `refx` 之前，被引用的变量 `x` 必须先定义。

注意，此时用 `typeid` 查看 `refx` 的类型，仍为 `int`，而不是 `int&`。

例 8-24 引用变量的简单使用。

```
#include <iostream>
#include <typeinfo>
using namespace std;
int main(void){
    int x, y = 36;
    int &refx = x, &refy = y;           //A
    refx = 12;
    cout<<"x="<<x<<"    refx="<<refx<<endl;
    cout<<"y="<<y<<"    refy="<<refy<<endl;
    refx = y;                           //B
    cout<<"x="<<x<<"    refx="<<refx<<endl;
    cout<<"&refx = "<<&refx<<" "; "<<"&x="<<&x<<endl;
    cout<<"&refy = "<<&refy<<" "; "<<"&y="<<&y<<endl;
    cout<<typeid(refx).name()<<endl;
    system("pause");
    return 0;
}
```

执行程序，输出如下：

```
x=12  refx=12
```

```

y=36  refy=36
x=36  refx=36
&refx = 0x0012FF7C; &x=0x0012FF7C
&refy = 0x0012FF78; &y=0x0012FF78
int

```

可以看出，系统并不为引用变量分配存储空间，它的存储空间就是被引用变量的空间。引用变量与被引用变量之间的绑定是一次性的。因此对于 B 行语句 `refx=y`，不能理解为“使变量 `refx` 来引用变量 `y`”，而应理解为“将 `y` 赋给变量 `refx` 所引用的变量”。此时，`refx` 就是 `x` 的别名，它们具有相同的操作语义。

一个引用变量，如 `int &refx`，用 `typeid` 查看其类型是 `int` 型，而不是 `int&` 型。

一般来说，左值引用变量只能引用变量，而不能引用字面值，但用 `const` 修饰的左值引用变量称为常量左值引用，可引用字面值或表达式(如 `a+3`)。例如：

```
const int &ri = 44;
```

`const` 引用往往用于说明函数的形参，在函数体中不能改变该引用的值，调用方可用字面值或表达式作为实参。后面将详细说明常量左值引用的特殊用途。

左值引用经常作为函数的形参，调用时实参类型应与引用类型一致，函数中可通过引用改变实参。下面函数计算 `double` 的平方值。

```

void square(double &d){           //引用作为形参
    d *= d;
}
void f(){
    double d1 = 2.2;
    square(d1);                   //调用函数
    cout<<d1<<endl;              //实参改变为其平方值
}

```

上面函数并未用 `return` 返回计算结果，函数体中通过引用改变了被引用的值。

引用作为函数形参可改变实参，因此引用形参既可做函数的输入，也可做函数的输出。

## 8.7.2 左值引用与数组、指针的关系

### 1. 引用与数组

对一个已定义的数组可说明其引用变量。例如：

```

int a[] = {1,2,3};
int (&ra)[3] = a;

```

说明了一个对 `int[3]` 数组的引用变量 `ra`，并用数组 `a` 进行初始化。下面对 `ra[i]` 的访问就等同于对 `a[i]` 的访问。

对数组可说明引用，但引用只能是单个值，不能说明引用的数组。

### 2. 引用与指针

引用与指针有相似之处。我们经常说“定义一个引用让它指向...”。引用用别名来指向特定变量，而指针是用地址来指向特定变量。它们都能指向确定类型的变量，只是定义和使用方式不同。要完成一项计算，往往既可用引用，也可用指针。

引用的指针。对于一个引用变量，可以说明其指针或地址，但本质上还是被引用的变量的指针，这是因为引用并没有自己的存储空间。例如：

```
int i = 3;
int &r = i;           //r 是 i 的引用
int *rp = &r;         //&r 是 i 的地址
```

表面上，指针 `rp` 指向引用变量 `r`，但实际上是指向 `i`。此时 `rp` 的值就是 `&i`。  
指针的引用。对于一个指针，可以说明其引用。例如：

```
int i = 3;
int * pi = &i;
int * &rpi = pi;
rpi 是一个指针的引用，等同于 pi。
```

指针的引用常作为函数形参，函数中可改变指针或指针所指内容。后面介绍例子。

不能说明引用的引用，即不存在二级引用。

至此“&”有三种含义，一是作为双目按位与运算符；二是作为单目取地址运算符；三是作为引用说明符，用于定义引用类型的变量或形参。

### 8.7.3 左值引用与函数

左值引用经常作为函数的形参和返回值，这是引用最主要用途。

#### 1. 引用作为函数形参

如果函数形参是一个引用变量，在编译时将检查实参类型是否与引用变量类型一致。在函数调用时，引用形参将绑定实参，然后在函数中就能访问实参的内容。

例 8-25 用引用形参实现两个数据的交换。

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

void swap1(int &x, int &y){
    int t=x; x=y; y=t;
}
void swap2(int x, int y){
    int t=x; x=y; y=t;
}
int main(void){
    int a = 11, b = 22, c = 33, d = 44;
    cout<<"交换前:\t a="<<a<<" b="<<b<<endl;
    swap1(a, b);                                     //A
    cout<<"交换后:\t a="<<a<<" b="<<b<<endl;         //B
    cout<<"交换前:\t c="<<c<<" d="<<d<<endl;         //C
    swap2(c, d);
    cout<<"交换后:\t c="<<c<<" d="<<d<<endl;         //D
    system("pause");
    return 0;
}
```

执行程序，输出如下：

```
交换前： a=11 b=22
交换后： a=22 b=11
```

```
交换前:  c=33  d=44
交换后:  c=33  d=44
```

上面两个交换函数中的语句以及调用方式都相同,但形参不同,结果就不同。函数 `swap1` 是引用形参,在 A 行调用 `swap1` 时使形参 `x` 和 `y` 分别作为实参 `a` 和 `b` 的别名,在函数内的交换起作用,因此在 B 行输出交换后的结果。当在函数体内改变了引用类型的形参后,就改变了实参的值,能将新的值带回给调用者。函数 `swap2` 的形参属于值传递,在函数体内虽然作了交换,但并不能改变实参 `c` 和 `d` 的值,所以 C 行和 D 行输出结果相同。

函数形参如果是一个引用变量,函数调用的实参作为被引用变量,这种传递方式就是引用传递。区别于以值传递和地址传递,引用传递是将实参起一个别名给形参,函数中对形参的操作就像对实参操作一样。下面我们总结这三种传递方式的特点:

- **以值传递**是形参赋值给形参,形参独立于实参变化,因此以值传递对于函数只是输入,而不能输出结果。
- **地址传递**的本质也是以值传递,实参地址传给形参指针,对形参指针的改变独立于实参,通过形参指针可改变实参的内容,除非用 `const` 修饰符限制它不能改变,因此地址传递对于函数是输入和输出。
- **引用传递**与地址传递有些类同,可作为函数的输入,但函数中通过引用形参可改变实参的内容,故此也可作为函数的输出。

例 8-26 用二级指针形参和指针的引用形参来实现一级指针的交换。

```
#include <iostream>
using namespace std;
void swap1(int **p, int **q){           //二级指针做形参,要求二级指针做实参
    int *t;                             //交换的是一级指针
    t = *p; *p = *q; *q = t;           //较复杂
}
void swap2(int *&p, int *&q){           //一级指针的引用做形参,要求一级指针做实参
    int *t;                             //交换的是一级指针
    t = p; p = q; q = t;               //较简单
}
int main(){
    int a = 3, b = 4;
    int *p1 = &a, *q1 = &b;
    cout<<"交换前:"<<*p1<<' ' <<*q1<<endl;
    swap1(&p1, &q1);                   //用二级指针实参调用 swap1
    cout<<"交换后:"<<*p1<<' ' <<*q1<<endl;
    cout<<"原数据:"<<a<<" "<<b<<endl;
    p1 = &a, q1 = &b;
    cout<<"交换前:"<<*p1<<' ' <<*q1<<endl;
    swap2(p1, q1);                     //用一级指针实参调用 swap2
    cout<<"交换后:"<<*p1<<' ' <<*q1<<endl;
    cout<<"原数据:"<<a<<" "<<b<<endl;
    system("pause");
    return 0;
}
```

执行程序,输出如下:

```
交换前:3 4
交换后:4 3
```

```
原数据:3 4
交换前:3 4
交换后:4 3
原数据:3 4
```

通过这个例子可以看到，引用形参与指针形参的区别。

上面两个交换函数执行相同功能，都是交换一级指针。不同的是形参及实现方式。函数 `swap1` 的形参是二级指针，实现代码中语句操作一级指针(都带\*)。函数 `swap2` 的形参是一级指针的引用，实现代码中直接操作形参(不带\*)，显然更简单易懂。

这两个函数的调用不同，对于 `swap1` 函数，要用二级指针来调用。而调用 `swap2` 函数，只需一级指针做实参，显然更简单。

用 `const` 修饰的引用形参，可用字面值作为实参来调用，当然也可变量来调用。

## 2. 引用作为函数返回值

当函数返回引用类型时，它所返回的值是某个变量的别名，相当于返回了一个变量，可对其返回值直接进行访问，即可作为表达式左值。

例 8-27 函数返回值为引用。

```
#include <iostream>
using namespace std;
int &f1(void){
    static int count = 1;          //局部静态变量
    return ++count;
}
int index;                        //全局变量
int &f2(void){ return index; }
void main(void){
    f1() = 100;                    //A
    for(int i = 0; i < 5; i++) cout<<f1()<<" "; //B
    cout << '\n';
    f2() = 100;                    //C
    int n = f2();                  //D
    cout << "n = " << n << '\n';
    f2() = 200;
    cout << "index = " << index << '\n';
    system("pause");
    return 0;
}
```

执行程序，输出如下：

```
101 102 103 104 105
n = 100
index = 200
```

函数 `f1` 返回局部静态变量 `count` 的引用，函数 `f2` 返回全局变量 `index` 的引用。A 行中的 `f1()=100`，由于赋值运算符的优先级低于函数调用，因此先执行对函数 `f1` 的调用。函数的返回值为 `count` 的引用，即 `count` 的一个别名，然后执行赋值运算，就是将 100 赋给 `count`。B 行中 5 次调用函数 `f1`，先使 `count` 的值加 1，然后返回 `count` 的引用，并输出 `count` 值。同理，C 行等同于把 100 赋给变量 `index`，D 行等同于将 `index` 的值 100 赋给变量 `n`。

注意，在主函数 `main` 中，不能直接访问静态局部变量 `count`，这是因为该变量定义在 `f1` 函数之内。通过函数 `f1` 就可以对其赋值或读取。变量 `count` 是一个静态局部变量，这种变量的特性是在函数返回后，仍保存为其分配的存储空间，具有全局生存期。

当一个函数返回一个引用时，在函数中不能返回函数内的非静态局部变量。这是因为非静态局部变量都存放在当前函数堆栈中，当函数返回时，堆栈弹出，这些局部变量就不存在了，所以对它的引用是无效的。这个规则与返回指针一样。

#### 8.7.4 右值引用&&

前面介绍的引用在 C11 之后都称为左值引用。C11 提出另一种引用，右值引用&&。

右值引用&&，是与右值引用&相对立的一种引用，就是针对右值表达式的引用。

此时区别左值和右值的概念就很重要了。

左值与右值最简单的区别是赋值语句，赋值运算符左边为左值，右边为右值。`++i` 与 `--i` 也是左值。

指针概念也有助于区别左值与右值。可以取地址的、有名字的值就是左值。反之，不能取地址的、没有名字的值就是右值。

右值引用与左值引用之间区别是，非常量左值引用不能绑定字面值和表达式，如 `3`，`a+3` 等，而常量左值引用(`const&`)与右值引用(`&`)可绑定字面值和表达式。

两者都可作为函数形参和返回值。

说明一个右值引用的语法形式只比左值引用多一个&符号。例如：

```
int a = 2;
int && rr1 = a + 3;      //表达式
auto && rr2 = 4;         //整数字面值
auto && rr3 = true;      //逻辑字面值
const auto &&rr4 = 55;    //const int&&
```

前 3 个引用变量的类型都是 `int&&`，称为非常量右值引用。

最后一个引用变量的类型是 `const int &&`，称为常量右值引用。

右值引用与左值引用一样，经常作为函数的形参或返回值。比如：

```
void f(int && r) {
    ++r;
    cout << "f(int&&)=" << r << endl;
}
```

上面函数 `f` 的形参是一个非常量右值引用。你可用一个右值表达式来调用，如 `f(a+3)`。

下面例子包含 4 个重载函数，用不同的左值引用和右值引用作为形参。

```
#include<iostream>
using namespace std;
void f(const int &r) {          //第 1 个函数，常量左值引用做形参
    cout << "f(const int&)=" <<r<< endl;
}
```

```

void f(int & r) {                //第 2 个函数，非常量左值引用做形参
    r++;
    cout << "f(int&)= " << r << endl;
}
void f(const int && r) {         //第 3 个函数，常量右值引用做形参
    cout << "f(const int&&)= " << r << endl;
}
void f(int && r) {               //第 4 个函数，非常量右值引用做形参
    ++r;
    cout << "f(int&&)= " << r << endl;
}
int main() {
    const int b = 4;
    f(b);                       //A 常量左值做实参
    int a = 3;
    f(a);                       //B 非常量左值做实参
    f(3);                       //C 常量右值做实参
    f(a + 3);                   //D 非常量右值做实参
    system("pause");
    return 0;
}

```

执行程序，输出结果如下：

```

f(const int&)=4
f(int&)=4
f(int&&)=4
f(int&&)=8

```

上面程序需要 C11 最好 C14 编译。DevC++应人工设置编译选项-std=c++1y。

A 行实参是常量左值，只能被第 1 个函数形参绑定，因此执行第 1 个函数。

B 行实参是非常量左值，可被前 2 个函数绑定，但第 2 个重载选择优先级更高，执行第 2 个函数。a 的值被改为 4。

C 行实参是常量右值，看起来应该执行第 3 个函数，但实际上执行第 4 个函数，这是因为字面值 3，C11 之后将其作为可变量，因此执行第 4 个函数。执行中将 3 改为 4。

D 行实参 a+3 非常量右值表达式，执行第 4 个函数，实参值为 7，函数体中改为 8 输出，但 a 的值 4 并非改变。

然后将上面 4 个函数逐个删除，再执行，就能得到下面一个表，描述形参引用与绑定关系的表，也能体现多个重载函数在同一个调用时执行的优先级。

表 8-3 不同引用类型与可绑定的值类型

引用类型 (以 T=int 为例)	可绑定的值类型				说明
	常量左值	非常量左值	常量右值	非常量右值	

	<code>const T b=3; f(b)</code>	<code>T a=3; f(a)</code>	<code>f(3)</code>	<code>f(a+3)</code>	
常量左值引用 <code>f(const T&amp;)</code>	Y	Y 低优先	Y 低优先	Y 低优先	全能类型， 拷贝语义
非常量左值引用 <code>f(T&amp;)</code>	N	Y 高优先	N	N	引用传递
常量右值引用 <code>f(const T&amp;&amp;)</code>	N	N	Y 中优先	Y 中优先	暂无用途
非常量右值引用 <code>f(T&amp;&amp;)</code>	N	N	Y 高优先	Y 高优先	移动语义、 完美转发

注1， 列出 4 行分别对应前面 4 种引用的形参，4 列分别对应主函数中的 4 个调用。

注2， 单元格中 Y 表示绑定可执行，N 表示不能绑定执行。

注3， 对于某一列调用，可能有多个函数能执行，根据重载选择优先级，从高到低选择。

从上面表中可以得到如下结论：

1、如果形参是右值引用&&，任何左值做实参都不能调用，除非先调用 `move` 函数将左值转换为右值后再调用。不过调用之后就不能对该左值实参的值做任何假设，只能被撤销或者重新赋值。

2、如果形参是常量左值引用，任何左值右值实参都可调用，所谓“全能类型”。重载选择最低优先。支持拷贝构造函数和赋值操作函数，详见第 10 章。

3、如果形参是非常量右值引用&&，能绑定对应的右值实参。重载选择最高优先。支持移动语义(详见第 10 章)与完美转发(详见第 13 章)。

4、貌似相同的函数调用却执行不同函数。如 `const int b = 3; f(b)`与 `f(3)`，而且每个调用都可能执行多个重载函数。说明 C11 之后类型多样性得到扩展。

## 8.8 Lambda 表达式

Lambda 表达式(下文简称 L 式)是 C11 引入的一种语言构造。其作用是以表达式的形式表示一个匿名函数，其中包含简短代码，往往作为算法函数调用的实参。其中涉及到以值传递和引用传递。第 6 章曾简单使用 L 式来处理数组，如排序、遍历等。本节说明其语法构造和基本用法。

### 8.8.1 语法构造

从语法上看，一个 L 式由 3 个主要部分组成：

[捕获](参数表) {函数体}

第 6 章有一个函数计算一个 `int` 数组元素的平均值：

```
double getAverage(int a[], int n) {
    double sum = 0;
```



```

    for_each(a, a + n, [&sum](auto x) {sum += x; });
    return sum / n;
}

```

上面例子调用了<algorithm>中的 for\_each 函数，第 3 个实参就是一个 L 式。其中：

[&sum]，采用引用捕获外边的 sum 变量，函数体中要改变它。

(auto x)，参数表中是遍历到的一个元素 x，以值传递方式，函数体中对 x 只读。

{ sum += x; }，函数体，对每个元素 x 加入 sum 变量。

1、**[捕获]**，说明函数体如何从外界捕获变量。[]表示不获取外界变量，函数体仅依赖参数表中的参数和静态作用域中的变量，如全局变量或静态局部变量。捕获外界变量有两种方式：以值捕获=、引用捕获&。

- 以值捕获=，相当于函数参数的以值传递，L 式中改变该值并不影响实参。

- 引用捕获&，相当于函数参数的引用传递，L 式中改变该值将改变实参。

下面举例说明：

```

[&total, factor]    //引用捕获访问 total，以值捕获访问 factor
[factor, &total]    //与上面一样，次序无所谓
[&, factor]        //用引用访问所有，但 factor 是以值捕获访问
[factor, &]        //与上面一样
[=, &total]        //以值捕获访问所有，但 total 是引用捕获
[&total, =]        //与上面一样

```

如果子句包含&，就不允许再出现“&标识符”；

如果子句包含=，就不允许再出现“=标识符”或 this；

标识符和 this 在[]中最多出现 1 次。

以值捕获与引用捕获更深层的区别是，前者是在定义时静态捕获变量当前值，而后者是在执行过程中动态捕获变量的当前值。如果先定义，后执行，而且中间改变被引用捕获的变量，后执行时要动态捕获改变后的值。例如：

```

int i = 3, j = 5;
function<int(void)> f = [i, &j] { return i + j; }; //i 的值为 3，j 的值在执行时再捕获
i = 22;
j = 44;
cout << f() << endl;    //输出 47 = 3+44，此时捕获 j 的值

```

上面 function 是定义在<functional>中的一个类模板，每个 L 式都是它的一个实例。

类的函数成员的函数体中可出现 L 式，但应捕获 this 指针，以访问数据成员或其它成员函数。(类在第 9 章开始介绍)

C14 还支持所谓的初始捕获或通用捕获，允许对捕获成员用任意表达式初始化。例如：

```

auto lambda = [value{ 4 }]{ return value; };    //value 并非外边定义的变量
cout << lambda() << endl;    //输出 4

```

2、**(参数表)**，与函数的形参表相同，可接受输入的参数。参数类型按需设置，单个参数或两个参数，也可能无参。在排序 sort 函数中的第 3 个实参可提供一个 L 式，要描述相邻两

个元素的排序规则，就需要两个形参，并返回一个逻辑值，称为二元谓词。在 `for_each` 函数中，单个形参表示遍历到的一个元素，可选择以值传递或引用传递。如果不改变元素，应选择以值传递。称为一元函数，不返回任何值。

C11 要求必须说明参数的具体类型。C14 放宽了这一要求，允许参数类型使用 `auto`，让编译器自动推导具体类型，即所谓的泛型 `generic L` 式或 `L` 函数。

如果参数表为空，可省略()<sub>0</sub>。

3、**{函数体}**，与普通函数体一样，包含一条或多条语句。如果 `L` 式做谓词(Predicate)就应返回逻辑值。可利用返回值自动推导，因此大多都可省略函数体的返回类型的描述。函数体中可访问以下变量：

- [捕获]而来的变量；
- (参数表)中的形参；
- 本地说明的变量；
- 具有静态作用域的任何变量(如全局变量)
- 类的数据成员 (`L` 式出现在类的成员函数中，而且要先捕获[this])

上面介绍的 3 个部分是最关键部分，未说明返回类型和异常说明这两个子句。

### 8.8.2 简单用法

`L` 式就像一个普通表达式，只要每个形参都绑定实参，就能当场执行。

一个 `L` 式的函数体后添加(实参表)，就立即计算该 `L` 式的值：

```
cout<< [](int x, int y) { return x + y; }(5, 4) << endl;    //(5,4)就是实参，对应(x,y)
```

即便有捕获也能执行，例如：

```
int m = 0, n = 0;
```

```
[&, n] (int a) mutable { m = ++n + a; }(4);
```

上面 `L` 式并不返回任何值，但改变了 `m` 的值为 `5=1+4`。函数体中改变了 `n` 的值，由于 `n` 是以值捕获的，而且函数体中改变了 `n`，因此需要 `mutable` 修饰，否则编译出错。内部改变 `n` 并不改变外边的变量 `n`。

`L` 式一般作为匿名函数，但也可对其命名，然后再用名称(实参)来调用。例如：

```
int i = 3, j = 5;
```

```
function<int(void)> f = [i, &j] { return i + j; };
```

`function` 模板实参应该是该 `L` 式的调用基调 `call signature`：`int(void)`表示无参并返回 `int`。

如果嫌麻烦可用 `auto` 代替：

```
auto f = [i, &j] { return i + j; };
```

```
cout << f() << endl;
```

但 `auto` 推导的类型不能实现递归的 `L` 式。要定义一个递归的 `L` 式就需用 `function` 类型命名：

下面用 `L` 式实现 `n` 的阶乘的递归计算：

```
function<int(int)> f1;
```

```
f1 = [&f1](int n) {return n == 1 ? 1 : n * f1(n - 1); };
```

```
cout << f1(5) << endl;
```

注意，需要引用捕获 L 式的名称[&f1]，使函数体可调用该名称。

### 8.8.3 嵌套 L 式与高阶函数

一个 L 式的形参或返回值可以是另一个 L 式。例如：

```
[] (int x) { return [] (int y) { return y * 2; } (x)+3; } (5)
```

上面 L 式是一个嵌套 L 式，下划线标出内层嵌套的一个 L 式。

外层 L 式返回表达式中包含另一个 L 式，而且尾部带有实参 x，来自外层的形参 x。

外层 L 式的实参 x=5，然后 x 赋给内层变量 y，内层返回 5\*2，外层再返回 5\*2+3。

这种嵌套 L 式称为高阶函数 high order function。高阶函数就是采用另一个 L 式作为其形参，或其返回表达式中包含另一个 L 式。

上面 L 式是无命名的方式，如果嵌套多次都无命名，导致该 L 式难以阅读难以理解。因此应采用适当命名来简化高阶函数的描述。再看一个例子：

```
auto add = [] (auto x) { return [=] (auto y) { return x + y; }; }; //A
```

```
auto higherorder = [] (auto& f, auto z) { return f(z) * 2; }; //B
```

```
auto answer = higherorder(add(7), 8); //C
```

A 行中下划线标出嵌套的 L 式，以值捕获 x 并带有形参 y，

B 行的第 1 个形参 f 是一个函数引用，实际上就是准备绑定一个嵌套的 L 式。函数体中执行 f 并用第 2 个形参绑定 f 的实参。

C 行将 higherorder 的第 1 个形参 f 绑定到 add(7)，x=7，第 2 个实参 8 被绑定到 z，z 再绑定到 A 行嵌套 L 式的 y=8，最后 f 返回 7+8，外层返回(7+8)\*2。

A 行说明的 add 可独立执行，先确定外层实参 x 再确定内层实参 y，例如：

```
cout<<add(3)(4); //x = 3,y = 4, 输出 7
```

B 行与 C 行可合并为 1 行：

```
auto answer = [] (auto& f, auto z) { return f(z) * 2; } (add(7), 8);
```

L 式也称为闭包，意思是 L 式无论如何嵌套，最终得到的还是一个 L 式。

从编程风格来讲，L 式提供一种函数式编程风格，将一个 L 式所表示的匿名函数作为调用另一个函数的实参，或者将函数当做数据来看待。

从功能上看，L 式所能实现的，用命名函数都能实现。区别只是 L 式编程是“现场解决”，而不是先转出到当前类或函数的外部，定义一个命名函数，然后再转回现场，再调用该函数来解决问题。

L 式最实用之处还在于调用 STL 算法。

### 8.8.4 调用 STL 算法

第 6 章数组中介绍了调用 STL 算法来处理一维数组，如随机生成、排序、二分查找、遍历等。STL 容器与算法的详细介绍在第 13 章。我们可将一维数组作为容器，来调用 STL 算法，以简化编程，提高编程质量。其中 L 式的作用非常重要。

L 式作为 STL 函数实参区别于一般实参。一般实参在调用函数之前先计算实参表达式的

值，然后再传给函数形参，执行函数体。L 式作为实参，只是将匿名函数传给被调用函数，被调用函数再调用执行匿名函数。这种机制被称为“回调 call back”。

对于每个 STL 算法都有自己的文档说明，调用时应根据形参要求来编写 L 式。

表 8-4 STL 算法常用 Lambda 式的要求

形参名称	所在 STL 函数	L 式要求
Generator	generate	零元无参，返回所需类型数据
Function	for_each	一元函数，返回 void，参数 x 表示遍历到一个元素
Predicate	find_if	一元谓词，返回逻辑值，参数 x 表示遍历到一个元素
Predicate	sort	二元谓词，返回逻辑值，参数 x, y 表示两个相邻元素

下面例子对一个 float 数组元素先按升序排序，以验证 L 式如何实现比较函数，灵活定制各种排序条件。

```
#include <iostream>
#include <algorithm>
using namespace std;
int main() {
    float a[] = { 4.4f, -2.2f, -5.5f, 3.3f, -1.1f };
    sort(a, a + sizeof(a) / sizeof(float));    //A 默认升序
    for (auto y : a)
        cout << y << " ";
    cout << endl;
    system("pause");
    return 0;
}
```

A 行调用了<algorithm>中的 sort 函数，没有确定比较规则，缺省为升序排序。

如果要改变为降序排序，只需向 sort 添加第 3 个实参，一个 L 式，描述比较规则：

```
sort(a, a + sizeof(a) / sizeof(float),
    [](float a, float b) {return (b < a);});
```

该 L 式有 2 个形参，称为二元谓词。形参 a 与 b 表示前后相邻两个元素。函数体应返回一个逻辑值，表示相邻元素应满足的条件。例如：

- 如果要升序，应返回 a<b 作为满足条件；
- 如果要降序，应返回 b<a；
- 如果要按绝对值升序排序，应返回 abs(a) < abs(b)。

改变排序规则只需改变 L 式中的函数体即可实现。

STL 算法也能处理指针的数组。前面例 8-18 对一组学生进行各种排序，现在可用 STL 算法来简化。不再需要自行编写排序算法，也能简化所有下标操作。下面是简化后的程序。

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

```
#include<algorithm>
using namespace std;
struct Student{
    char num[12], name[20];
    char sex;
    float mathscore;
};
void printTitle(){
    cout<<setw(6)<<"学号"<<setw(10)<<"姓名";
    cout<<setw(5)<<"性别"<<setw(6)<<"成绩"<<endl;
}
void printASTud(const Student *s){
    cout<<setw(6)<<s->num<<setw(10)<<s->name;
    cout<<setw(5)<<s->sex<<setw(6)<<s->mathscore<<endl;
}
void printStuds( Student *s[], int n){           //打印一组学生
    printTitle();
    for_each(s, s + n, [](auto stu) { printASTud(stu); });    //A
}
void printChoice(){
    cout<<"选择 0, 退出\n";
    cout<<"选择 1, 按学号升序排序\n";
    cout<<"选择 2, 按姓名升序排序\n";
    cout<<"选择 3, 按性别排序\n";
    cout<<"选择 4, 按数学成绩降序排序\n";
}
int main(void){
    Student st[] = {"002", "Wangping", 'f', 84}, {"001", "Zhaomin", 'm', 64},
        {"004", "Wanghong", 'f', 54}, {"003", "Lilei", 'm', 92},
        {"006", "Liumin", 'm', 75}, {"005", "Meilin", 'm', 74},
        {"008", "Yetong", 'f', 89}, {"007", "Maomao", 'm', 78},
        {"010", "Zhangjie", 'm', 66}, {"009", "Wangmei", 'f', 39}};
    const int num = sizeof(st) / sizeof(Student);
    Student * t = st, *stp[num];
    generate(stp, stp + num, [&t] {return t++; });    //B
    printStuds(stp, num);           //先打印排序前的名单
    while(1){
        printChoice();
        int choice = 0;
```

```

        cin>>choice;
        if (choice == 0)
            break;
        if (choice >=1 && choice <= 4){
            switch (choice){
                case 1: sort(stp, stp + num, [](auto s1, auto s2)
                    {return strcmp(s1->num, s2->num) < 0; }); break;
                case 2: sort(stp, stp + num, [](auto s1, auto s2)
                    {return strcmp(s1->name, s2->name) < 0; }); break;
                case 3: sort(stp, stp + num, [](auto s1, auto s2)
                    {return s1->sex - s2->sex < 0; }); break;
                case 4: sort(stp, stp + num, [](auto s1, auto s2)
                    {return s1->mathscore - s2->mathscore > 0; }); break;
            }
            printStuds(stp, num);
        }
    }
    system("pause");
    return 0;
}

```

A 行调用 `for_each` 来遍历每个元素输出。B 行调用 `generate` 生成指针数组的各个元素。C 行开始调用 4 次 `sort` 来执行不同的排序。

## 8.9 单向链表及其应用\*

链表(linked list)是一种动态可伸缩的数据结构，其应用非常广泛。作为一个综合实例，本节介绍单向链表及其应用。

### 8.9.1 链表概述

C++语言中可以用数组来处理一组类型相同的数据，但不允许动态改变数组的大小，即在使用一个数组之前必须确定其大小。在实际应用中，经常在使用数组之前无法确定数组的大小，只能将数组定义得足够大，这样数组中有些空间不能有效利用，造成内存空间浪费。

链表是一种常用的数据组织形式，它采用动态分配内存来实现动态伸缩。需要添加新元素时就用 `new` 分配内存空间，保存其指针。不需要某个元素时就用 `delete` 回收，不会浪费内存空间。

一个链表就像一个一维数组，管理一组有序的数据。在一个链表中，逻辑上相邻的两个元素  $a_i$  与  $a_{i+1}$  在内存中并不一定相邻。为了表示  $a_i$  与  $a_{i+1}$  之间的前后关系，对数据元素  $a_i$  来说，除了要存储本身信息之外，还要存储一个指向下一个元素  $a_{i+1}$  的一个指针。习惯上将链

表中的数据元素称为结点(Node)。结点结构如下图所示。

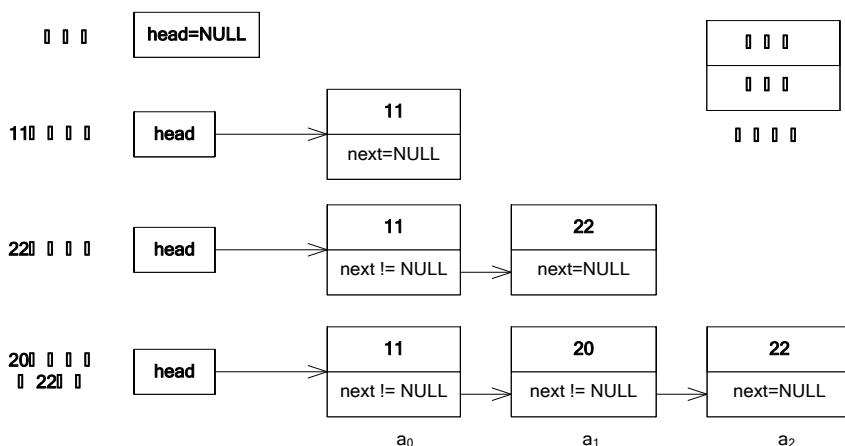


图 8-11 结点和链表的结构

每个结点包含一个数据域和一个指针域。数据域存放数据元素本身的一组信息；指针域存放下一个结点的地址。一个链表是由  $n(n \geq 0)$  个结点组成的。当  $n$  为 0 时表示空链表。图中最后给出由 3 个结点组成的一个链表。

图中的 `head` 称为头指针，它指向链表的第 0 个结点  $a_0$ ， $a_0$  的 `next` 指针指向第 1 个结点  $a_1$ ，...，直到最后一个结点，将  $a_2$  的 `next` 指针置为空，表示后面再没有结点。这是一个单向链表结构。也就是说，对链表的访问必须从头指针 `head` 开始，然后按照结点的先后顺序依次访问后面的各结点。

一个链表就好像排队购买火车票的一个队列。头指针指向队列中的头一个结点，就是头一个购票人。依次排列，后来者应作为尾结点，可不断延长。这个队列没有长度限制，只受最大可用内存限制。头结点购买完后应离开队列，此时下一个结点就作为头结点。一般情况下，“插队”是不允许的，但在链表中却很容易将一个新结点插入到某个结点之前。同样，任何一个结点(购票人)都可以中途离开队列，使后面的结点前移。当尾结点离开时，会使其前一个结点作为尾结点。当头结点就是尾结点时，链表中就只有一个结点了。此时如果尾结点离开，就成了空链表了。

注意有一点，链表不同于购票队列。链表中每个结点都只需知道它后面的一个结点，就是“向后看”。这种链表被称为“单向链表”。而购票队列中，每个排队购票人都是“往前看”，只需知道其前面一个购票人，而不关心谁在他后面。

链表具有从前向后按顺序存取的特点。头指针 `head` 至关重要，如果在程序中弄错了头指针，链表将无法访问或者出错。

由于结点结构中包含两个域，可以采用结构体类型实现，其数据类型为：

```
struct Node{
    <ElemType> data;
    Node *next;
```

```
};
```

其中, `<ElemType>` 为结点数据 `data` 的类型, 它是一个抽象的概念, 在实际应用中可将其确定为一个具体的数据类型, 如 `int` 型、字符串或结构类型。简单起见, 现假设每个结点上只包含一个 `int` 值。`next` 为指向下一个结点的指针。`Node` 结构如下:

```
struct Node{
    int data;
    Node *next;
};
```

为了将一个链表作为一个变量来处理, 设计一个结构来表示链表:

```
struct LinkedList{
    Node * head;
    int size;
};
```

一个链表就是 `LinkedList` 结构的一个变量。其中 `head` 指针指向头结点, `size` 表示当前链表中结点数。显然一个空表的 `head` 值应为 `NULL`, 而且 `size` 值应为 0。设计这个结构的好处是方便管理链表变量, 避免直接传递操作 `head` 指针, 也能直接得到链表中元素的动态数量, 以使用下标(范围为 0 到 `size-1`)来访问各元素。

对一个链表的基本操作一般包括下面功能:

- 建立一个链表, 并可以用一个数组进行初始化。
- 对一个链表, 按前后次序输出所有结点的数据, 称为一次遍历。
- 对一个链表, 撤销其中全部元素, 成为空表。
- 对一个链表, 把一个新元素插入到指定下标位置。
- 对一个链表, 根据一个给定值, 按次序查找是否有某结点持有这个值。
- 对一个链表, 删除指定下标位置上的一个结点。
- 对一个链表, 撤销全部结点。这应该是程序最后一步, 回收全部动态分配的内存。

还有其它操作。每个操作都实现为一个函数。

单向链表编程是一个典型的综合性实例, 涉及到前面介绍的多个概念:

- 结构的指针。
- 指针作为结构成员。
- 一维数组元素下标的用法。
- 动态使用内存。
- 指针的引用作为函数形参、函数返回引用。

设计要点:

- 用 `LinkedList` 结构来表示链表, 其中封装了头指针 `head` 和 `size` 动态大小。
- 可以创建一个空表, 也可以用一个 `int` 数组来初始化各元素。
- 处理函数都以 `LinkedList` 引用作为第一个形参, 方便调用。
- 用下标 `[0, size-1]` 来控制元素的插入、定位、查找、删除。如果下标越界, 引发异常来通知调用方。

设计要求:



- 每个函数仅实现单一功能，例如查找和删除作为两个独立函数，以简化设计。
- 函数设计尽可能不用指针做形参，也不返回指针，以方便使用。
- 使用方无需定义和操作 **Node** 变量，以简化使用。

单向链表编程比较复杂，我们逐步建立整个程序。分为以下三个步骤：

第1步：建立链表并初始化、遍历输出、撤销链表，将介绍以下4个函数：

```
LinkedList create();
```

创建一个空链表。

```
LinkedList create(const int a[], int n);
```

创建一个链表，并用指定的 `int` 数组来初始化。

```
void print(const LinkedList &list);
```

打印链表中各元素。

```
void removeAll(LinkedList &list);
```

撤销链表中的所有元素。

第2步：增加函数：按下标访问元素、添加新元素、删除元素、查找元素，将介绍以下4个函数：

```
int &elemAt(const LinkedList &list, const int index)
```

返回指定下标的结点的数据引用，按下标就能访问元素。此函数调用可作为左值。

```
void insertAt(LinkedList &list, const int a, const int index)
```

把指定数据 `a` 插入到指定下标 `index` 位置。若 `index=0`，则插入做头结点。若 `index=size`，则插入做尾结点。

```
int remove(LinkedList &list, const int index)
```

删除指定下标 `index` 结点，并返回被删结点的数据。

```
int search(const LinkedList &list, const int data)
```

在链表中查找指定的 `data`，返回持有该值的第一个结点的下标位置。

第3步：增加函数，实现直接插入排序，介绍下面1个函数：

```
void insertAsc(LinkedList &list, const int data)
```

假设链表 `list` 中已有元素按升序排列，插入指定数据 `data`，仍保持升序。

说明：

- 在每一步，新加函数可能会调用前面已定义的函数，但不会改变已定义的函数。
- 第一步将给出完整的可执行程序，下面两步仅介绍添加函数和相应的 `main` 函数。

### 8.9.2 建立无序链表、遍历和撤销

例8-27 建立一条无序链表，再输出该链上各结点的数据，然后撤销链表。编程如下：

```
#include <iostream>
using namespace std;
struct Node{
    int data;
    Node *next;
};
struct LinkedList{
    Node * head;
    int size;
```

```
};  
//创建一个空链表  
LinkedList create(){  
    LinkedList list;  
    list.head = 0;  
    list.size = 0;  
    return list;  
}  
//创建一个链表,并用指定的 int 数组来初始化  
LinkedList create(const int a[], int n){  
    LinkedList list = create();  
    Node *p1, *p2;  
    for (int i = 0; i < n ; i++){  
        p1 = new Node;  
        p1->data = a[i];  
        p1->next = NULL;  
        if (list.head == 0)  
            list.head = p1;  
        else  
            p2->next = p1;  
        p2 = p1;  
    }  
    list.size = n;  
    return list;  
}  
//打印链表中各元素  
void print(const LinkedList &list){  
    cout<<"size="<<list.size<<":";  
    Node *p = list.head;  
    while (p != NULL){  
        cout <<p -> data << " ";  
        p = p -> next;  
    }  
    cout<<"\n";  
}  
//撤销链表中的所有元素,结果为空表  
void removeAll(LinkedList &list){  
    Node *p1;  
    while(list.head != NULL){  
        p1 = list.head;  
        list.head = list.head -> next;  
        delete p1;  
    }  
    list.size = 0;  
}  
int main(){  
    int a[] = {200,400,300,500};  
    LinkedList list1 = create(a, sizeof(a)/4);  
    print(list1);  
    removeAll(list1);  
    system("pause");  
    return 0;  
}
```

执行程序,输出如下:

```
size=4:200 400 300 500
```

第1个 create 函数用来创建一个空链表,第2个 create 函数用来创建一个链表,并用一个 int 数组元素来初始化,就是将数组中的各元素加入到新建链表中。对于每个元素的处理过程有两步:第1步取出一个整数元素,建立一个新结点;第2步将这个结点插入到链尾。循环这两步,直到所有元素都加入。在每一次循环中, p1 指向新建立结点, p2 指向尾结点。当要加入第一个结点时,为空链,要设置 head 的值。再加入结点时,就要把新结点作为链尾。下图表示了3个元素连续加入的过程。读者可尝试用键盘输入一组数据来建立链表并初始化。

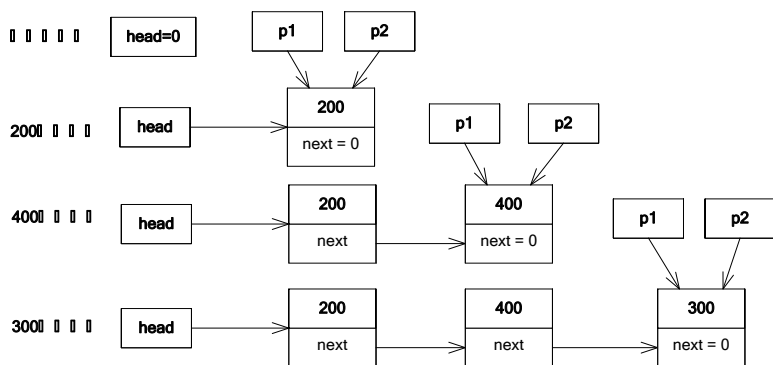


图 8-12 连续加入元素的链表状态

函数 print 从头到尾遍历各结点并打印数据。函数 removeAll 撤销链表中所有结点,也同样需要一次遍历。

### 8.9.3 添加、查找与删除元素

下面添加4个处理函数和相应的 main 测试函数。

```

/*
返回指定下标的结点的数据引用,目的是按下标访问结点数据
如果下标 index 不在[0, size-1]范围,引发 index 异常
此函数调用可作为左值
*/
int &elemAt(const LinkedList &list, const int index){
    if (index < 0 || index >= list.size){
        throw index;
    }
    Node * p = list.head;
    for(int i = 0; i < index; i++){
        p = p -> next;
    }
    return p -> data;
}
/*
把指定数据 a 插入到指定下标 index 位置
如果 index 不在[0, size]范围,引发 index 异常
如果 index==0,插入为头结点
如果 index==size,插入为尾结点
*/
void insertAt(LinkedList &list, const int a, const int index){
    if (index < 0 || index > list.size)

```

```

        throw index;
    Node * newNode = new Node;
    newNode->data = a;
    newNode->next = NULL;
    list.size++;
    if (index == 0){ //插入到头结点
        newNode->next = list.head;
        list.head = newNode;
        return;
    }
    Node *p1 = list.head, *p2 = p1;
    for(int i = 0; i < index; i++){ //使插入位置位于 p1 和 p2 之间, p1 是 p2 前结点
        p1 = p2;
        p2 = p2->next;
    }
    newNode->next = p2;
    p1->next = newNode;
}
/*
删除指定下标 index 结点, 并返回被删结点的数据
如果是空表, 即 size = 0, 引发 index 异常。
如果 index 不在[0, size-1]范围, 引发 index 异常
*/
int remove(LinkedList &list, const int index){
    if (list.size == 0 || index < 0 || index >= list.size)
        throw index;
    if (index == 0){ //删除头结点
        list.size--;
        Node *p = list.head;
        list.head = list.head->next;
        int a = p->data;
        delete p;
        return a;
    }
    Node *p1 = list.head, *p2 = p1;
    for(int i = 0; i < index; i++){
        p1 = p2;
        p2 = p2->next;
    }
    list.size--;
    p1->next = p2->next;
    int a = p2->data;
    delete p2;
    return a;
}
/*
在链表中查找指定的 data, 返回持有该值的第一个结点的下标位置
如果没有找到, 返回-1
*/
int search(const LinkedList &list, const int data){
    Node *p = list.head;
    for(int i = 0; p != NULL; i++){
        if (p->data == data)
            return i;
        p = p->next;
    }
    return -1;
}
int main(){

```

```

LinkedList list1 = create();
cout<<"add 11 at index 0"<<endl;
insertAt(list1, 11, 0);
print(list1);
cout<<"add 22 at index 1"<<endl;
insertAt(list1, 22, 1);
print(list1);
cout<<"add 33 at index 1"<<endl;
insertAt(list1, 33, 1);
print(list1);
cout<<"add 44 at the last"<<endl;
insertAt(list1, 44, list1.size);
cout<<"Now there are "<<list1.size<<" elements:"<<endl;
for(int i = 0; i < list1.size; i++){
    cout<<"["<<i<<"]="<<elemAt(list1, i)<<endl;
}
if (search(list1, 55) == -1)
    cout<<"search 55, not found"<<endl;
if (search(list1, 44) == 3)
    cout<<"search 44, found at index 3"<<endl;
remove(list1, 3);
cout<<"remove element at index 3"<<endl;
print(list1);
remove(list1, 1);
cout<<"remove element at index 1"<<endl;
print(list1);
remove(list1, 0);
cout<<"remove element at index 0"<<endl;
print(list1);
remove(list1, 0);
cout<<"remove element at index 0, now the list is empty"<<endl;
print(list1);
system("pause");
return 0;
}

```

执行程序，输出如下：

```

add 11 at index 0
size=1:11
add 22 at index 1
size=2:11 22
add 33 at index 1
size=3:11 33 22
add 44 at the last
Now there are 4 elements:
[0]=11
[1]=33
[2]=22
[3]=44
search 55, not found
search 44, found at index 3
remove element at index 3
size=3:11 33 22
remove element at index 1
size=2:11 22
remove element at index 0
size=1:22
remove element at index 0, now the list is empty
size=0:

```

上面 main 函数对前面 4 个处理函数进行测试。先创建一个空表，再调用 insertAt 函数将 4 个元素插入链表。对于这个链表，调用 elemAt 函数按下标来读取各结点的数据。然后调用 search 函数查找两个数据。最后调用 remove 函数 4 次将链表中的结点删除，成为空表。

上面 4 个处理函数都与下标有关。采用下标来访问链表和数组是通行做法。如果有下标作为函数形参，那么函数中第一条语句往往就是检查下标是否越界。如果下标越界就用 throw 语句来引发一个 int 型异常来通知调用方。这也是一种通行做法。调用方可以捕获这种异常，也可以置之不理，就像上面 main 函数，如果真的引发异常，就会终止程序。在第 15 章将介绍如何捕获并处理异常。

上面函数中比较复杂的是插入 insertAt 和删除 remove 函数。插入元素作为头结点，或者删除头结点的处理都比较简单，而比较复杂的是处理非头结点的情形。在两个函数中，有下面一段代码是共同的：

```
Node *p1 = list.head, *p2 = p1;
for(int i = 0; i < index; i++){
    p1 = p2;
    p2 = p2->next;
}
```

定义了两个指针，从前往后推移到 index 指定的位置。当循环结束时，p2 指向 index 位置结点，同时 p1 指向其前一个结点。

对于 insertAt 函数，就是要把新结点 newNode 插入到 p2 和 p1 之间。此时需要下面两条语句将新结点插入：

```
newNode->next = p2;
p1->next = newNode;
```

下图表示了插入结点的过程。

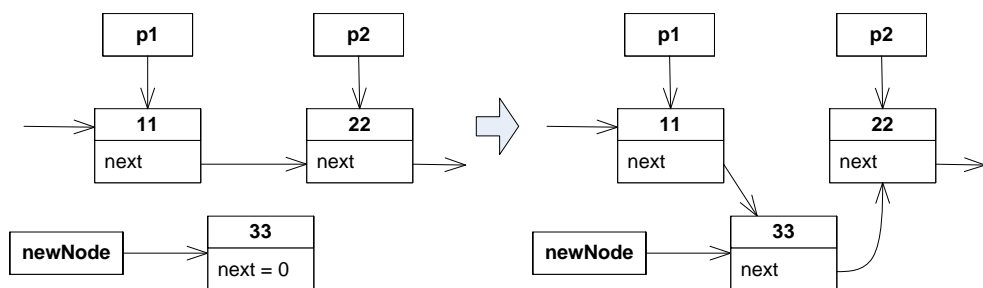


图 8-13 插入结点的过程

在 insertAt 函数中，我们约定如果 index==size，就表示要将新结点作为链表的尾结点。此时 p2 为空 NULL，p1 指向原先的尾结点。同样使用这两条语句把新结点作为尾结点。

对于 remove 函数，p2 指向要删除的结点，p1 指向其前一个结点。此时需要下面语句将 p2 结点从链表中删除：

```
list.size--;
p1->next = p2->next;
```

```
int a = p2->data;
delete p2;
return a;
```

下图表示了删除结点的过程。

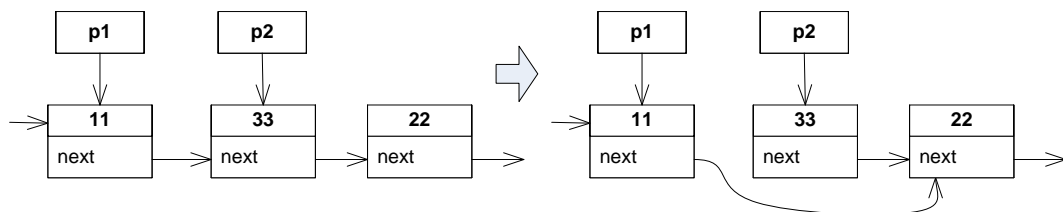


图 8-14 删除结点的过程

在 `remove` 函数中, `p2` 可能指向链表的尾结点, 此时 `p2->next` 为空 `NULL`。对于这种情形, 同样可用上面语句来删除, 使 `p1` 指向的结点称为新的尾结点。

#### 8.9.4 建立有序链表

有时我们需要按各结点中的数据从小到大排列。有一种排序算法称为直接插入排序。先假定一个链表中已有的结点数据都按从小到大次序排列, 然后添加一个新结点到链表中合适的位置, 使各结点还能保持升序排列。这个添加过程反复多次就能建立一个有序链表。

下面添加一个函数来实现这种直接插入排序, 再用一个 `main` 函数进行测试。

```
/*
假设链表中已有元素按升序排列, 插入指定数据 data, 仍保持升序。
如果只调用此函数来加入数据元素, 该链表是升序排列
*/
void insertAsc(LinkedList &list, const int data){
    Node *p = new Node;
    p->data = data;
    list.size++;
    if (list.head == NULL || p->data <= list.head->data){ //作为头结点
        p->next = list.head;
        list.head = p;
        return;
    }
    Node *p1 = list.head, *p2 = p1; //作为中间结点或尾结点
    while (p2 != NULL && p2->data < p->data){
        p1 = p2;
        p2 = p2->next;
    }
    p->next = p2;
    p1->next = p;
}

void main(){
    int a[] = {22,11,44,33};
    LinkedList list1 = create();
    for(int i = 0 ; i < 4; i++){
        insertAsc(list1, a[i]);
        cout<<"insert "<<a[i]<<endl;
```

```
        print(list1);  
    }  
    removeAll(list1);  
}
```

执行程序，输出如下：

```
insert 22  
size=1:22  
insert 11  
size=2:11 22  
insert 44  
size=3:11 22 44  
insert 33  
size=4:11 22 33 44
```

函数 `insertAsc` 要处理 2 种情形：新结点作为头结点、作为中间结点和尾结点。显然处理第 2 种情形比较麻烦。先定义两个指针 `p1` 和 `p2`，从前往后推移到合适的位置。当循环结束时两种情形：`p2` 为空 `NULL`，此时新结点应作为尾结点。`p2` 不为空，且 `p2` 结点数据不小于新结点数据。此时新结点应插入到 `p2` 之前，此时 `p1` 指向其前一个结点。无论那一种情形，函数中最后两条语句都能插入新结点，作为尾结点或者插入两者之间，就像前面 `insert` 函数一样。

使用 `insertAsc` 函数建立的有序链表就不能再调用 `insert` 函数来处理，因为这样就可能破坏有序性。同样，如果一个链表先前被 `insert` 函数处理过，也不能再调用 `insertAsc` 函数，因为假设条件可能不成立，已有结点可能是无序的。

要把一个无序链表转变为一个有序链表，方法有多种。用 `elemAt` 函数能读写各下标元素的值，因此前面介绍的选择排序和冒泡排序算法都可以使用。也可循环调用 `elemAt` 函数和 `insertAsc` 函数来建立一个新的有序表，再用 `removeAll` 函数来删除原先无序表。

上面程序能实现单向链表的工作原理，但也存在如下问题：

- 1、任何函数都能对 `LinkedList` 变量的两个成员 `size` 和 `head` 进行任意改变，而这容易造成严重错误。这是由于结构中的成员都是公开的，没有任何封装和保护。后面我们将引入面向对象编程的封装性来解决此问题。

- 2、当不再使用一个链表时，往往会忘记调用 `removeAll` 函数来回收该链表所占内存，例如上面 `main` 函数中的最后一条语句。如果在循环程序中多次创建链表而忘记回收，就可能导致严重错误。后面我们将引入面向对象编程的析构函数，使回收能自动进行。

- 3、例子中的结点数据是一个简单的 `int` 型，如果需要其它类型作为结点数据的话，例如一个结构类型作为结点数据，那么前面多数处理函数都需要改变。

## 8.10 小结

指针的本质就是内存地址。指针变量就是命名的地址。指针的用途主要有三：

- 关联数据：通过指针指向被关联的对象，可以表示更复杂的数据结构。例如链表。
- 提高效率：指针作为函数形参能实现地址传递，方便函数处理大内存的数据。
- 动态编程：动态申请使用内存、动态回收，提高内存使用效率。

“\*” 有三种含义：



- 乘法运算符, 如  $i = j * k$ ; 作为双目运算符
- 在说明语句或函数形参中, 说明一个指针变量, 如  $\text{int } *p = \&j$ ;
- 间接引用运算符, 作用于指针变量之前, 访问指针所指向的数据, 作为单目运算符。

“&”有三种含义:

- 按位与运算符, 双目运算符, 如  $i = j \& k$ ; 作用于整数或字符
- 求址运算符, 在已说明的变量之前, 得到该变量的地址, 即指针, 如  $p = \&i$ ; 这是单目运算符。
- 在说明语句或函数形参中, 说明引用变量。
- 对于指针只能进行赋值运算、间接引用运算、算术运算、两个指针间的减运算和关系运算。其中算术运算和两个指针之间的减运算只能在访问数组时才有用。其中算术运算只包括与整数的加减运算和自增自减运算。

“&”所表示的求址运算与“\*”所表示的间接引用运算是一对互逆的运算:

- 对于任一个变量  $v$ ,  $*\&v == v$ 。
- 对于任一个指针  $p$ ,  $\&*p == p$ 。
- 一个数组的名字就是头一个元素的地址, 但这个指针的值是一个常量, 不能改变它的值。
- 通过一个结构指针变量, 能访问该结构变量内的各个成员。格式为:  
    <结构指针变量> -> 成员名
- 指针也可作为结构的成员。
- 一个数组名本质上就是一个指针。例如  $\text{int } a[5]$ ,  $a$  是常量指针, 指向首元素, 那么  $a+i$  就指向第  $i$  个元素, 有下面恒等式:  
     $a[i] == *(a+i)$  或者  $\&a[i] == (a+i)$
- 二维数组是一维数组的数组, 因此二维数组名也是指针, 有下面恒等式:  
     $b[i][j] == *(b[i] + j) == *((b + i) + j) == (*(b + i))[j]$
- 数组的指针, 单个指针的一种类型, 用来指向一维数组。在定义一维数组的一个指针变量时, 必须确定一维数组的类型和大小。例如  $\text{int } b[3][4]; \text{int } (*bp)[4] = b$ 。二维数组名的类型就是一维数组的指针。数组的指针可访问二维数组元素, 也能作为函数的形参。
- 字符的指针, 可用来定义字符串常量, 也常作为函数的形参表示字符串形参。
- 指针的数组, 其中每个元素是某种类型的一个指针。例如  $\text{int } b[3][4]; \text{int } *p[] = \{b[0], b[1], b[2]\}$ 。可访问二维数组元素, 也能作为函数的形参。可用来处理字符串的数组和结构的数组。
- 二级指针就是指针的指针, 可以指向二维数组, 用来访问二维数组元素。例如:  
     $\text{int } b[3][4];$   
     $\text{int } *p[] = \{b[0], b[1], b[2]\};$   
     $\text{int } **pp = p;$
- 二级指针可作为函数形参来传递二维数组。主调方在调用前要先定义一个指针的数组  $p$ , 并使各元素指向二维数组的各行, 然后才能作为调用的实参。

- 指针不仅可作为函数的形参，也能作为函数的返回值。
- 数组和指针交替作为形参和实参，有以下 3 个等价关系：
  - 一级指针(如 `int *`)等价于一维数组(如 `int[n]`)。
  - 数组的指针(如 `int (*)[n]`)等价于二维数组(如 `int a[][n]`)。
  - 二级指针(如 `int **`)等价于指针的数组(`int *[n]`)。
- 函数可以返回指针，但不能指向函数体内定义的非静态局部变量。
- 函数的指针。函数也有指针，函数名就是其地址。说明函数的指针变量的语法格式为：

<类型> (\*<变量名>) (<形参表>) ; //两对圆括号有鲜明特点

只能将具有相同的函数形参表和相同返回类型的函数名赋给函数指针变量。

通过函数指针调用它所指向的函数。

- `void` 型指针是不确定类型的，它可以指向任何类型的变量或函数。任何类型指针都可以赋给 `void` 指针。`void` 指针常作为通用性函数的形参。对于一个 `void` 指针，在对其进行间接引用或算术运算之前，必须将其强制转换为某一种具体类型的指针。
- `const` 指针。`const` 在指针变量的类型之前，表示不能改变指针所指向的数据值。`const` 放在\*之后、变量名之前，表示不能改变指针变量的值。
- 动态使用内存。用 `new` 运算符来申请内存，使用完之后再使用 `delete` 运算符回收内存。必须用指针才能动态使用内存。注意，对于已回收的内存不能再访问。一块内存空间只能回收一次。
- 引用是对已有变量的一种别名机制，主要用于说明函数形参。引用传递是 C++ 提供的一种高级传递方式。引用类型的变量是其它变量的别名，对引用变量的操作实际上就是对被引用变量的操作。
- 对一个数组可说明其引用变量。但引用变量只能是单个值，不能说明引用的数组。
- 对于一个引用变量，可以说明其指针或地址，但本质上还是被引用的变量的指针，这是因为引用没有自己的存储空间。
- 对于一个指针，可以说明其引用。指针的引用常作为函数形参，函数中可改变指针或指针所指内容。
- 引用作为函数形参比较简单，容易使用。
- 引用作为函数的返回值，可对其返回值直接进行访问，即可作为左值。但函数内不能返回非静态局部变量。
- 链表可以看做一种可伸缩的一维数组，作为一个典型的综合实例，包括了数组下标、结构、指针、引用、动态使用内存等等内容。本章介绍了单向整型链表的工作原理，也介绍了一个应用实例。

## 8.11 练 习 题

1. 设有语句 `int k=8, *p=&k; *p` 的值是\_\_\_\_\_。

- A 指针变量 `p` 的地址值    B 变量 `k` 的地址值    C 变量 `k` 的值 8    D 无意义

2. 对一个指针变量 `int *p` 赋值, 下面哪一个赋值是正确的?

A `float f; p = &f;`  
B `p = 0x3000`  
C `int k; *p = &k;`  
D `int k; p = &k;`

3. 下面代码段执行输出结果为\_\_\_\_\_。

```
int a = 2, *pa = &a;  
int b = 3, *pb = &b;  
*pa*=*pa**pb;  
cout<<a<<endl;
```

A 2      B 6      C 12      D 语法错

4. 对于指针运算, 下面哪一个错误的?

A 可以用一个空指针赋值给某个指针变量。  
B 两个指针可以进行加法运算。  
C 如果一个指针指向数组元素, 指针可以加上一个整数。  
D 如果一个指针指向数组元素, 该指针可以执行自增自减运算。

5. 执行下面代码段后, `b` 的值为\_\_\_\_\_。

```
int a[] = {1,2,3,4,5,6,7,8};  
int *p = a + 4;  
int b = 4+ ++*p++;
```

A 8      B 9      C 10      D 语法错

6. 下面哪一个表达式不能访问二维数组 `b` 的第 `i` 行第 `j` 列元素?

A `b[i][j]`      B `*(b[i]+j)`      C `*(*b+i)+j`      D `(*b+i)[j]`

7. 假设函数原型和变量说明如下:

```
void f3(int (*p)[4]);  
int a[4] = {1,2,3,4};  
int b[3][4] = {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
```

下面哪一个调用是非法的?

A `f3(&a);`      B `f3(b[1]);`      C `f3(&b[1]);`      D `f3(b);`

8. 设有下面语句:

```
char *s1 = "C++ Programming";  
char s2[] = "C++ Programming";
```

下面哪一种操作不会导致错误?

A `strcpy(s1, "C++");`      B `s1 = "C++";`  
C `s2 = s1;`      D `strcpy(s2, "Java Programming");`

9. 设有语句: `int *p[10];` 下面哪一个描述是正确的?

A `p` 是指向数组中第 10 个元素的指针。  
B `p` 是具有 10 个元素的指针的数组, 每个元素是一个 `int` 型指针。  
C `p` 是指向数组的指针。

- D `p[10]`表示数组的第 10 个元素的值。
10. 设有语句 `int b[3][5];` 下面哪一条语句是正确的?
- A `int (*p)[5] = b;`                      B `int *p[] = b;`  
C `int *p[5] = b;`                      D `(int *)p[5] = b;`
11. 设有语句 `int b[3][4];` 下面哪一条语句是正确的?
- A `int *p[] = {b[0], b[1], b[2]};`                      B `int *p[] = b;`  
C `int *p[2] = {b[0], b[1], b[2]};`                      D `int *p[] = (int *[])b;`
12. 假设函数原型和变量说明如下:
- ```
void f4(int **p);  
int a[4] = {1,2,3,4};  
int b[3][4] = {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};  
int *q[3] = {b[0],b[1],b[2]};
```
- 下面哪个调用是合法的?
- A `f4(a);`                      B `f4(&a);`                      C `f4(b);`                      D `f4(q);`
13. 设有语句如下:
- ```
char *c1[] = {"Red","Green","Blue"};  
char c2[][6] = {"Red","Green","Blue"};
```
- 下面哪一种说法是正确的?
- A `sizeof(c1)` 等于 `sizeof(c2)`                      B `sizeof(c1)` 小于 `sizeof(c2)`  
C `sizeof(c1)` 大于 `sizeof(c2)`                      D `sizeof(c1)` 加 6 等于 `sizeof(c2)`
14. 下面哪一个函数原型能接受处理任意行、任意列的 `int` 矩阵, 形参 `row` 表示行数, `col` 表示列数:
- A `void print(int a[row][col]);`  
B `void print(int a[][col], int row);`  
C `void print(int *a[], int row, int col);`  
D `void print(int a[][], int row, int col);`
15. 关于函数返回指针, 下面哪一种说法是错误的?
- A 可以返回函数体中定义的局部变量的地址。  
B 可以返回形参指针。  
C 可以返回动态创建的对象。  
D 可以返回静态变量或全局变量, 但很少用。
16. 设有说明语句: `int * (*fun)(int);` `fun` 表示\_\_\_\_\_。
- A 一个返回值为指针型的函数名。                      B 一个指向函数的指针变量。  
C 一个指向一维数组的指针。                      D 一个用于指向 `int` 型的指针变量。
17. 设有语句: `int k=2, *intp = &k;` 那么表达式: `(*fun)(*intp)` 表示\_\_\_\_\_。
- A 说明一个函数指针 `fun`。                      B 将 `int` 变量 `k` 转换为指针类型 `fun`。  
C 通过函数指针 `fun` 来调用函数, 实参为 `*intp`。                      D 这是一个错误表达式。
18. 对于 `void` 指针, 下面哪一种说法是错误的?
- A `void` 指针可作为函数的形参, 调用实参可以是任何类型的指针, 在语法上是合法的。  
B `void` 指针可实现通用性编程, 使一个函数可以处理多种类型的数据。

- C 对 void 指针可通过强制类型转换为指定类型的指针, 然后再通过该指针来访问数据。  
D void 指针可作为调用函数的实参, 可匹配任意类型的指针形参。
19. 设有说明语句: `const char *ps;` `ps` 表示\_\_\_\_\_。
- A 指向字符串的指针。 B 指向字符串的 `const` 型指针。  
C 指向 `const` 型字符串的指针。 D 指向 `const` 型字符串的 `const` 型指针。
20. 下面哪一个指针变量不能改变, 而指针所指的内容可以改变?
- A `const int *pa;` B `int const * pa;`  
C `int * const pa;` D `const int * const pa;`
21. 下列哪一个 `main` 函数原型是错误的?
- A `main(int argv, char *argc[])` B `main(int arc, char **arv)`  
C `main(int argc, char *argv)` D `main(int a, char *c[])`
22. 关于 `new` 运算符, 下面哪一种说法是错误的?
- A 创建单个基本类型对象时, 可以指定初值。 B 创建多个对象时, 可以指定初值。  
C 创建的对象可以用 `delete` 撤销。 D 可以动态创建单个对象或多个对象。
23. 关于 `delete` 运算符, 下面哪一种说法是错误的?
- A 它必须作用于 `new` 返回的指针。  
B 对 `new` 返回的一个指针可多次使用 `delete` 运算符。  
C 当程序执行结束时, 不会对未回收的内存自动执行 `delete`。  
D `delete []pa`, 指针名前用方括号 `[]` 表示释放多个对象。
24. 设有说明语句: `float x=6.2;` 下列哪一条能说明 `x` 的引用?
- A `float &y = &x;` B `float &y = x;`  
C `float y=&x;` D `float &y;`
25. 设有说明语句: `int m, n=2, *p=&n;` 下面哪一条语句能完成 `m=n` 赋值功能?
- A `m = *p;` B `*p = *&n;` C `m = &n;` D `m = **p;`
26. 下面程序的输出结果是\_\_\_\_\_。
- ```
#include <iostream.h>

void main(void){
    char *p="abcdefgh", *r;
    int *q;
    q=(int *)p; q++; r=(char*)q;
    cout<<r<<endl;
}
```
- A abcd B a C efgh D b
27. 下面程序的输出结果是\_\_\_\_\_。
- ```
#include <iostream.h>

void fun(int *a, int *b)
{ int *k; k=a; a=b; b=k; }

void main(void){
    int a=3, b=6, *x=&a, *y=&b;
```

```
    fun(x,y);cout<<a<<","<<b<<endl;
}
```

A 3,3                      B 3,6                      C 6,3                      D 6,6

28. 下面程序的输出结果是\_\_\_\_\_。

```
#include <iostream.h>
void amovep(int *p, int a[], int n){
    int i;
    for(i=0;i<n;i++){
        *p=a[i]; p++;
    }
}
void main(void){
    int *p,a[9]={1,2,3,4,5,6,7,8,9};
    p=new int[10]; amovep(p,a,9);
    cout<<p[2]<<","<<p[5] <<endl;
    delete []p;
}
```

A 3,3                      B 3,6                      C 6,3                      D 6,6

29. 下面程序的输出结果是\_\_\_\_\_。

```
#include <iostream.h>
void sub(int *a, int n, int k){
    if(k<=n) sub(a, n/2, 2*k);
    *a += k;
}
void main(void){
    int x=0;
    sub(&x,8,1);
    cout<<x<<endl;
}
```

A 8                      B 2                      C 6                      D 7

30. 下面程序的输出结果是\_\_\_\_\_。

```
#include <iostream.h>
const int n=5;
void main(void){
    int a[n]={3,10,5,6,12};
    int *p1=a,*p2=a+n-1;
    while(p1<p2){
        int x=*p1;*p1=*p2;*p2=x;
        p1++;p2--;
    }
}
```

```
    }  
    for(int k=0;k<n;k++)    cout<<*(a+k)<<" ";  
    cout<<endl;  
}  
A 3 6 5 10 12  
C 12 10 6 5 3  
B 3 5 6 10 12  
D 12 6 5 10 3
```

31. 根据要求, 编写完整的程序。

(1) 设计一个函数, 对一个 `int` 数组查找最大值, 要求得到最大值及其下标, 要求用两种方法实现: ① 传地址方式, ② 引用方式。

(2) 设计一个函数, 形参为任意一个字符串, 将其中的大写字母转换成小写字母。

(3) 编写一个程序, 在命令行可输入多个字符串作为参量, 将各个参量按升序排列输出。

(4) 编写一个程序, 管理一组学生的考试成绩, 要记录考号、姓名、性别、语文分数、数学分数、外语分数, 输入 5 名学生的信息, 然后分别按考号、姓名、性别、总分数排序输出。

(5) 定义一个函数指针能指向一类查找条件函数, 如 `bool searchCond1(float f)`, 如果形参 `f` 满足特定条件就返回 `true`。再设计一个函数, 对一个 `float` 数组元素按条件查找, 查找的条件由函数指针所指定的函数完成, 返回符合条件的第一个元素的地址, 或者返回空。测试验证程序, 至少编写 2 个函数, 并能通过函数指针来调用。

(6) 定义一个指向字符串的指针数组, 设计一个函数完成 `n` 个不等长字符串的输入, 用 `new` 运算符根据实际输入的字符串长度分配存储空间, 依次使指针数组中的元素指向每一个输入的字符串。再设计一个函数完成 `n` 个字符串排序 (在排序的过程中, 要求只交换指向字符串的指针值, 不交换字符串)。在主函数中完成将排序后的字符串输出。

(7) 输入一个字符串, 串内有数字和非数字字符, 如:

```
abc2345 345rrf678 jfkld945
```

将其中连续的数字作为一个整数, 依次存放到另一个整型数组 `b` 中。如对于以上的输入, 将 2345 存放到 `b[0]`, 345 放入 `b[1]`, ...。统计出字符串中的整数个数, 并输出这些整数。要求在主函数中完成输入和输出, 设计一个函数, 把指向字符串的指针和指向整数的指针作为函数形参。

(8) 建立一条无序链表, 每一个结点包含: 学号、姓名、C++成绩。设计一个函数建立链表, 另一个函数输出链表上各结点值, 再一个函数释放链表结点占用的动态存储空间。最后, 尝试将该链表元素按 C++成绩降序排序。

## 第9章 类和对象

面向对象编程和设计以类为基础。类(class)是一种用户定义数据类型。一个类是一组具有共同属性和行为的对象的抽象描述。面向对象的程序就是一组类构成的。一个类中描述了一组数据来表示其属性, 以及操作这些数据的一组函数作为其行为。一个类中的数据和函数都称为成员。对象是类的实例, 每个对象持有独立的数据值。本章主要探讨类和对象的基本概念。

面向对象编程具有封装性、继承性、多态性的特性, 本章主要介绍封装性的初步知识。

### 9.1 类

定义一个类就是描述其类名及其成员。对于成员, 还要描述各成员的可见性。本节也介绍了类与结构之间的区别。

#### 9.1.1 类的定义

如何定义一个类? 习惯上将一个类的定义分为两个部分: 说明部分和实现部分。说明部分包括类中包含的数据成员和成员函数的原型, 实现部分描述各成员函数的具体实现。一个类的一般格式如下:

```
//类的说明部分
class <类名>{
private:
    <一组数据成员或成员函数的说明>                //私有成员
protected:
    <一组数据成员或成员函数的说明>                //保护成员
public:
    <一组成员函数或数据成员的说明>                //公有成员, 外部接口
};
//类的实现部分
<各个成员函数的实现>
```

其中, class 是说明类的关键字; <类名>是一个标识符; 一对花括号表示类的作用域范围, 称为类体, 其后的分号表示类定义结束。

一个类中可以没有成员, 也可以有一组成员。成员可分为数据成员和成员函数两部分。一个数据成员描述了每个对象都持有的一个独立的值, 就像结构成员。一个成员函数描述了该类对象能被调用而提供的一项服务或一种计算。成员函数区别于普通函数, 就是在调用时



必须确定一个作用对象，也称为当前对象。

与结构一样，C14 允许对类中的非静态数据成员添加初始化。例如：

```
class Myclass{
    int a = 7;
    bool b ( true );
    decltype(3.14) c = 3.14; //double
};
```

成员初始化与变量初始化形式相同，只是不能用 `auto` 来自动推导成员类型，但可用 `decltype` 来推导成员类型。

关键字 `public`、`private` 和 `protected` 称为访问控制修饰符，描述了类成员的可见性。每个成员都有唯一的可见性。下一节详细介绍。

一个类中的成员没有前后次序，但最好把所有成员都按照其可见性放在一起。私有成员、保护成员、公有成员分别组成一组。这三组之间没有次序要求，而且每一组内的多个成员之间也没有次序要求。一个类不一定同时都具有这三组成员。

成员函数的实现既可以在类体内描述，也可以在类体外描述。如果一个成员函数在类体内描述，就不用再出现在类外的实现部分。如果所有的成员函数都在类体内实现，就可以省略类外的实现部分。在类体外实现的函数必须说明它所属的类名，格式如下：

<返回值> <类名>::<函数名>(形参表){...}

例 9-1，一个日期 `Date` 类。该类的每个对象都是一个具体的日期。例如，2009 年 4 月 3 日就是 `Date` 类的一个对象。编程如下：

```
#include <iostream>
using namespace std;
class Date{
private:
    int year, month, day;
public:
    void setDate(int y, int m, int d);
    bool isLeapYear();
    void print(){
        cout<<year<<". "<<month<<". "<<day<<endl;
    }
};
void Date::setDate(int y, int m, int d){
    year = y;
    month = m;
    day = d;
}
bool Date::isLeapYear(){
    return year%400 == 0 || year%4 == 0 && year%100 != 0;
}
int main(void){
    Date date1, date2;
    date1.setDate(2000,10,1);
    date2.setDate(2009,4,3);
    cout<<"date1: ";
    date1.print();
    cout<<"date2: ";
```

```

    date2.print();
    if (date1.isLeapYear())
        cout<<"date1 is a leapyear."<<endl;
    else
        cout<<"date1 is not a leapyear."<<endl;
    if (date2.isLeapYear())
        cout<<"date2 is a leapyear."<<endl;
    else
        cout<<"date2 is not a leapyear."<<endl;
    system("pause");
    return 0;
}

```

执行程序，输出如下：

```

date1: 2000.10.1
date2: 2009.4.3
date1 is a leapyear.
date2 is not a leapyear.

```

`Date` 类中定义了 3 个私有 `int` 型数据成员 `year`、`month` 和 `day`，分别表示某个日期的年、月、日。还定义了 3 个公有成员函数，`setDate` 函数用来为对象设置年月日；`isLeapYear` 函数判断是否为闰年，`print` 函数用来输出。其中 `print` 函数在类中给出实现，而另外两个函数在类外实现。在类外实现要在函数名之前添加类名和作用域运算符“`::`”。

在 `main` 函数中说明了 `Date` 类的两个对象 `date1` 和 `date2`。然后对这两个对象调用成员函数进行设置、输出等操作。

我们往往用图来描述类的结构。一个类的封装结构如图 9.1 所示。一个类可抽象为一个封装体，描述为一个矩形框，其中描述了类的名字。也可以在类名下面隔间里描述一组数据成员。比较完整的形式是用 3 个隔间，分别描述类名、数据成员和成员函数原型。一个成员占一行，成员前面的减号表示私有成员，加号表示公有成员。这样的图被称为类图。在类图上能清楚地看到一个类的成员构成，而暂时忽略成员函数内的实现细节。

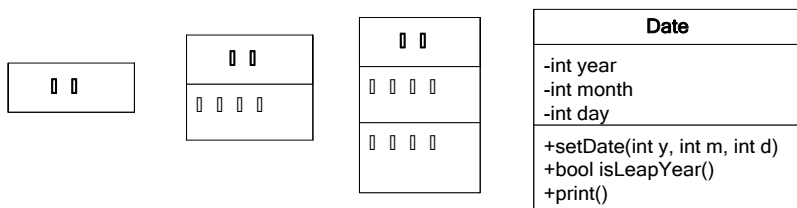


图 9-1 类的封装结构

类的外部代码只能看到类名和公有成员，就可用类名来创建对象，然后就调用公有成员来操作对象、改变或读取对象状态。

一个类应该有哪些成员？一个类往往反映现实存在的一个概念或一种实体，具有自身的一组属性。首先，封装性要求将这一组属性封装到类中，与该类不相关的属性不应纳入。其次，操作管理这些属性需要一组函数，通常进行写入、读出、计算等功能，这些函数应封装在该类中，而其它函数不应纳入。总之，一个类中的各个成员之间存在直接的语义关系。

### 9.1.2 类成员的可见性

一个类作为一个封装体，各成员具有不同的可见性。例如前面介绍的类 `Date`，成员函数 `setDate`、`isLeapYear` 和 `print` 是公有的，那么类外部程序可见，即可调用。而数据成员 `year`、`month` 和 `day` 是私有的，类外部程序就不可见，即不能访问。

C++提供了3种访问控制修饰符：`private`(私有)、`protected`(保护)和`public`(公有)。每个成员只能选择其中之一。

- 私有成员只允许本类的成员函数来访问，对类外部不可见。数据成员往往作为私有成员。
- 保护成员能被自己类的成员函数访问，也能被自己的派生类访问，但其它类不能访问。派生类将在第11章详细介绍。
- 公有成员对类外可见。当然类内部也能访问。公有成员作为该类对象的操作接口，使类外部程序能操作对象。成员函数一般作为公有成员。

按不同的可见性，一个类的各成员形成一种封装结构，如图9.2所示。一个类外部的函数或者其它类只能访问该类的公有成员。一个类 `A` 的派生类除了能访问类 `A` 的公有成员之外，还能访问类 `A` 的保护成员。类中的私有成员只能被该类中的其它成员访问。

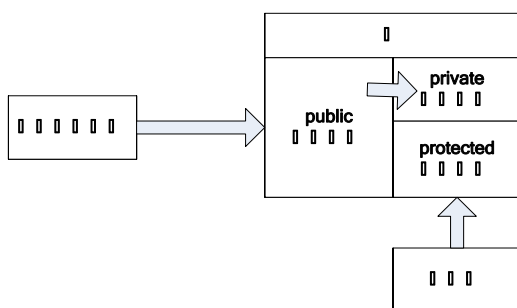


图 9-2 类成员的可见性

对于前面介绍的 `Date` 类，假如有一个全局函数如下：

```
void datePrint(Date & d){  
    cout<<d.year<<endl;           //错误：year 是私有成员  
    d.print();                     //正确：print 是公有成员  
}
```

这个函数 `datePrint` 并不是成员函数，形参是类 `Date` 的一个对象。由于 `year` 是类 `Date` 中的私有成员，因此在非成员函数中对私有成员 `year` 的访问是非法的。成员函数 `print` 是类 `Date` 中的成员，因此该函数中调用 `d.print()` 就是合法的。编译器能指出所有的非法访问。

为什么常把数据成员设置为私有，而将成员函数设置为公有？从类的内部来看，数据是被动的，只能被函数所改变，这样数据应该被隐藏在函数的后面。从类的外部来看，一个类不希望外部程序直接访问或改变对象内部的数据，虽然效率高，但不够安全，而希望通过调用自己类的成员函数来间接地访问数据。这样使调用方编程既安全，也简单，因为调用方不

需要知道类内部的数据构成，只需知道有哪些公有成员函数可供调用即可。

从形式上看，一个类中的访问权限修饰符可以任意顺序出现，也可出现多次，但一个成员只能具有一种访问权限。例如，类 **Date** 的定义可修改如下：

```
class Date{
public:
    void setDate(int y, int m, int d);
private:
    int year, month, day;
public:
    int isLeapYear();
    void print();
};
```

这种定义与前面定义完全相同。但最好将同一种可见性的成员放在一起。

可将类 **Date** 定义为如下形式：

```
class Date{
    int year, month, day;
public:
    void setDate(int y, int m, int d);
    int isLeapYear();
    void print();
};
```

C++约定缺省的访问权限为私有，即 **year**、**month** 和 **day** 作为私有成员。这种缺省方式要求将私有成员列在类的最前面。这样定义类 **Date** 的效果与前面定义完全相同。

### 9.1.3 类的数据成员

类中的数据成员描述对象的属性。数据成员在类体中进行定义，其定义方式与一般变量相同，但对数据成员的访问要受到访问权限修饰符的限制。

在定义类的数据成员时，要注意以下几个问题。

1、类中的数据成员可以是任意类型，包括基本类型、以及基本类型的数组、指针或引用，也可以是自定义类型，以及自定义类型的数组、指针或引用。但一个类的对象不可作为自身类的成员，而自身类的指针可作为该类的成员。例如：

```
class A{
    A a;           //错误
    A* pa;         //正确
};
```

原因是创建类的一个对象时，也要自动创建其成员的对象。

2、一个类中说明的多个数据成员之间不能重名。一个类作为一个作用域，不允许出现命名冲突。

3、一般来说，类定义在前，使用在后。但如果类 **A** 中使用了类 **B**，而且类 **B** 也使用了类 **A**，应该怎样处理？此时就需要“前向引用说明”。例如：

```
class B;           //前向引用说明，B 是一个类
class A{
    ...
public:
    void f(B b);    //类 A 使用了类 B
};
class B{           //类 B 的说明
```

```
...  
public:  
    void g(A a);           //类B使用了类A  
};
```

对于一个类，如何确定其数据成员至关重要，即一类对象应先确定要描述哪些属性和状态。下面从实际需求出发来分析几类对象的属性和状态。

- 把一个普通人作为一个对象时，就要描述其姓名、性别、出生日期、身份证号等属性。每一个人在任何时刻都具有这些属性的值来表示其状态。
- 把手机通讯录中的一个联系人作为一个对象，就要描述其姓名、固定电话、移动电话、电子邮箱等属性。
- 把一名大学生作为一个对象，除了要描述作为普通人的属性之外，还要描述其学号、专业、所在学院/系、入学日期、奖惩信息、课程成绩等属性。
- 把一名大学教师作为一个对象，除了要描述作为普通人的属性之外，还要描述其员工编号、专业、所在学院/系、职务、职称、开始工作日期、奖惩信息、代课信息等属性。
- 把一门大学课程作为一个对象，就要描述课程的名称、类别、编号、选修还是必修、学分、授课学时、上机学时、实践学时、上课学期、先修课程、内容简介等属性。
- 把二维平面上的一个点作为一个对象，就要描述其坐标，即(x,y)。每一个点都具有明确的坐标值来表示其状态。
- 把一个时刻作为一个对象，除了要描述日期的年、月、日属性之外，还要描述时、分、秒。

一般地，一个属性都表现为一个名词。类的一个属性表示了该类所有对象都所持有的一项信息。每个属性都具有确定的类型。例如，姓名应是一个字符数组或字符串 `string` 类型；性别应该是一个字符 `char`；出生日期应该是一个 `Date`。有些属性可用基本类型表示，而有些属性就需要自定义类型。

一个属性的值可能是单个值，也可能是多个值。例如，一名大学生具有多门课程的成绩。

对于一个属性，应区分原生属性还是派生属性。例如，一个人的出生日期是原生属性，而年龄只是一个派生属性，而不是原生属性，它可以由当前日期和出生日期计算出来，所以年龄一般不作为类的基本属性。可设置一个 `getAge()` 函数来返回一个人的年龄，而不能设计一个 `int age` 数据成员。

一个类的多个原生属性之间应该能独立改变，应尽量避免重复的或易产生冲突的设计。例如对于一门大学课程，通常有下面计算公式：

授课学时+上机学时+实践学时=总学时；总学时/16=学分。

如果在类中将这 5 个属性都作为原生属性而独立改变，就可能违背上面公式。比较合理的设计是将授课学时、上机学时、实践学时作为原生属性，而用 `getCredit()` 函数来计算学分。

在描述一类对象的属性时应根据实际需求，完整地描述，不能遗漏重要属性，也不能描述不相关的数据。例如，通讯录中的联系人可能就不需要描述其性别、出生日期、身份证号等属性。

一个类中的数据成员是最重要的部分。在某种程度上，数据决定了该类的成员函数。

### 9.1.4 类的成员函数

一个类的成员函数描述了该类对象的行为。对于一个类，一个成员函数表示了对该类对象所能执行的一种操作，目的是为了完成一项功能，而且向类外提供一种服务。

按面向对象设计惯例，类的成员函数一般是公有的，使该类的外部程序能调用。但成员函数内部实现的过程细节仍然不为外部程序所知。一个类的外部程序仅需知道该类中的公有成员函数的函数名、形参、返回值，就能调用这些成员函数来作用于特定对象。

在类的数据成员确定之后，设计一组成员函数有以下模式：

- 1、如果一个数据成员可以被改变，往往用一个 **setXxx**(一个形参)函数来实现，“Xxx”就是数据成员的名字，而且形参类型往往与数据成员的类型一致。
- 2、如果一个数据成员可被读取，往往用一个 **getXxx()**函数来返回这个数据成员，“Xxx”就是数据成员的名字，返回类型往往与数据成员的类型一致。
- 3、如果希望一个数据成员是只读的(**read only**)，即不能改变，那么该数据成员应设为私有，再设计一个公有的 **getXxx()**函数来读取它。
- 4、如果要从已有数据成员中计算并返回一个值，往往用一个 **getXxx()**成员函数来实现。

如果成员函数返回 **bool** 类型，往往用 **isXxx()**函数来实现。

在定义类的成员函数时，应注意以下问题：

- 1、对于一个私有数组成员，不能用一个函数来返回一个指针来指向该数组，这样外部程序就能随意改变各元素，那么私有可见性就形同虚设。例如：

```
class A{
    int a[3];                //私有数据
public:
    int* getA(){return a;}
    int getAvg(){return (a[0]+a[1]+a[2])/3;}
};

int main(){
    A a1;
    int *pa = a1.getA();
    pa[0] = 1;               //私有数据在类外被改变
    pa[1] = 2;
    pa[2] = 3;
    cout<<a1.getAvg()<<endl; //输出 2
    return 0;
}
```

- 2、基于同样的理由，不要设计公有成员函数来返回类中私有数据成员的指针或引用，否则会使私有访问权限失效。例如：

```
class A{
    double d;                //私有数据
public:
    double *getD(){return &d;} //返回私有数据的指针
    double &getDref(){return d;} //返回私有数据的引用
};

int main(){
    A a1;
    double *dp = a1.getD();
    *dp = 3.4;               //私有数据在类外被改变
}
```

```

    a1.getDref() = 4.5;    //私有数据在类外被改变
    return 0;
}

```

- 3、如果在类体外定义成员函数，必须在成员函数名前加上类名和作用域运算符(::)，但不要再添加可见性修饰符。作用域运算符用来标识一个成员属于某个类。格式如下：  
 <返回值> <类名>::<成员函数名>(<形参表>) { ... }
- 4、一个类中的多个成员函数可以重载(overload)，即函数名相同，但形参个数或类型不同。一个函数的名称及其形参作为一个整体，称为该函数的基调或特征(signature)。一个类中的各个成员函数都具有不同的基调。
- 5、对成员函数的最后几个形参能设置缺省值，但在调用时应注意区别于重载函数。
- 6、要调用一个成员函数，必须先确定一个作用对象，格式为：  
 <作用对象>.<成员函数名>(<实参表>)

例 9-2 设计一个类 **Person** 表示人，类图如图 9.3 所示。除了要表示一个人的姓名、性别

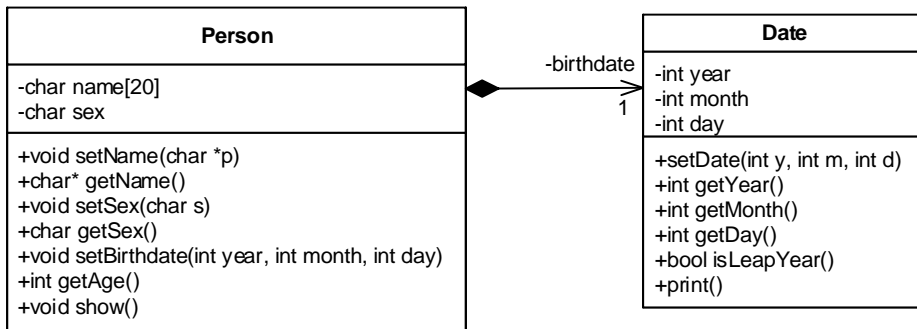


图 9-3 Person 类的设计

之外，还要表示一个人的出生日期，因此 **Person** 类使用了前面介绍的 **Date** 类，将 **Date** 类的一个对象作为自己的一个数据成员 **birthdate**。编程如下：

```

#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
#include <time.h>
using namespace std;
class Date{                                     //先说明 Date 类
    int year, month, day;
public:
    void setDate(int y, int m, int d){
        year = y;
        month = m;
        day = d;
    }
    int getYear(){return year;}
    int getMonth(){return month;}
    int getDay(){return day;}
    bool isLeapYear(){

```

```

        return year%400 == 0 || year%4 == 0 && year%100 != 0;
    }
    void print(){
        cout<<year<<". "<<month<<". "<<day;
    }
};

class Person{
    char name[20];
    char sex;          //f=女性, m=男性, u=未知
    Date birthdate;
public:
    void setName(char *p){
        if (strlen(p) <= 19)
            strcpy(name, p);
        else
            strcpy(name, "unknown");
    }
    char * getName(){ return name; }
    void setSex(char s){
        if (s == 'm' || s == 'f')
            sex = s;
        else
            sex = 'u';
    }
    char getSex(){return sex;}
    void setBirthdate(int year, int month, int day){
        birthdate.setDate(year, month, day);
    }
    int getAge(){
        //计算周岁
        time_t ltime = time(NULL);          //取得当前时间
        tm * today = localtime(&ltime);      //转换为本地时间
        int cyear = today->tm_year + 1900;    //取得当前年份
        int cmonth = today->tm_mon + 1;       //取得当前月份
        int cday = today->tm_mday;
        if (cmonth > birthdate.getMonth() ||
            cmonth == birthdate.getMonth() && cday >= birthdate.getDay())
            return cyear - birthdate.getYear();
        else
            return cyear - birthdate.getYear() - 1;
    }
    void show(){
        cout << (sex == 'f' ? "她是" : "他是") << getName();
        cout << ";出生日期为";
        birthdate.print();
        cout<<";年龄为"<<getAge()<<endl;
    }
};

int main(){
    Person a, b;
    a.setName("王翰");
    a.setSex('m');
    a.setBirthdate(1990,4,9);
    a.show();
    b = a;                      //A
    b.show();
    b.setName("李丽");
    b.setSex('f');
}

```



```
b.setBirthdate(1989,9,2);  
b.show();  
system("pause");  
return 0;  
}
```

执行程序，输出如下：

```
他是王翰;出生日期为 1990.4.9;年龄为 26  
他是王翰;出生日期为 1990.4.9;年龄为 26  
她是李丽;出生日期为 1989.9.2;年龄为 26
```

在 `Person` 类中对 3 个成员数据设计了一组函数。这些函数以 `setXxx` 或 `getXxx` 形式出现，被称为访问函数。其中“Xxx”表示某个属性的名字。`setXxx` 函数被称为设置函数(setter)，用于改变某个属性的值。`getXxx` 函数被称为读取函数(getter)，用于读取某个属性的值。在面向对象编程中，这种编程模式经常出现。

对于这种模式有这样疑问：对于一个属性，如 `sex`，既可以用 `setSex` 来改变，也能用 `getSex` 来读取，那为什么不把该属性设为公有 `public`，而要多加这两个访问函数？如果将该属性设为公有 `public`，这两个访问函数就无效了。但如果这样的话，我们就不能控制类外部程序任意来改变这个属性。我们对性别 `sex` 的值有如下约定：'f'表示女性，'m'表示男性，'u'表示未知，不允许其它值。如果外部程序可任意改变的话，那么上面约定就无效了。而外部程序想要合理使用 `Person` 类，就要自己建立约定，这样导致 `Person` 类仅仅是一个数据类，就与一个结构类型一样了，就不能发挥封装性的作用。一个类对于外部编码必须建立合理的、充分的、明确的、稳定的约定。外部程序只需遵循这种约定，就能简化自己的设计。不合理、不充分的约定、含糊或易变的约定都可能导致稍大规模的程序开发困难。

`Person` 类将 `Date` 类的一个对象 `birthdate` 作为自己的私有成员，但是 `Person` 类中的成员函数仍然不能直接访问对象 `birthdate` 中的私有成员：`year`、`month` 和 `day`，只能调用 `Date` 类中的公有函数。

`Person` 类中的 `getAge` 函数由出生日期来计算当前年龄，使用了 `time.h` 文件中定义的结构和函数。`show` 函数用于输出一个对象的当前状态，其中调用了本类中的一些函数和 `Date` 类的公有函数。

在 `main` 函数中创建了 `Person` 类的两个对象 `a` 和 `b`，然后分别调用公有成员函数来操作这两个对象。在 A 行将 `a` 赋给 `b`，就是将 `a` 的所有数据成员都复制给 `b`，所以你会看到前两行输出一样结果。

### 9.1.5 类与结构的区别

前面介绍了结构类型，只是定义了结构中的数据成员。这是传统 C 语言的用法。而 C++ 语言的结构类型中也能定义成员函数，就像类一样。结构中也可使用关键字 `private`、`public` 和 `protected` 来确定成员的可见性。

结构与类的一个重要区别是类成员的缺省访问权限为私有 `private`，而结构成员的缺省访问权限为公有 `public`。另一个区别是说明一个结构变量可用花括号来初始化，但创建一个对象时则不能用花括号初始化，只能用圆括号带实参来初始化，而这又需要编写构造函数才能实现。

如果只需描述一组相关数据,而不需要描述对这些数据的处理函数,建议使用结构类型,这样更简单。如果大多数数据成员和成员函数都是公有的,也适合采用结构。如果要求规范的封装性就应使用类。面向对象设计和编程主要是使用类,C++程序也往往使用结构、枚举等类型,但很少用结构来替代类。

## 9.2 对象

如何使用一个类?先创建该类的若干对象,然后操作这些对象来完成计算。下面介绍如何创建对象以及如何访问对象的成员。

### 9.2.1 对象的创建

对象是什么?一个对象(object)是某个类的一个实例。那么实例是什么?一个实例(instance)是某个类型经实例化所产生的一个实体。例如 3 就是 int 类型的一个实例。但通常 3 并不作为一个对象,这是因为 int 并不是一个类,而是一种类型。一个类是一种类型,但并非每一种类型都是类。

一个对象必须属于某个已知的类。在创建一个对象时,必须先说明该对象所属的类。创建一个新对象就是一个类的实例化。创建一个对象的一种格式如下:

<类名> <对象名>[(<实参表>)];

其中,<类名>是对象所属类的名字。<对象名>中可以有一个或多个对象名,多个对象名之间用逗号分隔。一个对象名之后可以用一对圆括号说明<实参表>,用来初始化该对象的一些数据成员,但这需要定义类的构造函数,构造函数将在下一章介绍。

创建一个对象数组的一种格式为:

<类名> <对象数组名>[正整数常量];

例如,对于前面介绍的 Person 类,有下面代码:

```
Person a, b;           //创建两个 Person 对象
Person ps[20];         //创建一个数组,包含 20 个 Person 对象作为该数组的元素
Person *pa = &a;       //说明 Person 类的一个指针,并指向对象 a
Person &rb = b;         //说明 Person 类的一个引用,并作为对象 b 的别名
Person *pa2 = new Person; //用 new 动态创建一个对象
delete pa2;            //用 delete 撤销一个对象
```

这种创建对象格式与前面介绍的基本类型和结构类型说明变量格式一样,不过后面还将介绍特殊的格式。

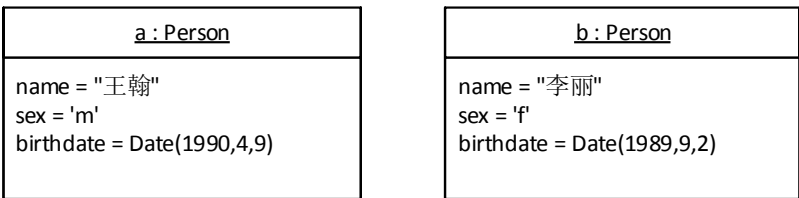


图 9-4 对象的结构

一个对象具有封装性,如图 9.4 所示。一个对象描述为一个封装体,先说明对象的名字

并在冒号后说明它所属的类名，用下划线表示它是一个实例。然后描述各个数据成员的名字及值。在描述一个对象时不需要描述其成员函数，这是因为同一类对象的成员函数都来自类的定义。实际上，创建对象时才需要为对象占据一块连续内存空间，这一块内存空间仅存放数据成员，而不包括成员函数，因此 `sizeof(Person)` 与 `sizeof(a)` 是一样的。

一个对象内可以包含其它类的一个或多个对象作为其成员对象。例如一个 `Person` 对象就包含了一个 `Date` 对象作为其成员对象。下一章将详细介绍成员对象。

### 9.2.2 访问对象的成员

操作一个对象是通过访问该对象的成员来实现的。一个对象的成员就是该对象的类中所定义的成员，包括数据成员和成员函数。如何访问对象的成员？说明对象后，就可以使用“.”运算符或者“->”运算符来访问对象的成员。其中，“.”运算符适用于一般对象和对象引用，而“->”运算符适用于对象指针。访问对象成员的一般格式如下：

<code>&lt;对象名&gt;.&lt;数据成员名&gt;</code>	或	<code>&lt;对象指针&gt;-&gt;&lt;数据成员名&gt;</code>
<code>&lt;对象名&gt;.&lt;成员函数名&gt;(&lt;实参表&gt;)</code>	或	<code>&lt;对象指针&gt;-&gt;&lt;成员函数名&gt;(&lt;实参表&gt;)</code>

无论哪一种格式都要求符合成员的访问权限控制。例如，假设 `a` 是 `Person` 类的一个对象：

```
a.setName("王翰");
```

对对象 `a` 调用了成员函数 `setName`，或者说，调用成员函数 `setName` 作用于对象 `a`。注意这个操作不能理解为，对象 `a` 自己调用了成员函数 `setName`。

由于成员函数 `setName` 是公有的，所以这条语句(确切说，该表达式)可以出现在任何地方。反之，如果被访问的成员是私有的，那么该表达式只能出现在该类的内部。

也可以通过对象指针来访问成员。例如，假设 `pa` 指向对象 `a`：

```
pa -> setSex('m');
```

用指针访问成员也能等价表示为指针间接引用方式，形式上有所不同，但本质上是相同的。

<code>&lt;对象指针&gt;-&gt;&lt;数据成员名&gt;</code>	等价于	<code>(*&lt;对象指针&gt;).&lt;数据成员名&gt;</code>
<code>&lt;对象指针&gt;-&gt;&lt;成员函数名&gt;(&lt;实参表&gt;)</code>	等价于	<code>(*&lt;对象指针&gt;).&lt;成员函数名&gt;(&lt;实参表&gt;)</code>

上面语句等价于下面语句：

```
(*pa).setSex('m');
```

可以看出，访问对象成员与访问结构体变量的成员是相同的。

一次只能操作一个对象的成员，没有简单办法能同时操作多个对象。

例 9-3 描述二维平面上的点 **Point**，一个点作为一个对象。建立一个 **Point** 类。**Point** 类的设计如图 9.5 所示。不仅能表示点的相对移动，还能计算两个点之间的距离。编程如下：

```
#include <iostream>
#include <math.h>
using namespace std;
class Point{
    int x, y;
public:
    void setPoint(int x, int y){
        //设置坐标
        this->x = x;
        this->y = y;
    }
    void moveOff(int xOff, int yOff){ //相对移动
        x += xOff;
        y += yOff;
    }
    int getX() { return x; }
    int getY() { return y; }
    double distance(Point & p){ //计算当前点与另一个点 p 之间的距离
        double xdifff, ydifff;
        xdifff = x - p.x; //访问 p 对象的私有成员 x
        ydifff = y - p.y; //访问 p 对象的私有成员 y
        return sqrt(xdifff*xdifff + ydifff*ydifff);
    }
    void show(){ //显示当前点对象的坐标
        cout<<"("<<getX()<<","<<getY()<<") "<<endl;
    }
};

int main(){
    Point p1, p2;
    p1.setPoint(1,2);
    p2.setPoint(3,4);
    cout<<"p1 is "; p1.show();
    cout<<"p2 is "; p2.show();
    cout<<"Distance is "<<p1.distance(p2)<<endl;//A
    p1.moveOff(5,6);
    p2.moveOff(7,8);
    cout<<"after move"<<endl;
    cout<<"p1 is "; p1.show();
    cout<<"p2 is "; p2.show();
    cout<<"Distance is "<<p1.distance(p2)<<endl;//B
    system("pause");
    return 0;
}
```

Point
-int x -int y
+void setPoint(int x, int y) +void moveOff(int xOff, int yOff) +int getX() +int getY() +double distance(Point &p) +void show()

图 9-5 Point 类的结构

执行程序，输出如下：

```
p1 is (1,2)
p2 is (3,4)
Distance is 2.82843
after move
```

```
p1 is (6,8)
p2 is (10,12)
Distance is 5.65685
```

成员函数 `setPoint` 用来设置坐标位置，其中使用了 `this->x` 来表示当前对象的 `x` 坐标，下一节将介绍 `this` 指针的作用。

成员函数 `distance` 用来计算当前点与形参点 `p` 之间的距离。在 `A` 行和 `B` 行调用了此函数，`p1` 作为当前对象，而将 `p2` 作为实参，这与 `p2.distance(p1)` 结果一样。

### 9.2.3 类与对象的关系

类是创建对象的样板，由这个样板可以创建多个具有相同属性和行为的对象。类是对对象的抽象描述，类中包含了创建对象的具体方法。在运行时刻，类是静态的，不能改变的。一个类的标识就是类的名称。C++程序中的类都是公共的，可以随时对一个类创建对象。

一个对象是某个类的一个实例。创建一个对象的过程被称为某个类的一次实例化。在运行时刻，对象才真正存在。对象是动态的，具有一定的生存期。一个对象在其生存期中不能改变它所属的类。每个对象都具有内在的对象标识，称为 `OID(object identity)`，由系统管理。程序员要通过对象的名字、数组下标、指针或引用来区分对象。

某个类的多个对象分别持有自己的成员变量的值，相互间是独立的。例如，`Point p1, p2;` 创建了 2 个对象，这两个对象分别持有自己的 `x` 和 `y`。当执行 `p1.setPoint(1,2);` 语句，改变 `p1` 的 `x` 和 `y` 时，对其它对象没有影响。但是我们不能认为每个对象都持有一份成员函数的拷贝。例如执行 `p2.setPoint(3,4);` 看起来 `p1` 和 `p2` 分别持有自己的 `setPoint` 函数，实际上不论创建多少对象，成员函数的空间只在多个对象之间的共享，而没有独立的拷贝，否则内存空间浪费太大，也没有这个必要。

## 9.3 this 指针

在前面 `Point` 类中说明了 `setPoint` 成员函数如下：

```
void setPoint(int x, int y){
    this->x = x;
    this->y = y;
}
```

函数体中的 `this` 是什么？C++为每个成员函数都提供了一个特殊的对象指针——`this` 指针，它指向当前作用对象，使成员函数能通过 `this` 指针来访问当前对象的各成员。`this` 指针的类型为：“`class <类名>*`”。

其中，`<类名>`就是当前类的名字。另外，`this` 指针是常量，不能在函数中改变它，使其指向其它对象，它只能指向当前作用对象。

当前作用对象是什么？当前作用对象就是由成员函数调用时所确定的一个对象。例如，当程序执行 `p1.setPoint(1,2);` 时，`p1` 对象就是 `setPoint` 函数的当前作用对象，也称为当前对象。每个非静态的成员函数在运行时都要确定一个当前作用对象，所以在每个非静态的成员函数内都可以使用 `this` 指针。

`this` 指针是隐含的，它隐含于每个类的成员函数中。例如，另一个成员函数：

```
void moveOff(int xOff, int yOff){  
    x += xOff; //等价于 this->x += xoff;  
    y += yOff; //等价于 this->y += yoff;  
}
```

函数体中的访问成员 `x` 和 `y` 都隐含着 “`this->x`” 和 “`this->y`”，表示访问当前作用对象的 `x` 和 `y`。当调用一个对象的成员函数时，编译程序先将对象的地址赋给 `this` 指针，然后调用该成员函数，每次成员函数访问数据成员时，则隐含地使用了 `this` 指针。

通常，`this` 指针的使用都是隐含的，但在特定场合必须显式地使用它。例如上面的 `setPoint` 函数。如果省去 “`this->`”，将变成 “`x = x; y = y;`”。而这里的 `x` 和 `y` 指的是函数的形参，而不是类 `Point` 的成员。语法虽然没有错，但达不到给成员 `x` 和 `y` 赋值的目的。所以 `setPoint` 函数中不能省略 “`this->`”。

关于 `this` 指针的使用要说明两点。

1、`this` 指针是一个 `const` 型常量指针，因此在成员函数内不能改变 `this` 指针的值，但能通过 `this` 指针来改变对象的值。就像 “`this->x = x;`”。

2、只有非静态成员函数才有 `this` 指针，静态成员函数没有 `this` 指针。静态成员函数将在下一章介绍。

## 9.4 类中还有什么？

在一个类中，除了数据成员和成员函数之外，还可定义以下内容：

- 构造函数和析构函数：构造函数用来描述该类在创建一个对象过程中如何对数据成员进行初始化，而析构函数则描述撤销一个对象时要执行的一些必要清理工作。在 10.1 和 10.2 节介绍。
- 特殊的数据成员：
  - ◆ 复合对象和成员对象：如果一个数据成员是另一个类的对象，那么当前类就是一个复合对象类，该数据成员就是一个成员对象。此时要求构造函数和析构函数按特定次序执行。在 10.3 节介绍。
  - ◆ 对象数组作为数据成员：如果一个数据成员是一个类的一个数组，此时数组元素类的构造函数就有特殊要求，在 10.4 节介绍。
  - ◆ 静态成员：与结构中的静态成员类似，静态数据成员表示该类所持有的变量，而非对象所持有的变量。在 10.6 节介绍。也包括静态成员函数。
  - ◆ 指针成员：如果一个数据成员是指针类型，就表示动态内存管理或者对象之间的关联，就要求特殊的拷贝构造函数、赋值操作函数和析构函数设计，在 10.7 节介绍。
  - ◆ 引用成员：如果一个数据成员是引用类型，表示对象之间一种紧密关联，也要求特别的构造函数、拷贝构造函数、赋值操作函数和析构函数设计，在 10.8 节介绍。
- 基类：一个类作为某个类的派生类，表示一种继承性关系。在 10.1 和 10.2 节介绍，这要求派生类的构造函数应以合理方式来调用基类的某个构造函数。还要选择继承方式，有 3 种继承方式可选，缺省为私有的，但常用的是公用继承。

- 虚基类：避免在多继承时产生二义性，用 `virtual` 修饰继承关系。11.4 节介绍。
- 特殊的成员函数：
  - ◆ 虚函数：可以被派生类改写的函数，在运行时刻实现动态的多态性，用 `virtual` 修饰函数。11.6 节介绍
  - ◆ 纯虚函数：只说明函数原型和语义而不提供实现的虚函数，在 11.6 节介绍。
  - ◆ 运算符重载函数：用成员函数实现指定的运算符，可简化对象操作，用 `operator` 开头的函数。12.1 节介绍。
- 友元函数说明：在类中可说明类外某个函数作为其友元函数，使之能访问类内所有成员，用来实现特定的运算符重载函数。在 12.2 节介绍。
- 在类中还可定义嵌套类、枚举类型、类型定义，这些定义有可见性控制。
  - ◆ 嵌套类：在一个类中 A 可以定义另一个类 B，用 `A::B` 的形式来访问嵌套类。类中也可定义结构。
  - ◆ 枚举：在类中可定义枚举类型，使枚举常量作为类的命名常量。例如 `ios` 类中定义了枚举类型 `open_mode`。
  - ◆ 类型定义：在类 A 中可用 `typedef` 定义类型 B，用 `A::B` 的形式来访问类型 B。在类模板中经常有这种类型定义。

## 9.5 小 结

- 类(class)是一种用户定义数据类型。一个类是一组具有共同属性和行为的对象的抽象描述。一个类中描述了一组数据来表示其属性，以及操作这些数据的一组函数作为其行为。类中的数据和函数都称为成员。
- 一个类中的成员分为数据成员和成员函数。
- 成员函数的实现既可以在类体内描述，也可以在类体外描述。
- 成员有 3 种访问控制修饰符：`private`(私有)、`protected`(保护)和 `public`(公有)。每个成员只能选择其中之一。
  - ◆ 私有成员只允许本类的成员函数来访问，对类外部不可见。数据成员往往作为私有成员。
  - ◆ 保护成员能被自己类的成员函数访问，也能被自己的派生类访问，但其它类不能访问。
  - ◆ 公有成员对类外可见。当然类内部也能访问。公有成员作为该类对象的操作接口，使类外部程序能操作对象。成员函数一般作为公有成员。
- 一个类的对象不可作为自身类的成员，而自身类的指针可作为该类的成员。
- 类中不允许对数据成员进行初始化。
- 一个类中说明的多个数据变量之间不能重名。一个类作为一个作用域，不允许出现命名冲突。
- 对于一个私有数组成员，不能用一个函数来返回一个指针来指向该数组。

- 不要在函数中返回类中的私有数据成员的指针或引用。
- 一个类中的多个成员函数可以重载(overload), 即函数名相同, 但形参数或类型不同。
- 对成员函数的最后几个形参能设置缺省值, 但在调用时应注意区别于重载函数。
- 调用成员函数, 必须先确定一个作用对象。
- 结构与类有很多相似。区别在于成员的缺省访问权限和初始化。
- 一个对象是某个类的一个实例。每个对象持有独立的数据成员的值。
- 在创建一个对象时, 必须先说明该对象所属的类。
- 通过一个对象或者一个对象指针能访问对象的成员。要使用成员访问运算符。
- 一个对象可以赋值给同类的另一个对象, 缺省情况下复制各个数据成员。
- 在运行时刻, 类是静态的, 不能改变的。在运行时刻, 对象才真正存在。对象是动态的, 具有一定的生存期。
- 每个非静态成员函数都隐含有一个 `this` 指针, 指向当前作用对象。当访问当前对象的成员时自动起作用, 但如果成员与函数形参或局部变量发生命名冲突时就需要显式指明。在成员函数内不能改变 `this` 指针的值。

## 9.6 练 习 题

1. 关于类的成员, 下面哪一种说法是错误的?
  - A 类中的一个数据成员表示该类的每个对象都持有的一个值。
  - B 调用类中的一个成员函数必须确定一个作用对象。
  - C 类中至少应包含一个成员。
  - D 类中的各个成员的说明没有严格次序。
2. 关于类的成员的可见性, 下面哪一种说法是错误的?
  - A 私有 `private` 成员只能在本类中访问, 而不能被类外代码访问。
  - B 一般将类的数据成员说明为私有成员, 但不是绝对的。
  - C 公有 `public` 成员能被类外代码访问, 而不能被同一个类中的代码访问。
  - D 一般将类的成员函数说明为公有成员, 但不是绝对的。
3. 关于类的数据成员, 下面哪一种说法是错误的?
  - A 假设一个类名为 `A`, 那么 “`A a;`” 不能作为类 `A` 的数据成员。
  - B 在说明一个数据成员时, 可以说明其初始化值, 就像函数中说明一个变量一样。
  - C 类中的多个数据成员变量不能重名。
  - D 如果有两个数据成员的可见性不同, 它们就可以重名。
4. 关于类的成员函数, 下面哪一种说法是错误的?
  - A 一般来说, 一个类的成员函数对该类中的数据成员进行读写计算。
  - B 如果一个数据成员希望是只读的, 那么该成员应说明为私有的, 而且用一个公有的 `getXxx` 成员函数来读取它的值。
  - C 一个类中的一组成员函数不能重名。



D 公有成员函数不应该返回本类的私有成员的指针或引用。

5. 关于类与对象，下面哪一种说法是错误的？

A 一个对象是某个类的一个实例。

B 一个实例是某个类型经实例化所产生的一个实体。

C 创建一个对象必须指定被实例化的一个类。

D 一个类的多个对象之间不仅持有独立的数据成员，而且成员函数也是独立的。

6. 关于对象成员的访问，下面哪一种说法是错误的？

A 对于一个对象，可用“.”运算符来访问其成员。

B 对于一个对象引用，可用“->”运算符来访问其成员。

C 如果被访问成员是公有的，该访问表达式可以出现在 main 函数中。

D 如果被访问成员是私有的，该访问表达式只能出现在类中。

7. 关于 this 指针，下面哪一种说法是错误的？

A 每个非静态成员函数都隐含一个 this 指针。

B this 指针在成员函数中始终指向当前作用对象。

C 在成员函数中直接访问成员 m，隐含着 this->m。

D 在使用 this 指针之前，应该显式说明。

8. 定义一个类 Cat 来描述猫，一只猫作为一个对象，应描述 age、weight、color 等属性，以及对这些属性的读写函数。实现并测试这个类。

9. 设计一个矩形类 Rectangle，要求有下述成员：

数据成员：左上角坐标(x, y)，宽度 width 和高度 high，可用 int 类型。

成员函数：对以上数据成员的读 getXxx 和写 setXxx。另外：

void move(int, int): 相对移动，从一个位置移动到另一个位置。

int getArea(): 计算矩形的面积。

实现并测试这个类。

## 第10章 构造函数与析构函数

在运行时刻，对象具有生命周期。从创建对象开始，然后调用其成员函数来操作对象，最后对象被撤销，结束其生命周期。当从一个类创建一个对象时，就要为新对象分配存储空间，并进行必要的初始化。这是由构造函数来完成的，每个对象都经过构造函数的执行来初始化，每个类都有构造函数，可以显式定义，也可由系统自动生成。一个类中可以有多个构造函数，具有不同作用。

当对象被撤销时，要自动调用析构函数。每一个类均有一个构造函数，它可以显式定义，也可由系统自动生成。

本章介绍构造函数和析构函数，进一步介绍类中的各种成员，包括复合对象与成员对象、对象数组、静态成员、指针成员等。

### 10.1 构造函数

每个类都有构造函数，由系统调用，用于创建该类的对象并进行初始化。

#### 10.1.1 构造函数的定义

构造函数(constructor)是一种特殊的函数，其作用是在创建对象时，由系统来调用，对新建对象的状态进行初始化。

构造函数有以下特点：

- (1) 名字必须与类名相同。
- (2) 不指定返回值类型。
- (3) 可以无参，也可有多个形参，因此一个类中可重载定义多个构造函数。
- (4) 创建一个对象时，系统会根据实参来自动调用某个构造函数。

构造函数的格式如下：

类名(形参表)：成员初始化表 {函数体}

其中，成员初始化表中包含对本类多个数据成员的初始化，格式为：成员名(初始值)，其中的初始值往往来自构造函数的形参。一个成员的初始化相当于函数体中的一条语句：

成员名 = 初始值;

创建对象至少包括以下三种情形：

- 说明一个对象变量或数组，也包括函数形参对象。
- 使用 new 运算符动态创建对象。
- 创建临时匿名对象，后面介绍。

构造函数一般是公有的，使类外代码能按形参要求来创建对象。在特定情况下，构造函

数也可能作为私有，以限制外部程序随意创建对象。

例 10-1 对 **Date** 类添加构造函数，编程如下：

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <time.h>
using namespace std;
class Date{
    int year, month, day;
public:
    Date(int y, int m, int d)           //构造函数 1
        : year(y), month(m), day(d) {}
    Date(){}                           //构造函数 2
        time_t ltime = time(NULL);     //取得当前时间
        tm * today = localtime(&ltime); //转换为本地时间
        year = today->tm_year + 1900;   //取得当前年份
        month = today->tm_mon + 1;      //取得当前月份,0-11
        day = today->tm_mday;           //取得当前日,1-31
    }
    int getYear(){return year;}
    int getMonth(){return month;}
    int getDay(){return day;}
    bool isLeapYear(){
        return year%400 == 0 || year%4 == 0 && year%100 != 0;
    }
    void print(){
        cout<<year<<". "<<month<<". "<<day;
    }
};

int main(void){
    Date date1(2002,10,1),date2;       //A
    cout<<"date1: ";
    date1.print();
    cout<<"\ndate2: ";
    date2.print();
    if (date2.isLeapYear())
        cout<<"\ndate2 is a leapyear."<<endl;
    else
        cout<<"\ndate2 is not a leapyear."<<endl;
    system("pause");
    return 0;
}
```

执行程序，输出如下：

```
date1: 2002.10.1
date2: 2016.7.26
date2 is not a leapyear.
```

类 **Date** 中重载定义了两个构造函数，第 1 个构造函数有 3 个形参，分别对应年、月、日。第 2 个构造函数没有形参，取出系统当前日期来初始化新建对象。注意构造函数的名字就是类的名字 **Date**，而且构造函数没有返回值，也不能指定为 **void**。构造函数形参也可设缺省值。

A 行创建了 2 个对象 **date1** 和 **date2**，由系统自动调用构造函数。在创建对象 **date1** 时，调用了第 1 个构造函数，有 3 个形参。在创建对象 **date2** 时，调用了第 2 个构造函数，无参构造函数。从执行结果能看出 **date2** 得到了当前日期。

构造函数也可以在类体中说明，在类外定义。这与普通成员函数一样。例如：

```
Date::Date(int y, int m, int d){...} //构造函数 1
```

构造函数的好处是明确规范了类的使用者如何来创建对象，要求提供什么实参来初始化新建对象的状态，从而简化了类的使用。

### 10.1.2 缺省构造函数

每个类都要有构造函数，否则就不能实例化创建对象。如果一个类中没有显式定义任何构造函数，那么编译器将自动生成一个无参的公有的构造函数，该构造函数就是一个缺省构造函数(default constructor)。自动生成的缺省构造函数的函数体是空的。前面介绍的类中没有定义构造函数而能创建对象，就是因为调用了自动生成的缺省构造函数。如果类中显式定义了一个构造函数，那么系统就不会再自动生成缺省构造函数。

类中也可以显式定义一个缺省构造函数。如果一个构造函数无参，或者所有形参都有缺省值，它也是一个缺省构造函数。

缺省构造函数的好处是简化了类的实例化过程，不需要提供任何实参就能创建对象。缺省提供的构造函数不做任何初始化，因此所有数据成员的值都可能是随机值。因此类的设计往往要求显式定义缺省构造函数，对缺省初始化提供一种确定的实现。例如前面 **Date** 类的缺省构造函数就是将当前日期作为缺省初始化。

一个类最多只能有一个缺省构造函数。在上面类 **Date** 中，第二个构造函数就是显式定义的唯一的一个缺省构造函数。你不能再重载定义一个缺省构造函数，使形参带缺省值而成为另一个缺省构造函数。

一个类中也可以没有缺省构造函数。在上面 **Date** 类中，如果删除缺省构造函数，那么 **A** 行中创建 **date2** 就不允许，这是因为类中只有第 1 个构造函数，它要求 3 个实参。

一般情况下，构造函数都是公有的，但特殊情况下，保护的或者私有的构造函数也是有用的。此时类外程序就不能直接创建该类的对象，该类往往要提供一个静态函数来创建对象并返回对象的指针，使类外程序能通过该指针来操作对象。此时往往要对创建对象的条件进行某些限制。

### 10.1.3 委托构造函数

同一个类中往往有多个构造函数，如果多个构造函数具有一些相同行为，要避免重复编码就有两个办法。一个办法是建立一个成员函数，然后让多个构造函数来调用。另一个办法就是委托构造函数。

一个构造函数是否允许，以及如何调用另一个构造函数？在 C11 之前，一个构造函数不能调用另一个。C11 提出的委托构造函数允许调用。

委托构造函数是调用本类另一个构造函数的机制。被调用方称为目标构造函数(target constructor)，调用方称为委托构造函数(delegating constructor)。委托构造函数的语法形式如下：

类名(形参表)：类名(实参表){函数体}

其中实参表往往是形参、字面值或表达式。注意，实参表基调设计应避免调用自己。

执行过程是，当该构造函数开始执行，先按实参表去执行一个目标构造函数，然后再执

行自己的函数体。

委托构造函数有两个限制：1，不能添加成员初始化；2，不能在函数体中调用目标构造函数(所有构造函数体中都不允许)。

对目标构造函数没有特殊限制。下面是一个例子：

```
class X {
    int type = 1;
    char name = 'a';
    void initRest() { /* 其他初始化 */ }
public:
    X() { initRest(); }
    X(int x) : X() { type = x; }
    X(char e) : X() { name = e; }
};
```

上面第 1 个构造函数调用了一个私有的成员函数 `initRest`。

第 2 个构造函数是一个委托构造函数，先调用第 1 个构造函数，然后函数体中对 `type` 成员初始化。

第 3 个构造函数也是一个委托构造函数，先调用第 1 个构造函数，然后函数体中对 `name` 成员初始化。

从上面例子可以看出，2 个委托构造函数的函数体中包含了数据成员的初始化，而且这种初始化不能移动到成员初始化中。此时如果函数包含了对任何一个数据成员的改变，委托构造函数就容易出错了。

可设计私有的目标构造函数，使委托构造函数得到简化并避免可能的错误：

```
class X {
    int type = 1;
    char name = 'a';
    X(int i, char e) : type(i), name(e) { /* 其他初始化 */ }
public:
    X() : X(1, 'a') { }
    X(int x) : X(x, 'a') { }
    X(char e) : X(1, e) { }
};
```

委托构造函数自己也可能作为目标构造函数。下面修改无参构造函数如下：

```
X() : X(1) { } //调用下一个
X(int x) : X(x, 'a') { } //目标函数，同时也是委托函数
```

这样就形成一种链式委托构造，注意不能形成委托环(delegation cycle)。

## 10.2 析构函数

析构函数(destructor)与构造函数的作用相反，用来完成对象被撤销前的扫尾清理工作。析构函数是在撤消对象前由系统自动调用的，析构函数执行后，系统回收该对象的存储空间，该对象的生命周期也就结束了。

析构函数是类中的一种特殊的函数，它具有以下特性：

- (1) 析构函数名是在类名前加“~”构成。该符号曾作为按位求反的单目运算符。
- (2) 不指定返回类型。

(3) 析构函数没有形参, 因此也不能被重载定义, 即一个类只能有一个析构函数。

(4) 在撤销一个对象时, 系统将自动调用析构函数, 该对象作为析构函数的当前对象。

(5) 如果没有显式定义析构函数, 编译器将生成一个公有的析构函数, 称其为缺省析构函数, 函数体为空。

在以下三种情况下要撤销对象调用析构函数:

- ◆ 当程序执行离开局部对象所在的作用域时, 要撤销局部对象。当程序终止时, 要撤销全局对象和静态对象。
- ◆ 用 `delete` 运算符回收先前用 `new` 创建的对象。
- ◆ 临时匿名对象使用完毕。

例 10-2 修改前面 `Date` 类, 添加析构函数, 并对构造函数和析构函数添加输出加以提示, 以分析每个对象的生命周期。编程如下:

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <time.h>
using namespace std;

class Date{
    int year, month, day;
public:
    Date(int y, int m, int d)                //构造函数 1
        : year(y), month(m), day(d) {
        cout<<"Constructor1 of Date:";
        print();
        cout<<endl;
    }
    Date() {                                //缺省构造函数
        time_t ltime = time(NULL);
        tm * today = localtime(&ltime);
        year = today->tm_year + 1900;
        month = today->tm_mon + 1;
        day = today->tm_mday;
        cout<<"Default Constructor of Date:"; //添加的输出
        print();
        cout<<endl;
    }
    ~Date() {                               //析构函数
        cout<<"Destructor of Date:";        //添加的输出
        print();
        cout<<endl;
    }
    int getYear(){return year;}
    int getMonth(){return month;}
    int getDay(){return day;}
    bool isLeapYear(){
        return year%400 == 0 || year%4 == 0 && year%100 != 0;
    }
    void print(){
        cout<<year<<"."<<month<<"."<<day;
    }
};

void test(){
    Date date1(2002, 10, 1);                //A
    cout << "date1: "; date1.print(); cout << endl;
```

```
Date *date2 = new Date; //B
cout << "date2:";      date2->print(); cout << endl;
delete date2;          //C
date1 = Date(2003, 12, 12); //D
cout << "date1: "; date1.print(); cout << endl;
}
int main(){
    test();
    system("pause");
    return 0;
}
```

执行程序, 输出如下(行号是为了方便说明而加的):

```
1 Constructor1 of Date:2002.10.1
2 date1: 2002.10.1
3 Default Constructor of Date:2016.7.26
4 date2:2016.7.26
5 Destructor of Date:2016.7.26
6 Constructor1 of Date:2003.12.12
7 Destructor of Date:2003.12.12
8 date1: 2003.12.12
9 Destructor of Date:2003.12.12
```

执行 A 行调用了第 1 个构造函数, 输出第 1 行。B 行用 `new` 来创建一个对象, 调用了缺省构造函数, 输出了第 3 行。C 行用 `delete` 撤销此对象, 输出了第 5 行。D 行先创建一个临时匿名对象, 输出了第 6 行, 完成赋值之后该对象自动撤销, 输出了第 7 行。最后当 `main` 函数执行结束, `date1` 对象作为局部对象被撤销, 输出最后 1 行。

何时需要自行定义析构函数? 如果类的数据成员中含有指针, 而且在构造函数中用 `new` 来动态申请内存, 此时就需要自行定义析构函数, 用 `delete` 来动态回收内存。实际上, 此时也要求自行定义拷贝构造函数和赋值操作函数。

一般情况下, 析构函数都是公有的, 但在特殊情况下, 可能定义私有的析构函数, 这样就能阻止创建局部对象和 `delete` 命令的编译。可以通过该类的静态成员函数来撤销对象, 此时往往要对某些条件进行判断来限制是否要撤销对象。

### 10.3 拷贝构造函数与单参构造函数

一个类不仅自动生成缺省构造函数, 还会生成拷贝构造函数和赋值操作函数。单参构造函数具有隐式类型转换的作用。注意下面所介绍的概念与规则依据 C11/C14 标准。下面例子运行在 VS2015, DevC++ 需要添加编译选项 `-std=c++1y`。

#### 10.3.1 拷贝构造函数

创建一个对象有两种来源: 要么是从类中创建而来, 要么是从一个已有的同类对象复制而来。后者就需要调用拷贝构造函数。

拷贝构造函数(copy constructor)是一种特殊的构造函数, 用一个已有的同类对象来初始化新建对象, 并复制对象的所有数据成员。拷贝构造函数有一个特殊的形参, 格式如下:

```
<类名>::<类名>(const <类名>& <对象名>)
```

：成员初始化表 { 函数体 }

拷贝构造函数是一种特殊的构造函数，它只有一个形参，就是同类对象的一个引用。修饰词 `const` 表示函数体中不能改变被复制对象的状态。

每一个类中都有一个拷贝构造函数。如果类中没有显式定义拷贝构造函数，编译器就自动生成一个公有的拷贝构造函数，而且函数体中自动复制每个非静态数据成员到新建对象。例如前面 `Date` 类例子中，没有显示定义拷贝构造函数，就会生成一个拷贝构造函数。

在前面例 10-2 的 `main` 函数的最后添加如下语句：

```
Date date3(date1);           //A
date3.print();
Date date4 = date1;          //B
date4.print();
```

A 行创建一个 `Date` 对象 `date3`，并在圆括号中提供一个实参 `date1`，表示在创建对象 `date3` 时，用对象 `date1` 的状态来初始化新建对象，就是复制了 `date1` 对象中的 3 个数据成员到 `date3` 中。A 行调用了缺省提供的拷贝构造函数。

B 行是调用拷贝构造函数的另一种形式。这种形式类似于基本类型变量说明加初始化，如：`int i = j;` `j` 是前面已说明的同类变量。这种形式虽然用了赋值运算符，但不是赋值语句，而是变量说明语句。

在以下情形会调用拷贝构造函数：

1、用说明语句来创建一个对象时，用一个已有对象来初始化新建对象。如 A 行 B 行。

2、调用某个函数，该函数形参是以值传递一个对象，实参是一个已有对象。

3、返回一个对象时，是否执行拷贝构造函数与编译优化选项有关。C14 优化了临时对象的创建和撤销。如果编译选项未启动返回值优化(RVO, Return Value Optimization)或命名返回值优化(NRVO, Named Return Value Optimization)，就会执行拷贝构造函数，创建临时对象。如果启动优化，就不执行拷贝构造函数。VS2015 在 Debug 版本配置中不启动优化，而在 Release 版本配置中启动优化。DevC++ 默认启动优化。下面以优化方式运行。

下面对 `Date` 类定义拷贝构造函数，在复制数据成员之后添加一个输出语句，使我们能看到拷贝构造函数得到执行。

```
Date(const Date & d)
:year(d.year), month(d.month), day(d.day){
    cout<<"Copy constructor of Date:";//添加输出
    print();
    cout<<endl;
}
```

然后将 `Date` 类封装到一个单独文件中 `date1.h`，注意去掉其中 `main` 函数，使下面其它程序能使用。

例 10-3 拷贝构造函数的调用执行。编程如下：

```
#include "date1.h"
void fun1(Date d){
    d.print();
    if (d.isLeapYear())
        cout<<"，是闰年\n";
    else
        cout<<"，不是闰年\n";
}
Date getToday(){
```



```

    Date d;
    return d;
}
void test(){
    Date date1(2000, 1, 1);           //A
    Date date2(date1);                //B
    Date date3 = date2;               //C
    fun1(date1);                      //D
    Date date4 = getToday();          //E
    cout << "date4:"; date4.print(); cout << endl; //F
    getToday();                      //G
}
int main(){
    test();
    system("pause");
    return 0;
}

```

执行程序，输出如下(行号是为了方便说明而加的):

```

1 Constructor1 of Date:2000.1.1
2 Copy constructor of Date:2000.1.1
3 Copy constructor of Date:2000.1.1
4 Copy constructor of Date:2000.1.1
5 2000.1.1, 是闰年
6 Destructor of Date:2000.1.1
7 Default Constructor of Date:2016.7.26
8 date4:2016.7.26
9 Default Constructor of Date:2016.7.26
10 Destructor of Date:2016.7.26
11 Destructor of Date:2016.7.26
12 Destructor of Date:2000.1.1
13 Destructor of Date:2000.1.1
14 Destructor of Date:2000.1.1

```

A 行调用了第 1 个构造函数，输出第 1 行。

B 行调用了拷贝构造函数，输出第 2 行。

C 行也调用了拷贝构造函数，输出第 3 行。

D 行用 `date1` 做实参调用函数 `fun1`，这是传值调用方式，调用拷贝构造函数，将实参对象传给形参对象 `d`，输出了第 4 行。执行函数，输出第 5 行。当函数返回时，形参对象 `d` 被撤销，输出第 6 行。

E 行调用 `getToday` 函数，该函数中调用缺省构造函数创建一个对象 `d`，输出第 7 行显示当前日期。然后 `return` 语句返回对象，此时不执行任何构造函数，将对象 `d` 传给一个新建对象 `date4`。注意，E 行并未执行拷贝构造函数，也没有执行析构函数。

F 行执行输出第 8 行。

G 行再次调用 `getToday` 函数，但对返回对象不做处理。可以看到，先调用缺省构造函数创建对象 `d`，输出第 9 行显示当前日期。然后 `return` 语句执行，撤销局部对象 `d`，输出第 10 行。

至此 `test` 函数结束，有 4 个局部对象要撤销，按 `date4`、`date3`、`date2`、`date1` 的次序执行析构函数，分别输出第 11 行到第 14 行。

调用一个以值传递对象的函数，注意区别传递的是匿名临时对象还是命名对象，分别执行不同的过程。

```

void g(A a) {}           //g 函数以值传递对象 A
void test1() {
    g(A());              //先构造匿名对象给形参 a，形参 a 析构
    A a;                 //构造
    g(a);                //传递命名对象，先拷贝构造，形参 a 析构，
}                        //变量 a 析构

```

调用一个返回对象的函数，至少有以下三种形式，分别执行不同过程：

```

A f() { return A(); }   //f 函数返回 1 个 A 对象
void test2() {
    f();                 //先构造，后析构
    A b = f();           //仅构造一次
    b = f();             //f 先构造，b 构造，赋值操作，f 析构
}                        //b 析构

```

上面最后一行代码调用了赋值操作函数，下面介绍。

如果一个类中仅显式定义了一个拷贝构造函数，而没有定义其它构造函数，该类能否创建对象？回答是否定的。因为编译器不会提供缺省构造函数，而调用拷贝构造函数执行需要一个已有的对象，此时又不能提供，故此该类就不能直接创建对象。

如果一个类中所有构造函数都是私有的，该类能否创建对象？此时类外程序不能直接创建对象，但可以设计静态的公有的成员函数来创建对象并返回对象。对于一些特殊的类，只能创建单个对象，就不能将构造函数作为公有。

### 10.3.2 赋值操作函数

用赋值语句把一个对象赋给另一个已有的同类对象时，将调用该类的赋值操作函数。赋值操作函数的一般格式如下：

```

<类名>& <类名>::operator =(const <类名>& <对象名>){
    ...                //赋值操作函数体
    return *this;
}

```

例如，语句：date2=date3;就是调用函数 data2.operator=(date3);

赋值操作函数具有以下特点：

- (1) 函数名为 **operator =**，一种特殊的运算符重载函数。
- (2) 有一个形参，对该类对象的常量左值引用，与拷贝构造函数一样。
- (3) 赋值操作函数是一个成员函数，而不是构造函数，因此必须说明其返回类型。其返回值是赋值语句的左值对象的引用，就是赋值运算符“=”左边的对象。函数体中返回语句一般是“return \*this;”。在赋值语句中左值对象就是当前对象，右值就是函数调用的实参。
- (4) 每个类都有一个赋值操作函数。如果类中没有显式定义赋值操作函数，编译器就会自动生成一个公有的赋值操作函数，函数体中复制每个非静态数据成员，就像缺省拷贝构造函数一样。

赋值操作函数与拷贝构造函数功能相似，但极易混淆。赋值操作是将一个已有对象的数据复制给另一个同类对象，而拷贝构造函数则要创建一个新对象。

对前面 Date 类添加一个赋值操作函数，并封装为 date2.h。

```

    Date & operator=(const Date & d){           //赋值操作函数
        year = d.year;
        month = d.month;
        day = d.day;
        cout<<"operator= of Date:";
        print();
        cout<<endl;
        return *this;
    }

```

可以看出,上面复制操作函数与前面拷贝构造函数的函数体是一样的,前 3 行代码都一样,但这两者出于不同目的而设计,不能相互替代。

例 10-4 赋值操作函数的调用。

```

#include "date2.h"
Date getToday(){
    Date d;
    return d;
}
void test(){
    Date date1(2000,1,1);           //A
    Date date2;                     //B
    date2 = date1;                   //C
    date2 = getToday();              //D
    date2 = Date(2011,12,23);        //E
}
int main() {
    test();
    system("pause");
    return 0;
}

```

执行程序,输出如下(行号是为了方便说明而加的):

```

1 Constructor1 of Date:2000.1.1
2 Default Constructor of Date:2016.7.26
3 operator= of Date:2000.1.1
4 Default Constructor of Date:2016.7.26
5 operator= of Date:2016.7.26
6 Destructor of Date:2016.7.26
7 Constructor1 of Date:2011.12.23
8 operator= of Date:2011.12.23
9 Destructor of Date:2011.12.23
10 Destructor of Date:2011.12.23
11 Destructor of Date:2000.1.1

```

执行 A 行和 B 行分别输出第 1 行和第 2 行。

C 行是一条赋值语句,将调用赋值操作函数,输出第 3 行。

D 行先调用 `getToday` 函数,调用缺省构造函数创建对象 `d`,输出第 4 行,然后执行 `return` 语句,将局部对象 `d` 赋值给 `date2` 对象,调用了赋值操作函数,输出第 5 行。`getToday` 函数最后撤销局部变量 `d`,调用析构函数,输出第 6 行。

E 行先创建一个临时对象,输出第 7 行,再赋值 `date2`,调用赋值操作函数,输出第 8 行,最后撤销临时对象,调用析构函数,输出第 9 行。

至此 `test` 函数结束,先撤销 `date2`,再撤销 `date1`,分别输出第 10 行和第 11 行。

上面过程中未执行拷贝构造函数。

下面总结各种构造函数、析构函数、赋值操作函数的运行规律。

表 10-1 简单语句所执行的各种特殊函数

简单语句	被执行的特殊函数	说明
A a; A *pa=new A(); A a2[4]; return A(); //A f()	A() 无参构造函数	创建命名对象，说明一个变量 显式 new 创建单个对象； 创建数组 a2 要执行 4 次； 返回前创建对象，该对象可能会直接传给调用方新变量，也可能调用拷贝构造函数赋给已有变量，也可能被丢弃。取决于调用方。
A a(2); f(A(3)); //f(A) A a3[]={A(5),A(6),A(7)}; return A(4); //A f()	A(x) 有参构造函数	创建命名对象 创建对象并直接传给 f 形参； 创建数组 a3 要执行 3 次 返回前创建对象；
delete pa; 变量或形参离开其作用域}	~A() 析构函数	显式执行只有 delete; 隐式执行有多种形式； 临时对象被撤销
A b = a; A c(a) f(a); //f(A)	A(const A &a) 拷贝构造函数	右值 a 是已有对象， 左值 b 是当前新建对象，克隆 a； 以值传递，传递命名对象实参
a = b; a = f(); //A f()	A&operator=(const A&a) 赋值操作函数	左值右值都是已有对象； 左值是已有对象，右值是函数返回的对象

注1， 按 C11，调用函数时所创建并传递的实参对象不再使用临时对象。

注2， 按 C11，从一个函数中返回一个临时对象时，如果调用方是新建对象，那么临时对象就直接作为新建对象，而不再执行拷贝构造函数。

上表可以看出，C11 简化了临时对象的创建。假设一个对象 a 已存在，下面语句将出现临时对象：a = A(2); 先创建临时对象 t，t 赋值给 a，最后撤销 t。

对于 Date 类来说，缺省提供的拷贝构造函数和赋值操作函数复制数据成员，这是合理的，无需自行定义。那么对于一个类，何时需要自行定义拷贝构造函数和赋值操作函数？

一般来说，如果类中有指针数据成员，而且在构造函数执行中用 new 来动态申请内存，那么在对象撤销时就要用 delete 来回收内存。这类对象除了自身数据成员的值之外，还要关联一块动态内存来保存数据内容。对这样的对象如果调用了缺省提供的拷贝构造函数或者赋值操作函数，就会复制指针成员的值，会导致多个对象的指针成员指向同一块内存空间。当这些对象撤销时，析构函数分别执行就会导致同一块内存要回收多次，从而出现运行错误。要避免这种错误，就要显式定义拷贝构造函数和赋值操作函数，避免复制指针成员，一般要复制动态内容。

例 10-5 设计一个 Person 类，一个人作为一个对象，描述姓名和性别，而且姓名不限长，需自行定义析构函数、拷贝构造函数、赋值操作函数。

前面介绍了一个 **Person** 类，其中表示人的姓名使用了字符数组 “**char name[20]**”，这有两个问题。1、如果人的姓名超过 19 个字节就不能表示；2、大多数人名不超过 6 个字节(3 个中文字符)，但每个人名都将占用 20 个字节，造成内存浪费。一种合理的解决方法就是使用动态内存和指针成员。编程如下：

```
#include <iostream>
#include <string>
using namespace std;
class Person{
    char *name;                //指针成员
    char sex;
public:
    Person(char * name, char sex)    //构造函数
        :sex(sex), name(nullptr){
        setName(name);              //调用成员函数来设置姓名
    }
    ~Person(){                    //析构函数，回收动态内存
        if (name != nullptr)
            delete []name;
    }
    Person(const Person &p)
        :sex(sex), name(nullptr){
        setName(p.name);
    }
    Person & operator=(const Person &p){    //赋值操作函数
        setName(p.name);
        sex = p.sex;
        return *this;
    }
    void setName(const char *p){            //设置姓名
        if (name != nullptr)
            delete []name;                //如果原先有名字，先撤销原名
        if (p != nullptr){
            name = new char[strlen(p) + 1]; //根据新名大小申请一块空间
            strcpy_s(name, strlen(p) + 1, p); //复制新名
        }else
            name = nullptr;
    }
    const char * getName(){                //const 防止人名被随意更改
        if (name == nullptr)
            return "unnamed";
        return name;
    }
    char getSex(){return sex;}
    void show(){
        cout<< (sex=='f'?"她是":"他是" )<<getName()<<endl;
    }
};

int main(){
    Person a("张三",'m');
    a.show();
    Person b = a;                //A 调用拷贝构造函数
    b.setName("张三丰");        //B
    b.show();
    a = b;                        //C 调用赋值操作函数
}
```

```

    a.show();
    system("pause");
    return 0;
}

```

Person 类中使用了 char 指针来表示姓名，设计了一个 setName 成员函数来设置姓名，其中要根据名字大小动态申请内存来存放人名。这样一个 Person 对象就关联一块动态空间，当这个对象被撤销时，就应该在析构函数中撤销动态内存，否则就不能再利用内存空间。

此时自行定义拷贝构造函数和复制操作函数就必不可少，在 main 函数中就调用了这两个函数。如果你遗漏了任何一个，就会执行缺省提供的函数，而这些函数仅复制指针 name 的值，就会导致两个对象的 name 指针指向同一块动态内存。当这些对象被撤销时就会对同一块内存回收两次，从而导致 main 函数运行错误。自行定义这两个函数就是复制 name 指针所指向的内容。如下图所示。

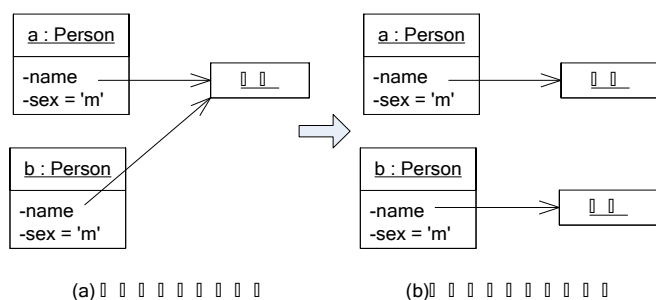


图 10-1 缺省函数与自定义函数的区别

类中的 getName 函数的返回值修饰为 const，限制程序不能通过返回的指针来随意改变人名，例如 main 函数中如果有：strcpy(a.getName(), "张三丰"); 就会导致编译错误。要改变人名应调用 setName 函数，如 B 行。

这个例子中使用了指针成员和动态内存，此时自定义拷贝构造函数和赋值操作函数就是必要的。不仅如此，后面会看到，还需要基于右值引用的移动语义。

### 10.3.3 用 string 替代 char\*

对于 Person 类的 name，典型的可变长字符串，应采用 C++ 标准模板库 STL 提供的 string 类型来表示，无需指针成员，也就避免了自行定义析构函数、拷贝构造函数和赋值操作函数。简化编程如下：

```

#include <iostream>
#include <string>
using namespace std;
class Person{
    string name;
    char sex;
public:
    Person(char * pname, char sex)
        :name(pname) {

```

//A 构造函数，对成员 name 初始化

```

        setSex(sex);
    }
    void setName(char *p){
        name = p;
    }
    const string & getName(){
        return name;
    }
    char getSex(){return sex;}
    void show(){
        cout<< (sex=='f'?"她是":"他是" )<<getName()<<endl;
    }
};

```

主函数 main 无需改变就能执行。

这个例子中 name 的类型为 string，name 就是 Person 类的一个成员对象，Person 就是一个复合对象。当创建一个 Person 对象时，就要自动创建其成员对象 name。这是通过构造函数中的成员初始化来实现的(A 行)。当撤销一个 Person 对象时将自动执行其成员对象的析构函数，而无需自行定义析构函数。

采用 string 的 Person 类不需要显式定义拷贝构造函数和赋值操作函数，这是因为缺省提供的拷贝构造函数和赋值操作函数能自动调用成员 string 的拷贝构造函数和赋值操作函数，而不会出现内存回收的问题，可有效简化编程，因此推荐采用 string 来表示任意长度的字符串，以取代传统的用字符指针或字符数组所表示的字符串。下面简单介绍 string 类型的用法。

首先，包含<string>头文件，再加上 using namespace std; 语句，就像上面例子一样。然后尝试用 string 取代所有 char\* 出现的地方，采用运算符来取代字符串处理函数。例如，用赋值运算符“=”来取代 strcpy 函数，用关系运算符“<、<=、>、>=、==、!= ”来取代 strcmp 函数，用加法运算符“+”或者“+=”来取代 strcat 函数。这些运算符的右操作数也可以是字符串常量。用成员函数 length 或 size 来取代传统的 strlen 函数。

C++编程是高级语言编程，应尽量避免因采用基本类型而导致编程复杂化，尽量采用系统提供的类型来简化编程。字符串 string 就是一种使用最普遍的类型。下表给出了 string 类型的常用操作函数，具体详情参见相关文档。

表 10-2 string 类型的常用操作函数

函数	功能
string()	缺省构造函数
string(const string&)	拷贝构造函数
string(const string&, int pos, int n)	构造函数，从 pos 开始取 n 个字符，取子串
string(const char *)	构造函数，从 char* 构造字符串对象
string(const char *, int n)	构造函数，取前 n 个字符
string(int n, char c)	构造函数，有 n 个 c 字符
operator=	赋值操作函数
int size() 或 length()	返回字符串的长度，相当于 strlen
char &at(int pos)	返回第 pos 个字符的引用，pos 合理范围

	[0...size-1], 越界引发 out_of_range 异常
const char * c_str()	返回指针指向串, 串不可变。
const char * data()	返回指针指向串, 串不可变。
int max_size() const	返回最长串的大小, 4294967293
void resize(int n, char c)	改变串的大小, 如果延长, 就用 c 填充
bool empty() const	是否为空
string& append(...)	字符串拼接
string& assign(...)	字符串赋值
string& insert(...)	插入子串
string& erase(...)	删除子串
string& replace(...)	子串替换
void swap(string & str)	当前串与 str 串交换
int find(...)	查找串, 返回首次出现的位置
operator+=	字符串拼接, 改变当前串
operator+	字符串拼接, 不改变当前串
operator==	字符串判断是否相等
operator!=	字符串判断是否不相等
operator<	小于
operator<=	小于等于
operator>	大于
operator>=	大于等于
operator<<	输出字符串
operator>>	输入字符串
getline(cin, string, char delim)	输入字符串, 直到 delim

对于一些类, 是否真的需要拷贝构造函数和赋值操作函数, 就要对其数据成员做具体分析。假如一个 **Person** 类包含了身份证号码作为数据成员, 假如一个 **Student** 类包含了学号, 假如一个账户 **Account** 类包含了账号, 等等。这些数据成员用来唯一标识一个对象, 而且不能随意改变。假如这些类执行了拷贝构造函数或者赋值操作函数, 就会导致两个或多个对象具有相同的标识, 就会破坏系统的一致性。对于这些类, 如何避免类外代码调用拷贝构造函数和赋值操作函数? 一种简单的办法是显式定义拷贝构造函数和赋值操作函数, 并将它们说明为私有的。这样类外的代码就不能再调用它们。如果要使类内部代码也避免执行它们, 可以在这两个函数中用 **throw** 语句引发异常来终止执行。

### 10.3.4 单参构造函数

单参构造函数是持有单个形参的构造函数, 形参类型不同于本类, 可实现隐式类型转换, 就是将其它类型数据转换为本类对象, 因此也称为转换构造函数 **conversion constructor**。

例 10-6 设计一个 **int** 包装类 **Integer**, 一个对象表示一个整数值。编程如下:



```

#include <iostream>
using namespace std;
class Integer{
    int value;
public:
    Integer(int i = 0){                //单参构造函数，同时也是缺省构造函数
        value = i;
        cout<<"Constructor of "<<value<<endl;
    }
    Integer(Integer & a){              //拷贝构造函数
        value = a.value;
        cout<<"copy constructor on "<<value<<endl;
    }
    Integer & operator=(const Integer & a){ //赋值操作函数
        value = a.value;
        cout<<"operator= "<<value<<endl;
        return *this;
    }
    int getValue(){return value;}
};
void fun1(Integer a){ cout<<a.getValue()<<endl; }
Integer fun2(){ return 40; }
int main(){
    Integer i1 = 10;                    //A
    Integer i2 = 20 + 10;                //B
    fun1(30);                           //C
    Integer i3 = fun2();                 //D
    i3 = 50;                             //E
    system("pause");
    return 0;
}

```

执行程序，输出如下：

```

Constructor of 10
Constructor of 30
Constructor of 30
30
Constructor of 40
copy constructor on 40
Constructor of 50
operator= 50

```

类 `Integer` 中定义了构造函数 `Integer(int i = 0)`，只有一个形参，而且带缺省值，是一个单参构造函数，同时也是缺省构造函数。

A 行和 B 行调用了单参构造函数，输出了前两行。赋值符号左边是一个对象，右边是单参构造函数的形参类型的一个对象或值，就会自动调用单参构造函数。A 行和 B 行等价于：

```

Integer i1(10);                        //A
Integer i2(20 + 10);                    //B

```

C 行调用 `fun1` 函数，该函数的形参是 `Integer` 类型，而实参为一个 `int` 值，此时就自动调用单参构造函数，创建了一个对象，输出了第 3 行。

D 行调用了 `fun2` 函数，该函数返回一个 `Integer` 对象，函数体中返回一个 `int` 值，此时自动调用单参构造函数，创建一个临时对象，再调用拷贝构造函数来初始化新建对象 `i3`。输出了两行 40。函数返回后，临时对象被撤销。

E 行是一条赋值语句，但包含了创建对象，等价于 `i3 = Integer(50)`；先调用单参

构造函数创建一个临时对象，再调用拷贝构造函数赋给对象 i3，然后临时对象被撤销。

一个单参构造函数就是将形参类型的一个对象或值转换为当前类的一个对象，相当一个类型转换，因此单参构造函数往往会被自动地隐式地调用。在说明语句、赋值语句、函数调用语句、返回语句中都可能自动调用。当需要一个 `Integer` 对象时，如果提供了一个 `int` 值，就会自动调用单参构造函数，先创建一个 `Integer` 对象，然后再做下一步处理。

有时这种隐式创建对象可能不是你所期望的，你可能只是犯了一个简单的输入错误。例如 C 行、E 行和函数 `fun2` 中的 `return 40`。如果要避免一个单参构造函数被隐式调用，可用关键字 `explicit` 来修饰构造函数：`explicit Integer(int i = 0){...}`，这样编译器就能检查隐式转换。

与单参构造函数的转换方向相反，有一种运算符重载函数称为转换函数，用来将本类的一个对象转换为指定类型的一个对象或一个值。在后面第 12 章运算符重载中介绍。

### 10.3.5 移动语义

前面第 8 章介绍了右值引用`&&`的基本概念和简单语法。右值引用主要支持所谓的移动 `move` 语义和完美转发 `forward`。下面介绍移动语义。完美转发涉及到模板，后面章节介绍。

#### 1. 移动语义

移动语义(move symantics)就是将对象的动态内存资源从一个对象传递给另一个对象，通常从临时对象传递给当前对象，提高对象转储效率。当该类对象要纳入某个 STL 容器(如 `vector`)时，这种效率提高是显著的。VS2010 开始对 STL 容器中诸多操作函数添加右值引用类型的重载形式。移动语义对于通用函数库类库的设计很重要。

典型例子是字符串 `string`。比如 `string s = string("h") + "e" + "ll" + "o"`；看似简单。实际上每个加号+都意味着要新建一个对象，并且做一次串接，就涉及内存移动，背后的代价很大。

另一个例子就是一种常用容器 `vector`，一个 `vector` 在任意时刻都有一个最大容量限制。当加入元素数量达到最大容量时，就要重新申请一块更大内存空间，再把已有元素转储到新空间中，最后释放已有元素所占空间。这种“内存搬家”需要大量的计算开销。

移动语义就是针对这些内存搬家的性能问题而提出的。

对一个类要实现移动语义，就要为其建立移动构造函数(类似拷贝构造函数)和移动赋值函数 `operator=(类似赋值操作函数)`。

移动构造函数的语法形式：

```
className(className&& obj){...} //形参为右值引用&&即为移动构造函数
```

其语义是从已有对象 `obj` 来构建当前对象，已有对象 `obj` 是临时对象，函数体中将自己的动态内存指针指向 `obj` 已有内存，并置空 `obj` 的内存指针，使 `obj` 对象回收时不影响当前对象。简言之，就是将临时对象 `obj` 的内存“移动”到当前对象。真实做法是仅移动指针，而不移动内存块。因此这个过程中不会出现 `new` 申请内存。

移动赋值操作函数的语法形式：

```
className & operator=(const className&& obj){...} //形参为右值引用&&即为移动赋值
```

其语义是当 `a=obj` 时, 把临时对象 `obj` 的内存“移动”到当前对象 `a` 中。真实做法是仅移动指针, 而不移动内存块。先释放自己的内存空间, 再指向 `obj` 的内存块, 最后置空 `obj` 的内存指针, 使 `obj` 析构时该内存块不会被系统回收, 不影响当前对象 `a`。同样这个过程中不会出现 `new` 申请内存。

移动语义就是移动对象的动态内存指针, 避免用 `new` 动态申请内存, 避免移动内存块数据。前提是被移动的是临时对象, 移动完成后就被自动回收。

移动构造函数可与拷贝构造函数以重载形式共存于同一个类, 同理, 移动赋值操作函数可与赋值操作函数重载共存。

编译器不会自动提供缺省的移动构造函数和移动赋值操作函数。

## 2. 移动实例分析

下面通过一个例子来说明移动构造函数与移动赋值函数。

假设有一个类 `MemoryBlock`, 其对象通过一个指针成员指向一块动态内存, 就像上面例子 `Person` 类中的 `char*name`, 就要为其设计析构函数、拷贝构造函数与赋值操作函数, 使其指针与所指内容在对象创建与操作时保持一致。但这两者需要频繁转储内存, 性能低。

此时基于右值引用`&&`的移动构造函数与移动赋值函数就能提高性能。

```
#include <iostream>
#include <vector>
using namespace std;
class MemoryBlock{
    size_t _length;    // 以 int 为单位的内存大小.
    int* _data;        // 指向动态内存的指针
public:
    explicit MemoryBlock(size_t length=10) //缺省构造函数
        : _length(length), _data(new int[length]){
        cout << "In MemoryBlock(size_t). length = " << _length << "." << endl;
    }
    ~MemoryBlock() {          //析构函数
        cout << "In ~MemoryBlock(). length = " << _length << ".";
        if (_data != nullptr){
            cout << " Deleting resource.";
            delete[] _data;
        }
        cout << endl;
    }
    MemoryBlock(const MemoryBlock& other) //拷贝构造函数
        : _length(other._length), _data(new int[other._length]) {
        cout << "In MemoryBlock(const MemoryBlock&). length = "
            << other._length << ". Copying resource." << endl;
        copy(other._data, other._data + _length, _data);    //调用 std::copy 转储内存
    }
    MemoryBlock& operator=(const MemoryBlock& other){ //赋值操作函数
        cout << "In operator=(const MemoryBlock&). length = "
            << other._length << ". Copying resource." << endl;
```

```

        if (this != &other) {
            delete[] _data; //释放自己的内存
            _length = other._length;
            _data = new int[_length]; //重新申请内存
            copy(other._data, other._data + _length, _data); //转储数据到新内存
        }
        return *this;
    }

    MemoryBlock(MemoryBlock&& other) // 移动构造函数 Move constructor.
    : _data(nullptr), _length(0){
        cout << "In MemoryBlock(MemoryBlock&&). length = "
            << other._length << ". Moving resource." << endl;
        _data = other._data; //复制指针
        _length = other._length;
        other._data = nullptr; //将 other 指针置空, 使 other 对象析构时不回收其内存
        other._length = 0;
    }

    MemoryBlock& operator=(MemoryBlock&& other) { // 移动赋值操作函数
        cout << "In operator=(MemoryBlock&&). length = " << other._length << "." << endl;
        if (this != &other) {
            delete[] _data; //释放自己的内存
            _data = other._data; //复制指针
            _length = other._length;
            other._data = nullptr; //将 other 指针置空, 使 other 析构时不回收其内存
            other._length = 0;
        }
        return *this;
    }

    size_t Length() const{
        return _length;
    }

    int* getDate() { return _data; }
};

void test1() {
    MemoryBlock m1(44); //A
    MemoryBlock m2(55); //B
    m1 = m2; //C
    cout << "m1.Length=" << m1.Length() << endl;
    m1 = MemoryBlock(88); //D call operator=(&&)
    cout << "m1.Length=" << m1.Length() << endl;
}

int main(){
    test1();
    system("pause");
    return 0;
}

```

右值引用的编译对于 DevC++, 要用编译选项-std=c++1y, 表示 C14 标准(兼容 C11)。

代码中使用了 std::copy 来转储内存, VS2015 编译时认为不安全, 提示添加编译选项-D\_SCL\_SECURE\_NO\_WARNINGS。

上面代码中除了构造函数, 析构函数, 拷贝构造函数, 赋值操作之后之外, 用右值引用

&&作为形参，定义了一个移动构造函数和一个移动赋值函数。

执行上面测试函数，输出如下(编号是为了说明而添加的):

```
1 In MemoryBlock(size_t). length = 44.
2 In MemoryBlock(size_t). length = 55.
3 In operator=(const MemoryBlock&). length = 55. Copying resource.
4 m1.Length=55
5 In MemoryBlock(size_t). length = 88.
6 In operator=(MemoryBlock&&). length = 88.
7 In ~MemoryBlock(). length = 0.
8 m1.Length=88
9 In ~MemoryBlock(). length = 55. Deleting resource.
10 In ~MemoryBlock(). length = 88. Deleting resource.
```

A 行 B 行分别创建两个对象，对应输出第 1 行和第 2 行。

C 行调用赋值操作函数，这是由于右值 m2 并非临时对象，其中出现了 new 申请内存和 copy 转储数据。输出第 3 行，第 4 行验证大小。

D 行赋值语句的右值是一个临时对象，先创建临时对象，输出第 5 行，然后调用移动赋值操作函数，输出第 6 行，其中没有出现 new 申请内存和 copy 转储数据。临时对象使用完成之后自动回收，析构函数执行，输出第 7 行，注意此时其内存大小为 0，这说明它已将自己的内存移动到 m1 对象。第 8 行验证 m1 的大小。

测试函数 test1 结束，其中局部变量 m2,m1 依次回收，输出第 9 行和第 10 行。

下面是另一个与容器 vector 有关的测试 (vector 详见第 13.4.4 节):

```
void test(){
    vector<MemoryBlock> v;
    v.push_back(MemoryBlock(25));
    v.insert(v.begin(), MemoryBlock(50));
}
```

先创建一个 vector 容器，然后加入一个对象 25，再把第 2 个对象 50 插入到第 1 个对象之前。

执行结果如下:

```
In MemoryBlock(size_t). length = 25.
In MemoryBlock(MemoryBlock&&). length = 25. Moving resource.
In ~MemoryBlock(). length = 0.
In MemoryBlock(size_t). length = 50.
In MemoryBlock(MemoryBlock&&). length = 25. Moving resource.
In ~MemoryBlock(). length = 0.
In MemoryBlock(MemoryBlock&&). length = 50. Moving resource.
In MemoryBlock(MemoryBlock&&). length = 25. Moving resource.
In operator=(MemoryBlock&&). length = 50.
In operator=(MemoryBlock&&). length = 25.
In ~MemoryBlock(). length = 0.
In ~MemoryBlock(). length = 0.
In ~MemoryBlock(). length = 50. Deleting resource.
In ~MemoryBlock(). length = 25. Deleting resource.
```

我们不用认真分析每一步的结果，只需统计移动函数被调用的次数。可以看出，移动构造函数被执行 4 次，移动赋值执行 2 次。除了 2 个对象之外没有再用 new 申请内存。

将代码中的移动构造函数和移动赋值函数注释掉，重新编译运行，再看结果。

```
In MemoryBlock(size_t). length = 25.
In MemoryBlock(const MemoryBlock&). length = 25. Copying resource.
In ~MemoryBlock(). length = 25. Deleting resource.
In MemoryBlock(size_t). length = 50.
```

```
In MemoryBlock(const MemoryBlock&). length = 25. Copying resource.  
In ~MemoryBlock(). length = 25. Deleting resource.  
In MemoryBlock(const MemoryBlock&). length = 50. Copying resource.  
In MemoryBlock(const MemoryBlock&). length = 25. Copying resource.  
In operator=(const MemoryBlock&). length = 50. Copying resource.  
In operator=(const MemoryBlock&). length = 25. Copying resource.  
In ~MemoryBlock(). length = 25. Deleting resource.  
In ~MemoryBlock(). length = 50. Deleting resource.  
In ~MemoryBlock(). length = 50. Deleting resource.  
In ~MemoryBlock(). length = 25. Deleting resource.
```

可以看出, 拷贝构造函数执行 4 次, 赋值操作函数执行 2 次, 共出现 6 次内存申请与释放。由此可知, 加入 2 个移动函数有效减少内存申请和内存复制的次数, 提高了计算效率。

上面例子是关于移动语义的, 右值引用还涉及到转发 forward 机制, 详见第 13 章。

## 10.4 复合对象与成员对象

一个类的数据成员可以是基本类型的值, 也可以是其它类的对象。如果一个类 A 的数据成员中有另一个类 B 的一个或多个对象, 那么类 A 对象就是复合(composite)对象, 类 B 的对象就是其成员对象。当创建一个复合对象时就要先创建其成员对象。当要撤销一个复合对象时, 也要撤销其成员对象。

### 10.4.1 构造过程

如果一个类中有成员对象, 该类的构造函数应说明对其各成员对象的初始化, 否则成员类就必需有缺省构造函数。带成员初始化的构造函数的格式如下:

<类名>::<类名>(<形参表>)

: <成员对象 1>(<实参表 1>), <成员对象 2>(<实参表 2>), ...

{ 函数体 }

其中, 在冒号之后就是成员初始化列表, 列出了一组成员对象, 并用逗号分隔多个成员。每个成员对象名字之后是相应的初始化实参。<实参表>中可以使用前面构造函数的<形参表>中的形参, 也可使用常量表达式。

在设计复合对象时要注意以下几点。

(1) 各成员对象的初始化应在该类的构造函数的成员初始化列表中显式列出。如果没有列出, 编译器就自动生成该成员类的缺省构造函数调用。此时如果成员类中没有缺省构造函数, 就指出编译错误。

(2) 当创建一个复合对象时, 要先创建其成员对象。在执行构造函数时, 要先执行所有成员对象的构造函数, 再执行复合对象的构造函数体。当类中有多个成员对象时, 其构造函数的执行顺序与成员对象在类中的说明顺序有关, 而与成员初始化表中的顺序无关。

(3) 析构函数的执行顺序与构造函数的执行顺序相反。先执行复合对象的析构函数, 再执行各成员对象的析构函数。

(4) 复合对象类缺省提供的拷贝构造函数和赋值操作函数将分别调用成员对象的拷贝构造函数和赋值操作函数。

例 10-7 以 Person 类为例，一个人不仅需要表示姓名、性别，还要表示出生日期，就要用到前面介绍的 Date 类，作为一个成员对象。Person 类的设计如图 10.2 所示。

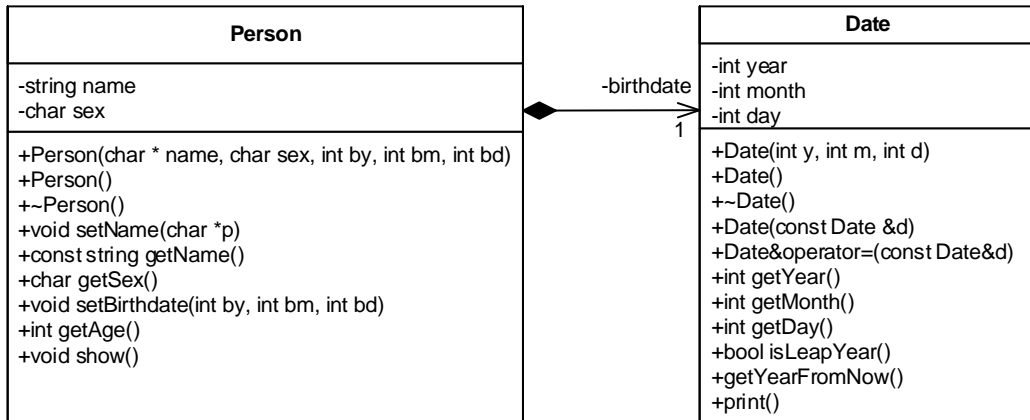


图 10-2 Person 类的设计

对于原先的 Date 类，添加一个成员函数 getAge()，更改为新文件 date3.h:

```

//date3.h
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <time.h>
using namespace std;
class Date{...//其它不变
    int getYearFromNow() {
        time_t ltime = time(NULL);           //返回与现在日期相差多少年
        tm * today = localtime(&ltime);       //取得当前时间
        int cyear = today->tm_year + 1900;      //转换为本地时间
        int cmonth = today->tm_mon + 1;          //取得当前年份
        int cday = today->tm_mday;
        if (cmonth > month ||
            cmonth == month && cday >= day)
            return cyear - year;
        else
            return cyear - year - 1;
    }
};
  
```

Date 类中的析构函数、拷贝构造函数和赋值操作函数都仅提示信息，没有实际作用。

Person 类的编程如下:

```

#include "date3.h"
#include <iostream>
#include <string>
using namespace std;

class Person{
    string name;
  
```

```

    char sex;
    Date birthdate;
public:
    Person(char * name, char sex, int by, int bm, int bd) //构造函数
        :name(name), sex(sex), birthdate(by, bm, bd){ //对成员对象初始化
        cout<<"Constructor of Person"<<endl;
    }
    Person() :name("无名氏") , sex('f') {} //缺省构造函数
    ~Person(){ //析构函数, 仅提示信息
        cout<<"Destructor of Person"<<endl;
    }
    void setName(const char *p){
        name = p;
    }
    const string & getName(){
        return name;
    }
    char getSex(){return sex;}
    void setBirthdate(int by, int bm, int bd){
        birthdate = Date(by, bm, bd);
    }
    int getAge(){
        return birthdate.getYearFromNow(); //返回当前年龄
    }
    void show(){
        cout<< (sex=='f'?"她是":"他是" )<<getName();
        cout<<"出生日期为";
        birthdate.print();
        cout<<"年龄为"<<getAge()<<endl;
    }
};
void test(){
    Person a("张三",'m', 1982,4,23); //A
    a.show();
    Person b = a; //B
    b.setName("张小三");
    b.show();
    a = b; //C
    a.show();
    Person c; //D
    c.show();
    c.setName("李四");
    c.setBirthdate(1990,2,12); //E
    c.show();
}
int main() {
    test();
    system("pause");
    return 0;
}

```

执行程序, 输出如下(行号是为了方便解释而加的):

```

1 Constructor1 of Date:1982.4.23
2 Constructor of Person
3 他是张三; 出生日期为 1982.4.23; 年龄为 34
4 Copy constructor of Date:1982.4.23
5 他是张小三; 出生日期为 1982.4.23; 年龄为 34

```



```
6 operator= of Date:1982.4.23
7 他是张小三;出生日期为 1982.4.23;年龄为 34
8 Default Constructor of Date:2016.8.6
9 她是无名氏;出生日期为 2016.8.6;年龄为 0
10 Constructor1 of Date:1990.2.12
11 operator= of Date:1990.2.12
12 Destructor of Date:1990.2.12
13 她是李四;出生日期为 1990.2.12;年龄为 26
14 Destructor of Person
15 Destructor of Date:1990.2.12
16 Destructor of Person
17 Destructor of Date:1982.4.23
18 Destructor of Person
19 Destructor of Date:1982.4.23
```

A 行创建一个 Person 对象, 先调用成员 Date 构造函数, 再调用 Person 构造函数, 输出前 2 行。

B 行将调用 Person 类缺省提供的拷贝构造函数, 其中自动调用了 Date 类的拷贝构造函数, 输出第 4 行。

C 行是一条对象赋值语句, 将调用 Person 类缺省提供的赋值操作函数, 其中自动调用了 Date 类的赋值操作函数, 输出第 6 行。

D 行调用 Person 类的缺省构造函数来创建对象, 只对 name 初始化“无名氏”, 没有对 birthdate 进行初始化, 此时就自动调用 Date 类的缺省构造函数, 用当前日期进行初始化。下一行输出能看到结果。

E 行设置出生日期, 函数中执行“birthdate = Date(by, bm, bd);”语句, 先创建一个临时匿名对象, 输出第 10 行, 然后调用 Date 类的赋值操作函数将 3 个成员赋给 birthdate 对象, 输出第 11 行, 然后撤销这个临时对象, 输出第 12 行。

主函数 main 执行结束, 对其中的局部对象 a、b、c 按构造次序的反序依次撤销。对每个对象先执行复合对象 Person 的析构函数, 再执行成员对象 Date 的析构函数, 与构造过程正相反, 输出最后 6 行。

#### 10.4.2 复合对象设计实例

前面第 9 章介绍了 Point 类表示二维平面上的点。如何设计二维平面上的圆?

例 10-8 设计一个 **Circle** 类，表示二维平面上的圆，一个圆作为 **Circle** 类的一个对象。一个圆是由一个圆心(一个点)和一个半径组成的。显然 **Circle** 类需要两个数据成员，一个是圆心点，用来确定圆的位置，另一个是半径，确定圆的大小。**Point** 类表示二维平面上的点，可以作为 **Circle** 类的数据成员来表示圆心。

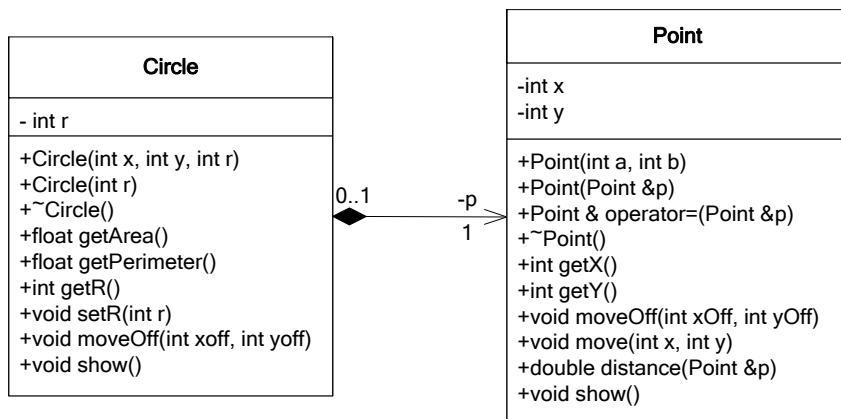


图 10-3 Circle 类的设计

**Circle** 类的设计如图 10.3 所示。设计要点如下：

- ◆ **Circle** 类中需要 **Point** 类的一个对象 **p** 作为圆心，表示为一个复合结构关系。
- ◆ 设计两个构造函数和析构函数。没有显式定义拷贝构造函数和赋值操作函数。
- ◆ 对于一个圆的操作包括：改变大小，即改变半径(**setR** 函数)；相对移动(**moveOff** 函数)；计算面积(**getArea** 函数)；计算周长(**getPerimeter** 函数)等。

**Point** 类的文件命名为 **Point2.cpp**，包含此文件就能使用 **Point** 类。该类中的析构函数、拷贝构造函数和赋值操作函数都只是提示信息。编程如下：

```

//Point2.cpp
#include <iostream>
#include <math.h>
using namespace std;

class Point{
    int x, y;
public:
    Point(int a = 0, int b = 0){           //缺省构造函数
        x = a; y = b;
        cout<<"Constructor! x="<<x<<"<<y<<endl;
    }
    Point(Point &p){                       //拷贝构造函数
        x = p.x; y = p.y;
        cout<<"Copy Constructor! x="<<x<<"<<y<<endl;
    }
    Point & operator=(Point &p){           //赋值操作函数
        x = p.x; y = p.y;
        cout<<"operator= x="<<x<<"<<y<<endl;
        return *this;
    }
    ~Point(){                             //析构函数

```

```

        cout<<"Destructor! x="<<x<<"y="<<y<<endl;
    }
    int getX() {return x;}
    int getY() {return y;}
    void moveOff(int xoff, int yoff){ x += xoff; y += yoff;}
    void move(int x, int y){this->x = x; this->y = y;}
    void show(){
        cout<<"("<<getX()<<" , "<<getY()<<")"<<endl;
    }
    double distance(Point &p){
        double xdiff, ydiff;
        xdiff = x - p.x;
        ydiff = y - p.y;
        return sqrt(xdiff*xdiff + ydiff*ydiff);
    }
};

```

Circle 类编程如下:

```

#include <iostream>
#include "Point2.cpp"
class Circle{
    Point p;
    int r;
public:
    Circle(int x, int y, int r):p(x, y){
        this->r = r;
        cout<<"Constructor 1 of Circle"<<endl;
    }
    Circle(int r){
        this->r = r;
        cout<<"Constructor 2 of Circle"<<endl;
    }
    ~Circle(){
        cout<<"Destructor of Circle"<<endl;
    }
    float getArea(){
        return float(r * r * 3.14159);
    }
    float getPerimeter(){
        return float(2 * r * 3.14159);
    }
    int getR(){return r;}
    void setR(int r){this->r = r;}
    void moveOff(int xoff, int yoff){
        p.moveOff(xoff, yoff);
    }
    void show(){
        p.show();
        cout<<"r = "<<r<<" area="<<getArea();
        cout<<" perimeter="<<getPerimeter()<<endl;
    }
};

void test() {
    Circle c1(1, 2, 3);           //A
    c1.show();                    //B
    Circle *c2 = new Circle(4);   //C
    c2->show();                    //D
    c2->setR(2);
    c2->moveOff(2, 1);
    c2->show();                    //E
}

```

```
        delete c2;                                //F
    }
    int main(){
        test();
        system("pause");
        return 0;
    }
```

执行程序，输出如下(序号是为了方便解释而加的):

```
1 Constructor! x=1;y=2
2 Constructor 1 of Circle
3 (1, 2);r = 3; area=28.2743; perimeter=18.8495
4 Constructor! x=0;y=0
5 Constructor 2 of Circle
6 (0, 0);r = 4; area=50.2654; perimeter=25.1327
7 (2, 1);r = 2; area=12.5664; perimeter=12.5664
8 Destructor of Circle
9 Destructor! x=2;y=1
10 Destructor of Circle
11 Destructor! x=1;y=2
```

A 行创建 Circle 对象 c1，调用了构造函数 1。先执行成员初始化 p(x,y)，执行 Point 的构造函数，输出第 1 行，然后再执行 Circle 类的构造函数 1，输出第 2 行。

B 行输出 c1 的状态，输出第 3 行。

C 行创建 Circle 对象 c2，调用了构造函数 2。这个构造函数仅确定半径，而圆心缺省为原点(0,0)。构造函数 2 中没有显式列出成员初始化，实际上是调用了 Point 缺省构造函数，输出第 4 行。然后再执行构造函数 2，输出第 5 行。

D 行输出 c2 的状态，输出第 6 行。

下面对 c2 对象改变半径并移动，E 行再输出 c2 的状态，输出第 7 行。

F 行撤销了 c2 对象，此时要先执行 Circle 类的析构函数，输出第 8 行，然后再执行 Point 类的析构函数，输出第 9 行。

当 main 函数结束时，要撤销局部对象 c1，也是先执行 Circle 类的析构函数，输出第 10 行，然后再执行 Point 类的析构函数，输出第 11 行。

### 10.4.3 复合对象设计要点

复合对象是普遍存在的。一个复杂对象往往可分解为若干成员。例如，一个窗口 Window 就是一个复合对象。一个窗口包含了一个标题、一组菜单、一组工具栏、一个或多个显示区、垂直滚动棒和水平滚动棒等作为其成员对象。

一个复合对象包含若干成员对象，每个成员对象都是其复合对象的组成部分。当创建一个复合对象时，就要自动创建其所有的成员对象。在一个复合对象的生存期间，不能撤销其成员对象，但可改变成员对象的状态。当一个复合对象被撤销时，其成员对象也要被自动撤销。

复合对象与成员对象之间的关系可以嵌套构成，例如描述圆柱体的类 Cylinder。一个圆柱体对象应包含一个圆作为其成员，再加上一个高度。此时圆柱体就是一个复合对象，而圆 Circle 就作为其成员对象。这种嵌套的复合关系使我们能逐步描述更加复杂的对象。

一个复合对象类知道自己需要哪些成员对象，而成员对象类自己并不知道要作为哪些类

的成员。例如, **Point** 类并不知道自己要作为圆的圆心, 也可能作为矩形的顶点、三角形的顶点、或多边形的顶点等。

复合对象类的设计关键是对象结构的分析和设计。考虑如何建立一个 **Rectangle** 类表示矩形。

首先, 考虑二维平面上的一个矩形如何构成。第一种方案, 一个左上角顶点和一个右下角顶点就能构成一个矩形, 需要设计两个成员 “**Point topLeft; Point bottomRight;**”。另一种构成方案是一个左上角顶点 “**Point topLeft;**”, 再加上一个高度 **high** 和一个宽度 **width**。当一个矩形对象被撤销时, 其顶点成员对象也应该自动撤销。

其次, 考虑可能的构造过程。对于第 1 种方案, 直接提供两个点对象, 就可以设计一个构造函数, 也可以提供两对坐标, 作为另一个构造函数。对于第 2 种方案, 直接提供一个顶点, 再加上一个长度和一个宽度, 可以设计一个构造函数。也可以提供顶点的坐标, 再加上一个长度和一个宽度, 作为另一个构造函数。

然后, 考虑对于一个矩形对象有哪些操作。例如, 可以改变高度、宽度, 可以移动顶点来移动矩形, 还可以计算周长和面积等, 可以设计一组公共成员函数来分别实现。

复合对象与其各成员对象的生存期具有特殊规律。当一个复合对象被撤销, 它的成员对象也要被撤销, 即成员对象的生存期不会比其复合对象更长。

利用这个规律能判断一个复合结构设计的合理性。考虑如何建立一个 **Family** 类表示家庭。除了描述家庭的常住地址, 还要描述一个家庭中有一人(一个 **Person** 对象)作为户主, 此时你可能要为 **Family** 类设计一个成员对象 “**Person owner**”。还要描述一个家庭中的多个成员, 就可能再设计一个 “**Person members[MAX]**”。如此设计将 **Person** 作为 **Family** 的成员, 看似合理, 也能完成很多功能。但分析复合对象的生存期就会发现问题。当一个家庭对象被撤销时, 其中户主和家庭成员的信息是否都要被撤销? 如果从实际户籍管理角度来看, 这就是错误的设计。因为一个人的生存期往往比其所在家庭的生存期更长。撤销一个家庭并不能自动撤销其家庭成员的信息, 因此不能简单使用复合对象关系。

在一个复合结构关系中, 作为成员对象往往不知道它作为哪些类的成员。但如果成员对象也要知道其复合对象的话, 那么这个复合结构关系设计就值得怀疑了。例如, 一个家庭作为一个复合对象, 知道它有哪些人作为其成员。但同时家庭成员自己应该能知道其所在的家庭信息, 如家庭住址、户主等。从这个角度来分析, **Family** 类与 **Person** 类之间就不能建立复合关系。

当我们说 “包含” 或 “由...组成”, 不能简单地一概作为复合对象设计。类似的例子还有人事管理系统中的 “部门” 与 “员工”、学籍管理系统中 “班级” 与 “学生” 等、城市交通中的 “公交线路” 与 “公交站点”。此类设计需要建立对象之间比较松散的关联, 往往要把指针作为类的数据成员, 将在后面介绍。

## 10.5 对象数组

我们常常要将同一类的多个对象作为一个集合来进行操作。一个对象数组就是一类对象的一个有序集合, 数组中所有元素是同一类的对象。对象数组的定义、赋值和使用与普通数组一样。

### 10.5.1 定义和使用

一维对象数组的定义格式为：

<类名> <数组名>[<常量表达式>] [= {初始化列表}];

例如：Point points[3] = {Point(1, 2), Point(3, 4)};

定义了 Point 类的一个数组 points，有 3 个元素，类名加实参，对各元素进行初始化。如果没有带初始化列表，就调用 Point 类的缺省构造函数来创建 3 个对象。上例中对前 2 个元素显式给出了初始化，这时系统根据实参个数和类型去调用相应的构造函数来创建对象并作为数组元素。如果没有显式初始化，就调用缺省构造函数来进行初始化。

对数组元素对象的赋值与一般对象的赋值一样，有两种方式：一种是用一个已存在的同类对象进行赋值，例如，dates[2] = dates[0]，这需要调用赋值操作函数。另一种是创建一个临时对象来赋值。例如，points[2] = Point(5, 6)。先根据 Point(5,6)调用构造函数来创建一个临时对象，再调用赋值操作函数把临时对象赋给 points[2]，最后调用析构函数来撤消临时对象。显然这种赋值代价颇大。

访问数组元素的成员的一般格式为：

<数组名>[<下标表达式>].<成员名>

例如：points[i].move(3, 4);

例 10-9 对象数组的定义和使用。

```
#include <iostream>
#include "Point2.cpp"
using namespace std;
void test(){
    Point points[3] = {Point(1,2), Point(3,4)};           //A
    points[2] = Point(5,6);                               //B
    for(int i=0; i<3; i++){                                //C
        points[i].show();
        cout<<endl;
    }
}
int main() {
    test();
    system("pause");
    return 0;
}
```

执行程序，输出如下(序号是为了方便解释而加的)：

```
1.Constructor! x=1;y=2
2.Constructor! x=3;y=4
3.Constructor! x=0;y=0
4.Constructor! x=5;y=6
5.operator= x=5;y=6
6.Destructor! x=5;y=6
7.(1, 2)
8.(3, 4)
9.(5, 6)
10.Destructor! x=5;y=6
11.Destructor! x=3;y=4
12.Destructor! x=1;y=2
```

A 行定义 Point 类的一个对象数组 points。在对数组元素对象进行初始化时，显式给定了

前 2 个元素对象的初始化值，这时根据给定的实参个数和类型去调用相应的构造函数对这 2 个元素进行初始化，程序输出第 1 行和第 2 行。最后一个元素没有指定初始化值，系统调用缺省构造函数对该数组元素对象进行初始化，输出第 3 行。

B 行先根据 `Point(5,6)` 创建一个临时对象，调用构造函数输出第 4 行；临时对象再赋值给数组元素，调用赋值操作函数，输出第 5 行，然后撤消该临时对象时，调用析构函数输出第 6 行。

C 行循环语句执行依次输出第 7 至第 9 行。`main` 函数结束时，撤消数组 `points` 中的各个对象，调用析构函数依次输出第 10 至 12 行。析构函数的执行顺序与构造函数的执行顺序相反。

### 10.5.2 对象数组作为成员

上面例子中定义并使用了对象数组。对象数组经常作为一个类的数据成员。

在 `Point` 类的基础上，考虑如何设计一个三角形的类 `Triangle`。一个三角形作为一个对象，由 3 个点组成，即 3 个 `Point` 对象。即便这 3 个点碰巧在一条直线上，也可认为面积为 0 的一种特例。这样就能确定 `Triangle` 类的数据成员可以是 3 个任意位置的 `Point` 对象。那么使用 3 个 `Point` 变量还是 1 个数组？我们尝试用 1 个数组来实现。这种选择受到多边形结构的影响，因为一个多边形是由 3 个以上的有序顶点构成，用数组比较方便。三角形可看作 3 个有序顶点构成。

其次，考虑可能的构造过程。直接提供 3 个 `Point` 对象就是一种构造函数方案，也可以提供 3 对坐标作为另一个构造函数。

然后，再考虑一个三角形作为一个对象有哪些操作。你可以移动任何一个点，也可以移动整个三角形，可以计算其周长和面积等。

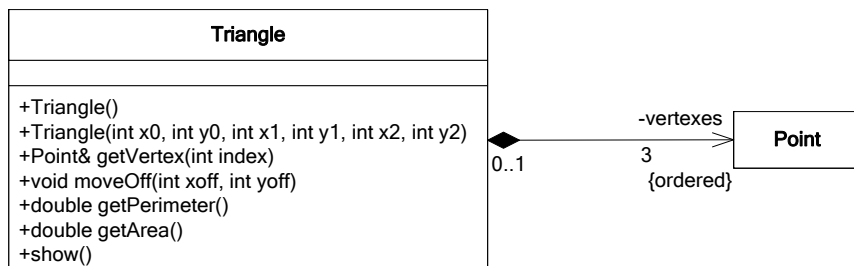


图 10-4 `Triangle` 类的结构设计

例 10-10 三角形 `Triangle` 类，对象数组作为数据成员的例子。类 `Triangle` 的设计如图 10.4 所示。三角形类 `Triangle` 包含了 3 个 `Point` 对象作为其顶点，表示为一个复合关系。编程如下：

```

#include <iostream>
#include "Point2.cpp"
using namespace std;
class Triangle{
    Point vertexes[3];
public:
    Triangle() {}

```

// 缺省构造函数

```

Triangle(int x0, int y0, int x1, int y1, int x2, int y2)
    :vertexes{ Point(x0,y0),Point(x1,y1),Point(x2,y2) }    //A
{}
Point &getVertex(int index){          //取得顶点的引用
    return vertexes[index];
}
void moveOff(int xoff, int yoff){    //相对移动
    vertexes[0].moveOff(xoff, yoff);
    vertexes[1].moveOff(xoff, yoff);
    vertexes[2].moveOff(xoff, yoff);
}
double getPerimeter(){               //计算周长
    double per = 0;
    per += vertexes[0].distance(vertexes[1]);
    per += vertexes[1].distance(vertexes[2]);
    per += vertexes[2].distance(vertexes[0]);
    return per;
}
double getArea(){                   //计算面积
    double a = vertexes[0].distance(vertexes[1]);
    double b = vertexes[1].distance(vertexes[2]);
    double c = vertexes[2].distance(vertexes[0]);
    double s = (a + b + c) / 2;
    return sqrt(s*(s-a)*(s-b)*(s-c));
}
void show(){
    vertexes[0].show();
    vertexes[1].show();
    vertexes[2].show();
    cout<<endl;
    cout<<"周长为"<<getPerimeter();
    cout<<"面积为"<<getArea()<<endl;
}
};
void test() {
    Triangle t1;
    t1.show();
    Triangle t2(1, 2, 3, 4, 4, 1);
    t2.show();
    t2.getVertex(0).move(2, 3);
    t2.show();
}
int main(){
    test();
    system("pause");
    return 0;
}

```

执行程序，输出如下：

```

Constructor! x=0;y=0
Constructor! x=0;y=0
Constructor! x=0;y=0
(0, 0)(0, 0)(0, 0)
周长为 0;面积为 0
Constructor! x=1;y=2
Constructor! x=3;y=4
Constructor! x=4;y=1
(1, 2)(3, 4)(4, 1)

```



```

周长为 9.15298;面积为 4
(2, 3) (3, 4) (4, 1)
周长为 7.40492;面积为 2
Destructor! x=4;y=1
Destructor! x=3;y=4
Destructor! x=2;y=3
Destructor! x=0;y=0
Destructor! x=0;y=0
Destructor! x=0;y=0

```

第 1 个构造函数是缺省构造函数，隐含要调用 `Point` 类的缺省构造函数。

第 2 个构造函数由 3 个点坐标来创建 3 个 `Point` 对象，其中 A 行是成员初始化，函数体为空。

## 10.6 静态成员

类中除了非静态成员之外，还可定义静态成员，就是用 `static` 修饰的成员。一般所说的成员都是指非静态成员，但静态成员仍具有不可替代的作用。静态成员包括静态数据成员和静态成员函数。

### 10.6.1 静态数据成员

类中用关键字 `static` 修饰的数据成员称为静态数据成员。一个静态数据成员表示类的多个对象所共享的一个成员。无论是否创建对象，类的静态成员随类而存在。本质上，静态数据成员并不是对象的成员，而是类的成员。

类中的静态数据成员与结构中的静态成员的语法语义一样。

例 10-11。以 `Point` 类为例，要求在缺省构造函数中用一对缺省坐标来代替原先的缺省坐标(0,0)，要求缺省坐标可以在运行时刻改变，还要求在任意时刻能知道 `Point` 对象的数目。

`Point` 类的设计如图 10.5 所示。图中用下划线标出静态成员。`Point` 类中说明了 3 个私有的静态数据成员，其中 `defx` 和 `defy` 表示缺省坐标。当执行缺省构造函数时，就用这个坐标作为新建点的坐标。静态数据 `pcount` 是一个计数值，表示当前对象数量，构造函数每执行一次就加 1，析构函数每执行一次就减 1。另外还设计了 2 个静态成员函数来管理这些静态数据。编程如下：

```

#include <iostream>
#include <math.h>
using namespace std;
class Point{
    int x, y;
    static int defx, defy;    //缺省坐标
    static int pcount;       //对象计数
public:

```

Point
<code>-int x</code> <code>-int y</code> <u><code>-int defx</code></u> <u><code>-int defy</code></u> <u><code>-int pcount</code></u>
<code>+Point()</code> <code>+Point(int a, int b)</code> <code>+Point(Point &amp;p)</code> <code>+~Point()</code> <u><code>+void setDef(int x, int y)</code></u> <u><code>+int getCount()</code></u> <code>+int getX()</code> <code>+int getY()</code> <code>+void move(int x, int y)</code> <code>+void moveOff(int xOff, int yOff)</code> <code>+double distance(Point &amp;p)</code> <code>+void show()</code>

图 10-5 `Point` 类中的静态成员

```

Point() {                                //缺省构造函数
    x = defx; y = defy;                  //用静态数据成员来初始化
    pcount++;                            //对象计数加 1
}
Point(int a, int b){
    x = a; y = b;
    pcount++;                            //对象计数加 1
}
Point(Point& p){                          //拷贝构造函数
    x= p.x; y = p.y;
    pcount++;
}
~Point(){                                //析构函数
    pcount--;
}
static void setDef(int x, int y){//改变缺省坐标
    defx = x; defy = y;
}
static int getCount(){return pcount;}
int getX() {return x;}
int getY() {return y;}
void move(int x,int y){this->x = x; this->y = y;}
void moveOff(int xoff, int yoff){this->x += xoff; this->y += yoff;}
double distance(Point &p){
    double xdifff, ydifff;
    xdifff = x - p.x;
    ydifff = y - p.y;
    return sqrt(xdifff*xdifff + ydifff*ydifff);
}
void show(){
    cout<<"("<<getX()<<"", "<<getY()<<"");
}
};
int Point::defx;                          //缺省初始化为 0
int Point::defy = 2;
int Point::pcount = 0;
int main(){
    cout << Point::getCount() << " points existing." << endl;
    Point p1;
    p1.show();cout<<endl;
    Point::setDef(11, 22);
    Point *p2 = new Point();
    cout<<Point::getCount()<<" points existing."<<endl;
    p2->show();cout<<endl;
    delete p2;
    cout<<Point::getCount()<<" points existing."<<endl;
    Point p3(33,44);
    p3.show();cout<<endl;
    cout<<Point::getCount()<<" points existing."<<endl;
    system("pause");
    return 0;
}

```

执行程序，输出如下：

```

0 points existing.
(0, 2)
2 points existing.

```

```
(11, 22)
1 points existing.
(33, 44)
2 points existing.
```

对静态数据成员，注意以下几点。

(1)类的静态数据成员在加载类时为其分配存储空间，区别于非静态数据成员。在创建类的对象时，只为类中的非静态数据成员分配存储空间。用 `sizeof(类名)` 不包括静态成员。

(2)静态数据成员的初始化应该在加载类时进行，因此静态数据成员必须在类的外部，即文件作用域中，做定义性说明，并进行初始化。如果没有显式初始化，缺省初始化为 0。注意在类外定义时，不用加修饰词 `static` 和访问控制修饰词。

(3)静态数据成员与类共存，因此在类外应通过作用域运算符(类名::静态数据成员)来访问。虽然允许通过对象来访问静态数据成员，但仍然是类的成员，而非对象的成员。因此应避免用对象来访问静态成员。在构造函数和成员函数中可直接用名字来访问类中的静态数据成员，而无需加类名，因为它们同属于同一个类的作用域。

(4)静态数据成员不同于全局变量。虽然它们都是静态分配存储空间，全局变量没有封装性，在程序任何位置都可访问，而静态数据成员受到封装性和访问权限的约束。管理私有的静态数据成员一般需要静态成员函数。

### 10.6.2 静态成员函数

在类中用 `static` 修饰的成员函数就是静态成员函数。静态成员函数用于管理类中的静态数据成员，或者提供类的某种服务，也包括创建该类的对象和撤销对象。

上面 `Point` 类中，为了访问私有静态数据成员 `defx` 和 `defy`，设计了一个公有静态成员函数 `setDef`，使类的使用者能改变缺省坐标。类中的私有静态数据成员 `pcount`，对于类外应该为只读而不能修改，故此只有一个 `getCount` 静态函数。

对于静态成员函数，注意以下几点。

(1)在类的外部调用一个静态成员函数时，应通过类名加上作用域操作符。虽然也允许通过对象来调用静态成员函数，但本质上仍然是类的操作，而不是对象的操作。因此应避免用对象来访问静态成员。

(2)静态成员函数中没有隐含的 `this` 指针，即没有当前对象，因此静态函数中不能直接访问该类的非静态成员，只能访问静态成员。静态成员函数如果要访问该类的非静态成员，必须将类的对象、或对象引用、或对象指针作为函数形参。例如，`distance(Point &p)` 函数是一个非静态成员函数，在调用时必须作用于一个当前对象，例如 `"p1.distance(p2)"` 来计算 `p1` 与 `p2` 之间的距离。但如果你要用另一种方式来计算两个点之间距离：`"Point::distance(p1, p2)"`，可添加一个静态成员函数如下：

```
static double distance(Point &p1, Point &p2){
    return p1.distance(p2);           //静态函数中调用了非静态成员函数
}
```

(3)静态成员函数可以被派生类继承，但不能定义为虚函数。虚函数将在下一章介绍。

(4)如果静态成员函数的实现部分在类外定义，不能加修饰符 `static` 和访问控制修饰符。

## 10.7 const 和 volatile 修饰符

用关键词 `const` 和 `volatile` 修饰类的成员函数和对象，就限制了成员函数和对象的行为。`const` 表示不变性，而 `volatile` 表示可变性，两者统称为 `cv` 限定符。在编译时编译器将检查不变性和可变性的相关规则。

### 10.7.1 修饰符 `const`

前面我们用 `const` 来定义命名常量，还用 `const` 来限制一个指针不能改变或指针所指内容不能改变。下面介绍 `const` 修饰类的成员函数、函数的形参及返回值。

#### 1.const 成员函数

用 `const` 修饰一个成员函数就限制了该函数中不能改变该类当前对象的数据成员。在形参表之后、函数体之前添加 `const`：

<返回类型> <函数名>(<形参表>) `const` { ... }

`const` 成员函数体中不能改变当前对象的数据成员，而且只能调用 `const` 成员函数。

如果一个成员函数确实不应改变对象状态，就应添加 `const` 修饰，以明确告知调用方。例如：

```
double distance(Point &p) const{...}
```

你会发现很多成员函数都应该修饰为 `const`，例如所有的 `getXxx` 函数。

`const` 修饰符作为成员函数基调的一部分，可支持重载定义成员函数。例如：

```
class A{
public:
    void f1(){cout<<"f1()"<<endl;}//第 1 个 f1 函数
    void f1() const{cout<<"f1() const"<<endl;}//重载 f1() 函数
    void f2() const{f1();}          //执行的是第二个 f1 函数
};
void f1(A &a){a.f1();}              //执行的是第一个 f1() 成员函数
void f2(const A &a){a.f1();}        //执行的是第二个 f1() 成员函数
void main(){
    A a;
    f1(a);          //输出 f1()
    f2(a);          //输出 f1() const
    a.f2();         //输出 f1() const
}
```

如果在类外实现 `const` 成员函数，应保持 `const` 修饰符，否则编译器会误认为是在定义重载函数。

#### 2.const 对象

用 `const` 修饰的对象是不可变的，称为常量对象。`const` 添加在类名之前：

`const` <类名> 对象名、或对象引用；

`const` 对象说明可能出现在类的外面，作为对象类型的一部分。`const` 对象不能赋值给非 `const` 对象，作为函数实参调用也不行。

`const` 对象说明也可出现在类的数据成员中，构造函数中的成员初始化列表中必须显式给出该成员如何初始化，而且该成员不能被其它成员函数改变。

### 3.const 形参

函数形参表中, `const` 形参就是常量, 函数体中不能改变其数据成员的值。对于一个常量对象, 只能调用其 `const` 成员函数, 而不能调用其它非 `const` 成员函数。

如果一个函数中确实不会改变某个形参, 那么就应该用 `const` 来修饰, 以明确告知调用方。例如:

```
double distance(const Point &p);
```

注意, 形参是否带 `const` 修饰也作为函数基调的一部分, 可用来定义重载函数。例如:

```
void f1(A &a);
```

```
void f1(const A &a);    //重载 f1 函数, 根据实参是否为 const 来区别
```

### 4.const 返回值

用 `const` 可说明一个函数的返回值不可改变。当返回一个对象的指针或引用时, 如果不希望调用方通过返回的指针或引用来改变对象的状态, 就应该用 `const` 来修饰返回值。

修饰符 `const` 有两方面的好处。一方面对于类的设计人员, 当在函数体中发生误操作而改变成员数据或 `const` 形参时, 编译系统将报错。将可变性限制到最小范围, 可简化编程。另一方面是对于类的使用方, 可以信任函数体的实现, 不会改变成员数据或 `const` 形参。`const` 修饰是行为约定的重要组成部分, 自觉使用 `const` 修饰符是程序员成熟的重要标志。

## 10.7.2 修饰符 `volatile`\*

修饰符 `volatile` 可修饰成员函数、函数形参及返回值。修饰符 `volatile` 假设当前对象是“易变的”, 随时可能被程序外部其它东西所改变, 如操作系统、硬件、并发线程等。对 `volatile` 修饰的对象或值的访问在编译时不做优化。

用 `volatile` 修饰一个成员函数时, 其格式为:

```
<返回类型> <函数名>(<形参表>) volatile{ ... }
```

注意, 关键字 `volatile` 属于成员函数说明的组成部分, 因此如果在类外实现一个 `volatile` 成员函数时, 也要添加 `volatile` 修饰, 否则编译器会误认为是在定义一个重载函数。

用 `volatile` 修饰的对象称为易变对象。说明 `volatile` 对象的一般格式为:

```
volatile <类名> 对象名;
```

表示对象中的数据成员是易变的, 编译程序不做优化。例如当请求该对象时, 系统总应读取其当前状态, 即使前面一个指令刚刚读过。在它赋值时, 该对象的值应该立即写入, 而不是等待某个时刻。对于一个易变对象, 只能调用其 `volatile` 成员函数, 而不能调用其它成员函数。

例 10-12 用 `const`, `volatile` 修饰的成员函数和对象。

```
#include <iostream.h>
class A{
    int i, j;
public:
    A(int a=0,int b=0){i=a;j=b;}
    void setData(int a, int b ){i=a;j=b;}
    int geti() const { return i;}
    void show( ) volatile;
    void getData(int * a, int *b)const volatile {*a=i; *b=j;}
};
```

```
void A::show( ) volatile {cout << "i="<<i<<'\t'<< "j="<<j<<'\n'; }
void main(void){
    A a1(100,200);
    const A b1(50,60);
    volatile A c1(200,300);
    a1.show();
    cout << "b1.i="<<b1.geti()<<'\n';
    c1.show();
}
```

对于对象 `a1`，可以调用其任一成员函数。而对象 `b1` 是 `const` 对象，只能调用成员函数 `geti` 和 `getData`。如果 `main` 函数中增加如下语句：

```
b1.setData(500,100);
```

就会导致编译错误，这是因为 `b1` 是常量对象，不能改变其成员数据的值。

对象 `c1` 是一个易变对象，只能调用成员函数 `show` 和 `getData`。

实际编程中，只有设计系统程序或涉及中断处理程序时才使用 `volatile` 对象和 `volatile` 成员函数，其它情况很少使用。

## 10.8 类成员的指针\*

前面介绍的指针用来指向特定类型的对象或者全局函数。对于类还有一种特殊的指针称为类的成员指针，专门指向特定类中的特定数据成员或成员函数。对一个类中的数据成员或者成员函数可以定义成员指针来指向这些成员，然后就可通过这样的指针来间接地访问这些成员。下面分别介绍数据成员的指针和成员函数的指针。

### 10.8.1 数据成员的指针

数据成员的指针就是对特定类中的特定类型的数据成员所定义的指针。一个数据成员的指针只能指向特定类中的具有特定类型的数据成员。如果一个类中具有多个公有的数据成员具有相同的类型，就可以定义指向数据成员的一个指针，并通过该指针来访问这些成员。

对于一个类，定义指向该类中的数据成员的指针变量的格式为：

```
<类型> <类名>:: *<指针变量名> [ = &<类名>::<数据成员名>];
```

其中，`<类型>`是指针变量所指向的数据成员的类型，它必须是该类中某一数据成员的类型。`<类名>`就是被指向的数据成员所在类的名字。在指针说明之后可以初始化，可以用已有的同类指针，也可以是类中某个数据成员的名称，此时要求该数据成员的类型与`<类型>`一致。

在赋值语句中，可对数据成员的指针进行赋值，格式如下：

```
<指针变量名> = &<类名>::<数据成员名>;
```

注意，在赋值时要检查当前代码对数据成员的访问权限。如果是类外代码，就要求数据成员是公有的。这里的求址运算符`&`是计算该成员在该类中的相对偏移地址，而不是运行时刻的地址。

如果一个数据成员的指针指向了特定数据成员，就可通过该指针来访问某个对象的这个数据成员，有如下两种形式：

```
<对象指针> ->* <数据成员指针变量名>
```

<对象名/对象引用> .\* <数据成员指针变量名>

这里的“->\*”和“.\*”是一种**成员指针运算符**，中间不能有空格。这两个都是双目运算符。左边的操作数是该类对象的一个指针，或者一个对象，或者一个对象引用，右边的操作数是该类的数据成员的一个指针，并且已指向某个成员。

例 10-16 数据成员的指针。

```
#include <iostream>
using namespace std;
class A{
    float x;
public:
    float a, b;
    A(float x=0, float a=0, float b=0){this->x=x;this->a=a;this->b=b;}
    float getx() {return x;}
};
int main(void){
    A s1(1,2,3), *pa = &s1;
    float A::*mptr = &A::a;           //A 定义指向 S 类中 float 成员的指针
    cout <<"pa->a="<<pa->a<<"\t";
    cout <<"pa->*mptr="<<pa->*mptr<<"\t"; //B
    cout <<"s1.*mptr="<<s1.*mptr<<endl;   //C
    mptr = &A::b;                          //D
    cout <<"pa->b="<<pa->b<<"\t";
    cout <<"pa->*mptr="<<pa->*mptr<<"\t";
    cout <<"s1.*mptr="<<s1.*mptr<<endl;
    cout<<typeid(mptr).name()<<endl;       //float A::*
    system("pause");
    return 0;
}
```

执行程序，输出如下：

```
pa->a=2 pa->*mptr=2      s1.*mptr=2
pa->b=3 pa->*mptr=3      s1.*mptr=3
float A::*
```

A 行说明了 A 类中 float 成员的一个指针 mptr，并初始化，使其指向成员 a。

B 行 pa->\*mptr 通过一个对象指针 pa 和成员指针 mptr 来访问成员。

C 行 s1.\*mptr 通过一个对象名和成员指针 mptr 来访问成员。

D 行使成员指针 mptr 指向另一个数据成员 b，然后再访问该成员。

最后显示成员指针 mptr 的类型为“float A::\*”，即类 A 中的 float 成员指针。

数据成员的指针有什么实际用处？用处很有限。一个数据成员的指针变量实际上反映了特定类中的特定类型的一组数据成员的一个集合。如果一个类中有一组相同类型的、公有的数据成员，而且这些数据成员具有某种共同语义，就可以定义一个数据成员指针来间接地访问这些成员。

## 10.8.2 成员函数的指针

一个函数的形参及其返回类型作为该函数的类型，函数的指针就是指向特定类型的函数的一种指针。前面章节介绍过全局函数的指针。下面探讨类中的成员函数的指针。

成员函数的指针就是专门指向特定类中的特定类型的成员函数的一种指针。如果一个类中多个公有成员函数都具有相同的形参和返回值，只是函数名不同，就可以定义该类的成员

函数指针，让这个指针指向某个成员函数，就能间接地调用这个函数。

对于一个类，定义指向该类的成员函数的指针变量的格式为：

<返回类型> (<类名>::\*<成员函数指针变量名>) (<形参表>) [= <类名>::<成员函数名>]

其中，<类名>是已定义的一个类名；<指针变量名>是指向该类中某个成员函数的一个指针变量名；<返回类型>和<形参表>确定了这种函数的特征。在说明之后可以进行初始化，可以是已有的一个指针，也可以指向某个成员函数。

对成员函数的指针进行赋值的语法形式是：

<成员函数指针变量名> = & <类名>::<成员函数名>; 或

<成员函数指针变量名> = <类名>::<成员函数名>;

注意，在赋值时要检查成员函数是否是公有的，只能指向公有的成员函数。这种赋值只是将指定成员函数的相对偏移地址赋给指针变量。

通过成员函数指针变量就能调用某个成员函数作用于某个对象，有两种格式如下：

(<对象指针> ->\*<成员函数指针变量名>) (<实参表>)

(<对象名/对象引用> .\* <成员函数指针变量名>) (<实参表>)

这里的“->\*”和“.\*”是一种成员指针运算符，中间不能有空格。这两个都是双目运算符。左边操作数是该类对象的一个指针，或者一个对象或引用；右边操作数是该类的成员函数的一个指针，并且已指向某个成员函数。

例 10-17 使用成员函数的指针变量。

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

class S{
    float x, y;
public:
    S(float a=0, float b=0){ x=a;y=b;}
    float getx() { return x;}
    float gety() { return y;}
    void setx(float a) { x=a;}
    void sety(float a) { y=a;}
};

int main(void){
    S s1(1, 2), *ps = &s1;
    float (S::*mptr1)() = S::getx; //A 定义成员函数指针变量
    void (S::*mptr2)(float); //B 定义成员函数指针变量
    cout <<"ps->getx()="<<ps->getx()<<"\t";
    cout <<"(ps->*mptr1)()="<<(ps->*mptr1)()<<"\t"; //C 调用成员函数
    cout <<"(s1.*mptr1)()="<<(s1.*mptr1)()<<endl; //D 调用成员函数
    mptr1 = S::gety;
    cout <<"ps->gety()="<<ps->gety()<<"\t";
    cout <<"(ps->*mptr1)()="<<(ps->*mptr1)()<<"\t";
    cout <<"(s1.*mptr1)()="<<(s1.*mptr1)()<<endl;
    mptr2 = S::setx;
    (ps->*mptr2)(3);
    cout <<"ps->x="<<ps->getx()<<endl;
    cout<<typeid(mptr1).name()<<endl;
    cout<<typeid(mptr2).name()<<endl;
    system("pause");
    return 0;
}
```



执行程序，输出如下：

```
ps->getx()=1      (ps->*mptr1)()=1      (s1.*mptr1)()=1
ps->gety()=2      (ps->*mptr1)()=2      (s1.*mptr1)()=2
ps->x=3
float (__thiscall S::*)(void)
void (__thiscall S::*)(float)
```

例子中 A 和 B 两行分别定义了两个成员函数指针，并对 `mptr1` 进行了初始化。C 和 D 两行分别通过对象指针和对象名，成员函数指针 `mptr1`，调用同一个成员函数。最后两行显示了两个成员函数指针的内部类型名，其中“`__thiscall`”表示了成员函数调用的一种缺省方式。

对成员函数的指针，应注意以下几点。

(1) 仅当成员函数的形参表和返回类型均与这种指针变量相同时，才能将成员函数的相对指针赋给这种变量。

(2) 使用成员函数指针只能调用公有成员函数。当这种指针变量指向一个虚函数，并且通过基类的指针或引用来调用虚函数时，同样具有动态多态性。虚函数在下一章介绍。

(3) 不能用成员函数指针变量来指向类中的静态成员函数，可用普通函数指针来指向静态成员函数，在调用时也不需要确定对象指针、或对象名或对象引用。

例 10-18 用普通函数指针来调用静态成员函数。

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

class S{
    float x;
    static int defx;
public:
    S(){x = defx;}
    S(float a){x=a;}
    float getx(){return x;}
    static float getdef(){return defx;}
    static float setdef(float a) {defx = a; return a;}
};
int S::defx = 1;
int main(void){
    S s1(2), *ps = &s1;
    float (*mptr1)() = S::getdef;           //A
    float (*mptr2)(float) = S::setdef;      //B
    cout <<"mptr1()="<<mptr1()<<endl;      //C
    cout <<"mptr2(3)="<<mptr2(3)<<endl;     //D
    cout<<typeid(mptr1).name()<<endl;
    cout<<typeid(mptr2).name()<<endl;
    system("pause");
    return 0;
}
```

执行程序，输出如下：

```
mptr1()=1
mptr2(3)=3
float (__cdecl*)(void)
float (__cdecl*)(float)
```

程序中的 A 行和 B 行说明了两个普通函数指针变量，并用将静态成员函数来初始化。C 行和 D 行分别调用了静态成员函数。最后显示两个普通函数指针变量的类型，其中“`__cdecl`”

是 C++ 的一个扩展的修饰符，表示了 C/C++ 函数调用的一种缺省方式。可以看出，静态成员函数在系统内部仍作为普通函数。

成员函数的指针有什么实际用处？用处比较有限。如果一个类中有一组相同类型的、公有的成员函数，而且这些成员函数具有某种共同的语义，就可以定义一个成员函数指针来选择指向其中某一个函数，从而能选择执行不同的成员函数。

## 10.9 小 结

- 每个对象都是经过某个构造函数的执行才创建出来的。构造函数是特殊的成员函数，目的是对新建对象进行初始化。至少在以下情形会创建对象：
  - ◆ 说明对象变量或数组。
  - ◆ 用 `new` 运算符来创建动态对象。
  - ◆ 创建临时匿名对象。
- 一个类至少有一个构造函数。如果没有显式定义任何构造函数就自动生成一个缺省构造函数。所谓缺省构造函数就是无需提供实参就能创建对象时所执行的构造函数，可能是自动生成的，也可能是显式定义的；可能是无参的，也可能是有参而带缺省值的。一个类中也可能没有缺省构造函数。
- 创建对象有两种来源：要么是从类中创建而来，要么从已有同类对象复制而来，即执行拷贝构造函数而来的。如果没有显式定义拷贝构造函数就自动生成一个缺省的拷贝构造函数，用来复制各个数据成员到新建对象。
- 至少在以下情形会调用拷贝构造函数：
  - ◆ 用说明语句来创建一个新对象时，用一个已有对象来初始化新建对象。
  - ◆ 调用函数时，用传值调用方式，把对象实参传递给形参。
  - ◆ 函数返回一个对象。
- 与拷贝构造函数相似，还有赋值操作函数，将一个对象的数据复制给另一个同类对象。如果类中没有显式定义赋值操作函数，就自动生成一个，复制各个数据成员给当前对象。在赋值语句或函数调用时会自动调用赋值操作函数。
- 拷贝构造函数就是一种单参构造函数，单参构造函数具有特殊用法，可以用于隐式的类型转换，就是将形参类型的值或对象自动转换为当前类的对象。当需要某个类的一个对象时，而提供了其它类型的值或对象，就可能自动调用单参构造函数创建了新对象。
- 每个对象被撤销时都会执行析构函数。一个类只能有一个析构函数，如果没有显式定义就自动生成一个。在以下情形要调用析构函数：
  - ◆ 程序执行离开对象所在的作用域。
  - ◆ 用 `delete` 运算符回收用 `new` 运算符创建的动态对象。
  - ◆ 临时对象使用完毕。

A	
- int a	
+A()	// 0 0 0 0 0 0 0 0
+~A()	// 0 0 0 0 0 0 0 0
+A(const A&a)	// 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
+A&operator=(const A&a)	// 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

上图表示了当建立一个类 `class A{int a;};` 系统自动生成 2 个缺省构造函数、1 个析构函数、1 个赋值操作函数的框架。

- 如果一个类 A 的数据成员中有另一个类 B 的一个或多个对象，那么类 A 对象就是复合对象，类 B 的对象就是成员对象。当创建一个复合对象时就要先创建其成员对象。当要撤销一个复合对象时，也要自动撤销其成员对象。
- 复合对象类的构造函数中要在成员初始化列表中确定如何对其成员对象进行初始化。如果没有确定，就调用成员对象类的缺省构造函数。复合对象执行构造函数之前，要先执行所有成员对象的构造函数，再执行复合对象的构造函数体。
- 析构函数的执行顺序与构造函数的执行顺序相反。先执行复合对象的析构函数，再执行各成员对象的析构函数。
- 一个对象数组就是一类对象的一个有序集合，数组中所有元素是同一类的对象。对象数组的定义、赋值和使用与普通数组一样。定义一个对象数组时，可以初始化，类名加实参。如果没有初始化，就调用该类的缺省构造函数。
- 对象数组经常作为一个类的数据成员。对象数组作为一个类的数据成员时，该类的构造函数中不能对数组元素进行显式初始化。数组元素的类应提供缺省构造函数。如果一个类没有缺省构造函数，该类的数组就不能作为其它类的数组成员。
- 静态成员就是用 `static` 修饰的成员，包括静态数据成员和静态成员函数。访问静态成员应该用“类名::成员名”，而不应通过对象，这是因为静态成员本来就是类的成员，而不是对象的成员。
- 一个静态数据成员表示类的多个对象所共享的一项数据。无论是否有对象，类的静态数据成员都存在。类中的静态数据成员与结构类型中的静态成员的语法规义一样。静态数据成员必须在类的外部，即文件作用域中，做定义性说明，并进行初始化。如果没有初始化，缺省为 0。
- 静态成员函数用于管理类中的静态数据成员，或者提供类的某种服务。静态成员函数中没有隐含的 `this` 指针，即没有当前对象，因此静态函数中不能直接访问该类的非静态成员，只能访问静态成员。
- `const` 可修饰类的成员函数、函数形参及返回值。
- 用 `const` 修饰一个成员函数就限制了该函数中不能改变该类对象的数据成员。
- 用 `const` 修饰一个函数形参对象，表示形参对象的数据成员都是常量，函数体中不会

改变其数据成员的值，只能调用其 `const` 成员函数，而不能调用其它成员函数。

- 用 `const` 修饰函数的返回值不可改变。当返回一个对象的指针或引用时，如果不希望通过返回的指针或引用来改变返回对象，就应该用 `const` 来修饰返回值。
- 指针可作为类的数据成员，可能是单个指针，也可能是指针的数组。指针成员有多种用途，可以管理对象内部的动态数据，也可以表示对象之间关联，在实际软件开发中经常用到。一个类中如果有指针成员，该类的拷贝操作函数、赋值操作函数、析构函数往往都需要谨慎设计。
- 类成员的指针指向某一个类中的数据成员或者成员函数，能间接地访问这些成员。
- 对于一个类，定义指向该类中的数据成员的指针变量的格式为：  
`<类型> <类名>::*<指针变量名> [= &<类名>::<数据成员名>];`
- 通过数据成员指针变量来访问某个对象的某个成员，有两种格式：  
`<对象指针>->*<指针变量名>`  
`<对象名/对象引用>.*<指针变量名>`  
 这里的“`->*`”和“`.*`”是成员指针运算符。
- 对于一个类，定义指向该类的成员函数的指针变量的格式为：  
`<返回类型> (<类名>::*<指针变量名>)(<形参表>) [= <类名>::<成员函数名>]`
- 通过成员函数指针变量就能调用某个成员函数作用于某个对象，有两种格式：  
`(<对象指针>->*<指针变量名>)(<实参表>)`  
`(<对象名/对象引用>.*<指针变量名>)(<实参表>)`  
 这里的“`->*`”和“`.*`”是成员指针运算符。
- 对于类中的静态成员函数，不能用成员函数指针来指向，可用普通函数指针来指向静态成员函数，在调用时需要确定类作用域。

## 10.10 练 习 题

1. 对于构造函数的特性，下面哪一种说法是错误的？
  - A 每一个对象都是经过某个构造函数的执行才被创建出来的。
  - B 一个类可定义多个构造函数，它们的名字或形参不同。
  - C 假设 A 是一个类，那么语句“`A as[4];`”将执行构造函数 4 次。
  - D 构造函数不能说明返回值的类型。
2. 对于缺省构造函数，下面哪一种说法是错误的？
  - A 只有当类中没有显式定义任何构造函数时，编译器才自动生成一个公有的缺省构造函数。
  - B 一个无参构造函数是缺省构造函数。
  - C 缺省构造函数一定是一个无参构造函数。
  - D 一个类中最多只能有一个缺省构造函数。
3. 对于析构函数，下面哪一种说法是错误的？
  - A 一个对象的生命周期始于构造函数执行，终于析构函数执行。
  - B 如果类中没有显式定义析构函数，编译器就自动生成一个公有的析构函数。

C 一个类中可重载定义多个析构函数。

D 假设 A 是一个类，那么语句“A as[4];”将执行析构函数 4 次。

4. 假设对象 a 和 b 都是类 A 的对象，下面哪一条语句不会执行类 A 的拷贝构造函数？

A b = a;

B void f(A x); f(a);

C A b = a;

D return a;

5. 指出以下程序中的错误，说明错误原因并改正。

```
#include <iostream.h>
class Test{
    int *ptr;
public:
    Test(int i) { ptr = new int(i); }
    ~Test(){delete ptr; }
};
void main(void){
    Test obj1(1);
    Test obj2(obj1);
}
```

6. 写出以下程序的运行结果。

```
#include <iostream.h>
class A{
public:
    A(){cout<<"def. constructor of A\n";}
    ~A(){cout<<"destructor of A\n";}
    A(const A&a){cout<<"copy constructor of A\n";}
};

class B{
    A a;
public:
    B(){cout<<"def. constructor of B\n";}
    ~B(){cout<<"destructor of B\n";}
};

void main(){
    B b1;
    B b2 = b1;
}
```

7. 写出以下程序的运行结果。

```
#include <iostream.h>
class A{
    int i;
    static int x,y;
public:
    A(int a=0,int b=0,int c=0)
    { i = a; x = b; y = c;}
    void show(){
        cout << "i="<<i<<'\\t';
        cout << "x="<<x<<'\\t'<<"y="<<y<<endl;
    }
};
int A::x=0;
int A::y=0;
```

```

void main(){
    A a(2,3,4);
    a.show();
    A b(100,200,300);
    b.show();
    a.show();
}

```

8. 写出以下程序的运行结果。

```

#include <iostream.h>
class A{
    int a, b;
    static int c;
public:
    A(int x) { a=x;}
    void f1(float x){b=a * x; }
    static void setc(int x) { c=x;}
    int f2() { return a+b+c;}
};
int A::c =100;
void main(void){
    A a1(1000), a2(2000);
    a1.f1(0.25); a2.f1(0.55);
    A::setc(400);
    cout<<"a1="<<a1.f2()<<"\n"<<"a2="<<a2.f2()<<"\n";
}

```

9. 利用 **Point** 类作为成员对象，设计实现一个矩形类 **Rectangle**，并测试该类。该类要求有下述成员：

数据成员：左上角点(x, y)，宽度 **width** 和高度 **high**，可用 **int** 类型。

成员函数：对宽度和高度属性的读 **getXxx** 和写 **setXxx**，另外还有：

**void moveOff(int xoff, int yoff)**：相对移动。

**float getArea()**：计算矩形的面积。

**float getPerimeter()**：计算矩形的周长。

**void show()**：显示矩形的数据成员，以及面积和周长。

10. 利用 **Date** 类作为成员对象，设计实现一个 **DateTime** 类，表示日期和时间，包含年月日、时分秒。

要求：类中包含 1 个数据成员 **date** 表示日期，还有 3 个数据成员：时(hour)、分(min)、秒(sec)。

缺省构造函数要求读取当前系统日期和时间。注意使用 **time.h** 中的库函数。

另一个构造函数为 **DateTime(int year, int month, int day, int hour, int min, int sec)**；

一组成员函数(setter/getter)对各成员进行读写。

一个成员函数 **void show()**；以中国人习惯格式显示各个数据成员，如果可能的话，也显示星期几。

11. 利用上题 **DateTime** 类作为成员对象，尝试设计一个 **File** 类，表示磁盘文件，包含文件名、大小、创建日期和修改日期(**DateTime** 类)等属性。

12. 改进上题 **File** 类，利用静态成员来管理一组 **File** 对象的指针的数组，这样不仅能知道当前有哪些 **File** 对象，还能按文件名、或大小、或创建日期进行排序。

13. 构造一个 **Person** 类，表示人，除了姓名、性别等属性之外，还要描述一个人的父亲和母亲，根据父母属性就能计算：1、某个人的兄弟姐妹有哪些人；2、某个人的祖父母和外祖父母有哪些人，等等。提示，利用指针成员来表示 **Person** 之间的关联。

14. 一个简单的扑克牌游戏。设计一个类 **Card**，使每一张扑克牌作为其一个对象，大王、小王这两张牌

可暂时除外。该类中需要一个枚举类型来说明 4 种花色：

```
enum Suit{heart = 3, diamond, club, spade}; //3, 4, 5, 6 可显示 4 种花色字符
```

设计合适的构造函数，一个成员函数用来显示一张牌的花色和点数。

再设计一个类 Deck，使一幅扑克牌作为该类的一个对象，还要提供洗牌和发牌功能。

设计一个缺省构造函数来构造 52 张牌。此时考虑 Card 类是否需要一个缺省构造函数。

设计一个成员函数来打印所有牌的序列。

设计一个成员函数 void shuffle(...)来完成洗牌，应提供足够的随机性。

设计一个成员函数 void deal(int n)将 52 张牌循环发给 n 个玩家。

设计一个成员函数 void showCards(int player)显示发给玩家 player 的所有牌。

尝试设计一个类玩 2 个人(或人机对抗)的一种简单游戏，先发牌，再出牌决输赢。

发牌时，发给 2 个人，每人 2 张牌。实现一个简单游戏规则，比如每人两张牌，制定一个比较牌面大小的规则。比如，对子大于顺子，顺子大于散牌，对散牌比较最大牌点数。如最大牌点一样，再比较第二张。如果两个人的两张牌的点数都一样，还可以定义花色的大小。

尝试增强该类的功能，如记录每一局的成绩，最后得到汇总。也可以加一点赌注来试试。

15. 仿制手机上的通信录(名片夹)，先建立一个类表示通信录中的个人记录，一个记录对应一个联系人，要记录其姓名、手机号、座机号、电子邮箱等信息。再建立一个类表示分组，方便查找和发送短信。一个分组中包含多个联系人。一个联系人可加入多个分组。在分组中可以看到各联系人的信息。当删除一个分组时不删除其成员的个人记录。分析下面问题。在一款 Nokia 手机的名片夹中发现一个问题，当对一个分组中的多个成员发送一个短信时(称为群发)，如果一个联系人既有手机号，也有座机号，就要选择发送到哪一个号码，而且每次都要选择。因为现在的座机号码可能是小灵通，也能接收短信。你有什么办法能简化这种群发操作？

16. 尝试设计一个整数集合类 IntSet，一个整数集合作为一个对象。该类包括以下数据成员：

```
int element[100];           //保存整数集中的数据
int endPosition;            //指示整数集的最后一个元素位置，初始为 0
注意：集合中不允许有相同元素存在。该类包括以下成员函数：
IntSet();                  //缺省构造函数
int size()                 //返回当前集合中元素个数
void clear();              //清空该整数集合
bool isEmpty();            //判断整数集合是否为空
bool isMemberOf(int a);    //判断某个整数 a 是否在当前集合中
void add(int a);           //增加一个整数 a 到当前集合中
void remove(int a);        //从当前集合中删除一个整数元素 a
bool isEqual(IntSet & other); //判断两个集合是否相等
IntSet intersection(IntSet & other); //求两个整数集合的交集
IntSet merge(IntSet & other);  //求两个整数集合的并集
void print();              //依次打印该集合中各元素
```

该类是否需要自行定义拷贝构造函数和赋值操作函数？哪些成员函数中将调用拷贝构造函数？

本题目设计一种整数的集合容器类，第 13 章将介绍标准模板库 STL 中的相关容器。

## 第11章 类的继承

前面介绍了面向对象程序设计的一个重要特性——封装性(encapsulation)。本章介绍另外两个重要特性——继承性(inheritance)和多态性(polymorphism)。

继承性是类之间的一种关系，表示了一个比较抽象的类与一个比较具体的类之间关系。比较抽象的类称为基类、而比较具体的类称为派生类。派生类的一个对象也是基类的一个对象。派生类继承了基类中定义的成员，而且可以扩展新成员。继承性是软件复用的一种形式。

多态性的一般含义是一个名字有多种具体解释，C++语言中的多态性有编译时刻的静态的多态性，如函数重载、运算符重载函数，还有运行时刻的动态多态性。动态多态性与继承性相关，本章将介绍动态多态性。后面章节将介绍运算符重载函数。

### 11.1 继承与派生

下面介绍如何建立类之间的继承性关系。

#### 11.1.1 基类与派生类

考虑人与大学生之间的关系。首先，一个人作为一个对象，建立一个 **Person** 类如下：

```
class Person{
    string name;           //姓名
    char sex;              //性别, f=Female; m=Male
    string idno;           //身份证号
    Date birthdate;        //出生日期, 成员对象
    ...                    //其它成员
};
```

学生如何描述？一名学生作为一个对象，建立一个 **Student** 类可能如下：

```
class Student{
    string name;           //姓名
    char sex;              //性别, F=Female; M=Male
    string idno[20];       //身份证号
    Date birthdate;        //出生日期, 成员对象
    string schoolName;     //所在学校名称
    Date enrollDate;       //入学日期
    ...                    //其它成员
};
```

我们发现一名学生完全包含了一个人的全部属性和行为。这是因为一名学生本来就是一个人，只是扩展了一些特殊属性，如学校名称、入学日期等。我们不能在 **Student** 类中包含一个 **Person** 对象作为其成员对象，否则就形成了错误的语义描述。

正确的描述应该是，一名学生作为 **Student** 类的一个对象，同时也是 **Person** 类的一个对



象,他/她继承了 **Person** 类为其提供的作为人的属性和行为,再扩展自己的属性和行为。简言之,一名学生是具有学校、入学日期等属性的一个人,描述如下:

```
class Student : public Person{//Student 类作为 Person 的派生类
    string schoolName;        //所在学校名称
    Date enrollDate;          //入学日期
    ...                        //其它成员
};
```

**Student** 类是从类 **Person** 派生而来的。**Person** 类就是 **Student** 的基类(base class)、或超类(super class)、或父类(parent class)。**Student** 类继承了 **Person** 类的所有成员,再扩展自己的一些成员。**Student** 类是 **Person** 的一个派生类或衍生类(derived class)、或子类(subclass)、或扩展类(extend class)。这种派生类从基类中继承成员的关系就称为类的继承性。

在继承性关系中,基类表示比较抽象的概念,拥有较少的内涵(即较少的属性),而具有较大的外延(即较大的集合范畴)。而派生类表示比较具体的概念(也称为具象),拥有较多的内涵(即较多的属性),而具有较小的外延(即较小的集合范畴,派生类对象集合是其基类对象集合的子集)。

在 C++ 派生类中,仅描述扩展部分的属性,而派生类对象则拥有其所有的直接或间接的基类中描述的属性。因此派生类的定义是基于其基类而存在的。

从派生类的角度,根据它所拥有的基类数目不同,可分为单继承和多继承。如果一个派生类只有一个直接的基类,称为单继承。如果一个类同时有多个基类,则称为多继承或多重继承。单继承和多继承的关系如图 11.1 所示。图中用一个三角箭头从派生类指向其基类,表示继承关系。

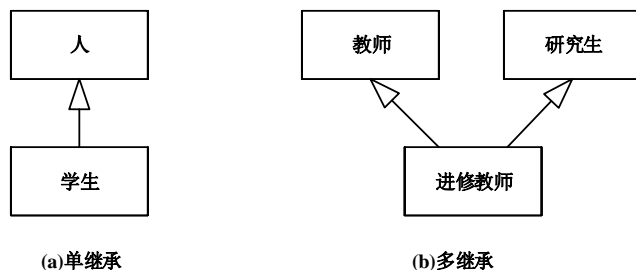


图 11-1 单继承与多继承

抽象与具象是相对而言的,因此基类与派生类之间的关系是相对而言的。一个派生类又可作为另一个类的基类,如此形成继承层次结构。基类与派生类之间的关系如下:

(1) 基类是对派生类的抽象,派生类是对基类的具象。基类抽取其派生类的一般特征作为成员,而派生类通过扩展新成员来表示更具体的类型,派生类是基类定义的延续和扩展。

(2) 派生类的一个对象属于其基类,也是基类的一个对象。当需要一个基类对象的地方,而实际提供了派生类的一个对象,应该是无条件满足要求的。这就是继承性的最基本原则——里氏替代原则。如果这种替代是有条件的,那么这个继承性关系就存在潜在问题。

### 11.1.2 派生类的定义与构成

定义派生类的一般格式如下:

```

class <派生类名> : <继承方式 1> <基类名 1>,
                  <继承方式 2> <基类名 2>,
                  ...,
                  <继承方式 n> <基类名 n>{
    <派生类扩展的成员>
};

```

其中, <基类名>是已有的类名, 单继承时只有一个基类名, 多继承时有多个基类名。<继承方式>规定了派生类对基类成员的访问控制方式, 控制基类成员在什么范围内能被派生类访问。每一种继承方式, 只对紧随其后的一个基类进行限定。

继承方式有三种: **public**(公有继承)、**private**(私有继承)和 **protected**(保护继承)。如果省略继承方式, 缺省为 **private** 私有继承, 而最常用的是 **public** 公有继承。下一节将讨论这三种方式的差别。

派生类的一个对象的成员由两部分构成: 一部分是从基类继承来的, 另一部分是自己扩展的新成员, 所有这些成员仍然分为 **public**、**private** 和 **protected** 三种。其中, 从基类继承而来的全部成员构成派生类的继承部分, 此部分的私有成员是派生类不能直接访问的, 其公有和保护成员则是派生类可以直接访问的, 但是它们在派生类中的访问属性将随着派生类对基类的继承方式而改变。只有单个基类的派生类对象的构成如图 11.2 所示。

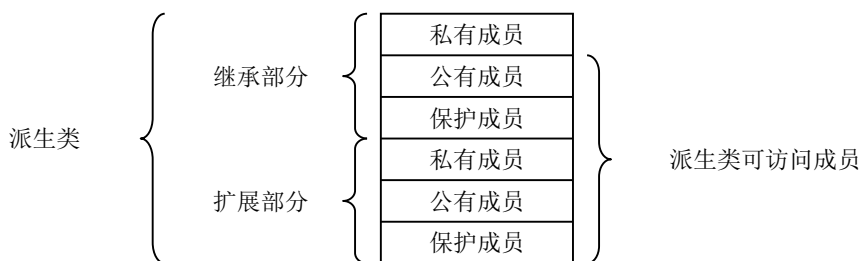


图 11.2 派生类的构成

从对象存储来看, 每个派生类对象所占有的存储空间就是其基类部分的所有非静态数据成员与扩展部分的所有非静态数据成员的总和。

### 11.1.3 继承方式与访问控制

派生类继承了基类的全部数据成员和成员函数, 但是这些成员在派生类中的访问控制属性是可以调整的, 继承方式就确定了基类成员在派生类中的可见性。

派生类对于一个基类要确定一种继承方式, 有 **private**、**public**、**protected** 三种选择(如图 11.3)。每一个继承都必须选择一种方式。缺省为“私有继承 **private**”。但最常用的却是“公有继承 **public**”。不

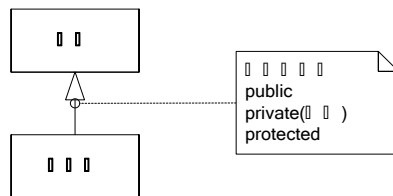


图 11-2 3 种不同的继承方式

同的继承方式导致被继承的基类成员在派生类中的可见性有所改变。如表 11.1 所示。表中√表示派生类可以访问, ×表示不可访问, 如果基类成员的可见性在派生类中改变就用括号说

明。

表 11-1 派生类对基类成员的访问能力

<div>继承方式 基类成员</div>	公有继承 public	私有继承 private (缺省)	保护继承 protected
私有成员 private	×	×	×
公有成员 public	√	√(私有)	√(保护)
保 护 成 员 protected	√	√(私有)	√(保护)

从基类的成员来看，不同的继承方式具有不同的作用：

- 对于基类中的私有成员(表中第 1 行)，在派生类中是不可见的。在派生类中不能直接访问基类中的私有成员。当然外部程序也不能访问私有成员。任何继承方式都不能改变其私有可见性。
- 对于基类中的公有成员(表中第 2 行)，可被派生类、外部程序访问。如果是私有继承方式，基类中的公有成员就改变为派生类自己的私有成员，这样就阻止了外部程序对一个派生类对象访问其基类的公有成员。如果是保护继承方式，基类中的公有成员就改变为派生类自己的保护成员，只能被自己的派生类访问。
- 对于基类中的保护成员(表中第 3 行)，派生类可访问，但外部程序不能直接访问保护成员。如果是私有继承方式，基类中的保护成员就改变为派生类自己的私有成员，阻止了下一层派生类中对这些保护成员的访问。

派生类可以访问其基类中的公有成员和保护成员。派生类对基类中的公有成员和保护成员的访问能力与继承方式无关，但继承方式会影响基类成员在派生类中的可见性：

- 公有继承时(表中第 1 列)，基类的公有成员和保护成员在派生类中仍为公有成员和保护成员，并非改变为公有成员。这是最常用的继承方式。实际上后面只用这种方式。
- 私有继承时(表中第 2 列)，基类的公有成员和保护成员在派生类中都改变为私有成员。这就阻止了外部程序对派生类对象访问其基类的公有成员，也阻止了下一层派生类来访问基类的保护成员。这是缺省继承方式，但不常用。
- 保护继承时(表中第 3 列)，基类的公有成员在派生类中改变为保护成员，基类的保护成员在派生类中仍为保护成员。这就阻止了外部程序对于派生类的对象访问其基类的公有成员。这是最不常用的继承方式。

前面我们主要使用类中的公有成员和私有成员。在类的继承关系中，保护成员具有自己的特色。基类中的保护成员是专门提供给自己的“同类”访问的。这里“同类”指的是具有同一个基类的一组直接或间接的派生类。

分析下面程序。

```
class A{
protected:
    int x;
};
class B : public A{
```

```

public:
    void fun() { x=1; }                //A
};
void main(void){
    A objA;
    B objB;
    objA.x=2;                        //B
    objB.fun();
}

```

类 A 中有一个保护成员 x，类 B 是类 A 的派生类。那么类 B 就能访问其基类中的保护成员，如 A 行。但 B 行在派生类之外试图访问基类的保护成员，将导致编译错误。

在派生类的成员函数中要访问其基类的某个数据成员的格式如下：

[this->][<基类名>::]<数据成员名>

如果该成员名在其基类中是唯一的，可以省略基类名。

在派生类的成员函数中要调用其基类的某个成员函数的格式如下：

[this->][<基类名>::]<成员函数名>(<实参表>)

如果该成员函数在其基类中是唯一的，则可以省略基类名。

## 11.2 派生类的构造和析构

派生类的一个对象也是其基类的一个对象，因此在创建派生类的一个对象时，就要调用基类的构造函数来初始化基类成员，还要调用派生类自己的构造函数对新增成员进行初始化。当派生类对象被撤销时，不仅要调用自己的析构函数，还要调用其基类的析构函数。

### 11.2.1 派生类的构造函数

派生类对象的数据成员由所有基类的数据成员与派生类自己扩展的数据成员共同组成。派生类扩展成员中还可以包含成员对象。因此创建派生类的一个对象时，必须对基类数据成员、扩展的成员对象和扩展的基本类型成员进行初始化。派生类的构造函数必须用合适的初值作为实参，调用基类和成员对象的构造函数，以便初始化它们各自的数据成员，然后再执行函数体。

派生类构造函数的一般格式为：

```

<派生类名>::<派生类名>(<形参表>)
: <基类名 1>(<实参表 1>), ..., <基类名 n>(<实参表 n>),
<成员名 1>(<实参表 n+1>), ..., <成员名 m>(<实参表 n+m>)
{
    <派生类构造函数体>
}

```

在形参表之后，函数体之前是一个基类/成员初始化列表，其中包括每个直接的基类名及其实参表，对应各基类的构造函数形参，每个直接的成员名及其实参表，对应各成员类的构造函数形参。各项之间用逗号分隔，但没有严格次序。如果没有确定任何一项，就自动调用基类或成员类的缺省构造函数，而此时如果没有提供缺省构造函数，就导致编译错误。

实际上，基本类型的成员也可以放在成员初始化列表中进行初始化，而不用放在函数体

中进行初始化, 如果是简单赋值的话。

派生类的构造函数负责对其基类成员、成员对象以及基本类型成员进行初始化, 它们的执行顺序如图 11.4 所示:

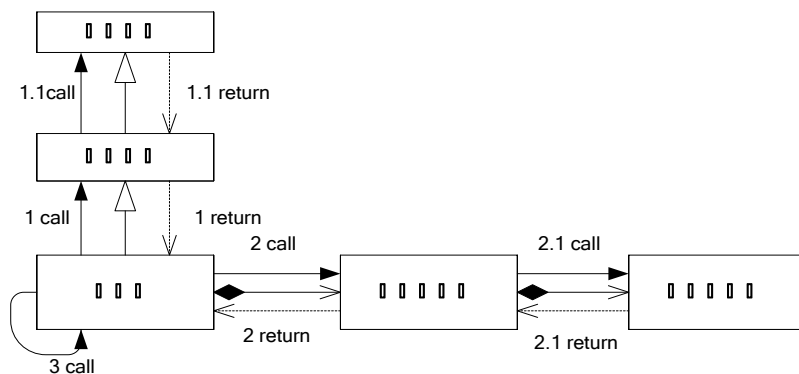


图 11-3 派生类创建对象的过程

(1) 先调用各个直接基类的构造函数, 调用顺序按照派生类中的说明顺序 (自左向右)。注意这是一个递归的调用过程, 从直接基类到间接基类, 逐层向上调用。执行完成的次序正相反, 最上层的基类先执行完, 再到下一层。

(2) 再调用各个成员对象的构造函数, 调用顺序按照它们在派生类中的说明顺序 (自上而下)。这也是一个递归的调用过程, 从直接成员类到间接成员类, 逐层向内调用。执行完成的次序正相反, 最内层的成员先执行完, 再到外一层。

(3) 最后执行派生类自己的构造函数体, 对基本类型成员初始化。

缺省提供的构造函数也按以上过程来创建对象, 并要求被调用的所有类都要提供缺省构造函数。如果某个基类或成员类没有缺省构造函数, 派生类就必须显式定义构造函数, 并在基类/成员初始化列表中明确给出调用实参。

缺省提供的拷贝构造函数也按以上过程来创建对象, 也同样在其基类/成员初始化列表中调用基类和成员类的拷贝构造函数, 并在函数体中复制基本类型成员。

派生类中也可设计委托构造函数与目标构造函数。其中目标构造函数应调用基类构造函数 (显式调用非缺省, 或者隐式调用缺省构造), 而委托构造函数则不能调用基类构造函数。例如:

```
class Base {
public:
    Base(int b) { cout << b << endl; }
};
class X : public Base{
    int type = 1;
    char name = 'a';
    X(int i, char e) : Base(i), type(i), name(e) { /* 其他初始化 */ }
public:
    X() : X(1) { }
    X(int x) : X(x, 'a') { }
    X(char e) : X(1, e) { }
};
```

目标构造函数的初始化中，必须调用基类 **Base** 的构造函数，而下面几个委托构造函数都不能调用基类构造，否则编译错误。

### 11.2.2 派生类继承构造函数

基类中定义了一组构造函数，而派生类中如果没有扩展自己的数据成员，因此不需要自己的初始化，但必须以相同的形参来“透传”基类的构造函数。例如：

```
class A {
public:
    A(int i) {}
    A(double d, int i) {}
    A(float f, int i, const char* c) {}
    // ...
};

class B : public A {
public:
    B(int i) : A(i) {}
    B(double d, int i) : A(d, i) {}
    B(float f, int i, const char* c) : A(f, i, c) {}
    virtual void extraInterface() {} //A
};
```

派生类 **B** 仅仅为了添加 1 个虚函数(A 行)，但必须设计自己的 3 个构造函数，分别调用基类的 3 个构造函数。其间并没有自己的行为，只是重复基类的构造函数。

**C11** 允许派生类继承基类的构造函数，免除重复的构造函数设计。需要派生类中添加 using 说明语句：

```
class B : public A {
public:
    using A::A;           //说明继承基类的全部构造函数
    virtual void extraInterface() {}
};
```

**C14** 允许数据成员初始化。可为派生类扩展带初始化的数据成员，而无需显式定义构造函数：

```
class B : public A {
public:
    using A::A;
    int b = 0;           //A 带初始化的数据成员
};

int main() {
    B b(11);             //B 自动具有 B(int) 构造函数
    return 0;
}
```

A 行添加一个带初始化的数据成员，B 行创建派生类对象时调用了 **B(int)**构造函数。

派生类继承构造函数并不妨碍添加自己的构造函数。

```
class B : public A {
    int b = 0;
public:
    using A::A;
    B(int x) : A(x), b(x) {}           //A 添加自己的构造函数
};
```

```

int main() {
    B b1(11);                //B 调用自己的构造函数
    B b2(3.4, 5);            //C 调用继承构造函数
    B b4(2.3f, 4, nullptr);
    return 0;
}

```

派生类只能继承其基类的所有构造函数，而无法选择继承特定的构造函数。

继承构造函数本质上是，按基类构造函数的基调自动生成派生类的构造函数。

### 11.2.3 派生类的析构函数

派生类的析构函数的执行过程与构造函数严格相反。撤销一个派生类对象的过程如下：

(1) 先执行派生类自己的析构函数，对基本类型成员进行清理。

(2) 再调用各成员对象的析构函数，对各成员对象进行清理。这是一个递归调用过程，与构造过程的次序相反。

(3) 最后调用各基类的析构函数，对各基类中的数据成员进行清理。这是一个递归调用过程，与构造过程的次序相反。

注意，派生类的构造函数体中并没有调用成员类或基类的析构函数，上面的析构过程是系统自动调用执行的。

例 11-1 前面例 10-7 中设计了 **Person** 类和 **Date** 类，在此基础上设计一个 **Student** 类。这三个类的设计如图 11.5 所示。

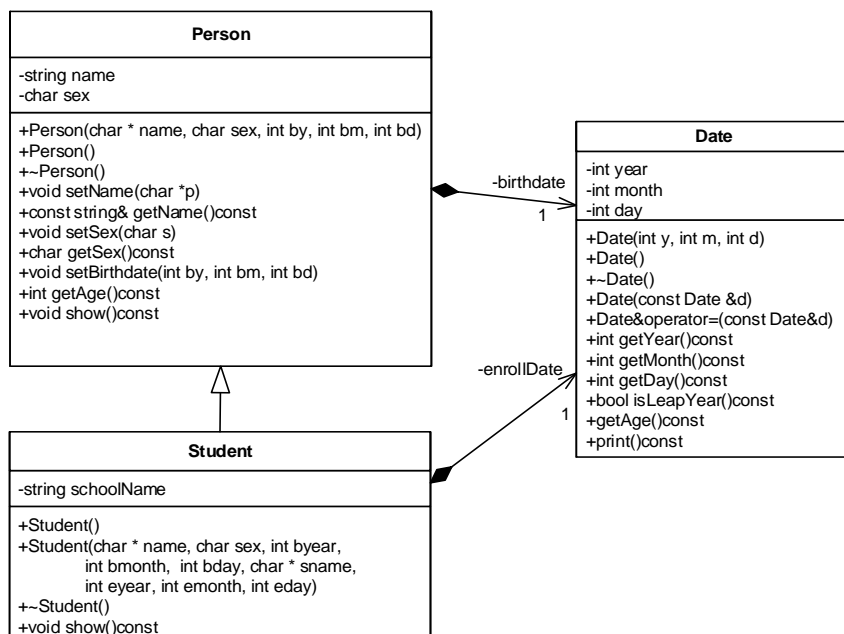


图 11-4 Student 类的设计

这个例子有 3 个源文件，分别对应 3 个类：Date.h、Person.cpp 和 Student.cpp。前两个类与例 10-7 大致相同，只是添加了一些 const 修饰，不赘述，下面是 Student 类的编码：

```
//Student.cpp
```

```

#define _CRT_SECURE_NO_WARNINGS
#include <string>
#include "Person.cpp"
using namespace std;

class Student: public Person{           //说明基类
    string schoolName;
    Date enrollDate;
public:
    Student(){cout<<"Default constructor of Student"<<endl;}
    Student(char * name, char sex, int byear, int bmonth, int bday,
            char * sname, int eyear, int emonth, int eday)
        :Person(name, sex, byear, bmonth, bday),    //基类初始化
          schoolName(sname),                        //成员 1 初始化
          enrollDate(eyear, emonth, eday){          //成员 2 初始化
        cout<<"Constructor of Student"<<endl;
    }
    ~Student(){
        cout<<"Destructor of Student"<<endl;
    }
    void show()const{
        Person::show();
        cout<<"所在学校:"<<schoolName<<";入学日期:";
        enrollDate.print();cout<<endl;
    }
};

void test1() {
    Student s1("张三", 'm', 1991, 3, 23, "南京理工大学", 2009, 9, 3); //A
    s1.show(); //B
}

```

执行程序，输出如下：

```

Constructor1 of Date:1991.3.23
Constructor of Person
Constructor1 of Date:2009.9.3
Constructor of Student
他是张三;出生日期为 1991.3.23;年龄为 25
所在学校:南京理工大学;入学日期:2009.9.3
Destructor of Student
destructor of Date:2009.9.3
Destructor of Person
destructor of Date:1991.3.23

```

A 行调用 **Student** 类的构造函数来创建一个对象，先调用基类 **Person** 的构造函数，其中先执行 **Date** 构造函数来构建基类的成员，再执行 **Person** 构造函数，然后再执行 **Date** 构造函数来构建 **Person** 的成员 **enrollDate**，最后再执行派生类 **Student** 构造函数体。输出了前 4 行。

B 行执行输出第 5、6 行。

当 **main** 函数返回时就要撤销对象，严格按构造过程相反的次序执行析构函数，输出了最后 4 行。

**Student** 类中设计了一个缺省构造函数，其中虽然没有显式确定对基类和成员如何初始化，但实际上系统将自动调用其基类和成员类的缺省构造函数。执行下面测试函数：

```

void test2(){
    Student s2;           //调用缺省构造函数
    s2.show();
}

```



```
}
```

执行程序，输出如下：

```
Default Constructor of Date:2016.7.15
Default Constructor of Person
Default Constructor of Date:2016.7.15
Default constructor of Student
她是无名氏;出生日期为 2016.7.15;当前年龄为-1
所在学校:;入学日期:2016.7.15
Destructor of Student
Destructor of Date:2016.7.15
Destructor of Person
Destructor of Date:2016.7.15
```

**Student** 类中虽然没有显式定义拷贝构造函数，但缺省提供的拷贝构造函数将自动完成基类和成员的初始化。缺省提供的拷贝构造函数大致如下：

```
Student(const Student& s)
    :Person(s), schoolName(s.schoolName), enrollDate(s.enrollDate){}
```

执行下面测试：

```
void test3(){
    Student s1("张三", 'm', 1991, 3, 23, "南京理工大学", 2009, 9, 3);
    Student s2 = s1;    //A 调用拷贝构造函数
    s2.show();
}
```

执行程序，输出如下：

```
Constructor1 of Date:1991.3.23
Constructor of Person
Constructor1 of Date:2009.9.3
Constructor of Student
Copy constructor of Date:1991.3.23
Copy constructor of Date:2009.9.3
他是张三;出生日期为 1991.3.23;当前年龄为 25
所在学校:南京理工大学;入学日期:2009.9.3
Destructor of Student
Destructor of Date:2009.9.3
Destructor of Person
Destructor of Date:1991.3.23
Destructor of Student
Destructor of Date:2009.9.3
Destructor of Person
Destructor of Date:1991.3.23
```

在上面 A 行调用了 **Student** 类的拷贝构造函数。可以看到，先执行了基类的成员 **birthdate** 的拷贝构造函数，再执行基类 **Person** 拷贝构造函数，然后执行派生类的成员 **enrollDate** 的拷贝构造函数，最后再执行自己的拷贝构造函数体，缺省为空体。

**Student** 类中虽未显式定义赋值操作函数，但缺省提供的赋值操作函数将自动完成对基类和成员的初始化。缺省提供的赋值操作函数大致如下：

```
Student & operator=(const Student& s){
    Person::operator =(s);    //调用基类的赋值操作函数
    schoolName = s.schoolName;
    enrollDate = s.enrollDate;    //调用 Date 赋值操作函数
    return *this;
}
```

其中先调用基类 **Person** 的赋值操作函数来复制基类的成员，然后再复制自己的成员。执行测试函数如下：

```
void test4(){
    Student s1("张三", 'm', 1991, 3, 23, "南京理工大学", 2009, 9, 3);
    Student s2;
    s2 = s1;          //A
    s2.show();
}
```

执行程序，输出如下：

```
Constructor1 of Date:1991.3.23
Constructor of Person
Constructor1 of Date:2009.9.3
Constructor of Student
Default Constructor of Date:2016.7.15
Default Constructor of Person
Default Constructor of Date:2016.7.15
Default constructor of Student
operator= of Date:1991.3.23
operator= of Date:2009.9.3
他是张三; 出生日期为 1991.3.23; 当前年龄为 25
所在学校:南京理工大学; 入学日期:2009.9.3
Destructor of Student
Destructor of Date:2009.9.3
Destructor of Person
Destructor of Date:1991.3.23
Destructor of Student
Destructor of Date:2009.9.3
Destructor of Person
Destructor of Date:1991.3.23
```

可以看到，在 A 行执行赋值操作函数时，先调用基类 **Person** 的赋值操作函数，其中先复制基类的成员 **birthdate**，再执行基类的赋值操作函数，最后再执行派生类成员的赋值操作函数对 **enrollDate** 进行复制。

## 11.3 二义性问题

当用一个成员名字来访问某个成员时，如果不能确定究竟是哪一个，就形成了二义性问题。这里讨论的二义性问题来自多继承。

### 11.3.1 多继承造成的二义性

在派生类中对基类成员的访问应该是唯一确定的。但在多继承情况下，就可能出现不唯一的情形，这时就对基类成员的访问产生了二义性问题。

如下图所示，在两个直接基类中都有一个公有的同名成员函数 f()。代码如下：

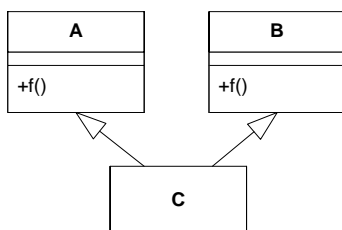


图 11-5 在多继承结构中出现二义性

```

#include <iostream.h>
class A{
public:
    void f(){ cout<<"f() in Class A.\n ";}
};
class B{
public:
    void f(){ cout<<"f() in Class B.\n ";}
};
class C : public A, public B{};
void main(void){
    C obj;
    obj.f();           //A 产生二义性
}
  
```

编译程序时 A 行产生二义性错误。无法确定调用基类 A 中的成员函数 f，还是基类 B 中的成员函数 f，这样就产生了二义性。

产生这种二义性问题的原因是多继承，来自不同基类中的数据成员或成员函数有同名。此时在派生类中访问这些成员时就不能确定究竟访问哪一个成员，从而导致二义性。

对于这类问题，有两种解决方法：

(1) 修改基类中发生重名的成员的名字。显然这不是好办法，因为改名的类可能被其它类使用，这样会造成其它多个使用类都要修改。

(2) 通过作用域运算符(::)明确指出被访问的成员属于哪个基类。例如，将 D 行写为：

obj.A::f();或 obj.B::f();

使用作用域运算符对类的成员进行限定的一般格式是：

<对象名>.<基类名>::<数据成员名>

<对象名>.<基类名>::<成员函数名>(<实参表>)

也可以用对象指针来访问成员：

<对象指针>-><基类名>::<数据成员名>

<对象指针>-><基类名>::<成员函数名>(<实参表>)

### 11.3.2 支配规则

如果在派生类中定义了与基类同名的成员，是否会导致二义性？回答是不会导致二义性，原因是派生类的作用域优先于其基类的作用域。

前面介绍过作用域的概念。在有包含关系的两个作用域中，外层说明的标识符如果在内

层没有说明同名标识符，那么它在内层可见。如果内层说明了同名标识符，那么外层标识符在内层就不可见，这时内层标识符“隐藏了”外层同名的标识符。

在类的继承层次中，基类的成员和派生类扩展的成员都具有类作用域，二者的作用域范围不同，基类在外层，而派生类在内层。如果派生类定义了一个与基类成员同名的成员（如果是成员函数，则形参的个数和类型也相同），那么派生类的成员就隐藏了外层同名成员，直接使用成员名只能访问到派生类的成员。

支配规则(dominance rule)规定了基类与派生类之间同名成员的访问规则，如下：

类 X 是类 Y 的一个派生类，那么类 X 中的成员 N 就支配类 Y 中同名的成员 N。如果一个名字支配另一个名字，那么二者之间不存在二义性。此时使用该名字 N，指的就是派生类中的支配成员。

例 11-2 支配规则和作用域运算符。程序结构如下图所示。

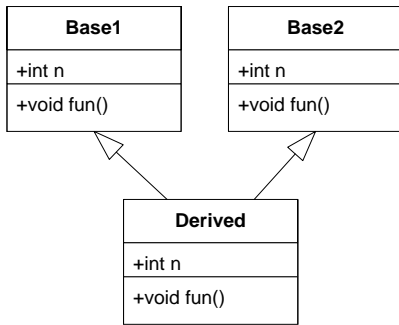


图 11-6 支配规则的例子

```

#include <iostream>
using namespace std;
class Base1{
public:
    int n;
    void fun() { cout<<"Member of Base1:"<<n<<endl; }
};
class Base2{
public:
    int n;
    void fun() { cout<<"Member of Base2:"<<n<<endl; }
};
class Derived : public Base1, public Base2{
public:
    int n;
    void fun() {
        n = 2; //A
        cout<<"Member of Derived:"<<n<<endl;
    }
};
int main(void){
    Derived obj;
    obj.n = 1; //B
    obj.fun(); //C
    obj.Base1::n = 3; //D
}
    
```

```

obj.Base1::fun();           //E
obj.Base2::n = 4;
obj.Base2::fun();
system("pause");
return 0;
}

```

执行程序，输出如下：

```

Member of Derived:2
Member of Base1:3
Member of Base2:4

```

类 **Derived** 由两个基类 **Base1** 和 **Base2** 公有派生，在这 3 个类中都有同名的数据成员 **n** 和成员函数 **fun**。

A 行和 B 行都是访问派生类 **Derived** 中的成员 **n**。这是因为派生类的成员名字支配了其基类中的同名成员，同样 C 行调用执行了派生类 **Derived** 的成员函数 **fun()**。如果要访问基类中被支配的成员，就要使用作用域运算符，如 D 行和 E 行，限定了 **Base1** 的成员 **n** 和 **fun**。同理，**obj.Base2::n** 和 **obj.Base2::fun** 分别访问 **n** 和 **fun** 时，采用作用域运算符明确限定了访问的是 **Base2** 类中的成员 **n** 和 **fun**。

在多继承结构中，来自两个基类的同名成员如果可见性不同，仍有二义性。来自两个基类的同名成员函数如果形参不同，也导致二义性，而不会自动作为重载函数对待。这种情况下支配规则不起作用。例如下面程序：

```

class Base1{
    int n;
public:
    void fun(int i) { cout<<"Member of Base1."<<endl; }
};
class Base2{
public:
    int n;
    void fun() { cout<<"Member of Base2."<<endl; }
};
class Derived : public Base1, public Base2{
public:
    void fun(double d, int a) {
        n = 3;           //A 希望访问 Base2::n, 错误
        fun();           //B 希望调用 Base2::fun(), 错误
        fun(5);          //C 希望调用 Base1::fun(int), 错误
    }
};
void test(void){
    Derived obj;
    obj.n = 3;           //错误
    obj.fun();           //错误
    obj.fun(5);          //错误
}

```

因为 **Base1** 中的成员 **n** 是私有的，派生类中不可能访问。A 行希望能访问 **Base2** 中的公有的成员 **n**，在语义上是合理的，但编译器仍产生二义性错误。

B 行希望能调用 **Base2** 中的公有的成员函数 **fun()**，它不同于 **Base1** 中的成员函数 **fun(int)**，然而 B 行也是错误的。这是因为来自两个基类的同名函数 **fun**，尽管形参不同，在派生类中也不能形成合法的重载函数。故此 C 行也会导致编译错误。

同理，上面 main 函数中的成员访问也导致同样的编译错误。

此时使用作用域运算符能消除这些语法错误。

更好的办法是采用 using 说明语句，使派生类继承基类中的同名但形参不同的公有成员函数，同时消除二义性。在上面 Derived 类中添加 3 个 using 说明如下：

```
class Derived : public Base1, public Base2 {
public:
    using Base2::n;           //将 Base2::n 纳入本类成员
    using Base2::fun;         //将 Base2::fun 成员函数纳入本类成员
    using Base1::fun;         //将 Base1::fun 成员函数纳入本类成员
    void fun(double d, int a) {
        n = 3;                //A 访问 Base2::n
        fun();                 //B 调用 Base2::fun()
        fun(5);                //C 调用 Base1::fun(int)
    }
};
```

上面 3 个 using 说明将基类中的成员纳入本类，使本类能将基类的两个 fun 成员函数作为重载函数进行调用。这样就免除了调用时必须指定基类和作用域运算符的麻烦。

## 11.4 虚基类

多继承导致的二义性可以通过作用域运算符来解决语法问题，但是有一种情形是多个基类之间还存在共同的基类，虽然作用域运算符能消除编译错误，但语义错误仍然存在，这就需要定义虚基类。

### 11.4.1 共同基类造成的二义性

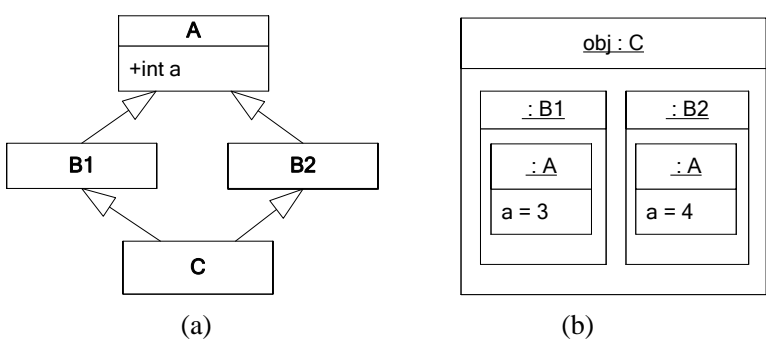


图 11-7 共同基类的多继承结构

在多继承结构中，产生二义性还有另一种情形。如图 11.7(a)所示。当一个派生类由多个基类派生，而这些基类又有一个共同的基类(可能是间接的)。这种菱形的继承结构被称为“钻石继承结构”。只要允许多重继承，就可能出现钻石结构。当访问共同基类中的数据成员时，就会出现二义性。程序如下：

```
class A{
public:
    int a;
```

```
};
class B1 : public A{};
class B2 : public A{};
class C : public B1, public B2{};
void main(){
    C obj;
    obj.a = 3;                                //A 产生二义性
}
```

编译 A 行产生二义性。尝试用作用域运算符来解决二义性问题。但要注意，派生类 C 的直接基类 B1 和 B2 拥有一个共同基类 A，因此 obj.A::a 这种限定形式是错误的，它不能指出有效的访问路径，正确的形式应该是 obj.B1::a 或 obj.B2::a。例如：

```
obj.B1::a = 3;
obj.B2::a = 4;
```

这种方法虽能解决语法问题，但却不能避免一个突出的语义问题。如图 11.7(b)所示。为了说明派生类的对象的内部结构，引入“子对象”的说法。派生类的一个对象包含了每个基类的一个对象，称为一个子对象。子对象区别于成员对象，表示了基类的数据成员部分。类 C 的一个对象中包含了 B1 和 B2 两个子对象，而它们分别包含了类 A 的一个子对象，这样类 C 的一个对象中就包含了类 A 的两个子对象。

这就存在一个问题，类 C 的一个对象应该是类 A 的一个对象，还是两个对象？如果是一个对象，那么应该只持有唯一的 a 值，而不是两个值。如果是两个对象，这又违背了继承性的基本原则：**派生类的一个对象是其基类的一个对象**。显然类 C 的一个对象应包含类 A 的一个子对象，应持有唯一的 a 值。

产生这种二义性的原因是基类 A 在派生类 C 中产生了两个基类成员 a，从而导致了对于基类 A 的成员 a 的访问是不唯一的。解决这个问题，只要使这个公共基类在派生类对象中只产生一个子对象，只需一次初始化。定义虚基类可以解决这个问题。

#### 11.4.2 虚基类的说明

虚基类的说明格式为：

```
class <派生类名> : virtual <继承方式> <基类名>{ ... };
```

其中，关键字 virtual 与继承方式的位置无关，但必须位于虚基类名之前，且 virtual 只对紧随其后的一个基类名起作用。关键字 virtual 告诉编译器，该基类无论经过多少次派生，在其派生类中只能有该基类的一个成员。该基类就是这个派生类的虚基类。虚基类的所有直接和间接的派生类的构造函数中都要对这个虚基类进行直接初始化，而所有的间接初始化都被取消。

例如：

```
class A{
public:
    int a;
};
class B1 : virtual public A{};    //加 virtual
class B2 : public virtual A{};   //加 virtual
class C : public B1, public B2{};
void main(void){
    C obj;
    obj.a=1;                      //D
}
```

重新编译以上程序时，D 行不产生二义性。在派生类 C 中只有基类 A 的一个拷贝，即派生类对象 obj 中只有类 A 的一个子对象。注意，上面对类 A 的两个派生关系都加 virtual，缺一不可。而类 C 的继承关系就不需要再加 virtual，这是因为虚基类可继承，即 B1 和 B2 所有直接或间接的派生类都将类 A 作为虚基类。

当类 C 实例化创建一个对象 obj 时，它包含了 B1 的一个子对象和 B2 的一个子对象，也包含了基类 A 的一个子对象。如图 11.8 所示。

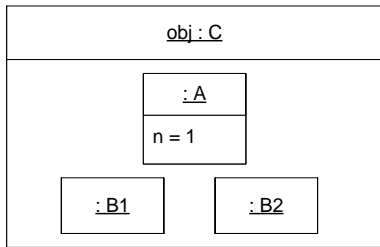


图 11-8 一个派生类对象包含其基类的一个子对象

对于一个虚基类，其派生类的一个对象中应该只包含其虚基类的一个子对象。所以在派生类创建一个对象时，为保证虚基类只被初始化一次，**虚基类的构造函数只能被调用一次**。例如，上面例子中创建 C 的一个对象，就要分别创建 B1 和 B2 的两个子对象，而这两个子对象不是各自创建虚基类 A 的一个子对象，而只有虚基类 A 的一个子对象。那么这个子对象由哪个类来创建？不是 B1 也不是 B2，而是 C。正是类 C 实例化创建 A 的子对象。

虽然继承结构的层次可能很深，但要实例化的类只是继承结构中的一个类。我们把实例化指定的类称为**当前派生类**。例如上面的类 C。**虚基类子对象由当前派生类的构造函数通过调用虚基类的构造函数进行初始化**。因此，当前派生类的构造函数的基类/成员初始化列表中必须列出对虚基类某个构造函数的调用；如果未列出，就隐含调用该虚基类的缺省构造函数。因此上面，类 B1、B2、C 提供的缺省构造函数中都调用了虚基类 A 的缺省构造函数。

由于当前派生类总是相对的，因此从虚基类直接或间接派生出的派生类中的构造函数的成员初始化列表中都要列出对虚基类构造函数的调用。**只有当前派生类的构造函数真正执行了虚基类的构造函数**，而其它调用被取消，从而保证对虚基类子对象只初始化一次。例如，上面例子中 B1 和 B2 的构造函数中调用虚基类 A 都没有执行，只有类 C 中调用执行了虚基类 A 的缺省构造函数。

在一个基类/成员初始化列表中如果同时出现对虚基类和非虚基类构造函数调用时，虚基类的构造函数就要先执行，然后再执行非虚基类的构造函数。

派生类不能从其虚基类中继承构造函数。



## 11.4.3 虚基类的例子

例 11-3 虚基类的派生类设计。类的继承结构如下图所示。编程如下：

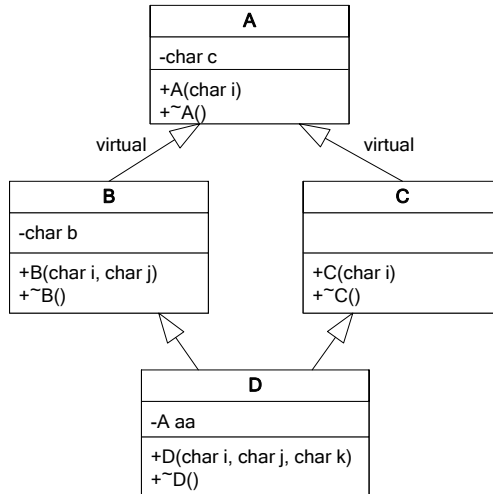


图 11-9 虚基类的派生类

```

#include <iostream>
using namespace std;
class A{
    char c;
public:
    A(char i) {c = i; cout<<"A constructor: i="<<i<<endl;}
    ~A() {cout<<"A destructor."<<endl;}
};
class B : virtual public A{
    char b;
public:
    B(char i, char j) : A(i)
        {cout<<"B constructor: i="<<i<<"j="<<j<<endl;}
    ~B() {cout<<"B destructor."<<endl;}
};
class C : virtual public A{
public:
    C(char i) : A(i)
        {cout<<"C constructor: i="<<i<<endl;}
    ~C() {cout<<"C destructor."<<endl;}
};
class D : public B,public C{
    A aa;
public:
    D(char i, char j, char k)
        : C(j), B(i,j), A(i), aa(k) //A 虚基类初始化
        {cout<<"D constructor."<<endl;}
    ~D() {cout<<"D destructor."<<endl;}
};
void test() {
    D obj('a', 'b', 'c');
}
  
```

```

}
int main() {
    test();
    system("pause");
    return 0;
}

```

执行程序，输出如下：

```

A constructor: i=a
B constructor: i=a;j=b
C constructor: i=b
A constructor: i=c
D constructor.
D destructor.
A destructor.
C destructor.
B destructor.
A destructor.

```

由于类 A 是 B 和 C 的虚基类，而类 D 是 B 和 C 的派生类，因此 A 也是 D 的虚基类，D 的构造函数中也要对 A 进行初始化。此时类 A 没有缺省构造函数，这要求类 A 的每个派生类的每个构造函数的基类/成员初始化列表中要显式说明对 A 类的初始化。

类 D 的一个对象中不仅包含类 A 的一个子对象，还包含类 A 的一个成员对象 aa。类 D 的一个对象的结构如图 11.10 所示。

在 main 函数中当创建 D 类对象 obj 时，就要调用其构造函数。由于 D 是一个派生类，应先调用执行虚基类 A 的构造函数、基类 B 和 C 的构造函数，然后再调用成员对象 aa 的构造函数，最后执行派生类 D 的构造函数体。

定义派生类 D 时两个直接基类 B 和 C 的定义顺序是先 B 后 C，而在类 D 的构造函数的成员初始化列表中，两个基类的顺序是先 C 后 B。此时两个直接基类的执行顺序应该是先 B 后 C，即执行基类构造函数的顺序取决于定义派生类时基类的顺序，而与派生类构造函数的成员初始化列表中给定的顺序无关。

类 B 是虚基类 A 的一个派生类。由于虚基类子对象在整个类结构中只初始化一次，这个实例已经由派生类 D 初始化，因此就不再执行类 B 对虚基类 A 的构造函数调用。类 C 与类 B 情况相同。成员对象 aa 是类 A 的对象，因此调用类 A 的构造函数对其进行初始化。

综合以上分析，建立 D 类对象时构造函数的执行顺序是：A、B、C、A、D。

当 main 函数结束时，对象 obj 的生存期结束，在撤消对象 obj 时自动调用析构函数。析构函数的执行顺序与构造函数的执行顺序严格相反。由于 D 是一个派生类，它要负责调用其基类和成员对象的析构函数。派生类析构函数的执行顺序是：先执行派生类的析构函数，再执行成员对象的析构函数，最后执行基类的析构函数，其顺序与该派生类对象建立时的顺序完全相反。因此析构函数的执行顺序是：D、A、C、B、A。

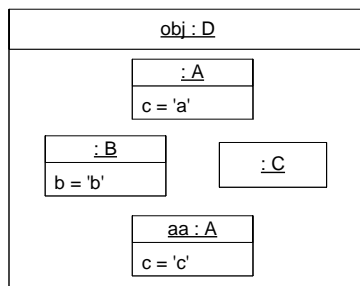


图 11-10 派生类的对象结构

如果将上面例子中的两个 `virtual` 去掉一个, 会发生什么情形? 假设去到 `B` 类的虚基类继承, 重新编译执行, 就会发现在类 `B` 构造函数执行之前, 类 `A` 的构造函数被执行了两次。当调用类 `D` 的构造函数时, 先执行了一次。而当调用类 `B` 的构造函数时, 再一次执行了类 `A` 的构造函数。此时类 `D` 的一个对象结构变化如图 11.11 所示。类 `B` 子对象中包含了其基类 `A` 的子对象, 最终导致类 `D` 的对象 `obj` 中包含了类 `A` 的两个值, 导致语义错误。

总之, 多继承虽然强大, 但容易导致二义性。虽然作用域运算符和虚基类能解决部分问题, 但代价是要修改继承方式, 而且要修改虚基类的所有间接派生类的所有构造函数, 如果虚基类没有缺省构造函数的话。建议在设计中尽量避免多继承(后来的 `Java` 语言干脆禁止了类的多继承)。

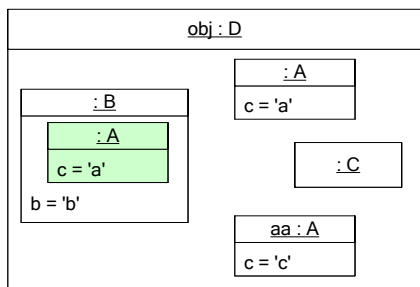


图 11-11 派生类对象的结构变化

## 11.5 子类型关系

多态性是面向对象编程的特征之一。多态性的基本含义是一个名字有多种具体解释。`C++` 提供了两种多态性: 编译时刻的静态的多态性和运行时刻的动态的多态性。静态多态性是通过重载(`overload`)函数或运算符重载函数来实现的。运算符重载将在后面介绍。动态多态性与继承性密切相关, 有两个方面: 子类型关系所实现的类型多态性和虚函数所实现的行为多态性。下面讨论子类型关系和类型多态性。

子类型(`subtype`)关系的正规定义如下:

一个特定类型 `S`, 当且仅当它提供了类型 `T` 的行为时, 就称类型 `S` 是类型 `T` 的一个子类型。这里所说的行为是指对类的公有成员的访问方式。

如果派生类 `S` 公有继承其基类 `T`, 那么 `S` 就是类型 `T` 的一个子类型。这是因为通过公有继承, 派生类就继承了其基类中除构造函数、析构函数之外的所有成员, 而且继承而来的成员的可见性与基类定义完全相同。因此派生类的一个对象就可作为基类的一个对象来用。

**子类型关系具有传递性, 而且不可逆。**例如, `Derived` 是 `Base` 一个子类型, 而 `DDerived` 又是 `Derived` 的一个子类型, 那么 `DDerived` 也是 `Base` 的一个子类型。

**类型多态性的含义是派生类的一个对象不仅属于派生类, 而且属于其所有直接或间接的基类。**简言之, 一个对象具有多种类型的形态。这样就建立了一种可替代性, 使得在需要某个基类对象的地方都可用公有派生类的对象来替代。

具体来说, 子类型关系使派生类的一个对象可作为基类的一个对象来使用, 有以下 3 种具体情形。

(1) **派生类对象可赋值给基类对象**, 把派生类对象中的基类子对象的成员逐个赋给基类对象。这种赋值可能出现在说明语句、赋值语句、函数调用、函数返回语句中。例如:

```
Derived d;
```

```
Base b = d;           //调用了 Base 类的拷贝构造函数
```

对象 `b` 只包含了 `Base` 类中定义的成员(即基类子对象), 而不包含派生类扩展的成员。

一个对象能赋给其基类对象，但基类对象不能赋给一个派生类对象。例如：`d = b` 是不允许的，除非在 `Derived` 类中定义了拷贝构造函数或者赋值操作函数，其形参类型为“`const Base &`”。

尽管派生类对象可以赋给基类对象，但实际上很少如此使用。常见的是下面两种情形。

(2) 派生类的对象可赋值给基类的引用，即基类的引用可引用派生类的对象。例如：

```
Base &ref = d;
```

此时 `ref` 作为基类的引用，通过该引用只能访问对象 `d` 的基类成员的名字，而不能访问派生类扩展的成员名字。

**通过基类的引用来调用一个非虚成员函数，被执行的就是基类的成员函数。**

基类的引用经常作为函数的形参，使其任意一个派生类的对象都可以作为实参。

基类的引用也可作为函数的返回值，使函数体中可以返回其任意一个派生类的一个对象。

基类的引用可通过强制类型转换到其派生类的引用，此时转换的正确性取决于被引用的对象是否是指定的派生类。

(3) 派生类对象地址可赋值给基类指针，即基类的指针可指向派生类的对象。例如：

```
Base *pb = &d;
```

此时 `pb` 作为基类的指针，通过该指针只能访问对象 `d` 的基类成员的名字，而不能访问派生类扩展的成员名字。

**通过基类的指针来调用一个非虚成员函数，被执行的就是基类的成员函数。**

基类的指针经常作为函数的形参，使其任意一个派生类的对象地址都可以作为实参。

基类的指针也可作为函数的返回值，使函数体中可以返回其任意一个派生类的一个对象的地址。

基类的指针要通过强制类型转换才能赋给其派生类的指针，此时转换的正确性取决于被指定的对象是否是指定的派生类。如果不是，将发生不可预料的运行错误。

例 11-4 在子类型关系中调用非虚成员函数。下面有 4 个类，继承关系如图 11.12 所示，每个类中都定义了一个非虚成员函数 `who`。编程如下：

```
#include <iostream>
using namespace std;
class AbsBase{
public:
    void who() { cout<<"AbsBase."<<endl; }
};
class Base : public AbsBase{
public:
    void who() { cout<<"Base."<<endl; }
};
class Derived1 : public Base{
public:
    void who()
    { cout<<"Derived1."<<endl; }
};
class Derived2 : public Base{
public:
    void who() { cout<<"Derived2."<<endl; }
};
void f(AbsBase &ref){
```

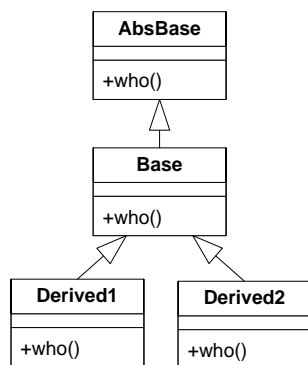


图 11-12 子类型结构中的

```

        ref.who();
    }
    void f(Base *pb) {
        pb->who();
    }
    int main() {
        Base obj1;
        Derived1 obj2;
        Derived2 obj3;
        f(obj1);
        f(obj2);
        f(obj3);
        f(&obj1);
        f(&obj2);
        f(&obj3);
        Derived1 *pd = &obj2;
        pd->who();
        f(pd);
        Base *pb = pd;
        pb->who();
        Derived1 *pd2 = (Derived1 *)pb;           //A
        pd2->who();                                //B
        Derived2 *pd3 = (Derived2 *)pd2;          //C
        pd3->who();                                //D
        system("pause");
        return 0;
    }

```

执行程序，输出如下：

```

AbsBase.
AbsBase.
AbsBase.
Base.
Base.
Base.
Derived1.
Base.
Base.
Derived1.
Derived2.

```

主函数中分别从 **Base**、**Derived1** 和 **Derived2** 类创建了 3 个对象。用这 3 个对象作为实参来调用 **f(AbsBase &ref)** 函数，这 3 个对象都属于 **AbsBase** 类，因此调用合法。在函数中通过基类的引用来调用非虚成员函数 **who**，就执行了基类 **AbsBase** 类中定义的函数。然后再用这 3 个对象的地址作为实参来调用函数 **f(Base \*pb)**。同理，调用合法而且执行了 **Base** 类中定义的函数。通过基类的指针来调用非虚成员函数 **who**，就执行了基类 **Base** 中定义的函数。

**obj2** 对象是由 **Derived1** 类创建而来，因此该对象的实际类型就是 **Derived1** 类。下面对 **obj2** 对象进行操作，先用实际类型 **Derived1** 的指针来调用 **who** 函数，**Derived1** 类中定义的 **who** 函数执行，而用该指针来调用 **f(Base \*pb)** 时，却执行了 **Base** 类中定义的函数。由此可知，对于同一个对象，既可以用基类指针来操作，也可以用自己的实际类型的指针来操作。当用基类指针来调用非虚函数时，执行的是基类定义的函数，而用派生类指针来调用时，执行的是派生类定义的函数。

A 行将基类 **Base** 的指针 **pb** 强制转换为派生类 **Derived1** 的指针 **pd2**，使 **pd2** 也指向对象

obj2, 然后 B 行通过该指针调用函数 who, 可以看到 Derived1 类的函数执行。A 行的强制类型转换是正确的, 因为 pb 指针确实指向 Derived1 类的对象。

C 行将 pd2 指针强制转换为派生类 Derived2 的指针 pd3, 使 pd3 也指向对象 obj2。此时被指向的对象仍然是 Derived1 类的对象, 但却被 Derived2 的指针所指, 这样就存在潜在错误。然后 D 行通过 pb3 指针调用执行了 Derived2 类中定义的函数 who。显然, 最后这两行在语义上是错误的, 但并没有出现运行错误提示。

**一个对象具有唯一的实际类型**, 就是被直接实例化的哪个类。例如上面对象 obj2 的实际类型为 Derived1。一个对象在被访问时具有多种使用类型(即所有直接和间接的基类)和多种方式(通过指针或引用)。

## 11.6 虚函数

行为多态性是指一个函数调用具有多种具体的执行方式。行为多态性是通过类的继承性和虚函数来实现的。

### 11.6.1 定义和使用

**虚函数就是用 virtual 关键字修饰的成员函数**。一个类中的虚函数可以在其派生类中重新定义, 这就是改写或覆盖(override, 注意区别于 overload 重载)。所谓改写就是派生类中对其基类中的虚函数按相同的基调(函数名和形参表)和返回值, 进行重新定义。当通过基类的引用或指针来调用虚函数时, 实际执行的将是派生类改写后的虚函数, 而不是基类中定义的虚函数。

基于函数的行为多态性需要 3 个条件:

- (1) 基类中定义了虚函数;
- (2) 派生类改写了虚函数;
- (3) 以基类的指针或引用来调用虚函数, 且作用于派生类对象。

对于一个基类, 大部分成员函数都应修饰为虚函数。派生类继承了基类中的虚函数。如果发现继承而来的虚函数不能满足自己的特殊需求, 就可以改写虚函数, 自己提供一个具体实现, 来取代基类的实现, 而基类对虚函数的调用方式和操作语义不变。

派生类改写后的函数即便不加 virtual 修饰, 它也是虚函数。

当派生类改写虚函数时, 不仅要保持函数的基调和返回值一致, 还应保持虚函数的语义不变。

当通过对象的指针或引用(称为使用类型, 往往是基类)来调用一个虚函数时, 对象的实际类型(往往是派生类)决定了被执行的函数, 而不是指针或引用的说明类型决定被执行的函数。

例 11-5 虚函数的定义和使用。与上面例 11-4 一样, 只是将最上层基类 AbsBase 中的 who 函数添加一个 virtual 修饰, 这样使其 3 个派生类中的 who 成员函数都成为改写后的虚函数。执行结果完全不同。

```
class AbsBase{
public:
    virtual void who() { cout<<"AbsBase."<<endl; }    //虚函数
```

```
};
```

执行程序，输出如下：

```
Base.
Derived1.
Derived2.
Base.
Derived1.
Derived2.
Derived1.
Derived1.
Derived1.
Derived1.
Derived1.
```

A 行开始的 3 行调用了函数 `f(AbsBase&ref)`，函数中通过基类 `AbsBase` 的引用来调用虚函数 `who`，而实际执行的是这 3 个对象的实际类型各自改写的虚函数，而不是基类的函数。

B 行开始的 3 行调用了函数 `f(Base *pb)`，函数中通过基类 `Base` 的指针调用虚函数 `who`，而实际执行的是这 3 个对象的实际类型各自改写的虚函数，而不是基类的函数。

C 行开始针对 `obj2` 对象进行操作，它的实际类型是 `Derived1`，无论是通过基类的指针，还是强制类型转换后的派生类指针，调用虚函数 `who`，执行的都是实际类型 `Derived1` 中的虚函数，即便 D 行和 E 行有语义错误。

关于虚函数，说明以下几点。

(1) 虚函数不能是静态成员函数或友元函数。

(2) 基类的虚函数不一定在派生类中都改写。如未被改写，执行的仍然是基类中的虚函数。

(3) 构造函数不能修饰为虚函数，但析构函数应作为虚函数。

### 11.6.2 成员函数中调用虚函数

在成员函数中可以直接调用本类中的虚函数。但应注意，直接调用成员函数往往省略“`this->`”，也就是说，直接调用虚函数实际上就是通过当前对象的指针 `this` 来调用的。此时如果虚函数被派生类改写，而且调用虚函数作用于一个派生类对象，那么执行的就是改写后的虚函数。

例 11-6 成员函数中调用虚函数。两个类和两个对象之间的关系如图 11.13 所示。编程如下：

```
#include <iostream>
using namespace std;
class A{
public:
    virtual void fun1() { cout <<"A::fun1"<<"\t"; fun2(); }
    void fun2() { cout <<"A::fun2"<<"\t"; fun3(); }
    virtual void fun3() { cout <<"A::fun3"<<"\t"; fun4(); }
    virtual void fun4() { cout <<"A::fun4"<<"\t"; fun5(); }
```

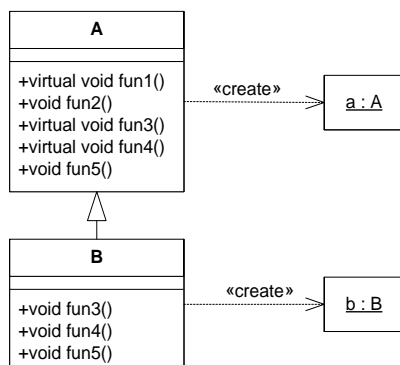


图 11-13 成员函数中调用虚函数的例子

```

    void fun5() { cout<<"A::fun5"<<'\n'; }
};
class B:public A{
public:
    void fun3() { cout <<"B::fun3"<<'\t'; fun4(); }
    void fun4() { cout <<"B::fun4"<<'\t'; fun5(); }
    void fun5() { cout <<"B::fun5"<<'\n'; }
};
int main(void){
    A a;
    a.fun1(); //A 输出第 1 行
    B b;
    b.fun1(); //B 输出第 2 行
    system("pause");
    return 0;
}

```

执行程序，输出如下：

```

A::fun1 A::fun2 A::fun3 A::fun4 A::fun5
A::fun1 A::fun2 B::fun3 B::fun4 B::fun5

```

基类中的 `fun1()` 函数是虚函数，但没有被派生类 `B` 改写。派生类 `B` 中改写了 `fun3()` 和 `fun4()` 这两个虚函数。基类中的 `fun5()` 不是虚函数，那么派生类中的 `fun5()` 只是用支配规则隐藏了基类中的同名函数。注意类中调用成员函数都省略了 “`this->`”。

`A` 行执行的输出结果容易理解，它与派生类 `B` 无关，执行的都是基类的函数。

`B` 行中 `b.fun1` 先调用执行了基类中的 `fun1` 函数，`fun1` 中又调用执行类 `fun2`。在基类 `A` 中的 `fun2` 函数中调用 `fun3`，实际上是 “`this->fun3()`”，此时当前对象是派生类 `B` 的对象 `b`，因此派生类中改写的 `fun3` 函数执行，而不是基类 `A` 中的 `fun3`，输出 “`B::fun3`”。同样原因，再调用派生类 `B` 中的虚函数 `fun4`，输出 “`B::fun4`”。最后调用执行了派生类中的 `fun5` 函数，输出 “`B::fun5`”。

基类中的虚函数即使是私有的，而派生类也能改写为公有成员。上面例子中，将 `fun3` 和 `fun4` 这两个虚函数改为基类 `A` 的私有成员，执行结果仍然相同。这说明私有虚函数也能被派生类改写，但通常虚函数需要被类外程序调用，多为公有。

### 11.6.3 构造函数中调用虚函数

在构造函数中可以直接调用本类中的虚函数。尽管派生类可以改写虚函数，而且创建的是派生类的对象，也不会执行改写后的虚函数，而只能执行本类自己的虚函数，这是由于构造函数的特殊性。

例 11-7 构造函数中调用虚函数，此时多态性不起作用。

```

#include <iostream>
using namespace std;
class A{
public:
    virtual void fun() {cout <<"A::fun"<<'\t'; } //虚函数
    A(){ fun(); }
};
class B:public A{
public:
    B() { fun(); }
}

```



```

        void fun(){ cout<<"B::fun"<<'\\t'; }           //改写虚函数
        void g(){ fun(); }
    };
    class C:public B{
    public:
        C() { fun(); }
        void fun(){ cout<<"C::fun"<<'\\n'; }           //改写虚函数
    };
    int main(){
        C c;                                           //A
        c.g();                                         //B
        system("pause");
        return 0;
    }

```

执行程序，输出如下：

```

A::fun B::fun C::fun
C::fun

```

A 行创建派生类对象 c 时，基类 A 的构造函数中调用虚函数 fun()，执行的是基类 A 的虚函数 fun，输出“A::fun”。然后执行 B 类的构造函数，再执行 B 类的虚函数 fun，输出“B::fun”；最后调用 C 类的构造函数，输出“C::fun”。这是第一行的输出。

B 行对对象 c 调用函数 g()，将执行 B 类中的 g() 函数，其中再调用虚函数 fun，此时当前对象是派生类 C 的对象，因此执行派生类 C 中的虚函数 fun()，输出第二行。

在析构函数中调用被改写的虚函数，同样多态性不起作用。

#### 11.6.4 虚析构函数

当撤销一个派生类对象时，应该先执行派生类的析构函数，再执行基类的析构函数。但如果用“delete 基类指针;”来撤销一个派生类对象时，就只执行基类的析构函数，而不执行派生类的析构函数，除非将基类的析构函数说明为虚函数。

例 11-8 虚析构函数。

```

#include <iostream.h>
class A{
public:
    ~A(){cout<<"destructor A"<<endl;}
};
class B:public A{
public:
    ~B(){cout<<"destructor B"<<endl;}
};
void main(){
    A *pa = new B;
    delete pa;                                     //A
    B *pb = new B;
    delete pb;
}

```

执行程序，输出如下：

```

destructor A
destructor B
destructor A

```

A 行 `delete pa` 中 `pa` 是基类 A 指针，被撤销的对象则是派生类 B 的对象，此时只执行了基类 A 的析构函数，输出了第 1 行，而派生类的构造函数却没有执行，这样就没有完整地撤销对象。为了纠正这种错误，要将基类 A 的析构函数说明为虚函数，其派生类可以不说明虚析构函数。一般来说，如果一个类将来要作为基类，那么其析构函数应说明为虚函数。

对于一个类，哪些成员函数应该说明为虚函数？一个成员函数完成一项功能或提供一项服务。一项功能或服务包括两方面：一个行为规范和多种可能的具体实现。

- 行为规范确定了该函数的名字、形参、返回值的形式和语义。行为规范作为类的接口，提供给类外程序，使类外程序能通过引用或指针调用该函数来提供服务。类外程序无需知晓被调用的函数内部如何实现。
- 一种实现作为一种方案，是一个具体的过程描述，表现为函数体内的一组语句序列。当改变具体实现方案时，不应影响到类外程序的函数调用。

对于一个函数，如果其规范可能隐含多种具体实现，而不是唯一实现，这个函数就应说明为虚函数。反之，如果一个函数的实现是唯一确定的，只希望被派生类继承，而不希望被改写，那么此函数就不能说明为虚函数。

按惯例，如果从多个已有类中提取一个类作为基类，那么此类中的多数成员函数应说明为虚函数。

虚函数的核心思想是将行为规范与具体实现分离，将函数调用与具体执行分离。规范是抽象的，由基类说明并提供缺省实现，而实现是具体的，派生类可继承也能改写。这样的好处是灵活性、适应性、可扩展性，而不失规范性。

### 11.6.5 纯虚函数与抽象类

在定义一个基类时，对于一个虚函数能确定其行为规范，但往往还不能提供一种具体实现，其具体实现完全依赖于派生类。这时可把这个虚函数定义为纯虚函数。纯虚函数就是在类中只提供行为规范而没有具体实现的虚函数。

定义纯虚函数的一般格式为：

```
virtual <返回类型> 函数名(<形参表>) = 0;
```

纯虚函数的函数名被赋值为 0，没有函数体。如果一个类含有纯虚函数，那么此类就称为抽象类(abstract class)。抽象类的特点是不能直接实例化创建对象，这是因为其中虚函数还没有实现。抽象类作为基类，其派生类有责任以改写(override)形式提供纯虚函数的具体实现，而且派生类的对象作为抽象类的对象。可直接实例化创建对象的类称为具体类(concrete class)。

抽象类可以定义指针或引用，通过这些指针或引用能调用纯虚函数。在编译时刻，并不确定到底执行的是哪个派生类所提供的实现。在运行时刻，由对象的实际类型来决定执行哪一个具体实现。

例 11-9 纯虚函数的定义和使用。修改前面一个例子，如图 11.14 所示。类 `AbsBase` 是一个抽象类，表

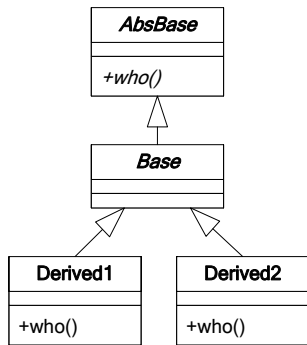


图 11-14 纯虚函数与抽象类

示为斜体字，其中的纯虚函数 *who* 也表示为斜体字。由于其派生类 **Base** 中没有提供这个纯虚函数的实现，因此该类仍是抽象类，仍不能创建对象。下面两个派生类分别提供了纯虚函数的实现，这两个派生类能创建对象。编程如下：

```
#include <iostream.h>
class AbsBase{
public:
    virtual void who() = 0; //纯虚函数
};
class Base : public AbsBase{ }; //仍是抽象类
class Derived1 : public Base{
public:
    void who() { cout<<"Derived1."<<endl; } //纯虚函数的实现
};
class Derived2 : public Base{
public:
    void who() { cout<<"Derived2."<<endl; } //纯虚函数的实现
};
void f(AbsBase &ref){ //通过抽象类的引用来调用纯虚函数
    ref.who();
}
void f(Base *pb){ //通过抽象类的指针来调用纯虚函数
    pb->who();
}
void main(void){
    Derived1 obj1;
    Derived2 obj2;
    f(obj1);
    f(obj2);
    f(&obj1);
    f(&obj2);
    Derived1 *pd = &obj1;
    pd->who();
    Base *pb = pd;
    pb->who();
    Derived1 *pd2 = (Derived1 *)pb;
    pd2->who();
}
```

执行程序，输出如下：

```
Derived1.
Derived2.
Derived1.
Derived2.
Derived1.
Derived1.
Derived1.
```

如果主函数中要创建 **AbsBase** 或者 **Base** 类的对象，编译时将报错，因为它们都是抽象类。尽管抽象类自己不能创建对象，如果其派生类提供了所有纯虚函数的实现，派生类就能创建对象，而这些对象也是抽象类的对象，因此我们不能说抽象类就没有对象。

抽象类不能说明变量，也不能说明为其它类或结构的成员对象。

关键字 **abstract** 一般不用于修饰普通 C++ 类。

### 11.6.6 关键字 final

C11 标准引入了 **final** 关键字，限制虚函数的改写 **override** 和类的派生。

如果一个虚函数修饰为 **final**，那么派生类将不能再改写它，其行为完全固定。纯虚函数不能修饰为 **final**。例如：

```
class BaseClass{
    virtual void func() final {};                //final 虚函数
};
class DerivedClass : public BaseClass{
    void func() {};                            //错误
};
```

如果一个类修饰为 **final**，该类就不能再说明派生类，它所有的虚函数都不能被改写了。

抽象类不能修饰为 **final**。例如：

```
class BaseClass final{                          //final 类
    virtual void func(){};
};
class DerivedClass : public BaseClass{          //错误
    void func() {};
};
```

### 11.6.7 显式虚函数改写 override

派生类中如果明确要改写某个虚函数，应显式说明，就是对该函数添加 **override** 修饰，以强制编译器检查是否真的改写。如果基类中没有找到被改写的虚函数，或者该函数为 **final**，就报错。例如：

```
class Base {
public:
    virtual void f1(float) {}
    virtual void f1(double)final {}

};

class Derived : public Base {
public:
    virtual void f1(int) override {}           //错误
    virtual void f1(double) override {}        //错误
    virtual void f1(float) override {}

};
```

### 11.6.8 抽象类设计实例

当我们设计较多的类时，就会发现某几个类之间存在相同的成员，而且这些成员具有相同的语义。此时就可以为这些类建立一个基类，并提取这些共同成员作为基类的成员，使这些派生类能继承基类的成员，而不再重复定义。继承结构往往就是这样建立起来的。

当我们在基类中描述成员函数时,就会发现一些函数的规范(名称、形参和返回值)和语义可以确定,而具体实现却不能确定。此时这些函数就可以修饰为纯虚函数,基类就成为抽象类。

例 11-10 我们前面介绍了在二维平面上如何用点 **Point** 来构造几何图形,如圆 **Circle**、三角形 **Triangle**、矩形 **Rectangle** 等。我们考虑这些图形之间有哪些共同属性或行为。它们都是闭合图形,具有确定的面积和周长。每个闭合图形都具有明确的位置,也能移动。我们建立一个抽象类 **ClosedShape** 作为所有闭合图形的抽象类,如图 11.15 所示。这样就可以抽象描述这些行为规范,并且能进行抽象编程。

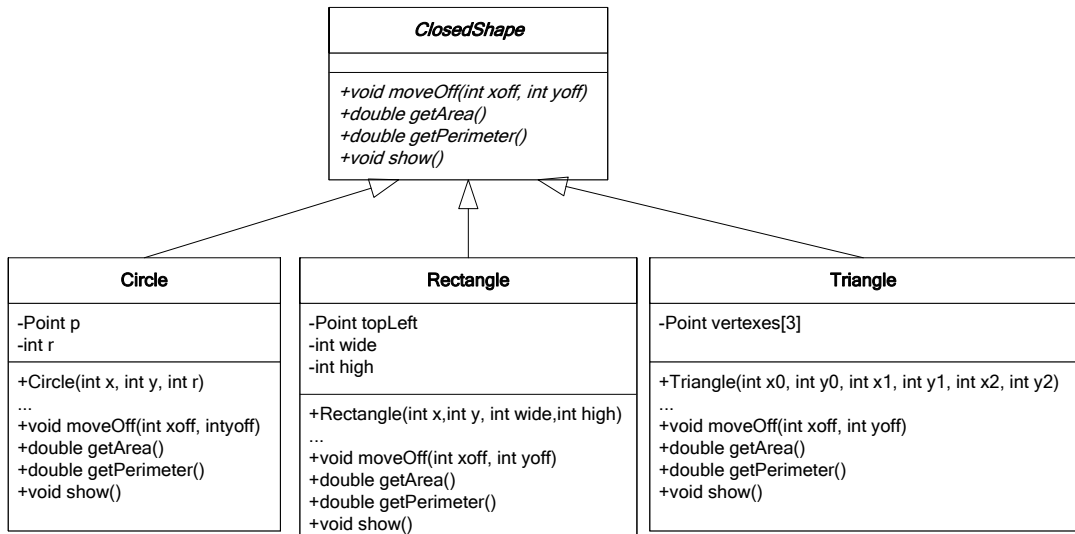


图 11-15 闭合图形类的继承结构

图中 **ClosedShape** 是一个抽象类,描述为斜体,其中 4 个纯虚函数也描述为斜体。其中:  
**moveOff** 函数表示图形相对移动到某个位置;  
**getArea** 函数表示计算面积;  
**getPerimeter** 函数表示计算周长;  
**show** 函数用来显示对象当前状态。对于任何一种闭合图形,这 4 个函数的计算规范都是一致的,只是 **ClosedShape** 类自身不能提供具体实现,而它的派生类能提供具体实现。**ClosedShape** 类的编程如下:

```

//ClosedShape.h
#ifndef CLOSEDSHAPE
#define CLOSEDSHAPE
class ClosedShape{
public:
    virtual double getArea() = 0;
    virtual double getPerimeter() = 0;
    virtual void moveOff(int xoff, int yoff) = 0;
    virtual void show() = 0;
};
#endif
  
```

在抽象类基础上就可以进行抽象编程。例如在一个图形编辑器中,往往要选择几个图形作为一个整体进行相对移动。可建立一个函数来完成这样的功能:

```

void moveOff(ClosedShape *selectedShapes[], int num, int xoff, int yoff){
  
```

```

    for (int i = 0; i < num; i++)
        if (selectedShapes[i] != NULL)
            selectedShapes[i]->moveOff(xoff, yoff);
}

```

这个函数对一个图形集合中的多个图形进行相对移动。第一个形参是 `ClosedShape` 指针的数组，确定了一个图形集合，该数组的元素可以指向任何已知的图形，如圆、矩形、三角形，也可以指向将来可能添加的图形，如多边形。第二个形参确定了集合中图形对象的个数。后面形参确定了相对移动的坐标。函数体中通过数组中的指针调用了纯虚函数 `moveOff`。

如果要显示一个图形集合中各图形的状态，可以建立一个函数如下：

```

void show(ClosedShape *selectedShapes[], int num){
    for (int i = 0; i < num; i++)
        if (selectedShapes[i] != NULL)
            selectedShapes[i]->show();
}

```

这个例子中的多个源文件之间的包含关系如下图所示。主函数和普通函数都在 `Main.cpp` 文件中。虽然这个例子由多个文件组成，但不需要多文件项目配置，`Main.cpp` 程序直接或间接地包含了全部文件，因此只要启动 `Main.cpp` 编译即可。

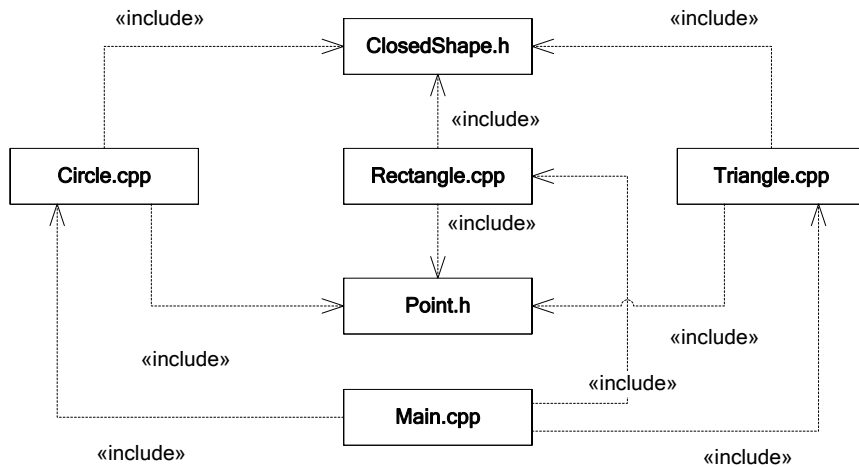


图 11-16 源文件之间的包含关系

下面给出各个源文件的设计。

```

//Point.h
#ifndef POINT
#define POINT
#include <iostream>
#include <math.h>
using namespace std;
class Point{
    int x, y;
public:
    Point(int x=0, int y=0){
        this->x = x;
        this->y = y;
    }
}

```

```

    int getX() {return x;}
    int getY() {return y;}
    void move(int a, int b){x = a; y = b;}
    void moveOff(int xoff, int yoff){x += xoff; y += yoff;}
    virtual void show(){
        cout<<"("<<getX()<<", "<<getY()<<")";
    }
    double distance(const Point &p){
        double xdiff, ydiff;
        xdiff = x - p.x;
        ydiff = y - p.y;
        return sqrt((xdiff*xdiff + ydiff*ydiff));
    }
};
#endif

```

下面是 **Circle** 类的设计。

```

//Circle.cpp
#include <iostream>
#include "Point.h"
#include "ClosedShape.h"
using namespace std;

class Circle : public ClosedShape{
    Point p;
    int r;
public:
    Circle(int x, int y, int r): p(x, y){
        this->r = r;
    }
    double getArea(){
        return r * r * 3.14;
    }
    double getPerimeter(){
        return r*2*3.14;
    }
    int getR(){return r;}
    void setR(int r){this->r = r;}
    void moveOff(int xoff, int yoff){p.moveOff(xoff, yoff);}
    void show(){
        cout<<"一个圆:";
        p.show();
        cout<<" r = "<<r<<endl;
        cout<<"周长为"<<getPerimeter();
        cout<<"面积为"<<getArea()<<endl;
    }
};

```

下面是 **Rectangle** 类的设计。

```

//rectangle.cpp
#include <iostream>
#include "Point.h"
#include "ClosedShape.h"
using namespace std;

class Rectangle : public ClosedShape{
    Point topLeft;
    int wide, high;
public:
    Rectangle(int x, int y, int wide, int high)

```

```

        :topLeft(x, y){
            this->wide = wide;
            this->high = high;
        }
    void moveOff(int xoff, int yoff){
        topLeft.moveOff(xoff, yoff);
    }
    void size(int wide, int high){
        this->wide = wide;
        this->high = high;
    }
    double getArea(){
        return double(wide * high);
    }
    double getPerimeter(){
        return double(wide + high) * 2;
    }
    void show(){
        cout<<"一个矩形:";
        topLeft.show();
        cout<<" 宽度为"<<wide<<" 高度为"<<high<<endl;
        cout<<"周长为"<<getPerimeter();
        cout<<"面积为"<<getArea()<<endl;
    }
};

```

下面是 **Triangle** 类的设计。

```

//Triangle.cpp
#include <iostream>
#include "Point.h"
#include "ClosedShape.h"
#include <math.h>
using namespace std;

class Triangle : public ClosedShape{
    Point vertexes[3];
public:
    Triangle(){}
    Triangle(int x0, int y0, int x1, int y1, int x2, int y2){
        vertexes[0].move(x0, y0);
        vertexes[1].move(x1, y1);
        vertexes[2].move(x2, y2);
    }
    Point &getVertex(int index){
        return vertexes[index];
    }
    void moveOff(int xoff, int yoff){
        vertexes[0].moveOff(xoff, yoff);
        vertexes[1].moveOff(xoff, yoff);
        vertexes[2].moveOff(xoff, yoff);
    }
    double getPerimeter(){
        double per = 0;
        per += vertexes[0].distance(vertexes[1]);
        per += vertexes[1].distance(vertexes[2]);
        per += vertexes[2].distance(vertexes[0]);
        return per;
    }
    double getArea(){
        double a = vertexes[0].distance(vertexes[1]);

```



```

        double b = vertexes[1].distance(vertexes[2]);
        double c = vertexes[2].distance(vertexes[0]);
        double s = (a + b + c) / 2;
        return sqrt(s*(s-a)*(s-b)*(s-c));
    }
    void show(){
        cout<<"一个三角形:";
        vertexes[0].show();
        vertexes[1].show();
        vertexes[2].show();
        cout<<endl;
        cout<<"周长为"<<getPerimeter();
        cout<<"面积为"<<getArea()<<endl;
    }
};

```

下面是 Main.cpp 源文件的内容。

```

//Main.cpp
#include "Circle.cpp"
#include "Rectangle.cpp"
#include "Triangle.cpp"
#include <iostream>
using namespace std;
/*
对 num 个 ClosedShape 对象统一进行相对移动
*/
void moveOff(ClosedShape *const selectedShapes[], int num, int xoff, int
yoff){
    for (int i = 0; i < num; i++)
        if (selectedShapes[i] != NULL)
            selectedShapes[i]->moveOff(xoff, yoff);
}
/*
重新显示 num 个 ClosedShape 对象的状态
*/
void show(ClosedShape *const selectedShapes[], int num){
    for (int i = 0; i < num; i++)
        if (selectedShapes[i] != NULL)
            selectedShapes[i]->show();
}
/*
对 num 个 ClosedShape 对象按各自面积进行升序排序
第一个形参保存了排序之后的各图形的指针
*/
void ascSortByArea(ClosedShape *selectedShapes[], int num){
    if (num < 2) return;
    //先计算并保存各个图形的当前面积
    double *areas = new double[num];    //申请动态内存
    for (int i = 0; i < num; i++)
        if (selectedShapes[i] != NULL)
            areas[i] = selectedShapes[i]->getArea();
    //再按面积进行冒泡升序排序
    ClosedShape *temp;
    double td;
    for(int i = 0; i < num-1; i++)
        for(int j = 0; j < num-i-1; j++)
            if(areas[j] > areas[j+1]){
                temp = selectedShapes[j+1]; //交换指针
                selectedShapes[j+1] = selectedShapes[j];
            }
    }
}

```

```

        selectedShapes[j] = temp;
        td = areas[j+1];           //交换面积
        areas[j+1] = areas[j];
        areas[j] = td;
    }
    delete []areas;               //回收
}
int main(){
    Rectangle r1(3,4,5,6);
    Circle c1(1,2,3);
    Rectangle r2(4,3,6,7);
    Triangle t1(1,2,3,4,4,1);
    ClosedShape *shapes[10];
    shapes[0] = &r1;
    shapes[1] = &c1;
    shapes[2] = &r2;
    shapes[3] = &t1;
    show(shapes,4);
    moveOff(shapes, 4, 1,2);
    cout<<"相对移动之后:"<<endl;
    show(shapes,4);
    ascSortByArea(shapes,4);
    cout<<"按面积升序排序:"<<endl;
    show(shapes,4);
    system("pause");
    return 0;
}

```

执行程序，输出如下：

```

一个矩形: (3, 4) 宽度为 5;高度为 6
周长为 22;面积为 30
一个圆: (1, 2) r = 3
周长为 18.84;面积为 28.26
一个矩形: (4, 3) 宽度为 6;高度为 7
周长为 26;面积为 42
一个三角形: (1, 2) (3, 4) (4, 1)
周长为 9.15298;面积为 4
相对移动之后:
一个矩形: (4, 6) 宽度为 5;高度为 6
周长为 22;面积为 30
一个圆: (2, 4) r = 3
周长为 18.84;面积为 28.26
一个矩形: (5, 5) 宽度为 6;高度为 7
周长为 26;面积为 42
一个三角形: (2, 4) (4, 6) (5, 3)
周长为 9.15298;面积为 4
按面积升序排序:
一个三角形: (2, 4) (4, 6) (5, 3)
周长为 9.15298;面积为 4
一个圆: (2, 4) r = 3
周长为 18.84;面积为 28.26
一个矩形: (4, 6) 宽度为 5;高度为 6
周长为 22;面积为 30
一个矩形: (5, 5) 宽度为 6;高度为 7

```

周长为 26;面积为 42

主函数中创建了 4 个图形,放入一个数组中,调用 show 函数打印各图形的状态,然后调用 moveOff 函数相对移动,最后按面积升序排序,再次打印各图形的状态。读者可自行测试。

## 11.7 继承性设计要点

继承性反映了类型之间的概念结构。这种概念结构可抽象为“is a”关系,即派生类“是一种”基类,或者派生类的一个对象“是基类的一个对象”。如果继承关系设计不当,就会导致概念表达的错误。由于继承性结构是公开的,因此在使用某个派生类时就一定要使用其基类,也就是说,当使用一个派生类是必须要知道其基类中提供了哪些公共的构造函数和成员,因为它们就是派生类对象的一部分。

我们经常遇到不恰当的继承性设计。假如已有一个类 Point 表示二维平面上的点,此时要建立一个 Circle 类来表示圆。一种设计将 Circle 作为 Point 的派生类,只要扩展一个半径就能表示一个圆。如此设计固然能使 Circle 类实现自己的功能(例如移动、改变大小、计算面积、周长等),但却表示了一个错误的概念。尽管一个圆需要一个点作为圆心,但一个圆毕竟不是一个点。换言之,当需要一个点时,我们不能提供一个圆来替代一个点。这种错误比较隐蔽,往往在建立其它类时要用到 Circle 类时才会发现。假如建立多边形的类 Polygon。一个多边形由 3 个以上的点按一定次序构成。如果 Circle 是 Point 的派生类,那么用 4 个圆就能用来构成一个四边形,这显然不合理。要纠正这种继承关系错误就要改变继承性关系,会导致派生类必须做很大改动,甚至要重新设计。

同理,把一个圆柱体作为一个对象,建立一个 Cylinder 类。我们不能将一个圆柱体作为一个圆,所以不能将 Cylinder 类作为 Circle 类的派生类。正确的设计应该是一个圆柱体“包含”有一个圆和一个高度作为其成员。

出现类似的继承性错误的原因是只考虑到派生类能继承基类的成员,而没有考虑替代性原则。换言之,只考虑到特征重用,而没有考虑概念表达的合理性。一个圆“有一个”圆心点,这应该是复合对象和成员对象之间的关系,而不应该是继承性关系。

类似的例子,考虑 Point 与 Rectangle 类(矩形)的关系。尽管一个矩形需要左上角的一个点,加上一个长度和一个宽度,但我们不能将 Rectangle 类作为 Point 类的一个派生类,这是因为一个矩形并不是一个点。合理的设计是将 Point 类的一个对象作为 Rectangle 类的一个复合对象。当我们使用一个复合对象类时,往往不用考虑封装在内部有哪些私有成员对象。

分析一个现实例子,一辆自行车有两个车轮,但自行车类并不能作为车轮类的派生类。如果将自行车类作为车轮类的派生类,那么在需要一个车轮时,而提供一辆车就满足要求,继承性允许这种替代。这显然有错误,而且这种错误不会出现在自行车类(派生类)的设计中,而是出现在使用继承关系时。

为什么继承关系如此敏感?当我们在使用一个派生类时,必须要知道其基类是什么,才能知道它继承了哪些公共成员可供调用,因此继承性关系必须是公开的,而且作为派生类设计的一部分。当我们在使用一个派生类时,并非简单地创建对象,再调用其公用成员函数,可能要将该对象作为基类对象来使用。不恰当的继承性就允许错误的使用。例如,如果将

Rectangle 类作为 Point 类的派生类,那么当需要  $n$  个点来构造一个多边形时,就允许提供  $n$  个矩形来替代。而这个错误只有当使用多边形类的构造函数时才会发现。此时你要纠正这个错误就会发现派生类(Rectangle)要彻底重新设计,而且所有使用该类的程序都可能要重新设计。

在建立继承性关系时,派生类要继承其基类的全部数据成员,而不是一部分。如果派生类继承了自己并不需要的数据成员,那么这个继承性关系就可能是错误的。

考虑手机类 MobilePhone、音乐手机类 MusicPhone 和 MP3Player 类之间的关系。MusicPhone 类表示能播放 MP3 音乐的手机,而 MP3Player 类表示 MP3 播放器。因为 MusicPhone 是一种手机 MobilePhone,可作为 MobilePhone 的派生类,就继承了数据成员,如电池种类、容量等。但如果将 MusicPhone 类也作为 MP3Player 类的派生类,试图继承 MP3 播放器的功能时,就会发现 MP3 播放器也有自己的电池种类、容量等数据成员。这样一部 MusicPhone 手机就会有两块电池,这显然不符合实际情况。即便你将 MP3Player 类作为 MusicPhone 的成员对象,也存在同样问题。

总之,继承性切不可滥用,在设计一个继承性时应同时关注以下三个要点:

- 概念表达:从使用者的角度来判断是否真正表达了派生类与基类之间的“is a”概念。如果派生类的某个对象不属于基类,或者有条件地属于基类,就不能建立继承性。
- 特征重用:基类中是否提供了合适的成员供其派生类继承。如果派生类只需要继承基类的部分成员,而不是全部成员,就不适合建立继承性。
- 抽象设计:动态多态性是抽象设计的基础,虚函数是关键,基类中是否有合适的纯虚函数表示行为规范,是否有合适的虚函数表示行为规范及缺省实现。如果基类中根本就没有虚函数,这往往表明该类在设计时就没打算作为基类被继承。

根据以上要点,当发现一个类要使用另一个类的数据成员时,首先判断是否有复合对象与成员对象之间的关系,而不是派生类与基类的关系。很多初学者过于强调特征重用,而忽略了概念表述的正确性。在继承性关系中,有一个简单方法来判断继承关系的合理性,判断派生类对象集合是否是基类对象的一个子集。如果不是或者不完全肯定,就不能建立继承性关系。但要注意,并非所有的子集关系都要建立继承性。例如,男生和女生分别是学生的子集,但往往不需要建立男生类作为学生类的派生类,而应在学生类中添加一个性别属性,取不同的值来区别男女即可。

## 11.8 小 结

- 封装性、继承性和多态性是面向对象编程的三大特征。本章主要介绍继承性和多态性。
- 继承性(inheritance)表示基类与派生类之间的关系。基类是派生类的抽象,而派生类是基类的具象。派生类的一个对象也是其基类的一个对象。派生类继承了基类中定义的成员,也可扩展新成员。
- 建立继承性关系的最基本原则是里氏替代原则:在需要一个基类对象的地方,而实际提供了一个派生类对象,应该是无条件满足要求的。如果有条件,那么这个继承

性关系就存在潜在问题。

- 如果一个派生类只有一个直接的基类，称为单继承。如果一个类同时有多个基类，则称为多继承或多重继承。
- 对于一个继承性关系，派生类持有这个关系，而基类自己并不知道什么类作为其派生类。所以要先定义基类，再定义其派生类，派生类单方向依赖其基类。
- 派生类中仅描述扩展的新成员，包括改写的虚函数。但派生类的一个对象的成员包括其所有直接和间接的基类的成员，以及派生类扩展的新成员。
- 尽管派生类继承了基类的所有成员，但仍不能直接访问基类的私有成员。
- 派生类对于一个基类要确定一种继承方式，有 `private`、`public`、`protected` 三种选择。继承方式确定了基类成员在派生类中的可见性。缺省为私有继承，但常用的却是公有继承。
- 一个派生类对象的数据成员由两部分构成：(1)来自直接和间接的基类(称为子对象)的成员；(2)自己类中定义的成员对象、基本类型及其数组的成员。因此创建派生类的一个对象需要对以上数据成员进行初始化。
- 派生类构造函数中的“基类/成员初始化列表”负责对基类子对象和成员对象的初始化，函数体负责对基本类型及其数组的成员的初始化。执行次序如下：
  - (1) 先调用各个直接基类的构造函数，调用顺序按照派生类中的说明顺序（自左向右）。注意这是一个递归的调用过程，从直接基类到间接基类，逐层向上调用。执行完成的次序正相反，最上层的基类先执行完，再到下一层。
  - (2) 再调用各成员对象的构造函数，调用顺序按照它们在派生类中的说明顺序（自上而下）。这也是一个递归调用过程，从直接成员类到间接成员类，逐层向内调用。执行完成的次序正相反，最内层的成员先执行完，再到外一层。
  - (3) 最后执行构造函数体，对基本类型成员初始化。
- 缺省提供的构造函数也按以上过程来创建对象，并要求被调用的所有类都要提供缺省构造函数。如果某个基类或成员类没有缺省构造函数，派生类就必须显式定义构造函数，并在基类/成员初始化列表中明确给出调用实参。
- 缺省提供的拷贝构造函数也按以上过程来创建对象，也同样在其基类/成员初始化列表中调用基类和成员类的拷贝构造函数，然后在函数体中复制基本类型成员。
- 缺省提供的赋值操作函数先调用基类的赋值操作函数来复制基类的成员，然后再复制自己的成员对象，最后再复制基本类型的成员。
- 派生类的析构函数的执行过程与构造过程严格相反。撤销派生类对象的过程如下：
  - (1) 先执行派生类自己的析构函数，对基本类型成员进行清理。
  - (2) 再调用各成员对象的析构函数，对各成员对象进行清理。这是一个递归调用过程，与构造过程次序相反。
  - (3) 最后调用各基类的析构函数，对各基类中的数据成员进行清理。这是一个递归调用过程，与构造过程次序相反。
- 派生类的析构函数体中并没有显式调用基类或成员类的析构函数，上述析构过程是系统自动调用执行的。
- 在派生类的函数中调用其基类的成员函数的格式：<基类名>::<成员函数名>(<实参

表>)

- 当用一个成员名字来访问某个成员时，如果不能确定究竟是哪一个，就出现二义性问题。二义性问题大多来自多继承关系中两个基类含有同名成员。
- 在多继承关系中，来自两个基类的同名成员可见性不同，但仍有二义性。来自两个基类的同名成员函数如果形参不同，也导致二义性。
- 使用作用域运算符::可以解决二义性问题，对类的成员进行限定的一般格式为：
  - <对象名>.<基类名>::<数据成员名>
  - <对象名>.<基类名>::<成员函数名>(<实参表>)

通过对象指针来访问成员：

- <对象指针> -> <基类名>::<数据成员名>
- <对象指针> -> <基类名>::<成员函数名>(<实参表>)
- 派生类中的成员与基类成员同名不会导致二义性，原因是派生类的作用域优先于其基类的作用域，这是支配规则确定的。
- 当一个派生类由多个基类派生，而这些基类又有一个共同的基类，就形成了“钻石继承结构”。当访问共同基类中的数据成员时，就会出现二义性。虽然作用域运算符可以消除语法错误，但这又违背继承性的基本原则：派生类的一个对象应该是其基类的一个对象，应持有基类数据成员的一个值。
- 用关键字 **virtual** 来修饰继承方式就定义了一个派生类的虚基类。虚基类的所有直接和间接的派生类的构造函数中都要对这个虚基类进行直接初始化，而所有的间接初始化都被取消。因此当前派生类的构造函数的基类/成员初始化列表中必须列出对虚基类某个构造函数的调用；如果未列出，就隐含着调用该虚基类的缺省构造函数。在一个基类/成员初始化列表中如果同时出现对虚基类和非虚基类构造函数调用时，虚基类的构造函数就要先执行，然后再执行非虚基类的构造函数。
- 多态性的基本含义是一个名字有多种具体解释。
- C++提供了两种多态性：编译时刻的静态的多态性和运行时刻的动态的多态性。
- 静态多态性是通过重载(overload)函数或运算符重载函数来实现的。运算符重载将在后面介绍。
- 动态多态性有两个方面：子类型关系所实现的对象类型的多态性和虚函数所实现的行为多态性。
- 对象类型的多态性是指派生类的一个对象不仅属于派生类，而且属于其所有直接或间接的基类，即一个对象具有多种类型形态。在需要某个基类对象的地方都可用派生类的对象来替代，即派生类的一个对象可作为一个基类对象来使用。
- 一个对象具有唯一的实际类型，就是被直接实例化的哪个类，但有多种使用类型，就是直接或间接的基类类型。
- 派生类对象可赋值给基类对象，反之不可以。
- 基类的引用可引用派生类的对象，通过该引用只能访问基类成员，而不能访问派生类扩展的成员。通过基类的引用来调用一个非虚成员函数，被执行的就是基类的成员函数。基类的引用可通过强制类型转换到其派生类的引用，此时转换的正确性取决于被引用的对象是否是指定的派生类。

- 基类的指针可指向派生类的对象，通过该指针只能访问基类成员，而不能访问派生类扩展的成员。通过基类的指针来调用一个非虚成员函数，被执行的就是基类的成员函数。基类的指针要通过强制类型转换才能赋给其派生类的指针，此时转换的正确性取决于被指定的对象是否是指定的派生类。如果不是的话，将发生不可预料的运行错误。
- 行为多态性是指一个函数调用具有多种具体执行。行为多态性是通过类的继承性、虚函数、指针或引用来实现的。
- 虚函数就是用 `virtual` 关键字修饰的成员函数。一个类中的虚函数可以在其派生类中重新定义，这就是改写(override)。所谓改写就是派生类中对其基类中的虚函数按相同的基调(函数名和形参表)和返回值，进行重新定义。当通过基类的引用或指针来调用一个虚函数作用于一个派生类对象时，实际执行的是派生类改写后的虚函数，而不是基类定义的虚函数。此时对象的实际类型决定了被执行的函数，只有在运行时刻才能知道对象的实际类型，因此这是动态的多态性。
- 派生类继承了基类中的虚函数，如果发现继承而来的虚函数不能满足自己的特殊需求，就可以改写虚函数，自己提供一个具体实现，来取代基类的实现，而基类对虚函数的调用方式和操作语义不变。
- 派生类改写后的函数即便不加 `virtual` 修饰，它也是虚函数。
- 当通过对象的指针或引用(称为使用类型，往往是基类)来调用一个虚函数时，对象的实际类型(往往是派生类)决定了被执行的函数，而不是指针或引用的说明类型。
- 虚函数不能是静态成员函数，也不能是友元函数。
- 构造函数不能修饰为虚函数，但析构函数可作为虚函数。
- 在成员函数中可以直接调用本类中的虚函数。如果虚函数被派生类改写，而且调用虚函数作用于一个派生类对象，那么执行的就是改写后的虚函数。
- 在构造函数中可以直接调用本类中的虚函数，但改写后的虚函数不会执行。
- 如果通过一个基类指针来撤销一个派生类对象时，就只执行基类的析构函数，而不执行派生类的析构函数，除非将基类的析构函数说明为虚函数。基类的析构函数往往说明为虚函数。
- 一个虚函数完成一项功能或提供一项服务。一项功能或服务包括两方面：一个行为规范和多种可能的具体实现。
  - 行为规范确定了该函数的名字、形参、返回值的形式和语义。行为规范作为类的接口，提供给类外程序，使类外程序能通过引用或指针调用该函数来提供服务。类外程序无需知晓被调用的函数内部如何实现。
  - 一种实现方案是一个具体的过程描述，表现为函数体内的一组语句序列。当改变具体实现方案时，不应影响到类外程序的函数调用。
- 对于一个函数，如果其规范可能有多种具体实现方案，而不是唯一实现，这个函数就应说明为虚函数。反之，如果一个函数的实现是唯一确定的，只希望被派生类继承，而不希望被改写，那么此函数就不能说明为虚函数。如果从多个已有类中提取一个类作为基类，那么基类中的多数成员函数应说明为虚函数。
- 虚函数将行为规范与实现方案分离，将函数调用与具体执行分离。规范是抽象的，

由基类说明并提供一个缺省实现，而实现方案是具体的，派生类可以继承也能改写。这样的好处是灵活性、适应性、可扩展性，而不失规范性。

- 纯虚函数就是只提供行为规范而没有具体实现的虚函数。
- 如果一个类含有纯虚函数，那么此类就称为抽象类(abstract class)。抽象类的特点是不能直接实例化创建对象，这是因为其中虚函数还没有实现。抽象类作为基类，其派生类有责任以改写形式提供纯虚函数的具体实现，而且派生类的对象就是抽象类的对象。可以直接实例化创建对象的类就是具体类(concrete class)。
- 抽象类可以定义指针或引用，通过这些指针或引用能调用纯虚函数。在编译时刻，并不确定到底执行的是哪个派生类所提供的实现。在运行时刻，由对象的实际类型来决定执行哪一个具体实现。
- 对于一个继承性设计，应关注以下三个要素：
  - 概念表述：从使用者的角度来判断是否正确表示了“is a”概念。
  - 特征重用：派生类继承基类成员，基类中是否有合适的成员供派生类重用。
  - 抽象设计：动态多态性是抽象设计的基础，虚函数是关键，基类中是否有合适的纯虚函数表示行为规范，是否有合适的虚函数表示行为规范及缺省实现。

## 11.9 练 习 题

1. 下面哪一个特征不是面向对象编程的特征？  
A 封装性    B 一致性    C 继承性    D 多态性
2. 关于派生类与基类之间的关系，下面哪一种说法是错误的？  
A 基类表示比较抽象的、一般性的、较大范畴的对象，而派生类表示比较具体的、特殊性的、较小范畴的对象。  
B 派生类的一个对象也是其基类的一个对象，这是无条件的。  
C 派生类创建一个对象，那么该派生类的所有直接或间接的基类也要实例化。  
D 越具体的派生类包含越少的属性。
3. 对于继承方式和可见性，下面哪一种说法是错误的？  
A 继承方式有 3 种选择：私有、公有和保护，继承方式决定了基类成员在派生类中的可见性。  
B 缺省为私有继承，但常用的却是公有继承。  
C 无论哪一种继承方式，派生类中都不能访问其基类中的私有成员，但能访问基类中的保护成员和公有成员。  
D 在公有继承方式中，派生类没有继承基类的私有成员。
4. 对于派生类对象的构建过程，下面哪一种说法是错误的？  
A 先调用各个直接基类的构造函数，对基类子对象进行初始化，然后再调用各个成员对象类的构造函数，对成员对象进行初始化，最后再执行构造函数体，对基本类型成员进行初始化。  
B 对基类子对象和成员对象的初始化都是递归调用。  
C 缺省提供的构造函数和拷贝构造函数都是按以上次序完成初始化。  
D 如果基类没有提供缺省构造函数，那么派生类也不能定义缺省构造函数。



5. 下列哪一种情形不存在二义性?

- A 派生类中定义的成员函数与其基类中定义的成员同名同参。
- B 在多继承关系中, 两个基类定义了同名的数据成员, 但一个私有的, 一个公有的。
- C 在多继承关系中, 两个基类定义了同名的成员函数, 但形参不同。
- D 一个派生类有多个基类, 而这些基类又有一个共同的基类, 在此基类中定义了数据成员。

6 在下列程序中, 判断语句①~⑤是否有二义性。

```
class Base1{
public:
    int x;
    int fun1();
    int fun2();
    int fun2(int);
    int fun3();
};
class Base2{
    int x;
    int fun1();
public:
    char fun2();
    int fun3();
};
class Derived:public Base1,public Base2{};
void main(){
    Derived obj;
    Derived *ptr = &obj;
    ptr->x=1;                //①
    ptr->fun1();              //②
    ptr->fun2();              //③
    ptr->fun2(10);            //④
    ptr->fun3();              //⑤
}
```

7. 写出以下程序的运行结果。

```
#include <iostream.h>
class Data{
    int x;
public:
    Data(int x) {
        Data::x=x;
        cout<<"Data cons."<<endl;
    }
    ~Data() { cout<<"Data des."<<endl; }
};
class Base{
    Data d1;
public:
    Base(int x):d1(x) { cout<<"Base cons."<<endl; }
    ~Base() { cout<<"Base des."<<endl; }
};
class Derived:public Base{
    Data d2;
public:
    Derived(int x):Base(x),d2(x) { cout<<"Derived cons."<<endl; }
    ~Derived() { cout<<"Derived des."<<endl; }
};
void main()
```

```
{ Derived obj(5); }
```

8. 写出以下程序的运行结果。

```
#include <iostream.h>
class A{
    int ax;
public:
    A(int x=10){ax=x;cout<<"调用构造函数 A"<<ax<<' ';}
    void f(){cout<<ax<<' ';}
    virtual ~A(){cout<<"调用析构函数 A"<<' ';}
};

class B:virtual public A{
    int bx;
public:
    B(int x):A(20){bx=x;cout<<"调用构造函数 B"<<bx<<' ';}
    void f(){cout<<bx<<' ';}
    ~B(){cout<<"调用析构函数 B"<<' ';}
};

class C:virtual public A{
    int cx;
public:
    C(int x):A(30){cx=x;cout<<"调用构造函数 C"<<cx<<' ';}
    void f(){cout<<cx<<' ';}
    ~C(){cout<<"调用析构函数 C"<<' ';}
};

class D:public B,public C{
    int dx;
public:
    D(int x,int y,int z):B(x),C(y)
{dx=z;cout<<"调用构造函数 D"<<dx<<' ';}
    void f(){cout<<dx<<' ';}
    ~D(){cout<<"调用析构函数 D"<<' ';}
};

void main(){
    A *pa=new D(20,30,40);
    cout<<endl;
    pa->f();
    cout<<endl;
    delete pa;
}
```

9. 写出以下程序的运行结果。

```
#include <iostream.h>
class Instrument{
public:
    virtual void display() const { cout<<"Instrument::display"<<endl; }
};
class Piano:public Instrument{
public:
    void display() const { cout<<"Piano::display"<<endl; }
};
class Guitar:public Instrument{
public:
    void display() const{ cout<<"Guitar::display"<<endl; }
};
void tone(Instrument & i)
```

```

{ i.display(); }
void main(void){
    Guitar guitar1;
    tone(guitar1);
    Piano piano1;
    tone(piano1);
}

```

10. 写出以下程序的运行结果。

```

#include <iostream.h>
class Base{
    int a;
public:
    Base(int a1 = 0){a = a1; show(); f1();}
    virtual void show(){cout<<"Base::show()"; cout<<"a="<<a<<endl;}
    void f1(){cout<<"Base::f1()"; cout<<"a="<<a<<endl; show();}
};
class Derived : public Base{
    int a;
public:
    Derived(int a1 = 1){a = a1; show(); f1();}
    void show(){cout<<"Derived::show()"; cout<<"a="<<a<<endl;}
};
void main(){
    Base *p1 = new Derived;
    p1->show();
    delete p1;
}

```

11. 考虑大学里的几类人员：学生、教师、进修教师，其中进修教师是具有学生身份的教师。分别给出这几类人员的类描述。提示：先提取公共信息构成基类，由基类派生出各类人员。

12. 建立一个 PC(Personal Computer)类表示个人计算机，建立一个 TablePC 类表示台式 PC，建立一个 NoteBook 类表示笔记本。建立这三个类之间的关系，分别描述各个类的数据成员，然后分别描述其构造函数、成员函数等，分析基类中的虚函数，实现这些类并测试。

13. 建立一个 File 类表示所有的文件，应描述文件名、大小、创建日期、修改日期等属性。建立一个 MediaFile 表示所有的多媒体文件，包括所有格式的音频、视频文件，不仅要描述其普通文件属性，还要描述媒体类别(如音频、视频)、播放时间等属性。

14. 分析各种打印机，先建立两个类分别描述激光打印机和喷墨打印机，分析两者之间的共同属性和操作函数，建立一个更抽象的类作为这两个类的基类。然后分析下面情形，看是否需要建立新的类，或者改进已有的类：

(a)如何描述一款 HP 生产的彩色激光打印机，如 HP LaserJet5500。

(b)如何描述一款银行使用的票据打印机(一种针式打印机)，如富士通 8600E。

15. 分析带 USB 接口的各种数码产品，先建立若干类分别描述 U 盘、MP3/4 播放器、数码相机，分析这些类有哪些基类，建立这些基类以及继承关系。然后尝试描述一款新型手机，如 Nokia N78，具有 U 盘、MP3 播放器、数码相机、GPS 导航、视频播放器等功能，使用已有的类如何来描述这款手机。

16. 在 Point 类基础上，建立一个 Pixel 类表示屏幕上的像素。一个像素除了要描述其坐标(x,y)之外，像素还有一个颜色，因此还需要建立一个 Color 类，一种颜色作为 Color 类的一个对象，根据三基色原理，每一种颜色都是由红 Red、绿 Green、兰 Blue 三种基色组合而成，每一种基色用 8 位表示，24 位色就是一种常见的真彩色规范。对 Pixel 和 Color 类进行编程测试。考虑这样的问题，像素的坐标(x, y)不允许出现负值，

而且因屏幕分辨率的限制，有最大值限制(例如 1280\*1024)。此时 Pixel 类的构造函数应如何设计？

# 第12章 运算符重载

普通运算符(如加+、减-)不能作用于对象，但运算符重载函数能使特定运算符作用于特定类的对象。运算符重载的本质是用成员函数或者友元函数来实现运算符的功能。运算符重载的好处是利用运算符来简化函数调用，从而使对象操作更直观更方便。运算符重载是 C++ 语言的一个特色，体现了面向对象编程的静态多态性与多样性灵活性。

## 12.1 一般运算符重载

一部分双目运算符和单目运算符可以被重载定义，就是按照运算符原先的语义和语法，采用成员函数或者友元函数来实现特定的运算。本节介绍运算符重载函数，如何用成员函数来定义双目运算符和单目运算符的重载函数定义，以及如何使用这些重载函数。

### 12.1.1 运算符重载函数

运算符重载函数是一种特殊的函数，名称为 `operator<运算符>`。编译器将对象运算表达式自动转换为相应的运算符重载函数的调用。运算符重载函数通常是类的成员函数或者友元函数。运算符可以是双目，也可以是单目，操作数中要求至少有一个是对象。

用成员函数来定义运算符重载函数的一般格式为：

```
<返回类型> <类名>::operator<运算符>(<形参表>)  
{  
    ...  
}
```

//函数体

其中，`operator` 是 C++关键字，其后的<运算符>可以是单目运算符，如“++”，也可以是双目运算符，如赋值运算符“=”、加法运算符“+”等。

由于运算符重载函数的函数名有特殊关键字，编译器容易与其它函数区分。

成员函数体中，当前对象作为一个操作数，无形参形式可定义单目运算符。如果有一个形参，这可定义双目运算符，当前对象作为左操作数，而形参作为右操作数。

用成员函数来定义运算符重载函数，如果有形参的话，最多只有一个。

大多数运算符都可以定义运算符重载函数，但下表中的运算符不允许重载。

表 12-1 C++中不允许重载的运算符

运算符	运算符的含义
? :	三目运算符
.	成员运算符
.*	成员指针运算符

::	作用域操作符
sizeof	求字节数操作符
#和##	预编译指令符号

运算符重载属于一种静态的多态性，C++编译器在编译时将运算符转换为函数调用，一般处理过程如下：

1. 当遇到对象参与运算时，先查找该类中是否有成员函数重载了该运算符。
2. 若有，则调用相应的成员函数来实现这种运算。若没有，就查看是否用友元函数重载了该运算符。
3. 若有友元重载，则调用相应的友元函数。若没有，就用该类中定义的转换函数将对象转换为其它类型，再返回第 1 步查找。
4. 如果没有转换函数，或者即便有转换函数但仍未找到转换后能匹配的重载函数，就给出编译错误。

### 12.1.2 双目运算符的重载

先分析下面用成员函数定义双目运算符的例子。

例 12-1 定义一个复数类 **Complex**，用+运算符完成复数加法运算。可以是两个复数相加，也可以是一个复数加一个实数。不仅包括“+”，还应包括“+=”。编程如下：

```
#include <iostream>
using namespace std;
class Complex{
    float real, image;
public:
    Complex(float r=0, float i=0){ real=r; image=i;}
    float getR(){return real;}
    float getI(){return image;}
    void show() { cout <<real<<'+ '<<image<<"i\n"; }
    Complex operator +(Complex &c){ //重载运算符+
        Complex t;
        t.real = real + c.real;
        t.image = image + c.image;
        return t;
    }
    Complex & operator +=(Complex &c){ //重载运算符+=
        real += c.real;
        image += c.image;
        return *this;
    }
    Complex operator +(float f){ //重载运算符+
        Complex t;
        t.real = real + f;
        t.image = image;
        return t;
    }
};

int main(){
    Complex c1(1,2), c2, c3(3,2);
    Complex c, c4(2,3);
    c2 = c1;           c2.show(); //A
```

```
c = c1 + c3;          c.show();          //B
c = c4 + 4;           c.show();          //C
c += c1;              c.show();          //D
c4 += c1 + c2 + c3; c4.show();          //E
system("pause");
return 0;
}
```

执行程序，输出如下：

```
1+2i
4+4i
6+3i
7+5i
7+9i
```

例子中没有显式定义赋值运算符，但编译器提供的缺省赋值函数能起作用。对运算符+作了两次重载，一个实现两个复数相加，另一个实现一个复数加一个实数。本质上，运算符的重载也是函数的重载，不同的是系统约定了重载运算符的函数名。

A 行调用了赋值操作函数。

B 行转换为 `c.operator=(c1.operator+(c3))`，先计算 `c1+c3`，然后再将返回值赋给 `c`。

C 行转换为 `c.operator=(c4.operator+(4))`，先计算 `c4+4`，然后再将返回值赋给 `c`。

D 行转换为 `c.operator+=(c1)`

E 行比较复杂，转换为 `c4.operator+=c1.operator+(c2).operator+(c3)`，即先计算 `(c1+c2)+c3`，然后再计算 `+=`。

对运算符的重载，注意以下要点。

(1) 定义一个运算符重载函数应保持该运算符的语义，而不能随意改变。例如，对于加法运算符函数，不能在函数体中实现减法或其它计算。并非所有的类都适合定义任意一个运算符。例如 `Person` 类表示人，人作为对象，加法并不能表示有意义的计算。

(2) 将表达式转换为运算符重载函数调用，要符合运算符的优先级和结合性。例如，先加法后赋值，加法要从左向右运算，赋值要从右向左运算。

(3) 用成员函数实现双目运算符，当前对象作为左操作数，右操作数可以是一个对象，也可以是基本类型值，如 `float` 值。上面例子中，不能计算 `4f+c4`，这是因为例子中没有定义这样的运算符函数：左操作数是一个 `float`，右操作数是一个 `Complex` 对象。实际上用成员函数无法定义这样的重载函数，只能用友元函数来定义。

(4) 一个运算符重载函数的结果与返回值应该与该运算符的计算相一致。例如，加法运算不能改变左右两个操作数，因此函数体中建立一个对象并返回作为结果。而“+=”和“=”运算要改变左操作数，就是改变当前对象并作为返回结果，因此函数的返回类型应该是“<类名>&”，函数体中最后一条语句往往是 `return *this`。

### 12.1.3 单目运算符的重载

几乎所有的单目运算符都可以定义重载函数。用成员函数重载单目运算符时，对于++和--运算符，必须区分前置和后置运算，以便编译器调用不同的运算符重载函数。++和--运算符的重载方法类同，下面以++为例说明。

++为前置运算符时，用无参成员函数格式：

```
<类名> &<类名>::operator ++( ){
    ...                                //改变当前对象
    return *this;                      //返回当前对象作为表达式的值
}
```

++为后置运算符时，要带一个 int 形参，但函数体中无用：

```
<类名> <类名>::operator ++(int){
    <类名> <变量名> = *this;           //先创建一个对象保存当前对象状态
    ...                                //改变当前对象
    return <变量名>;                  //返回原先保存的对象
}
```

例 12-2 对包装类 Integer 添加前置和后置自增运算。

```
#include <iostream>
using namespace std;
class Integer{
    int data;
public:
    Integer(int x=0) {data=x;}
    void setData(int x){data=x; }
    int getData(){return data;}
    Integer &operator++(){          //前置++
        ++data;
        return *this;
    }
    Integer operator++(int){        //后置++
        Integer i = *this;
        ++data;
        return i;
    }
    void show(){ cout <<"int data = "<< data <<'\n';}
};

int main(){
    Integer i1;
    ++i1;                          //A
    i1.show();
    Integer i2 = i1++;              //B
    i2.show();
    i1.show();
    Integer i3 = (++i1)++;          //C
    i3.show();
    i1.show();
    Integer i4 = ++i1++;            //D
    i4.show();
    i1.show();
    system("pause");
    return 0;
}
```

执行程序，输出如下：

```
int data = 1
int data = 1
int data = 2
```



```
int data = 3
int data = 4
int data = 5
int data = 5
```

A 行调用了前置自增函数，然后输出第 1 行：1。

B 行调用了后置自增函数，然后输出第 2 行：1 和 2。这符合后置自增运算规则。

C 行先调用前置自增，然后调用后置自增，即 `i1.operator++().operator++(int)`。输出 `i3` 值为 3，`i` 的值为 4，也符合运算规则。

D 行代码中，如果 `i1` 作为 `int` 类型，编译就要出错，这是因为后置自增运算符的优先级比较高，因此 `++i1++` 就等价于 `++(i1++)`。在做前置自增运算时就没有左值，因此报错。但现在 `i1` 是一个对象，严格按优先级转换为 `i1.operator++(int).operator++()`。先调用后置自增函数，返回原值 4 之前 `i1` 的值成为 5，然后对返回的值 4 再调用前置自增，成为 5，因此 `i4` 的值为 5，而且 `i1` 的值也是 5，就是最后输出的两行。

运算符重载函数应该返回什么？首先运算符重载函数不应返回 `void`，原因是任何表达式都具有特定的类型，因此运算符函数都应返回某种类型，但问题是应返回对象还是对象引用。有以下情形：

1、如果运算符将得到一个新结果，而不是将某个操作数作为结果，就应返回对象，而不是对象引用。例如 `a+b` 的结果是一个新对象，而 `a` 和 `b` 都不变，因此 `operator+` 应返回对象。再如后置 `++` 运算符，`a++` 的结果应该是 `a` 加 1 前的原值，而不是操作数 `a`，因此 `operator++(int)` 应返回对象。这种情形都要求函数体中应创建一个局部对象，最后返回该对象。

2、如果运算符将某个操作数作为结果，那么就应返回对象引用，而不是对象。例如 `a+=b`，结果是操作数 `a`，因此 `operator+=` 函数应返回对象 `a` 的引用。再如前置 `++`，如 `++a` 的结果是加 1 后的操作数 `a`，因此 `operator++()` 应返回对象引用，而不是对象。否则，当多个运算符作用于同一个对象时就可能出错。这种情形下，函数体中往往不创建对象，只是返回当前对象或形参对象的引用。

3、一些特殊运算符函数的原型是确定不变的。后面 12.3 节介绍。

4、不应出现的错误情形是，当重载函数要求返回对象时，但函数体中却返回了某个操作数，此时语法没有错误，但要执行拷贝构造函数而创建新对象。另一种情形是重载函数要求返回对象引用，但函数体中却将创建的局部对象返回，此时编译有警告。

## 12.2 用友元函数实现运算符重载

除了成员函数之外，也可用友元函数来实现部分运算符重载。

### 12.2.1 友元函数

对于类中的私有成员，只有该类的成员函数才能访问，类外函数无权访问。但如果把类外一个函数定义为该类的一个友元函数，该函数就能访问该类中的所有成员。

友元函数就是用关键字 `friend` 修饰的一个非成员函数，使其能访问该类中的所有成员。友元函数可以是一个全局函数，也可以是另一个类中的某个成员函数。也可用 `friend` 修饰另

一个类，将该类的所有成员函数都作为本类的友元函数。

本文下面介绍如何使用全局友元函数来实现运算符重载。

有关友元函数，注意以下几点。

(1) 友元函数不是类的成员函数，因此函数体中没有 `this` 指针，往往就需要把对象的引用或指针作为友元函数的形参，在函数体内通过这些形参来访问对象的成员。

(2) 类中指定的访问权限对友元函数是无效的，因此把友元函数说明放在类的私有部分、公有部分或保护部分，效果都一样。友元函数的定义可以在类外，也可以在类内。

(3) 慎用友元函数。类的一个重要特性是封装性，而友元函数破坏了类的封装性。友元函数除了实现一些特定的运算符重载函数之外，应尽量避免使用。

(4) 一个类的友元函数不能被该类的派生类所继承。

### 12.2.2 定义重载函数

用友元函数重载双目运算符的一般格式为：

<返回类型> operator <运算符> (形参 1, 形参 2)

其中，<运算符>是一个双目运算符，形参 1 和形参 2 就是参与双目运算的两个操作数，这两者之中至少有一个是对象类型或对象引用类型。该函数应说明为形参对象类的友元。例如：

```
Complex operator+(const Complex &c1, const Complex &c2);
```

实现了两个 `Complex` 对象之间的加法计算，那么 “`c1 + c2`” 表达式就等价转换为函数调用 `operator+(c1, c2)`。

用友元函数重载单目运算符的一般格式为：

<返回类型> operator <运算符> (<类名> 形参名)

其中，运算符是一个单目运算符，形参是对象类型或对象引用类型。对于 `++`、`--` 运算符来说，形参只能是类的引用。同样，`++` 或 `--` 运算符要区分前置和后置运算，后置运算要添加一个 `int` 形参，以区别前置运算符函数。例如：

```
Complex &operator++(Complex &c); //前置自增
```

实现了 `Complex` 对象的前置 `++` 运算，那么 “`++c`” 表达式就等价转换为函数调用 `operator++(c)`。再例如：

```
Complex operator++(Complex &c, int ); //后置自增
```

实现了 `Complex` 对象的后置 `++` 运算，那么 “`c++`” 表达式就等价转换为函数调用 `operator++(c, 0)`。第二个实参并不起作用。

VS2015 允许友元运算符重载函数定义在类中。

例 12-3 对 `Complex` 类设计一组友元重载函数，编程如下：

```
#include <iostream>
using namespace std;
class Complex{
    float real, image;
public:
    Complex(float r=0, float i=0){ real=r; image=i;}
    float getR(){return real;}
    float getI(){return image;}
    void show() { cout <<real<<'+ '<<image<<"\n"; }
```

```

    friend Complex operator+(const Complex &c1, const Complex &c2);
    friend Complex operator+(const Complex &c1, float f);
    friend Complex &operator+=(Complex &c1, const Complex &c2);
    friend Complex &operator++(Complex &c);           //前置++
    friend Complex operator++(Complex &c, int);       //后置++
    friend bool operator==(const Complex &c1, const Complex &c2);
};
Complex operator+(const Complex &c1, const Complex &c2){
    return Complex(c1.real + c2.real, c1.image + c2.image);
}
Complex operator+(const Complex &c1, float f){
    return Complex(c1.real + f, c1.image);
}
Complex operator+(float f, const Complex &c){//不是友元函数, 只是运算符函数
    return operator+(c, f);
}
Complex &operator+=(Complex &c1, const Complex &c2){
    c1.real += c2.real;
    c1.image += c2.image;
    return c1;
}

Complex &operator++(Complex &c){                     //前置自增
    c.real++; c.image++;
    return c;
}
Complex operator++(Complex &c, int){                 //后置自增
    Complex t = c;
    c.real++; c.image++;
    return t;
}
bool operator==(const Complex &c1, const Complex &c2){
    return c1.real == c2.real && c1.image == c2.image;
}
int main(){
    Complex c1(1, 2), c2(2, 3);
    Complex c3 = c1 + c2;
    c3.show();
    Complex c4 = 4 + c1 + 5;    //A
    c4.show();
    c4 += c1;
    c4.show();
    Complex c5 = ++c1;
    c5.show();
    Complex c6 = c2++;
    c6.show();
    c2.show();
    if (c2 == Complex(3, 4))
        cout<<"operator== ok"<<endl;
    system("pause");
    return 0;
}

```

执行程序, 输出如下:

```

3+5i
10+2i
11+4i
2+3i
2+3i

```

```
3+4i
operator== ok
```

前面例子中, `Complex` 类用成员函数定义+运算符重载函数, 能计算 `c + 4f`(`c` 是一个 `Complex` 对象), 但不能计算 `4f + c`, 这是因为成员重载函数要求二元运算符的左操作数是一个对象。上面例子中用一个运算符函数来实现一个 `float` 与一个 `Complex` 对象加法:

```
Complex operator+(float f, const Complex &c);
```

这样就能计算表达式: `4f + c`。

注意, 这个函数没有必要说明为 `Complex` 类的友元函数, 这是因为运算符函数体中调用了另一个运算符函数, 而没有访问类中的私有成员, 因此 `Complex` 类中无需进行友元说明来授权访问。实际上, 全局运算符函数能定义对象运算符, 只是如果不说明为类的友元就不能访问对象内的非公有成员。

从上例可知, 友元函数能实现多数成员函数能实现的运算符函数, 但对一个类来说, 对于一种运算符形式只需一种实现方式。友元函数可作为成员函数的补充, 用来实现成员函数所不能实现的一些运算符重载函数, 如双目运算符的右操作数为本类对象, 而左操作数不是本类对象。常见例子是流输出运算符<<, 如 `cout<<c`。

重载<<运算符的一般格式为:

```
friend ostream & operator <<(ostream &, ClassName &);
```

函数的返回值是类 `ostream` 的引用, 这样可以连续使用<<运算符; 第一个形参是类 `ostream` 的引用, 它是<<运算符的左操作数, 一般调用实参就是 `cout` 对象; 第二个形参为自定义类的对象引用, 它是<<运算符的右操作数。这样表达式 “`cout<<c`” 就能编译为一个友元函数调用 “`operator<<(cout, c)`”, 其中 `c` 就是某个类(如 `Complex`)的一个对象。

在上面 `Complex` 类中添加一个友元函数说明:

```
friend ostream & operator<<(ostream & os, Complex & c);
```

然后在类外定义该函数, 如下:

```
ostream & operator<<(ostream & os, Complex & c) {
    os<<"("<<c.real<<"", "<<c.image<<"")";
    return os;
}
```

这样就可以用 `cout<<c` 来显示对象 `c` 的状态。这个函数可以替代成员函数 `show()` 或 `print()`, 用来输出当前对象的当前状态。

注意到上面这个友元函数中访问了对象 `c` 的私有成员, 但该类提供了公有成员函数 `getR()` 和 `getI()` 来读取私有成员, 就可以不用在类中说明友元函数, 而用一个全局运算符函数来实现:

```
ostream & operator<<(ostream & os, Complex & c) {
    os<<"("<<c.getR()<<"", "<<c.getI()<<"")";
    return os;
}
```

重载流输入>>运算符的一般格式为:

```
friend istream & operator >>(istream &, ClassName &);
```

对于流输出<<运算符和输入>>运算符, 在第 14 章中将详细介绍。

标准模板库 STL 中提供了<complex>复数类型，C11 还添加了用户自定义字面值。

### 12.2.3 用户自定义字面值 UDL

C11 支持用户自定义字面值(User-Defined Literals, 简称 UDL)。在已有字面值的基础上, 用户以后缀形式扩展自己的字面值, 既能符合公共习惯用法, 也能增强类型安全性。

例如, <complex>中的复数 Complex, UDL 支持如下字面值(后缀 i 表示虚部):

```
complex<double> num = (2.0 + 3.01i) * (5.0 + 4.3i);
```

再如, 距离 Distance 的单位可用千米 km, 也可用英里 mile, 那么 10.0\_km + 20.0\_mi 这样的表达式就包含了用户自定义字面值。前面是一个 double 字面值, 之后是一个用户自定义后缀, \_km 或者 \_mi。系统要将这样的字面值对应到一个 Distance 对象, 就需要在单参构造函数基础上, 扩展两个特殊的字面值运算符函数。字面值运算符函数语法如下:

返回类型 operator"" 自定义后缀(已有字面值类型)

例如:

```
Distance operator"" _km(long double)
```

就可将 10.0\_km 这样的字面值转换为一个 Distance 对象。

```
Distance operator"" _mi(long double)
```

就可将 20.0\_mi 这样的字面值转换为一个 Distance 对象。

要求 1, 自定义后缀要用下划线开始, 只有标准库(如 Complex)才能省去下划线。

要求 2, 已有字面值类型应采用 long double 而不是 double。

如果要支持 100.0\_km + 200.0\_mi 这样的计算, 还要在 Distance 类中实现一个 operator+ 双目运算符。下面是具体编码:

```
#include<iostream>
using namespace std;
class Distance{
private:
    double kilometers;
    explicit Distance(double val) : kilometers(val)    {}           //A
    friend Distance operator"" _km(long double val);    //B用户自定义后缀_km
    friend Distance operator"" _mi(long double val);    //C用户自定义后缀_mi
public:
    double get_kilometers() { return kilometers; }
    Distance operator+(Distance& other) {                //D用成员函数实现operator+
        return Distance(get_kilometers() + other.get_kilometers());
    }
};
Distance operator"" _km(long double val) {
    return Distance(val);
}
Distance operator"" _mi(long double val) {
```

```

    return Distance(val * 1.6);
}

int main(){
    Distance d1( 10.0_km );
    cout << "Kilometers in d1: " << d1.get_kilometers() << endl;
    Distance d2( 20.0_mi );
    cout << "Kilometers in d2: " << d2.get_kilometers() << endl;
    auto d3 = 10.0_km + 20.0_mi ;
    cout << "d3 value = " << d3.get_kilometers() << endl;
    system("pause");
    return 0;
}

```

A 行定义一个单参构造函数，将一个 `double` 类型转换为一个 `Distance` 对象，此时缺省单位为 `km`。为支持用户自定义后缀，B 行 C 行添加了两个自定义字面值运算符函数，并采用友元函数实现。D 行是用成员函数实现的 `operator+`，支持加法运算。

执行程序，输出结果如下：

```

Kilometers in d1: 10
Kilometers in d2: 32
d3 value = 42

```

为什么字面值的形参要求为 `long double`？原因是作为基础的字面值类型只有如下几种：

```

ReturnType operator "" _a(unsigned long long int);    //整型字面值
ReturnType operator "" _b(long double);               //浮点型字面值
ReturnType operator "" _c(char);                      //字符型 char 字面值
ReturnType operator "" _d(wchar_t);                   //字符型 wchar_t 字面值
ReturnType operator "" _e(char16_t);                  //字符型 char16_t 字面值
ReturnType operator "" _f(char32_t);                  //字符型 char32_t 字面值
ReturnType operator "" _g(const char*, size_t);       //字符串 char*字面值
ReturnType operator "" _h(const wchar_t*, size_t);    //字符串 wchar_t*字面值
ReturnType operator "" _i(const char16_t*, size_t);   //字符串 char16_t*字面值
ReturnType operator "" _j(const char32_t*, size_t);   //字符串 char32_t*字面值
ReturnType operator "" _r(const char*);               // Raw 字面值
template<char...> ReturnType operator "" _t();        //模板字面值

```

注意上面列出的后缀仅仅是一个占位符，用户应定义自己的后缀。

上面例子采用了第 2 个形式。所有浮点型字面值都统一采用 `long double`，并非 `double`。

可以看出，`bool` 类型字面值不能添加自定义后缀。

上面列出的运算符函数没有当前对象，因此只能用友元函数来实现。

## 12.3 特殊运算符的重载

本节讨论几个特殊的运算符：赋值操作函数、类型转换函数、下标运算符和函数调用运算符，这些运算符函数只能用成员函数来实现。

### 12.3.1 赋值操作函数

赋值运算符重载函数(简称为赋值函数，或者赋值操作函数)是最重要的一个运算符。每个类中都有。如果不显式定义，编译器就会自动生成一个缺省的赋值函数，而且函数体中先复制形参对象的所有基类的所有数据成员，再复制各个成员对象，最后再复制基本类型的各个数据成员给当前对象。在前面第 10 章已经介绍，下面仅简单总结要点。

赋值函数的格式如下：

```
<类名> &<类名>::operator=(const <类名>&)
```

对于赋值函数，应注意以下要点。

(1)赋值函数虽然是一种成员函数，但不能被派生类继承，也不能说明为虚函数。原因是派生类对象的赋值语义与基类可能不一样，当派生类增加了新的数据成员后，就可能改变了赋值语义。缺省生成的赋值函数要复制所有数据成员，而显式定义的赋值函数就需要自行处理各个数据成员，也包括基类的数据成员。

(2)注意缺省生成的赋值函数的副作用，如果类中有指针数据成员或引用数据成员，往往都隐含着副作用。

(3)一些实体对象往往持有标识性的数据成员，例如人的身份证号、学生的学号、产品的序列号等，这些标识属性唯一确定对象的身份，往往是不可更改的，但缺省提供的赋值操作函数会自动复制这些数据成员，而导致重复标识出现在不同对象中，导致对象标识混乱。这些类的赋值操作函数不应被随意调用，或者需要特别的设计。如果将赋值操作函数说明为私有，编译器就能阻止类外代码执行对象赋值，这样迫使对象传递只能用引用或指针来进行。

### 12.3.2 类型转换函数

类型转换函数(简称为转换函数)能将一个对象转换为另一种类型的一个对象或值。它即可显式转换，也可隐式转换。类型转换是一种无参的成员函数，其定义格式：

```
类名:: operator 目标类型()
```

其中，“类名”就是当前类的名字，“目标类型”是该函数返回类型，它可以是任一种数据类型，但不应与“类名”相同。`operator` 与目标类型一起构成了一个转换函数的名称。一个类中可定义多个转换函数，各自目标类型不同。该函数没有形参，也不用指定返回值。但该函数体中应返回一个目标类型的对象或值。

对于表达式“目标类型(对象)”或者“(目标类型)对象”，编译器将其解释为“对象.`operator 目标类型()`”的函数调用。

前面介绍过“单参构造函数”，用于将指定类型的一个对象或值转换为本类的一个对象。转换函数的作用与单参构造函数正相反。

例 12-4 定义一个 **RMB** 类，将一笔人民币金额作为一个对象，能表示元、角、分，分别

用转换函数转换为一个 `float` 值和一个字符串。编程如下:

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
#include<string>
#include<stdlib.h>
using namespace std;
class RMB{
    int yuan, jiao, fen;
public:
    RMB(int y=0, int j=0, int f=0){                //构造函数 1
        int ff = y*100 + j*10 + f;
        fen = ff % 10;
        jiao = ff / 10 % 10;
        yuan = ff /100;
    }
    RMB(double f){                                //单参构造函数
        yuan = (int)f;
        jiao = int(f * 10) % 10;
        fen = int(f * 100) % 10;
    }
    operator double(){                            //转换函数 1: 转为 float
        return (yuan*100.0 + jiao*10.0 + fen)/100;
    }
    operator string(){                             //转换函数 2:转为 string
        char buffer[20];
        string str(_itoa(yuan, buffer, 10));
        str += "元" ;
        str += _itoa(jiao, buffer, 10);
        str += "角" ;
        str += _itoa(fen, buffer, 10);
        str += "分";
        return str;
    }
};

int main(){
    RMB b1 = 32.456;                               //A
    float f = b1;                                   //B
    cout<<f<<endl;
    cout<<string(b1)<<endl;                         //C
    RMB b2(12, 34, 56);                             //D
    cout<<b2<<endl;                                 //E
    cout<<string(b2)<<endl;
    system("pause");
    return 0;
}
```

执行程序, 输出如下:

```
32.45
32 元 4 角 5 分
15.96
15 元 9 角 6 分
```

类中定义了两个构造函数, 其中第二个是单参构造函数, 用来将一个 `float` 值转换为 `RMB` 类的一个对象。类中定义了两个转换函数, 分别转换到 `float` 和 `string`。

A 行调用了单参构造函数, 用一个 `float` 数值创建一个 `RMB` 对象。



B 行有一个隐式转换，调用了转换函数 1，将对象转换为一个 float 值再赋值。

C 行调用了转换函数 2，将对象转换为一个 string 对象再输出。

D 行调用了第一个构造函数，该构造函数中的角和分的实参值允许大于 10。例如，12 元、34 角、56 分加起来就是 15.96 元，对象内部的数据成员的值就是 15、9、6。

E 行有一个隐式转换，调用了转换函数 1，转为一个 float 值再输出。

例子中使用了 `_itoa` 函数将 int 值转换为 char 数组，例如 `_itoa(yuan, buffer, 10)`。第一个实参是被转换的一个 int 值，第二个实参是一个 char 数组，应该有足够大小，第 3 个实参确定了十进制转换。要使用这个函数就要包含 `stdlib.h` 文件。

转换函数在说明语句、赋值语句、函数调用、函数返回等语句中会自动调用，就像单参数构造函数一样。转换函数是一种成员函数，可以定义为虚函数，可以被派生类继承或改写。

一个类中如果转换函数转换为 `const char *` 或者 `string`，就能将该类对象的当前状态显示出来，以方便测试，这是转换函数最常见的用法。

如果不希望出现隐式转换，对转换函数添加 `explicit` 修饰，只允许显式转换。上面 E 行的隐式转换就被禁止而导致编译错误。

### 12.3.3 下标运算符

在 C++ 程序中用下标访问数组元素时，并不检查下标是否越界。通过重载下标运算符可以实现下标越界的检查，也能简化访问元素的形式。下标运算符重载函数是一种成员函数，持有一个 int 形参作为下标值，返回指定类型的一个值作为下标对应的元素，一般是返回某种类型的引用。重载下标运算符函数的格式为：

<返回类型> <类名>::operator[] (<形参>)

其中，形参往往是一个 int 类型，也可以是其它类型，只要能对应一个元素即可。<返回类型>往往是对象引用，目的是可作为表达式的左值。下标运算符不能用友元函数定义。

对于表达式“对象[实参]”，编译器将其转换为“对象.operator[](实参)”。该表达式的左操作数必须是一个对象，一个实参作为右操作数。

下标运算符函数往往定义在 STL 容器中，容器 container 的一个对象中要管理多个元素，如 vector 向量等。用一个下标就可以确定一个对象。

下标运算符是一个双目运算符，它的左操作数在“[”的左边，而且必须是一个对象；右操作数在方括号之中，它可以是任意类型，多为整数 int。对函数返回值的类型没有限制。

由于下标运算符函数只有一个形参，故此只能对一维数组进行下标重载。对于二维数组可用函数调用运算符来处理。

### 12.3.4 函数调用运算符

在一个对象名后加一对圆括号，可指定一组实参，就能调用一种特殊的成员函数，就是函数调用运算符函数，例如 `obj(实参)`。重载该运算符的格式为：

<返回类型> <类名> :: operator() (<形参表>)

其中，`operator()` 为函数名。与通常的成员函数一样，该函数可以带有 0 个或多个形参，但不能带缺省值。<返回类型>可以是对象，也可以对象引用。

对于表达式“对象(实参表)”，编译器将处理为“对象.operator()(实参表)”。

假设有一个矩阵类 `FloatMatrix`，将一个 `float` 矩阵作为一个对象。其中访问矩阵中的元素需要通过调用下面函数来实现：

```
float & FloatMatrix::elemAt(int r, int c){           //按下标访问元素
    if (r < 0 || r >= row)                          //如果下标 r 越界，就引发异常
        throw r;
    if (c < 0 || c >= col)                          //如果下标 c 越界，就引发异常
        throw c;
    return dp[r * col + c];
}
```

调用此函数能按下标来访问元素，例如，`m1` 是 `FloatMatrix` 的一个对象：

```
m1.elemAt(i, j) = aa[i][j];           //aa 是一个二维 float 数组
```

只需在类中添加一个函数调用运算符重载函数如下：

```
float & FloatMatrix::operator()(int r, int c)
{return elemAt(r,c);}
```

就能用一种新的方式来访问二维数组元素，例如：

```
m1(i, j) = aa[i][j];
```

这种形式更接近二维数组的访问形式。

注意，函数调用运算符函数只能是成员函数，其左操作数必须是对象。

函数调用运算符可支持所谓的函数对象 `function object`。如果某个类实现了一个函数调用运算符，那么该类的对象就被称为函数对象，也称为 `functor` 函子。函数对象在 STL 算法调用时作为实参，可提供算法所要求的排序规则、数据生成、一元或二元谓词等，能深入定制 Lambda 表达式或函数指针难以实现的特殊功能。关于函数对象在后面介绍。

究竟哪些类适合定义哪些运算符重载函数？

首先，定义运算符重载函数与定义普通函数一样，都是定义某种操作行为。这种操作是命名为运算符，还是命名为标识符，只是用法不同，而没有功能差别。运算符函数的作用只是用表达式来简化函数调用，使对象操作更简便更直观。因此定义运算符重载函数并不能增强类的功能。

一个运算符对于不同类的对象可能具有特殊含义，也可能根本无用。例如，加法“+”作用于字符串，可以解释为字符串拼接，而自增运算符“++”作用于字符串就很难解释有什么意义。对多数类来讲，除了赋值函数之外并不需要定义运算符函数，如果需要，最常用的运算符如下：

- 流输出 `operator<<`运算符，用来观察对象状态。
- 判断相等 `operator==`，定义同一类的两个对象在什么条件下相等。
- 判断小于 `operator<`，定义同一类的两个对象之间的排序规则。
- 类型转换 `operator<类型>`，将当前对象转换为指定类型的对象或值。

对于一些运算符应谨慎处理，如 `new` 和 `delete`，涉及到动态内存管理，具有充分经验之后再考虑重载。

## 12.4 小 结

- 运算符重载的本质是用成员函数或者友元函数来实现运算符的功能。运算符重载的好处是利用运算符来简化函数调用，从而使对象操作更直观更方便。
- 运算符重载函数是一种特殊的函数，名为 `operator<运算符>`。
- 运算符重载函数通常是类的成员函数或者友元函数。运算符可以是双目，也可以是单目，操作数中要求至少有一个是对象。
- 如果用成员函数来定义运算符函数，当前对象就作为单目运算符的操作数，此时不需要形参。如果定义双目运算符函数，当前对象就作为左操作数，单个形参作为右操作数。右操作数可以是一个对象，也可以是基本类型值。
- 定义运算符重载函数应保持该运算符的语义。将表达式转换为运算符重载函数调用，要符合运算符的优先级和结合性。
- 运算符重载函数的返回值应该与该运算符的计算结果相一致。
- 用成员函数重载前置自增自减运算符时，无形参，而且要改变当前对象，最后返回当前对象 `return *this`。
- 用成员函数重载后置自增自减运算符时，有 1 个 `int` 形参，要改变当前对象，但返回的是当前对象原先的状态。
- 一个类的友元函数就是用关键字 `friend` 修饰的一个类外函数，使其能访问该类中的所有成员。定义友元函数的本质就是授权访问。
- 一个类的友元函数可以是一个全局函数，也可以是另一个类中的某个成员函数，也可以是某个类中的所有成员函数(用 `friend class <类名>;`)
- 友元函数不是本类的成员函数，因此函数体中没有 `this` 指针，往往要把对象、或对象的引用或指针作为友元函数的形参，并在函数体内通过形参来访问对象的成员。
- 类中指定的访问权限对友元函数是无效的，因此把友元函数放在类的私有部分、公有部分或保护部分，效果都一样。
- 类的友元函数不能被其派生类所继承。
- 慎用友元函数。友元函数破坏了类的封装性，一般仅用于定义运算符重载函数。
- 友元函数可作为成员函数的补充，用来实现成员函数不能实现的一些运算符重载函数，如双目运算符的右操作数为对象的情形。
- 类型转换函数 (`operator<类型>`，简称为转换函数) 将当前对象转换为指定类型的一个对象或一个值。转换函数的作用与单参构造函数正相反，它们都可以被隐式调用。
- 赋值操作函数 `operator=` 每个类都有，如果不显式定义，编译器就会自动生成一个缺省的赋值函数。问题就是要判断缺省提供的赋值操作函数何时会有副作用。
- 下标运算符函数 `operator[]` 用一个形参下标来指定一个元素，就像访问一个数组的一个元素。
- 函数调用运算符 `operator()` 是在对象名后用圆括号和实参表来完成某种计算。
- 运算符重载不仅可简化函数调用和对象操作，在基于模板的容器中还将用于管理对象。

## 12.5 练 习 题

1. 对于用成员函数重载运算符，下面哪一种说法是错误的？

- A 无参函数只能用来定义单目运算符，当前对象作为操作数。
- B 单参函数可定义双目运算符，当前对象作为左操作数，形参作为右操作数。
- C 定义后置++或后置--运算是特例，它们是单目运算符，但需要一个 int 形参。
- D 函数体中最后要返回当前对象作为结果。

2. 用友元函数实现运算符重载，下面哪一种说法是错误的？

A 因为友元函数没有当前对象，因此要定义单目运算符，就需要单参函数，要定义双目运算符，就需要双参函数。

B 定义后置++或后置--运算是特例，它们是单目运算符，但需要两个形参，头一个形参是作用对象，后一个是 int 形参。

C 用友元函数可以定义成员函数不能实现的运算，例如一些双目运算符，右操作数是本类对象，而左操作数不是本类对象。

D 一个类说明的友元函数可以被派生类继承。

3. 写出以下程序的执行结果。

```
#include <iostream.h>
class A{
    int x;
public:
    A(int a=0){ x=a;cout<<"x="<<x<<"\t"<<"构造函数\n"; }
    A(A & e){ x = e.x;cout<<"x="<<x<<"\t"<<"拷贝构造函数\n"; }
    A & operator=(A & e) {
        x=e.x; cout<<"x="<<x<<"\t"<<"operator=\n";
        return *this;
    }
    operator int(){
        cout<<"x="<<x<<"\t"<<"operator int\n";
        return x;
    }
};
void main(void){
    A x1(50);
    A x2 = 100;
    x1 = x2;
    A x3 = x1;
    int i = x1;
    x2 = (A)200;
}
```

4 VC++支持最大整数为 32 位，大致范围是从-21 亿到+21 亿，最多 10 位十进制数，但有时我们需要更大的整数，如 30 位十进制数。下面程序尝试建立一个 BigInt 类，表示更大的整数，而且要能支持算术四则运算。分析下面程序，完成下面工作：

- a) 该类所表示的整数有什么限制？如何能表示负整数，并且能支持四则运算？
- b) 重载减法运算符。

- c) 重载乘法运算符。
- d) 重载除法运算符。
- e) 重载定义 6 个关系运算: <、<=、>、>=、==、!=。

```
#include <iostream.h>
#include <string.h>
#define MAX 30          //十进制位数
class BigInt{
    char integer[MAX];    //char 数组中每个元素表示一个十进制位
public:
    BigInt(int num = 0){
        int i;
        for(i = 0; i <= MAX; i++)
            integer[i] = 0;
        for(i = MAX - 1; num != 0 && i >= 0; i--){
            integer[i] = num % 10;
            num /= 10;
        }
    }
    BigInt(const char * str){
        int i, j;
        for(i = 0; i <= MAX; i++)
            integer[i] = 0;
        for(i = MAX - strlen(str), j = 0; i <= MAX-1; i++,j++){
            integer[i] = str[j] - '0';
        }
    }
    friend ostream & operator<<(ostream& os, BigInt & num);
    BigInt operator+(const BigInt & op2){
        BigInt result;
        int carry = 0;
        for (int i = MAX - 1; i >= 0; i--){
            result.integer[i] = integer[i] + op2.integer[i] + carry;
            if (result.integer[i] > 9){
                result.integer[i] %= 10;
                carry = 1;
            }else
                carry = 0;
        }
        return result;
    }
    BigInt operator+(int op2){
        return *this + BigInt(op2);
    }
    BigInt operator+(const char * op2){
        return *this + BigInt(op2);
    }
};

ostream & operator<<(ostream& os, BigInt & num){
    int i = 0;
    while(num.integer[i]==0 && i <= MAX - 1)
        i++;
    if (i == MAX)
        os << 0;
    else
        for(; i <= MAX - 1; i++)
            os<<(int)num.integer[i];
    return os;
}
```

```
void main() {
    BigInt n1 = 123456789, n2 = 987654321;
    cout<<"n1="<<n1<<"n2="<<n2<<endl;
    BigInt n3 = n1 + n2;
    cout<<"n1+n2="<<n3<<endl;
    BigInt n4 = "99999999999999999999";
    cout<<"n4="<<n4<<endl;
    BigInt n5 = n4 + "1";
    cout<<"n4+1="<<n5<<endl;
}
```

5. 建立一个分数类 **Fraction**，使一个分数作为一个对象，如  $1/2$ ， $2/3$  等。分子分母都是 **int** 型。
  - a) 建立构造函数，要求分母不能为 0，也不能为负数，而且能对分子分母约分化简。
  - b) 重载定义四则运算：加法、减法、乘法、除法，也包括与 **double** 型的四则运算。
  - c) 重载定义 6 个关系运算符：<、<=、>、>=、==、!=。
  - d) 重载定义类型转换函数到 **double** 类型。
6. 建立一个多项式类 **Polynomial**，使一个一元  $n$  次多项式作为一个对象，如  $3x^2+6x+4$ 。一个  $n$  次多项式有  $n+1$  个系数，要求系数为 **float** 类型。
  - a) 建立适当的构造函数，由一个 **float** 数组和一个整数  $n$  确定一个多项式。注意动态分配内存。
  - b) 建立适当的拷贝构造函数、赋值操作函数、析构函数。
  - c) 建立一个成员函数，**double** `getValue(double x)`，返回多项式的值。注意，简化为  $n$  个一次式的计算。
  - d) 重载定义加法、减法、乘法，以支持两个多项式之间的加法、减法、乘法，以及一个多项式与一个 **double** 之间的乘法，分别适用左右操作数。
  - e) 重载定义==和!=运算符。
- 7 第 10 章介绍了一个 **float** 矩阵类 **FloatMatrix**，尝试用运算符重载函数来改进该类。
  - a) 重载定义加法和乘法运算符。
  - b) 重载定义==和!=运算符。

## 第13章 模板

模板(template)是 C++ 语言中类型参数化的一种机制。对于函数或类可将一些类型定义为类型形参,就能描述不同具体类型的共同的结构或行为,使程序具有通用性。当使用这些函数或类时,再将这些类型形参实例化,用具体类型代替,就能作用于某一种具体类型。本章介绍模板的概念,然后介绍函数模板和类模板,最后简单介绍标准模板库 STL 中的各种容器。

### 13.1 模板的概念

我们经常要设计多个重载函数来处理多种类型的数据,而计算语义和处理过程却相同。例如,下面几个函数对不同类型的形参求绝对值:

```
int abs(int x){ return x>0 ? x : -x; }
double abs(double x){ return x>0 ? x : -x; }
long abs(long x){ return x>0 ? x : -x; }
```

可以看出,这些函数的形参和返回值的类型各不相同,但函数体的实现完全相同,而且具有相同的语义,就是计算某一种类型数据的绝对值。能不能避免以上函数的重复定义?使用函数模板能解决这个问题。例如,定义一个函数模板如下:

```
template <class T> T abs(T x){return x>0 ? x : -x; }
```

这个函数模板就能取代前面 3 个函数,而且可能更多。

其中,template <class T>说明了一个类型形参 T。任何具体类型都可替代形参 T,包括基本类型和自定义类型,只要这些类型能支持函数体中所要求的计算  $x > 0$  和负号运算。

紧随其后的 `T abs(T x)`,就是利用类型形参 T 的一个函数模板,该函数的名称为 abs,形参和返回值都是 T 类型。之后是该函数的函数体,函数体中往往要对类型形参 T 的对象或数据进行操作,在运行时刻的实际类型要能支持这些操作。例如,上面函数体中对 x 做 > 计算和负号运算,如果实际类型为基本类型,就能支持这些计算,但如果是自定义类型,就需要定义运算符重载函数,否则编译出错。

模板是实现代码重用的一种常见方式,它通过将类型定义为形参,即类型的参数化,使一个函数或一个类能处理多种类型的数据,而无需为每一种具体类型都设计一个函数或一个类,从而实现代码重用。

使用模板的关键是用具体类型替代模板中的类型形参。对于上面定义的函数模板,就直接用实参的类型来替代类型形参 T。例如:abs(-34.5)调用中实参类型为 double,那么 T 就被替换为 double,这就相当于将函数模板进行了一次实例化,生成了一个 double abs(double x) 函数,再进行调用,所生成的函数被称为模板函数。

模板包括函数模板和类模板：

- 函数模板：包含类型形参、用于生成函数的模板，所生成的函数被称为模板函数。
- 类模板：包含类型形参、用于生成类的模板，所生成的类被称为模板类。

定义模板的一般格式如下：

```
template <模板形参表> 函数或类定义
```

其中，`template` 是关键字，说明后面定义的是一个模板。在模板形参表中至少包含一个类型形参，并用逗号隔开多个类型形参。每个类型形参以 `class` 或 `typename` 开头。`class` 和 `typename` 是关键字，说明模板的类型形参。类型形参的名字通常用大写单字母来表示，如 `S`、`T`，以区别于普通形参。用尖括号将类型形参表括起来。类型形参就像函数形参一样，可带缺省值，即指定一个缺省的具体类型。

类型形参之后可定义一个函数或一个类，可使用前面定义的类型形参，而且必须使用。例如，上面函数模板例子中，定义了一个类型形参 `T`，用于函数模板 `abs` 自己的形参和返回值。

## 13.2 函数模板

函数模板就是用 `template` 说明的函数，包含一个或多个类型形参。在函数模板中可用这些类型形参来说明形参或返回值，也可在函数体内使用。函数模板提供了强大的代码重用机制。

当发现要编写不同的函数对不同类型数据进行相同语义的处理时，就可以考虑使用函数模板。只要对函数模板定义一次，系统根据调用函数时提供的实参的类型，自动生成模板函数来处理具体类型的调用。

### 13.2.1 定义

下面通过一个例子来说明如何定义一个函数模板。定义一个求最大值函数的模板。

```
template <class T>
T max(T x, T y){    return (x > y) ? x : y; }
```

模板中定义了一个类型形参 `T`，并作为函数 `max` 的形参类型和返回类型。类型形参的作用域仅限于当前函数的范围。如果需要的话，函数体中也可使用类型形参。

函数体中要求对 `T` 类型的两个对象之间进行 `>` 运算，这样就要求 `T` 类型必须能支持 `>` 计算。所有的基本类型都能满足这个要求，但对于自定义类型，例如类，就需要定义相应的运算符重载函数。而且对于类，还要求提供公有的拷贝构造函数，来支持 `return` 语句执行。

在函数模板中，类型形参所说明的函数形参也能指定缺省值，例如，改变 `max` 如下：

```
T max(T x, T y=9)
```

函数形参 `y` 的缺省值为 9，但这并不意味着 `y` 的类型是 `int` 型，实际上，`y` 的类型取决于 `x` 的实参的类型。

函数模板可以定义在类外作为全局函数，也可定义在类中作为成员函数。函数模板也可以用来定义运算符重载函数。



### 13.2.2 模板实例化

调用函数模板与调用一般函数的形式相同，由函数名和实参表组成。当遇到一个函数模板调用时，编译器按如下过程来处理：

- 1、先按实参类型生成一个具体函数，并检查类型一致性；
- 2、然后用实参来调用该函数，完成编译。

由函数模板在调用时生成的具体函数称为模板函数，它是函数模板的一个实例。

例 15-1 调用函数模板。

```
#include <iostream>
using namespace std;
template <class T>
T max(T x, T y){ return (x > y) ? x : y; }
int main(void){
    int x1 = 1, y1 = 2;
    double x2 = 3.4, y2 = 5.6;
    char x3 = 'a', y3 = 'b';
    cout<<max(x1, y1)<< '\t';           //A
    cout<<max(x2, y2)<< '\t';           //B
    cout<<max(x3, y3)<<endl;            //C
    system("pause");
    return 0;
}
```

执行程序，输出如下：

```
2      5.6      b
```

在 A 行、B 行和 C 行调用模板函数时，编译器分别产生了函数模板的三个实例。A 行中用模板实参 `int` 对类型形参 `T` 进行实例化；B 行中用模板实参 `double` 对类型参数 `T` 进行实例化；C 行用模板实参 `char` 对类型参数 `T` 进行实例化。类型参数 `T` 被实例化后，编译器以函数模板为样板，对 A 行中的实例，生成如下模板函数：

```
int max(int x, int y)
{ return (x>y) ? x : y; }
```

并使 A 行中的 `max(x1, y1)` 来调用该函数。编译器对 B 行和 C 行作类同的处理。

一个函数模板是对具有相同操作的一组函数的抽象。因调用函数模板而生成的模板函数是一个具体的函数。函数模板与生成的模板函数之间的关系如图 13.1 所示。

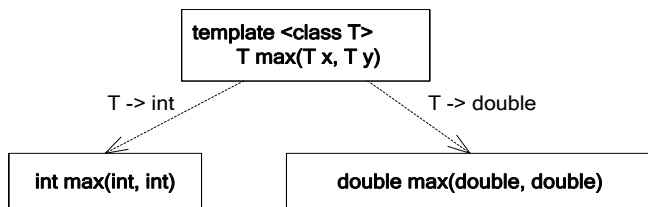


图 13-1 函数模板与其生成的模板函数

注意，调用 `max` 的两个实参的类型必须完全相同，否则就会出现类型形参的二义性错误。例如下面调用都是错误的：

```
max(3, 4.5);           //尝试生成 max(int, double)
max(3.0, 4);           //尝试生成 max(double, int)
max(3.1, 4.3f);        //尝试生成 max(double, float)
```

出错的原因是在第 1 步, 生成模板函数时就要检查两个形参类型是否一致, 如果不一致就会报错而停止编译, 而不会在第 2 步用隐式类型转换来完成编译。

在函数模板实例化过程中, 也可直接指定类型实参, 例如:

```
max<double>(3.1, 4.3f);    //生成 max(double, double)
```

显式确定类型形参 `T` 的实参为 `double`, 就可以生成一个 `max(double, double)` 模板函数, 那么隐式类型转换就可以起作用, 就能避免许多编译错误。

### 13.2.3 函数模板与有参宏的区别

函数模板在许多方面与有参宏相似, 有参宏也能实现一些函数模板所定义的功能, 但两者之间仍存在较大差别。例如, 求任意类型的两个值之间的较大值可以用一个有参宏实现如下:

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

这个宏可以替代函数模板, 使上面例子可运行。但在以下方面存在区别:

1、在宏展开时编译器不能检查类型兼容性。如果实参对象不支持 `operator>` 运算, 那么实参类型就不能进行宏展开, 而宏展开时却不能给出错误信息。

2、注意到两个宏参 `x` 或 `y` 将计算 2 次。如果调用方表达式中有自增或自减运算, 那么就会执行两次, 如: `cout<<max(x1, ++y1);` 的结果为 4, 而不是预料的 3。这是因为 `y1` 原值为 2, 在表达式中对 `y` 计算了 2 次。

3、由于宏扩展属于预处理, 编译是针对宏扩展之后的代码进行, 编译错误只能给出宏扩展之后的代码错误, 而不是宏定义自身的错误。另外在调试程序时也只能看到宏扩展之后的形式。

由此分析, 函数模板与有参宏相比较, 函数模板具有类型安全的优点。

### 13.2.4 模板函数的重载

模板函数与函数重载密切相关。从一个函数模板生成的多个模板函数都是同名的, 因此编译器采用重载方式来调用相应函数。

多个函数模板之间可以重载, 而且函数模板与普通函数之间也能重载。

当一个函数模板与一个同名的普通函数发生重载时, 编译器的处理过程比较复杂, 通过下面匹配过程来确定应该调用哪一个函数, 还是应该报错。

(1) 查找和使用最符合函数名和形参类型的函数调用, 即优先调用普通函数(即非模板函数)。

(2) 如果找不到普通函数, 编译器就检查是否可用从已有的函数模板中生成符合函数名和形参类型的模板函数。若找到, 则生成模板函数并调用。

(3) 如果没有找到合适的函数模板, 就对实参进行类型转换(此时实参对象类应提供类型转换函数), 然后再按照步骤 1 和 2 进行匹配。如果能匹配, 就进行调用。

(4) 如果仍找不到相匹配的一个函数, 就产生编译错误。如果找到多个函数, 那么这个函数调用就有二义性, 也是一种编译错误。

#### 例 13-2 模板函数重载。

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

template <class T>
T sum(T* array, int size){ //A
    T total=0;
    for(int i = 0; i < size; i++)
        total += array[i];
    return total;
}

template <class T1, class T2>
T2 sum(T1* array1, T2* array2, int size=0){ //B
    T2 total=0;
    for(int i=0; i<size; i++)
        total += array1[i] + array2[i];
    return total;
}

string sum(char* s1, char* s2){ //C
    return string(s1)+string(s2);
}

int main(){
    int iArr[]={1,2,3,4,5,6,7,8,9,10};
    double dArr[]={1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9,10.0};
    int iTotall = sum(iArr, 10); //D
    char *p1 = "Hello,";
    char *p2 = "everyone!";
    string s1 = sum(p1, p2); //E
    double dTotall1 = sum(dArr, 10); //F
    double dTotal2 = sum(iArr, dArr, 10); //G
    cout<<"The sum of integer array is: "<<iTotall<<endl;
    cout<<"The sum of double array is: "<<dTotall1<<endl;
    cout<<"The sum of double array is: "<<dTotal2<<endl;
    cout<<"The sum of two strings is: "<<s1<<endl;
    system("pause");
    return 0;
}
```

执行程序, 输出如下:

```
The sum of integer array is: 55
The sum of double array is: 59.5
The sum of double array is: 114.5
The sum of two strings is: Hello,everyone!
```

在 A 行和 B 行说明了两个函数模板, C 行是一个普通函数, 这三个函数具有相同的函数名 `sum`, 这三个函数之间是重载关系。D 行调用的实参为(`int *`, `int`)。首先尝试与 C 行说明的重载函数相匹配, 失败, 原因是第一个形参不相符。然后尝试与 A 行说明的函数模板相匹配, 相符合, T 的类型为 `int` 即可。与 B 行说明的函数模板不相符, 原因是第二个形参要求一个指针, 而 `int` 型不能转换为一个指针型。这样 D 行调用确定了一个函数模板, 而且没有二义性。

E 行调用与 C 行说明的普通函数完全相符, 虽然与 B 行函数模板也相符, 但普通函数优

先，因此没有二义性。

F 行调用也能确定 A 行的函数模板。

G 行调用的 3 个实参只能确定 B 行的函数模板，也没有二义性。

### 13.2.5 可变参量的函数模板

C11 标准支持可变参量的模板 *variadic template*。包括可变参数的函数模板和类模板。本节介绍函数模板。可变模板参数与可变函数参数(5.7.2 节)、可变宏参数(5.12.3 节)相似，在调用这种函数模板时，实参数量可变。

连续 3 个小数点...称为一个省略符，C 语言中表示可变参数，作为一个符号出现在函数形参表和宏参数列表中。现在省略符出现在模板的类型形参表中，表示可变参量的模板。格式如下：

```
template <class ... Arguments>
returntype functionname(Arguments... args);
```

其中 **Arguments** 表示可变参量的命名，下面函数形参表中使用。函数形参表中省略符之后要对参数包(parameter pack)给出一个命名。

例如：

```
template<class ... Args>
void print(Args ...rest); //rest 是参数包的名称
上面参数说明允许 0 个到多个类型参数。
```

如果至少需要 1 个参数，格式如下：

```
template <class First, class ... Arguments>
returntype functionname(First &first, Arguments... args);
```

可变参数只能作为形参表中最后一个元素。

例如：

```
template<class First, class ... Args>
void print(First & first, Args &... rest);
函数体中可使用 sizeof...运算符来返回参数包 rest 中还有多少参数。
```

函数模板往往以递归调用来处理每个 first 以及其余参数 rest。

下面是一个例子：

```
#include <iostream>
using namespace std;
template <class T> //A
void print(const T& t) {
    cout << t << endl;
}
template <class First, class... Rest> //B
void print(const First& first, const Rest&... rest){
    cout << first << ", --"<<sizeof...(rest)<<"--"; //C
    print(rest...); //D
}
int main(){
    print(1);
    print(10, 20);
    print(100, 200, 300);
```

```

    print("first", 2, "third", 3.14159, "你好");
    system("pause");
    return 0;
}

```

A 行说明一个单参模板函数，该函数在递归调用中最后一次调用时执行。

B 行说明一个至少 1 个参数的可变参数的函数模板。

C 行打印第 1 个参数，并显示剩余参与个数。

D 行递归调用，对剩余参数进行调用。

注意到 B 行...rest 和 C 行的 rest...之间的区别：前者表示将剩余多个参数打包并命名为 rest，后者表示将参数包 rest 解包。这是可变模板参数处理的特色。

主函数中对函数模板调用了 4 次，分别用 1 个、2 个、3 个和 4 个实参。

执行结果如下：

```

1
10, --1--20
100, --2--200, --1--300
first, --4--2, --3--third, --2--3.14159, --1--你好

```

可变参数模板多用于模板库的编程。

### 13.2.6 函数模板例子

我们经常要对不同类型的一维数组进行排序，如 int、float、double 数组。如果不用模板，就需要对不同类型的数组分别设计不同的函数来处理。下面我们用函数模板设计一个程序，可对任何类型的数组进行排序输出。

例 13-3 任意类型的数组的排序和输出。

```

#include<iostream>
using namespace std;

template <class T> void TSwap(T& a, T& b){           //任意类型数据之间的交换
    T c = a;
    a = b;
    b = c;
}

template <class T> void TSortArray(T* a, int count){//对 T 类型数组 a 元素排序
    for(int i = 0; i < count-1; i++)
        for(int j = i + 1; j < count; j++)
            if(a[i] > a[j])                        //要求 T 支持 operator >
                TSwap(a[i], a[j]);
}

template <class T> void TPrintArray(T *a, int count){//打印 T 类型数组元素
    for (int i = 0; i < count; i++)
        cout<<a[i]<<" ";
    cout<<endl;
}

int main(){
    int intA[10] = {15,8,0,-6,2,39,-53,12,10,6};
    TSortArray(intA, 10);
    TPrintArray(intA, 10);
    double doubA[10] = {15.4,8.3,0,-6.2,2.7,39.5,-53.5,12.6,10.4,6.4};
    TSortArray(doubA, 10);
    TPrintArray(doubA, 10);
    char charA[18] = "CharSortTest";
    TSortArray(charA, strlen(charA));
}

```

```

TPrintArray(charA, strlen(charA));
system("pause");
return 0;
}

```

执行程序，输出如下：

```

-53 -6 0 2 6 8 10 12 15 39
-53.5 -6.2 0 2.7 6.4 8.3 10.4 12.6 15.4 39.5
C S T a e h o r r s t t

```

分别定义了三个模板函数。第二个模板函数 `TSortArray` 采用了选择排序法，调用了第一个模板函数以交换 2 个元素。第三个模板函数用于打印任意类型的一维数组中的各个元素。

在主函数中我们测试了 3 种具体类型：`int`、`double`、`char` 数组，从输出结果可看出达到了升序排序的目的。

上面函数模板是否能作用于基于字符数组的字符串？例如，在 `main` 函数中添加下面代码：

```

char *strs[] = {"one", "two", "three", "four"};
TSortArray(strs, 4);
TPrintArray(strs, 4);

```

编译没有错误，但执行结果却是：`four three two one`

原因是在 `TSortArray` 函数中比较的并不是字符串的内容，而是字符串的地址，因此显示结果是地址的排序，而不是字符串内容的排序。

尝试 `<string>` 中的 `string` 类型。

```

string strs[] = { "one", "two", "three", "four" };
TSortArray(strs, 4);
TPrintArray(strs, 4);

```

执行结果为 `four one three two`，排序正确。

上面 3 个函数模板能正常工作需满足以下条件：

- 1、对象之间能比较大小，能支持 `>` 运算符，基本类型与 `string` 都支持 `>` 运算符。
- 2、对象交换时要调用赋值运算符重载函数。`string` 类型能提供。
- 3、在 `cout<<a[i]` 中要求该类提供转换到 `char*` 的类型转换函数。`string` 符合上面条件。

如果不用模板而要实现某种通用性函数设计，往往就要用 `void` 指针作为形参(第 8 章)。现在我们可用函数模板来实现通用性函数设计。模板具有类型安全的优点，也易于编写、易于理解。但模板的代价是其每个实例都要占用目标文件和可执行文件的空间，最终会占用较多内存空间。在内存紧张的实时性系统中，如嵌入式系统(例如手机编程)，使用模板编程往往会受到限制。另外模板的编译也比较费时。

### 13.2.7 完美转发

第 8 章介绍右值引用 `&&` 概念，第 10 章用于实现移动语义。右值引用 `&&` 可用于函数模板，可完美解决转发问题，以支持通用性函数模板的设计。

转发 `forwarding` 问题是什么？简言之，函数 `G` 中要将其左值引用形参转发给一个被调用函数 `F` 实参的协调性与灵活性的问题。

如果 `G` 的形参类型是 `const T&`，一个常量左值引用，那么 `F` 就不能改变参数。

如果  $G$  的形参类型是  $T\&$ ，一个非常量左值引用，那么  $G$  就不能用一个右值来调用，如临时对象、字面值、表达式等，显然影响其通用性。除非将所有临时对象和字面值都转为命名对象，再调用  $G$ 。这样命名太多，又缺乏灵活性，不完美。

如果确实需要这样的灵活性和通用性，一个方法是，为  $G$  函数编写 2 个重载形式，对应 2 种形参类型。但如果参数数量增加，那么重载数量将以指数级增长。比如 2 个形参就需要定义 4 个重载函数。这也不完美。

一种完美转发 **perfect forwarding** 的解决方案就是函数  $G$  采用右值引用  $\&\&$  做形参，仅需一个  $G$  函数版本即可，要用到 `std::forward(x)` 模板。

下面是一个关于对象工厂的例子。根据实参的不同类型来选择调用不同的类或结构的构造函数。简单起见，采用结构及其构造函数。

```
#include<iostream>
using namespace std;
struct W{ W(int&, int&) {} };
struct X{ X(const int&, int&) {} };
struct Y{ Y(int&, const int&) {} };
struct Z{ Z(const int&, const int&) {} };
template <class T, class A1, class A2>//第 1 个函数模板
T* factory(A1& a1, A2& a2){
    cout << "first factory" << endl;
    return new T(a1, a2);
}
template <class T, class A1, class A2>//第 2 个函数模板
T* factory(A1&& a1, A2&& a2){
    cout << "second factory" << endl;
    return new T(forward<A1>(a1), forward<A2>(a2)); //forward 需要<iostream>
}
int main(){
    int a = 4, b = 5;
    W* pw = factory<W>(a, b);    //A, (左值, 左值)
    X* px = factory<X>(2, b);    //B, (右值, 左值)
    Y* py = factory<Y>(a, 2);    //C, (左值, 右值)
    Z* pz = factory<Z>(2, 2);    //D, (右值, 右值)
    delete pw;
    delete px;
    delete py;
    delete pz;
    system("pause");
    return 0;
}
```

第 1 个函数模板的 2 个形参都是左值引用  $\&$ ，第 2 个函数的 2 个形参都是右值引用  $\&\&$ 。测试函数中 ABCD 有 4 种调用方式，变量作为左值，常量作为右值，有 4 种组合方式。执行程序，输出结果如下：

```
first factory
second factory
second factory
second factory
```

可以看出，只有 A 行调用执行了第 1 个函数模板，其它 BCD 都调用了第 2 个函数模板。将第 1 个函数模板去掉，重新编译执行，输出结果如下：

```
second factory
second factory
second factory
```

```
second factory
```

由此看出，基于右值引用&&的函数模板具有真正通用性，与 `forward` 搭配，只需一个版本就可完美解决转发问题，简化编码。

函数模板 `forward` 定义在 `<utility>` 或 `<iostream>` 中，专门与右值引用形参配合，将右值引用形参转换为实际类型，最后传给被调用函数做实参。

### 13.2.8 自动类型推导 `decltype`

C++ 是强类型语言，这意味着出现任何变量都要明确说明其类型。但在模板中，要说明函数模板的返回类型有时候就遇到麻烦。一些模板函数的返回类型与模板实参的类型相关。在这种情形下，要说明变量来存储中间结果也麻烦。因为这些变量类型在编码期不能确定，只有在编译期才能从模板实参中推导而来。为此 C11 提出 `auto` 和 `decltype` 两个解决方案，C14 针对函数返回类型推导又提出将两者相结合的一种简化方案。

前面章节介绍过 `auto` 的用法。`auto` 相对简单。在说明一个变量时用 `auto` 代替类型，让编译器从变量初始化表达式中自动推导出合适的类型，作为该变量的类型(第 2 章)，或者从该变量的上下文中推导，比如基于范围的 `for` 语句或者 `Lambda` 表达式中，从被循环的数组类型(或容器类型)来推导元素的类型。此外在使用 `auto` 时还可附加 `const/volatile`、指针\*、左值引用&、右值引用&&等，编译器将从初始化表达式并结合这些信息来推导变量类型。

尽管 `auto` 也可推导普通函数的返回类型，但难以推导函数模板的返回类型，除非返回类型与模板实参无关。

如何推导函数模板的返回类型，就需要 `decltype` 与 `auto` 相结合来解决此问题。

#### 1. 关键字 `decltype`

关键字 `decltype` 的语法形式就像一个函数调用：

**`decltype( expr )`**

返回表达式 `expr` 的类型，并用于说明后面的变量类型或者函数的返回类型。

表达式 `expr` 可以是一个标识符、或类成员，或函数(包括运算符重载函数)、左值或右值表达式。比如：

`auto a = 3;` 就等同于：

`decltype(3) a = 3; //int`

结构成员类型描述中不能用 `auto` 推导，但可用 `decltype`：

```
struct SS{
    int a;
    decltype(a) b;    //int
    decltype(3.4) c;  //double
};
```

`decltype(expr)` 规则：

(1) 如果 `expr` 是一个标识符或类成员，那么返回该实体的类型。



- (2) 如果 `expr` 是一个函数调用或运算符重载函数, 那么返回该函数的返回类型。
- (3) 如果 `expr` 是一个右值, 则返回其类型。如果是(左值), 则是该类型的左值引用。用 `decltype` 所推导出的类型可能与 `auto` 有所不同。看下面例子:

```
#include<vector>

int main(){
    const std::vector<int>v(1);
    auto a = v[0];           //a 为 int 类型
    decltype(v[0]) b = 0;    //b 为 const int&类型
    auto c = 0;              //c 为 int 类型
    auto d = c;              //d 为 int 类型
    decltype(c) e;           //e 为 int 类型, c 实体的类型
    decltype((c)) f = e;     //f 为 int&类型, 因为(c)是左值
    decltype(0) g;           //g 为 int 类型, 因为 0 是右值
    return 0;
}
```

注意, 用 `typeid` 来查看 `auto` 或 `decltype` 说明的变量类型可能不准确。

## 2. 另一种函数定义

`decltype` 与 `auto` 相结合, C11 提供了另一种函数定义形式:

**auto** funcName(params) **const/volatile**<sub>opt</sub> ->**decltype**(expr) **throw**()<sub>opt</sub>{body}

用 `auto` 替代返回类型, 用 `decltype(expr)` 中的 `expr` 来推导返回类型, 但要求 `expr` 应匹配函数体中的 `return` 表达式。这个用 `decltype` 说明的是一个所谓后定返回类型(late-specified)返回类型。

C14 对函数形式简化了 ->`decltype` 子句:

**decltype(auto)** funcName(params) **const/volatile**<sub>opt</sub> **throw**()<sub>opt</sub>{body}

通用性函数模板设计往往要满足几个条件: 1、能接收多种类型实参; 2 能将实参转发到被调用的包装函数; 3 能将包装函数返回值作为自己的返回值; 4 能支持自定义类型。

上一节完美转发例子说明了前 2 个条件是如何满足的: (1)采用非常量右值引用`&&`作为函数形参, 使之能接收字面值、表达式等多种形式作为实参; (2)调用 `std::forward` 模板将右值引用的形参还原为实际类型, 再转发给包装函数或构造函数。

针对返回类型的自动推导则需要上面介绍的新型函数形式。

下面是一个通用的 `Plus` 函数模板, 可实现基本类型和自定义类型的加法操作。

```
template<typename T, typename U>
decltype(auto) Plus(T&& t, U&& u){
    return forward<T>(t) + forward<U>(u);
};
```

采用 C14 的形式。其要点是, (1)用 `decltype(auto)` 说明返回类型是自动推导的; (2)返回类

型取决于形参 `t` 和 `u` 的实参类型，以及加法`+`运算符操作函数的返回类型。

如果 `t` 和 `u` 的实参类型都是基本类型，就无需定制运算符`+`操作函数。

比如 `Plus(2, 3.4)`，一个 `int` 加一个 `double`，结果类型为 `double`。加法是由基本类型实现的。

如果 `Plus` 要作用于一个用户自定义类型，该类型就需要定制一个运算符`+`操作函数。

下面是完整例子。

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

template<typename T, typename U>
decltype(auto) Plus(T&& t, U&& u){
    return forward<T>(t) + forward<U>(u);
};

class X{
    friend X operator+(const X& x1, const X& x2){    //友元函数实现运算符+
        return X(x1.m_data + x2.m_data);
    }
public:
    X(int data) : m_data(data) {}
    int Dump() const { return m_data; }
private:
    int m_data;
};

int main(){
    // Integer
    int i = 4;
    const int a = 2;
    cout << "Plus(i, 9) = " << Plus(i, 9) << endl;           //A
    cout << "Plus(i+3, a) = " << Plus(i+3, a) << endl;         //B
    cout << "Plus(a, 9) = " << Plus(a, 9) << endl;
    // Floating point
    float dx = 4.0;
    float dy = 9.5;
    cout << "Plus(dx, dy) = " << Plus(dx, dy) << endl;
    cout << "Plus(dx, i) = " << Plus(dx, i) << endl;
    // String
    string hello = "Hello, ";
    string world = "world!";
    cout << Plus(hello, world) << endl;
    // Custom type
    X x1(20);
    X x2(22);
    X x3 = Plus(x1, x2);
    cout << "x3.Dump() = " << x3.Dump() << endl;
    system("pause");
    return 0;
}
```

其中 `X` 类中用友元函数定义了一个运算符`+`重载函数，在其中 `return` 表达式中用显式类型转换返回同类对象 `X`，背后调用了自己的构造函数创建一个对象，然后返回。

执行程序，输出结果如下：

```

Plus(i, 9) = 13
Plus(i+3, a) = 9
Plus(a, 9) = 11
Plus(dx, dy) = 13.5
Plus(dx, i) = 8
Hello, world!
x3.Dump() = 42

```

分别测试了 `int`、`float` 的自加、互加，`string` 和类型 `X` 自加。

### 13.2.9 引用塌缩规则

第 8 章介绍了普通函数中非常量右值引用 `T&&` 做形参的规则。该规则限制了任何左值都不能做其实参。上面例子中的 `Plus` 函数的两个形参都是非常量右值引用 `T&&`，那么为何能容许 `i` 和 `a` 这样的左值作为其实参？看上面 A 行 B 行。

原因是在编译期由模板产生函数时，按一定规则进行了类型推导与转换，使所产生的函数形参具有相匹配的左值或右值。

表 13-1 模板形参 `T&&` 到函数形参的转换，以 `int` 为例

类型	<code>const int a=2;</code> a 常量左值	<code>int i=4;</code> i 非常量左值	9 常量右值	i+3 非常量右值
模板实参类型	<code>const int&amp;</code>	<code>int&amp;</code>	<code>int</code>	<code>int</code>
函数实参类型	<code>const int&amp;</code>	<code>int&amp;</code>	<code>int&amp;&amp;</code>	<code>int&amp;&amp;</code>

按上表，`Plus(i, 9)` 所产生的模板函数为 `int Plus<int&, int>(int&, int&&)`。

编译器由函数模板调用的实参类型，推导出模板实参类型和函数实参类型，所遵循的规则被称为引用塌缩规则 **reference collapsing rules**。完整规则如下表所示。

表 13-2 模板引用塌缩规则

序号	模板形参类型	调用实参类型—扩展	塌缩类型—结果
1	<code>T&amp;</code>	<code>&amp;</code> 左值	<code>T&amp;</code>
2	<code>T&amp;</code>	<code>&amp;&amp;</code> 右值	<code>T&amp;</code>
3	<code>T&amp;&amp;</code>	<code>&amp;</code> 左值	<code>T&amp;</code>
4	<code>T&amp;&amp;</code>	<code>&amp;&amp;</code> 右值	<code>T&amp;&amp;</code>

可以看出，上面例子用到第 3 和第 4 行规则，分别得到左值和右值。

该规则与 `decltype` 说明无关。该规则适用于引用类型做模板形参的所有情形。

## 13.3 类模板

类模板就是用 `template` 说明的类，包含一个或多个类型形参。在类模板中可用这些类型形参来说明数据成员的类型、构造函数或成员函数的形参或返回值，也可在函数体内使用。类模板提供了强大的代码重用机制。

### 13.3.1 定义

下面通过一个简单例子来说明如何定义一个类模板。

```
template<class T>
```

```

class Wrapper{
    T a;
public:
    Wrapper(const T a):a(a){}    //构造函数
    T get()const{return a;}      //成员函数
    void set(const T a);         //成员函数，下面定义如何实现
};
template<class T>
void Wrapper<T>::set(const T a){this->a = a;}

```

定义一个类模板用 **template** 开始，后跟一个或几个类型形参，用尖括号括起。类模板的名字是类名加类型形参。例如 **Wrapper<T>** 就是类模板的名字。这个名字在模板之外起作用。当我们阅读这个模板类时，可以读为“**Wrapper of T**”，即“**T 类型的包装类**”。

**Wrapper** 模板中将类型形参 **T** 用于说明类中的一个数据成员 **a**、构造函数的形参、以及成员函数的形参及返回类型。

如果将成员函数的实现放到类外定义时，就要用函数模板的格式进行定义，如上面成员函数 **set**。

实际上，类模板中的构造函数和成员函数都是函数模板，将类模板的类型形参作为自己的类型形参，而且可以添加自己的类型形参，只是注意取不同名字来区分，否则按支配规则会导致函数模板的形参隐藏了类模板的形参。

类模板的类型形参的作用域就是当前类。构造函数和成员函数自己的类型形参的作用域仅限于函数本身定义。

### 13.3.2 模板实例化

一个类模板是用来生成类的样板。类模板的使用就是将类模板实例化为具体的类，称为**模板类**，然后再通过模板类来创建对象、操作对象。

模板类格式：类模板名 <类型实参表>。

其中，<类型实参表>用尖括号括起，必须对类模板中的全部类型形参指定具体类型。如果有多个类型形参，要按定义次序排列，并用逗号分隔。如果某个类型形参有缺省值，在<类型形参表>中就可缺省。例如：**Wrapper<char>** 就是一个模板类，可作为一个普通类来使用。

创建模板类的对象的一种格式如下：

类模板名 <类型实参表> <对象 1>, ... , <对象 n>;

编译器根据类型实参替换类模板定义中的相应形参，生成一个具体的类，一个模板类，这个过程就是类模板实例化的过程。类模板只有被实例化后得到模板类才能创建对象，或者说明该类型的指针或引用。

例 13-4 用类模板创建对象。

```

#include <iostream>
using namespace std;
template<class T>
class Wrapper{
    T a;
public:
    Wrapper(const T a):a(a){}
    T get()const{return a;}
    void set(const T a);
};

```

```

template<class T>
void Wrapper<T>::set(const T a){this->a = a;}
int main(){
    Wrapper<char> c1('A');           //A
    Wrapper<int> i1(12);             //B
    Wrapper<double> d1(3.14);        //C
    cout<<c1.get()<<endl;
    cout<<i1.get()<<endl;
    cout<<d1.get()<<endl;
    system("pause");
    return 0;
}

```

执行程序，输出如下：

```

A
12
3.14

```

A 行、B 行和 C 行对类模板 `Wrapper` 建立了 3 个模板类 `Wrapper<char>`、`Wrapper<int>` 和 `Wrapper<double>`，并对这 3 个模板类分别创建了 1 个对象。

由类模板实例化的类，如 `Wrapper<int>`、`Wrapper<char>` 等，称为模板类。模板类的名字由类模板名字与其后尖括号 `<>` 括起的类型名构成。模板类与普通类一样使用，可以实例化创建对象，可以作为基类定义派生类，也能作为其它类的成员对象。类模板、模板类、对象之间的实例化关系如图 13.2 所示。

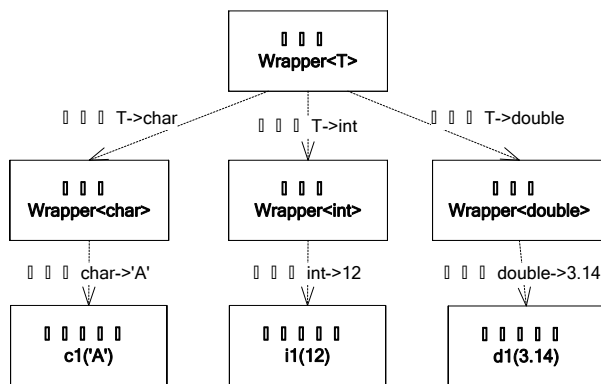


图 13-2 类模板、模板类与对象之间的关系

一个类模板可作为基类来定义派生模板，也可以作为复合类模板的成员。下面例子说明一个复合类模板。

#### 例 13-5 复合类模板的例子

```

#include <iostream>
using namespace std;

template<class T>
class Wrapper{
    T a;
public:

```

```

    Wrapper(const T a):a(a){}
    T get()const{return a;}
    void set(const T a);
};
template<class T>
void Wrapper<T>::set(const T a){this->a = a;}

template<class S>
class MyClass{
    Wrapper<S> obj;           //类模板 Wrapper 作为成员对象
public:
    MyClass(const S s):obj(s){}
    S get()const{return obj.get();}
    void set(const S s){obj.set(s);}
};

int main(){
    MyClass<double> ob1(3.14);
    double d = ob1.get();
    cout<<"d="<<d<<endl;
    ob1.set(2*d);
    cout<<"2*d="<<ob1.get()<<endl;
    system("pause");
    return 0;
}

```

执行程序，输出如下：

```

d=3.14
2*d=6.28

```

上面程序中 `MyClass<S>` 是一个类模板，将 `Wrapper<S>` 作为一个成员对象类，因此 `MyClass<S>` 是一个复合类模板。复合类模板的类型形参定义为 `S`，而成员类模板 `Wrapper<S>` 的形参由原先定义的形参 `T` 被替换为 `S`。当复合类模板实例化，`S` 被绑定一个具体类型时(如 `double` 型)，此时成员对象类模板也跟着实例化，绑定这个具体类型(如 `double` 型)。

在定义一个类模板时，可以利用已有的模板(作为基类或作为成员)，就像利用已有的类一样，只是要协调类模板的类型形参与被使用模板的类型形参之间的关系。

### 13.3.3 类模板的继承

模板类与普通类一样，也具有继承与派生关系。有 4 种形式。

#### 1. 普通类继承类模板

例如：

```

template<class T>
class TBase {
    T data;
public:
    TBase(const T a) :data(a) { cout << "TBase" << endl; }
    T get()const { return data; }
};

class Derived :public TBase<int> {

```

```
public:
    Derived(int a):TBase(a) { cout << "Driven" << endl; }
};
```

基类是类模板的一个实例，即一个模板类。

## 2. 类模板继承普通类

例如：

```
class TBase2 {
};
template<class T>
class TDerived2 :public TBase2 {
    T data;
public:
    TDerived2(const T a) :data(a) { cout << "TDrievd" << endl; }
    T get()const { return data; }
};
```

派生类添加了类型参数，成为类模板。

## 3. 类模板继承类模板

例如：

```
template<class T>
class TBase3 {
    T data1;
public:
    TBase3(const T a) :data1(a) { cout << "TBase3" << endl; }
    T get()const { return data1; }
};
template<class T1, class T2>
class TDerived3 :public TBase3<T1> {
    T2 data2;
public:
    TDerived3(const T1 a, T2 b) : TBase3(a),data2(b)
    { cout << "TDerived3: " <<a<<" "<<b<< endl; }
    T2 get2()const { return data2; }
};
```

派生类模板包含基类模板的类型参数，并可扩展新的类型参数。

## 4. 类模板继承类模板，类型形参做基类

例如：

```
#include<iostream>
using namespace std;
class BaseA {
public:
    BaseA() { cout << "BaseA" << endl; }
};
class BaseB {
public:
```

```

    BaseB() { cout << "BaseB" << endl; }
};
template<class T, int rows>
class BaseC {
private:
    T data;
public:
    BaseC() :data(rows) {
        cout << "BaseC:" << data << endl;
    }
    T get() { return data; }
};
template<class T>
class Derived :public T { //A 类型形参做基类
public:
    Derived() :T() { cout << "Derived" << endl; }
};
int main(){
    Derived<BaseA> x;// BaseA 作为基类
    Derived<BaseB> y;// BaseB 作为基类
    Derived<BaseC<int, 3> > z; // BaseC<int,3>作为基类
    cout << z.get() << endl;
    system("pause");
    return 0;
}

```

A 行类型形参作为基类，意味着任何类都可作为 `Derived` 的基类，换言之，`Derived` 类可作为任何类的派生类，这要求该类所描述的语义应该是多个类所共有的扩展，但现实中很少见，只是语法上允许存在。

### 13.3.4 可变参数的类模板

与函数模板一样，类模板也可变参。前面介绍可变参数的函数模板，类模板与此相似。可变参数的类模板的语法形式如下：

```

template<class ... Arguments>
class classname;

```

其中，`class` 之后的省略号表示可变参数，`Arguments` 作为一个参量包。例如：

```

#include<iostream>
#include<vector>
using namespace std;
template<class... Arguments> class vtclass {}; //A

vtclass< > vtinstance1; //B
vtclass<int> vtinstance2; //C
vtclass<float, bool> vtinstance3; //D
vtclass<long, vector<int>, string> vtinstance4; //E

```

A 行说明一个可变参数的类模板，`vtclass<...>`，下面 4 行分别说明了 0 个，1 个，2 个，3 个参数的模板类，并说明对象。

多数情况需要至少 1 个参数，语法形式如下：

```

template <class First, class... Rest>
class classname;

```

如果类模板持有可变参数，那么该类的构造函数或成员函数就需要消化这些可变参数。



这与可变参数的函数模板相同。参见 13.2.5 节。

## 13.4 标准模板库 STL

标准模板库 STL(Standard Template Library)是用模板定义的一组类和函数, 作为 ANSI/ISO C++标准的一部分。STL 定义了一组常用的数据结构, 如向量、链表、集合、映射、堆栈、队列等。这些数据结构都被称为容器 container。这是因为它们都是用来管理一组元素。STL 还定义了一组常用的算法, 如排序、查找、集合计算(如子集、差集、交集、并集等)。STL 为程序员提供了大量的类模板和函数模板, 只要包含相应头文件就能使用这些模板。采用 STL 编程的好处就是简化编程, 提高编程效率和程序质量, 避免了大量的底层的、重复性的编程。

### 13.4.1 容器

容器 container 是什么概念? 例如, 一个数组就是一个容器, 管理某种类型的一个元素序列, 每个元素都可按下标随机访问。数组作为容器的特点是连续空间、固定大小、随机访问。但不容易在数组中间插入或删除一个元素, 会导致后面元素逐个后移或前移。

STL 容器是一组类模板, 用来管理元素的序列。一个容器中的多个元素都具有相同类型, 可以是基本类型, 也可以是自定义类型。自定义类型一般要提供缺省构造函数、拷贝构造函数、析构函数和赋值操作函数, 一般还要提供运算符 `operator==` 和 `operator<`, 由此来支持其它 4 种关系运算符。例如要纳入集合 set 的元素。

容器可分为三大类: 序列容器(sequence container)、关联容器(associative container)和容器适配器(container adapter)。

- 序列容器能维护插入元素的顺序, 包括 vector 向量、deque 双端队列、list 链表等。
- 关联容器按预定义顺序管理所插入元素, 包括 set 集合、multiset 多集、map 映射、multimap 多射。序列容器和关联容器统称为第一类容器(first-class container)。
- 容器适配器是基于第一类容器而建立的简单容器, 包括 stack 堆栈、queue 队列、priority\_queue 优先级队列。各类容器的名称及特点见表 13.1。

表 13-1 各类容器的特点

容器名	中文名	特点	头文件	备注
vector<T>	向量	适合在序列尾端加入或删除元素。适合随机访问各元素	<vector>	随机访问按下标, 下标范围为 $[0-\text{size}()-1]$ 不适合中间插入或删除
deque<T>	双端队列	适合在序列两端加入或删除元素。适合随机访问各元素	<deque>	全称 double-ended queue 不适合中间插入或删除
list<T>	链表	适合在序列中间插入或删除元素。适合双向遍历元素	<list>	不适合随机访问。
set<T>	集合	各元素不重复 适合双向遍历	<set>	元素按值升序排序。各元素值唯一, 称为键 key。

				不适合随机访问。
<code>multiset&lt;T&gt;</code>	多集	可重复元素 可双向遍历元素	<code>&lt;set&gt;</code>	元素按值升序排序。各元素值不唯一，同值元素相邻。不适合随机访问。
<code>map&lt;K,V&gt;</code>	映射 单射	元素是对偶 <code>pair&lt;key, value&gt;</code> 一对一、多对一 键值不重复。适合双向遍历	<code>&lt;map&gt;</code>	各元素按键值升序排列。各元素的键是一个集合。
<code>multimap&lt;K,V&gt;</code>	多射 多值映射	元素是对偶 <code>pair&lt;key, value&gt;</code> 一对多、多对多 键可重复。适合双向遍历	<code>&lt;map&gt;</code>	各元素按键值升序排列。各元素的键是一个多集。
<code>stack&lt;T&gt;</code>	堆栈	先进后出 LIFO 不支持迭代器	<code>&lt;stack&gt;</code>	用成员函数来操作元素 基于 <code>deque</code> 实现
<code>queue&lt;T&gt;</code>	队列	先进先出 FIFO 不支持迭代器	<code>&lt;queue&gt;</code>	用成员函数来操作元素 基于 <code>deque</code> 实现
<code>priority_queue&lt;T&gt;</code>	优先队列	最高优先级先出 不支持迭代器	<code>&lt;queue&gt;</code>	用成员函数操作元素 基于 <code>vector</code> 实现

除了前面 3 大类容器之外，STL 还提供了 4 种“近容器 near container”：

- 数组 `array`。任何一个数组都可以看做是一个容器，下标可看作迭代器。
- `string` 字符串类型，从 `basic_string` 类模板。
- `bitset` 位集，任意长的二进制位。
- `valarray` 可变量数组，可支持高性能的数学矢量运算。

可以看出，不同的容器适合不同的用途，应根据所描述的事物特点来选用合适的容器。

容器中的元素可以是实体，也可以指针。例如，`vector<Person>` 中的元素就是对象实体，那么加入一个元素就是要执行 `Person` 的拷贝构造函数，删除一个元素就要执行析构函数。再如，`vector<Person*>` 中的元素就是 `Person` 对象的指针，那么写入一个元素就是复制一个指针，删除一个元素就是撤销一个指针。容器中的元素不能是对象的引用，例如 `vector<Person&>` 是不合法的。

在运行时刻一个容器是一个对象，具有特定的生命周期。例如，`vector<int> vint;` 语句就是创建了一个容器对象，元素类型为 `int`，容器的名字为 `vint`。创建一个容器要执行容器类的某个构造函数。当要撤销一个容器对象时，要先撤销其中每个元素，如果元素是对象实体，就要执行每个元素的析构函数，最后再执行容器的析构函数。

容器相关的操作可分为容器间操作和针对单个容器的操作。容器间的操作一般包括拷贝构造函数和赋值操作函数、6 种关系运算、`swap` 交换元素等。针对单个容器的操作主要是对其中元素的操作，包括插入元素、读取元素、删除元素等。

一个容器中的多个元素总是有序的，不同容器有不同的次序标准。对于序列容器，元素次序取决于加入元素的相对位置或前后次序。对于关联容器，元素次序取决于元素的值，如 `set` 中各元素按值升序排序，而 `priority_queue` 按各元素值降序排序。

对于一类容器，要访问各元素就需要使用迭代器。

### 13.4.2 迭代器

迭代器 `iterator` 是什么概念？以数组为例，访问一个数组元素要使用下标，如 `a[i]`，下标就是数组元素的迭代器。下标本质上是一个指针，指向你要访问的那个元素。迭代器是用来访问容器中元素的一种数据结构，就是指针的抽象，但更安全，功能更强。

在运行时刻，一个迭代器就是一个对象或者指针，但一般不用调用构造函数来创建迭代器对象，而是从已有容器对象中读取迭代器。假设有 `vector<int> vint;` 那么 `vint.begin()` 返回一个迭代器，它指向容器 `vint` 中的第一个元素，`vint.end()` 返回迭代器指向最后一个元素后面的空位置。一个容器 `c` 中的元素的位置范围就是 `[c.begin(), c.end())`。如果容器 `c` 中没有元素，那么 `c.begin()==c.end()`。迭代器往往作为函数形参，用前后两个迭代器 (`first` 和 `end`) 来确定一个序列范围 `[first, end)`，注意不包括 `end` 所指元素。

设 `it` 是一个迭代器 `iterator` 对象，而且指向某个元素，那么 `*it` 就是迭代器所指向的元素，`++it` 和 `it++` 就是后移指向下一个元素，`--it` 和 `it--` 就是前移指向前一个元素。

迭代器的作用如下：

- 1、用来访问容器中的各个元素，随机访问或按顺序访问。
- 2、作为 STL 算法与容器之间的中介，使 STL 算法能作用于容器中的元素。

迭代器定义在头文件 `<iterator>` 中，但如果你已包含了一种容器，就不需要再包含此文件。

迭代器的类别和功能如下表所示。

表 13-2 迭代器的类别和功能

类别	形参名	功能	备注
Output 输出	OutIt	向容器中写入元素。迭代器 <code>x</code> 仅对应一个值 <code>v</code> ，写入过程相当于 <code>*x++ = v</code> 。容器中添加了一个元素。	这里的输出是将元素写入容器，只能从前向后移动，而且只能遍历一遍。
Input 输入	InIt	从容器中读出元素。如果迭代器不等于 <code>end()</code> ，就指定了一个元素，可以多次读取， <code>v = *x</code> 。要读取下一个元素，就要递增， <code>v = *x++</code> 。容器中元素不变。	输入是从容器中读出元素。迭代器只能从前向后移动，而且只能遍历一遍。 一类容器都持有 <code>const_iterator</code> 和 <code>const_reverse_iterator</code> 分别用来正向和逆向读取元素。
Forward 正向	FwdIt	正向迭代器可取代输出迭代器用于写入元素，也可替代输入迭代器用于读出元素。用 <code>v=*x</code> 读出刚用 <code>*v=x</code> 写入的元素。可用多个迭代器独立操作。	一类容器都持有 <code>iterator</code> 类型，作为正向迭代器，从 <code>begin()</code> 到 <code>end()</code> 之前，自增后移，自减前移。
Bidirectional 双向	BidIt	双向迭代器 <code>x</code> 可取代正向迭代器，而且增加了前移操作， <code>v = *x--</code>	一类容器都持有 <code>reverse_iterator</code> 类型，作为逆向迭代器，从 <code>rbegin()</code> 到 <code>rend()</code> 之前，自增前移，自减

			后移。
Random 随机访问	RanIt	随机访问迭代器可取代双向迭代器，而且增加了用整数作为下标，就像数组下标，如 N 是一个整数，那么 $x[N]$ ， $x+N$ ， $x-N$	只有 vector 和 deque 支持随机访问

注意，这里的输出指的是元素写入容器，输出是从容器中读取元素。  
下图表示了 5 类迭代器之间的替代关系。

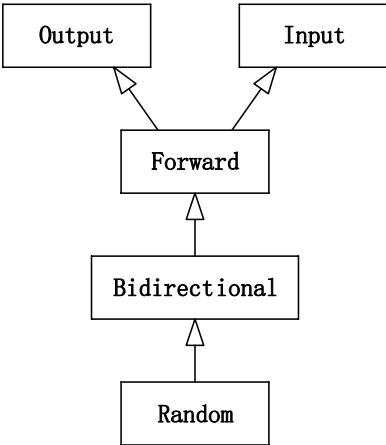


图 13-3 迭代器之间替代关系

根据上面替代关系，假如某个函数形参要求 Input 迭代器，那么你可使用除 Output 之外的任何一个作为函数调用的实参。

STL 定义了多种迭代器类模板，常用的类模板如表 13.3 所示。

表 13-1 常用迭代器模板

名称	说明
iterator<C, T, Dist>	迭代器基类
reverse_bidirectional_iterator<BidIt,T, Ref, Ptr, Dist>	逆向、双向迭代器，不能随机访问
reverse_iterator<RanIt,T, Ref, Ptr, Dist>	随机访问迭代器，可逆向、双向遍历
insert_iterator<Cont>	输出迭代器，支持 insert 函数，插入元素到任何位置
back_insert_iterator<Cont>	输出迭代器，支持 push_back 函数，添加到尾部
front_insert_iterator<Cont>	输出迭代器，支持 push_front 函数，插入到头部
istream_iterator<T, Dist>	输入流，从输入流中提取元素，可说明分隔符
ostream_iterator<T>	输出流，插入元素到输出流，可说明分隔符
istreambuf_iterator<E, T>	输入流，从输入流缓冲区中提取元素，可说明分隔符
ostreambuf_iterator<E, T>	输出流，插入元素到输出流缓冲区，可说明分隔符

上表中最后 4 种迭代器与输入输出流相关，一般需要自行调用构造函数来实例化，而其

它迭代器的实例化主要是通过具体容器的成员类型来进行的。例如，下面代码输出 `int` 数组中的各个元素：

```
ostream_iterator<int> output(cout, " "); //创建输出流迭代器
int a[5] = {5, 2, 3, 4, 1};
sort(a, a + 5); //升序排序
copy(a, a + 5, output); //输出 1 2 3 4 5
```

第 1 行创建了 `ostream_iterator` 对象，第 1 个实参 `cout` 是说明在 `<iostream>` 中的屏幕输出对象；第二个实参确定了输出对象之间的分隔符。第 3 行调用了 `<algorithm>` 中的 `sort` 函数：`void sort(RanIt first, RanIt last)`；该函数的两个形参都是随机迭代器，调用实参是 `int*`。第 4 行调用了 `<algorithm>` 中的 `copy` 函数，该函数的 3 个形参都是迭代器：`OutIt copy(InIt first, InIt last, OutIt x)`；而前两个调用实参是 `int*` 类型，确定了一个元素序列的范围。

### 13.4.3 容器的共同成员类型和操作

一类容器都定义了一些共同的成员类型，如下表所示。

表 13-3 一类容器中的共同成员类型(T 为元素类型)

类型名	说明
<code>allocator_type</code>	分配器类型，一般是 <code>allocator&lt;T&gt;</code>
<code>size_type</code>	表示大小的类型，一般是 <code>unsigned int</code>
<code>difference_type</code>	两个迭代器相减的结果的类型，一般是 <code>int</code>
<code>reference</code>	元素类型的引用， <code>T&amp;</code>
<code>const_reference</code>	元素类型的只读引用， <code>T const&amp;</code>
<code>value_type</code>	元素类型， <code>T</code>
<code>iterator</code>	正向迭代器类型
<code>const_iterator</code>	只读迭代器类型
<code>reverse_iterator</code>	逆向迭代器类型， <code>reverse_iterator</code> 或 <code>reverse_bidirectional_iterator</code>
<code>const_reverse_iterator</code>	只读、逆向迭代器类型， <code>reverse_iterator</code> 或 <code>reverse_bidirectional_iterator</code>

这些成员类型主要用于说明类中的成员函数的形参类型和返回类型。

这些成员类型都用 `typedef` 定义，在类外要使用这些类型，就要用作用域运算符 `::`。例如：`vector<int>::const_iterator it` 语句创建一个只读迭代器 `it`，其类型为 `vector<int>::const_iterator`。

一类容器都具有一些共同的操作函数，如下表所示。

表 13-4 一类容器的共同函数

函数	说明
缺省构造函数	通常一种容器还要提供几种构造函数
拷贝构造函数	复制容器中的所有元素
<code>operator=</code>	赋值操作函数，复制所有元素

析构函数	析构各元素，回收内存
<code>operator==</code>	相等，元素个数和各元素都相同，要调用 <code>size</code> , <code>equal</code> 函数
<code>operator!=</code>	不等
<code>operator&lt;</code>	小于
<code>operator&lt;=</code>	小于等于
<code>operator&gt;</code>	大于
<code>operator&gt;=</code>	大于等于
<code>iterator begin()</code>	返回迭代器指向头一个元素，如果无元素，则指向空位置
<code>const_iterator begin() const;</code>	<code>begin</code> 的只读版本
<code>iterator end();</code>	返回迭代器指向尾元素之后的空位置
<code>const_iterator end() const;</code>	<code>end</code> 的只读版本
<code>reverse_iterator rbegin();</code>	<code>begin</code> 的逆向迭代器版本
<code>const_reverse_iterator rbegin() const;</code>	<code>begin</code> 的只读、逆向迭代器版本
<code>reverse_iterator rend();</code>	<code>end</code> 的逆向迭代器版本
<code>const_reverse_iterator rend() const;</code>	<code>end</code> 的只读、逆向迭代器版本
<code>size_type size() const;</code>	返回当前元素个数
<code>size_type max_size() const;</code>	返回容器中可容纳最多元素个数
<code>bool empty() const;</code>	如果 <code>size()&gt;0</code> ，则返回 <code>true</code> ，否则 <code>false</code>
<code>iterator erase(iterator it);</code>	删除形参迭代器 <code>it</code> 所指向的一个元素，返回迭代器将指向剩下的头一个元素
<code>iterator erase(iterator first, iterator last);</code>	删除形参迭代器 <code>[first, last)</code> 范围中的所有元素 (不包括 <code>last</code> )，返回迭代器将指向剩下的头一个元素
<code>void clear();</code>	删除全部元素
<code>void swap(Cont x);</code>	当前容器与形参容器 <code>x</code> 进行元素交换

#### 13.4.4 算法<algorithm>

在<algorithm>中定义了 85 个容器相关的函数模板，如 `sort`、`find`、`copy` 等。

这些算法具有通用性，可作用于多种 STL 容器，也适用于数组、用户自定义类型。这些算法通过迭代器与被操作的容器关联，而且函数的形参往往就是某种迭代器。

从这些算法的名称的前缀或后缀可看出该算法的特点，例如：

- `is_xxx`，前缀 `is` 表示某种判断，如 `is_sorted`，不改变元素内容。
- `xxx_if` 后缀，如 `find_if`，针对元素是否特定条件来执行 `xxx` 操作。
- `xxx_copy` 后缀，如 `reverse_copy`，不仅执行操作，而且还将改变后的值复制到目标中。
- `xxx_n`，后缀，如 `copy_n`，指定元素数量的 `xxx` 操作。

从功能上可将算法分为 3 类，如下表所示。

表 13-5 算法分类<algorithm>

<b>1、不改变操作对象内容</b>
通用判断: all_of; any_of; none_of;
线性查找: find; find_if; find_if_not; find_end; adjacent_find; find_first_of
子序列匹配: search; search_n
计数: count; count_if
遍历: for_each
比较两个区间: equal; equal_range; mismatch; lexicographical_compare
最大与最小值: min; max; min_element; max_element; minmax; minmax_element
<b>2、改变操作对象内容</b>
复制区间: copy; copy_backward; copy_if; copy_n
互换元素: swap; iter_swap; swap_ranges
转换: transform
替换: replace; replace_if; replace_copy; replace_copy_if
生成与填充: fill; fill_n; generate; generate_n
移动: move; move_backward
移除元素: remove; remove_if; remove_copy; remove_copy_if; unique; unique_copy
逆序、轮转、排列: reverse; reverse_copy; rotate; rotate_copy; is_permutation; next_permutation; prev_permutation.
分割: is_partitioned; partition; partition_copy; partition_point; stable_partition
随机重排: shuffle; random_shuffle
<b>3、高级功能</b>
排序: is_sorted; is_sorted_until; sort; stable_sort; partial_sort; partial_sort_copy; nth_element;
二分查找: binary_search; lower_bound; upper_bound;
合并: merge; inplace_merge
有序区间的集合操作: includes; set_union; set_intersection; set_difference; set_symmetric_difference
堆操作: is_heap; is_heap_until; make_heap; push_heap; pop_heap; sort_heap;

注1， 一些算法与容器的成员方法一样。如<set>中 lower\_bound, upper\_bound 等。

注2， 函数形参所要求的迭代器类型决定类被操作容器的种类。例如，排序 sort 函数 void sort(RanIt first, RanIt last) 对[first, last)序列中的元素升序排序。该函数要求提供随机迭代器，而随机迭代器只有数组、向量 vector 和双端队列 deque 能提供，而其它容器则不能提供随机访问，例如 list 就不能随机访问，因此 list 自己提供成员函数 sort 来实现排序。

再如，查找 find 函数 InIt find(InIt first, InIt last, const T& val) 在[first, last)序列中查找等于 val 的元素，返回的迭代器指向等于 val 的第一个元素，如果未找到，就指向该序列的 end()。该函数要求提供输入迭代器，而所有一类容器迭代器都能提供，

因此该函数可作用于全部一类容器。

在 VS2015 文档中不仅详细解释这些算法函数,而且提供了部分例子,读者可自行尝试。

### 13.4.5 基于 C11 简化编程

C11 标准之后产生了一些新的编程模式,可简化编码。

方法 1、用基于范围 for 循环替代迭代器遍历。

方法 2、用 Lambda 表达式替代函数指针和命名函数。

例 13-6, 对 vector 容器分别执行并验证两个算法: random\_shuffle 和 partition。前者将容器中的元素次序打乱,称作“随机洗牌”。后者按某个条件(是否大于 5)将容器中元素分为满足和不满足两部分,称作“分割”。编码如下:

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;
bool greater5(int value) {                                //A 一元谓词函数
    return value > 5;
}
int main() {
    vector<int> v1;
    vector<int>::iterator Iter1;                          //B 迭代器变量
    for (int i = 0; i <= 10; i++){
        v1.push_back(i);
    }
    random_shuffle(v1.begin(), v1.end());
    cout << "Vector v1 is ( ";
    for (Iter1 = v1.begin(); Iter1 != v1.end(); Iter1++) //C 用迭代器遍历
        cout << *Iter1 << " ";
    cout << ")." << endl;
    // Partition the range with predicate greater5
    partition(v1.begin(), v1.end(), greater5);            //D 第 3 个形参
    cout << "The partitioned set of elements in v1 is: ( ";
    for (Iter1 = v1.begin(); Iter1 != v1.end(); Iter1++) //E 再次遍历
        cout << *Iter1 << " ";
    cout << ")." << endl;
    system("pause");
    return 0;
}
```

A 行函数有单个参数并返回逻辑值,称为一元谓词,描述了一个条件,将对容器中每个元素按此条件进行判断,得到一个真或假。

B 行定义迭代器变量, C 行和 E 行用传统 for 语句与迭代器遍历 vector 中每个元素。

D 行调用函数 partition 的第 3 个实参是 A 行定义的谓词函数。

上面代码是从 VC6 开始支持的。

执行程序,输出结果如下:

```
Vector v1 is ( 10 1 9 2 0 5 7 3 4 6 8 ).
The partitioned set of elements in v1 is: ( 10 8 9 6 7 5 0 3 4 2 1 ).
```

下面简化后的程序在 VS2015 或 DevC++5.11(GCC4.9.2)上执行:

```
#include <vector>
#include <algorithm>
```



```

#include <iostream>
using namespace std;
int main() {
    vector<int> v1;
    for (int i = 0; i <= 10; i++){
        v1.push_back(i);
    }
    random_shuffle(v1.begin(), v1.end());
    cout << "Vector v1 is ( ";
    for (auto y : v1)                                //A
        cout << y << " ";
    cout << ")." << endl;
    // Partition the range with predicate greater5
    partition(v1.begin(), v1.end(), [](int v) {return v > 5;}); //B
    cout << "The partitioned set of elements in v1 is: ( ";
    for (auto y : v1)                                //C
        cout << y << " ";
    cout << ")." << endl;
    system("pause");
    return 0;
}

```

A 行采用基于范围的 for 循环，替代了迭代器遍历，就省去了迭代器变量。C 行再次简化。

B 行采用 lambda 表达式描述分割条件，这样就省去了前面一元谓词函数。

简化后的程序中不再需要定义和操作迭代器变量。

此时发现 A 行与 C 行开始的 3 行是冗余编码，可以编写一个 print 函数，调用 for-each 函数遍历各个元素。简化程序如下：

```

#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;
void print(vector<int> &v) {
    for_each(v.begin(), v.end(), [](auto x) {cout << x << " "; });
    cout<< ")." << endl;
}
int main() {
    vector<int> v1;
    for (int i = 0; i <= 10; i++){
        v1.push_back(i);
    }
    random_shuffle(v1.begin(), v1.end());
    cout << "Vector v1 is ( ";
    print(v1);
    partition(v1.begin(), v1.end(), [](int v) {return v > 5; });
    cout << "The partitioned set of elements in v1 is: ( ";
    print(v1);
    system("pause");
    return 0;
}

```

可以看出，简化编程的效果是消除了传统迭代器的定义和使用。编程简单清晰，不易出错。

### 13.4.6 函数对象

C11 提出的函数对象。函数对象类是实现函数调用运算符 `operator()` 的类。该类的对象就是函数对象。函数对象也称为函子 `functor`。函数对象在调用 STL 算法时用于定制特殊行为。

前面介绍调用 STL 算法所使用的 Lambda 表达式, 实际上一个 L 式最终转换为一个函子。

调用 STL 算法除了迭代器往往还需要函子实参。函子实参有 3 种形式: L 式、函数指针和函数对象。其中函数对象是功能可扩展性最强一种方式, 在被调用时可产生或保存一些附加信息。

例如: 调用 `generate` 生成一组伪随机数, 你可以直接调用缺省形式:

```
int a[10];
```

```
generate(a, a + 10, rand);
```

直接调用 `rand` 函数产生随机数的范围很大, 而你需要的是 1-100 之间的随机数。

此时你可用一个 L 式来替代 `rand` 调用。

```
generate(a, a + 10, [] {return rand() % 100 + 1; }); //Lambda 表达式做实参来调用
```

也可定义一个命名函数, 然后再用函数指针来调用:

```
int getRand100() { return rand() % 100 + 1; }
```

```
...
```

```
generate(a, a + 10, getRand100); //函数指针做实参来调用
```

函数指针与 L 式的功能一样。L 式本身就是匿名函数, 省略了函数命名。

此时你又要统计生成多少个偶数, 又不原意做一次遍历, 希望在生成时就能完成统计。

此时 L 式与函数指针的编程方式就比较复杂了, 你可能要用到全局变量。而全局变量肯定不是好办法。

此时就需要建立一个函数对象类, 并用成员数据来表示偶数统计数, 实现函数调用运算符, 在生成数据时进行统计。

```
class MyRand {
    int &evenCount; //A 引用数据成员
public:
    MyRand(int& count) :evenCount(count) {} //B 构造函数, 参数是引用
    int operator()() { //C 函数调用运算符
        int a = rand() % 100 + 1;
        if (a % 2 == 0)
            evenCount++; //D
        return a;
    }
};
```

上面 `MyRand` 类就是一个函数类, 其中 C 行实现了函数调用运算符。

注意到 A 行是一个引用成员, 用于保存偶数的统计数量。相应地 B 行构造函数的形参也是一个引用, 将一个外部变量的引用赋值给 A 行的引用成员, 这样 D 行改变的就是外部变量。下面是函数调用:

```
int evencount = 0;
generate(a, a + 10, MyRand(evencount));           //函数对象做实参来调用
cout << "Even count is " << evencount << endl;
```

先创建函数对象，将外部变量绑定到函数对象，然后再执行 `generate` 算法。

通过上面例子可以看出，函数对象能保持计算状态，这是 L 式和函数指针所难以实现的。当需要深入定制 STL 算法调用的函数计算过程时，就需要函数对象，否则 L 式应做首选。

### 13.4.7 vector、deque 和 list

在 STL 容器中，`vector` 向量、双端队列 `deque` 和列表 `list` 具有相似性，它们都是管理一个元素序列，加入元素时要确定元素之间的相对位置。元素可以重复。每个元素都是一个值或一个对象，只是加入元素、访问元素和删除元素的方式不同。

#### 1.vector 向量

数学中的向量 `vector` 是既有大小、也有方向的量，也称为矢量。STL 中的向量 `vector` 与数学概念没有关系。一个 `vector` 管理一个元素序列，只是**加入和删除元素适合在序列的尾端操作**。也就是说，不适合在序列的头端或中间加入或删除元素，这是因为向量的内部实现采用连续内存空间，在头端加入一个元素将导致所有元素都后移，在头端删除一个元素将导致后面元素都前移，性能太低。

具体来说，加入一个元素最好调用 `push_back` 将其添加到尾端，用 `pop_back` 删除尾端元素，而不要调用 `insert` 来插入元素到任意位置，也不要使用 `erase` 函数来删除元素。

向量容器具有广泛应用。例如，超市柜台为一名顾客结账时，要逐件加入商品的品名、单价、件数及金额，新加入的商品信息总是在收据的尾端罗列。

例 13-8 建立一组人员，{姓名，性别，出生日期，电话}，执行特定条件查询(如 18-25 岁的男性)，特定条件排序(姓名，年龄，包括降序)，对年龄的各种聚合运算，如最小值、最大值、总和、平均值等，特定条件的计数，如男性人数。

本程序由 3 个文件组成：`date4.h` 包含 `Date` 类，`person.h` 包含 `Person` 类，`ex1308.cpp` 包含主函数。

```
//date4.h
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <time.h>
#include<iostream>
using namespace std;
class Date{
    int year, month, day;
public:
    Date(int y, int m, int d){           //构造函数 1
        year = y;
        month = m;
        day = d;
    }
```

```

Date(){
    time_t ltime = time(NULL);           //缺省构造函数
    tm * today = localtime(&ltime);
    year = today->tm_year + 1900;
    month = today->tm_mon + 1;
    day = today->tm_mday;
}
int getYear()const{return year;}
int getMonth()const{return month;}
int getDay()const{return day;}
int getInt()const { return year * 10000 + month * 100 + day; }
bool isLeapYear()const{
    return year%400 == 0 || year%4 == 0 && year%100 != 0;
}
int getAge()const{
    time_t ltime = time(NULL);           //取得当前时间
    tm * today = localtime(&ltime);       //转换为本地时间
    int cyear = today->tm_year + 1900; //取得当前年份
    int cmonth = today->tm_mon + 1;
    int cday = today->tm_mday;
    if (cmonth > month ||
        cmonth == month && cday >= day)
        return cyear - year;
    else
        return cyear - year - 1;
}
void print()const{
    cout<<year<<"."<<month<<"."<<day;
}
friend ostream & operator<<(ostream & os, const Date &d);
friend bool operator<(const Date & d1, const Date & d2);
bool operator==(const Date & d) {
    return this->getInt() == d.getInt();
}
};
ostream & operator<<(ostream & os, const Date &d) {
    os << d.year << "-" << d.month << "-" << d.day;
    return os;
}
bool operator<(const Date & d1, const Date & d2) {
    return d1.getInt() < d2.getInt();
}
}

```

相对于以前的Date类，添加了几个运算符重载函数。

```

//Person.h
#include"date4.h"
#include<string>
class Person {
public:
    enum Sex {FEMALE, MALE };
private:
    string name;
    Sex gender;
    Date birthdate;
    string phonenumber;
public:
    Person() {}
    Person(string name, Sex gender, int by, int bm, int bd, string phone)
        :name(name), gender(gender), birthdate(by, bm, bd),
        phonenumber(phone) {}
}

```

```

    string getName() const{ return name; }
    int getAge() const{ return birthdate.getAge(); }
    Sex getGender() const{ return gender; }
    Date getBirthdate() const { return birthdate; }
    friend bool operator<(const Person & p1, const Person & p2);
    bool operator==(const Person & p) {
        return this->name == p.name;
    }
    void print()const {
        cout << name << ", " << getAge() << ", "
            <<(gender==Sex::FEMALE ? "F" : "M")<< ", "<<phonenum<<endl;
    }
    friend ostream & operator<<(ostream & os, const Person &p);
};
bool operator<(const Person & p1, const Person & p2) {
    return p1.getName() < p2.getName();
}
ostream & operator<<(ostream & os, const Person &p) {
    os << p.name << ", " << p.birthdate << ", Age=" << p.getAge() << ", "
        << (p.gender == Person::Sex::FEMALE ? "F" : "M") << ", "
        << p.phonenum << endl;
    return os;
}

```

Person 类添加了几个运算符重载函数。下面是主程序。

```

#include "Person.h"
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<Person> roster;
    roster.push_back(Person("Fred", Person::MALE, 1980, 6, 20, "18974749275"));
    roster.push_back(Person("Jane", Person::FEMALE, 1990, 7, 15, "13968187475"));
);
    roster.push_back(Person("George", Person::MALE, 1991, 8, 13, "13983543485"));
);
    roster.push_back(Person("Bob", Person::MALE, 2000, 9, 12, "16392370175"));

    for (const auto &p : roster)
        cout << p;
    //条件查询
    cout << "Select MALE and 18-25:" << endl;
    for (auto &p : roster)
        if (p.getGender() == Person::MALE &&
            p.getAge() >= 18 && p.getAge() <= 25)
            p.print();
    //按姓名排序
    vector<Person*> sorted;
    for (auto &p : roster)
        sorted.push_back(&p);

    sort(sorted.begin(), sorted.end(),
        [](Person * p1, Person * p2) {
            return p1->getName() < p2->getName(); });
    cout << "Sorted by name:" << endl;
    for (auto p : sorted)
        p->print();
    //按年龄排序
    sort(sorted.begin(), sorted.end(),

```

```

        [](Person * p1, Person * p2) {
            return p1->getAge() < p2->getAge(); });
    cout << "Sorted by age asc:" << endl;;
    for (auto p : sorted)
        p->print();
    //按年龄逆序输出
    cout << "Sorted by age desc:" << endl;
    for (auto rit = sorted.rbegin(); rit != sorted.rend(); rit++)
        (*rit)->print();
    cout << endl;
    //逆序的另一种方式
    cout << "Another sort by age desc:" << endl;
    for (int i = sorted.size() - 1; i >= 0; i--)
        sorted[i]->print();
    cout << endl;
    //年龄最大值
    auto maxp = max_element(roster.begin(), roster.end(),
        [](Person &p1, Person &p2) {return p1.getAge() < p2.getAge(); });
    cout << "Max age is "; maxp->print();
    //年龄最小值
    auto maxp2 = min_element(roster.begin(), roster.end(),
        [](Person &p1, Person &p2) {return p1.getAge() < p2.getAge(); });
    cout << "Min age is "; maxp2->print();
    //男性人数
    int count1 = count_if(roster.begin(), roster.end(),
        [](Person &p1) {return p1.getGender() == Person::MALE; });
    cout << "Males count is " << count1 << endl;
    //年龄的总和与平均值
    int sum = 0;
    for_each(roster.begin(), roster.end(),
        [&sum](Person &p) { sum += p.getAge(); });
    cout << "Sum of age is " << sum << endl;
    cout << "Average of age is " << sum / 4.0 << endl;

    system("pause");
    return 0;
}

```

该程序大量采用简化方式编程，基于范围的 for 语句来避免迭代器操作，用 auto 变量简化编码，让编译器自动推导变量类型，用 Lambda 表达式替代命名函数。其中排序结果作为指针的 `vector<Person*>`，避免对实体对象交换，提高效率。

执行程序，输出如下：

```

Fred,1980-6-20, Age=36,M,18974749275
Jane,1990-7-15, Age=26,F,13968187475
George,1991-8-13, Age=24,M,1398354348
Bob,2000-9-12, Age=15,M,16392370175
Select MALE and 18-25:
George,24,M,13983543485
Sorted by name:
Bob,15,M,16392370175
Fred,36,M,18974749275
George,24,M,13983543485
Jane,26,F,13968187475
Sorted by age asc:
Bob,15,M,16392370175
George,24,M,13983543485
Jane,26,F,13968187475

```

```
Fred,36,M,18974749275
Sorted by age desc:
Fred,36,M,18974749275
Jane,26,F,13968187475
George,24,M,13983543485
Bob,15,M,16392370175

Another sort by age desc:
Fred,36,M,18974749275
Jane,26,F,13968187475
George,24,M,13983543485
Bob,15,M,16392370175

Max age is Fred,36,M,18974749275
Min age is Bob,15,M,16392370175
Males count is 3
Sum of age is 101
Average of age is 25.25
```

读者可参照文档容易理解和掌握。

## 2.deque 双端队列

双端队列 `deque` 与向量相似，采用连续内存，支持随机读取，只是**适合在序列两端添加或删除元素**。双端队列具有广泛用途。

例 1，超市里排队等待结账的顾客，新加入的顾客一般在队列尾端，而已结账的顾客离开队列一般都在头端。队列中的每个对象都知道自己处于“第几名”，这样就能随机访问。

例 2，在用户界面设计中事件是用户在键盘或鼠标上的操作，而系统处理这些事件总是将它们放在一个队列中，先来先处理。

例 3，一台网络打印机可为多名用户提供服务，但打印机要将多名用户提交的多个打印任务形成一个队列，先来先服务。虽然多名用户可并发提交打印任务，而实际上真正的处理是按队列来处理的。一个打印任务处理完才能继续进行下一个。

相对于向量 `vector`，`deque` 添加了下面两个成员函数：

```
void push_front(const T& x);    //插入元素 x 作为头元素
void pop_front();               //删除头元素
```

操作队列元素时，建议不要用 `insert` 函数将元素插入中间，或者用 `erase` 将中间元素删除。

可以看到，`vector` 能实现的，`deque` 都能实现。在实际编程中，`vector` 能被 `deque` 取代。

## 3.list 列表

列表 `list` 也称为线性表，基于双向链表，适合在任何位置插入或删除元素，适合各种排序，但**不支持按下标随机访问**，只能正向或逆向遍历。列表具有广泛用途。

例 1，一名银行客户 `Client` 拥有多个账户 `Account`，形成了一个列表 `list<Account>`。当删除一个账户时，很可能是中间元素，而不是两端元素。

例 2，一条公交汽车线路包含了一序列站点 `Stop`，有时需要在中间插入新站点，也可能

在两端延长新站点。撤销站点也可发生在中间元素。一条公交线路就是一系列站点的一个列表：`list<Stop>`

例 3，购物车上的货物。加入货物时保持次序，但可任意选取货物然后去掉。

相对于 `vector` 和 `deque`，`list` 不支持按下标访问，没有 `at` 或 `operator[]` 这样的成员函数。

要插入 `list` 元素，推荐使用下面 `insert` 成员函数：

```
iterator insert(iterator it, const T& x = T());    //将 x 插入到 it 之前
void insert(iterator it, size_type n, const T& x); //插入 n 个 x 到 it 之前
void insert(iterator it, const_iterator first, const_iterator last);
void insert(iterator it, const T *first, const T *last); //插入序列
```

下面成员函数加强对元素的操作功能：

```
void remove(const T& x); //删除等于 x 的多个元素，要调用 operator== 和析构函数
void unique(); //删除相同元素，要调用 operator== 和析构函数
void sort(); //按升序排序，要调用 operator<
void sort(Pred); //按二元谓词排序，注意不能调用 <algorithm> 中的 sort
void reverse(); //按逆序排列
```

序列中的元素常常是自定义类型的，如类或结构。一般来说，这些类型要提供构造函数、析构函数、拷贝构造函数和赋值操作函数。为了操作其中元素，还要求这些类型提供关系运算符操作函数，如 `operator==` 和 `operator>`，由这两个运算符可推出其它 4 种关系运算。

例 13-9，建立一组人员，{姓名，性别，出生日期，电话}，执行各种计算。比较特殊的是在序列中间要查找、删除中间元素、中间添加元素、实体排序等。基于 `Date` 和 `Person` 类，不赘述。下面是主程序。

```
#include "Person.h"
#include <algorithm>
#include <iostream>
#include <list>
using namespace std;
int main() {
    list<Person> roster;
    roster.insert(roster.end(), Person("Fred", Person::MALE,
1980,6,20,"18974749275"));
    roster.insert(roster.begin(), Person("Jane", Person::FEMALE,1990,7,15,
"13968187475"));
    roster.push_back(Person("George", Person::MALE, 1991, 8, 13,
"13983543485"));
    roster.push_back(Person("Bob", Person::MALE, 2000, 9, 12,
"16392370175"));
    for (auto &p : roster)
        p.print();
    cout << "Select MALE and 18-25:" << endl;
    for (auto &p : roster)
        if (p.getGender() == Person::MALE && p.getAge() >= 18 &&
p.getAge() <= 25)
            p.print();
    //按姓名查找
    auto cit = find_if(roster.begin(), roster.end(),
        [](Person &p) {return p.getName() == "George"; });
    if (cit != roster.end())
```



```

        cout << "George found" << endl;
    else
        cout << "George not found" << endl;
    //删除一个元素
    auto it = roster.erase(cit);
    cout << "George is erased" << endl;
    for (auto &p : roster)
        p.print();
    //添加一个元素
    roster.insert(it, Person("Tom", Person::MALE, 1999, 12, 12,
"1639232325"));
    cout << "Tom is inserted" << endl;
    for (auto &p : roster)
        p.print();
    //按年龄升序排序
    cout << "sorted by age asc:" << endl;;
    roster.sort([](Person &p1, Person &p2) {return p1.getAge() <
p2.getAge(); });
    for (auto p : roster)
        p.print();
    cout << endl;
    //按年龄降序排序
    cout << "sorted by age desc:" << endl;
    roster.reverse();
    for (auto p : roster)
        p.print();
    cout << endl;

    system("pause");
    return 0;
}

```

执行程序，输出如下：

```

Jane,26,F,13968187475
Fred,36,M,18974749275
George,24,M,13983543485
Bob,15,M,16392370175
Select MALE and 18-25:
George,24,M,13983543485
George found
George is erased
Jane,26,F,13968187475
Fred,36,M,18974749275
Bob,15,M,16392370175
Tom is inserted
Jane,26,F,13968187475
Fred,36,M,18974749275
Tom,16,M,1639232325
Bob,15,M,16392370175
sorted by age asc:
Bob,15,M,16392370175
Tom,16,M,1639232325
Jane,26,F,13968187475
Fred,36,M,18974749275

sorted by age desc:
Fred,36,M,18974749275
Jane,26,F,13968187475
Tom,16,M,1639232325

```

```
Bob,15,M,16392370175
```

链表中交换两个元素并不需要调用赋值操作函数或拷贝构造函数，而是交换元素结点来实现的，性能高。

列表 `list` 还提供了实现列表拼接与合并的成员函数：

```
void splice(iterator it, list& x); //将 x 中元素拼接到 it 前，保持原序，x 中元素删除
void splice(iterator it, list& x, iterator first); //单个元素拼接到 it 之前
void splice(iterator it, list& x, iterator first, iterator last); //拼接序列
void merge(list& x); //有序拼接，x 与当前表都按升序，结果也有序，x 中元素删除
```

成员函数 `unique(binPred)` 作用于一个升序列表，根据二元谓词所描述的条件（相等判断），移除相等元素，最后不重复的元素就构成一个集合 `set`。

### 13.4.8 set 和 multiset

集合 `set` 管理一组不重复的元素，而多集 `multiset` 中的元素可以重复。两者中元素经常称为键 `key`。在数学上，集合中的元素应该无序，但 STL 中 `set` 和 `multiset` 中的元素都默认按键值升序排序。

#### 1.set 集合

集合是使用最广泛的结构之一。

例 1，车管所管理机动车号牌组成一个集合，其中每个号牌都不重复。

例 2，一条公交线路所经过的各个站点名称应该是不重复的，能构成一个集合。实际上，一座城市中的所有站点名称都应该是唯一的，构成一个集合。

例 3，一个目录中的多个文件，文件名应该不重复，可按文件名排序，可在任何位置增删文件，也能改变文件名，改变之后仍按文件名重新排序。

集合 `set` 中的多个元素相互之间不同，因此在加入一个元素时，可能成功，也可能因已有相同元素而失败。加入元素要与已有元素比较是否相等，因此自定义类型应提供 `operator==`。加入一个元素 `x` 时不需要确定位置，只要调用 `insert(x)` 就行，它的位置取决于其值大小。

集合中的元素类型必须要能比较大小，因此自定义类型应提供 `operator<`。

例 13-10，建立一组人员，{姓名，性别，出生日期，电话}，执行各种计算。比较特殊的是在集合中间查找，并尝试替换一个元素，将验证新元素被自动排序，然后尝试添加一个重名元素，验证集合 `set` 将自动去掉重复。集合 `set` 不能像列表 `list` 任意排序，最多按逆序访问。

```
#include "Person.h"
#include <algorithm>
#include <iostream>
#include <set>
using namespace std;
void print(set<Person> r) {
    cout << r.size() << " elements:" << endl;
    for (const Person & p : r)
        cout<<p;
```

```

}
int main() {
    set<Person> roster;
    roster.insert(Person("Fred", Person::MALE, 1980, 6, 20, "18974749275"));
    roster.insert(Person("Jane", Person::FEMALE, 1990, 7, 15,
"13968187475"));
    roster.insert(Person("George", Person::MALE, 1991, 8, 13,
"13983543485"));
    roster.insert(Person("Bob", Person::MALE, 2000, 9, 12,
"16392370175"));
    print(roster);
    //按姓名查找
    auto cit = find_if(roster.begin(), roster.end(),
        [](const Person & p) {return p.getName() == "George"; });
    if (cit != roster.end())
        cout << "George found" << endl;
    else
        cout << "George not found" << endl;
    //删除元素
    auto it = roster.erase(cit);
    cout << "George is erased" << endl;
    print(roster);
    //尝试在删除位置添加元素, 希望替换
    roster.insert(it, Person("Tom", Person::MALE, 1999, 12, 12,
"1639232325"));
    cout << "Tom is inserted" << endl;
    print(roster); //新元素自动重新排序
    //尝试添加重名元素
    cout << "Try insert another Tom..." << endl;
    roster.insert(it, Person("Tom", Person::MALE, 1999, 12, 12,
"1639232325"));
    print(roster); //添加不成功
    //按姓名降序输出
    cout << "按姓名降序输出" << endl;
    for (auto rit = roster.rbegin(); rit != roster.rend(); rit++)
        cout << (*rit);
    system("pause");
    return 0;
}

```

集合 `set` 中的成员函数并未提供集合之间的计算, 如并集、交集、差集等。在 `<algorithm>` 中定义了一组函数模板来进行集合计算:

- 函数 `includes` 判断一个容器中的元素是否包含另一个容器元素, 即子集判断。
- 函数 `set_union` 将两个容器中元素合并, 即并集  $A \cup B$ 。
- 函数 `set_intersection` 计算两个容器中的共同元素, 即交集  $A \cap B$ 。
- 函数 `set_difference` 计算在第一个容器中有、而在第二个容器中没有的元素, 即差集  $A - B$ 。
- 函数 `set_symmetric_difference` 计算对称差, 即  $A - B \cup B - A$ 。

注意调用这些函数要求容器中元素按升序排序, 而且无重复, 否则结果不可预知。

## 2.multiset 多集

多集 `multiset` 与 `set` 相似，只是键可重复，但仍按键值排序。多集中相同值的元素位置相邻，相近值的元素相对集中。多集的特点就是元素可重复、按值升序排序。

多集中的元素往往来自其它容器，例如将两个 `set` 中的元素合并起来就是一个多集中。再将 `vector`、`deque` 或 `list` 中的元素加入到一个多集中，并保持元素个数，就形成了多集。

多集类模板与集合 `set` 并没有多大区别。集合可以看做是多集的元素无重复的一种特例。两个或多个集合中的元素合并在一起，如果有重复，就是一个多集。

对一组对象，如人员{姓名，姓名，生日，电话}，纵向取出某个属性值，如性别，就构成一个多集。

要将一个多集转换或生成一个集合，一般可调用 `<algorithm>` 中的 `unique` 函数，也可用 `set` 构造函数加入多集中所有元素。

由于多集中的元素可重复，因此一些成员函数呈现一些特点。例如：

成员 `find` 函数从前向后查找 `key`，返回迭代器指向第一个等于 `key` 的元素。

```
const_iterator find(const Key& key) const;
```

成员函数 `count` 函数返回的值可能大于 1，而对于集合，返回的值最大为 1。

```
size_type count(const Key& key) const;
```

由于多集允许重复而且自动排序，具有广泛用途。

下面代码计算一组得分数据(体操、跳水之类)，先去掉一个最低分和一个最高分，然后剩余求平均分：

```
multiset<int> mm({7,10,8,9,9,8,10});
int sum = 0;
auto f = mm.begin(); f++; //去掉最低分
auto t = mm.end(); t--; //去掉最高分
for_each(f, t, [&sum](auto x) {sum += x; });
cout << "The average is " << (double)sum / (mm.size() - 2) << endl;
```

### 13.4.9 map 和 multimap

前面介绍的容器元素都是单个值，而映射表示的是一个键 `Key` 到一个值 `Value` 之间的关系。STL 提供了两种映射，映射 `map` 和多值映射 `multiset`。

#### 1. map 映射

一个映射就是从一个键到一个值的对偶，表示为一个有序对(或对偶)`pair<key, value>`。一个 `map` 容器中包含了多个元素，其中每个元素都是一个有序对 `pair<key, value>`。一个 `map` 容器中的所有键 `key` 组成一个集合 `set`，即键不重复。映射表示从键到值的一对一或多对一的关系。在一个映射中，从一个键容易计算出其对应的值。映射具有广泛用途。

例 1，对于一个人员名单{姓名，性别，出生日期，电话}，如果在确定范围内姓名是唯一的，那么<姓名，性别>，<姓名，生日/年龄>，<姓名，电话>等等各种映射。

例 2，一路公交线路作为一个对象，对应一个 `list` 对象：`list<stop>`。那么 `map<line, list<stop>>`

就表示多条公交线路。

例 3，一个有效的汽车号牌对应一个人作为其车主。汽车号牌作为键，人作为值，那么从汽车号牌到车主就是一个映射。如果一个人拥有多辆汽车，就有多个不同的号牌都映射到同一名车主。这是一个一对多映射。这个映射的类型可表示为 `map<number, person>`。

例 4：一个投票统计系统中，假定候选人名不重复，那么从人名到得票数之间的映射就是一个 `map`。人名按字符串次序排列，而且不重复，每个人名都对应一个得票数。`map<string, int>`

例 5，对于任意一个字符串，统计各字符出现的次数 `map<char, int>`

在使用映射时，应注意以下要点。

1、在 `map<Key, T>` 中定义了从 `Key` 映射到 `T` 的 `pair<Key, T>` 作为其值类型：

```
typedef pair<const Key, T> value_type;
```

其中对偶 `pair` 的键值 `Key` 不能改变，对应的值可以改变。如果要改变键值就要先删除原对偶，再插入一个包含新键值的对偶。

2、在 `map` 类模板中有下标运算符重载函数，那么“映射对象[key]”就是键 `key` 所对应的值 `T` 引用，该表达式即可做右值，也可做左值。这是最简便常用的一种操作方式。例如：

```
map<string, int> myMap;
```

```
myMap["January"] = 1;    //加入 pair<"January", 1>
```

```
myMap["February"] = 2;   //加入 pair<" February", 2>
```

注意，用此方式读取一个不存在的键值，就会自动添加一个对偶<新键，缺省值>，而且返回。比如：`cout<<myMap["Sunday"]`将添加新对偶并返回输出 `int` 缺省值 0。

在用一个键 `k` 来查找映射之前，可用 `find` 成员函数先确定键 `k` 是否存在。例如：

```
if (myMap.find("Tom") == psex.end())
```

```
//do sth
```

3、对一个 `map` 可采用基于范围的 `for` 循环，例如：

```
for (auto const &ps : myMap)
```

```
    cout << ps.first << ":" << ps.second << endl;
```

其中 `ps` 就是一个对偶元素，`ps.first` 是键值，`ps.second` 就是对应的值。如果两者类型都支持 `operator<<`，就可如此输出。

注意，针对一个映射的基于范围 `for` 循环中如果要删除映射中的某个元素，编译无错，但运行时出错并中止程序。比如：

```
for (auto &y : myMap)
```

```
    if (y.second == 2)
```

```
        myMap.erase(y.first); //希望删除值=2 的首个元素，运行出错
```

4、如果 `it` 是映射 `map<Key, T>` 的一个迭代器，那么表达式“`*it`”是 `pair` 元素，“`(*it).first`”就是 `Key` 型值，“`(*it).second`”就是对应的 `T` 型值。

例 13-11. 建立一组人员，{姓名，性别，生日，电话}，执行各种计算。以人员形成的集合为基础，包含 6 个相对独立的小功能，每个功能都需要映射，作为中间结果或最终结果。主要采用的是集合 `set` 与映射 `map` 的成员函数，未涉及算法。

```

#include "Person.h"
#include <iostream>
#include <set>
#include <map>
#include <list>
using namespace std;

template <class T> //从多集转换到映射的通用函数
void multiset2map(const multiset<T> &from, map<T, int> &to) {
    for (auto y : from)
        if (to.find(y) != to.end())
            to[y]++;
        else
            to[y] = 1;
}

template <class T> //打印集合或映射的各个元素的通用函数
void print_collection(const T& t) {
    cout << t.size() << " elements: " << endl;
    for (const auto& p : t)
        print_elem(p);
    cout << endl;
}

template <class A, class B> //打印一个对偶的通用函数
void print_elem(const pair<A, B>& p) {
    cout << "<" << p.first << ", " << p.second << "> " << endl;
}

template <class T> //打印一个对象的通用函数, 要求支持 operator<<
void print_elem(const T & p) {
    cout<<p;
}

int main() {
    set<Person> roster;
    roster.insert(Person("Fred", Person::MALE, 1991, 8, 12,
"18974749275"));
    roster.insert(Person("Jane", Person::FEMALE, 2000, 9, 12,
"13968187475"));
    roster.insert(Person("George", Person::MALE, 1991, 8, 12,
"13983543485"));
    roster.insert(Person("Bob", Person::MALE, 2000, 9, 12, "16392370175"));
    print_collection(roster);
    //计算姓名与性别
    cout << "姓名到性别, 性别 1 为 Male, 0 为 Female: " << endl;
    map<string, Person::Sex> psex;
    for (auto & p : roster)
        psex[p.getName()] = p.getGender();
    print_collection(psex);
    cout << "Bob 's gender is " << psex["Bob"] << endl;
    cout << "Jane 's gender is " << psex["Jane"] << endl; //0
    //按姓名查找性别
    if (psex.find("Tom") == psex.end())
        cout << "Tom not found " << endl;
    else
        cout << "Tom's gender is " << psex["Tom"] << endl;
    //计算 20 岁以上的人员姓名及其生日
    cout << "20 岁以上的人员姓名及其生日" << endl;
    map<string, Date> pdate;
}

```

```

for (auto & p : roster) {
    if (p.getAge() >= 20)
        pdate[p.getName()] = p.getBirthdate();
}
print_collection(pdate);
//按性别分组, 计算各性别的人数
cout << "各性别的计数, 1 为 Male, 0 为 Female: " << endl;
map<Person::Sex, int> sexcount;
for (auto & p : roster)
    if (sexcount.find(p.getGender()) == sexcount.end())
        sexcount[p.getGender()] = 1;
    else
        sexcount[p.getGender()]++;
print_collection(sexcount);
sexcount.clear();
//创建多集, 调用自定义函数, 转换到映射
cout << "另一种方法计算各性别的计数: 多集转换映射" << endl;
multiset<Person::Sex> sexes;
for (auto & p : roster) sexes.insert(p.getGender());
MultisetToMap(sexcount, sexes);
print_collection(sexcount);

//用多集和映射计算相同生日的人
cout << "计算相同生日的人: " << endl;
multiset<Date> dates;
map<Date, int> bdcoun;
for (auto & p : roster) dates.insert(p.getBirthdate());
MultisetToMap(bdcoun, dates);
for(auto const &k : bdcoun)
    if (k.second != 1)
        for (auto const & p : roster)
            if (p.getBirthdate() == k.first)
                cout << p;
//按性别分组, 计算各性别的平均年龄
map<Person::Sex, int> sexagesum;
int malecount = 0;
for (auto & p : roster) {
    if (p.getGender() == Person::MALE) malecount++;
    if (sexagesum.find(p.getGender()) == sexagesum.end())
        sexagesum[p.getGender()] = p.getAge();
    else
        sexagesum[p.getGender()] += p.getAge();
}
sexagesum[Person::MALE] /= malecount;
sexagesum[Person::FEMALE] /= (roster.size()-malecount);
cout << "不同性别的平均年龄: 1 为 Male, 0 为 Female" << endl;
print_collection(sexagesum);

//按性别分组, 计算各性别的最大年龄的人的姓名、生日等信息
map<Person::Sex, const Person*> sexmaxage =
    { { Person::MALE, nullptr }, { Person::FEMALE, nullptr } };
for (auto & p : roster) {
    if(sexmaxage[p.getGender()]==nullptr ||
        p.getBirthdate() < sexmaxage[p.getGender()]->getBirthdate() )
        sexmaxage[p.getGender()] = &p;
}
cout << "各性别最大年龄: " << endl;
cout << *(sexmaxage[Person::FEMALE]);

```

```

    cout << *(sexmaxage[Person::MALE]);
    system("pause");
    return 0;
}

```

执行程序，输出如下：

```

4 elements:
Bob,2000-9-12, Age=15,M,16392370175
Fred,1991-8-12, Age=24,M,18974749275
George,1991-8-12, Age=24,M,13983543485
Jane,2000-9-12, Age=15,F,13968187475

姓名到性别，性别 1 为 Male，0 为 Female:
4 elements:
<Bob, 1>
<Fred, 1>
<George, 1>
<Jane, 0>

Bob 's gender is 1
Jane 's gender is 0
Tom not found
20 岁以上的人员姓名及其生日
2 elements:
<Fred, 1991-8-12>
<George, 1991-8-12>

各性别的计数，1 为 Male,0 为 Female:
2 elements:
<0, 1>
<1, 3>

另一种方法计算各性别的计数：多集转换映射
2 elements:
<0, 1>
<1, 3>

计算相同生日的人:
Fred,1991-8-12, Age=24,M,18974749275
George,1991-8-12, Age=24,M,13983543485
Bob,2000-9-12, Age=15,M,16392370175
Jane,2000-9-12, Age=15,F,13968187475
不同性别的平均年龄：1 为 Male,0 为 Female
2 elements:
<0, 15>
<1, 21>

各性别最大年龄:
Jane,2000-9-12, Age=15,F,13968187475
Fred,1991-8-12, Age=24,M,18974749275

```

上面例子中定义了一个函数模板 `multiset2map<T>`，将一个多集 `multiset<T>` 转换到一个映射 `map<T,int>`，计算每个 `T` 元素的出现次数。该函数模板具有一定通用性，下面例子就用到。



## 2. multimap 多射

多值映射 `multimap`，简称多射，与映射相似，只是键可重复。多射表示一对多、多对多关系，其中包含一对一、多对一关系作为特例。映射 `map` 相对于多射，可称作单射(数学概念)。单射是多射的一种特例。单射的键值逆转就是多射。多射中的键集构成一个多集。多射有很多用途。

例 1：银行客户与账户之间的关系。一个客户 `client` 作为键，其账户 `account` 作为值，一个客户可拥有多个账户，而一个账户只能对应一个客户，可表示为一个映射 `map<account, client>`，那么从客户到账户的映射就是一对多，可表示一个多射：`multimap<client, account>`

例 2：部门与员工之间的关系。一个公司有多部门，一个部门拥有多名员工作为其成员，而一名成员最多只能属于一个部门。那么部门作为 `key`，员工作为 `value`，从部门到员工就是一对多的多射：`multimap<department, person>`

例 3：学期教学计划中教师与课程之间关系。一名教师在一个学期可讲授多门课程，一门课程也可由多名教师讲授。那么从教师(`key`)到其所讲授课程(`value`)就是一个多对多的多射：`multimap<person, course>`，而且从课程(`key`)到其教师(`value`)也是一个多射：`multimap<course, person>`。

例 4，项目与成员之间的关系。一个项目有多名成员参与，而一名成员可同时参与多个项目。那么这就是两个多对多的多射：`multiset<project, person>`，`multiset<person, project>`

例 5：公交线路 `line` 与公交站点 `stop` 之间的关系。可用 `map<line, list<stop>>` 表示多条公交线路。一个公交站点 `stop` 可能有多条经过的线路 `line`，那么站点作为键，线路作为值，就是一个多对多的多射：`multimap<stop, line>`。显然后者可由前者推出。

多射类模板 `multimap` 与映射 `map` 相似，但因多射中的键可重复，没有提供下标运算符函数，就不能象映射那样用下标来直接操作。实际上，将一个映射 `map<K,V>` 中的键与值反转就得到一个多射。

例 13-12 实现一个交互式投票统计过程，逐个输入各选票上的候选人姓名，候选人姓名不重复，选票数量未知，最后统计各姓名出现的次数作为得票数，并按得票数降序输出。

程序如下：

```
#include <iostream>
#include <set>
#include <map>
#include <string>
#include <algorithm>
using namespace std;
template <class T>
void multiset2map(const multiset<T> &from, map<T, int> &to) {
    for (auto y : from)
        if (to.find(y) != to.end())
            to[y]++;
        else
            to[y] = 1;
}
int main() {
    multiset<string> strs;
    string MyBuffer;
```

```

while (1) {
    cout << "Input name('end' to stop):";
    cin >> MyBuffer;
    if (MyBuffer == "end")
        break;
    strs.insert(MyBuffer);
}
map<string, int> nameCount;
multiset2map(strs, nameCount);           //A 调用函数，将多集转换为映射
multimap<int, string> mm;
for (auto const &k : nameCount)
    mm.emplace(k.second, k.first);       //映射转换为多射
for (auto rit = mm.rbegin(); rit != mm.rend(); rit++) //多射的逆序
    cout << (*rit).second << " gets " << (*rit).first << endl;
system("pause");
return 0;
}

```

真正的计算从 A 行开始，输入数据进入多集，多集转换为映射，映射反转为多射，最后多射的逆序作为结果。

现实中有大量的多对多的关联关系，往往从一个方向具有已知条件，需要从另一个方向计算结果。多射往往作为中间结果。

例 13-13 已知多条公交线路中的各个站点，要求计算有哪些站点，以及各站点经过哪些线路，哪一个站点经过线路最多，等等。线路和站点都简化为字符串名称。假设所有站点名称唯一，而且在各条线路上都是统一的。

部分编程如下：

```

#include <iostream>
#include <string>
#include <list>
#include <map>
#include <iterator>
using namespace std;
using Line=list<string>;
using Lines=map<string, Line>;
void print(const Lines &lines) {
    for (auto const &line : lines) {
        cout << line.first << ": ";
        for (auto const &stop : line.second)
            cout << stop << ";";
        cout << endl;
    }
}
using StopToLine= multimap<string, string>;
int main(){
    Line line1 = { "s1", "s2", "s3", "s4" };
    Line line2 = { "s5", "s2", "s3", "s6" };
    Line line3 = { "s7", "s3", "s8", "s9" };

    Lines lines = { { "B1",line1 }, { "U2",line2 }, {"U3",line3} };
    print(lines);

    StopToLine s2l;
    for (auto const &line : lines)
        for (auto const &stop : line.second)
            s2l.emplace(stop, line.first);
}

```

```

    for (auto const &s : s2l)
        cout << s.first << ":" << s.second << endl;

    map<string, list<string>> result;
    for (auto const &s : s2l) {
        if (result.find(s.first) == result.end()){
            list<string> ls;
            ls.push_back(s.second);
            result[s.first] = ls;
        }else
            result[s.first].push_back(s.second);
    }
    print(result);

    system("pause");
    return 0;
}

```

执行程序，输出如下：

```

B1: s1;s2;s3;s4;
U2: s5;s2;s3;s6;
U3: s7;s3;s8;s9;
s1:B1
s2:B1
s2:U2
s3:B1
s3:U2
s3:U3
s4:B1
s5:U2
s6:U2
s7:U3
s8:U3
s9:U3
s1: B1;
s2: B1;U2;
s3: B1;U2;U3;
s4: B1;
s5: U2;
s6: U2;
s7: U3;
s8: U3;
s9: U3;

```

计算过程，先用映射表示已知条件，其中每个对偶的值是一个 `list<stop>`，这样就能表示多对多关联。下一步是将映射转换为一个多射，表示站点与线路的对偶。最后一步是将多射再转换为单射，作为最终结果。

以前采用 `typedef` 定义 `list<string>` 的同义词 `Line`，C11 之后可用 `using` 来定义。

#### 13.4.10 stack、queue 和 priority\_queue

STL 还提供了堆栈 `stack`、队列 `queue` 和优先级队列 `priority_queue`，这些容器也被称为容

器适配器，依赖前面一类容器而定义。这些容器的特点是使用简单，没有迭代器，不能遍历所有元素。

这些容器都具有一些共同的特征，例如，都支持 6 种关系运算，还包括下面共同特征：

allocator_type	成员类型，内存分配器类型
value_type	成员类型，值类型，T
size_type	成员类型，大小类型，一般为 unsigned int
缺省构造函数	创建空容器
value_type size();	返回当前元素个数
bool empty();	是否为空，即 size() == 0
void push(const value_type&x);	将元素 x 加入到特定位置
void pop();	将特定位置的元素删除
value_type &top();	头端元素的引用

### 1.stack 堆栈

后入先出 LIFO，用 push(x)将元素 x 加入序列尾部，用 top()访问尾部元素，并用 pop()将尾部元素删除。

### 2.queue 队列

先入先出 FIFO，用 push(x)将元素 x 加入序列尾部，用 front()访问头部元素，并用 pop()将头部元素删除。

### 3.priority\_queue 优先级队列

一种有序的队列，也是先入先出，只是值最大的先出，输出序列按值降序排列。用 push(x)将元素 x 加入到序列中特定位置，相当于插入排序。用 top()访问头端、值最大的元素，并用 pop()将头端元素删除。

注意，在执行 pop 函数时应确保容器非空，否则将导致不可预料的错误。

例 13-14 堆栈、队列和优先级队列的例子。

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

void test1(){
    cout<<"testing stack-----"<<endl;
    stack<int> intstk;
    intstk.push(1);
    intstk.push(2);
    intstk.push(3);
    cout<<"now size is "<<intstk.size()<<endl;
    while(!intstk.empty()){
        int i = intstk.top();
        intstk.pop();
        cout<<"pop top = "<<i<<"; now size is "<<intstk.size()<<endl;
    }
}

void test2(){
    cout<<"testing queue-----"<<endl;
    queue<int> intque;
    intque.push(1);
    intque.push(2);
```

```

    intque.push(3);
    cout<<"now size is "<<intque.size()<<endl;
    while(!intque.empty()){
        int i = intque.front();
        intque.pop();
        cout<<"pop front = "<<i<<"; now size is "<<intque.size()<<endl;
    }
}

void test3(){
    cout<<"testing priority_queue-----"<<endl;
    priority_queue<int> intque;
    intque.push(1);
    intque.push(3);
    intque.push(2);
    cout<<"now size is "<<intque.size()<<endl;
    while(!intque.empty()){
        int i = intque.top();
        intque.pop();
        cout<<"pop top = "<<i<<"; now size is "<<intque.size()<<endl;
    }
}

int main(){
    test1();
    test2();
    test3();
    system("pause");
    return 0;
}

```

执行程序，输出如下：

```

testing stack-----
now size is 3
pop top = 3; now size is 2
pop top = 2; now size is 1
pop top = 1; now size is 0
testing queue-----
now size is 3
pop front = 1; now size is 2
pop front = 2; now size is 1
pop front = 3; now size is 0
testing priority_queue-----
now size is 3
pop top = 3; now size is 2
pop top = 2; now size is 1
pop top = 1; now size is 0

```

在实际编程中，此类容器往往作为基础性数据处理机制，往往不能作为计算结果。如果这些容器够用，就无需使用第一类容器。

## 13.5 小 结

- 模板是对具体类型的抽象，有模板的函数或类可处理多种具体类型的数据，而无需重复编程，这样模板就具有一定的通用性，模板编程的目的就是增强可重用性。
- 定义模板的格式：`template <class T>`，这两个关键字加上尖括号就是模板的独特形式。

其中 **T** 被称为该模板的类型形参。

- 模板包括函数模板和类模板：
  - 函数模板：用于生成函数的模板，生成的函数被称为模板函数。
  - 类模板：用于生成类的模板，生成的类被称为模板类。
- 函数模板中函数形参或返回值往往要使用类型形参。函数体中也可使用类型形参，但应考虑将来具体类型能否支持所需要的计算。函数模板的一个类型形参的作用域就是从定义开始到函数体结束。
- 函数模板的实例化需要两个步骤：先根据实参类型生成一个具体函数(可以是隐式的，也可以显式确定类型实参)，再用实参值来调用该函数，再完成编译。
- 模板具有易编写、易理解、类型安全的优点，但也有费时费空间的缺点。
- 函数模板与函数重载具有本质的联系。多个函数模板之间可以重载，而且函数模板与普通函数之间也能重载。
- 类模板中含有类型形参，类型形参可用来说明数据成员，也能在构造函数或成员函数中作为形参或返回类型，也可以在函数体内使用。类模板的类型形参的作用域就是从定义开始到类体结束。
- 类模板的实例化就是将其类型形参绑定具体类型，得到具体类型，再完成编译。
- 类模板适合实现任意类型数据的某种集聚形式，如数组、堆栈、队列、列表(链表)、向量等。这些类也称为容器类 **Containers**。介绍了一个堆栈和一个矩阵。
- **STL** 是一组类模板和函数模板，作为 **ANSI/ISO C++** 草案标准的一部分。**STL** 中最重要的部分是一组容器、用来访问容器元素的一组迭代器、以及操作容器元素一组算法函数。使用 **STL** 可以简化编程，提高编程效率和程序质量。

## 13.6 练 习 题

1. 对于模板，下面哪一种说法是错误的？
  - A 函数模板的作用是使一个函数能操作多种不同类型的数据，而不用重复定义。
  - B 类模板的作用是使一个类能处理多种不同类型的数据，而不用重复定义。
  - C 用 **template** 定义的类型形参，在后面的函数或类中必须使用。
  - D 用 **template** 定义的类型形参表中可以为空。
2. 对于函数模板，下面哪一种说法是错误的？
  - A 一个函数模板是一组同名函数的抽象。
  - B 从一个函数模板生成的一个函数作为该模板的一个实例。
  - C 函数调用的实参决定了模板形参的具体类型。
  - D 在一个函数模板中不能调用另一个函数模板。
3. 写出以下程序的运行结果。

```
#include <iostream.h>
template<class T>
void f1(T t){ cout<<"t="<<t<<endl; }
```

```
void f1(double d){ cout<<"d="<<d<<endl;}
void main(){
    f1('A');
    f1(3);
    f1(3.14);
    f1(3.14f);
}
```

4. 分析下面程序的编译结果。

```
#include <iostream.h>
template<class T>
void f1(T *t){cout<<"*t="<<"*t"<<endl;}
template<class T>
void f1(T t){cout<<"t="<<t<<endl;}
void main(){
    int i = 3;
    f1(i);           //A
    f1(&i);          //B
}
```

A 程序编译没有错误。      B 第二个函数模板有错。

C A 行有错                      D B 行有错

5. 在第 12 章的 12.3.3 节介绍了一个可变长的 float 数组类 FloatArray。以此为基础, 尝试建立一个模板类 TArray, 使该数组的元素为任意类型 T。该类模板定义如下:

```
template<class T>
class TArray{
    int len;                //元素个数,即数组的长度
    T *arp;                 //指向动态的一维数组
    static T defV;          //缺省值,静态数据成员
    void copy(TArray<T> &fa); //将 fa 复制到当前对象
public:
    TArray(int n=0);        //缺省构造函数
    TArray(TArray &);       //拷贝构造函数
    TArray & operator=(TArray &); //赋值操作函数
    ~TArray(){ if (arp) delete [ ] arp; } //析构函数
    int getLen() const{ return len; } //读取长度,下标范围[0..len-1]
    void setLen(int newLen); //增加长度
    T & elemAt(int index);   //按下标访问元素
    T & operator[] (int index); //按下标访问元素
    static void setDef(T &def){defV=def;} //设置缺省值
};
```

先按以上定义完成各函数的编程。然后测试一种基本类型(如 int)和一种自定义类(如 Point)作为该数组的元素。

6. 在前面 TArray 模板类的基础上, 建立一个 TQueue 模板类, 表示任意类型的队列, 实现 FIFO, 即先入先出。完成编程并测试。

7. 在 13.3.4 节介绍了一个矩阵类模板 TMatrix, 它是基于指针和动态内存实现的。尝试基于 vector 向量提供一种实现, 要求不改变 TMatrix 的公有函数的定义, 使原有测试代码能通过测试。

## 第14章 输入输出流

输入/输出(Input/Output, 简称 I/O)是指程序与计算机外部设备之间所进行的信息交换。输出就是将对象或值转换为一个字节序列后在设备上输出, 如显示器、打印机; 输入就是从输入设备上接收一个字节序列, 然后将其转换为程序能识别的格式后赋给对象或值, 如键盘输入一个浮点数。接收输出数据的地方称为目的 target, 而输入数据的来源称为源 source。I/O 操作可以看成是字节序列在源和目的之间的流(stream)。将执行 I/O 操作的类称为流类, 实现该流类的体系称为流类库。C++提供了功能强大的流类库。本章主要介绍流类库提供的格式化 I/O 和文件 I/O。

### 14.1 概述

虽然 VC++语言没有专门的 I/O 语句, 但提供了三套 I/O 函数库或类库。第一套是用 C 语言实现的 I/O 库函数(大多定义在<stdio.h>和<conio.h>中), 在 C++程序中可用但不推荐。第二套是 I/O 流类库, 在非 Windows 编程中推荐使用, 前面章节就用它。第三套是为 Windows 编程提供的 I/O 类库。本章主要介绍第二套 I/O 类库。

#### 14.1.1 流 Stream

I/O 逻辑设备的概念。计算机系统的 I/O 设备繁多, 不同 I/O 设备的操作方式不同。为了简化操作, C++提供了逻辑设备的抽象概念。逻辑设备的操作方式是相同的, 如从逻辑设备上读取数据, 将数据写入逻辑设备等, 按照相同的方式来操作以简化 IO。由于逻辑设备采用统一的操作方式, 程序员容易理解和使用。将逻辑设备的操作转换成物理设备的 I/O 操作是由 I/O 流自动完成的。例如, 同一个写操作可以实现对一个磁盘文件的写操作, 也可以实现将输出信息送向显示器显示, 还可实现将输出信息送打印机打印。

流可分为文本流和二进制流。

- 文本流(text stream)是一串按特定字符规范编码的字符流, 除了最通用的 ASCII 之外, 还有 Unicode、GB2312、GBK 等规范。例如源程序(如.h 和 .cpp 文件)、HTML、XML、txt 文件都是文本文件。凡是用 Windows 记事本能正确打开显示的文件都是文本文件。文本文件的最后一个字节可以是 0 值, 而中间其它地方不能有 0 值(0 值作为文本文件结尾标志)。文本文件可作为文本流直接输出到显示器或打印机。
- 二进制流(binary stream)是以二进制形式存储数据。例如可执行文件(如.exe 文件)、目标文件和库文件(如.obj, .lib, .dll 文件)、多媒体文件(如.jpg、.mp3、.avi 等)、文档文件(如.doc、.xls、.ppt)都是二进制文件。这种流在数据传输时不需作任何变换, 但



不能直接输出到显示器或打印机，二进制流往往需要特殊程序处理。

为何要区分文本流和二进制流？原因是对它们的处理方式不同，文本流的基本单位是字符，字符要符合一定的编码标准和格式控制，而且可出现的字符都是可见字符。而二进制流的基本单位是字节，如一个 `int` 值要占用固定的 4 个字节，它依据的内存数据的标准，处理二进制流与处理内存数据基本一样。

流是按前后次序进行操作的。这意味着，要在流的前面或中间插入一个数据，或者删除流中已输出的某个数据，都是很困难的，甚至不可行。对于文件流，利用随机访问能更改或覆盖文件流中的已有数据，但也不能插入或删除。

### 14.1.2 文件

抽象文件与具体文件。流是 C++ 对所有 I/O 设备的逻辑抽象，而文件(file)是对具体设备的抽象。例如一个源程序是一个文件，而一个键盘、一台显示器、一台打印机也可分别看作一个文件，磁盘文件是一种具体的文件。把设备看作文件来读写，就能用统一的文件操作方式来实现不同设备的 I/O，具有基本相同的操作方式，简化的流的操作。

对不同类型的文件可进行的操作不同。例如对于磁盘文件，可将数据写入文件中，也可以从文件中取出数据。对于打印机文件，只能输出而不能输入；而对于键盘文件，只能输入而不能输出。

输入也被称为读取或读入，输出也称为写出。

### 14.1.3 缓冲

系统在内存中开辟一个专用区域用来临时存放 I/O 数据，这种 I/O 专用的存储区域称为缓冲区(buffer)。输入输出流可以是带缓冲的流，也可以是不带缓冲的流。

- 对于无缓冲的流，一旦将数据送入流中，系统立即进行处理，反应快但处理大批数据的性能比较低。
- 对带缓冲的流，只有当缓冲区已满或执行 `flush` 控制符或函数时，系统才对流中的数据进行处理。磁盘文件一般都是带缓冲的流。带缓冲的流适合处理大批量数据。大多数流都是带缓冲的。

引入缓冲区可大大减少 I/O 实际操作次数，从而提高了系统的整体效率。但如果缓冲区操作不当也会带来错误或遗漏。

## 14.2 基本流类

基本流类由一组类模板形成继承结构，再由一组用 `typedef` 定义的类型别名，再加上 8 个标准输入输出对象，这些组成了基本流类的结构。本节介绍基本流类的结构和分类、预定义标准对象、流的格式控制与错误处理。



当我们要使用某种类型时，应先找最具体类型。当确定具体类型后，其直接间接的基类也就找到了。头文件<iostream>包含了上面大部分类型。

## 14.2.2 预定义标准对象

I/O 流类库中定义了 8 个标准对象，窄字符 cin、cout、cerr 和 clog，宽字符 wcin、wcout、wcerr 和 wclog，提供最基本输入输出，操作键盘和显示器，以字符流实现 IO。包含<iostream>就可使用这些对象，如下表所示。

表 14-1 预定义标准对象

名称	功能	是否可重定向	缓冲	关联对象
cin, wcin	标准输入，缺省为键盘	可以	全缓冲	stdin
cout, wcout	标准输出，缺省为显示器	可以	全缓冲	stdout
cerr, wcerr	标准错误输出，关联显示器	不可	受限缓冲	stderr
clog, wclog	标准日志输出，关联显示器	不可	全缓冲	stdout

流是一个抽象的概念，在进行实际的 I/O 操作前必须将流与一种具体物理设备联系起来。流 cin 和 cout 在缺省情况下分别关联键盘和显示器。流 cerr 和 clog 缺省关联显示器。

标准流通过重载运算符>>和<<来实现 I/O 操作。输入操作是从流中**提取**一个字符序列，为此将运算符>>称为**提取运算符**。输出操作是向流中**插入**一个字符序列，因此将运算符<<称为**插入运算符**。cin 和 wcin 用>>实现数据输入，其余标准流用<<实现数据输出。

这些标准流在进行 I/O 时，自动完成数据类型的转换。对于输入流，要将输入的字符序列根据变量的类型转换为内部格式的数据再赋给变量。对于输出流，将要输出的数据变换成字符序列后再输出。

例 14-1 4 种标准流对象的使用。

```
#include <iostream>
#include <cmath>
using namespace std;
int main(void){
    int x;
    cout << "输入 x = ";
    cin >> x;
    if (x < 0)
        cerr << "输入的是负数，不能开平方"<<endl;
    else
        clog << "x 的平方根为: "<<sqrt(x) << '\n';
    system("pause");
    return 0;
}
```

程序中用户输入的字符串要先转换为一个 int 值，计算平方根之后得到一个 double 值，再转换为一个字符串显示出来。这些转换都隐藏在程序的背后，以简化编程。

14.2.3 流的格式控制

流的格式控制符(manipulator)用来指定输入输出的格式。例如在输出一个整数时，可指定以八进制、十进制数或十六进制输出，也可指定输出数据占用的宽度(字符个数)。流的格式控制适用于文本流。I/O 流库提供了许多格式控制，下面只介绍常用的格式控制。

表 14.2 列出了常用的预定义的格式控制符，用于控制 I/O 数据的格式。

表 14-2 常用预定义格式控制符

控制符名称	功能	适用于
dec	十进制，默认	I/O
hex	十六进制	I/O
oct	八进制	I/O
binary	设置流的模式为二进制(关联 filebuf)	I/O
text	设置流的模式为文本、缺省(关联 filebuf)	I/O
endl	插入一个换行符	O
ends	插入字符串结束符 NULL，仅用于 ostream 对象	O
flush	刷新流，将缓冲区中的数据进行处理并清除	O
ws	跳过前面输入的空格和 Tab	I
以下是带参控制符	要使用下面控制符，须包含<iomanip>	
setioflags(long)	设置指定的标志，参见表 14.3	I/O
setbase(int X)	设置整数的 x 进制	
resetioflags(long)	取消指定的标志，参见表 14.3	I/O
setfill(int)	设置填充字符，缺省为空格，作为分隔符	O
setprecision(int)	设置浮点数的精度，缺省为 6 位。 对于科学表示法和定点表示法，指的是小数点后位数。 对于自动格式，指的是全部位数。	O
setw(int)	设置下一项数据的显示宽度(域宽)，以字符为单位， 仅对下一次插入有效	O
get_money(paras)	从一个流中按指定格式提取一个金额	I
get_time(paras)	从一个流中按指定格式将一个 time 值提取出来	I
put_money(paras)	把一个金额值按指定格式插入到一个流中	O
put_time(paras)	把一个 time 值从一个结构按指定格式写到一个流中	O
quoted(paras)	C14 将一个串的前后多个空格去掉	I/O

表 14.3 列出了控制格式的标记。这些标记定义在 ios\_base::fmtflags 中，也支持 ios 标记，可用 ios::前缀。

表 14-3 流的控制标记

名称	功能
----	----

skipws	跳过输入的空格
boolalpha	对 bool 值按 true, false 输入输出
left	左对齐, 右边加入填充字符
right	右对齐, 左边加入填充字符, 缺省
internal	居中, 左右填充字符
dec	十进制, 缺省, 可用控制符设置
oct	八进制, 可用控制符设置
hex	十六进制, 可用控制符设置
showbase	显示基数, 对十进制没作用。对十六进制用 0x 开头。对八进制, 用 0 开头
showpoint	显示十进制数点, 以及浮点数中的尾零
uppercase	十六进制中显示大写 A-F, 科学表示法中用大写 E
showpos	对正数显示符号+, 缺省不显示
scientific	浮点数的科学表示法
fixed	浮点数的定点表示法
unitbuf	每次插入后都刷新流
stdio	对 stdout 和 stderr 每次插入后都刷新流

例 14-2 格式控制符的例子。

```
#include <iostream>
#include <iomanip>
using namespace std;
int main(void){
    int a = 256, b = 128;
    cout<<"a="<<setiosflags(ios::showbase)<<setw(8)<<a<<endl;//A
    cout<<"b="<<b<<'\n';
    cout<<"a="<<hex<<a<<endl;                                //B
    cout<<"b="<<oct<<b<<'\n';                                //C
    float f = 12345678;
    double d = 87654321, d2 = 3.1415926;
    cout << "f=" << f << endl;
    cout << "d=" << setiosflags(ios::fixed) << d << endl;
    cout << "d2=" << d2 << endl;
    cout << "d2=" << setprecision(3) << d2 << endl;
    system("pause");
    return 0;
}
```

执行程序, 输出如下:

```
a=      256
b=128
a=0x100
b=0200
f=1.23457e+07
d=87654321.000000
d2=3.141593
d2=3.142
```

A 行指定输出 a 值的域宽为 8 个字符, 下面的 b 按缺省的域宽输出。B 行指定 a 的值按

十六进制输出，以“0x”开头。C 行的 b 按八进制输出，以“0”开头。

setw 设置的域宽仅对其后一次插入有效；而 hex、dec、oct 的设置会一直延续到下一次数制设置。

除了格式控制符之外，类 ios\_base 还提供了一组公有成员函数来管理标记和格式控制。

- 函数 flags 用来读取或设置标记字。
- 函数 setf 用来设置标记。
- 函数 unsetf 用来取消标记
- 函数 precision 用来设置浮点数的显示精度。
- 函数 width 用来设置或读取域宽。
- 函数 imbue 改变本地标记，对于 wcin, wcout 有用
- 函数 getloc 返回存储的本地 locale 对象。

以上介绍的内部不仅可用于键盘、显示器的输入输出，也可用于二进制文件的输入输出。在后面章节将详细介绍。

#### 14.2.4 流的错误处理

在 I/O 过程中可能发生各种错误，比如希望输入一个浮点数，但输入的字符串却不能转换为一个浮点数。当发生错误时可利用 ios\_base 类提供的错误检测功能，来检测发生错误的原因和性质。类 ios\_base 中定义的数据成员 iostate 记录 I/O 的状态。

```
goodbit;           //输入/输出操作正常
eofbit;            //已到达流的尾，也就是文件尾
failbit;           //输入/输出操作出错
badbit;            //非法输入/输出操作
```

在 I/O 过程中，状态为 goodbit 时，表示当前流的 I/O 正常。

状态为 eofbit 时，表示从输入流中取数据时，已读到文件尾，就不能再读取。

状态为 failbit 时，表示 I/O 过程中出现了错误，比如希望输入一个整数，但流中的字符串却不能转换为一个整数。如果打开文件失败，也将设置 failbit 位。

状态为 badbit 时，表示非法的 I/O 操作，例如尝试写入到只读文件中，或从只写文件中读取数据。

一个流在每次 I/O 操作之后，都可根据本次操作情况来设置状态位。程序中可以使用 basic\_ios 中提供的公有成员函数来读取状态，并根据不同状态作出相应处理。这些成员函数如下：

```
int rdstate() const    //读取 I/O 状态字
void clear(int _i=0)   //清除流中的状态字
int good() const       //是否正常
int bad() const        //是否是非法 IO 操作
int eof() const        //是否读到流的尾部
int fail() const       //是否非法操作或操作出错
```

通常程序在执行每次 I/O 操作后，就应立即检测是否发生了某种错误。如果发生了 I/O 错误，就要对错误作出处理，最后用函数 clear 来清除流中的错误，以便进行下一个 I/O 操作。

例 14-3 输入错误的处理。下面程序尝试输入一个整数，却可能因输入了字符、标点或其它符号而失败，你可能尝试重新输入，但却导致死循环。

```
#include <iostream>
using namespace std;
int main(void){
    int i;
    cout << "输入一个整数: ";
    cin >> i;                                //A
    while (!cin.good()){
        cin.clear();                          //B
        cout<< "输入错误, 重新输入一个整数: ";
        cin >> i;                             //C
    }
    cout <<"num="<<i<<'\n';
    system("pause");
    return 0;
}
```

执行程序时，如果输入了诸如“b67”之类的字符串，就会导致 while 死循环。原因是 cin 是带缓冲的，当读取缓冲区内容发现错误时，缓冲区中的内容并没有自动清除。B 行仅清除了出错标记，但没有清除输入缓冲区中导致出错的字符串。这样当 C 行重新要求输入时，缓冲区中的内容又被读出尝试转换为整数，再次失败，导致 while 语句死循环。

一种解决办法是在检测到输入错误时，清空缓冲区中的内容，使 cin>>i 等待键盘重新输入。编程如下：

```
#include <iostream>
using namespace std;
int main(void){
    int i;
    char buffer[80];                          //添加一个字符数组
    cout << "输入一个整数: ";
    cin >> i;
    while (!cin.good()){
        cin.clear();
        cin.getline(buffer, 80);              //将输入缓冲区中内容读到 buffer
        cout<< "输入错误, 重新输入一个整数: ";
        cin >> i;
    }
    cout <<"num="<<i<<'\n';
    system("pause");
    return 0;
}
```

当出现 I/O 错误时，调用 getline 函数将输入缓冲区中的内容读入到一个字符数组 buffer 之中，这样缓冲区中内容就被清空，然后就能接收键盘重新输入。函数 getline 是 istream 的一个成员函数。

### 14.3 标准输入/输出

标准 I/O 可以分为三大类：字符类、字符串类和数值（整数、浮点数）类。整数类又可以分为多种类型，如长整型、短整型、有符号型和无符号型等。类 istream 通过多次重载>>运算符实现不同类型数据的输入。类 ostream 通过多次重载<<运算符实现不同类型数据的输出。

### 14.3.1 cin 输入要点

在 C++ 中, 通常利用 `cin` 实现不同类型数据的输入。下面通过示例说明使用 `cin` 时应注意的事项。

例 14-4 `cin` 的示例。

```
#include <iostream>
using namespace std;
int main(void ){
    int i, j;
    cout << "输入一个整数(十进制): ";
    cin >> i;                                //提取十进制数
    cout << "输入一个整数(十六进制): ";
    cin>>hex>>j;                            //提取十六进制数
    cout <<"i="<<hex<<i<<'\n';             //按十六进制输出
    cout <<"j="<<oct<<j<<'\n';             //按八进制输出
    system("pause");
    return 0;
}
```

程序执行时, 需要输入两个整数, 可以一行内输入(用空格或 Tab 分开), 也可以分为两行输入, 程序都能正确执行。例如, 用一行输入:

```
输入一个整数(十进制): 45 ff
输入一个整数(十六进制): i=2d
j=377
```

在要求输入第二个整数时, 直接从缓冲区中读取 `ff`, 而没有等待键盘。如果第一个输入错误, 第二个输入也是错误的。再次执行程序:

```
输入一个整数(十进制): a4 ff
输入一个整数(十六进制): i=cccccccc
j=31463146314
```

输入第一个整数出错, 变量 `i` 被赋予 0, 而当提取第二个字符失败, `j` 未被赋值, 因此 `j` 的值为随机值。

在使用 `cin` 输入数据时, 要注意以下要点。

(1)`cin` 是缓冲流, 键盘输入的数据先送到缓冲区中, 只有当输入换行符 (Enter 键), 系统才开始进行提取数据, 只有把输入缓冲区中的数据提取完后, 才等待键盘输入。

(2)输入多个数据时, 在缺省情况下, 把空格键、Tab 键、Enter 键作为数据之间的分隔符, 在输入两个数据之间可用一个或多个分隔符将数据分开。例如:

```
char c1, c2, str1[80];
cin>>c1>>c2>>str1;
```

输入给变量 `c1` 和 `c2` 的字符不能是分隔符, 输入给 `str1` 的字符串中也不能包含分隔符。

(3)输入的数据类型必须与对应要提取的变量类型相一致, 否则就会出错。这种错误只是在流的状态标志字中置位, 并不终止程序的执行。如果程序中没有及时检测处理, 就会导致数据不正确。尤其是当输入一个整数 `int`、浮点数 `float` 或 `double` 时, 都可能因字符串转换而出错, 对这样的每次操作都应及时进行错误检测和处理。



### 14.3.2 输入操作的成员函数

用提取>>>运算符不能提取分隔符，如空格、Tab 和换行符。如果要提取这些分隔符，就要调用 `basic_istream` 的成员函数来实现输入。

下表列出一组 `get/getline` 函数，提取单个字符或字符串，可输入空格或 Tab 字符，用回车结束输入并开始提取。

表 14-4 `get` 与 `getline` 函数

函数名	说明
<code>int get()</code>	从输入流中提取单个字符并返回
<code>basic_istream&amp; get(char &amp;);</code>	从输入流中提取单个字符并存入形参引用中，返回当前输入流
<code>basic_istream&amp; get(char *a, int n);</code>	从输入流中提取最多 <code>n</code> 个字符，或读到换行符，或者到文件尾 EOF，读入的字符放入 <code>a</code> 数组中，并用 <code>null</code> 结尾。返回当前输入流
<code>basic_istream &amp; get(char*a, int n, char delim = '\n');</code>	从输入流中提取最多 <code>n</code> 个字符，或读到 <code>delim</code> 字符 (缺省为换行符，即 Enter 键)，或者到文件尾 EOF，读入的字符放入 <code>a</code> 数组中，并用 <code>null</code> 结尾。返回当前输入流。
<code>basic_istream &amp; getline(char* a, int n, char delim='\n');</code>	从输入流中提取最多 <code>n</code> 字符，或读到 <code>delim</code> 字符 (缺省为换行符，即 Enter 键，提取出来)，或者到文件尾 EOF，字符放入 <code>a</code> 所指数组中，并用 <code>null</code> 结尾。

EOF 表示 End of File，即文件结尾，在头文件中定义为一个宏，值为 -1。如果要从键盘上输入 EOF，按 Ctrl+Z 键。如果输入流读到文件尾，那么 `ios::eof()` 返回 1，此时从该输入流中就不能再提取字符或数据。所有返回 `basic_istream&` 的 `get` 和 `getline` 函数，也包括提取运算符>>>，如果读到文件尾，就返回 0。

前 3 个 `get` 函数可提取分隔符。第 4 个 `get` 函数不提取指定的 `delim` 字符 (缺省为换行符)，也不保存。而 `getline` 函数将提取输入流中的 `delim` 字符，但不保存该字符。

例 14-5 使用 `get` 和 `eof` 的例子。

```
#include <iostream>
using namespace std;
int main(){
    char c;
    cout<<"before input, cin.eof() is "<<cin.eof()
        <<"\nenter a sentence followed by eof(Ctrl+Z)\n";
    while((c = cin.get()) != EOF)
        cout<<c;
    cout<<"\nEOF in this system is "<<(int)c;
    cout<<"\nafter input, cin.eof() is "<<cin.eof()<<endl;
    system("pause");
    return 0;
}
```

执行程序，前 2 行是输出，第 3 行输出一个串：

```
before input, cin.eof() is 0
enter a sentence followed by eof(Ctrl+Z)
```

```
testing get() and eof
testing get() and eof
^Z

EOF in this system is -1
after input, cin.eof() is 1
```

当执行 `get()` 函数时，你可以输入一个串，以换行结束输入并开始提取。由于 `get` 函数可提取换行符，因此你可输入多行字符串，直到输入了 EOF，即 `Ctrl+Z` 加 `Enter`，停止循环。此时 `cin.eof()` 函数返回 `true` 表示 `cin` 流到达文件尾。

例 14-6 使用 `get` 和 `getline` 的例子。

```
#include <iostream>
using namespace std;
int main(void){
    char c, a1[80];
    cout<<"testing get(char *, int), enter a sentence followed by
enter\n";
    cin.get(a1, 80);
    c = cin.get();           //A 提取换行符
    cout<<"the sentence is ";
    cout<<a1;
    cout<<"\ntesting getline, enter a sentence followed by enter\n";
    cin.getline(a1, 80);
    cout<<"the sentence is ";
    cout<<a1<<endl;
    system("pause");
    return 0;
}
```

执行程序，第 1 行是输出，第 2 行输入了一个串：`testing get`，第 5 行再输入一个串。

```
testing get(char *, int), enter a sentence followed by enter
testing get
the sentence is testing get
testing getline, enter a sentence followed by enter
testing getline
the sentence is testing getline
```

A 行的 `cin.get()` 用来提取前面 `get` 调用所输入的换行符。如果缺少此语句，下面的 `getline` 函数在执行时，因缓冲区已有回车键而读到一个空串，就直接返回了。

使用 `get/getline` 成员函数时要注意以下两点。

(1) 用 `get` 函数提取单个字符或一个串时，就要输入换行符来启动提取，而缓冲区中的换行符就需要单独提取。A 行就是提取换行符。

(2) 用 `getline` 提取字符串时，当实际提取的字符个数小于第 2 个形参指定的字符个数时，则将输入流中的 `delim` 字符也被提取出来，但不保存。

除了 `get` 和 `getline` 之外，还有其它输入函数，读者可参考文档。大部分输入成员函数不仅用于键盘输入，也可用于其它输入，如磁盘文件的读入。

### 14.3.3 cout 输出要点

C++ 的标准输出流对象是 `cout`、`cerr` 和 `clog`，按下面格式输出：

- 输出整数时，缺省设置为：数制为十进制、域宽为 0、数字右对齐、以空格填充。
  - 输出实数时，缺省设置为：精度为六位小数、浮点输出、域宽为 0、数字右对齐、以空格填充。
  - 当输出实数的整数部分超过 7 位或有效数字在小数点右边第 4 位之后，则转换为科学计数法输出。
  - 输出字符或字符串时，缺省设置为：域宽为 0、字符右对齐、以空格填充。
- 域宽为 0 是指按数据的实际占用的字符位数输出，在输出的数据之间没有空格。

在使用标准输出 `cout` 时，应注意以下要点：

- 1、在输出一个 `char` 类型值时，将按 ASCII 字符标准转换为可显示字符。如果要显示其整型值，就需要转换为 `int` 类型再输出。例如，`cout<<(int)c` 将显示整数值。
- 2、在输出 `char*` 类型值时，将其作为字符串来显示。而对其它指针类型，则以十六进制显示指针的值。例如，`cout<<&i`，如果 `i` 是 `int` 类型，将以十六进制显示其地址值。

#### 14.3.4 输出操作的成员函数

除了前面介绍的 `<<` 插入运算符之外，还可用类 `basic_ostream` 中的成员函数。部分成员函数如下表所示：

表 14-5 输出成员函数

函数	说明
<code>basic_ostream&amp; put(char ch);</code>	将单个字符 <code>ch</code> 插入到输出流中。
<code>basic_ostream&amp; write(const char* a, int n);</code>	将数组 <code>a</code> 中的 <code>n</code> 个字符插入到输出流中。 主要用于二进制模式的流的输出
<code>ostream&amp; flush();</code>	刷新流的缓冲区，将缓冲区中内容输出并清除。

例如下面代码：

```
char c1 = 'A';
char pa[10] = "Hello";
cout.put(c1);
cout.put(' ');
cout.write(pa, 5);
cout.flush();
```

执行代码将输出 “A Hello”。

#### 14.3.5 重载提取和插入运算符

重载 `<<` 和 `>>` 运算符可实现对象的输入和输出。在重载这两个运算符时，在自定义的类中，只能用友元函数来实现这两个运算符的重载。

重载 `>>` 运算符的一般格式为：

```
friend istream & operator >>(istream &, ClassName &);
```

函数返回值是类 `istream` 的引用，这是为了在 `cin` 中可连续使用 `>>` 运算符；第一个形参也是类 `istream` 的引用，它是 `>>` 运算符的左操作数；第二个形参为自定义类的引用，它是 `>>` 运算符的右操作数。

重载<<运算符的一般格式为:

```
friend ostream & operator <<(ostream &, ClassName &);
```

函数返回值是类 `ostream` 的引用, 以便可连续使用<<运算符; 第一个形参也是类 `ostream` 的引用, 它是<<运算符的左操作数; 第二个参数为自定义类的引用, 也可以是类的对象, 它是<<运算符的右操作数。

例 14-7 重载提取和插入运算符, 实现对象的输入和输出。

```
#include<iostream>
using namespace std;
class RMB{
    int yuan, jiao, fen;
public:
    RMB(int y=0, int j=0, int f=0){                //缺省构造函数
        float ff = float(y*100.0 + j*10.0 + f)/100;
        yuan = (int)ff;
        jiao = int(ff * 10) % 10;
        fen = int(ff * 100) % 10;
    }
    RMB(float f){                                    //单参构造函数
        yuan = (int)f;
        jiao = int(f * 10) % 10;
        fen = int(f * 100) % 10;
    }
    operator float(){                                //转换函数, 转为 float
        return float(yuan*100.0 + jiao*10.0 + fen)/100;
    }
    friend istream &operator>>(istream & is, RMB &m);
    friend ostream &operator<<(ostream & os, RMB &m);
};
istream &operator>>(istream & is, RMB &m){
    cout<<"输入元 角 分\n";
    is>>m.yuan>>m.jiao>>m.fen;
    return is;
}
ostream &operator<<(ostream & os, RMB &m){
    os<<m.yuan<<"元"<<m.jiao<<"角"<<m.fen<<"分";
    return os;
}
int main(){
    RMB b2;
    cin>>b2;                                         //A
    cout<<b2<<endl;                                 //B
    system("pause");
    return 0;
}
```

执行程序, 第 1 行是输出, 第 2 行是输入, 第 3 行是输出:

```
输入元 角 分
12 3 4
12 元 3 角 4 分
```

上面例子中定义了提取>>和插入<<运算符, 可实现了对象的标准输入和输出。对于 A 行的 `cin >>b2`, 编译器将其变换为对友元函数的调用:

```
operator >>(cin, b2)
```

通过调用运算符重载函数实现对象 `b2` 的各个成员的输入。对于 `B` 行中的 `cout<<b2`，编译器将其变换为对友元函数的调用：

```
operator<<(cout, b2)
```

如果你不重载插入运算符 `<<`，`B` 行代码将自动调用转换函数，先转换为 `float` 再输出。

可以看出，不管如何重载 `<<` 和 `>>` 运算符，要实现数据的输入和输出，在运算符重载函数体内，还是要通过输出流对象(`cout`)和输入流对象(`cin`)来完成数据的输出和输入。

## 14.4 文件流

在 `C++` 中，文件 `file` 可以是指一个具体的外部设备（例如打印机可看作一个文件）。本节所介绍的文件指的是磁盘文件，将讨论磁盘文件的建立、打开、读写和关闭等操作。

### 14.4.1 文件概述

一个文件是一组有序的数据集合。文件通常存放在磁盘上，每一个文件有一个文件名。文件名的命名规则由操作系统规定，在不同的操作系统中文件名的组成规则有所不同。

在 `C++` 语言中，根据文件中存放数据的格式不同，将文件分为二进制文件（二进制数据组成）和文本文件（字符序列组成）。

文件的使用方法基本相同。首先打开一个文件，然后从文件中读取数据或将数据写入到文件中。在读写数据之前，必须先打开它。当不再使用该文件时，应关闭文件，以保存文件并释放系统资源。关闭之后的文件就不能再读写，除非再次打开。

以某一种格式的数据写入到一个文件后；以后从该文件中读取数据时，只有按写入的格式依次读取数据时，读取的数据才是正确的，否则读取的数据不正确。从文件中读取数据的类型是否正确，系统是无法检查判断的。保证从文件中依次取出的数据与原先写入的数据类型相一致是设计者的责任。

图 14.1 包含了文件流的几个关键类：

- 类 `ofstream`，用于把数据写入该文件。
- 类 `ifstream`，用于从该文件中读取数据。
- 类 `fstream`，既可读文件，也可写文件。

从类型体系结构上可以看出，对于磁盘文件的读和写分别对应流的输入和输出。前面介绍的多数输入输出函数都可以直接用来进行文件读写。

磁盘文件作为一种特殊的流，具有自己的特点。文件流有以下特点：

1、在读写之前必须先打开，读写之后应关闭。3 个文件流类都提供了打开 `open` 和关闭 `close` 函数。

2、在读取文件流时要特别关注文件尾 `EOF` 的判断。有多种方式来判断文件尾。

3、随机访问。文件读写依赖读写指针，一般情况下读写指针从头向尾逐个字符或字节移动，但指针可以按顺序移动，也可随机移动，能随机访问文件中任何数据，而限于按顺序访问。

#### 14.4.2 文件处理的一般过程

要使用一个文件，必须先打开文件。打开文件就是将一个文件流类对象(称为控制对象)与某个磁盘文件联系起来；然后调用文件流类的成员函数，将数据写入到文件，或从文件中读取数据。当不再读写该文件时就要关闭文件，以断开磁盘文件与文件流类对象的联系。使用文件的步骤可概括为以下四步。

(1) 创建文件流对象。它只能是类 `ifstream`、`ofstream` 或 `fstream` 的对象。例如：

```
ifstream  infile;
ofstream  outfile;
fstream   iofile;
```

(2) 打开文件。使用文件流对象的成员函数 `open`，或者构造函数打开一个文件，在文件流对象与磁盘文件名之间建立联系。例如：

```
infile.open("myfile1.txt");
outfile.open("myfile2.txt");
```

前两步可合并为一步完成，这要求在第一步创建对象时应调用含文件名的构造函数，而不是缺省构造函数。例如：

```
ifstream  infile(filename1, ios::in );
ofstream  outfile(filename2);
```

(3) 读写。使用提取`>>`运算符、或插入`<<`运算符或成员函数(如 `get/put`、`read/write`)对文件进行读写操作。例如：

```
infile>>ch;
outfile<<ch;
可用 infile.eof() 来判断是否读到文件尾。
```

(4) 关闭文件。读写操作完成之后，调用成员函数 `close` 来关闭文件。例如：

```
infile.close();
outfile.close();
```

下面先讨论文件的打开和关闭。

#### 14.4.3 文件的打开与关闭

输入文件流 `ifstream`、输出文件流 `ofstream` 和 I/O 文件流 `fstream` 分别提供了打开文件函数如下：

```
void  ifstream::open(const char *, int=ios_base::in);
void  ofstream::open(const char *, int=ios_base::out);
void  fstream::open(const char *, int);
```

其中，第 1 个形参是文件名或文件的全路径名。第 2 个形参指定打开文件的模式，输入文件流的缺省值为 `ios_base::in`，表示以输入方式打开文件，只读不写；输出文件流的缺省值为 `ios_base::out`，表示以输出文件方式打开，只写不读；I/O 文件流则没有缺省值，就需要显式指定。

`open` 函数中，第 2 个形参指定了打开文件的操作模式。`ios_base` 定义操作模式如下：

```

in                //按读方式打开文件
out               //按写方式打开文件
ate               //当控制对象首次创建时，指针移到文件尾处
app               //增补方式，每次写入的数据总是增加到尾端
trunc             //当控制对象创建时，将已有文件长度截为 0，清除文件原有内容
binary            //以二进制方式打开文件，主要控制读取方式

```

例如：`file.open( "rm.txt", ios_base::out | ios_base::trunc );`

下面介绍各种具体模式。

- 以 `in` 方式打开，只能从文件中读取数据，读取 `get` 指针放在文件头位置。如果打开的文件不存在，就建立一个空文件。
- 以 `out` 方式打开的文件，只能将数据写入文件中。`out` 经常与 `app`、`ate`、`trunc` 等配合使用。单独用 `out` 方式打开文件时，若文件不存在，则产生一个空文件；若文件存在，则先删除文件已有内容。
- `ate` 方式打开文件，将文件指针移到文件尾，以便于添加数据到文件尾端，首次写字节将添加到尾端，但再次写就写在当前位置，受 `put` 指针限定当前位置。`ate` 方式不能单独使用，要与 `out` 结合使用。
- `app` 方式是指写入数据总是添加到文件尾端，即便是调用 `seekp` 函数改变 `put` 指针。
- `trunc` 方式打开文件写前，删除原有内容。若单独使用，与 `out` 相同。`trunc` 不能与 `ate`、`app`、`in` 结合。
- `binary` 方式打开文件作为二进制流进行操作，若不指明以 `binary` 方式都作为文本文件。这种方式会影响 `eof` 函数的不同作用。如果磁盘文件确实是二进制文件，就应明确以 `binary` 方式来打开，否则就可能导致错误。如果磁盘文件是文本文件，也可用 `binary` 方式打开，只是不如文本流操控方便。

以上三个文件流类中都提供了相应的构造函数来打开文件：

```

ifstream::ifstream(const char *, int=ios::in);
ofstream::ofstream(const char *, int=ios::out);
fstream::fstream(const char *, int);

```

这些构造函数的形参与各自的成员函数 `open` 完全相同。因此在说明这三种文件流类的对象时，可通过这些构造函数直接打开文件。例如：

```

ifstream  f1("file.dat");                                //A
ofstream  f2("file1.txt");
fstream   f3("file2.dat",ios_base::in);

```

不论是用成员函数 `open` 打开文件，还是用构造函数打开，都应立即判断打开是否成功。若成功，则文件流对象的值为非零值；否则其值为 0。实际上，基类 `basic_ios` 定义了一个运算符重载成员函数如下：

```
bool operator!() const;
```

如果当前流的错误状态的 `failbit` 或 `badbit` 被置位，就返回一个非 0 值。也就是说，如果

返回了非零值，说明流的操作失败或非法操作，相当于调用成员函数 `fail()`。

打开文件的一般形式如下：

```
ifstream f1("file.dat");  
  
if (!f1){                                     //判断打开文件是否成功  
    cout <<"不能打开输入文件: "<<"file.dat"<<"\n";  
    return;  
}
```

也可先输入要打开的文件名，然后再打开文件：

```
char filename[256];  
cout <<"输入要打开的文件名: ";  
cin >> filename;  
ifstream f2(filename, ios_base::in);  
  
if (!f2){                                     //判断打开文件是否成功  
    cout <<"不能打开输入文件: "<<filename<<"\n";  
    return;  
}
```

如果输入文件名出错，打开文件就会失败，此时你可能希望重新输入文件名，再次尝试打开文件。注意，因前一次打开失败，该输入流对象的状态标记 `failbit` 被置位，如果不清除出错标记，再一次尝试打开也会失败。因此再次打开之前应先执行 `clear` 函数。例如：

```
char filename[256];  
cout <<"输入要打开的文件名: ";  
cin >> filename;  
ifstream f2(filename);  
  
while (!f2){                                  //判断打开文件是否成功  
    cout <<"不能打开输入文件: "<<filename<<"\n";  
    cout<<"再次输入文件名: ";  
    cin>>filename;  
    f2.clear();                               //先清除出错标记，再尝试打开  
    f2.open(filename);  
}
```

打开文件成功后，才能对文件进行读写操作。读写完成后，应关闭文件。尽管在程序执行结束或在撤消文件流对象时，系统会自动关闭已打开文件，但仍应显式关闭文件。这是因为打开一个文件时，系统要为打开的文件分配一定的资源，如缓冲区等，在关闭文件后，系统就收回了该文件所占资源。这样及时关闭文件可提高资源利用率。另一个原因是，操作系统可能限制一个程序同时打开的文件数。

三个文件流类各自提供一个关闭文件的成员函数如下：

```
void ifstream::close();  
void ofstream::close();  
void fstream::close();
```



这三个成员函数的用法完全相同。例如：

```
ifstream infile("f1.dat");
...
infile.close(); //关闭文件 f1.dat
```

关闭文件时，系统断开磁盘文件与控制对象之间的联系。关闭文件后，就不能再对该文件读写。如果要再次使用该文件，必须重新打开该文件。

#### 14.4.4 文本文件的使用

文本文件是按一定字符编码标准来编写字符的。例如英文字符通常采用 ASCII 标准，中文字符通常采用 GB2312、GBK、GB18030 等。一个文本文件通常由多行组成，就像我们编写的 `cpp` 后缀的源文件。一个文本文件是行的一个序列，而每一行是一个字符序列，并以换行符结尾。读写文本文件既可按字符读写，也可以按行读写。对于文本文件，如果读到 0 就到达文件尾。

在 Windows 系统中，可用记事本 Notepad 打开正常观看的文件都是文本文件。

类 `ifstream`、`ofstream` 和 `fstream` 并没有直接定义文件读写操作的成员函数。对文件的操作由基类 `ios`、`istream`、`ostream` 中定义的成员函数来实现。对于文本文件，文件的读写操作与标准 I/O 流相同，也是通过提取运算符 `>>` 和插入运算符 `<<`、`get` 和 `put` 函数来读写。

例 14-8 复制文本文件。使用构造函数打开文件，并把源程序文件拷贝到目的文件中。

分析：先打开源文件和目的文件，依次从源文件中读一个字节，并把所读的字节写入目的文件中，直到把源文件中的所有字节读写完为止。程序如下：

```
#include<iostream>
#include <fstream>
using namespace std;
int main(void) {
    char filename1[256], filename2[256];
    cout << "输入源文件名: ";
    cin >> filename1;
    cout << "输入目的文件名: ";
    cin >> filename2;
    ifstream infile(filename1, ios::in );
    ofstream outfile(filename2);
    if (!infile) {
        cout << "不能打开输入文件: " << filename1 << '\n';
        system("pause");
        return 0;
    }
    if (!outfile) {
        cout << "不能打开目的文件: " << filename2 << '\n';
        system("pause");
        return 0;
    }

    infile.unsetf(ios::skipws); //A
    char ch;
    while (infile >> ch)
        outfile << ch; //B

    infile.close();
```

```

    outfile.close();

    system("pause");
    return 0;
}

```

A 行设置为不跳过文件中的分隔符。在缺省情况下，提取运算符要跳过分隔字符，而文件的拷贝必须连同分隔字符一起拷贝。B 行依次从源文件中取一个字符，并将该字符写到目的文件中。当到达源文件的结束位置时（无数据可取），表达式 `infile >> ch` 的返回值为 0，结束循环；否则其返回值不为 0，继续循环。该循环语句完成文件的拷贝。注意，文本文件中只允许最后一个字节为 0 值。

上面例子中真正的拷贝操作就是从 A 语句到 B 语句，还有多种实现方式。下面是采用 `get` 和 `put` 函数来实现的。

```

char ch;
while (infile.get(ch))
    outfile.put(ch);           //C

```

用成员函数 `infile.get` 逐个读取字符，包括所有的分隔符。C 行从源文件中取出一个字符，并将取出的字符写到目的文件中。当没有到达源文件结束位置时，`infile.get` 的返回值不为 0，继续循环。当到达源文件结束位置时，该函数的返回值为 0，结束拷贝。

如果我们假定所拷贝的文本文件中每行最多字符为 300，就可以按行来读取，而无需按字符来读，这样复制大文件时执行效率更高。下面就是采用 `getline` 函数按行读取的。

```

char buff[300];
while (infile.getline(buff,300))    //D
    outfile<<buff<<'\n';           //F

```

D 行中的 `infile.getline(buff,300)` 从源文件中读取一行字符，F 行将读取的一行字符写到目的文件中。到达源文件尾时，`infile.getline` 函数的返回值为 0，读取结束；否则返回值不为 0，表示要继续拷贝。F 行中插入换行符 `'\n'` 是必要的，这是因为 `getline` 读取一行时，换行符取出来后并不放入 `buff` 中，所以写入目的文件中时，要加入一个换行符。

这个程序稍加改动就能统计一个文本文件的行数，进一步可统计空行数和非空行数，也能搜索指定字符串在文本文件中出现的行号和行中的位置(第几个字符)。

注意，在 VS2015 环境中执行该程序，需要(1)从可执行程序所在目录用命令行启动程序；(2)被读写的文件也位于该目录中。

注意，该程序只能实现文本文件的拷贝，不能实现二进制文件的拷贝。

例 14-9 设文本文件 `data.txt` 中有若干个实数，各个实数之间用分符隔分开。例如：

```

24  56.9  33.7  45.6
88  99.8  20    50

```

求出文件中的这些实数的个数和平均值。

分析：设一个计数器和一个累加器，初值均为 0。从文件中每读取一个实数时，计数器加 1，并把该数加到累加器中，直到把文件中的数据读完为止。把累加器的值除以计数器的值得到平均值。编程如下：

```

#include <iostream>
#include <fstream>
using namespace std;
int main(void){
    ifstream infile("data.txt",ios::in);
    if (!infile ) {

```

```
        cout << "不能打开输入文件:\n";
        system("pause");
        return 0;
    }
    float sum = 0, temp;
    int count = 0;
    while (infile >> temp){                //依次读一个实数
        sum += temp;                        //累加
        count++;
    }
    cout<<"个数="<<count<<";平均值="<<sum/count<<endl;
    infile.close();
    system("pause");
    return 0;
}
```

执行程序，输出如下：

```
个数=8;平均值=52.25
```

上面程序所打开的文件中的所有实数都是正确格式，但如果其中存在错误格式，例如，文件中的 88 改为 8b，然后执行程序，输出如下：

```
个数=5;平均值=33.64
```

程序读取前 5 个实数，其中最后一个为 8，后面 3 个实数就不再读取了。

### 14.4.5 二进制文件的使用

二进制文件不同于文本文件，例如，可执行程序.exe 文件是二进制文件，目标文件.obj、静态库.lib、动态链接库.dll，所有的图片、视频、音频文件、office 文件等，都是二进制文件。二进制文件的内容看做是字节的序列，其中每个字节可以是任何值。

二进制文件内容是内存的等值映射。例如，写入一个 char 值就是写入 1 个字节，写入一个数组 char a[40]就是 40 字节，写入一个数组 int b[20]就是写入 20\*4 字节。

二进制文件内容与内存中内容相同，无需数据类型转换。但应注意，二进制文件与文本文件的本质区别。比如一个 int 值，作为二进制存储占 4 字节，而作为文本存储就不能确定大小。如 3 占 1 字节，而-2147483648 要占 11 字节。文本存储需要分隔符，而二进制存储则不需要分隔符，但要计算相对位置，并进行随机访问。

如果要打开一个二进制文件进行观察或修改，需要支持十六进制编辑工具，例如 UltraEdit、WinHex 等工具。

打开二进制文件，应指明以二进制 ios::binary 方式打开。对二进制文件的读或写应通过文件流对象的成员函数来实现，而不能使用提取运算符>>和插入运算符<<来读写文件。

对二进制文件的常用操作如下表所示。

表 14-6 二进制文件操作常用函数

函数	说明
----	----

<code>basic_istream&amp; read(char* pch, int nCount);</code>	从二进制输入流中提取 nCount 个字节，或者到文件尾，提取的字节放入到 pch 数组中，并返回当前流对象。
<code>basic_ostream&amp; write(const char * pch, int nCount);</code>	将数组 pch 中的 nCount 个字节插入到二进制输出流中，并返回当前流对象。
<code>bool basic_ios::eof();</code>	当读到文件尾时，函数返回 true；否则返回 false。文本文件和二进制文件都适用。
<code>int basic_istream::gcount() const;</code>	返回最后一次读取到的字符个数。常与 read 函数配合。

例 14-10 生成一个二进制数据文件 data.dat，将 100 之内的所有素数写入文件。编程如下：

```

#include <iostream>
#include <fstream>
using namespace std;
bool isPrime(int n){
    if (n < 2)
        return false;
    if (n == 2 || n == 3 || n == 5 || n == 7)    //10 以内的素数
        return true;
    for (int i = 2; i * i <= n; i++)
        if (n % i == 0)
            return false;
    return true;
}

int main(void){
    ofstream outfile("data.dat", ios::out | ios::binary);    //A
    if (!outfile) {
        cout << "不能打开目的文件 data.dat\n";
        system("pause");
        return 0;
    }
    int i = 2, count = 1;
    outfile.write((char*)&i, sizeof(int));    //B
    for(i = 3; i < 100; i += 2)
        if (isPrime(i)){
            outfile.write((char*)&i, sizeof(int));    //C
            count++;
        }
    cout<<"write "<<count<<" integers\n";
    outfile.close();
    system("pause");
    return 0;
}

```

执行程序，输出如下：

```

write 25 integers

```

A 行以二进制方式打开输出文件 data.dat。在 B 行和 C 行中将 i 的地址强制转换成字符指针，这是因为函数 write 的第一个形参为 char 指针。一般情况下，如果你要将 T 类型的变量 v 写入二进制文件，T 类型可以是基本类型、也可以是自定义类型，既可以是单个变量，

也可以是一个数组，一般的写入形式如下：`write((char*)&v, sizeof(T))`。

上面程序执行写入了 25 个整数，那么该文件大小应该是  $25 \times 4 = 100$  字节。如果用十六进制方式观察该文件，可以看到前 4 个素数如下：

02 00 00 00 03 00 00 00 05 00 00 00 07 00 00 00

最后一个整数为 61 00 00 00。

例 14-11 从上例中产生的数据文件 `data.dat` 中读取素数，并在显示器上按十进制显示，每行 5 个数。编程如下：

```
#include<iostream>
#include <fstream>
using namespace std;
int main(void){
    ifstream infile("data.dat",ios::in | ios::binary);    //A
    if (!infile) {
        cout << "不能打开目的文件 data.dat\n";
        system("pause");
        return 0;
    }
    int i, a[25];
    infile.read((char *)a, sizeof(int)*25);                //B
    for(i=0; i < 25; i++){
        cout<<a[i]<<'\t';
        if((i+1) % 5 == 0 )
            cout<<'\n';
    }
    cout<<'\n';
    infile.close();
    system("pause");
    return 0;
}
```

执行程序，输出如下：

2	3	5	7	11
13	17	19	23	29
31	37	41	43	47
53	59	61	67	71
73	79	83	89	97

A 行以输入文件方式打开已有的二进制文件 `data.dat`。如果编程时知道要读取的整数的个数，而且内存可容纳，就可把文件中的所有数据一次性全部读入，如 B 行中一次从文件 `data.dat` 中读取 25 个整数。如果事先不知道整数的个数，或者内存不够大，就需要分批读入处理，此时就要调用 `eof()` 函数来判断文件尾。

例 14-12 复制文件。使用成员函数 `read` 和 `write` 来实现文件的拷贝。

```
#include<iostream>
#include <fstream>
using namespace std;
int main(void) {
    char filename1[256], filename2[256];
    cout << "输入源文件名:";
    cin >> filename1;
    cout << "输入目的文件名:";
    cin >> filename2;
```

```
fstream infile, outfile;
infile.open(filename1, ios::in | ios::binary );
outfile.open(filename2, ios::out | ios::binary);
if (!infile) {
    cout << "不能打开输入文件:" << filename1 << '\n';
    system("pause");
    return 0;
}
if (!outfile) {
    cout << "不能打开目的文件:" << filename2 << '\n';
    system("pause");
    return 0;
}
char buff[4096];
int n;
while (!infile.eof()) {
    infile.read(buff, 4096);
    n = infile.gcount();
    outfile.write(buff, n);
}
infile.close(); outfile.close();
system("pause");
return 0;
}
```

该程序不仅可复制二进制文件，也能复制文本文件。因为文本文件也可按二进制方式打开，将字符序列作为字节序列一样处理。在 `while` 循环中，使用函数 `eof` 来判断是否已到达文件的结尾。由于从源文件中最后一次读取的数据可能不到 4096 个字节，所以使用函数 `gcount` 来获得实际读入的字节数，并按实际读的字节数写到目的文件中。

#### 14.4.6 文件的随机访问

前面介绍的文件读写操作，都是按文件中数据的先后顺序依次进行读写的。实际上，在文件读写操作中有一个文件指针的概念。在打开一个输入文件时，系统为此文件建立一个长整数 `long` 变量（设变量名为 `point`），它的初值为 0，指向文件开头准备读取。文件中的内容可以看成是由若干个有序字节所组成，依次给每一个字节从 0 开始顺序编号，就像一个数组一样。如果文件的当前字节长度为 `S`，那么指针的有效范围是  $[0, S]$ 。

每个输入文件流(`ifstream` 对象)有一个 `get` 指针，指向读取位置。当读取第 `n` 个字节时，系统修改指针的值为 `point += n`。每次读取数据时，均从指针所指位置开始读取，读完后再增加 `point` 值。这样指针总是指向下一次读取数据的开始位置。当指针等于 `S`，就到达文件尾 EOF，就不能再读。

每个输出文件流(`ofstream` 对象)有一个 `put` 指针，指向写入位置。该指针总是指向下一次写入的位置。每次写入之后，都要增加 `point` 的值，使它指向下一个写入位置。如果该指针指向文件尾，就会添加数据。如果该指针定位不是文件尾，那么写入的数据就覆盖了原有的数据。注意，不能在中间插入数据，也不能删除已写数据。

一个输入输出文件流(`fstream` 对象)既有一个 `get` 指针，也有一个 `put` 指针。这两个指针可以独立移动。

文件指针可以自由移动，就可以随机读写文件。当文件指针值从小向大方向移动，称为

后移,反之称为前移。文件指针既可按绝对地址移动(以文件开头位置作为参照点),也可按相对地址移动(以当前位置或者文件尾作为参照点,再加上一个位移量)。

C++允许从文件中的任何位置开始进行读或写数据,这种读写就被称为文件的随机访问。在文件流类的基类中提供了文件指针操作函数,如下表所示。

表 14-7 文件指针的操作函数

函数	说明
<code>basic_istream&amp; seekg(streampos pos);</code>	将输入流的 get 指针定位到 pos 绝对位置
<code>basic_istream&amp; seekg(streamoff off, ios_base::seek_dir dir);</code>	将输入流的 get 指针定位到 off 相对位置,相对于 dir
<code>streampos tellg();</code>	返回输入流的当前定位
<code>basic_ostream&amp; seekp(streampos pos);</code>	将输出流的 put 指针定位到 pos 绝对位置
<code>basic_ostream&amp; seekp(streamoff off, ios_base::seek_dir dir);</code>	将输出流的 put 指针定位到 off 相对位置,相对于 dir
<code>streampos tellp();</code>	返回输出流的当前定位

上表中类型 `streampos` 和 `streamoff` 都是 `long` 同义词,分别表示绝对位置和相对位移量。

相对定位需要指定参考点。`seek_dir` 是类 `ios_base`(`ios` 可看做 `ios_base` 派生类,因此多用 `ios`)中定义的一个公有枚举类型,表示了相对位移的参照点如下:

```
enum seek_dir{
    beg = 0,    //文件开始处作为参照点
    cur = 1,    //文件当前位置作为参照点
    end = 2     //文件结束处作为参照点
};
```

在相对位移时,如果参照点为 `ios::beg`,则将第一个形参值作为文件指针的值。若为 `ios::cur`,则将文件指针当前值加上第一个形参值的和作为文件指针的值。若为 `ios::end`,则将文件尾的字节编号值加上第一个形参值的和作为文件指针的值。假设按输入方式打开了一个文件流对象 `f`,移动文件指针如下:

```
f.seekg(50, ios::cur);           //当前文件指针值后移 50 个字节
f.seekg(-40, ios::cur);          //当前文件指针值前移 40 个字节
f.seekg(-50, ios::end);          //设文件尾的编号为 5000,则指针移到 4950 处
f.seekg(0, ios::end);            //当前文件指针移到文件尾
```

在随机访问时,要注意以下要点:

(1)在移动文件指针时,必须保证指针值大于等于 0 且小于等于文件尾字节编号,否则将导致读写数据不正确。程序中可调用函数 `tellg` 和 `tellp` 来观察当前文件指针的值。

(2)使用 `eof()`函数判断读到文件尾,如果还要移动指针再次读写,就要先清除 `eof` 标记,调用 `clear()`函数,然后再移动指针再读写。

例 14.13 前面例 14-10 产生了一个文件 `data.dat`，保存了前 25 个素数，并按从小到大次序排列。实现一个函数，用于读取第 `index` 个素数。`index` 的合理范围是[0, 24]，如果越界则返回-1。但因 `data.dat` 文件随时可能添加新的素数，因此需要根据当前素数个数动态计算。编程如下：

```
#include <iostream>
#include <fstream>
using namespace std;
//读取第 index 个素数，如果 index 越界，返回-1
int getPrime(ifstream & ifs, int index){
    ifs.seekg(0, ios::end);
    int size = ifs.tellg() / sizeof(int);
    if (index < 0 || index >= size)
        return -1;
    ifs.seekg(index*sizeof(int));
    int i = 0;
    ifs.read((char *)&i, sizeof(int));
    return i;
}
int main(void){
    ifstream file("data.dat", ios::in|ios::binary);    //C
    if (!file) {
        cout << "不能打开目的文件 data.dat\n";
        system("pause");
        return 0;
    }
    for(int i = 15; i <= 25; i++)
        cout<<getPrime(file, i)<<" ";
    cout<<endl;
    file.close();
    system("pause");
    return 0;
}
```

执行程序，输出如下：

```
53 59 61 67 71 73 79 83 89 97 -1
```

函数 `getPrime` 演示了 `seekg` 函数的相对定位和绝对定位，以及如何使用 `tellg` 来计算当前素数的个数。这个函数适用于从很大文件中读取指定素数，当需要多次求很大序号的素数时，该函数就具有实用价值。比如求第 50001 个素数，如果不采用文件存储和随机访问，就要先计算前 50000 个素数，这是很大的计算负担。

随机访问通常作用于二进制文件，也可以作用于文本文件，前提是文本文件中的记录和字段能确定大小或者容易区分，这样才能计算定位。

## 14.5 小 结

- 输入/输出是内存中的程序与外部设备之间的信息交换。输出就是程序中的对象或值写入到外部设备，输入就是程序从外部设备读取对象或值。IO 被称为流是因为以字符或字节的序列从源到目的之间的流动。



- 流可分为文本流和二进制流。文本流是按特定规范编码的字符流，而二进制流是字节流。
- 在流的概念中，文件是对外部设备的抽象。磁盘文件仅仅是一种具体形式。
- 大多数流都是带缓冲的，以提高处理大批量数据的性能。
- C++提供了一个流类体系，由 ios 基类和一组派生类组成。
- I/O 流类库中定义了四个标准流对象：cin、cout、cerr 和 clog，提供最基本的输入和输出功能，操作键盘和显示器设备，以字符流实现 IO。
- ios 类提供了一组的格式控制符、控制标记以及相关函数。这些对于文本流的格式控制具有重要作用。
- 为了检测流中的错误，ios 类提供了一组函数来检测错误和清除错误状态。但在文本流输入操作出错时，要重新输入就要注意清除键盘缓冲区，然后再重新输入。
- 在用 cin 输入数据时，应注意换行提取、分隔符、及时检测错误。
- 应注意提取运算符>>、get 函数、getline 函数之间的区别。这些操作也可作用于文本文件流。
- 用 cout 输出的主要方式一般是插入运算符<<和 put 函数。
- 对于一个类可重载定义提取运算符>>和插入运算符<<，以方便对象值的输入和输出。
- 文件流是从基本 IO 体系中派生而来的，用于磁盘文件的操作，因此对于一般流的性质和操作，也同样适用于文件流。
- 文件流在读写之前必须先打开，读写之后应该关闭。在读取文件流时要特别关注文件尾 EOF 的判断。对文件流可以进行随机访问。
- 磁盘文件可分为文本文件和二进制文件两大类。当打开文件时应选择正确的打开模式。对于二进制文件，一定要用 ios::binary 方式打开，否则就可能导致文件尾的判断错误，从而导致处理出错。运算符>>/<<、get/getline/put 适合文本文件读写，而 read/write 适合二进制文件的读写。
- 文件指针可以自由移动，可随机读写文件。当文件指针值从小向大方向移动，称为后移，反之称为前移。文件指针既可按绝对地址移动(以文件开头位置作为参照点)，也能按相对地址移动(以当前位置或者文件尾作为参照点，再加上一个位移量)。

## 14.6 练 习 题

1. 编写一个函数 `int getInt(const char * msg)`，先输出形参字符串要求输入一个整数，然后从键盘读取一个整数并返回。如果发生错误就给出错误提示，并再次输入，直到得到一个整数为止。
2. 下面哪一个输入操作不能读取空格符？  
A >>运算符    B `get()` 函数    C `get(char)` 函数    D `getline` 函数
3. 下面哪一个输入操作能读取换行符？  
A >>运算符    B `get()` 函数    C `get(char*, int)` 函数    D `getline` 函数
4. 编写一个函数 `int getLine(const char * file)`，读取文本文件 `file` 的总行数。

再编写一个函数 `int getLineNoEmpty(const char * file)` 读取文本文件 `file` 的总行数, 不包括空行, 即仅由分隔符组成的行。

分别测试这些函数。

5. 编写一个程序, 输入一个文件名, 然后将键盘输入的内容写入到该文件中, 要求保持所有的分隔符, 并以 `Ctrl+Z` 结束输入并关闭文件。

6. 有一类特殊的文本文件, “\*.ini” 后缀文件, 作为配置文件或初始化文件, 在 Windows 系统中可以发现很多这样的文件, 其中某个文件如下:

```
[connect default]
;If we want to disable unknown connect values, we set Access to NoAccess
Access=NoAccess
[connect CustomerDatabase]
Access=ReadWrite
Connect="DSN=AdvWorks"
[sql CustomerById]
Sql="SELECT * FROM Customers WHERE CustomerID = ?"
[connect AuthorDatabase]
Access=ReadOnly
Connect="DSN=MyLibraryInfo;UID=MyUserID;PWD=MyPassword"
```

此类文件中有 3 种行格式:

- 用分号;开头的是注释行。
- 用[字符串]说明的是一个类别名称。
- 用“属性名=值”的格式说明一个属性的名字及对应的值。值可以用双引号括起来的一个串, 也可以不带双引号。如果值中有等号, 就要用双引号。

在一个类别下面可说明多个属性, 这些属性不重名。在不同的类别中, 属性可以重名。

- a) 编写一个函数 `bool getValue(const char * kind, const char * attr, char * value)`, 读取 `kind` 类别中的 `attr` 属性的值, 将值写入到形参 `value` 中。如未找到类别或属性, 就返回 `false`, 如找到, 就返回 `true`。
- b) 编写一个函数 `bool setAttr (const char * kind, const char * attr, const char * value)`, 设置或更改 `kind` 类别中的 `attr` 属性的值为 `value`。如果作为新属性, 就返回 `true`。如果更改属性, 就返回 `false`。

7. 编写一个通用的文件拷贝程序 `mycopy`。要求(1)从命令行读取源文件名和目标文件名, (2)能拷贝任何文件,(3)可处理文件的相对路径名或全路径名。例如: `mycopy mycopy.exe temp\yourcopy.exe`。

注: 当输入的一个文件名为 “test\abc.exe”, 要将这文件名转换为 “test\\abc.exe” 或 “test/abc.exe” (因为 C++ 把字符 “\” 作为一个转义字符, 而操作系统将它作为分隔符)。

8. 编写一个函数将一个 `row` 行 `col` 列的 `float` 型矩阵写入到一个文本文件 `matrix.txt` 中, 该文件中头两个数据是整数, 保存 `row` 行数和 `col` 列数, 其后为 `row*col` 个浮点数。

再编写一个函数将该文件中的数组读到一个矩阵中。此时需要使用上一章介绍的 `TMatrix<T>`, 任意类型的、任意大小的矩阵。就是用这样一个文本文件来构造一个 `TMatrix<float>` 对象。

9. 分析 14.5 节中介绍的例子, 学生分数管理, 尝试添加如下功能:

- a) 计算平均分, 计算不及格人数所占百分比。
- b) 对文件中的记录按学号升序排序、按成绩降序排序。只需要显示, 不需要更新文件。
- c) 对文件中的记录按某些指定条件加载, 例如, 大于 90 分的记录, 不及格的记录, 等等。

## 第15章 异常

程序中经常要检查处理各种错误情形，如果用传统的流程控制语句来处理，很容易使程序逻辑混乱。异常(exception)就是一种专门用于检测错误并处理的一种机制，使程序保持逻辑清晰，并改进程序的可靠性。C++语言提供了基本的异常处理机制。本章主要介绍异常的概念、语句、异常类型架构及应用。

可靠的编程应尽可能地、及时地检测到各种异常情形，并尽可能在本地处理。尽管有时本地不能处理，也应向调用方提供详细的出错信息，使调用方能得到充分信息，从而采取合适方式来处理异常。

### 15.1 异常的概念

异常是什么概念？异常 exception 就是在程序运行中发生的难以预料的、不正常的事件而导致偏离正常流程的现象。例如：

- 访问数组元素的下标越界，在越界时又写入了数据；
- 用 new 动态申请内存而返回空指针(可能是因内存不足)；
- 算术运算溢出；
- 整数除法中除数为 0；
- 调用函数时提供了无效实参，如指针实参为空指针(如用空指针来调用 strlen 函数)；
- 通过挂空指针或挂空引用来访问对象；
- 输入整数或浮点数失败；
- I/O 错误，等等。

上面列出的情形之一如果发生，就可能导致运行错误而终止程序。

发生异常将导致正常流程不能进行，就需要对异常进行处理。那么异常处理是什么概念？异常处理(exception handling)就是在运行时刻对异常进行检测、捕获、提示、传递等过程。如果采用传统的 if-else 语句来检测处理所有可能发生的异常，很容易导致程序流程混乱，分不清正常流程与异常流程，而且在处理一个异常时往往又引入了新的异常。

假设要设计一个函数，从一个文本文件中读取数据得到一个 float 矩阵。该文件应存放一个  $m \times n$  的 float 矩阵，头两个整数说明其行数  $m$  和列数  $n$ 。你要把它读入并创建一个矩阵对象，以备下一步计算。如果你认为文本文件不会有错，完全按正常编程，不超过 10 条语句就能完成。如果这个文本文件是别人提供的，而且你的函数将提供给其它人使用，那么你在每一步都要考虑可能出现的错误，此时就可能需要 30 条语句来处理。例如，可能的出错情形如下：

- 打开文件出错，文件名可能有误；
- 读取行数 *m* 或者列数 *n* 可能出错；
- 读取每个元素时都可能出错；
- 矩阵数据可能不完整，也会出错。

如果你用传统方式来判断处理以上这些问题，就会发现正常流程被淹没在多种异常判断处理之中。此时就需要有一种统一的机制能将正常流程与异常处理分开描述，而保持程序逻辑清晰可读，同时各种异常情形能被集中处理。

C++提供了引发异常语句 `throw` 和捕获处理异常语句 `try-catch`。它们构成了一种特殊的流程控制。

用 `throw` 引发的每个异常都可以描述为一个对象或一个值。在程序中，每一种异常都可以描述为一种类型，可能是自定义的类，也可能是简单的整数或字符串。在比较完善的编程中，经常用不同的类来描述不同的异常，建立一个异常类型的继承结构，以方便对异常类型的管理和重用。

一个函数中当检测到某种异常发生，但自己往往不知道应该如何处理，此时就应该通知调用方知道发生了什么异常。处理异常的一般方式是：在一个函数中发现一个错误但不能处理，就用 `throw` 语句引发一个异常，希望它的（直接或间接）调用方能够捕获并处理这个异常。函数的调用方如果能解决该异常，就可用 `try-catch` 语句来捕获并处理这种异常。如果调用方不能捕获处理该异常，异常就被传递到它自己的调用方，最后到达 `main` 函数。

异常的发生、传递与处理的过程与函数调用堆栈相关。如图 15.1 所示。`main` 函数中调用 `f` 函数，`f` 函数再调用 `g` 函数。如果 `g` 函数执行 `return` 就正常返回到 `f`。如果 `f` 执行到 `return` 就正常返回到 `main`。这是正常流程。

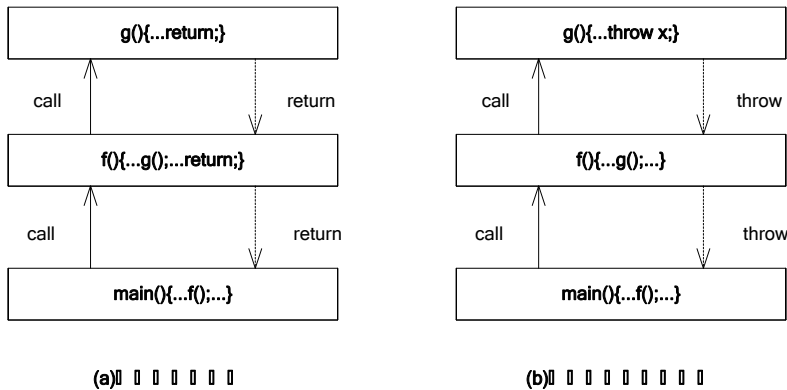


图 15-1 函数调用堆栈与异常传递

如果 `g` 函数在运行时因检测到某种错误而用 `throw` 语句引发一个异常，而自己也没有捕获处理，此时该异常就被传递到 `f` 的调用方 `g` 函数，而且 `g` 函数执行终止(注意，不是返回)。对于 `f` 来说就是 `g` 函数调用发生异常。此时如果 `f` 函数没有捕获该异常，那么异常又被传递到它的调用方 `main` 函数，此时 `f` 函数执行终止。同理，此时如果 `main` 也没有捕获该异常，那么程序就必须终止。此时系统可能会跳出一个对话框告知你发生了运行错误。

在发生异常、传递异常的过程中，如果有一个函数用 `try-catch` 捕获了该异常，就不会导致程序终止。在运行时刻，一个异常只能被捕获一次。假设 `f` 函数捕获了这个异常，那么对于它的调用方 `main` 函数来说，就等于没有发生异常。

异常编程的目的是改善程序的可靠性。在大型复杂程序中，完全不发生异常几乎不可能，用传统的 `if-else` 语句来检查所有可能的异常情形，也有很大困难。编程正确性总是依赖某些假设成立为前提，异常编程就是要分析识别这些假设不成立的情形，采用面向对象编程技术，建立各种异常类型并形成继承性架构，以处理程序中可能发生的各类异常。

## 15.2 异常类型的架构

C++的异常类型可以是任何类型，既可以是基本类型，如 `int` 整数、`char*` 字符串，也可以是自定义类型。在比较规范的编程中，往往不能将基本类型作为异常类型。这是因为基本类型所能表示的异常种类有限。例如在一个程序中 `int` 类型只能表示一种异常情形，如果在不同函数中多处引发不同语义的 `int` 异常，就很难区别不同 `int` 值的含义。可能表示访问数组的下标越界，也可能表示打开文件不成功。

在比较规范的编程中往往根据各种错误情形，利用类的继承性建立一个异常类型的架构，作用如下：

- 对所处理的各种错误情形进行准确描述、抽象和归类。
- 方便扩展新的异常类型。
- 在编程中方便选取引发正确的异常类型，也方便按类型来捕获处理异常。

图 15.2 是定义在 `<exception>` 和 `<stdexcept>` 中的一个异常类型架构。在头文件 `<exception>` 中定义了基类 `exception` 和一组函数，在 `<stdexcept>` 中定义了一组派生类，表示各类具体的异常。标准模板库 STL 中的部分函数就利用了这个架构。下面简单介绍各种异常类型。

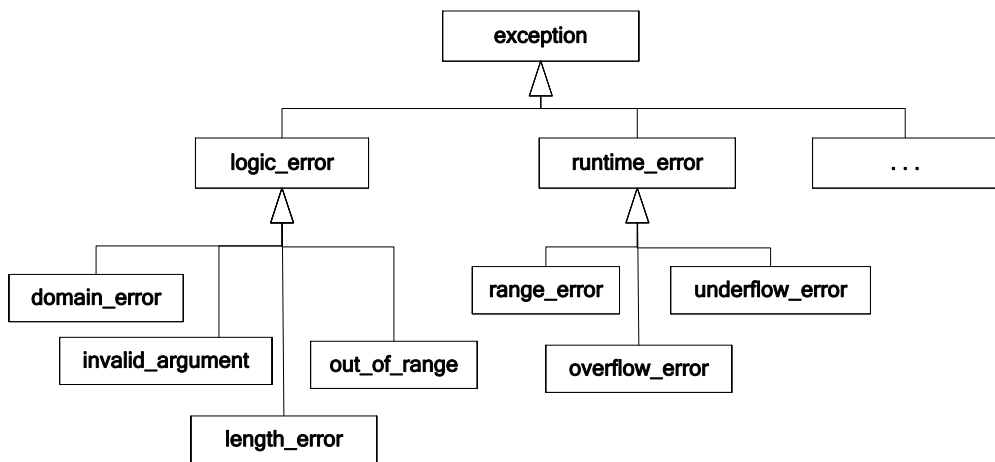


图 15-2 异常类型架构

类 `exception` 是所有异常类的基类，其公共成员如下：

```
class exception{
    ...
```

```

public:
    exception() throw(); // 缺省构造函数
    exception(const exception& rhs) throw(); // 拷贝构造函数
    exception& operator=(const exception& rhs) throw(); // 赋值操作函数
    virtual ~exception() throw(); // 虚析构函数
    virtual const char *what() const throw(); // 虚函数
};

```

注意到每个函数原型末尾都有“throw()”，称为函数的异常规范(exception-specification)，括号中为空说明该函数中不会引发任何异常出来。如果一个函数在执行时可能引发某种异常类型，就应该在“throw(异常类型表)”中说明，以告知调用方。

每个异常对象都至少包含一个字符串，来说明异常发生的原因或出错性质，称为出错信息。因此大多派生异常类都提供含字符串形参的构造函数。例如：

```

class invalid_argument : public logic_error{
public:
    invalid_argument(const string& what_arg); // 构造函数
};

```

一般地，派生类继承基类的虚函数 what()，返回出错信息。如果需要的话，派生类可改写这个虚函数，以提供更多信息。

类 logic\_error 表示逻辑错误的异常类型，此类错误是在特定代码执行之前就违背了某些前置条件，例如，数据越界 out\_of\_range，函数调用实参无效 invalid\_argument 等，也包括特定领域相关的错误 domain\_error。读者可自行扩展新类型。例如，访问数据的下标越界可作为 out\_of\_range 的派生类。一些逻辑错误意味着编程有误，一般通过改进编程能避免。

类 runtime\_error 表示运行期错误，在程序执行期间才能检测的错误。例如算术运算可能导致上溢出 overflow\_error、下溢出 underflow\_error、数值越界 range\_error 等。读者可自行扩展新的类型，如空指针错误可作为 runtime\_error 的派生类。一些运行期错误有一定偶然性，与执行环境有关，如内存不足、打开文件失败等，此类错误并不能通过改进自身编程来消除。

一个异常类所包含的信息越多，对于此类错误的检测和处理就越有利。例如，要说明下标越界错误，就应该说明该下标的当前值是多少，可能的话，还应说明合理的下标范围。这需要添加新的数据成员以及相应的成员函数。例如：

```

class Index_out_of_range : public out_of_range{
    const int index;
public:
    Index_out_of_range(int index1, const string& what_arg)
        : index(index1), out_of_range(what_arg){}
    int getIndex() const {return index;}
};

```

再如，要说明读取文件到特定位置时发生数据错误，就应该说明文件名、出错位置、所读到的数据等信息。后面将详细介绍这些派生类的设计。

大多数异常派生类都很简短，关键是对异常的识别和命名。

C++异常分为两类：有命名的和未命名的。有命名的异常是有类型的，基本类型(如 int)或字符串(char\*)或自定义类型。未命名的异常是在运行时时刻某种底层错误引起的，例如，整数相除时除数为 0，通过挂空指针或挂空引用来访问对象，破坏当前函数的堆栈使函数返回到错误地址等。未命名异常虽然也能被捕获，但不能提供确切的出错信息。建立异常类型架构本质上就是对各种异常情形的识别与命名，对于异常处理具有重要作用。

## 15.3 异常处理语句

C++语言的异常处理语句包括引发异常语句 `throw` 和捕获处理语句 `try-catch`。

### 15.3.1 `throw` 语句

引发异常语句的语法格式为：

```
throw <表达式>;或者 throw;
```

其中，关键字 `throw` 表示要引发一个异常到当前作用域之外。表达式值的类型作为异常事件的类型，并将表达式的值传给捕获处理该类型异常的程序。表达式的值可能是一个基本类型的值，也可能是一个对象。如果要引发一个对象，对象类应事先设计好。一个类表示了一种异常事件，应描述该类异常发生的原因、语境以及可能的处理方法等。

不带表达式的 `throw` 语句出现在 `catch` 子句中，重新引发当前正在处理的异常。

如果在一个函数编程中发现了自己不能处理的错误情形，就可使用 `throw` 语句引发一个异常，将它引发到当前作用域之外。如果当前作用域是一个函数，就将异常传递给函数的调用方，让调用方来处理。

`throw` 与 `return` 相似，表达式也相似，都会中止后面代码的执行。`throw` 语句执行将控制流转到异常捕获语句处理。这将导致 `throw` 语句下面相邻语句不能执行，而且会自动回收当前作用域中的局部变量。例如：

```
throw index;                //引发一个 int 异常，index 是一个 int 变量
throw "index out of range";  //引发一个 const char *异常
throw invalid_argument("denominator is zero");//引发 invalid_argument 异常
```

最后一个 `throw` 语句执行过程是，先创建一个 `invalid_argument` 对象，然后再将该对象引发到当前作用域之外。

`throw` 与 `return` 的含义不同。一个函数的返回值表示正常执行的结果，要作为显式说明的函数规范。`throw` 语句虽然也能终止当前函数的执行，但表示不正常的执行结果。一个函数只有一种返回类型，但可能引发多种类型的异常。

为了说明一个函数可能引发哪些类型的异常，可用异常规范(exception specification)来说明，就是“`throw(异常类型表)`”。例如下面是一个求商函数：

```
double quotient(int numrator, int denominator) throw(invalid_argument){
    if (denominator == 0)
        throw invalid_argument("denominator is zero");
    return double(numrator) / denominator;
}
```

该函数的第一个形参除以第二个形参，返回商作为结果。该函数的原型中包含了异常规范：`throw(invalid_argument)`，说明该函数的调用可能引发 `invalid_argument` 异常。函数体中检查第二个形参(即除数)，如果除数为 0，就引发该异常。

尽管异常规范目前还起不到语法检验的作用，但起码能告知函数的调用方注意捕获哪些类型的异常，而不是仅仅等待函数的返回值。

异常对函数设计具有重要作用。传统的 C 函数设计不用异常。如果函数有多种结果，返

回类型只能有一个，就要添加形参来表示其它结果。添加的形参往往是指针类型，在调用时要提供变量地址作为实参，在调用返回之后再判断得到什么结果。例如，一个求商函数可能设计如下：

```
double quotient(int numerator, int denominator, int * isValid){
    if (denominator == 0){
        *isValid = 0;          //用 0 表示无效除数
        return 0;
    }
    *isValid = 1;              //用 1 表示有效除数
    return double(numerator) / denominator;
}
```

上面函数中商作为返回值，添加了一个引用形参来表示除数是否有效。另一种可行的 C 函数设计是返回一个 `int` 值，0 表示无效除数，1 表示有效除数，而将商作为形参，如下所示：

```
int quotient(int numerator, int denominator, double* result){
    if (denominator == 0)
        return 0;
    *result = double(numerator) / denominator;
    return 1;
}
```

可以看出，传统的 C 函数设计把除数为 0 作为一种特殊情形，用 `if` 语句加以判断处理，需要更多的函数形参，导致函数定义复杂化。另一方面，调用方在调用函数返回之后，就要立即用一个 `if` 语句来判断返回值是否为 1，如为 1，商才有效，如不为 1，商无效。这对调用方有一种强迫性，必须立即做出判断，这将导致程序逻辑复杂化，而且难以清晰表达正常流程。

异常是 C++ 提供的一种新概念，表示了偏离正常流程的小概率事件。异常不应该使正常流程的描述复杂化，也不应该让调用方忽视可能发生的异常。调用方可以选择在适当的地方集中捕获处理多种异常，就要用到 `try-catch` 语句。

使用 `throw` 语句，应注意以下要点：

(1) 根据当前异常情形，应选择更准确、更具体的异常类型来引发，而避免引发抽象的类型。例如，如果在 `new` 申请内存之后，如果发现返回空指针，此时应引发 `OutOfMemory` 类型的异常，而不是 `NullPointer` 异常，也不是更抽象的 `runtime_error` 或者 `exception`。准确具体的异常信息对于调用方的处理非常重要，否则就可能导致误解。

(2) 如果一个函数中使用 `throw` 语句引发异常到函数之外，应该在函数原型中用异常规范准确描述，即“`throw(异常类型表)`”，使调用方知道可能引发的异常类型，提醒调用方注意捕获异常并及时处理。

(3) 虽然 `throw` 语句可以在函数中任何地方执行，但应尽可能避免在构造函数、析构函数中使用 `throw` 语句，因为这将导致对象的构建和撤销过程中出现底层内存错误，可能会导致程序在捕获到异常之前就被终止。后面 15.9 节将分析其原因。

(4) 一般来说，异常发生总是有条件的，往往在一条 `if` 语句检测到某个假设条件不成立时，才用 `throw` 语句引发异常，以阻止下面代码执行。在一个函数中无条件引发异常，只有一个理由，就是不想让其它函数调用，例如，一些实体类的拷贝构造函数和赋值操作函数如果不想被调用，就将这些函数设为私有，同时用一条 `throw` 语句避免本类其它函数执行。



### 15.3.2 try-catch 语句

捕获处理异常的语句是 try-catch 语句，一条 try-catch 语句由一个 try 子句(一条复合语句)和多个 catch 子句组成。一个 catch 子句包括一个异常类型及变量和一个异常处理器(一条复合语句)。语法格式如下：

```
try{
    可能引发异常的语句序列;           //受保护代码
}catch(异常类型 1 异常变量 1){
    处理代码 1;                         //异常处理器 1
}catch(异常类型 2 异常变量 2){
    处理代码 2;                         //异常处理器 2
}...
}catch(...){
    处理代码;                           //异常处理器
}
```

其中，关键字 try 之后的一个复合语句称为 try 子句。这个复合语句中的代码被称为受保护代码，包含多条语句。受保护代码描述正常的执行流程，但这些语句的执行却可能引发异常。如果执行没有发生异常，try-catch 语句就正常结束，开始执行其下面语句。如果引发了某种类型的异常，就按 catch 子句顺序逐个匹配异常类型，捕获并处理该异常。如果异常被捕获，而且处理过程中未引发新的异常，try-catch 语句就正常结束。如果异常未被捕获，该异常就被引发到外层作用域。图 15.3 表示了 try-catch 语句的组成结构。

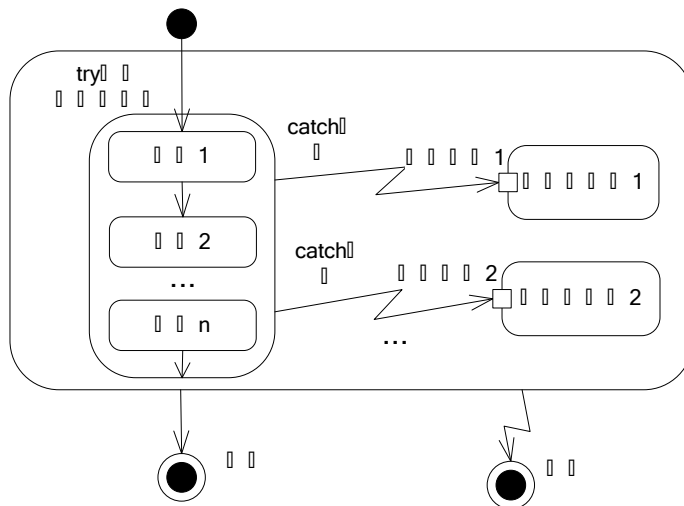


图 15-3 try-catch 语句的组成结构

一条语句执行引发异常，有以下 3 种可能的原因：

- 1、该语句是 `throw` 语句。
- 2、调用函数引发了异常。
- 3、表达式执行引发了未命名的异常，如整数除数为 0、挂空访问等。

例如下面 `try-catch` 语句，调用了前面介绍的求商函数 `quotient`。

```
try{
    result = quotient(n1, n2);                //A
    cout<< "The quotient is "<<result;        //B
    //...
} catch (invalid_argument ex){                //C
    cout<<"invalid_argument:"<<ex.what();    //D
}
```

A 行调用 `quotient` 函数，如果没有引发异常，就执行 B 行，然后 `try-catch` 语句就执行完毕。如果 A 行引发了某种异常，B 行就不执行，从 C 行开始匹配异常类型，因为 A 行函数调用可能引发的异常类型正式 `catch` 子句要捕获的异常类型 `invalid_argument`，故此该异常对象就替代了 `ex` 形参，之后再执行后面的一个复合语句，D 行调用异常对象 `ex` 的成员函数得到错误信息，然后打印出来。`try-catch` 语句执行完毕。无论是否发生异常，这个 `try-catch` 语句都能执行完毕，下面语句都能执行。

异常是按其类型进行捕获处理的。一个 `catch` 子句仅捕获一类异常。一个 `catch` 子句由一个异常类型及变量和一个异常处理器(一条复合语句)构成。异常类型及变量指明要捕获的异常的类型，以及接受异常对象的变量。例如 `catch(invalid_argument ex)`，要捕获的异常类型为 `invalid_argument`，如果真的捕获到该类异常，那么变量 `ex` 就持有这个异常对象，这个对象就是前面用 `throw` 语句引发出来的。

有一种特殊的 `catch` 子句，就是 `catch(...)`，该子句能匹配任何类型的异常，包括未命名的异常，不过异常对象或值不能被变量捕获，故此不能提供确切的错误信息。在多个 `catch` 子句中，这种 `catch` 子句应该排在最后。

在执行 `try` 子句中的受保护代码时，如果引发一个异常，系统就到 `catch` 子句中寻找处理该异常类型的入口。这种寻找过程称为异常类型匹配。按如下步骤进行：

(1) 由 `throw` 语句引发异常事件之后，系统依次检查 `catch` 子句以寻找相匹配的处理异常事件入口。如果某个 `catch` 子句的异常类型说明与被引发出来的异常事件类型相一致，该异常就被捕获，然后执行该子句的异常处理器代码。如果有多个 `catch` 子句的异常类型相匹配，按照前后次序只执行第一个匹配的异常处理代码。因此较具体的异常派生类应该匹配在前，以提供最具体详细信息，而较抽象的异常基类应排在后面。

(2) 若没有找到任何相匹配的 `catch` 子句，该异常就被传递到外层作用域。如果外层作用域是函数，就传递到函数的调用方。

一个异常的生命期从创建、初始化之后，被 `throw` 引发出来，然后被某个 `catch` 子句捕获，其生命期就结束了。一个异常从引发出来到被捕获，可能穿越多层作用域或函数调用。如果到 `main` 函数都未被捕获，将导致程序被迫终止。

从图 15.3 中可以看出，`try-catch` 语句的执行结果有两个：正常和异常。表 15.1 分析了 `try-catch` 语句的 4 种具体情形。

表 15-1 try-catch 语句执行结果

序号	结果	具体情形
1	正常完毕	受保护代码未引发异常
2	正常完毕	受保护代码引发了异常，但异常被某个 catch 子句捕获
3	异常退出	受保护代码引发了异常，但未被 catch 子句捕获
4	异常退出	受保护代码引发了异常，而且被某个 catch 子句捕获，但在异常处理器中又引发了新的异常，或者用“throw;”语句把刚捕获的异常又重新引发出来

分析下面 try-catch 语句的可能结果：

```
try{
    result = quotient(n1, n2);
    cout<< "The quotient is "<<result;
    //...
}catch(invalid_argument ex){
    cout<<"invalid_argument:"<<ex.what();
}catch(logic_error ex){
    cout<<"logic_error:"<<ex.what();
}catch(exception ex){
    cout<<"exception:"<<ex.what();
}catch(...){
    cout<<"some unexpected exception";
}
```

上面 try 子句中调用了可能引发异常的函数 quotient。这个 try 语句包含了 4 个 catch 子句，这 4 个 catch 子句的次序是较具体的派生类放在前面，较抽象的基类放在后面。最后一个 catch 子句可匹配捕获任意类型的异常，但因得不到异常对象，故此不能提供更多信息。在一次执行时，如果引发异常，只能有一个 catch 子句捕获处理该异常。由于最后一个 catch 子句能捕获所有类型的异常，而且所有的异常处理器代码中都不会引发异常，因此该 try-catch 语句的执行结果是表中第 1 种或者第 2 种情形。

对于 try-catch 语句的理解和应用，应注意以下几点。

(1)try 子句中的代码，称为受保护代码，实际上是受到下面若干 catch 子句的保护，使得 try 子句代码可以放心去描述正常处理流程，而无需每执行一步都要用 if 语句来判断是否发生异常情形。

(2)并非 try 子句都可能引发异常，也并非 catch 子句要捕获 try 子句所引发的所有异常，当前函数只需捕获自己能处理的异常。

(3)多个 catch 子句之间，不允许基类异常在前、派生类在后，否则将出现语法警告，这使得列在后面的派生类捕获不到异常，而排在前面的基类先捕获到了。

(4)try-catch 语句仅适合处理异常，并不能将其作为正常流程控制。

(5)相对于 Java 语言的 try-catch-finally 语句，C++的 try-catch 缺少 finally 子句。VC2015 自行扩展了 finally 子句，但仅限于 CLR(Common Language Runtime)使用。

### 15.3.3 异常处理的例子

例 15-1 控制流程测试。

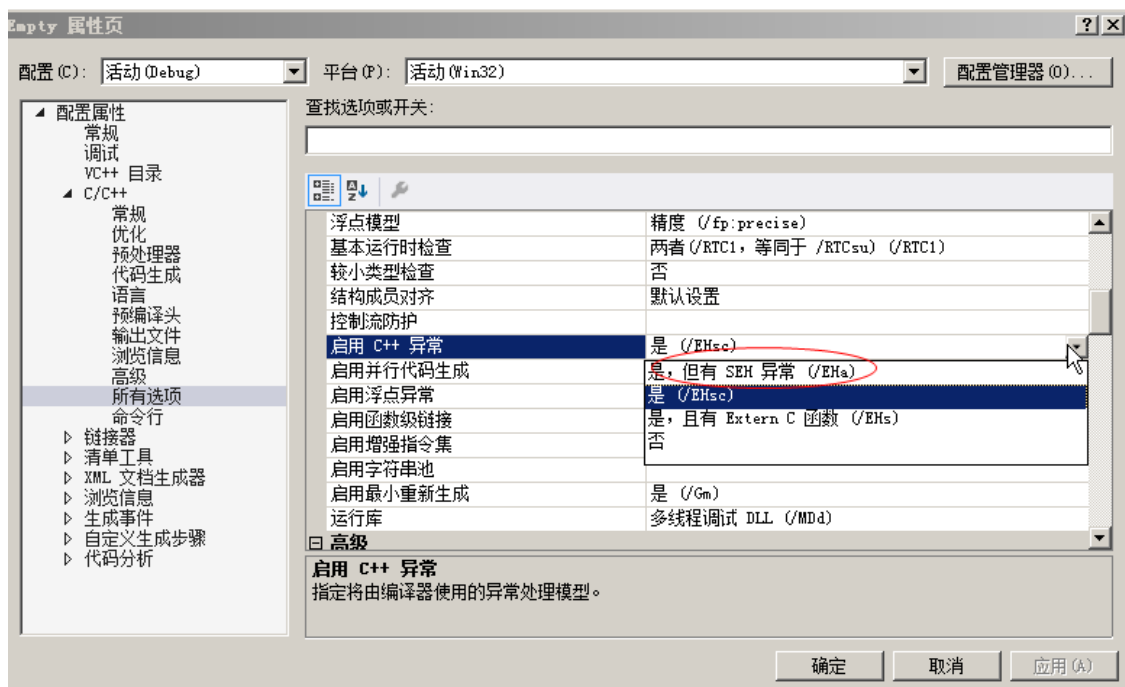
```
#include <iostream>
```

```
using namespace std;
void testExcept(int i){
    try{
        if (i == 1)
            throw "catch me when i == 1";           //A
        if (i == 2)
            throw i;                                 //B
        if (i == 0){
            int d = (i+1) / i;                        //C
            cout<<d<<endl;                          //D
        }
        cout<<"i="<<i;
    }catch(int i){                                   //E
        cout<<"catch an int: "<<i;
    }catch(char * ex){                             //F
        cout<<"catch a string:"<<ex;
    }catch(...){                                    //G
        cout<<"catch an exception unknown";
    }
    cout<<"\tfunction return\n";                    //H
}

int main(){
    testExcept(0);      //C 行引发异常，G 行捕获
    testExcept(1);      //A 行引发异常，F 行捕获
    testExcept(2);      //B 行引发异常，E 行捕获
    testExcept(3);      //未引发异常
    system("pause");
    return 0;
}
```

上面 try-catch 语句中的受保护代码中可能引发 3 种类型的异常。A 行引发 `const char *` 类型的异常，B 行引发 `int` 异常，C 行执行整数除法时，除数为 0，将引发底层的未命名的异常，故此 D 行不会执行。

注意 VS2015 中如果未添加编译选项 /EHa，会导致 catch(...) 不能捕获 C 行被零除的异常，将导致程序终止。如果需要 catch(...) 能捕获所有异常，应选择编译选项 /EHa，如下图所示。



上面 try-catch 语句包含 3 个异常处理器。E 行捕获 int 异常，F 行捕获 char\* 异常，G 行捕获底层未命名异常。注意，第 1 个和第 2 个异常处理器的次序无所谓，但第 3 个异常处理器必须排在最后，因为排在它后面将捕获不到任何异常。H 行是 try-catch 语句后的一条语句，如果 try-catch 语句正常退出，那么 H 语句将执行。如果受保护代码中引发的异常未被捕获，或者异常处理代码中引发异常，H 语句都不能执行。

执行程序，输出如下：

```
catch an exception unknown      function return
catch a string:catch me when i == 1    function return
catch an int: 2 function return
i=3      function return
```

在主函数中测试了 4 种情形，前 3 次调用都引发异常而且被捕获，最后一次调用未引发异常。上面例子中的异常是直接 using throw 语句或者除数为 0 的表达式来引发的，异常也可能是因调用函数而引发的。

例 15-2 测试向量 vector<T> 的下标越界异常。标准模板库 STL 提供的向量 vector<T> 是支持元素随机访问的一种常用容器，它有两种按下标随机访问形式：operator[] 和 at()，后者可引发 out\_of\_range 异常。

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    try{
        vector<int> vec(4);           //A
        int i = 0;
        for(i = 0; i < 4; i++)        //B
            vec[i] = i + 1;
```

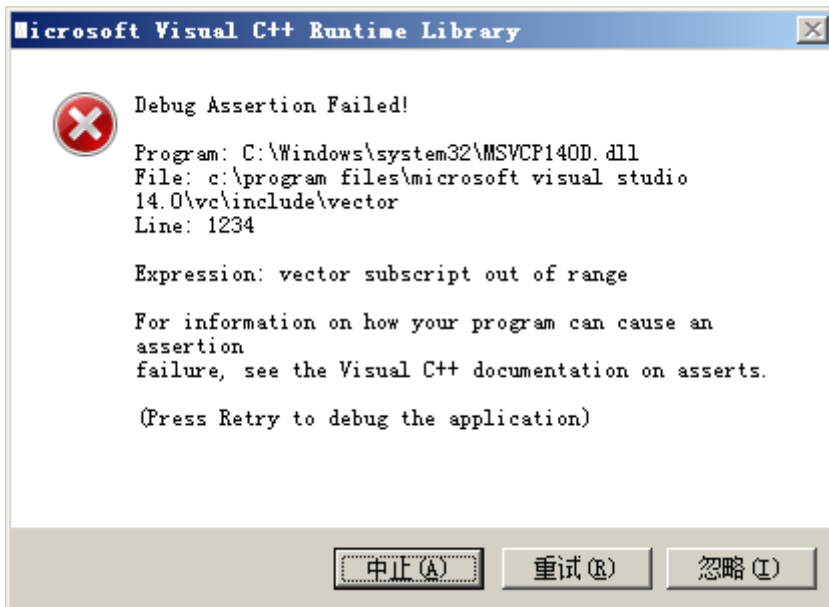
```
for(i = 0; i <= 4; i++)
    cout<<vec[i]<<" ";    //C no exception
cout<<endl;
for(i = 0; i <= 4; i++)
    cout<<vec.at(i)<<" "; //D throw exception when i==4
cout<<endl;
} catch(out_of_range ex) {
    cout<<"out of range:"<<ex.what()<<endl;
} catch(...) {
    cout<<"unexpected\n";
}
system("pause");
return 0;
}
```

执行程序，输出如下：

```
1 2 3 4 -33686019
1 2 3 4 out of range:invalid vector<T> subscript
```

上面程序测试两种按下标随机访问元素的成员函数。A 行先创建了一个向量，包含 4 个 int 元素。B 行对这 4 个元素初始化。C 行调用 operator[] 来访问元素，输出第 1 行，当下标越界时，并没有引发任何异常，只是读取的 vec[4] 元素的值是随机值。D 行调用 at(int index) 来访问元素，输出第 2 行。当下标越界时，引发了 out\_of\_range 异常，而不会按非法下标读取值。

注意在 VS2015 中的 Debug 模式下，C 行引发的异常并未被捕获，导致程序终止并提示如下：



原因是 VS2015 的 Debug 模式中生成代码中添加了下标检查断言，断言失败导致程序终止。将 Debug 模式改为 Release 模式，重新生成后再运行，C 行不再引发异常。

例 15-3 除数为 0 的异常处理。在整数除法中，如果除数为 0 就引发底层未命名异常，因此有必要在除法执行之前判断除数是否为 0，如果除数为 0 就引发一个命名的异常来通知调

用方。编程如下：

```
#include <iostream>
#include <stdexcept>
using namespace std;
double quotient(int numerator, int denominator) throw(invalid_argument){
    if (denominator == 0)
        throw invalid_argument("denominator is zero"); //A
    return double(numerator) / denominator;
}
int main(){
    int n1, n2;
    double result;
    cout<<"Enter two ints(end-of-file ^Z to end):";
    while (cin>>n1>>n2){
        try{
            result = quotient(n1, n2);
            cout<< "The quotient is "<<result;
        }catch(invalid_argument ex){ //B
            cout<<"invalid_argument:"<<ex.what();
        }catch(logic_error ex){ //C
            cout<<"logic_error:"<<ex.what();
        }catch(exception ex){ //D
            cout<<"exception:"<<ex.what();
        }catch(...){ //E
            cout<<"some unexpected exception";
        }
        cout<<"\nEnter two ints(end-of-file ^Z to end):";
    }
    system("pause");
    return 0;
}
```

执行程序，输出如下：

```
Enter two ints(end-of-file ^Z to end):23 45
The quotient is 0.511111
Enter two ints(end-of-file ^Z to end):34 0
invalid_argument:denominator is zero
Enter two ints(end-of-file ^Z to end):34 56
The quotient is 0.607143
Enter two ints(end-of-file ^Z to end):^Z
```

执行了 3 次求商函数，其中第 2 次除数为 0，此时 A 行引发了 `invalid_argument` 类型的异常，那么 B 行的 `catch` 子句就捕获了该类异常。做如下修改、再测试验证：

- 1、如果修改 A 行，将 `invalid_argument` 改变为 `logic_error`，那么 C 行的 `catch` 子句就捕获了该类异常。
- 2、如果再修改 A 行，改变为 `exception`，那么 D 行的 `catch` 子句就捕获了该类异常。
- 3、如果删除 B、C、D 行的 `catch` 子句，那么 E 行的 `catch` 子句就捕获了该类异常。

- 4、如果删除所有的 catch 子句，那么异常将引发到 main 函数之外，由系统给出一个提示如图 15.4。这个对话框中不能给出有效错误信息，只能“中止”程序。

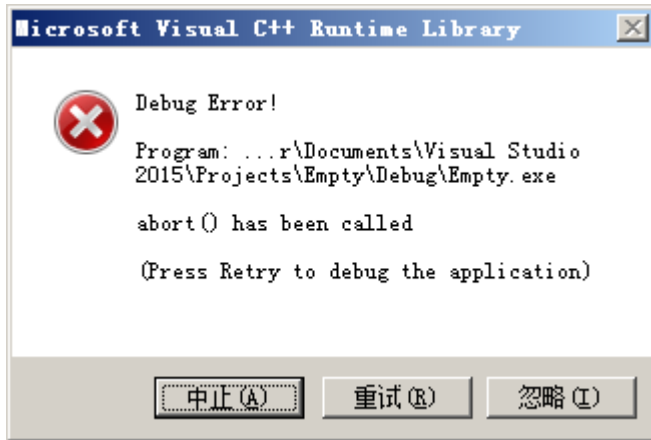


图 15.4 main 函数未捕获的异常提示

这个例子说明了因调用函数而引发异常的捕获。这个例子也说明了 catch(...)可以捕获任何类型的异常，但编程中往往采用另一种方法来处理未捕获的异常。

#### 15.3.4 C11 修饰符 noexcept

上面例 15-3 的函数说明：

`double quotient(int numerator, int denominator) throw(invalid_argument);`

其中 throw(异常类型)称为函数的异常规范，说明该函数可能引发的异常类型。在 C++11 标准中被废弃，主要原因是编译时对调用方是否捕获该异常不起任何约束作用。VS2015 编译仅给出警告。

C11 认为说明一个函数不会引发任何异常对于调用方更有意义。这样调用方就能放心调用而无需操心如何捕获处理异常。为此 C11 提出 noexcept 修饰符，一个新的关键字。语法有下面两种形式：

`ReturnType FuncName(params) noexcept;`

`ReturnType FuncName(params) noexcept(expr 常量表达式);`

第 1 种形式说明该函数不会引发异常，相当于原先的 throw()说明。当调用此函数时，调用方无需考虑捕获异常。当该函数在运行时确实引发了异常，该异常将导致调用 terminate()终止处理器来终止该函数，以防止异常传播到调用方，阻断了异常传出。

第 2 种形式是“有条件的无异常”。带有一个常量表达式，将得到一个逻辑值。如果为真，则 except(true)等同于 noexcept，异常会被拦截并终止程序。如果为假，则说明该函数可能引发异常并传播到调用方，调用方应捕获异常，但未说明会引发何种类型的异常。

例 15-4 修饰符 noexcept 用法。

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



```

void Throw() { throw 1; }
void NoBlockThrow() noexcept(false){ Throw(); }           //A
void BlockThrow() noexcept { Throw(); }                   //B
int main() {
    try {
        Throw();
    }catch (...) {
        cout << "Found throw." << endl;    // Found throw.
    }
    try{
        NoBlockThrow();
    }catch (...) {
        cout << "Throw is not blocked." << endl; // Throw is not blocked.
    }
    try {
        BlockThrow();
    }catch (...) {
        cout << "Found throw 1." << endl;        //C
    }
    system("pause");
    return 0;
}

```

A 行说明 **noexcept(false)**，与省略一样效果。B 行说明该函数无异常传到调用方，导致 C 行 catch 子句捕获不到该异常。

执行程序，输出如下：

```

Found throw.
Throw is not blocked.

```

然后就是中止程序对话框。

修饰符 **noexcept** 的好处是阻断异常传播，当异常发生时尽可能本地处理，使程序能掌握控制流程的主动权。

## 15.4 终止处理器

如果引发的异常未能被捕获，程序将被迫终止。在程序终止之前，系统提供了终止处理器(terminate handler)，提供一个机会来清理系统资源，然后再终止程序。在<eh.h>和<exception>中分别提供了 **terminate()**函数，前者是老版本，作为全局函数，后者是新版本，定义在 std 命名空间之中。

在发生下面情形之一时将自动执行 **terminate()**函数：

- 1、引发异常最终未能捕获。
- 2、析构函数在系统堆栈释放内存时引发异常。
- 3、在引发某个异常之后系统堆栈遭到破坏。

缺省的 **terminate** 函数将调用 **abort** 函数，但 **abort** 函数不执行清理而简单终止程序，因此常常需要自行定义一个函数，使 **terminate** 函数能调用。这要先准备一个无参且无返回的函数 **f**，然后调用 **set\_terminate(f)**，将函数 **f** 作为终止处理器。

例 15-5 **terminate** 函数的例子。

```

#include <exception>
#include <iostream>
using namespace std;
void term_func() { //A
    cout << "term_func() was called by terminate().\n";
    // ... cleanup tasks performed here
    // If this function does not exit, abort is called.
    system("pause");
    exit(-1);
}
int main() {
    int i = 10, j = 0, result;
    set_terminate( term_func ); //B
    try {
        if( j == 0 )
            throw "Divide by zero!"; //C
        else
            result = i/j;
    } catch(int) {
        cout << "Caught an integer exception.\n";
    }
    cout << "This should never print.\n";
    system("pause");
    return 0;
}

```

执行程序，输出如下：

```
term_func() was called by terminate().
```

A 行定义了一个函数要作为终止处理器，B 行调用 `set_terminate` 将此函数作为终止处理器。在头文件 `<exception>` 中的相关定义如下：

```

typedef void (*terminate_handler)(); //函数指针类型，终止处理器
terminate_handler set_terminate(terminate_handler ph) throw();
void terminate();

```

头一行说明了一种函数指针的类型名，第二行说明了一个函数 `set_terminate`，将一个函数 `ph` 说明为新的终止处理器。最后一行是异常处理器函数，缺省将调用 `abort` 函数。

C 行引发的异常类型为 `const char*`，显然不能被下面的 `catch` 子句捕获，该异常将导致程序终止，将执行 `terminate` 函数，因为 B 行设置了新的终止处理函数 `term_func`，那么新的函数得到执行。通常情况下，设置终止函数的目的是释放资源，然后调用 `exit` 函数来终止程序。

## 15.5 扩展新的异常类型

虽然前面图 15.2 给出了一个异常类型架构，但经常需要扩展自己的异常类型。例如，虽然 `out_of_range` 类能用于说明下标越界，但未说明发生异常的下标究竟值是什么。再如，前面例子中除数为 0 的异常使用了 `invalid_argument` 类，而实际上除数为 0 可能有多种情形，而不一定都作为函数实参，因此有必要自行定义除数为 0 的异常类。

图 15.4 给出了一组扩展的异常类型。扩展异常类主要是以 `logic_error` 和 `runtime_error` 为基类来定义派生类。下面构造了一个头文件 `exceptions.h`，包含了一组常用的异常类。

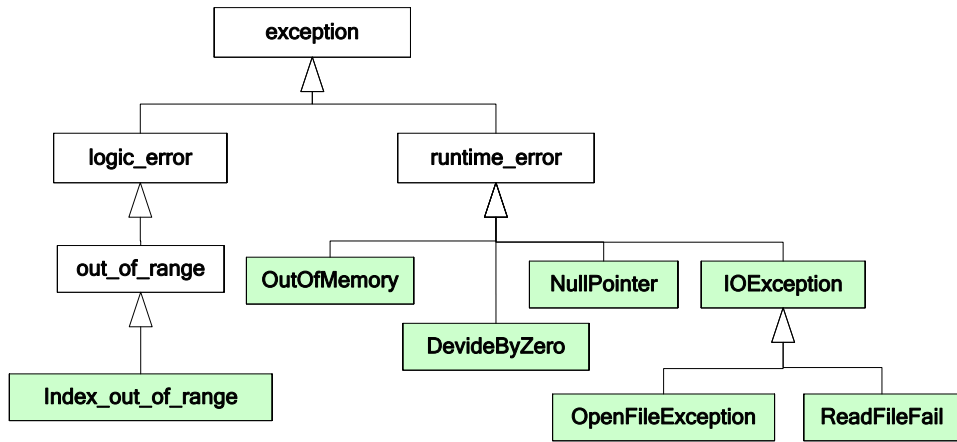


图 15-4 扩展异常类型

```

#ifndef EXCEPTIONS
#define EXCEPTIONS
#include <string>
#include <stdexcept>
using namespace std;
//下标越界,记录下标
class Index_out_of_range : public out_of_range{
    const int index;
public:
    Index_out_of_range(int index1, const string& what_arg)
        :index(index1), out_of_range(what_arg){}
    int getIndex()const {return index;}
};
//除数为0
class DivideByZero : public runtime_error{
public:
    DivideByZero(const string& what_arg)
        :runtime_error(what_arg){}
};
//空指针
class NullPointer : public runtime_error{
public:
    NullPointer(const string& what_arg)
        :runtime_error(what_arg){}
};
//无可用内存
class OutOfMemory : public runtime_error{
public:
    OutOfMemory(const string& what_arg)
        :runtime_error(what_arg){}
};
//一般 IO 异常的基类
class IOException : public runtime_error{
public:
    IOException(const string& what_arg)
        :runtime_error(what_arg){}
};
  
```

```

};
//打开文件失败, 记录文件名
//可能是要读的文件不存在, 也可能是要写的文件不能创建
class OpenFileException : public IOException{
public:
    OpenFileException(const string& filename)
        :IOException(filename){}
};
//读取文件到特定位置时转换失败, 记录文件出错位置
//出错位置可以是文本文件的数据位置, 第 n 项出错
//也可以是二进制文件的字节位置, 第 n 个字节出错
class ReadFileFail : public IOException{
    const long errPos;
public:
    ReadFileFail(long pos, const string& what_arg)
        :errPos(pos),IOException(what_arg) {}
    const long getErrPos()const{return errPos;}
};
#endif

```

读者可自行扩展合适的派生类, 以适合软件开发的具体需要。下面部分例子要使用这些异常类型。

## 15.6 异常类型的应用

利用扩展的异常类型, 就可对许多已有程序进行改进。

例 15-6 设计一个函数从一个文本文件中读取多个浮点数, 放入一个向量 `vector<float>` 中, 显示各元素, 给出元素的个数, 并按升序排序。要读取的文本文件包含任意多的浮点数, 用分隔符分开, 例如:

```

8.9 9.1 10.1 11.2
4.5 5.6 6.7 7.8
0.1 1.2 2.3 3.4

```

这个例子将演示多种异常类型的引发和处理。编程如下:

```

#include <vector>
#include <algorithm>
#include <fstream>
#include <iostream>
#include "exceptions.h"
using namespace std;
void getVectorFromFile(char * filename, vector<float> &vfs){
    if (filename == NULL)
        throw NullPointer("filename is null");
    ifstream ifs(filename, ios::in);
    if (!ifs){
        string msg = "open file:";
        msg += filename;
        msg += " fail for read";
        throw OpenFileException(msg);
    }
    float f;
    while(!ifs.eof()){
        ifs >> f;
        if (ifs.fail()){

```

```

        string msg = "read file fail:";
        msg += filename;
        throw ReadFileFail(vfs.size(), msg);
    }
    vfs.push_back(f);
}
ifs.close();
return;
}

int main(){
    try{
        char filename[200];
        cout<<"input file name to read:";
        cin>>filename;
        vector<float> vf;
        getVectorFromFile(filename, vf);
        cout<<"元素个数: "<<vf.size()<<": "<<endl;
        for (auto &y : vf)
            cout << y << " ";
        cout<<endl;
        sort(vf.begin(), vf.end());
        cout << "升序排序" << endl;
        for (auto &y : vf)
            cout << y << " ";
        cout<<endl;

        }catch(NullPointer ex){
            cout<<ex.what()<<endl;
        }catch(OpenFileException ex){
            cout<<ex.what()<<endl;
        }catch(ReadFileFail ex){
            cout<<ex.what()<<" at "<<ex.getErrPos()<<" item\n";
        }catch(out_of_range ex){
            cout<<"index out "<<ex.what()<<endl;
        }catch (exception ex){
            cout<<ex.what()<<endl;
        }catch (...){
            cout<<"exception unknown"<<endl;
        }
        system("pause");
        return 0;
    }
}

```

函数 `getVectorFromFile` 的第一个形参是一个文件名，第二个形参是 `vector<float>` 引用作为结果。该函数可能引发 3 种命名异常：

- 1、检查形参指针是否为空，可能引发 `NullPointer` 异常；
- 2、打开文件，可能引发 `OpenFileException` 异常，保存了出错的文件名；
- 3、读浮点数，可能引发 `ReadFileFail` 异常，保存了文件读错的位置。

在函数 `getVectorFromFile` 执行过程中只要引发任何一种异常，就不能正常返回。

主函数中使用 `try-catch` 语句来完成计算并捕获处理各种异常。先输入一个文件名，并说明一个 `vector<float>` 变量，再调用函数 `getVectorFromFile` 来得到结果，下面是显示、排序、再显示。

执行程序，在第 1 行输入一个文件名 `array.txt`，输出如下：

```
input file name to read:array.txt
元素个数: 12:
8.9 9.1 10.1 11.2 4.5 5.6 6.7 7.8 0.1 1.2 2.3 3.4
after sorted
0.1 1.2 2.3 3.4 4.5 5.6 6.7 7.8 8.9 9.1 10.1 11.2
```

读者可自行改变输入或改变文本文件，引入各种错误，看程序运行是否能正确判断各种异常。例如：

- 用 NULL 值来调用函数，看是否导致 `NullPointerException`。
- 输入错误的文件名是否会导致 `OpenFileException`。
- 把某个浮点数的字符该为字符，看是否导致 `ReadFileFail`。

`try` 子句中的代码描述了正常执行的逻辑，而各种异常的捕获处理都用 `catch` 子句描述。这样就能将正常流程与异常处理分割开，不仅提高了程序的可读性和可维护性，而且增强了应对多种错误的能力，提高了编程可靠性。

## 15.7 函数设计中的异常处理

在函数设计中何时要用到异常？有以下三个原则：

1、遇到小概率事件，应考虑使用异常。一种小概率事件往往就是一种异常情形，例如，函数的指针形参在调用时却得到了空指针实参。再例如，要输入一个浮点数，但实际输入错误。小概率事件也意味着在可靠性要求不高的前提下可以推迟处理、甚至忽略。前面很多例子都有这样一个前提，即小概率事件不会发生。反之，如果不是小概率事件，就不适合用异常。例如，读文件到文件尾 `eof` 判断，就不适合将读到文件尾作为一种异常来处理，它不是小概率事件，因为每一次读取都应判断是否到达文件尾。

2、遇到某种情形，根据当前信息不能确定应该如何处理，应考虑用异常来通知调用方处理。例如，一个函数从文本文件中读浮点数序列，文件名由形参提供，假如按调用方提供的文件名打开文件失败，应如何处理？此时合理的办法就是告诉调用方，这个文件名打开失败了，由调用方来决定是换一个文件名，还是放弃。反之，对于某种情形，如果函数可以处理而且不违背约定，那么这种情形就不适合作为异常。例如对于堆栈 `stack` 操作 `pop`，只有先判断堆栈不为空，才能弹出 `pop` 元素。堆栈为空这种情形不适合作为异常。

3、向调用方报告的某种结果的描述比较复杂，就应考虑使用异常。传统的 C 语言编程常用不同的 `int` 值来表示各种错误。例如一个函数从文本文件中读浮点数序列，可以让该函数返回一个 `int` 值，而且约定返回 0 表示正常，-1 表示实参空指针，-2 表示打开文件失败，-3 表示读取数据失败等。但返回 -2 时，还应告知打开失败的文件名。当返回 -3 时，不仅要告知文件名，还应告知导致读取失败的具体位置，即第几个元素读取失败，这样才方便调用方有效解决问题，此时就需要用异常来详细描述。

在一个函数设计中，要调用一个可能引发某种异常的函数时，有哪些处理方式？当前函数有下面 4 种选择：

- 1、捕获该异常并进行处理，使自己的调用方不需要捕获处理该异常。应采用 `noexcept` 显式说明。
- 2、捕获该异常，在处理代码中转换为另一种异常，再引发出去，让调用方来捕获处理新

的异常。

- 3、捕获该异常，处理(可能是记录异常发生)，再将捕获到的异常引发出去，让外层调用方来捕获处理。用不带表达式的 `throw` 语句可转发已捕获到的异常。
- 4、不捕获该异常，让外层调用方来捕获处理。可能是没有 `try-catch` 语句，也可能有 `try-catch` 但没有 `catch` 子句能匹配所发生的异常类型。

应采取何种处理方式取决于当前函数所承担的异常处理的责任。第 1 种方式完全承担了异常处理的责任，使调用方可以放心调用而无需关心会发生异常。第 4 种方式则完全不承担责任，调用方必须考虑如何处理间接引发的异常。第 2 种和第 3 种方式介于两者之间，承担了部分责任，能捕获处理异常，也能引发异常。

例 15-7 异常处理流程的例子。

```
#include <iostream>
#include <stdexcept>
using namespace std;
float getValue(int i){ //A
    try{
        if (i < 0)
            throw "index is out of range"; //B throw char *
        if (i == 0)
            throw 3.14f; //C throw float
        if (i == 1)
            throw i; //D throw int
        if (i == 2)
            throw 2.718; //E throw double
        cout<<"in try block, i = "<<i<<endl;
    }catch(int index){ //F catch int
        cout<<"catch int exception:"<<index<<endl;
    }catch(float f){
        cout<<"catch float exception:"<<f<<endl;
        throw; //G throw float
    }catch(char * msg){
        cout<<"catch char* exception:"<<msg<<endl;
        throw exception(msg); //H throw exception
    }
    cout<<"below try-catch, i = "<<i<<endl;
    return i+1;
}

void main(){
    for(int i = -1; i <= 3; i++){
        try{
            float f = getValue(i);
            cout<<"f = "<<f<<endl;
        }catch(float f){
            cout<<"main:catch a float exception:"<<f<<endl;
        }catch(double d){
            cout<<"main:catch a double exception:"<<d<<endl;
        }catch(exception ex){
            cout<<"main:catch an exception:"<<ex.what()<<endl;
        }catch(...){
            cout<<"catch an exception unknown"<<endl;
        }
    }
}
```

执行程序，输出如下(行号是为了方便解释而加入的):





```
//...
```

上面代码将一个 try-catch 语句封装到一个 do-while 语句之中，再设置一个 bool 标记值来控制何时需要重复执行 try 子句。

在一个函数中当检测到异常，或者捕获到异常，往往采用替代的办法来处理。例如，当读取一个浮点数失败时，就约定一个缺省值作为结果。当读取一个浮点数矩阵失败时，就约定一个缺省矩阵作为结果。这种替代法使后面的代码能正常执行。

这种替代法具有一定普遍性，符合事务处理的一般规律。例如，总经理将一项任务交付给一个部门经理来完成，部门经理再将该任务的一部分交付给员工 A 来完成，期望该员工能正常完成。但如果员工 A 遇到不能解决的问题而告知部门经理，此时部门经理就可以选择另换一名员工 B 来完成，或者自己来完成。无论那一种替代，总经理的任务总能完成，而无需关心该任务是如何完成的。

## 15.8 异常可能导致内存泄露

在一个函数执行过程中，因引发一个异常而导致控制流转移到外层作用域。此时当前作用域中可能已创建有若干局部对象，那么这些局部对象将被自动撤销，这些对象的析构函数会被自动调用执行，而且是在该异常被捕获之前。

当函数执行中用 new 创建一个对象之后，在用 delete 撤销该对象之前引发了异常，而且控制流离开了当前函数，此时动态对象就不能自动回收，造成内存泄漏。这是因为指向动态对象的指针作为该函数的局部变量在栈中被清除了。因此在一个函数中，在执行 throw 语句之前，应该先用 delete 撤销先前创建的动态对象，以防止内存泄漏。但如果调用一个可能引发异常的函数，要避免内存泄漏就比较困难了。

例 15-8 虽能捕获异常，但导致内存泄漏。

```
#include <iostream>
#include <math.h>
using namespace std;
double getSqrt(double * d){
    if (*d < 0)
        throw *d;
    return sqrt(*d);
}
int main(){
    try{
        double * d = new double(-3.4);
        double d2 = getSqrt(d);           //A 可能引发异常
        delete d;                         //B 如果 A 引发异常，此行不执行
    }catch(double d){
        cout<<"d < 0:"<<d<<endl;       //C 捕获异常之后，也不能再回收
    }
    system("pause");
    return 0;
}
```

如果 A 行引发了异常，B 行 delete 语句就不能执行，造成内存泄漏，根本原因是控制流离开了当前动态对象指针所在的作用域。

无论一个函数是正常返回，还是异常退出，其内部创建的动态对象都应清理干净，才能避免内存泄漏。解决此问题的一种方法是，用一个函数来取代 `new` 运算符，将指向动态对象的指针保存在一个特别设计的堆栈中(可以称之为清除栈 `cleanup stack`)。当前函数中所有动态创建的对象指针都保存在一块连续空间中。在函数执行中，如果发生异常而要退出当前函数，就将此空间中所有未回收的动态对象都用 `delete` 回收，然后再将控制流转向异常捕获。这样即便程序终止也能避免内存泄漏。

## 15.9 小 结

- 异常 `exception` 就是在程序运行过程中所发生的、难以预料的、不正常的事件，而导致偏离正常流程。异常处理 `exception handling` 就是在运行时刻对异常进行检测、捕获、提示、传递等过程。
- C++ 提供了引发异常的语句 `throw` 和捕获处理异常的语句 `try-catch`。
- 在运行时刻用 `throw` 引发的每个异常都可以描述为一个对象或一个值。在编译时刻，每一种异常都可以描述为一种类型，可能是自定义的类，也可能是简单的整数或字符串。
- 利用类的继承性建立一个异常类型的架构，对所处理的各种错误情形进行抽象和归类；方便扩展新的异常类型；在编程中方便选取正确的异常类型，也方便按类型来捕获处理异常。
- `<exception>` 和 `<stdexcept>` 提供了一个比较现成的架构。类 `logic_error` 表示逻辑错误，类 `runtime_error` 表示运行期错误，各自定义了一组派生类表示具体的异常。
- 引发异常要使用 `throw` 语句，其作用是中止当前控制流，转向该异常的捕获程序。可能到外层作用域，也可能到函数的调用方。如果该异常一直到 `main` 函数都没有被捕获，就要强制终止程序。
- 一条语句执行引发异常，有以下 3 种可能的情形：
  - ◆ 本身就是 `throw` 语句。
  - ◆ 调用函数引发了异常。
  - ◆ 表达式执行引发了底层异常，如整数除数为 0、挂空访问等。
- 捕获异常的语句是 `try-catch`，由一个 `try` 子句(一条复合语句，称为受保护代码)和多个 `catch` 子句组成。一个 `catch` 子句包括一个异常类型及变量和一个异常处理器(一条复合语句)。
- 如果受保护代码执行没有发生异常，`try-catch` 语句就正常结束。
- 如果受保护代码引发了某种类型的异常，就按 `catch` 子句顺序逐个匹配异常类型，捕获并处理该异常。
- 如果异常被捕获，而且处理过程中未引发新的异常，`try-catch` 语句就正常结束。
- 如果异常未被捕获，该异常就被引发到 `try-catch` 语句的外层作用域。
- 异常是按其类型进行捕获处理的。一个 `catch` 子句仅捕获一类异常。注意，基类的 `catch` 子句不能排在派生类之前。

- 有一种特殊的 `catch` 子句 `catch(...)`，该子句能匹配任何类型的异常，包括底层异常。
- 一个异常的生命期从创建、初始化开始，被 `throw` 引发出来，然后被某个 `catch` 子句捕获，其生命期就结束了，从引发到捕获可能穿越多层作用域或函数调用。
- 如果异常未能被捕获，程序将被迫终止。在终止之前，系统将自动调用函数 `terminate()`，来清理系统资源，然后终止程序。该函数缺省调用 `abort()` 函数，但用户可定义一个函数来取代。
- 在函数设计中需要使用异常的情形：(1)处理小概率事件；(2)当前不能确定如何处理而要通知调用方；(3)要告知的情形描述比较复杂，往往需要一个对象。
- 在引发异常之前，应注意先将动态创建的局部对象先回收，否则将导致内存泄露。

## 15.10 练 习 题

1. 对于关键词 `throw`，下面哪一种说法是错误的？
  - A 函数中 `throw` 语句用来引发异常到当前作用域之外。
  - B 函数原型中 `throw` 用来说明该函数可能引发哪些类型的异常。
  - C 用 `throw` 语句引发的异常，一定要用 `try-catch` 语句来捕获，否则语法错误。
  - D `throw` 语句的语法形式与 `return` 语句一样。
2. 如果一条语句执行引发了异常，那么可能的原因不包括下面哪一种情形？
  - A 该语句是 `throw` 语句。
  - B 该语句中调用的函数引发了异常。
  - C 该语句中的表达式计算引发了未命名的异常。
  - D 该语句外层没有 `try-catch` 语句。
3. 对于一条 `try-catch` 语句，下面哪一种说法是错误的？
  - A `try` 子句中的代码可以不引发任何异常。
  - B `try` 子句中的代码所引发的异常，一定要被某个 `catch` 子句捕获，否则语法出错。
  - C 多个 `catch` 子句所捕获的异常，不能出现基类在前，而派生类在后的情形。
  - D `catch(...)` 应该是最后一个 `catch` 子句。
4. 对于一条 `try-catch` 语句的执行结果，下面哪一种说法是错误的？
  - A `try` 子句执行没有引发任何异常，`try-catch` 下面的语句继续执行。
  - B `try` 子句引发了一个异常，该异常被某个 `catch` 子句捕获并处理，`try-catch` 下面的语句继续执行。
  - C `try` 子句引发了一个异常，但没有被任何一个 `catch` 子句捕获，此时 `try-catch` 语句引发异常，下面语句不能继续执行。
  - D 在 `catch` 子句中不能用 `throw` 语句引发异常。
5. 假设 `Index_out_of_range` 是 `out_of_range` 的派生类，`out_of_range` 是 `exception` 的派生类，给出下面程序运行结果。

```
#include <iostream.h>
#include "exceptions.h"
double getValue(int index){
```

```

    if (index < 0 || index > 9)
        throw Index_out_of_range(index, "valid range is [0..9]");
    return index;
}
void main(){
    try{
        double d = getValue(10);
        cout<<"d="<<d<<endl;
    }catch(int index){
        cout<<"index is out of range:"<<index<<endl;
    }catch(out_of_range ex){
        cout<<"catch a out_of_range:"<<ex.what()<<endl;
    }catch(exception ex){
        cout<<"catch an exception:"<<ex.what()<<endl;
    }catch(...){
        cout<<"catch an exception unknown"<<endl;
    }
}

```

6. 给出下面程序的执行结果，分析程序中有哪些语句不可能执行，分析前 3 个 catch 子句改变次序是否会影响执行结果。

```

#include<iostream.h>
void expTest(int i){
    try{
        if (i == 1)
            throw "catch me when i == 1";
        if (i == 2)
            throw i;
        if (i == 3)
            throw 3.14;
        if (i == 0){
            int d = (i+1) / i;
            cout<<d<<endl;
        }
        cout<<"i="<<i;
    }catch(double d){
        cout<<"catch a double:"<<d;
    }catch(int i){
        cout<<"catch an int: "<<i;
    }catch(char * ex){
        cout<<"catch a string:"<<ex;
    }catch(...){
        cout<<"catch an exception unknown";
    }
    cout<<"\tfunction return\n";
}

void main(){
    expTest(0);
    expTest(1);
    expTest(2);
    expTest(3);
    expTest(4);
}

```

7. 以 `IOException` 类为基类，扩展一个派生类表示矩阵读写错误，记录出错的行和列。编程如下：

```

class MatrixException : public IOException{
    const int row, col;
public:

```

---

```
MatrixException(int row, int col, const string& what_arg)
    :row(row), col(col), IOException(what_arg){}
const int getRow()const{return row;}
const int getCol()const{return col;}
};
```

(a) 编写一个函数从一个文本文件中读取 3 行 4 列 float 值，得到一个 TMatrix 对象。

(b) 分析该函数中可能引发哪些类型的异常，并引发异常。在 main 函数中通过键盘输入一个文件名，调用该函数，捕获异常并给出提示。

(c) 修改程序，当发生异常时，尝试再输入文件名，再打开文件进行处理。



## 第16章 C++标准语法补充

大多数 C++ 语言系统都支持 ANSI/ISO C++ 标准。本章介绍该标准中的部分内容，作为前面各章节的语法补充。本章将介绍静态断言、常量表达式 `constexpr`、命名空间 `namespace`、两个修饰符 `explicit` 和 `mutable`、运行时刻类型信息 RTTI 和 `typeid` 运算符、以及 4 种新型的强制类型转换运算符。本章各部分之间相对独立。

### 16.1 静态断言 `static_assert`

第 5 章介绍基于有参宏的动态断言 `assert`，运行时检查特定条件是否满足。

C11 引入了静态断言，在编译时检查特定条件是否满足。语法形式如下：

**`static_assert(const_expr, 提示串文字);`**

其中 `const_expr` 是一个常量表达式，转换为一个逻辑值。若为 0 表示假，断言失败，后面串文字就作为编译错误消息。若为非 0 则为真，则编译没有副作用。例如：

`static_assert(sizeof(int) == 8, "This is a 64-bit machine!");`

静态断言不依赖任何头文件。VS2015 与 DevC++ 都支持。

静态断言是由编译器执行的。VS2015 在代码编辑器上就能直接执行，把鼠标放在静态断言上就能看到执行结果，如下图所示。

```
static_assert(sizeof(int) == 8, "This needs a 64-bit machine!");
```

错误: 静态断言失败, 原因是 "This needs a 64-bit machine!"

如果表达式中不包含模板实参，则立即执行。VS2015 代码编辑器中书写完就能看到结果。

如果表达式中包含模板实参，每次模板实例化时就进行一次计算判断。启动编译后才能看到结果。

静态断言在调试模板时有用，因为模板实参可包含在常量表达式中。看下面例子：

```
#include <cstring>
using namespace std;
template <class T, class U>
int bit_copy(T& a, U& b) {
    static_assert(sizeof(b) == sizeof(a), //A
        "the parameters of bit_copy must have same width.");
    memcpy(&a, &b, sizeof(b));
};
int main() {
```



```
    assert(true);  
    int a = 0x2468;  
    double b;  
    float c;  
    bit_copy(a, c); //B 编译无错  
    bit_copy(a, b); //C 编译导致静态断言<int,douuble>  
    return 0;  
}
```

A 行的静态断言包含了模板实参 `a` 和 `b`，检查大小是否一致。B 行是实参为 `<int,float>`，模板实例化未产生编译错误，但 C 行实参为 `<int,double>` 导致 A 行断言失败，编译出错。

静态断言可用于命名空间中、或类中、或代码块中。

尽管静态断言没有引入新的命名，但静态断言属于说明语句。

在程序运行时刻，静态断言不执行，因此也没有运行负担，而动态断言在运行时执行，有运行负担。

静态断言的表达式中不能包含 `const` 修饰的函数形参。例如：

```
int positive(const int n) {  
    static_assert(n > 0, "value must >0");  
    //...  
}
```

上面静态断言编译语法出错，并非断言失败。

## 16.2 修饰符 `constexpr`

要确定某个值，应该明确选择是在编码期、编译期，还是运行期来确定。

比如需要定义一个数组时，可以在编码期确定其大小，也可推迟到编译期，就需要 `constexpr` 修饰的函数来计算其大小。如果在运行期，就应使用动态内存或某种 STL 容器(不用确定其大小)。

从性能和效率上看，某些值确定得越早，解决方案就越简单，性能就越高。反之，如果确定得越晚，所需资源就更多，性能和效率不会更高。

对于一些值和计算，传统 C++ 编程只能在运行期确定，而实际上可能在编译期就能确定。C11 提出 `constexpr` 关键字来支持编译期计算，C14 又放松了一些限制，增强了编译期计算和不变性约束，改善性能的同时减少内存开销。

`constexpr` 用于修饰一个变量、函数或构造函数，说明被修饰的值或函数返回值是常量，并且如果可能，就在编译时计算值或返回值。语法形式如下：

```
constexpr 类型名 标识符 = 常量表达式;    //修饰变量  
constexpr 类型名 标识符{常量表达式};    //修饰变量  
constexpr 返回类型 函数名(形参表);        //修饰函数，形参也是 constexpr  
constexpr 构造函数 (形参表);              //修饰构造函数，形参也是 constexpr
```

`constexpr` 与 `const` 的相同之处在于，都可修饰变量。如果任何代码试图修改该变量值，编译器将报错。两者不同之处在于，`const` 变量值的确定可以在编译期或者推迟到运行期。而 `constexpr` 修饰的变量值只能在编译期确定，如果不能确定就报错。

当需要 `const` 整数时（比如在模板参数和数组声明中），均可使用 `constexpr` 整数变量。



`constexpr` 与 `const` 第二个区别是，前者可修饰函数和构造函数，并且如果函数调用方在编译期就需要返回值，那么在编译时（而非运行时）就计算这个值。如果不需要，或者调用方提供的是变量实参而非 `constexpr` 实参，就作为普通函数在运行期计算。

`constexpr` 函数涉及到一个新概念，**字面类型 literal type**。字面类型就是其布局 layout 在编译期就可确定的一种类型，有如下形式：

- 1、`void`、标量类型(除 `bool` 之外的基本类型)、标量类型的引用；
- 2、标量类型的数组(或指针)；
- 3、某个类，直接或间接从标量类型构造而来，具有一个或多个 `constexpr` 修饰的构造函数(对析构函数没有要求)，没有移动或拷贝构造函数，所有非静态数据成员与基类也都是字面类型，而且数据成员没有 `volatile` 修饰。

基于字面类型的 `constexpr` 规则如下：

**规则 1**，`constexpr` 修饰的变量，其类型必须字面类型，而且要被初始化。如果由一个构造函数来初始化，该构造函数也应该有 `constexpr` 修饰。

**规则 2**，`constexpr` 修饰的函数，其形参类型和返回类型都必须是字面类型。函数体中要求没有 `goto`，没有 `try-catch`，没有未初始化变量，所有局部变量都是字面类型，没有静态变量，没有 `thread-local` 变量。可以是递归函数，可以有条件语句和循环语句，包括基于范围的 `for` 语句。

**规则 3**，`constexpr` 修饰的函数都隐含着 `inline`(内联函数)。

虽然条件很多，但实际工程中很多函数满足以上要求而未添加 `constexpr` 修饰。C14 标准建议对每个函数尽可能都添加 `constexpr` 修饰，当需要时就能加快执行速度并减少内存开销。

下面例子说明修饰符 `constexpr` 的用法：

```
#include <iostream>
using namespace std;
constexpr float exp(float x, int n){
    return n == 0 ? 1 :
        n % 2 == 0 ? exp(x * x, n / 2) :
        exp(x * x, (n - 1) / 2) * x;
};
constexpr float exp2(const float& x, const int& n){
    return n == 0 ? 1 :
        n % 2 == 0 ? exp2(x * x, n / 2) :
        exp2(x * x, (n - 1) / 2) * x;
};
constexpr int fac(int n) {
    return n == 1 ? 1 : n*fac(n - 1);
}
template<typename T, int N>
constexpr int length(const T(&ary)[N]){
    return N;
}
class Foo{
    int _i;
public:
    constexpr explicit Foo(int i) : _i(i) {}
    constexpr int GetValue() const {
        return _i;
    }
};
```



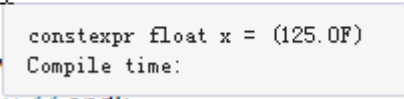


```
int main() {
    //Compile time:
    constexpr float x = exp(5, 3);           //A
    cout << "x=" << x << endl;
    constexpr float y{ exp2(2, 5) };         //B
    cout << "y=" << y << endl;
    constexpr int f5 = fac(5);               //C
    cout << "f5=" << f5 << endl;
    //foo is const:
    constexpr Foo foo(5);                   //D
    // foo = Foo(6); //Error!
    constexpr int val = foo.GetValue();
    const int nums[]{ 1, 2, 3, 4 };
    const int nums2[length(nums) * 2]{ 1, 2, 3, 4, 5, 6, 7, 8 }; //E

    //Run time:
    cout << "The value of foo is " << foo.GetValue() << endl; //F
    system("pause");
    return 0;
}
```

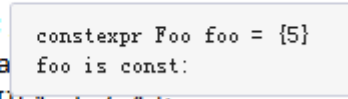
前 3 个函数都是普通的数学计算函数，满足 constexpr 函数要求，就可添加 constexpr 修饰。当主函数中 ABC 行调用这些函数并使用返回值来初始化 constexpr 变量时，这些函数就在编译期执行。在 VS2015 中，鼠标放在 constexpr 变量 x 上，就能看到编译执行的值：

```
28 int main(){
29     //Compile time:
30     constexpr float x = exp(5, 3);
31     cout << "x=" <<
32     constexpr float
33     cout << "y=" << y << endl;
```



上面 Foo 类是一个字面类型，因为它满足字面类型的构造条件。D 行创建的对象 foo 也是 constexpr 变量。鼠标放在 foo 上就能看到“foo is const”

```
36 //foo is const:
37 constexpr Foo foo(5);
38 // foo = Foo(6);
39 constexpr int va
40 const int nums[]{ 1, 2, 3, 4 };
```



E 行调用 length 函数计算前面定义的数组大小=4，然后定义一个新数组，大小=4\*2。

F 行虽然调用了一个 constexpr 成员函数，但调用代码并不需要它在编译期执行，就推迟在运行期执行了。

### 16.3 命名空间 namespace

命名空间(namespace)是解决大程序中多个元素(如类、全局函数和全局变量)命名冲突的一种机制。当我们要把几个部分(往往来自不同的人员或团队)合并成为一个大程序时，往往就会出现命名冲突的问题：类名、全局函数名、全局变量名都可能重名。解决的方法就是把这



些名字放在不同的命名空间中，在访问这些名字时使用各自的命名空间作为限定符。

### 16.3.1 命名空间的定义

命名空间类似文件系统目录，空间中的成员类似目录中的文件。全局空间相当根目录，一个目录名作为其中多个文件的命名空间，子目录作为嵌套空间，文件作为空间中的成员。同时一个命名空间也是一个作用域。

命名空间的基本规则如下：

- 一个程序所用的多个命名空间在相同层次上不重名；
- 在同一个命名空间内的所有成员名字不重复；
- 在一个命名空间中可嵌套定义其内层的多个子空间。

定义命名空间的语法格式如下：

```
namespace [<空间名>]{一组成员}
```

其中，**namespace** 是关键字，后面给出一个空间的名字标识符。后面用花括号括起来一组成员，可以是一组类、一组函数、一组变量。如果空间名缺省则为无名空间。无名空间中的元素类似于全局变量，只是限制在本文件中访问，而全局空间中的成员可以被其它文件访问。

命名空间可嵌套说明，就如同目录与子目录之间的关系。

例如下面代码：

```
int myInt = 98;                //全局空间中的变量
namespace Example{            //说明了一个 Example 空间
    const double PI = 3.14159; //Example 中的变量成员
    const double E = 2.71828;  //Example 中的变量成员
    int myInt = 8;              //Example 中的变量成员，隐藏了全局同名变量
    void printValues();         //Example 中的函数成员的原型
    namespace Inner{           //嵌套空间，名为 Inner
        enum Years{FISCAL1 = 2005, FISCAL2, FISCAL3}; //嵌套空间中的成员
        int h = ::myInt;        //用全局变量进行初始化,98
        int g = myInt;          //用外层成员进行初始化,8
    }
}
namespace{                     //无名空间
    double d = 88.22;          //无名空间中的变量成员
    int myInt = 45;            //无名空间中的变量成员
}
```

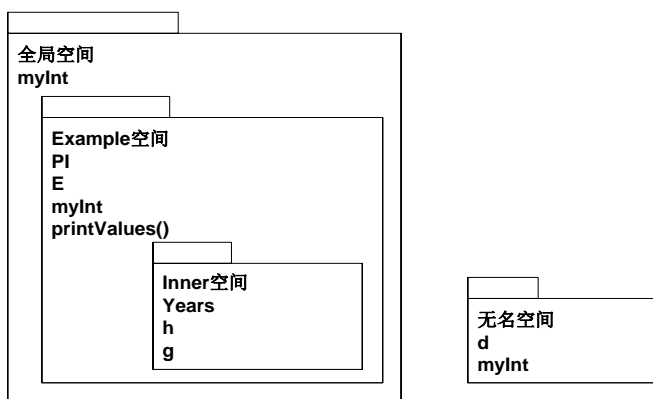


图 16-1 命名空间的结构

图 16.1 表示了上面例子所说明的空间的结构。

在描述成员的全称时，要使用作用域解析运算符“::”。例如，Example 是一个空间，其中各成员的全称为：

```
Example::PI;
Example::E;
Example::myInt;
Example::printValues();
```

Example::Inner 是一个嵌套空间，Example 被称为 Inner 的外层空间，嵌套空间中的成员的全称为：

```
Example::Inner::Years
Example::Inner::h
Example::Inner::g
```

### 16.3.2 空间中成员的访问

如何访问某空间中的成员？这与访问文件系统中的文件相似。访问文件可用相对路径，也可用全路径。全路径就是把全部路径名作为文件名的限定符。

如何查找一个成员？在当前空间中对一个名字 k 的访问需要查找过程，在编译时确定被访问的实体。有以下 3 种形式：

- 1、限定名形式：“空间名::k”。先在当前空间的限定嵌套空间中查找 k，相当于一个相对路径。在嵌套空间中如果未找到成员 k，就将该空间名作为全路径的空间名再查找成员 k，如果仍未找到，就给出错误信息。这种访问形式对相对路径的空间名优先。
- 2、全局限定形式：“::k”。在全局空间中查找，如果未找到就出错。
- 3、无限定形式：“k”。按局部优先原则，先在当前空间中查找。如果未找到，就向外层空间找，直到全局空间。如果未找到，就从导入空间中查找，如果找到一个就确定，如果在全局空间和导入空间中找到多于一个就指出二义性错误。如果仍未找到 k 就是一个错误名字。



如果要多次访问某个空间中的成员，你可能觉得成员名前面总是挂上一串空间名很麻烦。有一个简单方法就是用 `using namespace` 语句来导入(import)一个命名空间。格式如下：

```
using namespace [::] [空间名::] 空间名;
```

其中，`using` 和 `namespace` 是关键字，然后指定一个空间名，或者一个嵌套空间名。例如：

```
using namespace std;
```

就是导入空间名“`std`”，所有标准 C++ 库都定义在命名空间 `std` 中，也包括标准模板库 STL，因此这条语句经常使用。

导入空间语句仅在当前文件中有效。在一个文件中可以有多条导入语句，来导入多个空间名。

实际上，一个无名空间的定义都隐含着以下两条语句：

```
namespace unique{...}
```

```
using namespace unique;
```

其中 `unique` 是系统隐含的一个唯一的空间名。导入命令使得无名空间中的成员仅能在本文件中访问，因此无名空间就是一种缺省导入的空间。

如果只想导入某空间中的某个成员，而不是全部成员，可按下面格式说明：

```
using [::]空间名::成员名;
```

下面例子说明命名空间的定义和访问方法。

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

int myInt = 98;

namespace Example{
    const double PI = 3.14159;
    const double E = 2.71828;
    int myInt = 8;
    void printValues();
    namespace Inner{
        enum Years{FISCAL1 = 2005, FISCAL2, FISCAL3};
        int h = ::myInt;
        int g = myInt;
    }
}

namespace{
    double d = 88.22;
    int myInt = 45;
}

int main(){
    cout<<"In main";
    cout<<"\n(global) myInt = "<<::myInt;           //A
    cout<<"\nd = "<<d;                               //B
    cout<<"\nExample::PI = "<<Example::PI
        <<"\nExample::E = "<<Example::E
        <<"\nExample::myInt = "<<Example::myInt
        <<"\nExample::Inner::FISCAL3 = "<<Example::Inner::FISCAL3
        <<"\nExample::Inner::h = "<<Example::Inner::h
        <<"\nExample::Inner::g = "<<Example::Inner::g<<"\n";
    Example::printValues();
    system("pause");
    return 0;
}
```



```
}

void Example::printValues() {
    cout<<"\nIn Example::printValues:\n"
        <<"myInt = "<<myInt //C
        <<"\nd = "<<d
        <<"\nPI = "<<PI
        <<"\nE = "<<E
        <<"\n::myInt = "<<::myInt //D
        <<"\nInner::FISCAL3 = "<<Inner::FISCAL3 //E
        <<"\nInner::h = "<<Inner::h //F
        <<"\nExample::Inner::g = "<<Example::Inner::g<<endl; //G
}
```

执行程序，输出结果如下：

```
In main
(global) myInt = 98
d = 88.22
Example::PI = 3.14159
Example::E = 2.71828
Example::myInt = 8
Example::Inner::FISCAL3 = 2007
Example::Inner::h = 98
Example::Inner::g = 8

In Example::printValues:
myInt = 8
d = 88.22
PI = 3.14159
E = 2.71828
::myInt = 98
Inner::FISCAL3 = 2007
Inner::h = 98
Example::Inner::g = 8
```

函数 `main` 是全局函数，其中代码要访问某个空间中的成员就要给出绝对路径全称。注意到在全局空间和无名空间中都定义了 `myInt` 成员，因此在全局空间中如果直接用“`myInt`”来访问，就会造成二义性。假如 A 行用“`myInt`”来访问，就会产生二义性。用“`::myInt`”是就访问全局变量。B 行用“`d`”或者“`::d`”来访问无名空间中的成员，此时无二义性。

函数 `printvalues` 是 `Example` 空间中的成员，其中代码可用绝对路径，也可用相对路径来访问成员。C 行用“`myInt`”访问当前空间中的成员，尽管在全局空间和无名空间中都有同名成员，但当前命名空间中的成员具有优先权，故此无二义性。D 行用“`::myInt`”限定全局变量。E 行和 F 行采用了相对路径，而 G 行采用了绝对路径，给出了全称。

对于命名空间的使用，应注意以下要点：

- 尽量使全局空间中的成员最少，这样发生冲突的可能性就会减少。
- 空间的命名应尽可能地表示自然结构，而不仅仅是为了避免命名冲突。
- 在一个源文件中应尽可能避免说明多个平行的空间。
- 二义性往往发生在全局空间和导入空间中含有同名元素，而程序中用无限定的形式来访问该元素。



### 16.3.3 C11 inline 空间

C11 用 `inline` 修饰一个命名空间,该空间中的名称同时作为其外层空间直接包含的成员。被 `inline` 修饰的空间一般是嵌套空间,效果是成为其外层的一个“缺省的”空间。

下面例子说明 `inline` 空间的用法。

```
#include<iostream>
using namespace std;
namespace Mine{
    namespace V98 {
        void f(int a) { cout << "V98" << endl; }
    }
    inline namespace V99 { //A
        void f(int a) { cout << "V99" << endl; }
        void f(double) {}
    }
}
int main() {
    using namespace Mine;
    V98::f(1); // V98
    V99::f(1); // V99
    f(1); // V99 B
    system("pause");
    return 0;
}
```

A 行说明 V99 是 `inline` 空间, B 行用非限定方式调用的函数 f 是来自这个 `line` 空间。

## 16.4 修饰符 `explicit`

我们在第 10 章介绍过,在一个类中,如果说明了单参构造函数,就可用于隐式的类型转换。这种隐式的类型转换可能会被误用,尤其是在函数调用时,实参到形参的隐式转换,可能在不经意之间就创建了对象。

例如,如果一个函数的形参是一个对象或对象引用,而该对象类型恰好有单参构造函数,那么调用方就可利用此形参隐式创建新对象,再传给函数做实参。用 `explicit` 修饰符就能避免这种无意的创建对象。

如果将一个单参构造函数修饰为 `explicit`,该函数就不能用于隐式的类型转换,而只能用于显式地创建对象。

例 16-3 分析下面例子。

```
#include<iostream.h>
#include<assert.h>
class IntArray{ //一个整数数组类
    int size;
    int *ptr;
    friend ostream &operator<<(ostream &, const IntArray &); //运算符<<
public:
    IntArray(int = 10); //A 单参构造函数,同时也是缺省构造函数
    ~IntArray();
};
IntArray::IntArray(int arraySize){ //单参构造函数
```



```
size = (arraySize>0 ? arraySize : 10);
ptr = new int[size];
assert(ptr != 0);           //用断言检查点
for (int i = 0; i < size; i++) //置缺省值
    ptr[i] = 0;
}
IntArray::~IntArray() {delete[] ptr;} //析构函数
ostream &operator<<(ostream &output, const IntArray &a){ //运算符<<重载函数
    for (int i = 0; i < a.size; i++) //输出各元素
        output<<a.ptr[i]<<" ";
    return output;
}
void outputArray(const IntArray &a){ //B 全局函数, 调用<<输出各元素
    cout<<"The array:\n"<<a<<"\n";
}
void main(){
    IntArray integers1(7); //说明一个整数数组
    outputArray(integers1); //调用全局函数, 输出个元素
    outputArray(15);       //C 隐式创建对象
}
```

A 行说明了一个单参构造函数, 形参为一个 `int`, 这意味着可以从一个 `int` 值来创建一个 `IntArray` 对象。例如: `IntArray a1= 15;` 在需要一个 `IntArray` 对象的地方, 提供一个 `int` 值就能自动创建一个 `IntArray` 对象, 再提供给它。B 行函数形参恰好就是 `IntArray` 对象, 那么 C 行调用函数时提供了一个 `int` 值 15, 实际上转换为

```
outputArray(IntArray(15));
```

所以语法上没有错误。但实际上可能不是你想要的。你只是无意中犯了一个错误, 所以希望 C 行能给出编译错误, 而不想自动创建一个 `IntArray` 对象。

此时修饰符 `explicit` 就有用了。只需对类中构造函数原型添加 `explicit` 修饰:

```
explicit IntArray(int = 10);
```

就可避免这种隐式的创建对象, 此时 C 行代码编译报错。

对于一些“比较大”的类, 比如 STL 容器类(如 `vector`、`deque`、`list` 等), 它们的单参构造函数都用 `explicit` 修饰, 目的就是避免误建对象。

C11 开始 `explicit` 从转换构造函数扩展到类型转换函数(12.3.2 节介绍)。

## 16.5 修饰符 `mutable`

我们在第 10 章介绍过, 用 `const` 修饰的成员函数不能改变对象的状态, 即不能改变非静态数据成员的值。但有个例外, 用 `mutable` 修饰的非静态非 `const` 数据成员可以被 `const` 成员函数改变。

例 16-4 分析下面例子。

```
class TestMutable{
    mutable int value;           //A 用 mutable 修饰的数据成员
public:
    TestMutable(int v = 0){value = v;}
    void modifyValue() const {value++;} //B const 函数中可改变 mutable 成员
    int getValue() const {return value;}
```



```
};  
void main() {  
    const TestMutable t(99);  
    cout<<"Initial value is "<<t.getValue();  
    t.modifyValue();  
    cout<<"\nModified value is "<<t.getValue()<<endl;  
}
```

A 行用 `mutable` 说明了一个数据成员。B 行是一个 `const` 函数，但能改变这个数据成员。修饰符 `mutable` 的用途就是说明一个数据成员是可变的，在任何成员函数中都可改变。

注意，`mutable` 与修饰符 `volatile` 的区别。`volatile` 修饰符说明一个数据成员是易变的，随时可能被程序外部其它东西所改变，如操作系统、硬件、并发线程等。它要求编译器对它的访问不能进行优化。`volatile` 对于一般的编程没有直接影响。而 `mutable` 对于编程是有影响的，`const` 成员函数中不能改变数据成员的值，但除了 `mutable` 修饰的成员。

## 16.6 RTTI 与 typeid

RTTI 是 Run-Time Type Information 运行期类型信息的简写。在运行时刻我们常常要利用 RTTI 动态掌握有关类型的信息，尤其是对于抽象类编程和模板编程。在抽象层面上可根据具体类型做出不同处理，更方便通用性编程。

`typeid` 是一个关键字，就像 `sizeof` 一样，以函数的形式，在运行时刻获得指定对象或表达式的类型信息。`typeid` 有下面两种形式：

```
const type_info& typeid( type-id )      //求类型 type-id 的具体类型  
const type_info& typeid( expression )   //求表达式的具体类型
```

`typeid` 返回类型 `type_info` 是一个对象类，往往定义在 `typeinfo.h` 文件中：

```
class type_info {  
public:  
    virtual ~type_info();  
    int operator==(const type_info& rhs) const;  
    int operator!=(const type_info& rhs) const;  
    int before(const type_info& rhs) const;  
    const char* name() const;                //读取类型的名字  
    const char* raw_name() const;  
};
```

前面编程中我们用 “`typeid(<表达式>).name()`” 来获取<表达式>在运行时刻的类型名称。对于基本类型就直用其名，如 “`int`”、“`double`”。对于类名，要加前缀 “`class`”。

例 16-5 运行期类型信息的例子。

```
#include<typeinfo>  
#include<iostream>  
#include<string>  
using namespace std;  
  
template<class T>
```





```
T maxmum(T value1, T value2, T value3){
    T max = value1;
    if (value2 > max)
        max = value2;
    if (value3 > max)
        max = value3;
    const char * dataType = typeid(T).name();           //A
    cout<<dataType<<"s are compared."
        <<"\nLargest " <<dataType<<" is ";
    return max;
}
class A{
public:
    void f(){
        cout<<"f1 this is " <<typeid(this).name() <<endl;    //B
    }
    void f()const{
        cout<<"f2 this is " <<typeid(this).name() <<endl;    //C
    }
};

int main(){
    int a = 2, b = 3, c = 1;
    double d = 4.3, e = 7.7, f = 2.3;
    string s1 = "one", s2 = "two", s3 = "three";
    cout<<maxmum(a, b, c) <<endl;
    cout<<maxmum(d, e, f) <<endl;
    cout<<maxmum(s1, s2, s3) <<endl;
    A a1;
    a1.f();
    const A a2;
    a2.f();
    const char * str1 = "const string";                 //D
    cout<<typeid(str1).name() <<endl;
    char const * str2 = "const string";                 //E
    cout<<typeid(str2).name() <<endl;
    cout<<typeid("const").name() <<endl;                //F
    cout<<"type of a > b is " <<typeid(a > b).name() <<endl; //G
    system("pause");
    return 0;
}
```

执行程序，输出结果如下：

```
ints are compared.
Largest int is 3
doubles are compared.
Largest double is 7.7
class std::basic_string<char,struct std::char_traits<char>,class
std::allocator<char> >s are compared.
Largest class std::basic_string<char,struct std::char_traits<char>,class
std::allocator<char> > is two
f1 this is class A *
f2 this is class A const *
char const *
char const *
char [6]
type of a > b is bool
```



A 行读取模板形参 T 的运行时刻具体类型的名称，下面再显示出来。A 行中也可对某个形参变量来处理，如 `...=typeid(value1).name()`，得到一样结果。

可看到 `string` 的运行时刻的具体类型，“`class std::basic_string<char,struct std::char_traits<char>,class std::allocator<char>>`”，这些信息对于通用性模板的编程分析具有重要作用。

B 行和 C 行分别显示在非 `const` 成员函数和 `const` 成员函数中的 `this` 的类型。可以看到，在非 `const` 成员函数中的 `this` 类型为 `class A *`，而在 `const` 成员函数中则为 `class A const *`。实际上，我们习惯将后者表示为 `class const A *`，将 `const` 放在类型名之前，与放在类型名与 `*` 之间是一样的。因此 D 行与 E 行描述的两个字符串类型是一样的。

对于字符串字面值的类型，也可以看到，F 行显示 `char [6]`。

关系表达式的 `typeid` 类型是 `bool` 型，而不是 `int`，G 行显示可以验证。

## 16.7 强制类型转换

强制类型转换 `cast`，也称为显式类型转换，可作用于基本类型的变量或表达式，也可作用于用户定义类型，将一个基类的指针或引用强制转换为其派生类的指针或引用。强制转换一般称为“向下转换”或者“窄化”。传统的类型转换形式“(类型名)变量名/表达式”，存在的问题是过于笼统、功能过强而易失控、不够安全。

C++ 标准中引入了 4 个新的强制类型转换运算符，以取代传统的强制类型转换，好处是更具体、功能弱化、相对安全。表 16.1 给出了这 4 种新型类型转换的特点。

表 16-1 强制类型转换

类型转换	语法 (注意尖括号不能缺少)	特点
静态转换	<code>static_cast&lt;目标类型名&gt;(变量名/表达式)</code>	编译时检查合法性。比较安全
动态转换	<code>dynamic_cast&lt;目标类型名&gt;(变量名/表达式)</code>	运行时刻检查合法性，适用于指针转换。转换之后应判断指针是否为空，若为空则转换失败，若非空则转换成功。比较安全。
常量转换	<code>const_cast&lt;目标类型名&gt;(变量名)</code>	专门对 <code>const</code> 或 <code>volatile</code> 修饰的变量进行转换。会破坏既有的 <code>const</code> 约定，建议慎用。
重释转换	<code>reinterpret_cast&lt;目标类型名&gt;(变量名/表达式)</code>	专门处理指针转换。指针之间的转换，也可将指针随意转换到其它类型，或将其它类型转换到指针。技巧性和危险性并存。

下面介绍这 4 种运算符。



### 16.7.1 static\_cast 运算符

静态转换在编译时刻进行类型检查。对于非法的转换将给出错误信息。例如，将 `const` 类型转换为非 `const` 类型、把基类转换到非公有继承的派生类、从一个类型转换到不具有特定构造函数或转换函数的类型等。对于基本类型，采用静态转换替代传统转换是比较安全的。

语法格式如下：

`static_cast<目标类型名>(变量名/表达式)`

例 16-6 静态转换的例子。

```
#include<iostream>
using namespace std;
class BaseClass{
public:
    void f()const{cout<<"BASE\n";}          //A 非虚函数
};
class DerivedClass: public BaseClass{
public:
    void f()const{cout<<"Derived\n";}        //B
};
void test(BaseClass * basePtr){
    DerivedClass *derivedPtr;
    derivedPtr = static_cast<DerivedClass *>(basePtr);    //C
    derivedPtr = (DerivedClass *)basePtr;                //D
    derivedPtr->f();
}
int main(){
    double d = 8.22;
    int x = d;                                           //E warning: possible loss of data
    int y = (int)d;                                     //F
    int z = static_cast<int>(d);                         //G double -> int
    double d2 = y;                                       //H
    cout<<"x is "<<x<<endl;
    cout<<"y is "<<y<<endl;
    cout<<"z is "<<z<<endl;
    cout<<"d2 is "<<d2<<endl;
    char * str1 = "C++";
    void * vp = str1;                                   //I
    char * str2 = (char *) (vp);                        //J
    char * str3 = static_cast<char *>(vp);              //K
    if (str2 == str3)
        cout<<"str2 == str3"<<endl;
    cout<<str2<<endl;
    cout<<str3<<endl;
    BaseClass base;
    test(&base);
    system("pause");
    return 0;
}
```

执行程序，输出结果如下：

```
x is 8
y is 8
z is 8
d2 is 8
str2 == str3
C++
```

```
C++
Derived
```

A 行定义的函数是非虚函数，B 行派生类定义的同名同参函数并非改写 `override`。

C 行使用了静态转换，效果与 D 行的传统转换一样。

E 行将一个 double 赋给一个 int，导致一个警告。

F 行是传统转换，与 G 行的静态转换效果一样。

H 行将一个 `int` 赋给一个 `double`，不会导致警告。

I 行将一个 `char*` 赋给一个 `void*`，不会导致警告。

J 行使用传统转换，与 K 行的静态转换效果一样。

可以看出，多数传统的类型转换都可用静态转换来实现，只是静态转换的类型检查更严格。如果将公有继承改变为缺省的私有继承，静态转换的 C 行将报错，而传统转换的 D 行却不报错。

### 16.7.2 dynamic\_cast 运算符

这种转换被称为“动态转换”。在运行时刻进行类型检查，一般作用于类的指针或引用，也包括 void 指针。对于指针进行动态转换之后，应立即判断新的指针值是否为 0，如果为 0，表示转换失败，如果非 0，表示转换成功，然后才能通过该指针进行操作。

语法格式如下：

dynamic cast<目标类型名>(变量名/表达式)

例 16-7 动态转换的例子。

[illegible]



```
    return 0;
}
```

执行程序，输出结果如下：

```
class D object is provided
Driven
Driven
class B object is provided
Base
pd2 == 0
```

A 行在基类中说明一个虚函数，使 A 类成为多态性基类。派生类 D 在 B 行改写了这个虚函数。C 行用 typeid 获得对象的实际类型，再用 name 函数读取名字并显示出来。

D 行用静态转换将形参 B\* pb 转换为派生类指针 D\*pd1。这对于 F 行调用是安全的，因为实参就是派生类的对象。但对于 G 行调用就出错了，因为实参是基类对象，而用派生类指针来操作，虽然能调用虚函数，但仍执行基类的函数，而不是派生类的函数。

E 行用动态转换形参 B\*pb 转换为派生类指针 D\*pd2。如果转换成功，指针为非 0；如果失败，指针为 0。对于 0 指针不能进行任何操作。因此对于类的指针的转换，使用动态转换是比较安全的。

### 16.7.3 const\_cast 运算符

这种转换被称为“常量转换”，专门对 const 或 volatile 修饰的变量进行转换。

语法格式如下：

const\_cast<目标类型名>(变量名)

例 16-8 常量转换的例子。

```
#include<iostream>
using namespace std;
class ConstCastClass{
    int number;
public:
    void setNumber(int num){number = num;}
    int getNumber()const{return number;}
    void printNumber() const{                               //A
        ConstCastClass* newThis;
        newThis = const_cast<ConstCastClass* const>(this); //B
        newThis = (ConstCastClass* const)this;             //C
        newThis->number--;                                   //D
        cout<<"\nNumber after modification:"<<number<<endl;
    }
};
int main(){
    ConstCastClass x;
    x.setNumber(8);
    cout<<"Initial value of number is "<<x.getNumber();
    x.printNumber();
    system("pause");
    return 0;
}
```

执行程序，输出结果如下：



```
Initial value of number is 8  
Number after modification:7
```

A 行定义成员函数为 `const` 函数，那么该函数中就不能改变数据成员 `number` 的值。这是利用 `this` 指针的 `const` 性质来实现的。

在非 `const` 成员函数中 `this` 指针的类型为 “`class 类名*const`”，不能改变 `this` 的值使其指向另一个对象，但能改变当前对象的值。

在 `const` 成员函数中 `this` 指针的类型为 “`const class 类名*const`”，不能改变 `this` 的值，也不能通过 `this` 指针来改变数据成员的值，也不能通过 `this` 指针调用非 `const` 函数。

B 行用常量转换将 `this` 的类型由 “`const ConstCastClass *const`” 转换为 “`ConstCastClass*const`”，这样通过该指针就能改变数据成员 `number` 的值。这个效果与 C 行传统转换一样。最终结果是在一个 `const` 函数中改变了一个数据成员的值。使用 `mutable` 来修饰一个数据成员也能达到同样目的，但常量转换后能改变所有的数据成员，而不限一个。

可以看出，使用常量转换会破坏函数的约定，建议慎用。

#### 16.7.4 reinterpret\_cast 运算符

这种转换被称为“重释转换”，只能对指针进行转换或者转换到指针，最具技巧性，也最危险。语法格式如下：

`reinterpret_cast<目标类型名>(变量名/表达式)`

例 16-9 分析下面例子。

```
#include<iostream>  
using namespace std;  
int main(){  
    unsigned x = 22, *unsignedPtr;  
    void * voidPtr = &x; //A  
    char * charPtr = "C++";  
    unsignedPtr = reinterpret_cast<unsigned*>(voidPtr); //B  
    cout<<"*unsignedPtr is "<<*unsignedPtr  
        <<"\ncharPtr is "<<charPtr;  
    cout<<"\nchar * to unsigned result in:"  
        <<(x = reinterpret_cast<unsigned>(charPtr)); //C  
    cout<<"\nunsigned to char * result in:"  
        <<reinterpret_cast<char*>(x)<<endl; //D  
    int i = 9;  
    double *d = reinterpret_cast<double*>(&i); //E  
    cout<<*d<<endl;  
    system("pause");  
    return 0;  
}
```

执行程序，输出结果如下：

```
*unsignedPtr is 22  
charPtr is C++  
char * to unsigned result in:15371056  
unsigned to char * result in:C++  
-9.25596e+61
```

A 行使一个 `void*` 指针指向一个 `unsigned` 变量，B 行用重释转换将这个 `void*` 指针转换为



`unsigned*`指针, 使 `unsignedPtr` 指针指向 `x`。然后用 `cout` 语句通过该指针访问 `x` 的值, 没有错误。B 行业没有错误。

C 行用重释转换将一个 `char*`指针转换为一个 `unsigned` 值, 赋给 `x` 并输出, 可以看到该指针的值, 就是字符串 "C++" 的存储地址。此时你可以随便改变 `x` 的值。D 又用重释转换将 `unsigned` 变量 `x` 转换为一个 `char*`, 并输出。

E 行用重释转换将一个 `int*`转换为一个 `double*`, 而得到与原值不相干的结果。

可以看出, 重释转换可以将指针随意转换到其它类型, 或将其它类型转换到指针, 最具危险性, 建议读者慎用。

## 16.8 小 结

- 命名空间(namespace)是解决大程序中多个元素(如类、全局函数和全局变量等)命名冲突问题的一种机制。
- 一个程序所用的多个命名空间在相同层次上不重名; 在同一个命名空间中的所有成员名字不重复; 一个命名空间可以嵌套定义其内层的多个不重名的空间。
- 使用 `namespace` 关键字来定义命名空间, 及其成员。嵌套空间也如此定义。
- 用作用域运算符 “`::`” 来分隔空间名以及成员名。
- 使用 `using namespace` 来导入空间, 以方便访问特定空间中的成员。
- 当用一个名字 `k` 来访问成员时, 按局部优先原则, 先在当前空间中查找。如果未找到, 就向外层空间找, 直到全局空间。如果未找到, 就从导入空间中查找。如果在全局空间和导入空间中找到多于一个就是二义性错误。
- 修饰符 `explicit` 用来限制单参构造函数, 避免隐式地创建对象, 对于大对象类有用。
- 修饰符 `mutable` 使类的数据成员可改变, 而不管是否在 `const` 成员函数之中。`const` 成员函数中不能改变数据成员的值, 但除了 `mutable` 修饰的成员。
- 运行期类型信息 RTTI, Run-Time Type Information 在运行时刻掌握类型信息对于通用性编程, 对于复杂程序的分析都有用。`typeid` 是关键词, 需要包含 `typeinfo.h` 文件, 常用 “`typeid(<表达式>).name()`” 来获取<表达式>在运行时刻的类型名称。
- 传统的强制类型转换存在的问题是过于笼统、功能过强而易失控、不够安全。标准 C++ 引入了 4 个新的强制类型转换运算符, 以取代传统的强制类型转换, 好处是更具体、功能弱化、相对安全。



# 附录 A    ASCII 码表

ASCII 值	控制字符	ASCII 值	字符	ASCII 值	字符	ASCII 值	字符
0	NUL	32	(space)	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	“	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	\$	68	D	100	d
5	END	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	‘	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	(del)





## ASCII 控制字符

ASCII 值	简写	全称	含义及显示	转义符	输入法
0	NUL	Null char	空字符	\0	
1	SOH	Start of Header	标题起始, 显示为☺		Ctrl+A
2	STX	Start of Text	文本起始, 显示为☹		Ctrl+B
3	ETX	End of Text	文本结束, 显示为♥		Ctrl+C
4	EOT	End of Transmission	传输结束, 显示为♦		Ctrl+D
5	ENQ	Enquiry	询问, 显示为♣		Ctrl+E
6	ACK	Acknowledgement	应答, 显示为♠		Ctrl+F
7	BEL	Bell	响铃	\a	Ctrl+G
8	BS	Backspace	退格, 光标回退一个位置	\b	Ctrl+H
9	HT	Horizontal Tab	水平制表, 光标移到下一个制表位(以 8 个字符为单位)	\t	Ctrl+I
10	LF	Line Feed	换行, 光标移到下一行	\n	Ctrl+J
11	VT	Vertical Tab	垂直制表, 显示为♂	\v	Ctrl+K
12	FF	Form feed	换页, 显示为♀	\f	Ctrl+L
13	CR	Carriage return	回车, 光标移到行头位置	\r	Ctrl+M
14	SO	Shift out	移出, 显示为♪		Ctrl+N
15	SI	Shift in	移入, 显示为♫		Ctrl+O
16	DLE	Data Link Escape	数据链丢失, 显示为▶		Ctrl+P
17	DC1	Device control 1	设备控制 1, 显示为◀		Ctrl+Q
18	DC2	Device control 2	设备控制 2, 显示为↑		Ctrl+R
19	DC3	Device control 3	设备控制 3		Ctrl+S
20	DC4	Device control 4	设备控制 4		Ctrl+T
21	NAK	Negative Acknowledgement	否定应答		Ctrl+U
22	SYN	Synchronous Idle	同步闲置符		Ctrl+V
23	ETB	End of Trans. Block	传输块结束		Ctrl+W
24	CAN	Cancel	取消, 显示为↑		Ctrl+X
25	EM	End of Medium	媒介结束, 显示为↓		Ctrl+Y
26	SUB	Substitute	替换, 显示为→		Ctrl+Z
27	ESC	Escape	退出, Esc 键, 显示为←		
28	FS	File Separator	文件分隔符		
29	GS	Group separator	组分分隔符, 显示为↔		
30	RS	Record separator	记录分隔符, 显示为▲		
31	US	Unit Separator	单元分隔符, 显示为▼		



## 附录 B 常用库函数

本文主要涉及两类库函数，运行期库(run-time library)和 C++标准库。下面简单介绍这些库的使用要点。运行期库是用 C 语言实现的基础程序库，其它库都以此为基础。运行期库按功能划分，如表 B-1 所示。

表 B-1 运行期库的功能分类

分类	功能	相关头文件(不完全)
可变参数	用于定义可变参数的函数	<stdarg.h>
缓冲区管理	按字节管理内存缓冲区	<string.h><memory.h>
按字节分类	多字节字符分类，与当前多字节代码页相关	<ctype.h>
按字符分类	对单字节字符、宽字符、多字节字符进行分类。比较常用，如 isalpha, isprint。	<ctype.h>
数据对齐	按对齐边界分配内存、回收内存	<malloc.h>
数据转换	一种数据转换到另一种，例如字符串到 int 或 double，或反之。有很多转换既有函数实现，也有宏实现，可选择。	<math.h><stdlib.h>
调试程序	debug 调试，函数库中有专门的调试版本，支持单步执行、断言、错误检测、异常，跟踪堆空间分配，避免内存泄露，以及调试信息报告等。	<assert.h><crtdbg.h>
目录控制	读取或改变目录，创建、删除目录等，也包括使用环境路径来搜索文件	<stdlib.h><direct.h>
错误处理	包括断言、检测 IO 错误、清除错误标记、判断低级 IO 的文件尾 eof 等。	<assert.h><crtdbg.h><stdio.h> <io.h>
异常处理程序	程序终止处理(terminate)、意外处理(unexpected)	<eh.h>
文件处理	对磁盘文件的建立、删除、改名、文件访问授权等操作。	<stdio.h><io.h><sys/locking.h> <errno.h>
浮点数支持	专门针对浮点数的计算，如指数、对数、三角函数、双曲函数等，也包括错误检测，如溢出。	<math.h> <stdlib.h><float.h>
输入输出	从文件或设备中读入数据或写出数据。文件 IO 要区分文本模式和二进制模式。IO 分为以下三类： 1、流式 IO，将数据作为字符或字节序列，有缓冲。 2、低级 IO，直接调用操作系统，无缓冲。	<stdio.h><io.h><conio.h>



	3、控制台与端口 IO，对键盘和字符显示器的直接读写，对 IO 设备，如打印机、串行口的直接读写。	
国际化	适应不同语言与地域 locale 相关程序、宽字符、多字节字符、通用文本等等。	<locale.h><wchar.h><stdio.h> <string.h><ctype.h><mbstring.h>
内存分配	动态分配、回收内存，如 malloc、free 等函数。	<stdlib.h> <malloc.h><new.h>
进程与环境控制	进程的启动、停止与管理，也包括线程的启停。操作系统环境信息的读取与改变。	<process.h> <stdlib.h>
鲁棒性	Win32 异常处理，终止函数等	<eh.h>
运行期错误检查	run-time error checks (RTC).错误报告	<rtcap.h>
排序与查找	对任意类型数组进行排序，折半查找与线性查找。	<stdlib.h> <search.h>
字符串管理	对窄串、宽串、多字节字符串进行操作。也包括缓冲管理。数量庞大的一组函数。	<string.h><memory.h><wchar.h> <mbstring.h>
系统调用	用来查找文件的 3 个函数	<io.h>
时间管理	获取当前系统日期时间、转换、调整等操作	<time.h><sys/timeb.h>

注 1 以上列出 22 个功能分组。

注 2 同一个头文件可能出现在多个功能分组中，同一个函数也可能出现在不同的头文件中。

注 3 运行期库是纯 C 语言实现，不包含 C++内容(没有重载、形参缺省值、引用、模板等)。

表 B-2 运行期库头文件

头文件名	功能	C++包装头文件名
<assert.h>	断言设置	<cassert>
<ctype.h>	字符分类	<cctype>
<errno.h>	由库函数执行，检测错误代码	<cerrno>
<fenv.h>	浮点环境控制及异常相关的函数和宏	<cfenv>
<float.h>	浮点数计算	<cfloat>
<inttypes.h>	基于宽度的整数类型	<cinttypes>
<iso646.h>	ISO 646 字符集处理	<ciso646>
<limits.h>	检测整数类型的性质	<climits>
<locale.h>	不同地域文字适应性	<clocale>
<math.h>	公共数学计算	<cmath>
<setjmp.h>	执行非本地 goto 语句	<csetjmp>
<signal.h>	控制各种异常条件	<csignal>
<stdarg.h>	可变参数的函数	<cstdarg>
<stdbool.h>	标准逻辑类型	<cstdbool>
<stddef.h>	多种有用的类型 typedef 和宏的定义	<cstddef>
<stdint.h>	标准整数类型	<cstdint>
<stdio.h>	输入和输出	<cstdio>



附录 B

<stdlib.h>	多种操作函数	<cstdlib>
<string.h>	多种字符串的处理	<cstring>
<tgmath.h>	包含<ccomplex>和<cmath>	<ctgmath>
<time.h>	系统时间处理	<ctime>
<wchar.h>	宽字符流，以及特殊字符串处理	<cwchar>
<wctype.h>	宽字符分类	<cwctype>

注 1，表中列出的 23 个头文件是作为 C++标准库，而运行期库的头文件还有许多未列入。

注 2，C++标准库的头文件大多不含.h 后缀。

注 3，左边头文件内容被包装到 C++标准的命名空间 std 中。例如，<cassert>文件大致如下：

```
namespace std {#include <assert.h> };
```

表 B-3 标准 C++库头文件

头文件名	功能
<algorithm>	算法，提供了 80 多个模板函数，通过迭代器作用于各种容器，实现排序、查找、集合运算等算法。
<numeric>	提供若干模板函数，用于数值计算，如求和、求乘积、求部分和等。
<vector><deque> <list><forward_list><array><valarray>	序列容器，其中<forward_list>是单向链表，<array>是不变长的数组。<valarray>是可变长的数组
<map><set>	有序关联容器
<unordered_map><unordered_set>	无序关联容器
<queue><stack>	适配器容器，队列与堆栈
<iterator>	迭代器
<functional>	模板类，函数对象，供各种容器和算法使用
<exception><stdexcept> <system_error>	异常处理，错误处理
<iostream> <fstream> <sstream><filesystem> <iomanip><ios><iosfwd> <ostream><istream> <streambuf>	输入输出流
<strstream>	支持对字符数组对象的 iostream 操作，支持与 C 字符串的转换
<codecvt><cvt/wbuffer> <cvt/wstring>	本地化
<locale>	提供一组模板类和函数，封装和管理地域 locale 信息，以支持多国文字习惯用法。
<bitset>	位集，一个模板类和两个支持模板函数。



<complex>	复数，一个模板类和若干模板函数
<numeric><random> <ratio>	数学与数值运算
<limits>	定义了模板类 <code>numeric_limits</code> ，其中规范了算术计算中各种类型的值范围
<allocators><memory><new> <scoped_allocator>	内存管理
<atomic><thread><mutex> <condition_variable><future>	多线程支持
<chrono>	一种新的时间库，不在 <code>std</code> 命名空间中，在 <code>std::chrono</code> 空间之中。能计算时间间隔。
<initializer_list>	访问初始化列表中的值。
<tuple>	元组，一个 <code>tuple</code> 对象包含 2 个以上元素，各元素可能有不同类型，典型的有二元组，三元组等。
<type_traits>	一组类，用于编译器获取类型信息
<typeinfo>	RTTI 与 <code>typeid</code> 运算符
<typeindex>	<code>type_index</code> 类，以及该类的 Hash 规范
<utility>	定义了对偶 <code>pair</code> ，作为映射和多射的基本元素
<regex>	正则表达式
<string>	定义了 <code>basic_string</code> 模板类，一种字符的容器，其中用 <code>typedef</code> 定义了 <code>string</code> 类型。

注 1，上表列出的头文件，再加上前面 23 个包装头文件，共同组成 C++ 标准库。

注 2，还有几个头文件未列入，`<hash_map>`、`<hash_set>`，它们没有被完整实现或者未成为标准。

下面是一些常用的运行期库。

表 B-5 数学函数<math.h>

函数原型	功 能	返回值	说 明
<code>int abs(int x)</code> <code>long labs(long x)</code> <code>double fabs(double x)</code>	求绝对值	绝对值	
<code>double pow(double x, double y)</code>	求 x 的 y 次方	计算结果	
<code>double sqrt(double x)</code>	求 x 的平方根	计算结果	
<code>double fmod(double x, double y)</code>	求 x 除以 y 的余数	余数	使 $x=i*y+f$ ，f 是返回值，i 是整数，且 f 与 x 相同符号
<code>double ceil(double x)</code>	大于等于 x 的最小整数		如 <code>ceil(2.8) == 3</code>
<code>double floor(double x)</code>	小于等于 x 的最大整数		如 <code>floor(2.8) == 2</code>



附录 B

double modf(double x, double *y)	取 x 的整数部分送到 y 所指向的单元中	x 的小数部分	将浮点数 x 分解为整数部分和小数部分, 如-2.3 分解为-2 和-0.3
double exp(double x)	e 的 x 次方		
double log(double x)	自然对数 ln(x), 以 e 为底的对数		x>0
double log10(double x)	以 10 为底的对数		x>0
三角函数			
double sin(double x) double sinh(double x)	正弦 sin(x) 双曲正弦 sinh(x)	计算结果	x 为弧度值
double cos(double x) double cosh(double x)	余弦 cos(x) 双曲余弦 cosh(x)	计算结果	x 为弧度值
double tan(double x) double tanh(double x)	正切 tan(x) 双曲正切	计算结果	x 为弧度值
double asin(double x)	反正弦 arcsin(x)		-1≤x≤1
double acos(double x)	反余弦 arccos(x)	计算结果	-1≤x≤1
double atan(double x)	反正切 arctan(x)	计算结果	

表 B-6 C 标准库<stdlib.h>

函数原型	功 能	返回值	说 明
void srand(unsigned int seed)	设置伪随机数序列的起点, 即随机数生成种子		先设置种子, 再调用 rand 生成随机数
int rand(void)	生成一个伪随机整数	随机正整数, >0	
void abort(void)	终止进程, 而没有刷新缓冲区, 也没有执行清理		不到万不得已, 不要调用
void exit(int status)	先执行清理, 刷新缓冲区, 关闭打开的文件, 最后终止进程		返回 0 表示正常, 其它值表示错误。返回值可被操作系统的批处理命令获得。
int system(const char *command)	执行 command 串的操作系统命令	返回值就是指命令执行所返回的值, 0 表示正常	启动命令后等待返回。
void qsort( void *base, size_t num, size_t width, int (__cdecl *compare )(const void *elem1, const void *elem2 ) );	对任意类型的数组进行快速排序(冒泡排序的改进)。base 是数组名, num 是元素个数, width 是元素的字节大小, 最		比较函数返回 0, 表示两元素相等。升序排序要求: 返回值小于 0, 表示 elem1 小于 elem2; 返回大于 0, 表示 elem1 大于



	后形参是比较函数指针		elem2。降序相反。
<code>void *bsearch(const void *key, const void *base, size_t num, size_t width, int ( __cdecl *compare ) ( const void *elem1, const void *elem2 ) );</code>	折半查找，base 数组元素要按升序排序，元素个数为 num，元素大小为 width 字节，查找 key，最后一个形参是比较函数的指针。	如果未找到，就返回 NULL。如果找到就返回指针指向数组中的 key	如果数组未按升序排序，或者元素有重复，那么结果不可预测。比较函数与 qsort 要求相同。
动态内存管理			
<code>void *malloc(size_t size);</code>	动态请求分配 size 字节的内存，但可能得到更大空间，因为内存分块管理。	如果内存不够，就返回 NULL。否则返回指针指向所分配的内存	用 free 函数来回收内存。C 基础函数，许多其它函数要调用该函数。
<code>void *calloc(size_t num, size_t size);</code>	动态请求分配一个数组，而且初始化为 0。num 个元素，每个元素大小为 size 个字节	同上	
<code>void *realloc(void *mblock, size_t size);</code>	对已分配的空间重新分配，可改变大小。如果第一个形参为 NULL，就等同于 malloc 函数	同上	
<code>void free(void *mblock);</code>	动态回收所分配的内存，实参指针一定是用 malloc、calloc 或 realloc 得到的		如果实参指针错误，可能导致不可预料的错误
数据转换			
<code>int tolower(int c);</code>	将字符 c 转换为小写，如果可能的话	小写字符	
<code>int toupper(int c);</code>	将字符 c 转换为大写，如果可能的话	大写字符	
<code>int atoi(const char * string)</code>	字符串转换为整数	整数	
<code>double atof(const char * string)</code>	字符串转换为 double	double 值	
<code>double strtod(const char *string, char **endptr);</code>	字符串转换到 double，而且得到停止指针	double 值	第 2 个形参得到字符串中停止扫描的字符指针。 strtoul 处理 unsigned long
<code>char *_itoa(int value, char *string,</code>	将 int 型 value 按基数	返回结果串	基数范围 2-36。



## 附录 B

int radix);	radix 转换到字符串 string		_ltoa 函数针对 long 型。 _ultoa 针对 unsigned long 型
char *_gcvt(double value, int digits, char *buffer);	将 double 型 value 转换 到字符缓冲区 buffer 中	返回结果串	第 2 个形参确定有效位 数
char *_ecvt(double value, int count, int *dec, int *sign);	将 double 型 value 转换 到字符串, 第 2 个形参 确定总的有效位数	返回结果串, 串中无小数点	第 3 个形参得到小数点 位置(0 和负值表示小数 点在数字左边), 第 4 个 形参得到符号位(0 为 正, 1 为负)
char *_fcvt(double value, int count, int *dec, int *sign);	将 double 型 value 转换 到字符串, 第 2 个形参 确定小数点后的有效位 数	返回结果串, 串中无小数点	第 3 个形参得到小数点 位置(0 和负值表示小数 点在数字左边), 第 4 个 形参得到符号位(0 为 正, 1 为负)

表 B-7 字符串函数<string.h>

size\_t 是用 typedef 定义的 unsigned int 的同义词。NULL 是值为 0 的宏, 每个串 char\* 都以 NULL 结尾。形参中所有 const 修饰的串都不可改变, 反之, 无 const 修饰的串都可改变, 而且作为结果。注意, 用 NULL 作为实参调用下面函数将导致运行错误。

函数原型	功 能	返回值	说 明
int strlen(const char *string)	求字符串的长度	字符串包含的字符个数	
char * strcpy(char *s1, const char *s2)	将 s2 串拷贝到 s1 中	目的存储区的始址 s1	
char *strncpy(char *p1, const char *p2, size_t count );	将 s2 串拷贝到 p1 中, 只 拷贝 count 个字符	同上	
int strcmp(const char *s1, const char *s2)	比较两个字符串 s1 和 s2	如两串相同, 就返回 0。 s1 串小于 s2 串, 返回负 数, 否则返回正数	从前向后逐个 字符比较
int strncmp(const char *s1, const char *s2, size_t count );	比较两个字符串 s1 和 s2, 只比较前 count 个字符	同上	
int _strcmpi(const char *s1, const char *s2);	比较两个字符串 s1 和 s2, 而且忽略大小写	同上	
int _strnicmp(const char *s1, const char *s2, size_t count );	比较两个字符串 s1 和 s2, 只比较前 count 个字符, 而且忽略大小写	同上	
char * strcat(char *s1, const char	将 s2 串拼接到 s1 串的后	目的存储区的始址 s1	





* s2)	面		
char *strncat(char *s1, const char *s2, size_t count );	将 s2 串拼接到 s1 串的后 面，只拼接前 count 个字 符	同上	
char *_strrev( char *string );	转换为逆向串	返回结果串	
char *_strlwr( char *string);	转换为小写串	返回结果串	
char *_strupr( char *string);	转换为大写串	返回结果串	
char *strchr(const char *s, int c);	在 s 串中查找字符 c 的首 次出现位置	如找到，返回指针指向该 字符位置。如未找到，返 回 NULL	
char *strrchr(const char *s, int c);	在 s 串中查找字符 c 的最 后出现位置	如找到，返回指针指向该 字符位置。如未找到，返 回 NULL	
char * strstr(const char *s1, const char *s2)	查找子串，在 s1 串中从 前向后查找 s2 串首次出 现位置	如找到，就返回 s1 中 s2 出现的位置，否则就返回 NULL	s2 作为一个串
char *strpbrk(const char *s1, const char *s2);	在 s1 串中从前向后查找 s2 中某个字符出现的位 置	如找到，返回 s1 中的位 置。如果 s1 和 s2 没有共 同字符，返回 NULL	s2 作为一个字 符集，而不是 串。如， s1= "xyzabg" s2="abc" 返回"abg"
size_t strspn(const char *s1, const char *s2);	在 s1 串中从前向后计数 s2 中字符的个数。即求 s1 串中前面有多少个字符 在 s2 范围中。	返回整数表示在 s1 串中 第一个不在 s2 中的字符 的位置。如果 s1 串的第 1 个字符不在 s2 中，就返 回 0。	s2 作为一个字 符集，而不是 串。如， s1= "cabbage" s2="abc" 返回 5
size_t strcspn(const char *s1, const char *s2);	在 s1 串中从前向后计数 不在 s2 中字符的个数。 即求 s1 串中前面有多少 个字符都不在 s2 范围中。	返回整数表示在 s1 串中 第一个在 s2 中的字符的 位置。如果 s1 串第 1 个 字符在 s2 中，就返回 0。	s2 作为一个字 符集，而不是 串。如， s1= "xyzabc" s2="abc" 返回 3
char *strtok(char *s1, const char *s2);	在 s1 串中查找 s2 中的分 割符，并用 NULL 替代分 隔符，使 s1 分割为多个 子串标记 token	返回第一个分隔符所分 割的 token 串。下面调用 用 NULL 作为 s1 的实参， 可获取后面的字串标记	s2 作为一个分 隔字符集，而不 是串。



## 附录 B

缓冲区管理(按字节处理)			
void * memcpy(void *s1, const void *s2, size_t count)	将 s2 所指的共 count 个字节拷贝到 s1 所指存储区中	目的存储区的始址 s1	内存拷贝
int memcmp(const void *s1, const void *s2, size_t count);	比较 s1 和 s2 所指的区域中各字节的值，比较 count 个字节	如全相同，返回 0。 如果 s1 小于 s2，返回负值，否则就返回正值	内存比较
void * memset(void *buf, int c, size_t count)	将 buf 所指区域设置为 c 值，区域大小 count 字节	该区域的起始地址 buf	内存设置
void *memchr(const void *buf, int c, size_t count);	在 buf 所指区域中查找 c 值，区域大小 count 字节	如找到，返回指针指向找到的字节地址，如未找到，返回 NULL	内存查找

表 B-8 时间函数<time.h>与<sys/timeb.h>

表中 time\_t 是 long 的同义词。

函数原型	功 能	返回值	说 明
time_t time( time_t *timer);	取得系统当前时间，形参用来保存结果。如实参为 NULL，返回值但不保存	一个整数值，表示从 1970-1-1 00: 00: 00 到当前时间的秒数	time_t 是 long 型。 该值每秒改变，常作为伪随机数种子
struct tm *localtime (const time_t *timer);	将 time_t 时间转换为 tm 时间，而且按本地时区调整。	返回 tm 结构值，包括： tm_year:从 1900 开始 tm_mon:0-11, 1 月为 0 tm_day:1-31 tm_hour:0-23 tm_min:0-59 tm_sec:0-59 tm_wday:0-6,周日为 0 tm_yday:0-365,1 月 1 日为 0	函数 gmtime 转换到当前国际标准时间 UTC
char *asctime( const struct tm *timeptr);	将 tm 时间转换为字符串。	返回字符串，格式为： Wed Jan 02 02:03:55 1980	
char *ctime( const time_t *timer);	将 time_t 时间转换为字符串，带时区调整	同上	
time_t mktime( struct tm *timeptr);	将 tm 时间转换为 time_t 时间，tm 结构中前 6 项就可以构造一个有效时	如果转换错误，返回-1。判断方法如下： mktime() !=(time_t)-1	与 localtime 函数作用相反



	间。		
<code>char *_strtime(char * timestr);</code>	把当前系统时间转换为字符串，形参是输出串	指向结果串	格式为： hh:mm:ss
<code>char *_strdate( char *datestr );</code>	把当前系统日期转换为字符串，形参是输出串	指向结果串	格式为： mm/dd/yy
<code>size_t  strftime(char  *strDest, size_t  maxsize, const  char *format, const  struct  tm *timeptr );</code>	对 <code>tm</code> 时间转换为一个格式化字符串，用于显示。	结果串的长度	第 3 个实参要使用大量的格式控制符，请参看文档。
<code>clock_t clock( void );</code>	计算处理器所用时间。可用于延迟或时间区间度量，精确到毫秒。	返回时钟滴答数量。 <code>clock_t</code> 是一个 <code>long</code> 。 <code>CLOCK_PER_SEC</code> 表示每秒时钟滴答数的常量，如 1000。	两次调用返回值之差就是间隔的毫秒数。
<code>void  _ftime( struct  _timeb *timeptr );</code>	获取系统当前时间，形参用来保存结果。 <code>_timeb</code> 结构中包含成员： dstflag: 非 0 表示夏令时 millitm: 毫秒数 time: 即 <code>time_t</code> 值 timezone: 相对 UTC 的时间差(以分钟为单位)，如中国为-480，比 UTC 早 8 小时		比 <code>localtime</code> 函数得到更多信息
<code>void _tzset( void );</code>	根据当前环境变量来设置 3 个全局变量： <code>_daylight</code> ， <code>_timezone</code> 和 <code>_tzname</code> ，详见文档		在执行 <code>_ftime</code> 、 <code>time</code> 或 <code>localtime</code> 函数前应先执行该函数

表 B-9 可变参数&lt;stdarg.h&gt;

函数原型(有参宏)	功 能	返回值	说 明
<code>void  va_start(va_list  arg_ptr, prev_param );</code>	可变参数初始化，使参数表 <code>arg_ptr</code> 指向固定形参 <code>prev_param</code> 之后的头一个可变形参		先用 <code>va_list</code> 说明一个变量，再作为本函数的头一个实参
<code>va_arg( va_list arg_ptr, type );</code>	将 <code>arg_list</code> 中的当前实参转换为指定类型 <code>type</code> 之后，再返回。返回前指向下一个可变实参。	返回读取到的实参值	循环调用该函数将得到多个可变实参



附录 B

void va_end( va_list arg_ptr );	对可变参数表进行复位		最后一步
---------------------------------	------------	--	------

表 B-10 断言<assert.h>与<crtdbg.h>

函数原型(有参宏)	功 能	返回值	说 明
void assert( int expression ); 定义在<assert.h>	计算指定的表达式，如果为 0(false)，就打印该表达式、文件名及行号，然后调用 abort()函数终止程序。如果表达式计算为非 0(true)，不显示任何信息，也没有其它动作。  该宏在调试和运行时都可起作用。如要停止起作用，只需在包含<assert.h>前定义 NDEBUG 宏。		在程序中添加诊断断言，以确保满足特定条件才能继续执行。这是最简单的程序检测手段。
ASSERT( boolExpr); _ASSERTE(boolExpr); 定义在<crtdbg.h>中	这两个宏与 assert 相似。只在_DEBUG宏有效时才起作用，即只在调试程序时才起作用，在运行时刻不起作用。		



## 参考文献

- 1、张岳新等, Visual C++程序设计, 兵器工业出版社, 2004
- 2、Harvey M. Deital, Paul James Deitel 著, 邱仲潘等译, C++大学教程(第二版), 电子工业出版社, 2001
- 3、谭浩强, C++程序设计, 清华大学出版社, 2004
- 4、Bjarne Stroustrup, The C++ Programming Language (3<sup>rd</sup> Edition)。 Addison-wssley Pub Co, 1997
- 5、Microsoft MSDN Library Visual Studio 6.0 , 网 址 : <http://msdn.microsoft.com/en-us/library/ms950417.aspx>
- 6、