



中国科学院希望高级电脑技术公司高级程序设计丛书之五

面向对象的程序设计

**Turbo C++**

程 序 设 计 方 法

叶欣 晓飞 薛梅

编译

魏群 皇甫广宇

海 洋 出 版 社

1991 年 4 月 · 北京

## 内 容 提 要

Turbo C++ 1.0是在Turbo C基础上推出的最新面向对象的程序设计软件包。该书共五册,本书是其中的一册。主要内容分三部分,共三十一章。第一部分综述了Turbo C++,介绍了其集成环境,讨论了编辑器。第二部分介绍C语言的基本知识,并说明了C++中类似C的性质。第三部分完整地描述了C的C++扩充及许多库函数。书中几乎对Turbo C++的每个主要性质都给出了编译及运行的典型程序。书末的六个附录,列出了常用C库函数, Turbo C++ 调试器使用、存储模式、VROOMM覆盖技术、使用命令行编译程序、编译多文件程序等资料。

本书内容丰富,结构合理,它使已熟悉C语言的读者不必再重复阅读熟知的内容,而又可使初次接触C和C++的读者方便地获得充分的背景材料。

### Turbo C++ 程序设计方法

叶欣 晓飞 薛梅  
魏群 皇甫广宇 编译

审 校 刘莉蕾

责任编辑: 阎世尊

海洋出版社出版(北京市复兴门外大街1号)  
海洋出版社发行 双青印刷厂印刷  
开本: 787×1092 1/16 印张: 字数: 560千字  
1991年4月第一版 1991年4月第一次印刷  
印数: 1—3000  
ISBN 7-5027-1392-1/TP·21  
定价: 15.00元

## 前 言

随着计算机软件设计方法的不断提高和改进,面向对象的程序设计在我国也逐渐流行起来并大有迅猛发展的趋势,它提出了一种全新的程序设计思想,把数据和对数据进行的操作融为一体。

美国 BORLAND 公司在 Turbo C 的基础上,推出最新面向对象的程序设计软件包——Turbo C++ 1.0 版。它继承并发挥了原来 Turbo C 集成环境的优良特性,并包含了面向对象的基本思想和设计方法,是目前国际上最受欢迎的面向对象程序设计软件包。

我们在长期从事 Turbo C 程序设计的基础上,加上近几年对面向对象的研究,根据 BORLAND 公司的 Turbo C++ 软件和资料,特编译了 Turbo C++ 程序设计丛书。

本丛书共有五册,分别为《入门》、《用户手册》、《程序员指南》、《库函数参考》和《程序设计方法》,全面地介绍了 Turbo C++ 的基础知识和高级技术,是一套引导读者进行面向对象程序设计的系统参考书。

《程序设计方法》是一本介绍如何利用 Turbo C++ 进行程序设计的教科书,本书从低级到高级、从简单到复杂、从 C 到 C++,循序渐进地介绍了程序设计的一般方法和注意事项。

由于面向对象的概念在国内还处于初步阶段,加上我们水平有限,书中难免会有错误和缺点,敬请广大读者批评指正,以备再版时修订。来信请寄中央财政金融学院信息系叶欣收。

本书在编译过程中,得到了许多同志的帮助和支持,刘学功同志、姜涛同志为本书的成稿做了有益的工作,在此表示谢意。

本丛书出版过程中,得到海洋出版社的编辑同志以及中国科学院希望高级电脑技术公司资料部秦人华经理、杨淑新老师的大力帮助和支持,在此表示衷心的感谢。

编译者

1991 年 4 月于北京

# 目 录

引言	1
第一部分 Turbo C++入门	3
第一章 Turbo C++综述	4
1.1 C++的起源	4
1.2 什么是面向对象程序设计?	4
1.3 C++忠实于C之精华	5
1.4 C++ 的用途	5
1.5 编译程序及解释程序	6
1.6 关于C程序员	6
第二章 Turbo C++集成开发环境	7
2.1 运行 Turbo C++	7
2.2 使用鼠标	8
2.3 主菜单	8
2.3.1 对话框	9
2.4 打开完整菜单	10
2.5 对主菜单的考察	10
2.5.1 系统菜单	10
2.5.2 File	11
2.5.3 Edit	12
2.5.4 Search	12
2.5.5 Run	12
2.5.6 Compile	12
2.5.7 Debug	13
2.5.8 Project	13
2.5.9 Options	13
2.5.10 Window	14
2.5.11 Help	15
2.6 热键	16
2.7 使用 Turbo C++的上下文敏感帮助	17
2.8 理解窗口	17
2.8.1 窗口的大小设置及移动	18
2.9 编辑窗口	19
2.10 信息窗口	19
2.11 状态行	19

<b>第三章 使用 Turbo C++ 编辑器</b>	<b>20</b>
3.1 编辑器命令	20
3.2 激活编辑器并键入文本	20
3.3 删除字符、字和行	22
3.4 文本的移动、拷贝和块移动	22
3.5 剪裁板的使用	23
3.6 光标移动的进一步说明	24
3.7 搜索和替换	24
3.8 设置和搜索位置标识	26
3.9 文件的存储和装入	27
3.10 自动缩格	27
3.11 文本块移入和移出磁盘文件	27
3.12 对匹配	28
3.13 其它命令	28
3.14 命令综述	29
3.15 修改编辑器缺省	30
3.16 用文件名激活 Turbo C++	31
 <b>第二部分 C 语言</b>	 <b>32</b>
<b>第四章 C 语言要素</b>	<b>33</b>
4.1 准备 IDE	33
4.2 C 对大小写敏感	33
4.3 一个简单的 C 程序	33
4.3.1 进一步讨论	34
4.4 错误处理	35
4.5 错误与警告比较	36
4.6 第二个程序	36
4.6.1 一种变化	37
4.7 快速复习	38
4.8 什么是 C 函数?	38
4.8.1 函数和自变量	38
4.8.2 返回值的函数	39
4.8.3 函数的一般形式	40
4.9 两个简单命令	40
4.9.1 if 命令	40
4.9.2 for 循环命令	41
4.10 代码块	41
4.11 字符和字符串	41
4.11.1 字符串	42
4.12 printf(): 快速复习	43

4.13	scanf( )快速回顾	44
4.14	分号、括号和注释	44
4.15	缩排练习	45
4.16	C 库	45
4.17	C 的关键字	45
4.18	术语复习	46
	第五章 变量、常量、操作符和表达式	47
5.1	标识符名称	47
5.2	数据类型	47
5.2.1	类型修饰符	48
5.3	变量说明	50
5.3.1	变量说明的位置	50
5.4	常量	51
5.4.1	十六进制和八进制常量	52
5.4.2	字符串常量	52
5.4.3	转义字符常量	52
5.4.4	变量初始化	53
5.5	操作符	54
5.5.1	算术操作符	54
5.5.2	增量和减量	55
5.5.3	关系和逻辑操作符	55
5.5.4	赋值操作符	58
5.6	表达式	59
5.6.1	表达式中的类型转换	59
5.6.2	空格和圆括号	60
	第六章 程序控制语句	61
6.1	if 语句	61
6.1.1	else 语句的使用	62
6.1.2	If-else-if 阶梯	63
6.1.3	条件表达式	64
6.1.4	嵌套 If	64
6.2	switch 语句	65
6.2.1	default 语句	67
6.2.2	break 语句的进一步讨论	67
6.2.3	嵌套的 switch 语句	68
6.3	循环	70
6.4	for 循环	70
6.4.1	for 循环的基本知识	70
6.4.2	for 循环的变化	71
6.4.3	无穷循环	73

6.4.4	for 循环的中断	73
6.4.5	无循环体循环的使用	73
6.5	while 循环	73
6.6	do_while 循环	75
6.7	嵌套循环	76
6.8	循环中断	78
6.9	continue 语句	79
6.10	标号和 goto	80
第七章	数组和字符串	82
7.1	一维数组	82
7.1.1	无界检测	83
7.1.2	一维数组是一个表	83
7.2	字符串	83
7.2.1	从键盘上读字符串	84
7.2.2	一些字符串库函数	84
7.2.3	空终止符的使用	87
7.2.4	printf( )的一种变化	88
7.3	二维数组	88
7.3.1	字符串数组	90
7.4	多维数组	91
7.5	数值初始化	91
7.5.1	变界数组初始化	92
7.6	一个水下搜索游戏	92
第八章	指针	97
8.1	指针是地址	97
8.2	指针变量	97
8.3	指针操作符	98
8.4	指针表达式	99
8.4.1	指针赋值	99
8.4.2	指针运算	100
8.4.3	指针比较	101
8.5	指针和数组	101
8.5.1	索引指针	102
8.5.2	指针和字符串	102
8.5.3	如何得到一个数组元素的地址	103
8.5.4	指针数组	104
8.5.5	一个使用数组和指针的有趣实例	105
8.6	指针的指针	109
8.7	初始化指针	110
8.8	指针的一些问题	111



第九章 函数：更详尽的说明	113
9.1 函数的一般形式	113
9.2 return 语句	113
9.2.1 从一个函数中返回	113
9.2.2 返回值	114
9.3 函数返回非整型值	116
9.3.1 使用函数原型	116
9.3.2 返回指针	118
9.3.3 void 类型函数	119
9.4 更多的有关原型的知识	119
9.4.1 参数不匹配	119
9.4.2 头文件：更详尽的说明	120
9.4.3 无任何参数的原型函数	121
9.4.4 有关旧式 C 程序	121
9.5 作用域规则	122
9.5.1 局部变量	122
9.5.2 形式参数	124
9.5.3 全局变量	124
9.5.4 作用域最后的例子	125
9.6 函数的参数和自变量：更详尽说明	125
9.6.1 赋值调用和赋地址调用	125
9.6.2 建立一个赋地址调用	127
9.6.3 函数调用与数组	128
9.7 argc、argv 和 env——main 中的参数	131
9.8 从 main() 中返回值	133
9.9 递归	134
9.10 参数说明的传统风格和现代风格	135
9.11 补充问题	136
9.11.1 参数和通用函数	136
9.11.2 效率	137
第十章 输入、输出和磁盘文件	138
10.1 两个预处理指令	138
10.1.1 #define 指令	138
10.1.2 #include 指令	140
10.2 流与文件	140
10.3 流(streams)	140
10.3.1 文本流	141
10.3.2 二进制流	141
10.3.3 文件	141
10.4 概念和实际	141

10.5	控制台 I/O	142
10.5.1	字符读写	142
10.5.2	字符串读写	143
10.6	控制台格式化 I/O	143
10.6.1	printf() 函数	144
10.6.2	scanf() 函数	145
10.7	缓冲型 I/O 系统(ANSI 型 I/O 系统)	148
10.7.1	文件指针	149
10.7.2	打开文件	149
10.7.3	写字符	150
10.7.4	读字符	151
10.7.5	feof() 的使用	151
10.7.6	关闭文件	151
10.7.7	ferror() 和 rewind() 函数	152
10.7.8	fopen(), getc(), putc() 和 fclose() 函数的用法	152
10.7.9	getw() 和 putw() 函数的使用	154
10.7.10	fgets() 和 fputs() 函数	154
10.7.11	fread() 和 fwrite() 函数	154
10.7.12	fseek() 函数和随机访问 I/O	156
10.7.13	标准流	158
10.7.14	fprintf() 和 fscanf() 函数	159
10.7.15	删除文件	161
10.8	非缓冲型 I/O——UNIX 型文件系统	161
10.8.1	open(), creat() 和 close() 函数	162
10.8.2	read() 和 write() 函数	163
10.8.3	unlink() 函数	165
10.8.4	随机访问文件和 lseek() 函数	165
第十一章	高级数据类型	167
11.1	访问修饰符	167
11.1.1	const 常量	167
11.1.2	volatile 易变量	168
11.2	存储类型说明符	169
11.2.1	auto(自动变量)	169
11.2.2	extern(外部变量)	169
11.2.3	static variables(静态变量)	170
11.2.4	static local variables(静态局部变量)	170
11.2.5	static global variables(静态全局变量)	171
11.2.6	register variables(寄存器变量)	172
11.3	赋值语句中的类型转换	173
11.4	函数类型修饰符	175

11.4.1	pascal	175
11.4.2	cdecl	175
11.4.3	interrupt	175
11.5	指向函数的指针	175
11.6	动态分配	177
11.7	分配和释放内存	178
第十二章	用户定义的数据类型	180
12.1	结构	180
12.1.1	访问结构元素	181
12.2	结构数组	182
12.2.1	通讯录实例	182
12.3	结构赋值	188
12.4	将结构传递给函数	188
12.4.1	将结构元素传递给函数	188
12.4.2	将整个结构传递给函数	189
12.5	结构指针	190
12.5.1	结构指针说明	190
12.5.2	使用结构指针	190
12.6	结构内部的数组和结构	193
12.7	位域	194
12.8	联合(union)	197
12.9	枚举	200
12.10	使用 sizeof 来确保可移植性	202
12.11	typedef	203
第十三章	高级运算符	205
13.1	按位运算符	205
13.2	?运算符	211
13.3	C 语言的简写	212
13.4	逗号运算符	213
13.5	方括号和圆括号	213
13.6	运算符优先级表	213
第十四章	屏幕控制函数	214
14.1	基本正文模式函数	214
14.1.1	正文窗口	214
14.1.2	清除窗口	215
14.1.3	光标定位	215
14.1.4	清除到行末	215
14.1.5	删除和插入行	216
14.1.6	建立窗口	217
14.1.7	一些窗口 I/O 函数	219

14.1.8	正文模式	221
14.1.9	用彩色输出正文	221
14.2	Turbo C++的图形子系统介绍	223
14.2.1	一个有别名的窗口	223
14.2.2	初始化显示器适配器	223
14.2.3	退出图形模式	225
14.2.4	颜色和调色板	225
14.2.5	基本图形函数	227
14.2.6	改变绘图色	228
14.2.7	区域填充	228
14.2.8	rectangle()函数	229
14.2.9	创建视口	230
第十五章	C预处理指令	232
15.1	C预处理指令	232
15.2	#define 指令	232
15.3	#error 指令	233
15.4	#include 指令	234
15.5	条件编译指令	234
15.5.1	#if、#else、#elif 和#endif	234
15.5.2	#ifdef 和#ifndef 指令	236
15.6	#undef 指令	237
15.7	#line 指令	237
15.8	#pragma 指令	237
15.9	预定义的宏替换名	239
第三部分	使用 Turbo C++的面向对象性质	241
第十六章	C++概述	242
16.1	什么是面向对象程序设计?	242
16.1.1	对象(object)	242
16.1.2	多态性(polymorphism)	243
16.1.3	继承(inheritance)	243
16.2	C++的一些基本原则	243
16.3	编译C++程序	245
16.4	类及对象的引入	245
16.5	函数重载	247
16.6	操作符重载	249
16.7	再谈继承	250
16.8	构造函数与析构函数	252
16.9	C++关键字	255

第十七章 对类的进一步考察	256
17.1 参数化的构造函数	256
17.2 友元函数	258
17.3 缺省函数变元	262
17.4 正确地使用缺省变元	264
17.5 类与结构之相关性	264
17.6 联合与类之相关性	265
17.7 内部函数	266
17.7.1 在类中建立一个内部函数	267
17.8 对继承的进一步讨论	268
17.9 多重继承	272
17.10 传递对象到函数	276
17.11 对象数组	276
17.12 对象指针	277
第十八章 函数和操作符重载	280
18.1 构造函数重载	280
18.2 C++中的局部变量	281
18.2.1 动态初始化	282
18.3 将动态初始化用于构造函数	283
18.4 关键字 this	284
18.5 操作符重载	285
18.5.1 友元操作符函数	289
18.6 引用	292
18.6.1 非参数的引用变量	294
18.6.2 使用引用来重载单目操作符	295
18.7 操作符重载的又一例子	297
第十九章 继承、虚函数及多态性	301
19.1 派生类指针	301
19.2 虚函数	303
19.3 为什么使用虚函数	305
19.4 纯虚函数及抽象类型	308
19.5 先期联编与迟后联编	310
19.6 派生类中的构造函数及析构函数	311
19.7 多重基类	313
第二十章 使用 C++ 的 I/O 类库	314
20.1 C++ 为何有自己的 I/O 系统	314
20.2 C++ 流	315
20.2.1 C++ 预定义流	315
20.3 C++ 流类	315
20.4 建立自己的插入符和抽取符	315

20.4.1	建立插入函数	315
20.4.2	重载抽取函数	317
20.5	格式化 I/O	319
20.5.1	用 <code>ios</code> 成员函数格式化	319
20.5.2	使用操纵函数	322
20.5.3	建立自己的操纵函数	323
20.6	文件 I/O	325
20.6.1	打开和关闭文件	325
20.6.2	读写文本文件	326
20.6.3	二进制 I/O	327
20.6.4	检测 EOF	329
20.6.5	随机访问	329
第二十一章	其它 C++ 内容	332
21.1	用 <code>new</code> 与 <code>delete</code> 进行动态分配	332
21.1.1	重载 <code>new</code> 和 <code>delete</code>	335
21.2	静态类成员	338
21.3	C 与 C++ 的区别	341
21.4	Turbo C++ 的复数及 BCD 类	341
21.5	基于消息的哲学	344
21.6	最后的话	347
附录 A	常用的一些 C 库函数	348
A.1	串和字符函数	348
A.2	数学函数	356
A.3	操作系统相关函数	362
A.4	其它函数	366
附录 B	使用 Turbo C++ 调试器	375
B.1	为调试准备程序	375
B.2	什么是源级调试器	375
B.3	调试器的基本内容	375
B.3.1	单步调试	376
B.3.2	设置断点	376
B.3.3	监视变量	377
B.3.4	监视表达式格式码	377
B.3.5	监视栈	378
B.3.6	计算一个表达式	379
B.3.7	检测一个变量	379
B.3.8	使用寄存器窗口	379
附录 C	Turbo C++ 的存储模式	380
C.1	8086 处理器系列	380
C.2	地址计算	380

C.3	近指针及远指针	381
C.4	存储模式	381
C.4.1	微模式(Tiny Model)	381
C.4.2	小模式(Small Model)	381
C.4.3	中模式(Medium Model)	381
C.4.4	紧缩模式(Compact Model)	381
C.4.5	大模式(Large Model)	382
C.4.6	巨模式(Huge Model)	382
C.4.7	模式选择	382
C.4.8	存储模式编译选项	382
C.5	强制转换存储模式	383
C.5.1	far	383
C.5.2	near	383
C.5.3	huge	384
C.6	Turbo C++的段指示符	384
附录 D	使用 VROOMM 覆盖技术	385
附录 E	使用命令行编译程序	386
附录 F	编译多文件程序	388

## 引 言

在 80 年代末期, 一种新的程序设计方法开始形成, 这就是所谓的面向对象的程序设计。面向对象程序设计(Object oriented Programming, 以下简称 OOP)融合了结构化程序设计(及其前身)的所有性质, 并为程序员提供了分析及解决程序设计任务的一种令人激动的新方法, 在本书中, 你将会学到 OOP 为什么是重要的, 以及怎样用 Turbo C++来实现 OOP。

C++是 C 的面向对象版本。在 80 年代, C 是最流行的语言, 并且在很长时期内, C 仍将被广泛使用。C 是以其效率、功能及精致而著称的。C++保留了这些重要的性质, 但另增加了对 OOP 的支持。正是这种组合, 才使 C++成为当前仅有的最重要的面向对象程序设计语言。

1988 年, 美国 Borland 公司开始了其对 C++的秘密工作。他们以其极为有力的 Turbo C 作为起步, 在之上加入了 C++的 OOP 扩充。这不是一项轻而易举的任务。虽然 C++易于为程序员所使用, 但对编译器的构造却相当困难。事实上, Turbo C++是 Borland 曾经进行过的最大的语言工程。为创建 Turbo C++, Borland 汇集了当今一些最优秀的编译器设计人员。在 Turbo C++的开发期间, Borland 还改进了与其交互式开发环境的用户接口。

正如你将从本书中所看到的那样, Turbo C++提供了一种有力而灵活的程序设计环境。它产生紧凑而有效的代码, 并且支持几百种库函数和类。

### 关于本书

本书以基础语言 C 开始, 讲解 Turbo C++程序设计语言。由于 C++是 C 语言的一个超集, 所以在学习 C++这前, 应当了解 C。但是, 即使你尚未知晓 C, 仍可以使用本书, 因为它含有你需要的所有背景知识。本书还描述了 Turbo C++的集成环境, 编辑器及许多库函数。在读完本书后, 你将能称自己为一名 Turbo C++程序员。

本书基于如下哲学: 最好的学习方法是实践。为此, 本书含有许多将对其进行编译及运行的典型程序。事实上, 对 Turbo C++的几乎每个主要性质, 都有一典型程序来加以说明。

### 本书的组成

本书的组成是基于如下考虑的: 它应能为两种类型的程序员有效使用, 即已经了解 C 但想要学习 C++ 扩充的程序员 和对 C 及 C++都是第一次接触的程序员。

第一部分综述了 Turbo C++, 介绍了 Turbo C++的集成环境, 并且讨论了编辑器。第二部分说明了 C++中的类似 C 的性质。如果你尚未了解 C, 一定要仔细阅读这部分。但是, 如果你对 C 很熟练, 就可以跳到第三部分, 在那里, 完整地描述了 C 的 C++扩充。

以上述方式组织本书使得有经验的 C 程序员不必再次面对已经熟知的东西, 而另一方面, 又使得初学者能方便地获得充分的背景材料。



### 为何本书适合于你

如果你想要学习用 Turbo C++ 编程, 并且想了解所谓面向对象程序的设计思想, 那么, 本书就很适合于你, 不管你是程序设计的新手还是老手。如你所知, C++ 是基于 C 语言的。事实上, 了解 C 是学习 C++ 的前提。由于本书的组成方式, 如果你已熟悉 C, 就可以不重温已了解的内容而很快进入处理 C 向 C++ 扩充的章节。如果你对 C 和 C++ 都是初次接触, 那么就可从头开始, 一次一章地直到本书结束。

本书除了讲解 Turbo C++ 语言外, 还描述了 Turbo C++ 的集成编程环境、Turbo C++ 编辑器以及各种编译器选项。

# 第一部分 Turbo C++ 入门

# 第一章 Turbo C++综述

在开始探索 Turbo C++令人激动的世界之前,考察它与其它程序设计语言及它所基于的 C 语言的关系是重要的。本章的目的就是要对 C++这种程序设计语言,包括其起源、用途及哲学进行一般性的综述。如果你多少已经了解 C++,并且是一名有相当经验的程序员,就可直接跳到第二章。

本章以 C++的简史开始,如果你对 C++(及 C)本身完全不了解,或初次接触程序设计,将会发现这里的背景信息是有价值的。本章还讨论编译程序及解释程序的差异。如果 Turbo C++是你遇到的第一个编译化语言,就应阅读此节。最后,本章对正走向 Turbo C++的 C 用户安排了一些信息。

## 1.1 C++的起源 70

C++的故事是从 70 年代随着导出 C++的 C 的发明而开始的。C 是由 Dennis Ritchie 发明并最初利用 UNIX 操作系统而实现在 DEC 的 PDP-11 上。C 是始自于一较老的名叫 BCPL 的语言之开发过程的结果;BCPL 由 Martin Richards 所开发,并影响及由 Ken Thompson 所发明的名为 B 的语言,该语言又导致 C 的开发。

如你所可能知道的,C 已成为最广泛使用的编程语言之一。它灵活而有力,并且已被用来建立过去几年中一些最为重要的软件产品。但是,当一项工程的规模超过一定程度时,即使 C 也达到了其极限。虽然实际限制因具体工程不同而不同,但当一程序有 25,000 至 100,000 长时,就很难对之加以维护——因为很难将它作为一个整体而加以理解。为解决此问题,当时正在 Bell 实验室工作的 Bjarne Stroustrup 于 1980 年对 C 语言增加了一些功能,并且他将这种新的扩充语言称为“C with classes”。在 1982 年,名字被改为 C++。

Stroustrup 所做的对 C 的大多数扩充都支持面向对象程序设计。Stroustrup 声称 C++的面向对象性质中的一些是由 Simula 67(这是另一种面向对象语言)激发而来的。因此,C++代表了两种强有力的程序设计方法之融合。

从最初发明 C++以来,它经历了两次修改,分别在 1985 年和 1989 年。当前版本是 2.0, Turbo C++实现的正是 C++的这一版本。

## 1.2 什么是面向对象程序设计?

虽然对什么是面向对象程序设计这一问题的全面描述要等到本书的第三部分才进行,但下面简单的讨论会使你了解大概意思。

面向对象程序设计的最根本特点是让程序员能维护及理解更大、更复杂的程序。实现该目标的关键就是对象(object)。本质上说,面向对象程序设计涉及将数据和在此数据上的操作代码组合与封装起来的对象的创建。对象可含公共元素及私有元素。当对象的一个元素是私有时,只有该对象的其它元素才可存取此元素。公共元素则可为程序的任何其它部分所存取。通过使用私有元素,就使严格控制怎样存取一对象成为可能。使用对象的优点是当正确地运用时,一对象就是一个比组成该对象的各个元素更易于理解与维护的单独

逻辑实体。

面向对象程序设计的另一性质是它允许你建立一个由从最一般到最专门的对象组成的层次。在此层次中，每个对象都继承了在它之前出现的那些对象的性质。创建层次结构这一能力就使得程序员能将程序的各部分仔细地组织成清楚而自我包含的单元。

### 1.3 C++忠实于C之精华

C常被看成是一位于汇编语言(低级层次)及Pascal(高级层次)之间的中级语言，发明C的部分原因是赋予程序员一种能用作汇编语言之替换的高级语言。如你可能知的，汇编语言使用由计算机执行的实际指令的符号表示，在每条汇编语言指令及机器指令之间存在着——对应的关系。虽然这种关系使得编写高效程序成为可能，但这样做却是相当麻烦并易于出错的。另一方面，诸如Pascal这样的高级语言与实际机器距离很大。一条Pascal语句几乎与最终执行的机器指令序列毫无关系。但是，尽管C保留了如Pascal中那样的高级控制结构，它仍能让程序员以一种更接近于机器而非其它高级语言所展示的抽象的方式来对位、字节和地址进行操作。为此，C有时被称作“高级汇编代码”。由于C的这种双重性质，它使程序员能建立不必求助于汇编语言就能非常快而有效的程序。

C的哲学是程序员知道自己想干什么。为此，C语言几乎从不会使程序员感到无法使用，程序员可以以其觉得合适的方式自由地使用该语言。很少有(如果真有的话)运行时间错误检查。例如，如果为了某一奇怪的理由，想覆盖程序当前正驻留的内存，C编译程序将不会采取任何行动来阻止你。这种“程序员即上帝”作法的原因是它能让C编译器创建非常快而有效的代码——因为它将类似于错误检查这样的责任放到了你自己的身上。简而言之，C假定你有足够的聪明来在需要时加上自己的错误检查。

当Bjarne Stroustrup发明C++时，他知道保持C原有精华的重要性，这些精华有效率、中级语言性质以及作主的是程序员而非语言这一哲学。另外，他还要加入对面向对象程序设计的支持。如你将看到的那样，这一目标已经实现了。C++在融入了对象的能力的同时，仍为程序员提供了C的自由性及对C的控制。C++中的面向对象性质，用Stroustrup的话来说，能将程序结构变成清晰而可扩充，易于维护但又不失其效率。

### 1.4 C++的用途

虽然C++最初是设计成用来帮助维护非常大的程序的，但它完全不局限于该用途。事实上，C++的面向对象性质可以被有效地应用到几乎任何编程任务中。将C++用来开发诸如编辑器、数据库、个人文件系统以及通讯程序这样的工程并非不同寻常。此外，由于C++保持了C的高效，很多高性能的系统软件都是用C来构造的(系统软件大体上就是与操作系统相关的那些程序)。

如前所述，用C++可创建一个由相关对象组成的层次。这一性质允许创建专门的面向对象库——它可为许多程序员所共享。因此，即使少得足以用C来方便地维护的程序也可用C++来编写，这只需利用在一个面向对象库中所发现的性质。

C++正在开始为所有编程任务所广泛使用的另一原因是它使得程序能被方便地维护及扩充。如你在本书后面将会知道的那样，当需要实现一新的性质时，将它加到一当前已定义的对象定义中是很方便的。基本上，C++提供了扩充一对象能力的一种很好定义了的方法。

## 1.5 编译程序及解释程序

Turbo C++ 是一编译程序。相反，伴随着计算机的标准 BASIC 是一解释程序。如果你先前从未和编译程序打过交道，请阅读下面几段。“编译程序”及“解释程序”这两个术语是指执行一程序的方法。理论上，任何程序设计语言都可被编译或被解释，但是有些语言通常只是以其中一种方式而执行的。例如，BASIC 通常是解释的，而 C++ 通常是编译的。一程序的执行方式不是由用来书写该程序的语言所定义的。解释程序及编译程序只是操作于程序的源代码的复杂程序。

解释程序每次读入程序源代码的一行，并且完成该行中所含的特定指令。在每次运行程序时，都必须有解释程序。编译程序读入整个程序，并且将它转换成可执行的代码。可执行的代码也被称作为目标、或二进制或机器代码。一旦程序被编译，该代码的一行对执行你的程序就不再有意义。一旦程序被编译后，运行程序时就不再需要编译程序。

在本书及 Turbo C++ 编译器手册中经常会出现的两个术语是“编译时间”及“运行时间”。运行时间指的是在程序实际执行时发生的事件。不幸的是，你将常会看到它们与“错误”相联，即“时间错误”，和“运行时间错误”。但当你成为更优秀的 Turbo C++ 程序员时，这些信息就会很少出现了。

## 1.6 关于 C 程序员

如果你已知道 C，那么就可以立即开始学习 C++。你所了解的有关 C 的几乎一切知识都适用于 C++。事实上，由于 C++ 是 C 的一个超集，所有的 C 程序都隐含地为 C++ 程序。但是要记住，为了编写有效的面向对象的代码，必须重新形成你对程序的思考方式。如果保持开放的头脑，就不会有任何麻烦。

## 第二章 Turbo C++ 集成开发环境

Turbo C++有两种独特的操作方式。第一种是你几乎肯定会在一开始时所使用的方式，叫作集成开发环境(Integrated Development Environment, 简称 IDE)。利用 IDE, 所有的编辑、编译及运行都可为单个的按键及易使用的菜单来控制。事实上, IDE 的使用是如此方便, 以至于对它的操作几乎可凭直觉来进行。另一种操作方式是传统的命令行方法。在这里, 你先用编辑器来建立一个源程序文件, 然后对它进行编译、连接及运行。本书的第一部分将只使用集成环境——由于它更易于操作, 并且可以利用其联机帮助。在命令行中使用编译程序将在附录 F 中说明。

本章的目的是展示 Turbo C++ IDE, 但不进入具体的细节。这时对 IDE 的简单介绍只是使你能了解其功能。你所看到的许多选项起初会显得令人费解, 但是当继续读下去后, 它的意义将会变得清晰。如果已经熟悉了 IDE 的功能, 可以直接进入第三章。

### 2.1 运行 Turbo C++

为运行 Turbo C++的集成式版本, 只需打入 TC, 再按回车键即可。当 Turbo C++开始执行时, 将会看到如图 2-1 如示的屏幕。它由如下四部分组成(次序为从顶到底):

- 主菜单
- 编辑窗口
- 信息窗口
- 状态行

本章就来简单地考察上述各个区域。

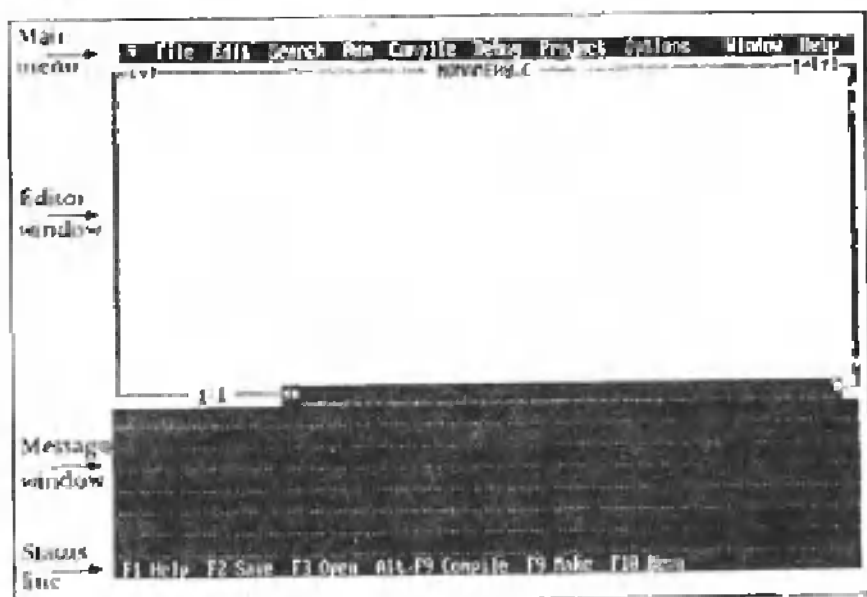


图 2-1 Turbo C++初始屏幕

## 2.2 使用鼠标

Turbo C++的 IDE 能用键盘或者用鼠标来操作。虽然鼠标并非一定必需,但 Turbo C++的 IDE 周全地融入了鼠标支持,而鼠标显然是一很好的改进。

为使下面对 IDE 的讨论更为容易,现在定义几个鼠标操作及术语。一般来说,为选择一项,将鼠标指针置位到该项上,然后按下鼠标左开关。这样做被称为“按动”(click)。有时你需要用“双重按动”(double click)以选择某项。双重按动意味着在两次按动间不移动鼠标而迅速连续地两次按动鼠标左开关。有些对象可在整个屏幕上“拖动”(drag)。为用鼠标来拖动一对象,在该对象的合适部份定位鼠标指针,按下并按住鼠标左开关,然后移动鼠标。当你移动鼠标时,此对象将会以同样方向移动,当它到达所期望的屏幕中的位置时,停止移动鼠标并放开左开关。

## 2.3 主菜单

为激活主菜单,按下 F10 键。这样做了之后,菜单项中的一个将会被高亮度显示。

主菜单用来告诉 Turbo C++完成诸如装入一文件或编译一程序这样的事,或者设置一选项。一旦主菜单被激活,就有两种使用键盘的方法来进行主菜单的选择。第一,可用箭头来移动亮条到所希望的项上,然后按下 ENTER。第二,可只打入所期望菜单项的第一个字母。例如,为选择 Edit,打入 E。打入的字母大小写均可。如果有鼠标,就可在想要激活的主菜单项上按动一次开关。表 2-1 总结了每个菜单选项的功能。

表2-1 主菜单项总结

项	选项(功能)
≡	显示版本号、清除或恢复屏幕以及执行各种和 Turbo C++一起提供的实用程序。
File	装入及保存文件、处理目录、激活 DOS 以及退出 Turbo C++。
Edit	完成各种编辑功能。
Search	完成各种正文搜索及替换。
Run	编译、连接及运行当前被装入环境的程序。
Debug	设置各种编译器选项,包括设置断点。
Project	处理多文件的工程。
Options	设置编译器、连接程序及环境的各种选项。
Window	控制各种窗口显示的方式。
Help	激活上下文敏感的 Help 系统。

当选择了一主菜单项后,就显示一个含若干个选择的下拉菜单。该菜单使你可选择一个与主菜单项相关的行动。可用箭头键来作选择,将亮条移到所期望的项上,并按下 ENTER。或者,可以用不同颜色显示的选项字母。此不同颜色的字母大部分时间都是第一个字母。如果有鼠标,只需对所期望的选项上按动一下开关。在任何时候可按下 ESC 键,或在屏幕的另一部分按动鼠标开关来删除任何菜单。

有时, 在一给定情景下无法利用一菜单项。这时, 在此菜单项中没有显示不同颜色的字母, 并且如果你将亮条移到此选项(或用鼠标对它作一次按动), 那么它将被显示成一条亮条。

有些下拉菜单产生一个与第一个菜单相关的含更多选项的下拉菜单。对此二级下拉菜单的操作与对主菜单的操作完全相同。当一菜单将会产生另一菜单时, 前者就显示出一指向其右部的黑色箭头。

有些菜单项只含 On/Off 选择。为改变一个 On/Off 项的状态, 将亮条移到该项, 然后按下回车键。这将切换当前状态。你还可以用鼠标按动开关或打入以不同颜色显示的字母来完成此改变。

### 2.3.1 对话框

如果一个下拉菜单项后面跟着三个句点, 那么选择此选项就会导致显示一对话框(dialog box)。对话框使得可利用一菜单来完成不易进行的输入, 对话框由下列项的一个或多个组成:

- 动作按钮(action buttons)
- 确认框(check boxes)
- 输入框(input boxes)
- 列表框(list boxes)
- 收音按钮(radio buttons)

让我们来看一下这些项的功能是什么。大多数对话框都有这些动作按钮: Delete, cancel 及 help。为用键盘来激活其中之一选项, 按 TAB 键直到所期望的动作被亮度显示, 然后按下 ENTER。如果有鼠标, 只需在合适的选项上按动一下开关。在一对话框中还可有与此对话框的特定功能相关的其它动作按钮。

一个确认框的形式如下:

☒ option

这时, option 是可被激活或关闭的一选项。对此框中有一“X”时, 该选项就被选择。如果此框为空, 那么就未选择该选项。为改变一确认框的状态, 跳到该框, 然后按空格键。空格键在这时的功能是切换: 每次按下它, 就改变框中的状态。此外, 还可用按动鼠标开关来做到这一点。

一个输入框允许你输入诸如一文件名那样的正文。为激活输入框, 可以不断按 TAB 键直到该框被激活, 也可用鼠标。一旦该框被选择, 就用键盘输入正文。然后按下 ENTER。

一个列表框给出了你可从中加以选择的一组项。为激活该框, 可以不断按 TAB 键, 或者按动鼠标开关。一旦此框被激活, 可通过移动亮条到合适的项, 然后按下 ENTER。或者通过对该框用鼠标作双重按动来选择该项。

收音按钮是一组相互排斥的选项。其一般形式如下:

- ☐ option 1
- ☐ option 2



### ( ) option N

为激活此收音按钮, 可用 TAB 跳至或用鼠标。用箭头键来改变选择或用鼠标来对所期望的选择按动开关

Turbo C++ 对话框的一个例子在图 2-2 中给出。既然你已经了解了怎样使用 Turbo C++ 菜单, 现在就让我们来考察 IDE。

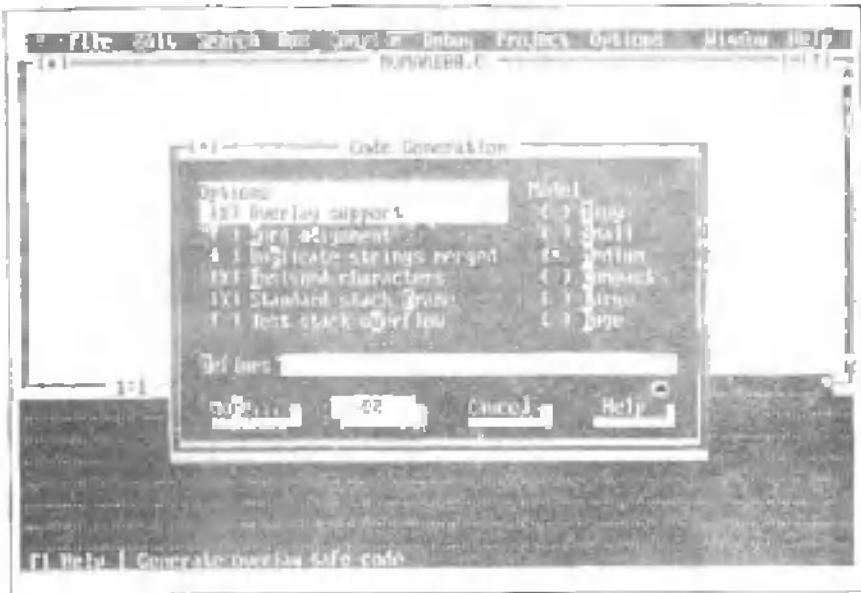


图 2-2 Turbo C++对话框的一个例子

## 2.4 打开完整菜单

由于 Turbo C++ 有着非常丰富和灵活的选项, 所以缺省时并不显示所有的菜单项, 而只给出最常用到的选项。但是, 为了使你能看到所有可能的选项, 有必要显示完整的菜单。为此, 按下 F10 激活主菜单(如果你未), 接着将亮条移到主菜单上的 Options 项, 然后按下 ENTER。屏幕形状如同图 2-3 所给出的那样。如果由于先前的使用, 已打开了完整菜单(Full menus), 则不需做什么, 否则将光标移到 Full menus 处, 然后按下回车键以使之处于 On 状态。按 F10 重新激活主菜单。

现在, 利用左箭头键, 将亮条移到系统菜单项(系统菜单在主菜单的最左边)

## 2.5 对主菜单的考察

在本节中, 我们将简单地考察主菜单中的各项。

### 2.5.1 系统菜单

如果尚未将亮条移到系统菜单项, 那么就先这样做, 然后按下 ENTER。系统菜单告诉你有关 Turbo C++ 版本的信息, 它还让你清除工作区并重新显示屏幕。想要重新显示屏幕的原因是有时一程序在执行时会覆盖视频内存, 从而使屏幕变得混乱。

利用系统菜单, 你还能执行由 Turbo C++ 所提供的一些实用程序

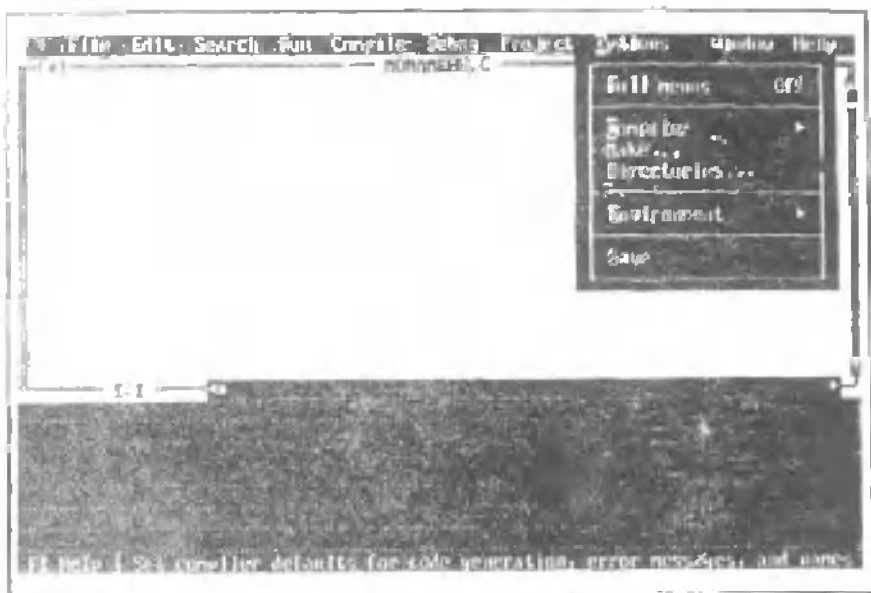


图 2-3 Options 下拉菜单

### 2.5.2 File

将亮条移到 **File**。这将激活如图 2-4 所示的 **File** 下拉窗口。下面就来考察 **File** 中的各个选项。

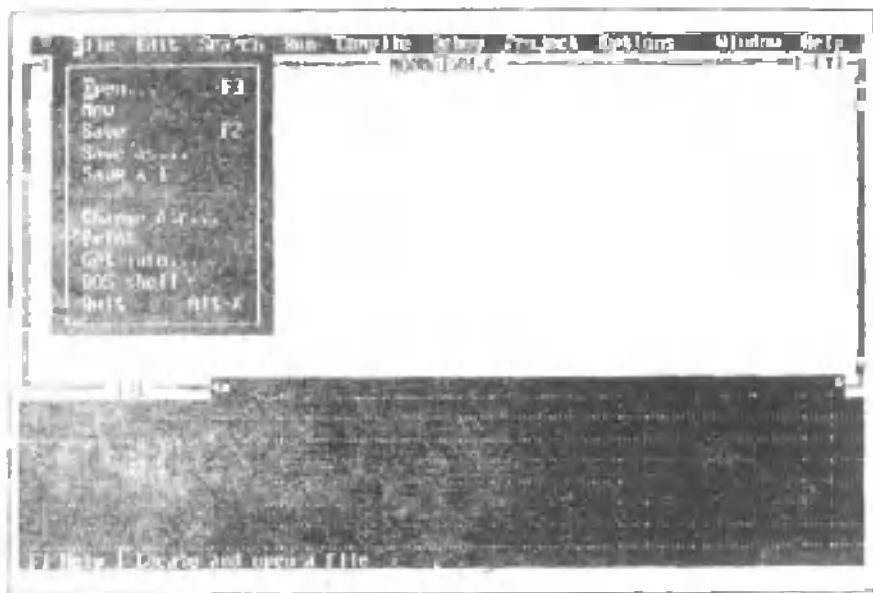


图 2-4 File 下拉菜单

**Open** 选项提示你输入一文件名，然后将该文件装入编辑器。如果此文件不存在，那么就创建它。**Open** 选项还显示可从中加以选择的一组文件。为装入一文件，可用箭头键

来将亮条移到此文件上, 然后按下 ENTER, 也可在该文件名上双重按动鼠标开关。New 打开另一编辑窗口, 并使你创建一新文件, 该文件被称作 NONAMEn.c, 这里 n 为从 0 到 99 的数之一。但是, 你也可以在保存该文件时将它重新命名成所希望的任何名字。Save 选项使你能用一不同的文件名来保存文件。Save all 选项保存所有打开窗口中的文件。Change dir 将当前目录改变成你所说明的。Print 选项打印活动窗口中的文件。Get info 选项显示活动窗口中有关文件的信息。DOS SHELL 选项装入 DOS 命令处理器, 并让你执行 DOS 命令。必须打入 EXIT 以返回到 Turbo C++。最后, Quit 选项退出 Turbo C++。

### 2.5.3 Edit

此时, 按下右箭头键选择 Edit 主菜单项。

Edit 选项使你能完成几种编辑器操作。这些命令及编辑器操作将在下一章中详细讨论。

### 2.5.4 Search

按下右箭头键以选择 Search 主菜单项。

Search 选项使你能对活动窗口中的正文完成几种类型的搜索以及搜索—替换。由于 Search 选项与编辑器关系密切, 它们将在下一章中讨论。

### 2.5.5 Run

Run 选项激活一个含下列六种选择的子菜单:

Run

Program reset

Go to cursor

Trace into

Step over

Arguments

Run 选项执行当前程序。如果该程序尚未被编译, Run 编译它。下面四个选项与使用调试器来执行程序有关。为使用它们, 必须在编译程序时将调试信息选项打开(如缺省时的那样)。虽然调试器的操作在附录 C 中给出, 下面的描述将会简单地说明这些选项的功能。Program reset 选项在你的程序正以调试方式运行时中止它。Go to cursor 执行程序直到它到达光标所在的代码行上。Trace into 选项一次执行程序的一条语句, 如果下一条语句含一子例程调用, 那么就追踪进入对该子例程的执行。Step over 选项执行下一行代码, 但却不追踪进入任何子例程。

Arguments 项被用来将命令行参数传递给从 IDE 中运行的一程序。如果你初次接触程序设计, 并且不知道什么是命令参数, 可参看本书后面的章节。

### 2.5.6 Compile

此时按下右箭头键, 这将激活 Compile 菜单。将会看到如图 2-5 所示的屏幕。第一个选项允你当在编辑器中的文件编译成一个 OBJ 文件。第二个选项将你的程序直接编译成一个可执行文件。第二个选项使你连接当前程序。

**Build all** 选项重新编译与你的程序有关的所有文件, **Remove message** 选项清除 Message 窗口。

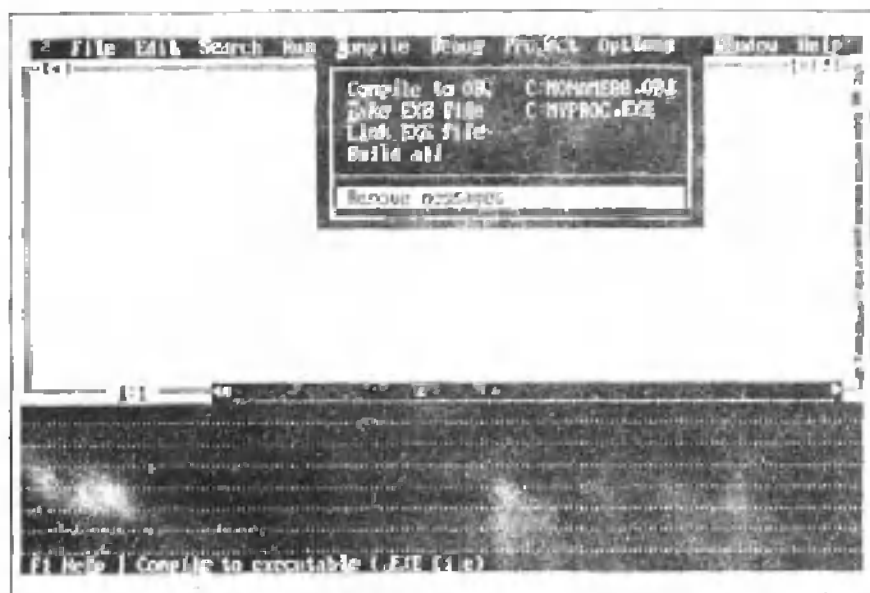


图 2-5 Compile 下拉菜单选项

### 2.5.7 Debug

按下右箭头键就将激活 Debug 主菜单选项。

Debug 选项让你控制 Turbo C++ 集成调试器操作方式。对于目前的情况, 缺省设置将很好地工作, 所以你不必有何担心。

### 2.5.8 Project

Project 主菜单项是用来帮助开发及维护大型的多文件程序的。这将在附录 B 中进一步讨论。

### 2.5.9 Options

现在选择 Options 选项, 下面给出的是在此下拉菜单中的各项(如果你所见的比这里给出的少, 那么可能是忘记打开本章先前描述的 Full menus)。

Full menus

Compiler

Transfer...

Make...

Linker...

Debugger...

Environment...

Save...

除了第一项之外的所有这些选项都可用于改变 Turbo C++ 操作的方式。目前你不必对这些选项多加考虑, 因为 Turbo C++ 的缺省操作方式能很好地工作。

#### 2.5.10 Window

此时激活 Window 选项。

Turbo C++ 的 IDE 是基于窗口的。Turbo C++ 的窗口非常通用而灵活。Window 选项可使你完成对一窗口的各种操作。图 2-6 说明了 Window 下拉菜单。

前面的几项使你对活动窗口完成各种操作。第一选项是 Size/Move。如果选择此选项, 将能改变活动窗口的大小或将它移至屏幕上的一新位置。Zoom 选项增大活动窗口的大小以使它填充整个屏幕。一旦某窗口被放大, 再次选择 Zoom 就将此窗口返回到其原先大小。

IDE 允许同时有几个窗口被打开。Turbo C++ 用来显示多重窗口的方式有两种: 瓦式 (tiled) 或瀑布式 (cascaded)。缺省时, 窗口为瀑布式; 这意味着每次创建一新窗口时, 它将部分地覆盖一个或几个其它窗口。图 2-7 给出了含几个瀑布式窗口的一个例子。相反, 如果选择 Tile 选项, 那么没有窗口将会覆盖另一窗口。每一窗口都为屏幕上一独立部分。图 2-8 说明了与图 2-7 所示的相同的窗口, 差别在于图 2-8 所示的是瓦式格式。

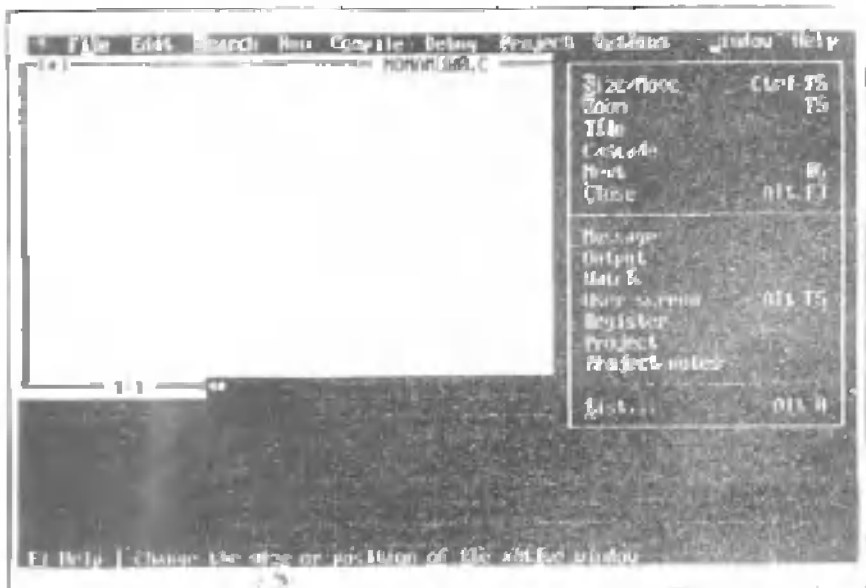


图 2-6 Window 下拉菜单

如果有几个打开窗口, 那么可以通过选择 Next 从一窗口跳至另一窗口。你可选择 Close 来将一窗口从屏幕中移去。

Window 菜单的第一部分允许你激活 Turbo C++ 内部窗口中的一个。Message 窗口是 Turbo C++ 用来输出信息的窗口。Output 窗口在程序于 IDE 的一窗口中执行时显示所产生的输出。如果选择该选项, 那么为返回到 IDE 屏幕, 必须按 F5。Watch 窗口用在调试中。Register 窗口显示 CPU 的每个寄存器的内容。Project 和 project notes 窗口与将在本书后面加以解释的工程相关。

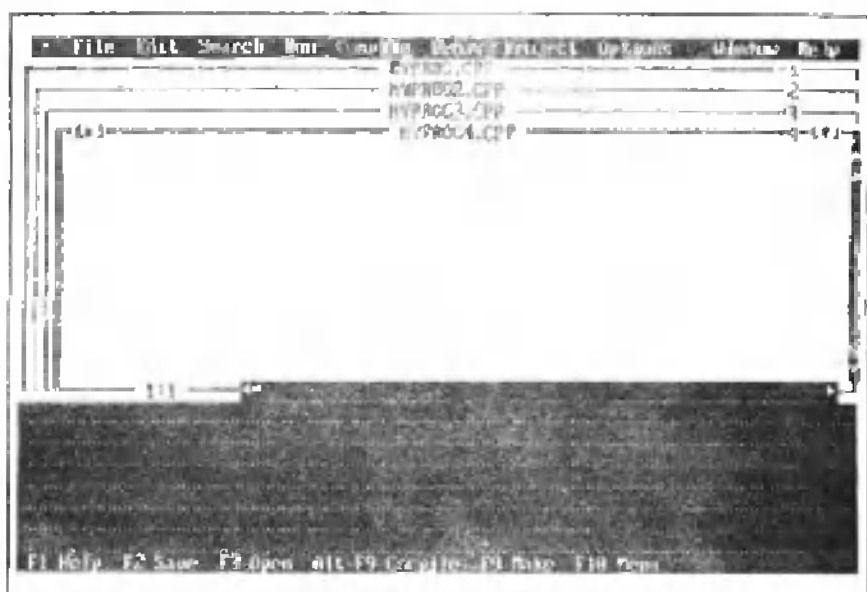


图 2-7 瀑布式窗口

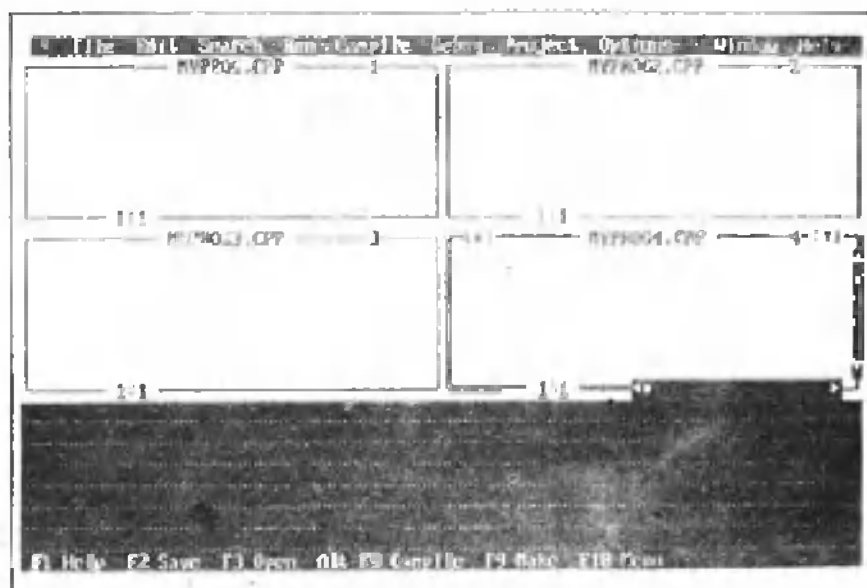


图 2-8 瓦式窗口

为列出所有打开的窗口，选择 **l1st**。你可从此表中选择一项来激活一窗口。

### 2.5.11 Help

现在激活 **Help** 选项。你将会看到下列菜单选择

Contents

Index

Previous topic

Help on help

Contents 选项显示 Help 系统的内容表。Index 激活由 Help 系统所涉及的主题索引。为作出一选择, 将亮条移至你期望的主题, 然后按下 ENTER。这样, 将会看到与所选择有关的信息。为退出 Help 系统, 按下 ESC 键。

Turbo C++ 帮助系统的一个非常方便的性质是当你选择 Topic search 选项时, 光标当前所位于的关键字将引起有关该关键字的信息的显示。为重温先前的主题, 选择 Previous Topic。选择 Help on help, 你就能得到有关 Help 系统的帮助。

现在按下 ESC 键, 这将激活编辑器窗口。

## 2.6 热键

Turbo C++ 最常用的操作可不必进入主菜单就直接激活。这些操作是通过使用热键 (hot key) 来激活的。热键是显示在各个菜单项右边的各种键组合。这些键在你任何需要它们的时候都可使用。表 2-2 总结了各种热键。

表 2-2 热键

热键	意义
F1	激活联机 Help 系统
F2	保存当前正被编辑的文件
F3	装入一文件
F4	执行程序直到遇到光标所在行
F5	调整活动窗口
F6	在窗口间转换
F7	追踪程序进入函数调用
F8	追踪程序, 跳过函数调用
F9	编译及连接程序
F10	激活主菜单
ALT-O	列出打开窗口
ALT-n	激活窗口 (n 必须在 1 到 9 之间)
ALT-F1	说明先前的帮助屏幕
ALT-F3	删去活动窗口
ALT-F4	打开检测窗口
ALT-F5	在用户屏幕与 IDE 之间进行切换
ALT-F7	先前错误
ALT-F8	后面错误
ALT-F9	将文件编译成 .OBJ

---

ALT-空格	激活主菜单
ALT-C	激活Compile菜单
ALT-D	激活Debug菜单
ALT-E	激活Edit菜单
ALT-F	激活File菜单
ALT-H	激活Help菜单
ALT-O	激活Options菜单
ALT-P	激活Project菜单
ALT-R	激活Run菜单
ALT-S	激活Search菜单
ALT-W	激活Window菜单
ALT-X	退出Turbo C++
CTRL-F1	请示对光标所处项的帮助
CTRL-F2	重新设置程序
CTRL-F3	说明函数调用机
CTRL-F4	计算一表达式
CTRL-F5	改变活动窗口的大小或位置
CTRL-F7	设置一查看表达式(调试)
CTRL-F8	设置或清除一断点
CTRL-F9	执行当前程序

---

## 2.7 使用 Turbo C++ 的上下文敏感帮助

Turbo C++ 的 Help 系统在用 F1 热键激活时及使用主菜单激活时的工作有些差别。当按下 F1 激活时, Help 系统是上下文敏感的。这意味着它将显示与你当前正在进行的工作相关的帮助信息。例如, 如果编辑器当前是活动的, 那么按 F1 激活 Help 系统就显示有关编辑器的信息。如果某菜单项是亮度显示的, 那么按 F1 就给出有关亮度显示项的信息。

为了解这是怎样工作的, 激活主菜单并亮度显示 Options 项。现在按下 F1。如你将会看到的, 显示的是与 Options 项有关的信息。当你想结束 Help 系统时, 按下 ESC 键。

## 2.8 理解窗口

Turbo C++ IDE 所基于的窗口是屏幕中的一部分。所有的窗口都有类似的性质。图 2.9 给出了大多数窗口所共有的性质。每个窗口都有一描述此窗口用途的标题, 大多数窗口有一标识该窗口的窗口号, 此外, 所有窗口都含一个用来放大或缩小其大小的调整框 (zoom box)、一个用来移去窗口的关闭框 (close box) 以及一个用来改变窗口大小的重改变角 (resize corner)。调整框、关闭框及改变角可以用鼠标来存取。如果无鼠标, 仍然可完成同样的操作, 但却需要使用专门的键盘命令。

有些窗口, 但非所有窗口, 还有水平及垂直的翻卷杠 (scroll bars)。翻卷杠允许你在窗口中翻卷正文, 它们只能用鼠标操作。当垂直翻卷时, 可通过在垂直翻卷杠的上或下箭头按动鼠标开关来一次翻卷一行。幻灯框 (slider box) 将沿着翻卷杠而移动, 说明在文件中



的相对位置。如果在一箭头上按下并握住鼠标左开关，那么就会产生一连续的翻卷。可在翻卷杠的任何地方按动鼠标左开关，而文件中的相应位置就将被显示。最后，可以沿着翻卷杠拖动幻灯框，而正文就将作相应的翻卷。水平翻卷杠的操作也完全类似。

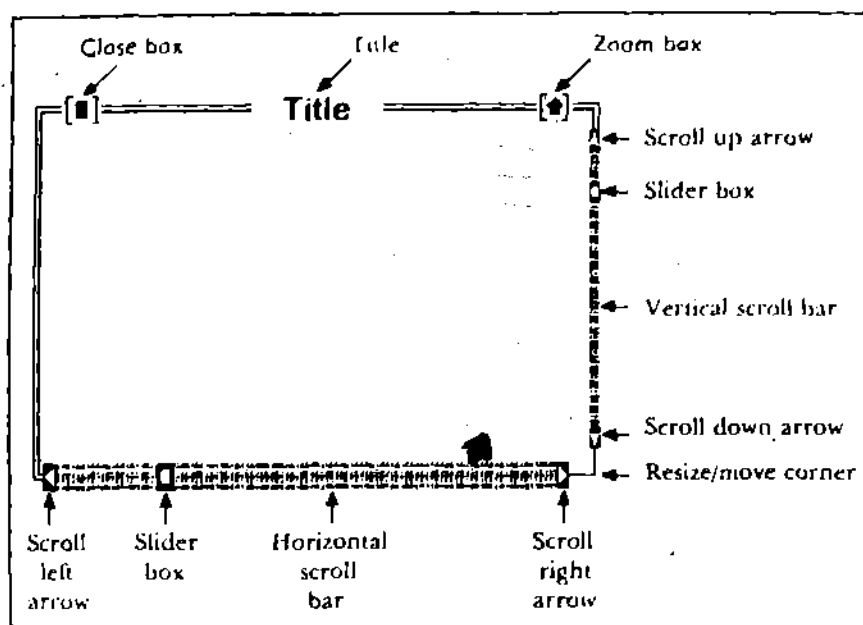


图 2-9 常见的窗口性质

当一个窗口在屏幕上时，就称此窗口是打开的。虽然在一个时刻屏幕上可以有几个打开窗口，但只有一个窗口可以是活动的。此活动窗口是任何从键盘上产生的输入的所在地。有几种使一窗口成为活动的方法。第一，如果知道窗口号，可以按住 ALT 键并按下你所期望的窗口号。窗口号可为 1 至 9 中的一个数。按下 ALT-0 就显示列举出当前正在使用的所有窗口。你还可以从上述列举中选择一窗口来激活该窗口，如果有鼠标，可以对一窗口按动开关来激活它。

### 2.8.1 窗口的大小设置及移动

到目前为止，重新设置窗口大小或移动窗口的最简方法是使用鼠标。为移动一窗口，移动鼠标指针以使它在窗口的顶部边界上，按下并握住左开关，然后将此窗口拖动到新位置处。为改变窗口大小，将鼠标指针移到窗口的改变角上，按下并握住鼠标左开关，然后沿合适的方向移动鼠标。

如果没有鼠标，那么为了重新设置窗口大小及移动窗口，首先应使你想要操作的窗口成为活动的。接着，激活 Window 主菜单项，并且选择 Size/Move 选项。现在，利用箭头键，就可以将窗口在屏幕上四处移动。为改变其大小，按住 SHIFT 键，并同时使用箭头键。

记住对话框及菜单不是窗口，它们不能被改变大小或移动。

## 2.9 编辑窗口

编辑窗口是你为 C++ 程序创建源代码的地方。该窗口的标题是 NONAME00.C。这样的原因是当执行 Turbo C++ 时，你没有说明任何文件名，所以编辑器自动地给出该文件一临时名字。在保存一文件时可以更改其名。注意：编辑窗口的窗口号为 1 要记住的是在任何时候都可以有几个打开的编辑窗口。

由于编辑窗口是一窗口，所以它可以被移动或改变大小。下一章将更详细地讨论编辑窗口。

## 2.10 信息窗口

信息窗口位于编辑窗口之下，它被用来显示各种编译或连接信息。大多数时候，它是用来显示由编译器所产生的错误信息的。你可以移动信息窗口或改变它的大小。

## 2.11 状态行

屏幕底部的行被称为状态行，它显示有关当前正在完成的动作的一简短注释。例如，当选择了 Window 主菜单选项时，状态行显示出下列信息：

F1 Help | Open, array, and list windows

显示在状态行上的信息的价值在于它提供了了解 IDE 当前主要操作的线索。

现在，你已经了解了 Turbo C++ 编程环境的基本内容，下面就应该来学习使用编辑器了。

## 第三章 使用 Turbo C++ 编辑器

在本章中你将学会使用建立在 Turbo C++ 集成开发环境中的编辑器。其操作与原 Turbo C 编辑器类似。在 Turbo C++ 编辑器中增加了如下内容：支持鼠标；可用菜单执行一些文本操作和搜索命令；可建立多编辑窗；同时编辑两个或更多的文件。若你已知道如何使用原 Turbo C 编辑器，那么可能仍还希望阅读本章以了解这些内容。

在 Turbo C++ 编辑器中含有约 50 条命令，并且其功能很强。然而，你没有必要立即就想掌握全部的命令。最重要的是掌握插入、删除、块移动、搜索和替换。一旦你掌握了这些基本内容，将会很容易地学会其它的编辑命令并在需要时使用它们。实际上，学习使用编辑器是很简单的，因为可以随时调用 Turbo C++ 的联机上下文敏感 Help 系统。

若当前没有执行 Turbo C++ IDE，在提示符下键入 TC 以立即启动 Turbo C++。

### 3.1 编辑器命令

首先，了解你如何把命令给 Turbo C++ 编辑器是很重要的。除很少的例外，几乎所有的编辑命令都以一个控制字符开始，许多后面还跟着别的字符。例如，CTRL+Q F 告诉编辑器搜索一 Q 字符串的命令（本书用缩写 CTRL 表示控制键）。若要执行该命令，则按下 CTRL 控制键，并同时按 Q，然后再按 F 或 L。

尽管所有的编辑命令都可以从键盘上输入，但一些可由菜单输入，一些可用鼠标执行。用菜单还是鼠标需加以指出。

### 3.2 激活编辑器并键入文本

当 Turbo C++ 开始执行时，编辑窗口是活动的。当你激活主菜单执行一些操作时，可以通过按 Esc 键返回到编辑窗口。

编辑窗口的顶行显示当前正在被编辑的文件名、编辑窗口的标题，在编辑窗口的底行左边显示的是光标位置的当前行和列。

当编辑窗口是活动的并且你不发出命令时，它将准备接收输入。这就是说，当你在键盘上敲入一个键时，它们将出现在编辑器中当前光标位置。

缺省时，编辑器处于插入方式。这就是说当你输入文本时，它（若有的话）将被插入光标所在的任何一个地方；相反为覆盖方式。在这种操作方式下，新文本可覆盖已存在的文本。你可以通过按 Ins 键在这两种方式之间进行切换。通过光标的形状可以分清哪种方式是活动的。在插入方式中，光标用一个闪烁的下划线表示。在覆盖方式下，其是一个闪烁的矩阵。

注意编辑窗口是活动的并键入下列行：

```
This is a  
test of the  
Turbo C++ editor
```

若在键入中有错误，可用 BACKSPACE 更正之。屏幕将是图 3-1 中的情形。注意光标位置及与之相关的行、列显示在编辑窗口的左下方。此外，注意现在还有一个星号出现在

行、列指示器之左端，星号的出现仅在文件被修改之后。

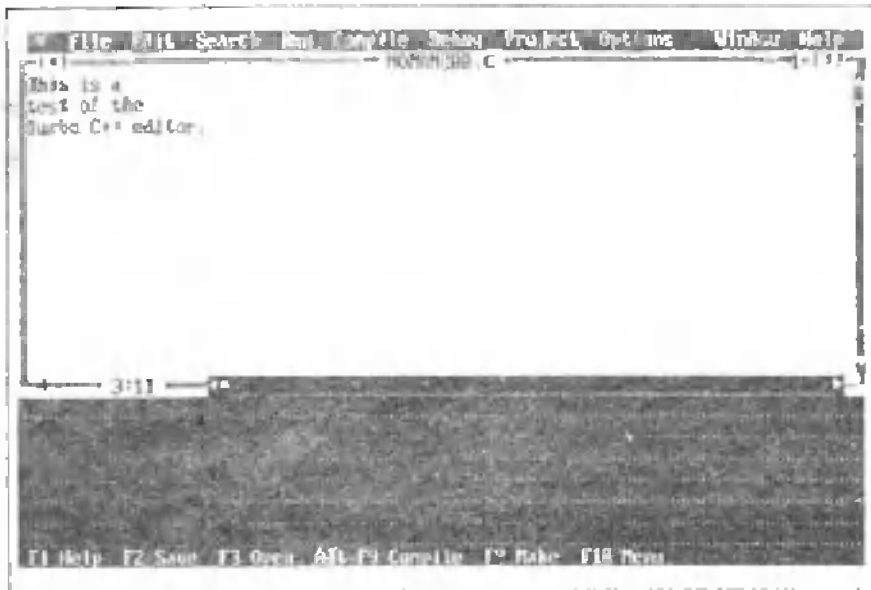


图 3-1 键入文本时的屏幕

由于 Turbo C++ 编辑器是一个全屏编辑器，可以用箭头键随意地在文本中移动光标。另外，按鼠标时，光标将移到鼠标指示器的位置。此时，用箭头键或鼠标把光标移至“test of the”行的最左端。现在键入 very small 并按 ENTER。做完之后，观察已有行被移到右边而不是被覆盖的过程，这就是编辑器在插入方式的结果。若你使编辑器处于覆盖方式，原行可能将被覆盖。屏幕将有图 3.2 所示的情形。

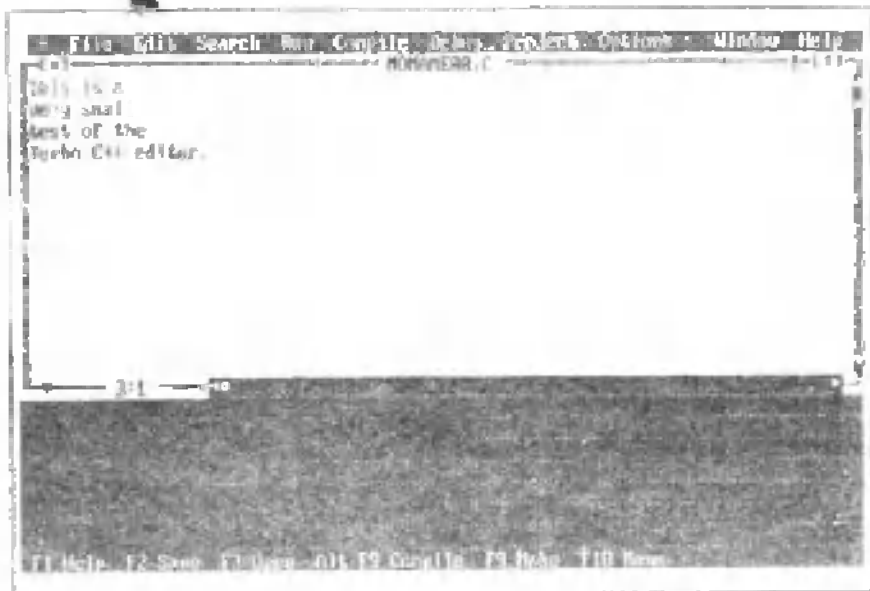


图 3-2 插入一行后的屏幕

### 3.3 删除字符、字和行

可以用两种方法删除一个字符：用 BACKSPACE 键或用 DEL 键。BACKSPACE 键删除光标左边的字符；而 DEL 键删除光标处的字符。

可以用 CTRL-T 删除光标右边的整个字，一个字即为由下列字符定界的字符集：

空格 \$ / - + \* ' ^ [ ] ( ) . , ; < >

可以用 CTRL-Y 删除一整行。无论光标处于行中的何处——整行将被删除。现在你可试着删除一些行和字了。

若希望删除行中从当前光标位置到行尾的内容，用 CTRL-Q Y。

### 3.4 文本的移动、拷贝和块移动

Turbo C++ 编辑器允许你操作文本中的块。可以把它移动或拷贝到另一个地方或把它删除。要做这些操作，首先必须定义一个块，该块可以短至一个字符长至整个文件。然而一般情况下块居于这两个极端之中。可以用两种方法定义一个块：用键盘或用鼠标。若要用键盘定义一个块，则把光标移至块首并键入 CTRL-K B；然后，把光标移至块尾并键入 CTRL-K K，你定义的块将亮度显示。若要用鼠标定义一个块，首先把鼠标指示器置于块头；然后，按着鼠标左按钮并把鼠标移至块尾；最后，放开该按钮。

例如，把光标移至第二行之首“.”并键入 CTRL-K B；然后，把光标移至最后一行之尾并键入 CTRL-K B；然后，把光标移至最后一行之尾并键入 CTRL-K K(或用鼠标)。屏幕应为图 3-3 所示的情形。

若要移动文本中的块，把光标置于你希望文本所置之处并键入 CTRL-K V。这将导致原定义的文本块从其当前位置删去并置于该新的位置。

若要拷贝一块，键入 CTRL-K C。屏幕将有图 3-4 所示的情形。现在可以用这些命令试一试。

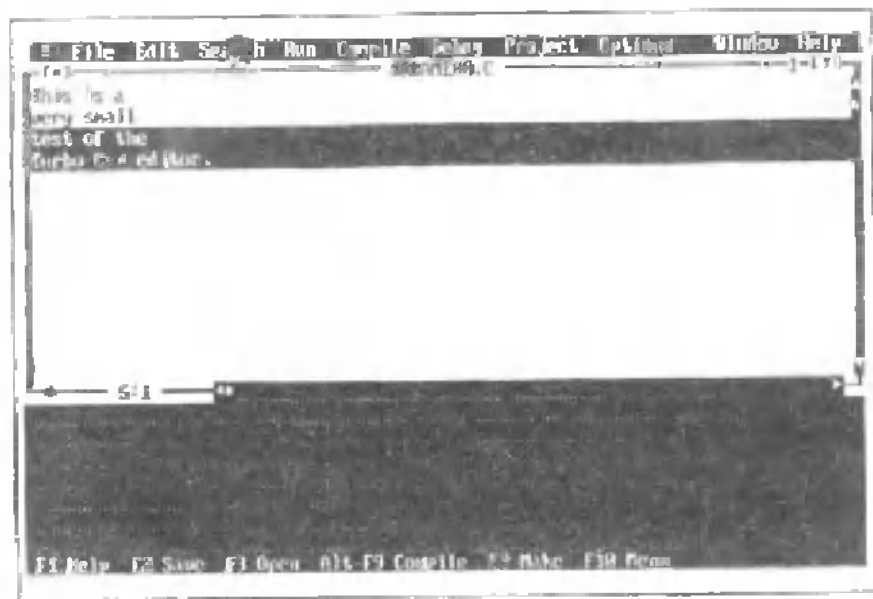


图 3-3 定义块后的屏幕

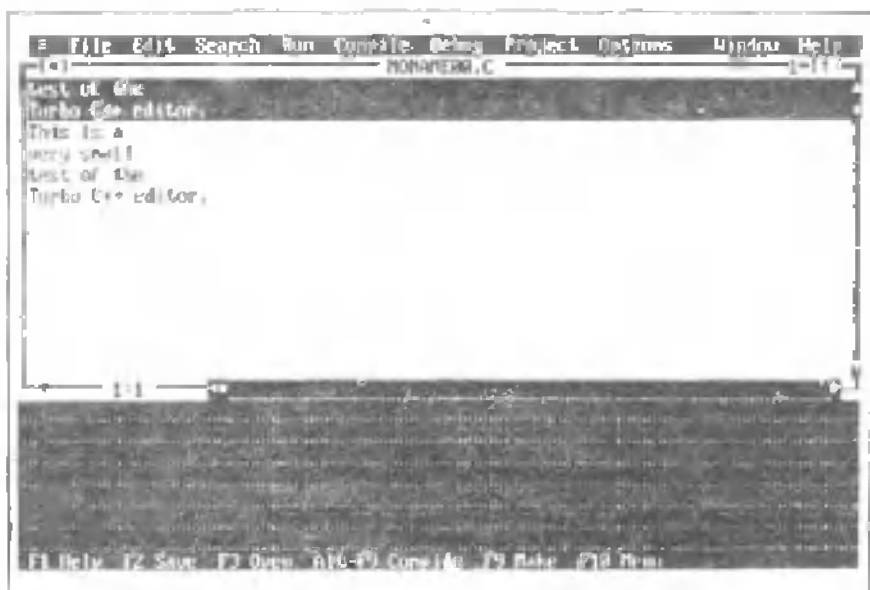


图 3-4 移动块后的屏幕

要删除当前标识块，键入 **CTRL-K Y** 还可以通过激活主菜单上 **Edit** 选项并选择 **Cut** 项来执行该命令。在执行该命令的两种方法中，你删除的块都将自动地被放置于一个叫做剪裁板的特殊编辑窗口中

可以通过将光标移至一个字的首字符之处，键入 **CTRL-K T** 来标识一个字。

若要使一整块都凹进去一个字符，用 **CTRL-K I** 命令，其逆过程可用 **CTRL-K U**。

### 3.5 剪裁板的使用

使用 **Edit** 主菜单选项，可以利用剪裁板增加文本移动和拷贝的灵活性，而且，正如你将要看到的那样，使用剪裁板可以容易地在两个编辑窗口之间移动文本

通常，剪裁板是从另一文件拷贝而来的文本段的临时存放处。若要把文本移入剪裁板，需要标识区域并删除该块或用 **Edit** 菜单选择 **Copy** 选项；若选择了 **Copy**，该块则不从文件中删除，但它还被拷贝到剪裁板中

若要从剪裁板中提取一个文本块，用 **Edit** 菜单 **Paste** 命令。这将导致剪裁板中最新删除或拷贝的文本块到当前编辑窗口中的当前光标位置。

可以通过选择 **Edit** 菜单上 **Show clipboard** 选择看到剪裁板上的内容，这还将激活剪裁板内容的任何部分

希望删除一个块而不把它拷贝到剪裁板中，则首先选择该块并执行 **Edit** 菜单上的 **Clear** 选项。这将删除该块，但不把它拷贝到剪裁板中

可以通过装入另一个文件或选择 **File** 菜单上的 **New** 选择项来同时对多个文件进行编辑，只要简单地在源窗口中定义块，把它拷贝到剪裁板中。这将导致另一个编辑窗口的建立。若要从一个窗把文件拷贝到剪裁板中，然后再把它粘帖到目标窗口。

最后，当激活 **Help** 系统询问有关 C++ 特征的信息时，可能会通过选择 **Edit** 菜单上的

copy example 选项来把该例子代码自动地移到剪裁板上。

### 3.6 光标移动的进一步说明

Turbo C++编辑器有大量的特殊光标命令。这些命令列于表 3-1 中。现在你可以用这些命令试一试。当然，可以通过把鼠标指示器置于你希望的位置并按键以移去光标。

表 3-1 光标移动命令

命令	功能
CTRL-A	移至光标左边字的字首
CTRL-S	左移一个字符
CTRL-D	右移一个字符
CTRL-F	移至光标右边字的字首
CTRL-E	光标上移一行
CTRL-R	光标上移一屏
CTRL-X	光标下移一行
CTRL-C	光标下移一屏
CTRL-W	下翻屏幕
CTRL-Z	上翻屏幕
PGUP	光标上移一屏
PGDN	光标下移一屏
HOME	光标移至行首
END	光标移至行尾
CTRL-QE	光标移至屏幕顶部
CTRL-QX	光标移至屏幕底部
CTRL-QR	光标移至文件头
CTRL-QC	光标移至文件尾
CTRL-PGUP	光标移至文件头
CTRL-PGDN	光标移至文件尾
CTRL-HOME	光标移至屏幕顶部
CTRL-END	光标移至主屏幕底部

### 3.7 搜索和替换

若要搜索一特定的字符系列，用 CTRL-Q F 命令。将得到如图 3-5 所示的对活窗口的提示。你还可以指定不同的搜索选项，这些搜索选项修改执行搜索的方法。缺省选项即是自当前光标位置向前搜索，并且大小写敏感及允许子串匹配。现在让我们来看看各个搜索选项。

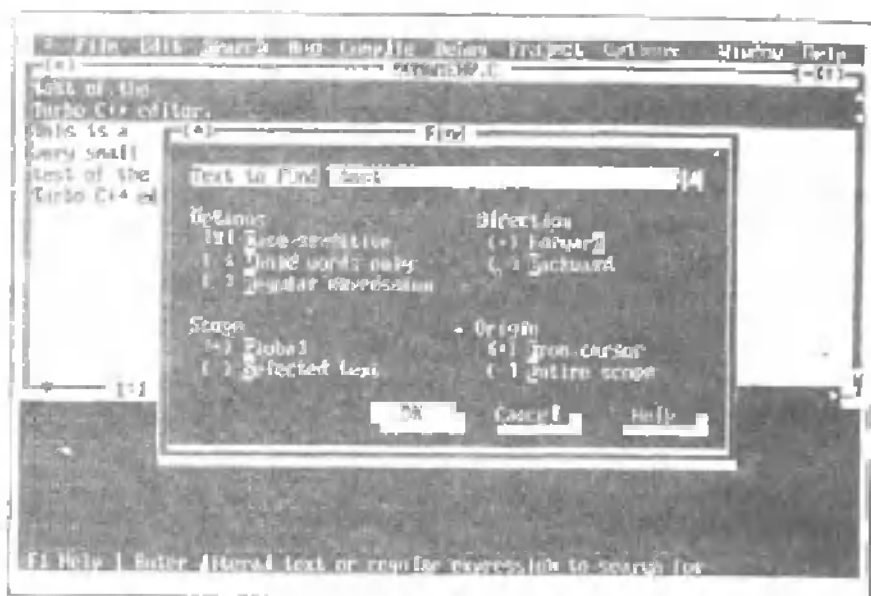


图 3-5 Find 对话框

缺省时，对输入串进行的搜索是从当前光标位置开始对文件进行向前搜索(直到文件结束)。可以通过选择 **Backward** 选项把之改为搜索沿相反的方向。还可通过选择 **Entire scope** 选项搜索覆盖整个文件。

缺省时，搜索是大小写敏感的。这就是说大写字母和小写字母被当作不同的字符对待。可以通过选择 **Case sensitive** 选项把之改为搜索沿相反的方向。还可通过选择 **Entire scope** 选项搜索覆盖整个文件。

缺省时，若输入串包含在另一个较长的串中，这也会引起匹配(这叫做子串匹配)。例如，若输入 **is** 作为搜索串，则编辑器将在字 **"this"** 找到匹配。可以通过确认 **Whole words only** 框使得搜索时只有整个字匹配才算匹配。

可以通过选择 **Selected text** 选项把搜索限制在一个块中。

若确认了 **Regular Expression** 框，则在串搜索中可使用表 3-2 中的通配符。下面是一些例子。

Expression	Matches
hello	hello (and others)
^st	test (at start of line)
test\$	test (at end of line)
[two]	t, w, or o
x*	x, xx, xxx, and so on

记住，若要使用正则表达式，必须确认 Find 对话框中 **regular expression** 框。

可以按 **CTRL-L** 重复一个搜索过程。当在文件中寻找某些特定的东西时，这显得很方便。

若要激活替换命令，键入 **CTRL-QA**。其操作除了允许你用另一个串替代寻找的串之外，其它与搜索命令是一样的。你将看到如图 3-6 所示的对话框。



表 3-2 正则表达式的通配字符

字符	用途
^	匹配一行之首。
\$	匹配一行之尾。
.	匹配任一字符。
*	匹配其前面的字符出现次数(包括0)。
+	匹配其前面的字符出现次数(不包括0)。
[string]	匹配string中任一字符的一次。可用连字符指定一个范围。若串中的第一个字符是^, 则将匹配除串中字符之外的任何字符。
\	使限在其后的字符当作其本身的含意理解而不是当作一个通配符。

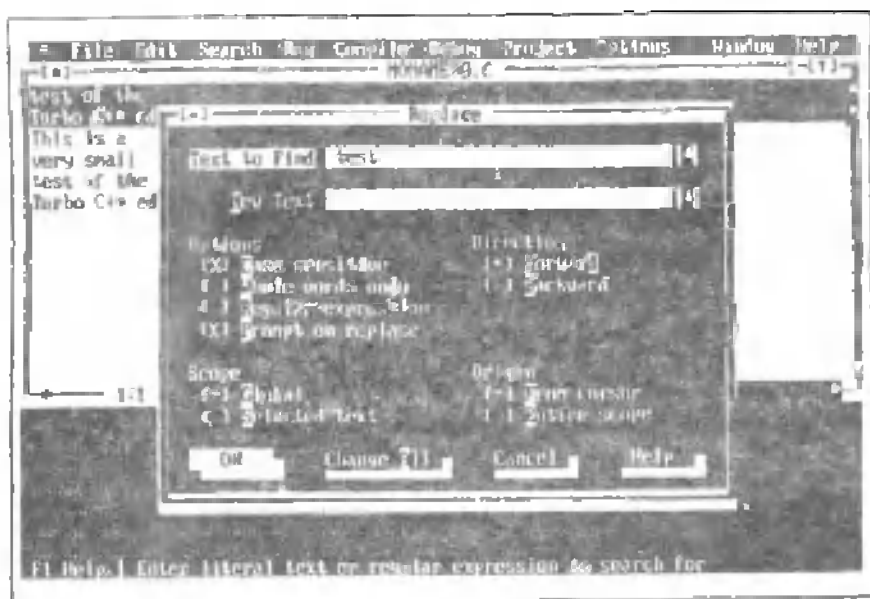


图 3-6 替换对话框

正如你所看到的, 在 Replace 对话框中的选项与 Find 对话框中的选项很相似, 只是前者比后者多一项。可以通过不选择 Prompt on replace 而去掉该特征。

可以通过首先键入 CTRL-P, 后面跟着希望的控制字符来把控制字符输入到搜索串中。用 search 菜单选项也可以激活 Find 和 replace 选项。

### 3.8 设置和搜索位置标识

可以通过键入 CTRL-K n 文件中设置多达四个位置标识符, 其中 n 是位置标识符的个数(0~3)。在一个位置标识符被设置之后, 命令 CTRL-Qn(这里 n 为标识符号)使光标移至该标识符, 位置标识符在大文件中使用是很方便的。

### 3.9 文件的存储和装入

有三种存储文件的方法。其中的两种是把文件存入具有以编辑窗口标题为名的文件中。第三种方法是使文件存在一个不同的名下，然后使其变成文件的当前名。让我们来看看这三种方法是如何工作的。

现在，按 F10 退出编辑器并返回主菜单，选择 File 选项。正如你在第二章了解到的，Save 选项把当前编辑器中的内容存放在一个磁盘文件中，该文件具有窗口标题相同的名字。若你没有指定待编辑的文件，则激活该选项将导致文件存入 NONAME00.C 中。当然这并不会影响到什么。你可能希望使用一个另外的名字。为此，Turbo C++ 将提示你给出一个别的文件名，仅当 NONAME00.c 是一个源文件名时，该提示才出现。否则，文件的存储将不经进一步的交互。

若希望把编辑器中的内容存入一个其名不为编辑器状态行上的名的文件中，用 Save as 选项。(要使用该选项，Full menus 必须为 On)。这就允许输入一个文件名，它为你希望写入当前编辑器中内容的文件，它还使之成为缺省文件名。现在选择该选项。当提示输入文件名时，输入 test。这导致文件的存储，按 F2 键还可以从编辑器中存文件，这与 File 菜单中的 Save 选项相同。

若要装入一个文件，可在编辑窗口中按 F3 或从 File 菜单选择 Open 选项。这将引起一个对话框的显示，提示你输入希望装入的文件名，有两种指定文件名的方法。首先，可以键入之；其次，可以通过对话框中显示的文件列表来加以选择。缺省时，所有以 C 为扩展名的文件都被显示。若有一个鼠标，可以在所需的文件外选择之，这样即被装入。

缺省时，当存储一个磁盘上已有的文件时，文件的旧版本不被覆盖。它将被保存在一个备份文件中，其扩展名被改为.BAK。

### 3.10 自动缩格

很清楚，好的程序员会使用缩格以使其书写的程序清晰并易于理解。为方便这一过程，在你按 ENTER 之后，Turbo C++ 编辑器将自动把光标移动至上一行输入所在的列上，假定此时自动缩格有效。你可以按 CTRL-O 1 使之有效或无效。要了解自动缩格是如何工作的，严格按照下列显示的格式输入下列行：

```
This is an illustration
      of the auto-indentation
      mechanism
      of the Turbo C++
      editor.
```

当你输入该文本之后，注意 Turbo C++ 是如何自动完成缩格的。在输入 C++ 源码时，将会发现这一功能是很方便的。

按 CTRL-O 1 可使自动缩格无效。

### 3.11 文本块移入和移出磁盘文件

可能需要把一个文本块移到一个磁盘文件以备以后使用，这可以通过先定义一个块并按 CTRL-K W 来完成。在你做完上述操作之后，将提示输入希望存放该块的文件的文件名。文本的源块不从程序中删除。

若要读入一个块，键入命令 CTRL-KR。将得到输入文件名的提示。文件内容将被读

人在当前光标位置处。

对于把文本在两个或多个文件中移动，这两条命令是最为重要的，而这又是在程序开发的过程中经常碰到的。

### 3.12 对匹配

正如你将要看到的，C++中有几组界符是成对的。例如，{}，[]和()。在很长或很复杂的程序中，有时要手工去寻找一个界符的配对是很困难的，而用编辑器自动寻找相应的配对界符是可能的。

Turbo C++编辑器将能搜索如下界符对的配对：

```
{ }  
[ ]  
( )  
< >  
/* */  
" "
```

若要搜索匹配界符，则把光标放在你希望匹配的界符处，并键入 CTRL-Q[进行向前匹配(用 CTRL-Q]进行向后匹配)。

有些界符是可嵌套的，这些界符有{}、[]、()、<>，有时候还有注释符(当选择了嵌套注释选项时)。编辑器将按 C++方法搜索适当的匹配符。若因某种原因编辑器找不到适当的匹配界符，光标将不移动。

### 3.13 其它命令

可以在提示符下按 CTRL-U 或 ESC，或在屏幕上对话框之外的任何地方按鼠标器终止请求输入的任何一条命令。例如，若你执行 Find 命令，并且又改变了主意，则可按 ESC 或在框外按鼠标器。

若希望把一个控制字符输入文件，键入 CTRL-P 后面跟着你要输入的控制字符。控制字符用低亮度显示，这依赖于系统配置。

若要在光标移开一行之前恢复对该行所做的修改，则按 CTRL-Q L。你还可以通过选择 Edit 菜单中的 Rrestore line 来消除对一行的修改。记住，一旦光标移开了该行，所有对之的修改都被确认。

若希望移至一块之首，输入 CTRL-Q B，输入 CTRL-Q K 将移至一块之尾。

可用 CTRL-K P 命令打印文件。若没有块被定义，该命令将打印整个文件。否则，其将仅打印一块。

还有一个特别有用的命令 CTRL-Q P，它把光标移至其原先的位置。若希望搜索某个对象然后返回原来所在之处时，这是很有用的。

缺省时，当你按 TAB 键时，一个制表字符就被输入到你的文件中，然而，用命令 CTRL-O T 将使等量的空格插入一个制表符。CTRL-O T 命令变换处理制表符的两种方法。

缺省时，当你在一新行之首按 BACKSPACE 键时，光标将自动移至缩格处。你还可以

用 CTRL-O U 命令来变更这一功能。当它无效时, 每次按 BACKSPACE 键时, 不管其缩格多深, 光标都将只退加一个空格。

### 3.14 命令综述

表 3.3 列出了所有的 Turbo C++ 编辑命令。

表 3-3 Turbo C++ 编辑命令

功能	命令
光标命令	
左移一个字符	左箭头或 CTRL-S
右移一个字符	右箭头或 CTRL-D
左移一个字	CTRL-A
右移一个字	CTRL-F
上移一行	上箭头或 CTRL-E
下移一行	下箭头或 CTRL-X
上翻	CTRL-W
下翻	CTRL-Z
上翻一页	PGUP 或 CTRL-R
下翻一页	PGDN 或 CTRL-C
移至行首	HOME 或 CTRL-Q S
移至行尾	END 或 CTRL-Q D
移至屏幕顶	CTRL-Q E
移至屏幕底	CTRL-Q X
移至文件头	CTRL-Q R
移至文件尾	CTRL-Q C
移至块首	CTRL-Q B
移至块尾	CTRL-Q K
移至原光标位置	CTRL-Q P
插入命令	
选择插入方式	INS 或 CTRL-V
插入一空行	ENTER 或 CTRL-N
删除命令	
删除整行	CTRL-Y
删除至行尾	CTRL-Q Y
删除左边字符	BACKSPACE
删除光标处字符	DEL 或 CTRL-G
删除右边字符	CTRL-T

## 块命令

标志块首	CTRL-K B
标志块尾	CTRL-K K
标志一个词	CTRL-K T
拷贝一个块	CTRL-K C
删除一个块	CTRL-K Y
藏匿或显示一个块	CTRL-K H
移动一个块	CTRL-K V
把一个块写入磁盘	CTRL-K W
从磁盘中读入一个块	CTRL-K R
缩格一块	CTRL-K I
退格一块	CTRL-K U
打印一块	CTRL-K P

## 搜索命令

搜索	CTRL-Q F
搜索和替换	CTRL-Q A
搜索一个位置标识符	CTRL-Q(NUM)
重复搜索	CTRL-L

## 对匹配

向前匹配对	CTRL-Q[
向后匹配对	CTRL-Q]

## 其它命令

中断	CTRL-U或ESC
变换自动缩格方式	CTRL-O I
控制字符前缀	CTRL-P
退出编辑器	F10
新文件	F3
存储覆盖错误信息	CTRL-Q W
存储	F2
设置一个位置标识符	CTRL-K(NUM)
变换制表方式	CTRL-Q T
撤消修改	CTRL-Q L
变换退格方式	CTRL-O U

---

### 3.15 修改编辑器缺省

可以通过从主菜单中选择 Options, 然后选择 Environment 项来修改编辑器操作的一些方法 (在做这一工作需要使 Full menus 为 On)。你将看到如图 3-7 所示的对话框。

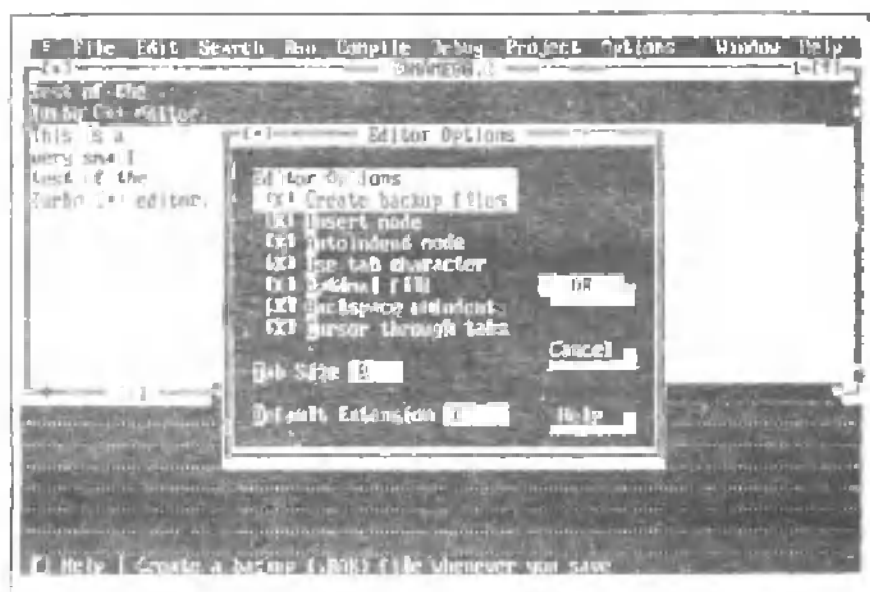


图 3-7 编辑选择项对话框

Creat backup files, Insert mode 和 Autoindent mode 选项是自选择的。若你使用 tab character 选项无效, 则适当的空格数将用来替代制表符。Optimal fill 选项控制 Turbo C++ 在自动缩格时使用什么字符——当有效时, 它融合空格符和制表符; 当无效时, 它仅使用空格符。当 Backspace unindents 有效时, 每次在一空行上按 BACKSPACE, 光标都将退回一个缩格级。若该选项为无效时, 每按 BACKSPACE 时光标后退一个字符。当 Cursor through tabs 有效时, 当你用制表符移动光标时, 其并不是跳到下一个制表位置, 而是每次只移动一个空格; 当该选项无效时, 光标跳到下一个制表位置。

你还可以修改制表符的宽度和缺省文件扩展名。

### 3.18 用文件名激活 Turbo C++

当你激活 Turbo C++ 时, 可以指定要编辑的文件名。还可以在命令行中 “TC” 之后键入文件名来完成这一过程。例如, 在 “TC” 之后输入 MYFILE 将执行 Turbo C++ 并使 MYFILE.C 装入编辑器中。扩展名 .C 将被自动地由 Turbo C++ 添入。若 MYFILE.C 不存在, 其将被建立。若由于某种原因, 不希望在文件名中用扩展名, 在该名之后加上一个句号, 这将使 Turbo C++ 不给文件加 .C 扩展名。

## 第二部分 C 语言

## 第四章 C 语言要素

当开始讨论编程语言时，古老的格言“温故而知新”不再准确。程序设计语言中的每个元素不是孤立存在，而是与其它元素相互联系的。为解决这一个问题，本章列举了很多简单的程序，用于研究和讨论，但不作详细的论述。本章是为初学程序员或者以前从未使用过结构化语言的程序员设计的，它将提供有关 C 如何工作的大致情况。这里介绍的大部分资料将在以后深入讨论，所以如果你已经知道一些有关 C 的知识，就可以跳过本章，直接看第五章。

### 4.1 准备 IDE

在编程之前，必须在集成开发环境中改变一个任选项。正如你以后将学到的，Turbo C++ 在编译程序时，将显示两种信息：错误信息和警告信息。你需要关闭警告信息，原因是 C 有许多特性，本章不可能一一论述，事实上在你理解为什么需要这样的特点以及如何使用它们之前，只知道一点有关 C 的知识就够了。尽管本书的所有程序都是正确的，但在编译本章和后面几章中的程序时，都会显示一个令人烦恼的警告，除非你将它们关闭。警告对于经验丰富的 C 程序员非常有用，但同那些仅仅学习 C 的人没什么关系。

关闭警告信息可按以下过程。首先，确保 Full menus 打开。其次，选择主菜单中的 Option 项，且选择 Compiler 项。然后选定 message 项，按 Tab 键将光条移到 Frequent Errors 框中并选择它。第一个确认框为 Function should return a value。如果这个框中有“X”（应该省略），则按下空格键删除之；如果框内为空，则不需要做什么。最后，按下 ESC 键直至退出所有菜单。

注意，在第九章要求你打开这个选择项之前，一直关闭它。

### 4.2 C 对大小写敏感

理解 C 对大小写敏感是很重要的。它的意思是大小写字母被认为是不同的字符。例如，有些语言中，变量名 count, Count 和 COUNT 是定义相同变量的三种方法。但在 C 语言中，这分别表示了三个不同的变量。所以，当你输入本节列举的程序时，要非常仔细使用正确的大小写。

### 4.3 一个简单的 C 程序

开始执行 Turbo C++ 并输入下面的短程序：

```
#include <stdio.h>

/* Sample program #1. */

main()
{
    int age;

    age = 39;
```



```
    printf("My age is %d\n", age);
}
```

一旦编译完毕,选择 RUN 主菜单选择项和 RUN 子菜单选择项(或按 CTRL-F9) Turbo C++ 就会编译这个程序,与必要的库函数连接(更多的是直接与有关的库进行连接),并执行它。

随着编译的开始, Turbo C++ 打开 compiler/linker 窗口, 允许你监督编译的进展情况。编译过程中, 屏幕与图 4-1 非常相像。编译完成时, 将清屏, 累加换行之后, Turbo C++ 集成开发环境立即重新显示。如果你想再检查执行屏幕, 按 ALT-F5 在两屏幕之间切换。

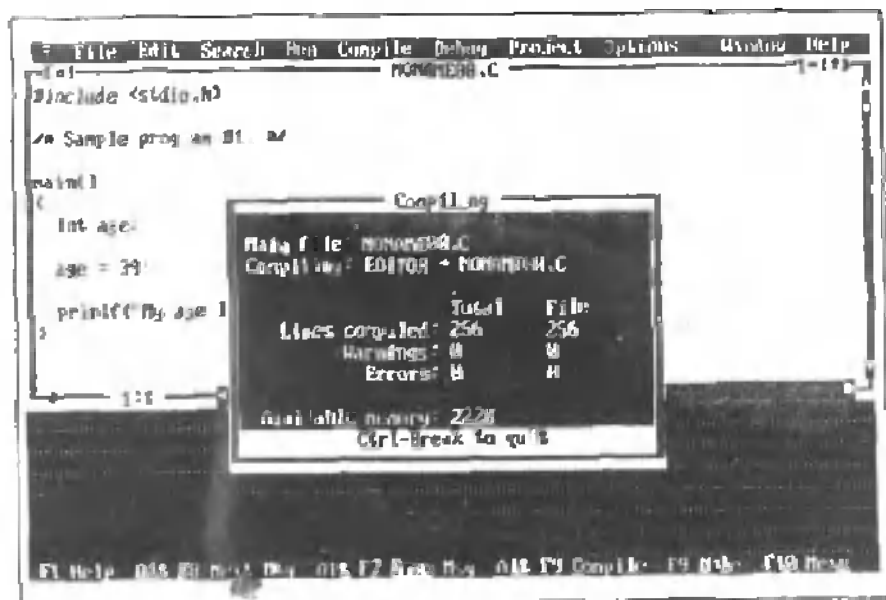


图 4.1 Turbo C++编译窗口

我们来详细讨论这个程序。

#### 4.3.1 进一步讨论

让我们进一步讨论第一个例子程序(#1)的每一行。第 1 行:

```
#include <stdio.h>
```

告知编译程序在编译过程中包含 `stdio.h` 文件。这个文件包含程序所需要的信息, 以保证 C 的标准 I/O 库函数的正确操作。Turbo C++ 支持很多种类型的头文件。有的程序需要一个以上的头文件, 在用户程序中必须包含这些行。我们将在本书的后面部分中讨论头文件和 `#include` 伪指令。

第 2 行:

```
/* sample program #1. */
```

是一个注释。在 C 中, 注释以 `/*` 开始, `*/` 结束。对于在开始和结束注释符号之间的任何语句, 编译程序都将忽略。

如果你仔细看一下例子程序, 将会注意到有一个空行跟在注释行之后。在 C 中, 允许

空行存在，它对程序没有影响。

这一行：

```
main( )
```

指定了一个函数名。所有的 C 程序都是由调用 `main( )` 函数开始执行的。不久你将学到更多的函数。

下一行由单个大括号组成，它标志着 `main( )` 函数的开始。

函数 `main( )` 中的第一行代码是：

```
int age;
```

这一行说明了一个 `age` 变量，告诉编译程序它是一个整数。在 C 中，所有变量必须在使用前进行说明。说明过程包括变量名和类型的说明。在这个语句中，`age` 属于 `int` 型，`int` 是代表整数的关键字。整数是 -32768 到 32768 之间的任何值。

下一行是：

```
age=39;
```

这是一个赋值语句。它把值 39 赋给变量 `age`。注意，C 使用单个等号表示赋值。同时还要注意这个语句以分号结束。C 中的所有语句都以分号结束。

下一行，用来输出信息到屏幕上：

```
printf("My age is %d\n",age);
```

这个语句非常重要，有两个原因。首先，它是一个函数调用的例子。其次，它使用 C 的标准输出函数 `printf( )`。这一行代码由两部分组成：函数名 `printf( )` 和它的两个自变量 `"My age is %d\n"` 和 `age`。因为这是一个 C 语句，所以以分号结束。

在 C 中，没有内部 I/O 指令，而是由 C 的标准库提供函数来完成这些（或其它的）动作。当需要时，程序可以直接调用适当的函数（事实上，Turbo C++ 的标准库函数包括很多有用的函数。）除了库函数，用户程序也可以包括自己编写的函数。这样，调用一个库函数就十分容易：直接写出它的名字，提供所需的所有自变量即可（一个自变量是在函数调用时，传递到该函数中的值。）

`printf( )` 函数是这样工作的。第一个自变量是一个用引号括起来的字符串（有时称为控制串），该字符串可以包含一般字符或以百分号开始的格式代码。一般字符在屏幕上按顺序显示。格式代码告诉 `printf( )` 函数显示的是一个非字符项。如 `%d` 的意思是输出十进制整数。被显示的值会在第二个变量中找到，对于本例来说是 `age`。`\n` 是一个特殊代码，告诉 `printf( )` 函数执行换行操作，在 C 的专有名词中称为“新行”。为弄清一般字符和格式代码之间的关系，改变该行为：

```
printf("My %d age is\n",age);
```

并重新运行该程序。现在显示的内容为 `"My 39 age is."` 字符串中格式命令出现的地方，就是 `printf( )` 函数第二个变量将被打印的位置。不久你将看到，`printf( )` 还要比这个例子所示的功能强得多。

这个程序的最后一行是一个闭大括号，它标志着 `main( )` 函数的结束。当到达了 `main( )` 函数的末尾时，程序执行也随之终止。

#### 4.4 错误处理

使用编辑器，删去用于终止 `"age=39."` 这一行的分号。试编译这个程序。同你想的一

样，产生了错误。这将在信息窗口显示出来。屏幕显示如图 4-2 所示。注意，错误在窗口中是高亮度的，一个块光标指示 Turbo C++ 检查出的错误在程序中的位置。记住，由于编译程序企图弄懂你给出内容的含义，所以它显示检测的错误可能是在下一行，因为在那里，Turbo C++ 最后判定你犯了错误。

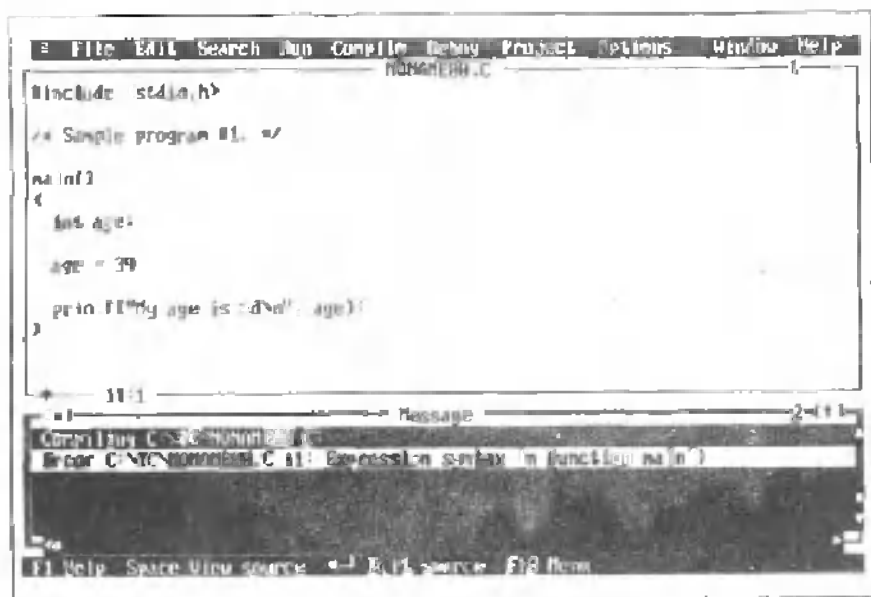


图 4-2 编译出错

一种了解 Turbo C++ 的集成环境的最好办法是，你可以在程序中设置一个错误。按 ALT-F8，可以到达下一个错。按 ALT-F7，可回到上一个错。按 ENTER 激活编辑器。这时就可以修改程序了。

#### 4.5 错误与警告比较

C 语言被设计为一种非常宽宏大量的语言，实际上允许所有语法上正确的语句被编译。然而，有些语句尽管语法上正确，但是值得怀疑的。当 Turbo C++ 遇到这样的情况时，它就会打印一个警告。你，作为一个程序员，随后决定这个疑问是否正确。只要你使用 Turbo C++ 的缺省设置，关闭掉前面讨论的警告信息，本书中的程序就不会产生警告。有时候在有一个真正错误时，其副作用会产生一些警告信息，在继续编写 C 程序时，毫无疑问会遇到几个合法的警告。

除警告信息外，Turbo C++ 中还有一些选择项，在编译时显示有关被显示程序的附加信息。尽管没有专门警告你提防什么，但这个信息还是以警告信息的形式显示出来。在本章开始时处理的那个选择项就属于这一类。当你是一个经验丰富的 C 程序员时，记录程序的信息非常有用，但可能与你正在编写的程序无关。

#### 4.6 第二个程序

尽管第一个例子程序解释了一些有关 C 的重要内容，但它还是相当简单的。第二个例子程序做了些有用的事情：把尺转换为米。同时，还说明了第二个库函数称为 `scanf()`，用

来读取用户从键盘上输进来的信息。现在将下面的程序输入计算机：

```
#include <stdio.h>

/* Sample program #2 - feet to meters. */

main()
{
    int feet;
    float meters;

    printf("Enter number of feet: ");
    scanf("%d", &feet);

    meters = feet * 0.3048; /* feet to meters conversion */
    printf("%d feet is %f meters\n", feet, meters);
}
```

有一些重要的新概念需要介绍。首先，说明了两个变量：`feet` 是一个整数，`meter` 是浮点型，表示可有一个小数部分，这称为一个浮点数。

库函数 `scanf()` 用于从键盘上读取一个整数。第一个自变量中的 `%d` 要求 `scan()` 读取一个整数，将其值赋给后面的那个变量。`feet` 前面的 `&` 对于 `scanf()` 正常工作是十分必要的。在你对 C 如何工作有更多的了解之前，先相信这一点。

接着就将尺数字转换为米。注意尽管 `feet` 是一个整数，但它仍能被浮点数除和将其值赋给一个浮点型变量。与其它高级语言不同，C 允许在一个表达式中使用不同的数据类型。星号和其它可编程语言一样，表示乘法。

通过调用 `printf()` 显示转换结果。正如你看到的，这次 `printf()` 采用了三个参数：控制字符串和变量 `feet`, `meters`。`printf()` 的一般规则是：控制字符串中有多少格式代码，后面就有多少个参数。由于这里有两个格式代码，也就需要两个附加参数。这些参数从左到右按顺序同格式命令相匹配。如果看得仔细些，你就会注意到 `%f` 是用来打印 `meters` 的，而不是 `%d`。这是因为 `printf()` 必须精确知道将要显示字符的数据类型。`%f` 表示后面的值为 `float` 型。

#### 4.6.1 一种变化

例子程序 #2 有一个限制，即只能将所有的尺数字转换成米。更灵活些的程序应该还能将浮点型值转换成米。下面的程序可完成这种转换：

```
#include <stdio.h>

/* Sample program #2, 2nd version - feet to meters. */

main()
{
    float feet, meters; /* make feet a float */
    printf("Enter number of feet: ");
    scanf("%f", &feet); /* read a float */
    meters = feet * 0.3048; /* feet to meters conversion */
}
```

```
    printf("%f feet is %f meters\n", feet, meters);
}
```

正如你看到的，第一种变化是将 `feet` 改为现在的浮点型。注意，你可以使用一个逗号分隔来说明几个相同类型的变量。然后，调用 `scanf()` 语句时用 `%f` 代替 `%d`。这样，该语句就会读取一个浮点变量。（你注意到 `printf()` 和 `scanf()` 的格式代码了吗？它们是相同的。）最后，`printf()` 语句用 `%f` 格式代码显示 `feet`。

#### 4.7 快速复习

在继续学习之前，让我们先回顾一下所学的重要内容。

- 所有 C 程序都必须有一个 `main()` 函数——程序开始执行的部分。
- 所有的变量在使用之前都必须进行说明。
- C 支持不同的数据类型，包括整数、浮点数。
- `printf()` 函数用来将信息输出到屏幕上。
- `scanf()` 函数从键盘上读取数据。
- 到 `main()` 结尾时，程序结束。

#### 4.8 什么是 C 函数？

C 语言是以建立块的概念为基础的。一个块单元称为函数。一个 C 程序就是一个或多个函数的集合。编写一个程序，首先要生成一些函数，然后把它放到一起。

在 C 中，函数是一个子程序，包含一条或多条语句，完成一个或多个任务。在写好的 C 代码中，每个函数完成一个任务。每个函数都有一个名字和一个将要接收的自变量表。通常，除 `main` 以外，你可以随意给一个函数取名（当然，函数名不能与保留字相同。），但 `main` 是专供程序开始执行的函数。

当表示函数时，本节使用一个约定，在写 C 程序时遵守一个标准。在函数名后有一对圆括号。例如，如果一个函数名为 `max`，在文本中使用时应写为 `max()`。这种约定可以帮助你将在本书中的变量名和函数区别开。

你可以用编写 `main()` 函数相同的方法编写其它函数，在程序的其它部分调用它们。

例如，下面这个程序用 `hello()` 函数在屏幕上打印“hello”：

```
#include <stdio.h>

/* A simple program with two functions. */

main()
{
    hello(); /* call the hello function */
}

hello()
{
    printf("hello\n");
}
```

##### 4.8.1 函数和自变量

函数的自变量只是一个函数调用时传递到函数中的值。你已经看到两个函数中有自变

量: `printf()` 和 `scanf()`。你也可以编写有参数的函数。例如, 下面这个程序中的函数 `sqr()` 采用一个整型自变量显示它的平方根:

```
#include <stdio.h>

/* A program that uses a function with an argument. */

main()
{
    int num;

    num = 100;

    sqr(num); /* call sqr() with num */
}

sqr(int x) /* parameter declaration is inside parentheses */
{
    printf("%d squared is %d\n", x, x*x);
}
```

你会看到在 `sqr()` 的说明部分中, 变量 `x` 在函数名之后的括号内进行说明, 它将接收传递给 `sqr()` 的值 (没有自变量的函数不需要任何变量, 所以括号内为空。) 当调用 `sqr()` 时, `num` 的值——这里为 100——传递给 `x`, 这样就会显示 "100 squared is 10000"。你应该输入这个程序, 就会相信的确和预料一样。

保持二者一致性是很重要的。首先, 自变量提供用于调用函数的值。用于接收函数调用中的自变量值的变量称为该函数的形式参数。事实上, 有自变量的函数称为参数函数。重要的是, 用作函数调用的变量与接收其值的形式参数无关。

另一个简单的参数函数如下所示。函数 `mul()` 打印两个整数自变量的乘积。注意用逗号分隔 `mul()` 中的参数。

```
#include <stdio.h>

/* Another example of function arguments. */

main()
{
    mul(10, 11);
}

mul(int a, int b)
{
    printf("%d", a*b);
}
```

记住: 用于调用函数的自变量的类型必须与接收这个自变量的形式参数的类型相同。例如, 你不能调用以两个浮点数为自变量的 `mul()`。(C 能自动进行类型转换, 即这里允许一定的灵活性, 但最好开始就保证自变量的类型匹配参数类型。)

## 4.8.2 返回值的函数

在讨论完函数之前, 有必要稍微谈谈函数返回值的问题。你将使用的很多 C 的库函数都返回一个值。在 C 中, 函数可以用 `return` 关键字返回调用它的程序一个值。为说明这

个问题，前面用于打印两个数的结果的程序可被改写为如下所示的形式。注意把函数放在赋值语句的右边可以把返回值赋给一个变量。

```
#include <stdio.h>

/* A program that uses return. */

main()
{
    int answer;

    answer = mul(10, 11); /* assign return value */

    printf("The answer is %d\n", answer);
}

/* This function returns a value */
mul(int a, int b)
{
    return a*b;
}
```

在这个例子中，mul( )使用 return 语句返回 a\*b 的值。该值随之赋给 answer。这就是说，在调用程序中，return 语句返回的值成为 mul( )的值。

警惕：正如有不同类型的变量一样，也存在着不同类型的返回值。必须确保接收函数返回值的变量与函数返回值类型相同。mul( )程序返回的类型缺省为 int。（不久，你将看到如何返回不同类型的值。）

如果 return 语句不带任何值，就有可能使函数返回未定义值。还有，可以在函数中有一个以上的 return 语句。

### 4.6.3 函数的一般形式

函数的一般形式如下：

```
return-type function-name(parameter list)
{
    body of code
}
```

对于没有参数的函数，不存在参数表。

## 4.9 两个简单命令

为理解后面几章中的例子，有必要弄懂两个最简单形式的 C 命令：if 和 for。在后面几章里，将对它们进行彻底讨论。

### 4.9.1 if 命令

C 的 if 语句的操作规则大致与其它语言中的 IF 语句一样。其最简单的形式为：

```
if(condition) statement;
```

condition 是一个判断真或假的表达式。在 C 中，真为非零值。下面这个程序段在屏幕上打印词组 "10 is less than 11"。

```
if(10<11) printf("10 is less than 11");
```

比较操作符同其它语言中的相类似，如 < 表示小于，>= 表示大于或等于。但在 C 中，相等操作符为 ==。因此，下面的语句不能打印出信息 "hello"。

```
if(10 == 11) printf("hello");
```

#### 4.9.2 for 循环命令

C 中的 for 循环与其它语言，包括 Turbo Pascal 和 BASIC 中的 FOR 循环很相像。其最简单的形式为：

```
for(initialization,condition,increment) statement;
```

其中 initialization 用于给循环控制变量设置初始值。condition 是每次循环重复都要进行测试的表达式。只要它为真（非零），循环将继续执行。increment 部分使循环控制变量递增。例如，下面的程序在屏幕上打印 1 到 100 之间的数字：

```
#include <stdio.h>

/* A program that illustrates the for loop. */

main()
{
    int count;

    for(count=1; count<=100; count++) printf("%2d ", count);
}
```

你可以看到，count 初始化为 1。每次循环重复，都要检测条件 count<=100。如果为真，则执行 printf( ) 语句，且 count 加 1。count 后面的两个加号告诉 C 每次将 count 加 1。当 count 大于 100 时，条件变为假，则循环终止。

#### 4.10 代码块

由于 C 是一个结构化的语言，它支持代码块的生成。一个代码块是一组逻辑上相互联系、可看成为一个单元的语句。在 C 中，将一系列语句放入大括号内就形成了一个代码块。在这个例子中：

```
if(x<10) {
    printf("too low,try again");
    scanf("%d",&x);
}
```

在 if 之后，两个大括号之间的两个语句，如果 x 小于 10，则都将执行。用大括号括起的这两个语句代表一个代码块。它们是一个逻辑单元；其中的一个语句不执行，则另一个语句也不会执行。在 C 中，大多数命令的目标是单个语句或代码块。代码块不仅允许使用众多透明、精巧、高效的算法，而且还帮助程序员认识到程序的本质。

#### 4.11 字符和字符串

C 的另外一个重要数据类型是 char，它表示一个字符。一个字符是一个单字节值，用于表示可打印字符或 0 到 255 范围内的整数。字符常量用两个单引号括起来。例如，下面这个程序在屏幕上打印字母 "ABC"。注意，这里介绍了一个新的 printf( ) 格式代码，用于打印一个单字符。



```

#include <stdio.h>

/* A simple example using characters. */

main()
{
    char ch;

    ch = 'A';
    printf("%c", ch);

    ch = 'B';
    printf("%c", ch);

    ch = 'C';
    printf("%c", ch);
}

```

尽管使用 `scanf()` 可以从键盘上读入一个字符，但更普遍的方法是使用 Turbo C++ 中的库函数 `getche()`。`getche()` 函数一直等待，直到有一个键按下并返回其值。例如，如果你按下 H，下面的程序将打印 "you pressed my magic key"。

```

#include <stdio.h>
#include <conio.h>

main()
{
    char ch;

    ch = getche(); /* read one character from the keyboard */

    if(ch=='h') printf("you pressed my magic key\n");
}

```

这个程序也说明了在 `if` 语句中可以使用字符。

#### 4.11.1 字符串

在 C 中，一个字符串是一个以空字符结尾的字符数组。（在 C 中，空字符本质上与 0 相同。）C 没有字符串类型，但你可以说明一个字符数组，使用在库中可以找到的各种字符串函数来操纵它们。尽管后面将讨论有关数组的内容，但这里先介绍一些基本规则。

在 C 中，数组可以是一维到多维。但本章只集中介绍一维数组。一个一维数组是一个相同类型的变量表。可以把数组的大小用方括号括起来，放在数组名之后，从而生成这样一个数组。下面这个程序段说明了一个叫做 `str`、具有 80 个元素的字符数组。

```
char str[80];
```

为访问一个特定元素，将其下标放在数组名后的中括号内。C 中所有的数组都从 0 开始计算偏移量。因此，`str[0]` 为第一个元素、`str[1]` 为第二个元素、`str[79]` 为第 80 个元素，也是最后一个元素。

最重要的一点是，你要记住 C 中的数组没有边界检测。这意味着如果你不仔细，程序有可能将一个数组“用尽”。现在，最简单的办法通常是使用大的数组，足以容纳你要装入的东西。记住所有的字符串都以空字符结束。所以你的数组比要装入的字符串至少要大一个字符，以便存放空终结符。在 C 中，一个空字符规定为字符常量 `'\0'`。因此，一个数

组若要足够表示单词"hello", 那么它最少要有六个字符长: 五个用来存放字符串, 一个用来存放空终结符。如下所示:

```
h e l l o '\0'
```

为了从键盘上读取一个字符串, 首先要生成一个字符数组来存放字符串, 然后使用库函数 `gets()`。`gets()` 函数使用该字符串的名字作为自变量, 从键盘上读取字符, 直到按下 ENTER 键。ENTER 不被存储, 但被空终止符代替。下面的程序说明了这个规则:

```
#include <stdio.h>

/* A string example. */

main()
{
    char str[80];

    printf("enter your name: "),
    gets(str);

    printf("hello %s", str);
}
```

注意格式代码 `%s` 用于告知 `printf()` 函数打印一个字符串。

#### 4.12 printf(): 快速复习

几乎每个在本书第二部分中执行控制输出的例子都将用到 `printf()` 函数。在前面的程序中你已看到了几个例子。现在正式讨论一下 `printf()`。

`printf()` 的一般形式为:

```
printf("control string", argument list)
```

在 `printf()` 函数中, 控制字符串可以包含将在屏幕上显示的字符串, 或包含规定如何显示自变量的剩余部分的格式代码, 或者二者都包含。下面是你迄今为止所学过的格式代码:

代码	含义
<code>%d</code>	显示一个十进制整数
<code>%f</code>	显示一个十进制浮点数
<code>%c</code>	显示一个字符
<code>%s</code>	显示一个字符串

以后还将介绍其它一些格式代码。

格式控制命令可以嵌套在控制字符串的任何地方。调用 `printf()` 时, 将搜索控制字符串。遇到一个格式代码时, `printf()` 将记住它并在打印相应自变量时使用。格式代码和自变量自左至右相互匹配。控制字符串中的格式代码的个数告知 `printf()` 随后将有多少个自变量。

下面的例子显示了 `printf()` 函数是如何工作的:

```
printf("%s %d", "this is a string", 100);
```

显示:

```
this is a string 100
```

```
printf("this is a string %d",100);
```

显示:

```
this is a string 100
```

```
printf("number %d is decimal,%f is float.",10,110.789);
```

显示:

```
number 10 is decimal, 110.789 is float.
```

```
printf("%s", "HELLO/n");
```

显示:

```
HELLO
```

你必须使自变量数与控制字符串中的格式代码数相同。否则,屏幕上将显示杂乱无用的信息,有用信息将不显示。

#### 4.13 scanf()快速回顾

scanf()函数是一个C的输入函数。尽管它可用来从键盘上有效读取任何类型的数据,但通常用来输入整数和浮点数。scanf()的一般形式为:

```
scanf("control string",argument list);
```

现在,假设控制字符串只包含格式代码。(实际上,在后面详细研究scanf()之前,除了格式代码,你不要在控制字符串中放进任何字符,否则会搞混。)你需要的两个代码是%d和%f,告诉scanf()分别读取整数和浮点数。自变量表中的自变量数必须与控制字符串中的格式代码数完全一致。如果不是这样,那么任何事都有可能发生——包括程序遭到毁坏。在调用scanf()返回之后,控制字符串后面的变量将包含你从键盘上输入的值。

从键盘接收数值的变量在自变量表中必须加以前缀&。现在解释其必要性是很复杂的,只能说它使scanf()将一个值赋给它的自变量。

#### 4.14 分号、括号和注释

可能你会奇怪为什么这么多的语句都以分号结尾。在C中,分号是语句终结符。这就是说,每个单独语句都必须以分号结尾。它暗示着一个逻辑单元的结束(对于懂Pascal的读者,要小心使用。Pascal中的分号是语句分隔符;C中的分号是语句终结符)。

在C中,块是相互联系的语句的逻辑集合,包含在开花括号和闭花括号之内。如果你认为一个块是一组语句,它的意义是块不以分号结束。

C不将行末认做一个终止符。这意味着对语句的位置没有限制。这便于将语句分组或分离,以使视觉清晰,例如下面两个程序段是等效的:

```
x=y;
```

```
y=y+1;
```

```
mul(x,y);
```

这等同于:

```
x=y;
```

```
y=y+1;
```

```
mul(x,y);
```

C 中的注释可以放在程序中任何地方,用标号括起来。开始注释的标号为/\*,结束的标号为\*/。在 ANSI 标准 C 中,注释不能嵌套。例如,下列注释中的注释将产生编译时间错:

```
/*this is /*an error*/*/
```

Turbo C++ 确实有一个允许嵌套注释的选择项,但使用它会使你的代码变得复杂。

#### 4.15 缩排练习

在前面的例子中你会注意到,有些语句被缩排了。由于 C 不关心你将一行中相互关联的语句放在什么位置,所以可以随意安排程序的格式。但是,经过这么多年,已经对可读程序形成了一种共同的可接受的缩排格式。本书将遵照这种格式并建议你也使用它。使用这种格式,将在每个开花括号后缩进一些位置,每个闭花括号都退回一些。有某些语句需要另外一些缩排格式,这些将在后面讲述。

有时,在特别复杂的程序中,缩排很大以致于代码行绕回到下一行。为避免这种现象,你可以将一个语句分为两个部分,将它们放在分开的两行中。例如,下面是个完全正确的语句:

```
count=10*unit/
```

```
amount_left;
```

通常,你可以在放置空格的地方断开一行。只是有必要时才断开某些行,然而不管怎样,它都会干扰别人读懂这段程序。

#### 4.16 C 库

本章经常提到 C 库和库函数。所有的 C 编译程序都有一个库,以提供完成最一般需要的函数。Turbo C++ 的设计者实现了一个超出 C 的 ANSI 标准中定义的库。它包含大部分你将使用的完成一般目标的函数。你应该看看《Turbo C++ 库函数参考》这一部分,它说明了这些函数。本书将尽可能地介绍库函数。附录 A 讨论了最重要的几个函数。

当使用一个不是自己编写的程序中的一部分函数时,编辑程序会记住它的名字。当连接程序连接时,将找到这个缺少的函数并把它加到目标代码中。存在于库中的函数处于可调用状态,这意味着对于不同的机器代码指令,内存地址并没有绝对规定,而是只存储了偏移量用来形成实际地址。有许多技术手册和书都详细解释了这个过程。总之,在 C 中编程,你不需要对实际寻址过程做任何深入的解释。

#### 4.17 C 的关键字

同其它可编程语言一样,C 也是由关键字和支持关键字的语法规则组成。一个关键字本质上是一个命令,在很大程度上可以说,一种语言的关键字规定了做什么和怎样做。

Turbo C++支持所有由ANSI C标准规定的关键字集合。如表4-1所示。

表4-1 ANSI C标准关键字

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Turbo C++有16个附加关键字,用来更好地运用8088/8086处理机的内存组织,其中3个支持内部编程语言和中断。扩展的关键字如表4-2所示。

表4-2 扩展关键字

asm	cdecl
_cs	_ds
_es	_export
far	huge
interrupt	_loadds
near	pascal
_regparam	_saveregs
_seg	_ss

C的所有关键字都是小写。前面已说明过,C中的大小写是不同的;因此,else是一个关键字,ELSE则不是。在一个Turbo C++程序中,关键字不能用作其它用途。例如,它不能用作变量名。

#### 4.18 术语复习

在继续下面的内容之前,你应该复习一下这些术语:

- 源代码:用户可以读懂的程序内容;一般指“程序”。
- 目标代码:将程序源代码翻译成机器代码,即计算机可以识别并直接执行的代码。
- 连接程序:将分开的编译过函数连接一起的程序;用来将标准C库中的函数与你写的程序连接起来。
- 库:一个标准函数的集合,可由用户程序使用。这些函数包括所有I/O操作和其它有用程序。
- 编译时间:当用户程序被编译时所发生的事情。一般编译时间发生的事情为系统错误。
- 运行时间:当用户程序执行时所发生的事情。

## 第五章 变量、常量、操作符和表达式

变量和常量由操作符操作而组成表达式。这些是 C 语言的基础。如果你想更深入地学习 C 语言，必须先弄懂本章所陈述的概念。不同于其它计算机语言，尤其是 BASIC 具有非常简单的变量、操作符和表达式，C 语言赋予这些元素更大的功能和重要性。尽管你迫切希望直接接触到这个语言的实质部分，但不要这样做，因为本章陈述了一些非常重要的规则。

### 5.1 标识符名称

C 语言规定了一些名称用来将变量、函数、标号和各种由用户定义的事物表示为标识符。一个 C 中的标识符可由一个或几个字符组成。第一个字符必须是字母或下划线。后面的字符可以是字母、数字或下划线，下面举例说明一些正确的和不正确的标识符名称：

<u>正确的</u>	<u>不正确的</u>
count	count
test23	hi!there
high_balance	high..balance

在 Turbo C 中，标识符名的前 32 个字符很重要。这意味着如果两个变量的前 32 个字符相同，而只是第 33 个字符不同，Turbo C 不能将它们区别开。例如，这两个标识符：

```
this_is_a_very_long_name_used_as_an_example
this_is_a_very_long_name_used_as_an_example_too
```

在 Turbo C 中将这样表示：

```
this_is_a_very_long_name_used_as
```

但在 Turbo C++ 中，标识符可为任意长度。

你必须牢记，在 C 语言中，大写和小写字符作为不同字符处理。这样，count, Count 和 COUNT 是三个不同的标识符。

一个标识符不能与一个键值相同，也不能与一个函数名相同——你编写的或在 C 库中的。

### 5.2 数据类型

正如你在第四章所看到的，C 中的所有变量必须在使用之前定义。这很有必要，因为编译程序在编译使用该变量的所有语句之前，必须先知道这个变量的类型。C 语言中有五种基本数据类型：字符型、整型、浮点指针型、双浮点指针型和（有点与众不同）无值型。用来说明这些变量类型的键值分别为 char, int, float, double 和 void。IBM PC 中的 Turbo C++，每个数据类型的大小和范围如表 5-1 所示。

char 类型的变量可表示 8 位 ASCII 字符如 'A', 'B', 'C' 或其它任意 8 位值。int 类型的变量可以为任意不带小数部分的整数值。这个类型的变量常用于控制循环和条件语句。float 和 double 类型的变量在需要带小数部分或应用程序需要较大或较小的情况下使用。float 和 double 型变量的区别在于它们表示的最大（最小）数值。如表 5-1 所示，一个 double

表 5-1 Turbo C++ 的基本数据类型的大小和范围

类型	位宽	范围
char	8	-128~127
int	16	-32768~32767
float	32	3.4E-38~3.4E+38
double	64	1.7E-308~1.7E+308
void	0	无值

型可表示大于 float 型很多倍的数值。void 类型的用途将在本书后面章节中论述。

### 5.2.1 类型修饰符

由于包含 void 类型的概念，基本数据类型可以在其前面包含各种各样的修饰符。修饰符用于改变基本类型的含义，以便更精确地适应各种需要。修饰符列表如下：

signed

unsigned

long

short

修饰符 signed, unsigned, long 和 short 可与字符型和整型配合使用。但 long 只能与 double 一起使用。表 5-2 列出了允许的基本类型和修饰符的组合。

表 5-2 Turbo C++ 的基本类型和变址数的所有可能组合

类型	位宽	范围
char	8	-128~127
unsigned char	8	0~255
signed char	8	-128~127
int	16	-32768~32767
unsigned int	16	0~65535
signed int	16	-32768~32767
short int	16	-32768~32767
unsigned short int	16	0~65535
signed short int	16	-32768~32767
long int	32	-2147483648~2147483647
signed long int	32	-2147483648~2147483647
unsigned long int	32	0~4294967295
float	32	3.4E-38~3.4E+38

double	64	1.7E-308~1.7E+308
long double	80	3.4E-4932~1.1E+4932

尽管允许，但整数中的 `signed` 是多余的，因为缺省的整数说明假定为一个有符号整数。

有符号和无符号整数的区别在于对高位的不同解释。如果为有符号整数，Turbo C 编译程序将产生表示该整数的高位为符号标志位的代码。如果该标志位为 0，则这个数为正数；如果为 1，则为负数。负数可由 2 的补码表示。在这个方法中，数值的所有位（不包括符号标志位）取反且加 1，最后，符号标志位置 1。

有符号整数对于很多算法非常重要，但是它们只有相应无符号数的绝对范围的一半。例如，一个以二进制表示的数 32, 767:

```
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

如果最高位设置为 1，该值将被解释为 -1（假定为 2 的补码形式）。但如果定义该数为 `unsigned int`，当高位设置为 1 时，该数变为 65, 535。

为理解对有符号和无符号整数的解释方式的不同，请运行下列小程序：

```
#include <stdio.h>

/* Show the difference between signed and unsigned
   integers.
*/
main()
{
    int i;    /* a signed integer */
    unsigned int j; /* an unsigned integer */

    j = 60000;
    i = j;
    printf("%d %u", i, j);
}
```

当程序运行时，输出结果为 -5536 60000。原因是表示 60000 为一个无符号整数的位模式被有符号整数解释为 -5536。你知道，`%d` 为另一种形式，表示 `printf()` 将要显示的是一个无符号整数值。

C 语言允许一个说明 `unsigned`、`short` 或 `long` 整数的简写形式，可只用 `unsigned`、`short` 或 `long` 而不必加上 `int`，`int` 被隐含。例如：

```
unsigned X;
unsigned int Y;
```

两种方式都说明了无符号的整数变量。

`char` 类型的变量可用来表示除 ASCII 字符集以外的值，也可用于表示 -128~127 之间的“小”整数，当某一位置不需要大数值整数的时候，可用 `char` 变量代替这个整数。例如，下面的程序使用一个 `char` 变量控制在屏幕上打印字母的循环：

```
/* This program prints the alphabet. */
#include <stdio.h>
```



```

main()
{
    char letter;
    for(letter = 'A'; letter <= 'Z'; letter++)
        printf("%c ", letter);
}

```

如果对你来说 `for` 还很陌生, 请记住在计算机中字符'A'代表一个数值, 而且 A~Z 的值按升序排列。

### 5.3 变量说明

变量说明语句的一般形式如下:

```
type variable_list;
```

这里, `type` 必须是一个正确的 C 数据类型, `variable_list` 可包含一个或多个用逗号分开的标识符名字。下面举一些例子:

```

int i, j, k;
short int si;
unsigned int ui;
double balance, profit, loss;

```

同其它计算机语言不同, 在 C 中, 一个变量的名字与其类型无关。

#### 5.3.1 变量说明的位置

变量说明的位置在很大程度上影响到程序中其它部分对该变量的使用。一个变量的使用规则基于它被定义的位置, 这在语言中称作“范围规则”。有关这些规则和细节的详细讨论将在以后当你了解 C 语言有些了解之后再讨论。现在只讨论一个基本知识。

在一个 C 程序中有三个地方可以说明变量。一个是在所有函数, 也包括 `main()` 函数之外。这种变量称为全局变量, 可在程序中的任何部分使用。另一个说明变量的地方是在函数中。这种方式说明的变量称为局部变量且只能用于该函数的语句中。在本质上, 一个局部变量只对该函数的代码已知而在该函数外未知。

还有一个说明变量的地方是在一个函数的形式参数说明部分(回忆一下第四章, 当一个函数被调用时, 形式参数用于接收参数值)。除了执行接收传递给函数的信息以外, 这些参数具有同其它局部变量相同的功能。图表 5-1 列出了一个小程序, 该程序在上述地方说明变量并产生以下结果:

```

.....the current sum is 0
.....the current sum is 1
.....the current sum is 3
.....the current sum is 6
.....the current sum is 10
.....the current sum is 15
.....the current sum is 21
.....the current sum is 28
.....the current sum is 36
.....the current sum is 45

```

```

/* Sum the numbers 0 through 9. */
#include <stdio.h>
int sum; Global variable

main()
{
    int count; Local variable

    sum = 0; /* initialize */
    for(count=0; count<10; count++) {
        total(count);
        display();
    }

    /* add to running total */
    total(int x) Formal parameter
    {
        sum = x + sum;
    }
    display()

    {
        int count; Local variable
        /* this count is different from
           the one in main()
           */
        for(count=0; count<10; count++) printf(".");
        printf("the current sum is %d\n", sum);
    }
}

```

图 5.1 使用全局和局部变量

你可看到，全局变量可由程序中的任何一个函数使用。然而，main()中的局部变量count不能被total()函数使用，必须以参数值传递。这很有必要，因为一个局部变量只能由说明它的函数中的代码使用。最后，注意display()中的count与main()中的count完全不同。因为一个局部变量只对说明它的函数已知，C处理main()中的count完全不同于display()中的count。

有关变量，有两点必须弄清。第一，两个全局变量不能有相同的名字。如果相同，编译程序将不知道使用的是哪一个变量。说明两个相同名字的全局变量将导致一个错误信息。第二，在没有冲突的条件下，一个函数中的局部变量可与另一个函数中的局部变量相同。原因是一个函数中的代码和数据与另一个函数中的代码和数据完全不同。简单地说，一个函数中的语句与另一个函数中的语句毫无联系。当然，同一个函数中的两个局部变量名不能相同。在后面的一些章节，当你对函数有更多的了解之后，这些基本概念将被扩充和细化。

#### 5.4 常量

在C语言中，常量是指不能被程序改变的固定值。常量及其用法多半都很直观，它们已由前面的程序例子以各种各样的形式使用，但须给常量以正式定义。

常量可以是基本数据类型的任何一种。每个常量的表示形式取决于它的类型。字符常量以引号括起来，例如，'a'和'%均是字符常量。整数常量定义为不带小数部分的数值，例如，10 和-100 是整数常量。浮点常数可用于表示带有小数点及小数部分的数值，如 11.123 是一个浮点常数。举例如下：

<u>数据类型</u>	<u>常量举例</u>
char	'a' '\n' '9'
int	1 123 21000 -234
long int	35000 -34
short int	10 -12 90
unsigned int	10000 987 40000
float	123.23 4.33e-3
double	123.23 1231233 -0.9876324

#### 5.4.1 十六进制和八进制常量

你大概知道，在编程时基于 8 或 16 进制的数据系统往往比 10 进制的简单。基于 8 的数据系统称为八进制，使用 0~7 数字。在八进制中数字 10 等同于十进制中的 8。基于 16 的数据系统称为十六进制，使用数字 0~9 再加上字母 A~F，分别表示 10，11，12，13，14 和 15。例如，十六进制数字 10 在十进制中表示 16。因为使用了这两种数据系统，C 允许你使用十六进制或八进制替代十进制定义整数常量。一个十六进制常数必须以\*0x\*开始（0 后面跟 x），后面是十六进制形式的常数。八进制常数以一个 0 开始。举一些例子：

```
hex=0xFF; /* 255 in decimal */
oct=011; /* 9 in decimal */
```

#### 5.4.2 字符串常量

C 支持除了上述数据类型以外的另一个类型：字符串。一个字符串是一个以双引号括起来的字符集。如"this is a test"是一个字符串。在程序例子中的 printf()语句中，你已见到了一些字符串的例子。

注意不能将字符与字符串混淆。单字符常量是以单引号括起来的，如'a'。但'a"是一个只包含一个字母的字符串。

#### 5.4.3 转义字符常量

多数打印字符都需用单引号将所有字符常量括起来，但也有一些，比如回车键，不可能将其键入一个字符串中。因此，C 提供了一些特定的转义字符常量。表 5-3 列出了这些代码。

你可以像使用其它字符一样使用转义字符。例如：

```
ch='r';
printf("this is a test\n");
```

这个代码段首先将制表符赋予 ch，然后在屏幕下一行打印"this is a test"。在本书稍后的部分你将阅读到更多有关转义字符的例子。

表 5-3 转义字符

<i>Code</i>	<i>Meaning</i>
<code>\b</code>	Backspace
<code>\f</code>	Form feed
<code>\n</code>	Newline
<code>\r</code>	Carriage return
<code>\t</code>	Horizontal tab
<code>\"</code>	Double quote
<code>\'</code>	Single quote character
<code>\0</code>	Null
<code>\\</code>	Backslash
<code>\v</code>	Vertical tab
<code>\a</code>	Bell (alert)
<code>\N</code>	Octal constant (where <i>N</i> is an octal constant)
<code>\xN</code>	Hexadecimal constant (where <i>N</i> is a hexadecimal constant)

#### 5.4.4 变量初始化

在 C 语言中，可以在变量说明部分将一个符号和常数值置于变量名后从而给变量赋值。初始化的一般形式是：

```
type variable_name = constant;
```

给出一些例子：

```
char ch = 'a';
```

```
int first = 0;
```

```
float balance = 123.23;
```

全局变量只能在程序的开始进行初始化。局部变量在程序每次进入定义它的函数时均需初始化。所有的全局变量如果没有初始化，则自动初始化为 0。未经初始化的局部变量在赋值之前无确定值。

初始化变量的主要优点是减少了程序中的代码数量。举一个变量初始化的简单例子，下面是图 5-1 列出的运行程序的改写本，这个改写本要求你输入一个数字，然后计算从 1 到输入数字之间的总和：

```
/* An example using variable initialization. */
#include <stdio.h>
main()
{
    int t;

    printf("enter a number: ");
    scanf("%d", &t);
    total(t);
}
```

```

total(int x)
{
    int sum=0, i, count;

    for(i=0; i<x; i++) {
        sum = sum + i;
        for(count=0; count<10; count++) printf(".");
        printf("the current sum is %d\n", sum);
    }
}

```

你将在第三部分中学到，在 C++ 中，已经对变量初始化加以扩展。

## 5.5 操作符

C 语言有着丰富的内部操作符。操作符是一个表示编译程序执行特定的数学或逻辑操作的符号。C 有三个一般类型的操作符：算术、关系和逻辑，以及逐位操作符，此外，C 还有一些完成特定功能的特殊操作符。本章只讲述算术、关系与逻辑以及赋值操作符。它将在以后讨论。

### 5.5.1 算术操作符

表 5-4 列出了 C 语言中的算术操作符。C 中的 +, -, \* 和 / 操作符与其它计算机语言中的相同。这些操作符适用于任何一种 C 语言允许的内部数据类型。当 / 用于一个整数或字符时，余数将被截掉；例如，在整数除法中，10/3 将等于 3。

表 5-4 算术操作符

<i>Operator</i>	<i>Action</i>
-	Subtraction, also unary minus
+	Addition
*	Multiplication
/	Division
%	Modulus division
--	Decrement
++	Increment

取模操作符%，其结果为整数除法的余数。照此而论，%不能用于 float 或 double 型。下面的程序计算用户输入整数的商和余数。

```

#include <stdio.h>

main()
{
    int x, y;

    printf("enter dividend and divisor: ");
    scanf("%d%d", &x, &y);
}

```

```
printf("quotient %d\n", x/y);
printf("remainder %d ", x%y);
```

单目减法实际上是将其操作数乘以 -1, 也就是, 任何一个前面带减号的数均转换了它的符号。

### 5.5.2 增量和减量

C 允许两个非常有用的操作符, 这两个操作符在其它计算机语言中不存在。即增量和减量操作符: ++ 和 --。操作符 ++ 将其操作数加 1, -- 减 1。因此, 下面的操作:

```
x++;
```

```
x--;
```

等同于:

```
x=x+1;
```

```
x=x-1;
```

增量和减量操作符可用于操作数的前面和后面。例如:

```
x=x+1;
```

可写为:

```
++x;
```

或

```
x++;
```

但是, 当用于表达式时, 这两种方式是有区别的。当增量和减量操作符在操作数前面时, C 将先执行增量或减量, 然后再使用操作数的数值。如果操作符在操作数的后面, 则 C 将先使用操作数的数值然后再对它增量或减量。请看这个例子:

```
x=10;
```

```
y=++x;
```

在这种情况下, y 将被设置为 11, 因为 x 先被增加 1 然后才赋给 y。但如果代码写成:

```
x=10;
```

```
y=x++;
```

y 将被设置为 10, 而后 x 增加 1。在两种方式下, x 均被设置为 11, 区别在于何时增加。这样主要的优点是便于控制何时进行增量和减量, 关于这一点你将在后面的章节中看到。

算术运算符的优先次序为

高优先级      ++ -- -(单目)

低优先级      \* / % +

同等优先权的操作符由编译程序从左到右计算。当然, 可用圆括号改变执行顺序。C 处理圆括号实际上同其它计算机语言相同, 它们将一个操作或操作集变成一个高优先级的操作。

### 5.5.3 关系和逻辑操作符

本节介绍关系操作符和逻辑操作符, 关系是指相互间的关系值, 逻辑是指这些关系被

连接的方式。关于关系和逻辑操作符概念的关键是 `true` 和 `false` 的含义。在 C 中，“真”是指除零以外的任意值，“假”是零值。使用关系或逻辑操作符的表达式将返回 0 表示假，返回 1 表示真。表 5-5 列出了所有的关系和逻辑操作符。

表 5-5 关系和逻辑操作符

<i>Relational Operators</i>	
<i>Operator</i>	<i>Action</i>
<code>&gt;</code>	Greater than
<code>&gt;=</code>	Greater than or equal
<code>&lt;</code>	Less than
<code>&lt;=</code>	Less than or equal
<code>=</code>	Equal
<code>!=</code>	Not equal
<i>Logical Operators</i>	
<code>&amp;&amp;</code>	AND
<code>  </code>	OR
<code>!</code>	NOT

关系操作符用于确定一个值和另一个值之间的关系。它们常返回一个依赖于测试结果的 1 或 0。下面的程序说明了每个操作的结果并显示是 0 还是 1：

```
/* This program illustrates the relational operators. */
#include <stdio.h>

main()
{
    int i, j;

    printf("enter two numbers: ");
    scanf("%d%d", &i, &j);

    printf("%d == %d is %d\n", i, j, i==j);
    printf("%d != %d is %d\n", i, j, i!=j);
    printf("%d <= %d is %d\n", i, j, i<=j);
    printf("%d >= %d is %d\n", i, j, i>=j);
    printf("%d < %d is %d\n", i, j, i<j);
    printf("%d > %d is %d\n", i, j, i>j);
}
```

你可以输入这个程序，并利用不同的数值组合进行测试。

关系操作符适用于任何一种基本数据类型。例如，下一代码段显示信息“greater than”，因为在 ASCII 排序中，“B”大于“A”。

```
ch1='A';
ch2='B';
if (ch2>ch1) printf("greater than");
```

在本书后面章节中，你将阅读到更多有关关系操作符应用的内容。

逻辑操作符按下面的真值表，支持 AND，OR，NOT 这些基本的逻辑操作。该真值表用 1 代表真，0 代表假。

p	q	p AND q	p OR q	NOT p
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

下面的程序说明了逻辑操作符的操作：

```
/* This program illustrates the logical operators. */
#include <stdio.h>

main()
{
    int i, j;

    printf("enter two numbers (each being either 0 or 1): ");
    scanf("%d%d", &i, &j);

    printf("i AND j is %d\n", i, j, i & j);
    printf("i OR j is %d\n", i, j, i || j);
    printf("NOT i is %d\n", i, !i);
}
```

你可以输入这个程序，并输入真和假的不同组合值进行测试，直到熟悉这些操作为止。

关系操作符和逻辑操作符的优先级均比算术操作符的优先级低。这意味着表达式  $10 > 1 + 12$  的执行将等同于  $10 > (1 + 12)$ ，结果当然为假。

允许在一个表达式中组合使用一些操作，例如：

$10 > 5 \ \&\& \ !(10 < 9) \ || \ 3 <= 4$

执行结果为真。

下表列出了关系的逻辑操作符的相对优先级：

```
高优先级    !
              >  >=  <  <=
              ==  !=
              &&
低优先级    ||
```

同算术表达式一样，在一个关系或逻辑表达式中，允许使用圆括号改变执行的自然顺序。例如：

$1 \ \&\& \ !0 \ || \ 1$

结果为真，因为将首先执行  $!$ ，使 AND 为真。但如果相同的表达式按如下方式加上圆括号，结果为假：

$1 \ \&\& \ !(0 \ || \ 1)$

因为  $(1 \ || \ 0)$  为真，NOT 将其变为假，最后将使 AND 为假。



记住所有的关系和逻辑表达式产生的结果或为 0 或为 1。所以下面的程序不仅正确，并且在显示器上显示数字 1：

```
#include <stdio.h>

main()
{
    int x;

    x = 100;
    printf("Zd", x>10);
}
```

关系和逻辑操作符用于支持包括所有循环和 if 语句的程序控制语句。例如，下面的程序使用一个 if 语句来打印 1 至 100 中的所有偶数：

```
/* Print the even numbers between 1 and 100. */

#include <stdio.h>

main()
{
    int i;
    for(i=1; i<=100; i++)
        if(!(i%2)) printf("Zd ",i);
}
```

在这个例子中，当一个数为偶数时，求模操作将产生一个 0（假）结果，而后该结果被 NOT 取反。

在后面的章节中，你将阅读到更多有关关系和逻辑操作符的例子。

#### 5.5.4 赋值操作符

在 C 中，赋值操作符是一个等号。不同于其它计算机语言，C 允许将赋值操作符用于那些包含关系或逻辑操作符的表达式中。例如，注意下面程序中的 if 语句：

```
#include <stdio.h>

main()
{
    int x, y, product;

    printf("enter two numbers: ");
    scanf("Zd", &x, &y);

    if( (product=x*y) < 0 )
        printf("one number is negative\n");
    else
        printf("positive product is: Zd", product);
}
```

注意 if 语句中的表达式。首先，将  $X*Y$  的值赋给 product。然后，被括起来的赋值表达式与 0 进行比较。这个代码是非常有用的。实际上，这种类型的表达式在专门编写的 C 代码中非常普遍。让我们仔细分析一下它是怎样工作和为什么能这样工作的。

在 C 语言中，赋值操作符可用来做两件事。第一，它将右边的值赋给左边的变量，但当它用于一个较长的表达式时，该赋值操作符将计算右边表达式的结果。因此表达式

中的(`product=X*Y`)部分除了返回那个值以外, 还将 `X*Y` 的值赋给 `product`, 然后在 `if` 语句中这个值与 0 进行比较。圆括号很有必要, 因为赋值操作符的优先级低于关系操作符的优先级。

## 5.6 表达式

操作符、常量和变量是表达式的组成部分。一个 C 中的表达式是这些元素的任意有效组合。因为大多数的表达式趋向于符合代数的一般规则, 所以它们不无一般性。但是, 其中 C 语言的表达式有一些特有方面, 下面将给以讨论。

### 5.6.1 表达式中的类型转换

当一个表达式中包含有不同类型的常量和变量时, 它们将被转换成同一类型。C 编译器将所有操作数转换为最大操作数的类型。这个转换是逐个操作进行的, 按照下面的类型转换规则执行:

1. 所有的 `char` 和 `short int` 转换为 `int`, 所有的 `float` 转换为 `double`。
2. 对于所有的操作数对, 如果一个操作数为 `long double`, 另一个将转换为 `long double`。如果一个操作数为 `double`, 另一个将转换为 `double`。如果一个为 `long`, 加一个将转换为 `long`。如果一个为 `unsigned`, 另一个将转换为 `unsigned`。

如果采用了这些转换规则, 每个操作数对将为同一类型, 操作的结果将与两个操作数的类型一致。请注意规则 2 有一些条件, 即必须按顺序使用。

例如, 思考一下图 5-2 中的类型转换。首先, 字符 `ch` 转换为一个整数, `float f` 转换为 `double`, 然后 `ch/I` 的结果转换为 `double`, 因为 `f*d` 为 `double` 型。最后的结果为 `double` 型, 因为现在两个操作数均为 `double` 型。

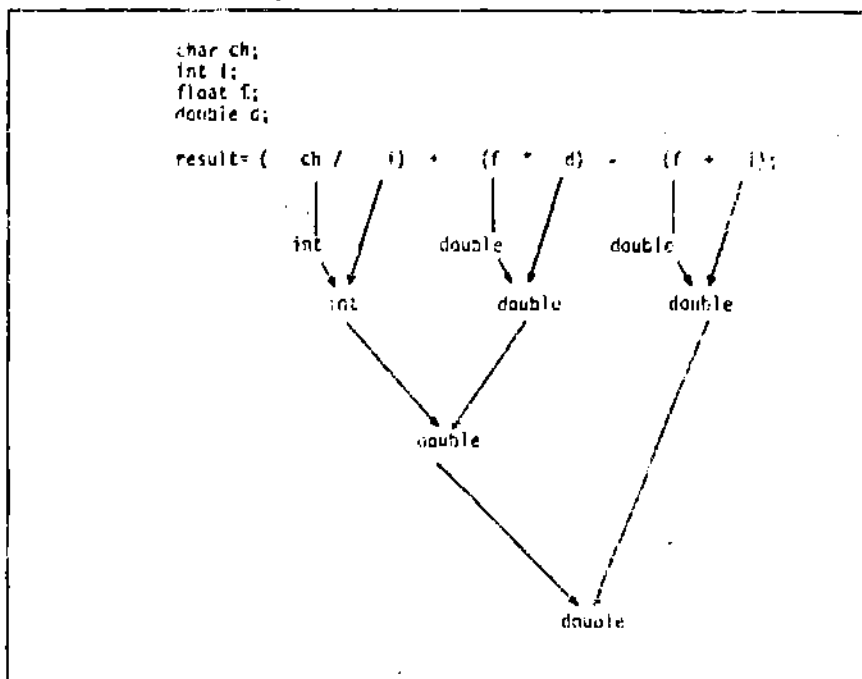


图 5-2 类型转换示例

## 强制转换结构

使用一个叫做强制转换的结构可以将一个表达式强制转换为一个特定类型。一个强制转换结构的一般形式是：

`(type) expression`

`type` 是 C 的标准数据类型之一。例如，如果 `X` 为一个整数，你想将 `X/2` 的执行结果变为 `float` 型，从而保证得到一个小数部分，可以这样写：

`(float) X/2`

这里，强制转换 `(float)` 与 `X` 相联系，使 `2` 转换为 `float` 型并使结果为 `float` 型。但必须注意，如果你写成如下形式，计算结果将没有小数部分：

`(float)(X/2)`

在这种情况下，执行的是整数除法，其结果将被转换为 `float` 型。

强制转换通常被认为是操作符。作为一个操作符，它是一元的并具有与其它单目操作符相同的优先级。

在很多情况下，强制转换都是非常有用的。假设有想用一个整数进行循环控制，而且有关的一些计算需要有小数部分，如在下面的程序中：

```
#include <stdio.h>

main() /* print i and i/3 with fractions */
{
    int i;

    for(i=1; i<=100; ++i )
        printf("%d / 2 is: %f\n", i, (float) i/3);
}
```

如果没有强制转换 `(float)`，那么只能执行整数除法计算，但强制转换保证了计算结果的小数部分将被显示在屏幕上。

### 5.6.2 空格和圆括号

在一个表达式中，可以随意加入一些空格使其易读。例如，下面的两个相同的表达式：

`X=645/(num_entry)-Y*(3127/balance);`

`X=645/(num_entry)-Y*(3127/balance);`

使用多余的或附加的圆括号不会产生错误或减慢执行速度。为了你和以后阅读程序的人，应该使用一些圆括号以明确执行顺序。例如，下面的两个表达式哪个更易读？

`X=Y/3-34*temp-127;`

`X=(Y/3)-(34*temp)-127;`

## 第六章 程序控制语句

在某种意义上说,程序控制语句是任何一种计算机语言的精髓。因为它们控制了整个程序执行的流程。它们各种各样的执行方式确定了语言的风格。C 语言的程序控制语句既丰富又强有力,这也是 C 受人欢迎的原因之一。

程序控制语句可分为三种类型。一种是由条件指令 `if` 和 `switch` 组成。一种是循环控制语句 `while`, `for` 和 `do_while`。还有一种是无条件转移指令 `goto`。

记住一个语句可由下列任意一种组成:一个单语句;一个语句块;或什么也没有,称为一个空语句。这里所描述的语句是指上述的所有可能。

### 6.1 if 语句

尽管你已在第四章学习过有关 `if` 语句的简单介绍,但现在需要进一步讨论之。

`if` 语句的一般形式是:

```
if (condition) statement;
else statement;
```

`else` 语句是任选的。如果条件执行结果为真(不等于 0 的任何值),则组成 `if` 目标的语句或块将被执行,否则,如果有 `else` 语句,组成 `else` 目标的语句或块将被执行。记住只有与 `if` 相关联的代码或者与 `else` 相关联的代码才被执行,但不可能二者同时执行。`if` 和 `else` 的目标语句可以是单语句或语句块。

为了说明 `if` 是如何起作用的,我们将研究一个执行数据基制转换的简单程序。该程序可显示如下转换:

- 十进制转换为十六进制
- 十六进制转换为八进制
- 十进制转换为八进制
- 八进制转换为十进制

此程序首先让你在一个菜单中选择转换的类型,然后提示输入转换的数据。最后将显示该数据在所选格式中的形式。这就形成了一个很好的应用程序。

此程序的转换方法的关键在于两个特殊的 `printf()` 和 `scanf()` 格式指令: `%X` 和 `%O`。当在 `printf()` 中使用 `%X` 格式指令时,可将一个整数以十六进制显示。如果在 `scanf()` 调用中使用 `%X`,将使 `scanf()` 函数输入一个十六进制形式的整数。同样, `%O` 格式代码将使 `printf()` 函数输出一个八进制整数,使 `scanf()` 函数读入一个八进制形式的整数。

下面是实际的转换程序,它使用一系列 `if` 以决定将执行何种转换。因为等式操作只能匹配一个菜单选择项,每次程序运行只能执行一次转换。注意这个程序中每个 `if` 的目标语句是一个代码块。

```
/* Number Base converter program #1.

    decimal --> hexadecimal
    hexadecimal --> decimal
```

```

        decimal --> octal
        octal --> decimal

*/

#include <stdio.h>

main()
{
    int choice;
    int value;

    printf("Convert:\n");
    printf("    1: decimal to hexadecimal\n");

    printf("    2: hexadecimal to decimal\n");
    printf("    3: decimal to octal\n");
    printf("    4: octal to decimal\n");
    printf("enter your choice: ");
    scanf("%d", &choice);

    if(choice==1) {
        printf("enter decimal value: ");
        scanf("%d", &value);
        printf("%d in hexadecimal is: %x", value, value);
    }

    if(choice==2) {
        printf("enter hexadecimal value: ");
        scanf("%x", &value);
        printf("%x in decimal is: %d", value, value);
    }

    if(choice==3) {
        printf("enter decimal value: ");
        scanf("%d", &value);
        printf("%d in octal is: %o", value, value);
    }

    if(choice==4) {
        printf("enter octal value: ");
        scanf("%o", &value);
        printf("%o in decimal is: %d", value, value);
    }
}

```

### 6.1.1 else 语句的使用

可以将 else 与任何一个 if 联结起来。如果该 if 的条件表达式为真，它的目标语句将被执行。如果为假，则执行 else 的目标语句。下面的程序说明了这个基本原则。

```

/* A simple if-else example. */

#include <stdio.h>

main()
{
    int i;

```

```

printf("enter a number: ");
scanf("%d", &i);

if(i<0) printf("number is negative");
else printf("number is positive or zero");
}

```

对于你输入每个整数，该程序将告诉你它是否为负数。

### 6.1.2 If-else-if 阶梯

一个常用的程序设计结构为 if-else-if 阶梯。下面的例子说明了这个结构：

```

if (condition)
    statement;
else if (condition)
    statement;
else if (condition)
    statement;
.
.
.
else
    statement;

```

条件表达式是自顶向下执行的。一旦发现一个真条件，与该条件相联结的语句将被执行，而梯形的其它部分将被忽略。如果没有一个条件为真，则执行最后的 else。该 else 语句通常为一个缺省条件，也就是，如果所有其它条件测试均失败，则执行最后一个 else 语句。如果最后的 else 语句不存在且所有条件测试为假，则不执行任何操作。

可以使用一个 if-else-if 阶梯来改进前面设计过的数据基制转换程序。在原来的版本中，仅当前面的每个语句均执行成功后，才逐一执行每个 if 语句。在这种情况下，尽管没有什么重要性，但所有的 if 冗余计算在原则上不是很有效而且在结构上不是很精巧。下面的程序解决了这一问题。在这个 if-else-if 阶梯版本中，一旦有一个 if 语句成功，其它的语句将被忽略。

```

/* Number Base converter program #2 - if-else-if ladder.
   decimal --> hexadecimal
   hexadecimal --> decimal
   decimal --> octal
   octal --> decimal

*/

#include <stdio.h>

main()
{
    int choice;
    int value;

    printf("Convert:\n");
    printf("    1: decimal to hexadecimal\n");
    printf("    2: hexadecimal to decimal\n");

```

```

printf("      3: decimal to octal\n");
printf("      4: octal to decimal\n");
printf("enter your choice: ");
scanf("%d", &choice);

if(choice==1) {
    printf("enter decimal value: ");
    scanf("%d", &value);
    printf("%d in hexadecimal is: %x", value, value);
}
else if(choice==2) {
    printf("enter hexadecimal value: ");
    scanf("%x", &value);
    printf("%x in decimal is: %d", value, value);
}
else if(choice==3) {
    printf("enter decimal value: ");
    scanf("%d", &value);
    printf("%d in octal is: %o", value, value);
}
else if(choice==4) {
    printf("enter octal value: ");
    scanf("%o", &value);
    printf("%o in decimal is: %d", value, value);
}
}

```

### 6.1.3 条件表达式

有时，C 的初学者常感到迷惑，即任何一个正确的 C 表达式均可用于控制 if 语句，换句话说，表达式的类型不必只限于包括关系和逻辑操作符（例如在 BASIC 语言中的情况）。所必需的是表达式的计算结果或为 0 或不为 0。例如，下面这个程序从键盘读入两个整数并显示两数之商。为避免被 0 除的错误，将使用一个由第二个数控制的 if 语句。

```

/* Divide the first number by the second. */

#include <stdio.h>

main()
{
    int a, b;

    printf("enter two numbers: ");
    scanf("%d%d", &a, &b);

    if(b) printf("%d\n", a/b);
    else printf("cannot divide by zero\n");
}

```

这种方法的原理是：因为 b 等于 0，则控制 if 的条件为假，所以执行 else 语句。否则，若条件为真（不为 0），则进行除法。没有必要将这个 if 语句写成如下形式：

```

if (b==0) printf("%d\n", a/b);

```

因为这是多余的。

### 6.1.4 嵌套 if

任何一个程序设计语言中最易混淆的问题之一为嵌套 if 语句。嵌套 if 是指一个作为 if

或 else 的目标语句的 if 语句。嵌套 if 之所以麻烦是因为难以弄清哪个 else 与哪个 if 相联。思考下面这个例子：

```
if (X)
    if (Y) printf("1");
    else   printf("2");
```

这个 else 与哪个 if 相对应？幸好 C 提供了一个简单的规则来解决这个问题。在 C 中，else 与同一代码段中没有 else 语句相联系的最近 if 相对应。这样，该 else 与 if(Y) 语句联结。为使 else 与 if(X) 相联，必须使用花括弧以取消其正常的联结，如下所示：

```
if (X) {
    if (Y) printf("1");
}
else   printf("2");
```

现在 else 与 if (X) 联结，因为它不再是 if (Y) 代码段的一部分。

## 6.2 switch 语句

尽管 if-else-if 梯形可以完成多种形式的测试，但它在结构上并不精巧。代码很难弄懂，而且日后甚至连其设计者也会迷惑。正是由于这些原因，C 有一个固定的多路判断语句称为 switch。在 switch 中，一个变量有效地与一系列整数或字符常量相比较。一旦发现匹配，则执行与那个常量相联的语句或指令序列。常量不用特定的次序。switch 语句的一般形式为：

```
switch(variable) {
    case constant1:
        statement sequence
        break;
    case constant2:
        statement sequence
        break;
    case constant3:
        statement sequence
        break;
    .
    .
    .
    default:
        statement sequence
}
```

如果没有发现匹配，则执行 default 语句。default 是任选的，如果不存在，当所有的匹配失败时，将不发生任何动作。一旦发现匹配，与那个 case 相联的语句将被执行，直到 break 语句，或在 default 语句中（若 default 不存在，则为最后一个 case），将执行到 switch 的最后一个语句。（switch 语句类似于 BASIC 的 ON-goto 语句和 Pascal 的 CASE 语句。）

有关 switch 语句须知道下列三点：



1. 与 if 语句的不同点在于 switch 仅测试相等，而 if 条件表达式可为任何一种形式。
2. 在同一个 switch 中，两个 case 常量不能为同一值。当然，嵌套的两个 switch 语句可以有相同的 case 常量值。

3. 一个 switch 语句比 if-else-if 更为有效。

通常，可使用 switch 将一个菜单选择设计为适当的程序。根据这种方法，可以使用一个 switch 语句将前面的数据基制转换程序做个较大的改进。下面的版本删去了不美观的 if 序列，代之以一个清晰的 switch 语句：

```
/* Number Base convertor program #3 using the switch statement.
   decimal --> hexadecimal
   hexadecimal --> decimal
   decimal --> octal
   octal --> decimal

*/

#include <stdio.h>

main()
{
    int choice;
    int value;

    printf("Convert:\n");
    printf("    1: decimal to hexadecimal\n");
    printf("    2: hexadecimal to decimal\n");
    printf("    3: decimal to octal\n");
    printf("    4: octal to decimal\n");
    printf("enter your choice: ");
    scanf("%d", &choice);

    switch(choice) {
        case 1:
            printf("enter decimal value: ");
            scanf("%d", &value);
            printf("%d in hexadecimal is: %x", value, value);
            break;
        case 2:
            printf("enter hexadecimal value: ");
            scanf("%x", &value);
            printf("%x in decimal is: %d", value, value);
            break;
        case 3:
            printf("enter decimal value: ");
            scanf("%d", &value);
            printf("%d in octal is: %o", value, value);
            break;
        case 4:
            printf("enter octal value: ");
            scanf("%o", &value);
            printf("%o in decimal is: %d", value, value);
            break;
    }
}
```

### 6.2.1 default 语句

可以在 switch 中加入一个 default 语句, 在没有发现任何匹配时执行。default 语句是一种解决 switch 语句中不紧密结尾的好方法。如在数据基制转换程序中, 可以使用一个 default 语句通知用户输入了一个无效回答并请重新输入, 如下所示:

```
switch(choice) {
    case 1:
        printf("enter decimal value: ");
        scanf("%d", &value);
        printf("%d in hexadecimal is: %x", value, value);
        break;
    case 2:
        printf("enter hexadecimal value: ");
        scanf("%x", &value);
        printf("%x in decimal is: %d", value, value);
        break;
    case 3:
        printf("enter decimal value: ");
        scanf("%d", &value);
        printf("%d in octal is: %o", value, value);
        break;
    case 4:
        printf("enter octal value: ");
        scanf("%o", &value);
        printf("%o in decimal is: %d", value, value);
        break;
    default:
        printf("invalid selection, try again\n");
        break;
}
```

### 6.2.2 break 语句的进一步讨论

尽管在 switch 中一般都需要 break 语句, 但语法上它们是任选的。break 语句用于结束与每个常量相联的语句序列。但如果被省略, 语句的执行将持续到一个 case 语句直到 break 或 switch 结束。你可将 case 认做标号。语句的执行将从匹配的标号开始, 直到发现 break 语句或 switch 结束。特别注意在下面这个相当笨拙的程序中的 switch 语句:

```
/* A very silly program. */

#include <stdio.h>

main()
{
    int t;

    for(t=0; t<10; t++)
        switch(t) {
            case 1:
                printf("Now");
                break;
            case 2:
                printf(" is ");
            case 3:
                printf("the");
                printf(" time for all good men\n");
                break;
        }
```

```

        case 5:
        case 6:
            printf("to ");
            break;
        case 7:
        case 8:
        case 9:
            printf(".");
    }
}

```

程序执行时，产生下列输出：

```

Now is the time for all good men
the time for all good men
to to ...

```

这个程序也说明了可使用空 `case` 语句。当一些条件使用相同的语句序列时，可使用这种方式。你大概已经想到，当没有 `break` 时，所有的 `case` 一起执行会使程序高效而且避免列出不必要的重复代码。

必须重点理解与每个标号相联结的语句不是代码块，更确切地说是语句序列。（当然，整个 `switch` 语句定义了一个块）。这种技术上的区别一般不很重要，除了在某个特定的位置上，这一点将在本书后面章节中讨论。

### 6.2.3 嵌套的 `switch` 语句

`switch` 有可能作为一个外层 `switch` 的语句序列的一部分。尽管外层 `switch` 和内层 `switch` 包含相同值，但不会发生任何冲突。例如，下面的代码段是非常完美的：

```

switch(x) {
    case 1:
        switch(y) {
            case 0: printf("divide by zero error");
                    break;
            case 1: process(x, y);
        }
        break;
    case 2:
        .
        .
        .
}

```

又如另一个如下所示的非常简单的数据库程序，说明了如何使用嵌套的 `switch` 语句。该程序要求用户输入区域和售货员姓名的头一个字母，然后显示这个人当前的销售数字。因为一些售货员的姓名可能具有相同的头字母，所以需要用嵌套的 `switch`。注意一个新的标准库函数，`toupper`，它返回其字符自变量的等值大写字符。在这个程序中使用该函数可以允许用户输入回答时用大写或小写形式。（与 `toupper()` 互补的是 `tolower`，它将大写字符转换为小写字符。）`toupper()` 需要的头文件是 `CTYPE.H`。

```

/* A simple regional salesperson database. */

#include <stdio.h>
#include <conio.h>
#include <ctype.h>

```

```

main()
{
    char division, salesperson;

    printf("Divisions are: East, Midwest, and West\n");
    printf("Enter first letter of division: ");
    division = getche();
    division = toupper(division); /* make uppercase */
    printf("\n");
    switch(division) {
        case 'E':
            printf("Salespersons are: Ralph, Jerry, and Mary\n");
            printf("Enter the first letter of salesperson: ");
            salesperson = toupper(getche());
            printf("\n");

            switch(salesperson) {
                case 'R': printf("Sales: $zd\n", 10000);
                    break;
                case 'J': printf("Sales: $zd\n", 12000);
                    break;
                case 'M': printf("Sales: $zd\n", 14000);
                    break;
            }
            break;

        case 'M':
            printf("Salespersons are: Ron, Linda, and Harry\n");
            printf("Enter the first letter of salesperson: ");
            salesperson = toupper(getche());
            printf("\n");

            switch(salesperson) {
                case 'R': printf("Sales: $zd\n", 10000);
                    break;
                case 'L': printf("Sales: $zd\n", 9500);
                    break;

                case 'H': printf("Sales: $zd\n", 13000);
                    break;
            }
            break;

        case 'W':
            printf("Salespersons are: Tom, Jerry, and Rachel\n");
            printf("Enter the first letter of salesperson: ");
            salesperson = toupper(getche());
            printf("\n");

            switch(salesperson) {
                case 'R': printf("Sales: $zd\n", 5000);
                    break;
                case 'J': printf("Sales: $zd\n", 9000);
                    break;
                case 'T': printf("Sales: $zd\n", 14000);
                    break;
            }
            break;
    }
}

```

为理解它是怎样工作的，键入 M 选择中西部区域。这意味着 case 'M' 是由外层 switch

语句选取的。若要看 Harry 的销售总额键入 H，将显示 13000 值。

注意一个嵌套 switch 中的 break 语句对外层 switch 毫无影响。

### 6.3 循环

循环允许重复一系列指令，直到达到某个条件为止。C 支持同其它新的结构化语言相同类型的循环。C 循环是指 for、while 和 do\_while。以下将按顺序讨论。

#### 6.4 for 循环

尽管 for 循环的简单形式已经在第四章介绍过，但在这里你将惊异于它的强有力和灵活性。先回顾一下已学过的知识。

##### 6.4.1 for 循环的基本知识

for 循环的一般形式为：

```
for(initialization; condition; Increment) statement;
```

在这个最简单的形式中，initialization 是一个用于设置循环控制变量初始值的赋值语句。condition 通常为一个关系表达式，通过将循环控制变量同某个值比较，从而决定何时退出循环。Increment 通常规定了循环重复时循环控制变量的变化方式。这三个主要部分必须用分号隔开。只要条件为真，for 循环将一直执行。一旦条件为假，程序将继续执行 for 语句后面的语句。

举一个简单的例子，下面的程序在终端上打印数字 1~100：

```
#include <stdio.h>

main()
{
    int x;

    for(x=1; x<=100; x++) printf("2d ", x);
}
```

在这个程序中，X 初始设置为 1。因为 X 小于 100，则调用 printf()。在 printf() 函数返回后，X 被加 1，测试它是小于 100 或等于 100。这个过程将重复执行，直到 X 大于 100，此时循环终止。在这个例子中，X 是循环控制变量，每次循环重复时，它将被改变和测试。

for 循环不一定总是正向运行。减小而不是增大循环控制变量导致反向运行循环。例如，下面的程序在屏幕上打印数字 100~1：

```
#include <stdio.h>

main()
{
    int x;

    for(x=100; x>0; x--) printf("2d ", x);
}
```

但并不限制将循环控制变量增 1 或减 1。可以任意改变循环控制变量。例如，这个循环在 0~100 之间每隔 5 个数打印一个数：

```

#include <stdio.h>

main()
{
    int x;

    for(x=0; x<=100; x=x+5) printf("%d ", x);
}

```

通过使用一个代码块，可以得到 for 循环复合语句，如下例所示，它将打印 0~99 的平方数：

```

#include <stdio.h>

main()
{
    int i;

    for(i=0; i<100; i++) {
        printf("this is i: %d", i);
        printf(" and i squared: %d\n", i*i);
    }
}

```

有关 for 循环，值得一提的是条件检测一般都在循环顶部执行。这意味着如果开始的条件为假，循环内的代码根本不执行。例如：

```

x=10;
for (y=10; y!=x; ++y)    printf("%d",y);
printf("%d", y);

```

这个循环永远不会执行，因为当进入循环时，X 和 Y 实际上相等。故而条件表达式计算为假，无论循环体还是循环的增值部分都将不执行。从而 Y 仍为原来赋给它的值 10，输出只是在屏幕上显示 10。

#### 6.4.2 for 循环的变化

前面讨论了 for 循环的最后一般形式。但也允许一些变化来提高它的功能灵活性和对某个设计环境的适应性。

一个最一般的变化是使用两个或更多的循环控制变量。下面的例子中使用 X 和 Y 两个变量控制循环：

```

#include <stdio.h>

main()
{
    int x,y;

    for(x=0, y=0; x+y<100; ++x, y++)
        printf("%d ", x+y);
}

```

这个程序在 0~98 中每隔两个数打印一个数。注意使用了逗号分隔初始化语句和增值语句。逗号通常为一个 C 操作符，基本含义为“做这个和那个”，将在本书后面章节中作

详细讨论。每次通过循环，X 和 Y 都被增值，并且最后 X 和 Y 必须为某个适当的值以使循环终止。

条件表达式不一定是循环控制变量与某个目标值的比较测试。实际上，条件可为任一有效的 C 表达式。这表明你可以测试一些可能的终止条件。例如，下面的程序帮助儿童练习加法。如果小孩疲劳了，想停止，当程序请求输入时，可键入 N。特别注意 for 循环中的条件部分。该条件表达式将使 for 循环执行 99 次或直到用户回答提示为“不” (N)。

```
/* Addition drill. */
#include <stdio.h>
#include <conio.h>

main()
{
    int i, j, answer;
    char done = ' ';

    for(i=1; i<100 && done!='N'; i++) {
        for(j=1; j<10; j++) {
            printf("what is %d + %d? ", i, j);

            scanf("%d", &answer);
            if(answer != i+j) printf("wrong\n");
            else printf("right\n");
        }
        printf("more? ");
        done = getche();
        printf("\n");
    }
}
```

for 循环的另一个有趣的变化可能是，for 语句的三个部分均由任何有效的 C 表达式组成，实际上没有必要同标准状况下的各部分完全相同。基于这种思想，请看下面这个例子：

```
/* An unusual use of the for loop. */
#include <stdio.h>

main()
{
    int t;

    for(prompt(); t=readnum(); prompt())
        sqnum(t);
}

prompt()
{
    printf("enter an integer: ");
}

readnum()
{
    int t;

    scanf("%d", &t);
    return t;
}
```

```

sqnum(int num)
{
    printf("2d\n", num*num);
}

```

这个程序首先显示一个提示，然后等待输入。当输入一个数字后，将显示出它的平方数，然后再提示输入。这个过程将持续到输入0为止。如果你仔细看一看 `main()` 函数中的 `for` 循环，将发现 `for` 语句的每个部分均由提示用户和从键盘读入数据的函数调用组成。如果输入数字为0，则条件表达式为假，循环终止；否则，将求出数据的平方数。这样，在这个 `for` 循环中，初始化和增值部分的使用不因过于陈旧却非常有效。你可以输入到程序，并试着加以调整，使其按不同的方式工作。

`for` 循环的另一个有趣特征为循环定义中的各部分未必都存在。实际上，没有必要每个部分都有一个表达式与之对应——表达式是任选的。例如，这个循环将运行到数字10的输入：

```
for (x=0; x!=10; ) scanf("%d", &x);
```

注意 `for` 定义中的增值部分为空。这意味着每次循环重复，都将测试 `x` 是否等于10，但并不改变 `x`。如果你键入10，循环条件变为假，于是循环中止。

### 6.4.3 无穷循环

`for` 循环的最有趣的应用之一为无穷循环的产生。组成循环的三个表达式都不需要，通过将条件表达式置为空，可能产生一个无终止循环，如下例所示：

```
for ( ; ; ) printf("this loop will run forever. \n");
```

### 6.4.4 `for` 循环的中断

`for( ; ; )` 结构不一定产生无穷循环，因为可以利用 C `break` 语句的一个新的应用。在循环体的任意地方遇到 `break`，都将使循环立即终止。程序接着转到循环后面的代码，如下所示：

```

for( ; ; ) {
    ch=getche(); /* get a character */
    if (ch=='A') break; /* exit the loop */
}
printf("you typed an A");

```

这个循环将一直运行到 A 的键入。`break` 语句将在后面的章节中更详细地讨论。

### 6.4.5 无循环体循环的使用

一个由 C 语法定义的语句可以为空。这说明 `for` 的体（在这一点上，也可以是其它循环）也可为空。这个事实可用于提高某个算法的效率，而且可以产生延时循环。下面举例说明用 `for` 产生延时的方法：

```
for (t=0; t<SOME_VALUE; t+ );
```

## 6.5 `while` 循环

C 中的第二个可用循环为 `while`。一般形式为：



**while (condition) statement;**

**statement**, 如前所述, 可为一个空语句、单语句或一个重复语句块。**condition** 为任一有效的表达式。当条件为真时, 循环重复执行。当条件为假, 程序控制将转向循环代码后的那个行。

下面的例子是一个键盘输入程序, 该程序为一个简单的循环直至按下字符 A 为止:

```
wait_for_char()
{
    char ch;
    ch = '\0';    /* Initialize ch */
    while(ch != 'A') ch = getch();
}
```

首先, **ch** 初始化为空。作为一个局部变量, 当 **wait\_for\_char** 执行时, 其值未知。若检测 **ch** 不等于 A, 则 **while** 循环开始。因为 **ch** 已预先初始化为空, 检测为真, 于是循环开始。每次键盘按下一个键, 测试将进行一次。一旦 A 被按下, 因 **ch** 等于 A, 条件为真, 于是循环终止。

同 **for** 循环相同, **while** 循环在循环顶部检查测试条件, 这说明循环代码有可能根本不执行。这就解释了前面例子中 **C** 为什么必须初始化, 是为了防止它偶然包含 A。因为条件测试在循环顶部执行, 因此 **while** 适用于循环可能不执行的情况。这就消除了循环之前的分散的条件测试。

例如, 下面程序中的 **center()** 函数使用一个 **while** 循环输出正确的空格数以便中心对准一个文本行。如果 **len** 等于 0, 即将要中心对准的行为 80 字符长, 循环将不执行。该程序还使用了另一个库函数叫做 **strlen()**, 返回其字符串自变量的长度。它使用头文件 **STRING.H**。

```
/* A program that centers text on the screen. */
#include <stdio.h>
#include <string.h>

main()
{
    char str[255];
    int len;

    printf("enter a string: ");
    gets(str);

    center(strlen(str));
    printf(str);
}

/* Compute and output proper number of spaces to
   center a string of len length.
*/
center(int len)
{
    len = (80-len)/2;

    while(len>0) {
        printf(" ");
    }
}
```

```

        len--;
    }
}

```

在一些分散条件终止 while 循环的地方，通常使用单一变量作为条件表达式，设置其值可以在整个循环的任何地方。例如：

```

func1()
{
    int working;

    working = 1;    /* i.e., true */

    while(working) {
        working = process1();
        if(working)
            working = process2();
        if(working)
            working = process3();
    }
}

```

这里，三个程序均可返回假值而使循环退出。

while 循环体中可以没有语句。例如，

```
while ((ch=getche())!='A');
```

将简单地循环直至键入字符 A。如果你不习惯于在 while 条件表达式中赋值，请记住等号是一个计算其右边操作数值的操作符。

## 6.6 do\_while 循环

for 和 while 循环是在循环顶部测试循环条件，与此不同，do\_while 循环是在循环底部测试。这意味着一个 do while 循环至少执行一次。do\_while 循环的一般形式为

```

do {
    statement;
} while (condition);

```

尽管在只有一个语句时，大括号并不必要，但通常加上大括号是为了提高可读性及避免与 while 混淆（对设计者来说，而不是编译程序）。

下面的程序利用一个 do\_while 循环从键盘上读入数据，直到有一个数小于 100：

```

#include <stdio.h>

main()
{
    int num;

    do {
        scanf("%d", &num);
    } while(num>100);
}

```

大概 do\_while 循环的最一般应用是在菜单选择程序中。因为你往往希望一个菜单选择程序至少运行一次，由于是在循环体的底部测试是否有效回答。可以重新提示用户直到用户输入有效回答。下面这段显示了如何在数据基制转换程序的菜单中加入一个 do\_while 循环：

```

/* make sure that the user specifies a valid option */
do {
    printf("Convert:\n");
    printf("      1: decimal to hexadecimal\n");
    printf("      2: hexadecimal to decimal\n");
    printf("      3: decimal to octal\n");
    printf("      4: octal to decimal\n");
    printf("enter your choice: ");
    scanf("%d", &choice);
} while(choice<1 || choice>4);

```

在列出选择项以后，程序将循环等待直到一个有效选择的产生。

## 6.7 嵌套循环

当一个循环在另一个循环之中，里面的循环称之为嵌套。嵌套循环提高了解决一些趣味程序设计问题的方法。例如，这个小程序显示数字 1~9 的 1~4 次方幂：

```

/* Display a table of the first four powers of the
   numbers 1 to 9.
*/

#include <stdio.h>

main()
{
    int i, j, k, temp;

    printf("      1      1^2      1^3      1^4\n");
    for(i=1; i<10; i++) { /* outer loop */
        for(j=1; j<5; j++) { /* 1st level of nesting */
            temp = 1;
            for(k=0; k<j; k++) /* innermost loop */
                temp = temp*i;
            printf("%9d", temp);
        }
        printf("\n");
    }
}

```

该程序运行时，产生的结果如图 6-1 所示。注意图中每列数字均对齐。这是因为在打印数据的 `printf()` 语句中使用了“最小区域宽度说明符”。如果在 % 号和 d 之间有一个数字，它告知 `printf()` 加入必要的空格数以达到规定的宽度。这样可以使各列中的数字对齐。

i	1^2	1^3	1^4
1	1	1	1
2	4	8	16
3	9	27	81
4	16	64	256
5	25	125	625
6	36	216	1296
7	49	343	2401
8	64	512	4096
9	81	729	6561

图 6-1 power 程序的输入

有时决定内层循环体的执行次数很重要，该数等于外层循环的重复次数乘以每次外层

循环重复执行时内层循环的重复次数。在这个乘方例子中，外层循环重复 9 次，每次执行时，第二层循环重复 4 次；这样，第二层循环体实际执行 36 次。最内层循环执行 2 次，所以它总的执行次数为 72。

下面是嵌套循环的最后一个例子，使用嵌套循环对数据基制转换程序作最后的改进。外层循环使程序一直运行到用户让其停止。内层循环一直持续到用户输入一个菜单选择项。现在，不是每次执行只转换一个数字，而是一直重复执行直到用户让它停止。这就允许转换一批数据而不必每次重新启动程序。

```

/* Number Base converter program: Final version using
   nested while loops.

       decimal --> hexadecimal
       hexadecimal --> decimal
       decimal --> octal
       octal --> decimal

*/

#include <stdio.h>

main()
{
    int choice;
    int value;

    /* repeat until user says to quit */
    do {
        /* make sure that the user specifies a valid option */
        do {
            printf("Convert:\n");
            printf("    1: decimal to hexadecimal\n");
            printf("    2: hexadecimal to decimal\n");
            printf("    3: decimal to octal\n");
            printf("    4: octal to decimal\n");
            printf("    5: quit\n");
            printf("enter your choice: ");
            scanf("%d", &choice);
        } while(choice<1 || choice>5);

        switch(choice) {
            case 1:
                printf("enter decimal value: ");
                scanf("%d", &value);
                printf("%d in hexadecimal is: %x", value, value);
                break;
            case 2:
                printf("enter hexadecimal value: ");
                scanf("%x", &value);
                printf("%x in decimal is: %d", value, value);
                break;
            case 3:
                printf("enter decimal value: ");
                scanf("%d", &value);
                printf("%d in octal is: %o", value, value);
                break;
            case 4:
                printf("enter octal value: ");
                scanf("%o", &value);

```

```

        printf("Zo in decimal is; %d", value, value);
        break;
    }
    printf("\n");
} while(choice!=5);
}

```

### 6.8 循环中断

**break** 语句有两个用途。其一是在 **switch** 语句中终止 **case**，这一点已在本章前面的 **switch** 部分中讨论过；其二是强制一个循环立即终止，忽略一般的循环条件测试。下面讨论之。

在一个循环中遇到 **break** 语句时，循环将立即终止，程序控制将转向循环后面的那条语句。例如：

```

#include <stdio.h>

main()
{
    int t;

    for(t=0; t<100; t++) {
        printf("%d ", t);
        if(t==10) break;
    }
}

```

这个程序将在屏幕上打印数字 0~10，然后终止。因为 **break** 将导致程序立即退出循环，而越过循环内部的条件测试 **t<100**。

当一个外部事件控制循环时，**break** 特别适用。下面这个测试时间观念的例子正说明了这一点。它要求在开始和结束程序间等待 5 秒钟。如果你认为 5 秒钟已到，敲任意键。如果你的感觉是正确的，就赢了。

```

/* How's your internal timer? */

#include <stdio.h>
#include <time.h>
#include <conio.h>

main()
{
    long tm;

    printf("This program tests your sense of time!\n");
    printf("When ready, press return, wait five seconds\n");
    printf("and strike any key: ");
    getche();
    printf("\n");

    tm = time(0);
    for(;;)
        if(kbhit()) break;
    if(time(0)-tm==5) printf("You win!!!");
    else printf("Your timing is off");
}

```

这个程序使用 Turbo C 的 `time()` 函数以秒数读当前的系统时间。`time()` 函数需要头文件 `TIME.H` 以保证其正确操作。因为秒数有可能超过一个整数的范围，因此需用一个 `long int` 型变量。(以 0 为自变量的 `time()` 函数将返回时间值。在你懂得指针——后面章节中的内容之前，不要使用其它值作为该函数的自变量。) 这个程序还介绍了另一个库函数，`kbhit()`，它用于检查是否已敲下一个键。如果有，则返回其值；否则，返回值为假。它的头文件为 `CONIO.H`。

认识到一个 `break` 只导致从最内层循环的退出是很重要的。如考虑下面这段：

```
for(t=0; t<100; ++t) {
    count=1;
    for(;;) {
        printf("%d ", count);
        count++;
        if(count==10) break;
    }
}
```

它将在屏幕上将数字 1~10 打印 100 次，每次遇到 `break`，程序控制将返回到外层 `for` 循环。一个 `switch` 语句中的 `break` 只影响其自身，而不影响它所在的任一循环。

## 6.9 continue 语句

`continue` 语句的作用方式与 `break` 语句有点类似。但 `continue` 不是强制终止，而是使下一个循环迭代开始，跳过中间的任何代码。例如，下面的程序将只显示偶数：

```
#include <stdio.h>

main()
{
    int x;

    for(x=0; x<100; x++) {
        if(x%2) continue;
        printf("%d ", x);
    }
}
```

每当产生一个奇数，`if` 语句都将执行，因为奇数与 2 求模等于 1，为真。这样，奇数将使 `continue` 执行，再一次重复循环，略过 `printf()` 语句。

在 `while` 和 `do_while` 循环中，`continue` 语句将使程序控制直接转向条件测试，然后继续循环过程。如果是 `for` 语句，将首先执行循环的增值部分，然后再进行条件测试，最后继续循环过程。

正如你将在下面的例子中看到的，一旦遇到终止条件，`continue` 将强制执行条件测试，这样便能加快循环的终止。思考下面的程序，其作用就像一个简单的代码机器：

```
/* A simple code machine. */
#include <stdio.h>
#include <conio.h>

main()
{
    printf("Enter the letters you want coded.\n");
    printf("Type a $ when you are done.\n");
```

```

    code();
}

/* code the letters */
code()
{
    char done, ch;

    done = 0;
    while(!done) {
        ch = getch();
        if(ch=='$') {
            done = 1;
            continue;
        }
        printf("%c", ch+1); /* shift the alphabet one
                             position */
    }
}

```

利用这个函数，你可将所有字符分别变为高一级的字母；如，a 变为 b。当读到\$时，函数将终止执行。不可能有更多的重复，因为由 continue 作用的条件测试将发现 done 为真的情况而使循环退出。

#### 6.10 标号和 goto

尽管几年前 goto 不再流行，但最近又对它作了一些改进。本节并不将它的有效性作为程序控制的一种形式，但值得一提的是没有任何程序设计环境需要它，它并不是使语言完整的必要项目。然而，如果能够巧妙运用，对于某个程序设计环境将有益处。因此，goto 在本书的其它部分并未使用。（在像 C 这样的具有丰富的控制结构，允许使用 break 和 continue 作附加控制的语言中，很少需要使用 goto）关于 goto，大多数程序设计者最担心的是它可能使程序变得混乱而几乎不可读。但有时 goto 的使用实际上能使程序流程更加清晰，而不是混乱。

goto 操作需要一个标号。标号是指一个有效的 C 标识符带一个冒号，而且标号与使用它的 goto 必须在同一个函数中。例如，下面的从 1 到 100 的循环使用了一个 goto 和一个标号：

```

x=1;
loop1:
    x++;
    if (x<100) goto loop1;

```

在从一个深层嵌套的程序中退出时，使用 goto 效果好。例如，请看下面的代码段：

```

for(...) {
    for(...) {
        while(...) {
            if(...) goto stop;
            .
            .
        }
    }
}

```

```
stop:
    printf("error in program\n");
```

删除 goto 语句将使一些附加测试强制执行。在这里，不能使用简单的 break 语句，因为它只能存在于最里层的循环。如果你在每个循环中代之以检测，则程序将会变成这种形式：

```
done = 0;
for(...) {
    for(...) {
        while(...) {
            if(...) {
                done = 1;
                break;
            }
            .
            .
            .
        }
        if(done) break;
    }
    if(done) break;
}
```

你应该小心使用 goto 语句或者根本不用。但如果没有 goto 语句，程序将变得非常难读，或者执行速度不尽人意，这样看来也可使用 goto。

由于已经完成对程序控制语句的讨论，因此下面转到数组和字符串方面的内容。



## 第七章 数组和字符串

一个数组是一个由共同名字相关联的、具有相同类型的变量的集合。在 C 中,所有的数组均由连续的内存地址组成。最低位地址与第一个元素相对应,最高位地址与最后一个元素相对应,数组可以有一维到多维。数组中指定元素通过下标来查找。

最常用的数组是字符数组。因为在 C 中没有建立字符串数据类型,所以采用字符数组。你将看到,这种字符串的操作方式要比使用专门的字符串类型的语言更有力、更灵活。

### 7.1 一维数组

一维数组说明的一般形式为:

```
type var_name[size];
```

这里, type 说明了数组的基类型。基类型决定了每个构成数组的元素的数据类型, size 定义了数组中可以容纳的元素个数。例如,下面说明了一个有 10 个元素、叫做 sample 的整数数组:

```
int sample[10]
```

在 C 中,所有的数组都把零当作第一个元素的下标。因此,它说明一个整数数组有 10 个元素, sample[0] 到 sample[9]。下面的程序把数字 0~9 装入到一个数组中:

```
main()
{
    int x[10]; /* this reserves 10 integer elements */
    int t;

    for(t=0; t<10; ++t) x[t]=t;
}
```

对于一个一维数数组,以字节数表示的数组大小按下列方式计算:

总字节数 = 基本类型数据大小 × 元素个数

因为数组很容易处理大量有关的数据,因此在编程时被广泛使用。例如,使用数组将比较容易计算数值表的平均值,如下面程序所示,它读取用户键入的 10 个整数并显示平均值:

```
/* Find the average of ten numbers. */
#include <stdio.h>

main()
{
    int sample[10], i, avg;

    for(i=0; i<10; i++) {
        printf("enter number %d: ", i);
        scanf("%d", &sample[i]);
    }

    avg = 0;
```

```

/* now, add up the numbers */
for(i=0; i<10; i++) avg = avg+sample[i];
printf("The average is %d\n", avg/10);
}

```

### 7.1.1 无界检测

C 对数组不进行边界检测；即可以超出数组的已定长度。如果这种超出是在赋值过程中，那么就要赋值给其变量的数据甚至于一段程序代码。大小为 N 的数组在寻址时若超过 N，不会产生编译或运行时间错误信息，但很可能毁坏你的程序。作为一个程序员，你应该保证所有数组的长度足够容纳程序将存入的数据，以及必要时提供边界检测。例如，下面的程序尽管数组 `crash` 将超界，但仍能编译和运行（千万不可调试这个例子，它将破坏你的系统！）。

```

/* An incorrect program. Do Not Execute! */
main()
{
    int crash[10], i;

    for(i=0; i<100; i++) crash[i]=i;
}

```

尽管 `crash` 只有 10 个元素的长度，但循环仍将重复 100 次。这样将导致重要的信息被覆盖，结果该程序将失败。

可能你会不理解 C 为什么没有提供数组的边界检查。其答案是大部分情况下，C 是为代替汇编语言代码而设计的。因此，实际上不包括错误检测，因为它减慢了程序的执行速度（常常很明显）。C 希望程序员首先十分负责地防止数组越界。

### 7.1.2 一维数组是一个表

一维数组本质上是相同类型的信息表。例如，这个程序运行之后，

```

char str[7];

main()
{
    int i;

    for(i=0; i<7; i++) str[i] = 'A'+i;
}

```

`str` 的内容如下：

str[0]	str[1]	str[2]	str[3]	str[4]	str[5]	str[6]
A	B	C	D	E	F	G

## 7.2 字符串

一维数组最广泛的应用是生成一个字符串。在 C 中，一个字符串定义为以空字符终止的字符数组。空字符规定为 `'\0'`，即零。由于这个空终止符，有必要说明字符数组比要装入的最大字符串长一个字符。例如，如果想说明一个可装入 10 个字符串的数组 `str`，应这

样写：

```
char str[11];
```

这样就给空字符在字符串末尾留下空间。

在第五章你已经看到，尽管 C 没有字符串数据类型，但仍允许字符串常量。字符串常量是一个以双引号括起来的字符表。例如：

```
"hello there" "This a test"
```

不需要人为地在字符串常量末尾加上空字符——编译器会自动加上。这意味着字符串“Turbo”在内存中这样出现：

T	u	r	b	o	'\0'
---	---	---	---	---	------

### 7.2.1 从键盘上读字符串

从键盘上输入一个字符串的最简单办法是使用库函数 `get()`，`gets()` 调用的一般形式为：

```
gets(array_name);
```

调用 `gets()`，以没有下标的数组名作为自变量来读取字符串。根据 `gets()` 的返回值，数组将存贮从键盘上输入的字符串。`gets()` 函数一直读取字符，直到键入回车键。`gets()` 使用的头文件是 `STDIO.H`。

例如，下面的程序只简单重复从键盘上输入的字符串：

```
/* A simple string example. */  
  
#include <stdio.h>  
  
main()  
{  
    char str[80];  
  
    printf("enter a string: ");  
    gets(str); /* read a string from the keyboard */  
  
    printf("%s", str);  
}
```

注意 `str` 可以用作 `printf()` 的自变量。还要注意使用的数组名没有下标。在学习了后面的一些章节之后，你就会明白不带下标的、存储字符串的数组的名字可以在任何使用字符串常量的地方使用。

记住 `gets()` 不对所有调用数组进行边界检测。因此，如果用户输入了一个大于数组大小的字符串，`gets()` 将其写入到数组末端的后面部分。

### 7.2.2 一些字符串库函数

C 支持一整套的字符串操作函数。最常用的是：

```
strcpy()  
strcat()  
strlen()  
strcmp()
```

字符串函数都使用相同的头文件：STRING.H。现在，让我们来看看这些函数：

### strcpy( )函数

strcpy 的调用采用这种形式：

strcpy( to,from )

strcpy( )函数用来将字符串 from 的内容拷贝到字符串 to 中去。记住构成 to 的数组必须足以存储包含在 from 中的字符串。如果不是这样，数组将超界——有可能破坏你的程序。

strcpy( )函数返回一个值，但在学习下一章之前你不理解，所以现在可忽略这一点。

下面的程序把“vend”拷贝到字符串 str 中：

```
#include <stdio.h>
#include <string.h>

main()
{
    char str[80];

    strcpy(str, "hello");
    printf("%s", str);
}
```

### strcat( )函数

调用 strcat( )函数采用这种形式：

strcat(s1 s2);

strcat( )函数把 s2 附加到 s1 上；s2 未改变。两个字符串必须都以空字符终止，最后结果是空字符终止。例如，下面这个程序在屏幕上打印“hello there”。

```
#include <stdio.h>
#include <string.h>

main()
{
    char s1[20], s2[10];

    strcpy(s1, "hello");

    strcpy(s2, " there");

    strcat(s1, s2);

    printf("%s", s1);
}
```

与 strcpy( )一样，strcat( )返回一个值，但你必须等到下一章才能理解它的意义。

### strcmp( )函数

调用 strcmp( )函数采用这种形式：

strcmp(s1,s2);

strcmp( )函数比较两个字符串，如果相等，返回 0。如果 s1 按字典顺序大于 s2，则返回一个正数；如果小于 s2，则返回负数。

下面的函数用于鉴别口令:

```
/* Return true if password accepted; false otherwise. */
password()
{
    char s[80];

    printf("enter password: ");

    gets(s);

    if(strcmp(s, "password")) { /* strings different */
        printf("invalid password\n");
        return 0;
    }

    /* strings compared the same */
    return 1;
}
```

使用 `strcmp()` 函数的关键是当字符串匹配时, 返回假。因此, 当字符串相同时, 如果希望做某事时, 需要使用 NOT 操作符。例如, 下面的程序一直要求输入直到键入单词 quit:

```
#include <stdio.h>
#include <string.h>

main()
{
    char s[80];

    for(;;) {
        printf("Enter a string: ");
        gets(s);
        if(!strcmp("quit", s)) break;
    }
}
```

### `strlen()` 函数

调用 `strlen()` 函数的一般形式是,

`strlen( s )`

这里, `s` 是一个字符串。

`strlen()` 函数返回字符串的长度。

下面的程序打印由键盘输入的字符串的长度:

```
#include <stdio.h>
#include <string.h>

main()
{
    char str[80];

    printf("enter a string: ");
```

```

    gets(str);

    printf("%d", strlen(str));

}

```

例如，如果键入“hi there”，该程序将显示“8”。空终止符不在 `strlen()` 的计算范围内。

下面的程序按逆序显示从键盘上输入的字符。例如，“hello”将显示成“olleh”。记住字符串是简单的字符数组；这样，每一个字符可逐一访问。

```

/* Print a string backwards. */
#include <stdio.h>
#include <string.h>

main()
{
    char str[80];
    int i;

    printf("enter a string: ");
    gets(str);

    for(i=strlen(str)-1; i>=0; i--) printf("%c", str[i]);
}

```

最后一个例子，下面的程序说明了这几个字符串函数的使用：

```

#include <stdio.h>
#include <string.h>

main()
{
    char s1[80], s2[80];

    printf("enter two strings: ");

    gets(s1); gets(s2);

    printf("lengths: %d %d\n", strlen(s1), strlen(s2));

    if(!strcmp(s1, s2)) printf("The strings are equal\n");

    strcat(s1, s2);
    printf("%s\n", s1);
}

```

如果运行这个程序，输入字符串“hello”和“hello”则结果为

lengths:5 5

The strings are equal

hellohello

记住如果字符串相等，`strcmp()` 函数返回假这点很重要，所以若要检测相等，一定要用！将条件取反，如这个例子所示。

### 7.2.3 空终止符的使用

所有字符串均以空字符结束，这一事实可以经常充分利用于精简对字符串的各种操

作。例如，下例用较少代码将字符串中的小写字母转换为大写字母：

```
/* Convert a string to uppercase. */
#include <stdio.h>
#include <string.h>
#include <ctype.h>

main()
{
    char str[80];
    int i;

    strcpy(str, "this is a test");

    for(i=0; str[i]; i++) str[i] = toupper(str[i]);

    printf("%s", str);
}
```

这个程序将显示“THIS IS A TEST”。它使用库函数 `toupper()` 转换字符串中的每一个字符，该函数返回其字符自变量的等值大写字母。其头文件为 `CTYPE.H`。注意 `for` 循环中的检测条件是以控制变量为下标的数组。这种做法的原因是任何非零值均为真。因此，循环一直进行直到碰到值为零的空终止符为止。由于这个空终止符标志着字符串的结束，因此循环可以确切地知道在哪里应该停止。随着水平的提高，你可以看到类似使用空终止符的例子。

#### 7.2.4 printf() 的一种变化

迄今为止，利用 `printf()` 显示存贮在字符数组中的字符串，都是采用下面的基本形式：

```
printf( "%s",array_name);
```

无论怎样，都应该记住 `printf()` 的第一个自变量是一个字符串，而且打印的是除格式命令外的所有字符。因此，如果你只是想打印一个字符串，可以使用下面的形式：

```
printf( array_name);
```

例如，下面的程序在屏幕上打印“Hello Tom”；

```
#include <stdio.h>
#include <string.h>

main()
{
    char str[80];

    strcpy(str, "Hello Tom");

    printf(str);
}
```

### 7.3 二维数组

C 允许有多维数组。多维数组最简单的形式是二维数组。一个二维数组实质上是一个一维数组表。为说明一个大小为 10 20 的二维整数数组 `twod`，应这样写：

```
int twod[10][20];
```

仔细看一下这个说明：与其它计算机语言使用逗号分隔数组各个维不同，C 将各个维放在各自的括号中：

```
main()
{
    int t, i, num[3][4];

    for(t=0; t<3; ++t)
        for(i=0; i<4; ++i)
            num[t][i] = (t+4)*i+1;
}
```

在这个例子中，num[0][0]存储数值 1，num[0][1]存储数值 3，num[0][2]存储数值 3，以此类推。num[2][3]的数值为 12。

二维数组以行列矩阵形式存储，第一个下标表明行，第二个下标表明列。这说明当按实际存储顺序访问数组元素时，最右下标要比最左下标变化得快。图 7-1 为一个二维数组在内存中的图解表示。实质上，第一（最左）下标可想象为指向存储中的二维数组某适当的指针。

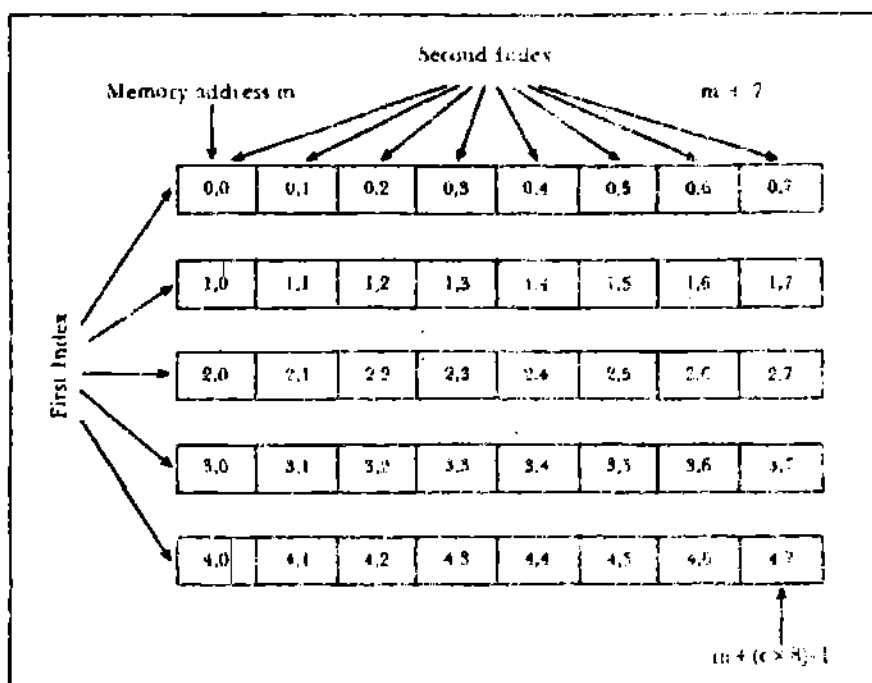


图 7-1 内存中的二维数组

应该记住编译时将全局数组元素定位，这意味着在程序执行的整个过程中，存储该全局数组的内存随时可用。

对于二维数组，下列公式用来计算内存的字节数：

字节数 = 行 × 列 × 数据类型的大小

因此，一个维数为 10, 5 的整数数组应占有：

$$10 \times 5 \times 2$$

或 100 个字节。



我们常用二维数组处理信息表。例如，下面这段程序打印一个棒球队的击球平均数：

```
#define PLAYERS 9
#define ATBATS 100

int battingavg[PLAYERS][ATBATS];

.
.
.

display_averages()
{
    int i, j;
    int hits;

    for(i=0; i<PLAYERS; i++) {
        for(j=0, hits=0; j<ATBATS; j++)
            hits = hits + battingavg[i][j];
        printf("Player %d hit %f\n", i+1,
            (float) hits/(float) ATBATS);
    }
}
```

在本章末尾你将看到另一个用二维数组工作的程序。

### 7.3.1 字符串数组

在编程过程中，使用字符串数组是不少见的。例如，数据库的输入处理程序可从有效命令的字符串数组中辨别出用户命令。为生成一个字符串数组，可使用一个二维字符数组，其左下标的大小决定字符串的个数，右下标的大小规定每个字符串的最大长度。例如，下面的程序段说明了一个有 30 个字符串的数组，每个字符串的最大长度为 80 个字符：

```
char str_array[30][80];
```

访问某个单独字符串是十分容易的：只需简单指明左下标即可。例如，这个语句调用 `gets()` 访问 `str_array` 中的第三个字符串，

```
gets(str_array[2]);
```

这在功能上等同于

```
gets(&str_array[2][0]);
```

但前者在描写 C 程序时显得更专业化。后者需要使用“&”，其原因将在下一章阐明。

为帮助你更好地了解字符串是如何工作的，研究下面这个小程序，该程序接收从键盘上输入的正文行，输入空行时将它们重新显示。

```
/* Enter and display strings. */
#include <stdio.h>

main()
{
    register int t, i;
    char text[100][80];

    for(t=0; t<100; t++) {
        printf("%d: ", t);
        gets(text[t]);
    }
}
```

```

    if(!text[t][0]) break; /* quit on blank line */
}

/* redisplay the strings */
for(i=0; i<t; i++)
    printf("%s\n", text[i]);
}

```

该程序接收正文行，直到键入一个空行，然后重新显示各行。

## 7.4 多维数组

C 允许多于两维的数组。多维数组说明的一般形式为：

```
type name[size1][size2]...[sizeN]
```

例如，生成一个  $4 \times 10 \times 3$  的整数数组：

```
int threed[4][10][3]
```

三维数组不常使用，因为它们需要大量的内存空间。在程序的执行过程中，全局数组元素将常驻内存。例如，一个维值为 10, 6, 9, 4 的四维字符型数组需要

$10 \times 6 \times 9 \times 4$

或 2, 160 个字节。

如果该数组为双字节整数，则需要 4320 个字节。如果为 double (8 个字节长)，则需要 34560 个字节。所需内存按维数指数增长，一个具有高于三维或四维的程序会很快发现内存不够。

## 7.5 数值初始化

C 允许对数组进行初始化。数组初始化的一般形式和其它变量相类似，如下所示：

```
type_specifier array_name[size1][size2]...[sizeN] = {value_list};
```

value\_list 是用逗号分开的常量表，其类型与数组基本类型一致。第一个常量放在数组的第一个位置上，第二个常量放在数组的第二个位置上，以此类推。注意)后有一个分号。在下面的例子中，用数字 1 到 10 初始化具有 10 个元素的整数数组。

```
int i[10] = {1,2,3,4,5,6,7,8,9,10};
```

这表示 i[0] 的值为 1，i[9] 的值为 10。

用于存储字符数组允许采用下面这种简写形式进行初始化：

```
char array_name[size] = "string";
```

例如，下面这段程序将 str 初始化为 hello：

```
char str[6] = "hello";
```

这种写法算同于

```
char str[6] = {'h','e','l','l','o','\0'};
```

因为 C 中的字符串必须以空字符结束，所以必须确保说明的数组长度足以包括它。这就是为什么“hello”只有五个字符，而 str 为六个字符长度。当使用字符常量时，编译器自动给它加上空终止符。

多维数组的初始化与一数组相同。例如，下面的程序将 str 初始化为数字 1 到 10 和它们的平方数：

```

int sqrs[10][2] = {
    1, 1,
    2, 4,
    3, 9,
    4, 16,
    5, 25,
    6, 36,
    7, 49,
    8, 64,
    9, 81,
    10, 100
};

```

### 7.5.1 变界数组初始化

假设使用数组初始化来建立一个错误信息表，如下所示：

```

char e1[14]="invalid input\n";
char e2[23]="selection out_of_range\n";
char e3[21]="authorization denied\n";

```

可能猜到，人为地计算每个信息中的字符数以决定相应的数组维数是很乏味的。可以让 C 使用变界数组自动计算这个例子中的维数。在这个数组初始化的语句中，如果数组大小未定，那么编译程序将自动产生足够大的数组来存储当前的所有初始值。

使用这种方法，则该信息表变为：

```

char e1[]="invalid input\n";
char e2[]="selection out_of_range\n";
char e3[]="authorization denied\n";

```

除了减少单调以外，变量数组的初始化方法允许改变任何信息，而不必担心会偶然漏掉什么。

变界数组的初始化不仅仅局限于一维数组。对于多维数组，必须规定除最左边的维值以外的其它数值，以便 C 正确地索引数组。按这种方法，你可以建立可变长度表，编译程序自动为它划定足够的空间。例如，sqr 为一个变界数组，其说明如下：

```

int sqrs[][2] = {
    1, 1,
    2, 4,
    3, 9,
    4, 16,
    5, 25,
    6, 36,
    7, 49,
    8, 64,
    9, 81,
    10, 100
};

```

这种方法的优点在于可伸长或缩短，而不用变化数组的维值。

## 7.6 一个水下搜索游戏

二维数组经常用来模拟边界游戏阵，比如国际象棋和棋盘。这里，我们将研究一个简单的水下搜索游戏。

在这个水下搜索程序中，计算机控制一个潜艇，而人控制一个战列舰。这个游戏的目标是一个船只击沉另一个船只。两只战舰都有有限的雷达能力。潜艇能够找到战舰的唯一办法是正好在它的下面。而战舰要发现潜艇，则必须在它的正上方。首先发现另一方的判为胜者，潜艇和战舰在“海洋”中交替移动。

“海洋”是一个只有 3x3 平方英寸的运动场。潜艇和战舰都把它当作水平(X, Y)坐标，左上角的坐标为(0, 0)。二维数组 `matrix` 用于存储游戏边框，其中的每一个元素均被初始化为代表空元素的空格符。表示空元素的空格用于简化显示矩阵的函数，你将看到这一点。

`main()` 函数和全局变量如下所示：

```
/* Battleship VS. Submarine - A Simple Computer Game. */

#include <stdio.h>
#include <stdlib.h>
#include <time.h> /* needed by randomize() */

char matrix[3][3] = {
    ' ', ' ', ' ',
    ' ', ' ', ' ',
    ' ', ' ', ' '
};

int compX, compY, playerX, playerY;

main()
{
    /* randomize the random number generator */
    randomize();

    compX = compY = playerX = playerY = 0;
    for(;;) {
        if(sub_tries()) {
            printf("Submarine wins!\n");
            break;
        }
        if(player_tries()) {
            printf("Battleship (you) win!\n");
            break;
        }
        display_board();
    }
    display_board();
}
```

`randomize()` 函数用来使 C 的随机数生成程序随机化，不久你将看到，该程序用来生成计算机的行动步骤。

你需要的第一个函数是生成计算机行动步骤的函数。计算机（潜艇）通过使用 C 的随机函数生成程序 `rand()` 产生它自己的行动步骤，该函数返回一个 0 到 32,767 的随机数。

(`rand()` 和 `randomize()` 函数都使用头文件 `STDLIB.H`。`randomize()` 同时还需要头文件 `TIME.H`) 每次调用计算机生成一个行动步骤时，生成的随机数用于表示 X 和 Y 坐标。然后这些值经过求模运算，提供一个 0 到 2 之间的数。最后，计算机检验自己是否找到战舰。如果生成的行动步骤与战舰的当前位置坐标相同，则潜艇获胜。如果计算机赢了，函数返回真；否则，返回假。`sub_tries()` 函数如下所示：

```

/* Generate the computer's next move using the random
number
generator.
*/
sub_tries()
{
    matrix[compX][compY] = ' ';

    compX = rand() % 3;
    compY = rand() % 3;

    if(matrix[compX][compY] == 'B')
        return 1; /* submarine won the fight */
    else {
        matrix[compX][compY] = 'S';
        return 0; /* it missed */
    }
}

```

战舰从参赛者那里得到下一个行动步骤。下面这个函数提醒用户输入新坐标并检查是否已找到潜艇。如果找到则参赛者胜，函数返回真。否则，函数返回假。play\_tries()函数如下所示：

```

/* Get the player's next move. */
player_tries()
{
    matrix[playerX][playerY] = ' ';

    do {
        printf("Enter new coordinates (X,Y): ");
        scanf("%d%d", &playerX, &playerY);
    } while(playerX < 0 || playerX > 2 || playerY < 0 ||
        playerY > 2);

    if(matrix[playerX][playerY] == 'S')
        return 1; /* battleship won the fight */
    else {
        matrix[playerX][playerY] = 'B';
        return 0; /* it missed */
    }
}

```

每一步之后，display\_board()函数将显示游戏边框。未使用的单元为空白，存储潜艇最后一个位置的框包含 S，存储战舰最后一个位置的边框包含了 B。

```

/* Display the playing board. */
display_board()
{
    printf("\n");

    printf("%c | %c | %c\n", matrix[0][0],
        matrix[0][1], matrix[0][2]);
    printf("----|----|----\n");
    printf("%c | %c | %c\n", matrix[1][0],
        matrix[1][1], matrix[1][2]);
    printf("----|----|----\n");
    printf("%c | %c | %c\n", matrix[2][0],
        matrix[2][1], matrix[2][2]);
}

```

整个的水下搜索程序如下所示。现在，可以输入这个程序并进行调试。经过思考以后，应该使它更加完善。

```

/* Battleship VS. Submarine - A Simple Computer Game. */

#include <stdio.h>
#include <stdlib.h>
#include <time.h> /* needed by randomize() */

char matrix[3][3] = {
    { ' ', ' ', ' ' },
    { ' ', ' ', ' ' },
    { ' ', ' ', ' ' }
};

int compX, compY, playerX, playerY;

main()
{
    /* randomize the random number generator */
    randomize();

    compX = compY = playerX = playerY = 0;
    for(;;) {
        if(sub_tries()) {
            printf("Submarine wins!\n");
            break;
        }
        if(player_tries()) {
            printf("Battleship (you) win!\n");
            break;
        }
        display_board();
    } display_board();

    /* Generate the computer's next move using the random
       number generator.
    */
    sub_tries()
    {
        matrix[compX][compY] = ' ';

        compX = rand() % 3;
        compY = rand() % 3;

        if(matrix[compX][compY] == 'B')
            return 1; /* submarine won the fight */
        else {
            matrix[compX][compY] = 'S';
            return 0; /* it missed */
        }
    }

    /* Get the player's next move. */
    player_tries()
    {
        matrix[playerX][playerY] = ' ';
        do {
            printf("Enter new coordinates (X,Y): ");
            scanf("%d%d", &playerX, &playerY);
        } while(playerX < 0 || playerX > 2 || playerY < 0 ||
            playerY > 2);
    }
}

```

```

        if(matrix[playerX][playerY] == 'S')
            return 1; /* battleship won the fight */
        else {
            matrix[playerX][playerY] = 'B';
            return 0; /* it missed */
        }
    }

    /* Display the playing board. */
    display_board()
    {
        printf("\n");

        printf("%c | %c | %c\n", matrix[0][0],
               matrix[0][1], matrix[0][2]);
        printf("----|----|----\n");
        printf("%c | %c | %c\n", matrix[1][0],
               matrix[1][1], matrix[1][2]);
        printf("----|----|----\n");
        printf("%c | %c | %c\n", matrix[2][0],
               matrix[2][1], matrix[2][2]);
    }
}

```

现在你已经掌握了数组，该学习 C 的另外一个重要的特性：指针了。

## 第八章 指针

理解和正确使用指针是编写最成功 C 程序的关键。原因有三点：首先，指针提供了一种函数修改其调用自变量的方法。第二，指针可用于支持 C 的动态分配例程。第三，指针可在任何地方代替数组——这样就大大提高了效率。同时，C++ 的很多特性完全依赖于指针，因此对指针的透彻理解非常重要。

指针是 C 具有的最强有力的特性，但它也是最具危险性的。例如，未初始化的或野指针可能导致系统崩溃。更糟的是，使用指针很容易产生错误，而且很难发现。

正是由于指针的重要性和潜在的弊病，本章将对它作详细讨论。

### 8.1 指针是地址

一个指针是存储另一个对象内存地址的变量。最通常的情况，这个地址是另一个变量在内存中的地址，尽管它可以是端口或专用的 RAM 的地址，如视频缓冲区。如果一个变量包含了另一个变量的地址，那么可以说是第一个指向第二个。这种情况说明如图 8-1。

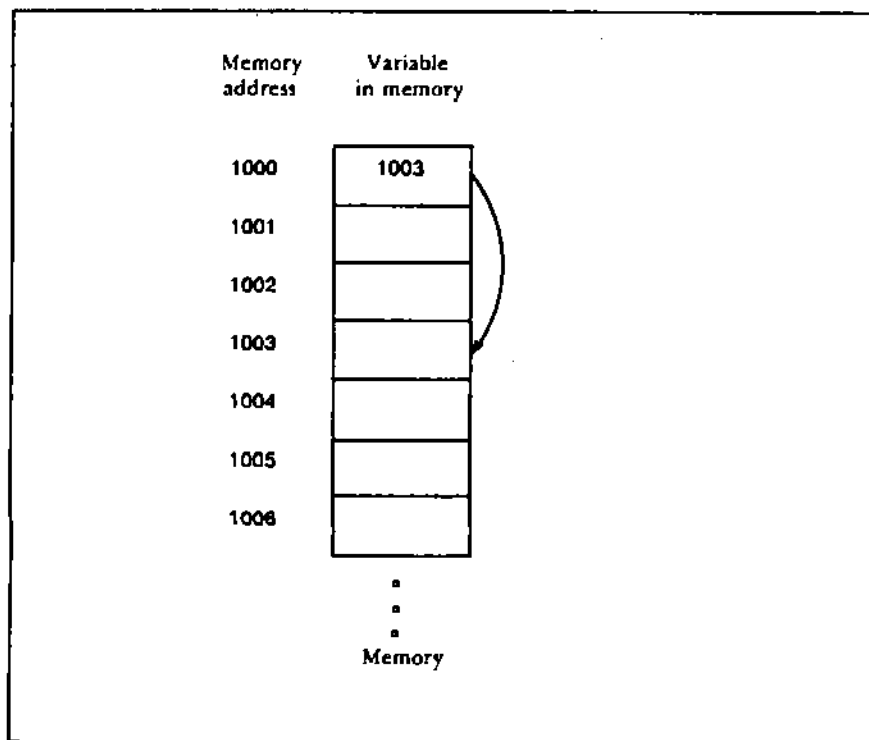


图 8-1 指向另一个变量的变量

### 8.2 指针变量

如果一个变量要存储一个指针，那么它必须进行说明。指针说明由一个基本类型、一个\*以及变量名组成，它的一般形式为。



```
type *var_name;
```

type 可以是任何有效的 C 类型, var\_name 为指针变量名。指针基本类型定义了指针可以指向的变量的类型。例如, 下面这些语句说明了一个字符型指针和两个整型指针。

```
char *p;  
int *temp, *start;
```

### 8.3 指针操作符

有两个指针操作符: &和\*。&是一个单目操作符, 返回运算对象的内存地址。(一个单目操作符只要求一个运算对象。)例如:

```
count_addr=&count;
```

将变量 count 的内存地址存放在 count\_addr 中。这个地址是变量的计算机内部地址, 它与 count 的值毫无关系。

由于&操作符返回变量的地址, 所以上条赋值语句的意思是“将 count 的地址赋给 count\_addr”

为了能够更好地理解上述赋值语句, 可以假设变量 count 的地址为 2000, 赋值后, count\_addr 将变为 2000。

另一个单目操作符为\*, 它与&互补, 即它返回指针所指处的变量值。比如, 假设 count\_addr 保存变量 count 的内存地址, 那么语句:

```
val=*count_addr;
```

将把 count 的值赋给 val。如果 count 的初始值为 100, 则 val 的值为 100, 因为 100 被存在分配给 count\_value 的内存地址 2000 处。

\*也称为“间接引用操作符”, 上条赋值语句可理解为“将 count\_addr 处的值赋给 val”。

指针操作符\*与乘法符号相同, 这常常使得 C 的初学者搞不清楚。

指针操作符&和\*之间并没有联系, 它们的优先级与单目减相同, 但比其它所有算术操作符高。

以下是一个使用\*和&的程序, 它在屏幕上显示 100。

```
#include <stdio.h>  
main()  
{  
    int*count_addr, count, val;  
    count=100;  
    count_addr=&count; /*取 count 和地址*/  
    val=*count_addr; /*取此地址中的值*/  
    printf( "%d val" ,); /*显示 100*/  
}
```

#### 基本类型是重要的

在前面的说明中, 我们已经看到, 可以通过指向 count 的指针间接地把 count 的值赋给 val。那么编译程序如何知道从 count\_addr 所指地址起送多少字节给 val 呢?更一般地讲, 对于使用了指针的赋值语句, 编译程序如何传送适当的字节数呢?答案在于指针的基本类

型，它决定了编译程序假定的指针所指向的数据类型。如在上面的例子中，因为 `count` 是 `int` 型指针，所以要从 `count_addr` 所指地址传送两个字节给 `VAL`。如果 `count_addr` 是 `double` 型指针，则要传送 8 个字节。

必须保证指针变量总是指向正确的数据类型。如当说明指针是 `int` 型时，编译程序就假定指针保存的任何地址均指向 `int` 型变量。由于 C 语言允许把任意一个地址赋给指针变量，所以以下有错误的代码仍能通过编译，但 Turbo C++ 将显示警告信息：

```
#include <stdio.h>
/*This program will not work correctly.*/
main ( )
{
    float x=10.1, y;
    int *p;
    p = &x;
    x = *p;
    printf ( "%d,Y ");
}
```

## 8.4 指针表达式

通常情况下，包含指针的表达式同其它 C 表达式遵循同样的规则。在这一节中，我们将研究指针表达式的一些特殊情况。

### 8.4.1 指针赋值

同任何变量一样，把一个指针放在赋值语句的右边，将其值赋给了另一个变量。例如，

```
#include <stdio.h>

main()
{
    int x;
    int *p1, *p2;

    x = 101;

    p1 = &x;
    p2 = p1;

    /* print the hexadecimal value of the
       address of x -- not x's value! */
    printf("at location %p ", p2);

    /* now print x's value */
    printf("is the value %d\n", *p2);
}
```

`x` 的地址以十六进制显示，所使用的是 `printf()` 的另一种格式符号。`%p` 指明指针地址将以十六进制形式显示。

### 8.4.2 指针运算

有两个指针操作符：递增和递减。为弄清指针算法的内容，假设 `p1` 为指向一个整数的指针，当前值为 2000。经过下面的表达式计算之后：

`p1++;`

`p1` 的值为 2002，而不是 2001。每次 `p1` 都增值，指向下一个整数。同样，递减也是这样。例如：

`p1--;`

将使 `P1` 的值为 1998，假定其原值为 2000。

每当指针增值时，它将指向其基本类型的下一个元素的内存地址。每当指针递减时，将指向前一个元素的地址。这样，指针为字符型时，似乎为“正常”算法。无论怎样，其它的所有指针均将增加或减少所指数据的长度。例如，假定单字节字符和双字节整数，当一个字符指针递增时，其值加 1。但为一个整数指针递增时，其值加 2。图 8-2 说明了这个概念。

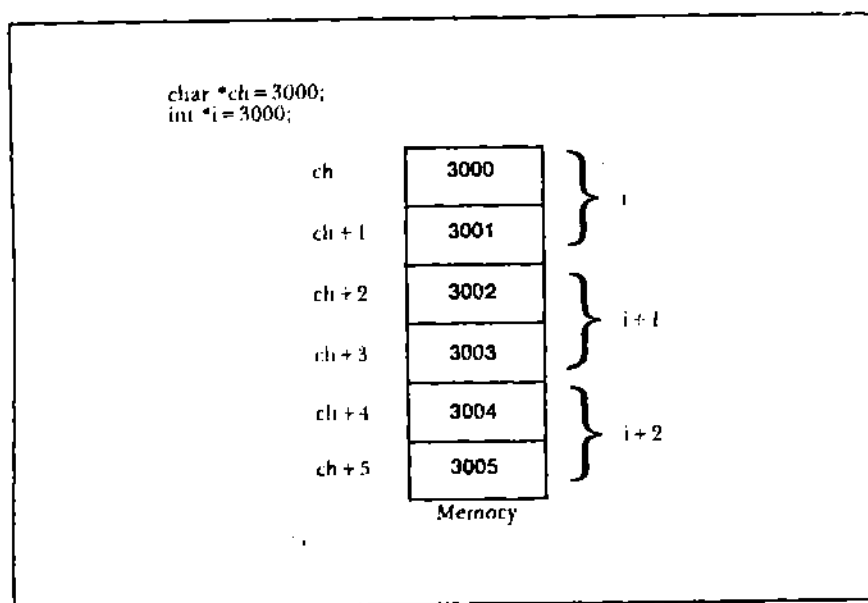


图 8-2 指针运算与基类型的关系

无论怎样，你不要仅局限于递增和递减。指针还可以加上或减去一个整数。表达式：

`p1 = p1 + 9;`

将使 `p1` 指向它当前所指基本类型后的第九个元素。

你可以从一个指针中减去另一个指针。如果两者都指向同一数组中的不同元素，那么相减的结果与所指元素的下标相减的结果相同。

除了指针和整数相加减、指针和指针相减以外，不允许对指针进行其它算术操作。特别注意，不允许：

- 指针间相乘或相除
- 两个指针相加
- 对指针使用位变换和屏蔽操作符

## . float 和 double 类型的指针相加减

### 8.4.3 指针比较

在关系表达式中允许对两个指针进行比较。例如，已知两个指针 *p* 和 *q*，下面的语句是完全正确的：

```
if(p<q) printf("p points to lower memory than q\n");
```

通常，指针比较仅用于指向共同对象的两个或多个指针。

### 8.5 指针和数组

指针和数组的关系非常密切。考虑下面的程序段：

```
char str[80], *p1;
```

```
p1 = str;
```

这里，将 *p1* 设置为 *str* 中的第一个数组元素的地址。在 C 中，不带下标的数组名为数组的起始地址。本质上，它是指向数组的指针。指向数组 *str* 第一个元素的指针还可以这样表示：

```
p1 = &str[0];
```

但多数程序员认为这种形式很不好。

如果你希望访问 *str* 中的第五个元素，可以这样写：

```
str[4]
```

或  $*(p1+4)$

两个语句都将返回第五个元素。

记住：数组从 0 开始，所以 *str* 的下标为 4。还可以将指针 *p1* 加 4，以存取第五个元素，因为 *p1* 当前指向 *str* 的第一个元素。

本质上，C 允许两种存取数组元素的方法。这很重要，因为指针运算要比数组下标运行快些。由于运行速度是编程中经常需要考虑的因素，因此在 C 程序中，使用指针来存取数组元素的方法就很流行了。

让我们看一个说明如何在数组下标运算中使用指针的例子，比较这两个程序——一个使用下标，一个使用指针，两者都用小写字母显示字符串的内容。

```
#include <stdio.h>
#include <ctype.h>

/* array version */
main()
{
    char str[80];
    int i;

    printf("enter a string in uppercase: ");
    gets(str);

    printf("here's the string in lowercase: ");

    for(i=0; str[i]; i++) printf("Zc", tolower(str[i]));
}
```

```

#include <stdio.h>
#include <ctype.h>

/* pointer version */
main()
{
    char str[80], *p;

    printf("enter a string in uppercase: ");
    gets(str);

    printf("here's the string in lowercase: ");

    p = str; /* get the address of str */
    while(*p) printf("%c", tolower(*p++));
}

```

数组程序要比指针程序慢，原因是索引数组的时间比使用\*操作所花时间长。

有时，C的初学者会产生这样的错误，即认为不应使用数组索引，因为指针效率非常高。但并非如此。如果严格按上升或下降顺序访问数组，则指针的使用显得更快且容易一些。但如果是随机访问数组，则数组索引更好，因为它通常与计算一个复杂的指针表达式一样快，而且易于译码和理解。还有，如果使用数组索引，可以使编译程序帮你做一些事情。

#### 8.5.1 索引指针

在C中，有时要象索引数组一样索引指针。这就深刻阐明了指针和数组间的紧密联系。例如，下面的程序段是非常正确的，在屏幕上打印从1到5各个数字：

```

/* Indexing a pointer like an array. */

#include <stdio.h>

main()
{
    int i[5] = {1, 2, 3, 4, 5};
    int *p, t;

    p = i;

    for(t=0; t<5; t++) printf("%d ", p[t]);
}

```

在C中，语句 `p[t]` 等同于 `*(p+t)`。

#### 8.5.2 指针和字符串

由于没有下标的数组名是一个指向数组的第一个元素的指针，当你使用在前面章节中讨论过的字符串函数时，只有一个指向字符串的指针被传递到函数中，而不是字符串本身。为理解这一过程。这里有编写 `strlen()` 函数的一种方法：

```

strlen(char *s)
{
    int l=0;
    while (*s) {
        l++;
    }
}

```

```

        S++;
    }
    return l;
}

```

记住，C 中的所有字符串都从空终止符结束，其值为假。因此，这样的语句：

```
while (*s)
```

在未达到字符串末尾之前为真。这里，如果调用的字符串长度为 0，则返回 0，否则，返回字符串长度。

你可能已对 `strlen()` 可用常量作为自变量感到怀疑。例如，你可能不明白下面这段程序是如何工作的。

```
printf("length of TEST is %d", strlen("TEST"));
```

方案是，当使用一个字符串常量作为自变量时，只有一个指针传递到 `strlen()`。实际字符串由 Turbo C++ 自动存储。

更一般的是，当一个字符串常量用于任何类型的表达式中时，把它处理为一个指向字符串的第一个字符的指针。例如，下面的程序很正确，在屏幕上打印词组 "this program works"：

```

#include <stdio.h>

main()
{
    char *s;

    s = "this program works";

    printf(s);
}

```

组成字符串常量的字符存储在由 Turbo C++ 存储的特别“字符串表”中。用户程序只使用一个指针指向该表即可。

### 8.5.3 如何得到一个数组元素的地址

到目前为止，所有的例子都集中在把数组的起始地址赋给一个指针。然而，还有可能使用 `&` 将数组的某个特定元素的地址赋给指针。例如，这个程序段将 `x` 的第三个元素地址放在 `p` 中：

```
p=&x[2];
```

这种练习特别用于查找一个子字符串。例如，下面这个程序将打印从键盘键入的字符串中，第一个空格出现后的剩余部分：

```

#include <stdio.h>

/* Display the string left after the first space
   is encountered. */
main()
{
    char s[80];
    char *p;
    int i;

    printf("enter a string: ");

```

```

    gets(s);

    /* find first space or end of string */
    for(i=0; s[i] && s[i]!=' '; i++) ;

    p = &s[i];

    printf(p);
}

```

因为 `p` 将指向一个空格或空(如果字符串中没有空格)。如果指向空格,则打印空格后字符串的剩余部分,如果指向空字符,则 `printf()` 不打印。例如,如果键入“hi there”,则显示“there”。

#### 8.5.4 指针数组

指针可以组成任何数据类型的数组。如大小为 10, `int` 型的指针数组说明形式为:

```
int* x[10];
```

为把一个整型变量 `VAR` 的地址赋给指针数组的第三个元素,可以这样写:

```
x[2] = & var;
```

为得到 `var` 的值,可从这样写:

```
*x[2]
```

指针数组通常用于存储表示错误信息的指针。可以生成一个输出信息的函数,给出它的代码数字,如下 `serror()` 所示:

```

char *err[] = {
    "cannot open file\n",
    "read error\n",
    "write error\n",
    "media failure\n"
};

serror(int num)
{
    printf("%s",err[num]);
}

```

可以看到,在 `serror()` 中调用的 `printf()` 带有一个字符型指针,它是指向一个由传递给函数的错误号码索引出的错误信息。例如,如果传递给 `num` 的号码为 2,则显示信息 `write error`。

另一个有趣的初始化的字符指针数组应用程序使用了 C 的 `system()` 函数,让程序发送给操作系统一个命令。`system()` 的调用方式如下所示:

```
system("command");
```

这里, `command` 为将要执行的操作系统的命令。例如,在 DOS 环境下,下面的语句将显示缺省的目录内容:

```
system ("dir");
```

下面的程序实现了一个非常小的操纵菜单用户接口,可执行四个 DOS 命令: `dir`, `chkdsk`, `time` 和 `date`。

```

/* A very simple menu-driven DOS user interface. */
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

main()
{
    /* create an array of strings */
    char *command[] = {
        "DIR",
        "CHKDSK",
        "TIME",
        "DATE"
    };
    char ch;

    for(;;) {
        do {
            printf("1: directory\n");
            printf("2: check the disk\n");
            printf("3: set time\n");
            printf("4: set date\n");
            printf("5: quit\n");
            printf("\nselection: ");
            ch = getch();
            printf("\n");
        } while((ch < '1') || (ch > '5'));
        if(ch == '5') break; /* end */

        /* execute the specified command */
        system(command[ch - '1']);
    }
}

```

你会发现扩展这个程序，以便用户使用更多的 DOS 命令，非常有趣。例如，可以加入 copy 命令，并且当选择它时，提示用户输入源文件名和目标文件名。copy 命令和文件名可连接以形成字符串传递给 DOS。

### 8.5.5 一个使用数组和指针的有趣实例

为进一步阐明数组和指针间的相互联系，这里开发一个非常简单的从英文到德文的转换程序。该程序使用一个二维的字符串数组，提供了一个很小的英文和德文单词的对应表。它先让你输入一个英文句子，然后输出一个非常粗略的德文译文(德文中如果性别不同，用词也不同，该程序忽略了这一点。而且，也没有完成动词的变化。)

首先需要——德文对照表，如下所示，你可以随意扩展：

```

char trans[][20] = {
    "is", "ist",
    "this", "das",
    "not", "nicht",
    "a", "ein",
    "book", "Buch",
    "apple", "Apfel",
    "I", "Ich",
    "bread", "Brot",
    "drive", "fahren",
    "to", "zu",

```



```

        "buy", "kaufen",
        "", ""
    };
};

```

每个英文单词与德文单词成对出现。注意最长的单词不能超过 19 个字符。

该翻译程序的 main( ) 函数和所使用的全局变量如下所示:

```

char input[80];
char word[80];
char *p;

main()
{
    int loc;

    printf("Enter English sentence: ");
    gets(input);
    p = input; /* give p the address of the array input */
    printf("Rough German translation: ");

    get_word(); /* get the first word */

    /* This is the main loop. It reads a word at a time
       from the input array and translates each word
       into German.
    */
    do {
        /* find the index of the English word in trans */
        loc = lookup(word);

        /* printf the German if a match is found */
        if(loc != -1) printf("%s ", trans[loc+1]);
        else printf("<unknown> ");

        get_word(); /* get next word */
    } while(*word); /* repeat until a null string is returned */
}

```

该程序的操作过程如下: 首先, 提示用户输入一个英文句子, 读到 input 字符串中。指针 P 则被赋予 input 的起始地址。get\_word( ) 使用这个指针每次从 input 字符串中读取一个单词。每个单词都存入 word 数组中。然后主循环用 lookup( ) 函数访问每个单词, 它返回英文单词的编号或 -1, 表示该英文单词不在表中。下标加 1, 即可查到相应的德文。

lookup 函数如下所示:

```

/* This function returns the location of a match
   between the string pointed to by s and the trans
   array.
*/
lookup(char *s)
{
    int i;

    for(i=0; *trans[i]; i++)
        if(!strcmp(trans[i], s)) break;

    if(*trans[i]) return i;
    else return -1;
}

```

调用 `lookup()` 函数, 指针指向英文单词, 如果该单词在表中, 则返回其下标。如果在表中没有发现这个单词, 则返回-1。

`get_word()` 函数如下所示。就这个问题而言, 单词只能由空格和终止符定界。

```
/* This function will read the next word from the input
   array. Each word is assumed to be separated by a space
   or the null terminator. No other punctuation is allowed.
   The word returned will be a null length string when the
   end of the input string is reached.
*/
get_word()
{
    char *q;

    /* reload address of word each time function is called */
    q = word;

    /* get the next word */
    while(*p && *p != ' ') {
        *q = *p;
        p++;
        q++;
    }
    if(*p == ' ') p++;
    *q = '\0'; /* null terminate each word */
}
```

从 `get_word()` 返回后, 全局变量 `word` 包含句子中下一个单词或者为空。

整个的翻译程序如下所示:

```
/* A (very) simple English to German Translator. */
#include <stdio.h>
#include <string.h>

char trans[][20] = {
    "is", "ist",
    "this", "das",
    "not", "nicht",
    "a", "ein",
    "book", "Buch",
    "apple", "Apfel",
    "I", "Ich",
    "bread", "Brot",
    "drive", "fahren",
    "to", "zu",
    "buy", "kaufen",
    ""
};

char input[80];
char word[30];
char *p;

main()
{
    int loc;

    printf("Enter English sentence: ");
    gets(input);
    p = input; /* give p the address of the array input */
    printf("Rough German translation: ");
```

```

get_word(); /* get the first word */

/* This is the main loop. It reads a word at a time
   from the input array and translates each word
   into German.
*/
do {
    /* find the index of the English word in trans */
    loc = lookup(word);

    /* printf the German if a match is found */
    if(loc!=-1) printf("%s ", trans[loc+1]);

    else printf("<unknown> ");

    get_word(); /* get next word */
} while(*word); /* repeat until a null string returned */
}

/* This function returns the location of a match
   between the string pointed to by s and the trans
   array.
*/
lookup(char *s)
{
    int i;

    for(i=0; *trans[i]; i++)
        if(!strcmp(trans[i], s)) break;

    if(*trans[i]) return i;
    else return -1;
}

/* This function will read the next word from the input
   array. Each word is assumed to be separated by a space
   or the null terminator. No other punctuation is allowed.
   The word returned will be a null length string when the
   end of the input string is reached.
*/
get_word()
{
    char *q;

    /* reload address of word each time function is called */
    q = word;

    /* get the next word */
    while(*p && *p!=' ') {
        *q = *p;
        p++;
        q++;
    }
    if(*p==' ') p++;
    *q = '\0'; /* null terminate each word */
}

```

举个翻译的例子，如果键入“I drive to buy a book”，则程序回答为“Ich fahren zu kaufen ein Buch”这不合乎正确的德语语法，但大约是这样的。

你应该研究一下这个程序中指针和数组的相互作用。有一点应该牢记，trans[0]等同

于指向表中第一个单元的指针, `trans[1]` 等同于指向表中第二单元的指针, 以此类推。另外, `trans` 数组实际上是指向说明部分中的字符串的指针数组。字符串本身存储在 Turbo C++ 的字符串表中。

## 8.6 指针的指针

指针数组的概念易于理解, 因为下标使其意义显得清晰。但是, 指针的指针却很容易混乱。

指向指针的指针是一种“多重间接”的形式, 或者为一个指针链。在图 8.3 中你可以看到, 如果是一个标准指针, 指针的值为包含所需数值的变量的地址。如果是指针的指针, 第一个指针包含第二个指针的地址, 第二个指针指向某个变量, 该变量包含所需的数值。

多重间接无论在多大的范围内都能进行下去, 但很少需要多个指针指向一个指针, 或者真正地灵活使用。过分的间接难以理解而且容易犯概念性的错误(不要将多重间接与链接表混淆, 链接表用于数据库及类似的地方。)

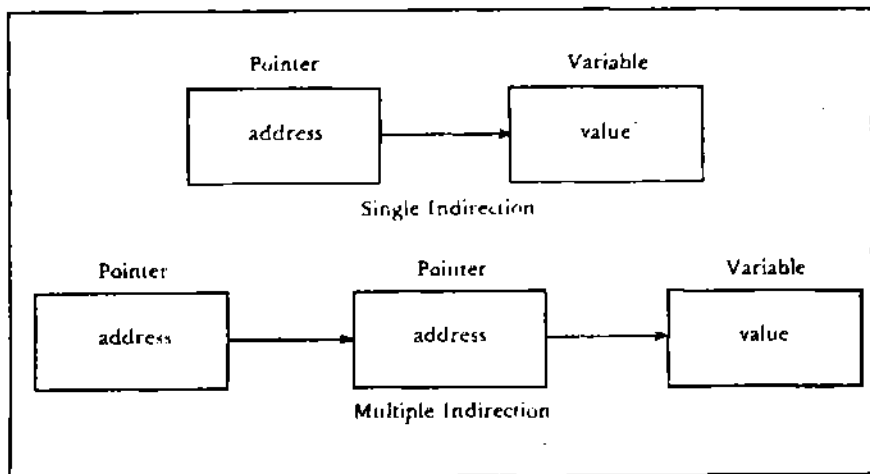


图 8.3 单一和多重指针

一个指向指针的指针变量必须按下面的方法说明。即把附加星号放在它的名字前面。例如, 这个说明告诉编译程序, `newbalance` 是一个指向 `float` 型指针的指针:

```
float **newbalance;
```

重点理解 `newbalance` 不是一个指向浮点数的指针而是指向浮点指针的指针。

为通过指针的指针间接访问目标值, 必须使用两个星号操作符, 如下面这个小例子所示:

```
#include <stdio.h>

main()
{
    int x, *p, **q;

    x = 10;
    p = &x;
    q = &p;
```

```
    printf("%d", **q); /* print the value of x */
}
```

这里，`p` 被说明为一个指向整数指针，`q` 为一个指向这个类型指针的指针。调用 `printf` 将在屏幕上打印数字 10。

### 3.7 初始化指针

指针在说明之后，赋值之前，它包含的值未知。如果你想在赋值前使用指针，不仅你的程序，甚至操作系统也会遭到破坏——一种非常危险的错误类型。

按规定，当前不指向任何事物的指针应定义其值为空，以表示它无所指。但正因为指针有空值而使它变得不“安全”。如果在一个赋值语句的右边使用一个空指针，将一直冒着毁坏程序和操作系统的危险。

空值指针可使用户程序易于编码且能提高效率。例如，可以使用一个空指针标志指针数组的结尾。这样，访问该数组的程序遇到空值时，就知道已到达数组的末尾了。下面显示的 `for` 循环说明了这个方法：

```
/*look up a name assuming last element of p
is null */
for(t=0;p[t];++t)
    if(! strcmp(p[t], name)) break;
```

该循环将执行到发现匹配或遇到空指针的时候。因为数组末端以空终止符标志，所以当到达末端时，控制循环的条件将失败。

在编写专业性 C 程序中，初始化指针是最一般的训练。在本章前面有关指针数组的部分中你已经看到了两个例子。这方面的另一个变化是如下所示的字符符说明的类型：

```
char *p="hello world\n";
```

你已看到，指针 `P` 不是一个数组。原因是这种初始化的工作不得不与 C 处理字符串常量的方法相联系。值得一提的是。Turbo C++ 生成了由编译程序内部使用的字符串常量。因此，这个说明语句把存放在字符串表中的字符串“hello world”的地址存放在指针 `P` 中。在整个程序中，可以像使用其它任意字符串一样使用 `p`。例如，下面的程序非常正确：

```
#include <stdio.h>
#include <string.h>

char *p="hello world";

main()
{
    register int t;

    /* print the string forward end backwards */
    printf(p);
    for(t=strlen(p)-1; t>=0; t--) printf("%c", p[t]);
}
```

然而，用户程序不应通过 `p` 给字符串表赋值，否则表会变得混乱不堪。

## 8.8 指针的一些问题

没有比“野”指针给你带来的麻烦更大的事情了!指针非常复杂。对某些问题,指针给你带来巨大的力量和必要的帮助。但当一个指针偶然包含了一个错误时,则最难发现和追踪。

一个错误指针的故障很难找到,因为指针本身不是问题所在,问题在于你每次使用指针完成一种操作,都是到未知内存区域读取或写入。如果从中读取,最糟糕的是碰到无用数据。而当写入的时候,将覆盖掉原来的代码或数据。这个只有在程序执行时才能发现,而且会使你在错误中寻找错误。这个错误常常可使程序员屡次失眠。

因为指针错误如此难缠,你应尽最大努力不造这种错误!这里讨论一些共同的错误,以一个指针错误的典型例子开始:未初始化指针。考虑下面的程序:

```
/* This program is wrong. Do Not Execute. */
main()
{
    int x, *p;

    x = 10;
    *p = x;
}
```

这个程序把数值 10 赋给某个未知的内存位置。指针 P 未被赋给任何一个值;因此,它包含一个无用数据。尽管 Turbo C++ 将对本例中的错误发出警告信息,但当一个指针简单地指向该错误地点时,还要出现同样的错误。例如,你可能偶然赋给指针一个错误地址。这种类型经常在你的程序很小时,不被留心注意,因为其差别是 P 包含一个“安全”地址——不在用户代码、数据区或者操作系统中的,然而,随着程序的增大, p 指向某个致命错误的可能性也随之增长。最后,程序停止工作。为避免这些麻烦,通常办法是在指针使用前,让它指向某个有效数据。

常犯的第二种错误是对如何使用指针的误解。考虑下面这个程序:

```
#include <stdio.h>

/* This program is wrong. Do Not Execute. */
main()
{
    int x, *p;

    x = 10;
    p = x;
    printf("%d", *p);
}
```

调用 printf() 不能在屏幕上打印 x 的值,等于 10。所打印的是一些未知值。原因是赋值语句

**p = x;**

是错误的。这个语句将数值 10 赋给指针 p, 它已假定包含一个地址, 而不是值。这样写才能保证程序正确:

**p = &x;**

在这个简单的例子中, Turbo C++ 将警告你程序中的错误。但并不是所有这种类型的

错误都可由编译程序发现。

不要仅仅因为处理不正确 将会引起非常复杂的错误，就不使用指针了。你应该仔细一些，使用前就应保证知道指针指向哪里。

由于已经对指针有了基本的了解，所以应准备解决 C++ 函数中的精华部分了。

## 第九章 函数：更详尽的说明

函数是 C 语言的构造模块。基于在第四章中提到的总述,你已经能够或多或少地用直观的方法使用函数。在这一章你将学到函数的细节,学习如何构造一些函数来改变它们的参数,如何返回不同类型的数据,以及如何使用函数原型。除此之外,那些作用域规则和变量生存期也在研究之列,本章还将讨论如何构造递归函数和一些有关 `main()` 的特殊性质。

### 9.1 函数的一般形式

函数的一般形式为:

类型定义符 函数名(参数说明)

```
{  
    函数体  
}
```

类型定义符定义了函数通过 `return` 语句返回的数据类型,它可以是任何一种有效的类型。如果类型定义缺省,函数将返回一个整型量。先前你用过的这方面的程序就属于这种情况的实际使用(稍后在本章里将学习到如何返回不同类型的值)。参数表是以逗号分隔的变量名称表,它接收函数调用时的参数值。一个函数可以不带任何参数,在这种情况下,参数表将是空的。但是,即使没有参数,括号仍然是必要的。

懂得函数的参数说明表是很重要的事情。不像变量说明那样,可以用逗号分隔的变量名称表来说明许多具有相同类型的变量,所有函数的参数必须包含其类型和变量名。就是说,一个函数的变量说明表一般采用这种形式:

```
f(类型 变量名 1, 类型 变量名 2,...,  
   类型 变量名 N)
```

例如,这是一个正确的函数说明:

```
f(int x,int y,float z)
```

下面的说明是不正确的,因为每个参数必须有自己的类型。

```
f(int x,y,float z)
```

### 9.2 return 语句

尽管 `return` 语句在第四章作为从一个函数返回一个值的手段已提到过,在这里它将被更细地研究。

`return` 语句有两个重要用途。首先,它能立即从所在函数中退出。也就是 `return` 语句可以在程序出错时返回到调用点。其次,它可以用于返回一个值。下面将讨论这两种用途。

#### 9.2.1 从一个函数中返回

有两种方法可以终止函数的运行,并返回到调用它的调用语句。第一种方法是,执行



到函数的最后一条语句，具体地说，当遇到了函数的结束符号“}”后即刻返回(当然，实行上花括号不会出现在目标代码中，但是你可以认为它采用的是这种方法)。例如，下一个函数反向显示字符串：

```
void pr_reverse(char*s)
{
    register int t;
    for(t=strlen(s)-1,t>=0;t--) printf("%c",s[t]);
}
```

当字符串显示完后，函数中无其它可执行语句，就返回到调用点。

第二种方法是使用 return 语句。使用该语句可以不带任何值。如下个函数显示正数的正整数次幂，如果指数为负，则在 return 语句处就返回，且没有返回值。

```
void power(int base,int exp)
{
    int i;
    if(exp<0) return;      /*负指数不行*/
    i=1;
    for(; exp; exp--)
        i=base*i;
    printf("the answer is:%d.",i);
}
```

### 9.2.2 返回值

想要从函数中返回值，必须在 return 语句后面加上要返回的值，如下个函数返回两个实参中的较大数：

```
max(int a,int b)
{
    int temp;
    if(a>b) temp=a;
    else temp=b;
    return temp;
}
```

注意，由于函数说明中没有明确说明返回类型，因此返回缺省类型即为整型值。

在一个函数中可以包含两个以上的 return 语句。有时候，为了简化函数并使某些算法更加有效，经常用到多个 return 语句，如下面的 max() 函数就比上一个精炼：

```
max(int a ,int b)
{
    if (a<b) return a;
    else return b;
}
```

下面的 find\_substr() 函数使用了两个 return 语句。如果没有找到相匹配的子串，函数

返回-1, 否则, 返回字符串内子串的起始下标。如果不使用第二个 return 语句, 则需增加一个临时变量和一条赋值语句:

```
find_substr(char *srb, char *str)
{
    register int t;
    char *p, *p2
    for(t=0;str[t];t++) { /*get starting point*/
        p=&str[t];
        p2=srb;
        while(*p2 && *p2==*p) { /*while equal advance*/
            p++; p2++; /*through the string*/
            p++;p2++;
        }
        if (!*p2) return t; /*if at the end of sub then match*/
    }
    return -1;
}
```

如果这个函数用包括“ere”的子串调用且其主要字符串包含“Hi there,”那么此函数返回值 5。

尽管你刚看到的两个例子是比单个 return 语句的用处多的有效的函数, 但必须说明一点, 过多的 return 语句会破坏函数并且导致语义混乱。建议最好在非常有必要时使用多个 return 语句。

除了 void 类型外, 所有函数都返回一个值(用 void 说明的函数将在下节讨论。)这个值由 return 语句明确地指出, 或者如果没有 return 语句则为未知。这就意味着函数可以在有效的 C 语言表达式中作为操作数。因而, 在 C 和 C++ 中, 下面的几个表达式都是有效的:

```
x = abs(y);
if(max(x,y) > 100) printf("greater");
for(ch=getchar(); isdigit(ch);) ... ;
```

然而, 函数不能是赋值语句的目标。像下面的语句是错误的。

```
swap(x,y)=100; /*incorrect statement*/
```

Turbo C++ 将会显示出错信息, 而且不会对有这种语句的程序进行编译。

虽然除了 Void 型以外的所有函数都有返回值, 但在你编制程序时, 函数一般有三种类型。第一种只做简单的计算, 它们专门用于对一系列的参数作运算, 并且基于该运算返回一个值——这实质上是“纯”函数。这种类型的函数例如库函数 sqrt() 和 sin(), 将各自返回一个数的平方根和正弦值。

第二种函数类型产生信息并返回一个值用于简单标明操作的成功与失败。fwrite() 就是一个例子, 它用于往磁盘文件中写信息。如果写操作成功, fwrite() 返回所写入的字节数; 其他值表明发生了错误(将在下一章学习文件 I/O。)

最后一种类型的函数没有明确的返回值。本质上说来, 这种函数是一个单纯的过程,

并且不产生数值。由于一些难以说明的历史性原因,有些函数确实不产生令人感兴趣的结果,但它们还是经常返回一些值。例如, `printf()` 返回所写入的字符数量,然而程序中很少用到这一返回值。因此,虽然所有函数(除 `void` 类型之外)都有返回值,而你并不一定要用到它们。关于函数返回值的一个普遍的问题是,“如果返回了值,一定要把它赋给其它变量吗?”

```
#include <stdio.h>

main()
{
    int x, y, z;

    x = 10;    y = 20;
    z = mul(x, y);    /* 1 */
    printf("Zd", mul(x, y));    /* 2 */
    mul(x, y);    /* 3 */
}

mul(int a, int b)
{
    return a*b;
}
```

第一行中 `mul()` 返回值赋给了 `z`。第二行中返回值没有赋给别的变量,然而它被 `printf()` 函数用到了。最后,第三行中返回值被丢弃了,因为它既没有赋值给其它变量,也没有用作表达式的一部分。

### 9.3 函数返回非整型值

当一个函数的类型没有明确说明时,它的缺省值为 `int` 型。对多数函数来说这个缺省还不错。但是当需要返回不同类型的数据时,要求两个步骤处理。首先,必须在函数被初次调用之前确定函数类型。只有用这种方法, Turbo C++ 才能产生正确的代码,以保证函数能返回非整型值。

函数经过适当的说明后可以返回 C 语言中任何有效的数据类型。函数的说明方法与变量说明相似:类型说明符放在函数名的前面。类型说明符告诉编译程序函数要返回的数据类型。如果想使程序正确运行,这个信息是至关重要的,因为不同的数据类型要求有不同的长度和内部表达形式。

在使用返回非整型量的函数以前,必须使程序的其余部分知道其类型。这个原因是很容易弄懂的。在 C++ 中,除非另加说明,否则函数将返回整型量。如果你的程序调用了—一个函数,而该函数又返回了一个与其说明类型不符的值,编译程序将会错误地产生该函数调用的代码。通知编译程序有关函数返回值的类型的最好的办法是用一个函数原型。下面讨论这种方法。

#### 9.3.1 使用函数原型

函数原型是一个最重要的新概念。在这一章中你将学到。函数原型担负着两个特殊任务:首先,它确定函数返回的类型使编译程序能产生函数返回数据类型的正确代码。其次,它确定了函数使用的变量的类型和数量。它一般采用的形式为:

## 类型 函数名(参量表)

原型通常在程序的顶部，或者在头文件中，而且在该函数调用以前就已被包含在其中。

函数原型不是最初的 K&R C 中的部分，而是由 ANSI 标准化委员会增补的。他们使 C 能够提供更强的类型检查，有点类似于 Turbo Pascal 等语言所提供的检查，并且将函数的返回类型告诉编译程序。当原型被使用时，它们允许编译程序对下列情况给出错误信息：一是当被调用函数的参数类型定义和类型转换非法时；二是当一个函数的参数个数不符时。关于原型的其他方面将在这一章中看到。在这一节里，你将看到如何使用函数原型来返回一个非整型数据类型。

作为一个函数返回非整型类型的介绍性示例，下面这个程序用了一个叫 `sum()` 的函数，此函数返回其两个 `double` 参数之和，也是 `double` 值：

```
#include <stdio.h>

double sum(double a, double b); /* prototype the function */

main()
{
    double first, second;

    first = 1023.23;
    second = 990.9;
    printf("%f", sum(first, second));
}

double sum(double a, double b) /* return a float */
{
    return a+b;
}
```

原型告诉编译程序 `sum()` 将返回一个双精度浮点数据类型。这要求编译程序正确地产生调用 `sum()` 的代码。如果将原型从这个程序中去掉，它将产生错误的结果。

下面是一个更实际的例子程序，用于计算一个给定半径的圆的面积：

```
#include <stdlib.h>

float area(float radius); /* prototype */

main()
{
    float r;

    printf("Enter radius: ");
    scanf("%f", &r);
    printf("Area is: %f\n", area(r));
}

float area(float radius)
{
    return 3.1416 * radius * 2;
}
```

这里，`area()` 函数返回浮点数值，所以必须告诉编译程序这个函数调用的前题。

关键是要懂得返回非整型的数据必须让编译程序先知道返回类型。最好是用函数原型

来实现之。现在你已学了原型，这本书将在所有示例中用到它们并且用户代码一般包含原型，即使函数只返回一个整型量。

### 9.3.2 返回指针

虽然返回指针的函数其处理方式与其它任何类型的函数都一样，但还是有几个重要概念要讨论。

正如你所知，变量的指针既非整型又非无符号整型。它们是一个特定类型数据的内存地址。这一区别的原因在于指针的算术运算与它们指向的基本类型有关。也就是说，如果一个整型指针增加一，则该指针的值就会比原来增加 2。更一般地说，指针每增加一次，都指向其类型的下一个数据。由于不同的数据类型可能具有不同的长度，所以编译程序必须知道指针所指数据的类型，以保证指针能确实指到下一个数据。因此，一个返回指针的函数必须加以说明。

例如，下面的函数在找到一个相同的字符时，将返回一个指向该字符串中这一位置的指针：

```
/* Return a pointer to the first character
   in s that matches c. */

char *match(char c, char *s)
{
    int count;

    count = 0;

    /* look for match or null terminator */
    while(c!=s[count] && s[count]!='\0') count++;

    /* if match, return pointer to location;
       otherwise return a null pointer
    */
    if(s[count]) return(&s[count]);
    else return (char *) '\0';
}
```

函数 `match()` 返回一个指向字符串中第一个与 `C` 相配的字符的指针。如果没有找到，就返回一个指向空结束符(`null`)的指针。这儿是一个使用 `match()` 函数的小程序：

```
#include <stdio.h>
#include <conio.h>

char *match(char c, char *s);

main()
{
    char s[80], *p, ch;

    printf("enter a string and a character: ");
    gets(s);
    ch = getch();
    p = match(ch, s);
    if(p) /* there is a match */
        printf("%s ", p);
    else
        printf("no match found");
}
```

该程序读入一个字符串和一个字符。如果该字符在字符串里面，则从该字符开始打印字符串。否则打印“no match found。”例如，你输入“hi there”作为字符串和t作为字符，程序将回答“there”。

### 9.3.3 void 类型函数

当一个函数不返回任何值时，可以用void来描述。这样做是为了防止它在任何表达式中的使用和帮助消除滥用。例如，函数print\_vertical()打印其字符串参数，竖直排列在屏幕上。因为它不返回任何值，则被说明为void。

```
void print_vertical(char *str)
{
    while(*str)
        printf("%c\n", *str++);
}
```

必须在任何程序用到的print\_vertical()中包含一个原型。如果没这么做，Turbo C++将指定它返回一个整型量。而后，当编译程序实际用于此函数时，它将断定是类型未标明的错误。下面的程序是一正确的例子：

```
#include <stdio.h>

void print_vertical(char *str);

main()
{
    print_vertical("hello");
}

void print_vertical(char *str)
{
    while(*str)
        printf("%c\n", *str++);
}
```

过去，不返回值的函数简单地被缺省为类型int。然而，你可以将所有不返回值的函数都说明为void，基于此，本书中所有需要的地方都用void。

## 9.4 更多的有关原型的知识

在这一节里，我们将研究函数原型的其他一些方面。

### 9.4.1 参数不匹配

除了告诉编译程序函数的返回类型，原型还防止函数被错误的参数调用。虽然C能自动将一个变量类型转化为参数接收类型，但是一些类型转换是非法的。如果一个函数有原型，任何非法状态转换将被发现而且错误信息出现。例如，下面程序因为企图调用sqr\_int()用一个整型参数而不是要求的整型指针而产生错误信息(将整型变为指针是非法的)。

```
/* This program uses a function prototype to
   enforce strong type checking.
*/
```

```

int sqr_it(int *i); /* prototype */

main()
{
    int x;

    x = 10;
    sqr_it(x); /* type mismatch */
}

sqr_it(int *i)
{
    *i = *i * *i;
}

```

另一个例子程序不能编译是因为 `sqr_it()` 被错误数目的参数调用。

```

/* This program shows how a function must
   be called with the proper number of arguments.
*/

int sqr_it(int *i); /* prototype */

main()
{
    int x;

    x = 10;
    sqr_it(&x, 10); /* wrong number of arguments */
}

sqr_it(int *i)
{
    *i = *i * *i;
}

```

除了告诉编译程序关于函数的返回数据类型外，用函数原型还可以帮助你防止函数被非法参数调用并在发生前捕捉到故障。此外，原型还可以用于证明你的程序是否正确工作，它不允许函数被错误数量的参数调用。

#### 9.4.2 头文件：更详尽的说明

早在此书中，我们就谈到过标准头文件。然而，你仅仅知道它们包含着一些库函数需要的信息。但我们不能以偏概全。Turbo C++ 的头文件包含两个主要方面。函数的定义和标准函数的原型与头文件的联系。例如，之所以 `STDIO.H` 被包含在本书几乎所有的程序中是因为它包含函数 `printf()` 的原型。通过为你程序中的每个库函数包含适当的头文件，编译程序可以得知在使用过程中可能会出的任何偶然性错误。更进一步来说，养成包含所有适当头文件的习惯是有好处的，因为在 C++ 中要求这么做。

你一定意识到了在程序中，一个函数原型不被任何一个函数包含时，可以让 Turbo C++ 发出警告。在集成环境中要做到这样，须用到 Options 主菜单。如果用命令行编译程序，用 `_wpro` 选项。

### 9.4.3 无任何参数的原型函数

正如你知道的那样,一个函数原型通知编译程序有关函数返回值的类型以及函数使用的参数的类型和数目。然而,虽然C语言最初版本中没有原型部分,但当你需要一个不带任何参数的函数原型时产生了一个特殊情况。这是因为ANSI C标准规定当一个函数原型不包含任何参数时,也就没有信息用于定义函数的参数类型和数目时,为了保证旧式C语言程序能被现代编译程序例如Turbo C++编译,上述是很必要的。不过应怎样通知Turbo C++——一个函数不带任何参数呢?答案是用关键字void放在参数表中。例如,请看下面这个程序:

```
#include <stdio.h>

void myname(void);

main()
{
    myname();
}

void myname(void)
{
    printf("Herb");
}
```

在这个程序中,myname()的原型明确通知编译程序它没有参数。由于函数参数表必须与其原型保持一致,void也必须包含在myname()的说明部分。基于此原型,Turbo C++就不会对象这样的对myname()的调用进行编译:

```
myname("Here ");
```

不过,如果在参数表说明中丢掉了void,就不会有任何错误信息。

在这一方面总观全书,每当函数不带参数时将被说明为void参数表。尽管没有必要,但由于一致性要求,main()函数当其无参数时也被说明为main(void)。

### 9.4.4 有关旧式C程序

原型是被ANSI C标准加入C语言的。在此之前,完整地定义函数原型是不可能的。只有函数的返回值可被说明,但它的参数却不能说明。因而编译程序能够为返回值产生正确的代码。然而由于函数的参数没有任何说明,函数可以被那些类型、数量不符的参数调用而编译程序毫无办法。因为ANSI C标准需要与K&R标准向下兼容,它规定这种部分原型仍是合法的。这样,旧程序可以用Turbo C++编译。不过,对于新程序,你通常用完全原型。这里讨论的旧体不能用在C++程序中。

一个部分原型语句一般具有这种形式:

类型定义 函数名()

即使函数带有参数,在类型说明中也不列出。采用这种部分的方法,早些时间出现的那个sum()程序将像下面这样:

```
#include <stdio.h>

double sum(); /* partial prototype */

main(void)
{
```



```

double first, second;

first = 1023.23;
second = 990.9;
printf("%f", sum(first, second));
}

double sum(double a, double b) /* return a float */
{
    return a+b;
}

```

懂得部分原型法对你来说是很重要的,原因是还有大量的书籍和论文仍包含旧式 C 程序代码。如果发现在这些程序中只有部分原型,不用介意; Turbo C 将针对它们进行正确的编译。如果想在 C++ 程序中用一些旧代码,则需要将任何部分原型转换为完全原型。

## 9.5 作用域规则

一种语言的作用域规则,决定了一段程序是否被另一段所“知道”,或能否被另一段程序访问。这个问题在第四章已被粗略提及;现在我们更详细地讨论一下。

C 语言中的每个函数都是独立的代码块。函数代码对于该函数来说是私有的,除了对函数的调用以外,其它任何函数中的任何语句都不能访问它。(例如,用 goto 语句转跳到其它函数中间是不允许的。)组成函数体的程序代码是与程序的其余部分独立的,除非使用全局变量或数据,否则不会影响程序的其它部分,也不会受到其它部分的影响。换句话说,在一个函数内定义的程序代码和数据,不会与另一个函数内的程序代码和函数互相影响,因为这两个函数的作用域不同。

有三种类型变量:局部变量、形式参数和全局变量。作用域决定了它们如何作用于程序的其它部分和建立其生存期。下面是对作用域的详细讨论。

### 9.5.1 局部变量

在函数内部说明的变量称为局部变量。C 语言支持更加广泛的局部变量,变量可以放在任何代码块中被说明。实际上,函数的局部变量简单说来是一特殊情况。局部变量可以仅被块中的语句说明。换句话说,在其本身代码块以外并不知道该变量的存在。请你记住,代码块以左花括号开始,到右花括号结束。从这里可知道代码块实际是指复合语句。

掌握局部变量的最重要的一个问题是,仅当说明它们的语名被执行时,局部变量才存在。也就是说,当程序运行到这个复合语句时,它们才产生,退出时它们则消失。

可以定义局部变量的最常用的代码块是函数。例如,考虑下面这两个函数:

```

void func1(void)
{
    int x;

    x = 10;
}

void func2(void)
{
    int x;

    x = -199;
}

```

整型变量 `x` 被说明两次，一次是在函数 `func1()` 中，另一次在函数 `func2()` 中。`func1()` 中的 `x` 与 `func2()` 中的 `x` 没有任何因果联系，原因是每个 `x` 只在定义它的函数内有效。

为了证明这一点，试运行下列程序：

```
#include <stdio.h>

void f(void);

main(void)
{
    int x;

    x = 10;

    printf("x in main is %d\n", x);
    f();
    printf("x in main is still %d\n", x);
}

void f(void)
{
    int x;

    x = 100;

    printf("the x in f() is %d\n", x);
}
```

C 语言中有一个关键字 `auto`，可以用来说明局部变量。然而，由于所有的非全局变量在 `auto` 省略时都被认为是局部变量，所以 `auto` 实际上从来不用。因此，你在本书中看不到任何使用它的例子。

在函数说明中，习惯是首先说明函数所需的全部变量。这样做主要是为了使读者对所使用的变量一目了然。但是并不一定这么做，因为局部变量可以在任一复合语句的任何地方说明。为了了解其工作过程，请看下面的函数：

```
void f(void)
{
    char ch;

    printf("continue (y/n)? :");
    ch = getche();

    /* enter this block only if answer is yes */
    if(ch == 'y') {
        char s[80]; /* this is created only upon
                     entry into this block */
        printf("enter name:");
        gets(s);
        process_it(s); /* do something */
    }
}
```

这里，当程序运行到 `if` 复合语句时，局部变量 `s` 才生成，而当退出该复合语句时，`s` 则消失。此外，`s` 只在 `if` 复合语句中有效，而在别处无效，即使是在该函数内的其它部分也是如此。

由于局部变量是随一个函数或复合语句建立和消失的,所以当函数或复合语句运行后它们所定义的局部变量内容也就丢失了。这一点在函数调用时尤其值得重视。当函数被调用时,其中的局部变量就建立,而当函数返回时,这些变量就不复存在了。这意味着在函数调用中局部变量无法返回(然而,这有一个例外,稍后有更详尽的说明)

除非特别加以说明,局部变量是存储在堆栈中的。而堆栈是内存中的一个动态变化的区域,这就说明了为什么局部变量通常不能在函数被调用后保存它的值。

### 9.5.2 形式参数

正如你知道的,如果一个函数需要使用参数,那么就必须说明变量来接受这些参数。除了接受函数的输入参数,它们在函数内部可以像其它局部变量那样来使用。这些变量称为函数的形式参数。

记住:必须保证说明的形式参数和调用函数时使用的参数具有相同的类型。而且,尽管形参承担了把外部参数值传给函数的特殊任务,却可以像其他局部变量那样来使用。

### 9.5.3 全局变量

和局部变量不同的是,全局变量在整个程序内都是“可见的”,可以被任一段程序使用。事实上,它们的作用域是对整个程序的。而且它们在整个程序的运行中都保存其值。说明全局变量是在所有函数之外进行的,它们可以被任何一个表达式使用,不管这个表达式在哪个函数内。

在下面的程序中,你可看到变量 `count` 的说明语句在全部函数之外,也在 `main()` 函数之前。然而,全局变量可以在任何地方说明,只要求说明在使用之前,并且在函数以外即可。习惯上最好是在程序的一开始就说明。

```
#include <stdio.h>
int count; /* count is global */

void func1(void), func2(void);

main(void)
{
    count = 100;
    func1();
}

void func1(void)
{
    func2();
    printf("count is %d", count); /* will print 100 */
}

void func2(void)
{
    int count;

    for(count=1; count<10; count++)
        printf(".");
}
```

仔细研究一下上面这段程序就可以发现,虽然 `main()` 和 `func1()` 都未曾说明 `count`,但它们都可以使用该变量。而 `func2()` 中说明了一个叫做 `count` 的局部变量,当在 `func2()` 中使

用变量名称 `count` 时,所指的是局部变量 `count` 而不是全局变量 `count`。请记住,如果局部变量和全局变量同名,则在说明该局部变量的函数中,该变量名仅代表局部变量,而与同名的全局变量无关。这很有利,不过若忘了这一点,就难以解释程序中有时出现的怪现象,虽然程序“看上去”是正确的。

全局变量在内存中有固定的区域,由编译程序 Turbo C++ 为其专门开辟。当程序中有多个函数需要共享某些数据时,使用全局变量是很有好处的。然而,你应该避免使用不必要的全局变量,原因有三个:

- 全局变量在程序的整个运行过程中都占据着内存,而不是仅仅在使用它们时才占用。
- 在只使用局部变量就可以的地方使用全局变量,使得函数的适用性变差。因为该函数将依赖于在其外部定义的某些变量。
- 大量使用全局变量的程序,容易被程序中一些未知的,不需要的副作用影响而导致出错。

最后一点在 BASIC 中得到了证实,其中所有的变量都是全局变量。在编制大型程序过程中所遇到的一个主要问题是,由于某变量在别处被使用而使其值偶然改变。在编制 C 语言程序中如果使用大量的全局变量则会遇到同样的问题。

结构化语言的一个基本原则是实现代码和数据的分隔化。在 C 语言中,这种分隔化是通过使用局部变量和函数来实现的。例如,以下是计算两个整数乘积的简单函数 `MUL()`,可以有以下两种写法:

<i>General</i>	<i>Specific</i>
<pre>mul(int x, int y) {     return(x*y); }</pre>	<pre>int x,y; mul() {     return(x*y); }</pre>

这两个函数都返回变量 `x` 和 `y` 的乘积。然而,按一般形式编写,即使用形式参数的函数可以用于返回任何两个整数的乘积,而按另一种特别的形式编写的函数只能用于计算全局变量 `x` 和 `y` 的乘积。

#### 9.5.4 作用域最后的例子

在一个短程序中的各种作用域在图 9-1 中用图形描述出来。在图中,在内层中的代码可具有外层的知识,反之则不行。外层中的代码不会受到内层的作用和影响。

如果仔细研究一下下页的图,就会理解作用域的意义。你也许很想试一下,看看变量的改变是如何影响程序的。

### 9.6 函数的参数和自变量:更详尽说明

这一章将详细研究 C 语言如何处理函数的参数和自变量。

#### 9.6.1 赋值调用和赋地址调用

通常,可以用两种方法之一来给予程序传递参数。一种方法称为赋值调用。这种方法

把实参的值复制给子程序的形参，而子程序中形参的改变对调用它的实参没有影响。

```
/* SCOPE: A program with various scopes */
#include <stdio.h>
#include <string.h>

int count; /* global to entire program */

void play(char *p);

main(void)
{
    char str[80]; /* local to main() */

    printf("enter a string: ");
    gets(str);
    play(str);
}

void play(char *p) /* p is local to play */
{
    if(!strcmp(p, "add")) {
        int a, b; /* local to if block inside play */
        printf("enter two integers: ");
        scanf("%d%d", &a, &b);
        printf("%d\n", a+b);
    }

    /* int a, b not known here */
    else if (!strcmp(p, "beep")) printf("%c", 7);
}
```

图 9-1 SCOPE 程序的使用域

赋地址调用是第二种方法。这种方法是把实参的地址复制给形参。在子程序中，该地址用来访问在调用程序中的实参。这就意味着对形参的改变会影响调用程序中的实参。

C 语言通常是用赋值调用的方法传递参数。这意味着通常你不能改变调用该函数的变量。(你会在本章的后面发现如何“强制”使用指针传址来改变调用变量。)考虑下面的函数：

```
#include <stdio.h>

sqr(int x);

main(void)
{
    int t=10;

    printf("%d %d", sqr(t), t);
}
```

```

sqr(int x)
{
    x = x*x;
    return(x);
}

```

在这个例子中，sqr()的参数值 10 被复制到形参 x 中。当执行 `x=x*x` 语句时，只是改变了局部变量 x，用来调用 sqr() 的变量 t 仍然是 10，因此输出结果是 100 和 10。

记住：只是实参值的复制品被传递到那个函数，对于该函数内部所发生的事情，并不影响到函数调用时所用的实参。

### 9.6.2 建立一个赋地址调用

尽管常规上 C 语言的参数传递是赋值调用，但可以用传递一个指针给形参的方式来模拟赋地址调用过程。由于这是将实参的地址传递给该函数，所以可以在该函数外面改变形参的值。

指针如同其它值一样传递给函数。当然，必须把参数说明为指针类型。例如，考虑函数 swap()，交换它的两个整型参数的值，如下所示：

```

void swap(int *x, int *y)
{
    int temp;

    temp = *x; /* save the value at address x */
    *x = *y;   /* put y into x */
    *y = temp; /* put x into y */
}

```

运算符\*用于访问由它的操作数所指的变量。因此，用来调用该函数的变量的内容已经被改变了。

一定要记住，像 swap() 这种函数(或者使用指针参数的任何其它函数)被调用时，必须用实参的地址。下面的程序说明了调用 swap() 的正确方法：

```

#include <stdio.h>

void swap(int *x, int *y);

main(void)
{
    int x, y;

    x = 10;
    y = 20;

    printf("initial values of x and y: %d %d \n", x, y);
    swap(&x, &y);
    printf("swapped values of x and y: %d %d \n", x, y);
}

```

在这个例子中，变量 x 被赋值 10，y 被赋值 20。然后以 x 和 y 的地址作为参数调用 swap()。单目运算符&用来产生变量的地址。因而，传递给 swap() 的是 x 和 y 的地址，而不是它们的值。

这时候你应该懂得了为什么必须用 scanf 接受数值时在参数之前加上&。实际上，

你已经将它们的地址传递出去以便调用值可以被修改。

正如将在第三部分见到的，C++具有能够产生更简单可靠的赋地址调用的特征。

### 9.6.3 函数调用与数组

当数组作为函数的实参时，只传递数组的地址，而不是将整个数组复制到函数中去。当用数组名作为实参调用函数时，指向该数组的第一个元素的指针就被传递到函数中(请记住，在C语言中，没有任何下标的数组名，是一个指向数组第一个元素的指针)。这意味着参数说明必须具有相同的指针类型。有三种方式可以说明接收数组指针的形参。第一种是作为一个数组说明，如下所示：

```
#include <stdio.h>

void display(int num[10]);

main(void) /* print some numbers */
{
    int t[10], i;

    for(i=0; i<10; ++i) t[i]=i;
    display(t);
}

void display(int num[10])
{
    int i;

    for(i=0; i<10; i++) printf("%d ", num[i]);
}
```

即使参数 `num` 是被说明为有 10 个元素的整型数组，C 编译程序也会自动地将它转换成一个整型指针。这是必要的，因为实际上没有哪个形参能接收整个数组。只是将数组指针传递给函数，所以必须要有指针形参来接收它。

第二种说明数组形参的方法是把它说明为可变长度的数组。如下所示：

```
void display(int num[])
{
    int i;

    for(i=0; i<10; i++) printf("%d ", num[i]);
}
```

这里，`num` 被说明为可变长度的整型数组。由于 C 编译程序不提供边界检查，所以数组的实际长度与该形参无关。(当然在程序中就不是这么回事了)这种说明方法实际上也把 `num` 定义为整型指针。

最后一种方法是将 `num` 说明为一个指针。这也是 C 语言程序最普遍的专业书写形式。如下所示：

```
void display(int *num)
{
    int i;

    for(i=0; i<10; i++) printf("%d ", num[i]);
}
```

之所以允许这么做，是因为任何指针都能用[]来作索引，就像数组一样。

所有这三种说明数组形参的方法得出同样的结果：指针。

另外当数组的元素作为实参时，可以像任何其它简单变量一样对待。如下所示：

```
#include <stdio.h>
void display(int num);

main(void) /* print some numbers */
{
    int r[10], i;

    for(i=0; i<10; ++i) r[i]=i;
    for(i=0; i<10; i++) display(r[i]);
}
void display(int num)
{
    printf("%d ", num);
}
```

可以看到，display()中的参数是一个整型数。display()用数组元素调用是不恰当的，因为一次只用到了数组的一个值。

了解在什么情况下数组作为函数的实参是十分重要的，因为这时传递给函数的是它的地址。这不同于C语言赋值调用的参数传递的习惯用法。它意味着函数中的程序对数组进行操作，并且可能改变调用该函数的数组的内容。例如，考虑函数print\_upper()，它用大写字母打印字符串参数：

```
/* Print a string in uppercase. */
#include <stdio.h>
#include <ctype.h>

void print_upper(char *string);

main(void)
{
    char s[80];

    printf("enter a string: ");
    gets(s);
    print_upper(s);
    printf("\noriginal string is altered: %s", s);
}

void print_upper(char *string)
{
    register int t;

    for(t=0; string[t]; ++t) {
        string[t] = toupper(string[t]);
        printf("%c", string[t]);
    }
}
```

调用print\_upper()函数后，在main()中数组s的内容就变成了大写字母。如果不想这样做，就应该将程序改写为：

```
/* Print string as uppercase. */
#include <stdio.h>
#include <ctype.h>
```



```

void print_upper(char *string);

main(void)
{
    char s[80];

    printf("enter a string: ");
    gets(s);
    print_upper(s);
    printf("\noriginal string is unchanged: %s", s);
}

void print_upper(char *string)
{
    register int i;

    for(i=0; string[i]; ++i)
        printf("%c", toupper(string[i]));
}

```

在这段程序中，数组 *s* 的内容保留了原样，没有改变，因为它的值没被更改。

将一个数组传递给函数的典型例子是标准库函数 *strcat*。虽然在标准库函数中有所不同，但下面的一段程序将告诉你它是如何工作的。为了避免与标准库函数混淆，这里的函数称为 *hsstrcat*。(*strcat*) 将两个字符串联接起来，并且返回第一个字符串的指针，尽管这个指针很少用到。)

```

/* Demonstrate hsstrcat() */

#include <stdio.h>

char *hsstrcat(char *s1, char *s2);

main(void)
{
    char s1[80], s2[80];

    printf("Enter two strings: ");
    gets(s1);
    gets(s2);

    hsstrcat(s1, s2);

    printf("concatenated: %s", s1);
}

char *hsstrcat(char *s1, char *s2)
{
    char *temp;

    temp = s1;
    /* first, find the end of s1 */
    while(*s1) s1++;

    /* add s2 */
    while(*s2) {
        *s1 = *s2;
        s1++;
        s2++;
    }
}

```

```

    *a1 = '\0'; /* add the null terminator */
    return temp;

```

**bsstrcat()**函数必须用两个字符数组调用,用字符指针定义。进入 **bsstrcat()**后,函数找到第一个字符串的末尾,然后把第二个字符串加在其后,最后返回第一个字符串指针。

## 9.7 argc, argv 和 env——main 中的参数

在运行程序的时候,有时需要将信息传递给它。传递信息到 **main()**中的最常用的方法是用命令行实参。命令行实参是在操作系统下键入命令行时附于程序名后的信息。例如,当用命令行方式编译程序时,需要键入如下一些内容:

```
>tcc filename
```

其中 **filename** 是要编译的程序,它作为实参传递给 Turbo C++。

**main()**有三个特殊的内部形参。前两个, **argc** 和 **argv** 是用来接收命令行实参的。第三个为 **env**, 当程序开始执行时用于存取 DOS 环境参数。这些都是 **main()**所独有的参数。让我们来仔细研究一下。

**argc** 参数保存命令行的参数个数,是个整型量。它至少是 1, 因为至少程序名就是第一个实参。**argv** 参数是指向字符指针数组的指针。这个数组里的每个元素都指向命令行实参。所有命令行实参就是字符串——任何数字都必须由程序转变成为适当的格式。下面给出的简单程序就说明了命令行实参的用法,它在屏幕上显示“hello”,如果在程序名后直接键入了名字的话,它接下去就显示:

```

#include <stdio.h>
#include <process.h>

main(int argc, char *argv[]) /* name program */
{
    if(argc!=2) {
        printf("You forgot to type your name\n");
        exit(0);
    }
    printf("Hello %s", argv[1]);
}

```

如果命名该程序为 **NAME** 并且你的名字是 Tom, 运行这个程序时应键入 **name tom**。则程序输出为“Hello Tom”。例如,若是在 DOS 下 A 驱动器中运行程序后,将会看到:

```
A>NAME Tom
```

```
hello Tom
```

```
A>
```

命令行实参必须由空格(space)或制表符(tab)分隔。逗号,分号等都不能被认为是分隔符。例如:

```
one, two, and three
```

由四个字符串组成,而

```
one,two,and three
```

是两个字符串——逗号是非法的分隔符。

如果需要传递一个包含空格的命令行参数,必须将其置于引号中。例如下个字符串可

被当作一个单独的命令行参数:

```
"this is one argument"
```

正确地说明 `argv` 是十分重要的。最通常的方法是:

```
char *argv[];
```

空方括号表示数组是可变长度的。这样可以由 `argv` 引导访问各个实参。例如, `argv[0]` 指向第一个字符串, 这总是程序的名字; `argv[1]` 指向第一个实参以此类推。(在 DOS 3.0 以前的版本中 `argv[0]` 为空。)

下面是一个有趣实用的运用命令行参数的程序。在命令行输入的一系列 DOS 命令被其执行, 并且用到 `system()` 库函数。这个函数传递任何用于操作系统调用的字符串。如果此字符串包含一有效的操作系统命令, 则命令被执行, 然后程序恢复。

```
/* COMLINE: a program that executes whatever DOS
   commands are specified on the command line.
*/
#include <stdio.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
    int i;

    for(i=1; i<argc; i++)
        system(argv[i]);
}
```

假定此程序名叫 `COMLINE`, 下面的命令行将会执行 DOS 命令 `VER`、`CHKDSK` 和 `DIR *.C`。

```
C>COMLINE VER CHKDSK "DIR *.C"
```

你将会惊奇地发现这个程序有如此多的用途。

若要访问字符串命令中的单个字符, 就要为 `argv` 增加第二个下标。例如, 下面的程序是在屏幕上显示所有的实参, 一次显示一个字符, 如下所示:

```
/* The program prints all command line arguments it is
   called with. */
#include <stdio.h>

main(int argc, char *argv[])
{
    int t, i;

    for(t=0; t<argc; ++t) {
        i = 0;
        while(argv[t][i]) {
            printf("%c", argv[t][i]);
            ++i;
        }
        printf(" ");
    }
}
```

记住, 第一个下标用于访问字符串, 第二个下标用于访问字符串的字符。

通常可以用 `argc` 和 `argv` 将初始命令传入程序。在 Turbo C++ 中, 可以设置操作系统允许下尽可能多的命令行参数。在 DOS 中限定一行最多为 128 个字符。通常可以使用这

些参数来表示文件名或选择项。使用命令行参数一方面可以使你的程序更显得专业化, 另一方面也便于程序用在批处理文件中。

参数 `env` 和 `argv` 参数的说明相同。它是一个包含环境设置的的字符数组的指针。数组中的最后一个字符串为空, 作为结束标记。下面这个程序打印出所有当前环境设置字符串:

```
#include <stdio.h>

main(int argc, char *argv[], char *env[])
{
    int i;

    for(i=0; env[i]; i++) printf("%s\n", env[i])
}
```

注意在说明 `env` 参数时, 即使没用到 `argc` 和 `argv`, 也必须说明它们, 因为参数说明是有状态依赖性的。如果 `env` 参数未用到则不需说明, 这是有效的。

最后一点: 在本章中早些时候提到的, 当 `main()` 无参数时, 通常将 `main()` 的参数表说明为 `void`。

### 9.8 从 `main()` 中返回值

尽管迄今为止你还未见到这样的程序: 从 `main()` 中返回一个整型数, 但是这是可能的, 该值返回到调用过程, 通常是操作系统。你可以像在函数体内一样用 `return` 语句从 `main()` 中返回值。对 DOS 和 OS/2 来说, 返回值为 0 代表程序执行成功。其它数值表明程序由于某些错误而中止运行。

为了解这是如何工作的, 可以改进前节的 `COMLINE` 程序使其具有当命令出错时能向操作系统返回一个出错码的功能, 可以利用 Turbo C++ 中的 `system()` 函数的特点来达到目的。这个函数当成功时返回 0, 否则为 1。

```
/* COMLINE: a program that executes whatever DOS
   commands are specified on the command line.

   Return error code to the operating system if
   an operation fails.
*/
#include <stdio.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
    int i;

    for(i=1; i<argc; i++) {
        if(system(argv[i])) {
            printf("%s failed\n", argv[i]);
            return -1; /* failure code */
        }
    }
    return 0; /* return success code */
}
```

一些编程人员喜欢把不返回值的 `main()` 确定为 `void`, 用下面的语句:

```
void main(void)
```

然而，这是不必要的，本书也不打算采用这种形式。另一种方法是总让 `main()` 返回值。现在你知道从 `main()` 中返回值，并且本书中的全部子程序都这么做。

注意：尽管从 `main()` 中返回值和函数原型都已涉及到了，你还需重温第四章的警告信息。这个信息将被函数随时用于显示，如果函数不返回值又没有用 `void` 说明的话。

## 9.9 递归

在 C 语言和 C++ 中，函数可以自己调用自己。所谓函数递归就是在函数体中有调用自己的语句。递归有时被称为循环定义，即用其自身来定义自己的过程。

递归的例子很常见。用递归方法定义一个无符号整数，就如同用数字 0, 1, 2, 3, 4, 5, 6, 7, 8 和 9 加或减另一个整数。例如，数字 15 是 7 加 8; 21 是 9 加 12; 12 是 9 加 3 等等。

对于计算机语言来说，若要实现递归函数必须能够调用自己。典型的递归例子是函数 `factr()`，它计算整数的阶乘。数  $N$  的阶乘是 1 到  $N$  之间所有整数的乘积。例如，3 的阶乘是  $1*2*3$ ，得 6。 `factr()` 和它的循环等价程序如下所示：

```
factr(int n) /* recursive */
{
    int answer;
    if(n==1) return(1);
    answer = factr(n-1)*n;
    return(answer);
}

fact(int n) /* non-recursive */
{
    int t, answer;

    answer = 1;
    for(t=1; t<=n; t++) answer = answer*(t);
    return(answer);
}
```

无递归操作的程序 `factr()` 可以说是一目了然。它使用从 1 开始到指定数字的循环，逐次对每个数进行累乘。

递归操作函数 `factr()` 并不很复杂。当用实参 1 调用 `factr()` 时，函数返回 1; 否则返回值为 `factr(n-1)*n`。当计算该表达式的值时，又以  $n-1$  作为参数调用 `factr()`，直到  $n$  等于 1 并返回到调用它的函数。

现计算 2 的阶乘，第一次调用 `factr()` 导致了第二次调用，此时参数为 1，这次调用返回值为 1，然后乘以 2 ( $n$  的初值)，最后得 2。如果将 `printf()` 语句插入到 `factr()` 中，会显示出最终结果和每一级调用的中间结果。

当函数调用自己时，新的局部变量和参数就会在堆栈中分配存储单元，同时函数代码以新变量重新开始执行。递归调用不是重新复制该函数，只是参数是新的。每次递归调用返回时，过去的局部变量和参数会从堆栈中弹出来并且恢复到函数内上次调用自身的地方执行。函数递归可说成是“望远镜”的伸出和收回。

大多数递归过程并不能明显地减小代码大小或变量存储大小，它们也许还要比其它

循环等价程序执行得稍慢些，这是因为增加了调用次数。但在大多数情况下，这种变化并不明显。许多递归调用函数会造成栈超限，但也未必都如此。因为函数参数和局部变量是存储在堆栈中的，每新调用一次该函数，就会产生这新变量的新拷贝，堆栈空间就减少。你也许不必为此担忧，除非递归函数出现运行异常。

递归函数的主要优点是能够实现一些比同类型的循环更简单明了的算法。例如，用循环算法实现“快速分类”是相当困难的，有些问题也似乎要借助于递归的方法来解决，尤其是像一些与人工智能有关的问题。此外，有些人似乎按递归的方式思考问题比按迭代方式更容易。

当编写递归函数时必须有个 `if` 语句迫使函数返回而不再作递归调用。假如没有这样做，当调用该函数时，它就会无休止地执行下去而不会返回，这是写递归函数时经常发生的错误。在编制程序中可在一些地方先使用 `printf()` 和 `getchar()` 函数，这样在运行中就可以随时看到执行情况，若发现了问题，可以及时终止程序运行。

这儿是递归函数的另一例子，函数名字叫 `siren()`，用到了 Turbo C++ 中的 `sound()` 和 `nosound()` 函数发出警报声。函数 `sound()` 用一整型参数调用并变成计算机扬声器中发出的连续的声音，持续到执行 `nosound()` 函数。此函数无参数。当频率小于 100 `siren()` 函数调用自己时，此时递归调用开始并发出警报声。

```
/* This program sounds a siren. */
#include <stdio.h>
#include <dos.h>
int siren(int freq);

main(void)
{
    siren(1000);
    return 0;
}

siren(int freq)
{
    int i;

    if(freq>100)
        siren(freq-100);

    /* turn on sound */
    sound(freq);

    /* delay just a bit */
    for(i=0; i<10000; i++) ;
    nosound(); /* turn off sound */
}
```

## 9.10 参数说明的传统风格和现代风格

当发明 C 语言时，用到了一种不同的函数参数说明。这种老办法有时叫作传统形式。在本书中用到的最为广泛的参数说明方法是现代形式。Turbo C++ 支持这两种形式。(然而，ANSI C 标准重视现代形式。)知道传统形式对你来说是很重要的，因为仍有几百万行 C 程序用的是传统形式。而且书上和杂志上印的程序也采用这种形式，因为它是为编程服务的——特别是老的人员。让我们来看一下传统风格和现代风格的区别。

传统的函数变量说明中包括两部分：一个参数表，紧跟在函数名之后放在括号里；实际的参数说明，在闭括号和函数的花括号之间。传统风格的参数说明的一般形式如下：

```
类型    函数名(参数 1, 参数 2, .....参数 N)
类型    参数 1
类型    参数 2
```

.

.

.

```
类型    参数 N
```

```
{函数体}
```

例如，现代风格的说明为：

```
float f(int a, int b, char ch)
{
    .
    .
    .
}
```

传统风格时将为：

```
float f(a, b, ch)
int a, b;
char ch;
{
    .
    .
    .
}
```

注意在传统风格中，类型名后可以列有多于一个的参数。

记住现代风格稍优于传统风格有一些十分微妙的原因，因而本书用的是现代风格。但是，如果你看到一个用传统风格写的程序，记住 Turbo C++ 能够很容易地对其编译——尽管传统函数说明的使用将会导致显示一些警告信息，但這些是可以忽略的。

## 9.11 补充问题

在建立函数时，应该记住几个重要事情，它们影响函数的效率和实用性。本节将讨论这些问题。

### 9.11.1 参数和通用函数

通用函数是适用于任何场合，或许还为许多不同的编程者所共同使用的函数。一个典型的特征是，不应将通用函数建立在全局变量基础上。函数需要的所有信息都应由它的参数来传递。

除了使你的函数通用以外，参数还保证了程序代码的可读性，并且部分地避免了由于副作用而引起的错误。

### 9.11.2 效率

函数是构造 C 语言的模块。除了个别小程序以外它几乎是编写所有程序的关键。本节所述都证明了这一点。然而在某些特定应用中你也许需要放弃使用函数,而用内部代码代替它。内部代码与函数的作用相同,但不调用函数。仅仅是在运行速度很关键的情况下,内部代码才用于代替函数调用。

内部代码比函数调用要快,有两点原因,首先,一个调用结构要花费一定的执行时间,其次,所传递的参数要放在堆栈中也耗费时间。对于几乎所有的应用程序来说,这点执行时间的增加是无足轻重的。但是,如果是这样的话,要注意若用了内部代码来代替程序中的函数调用,则每次调用函数的时间就会省下来。例如,下面的两条程序都打印 1 到 10 的平方值。内部代码形式的程序运行速度比函数调用快因为后者要花费时间。

#### *In Line*

```
#include <stdio.h>

main(void)
{
    int x;

    for(x=1; x<11; ++x)
        printf("%d", x*x);
    return 0;
}
```

#### *Function Call*

```
#include <stdio.h>
int sqr(int a);

main(void)
{
    int x;

    for(x=1; x<11; ++x)
        printf("%d", sqr(x));
    return 0;
}

sqr(int a)
{
    return a*a;
}
```

编程序时,在注重提高程序的可读性、可修改性和可移植性的同时出一定要权衡函数执行时间的开销。

现在你已看到了 Turbo C++ 函数的功能,是该进入 I/O 系统的时候了。



## 第十章 输入、输出和磁盘文件

C 语言中的输入和输出贯穿了库函数使用的全过程, C 语言没有关键字代表 I/O 操作。ANSI 标准在 Turbo C++ 之后定义了一个 I/O 函数的完备集。不过, 老的 UNIX 标准包含了处理 I/O 操作的两个不同系统。第一种方法叫做 ANSI 文件系统, 是由 ANSI 标准和 UNIX C 共同制定的。(有时也叫做格式化或缓冲型系统)。第二种是 UNIX 型文件系统(有时也叫做非格化或非缓冲型系统), 是由 UNIX 单独定义的。

ANSI 标准不定义 UNIX 型文件系统。这是有许多理由的, 况且定义两组文件系统太多余了。然而, 考虑到目前这两种标准都被广泛地应用, Turbo C++ 支持这两种标准。因此这一章包含二者, 但是本章侧重于 ANSI 标准的 I/O 系统。这是因为 UNIX 型文件系统用得越来越少了, 因此建议新程序最好按照 ANSI 的 I/O 函数来编写。

本章将讲述 C 的 I/O 操作概况, 并将说明上述两组文件系统的主要函数是如何工作的。记住 C 的函数库中包含了非常丰富的各种各样的 I/O 函数——这里只讲一部分。你可以从用户手册上学到其余部分。

注意: 尽管 C++ 支持本章中讨论的 I/O 函数, 它还提供了自己的面向对象的方法来实现 I/O 操作。因而有三个理由使你需要学习 C 语言的 I/O。首先, C 语言的 I/O 系统中隐含的许多概念 C++ 也用到。第二, 有几千(或许几百万) C 语言程序存在, 这些不会在一夜之间全变为 C++ 而是需要很多年才能向 C++ 演化, C++ 的特点也在一点点增加。因此, 你完全有必要接触本章中的 I/O 函数, 而且懂得它们的用途是很重要的。(这将特别有助于将 C 语言程序转化为 C++ 程序。)第三, C 语言的 I/O 系统是非常丰富的。你将会发现一些比 C 型函数来得简单得多的操作。

在开始学习 C 语言的 I/O 系统之前需要了解两个特殊的编译命令和一些术语。

### 10.1 两个预处理指令

在 C 语言的源程序中可能包含着各种各样的 C 编译指令。我们称这些为预处理指令, 虽然它们并不真属于 C 语言的内容, 但可用来扩展 C 语言程序的编程环境。所有的预处理指令都以符号 # 开头。绝大多数的预处理指令以后再涉及, 可现在用 C 的文件系统时需要其中的两条指令。

#### 10.1.1 #define 指令

#define 指令用来定义一个标识符和一个字符串, 在程序中每次遇到该标识符时就用所定义的字符串替换它。这个标识符叫做宏替换名, 替换过程叫做宏替换。宏定义指令 #define 的一般形式是

#define 宏替换名 字符串

注意这里没有分号。字符串是指任何字符序列, 字符不需要引号。在标识符和字符串之间可以有任意个空格, 但是字符串结束后一定要换行。例如你想用 TRUE 表示数值 1, 用 FALSE 表示数值 0, 可以用下面 2 个 #define 宏定义说明:

```
#define TRUE 1
```

```
#define FALSE 0
```

这样在编译时每当源程序中遇到 TRUE 或 FALSE 时, 编译程序就自动用 1 或 0 替代。例如, 下述语句在屏幕上显示 “0 1 2: ”:

```
printf("%d %d %d :", FALSE, TRUE, TRUE+1)
```

若定义了一个宏替换名, 这个名字还可以做为其它宏定义的一个部分来使用。例如, 下述语句定义了 ONE、TWO 和 THREE 所代表的值:

```
#define ONE 1
```

```
#define TWO ONE+ONE
```

```
#define THREE ONE+TWO
```

要充分理解宏替换仅仅是简单地用所说明的字符串来替换对应的标识符。如果你想定义一个标准出错信息可以像下面这样书写:

```
#define E_MS "standard error on input\n"
```

```
·  
·  
·
```

```
printf(E_MS);
```

编译程序只是在遇到了标识符 E\_MS 时才用 “standard error on input\n” 来替换。对编译程序来说, printf() 语句的实际形式是

```
printf("standard error on input\n");
```

否则, 若宏替换名出现在某一字符串中则不进行替换。例如,

```
#define XYZ this is a test
```

```
·  
·  
·
```

```
printf('XYZ');
```

语句不会显示 “this is a test”, 而是显示 “XYZ”。

#define 的常见用法是定义一些东西的大小, 例如一个数组的大小, 它们会随着程序的变化而变化。在下面的简单例子里, 宏 MAX\_SIZE 用于一整型数组的大小, 并且控制 for 循环的循环条件。

```
#include <stdio.h>
```

```
#define MAX_SIZE 16
```

```
unsigned int pwr_of_two[MAX_SIZE];
```

```
/* Display powers of 2. */
```

```
main(void)
```

```
{
```

```
    int i;
```

```
    pwr_of_two[0] = 1; /* start the sequence */
```

```
    for(i=1; i<MAX_SIZE; i++)
```

```
        pwr_of_two[i] = pwr_of_two[i-1] * 2;
```

```
    printf("The first 16 powers of 2: \n");
```

```
    for(i=0; i<MAX_SIZE; i++)
```

```

        printf("%u ", pwr_of_two[i]);
    }
    return 0;
}

```

在本章中介绍 `#define` 的原因是因为 I/O 系统用到的头文件中包含你将会用到的各种各样的常数。

### 10.1.2 `#include` 指令

到现在为止，你已经感性地使用过 `#include` 指令，现在我们来仔细研究这条重要指令。

`#include` 预处理指令直接指示编译程序将该指令所指的另一个源文件嵌入到 `#include` 指令所在的程序。文件名应使用双引号或尖括号起来。如下面两种形式都会指示编译程序读取和编译叫 TEST 的文件。

```
#include "TEST"
```

```
#include <TEST>
```

被包含文件带有 `#include` 指令是合法的。这叫做嵌套包含。

如果文件的明确路径是作为文件名标识符的一部分给出的，那么编译时将只在指定的目录中查找包含文件。否则如果文件名包括在双引号内，将首先查找当前工作目录。如果文件没找到则在命令行所指定的目录中继续搜索，用 `-I` 选择项。最后若仍未找到这个文件，就在由环境工具所指定的标准目录中查找。

如果包含文件在尖括号内且没给出明确的路径名，首先在编译命令 `-I` 选择项所指定的目录中查找。如果文件未找到，则搜索标准目录，而不会在当前工作目录中寻找这个文件。

如果在 Borland 的指导下安装好了 Turbo C++，那么应选用尖括号——本书将采用这种方法。

## 10.2 流与文件

在讨论 C 的 I/O 系统之前，有必要先搞清楚“流”和“文件”这两个术语的区别。C 语言 I/O 系统为 C 语言编程人员提供了一个统一的接口，与具体的访问设备无关。也就是说，C 语言 I/O 系统在编程者和被使用的设备之间提供了一套抽象的东西，这个抽象的东西叫做“流”。具体的实际设备叫“文件”。必须充分重视流与文件两者之间的内在联系。

### 10.3 流(streams)

ANSI C 文件系统设计上可以支持各种不同设备，包括终端、磁盘驱动器和磁带机等。虽然各种设备差别很大，但是 ANSI C 文件系统把每个设备都转换为一个逻辑设备，叫做流。所有的流都有相同的行为。因为流在很大程度上与设备无关，这样，一个用来进行磁盘文件写入操作的函数也可以用来进行控制台写入。有两种类型的流：文本流(text stream)和二进制流(binary stream)。

### 10.3.1 文本流

文本流是一连串的字符。在文本流中,某些字符的变换由环境工具的需要来决定。例如,一个换行符可以变换为回车换行,这是 Turbo C++ 的工作方式。因此,所读写的字符与外部设备之间没有一一对应的关系,而且所读写的字符个数和外部设备中的也可以不同。

### 10.3.2 二进制流

一个二进制流是由与外部设备中的内容一一对应的一连串字节组成的。使用中没有字符翻译过程。而且所读写的字节数目也与外设中的数目相同。不过,一个二进制流可以加一些空字节使之占满磁盘的一扇区。

### 10.3.3 文件

在 C 语言中,“文件”是一个逻辑概念,可以用来表示从磁盘文件到终端所有东西。用一个打开操作使流和一个特定文件发生联系。一旦一个文件被打开,用户程序就可以与该文件之间交换信息。

并不是所有的文件都有相同的功能。例如,一个磁盘文件可以允许随意存取,但一个终端就不行。这说明 C 语言 I/O 系统的一个重要观点:所有的流都是相同的,而文件就不一定。

如果一个文件支持随机存取(有时称为“位置请求”),打开该文件时先把文件位置指示器设置到它的开头处。每当从该文件中读取或写入一个字符后,该位置指示器就增加,以保证整个文件的读写顺序。

关闭操作使文件脱离一个特定的流。对于用来输出的已打开的流,关闭这个流时则将与这个流有关的缓冲区的内容写入外部设备。这个过程一般叫做“刷新”这个流,以保证没有残存信息偶尔留在磁盘缓冲区内。当用户程序正常结束时,所有的文件都自动关闭。

每一个与文件相结合的流有一个 FILE 型的文件结构。这个结构在头文件 `stdio.h` 中定义。你不能对这个文件控制块做改动。(你将在下一章学习结构,但就其要点,结构是一种归在同一名字下各种变量的组合。这与 Turbo Pascal 中的 RECORD 相似。不管怎样,在使用 I/O 程序时不需要懂得任何有关结构的知识。)

## 10.4 概念和实际

让我们来进一步讨论,以下概括 C 语言 I/O 系统的操作方法。正如编程人员所关心的,所有的 I/O 通过流来进行。所有的流都一样,因为都是一系列字符。文件系统把流与文件连接起来。由于各个设备具有不同的功能,所有文件并不都一样,但是这种差别对于编程人员来说微乎其微。C 语言的 I/O 系统把来自设备的原信息转换到流之中,或者反过来把流中的信息转换给各设备。除了需要了解哪类文件可以随机存取这一点之外,而只针对被称为“流”的逻辑设备去考虑编程就行了。

如果对这种方法感到奇怪或含糊不清,则需参考一下 BASIC 或 FORTRAN 语言的内容,其中系统所支持的各设备有各自的完整的独立系统。在 C 语言中,编程者只需要考虑流这个概念并且只使用一个文件系统来完成所有的 I/O 操作。

## 10.5 控制台 I/O

控制台 I/O 指的是发生在计算机键盘和显示器上的操作。通常控制台 I/O 由 ANSI 文件系统的—个专用子系统来完成(将 Turbo C++ 的一些特殊函数加在一起支持计算机和用户之间更好的交互作用)。因为控制台输入和输出用得很多,所以需要由 ANSI 文件系统的—个子系统来专门处理控制台 I/O。从技术上来说这些函数是直接用于标准输入和标准输出的。在包括 DOS 在内的很多操作系统里,控制台 I/O 可以重新定向到别的设备。然而,为了简便起见,控制台就是最常用的设备。稍后你将学习到这些函数如何与其它文件系统函数连接起来。

### 10.5.1 字符读写

最简单的控制台 I/O 函数是用于从键盘读入一个字符的 `getche()` 和把一个字符显示到屏幕上的 `putchar()`。函数 `getche()` 等待从键盘上键入一个字符,返回它的值并且在屏幕上自动回显该字符。函数 `putchar()` 把它的字符参数显示在光标的当前位置上。下面是 `getche()` 和 `putchar()` 的原型:

```
int getche(void);
int putchar(int c);
```

不要被 `getche()` 返回一个整数而迷惑;其低位字节包含着字符。即使 `putchar()` 用一个整型参数说明,实际上输出到屏幕上也只有低位字节。`getche()` 包含在头文件 `CONIO.H` 中而 `putchar()` 包含在头文件 `STDIO.H` 中。

下面的程序从键盘读入一些字符并把它们经大小写变换后显示出来。也即,大写字符改为小写,小写字符改为大写,打入句号后程序停止运行。头文件 `CTYPE.H` 是由于 `islower()` 库函数需要而设置的,此函数当其参数为小写时返回真,否则为假。

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>

main(void) /* case switcher */
{
    char ch;

    printf("enter chars, enter a period to stop\n");
    do {
        ch = getche();
        if(islower(ch)) putchar(toupper(ch));
        else putchar(tolower(ch));
    } while (ch!='.'); /* use a period to stop*/
    return 0;
}
```

`getche()` 有两个重要的变体。第一个是 `getchar()`, 它是 UNIX 字符输入函数的原型。这个函数的麻烦在于它的输入缓冲区直到键入一个回车符才返回给系统。这是由于最初的 UNIX 系统的行缓冲区就是这样设计的,必须回车才能把你键入的字符送往程序。这样就可能引起 `getchar()` 返回之后仍留下一个或多个字符在输入排队流中。这与现在使用的内部环境很不协调,所以建议不用这个函数。为了保证 Turbo C++ 对 UNIX 系统的程序移植性而且由于它是由 ANSI C 标准定义的才使其得到保留。你只需要对此函数有一些了解就可以了。函数 `getchar()` 包含在头文件 `STDIO.H` 中。

`getche()`的第二个变体 `getch()` 非常有用，它的作用与 `getche()` 基本一致，只是不把读入的字符回显到屏幕上。它包含在头文件 `CONIO.H` 中。注意：`getche()` 和 `getch()` 都不是 ANSI 标准定义的。然而，由于 `getchar()` 的缺点二者才被 Turbo C++ 采用。(其它多数基于 PC 的 C 语言和 C++ 也一样。)

### 10.5.2 字符串读写

进一步来说，就复杂性和功能强弱来就要算函数 `gets()` 和 `puts()`。它们用来在控制台读写字符串。

`gets()` 函数用来从键盘读入一字符串，并把它们送到 `gets()` 函数中字符型指针变量所指定的地址。你可以在键盘上键入多个字符并以回车符结束。回车符不是这字符串的一部分，而是由一个空终结符在串的最后来代替它。事实上，用 `gets()` 来返回一个回车符是不可能的(虽然 `getchar()` 可以做到)。字符打错时可以用光标左移键在回车之前更改。`gets()` 函数的原型是：

```
char *gets(char *s);
```

其中 `s` 是一个字符数组，用来接收用户输入的字符。它的原型在 `STDIO.H` 中，下面的程序读入一个字符串到 `str` 数组中并显示它的长度：

```
#include <stdio.h>
#include <string.h>

main(void)
{
    char str[80];

    gets(str)
    printf("length is %d", strlen(str));
    return 0;
}
```

`gets()` 函数返回一个指针给 `s`。

函数 `puts()` 把它的字符串参数写到屏幕上并换行。它的原型为：

```
int puts(char *s);
```

`puts()` 函数中可以使用与 `printf()` 一样的转义控制符。如 `\t` 表示水平制表符 TAB。使用 `puts()` 函数比使用 `printf()` 函数带来的冗余操作少得多。它仅用来输出字符串，不能输出数组和进行格式变换。因而，`puts()` 比 `printf()` 所占内存小，执行速度也快。所以在需要很高优化的代码中经常使用 `puts()`。函数 `puts()` 返回一个指向这个字符串的指针，如果成功则是非负值，否则为 EOF。下面这个语句将 `hello` 写在屏幕上。

```
puts("hello");
```

函数 `puts()` 使用 `STDIO.H` 头文件。

用于进行控制台 I/O 操作的最简单的函数在表 10-1 中列出。

## 10.8 控制台格式化 I/O

除了上述简单的控制台 I/O 函数外，C 标准库还包含两个用来对内部类型数据进行格式化输入输出的函数：`printf()` 和 `scanf()`。格式化是指在你的控制下，这些函数可以用各种不同的格式要求来读写数据。函数 `printf()` 用来向控制台写数据；相反，`scanf()` 用来从键盘

表 10-1 基本控制台 I/O 函数

函数	操作
<code>getchar()</code>	从键盘读入一个字符，回车返回
<code>getche()</code>	从键盘读入一个字符并回显，不用回车
<code>getch()</code>	从键盘读入一个字符不回显，不用回车
<code>putchar()</code>	向屏幕写一个字符
<code>gets()</code>	从键盘读入一个字符串
<code>puts()</code>	向屏幕写一个字符串

读数据。`printf()`和`scanf()`这两个函数都可以对任何一种类型的内部数据，包括字符、字符串和数字进行操作。虽然在本书一开头就用过这两个函数，这里还将详细讲解它们的用法。

### 10.6.1 `printf()`函数

`printf()`函数的原型是：

```
int printf(char *控制字符串, ...);
```

`printf()`的原型在 `STDIO.H` 中。注意在 `printf()`原型的末尾的三个点，当函数能够接受数目可变的参数时，它的原型就用三个句点表示。

控制字符串由两种不同类型的内容组成。第一类是那些显示在屏幕上的字符组成的。第二类包含格式化命令来定义参数的显示格式。一个格式化命令的开头带有一个百分号(`%`)，后面跟一个格式码。格式化命令见表 10-2。参数的个数与格式化命令所说明的个数一样多并且在顺序上要一一对应。例如调用 `printf()`：

```
printf("Hi %c %d %s", 'c', 10, "there!");
```

就显示：Hi c 10 there!

一个格式说明中还可以带有修饰项用确定显示宽度、小数位数及左端对齐等。在百分号和类型字符(格式码)之间可以写入一个整数作为最小宽度说明项。这个插入项使输出带有若干空格或者 0 以保证所说明的最小宽度。如果字符串的长度或数值大小超过了说明宽度，将按其实际长度显示。如果想在显示值前加一些 0，就在宽度项前加个 0。例如 `%05d` 将在显示一个小于 5 位的数值时在它前面补 0，使其宽度保持 5 位。缺少时补入空格。

表 10-2 `printf()`格式化命令

说明符	格式
<code>%c</code>	单个字符
<code>%d</code>	十进制数

%d	十进制数
%e, %E	科学计数
%f	浮点十进制数
%g, %G	使用%e或%f中表达较短者
%o	八进制数
%s	字符串
%u	无符号十进制数
%x	十六进制数
%%	显示百分号%
%p	显示一个指针
%n	变量应是一整型指针，其中存放已写字符的个数

为了确定浮点数小数点的位置，将一个小数点放在宽度修饰符之后，具体位置由体决定。例如，%10.4f 显示数值时宽度至少 10 位，带有 4 位小数。当这种格式出现在字符或整型量的输出格式化命令中，小数后的数字代表最大宽度。例如%5.7s 将显示一个不小于 5 个而且不超过 7 个的字符串。若大于 7 个字宽，第 7 个以后的字符被删除。

缺少说明时，所有的输出均为右对齐格式。如果显示宽度大于被显示数据位数时，数据尾部都以显示区的右端对齐。你可以在%后面加一个负号来说明要求显示的方式为左端对齐。例如%-10.2f 就用左对齐方式把一个二位小数的浮点数显示在 10 个字符宽的区域內。

有两个格式化命令修饰符来使 printf() 显示 short 和 long 整型数。它们可以用在 d, l, o, u 和 x 类型说明中。修饰符 l 告诉 printf() 函数其参数是长整型数。例如%ld 是说明要显示长整型。修饰符 h 指示 printf() 显示短整型量。因此%hu 说明数据类型是 short unsigned int。

修饰符 l 也可以用在浮点数说明 %e, %f 和 %g 中，说明要显示的是 double 型。修饰符 L 则是指 long double。

可以按需要利用 printf() 输出各种形式的数据，图 10-1 给出了一些简单的例子。

printf() Statement	Output
("%-5.2f", 123.234)	: 123.23
("%.5f", 3.234)	: 3.23400
("%10s", "hello")	:       hello
("%-10s", "hello")	: hello
("%.5s", "123456789")	: 12345

图 10-1 printf() 的一些简单例子

### 10.5.2 scanf() 函数

控制台或输入函数是 scanf()。它能够读取各种内部类型的数据并自动把它转化为预先指定的格式。它可以看作是 printf() 的反函数。scanf() 的一般形式为：

```
int scanf(char* 控制字符串, ...)
```



scanf()的原型在 STDIO.H 中。控制字符串包括三类不同的字符内容:

- 格式说明(format specifiers)
- 空白字符(white-space characters)
- 非空白字符(non-white-space characters)

输入格式说明前也有一个百分号用来告诉函数 scanf()下一个将读入什么类型的数据。这些格式说明码已被列在表 10-3 中。例如%s 表示读一个字符串而%d 表示读一个整型数。

表 10-3 scanf()格式说明码

说明码	含义
%c	读入一个字符
%d	读入一个十进制整数
%i	读入一个十进制整数
%E	读入一个浮点数
%f	读入一个浮点数
%h	读入一个短整型数
%o	读入一个八进制数
%s	读入一个字符串
%x	读入一个十六进制数
%p	读入一个指针
%n	接受一个整型数, 其值为已读入的字符个数

在格式控制字符中的一个空白字符会使 scanf()函数在读操作中跳过输入流中的一个或多个空白字符。空白字符可以是空格、制表符/t 或换行符\n。实质上在控制字符串中有空白字符使 scanf()函数在读操作中读入所有的空白字符但是不存储它们。遇到非空白字符(包括 0 在内)才存储起来。

一个非空白字符会使 scanf()函数在读入时剔除与这个非空白字符相符的字符。例如"%d,%d"使 scanf 先去读一个整型数, 然后把接着读的逗号剔除, 最后再读入另外一个整型数。如果这一特定字符未找到, scanf()函数就终止。如果这一特定字符未找到, scanf()函数就终止。

scanf()函数中使用的所有用来接收数值的变量必须用它们的地址表示。也就是说参数表中列出的必须是指向这些变量的指针, 这就是 C 语言的参量调用法, 且允许函数改它的参数内容。例如读入一个整数并放在变量 count 中, 可以用下面形式的 scanf():

```
scanf("%d", &count);
```

字符串被读入到一个字符数组中时, 不带下标的数组名就是它的第一个元素的地址。因此要读入一个字符串并放到一个字符数组 address 中时, 可以这样写:

```
scanf("%s", address);
```

在这种情况下 address 已经是一个指针, 不需要在它的前面加&操作符。

输入数据项必须用空格、制表符或回车符分开。标点符号, 如逗、引号都不能作为分

隔符使用。这就是说,

```
scanf("%d%d", &r, &c);
```

可以接受 10 20 这种形式的输入, 而 10,20 不行。与 printf() 一样, scanf() 中的各格式说明码在个数和顺序上与参数表中的变量必须一一对应。

在百分号和格式码之间若写入一个星号\*时, 函数仍将去读入数据但不赋值。例如, 在输入 10/20 时, 则把 10 赋给 X, 忽略除法号/ 并把 20 赋给 Y。

```
scanf("%d%c%d", &x, &y);
```

格式化命令中还可以用修饰项说明最大读入长度。在 % 和格式码之间写入一个整型数, 它表示任何读操作中最大字符个数。例如希望读入 str 的字符不超过 20 个, 可以这样写:

```
scanf("%20s", str);
```

如果输入流大于 20 个字符, str 满 20 个字符就不再继续读入了。而后边一个读入函数就会从第 21 个开始读。例如键入:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

作为参数调用上句 scanf(), 因为限制了最大长度, 只有 A~T20 个字符被读入 str。也就是说, 剩下的 U~Z 这 6 个字符没有用到。如果再另外调用一次 scanf(), 例如:

```
scanf("%s", str);
```

则 U~Z 6 个字符被放入 str。输入字符的个数小于说明的最大宽度是可以的, 一旦读入的字符串中有空格, scanf() 就认为这个字段已被读完, 并去读第二个字段。

虽然空格、制表符和换行符都可以作为字段分隔符来使用, 但在读入单个字符的时候, 它们也被像其他单个字符一样被读入。例如输入流为 x y 时,

```
scanf("%c%c%c"&a, &b, &c );
```

则返回 x 值给 a, 空格给 b, y 给 c。

警告: 1. 如果在控制字符串中含有任何其它字符如空格、制表符、换行符等, 就会使读入逐一对照, 遇到与这些相同的字符就会将其从中剔除。例如, 给一个 10 20 的输入流,

```
scanf("%st%s", &x, &y);
```

将把 10 给 X, 20 给 Y。因为控制字符中有 t, 所有 t 被剔除了。又例如,

```
scanf("%s ", name);
```

直到在空白字符后键入一字符时函数才返回。因为 %s 后的一个空格指示 scanf() 函数要求输入并剔除一个空格、制表符或换行符。

2. 不要用 scanf() 去显示一个提示信息。因为所有的提示必须在调用 scanf() 之前完成。函数 scanf() 还包含功能强大的特点称为搜索集。搜索集中定义了 scanf() 中特殊的字符表。scanf() 将继续读字符只要它们在搜索集中。一旦输入了不包含在搜索集中的字符, scanf() 则自动转向下一个格式说明。搜索集是由你想搜索的字符的列表来定义的, 这些字符在方括号中。左方括号前一定要有一百分号 %, 例如, 下面这个搜索集告诉 scanf() 只读数字 0~9。

```
%[123456789]
```

搜索集的对应参数必须是指向字符数组的指针。从 scanf() 返回后, 数组将包含一个由读入字符组成的非空字符串。想知道是怎样工作的, 请看此程序:

```
#include <stdio.h>
```

```

main(void)
{
    char s1[80], s2[80];

    scanf("%1234567890] %s", s1, s2);
    printf("\n%s||%s", s1, s2);
    return 0;
}

```

用下面的输入运行程序

**"123456789abcdefg987654"**

随后键入回车，程序将显示

**123456789 || abcdefg987654**

因为 **a** 不是搜索集中的字符，`scanf()` 则停止将字符读入 `s1` 并且将剩余字符送入 `s2`。可以用连字号来定义搜索集的范围。例如，想告诉 `scanf()` 接收从 **A~Z** 的字符，可以用以下控制码：

**%[A-Z]**

可以在搜索集中定义不只一个范围。例如，这个程序读入数字和字母，说明可以用搜索集定义的最大域。

```

#include <stdio.h>

main(void)
{
    char str[80];

    printf("Enter digits and letters: ");
    scanf("%78[a-zA-Z0-9]", str);
    printf("\n%s", str);
    return 0;
}

```

你也可以定义一个相反集，如果集中第一个字符是一个 **^**。当 **^** 出现时，它指示 `scanf()` 接收任何没有在搜索集中定义的字符。最重要的一点是要记住搜索集的要求是严格明确的，因此，如果想搜索大写和小写字母，必须对它们分别定义。

## 10.7 缓冲型 I/O 系统(ANSI 型 I/O 系统)

缓冲型 I/O 系统由若干个有内在联系的函数构成。最常用的函数已在表 10-4 中列出。使用这些函数时需要把头文件 **STDIO.H** 包含到调用这些函数的程序中。

表 10-4 最常用缓冲型文件系统函数

函数名	操作
<code>fopen()</code>	打开一个流
<code>fclose()</code>	关闭一个流
<code>puts()</code>	向流里写一个字符

<code>gets()</code>	从流里读一个字符
<code>fseek()</code>	从流里寻找一个指定字符
<code>fprintf()</code>	向流里写像在屏幕上写一样
<code>fscanf()</code>	从流里读像从键盘上读一样
<code>feof()</code>	遇到文件结束指示时返回真
<code>ferror()</code>	遇到错误发生时返回真
<code>rewind()</code>	重新把文件指针设置到文件开头
<code>remove()</code>	删除一个文件

---

### 10.7.1 文件指针

文件指针是贯穿缓冲型 I/O 系统的主线。一个文件指针是一个指向文件有关信息的指针，这些信息定义了许多东西，包括文件名、状态和当前位置。在概念上文件指针标志一个指定的磁盘文件。与文件指针相组合的流用来告诉系统的每个缓冲型 I/O 函数应该到什么地方去完成操作。文件指针是一个 `FILE` 型指针变量，在 `STDIO.H` 中定义。为了读写文件，用户程序需要文件指针。为了获得各种各样的指针，使用下面的语句：

```
FILE *fp;
```

### 10.7.2 打开文件

`fopen()` 函数有两个目的，其一是打开一个流并把一个文件与这个流连接，其二是返回分配给文件的文件指针。函数 `open()` 的原型是这样的：

```
FILE *fopen(char *filename, char *mode);
```

这里 `mode` 是说明文件打开方式的字符串。`filename` 必须是由一个字符串组成的在操作系统下有效的文件名并允许带有路径名。

合法有效的模式值见表 10-5。“r”代表正文，“b”代表二进制，如果不定义，那么文件将按照 Turbo C++ 的全局变量 `_fmode` 打开。这个变量既可被设置为 `O_TEXT` 代表正文模式也可以设置成 `O_BINARY` 代表二进制模式。缺省时为 `O_TEXT`。宏 `O_BINARY` 和 `O_TEXT` 可以在 `FCNTL.H` 中找到。本书将设置 `_fmode` 为缺省值。

表 10-5 `fopen()` 有效的模式值

模式	含义
"r"	打开一个文本文件用于读
"w"	生成一个文本文件用于写
"a"	对一个文本文件用于添加
"rb"	打开一个二进制文件用于读
"wb"	生成一个二进制文件用于写
"ab"	对一个二进制文件用于添加
"r+"	打开一个文本文件用于读/写

---

"w+"	生成一个文本文件用于读/写
"a+"	打开或生成一个文本文件用于读/写
"r+b"	打开一个二进制文件用于读/写
"w+b"	生成一个二进制文件用于读/写
"a+b"	打开或生成一个二进制文件用于读/写
"rt"	打开一个文本文件用于读
"wt"	生成一个文本文件用于写
"at"	对一个文本文件用于添加
"r+t"	打开一个文本文件用于读/写
"w+t"	生成一个文本文件用于读/写
"a+t"	打开或生成一个文本文件用于读写

---

`fopen()` 返回一文件指针。用户程序是永远不能改变这个指针值的。如果试图打开一个文件时有错误发生, 则 `fopen()` 返回一空值 `null`。

如表 10-5 所示, 一个文件可以用正文模式也可以用二进制模式打开。在正文模式中, 输入时, 回车换行被译为另起一行; 输出就反过来, 把另起一行译为回车换行。在二进制文件中则没有这样的翻译过程。

如果想打开一个名叫 `test` 的文件并准备写操作, 可以用语句

```
fp=fopen("test", "w");
```

不过常会看到这样的写法:

```
FILE *fp;

if ((fp = fopen("test", "w")) != NULL) {
    puts("cannot open file\n");
    exit(1);
}
```

宏 `NULL` 是在 `STDIO.H` 中定义的。这种写法可以在写文件之前先检验已打开的文件是否有错, 如写保护或磁盘已满等。定义使用 `null` 是因为文件指针不会永远是该值。顺便再介绍一个库函数 `exit()`, 调用 `exit()` 可引起程序立即终止, 不管 `exit()` 是由哪个函数调用的。它的原型在 `STDLIB.H` 中:

```
void exit(int val);
```

`val` 的值返回操作系统。前面的章节已经学过, 返回值为 0 代表成功结束操作, 其它值则表示由于一些问题使程序中止。

如果用 `fopen()` 打开一个文件准备写操作, 则原先在该文件名下的内容全部抹去, 并开始新内容的存放。如果原先无此文件, 则生成这个文件。如果想往文件的尾部再加写些内容, 就必须使用操作模式 `a`。在打开一个文件进行读操作时, 文件必须存在, 如果不存在这个文件, 则返回一个出错信息。打开一个文件读/写时, 如果文件存在, 它不会被抹掉; 如果文件不存在, 则生成这个文件。

### 10.7.3 写字符

函数 `putc()` 是用来向事先已用 `fopen()` 函数打开的一个写操作流中写字符。函数描述

为:

```
int putc(int ch, FILE *fp);
```

这里 `fp` 是由 `fopen()` 返回的文件指针, `ch` 表示输出的字符。文件指针 `fp` 告诉 `putc()` 函数写字符输到哪一个磁盘文件中去。由于历史原因, `ch` 名义上称为 `int`, 但它只使用低位字节。

如果 `putc()` 操作成功, 就返回那些所写人的字符; 如果操作失败返回 `EOF`。 `EOF` 是 `STDIO.H` 中定义的一个宏, 含义是“文件结束”。

#### 10.7.4 读字符

函数 `getc()` 用来从一个已由 `fopen()` 函数打开了的读操作流中读取字符。函数描述为:

```
int getc(FILE*fp);
```

这里 `fp` 是一个由 `fopen()` 返回的 `FILE` 型文件指针。由于历史的原因, `getc()` 返回一个整型量而且高位字节为 0。

当读到文件末时, `getc()` 返回一个 `EOF` 标记。可以用下面的程序一直读到文件末:

```
ch = getc(fp);  
while(ch!=EOF) {  
    ch = getc(fp);  
}
```

#### 10.7.5 feof()的使用

缓冲型文件系统也可对二进制数据操作。当一个文件为二进制输入打开时, 一个与 `EOF` 标志相等的整型量会被读入。这会造成程序判定文件已经结束, 而实际上文件物理结尾并未到达。为了解决这个问题, ANSI C 中包含了 `feof()` 函数, 用于在读二进制数据文件时测定文件的结束。函数 `feof()` 形式如下:

```
int feof(FILE*fp);
```

函数的原型在 `STDIO.H` 中。如果到达了文件末则返回值为真; 否则, 返回零值。因此, 下面的程序读二进制文件直到遇到文件结尾。

```
while(!feof(fp)) ch=getc(fp);
```

当然, 这样的方法同样也可适应于文本文件。

#### 10.7.6 关闭文件

`fclose()` 函数用来关闭一个由 `fopen()` 打开的流。这个函数把留在磁盘缓冲区里的内容都传给文件, 并且执行正规的操作, 关闭系统级的文件。流未关闭会引起很多问题, 例如数据丢失、文件损坏及其它一些可能的错误。`fclose()` 函数释放了与这个流有关的文件控制块, 以便再被使用。操作系统对同时打开的文件数目有一定的限制, 所以先关闭一个文件再打开另一个文件可能是必要的。

函数 `fclose()` 的调用方式为:

```
int fclose(FILE*fp);
```

其中 `fp` 是一个调用 `fopen()` 时返回的文件指针。文件关闭成功, 返回一个 0; 若返回其它值说明出错了。可以使用标准函数 `ferror()` (下面讨论) 来确定和显示错误类型。通常只是

在磁盘已被取出驱动器或磁盘已被写满时才会发生关闭错误。

### 10.7.7 ferror()和rewind()函数

ferror()函数用来确定文件操作中是否出错。如果一个文件以文本方式打开并且在读或写中发生错误,就返回 EOF。ferror()可用于确定究竟出了什么事。它的原型是:

```
int ferror(FILE *fp);
```

其中 fp 是一个有效的文件指针。在文件操作中发生错误时函数值返回“真”,否则返回“假”。由于每个文件操作都可能出错,所以应该在每次文件操作后立即调用 ferror(),否则有可能使错误被遗漏掉,ferror()函数原型在 STDIO.H 中。

函数 rewind()将文件的指针重新设置到该文件的起点。它的原型是:

```
void rewind(FILE*fp)
```

其中 fp 是一个有效的文件指针。rewind()的原型在 STDIO.H 中。

### 10.7.8 fopen(),getc(),putc()和fclose()函数的用法

函数 fopen(),getc(),putc()和fclose()构成文件操作程序的最小集合。下例程序 ktod 是 putc(),fopen()和fclose()的用法简例。它从键盘上读字符然后写到一个磁盘文件中,当读到符号\$时结束。文件名由命令行指定。例如在键盘上键入 KTOD TEST,可以向名叫 TEST 的文件输入文本。

```
/* ktod: A key to disk program. */

#include <stdio.h>
#include <stdlib.h>

main(int argc, char *argv())
{
    FILE *fp;
    char ch;

    if(argc!=2) {
        printf("You forgot to enter the filename\n");
        exit(1);
    }

    if((fp=fopen(argv[1], "w"))==NULL) {
        printf("cannot open file\n");
        exit(1);
    }

    do {
        ch = getchar();
        if(EOF==putc(ch, fp)) {
            printf("File Error");
            break;
        }
    } while (ch!='$');

    fclose(fp);
    return 0;
}
```

下面补充的程序 DTOS,可以读任何 ASCII 文件,并把内容显示在屏幕上。

```

/* dtos: A program that reads files and displays them
   on the screen.
*/

#include <stdio.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
    FILE *fp;
    char ch;

    if(argc!=2) {
        printf("You forgot to enter the filename\n");
        exit(1);
    }

    if((fp=fopen(argv[1], "r"))==NULL) {
        printf("cannot open file\n");
        exit(1);
    }

    ch = getc(fp); /* read one character */

    while (ch!=EOF) {
        putchar(ch); /* print on screen */
        ch = getc(fp);
    }

    if(ferror(fp)) printf("File Error");

    fclose(fp);
    return 0;
}

```

下面这段程序可以拷贝任何类型的文件。请注意，文件是以二进制模式打开的，使用 `feof()` 来检查文件是否结束。

```

/* Copy a file. */

#include <stdio.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
    FILE *in, *out;
    char ch;

    if(argc!=3) {
        printf("You forgot to enter a filename\n");
        exit(1);
    }

    if((in=fopen(argv[1], "rb"))==NULL) {
        printf("cannot open source file\n");
        exit(1);
    }

    if((out=fopen(argv[2], "wb")) == NULL) {
        printf("cannot open destination file\n");
        exit(1);
    }
}

```



```

/* this code actually copies the file */
while(!feof(in)) {
    ch = getc(in);
    if(ferror(in)) {
        printf("Error reading file");
        break;
    }
    putc(ch, out);
    if(ferror(out)) {
        printf("Error writing file");
        break;
    }
}

fclose(in);
fclose(out);
return 0;
}

```

### 10.7.9 getw()和 putw()函数的使用

除了 getc()和 putc()之外, Turbo C++ 提供了另外两个缓冲型 I/O 函数: putw()和 getw()。它们用于从磁盘文件中读或写整型量, 这些函数是由 ANSI C 标准定义的, 其用法与 getc()和 putc()完全相同, 所不同的只是读写整型量而不是字符。它们的原型如下:

```
int putw(int i, FILE *fp);
```

```
int getw(FILE *fp);
```

下面的语句用来向文件指针 fp 所指的磁盘文件写一个整型数:

```
putw(100, fp);
```

getw 和 putw()的原型在 STDIO.H 中。

### 10.7.10 fgets()和 fputs()函数

缓冲型 I/O 系统中还有两个函数: fgets()和 fputs(), 是用来从流中读写字符串用的。它们的原型如下:

```
char *fputs(char *str, FILE *fp);
```

```
char *fgets(char *str, int length, FILE *fp);
```

函数 fputs()与 puts()几乎完全一样, 只是它用来向指定的流中写字符串。函数 fgets()从指定的流中读取字符串, 直到读到换行符或第 length-1 字符为止。如果读入的是换行符, 它将作为字符串的一部分(这与 gets()不同)。但若 fgets()被中断, 则这个字符串是空。fgets()和 fputs()函数原型在 STDIO.H 中。

### 10.7.11 fread()和 fwrite()函数

fread()和 fwrite()是缓冲型 I/O 提供的两个用来读写数据块的函数。其原型如下:

```
unsigned fread(void *buffer, int numbytes, int count, FILE *fp);
```

```
unsigned fwrite(void *buffer, int numbytes, int count, FILE *fp);
```

对于 fread(), buffer 是一个指向用来接受从文件中读取的数据存储区的指针。对于 fwrite(), buffer 是一个指向将被写到文件中去的那些数据的指针。读写的字节数用

**num\_bytes** 表示。参数 **count** 指示共有多少个字段(每个字段长度为 **num\_bytes**)要被读或写。**fp** 是已打开的流的文件指针。这两个函数的原型在 **stdio.h** 中定义。

函数 **fread()** 返回读入的字节数, 如果到达了文件尾或发生了错误则可能小于 **count**。函数 **fwrite()** 返回写的字节数。这个数与 **count** 相等, 除非发生了错误。

只要文件以二进制文件方式打开, **fread()** 和 **fwrite()** 就可以读写任何 类型信息。下例中将写一个浮点数到磁盘文件中去。

```
/* Write a floating point number to a disk file. */
#include <stdio.h>
#include <stdlib.h>

main(void)
{
    FILE *fp;
    float f=12.23;
    if((fp=fopen("test","wb"))==NULL) {
        printf("cannot open file\n");
        exit(1);
    }

    if(fwrite(&f, sizeof(float), 1, fp)!=1)
        printf("File Error");

    fclose(fp);
    return 0;
}
```

正如程序所表明的那样, **buffer** 可以并且经常只是一个简单的变量。这个程序还介绍了另外一个 C 操作符: **sizeof**。 **sizeof** 操作符返回变量和数据类型字节数的大小。用 **sizeof** 确保程序工作正确。你将在此书中学到许多关于 **sizeof** 的知识。

**fread()** 和 **fwrite()** 的一个最有用的应用可以读写数组(或者, 将来可看到, 还有结构)。例如, 这段程序用一个单一的 **fwrite()** 语句将数组 **sample** 的内容写入文件 **sample** 中:

```
#include <stdio.h>
#include <stdlib.h>

main(void)
{
    FILE *fp;
    float sample[100];
    int i;

    if((fp=fopen("sample","wb"))==NULL) {
        printf("cannot open file\n");
        exit(1);
    }

    for(i=0; i<100; i++) sample[i] = (float) i;

    /* this saves the entire array in one step */
    if(fwrite(sample, sizeof(sample), 1, fp)!=1)
        printf("File Error");

    fclose(fp);
    return 0;
}
```

注意 `sizeof` 是如何用来说明 `sample` 大小的。

下一个程序用 `fread()` 读入先前程序写入的信息。并且显示这些数据在屏幕上。

```
#include <stdio.h>
#include <stdlib.h>

main(void)
{
    FILE *fp;
    float sample[100];
    int i;

    if((fp=fopen("sample", "rb"))==NULL) {
        printf("cannot open file\n");
        exit(1);
    }

    /* this reads the entire array in one step */
    if(fread(sample, sizeof(sample), 1, fp)!=1)
        printf("File Error");

    for(i=0; i<100; i++) printf("%f ", sample[i]);

    fclose(fp);
    return 0;
}
```

此书稍后你将看到其它一些例子完整地说明这些函数的用法。

#### 10.7.12 `fseek()` 函数和随机访问 I/O

可以借助于缓冲型 I/O 系统中的 `fseek()` 函数完成随机读写，它可以设置文件位置指示器。它的原型为：

```
int fseek(FILE *fp, long numbytes, int origin)
```

这里 `fp` 是调用 `fopen()` 时所返回的文件指针。`numbytes` 是个长整型量，表示由 `origin` 位置到当前位置的字节数。`origin` 是 `stdio.h` 中定义的几个宏之一：

origin	宏名	值
文件开头	<code>SEEK_SET</code>	0
当前位置	<code>SEEK_CUR</code>	1
文件尾部	<code>SEEK_END</code>	2

因此，为了从文件头开始搜索第 `numbytes` 个字节，`origin` 应该用 `seek_SET`。从当前位置起搜索用到 `seek_CUR`。从文件尾部开始搜索用 `seek_END`。

用下面的程序可以从一个叫 `test` 的文件中读取 234 个字节。

```
FILE *fp;
char ch;
```

```

if((fp=fopen("test", "rb"))==NULL) {
    printf("cannot open file\n");
    exit(1);
}

fseek(fp, 234, 0);
ch = getc(fp); /* read one character */
               /* at 235th position */
.
.
.

```

返回值为 0 时表明 `fseek()` 已正确执行，否则返回一个非零值。

下面的程序 `dump` 十分有用，它用 `fseek()` 函数使你可以对任意文件用 ASCII 和十六进制来查看其内容。通过一个可任何移动的“扇区”，可以看到 128 字节长的文件内容。当输入 D(即 `dump` 转储)命令时，它就以与 `debug` 相同的格式显示输出。如果想退出这个程序，就输入 -1。请注意，这里使用 `fseek()` 读文件，在读到文件的结束标记时，虽然所读入的字节数还不到 `size` 值，`fread()` 仍然将读入这个字节数并将读入的个数赋给 `display()` 函数(请记住，`fread()` 函数返回的是实际读了的个数)。请你输入这个程序至计算机中并了解它是如何工作的。

```

/* DUMP: A simple disk look utility using fseek */
#include <stdio.h>
#include <ctype.h> /* needed by isprint() library
                  function */
#include <stdlib.h> /* needed by exit() */

#define SIZE 128

char buf[SIZE];
void display(int numread);

main(int argc, char *argv[])
{
    FILE *fp;
    int sector, numread;

    /* if incorrect number of args, then error */
    if(argc!=2) {
        printf("usage: dump filename\n");
        exit(1);
    }

    if((fp=fopen(argv[1], "rb"))==NULL) {
        printf("cannot open file\n");
        exit(1);
    }

    for(;;) {
        printf("enter sector (-1 to quit): ");
        scanf("%ld", &sector);
        if(sector<0) break;
        if(fseek(fp, sector*SIZE, SEEK_SET)) {
            printf("seek error\n");
        }
        if((numread=fread(buf, 1, SIZE, fp)) != SIZE) {
            printf("EOF reached\n");
        }
    }
}

```

```

    display(numread);
}
return 0;
}

/* display the file */
void display(int numread)
{
    int i, j;

    for(i=0; i<=numread/16; i++) {
        for(j=0; j<16; j++) printf("23X", buf[i*16+j]);
        printf(" ");
        for(j=0; j<16; j++) {
            if(isprint(buf[i*16+j])) printf("%c", buf[i*16+j]);
            else printf(".");
        }
        printf("\n");
    }
}
}

```

请注意，函数 `isprint()` 用来确定哪些字符是可打印字符。当字符是可打印字符时，`isprint()` 返回真，否则返回假，该函数需要使用头文件 `CTYPE.H`，在程序开头处已把它连入程序。DUMP 程序把它自己的内容显示出来，如图 10-2 所示：

```

enter sector (-1 to quit): 0
D A 2F 2A 20 44 55 4D 50 3A 20 41 20 73 69 6D  ../+ DUMP: A sim
70 6C 65 20 64 69 73 68 20 6C 6F 6F 68 20 75 74  ple disk look ut
69 6C 69 74 79 20 75 73 69 6E 67 20 66 73 65 65  ility using fsee
68 20 2A 2F D A 23 69 6E 63 6C 75 64 65 20 3C k +/...#include <
73 74 64 69 6F 2E 68 3E D A 23 69 6E 63 6C 75  stdio.h>...#inclu
64 65 20 3C 63 74 79 70 65 2E 68 3E 20 20 20 2F de <ctype.h> /
2A 20 6E 65 65 64 65 64 20 62 79 20 69 73 70 72 * needed by ispr
69 6E 74 28 29 29 20 6C 69 62 72 61 72 79 D A int()) library..
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 .....
enter sector (-1 to quit): 1
9 9 9 66 75 6E 63 74 69 6F 6E 20 2A 2F D A  ...funcion +/..
23 69 6E 63 6C 75 64 65 20 3C 73 74 64 6C 69 62 #include <stdlib
2E 68 3E 20 20 2F 2A 20 6E 65 65 64 65 64 20 62 .h> /* needed b
79 20 65 78 69 74 28 29 20 2A 2F D A D A 23 y exit() */....#
64 65 66 69 6E 65 20 53 49 5A 45 20 33 32 38 D define SIZE 128.
A D A 63 68 61 72 20 62 75 66 5B 53 49 5A 45 ...char buf[SIZE
5D 38 D A 76 6F 69 64 20 64 69 73 70 6C 61 79 ];..void display
28 69 6E 74 20 6E 75 6D 72 65 61 64 29 38 D A (int numread);..
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 .....
enter sector (-1 to quit):

```

图 10-2 DUMP 程序输出实例

### 10.7.13 标准流

当一程序开始运行时，就自动打开三个流：标准输入(`stdin`)、标准输出(`stdout`)和标准错误(`stderr`)。一般情况下它们与控制台有关，但它们也可以由操作系统重定向到别的设备上。因为这是文件指针，缓冲型 I/O 系统可以使用它们完成控制台操作。例如 `putchar()` 函数可以定义为：

```

putchar(char c)
{
    putc(c, stdout);
}

```

可以用 `stdin`、`stdout` 和 `stderr` 作为文件指针，仅限于在用 `FILE` 型变量的函数中。

I/O 控制函数 `getchar()`、`putchar()`、`printf()` 和 `scanf()` 实际上用 `stdin` 和 `stdout` 实现它们的操作。由于 DOS 操作系统允许用 `>` 和 `<` 命令行操作符进行 I/O 重定向，这些函数还可以读写磁盘文件。例如，编译这个名叫 `IOTEST` 的程序：

```

#include <stdio.h>

main(void)
{
    printf("Hello there");
    return 0;
}

```

将看到，屏幕上不显示任何东西。然而，如果列出 `OUT` 的内容，将看到信息已被写入。

要记住 `stdin`、`stdout` 和 `stderr` 不是变量而是常量，而且不能转换。这些文件指针在程序开头自动地创立并在末尾自动关闭，不要试图去关闭它们。

除了这三个标准流外，Turbo C++ 还自动打开两个流 `stdprn` 和 `stdaux`。它们分别与打印机和辅助口有关。

#### 10.7.14 `fprintf()` 和 `fscanf()` 函数

除了基本 I/O 函数外，缓冲型 I/O 系统还有 `fprintf()` 和 `fscanf()`。它们除了用来操作磁盘文件这点外，这两个函数与 `printf()` 和 `scanf()` 完全相同。它们的原型是：

```

int fprintf(FILE *fp, char *控制字符串, ...);
int fscanf(FILE *fp, char *控制字符串, ...);

```

其中 `fp` 是由 `fopen()` 返回的文件指针。除了将其输出指向到由 `fp` 确定的文件之外，操作与 `printf()` 和 `scanf()` 完全相同。

为了说明如何使用这些函数，下面的程序用一个磁盘文件来存放一个简单的电话簿。可以输入姓名、电话，也可以给出一个名字查电话号码。

```

/* A simple telephone directory. */

#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

void add_num(void), lookup(void);
int menu(void);

main(void) /* fscanf - fprintf example */
{
    char choice;

    do {
        choice = menu();
    } while (choice != 'q');
}

```

```

        switch(choice) {
            case 'a': add_num();
                        break;
            case 'l': lookup();
                        break;
        }
    } while (choice != 'q');
    return 0;
}

/* Display menu and get request. */
menu(void)
{
    char ch;
    do {
        printf("(A)dd, (L)ookup, or (Q)uit: ");
        ch = tolower(getche());
        printf("\n");
    } while (ch != 'q' && ch != 'a' && ch != 'l');

    return ch;
}

/* Add a name and number to the directory. */
void add_num(void)
{
    FILE *fp;
    char name[80];
    int a_code, exchg, num;

    /* open it for append */
    if((fp=fopen("phone", "a"))==NULL) {
        printf("cannot open directory file\n");
        exit(1);
    }

    printf("enter name and number: ");
    fscanf(stdin, "%s%d%d%d", name, &a_code, &exchg, &num);
    fscanf(stdin, "%c"); /* remove CR from input stream */

    /* write to file */
    fprintf(fp, "%s %d %d %d\n", name, a_code, exchg, num);

    fclose(fp);
}

/* Find a number given a name. */
void lookup(void)
{
    FILE *fp;
    char name[80], name2[80];
    int a_code, exchg, num;

    /* open it for read */
    if((fp=fopen("phone", "r"))==NULL) {
        printf("cannot open directory file\n");
        exit(1);
    }

    printf("name? ");
    gets(name);
    /* look for number */
    while(!feof(fp)) {

```

```

        fscanf(fp, "%s2d2d2d", name2, &a_code, &exchg, &num);
        if(!strcmp(name, name2)) {
            printf("Xs: (%d) %d-%d\n", name, a_code, exchg, num);
            break;
        }
    }
    fclose(fp);
}

```

这个程序允许你往电话簿加入一个号码并可对其查询。它先显示一个短的菜单。当添加号码时，函数 `add_num()` 就被调用。当查询号码时就调用 `lookup()`。可以输入并运行这个程序。当输入数据时，必须确保用空格分开姓名、区码、前缀和号码。例如，下面是正确的输入：

ALEX 213 555 1234

在输入姓名和电话号码之外，检查一下文件 `phone`。正如你所期望的那样，它将与用 `printf()` 显示在屏幕上一样，具有同样的格式。

注意：虽然 `fprintf()` 和 `fscanf()` 是向磁盘文件读写各种数据的最容易的方法，但效率并不一定最高。因为它们以格式化的 ASCII 数据而不是二进制数据被写入，同在屏幕上显示的一样，调用这两个函数需要附加操作，如果要求速度快或文件很长时应使用 `fread()` 和 `fwrite()` 函数。

#### 10.7.15 删除文件

`remove()` 函数删除所指定的文件，用法为：

```
int remove(char *filename);
```

执行成功返回 0，否则返回非零值。

#### 10.8 非缓冲型 I/O——UNIX 型文件系统

因为 C 语言最早是在 UNIX 操作系统下开发的，第二组磁盘文件 I/O 早就形成了，它使用一些独立于缓冲型文件系统的函数。一些低级 UNIX 磁盘 I/O 函数在表 10-6 中列出。这些函数都需要把头文件 `IO.H` 连入任何要使用这些函数的程序的开头处。

表 10-6 UNIX 型非缓冲 I/O 函数

函数名	功能
<code>read()</code>	读一个数据缓冲区
<code>write()</code>	写一个数据缓冲区
<code>open()</code>	打开一个磁盘文件
<code>close()</code>	关闭一个磁盘文件
<code>lseek()</code>	搜索文件中指定字节
<code>unlink()</code>	使一个文件脱离连接



把由这些函数组成的磁盘 I/O 子系统叫做非缓冲型文件系统是因为编程者必须提供和维护所有的磁盘缓冲区, 函数不能自己完成这一工作。与 `getc()` 和 `putc()` 等函数不同, 它们不能从一个数据流中读写字符。函数 `read()` 和 `write()` 每次被调用时只能读或写一个完整信息缓冲区(这与 `fread()` 和 `fwrite()` 相似)。

本章一开头就已说明了非缓冲型文件系统不是由 ANSI 标准定义的。这说明如果使用这些函数就可能出现可移植性方面的问题。今后几年非缓冲文件系统的使用会减少。但因为很多现有的 C 语言程序中用到它们, 并且现有的各种 C 语言编译程序也支持它们, 所以本章还要介绍一下这些函数。

注意: UNIX 型文件系统使用的原型和有关必要信息可以在头文件 `IO.H` 找到。

### 10.8.1 `open()`, `creat()` 和 `close()` 函数

与 ANSI I/O 系统不同, UNIX 型系统不使用 `FILE` 型文件指针, 而是使用称为文件句柄的整型说明符。`open()` 的原型如下:

```
int open(char *filename, int mode, int access);
```

其中 `filename` 是任一有效文件名, `mode` 是下面三个由 `FCNTL.H` 定义的宏之一。

宏名	意义	值
<code>ORDONLY</code>	只读	1
<code>OWRONLY</code>	只写	2
<code>ORDWR</code>	读写	4

Turbo C++ 还允许一些其它选择项加在这几个基本类型后, 具体规定参考其它手册。

参数 `access` 仅与 UNIX 环境有关, 为具有移植性而保留的。Turbo C++ 也专为 DOS 版定义了一个叫做 `open()` 的函数, 如下:

```
int _open(char *filename, int mode);
```

此函数不使用 `access` 这一参数。本章中 `access` 设为 0。

成功调用 `open()` 后返回一个正整数叫做文件描述符。返回 -1 表明未能打开文件。文件描述符是其它 UNIX 型文件系统函数所要求的。文件描述符不同于 ANSI I/O 系统的文件指针。

你将常见到以下的调用格式:

```
if ((fd=open(filename, mode, 0)) == -1) {  
    printf("cannot open file\n");  
    exit(1);  
}
```

如果 `open()` 语句指定的文件不在磁盘上, 打开操作失败并不生成这个文件。

`close()` 的原型是:

```
int close(int fd);
```

如果 `close()` 函数返回 -1, 表明没能够关闭文件。例如, 磁盘已从驱动器中取出会出现这种情况。

调用 `close()` 表示释放该文件描述符, 以便其它文件可用。因为被打开的文件数目是有

限的，因此对不再需要的文件应该及时用 `close()` 函数关闭掉。更重要的是，关闭操作使操作系统的内部磁盘缓冲区里的全部信息写入磁盘。文件没有关闭常常导致数据丢失。

你可以使用 `creat()` 来为写操作生成一个新的文件。`creat()` 的原型如下：

```
int creat(char *filename, int access);
```

其中 `filename` 是任一有效文件名，参数 `access` 用来指定访问的模式和标明该文件为二进制文件还是文本文件。因为 `_creat()` 中的 `access` 与 UNIX 环境相联系，Turbo C++ 为 MS-DOS 版专门定义了 `creat()`，它用一个文件属性字节 (FILE ATTRIBUTE BYTE) 代替 `access`。DOS 中每个文件都与一个属性字节有关，它指定了各种信息位。表 10-7 是这个文件属性字节的意义。

表中的值可以相加，如果想生成一个只读隐含文件，可以用  $3 = (2+1)$  给 `access`。一般情况下，生成一个标准文件时 `access` 设为 0。

表 10-7 DOS 文件属性字节的值和含义

位号	值	含义
0	1	只读文件
1	2	隐含文件
2	4	系统文件
3	8	卷标号名
4	16	子目录名
5	32	数据档案
6	64	未定义
7	128	未定义

### 10.8.2 read()和 write()函数

一旦作为写操作的文件被打开后就可以用 `write()` 访问它。`write()` 函数的原型如下：

```
int write(int fd, void *buf, unsigned size);
```

每次调用 `write` 时，`size` 长度的字符就被从 `buf` 指向的缓冲区写到由 `fp` 指定的磁盘文件中。在写操作完毕后返回所写的字节数。写操作失败时返回 -1。

函数 `read()` 和 `write()` 的作用相反，原型为：

```
int read(int fd, void *buf, int size);
```

这里 `fd`，`buf` 和 `size` 与 `write` 中相同，只是 `read()` 把读入的数据放在由 `buf` 指向的缓冲区。读完后它返回实际所读字符数。在文件的物理结尾时返回 0，出错时返回 -1。

下例给出了非缓冲型 I/O 系统的一些用法。它从键盘上读入几行文字，写入磁盘文件，然后再从磁盘文件中读出来。

```
/* Read and write using UNIX-like I/O. */  
  
#include <fcntl.h>  
#include <io.h>
```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int input(char *buf, int fd1);
void display(char *buf, int fd2);

#define BUF_SIZE 128

main(void)
{
    char buf[BUF_SIZE];
    int fd1, fd2;

    if((fd1=_creat("test", O_WRONLY))==-1) { /* open */
        printf("cannot open file\n");
        exit(1);
    }

    /* read some text */
    input(buf, fd1);

    /* now close file and read back */
    close(fd1);

    if((fd2=open("test", 0, O_RDONLY))==-1) { /* open */
        printf("cannot open file\n");
        exit(1);
    }

    /* display text */
    display(buf, fd2);
    close(fd2);
    return 0;
}

/* Read some lines of text from the keyboard. */
input(char *buf, int fd1)
{
    register int c;

    printf("enter text, to stop enter 'quit' on a new line\n");
    do {
        for(t=0; t<BUF_SIZE; t++) buf[t]='\0';
        printf(": ");
        gets(buf); /* input chars from keyboard */
        if(write(fd1, buf, BUF_SIZE)!=BUF_SIZE) {
            printf("error on write\n");
            exit(1);
        }
    } while (strcmp(buf, "quit"));
}

/* Display the text. */
void display(char *buf, int fd2)
{
    for(;;) {
        if(read(fd2, buf, BUF_SIZE)==0) return;
        printf("%s\n", buf);
    }
}

```

### 10.8.3 unlink()函数

如果想从目录中删除一个文件可以使用函数 `unlink()`。虽然这一函数被看作是 UNIX 型 I/O 系统的一部分，但它可以删除任何文件。标准调用方式是：

```
int unlink(char *filename);
```

其中 `filename` 是指向一个任一文件的字符型指针。当它无法删除这个文件时，就返回出错信息(通常是 -1)。当文件不在磁盘上或磁盘有写保护时会出现这种情况。

### 10.8.4 随机访问文件和 lseek()函数

UNIX 型 I/O 系统下的随机访问文件 I/O 操作是通过调用 `lseek()` 函数支持的。它的原型如下：

```
long lseek(int fd, long numbytes, int origin);
```

这里 `fd` 由调用 `creat()` 或 `open()` 时返回的文件柄，`numbytes` 必须是个长整型。`origin` 必须是下面三个宏之一：

ORIGIN	宏名	实际值
文件开头	SEEK_SET	0
当前位置	SEEK_CUR	1
文件尾端	SEEK_END	2

因此，从文件开头搜索 `numbytes` 个字节时，起点 `origin` 用 `SEEK_SET`；从当前位置开始搜索时用 `SEEK_CUR`；从文件尾端开始搜索时用 `SEEK_END`。

`lseek()` 函数调用成功后返回偏移量。因此它返回的是一个长整型量，必须在文件开头加以说明。调用失败时返回 -1。

使用 `lseek()` 的一个实例是对前面讨论的 DUMP 程序作一些改动以适应 UNIX 型 I/O 系统。它不仅给出 `lseek()` 的用法，还说明了许多 UNIX 型 I/O 函数的用法。

```
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
#include <ctype.h>
#include <stdlib.h>

#define SIZE 128

char buf[SIZE];
void display(int num);

main(int argc, char *argv[]) /* read buffers */
{
    char s[10];
    int fd, sector, numread;
    long pos;
```

```

if(argc!=2) {
    printf("You forgot to enter the filename.");
    exit(1);
}

if((fd=open(argv[1], O_RDONLY, 0))!=-1) { /* open */
    printf("cannot open file\n");
    exit(1);
}

for(;;) {
    printf("\n\nbuffer: ");
    gets(s);

    sector = atoi(s); /* get the sector to read */
    if(sector<0) break;

    pos = (long) (sector*SIZE);
    if(lseek(fd, pos, SEEK_SET)!=pos)
        printf("seek error\n");

    numread = read(fd, buf, SIZE);
    if(numread==-1) {
        printf("File Error");
        break;
    }

    display(numread);
}
close(fd);
return 0;
}

void display(int numread)
{
    int i, j;

    for(i=0; i<=numread/16; i++) {
        for(j=0; j<16; j++) printf("X3X", buf[i*16+j]);
        printf(" ");
        for(j=0; j<16; j++) {
            if(isprint(buf[i*16+j])) printf("Xc", buf[i*16+j]);
            else printf(".");
        }
        printf("\n");
    }
}

```

## 第十一章 高级数据类型

到现在为止, 本书仅仅使用了五种基本数据类型。尽管这对于多数程序来说是足够的, 但它们不能满足许多程序编制人员的要求。实际上, C 语言允许各种类型修饰符用于基本数据类型。你已经看到两种类型修饰符: `long` 和 `short`。除了这些以外, Turbo C++ 还支持另外四种修饰符类别:

- 访问修饰符
- 存储类型修饰符
- Turbo C++ 专门函数类型修饰符
- Turbo C++ 专门存储模式修饰符

类型修饰符作为基本数据类型的前缀, 在说明语句中, 变量说明被扩展为如下形式:  
类型修饰符 类型说明符 变量表

存储模式、段和覆盖很复杂, 超出了本书的范围。因此, 存储模式修饰符 `near`, `far` 和 `huge` 在这里不讨论(在附录 D 中对它们进行说明)。一会儿我们将研究其它的修饰符。

除了类型修饰符, 这一章还研究一个特殊类型的指针: 函数型指针。

### 11.1 访问修饰符

C 语言中有两个用于控制访问和修改变量方式的修饰符。它们分别是常量(`const`)和易变量(`volatile`)。它们通常也叫做类型限定符(`type qualifiers`)。

#### 11.1.1 `const` 常量

用 `const` 修饰的变量在程序的运行过程中始终保持不变。你可以给它们赋初值。例如:  
`const float version=3.20;`

将产生浮点数变量 `version`, 其值不能被程序修改。不过, 它可以在其它类型的表达式中使用。`const` 型变量可以在其初始化时直接被赋值, 或通过某些硬件的方法来赋值。在变量说明中采用 `const` 修饰符可以确保变量的值不会被程序的其它部分改变。

`const` 型变量有一个非常重要的用途——可以防止函数的参数被其修改。也就是说, 当一个指针传递给函数时, 函数有可能修改指针所指的实值。不过, 如果指针在参数说明中用 `const` 来限定, 则它所指向的值不会被函数修改。例如, 下面这段程序中的函数 `code` 将每一个字符变换为字母表中其后字符。即“A”变成“B”, 依次类推。在参数说明中用 `const` 是为了保证函数中的代码不会改变参数指向的目标。

```
#include <stdio.h>

void code(const char *str);

main(void)
{
    code("this is a test");
    return 0;
}
```

```
void code(const char *str)
{
    while(*str) printf("%c", (*str++)+1);
}
```

如果，在某些情况下，用修改参数的方式来书写函数 code()，则编译将不会成功。例如，用如下形式书写 code()：

```
/* this is wrong */
void code(const char *str)
{
    while(*str) {
        *str = *str + 1;
        printf("%c", *str++);
    }
}
```

将看到编译错误信息：

cannot modify a const object in function code

const 的第二个用途是为一个事实提供验证：即用户程序是否在修改一个变量。记住一个 const 型变量可以在程序以外被说明。例如，硬件设备可以设置其值。用 const 定义一个变量可以保证变量发生的任何改变都是由于外部原因引起的。

### 11.1.2 volatile 易变量

修饰符 volatile 用于告诉编译程序，该变量的值可以通过程序中非明确定义的方法来改变。例如，一个全局变量的地址可以传递给操作系统的时钟例行程序，从而该变量可以用于存储系统的实时钟值。在这种情况下，变量的内容在程序中没有明确的赋值语句对它赋值时，也会发生改变。在 Turbo C++ 中变量外部变化是十分重要的，原因是在假定表达式内变量内容不变的前提下，Turbo C++ 自动地优化某些表达式。例如，假定 clock 被计算机时钟装置每隔 10 秒改变一次。如果它未用 volatile 定义，则下面的程序不能正常工作。

```
int clock, timer;

timer = clock;
/* do something */
printf("elapsed time is %d\n", clock-timer);
```

因为不是由程序改变的而且未用 volatile 定义，则 Turbo C++ 在 printf() 语句中会在不再检查 clock 值的情况下来优化代码。不过，如果将 clock 定义为：

```
volatile int clock;
```

则不会发生这样的问题，clock 的值在每次碰到时都被检查。

尽管开始看起来很奇怪，const 和 volatile 可以同时使用。例如，假定 0x30 是一个只随外部条件而变化的口地址值，那么就恰好需要用下述说明来避免偶然因素所产生的副作用。

```
const volatile unsigned char *port=0x30;
```

## 11.2 存储类型说明符

C 语言支持四类存储修饰符:

```
auto
extern
static
register
```

这些说明符用于告诉编译程序紧随其后的变量应如何存储。变量存储类型定义符放在其它说明之前。其一般形式是:

存储类修饰符 类型修饰符 变量表

以下分别讨论每一种修饰符。

### 11.2.1 auto(自动变量)

`auto` 修饰符用于说明局部变量,但程序中很少使用此关键字,因为它是局部变量的缺省说明。

### 11.2.2 extern(外部变量)

迄今你用到的程序都很小,实际上小到还不满屏幕的 25 行。然而,在实际编程工作中,程序倾向于更长更大。尽管 Turbo C++ 在编译时非常之快,但文件的编译时间持续增长仍使人烦恼。当这种情况发生时,可将程序分为两个或更多的文件。这样,一个文件的小改动不需要重新编译整个程序,这就意味着在大的工程中缩短了实质时间。

C 语言包括关键字 `extern`,可以帮助支持多重文件操作。尽管多重文件和分别编译在附录 B 中有详细描述,因为它与 `extern` 有关,我们将在这里简略介绍。

由于 C 语言允许将大型程序分成若干独立模式文件分别编译,然后将它们连接在一起,从而达到提高编译速度和便于管理大型软件工程的目的,所以必须设法将整个程序所需的全局变量通知所有的程序模块文件。对于一个全局变量只能说明一次。如果试图在一个文件中用同样的名字说明两个全局变量,编译程序就会简单地挑选一个来用(不过,在 C++ 中,用相同的名字定义两个全局变量是错误的)。如果试图说明多重文件程序中每个程序所需的全局变量,将发生错误。虽然编译程序在编译时不会显示错误信息,但实际上你是在试图把每个变量建立两份(或更多)。这样,当 Turbo C++ 试图把用户程序模块连在一起时就会遇到麻烦。解决这一问题的办法是在一个文件中说明所有全局变量,而在其它文件中用 `extern` 说明。如图 11-1 所示。

在第二个文件中,全局变量说明表是从第一个文件中复制过来的,但在其前面加了一个外部变量说明符 `extern`。说明符 `extern` 告诉编译程序,在其后的变量类型和名称已经在别的文件中说明。换句话说, `extern` 使编译程序知道这些全局变量的类型的名称,而不再为它们分配内存。当这两个模块连接时,所有的外部变量都得到统一。

在说明了全局变量的文件中,若在函数内部使用全局变量,也可以选择使用 `extern` 说明符,虽然这样做是不必要的,而且也很少这样做。

下页的程序说明此用法:



File 1	File 2
<pre> int x, y; char ch; main(void) {     .     . }  func1(void) {     x = 123; } </pre>	<pre> extern int x, y; extern char ch; func22(void) {     x = y/10; }  func23(void) {     y = 10; } </pre>

图 11-1 在分别编译的程序模块中使用全局变量

```

int first, last; /* global definition of first
                  and last */

main(void)
{
    extern int first; /* optional use of the
                      extern declaration */

```

虽然在说明全局变量的文件中也可以使用 `extern` 变量说明，但这并不是必要的。如果 C 编译程序遇到一个未曾说明的变量，则将检查它是否和某一个全局变量相匹配。若是，则编译程序就认为该变量就是已说明的这个全局变量。

### 11.2.3 static variables(静态变量)

静态变量在其所在的函数或文件中是长久变量。它们不同于全局变量，因为它们在函数或文件以外是未知的，但它们在函数调用之间可以保存其值。由于具有这一特征，静态变量很有用处，特别是当你编写通用函数和库函数时，这些函数可被其它编程者使用。由于静态局部变量与静态全局变量有较大不同，将单独讨论它们。

### 11.2.4 static local variables(静态局部变量)

当在局部变量名前加上静态变量说明符时，编译程序将为该变量建立长久存储单元，其方法与全局变量差不多。静态局部变量和全局变量的根本差别在于静态局部变量只有在它被说明的函数或复合语句中是有效的。用简单的说来说，静态局部变量是一种在两次函数调用之间仍然保存其值的局部变量。

在 Turbo C++ 中可以使用静态局部变量，这对于保持函数的独立性是很重要的。因为有几种类型的例子程序需要在多次调用之间保存其变量的值，如果不能用静态变量的话，就必须使用全局变量，这等于对各种可能产生的干扰因素敞开了大门。说明函数需要

静态局部变量的另一个较好的例子是一种在旧数的基础上产生新数的数字序列发生函数。你可以定义一个全局变量来保存这个数值。然而，每当这个函数在程序中使用，都得记住去说明一个全局变量，而且要确保它和已经说明过的其它全局变量不冲突，这真是一个大倒退。较好的解决办法是在函数中说明一个静态局部变量，并用它来保存所产生的数。如以下程序段所示：

```
#include <stdio.h>

int series(void);
main(void)
{
    int i;

    for(i=0; i<10; i++)
        printf("%d ", series());
    return 0;
}

series(void)
{
    static int series_num;

    series_num=series_num + 23;
    return(series_num);
}
```

在这个例子中，变量 `series_num` 在函数调用之间一直存在，而不是像普通的局部变量那样时生时灭。这就意味着每次调用 `series()` 时都能产生一个以上的数为基础的新的序列数，而又不需要因此而定义全局变量 `series_num`。

你大概已经注意到了，上例中的函数 `series()` 有些特别之处，就是变量 `series_num` 没有赋过初值。这就意味着第一次调用该函数时 `series_num` 是一个随机值。这在某些应用场合下是允许的。而大部分序列数产生函数都需要有一个明确定义的初始点。要做到这一点就需要在第一次调用 `series()` 之前先给 `series_num` 赋初值，只要 `series_num` 保证是全局变量则很容易做到这一点。而我们之所以把 `series_num` 定义为静态变量是为了不把它定义为全局变量。这就引出了静态变量的第二个用法。

### 11.2.5 static global variables(静态全局变量)

当在全局变量说明前加上静态变量修饰符 `static` 时，它就会告诉编译程序建立这样一个全局变量，该变量只在它被定义的那个文件中是可用的。也就是说即使这个变量是全局变量，但其它文件中的程序并不知道它的存在，也无法改变其内容。这就是说该变量不易受到副作用的影响。所以在静态局部变量不能满足要求的少数情况下，你可以建立一个小型的文件，它只包含需要使用这些全局变量的函数，然后独立地编译该文件。这样，就可以放心地使用其中的函数而不必担心副作用的影响。

为了说明如何使用静态全局变量，我们改写上例中的数字序列程序。通过调用第二个函数 `series_start()`，用一个值作为该序列的初始值。整个文件包含 `series()`，`series_start()` 和 `series_num`，如下所示：

```
/* this must all be in one file - preferably by itself */

static int series_num;
```

```

series(void)
{
    series_num = series_num+23;
    return(series_num);
}

/* initialize series_num */
void series_start(int seed)
{
    series_num=seed;
}

```

首先调用函数 `series_start()`，用一个已知的整数赋给数字序列作为初始值，然后调用函数 `series()` 产生数字序列的下一个元素。

记住，静态局部变量的名字仅仅在定义它们的函数或复合语句中是有效的，静态全局变量的名字也仅仅在它们所在的文件中是有效的。这就意味着如果把函数 `series()` 和 `series_start()` 放在一个独立的文件中，可以使用这些函数，但无法引用变量 `series_num`，它隐藏于你的程序之外。实际上，甚至可以在自己的程序中(当然是在另一个文件中)用 `series_num` 这一名称说明另一个变量，这不会造成任何混淆。实质上，静态存储修饰符 `static` 允许定义只在某几个函数中有效的变量，而不会使其它函数产生混淆。

静态变量使得你可以把部分程序隐藏于其它部分之外。当试图管理一个复杂的大型程序时这一点尤为突出。使用静态存储修饰符 `static` 可以编出更加通用的函数，然后把它们装进库里供以后使用。

### 11.2.6 register variables(寄存器变量)

C 语言中另一个重要的存储类型修饰符是 `register`，传统上它只能用于 `int` 和 `char` 型变量。在 C 语言最初版本中，修饰符 `register` 要求编译器将用这个修饰符定义的变量保存在 CPU 的寄存器，而不是像普通的变量那样在内存中。这就意味着使用寄存器变量比使用内存变量的操作速度快得多。因为这种变量不需要通过内存访问来确定和修改其值。(内存访问比寄存器访问耗费更多的时间。)不过，ANSI C 标准已经扩展了 `register` 的含义。现在，`register` 可以被那些企图用尽快访问 `register` 变量的数据类型和编译程序所采用。不过，在 Turbo C++ 中，当采用 `char` 型和 `int` 型变量时，一般是将它们放入 CPU 的寄存器中。

寄存器变量修饰符只适用于局部变量和函数的形式参数。所以，寄存器全局变量是不允许的。

修饰符 `register` 仅是编译程序的简单要求，而不是一个命令。理解这一点十分重要。编译程序可以忽略它。这样做的原因很简单：寄存器的数目有限或有其他快速访问内存方法。于某些方面而言这样做代价太大，而且其后的 `register` 变量也将被当成普通变量。对 Turbo C++ 而言，在任何函数中实际上只可能有两种变量存于 CPU 的寄存器中。你也不必为定义了过多的寄存器变量而担心，Turbo C++ 会自动地将超过限制数目的寄存器变量当作非寄存器变量来处理。

访问寄存器变量速度快这个事实使得寄存器变量用于循环控制。以下的例子说明如何

定义一个寄存器变量并用它来控制循环。该函数计算一个以整数为底的指数  $m^e$ ：

```
int_pwr(int m, register int e)
{
    register int temp;

    temp = 1;

    for( ; e ; e--) temp = temp * m;
    return temp;
}
```

在这个例子中，`e` 和 `temp` 都被定义为 `register` 变量，因为它们都用在循环中。在实际情况下，寄存器变量只用在同一变量名频繁出现的地方。

为了看清 `register` 变量的特别之处，下面的程序测量两个 `for` 循环的执行时间，二循环仅仅是控制量的类型不同。这个程序用到了 Turbo C++ 标准库中的函数 `time()`。

```
/* This program shows the difference a register
   variable can make to the speed of program
   execution.
*/
#include <stdio.h>
#include <time.h>

unsigned int i; /* non-register */
unsigned int delay;

main(void)
{
    register unsigned int j;
    long t;

    t = time('\0');
    for(delay=0; delay<10; delay++)
        for(i=0; i<64000; i++);
    printf("time for non-register loop: %ld\n", time('\0')-t);

    t = time('\0');
    for(delay=0; delay<10; delay++)
        for(j=0; j<64000; j++) ;
    printf("time for register loop: %ld", time('\0')-t);

    return 0;
}
```

当运行这个程序时，会发现寄存器控制循环只用了非寄存器控制循环近一半的时间。

最后一点：Turbo C++ 将自动地将最先两个字符和整型局部变量变成寄存器变量。这就是 `i` 为全局的原因。

### 11.3 赋值语句中的类型转换

类型转换是指不同类型的变量混合使用时的情况。在赋值语句中，类型转换的规则是很简单的：等号右边的值转换为符号左边变量所属的类型。如例所示：

```
#include <stdio.h>

int x;
char ch;
float f;
```

```

main(void)
{
    x = 1000;
    ch = x;    /* 1 */
    printf("%d ", ch);

    f = 100.23;
    x = f;     /* 2 */
    printf("%d ", x);

    ch = 'a';
    f = ch;    /* 3 */
    printf("%f ", f);

    x = 100;
    f = x;     /* 4 */
    printf("%f ", f);

    return 0;
}

```

在执行语句第一行中，左边整型变量 `x` 的高 8 位被截去，而只将其低 8 位赋给 `ch`，如果 `x` 的值在 0 到 256 之间，则 `ch` 与 `x` 的值相等。否则，`ch` 的值只代表 `x` 的低 8 位。在第二行中，`x` 只接收了 `f` 的非小数部分。在第三行中，`f` 将一个 8 位数值转换为 `ch` 要求的浮点形式存储。第四行也一样，只不过是 将一个 16 位整数转换为浮点型。这个程序输出：

```
-24 100 97.00 100.00。
```

如果将整数转换为字节，或将长整数转换为整数，其基本规则是将多出来的高位截去。这就意味着整数变为字符时将失去 8 位，而长整数变整数时失去 16 位。

表 11-1 概要地说明这些类型转换。你必须记住重要的一点：`int` 型到 `float` 型到 `double` 型等等，并不增加任何准确度和精确度。这种转换只改变数值的表示形式。

在使用表 11-1 时，若遇到没有列出的类型转换，可以将一种转换分步多次完成。例如，若要将 `double` 型转换为 `int` 型，则先将 `double` 型转换为 `float` 型，然后再将 `float` 型转换为 `int` 型。

表 11-1 普通类型转换的结果

目标	源	可能的数据丢失
<code>char</code>	<code>unsigned char</code>	若值 > 127 则目标为负数
<code>char</code>	<code>short int</code>	高 8 位
<code>char</code>	<code>int</code>	高 8 位
<code>char</code>	<code>long int</code>	高 24 位
<code>short int</code>	<code>int</code>	无
<code>short int</code>	<code>long int</code>	高 16 位
<code>int</code>	<code>long int</code>	高 16 位
<code>int</code>	<code>float</code>	小数部分，也许更多
<code>float</code>	<code>double</code>	精度降低，结果四舍五入
<code>double</code>	<code>long double</code>	精度降低，结果四舍五入

## 11.4 函数类型修饰符

Turbo C++定义了三种类型修饰符只适用于函数。他们是 `pascal`、`cdecl` 和 `interrupt`。这些修饰符不是由 ANSI 标准定义的而是由 Turbo C++提供, 用于获得 PC 编程环境最大便利。

### 11.4.1 pascal

`pascal` 类型修饰符告诉编译程序用 `pascal` 型参数传递来对函数参数操作而不是 Turbo C++通常的方法。这允许有两种可能性。第一, 可以建立能为其他编译程序所用的用 Turbo C++写的函数。第二, 可以在 C 程序的顶部将用 `pascal` 编译程序的库程序定义为 `pascal` 类型。

例如, 函数 `int_pwr()` 此版本可以用 `pascal` 编译程序来编译:

```
/* compile for Pascal compilers */
pascal int_pwr(int m, register int e)
{
    register int temp;

    temp = 1;

    for( ; e; e--) temp = temp * m;
    return temp;
}
```

如果想编译一个文件中所有 `pascal` 类型的函数, 不用 `pascal` 类型描述符而用主菜单选择项 `option` 也可做到。然后, 选择 `Compiler` 和 `code Generation`。将这些函数看成是为 `pascal` 编译程序所用的。

### 11.4.2 cdecl

关键字 `cdecl` 是 `pascal` 的反义词, 因为它告诉 Turbo C++编译一个函数使得它的参数按其于 C 语言函数相适的方式来传递。只有当编译程序已经用 `pascal` 调用集合设置而你有一些文件不想用 `pascal` 模式编译时才用到。

### 11.4.3 interrupt

修饰符 `interrupt` 告诉 Turbo C++它修饰的函数将被用于中断处理器。这造成函数每次进入时保存 CPU 寄存器, 而且函数用 `IRET` 指令(`return from interrupt`)退出。

## 11.5 指向函数的指针

C 语言中一个特别容易引起混淆而又相当有用的特征是函数指针。函数指针可以看作一个新的数据类型。尽管函数不是变量, 但在内存中仍有其物理地址, 可以赋给指针。赋给指针的地址就是函数的入口地址, 从而该指针就能用来代替函数名。它也使得函数可以作为实参传递给其它函数。

为了解函数指针是如何工作的, 必须对 Turbo C++中函数是如何被编译和调用的有所了解。第一, 当函数被编译时, 源代码转换成了目标代码, 同时确定了函数的入口点。当用户程序运行时, 若调用了函数, 机器语言的“`call`”就指向了这个入口点。因而指向函数的指针实际上包含了函数入口点的内存地址。

函数的地址可用不带有括号和参数的函数名得到。(这类似于获得数组地址的方法,即只用不带下标的数组名。)例如,考虑下面的程序,请仔细注意它的说明部分:

```
#include <stdio.h>
#include <string.h>

void check(char *a, char *b, int (*cmp)());

main(void)
{
    char s1[80], s2[80];
    int (*p)();

    p = strcmp; /* assign pointer to function */

    printf("enter the first string: ");
    gets(s1);
    printf("enter the second string: ");
    gets(s2);

    check(s1, s2, p);

    return 0;
}

void check(char *a, char *b, int (*cmp)())
{
    printf("testing for equality\n");
    if(!(*cmp)(a, b)) printf("equal");
    else printf("not equal");
}
```

正如程序中显示,函数指针用这条语句定义:

```
int (*p)();
```

在函数 `check()` 被调用时,有两个字符指针和一个函数指针被作为参数传递。在函数 `check()` 中,参数被说明为字符指针和一个函数指针。注意函数指针是如何说明的。在说明其它函数指针时必须准确地使用同样的方式,除非函数的返回类型不同。将 `*cmp` 用括号括起来,这对于编译程序正确地翻译该语句是很必要的,而不带有括号的 `*cmp` 会造成编译混乱。

一旦进入 `check()` 中,你就会明白函数 `strcmp()` 是如何被调用的。语句

```
(*cmp)(a, b)
```

执行了对该函数的调用,在这里即 `strcmp()` 函数。`cmp` 为该函数的指针,且实参为 `a` 和 `b`。`*cmp` 的括号是必需的。这条语句也再现了使用函数指针来调用它所指的函数的一般形式。

注意,也可以在调用 `check()` 时直接使用 `strcmp`, 如下所示:

```
check(s1, s2, strcmp)
```

这条语句可不必增加指针变量。

你也许会问,有准用这种方法写程序呢?在本例中非但没有得到好处,却带来了明显的混乱。不过,函数指针有许多优点。其一是创建一个类函数,可以用于执行不同数据类型上的相似类型操作。仔细研究一下上面程序的高级版本你就会知道这类用法的益处了。在这个程序中, `check()` 用于检查字母等式或数字等式,简单地用不同的比较函数来调用。

```

#include <ctype.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void check(char *a, char *b, int (*cmp)());
int numcmp(char *a, char *b);

main(void)
{
    char s1[80], s2[80];

    gets(s1);
    gets(s2);

    if(tolower(*s1) <= 'z' && tolower(*s1) >= 'a')
        check(s1, s2, strcmp);
    else
        check(s1, s2, numcmp);

    return 0;
}

void check(char *a, char *b, int(*cmp)())
{
    printf("testing for equality\n");
    if(!(*cmp)(a,b)) printf("equal");
    else printf("not equal");
}

numcmp(char *a, char *b)
{
    if(atoi(a)==atoi(b)) return 0;
    else return 1;
}

```

## 11.6 动态分配

在结束高级数据类型讨论之前，有必要讨论一下 C 语言的动态分配系统，它允许在用户程序执行过程中动态地创建变量。

在 C 语言中有两种原始方法可以在计算机主存中存储信息。第一种是用 C 语言定义的全局和局部变量。在全局变量情况中，存储器由栈空间分配。在每一种情况下，都要求程序员事先了解每个位置的存储量。第二种途径是，信息可用 C 语言动态分配系统存储。用这种方法(缺省使用小存储模式)，信息存储从程序和其永久存储空间、栈之间的自由内存区域来分配。

用 11-2 表示一个 C 语言程序在内存中的情况。栈是向下增长的，所以它所需内存决定于你的程序设计。例如，一个带有许多递归函数的程序要求较大的栈空间，而无递归函数的程序所需栈空间较小(记住，这是因为局部变量存于栈中，每个函数的递归调用要求额外的栈空间)。在程序执行过程中程序和全局变量要求的内存是固定的。动态分配要求的内存来自自由内存区域。在一些特殊的情况下，自由的内存将被耗尽。

记住，本节讨论 C 语言动态分配策略。尽管这些函数是由 C++ 支持的，C++ 还支持别的动态分配策略。然而，你很可能遇到 C 型动态分配函数，所以了解它们是十分重要的。



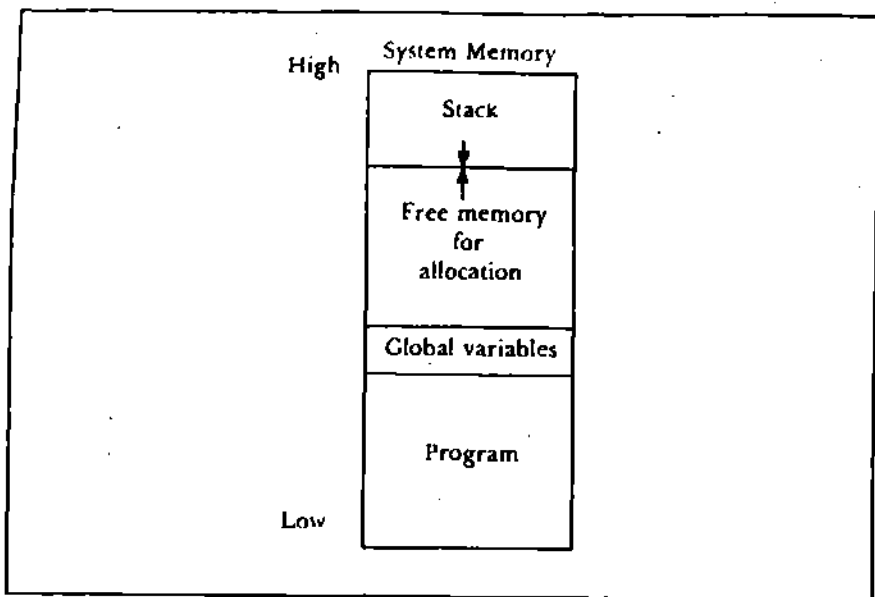


图 11-2 C 程序内存使用一览图

### 11.7 分配和释放内存

C 语言分配系统的核心是函数 `malloc()` 和 `free()`。(实际上, 为了增加灵活性, Turbo C++ 还有许多其它动态分配函数, 但这两种是最重要的。)它们在一起工作以建立和保存变量存储表。函数 `malloc()` 分配内存而函数 `free()` 释放内存。这就是说, 每次调用 `malloc()` 则申请内存, 剩下的自由内存则被分配一部分。每次调用 `free()`, 内存就会被归还系统。任何用到这两个函数的程序需包含头文件 `SIDLIB.H` (在 C++ 中, 必须包含所有相应头文件)。

函数 `malloc()` 原型如下:

```
void *malloc(unsigned number_of_bytes);
```

它返回一个 `void` 型指针, 这意味着你可以指定它为任何类型的指针。成功调用之后, `malloc()` 将返回指向分配的内存空间, 如果分配出错, `malloc()` 返回空。你可以用 `sizeof` 来判定每种数据类型所需的确切字节数。用这种方法可以使程序适用于不同的系统。

函数 `free()` 是 `malloc()` 的反义, 它将原先分配的内存返回给系统。一旦内存被释放, 它就可为其后的 `malloc()` 所用。函数 `free()` 的原型为:

```
void free(void *p);
```

需要记住的唯一重要的一点是: 绝对不能用无效参数调用 `free()` 使得自由块表被破坏。

下面这段短程序为 40 个整数分配了足够的存储空间, 打印其值, 并随后将空间释放回系统。

```
#include <stdio.h>
#include <stdlib.h> /* needed by malloc() and free() */

main(void) /* short allocation example */
{
    int *p, t;

    p = malloc(40*sizeof(int));
    if(!p) /* make sure it's a valid pointer */
```

```

        printf("out of memory\n");
    else {
        for(t=0; t<40; ++t) *(p+t) = t;
        for(t=0; t<40; ++t) printf("%d ",*(p+t));
        free(p);
    }

    return 0;
}

```

记住，在用 `malloc()` 返回的指针之前，要通过检查返回值非零来确保分配成功。不要用值为 0 的指针，因为这样会使用户系统受到严重破坏。

动态分配当你事先不知道要处理多少数据时十分有用。尽管动态分配程序既长又繁琐，下面的程序可看出它的优点。它首先询问用户要对多少数求平均，然后分配一个足以容纳它们的数组，接收输入并求平均值，最后释放这个数组。

```

/* This program averages an arbitrary number of integers. */
#include <stdlib.h>
#include <stdio.h>

main(void)
{
    int *p;
    int num, i, avg;

    printf("enter number of integers to average: ");
    scanf("%d", &num);

    /* allocate space */
    if((p = malloc(sizeof(int)*num)) == NULL) {
        printf("allocation error");
        exit(1);
    }

    for(i=0; i<num; i++) {
        printf("%d: ", i+1);
        scanf("%d", &p[i]);
    }

    avg = 0;
    for(i=0; i<num; i++) avg = avg + p[i];

    printf("average is: %d", avg/num);

    free(p);

    return 0;
}

```

你在本书第三部分可以看到，尽管 C++ 支持 `malloc()` 和 `free()`，它还提供动态分配的自转换策略。

## 第十二章 用户定义的数据类型

C 语言允许建立五种用户数据类型。第一种是结构，它是一种归在同一名字下各种变量的组合。有时也称为数据类型的集成体。第二种用户定义类型是位域，它是一种结构的变体，允许访问一个字节的各个位。第三种叫做联合，它使得两个域或两个以上的不同变量类型公用同一内存块。第四种数据类型叫做枚举，它是符号目录表。第五种用户定义类型是通过使用 `typedef` 创立的，它为一已存在类型产生一个新名。本章将研究这些用户定义类型。

### 12.1 结构

在 C 语言中，结构是一种被命名为一个标识符的各种变量的集合，这种方法为将各种信息汇集在一起提供了一个十分便利的途经。结构定义可以用来确立结构变量的格式。构成一个结构的各个变量称为结构元素(C 语言中的结构与 Pascal 中的记录相同)。

通常，结构中的所有元素有着彼此逻辑相关的联系。例如，通讯录中的姓名和地址信息通常可以用一个结构来表示。下面这段代码表示通讯录表的一个结构范例。关键字 `struct` 告诉编译程序一个结构的形式已经被定义了。

```
struct addr {  
    char name[30];  
    char street[40];  
    char city[20];  
    char state[3];  
    unsigned long int zip;  
};
```

注意这个定义用分号结束。这是由于结构定义本身为一条语句。而且结构标识符 `addr` 代表这个指定的数据结构。

这段代码有一点，结构定义并没有说明任何实际的变量。定义了的仅仅是这些数据的形式。为了说明这种结构的变量，必须写成：

```
struct addr addr_info;
```

这条语句说明一个 `addr` 形式的结构变量，它被命名为 `addr_info`。当定义一个结构时，实质上是将结构元素的各种复杂的变量形式组合在一起。在说明一个这种结构形式的实际变量前，它实质上是虚设的。

C 语言自动地分配适当的内存块结构或这个结构的各个变量。图 12-1 表明 `addr_info` 内存分配情况表。

也可以在定义结构的同时说明一个或几个结构变量，例如：

```
struct addr {  
    char name[30];  
    char street[40];  
    char city[20];  
    char state[3];  
    unsigned long int zip;  
} addr_info, binfo, cinfo;
```

这条语句定义了一个称为 `addr` 的结构类型，而且说明了变量 `addr_info`、`binfo` 和 `cinfo`

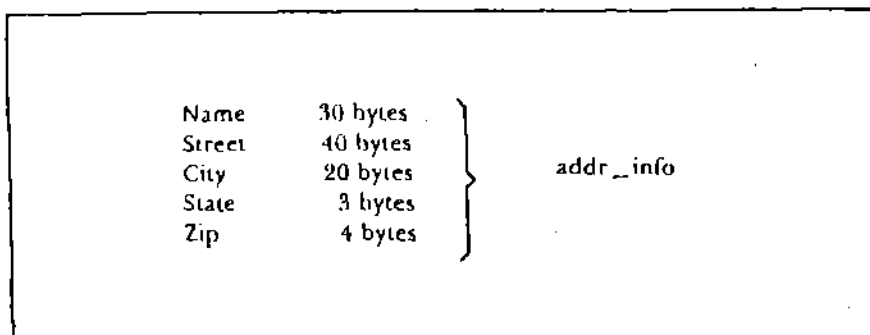


图 12-1 结构 addr\_info 内存分配表

属于这种结构类型。

如果仅需要一个结构变量，那么结构标识符可以省略，如：

```
struct {
    char name[30];
    char street[40];
    char city[20];
    char state[3];
    unsigned long int zip;
} addr_info;
```

这条语句说明了名为 addr\_info 的结构变量，它沿袭前边的结构形式。

结构定义的一般形式为：

```
struct 结构名{
    类型 变量名
    类型 变量名
    ....
} 结构变量
```

这里结构名和结构变量可忽略，但不能都忽略。

### 12.1.1 访问结构元素

结构中的单个元素可以用元素选择操作符“.”来访问。(元素选择操作符通常也叫做点操作符。)例如，下面的语句是将 12345 赋给结构变量 addr\_info 元素 zip：

```
addr_info.zip=12345;
```

结构变量名后跟点操作符而且元素名代表结构中的单个元素。所有结构元素都可以用下面这种常用方式访问它们。

结构名.元素名

因此，为了将 zip 的内容打印到屏幕上，可以写为。

```
printf("%d", addr_info.zip);
```

这条语句将显示结构变量 addr\_info 中元素 zip 的内容。

同样的形式，也可以用字符数组 addr\_info.name 来调用函数 gets()：

```
gets(addr_info.name);
```

这条语句表示将一个字符型指针传递给元素 name 的首址。

如果想访问 `addr_info.name` 的各个元素，可以检索 `name` 中的内容。例如，想再次显示 `addr_info.name` 中一个字符的内容，可用下列程序：

```
register int;  
for (t=0; addr_info.name[t]; ++t) putchar(addr_info.name[t]);
```

## 12.2 结构数组

结构最常见的用法是结构数组。为了说明一个结构数组，首先必须定义一个结构，然后说明这种类型的一个数组变量，例如，为了说明一个前面已定义过的 `addr` 形式的有 160 个元素的数组，你可以写成：

```
struct addr addr_info[100];
```

为了访问某一确定的结构，必须指定结构名的下标。例如，为了打印第三个结构中 `zip` 的内容，可以写成：

```
printf("%d", addr_info[1].zip);
```

同所有数组变量一样，结构数组的下标从 0 开始。

### 12.2.1 通讯录实例

为了帮助理解结构和结构数组的用法，我们来研究一个简单的通讯录例程。它使用结构数组来保存地址信息。程序中的各个函数以各种不同的方式联接结构和它们的元素，从而叙述了结构的用法。

在这个例子中，存储的信息包括：

姓名(name)

街道(street)

城市、州、邮政编码(city, state, zip)

为了定义基本结构 `addr` 来保存这些信息，可以这样写：

```
struct addr {  
    char name[30];  
    char street[40];  
    char city[20];  
    char state[3];  
    char zip[10];  
} addr_info[SIZE];
```

注意此结构使用字符串来保存邮政编码而不是无符号整数。这种用字符的邮编方法与数字一样，在加拿大和其它国家适用。数组 `addr_info [SIZE]` 沿袭结构 `addr`，这里的 `SIZE` 可用于特殊需要。

第一个函数是 `main()`，如下所示：

```
main(void)  
{  
    char choice;  
  
    init_llist();  
  
    for(;;) {  
        choice = menu();  
        switch(choice) {  
            case 'e': enter();  
            break;
```

```

        case 'd': display();
            break;
        case 's': save();
            break;
        case 'l': load();
            break;
        case 'q': return 0;
    }
}
)
)

```

首先, 函数 `init_list()` 用来将数组中结构的姓名域中第一个字节赋以空字符, 使得结构数组可用。这段程序假设, 若姓名域为空, 则这个结构变量没有用过。`init_list()` 函数如下:

```

/* Initialize the addr_info array. */
void init_list(void)
{
    register int t;

    for(t=0; t<SIZE; t++) *addr_info[t].name = '\0';
    /* a zero length name signifies empty */
}

```

函数 `menu()` 显示选择项信息, 并返回用户所选的项。

```

/* Get a menu selection. */
menu(void)
{
    char ch;

    do {
        printf("(E)nter\n");
        printf("(D)isplay\n");
        printf("(L)oad\n");
        printf("(S)ave\n");
        printf("(Q)uit\n\n");
        printf("choose one: ");
        ch = getche();
        printf("\n");
    } while(!strchr("edlsq", tolower(ch)));
    return tolower(ch);
}

```

该函数使用 Turbo C++ 的库函数: `strchr()`, 其函数原型如下:

```
char *strchr(char *str, char ch);
```

此函数查找由 `str` 所指的串, 找出 `ch` 中的字符。如果找到了字符, 就返回指向这个字符的指针, 这被定义为真值。如果未找到, 则返回一个空值。定义为假。它在此程序中用于查看用户是否输入了有效的选择。

函数 `enter()` 提示用户输入信息, 并将它写入另一个空的结构中。如果数组已满, 将在屏幕上显示信息 "list full"。

```

/* Enter names into list. */
void enter(void)
{
    register int i;

    /* look for an unused structure in the array */
}

```

```

    for(i=0; i<SIZE; i++)
        if(!*addr_info[i].name) break;

    /* i will equal SIZE only when no free structures
       are found.
    */
    if(i==SIZE) {
        printf("list full\n");
        return;
    }

    /* input the info */
    printf("name: ");
    gets(addr_info[i].name);

    printf("street: ");
    gets(addr_info[i].street);

    printf("city: ");
    gets(addr_info[i].city);

    printf("state: ");
    gets(addr_info[i].state);

    printf("zip: ");
    gets(addr_info[i].zip);
}

```

程序 save( )和 load( )如下所示, 用于程序保存和调入通讯录数据库。注意由于 fread( )和 fwrite( )函数的使用使得程序代码很短。

```

/* Save the list. */
void save(void)
{
    FILE *fp;
    register int i;

    if((fp=fopen("maillist", "wb"))==NULL) {
        printf("cannot open file\n");
        return;
    }

    for(i=0; i<SIZE; i++)
        if(*addr_info[i].name)
            if(fwrite(&addr_info[i], sizeof(struct addr), 1, fp)!=1)
                printf("file write error\n");
    fclose(fp);
}

/* Load the file. */
void load(void)
{
    FILE *fp;
    register int i;

    if((fp=fopen("maillist", "rb"))==NULL) {
        printf("cannot open file\n");
        return;
    }

    init_list();
}

```

```

    for(i=0; i<SIZE; i++)
        if(fread(&addr_info[i], sizeof(struct addr), 1, fp)!=1) {
            if(feof(fp)) {
                fclose(fp);
                return;
            }
            printf("file read error\n");
        }
    }
}

```

两个程序通过检查 `fread()` 或 `fwrite()` 的返回值来确定文件操作成功与否。而且 `load()` 必须通过使用 `feof()` 来检查文件结束，因为 `fread()` 不管到达文件尾还是出错都返回相同的值。

程序中的最后一个函数是 `display()`。它在屏幕上打印全部通讯录信息。

```

/* Display the address list. */
void display(void)
{
    register int t;

    for(t=0; t<SIZE; t++) {
        if(*addr_info[t].name) {
            printf("%s\n", addr_info[t].name);
            printf("%s\n", addr_info[t].street);
            printf("%s\n", addr_info[t].city);
            printf("%s\n", addr_info[t].state);
            printf("%s\n\n", addr_info[t].zip);
        }
    }
}

```

通讯录程序完整地列在这儿。如果你对结构的理解有一定的疑问，请将此程序输入到计算机中并研究它的执行过程，做一些改动并观察变化。进一步，你应该试着加入函数来查表，从表中删除一个地址，并在屏幕上打印通讯录。

```

/* A simple mailing list that uses an array
   of structures. */

#include <conio.h>
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>

#define SIZE 100.

struct addr {
    char name[40];

    char street[40];
    char city[30];
    char state[3];
    char zip[10];
} addr_info[SIZE];

void enter(void), init_list(void), display(void);
void save(void), load(void);
int menu(void);

```



```

main(void)
{
    char choice;

    init_list();

    for(;;) {
        choice = menu();
        switch(choice) {
            case 'e': enter();
                        break;
            case 'd': display();
                        break;
            case 's': save();
                        break;
            case 'l': load();
                        break;
            case 'q': return 0;
        }
    }

    /* Initialize the addr_info array. */
    void init_list(void)
    {
        register int t;

        for(t=0; t<SIZE; t++) *addr_info[t].name = '\0';
        /* a zero length name signifies empty */
    }

    /* Get a menu selection. */
    menu(void)
    {
        char ch;

        do {
            printf("(E)nter\n");
            printf("(D)isplay\n");
            printf("(L)oad\n");
            printf("(S)ave\n");
            printf("(Q)uit\n\n");
            printf("choose one: ");
            ch = getche();
            printf("\n");
        } while(!strchr("edlsq", tolower(ch)));
        return tolower(ch);
    }

    /* Input names into the list */
    void enter(void)
    {
        register int i;

        /* find the first free structure */
        for(i=0; i<SIZE; i++)
            if(!*addr_info[i].name) break;

        /* i will equal SIZE if the list is full */
        if(i==SIZE) {
            printf("list full\n");
            return;
        }
    }
}

```

```

/* enter the information */
printf("name: ");
gets(addr_info[i].name);

printf("street: ");
gets(addr_info[i].street);

printf("city: ");
gets(addr_info[i].city);

printf("state: ");
gets(addr_info[i].state);

printf("zip: ");
gets(addr_info[i].zip);
}

/* Display the list. */
void display(void)
{
    register int t;

    for(t=0; t<SIZE; t++) {
        if(*addr_info[t].name) {
            printf("%s\n", addr_info[t].name);
            printf("%s\n", addr_info[t].street);
            printf("%s\n", addr_info[t].city);
            printf("%s\n", addr_info[t].state);
            printf("%s\n\n", addr_info[t].zip);
        }
    }
}

/* Save the list. */
void save(void)
{
    FILE *fp;
    register int i;

    if((fp=fopen("maillist", "wb"))==NULL) {
        printf("cannot open file\n");
        return;
    }

    for(i=0; i<SIZE; i++)
        if(*addr_info[i].name)
            if(fwrite(&addr_info[i], sizeof(struct addr), 1, fp)!=1)
                printf("file write error\n");
    fclose(fp);
}

/* Load the file. */
void load(void)
{
    FILE *fp;
    register int i;

    if((fp=fopen("maillist", "rb"))==NULL) {
        printf("cannot open file\n");
        return;
    }
}

```

```

init_list();
for(i=0; i<SIZE; i++)
    if(fread(&addr_info[i], sizeof(struct addr), 1, fp)!=1) {
        if(feof(fp)) {
            fclose(fp);
            return;
        }
        printf("file read error\n");
    }
}

```

### 12.3 结构赋值

如果两个结构变量具有相同类型，则可以将一个赋给另一个。这样，左边的结构中所有元素将接收右边结构中相应元素的值。例如，这个程序将 `one` 的值赋给 `two` 并且显示其结果：

```

10 98.6

#include <stdio.h>

main(void)
{
    struct sample {
        int i;
        double d;
    } one, two;

    one.i = 10;
    one.d = 98.6;

    two = one; /* assign one struct to another */

    printf("%d %lf", two.i, two.d);
    return 0;
}

```

注意，对于不同类型的结构不允许相互赋值，那使它们元素相容。

### 12.4 将结构传递给函数

到目前为止，例子程序中的所有结构和数组要么是全局的，要么在使用它们的函数中已定义过。本节将重点讨论把结构及其元素传递给函数。

#### 12.4.1 将结构元素传递给函数

当将一个结构变量的元素传递给一个函数时，实际上是将这个元素的值传递给这个函数。因此，传递的只是一个简单的变量。(当然，除非这个元素是复合型的，如一个字符数组。)例如，考虑下面这个结构：

```

struct {
    char x;
    int y;
    float z;
    char s[10];
} sample;

```

下面是一个将每个元素传递给一个函数的例子：

```

func(sample.x); /* passes character value of x */
func2(sample.y); /* passes integer value of y */
func3(sample.z); /* passes float value of z */
func4(sample.s); /* passes address of string s */
func(sample.s[2]); /* passes character value of s[2] */

```

但是, 如果想将单个结构元素的地址传递给函数的话, 必须在结构名前用运算符&。  
例如, 为了将结构变量 sample 元素的地址传递给函数, 你应写成:

```

func(&sample.x); /* passes address of character x */
func2(&sample.y); /* passes address of integer y */
func3(&sample.z); /* passes address of float z */
func4(sample.s); /* passes address of string s */
func(&sample.s[2]); /* passes address of character s[2] */

```

请注意, 运算符&放在结构名前, 而不是单个元素名前。还应记住, 字符串元素 s 已经被指定为地址, 在这种情况下不需要用&。

#### 12.4.2 将整个结构传递给函数

当一个结构作为参数传递给函数时, 按标准的赋值调用方法可将整个结构传递给函数。这就是说, 函数内部的结构内容的改变不会影响到作为实参被调用的那个结构。

专用一个结构作为一个形式参数时, 最重要的一件事是: 形参的类型必须与实参类型一致。例如, 这个程序说明了参数 arg 和 parm 为同一结构类型:

```

#include <stdio.h>

/* define a structure type */
struct sample {
    int a, b;
    char ch;
};

void f1(struct sample parm);

main(void)
{
    struct sample arg; /* declare arg */
    arg.a = 1000;

    f1(arg);
    return 0;
}

void f1(struct sample parm)
{
    printf("Zd", parm.a);
}

```

可以看到此程序在屏幕上显示 1000。

正如这个例程所示，最好是定义一个全局结构，然后用结构名来说明所需的结构变量和参数。这还保证了形参和实参的类型是一致的。而且，别人阅读你的程序时也很容易得知 `parm` 和 `arg` 是同一类型的。

## 12.5 结构指针

C 语言允许与其它变量同样的方式使用结构指针。但是，结构指针还有它特殊的性质，这一点请特别注意。

### 12.5.1 结构指针说明

结构指针由在一个结构变量名前加上操作符“\*”来说明。例如，前面已定义过的结构 `addr`，下面这条语句是说明 `addr_pointer` 为一个这种数据类型的指针：

```
struct addr *addr_pointer;
```

### 12.5.2 使用结构指针

结构指针有各种用法。一个是通过函数调用的方式。另一个是建立链表和使用 Turbo C++ 分配系统来建立其它动态数据结构。这一章只讨论第一种用法。

除了最简单的结构外，其它所有将结构传递给函数的方法最主要的一个缺点是 将所有的结构元素压栈(退栈)需要额外的开销。在只有几个元素的简单的结构中，这种额外的开销并不是很重要，但如果所用的元素很多或者是数组的话，在运行时就会有有很大的影响。解决这个问题的方法是，仅将一个指针传递给函数。

当一个结构指针传递给函数时，实际上只是将结构地址压栈(或退栈)。这就意味着，它能很快地调用函数。另外，由于函数调用的是实际的结构，而不是复制，它能够在调用时修改结构中实际元素的内容。

为了寻找结构变量的地址，在结构名前加上运算符 `&`。例如，下面这段程序：

```
struct bal {  
    float balance;  
    char name[80];  
} person;  
  
struct bal *p; /* declare a structure pointer */
```

然后

```
p = &person;
```

将结构 `person` 的地址赋给指针 `p`。为了访问元素 `balance`，可以写成：

```
(*p).balance
```

结构元素很少用 \* 操作符，正如上面例子所示。因为借助指针访问一个结构元素十分普遍，C 语言定义了一个特殊的操作符来完成这个任务。C 程序员都把它叫做箭头操作符。这是由一个减号跟着一个大于号形成的。当用结构变量指针访问结构元素时，箭头操作符用于代替点操作符。例如，前面的语句通常写成如下形式：

```
p->person
```

为了理解一个结构指针的用法，下面举一个简单的例子，这个程序用软件延时器

显示时、分、秒。

```
/* Display a software timer. */

#include <stdio.h>
#include <conio.h>

struct time_struct {
    int hours;
    int minutes;
    int seconds;
};

void update(struct time_struct *t);
void display(struct time_struct *t);
void delay(void);

main()
{
    struct time_struct time;

    time.hours=0;
    time.minutes=0;
    time.seconds=0;

    for( ; kbhit(); ) {
        update(&time);
        display(&time);
    }
    return 0;
}

void update(struct time_struct *t)
{
    t->seconds++;
    if(t->seconds==60) {
        t->seconds = 0;
        t->minutes++;
    }
    if(t->minutes==60) {
        t->minutes = 0;
        t->hours++;
    }
    if(t->hours==24) t->hours=0;
    delay();
}

void display(struct time_struct *t)
{
    printf("%d:", t->hours);
    printf("%d:", t->minutes);
    printf("%d\n", t->seconds);
}

void delay(void)
{
    long int t;
    for(t=1; t<128000; ++t) ;
}
```

程序中时间的校准可以通过调整函数 `delay()` 中循环次数来实现。

全局结构 `time_struct` 在程序的开头处被定义，但并未说明变量。在 `main()` 中，`time` 结构被说明并被初始化为 `00:00:00`。这意味着 `time` 只在 `main()` 函数中有定义。

另两个函数——`update()`用来修改时间，`display()`显示时间——`time`的地址被传递给它们。在两个函数中，函数中的实参都被说明为 `time_struct` 类型结构指针，因此编译程序将知道如何访问结构元素。

实际上每个结构元素都可以用一个指针来访问。例如，如果想将小时的值重新设为0，当时间为 24:00:00 时，可以写：

```
if (t->hours == 24) t->hours = 0;
```

这条语句告诉编译程序取 `t` 的地址(它是 `main()` 中变量 `time` 的地址)，并且在调用 `hours` 时，将零值赋给这个元素。

请注意，当在结构自身上操作时用句点操作符来访问结构元素。当用结构指针时，必须使用箭头操作符。并且记住必须用 `&` 操作符将地址传递给结构。

关于时间，我们来研究一些 C 语言的时间和日期函数。系统时间和日期函数的原型在文件 `TIME.H` 中，而且此类文件还包括两个定义类型。类型 `time_t` 能够用一个长整数来代表系统时间和日期，这也叫做日历时间(calendar time)。结构类型 `tm` 的元素代表时间和时期。`tm` 结构定义如下：

```
struct tm {
    int tm_sec; /* seconds, 0-59 */
    int tm_min; /* minutes, 0-59 */
    int tm_hour; /* hours, 0-23 */

    int tm_mday; /* day of the month, 1-31 */
    int tm_mon; /* months since Jan, 0-11 */
    int tm_year; /* years from 1900 */
    int tm_wday; /* days since Sunday, 0-6 */
    int tm_yday; /* days since Jan 1, 0-365 */
    int tm_isdst; /* Daylight Savings Time indicator */
};
```

如果为夏令时间，`tm_isdst` 的值将为正数，否则为 0，如果无可用信息，则为负数，这种时间和日期的格式叫分离时间(broken\_down time)。

C 语言中时间和日期的最基本函数是 `time()`，其原型为：

```
time_t time(time_t *time)
```

函数 `time()` 返回距 1970 年 1 月 1 日的秒数。它可以被空指针调用，也可以被 `time_t` 型变量指针调用。如果使用后者，形参将被赋值为日历时间。

为了变换日历时间为分离时间，使用函数 `localtime()`，该函数原型如下：

```
struct tm *localtime(time_t *time)
```

函数 `localtime()` 以 `tm` 结构形式返回指向分离时间的指针。时间由当地时间表示。时间值是通过调用函数 `time()` 获得的。

`localtime()` 用到的结构是静态分配，它包含分离时间，而且函数每次被调用时都被重写。如果想保存结构内容，必须将它拷贝其他地方。

尽管用户程序可以使用时间和日期的分离时间形式，产生时间和日期的最简单的方法是使用 `asctime()`，其函数原型为：

```
char *asctime(struct tm *ptr);
```

函数 `asctime()` 返回一字符串指针，将结构中保存信息(由 `ptr` 所指的)变换到如下形式：

```
day month date hours:minutes:seconds year\n\0
```

传递给 `asctime()` 的结构指针是由 `localtime()` 获得的。

像 `localtime()` 一样,用于 `asctime()` 的缓冲区是静态分配的字符数组,而且每次函数调用时都要重写。因此,如果要保存字符串的内容,必须将它拷贝到别的地方。

下面的程序用时间函数将系统的时间和日期显示到屏幕上。

```
/* This program displays the current system time. */

#include <stdio.h>
#include <time.h>

main(void)
{
    struct tm *ptr;
    time_t lt;

    lt = time("\0");

    ptr = localtime(&lt);
    printf(asctime(ptr));

    return 0;
}
```

Turbo C++ 还有一些别的时间和日期函数。这些函数在《库函数参考》中有详细说明。

## 12.6 结构内部的数组和结构

结构元素可以是任何有效的 C 语言数据类型,包括数组和结构。你已经看到了数组元素的例子:在 `addr_info` 中用到的字符数组。

根据前面的例子,我们可以将它们推广到一个结构元素是一个数组的情况。例如,考虑下面的结构:

```
struct x {
    int a[10][10]; /* 10 x 10 array of ints */
    float b;
} y;
```

为了访问结构 `y` 中 `a` 的 (3, 7) 这个整型量,可以这样写:

`y.a[3][7]`

当一个结构是另一个结构中的元素时,它被称为嵌套式结构。例如,结构变量 `address` 是嵌套在结构 `emp` 内的一个嵌套结构:

```
struct emp {
    struct addr address;
    float wage;
} worker;
```

这里的 `addr` 在前面已经定义过,而结构 `emp` 被定义为它有两个元素。第一个元素是 `addr` 型结构,它包含有一个雇员的地址。第二个元素是 `wage`,它存储着雇员的工资。下面这段程序代码表示将邮政编码 98765 赋给 `worker` 的 `address` 域中的 `zip`:

`worker.address.zip=98765;`



正如你见到的那样，每个结构元素名从最外层到最内层从左至右逐个被列出。

## 12.7 位域

与多数计算机语言不同，C语言用内部法(build-in)来访问一个字节或一个字中的一个或多个位。由于以下原因表明这是很有用的：首先，如果存储单元被限制，你可以在一个字节中存储若干布尔量(逻辑真和逻辑假)；其次，某些外设接口是按一个字节的位代码来传递信息的；第三，某些编译程序需要按位访问一个字节的各个位。

C语言中访问位的方法是以结构为基础的，叫做位域(下一章你将学习到第二种访问位的方法)。一个位域实际上是结构元素的一个特殊形式，它需要定义位的长度。位域定义的一般形式是：

```
struct struc-type-name {  
    type name1 : length;  
    type name2 : length;  
  
    type nameN : length;  
}
```

一个位域必须被说明为 `int`、`unsigned` 或 `signed` 中的任一种。长度为 1 的位域被认为是 `unsigned` 类型，因为单个位不可能具有符号。

`int biosequip(void)`

函数 `biosequip()` 的原型在文件 `BIOS.H` 中。

函数 `biosequip()` 用一个 16 位值返回计算机中配置的设备列表，如下所示：

<u>位</u>	<u>设备</u>
0	必须从软驱中启动
1	8087数学协处理器安装
2, 3	母板RAM大小 0 0 : 16K 0 1 : 32K 1 0 : 48K 1 1 : 64K
4, 5	初始屏幕模式 0 0 : 未用 0 1 : 40X25 BW, 彩色适配器 1 0 : 80X25 BW, 彩色适配器 1 1 : 80X25, 单色适配器
6, 7	软盘驱动器数目 0 0 : 一个 0 1 : 两个

	1 0 : 三个
	1 1 : 四个
8	DMA配置
9, 10, 11	串行口数目
	0 0 0 : 零个
	0 0 1 : 一个
	0 1 0 : 两个
	0 1 1 : 三个
	1 0 0 : 四个
	1 0 1 : 五个
	1 1 0 : 六个
	1 1 1 : 七个
12	游戏卡配置
13	串行打印机配置
14, 15	打印机数目
	0 0 : 零个
	0 1 : 一个
	1 0 : 两个
	1 1 : 三个

用下面的结构可以代表位域:

```
struct equip {
    unsigned floppy_boot: 1;
    unsigned has8087: 1;
    unsigned mother_ram: 2;
    unsigned video_mode: 2;
    unsigned floppies: 2;
    unsigned dma: 1;
    unsigned ports: 3;
    unsigned game_adpter: 1;
    unsigned unused: 1;
    unsigned num_printers:2;
} eq;
```

下面的程序用到这个位域来显示软盘驱动器的个数和串行口的个数。

```
#include <stdio.h>
#include <bios.h>

main(void)
{
    struct equip {
        unsigned floppy_boot: 1;
        unsigned has8087: 1;
        unsigned mother_ram: 2;
        unsigned video_mode: 2;
        unsigned floppies: 2;
        unsigned dma: 1;
        unsigned ports: 3;
        unsigned game_adpter: 1;
        unsigned unused: 1;
        unsigned num_printers:2;
    } eq;
```

```

int *i;

i = (int *) &eq;

*i = biosequip();

printf("%d floppies\n", eq.floppies+1);

printf("%d ports\n", eq.ports+1);

return 0;
}

```

整型指针 `i` 被赋给 `eq` 的地址，而且 `biosequip()` 的返回值通过这个指针赋给 `eq`。这是必须的，因为 C 语言不允许一个整数进入结构。然而，在下一节，将学习一个解决这个问题更好的方法。

正如你从这个例子中看到的那样，每个位域都用句点操作符来访问。然而，如果结构使用指针，则必须用箭头(`->`)操作符。

不必给每个位域命名。这使得容易访问想要的位，忽略未用的那些位。`unused` 域可以无定义，如下所示：

```

struct equip {
    unsigned floppy_boot: 1;
    unsigned has8087: 1;
    unsigned mother_ram: 2;
    unsigned video_mode: 2;
    unsigned floppies: 2;
    unsigned dma: 1;
    unsigned ports: 3;
    unsigned game_adpter: 1;
    unsigned : 1;
    unsigned num_printers: 2;
} eq;

```

位域变量有某些限制。你不可取一个位域变量的地址。位域变量不允许是数组，也不允许超越整型量边界。在不同类型的 CPU 中，域是从右到左运行还是从左到右运行是不同的。任何使用位域的代码都可能与机器的特性有关。

将位域元素和一般的结构元素混合使用是合法的。例如：

```

struct emp {
    struct addr address;
    float pay;
    unsigned lay_off:1; /* lay off or active */
    unsigned hourly:1; /* hourly pay or wage */
    unsigned deductions:3; /* IRS deductions */
};

```

这个结构定义了一个有关雇员的记录，它仅用一个字节来存储三个信息：雇员的状态、雇员的工资是否发了以及扣除数目。若不用位域这些信息需要三个字节。

下一节的程序用位域来显示二进制形式的 ASCII 字符。

## 12.8 联合(union)

在 C 语言中, 联合表示几个变量公用一个内存位置。这些变量可以是不同的类型。联合(union)的定义与结构(structure)的定义十分相似, 例如:

```
union u_type {  
    int i;  
    char ch;  
};
```

与结构定义相同, 这个定义并不说明任何变量。你可以通过在定义的结尾处给出它的变量名来说明一个变量或单独使用说明语句。为了说明 union 变量 `cnvt`(类型为 `u_type`, 刚定义过的)是一个联合, 可以写成:

```
union u_type cnvt;
```

在 `cnvt` 中, 整型量 `i` 和字符 `ch` 公用同一内存位置(当然, `i` 占用两个字节, 而 `ch` 仅占一个字节)。图 12-2 表示 `i` 和 `ch` 公用同一地址的情况。

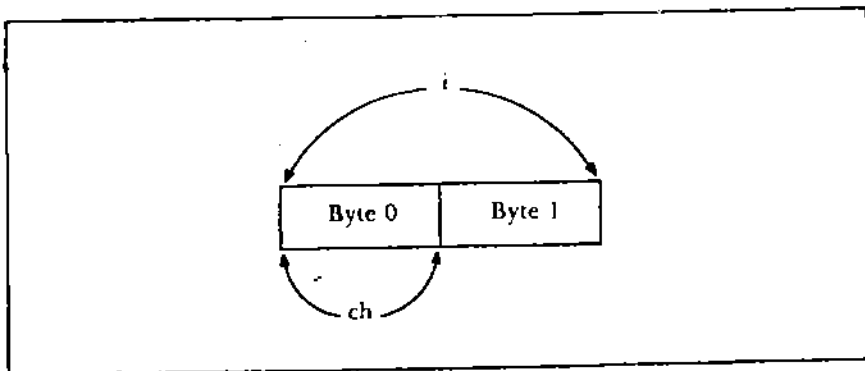


图 12-2 `i` 和 `ch` 公用一个联合 `cnvt`

当一个联合(union)被说明时, 编译程序自动地产生一个变量, 其长度为 union 中最大的变量的长度。

为了访问 union 的元素, 可以使用与结构相同的方法, 即句点运算符和箭头运算符。如果直接对 union 进行操作, 使用句点运算符; 若 union 变量是通过一个指针来访问的, 那么要用箭头运算符。例如, 为了将整型量 10 赋给 `cnvt` 中元素 `i`, 可以这样写

```
cnvt.i=10;
```

联合常用于需要对类型作转换的场合, 因为它们可让你用不只一种方法查看内存。例如, 标准函数 `putw()` 表示写一个二进制的整型量给盘文件。虽然有各种方法可以实现这个目的, 这里只介绍如何使用 union 来完成这项任务。首先, 确定一个 union, 它包括一个整型量和一个二字节的字符数组。

```
union pw{  
    int i;  
    char ch[2];  
};
```

现在可用联合来编写 `putw()` 函数:

```

void putw(union pw word, FILE *fp) /* putw with union */
{
    putc(word->ch[0], fp); /* write first half */
    putc(word->ch[1], fp); /* write second half */
}

```

虽然调用时用整型量，但 `putw()` 仍可用标准函数 `putc()` 写一个整型数给盘文件。

下面的程序综合位域和联合，当按下任一健时，显示相应的二进制 ASCII 码。union 允许 `getche()` 将键值赋给字符变量，同时位域用来显示单个位。研究这个程序并保证完全理解它的执行。(此程序的执行只是为了说明联合可用两种根本不同的方法来查找内存相同空间。有许多有效的方法来书写函数 `decode()` 以达到同样的结果。)

```

/* Display the ASCII code in binary for characters. */

#include <stdio.h>
#include <conio.h>

/* a bit-field that will be decoded */
struct byte {
    int a : 1;
    int b : 1;
    int c : 1;
    int d : 1;
    int e : 1;
    int f : 1;
    int g : 1;
    int h : 1;
};

union bits {
    char ch;
    struct byte bit;
} ascii;

void decode(union bits b);

main(void)
{
    do {
        ascii.ch = getche();
        decode(ascii);
    } while(ascii.ch != 'q'); /* quit if q typed */

    return 0;
}

/* Display the bit pattern for each character. */
void decode(union bits b)
{
    if(b.bit.h) printf("1 ");
    else printf("0 ");
    if(b.bit.g) printf("1 ");
    else printf("0 ");
    if(b.bit.f) printf("1 ");
    else printf("0 ");
    if(b.bit.e) printf("1 ");
    else printf("0 ");
    if(b.bit.d) printf("1 ");
    else printf("0 ");
}

```

```

    if(b.bit.c) printf("1 ");
    else printf("0 ");
    if(b.bit.b) printf("1 ");
    else printf("0 ");
    if(b.bit.a) printf("1 ");
    else printf("0 ");
    printf("\n");
}

```

图 12-3 显示一个例子的运行。

```

A:\>ascii
a: 0 1 1 0 0 0 0 1
b: 0 1 1 0 0 0 1 0
c: 0 1 1 0 0 0 1 1
d: 0 1 1 0 0 1 0 0
e: 0 1 1 0 0 1 0 1
f: 0 1 1 0 0 1 1 0
g: 0 1 1 0 0 1 1 1
h: 0 1 1 0 1 0 0 0
i: 0 1 1 0 1 0 0 1
j: 0 1 1 0 1 0 1 0
k: 0 1 1 0 1 0 1 1
l: 0 1 1 0 1 1 0 0
m: 0 1 1 0 1 1 0 1
n: 0 1 1 0 1 1 1 0
o: 0 1 1 0 1 1 1 1
p: 0 1 1 1 0 0 0 0
q: 0 0 1 1 1 1 0 1
r: 0 0 1 1 1 1 0 1
s: 0 0 1 1 1 1 0 1
t: 0 0 1 1 1 0 0 0
u: 0 0 1 1 1 0 0 1
v: 0 0 1 1 1 0 0 1
w: 0 0 1 1 1 0 0 1
x: 0 0 1 1 1 0 0 1
y: 0 0 1 1 1 0 0 1
z: 0 0 1 1 1 0 0 1
A:\>

```

图 12-3 ASCII 程序的试运行

为了结束这一节，让我们来看一下 union 如何提供一个将整型量装入位域的方法。回忆前节的内容，C 语言不允许一个整数直接赋给一个位域结构。然而，这个程序创立了一个联合，它包含一个整型量和一个位域。当 biosequip( ) 返回设备表(用一个整数表示)，则被赋给一个整数。然而，当输出结果时程序可自由使用位域。

```

/* Display the number of floppies and ports. */
#include <bios.h>
#include <stdio.h>

main(void)
{
    struct equip {
        unsigned floppy_boot: 1;
        unsigned has8087: 1;
        unsigned mother_ram: 2;
        unsigned video_mode: 2;
        unsigned floppies: 2;
        unsigned dma: 1;
        unsigned ports: 1;
        unsigned game_adapter: 1;
    };
}

```

```

    unsigned unused:    1;
    unsigned num_printers:2;
} ;

union {
    struct equip eq;
    unsigned i;
} eq_union;

eq_union.i = biosequip();

printf("%d floppies\n", eq_union.eq.floppies+1);

printf("%d ports\n", eq_union.eq.ports+1);

return 0;
}

```

## 12.9 枚举

枚举是一个被命名为整型常数的集合。这些常数指定了所有它们的类型已被定义的各种合法值。枚举在我们日常生活中十分常见。例如，美国使用的硬币的枚举为。

penny, nickel, dime, quarter, half\_dollar, dollar

枚举的定义与结构定义十分相似，用关键字 `enum` 来表示一个枚举。它的一般形式是：

`enum 枚举名 {枚举表} 变量表;`

枚举表是用逗号分隔的名表，代表枚举类型变量可能具有的值。枚举名和枚举变量表都是可选项。枚举名用来说明这种类型的变量。下面这段程序定义了一个称之为 `coin` 的枚举，并说明 `money` 属于这种类型；

```

enum coin { penny, nickel, dime, quarter,
            half_dollar, dollar};

enum coin money;

```

给出这些定义和说明后，下面这条语句完全有效：

```

money = dime;

if (money == quarter) printf("is a quarter\n");

```

正确理解枚举的关键是：一个枚举实际上是将每个符号用它们所对应的整数来代替，而且可以在任何一个整型量表达式中使用这些枚举值。除非进行了初始化，否则第一个枚举符号的值为 0，第二个为 1，依次类推。因此，

```
printf("%d %d", penny, dime);
```

将在屏幕上显示 0 2。

也可以用初始化方法指定一个或几个符号的值。这可以通过符号后加一个等号和一个整型量来实现。当使用初始化方式时，初始化后的各个符号所赋的值必须大于原先的初始值。例如，下面这条语句将 100 这个值赋给 `quarter`。

```

enum coin {penny, nickel, dime, quarter=100,
            half_dollar, dollar};

```

现在，这些符号的值如下：

penny	0
nickel	1
dime	2
quarter	100
half_dollar	101
dollar	102

通常认为枚举符号可以直接输入和输出，但这是不正确的。例如，下面这段程序与所要的结果完全不一致。

```
/* this will not work */
money = dollar;

printf("%s", money);
```

请注意，符号 `dollar` 仅仅是一个整型量的名，并不是字符串。由于同样原因，下面这段语句不可能达到预期目的。

```
/* this code is wrong */
gets(s);
strcpy(money, s);
```

也就是说，一个包括一符号名的字符串不能自动转换为那一符号。

实际上，产生输入和输出枚举符号的代码是毫无价值的(除非你想调整它们整型量的数值)。例如，下面这段程序需要显示 `money` 中硬币的类型：

```
switch(money) {
    case penny: printf("penny");
                break;
    case nickel: printf("nickel");
                break;
    case dime: printf("dime");
                break;
    case quarter: printf("quarter");
                break;
    case half_dollar: printf("half_dollar");
                break;
    case dollar: printf("dollar");
}
}
```

有时需要说明字符串数组，并将枚举值作为下标传送给它们相应的字符串。例如，下面这段程序同样给出正确的字符串。

```
char name()[20]= {
    "penny",
    "nickel",
    "dime",
    "quarter",
    "half_dollar",
    "dollar"
};

.
.
.
printf("%s", name[money]);
```



当然，这项工作只有在没有任何符号被初始化的情况下才能使用，因为字符串数组的下标是从零开始的。例如，下面的程序打印硬币名称：

```
#include <stdio.h>

enum coin { penny, nickel, dime, quarter,
            half_dollar, dollar};

enum coin money;

char name[][20]={
    "penny",
    "nickel",
    "dime",
    "quarter",
    "half_dollar",
    "dollar"
};

main(void)
{
    enum coin money;

    for(money=penny; money<=dollar; money++)
        printf("%s ", name[money]);

    return 0;
}
```

由于枚举值必须由人工转换成 I/O 控制中人所能读的字符串值，所以它们在不需要作这种转换的程序中十分有用。例如，经常用一个枚举来定义一个编译程序的符号表。

#### 12.10 使用 sizeof 来确保可移植性

尽管在前面章节简要地接触到了 sizeof，现在我们仍需进一步研究。你已知道，结构、联合和枚举都可以用来建立大小可变的变量，并且这些变量的实际大小可能随着不同的机器类型而变化。单目操作符 sizeof 用来计算任何变量或类型的大小，而且有助于消除程序中与机器有关的代码。

下面列出了操作符 sizeof 的两种形式：

**sizeof**    变量名

**sizeof**    (类型名)

当计算变量大小时，无须将变量名括在括号里(尽管这样做也不错。)然而，在计算类型的大小时，必须用括号括起来。例如，下面程序段中的两行都是显示一个 int 的大小：

```
int i;

printf("%d", sizeof i);
printf("%d", sizeof (int));
```

**Turbo C++** 规定了下列数据类型的大小：

类型	字节大小
char	1
int	2
longint	4
float	4
double	8
long double	10

因此，下面这段程序在屏幕上显示出 1, 2, 4 这三个值。

```
char ch;
int i;
double f;

printf("%d", sizeof ch);

printf("%d", sizeof i);

printf("%d", sizeof f);
```

操作符 `sizeof` 是一个编译阶段操作符；所有在计算变量大小时所需的信息在编译时知道。例如，考虑如下程序：

```
union x {
    char ch;
    int i;
    float f;
} tom;
```

`sizeof tom` 是 4，因为 `union` 的最大元素是 `float` 型。在运行阶段，它不管在 `union tom` 中包含的是什么，所考虑的仅仅是它可能包含的最大变量大小，因为 `union` 必须和最大元素一样大。

## 12.11 typedef

C 允许定义用关键字 `typedef` 所命名的新的数据类型。实际上，你并没有建立一个新的数据类型，而仅仅是对于存在的类型定义了一个新的名字。这个过程有助于使得与机器有关的程序更具有通用性，它仅仅需要改变一下 `typedef` 语句。由于允许使用标准的数据类型作为说明名，这有助于建立用户的源程序。`typedef` 语句的一般形式为。

`typedef` 类型 定义名；

这里，类型是任何一种许可的数据类型，所定义的新的定义名是现有的类型名的补充，而不是取代。

例如，可以用下面语句为 `float` 建立一个新的定义名

```
typedef float balance;
```

这条语句告诉编译程序将 `balance` 看作为实型量的另一定义名。此外，还可以用 `balance` 来建立一个实型量：

```
balance over_due;
```

这里，`over_due` 是一个 `balance` 类型的浮点变量，而 `balance` 仅仅是 `float` 的别名而已，也可以用 `typedef` 产生更为复杂形式的定义名。例如：

```

typedef struct client_type {
    float due;
    int over_due;
    char name[40];
} client;

client clist[NUM_CLIENTS]; /* define array of
                           structures of type client */

```

在这个例子中，`client` 不是 `client_type` 型变量而是结构 `client_type` 的别名。

使用 `typedef` 有助于使用户程序更加容易读和更容易移植。但是要记住，你不可能建立任何新的数据类型。

## 第十三章 高级运算符

前面你已经学到了许多一般的 C 运算符。和其它绝大多数语言不同, C 语言包括一些特殊的运算符使得它的功能更强, 也更灵活——特别是系统级的程序。在本章中你将学习到这些运算符。

### 13.1 按位运算符

C 语言和其他绝大多数语言不同的是, 它完全支持按位运算。既然 C 语言是设计成用来代替汇编语言完成大部分编程工作的, 那它就应该能支持汇编语言所做的全部(或至少是大部分)运算。按位运算是按字节或字中的实际位进行检测、设置和移位。这些字节或字必须对应于 C 语言标准的 `char`、`int` 和 `long` 数据类型。按位运算符不能用于 `float`、`double`、`long`、`void` 或其它更复杂的数据类型。表 13-1 列出了这些运算符。

表 13-1 按位运算符

操作符	作用
<code>&amp;</code>	位逻辑与
<code> </code>	位逻辑或
<code>^</code>	位逻辑异或
<code>~</code>	位逻辑反
<code>&gt;&gt;</code>	右移
<code>&lt;&lt;</code>	左移

位逻辑运算符 AND, OR 和 NOT 的真值表与逻辑运算符的真值表一样, 只不过前者的运算是逐位进行的。按位异或运算符 ^ 的真值表如下:

p	q	p	q
0	0	0	0
1	0	1	1
1	1	0	0
0	1	1	1

如上表所示, 按位异或运算(XOR)的结果仅仅在两个操作数只有一个为真时才为真, 否则为假。

位逻辑运算符常用在设备驱动程序中, 例如调制解调程序、磁盘文件程序以及打印程序。因为按位运算符能用于屏蔽某些位, 例如奇偶校验位(该位是用来确认字节其他位未变化。它通常是字节中的高位)。

就其最常见的用途来说, 你可认为位运算符 AND 用于将某些位设为 0。也就是说, 两个操作数中只要有一个为 0, 则会使结果的对应位为 0。例如, 下列函数通过库函数 `bioscom`

从调制解调输入口读入一字符，并将奇偶校验位设为 0。函数 bioscom() 用于访问 IBM PC 或兼容机上的异步串行口。

```
char get_char_from_modem(void)
{
    char ch;

    ch = bioscom(2, 0, 0); /* get a character from
                           COM1 */
    return ch & 127;
}
```

奇偶校验位是用第 8 位来表示的。若将该字符和一个第 1 到第 7 位均为 1 而第 8 位为 0 的数作 AND 运算，则可将该奇偶位设为 0。在上面的表达式中 `ch&127` 表示将 `ch` 和数字 127 的每一位作 AND 运算。最后的结果是将 `ch` 的第 8 位设置为 0。在下面的例子中，假定 `ch` 已接收了字符 'A' 并已作了奇偶校验位设置。

奇偶校验位

11000001	ch 中的 "A" 字符及其奇偶校验位
01111111	二进制的 127
& ————	位逻辑与运算
01000001	奇偶校验位为 0 的 "A" 字符

位逻辑或运算 OR 与 AND 恰恰相反，可以用于将某些位设置为 1。在两个操作数中只要有一个操作数其某一位为 1，则会使 OR 运算结果的对应位为 1。例如，128|3 为：

10000000	二进制 128
00000011	二进制 3
————	位逻辑或运算
10000011	结果

XOR 是位逻辑异或运算，它将两个操作数逐位比较，若不同则结果的对应位为 1，否则为 0，例如 127^120 为：

01111111	二进制 127
01111000	二进制 120
^ ————	位逻辑异或
00000111	结果

总的来说，位逻辑运算符 AND、OR 和 XOR 都是直接对变量的每一位单独进行操作。由于这个原因，通常这些按位运算符不能象关系和逻辑运算符那样用于条件语句中。例如，`X=7`，则 `X&&8` 的值为真（即 1），而 `X&8` 的值为假（即 0）。

请记住一点，关系运算符和逻辑运算符总是得出非 0 即 1 的结果。而与之类似的位逻辑运算符则会得出取决于其特定运算的任意数值。换句话说，位逻辑运算符的结果可以是 0 和 1 以外的值，而逻辑运算的结果只能是 0 和 1。

AND 运算符在检查某一位的值时也是非常有用的。例如，下面这条语句检查 `status` 的第 4 位是否为 1。

```
if(status&8) printf("bit 4 is on");
```

用 8 来运算是因为它的二进制形式是 00001000。也就是说，数字 8 翻译成二进制数时只有第 4 位为 1。因此，当 `status` 的第 4 位也为 1 时条件语句为真。这种处理在下面的

disp\_binary()函数中很有用。它显示其参数各位的二进制形式。你将在本章稍后用到这个函数,并且可以看到其它按位运算符的作用。

```
/* display the bits within a byte */
void disp_binary(int i)
{
    register int t;

    for(t=128; t>0; t = t/2)
        if(i & t) printf("1 ");
        else printf("0 ");
    printf("\n");
}
```

函数 disp\_binary()通过用位运算符 AND 来确定一个字节中每个位是 1 还是 0 来工作。如果是 1, 则显示数字“1”; 否则显示数字“0”。在前面章节用位来译码和显示二进制值。这里的程序更好些, 因为它速度快而代码少。

移位运算符 << 和 >> 将变量中的每一位向右或向左移动。右移运算语句的一般形式是:

变量名 >> 移的位数

左移语句的形式是

变量名 << 移的位数

经过移位后的一端的位被“挤”掉, 而另一端空出的位以 0 填补。记住, 移位并不是循环的。也就是说, 在一端被“挤”出的位并不转回来补到另一端。另一端的位以 0 填补, 而被“挤”出的位则被舍弃了。

移位运算可以用于对外部设备的输入进行译码, 比如 D/A 转换, 也可用于读状态信息。移位运算符还可以用非常快的速度实现整数乘法和除法运算。左移一位实现快速乘 2, 而右移一位则实现快速除以 2。如图 13-1 所示。

X 为每一语句执行		X 值
<hr/>		
unsigned char x		
x=7;	00000111	7
x<<1;	00001110	14
x<<3;	01110000	112
x<<2	11000000	192
x>>1;	01100000	96
x>>2;	00011000	24

图 13-1 用移位运算符做乘除法

说明: 左移一位相当于乘以 2。但当执行  $x \ll 2$  后,  $x$  的最高位因挤出而丢失, 右移一位相当于除以 2。但随后移位除法并不能将损失的位补回

下面的程序形象地表示出移位操作的结果:

```

/* Example of bitbiffting. */
#include <stdio.h>

void disp_binary(int i);

main(void)
{
    int i=1, t;

    for(t=0; t<8; t++) {
        disp_binary(i);
        i = i << 1;
    }

    printf("\n");

    for(t=0; t<8; t++) {
        i = i >> 1;
        disp_binary(i);
    }
    return 0;
}

/* Display the bits within a byte. */
void disp_binary(int i)
{
    register int t;

    for(t=128; t>0; t=t/2)
        if(i & t) printf("1 ");
        else printf("0 ");
    printf("\n");
}

```

它产生如下输出:

```

0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0
0 0 0 0 1 0 0 0
0 0 0 1 0 0 0 0
0 0 1 0 0 0 0 0
0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0

1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1

```

尽管 C 语言没有循环移位运算符, 建立一函数实现此功能是十分容易的。循环移位与左移运算符很相近, 除了前者需将某一头被挤出的部分加在另一头上外。例如, 1010 循环左移一位是 0101。实现循环移位的一个方法要求使用具有两种不同类型数据的 union(联合)。第一种类型是具有你想循环移位的数据类型的两个元素的数组。第二个是比你移位的数据类型大的类型。在此例中是按位循环左移。当然, 下面的 union 也要用到。

```

union rotate {
    char ch[2];
    unsigned int i;
} rot;

```

这个函数实际执行循环移位，如下所示：

```

/* Rotate a byte. */
void rotate_it(union rotate *rot)
{
    rot->ch[1] = 0; /* clear the high-order byte */

    rot->i = rot->i << 1; /* shift once to the left */

    /* see if a bit has been shifted out of ch[0] */
    if(rot->ch[1]) rot->i = rot->i | 1; /* OR it back in */
}

```

函数首先清除整型数 *i* 的高位字节，这是为了当一位被移进时能够测定出来。对整个数采用左移运算符。ch[0] 移出的位未被丢弃而到了 ch[1] 中。如果一位被“挤”出，则与 ch[0] 的低位字节做 OR 运算。下面程序用到了这个函数：

```

/* Do a rotation. */
#include <stdio.h>

union rotate {
    char ch[2];
    unsigned int i;
} rot;

void disp_binary(int i);
void rotate_it(union rotate *rot);

main(void)
{
    register int t;

    rot.ch[0] = 101;

    for(t=0; t<7; t++) {
        disp_binary(rot.i);
        rotate_it(&rot);
    }
    return 0;
}

/* Rotate a byte. */
void rotate_it(union rotate *rot)
{
    rot->ch[1] = 0; /* clear the high-order byte */

    rot->i = rot->i << 1; /* shift once to the left */

    /* see if a bit has been shifted out of ch[0] */
    if(rot->ch[1]) rot->i = rot->i | 1; /* OR it back in */
}

/* display the bits within a byte */
void disp_binary(int i)
{
    register int t;

```



```

for(t=128; t>0; t=t/2)
    if(1 & t) printf("1.");

    else printf("0 ");
printf("\n");
}

```

源字节循环移位七次后程序产生如下输出：

```

0 1 1 0 0 1 0 1
1 1 0 0 1 0 1 0
1 0 0 1 0 1 0 1
0 0 1 0 1 0 1 1
0 1 0 1 0 1 1 0
1 0 1 0 1 1 0 0
0 1 0 1 1 0 0 1

```

反码运算符~将变量中的每一位都取反。也就是将所有的1都变成0，0变成1。反码运算符的一个有趣的应用是来观察扩展的字符集合。键盘上显示的字符集合只是计算机支持的全部字符集的一部分。下面的程序将你键入的字符所有位用反码运算符取反。这些取反了的位式对应扩展字符集的部分。例如，当键入小写字符“d”，则显示%号，你将很惊奇竟有这么多可用字符。

```

/* A window into the PC's extended character set. */
#include <stdio.h>
#include <conio.h>

main(void)
{
    char ch;

    do {
        ch = getch();
        printf("%c", ~ch);
    } while(ch != 'q');

    return 0;
}

```

按位运算符多用于加密程序。如果想让磁盘文件不可读，需对其进行按位操作。最简单的方法之一是将每一字节的各位求反，如下所示：

```

源字节      00101100
经过一次求反  11010011
经过两次求反  00101100

```

注意对一数字连续求两次反码仍可得源码。因此，第一个反码代表那个字节的编码。而第二个则将其破译为原先的值。

可以使用函数 encode()来对字符进行编码。为了破译先前字符编码，只需再调用一次 encode()。

```

char encode(char ch) /* a simple cipher function */
{
    return(~ch); /* complement it */
}

```

### 13.2 ?运算符

?运算符可以用来代替如下形式的 if/else 语句:

```
if(条件)
    表达式
else
    表达式
```

关键是 if 和 else 后都必须是单个表达式,而不能是其他 C 语言语句。

“?”叫做三目运算符,因为它要求有三个操作数,形式如下:

**EXP1 ? EXP2 : EXP3**

其中 EXP1、EXP2、EXP3 是表达式。请注意冒号的用法和它的位置。

?表达式的值是这样的。先求表达式 1 的值,若为真则求表达式 2 的值并把它作为整个表达式的值。如果表达式 1 为假,则计算表达式 3 的值并作为整个表达式的值。例如:

```
x = 10;
y = x>9?100:200;
```

在这个例子中, y 被赋值 100。如果 x 小于 9, y 就会被赋值 200。这个同样的程序若用 if/else 来写则为:

```
x = 10;
if(x>9) y = 100;
else y = 200;
```

不过,“?”运算符代替 if/else 语句并不只限于赋值语句。为了看看它的扩展用途,要记住很重要的一点,所有函数(除了由 void 定义的)均有返回值。因此,用一个或多个函数调用 C 语言表达式是允许的。当遇到函数名时,函数执行而且其返回值是确定的。因此,调用由“?”运算符组成的表达式的一个或多个函数是可以执行的。

例如:

```
#include <stdio.h>

int f2(void);
int f1(int n);

main(void)
{
    int t;

    printf(": ");
    scanf("%d",&t);

    /* print proper message */
    t ? f1(t)+f2() : printf("zero entered");
    return 0;
}

int f1(int n)
{
    printf("%d ",n);
}
```

```

int f2(void)
{
    printf("entered");
}

```

在这个简单程序中，如果输入一个零，则函数 `printf()` 被调用并显示信息 “zero entered”。如果输入其他数，`f1()` 和 `f2()` 就会被执行。

重要的是要记住 “?” 表达式的值在此例中被忽略了。把它赋给别处是不必要的。然而，注意即使函数 `f1()` 和 `f2()` 不返回值，它们仍被定义为 `int` 型。这是必要的，因为一个 `void` 函数不能使用任何形式的表达式，即使返回值被忽略。这就是 C 语言的灵活性。

这儿是 “?” 运算符执行的最后一个例子。它用于防止除以 0 的错误。

```

/* This program uses the ? operator to prevent
   a division by zero. */

#include <stdio.h>
int div_zero(void);

main(void)
{
    int i, j, result;

    printf("Enter dividend and divisor: ");
    scanf("%d%d", &i, &j);

    /* this statement prevents a divide by zero error */
    result = j ? i/j : div_zero();

    printf("Result: %d", result);
    return 0;
}

div_zero(void)
{
    printf("cannot divide by zero\n");
    return 0;
}

```

### 13.3 C 语言的简写

C 语言有一种特殊的简写方式专门用于简化一种赋值语句。例如，

```
x = x + 10;
```

可以简写为

```
x += 10;
```

运算符 `+=` 号告诉编译程序将 `x` 值加 10 赋给 `x`。

这种简写方法适用于 C 语言中所有的双目运算符(也就是要求有两个操作数的运算符)。

简写的一般形式为：

变量 运算符 = 表达式；

再举一个例子：

```
x = x - 100;
```

等价于：

```
x -= 100;
```

在专业编程者的 C 语言程序中广泛采用这种简写。应当熟悉它们。

### 13.4 逗号运算符

逗号运算符用于将多个表达式串在一起，逗号运算符的左边总是不返回的，也就是说，逗号右边表达式的值才是整个表达式的值。例如：

```
x=(y=3,y+1);
```

首先把 3 赋给 y，然后把 4 赋给 x。整个等式右边的表达式要用括号括起来，因为逗号运算符的优先级低于赋值号。

实际上，逗号使一系列的运算逐个执行。逗号用在赋值语句的右边，将一系列表达式逐个分隔开来，而赋给变量的值是最后一个表达式的值。例如：

```
y=20;
```

```
x=(y=y-5,30/y);
```

执行后，x 的值为 2。因为 y 的初值是 20，减 5 后为 15，再将 30 除以 15 所得结果是 2。

你可以认为逗号运算符和英语中的 and 有相同的含义，就象词组 “do this and this and this” 一样。

### 13.5 方括号和圆括号

在 C 语言中，圆括号和方括号是运算符。圆括号的作用在于提高它所包含的运算符的优先级别。方括号起数组元素下标的作用。要记住不是所有的计算机语言都将圆括号和数组下标符号当成运算符。

### 13.6 运算符优先级表

图 13-2 列出了所有 C 语言运算符的优先次序。所有的运算符，除了单目运算符和“?”运算符以外，都是从左至右关联的。而单目运算符“x”，“&”，“.”和“?”是从右到左关联的。

Highest	( ) [ ] -> .
	! ~ ++ -- (type) * & sizeof
	* / %
	+ -
	<< >>
	< <= > >=
	= == !=
	&
	-
	~
	&&
	&
	>
	= += -= *= /=
Lowest	

图 13-2 C 语言运算符优先级

## 第十四章 屏幕控制函数

毫无疑问,在现今交互式环境下成功的程序的发展要求对屏幕的完全控制。这一章介绍一些 Turbo C++ 的屏幕控制函数。请记住 Turbo C++ 的屏幕操作子系统包括大量的函数,比这一章中涉及的多得多,所以一定要查阅其它参考手册。

Turbo C++ 屏幕控制软件包被分成两部分:正文模式屏幕控制和图形模式屏幕控制。正文模式函数当显示器适配器处于正文模式时管理屏幕的显示,图形函数当计算机处于图形显示模式时作用于屏幕。虽然二者在概念上有联系,但这两个子系统实际上是完全分离的。这一章开头讨论一些常用的正文模式函数,另外还讲一些图形子系统。

注意:这一章中描述的正文和图形屏幕函数在 Turbo C 版本 1.5 中最先被研究。它们并不是 Turbo C++ 所独有的,因此作为 C 语言的一部分来讨论。不过,它们完全适用于 Turbo C 和 Turbo C++ 环境。

### 14.1 基本正文模式函数

在研究一些最重要的正文模式屏幕操作和控制函数之前,需要记住两点。首先,所有的 Turbo C++ 正文模式函数要求在用到它们的程序中包含头文件 CONIO.H。其次,正文模式函数要求屏幕处于正文状态而不是图形状态。虽然正文模式是绝大多数 PC 操作系统缺省设置的,这一点仍然不可忽视。简而言之,你将学习到如何设置所要求的显示器模式。

#### 14.1.1 正文窗口

Turbo C 正文模式子系统中绝大多数例程都对窗口操作,而不是对屏幕本身。幸运的是,窗口的缺省值就是指整个屏幕,所以不必担心要建立专门的窗口才能使用字符和图形程序。但是,了解窗口的基本概念会从 Turbo C++ 的屏幕功能上获益更多,这是十分重要的。

窗口是一个矩形的门户,用户程序可以通过它传递信息。窗口可以大到整个屏幕,也可以小到几个字符。在复杂的软件中,屏幕上常常可同时有几个窗口,每个都由程序用于执行独立的任务。

Turbo C++ 允许你定义窗口的位置和大小。当定义了一个窗口后,正文操作程序就仅仅在所定义的窗口活动,而不再是对整个屏幕了。例如,函数 clrscr() 仅清除当前窗口,而不是清除整个屏幕(当然,在缺省时,当前窗口就是整个屏幕)。另外,所有的位置坐标都是相对于当前窗口而言,而不再是相对于整个屏幕。

Turbo C++ 窗口最重要的方面之一是:对窗口的输出可以自动防止向窗口外溢出。如果一些输出超出了边界,只有那些在窗口内的部分才被显示出来,其余的将被自动转到下一行。

对正文窗口而言,左上角坐标是(1, 1)。正文模式屏幕函数使用的是相对于当前窗口的坐标,而不是相对于屏幕。因此,如果操作两个窗口,当其中任一个是当前窗口时其左上角坐标都被定位为(1, 1)。

这里，我们不必着急创建任何窗口，而仅仅使用缺省窗口，即整个屏幕。

#### 14.1.2 清除窗口

最喜欢用的一个最常见的屏幕操作函数可以清除窗口内的所有正文。这个函数叫做 `clrscr()`，它的原型是

```
void clrscr(void);
```

记住虽然这个函数看起来像 `clear screen` 的缩写，实际上它是用于清除当前窗口的。

#### 14.1.3 光标定位

清除了窗口以后第二个最常用的屏幕控制函数是 `gotoxy()`，用于光标定位。其原型是

```
void gotoxy(int x, int y);
```

这里，`x` 和 `y` 代表相对于当前窗口的光标的 `X` 和 `Y` 坐标。如果两个坐标都越界，则不进行任何动作。不过，记住一个越界坐标不会被认为是一个错误。

为了看一下 `clrscr()` 和 `gotoxy()` 是如何工作的，运行下面这段程序，它在屏幕上打印句子 “Text screen Function are Fun! ”。注意打印格式。

```
/* Demonstrate clrscr() and gotoxy(). */

#include <conio.h>
#include <stdio.h>
#include <string.h>

char mess[] = "Text Screen Functions are Fun!";

main(void)
{
    register int x, y;
    char *p;

    clrscr();

    p = mess;
    for(x=1; x<=strlen(mess); x++) {

        for(y=1; y<=12; y++) {
            gotoxy(x, y);
            printf("%c", *p);
            gotoxy(x, 25-y);
            printf("%c", *p);
        }
        p++;
    }
    getch();
    gotoxy(1, 25);

    return 0;
}
```

#### 14.1.4 清除到行末

如果用户程序要求输入，函数 `clrscr()` 会十分有用。它从左到右清除一行中从光标所在处到窗口的末尾那一段正文。它的原型是：

```
void clreol(void);
```

可以将此函数和 `gotoxy()` 一起用来构造一个函数，此函数可在清除行以后在特定坐标处显示提示信息。下面这个程序使用的像 `prompt()` 这样的函数，可使屏幕左边的信息不受影响。

```
#include <conio.h>
#include <stdio.h>

void prompt(char *s, int x, int y);

main(void)
{
    clrscr();
    prompt("this is a prompt", 1, 10);
    getch();
    prompt("this is too", 1, 10);
    getch();

    return 0;
}

void prompt(char *s, int x, int y)
{
    gotoxy(x, y);
    clreol();
    printf(s);
}
```

#### 14.1.5 删除和插入行

有时会发现满屏幕都是字符而且准备删除一行或多行，同时把该行下面的所有行上移。相应的，有时又想插入一空行。为了解决这些问题，可用函数 `delline()` 和 `insline()`。它们的原型如下：

```
void delline(void);
```

```
void insline(void);
```

调用 `delline()` 函数可删除光标所在行，同时把该行下面的所有行都上移一行。调用 `insline()` 函数插入一个新的空白行到光标所在行，同时下面的所有行都向下顺移一行。

下面的程序说明 `delline()`，先将屏幕各行填满，第一行由“A”组成，第二行由“B”组成，以此类推。然后它删除其它所有行。

```
#include <conio.h>
#include <stdio.h>

main(void)
{
    int x, y;

    clrscr();

    /* fill the screen with some lines */
    for(y=1; y<25; y++)
        for(x=1; x<80; x++) {
            gotoxy(x, y);
            printf("%c", (y-1)+'A');
        }
}
```

```

    /* delete every other line */
    for(y=2; y<26; y+=2) {
        gotoxy(1, y);
        delline();
    }

    return 0;
}

```

这个程序用 `insline` 稍微改动一下，用来在删除行的相同位置插入空白行。为了看看结果如何，将删除循环改为下面所示语句段。

```

/* delete every other line and insert a blank one */
for(y=2; y<26; y+=2) {
    gotoxy(1, y);
    delline();
    insline();
}

```

#### 14.1.6 建立窗口

迄今为止，所有例程都是用的缺省窗口。然而，可以在任何位置创建任意大小的窗口，只要屏幕装得下。这需要用到 `window()` 函数。它的原型如下所示：

```
void window(int left, int top, int, right, int bottom);
```

如果有一个坐标是无效的，`window()` 将不起作用。当成功地调用一次 `window()` 后，所有对坐标的引用都被转换为相对于这个窗口的，而不再是相对于整个屏幕。例如，下面这段程序建立了一个窗口，并且在窗口内的 (2,3) 坐标位置写一行字符。

```

printf("at screen location 2, 3");
window(10, 10, 60, 15);
gotoxy(2, 3);
printf("at window location 2, 3");

```

这段程序的工作过程见图 14-1。

注意：用来调用 `window()` 的坐标是屏幕绝对坐标，而不是相对于当前窗口的。这就是说多窗口不需要嵌套，而且一个窗口可以和另外一个窗口重叠。

下面这段程序首先沿屏幕画边框，然后再建立两个独立的带边框的窗口。每个窗口里字符的位置是由 `gotoxy()` 语句确定，该语句所用的坐标是相对于每个窗口而言的。最后，第三个窗口创建并覆盖了其它两个。这个程序的输出见图 14-2。

```

/* A text window demonstration program */
#include <conio.h>
#include <stdio.h>

void border(int, int, int, int);

main(void)
{
    clrscr();

    /* draw a border around the screen for perspective */
    border(1, 1, 79, 25);
}

```



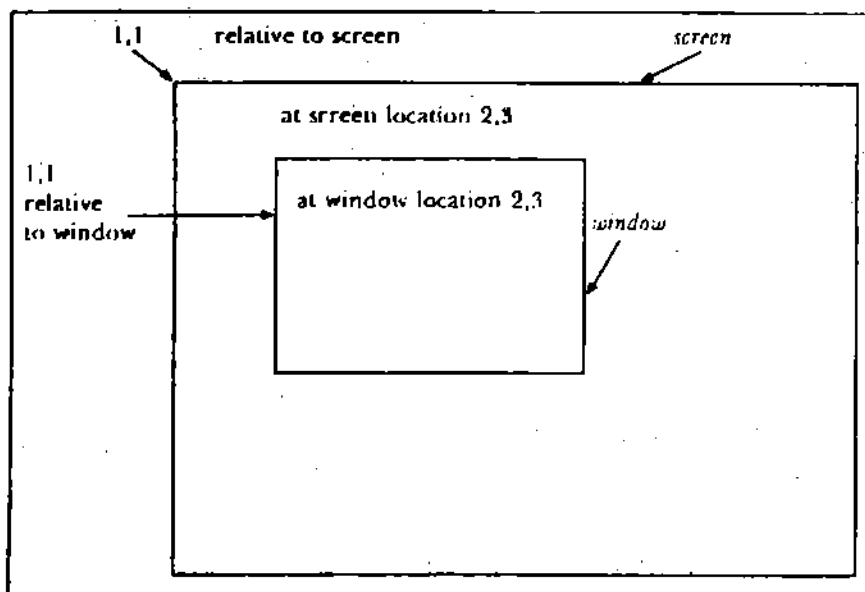


图 14-1 窗口内相对坐标的说明

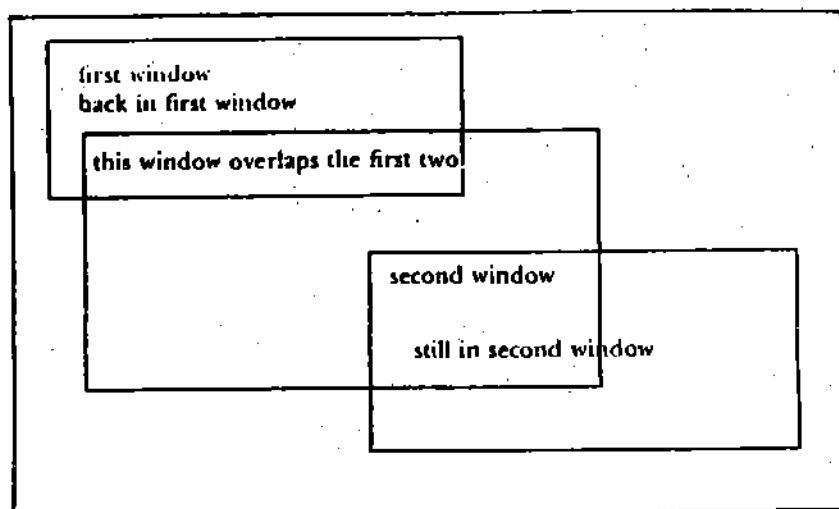


图 14-2 正文窗口示例程序输出

```
/* create first window */
window(3, 2, 40, 9);
border(1, 2, 40, 9);
gotoxy(3, 2);
printf("first window");

/* create a second window */
window(30, 10, 60, 18);
```

```

border(30, 10, 60, 18);
gotoxy(3, 2);
printf("second window");
gotoxy(5, 4);
printf("still in second window");

/* go back to first window */
window(3, 2, 40, 9);
gotoxy(3, 3);
printf("back in first window");
getch();

/* demonstrate overlapping windows */
window(5, 5, 50, 15);
border(5, 5, 50, 15);
gotoxy(2, 2);
printf("this window overlaps the first two");

getch();

return 0;
}

/* Draws a border around a text window. */
void border(int startx, int starty, int endx, int endy)
{
    register int i;

    gotoxy(1, 1);
    for(i=0; i<=endx-startx; i++)
        putchar('*');

    gotoxy(1, endy-starty);
    for(i=0; i<=endx-startx; i++)
        putchar('*');

    for(i=2; i<endy-starty; i++) {
        gotoxy(1, i);
        putchar('*');
        gotoxy(endx-startx+1, i);
        putchar('*');
    }
}

```

#### 14.1.7 一些窗口 I/O 函数

由于 C 语言的常规输出函数，如 `printf()` 没有被设计成适用于窗口屏幕环境，Turbo C++ 库中包含有可用于窗口的函数。当用缺省窗口即全屏幕时，是用基于窗口的 I/O 函数还是用标准函数都无关紧要。不过当用小窗口时，由于它们自动地防止字符写出该窗口，就需要用到窗口定向函数。当用窗口 I/O 函数时，正文自动在窗口边界下卷。新的或修整过的正文 I/O 函数可见表 14-1。

`cprintf()` 函数除了它承认窗口系统以外，操作使用完全与 `printf()` 一样。`cputs()` 函数在方式上也相当于 `puts()`，它与 `puts()` 的区别实际上仅仅在于承认窗口。与标准 I/O 函数不同，如 `printf()`，这些函数自动地防止输出超过当前窗口的部分溢出到屏幕的其它地方。`putch()` 也同样如此。它不允许字符被写到当前窗口以外。函数 `getche()` 也不会接收来自当前窗口以外的输入。最后，`cgets()` 也被修改为承认窗口并且不允许用户输入超过窗口边界

表 14-1 窗口用正文 I/O 函数

函数	目的
<code>cprintf()</code>	将格式化的输出送到当前窗口
<code>cputs()</code>	将一个字符串送到当前窗口
<code>putch()</code>	将单个字符送到当前窗口
<code>getche()</code>	读入一字符并回显到当前窗口
<code>cgets()</code>	读入一字符串并回显到当前窗口

的正文。

为了弄清窗口 I/O 函数和标准函数的区别，试运行下面的程序并观察它的结果。如程序所设想的那样，`cprintf()` 的输出行在窗口的边界下卷，而用 `printf()` 显示则不会这样。

```

/* Demonstration cprintf(). */
#include <conio.h>
#include <stdio.h>

void border(int, int, int, int);

main(void)
{
    clrscr();

    /* create a window */
    window(3, 2, 20, 9);
    border(3, 2, 20, 9);
    gotoxy(2, 2);
    cprintf("This will wrap around at the edge of the window");
    printf("This will not wrap at the edge of the window");

    getch();

    return 0;
}

/* Draws a border around a text window. */
void border(int startx, int starty, int endx, int endy)
{
    register int i;

    gotoxy(1, 1);
    for(i=0; i<=endx-startx; i++)
        putch('*');

    gotoxy(1, endy-starty);
    for(i=0; i<=endx-startx; i++)
        putch('*');

    for(i=2; i<=endy-starty; i++) {
        gotoxy(1, i);
        putch('*');
        gotoxy(endx-startx+1, i);
    }
}

```

```

        putchar('*');
    }
}

```

另一个需要弄清楚的是，这些基本 I/O 函数是不会重定向的。也就是说，C 语言的标准 I/O 函数允许输出重定向。即从一个磁盘文件或辅助设备输入改为向其输出，或输出改为输入，而基本窗口的字符屏幕函数却不能改变。

#### 14.1.8 正文模式

到现在为止，计算机的缺省显示器模式，绝大多数均为 25 行 80 列。不过，实际上仍然有五种不同的正文模式可供选择。如果你的计算机有图形适配器，就可以在 40 列与 80 列模式和彩色与黑白模式间自由选择。可以用函数 `textmode()` 改变正文模式。它的原型是：

```
void textmode(int mode);
```

参数 `mode` 必须是表 14-2 中所示值中的一种。既可以使用整型量也可使用宏替换名(这些宏在 `CONIO.H` 中有定义)。

表 14-2 正文显示器模式

宏名	数值	描述
<code>BW40</code>	0	40列黑白
<code>C40</code>	1	40列彩色
<code>BW80</code>	2	80列黑白
<code>C80</code>	3	80列彩色
<code>MONO</code>	7	80列单色
<code>LASTMODE</code>	-1	启用先前模式

将函数 `textmode()` 包含在程序中的主要原因是为了确保当前显示器模式正是程序要求的。

如果程序设置了显示器模式，在退出时必须将其保存起来。为了做到这点，用 `LASTMODE` 作为参数调用 `textmode()`。这能自动保存在程序中修改了的显示器模式。

#### 14.1.9 用彩色输出正文

如果有彩色监视器和彩色图形适配器，就可以用不同颜色来显示正文。你可以修改正文的颜色和背景的颜色。一定要记住：只有特殊的窗口 I/O 函数才能用于显示特定颜色的正文。像 `printf()` 这样的函数则不起作用。

函数 `textcolor()` 可改变正文颜色到指定颜色。它还可用于使字符闪烁。`textcolor()` 的原型是

```
void textcolor(int color);
```

参数 `color` 对应于不同的颜色。有从 0 到 15 这些数值。不过，在头文件 `CONIO.H` 中这些颜色均有宏替换名而且相当好记。这些宏替换名和相应数值见表 14-3。

表 14-3 正文颜色宏名和对应数值表

<i>Macro</i>	<i>Integer Equivalent</i>
BLACK	0
BLUE	1
GREEN	2
CYAN	3
RED	4
MAGENTA	5
BROWN	6
LIGHTGRAY	7
DARKGRAY	8
LIGHTBLUE	9
LIGHTGREEN	10
LIGHTCYAN	11
LIGHTRED	12
LIGHTMAGENTA	13
YELLOW	14
WHITE	15
BLINK	128

正文颜色的变化仅仅影响紧随其后的写操作，记住这一点十分重要；它不会改变已在屏幕上显示了正文。

要使正文闪烁。必须将想要的颜色和数值 128(BLINK)相或(OR 操作)。例如，下面这句代码使得其后输出的正文为绿色闪烁。

```
textcolor(GREEN|BLINK);
```

函数 `textbackground()` 用于设置正文屏幕的背景颜色。相对于 `textcolor()` 而言，调用 `textbackground()` 只影响随后写入的字符的背景颜色。其原型为：

```
void textbackground(int color);
```

`color` 的值必须是在 0 至 6 的范围内。这意味着背景颜色只能用表 14-3 所示的前七种。

下面这个程序表明正文颜色函数功能，它能显示所有前景和背景颜色的组合：

```
/* This program demonstrates color text. */
#include <conio.h>
main(void)
{
    register int fg, bg;

    textmode(C80);

    for(fg=BLUE; fg<=WHITE; fg++) {
        for(bg=BLACK; bg<=LIGHTGRAY; bg++) {
            textcolor(fg);
            textbackground(bg);
            cprintf("this is a test ");
        }
    }
}
```

```

        cprintf("\n\r");
    }
    textcolor(WHITE);
    textbackground(BLACK);
    cprintf("done");
    textmode(LASTMODE);

    return 0;
}

```

## 14.2 Turbo C++的图形子系统介绍

Turbo C++的第二个屏幕控制子系统是它的图形软件包。这个软件包包含了大量函数，用于完成从画线、画圆到构成棒状、饼状结构图的各种任务。这一节将向你介绍最常用的那些函数。

图形函数的原型在文件 GRAPHICS.H 中。这个函数必须被那些用到图形函数的程序包含。因为所有的图形函数都是 *far* 函数，而用户程序在使用它们之前必须知道这一点。

注意：为了试运行以后的程序，你的计算机必须配置图形适配器，有彩色监视器更好。

### 14.2.1 一个有别名的窗口

像正文屏幕控制函数一样，所有图形函数都通过窗口操作。在 Turbo C++术语中，一个图形窗口叫做视口(viewport)，视口实际上与正文窗口相似。窗口和视口之间的唯一区别是视口的左上角坐标是(0, 0)，而不是像窗口那样为(1, 1)。

缺省时，整个屏幕就是视口。不过，可以创建具有其它大小的视口。你将在本章稍后学到如何创建视口。当学习以下内容时，所有的图形输出都是相关于当前视口，而视口未必是整个屏幕。

### 14.2.2 初始化显示器适配器

在图形函数被使用之前，必须把显示器适配器设置为某一图形模式。缺省时，绝大多数计算机系统都使用 DOS 的 80 列正文模式。由于这不是图形模式，图形函数不能工作。若要将显示器适配器设置为图形模式，必须使用 `initgraph()` 函数。其原型如下所示：

```
void far initgraph(int far *drive, int far *mode, char far *path);
```

`initgraph()` 函数将图形驱动程序装入到内存。该驱动程序为 `driver` 所指的数字。如果没有装入图形驱动程序，图形函数将不能工作。参数 `mode` 是用来确定显示器模式的整型指针。最后，可以用由 `path` 所指的字符串确定进入该驱动软件的路径。如果没有指出路径，就在当前目录下搜寻。

图形驱动程序包含在.BGI 文件中，这个文件必须是系统可以得到的。这些驱动程序是由 Turbo C++提供的。不过你不必为文件的实际名字担忧，因为只需要用其数字来指定该驱动程序。在 GRAPHICS.H 中为用于此目的数字定义了符号值。如下所示：

Macro	Equivalent
DETECT	0

CGA	1
MCGA	2
EGA	3
EGA64	4
EGAMONO	5
IBM8514	6
HERCMONO	7
ATT400	8
VGA	9
PC3270	10

当选用了 DETECT 时, initgraph()会自动在当前系统下搜寻显示器硬件类型, 并且选用最大可能的分辨率的屏显模式。同时 driver 和 mode 被设成特定的值。

mode 的值必须是表 14-4 所示的图形模式值之一。例如, 为了将图形系统初始化成 CGA 四种颜色、320×200 的图形模式, 应该用下面这段程序。假设图形驱动程序的.BGI 文件在当前目录下。

```
#include <graphics.h>
.
.
int driver, mode;

driver = CGA;
mode = CGAC0;

initgraph(&driver, &mode, "");
```

表 14-4 Turbo C 图形驱动程序和模式宏定义

<i>Driver</i>	<i>Mode</i>	<i>Equivalent</i>	<i>Resolution</i>
CGA	CGAC0	0	320 × 200
	CGAC1	1	320 × 200
	CGAC2	2	320 × 200
	CGAC3	3	320 × 200
	CGAHI	4	640 × 200
MCGA	MCGAC0	0	320 × 200
	MCGAC1	1	320 × 200
	MCGAC2	2	320 × 200
	MCGAC3	3	320 × 200
	MCCAMED	4	640 × 200
EGA	MCGAHI	5	640 × 480
	EGALO	0	640 × 200
	EGAHI	1	640 × 350
EGA64	EGA64LO	0	640 × 200
	EGA64HI	1	640 × 350
EGAMONO	EGAMONOH1	3	640 × 350

HERC	HERCMONOH1	0	720 × 348
ATT400	ATT400C0	0	320 × 200
	ATT400C1	1	320 × 200
	ATT400C2	2	320 × 200
	ATT400C3	3	320 × 200
	ATT400CMED	4	640 × 200
VGA	ATT400CHI	5	640 × 400
	VGALO	0	640 × 200
	VGAMED	1	640 × 350
	VGAHI	2	640 × 480
PC3270	PC3270HI	0	720 × 350
IBM85	IBM8514LO	0	640 × 480
	IBM8514HI	1	1024 × 768

### 14.2.3 退出图形模式

若要终止图形模式并返回到正文模式，用 `closegraph()` 或 `restorecrtmode()` 函数。它们的原型如下：

```
void far closegraph();
```

```
void far restorecrtmode();
```

函数 `closegraph()` 用于当用户程序还要继续在字符模式下运行时。它释放由图形函数所占用的内存，同时将显示器模式恢复为调用 `initgraph()` 之前的模式。如果用户程序要终止运行，可以用 `restorecrtmode()` 将显示器适配器设置为原来的模式，即首次调用 `initgraph()` 之前的模式。

### 14.2.4 颜色和调色板

显示器适配器的类型是与系统要求颜色的类别和数目相关的，这限于图形模式中。CGA 和 EGA/VGA 之间的区别是适配器之间最大的区别。

CGA 四色图形模式给了四块调色板，每块板上有四种颜色可供选择。颜色数值从 0 到 3，0 总是为背景色。调色板的数值也是从 0 到 3。若要选择某调色板，就要设置参数 `mode` 为 `CGACx`，这里  $x$  为调色板号。调色板与其相关色见表 14-5。

表 14-5 CGA 的调色板与颜色值

Color Number				
Palette	0	1	2	3
0	background	GREEN	RED	YELLOW
1	background	CYAN	MAGENTA	WHITE
2	background	LIGHTGREEN	LIGHTRED	YELLOW
3	background	LIGHTCYAN	LIGHTMAGENTA	WHITE



在 EGA/VGA16 色图形模式中，一个调色板可以有 16 种颜色，它们是从 64 种颜色中选择出的。

用函数 `setpalette()` 可以改变调色板。该函数原型是：

```
void far setpalette(int index, int color);
```

初次使用此函数会有某些困难。实际上，该函数用一个表把 color 值和 index 联系起来，该表将所要求的颜色变换成屏幕上的实际颜色。颜色编号如表 14-6 所示。

表 14-6 函数 `setpalette()` 的颜色编号

<i>CGA (background only)</i>	
<i>Macro</i>	<i>Value</i>
BLACK	0
BLUE	1
GREEN	2
CYAN	3
RED	4
MAGENTA	5
BROWN	6
LIGHTGRAY	7
DARKGRAY	8
LIGHTBLUE	9
LIGHTGREEN	10
LIGHTCYAN	11
LIGHTRED	12
LIGHTMAGENTA	13
YELLOW	14
WHITE	15
 <i>EGA and VGA</i>	
<i>Macro</i>	<i>Value</i>
EGA_BLACK	0
EGA_BLUE	1
EGA_GREEN	2
EGA_CYAN	3
EGA_RED	4
EGA_MAGENTA	5
EGA_BROWN	20
EGA_LIGHTGRAY	7
EGA_DARKGRAY	56

---

EGA_LIGHTBLUE	57
EGA_LIGHTGREEN	58
EGA_LIGHTCYAN	59
EGA_LIGHTRED	60
EGA_LIGHTMAGENTA	61
EGA_YELLOW	62
EGA_WHITE	63

---

对于 CGA 模式, 只有背景色能改变。而设置背景色时 index 总取值 0。所以, 对于 CGA 模式, 下面的语句可将背景色改为绿色。

```
setpalette(0, GREEN);
```

EGA 能在总共 64 种颜色中同时显示 16 种。可以用 setpalette() 函数将某一种颜色设置为这 16 种不同颜色中的一种。例如, 下面的语句设置颜色 5 的值代表青色。

```
setpalette(5, EGA_CYAN);
```

#### 14.2.5 基本图形函数

最基本的图形函数不外乎画一个点、线和圆。这些函数分别叫做 putpixel(), line() 和 circle(), 它们的原型为:

```
void far putpixel(int x, int y, int color);
```

```
void far line(int startx, int starty, int endx, int endy);
```

```
void far circle(int x, int y, int radius);
```

putpixel() 函数将 color 所指定的颜色写到 x 和 y 坐标确定的点上。line() 函数是从指定点 (startx, starty) 到 (endx, endy) 之间画一条直线, 其颜色为当前色。circle() 函数画一个以 radius 为半径的圆, 颜色为当前绘图色, 圆心位置为 (x, y)。如果坐标超出了范围, 则超出部分不画。

下面的程序可用于说明这些函数:

```
/* Points, lines, circles demonstration. */
#include <graphics.h>
#include <conio.h>

main(void)
{
    int driver, mode;

    register int i;

    driver = DETECT;
    initgraph(&driver, &mode, "");

    line(0, 0, 200, 150);
    line(50, 100, 200, 125);

    /* some points */
    for(i=0; i<319; i+=10) putpixel(i, 100, RED);

    /* draw some circles */
    circle(50, 50, 35);
    circle(100, 160, 100);
```

```

    getch(); /* wait until keypress */
    restorecrtmode();

    return 0;
}

```

#### 14.2.6 改变绘图色

刚才的例子用白色画线和圆，白色是缺省绘图色。(记住，函数 `putpixel` 所画颜色由三个参数指定)。可以用函数 `setcolor()` 设置绘图的当前色。其原型如下：

```
void far setcolor(int color);
```

`color` 的值一定要在当前图形模式下的有效范围内。一旦改变了当前色，则其后的写操作沿用这个新颜色。

#### 14.2.7 区域填充

可以用函数 `floodfill()` 填充任何封闭的图形。其原型如下：

```
void far floodfill(int x,int y,int bordercolor);
```

若用该函数去填充一个封闭的图形，则应以图形中任一点为坐标，以图形建立时边线的颜色为其颜色来调用该函数。一定要确保所填充的图形是完全封闭的。如果不这样，图形以外的一些区域也将被填充。可以用当前的填充模式和颜色来填充图形。不过，也可以用函数 `setfillstyle()` 来改变填充对象的填充模式。该函数的原型如下：

```
void far setfillstyle(int pattern, int color);
```

`pattern` 的值及其宏各见表 14-7(在 `GRAPHICS.H` 中定义)。

表 14-7 填充模式

宏名	值	填充模式
<code>EMPTY_FILL</code>	0	以背景色着色
<code>SOLID_FILL</code>	1	全部着色
<code>LINE_FILL</code>	2	填水平线
<code>LTSLASH_FILL</code>	3	填左斜线
<code>SLASH_FILL</code>	4	填粗左斜线
<code>BKSLASH_FILL</code>	5	填粗右斜线
<code>LTBKSLASH_FILL</code>	6	填右斜线
<code>HATCH_FILL</code>	7	填浅阴影线
<code>XHATCH_FILL</code>	8	填重交叉阴影线
<code>INTERLEAVE_FILL</code>	9	填交替直线
<code>WIDEDOT_FILL</code>	10	填稀点
<code>CLOSEDOT_FILL</code>	11	填密点
<code>USER_FILL</code>	12	用户定义

### 14.2.8 rectangle()函数

函数 `rectangle()` 用当前绘图色画一个(left, top)和(right, bottom)为坐标的矩形。其原型为。

```
void far rectangle(int left, int top, int right, int bottom);
```

下面这个程序演示了迄今学到的所有图形函数，除了 `setpalette()`。它的输出见图 14-3。这个程序要求 VGA 适配器；不过，可以很方便地改动此程序适用于你的图形适配器类型。

```
/* Color, rectangle, and fills demonstration program. */
#include <graphics.h>
#include <conio.h>

main(void)
{
    int driver, mode;

    register int i;

    driver = VGA;
    mode = VGAMED;
    initgraph(&driver, &mode, "");

    /* outline the screen */
    rectangle(0, 0, 639, 349);

    setcolor(RED);
    line(0, 0, 639, 349);

    setcolor(GREEN);
    rectangle(100, 100, 300, 200);

    setcolor(BLUE);
    floodfill(110, 110, GREEN); /* fill part of a box */

    setcolor(CYAN);
    line(50, 200, 400, 125);

    /* draw some circles */
    setcolor(RED);

    for(i=0; i<640; i+=3) line(320, 174, i, 0);

    setcolor(GREEN);
    circle(50, 50, 35);
    circle(320, 175, 100);
    circle(500, 250, 90);
    circle(100, 100, 200);

    /* make a bullseye */
    setfillstyle(SOLID_FILL, GREEN);
    floodfill(500, 250, GREEN); /* fill part of a circle */

    setcolor(RED);
    circle(500, 250, 60);
    setfillstyle(SOLID_FILL, RED);
    floodfill(500, 250, RED);

    setcolor(GREEN);
    circle(500, 250, 30);
    setfillstyle(SOLID_FILL, GREEN);
    floodfill(500, 250, GREEN); /* fill part of a circle */
}
```

```

setcolor(RED);
circle(500, 250, 10);
setfillstyle(SOLID_FILL, RED);
floodfill(500, 250, RED);

getch(); /* wait until keypress */
restorecrtmode();

return 0;
}

```

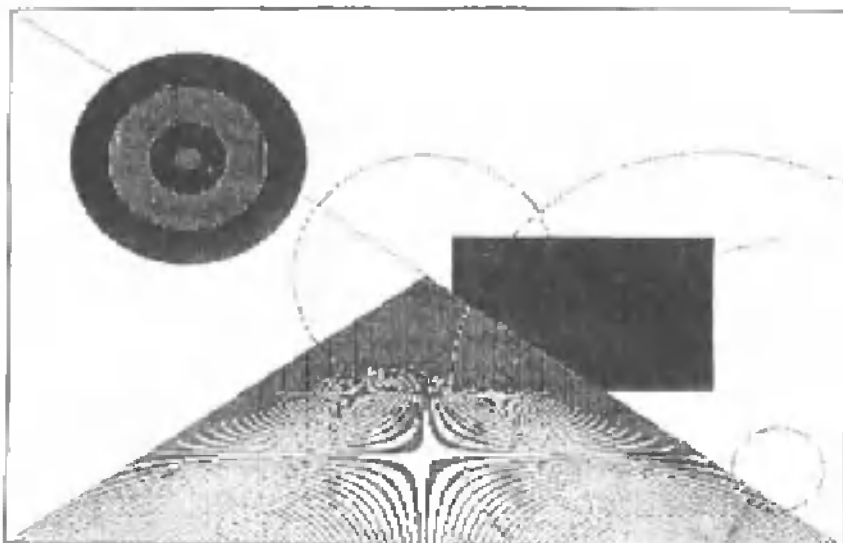


图 14-3 绘图演示程序的输出

#### 14.2.9 创建视口

可以像正文模式中创建窗口相似的格式来创建图形模式下的视口。如前所述,所有的图形输出都是与当前视口的坐标相关的。这意味着视口的左上角坐标为(0, 0),不管视口在屏幕上的什么位置。用于创建视口的函数叫做 `setviewport()`。它的原型如下:

```
void far setviewport(int left,int top,int right,int clipflag);
```

用此函数时,需要确定视口在屏幕上的左上角和右下角坐标。如果参数 `clipflag` 为非零值,则自动裁去超越视口边界的输出部分。否则,剪裁标识被关上,输出就有可能超出视口。记住一点,输出被裁去不算出错。

可以用函数 `getviewsettings()` 来获知当前视口的尺寸参数。它的原型是

```
void far getviewsettings(struct viewporttype far *info)
```

结构 `viewporttype` 在 `GRAPHICS.H` 中定义如下:

```

struct viewporttype{
    int left,top,right,bottom;
    int clipflag;
}

```

其中 left, top, right 和 bottom 域装有视口的左上角和右下角坐标。当 clipflag 为零时, 输出越过视口边界则不会被截掉。否则, 剪裁标识将防止边界溢出。

函数 `getviewsettings()` 最重要的用途是允许用户程序能自动调整以适应于使用中的显示器适配器类型, 这是通过检查视口的尺寸以及作相应的调整来实现的。例如, 下面的程序用到函数 `setviewport()` 和 `getviewsettings()` 构成一个以屏幕为中心的第二个视口。在第一个视口中, 画着垂直线。在第二个视口中, 画的是水平线。这个程序在任何图形适配器下均能工作。

```
/* Demonstrate viewports */
#include <graphics.h>
#include <stdlib.h>
#include <conio.h>

main(void)
{
    int driver, mode;
    struct viewporttype info;
    int width, height;
    register int i;

    driver = DETECT;
    initgraph(&driver, &mode, "");

    /* Get screen dimensions */
    getviewsettings((struct viewporttype far *) &info);

    /* draw some vertical lines in first viewport */
    for(i=0; i<info.right; i+=20)
        line(i, info.top, i, info.bottom);

    /* create new viewport in the center of the screen */
    height = info.bottom/4;
    width = info.right/4;
    setviewport(width, height, width*3, height*3, 1);
    getviewsettings((struct viewporttype far *) &info);

    /* draw some horizontal lines inside second viewport */
    for(i=0; i<info.right; i+=10)
        line(0, i, info.bottom, i);

    getch(); /* wait for keypress */
    restorecrtmode();

    return 0;
}
```

请记住这一章仅仅是对最常见的正文屏幕和图形函数作了一个介绍。你可以在《库函数参考手册》查到所有屏幕和图形函数的细节。

## 第十五章 C 预处理指令

Turbo C++ 源程序可以含有多种预处理指令。这些指令并不是真正属于 C 语言的内容，它们用来扩展 C 语言程序的编程环境。本章将介绍 Turbo C++ 的预处理指令和内部宏定义。

### 15.1 C 预处理指令

C 语言预处理指令包括下列指令：

```
#if
#ifdef
#ifndef
#else
#elif
#endif
#include
#define
#undef
#line
#error
#pragma
```

从上可看出，所有预处理指令前均带有符号#。下面来逐一介绍。

### 15.2 #define 指令

尽管以前你曾接触过#define，现在我们仍需详细研究一下。就其最简单形式而言，#define 用来定义一个标识符和一个字符串，在程序中每次遇到该标识符时就用所定义的字符串替换它。这个标识符叫做宏替换名，替换过程叫做宏替换。宏定义指令#define 的一般形式是：

**#define 宏替换名 字符串**

注意这个语句里没有分号。在标识符和字符串之间可以有任意个空格，但是字符串结束后一定要换行。

例如，可以定义 MAX 代表数值 100：

```
#define MAX 100
```

这样，每当在源程序中遇到宏替换名 MAX 时就自动用 100 来代替。

还可以用宏替换名来代替某一字符串。例如，下面的程序代码打印字符串“version 2.12”。

```
#define VERSION "version 2.12"
printf(VERSION);
```

记住，如果宏替换名出现在一个字符串中则不产生替换。例如，

```
printf("HARD DISK VERSION");
```

将不会将 VERSION 替换显示, 而只显示引号内原文。

如果字符串的长度大于一行, 可以用一个反斜杠符号写在每一行的行末, 如下例所示:

```
#define LONG_STRING "this is a how to handle a long\
string using #define statement."
```

在 C 语言程序中习惯上用大写字母作为宏替换名。这样在读程序时一眼就可以看出哪儿有宏替换。而且, 最好把 #define 语句都放在程序开头或写成单独的包含文件, 而不要在整个程序中任意插入。

宏替换最常见的用法是定义一个“数字”名。例如在程序中定义一个数组, 当有多个程序要访问这个数组时, 若用一个常数来说明数组的大小, 数组的大小就固定了。不如定义一个数组大小名, 并在用到此数组大小的地方用这个数组大小名替换。这样, 如果要改变数组的大小, 只须将程序开头处的 #define 语句改动一下并重新编译即可。例如:

```
#define MAX_SIZE 100
float balance[MAX_SIZE];
```

#define 指令还有一个重要的特点: 宏替换名可以带形式参数。在程序中遇到时, 实参就会代替这些形参。例如:

```
#include <stdio.h>

#define MIN(a,b) ((a)<(b)) ? (a) : (b)

main(void)
{
    int x, y;

    x = 10;
    y = 20;
    printf("the minimum is: %d", MIN(x,y));

    return 0;
}
```

当程序被编译时, 由 MIN(a,b) 代表的表达式就被替换了, 把 x 和 y 代入作为操作数来使用。也就是说, printf() 语句代换为如下形式:

```
printf("the minium is:%d",((x)<(y))?(x):(y));
```

在宏定义中 a 和 b 被括号括起来的原因是为了保证被 a 和 b 替换的任何表达式都完全等价。有时会因为不加括号而导致发生错误。

在实型函数处使用宏替换的主要好处是: 在函数调用时没有多余操作, 因而可以提高速度。然而, 提高速度的代价是程序变长了, 因为这中间有些是重复的。

### 15.3 #error 指令

编译中遇到 #error 指令时就停止编译。它原先用于调试。该指令的一般形式为

#error 出错信息

其中出错信息不加引号。当编译程序遇到这条指令时, 显示下述信息并停止编译:

Error 文件名 行数: 出错指令: 出错信息



## 15.4 #include 指令

**#include** 预处理指令的作用是指示编译程序将该指令所指的另一个源文件包含到 **#include** 指令所在的程序。源文件名应使用双引号或尖括号括起来。例如:

```
#include "stdio.h"
```

```
#include <stdio.h>
```

都会指示 C 编译程序读取并编译盘文件中库程序的头文件。

允许被包含的文件也带有 **#include** 指令。这叫做嵌套包含。

如果文件的明确路径名是与文件名一起给出的, 编译时将只在所指定的目录中查找包含文件。如果文件名包括在双引号内, 将首先查找当前工作目录。如果未找到, 则在命令行所指定的目录中继续搜索。最后, 若仍未找到这个文件, 就在由环境工具指定的标准目录中去查找。

在尖括号内的包含文件如果没有给出路径名, 由编译命令行所指定的目录中寻找。如果没有找到则搜索标准目录, 但绝不会在当前工作目录中寻找这个文件。

## 15.5 条件编译指令

有若干个条件编译指令允许你有选择地编译程序中的某些部分。这个过程叫做条件编译, 它们广泛地用在商业软件中, 为一个程序提供各种定做的版本。

### 15.5.1 #if、#else、#elif 和 #endif

**#if** 的通常含义是: 如果 **#if** 指令后的常数表达式为真, 则编译从 **#if** 到 **#endif** 之间的程序段。否则跳过这段程序。**#endif** 指令用来标识 **#if** 段的结束。

**#if** 指令的一般格式为

```
#if 常数表达式
```

```
    语句段
```

```
#endif
```

如果常数表达式为真, 这段程序则被编译; 否则它将被跳过去。例如:

```
/* simple #if example */  
  
#include <stdio.h>  
  
#define MAX 100  
  
main(void)  
{  
    #if MAX>99  
        printf("MAX greater than 99\n");  
    #endif  
  
    return 0;  
}
```

由于程序中 **MAX** 是大于 99 的, 因此在屏幕上显示信息。这个例子说明了很重要的一点, 即 **#if** 后的表达式是在编译时求值。因此, 它只能是由事先已定义过的宏替换名和常量组成, 而不能使用任何变量。

**#else** 的作用和 C 语言中 **else** 指令的作用十分相似: 当 **#if** 为假时它提供另一种选择

上。上面的例子可扩展如下:

```
/* simple #if/#else example */  
  
#include <stdio.h>  
  
#define MAX 10  
  
main(void)  
{  
    #if MAX>99  
        print("MAX greater than 99\n");  
    #else  
        printf("MAX less than 99\n");  
    #endif  
  
    return 0;  
}
```

这里, MAX 被定义为一个小于 99 的数, 所以 #if 后的程序段不被编译, 而编译 #else 后的那段程序。因而, 将显示信息 "MAX less than 99"。

注意, #else 既是 #if 段的结束标志, 也是 #else 段开始的标识。对于一个 #if 只能有一个 #endif 与它配对使用。

#elif 指令表示 "else if", 它用来建立一种 if/else/if 这样的阶梯状多重编译操作选择。#elif 后跟一个常量表达式。如果表达式值为真, 则编译它后面的那段程序, 并且不再继续检验后继的 #elif 表达式。否则继续检查下一个 #elif 条件。下面是其一般形式:

```
#if 表达式  
    语句段  
#elif 表达式 1  
    语句段  
#elif 表达式 2  
    语句段  
#elif 表达式 3  
    语句段  
.  
.  
.  
#elif 表达式 N  
    语句段  
#endif
```

例如, 下面这个程序用 ACTIVE\_COUNTRY 的值来决定货币符号:

```
#define US 0  
#define ENGLAND 1  
#define FRANCE 2  
  
#define ACTIVE_COUNTRY US  
  
#if ACTIVE_COUNTRY==US  
    char currency[]="dollar";  
#elif ACTIVE_COUNTRY==ENGLAND
```

```

    char currency[]="pound";
#else
    char currency[]="franc";
#endif

```

可以将**#if**和**#elif**指令与**#endif**、**#else**或**#elif**构成嵌套结构，并与最近的一个**#if**或**elif**配对。下例写法是有效的：

```

    #if MAX>100
        #if SERIAL_VERSION
            int port=198;
        #else
            int port=200;
        #endif
    #else
        char out_buffer[100];
    #endif

```

### 15.5.2 #ifdef 和 #ifndef 指令

另一种条件编译方法是使用**#ifdef**和**#ifndef**指令。它们的含义是“如果宏已定义”和“如果宏未定义”。

**#ifdef**指令的一般形式是：

```

    #ifdef 宏替换名
        语句段
    #endif

```

如果宏替换名在此之前已经由**#define**语句给出了定义，就编译**#ifdef**和**#endif**之间的语句。

**#ifndef**指令的一般形式是：

```

    #ifndef 宏替换名
        语句段
    #endif

```

如果宏替换名在此之前未经**#define**语句定义，则编译此代码段。

指令**#ifdef**和**#ifndef**都可以和**#else**指令一起使用，但不能和**#elif**指令一起使用。例如：

```

#include <stdio.h>

#define JON 10

main(void)
{
    #ifdef JON
        printf("Hi Jon\n");
    #else
        printf("Hi anyone\n");
    #endif
    #ifndef RACHEL
        printf("RACHEL not defined\n");
    #endif

    return 0;
}

```

这个程序将打印出“Hi Jon”和“RACHEL not defined”。但是如果事先没有定义

JON, 则显示 “Hi anyone” 和 “RACHEL not defined”。

可以像使用 #if 等指令那样使用 #ifdef 和 #ifndef 构成各种嵌套形式。

### 15.6 #undef 指令

#undef 指令用于删除事先定义了的宏定义。其通常形式为:

#undef 宏替换名

例如:

```
#define LEN 100
#define WIDTH 100

char array[LEN][WIDTH];

#undef LEN
#undef WIDTH
/* at this point both LEN and WIDTH are undefined */
```

在 #undef 语句出现之后, LEN 和 WIDTH 均不再具有以前的宏定义了。

#undef 主要用来使宏替换名只限定在需要使用它们的程序段中。

### 15.7 #line 指令

#line 指令用来改变编译程序中预定义的宏替换名 \_\_LINE\_\_ 和 \_\_FILE\_\_ 的内容。此命令的基本形式是:

#line 行号 “文件名”

其中行号为任一正整数, 可选项文件名可是任一有效的文件标识符。该行号为源程序中当前行的行号, 文件名是源程序文件名。#line 指令最初用于程序调试和一些特殊应用中。它在集成环境下被忽略而用于命令行编译中。

例如, 下列程序指定行计数从 100 开始。语句 printf() 是 #line 100 语句后第三行, 故将显示行号 102。

```
#include <stdio.h>

#line 100 /* reset the line counter */
main(void) /* line 100 */
{
    /* line 101 */
    printf("%d\n", __LINE__); /* line 102 */
    return 0;
}
```

### 15.8 #pragma 指令

#pragma 指令是由 ANSI 标准定义的, 它使编译程序发生器发出各种命令给编译程序。#pragma 指令的一般形式是:

#pragma 名字

这里名字就是所调用的 #pragma 的名字。Turbo C++ 定义了七类 #pragma 语句:

argsused

exit

startup

inline  
option  
saveargs  
warn

虽然这些**#pragma**命令的使用需要许多你未学到的知识, 这里我们简要介绍每一条命令。

**argsused** 指令必须用在一函数前。如果一个函数的参数在函数体中未被用到, 则该指令防止产生出错信息。通常, 这条指令很少被用到。

**exit** 指令指定当程序终止时将被调用到一个或多个函数。**startup** 指令指定当程序开始运行时需要调用的一个或多个函数。它们的一般形式是:

```
void func(void)
```

下面的例子定义了一个名叫 **start()** 的启动函数和一个叫做 **stop()** 的终止程序。

```
#include <stdio.h>

void stop(void);
void start(void);

#pragma exit stop 101
#pragma startup start 101
main(void)
{
    printf("In main.\n");
    return 0;
}

void stop(void)
{
    printf("Program is terminating.");
}

void start(void)
{
    printf("Program is starting.\n");
}
```

如例所示, 必须在**#pragma**语句前为所有 **exit** 和 **startup** 函数提供函数原型。记住, 不管是 **start()** 还是 **stop()** 都未被程序明确调用。它们在程序开始和结束执行时自动执行。当这个程序运行时, 输出显示如下内容:

```
Program is starting.
In main.
Program is terminating.
```

另外一个**#pragma**语句是 **inline**, 其一般形式为:

```
#pragma inline
```

这条指令告诉 Turbo C++ 在程序中包括一段汇编语言, 为达到最高的效率, Turbo C++ 需要预先知道这一情况。

**#pragma** 语句 **option** 允许你在程序中而不是在命令行上指定命令行选择项。它的一般形式:

```
#pragma option 选择项表
```

如果使用的是集成环境则无须使用此命令。

`saveregs` 指令防止一个函数改变任何 CPU 寄存器的值。你并不需要使用这个 `#pragma` 命令。

`warn` 指令使 Turbo C++ 抑制警告信息选择项。

它的形式是

`#pragma warn 设置`

这里设置(setting)是任一警告错误选择项。这些选择项请见 Turbo C++ 用户手册。

## 15.9 预定义的宏替换名

ANSI C 标准中指定了五个内部预定义宏替换名:

`__LINE__`  
`__FILE__`  
`__DATE__`  
`__TIME__`  
`__STDC__`

除了这些, Turbo C++ 还定义了下述宏替换名:

`__CDECL__`  
`__COMPACT__`  
`__HUGE__`  
`__LARGE__`  
`__MEDIUM__`  
`__MSDOS__`  
`__PASCAL__`  
`__SMALL__`  
`__TINY__`  
`__TURBOC__`  
`__cplusplus`  
`__OVERLAY__`

`__LINE__` 和 `__FILE__` 已在讨论 `#line` 指令时说明过了。这里介绍其它的宏替换名。

`__DATE__` 存储形式为月/日/年的一个字符串,它是原文件编译为目标文件时的日期。

一个源文件开始编译转换成目标文件的时间以一字符串形式存在 `__TIME__` 中。形式为小时:分钟:秒钟。

宏 `__STDC__` 当你用 ANSI Keywords Only 选项编译 C 语言程序时被定义为 1。否则,宏是不定义的。

当一个程序用覆盖编译时,宏 `__OVERLAY__` 被定义为 1,否则 `__OVERLAY__` 未定义。

宏 `__CDECL__` 是这样定义的:如果使用了标准 C 调用约定,即 Pascal 选择项没有使用时 `__CDECL__` 有定义,否则这个宏无定义。

对应于编译中定义的存储模式,下面几个宏中只有一个有定义:

`__TINY__`  
`__SMALL__`

**\_\_COMPACT\_\_**  
**\_\_MEDIUM\_\_**  
**\_\_LARGE\_\_**  
**\_\_HUGE\_\_**

当使用 MS-DOS 版 Turbo C 时, 宏 **\_\_MSDOS\_\_** 被定义为 1。

只有当使用 Pascal 调用约定来编译文件时, **\_\_PASCAL\_\_** 有定义; 否则它没有定义。

宏 **\_\_TURBOC\_\_** 保存所使用的 Turbo C/C++ 的版本号。它是一个十六进制常数。左边一位是主要修订版号, 右边两位是微变版号。例如, 数字 202 代表版本 2.02。

如果用户程序当作 C++ 程序编译, 则 **\_\_cplusplus** 有定义, 否则无定义。

下面这个例子说明了一些内部预定义宏名字的用法:

```
#include <stdio.h>

main(void)
{
    printf("%s %s %s %s\n", __FILE__, __LINE__, __DATE__,
        __TIME__);

    printf("Using version %X of Turbo C++.", __TURBOC__);
    return 0;
}
```

这里的大部分内部宏只用在较复杂的编程环境中, 如多个不同版本的程序——可能在不同类型的计算机上运行——的扩展和维护中。作为一个初级的 C++ 程序员, 记住这些宏的用途是很有好处的, 虽然有时并不需要用到它们中的任何一个。

### 第三部分 使用 Turbo C++ 的面向对象性质



## 第十六章 C++概述

C++是一面向对象(object oriented)的编程语言。在学习有关C++的专门知识前,应当先来理解面向对象程序设计中的基础理论。

### 16.1 什么是面向对象程序设计?

面向对象程序设计是一种编程的新方法。自从发明了计算机以来,程序设计的方法有了很大的变化。这种改变的基本原因是为了适应不断增长的程序复杂性。例如,在刚刚发明计算机时,程序设计是通过转换二进制机器指令来完成的。只要程序长度在几百条指令之内,这种方法就是可行的。随着程序的增大,发明了汇编语言以使程序员能用机器指令的符号表示来处理更大,且其复杂性不断增大的程序。由于程序进一步增大,高级语言就被引入以赋予程序员更多的处理复杂性的工具。第一个广为流行的高级语言无疑是FORTRAN。虽然FORTRAN是令人很难忘的第一步,但它却很少鼓励程序的清晰性与易于理解性。

到了60年代,结构化程序设计诞生了。这是为诸如C和Pascal这样的语言所鼓励的方法。正是使用了结构化语言,才第一次有可能很容易地写出比较复杂的程序。但是,即使是结构化编程方法,只要一工程达到某一定的大小,也对它无法控制。这是由于其复杂性超过了一程序员使用结构化编程技术所能处理的。在程序设计开发的每一里程碑上,方法被不断发明以使程序员能够对付不断增长的复杂性。在此过程中,每一新方法都保留了先前方法中的最好成份。今天,许多工程接近或已在结构化方法不能奏效的复杂性上。为解决该问题,发明了面向对象程序设计。

面向对象程序设计汲取了结构化程序设计中的最好思想,并且将它们与一些强有力的新概念融合起来。这种新概念鼓励以一种新的方式来看待程序设计任务。面向对象程序设计使你能够更容易地针将一问题分解成组成该问题的各相关部分的子模块。然后,使用该语言,就能将这些子模块翻译成被称为对象的自包含单元。

所有面向对象程序设计语言都有三个共同之处,即对象、多态性及继承。

#### 16.1.1 对象(object)

面向对象程序设计的最重要性质就是对象。简单地说,一对象就是既含数据又含操作(处理)该数据的代码的一逻辑实体。在一对象中,有些代码或数据是为该对象所有的,即不能为此对象之外的任何东西所直接存取。以此方式,对象就提供了防止程序中其它不相关成分无意修改或不正确使用此对象的私有部分的有效保护。将代码及数据以此方式联合在一起常称为封装(encapsulation)。

不论什么意图及目的,一个对象总是一个用户定义类型的变量。最初将联合了数据及代码的对象看成是一变量会有些奇怪。但是,在面向对象程序设计中,情形完全就是这样。定义了一对象也就是隐含地建立了一新的数据类型。

### 16.1.2 多态性(polymorphism)

面向对象程序设计语言支持多态性。本质上说,多态性意味着一个名字可为几个相关但多少有些不同的目的所使用。多态性的目的是允许一个名字能被用来说明行动的一通用类,取决于正在处理的是什么数据,此通用类的某一具体实例就被执行。例如,可能有一个定义了三种不同类型的栈的程序。一个栈用于态数值,另一个用于浮点数,还有一个用于 long 型值。由于多态性,可以为这些栈建立三组称为 push() 和 pop() 的函数,编译程序将根据调用函数时伴随着的是什么类型的数据而选择正确的例程。在本例中,通用的概念就是将数据从栈中推入、推出的概念。这些函数为每种类型的数据定义了怎样推入/推出的专门方法。

第一个面向对象程序设计语言是解释型的,所以多态性就是在运行时间时被支持的。但是,C++是一编译语言,所以在C++中,既支持运行时间的多态性,也支持编译时间的多态性。

### 16.1.3 继承(Inheritance)

继承是一对象获取另一对象之性质的过程。这一性质很重要,因为它支持分类概念。如果你思考一下,就会看到大多数知识之所以可以维护,正是由于其层次分类。例如,红香蕉苹果是苹果类的一部分,苹果又是水果类的一部分,水果又是食品类的一部分。如果不使用类,那么每个对象就都得定义其所有性质。但是,如果使用了类,一对象就只需定义使其区别于其类中其它对象的那些性质。它能继承那些与其更一般类共享的那些性质。正是这种继承机制才使得一对象能够成为一较一般类的实例。

## 16.2 C++的一些基本原则

由于C++是C的一个超集,所以C程序隐含地也就是C++程序(但在ANSI C和C++之间有几点细微差异,这将使得有一些但非常少的C程序不能为C++编译器所编译。这些差异将在后面讨论)。这意味着你能编写看上去就像C程序那样的C++程序。但是,如此做就如同用二档而驾车行驶在高速公路上:没有充分利用C++的长处。另外,虽然C++允许编写类似C的程序,大多数C++程序员却使用为C++所独有的一种风格和性质。由于编写看上去就像C程序的C++程序的重要性,在深入到C++细节之前,本节先来介绍有关的一些性质。

让我们以一例子作为开始。请看下面C++程序:

```
#include <iostream.h>

main(void)
{
    int i;
    char str[80];

    cout << "C++ is fun\n"; // this is a single line comment
    /* you can still use C style comments, too */

    printf("You can use printf() if you like\n");

    // input a number using >>
    cout << "enter a number: ";
    cin >> i;
```

```

// now, output a number using <<
cout << "100" << "\n";

// read a string
cout << "enter a string: ";
cin >> str;
// print it
cout << str;

return 0;
}

```

该程序与一般 C 程序看上去大不相同。首先，头文件 `iostream.h` 被包含进来。该文件是为支持 C++ 风格的 I/O 操作而定义的。

接着看上去不同的一行是：

```
cout << "C++ is fun\n"; /*this is a single line comment*/
```

该行引入了两个 C++ 的新性质。第一，语句

```
cout << "C++ is fun\n";
```

使得 `C++ is fun` (后跟一回车换行) 被显示在屏幕上。在 C++ 中，`<<` 的作用得到扩大。它仍然是左移操作符，但当用在本例中所示的地方时，它就是一输出操作符。字 `cout` 是与屏幕相联的一标识符 (实际上，C++ 也类似于 C，支持 I/O 重定向，但为便于讨论，可假定 `cout` 是指屏幕)。可用 `cout` 和 `<<` 来输出任一内部数据类型以及字符串。

需要指出的是，仍可使用 `printf()` (如程序所示) 或任何其它 C 的 I/O 函数。许多程序员只是感到使用 `cout <<` 更具有 C++ 的味道。

跟在输出表达式后面的是一条 C++ 注释。在 C++ 中，注释可以两种方式定义。第一，可用像 C 那样的注释，这在 C++ 中与在 C 中作用相同。但是，在 C++ 中，还可用 `//` 来定义单行注释。当用 `//` 来开始一注释时，其后的一切内容都被编译程序忽略直到行尾。一般来说，C++ 程序员在需用多行注释时，使用类似于 C 的注释，在只需一行注释时，使用 C++ 的单行注释。

其次，该程序提示用户输入一数。该数用下面的语句来从键盘上读入：

```
cin >> i;
```

在 C++ 中，`>>` 操作符仍然保持右移意义。但是，当如上使用时，它还使得 `i` 接受从键盘上输入的一值。标识符 `cin` 表示键盘。一般说来，可用 `cin >>` 来输入任何基本数据类型的一个变量。

虽然程序中没有表明，但可任意使用诸如 `scanf()` 这样的 C 输入函数，而不必用 `cin >>`。但是，如前所述，许多程序员感到 `cin >>` 更接近于 C++ 的风格。

程序中另一值得注意的行是。

```
cout << "your number is " << i << "\n";
```

这将使

```
your number is 100
```

被显示 (假如 `i` 的值为 100)，后跟一回车换行。一般说来，可使用你所期望的任意多个 `<<` 输出操作符。

该程序的其它部分表明怎样用 `cin >>` 和 `out <<` 来读写一串。

### 16.3 编译 C++ 程序

Turbo C++既可编译 C 程序,也可编译 C++程序。一般来说,如果一程序以.CPP 结束,它将作为一 C++程序而被编译。如果它以任何其它扩展名结束,则将作为一 C 程序而被编译。同此,最简单的使 Turbo C++将程序作为 C++程序而编译的方法就是使其扩展名为.CPP。

如果不想使 C++程序有.CPP 扩展名,就必须改变集成开发环境的缺省设置。为此,先选择 Options 菜单,然后是 Compile 选项;接着是 C++选项。这样,就使集成环境能将所有程序都当作 C++程序而编译。

### 16.4 类及对象的引入

既然已了解了 C++的一些约定及特殊性质,现在就该引入其最重要的性质了:类(class)。在 C++中,为创建一对象,必须首先用关键字 class 来定义其一般形式。一个类类似于一个结构。让我们以一个例子开始。该类定义了名为 queue 的一类型,它被用来创建一队列对象:

```
#include <iostream.h>

// this creates the class queue
class queue {
    int q[100];
    int sloc, rloc;
public:
    void init(void);
    void qput(int i);
    int qget(void);
};
```

现在我们仔细地考察此类声明。

一个类可以含私有部分,也可含公共部分。缺省时,所有在类中的项都是私有的。例如,变量 q, sloc 和 rloc 都为私有。这意味着它们不能为任何不是该类之成员的任何函数所存取。这是实现封装的一种方法——对数据的特定项的存取可通过使它们为私有的而严加控制。虽然本例中未加说明,但你还可定义私有函数,它们只可为该类的其它成员所调用。

为使一个类的某些部分成为公共的(即可为程序中其它部分所存取),必须在 public 关键字后对之加以说明。所有在 public 之后定义的变量或函数都可为程序中所有其它函数所存取。本质上说,用户程序的其它部分是通过其 public 函数和数据来存取一对象的。值得一提的是,虽然可有 public 变量,但原则上还是应该试图限制或取消对它们的使用。应使所有数据都为私有的,并且通过 public 函数来控制对它们的存取。注意 public 关键字之后跟着一冒号。

函数 init(), qput()和 qget()被称为成员函数(member functions),这是由于它们是 queue 类的一部分。记住一对象形成了在代码与数据之间的一种结合。只有在某类中所声明的函数才能存取该类的私有部分。这些函数就叫作成员函数。

一旦定义了一个类,就可以用该类名来建立一个对象。本质上说,类名成为一个新的数据类型说明符。例如,下个语句就建立了一个名为 Intqueue,类型为 queue 的对象:

```
queue intqueue;
```

还可以在定义类时, 通过将对象名放在闭花括号之后来创建对象, 正如对结构那样。总之, 在 C++ 中, 一个类建立了一个新的数据类型, 利用该数据类型, 就可创建该类的对象。

类说明的一般形式是

```
class 类名{  
    私有数据及函数  
public:  
    公共数据及函数  
} 对象表;
```

当然, “对象表”可为空。

在 `queue` 说明内部, 使用了成员函数的原型。必须记住在 C++ 中, 当需要告诉编译某一函数时, 必须使用它的完整原型式子。C++ 不支持原先传统的函数说明方法(此外, 在 C++ 中, 所有函数都必须是原型化的。原型在 C++ 中不是可选的)。

当到了实际编码为一类的成员函数时, 必须告诉编译程序该函数所属的哪个类, 这是通过用该函数所属的类名来修饰该函数名而完成的。例如, 下面就是编码 `qput()` 函数的一种方法。

```
void queue::qput(int i)  
{  
    if(sloc==100) {  
        cout << "queue is full";  
        return;  
    }  
    sloc++;  
    q[sloc] = i;  
}
```

`::`被称为作用域分辨操作符(scope resolution operator)。它告诉编译程序此版本的 `qput()` 属于 `queue` 类;或者说, 此 `qput()` 在 `queue` 域中。如你不久将看到的, C++ 中若干不同的类可使用相同的函数名。由于使用了作用域分辨操作符及类名, 编译程序能知道哪个函数属于哪个类。

为在不为某类一部分的程序中调用该类的一成员函数, 必须使用对象名及句点操作符。例如, 下面语句为对象 `a` 调用了 `init()`:

```
queue a,b;  
a.init();
```

理解 `a` 和 `b` 是两个独立的对象这一点非常重要。这意味着, 例如, 初始化 `a` 并不会使 `b` 也被初始化。`a` 和 `b` 的唯一关系就是它们为同一类型的两个对象。

另外一个重要之处是, 一个成员函数可以直接调用另一个成员函数, 而不必使用句点操作符。只有当一成员函数为不属于该类的代码所调用时, 才必须使用变量名及句点操作符。

下面的程序综合了前述的所有片断并补充了未说明的细节, 它说明了 `queue` 类。

```
#include <iostream.h>  
  
//this creates the class queue  
class queue {  
    int q[100];  
    int sloc, rloc;
```

```

public:
    void init(void);
    void qput(int i);
    int qget(void);
};

void queue::init(void)
{
    rloc = sloc = 0;
}

void queue::qput(int i)
{
    if(sloc==100) {
        cout << "queue is full";
        return;
    }
    sloc++;
    q[sloc] = i;
}

int queue::qget(void)
{
    if(rloc == sloc) {
        cout << "queue underflow";
        return 0;
    }
    rloc++;
    return q[rloc];
}

main(void)
{
    queue a, b; // create two queue objects
    a.init();
    b.init();

    a.qput(10);
    b.qput(19);

    a.qput(20);
    b.qput(1);

    cout << a.qget() << " ";
    cout << a.qget() << " ";
    cout << b.qget() << " ";
    cout << b.qget() << "\n";

    return 0;
}

```

一定要记住，一个对象的私有部分只可为作为该对象成员的函数所存取。例如，语句 `a.rloc=0;`

不能在先前程序的 `main()` 函数中。

注意，由于约定，大多数 C 程序将 `main()` 函数作为程序中的第一个函数。但是在 `queue` 程序中，`queue` 的成员函数被定义在 `main()` 函数之前。虽然 C++ 没有要求一定这样做(它们可定义在程序中的任何地方)，但这是编写 C++ 代码的最常见的方法。本书就遵循这一约定(实际上，在实际应用中，与一程序相关的类通常被包含在一个头文件中)。

## 16.5 函数重载

C++ 实现多态性的一种方法是通过使用函数重载(function overloading)。在 C++ 中，两个或更多的函数可以共享同一名字，只要其参数说明是不同的。在此情况下，共享同名的函数被称为重载，此过程也就是函数重载。例如，考虑如下程序：

```

#include <iostream.h>

// sqr_it is overloaded three ways
int sqr_it(int i);

```

```

double sqr_it(double d);
long  sqr_it(long l);

main(void)
{
    cout << sqr_it(10) << "\n";
    cout << sqr_it(11.0) << "\n";
    cout << sqr_it(9L) << "\n";

    return 0;
}

int  sqr_it(int i)
{
    cout << "Inside the sqr_it() function that uses ";
    cout << "an integer argument.\n";

    return i*i;
}

double  sqr_it(double d)
{
    cout << "Inside the sqr_it() function that uses ";
    cout << "a double argument.\n";

    return d*d;
}

long  sqr_it(long l)
{
    cout << "Inside the sqr_it() function that uses ";
    cout << "a long argument.\n";

    return l*l;
}

```

该程序建立了三个名为 `sqr_it()` 的类似但不完全相同的函数，其中每个函数都返回其变元的平方值。如程序所示，编译程序由于变元类型而知道何时用哪一函数。重载函数的意义在于它允许只使用一个名字就来存取相关的函数集。它可让你为某一操作建立起一通用名，而让编译程序来解决实际上用来完成该操作的到底应是哪一个函数的工作。

函数重载之重要的一个原因在于它有助于处理复杂性问题。为了理解为什么会如此，请考虑这个例子。大多数 C 编译程序在其标准库中都含类似于 `atoi()`、`atof()` 和 `atol()` 这样的函数。这些函数分别将一个数字串转换成一个整数、一个 `double` 数及一个 `long` 数的内部格式。即使这些函数完成几乎相同的功能，在 C 中还是必须使用三个多少有些差别的名词来表示各个任务，这就在概念上使得情形比起实际情况更为复杂。虽然各函数的内部概念是相同的，但对程序员来说，却得记住三件事，而不只是一件。但是，在 C++ 中，就有可能为三个函数用诸如 `atoum()` 这样的相同名字。这样，名 `atoum()` 就表示了要被完成的一般行为。在一特定情形下，由编译程序去选择正确的特殊版本。因此，程序员只须记住所要完成的一般行为。所以说，通过使用多态性，三件需被记住的事就归结成了一件事。尽管本例相当平凡，但如果扩展这里的概念，就会看到多态性有助于理解非常复杂的程序。

函数重载的一个更为实际的例子在下列程序中说明。如你所知，C(及 C++) 不含任何用来提示用户进行输入然后等待用户反应的库函数。该程序就建立了三个名为 `prompt()` 的函数，它为 `int`、`double` 和 `long` 型数据完成上述任务：

```
#include <iostream.h>

void prompt(char *str, int *i);
void prompt(char *str, double *d);
void prompt(char *str, long *l);

main(void)
{
    int i;
    double d;
    long l;

    prompt("Enter an integer: ", &i);
    prompt("Enter a double: ", &d);

    prompt("Enter a long: ", &l);

    cout << i << " " << d << " " << l;

    return 0;
}

void prompt(char *str, int *i)
{
    cout << str;
    cin >> *i;
}

void prompt(char *str, double *d)
{
    cout << str;
    cin >> *d;
}

void prompt(char *str, long *l)
{
    cout << str;
    cin >> *l;
}
```

一定要小心，你可以用相同的名字来重载不相关的函数，但不应如此。例如，可以用名 `sqr_int()` 来建立返回一整数平方值的函数，以及返回一双精度数的平方根的函数。但是，这两个操作本质上是不相同的，而以这种方式来应用函数重载就会破坏函数重载的整个目的。在实际中，你只应对密切相关的操作使用重载。

## 16.6 操作符重载

C++ 实现多态性的另一种方法是通过使用操作符重载。在 C++ 中，可以使用 `<<` 和 `>>` 操作符来完成控制台 I/O 操作。之所以能这样做的原因是在 `IOSTREAM.H` 头文件中，这些操作符被重载了。一操作符被重载就说明它具有了与一特定类相关的另外意义。但是，它仍然保留了其原先意义。

一般说来，可以通过定义任何 C++ 操作符某一专门类的含意来重载此操作符。例如，



再考虑一下在本章前面所开发的 `queue` 类。有可能重载 + 操作符以使它与 `queue` 类型的对象有关，从而能使它能将某一栈的内容加到另一栈的内容上。但是，对其它类型的数据来说，+ 仍保留了其原先意义。在开发实际例子之前，还得对 C++ 有更多的了解，但这里已给出了一般思想。

## 16.7 再谈继承

如前所述，继承是一面向对象程序设计语言的主要特点之一。在 C++ 中，继承是通过允许某一类能在其声明中包含另一类而得以支持的。为了解具体含意，请看一个例子。这里有一个名为 `road_vehicle` 的类，它在大体上定义了行驶在公路上的交通工具。它存储了一交通工具所能有的轮子数及它所能运载的乘客数：

```
class road_vehicle {
    int wheels;
    int passengers;
public:
    void set_wheels(int num);
    int get_wheels(void);
    void set_pass(int num);
    int get_pass(void);
};
```

这个对公路上交通工具的大略定义可被用来帮助定义特定对象。例如，如下代码利用 `road_vehicle` 说明了一个名为 `truck` 的类：

```
class truck : public road_vehicle {
    int cargo;
public:
    void set_cargo(int size);
    int get_cargo(void);
    void show(void);
};
```

注意 `road_vehicle` 是怎样被继承的。继承的一般形式是

`class 新类名: <access> 被继承的类 {...`

这里，`access` 是可选的。但若有的话，它必须是 `public`，`private` 或 `protected` 中的一个。在后面的章节中将进一步讨论这些选项。目前来说，所有被继承的类都使用 `public`。使用 `public` 就意味着其祖先的所有 `public` 元素对继承它的类也同样是 `public` 的。

因此，`truck` 类的成员就可以存取 `road_vehicle` 的成员函数，就仿佛这些成员函数是在 `truck` 中被声明的。但是，`truck` 类的成员函数不能存取 `road_vehicle` 的私有部分。下面就是说明继承的一个程序。它利用继承建立了 `road_vehicle` 的两个子类。一个是 `truck`，另一个是 `automobile`。

```
#include <iostream.h>

class road_vehicle {
    int wheels;
    int passengers;
public:
    void set_wheels(int num);
    int get_wheels(void);
    void set_pass(int num);
    int get_pass(void);
};
```

```

class truck : public road_vehicle {
    int cargo;
public:
    void set_cargo(int size);

    int get_cargo(void);
    void show(void);
};

enum type {car, van, wagon};

class automobile : public road_vehicle {
    enum type car_type;
public:
    void set_type(enum type t);
    enum type get_type(void);
    void show(void);
};

void road_vehicle::set_wheels(int num)
{
    wheels = num;
}

int road_vehicle::get_wheels(void)
{
    return wheels;
}

void road_vehicle::set_pass(int num)
{
    passengers = num;
}

int road_vehicle::get_pass(void)
{
    return passengers;
}

void truck::set_cargo(int num)
{
    cargo = num;
}

int truck::get_cargo(void)
{
    return cargo;
}

void truck::show(void)
{
    cout << "wheels: " << get_wheels() << "\n";
    cout << "passengers: " << get_pass() << "\n";
    cout << "cargo capacity in cubic feet: " << cargo << "\n";
}

void automobile::set_type(enum type t)
{
    car_type = t;
}

```

```

enum type automobile::get_type(void)
{
    return car_type;
}

void automobile::show(void)
{
    cout << "wheels: " << get_wheels() << "\n";
    cout << "passengers: " << get_pass() << "\n";
    cout << "type: ";
    switch(get_type()) {
        case van: cout << "van\n";
            break;
        case car: cout << "car\n";
            break;
        case wagon: cout << "wagon\n";
    }
}

main(void)
{
    truck t1, t2;
    automobile c;

    t1.set_wheels(18);
    t1.set_pass(2);
    t1.set_cargo(3200);

    t2.set_wheels(6);
    t2.set_pass(3);
    t2.set_cargo(1200);

    t1.show();
    t2.show();

    c.set_wheels(4);
    c.set_pass(6);
    c.set_type(van);

    c.show();

    return 0;
}

```

如此程序所示，继承的主要优点是能建立一个能被融入更专门类别的基类别。这样，各个对象就能准确地表示出其自己的类别。

注意 `truck` 和 `automobile` 都含有名为 `show()` 的成员函数，该函数显示有关各个对象的信息。这是多态性的另一方面。由于各个 `show()` 都与其自己的类相联，编译程序在任何情形下就都能容易地识别出该调用哪一个。

## 16.8 构造函数与析构函数

在使用一对象的某一部分之前往往需要对它进行初始化。例如，重新考虑一下前面开发的 `queue` 类。在使用该队列前，变量 `rloc` 和 `sloc` 必须被置为零。这是利用函数 `init()` 完成的。由于对初始化的要求非常普遍，C++ 允许对象在被创建时对其自身进行初始化。这种自动初始化是通过使用构造(constructor)函数完成的。

构造函数是一特殊函数，它是某类的一个成员函数并且与该类有相同的名字。例如，下面就是在用一构造函数对 `queue` 类进行初始化时 `queue` 类所变成的样子：

```

// this creates the class queue
class queue {
    int q[100];
    int sloc, rloc;
public:
    queue(void); // constructor
    void qput(int i);
    int qget(void);
};

```

注意构造函数 `queue()` 没有说明返回类型。在 C++ 中，构造函数不能返回值。

`queue()` 函数的代码如下：

```

// This is the constructor function.
queue::queue(void)
{
    sloc = rloc = 0;
    cout << "queue initialized\n";
}

```

记住信息 `queue initialized` 是作为说明该构造函数的一种方法而输出的。在实际情形中，大多数构造函数不会输出或输入任何东西。

在创建一对象时调用该对象的构造函数。这意味着是在执行该对象的声明同时将调用它。此外，对局部对象来说，每次遇到此对象的声明时都调用该构造函数。

与构造函数相对的是析构函数(destructor)。在许多情形中，一对象在被破坏时都需要完成某些行动(记住局部对象是在进入它们的块时被创建，而在退出块时被破坏的)。例如，一对象可能需要释放原先分配给它的内存。在 C++ 中，是通过析构函数来处理还原问题。析构函数与构造函数名字相同，但它前面必须有一个 ~ 号。下面是 `queue` 类及其构造函数和析构函数(记住 `queue` 类不需要一个析构函数，这里给出的目的只是为了说明)。

```

// this creates the class queue
class queue {
    int q[100];
    int sloc, rloc;
public:
    queue(void); // constructor
    ~queue(void); // destructor
    void qput(int i);
    int qget(void);
};

// This is the constructor function.
queue::queue(void)
{
    sloc = rloc = 0;
    cout << "queue initialized\n";
}

// This is the destructor function.
queue::~queue(void)
{
    cout << "queue destroyed\n";
}

```

为了解构造函数和析构函数是如何工作的，请看以下一个例子：

```
#include <iostream.h>

// this creates the class queue
class queue {
    int q[100];
    int sloc, rloc;
public:
    queue(void); // constructor
    ~queue(void); // destructor
    void qput(int i);
    int qget(void);
};

// This is the constructor function.
queue::queue(void)
{
    sloc = rloc = 0;
    cout << "queue initialized\n";
}

// This is the destructor function
queue::~~queue(void)
{
    cout << "queue destroyed\n";
}

void queue::qput(int i)
{
    if(sloc==100) {
        cout << "queue is full";
        return;
    }
    sloc++;
    q[sloc] = i;
}

int queue::qget(void)
{
    if(rloc == sloc) {
        cout << "queue underflow";
        return 0;
    }
    rloc++;
    return q[rloc];
}

main(void)
{
    queue a, b; // create two queue objects

    a.qput(10);
    b.qput(19);

    a.qput(20);
    b.qput(1);

    cout << a.qget() << " ";
    cout << a.qget() << " ";
    cout << b.qget() << " ";
    cout << b.qget() << "\n";
}
```

```
return 0;  
}
```

该程序显示下列信息:

```
queue initialized  
queue initialized  
10 20 19 1  
queue destroyed  
queue destroyed
```

## 16.9 C++关键字

除了由 C 语言所定义的关键字及为 Turbo C 本身所特有的关键字而外, C++ 还增加了下列关键字:

```
asm  
catch  
class  
delete  
friend  
inline  
new  
operator  
private  
protected  
public  
template  
this  
virtual
```

其中, `catch` 和 `template` 被保留作将来使用。不能将任何一个关键字用作变量名或函数名。

既然 C++ 主要性质中的许多已被介绍, 现在就可以更详细地来考察 C++ 了。

## 第十七章 对类的进一步考察

如上一章所述, 类是 C++ 的最重要性质。本章就来进一步对它及有关问题进行考察。

### 17.1 参数化的构造函数

在建立一对象时, 常常希望或必须用具体值来初始化各种数据元素。如你在上一章中所见, 通过使用构造函数就可以做到这一点。但在 C++ 中, 对象初始化的概念被扩充成了用程序员定义的值而初始化具体的对象。这是通过将变元传递给一对象的构造函数而完成的。作为一个简单例子, 可以改进上一章末尾描述的 `queue` 类以使它能接受作为该队列的标识号的一变元。首先, `queue` 被改成如下形式:

```
// this creates the class queue
class queue {
    int q[100];
    int sloc, rloc;
    int who; // holds the queue's ID number
public:
    queue(int id); // constructor
    ~queue(void); // destructor
    void qput(int i);
    int qget(void);
};
```

变量 `who` 被用来存放说明队列的标识号。其实际值将由在建立类型为 `queue` 的一对象时, 以 `id` 传递给构造函数的值来决定。`queue()` 构造函数形式如下:

```
// This is the constructor function.
queue::queue(int id)
{
    sloc = rloc = 0;
    who = id;
    cout << "queue " << who << " initialized\n";
}
```

为将一变元传递给构造函数, 必须在正说明一对象时将要被传递的值与此对象相联。C++ 支持两种实现上述功能的方式。第一种方法的说明如下:

```
queue a=queue(101);
```

该说明创建一个名为 `a` 的队, 并将值 `101` 传递给它。但是, 由于第二种方法——有时也称作简写 (*shorthand*) 方法——较短且更切中关键, 上述形式很少使用。在简写方法中, 变元必须跟在对象名后并用括号括起。例如, 下述语句的功能与上述声明的功能相同:

```
queue a(101)
```

由于简写方法几乎为所有 C++ 程序员所使用, 本书就只用简写形式。将变元传递给构造函数的一般形式为:

```
class 类型 变量(变元表)
```

这里, “变元表” 是被传递到构造函数的用逗号隔开的一组变元。

队列程序的下列版本说明了将变元传递给构造函数的方法:

```

#include <iostream.h>

// this creates the class queue
class queue {
    int q[100];
    int sloc, rloc;
    int who; // holds the queue's ID number
public:
    queue(int id); // constructor
    ~queue(void); // destructor
    void qput(int i);
    int qget(void);
};

// This is the constructor function.
queue::queue(int id)
{
    sloc = rloc = 0;
    who = id;
    cout << "queue " << who << " initialized\n";
}

// This is the destructor function.
queue::~queue(void)
{
    cout << "queue " << who << " destroyed\n";
}

void queue::qput(int i)
{
    if(sloc==100) {
        cout << "queue is full";
        return;
    }
    sloc++;
    q[sloc] = i;
}

int queue::qget(void)
{
    if(rloc == sloc) {
        cout << "queue underflow";
        return 0;
    }
    rloc++;
    return q[rloc];
}

main(void)
{
    queue a(1), b(2); // create two queue objects

    a.qput(10);
    b.qput(19);

    a.qput(20);
    b.qput(1);

    cout << a.qget() << " ";
    cout << a.qget() << " ";
    cout << b.qget() << " ";
    cout << b.qget() << "\n";
}

```



```

        return 0;
    }

```

该程序产生如下输出:

```

queue 1 initialized
queue 2 initialized
10 20 19 1
queue 2 destroyed
queue 1 destroyed

```

与 **a** 相联的队的标识号为 1, 与 **b** 相联的为 2。

虽然此例子在建立一对象时只传递一个变元, 但传递几个变元当然也是可能的。例如, 下面程序中 **widget** 类型的对象接受了两个值:

```

#include <iostream.h>

class widget {
    int i;
    int j;
public:
    widget(int a, int b);
    void put_widget(void);
};

widget::widget(int a, int b)
{
    i = a;
    j = b;
}

void widget::put_widget(void)
{
    cout << i << " " << j << "\n";
}

main(void)
{
    widget x(10, 20), y(0, 0);

    x.put_widget();
    y.put_widget();

    return 0;
}

```

程序显示:

```

10 20
0 0

```

## 17.2 友元函数

对一类的非成员函数来说, 有可能通过将它声明成该类的一个友元(**friend**)来存取该类的私有部分。例如, 下例中 **frd()** 被声明成本 **cl** 类的一个友元。

```

class cl {
    .
    .
    .

```

```

public:
    friend void frd(void);
    .
    .
};

```

关键字 **friend** 必须在整个函数说明之前。

虽然技术上并非必须，但是 C++ 允许友元函数的原因是为了适应两个类出于效率的考虑必须共享同一函数这一情形。作为一个例子，考虑一定义了两个名为 **line** 和 **box** 的类的程序。类 **line** 含从所说明的 X, Y 坐标开始，用指定颜色画任意长的水平破折线所需要的全部数据与代码。类 **box** 含用指定颜色在所说明的左上角和右下角坐标上画一方框所需要的全部代码与数据。这两个类都用到了 **same\_color()** 函数来确定是否以同样的颜色来画出线和框。这些类的说明如下：

```

class line;

class box {
    int color; // color of box
    int upx, upy; // upper left corner
    int lowx, lowy; // lower right corner
public:
    friend int same_color(line l, box b);
    void set_color(int c);
    void define_box(int xl, int yl, int x2, int y2);
    void show_box(void);
};

class line {
    int color;
    int startx, starty;
    int len;
public:
    friend int same_color(line l, box b);
    void set_color(int c);
    void define_line(int x, int y, int l);
    void show_line();
};

```

**same\_color()** 函数既非 **line** 的成员，也非 **box** 的成员，但却是 **line** 和 **box** 的友元。它在作为其参元的 **line** 对象和 **box** 对象以相同颜色画出时返回真；否则返回零。**same\_color()** 函数的内容如下：

```

// return true if line and bdx have same color.
int same_color(line l, box b)
{
    if(l.color==b.color) return 1;
    return 0;
}

```

**same\_color()** 函数为有效地完成其任务，需对 **line** 以及 **box** 的私有部分加以存取(记住，可以建立公共的接口函数以返回 **line** 和 **box** 的颜色，而任何函数都可对这两颜色加以比较。但是，这种方法需要增加函数调用，这有时会引起效率很低)。

注意在类说明开始的 **line** 的空声明。由于 **box** 中的 **same\_color()** 在 **line** 被声明之前引用 **line**，所以 **line** 必须被向前引用。如果不这么做，那么编译程序在 **box** 声明中遇到 **line**

时就无法知道是什么。在 C++ 中，对一类的向前引用只是关键字 `class` 后跟着该类的类型名。通常，需要向前引用的唯一时刻是当涉及到友元函数时。

下面的程序说明了类 `line` 和类 `box` 以及一友元函数怎样能够存取一类的私有部分。它利用了各种 Turbo C++ 的屏幕函数：

```
#include <iostream.h>
#include <conio.h>

class line;

class box {
    int color; // color of box
    int upx, upy; // upper left corner
    int lowx, lowy; // lower right corner
public:
    friend int same_color(line l, box b);
    void set_color(int c);
    void define_box(int x1, int y1, int x2, int y2);
    void show_box(void);
};

class line {
    int color;
    int startx, starty;
    int len;
public:
    friend int same_color(line l, box b);
    void set_color(int c);
    void define_line(int x, int y, int l);
    void show_line();
};

// return true if line and box have same color.
int same_color(line l, box b)
{
    if(l.color==b.color) return 1;
    return 0;
}

void box::set_color(int c)
{
    color = c;
}

void line::set_color(int c)
{
    color = c;
}

void box::define_box(int x1, int y1, int x2, int y2)
{
    upx = x1;
    upy = y1;
    lowx = x2;
    lowy = y2;
}

void box::show_box(void)
{
    int i;
```

```

    textcolor(color);

    gotoxy(upx, upy);
    for(i=upx; i<=lowx; i++) cprintf("-");

    gotoxy(upx, lowy-1);
    for(i=upx; i<=lowx; i++) cprintf("-");

    gotoxy(upx, upy);
    for(i=upy; i<=lowy; i++) {
        cprintf("|");
        gotoxy(upx, i);
    }

    gotoxy(lowx, upy);
    for(i=upy; i<=lowy; i++) {
        cprintf("|");
        gotoxy(lowx, i);
    }
}

void line::define_line(int x, int y, int l)
{
    startx = x;
    starty = y;
    len = l;
}

void line::show_line(void)
{
    int i;

    textcolor(color);

    gotoxy(startx, starty);

    for(i=0; i<len; i++) cprintf("-");
}

main(void)
{
    box b;
    line l;

    b.define_box(10, 10, 15, 15);
    b.set_color(3);
    b.show_box();

    l.define_line(2, 2, 10);
    l.set_color(2);
    l.show_line();

    if(!same_color(l, b)) cout << "not the same";
    cout << "\npress a key";
    getch();

    // now, make line and box the same color
    l.define_line(2, 2, 10);
    l.set_color(3);
    l.show_line();

    if(same_color(l, b)) cout << "are the same color";

    return 0;
}

```

### 17.3 缺省函数变元

在调用一函数时, 当与一参数相应的变元没有被说明时, C++ 允许该函数将一缺省值赋予参数。此缺省值以一种语法上类似于变量初始化的方式而被说明。下例将 `f()` 声明成一整数变量, 并声明一缺省值 1:

```
void f(int i=1)
{
    .
    .
    .
}
```

现在 `f()` 可以以如下例子所说明的两种方式之一而被调用:

```
f(10); //传递一个显式的值
f();   //使函数使用缺省值
```

第一个调用将值 10 传递给 `i`。第二个调用自动给 `i` 缺省值 1。

缺省变元被包含在 C++ 中的原因在于它们提供了使程序员能处理更大的复杂性的另一种方法。为了对付尽可能广泛的情形, 一函数常常含比其最常用方法所需要的参数更多的参数。因此, 当运用缺省变元时, 只需要记住并说明对具体的——而不是最为一般情形有意义的变元。

为理解缺省变元的原因, 让我们开发一个实际的例子。下面是一个没有包含在 Turbo C++ 库中的名为 `xyout()` 的有用函数:

```
// Output a string at specified X,Y location.
void xyout(char *str, int x = -1, int y = -1)
{
    if(x== -1) x = wherex();
    if(y== -1) y = wherey();
    gotoxy(x, y);
    cout << str;
}
```

该函数以正文方式从 `x` 和 `y` 所定义的位置处开始显示由 `str` 所指向的串。如果 `x` 和 `y` 都未被说明, 那么该串就在当前正文方式的 `x`, `y` 位置处被输出。函数 `wherex()`, `wherey()` 和 `gotoxy()` 都是 Turbo C++ 库的一部分。`wherex()` 和 `wherey()` 函数分别返回当前的 `x` 和 `y` 坐标。当前 `x`, `y` 坐标是下一输出操作开始的地方。`gotoxy()` 函数将光标移到所说明的 `x`, `y` 位置(如果尚未读过第十四章, 可以回过头去看看那里有关 Turbo C++ 屏幕控制函数的讨论)。

下面的简短程序说明了 `xyout()` 的用途:

```
#include <iostream.h>
#include <conio.h>

void xyout(char *str, int x = -1, int y = -1);
```

```

main(void)
{
    xyout("hello", 10, 10);
    xyout(" there");
    xyout("I like C++", 40); // this is still on line 10

    xyout("This is on line 11.\n", 1, 11);
    xyout("This follows on line 12.\n");
    xyout("This follows on line 13.");

    return 0;
}

void xyout(char *str, int x = -1, int y = -1)
{
    if(x == -1) x = wherex();
    if(y == -1) y = wherey();
    gotoxy(x, y);
    cout << str;
}

```

该程序产生类似于图 17-1 所示的输出。如该程序所示,虽然有时说明将要显示的正文的准确位置是有用的,但常常只需从上次输出发生的地方继续就可以了。通过使用缺省变元,可以使用同一函数来完成两个目标——没必要用两个单独的函数。

```

      hello there                      I like C++
This is on line 11.
This follows on line 12.
This follows on the third line 13.

```

图 17-1 程序 xyout 的输出

注意在 main() 中, xyout() 被以三个、两个或一个变元来调用。当只用一个变元调用时, X 和 Y 都是缺省的,但是,当用两个变元调用时,只有 Y 是缺省的。不能缺省 X 而说明 Y 来调用 xyout()。更一般地,当调用一函数时,所有变元都以从左到右的次序与它们相应的参数相匹配。一旦所有现存变元被匹配,就使用任何剩余的缺省变元。

所有取缺省值的参数都必须出现在不取缺省值的参数的右边。也就是说,一旦开始定义取缺省值的参数,就不可以再说明一个非缺省的参数。例如,如下定义 xyout() 是不正确的:

```

//wrong!
void xyout(int x=-1,int y=-1,char *str)

```

如下也是不正确地试图使用缺省参数的例子。

```

//wrong!
int f(int i,int j=10,int k)

```

一旦开始缺省参数,在表中就不可再出现非缺省的参数。

还可在一对象的构造函数中使用缺省参数。例如，下面是本章前面给出的 `queue()` 构造函数的一个不太相同的版本：

```
// This is the constructor function that uses
// a default value.
queue::queue(int id=0)
{
    sloc = rloc = 0;
    who = id;
    cout << "queue " << who << " initialized\n";
}
```

在该版本中，如果一个对象不带任何初始化值地被声明，那么 `id` 缺省为 0。例如，

```
queue a ,b(2);
```

建立 `a` 和 `b` 两个对象。这里，`a` 有一 `who` 的为 0 的标识值，`b` 的值为 2。

#### 17.4 正确地使用缺省变元

虽然在正确使用时，缺省变元可以是一非常强有力的工具，但在不正确使用时，它们却会产生麻烦。因为缺省变元的全部要点就是允许一函数能以一种有效而易于使用但仍保持很大灵活性的方式来完成其任务。为完成此目的，所有的缺省变元都应表示在大多数时候使用函数的方式，例如，如果在 90% 的时间内，一参数将含相同的值，那么一缺省变元就有意义。但是如果一值只在调用的 10% 中出现，而其余时间中与该参数相应的变元会发生很大变化，那么使用一缺省变元就可能不是一个好思想。

缺省变元的关键是它们的值为程序员通常将与一给定函数相联系的值。当没有通常与一参数相联系的值时，就没有理由使用缺省变元。事实上，在没有充分基础时声明缺省变元会破坏代码的结构——因为它会使任何阅读用户程序的人都感迷惑。在 10% 到 90% 之间，是否使用缺省变元当然是主观的。但 51% 看来是一个合理的转折点。

#### 17.5 类与结构之相关性

比起 C 的对应物来说，C++ 中的结构具有一些扩充的能力。在 C++ 中，类与结构是密切相关的，事实上，除了一个例外，它们是可以互相交换的。因为 C++ 的结构也能以与类相同的方式包括数据及对该数据操作的代码。C++ 的结构和类的差别在于缺省时类的成员是 `private` 的，而结构的成员是 `public` 的。除此而外，结构和类完成完全相同的功能。例如，考虑下列程序：

```
#include <iostream.h>

struct cl {
    int get_i(void); // these are public
    void put_i(int j); // by default
private:
    int i;
};

int cl::get_i(void)
{
    return i;
}
```

```

void cl::put_i(int j)
{
    i = j;
}

main(void)
{
    cl s;

    s.put_i(10);
    cout << s.get_i();

    return 0;
}

```

该简单程序定义了一名为 `cl` 的结构类型，这里，`get_i()` 和 `put_i()` 是公共的，而 `i` 是私有的。注意，结构使用关键字 `private` 来引入该结构的私有元素。

下列程序说明使用类而不是结构的等价程序：

```

#include <iostream.h>

class cl {
    int i; // private by default
public:
    int get_i(void);
    void put_i(int j);
};

int cl::get_i(void)
{
    return i;
}

void cl::put_i(int j)
{
    i = j;
}

main(void)
{
    cl s;

    s.put_i(10);
    cout << s.get_i();

    return 0;
}

```

大部分时间中，C++ 程序员都用一类来定义对象的形式。而以与 C 相同的方式使用结构，但是，你也会看到使用结构的扩充能力的 C++ 代码。

## 17.6 联合与类之相关性

在 C++ 中，一个联合本质上是一个其中所有元素都存储在同一位置上的结构。联合可含构造函数和析构函数、成员与友元函数。例如，下面程序使用一联合来显示组成一整数(假定为双字节)的低阶与高阶字节的字符：



```

#include <iostream.h>

union u_type {

    u_type(int a); // public by default
    void showchars(void);
    int i;
    char ch[2];
};

// constructor
u_type::u_type(int a)
{
    i = a;
}

// show the characters that comprise an int
void u_type::showchars(void)
{
    cout << ch[0] << " ";
    cout << ch[1] << "\n";
}

main(void)
{
    u_type u(1000);

    u.showchars();

    return 0;
}

```

由于联合类似于结构，所以其成员缺省时都为 `public`。事实上，关键字 `private` 不能作用于一联合(本章后面将说明关键字 `protected` 也不能作用于一联合)。

记住，只是由于 C++ 赋予联合以更大能力及灵活性并不意味着你就必须以这种方式来使用它们。在只需要一 C 式的联合时，可以以那种方式任意使用它。但是，在想要将一联合与对这处理的例程进行封装时，就可能会在程序中增加大量的结构。

## 17.7 内部函数

C++ 含有一个 C 中没有的非常重要的性质，虽然该性质不是面向对象程序设计所特有的。这就是内部函数(`inline function`)。内部函数在被调用时不是被实际调用而是被扩展在行中。这好比 C 中带参数的类似于函数的宏，但却更灵活。建立一内部函数的方式有两种：第一种是使用 `inline` 修饰符。例如，为创建一名为 `f` 的返回一整型值，且无参数的内部函数，必须这样对之加以声明：

```

inline int f(void)
{
    .
    .
    .
}

```

`inline` 的一般形式为：

`inline`    函数声明

`inline` 修饰符在一函数声明的所有其它部分之前。

**inline** 函数的引入是为了提高效率。每次调用一函数时，都必须执行一系列指令以建立函数调用。包括将所有变元压到栈上，以及从函数返回。有时，需用许多个 CPU 周期来完成这些过程。当一函数被扩展在行中时，就不存在这些额外开销，而程序的整个速度也会提高了，但是，在 **inline** 函数很大时，程序的整个大小也增大了，因此 最好的 **inline** 函数是那些很小的函数，较大的函数应保留作普通函数。

作为一个例子。下面程序利用了 **inline** 来使前一节中的程序效率更高：

```
#include <iostream.h>

class cl {
    int i; // private by default
public:
    int get_i(void);
    void put_i(int j);
};

inline int cl::get_i(void)
{
    return i;
}

inline void cl::put_i(int j)
{
    i = j;
}

main(void)
{
    cl s;

    s.put_i(10);
    cout << s.get_i();

    return 0;
}
```

如果编译此程序，并将它与先前程序的一编译版本作比较。那么此 **inline** 版本将会少几个字节。

从技术上来说，**inline** 是一请求，而不是一命令，它让编译命令产生内部代码。这里讨论最常见中的一些。对返回值的函数来说，如果存在一个循环，一个 **switch**、或一个 **goto** 语句，那么编译器就不会产生内部代码。对无返回值的函数来说，如果存在一 **return** 语句，那么就不会产生内部代码。不可有 **inline** 递归函数，也不可建立含 **static** 变量的 **inline** 函数。

#### 17.7.1 在类中建立一个内部函数

在 C++ 中，存在着建立一 **inline** 函数的另一方法。即定义在一类内部的函数代码。任何定义在一类内部的函数都自动地被作为一个 **inline** 函数。这里不需要用关键字 **inline** 来加在其声明前。例如，先前的程序可改写成。

```
#include <iostream.h>

class cl {
    int i; // private by default
```

```

public:
    // automatic inline functions
    int get_i(void) { return i; }
    void put_i(int j) { i = j; }
};

main(void)
{
    cl s;

    s.put_i(10);
    cout << s.get_i();

    return 0;
}

```

注意安排函数代码的方式。对每个短函数来说，这里的安排反映了 C++ 的通常风格。但是，以如下方式编写函数也是可以的：

```

class cl {
    int i; // private by default
public:
    // inline functions
    int get_i(void)
    {
        return i;
    }

    void put_i(int j)
    {
        i = j;
    }
};

```

在专家编写的 C++ 代码中。类似本例中所说明的短函数通常也定义在类定义内部。本书中其它 C++ 例子也将遵循这种约定。

## 17.8 对继承的进一步讨论

如你在前一章所看到的：一类有可能继承另一类的属性。这里将考察有关继承的进一步细节。

我们先来讨论一些术语。由另一类所继承的类被称为基类(base class)，有时也将它称为父类(parent class)，得以继承的类叫作派生类(derived class)。有时也叫子类(child class)。由于基类与派生类是传统的 C++ 术语，所以本书将使用这些术语。

在 C++ 中，一类可以将其元素分成三种类别。一元素可以是 public，private 和 protected。如你所知，一个 public 元素可为该程序中的任何其它函数所存取。一个 protected 元素也只可为成员或友元函数所存取。

当一类继承另一类时，基类的所有 private 元素都不可为派生类所存取。例如：

```

class X {
    int i;
    int j;
public:
    void get_ij(void);
    void put_ij(void);
};

```

```

class Y : public X {
    int k;
public:
    int get_k(void);
    void make_k(void);
};

```

这里，Y 的元素能够存取 X 的 public 函数 `get_k()` 和 `put_k()`，但是它们不能存取 `i` 和 `j`，因为 `i` 和 `j` 为 X 所私有。

可以通过使一类的 private 元素成为 protected 来让派生类对这些元素进行存取。例如：

```

class X {
protected://make protected
    int i;
    int j;
public:
    void get_ij(void);
    void put_ij(void);
};

class Y : public X {
    int k;
public:
    int get_k(void);
    void make_k(void);
};

```

现在，Y 可以存取 `i` 和 `j`，即使它们仍不可为程序的其余部分所存取。关键之处在于当使一个元素成为 protected 时。只有该类的成员函数能对它进行存取，但这种存取是可以被继承的。当一元素为 private 时，存取将不被继承。

如你从上一章中所知，继承一类的一般形式为：

```

class 类名:<access> 类名 {
    .
    .
    .
}

```

这里，access 必须是 private 或 public，它也可被省略。这时，如果基类为一结构，则假定为 public，如果基类为一类时，则假定为 private。如果 access 为 public，那么该基类的所有 public 元素都仍是派生类的 public 元素。基类的 protected 元素也是派生类的 protected 元素。如果 access 是 private，那么基类的所有 public 和 protected 元素就成为派生类的 private 元素。考虑如下程序：

```

#include <iostream.h>

class X {
protected:
    int i;
    int j;
public:
    void get_ij(void);
    void put_ij(void);
};

```

```

// In Y, i and j of X become protected members.
class Y : public X {
    int k;

public:
    int get_k(void);
    void make_k(void);
} ;

// Z has access to i and j of X, but not to
// k of Y, since it is private by default.
class Z : public Y {
public:
    void f(void);
} ;

void X::get_ij(void)
{
    cout << "Enter two numbers: ";
    cin >> i >> j;
}

void X::put_ij(void)
{
    cout << i << " " << j << "\n"
}

int Y::get_k(void)
{
    return k;
}

void Y::make_k(void)
{
    k = i*j;
}

void Z::f(void)
{
    i = 2;
    j = 3;
}

main(void)
{
    Y var;
    Z var2;

    var.get_ij();
    var.put_ij();

    var.make_k();
    cout << var.get_k();
    cout << "\n";

    var2.f();
    var2.put_ij();

    return 0;
}

```

由于 Y 将 X 声明成 public，所以 X 的 protected 元素就成为 Y 的 protected 元素。这意

味着它们也可为 Z 所继承, 并且该程序可正确地编译及运行。但是, 如下那样地在 Y 中改变 X 的状态会使得 Z 无法存取 i 和 j:

```
#include <iostream.h>

class X {
protected:
    int i;
    int j;
public:
    void get_ij(void);
    void put_ij(void);
};

// Now, i and j are converted to private members of Y.
class Y : private X {
    int k;
public:
    int get_k(void);
    void make_k(void);
};

// Because i and j are private in Y, they
// may not be accessed by Z.
class Z : public Y {
public:
    void f(void);
};

void X::get_ij(void)
{
    cout << "Enter two numbers: ";
    cin >> i >> j;
}

void X::put_ij(void)
{
    cout << i << " " << j << "\n";
}

int Y::get_k(void)
{
    return k;
}

void Y::make_k(void)
{
    k = i*j;
}

// This function no longer works.
void Z::f(void)
{
    // i = 2; i and j are no longer accessible
    // j = 3;
}

main(void)
{
    Y var;
    Z var2;
}
```

```

var.get_ij();
var.put_ij();

var.make_k();
cout << var.get_k();
cout << "\n";

var2.f();
var2.put_ij();

return 0;
}

```

当 X 在 Y 的声明中为 **private** 时，它使得 i 和 j 在 Y 中也被当作 **private**，这意味着它们不能被 Z 所继承。因此，Z 的函数 f() 就不可再存取它们。

有关 **private**、**protected** 和 **public** 的最后一点是：这些关键字可以任何次序，任意多次地出现在结构或类的声明中。例如，如下声明是完全正确的：

```

class my_class {
protected:
    int i;
    int j;
public:
    void f1(void);
    void f2(void);
protected:
    int a;
public:
    int b;
};

```

但是，在每个类或结构中。通常应只出现一次。

## 17.9 多重继承

对一类来说，有可能继承两个或更多个类的属性。为此，在派生类的基类表中使用用逗号隔开的继承表。一般形式是：

```

class      派生类名: 基类表
{
    .
    .
    .
};

```

例如，在下列程序中，Z 继承了 X 和 Y：

```

#include <iostream.h>

class X {
protected:
    int a;
public:
    void make_a(int i);
};

```

```

class Y {
protected:
    int b;
public:
    void make_b(int i);
} ;

// Z inherits both X and Y
class Z : public X, public Y {
public:
    int make_ab(void);
} ;

void X::make_a(int i)
{
    a = i;
}

void Y::make_b(int i)
{
    b = i;
}

int Z::make_ab(void)
{
    return a*b;
}

main(void)
{
    Z z;

    z.make_a(10);
    z.make_b(12);

    cout << z.make_ab();

    return 0;
}

```

在该例子中，Z 存取 X 及 Y 的 public 和 protected 部分。

在上面例子中，X、Y 和 Z 不含构造函数。但是，当一基类含一构造函数时情形就更为复杂。例如，可改变上例使类 X、Y 和 Z 都含一构造函数：

```

#include <iostream.h>

class X {
protected:
    int a;
public:
    X(void);
};

class Y {
protected:
    int b;
public:
    Y(void);
};

// Z inherits both X and Y
class Z : public X, public Y {

```



```

public:
    Z(void);
    int make_ab(void);
};

X::X(void)
{
    a = 10;
    cout << "initializing X\n";
}

Y::Y(void)
{
    cout << "initializing Y\n";
    b = 20;
}

Z::Z(void)
{
    cout << "initializing Z\n";
}

int Z::make_ab(void)
{
    return a*b;
}

main(void)
{
    Z i;

    cout << i.make_ab();

    return 0;
}

```

运行该程序时，显示如下输出：

```

initializing X
initializing Y
initializing Z
200

```

注意基类是以它们出现在 Z 的声明中的顺序而建立的。这一结论可以一般化。因为在 C++ 中，任何被继承的基类的构造函数都将以它们出现的次序而被调用。一旦基类已被初始化，就执行派生类的构造函数。

只要不存在着取变元的基类构造函数，派生类就不必有一构造函数。但是，当一基类含一用到了一个或多个变元的构造函数时，所有派生类也都必须含一构造函数。原因是为了能将变元传递给基类的构造函数。为传递变元至一基类，可在派生类构造函数声明的后面说明它们，其一般形式如下：

派生类构造函数(变元表)：

```

    base1(变元表), base2(变元表), ... ,baseN(变元表)
    {
        .
        .
    }

```

}

这里, base1 到 baseN 是为派生类所继承的基类名。注意冒号是用来将派生类构造函数与基类的变元表相隔开的。与基类相联的变元表可以由常量、全局参数及派生类构造函数的参数所组成。由于一对象的初始化是发生在运行时间,所以可以将定义在此类域内的任何标识符作为一变元。

下面程序说明了怎样通过修改前面的程序而将变元传递给一派生类的基类:

```
#include <iostream.h>

class X {
protected:
    int a;
public:
    X(int i);
};

class Y {
protected:
    int b;
public:
    Y(int i);
};

// Z inherits both X and Y
class Z : public X, public Y {
public:
    Z(int x, int y);
    int make_ab(void);
};

X::X(int i)
{
    a = i;
}

Y::Y(int i)
{
    b = i;
}

// Initialize X and Y via Z's constructor.
// Notice that Z does not actually use x or y
// itself, but it could, if it so chooses.
Z::Z(int x, int y) : X(x), Y(y)
{
    cout << "initializing\n";
}

int Z::make_ab(void)
{
    return a*b;
}

main(void)
{
    Z i(10, 20);
}
```

```

    cout << i.make_ab();

    return 0;
}

```

### 17.10 传递对象到函数

可以将一对象以对所有其它数据类型相同的方式来传递给一函数。对象传递给函数是使用了 C++ 的值调用参数传递约定。这意味着该对象的一个拷贝，而不是实际对象本身。因此对对象的任何修改都不影响用来调用函数的那一对象。下例程序就说明了这一点：

```

#include <iostream.h>
class OBJ {
    int i;
public:
    void set_i(int x) { i = x; }
    void out_i() { cout << i << " "; }
};
void f(OBJ x);

main(void)
{
    OBJ o;

    o.set_i(10);
    f(o);
    o.out_i(); // still outputs 10, value of i unchanged

    return 0;
}

void f(OBJ x)
{
    x.out_i(), // outputs 10
    x.set_i(100); // this affects only local copy
    x.out_i(); // outputs 100
}

```

还可以只将一对象的地址传递给一函数。当一对象的地址被传递时，在函数中对此对象的修改就会影响调用中所用到的对象。

### 17.11 对象数组

可以以创建任何其它数据类型的数组相同的方式来创建对象数组。例如，下面的程序建立了一个名为 `display` 的类。它包含有关可与一 PC 相连接的各种显示器的信息，它还专门包括了能被显示的颜色数及视频适配器的类型。在 `main()` 中，建立了含三个 `display` 对象的一数组，组成数组元素的对象可以通过通常的下标过程来存取。

```

// An example of arrays of objects

#include <iostream.h>

enum disp_type {mono, cga, ega, vga};

class display {
    int colors; // number of colors
    enum disp_type dt; // display type
public:
    void set_colors(int num) {colors = num;}
}

```

```

    int get_colors() {return colors;}
    void set_type(enum disp_type t) {dt = t;}
    enum disp_type get_type() {return dt;}
};

char names[4][5] = {
    "mono",
    "cga",
    "ega",
    "vga"
};

main(void)
{
    display monitors[3];
    register int i;

    monitors[0].set_type(mono);
    monitors[0].set_colors(1);

    monitors[1].set_type(cga);
    monitors[1].set_colors(4);

    monitors[2].set_type(vga);
    monitors[2].set_colors(16);

    for(i=0; i<3; i++) {
        cout << names[monitors[i].get_type()] << " ";
        cout << "has " << monitors[i].get_colors();
        cout << " colors" << "\n";
    }

    return 0;
}

```

该程序产生下列输出:

```

mono has 1 colors
cga has 4 colors
vga has 16 colors

```

虽然与对象数组无关。但请注意二维字符数组是怎样用来在一枚举值及其对应的字符串间进行转换的，在不含明确初始化的所有枚举中，第一个常量有值 0，第二个为 1，以此类推。同时，由 `get_type()` 返回的值可被用来检索 `names` 数组，从而打印出合适的名字。

多维对象数组的检索方式与其它数据类型的高维数组完全相同。

## 17.12 对象指针

如你所知，在 C 中可以直接存取一结构或通过一指向结构的指针来存取。类似地，在 C++ 中，可以直接地也可用对象指针来引用一对象。对象指针是 C++ 最重要的性质之一。

在使用实际对象本身时，为存取某对象的一元素使用句点(.)操作符。在使用一指向某对象的指针时，为存取该对象的一特定元素必须用箭头(->)操作符。对象中这两个操作符的使用与结构和联合中的用法是一致的。

声明对象指针的语法与声明任何其它数据类型指针的语法相同。下面程序建立一名为 `p_example` 的简单类，并定义了名为 `ob` 的该类的一对象以及名为 `p` 的指向类型为

**p\_example** 的一对象之指针，程序然后说明了怎样直接及间接地用指针来存取 **ob**。

```
// A simple example using an object pointer.

#include <iostream.h>

class P_example {
    int num;
public:
    void set_num(int val) {num = val;}
    void show_num();
};

void P_example::show_num()
{
    cout << num << "\n";
}

main(void, 程序
{
    P_example ob, *p; // declare an object and pointer to it

    ob.set_num(1); // access ob directly ...

    ob.show_num();

    p = &ob; // assign p the address of ob

    p->show_num(); // access ob using pointer

    return 0;
}
```

注意 **ob** 的地址是用地址操作符 **&** 而获得的，这与获得任何其它类型变量地址的方法是相同的。

当指针加 1 或减 1 时，它增加或减少的方式会使指针始终指向其基类型的下一元素。这也适用于对象指针。为说明这一点，修改了前面的程序以使 **ob** 是一类型为 **p\_example** 的含两个元素的数组。注意 **p** 是怎样被增 1 或减 1 以存取数组中的两个元素的。

```
// Incrementing an object pointer
#include <iostream.h>

class P_example {
    int num;
public:
    void set_num(int val) {num = val;}
    void show_num();
};

void P_example::show_num()
{
    cout << num << "\n";
}

main(void)
{
    P_example ob[2], *p;

    ob[0].set_num(10); // access objects directly
    ob[1].set_num(20);
```

```

    p = &ob[0]; // obtain pointer to first element
    p->show_num(); // show value of ob[0] using pointer

    p++; // advance to next object

    p->show_num(); // show value of ob[1] using pointer

    p--; // retreat to previous object
    p->show_num(); // again show value of ob[0]

    return 0;
}

```

此程序的输出为 10, 20, 10。

## 第十八章 函数和操作符重载

第十六章介绍了函数重载。本章考察有关此论题的更多内容,此外还将讨论操作符重载。这是 C++ 的两个最重要的性质。在进行这些讨论时,还将引入其它有关论题。

### 18.1 构造函数重载

虽然构造函数的功能很独特,但它们与其它类型的函数之差别并不大,它们也可被重载。为重载一个类的构造函数,只需说明其可能会有各种形式并定义与这些形式相对应的行为。例如,下面程序说明了一名为 `timer` 的作为减数计时器的类。当建立类型为 `timer` 的一对象时,此对象被赋予一初始时间值。当调用 `run()` 函数时,计时器减至零,然后敲响时钟。在该例中,构造函数被重载了以允许将时间说明成一整数、一串或两个说明分及秒的整数。

该程序利用了 Turbo C++ 的 `clock()` 函数,它返回从程序开始运行后,系统时钟嘀嗒的次数。将该值除以 `CLK_TCK` 就将 `clock()` 的返回值转换成了秒。`clock()` 的原型及 `CLK_TCK` 的定义都在头文件 `TIME.H` 中。

```
#include <iostream.h>
#include <stdlib.h>
#include <time.h>

class timer{
    int seconds;
public:
    // seconds specified as a string
    timer(char *t) { seconds = atoi(t); }

    // seconds specified as integer
    timer(int t) { seconds = t; }

    // time specified in minutes and seconds
    timer(int min, int sec) { seconds = min*60 + sec; }
    void run(void);
};

void timer::run(void)
{
    clock_t t1, t2;

    t1 = t2 = clock()/CLK_TCK;
    while(seconds) {
        if((t1/CLK_TCK+1) <= (t2=clock())/CLK_TCK) {
            seconds--;
            t1 = t2;
        }
    }
    cout << "\a"; // ring the bell
}

main(void)
{
    timer a(10), b("20"), c(1, 10);
```

```

    a.run(); // count 10 seconds
    b.run(); // count 20 seconds
    c.run(); // count 1 minute, 10 seconds
    return 0;
}

```

当在 `main()` 中创建 `a`, `b`, `c` 时, 使用了重载构造函数支持的三种不同方法来设置它们的初值。每个方法都引起合适的构造函数被使用, 因而正确地初始化了所有三个变量。

在上面的程序中, 由于仅仅决定说明时间的一种方法并不困难, 你可能看不出重载构造函数的价值。但是, 如果要为别人创建一类库, 那么就可能想要为最常用的初始化形式提供构造函数以赋予用户最大的灵活性。此外, C++ 还有一性质使得重载构造函数很有价值。下面将很快讨论这一点。

## 18.2 C++ 中的局部变量

在继续讨论重载函数之前, 先需要解决局部变量的说明问题。

在 C 中, 必须在一块的开始说明该块中所使用的所有局部变量, 而不能在已出现一语句之后再说明一变量。例如, C 中下列片断代码是错误的:

```

/* incorrect in C */
f()
{
    int i;

    i = 10;

    int j;
    .
    .
}

```

由于语句 `i=10` 插在对 `i` 和 `j` 的说明之间, C 编译程序就会发现错误并拒绝编译该函数。但是, 在 C++ 中, 该片断完全可以被接受, 并且将被正确编译。例如, 下面就是一个完全正确的 C++ 程序:

```

#include <iostream.h>
#include <string.h>

main(void)
{
    int i;

    i = 10;

    int j = 100; // perfectly legal in C++

    cout << i*j << "\n";

    cout << "Enter a string: ";
    char str[80];
    cin >> str;
}

```



```

// display the string in reverse order
int k; // declare k where it is needed
k = strlen(str);
k--;
while(k>=0) {
    cout << str[k];
    k--;
}
return 0;
}

```

如此程序所示，在 C++ 中，可以在一段代码块中的任何地方来说明局部变量。由于 C++ 将数据及代码封装起来的哲学，最好在接近使用变量的地方对之说明，而不是一味地在块之开始处说明。在此例中，对 *i* 和 *j* 的说明只是为清楚起见而加以隔开的。但是你会发现将 *k* 局部于其相关的代码有助于封装此例程。将变量说明在接近其使用处可以避免产生意外的副作用。

### 18.2.1 动态初始化

在 C++ 中，局部变量和全局变量都可在运行时被初始化。这有时也被称为动态初始化(dynamic initialization)。记住在 C 中，变量必须用常量表达式来初始化。这是由于 C 编译程序在编译时就固定初始代码。但是在 C++ 中，变量可在其被说明时用任何正确的 C++ 表达式来初始化。例如，下面就是 C++ 中完全正确的变量初始化：

```

int n = atoi(gets(str));

long pos = ftell(fp);

double d = 1.02 * count / deltax;

```

利用动态初始化可以改进上一节中的例子程序：

```

#include <iostream.h>
#include <string.h>

main(void)
{
    int i;

    i = 10;

    int j = 100;

    cout << i*j << "\n";

    cout << "Enter a string: ";
    char str[80];
    cin >> str;

    // initialize k dynamically at runtime
    int k = strlen(str)-1;

    while(k>=0) {
        cout << str[k];
    }
}

```

```

        k--;
    }
    return 0;
}

```

这里，由于对 `strlen()` 的调用是在运行时决定的，所以 `k` 被动态初始化。

### 18.3 将动态初始化用于构造函数

如同简单变量，对象可在创建时动态地被初始化。这一性质可使你利用只有在运行时间才能知道的信息来建立自己需要的那类对象。为说明这一点，重新考虑本章前面的计时器程序。

在计时器程序的第一个例子中，由于类型为 `timer` 的所有对象都是用常量来初始化的，因而重载 `timer()` 构造函数就无甚意义。但是，当需要在运行时间初始化一对象时，若能允许使用各种初始化格式就很有意义。这样可使程序员能灵活地在任何给定的时刻使用与数据格式最为匹配的构造函数。例如，在计时器程序的下面版本中，在运行时间利用动态初始化建立了 `a` 和 `b` 两个对象。

```

#include <iostream.h>
#include <stdlib.h>
#include <time.h>

class timer{
    int seconds;
public:
    // seconds specified as a string
    timer(char *t) { seconds = atoi(t); }

    // seconds specified as integer
    timer(int r) { seconds = r; }

    // time specified in minutes and seconds
    timer(int min, int sec) { seconds = min*60 + sec; }

    void run(void);
};

void timer::run(void)
{
    clock_t t1, t2;
    t1 = t2 = clock()/CLK_TCK;
    while(seconds) {
        if((t1/CLK_TCK) <= (t2=clock())/CLK_TCK) {
            seconds--;
            t1 = t2;
        }
    }
    cout << "\a"; // ring the bell
}

main(void)
{
    timer a(10);

    a.run();
}

```

```

    cout << "Enter number of seconds: ";
    char str[80];
    cin >> str;
    timer b(str); // initialize at runtime
    b.run();

    cout << "Enter minutes and seconds: ";
    int min, sec;
    cin >> min >> sec;
    timer c(min, sec); // initialize at runtime
    c.run();
    return 0;
}

```

对象 **a** 是用一整数常量来构造的，对象 **b** 是用由用户输入的信息来构造的。对 **b** 来说，由于用户输入了一串，就使得重载 `timer()` 以接受它很有意义。类似地，对象 **c** 也是在运行时间利用用户输入的信息而构造的。在这里，由于时间是以分和秒输入的，使用这种形式来构造对象 **c** 是合乎逻辑的。通过允许各种初始化格式，程序员在初始化一对象时就不必进行任何不必要的从一形式到另一形式的转换。

重载构造函数的要点是通过使对象能以相对于其特定用途来说最自然的方式，帮助程序员处理更大的复杂性。由于将时间值传递给一对象存在着三种常见方法，就有必要使 `timer()` 被重载以接受各种方式，但是重载 `timer()` 以接受小时或天数就可能不是个好想法，这是因为用户程序充斥着处理很少出现的情形的构造函数只会降低程序的稳定性。需要指出的是你必须对什么东西组成了正确的构造函数重载、什么东西是华而不实的作出决定。

#### 18.4 关键字 `this`

在讨论操作符重载前，有必要先来了解 C++ 的另一名为 `this` 的关键字，它是许多被重载操作符的基本成分。

每当一成员函数被激活时，就自动地将指向激活它的对象的一指针传递给它。可以用 `this` 来存取该指针。进一步，该指针还将在调用一成员函数时自动地被传递。指针 `this` 是所有成员函数的一个隐含参数。

一个成员函数可以直接存取其类的私有数据。例如，给出下类：

```

class   cl{
    int  i;
    .
    .
    .
};

```

一个成员函数可用下列语句将 `i` 赋以值 10：

```
i = 10;
```

实际上，上述语句是语句：

```
this->i = 10;
```

的简写。

为了解 `this` 指针的工作情况，考察下列短程序：

```
#include <iostream.h>

class cl {
    int i;
public:
    void load_i(int val) { this->i = val; } // same as i = val
    int get_i(void) { return this->i; } // same as return i
};

void main(void)
{
    cl o;

    o.load_i(100);
    cout << o.get_i();
}
```

该程序显示数 100。

虽然上面的例子很平凡——事实上，没有人会在实际中如此使用指针 `this`——但在下一节中你会看到为什么 `this` 指针是如此重要。

## 18.5 操作符重载

C++ 的另一个与函数重载相关的性质叫作操作符重载(operator overloading)。除去很少的例外，大多数 C++ 操作符都可被赋以与特定类相关的特定意义。例如，一个定义了一链接表的类可以使用 + 操作符来将一对象加至该表中。另一个类则可以用完全不同的方式来使用 + 操作符。当一操作符被重载时，其所有原先的意义都未失去。它只是定义了相对一特定类的一个新的操作符。因此，重载 + 以处理一链接表并不改变其相对于整数的意义。

为重载一操作符，必须定义该操作相对于它要作用于其上的类的意义，为此，产生了 operator 函数。该函数的一般形式如下：

```
type 类名:: operator#(变元表)
{
    //相对于该类而定义的操作
}
```

这里，`type` 是由所说明操作返回的值类型。通常，返回值的类型与类的类型相同(虽然它可以是所选择的任何类型)。这种类型相同的原因是由于它方便了对复杂表达式的使用，这将很快讨论。

操作符函数必须是它们所要用于的类的成员或友元函数。虽然非常相似。但在重载一成员操作符函数和重载——友元操作符函数的方式之间还有一些差别。在本节中，只有成员函数将被重载。以后将看到怎样重载友元操作符函数。

为了解操作符重载是怎样工作的，让我们以创建一名为 `three_d` 的简单例子作为开始，该类含三维空间中一对象的坐标。该程序重载相对于 `three_d` 类的 + 和 = 操作符：

```
#include <iostream.h>

class three_d {
    int x, y, z; // 3-d coordinates
public:
    three_d operator+(three_d t);
    three_d operator=(three_d t);
}
```

```

    void show(void) ;
    void assign(int mx, int my, int mz);
} ;

// Overload the +.
three_d three_d::operator+(three_d t)
{
    three_d temp;

    temp.x = x+t.x;
    temp.y = y+t.y;
    temp.z = z+t.z;
    return temp;
}

// Overload the =.
three_d three_d::operator=(three_d t)
{
    x = t.x;
    y = t.y;
    z = t.z;
    return *this;
}

// show X, Y, Z coordinates
void three_d::show(void)
{
    cout << x << ", ";
    cout << y << ", ";
    cout << z << "\n";
}

// Assign coordinates
void three_d::assign(int mx, int my, int mz)
{
    x = mx;
    y = my;
    z = mz;
}

main(void)
{
    three_d a, b, c;

    a.assign(1, 2, 3);
    b.assign(10, 10, 10);

    a.show();
    b.show();

    c = a+b; // now add a and b together
    c.show();

    c = a+b+c; // add a, b and c together
    c.show();

    c = b = a; // demonstrate multiple assignment
    c.show();
    b.show();
    return 0;
}

```

此程序产生如下输出:

```
1, 2, 3
10, 10, 10
11, 12, 13
22, 24, 26
1, 2, 3
1, 2, 3
```

仔细研究该程序,可能就会惊奇地发现两个操作符函数都只有一个参数。虽然它们重载了二元操作,这种表面上矛盾的原因是:当用一成员函数重载一个二元操作符时,只需明确地传递给它一个变元。另一变元是用 `this` 指针而隐含地加以传递的。因此,在语句

```
temp.x = x + (*x);
```

中, `x` 意指 `this->x`,这是与促使对操作符函数的调用对象相联的 `x`。无论如何,引起对操作符函数的调用的是在一操作左边的对象。右边的操作被传递给函数。

一般来说,在任何使用一成员函数时,为重载一单目操作符,无需任何参数。为重载一个二元操作符,只需一个参数。不管何种情形,激活操作符函数的对象都是由 `this` 指针隐含传递的。

为理解操作符重载的工作过程,让我们仔细地考察一下该程序:先从重载 `+` 操作符开始,当类型为 `three_d` 的两对象被 `+` 操作符所操作时,它们各自的坐标都被相加在一起,如与此类相联的 `operator()` 函数所说明的。但是要注意,该函数并不修改两个操作符中的任何一个。相反,类型为 `three_d` 的一对象由含操作结果的函数所返回。这一点很重要。

为理解为什么 `+` 操作符不可改变任一对象的内容,考虑一下标准的算术 `+` 操作: `10+12`。该操作的结果为 22,但是此结果对 10 及 12 都不作改变。虽然并没有强行要求一重载操作符必须以任何 C++ 内部类型相同的方式来使用,但是通常最好应与其原先用法的风格相同。

关于 `+` 操作符怎样被重载的另一关键之处是它返回一类型为 `three_d` 的对象。虽然该函数可以返回任何正确的 C++ 类型,但它返回了一个 `three_d` 对象的事实使得 `+` 操作符可被用在诸如 `a+b+c` 这样更复杂的表达式中。

与 `+` 操作符相反,赋值操作符却使其变元中的一个被修改(归根到底,这是赋值语句的本质)。由于 `operator=()` 函数被出现在赋值左边的对象所调用,所以正是该对象为赋值操作所修改。但是,即使是赋值操作,也必须返回一值,因为在 C++ (以及 C) 中,赋值操作产生出现在右边的值。因此,为允许语句

```
a=b=c=d;
```

的出现,有必要使 `operator=()` 返回由 `this` 所指向的对象,这将是出现在赋值语句左边的对象,这使得连续的赋值成为合法。

还可重载单目操作符,例如 `++` 或 `--`。如前所述,当重载一单目操作符时,没有对象被明确的传递给操作符函数。该操作是通过被隐含传递的 `this` 指针而作用在产生函数调用的对象上的。例如。

下面是为类型为 `three_d` 的对象定义加 1 操作这一先前程序的修正版:

```
#include <iostream.h>

class three_d {
    int x, y, z; // 3-d coordinates
public:
```

```

three_d operator+(three_d op2); // opl is implied
three_d operator=(three_d op2); // opl is implied
three_d operator++(void); // opl is also implied here

void show(void) ;
void assign(int mx, int my, int mz);
};

three_d three_d::operator+(three_d op2)
{
    three_d temp;

    temp.x = x+op2.x; // these are integer additions
    temp.y = y+op2.y; // and the + retains its original
    temp.z = z+op2.z; // meaning relative to them
    return temp;
}

three_d three_d::operator=(three_d op2)
{
    x = op2.x; // these are integer assignments
    y = op2.y; // and the = retains its original
    z = op2.z; // meaning relative to them
    return *this;
}

// Overload a unary operator.
three_d three_d::operator++(void)
{
    x++;
    y++;
    z++;
    return *this;
}

// show X, Y, Z coordinates
void three_d::show(void)
{
    cout << x << ", ";
    cout << y << ", ";
    cout << z << "\n";
}

// Assign coordinates
void three_d::assign(int mx, int my, int mz)
{
    x = mx;
    y = my;
    z = mz;
}

main(void)
{
    three_d a, b, c;

    a.assign(1, 2, 3);
    b.assign(10, 10, 10);

    a.show();
    b.show();
}

```

```

c = a+b; // now add a and b together
c.show();

c = a+b+c; // add a, b and c together
c.show();

c = b = a; // demonstrate multiple assignment
c.show();
b.show();

c++; // increment c
c.show();
return 0;
}

```

需要记住的是当重载++或--时,要想从此 operator 函数内部决定操作符是在操作数之前还是之后是不可能的。也就是说,operator 函数无法知道引起对该函数之调用的表达式是

++OBJ;

还是

OBJ++;

这里,OBJ 是此操作所影响的对象。

在为某类定义了一重载操作符后,将该操作符作用于该类不需要与在将该操作符作用于 C++ 的内部类型时,此操作符所表现出的缺省用法有任何关系。例如,作用于 cout 和 cin 的 << 和 >> 与作用于整数类型的同样操作符毫无关系。但是,为了程序的结构化及可读性,只要可能,一重载操作符就应反映出此操作符原先用法之实质。例如,相对于 three\_d 的 + 在概念上类似于相对于整数类型的 +。将相对于某类的 + 操作符以某种完全出人意料的方式来定义会是无甚价值的。这里的主要概念就是在要赋予重载操作符任何所希望的含意时,为了清晰起见,最好能使新的意义与其原先意义相关。

还存在着对重载操作符的一些限制。首先,不能更改任何操作符的优先级。第二,不可改变此操作符所需要的操作数的数目,虽然 operator() 函数可以略省一操作数。最后,除了 = 以外,重载操作符都可为任何派生类所继承。(当然,如果需要的话,可以重载相对于派生类的操作符)。如果每个类都需要一个 = 操作符,那么就都需要明确地定义其自己的 = 操作符重载。

下列是不能重载的所有操作符:

., ::, .\*, ?

### 18.5.1 友元操作符函数

一个操作符函数可能是一类的友元而不是成员。如你在本章前面所知,友元函数没有隐含的变元 this。因此,当一友元被用来重载一操作符时,如果该操作符是双目的,则两个操作数都被明确地传递给操作符;如果该操作符是单目的,那么就传递单个操作数。仅有的不能使用友元函数的操作符是 =, &, ! 和 +。其余的操作符可以使用成员或友元函数来实现相对于其类所说明的操作。例如,下面就是先前使用一友元函数而不是一成员函数来重载 + 操作的程序的修改版:



```

#include <iostream.h>

class three_d {
    int x, y, z; // 3-d coordinates
public:
    friend three_d operator+(three_d op1, three_d op2);
    three_d operator=(three_d op2); // op1 is implied
    three_d operator++(void); // op1 is implied here, too

    void show(void) ;
    void assign(int mx, int my, int mz);
};

// This is now a friend function.
three_d operator+(three_d op1, three_d op2)
{
    three_d temp;

    temp.x = op1.x + op2.x; // these are integer additions
    temp.y = op1.y + op2.y; // and the + retains its original
    temp.z = op1.z + op2.z; // meaning relative to them
    return temp;
}

three_d three_d::operator=(three_d op2)
{
    x = op2.x; // these are integer assignments
    y = op2.y; // and the = retains its original
    z = op2.z; // meaning relative to them
    return *this;
}

// Overload a unary operator.
three_d three_d::operator++(void)
{
    x++;
    y++;
    z++;
    return *this;
}

// show X, Y, Z coordinates
void three_d::show(void)
{
    cout << x << ", ";
    cout << y << ", ";
    cout << z << "\n";
}

// Assign coordinates
void three_d::assign(int mx, int my, int mz)
{
    x = mx;
    y = my;
    z = mz;
}

main(void)
{
    three_d a, b, c;

```

```

    a.assign(1, 2, 3);
    b.assign(10, 10, 10);

    a.show();
    b.show();

    c = a+b; // now add a and b together
    c.show();

    c = a+b+c; // add a, b and c together
    c.show();

    c = b = a; // demonstrate multiple assignment
    c.show();
    b.show();

    c++; // increment c
    c.show();
    return 0;
}

```

现在两个操作数都被传递给 `operator()`。左边的操作数在 `op1` 中, 右边的操作数在 `op2` 中。

在许多情形下, 当重载一操作符使用一友元函数而非成员函数是无益的。但是, 有一种情形下必须使用友元函数。原因在于指向激活一成员操作符函数的对象的指针在 `this` 中被传递。在二元操作符的情形中, 左边的对象激活函数。这在左边的对象定义了所说明的操作时不存在什么问题。例如, 假定某个名为 `O` 的对象有为之定义的赋值及加法, 这样, 下列语句就完全正确:

```
O = O + 10
```

由于对象 `O` 在 `+` 操作符的左边, 所以它激活其重载操作符函数, 该函数被假定能够将一整数加到 `O` 的某元素。但是, 语句:

```
O = 10 + O;
```

就不行。原因在于位于左边的对象是一整数, 这是一个没有为其定义涉及一整数及一个 `O` 类型对象的操作的内部类型。

操作左边内部类型的问题可用两个友元函数来重载 `+` 而解决, 在这种情形下, 操作符函数被明确传递以两个变元。如同所有其它被重载的函数, 操作符函数是根据其变元的类型而被激活。用友元来重载 `+` (或任何其它二元操作符) 可使得一内部类型出现在操作符的左边。下列程序说明了怎样完成这一点:

```

#include <iostream.h>

class CL {
public:
    int count;
    CL operator=(int i);
    friend CL operator+(CL ob, int i);
    friend CL operator+(int i, CL ob);
};

CL CL::operator=(int i)
{
    count = i;
    return *this;
}

```

```

// This handles ob + int
CL operator+(CL ob, int i,
{
    CL temp;
    temp.count = ob.count + i;
    return temp;
}

// This handles int + ob.
CL operator+(int i, CL ob)
{
    CL temp;

    temp.count = ob.count + i;
    return temp;
}

main(void)
{
    CL obj;

    obj = 10;
    cout << obj.count << " "; // outputs 10

    obj = 10 + obj; // add object to integer
    cout << obj.count << " "; // outputs 20

    obj = obj + 12; // add integer to object
    cout << obj.count;      // outputs 32
    return 0;
}

```

`operator+()` 函数被两次重载以接纳可出现在加法操作中的一整数及类型 CL 的对象的两种方式。

虽然可以使用一个友元函数来重载一个单目操作符(如 ++), 但首先需要了解 C++ 的另一个被称为引用(reference)的性质。

## 18.6 引用

缺省地, C 和 C++ 用值调用来将变元传递给函数。用值调用来传递变元将拷贝为函数所用的变元, 并阻止调用中所用的变元被函数所修改。在 C 中(或在 C++ 中), 当一函数需要能修改用作变元的变量值时, 参数需作为指针类型而被明确声明, 该函数必须用 \* 指针操作符来对调用的变量进行操作。例如, 下列程序实现了一交换其两个整数变元的名 为 `swap()` 的函数:

```

#include <iostream.h>

void swap(int *a, int *b);

main(void)
{
    int i, j;

    i = 10;
    j = 20;

```

```

    cout << i << " " << j << "\n";

    swap(&i, &j); // exchange their values

    cout << i << " " << j << "\n";
    return 0;
}

// C-like, explicit pointer version of swap().
void swap(int *a, int *b)
{
    int t;

    t = *a;
    *a = *b;
    *b = t;
}

```

当调用 `swap()` 时，用在调用中的变量必须在其前面加上 `&` 操作符以便产生指向各变元的一个指针。这是在 C 中生成引用调用的方法，虽然 C++ 仍然允许这一语法，但它通过使用所谓引用参数(reference parameter)来支持生成引用调用的一种更清晰方法。

在 C++ 中，有可能让编译器来自动地生成一引用调用而不是为一具体函数的一个或几个参数来生成一值调用。这是通过将 `&` 放在函数说明中的参数名前而完成的。例如，下面是一个带有一类型为 `float` 的引用参数的名为 `f()` 的函数：

```

void f(int &f)
{
    f = rand(); // this modifies calling argument
}

```

该声明式子也被用在函数原型中。注意语句 `f=rand()` 不使用 `*` 指针操作符。当说明一引用参数时，C++ 编译器自动知道它是一指针，并且将去掉引用。

一旦编译程序看到此声明，它将自动地传给 `f()` 它调用时用到的变量地址。例如，给出片段代码：

```

int val;

f(val); // get random value
printf("%d", val);

```

传递给 `f()` 的是 `val` 的地址，而不是值

如果熟悉 pascal，知道 C++ 的引用

助。

`f()` 可以修改 `val` 的值。

于 Pascal 中的 VAR 参数可能会有所帮

为了解引用参数的实际用途，`swap()` 函数在下面的程序版本中被使用而加以重写。注意 `swap()` 是怎样被说明及调用的。

```

#include <iostream.h>

void swap(int &a, int &b); // declare as reference parameters

main(void)
{
    int i, j;

```

```

    i = 10;
    j = 20;

    cout << i << " " << j << "\n";

    swap(i, j); // exchange their values

    cout << i << " " << j << "\n";
    return 0;
}

// Here, swap() is defined as using call-by-reference,
// not call-by-value.
void swap(int &a, int &b)
{
    int t;

    t = a;
    a = b; // this swaps i
    b = t; // this swaps j
}

```

### 18.6.1 非参数的引用变量

虽然引用在 C++ 中的被引入主要是作为支持引用调用的参数传递的，说明不为函数参数的引用变量仍是可能的。但是，必须说明，由于非参数引用变量可能会使用户程序变得含混及结构不良，应当尽可能限制对它们的使用。

非参数引用变量有时也被称为独立引用。由于一引用变量必须指向某一对象，一独立的引用必须在它被说明时加以初始化。一般来说，这意味着它将被赋以一个先前被说明的变量的地址。一旦如此之后，此引用变量就可在它引用的变量可被使用的任何地方加以使用。事实上，在这两者之间几乎没有什么区别。例如，考虑下面程序：

```

#include <iostream.h>

main(void)
{
    int j, k;
    int &i = j;

    j = 10;

    cout << j << " " << i; // outputs 10 10

    k = 121;
    i = k; // copies k's value into j
           // not k's address

    cout << "\n" << j; // outputs 121
    return 0;
}

```

该程序显示如下输出：

```

10    10
121

```

关键之处是由引用变量所指向的对象是固定不变的。因此，当计算语句  $i=k$  时，拷贝到  $j$  (由  $i$  所指) 的是  $k$  的值，而不是其地址，换句话说，引用不是指针。

对独立引用存在着几条限制。第一，不可引用一引用变量，即不能取其地址。第二，

不允许对位域使用引用。第三，不允许创建引用数组。最后，不能创建指向一引用的指针。

还可使用一独立的引用来指向一常量。例如，语句

```
int    &i=100;
```

这里，*i* 引用了程序常量表中值 100 被存储的位置。

如前所述，使用独立引用一般不是个好主意，因为它们是不必要的，并容易使程序变得含混。

### 18.6.2 使用引用来重载单目操作符

在前面一节的计时器程序的最近版本中，使用一友元函数没有重载++操作符，因为需要使用的是一个引用，在本节中你将了解其原因。

首先，回顾一下相对于 *three\_d* 类的++重载操作符的原先版本。为了方便，它在下面给出：

```
// Overload a unary operator.
three_d three_d::operator++(void)
{
    x++;
    y++;
    z++;
    return *this;
}
```

所有成员函数都有一个作为隐含变元的指向函数的指针，该指针就用关键字 *this* 来在成员函数内部加以引用。这就是为什么在用成员函数重载一单目操作符时，没有明确声明变元的原因。在此情形中所需要的唯一变元是指向激活调用重载操作符函数的对象的指针。由于 *this* 是一指向此对象的指针，对此对象的私有数据的任何改变都将影响到产生对操作符函数之调用的对象。与成员函数不同，友元函数不接受 *this* 指针，因此不能引用激活它的对象。为此，试图如下那样地创建一个 *friend operator++()* 函数是不正确的：

```
// THIS WILL NOT WORK
three_d operator++(three_d opl)
{
    opl.x++;
    opl.y++;
    opl.z++;
    return opl;
}
```

原因在于只有激活 *operator++()* 调用对象的一个拷贝被用参数 *opl* 传递给函数。因此，在 *operator++()* 内的改变不会影响被调用的对象。

起初，你可能会认为解决这问题的方法是利用指向激活调用之对象的一指针来定义如下的友元操作符函数：

```
// THIS WILL NOT WORK
three_d operator++(three_d *opl)
{
    opl->x++;
    opl->y++;
    opl->z++;
    return *opl;
}
```

虽然该函数到目前为止是正确的，但 C++ 却不知道怎样正确地激活它。例如，假定是 `operator++()` 函数的该版本，此代码片段无法编译：

```
three_d ob(1, 2, 3);
&ob++;    //will not compile
```

问题在于语句 `&ob++` 是二义的。

在重载单目 `++` 或 `--` 时使用友元的方法是使用一引用参数。以这种方式，编译器会事先知道在调用函数时，它必须生成一地址。这就避免了上述的二义性。下面是使用了一友元 `operator++()` 函数的完整 `three_d` 程序：

```
// This version uses a friend operator++() function.
#include <iostream.h>

class three_d {
    int x, y, z; // 3-d coordinates
public:
    friend three_d operator+(three_d op1, three_d op2);
    three_d operator=(three_d op2); // op1 is implied
    // use a reference to overload the ++
    friend three_d operator++(three_d &op1);

    void show(void) ;
    void assign(int mx, int my, int mz);
};

// This is now a friend function.
three_d operator+(three_d op1, three_d op2)
{
    three_d temp;

    temp.x = op1.x + op2.x; // these are integer additions
    temp.y = op1.y + op2.y; // and the + retains its original
    temp.z = op1.z + op2.z; // meaning relative to them
    return temp;
}

three_d three_d::operator=(three_d op2)
{
    x = op2.x; // these are integer assignments
    y = op2.y; // and the = retains its original
    z = op2.z; // meaning relative to them
    return *this;
}

// Overload a unary operator using a friend function.
// This requires the use of a reference parameter.
three_d operator++(three_d &op1)
{
    op1.x++;
    op1.y++;
    op1.z++;
    return op1;
}

// show X, Y, Z coordinates
void three_d::show(void)
{
    cout << x << ", ";
    cout << y << ", ";
    cout << z << "\n";
}
}
```

```

// Assign coordinates
void three_d::assign(int mx, int my, int mz)
{
    x = mx;
    y = my;
    z = mz;
}

main(void)
{
    three_d a, b, c;

    a.assign(1, 2, 3);
    b.assign(10, 10, 10);

    a.show();
    b.show();

    c = a+b; // now add a and b together
    c.show();

    c = a+b+c; // add a, b and c together
    c.show();

    c = b = a; // demonstrate multiple assignment
    c.show();
    b.show();

    c++; // Increment c
    c.show();
    return 0;
}

```

记住有一点很重要：一般说来，应该使用成员函数来实现重载操作符。记住 C++ 中允许友元函数主要是为了用来处理某些特殊情形。

### 18.7 操作符重载的又一例子

我们将开发另一个操作符重载的例子以结束本章，该例子实现一串类型并定义了与该串相关的几个操作。许多 C 及 C++ 的初学者都抱怨没有明确的串类型。即使 C 的将串实现为字符数组的方法比起将串本身作为一类型的方法更不灵活而有效，对初学者来说，串仍可能缺乏如同 BASIC 那样的语言的概念上的清晰性。但是，利用 C++，有可能通过定义一串类及与该类相关的操作来综合两者之精华。

首先，下面的类说明了类型 `str_type`：

```

#include <iostream.h>
#include <string.h>

class str_type {
    char string(80);
public:
    str_type(char *str = "\0") { strcpy(string, str); }

    str_type operator+(str_type str); // concatenate
    str_type operator=(str_type str); // assign

```



```
// output the string
void show_str(void) { cout << string; }
};
```

`str_type` 是在其私有部分中说明一串的。为了说明此例，串不可长于 80 字节。此类有一个构造函数，它可被用来用一特定值初始化数组 `string`，或者在没有初始化值时赋给它一个空串，它说明了用来完成串合并及赋值的两个重载操作符。最后，它支持了函数 `show_str()`，它将 `string` 输出到屏幕。重载操作符函数如下：

```
// Concatenate two strings.
str_type str_type::operator+(str_type str) {
    str_type temp;

    strcpy(temp.string, string);
    strcat(temp.string, str.string);
    return temp;
}

// Assign one string to another.
str_type str_type::operator=(str_type str) {
    strcpy(string, str.string);
    return *this;
}
```

给出这些定义，下列 `main()` 函数说明了其用途：

```
main(void)
{
    str_type a("Hello "), b("There"), c;

    c = a + b;

    c.show_str();
    return 0;
}
```

该程序在屏幕上输出 “Hello There”。它首先合并串 `a` 及 `b`，然后将此值赋给 `c`。记住 `=` 和 `+` 都只为类型为 `str_type` 的对象所定义。例如，下面语句是不正确的，因为它试图将一通常的 C++ 串赋给对象 `a`：

```
a="this is currently wrong";
```

但是，可以改进 `str_type` class 以允许这样的语句。

为扩充由 `str_type` 类所支持的操作的类型，以使你能将串赋给对象或合并一串到一对象，需要第二次重载 `+` 和 `=` 操作。首先，类说明被改成：

```
class str_type {
    char string[80];
public:
    str_type(char *str = "\0") { strcpy(string, str); }

    str_type operator+(str_type str); // concatenate objects
    str_type operator+(char *str);   // concatenate object with
                                     // a string

    str_type operator=(str_type str); // assign object to
                                     // object
    char *operator=(char *str);       // assign string to object
}
```

```
void show_str(void) { cout << string; }
};
```

接着，实现被重载的 `operator+()` 和 `operator=()`，如下所示：

```
// Assign a string to an object
str_type str_type::operator=(char *str)
{
    str_type temp;

    strcpy(string, str);
    strcpy(temp.string, string);
    return temp;
}

// Add a string to an object
str_type str_type::operator+(char *str)
{
    str_type temp;

    strcpy(temp.string, string);
    strcat(temp.string, str);
    return temp;
}
```

注意右边的变元不是类型为 `str_type` 的一对象，而只是指向一以空字符终结的字符数组——即 C++ 的一正常串——的指针。但是，两个函数都返回类型为 `str_type` 的一对象。虽然从理论上来说，该函数可以返回某些其它类型，但返回一对象最有意义，因为这些操作的目标也是对象。定义接受 C++ 正常串作为右边操作数的串操作之优点是允许可以用自然的方式书写一些语句。例如，下面现在是正确的语句：

```
str_type a, b, c;

a = "hi there"; // assign an object a string

c = a + " George"; // concatenate an object with a string
```

下列程序融合了 + 及 = 操作的附加意义，并说明了其用途：

```
// Expanding the string type.
#include <iostream.h>
#include <string.h>

class str_type {
    char string(80);
public:
    str_type(char *str = "\0") { strcpy(string, str); }

    str_type operator+(str_type str);
    str_type operator+(char *str);

    str_type operator=(str_type str);
    str_type operator=(char *str);

    void show_str(void) { cout << string; }
};

str_type str_type::operator+(str_type str) {
    str_type temp;
```

```

        strcpy(temp.string, string);
        strcat(temp.string, str.string);
        return temp;
    }

    str_type str_type::operator=(str_type str) {
        strcpy(string, str.string);
        return *this;
    }

    str_type str_type::operator=(char *str)
    {
        str_type temp;

        strcpy(string, str);
        strcpy(temp.string, string);
        return temp;
    }

    str_type str_type::operator+(char *str)
    {
        str_type temp;

        strcpy(temp.string, string);
        strcat(temp.string, str);
        return temp;
    }

    main(void)
    {
        str_type a("Hello "), b("There"), c;

        c = a + b;

        c.show_str();
        cout << "\n";

        a = "to program in because";
        a.show_str();
        cout << "\n";

        b = c + "C++ is fun";

        c = c + " +a+ " + b;
        c.show_str();
        return 0;
    }

```

程序在屏幕上的输出为:

```

Hello There
to program in because
C++ is fun to program in because C++ is fun

```

在继续之前，一定要理解这些输出是怎样产生的。此外，还应试着建立别的一些串操作。例如，可以定义一个减以使它完成一子串的删除。又如，如果对象 A 的串是 "This is a test"，对象 B 的串是 "is"，那么 A-B 就产生 "th a test"。这里，子串所有的出现都被从原来的串中所移出去。

## 第十九章 继承、虚函数及多态性

面向对象程序设计之关键在于多态性的概念。多态性概念是用于描述过程的，由此可用同一个名访问一个函数的不同实现。由于这个原因，多态性有时候被概括为“单界面，多方法”。这就是说，一般操作类可用相同的方式访问，即使与各操作相关的特定行为可能并不相同。

在C++中，运行时间和编译时间都支持多态性。操作符和函数重载就是编译时间多态性的例子。但即使操作符和函数重载功能如何强，它们也不能完成一个实际的、面向对象的语言所要求的所有任务。因而，C++还通过使用派生类和虚函数来支持运行时间多态性，这也是本章的主要内容。

本章首先简短地讨论指向派生类的指针。因为需要用这些指针来支持运行时间多态性。

### 19.1 派生类指针

指向基本类和派生类的指针是相关的，假设有一个基本类 `B_class` 和一个从 `B_class` 派生的类 `D_class`，在C++中，任何被说明为指向 `B_class` 的指针也可以是指向 `D_class` 的指针。例如有：

```
B_class *p; // pointer to object of type B_class
B_class B_ob; // object of type B_class
D_class D_ob; // object of type D_class

p = &B_ob; // p points to object of type B_class
p = &D_ob; /* p points to object of type D_class,
            which is an object derived from B_class. */
```

利用 `p`，所有从 `B_ob` 继承的 `D_ob` 元素就都被访问。但 `D_ob` 特定的元素不能用 `p` 访问(除非用了类型转换)。

作为一个具体例子，考虑如下短程序，其定义了一个基本类 `B_class` 和一个派生类 `D_class`。派生类实现了一个简单的自动通讯录：

```
// Using pointers on derived class objects.

#include <iostream.h>
#include <string.h>

class B_class {
    char name[80];
public:
    void put_name(char *s) {strcpy(name, s); }
    void show_name() {cout << name << " ";}
};

class D_class : public B_class {
    char phone_num[80];
public:
```

```

void put_phone(char *num) {
    strcpy(phone_num, num);
}
void show_phone() {cout << phone_num << "\n";}
};

main(void)
{
    B_class *p;
    B_class B_ob;

    D_class *dp;
    D_class D_ob;

    p = &B_ob; // address of base

    // Access B_class via pointer.
    p->put_name("Thomas Edison");

    // Access D_class via base pointer.
    p = &D_ob;
    p->put_name("Albert Einstein");

    // Show that each name went into proper object.
    B_ob.show_name();
    D_ob.show_name();
    cout << "\n";

    /* Since put_phone and show_phone are not part of the
       base class, they are not accessible via the base
       pointer p and must be accessed either directly,
       or, as shown here, through a pointer to the
       derived type.
    */
    dp = &D_ob;
    dp->put_phone("555 555-1234");
    p->show_name(); // either p or dp can be used in this line
    dp->show_phone();
    return 0;
}

```

在这个例子中，指针被定义为 `B_class` 的指针，但它可以指向派生类 `D_class` 的一个对象并可以用来访问由基本类定义的派生类的元素。但要记住，一个基本指针在没有类型转换时不能访问派生类特定的元素。这就是为什么要用 `dp` 指针访问 `show_phone()` 的原因。`dp` 指针为指向派生类的指针。

若希望用基本类指针访问由一个派生类定义的元素，必须把它转换为派生类的指针。例如，下列代码行将适当地调用 `D_ob` 的 `show_phone()` 函数：

```
((D_class*)p)->show_phone();
```

外层括号对于由用 `p` 强制转换而不是由 `show_phone()` 返回的类型是必须的。在强制转换一个指针时，如果没有什么问题时，最好把它省去，因为它会增加代码的模糊性。

要清楚的另一点是，可以用一个基本指针指向派生对象的任一类型，相反却不是正确的，即，不能用一个派生类指针访问一个基本类对象。

最后一点是一个指针的增大和减小与其基本类相关，因此，当一个基本类指针指向一个派生类时，指针大小的增减不能使它指向派生类的另一个对象。因而，当一个指针指向派生对象时，应认为增大或减小该指针是非法的。

一个指向基本类的指针可用来指向从基本类派生的任何对象。这一事实是非常重要的

并且是 C++ 的基础。实际上, 你将会发现, 这是 C++ 实现运行时间多态性的途径之关键。

## 19.2 虚函数

运行时间多态性是通过使用派生类和虚函数来实现的。简而言之, 一个虚函数即是一个在一个基本类中被说明为 `virtual` 并在一个或多个派生类中被重定义的函数。`virtual` 函数特别之处在于当用一个指向派生类的一个对象的基本类指针访问一个虚函数时, C++ 根据指向的对象类型确定在运行时间调用哪一个函数。因此, 当指向不同的对象时, 执行该虚函数的不同版本。

一个虚函数是通过关键字 `virtual` 放在其基类说明的前面来加以说明的。当一个派生类重定义一个 `virtual` 函数时, 关键字 `virtual` 不用重复(尽管这样也没错)。

作为 `virtual` 函数的第一个例子, 考虑下列短程序:

```
// A short example that uses virtual functions.
#include <iostream.h>

class Base {
public:
    virtual void who() { // specify a virtual
        cout << "Base\n";
    }
};

class first_d : public Base {
public:
    void who() { // define who() relative to first_d
        cout << "First derivation\n";
    }
};

class second_d : public Base {
public:
    void who() { // define who() relative to second_d
        cout << "Second derivation\n";
    }
};

main(void)
{
    Base base_obj;
    Base *p;
    first_d first_obj;
    second_d second_obj;

    p = &base_obj;
    p->who(); // access Base's who

    p = &first_obj;
    p->who(); // access first_d's who

    p = &second_obj;
    p->who(); // access second_d's who

    return 0;
}
```

该程序产生下列输出:

```
Base
first derivation
second derivation
```

让我们来仔细地考察之以了解它是如何工作的。

正如你所看到的, 在 `Base` 中, 函数 `who()` 被说明为 `virtual`, 这就是说该函数由一个派生类重定义。在 `first_d` 和 `second_d` 中, `who()` 与每个类相关地被重定义, 在 `main()` 中, 有四个变量被说明——`base_obj`, 它是类 `Base` 的一个对象; `p`, 它是一个指向 `Base` 对象的指针, 以及 `first_obj` 和 `second_obj`, 它们是两个派生类的对象。然后, `p` 被赋以 `base_obj` 的地址, 并且 `who()` 函数被调用。由于 `who()` 被说明为 `virtual`, C++ 在运行时间通过 `p` 指向的对象类型来确定引用 `who()` 的哪一个版本。在这种情形下, 是 `Base` 类的一个对象。它即是被执行的在 `Base` 中说明的 `who()` 的版本。然后, `p` 被赋以 `first_obj` 的地址。记住, 一个基本类指针可能被用于访问任何派生类。现在, 当 `who()` 被调用时, C++ 再次确定什么对象类被 `p` 所指向以确定调用哪一个 `who()` 的版本。由于 `p` 指向类 `first_d` 的一个对象, 所以用该 `who()` 的版本。类似地, 当 `p` 被赋以 `second_obj` 的地址, 在 `second_d` 中说明的 `who()` 版本将被执行。

用虚函数实现运行时间多态性的关键之处是必须用指向该基本类的指针访问这些函数。尽管可以象调用任何其它成员函数那样显式地用对象名来调用一个虚函数, 但只有在用一个指向该基本类的指针访问一个虚函数时运行时间多态性才能实现。

在一个派生类中重定义一个虚函数是函数重载的一个特殊形式。但该术语没有在前面的讨论中使用, 其原因在于有几个限制, 正如你所知道的, 当重载一个一般函数时, 返回类型和参数的数量及类型可能是不相同的。但当重载一个虚函数时, 这些元素必须是不变的。若函数原型不同, 则函数被认为重载的, 并且其虚特性丢失。若仅仅函数的返回类型不同, 这是错误的(仅有其返回类型不同的函数在本质上是含糊的), 另一个限制是一个虚函数必须是定义类的一个成员, 而不是一个友元。但一个虚函数还可以是另一个类的友元; 另外, 析构函数允许是 `virtual` 的, 但构造函数不能。

由于这些限制以及重载一般函数和重载虚函数之间的差异, 术语“重载”用于描述虚函数的重定义。

一旦一个函数被说明为 `virtual`, 它将保持 `virtual` 特性, 而不管其经过了多少派生类层。例如, 若 `second_d` 由 `first_d` 而不是由 `Base` 派生, 如下列所示, `who()` 仍是 `virtual` 的, 并选用了其适当的版本:

```
// Derive from first_d, not Base
class second_d : public first_d {
public:
    void who() { // define who() relative to second_d
        cout << "Second derivation\n";
    }
};
```

当一个派生类不优于一个虚函数时, 则使用在基本类的函数版。例如, 前面程序的该版本为。

```
#include <iostream.h>

class Base {
```

```

public:
    virtual void who() {
        cout << "Base\n";
    }
};

class first_d : public Base {
public:
    void who() {
        cout << "First derivation\n";
    }
};

class second_d : public Base {
    // who() not defined
};

main(void)
{
    Base base_obj;
    Base *p;
    first_d first_obj;
    second_d second_obj;

    p = &base_obj;
    p->who(); // access Base's who()

    p = &first_obj;
    p->who(); // access first_d's who()

    p = &second_obj;
    p->who(); /* access Base's who() because
               second_d does not redefine it */

    return 0;
}

```

该程序现在有如下输出：

```

Base
First derivation
Base

```

记住，继承特性是层次性的。因此，若 `second_d` 由 `first_d` 而不是由 `Base` 派生，则当相对于类 `second_d` 的一个对象调用 `who()` 时，在 `first_d` 中声明的 `who()` 版本被调用，因为这是与 `second_d` 最近的类——而不是在 `Base` 中的 `who()`。

### 19.3 为什么使用虚函数

正如在本章开始所述的，与派生类结合的虚函数使得 C++ 支持运行时间多态性。多态性对于面向对象的程序设计是必不可少的一个原因：其不但允许一个派生类指定几个或全部一般函数的特定实现，而且还允许一个一般类指示对于任何该类派生对象都是公共的函数。有时这一观点表示如下：基本类指示任何从该类派生的对象都有的一般界面，但允许派生类定义实际的方法(method)，这就是为什么通常用“单界面，多方法”来描述多态性的原因。

成功地应用多态性的关键之一部分是了解基本类和派生类形成了一个层次，其从较高



的一般性变为较低的一般性(从基本类到派生类)。因此,当使用正确时,基本类可提供派生类可直接使用的所有元素,外加那些派生类必须在其自身上实现的函数,但由于界面的格式是由基本类定义的,任何派生类仍将共享该公用的界面。当设计适当时,使用虚函数,基本类可定义一般界面,其将被所有派生类使用。

此时,你可能会问,为什么多种实现的一致界面是重要的。答案又回到了面向对象程序设计的中心目标:帮助程序员控制更大复杂性的程序。例如,若你正确开发程序,会发现从某基本类派生的所有对象以同样的方法被访问,即使特定的行为从一个派生类型到另一个在变化。这就是说只需记住一个界面而不是几个界面。另外,界面与实现的分离支持类库的建立。若能正确实现这些库,则它们将提供一个公共界面,可以用之以派生自己的类以满足特定的需要。

要了解“单界面,多方法”概念的功能,考察下面的短程序,其建立了一个叫做 `figure` 的基本类。该类用于存放各二维对象的维并计算它们的域。函数 `set_dim()` 是一个标准成员函数,因为操作将对所有派生类是公用的。但 `show_area()` 被说明为 `virtual` 的,因为计算各对象的域的方法是不同的。该程序用 `figure` 派生两个特定的类,称作 `square` 和 `triangle`。

```
#include <iostream.h>

class figure {
protected:
    double x, y;
public:
    void set_dim(double i, double j) {
        x = i;
        y = j;
    }
    virtual void show_area() {
        cout << "No area computation defined ";
        cout << "for this class.\n";
    }
};

class triangle : public figure {
public:
    void show_area() {
        cout << "Triangle with height ";
        cout << x << " and base " << y;
        cout << " has an area of ";
        cout << x * 0.5 * y << ".\n";
    }
};

class square : public figure {
public:
    void show_area() {
        cout << "Square with dimensions ";
        cout << x << "x" << y;

        cout << " has an area of ";
        cout << x * y << ".\n";
    }
};

main(void)
{
    figure *p; /* create a pointer to base type */
```

```

    triangle t; /* create objects of derived type */
    square s;

    p = &t;
    p->set_dim(10.0, 5.0);
    p->show_area();

    p = &s;
    p->set_dim(10.0, 5.0);
    p->show_area();

    return 0;
}

```

通过考察该程序，将会发现，面向 square 和 triangle 的界面是相同的，即使它们对应计算对象域的方法是不同的。

给出了 figure 说明后，派生一个给定半径计算一个圆面积的 circle 类是否可能？答案是肯定的。所要做的是建立一个新的计算一个圆面积的派生类。virtual 函数的功能是基于你可以容易地派生一个新的类，该类仍将象其它相应对象那样共享公用界面。例如，下面是方法之一：

```

class circle : public figure {
public:
    void show_area() {
        cout << "Circle with radius ";
        cout << x;
        cout << " has an area of ";
        cout << 3.14 * x * x;
    }
};

```

在打算使用 circle 之前，仔细考察 show\_area() 的定义。注意，它只用了 x 的值。其假定存放半径(记住，用公式  $\pi r^2$  计算一个圆的面积)。但假定传递给函数 set\_dim() 的值不是一个而是两个，由于 circle 不需要第二个值，那该怎么办呢？

有两个办法解决这个问题，首先一个并且是最坏的一种方法是，在使用一个 circle 对象时，用一个哑值作为第二个参数调用 set\_dim()。这样做显得很草率，并要求记住这一特殊的例外，其违反了“单界面，多方法”的使用。

解决该问题一个较好的方法是给 set\_dim() 中的 y 参数一个缺省值，在这种方法中，当对应一个圆调用 set\_dim() 时，只需指定半径。当对应一个三角形或一个正方形调用 set\_dim() 时，应指定该两个值。下面是被扩展的程序：

```

#include <iostream.h>

class figure {
protected:
    double x, y;
public:
    void set_dim(double i, double j=0) {
        x = i;
        y = j;
    }
    virtual void show_area() {
        cout << "No area computation defined ";
    }
};

```

```

        cout << "for this class.\n";
    }
};

class triangle : public figure {
public:
    void show_area() {
        cout << "Triangle with height ";
        cout << x << " and base " << y;
        cout << " has an area of ";
        cout << x * 0.5 * y << ".\n";
    }
};

class square : public figure {
public:
    void show_area() {
        cout << "Square with dimensions ";
        cout << x << "x" << y;
        cout << " has an area of ";
        cout << x * y << ".\n";
    }
};

class circle : public figure {
public:
    void show_area() {
        cout << "Circle with radius ";
        cout << x;
        cout << " has an area of ";
        cout << 3.14 * x * x;
    }
};

main(void)
{
    figure *p; /* create a pointer to base type */

    triangle t; /* create objects of derived types */
    square s;
    circle c;

    p = &t;
    p->set_dim(10.0, 5.0);
    p->show_area();

    p = &s;
    p->set_dim(10.0, 5.0);
    p->show_area();

    p = &c;
    p->set_dim(9.0);
    p->show_area();

    return 0;
}

```

#### 19.4 纯虚函数及抽象类型

当派生类中没有被覆盖的一个虚函数被该派生类的一对象所调用时，就使用该函数在基类中定义的版本。但是，在许多情形下，基类中对虚函数的定义是无意义的。例如，在前例中所有的基类 `figure` 中，对 `show_area()` 的定义只是形式而已，它并不计算及显示任何类型对象的面积。当你创建自己的类库时，也会发现在其基类的环境中不存在一虚函数的

有意义之定义是很正常的。当出现这种情况时，有两种处理方法。第一是如例所示的那样报告一警告信息。虽然这方法在某些场合是有用的，但是它并非对所有情形都适用。例如，有可能存在着必须被派生类所定义的虚函数——以便使派生类具备意义。考虑类 `triangle`，如果 `show_area()` 未被定义，它就无意义。在这类情形中，需要确保派生类定义了所有必要函数的方法。C++ 对此问题的解决方法就是引入纯(pure)虚函数。

一纯虚函数是一个在基类中说明的函数。它没有相对于此基类的定义。因此，任何派生类都必须定义其自己的版本。为说明一纯虚函数，使用下列一般形式：

```
virtual type func_name(参数表)=0;
```

这里，`type` 是函数的返回类型，`func_name` 是函数名。例如，在 `figure` 的该版本中，`show_area()` 是一纯虚函数：

```
class figure {
protected:
    double x, y;
public:
    void set_dim(double i, double j=0) {
        x = i;
        y = j;
    }
    virtual void show_area() = 0; // pure
};
```

通过将一虚函数说明成纯的，就使任何派生类都定义其自己的实现。如果一类未能如此，Turbu C++ 就会报告一错误。例如，请编译此 `figure` 程序的修改版本，其中对 `show_area()` 的定义已从 `circle` class 中移去了：

```
/*
   This program will not compile because the class
   circle does not override show_area().
*/
#include <iostream.h>

class figure {
protected:
    double x, y;
public:
    void set_dim(double i, double j) {
        x = i;
        y = j;
    }
    virtual void show_area() = 0; // pure
};

class triangle : public figure {
public:
    void show_area() {
        cout << "Triangle with height ";
        cout << x << " and base " << y;
        cout << " has an area of ";
        cout << x * 0.5 * y << ".\n";
    }
};

class square : public figure {
public:
    void show_area() {
        cout << "Square with dimensions ";
```

```

        cout << x << "x" << y;
        cout << " has an area of ";
        cout << x * y << ".\n";
    }
};

class circle : public figure {
// no definition of show_area() will cause an error
};

main(void)
{
    figure *p; /* create a pointer to base type */

    triangle t; /* create objects of derived types */
    square s;

    p = &t;
    p->set_dim(10.0, 5.0);
    p->show_area();

    p = &s;
    p->set_dim(10.0, 5.0);
    p->show_area();

    return 0;
}

```

如果一个类至少有一个纯虚函数，那么就称该类为抽象的。抽象类有一重要性质：可以没有该类的对象。一个抽象类必须是能被用作其它类将要继承的基类。抽象类不能被用来说明一对象的原因当然是其函数中的一个或多个无定义。但是，即使基类是抽象的，仍可用它来说明指针。这是为支持运行时间多态性所需要的。

## 19.5 先期联编与迟后联编

在讨论面向对象程序设计语言时，有两个经常用到的术语：先期联编(early binding)与迟后联编(late binding)。对 C++ 来说，这些术语指的是发生在编译时间及运行时间的事件。

先期联编意思是一对象在编译时被连接至其函数。即，所有为确定将要调用哪一个函数所需要的信息在程序被编译时就已知道。例如，标准函数调用，重载函数调用，以及重操作符函数调用等就是先期联编。其主要的优点是效率高——速度快且所需内存小，缺点是缺乏灵活性。

迟后联编意思是一对象在运行时被连接至其函数。这意味着到底哪个函数将被调用需到运行时才能决定。C++ 中，迟后联编是通过使用虚函数及派生类来完成的。其优点是允许更大的灵活性。它被用来支持允许各种对象的——共用接口，这些对象利用该接口来定义其各自的实现。进一步，它还可被用来帮助建立自己的类库，该库可被重新使用及扩充。

用户程序使用先期联编还是迟后联编取决于程序的功能(实际上，大多数大型程序都使用两者的结合)。迟后联编是 C++ 强于 C 的一个最主要性质。但是，为此付出了程序速度会有些慢的代价。因此，最好只在它能真正改进程序结构及可维护性时才使用，但在需要用它时，应该毫不犹豫。

## 19.6 派生类中的构造函数及析构函数

由于 C++ 多态性的元素很大地取决于派生类，现在应当更详细地来对之加以考察。派生类的一个重要性质可在执行其构造及析构函数时看出来。先从构造函数开始。

有可能基类和派生类都有一构造函数。事实上，在多重继承的情形中，所有涉及到的类都可能具有构造函数。当派生类包含一个构造函数时，在执行它的构造函数前先执行基类的构造函数。例如，考虑下面短程序：

```
#include <iostream.h>

class Base {
public:
    Base() {cout << "\nBase created\n";}
};

class D_class1 : public Base {
public:
    D_class1() {cout << "D_class1 created\n";}
};

main(void)
{
    D_class1 d1;

    // do nothing but execute constructors
    return 0;
}
```

该程序建立一类型为 `D_class1` 的对象。它显示如下输出：

**Base created**

**D\_class1 created**

这里，`d1` 是类型 `D_class1` 的一对象，它是用 `Base` 派生的，因此，当创建 `d1` 时，首先执行 `Base()`，然后调用 `D_class1()`。

由于基类对派生类一无所知，任何它需要完成的初始化都显然又与派生类相分离，但可能是派生类的先决条件，因此，必须先执行基类。

另一方面，在派生类中的一析构函数是在基类中的析构函数之前被执行的。原因也是易于理解的，由于对基类的破坏隐含了对派生类的破坏，所派生的析构函数必须在它被破坏前被执行。下面这个程序说明了构造及析构函数被执行的顺序：

```
#include <iostream.h>

class Base {
public:
    Base() {cout << "\nBase created\n";}
    ~Base() {cout << "Base destroyed\n\n";}
};

class D_class1 : public Base {
public:
    D_class1() {cout << "D_class1 created\n";}
    ~D_class1() {cout << "D_class1 destroyed\n";}
};
```

```

main(void)
{
    D_class1 d1;

    cout << "\n";

    return 0;
}

```

它产生如下输出：

```

Base created
D_class1 created

D_class1 destroyed
Base destroyed

```

在创建另一派生类时，一派生类本身有可能被用作一基类。如出现这种情况，构造函数就以派生的次序而执行。析构函数则以相反次序而执行，例如，考虑下列使用 D\_class1 来派生 D\_class2 的程序：

```

#include <iostream.h>

class Base {
public:
    Base() {cout << "\nBase created\n";}
    ~Base() {cout << "Base destroyed\n\n";}
};

class D_class1 : public Base {
public:
    D_class1() {cout << "D_class1 created\n";}
    ~D_class1() {cout << "D_class1 destroyed\n";}
};

class D_class2 : public D_class1 {
public:
    D_class2() {cout << "D_class2 created\n";}
    ~D_class2() {cout << "D_class2 destroyed\n";}
};

main(void)
{
    D_class1 d1;
    D_class2 d2;

    cout << "\n";

    return 0;
}

```

它产生如下输出：

```

Base created
D_class1 created

Base created
D_class1 created
D_class2 created

D_class2 destroyed
D_class1 destroyed
Base destroyed

```

```
D_class1 destroyed
Base destroyed
```

### 19.7 多重基类

在创建一派生类型时，有可能说明不止一个的基类。为此，使用将被继承的用逗号隔开的类表。例如，考虑下列程序：

```
#include <iostream.h>

class Base1 {
public:
    Base1() {cout << "\nBase1 created\n";}
    ~Base1() {cout << "Base1 destroyed\n\n";}
};

class Base2 {
public:
    Base2() {cout << "Base2 created\n";}
    ~Base2() {cout << "Base2 destroyed\n";}
};

// multiple base classes
class D_class1 : public Base1, public Base2 {
public:
    D_class1() {cout << "D_class1 created\n";}
    ~D_class1() {cout << "D_class1 destroyed\n";}
};

main(void)
{
    D_class1 d1;

    cout << "\n";

    return 0;
}
```

在该程序中，D\_class 是从 Base1 及 Base2 中派生的。它产生如下输出：

```
Base1 created
Base2 created
D_class1 created

D_class1 destroyed
Base2 destroyed
Base1 destroyed
```

如你所见，当使用一基类表时，构造函数被调用的顺序是从左到右。析构函数被调用的顺序则是从右到左。



## 第二十章 使用 C++ 的 I/O 类库

自第三部分之始,本书一直在讨论用 C++ 重载操作符 >> 和 << 的控制台输入和输出。尽管 Turbo C++ 支持所有 C 的 I/O 函数集,前面的章节因讨论 C++ 的 I/O 操作而忽略了它们。这有一个主要的原因:用 I/O 的 C++ 方法能够帮助你以面向对象的方面进行思考。并看到“单界面、多方法”哲学的价值。在本章中,将对 C++ 的 I/O 系统有一个更多的了解,包括如何重载 << 和 >> 操作符以便能够输入和输出用户设计的类对象。C++ 的 I/O 系统是很大的,本章不可能论及每个函数及其特性,但将介绍最为重要并用得很普遍的函数及其特性。首先,让我们来看看为什么 C++ 要定义自己的 I/O 系统。

### 20.1 C++ 为何有自己的 I/O 系统

若你曾用其它语言进行过程序设计,便会知道 C 的 I/O 系灵活性大、功能性强(实际上,我们说在已有的所有结构化语言中, C 的 I/O 系统是无可比的,这话一点也只不过份)。C 提供了强有力的 I/O 函数,你可能会问,为什么 C++ 要定义其自己的 I/O 函数呢?而它在很大程度上与 C 中原有的函数重复。答案是由于 C 的 I/O 系统没有提供用户定义对象的支持。例如,在 C 中,若要建立结构:

```
struct my_struct {
    int count;
    char s[80];
    double balance;
} cust;
```

没有办法能够借用或延伸 C 的 I/O 系统以使之能直接在变量 my\_struct 上执行 I/O 操作。不能够建立一个用于类型 my\_struct 的数据定义的新格式指示符并将之应用于对 printf() 的调用。例如,下列式子不能工作:

```
printf("%my_struct", cust);
```

由于 printf() 只能识别内部类型,没有办法把其功能延伸到新的数据类型上。

但是,若用 C++ 方法进行 I/O,可以重载 << 和 >> 操作符以便能够识别用户所建立的类型。这包括在前面四章及文件 I/O 中用到的两个控制台 I/O 操作(控制台和文件 I/O 在 C++ 中像在 C 中那样连接并且就好比一个硬币的两面)。

虽然没有不能用 C 来实现的 C++ I/O 操作,但 C++ 系统能够识别用户定义类型这一特点大大地增加了其灵活性并避免了一些错误。要弄清楚为什么,请看一列对 printf() 的调用: printf("%d%s", "Hello", 10);

在这个调用中,参数表中串和整数的位置被颠倒了, %d 将与 Hello 对应, %s 将与 10 对应。在 C 中,这可能并不是一个错误(也许在某些极其不寻常的情景下,实际上或许希望用像上式那样的调用。总之, C 被设计成使你能够用汇编语言做任何事情),然而在大多数情况下,对 printf() 的如此调用是错误的。简而言之,在调用 printf() 时, C 不提供类型检查。但在 C++ I/O 中,所有内部类型的操作都定义与 << 和 >> 操作符有关,以至像上面 printf() 调用中的这种倒置就不会发生,而操作数类型能够自动确定正确的操作,该特性还可延伸到用户定义对象中。

## 20.2 C++流

C 和 C++ 的 I/O 系统的一个共同的重要东西：它们都对流进行操作，这在本书第二部分已介绍过(详见第十章)。C 和 C++ 的流是相似的，这就是说你了解的有关流的知识可应用于 C++ 中；而且更为重要的是，除了少量的例外，可以在同一程序中混用 C 和 C++ 的 I/O 操作。因而，可以无须修改每个 I/O 操作就可以把已有的 C 程序改为 C++ 程序。

### 20.2.1 C++预定义流

象在 C 中，当开始执行 C++ 程序时，C++ 含有几个自动打开的预定义流。它们是 `cin`、`cout`、`cerr` 和 `clog`。`cin` 是与标准输入对应的流，`cout` 是与标准输出对应的流，`cerr` 和 `clog` 流被连到标准输出上，`cerr` 和 `clog` 之间的区别是 `cerr` 没有被缓冲，因而，发送给它的任何输出都立即输出。相反，`clog` 被缓冲，只有当缓冲区满时才有输出。

缺省时，C++ 的标准流被连到控制台上，但它们也可能被程序引向别的设备或文件中；而且，它们还有可能被操作系统控制。

## 20.3 C++流类

Turbo C++ 的 I/O 系统被与流相关的类所定义。这些定义放在头文件 `IOSTREAM.H` 中。最低层类叫做 `streambuf`，它提供基本流操作，但不提供格式支持。其下一级类叫做 `ios`。`ios` 类为格式 I/O 提供基本操作，它还用来派生三个类：`istream`、`ostream` 和 `iostream` 它们可用来建立流。用 `istream` 可以建立一个输入流；用 `ostream` 可建立一个输出流；用 `iostream` 可建立一个能输入输出的流。

### 20.4 建立自己的插入符和抽取符

当一个程序需要输出或输入与一个类相连的数据时，将建立特定的成员函数，其仅有的用途是输出或输入该类数据。这种方法没有错误。C++ 将通过重载 `<<` 和 `>>` 操作符提供一种更佳的 I/O 操作类。

在 C++ 语言中，`<<` 操作符通常被认为是插入操作符，因为它在流中插入字符。类似地，`>>` 操作符被叫做抽取操作符，因为它从流中抽取字符。重载插入和抽取操作符的操作函数通常分别被称作插入函数和抽取函数。

插入和抽取操作符已被重载(在 `IOSTREAM.H` 中)，因而它们能够在任何 C++ 内部类型上执行流 I/O。但是，正如本章一开始所指出的，可以定义与自己建立的类相关的这些操作符，在本节中将看到如何定义。

#### 20.4.1 建立插入函数

C++ 最好的特性之一是你能够很容易地为自己建立的类生成插入函数。作为第一个简单的例子，让我们来为 `three_d` 类建立一个插入函数(第一次在第十八章中定义)，如下所示：

```
class three_d {
public:
    int x, y, z; // 3-d coordinates
    three_d(int a, int b, int c) {x=a; y=b, z=c;}
};
```

要为一个类 `three_d` 的对象建立一个插入函数，必须定义一个与之有关的插入操作。要做到这一点，必须重载 `<<` 操作符，如下所示：

```
// Display X, Y, Z coordinates (three_d's insertor).
ostream &operator<<(ostream &stream, three_d obj)
{
    stream << obj.x << ", ";
    stream << obj.y << ", ";
    stream << obj.z << "\n";
    return stream; // return the stream
}
```

让我们来仔细考察一下该函数，因为其许多特性都是所有插入函数所共有的。首先，注意它被说明为返回一个对类 `ostream` 的对象的引用。有必要允许几个这类插入符连在一起。另外，该函数有两个参数。第一个是对流的引用，其出现在 `<<` 操作符的左边。第二个参数是出现在右边的对象。在参数内部，类型 `three_d` 对象中的三个值被输出，`stream` 被返回。下面是一个说明该插入函数的短程序：

```
#include <iostream.h>

class three_d {
public:
    int x, y, z; // 3-d coordinates
    three_d(int a, int b, int c) {x=a; y=b; z=c;}
};

// Display X, Y, Z coordinates - three_d insertor
ostream &operator<<(ostream &stream, three_d obj)
{
    stream << obj.x << ", ";
    stream << obj.y << ", ";
    stream << obj.z << "\n";
    return stream; // return the stream
}

main(void)
{
    three_d a(1, 2, 3), b(3, 4, 5), c(5, 6, 7);

    cout << a << b << c;

    return 0;
}
```

若删去那些与 `three_d` 有关的代码，剩下的便是插入函数的框架，如下列式子：

```
ostream &operator<<(ostream &stream, class_type obj)
{
    // type specific code goes here
    return stream; // return the stream
}
```

你可能会奇怪为什么该函数不是下面的样子：

```
// Limited version - don't use.
ostream &operator<<(ostream &stream, three_d obj)
{
    cout << obj.x << ", ";
    cout << obj.y << ", ";
    cout << obj.z << "\n";
    return stream; // return the stream
}
```

在这个版本，cout 流是固定编入该函数中的。但记住，<<操作符可用于任何流。因此，若想在所有情形下正确工作，必须使用传递给函数的流。

在前面的程序中，重载插入函数不是 three\_d 的成员函数。实际上，插入和抽取函数都不能是一个类的成员。原因是一个操作函数是一个类的成员时，左操作数被假定为产生对操作函数调用的类的对象，没有别的办法能改变之。但当重载插入函数时，左参数是一个流，右参数是类的对象。因此，重载插入函数必须是非成员函数。

插入函数不能被定义成操作类的成员函数带来了一个严重问题：一个重载插入函数如何能够访问一个类的私有元素呢？在前面的程序中，变量 x、y 和 z 是公共的，插入函数能访问它们。数据的隐蔽性是 OOP 的一个重要部分，强制所有数据为公共的是一严重的问题，有一个解决的办法：一个插入函数可能定义为一个类的友元，它可访问该类的私有数据。考虑这个例子，three\_d 类和样本程序又在这里出现了，重载插入函数被说明为类的一个友元。

```
#include <iostream.h>

class three_d {
    int x, y, z; // 3-d coordinates - - now private
public:
    three_d(int a, int b, int c) {x=a; y=b, z=c;}
    friend ostream &operator<<(ostream &stream, three_d obj);
};

// Display X, Y, Z coordinates - three_d insertor
ostream &operator<<(ostream &stream, three_d obj)
{
    stream << obj.x << ", ";
    stream << obj.y << ", ";
    stream << obj.z << "\n";
    return stream; // return the stream
}

main(void)
{
    three_d a(1, 2, 3), b(3, 4, 5), c(5, 6, 7);

    cout << a << b << c;
    return 0;
}
```

注意，变量 x、y 现在是 three\_d 私有的，但它们仍可被插入函数直接访问。使插入函数（或抽取函数）为类的友元遵循了 OOP 的数据隐藏规则。

#### 20.4.2 重载抽取函数

要重载一个抽取函数，用如同重载一个插入函数那样的一般方法。例如，下面的抽取

函数输入 3-D 坐标。注意，它还给用户以提示：

```
// Get three dimensional values - extractor.
istream &operator>>(istream &stream, three_d &obj)
{
    cout << "Enter X,Y,Z values: ";

    stream >> obj.x >> obj.y >> obj.z;
    return stream;
}
```

抽取函数必须返回一个对类 `istream` 的对象的引用。并且第一个参数必须是对类 `istream` 的对象的引用，注意，第二个参数是一个引用，这是必须的，以便其值可被修改。

抽取符的一般格式如下：

```
istream &operator>>(istream &stream, object_type &obj)
{
    // put your extractor code here
    return stream;
}
```

下面是说明具有 `three_d` 类型的对象的抽取符的程序：

```
#include <iostream.h>

class three_d {
    int x, y, z; // 3-d coordinates
public:
    three_d(int a, int b, int c) {x=a; y=b, z=c;}
    friend ostream &operator<<(ostream &stream, three_d obj)
    friend istream &operator>>(istream &stream, three_d &obj);
};

// Display X, Y, Z coordinates - insertor.
ostream &operator<<(ostream &stream, three_d obj)
{
    stream << obj.x << ", ";
    stream << obj.y << ", ";
    stream << obj.z << "\n";
    return stream; // return the stream
}

// Get three dimensional values - extractor
istream &operator>>(istream &stream, three_d &obj)
{
    cout << "Enter X,Y,Z values: ";
    stream >> obj.x >> obj.y >> obj.z;
    return stream;
}

main(void)
{
    three_d a(1, 2, 3);

    cout << a;

    cin >> a;
    cout << a;
}
```

```
    return 0;
}
```

象插入函数那样，抽取函数不能为类成员。它们可以象例子中所述的那样，是友元或是简单的独立函数。

若不是必须返回一个对类 `istream` 的引用，可以在一个抽取函数中做任何想要做的事。由于结构和清晰的原因，最好对输入操作限制抽取符的行为。

## 20.5 格式化 I/O

可以使用 `printf()` 控制在屏幕上显示信息的格式。例如，可以指定域的宽度并作左右调整。用 C++ 方法进行 I/O，也可以完成同样的格式操作。有两种方法进行格式化输出。第一是使用 `ios` 类的成员函数；第二是使用称作操纵符 (manipulator) 的特殊类型函数。让我们首先来看用 `ios` 成员函数的格式化，然后再讨论操纵符。

### 20.5.1 用 `ios` 成员函数格式化

在 `IOSTREAM.H` 中，定义了如下枚举类型：

```
// formatting flags
enum {
    skipws = 0x0001,
    left = 0x0002,
    right = 0x0004,
    internal = 0x0008,
    dec = 0x0010,
    oct = 0x0020,
    hex = 0x0040,
    showbase = 0x0080,
    showpoint = 0x0100,
    uppercase = 0x0200,
    showpos = 0x0400,
    scientific = 0x0800,
    fixed = 0x1000,
    unitbuf = 0x2000,
    stdio = 0x4000
};
```

该枚举定义的值用于设置或清除标志，这些标志控制流信息格式化的方法。

当设置 `skipws` 后，在一个流上进行输入时，前导空白字符被丢弃。当清除 `skipws` 后，空白字符不被丢弃。

当设置 `left` 标志后，则输出左对齐；当设置 `right` 时，输出右对齐；当设置 `internal` 标志，则一个数字值将通过在符号或基本字符间插入空格以将之填入一个域中。

缺省时，数字值根据其基数输出。但你可以覆盖该缺省。例如，若要按十进制输出，设置 `dec` 标志；设置 `oct` 将使输出用八进制表示；设置 `showbase` 将使数字值表示以基数显示。

缺省时，当用科学表示法时，“e”用小写。当用十六进制值表示时，“x”用小写。当设置 `uppercase` 时，这些字符都用大写显示。

设置 `showpos` 将使正整数值前导加号得以显示。当设置 `showpoint` 将使小数字和其后的零全部显示——无论需要或不需要。

设置 `scientific` 标志，则浮点数字值用科学表示法显示；当 `fixed` 被设置，浮点值用—

般表示法显示, 并且缺省时, 保留到小数点后六位; 当没有设置标志时, 编译程序将选择一个适当的方法。

当设置 `unitbuf` 时, 将增进 C++ I/O 系统的功能, 但这已超过本书范围, 所以不将讨论。在 Turbo C++ 中, 该标志为缺省设置。

当 `stdio` 被设置时, 每次输出之后, 每个流都被刷新。流刷新将导致输出被写入连在该流上的物理设备。

格式标志存放在一个 `long` 整数中。要设置一个标志, 用 `setf()` 函数, 其最一般的格式为:

```
long setf(long flags);
```

该函数返回格式标志的原来设置并使 `flags` 指示的标志有效。例如, 要设置 `showbase` 标志, 可以使用如下语句:

```
stream.setf(ios::showbase);
```

这里, `stream` 是所要影响的流。例如, 下面程序设置 `showpos` 和 `scientific` 标志。

```
#include <iostream.h>

main(void)
{
    cout.setf(ios::showpos);
    cout.setf(ios::scientific);
    cout << 123 << " " << 123.23 << " ";

    return 0;
}
```

本程序的输出表示如下:

```
+123 +1.2323e+02
```

可以在一次调用中把任意多的标志或起来。例如, 可以修改程序, 把 `scientific` 和 `showpos` 或起来, 以便只对 `setf()` 调用一次, 如下所示:

```
cout.setf(ios::scientific | ios::showpos);
```

要清除一个标志, 用 `unsetf()` 函数, 其原型表示如下:

```
long unsetf(long flags);
```

该函数返回原来的标志设置并清除 `flags` 指示的标志。

有时, 了解当前的标志设置是很有用的。可以用 `flags()` 函数提取当前标志值, `flags()` 函数原形如下:

```
long flags(void);
```

该函数返回与流相关的标志的当前值。

下列 `flags()` 的格式把标志值置为 `flags` 的标志并返回标志的原值:

```
long flags(long flags);
```

要搞清楚 `flags()` 和 `unsetf()` 是如何工作的, 请看下列程序。它包含称作 `showflags()` 的函数, 其显示标志的状态:

```
#include <iostream.h>

void showflags(long f);
main(void)
{
    long f;
```

```

    f = cout.flags();

    showflags(f);
    cout.setf(ios::showpos);
    cout.setf(ios::scientific);

    f = cout.flags();
    showflags(f);

    cout.unsetf(ios::scientific);

    f = cout.flags();
    showflags(f);

    return 0;
}

void showflags(long f)
{
    long i;

    for(i=0x4000; i; i = i >> 1)

        if(i & f) cout << "1 ";
        else cout << "0 ";

    cout << "\n";
}

```

运行后，程序产生如下输出：

```

0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 1
0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 1

```

除了格式化标志之外，可设置域宽，填充字符和小数点后保留的有效数字，用下列函数：

```

int width(int len);
char fill(char ch);
int precision(int num);

```

**width()**函数返回当前域宽并把域宽设置为 *len*。缺省时，域宽随用来表示数据的字符个数而变化。**fill()**函数返回当前填充字符，缺省时是一空格，并使当前填充字符与 *ch* 相同。填充字符是用于把输出填充到指定域宽中的字符，**precision()**函数返回显示在小数点后的数字数，并把值设置为 *num*。下面是说明这三个函数的程序：

```

#include <iostream.h>

main(void)
{
    cout.setf(ios::showpos);
    cout.setf(ios::scientific);
    cout << 123 << " " << 123.23 << "\n";

    cout.precision(2); // two digits after decimal point
    cout.width(10);    // in a field of ten characters
    cout << 123 << " " << 123.23 << "\n";
}

```



```

    cout.fill('#'); // fill using #
    cout.width(10); // in a field of ten characters
    cout << 123 << " " << 123.23;

    return 0;
}

```

程序输出为

```

+123 +1.2323e+02
+123 +1.23e+02
##### +123 +1.23e+02

```

## 20.5.2 使用操纵函数

C++ I/O 系统还有另一种改变一个流中格式参数的方法，这种方法使用称作操纵符的特殊函数，它可以包含在 I/O 语句中。标准操纵函数如表 20-1 所示。要访问它们必须在程序中包含 `IOMANIP.H`。

表 20-1 标准操纵函数

操纵函数	用途	输入/输出
<code>dce</code>	格式化十进制数字数据	输入和输出
<code>endl</code>	输出一新行字符并刷新流	输出
<code>ends</code>	输出一个空字符	输出
<code>flush</code>	刷新一个流	输出
<code>hex</code>	格式化十六进制数字数据	输入和输出
<code>oct</code>	格式化八进制数字数据	输入和输出
<code>resetioflags(long f)</code>	清除 <i>f</i> 指定的标志	输入和输出
<code>setbase(int base)</code>	把数字基置为 <i>base</i>	输出
<code>setfill(int ch)</code>	把填充字符置为 <i>ch</i>	输入和输出
<code>setiosflagw(long f)</code>	设置 <i>f</i> 指定的标志	输入和输出
<code>setprecision(int p)</code>	设置小数点后显示的数字数	输入和输出
<code>setw(int w)</code>	把域宽置为 <i>w</i>	输入和输出
<code>wsn</code>	跳过引导空格符	输入

所有操纵函数都把它们刚作用的流作为参数并返回该流，在这种方法中，一个操纵函数可用作一条 I/O 表达式的部分，下列是一个程序例子，它用操纵格式改变输出格式：

```

#include <iostream.h>
#include <iomanip.h>

main(void)
{
    cout << setprecision(2) << 1000.243 << endl;
}

```

```

    cout << setw(20) << "Hello there.";
    return 0;
}

```

程序输出为:

1000.24

Hello there.

注意在 I/O 操作链中操纵是如何出现的, 还要注意当操纵函数没有参数时, 就像 `endl` 中, 其后面不跟标号。这是因为传递给重载 `<<` 操作符的是函数的地址。

下列程序用 `setiosflag()` 设置 `scientific` 和 `showpos` 标志;

```

#include <iostream.h>
#include <iomanip.h>

main(void)
{
    cout << setiosflags(ios::showpos);
    cout << setiosflags(ios::scientific);
    cout << 123 << " " << 123.23;

    return 0;
}

```

下列程序用 `ws` 在把串输入到 `s` 中时跳过引导空格空符:

```

#include <iostream.h>

main(void)
{
    char s[80];

    cin >> ws >> s;
    cout << s;
}

```

### 20.5.3 建立自己的操纵函数

你可以建立自己的操纵函数。其中最简单的一种是不带参数的, 下面将介绍如何建立它们(建立带参数的操纵函数已超出了本书的范围)。

所有不带参数的输出操纵函数均有以下形式:

```

ostream &manip_name(ostream &stream)
{
    //your code
    return stream;
}

```

其中, `manip_name` 是操纵函数的名字。虽然操纵函数带有一个指向被操作流的指针的参数, 但要记住它用于输出操作时不带参数, 这一点是很重要的。

以下程序建立一个名字为 `setup()` 的操纵函数, 它设置左对齐标志, 置域宽为 10, 并定义填充符为 "\$":

```

#include <istream.h>
#include <iomanip.h>
ostream & setup(ostream &stream)
{
    stream.setf(ios::left);
    stream << setw(10) << setfill('$');
    return stream;
}
main(void)
{
    cout<< 10 << " " << setup << 10;
    return 0;
}

```

操纵函数之所以有用的原因有两个：第一，当要对预先没有定义操纵函数的设备(如绘图仪)进行 I/O 操作时，定义自己的操纵函数将使得向这类设备的输出变得方便；第二，当多次重复相同的操作序列时，可以像前面程序所述的那样，将这些操作合并在一个操纵函数中。

所有不带参数的输入操纵函数均有以下形式：

```

istream &manip_name(istream &stream)
{
    //your code
    return stream;
}

```

例如，以下程序建立一个名字为 `prompt()` 的操纵函数，它将输入转换为十六进制，并提示你以十六进制输入：

```

#include <iostream.h>
#include <iomanip.h>
istream & prompt(istream &stream)
{
    cin>> hex;
    cout<<"Enter number using hex format:";
    return stream;
}
main (void)
{
    int i;
    cin>>prompt>>i;
    cout << i;
    return 0;
}

```

## 20.6 文件 I/O

可以用 C++ I/O 系统执行文件 I/O。尽管最后结果是相同的, C++ 的文件 I/O 与 ANSI C 的 I/O 系统还是有区别的。由于这个原因, 我们特别地在本章对之作讨论。

### 20.6.1 打开和关闭文件

在 C++ 中, 一个文件是通过把它连到流中来打开的。有三种类型的流: 输入、输出和输入/输出。要打开一个输入流, 必须说明该流有类型 `ifstream`; 要打开一个输出流, 其必须被说明有类型 `ofstream`; 要执行输入和输出操作的流必须被说明为类型 `fstream`。例如, 下一程序段建立一个输入流、一个输出流以及能输入/输出的流:

```
ifstream in;    //input
ofstream out;   //output
fstream both;   //input and output
```

一旦建立了一个流, 把它与一个文件相连的一种方法是使用函数 `open()`。该函数是这三种流的成员。其原型如下:

```
void open(char *filename, int mode, int access);
```

这里, `filename` 是文件名, 其可以包含一个路径指示符。 `mode` 的值确定如何打开文件, 其必须是下列值(在 `FSTREAM.H` 中定义)之一(或更多):

```
ios::app
ios::ate
ios::in
ios::nocreate
ios::noreplace
ios::out
ios::trunc
```

可以把这些中的两个或多个值用 `OR` 连起来。让我们来看看这些值的含意。

包含 `ios::app` 使得把所有输出到文件的内容添加到文件末尾。该值只可以与能输出的文件一起用。包含 `ios::ate` 是为了使得当文件被打开时, 对文件结尾进行搜索。

`ios::in` 指定能输入的文件, `ios::out` 指定能输出的文件。但用 `ifstream` 建立一个流隐含着输入, 用 `ofstream` 建立一个流隐含着输出, 因而, 在这种情形下, 没有必要提供这些值。

包含 `ios::nocreate` 使 `open()` 函数在文件不存在的时候失败。 `ios::noreplace` 值使 `open()` 函数在文件已存在时失败。

`ios::trunc` 值使具有同名的已存在文件的内容删去并把该文件长度变为零。

`access` 值确定如何访问文件, 该值与 DOS 文件特性代码对应, 表示如下:

特性	含意
0	一般文件——打开访问
1	只读文件
2	隐含文件

4                                   系统文件  
8                                   档案位设置

可以把上述两项或两次以上用 OR 连起来。

下列程序段打开一个一般输出文件:

```
ofstream out;  
  
out.open("test", ios::out, 0);
```

但你将会很少(若有的话)看到象上面的 `open()` 调用, 因为 `mode` 和 `access` 参数都有缺省值。对于 `ifstream` 和 `ofstream`, `mode` 参数都有一个缺省值, 对于 `ifstream`, 其为 `ios::in`, 对于 `ofstream`, 其为 `ios::out`, `access` 参数还有一个缺省值 0 (一般文件)。因而, 前面的语句通常有下列形式:

```
out.open("test");     //defaults to output and normal file
```

要为输入和输出打开一个流, 必须指定 `ios::in` 和 `ios::out` 的 `mode` 值, 如下所示:

```
fstream mystream;  
  
mystream.open("test", ios::in | ios::out);
```

若 `open()` 失败, `mystream` 将为 0。

尽管用 `open()` 函数打开一个文件是完全适当的, 但大多数时候你不会这样做, 因为 `ifstream`、`ofstream` 和 `fstream` 类有自动打开文件的构造函数。构造函数有与 `open()` 函数相同的参数和缺省值。因此, 打开文件最常用的方法如下:

```
ifstream mystream("myfile");     //open file for input
```

若由于某种原因文件不能打开, 相应的流变量值将为零。因此, 为确认文件确实被打开, 可使用如下代码:

```
ifstream mystream("myfile"); // open file for input  
if(!mystream) {  
    cout << "cannot open file";  
    // process error  
}
```

要关闭一个文件, 可用成员函数 `close()`。例如, 关闭连接到称作 `mystream` 流的文件。用下列语句:

```
mystream.close()
```

`close()` 函数没有参数且没有返回值。

## 26.8.2 读写文本文件

对一个文本文件进行读写是很简单的事, 因为只需用 `<<` 和 `>>` 操作符即可。例如, 下列程序把一个整数、一个浮点值和一个串写到 TEST 文件中:

```
#include <iostream.h>  
#include <fstream.h>  
  
main(void)  
{  
    ofstream out("test");  
    if(!out) {  
        cout << "Cannot open file";  
    }
```

```

        return 1;
    }

    out << 10 << " " << 123.23 << "\n";
    out << "This is a short text file.";

    out.close();

    return 0;
}

```

下列程序从上面程序建立的文件中读入一个整数、一个 float、一个字符和一个串：

```

#include <iostream.h>
#include <fstream.h>

main(void)
{
    char ch;
    int i;
    float f;
    char str[80];

    ifstream in("test");
    if(!in) {
        cout << "Cannot open file";
        return 1;
    }

    in >> i;
    in >> f;
    in >> ch;
    in >> str;

    cout << i << " " << f << " " << ch << "\n";
    cout << str;

    in.close();
    return 0;
}

```

当用>>操作符读文本文件时，注意将会发生字符转换。例如，空格字符被略去。若希望避免字符转换，必须使用 C++ 的库 I/O 函数，其将在下一章中讨论。

### 20.6.3 二进制 I/O

有两种对文件进行读写二进制的操作，第一，可以用成员函数 put() 写一个字节，用成员函数 get() 读一个字节。get() 函数有许多形式，但用得最多的是下列形式(以及 put()：

```

istream &get(char &ch);
ostream &put(char ch);

```

get() 函数从一个相连流中读入一个字符，把值放入 ch 中，并返回该流。put() 函数把 ch 写到流中并返回流。

以下程序将在屏幕上显示任何文件的所有内容。它使用 get() 函数：

```

#include <iostream.h>
#include <fstream.h>

```

```

main(int argc, char *argv[])
{
    char ch;

    if(argc!=2) {
        cout << "Usage: PR <filename>\n";
        return 1;
    }

    ifstream in(argv[1]);

    if(!in) {
        cout << "Cannot open file";
        return 1;
    }

    while(in) { // in will be 0 when eof is reached
        in.get(ch);
        cout << ch;
    }

    return 0;
}

```

当 `in` 到达文件尾，其将为零。引起 `while` 循环停止。

实际上有一个更为紧凑的方法编写读文件和显示文件的循环，如下所示：

```

while(in.get(ch))
    cout << ch;

```

这是因为 `get()` 返回流 `in`，并且当碰到文件末时，`in` 将为零。

以下程序用 `put()` 把一个串写到一个文件中：

```

#include <iostream.h>
#include <fstream.h>

main(void)
{
    char *p = "hello there";

    ofstream out("test");
    if(!out) {
        cout << "Cannot open file";

        return 1;
    }

    while(*p) out.put(*p++);

    out.close();

    return 0;
}

```

要读写一个二进制数据块，用 C++ 的 `read()` 和 `write()` 成员函数。其原型如下：

```

istream &read(unsigned char *buf, int num);
ostream &write(const unsigned char *buf, int num);

```

`read()` 函数从相连的流中读入 `num` 字节，并把它们放入 `buf` 指示的缓冲区。`write()` 函数把 `buf` 指示的缓冲区 `num` 字节写到相连的流中。

下列程序写然后读一整数数组：

```

#include <iostream.h>
#include <fstream.h>

main(void)
{
    int n[5] = {1, 2, 3, 4, 5};
    register int i;

    ofstream out("test");
    if(!out) {
        cout << "Cannot open file";
        return 1;
    }

    out.write((unsigned char *) &n, sizeof n);

    out.close();

    for(i=0; i<5; i++) // clear array
        n[i] = 0;

    ifstream in("test");
    in.read((unsigned char *) &n, sizeof n);

    for(i=0; i<5; i++) // show values read from file
        cout << n[i] << " ";

    in.close();

    return 0;
}

```

注意，当对一个没有定义为字符数组的缓冲区进行操作时，在 `read()` 和 `write()` 调用中的类型转换是必要的。

若在 `num` 个字符读出之前到达文件末尾，则 `read()` 停止并且缓冲区尽可能多地含有有效字符。可以用另一个成员函数 `gcount()` 来了解有多少字符被读出，该函数的原型为：

```
int gcount();
```

其返回最后一次二进制读操作所读出的字符数。

#### 20.6.4 检测 EOF

可以用成员函数 `eof()` 检查是否到达文件末尾，其原型为：

```
int eof();
```

当到达文件末尾时，它返回非零。否则，返回零。

#### 20.6.5 随机访问

在 C++ 的 I/O 系统中，可用 `seekg()` 和 `seekp()` 函数进行随机访问。其一般形式为：

```
istream &seekg(streamoff offset, seek_dir origin);
```

```
ostream &seekp(streamoff offset, seek_dir origin);
```

这里，`streamoff` 是定义在 `IOSTREAM.H` 中的一个类，`IOSTREAM.H` 能够含有 `offset` 所能具有的最大合法值。`seek_dir` 是一个枚举。其可以有如下列值：

```
ios::beg
```



```
ios::cur
```

```
ios::end
```

C++ I/O 系统管理与一个文件相连的两个指针。一个是读指针，其指示下一个输入操作将发生在文件中的何处。另一个是写指针，其指示下一个输出操作将发生在文件中的何处。每次进行输入或输出操作时，相应的指针自动变化，但用 `seekg()` 和 `seekp()` 函数，可使对文件的访问采用非连续方式。

`seekg()` 函数把与文件相连的 `get` 指针从 `origin` 处移动 `offset` 个字节数，`origin` 必须是下列三个值之一：

<code>ios::beg</code>	文件首
<code>ios::cur</code>	当前位置
<code>ios::end</code>	文件尾

`seekp()` 函数把与文件相连的 `put` 指针从 `origin` 处移动 `offset` 个字节数，`origin` 必须是上述三个值之一。

下面说明 `seekp()` 函数。它允许在命令行指定一个文件名，该命令行之后跟有要改变的文件中的特定字节，然后其在指定位置写一个“X”：

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
    if(argc!=3) {
        cout << "Usage: CHANGE <filename> <byte>\n";
        return 1;
    }

    ofstream out(argv[1]);
    if(!out) {
        cout << "Cannot open file";
        return 1;
    }

    out.seekp(atoi(argv[2]), ios::beg);

    out.put('X');
    out.close();

    return 0;
}
```

下列程序用 `seekg()`。它从指定的位置开始显示一个文件中的内容：

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
    char ch;

    if(argc!=3) {
        cout << "Usage: PR <filename> <starting location>\n";
    }
}
```

```

    return 1;
}

ifstream in(argv[1]);
if(!in) {
    cout << "Cannot open file";
    return 1;
}

in.seekg(atoi(argv[2]), ios::beg);

while(in.get(ch))
    cout << ch;

return 0;
}

```

可以用下列函数确定每个文件指针的当前位置。

**streampos tellg( );**

**streampos tellp( );**

这里，**streampos** 是定义在 **IOSTREAM.H** 中的类。**IOSTREAM.H** 能够容纳上述两函数返回的最大值。

C++ 的 I/O 系统功能强、灵活性大。尽管本章讨论了最为重要的并且用得最多的函数，C++ 还包括其它 I/O 函数。可以查阅其它手册，看看 C++ 系统中还有哪些函数。

## 第二十一章 其它 C++ 内容

若你已读过并试过前 20 章中的例子，则可以肯定把自己称作一个 C++ 程序员。本书最后一章讨论前面章节中没有讨论过的几个方面。其还考察了一些 C 和 C++ 的不同之处，以及一些设计哲学。

### 21.1 用 new 与 delete 进行动态分配

C 用函数 `malloc()` 和 `free()` 动态分配存储空间及动态释放已分配空间。但 C++ 含有两个操作符。它们执行以更好更易的方式分配和释放存储空间的函数。这两个操作符是 `new` 和 `delete`。其一般格式为：

```
pointer_var = new var_type;
delete pointer_var;
```

这里，`pointer_var` 是一个类型 `var_type` 的指针。操作符分配足够的存储空间存放一个类型 `var_type` 的值并返回其地址。任何合法数据都可用 `new` 分配。`delete` 操作符释放 `pointer_var` 指向的存储空间。

像 `malloc()` 一样，若分配要求失败，`new` 返回一个空指针。因此，在使用之前，必须检查 `new` 生成的指针。而且像 `malloc()` 那样，`new` 从堆中分配存储空间。

动态分配的管理方法限制了 `delete` 函数只能与指向用 `new` 分配的存储空间的指针一起使用。用 `delete` 时使用任何其它地址类型将导致严重错误。

使用 `new` 比用 `malloc()` 有几个优点。第一，`new` 自动计算待分配类型的大小。而不用使用 `sizeof` 操作符。这就较为省事。更重要的是，其避免偶然地分配错误存储量。第二，它自动返回正确的指针类型——不必进行类型转换。第三，正如你将要看到的，它可以用 `new` 对待分配对象进行初始化。最后，它可以重载与一个类相关的 `new` (和 `delete`)。

下面是 `new` 和 `delete` 的一个简单例子：

```
#include <iostream.h>

main(void)
{
    int *p;

    p = new int; // allocate memory for int
    if(!p) {
        cout << "allocation failure\n";
        return 1;
    }

    *p = 20; // assign that memory the value 20
    cout << *p; // prove that it works by displaying value

    delete p; // free the memory

    return 0;
}
```

该程序把一存储区地址赋给 `p`。存储区足够能容纳一个整数。然后，把值 20 赋给该存

储区并把其内容显示在屏幕上。最后，释放该动态分配的存储区。

如前所述，可以用 `new` 操作符初始化存储区。为此，在类型名之后在括号中指定初始化值。例如，下列程序用初始化过程把值 99 分配给 `int` 类型的存储区：

```
#include <iostream.h>

main(void)
{
    int *p;

    // ... (omitted) ...

    return 1;
}

cout << *p;

delete p;

return 0;
}
```

可以用 `new` 分配数组。对于一维数组来说，一般格式为：

```
pointer_var=new var_type[size];
```

这里，`size` 指定数组中的元素个数。

当释放一个动态分配的数组时，必须使用 `delete` 的下列格式：

```
delete[size]pointer_var;
```

`size` 指定数组中的元素个数。必须指定数组长度的原因是当释放一个含有对象的数组空间时，对于每个对象必须执行析构函数(若存在的话)(很快将看到这样的例子)。

下面程序为十个浮点数分配空间。赋值 100 到 109，并在屏幕上显示数组内容：

```
#include <iostream.h>

main(void)
{
    float *p;
    int i;

    p = new float [10]; // get a 10-element array
    if(!p) {
        cout << "allocation failure\n";
        return 1;
    }

    // assign the values 100 through 109
    for(i=0; i<10; i++) p[i] = 100.00 + i;

    // display the contents of the array
    for(i=0; i<10; i++) cout << p[i] << " ";

    delete [10] p; // delete the entire array

    return 0;
}
```

分配一个数组要记住的重要一点是不能对之进行初始化。

可以为任何合法类型分配存储空间。这包括对象。例如，在下面程序中，`new` 为一个类型为 `three_d` 的对象分配存储空间：

```
#include <iostream.h>

class three_d {
public:
    int x, y, z; // 3-d coordinates
    three_d(int a, int b, int c);
    ~three_d() {cout << "destructing\n";}
};

three_d::three_d(int a, int b, int c)
{
    cout << "constructing\n";
    x = a;
    y = b;
    z = c;
}

// Display X, Y, Z coordinates - three_d inserter
ostream &operator<<(ostream &stream, three_d obj)
{
    stream << obj.x << ", ";
    stream << obj.y << ", ";
    stream << obj.z << "\n";
    return stream; // return the stream
}

main(void)
{
    three_d *p;

    p = new three_d (5, 6, 7);
    if(!p) {
        cout << "allocation failure\n";
        return 1;
    }

    cout << *p;

    delete p;

    return 0;
}
```

注意，本程序中利用了插入函数。运行程序时，将看到当遇到 `new` 时，`three_d` 的构造函数被调用，当遇到 `delete` 时，析构函数被调用，并且注意，初始化值被 `new` 自动传递给构造函数。

如下列程序所示，可以用 `new` 对一个用户定义对象的数组分配空间：

```
#include <iostream.h>
class three_d {
public:
    int x, y, z; // 3-d coordinates
    three_d(int a, int b, int c);
    three_d(){cout << "constructing\n";} // needed for arrays
```

```

    ~three_d() {cout << "destructing\n";}
};

three_d::three_d(int a, int b, int c)
{
    cout << "constructing\n";
    x = a;
    y = b;
    z = c;
}

// Display X, Y, Z coordinates - three_d insertor
ostream &operator<<(ostream &stream, three_d obj)
{
    stream << obj.x << ", ";
    stream << obj.y << ", ";
    stream << obj.z << "\n";
    return stream; // return the stream
}

main(void)
{
    three_d *p;
    int i;

    p = new three_d [10];
    if(!p) {
        cout << "allocation failure\n";
        return 1;
    }

    for(i=0; i<10; i++) {
        p[i].x = 1;
        p[i].y = 2;
        p[i].z = 3;
    }

    for(i=0; i<10; i++) cout << *p;

    delete [10] p;

    return 0;
}

```

注意，第二个构造函数被加到 `three_d` 类中。由于给数组分配空间时不能进行初始化，所以需要有一个没有参数的构造函数。若没有提供这样的构造函数，将有错误信息显示。在本例中，这个构造函数没有做任何事。但在其它类中，它可能要完成某些工作。

### 21.1.1 重载 new 和 delete

重载 `new` 和 `delete` 是可能的。要这么做的一个原因是希望使用某种特别的分配方法。例如，可能希望有一些分配子程序，它们能在堆已耗尽时自动开始把一个磁盘文件当作虚存储区使用，不管是什么原因，重载这些操作符是件简单的事。

重载 `new` 和 `delete` 的函数之框架如下：

```

void *operator new(size_t size)
{
    // perform allocation
    return pointer_to_memory;
}

void operator delete(void *p)
{
    // free memory pointed to by p
}

```

类型 `size_t` 被 Turbo C++ 定义为能够容纳可分配的单一的存储块的最大值。它是一个整型类型。参数 `size` 将含有需要存放待分配对象的字节数。重载 `new` 函数必须返回一个指向分配存储空间的指针或若分配出错，返回零。除这些限制之外，重载 `new` 函数能作任何要求它做的事。

`delete` 函数接收一个指向存储区的指针，并将其释放。

`new` 和 `delete` 操作符可以全局性地重载，以使所有这些操作符的使用调用此版本，或者它们可以相对于一个或多个类重载。让我们首先来看一个与 `three_d` 相关的 `new` 和 `delete` 的重载例子。为清楚起见，不用新的分配方案。

要重载与一个类相关的 `new` 和 `delete` 操作符，只需使重载操作符函数成为类成员。例如，下面是关于 `three_d` 类重载 `new` 和 `delete` 操作符：

```

#include <iostream.h>
#include <stdlib.h>

class three_d {
public:
    int x, y, z; // 3-d coordinates
    three_d(int a, int b, int c) ;
    ~three_d() {cout << "destructing\n";}
    void *operator new(size_t size);
    void operator delete(void *p);
};

three_d::three_d(int a, int b, int c)
{
    cout << "constructing\n";
    x = a;
    y = b;
    z = c;
}

// Overload new relative to three_d
void * three_d::operator new(size_t size)
{
    cout << "in three_d new\n";
    return malloc(size);
}

// Overload delete relative to three_d
void three_d::operator delete(void *p)
{
    cout << "in three_d delete\n";
    free(p);
}

```

```

// Display X, Y, Z coordinates - three_d inserter
ostream operator<<(ostream &stream, three_d obj)
{
    stream << obj.x << " ";
    stream << obj.y << " ";
    stream << obj.z << "\n";
    return stream; // return the stream
}

main(void)
{
    three_d *p, *pl;

    p = new three_d (1, 2, 3);
    pl = new three_d (4, 5, 6);
    if(!p || !pl) {
        cout << "allocation failure\n";
        return 1;
    }

    cout << *p << *pl;

    delete p;
    delete pl;

    return 0;
}

```

注意当 `new` 和 `delete` 相对一特定的类重载时，这些操作符在其它数据类型上将使用其原先的版本。重载操作符仅用于那些为之定义的类，这就是说若把下列行加到 `main()` 中，将执行全局 `new`：

```
int *l=new int;
```

可以通过在任何类说明之外重载 `new` 和 `delete` 使它们成为全局的。当 `new` 和 `delete` 被全局重载时，C++ 原来的 `new` 和 `delete` 被忽略并且该重载的操作符用于所有分配要求。当然，若已相对于一个或多个类定义了 `new` 和 `delete` 版本，则当分配相应类对象时使用特定类版本。换句话说，当遇到 `new` 或 `delete` 时，编译器首先检查它们是否相对于正在操作的类定义的，若是这样，则该特定版本被使用，若不是这样，C++ 将使用全局定义的 `new` 和 `delete`。倘若它们已重载，则重载版本将被使用。

要看一个全局重载 `new` 和 `delete` 的例子，请看下列程序：

```

#include <iostream.h>
#include <dlfcn.h>

class three_d {
public:
    int x, y, z; // 3-d coordinates
    three_d(int a, int b, int c) {
        *three_d() {cout << "destructing\n";}
    }
    three_d::three_d(int a, int b, int c)
    {
        cout << "constructing\n";
        x = a;
        y = b;
    }
};

```



```

    z = c;
}

// Overload new globally.
void * operator new(size_t size)
{
    cout << "in new new\n";
    return malloc(size);
}

// Overload delete globally.
void operator delete(void *p)
{
    cout << "in new delete\n";
    free(p);
}

// Display X, Y, Z coordinates - three_d inserter
ostream &operator<<(ostream &stream, three_d obj)
{
    stream << obj.x << ", ";
    stream << obj.y << ", ";
    stream << obj.z << "\n";
    return stream; // return the stream
}

main(void)
{
    three_d *p, *pl;

    p = new three_d (1, 2, 3);
    pl = new three_d (4, 5, 6);
    if(!p || !pl) {
        cout << "allocation failure\n";
        return 1;
    }

    cout << *p << *pl;

    delete p;
    delete pl;

    int *i = new int;
    if(!i) {
        cout << "allocation failure\n";
        return 1;
    }

    *i = 10;
    cout << *i << "\n";
    return 0;
}

```

应运行一下该程序以证明内部 new 和 delete 操作符已实际被重载。

## 21.2 静态类成员

关键字 static 可用于一个类的成员，其含义与 C 中的含义相似。当把一个类的成员说明为 static 时，即是在告诉编译程序无论有多少类的对象被建立，只有 static 成员的一个

拷贝。一个 static 成员被所有类对象共享。当其类的第一个对象被建立时，所有 static 数据被初始化为零，并且不做其它初始化。

作为第一个例子。考察下列程序并理解它是如何工作的

```
#include <iostream.h>

class counter {
    static int count;
public:
    void setcount(int i) {count = i;};
    void showcount() {cout << count << " ";}
};

main(void)
{
    counter a, b;

    a.showcount(); // prints 0
    b.showcount(); // prints 0

    a.setcount(10); // set static count to 10

    a.showcount(); // prints 10
    b.showcount(); // also prints 10
}
```

Turbo C++ 把 count 初始化为 0。这就是为什么第一次对 showcount() 的调用都显示 0。对象 a 把 count 置为 10，然后，a 和 b 都用 showcount 显示其值。由于 a 和 b 共享一个 count 的拷贝，它们都显示 10。

记住，当把一个类的成员说明成 static 时，即是在建立该成员的一个拷贝然后被所有该类的对象共享。

还可以有 static 成员函数，当一个成员函数被说明为 static 时，则其仅有一个拷贝存在而且被其所属的所有对象使用，static 成员函数访问所有静态数据以及在类中说明的其它 static 函数。但它们有个限制：即不能操作非 static 数据或调用非 static 函数。其原因是一个 static 成员函数没有 this 指针。这就是说没有办法知道要访问哪个对象的非 static 数据。例如，若有一个类的两个对象含有一个静态函数 f()，并且若 f() 试图访问一个在类中定义的称作 var 的非 static 变量，那么调用跟踪哪一个 var 的拷贝呢？编译程序没有办法知道。这就是为什么 static 函数仅能够访问其它 static 函数或数据的原因。

我们来看一个 static 函数的例子。下面是一个给出其使用方法的一个短程序。一个对象要求访问某些紧缺资源是很普遍的。如要求访问一个网络中的共享文件，作为程序说明，使用 static 数据和函数使得一个对象可以检查资源状态以及在其可用时访问：

```
#include <iostream.h>

enum access_t {shared, in_use, locked, unlocked};

// a scarce resource control class
class access {
    static enum access_t acs;
    // ...
public:
    static void set_access(enum access_t a) {acs = a;}
    static enum access_t get_access()
    {
        return acs;
    }
}
```

```

    }
    // ...
};

main(void)
{
    access obj1, obj2;

    obj1.set_access(locked);

    // ... intervening code

    // see if obj2 can access resource

    if(obj2.get_access()==unlocked) {
        obj2.set_access(in_use);
        cout << "access resource\n";
    }
    else cout << "locked out\n";

    // ...
}

```

若编译该程序，将看到有“locked out”被显示。你可以不断分析该程序，直到认为已了解了 static 在数据和函数上的作用为止。

如前所述，static 函数只可以访问相同类中的其它 static 函数或 static 数据。要证明这一点，编译下列程序：

```

#include <iostream.h>

enum access_t {shared, in_use, locked, unlocked};

// a scarce resource control class
class access {
    static enum access_t acs;
    int i; // non-static
    // ...
public:
    static void set_access(enum access_t a) {acs = a;}
    static enum access_t get_access()
    {
        i = 100; // this will not compile
        return acs;
    }
    // ...
};

main(void)
{
    access obj1, obj2;

    obj1.set_access(locked);

    // ... intervening code

    // see if obj2 can access resource

    if(obj2.get_access()==unlocked) {
        obj2.set_access(in_use);
        cout << "access resource\n";
    }
}

```

```

    }
    else cout << "locked out\n";

    // ...
}

```

Turbo C++将给出一个错误信息，并不编译此程序，因为 `getaccess()` 试图要访问一个非 `static` 变量。

尽管开始时没有用 `static` 成员的必要，但当继续书写 C++ 程序时，将会发现在有些时候它们是很有用的，因为它们能够避免使用全局变量。

### 21.3 C 与 C++ 的区别

从很大程度上讲，C++ 是 ANSI 标准 C 的超集。实际上所有 C 程序也是 C++ 程序。但它们也确实存在不同之处，其最重要的在这里将作讨论。

C 与 C++ 之间最重要的区别是在 C 中，函数说明为：

```
int f();
```

没有对该函数的参数作说明，即当在函数名之后的括号中没有指定什么时，在 C 中意味着没有对函数中的参数作任何说明。它可能有参数，也可能没有参数，但在 C++ 中。像这样的函数说明意味着函数没有参数，即在 C++ 中，下列两种说明是一样的：

```
int f();
```

```
int f(void);
```

在 C++ 中，`void` 是可选的。许多 C++ 程序员用 `void` 以明确说明一个函数没有参数，但这是不必要的。

在 C++ 中，所有函数必须被原型化。这在 C 中是可选的，尽管程序设计经验告诉我们在 C 程序中最好使用原型。

C 与 C++ 还有一个较小的潜在的区别是在 C 中一个字符常量能被自动地转换成一个整型量。在 C++ 中则不能。

在 C 中，多次说明一个全局变量不是一个错误，尽管这种程序设计方法很糟。在 C++ 中，这却是错误的。

在 C 中，标识符可达到 31 个字符长。在 C++ 中，没有这样的限制。然而，从实用角度看，过长的标识符是不需要的也很少使用。

### 21.4 Turbo C++ 的复数及 BCD 类

除 `IOSTREAM.H` 定义的类及重载操作符之外，Turbo C++ 还包括另外两类库，其执行复数及 BCD 算术。让我们现在先来对它们做一些考察。

一个复数有两个部分：实部和虚部。实部为一个一般的数，而虚部是一个与 -1 的平方根之积。要使用复数，必须把 `COMPLEX.H` 包含在程序中。

要建立一个复数，使用 `complex` 构造函数。其原型如下：

```
complex(double real_part, double imaginary_part);
```

<<和>>是相对于复数重载的操作符。例如，下列程序构造一个虚数并把它显示在屏幕上。

```
#include <iostream.h>
#include <complex.h>

main(void)
{
    complex num(10, 1);

    cout << num;

    return 0;
}
```

程序输出如下:

(10, 1)

该输出还显示了复数输出的一般格式。

可以把复数与任何其它类型的数混起来, 包括整型数, floats 和 doubles。算术操作符像关系操作符==和!=那样相关于复数重载。下列程序说明如何在一个表达式中同时使用复数和一般数。

```
#include <iostream.h>
#include <complex.h>

main(void)
{
    complex num(10, 1);

    num = 123.23 + num / 3;

    cout << num;

    return 0;
}
```

Turbo C++重载了许多数学函数, 如对应复数的 sin() (它返回其参数的正弦值)。它还定义了专门应用于复数的几个函数, 复数函数列在表 21-1 中。

表 21-1 复数函数

函数	返回
complex abs(complex n)	n 的绝对值
double acos(complex u)	u 的反余弦值
double arg(complex u)	在复数坐标系中 u 的角度
complex asin(complex n)	n 的反正弦值
complex atan(complex n)	u 的反正切值
complex atan2(complex u)	n 的反正切值
double conj(complex n)	n 的共轭数
complex cos(complex n)	n 的余弦
complex cosh(complex n)	n 的双曲余弦
complex exp(complex n)	$e^n$

<code>double imag(complex n)</code>	n的虚部
<code>complex log(complex n)</code>	n的自然对数
<code>complex log10(complex n)</code>	以10为底的n的对数
<code>double norm(complex n)</code>	n的平方
<code>complex polar(double magnitude, double angle)</code>	给出复数的极坐标
<code>complex pow(complex x, complex y)</code>	
<code>complex pow(complex x, double y)</code>	
<code>complex pow(double x, complex y)</code>	$x^y$
<code>double real(complex n)</code>	n的实部
<code>complex sin(complex n)</code>	n的正弦值
<code>complex sinh(complex n)</code>	n的双曲正弦值
<code>complex sqrt(complex n)</code>	n的平方根
<code>complex tan (complex n)</code>	n的正切
<code>complex tanh(complex n)</code>	n的双曲正切值

Turbo C++还定义了BCD类。一个实数能在计算机中用许多种方法表示,最通常的是使用二进制浮点数,但第一种表示实数的方法为二——十进制码,即BCD码。在BCD中,以10而不是以2为基表示一个数。BCD表示的主要优点是不会有舍入误差。例如,数100.23在二进制中不能被精确地表示,将被近似的表示为100.230003。但用BCD则不会有这种舍入误差。由于这个原因,BCD数常用在记帐之类的程序中。BCD数的缺点是它的计算慢于二进制浮点计算。要用BCD数,必须把BCD.H包含在程序中:

bcd类有如下构造函数:

```
bcd(int n);
```

```
bcd(double n);
```

```
bcd(double n, int digits);
```

前两个是自说明的。最后一个是建立一个在小数点后使用digits(数字个数)的BCD数。在Turbo C++中精度为17个数字的BCD数范围在 $10^{-125}$ 到 $10^{125}$ 之间。

要把一个数从BCD格式转换成一般二进制浮点格式,用real)。其原型如下:

```
long double real(bcd n);
```

bcd类重载了算术和关系操作符以及在表21-2中所列的函数。

表 21-2 BCD 函数

函数	返回
<code>bcd abs(bcd n)</code>	n的绝对值
<code>bcd acos(bcd n)</code>	n的反余弦值
<code>bcd asin(bcd n)</code>	n的反正弦值
<code>bcd atan(bcd n)</code>	n的反正切值

---

<code>bcd cos(bcd n)</code>	n的余弦值
<code>bcd cosh(bcd n)</code>	n的双曲余弦
<code>bcd exp(bcd n)</code>	$e^n$
<code>bcd log(bcd n)</code>	n的自然对数
<code>bcd log10(bcd n)</code>	以10为底n的对数
<code>bcd pow(bcd x, bcd y)</code>	
<code>bcd sin(bcd n)</code>	u的正弦
<code>bcd sinh(bcd n)</code>	n的双曲正弦值
<code>bcd sqrt(bcd n)</code>	u的平方根
<code>bcd tan(bcd n)</code>	n的正切
<code>bcd tanh(bcd n)</code>	u的双曲正切

---

下面是一个样本程序，它说明当避免舍入误差是很重要的时候 BCD 数显示的优点：

```
#include <iostream.h>
#include <bcd.h>

main(void)
{
    float f = 100.23, f1 = 101.337;
    bcd b(100.23), b1(101.337);

    cout << f+f1 << " " << b+b1;

    return 0;
}
```

该程序在屏幕上显示：

201.567001 201.567

## 21.5 基于消息的哲学

在结束本书之前，我们再来看看面向对象程序设计的哲学。程序设计的哲学是基于消息的概念。在基于消息的方法中，多数数据被专门保存在一个类中。要提取或修改一个数据项，向该对象发送一条消息。在类之外的代码从不直接操作保存在类中的任何数据。能够修改数据的唯一的東西是含有数据对象的成员(或友元)函数。该方法可能产生偶然的副作用，它还使你能够准确控制一个对象的私有数据可以拥有什么值，因为能访问数据的成员能滤掉错误值。

在 C++ 中，向一个对象发送一条消息意味着调用一成员(或友元)函数。为更好地理解该概念。考虑管理访问一个数据库的类。在一般的 C 代码中，修改数据库中的一项，只需写出下列代码行：

```
database[record].balance = 100.75;
```

(假设 database 是一个有某种类 C 结构的类型的数组)。

但使用一个基于消息的方法和 C++，你可以用记录号和新值作为参数调用一个成员函数。例如，可使用下列语句：

```
database.newbalance(record, 100.75);
```

(这里, `database` 是一个对象)。在这种情形下, 没有其它代码实际能够“碰”保护在对象中的数据。

要看基于消息的方法实际是如何工作的, 下面是一个模拟秒表的类, 并写了一个短的 `main` 程序以显示其应用:

```
#include <iostream.h>
#include <time.h>
#include <conio.h>

class stopwatch {
    clock_t time1, time2;
public:
    stopwatch() {time1 = time2 = 0;}
    void reset() {time1 = time2 = 0;}
    void start() {time1 = clock()/CLK_TCK;}
    void stop() {time2 = clock()/CLK_TCK;}
    clock_t elapsed() {return time2 - time1;}
};

main(void)
{
    stopwatch timer;

    cout << "wait a while, then press a key\n";
    timer.start();

    while(!kbhit()) ; // wait for keypress

    timer.stop();
    cout << (long) timer.elapsed();

    cout << " seconds have elapsed\n";

    return 0;
}
```

本程序显示了从起跑开始与按键时之间的秒数。其使用了 Turbo C++ 的 `clock()` 函数。该函数返回一个值, 其为程序开始运行以来的秒数。类型 `clock_t` 被定义在 `clock()` 函数所需的 `TIME.H` 头文件中。该类型实质上是一个长整型数。宏 `CLK_TCK` 也定义在该文件中。

`stopwatch` 类把 `time1` 和 `time2` 说明成私有成员。只有通过成员函数发送消息, 才能访问它们。例如, 要启动秒表, 调用 `start()` 函数发送消息 “start the clock”。该函数然后设置 `time1` 的值。要停止秒表, 调用 `stop()` 把发送消息 “stop the clock”。要获得所用去的时间, 调用 `elapsed()` 函数。这里的关键是在任何时候都没有程序的任何部分直接访问 `time1` 或 `time2`。

尽管象这样实现 `stopwatch` 是正确的, 但它并没有向 `time1` 和 `time2` 提供应有的保护。例如, 没有措施防止在调用 `stop()` 之前 `start()` 函数被第二次调用。并且也没有措施防止在 `start()` 被调用之前 `stop()` 被调用。但使用基于消息的结构, 可以正常访问私有数据并能避免其被误用。例如, 下面是一个能避免秒表偶然误用的程序的改进:

```
#include <iostream.h>
#include <time.h>
#include <conio.h>
```



```

class stopwatch {
    clock_t time1, time2;
    int ready;
public:
    stopwatch() {time1 = time2 = 0; ready = 1;}
    void reset() {time1 = time2 = 0; ready = 1;}
    void start();
    void stop();
    clock_t elapsed();
} ;

void stopwatch::start()
{
    if(!ready)
        cout << "timer has not been reset\n";
    else {
        time1 = clock()/CLK_TCK;
        ready = 0;
    }
}

void stopwatch::stop()
{
    if(ready)
        cout << "timer has not been started\n";
    else {
        time2 = clock()/CLK_TCK;
        ready = 1;
    }
}

clock_t stopwatch::elapsed()
{
    if(!ready) {
        cout << "timer has not been stopped\n";
        return -1;
    }
    else
        return time2 - time1;
}

main(void)
{
    stopwatch timer;

    cout << "wait a while, then press a key\n";
    timer.start();

    while(!kbhit()) ; // wait for keypress
    getch(); // read and dispose of keystroke

    timer.stop();
    cout << (long) timer.elapsed();
    cout << " seconds have elapsed\n";

    timer.stop(); // this will cause error message because
                  // the timer is not currently running
}

```

```

timer.start();
cout << "now running, wait a while, then press a key\n";

while(!kbhit()) ; // wait for keypress

timer.stop(); // now, this will work
cout << (long) timer.elapsed();
cout << " seconds have elapsed\n";

return 0;
}

```

在上述程序中，不可能发生秒表的误用，因为只有当秒表当前不运行时 ready 标志才设置。

尽管不一定要使用基于消息的方法进行 C++ 程序设计，但不这样可能会丢掉许多 C++ 的功能。若掌握了这种方法，书写程序时会更加灵活自如。

## 21.6 最后的话

若你是一个面向对象程序设计的新手，但又希望精通它，最好的办法是书写大量的面向对象程序。学习程序设计最好的办法是亲自写，并看看其他人写的 C++ 程序。若可能的话，研究几个不同程序员写的 C++ 代码，注意程序是如何设计和实现的，寻找其中的优缺点。这将扩展程序设计的思路。最后要注意多实践，这样你就会很快发现自己成了一个 C++ 程序设计专家。

## 附录 A 常用的一些 C 库函数

正如你所知道的, C++ 是建立在 C 之上的。C 的许多功能都是由其库函数提供的。C++ 同样含有这些函数, 它们同样对 C++ 程序是很重要的。由于这个原因, 大量常用的库函数在这里讨论。若你曾看过《库函数参考》手册, 无疑会了解大量的库函数。讨论每个库函数超过了本书的范围, 然而, 在这里所讨论的函数以及在本书正文中讨论的类库都是你在大多数程序设计中要用到的。

大多数在本书中未曾讨论过的库函数可分为下列几类:

- 串和字符函数
- 数学函数
- 操作系统相关函数
- 其它函数

本章将依次讨论每一类。在这里讨论的一些函数已在前面章节介绍过。为完整起见, 也把它们归到这里。

每个函数说明都以后面跟着其原形的函数所要求的头文件开头。记住, 原形为你了解函数的参数类型、数量及其返回值类型提供了方便。在每一类中, 所有函数都按字母顺序排列。

### A.1 串和字符函数

Turbo C++ 标准库拥有丰富的串和字符操作函数。在 C 中, 一个串是一个以空字符结尾的字符数组。串函数的说明在头文件 `string.h` 中找到。字符函数用 `ctype.h` 作为它们的头文件。

由于 C 中没有数组操作的边界检查, 因而程序员要负责避免数组的溢出。

在 Turbo C++ 中, 一个可打印字符即是可在终端上显示的字符。控制字符的值在 0 和 0x1F 之间, 以及 DEL(0x7F)。

字符函数被说明有一个整型参数。在这种情形下, 只有低位字节被函数使用。通常, 可以自由地使用一个字符参数, 因为在调用时其被自动地提高为 `int` 类型。

```
#include <ctype.h>
```

```
int isalnum(int ch)
```

说明 `isalnum()` 函数在其参数为一个字母或一个数字时返回非零。若字符不为数字字符型, 则返回 0。

示例 该程序检查每一个从键盘读入的字符并报告所有数字字符型字符:

```
#include <ctype.h>
#include <stdio.h>
#include <conio.h>
```

```
main(void)
```

```
{
    char ch;
```

```

for(;;) {
    ch = getche();
    if(ch==' ') break;
    if(isalnum(ch)) printf("%c is alphanumeric\n", ch);
}
return 0;
}

```

**#include <ctype.h>**

**int isalpha(int ch)**

**说明** 若 ch 是一个字母, isalpha() 函数返回非零。否则, 返回 0。

**示例** 该程序检查每一个从键盘读入的字符并报告所有字母型字符:

```

#include <ctype.h>
#include <stdio.h>
#include <conio.h>

main(void)
{
    char ch;
    for(;;) {
        ch = getche();
        if(ch==' ') break;
        if(isalpha(ch)) printf("%c is a letter\n", ch);
    }
    return 0;
}

```

**#include <ctype.h>**

**int iscntrl(int ch)**

**说明** 若 ch 在 0 和 0x1F 之间或等于 0x7F(DEL)。iscntrl() 函数返回非零。

**示例** 该程序检查每一个从键盘读入的字符并报告所有控制型字符:

```

#include <ctype.h>
#include <stdio.h>
#include <conio.h>

main(void)
{
    char ch;
    for(;;) {
        ch = getche();
        if(ch==' ') break;
        if(iscntrl(ch)) printf("%c is a control character\n", ch);
    }
    return 0;
}

```

**#include <ctype.h>**

**int isdigit(int ch)**

**说明** 若 ch 是一个数字, 即 0 到 9, isdigit() 函数返回非零。

**示例** 该程序检查每一个从键盘读入的字符并报告所有数字型字符:

```

#include <ctype.h>
#include <stdio.h>
#include <conio.h>

main(void)
{
    char ch;

    for(;;) {
        ch = getche();
        if(ch==' ') break;
        if(isdigit(ch)) printf("%c is a digit\n", ch);
    }
    return 0;
}

```

**#include <ctype.h>**

**int isgraph(int ch)**

**说明** 若 **ch** 是任何可打印字符并且不是一个空格, **isgraph()** 函数返回非零。否则, 返回 0。可打印字符在 0x21 和 0x7E 之间。

**示例** 该程序检查每一个从键盘读入的字符并报告所有可打印字符:

```

#include <ctype.h>
#include <stdio.h>

main(void)
{
    char ch;
    for(;;) {
        ch = getche();
        if(ch==' ') break;
        if(isgraph(ch)) printf("%c is a printing character\n", ch);
    }
    return 0;
}

```

**#include <ctype.h>**

**int islower(int ch)**

**说明** 若 **ch** 是一个小写字母(a 到 z), **islower()** 函数返回非零。否则, 返回零。

**示例** 该程序检查每一个从键盘读入的字符并报告所有小写字母:

```

#include <ctype.h>
#include <stdio.h>
#include <conio.h>

main(void)
{
    char ch;

    for(;;) {
        ch = getche();
        if(ch==' ') break;
        if(islower(ch)) printf("%c is lowercase\n", ch);
    }
    return 0;
}

```

```
#include <ctype.h>
```

```
int isprint(int ch)
```

说明 若 `ch` 是一个可打印字符，包括空格，`isprint()` 函数返回非零。否则，返回 0，可打印字符通常在 0x20 和 0x7E 之间。

示例 该程序检查每一个从键盘读入的字符并报告所有可打印字符：

```
#include <ctype.h>
#include <stdio.h>
#include <conio.h>

main(void)
{
    char ch;

    for(;;) {
        ch = getche();
        if(ch=='\n') break;
        if(isprint(ch)) printf("%c is printable\n", ch);
    }
    return 0;
}
```

```
#include <ctype.h>
```

```
int ispunct(int ch)
```

说明 若 `ch` 是一个标点字符，空格除外，`ispunct()` 函数返回非零。否则，返回 0。本函数定义的标点字符包括所有可打印字符。它们既不是数字字母型，也非空格。

示例 该程序检查每一个从键盘读入的字符并报告所有标点符号。

```
#include <ctype.h>
#include <stdio.h>
#include <conio.h>

main(void)
{
    char ch;

    for(;;) {
        ch = getche();
        if(ch=='\n') break;
        if(ispunct(ch)) printf("%c is punctuation\n", ch);
    }
    return 0;
}
```

```
#include <ctype.h>
```

```
int isspace(int ch)
```

说明 若 `ch` 是下列之一：空格、制表符、竖杠、换行、回车或换行字符，则 `isspace()` 函数返回非零。否则，返回 0。

示例 该程序检查每一个从键盘读入的字符并报告所有空白。

```
#include <ctype.h>
#include <stdio.h>
#include <conio.h>
```

```

main(void)
{
    char ch;

    for(;;) {
        ch = getche();

        if(ch==' ') break;
        if(isspace(ch)) printf("%c is white-space\n", ch);
    }
    return 0;
}

```

**#include <ctype.h>**

**int isupper(int ch)**

**说明** 若 **ch** 是一个大写字母(A到Z), **isupper** 函数返回非零。否则, 返回 0。

**示例** 该程序检查每一个从键盘读入的字符并报告所有大写字母:

```

#include <ctype.h>
#include <stdio.h>
#include <conio.h>

main(void)
{
    char ch;

    for(;;) {
        ch = getche();
        if(ch==' ') break;
        if(isupper(ch)) printf("%c is uppercase\n", ch);
    }
    return 0;
}

```

**#include <ctype.h>**

**int isxdigit(int ch)**

**说明** 若 **ch** 是一个十六进制数字, **isxdigit()** 函数返回非零; 否则, 返回 0。一个十六进制数字是下列区间内的一个数字: A 到 F, a 到 f 或 0 到 9。

**示例** 该程序检查每一个从键盘读入的字符并报告所有十六进制数字:

```

#include <ctype.h>
#include <stdio.h>
#include <conio.h>

main(void)
{
    char ch;

    for(;;) {
        ch = getche();
        if(ch==' ') break;
        if(isxdigit(ch)) printf("%c is hexadecimal\n", ch);
    }
    return 0;
}

```

```
#include <string.h>
```

```
char *strcat(char *str1, const char *str2)
```

**说明** `strcat()`函数把 `str2` 的一个备份添在 `str1` 之后并使 `str1` 以空字符结束。原来作为 `str1` 的空字符被 `str2` 的第一个字符覆盖。该操作不影响 `str2`, `strcat()`函数返回串 `str1`。记住, 没有进行边界检查, 因而程序员应负责保证 `str1` 足够的大小能容纳其原有内容及 `str2` 的内容。

**示例** 该程序把从键盘读入的第一个串添加到第二个串中, 例如: 假设用户输入 `hello` 和 `there`, 则程序将打印 `"there hello"`。

```
#include <string.h>
#include <stdio.h>

main(void)
{
    char s1[80], s2[80];

    printf("enter two strings: ");
    gets(s1);
    gets(s2);

    strcat(s2, s1);
    printf(s2);
    return 0;
}
```

```
#include <string.h>
```

```
char *strchr(const char *str, int ch)
```

**说明** `strchr()`函数返回一个指针, 它指向 `str` 指向的串中的 `ch` 的低位字节的第一次出现之处。

**示例** 该程序打印串 `"is a test"`:

```
#include <string.h>
#include <stdio.h>

main(void)
{
    char *p;

    p = strchr("this is a test", (int) ' ');
    printf(p);
    return 0;
}
```

```
#include <string.h>
```

```
int strcmp(const char *str1, const char *str2)
```

**说明** `strcmp()`函数比较两个以空字符结尾的串, 并根据结果返回一个整数。如:

值	含意
小于0	<code>str1</code> 小于 <code>str2</code>
0	<code>str1</code> 等于 <code>str2</code>
大于0	<code>str1</code> 大于 <code>str2</code>



**示例** 下列函数将用作一口令检查子程序。失败时返回 0，成功时返回 1。

```
#include <string.h>

password()
{
    char s[80];

    printf("enter password: ");
    gets(s);

    if(strcmp(s,"pass")) {
        printf("invalid password\n");
        return 0;
    }
    return 1;
}
```

**#include <string.h>**

**char \*strcpy (char \*str1,const char \*str2)**

**说明** str2 必须为一指向一个以空字符结束的串的指针。strcpy() 函数返回一个指向 str1 的指针。

若 str1 和 str2 重叠。strcpy() 的行为则是不确定的。

**示例** 下列代码段将把“hello”拷贝到串 str 中:

```
char str[80];
strcpy(str, "hello");
```

**#include <string.h>**

**unsigned int strlen(const char \*str)**

**说明** strlen() 函数返回 str 指向的以空字符结尾的串的长度，结尾空字符不被计入。

**示例** 下列代码段将在屏幕上显示数字 5。

```
strcpy(s, "hello");
printf("%d", strlen(s));
```

**#include <stdio.h>**

**char \*strstr(const char \*str1, const char \*str2)**

**说明** strstr() 函数返回一个指针，它指向 str1 指向的串中 str2 指向的串的第一次出现之处。若没有找到匹配，返回一个空指针。

**示例** 该程序显示信息“is a test”：

```
#include <string.h>
#include <stdio.h>

main(void)
{
    char *p;

    p = strstr("this is a test","is");
    printf(p);
    return 0;
}
```

```
#include <string.h>
```

```
char *strtok(char *str1, const char *str2)
```

**说明** `strtok()` 函数返回一个指向 `str1` 指向的串中的下一个符号的指针。组成 `str2` 指向的串的字符是确定该符号的定界符。若没有符号可返回，则返回一个空指针。

第一次调用 `strtok()` 时，实际用的是 `str1`，以后的调用使用第一个参数为空指针。这样整个串可变为多个符号。

注意 `strtok()` 函数修改了 `str1` 指向的串。每当找到一个符号，就把一个空格符放在界符所在之处，这样，`strtok()` 可以继续往前处理该串。

每次调用 `strtok()` 时，可以使用不同的界符集。

**示例** 该程序用空格和逗号作为定界符把串 “the summer soldier, the sunshine patriot” 符号化，其输出将为：

The|summer|soldier|the|sunshine|patriot

```
#include <string.h>
#include <stdio.h>

main(void)
{
    char *p;

    p = strtok("The summer soldier, the sunshine patriot", " ,");
    printf(p);
    do {
        p = strtok("", " ,");
        if (p) printf(" %s", p);
    } while (p);
    return 0;
}
```

```
#include <ctype.h>
```

```
int tolower(int ch)
```

**说明** 若 `ch` 是一个字母。`tolower()` 函数返回 `ch` 的小写字母的等价值。否则，照原样返回 `ch`。

**示例** 该代码段显示 “q”。

```
putchar(tolower('Q'));
```

```
#include <ctype.h>
```

```
int toupper(int ch)
```

**说明** 若 `ch` 是一个字母，`toupper()` 函数返回 `ch` 的大写字母的等价值。否则照原样返回 `ch`。

**示例** 该代码段显示 “A”。

```
putchar(toupper('a'));
```

## A.2 数学函数

Turbo C++有几个数学函数, 它们有 `double` 型参数并返回 `double` 型值。这些函数可分为下列几类:

- 三角函数
- 双曲函数
- 指数和对数函数
- 其它函数

所有数学函数都要求任何使用它们的程序中包含头文件 `math.h`。除了说明对应数学函数之外, 头文件还定义叫作 `EDOM`、`ERANGE` 和 `HUGE_VAL` 的宏。若一个数学函数的参数不在其定义域中, 则返回 0 并且全局 `errno` 被置为 `EDOM`。若一个子程序产生一个过大而不能用一个 `double` 表示的结果, 则出现上溢, 这便导致子程序返回 `HUGE_VAL`, 并且 `errno` 被置为 `ERANGE`, 指示一个范围错误。若出现下溢, 子程序返回 0, 并且置 `errno` 为 `ERANGE`。

许多数学函数因操作复数而重载。这些函数的复数版本在第二十一章中已讨论。这里, 仅讨论类似的版本。

```
#include <math.h>
```

```
double acos(double arg)
```

说明 `acos()` 函数返回 `arg` 的反余弦值。传给 `acos()` 的参数必须在 -1 到 1 之间。否则出现域错误。

示例 本程序打印反余弦值, 在 -1 到 1 之间, 取 1/10 为增量。

```
#include <math.h>
#include <stdio.h>

main(void)
{
    double val = -1.0;

    do {
        printf("arc cosine of %f is %f\n", val, acos(val));
        val += 0.1;
    } while(val <= 1.0);
    return 0;
}
```

```
#include <math.h>
```

```
double asin(double arg)
```

说明 `asin()` 函数返回 `arg` 的反正弦值, 传给 `asin()` 的参数必须在 -1 到 1 之间。否则出现域错误。

示例 本程序打印反正弦值, 在 -1 到 1 之间, 取 1/10 为增量。

```
#include <math.h>
#include <stdio.h>

main(void)
{
    double val = -1.0;
```

```

do {
    printf("arc sine of %f is %f\n", val, asin(val));
    val += 0.1;
} while(val<=1.0);
return 0;
}

```

**#include <math.h>**

**double atan(double arg)**

**说明** atan()函数返回 arg 的反正切值。

**示例** 本程序打印反正切值，在-1 到 1 之间，取 1/10 为增量。

```

#include <math.h>
#include <stdio.h>

main(void)
{
    double val=-1.0;

    do {
        printf("arc tangent of %f is %f\n", val, atan(val));
        val += 0.1;
    } while(val<=1.0);
    return 0;
}

```

**#include <math.h>**

**double atan2(double y,double x)**

**说明** atan()函数返回 y/x 的反正切值。它利用其参数的符号来计算返回值的象限。

**示例** 本程序打印反正切值，y 在-1 到 1 之间。取 1/10 为增量。

```

#include <math.h>
#include <stdio.h>

main(void)
{
    double y=-1.0;

    do {
        printf("atan2 of %f is %f\n", y, atan2(y, 1.0));
        y += 0.1;
    } while(y<=1.0);
    return 0;
}

```

**#include <math.h>**

**double ceil(double num)**

**说明** ceil()函数返回不小于 num 的一个 double 表示的最小整数。例如，给出 1.02，ceil() 将返回 2.0。又如，给出-1.02，ceil()将返回-1。

**示例** 本代码段在屏幕上显示 "10"

```
printf("%f",cell(9.9));
```

```
#include <math.h>
```

```
double cos(double arg)
```

说明 cos()函数返回 arg 的余弦值, arg 的值必须以弧度给出。

示例 本程序打印余弦值, 在-1 到 1 之间, 取 1/10 为增量:

```
#include <math.h>
#include <stdio.h>

main(void)
{
    double val=-1.0;

    do {
        printf("cosine of %f is %f\n", val, cos(val));
        val += 0.1;
    } while(val<=1.0);
    return 0;
}
```

```
#include <math.h>
```

```
double cosh(double arg)
```

说明 cosh()函数返回 arg 的双曲余弦值。arg 的值必须以弧度给出。

示例 本程序打印双曲余弦值, 在-1 到 1 之间, 取 1/10 为增量:

```
#include <math.h>
#include <stdio.h>

main(void)
{
    double val=-1.0;

    do {
        printf("hyperbolic cosine of %f is %f\n", val, cosh(val));
        val += 0.1;
    } while(val<=1.0);
    return 0;
}
```

```
#include <math.h>
```

```
double exp(double arg)
```

说明 exp()函数返回自然对数 e 的 arg 次方。

示例 本代码段显示 e 的值(精确到 2.718282):

```
printf("value of e to the first: %f", exp(1.0));
```

```
#include <math.h>
```

```
double fabs(double num)
```

说明 fabs()函数返回 num 的绝对值。

示例 本程序在屏幕上显示 "1.0 1.0"

```

#include <math.h>
#include <stdio.h>

main(void)
{
    printf("Zl.1f Zl.1f", fabs(1.0), fabs(-1.0));
    return 0;
}

```

```
#include <math.h>
```

```
double floor(double num)
```

**说明** floor()函数返回不大于 num 的最大整数。(用 double 表示)。例如, 给出 1.02, floor()将返回 1.0; 又如给出 -1.02, floor()将返回 -2.0。

**示例** 本代码段在屏幕上显示 “10”。

```
printf("%f", floor(10.9));
```

```
#include <math.h>
```

```
double log(double num)
```

**说明** log()函数返回 num 的自然对数。若 num 为负数, 则出现域错。若参数为 0, 出现域错。

**示例** 本程序显示从 1 到 10 的自然对数:

```

#include <math.h>
#include <stdio.h>

main(void)
{
    double val=1.0;

    do {
        printf("Zf Zf\n", val, log(val));
        val++;
    } while (val<11.0);
    return 0;
}

```

```
#include <math.h>
```

```
double log10(double num)
```

**说明** log10 函数返回以 10 为底的 num 的对数。若 num 为负, 出现域错误。若参数为 0, 也出现域错。

**示例** 本程序打印以 10 为底从 1 到 10 的对数:

```

#include <math.h>
#include <stdio.h>

main(void)
{
    double val=1.0;

    do {
        printf("Zf Zf\n", val, log10(val));
    }
}

```

```

        val++;
    } while (val<11.0);
    return 0;
}

```

**#include <math.h>**

**double pow(double base ,double exp)**

**说明** pow()函数返回 base 的 exp 次方。若 base 为 0 并且 exp 小于或等于 0。出现域错。若 base 为负值并且 exp 不为整数，也会出现域错。溢出将导致域错。

**示例** 本程序显示 10 的 0 到 10 次方

```

#include <math.h>
#include <stdio.h>

main(void)
{
    double x=10.0, y=0.0;

    do {
        printf("zf",pow(x, y));
        y++;
    } while(y<11);
    return 0;
}

```

**#include <math.h>**

**double sin(double arg)**

**说明** sin()返回 arg 的正弦值。arg 的值必须用弧度给出。

**示例** 本程序打印正弦值，在 -1 到 1 之间，取 1/10 为增量。

```

#include <math.h>
#include <stdio.h>

main(void)
{
    double val=-1.0;

    do {
        printf("sine of zf is zf\n", val, sin(val));
        val += 0.1;
    } while(val<=1.0);
    return 0;
}

```

**#include <math.h>**

**double sinh(double arg)**

**说明** sinb()函数返回 arg 的双曲正弦值。arg 的值必须用弧度给出

**示例** 本程序显示双曲余弦，在 -1 到 1 之间，以 1/10 为增量。

```

#include <math.h>
#include <stdio.h>

```

```

main(void)
{
    double val=-1.0;

```

```

do {
    printf("hyperbolic sine of %f is %f\n", val, sinh(val));
    val += 0.1;
} while(val<=1.0);
return 0;
}

```

**#include <math.h>**

**double sqrt(double num)**

**说明** sqrt()函数返回 num 的平方根。若其参数为负值，则出现域错。

**示例** 本代码段在屏幕上显示“4”。

```
printf("%f",sqrt(16.0));
```

**#include <math.h>**

**double tan (double arg)**

**说明** tan()函数返回 arg 的正切。arg 的值必须以弧度给出。

**示例** 本程序打印正切值，在-1 到 1 之间，取 1/10 为增量。

```

#include <math.h>
#include <stdio.h>

main(void)
{
    double val=-1.0;

    do {
        printf("tangent of %f is %f\n", val, tan(val));
        val += 0.1;
    } while(val<=1.0);
    return 0;
}

```

**#include <math.h>**

**double tanh(double arg)**

**说明** tanh()函数返回 arg 的双曲正切值。arg 的值必须以弧度给出。

**示例** 本程序打印双曲正切值，在-1 到 1 之间，取 1/10 为增量。

```

#include <math.h>
#include <stdio.h>

main(void)
{
    double val=-1.0;

    do {
        printf("Hyperbolic tangent of %f is %f\n", val, tanh(val));
        val += 0.1;
    } while(val<=1.0);
    return 0;
}

```



### A.3 操作系统相关函数

本节讨论在某些方面比其它函数对于操作系统较为敏感的函数。在 Turbo C++ 库中, 这些函数包括时间和日期函数以及允许直接与操作系统交互的函数。

时间和日期函数要求其原型的头文件 `time.h`。头文件还定义两种类型: `time_t` 类型作为一个长整数能够表示系统时间和日期, 这称作日历时间; 结构类型 `tm` 容纳日期和时间, 其结构由下式定义:

```
struct tm {
    int tm_sec; /* seconds, 0-59 */
    int tm_min; /* minutes, 0-59 */
    int tm_hour; /* hours, 0-23 */
    int tm_mday; /* day of the month, 1-31 */
    int tm_mon; /* months since Jan, 0-11 */
    int tm_year; /* years from 1900 */
    int tm_wday; /* days since Sunday, 0-6 */
    int tm_yday; /* days since Jan 1, 0-365 */
    int tm_isdst /* Daylight Savings Time indicator */
};
```

若夏令时设置(Daylight Savings Time)有效, `tm_isdst` 的值为正值。若其为无效时, `tm_isdst` 值为 0; 若没有有用的信息, 其值则为负值。用这种方法表示的时间称作分散式时间(brokendown time)。

Turbo C++ 定义的 DOS 交互函数要求头文件 `dos.h`。`dos.h` 文件定义了一外联合, 其与 8088/86 CPU 寄存器对应并被用作一些系统交互函数。它被定义两个结构的联合以便允许每个寄存器可以用字或字节进行访问。

```
/*      dos.h

Defines structs, unions, macros, and functions
for dealing with MSDOS and the Intel iAPX86
microprocessor family.

Copyright (c) Borland International Inc. 1987, 1988,
1990 All Rights Reserved.
*/

struct WORDREGS
{
    unsigned int    ax, bx, cx, dx, si, di, cflag, flags;
};

struct BYTEREGS
{
    unsigned char   al, ah, bl, bh, cl, ch, dl, dh;
};

union
{
    REGS            {
        struct WORDREGS x;
        struct BYTEREGS h;
    };
};
```

```
#include <time.h>
```

```
char *asctime (const struct tm *ptr)
```

说明 `asctime()` 函数返回一个指针, 它指向一个把 `ptr` 指向的结构中所存放的信息转换成如下形式的字符串:

day month date hours: minutes: seconds year\n\0

例如:

wed Jun 19 12:05:34 1999

传给 `asctime()` 的结构指针通常从 `localtime()` 或 `gmtime()` 中获得。

被 `asctime()` 用来容纳格式输出串的缓冲区是一个静态分配字符数组并在每次调用函数时被覆盖。若希望保存串的内容，则必须把它拷贝到别的地方。

示例 本程序显示系统定义的当地时间：

```
#include <time.h>
#include <stdio.h>

main(void)
{
    struct tm *ptr;
    time_t lt;
    lt = time(NULL);
    ptr = localtime(&lt);
    printf(asctime(ptr));
    return 0;
}
```

`#include <dos.h>`

`int bdos(int fnum, unsigned dx, unsigned al)`

说明 本函数不是 ANSI C 标准的一个部分。

`bdos()` 函数用来访问 `fnum` 指定的 DOS 系统调用。其首先把 `dx` 值放入 DX 寄存器并把 `al` 值放入 AL 寄存器。然后，其执行 INT 21H 指令。

`bdos()` 函数返回 AX 寄存器值，其被 DOS 用来返回信息。

`bdos()` 函数仅可用来访问那些没有参数或要求仅有 DX 或 AL 作为其参数的系统调用。

示例 本程序直接从键盘上读入字符，绕过所有 C 的 I/O 函数直到有 q 输入：

```
/* do raw keyboard reads */
#include <dos.h>

main(void)
{
    char ch;

    while((ch=bdos(1,0,0))!=\q) ;
    return 0;
}
```

`#include <time.h>`

`char *ctime(const time_t *time)`

说明 `ctime()` 函数返回一个指向有如下格式的串的指针：

day month date hours:minutes:seconds year\n\0

给出一个指向日历时间的指针。日历时间通常通过调用 `time()` 来获得。`ctime()` 函数与下式等价：

`asctime(localtime(time));`

被 `ctime()` 用来容纳有格式输出串的缓冲区是一个静态分配字符数组并在每次调用函数时被覆盖。若希望保存串的内容，则必须把它拷贝到别的地方。

示例 本程序显示系统定义的当地时间:

```
#include <time.h>
#include <stdio.h>
main(void)
{
    time_t lt;

    lt = time(NULL);
    printf("ctime(&lt);");
    return 0;
}
```

#include <time.h>

double difftime (time\_t time2, time\_t time1)

说明 difftime()函数返回按秒计的 time1 与 time2 之差, 即 time2 减 time1。

示例 本程序为 0 到 500000 次循环所花的秒数。

```
#include <time.h>
#include <stdio.h>

main(void)
{
    time_t start, end;
    long unsigned int t;

    start = time(NULL);
    for(t=0; t<500000L; t++) ;
    end = time(NULL);

    printf("loop required %f seconds\n", difftime(end, start));
    return 0;
}
```

#include <time.h>

struct tm \*gmtime(const time\_t \*time)

说明 gmtime()函数返回一个指针, 它指向以 tm 结构为格式的时间的分散格式。时间用格林尼治时间表示。time 的值通常通过调用 time()获得。

被 gmtime()用来分散时间的结构是静态分配的并在每次调用函数时被覆盖。若希望保存结构的内容, 则必须把它拷贝到别的地方。

示例 本程序打印系统当地时间和格林尼治时间:

```
#include <time.h>
#include <stdio.h>

/* print local and GM time */
main(void)
{
    struct tm *local, *gm;
    time_t t;

    t = time(NULL);
    local = localtime(&t);
    printf("Local time and date: %s", asctime(local));
    gm = gmtime(&t);
```

```

    printf("Greenwich mean time and date: %s", asctime(gm));
    return 0;
}

```

**#include <dos.h>**

**int int86(int int\_num, union REGS \*in\_regs, UNION REGS \*out\_regs)**

**说明** 本函数不是 ANSI C 标准的部分。

int86()函数用来执行一个由 int\_num 指定的软件中断, 首先把联合 in\_regs 中的内容拷贝到处理的寄存器中, 然后执行适当的中断。

在返回时, 联合 out\_regs 中将含有 CPU 从中断中返回的寄存器值。

联合 REGS 在头文件 dos.h 中定义。

**示例** 在 IBM PC 中, int86()函数通常用来调用 ROM 子程序。例如, 以下函数执行一个 INT 10H 函数代码 0, 它把视频方式赋给参数 mode 指示的对象:

```

#include <dos.h>

set_mode(char mode)
{
    union REGS in, out;

    in.h.al = mode;
    in.h.ah = 0; /* set mode function number */

    int86(0x10, &in, &out);
}

```

**#include <dos.h>**

**int intdos(union REGS \*in\_regs, union REGS \*out\_regs)**

**说明** 本函数不是 ANSI C 标准的一部分。

intdos()函数用来访问 in\_regs 指向的联合的内容所指定的 DOS 系统调用。它执行一个 INT 21H 指令, 并且该操作的结果被放在 out\_regs 指向的联合中。intdos()函数返回 AX 寄存器的值, 其被 DOS 用作返回信息。

intdos()函数主要用于以下两种情况:

1. 需要其它寄存器参数而不仅仅是 DX 或 AL 的系统调用;
2. 返回信息保存在其它寄存器而不是 AX 中的系统调用。

联合 REGS 定义 8088/86 族处理器的寄存器并在 dos.h 头文件中被说明。

**示例** 本程序直接从系统时钟中读入时间, 绕过所有 C 的时间函数:

```

#include <dos.h>
#include <stdio.h>

main(void)
{
    union REGS in, out;

    in.h.ah = 0x2c; /* get time function number */
    intdos(&in, &out);
    printf("time is %2.2d:%2.2d:%2.2d", out.h.ch, out.h.cl, out.h.dh);
    return 0;
}

```

```
#include <time.h>
```

```
struct tm *localtime(const time_t *time)
```

说明 `localtime()`函数返回一个指针，它指向以 `tm` 结构为格式的 `time` 的分散格式。时间用当地时间表示。`time` 的值通常通过调用 `time()`获得。

`localtime()`用来容纳分散时间的结构是静态分配的并在每次调用函数时被覆盖。若希望保存结构的内容，则必须把它拷贝到别的地方。

示例 本程序打印系统的当地时间和格林尼治时间。

```
#include <time.h>
#include <stdio.h>

/* print local and Greenwich mean time */
main(void)
{
    struct tm *local;
    time_t t;

    t = time(NULL);
    local = localtime(&t);
    printf("Local time and date: %s", asctime(local));
    local = gmtime(&t);
    printf("Greenwich mean time and date: %s", asctime(local));
    return 0;
}
```

```
#include <time.h>
```

```
time_t time(time_t *time)
```

说明 `time()`函数返回系统的当前日历时间。若系统没有时间，则返回-1。

`time()`函数可以用一个空指针或用一个指向 `time_t` 类型变量的指针调用，若是后者，则参数被赋于日历时间。

示例 本程序显示系统定义的当地时间。

```
#include <time.h>
#include <stdio.h>

main(void)
{
    struct tm *ptr;
    time_t lt;

    lt = time(NULL);
    ptr = localtime(&lt);
    printf(asctime(ptr));
    return 0;
}
```

#### A.4 其它函数

在本节讨论的函数属标准函数，它们不能归于某一类中。

```
#include <stdlib.h>
```

```
void abort(void)
```

**说明** `abort()`函数引起程序的立即中止。没有文件被冲掉,并且值 3 被返回到调用过程(通常为操作系统)。

`abort()`的主要用途在于避免一个运行程序关闭活动文件。

**示例** 在本程序中,若用户输入一个 A,程序将终止:

```
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

main(void)
{
    for(;;)
        if(getche()=='A') abort();
    return 0;
}
```

`#include <stdlib.h>`

`int abs(int num)`

**说明** `abs()`函数返回整数 `num` 的绝对值。

**示例** 本函数把用户输入的数转换成其绝对值:

```
#include <stdlib.h>
#include <stdio.h>

get_abs(void)
{
    char num[80];

    gets(num)

    return abs(atoi(num));
}
```

`#include <stdlib.h>`

`double atof(const char *str)`

**说明** `atof()`函数把 `str` 指向的串转换成一个双精度值,串必须含一个有效浮点数。若不是这样,返回值为 0。

数字可用不能作为一个合法浮点数部分的任何字符结束。其中包括空格、标点(除了英文句号),以及字符(除了“E”或“e”)。这就是说,若 `atof()`以“100.00HELLO”调用,返回值将为 100.00。

**示例** 本程序将读两个浮点数并显示它们的和:

```
#include <stdlib.h>
#include <stdio.h>

main(void)
{
    char num1[80], num2[80];

    printf("enter first: ");
    gets(num1);
```

```

    printf("enter second: ");
    gets(num2);
    printf("the sum is: %f",stof(num1)+stof(num2));
    return 0;
}

```

#include <stdlib.h>

int atoi(const char \*str)

**说明** atoi()函数把 str 指向的串转换成一个 int 值。该串必须含一个合法的整型值。若不是这样，返回值为 0。

数字可用不能作为一个合法整数部分的任何字符结束。其中包括空格、标点和其它字符，这即是说若 atoi()以“123.23”调用，将返回整型 123，0.23 被舍去。

**示例** 本程序读两个整数并显示它们之和：

```

#include <stdlib.h>
#include <stdio.h>

main(void)
{
    char num1[80], num2[80];

    printf("enter first: ");
    gets(num1);
    printf("enter second: ");
    gets(num2);
    printf("the sum is: %d",atoi(num1)+atoi(num2));
    return 0;
}

```

#include <stdlib.h>

int atol(const char \*str)

**说明** atol()函数把 str 指向的串转换成一个 long int 值。串中必须含有一个合法的长整型值。若不是这样，返回值为 0。

数字可用不能作为一个合法整数部分的任何字符结束。其中包括空格、标点和其它字符。这就是说若 atol()以“123.23”调用，将返回整型值 123，0.23 被舍去。

**示例** 本程序读两个长整型数，并以下列方式显示它们之和：

```

#include <stdlib.h>
#include <stdio.h>

main(void)
{
    char num1[80], num2[80];

    printf("enter first: ");
    gets(num1);
    printf("enter second: ");
    gets(num2);
    printf("the sum is: %ld",atol(num1)+atol(num2));
    return 0;
}

```

```
#include <stdlib.h>
```

```
void *bsearch(const void *key, const void *base, unsigned num, unsigned size,  
              int(compare)(const void *, const void *))
```

说明 `bsearch` 函数对 `base` 指向的有序数组进行二分搜索，并返回一个指向与指针 `key` 所指示的关键字相匹配的第一个成员的指针。在数组中的元素号由 `num` 指定，每一个元素的大小(用字节)用 `size` 描述。

`compare` 所指的函数用来把数组中的一个元素与关键字比较。比较函数的形式必须为：

- 若 `arg1` 小于 `arg2`，则返回值小于 0。
- 若 `arg1` 等于 `arg2`，则返回值为 0。
- 若 `arg1` 大于 `arg2`，则返回值大于 0。

数组必须按升序排列，其最小地址存放最小元素。

若数组中不含关键字，返回一个空指针。

示例 本程序读取来自键盘的输入(假定缓冲键盘 I/O)，并确定它们是否是字母：

```
#include <stdlib.h>
#include <ctype.h>
#include <stdio.h>
int comp();

char *alpha="abcdefghijklmnopqrstuvwxyz";

main(void)
{
    char ch;
    char *p;
    int comp();

    do {
        printf("enter a character: ");
        scanf("%c",&ch);
        ch = tolower(ch);
        p = (char *) bsearch(&ch, alpha, 26, 1, comp);
        if(p) printf("is in alphabet\n");
        else printf("is not in alphabet\n");
    } while(p);
    return 0;
}

/* compare two characters */
comp(const char *ch, const char *s)
{
    return *ch-*s;
}
```

```
#include <stdlib.h>
```

```
void exit(int status)
```

说明 `exit()` 函数引起程序的立即正常终止。

若环境支持，则 `status` 的值被传递给调用过程——通常是操作系统。通常，若 `status`



的值为 0，则假定正常程序终止，一个非零值可用来指示出错。

示例 本函数为一通讯录程序执行菜单选择。若选择 Q，程序将终止：

```
menu(void)
{
    char choice;

    do {
        printf("Enter names (E)\n");
        printf("Delete name (D)\n");
        printf("Print (P)\n");
        printf("Quit (Q)\n");
    } while(!strchr("EDPQ",toupper(ch)));
    if(ch=='Q') exit(0);
    return ch;
}
```

#include <stdlib.h>

char \*itoa (int num, char str, int radix)

说明 本函数当前没有被 ANSI C 标准定义。

itoa()函数把整型 num 转换成等价串并把结果放在指示的串中。输出串的基由 radix 确定，其可以在 2 到 36 之间。

itoa()函数返回一个指向 str 的指针。通常，str 没有错误返回值。注意要用足够长的串来调用 itoa()以存放被转换而来的结果。

itoa()主要的用途是把整型数据换成串，以便可把它们送往不由一般 C I/O 系统直接支持的设备——即一个非法设备。用 sprintf()可以完成同样的任务。在这里讨论 itoa()的原因在于其在已有的老代码中用得很多。

示例 本程序以十六进制显示值 1423(58F)。

```
#include <stdlib.h>
#include <stdio.h>

main(void)
{
    char p[20];

    itoa(1423, p, 16);

    printf(p);
    return 0;
}
```

#include <stdlib.h>

long labs(long num)

说明 labs()函数返回 long int 型 num 的绝对值。

示例 本函数把用户输入数转换成其绝对值。

```
#include <stdlib.h>
#include <stdio.h>

long int get_labs()
```

```

{
    char num[80];

    gets(num)

    return labs(atol(num));
}

```

**#include <setjmp.h>**

**void longjmp(envbuf, val)**

**jmp\_buf envbuf; int val**

**说明** longjmp() 函数引起程序执行在上一个 setjmp() 调用点恢复。这两个函数是 Turbo C++ 提供函数间转移的方法。注意要求头文件 setjump.h。

longjmp() 函数通过重置如 envbuf 中描述的栈来工作，该栈已由前一个 setjmp() 调用设置。这引起程序执行在 setjmp 调用之后的语句处恢复。即，计算机看上去似乎其从没有离开调用 setjmp() 的函数，生动点说，longjmp() 函数弯曲地越过时间和空间(存储空间)到达程序中的前一点而没有按照正常的函数返回过程执行。

缓冲区 envbuf 有 jmp\_buf 类型，其在头文件 setjmp.h 中定义。缓冲区必须通过在调用 longjmp() 之前调用 setjmp() 加以设置。

longjmp() 函数必须在调用 setjmp() 的函数返回前调用，这一点是很重要的。若不是这样，其结果不能确定。

显然，longjmp() 最为广泛的用途是当一个毁灭性错误发生时从一个嵌入很深的子程序中返回。

**示例** 本程序打印 1 2 3:

```

#include <setjmp.h>
#include <stdio.h>

void f2(void);

jmp_buf ebuf;

main(void)
{
    char first='1';
    int i;
    printf("1 ");
    i = setjmp(ebuf);
    if(first) {
        first = 'f';
        f2();
        printf("this will not be printed");
    }
    printf("%d", i);
    return 0;
}

void f2(void)
{
    printf("2 ");
    longjmp(ebuf, 3);
}

```

```
#include <stdlib.h>
```

```
void qsort(void *base, size_t num, size_t size,  
           int (*compare) (const void *, const void *))
```

说明 qsort()函数用快速排序法排序 base 指示的数组。快速排序法通常被认为是最好的通用排序算法。到最后时数组将被排序完毕。数组中的元素个数用 num 指示每个元素的大小(用字节计)用 size 给出。

compare 指示的函数用来把数组中的一个元素与关键字比较。比较函数的形式必须为:

```
func_name(const void *arg1, const void *arg2)
```

其必须返回下列值:

- . 若 arg1 小于 arg2, 则返回值小于 0。
- . 若 arg1 等于 arg2, 则返回值等于 0。
- . 若 arg1 大于 arg2, 则返回值大于 0。

数组按升序排列, 最低地址存放最小元素。

示例 本程序排序一个整数表, 并显示结果:

```
#include <stdlib.h>  
#include <stdio.h>  
  
int comp();  
  
int num[10] = {  
    1, 3, 6, 5, 8, 7, 9, 6, 2, 0  
};  
  
main(void)  
{  
    int i;  
  
    printf("original array: ");  
    for(i=0; i<10; i++) printf("%d ", num[i]);  
  
    qsort(num, 10, sizeof(int), comp);  
  
    printf("sorted array: ");  
    for(i=0; i<10; i++) printf("%d ", num[i]);  
    return 0;  
}  
  
/* compare the integers */  
  
comp(const int *i, const int *j)  
{  
    return *i - *j;  
}
```

```
#include <stdlib.h>
```

```
int rand(void)
```

说明 rand()函数生成一个随机数系列。每次调用它, 都返回一个 0 到 RAND\_MAX 之间的整数。

示例 本程序显示十个随机数:

```
#include <stdlib.h>
#include <stdio.h>

main(void)
{
    int i;

    for(i=0; i<10; i++)
        printf("%d ",rand());
    return 0;
}
```

**#include <setjmp.h>**

**int setjmp(jmp\_buf envbuf)**  
          **jmp\_buf envbuf)**

**说明** setjmp()函数把系统栈内容保存在缓冲区 envbuf 中以便以后 longjmp()使用。

在调用时, setjmp()函数返回 0。然而, 当在执行时, longjmp()把一个参数传递给 setjmp(), 并且在调用 longjmp()之后, setjmp()的值正是该值(总为非零)。

详见 longjmp()。

示例 本程序打印 1 2 3:

```
#include <setjmp.h>
#include <stdio.h>

void f2(void)
{
    jmp_buf ebuf;

    main(void)
    {
        char first=1;
        int i;

        printf("1 ");
        i = setjmp(ebuf);
        if(first) {
            first = !first;
            f2();
            printf("this will not be printed");
        }
        printf("%d", i);
        return 0;
    }

    void f2(void)
    {
        printf("2 ");
        longjmp(ebuf, 3);
    }
}
```

**#include <stdlib.h>**

**void srand(unsigned seed)**

**说明** srand()函数用来设置 rand 函数生成系列的起始点。rand 函数返回随机数系列常用

于支持多程序运行。

示例 本程序利用系统时间来随机地初始化使用 srand()的 rand()函数。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* Seed rand with the system time
   and display the first 10 numbers.
*/
main(void)
{
    int i, utime;
    long ltime;

    /* get the current calendar time */

    ltime = time(NULL);
    utime = (unsigned int) ltime/2;
    srand(utime);
    for(i=0; i<10; i++) printf("%d ", rand());
    return 0;
}
```

## 附录 B 使用 Turbo C++ 调试器

Turbo C++ 在其集成环境中包含了一个内部源级调试器, 本附录介绍调试器并讨论其一些最重要的特征。

### B.1 为调试准备程序

尽管按一个键便可使用 Turbo C++ 的调试器, 但必须记住程序在编译时需为调试作好准备。为使调试器工作, 必须把调试信息包含在程序的可执行文件中。缺省时, 适当的调试信息会自动地包含。控制调试信息之包含的选项在 Options 主菜单项的 Debugger 项下。包含在程序已编译版本中的调试信息能够帮助 Turbo C++ 把源代码连接到其目标码上。

### B.2 什么是源级调试器

在了解什么是一个源级调试器以及它为什么非常有价值前, 有必要先了解一个传统调试器是如何工作的。一个传统的调试器设计成提供目标代码调试, 其间可以控制 CPU 的寄存器或存储器的内容。要使用一个传统的调试器, 连接程序产生一个符号表, 它显示每一个函数的存储地址及存储器中的变量。要调试一个程序, 可使用该符号表并开始执行用户程序, 控制各寄存器和存储地址中的内容。大多数调试器允许单步调试程序, 每次一条指令, 并可在目标码中设置断点。传统调试器最大的缺点是程序的目标码通常与源代码相差很大。因而即使使用一长符号表, 也很难知道实际上执行的是什么。

源级调试器在原有的传统形式上有了很大程度的提高。一个源级调试器能利用源代码调试程序。调试器能够自动地把与每行相对应的已编译目标码与其对应的源代码连接起来。不再需要一长符号表, 可以通过在源代码中设置断点来控制程序的执行。还可以用变量名来观察各变量值, 也可以每次一条语句地单步调试程序并可观察程序调用栈中的内容。而且与 Turbo C++ 调试器的通讯是由类似 C 的表达式完成的, 因而没有什么新的东西还需要去学习。

### B.3 调试器的基本内容。

本节介绍最一般的调试命令。在开始之前先输入下列程序。其将用来作演示:

```
#include <iostream.h>

void sqr_it(int n);

main(void)
{
    int i;

    for(i=0; i<10; i++) {
        cout << i << " ";
        sqr_it(i);
    }
    return 0;
}
```

```

void sqr_it(int n)
{
    cout << n*n << " ";
}

```

输入该程序后，编译并运行之，以确认已正确地把它输入，其打印 0 到 9 的值以及它们的平方。

### B.3.1 单步调试

单步调试是每次执行一条语句的过程。用 Turbo C++ 作单步调试，可按 F7 键。Turbo C++ 中把 F7 键称作跟踪键，现按 F7，注意，含有 `main()` 函数说明的行呈亮度显示，这便是程序执行的开始之处。还要注意，行 `#include<iostream.h>` 和 `sqr_it()` 的原型被跳过。不产生代码的语句，如预处理指令，显然不被执行，因而调试器自动跳过它们。一个变量说明不是一条可跟踪的动作语句。因而 在单步调试时，变量说明语句也被跳过（但提供初始化的变量说明被跟踪）。

注：按 F7 与选择 Run 主菜单选项中的 Trace into 选项相同。

按 F7 几次，注意高亮条一行一行移动。当函数 `sqr_it()` 被调用时，亮条移入该函数并从之返回。F7 键引起程序执行进入函数调用跟踪。然而，有时候可能希望仅观察一个函数中的代码执行。此时，用 F8 键。按此键时，执行一条语句，但函数调用不被跟踪，当仅希望观察一个函数内容的情况时，F8 键是很有用的。按 F8 键与选择 Run 选项菜单中的 Step over 项是一样的。

现在用 F8 尝试一下。注意，亮条不会进入 `sqr_it()` 函数。

### B.3.2 设置断点

尽管单步调试是很有用的，但对于一个大程序来说它可能是很烦琐的——尤其是当希望调试的代码段深陷程序中时。为了避免重复地按 F7 或 F8 以达到希望调试的部分，在该部分之首设置一个断点是很容易并且是一个较好的方法。断点，顾名思义，即是程序执行的一个中断处，当执行到该点时，程序停止运行并将控制返回到调试器，允许你检查特定变量的值或开始子程序的单步调试。

要设置一个断点，把光标移到程序中相应的行，然后，激活 Debug 菜单并选择 Toggle Breakpoint 选项（还可以使用热键 CTRL-F8）。要设置断点的代码行将用高亮条或另一种颜色显示，这依赖于系统中视频适配器和监视器类型。记住，在一个程序中，可以设置多个断点。

一旦定义了一个或多个断点，用 Run/Run 选项执行程序。程序将一直运行到遇到一个断点。要看看这个动作，在 `sqr_it()` 中的下列行设置一个断点：

```
cout << n*n << " ";
```

并运行程序。正如你所看到的，执行在该行停止。

要删除一个断点，把光标放在希望删除的断点所在行并选择 Debug/Toggle breakpoint 选项（或按 CTRL-F8）。可以根据需要重复地变换断点的设置和删除。

### B.3.3 监视变量

在调试过程中,经常需要在程序执行时监视一个或多个变量的值。用 Turbo C++ 调试器做这项工作是很容易的。定义一个待监视的变量,选择 Debug/Watches 选项,然后并选择 Add watch(或按 CTRL-F7)。将看到有一个小窗口弹出。在该窗口中输入想要监视的变量名。调试器将自动地在监视窗口显示变量值。若该变量是全局的,则其值总可以提取。但若该变量是局部的,则其值仅可以在含有它的函数正在被执行时才可以提取。当执行进入另一个函数时,该变量值则不可知。记住,若有两个函数都使用了同一个变量名,则所看到的值将对应当前正在执行的函数。

让我们来看另一个例子。现激活 Watches 项。要观察例子程序中 i 值,输入 i。若该程序目前不在执行,或者执行在 `sq_root` 函数中停止,则看到如下信息:

Undefined symbol 'i'

但当执行时入 `main()` 函数时, i 的值将被显示。

你不但能够监视变量的内容,还能够在监视含有这些变量的任何合法 C 表达式。但这两个限制:第一,该表达式不能调用函数;第二,其不可以使用任何 `#define` 值。

### B.3.4 监视表达式格式码

Turbo C++ 调试器允许使用格式码使监视表达式输出格式化。用下列一般形式指定格式码:

`expression format_code`

该格式码列于表 B-1 中,若没有指定格式码,则调试器将自动提供一个缺省的格式,其通常是正确的。

可以用两种方法显示整数:用十进制或用十六进制。调试器能够自动区分 `long` 和 `short` 整数之间的区别,因为它要访问源代码。

表 B-1 调试格式码

格式码	含义
C	不作改变地显示一个字符
D	按十进制显示
F	按浮点显示
H	按十六进制显示
M	显示存储空间
P	显示指针
R	显示结构或联合名及值
S	作适当改变后显示一个字符
X	按十六进制显示(与 H 相同)

当指定一个浮点格式时,可以在 F 格式之后添上一个数以告诉调试器在小数点后保留几位有效数。例如,若 `average` 是一个 `float`,以下语句就告诉调试器保留 5 个有效位。



average.F5

记住,该数字是可选的。调试器能够自动区分 floats 与 doubles 之间的区别。

指针用段/偏移量显示。但一个 near 指针不显示段值,而代之以 DS,因为所有 near 指针停留在数据段中;另一方面,far 指针用全段/偏移显示。可以通过把\*操作符置于监视表达式中指针前面显示所指向的值。

字符数组作为串显示。缺省时,调试器把非 ASCII 字符转换成代码。例如,CTRL-D 被显示为\4。但若指定了 C 格式代码,则所有字符用 PC 扩展字符集按原样显示。

当一个结构成一个联合被显示时,与每个域相关的值用适当的格式显示。若包含了 R 格式命令,则每个域的名也被显示。试看一个例子,输入下列程序,观察 sample 和 sample, R。

```
#include <string.h>
struct inventory {
    char item [10];

    int count;
    float cost;
} sample;

main(void)
{
    strcpy(sample.item, "hammer");
    sample.count = 100;
    sample.cost = 3.95;

    return 0;
}
```

正如你所期望的,可以监视一个对象。在监视一个对象时,将看到包含在对象中的任何数据值。像结构和联合,若使用了 R 格式指示符,每个数据项的名字都被显示。

使用 Debug/Watches 菜单,可以删去、修改或删除观察表达式。缺省时,若修改一个表达式,则被修改的表达式是最后输入的那个。可以指定哪一个表达式要修改。首先移至监视窗口,然后把亮条移到要修改的表达式。最后,激活在 Watches 菜单上的 Edit watch 选项。

### B.3.5 监视栈

在程序执行中,可以使用 Debug 菜单下的 call stack 选项来显示调用栈的内容。该选项将按程序中各被调函数的顺序显示。它不显示任何变量或返回地址,在调用时显示任一函数参数的值。为了解该功能是如何工作的,输入下列程序

```
#include <iostream.h>
void f1(void), f2(int i);
main(void)
{
    f1();
    return 0;
}

void f1(void)
{
    int i;
    for(i=0; i<10; i++) f2(i);
}
```

```
void f2(int i)
{
    cout << "in f2, value is " << i << " ";
}
```

2.1) 中程序含有 `cout` 语句的行设置断点。第一次遇到该断点时, 调用栈将有如下内容:

```
f2(0)
f1(0)
main()
```

### B.3.6 计算一个表达式

可以选择 **Debug** 菜单上的 **Evaluate/modify** 选项来计算任何合法的 C 表达式。可以把正在调试的程序中定义的变量作为表达式的一部分, 还可以利用该选项改变一个变量的值。但不能调用任何函数或使用任何 `#defined` 值。

### B.3.7 检测一个变量

尽管用 **Watches** 选项监视一个变量通常已足够, 但在有些情况下, 可能还需要进一步控制变量的变化。要做这样的工作, 选择 **Debug** 菜单上的 **Inspect** 选项。该选项使得一个变量的内容和地址被显示。当有类似没有指向任何地方的指针(`wild pointer`)那样的错误时, 了解一个变量的地址是很有用的。

### B.3.8 使用寄存器窗口

你所拥有的最后一个调试工具便是 **Turbo C++** 的寄存器。若选择了 **Window** 主菜单项, 然后又选择了 **Register** 项, 将有一个小窗口弹出, 它显示 CPU 中的每一个寄存器的内容以及每个标志的状态。当在进行跟踪或每当遇到一个断点时, 寄存器窗口的内容将改变以反映 CPU 寄存器中的值。

## 附录 C Turbo C++ 的存储模式

在 Turbo C++ 中, 可以用 8086 处理器系列所定义的六种不同的存储模式来编译一个程序。每种模式按不同方式组织计算机的存储空间并管理程序所拥有的代码大小或数据大小, 或其两者。它还决定程序的执行速度。由于所用模式对用户程序的执行速度和其访问系统资源的能力都有很大的影响, 本附录将详细讨论各存储模式。

### C.1 8086 处理器系列

在了解各存储模式工作方法之前, 需要了解 8086 处理器族是如何寻址的。在本附录中 CPU 将针对 8086, 但其内容可用在该系列中的所有处理器上, 包括 8083, 80186, 80286 和 80386(对于 80286 和 80386, 后面的内容仅适于在 80386 仿真方式下运行的处理器)。

8086 含有 14 个寄存器, 信息被放于其中以备处理或用于程序控制。寄存器分为如下几类:

- 通用寄存器
- 基指针和索引寄存器
- 段寄存器
- 专用寄存器

8086 CPU 中的所有寄存器都是 16 位的(双字节)。

通用寄存器是 CPU 的工作寄存器。值正是被放在这些寄存器中以便处理, 包括算术操作, 如加或乘; 比较, 包括等于、小于和大于等等; 还有分支(转移)指令。每一个通用寄存器都可以用两种方式访问: 作为 16 位寄存器或作为两个 8 位的寄存器。

基址寄存器和索引寄存器提供对相对寻址、栈指针和块移动指令的支持。

段寄存器帮助 8086 实现段存储方案。CS 寄存器存放当前代码段。DS 寄存器存放当前数据段; ES 寄存器存放附加段; SS 寄存器存放堆栈段。

专用寄存器包括标志寄存器, 其存放 CPU 状态; 还有指令指针, 它指向 CPU 要执行的下条指令。

### C.2 地址计算

8086 总的地址空间为 1 兆字节(在该系列中较高级的 CPU 能有更大的地址空间, 但在 8086 仿真方式下却没有)。要访问 RAM 的 1 兆字节需要 20 位地址。但在 8086 中, 没有寄存器多于 16 位。这就是说 20 位的地址必须分在两个寄存器中。不幸的是, 分离 20 位的方法比人们想象的要复杂些。

对于 8086, 所有地址由一个段和一个偏移量构成。实际上, 8086 所用的寻址方法通常称作段/偏移方法。一个段是一个 64K 的 RAM 区域, 它必须始于一个 16 位的偶数倍地址。在 8086 的术语中, 16 字节叫做节(paragraph)。因而你有时会遇到术语节边界(paragraph boundary); 用于表示这些 16 字节的偶数倍地址。8086 定义了四个段: 代码段、数据段、堆栈段和附加段(这些段可能相互重叠, 也可能彼此分开)。段中任一字节的位置由偏移量确定。段寄存器中的值确定哪一个 64K 段被访问, 偏移量值确定段中哪个字节被实际寻

址。因而计算机中特定位的 20 位物理地址是段和偏移量的组合。

### C.3 近指针及远指针

若需要访问当前段内的地址,则只需把地址偏移量装入寄存器中即可。也就是说仅用 16 位地址访问的对象必须在当前段中。这称作近地址,或近指针。

若要访问不在当前段中的地址,所需地址的段及偏移量必须装入。这就称作远地址,或远指针。一个远指针可以访问 1 兆字节地址空间中的任何一个地址。

为了访问当前段中的存储空间,只需将 16 位偏移量装入。然而,若希望访问段外的存储空间,则段和偏移量都必须装入在对应的寄存器中。由于装入两个 16 位寄存器的时间比装入一个的时间多一倍,所以装入一个长指针的时间比装入短指针的时间长。因此,使用远指针的程序运行将比使用近指针的程序慢。而且用远指针将导致程序增大,但远指针可用于支持较大的程序和数据。

### C.4 存储模式

使用 8086 处理器系列的 Turbo C++ 可用六种方法编译程序,并且每一种方法都以不同方式组织计算机中的存储空间。每一种组织形式都影响程序执行的不同方面。这六种模式分别叫做:微模式、小模式、中模式、紧缩模式、大模式和巨模式。让我们来看看它们的区别。

#### C.4.1 微模式(Tiny Model)

在微模式下编译程序,所有段寄存器被置成相同的值,并且全按 16 位寻址(近指针)。这就是说代码、数据和栈都必须在同一个 64K 段内,这种编译方法生成的是最小、最快的代码。用该存储模式编译的程序可被转换成 COM 文件。

#### C.4.2 小模式(Small Model)

小模式是 Turbo C++ 的缺省编译模式,并且对于很多任务都是很有用的。尽管所有寻址都用 16 位偏移量,但代码段、数据段、堆栈段和附加段是分开的,它们都位于自己的段中。这就是说一个用这种方法编译的程序总规模是一个代码、数据之和最大为 128K 的程序。由于小模式仅用近指针,因而执行速度与微模式的执行速度同样令人满意,但程序规模则可比之大一倍左右。

#### C.4.3 中模式(Medium Model)

中模式是相对于大模式的,其代码超过了小模式的 1 段的限制。这里,代码可使用多个段并需要 20 位(远)指针,但栈、数据和附加段在其自己的段中,并用 16 位(近)寻址。这对于使用数据少的大程序来说是很好的。在进行函数调用时,程序运行得较慢,但对数据的引用与小模式同样快。

#### C.4.4 紧缩模式(Compact Model)

紧缩模式是中模式的补充模式。用这种模式时,程序代码被限制在一个段中,而数据则可占用几个段。这就是说所有对数据的访问需要 20 位(远)寻址,但代码用 16 位(近)寻

址。这对于要求数据量大但代码少的程序来说是很好的。除了在引用数据时运行较慢外,程序的运行将与小模式同样的快。

#### C.4.5 大模式(Large Model)

大模式允许代码和数据都可使用多个段——但最大的数据单项,如一个数组,被限制在 64K 内。当有大量代码和大量数据要求时用该模式,其运行速度比前面的所有模式都慢。

#### C.4.6 巨模式(Huge Model)

巨模式除了允许单个数据项可超过 64K 之外,其与大模式是相同的。使得子程序的运行速度更进步地降低(下一节中你将看到其原因)。

#### C.4.7 模式选择

通常,应该使用小模式,除非有别的原因不这样。若程序量大而数据量少,则选用中模式。若数据量大而程序量小,则选紧缩模式。若代码量大,数据量也大,则选用大模式——除非需要一个大于 64K 的单个数据项,在这种情形下,应选用巨模式。记住,大模式和巨模式的运行实际上慢于其它模式。

还有另一个考虑因素会影响程序的编译,若用紧缩模式或大模式编译程序,则所有指针数据的引用将通过远指针。然而,这就产生了问题。首先,多数指针比较将不会生成正确的结果。产生错误结果的原因是一个以上的段/偏移对可能会映射到相同的物理地址。当比较远指针时,仅有偏移量被检查。这就是说,两个指针可能实际指向的是同一个物理地址,但比较结果都认为是不同的,或者指向不同地址的指针比较结果却认为是相同的。对于远指针,保证正确的唯一比较是与(空指针)比较。

对于远指针的第二个问题是,当远指针值增加或减少时,仅有偏移量改变。这就是说当增加(或减少)越过一个段边界时,指针将会卷绕。

用巨模式编译所生成的指针称为巨型指针。从使用全 20 位寻址方面看,它们与远指针是相同的,但它们又不受远指针那样的限制。首先,巨型指针可以正确地进行比较,其原因是它们被规范化。规范化过程保证了对应每一个物理地址只有一段/偏移地址。因而,所有比较都是有效的,但规范化过程要花时间,因而这就减慢了执行速度,其次,当指针的增大或减小越过一个段边界时,该段则作相应调整,并且远指针所产生的指针卷绕问题便不存在。这就是如何能访问一个大于 64K 单个数据对象成功的原因所在。

#### C.4.8 存储模式编译选项

缺省时, Turbo C++ 用小模式编译程序。要使 Turbo C++ 用别的模式,必须给出适当的指令。在集成环境中,可以用 Options/Compiler 菜单选择存储模式。对于命令行版本,可用如下命令行选项之一:

选项	存储模式
-mc	紧缩模式
-mh	巨模式
-ml	大模式

<b>-mm</b>	中模式
<b>-ms</b>	小模式
<b>-mt</b>	微模式

## C.5 强制转换存储模式

你可能在前一章就考虑过,对另一段中的数据即使只访问一次也必须用紧缩模式而不能用小模式编译程序,这未免有些遗憾。在这种情形下,虽然程序中仅有一部分需要用远指针,但整个程序的执行速度也会大大降低。通常,这种情况有不同的表现。例如,要用 20 位寻址访问 PC 的视频 RAM,而程序的其余部分可能仅需近指针。

解决这一问题及其相关问题的方法是引进强制类型修饰符,它们是 Turbo C++ 的改进之处。其为: **near**、**far** 和 **huge**。

当这些修饰符应用于指针时,它将影响数据的访问方式。还可以把 **near** 和 **far** 修饰符应用于函数,在这种情形下,它们影响函数被调用和返回的方法。

这些修饰符跟在基本类型之后,而置于变量名之前。例如:下面是对一个称作 **f\_pointer** 的 **far** 指针进行说明:

```
char far *f_pointer;
```

下例中,函数 **myfarfunc()** 被说明为 **far**:

```
void far myfarfunc(int *p);
```

现在让我们来看看这些类型修饰符。

### C.5.1 far

最普遍的强制存储模式是 **far**。原因在于通常需要访问在数据段之外的某个存储区。但若程序是在大数据模式下编译的,所有对数据的访问都变得很慢——而不仅此一种。解决这一问题的方法是显式说明指向当前数据段之外的数据指针为 **far**,并用小存储模式编译程序。在这种方法中,仅有那些对实际位于缺省数据段之外的对象的引用才引起附加的内部操作。

**far** 函数的使用较少,其通常是在一个函数位于当前代码段之外的特定情形下才用,如一个基于 ROM 的子程序。在这种情况下,用 **far** 能保证使用正确的调用和返回系列。

显式说明 **far** 指针同样也有大数据模式下编译时隐式生成的麻烦。首先,指针的算术运算只影响偏移量,并且会导致指针重复。这就是说,若一个值为 0000:FFFF 的 **far** 指针增大时,其新值将为 0000:0000,而不是 1000:0000。段值永远不会改变。其次,两个 **far** 指针不应该用在一个关系表达式中,因为只有它们的偏移量被检查。两个实际指向相同物理地址的不同的指针,可能会有不同的段号和偏移量。若要比 20 位指针,必须用巨型指针。但可以将一个 **far** 指针与一个空指针比较。

### C.5.2 near

**near** 指针是一个 16 位的偏移量,它用适当的段号来确定实际存储位置。数据的 **near** 修饰符使 Turbo C++ 把指针作为相对于 DS 段的 16 位偏移量。当用中模式、大模式或巨模式编译程序时,可用一个 **near** 指针。

把 **near** 用于函数则使得该函数被当作其是用小代码模式编译的(该函数地址是用 CS

寄存器计算的)。当一个函数是用微模式、小模式或紧缩模式编译时，所有对该函数的调用都会把一个 16 位的返回地址置于栈上。用大代码模式编译的函数将导致一个 20 位地址被堆入栈中。因而，在用大代码模式编译的程序中，递归的函数应该用 `near` 说明(若可能的话)，以保留栈空间并减少执行时间。

### C.5.3 huge

`huge` 修饰符只能用于数据，而不能用于函数。一个 `huge` 指针如同一个在巨模式下编译所生成的指针一样。它被规范化，以使两个 `huge` 指针的比较有意义。当一个 `huge` 指针增大时，其段号和偏移量都有可能改变——其不会有如 `far` 指针那样的指针卷绕问题。用一个 `huge` 指针，可以访问大于 64K 的对象。

## C.6 Turbo C++ 的段指示符

除了 `near`、`far` 和 `huge` 之外，Turbo C++ 还支持下列四种另外的寻址修饰符：

```
_cs  
_ds  
_ss  
_es
```

当这些类型修饰符应用于指针的说明时，就使得指针成为特定段中的 16 位偏移量。

例如，给出如下语名：

```
int _es *ptr
```

`ptr` 将使得附加段含有一个 16 位偏移量。

Turbo C++ 还包含 `_seg` 修饰符，它建立 16 位长且仅含段地址的指针，偏移量被置为零。对 `_seg` 指针有几个限制。不能增大或减少它们，在一个表达式中，其对一个 `_seg` 指针增加或减去一个整型值，一个 `far` 指针将生成。当间接引用 `_seg` 指针时，其将会转变成一个 `far` 指针。可以将 `_seg` 指针加上一个 `near` 指针，结果将是一个 `far` 指针。

这些修饰符通常只用于特殊的情况。

记住，Turbo C++ 的 `near`、`far`、`huge`、`_es`、`_ds`、`_ss` 和 `_seg` 修饰符不是 ANSI 标准定义的，并且不是全部可移植的。但多数基于 8086 的 C 编译器即使不支持全部，也会支持其中一部分修饰符。

## 附录 D 使用 VROOMM 覆盖技术

在对 Turbo C++ 的介绍中, Borland 公司使所有的程序员都可以使用运行时刻面向对象的存储管理技术(VROOMM, Virtual Runtime Object Oriented Memory Manager)。或许正如你所知道的, 覆盖是一种处理那些程序过大而造成存储空间不够的技术。它以时间为代价, 当使用覆盖技术后, 仅当需要时, 程序块才在内存与磁盘之间进行交换。这就减少了程序对内存空间的要求量。由于要花费时间去装入模块, 然而它减慢了程序的执行速度。

通常, 若要使用覆盖技术, 需要一个覆盖管理器, 它把用户程序的不同部分从磁盘装入内存。还需要把程序分成几个较小的块以便覆盖技术使用。VROOMM 特殊之处在于它能自动地控制覆盖的细节。你不需要计算任何大小或确定调用依赖关系。实际上, 要使大程序利用 VROOMM, 仅需修改少量的编译选项, 而不用作任何别的修改。

当编译覆盖程序时, 必须使用中模式、大模式或巨模式。若忘了这么做, 则连接程序将不做连接工作。

由于一般只对一个很大的程序使用 VROOMM 技术, 所以很有可能需要将程序分在几个文件中。要使用 VROOMM, 首先记住全菜单是活动的, 并选择主菜单上的 Options 选项。然后, 选择 `_mode` 之后的 Compiler。打开 Overlays 检查框, 并选择中存储模式、大存储模式或巨型存储模式, 返回到主菜单。现在再次激活 Options 并选择 Linker。打开 Overlays 检查框。最后, 从主菜单选择 Project 项和 Local Options 这将让你指定希望什么文件用覆盖技术编译:

若使用 Turbo C++ 的命令行版本, 则必须在所希望的文件之前指定一个选项。如果希望程序的某一部分始终驻留内存, 则可在其文件名之前使用 `-Y`。由于必须使用中模式、大模式或巨型模式来编译程序, 还必须指定 `_mm`, `_ml` 或 `_mh` 选项。例如, 下条命令行将编译由文件 PROG.CPP、PROG1.CPP 和 PROG2.CPP 组成的程序。注意 PROG.CPP 永驻内存, 其余两个文件被覆盖。

```
tcc -ml -c -Yo prog1.cpp
tcc -ml -c -Yo prog2.cpp
tcc -ml -Y prog.cpp prog2.obj prog3.obj
```

VROOMM 工作过程如下: 当一段代码被调用时, 如果它不在内存中, 则将其从磁盘读入到一个为覆盖而设置的缓冲区中。若缓冲区已满, 则另一个模块将被丢弃。缺省时, Turbo C++ 为覆盖操作设置的缓冲区大小是最大可覆盖模块大小的两倍, 但你可以通过设置全局变量 `_overbuffer` 来增加缓冲区大小。缓冲区大小以节(16 个字节)计。试验是确定程序最适应的缓冲区尺寸的最佳途径。记住, 要不是考虑到内存空间有限, 使缓冲区尽可能的大是没有错的, 这能够减少对磁盘的访问。

当使用覆盖技术时, 记住总是把程序的某些部分留驻在内存, 而另一些部分作为覆盖。作为覆盖之用的最佳候选模块是那些偶尔执行的模块, 如一个帮助系统或一个排序算法, 但不能把依赖于时钟的模块作为覆盖模块。



## 附录 E 使用命令行编译程序

若你是一个 Turbo C 和 C++ 新手, 无疑将会发现 Turbo C++ 集成环境是开发程序的最为方便的工具。但若用编辑器进行程序设计有一段时间了, 可能会发现 Turbo C++ 的命令行版本更合你的胃口。对于有经验的程序员来说, 命令行版本体现了编译和连接的传统方法。命令行编译程序名为 TCC.EXE。本附录将对它作简单的介绍。

### E.1 用命令行编译程序编译

假定有一个叫做 X.CPP 的程序。若要用 Turbo C++ 的命令行版本编译该程序, 可用如下命令:

```
C>TCC X.CPP
```

假设程序没有错误, 就使得 X.CPP 被编译, 并与适当的库文件连接。这是命令行最简单的形式。

命令行的一般形式为:

```
TCC [option1 option2...optionN] frame1 frame2...frameN
```

这里, option 表示一个编译或连接选项, frame 为一个 C 或 C++ 源文件, OBJ 文件或一个库。

所有的编译/连接选项都以减号打头。通常, 在一个选项之后跟有一个减号即是关闭该选项。表 E-1 给出了 Turbo C++ 命令行版本的各选项。记住, 这些选项是大小写有关的。

例如, 要在检查栈溢出情况下编译 X.CPP, 可用如下命令行:

```
C>TCC -N X.CPP
```

表 E-1 命令行选项

选项	含意
-A	仅识别ANSI关键字
-AK	仅识别K&R关键字
-AU	仅识别UNIX C关键字
-a	用数据字对齐
-a-	用数据字节对齐
-B	源文件中包含汇编码
-C	允许嵌套注释
-c	仅编译成.OBJ文件
-Dname	定义一个宏名
-Dname=string	给一个宏名定义和给定一个值
-d	合并重复串

---

<b>-d-</b>	不合并重复串
<b>-Exxx</b>	指定汇编程序名
<b>-efname</b>	指定可执行文件名
<b>-f</b>	浮点仿真
<b>-ff</b>	快速浮点优化
<b>-f-</b>	不用浮点
<b>-f87</b>	用8087
<b>-f287</b>	用80287
<b>-G</b>	速度优化
<b>-gN</b>	N个警告信息后停止
<b>-Ipath</b>	指定包含且录路径
<b>-iN</b>	指定标识符长N
<b>-jN</b>	N个致命错误之后停止
<b>-K</b>	字符为无符号的(unsigned)
<b>-K-</b>	字符为有符号的(signed)
<b>-k</b>	用标准栈框架
<b>-Lpath</b>	指定库目录
<b>-lx</b>	传递一个选项给连接程序
<b>-M</b>	建立映射文件
<b>-mc</b>	用紧缩存储模式
<b>-mh</b>	用巨型存储模式
<b>-ml</b>	用大存储模式
<b>-mm</b>	用中存储模式
<b>-ms</b>	用小存储模式
<b>-mt</b>	用微存储模式
<b>-N</b>	检查栈溢出
<b>-Npath</b>	指定输出目录
<b>-O</b>	优化转移指令
<b>-P</b>	作为C++程序编译
<b>-p</b>	用Pascal调用约定
<b>-p-</b>	用C调用约定
<b>-Qe</b>	用所有EMS存储空间
<b>-Qe-</b>	不用EMS存储空间
<b>-Qx</b>	类似-Qe
<b>-r</b>	用寄存器变量
<b>-r-</b>	不用寄存器变量
<b>-rd</b>	仅用已说明的寄存器变量
<b>-S</b>	生成汇编码输出
<b>-T</b>	把一选项传递给汇编程序
<b>-Uname</b>	撤消一个宏名的定义

---

<b>-u</b>	产生下划线
<b>-Vx</b>	指定虚表选项
<b>-v</b>	包含调试信息
<b>-w</b>	显示警告信息
<b>-w-</b>	不显示警告信息
<b>-Y</b>	程序含有覆盖模块
<b>-Yo</b>	作为覆盖模块编译
<b>-y</b>	行号嵌入到目标码
<b>-Z</b>	寄存器优化
<b>-z</b>	指定段名
<b>-l</b>	生成80186/80286指令
<b>-l-</b>	不生成80186/80286指令
<b>-2</b>	生成80286保护方式指令

### E.1.1 文件名组成

若没有指定扩展名, Turbo C++ 命令行编译程序自动地把.C 扩展名加到文件名中。例如, 下面两个命令行在功能上是相同的:

```
C>TCC X.C
```

```
C>TCC X
```

这使得程序作为 C 而不是 C++ 编译。若希望用 C++ 编译程序, 文件必须使用.CPP 扩展名(并且必须显式指定之)或使用-P 编译选项。

可以通过指定一个不为.C 或.CPP 的扩展名来编译一个文件。例如, 若要编译 X.TMP, 命令行将为:

```
C>TCC X.TMP
```

没有.C 或.CPP 扩展名的文件将作为 C 程序加以编译。

可以在源文件之后指定将连接到正在编译的源文件中的附加目标文件。所有包含在这些文件中的内容必须预先编译好并且有.OBJ 扩展名。例如, 若用户程序由文件 P1.CPP、P2.CPP 和 P3.CPP 组成, 并且 P2 和 P3 已编译到.OBJ 文件中, 则下列命令行将首先编译 P1.CPP, 然后把它与 P2.OBJ 和 P3.OBJ 连接。

```
C>TCC P1.CPP P2.OBJ P3.OBJ
```

在这个例子中, 假定 P2.OBJ 和 P3.OBJ 存在。建立这些始于.CPP 源文件的方法是用-c 编译选项编译每一个文件。该选项使编译器建立.OBJ 文件, 但不进行连接。

若有不为 Turbo C++ 提供的附加库, 可以用.LIB 扩展名指定它们。

连接程序生成的可执行输出文件通常有用.EXE 扩展名编译的源文件的文件名, 但可以用-e 选项指定别的名字。跟在-e 之后的名即是编译程序用来作为可执行文件的名。在-e 和文件名之间不能有空格。例如下个命令编译文件 TEST.CPP 并建立一个叫做 RUN.EXE 的可执行文件。

```
C>TCC -eRUN TEST.CPP
```

## 附录 F 编译多文件程序

多数实际的 C++ 程序都太大而很难放在一个文件中。特别大的文件编辑起来很困难。而且，在程序中作了一点修改就需要编译整个程序。尽管 Turbo C++ 编译速度很快，但在一定程度——不管编译有多快——编译所花的时间将是难以容忍的。解决的方法是将程序分成几个小部分进行处理，这个过程被称作分散编译和连接。

### F.1 工程和工程选项

在 Turbo C++ 集成环境中，多文件程序称作工程。每个工程与一个工程文件对应，它确定哪些文件是工程的组成部分。Project 菜单选项支持工程文件的管理。所有工程文件必须以 .PRJ 为扩展名。

当选择 Project 选项时，将有下列选择显示：

Open project...

Close project

Add item...

Delete item

Local options...

Include files...

要建立一个工程，首先必须选择 Open project。将提示你输入工程名，其必须有 .PRJ 扩展名。要消除一个工程，用 Close project。一旦建立了一个工程文件，使用 Add item 把组成工程的文件名输入到工程文件中。例如，若工程文件称作 MYPROJ.PRJ，其含有两个文件 TEST1.CPP 和 TEST2.CPP，则需要键入 TEST1.CPP 和 TEST2.CPP。若要删除一项，选择 Delete item。可用 Local options 指定各选项。Include files 选项可让你看到工程中有什么包含文件。

为便于讨论，假设有一个工程文件，它含有文件 TEST1.CPP 和 TEST2.CPP。此外，假定 TEST1.CPP 和 TEST2.CPP 都没有被编译过。有两种编译和连接这些文件的方法。首先，可以选择 run 主菜单选项。当在 Project 选项中指定了一个 .PRJ 文件时，该文件被用于指导 Turbo C++，完成编译程序工作。PRJ 文件的内容被读出并且待编译的每个文件被编译到一个 .OBJ 文件中。然后，这些文件被连接到一块，并且程序被执行。

编译一个工程的第二种方法是使用内部 Make 功能。通过按 F9，或选择 Compile 主菜单选项下的 Make 选项可以使 Turbo C++ 编译和连接在工程文件中指定的所有文件。这与 Run 选项之间的唯一区别是不执行程序。实际上，可以把 Run 选项看作先执行 Make，然后执行 .EXE 文件。

无论何时 Make 一个程序时只有那些需要编译的文件将实际被编译。Turbo C++ 通过检查与每个源文件及其 .OBJ 文件相关的时间和日期来确定。若 .CPP 文件比 .OBJ 文件新，则 Turbo C++ 便知道 .CPP 文件已被修改，它将重编译之，否则它便直接使用 .OBJ 文件。在这种情形下，target.OBJ 文件被认为是依赖 .CPP 文件的。对于 .EXE 文件也有类似的情形。只要 .EXE 文件新于工程中所有 .OBJ 文件，则无须重编译，否则编译必要的文件并

重连接。

除了检查.CPC, .OBJ 和.EXE 文件中的日期外, Turbo C++还检查程序中所用的头文件是否被修改。若被修改, 则使用一个被修改头文件的任何文件都自动被重编译。

无疑, Turbo C++的工程功能是其最重要的部分, 因为它将使多个源文件的管理变得非常容易。

ISBN 7-5027-1352-1 / TP·21

定价: 15.00元