目　录

前　言 ………………………………………………………………………… (I)
第 1 章　几个重要问题 ……………………………………………………… (1)
1.1　数据类型转换 ……………………………………………………… (1)
1.1.1　各类整数之间的转换 ………………………………………… (1)
1.1.2　实数与整数之间的转换 ……………………………………… (2)
1.1.3　指针之间的转换 ……………………………………………… (3)
1.2　指针 ………………………………………………………………… (3)
1.2.1　指针说明 ……………………………………………………… (3)
1.2.2　指针与地址 …………………………………………………… (3)
1.2.3　指针运算 ……………………………………………………… (4)
1.2.4　指针分类 ……………………………………………………… (4)
1.2.4.1　近(near)指针 ……………………………………………… (5)
1.2.4.2　远(far)指针 ………………………………………………… (5)
1.2.4.3　巨(huge)指针 ……………………………………………… (6)
1.2.4.4　基(based)指针 ……………………………………………… (7)
1.2.5　各类指针之间的转换 ………………………………………… (7)
1.3　函数 ………………………………………………………………… (7)
1.3.1　有返回值的函数 ……………………………………………… (7)
1.3.2　无返回值的函数 ……………………………………………… (8)
1.3.3　修改参数的函数 ……………………………………………… (8)
1.3.4　递归函数 ……………………………………………………… (9)
1.3.5　参数个数不定的函数 ………………………………………… (9)
1.3.6　函数指针及其应用 …………………………………………… (10)

第 2 章　编译模式和内存组织 ……………………………………………… (14)
2.1　段与偏移量 ………………………………………………………… (14)
2.2　六种编译模式 ……………………………………………………… (16)
2.2.1　概述 …………………………………………………………… (16)
2.2.2　微模式 ………………………………………………………… (18)
2.2.3　小模式 ………………………………………………………… (19)
2.2.4　中模式 ………………………………………………………… (20)
2.2.5　紧凑模式 ……………………………………………………… (20)
2.2.6　大模式 ………………………………………………………… (20)
2.2.7　巨模式 ………………………………………………………… (20)

2.3 堆栈的组织 …… (20)
2.4 堆的组织 …… (22)
2.5 其它内存操作函数 …… (24)

第3章 鼠标输入 …… (26)
3.1 鼠标驱动程序的基本功能 …… (26)
3.2 与鼠标接口的C函数工具包 …… (27)
3.2.1 14个工具函数 …… (27)
3.2.2 工具包应用举例 …… (36)
3.3 Turbo C Tools的鼠标支持函数 …… (38)
3.3.1 鼠标的初始化 …… (39)
3.3.2 询问鼠标的状态 …… (39)
3.3.3 鼠光标的位置和速度控制 …… (41)
3.3.4 鼠光标的形状和开关控制 …… (42)
3.3.5 对鼠标硬件中断的处理 …… (43)

第4章 文本屏幕输出和文本窗口 …… (45)
4.1 概述 …… (45)
4.2 Turbo C 的文本屏幕处理 …… (46)
4.2.1 文本输出与操作 …… (47)
4.2.1.1 TTY输出规则 …… (47)
4.2.1.2 输出文本 …… (47)
4.2.1.3 对屏幕内容和光标的操作 …… (47)
4.2.1.4 屏幕与内存之间文本的移动 …… (48)
4.2.2 窗口和显示方式控制 …… (48)
4.2.3 属性控制 …… (49)
4.2.4 状态查询 …… (51)
4.3 pop__up 文本窗口工具包 …… (52)
4.3.1 窗口结构和窗口栈 …… (53)
4.3.2 16个工具函数 …… (54)
4.3.3 工具包应用实例 …… (65)
4.3.3.1 用"瓷砖"式窗口制作菜单 …… (65)
4.3.3.2 移动 pop__up 窗口 …… (67)
4.3.3.3 pop__up 错误信息和正常信息 …… (70)
4.4 Turbo C Tools 的文本屏幕处理 …… (77)
4.4.1 Turbo C Tools与Turbo C 的比较 …… (77)
4.4.2 显示设备和显示方式 …… (78)
4.4.3 页控制 …… (80)
4.4.4 清除和滚动 …… (81)

4.4.5 光标控制 …… (82)
4.4.6 显示属性和颜色 …… (83)
4.4.7 屏幕写入和读取 …… (84)
4.4.8 矩形区域写入和读取 …… (85)
4.4.9 各种文本输出函数的速度比较 …… (88)
4.5 Turbo C Tools 的窗口处理 …… (89)
4.5.1 概述 …… (89)
4.5.2 建立和注销窗口 …… (91)
4.5.3 显示和关闭窗口 …… (93)
4.5.4 窗口控制和状态 …… (95)
4.5.5 清除和滚动 …… (97)
4.5.6 光标位置查询和控制 …… (98)
4.5.7 属性控制 …… (98)
4.5.8 窗口输出／输入 …… (98)
4.5.9 虚拟窗口 …… (103)
4.5.10 常驻内存的窗口程序 …… (106)
4.6 Turbo C Tools 的帮助信息窗口 …… (106)
4.6.1 帮助信息源文件 …… (107)
4.6.2 缺省帮助信息窗口 …… (108)
4.6.3 帮助函数 …… (109)

第5章 键盘输入、菜单和编辑 …… (112)
5.1 Turbo C 的键盘输入 …… (112)
5.1.1 概述 …… (112)
5.1.2 控制台级(conio)键盘输入处理 …… (113)
5.1.3 标准文件级键盘输入处理 …… (116)
5.1.4 普通文件级键盘输入处理 …… (116)
5.1.5 BIOS 级键盘输入处理 …… (117)
5.1.5.1 中断 0x9 …… (117)
5.1.5.2 中断 0x16 …… (118)
5.1.5.3 bioskey 函数 …… (118)
5.2 <Ctrl Break> 和 <Ctrl C> …… (119)
5.3 Turbo C Tools 的键盘处理 …… (120)
5.3.1 键盘输入 …… (120)
5.3.2 缓冲区处理 …… (121)
5.3.3 状态控制键 …… (124)
5.3.4 使用加强键盘 …… (125)
5.3.5 使用键控制函数 …… (125)
5.3.6 取得键码 …… (126)

5.4 Turbo C Tools 的菜单处理 …… (126)
5.4.1 概述 …… (126)
5.4.2 建立、显示和注销菜单 …… (129)
5.4.3 定义菜单项和按键鼠标事件 …… (130)
5.4.4 读取用户的选择 …… (132)
5.4.5 菜单应用举例 …… (134)
5.4.5.1 [例 1]: 一个简单演示程序 …… (134)
5.4.5.2 [例 2]: 用菜单实现电子表格 …… (135)
5.5 Turbo C Tools 的域编辑 …… (140)
5.5.1 概述 …… (140)
5.5.2 域编辑 …… (144)
5.5.3 编辑键定义 …… (146)

第 6 章 基本文件处理 …… (149)
6.1 目录/文件系统概述 …… (150)
6.1.1 文件存取级别 …… (150)
6.1.2 文件属性 …… (151)
6.2 系统级输入/输出 …… (152)
6.2.1 文件柄 …… (152)
6.2.2 文件柄存取字节 …… (153)
6.2.3 文件柄属性字节 …… (154)
6.2.4 文件出错处理 …… (155)
6.2.5 建立文件 …… (155)
6.2.6 打开文件 …… (157)
6.2.7 读取和设置文件的特性 …… (158)
6.2.8 读、写和关闭文件 …… (159)
6.3 标准级(流式)输入输出 …… (160)
6.3.1 FILE 数据结构 …… (160)
6.3.2 建立/打开/关闭/删除文件 …… (162)
6.3.3 取文件状态和出错处理 …… (164)
6.3.4 控制文件缓冲区 …… (164)
6.3.5 移动文件指针 …… (166)
6.3.6 字节级的读/写 …… (167)
6.3.7 字符串级的读/写 …… (168)
6.3.8 记录级的读/写 …… (168)
6.4 基本文件处理工具包 …… (169)
6.4.1 10 个工具函数 …… (169)
6.4.2 工具包应用 …… (175)
6.5 驱动器和目录操作 …… (175)

6.5.1 驱动器和驱动器信息 …… (175)
6.5.2 目录操作 …… (177)
6.5.3 文件名操作 …… (178)
6.5.4 目录搜索 …… (178)

第7章 字符串处理 …… (182)
7.1 字符 …… (182)
7.1.1 字符数据和常数 …… (182)
7.1.2 字符输入／输出 …… (183)
7.1.3 字符的分类和转换 …… (184)
7.1.4 宏和宏的副作用 …… (184)
7.2 字符串 …… (186)
7.3 字符串的分析 …… (187)
7.4 字符串的综合 …… (191)
7.5 字符串的操作 …… (193)
7.6 文本字符串 …… (196)
7.7 数字字符串 …… (197)
7.8 国家和货币字符串 …… (202)
7.9 日期和时间字符串 …… (204)
7.10 文件名字符串 …… (208)
7.11 命令行字符串 …… (209)
7.12 环境字符串 …… (209)
7.13 错误级字符串 …… (210)
7.14 字符串处理工具包 …… (211)
7.14.1 13个工具函数 …… (211)
7.14.2 工具包应用举例:PASCAL 程序翻译为 C 程序 …… (215)

第8章 动态通用串处理 …… (223)
8.1 动态字符串 …… (223)
8.2 动态通用串 …… (225)
8.3 动态通用串工具包 …… (227)
8.3.1 8个工具函数 …… (227)
8.3.2 工具包应用举例:多边形表示法 …… (232)
8.4 通用串的排序与查找 …… (234)
8.5 动态通用串与链表的比较 …… (242)

第9章 高级文件处理 …… (244)
9.1 变长记录(VLR)文件 …… (244)
9.1.1 从文件中查找一个记录 …… (244)

9.1.2 插入和删除记录 …… (245)
9.1.3 碎片化问题 …… (245)
9.1.4 VLR 文件格式 …… (246)
9.1.5 VLR 记录格式和数据块格式 …… (246)
9.2 VLR 工具包 …… (247)
9.2.1 7个工具函数 …… (247)
9.2.2 索引处理 …… (255)
9.2.3 工具包应用举例:制作和显示幻灯片 …… (256)

第10章 内存和程序管理 …… (264)
10.1 PSP和环境 …… (264)
10.1.1 PSP …… (264)
10.1.2 环境 …… (265)
10.2 内存管理 …… (267)
10.2.1 内存块及其控制 …… (267)
10.2.2 内存分布映像程序 memrymap …… (268)
10.2.3 内存管理函数 …… (270)
10.3 多个程序的执行及通信 …… (270)
10.3.1 程序间的通信 …… (270)
10.3.2 spawn: 调用子进程 …… (271)
10.3.3 exec: 转到子进程 …… (275)
10.3.4 system:执行 DOS 命令 …… (275)
10.3.5 signal 和 raise:事件处理 …… (276)
10.4 标准输入/输出重定向 …… (278)
10.4.1 [例 1]:freopen.dem …… (279)
10.4.2 [例 2]:dup.dem …… (280)
10.4.3 [例 3]:利用 system …… (282)
10.5 程序的终止 …… (282)

第11章 MSC 6.0的基指针技术 …… (287)
11.1 6种基(Based)指针 …… (287)
11.1.1 变量值基指针 …… (287)
11.1.2 变量地址基指针 …… (289)
11.1.3 不定基指针 …… (289)
11.1.4 段名基指针 …… (290)
11.1.5 指针基指针 …… (291)
11.1.6 自参照基指针 …… (291)
11.2 基指针应用于链表管理的工具包 …… (292)
11.2.1 基指针应用于链表管理 …… (292)

11.2.2 基指针分配函数 …… (293)
11.2.3 16个工具函数 …… (294)
11.2.4 工具包应用三例 …… (304)
11.2.4.1 [例 1] …… (304)
11.2.4.2 [例 2]和[例 3] …… (305)

第12章 与BIOS和DOS的接口 …… (309)
12.1 中断概述 …… (309)
12.2 与BIOS的接口 …… (312)
12.2.1 与BIOS接口的函数 …… (312)
12.2.2 BIOS提供的部分服务 …… (314)
12.3 与DOS的接口 …… (317)
12.3.1 与DOS接口的函数 …… (317)
12.3.2 DOS提供的部分服务 …… (318)
12.4 标准输入/输出服务 …… (319)
12.4.1 BIOS提供的显示服务 …… (319)
12.4.2 BIOS提供的键盘服务 …… (320)
12.4.3 DOS提供的标准输入/输出服务 …… (320)
12.5 文件输入/输出服务 …… (321)
12.6 内存管理与程序执行服务 …… (322)
12.7 打印服务 …… (323)
12.8 时钟/日历服务 …… (325)
12.8.1 PC机上的时钟系统 …… (325)
12.8.2 PC/AT机上的时钟系统 …… (326)
12.8.3 DOS的时间/日历服务 …… (326)
12.8.4 延迟函数 …… (326)
12.8.5 声音函数 …… (327)
12.9 串行通信服务 …… (327)
12.10 错误处理服务 …… (328)
12.10.1 DOS怎样报告错误 …… (328)
12.10.2 Turbo C 库函数的错误报告特性 …… (329)
12.10.3 致命错 …… (331)
12.10.4 <Ctrl Break>和<Ctrl C> …… (331)

第13章 中断服务程序 …… (333)
13.1 一般概念 …… (333)
13.2 用Turbo C Tools写中断服务程序 …… (335)
13.2.1 工作原理 …… (335)
13.2.2 安装和驻留 …… (339)

13.2.3 过滤 …… (342)
13.2.4 探测和撤消 …… (342)
13.2.5 其它 …… (343)
13.3 中断服务程序实例 …… (344)
13.3.1 [例 1]:周期性地发声 …… (344)
13.3.2 [例 2]:检测 A 和 J 键的同时按下 …… (345)
13.3.3 [例 3]:发送格式化的输出 …… (349)
13.4 用 Turbo C Tools 写插入服务程序 …… (353)
13.4.1 DOS 的重入问题 …… (353)
13.4.2 插入服务技术 …… (354)
13.4.3 插入服务函数 …… (358)
13.4.4 插入服务程序举例 …… (362)
13.5 用 Turbo C 写中断服务程序 …… (366)

第 14 章 图形处理 …… (369)
14.1 Turbo C 图形处理函数 …… (369)
14.1.1 概述 …… (369)
14.1.2 图形系统控制函数 …… (369)
14.1.3 画图和填充函数 …… (373)
14.1.4 屏幕和视口管理函数 …… (375)
14.1.5 图形方式下的文本输出函数 …… (376)
14.1.6 颜色控制函数 …… (378)
14.1.7 错误处理函数 …… (380)
14.1.8 状态查询函数 …… (380)
14.2 Pop up 图形窗口工具包 …… (381)
14.2.1 图形窗口与文本窗口 …… (381)
14.2.2 6 个工具函数 …… (382)
14.2.3 工具包应用举例:移动窗口 …… (389)
14.3 图形方式下输出文本的若干问题 …… (392)
14.3.1 格式输出 …… (392)
14.3.2 重写 …… (393)
14.3.3 加亮 …… (395)
14.3.4 滚动 …… (396)
14.4 用 XOR 方式画旋转橡皮筋 …… (396)

第 15 章 混合模式和混合语言编程 …… (400)
15.1 混合模式编程 …… (400)
15.1.1 概述 …… (400)
15.1.2 说明一个函数为 near 或 far …… (400)

15.1.3 说明一个指针为 near、far 或 huge …… (401)
15.1.4 使用库文件 …… (402)
15.1.5 不同编译模式所生成模块的连接 …… (403)
15.2 C 和汇编语言混合编程 …… (404)
15.2.1 段的组合 …… (404)
15.2.1.1 汇编语言的段和组 …… (404)
15.2.1.2 Trubo C 的段和组 …… (405)
15.2.1.3 段和组的连接 …… (407)
15.2.2 变量和函数名的相互引用 …… (408)
15.2.3 参数传递规则 …… (409)
15.2.4 返回值传递规则 …… (410)
15.2.5 寄存器规则 …… (411)
15.2.6 混合编程示例 …… (411)
15.2.6.1 C 调用汇编 …… (411)
15.2.6.2 汇编调用 C …… (412)
15.3 行内汇编 …… (414)

附录 1 操作符表 …… (417)
附录 2 键盘码表 …… (419)
附录 3 Turbo C 2.0 函数简表 …… (422)
附录 4 Turbo C Tools 6.0 函数简表 …… (446)

第1章　几个重要问题

1.1　数据类型转换

1.1.1　各类整数之间的转换

C 语言中的数分 8 位、16 位和 32 位三种。属于 8 位数的有：带符号字符 char，无符号字符 unsigned char。属于 16 位数的有：带符号整数 int，无符号整数 unsigned int(或简写为 unsigned)，近指针。属于 32 位数的有：带符号长整数 long，无符号长整数 unsigned long，远指针。

IBM PC 是 16 位机，基本运算是 16 位的运算，所以，当 8 位数和 16 位数进行比较或其它运算时，都是首先把 8 位数转换成 16 位数。为了便于按 2 的补码法则进行运算，有符号 8 位数在转换为 16 位时是在左边添加 8 个符号位，无符号 8 位数则是在左边添加 8 个 0。当由 16 位转换成 8 位时，无论什么情况一律只是简单地裁取低 8 位，抛掉高 8 位。没有 char 或 unsigned char 常数。字符常数，像'C'，是转换为 int 以后存储的。当字符转换为其它 16 位数（如近指针）时，是首先把字符转换为 int，然后再进行转换。

16 位数与 32 位数之间的转换也遵守同样的规则。

注意，Turbo C 中的输入／输出函数对其参数中的 int 和 unsigned int 不加区分。例如，在 printf 函数中如果格式说明是%d，则对这两种类型的参数一律按 2 的补码(即按有符号数)进行解释，然后以十进制形式输出。如果格式说明是%u、%o、%x、%X，则对这两种类型的参数一律按二进制(即按无符号数)进行解释，然后以相应形式输出。在 scanf 函数中，仅当输入的字符串中含有负号时，才按 2 的补码对输入数进行解释。

还应注意，对于常数，如果不加 L，则 Turbo C 一般按 int 型处理。例如，语句 printf("%08lx"，-1L)，则会输出 ffffffff。如果省略 l，则输出常数的低字，即 ffff。如果省略 L，则仍会去找 1 个双字，这个双字的低字就是 int 常数-1，高字内容是不确定的，输出效果将是在 4 个乱七八糟的字符之后再跟 ffff。

在 Turbo C 的头文件 value.h 中，相应于 3 个带符号数的最大值，定义了 3 个符号常数：

```
#define  MAXSHORT   0X7FFF
#define  MAXINT     0X7FFF
#define  MAXLONG    0X7FFFFFFFL
```

在 Turbo C Tools 中，包括 3 对宏，分别把 8 位拆成高 4 位和低 4 位，把 16 位拆成高 8 位和低 8 位，把 32 位拆成高 16 位和低 16 位。

```
uthinyb(char value)     utlonyb(char value)
uthibyte(int value)     utlobyte(int value)
uthiword(long value)    utloword(long value)
```

在 Turbo C Tools 中，也包括相反的 3 个宏，它们把两个 4 位组成一个 8 位，把两个 8 位组成一个 16 位，把两个 16 位组成一个 32 位。

```
utnybbyt (HiNyb, LoNyb)
utwdlong (HiWord, Loword)
utbyword (HiByte, LoByte)
```

1.1.2 实数与整数之间的转换

Turbo C 中提供了两种实数：float 和 double。float 由 32 位组成，由高到低依次是：1 个尾数符号位，8 个偏码表示的指数位(偏值＝127)，23 个尾数位。double 由 64 位组成，由高到低依次是：1 个尾数符号位，11 个偏码表示的指数位(偏值＝1023)，52 个尾数位。通过下列公式，可以由尾数和指数计算出所代表的实数值：

$X = \pm 1.\text{尾数} * 2^{(\text{指数}-\text{偏值})}$

下列几种情况下，此公式不成立：

指数＝000...0 且尾数 ＝00...0，则 X ＝± 0

指数＝000...0 且尾数!＝00...0， 则 X ＝± 0.尾数 * $2^{(1-\text{偏值})}$

指数＝11....1 且尾数 ＝00...0，则 X ＝±∞

指数＝11....1 且尾数!＝00...0， 则 X 是一个无效数

在 Turbo C 的头文件 value.h 中，相应于实数所能达到的最大最小值，定义了如下几个符号常数：

```
#define  MAXFLOAT     3.37E+38
#define  MINFLOAT     8.43E-37
#define  MAXDOUBLE    1.797693E+308
#define  MINDOUBLE    2.225074E-308
```

实常数是按 double 格式存放的，如果想按 float 格式存放，则必须加后缀 F，如：1.23E+4F。

当把实数强制转换为整数时，采取的是“向零靠拢的算法”，如：

float值:	65432.6	−65432.6
转换为long:	65432	−65432
转换为unsigued:	65432	104

如果不希望“向零靠拢”，而希望“四舍五入”，则必须由程序员自己处理。一种办法是先加上(符号位／2)，然后再进行类型转换。应该注意的是：如果被转换的实数值超过了目标类型的表达范围，则会产生错误。例如上面的 float 值−65432.6 转换为 unsigned int 值时，由于超过了目标范围，所产生的 104 就是错误的。在 65432.6 转换为 unsigned int 的 65432 以后，可以用 printf 的%u 格式输出，如果用%d 格式输出，则会得到错误的结果。

1.1.3 指针之间的转换

关于各类指针之间的转换请见下一节以及第 11 章。

1.2 指　　针

1.2.1 指针说明

指针是包含另一变量的地址变量。它的一般说明形式，如 int　* fd，其 fd 是一个指向整型变量的指针。比较复杂的指针说明，如 *(* pfpi)( )，可按如下几个原则来理解：以标识符为中心，一对方括号一般表示数组，一对圆括号一般表示函数或强调某一优先顺序，方括号对和圆括号对为同一优先级，方括号和圆括号比 * 号优先级高。以下几例解释了这些原则的应用。

int　* fip( )，因圆括号优先级高，故 fip 先与圆括号结合，说明 fip 是一个函数，这个函数返回一个指向整数的指针。

int　(* pfi)( )，因两对圆括号为同一优先级，故从左到右，pfi 是一个指针，这个指针指向一个函数，这个函数返回一个整数。

int　* par[ ]，因方括号比 * 号优先级高，故 par 是一个数组，这个数组的每一个元素是指向整数的指针。

int　(* ptr)[ ]，因方括号与圆括号为同一优先级，故 ptr 是一个指针，这个指针指向一个数组，这个数组的每一个元素是一个整数。

int　*(* pfpi) ( )，pfpi 是一个指针，这个指针指向一个函数，这个函数返回一个指向整数的指针。

1.2.2 指针与地址

指针在使用之前必须初始化，给指针赋地址的方法一般有如下几种：

第一种很容易理解，通过取地址操作符取变量(包括结构变量)的地址，如：

```
char__c="a",        * ptr__char;
ptr__char=&c;
```

第二种是数组，因为不带方括号的数组名等效于数组中第一个元素的地址，即数组名也可看作是一个指针，所以有两种办法。如：

```
     char myname[31],     * ptr;
     ptr=myname;
或   ptr=&myname[0];
```

第三种是动态分配的一块内存，这时往往带有类型强制转换，但应注意当内存不够时，可能返回一个空(NULL)指针。如：

```
struct complex {double real, imag; };
struct complex *ptr;
ptr=(struct complex *)malloc(sizeof(struct complex));
```

第四种是函数，与数组名一样，函数名也可以当作一个地址，于是可以把函数名赋给一个指向函数的指针。函数名后面跟一个带圆括号的参数表意味着计算函数的值，但仅有一个函数名则意味着指向函数的指针。如:

```
double (*fx)( );
double quad_poly(double);
fx=quad_poly;
```

1.2.3 指针运算

常见的指针运算有：指针加或减一个数，指针增量，指针减量，指针比较等等。假设P是某数组第1个元素的指针，则P+N就是这个数组中第n个元素的地址，每个元素占多少存储单元则由指针所指数组的类型来确定。但有两点要注意:

第一，上面这段话是C语言对指针运算的普遍规律，但具体到各种C编译则有所不同，尤其是在80X86类型的机器上。Turbo C 和 Microsoft C 6.0以前的版本把指针分为near、far、huge、Microsoft C 6.0 又增加了based。在这几种指针中，只有huge严格遵守上面的指针运算规则，详见下一节。

第二，当指针应用于数组尤其是多维数组时，有时容易弄错，表1-1说明了数组法与指针法的区别。

表1-1　数组和指针表示法

	1 维	2 维	3 维
数组说明	int x[ ]	int y[ ][ ]	int z[ ][ ][ ]
指针说明	int *xptr	int *yptr[ ]	int *zptr[ ][ ]
用数组法表示某一元素的地址	&x[i]	&y[i][j]	&z[i][j][k]
用指针法表示某一元素的地址	ptr+i	*(yptr+i)+j	*(*(zptr+i)+j)+k
用数组法存取某一元素	x[i]	y[i][j]	z[i][j][k]
用指针法存取某一元素	*(ptr+i)	*(*(yptr+i)+j)	*(*(*(zptr+i)+j)+k)

1.2.4 指针分类

在C语言教科书上，指针就是指针，不存在分类的问题。我们经常说"指向整数的指针"、"指向结构的指针"、"指向函数的指针"等等，只是说指针指向不同的目标，而不是说指针本身有什么区别。但是，在以80X86为基础的微机上实现C语言时，由于80X86的物理地址空间不是线性连续的而是分段的，为了提高效率，就必须对指针加以分类。各

类指针的运算法则也不一样。Turbo C 2.0 及以前的版本，Microsoft C 6.0 以前的版本，指针都是分为三类，近(near)，远(far)，巨(huge)。Microsoft C 6.0 版本中，出现了一种新的指针类型，这就是基(based)指针。基指针综合实现了近和远指针的优点，它像近指针那么小那么快，又像远指针那样可以寻址缺省数据段以外的目标。基指针这个名字就反映了这类指针的实现方法：它是以程序员指定的某一地址为段基址。如果在 C 源程序中使用了基指针，编译程序一般先把所指定的段基址装在 ES 寄存器内。在缺省的数据段内，程序员一般不会使用基指针。但若同时要使用多个数据段，基指针则有其明显的优点。

1.2.4.1 近 (near) 指针

近指针是用于不超过 64K 字节的单个数据段或码段。对于数据指针，在微、小和中编译模式下产生的数据指针是近指针，因为此时只有一个不超过 64K 字节的数据段。对于码(即函数指针)指针，在微、小和紧凑编译模式下产生的码指针是近指针，因为此时只有一个不超过 64K 字节的码段。本章将只讨论数据指针。

近指针是 16 位指针，它只含有地址的偏移量部分。为了形成 32 位的完整地址，编译程序一般是把近指针与程序的数据段的段地址组合起来。因为在大部分情况下程序的数据段的段地址是装在 DS 寄存器内，因此一般没有必要重新装载这个寄存器。此外，当用汇编语言和 C 语言混合编程时，汇编语言总是假设 DS 含有数据目标的地址。

虽然近指针占用空间最小，执行速度最快，但它有一个严格的限制，即只能存取 64K 字节以内的数据，且只能存取程序的数据段内的数据。如果在小模式下编译一个程序，而这个程序企图增量一个近指针使之超过第 65536 个字节，则这个近指针就会复位到 0。下面就是这样一个例子：

```
char _near *p = (char _near *)0xffff;
p++;
```

由于近指针的这个严重限制，所以在比较大或比较复杂的程序中，都无法使用。

1.2.4.2 远 (far) 指针

远指针不是让编译程序把程序数据段地址作为指针的段地址部分，而是把指针的段地址与指针的偏移量直接存放在指针内。因此，远指针是由 4 个字节构成。它可以指向内存中的任一目标，可以用于任一编译模式，尽管仅在紧凑、大和巨模式下远指针才是缺省的数据指针。因为远指针的段地址在指针内，熟悉 80X86 汇编语言的人都知道，这意味着每次使用远指针时都需要重新装载段寄存器，这显然会降低速度。

应该注意：尽管远指针可以寻址内存中的任一单元，但它所寻址的目标也不能超过 64K 字节。这是因为，远指针在进行增量或减量之类的算术运算时，也只是偏移量部分参与运算，而段地址保持不变。因此，当远指针增量或减量到超过 64K 字节段边界时就出错。例如：

```
char far *fp = (char far *)0xb800ffff;
```

```
fp++;
```

在指针加 1 以后，fp 将指向 B800: 0000，而不是所希望的 C800: 0000。

此外，在进行指针比较时，far 指针还会引起另外一些问题。far 指针是由偏移量和段地址这样一对 16 位数来表示的，对于某一实际内存地址，far 指针不是唯一的，例如，far 指针 1234: 0005、1230: 0045、1200: 0345、1000: 2345、0900: 9345 等都是代表实际地址 12345，这样会引起许多麻烦。

第一，为了便于与"空"(NULL)指针(0000: 0000)进行比较，当关系操作符"= ="和"! ="用于对 far 指针进行比较时，比较的是全部 32 位。否则，如果只比较 16 位偏移量，那么任何偏移量为 0 的指针都将是"空"(NULL)指针，这显然不符合一般使用要求。但在进行这 32 位比较时，不是按 20 位实际地址来比较，而是把段地址和偏移量当作一个 32 位的无符号长整数来比较。对于上面这个例子，假设这些指针分别叫作 a、b、c、d、e，尽管这 5 个 far 指针指向的都是同一内存单元，但下列表达式运算的结果却都为"假"，从而得出错误的结论：

```
if (a= =b) ......
if (b= =c) ......
if (c= =d) ......
if (d= =e) ......
if (a= =c) ......
if (a= =d) ......
```

第二，当用">"、">="、"<"和"<="关系操作符对指针进行比较操作时，比较的仅仅是偏移量部分，即按无符号的 16 位整数进行比较。因此，对于上面这个例子，下列表达式运算的结果将都为"真"，也得出错误的结论：

```
if(e>d) ......
if(d>c) ......
if(c>b) ......
if(b>a) ......
if(e>a) ......
```

1.2.4.3 巨 (huge) 指针

只有巨指针才是一般 C 语言教科书上所说的指针，它像远指针一样，也占 4 个字节。与远指针的显著差别是：当增量或减量超过 64K 字节段边界时，巨指针会自动修正段基址的值。因此，巨指针不但可以寻址内存中的任一区域，而且所寻址的数据目标可以超过 64K 字节。例如：

```
char  huge  *hp  =  (char  huge  *)0xb800ffff;
hp++;
```

在指针加 1 后，hp 将指向 C800: 0000。但是，巨指针总是比较慢的，因为编译必须生成一小段程序对指针进行 32 位而不是 16 位的加减运算。

此外，由于 huge 指针是规则化指针，每一个实际内存地址只对应一个 huge 指针，

所以在指针比较时不会产生错误。

1.2.4.4 基(based)指针

前面已经说过，巨指针综合了近指针和远指针的优点。像近指针一样，基指针只占两个字节，这两个字节是地址的偏移量。像远指针一样，基指针可以寻址内存中的任一区域。近指针的段地址隐含地取自程序的数据段，远指针的段地址取自指针本身，基指针的段地址取法以及基指针的许多技术和应用问题，请见第 11 章。

1.2.5 各类指针之间的转换

far 指针可以强制转换为 near 指针，做法很简单，抛掉段地址只保留偏移量。near 指针也可以转换为 far 指针，Turbo C 的做法是从相应的段寄存器中取得段地址。

far 指针有时也需要转换为 huge 指针，以便对指针进行比较或做其它操作。一种方法是通过下面这样一个规则化函数：

```
void normalize(void far * * p) {
    * p=(void far * )(((long) * p&0xffff000f)+(((long) * p&0x0000fff0) < < 12));
}
```

另一种办法就是通常的强制类型转换，但强制类型转换不能自动使转换后的结果规则化。解决的办法是使转换后的 huge 指针再做一次加法。例如，设转换后的 huge 指针是 Hp，做一次 Hp+=0 就使 Hp 规则化了。

1.3 函数

从编程技术的角度来看，函数可以分为 5 种：有返回值的函数，无返回值的函数，修改参数的函数，递归函数，参数个数不定的函数。

1.3.1 有返回值的函数

这类函数是最常见的函数，如返回一个字符，返回一个整数，返回一个指针等，在函数的说明中就说明了要返回什么类型的数据。因返回整数是最常见的，所以在这种情况下函数说明前的 int 可以省略。这里要强调一点的是：函数除可以返回一些 C 语言标准类型的数外，还可以返回用户自定义的数据类型，如结构、联合、枚举等。例如，下面这个对两个字符串相加的程序就是返回一个结构。

```
struct string {
    char str[256];
    int  strlen;
};
struct string concat(struct string str1, struct string str2)
{
    struct string result;
```

```
    int i, j;
    result.str[0] = '\0';
    result.strlen = 0;
    if (str1.strlen > 0) {
        for ( i = 0; i < str1.strlen; i++)
            result.str[i] = str1.str[i];
        result.strlen = str1.strlen;
    }
    if (str2.strlen > 0) {
        j = str1.strlen;
        for ( i = 0; i < str2.strlen; i++)
            result.str[i+j] = str2.str[i];
        result.strlen += str1.strlen;
    }
    return result;
}
```

1.3.2 无返回值的函数

无返回值的函数与 PASCAL 等其它结构化语言中的过程很相似，它们既不返回结果，又不修改参数，而只是执行某一特定的任务。例如，下面的清屏函数就是这样一个函数。

```
void clrscr(void)
{
    printf("\x1b[2J");
}
```

既然不返回值，则调用的办法也不一样，不是把函数名放在某一表达式内调用，而是把函数名连同其调用参数单独作为一个语句。

1.3.3 修改参数的函数

由于 C 语言是按传值方式把参数传递给函数的，因此，被调用的函数不能直接改变调用函数中的变量。但有时确实需要修改调用函数的参数，尤其在返回值多于一个的函数中必须再借用参数来返回结果。 在这些情况下，必须利用指针来作为函数的参数，典型的例子是交换两个变量的值的函数。如下所示：

```
void swap int(int *i, int *j)
{
    int temp;
    temp= *i;
    *i= *j;
    *j=temp;
}
```

1.3.4 递归函数

C 语言是支持递归调用的。显然，当一个问题蕴含递归关系且结构比较复杂时，采用递归调用技巧将使程序变得简洁，并增加程序的可读性。但递归调用技巧的使用是在牺牲存储空间的基础上得到的，因为它必须在某处维护一个要处理的值的栈。同时，递归也不能提高执行速度，只是其代码比较紧凑易读。对于像树和链表这样的递归定义的数据结构，递归函数尤为适用。下面是用递归计算阶乘的例子。

```
double    factorial(int n)
{
    if (n>1) return factorial(n-1) * (double)n;
    else return 1.0L;
}
```

1.3.5 参数个数不定的函数

C 语言中的某些函数，如 vfprintf 和 vprintf，允许在一些固定参数之后再带一些不定数目的可变参数。不但如此，C 语言还允许用户自定义的函数也这样做。为了便于用户编程，Turbo C 中提供了以"va"开头的 4 个定义：va_list 数据类型，va_stat，va_arg 和 va_end 3 个宏(函数)。这些定义都在头文件 stdarg.h 中。借助于这些宏可以一步一步地通过整个参数表，尽管被调用函数事先不知道究竟有多少个参数，也不知道这些参数的类型。

为了编写具有不定数目的可变参数函数，应遵守如下几步：

第 1，在 C 源码中包含 stdarg.h 文件。

第 2，如果函数的返回值不是 int 型，则在调用函数中应做如下形式的函数说明：

<类型> <函数名> (<固定参数表>，...);

这个调用形式表明，参数表中至少必须有一个参数是固定的。

第 3，函数应按如下形式定义：

<类型> <函数名> (<固定参数表>，...);

第 4，定义一个表指针，其类型应是 va_list，以表明它指向可变参数表。如下所示：

va_list <可变参数表指针>

第 5，调用 va_start，初始化表指针：

va_start (<可变参数表指针>，<最后一个固定参数的名字>)

这样初始化后，表指针就指向了调用函数传来的可变参数中的第 1 个参数。

第 6，调用 va_arg，取可变参数：

<变量>=va_arg(<可变参数表指针>，<参数的数据类型>)

第 1 次调用 va_arg 时，它返回可变参数表中的第 1 个参数。随后每一次调用，它返回表中的下一个参数。每次调用之后自动修正表指针的值，使它指向随后的一个参数。为了正确地停止读可变参数表，应该在调用函数可变参数表的最后放一个表结束符（例如-1 或 0），在被调用函数中再去检查这个表结束符。While 循环很适合做这件事情，如下面的例子所示。

第 7，调用 va_end，返回到调用函数：

va_end（<可变参数表指针>）

它帮助被调用函数正常返回到调用函数。应在 va_arg 读完所有参数之后，才调用 va_end 返回，否则可能会引起意想不到的结果。

下面这个例子利用一个具有可变参数表的函数，从一个数字表中挑选值最大的那个数。

```
#include <stdio.h>
#include <stdarg.h>
#define  EOL -1
main ( )
{
     int big;
     void vmax(int *, char *, ...);
     vmax(&big, "The largest of 55, 67, 41 and 28 is",
                              55, 67, 41, 28, EOL);
     printf("%d\n", big);
}
void vmax(int *large, char *message, ...)
{
     int num;
     va_list num_ptr;
     va_start(num_ptr, message);
     printf("%s", message);
     *large=-1;
     while((num=va_arg(num_ptr, int))!=EOL)
          if(num>*(large))     *(large)=num;
     va_end(num_ptr);
}
```

1.3.6 函数指针及其应用

函数名后面跟一对圆括号（兴许括号内还有参数），将导致去计算这个函数。仅仅一个函数名则意味着是一个指针，是指向这个函数的指针。函数指针有两个特殊用途，不太熟练的程序员可能很少使用函数指针，但在某些场合下若借助于函数指针，则会使程序显得非常精练。

第 1 种用途是把函数名赋给一个指针，然后用这个指针去间接引用函数。请看下面这个例子：

```
#include <stdio.h>
main( )
{
     double x;
     const double delta = 1.0;
     const double first = 0.0;
     const double last=10.0;
     double (*fx)( );
```

```
    double quad_poly(double);
    fx = quad_poly;
    x = first;
    while (x <= last) {
        printf("f(%lf) = %lf\n", x, fx(x));
        x += delta;
    }
}
double quad_poly(double x)
{
    double a=1.0, b=-3.0, c=5.0;
    return ((x * a) * x + b) * x + c;
}
```

在这个例子里,语句 double(* fx)( )说明 fx 是一个函数指针,该函数返回一个 double 型数。然后把函数名 quad_poly 赋给这个指针。通过 fx(x)引用这个函数，取得函数的返回值。

有时候，程序中要用到多个函数，这些函数有相同的参数要求和相同的返回值类型。但这些函数不是同时都要用到，而是根据不同的情况每次仅调用其中的一个。比较笨拙的办法就是用 switch 语句去实现，虽然也还清楚，但程序显得冗长。用函数指针则显得精练多了，如下面的例子所示:

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
#define  MAX 3
main( )
{
    double x;
    const double delta = 1.0;
    const double first = 0.0;
    const double last = 10.0;
    double ( * fx[MAX])( );
    int i;
    char ch;
    double quad_poly(double);
    fx[0] = quad_poly;
    fx[1] = sqrt;
    fx[2] = log;
    for (i = 0; i < MAX; i++) {
        x = first;
        while (x <= last) {
            printf("f(%lf) = %lf\n", x, fx[i](x));
            x += delta;
        }
        printf("press any key to continue ");
        ch = getche( );
        clrscr( );
```

```
    }
}
double quad_poly(double x)
{
    double a = 1.0, b = -3.0, c =5.0;
    return ((x * a) * x + b) * x + c;
}
```

在这个例子里，有三个数学函数：quad_poly，sqrt 和 log，它们都只要一个 double 型参数，都返回一个 double 型数。程序中通过语句 double (* fx[MAX])( )说明 fx 是函数指针数组，每一个函数都返回 double 型数。然后用三个函数的名字初始化这个函数指针数组，通过 fx[i](x)在每次循环中各引用一个不同的函数，取得函数的返回值。

第 2 种用途是通过函数指针，把一个函数作为参数传递给另一个函数，请看下面这个例子：

```
#include <stdio.h>
main( )
{
    double x;
    const double delta = 0.01;
    const double first = 0.0;
    const double last = 10.0;
    double ( * fx) ( );
    double quad_poly(double);
    double find_largest(double, double, double, double ( * fx) ( ));
    fx =quad_poly;
    printf("The largest value in the range %lf -> %lf ", first, last);
    printf("is %lf\n", find_largest(first, last, delta, fx));
}
double quad_poly(double x)
{
    double a = 1.0, b = -3.0, c =5.0;
    return ((x * a) * x + b) * x + c;
}
double find_largest(double a, double b, double step, double ( * fx) ( ))
{
    double x = a, big = ( * fx) (a);
    while (x <= b) {
        if (big < ( * fx)(x))
            big = ( * fx)(x);
        x += step;
    }
    return big;
}
```

在这个例子里，函数 quad_poly 计算($a * x^3 + b * x + c$)的值，参数 x 和返回值都是 double 型数。函数 find_largest 根据某一计算法则求出在某一范围内(某一步长)的最大值，不但范围和步长是由调用参数指定，计算法则也是由调用参数指定，所有的参数和返

回值都是 double 型数。主函数 main 调用 find_largest 求最大值，通过函数指针把函数 quad_poly 作为参数传给 find_largest。

第 2 章　编译模式和内存组织

2.1　段与偏移量

8086 和 80286 的寄存器都是 8 位或 16 位的，而 8086 以及工作于实地址方式下的 80286／80386 的地址空间却是 20 位的。这样在寻找下一条将执行的指令时以及当用寄存器间接寻址内存中某一数据时，16 位的指针寄存器和地址寄存器却不足以存下 20 位地址。为了解决这个问题，便把 20 位地址分为两部分，分别称为段地址和偏移量。段地址可以是任一 16 字节边界处，即段地址的末 4 位一定是 0，故不需要保存，只把高 16 位存入段寄存器中。偏移量也是 16 位的，一方面便于寄存器间接寻址，但另一方面也限制了每段不能超过 64K 字节。考虑到程序和数据的寻址一般都是连续的，故这种设计还是合理的。

在 80X86 微机中，总是有 4 个 16 位段寄存器: CS，DS，SS，ES。在 Turbo C 编译产生的目标码中，一般只用到了其中的 3 个寄存器，CS 用来存放码的段地址，DS 用来存放全局变量和静态变量所在段的段地址，SS 用来存放局部变量，参数(以及其它属于某一个函数的信息)所在段的段地址。在 Microsoft C 6.0 中，ES 用来存放基指针的段地址。如果需要，在 Turbo C 中，程序员可以通过伪变量_DS、_CS、_SS、_ES 取得这些段寄存器的值。

如果程序的码、数据和堆栈分别都不超过连续 64K 字节，则在程序的整个执行过程中，CS、DS 和 SS 寄存器的值可以不变，仅仅通过对单字偏移量的操作就可以寻址到所有的码和变量，这样速度比较快。如果码或数据不能在连续 64K 字节内放下，则必须用双字的段: 偏移量来寻址，但速度也就变慢。在所有程序中，堆栈的操作都较频繁，所以都禁止使用双字的寻址方法，并限制堆栈不能超过 64K 字节，即不超过一段，尽管有时各个程序模块使用自己独立的堆栈。

有时，我们知道了某一个存储单元的段地址，也知道它的偏移量，但这个段地址和偏移量并没有构成一个远指针。Turbo C 中提供了如下 4 个宏，使得可以从这个内存单元中读(或往这个内存单元中写)1 个字节(或 1 个整数)。

```
char peekb(unsigned segment, unsigned offset)
int  peek (unsigned segment, unsigned offset)
void pokeb(unsigned segment, unsigned offset, char value)
void poke (unsigned segment, unsigned offset, int value)
```

下面是从 ROM 的地址 FFFF: 000E 中读 1 个字节的例子，这个字节实际上就是微机类型的标志字节.

```c
#include <general.h>
void main( ) {
{
    printf("PC model = hex %x",
        (unsigned char) peekb(0xffff, 0x000e));
}
```

这4个宏以及下面将介绍的3个宏的定义都在头文件dos.h中，但在头文件general.h中包含有#include <dos.h>这样一行，所以在上面这个例子中只写了#include <general.h>。因为下面经常要用到general.h，故单独把它列出来。

```c
/* general.h */
#include <conio.h>
#include <dos.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#define boolean int
#define TRUE     1
#define FALSE    0
#define BLANK    ' '
#define CR       13
#define beep     putchar('\a')
#define newline putchar('\n')
#define EMPTYSTR ""
#define FARNULL (void far *)NULL
#define INTA00  0x20
#define EOI     0x20
#define strempty(s)  (*(s)==0)
boolean strequalsf (  char *s, char *t, char **ps, char **pt);
boolean strequalsb (  char *s, char *t, char **pm, char **pn);
#define strchrf strchr
char     *strchrb  (char *s, char *p, char c);
#define strpbrkf strpbrk
char     *strstrf  (char *s, char *t);
char     *strstrb  (char *s, char *t);
char     *stralloc  (char ch, unsigned N);
char     *strsubst (char *s, char *a, char *b);
char     *strltoa  (long N, char *s, int radix);
char     *strshrte  (double x, char *s);
char     *strultoe  (unsigned long N );
char     *strtomoney  (double x, int code);
char     *strtime      (char *timestr, const struct tm *t,
                         boolean twentyfourhours, char separator);
char     *strday       (char *datestr, const struct tm *t, int format,
                         char separator);
#define leap(year) ((year)%4==0 && ((year)%100!=0 || (year)%400 ==0))
int      dayofyear  (int day, int month, int leapyear);
long     gregorian  (int day, int month, int year);
```

```
#define  weekday    (day, month, year) (gregorian(day, month, year)+5)%7
unsigned envseg    (unsigned PSP);
char      *progname  (unsigned PSP);
void      memrymap   (void);
unsigned extmem      (void);
void      ctryinfo (unsigned code, struct country  *c);
void      peep       (void);
int       rstorisr   (int N, char  *id);
int       rstorint   (char  *id);
```

有时，我们知道了段地址和偏移量，需要把它们合并成一个远指针。Turbo C 的宏 MK__FP 可以用来做这件事。

```
(void far  *)MK__FP(unsigned segment, unsigned offset)
```

Turbo C 的另外两个宏正好做相反的事情，它们从一个远指针中分解出段地址和偏移量。

```
(unsigned) FP__SEG(void far  *p)
(unsigned) FP__OFF(void far  *p)
```

Turbo C 中还提供了其它一些手段，使得可以像汇编语言中一样，知道了偏移量和段地址在哪一个段寄存器，就可以用如下这样的程序来存取某一内存单元：

```
char peekb(unsigned segment, unsigned offset) {
{
      __ES=segment;
      return  *(unsigned char __es  *)offset;
}
```

这个程序中用伪变量__ES 把参数 segment(段地址)存放到寄存器 ES 中，用修饰符__es 把参数 offset(偏移量)强制转换为指向字符的指针，但这个指针是相对于寄存器 ES 而不是缺省的 DS，然后再去取 1 个字节。由于这个程序(函数)的名字也是 peekb，与 Turbo C 提供的宏同名，如果要用用户定义的 peekb，则在源程序中不应包含头文件 dos.h。

2.2 六种编译模式

2.2.1 概述

Turbo C 提供了六种编译模式。编译模式有时也称为寻址模式或内存模式，因为它处理的就是如何在内存中为程序，数据，堆栈分配空间并存取它们，这六种模式是：微模式 tiny，小模式 small，紧凑模式 compact，中模式 medium，大模式 large，巨模式 huge。它们之间的关系如表 2-1 所示。

表 2-1　6 种编译模式

	小程序	大程序
小数据	微，小	中
大数据	紧凑	大，巨

所谓小程序就是只有一个程序段，当然不超过 64K 字节，缺省的码(函数)指针是 near。所谓大程序就是有多个程序段，每个程序段不超过 64K 字节，但总程序量可超过 64K 字节，缺省的码指针是 far。所谓小数据就是只有一个数据段，缺省的数据指针是 near。所谓大数据就是有多个数据段，缺省的数据指针是 far。下面还会逐个谈到它们之间的差别，并通过同一程序在六种不同模式下的输出结果，来进一步加深对这六种模式的理解。但先要强调一点：无论使用哪一种编译模式，单个的 Turbo C 源文件不可能生成大于 64K 字节的代码，也不能生成大于 64K 字节的静态(包括全局)数据。例如下面这个程序：

```
int  a[15000], b[20000];
void main( ) {}
```

在任何模式下都不能编译。这是因为，两个数组所要求的总存储量达 70K 字节。编译时会报出"Too much global data defined in file"的出错信息。为了处理大于 64K 字节的代码或静态数据，必须分成几个源文件。以上面这个程序为例，可以分成文件 A1.C 和 A2.C，分别用巨模式对这两个源文件进行编译，最后连接成一个可执行文件.

```
a1.c                 a2.c                 a.prj
int  a[15000];       int  b[20000];       a1
void main( ) {}                           a2
a1.obj(30k)          a2.obj(40k)          a.exe(71k)
```

六种编译模式的差别是：它们对来自不同源文件的码和数据段的处理不同，对动态分配的堆空间的处理不一样，对指针使用也不一样。此外，它们在所形成的.obj 文件中传给连接程序的信息也不一样，以便连接程序相应地安排码段和数据段，把相应的说明放在.exe 文件的头中并借此通知 DOS：当执行这个程序时如何装入码段和数据段，如何设置各个段寄存器。

用于演示六种编译模式的程序是由两个源文件 X.C 和 Y.C 组成的，如下所示：

```
/*  X.C  */
#include <general.h>
void a( )
{
    static int b;
    int c;
    printf("In function A              \n"     );
    printf("  CS               : %X      \n", _CS);
```

```
    printf("  DS              : %X        \n", _DS);
    printf("  SS              : %X        \n", _SS);
    printf("  Static      B   : %p        \n", &b );
    printf("  Automatic C     : %p        \n", &c );
}

/*   Y.C   */
#include <general.h>
int  d;
void main( ) {
    int e;
    a( );
    printf("In function main             \n");
    printf("  CS              : %X        \n", _CS);
    printf("  DS              : %X        \n", _DS);
    printf("  SS              : %X        \n", _SS);
    printf("  Global      D   : %p        \n", &d );
    printf("  Automatic   E   : %p        \n", &e );
    printf("  Heap address    : %p        \n", malloc(2));
#if defined(___TINY___)||defined(___SMALL___)||defined(___COMPACT___)
    printf("Function      A   : %Np       \n", a  );
    printf("Function   main   : %Np       \n", main);
}
#else
    printf("Function      A   : %Fp       \n", a  );
    printf("Function   main   : %Fp       \n", main);
}
#endif
```

第一个源文件包含函数 a 和一个静态(局部)变量 b，第二个源文件包含主函数 main 和一个全局变量 d。两个源文件中各含有一个自动变量 c 和 e。第二个源文件的主函数 main 调用了第一个源文件中的函数 a，还调用了 Turbo C 的库函数 malloc 去分配一块堆空间。两个源文件是分别编译的，然后再通过连接程序连接起来。

通过以六种不同模式编译这两个源文件，可以看到它们是如何为码、数据和堆栈段分配空间，可以看到静态变量、自动变量和堆变量分别存放在什么位置，函数放在什么地方。正如下面将要看到的那样，在某些模式下，数据指针是 near 而函数指针是 far；在另一些模式下情况又正好相反。对于数据指针，不管是 far 还是 near，printf 函数中的格式说明%p 都能把指针正确地打印出来。对于函数，指针%p 就没有这个功能。所以，在 main 函数中必须加条件编译控制行#if、#else 和#endif。

2.2.2 微模式

在微模式下，整个程序只有一段，这个段内包含码、静态和全局数据、堆栈和堆。因为只有一个段，在执行时 DOS 将把寄存器 CS、DS、SS 设置为相等，全部指向这个段。在这个段内，码是首先装入的，地址最低，接着是静态变量和全局变量，然后是堆，最后是堆栈。堆和堆栈都是动态的，堆从低地址往高地址增长，堆栈从高地址往低地址增长。

若两者相碰，则表示内存空间已耗尽。在微模式下，所有指针都是 near，且都是相对于寄存器 CS、DS 和 SS 的。对于用微模式编译并连接生成的.exe 文件，可以用 DOS 的 exe2bin 实用程序转换为.COM 文件。

从表 2–2 演示程序的输出结果可以看出，函数 a 比函数 main 的地址低，变量 b 比变量 d 的地址低。 这是因为，在连接时是 x.obj 在前，y.obj 在后。

表 2–2 演示程序的输出结果

	微模式	小模式	紧凑模式	中模式	大模式	巨模式
In function A						
CS :	74C8	74B1	74B1	74F9	74FD	74FE
DS :	74C8	75CC	7629	75EC	764A	7674
SS :	74C8	75CC	767A	75EC	76A0	76BB
Static B :	1704	048C	7629:04C8	049A	764A:04D6	7674:0002
Automatic C :	FFD0	FFD0	767A:0FD6	FFCC	76A0:0FD4	76BB:0FD0
In function main						
CS :	74C8	74B1	74B1	74FE	7502	7503
DS :	74C8	75CC	7629	75EC	764A	767B
SS :	74C8	75CC	767A	75EC	76A0	76BB
Global D :	1706	048E	7629:04CA	049C	764A:04D8	767B:0004
Automatic E :	FFD6	FFD6	767A:0FDE	FFD4	76A0:0FDC	76BB:0FDA
Heap Address :	1792	051A	777C:000C	0568	77A2:000C	77BD:000C
Function A :	0283	01A5	0167	74F9:000E	74FD:000D	74FE:0003
Function main :	02C1	01E3	01AE	74FE:0004	7502:000C	7503:0009

2.2.3 小模式

小模式是最常用的模式，本书中大部分例子都是用小模式编译的。虽然小模式与微模式一样，都是小数据、小程序模式，但它与微模式有两点重要的差别。第一，码和数据／堆栈／堆段是分开的，所以 CS 不等于 DS 和 SS。第二，除了和数据／堆栈共用一个段的堆外，还有一个远堆，以 far 指针进行存取。从数据／堆栈段的末尾直到常规内存的末尾都是属于远堆。

因为码、静态数据和(近)堆仍然在同一个段内，所以小模式下缺省的数据指针和函数指针都是 near。结果，在小模式下不能直接通过该模式下的 Turbo C 函数来处理远堆中的变量。然而，只要程序员提供自己的操作函数，就可以存取整个远堆中的任一单元，即可以使用整个常规内存。有关远堆的一些细节见 2.4 节。

2.2.4 中模式

在数据／堆栈／堆的分配方面，中模式与小模式是一样的，差别在于码段的分配。在中模式下，来自不同源文件的码模块放在不同的码段内。严格地讲，同一源文件内的各函数也是放在不同码段内。各码段的总空间数只受微机上可用内存的限制。因为有多个码段，所以 Turbo C 必须用 far 函数指针。在演示程序输出的结果中函数 a 的地址为 74F9: 000E，函数 main 的地址为 74FE: 0004。函数 a 的地址较低，是因为在连接时包含函数 a 的 x.obj 在前。在中模式下，堆仍然有近堆和远堆之分。

2.2.5 紧凑模式

紧凑模式在概念上是最简单的，码、静态数据、堆栈、堆各有其自己的段。堆只有远堆，没有近堆。像小模式和中模式中的远堆一样，堆是用 far 指针来存取的。可以用 Turbo C 的库函数来处理堆变量。所有数据指针都是 far，函数指针都是 near。从演示程序的输出中可以看出，CS、DS、SS 三个寄存器的值彼此不等。值得注意的是，静态数据的总量仍不可超过 64K 字节。

2.2.6 大模式

在静态数据／堆栈／堆的分配方面，大模式等同于紧凑模式。在码的分配方面，大模式等同于中模式。无论是数据指针还是函数指针，一律都是远指针。与紧凑模式一样，静态数据的总量不可超过 64K 字节。

2.2.7 巨模式

巨模式取消了静态数据总量不可超过 64K 字节的限制。来自不同源文件的码放在不同的段内，来自不同源文件的静态数据也是放在不同的段内，只有堆栈是合在一起的。2.2.1 节中举的那个例子就是利用了这一特点。从演示程序的输出中也可以看出，当从函数 main 内调用函数 a 时，不但 CS 改变了，DS 也改变了。当然，两个函数共用了同一个堆栈，否则就无法正确返回。应该注意的是，不要把巨模式和巨指针混为一谈，在巨模式下缺省的指针仍是 far 而不是 huge。

2.3 堆栈的组织

Turbo C 堆栈是用来存储其生命期与函数生命期相同的数据，这样的数据包括函数参数和函数体内定义的自动变量。为了阐明函数堆栈内部各数据的存放关系，设有这样一个函数定义：

```
long f(char a, int b)
{
    int  c;
    char d;
    :
```

:

每当调用函数 f 时，调用者首先按相反顺序，即按从右到左的顺序把调用参数压入堆栈。本例就是先压入 b，再压入 a。尽管参数 a 是字符型，但仍压入 16 位，因为 80X86 的机器没有 8 位的压栈指令。在压入参数之后，根据调用指令是 near 还是 far，再压入 2 个或 4 个字节的返回地址。

进入被调用函数 f 之后，它首先把寄存器 BP 的当前值压入堆栈，并把 SP 寄存器的值拷贝到 BP 寄存器。接着再次按照相反的顺序在堆栈内建立起函数体内的各个自动变量，本例里就是先 d 后 c。直到此时，堆栈的内容将会如下所示:

:
:
b
a
返回地址
保留的 BP
d
c

这里之所以要对 BP 和 SP 作如此处理，目的有 3 个。第一，为了利用 BP 作地址寄存器，通过[BP ± n]这样的寻址方式到堆栈中存取调用者传过来的参数和被调用函数自己的自动变量。因为在 80X86 中规定，当用 BP 作地址寄存器时，缺省的段地址是 SS 而不是 DS。第二，腾出 DS 和其它地址寄存器，仍用来存取缺省的数据段内的数据。第三，腾出 SP 以便在函数体内再调用其它函数。

当函数 f 完成了它的工作以后，它就把返回值放到相应的位置。如果返回值是 char 型，则在返回前先强制转换为 int 型。凡是返回值占两个字节的都通过寄存器 AX 返回，凡是返回值占 4 个字节的都通过寄存器对 DX: AX 返回。超过 4 个字节的 struct 返回值，则被放在一个静态变量内，返回的是指向这个变量的指针。double 返回值是放在数值协处理器的 top__of__stack 寄存器内或协处理器软件模拟包内与这个寄存器等效的地方。接着函数 f 把 BP 拷贝到 SP，从堆栈中弹出入口时保留的 BP 值到 BP 寄存器。最后执行一条 near 或 far 返回指令，返回到调用者。返回以后，调用函数必须把调用时压入堆栈的参数从堆栈中清除。

上面这一套函数调用规则就叫做 C 调用规则。从这个过程中可以看出，调用函数和被调用函数在参数数目上可以不一致。如果调用函数压入了过多的参数，被调用函数不存取这些多余参数是没有什么影响的，调用函数在重新获得控制权后，会正确地把这些参数清除掉。如果调用函数压入了过少的参数，被调用函数就可能把一些并非参数的内容取来作为参数而产生意想不到的结果。为了克服这个困难，如果参数数目是不定的，那么第一个参数最好是说明随后的参数的个数。

另一套不同的函数调用规则叫做 PASCAL 规则，它与 C 调用规则有两点重要的差别。第一，压入参数的顺序是从左到右。第二，被调用函数的工作完成以后，从堆栈中弹出参数是由被调用函数而不是由调用函数去完成。PASCAL 调用规则要求调用函数和

被调用函数参数个数完全一致。顺便说一句，Turbo PASCAL 语言使用的不是 PASCAL 调用规则，而是一种更为精心设计的堆栈格式，使得从被嵌套的函数内可以存取函数的自动变量。

2.4 堆的组织

前面已经说过，在小模式和中模式下，堆有近堆和远堆之分，处理办法也不一样。近堆和堆栈共享一个段，它们相向增长，如果相遇，则说明缺省数据段已耗尽。远堆使用了缺省数据段之上直至常规内存末尾的整个空间。为了管理这两个堆，Turbo C 提供了两组相应的函数:

coreleft	farcoreleft
realloc	farrealloc
malloc	farmalloc
free	farfree
calloc	farcalloc

左边的近堆函数用近指针寻址各个堆变量，所用的参数也都是 16 位的 unsigned 型。右边的远堆函数用远指针寻址各个堆变量，所用的参数也都是 unsigned long 型。

在微模式下没有远堆，在紧凑模式、大模式和巨模式下只有一个不改堆，其组织形式如同远堆。但在这三种模式下，既可以使用近堆函数也可以使用远堆函数存取堆中变量。这是因为，不管使用哪一种堆函数，这三种模式决定了所有数据指针是 far。如果使用近堆函数，则表示所需内存容量的参数 size 还必须是 unsigned 型的 16 位数。如果必须处理大于 64K 字节的内存块，还必须使用远堆函数。

分配和释放是随机进行的，没有一定的次序，结果就造成了各个堆变量在堆中是不连续的。Turbo C 用一个链表来处理这些堆变量。在每一个堆变量的前面都有一个头，头中包含两个信息: 此变量的长度和指向下一个堆变量的指针。对于小数据模式，每个头占 4 个字节，对于大数据模式，每个头占 8 个字节。

为了说明分配、释放、再分配在堆中是如何进行的，请看下面这个演示程序 htap.dem 的输出结果。

```
#include <general.h>
#define report printf("coreleft = %u\n", coreleft( ));

void main( )
{
    void *p, *q, *r;
    printf("                              "); report; p = malloc(1);
    printf("p = malloc(1)      = %p; ", p); report; q = malloc(2);
    printf("q = malloc(2)      = %p; ", q); report; p = realloc(p, 3);
    printf("p = realloc(P, 3)  = %p; ", p); report; r = malloc(1);
    printf("r = malloc(1)      = %p; ", r); report; free(q);
    printf("      free(q)                 "); report; free(p);
```

```
    printf("          free(p)                    "); report;
}
```

这个程序产生的输出如下:

```
                              coreleft = 63952
p =   malloc(1)    = 0500; coreleft = 63946
q =   malloc(2)    = 0506; coreleft = 63940
p = realloc(P, 3) = 050c; coreleft = 63932
r =   malloc(1)    = 0500; coreleft = 63932
      free(q)              coreleft = 63932
      free(p)              coreleft = 63946
```

这个演示程序是用小模式编译的。首先，coreleft 报告可用的内存量。其次，malloc 建立单字节堆变量 p 和双字节堆变量 q。因为总是以 2 字节整数倍进行分配的，所以单字节变量 p 实际上也占用两个字节的空间。每个堆变量还需要 4 个字节的头。这样，每分配一个堆变量，内存容量就减少 6 个字节。接着，realloc 把变量 p 扩大到 3 个字节，这就要求重新分配，返回的指针也指向了新地址 050C。重新分配的堆变量 p 占用了 8 个字节。包括它的头。尽管此时原来占用的 6 个字节已经释放了，但 coreleft 仍报告减少了 8 个字节，而不是减少了两个字节。这是因为 coreleft 报告的只是堆中最上面最后一个变量之后连续可用的内存容量。也就是说由于堆的碎片化，coreleft 报告的值是不准确的。接着，程序又分配了一个单字节变量 r，它占用了第一次分配给变量 p 后来又被释放的那 6 字节空间。在此之后，程序释放变量 q，在变量 r 和 p 之间留下一个空洞。应该注意，分配 r 和释放 q 都不影响 coreleft 报告的值。最后，程序释放变量 p。此时，coreleft 报告的值才是准确的，因为堆中只在其开始部分剩一个变量 r 了。

下面这个 farheap.dem 程序演示了如何从远堆中分配一个大于 64K 字节的数组 a。数组 a 是由 9000 个 double 型元素组成的，共需 72K 字节。把函数 farcalloc 返回的远指针强制转换为 huge 指针，以后就可以用这个 huge 指针存取数组中的各个元素。

```
#include <general.h>
void main( ) {
     int i, n = 9000;
     double huge *a;
     double sum;
     a = (double huge *) farcalloc(n, sizeof(double));
     for (i = 0        ;  i < n;  a[i++] = i);
     for (i = 0, sum = 0;  i < n;  sum    += a[i++]);
     printf("a[i] = i  for  i = 0 .. n-1 ;  n = %d\n", n);
     printf("sum of all  a[i] = %8.0f\n", sum);
     printf("(n-1)n/2            = %ld  \n", (long) n*(n-1)/2);
}
```

2.5 其它内存操作函数

Turbo C 中还提供了许多有关内存拷贝、比较、设置和查找的函数。这些函数的说明都在头文件 mem.h 中。一般来说，它们都不牵涉到什么结构，而是直接对内存进行操作。这些函数可对简单的字节数组进行操作，也可实现 C 语言不直接支持的对数据结构的操作，如用一个数组对另一个数组赋值，数组或 C 结构之间的比较等。

用于内存之间拷贝数据的 Turbo C 函数有如下 5 个:

```
void  * _Cdecl  memccpy (void  * dest, const void  * src,
                         int c, size_t n);
void  * _Cdecl  memcpy  (void  * dest, const void  * src, size_t n);
void  * _Cdecl  memmove (void  * dest, const void  * src, size_t n);
void    _Cdecl  movmem  (void  * src, void  * dest, unsigned length);
void    _Cdecl  movedata(unsigned srcseg, unsigned srcoff,
                         unsigned dstseg, unsigned dstoff, size_t n);
```

函数 memcpy 从源 src 拷贝 n 个字节到目 dest。如果源和目有重叠的地方，则结果不一定正确。

函数 memccpy 与 memcpy 类似，但若被拷贝的字节中有字符 c，则在拷贝完这个字符后也停止拷贝，返回的指针指向目 dest 中的下一个字节位置。若 n 个字节全部拷贝完，则返回的指针为空。

函数 memmove 和 movmem 也用于拷贝，但它们都解决了源和目重叠的问题。函数 movmem 一反通常"目＝源"这样一个参数顺序，而是源在前，目在后。

在小模式和中模式下，前面 4 个拷贝函数所接收的源和目指针都只能是近指针，不能用来拷贝远数据段内的数组。函数 movedata 克服了这个缺陷，它允许指定源和目的段地址和偏移量，它没有解决源和目重叠的问题，也要求源参数在前，目参数在后。

Turbo C Tools 中的函数 utmovmem 与 Turbo C 的 movedata 是类似的，但它自动解决了源和目的重叠问题。

```
void  utmovmem(const char far  * psource, char far  * ptarget,
               unsigned int length);
```

用于内存之间比较的 Turbo C 函数有如下两个:

```
int   _Cdecl  memcmp  (const void  * s1, const void  * s2, size_t n);
int   _Cdecl  memicmp (const void  * s1, const void  * s2, size_t n);
```

这两个函数都是比较两个字节数组的前 n 个字节，根据 s1 是小于、等于还是大于 s2，返回值分别为小于 0、等于 0 和大于 0。但函数 memcmp 是精确比较，把每个字节看作无符号 8 位数，而函数 memicmp 把每个字节看作一个字符，忽略大小写的差别。

用于内存设置的 Turbo C 函数有如下两个:

```
void  * _Cdecl  memset(void  * s, int c, size_t n);
```

```
void  __Cdecl  setmem(void *dest, unsigned length, char value);
```

这两个函数都是设置一块内存区域为某一个字节值，参数顺序不一样，返回值也不一样，但实际作用看不出有什么区别。

用于从一个内存块的头 n 个字节中查找某一个字符的 Turbo C 函数是 memchr:

```
void  *__Cdecl  memchr  (const void *s, int c, size_t n);
```

如果找到了，则返回的指针指向字符 c 第 1 次出现的位置。如果找不到，则返回的指针为空。

第3章　鼠标输入

这一章讲述程序员在他的程序中如何使用鼠标并提供了一些方便使用鼠标的函数。在购买鼠标器时，一般都同时提供一个鼠标驱动程序。如果你的鼠标是与 Microsoft 鼠标相兼容的，则本章提供的函数可以直接使用；如果不兼容，则可能要做少许修改。

3.1　鼠标驱动程序的基本功能

DOS 操作系统本身不支持鼠标，故在使用鼠标前必须安装鼠标驱动程序。或者是通过 CONFIG.SYS 文件，或者是通过 AUTOEXEC.BAT 文件。一旦安装了鼠标驱动程序，并对它进行了初始化，就可以打开鼠标，由鼠标驱动程序管理鼠标器的许多细节动作。例如，程序员根本不需要跟踪鼠标的移动而相应地更新鼠光标，鼠标驱动程序会自动做这件事。程序员所要做的就是询问鼠标器状态是否发生了改变，然后采取相应行动。鼠标驱动程序所提供的功能较多，各厂家的驱动程序也不完全一致。表 3-1 仅列出 Microsoft 公司鼠光标驱动程序的部分功能和其参数。这一部分是基本的，但对于一般应用也足够了。

表 3-1　鼠标驱动程序功能码和参数

码	功　能	入口参数	出口参数
0	复位鼠标	m=0	m1=鼠标状态，如果安装，则为-1，否则为 0。 m2=鼠标按钮数
1	显示鼠光标	m1=1	
2	不显示鼠光标	m1=2	
3	取按钮状态	m1=3	m2=各按钮状态，见注释 m3=新的 X 坐标 m4=新的 Y 坐标
4	设置鼠光标位置	m1 = 4 m3=新的 X 坐标 m4=新的 Y 坐标	
5	取某一按钮按下信息	m1=5 m2=按钮号 0 为左按钮	m1=各按钮状态，见注释 m2=自上一次调用以来该按钮被按下的次数

续表

码	功 能	入口参数	出口参数
		1 为右按钮	m3＝最后一次按下时鼠标的 X 坐标 m4＝最后一次按下时鼠标的 Y 坐标
6	取某一按钮释放信息	m1＝6 m2＝按钮号 0 为左按钮 1 为右按钮	m1＝各按钮状态，见注释 m2＝自上一次调用以来该按钮被释放的次数 m3＝最后一次释放时鼠标的 X 坐标 m4＝最后一次释放时鼠标的 Y 坐标

注释：鼠标驱动程序返回的各按钮状态的信息格式如下所示:

位	等于 0 时	等于 1 时
0	左按钮未按下	左按钮正在按下
1	右按钮未按下	右按钮正在按下
2	中按钮未按下	中按钮正在按下

在汇编语言一级，程序是通过中断 0x33 存取鼠标驱动程序，可以取鼠光标和鼠标器按钮状态，可以改变鼠光标的颜色和形状，可以使鼠光标可见或不可见，还可以执行许多其它操作。对于最常使用的那些功能，4 个参数总是分别存放在 ax、bx、cx、dx 4 个寄存器内，这使得与驱动程序的接口变得很简单，很规范，如下面工具包中的函数 mouse 所示。

3.2 与鼠标接口的 C 函数工具包

3.2.1 14个工具函数

直至 2.0 版，Turbo C 都还没有提供与鼠标接口的函数。为了方便读者在 C 语言中使用鼠标，本章提供了一个鼠标工具函数库。像本书中所有的工具包一样，本工具包也是由一个头文件和一个源程序文件组成。头文件内包含一大堆宏定义和函数说明。为了方便起见，尽管这个工具包是鼠标工具包，但在宏常数定义中也包含了许多键盘的宏定义。这些键盘定义虽然在这儿用不着，但在既允许鼠标输入又允许键盘输入的程序中是用得着的，所以把两者放在一起。在以后各章许多工具包中可以看到使用这个头文件的例子。这个工具包主要是面向文本应用，而不是图形应用。所要求的显示卡应该是 CGA、

MCGA、ATT400，大力神单色图形卡这样几种。如果是别的图形卡，则应该对工具包做些修改。

这个工具包中一共包括如下 14 个工具函数:

```
extern    void mouse (int *m1, int *m2, int *m3, int *m4);
extern    int  check__mouse__driver (int need__mouse);
extern    int  init__mouse (int need__mouse, int gd, int gm);
extern    int  mouse__reset (void);
extern    void move__mouse (int x, int y);
extern    void mouse__on (int code);
extern    void mouse__off (int code);
extern    void mouse__grph__posn (int *x,  int *y);
extern    void mouse__txt__posn (int *x, int *y);
extern    int mouse__in__box (int graphflag,int left,int right,
                    int top,int bottom);
extern    int  button__release (int b);
extern    int  button__press (int b);
extern    int  button__state (void);
extern    int  mouse__trigger (int button__dir);
```

(1) mouse。这个函数提供了 C 语言中直接调用鼠标驱动程序的办法。它先把调用参数装入相应的寄存器，然后调用中断 0x33。从中断返回以后，又把寄存器的内容赋回给相应的参数，最后返回。对于鼠标驱动程序中已经提供而在本工具包中没有提供的功能，都可以通过这个函数去调用。

(2) check__mouse__driver。这个函数检查是否已安装了鼠标驱动程序，方法是调用 Turbo C 的 getvect 函数，看看中断 0x33 的向量是否为 NULL 或一条 IRET 指令 (0xcf)。如果不是，则认为安装了鼠标驱动程序，否则认为没有安装。当然，即使中断 0x33 处理程序存在，它也可能不是鼠标驱动程序，这就没有任何办法正确判断它了。参数 need__mouse 是为了使应用程序对是否存在鼠标驱动程序是透明的。如果 need__mouse = 1，即一定要有鼠标，这时若没安装鼠标驱动程序，则显示出一个错误信息。如果希望鼠标是可选的，则应设置 need__mouse = 0。此时，若存在鼠标驱动程序，则返回 1，否则返回 0，不会报错。

(3) init__mouse。这个函数初始化鼠标。如果找不到鼠标驱动程序而又需要鼠标，则终止程序的运行，否则返回 mouse__initialized = 1。参数 graphdriver 说明是什么图形卡和驱动程序，graphmode 说明是什么图形方式。这两个参数更详细的情况可参考手册中 initgraph 函数的说明，它对图形的影响将在第 15 章详细讨论。应该注意的是，这里的图形方式号是 Turbo C 规定的图形方式号，与 PC 机 BIOS 中规定的显示方式完全不是一回事。如果仅仅想用文本方式，则这两个参数都应置为 0。

(4) mouse__reset。首先，如果鼠标是打开的，这个函数将关闭它。然后，复位鼠标状态。

(5) move__mouse。这个函数移动鼠光标到指定的文本位置。

(6) mouse__on 和 mouse__off。这两个函数牵涉到鼠光标的一些特殊问题。光标有两种，一种是普通光标，它由图形显示卡硬件实现，并由 DOS 加以维护。另一种是鼠光

标，它由软件实现，并由鼠标驱动程序维护。鼠光标是由鼠标驱动程序通过往 RAM 中直接写而画出来的。每次光标要移往一个新位置时，这个位置下的内容被暂时保存起来，当鼠光标再次被移开时，再恢复这些内容。一般情况下，这样会工作得很好，但是如果正巧要往鼠光标所在位置写，当光标被移开时，将仍然以原先保存的光标代替刚刚写入的内容，其结果将是光标上写不进内容，屏幕上留下许多光标。解决这个问题的办法，就是在每次往屏幕上写内容前，先临时关闭鼠光标，写完之后再打开鼠光标。当然，更聪明一点的做法是在写完之后只有光标原来是打开的才再次打开，否则仍然关掉。函数 mouse_on 和 mouse_off 就提供了这样的功能。参数的含义是:

restoreflog =0 : 无条件打开鼠光标，置状态为打开
=1 : 若状态标志原来是打开的则打开，否则仍关闭。
tempflag =0 : 无条件关掉鼠光标，置状态为关闭
=1 : 关闭鼠光标，根据鼠光标在此之前是打开的还是关闭的，相应地置状态

下面这个例子先关闭光标，有条件地置状态，然后更新屏幕，最后再有条件地打开光标。

```
mouse_off (1);
cprintf ("Hello World");
mouse_on (1);
```

以这种方式打开和关闭鼠光标，可以避免许多麻烦。以后的许多例子中使用的都是这种方式。

(7) mouse_grph_posn。如果鼠标已初始化了，则返回光标的图形坐标。若未初始化，则返回 (0,0)。

(8) mouse_txt_posn。如果鼠标已初始化了。则返回光标的文本坐标。若未初始化，则返回 (1,1)。

关于坐标，还有一点需要说明。Turbo C 的文本坐标是从 (1,1) 开始的，而鼠标驱动程序返回的总是图形坐标，并且是从 (0,0) 开始。但除了 Hercules (大力神) 单色图形卡以外，对于所有其它显示卡和图形显示方式，不论是什么显示卡和什么图形方式，鼠标驱动程序总是把屏幕看成是由 640＊200 个像素组成的。但对于 CGA、MCGA、ATT400 这样一些显示卡，当图形方式为 0～3 时，水平方向只有 320 个像素，这就是工具包中所谓的低分辩率方式。对于高低两种分辩率，当由底层鼠标驱动程序的坐标换算为工具包的图形坐标时，分别除以 1 和 2。这个工具包中，认为字符一律是由 8＊8 个像素构成的，所以，对于高低两种分辩率，当由底层鼠标驱动程序坐标换算为工具包的文本坐标时，分别除以 8 和除以 16。

对于这种坐标换算，还有一种需加以特殊考虑的情况是大力神单色图形卡。在文本方式下，大力神卡是模拟 IBM 单色显示卡的。但与 IBM 的单色显示卡不一样，大力神卡还提供了单色图形方式，其分辨率为 720＊350。遗憾的是，BIOS 是按 IBM 标准行事的，没有为大力神卡的单色图形方式指定一个方式号，别的软件 (包括鼠标驱动程序) 也就无法知道当前使用的是否大力神卡。BIOS 用 0x0040: 0x0049 开始的一块内存存储它

的各个显示参数，其第1个字节就是当前显示方式。当在大力神文本方式时，这个字节是7，然而当转到大力神卡图形方式时，这个字节仍然是7，因为BIOS不承认大力神显示卡。但是，鼠标驱动程序实际上是可以支持大力神卡的。解决的办法是：调用Turbo C的initgraph函数对显示卡进行自动检测（见第14章），如果发现是大力神卡（gd=7），则把BIOS数据区的第1个字节改写为6（使用第0页）或5（使用第1页）。这样改动以后，鼠标驱动程序就知道是处于大力神卡图形方式了。

(9) mouse_in_box。如果鼠标在指定的矩形框内，则返回1，否则返回0。如果graphflag=1，则坐标是图形坐标，否则是文本坐标。两种坐标都是绝对坐标。为了能正确使用该函数，应先调用mouse_txt_posn，mouse_grph_posn或某些其它函数，以便先正确设置全局鼠标位置变量。

(10) button_press、button_release和button_state。正确理解这3个函数之间的区别很重要。头两个函数返回的值表示自上一次调用该函数以来，按钮是否又被按下过或是否又被释放过。如果是，则返回1，否则返回0。参数b是说明查询左按钮（b=0）还是右按钮（b=1）。函数button_state返回的值表示按钮当前是否按下。返回的整数值的位2、位1、位0分别代表中按钮、右按钮、左按钮，如果某个按钮正按下，则相应位为1，否则为0。这3个函数都设置内部的鼠标位置变量，以反映鼠标当前的位置。一个典型的拖动鼠标的过程总是通过button_press函数开始，在拖动过程中不断通过button_state函数检查状态，最后通过button_release函数结束拖动过程。仅有button_state函数是不够的，因为按钮正在按下或释放时，程序可能不正在检查按钮状态，错过时机。

(11) mouse_trigger。这个函数查看键盘上的键是否按下或鼠标按钮是否曾按下或曾释放。如果同时动作，则键盘优先于鼠标，左按钮优先于右按钮。

在这个工具包中除了上面列出的14个函数外，还有2个内部函数，5个全局变量，3个全局静态变量。内部函数low_res_mode根据图形卡和图形方式，判断当前是否处于低分辨率方式。内部函数set_mouse_posn把鼠标的当前位置记录在全局变量mouse_text_x、mouse_text_y、mouse_grph_x、mouse_grph_y中。在上述的14个函数中，init_mouse、mouse_reset、move_mouse、mouse_grph_posn、mouse_tex_posn、button_release、button_press、button_state 8个函数在其执行过程中都调用了set_mouse_posn函数，即都根据鼠标的实际位置设置了4个全局鼠标位置变量。正是通过这4个变量，程序始终跟踪鼠标的实际位置。

```
/*  mouse.h  */
#define  lo (f)          ( (f)  & 0xff)
#define  hi (f)          (lo (f>>8))
#define  CTRLC           0X2E03
#define  CTRLH           0X2308
#define  CTRLI           0X1709
#define  CTRLL           0X260C
#define  CTRLK           0X250B
#define  CTRLJ           0X240A
#define  CTRLU           0X1615
```

```
#define    CTRLR                  0X1312
#define    CRKEY                  0X1C0D
#define    CTRLCRKEY              0X1C0A
#define    UPKEY                  0X4800
#define    DOWNKEY                0X5000
#define    LEFTKEY                0X4B00
#define    RIGHTKEY               0X4D00
#define    SHIFTLEFT              0X4B34
#define    SHIFTRIGHT             0X4D36
#define    DELKEY                 0X5300
#define    INSKEY                 0X5200
#define    BSKEY                  0X0E08
#define    SPACEBAR               0X3920
#define    PGUPKEY                0X4900
#define    PGDNKEY                0X5100
#define    SHFTUPKEY              0X4838
#define    SHFTDNKEY              0X5032
#define    SHFTPGUPKEY            0X4939
#define    SHFTPGDNKEY            0X5133
#define    HOMEKEY                0X4700
#define    ENDKEY                 0X4F00
#define    ESCKEY                 0X011B
#define    ALT_D                  0X2000
#define    ALT_E                  0X1200
#define    ALT_I                  0X1700
#define    ALT_R                  0X1300
#define    ALT_S                  0X1F00
#define    ALT_T                  0X1400
#define    ALT_X                  0X2D00
#define    F10KEY                 0X4400
#define    LEFT_MOUSE_PRESS       0XFF01
#define    RIGHT_MOUSE_PRESS      0XFF02
#define    LEFT_MOUSE_REL         0XFF11
#define    RIGHT_MOUSE_REL        0XFF12
#define    M_RESET                0
#define    M_SHOW_CURS            1
#define    M_HIDE_CURS            2
#define    M_GET_STATUS           3
#define    M_SET_CURS             4
#define    M_GET_PRESS            5
#define    M_GET_REL              6
#define    M_SET_X_BOUNDS         7
#define    M_SET_Y_BOUNDS         8
#define    M_SET_G_CURS           9
#define    M_SET_T_CURS           10
#define    MOUSE_NEEDED           1
#define    MOUSE_OPTIONAL         0
#define    MOUSE_TEXT_MODE        0
extern    int  mouse_text_x;
extern    int  mouse_text_y;
extern    int  mouse_grph_x;
```

```
extern    int  mouse_grph_y;
extern    int  mouse_initialized;
extern    void mouse (int *m1, int *m2, int *m3, int *m4);
extern    int  check_mouse_driver (int need_mouse);
extern    int  init_mouse (int need_mouse, int gd, int gm);
extern    int  mouse_reset (void);
extern    void move_mouse (int x, int y);
extern    void mouse_on (int code);
extern    void mouse_off (int code);
extern    void mouse_grph_posn (int *x,  int *y);
extern    void mouse_txt_posn (int *x, int *y);
extern    int mouse_in_box (int graphflag,int left,int right,
                          int top,int bottom);
extern    int  button_release (int b);
extern    int  button_press (int b);
extern    int  button_state (void);
extern    int  mouse_trigger (int button_dir);

/*   mouse.c   */
#include  <bios.h>
#include  <dos.h>
#include  <conio.h>
#include  <process.h>
#include  <stdio.h>
#include  <mouse.h>

int  mouse_text_x;
int  mouse_text_y;
int  mouse_grph_x;
int  mouse_grph_y;
int  mouse_initialized;

static int prev_cursor_state=0;
static char far *bios_video_area= (char far *) 0x00400049L;
static int low_resolution=0;
static void set_mouse_posn (int *x,int *y);
static int low_res_mode (int gd,int gm);

void mouse (int *m1,int *m2, int *m3, int *m4)
{
     union REGS inregs,outregs;
     inregs.x.ax= *m1;
     inregs.x.bx= *m2;
     inregs.x.cx= *m3;
     inregs.x.dx= *m4;
     int86 (0x33,&inregs,&outregs);
     *m1=outregs.x.ax;
     *m2=outregs.x.bx;
     *m3=outregs.x.cx;
     *m4=outregs.x.dx;
}
```

```
int  check_mouse_driver (int need_mouse)
{
     void far  *address;
     address=getvect  (0x33);
     if( (address==NULL)  ||  (*  (unsigned char  *) address==0xcf))  {
          if (need_mouse)  {
               printf ("mouse driver not installed\n");
               exit  (1);
          }
          else return  0;
     }
     return  1;
}

int  init_mouse (int need_mouse, int gd, int gm)
{
     int m1;
     mouse_initialized=0;
     if (check_mouse_driver (need_mouse))  {
          if  (gd==7)   *bios_video_area=6;
          if  (low_res_mode  (gd,gm))  low_resolution=1;
          m1=mouse_reset  ( );
          if (m1)  {
               mouse_initialized=1;
               move_mouse  (0,0);
               mouse_on  (0);
          }
          else {
               if (need_mouse)  {
                    printf ("ERROR activating mouse ......\n");
                    exit  (1);
               }
          }
     }
     return mouse_initialized;
}

static int low_res_mode (int gd, int gm)
{
     if( (gd==1 || gd==2 || gd==8)  &&  (gm>=0 && gm<=3))  return 1;
     return  0;
}

int  mouse_reset (void)
{
     int x1,m1,m2,m3,m4;
     mouse_off  (1);
     m1=M_RESET;
     mouse  (&m1,&m2,&m3,&m4);
     set_mouse_posn  (&m3,&m4);
```

```
    return m1;
}

void move_mouse (int x, int y)
{
    int m1,m2,m3,m4;
    if (!mouse_initialized)  return;
    m1=M_SET_CURS;
    m3=x * 8;
    m4=y * 8;
    mouse (&m1,&m2,&m3,&m4);
    set_mouse_posn (&m3,&m4);
}

void mouse_on (int restoreflag)
{
    int m1,m2,m3,m4;
    if (mouse_initialized)  {
        if (!restoreflag || prev_cursor_state)  {
            m1=M_SHOW_CURS;
            mouse (&m1,&m2,&m3,&m4);
            prev_cursor_state=1;
        }
    }
}

void mouse_off (int tempflag)
{
    int m1,m2,m3,m4;
    if (mouse_initialized)  {
        if (prev_cursor_state)  {
            m1=M_HIDE_CURS;
            mouse (&m1,&m2,&m3,&m4);
            if (!tempflag)  prev_cursor_state=0;
        }
    }
}

void mouse_grph_posn (int *x, int *y)
{
    int m1,m2;
    if (mouse_initialized)  {
        m1=M_GET_STATUS;
        mouse (&m1,&m2,x,y);
        set_mouse_posn (x,y);
    }
    else {
        *x=0;
        *y=0;
    }
    return;
```

```
}

void mouse_txt_posn (int *x, int *y)
{
     mouse_grph_posn (x,y);
     *x=mouse_text_x;
     *y=mouse_text_y;
     return;
}

int  mouse_in_box (int graphflag,int left,int top,
                   int right,int bottom)
{
     int x,y;
     if (mouse_initialized)  {
          if (graphflag)  {
               x=mouse_grph_x;
               y=mouse_grph_y;
          }
          else {
               x=mouse_text_x;
               y=mouse_text_y;
          }
          if( (y>=top)  &&  (y<=bottom)  &&  (x>=left)  &&  (x<=right))
       return 1;
     }
     return 0;
}

int  button_release (int b)
{
     int m1,m2,m3,m4;
     if (mouse_initialized)  {
          m1=M_GET_REL;
          m2=b;
          mouse (&m1,&m2,&m3,&m4);
          set_mouse (&m3,&m4);
          if  (m2)  return 1;
     }
     return 0;
}

int  button_press (int b)
{
     int m1,m2,m3,m4;
     if (mouse_initialized)  {
          m1=M_GET_PRESS;
          m2=b;
          mouse (&m1,&m2,&m3,&m4);
          set_mouse_posn (&m3,&m4);
          if (m2)  return 1;
```

```
    }
    return 0;
}

int  button_state ( )
{
    int m1,m2,m3,m4;
    if (mouse_initialized)  {
        m1=M_GET_STATUS;
        mouse (&m1,&m2,&m3,&m4);
        set_mouse_posn (&m3,&m4);
        return m2;
    }
    return 0;
}

static void set_mouse_posn (int *x, int *y)
{
    if (low_resolution)  *x>>=1;
    mouse_grph_x=*x;
    mouse_grph_y=*y;
    mouse_text_x=*x/8;
    mouse_text_y=*y/8;
}

int  mouse_trigger (int button_dir)
{
    int k;
    if (bioskey (1))  {
        k=bioskey (0);
    }
    else {
        k=0;
        if (button_dir)  {
            if (button_press (0))  k=LEFT_MOUSE_PRESS;
            else if (button_press (1))  k=RIGHT_MOUSE_PRESS;
        }
        else {
            if (button_release (0))  k=LEFT_MOUSE_REL;
            else if (button_release (1))  k=RIGHT_MOUSE_REL;
        }
    }
    return k;
}
```

3.2.2 工具包应用举例

下面是一个应用上节中提供的部分函数来管理鼠标的程序例子。程序中设立了一个变量 highlite，它只取两个值：零和非零，分别表示要正常显示和加亮显示。每按一次鼠标左按钮，这个变量的值就求反一次。如果在按下鼠标左按钮的同时拖动鼠标，屏幕上的鼠

光标将跟着走动，并使所穿过的字符颜色变亮（highlite 等于非零）或变为正常（highlite 等于 0)。只有窗口内文本的颜色受变量 highlite 的影响。一次修改 1 个字符，在修改颜色之前关闭鼠光标，修改之后再打开鼠光标。在鼠标拖动过程中，程序不断地循环，检查鼠标移动的距离是否够 1 个字符。只有真正移过 1 个字符时，才做上述修改颜色的工作，这样就避免了由于循环过程中关闭和打开光标而引起的光标在原地闪烁的问题。还应注意到，即使仅仅为了加亮文本而不是重写文本，即只是为了修改颜色，也应先临时关闭鼠光标。这是因为鼠标驱动程序在保留和恢复鼠光标时，处理的不仅仅是鼠光标的形状，还包括鼠光标的显示属性。程序是不断循环的，直至按一下右按钮才退出。

```
#include  <bios.h>
#include  <conio.h>
#include  "mouse.h"

typedef  struct  texel_struct {
     unsigned char  ch;
     unsigned char  attr;
} texel;

void my_box (int xul,int yul,int xlr,int ylr,int btype);

void main ( )
{
     int  highlite=0;
     int  x=1, y=1;
     texel     t;
     init_mouse(MOUSE_NEEDED,MOUSE_TEXT_MODE,MOUSE_TEXT_MODE);
     my_box (19,9,66,14,2);
     window (20,10,65,13);
     clrscr ( );
     mouse_off (1);
     cprintf ("Here is some text for yiu to highlight\r\n");
     cprintf ("Click left button to toggle highlighting\r\n");
     cprintf ("Drag w/left button down to highlight\r\n");
     cprintf ("Click right button to exit");
     mouse_on (1);
     do {
          if (button_press (1))  break;
          if (button_press (0))  highlite=!highlite;
          if (button_state ( ) ==1)  {
               if (mouse_in_box (0,20,10,65,13))  {
                    if (x!=mouse_text_x || y!=mouse_text_y)  {
                         x=mouse_text_x;
                         y=mouse_text_y;
                         mouse_off (1);
                         if (highlite)  {
                              gettext (x,y,x,y,&t);
                              t.attr=15;
                              puttext (x,y,x,y,&t);
```

```
                    }
                    else {
                        gettext (x,y,x,y,&t);
                        t.attr=7;
                        puttext (x,y,x,y,&t);
                    }
                    mouse__on (1);
                }
            }
        }
    } while (1);
    mouse__reset ( );
    window (1,1,80,25);
    clrscr ( );
}

void my__box (int xul,int yul,int xlr,int ylr,int btype)
{
    static int boxcar[2][6] = {
        {218,196,191,179,192,217},
        {201,205,187,186,200,188}
    };
    int i,hzchar,vtchar;
    if (btype)  {
        hzchar=boxcar[btype-1][1];
        vtchar=boxcar[btype-1][3];
        gotoxy (xul,yul);
        for (i=xul; i<=xlr; i++)  putch (hzchar);
        gotoxy (xul,ylr);
        for (i=xul; i<=xlr; i++)  putch (hzchar);
        for (i=yul; i<=ylr; i++)  {
            gotoxy (xul,i);
            putch (vtchar);
            gotoxy (xlr,i);
            putch (vtchar);
        }
        gotoxy (xul,yul);      putch (boxcar[btype-1][0]);
        gotoxy (xlr,yul);      putch (boxcar[btype-1][2]);
        gotoxy (xlr,ylr);      putch (boxcar[btype-1][5]);
        gotoxy (xul,ylr);      putch (boxcar[btype-1][4]);
    }
}
```

3.3 Turbo C Tools 的鼠标支持函数

Turbo C Tools 6.0 版中提供了 19 个鼠标支持函数。与上一个工具包一样，这些函数也为与标准 Microsoft 兼容的鼠标驱动程序提供了一个 C 语言接口，也可以看作一个工具包。与上一个工具包相比，Turbo C Tools 不但提供的函数数目多，而且许多函数功

能也更强。但是，Turbo C Tools 鼠标支持函数毕竟没有囊括上一个工具包的所有功能，而后面许多章的工具包又都是建筑在上一个工具包的基础上的，所以本章还是介绍了上一个工具包。读者如有兴趣，完全可以用 Turbo c Tools 提供的函数重写 mouse.h 和 mouse.c 工具包，这将会大大地减少源码长度。

这 19 个函数大约可以分为 5 类：鼠标的初始化，询问鼠标的状态，鼠光标的位置和速度控制，鼠光标的形状和开关控制，鼠标硬件中断的处理。

3.3.1 鼠标的初始化

属于这一类的函数有下列几个：

```
int cdecl   moequip (void);
int cdecl   mogate (const DOSREG *pinregs,DOSREG *poutregs);
#define     moreset (void)    ((b__mouse=MO__UNKNOWN) ,moequip ( ))
```

(1) 函数 moequip 报告是否已安装了与 Microsoft 兼容的鼠标驱动程序。如果安装了，则通过返回值报告鼠标器按钮的个数，这个值同时也保存在全局变量 b__mouse 中，第一次调用这个函数时，它同时也将初始化驱动程序。

(2) 函数 moreset 的返回值与 moequip 相同，也把这个值设置在全局变量 b__mouse 中。同时，它也对驱动程序进行初始化，即设置如下初始条件：

.使鼠标不可见；

.将鼠标移到显示屏幕的中央；

.将鼠标的移动限制在最初的 25 行 80 列屏幕中；

.禁止任何由鼠标驱动程序自动调用的用户例程；

.使光笔摸拟有效；

.将灵敏度值设置为它的缺省值。

(3) 函数 mogate 不严格属于"初始化"这一类，它与上一节 mouse 工具包中的 mouse 函数相类似，是通往鼠标驱动程序的总入口，是所有鼠标支持函数中最低级的一个。它的两个参数都是指向结构 DOSREG 的指针，实际上即是输入寄存器和输出寄存器的内容。DOSREG 在头文件 butil.h 中定义如下：

```
struct dreg
{
unsigned ax,bx,cx,dx,si,di,ds,es;
}
#define DOSREG struct dreg
```

3.3.2 询问鼠标的状态

属于这一类的函数有下列几个：

```
int cdecl  mobutton (int event,unsigned *pbuttons,unsigned *pcount,
                     unsigned *pvert,unsigned *phoriz);
int far cdecl  mocheck (unsigned long event,unsigned long ignore,
```

```
                int option,unsigned long *pfound,
                unsigned *pvert,unsigned *phoriz);
int cdecl  mostat (unsigned *pbuttons,unsigned *pvert,
                unsigned *phoriz);
```

(1) 函数 mostat 报告鼠标器的位置和按钮状态。与上一节的工具包一样，这儿也是用无符号整数参数 pbuttons 位 0、位 1、位 2 分别表示左按钮，右按钮，中按钮，并为此定义了三个符号常数 MO_LEFT (1)、MO_RIGHT (2)、MO_MIDDLE (4)。参数 pvert 和 phoriz 返回以像素为单位的鼠标器位置，屏幕左上角的坐标为 (0,0)。

(2) 函数 mobutton 报告鼠标器所发生的事件。所谓事件，就是指是否有某一个按钮被按下了或被释放了。参数 event 是说明要查询哪一类事件，它是一个整数，其位 0、位 1、位 2 分别表示左按钮、右按钮、中按钮，其位 3、位 4 分别表示询问是否按下或是否释放。对应这 5 位，分别定义了 5 个常数：MO_LEFT (1) ,MO_RIGHT (2)，MO_MIDDLE (4)，MO_PRESS (8)，MO_RELEASE (16)。参数 pcount 返回自上次查询以来该事件发生的次数，但这个计数器在查询之后会被清 0。所以，如果在本程序的两次查询之间，另一个程序也查询了这个事件，则本程序得到的次数将比实际值小。参数 pvert 和 phoriz 返回最后一次指定事件发生时鼠标器的坐标位置，单位为像素。参数 buttons 的含义与函数 mostat 相同。

(3) 函数 mocheck 也是报告鼠标器所发生的事件，但它的事件的含义比函数 mobuttons 更广泛，不但有某个按钮的按下或释放，还有揿一下或揿两下，甚至还有某些键的按下。参数 event 就是说明要查询的事件的，它的各位的定义及其相应的符号常数如下：

符号	值	意义
MO_LEFT	0X0001	鼠标左按钮
MO_RIGHT	0X0002	鼠标右按钮
MO_MIDDLE	0X0004	鼠标中按钮
MO_PRESS	0X0008	按钮按下
MO_RELEASE	0X0010	按钮释放
MO_CLICK	0X0040	揿一下按钮
MO_DCLICK	0X0080	揿二下按钮
MO_HOLD	0X0800	按钮暂时处于指定位置
MO_RSHIFT	0X1000	右 shift 键按下
MO_LSHIFT	0X2000	左 shift 键按下
MO_CSHIFT	0X4000	Ctrl 键按下
MO_ASHIFT	0X8000	Alt 键按下

参数 ignore 指示需忽略的事件位的位屏蔽，它可以为下列值之一：

MO_LEFT	MO_RIGHT	MO_MIDDLE
MO_RSHIFT	MO_LSHIFT	MO_CSHIFT

从某种意义上说，事件可以分为基本事件和组合事件。因为"揿一下"是由"按下"和"释放"组合而成的，所以"揿一下"是组合事件，"按下"和"释放"是基本事件。但对于"揿两下"来说，"揿一下"、"按下"和"释放"就都是基本事件了。在 Turbo C Tools 中是用全局变量 b_clicklimit 和 b_dclicklimit 来调整"揿一下"和"揿两下"的最大时间间隔，两个变量均以 BIOS 时钟为测算时间的单位。b_clicklimit 的缺省值是 4，大约 220ms，b_dclicklimit 的缺省值是 9，大约 500ms。在参数 option 中，如果指定了 MO_CLEAR (0X0010)，则在 mocheck 函数执行完毕后，清除所查询的事件以及组成该事件的所有基本事件，但不清除以该事件为基本事件构成的更高一级的组合事件。如果指定了 MO_NOCLEAR (0X0020)，则不清除所查询的事件，以后还可以查询到这个事件。

参数 pfound 返回探测到的事件。

参数 pvert 和 phoriz 返回最后一次被查询事件发生时鼠标器的位置，(以像素为单位)。

3.3.3 鼠光标的位置和速度控制

属于这一类的函数有如下几个:

```
int cdecl  mocurmov (unsigned vert,unsigned horiz);
int cdecl  mogetmov (int * pvert,int * phoriz);
int cdecl  mojump (unsigned speed);
int cdecl  morange (int option,unsigned low,unsigned high);
int cdecl  mospeed (unsigned vert,unsigned horiz);
```

(1) 函数 mocurmove 使鼠光标移到指定的位置，坐标以像素为单位。如果指定的坐标值超出了范围，则移到最接近这个值的地方。

(2) 函数 morange 设置鼠光标移动的范围，以像素为单位。参数 option 说明是垂直方向 (MO_VERT,3) 还是水平方向 (MO_HORIZ,0)。low 指的是左上角，high 指的是右下角。

(3) 函数 mospeed 设置鼠标器在垂直和水平方向上的灵敏度。正是通过这个函数，使鼠标器在平板上的移动与鼠光标在屏幕上的移动联系起来。鼠标器的移动是以米基 (miskeys) 为单位，1 米基等于 0.005 英寸。鼠光标的移动是以像素为单位。这个函数中，参数是 8 个像素所对应的米基数，范围是从 1～32767。参数值越小，鼠光标反应越快，有利于快速移动。参数值越大，鼠光标移动越慢，有利于对鼠标器的细微移动提供更有效的控制。缺省灵敏度在垂直方向上是每 8 个像素 16 米基，在水平方向是每 8 个像素 8 米基。

(4) 函数 mojump 设置一个阈值。阈值的单位是每秒米基数，缺省值是每秒 64 米基。当鼠标器移动的速度低于这个阈值时，鼠光标以正常灵敏度移动。当鼠标器移动的速度超过这个阈值时，鼠光标则以更高的灵敏度移动。这就有点儿象正反馈似的，使鼠光标能更容易地移过较大距离。灵敏度提高多少与鼠标驱动程序有关，对于 Microsoft 的鼠标，灵敏度增加一倍。

(5) 函数 mogetmov 报告自上次查询以来鼠标器在垂直和水平方向移过的物理距

离，单位是米基，返回的值带正、负号，正值表示向右下方移动，负值表示向左上方移动。

3.3.4 鼠光标的形状和开关控制

属于这一类的函数有下列几个:

```
int far cdecl  moavoid (unsigned u__row,unsigned u__col,
                        unsigned l__row,unsigned l__col);
int cdecl  mograph (const unsigned *pmasks,int hot__row,int hot__col);
int cdecl  mohard (int high,int low);
int far cdecl  mohide (int option);
int cdecl  mosoft (unsigned screen__mask,unsigned cursor__mask);
```

当用键盘作为输入设备时，屏幕上有一个光标，以表示当前位置。当用鼠标作为输入设备时，也应该有一个类似的光标作为辅助。字符方式下和图形方式下的鼠光标可以不一样。与键盘不同的是， 鼠光标是软光标，不是由硬件实现的。在 Microsoft 的鼠标驱动程序中，缺省的图形鼠光标是一个箭头，缺省的文本鼠光标是一个亮块。Turbo C Tools 中提供的函数可以让程序员定义他的图形鼠光标和文本鼠光标的形状，并在文本方式下可以用键盘硬光标来代替软光标。

(1) 函数 mosoft 设置文本鼠光标的形状。这个形状是由三个值经过逻辑运算形成的。第 1 个值是屏幕上当前光标位置下那个字符的显示属性 (高字节) 和字符编码 (低字节)。第 2 个值是参数 screen__mask 所指定的 16 位屏幕屏蔽值。第 3 个值是参数 cursor__mask 所指定的 16 位光标屏蔽值。运算顺序是: 第 1 个值和第 2 个值逻辑“与”，“与”后的结果再与第 3 个值逻辑“异或”，这样得到的 16 位值再当作显示属性和字符编码在屏幕上显示出来，这就是文本鼠光标。如:

mosoft (0X77ff, 0X7700);	字符编码不变，但颜色反转
mosoft (0X0000, 0X0f18);	高亮度白色向上箭头，因向上箭头的ASCII码是 0X18

(2) 函数 mograph 设置图形鼠光标的形状。这个值也由三个值经过逻辑运算形成。因为是在图形方式下，故三个值不是 16 位的数，而是 16 个 16 位的数，最后的图形鼠光标是由 16＊16 个像素构成的。第 1 个值也是屏幕数据，第 2 个和第 3 个值来自参数 pmasks 所指向的 32 个无符号整数构成的数组。前 16 个整数对应于第 2 个值，后 16 个整数对应于第 3 个值。三个值的逻辑运算顺序也与函数 mosoft 一样。参数 hot__row 和 hot__col 指明图形鼠光标″热点″的行列号。因为鼠光标是一个 16＊16 的块，而鼠光标的坐标却是以点为单位计算的，所以只有鼠光标中的某一个点是代表它的坐标，这个点就称为″热点″。当鼠光标为一个箭头时，通常指定箭头尖所在的点为″热点″。热点的行列号是相对于鼠光标块左上角的行列号，其范围可以从−16～+16，负值指的是鼠光标块的左上方。

(3) 函数 mohard 用显示适配器的闪动硬件光标代替软件产生的文本方式鼠光标。两个参数是说明高低扫描行值，范围是从 0～13。

(4) 函数 mohide 根据参数 option 是 1 (MO__HIDE) 还是 0 (MO__SHOW) 使内部鼠光标显示计数器减 1 或加 1。鼠标驱动程序通过内部鼠光标显示计数器来决定鼠光标是否可见，其范围是从 −32768～0。当计数器为 0 时，光标可见。当计数器小于 0 时，光标不可见。初始化以后计数器的值为 −1 ，所以不可见。这个函数的功能类似于上一节 mouse 工具包中的 mouse__on 和 mouse__off 函数，用于对屏幕输出前先临时关闭鼠光标，输出结束后，再显示鼠光标。

(5) 函数 moavoid 定义屏幕上的一个矩形区域。如果鼠光标处于可见状态而进入该区域，则该函数使鼠光标不可见。如果鼠光标本来就处于不可见状态，则该函数不起作用。

3.3.5 对鼠标硬件中断的处理

属于这一类的主要是如下一个:

```
int cdecl  mohandlr (PMOHANDLER pfunc,unsigned call__mask,char * pstack,
                     int stksize,int option);
```

参数 call__mask 规定哪些事件会引起硬件中断，可选的事件有如下一些。

符号常数	值	意义
MO__MOVE	0X0001	移动鼠标
MO__L__PRESS	0X0002	按下左按钮
MO__L__RELEASE	0X0004	释放左按钮
MO__R__PRESS	0X0008	按下右按钮
MO__R__RELEASE	0X0010	释放右按钮
MO__M__PRESS	0X0020	按下中按钮
MO__M__RELEASE	0X0040	释放中按钮

参数 pfunc 是一个函数名，说明当产生鼠标硬件中断时，应使用哪一个函数作为中断处理程序。对这个函数有一些特殊要求，它应接受指向结构变量 ALLREG 的指针作为参数。关于结构 ALLREG，可参看第 13 章。

参数 pstack 和 stksize 说明堆栈的起始地址和堆栈的大小，这个堆栈是供中断处理程序使用的。

参数 option 可取值 MO__INSTALL (0) 或 MO__REMOVE (1)，表示此次调用的目的是安装还是撤消鼠标硬件中断处理程序。

这个函数的使用比较困难，牵涉到较深的知识。读者应先仔细读一下第 13 章的内容，以便于理解这个函数。这里只简单地列出一个示意程序，以解释大致的过程.

```
#include <stdio.h>
#include <bmouse.h>
#include <bvideo.h>
#include <butil.h>
static char handler__stack[2000];
```

```
void char myhandler (const ALLREG * );
:
:
mohandlr (myhandler,MO_MOVE,handler_stack,sizeof (handler_stack) ,
        MO_INSTALL);
:
:
void cdecl myhandler (const ALLREG * preg)
{
    char buffer[80];
    static unsigned long calls = 0L;
    sprintf (buffer,"calls = %10lu", ++calls);
    vidspmsg (4,60,15,0,buffer);
    sprintf (buffer,"row=%2d,col=%2d",preg->dx.x / 8,preg->cx.x / 8);
    vidspmsg (5,60,15,0,buffer);
}
```

第4章　文本屏幕输出和文本窗口

4.1 概　述

Turbo C 中提供了许多用于文本屏幕输出的函数。为便于理解，可以把文本显示输出的函数按表 4–1 分类:

表 4–1　文本显示输出函数的分类

	基于 DOS		基于 BIOS	基于硬件
	stdio.h 普通文件级	stdio.h 标准文件级	conio.h 控制台级	conio.h 控制台级
显示一个字符	fputc	putchar	putch	putch
显示一个字符串	fputs	puts	cputs	cputs
显示带格式的字符串	fprintf	printf	cprintf	cprintf

stdio.h 中普通文件级函数与其说是为屏幕输出设计的，倒不如说是为文件输出设计的，在它们的调用参数中必须说明文件名。如果想用于显示输出,则应使用文件名 stdout。标准文件级与普通文件级是类似的，也是文件输出，但这几个函数是专用于 DOS 的标准输出设备(一般为显示器)的，在调用参数中不需要再指定文件名，它们的输出可以在运行时重定向到一个普通文件上去。因为 stdio.h 中的文件级函数是面向文件的，自然不能利用显示屏幕的许多独特的特性，不能有效地进行交互式输入／输出。有关这些函数的细节详见第 6 章的基本文件处理。

包含在 conio.h 中的显示输出函数是专门为屏幕处理而设计的，但它们的输出不可能重定向到一个文件。conio.h 中的函数在使用时又分两种情况。第 1 种情况是，这些 C 函数又使用了 DOS 面向显示输出的系统调用，而这些系统调用一般又都是通过调用 BIOS 中的中断处理程序来工作的。简言之,这些 C 函数是通过 BIOS 工作的，速度尽管比文件级快，但仍然较慢。第 2 种情况是，这些 C 函数不使用 DOS 的系统调用，而是直接对显示缓冲区进行寻址，直接对硬件编程，以达到可能的最高速度。基于 BIOS 还是基于硬件，从函数名上是看不出来的，而是由一个 Turbo C 的变量进行控制。若变量 directvideo＝1，则基于硬件，否则基于 BIOS。一般情况下，隐含的是 directvideo＝1。如果程序中使用的是控制台级显示输出函数，不通过 DOS 输出，则会获得更多的控制和更快的速度，但却失去了与某些软件和硬件的兼容性。这样的程序在运行时可能出现一些不正常现象，因为 DOS 不能控制输出，不能为其它程序进一步使用这个输出结果而进行

处理，特别是文本信息不可能重定向到其它文件或设备。

例如，如果在程序中使用控制台级输出，虽然<Shift-Prtsc>键仍可以打印出该程序执行过程中产生的某一幅屏幕的硬拷贝，但却不能重定向到一份文件上去，故也不能进行进一步的修改和编辑。为打印出该程序产生的多幅屏幕的硬拷贝，最直接的办法就是利用<Ctrl-Prtsc>键，但是<Ctrl-Prtsc>是 DOS 的功能，不能用于控制台级函数产生的显示输出。如果要用这个方法，则在编写程序时主要应使用文件级显示输出函数，而且<Ctrl-Prtsc>方法在 Turbo C 的集成软件开发环境下也不能工作，必须先编译，接着退出到 DOS，按<Ctrl-Prtsc>键，再运行那个 EXE 文件，这样逐步去做，才能得到所希望的结果。

与键盘、鼠标等任何一种设备相比，程序员在编写显示输出的程序时,都会经常直接调用 DOS 的系统服务，甚至直接对硬件编程。例如，你可能想关掉显示,可能想直接往显示缓冲区中写内容。这方面的内容留待第 12 章再述,本章只讲述通过 C 函数就能够进行的处理。

除了显示一个字符或一个字符串外,需要解决的有关显示的问题还很多。如:

· 往一个指定的窗口中显示;
· 清除屏幕的一部分或全部;
· 插入和删除屏幕上的某些显示行;
· 读取和设置光标位置;
· 设置显示卡的文本／图形方式;
· 设置文本的显示属性，如颜色，下划线，闪烁;
· 读取有关的信息;
· 在各窗口之间以及窗口与内存之间移动文本。

为了处理这许多问题，Turbo C 中还提供了许多其它函数，这些函数也将在本章中讨论。

虽然 Turbo C 的函数已经为显示输出提供了很方便的控制,但它们仍然不能提供方便有效的交互式输入／输出。例如:

· 不能对屏幕的某一部分只处理它的显示属性而不管它的内容;
· 窗口处理很简单，很初级;
· 不能处理多个显示页，即使显示卡已提供了这样的性能。

Turbo C Tools 提供了解决这些问题的办法，并且还增添了许多别的特性,这些将在 4.4 节中叙述，有关窗口的部分将在 4.5 节中叙述。像 Turbo C 中的 conio 级函数一样，Turbo C Tools 中的函数也主要通过 BIOS 来显示输出，有时甚至还直接存取显示缓冲区，而不采用高级的 DOS 显示输出服务。

4.2 Turbo C 的文本屏幕处理

这一节主要讲述 Turbo C 的 conio 级文本屏幕处理函数，这些函数除了提供最基本的字符／字符串输出外，还提供了开设窗口、光标定位、设置颜色、设置方式、直接显示缓冲区存取等许多功能。但 Turbo C 的屏幕输出还有些不足，尤其窗口功能很弱，

Turbo C Tools 在这些方面有很大的改进，这将在下两节中叙述。

4.2.1 文本输出与操作

4.2.1.1 TTY 输出规则

所有文本输出最终毕竟要输往某个设备，这些设备总是需要一些控制码。在输出的文本中，如果含有这些控制码,则将起相应的控制作用，而不被当作输出的文本。这就是说在文本和设备控制码之间总是有一点儿重叠。对于 Turbo C 的控制台级屏幕输出，有 5 个控制码。

ASCII 码	名称	控制作用
0	NULL	没有作用
7	BEL	响铃
8	BS	光标左移 1 列
10	LF	光标先移到第 1 列(回车)，然后移到下 1 行。如果需要，屏幕上滚 1 行，使最后 1 行为空行

每输出 1 个字符后，光标就向右移 1 格。如果右边已没有位置供输出了，则引起的反应就像输出了 1 个<LF>控制码似的。这种输出规则就叫做 TTY 规则，亦称为电传打字机规则。

4.2.1.2 输出文本

Turbo C 提供了下列 3 个以 TTY 方式输出文本的控制台函数:

```
int  _Cdecl  cprintf(const char *format, ...);
int  _Cdecl  cputs(const char *str);
int  _Cdecl  putch(int c);
```

像大家都很熟悉的 printf 函数一样，cprintf 的参数个数也是可变的。应该注意的是，根据 TTY 规则，ASCII 码 0 是没有什么控制作用的，但在函数 cputs 和 cprintf 中，ASCII 码 0 是当作字符串终止符。既然符合 TTY 规则，3 个函数的输出都是从当前光标位置开始，从左到右,从上到下进行。即除了上述的 5 个控制码以外，唯一能引起光标定位的就是往屏幕(窗口)的最右边写字符。

4.2.1.3 对屏幕内容和光标的操作

为了对已显示在屏幕上的文本和光标进行操作，Turbo C 提供了下列 6 个函数:

```
void  _Cdecl  clrscr(void);
void  _Cdecl  clreol(void);
void  _Cdecl  delline(void);
void  _Cdecl  gotoxy(int x, int y);
```

```
void   __Cdecl   insline(void);
int    __Cdecl   movetext(int left, int top, int right,
                  int bottom,int destleft, int desttop);
```

这些函数的物理意义从函数名上就可以了解得很清楚，分别是:清除屏幕／窗口,从光标处清除到本行尾，删除1行,移动光标到指定位置，插入1行,把屏幕内某一矩形块的内容移到另一矩形块内。在使用这几个函数时，应注意如下几点。第1，前5个函数都感觉窗口的存在，它们的行为都在当前窗口内，它们的坐标都是相对于窗口的左上角。但最后一个函数 movetext 忽略了窗口，它的坐标是相对于屏幕的左上角,是绝对坐标。第2，文本方式下,坐标原点在左上角，原点的坐标值是(1,1)而不是(0,0)，X 是列坐标，从左到右，Y 是行坐标，从上到下。第3，如果坐标(X,Y)的值越过边界，则函数 gotoxy 不起作用。第4，函数 clrscr 会把光标移到左上角，函数 gotoxy 把光标移到指定位置，其它4个函数不影响当前光标位置。

4.2.1.4 *屏幕与内存之间文本的移动*

下列两个函数分别从屏幕某一矩形区域内取文本到内存中，或做反方向的传输。

```
int gettext(int left,int top,int right,int bottom,void *destin)
int puttext(int left,int top,int right,int bottom,void *source)
```

这两个函数的坐标都是绝对坐标，不是相对于窗口的。大家知道,屏幕上显示的每一个字符在显示缓存中实际上是由一个 ASCII 码字节和一个属性字节组成的，这两个函数采取直接存取显示缓存的办法，来回传送的是这两个字节的内容，按从左到右从上到下的顺序。

如果显示卡是 IBM 的 CGA 及部分的兼容 CGA，由于设计不当，在直接存取显示缓存时会出现“雪花”。解决这个问题的办法只能是：仅当显示器的扫描线处在从右到左、从下到上的回扫过程中时才往显示缓存中写。但要判断什么时候处于回扫，需要直接存取一个口地址，这太麻烦了。因此，许多程序员写的直接存取显示缓存的程序，在 CGA 上往往有“雪花”。如果在用 TCINST.EXE 安装 Turbo C 时选择了合适的选择项，那么 Turbo C 编译出来的程序将总是等待回扫的时候才写。在 CGA 上滚动屏幕也会产生同样的问题。为了解决这个问题，BIOS 滚动程序会检查显示卡是否 CGA 卡。如果是，则在滚动时会把显示关掉，这就产生了滚动时特有的闪烁。对 IBM 的 CGA 来说,这种闪烁尽管也不舒服，但总比“雪花”强。对那些在设计上已克服了“雪花”毛病的兼容 CGA 卡，仅仅由于 BIOS 而在滚动时仍然产生闪烁则太可惜了，解决的办法是用第13章介绍的技术，代替 BIOS 中原来的屏幕输出服务程序。

4.2.2 窗口和显示方式控制

Turbo C 中提供了一个开设窗口的函数和一个设置显示方式的函数：

```
void window(int left,int top,int right,int bottom);
void textmode(int newmode);
```

函数 window 建立一个窗口,并把光标移到这个窗口的左上角,使这个窗口成为当前窗口，所有随后的控制台级函数，包括前面已讲过的 putch、cputs、cprintf、clrscr、clreol、delline、insline 和后面将要讲到的 wherex、wherey 等，所产生的输出和所用的坐标都是相对于这个窗口的。但随后的标准文件级函数产生的输出都无视这个窗口的存在。如果程序中从未用过 window 函数，缺省的窗口是整个屏幕。

函数 textmode 设置显示方式，文本显示方式只能是如下 6 种:

方式	符号名	适配器	列 * 行
-1	LASTMODE		
0	BW40	CGA	40 * 25
1	C40	CGA	40 * 25
2	BW80	CGA	80 * 25
3	C80	CGA	80 * 25
7	MONO	MDA	80 * 25

这个函数会清除屏幕,选择整个屏幕作为当前窗口，移动光标到屏幕的左上角。

4.2.3 属性控制

所谓属性控制，就是控制文本显示的颜色、亮度及其它一些特点。对于像颜色这样的显示属性，则控制其是与被显示的文本内容发生关系还是与屏幕上的显示区域发生关系。对于文本编辑，可能希望某一段文章具有某一种颜色，而不管这段文章显示在屏幕上的什么位置。对于填电子表格，可能希望某一个表项(即屏幕上的某一区域)具有某一种颜色，而不管往这个表项中填什么内容。Turbo C 的属性控制函数提供的仅仅是适合于前者的应用场合。

Turbo C 的属性控制函数有如下 6 个:

```
void lowvideo(void);
void highvideo(void);
void normvideo(void);
void textattr(int newattr);
void textbackground(int newcolor);
void textcolor(int newcolor);
```

前 3 个函数用来设置亮度，分别置为低亮度、高亮度、程序启动时的亮度。后 3 个函数用来设置颜色，分别设置前景(字符)、背景、前景和背景。在文本方式下是用 1 个字节来描述显示属性的，其各位分配如表 4-2 所示:

表 4-2　文本显示属性

前景 / 背景	背　景			前　景			
BLINK 闪烁	R 红	G 绿	B 蓝	I 加亮	R 红	G 绿	B 蓝

背景占 3 位，故函数 textbackground 的参数值范围只能是 0～7 。函数 textcolor 的参数范围是 0～15，以及 0～15 再加 128。这是因为，闪烁特性也归于前景。对于每一个颜色号，Turbo C 在 conio.h 中都定义一个符号名称,其对应关系如下：

```
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
```

函数 textattr 是同时设置前景和背景，但其前景和背景的组合应由程序员完成，而不是由函数自动完成。所以,欲设置在蓝色背景上显示闪烁的黄色字符,其调用办法应该是：

```
textattr(YELLOW + (BLUE<<4) + BLINK);
```

值得注意的是，属性控制函数只是如此设置属性字节中的 8 位，但对这 8 位的最终解释却是由显示卡完成的,可能与上表中列出的 Turbo C 的解释不完全相同。最明显的例子就是单色显示器，因为此时没有彩色，所以显示卡可能把属性字节解释为灰度等级。在不具备灰度控制的单色显示系统中，最常见的对属性字节的解释(也就是 IBM 单色显示器的解释)如表 4-3 所示。

从表 4-3 中可以看出，在单色显示系统上，主要差别是把蓝色解释为下划线。

对于 CGA，可能属性字节中的位 7 不是当作闪烁，而是当作背景加亮来解释，这样就可以使背景色也增加到 16 种。但这种解释需要往口地址 0X3D8 中写 1 个字节，而这个口地址中的其余各位有别的控制作用。在写时，必须保持其它位不变，这就要求在写之前先把这个口地址的内容读出来。但是,在 CGA 中这个口地址是只写的，无法读出，故无法做这件事。在 EGA 中，这个口地址内容可以读出来，所以在 Turbo C Tools 中提供了这样一个函数 scblink。这个函数在下一节中介绍。

在 CGA 中，16 种颜色是固定的。在 EGA 中，16 种颜色是从 256 种颜色中挑选出来的，Turbo C Tools 中提供了这样的挑选函数。这个函数也将在下一节中介绍。

表 4-3 单色显示属性

	背景 RGB	前景 RGB
不可见	BLANK 000	0000 BLANK
下划线	BLANK 000	0001 BLUE
正常显示(lowvideo)	BLANK 000	0111 LIGHTGRAY
高亮度下划线	BLANK 000	1001 LIGHTBLUE
高亮度(highvideo)	BLANK 000	1111 WHITE
反显	LIGHTGRAY 111	0000 BLANK

所有的属性控制函数不影响已在屏幕上显示的文本属性，只影响随后输出的文本属性。而且，只有控制台级函数产生的文本输出才受属性控制函数的影响.

4.2.4 状态查询

Turbo C 中提供了下列几个状态查询函数：

```
void gettextinfo(struct text_info *r);
int  wherex(void);
int  wherey(void);
int  biosequip(void);
```

在调用 gettextinfo 函数时，必须分配足够的内存来存放结构 text_info。这个结构的定义在 conio.h 头文件中，现摘录如下：

```
struct text_info {
      unsigned char  winleft;
      unsigned char  wintop;
      unsigned char  winright;
      unsigned char  winbottom;
      unsigned char  attribute;
      unsigned char  normattr;
      unsigned char  currmode;
      unsigned char  screenhight;
      unsigned char  screenwidth;
      unsigned char  curx;
      unsigned char  cury;
}
```

如果不需要知道这么多信息，而只需知道光标的位置，则可用 wherex 和 wherey 函数。这两个函数返回的值以及结构 text_info 中字段 curx 和 cury 的值，都是相对于当前窗口的坐标值。所谓相对，就带来这样一个问题，如果用 window 函数改变了当前窗口的位置和尺寸，那么，究竟是保持光标不动而使 curx 和 cury 的值(以及 wherex 和 wherey 返回的值)发生变化，还是移动光标的位置适应新的窗口而保持这些坐标值不变呢

? Turbo C 采取的是后者，它物理地移动光标在屏幕上的位置，但 wherex 和 wherey 的返回值不变。

函数 biosequip 的说明在头文件 bios.h 中，它返回的整数的位 4 和位 5 表明了在引导时用的是哪一种显示方式，如下表所示：

位 4、位 5 中的值	方式	Turbo C 符号名
1	0	BW40
2	2	BW80
3	7	MONO

函数 detectgraph 的说明在头文件 graphics.h 中，通过这个函数可以检测到用的是什么显示驱动程序／显示卡，可以设置各种图形显示方式。这个函数将在第 14 章中讨论。

除了 MDA 和 CGA 外，现在市面上经常使用的是 EGA、VGA，还有大力神(Hercules)单色图形卡。遗憾的是，无论是 Turbo C (直至 2.0 版)还是 Turbo C tools 都不完全支持 EGA 和 Hercules 卡，更谈不上完全支持 VGA 了。

EGA 和 VGA 是极其复杂的显示设备，它的能力随着显示卡上安装的显示缓存的容量不同而有所不同。EGA 和 VGA 最吸引人的地方是它能在与 IBM 单色、彩色、加强彩色显示器上产生单色或多色中分辨率和高分辨率的图形，所有 MDA 和 CGA 的显示方式在 EGA 和 VGA 上都模拟实现并且被加强了。在 EGA 下，文本方式的加强就可以达到如下的指标：

· 80 * 43 的文本屏幕；
· 可以往 ASCII / IBM 字符集中增加 256 个用户定义的字符；
· 可以把文本屏幕分成上下两个独立的部分；
· 可以有多个文本页。

在 Turbo C 中，不能通过现有的函数利用这些加强的特性。例如，尽管屏幕可以大到 80 * 43，但通过 window 函数设置的窗口却最大只能达到 80 * 25。在 Turbo C Tools 中，已部分地支持这些加强特性,这将在下一节中介绍。

Hercules 单色图形卡比较简单,它的主要特点是在 IBM 单色显示器上可以显示中分辨率的图形。它的图形显示方式与 CGA 根本不兼容，分辨率也不一样。但在文本方式下，Hercules 单色图形卡与 IBM 的 MDA 却没有多大差别，只是可以显示两页，而 IBM 的 MDA 只有一页。

实际上，几乎没有任何一份商品化的软件支持 Hercules 额外添加的一页，因为软件要得知机器上是否安装有 Hercules 单色图形卡是很困难的.

4.3 pop__up 文本窗口工具包

窗口是屏幕上的一块矩形区域，它与屏幕上的其余部分不发生关系，有它自己的光标，独立地进行存取，独立地进行滚动。从用户的角度看，窗口有两种。第 1 种是弹出式(pop__up)窗口，一个窗口突然弹出来，盖住屏幕上原来这一区域的内容。第 2 种是“瓷

砖"式(tiles)窗口，这种窗口把屏幕划分成单个的"铺面"，彼此不重叠。"弹出"式窗口很受欢迎，本节讨论的也是这种窗口。本节中的程序是用 Turbo C 屏幕支持函数写成的，由于不是直接存取显示缓存区，所以执行速度可能不够快，但很容易移植到将来的机器上去。如果是较高主频的 286／386 型微机，则速度还是相当快的。

Turbo C tools 比 Turbo C 提供了更多更强的窗口支持函数。如果利用 Turbo c tools 工具编写窗口应用软件，编写程序的难度就会大大减低，详见第 4.5 节。

4.3.1 窗口结构和窗口栈

为了制作"弹出式"窗口，必须以某种方式保留该窗口下屏幕区域的映像，以便在显示这个窗口随后又关闭这个窗口之后，可以恢复屏幕原来的形状。最好的办法就是定义一个结构，每个窗口都对应一个它自己的结构变量，在这个结构变量内记录下该窗口特有的信息。在本节的工具包中，窗口结构是这样定义的：

```
typedef struct winstruct {
      char  *name;
      void  *image;
      struct winstruct  *under,  *over;
      wincolors wc;
      char xul,yul,xlr,ylr,wd,ht;
      char xsave,ysave;
      enum windowtype wtype;
}windesc;
```

其中，窗口颜色结构 wincolors 和枚举 windowtype 的定义如下：

```
typedef struct wincolors__struct {
      char border__type;
      unsigned char border__color, text__color;
      unsigned char title__color,  hilite__color;
} wincolors;
enum windowtype {popup,tile};
```

在结构 wincolors 中，border__type 定义了边界类型，它的取值范围只能是 0 和非 0，分别表示无边界和有边界。border__color、text__color、title__color、hilite__color 分别表示边界颜色、文本颜色、标题颜色和着重颜色。关于颜色常数的定义请参考上一节。在下面的工具包中，一共定义了 5 组颜色，分别供彩色图形卡、正常显示的单色卡、反显的单色卡、错误信息显示、正常信息显示使用。如果需要，可以增加或修改这些颜色组。

枚举 windowtype 定义窗口类型，分弹出式(pop__up)和瓷砖式(tile)两种。

在实际应用中，屏幕上经常需要同时出现多个窗口，并要求这些窗口可以前后移动。为了做到这一点，必须知道当前哪一个窗口是可见的。当把这个窗口从屏幕上搬走后，必须能够退回到前一个窗口。这有点类似堆栈的操作。在上面那个窗口结构里，借助于 under 和 over 这两个指针，建立起一个双向链表，把所有窗口结构连接在一起。除某些类

型的窗口外，每个窗口都保留它下面的映像，因此必须记住窗口弹出来的顺序，以便当搬走某个窗口后，被保存的映像能正确地恢复。如果这个顺序弄错了，屏幕上显示的结果就会乱七八糟。为方便起见，把较早弹出来的一个窗口叫做“下面的窗口”或“前面的窗口”，较晚弹出来的一个窗口叫做“上面的窗口”或“后面的窗口”。“上面的窗口”不一定正好盖住“下面的窗口”，甚至都不一定重叠，而只是说它弹出来比较晚。

最下面的一个窗口就是整个屏幕，也叫做“基窗口”。

4.3.2 16个工具函数

在下面这个 popup 窗口工具包中，一共包含 16 个工具函数：

```
#define rmv__win(w)  view__win(w,0)
#define slct__win(w) view__win(w,1)
extern void      init__win(void);
extern windesc  *draw__win(int x,int y,int wd,int ht,char  *title,
                    enum windowtype wt, wincolors  *ws);
extern void      view__win(windesc  *this, int move__to__top);
extern void      clr__win(void);
extern void      draw__box(int xul,int yul,int xlr,int ylr,
                      int btype,int attr);
extern void      centerstr(int xul, int yul, int xlr, int ylr,
                        char  *s, unsigned char a);
extern void      mprintf(char  *fmt,...);
extern void      prtfstr(int x,int y,char  *fmt,unsigned char attr,int wd,...);
extern void      swap__image(windesc  *w);
```

(1) 函数 init__win 是对窗口栈进行初始化。它构造一个基窗口“base__win”，并调用 Turbo C 的 gettextinfo 函数，取得屏幕的当前信息，把这些信息填入基窗口结构中。这个基窗口成了栈中第 1 个窗口，它既是栈顶，也是当前窗口。在构造任何别的工具函数之前，必须先调用 init__win 函数。

(2) 函数 draw__win 在指定的屏幕位置建立一个指定大小的新窗口。应该注意，所有窗口坐标都包括边界，因为 Turbo C 的窗口函数不知道边界这回事。窗口左上角坐标参数(X 或 Y)除了可以是整数值外，还可是符号 CTRWIN，意味着窗口在水平方向(或垂直方向)的中心与屏幕的相应中心是重合的。如果 X 和 Y 都等于 CTRWIN，则窗口落在屏幕的正中央。函数的后 3 个参数分别指定窗口的标题、窗口的类型和窗口的颜色。关于窗口的类型，还需要补充几句。前面已经说过，类型分为“弹出”型和“瓷砖”型。如果是“弹出”型，窗口下的映像就会被保留，如果是“瓷砖”型，窗口下的映像不被保留。这样加以区分，主要是为了节省内存和加快执行速度。如果某一个窗口占据了很大一块屏幕面积，而这个窗口一旦显示出来就再也不搬走，那就没有理由一定要去保留它下面的映像了。两类窗口的其它处理也有些差别，下面还会谈到。这个函数还做了许多其它工作，如检查坐标是否有效，把参数存入新的窗口结构中，保留窗口下的映像，在窗口周围画一个边框，把新窗口放到窗口栈的栈顶，等等。

(3) 函数 view__win 根据参数 select＝1 还是 select＝0，用来选择或撤消一个窗口。

函数 slet__mim 和 rmv__win 是两个宏，分别调用 view__win 函数，只实现 view__win 的选择窗口或撤消窗口功能。这个函数的工作基本上由三步组成。第 1 步，如果被操作(无论是选择还是撤消)的窗口是“弹出窗口”，则把这个窗口从屏幕上搬走。第 2 步，把由于被操作窗口搬走而被断开的窗口链重新连接起来。第 3 步，根据是选择还是撤消进行不同的操作。如果是选择该窗口，则把该窗口移到栈顶，即把该窗口搬回到最上面，并使其成为当前窗口。如果是撤消该窗口，则使原栈顶的窗口成为当前窗口，并注销该窗口。改变当前窗口的工作是通过调用一个内部函数 chg__win 来完成的。chg__win 函数保留原当前窗口的光标位置，调用 Turbo C 的 window、textattr、gotoxy 等函数去制做窗口、设置窗口显示属性、设置光标位置，以后就可以在这个窗口上进行操作了。

上面三步最重要的核心就是把被操作窗口（不论是选择还是撤消）先移到栈顶。这样做的目的，是为了防止往其它可能被盖住的窗口中写内容。本节介绍的窗口工具包不够复杂，不够智能，它不知道一个窗口是否盖住了另一个窗口，对各窗口间的关系，它唯一知道的只是各窗口被弹出的次序。 因此，每选择一个窗口，必须修改窗口栈，把新选择的窗口移到栈顶。具体做法是先从窗口栈顶开始，往下直到(包括)被操作的那个窗口为止，依次把窗口结构中保存的映像与屏幕上该窗口的内容进行交换。每交换一次，给用户的感觉就是从屏幕上搬开了一个窗口。然后再从被操作窗口的上一个窗口开始，往上直到栈顶为止，再依次把窗口结构中保存的映像与该窗口在屏幕上将要占据区域的内容进行交换。每交换一次，给用户的感觉就是在屏幕上又回来了一个窗口。

上述的第 1 步其所以对“瓷砖”式窗口没有必要，其理由有两个。第 1，“瓷砖”式窗口的结构中没有映像保留区，也就谈不上进行交换。第 2，既然“瓷砖”式窗口不与别的窗口重叠，在它上面的各个窗口也就没有搬走又搬回来的必要。由此可以引伸出两个重要结论。第 1，如果知道一个窗口以后可能被另一个窗口覆盖，哪怕只是一部分，这个窗口就不能是“瓷砖”式窗口，否则写往这个窗口的内容可能写到它上面的窗口中去，产生不正确的效果。第 2，如果所有窗口都是“瓷砖”式窗口，那就不需要保留映像和恢复映像，不需要进行交换，在各窗口之间跳来跳去将是快捷的。从某种意义上讲，“瓷砖”式窗口是半永久性窗口，而“弹出”式窗口是临时性窗口。

(4) 上面讲的交换工作都是由函数 swap__image 完成的。这个函数调用了 Turbo C 的 gettext 和 puttext 函数保留和恢复屏幕上该窗口的内容。上面已经说过，swap__image 函数是个两位开关似的函数，调用它一次，该窗口会从屏幕上消失掉。再调用它一次，该窗口又会在屏幕上出现。这对于临时性地把一个窗口搬走而后又搬回来是非常方便的，用不着重新构造窗口内的文本，当然，这样做的条件是被操作窗口必须在窗口栈的栈顶。

为了进行变换，除了窗口结构中已有的保存映像用的内存外，函数 swap__image 还临时申请了一块同样大小的内存，以便通过 Turbo C 的 gettext 和 puttext 函数进行整块的交换。这种办法并非很好，需要较多的内存，且由于 gettext 和 puttext 不是直接存取显示缓存区，所以速度也较慢，比较好的办法是一次一个字符地进行交换，且通过存取显示缓存去实现。

(5) 函数 mprintf、prtfstr、centerstr、clr__win 和 draw__box 是 5 个往窗口中写内容的函数，它们比 Turbo C 的 cprintf、putch、cputs、clrscr 函数更高级且功能更强一些。像 Turbo C 的同类函数一样，这 5 个函数也总是往当前窗口中写，但它们在更新屏

幕时关闭鼠标，而 Turbo C 函数不关闭。这就是为什么要提供 mprintf 函数的原因，尽管它做的事情与 cprintf 函数一样，实际上它是调用 cprintf 去实现的。如果使用了鼠标，则总是应该使用 mprintf 而不使用 printf。函数 mprintf 和 prtstr 都接受可变数目的参数，关于这方面的问题可参考第 1 章。它们的格式化输出都不允许超过 255 个字符。

(6)函数 prtstr 在当前窗口显示一个格式化的字符串，它与 Turbo C 的 cprintf 函数类似，只是不支持自动拐弯。如果字符串太长，窗口中放不下，则字符串将被截短。 如果参数 attr=0 ，则显示属性不变，仍以窗口上原来的颜色显示，否则以所指定的颜色显示。attr 是属性字节，既包括前景也包括背景。除正常输出外，prtstr 函数还有如下几个用途:

第 1，强调某一个菜单条。这是因为，如果所有其它菜单条都是以 attr=0 显示的，而只有某一个菜单条是以 attr 为某一非 0 值显示的，则这个菜单条的显示颜色将与众不同，从而被强调。例如，下面这条语句就是以蓝色在第 10 列第 5 行显示"Hello world"11 个字符。

```
prtfstr(10,5,"Hello world",1,11);
```

第 2，利用它的截短特性。这个函数中牵涉到三个宽度数值：参数 wd 给定的宽度，欲输出的字符串的实际宽度，窗口的宽度(还需减去起始列参数 X)。最后在窗口上输出的字符串的实际长度是由这三个宽度数中的最小一个来确定的，以防产生滚动。例如：

```
w=draw__win(1,1,20,6,"my win",popup,&defcolors);
prtfstr(15,1,"this text will be truncated",0,80);
```

只在窗口的第 15 列第 1 行显示出"this t"这样 6 个字符，因为窗口只能从第 1 列到第 20 列，而起始显示列是第 15 列，从第 15～第 20 列只有 6 个字符。截短看来是件很容易的事，实际上却不完全如此。因为 Turbo C 的 print 函数总是移动光标，如果通过 printf 函数在窗口的最后 1 列输出 1 个字符，那么输出以后，光标就会移到下一行，可能引起窗口滚动。这就意味着，用 Turbo C 的 print 函数不可能实现只有 1 行高的窗口，因为一旦往最后 1 列送 1 个字符，窗口就会滚动，窗口内容就将看不见了。因为 Turbo C 不提供任何关掉滚动特性的函数，唯一可行的办法，就是通过 gettext 函数把窗口内容读入内存，修改以后再通过 puttext 函数送回到窗口上。正因为 gettext 和 puttext 函数不影响光标，所以还必须人为地修改光标位置。这两个函数比较慢，如果要提高速度，最好改为直接存取显示缓冲区。

第 3，填充。如果入口时宽度参数 wd 的值为负数，且欲显示的字符串只有 1 个字符，则将以这个字符填满 1 行，例如：

```
prtfstr(1,5,"c",0,-25);
```

将显示出 25 个字符"c"，除非被窗口宽度所限。

第 4，格式化输出。在参数 fmt 所指向的字符串中，可以有格式控制字符，被格式化的对象放在参数 wd 之后。例如：

```
prtfstr(1,5,"The mouse is at %d %d",0,80,mouse__text__x,mouse__text__y);
```

将在显示“The mouse is at ″之后，以十进制整数格式显示变量 mouse__text__x 和 mouse__text__y。又如：

```
prtfstr(1,5,"%-*.*s",112,25,25,25,"Menu choice #1");
```

将加亮一个 25 个字符宽的条，左对齐条内的字符串。这条语句用了 printf 函数的左对齐控制符“%-*.* s”，并把格式化字段的宽度和精度作为参数传给函数 prtstr 。应该注意，第一个“25”是 prtstr 函数本身的宽度参数，后两个“25”才是用于格式化字符串的。

(7) 函数 centerstr 以指定的属性在指定的矩形区域中心显示所指定的字符串，不能进行格式化输出。

(8) 函数 clr__win 清除当前窗口，与 Turbo C 的 clrscr 不同的是，它支持鼠光标。

(9) 函数 draw__box 是上一章在 mouse 工具包应用举例中已介绍的函数 my__box 的扩充。根据入口参数 btype 的值为 0、为 1 或为 2，分别不画框，或画一个单线框，或画一个双线框。为了在画右下角那个边框字符时不引起滚动，函数 draw__box 用了与前面已介绍的 prtstr 同样的办法，即用 gettext/puttext 的办法。

另外 7 个工具函数在第 4.3.3.3 节中讲述 pop__up 错误信息窗口时再介绍。

```
/*    popup.h    */
#define    CTRWIN    9999
typedef    struct    texel__struct {
     unsigned  char  ch;
     unsigned  char  attr;
} texel;
typedef struct wincolors__struct {
     char border__type;
     unsigned char border__color, text__color;
     unsigned char title__color,  hilite__color;
} wincolors;
enum windowtype {popup, tile};
typedef struct winstruct {
     char  *name;
     void  *image;
     struct winstruct  *under,  *over;
     wincolors wc;
     char xul,yul,xlr,ylr,wd,ht;
     char xsave,ysave;
     enum windowtype wtype;
}windesc;

extern      windesc  *base__win;
extern      windesc  *curr__win;
extern      wincolors   defcolors;
extern      wincolors   invcolors;
extern      wincolors   monocolors;
extern      wincolors   errcolors;
extern      wincolors   msgcolors;
```

```
#define rmv_win(w)  view_win(w,0)
#define slct_win(w) view_win(w,1)

extern void      init_win(void);
extern windesc  * draw_win(int x,int y,int wd,int ht,char  * title,
                        enum windowtype wt, wincolors  * ws);
extern void      view_win(windesc  * this, int move_to_top);
extern void      clr_win(void);
extern void      draw_box(int xul,int yul,int xlr,int ylr,int btype,int attr);
extern void      centerstr(int xul, int yul, int xlr, int ylr,char  * s, unsigned char a);
extern void      mprintf(char  * fmt,...);
extern void      prtfstr(int x,int y,char  * fmt,unsigned char attr,int wd,...);
extern void      swap_image(windesc  * w);

#define   SWRNF      0,1,__LINE__,__FILE__
#define   SERRF      0,2,__LINE__,__FILE__
#define   SMSGF      0,0,__LINE__,__FILE__
#define   FWRNF      1,1,__LINE__,__FILE__
#define   FERRF      1,2,__LINE__,__FILE__
#define   FMSGF      1,0,__LINE__,__FILE__

#define   SWRN       0,1,0,""
#define   SERR       0,2,0,""
#define   SMSG       0,0,0,""
#define   FWRN       1,1,0,""
#define   FERR       1,2,0,""
#define   FMSG       1,0,0,""

extern     int  errx;
extern     int  erry;
extern     windesc  * errw;
extern     int msgx;
extern     int msgy;
extern     windesc  * msgw;
extern     void numnewlines(char  * s, int  * n, int  * w);
extern     void popmsg(int x, int y, char  * msg, char  * title,
                   char soundout, wincolors  * wc);
extern     void reperr(int level, char  * msg);
extern     void repmsg(char  * msg);
extern     void sayerr(int ferr, int errflag, int lno,
              char  * pname, char  * fmt,...);
extern     void beep(void);
extern     unsigned int getkey(void);

/ *   popup.c   * /
#include   <stddef.h>
#include   <stdarg.h>
#include   <stdio.h>
#include   <stdlib.h>
#include   <alloc.h>
#include   <conio.h>
```

```
#include  <string.h>
#include  <process.h>
#include  "popup.h"
#include  "mouse.h"
#define  MAX(a,b)  ((a) > (b) ? (a) : (b))
#define  MIN(a,b)  ((a) < (b) ? (a) : (b))

windesc *base_win = NULL;
windesc *curr_win = NULL;

wincolors defcolors    = {1,  23,  30,  27,  94 };
wincolors invcolors    = {1, 112, 112, 112,  15 };
wincolors monocolors   = {1,   7,  15,  15, 112 };
wincolors errcolors    = {1,  79,  79,  79,   4 };
wincolors msgcolors    = {1,  47,  47,  46, 112 };

static windesc *top_win;
static void chg_win(windesc *this);
static windesc *make_window_node(void);
static windesc *push_window_node(void);
static void dispose_window_node(windesc *w);

void init_win(void)
{
     struct text_info ti;
     base_win = make_window_node( );
     gettextinfo(&ti);
     base_win->xul = ti.winleft - 1;
     base_win->xlr = ti.winright + 1;
     base_win->yul = ti.wintop - 1;
     base_win->ylr = ti.winbottom + 1;
     base_win->wd  = ti.screenwidth + 2;
     base_win->ht  = ti.screenheight +2;
     base_win->xsave = ti.curx;
     base_win->ysave = ti.cury;
     base_win->wc  = monocolors;
     base_win->wc.text_color = ti.attribute;
     base_win->name = "base";
     base_win->wtype = tile;
     base_win->wc.border_type = 0;
     top_win = base_win;
     curr_win = base_win;
}

windesc *draw_win(int x, int y, int wd, int ht, char *title,
                  enum windowtype wt, wincolors *wc)
{
     windesc *w;
     int xsave, ysave;
     mouse_off(1);
     w = push_window_node( );
```

```
        wd = MAX(wd,3);
        wd = MIN(wd,80);
        ht = MAX(ht,3);
        ht = MIN(ht,25);
        if (x == CTRWIN)  x = (80-wd) / 2;
        if (y == CTRWIN)  y = (25-ht) / 2;
        if (wc->border_type)  {
            x = MAX(x,1);
            y = MAX(y,1);
        }
        else  {
            x = MAX (x,0);
            y = MAX (y,0);
        }
        if  ((x+wd) > 80) x = 80-wd+1;
        if  ((y+ht) > 25) y = 25-ht+1;
        w->wd   = wd;
        w->ht   = ht;
        w->xul  = x;
        w->yul  = y;
        w->xlr  = x + w->wd - 1;
        w->ylr  = w->yul + w->ht - 1;
        w->wc  =  *wc;
        w->xsave  = 1;
        w->ysave  = 1;
        w->wtype  = wt;
        w->name   = strdup(title);
        if (wt == popup)  {
            w->image = calloc(wd * ht,2);
            swap_image(w);
        }
        if (curr_win == base_win)  {
            xsave = wherex( );
            ysave = wherey( );
        }
        else  {
            xsave = base_win->xsave;
            ysave = base_win->ysave;
        }
        chg_win(base_win);
        draw_box(w->xul,w->yul,w->xlr,w->ylr,w->wc.border_type,
                w->wc.border_color);
        centerstr(w->xul,w->yul,w->xlr,w->ylr,w->name,
                w->wc.title_color);
        gotoxy(xsave,ysave);
        chg_win(w);
        clr_win( );
        mouse_on(1);
        return w;
}
```

```
void view_win(windesc *this, int select)
{
    windesc *p;
    if (select && this == top_win) return;
    mouse_off(1);
    if (this->wtype == popup) {
        p = top_win;
        while (p !=this) {
            swap_image(p);
            p = p->under;
        }
        swap_image(this);
        p = this;
        while(p != top_win) {
            p = p->over;
            swap_image(p);
        }
    }
    if (this == top_win) {
        this->under->over = NULL;
        top_win = this->under;
    }
    else {
        this->under->over = this->over;
        this->over->under = this->under;
    }
    if (select) {
        top_win->over = this;
        this->under = top_win;
        top_win = this;
        swap_image(this);
        chg_win(this);
    }
    else {
        chg_win(top_win);
        dispose_window_node(this);
    }
    mouse_on(1);
}

static void chg_win(windesc *this)
{
    curr_win->xsave = wherex( );
    curr_win->ysave = wherey( );
    window(this->xul+1, this->yul+1, this->xlr-1, this->ylr-1);
    textattr(this->wc.text_color);
    gotoxy(this->xsave, this->ysave);
    curr_win = this;
}

void swap_image(windesc *w)
```

```
{
      int xstart, ystart, xfin, yfin;
      char * temp_image;
      unsigned int nbytes;
      if (w->wtype ==popup) {
           xstart = w->xul;
           ystart = w->yul;
           xfin  = w->xlr;
           yfin  = w->ylr;
           if (!w->wc.border_type) {
                xstart++;
                ystart++;
                xfin--;
                yfin--;
           }
           nbytes = (xfin-xstart+1) * (yfin-ystart+1) * 2;
           mouse_off(1);
           temp_image = malloc(nbytes);
           gettext(xstart, ystart, xfin, yfin, temp_image);
           puttext(xstart, ystart, xfin, yfin,(void * )w->image);
           memcpy(w->image, temp_image, nbytes);
           free(temp_image);
           mouse_on(1);
      }
}

void clr_win(void)
{
      mouse_off(1);
      textattr(curr_win->wc.text_color);
      clrscr( );
      mouse_on(1);
}

void mprintf(char * fmt,...)
{
      va_list arg_ptr;
      char t[255];
      va_start (arg_ptr,wd);
      vsprintf(t,fmt,arg_ptr);
      mouse_off(1);
      cprintf("%s", t);
      mouse_on(1);
      va_end(arg_ptr);
}

void prtfstr(int x,int y,char * fmt,unsigned char attr,int wd,...)
{
      va_list arg_ptr;
      char t[255];
      int len, i, n, xa, ya, fillflag;
```

```
    static texel line[80];
    va_start(arg_ptr,wd);
    vsprintf(t, fmt,arg_ptr);
    va_end(arg_ptr);
    n = abs(wd);
    n = MIN(n, curr_win->wd-x-1);
    xa = curr_win->xul + x;
    ya = curr_win->yul + y;
    len = MIN(strlen(t), n);
    t[len] = 0;
    if (len) {
        if (wd < 0 && (strlen(t) == 1)) {
            fillflag=1;
            len = n;
        }
        else {
            fillflag = 0;
        }
        mouse_off(1);
        gettext (xa, ya, xa+len-1, ya, line);
        for (i=0; i<len; i++) {
            if (fillflag) line[i].ch = *t;
                else line[i].ch = t[i];
            if (attr) line[i].attr = attr;
        }
        puttext(xa, ya, xa+len-1,ya, line);
        mouse_on(1);
        if (x+len == curr_win->wd-1) x--;
        gotoxy(x+len, y);
    }
    else {
        gotoxy(x,y);
    }
}

void centerstr(int xul, int yul, int xlr, int ylr,
               char *s, unsigned char a)
{
    int xs,ys,wd;
    if (*s != 0) {
        mouse_off(1);
        wd = xlr-xul+1;
        if ((xs = (wd-strlen(s)) / 2 + xul) < xul) xs = xul;
        if ((ys = (ylr-yul+1) / 2 + yul) < ul) ys = yul;
        prtfstr(xs, ys, s, a, wd);
        mouse_on(1);
    }
}

void draw_box(int xul,int yul,int xlr,int ylr,int btype,int attr)
{
```

```
    static int boxcar[2][6] = {
        {218,196,191,179,192,217},
        {201,205,187,186,200,188}
    };
    int i, hzchar, vtchar, oldattr;
    texel t;
    struct text_info ti;
    if (btype ) {
        mouse_off(1);
        gettextinfo(&ti);
        oldattr = ti.attribute;
        textatt(attr);
        hzchar = boxcar[btype-1][1];
        vtchar = boxcar[btype-1][3];
        gotoxy(xul+1,yul);
        for (i=xul+1; i<xlr; i++) putch(hzchar);
        gotoxy(xul+1,ylr);
        for (i=xul+1; i<xlr; i++) putch(hzchar);
        for (i=yul+1; i<ylr; i++) {
            gotoxy(xul,i);
            putch(vtchar);
            gotoxy(xlr,i);
            putch(vtchar);
        }
        gotoxy (xul,yul);
        putch(boxcar[btype-1][0]);
        gotoxy (xlr,yul);
        putch(boxcar[btype-1][2]);
        gotoxy (xul,ylr);
        putch(boxcar[btype-1][4]);
        gettext(xlr,ylr,xlr,ylr,&t);
        t.ch = boxcar[btype-1][5];
        t.attr = attr;
        puttext(xlr,ylr,xlr,ylr,&t);
        textattr(oldattr);
        mouse_on(1);
    }
}

static windesc *make_window_node(void)
{
    windesc *q;
    q = (windesc *) malloc (sizeof(windesc));
    q->image = NULL;
    q->under = NULL;
    q->over  = NULL;
    return q;
}

static windesc *push_window_node(void)
{
```

```
        windesc  * q;
        q = make__window__node( );
        top__win->over = q;
        q->under = top__win;
        top__win = q;
        return q;
}

static void dispose__window__node(windesc  * w)
{
        if (w != NULL) {
                if (w->wtype ==popup) free(w->image);
                free(w->name);
                free(w);
        }
}
```

4.3.3 工具包应用实例

4.3.3.1 用"瓷砖"式窗口制作菜单

下面这个示例程序首先在屏幕上弹出一个菜单，允许用户通过键盘(上移键和下移键)或鼠标在各菜单项之间移来移去，光标当前所在的项通过不同颜色而加以强调。菜单的形式如图 4–1 所示。

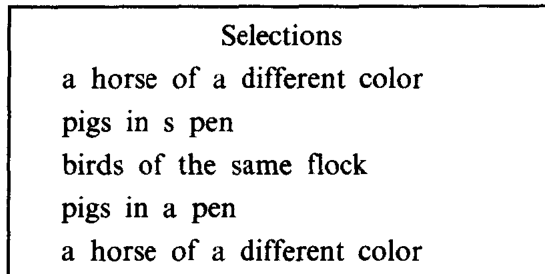

图 4–1 主菜单

按回车键(或按鼠标的左按钮)则选中光标所在的菜单项，同时被选中的项会被送往第 2 个窗口显示出来。然后又重新开始选择，直至用户按 ESC 键(或按鼠标的右按纽)退出为止。在经过几次选择后，第 2 个窗口的形式可能如图 4–2 所示：

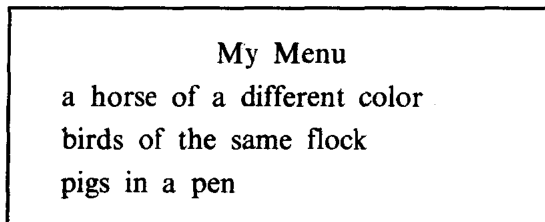

图 4–2 选中某一菜单项

因为两个窗口都是“瓷砖”式窗口,所以速度比较快。

从这个例子中也可以看到，如何做到在有鼠标时则用鼠标，无鼠标时也不影响程序运行。在调用函数 mouse_init 初始化鼠标时，告诉鼠标是可选择的(MOUSE_OPTIONAL)，这样即使鼠标不存在，函数也不会失败，而仅仅是忽略这一次的初始化请求。

下面就是这个例子的源程序:

```
#include  <bios.h>
#include  <conio.h>
#include  "popup.h"
#include  "mouse.h"

char  * menu__entries[ ] =  {
     "a  horse of a different color",
     "birds of the same flock",
     "pigs in a pen"
};
void disp__entry(windesc  * w,char  * entry[ ],int entryno,int hilite);

void main( )  {
     windesc  * w1,  * w2;
     int entryno  =  1,  i,  key;
     init__win( );
     init_mouse(MOUSE_OPTIONAL,MOUSE_TEXT_MODE,MOUSE_TEXT_MODE);
     defcolors  =  monocolors;
     w1  =  draw__win(CTRWIN,CTRWIN,30,5,"MY  MENU  ",tile,&defcolors);
     w2  =  draw__win(CTRWIN,1,50,7,"Selections",tile,&defcolors);
     slct__win(w1);
     for(i=0;  i<3;  i++)  prtfstr(1,i+1,menu__entries[i],0,80);
     do {
          do {
               disp__entry(w1,menu__entries,entryno,1);
               while(! (key  =  mouse__trigger(1)));
               disp__entry(w1,menu__entries,entryno,0);
               switch(key)  {
                    case UPKEY:
                         if (--entryno  <  1) entryno  =  3;
                         break;
                    case DOWNKEY:
                         if (++entryno  >  3) entryno  =  1;
                         break;
                    case LEFT__MOUSE__PRESS:
                         for (i  =  1;  i<=3;  i++)  {
                              if(mouse__in__box(0,w1->xul+1,w1->yul+i,
                                        w1->xlr-1,w1->yul+i))  {
                                   entryno  =  i;
                                   key  =  CRKEY;
                              }
```

```
                    }
                    break;
                case RIGHT_MOUSE_PRESS:
                    key=ESCKEY;
                    break;
                default: ;
            }
        } while ((key != CRKEY) && (key != ESCKEY));
        if (key == CRKEY) {
            slct_win(w2);
            mprintf("\r\n%s",menu_entries[entryno-1]);
            slct_win(w1);
        }
    }while (key != ESCKEY);
    rmv_win(w2);
    rmv_win(w1);
    mouse_reset( );
    clrscr( );
}

void disp_entry(windesc *w,char *entry[ ],int entryno,int hilite)
{
    int bar_color;
    if (hilite ) bar_color = w->wc.hilite_color;
        else    bar_color = w->wc.text_color;

    prtfstr(1,entryno,"%-*.*s",bar_color,80,
            w->wd-2,w->wd-2,entry[entryno-1]);
}
```

4.3.3.2 移动 pop_up 窗口

下面这个例子是说明如何在屏幕上选择窗口和在屏幕上移动窗口。如图 4-3 所示。它的工作如下:首先在屏幕上显示三个 pop_up 窗口,彼此有些重叠。每个窗口都有一个标

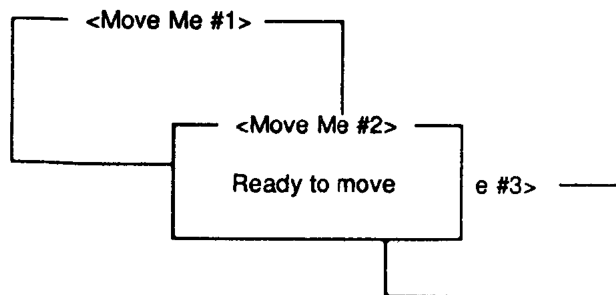

图 4-3 pop_up 窗口

题，标题中对窗口编号为 1～3。然后，等待用户的键盘或鼠标输入。若用户按下数字键 1

~3 中的某个数字，则选中相应的窗口，使它成为当前窗口。若用户按下某个光标移动键，则会引起当前窗口向相应方向移动，直至移到屏幕边界。若用户按下 ESC 键，则退出程序。若用户按任何其它键，则相应的字符会在当前窗口中显示出来。

这个例子是支持鼠标的。当鼠光标在某个窗口的四个边框上的某一处时，若揿一下鼠标的左按纽，则选中该窗口，使之成为当前窗口；若保持鼠标左按钮在按下状态而移动鼠标，则窗口就被“拖”着满屏幕移动。揿一下鼠标的右按钮，则退出该程序。

下面是这个例子的源程序：

```
#include <conio.h>
#include <bios.h>
#include "popup.h"
#include "mouse.h"
#define  MAX(a,b)  ((a) > (b) ? (a) : (b))
#define  MIN(a,b)  ((a) < (b) ? (a) : (b))

char msg[ ] = "Ready to move";
int  mouse_on_border(windesc * * w, int * i, int * xofs, int * yofs);
void move_curr_win(int x, int y);

void main( ) {
     windesc * w[3];
     int x, y, xofs, yofs, i;
     unsigned int k;
     init_win( );
     init_mouse(MOUSE_OPTIONAL, MOUSE_TEXT_MODE, MOUSE_TEXT_MODE);
     w[0] = draw_win(5,6,20,5,"\x11""Move Me #1\x10",popup,
                 &monocolors);
     w[1] = draw_win(7,9,20,6,"\x11""Move Me #2\x10",popup,
                 &monocolors);
     w[2] = draw_win(9,11,20,5,"\x11""Move Me #3\x10",popup,
                 &monocolors);
     mprintf(msg); / * print ready message in current window * /
     do {
          while (!(k = mouse_trigger(1)));
          if (k ==LEFT_MOUSE_PRESS) {
               if (mouse_on_border(w, &i, &xofs, &yofs)) {
                    clrscr( );
                    slct_win(w[i]);
                    mprintf(msg);
                    while(button_state( )) {
                         mouse_txt_posn(&x,&y);
                         move_curr_win(x-xofs,y-yofs);
                    }
               }
          }
          else {
               xofs = 0; yofs = 0;
               x = curr_win->xul;
```

```
            y = curr_win->yul;
            switch(k) {
                case UPKEY:
                    y--;
                    move_curr_win(x,y);
                    break;
                case DOWNKEY:
                    y++;
                    move_curr_win(x,y);
                    break;
                case LEFTKEY:
                    x--;
                    move_curr_win(x,y);
                    break;
                case RIGHTKEY:
                    x++;
                    move_curr_win(x,y);
                    break;

                case 0x0231:
                    clrscr( );
                    slct_win(w[0]);
                    mprintf(msg);
                    break;
                case 0x0332:
                    clrscr( );
                    slct_win(w[1]);
                    mprintf(msg);
                    break;
                case 0x0433:
                    clrscr( );
                    slct_win(w[2]);
                    mprintf(msg);
                    break;
                default:
                    if(k!=RIGHT_MOUSE_PRESS)
                        mprintf("%c",lo(k));
            }
        }
    } while (k!=ESCKEY && k!=RIGHT_MOUSE_PRESS);
    rmv_win(w[2]);
    rmv_win(w[1]);
    rmv_win(w[0]);
    mouse_reset( );
}

int mouse_on_border(windesc * * w,int * i,int * xofs,int * yofs)
{
    int x, y, j;
    mouse_txt_posn(&x, &y);
    for (j = 0; j<3; j++) {
```

```
            if ((x >=(w[j])->xul  && x <= (w[j])->xlr &&
                 y >=(w[j])->yul  && y <= (w[j])->ylr) &&
                (x ==(w[j])->xul  || x == (w[j])->xlr ||
                 y ==(w[j])->yul  || y == (w[j])->ylr))   {
                 *xofs = x - (w[j])->xul;
                 *yofs = y - (w[j])->yul;
                 *i =j;
                 return 1;
            }
        }
        return 0;
    }

    void move__curr__win(int x, int y)
    {
        int xsave, ysave;
        if (x !=curr__win->xul || y !=curr__win->yul) {
            xsave = wherex( );
            ysave = wherey( );
            swap__image(curr__win);
            curr__win->xul = x;
            curr__win->yul = y;
            curr__win->xul = MAX(curr__win->xul,1);
            curr__win->yul = MIN(curr__win->xul,81 - curr__win->wd);
            curr__win->xul = MAX(curr__win->yul,1);
            curr__win->yul = MIN(curr__win->yul,26 - curr__win->ht);
            curr__win->xlr = curr__win->xul + curr__win->wd -1;
            curr__win->ylr = curr__win->yul + curr__win->ht -1;
            swap__image(curr__win);
            window(curr__win->xul+1, curr__win->yul+1,
                   curr__win->xlr-1, curr__win->ylr-1);
            gotoxy(xsave, ysave);
        }
    }
```

4.3.3.3 pop__up 错误信息和正常信息

4.3.2 节已经介绍了 pop__up 窗口工具包 16 个函数中的 9 个函数。专门用来处理 pop__up 错误信息窗口和正常信息窗口的余下 7 个函数，组成一个子工具包。在这一小节，我们首先介绍这个子工具包的 7 个函数，然后列出子工具包的源程序。因为是子工具包，所以也没有相应的头文件，所需的内容已包括在 4.3.2 中那个头文件 popup.h 中了。最后，提供一个应用这个子工具包的例子。这个例子比前两个例子更复杂一些，但也更实用，这也是为什么抽出来单独作为一个子工具包的原因。

这个子工具包假设系统是按这样的方式工作的：它牵涉到两个窗口，分别用来处理错误信息和正常信息。这两个窗口仅当相应信息到来时才弹出到屏幕上，在处理过后可以消失，也可以继续留在屏幕上。子工具包的诸函数就是为了把各种信息送往相应的窗口，它们用到如下几个全局定义的变量和指针：

```
int            errx   = CTRWIN;
int            erry   = CTRWIN;
windesc        *errw  = NULL;
int            msgx   = CTRWIN;
int            msgy   = CTRWIN;
windesc        *msgw  = NULL;
```

窗口指针缺省为空，即 NULL。如果在相应的信息到来时，即相应的函数被调用时，候选的窗口指针为 NULL，则弹出一个临时窗口来显示这些信息，窗口在屏幕上的坐标位置分别由(errx,erry) (对错误信息)和(msgx,msgy)(对正常信息)来决定。这些坐标位置的缺省值为 CTRWIN，即在屏幕的中央。当用户按回车键或 ESC 键后，这个临时窗口就消失了。如果候选窗口的指针不是 NULL，则假设这个指针指向一个已经在屏幕上的窗口，该窗口被临时选作显示信息的窗口。信息显示过后该窗口不消失，但仍重新选择原来的当前窗口为当前窗口。

子工具包中的 7 个函数如下：

```
void numnewlines(char *s,int *n,int *w);
void popmsg(int x,int y,char *msg,char *title,char soundout,
         wincolors *wc);
void reperr(int level,char *msg);
void repmsg(char *msg);
void sayerr(int ferr,int errflag,int lno,char *pname,
         char *fmt,...);
void beep(void);
unsigned int getkey(void);
```

(1) 函数 beep 很简单，它调用 Turbo C 的 sound、delay、nosound 函数使扬声器发声。函数 sound 和 delay 都带有参数，可以调节发声的时间和频率。

(2) 函数 getkey 用来捕获 CTRL_C 键。它调用函数 bioskey(0)从缓冲区中取得下一个键。如果这个键是 CTRL_C，则弹出一个“流产”窗口，询问用户是否想“流产”。如果程序希望提供一种“流产”的办法，那么可以用 getkey 代替 bioskey。这对于调试程序是很有用的。应该注意，Turbo C 的 ctrlbrk 函数可以做类似的事情，但它不如 getkey 函数好，因为它只有在执行 DOS 系统调用时才检查 CTRL_C 键。而很多函数，如 bioskey、cprintf、putch 等，在其执行过程中不使用 DOS 的系统调用，所以也就得不到检查 CTRL_C 键的机会。另一方面，这些函数又是在窗口输出时经常要用到的，所以必须通过 getkey 函数来解决问题。

(3) 函数 sayerr 是与错误报告系统的高级接口。它的入口参数是不定的，但前面 5 个是固定的：

ferr	1 意味着输出最后一个DOS错误，0意味着不输出.
errflag	0 意味着一般信息，1意味着警告错误信息，2意味着严重错误信息，1和2统称为错误信息。
lno和pname	如果pname不是空字符串，则意味着它是一个文件名，lno就代表这个文件

内的第几行。如果 pname 是空字符串，则它们被忽略.

fmt,...　　　　类似于printf函数的格式化字符串及其它可选择参数。

函数 sayerr 输出的信息由三部分组成:首先报告 pname 指向的文件名和文件内的行号(lno)，接着报告 DOS 错误信息(ferr)，最后报告用户指定的格式化字符串(fmt)。每一部分都可以为空。参数 errflag 决定了最后组成的信息送往哪一个窗口。若是错误信息(包括警告错误信息和严重错误信息)，则送往错误信息窗口，否则送往正常信息窗口。能够报告文件名和行号，这对于调试 C 语言源程序是非常有用的。在 Turbo C 中，有两个预定义的伪变量__FILE__和__LINE__。在编译时，编译器将用被编译的文件名代替__FILE__，用正被编译的当前行号代替__LINE__。这就可以准确地报告出发生错误的文件名和文件内的行号。如:

```
handle=open("nosuchfile",O_RDONLY);
if(handle==-1)
     sayerr(1,2,__LINE__,__FILE__,"%s\r\n","nosuchfile");
```

在报告发生错误的文件名和行号之后，报告 DOS 错误信息"file not found"。

为了使用方便，这个子工具包中定义了如下 12 个宏以代替 sayerr 函数的前 4 个参数。

如果在错误信息报告中要包括文件名和行号:

```
#define  SWRNF   0,1,__LINE__,__FILE__
#define  SERRF   0,2,__LINE__,__FILE__
#define  SMSGF   0,0,__LINE__,__FILE__
#define  FWRNF   1,1,__LINE__,__FILE__
#define  FERRF   1,2,__LINE__,__FILE__
#define  FMSGF   1,0,__LINE__,__FILE__
```

如果在错误信息报告中不包括文件名和行号:

```
#define  SWRN    0,1,0,""
#define  SERR    0,2,0,""
#define  SMSG    0,0,0,""
#define  FWRN    1,1,0,""
#define  FERR    1,2,0,""
#define  FMSG    1,0,0,""
```

在这些宏定义中，第 1 个字母是"S"和"F"，前缀 S 表示不要包括 DOS 错误信息，F 表示要包括。第 2～第 4 个字母是"MSG"、"WRN"、"ERR"，分别表示一般信息、警告错误信息、严重错误信息。第 5 个字母是"F"或空，分别表示要报告文件名和行号或不要。

函数 sayerr 内部用到了 Turbo C 的许多特性，如参数数目不定技术，通过 sprintf 函数进行内部格式化，通过 strerror 取得最后一个 DOS 错误信息等等。

(4) 函数 reperr 在错误信息窗口报告错误。level=0 意味着警告性错误信息,level=1 意味着严重错误信息.

(5) 函数 repmsg 在一般信息窗口报告信息.

(6) 函数 numnewlines 计算信息字符串 S 内换行符的数目，同时也返回信息的行数和一行内最大的宽度。

(7) 函数 popmsg 在指定的屏幕绝对坐标位置(X,Y)弹出一个窗口，窗口高度和宽度正好可以放下信息 msg，在窗口的头上显示标题。如果 soundout 不等于 0，则能听到警铃声。直到用户按回车键或 ESC 键，这个窗口才会消失。信息内可以含有换行符，窗口能自动适合信息量的大小。应该注意，在信息的结尾应该有一个换行符，否则窗口内的信息可能发生滚动。popmsg 函数非常有用，当想显示多行信息而又不想具体规划窗口大小时，就可以用这个函数。这个函数的限制是：格式化后的输出不允许超过 256 个字符。

下面就是这个工具包的源程序:

```
#include <stdio.h>
#include <stdarg.h>
#include <conio.h>
#include <bios.h>
#include <string.h>
#include <dos.h>
#include <process.h>
#include <errno.h>
#include "popup.h"
#include "mouse.h"

static char  *level_msg[ ] = {"WARNING","ERROR"};
static char  *pmsg  = "press <Enter> or <ESC>...";
int          errx  =  CTRWIN;
int          erry  =  CTRWIN;
windesc      *errw  =  NULL;
int          msgx  =  CTRWIN;
int          msgy  =  CTRWIN;
windesc      *msgw  =  NULL;

void numnewlines(char  *s,  int  *n,  int  *w)
{
      int j,k;
      for (*n=0,j  =  0,*w=0,  k=0;  (s[j] !=0);  j++,k++)  {
        if (s[j]  ==  '\n') {
              (*n)++;
              if (k  >  *w)  *w  =k;  k  =  0;
           }
              if (k  >  *w)  *w  =  k;
      }
}

void popmsg(int x,  int y,  char  *msg,  char  *title,
             char soundout,  wincolors  *wc)
{
      int wd,ht,c,k;
      windesc  *w;
```

```
    numnewlines(msg,&ht,&wd);
    ht += 3;
    if (wd < (strlen(pmsg)+1)) wd = strlen(pmsg)+3;
    else wd += 3;
    w = draw_win(x, y, wd, ht, title, popup, wc);
    mouse_off(1);
    cprintf(msg);
    cprintf(pmsg);
    mouse_on(1);
    if (soundout) beep( );
    do {
        while(! (k = mouse_trigger(0)));
        if (k ==CTRLC) exit (0);
        if (k !=CRKEY && k != ESCKEY) beep( );
    } while(k !=CRKEY && k != ESCKEY);
    rmv_win(w);
}

void reperr(int level, char *msg)
{
    windesc *wsave;
    if (errw == NULL) { /* popup error window */
        popmsg(errx,erry,msg,level_msg[level],1,&errcolors);
    }
    else { /* else route to existing error window */
        wsave = curr_win;
        slct_win(errw);
        mprintf("%s: %s", level_msg[level],msg);
        slct_win(wsave);
    }
}

void repmsg(char *msg)
{
    windesc *wsave;
    if (msgw == NULL) { /* popup message window */
        popmsg(msgx, msgy, msg,"Msg ", 0, &msgcolors);
    }
    else { /* else route to existing message window */
        wsave = curr_win;
        slct_win(msgw);
        mprintf("Msg: %s", msg);
        slct_win(wsave);
    }
}

void sayerr(int ferr, int errflag, int lno,
            char *pname, char *fmt,...)
{
    va_list arg_ptr;
    char t[255];
```

```
    int j;
    if ( * pname) {
        j = sprintf(t,"On line %d in pgm %s\r\n",lno,pname);
    }
    else j = 0;
    if (ferr ==1) {
        j += sprintf(t+j,"%s\r",strerror(errno));
    }
    va_start(arg_ptr,fmt);
    vsprintf(t+j, fmt, arg_ptr);
    va_end(arg_ptr);
    switch(errflag) {
        case 1:
            reperr(0,t);
            break;
        case 2:
            reperr(1,t);
            break;
        default:
            repmsg(t);
    }
}

unsigned int getkey(void)
{
    windesc * w;
    int k;
    while(1) {
        k = bioskey(0);
        if (k == CTRLC) {
            w=draw_win(CTRWIN,CTRWIN,25,3,"",popup,&errcolors);
            mprintf("Abort program (Y / N) ?");
            k = bioskey(0);
            rmv_win(w);
            if((k==0x1559) || (k==0x1579)) {/ * "Y" or "y" * /
                mouse_reset ( );
                exit(1);
            }
        }
        else break;
    }
    return k;
}

void beep(void)
{
    sound(50);
    delay(25);
    nosound( );
}
```

下面是应用这个子工具包的一个例子，它必须与popup.obj、mouse.obj、sayerr.obj相连接。这个应用程序首先画一个“瓷砖”式主窗口，然后显示一个弹出式错误窗口，显示的错误信息包括DOS错误信息。接着，它又画一个“瓷砖”式错误窗口，在该窗口中显示几条错误信息，包括严重错误信息和警告性错误信息。此后，它又弹出一个一般信息窗口，在其上显示一些信息。最后，注销掉这两个窗口，清屏，退出。

```
#include  <conio.h>
#include  <fcntl.h>
#include  <io.h>
#include  "popup.h"
#include  "mouse.h"
char myfile[ ] = "ghost.dat";

void main( )
{
     int fh;
     windesc  * mainw;
     init_win( );
     mainw  =  draw_win(1,1,80,17,"Main Window",tile, &defcolors);
     mprintf("Will try to open file when you press return ...");
     getch( );
     if ((fh  =  open(myfile,O_RDONLY))  = =  -1)
          sayerr(FERRF,"—— %S ——\r\n", myfile);
     mprintf("\r\nPress return to construct static error window ...");
     getch( );
     errw  =  draw_win(1,17,80,9,"Err window", tile, &errcolors);
     slct_win(mainw);
     mprintf("\r\nReady to try opening again, press return ...");
     getch( );
     if ((fh  =  open(myfile,O_RDONLY))  = =  -1)
          sayerr(FERRF,"—— %s ——\r\n",myfile);
     mprintf("\r\nPress return to see eror message without"
          "line no and source file ...");
     getch( );
     if ((fh  =  open(myfile, O_RDONLY))  = =  -1)
          sayerr(FERR,"—— %s ——\r\n", myfile);
     mprintf("\r\nPress return to see it as a warning ...");
     getch( );
     if ((fh  =  open(myfile,O_RDONLY))  = =  -1)
          sayerr(FWRN,"—— %s ——\r\n", myfile);
     mprintf("\r\nPress return for a wise saying ...\r\n");
     getch( );
     sayerr(SMSG,"The cows jumped over the %s\r\n"
            "and landed on %s\r\n","moon", "mars");
     rmv_win(errw);
     rmv_win(mainw);
     clrscr( );
}
```

4.4 Turbo C Tools 的文本屏幕处理

4.4.1 Turbo C Tools 与 Turbo C 的比较

前面三节介绍的 Turbo C 的文本屏幕处理函数的功能，有如下几方面不足:

(1) 不能取得有关控制卡的详细状态信息;

(2) 不能控制某些显示特性;

(3) 不能使用多个显示页;

(4) 不能使显示属性仅与屏幕显示区域有关而与显示内容无关;

(5) 不能从屏幕上读文本;

(6) 不能跟踪多个屏幕窗口。

前三项的不足使得不能充分利用显示设备的显示能力，后三项不足使得不能有效地编写出交互式程序。Turbo C Tools 解决了这些问题中的大部分。它的文本屏幕处理函数可以分为三大类: 第 1 类是针对整个屏幕的; 第 2 类是专门针对窗口的; 第 3 类是专门针对帮助信息窗口的。这三类分别在 4.4、4.5 和 4.6 节中讨论.

根据函数的功能，把 Turbo C 和 Turbo C Tools 的函数分别进行分类，可以得到表 4–4 这样一张表格。

表 4–4 Turbo C 和 Turbo C Tools 显示输出函数的比较

功能分类	Turbo C 函数	Turbo C Tools 函数
窗口	window	4.5 节
显示设备和显示方式	biosequip textmode gettextinfo detectgraph	scequip scnewdev scchgde scmode scrows scgetvid scsetvid
页控制		scpages scpage scapage
清除和滚动	clrscr insline delline	scpclr viscroll vihoriz
光标控制	wherex wherey gotoxy	sccurst sccurset scpgcur
显示属性和颜色	textcolor textbackground textattr lowvideo highvideo normvideo	scpal1 scpalett viatrect scblink scborder scmode4

续表

功能分类	Turbo C 函数	Turbo C Tools 函数
屏幕输入／输出	putch cputs cprintf clreol puttext movetext gettext	scttywrt scread scwrite vidspmsg scclrmsg scattrib
矩形区域输入／输出		scbox screstpg scsavepg scttywin scwrap virdrect viresect viwrrect viwrsect

Turbo C Tools 的窗口函数共有 40 个，没在这个表中列出来，留待下一节中专门讨论。事实上，Turbo C 和 Turbo C Tools 的一个最重要差别是：Turbo C 的绝大部分函数遵守由 window 函数设置的当前窗口边界，而这里列出的 Turbo C Tools 的函数都忽略窗口的影响。另一个普遍的差别是：Turbo C 的函数不支持多个显示页，而所有有关的 Turbo C Tools 函数都支持，这一点以后还会多次看到。第三个差别是:Turbo C 和 Turbo C Tools 的坐标系统不一致，Turbo C 用的是 Borland 坐标系统，列(x)在前，行(y)在后，左上角的坐标是(1,1)。Turbo C Tools 用的坐标顺序是相反的，行在前，列在后，且左上角的坐标是(0,0)。如果在同一程序中同时用到 Turbo C 和 Turbo C Tools 的函数，坐标的不一致会带来很大的麻烦。因为 Turbo C Tools 的函数是以源程序方式提供的，因此读者可对它加以修改，使两者的坐标系统相一致。

从表 4-4 中也可以看到，Turbo C Tools 的函数命名规则很不容易记忆。本节列出的 Turbo C Tools 函数都是以"sc"或"vi"开头，它们的说明和有关的定义分别在头文件 bscreens.h 和 bvideo.h 中。以"sc"开头的函数是通过调用 BIOS 进行屏幕输入／输出，以"vi"开头的函数是通过直接存取显示缓冲区进行输入／输出。

4.4.2 显示设备和显示方式

这方面的函数有 7 个：

```
int cdecl   scchgdev(int device);
char cdecl  scequip(void);
void cdecl  scgetvid(ADAP_STATE *pstate);
int cdecl   scmode(int *pmode,int *pcolumns,int *papage);
int cdecl   scnewdev(int mode,int num_rows);
int cdecl   scrows(void);
int cdecl   scsetvid(const ADAP_STATE *pstate);
```

这些函数中的大部分是 Turbo C 中所没有的，即使 Turbo C 中有类似的函数，其

功能也不如这儿的函数强。下面将逐个介绍这些函数，最后给出一个例子来说明它们的用法。

(1) 函数 scequip 看起来格式简单，它不需要参数，返回的那个字节是微机的标志字节 ID，即在 ROM 内 FFFF:000E 处的那个字节，说明这台微机是什么样的微机。但由于 ID 没有统一的标准，故实用意义不大。scequip 的主要意义在于它设置了 Turbo C Tools 所用的许多全局变量。Turbo C Tools 的全局变量总共约 55 个，函数 scequip 设置其中的 13 个：

b_know_hw	b_mdpa	b_cga	b_ega	b_vga
b_mcga	b_herc	b_pgc	b_mem_ega	b_sw_ega
b_pcmodel	b_submod	b_biosrev		

从上表可以看出，scequip 函数的主要作用就是确定当前显示卡是什么类型。由于早期微机设计上的缺陷，无论是 IBM 的机器还是兼容机，做这件工作不是很容易的，因为显示控制卡上不提供这个信息。Turbo C Tools 是这样解决这个问题的：对于单色显示卡(MDA)，它首先往一个 CPU 输入输出口(这个口本来是用来读取和设置光标位置的)写一个无效的数，然后立即从这个口再读回一个数，两者进行比较。如果一致，则存在 MDA 卡；如果不一致,则不存在 MDA 卡。因为写的数是无效的，所以不会影响光标位置。当然，在测试前，应先把该口的内容读进来保存，测试后再写回去。对于 CGA 和 PGC， 测试过程是类似的。对于 EGA，问题就简单了，因为 EGA 的 ROM BIOS 中已提供了这样一个服务，通过 BIOS 调用就可以得知是什么显示卡和什么显示方式。

Turbo C Tools 的这个测试方法是根据 IBM 的微机设计的。如果兼容机在这些问题上的处理办法与 IBM 不一样，则 scequip 函数可能失效。

(2) 函数 scnewdev 设置字符显示方式和显示行数。字符显示方式一共有 5 种：

显示方式	符号常数	说 明
0	BW40	40X25，合成单色
1	C40	40X25，16 色
2	BW80	80X25，合成单色
3	C80	80X25，16 色
7	MONO	80X25，单色

对于方式 3 和方式 7，如果是 EGA，行数可设置为 25 和 43；对于 VGA，行数可设置为是 25、43 和 50。当在 43 行或 50 行下显示时，需要重新调整字库，调整 BIOS 参数，以便在清屏、滚动和屏幕硬拷贝时能够正确进行操作。有些软件设计得不好，它们不到 BIOS 中去取这些参数，只用缺省的 25 行，其结果将仅在屏幕的上半部操作，好像只有 25 行一样。

这个函数还会设置全局变量 b_device 和 b_curpage，清空适配器上所有的显示页。如果方式值是 0～3，则它认为是彩色显示设备。如果方式值是 7，则它认为是单色显示设备。系统中可以同时安装彩色和单色显示设备，但每种只能安装一台。既然可以有两套显示设备，就存在一个当前显示设备的问题。scnewdev 函数执行完后，它根据显示方

式号推断出的显示设备就成为当前显示设备，以后可以再度通过 scnewdev 切换当前显示设备为另一显示设备。但是，若另一显示设备也曾经被 scnewdev 函数初始化过，则应该用下面的 scchgdev 函数来切换。至于页，执行完这个函数后，页 0 将成为当前编辑页。

如果不成功，这个函数的返回值为-1。

(3) 函数 scchgdev 使所指定的设备成为当前显示设备。它测试所需设备是否存在，设置 BIOS 中的设备标志字。恢复 BIOS 显示状态变量，恢复全局变量 b__device 和 b__curpage，从而不但使设备成为当前设备，而且回到了以前离开这个设备时的页号。参数 device 可以是 SC__MONO(0)和 SC__COLOR(1)，分别表示单色和彩色设备，设备必须是以前用函数 scnewdev 初始化过的，否则不成功，且返回值为 1。如果成功，返回值为 0。

(4) 函数 scmode 是一个重要的常用函数，用来取得当前显示卡上的一些信息,它的三个参数分别返回显示方式值、列数、当前显示页号。整个函数返回值指明是彩色(SC__COLOR,1)还是单色(SC__MONO,0) 显示设备。

(5) 函数 scrows 返回屏幕的字符行数。

(6) 函数 scgetvid 取当前显示适配器的全部状态，并记录在参数 pstate 所指向的 ADAP__STATE 结构变量中。结构 ADAP__STATE 的定义如下：

```
typedef  struct
{
     int high,low;                       /＊光标高、低扫描行＊/
}CUR__TYPE;
typedef  struct
{
     int  mode;                          /＊当前显示方式＊/
     int  act__page;                     /＊当前显示页＊/
     int  cur__page;                     /＊当前编辑页＊/
     int  rows,columns;                  /＊屏幕的行数和列数＊/
     int  curs__off;                     /＊光标打开或关闭＊/
     CUR__TYPE curs__size;               /＊光标高、低扫描行＊/
}ADAP__STATE;
```

(7) 函数 scsetvid 根据参数 pstate 所指向的结构变量内各字段的值，重新恢复显示器的状态。

最后两个函数相当特殊，通过这两个函数，实际上可以用软件来开关显示卡。开关显示卡及恢复显示状态都牵涉到对硬件口地址的直接存取，这里就不介绍了。应该注意，这两个函数的执行一定要求硬件兼容性好。

4.4.3 页控制

页控制包括下列三个函数

```
int cdecl   scapage(int page);
int cdecl   scpage(int page);
int cdecl   scpages(void);
```

大部分显示卡上的显示缓冲区都比存储一页文本所需的容量多。例如 CGA 的显示缓冲区是 16K，可以存储 8 页 40 * 25 的文本或 4 页 80 * 25 的文本。EGA 和 VGA 也可以存储多页，MDA 只能存储一页。可以存储多页的显示卡往往带有这样一种机制：它能把某一页处理为当前显示页，即正在屏幕上显示；而把另一页(或同一页)处理为当前编辑页，即不在显示器上显示，但可以往其中存取信息。这样带来的好处是可以在显示某一页的同时去准备另一页，准备好后，通过切换当前显示页就可以弹出到屏幕上去。PC 和 EGA 的 BIOS 中已实现了这些概念,但 Turbo C 中不支持，只在页 0 上操作，Turbo C Tools 改正了这一点。

(1) 函数 scpages 返回当前显示设备当前显示方式下可用的显示页。

(2) 函数 scapage 设置当前显示页。

(3) 函数 scpage 设置当前编辑页。

下面是 4.4.2 和 4.4.3 节中介绍的部分函数的演示程序。例子很简单，这里不多加解释了。

```
#include <bscreens.h>
#define C40  1
#define LO4  4
#define MONO 7

int OldDevice,OldMode,OldColumns,OldVisiblePage,OldRows;
:
OldDevice = scmode(&)OldMode,
     &OldColumns,&OldvisiblePage);
OldRows = scrows( );
if (scnewdev(C40,25)  == C40 ) {
     ....}
else Puts("Can't do mode  1 !");
if (scnewdev(LO4,25) == MONO) {
          ....}
else puts("Can't do mode  7 !");
scchgdev(COLOR);
:
scchgdev(MONO );
:
scnewdev(OldMode,OldRows);
scapage(OldVisiblePage);
```

4.4.4 清除和滚动

Turbo C Tools 中提供了下列三个函数来清除屏幕和滚动屏幕:

```
void cdecl  scpclr(void);
int  cdecl  viscroll(int num__rows,int attrib,int u__row,int u__col,
                     int l__row,int l__col,int dir);
int  cdecl  vihoriz(int num__cols,int attrib,int u__row,int u__col,
                     int l__row,int l__col,int dir);
```

(1) 函数 scpclr 清除当前编辑页，不管它是否当前显示页，以空格符填充该页，属性为正常的黑底白字属性，如果希望使用别的属性或使用清除前原来的属性，则应该用下面将要讲到的 viscroll 或 scwrite 函数。

(2) 函数 viscroll 使当前显示页一个矩形区域向上(SCR_UP,0) 或向下(SCR_DOWN,1)滚动指定的行数(包括字符和属性)，空行使用指定的属性。如果行数为 0，则清除该矩形区域。

(3) 函数 vihoriz 使当前显示页一个矩形区域向左(SCR_LEFT,1)或向右(SCR_RIGAT,0)滚动指定的列数(包括字符和属性)，空列使用指定的属性。如果列数为 0，则清除该矩形区域。

4.4.5 光标控制

Turbo C Tools 提供了如下 3 个光标位置和光标形状控制函数:

```
int cdecl    sccurset(int row,int col);
int cdecl    sccurst(int *prow,int *pcol,int *phigh,int *plow);
int cdecl    scpgcur(int off,int high,int low,int adjust);
```

(1) 函数 sccurset 将光标移到由 row 和 col 指定的当前编辑页的一个位置上。如果当前编辑页不是当前显示页，则这种光标移动是看不见的，但一旦以后当前编辑页被置为当前显示页，则光标就在新位置上了。使用这个函数时，应切记 Turbo C Tools 的坐标系统与 Turbo C 不一样。返回的整数的高低字节分别表示当前光标所在的行号和列号。

(2) 函数 sccurst 读取当前编辑页上光标的位置，报告光标形状，返回值报告光标是打开的(0)还是关闭的(1)。这个函数是通过查询 BIOS 维护的数据来取得信息的。如果某一程序改变了光标形状而不通知 BIOS，则 BIOS 报告的光标形状将是不正确的，导致 sccurst 报告的光标形状也将是不正确的。应该注意的是，BIOS 和 Turbo C Tools 都只记录一个光标的打开／关闭状态，以及一个光标的形状。简言之，尽管一块显示卡上可以有多个页，每页都有自己的光标位置，但光标形状和光标的打开／关闭状态却只有一个，无所谓属于哪一页。

(3) 函数 scpgcur 设置光标尺寸和打开／关闭光标。参数 off 为 0 则打开，不为 0 则关闭。参数 adjust 可以是 CUR_ADJUST(1)或 CUR_NO_ADJUST(0)，其含义是进行调整或不调整。所谓调整，则不论 CGA 还是 EGA、VGA、MCGA，光标将按下表含义对上扫描行(high)和下扫描行(low)进行解释:

上扫描行	下扫描行	光标形状
0	1	顶部一横条
0	3	上半一方块
0	13	全方块
8	13	下半一方块
12	13	底部一横条

如果指定不调整，而所给的 high 和 low 不合适，则可能导致一个不可见的或不连续的光标。

这个函数应对当前显示页进行操作，否则没有任何作用。

4.4.6 显示属性和颜色

Turbo C Tools 提供了如下 6 个控制显示属性和颜色的函数:

```
int cdecl   scblink(int option);
int cdecl   scborder(unsigned color);
int cdecl   scmode4(int palett,int backgrd);
int cdecl   scpal1(unsigned reg,unsigned value);
int cdecl   scpalett(const char * ptable);
int cdecl   viatrect(int u__row,int u__col,
                     int l__row,int l__col,
                     int fore, int back);
```

(1) 函数 scpalett 定义 EGA、VGA 或 MCGA 颜色的整个色板。EGA 显示卡若配上相应的加强彩色显示器 ECD（它们之间的连线有 rgbRGB 6 根彩色信号线），则可以同时显示 64 种颜色。但是，显示缓冲区的组织形式却限制了同时只能显示 16 种颜色。这是因为在文本方式下，属性字节中只有 4 位 IRGB 用来表示文本前景颜色；在图形方式下，也只有 4 个色平面，都限制最多 16 种颜色。但是在 EGA 中，这 16 种颜色可以从 64 种颜色中任意挑选。在 VGA 中，这 16 种颜色可以从 256 种颜色中挑选，但现在的 C 版本不支持。在 CGA 中，由 IRGB 所代表的 4 位彩色码直接送往与显示器的接口上。在 EGA 和 VGA 中，4 位彩色码通过一个属性控制器以后再送到接口上。在 200 行扫描方式时，可通过属性控制器把送来的 4 位 IEGB 码转换成任意的另一个 4 位的 rgbRGB 码。在 350 行扫描方式时，则可把 4 位 IRGB 码转换成任意的另一个 6 位的 rgbRGB 码。这里的小写 rgb 代表三分之一亮度的红绿蓝，大写的 RGR 代表三分之二亮度的红绿蓝。这个转换是由 EGA 上 16 个花色寄存器中的值控制的。函数 scpalett 就是设置这 16 个寄存器的值，并加上最后一个屏幕边界颜色寄存器的值，一共 17 个。入口参数 ptable 指向一个由 17 个字节构成的数组，每个字节的高 2 位被忽略，后 6 位就是 rgbRGB 的值。这 17 个寄存器的隐含值如下:

寄存器号		rgbRGB	颜色
0	0x00	000000	黑
1	0x01	000001	蓝
2	0x02	000010	绿
3	0x03	000011	青
4	0x04	000100	红
5	0x05	000101	品红
6	0x14	010100	棕
7	0x07	000111	白
8	0x38	111000	灰
9	0x39	111001	浅蓝
10	0x3a	111010	浅绿

11	0x3b	111011	浅青
12	0x3c	111100	浅红
13	0x3d	111101	浅品红
14	0x3e	111110	黄
15	0x3f	111111	高亮白
16(边界)	0x00	000000	黑

(2) 函数 scpal1 只是设置某一个花色寄存器的值。请参见上一个 scpalett 函数。

(3) 函数 scmode4 选择显示方式 4 下的调色板和背景色。在显示方式 4 下，颜色更受限制，只能同时显示 4 色，有两个调色板，0 号和 1 号，由参数 palette 选定。每个调色板除背景色外，还有另外 3 种颜色，分别为绿／红／黄和淡绿／紫红／白。背景色由参数 backgrd 选定，其值可以由 0～31。

(4) 函数 scborder 设置屏幕四周的边界颜色。

(5) 函数 scblink 选择文本方式下属性字节的最高位(位 7)作为前景闪烁控制用(若参数 option＝SC_BLINK,1)还是作为背景加亮用(若参数 option＝SC_INTENSITY,0)。缺省值是作为前景闪烁用，这时前景可达 16 色，可闪烁，背景 8 色。若改为作背景加亮用，则前景和背景都可达到 16 色，但前景不能闪烁。虽然 CGA 的属性字节的位 7 也是既可做前景闪烁又可做背景加亮用，但这个函数只对 EGA 起作用，因为不容易对 CGA 卡的这一位进行控制。

(6) 函数 viatrect 用所给属性填充当前编辑页上一个矩形区域，而不影响在那儿的字符编码，光标保持不动。如果指定的矩形区域超过了屏幕边界，则自动调节到使它在一个屏幕内。这个函数仅工作在文本方式(0,1,2,3,7)下。

4.4.7 屏幕写入和读取

属于这方面的函数有如下几个：

```
int cdecl      scattrib(int fore,int back,char ch,unsigned cnt);
int cdecl      scclrmsg(int row,int col,int len);
char cdecl     scread(int *pfore,int *pback);
int cdecl      scttywrt(char ch,int fore);
int cdecl      scwrite(char ch,unsigned cnt);
int cdecl      vidspmsg(int row,int col,int fore,int back,const char *pmsg)
char far * cdecl  viptr(int row,int col);
```

(1) 函数 scattrib 在当前编辑页上从当前光标位置开始写字符 ch，重复 cnt 次，光标保持不动，不滚动屏幕。

(2) 函数 scwrite 与函数 scattrib 类似，只是显示属性不改。

(3) 函数 scttywrt 以 tty 方式在当前编辑页当前光标处写 1 个字符 ch。Turbo C Tools 一共提供了三个 tty 方式输出的函数，还有下一小节将介绍的函数 scttywin 和 scwrap。tty 方式在 4.2.1.1 节中已经介绍过了。只有在当前读写页同时也是当前显示页的情况下，函数 scttywrt 才会正确工作。在图形方式下，参数 fore 用来设置写入字符的颜色。如果发生滚动，还用来设置滚动正文的颜色。只有当前编辑页也是当前显

示页时才会在图形方式下发生滚动。在文本方式下，参数 fore 被忽略，显示属性仍为光标处原来的属性。当滚动发生时，如果滚动是由于 1 个换行字符引起的，则新空行的属性取自滚动前最后 1 个字符的属性；如果滚动是由于屏幕 1 行的溢出而引起的，则新空行的属性取自最后 1 行最左 1 列处的属性。

(4) 函数 vidspmsg 从当前编辑页的指定位置开始，以指定的属性写一条消息，光标保持不动。消息不折转到屏幕的下一行，也不滚动屏幕。这个函数是直接访问显示内存，所以速度很快，但它不能工作在图形方式下。还有一点值得注意，由于这个函数是用一个宏实现的，它多次使用某些参数，这样就可能出现副作用。

(5) 函数 scread 从当前编辑页当前光标位置读 1 个字符的编码和前景／背景属性，分别通过返回值和两个参数返回。如果当前编辑页处于图形方式下，则这些返回值没有意义。

(6) 函数 scclrmsg 从当前编辑页的指定位置连写 len 个空格，但显示属性保持不变，也不移动光标。事实上，这是清除以前已经写的一条消息，为写新的信息做准备。在图形方式下，被清除区域不能超出当前行而发生折转。在文本方式下，被清除区域可以折转超出当前行。这个函数不检查被清除区域是否超出显示缓冲区，这类错误应由程序员防止。

(7) 函数 viptr 不是为了进行读写，而是根据显示位置的行列号计算这个坐标在当前编辑页的显示缓冲区的地址。如果当前不在文本方式下或行列号不合理，则返回远的空指针。

4.4.8 矩形区域写入和读取

属于这一类的函数有如下几个：

```
int cdecl     scbox(int u_row,int u_col,int l_row,int l_col,
                int boxtype,char boxchar,int attrib);
int cdecl     screstpg(const PAGE_STATE *ppage_state);
int cdecl     scsavepg(PAGE_STATE *ppage_state);
void cdecl    scttywin(int u_row,int u_col,int l_row,int l_col,
                     char ch,int fore,int back,
                     int scr_fore,int scr_back);
void cdecl    scwrap(int u_row,int u_col,int l_row,int l_col,
                     int num_spaces,const char *pbuffer,
                     int fore,int back,int option);
int cdecl     virdrect(int u_row,int u_col,int l_row,int l_col,
                     char *pbuffer,int option);
int cdecl     virdsect(int u_row,int u_col,int l_row,int l_col,
                     char *pbuffer,unsigned gap,int option);
int cdecl     viwrrect(int u_row,int u_col,int l_row,int l_col,
                      const char *pbuffer,int fore,
                      int back,int option);
int cdecl     viwrsect(int u_row,int u_col,int l_row,int l_col,
                      const char *pbuffer,unsigned gap,
                      int fore,int back,int option);
```

应该注意，矩形区域尽管给用户的感觉可能像一个窗口，但矩形区域不是窗口，它没

有相应的数据结构，不处理重叠问题，边界和颜色等都需要程序员自己维护。4.5 节将叙述 Turbo C Tools 窗口 。

(1) 函数 scbox 在当前编辑页按指定的左上角和右下角坐标画一个方框。框线在指定的矩形区域内，因此矩形内实际可使用的区域比给定的方框少两行两列。所指定的属性只应用于框线，而不涉及到框内。框线是由字符组成的。如果参数 boxtype 小于 0，则参数 boxchar 就指定了框线字符。如果参数 boxtype 为 0～15，则参数 boxchar 被忽略，固定使用 ASCII 字符集中的框线字符。因为 ASCII 码中的框线分单线和双线，且上、下、左、右四条边框不必都是单线或双线，所以可分为 16 种情况，由 boxtype 指定，具体对应关系如下:

方框	类型	下	右	上	左
0	(0000)	单	单	单	单
1	(0001)	单	单	单	双
2	(0010)	单	单	双	单
3	(0011)	单	单	双	双
4	(0100)	单	双	单	单
5	(0101)	单	双	单	双
6	(0110)	单	双	双	单
7	(0111)	单	双	双	双
8	(1000)	双	单	单	单
9	(1001)	双	单	单	双
10	(1010)	双	单	双	单
11	(1011)	双	单	双	双
12	(1100)	双	双	单	单
13	(1101)	双	双	单	双
14	(1110)	双	双	双	单
15	(1111)	双	双	双	双

(2) 函数 viwrrect 和 virdrect 是一对，分别在当前编辑页向(从)指定的矩形区域写(读)数据。所有数据原封不动地逐行写入(读出)，光标保持不动。写入时可以写'\0'，读出时也不在缓冲区尾加'\0'。参数 option 可以取两个值。如果指定为 CHAR_ATTR，则参数 pbufftr 所指向的缓冲区包含了写入(读出)字符的编码和显示属性，此时函数 viwrrect 中的参数 fore 和 back 就没有意义了。如果指定为 CHARS_ONLY，则缓冲区只包含字符的编码，在写入时，还需通过函数 viwrrect 的参数 fore 和 back 来设置显示属性。如果 fore 和 back 等于-1，则属性保持不变。

这两个函数只在字符方式下工作，必要时会自动调整矩形的尺寸，使它不超出屏幕尺寸。

(3) 函数 viwrsect 和 virdsect 也是一对，它们是 (2) 中所述一对的扩充。实际工作中可能发生这样的情况: 需要往一个大矩形区域的子矩形区域写入或读出数据，而保持大矩形区域内的其它部分不受影响。如果这个子矩形区域是一个横向条块，则把这个横向条块当作一个矩形区域来处理。如果这个子矩形区域是一个竖向条块，则这一对函数提供了一个解决办法，它们比上一对函数多了一个参数 gap，用来说明在往(从)矩形区域写入 (读出) 1 行后，跳过多少个字符位置，再去写入(读出)下一行。例如:

```
#include <bvideo.h>
char buffer[25][80][2];
int  num__writ;
virdrect(0,0,24,79,&buffer[0][0][0],CHAR__ATTR);
viwrrect(0,0,24,79,&buffer[0][0][0],0,0,CHAR__ATTR);
viwrsect(0,10,24,49,&buffer[0][10][0],30,0,0,CHAR__ATTR);
```

(4) 函数 scsavepg 和 screstpg 也是一对，分别从(往)当前编辑页保留(恢复)一屏幕的信息以及光标位置。信息是以压缩格式存放的。信息和光标位置存放在结构 PAGE__STATE 中，两个函数的参数都指向这个结构的指针。结构 PAGE__STATE 定义如下：

```
typedef struct
{
        int  curs__row,curs__column;
        char  * pimage;
        int  image__length;
}PAGE__STATE;
```

(5) 函数 scttywin 与前面已介绍过的 scttywrt 类似，都是以 tty 方式写入 1 个字符。差别是：第 1，scttywin 在当前编辑页的指定矩形区域内当前光标处写，若当前光标不在此区域内，则写到该区域的左上角。第 2，可以通过参数 fore 和 back 指定写字符的前景和背景色。若参数为-1，则在文本方式下用原显示属性，在图形方式下用颜色 1。 第 3，可以通过参数 scr__fore 和 scr__back 指定滚动后产生的空行的前景和背景颜色。若参数为-1，则在文本方式下按函数 scttywrt 的办法处理，在图形方式下用颜色 1。

(6) 函数 scwrap 以 tty 方式向当前编辑页的一个矩形区域内写 1 个字符串。

如果参数 option 为 CHARS__ONLY(0)，则：

第 1，pbuffer 所指向的缓冲区内只包含字符编码，显示属性由参数 fore 和 back 决定。若它们为-1，则在文本方式下，显示属性不变，在图形方式下为颜色 1。

第 2，如果参数 num__spaces 为 0，则缓冲区中的字符串是 ASCIIZ 串，往屏幕上写入这个 ASCIIZ 串，否则写入 num__spaces 个字符。

如果 option 为 CHAR__ATTR(2)，则：

第 1，缓冲区内包含字符编码和显示属性，参数 fore 和 back 被忽略。

第 2，参数 num__spaces 指明要写入的字符的数目。

除了一般的 tty 方式外，函数 scwrap 还做如下几种处理：

第 1，制表符 TAB('＼t')被当作空格符处理。

第 2，DEL 字符('＼177') (八进制)也被当作空格写入屏幕，但它保持与前面的非空格字符的连接。

第 3，NULL('＼0')根据参数 option 和 num__spaces 值的不同有不同的意义。如果指明了 CHARS__ONLY 而 num__spaces 为 0，则 NULL 表示字符串的末尾，否则它也是一个可打印字符，看起来像一个空格。

第 4，保证不在 1 个"字"的中间换行，除非字的长度大于矩形的宽度。"字"定义为一个非"空白字符"序列，即"字"是由"空白字符"括起来的。"空白字符"包括空格、制表符('\t')、响铃('\7')、退格('\8')、换行('\12')和回车('\15')。

第 5，空格不出现在一行的开头。

4.4.9 各种文本输出函数的速度比较

各种文本输出函数的执行速度很能说明各个函数的特点。表 4–5 列出了各种函数以单一字符填满整个屏幕所需的时间，单位为秒。

表 4–5 各种文本输出函数的速度比较

输出函数	4.77M 的 PC 机 大力神显示卡	10M 的 AT 兼容机 EGA 卡	加速 倍数
printf	20.0	1.6	10
puts	20.0	1.6	10
scwrap(pairs)	13.0	3.5	4
scwrap(chars)	12.0	3.4	4
scttywin	12.0	3.2	4
scttywrt	5.3	1.0	5
puttext(BIOS)	2.8	1.1	3
cprintf(BIOS)	2.8	1.1	3
cputs(BIOS)	2.7	1.1	2
cprintf(direct)	0.87	0.14	6
cputs(direct)	0.77	0.12	6
vidspmsg	0.15	0.048	3
scattrib	0.13	0.033	4
scwrite	0.13	0.027	5
viwrrect(chars)	0.033	0.013	3
viwrrect(pairs)	0.024	0.011	2
puttext(direct)	0.022	0.011	2

测试是在两台机器上进行的，一台是 4.77M 主频的老 PC 机，配的是大力神单色图形卡，另一台是 10M 主频的 AT 兼容机，配的是 Everex 的 EGA 卡。表中最后一列列出了两种机器对同一函数的速度比值。先按纵向进行分析，从表中可以看出，速度从上到下分为三档。通过 DOS 的 stdio 标准输出最慢，通过 BIOS 输出快一些，通过直接往显示缓冲区中写已格式化了的文本最快。最快和最慢相差近 1000 倍。再按横向进行分析，从表中可以看出，虽然 AT 机比 PC 机都要快，但快的程度却因函数的不同而不一样。基本上可以这样说，速度越慢的函数，总的输出时间中 CPU 占的时间越多，所以 AT 机对 PC 机有更大的优势。速度越快的函数，总的输出时间中 CPU 占的时间越少，而显示卡

占的时间越多，但两种显示卡的速度差别不大，所以 AT 机对 PC 机的优势减小。

下面是进行这个测试的程序的片断：

```
for(k=0;k<80;s[k++]='@'+i);
s[80]=0;
m=50;
t=biostime(0,0);
for(k=0;k<m;++k)
    for(row=0;row<25;++row)
        vidspmsg(row,0,7,0,s);
dt[i]=(biostime(0,0)-t)/(18.2*m)-dt[0];
```

4.5 Turbo C Tools 的窗口处理

4.5.1 概述

前面已经说过，直至 2.0 版，Turbo C 对窗口的支持都是很弱的，只提供了一个 window 函数指定窗口的范围，随后的输入输出就在这窗口内进行。如果只有一个窗口，这也未尝不可，但如果有多个窗口， 问题就麻烦了，所有窗口管理方面的事都须由程序员来完成（见 4.3 节）。Turbo C Tools 则在这方面大大地加强了。Turbo C Tools 相对于 Turbo C 最重要的改进大概就在窗口方面。

在 Turbo C Tools 6.0 的手册中，一共列出了数十个有关窗口方面的函数，分为 11 类。本书不准备详细介绍所有这些函数。但是，因为窗口是程序用户接口最重要部分，是一个商品软件能否为广大用户所接受的最重要的品质之一，所以本书尽可能详尽透彻地介绍有关技术。

概括起来讲，Turbo C Tools 的窗口支持函数具有如下特点：

第 1，可以建立多个窗口.这些窗口不但可以在同一页上，也可以在不同页上，甚至还可以在不同显示设备上。

第 2，每个窗口都有自己独立的可控制的光标，包括光标的形状、颜色和位置。还可以有自己的边界和标题等等。

第 3，窗口彼此可以重叠，与 4.3 节那个 pop__up 工具包不同的是，任何时候都可以往这些重叠窗口中的任意一个输出。当然，如果某一窗口被别的窗口盖住了，即使只盖住了一部分，输出的内容也不会立即显示，而是等到上面的窗口移走后才会显示。

第 4，窗口可以延迟显示，这在某些情况下非常有用。例如，一个很长的计算过程不断地输出一些信息，如果随时输出随时显示，给用户的感觉可能是无次序的。延迟到整个计算结束后，一次显示出所有信息,给用户的感觉可能更好一些。

第 5，可以有虚拟窗口，也就是说，窗口可以比显示屏幕还大，最大可容纳 32K 字节。这时，借助于一个视口(或称为观察口)来浏览整个窗口的内容，就像拿一个放大镜来看报纸一样。不同的是，此时移动的是报纸(虚拟窗口)而不是放大镜(视口)。

为了进行这么复杂的处理，必然需要一个很好的窗口结构，借助于这个结构保存每个窗口的重要信息。粗略地讲，有关窗口的信息可以分为三大类：

第 1 类是静态信息。这类信息是在窗口建立时就固定下来的，以后不再改动。如:窗口的尺寸，光标的形状，缺省的文本属性，是否延迟显示，窗口边框的说明。边框说明包括框线字符和颜色，标题文本和颜色，标题相对于窗口的位置。

第 2 类是动态信息。这类信息是随时可变的，不是固有的。如:当前光标位置，光标是打开还是关闭，当前显示的文本内容及其颜色。

第 3 类介于上两类之间，它与窗口的显示和关闭有关。如：窗口在哪个设备的哪一页上显示，窗口在屏幕上的位置，窗口是否完全可见，被这个窗口盖住的其它窗口的内容，与上一个窗口和下一个窗口连接的指针，等等。

关于窗口的边框，还需要说明一下。这里虽然把它归于第 1 类（这是一般的想法），但在 Turbo C Tools 中，每次显示一个窗口时，可以指定不同形式的边框，而不是在建立窗口时就确定下来的，所以应把它归于第 3 类。

表 4–6 是 Turbo C Tools 窗口函数的总表。纵向是按窗口的操作过程划分的，从窗口的建立、显示，到往窗口内输出内容，到关闭窗口，最后注销窗口。横向是按数据操作的区域来划分的，从程序的数据区开始，到一个窗口，到窗口的一个矩形，到某一行某一列，到当前光标位置。

表 4–6 Turbo C Tools 窗口函数分类表

	数据区域				
操作	程序	窗口	矩形	行，列	光标
建立	wncreate				
显示	wnredraw	wndsplay wnupdate			
激活		wncursor wnselect			
状态	wnerror	wngetopt			
控制	wnsetbuf	wnsetopt			
清除和滚动		wnscroll wnhoriz	wnscrblk		
光标				wncurmov	wncurpos
属性		wnattr	wnatrblk	wnatrstr	
输出			wnwrrect	wnwrbuf	wnwrap wnprintf wnwrstr wnwrstrn wnwrtty

续表

	数 据 区 域				
操作	程序	窗口	矩形	行，列	光标
输入		wnrevupd wnquery	wnfield		
虚拟窗口		wnread wnscrlbr wninitev wnchgevn wnremevn wnzapevn	wnvdisp wnorigin wnshoblk	wnrdbuf	
关闭		wnremove			
注销	wndstroy				

4.5.2 建立和注销窗口

属于这一类的函数有下列两个：

```
BWINDOW  *wncreate(int height,int width,int attr);
int cdecl  wndstroy (BWINDOW  *pwin);
```

(1) 函数 wncreate 建立一窗口，调用时只需说明窗口的高度、宽度(单位是字符)和数据区的缺省显示属性，返回的是一个指向结构 BWINDOW 的指针。这个结构保存了与这个窗口有关的所有信息。函数 wncreate 为结构 BWINDOW 分配空间，用缺省值填充结构内的各个字段，包括分配窗口的数据区，用具有指定属性的空格填充这个数据区。窗口的总面积不能超过 32767。

BWINDOW 的定义如下。对于这个定义，这儿不统一加以解释，而是结合到每一个函数中去解释。

```
typedef struct                          /* BWINDOW structure:              */
{                                       /*   everything needed to control  */
                                        /*   an individual window.         */
    unsigned signature;                 /* Identifying signature.          */
    LOC      cur_loc;                   /* Location of window's own        */
                                        /* cursor relative to window's     */
                                        /* data area.                      */
    CUR_TYPE cur_type;                  /* Cursor type.                    */
    IMAGE    img;                       /* Contents of data area.          */
    DIM      view_size;                 /* Size of window's viewport.      */
    LOC      data_origin;               /* Coordinates of data area which  */
                                        /* appear at (0,0) in the viewport.*/
    WHERE    where_shown;               /* Where window is currently       */
                                        /* shown.                          */
    IMAGE    prev;                      /* Previous contents of screen     */
```

```
                                        /* (before window was shown),       */
                                        /* including data under border.     */
    WHERE      where_prev;              /* Region occupied by window        */
                                        /* (including border).              */
    struct bwin_node  *pnode;           /* Pointer to window node           */
                                        /* among all windows on a video     */
                                        /* display page.                    */
    BORDER     bord;                    /* Border description.              */
    WN_SENSOR  *psensors;               /* Window sensor list.              */
    WN_EVENT_LIST  *pevent_list;        /* Window event list.               */
    int        attr;                    /* Default attributes for data      */
                                        /* area.                            */
    unsigned cur_left_marg,             /* Left, right, top, and bottom     */
             cur_right_marg,            /* margins to maintain during       */
             cur_top_marg,              /* cursor "auto-track" feature      */
             cur_bottom_marg;           /*                                  */
    struct                              /* "options" substructure:          */
    {                                   /*   items set by user request      */
        unsigned delayed    : 1;        /* Output postponed until next      */
                                        /* update.                          */
        unsigned cur_off    : 1;        /* Cursor invisible.                */
        unsigned removable  : 1;        /* Removable (hence previous        */
                                        /* contents of screen must be       */
                                        /* preserved).                      */
        unsigned hidden     : 1;        /* Invisible, yet attached to a     */
                                        /* location on a display page.      */
        unsigned cur_track  : 1;        /* Always keep cursor visible in    */
                                        /* the viewport.                    */
        unsigned _dummy     :11;        /* Pad to word boundary             */
    } options;
    struct                              /* "internals" substructure:        */
    {                                   /*   not directly set by user.      */
        unsigned frozen     :1;         /* Updates postponed involuntarily  */
        unsigned dirty      :1;         /* Output request(s) are            */
                                        /* awaiting update.                 */
        unsigned                        /* Some portion of data area is     */
            any_data_covered:1;         /* covered by another window.       */
        unsigned cur_frozen :1;         /* Not selected for visible         */
                                        /* cursor.                          */
        unsigned temp_hid   :1;         /* Temporarily hidden by            */
                                        /* internal operations.             */
        unsigned _dummy     :11;        /* Pad to word boundary             */
    } internals;
} BWINDOW;
```

(2) 函数 wndstroy 注销一个窗口。所谓注销一个窗口，不只是从屏幕上消去这个窗口，而是毁掉了它的窗口结构，释放它占用的内存，以后不能再使用这个窗口了。

4.5.3 显示和关闭窗口

属于这一类的函数有如下 6 个:

```
BWINDOW  * cdecl  wncursor (BWINDOW  * pwin);
BWINDOW  * cdecl  wndsplay(BWINDOW  * pwin,const WHERE  * pwhere,
                      const BORDER  * pbord);
int        cdecl  wnredraw (int dev,int page);
BWINDOW  * cdecl  wnremove (BWINDOW  * pwin);
BWINDOW  * cdecl  wnselect (BWINDOW  * pwin);
BWINDOW  * cdecl  wnupdate (BWINDOW  * pwin);
```

(1)函数 wndsplay 显示一个窗口，三个调用参数都是指针，所指向的结构分别说明显示哪一个窗口，在什么位置显示这个窗口，边框和标题如何。

结构 WHERE 的定义如下:

```
typedef struct {
     int row,col;
}LOC;
typedef struct {
     int  dev;
     int  page;
     LOC  corner;
}WHERE;
```

从这个结构中可以看到，窗口可以显示在任一显示设备任一页的任一坐标位置。当然，如果这一页不是当前活动的显示页，则屏幕上实际上是看不到的。在上一节讲述 Turbo C Tools 的一般文本屏幕处理函数时，页被分为当前页和非当前页，当前页又分为当前显示页和当前编辑页。 在这一节讲述窗口时，由于输入／输出的基本对象是窗口而不是页，且 Turbo C Tools 的窗口本来就具有解决重叠和延迟显示的能力，所以，对程序员来说，在实际窗口应用中，当前编辑页这个概念实际上不存在，或者说是由各个函数自动来处理。窗口在当前显示页上就显示，不在当前显示页上就不显示。程序员只需要有一个当前显示页 (简称为当前页) 概念就够了。

对于窗口操作，更重要的概念是当前窗口。不管有多少页，不管有多少个窗口，当前窗口只有一个，而不是每页有一个当前窗口。当前窗口就是当前进行输入／输出的窗口，如果这个窗口正好在(一般是如此)当前页上，这个窗口就会在屏幕上显示出来。如果程序员编的程序连接了 Turbo C Tools 的 NW_???.OBJ 文件，则 Turbo C 的屏幕输出函数，如 printf、cputs 等，是知道这个窗口的存在的，即可以用 Turbo C 的屏幕输出函数往 Turbo C Tools 建立的窗口中写内容。但这些内容是直接写到屏幕上，而不是写到窗口结构中的数据区内，故必须通过函数 wnrevupd 存到窗口结构内。应该注意，一个窗口为当前窗口，不意味着这个窗口正显示在屏幕上，更不意味着没有别的窗口盖住这个窗口，也不意味着这个窗口的光标是被激活的。每一页只有一个窗口的光标是被激活的，但这个窗口不一定是当前窗口。函数 wndsplay 执行完毕后，所指定的窗口就成为当前窗口，而且还激活该窗口的光标(虽然光标可能是不可见的)，那一页上其它窗口的光标被关

闭。

结构 BORDER 定义窗口的边界，包括边界字符及其显示属性，标题及其显示属性，标题的位置，如下所示：

```
typedef struct {
    int     type;          /* Type of border.                  */
    int     attr;          /* Attribute of border.             */
    char    ch;            /* Border character (if type==0).   */
    char    _dummy;        /* Structure alignment.             */
    char    *pttitle;      /* Title on top border.             */
    char    *pbtitle;      /* Title on bottom border.          */
    int     ttattr;        /* Attributes of top title.         */
    int     btattr;        /* Attributes of bottom title.      */
    BJOINT  *pjoints;      /* Pointer to linked list of joints. */
} BORDER;
```

字段 type 说明边界字符及标题位置。为边界字符定义了 18 个宏常数，为标题位置定义了 7 个宏，type 是由这两个部分"或"起来的。

边界字符宏常数定义：

```
#define BBRD_NO_BORDER 0x000
#define BBRD_CHAR      0x01f
#define BBRD_SSSS      0x001
#define BBRD_SSSD      0x002
#define BBRD_DSSS      0x003
#define BBRD_DSSD      0x004
#define BBRD_SDSS      0x005
#define BBRD_SDSD      0x006
#define BBRD_DDSS      0x007
#define BBRD_DDSD      0x008
#define BBRD_SSDS      0x009
#define BBRD_SSDD      0x00a
#define BBRD_DSDS      0x00b
#define BBRD_DSDD      0x00c
#define BBRD_SDDS      0x00d
#define BBRD_SDDD      0x00e
#define BBRD_DDDS      0x00f
#define BBRD_DDDD      0x010
```

标题位置宏常数定义：

```
#define BBRD_NO_TITLE   0x000
#define BBRD_TLT        0x020
#define BBRD_TCT        0x040
#define BBRD_TRT        0x080
#define BBRD_BLT        0x100
#define BBRD_BCT        0x200
#define BBRD_BRT        0x400
```

(2) 函数 wnselect 选择一个窗口，使它成为当前窗口。但该函数并不激活该窗口光标。那个显示页上的另一个窗口可以通过 wncursor 取得活动光标。

(3) 函数 wncursor 激活指定窗口的光标(虽然光标可能是不可见的)，在那页上的其它窗口的光标被关闭。

(4) 函数 wnupdate 将挂起的输出写入窗口。如果由于屏幕输入或由于延迟更新，或由于其它软件的侵入，使得屏幕上的内容与内存中的窗口映像不一致,则可以通过 wnupdate 函数更新某一指定的窗口，倘若这个窗口不被其它窗口盖住。

(5) 函数 wnredraw 重新显示指定设备指定页上的所有窗口，并使指定的设备和页成为当前设备和当前页。如果已指定页内某一个显示窗口具有活动光标，则该光标被恢复。

(6) 函数 wnremove 关闭一个已在屏幕上显示的窗口，关闭该窗口的光标，恢复屏幕原来的内容。如果该窗口是当前窗口，且程序已同一个 NW_???.OBJ 文件连接，则 Turbo C 字符窗口尺寸被修改成全屏幕，尽管其字符窗口属性未被改变。

函数 wnread 和 wnvdisp 也可以用来显示一个窗口，但它们与虚拟窗口有更密切的关系，故放在 4.5.9 节中叙述。

4.5.4 窗口控制和状态

属于这一类的函数有如下 4 个:

```
int cdecl          wnerror(int ercode);
BWINDOW  *cdecl    wngetopt (BWINDOW  *pwin,int item,int  *pvalue);
unsigned cdecl     wnsetbuf (unsigned size);
BWINDOW  *cdecl    wnsetopt (BWINDOW  *pwin,int item,int value);
```

(1) 函数 wnsetopt 和函数 wngetopt 是一对，用来设置和取得窗口的某一项特征和状态。

可以设置的项有如下一些，其说明与下面的 wngetopt 函数的参数说明是相同的。

WN_CUR_OFF, WN_CUR_HIGH, WN_CUR_LOW, WN_ATTR,
WN_REMOVABLE, WN_DELAYED, WN_PREV_ALLOC, WN_CUR_TRACK,
WN_CUR_T_MARG, WN_CUR_B_MARG, WN_CUR_L_MARG, WN_CUR_R_MARG.

wnsetopt 设置内部的控制项值，但不对屏幕做任何可见的操作。例如，即使设置为立即显示，也必须在调用 wnupdate 之后才能使任何挂起的输出写到屏幕上。

这些项的物理含义大部分都是很清楚的，唯有自动跟踪还需要解释一下，但留在 4.5.9 节和虚拟窗口一并加以说明。

wngetopt 读取窗口的某一项控制值或某一状态，可以取得的项如下表所示:

符号名	项	值	意义
WN_DEVICE	1		窗口所在的设备(SC_COLOR(1)SC_MONO (0). 如果未显示,则为 SC_ABSENT (-2))
WN_PAGE	(-2)	0-7	窗口所在的屏幕页

符号名	项	值	意义
WN__ROW__CORNER	(-3)	0-79	视口左上角的行号
WN__COL__CORNER	(-4)	0-49	视口左上角的列号
WN__HIDDEN	(-5)	0	窗口是隐藏的
		1	窗口是不隐藏的
* WN__CUR__OFF	6	0	光标打开
		1	光标关闭
* WN__CUR__HIGH	7	0-13	光标高扫描行
* WN__CUR__LOW	8	0-13	光标低扫描行
* WN__DELAYED	9	0	输出可以立即执行
		1	输出延迟，直至 WNUPDATE
* WN__REMOVABLE	10	0	窗口不可删除
		1	窗口可删除
WN__FROZEN	11	0	窗口可以更新
		1	窗口不能更新
WN__DIRTY	12	0	窗口保持最新
		1	输出被挂起
WN__ANY__DATA__COVERED	13	0	窗口未被遮盖
		1	窗口被遮盖
WN__ROWS	14	1-32767	数据区行数
WN__COLS	15	1-32767	数据区列数
WN__ROW__REL	16	0-32767	相对于窗口的光标行号
WN__COL__REL	17	0-32767	相对于窗口的光标列号
WN__ROW__ABS	(-18)	0-49	相对于屏幕的光标行号
WN__COL__ABS	(-19)	0-79	相对于屏幕的光标列号
* WN__ATTR	20	0-255	缺省属性
WN__BOR__TYPE	22		边界类型：见 WNDSPLAY
WN__BOR__CHAR	23	0-255	边界字符
WB__BOR__ATTR	24	0-255	边界属性
WN__IS__CURRENT	25	0	不是当前窗口
		1	是当前窗口
WN__ACTIVE__CUR	(-26)	0	该窗口光标不是活动的
		1	该窗口光标是活动的
WN__ROW__OVERALL	(-27)	0-49	被覆盖区左上角的行号
WN__COL__OVERALL	(-28)	0-79	被覆盖区左上角的列号
WN__HT__OVERALL	(-29)	1-80	被覆盖区的高度
WN__WID__OVERALL	(-30)	1-50	被覆盖区的宽度
* WN__PREV__ALLOC	31	0	缓冲区未分配给原屏幕数据
		1	缓冲区已分配给原屏幕数据
WN__BOR__TTATTR	32	0-255	上标题属性

符号名	项	值	意义
WN_BOR_BTATTR	33	0–255	下标题属性
* WN_CUR_TRACK	34	0	光标可能移出视口
		1	在视口中显示的区域自动移动，显示出光标("自动跟踪")
* WN_CUR_L_MARG	35	0–79	视口上间隙的行数
* WN_CUR_R_MARG	36	0–79	视口下间隙的行数
* WN_CUR_T_MARG	37	0–49	视口左间隙的列数
* WN_CUR_B_MARG	38	0–49	视口右间隙的列数
WN_HT_VIEW	(−39)	1–50	视口高度

其中，带星号的项可通过函数 wnsetopt 进行设置。

(2) 函数 wnsetbuf 分配一块内存，供函数 wnprintf 使用，所分配的内存保存在全局变量 b_wnprbf 中。在调用 wnprintf 函数时，若发现尚未分配缓冲区，则 wnprintf 函数自己会去申请分配 B_WNPBRF_SIZE(1024)字节。因此，除非在下列两种情况下，否则没有必要调用 wnsetbuf。第 1，希望缓冲区不是 1024 字节。第 2，希望不是直到调用 wnprintf 时才分配内存，如常驻内存程序在驻留内存之前或一般程序在分配大量内存之前。可以多次调用 wnsetbuf 来调整缓冲区的大小。

(3) 函数 wnerror 通过返回值返回最近发生的窗口和菜单错误的代号。但是，这个函数还要求一个调用参数，说明要设置的新的错误代号。设置一个错误代号从逻辑上是讲不通的。实际应用中都是设置新错误代号为 WN_NO_ERROR(0)，意思是没有错误。这就相当于返回最近发生的错误号，同时清除错误记录器 b_wnerr。

4.5.5 清除和滚动

属于这一类的函数有如下 3 个：

```
BWINDOW  * cdecl  wnhoriz (int num_cols,int fore,int back,int dir);
BWINDOW  * cdecl  wnscrblk (BWINDOW  * pwin,int r1,int c1,int r2,
                  int c2,int fore,int back,int dir,
                  int count,int option);
BWINDOW  * cdecl  wnscroll (int num_rows,int fore,int back,int dir);
```

(1) 函数 wnscroll 和 wnhoriz 分别在垂直和水平方向滚动窗口，包括字符编码和显示属性。因为参数中不包含窗口结构指针，所以一定是当前窗口。参数 dir 规定方向:向上，SCR_UP(0); 向下，SCR_DOWN(1); 向右，SCR_RIGHT(0); 向左，SCR_LEFT(1)。参数 for 和 back 指定滚动后产生的空格的前景和背景颜色。如果参数为-1，则颜色不变。这两个函数都修改窗口的整个数据区但不移动窗口的光标。如果滚动行数(或列数)为 0，则相当于清除窗口.

(2) 函数 wnscrblk 不是滚动整个窗口，而是滚动窗口内的一个矩形区域。方向参数 dir 规定如下：向上，WNSCR_ UP(0); 向下，WNSCR_ DOWN(1); 向右，WNSCR_ RIGHT (2); 向左，WNSCR_ LEFT(3)。参数 option 有两种选择：WN_UPDATE(0)表示更新窗口，除非窗口是“延迟”的；WN_NO_UPDATE(4)表示不更新窗口。其余的说明与上两个函数相同。

4.5.6 光标位置查询和控制

属于这一类的函数有如下两个，窗口光标的形状应通过 wnsetopt 函数控制。

```
BWINDOW  *cdecl  wncurmov (int row,int col);
BWINDOW  *cdecl  wncurpos (int *prow,int *pcol);
```

(1) 函数 wncurmov 移动当前窗口的光标到指定的位置。如果“自动跟踪”有效，则虚拟窗口会发生移动，以保持光标在视口的自动跟踪区域内是可见的。前面已经说过，当前窗口的光标不一定是活动光标，除非通过 wncursor 函数使当前窗口的光标被激活，否则发生的变化看不见。如果窗口被“延迟”，则新光标位置也不可见，直至窗口被更新。如果程序与 NW_???.OBJ 文件连接在一起，则 Turbo C 的窗口输出函数也会得知这个光标位置的变化。

(2) 函数 wncurpos 返回当前窗口的光标位置。

4.5.7 属性控制

属于这一类的函数有如下 3 个：

```
BWINDOW  *cdecl  wnatrblk (BWINDOW *pwin,int r1,int c1,int r2,
                           int c2,int fore,int back,int option);
BWINDOW  *cdecl  wnatrstr (BWINDOW *pwin,int row,int col,
                  int count,int fore,int back,int option);
BWINDOW  *cdecl  wnattr(int fore,int back);
```

(1) 函数 wnattr 改变当前窗口的显示属性。但这种改变是临时性的，不影响窗口的缺省属性。如果希望影响缺省属性，应该用 wnsetopt 函数。

(2) 函数 wnatrblk 改变指定窗口(不一定是当前窗口)的一个矩形区域(不是整个窗口)的显示属性，其余与上一个函数一样。调用参数 option 可以有如下两个值，WN_UPDATE(0)表示更新窗口，除非窗口被延迟，WN_NO_UPDATE(4)表示不更新窗口。

(3) 函数 wnatrstr 从指定窗口指定坐标位置开始，连续改变 count 个字符的显示属性，可以跨行，直至窗口末端。窗口不发生滚动，光标也不移动。其余说明与上一个函数相同。前一个函数实现的是一个“域”的概念,而这个函数是“流”的概念。

4.5.8 窗口输出 / 输入

这一类函数，有如下 10 个，前 7 个用于输出，后 3 个用于输入：

```
int          cdecl  wnprintf ( const char  * pfmt,  ...);
void         cdecl  wnwrap (int num__spaces,const char  * pbuffer,
                      int fore,int back,int option);
int          cdecl  wnwrbuf (int row,int col,int num__spaces,
                      const char  * pbuffer,int fore,
                      int back,int option);
BWINDOW    * cdecl  wnwrrect (BWINDOW  * pwin,int u__row,int u__col,
                      int l__row,int l__col,const char  * pbuffer,
                      int fore,int back,int option);
void         cdecl  wnwrstr(const char  * pbuffer,int fore,int back);
void         cdecl  wnwrstrn (BWINDOW  * pwin,const char  * pstring,
                      int num__spaces,int fore,int back,int option);
void         cdecl  wnwrtty(char ch,int fore,int back);
int          cdecl  wnfield(BWINDOW  * pwin,char  * pinitstr,
                      char  * pretstr,int size,ED__CONTROL  * pctrl,
                      KEY__SEQUENCE  * pfinal);
char         cdecl  wnquery (char  * presp,int resp__size,int  * pkey);
BWINDOW    * cdecl  wnrevupd (void);
```

由于窗口输入输出函数较多，尤其是窗口输出，使程序员有时不知道究竟应该使用哪一个。在往窗口输出以前，先想清楚下列问题会有助于挑选出合适的函数。

(1) 往哪儿写？这包括往哪一个显示设备、哪一页、哪一个窗口(当前窗口还是指定窗口)、从哪儿开始(当前光标位置还是指定坐标位置)。如果是往一个矩形区域内写，还要说明矩形的尺寸和位置。

(2) 数据源从哪儿来?可以是 1 个字符，此时往往在参数中直接给定这个字符，也可以是多个字符，此时往往来自一个缓冲区。

(3) 写多少？可以是 1 个或多个字符。在多个字符时，可以是 1 个 ASCIIZ 串，遇到字符('\0')则结束写入操作；也可以是一个非 ASCIIZ 串，此时可以"源"为限，即写满指定的字符数，也可以"目"为限，即写满指定的区域。如果指定的字符数在窗口内容纳不下，可以窗口为限截短，也可不以窗口为限。

(4) 属性如何？可以缺省的属性，也可以当前的属性，还可以指定的属性。可统一指定，也可分别指定每一字符的显示属性。写的过程如果发生滚动,还可以不同的办法指示滚动后产生的空行的显示属性。

(5) 其它特性如何？这包括是否为 TTY 方式写，写的过程光标位置是否移动，屏幕上的显示内容是否立即更新，视口内的内容是否自动跟踪，是否进行回车／换行符对与换行字符之间的转换，是否允许格式化输出，等等。

下面分别介绍这些函数。

(1) 函数 wnwrrect 用指定的缓冲区中数据填充窗口的一个矩形区域。所谓数据，就不一定是 ASCII 字符，可以是任意的二进制数据，包括 NULL('\0')。数据原封不动逐行地写，光标保持不动。必要时调整矩形尺寸，以便不超过窗口的边界。如果缓冲区中的数据格式像显示缓冲区似的，由字符编码和显示属性组成，则参数 option 应为 CHAR__ATTR(2)，此时参数 fore 和 back 不起作用。如果缓冲区中的数据仅包括字符编码，则参数 option 应为 CHARS__ONLY(0)，此时参数 fore 和 back 用于显示属性。

表 4-7　窗口输出函数的比较

	往哪儿写	写数据源	写多少	属性如何	其它特性
wnwrrect	指定窗口 一个矩形区域	缓冲区	写满矩形	缺省 统一指定 分别指定	一切字符均写 光标不动
wnwrbuf	当前窗口 指定起始位置 一片连续区域	缓冲区	ASCIIZ 串 或指定字符数 以窗口为限	缺省 统一指定 分别指定	一切字符均写 光标可动 或不动 不滚动窗口
wnwrap	当前窗口 当前光标位置 一片连续区域	缓冲区	ASCIIZ 串 或指定字符数	缺省 统一指定 分别指定	光标移动 TTY 方式 滚动窗口 字不跨行
wnwrtty	当前窗口 当前光标位置	参数内	1 个字符	缺省 指定	光标移动 TTY 方式 滚动窗口
wnwrstr	当前窗口 当前光标位置 一片连续区域	缓冲区	ASCIIZ 串	缺省 统一指定	光标移动 TTY 方式 滚动窗口
wnwrstrn	指定窗口 当前光标位置 一片连续区域	缓冲区	ASCIIZ 串	缺省 统一指定 分别指定	光标移动 TTY 方式 滚动或不滚动窗口 延迟或不延迟 跟踪或不跟踪 转换或不转换
wnprnitf	当前窗口 当前光标位置 一片连续区域 可带格式化 往哪儿写	缓冲区 写数据源	ASCIIZ 串 写多少	缺省 属性如何	光标移动 TTY 方式 滚动窗口 其它特性

(2) 函数 wnwrbuf 往窗口中写 1 个字符串，但不一定是 ASCIIZ 串。它与上一个函数有很大差别。第 1，它是往当前窗口而不是指定窗口中写。第 2，它是从指定坐标开始写，按窗口的尺寸像“字符流”似地写下去,而不是写到一个矩形区域。第 3，它结束写的条件不是写满一个矩形，而是由下列三个条件之一决定：写满了参数 num__spaces 指定的字符数；如果 option 为 CHARS__ONLY 且 num__spaces 为 0，则缓冲区的字符串是 ASCIIZ 串，并写完这个 ASCIIZ 串；写到了窗口的末端。第 4，它移动光标，移动的办法由参数 option 决定。除以前见过的 CHAR__ATTR 和 CHARS__ONLY 外，参数 option 还有其它选择，可以把多个合理选择“或”起来。

符号	值	意　义
CUR_BEG	0	光标放在指定的行列处
CUR_AFTER	1	光标放在字符串的后面
CHARS_ONLY	0	缓冲区仅包含字符编码，属性为fore和back
CHAR_ATTR	2	缓冲区包含字符编码和属性
MOVE_CUR	0	根据CUR_BEG／CUR_AFTER设置光标位置
NO_MOVE_CUR	4	保持光标位置不变

(3) 函数 wnwrap 也是写一个并非一定是 ASCIIZ 串的字符串。与函数 wnwrbuf 主要的差别在于它是以 TTY 方式写，其次是它从当前光标位置开始写，写完后光标一定移到写入字符串的末尾。所谓 TTY 规则，在这个函数以及在随后的各个窗口函数中，包括下列内容：

第 1，按 4.2.1 所述的办法处理回车('\r')、换行('\n')、退格('\b')和响铃('\a')字符。

第 2，对下列字符作特殊处理：

TAB制表符('\t')	作为空格
DEL字符('\177')	作为空格写入屏幕(0x20)，但不作为"字"的分界符
NULL('\0')	仅当参数num_spaces为0并且option为CHARS_ONLY时，才被看作是字符串的结束符，否则与 DEL 同等对待，显示的是 1 个空格，但写往屏幕的值是 0

此外，唯有这个函数还有"字"的概念，使"字"不跨行。"字"定义为 1 个由空的字符或缓冲区开头及结尾包围的非空字符序列。"空的"字符(分界符)包括空格(' ')、制表符('\t')、响铃('\a')、退格('\b')、换行('\n')和回车('\r')。当为避免 1 个字跨行而加入空格时，这些空格使用窗口原来的属性。空格不出现在行首。

(4) 函数 wnwrtty 以 TTY 方式向当前窗口当前光标位置写入 1 个字符，光标移到所显示字符的后面。这个函数还可能引起滚动。

(5) 函数 wnwrstr 以 TTY 方式从当前窗口当前光标位置开始写入 1 个 ASCIIZ 串。

(6) 函数 wnwrstrn 以 TTY 方式向指定窗口写入 1 个字符串，但不一定是 ASCIIZ 串。这个函数还有许多别的有别于 wnwrstr 函数的之处，这从其参数 option 的可选项中可以看出。这些选择项分为 5 组，可以"或"起来，如下所示：

符　号	值	意　义
CHARS_ONLY	0X00	缓冲区仅包含字符编码。如果参数num_spaces为0，则缓冲区中是1个ASCIIZ串
CHAR_ATTR	0X02	缓冲区中包含字符编码和显示属性
WN_UPDATE	0X00	除非窗口具有"延迟"特性，否则更新该窗口
WN_NO_UPDATE	0X04	不更新窗口
WN_LF_CRIF	0X00	换行符('\n')作为一对回车符('\r')和换行符('\n')处理
WN_LF_LF	0X08	换行符('\n')仅作为换行处理
WN_SCROLL	0X00	当换行符('\n')出现在窗口的末行或窗口写满时，窗口上滚

WN_NO_SCROLL	0X10	以窗口边框为限，如果到达窗口的末端，则停止写入，光标放在左下角
WN_NO_TRACK	0x00	不自动跟踪
WN_CHAR_TRACK	0X20	自动跟踪,在视口中移动窗口，以保持从自动跟踪区域中可以看到光标

(7) 函数 wnprintf 以标准的 printf 函数形式向当前窗口写 1 个格式化的 ASCIIZ 串。它也是以 TTY 方式写，但换行符是当作一对回车符和换行符处理的。视口中显示的内容将跟踪光标的移动。wnprintf 在第一次被调用时，它会查询全局变量 b_wnprbf。如果在此之前没有通过 wnsetbuf 函数分配一块缓冲区，则它会调用 calloc 函数分配一个内部字符串缓冲区，其缺省值为 1024 字节。

(8) 函数 wnquery 输入用户从键盘上敲入的内容，将其送入参数 presp 指向的缓冲区，参数 resp_size 说明了缓冲区的尺寸。在键盘输入过程中，除下列键外，其余敲入的 ASCII／IBM 字符一律存入缓冲区，并在屏幕上回显，光标也跟着移动。即使窗口是“延迟”的，以前挂起的输出也仍然会显示出来，并回显键盘输入，退出时再恢复“延迟”状态。

· 退格或ctrl_H	删除以前输入的字符并将光标移到那个位置。若输入刚开始时键入这个字符，则引起鸣叫
· NUL(ASCII0):	被忽略，不存入缓冲区
· Ctrl_G	引起鸣叫，不存入缓冲区
· 回车	结束输入，不存入缓冲区
· Ctrl_M	结束输入，不存入缓冲区
· Ctrl_J	结束输入，不存入缓冲区
· 非ASCII／IBM键	结束输入，不存入缓冲区

退出输入只能按回车键或 Ctrl_M 或 Ctrl_J 或任一非 ASCII／IBM 键(如某一功能键)。如果已到达窗口的末尾或缓冲区已满，用户还继续输入，则输入的内容(除退格键和结束键外)将被忽略，并将引起鸣叫，直至用户敲入某一个结束键。结束键本身不存入缓冲区，但结束时，在缓冲区内输入内容之后补了一个 NUL('\0')字节。结束键的编码通过返回值返回，其扫描码通过指针参数 pscan 返回。

(9) 函数 wnfield 对窗口中的一个域(即一个矩形区域)进行编辑。详见第 5 章。

(10) 函数 wnrevupd 用当前显示在屏幕上的数据更新当前窗口数据区已保存的拷贝，使得 Turbo C Tools 的窗口与通过 Turbo C 字符窗口或通过其它手段显示的数据保持同步。如果屏幕光标位于当前窗口的视口内，则移动当前窗口光标，使它与屏幕光标相重合。如果程序与一个 NW_???.OBJ 的文件相连接，则窗口的缺省属性被设置为与 Turbo C 字符窗口的缺省属性相匹配。

在下列情况下，则执行出错，错误号保存在 b_wnerror 中，返回一个 NIL 指针。

① 没有一个窗口被指定为当前窗口。

② 当前窗口不正在显示。

③ 当前窗口的视口被另一个窗口所遮挡。

④ 程序已与一个 nw_???.obj 文件连接，而视口尺寸又与 Turbo C 字符窗口尺寸不符。

4.5.9 虚拟窗口

虚拟窗口是 Turbo C Tools 窗口特性中很具特色的一个特性。Turbo C Tools 的窗口最大容量可达 32k 字节，而一个屏幕一般只能显示 2k 字节，故同时在屏幕上显示的只能是窗口的一部分。用来显示这一部分内容的屏幕区域就称为视口，而原来的那个大窗口就称为虚拟窗口，仍简称为窗口。在进一步讲述各个虚拟窗口函数以前，应先弄清下列概念。

虚拟窗口与窗口没有本质上的差别，仅在于屏幕上的显示方法不同而已。这就是说，建立虚拟窗口的概念后，只需要增加少许有关显示虚拟窗口的函数，而无需另一套有关窗口建立和注销以及窗口输入之类的函数。在建立窗口时，不需要说明一个窗口是普通窗口还是虚拟窗口，只是在显示时，若按普通窗口显示，则调用函数 wndsplay；若按虚拟窗口显示，则调用函数 wnvdisp 或 wnread。事实上，即使一个窗口小到可以在一个屏幕内放下，也可以按虚拟窗口的办法去显示它，尽管没有必要这样做。

在显示一个虚拟窗口时，屏幕上实际看到的是它的视口，或者说此时视口"冒充"为"窗口"。所以，在调用 wnvdisp 或 wnread 函数显示一个虚拟窗口时，所指定的显示位置和显示边界，指的是视口的显示位置和加在视口周围的边界。视口的尺寸显然不应超过窗口的尺寸。如果视口起点坐标的选择使得它越过了窗口，则会自动调节使它适合于窗口。

为了在视口中看到整个窗口的内容，必须进行移动。移动是相对的，可以说成是视口在窗口内移动，就像拿放大镜看报纸一样；也可以说窗口在视口内移动，就像在显微镜下看切片一样。考虑到视口在屏幕上是不动的，所以本书采用"窗口在视口内移动"的说法，尽管这里的"内"字并不合适，因为窗口比视口大，它不可能在视口"内"。

因为视口不能同时显示窗口的所有内容，故必须提供让窗口在视口中移动的办法。

第 1 种办法是让用户控制窗口在视口内的移动，这可通过键盘的光标移动键进行，也可以通过鼠标进行。如果通过鼠标进行，则边框上还要有滚动指示条，这就是说，每个视口都应辅之以一个结构，说明它对哪些键和鼠标按钮有反应，如何反应。见 wnread 函数。

第 2 种办法是使窗口自动移动以保持光标(从而也包括光标所指处附近的内容)始终显示在视口内，使注意力集中在当前读写位置处，这就叫做自动跟踪。自动跟踪可通过前面已讲过的 wnsetopt 函数建立或撤消，缺省值是建立。为了取得更好的自动跟踪效果，最好使光标靠近视口的中央(或某一边)，以便看到尽可能多的相关连的内容。为此，可以限制光标离视口四条边的距离 (这些距离称为视口边隙)。视口边隙初始化时为 0，但可以通过 wnsetopt 函数加以修改。

第 3 种办法是通过程序控制窗口的移动，见 wnorigin 函数。

因为实际上在屏幕上显示的是视口，所以，如果程序与 nw__???.obj 文件相连接，则 Turbo C 字符窗口被设置为与视口相匹配，而不是与虚拟窗口相匹配。

前面介绍的函数中，许多与虚拟窗口的概念有关，但下面 9 个函数是专门针对虚拟窗口的。有了上面这些概念后，很多函数的意义就很清楚了。

```
int            cdecl   wnchgevn (WN__EVENT__LIST  * * ppfirst,
                          const WN__EVENT  * pevent,WN__ACTION action);
int            cdecl   wninitev (BWINDOW  * pwin);
BWINDOW      * cdecl   wnorigin (BWINDOW  * pwin,int row,int col,int option);
WN__EVENT    * cdecl   wnread (BWINDOW  * pwin,const WHERE  * pwhere,
                        int view__ht,int view__wid,int org__row,
                        int org__col,const BORDER  * pbord,
                        WN__EVENT  * pfinal,int option);
int            cdecl   wnremevn(WN__EVENT__LIST  * * ppfirst,
                        const WN__EVENT  * pevent);
BWINDOW      * cdecl   wnscrlbr (BWINDOW  * pwin,int view__ht,int view__wid,
                        unsigned attr,int option);
BWINDOW      * cdecl   wnshoblk (BWINDOW  * pwin,const REGION  * pblock,
                        const LOC  * phot1,const LOC  * phot2,
                        int option);
BWINDOW      * cdecl   wnvdisp (BWINDOW  * pwin,const WHERE  * pwhere,
                        int view__ht,int view__wid,int org__row,
                        int org__col,const BORDER  * pbord);
void           cdecl   wnzapevn(WN__EVENT__LIST  * * ppfirst);
```

(1) 函数 wnvdisp 在视口中显示一个虚拟窗口。此函数与 wndsplay 函数类似，只不过多了一个虚拟的概念，所以在参数中多了一个视口宽度和高度，以及第一次显示视口左上角在原虚拟窗口中的坐标位置。

(2) 函数 wnorigin 提供了一种程序中控制窗口在视口中移动的办法，实际上就是重新指定视口左上角在虚拟窗口中的坐标。因为程序控制的移动可能与自动跟踪相矛盾，所以在执行这个函数前，应先关闭自动跟踪特性。参数 option 可以取下列两个值：

WN__UPDATE(0)：　立即更新窗口

WN__NO__UPDATE(4)：　不立即更新窗口

(3) 函数 wnshoblk 也提供了一种程序中控制窗口在视口内移动的办法，但它指定待显示的数据块，并指定块内的两个热点。显示时，首先保证热点 1 在视口内，然后照顾到热点 2，最后照顾到指定的数据块，尽可能多地显示数据块内的内容。出于与上一函数同样的理由，最好先关掉自动跟踪特性。

(4) 函数 wnread 提供了让用户在视口中浏览整个虚拟窗口的办法。它除了具有函数 wnvdisp 所有的功能外，还可以接受用户输入，为此多了两个参数。

多出的第 1 个参数是 pfinal，它指向一个结构 WN__EVENT，该结构用来存放用户的最后一次输入事件。结构 WN__EVENT 的定义如下：

```
typedef struct {
      unsigned long event;
      unsigned long ignore;
      unsigned int  location;
}MOUSE__EVENT;
typedef struct {
      unsigned char character__code;
```

```
    unsigned char key__code;
}KEY__SEQUENCE;
typedef struct {
    unsigned event__type;
    union
    {
        KEY__SEQUENCE keystroke;
        MOUSE__EVENT  mouse;
    }event;
    void     * pdata;
    unsigned data__size;
}WN__EVENT;
```

在这个结构中，字段 event__type 只能取值 WN__KB__EVENT 和 WN__MOUSE__EVENT，分别说明最后一次输入事件是键盘操作还是鼠标器操作。如果是键盘操作，则 event.keystroke 记录按键的 ASCII 码和扫描码。如果是鼠标器操作，则 event.mouse 记录这个鼠标器事件。有关鼠标器事件的说明，即 event 和 ignore 的值的含义，请参考第 3 章的 mocheck 函数。

字段 pdata 是指向一个枚举型变量 WN__ACTION 的指针，data__size 是这个变量的长度，即所占的字节数。该变量可以是下列值之一：

```
typedef enum
{
    WN__NULL,
    WN__SCROLL__LEFT,
    WN__SCROLL__RIGHT,
    WN__SCROLL__UP,
    WN__SCROLL__DOWN,
    WN__PAGE__LEFT,
    WN__PAGE__RIGHT,
    WN__PAGE__UP,
    WN__PAGE__DOWN,
    WN__LEFT__EDGE,
    WN__RIGHT__EDGE,
    WN__TOP__EDGE,
    WN__BOTTOM__EDGE,
    WN__ABORT,
    WN__TRANSMIT
}WN__ACTION;
```

比函数 wnvdisp 多出的第 2 个参数是 option，它可以是下列位值的组合：

符 号	值	意 义
WN__KNOWN__EVENTS	0X00	忽略未认可的事件
WN__UNKNOWN__TRANSMIT	0X01	按下未定义的按键，则返回
WN__ALL__TRANSMIT	0X03	在每次认可的按键或鼠标器事件发生后返回
WN__KBIGNORE	0X04	忽略并废弃所有的按键
WN__USE__MOUSE	0X00	如果有鼠标，则用鼠标
WN__NO__MOUSE	0X08	即使有鼠标，也不用鼠标

下面将介绍的其余 5 个函数实际上都是为 wnread 函数服务的。

(5) 函数 wnscrlbr 往视口边界上加一个滚动箭头。如果 option = WN__HORIZONTAL(0)，则在视口下边界上加一个左右滚动箭头。如果 option = WN__VERTICAL(2)，则在视口右边界上加一个上下滚动箭头。因为这个函数只是在数据结构上增加一个箭头，而不是显示一个箭头，所以在调用 wnread 函数显示一个窗口之前，应先调用 wnscrlbr 设置滚动箭头。为了设置滚动箭头，窗口必须具有边界，即边界类型不可以是 BBRD__NO__BORDER(0)。

(6) 函数 wninitev 为指定的窗口安装(初始化)一个 wnread 函数认可的事件表。每个窗口都可以有这样一个事件表，存放在窗口结构 BWINDOW 的 pevent__list 字段内，这个函数的参数中不要求给定事件表的地址。也就是说，事件表是缺省的。再加之 wnread 会自动调用这个函数，所以这个函数对程序员来说意义不大，只是在考虑内存分配时才用到它，见 4.5.10 节。

(7) 函数 wnchgevn 往 wnread 认可的事件表中添加一项或修改一项。参数 ppfirst 指向窗口事件表的第一项，假设 pwin 是指向窗口结构的，则&pwin->pevent__list 就可以作为这第 1 个参数使用。参数 pevent 是指向欲添加或修改的某一事件，参数 action 是说明对这一事件如何反应。事件结构 WN__EVENT 可参考函数 wnread，说明反应的变量 WN__ACTION 也可参考函数 wnread。如果事件表中已含有 pevent 所指定的事件，则意味着是修改，用新的反应 action 代替原来的反应，否则意味着是添加。

(8) 函数 wnremevn 从 wnread 认可的事件表中删除一项。

(9) 函数 wnzapevn 删除 wnread 认可的整个事件表，将 * ppfirst 置为 NIL。

4.5.10 常驻内存的窗口程序

在众多的窗口函数中，有些函数是利用标准的内存分配函数 malloc 和 calloc 来动态建立数据结构和缓冲区，对于编写常驻内存程序这是要认真考虑的。如果希望确保在程序终止并驻留之前使窗口数据结构和缓冲区得到分配，则必须在程序终止之前对每个窗口执行下面的语句，分配必需的数据结构变量。

```
wnsetopt(PWINDOW, WN__PREV__ALLOC,1);
```

如果使用 wnprintf 函数，则应在程序终止之前执行一次 wnsetbuf 函数，分配必需的缓冲区。

如果使用 wnread 和 wndsplay 函数，则应在程序终止之前执行一次 wninitev 和 wnscrlbr 函数，分配必需的信息项。

如果建立的窗口是用于显示帮助信息或显示菜单，则还要参考相应章节的说明。

4.6 Turbo C Tools 的帮助信息窗口

临时在屏幕上弹出一个窗口,在这个窗口内显示一些联机帮助信息，以指导用户下一步的行动，这是窗口应用一个很重要的方面。为了使程序员编程方便，Turbo C Tools 在提供了 40 个窗口函数的基础上，又特地为帮助信息窗口提供了一个实用程序和 5 个函数。这一节就是讨论如何利用这些工具。

4.6.1 帮助信息源文件

实用程序的名字是 buildhlp.exe，它的作用是把一个普通的 ASCII 码形式的帮助信息源文件转换成一个以压缩形式存放在磁盘上的二叉帮助文件。帮助信息源文件的扩展名建议为.txt，文件是一行一行组成的，每一行或者以回车符和换行符结束，或者以 Ctrl_Z 字符(0x1a)结束。

帮助信息源文件中可以有制表符 TAB，但它并不起制表符的作用，而是当作一个普通字符一样被显示，样子像一个圆圈，所以最好不要使用制表符 TAB。

在帮助信息源文件中可以出现三种文本行。

(1) 注释行。以一个点和星号(*)开头的行，实用程序在转换时将忽略这一行，不予转换，将来在帮助信息窗口中也将看不到这一行。

(2) 命令行。以一个点和一个命令字开头的行，点和命令字之间没有空格。命令字不区分大小写。

命令有两种，一种是标识命令，其作用是为一条帮助信息取一个名字。另一种是窗口控制命令，其作用是规定帮助信息窗口的有关特性，如下所示。

标识命令

id	定义一条帮助信息的名字字符串
describe	保留

窗口控制命令

xupper	视口左上角相对于屏幕的x坐标，不包括边界
yupper	视口左上角相对于屏幕的y坐标，不包括边界
xlower	视口右下角相对于屏幕的x坐标，不包括边界
ylower	视口右下角相对于屏幕的y坐标，不包括边界
datawidth	窗口宽度
forecolor	窗口前景
backcolor	窗口背景
bordertype	视口边界类型
borderfore	视口边界前景
borderback	视口边界背景
title	视口标题
titlefore	视口标题前景
titleback	视口标题背景

在上述 14 条命令中，id 和 title 之后必须跟随正文。id 命令之后必须跟随一个 1～12 个字符长的标识字符串。字符串用双引号括起来，但不计作标识字符串。字符串不区分大小写，不计前导和尾随的空格符和制表符。字符串中间的空格认为有意义，但连续多个空格也只算作是 1 个空格。

title 命令之后的标题正文用单引号或双引号括起来，但不作为标题正文的内容。字符串可以为空，这意味着清除标题。也就是说，标识必须有，但标题可以没有。

6 条显示属性命令之后必须跟随 16 个颜色常数(0～15)之一。

5 条有关坐标的命令之后应该跟随相应的坐标值。

视口边界类型命令 bordertype 可参考函数 wndsplay 的说明。

(3) 信息行。不以点开头的行是信息行，这些行就是将来在帮助信息窗口显示的内容。

总之，帮助信息源文件是由一条条帮助信息组成的，总的条数不受限制。每条帮助信息以 id 命令开头，说明它的名字，以便以后检索，随后几条就是这条帮助信息的原文，一条帮助信息可以有多行，其总行数应保持行数乘以窗口宽度不超过 32767。

在各条帮助信息之间，可以有一个或多个窗口控制命令，用来改变显示帮助信息的窗口或视口。这些窗口控制命令对随后的各条帮助信息都保持有效，直至被另一个同类窗口命令取代。

帮助信息在窗口中的显示形式与在帮助信息源文件中的形式相同。但是，若帮助信息源文件中的行过长，则会自动被截断，以适合于所规定的窗口宽度。应该注意的是，在窗口控制命令中没有窗口高度命令，窗口高度是自动调节的，保持可容纳下一条帮助信息的所有各行。如果一条帮助信息正文的尺寸比视口大，则用户可以按动光标移动键，使窗口在视口中移动，以浏览信息的各个部分。

帮助信息在显示时使用窗口控制命令所规定的显示属性。但是，如果希望在显示一条信息的中间改变显示属性，则可以嵌入`A(高位置 1 的大写 A)，后随两个定义新属性的十六进制字符。前一个字符定义背景，后一个字符定义前景。这种改变一直有效，直至被重新改变或直至显示完整条帮助信息。

帮助信息源文件中也可以包括特殊的 ASCII／IBM 字符，办法是先嵌入`C(高位置 1 的大写 C)，后随两个十六进制字符，这两个字符定义了特殊字符的 ASCII 值。

4.6.2 缺省帮助信息窗口

如上所述，在帮助信息源文件中可以通过窗口控制命令来设置窗口的许多特性，如果不设置，则使用缺省的窗口特性，其缺省值如下：

窗口控制项	**值**
左上角	行 6，列 10
右下角	行 20，列 70
数据区中的行数	与帮助信息中的行数相匹配
数据区中的列数	61
数据区属性	蓝绿背景，高亮白前景
边界类型	单线
边界属性	蓝绿背景，高亮白前景
标题	没有
标题属性	蓝色背景，高亮前景

表中行列值均相对于屏幕的左上角(1,1)，“角”指的是视口数据区而不是边界。

4.6.3 帮助函数

Turbo C Tools 提供了如下 5 个帮助函数:

```
int          cdecl     hlclose(void);
WN_EVENT     cdecl     hldisp (const char *ppath,const char *pid_string,
                              unsigned long offset,WN_EVENT *pfinal,int option);
int          cdecl     hllookup(const char *ppath,const char *pid_string,
                              unsigned long offset,HL_WINDOW *phelp_win,
                              char **pptext,int *plength,int option);
int          cdecl     hlopen(const char *ppath);
WN_EVENT     *cdecl    hlread(BWINDOW *pwin,const HL_WINDOW *phelp_win,
                              const char *ptext,int length,
                              WN_EVENT *pfinal,int option);
```

(1) 函数 hlopen 为指定的二叉帮助文件在内存建立一个索引，以便加快访问这个文件的速度。如果已经为另一个二叉帮助文件建立了内存索引，则废弃前一个索引，建立新的索引。这个函数还在全局变量 b_help_path 中记录下二叉帮助文件的名字，在 b_phelp_index 中记录下索引的初始结点。

(2) 函数 hlclose 释放由 hlopen 建立的内存索引。

(3) 函数 hllookup 从一个二叉帮助文件中读取一段帮助信息，包括帮助信息的正文和用于显示这段帮助信息的窗口／视口的描述，但并不真正显示这段帮助信息。参数 ppath 和 pid_string 说明从哪一个二叉帮助文件中读哪一段帮助信息。参数 phelp_win 和 pptext 说明把读来的窗口／视口信息和帮助信息正文存放在哪儿。说明窗口／视口显示特性的结构 HL_WINDOW 定义如下:

```
typedef struct
{
    int  view_top;              视口上沿
    int  view_left;             视口左沿
    int  view_bottom;           视口下沿
    int  view_right;            视口右沿
    int  origin_row;            最初显示在视口原点的数据区行号
    int  origin_col;            最初显示在视口原点的数据区列号
    int  data_fore;             数据区前景
    int  data_back;             数据区背景
    int  border_type;           视口边界类型
    int  border_fore;           边界前景
    int  border_back;           边界背景
    int  title_type;            窗口标题类型
    int  title_fore;            标题前景
    int  title_back;            标题背景
    int  data_rows;             映象的行数
    int  data_columns;          映象的列数
    int  xref_fore;             交叉参考项的前景
    int  xref_back;             交叉参考项的背景
    int  highlight_fore;        加亮条的前景
```

```
    int   highlight__back;              加亮条的背景
    char  * pwindow__title;             窗口标题的正文
} HL__WINDOW;
```

其中，border__type 说明边界字符，取值如下：

值	意 义
0~15	用函数 scbox 定义的 16 种边界字符
255	没有边界
其它	边界字符码

title__type 说明标题的位置，取值如下：

值	意义
0	标题在上方中部
1	标题在上方右部
2	标题在上方左部
3	标题在下方中部
4	标题在下方右部
5	标题在下方左部

参数 plength 返回这段帮助信息正文的长度，参数 option 说明帮助信息正文的存放格式，可以是下列三个值之一：

HL__CHARS__ONLY(0)：　返回的正文已被展开，正文中只包含字符编码，不包含显示属性。

HL__CHAR__ATTR(2)：　返回的正文已被展开，正文中包含字符编码和显示属性。

HL__COMPRESSED(0X40)：　返回的正文保持压缩格式。

因为 hllookup 并不显示这段帮助信息,若要显示，则可以调用 hlread 函数实现，或自已建立窗口去显示。

如果必要，hllookup 会自动调用 hlopen 函数，为二叉帮助文件在内存建立一个索引。

(4) 函数 hlread 在视口中显示帮助信息，用户通过键盘或鼠标浏览帮助信息。帮助信息窗口也是一个窗口，其内部仍然是通过 4.5 节所讲述的那些函数去显示。为了显示一个窗口，也需要有一个结构 BWINDOW。函数的第 1 个参数 pwin 就是指向这样一个结构的指针。如果调用时没有分配这样一个结构，这个指针为 NIL，则函数 hlread 会自动建立一个临时窗口，事后再注销。这个临时窗口的许多结构信息来自参数 phelp__win 所指向的结构 HL__WINDOW，窗口内的内容来自参数 ptext 所指向的文本,文本的长度由参数 length 规定。与 4.5 节介绍的 wnread 函数比较一下就可以知道，这两个函数很接近，但函数 hlread 没有必要说明在哪儿显示，视口特性如何，边界特性如何，因为这些信息都已包含在结构 HL__WINDOW 中了。两个函数中关于结构 WN__EVENT 的说明是一样的，都是用来接受用户输入的最后一个键盘或鼠标事件。函数 hlread 的选择项 option 更多一些，可以是下列位值的组合。

符 号	值	意 义
HL_CHARS_ONLY	0X00	正文已展开，仅包含字符编码
HL_CHAR_ATTR	0X02	正文和属性信息均已展开
HL_COMPRESSED	0X40	正义以压缩形式提供
HL_KBIGNORE	0X04	忽略并废弃所有的按键
HL_USE_MOUSE	0X00	如果有一个鼠标器,则使用它
HL_NO_MOUSE	0X08	不使用鼠标器
HL_REMOVE_WIN	0X10	即使窗口以前存在，事后取消之
HL_DESTROY_WIN	0X30	即使窗口以前存在，事后取消并废弃之

(5) 函数 hldisp 集函数 hllookup 和 hlread 的功能于一体，读取并显示一条帮助信息。

由于帮助函数利用了 Turbo C Tools 窗口,所以也需要像 4.5.10 节中所述那样考虑内存分配问题。帮助函数内部也将调用标准的 malloc 函数为二叉帮助文件的索引、为帮助信息正文、为键盘和鼠标事件、为滚动指示条分配所需的内存。要想编写常驻内存程序(即希望不实际显示信息，而提前执行所有这些内存分配操作，以免以后产生内存不够的问题)，则可以按如下步骤去做:

(1) 用函数 hlopen 为二叉帮助文件建立一个内存索引。

(2) 用函数 wncreat 建立一个窗口。这个窗口的尺寸等信息,可以通过调用 hllookup 函数来获得。用函数 hllookup 也可取得帮助信息，并存放到这个新建立的窗口中。

(3) 对该窗口执行下面这样一条语句，预先分配为保留屏幕数据而需要的缓冲区。

```
wnsetopt(pwin, WN_PREV_ALLOC, 1);
```

(4) 用函数 wninitev 建立对键盘和鼠标的缺省反应，并用函数 wnscrlbr 建立滚动指示条，为函数 hlread 作好准备。

执行完上述各步之后就可以显示帮助信息，用户可以通过键盘或鼠标来浏览帮助信息而不用担心内存分配问题。

第 5 章　键盘输入、菜单和编辑

5.1　Turbo C 的键盘输入

5.1.1　概述

为了接收键盘输入，Turbo C 中提供了许多函数。本节将分别介绍这些函数，重点是阐明这些函数的特点，也指出它们的不足。事实上，Turbo C 的函数几乎没有提供任何交互式输入能力，Turbo C Tools 在这方面有相当大的改进。

为了便于理解，众多的键盘输入和用于键盘输入的函数，可以按表 5–1 进行分类。

表 5–1　键盘输入函数的分类

	基　于　DOS			基于 BIOS
	stdio.h		conio.h	
	普通文件	标准文件	控制台	BIOS
回退 1 个字符	ungetc		ungetch	
取 1 个字符	fgetc	fgetchar	getch	bioskey
	getc	getchar	getche	
查键盘是否准备好			kbhit	bioskey
取 1 个字符串	fgets	gets	cgets	
			getpass	
取 1 个格式化字符串	fscanf	scanf	cscanf	

从这个表中可以看到，键盘输入函数基本上可分为 4 级:普通文件级，标准文件级，控制台级，BIOS 级。前 3 级都是基于 DOS 的，但文件级是把键盘当作一个文件看待，而控制台级是基于 DOS 直接与键盘相关的一部分功能，不利用 DOS 的文件特性。文件级又分为标准文件级和普通文件级，从函数名上可以看出，标准文件级是对键盘进行操作，不需要说明文件名。普通文件级实际上是普通的文件操作函数，如果想用于键盘，参数中必须指明 DOS 赋予键盘的文件名(流名为 strin，文件柄为 0，见第 6 章)。由于 DOS 对键盘处理得不够好。故基于 DOS 的那 3 级对键盘处理也不够好。为了得到有效的交互式键盘输入，必须使用 BIOS 这一级。本章的重点将放在控制台级和 BIOS 级。

程序员在挑选他们所需的键盘输入函数时，应该清楚如下几方面的问题:

(1) 该函数是否能返回足够多的信息?

(2) 该函数是否支持某些编辑功能，如删除 1 个字符?

(3) 该函数能否正确检测并处理<Ctrl_Break>? 对其它特殊键的处理能力如何?

(4) 该函数能否在屏幕上回显字符？如果能，在屏幕什么位置回显？用什么显示属性回显？光标是否移动？控制键和非 ASCII 键如何回显？

(5) 该函数能否重定向到某个文件上去或从某个文件中来？

(6) 无效键盘输入是否会破坏屏幕显示效果或引起程序崩溃？

Turbo C 函数对上述 6 个问题中的某些问题可以回答："是"或"否"。然而完整地回答某些其它问题,却往往是困难的，这是因为，DOS 本身的说明是不完整的。从 Turbo C 的源码可以看到库函数用了哪些 DOS 系统调用，但由于 DOS 系统调用本身没有清楚的说明，要弄清这些 Turbo C 函数的行为细节，只能是把所有的问题都试验到。

5.1.2 控制台级(conio)键盘输入处理

控制台级键盘输入的 Turbo C 函数有如下这些：

```
char  *cgets(char  *str);
int  cscanf(char  *format[,address,...]);
int  getch(void);
int  getche(void);
char  *getpass(const char  *prompt);
int  kbhit(void);
int  ungetch(int ch);
```

下面分别讨论这些函数，看看它们能做什么和不能做什么。

(1) ungetch。这个函数回退字符 ch，以便同一程序中下次调用 getch 或 getche 时又能得到这个字符，就好像它刚刚从键盘上输入进去似的。函数的实现办法很简单，它把字符 ch 存放在一个与其它函数共享的全局变量中。变量的值为 0 表示字符为空，非 0 表示已有了 1 个字符。如果变量中已存有 1 个字符，则函数返回值为-1，否则返回值为字符 ch。这个函数的目的就是给程序这样一种机会，让它去检查输入，如果是所需要的字符，则处理它，否则退回去，让同一个程序的另一个函数以后去处理它。但是在下列几种情况下，ungetch 函数不能正确工作。第 1，它只能回退 1 个字符，不能回退多个。第 2，它只能回退 ASCII 字符集中的字符，不能回退扩展的 ASCII 码字符，如光标移动键。第 3，它回退的字符只能供同一程序的各个函数处理，不能供其它程序处理。Turbo C Tools 中的键盘输入函数完全解决了这些问题。

(2) getch。如果 ungetch 已回退了 1 个字符，函数 getch 则返回这个字符，否则从键盘上取 1 个字符，如果需要还进行等待。它只返回 1 个字符，而不管这个字符是用什么办法敲入的。例如，无论是敲<Ctrl_A>还是<Alt_1>，返回的都是 ASCII 码 1。通过 getch 函数也能取到扩展 ASCII 码，但要调用两次，第 1 次调用返回 1 个 0。第 2 次调用才返回扩展的 ASCII 码。关于扩展 ASCII 码，见 5.5 节。还有一点值得注意，如果用户敲入了<Ctrl_Break>，则函数仅仅返回一个 ASCII 码 3，而不做任何处理。当然，你的程序可以把 ASCII 码 3 解释为一个程序终止信号，但这是程序的事，不是 getch 函数的事。程序中有时希望看看是否已从键盘上敲入了 1 个字符，如果没有，则转去做别的事情。getch 函数和许多其它控制台级输入函数不能用来做这件事，因为它们都等待键盘输入。

(3) getche。这个函数与上一个相类似，只是还回显这个字符。事实上，getche 函数的实现是首先调用 getch 函数取字符，然后再调用单字符 TTY 输出函数 putch 显示这个字符，故回显的位置是在当前输出窗口内，回显的属性是当前的文本属性。

(4) kbhit。这个函数弥补了上述控制台级输入函数必须等待的缺陷。它仅仅查看键盘是否已准备好输入字符，如果准备好了，则返回非 0 值，但对字符本身不做任何处理。一般的使用办法可能是这样的：

```
void main( )
{
      while(!kbhit( ));
      printf("Input ready!");
}
```

这个小程序将不断循环，直至按下任一键后，才输出信息。

使用 kbhit 函数要注意两点。第 1，与 getch 函数不同，它把<Ctrl_Break>解释为用户终止程序运行的信号，因而终止程序运行而不输出任何信息。第 2，它不从 BIOS 缓冲区中取走字符。因此，下面这个程序片断：

```
void main( )
{
      for(;;)
            if(kbhit( )) printf("Input ready!\n");
}
```

将等待从键盘敲入 1 个字符，然后没完没了地输出信息"Input ready!"。

(5) cgets。这个函数用来输入有限长度的字符串，带回显，在输入过程中允许通过<Backspace>键进行编辑。函数的说明形式没有完全表明这个函数的复杂性。如果希望输入 N 个字符的字符串，则应该分配 N+3 个字节的空间给字符串 s，但在调用时设置 s[0]=N+1 函数 cgets 把输入的字符从 s[2]开始依次存放。返回时，s[1]中存放的是实际输入的字符数，包括最后键入的"回车"符，但在字符串 s 中"回车"符已被"\0"代替。cgets 返回的指针是指向 s[2]的。这个函数是常用的，它可读任何 ASCII 串，包括空字符串。输入是以 TTY 方式回显的，在当前的光标位置以当前的屏幕显示属性，却不管当前的 Turbo C 窗口特性，也不移动光标。cgets 函数忽略掉扩展的 ASCII 码，但把<Ctrl_Break>键(即 ASCII 码 3)当作退出程序处理。

(6) getpass。这个函数主要用来输入口令，它首先在当前光标位置(但不管当前窗口)以 TTY 方式和当前的显示属性，显示由参数 prompt 指向的字符串，然后接收键盘输入，最多 8 个字符，接收时不回显，接收的字符串存放在一个静态变量中，返回的指针就指向这个字符串。

(7) cscanf。这个函数是一个高级的带格式的控制台键盘输入函数，它与标准文件级的 scanf 函数非常相似。事实上，两个函数是用同一个低级函数去解释输入的格式。关于格式的说明，请参考 7.7 节。 但是,这两个函数在如何取得输入和如何回退字符上却是不同的。cscanf 是调用函数 getche 取得输入，而 scanf 是调用 stdio 文件输入函数 fgetc 取得

输入。这就是说，cscanf 用的是 DOS 特别面向键盘输入的服务，而 scanf 用的是 DOS 一般文件服务。另外，cscanf 调用 ungetch 回退 1 个字符，而 scanf 调用的是 ungetc。其余的差别还有：cscanf 回显字符时用的是 TTY 方式，在当前光标位置和当前窗口内，以当前的显示属性显示，不回显用户键入的回车符。像 getch 函数一样，cscanf 不提供编辑功能，不把用户敲入的 <Ctrl_Break> 当作终止程序运行处理，而仅把它看作 ASCII 码 3。cscanf 不允许键入空字符串，更糟糕的是不允许键入扩展的 ASCII 码。如果在键入时用户键入了扩展 ASCII 码，后果将是不可预料的。scanf 函数在回显时不管当前窗口，用户键入的回车符也回显出去，而且把 <Ctrl_Break> 解释为终止程序运行的信号。scanf 也同样不允许输入空字符串和扩展 ASCII 码。

cscanf 和 scanf 函数都有一个处理不当之处，这就是它们都会在输入缓冲区留下一个垃圾，在调用这两个函数后，除非以某种办法刷清了缓冲区，否则下次再调用同类型的输入函数时就会取到这个垃圾，从而可能导致程序失败。cscanf 留下的垃圾是回车符(ASCII 码 13)，scanf 留下的垃圾是换行符(ASCII 码 10)。下面的例子 leftover.dem 解释了这种设计上的错误。程序先使用 cscanf 及其相应底层函数，后使用 scanf 及其相应底层函数，提示用户输入 1 个字符串，显示这个字符串，显示字符串的长度，回退 1 个字符，取 1 个字符，再取 1 个字符。ungetch 在回退 1 个字符时会出错，因为 cscanf 已留下一个垃圾在那儿。程序中没有用 ungetc 去回退 1 个字符，因为尽管 scanf 留了一个垃圾，但 ungetc 对此不敏感，测试不出来。因为留下了一个垃圾，所以第 1 次调用 getch / fgetchar 函数去取 1 个字符时，不需等待。第 2 次调用 getch / fgetchar 函数才需要等待用户敲入一个键。解决垃圾问题的办法就是刷清缓冲区。对 cscanf 就是执行一次 getch 并忽略它的返回值，对 scanf 就是执行一次 fgetchar 或 fflush(stdin)。fflush 函数将在第 6 章中介绍。

```
/ * leftover.dem * /
#include <conio.h>
#include <stdio.h>
void main( )
{
    char s[81];
    printf("\nEnter a string:          ");
    cscanf("%s",&s);
    printf("\ncscanf  read          %s",s );
    printf("\nstring length         %d",strlen(s) );
    printf("\nungetch('A') =        %d",ungetch('A'));
    printf("\ngetch( ) =      ASCII %u",getch( ) );
    getch( );
    cputs("\n");
    printf("\nEnter a string:     ");
    scanf ("%s",&s);
    printf("\nscanf  read           %s",s );
    printf("\nstring length         %d",strlen(s) );
    printf("\nfgetchar( ) = ASCII %u",fgetchar( ) );
    fgetchar( );
}
```

5.1.3 标准文件级键盘输入处理

标准文件级键盘输入的 Turbo C 函数有如下这些:

```
int  getchar(void);
int  fgetchar(void);
char  * gets(char  * s);
int  scanf(const char  * format[,address,...]);
```

在 5.1.1 节中，之所以称这几个函数为标准文件级，是因为它们利用了文件服务，但又不需要指定文件名，而隐含指的是键盘。但实际上，这几个函数与普通文件级函数都是基于 fgetc 函数的，也就是说，它们最终还是利用了 DOS 的普通文件服务。虽然如此，但由于已应用到标准输入文件和键盘设备，故它们是以行缓冲方式工作的，在输入过程中，可以使用下列标准编辑键，就像编辑 DOS 命令行一样:

< <- > < -> > <BS> <Del> <Esc> <F1> <F3> <Ins>

回车<CR>是终止编辑过程，引起对程序的真正输入。 函数 gets、getchar、fgetchar 与 scanf 一样地处理<Ctrl_Berak>，但没有 scanf 恼人的垃圾问题。

所有文件级(包括标准的和普通的)键盘输入函数都有一个致命问题: 它们永远不会从输入缓冲区中移走<Ctrl_Z>(即 ASCII 码 26)。这是因为，C 语言把<Ctrl_Z>看作一个文件结束标志。一旦用户从键盘上敲入了<Ctrl_Z>，这些函数就不再等待输入，而是立即返回，返回值为 EOF=65535。更糟糕的是，程序再也不能从缓冲区中把这个 EOF 去掉，不但调用 getchar 不行，甚至调用 fflush 也不行。

(1) gets。这个函数输入 1 个字符串，在调用时必须分配足够大的空间来存放这个字符串。没有什么办法能限制输入字符的个数，所以，gets 输入很可能会写过已分配空间的末尾，引起混乱。另外，gets 的回显也很可能干扰屏幕上别的显示内容。

(2) getchar 和 fgetchar。这两个函数看起来是一样的，前者实际上在 stdio.h 中定义一个宏。因此，要使用 getchar，必须在程序中包括头文件 stdio.h，这一点不如 fgetchar 方便。前面已经说过，这两个函数都带有行缓冲，但实际上使用这两个函数只取 1 个字符，带有行缓冲反而很不方便，故很少使用。

(3) scanf。这个函数输入格式化字符串。它与 cscanf 函数很相似，在上一节做了详细说明。

5.1.4 普通文件级键盘输入处理

普通文件级键盘输入的 Turbo C 函数有如下这些

```
int  fgetc(FILE  * stream);
char  * fgets(char  * s,int n,FILE  * stream);
int  fscanf(FILE  * stream,const char  * format[,address,...]);
int  getc(FILE  * stream);
int  ungetc(int c,FILE  * stream);
```

这些函数与其说是用于键盘输入还不如说是用于文件输入，因此它们将归并在讲述基

本文件处理的第 6 章。当这些函数用于键盘输入时，正如前面所述，它们与标准文件级没有多大差别。

5.1.5 BIOS 级键盘输入处理

Turbo C 函数库中用于键盘输入的上述 3 级都是建筑在 DOS 的基础上，它们都不能处理非 ASCII 键输入。如果输入非 ASCII 键，则会引起混乱。更糟糕的是，如果程序一定要避免这种情况的发生，则必须取得足够的信息，可这 3 级函数连这样的完整信息也不能提供。还有其它许多问题也不能解决，如：如何区分能产生同样字符的不同的击键或击键系列？如何读取和设置小键盘上数字键的状态？如何检测到一些意想不到的键组合(如同时按下 P 和 Q)？键盘输入是带缓冲的，用户击键的速度有时比软件对击键的处理还要快，当用户响应某个提示而敲入相应的键时，软件取得的却可能是以前敲入的键，怎样才能避免这个问题呢？如此等等。要彻底解决这些问题，必须对 PC 键盘控制和 BIOS 对键盘的操作有更深一步的理解。

5.1.5.1 中断 0x9

中断 0x9 是由硬件触发的中断，每次按下键和释放键都会产生这个中断。在 IBM PC 技术手册中，已详细公布了这个中断处理程序的源码，名字为 KB__INT。首先决定此次中断究竟是按下键还是释放键。如果是释放键，则一般不做任何处理，除非是左 shift、右 shift、Ctrl、Alt 4 个键。这 4 个键用来表示处于什么状态，按下时是一种状态，释放时是另一种状态。Capslock、Numlock、scrolllock、Ins 4 个键也是用来表示处于什么输入状态，但它们是带锁定性质的，每触发一次(包括一次按下和一次释放)改变一次状态。所以前 4 个键应称作状态移位键，后 4 个应称作状态锁定键，统称为状态控制键。KB__INT 记录下这 8 个键的状态,每按一次键，KB__INT 根据键的号码以及左 shift、右 shift、Ctrl、Alt、capslock、Numlock 6 个键的状态来决定用户此次按键的意图，并把相应的码存放在键盘缓冲区内。键盘缓冲区是由 BIOS 维护的，是一个环形缓冲区，可以存放 15 次击键，它与前面几节所述的 DOS 标准输入文件缓冲区根本不是一回事，两者没有任何关系。

缓冲区中存放的键码由两个字节组成，可能有下述 3 种情况：第 1 种情况击下的是普通键，此时低字节(偶缓冲区地址)存放的是 ASCII／IBM 码，高字节(奇缓冲区地址)存放的是键的编号，也称为键的扫描码。第 2 种情况击的是某些功能键或某些键的组合，标准的 ASCII 码内不包括这些键，此时低字节存放的是 0x00，高字节存放的是扩展 ASCII 码。第 3 种情况是一手按下 Alt 键，另一手击打小键盘上的数字键而直接输入的 ASCII 码。此时低字节存放的是 ASCII／IBM 码，高字节存放的是 0x00。标准 ASCII 码共有 256 个，从 0～255，但许多 ASCII 码不能像第 1 种情况那样直接从键盘上找到对应的键(如扑克牌图形)，此时只能通过第 3 种办法输入。

附录 2 列出了中断 9 解释的所有击键及键的组合，它们产生的 ASCII 码或扩展 ASCII 码包括了上述第 1 种情况和第 2 种情况。第 3 种情况实际上可用来键入任何 ASCII 码，从 0～255，在这个表中也就没有必要列出来了。表中以符号"X"开头的码是扩展 ASCII 码。表中的空项表示 KB__INT 对这种键或键的组合不起作用。在 83 个键中，

状态锁定键 NUMLOCK 只影响 71 号至 83 号这 13 个键。有几种键的组合，KB__INT 虽然不产生编码，但却采取相应的动作。这些键的组合有：

<shift-prtsc>	执行中断5，一般将产生屏幕硬拷贝。
<ctrl-alt-del>	执行中断0x19，一般将引起热启动。
<ctrl-numlock>	设置一个暂停标志，进入一个空循环，直至这个标志被清除。当处理普通击键时，KB__INT 将清除这个标志，所以，这个键组合的作用就是暂停当前程序的执行，等待一个普通击键。
<ctrl-break>	执行中断0x1B，把码00:00放在键盘缓冲区。

上面讲的是 IBM 技术手册中标准中断 0x9 的处理程序。但是，不少程序员写了自己的中断 0x9 处理程序，以扩充中断 0x9 的处理能力。例如，键组合<Ctrl__ $ >在标准 0x9 处理程序中是不用的，如果程序员希望用这个键组合来做某件事，则可以修改中断 0x9 处理程序。在内存驻留程序中，这是经常使用的技术。

5.1.5.2　中断 0x16

中断 0x16 是一个软中断，但也是由 BIOS 实现的。调用这个中断可以取 1 个字符，可以只查看是否已准备好 1 个字符，还可以查看各状态键的状态。但是，在 C 语言中使用中断不太方便，所以在 Turbo C 中提供了一个函数 bioskey，其功能与中断 0x16 相同。

5.1.5.3　bioskey 函数

这个函数的格式如下：

```
int bioskey(cmd)
```

参数 cmd 有 3 种值。如果 cmd=0，函数从缓冲区中取下一个键，如果需要等待则等待。返回码的规则如 5.1.5.1 节中所述，是 16 位的完整编码。 如果 cmd=1，则查看是否有键可读。如果有，则返回 1 个非 0 的 16 位值，这 16 位值也就是键的编码，但此键不从缓冲区移走。如果没有，则返回 1 个 0。如果 cmd=2，则返回状态控制键的状态，其各位定义如下：

位 7　Insert 键处在插入状态
位 6　Caps Lock 处在大写字母状态
位 5　Num Lock 处在数字状态
位 4　Scroll Lock 处在滚动状态
位 3　Alt 键正被按下
位 2　Ctrl 键正被按下
位 1　左 Shift 键正被按下
位 0　右 Shift 键正被按下

这个函数能够取回状态控制键的状态，使程序员可以利用 BIOS 中中断 0x9 所忽略的一些键组合来做特殊用途，如作为 Pop__up 窗口中的热键。这样的键组合很多，如 Alt__Enter，Ctrl__7 等等，见附录 2。

一般来说，程序员是不会用中断 0x9 取得键盘输入的，因为这太麻烦，与硬件关系太密切，要求程序员自己把硬件键号翻译为 ASCII 码和扫描码。使用中断 0x16 或 bioskey 函数比较方便而有效，然而它也有两个缺点。第 1，中断 0x16 不通过 DOS，所以 Ctrl_Break 或 Ctrl_C 将可能不起作用，一旦死机将无法退出。第 2，由于不通过 DOS，也就失去了输入输出重定向的功能。

5.2 <Ctrl_Break>和<Ctrl_C>

大家知道，<Ctrl_Break>和<Ctrl_C>一般是用来终止程序运行的，实际上却存在许多令人迷惑不解的地方。例如，为什么各个输入函数对<Ctrl_Break>的处理不一样？为什么<Ctrl_C>大部分时间与<Ctrl_Break>等效而有时却又不等效？为什么不在读键盘时，<Ctrl_Break>也能终止某些程序？在理解了上述各个函数以后，就可以解释这些问题了。中断 0x9 处理程序 KB_INT 在检测到<Ctrl_Break>后，就执行一次中断 0x1B，并把 ASCII 码 0 放在键盘缓冲区。而 KB_INT 在检测到<Ctrl_C>后，仅仅把 ASCII 码 3 放在键盘缓冲区，不调用中断 0x1B。DOS 的键盘输入服务从缓冲区中取字符时，如果遇到 ASCII 码 0，则把它改为 ASCII 码 3，大部分 DOS 服务此时还会把控制传给一个 DOS 子程序，终止当前应用程序的运行。从这个过程中可以看出，为什么通过 DOS 取键盘输入时，<Ctrl_Break>与<Ctrl_C>等效。

值得注意的是：这里说的大部分 DOS 服务，却不包括最经常使用的 Turbo C 的函数所调用的一个 DOS 服务，即中断 0x21,AH＝7 服务。如果一个应用程序从来也不通过 DOS 取得键盘输入服务，情况会怎样呢？当引导 DOS 时，引导程序会初始化中断向量 0x1B，使它指向一段很小的程序。每当按下<Ctrl_Break>，中断 0x9 处理程序就会调用中断 0x1B，进入这一小程序。这段小程序设置一个内部标志，表示 BIOS 已检测一个<Ctrl_Break>。另外，每过一段时间，DOS 就会检查键盘缓冲区，看看是否有<Ctrl_C>，同时也检查那个内部标志，看看是否有<Ctrl_Break>。如果有，则终止当前程序的运行。过多长时间检查一次是由 DOS 的 BREAK 开关(可通过 DOS 的 BREAK 命令设置为 ON 或 OFF)控制的。如果设置为 ON，则每次 DOS 系统调用都会去检查。如果设置为 OFF，则只有部分 DOS 系统调用去做这种检查。在这个检查中，<Ctrl_C>与<Ctrl_Break>有点不一样。在检查<Ctrl_C>时只检查缓冲区头，也就是说，<Ctrl_C>只有位于链头才能被检查到，否则没有什么作用。<Ctrl_Break>标志总是可以被检查到的。为了断定你的程序在执行过程中 DOS 是否会进行检查，必须看看源程序中用到了哪些 DOS 服务，这些 DOS 服务是否去检查，BREAK 开关是 ON 还是 OFF，等等。由于某些 DOS 服务的特殊性，使问题更加复杂化了。如 getch 函数所基于的 DOS 系统调用，仅仅把<Ctrl_C>和<Ctrl_Break>看作输入 1 个 ASCII 码 3，这实际上就关掉了<Ctrl_Break>标志。

5.3 Turbo C Tools的键盘处理

从5.1节中已经看到，Turbo C的键盘处理函数还有很多不足，存在很多问题。实际上，BIOS提供的键盘处理功能还很多，只是Turbo C中没有充分利用，而迫使程序员必须用汇编语言去实现。Turbo C Tools改进了这一点，使得程序员用C语言就可以方便地利用这些功能，免去了低级键盘接口的许多麻烦。

Turbo C Tools 6.0中提供的键盘函数有如下15个(不包括4.5节中已介绍过的窗口键盘输入函数wnquery)。按照用途的不同，可把15个函数分为6类。

· 键盘输入：kbready，kbgetkey，kbquery
· 缓冲区处理：kbflush，kbstuff，kbplace，kbqueue
· 状态控制键处理：kbstatus，kbset
· 使用加强键盘：kbextend，kbequip
· 使用键控制函数：kbwait，kbpoll，kbkcflsh
· 取得键码：kbscanf

5.3.1 键盘输入

```
int  cdecl  kbgetkey (int  * pkey);
char cdecl  kbquery (char  * presp,int resp__size,int  * pkey,
                         int  * pscrolled);
int  cdecl  kbready (char  * pch,int  * pkey);
```

(1) 函数kbready很类似于Turbo C的函数kbhit，根据键盘缓冲区空还是不空，返回0或1。如果不空，则通过pch和pscan返回两字节的完整键码。对于ASCII/IBM字符，pch是字符编码，pscan是扫描码。否则，pch是0,pscan是扩展ASCII码。这个函数不从缓冲区中取走字符。

(2) 函数kbgetkey读取下一个按键，如果必要，还进行等待。它返回的是ASCII/IBM码或0。与Turbo C函数不同的是，它还通过参数pscan返回扫描码或扩展ASCII码，这样在遇到扩展ASCII码时，不需要两次调用键盘输入函数。

(3) 函数kbquery接收一串按键，并在当前显示设备当前显示页上回显这些按键。键入的字符编码(不包括扫描码)存放在参数presp所指向的缓冲区中，最多可键入(resp__size-1)个字符。回显是以TTY方式进行的。在输入过程中，各种按键按如下方式处理：

退格或Ctrl__H:	从缓冲区中删除原先输入的1个字符，光标回退1格。
NUL(ASCII 0):	被忽略并被废弃。
制表符或Ctrl__I:	该字符存入缓冲区，但可能显示多个空格。
ctrl__G:	不存入缓冲区，引起鸣叫。
回车，Ctrl__M,Ctrl__J:	结束输入。
其它ASCII字符	均被回显，包括控制字符。

扩展ASCII字符:　　　结束输入。

在回显过程中，光标随之移动,一列接一列，一行接一行。满一屏后，屏幕发生滚动。一旦输入缓冲区满，光标便停留在最后 1 个输入字符的位置，随后的按键(除退格和结束键外)引起鸣叫并被废弃。当输入结束时执行一个回车和换行。

结束字符不作为输入的部分返回，它的 ASCII 码作为函数值返回，扫描码通过参数 pscan 返回。

5.3.2 缓冲区处理

```
int     cdecl   kbflush (void);
int     cdecl   kbplace (int at_head, char value, char key);
int     cdecl   kbqueue (int *ptotal);
char    *cdecl  kbstuff (int at_head,char *pstring);
```

(1) 函数 kbflush 清除 BIOS 的键盘缓冲区，并返回被清除的键的数目。这个函数常用来确保下一次被处理的键一定是对某种提示的响应。但是，应用过程中有两点值得注意。第 1，这个函数不关闭中断，在返回其调用者之前用户可能又敲入了一个按键，这个按键不会被清除掉。即使在调用这个函数之前关闭中断，也存在一个短暂的间隙，其间也许会出现按键。第 2，BIOS 的键盘缓冲区与 DOS 标准输入文件缓冲区没有任何关系，这个函数只清除 BIOS 的键盘缓冲区，而下一章将要讲到的 fflush 和 flushall 函数则是清除 DOS 的输入文件缓冲区。

下面是一个小演示程序，它循环地做下列三件事，直至回车。先用 Turbo C Tools 的 kbready 函数报告用户一次按键的完整的 16 位码，再调用 Turbo C 的 getch 函数报告同一次按键，兴许要调用两次，最后用 Turbo C Tools 的 kbflush 函数清除键盘缓冲区，为下一次循环做好准备。

```
#include <general.h>
#include <bkeybrd.h>
void main( )
{
    char ch = 0;
    int  key;
    cprintf("Touch Keys!\n");
    for (;ch != CR;)
        if (kbready(&ch,&key)) {
            cprintf ("ch,Key = %3u,%2d\n\r",ch,key);
            cprintf ("getch( ) = %3u\n\r",ch = getch( ));
            if (ch == 0)
                cprintf("getch( ) = %3u\n\r",getch( ));
            kbflush( );
        }
}
```

对于某些特殊按键，输出结果将如表 5–2 所示。

表 5-2　演示程序的输出

输出	说明
ch,key = 8,14	<BS>
getch( )= 8	
ch,key = 8,0	<Alt-(keypad)8>
getch( )= 8	
ch,key = 0,04	<Ctrl_Break>
getch( )= 3	
ch,key = 3,46	<Ctrl_C>
getch( )= 3	
ch,key = 0,35	<Alt_H>
getch( )= 0	extended ASCII
getch( )= 35	

(2) 函数 kbstuff 把一个 ASCIIZ 串插入到 BIOS 的环形键盘缓冲区，可以插入到原有键的前面或后面，由参数 at_head 指定：

KB_HEAD(1):　　　　把字符串放在原有键的前面
KB_TAIL(0):　　　　把字符串放在原有键的后面

参数 pstring 指向欲插入的 ASCIIZ 串。这个字符串只含有字符编码，但在插入到缓冲区中以后会自动补一个扫描码 0，就像是由 Alt_数字小键盘敲入这个键似的。字符串中可以含有扩展的 ASCII 码，方法是在扩展 ASCII 码前加 1 个"\377"。例如：

kbstuff(KB_TAIL，"\377\037dir\377\170")

如果是插到队列的前面，则字符数不应超过 15 个，如果是插到队列的后面，则字符数可以超过 15 个，但此时并不冲毁缓冲区中原来的内容，只是填满缓冲区剩余的空间。如果剩余空间放不下，则返回的指针指向还未插入部分的首字符，以便下次接着插入。如果剩余空间能放下，则返回的指针指向 NUL('\0')。

(3) 函数 kbplace 仅仅把 1 个字符插入到 BIOS 的键盘缓冲区，字符编码和扫描码都由参数指定，插入位置与上一个函数一样，也是由参数 at_head 指定。根据成功与否，返回值 0 或值 1。

(4) 函数 kbquere 通过参数 ptotal 报告 BIOS 键盘缓冲区总共可容纳多少个键，通过返回值报告缓冲区还剩多少个位置。在使用时应注意，如果要使用返回值，最好先关闭中断(通过 utintflg(UT_INTOFF))，再调用这个函数，最后通过 utintflg(UT_INTON)重新打开中断。

这一类函数主要应用在下面这样一些场合。第 1，在常驻内存的程序中，特别是在键盘加强的程序中，用来模拟操作者从键盘上敲入的键，甚至可以模拟一串比键盘缓冲区更长的字符串。第 2，在批处理文件.BAT 中，用来给下一个即将运行的程序提供头几个键盘输入字符，倘若下一个程序在运行之前不首先清除缓冲区。

但是，如果已经存在常驻内存的键盘加强程序，则 kbquere、kbstuff、kbplace 这几

个函数可能工作不正常，因为那些常驻内存的程序可能已用它们自己的键盘服务程序代替BIOS的键盘服务程序。Turbo C Tools虽已考虑到这些问题，但仍然可能存在不兼容性的情况。

下面是一个演示程序，说明这一类函数的用法。在演示这个程序时，字符串：s[0]、s[1]和s[2]的长度分别为L、M和N，应满足关系式L+M+N<15。例如：

s[0]，s[1]，s[2]="aaa"，"bbbb"，"ccccc"

这个程序首先进行三次循环，每次循环等待5s，把用户敲入的键分成三个键序列，不需要回车或其它终止符。程序把键入的头两个键系列从键盘缓冲区中取出，分别构成字符串s[0]和s[1]（在这个例子里分别是"aaa"和"bbbb"）。第3个键系列"ccccc"仍存在键盘缓冲区中。然后程序在缓冲区的前面插入s[1]，在缓冲区的后面插入s[0]，从而在缓冲区内构成字符串：

T=s[1]s[2]s[0]="bbbbcccccaaa"

程序最后通过函数kbplace往缓冲区的尾部送一个回车，以便通过函数gets取得最后构成的字符串T，函数gets要求字符串以回车符结尾，但它不管这个回车符是通过按回车键敲入的还是通过Alt_(小键盘)13敲入的。

```
#include <general.h>
#include <bkeybrd.h>
void main( )
{
     char *index = "LMN";
     int  i,spaceleft,buffersize,keystyped,k,key;
     char s[2][16],t[16];
     printf("Think of integers  %c,%c,%c  with sum  < 15 .",
          index[0],index[1],index[2]);
     for (i = 0; i <= 2; ++i) {
          printf("\nYou have 5 seconds to type %c characters: \a",
                    index[i]);
          delay(5000);
          if (i == 2) break;
          spaceleft = kbqueue(&buffersize);
          keystyped = buffersize - spaceleft;
          for (k = 0; k < keystyped; ++k)
               s[i][k] = kbgetkey(&key);
          s[i][k] = 0;
          printf("\n  %c = %d 。 ",index[i],keystyped);
          if (index[i] > 0) cprintf("You typed  %s .",s[i]);
     }
     printf("\n  Your last  %c  keystrokes are still in the buffer。"
            "\n  I'm stuffing your middle  %c  inputs ahead of them,"
            "\n  and your first  %c  after, followed by  <CR>。   ",
            index[2],index[1],index[0]);
     kbstuff(KB_HEAD,s[1]);
     kbstuff(KB_TAIL,s[0]);
```

```
    kbplace(KB_TAIL,CR,0);
    printf("\nNow  gets is reading and echoing them:\n  ");
    gets(t);
    getch( );
}
```

5.3.3 状态控制键

```
void  cdecl  kbset (const KEYSTATUS  * pkeybd);
int  cdecl  kbstatus(KEYSTATUS  * pkeybd);
```

(1) 函数 kbstatus 实际上是一个宏，它返回各个移位键的状态。状态是通过结构 KEYSTATUS 返回的，这个结构实际上就是存在 BIOS 数据区 0040:0017 一个字的内容。如果某位是 1，则表明状态是 ON，否则是 OFF。相对而言，这个字的低字节更重要一些。

```
typedef struct kstatus
{
   unsigned right_shift     : 1;
   unsigned left_shift : 1;
   unsigned ctrl_shift : 1;
   unsigned alt_shift : 1;
   unsigned scroll_state     : 1;
   unsigned num_state : 1;
   unsigned caps_state : 1;
   unsigned ins_state : 1;
   unsigned filler        : 3;
   unsigned hold_state : 1;
   unsigned scroll_shift : 1;
   unsigned num_shift : 1;
   unsigned caps_shift : 1;
   unsigned ins_shift : 1;
} KEYSTATUS;
```

(2) 函数 kbset 与上一个函数相反，它是把结构 KEYSTATUS 中的内容设置到 BIOS 的数据区，以后 BIOS 就按照这个新的设置去解释键盘输入。

下面这个小程序说明了这两个函数的用法。它只用了结构 KEYSTATUS 中的 left_shift 和 num_state 两项，它只监视<左 shift>和<numlock>两个键，这两个键状态的每一次变更都会在屏幕上显示出来。右面的 shift 键对程序运行没有任何影响，其它任何键则会终止程序的运行。

```
#include  <conio.h>
#include  <bkeybrd.h>
#include  <bvideo.h>
char  * shift  [2] = {"Up      ","down    "};
char  * numlock[2] = {"Cursor ","Numeric"};
void  main( )
```

```
{
    int x,y,high,low;
    KEYSTATUS old,new;
    clrscr( );
    cputs("\n\rLeft shift key     ");
    cputs("\n\r    Numlock key     ");
    sccurst(&y,&x,&high,&low);
    scpgcur(1,high,low,0);
    kbstatus(&old);
    while (kbhit( ) == 0) {
        kbstatus(&new);
        if (new.left_shift != old.left_shift ||
            new.num_state != old.num_state) {
            old = new;
            vidspmsg(y-1,x,-1,-1,shift[new.left_shift] );
            vidspmsg(y  ,x,-1,-1,numlock[new.num_state]);
        }
    }
    new.num_state = 0;
    kbset(&new);
    scpgcur(0,high,low,0);
}
```

为了设置移位键的状态，通常总是先读这些键的状态到结构 KEYSTATUS 中，然后改变这个结构中的某些位，最后再把修改后的结构写回到 BIOS 数据区。为了防止结构 KEYSTATUS 中的状态与 BIOS 数据区中的状态不一致，应该在这个先读后写的短暂时间内关掉可屏蔽中断。

5.3.4 使用加强键盘

```
int cdecl kbequip (void);
int cdecl kbextend (int selection);
```

(1) 函数 kbextend 选用扩展的或一般的 BIOS 键盘服务，根据参数 selection 是 KB_USE_EXTEND(1) 还是 KB_USE_NORMAL(0)。选用扩展键盘服务，则支持 102 键键盘。

(2) 函数 kbequip 探测键盘是否扩展键盘，探测的结果不但通过函数返回值返回，同时也设置全局变量 b_kbnhan 和 b_kbxten 的值，分别表示是否存在扩展键盘和扩展 BIOS。如果存在，则返回值和 b_kbxten 的值为 KB_EXTENDED(1)，b_kbnhan 的值为 KB_ENHANCED(1)。如果不存在，则返回值和 b_kbxten 的值为 KB_NOEXTENDED(0)，b_kbnhan 的值为 KB_NOENHANCED(0)。

5.3.5 使用键控制函数

```
void cdecl kbkcflsh (PKEY_CONTROL pfunc,void * pdata);
int cdecl kbpoll (PKEY_CONTROL pfunc,void * pdata,
```

```
                KEY__SEQUENCE  * pkey__seq,int option);
    KEY__SEQUENCE cdecl  kbwait(PKEY__CONTROL pfunc,void  * pdata);
```

这三个函数的功能与前面已经介绍过的 kbgetkey、kbready 和 kbflush 分别相似，但这三个函数在读取、查询或清除键盘时会调用一个称为键控制函数的特殊函数。指向这个特殊函数的指针以及指向传给这个特殊函数的其它信息的指针都是在调用这个三个函数时通过参数指定的。键控制函数技术功能很强，它可以使用户程序在等待一个按键时继续工作，也可以对按键进行筛选或修改,还可以动态地重新定向数据的输入源而不影响下面程序的设计。

Turbo C Tools 中其它一些执行 BIOS 键盘输入的函数也具有自动调用键控制函数的功能。函数 hlread、kbquery、mnlread、mnread、wnquery 和 wnread 都通过 kbwait 或 kbpoll 函数取得键盘输入，所以都可以使用键控制函数。它们假设指向键控制函数的指针是在全局变量 b__key__ctrl 中，这个全局变量的缺省值是 NIL。如果保持这个变量的值为 NIL，则这些函数不调用键控制函数，而仍然正常工作。编辑函数 enfield 和 wnfield 在其执行过程中也调用键控制函数，但它们假设指向键控制函数的指针是在结构 ED__CONTROL 中的 1 个字段内。

5.3.6 取得键码

```
int  cdecl  kbscanof(int ch);
```

这个函数根据字符的 ASCII 码(范围为 0～127)返回这个字符的扫描码。

5.4 Turbo C Tools 的菜单处理

5.4.1 概述

几乎任何一个商品软件中都用到了菜单技术。菜单是在窗口的基础上发展起来的，有时就称为菜单窗口。菜单窗口与普通窗口不同，它所显示的信息分为一个个菜单项，并能让用户通过光标移动键和其它键进行选择和确认，程序再采取相应的行动。菜单窗口与 4.6 节讲述的帮助窗口也有些不同，帮助窗口一次只显示一条帮助信息，只需让用户浏览这条帮助信息，并不需要选择和确认。但菜单窗口毕竟也是一个窗口，所以 4.5.1 节中所述的许多窗口概念也都适用于菜单，这里也就不再重复了。

与窗口一样，重要的是建立一个好的数据结构，以反应菜单的要求，这个结构名称就是 BMENU。这里不准备详细列出 BMENU 的定义，只是在用到某一部分时才介绍相应的内容。

菜单和窗口的主要区别仅在于菜单项和选单键两方面。

与帮助窗口不一样，Turbo C Tools 的各个菜单项是根据它们相对于菜单窗口的坐标，而不是根据"标识符"(名字)来加以区分的。除位置信息外，每个菜单项还具有下列属性：

- 文本字符串，即该项的简短描述；

· 前景和背景颜色;
· 处于保护还是非保护状态;
· 是否附有长的描述文本。

所谓保护状态，就是这个菜单项确实存在，在屏幕上也能让用户看到，但却不让用户选择。这个特性可以让整个菜单始终保持同样的外貌，尽管其中某些项并不总是合适的。例如，第 1 幅菜单让用户选择显示卡类型，第 2 幅菜单让用户选择显示方式号，用户如果在第 1 幅时选择了 CGA，则在第 2 幅时就不能选择大于 7 的显示方式号。

并不是每个菜单项都要求有长的描述文本，但如果有，一旦用户加亮了这个选择项，相应的长描述文本就会显示出来；一旦这个选择项不再被加亮，相应的长描述文本也就从屏幕上消失。长描述文本具有自己的位置和显示属性。有了长描述文本后，既可以对相应的菜单项进行详尽的描述，以便帮助不太熟悉的用户加深对该菜单项的理解，又可以保持整个菜单简洁明快。Turbo C Tools 的手册中称带有长描述文本的菜单项为 lotus 风格的菜单项。

在 Turbo C Tools 的菜单处理中牵涉到 7 组前景／背景显示属性：

(1) 不被保护的菜单项的短描述文本，以及被 Turbo C Tools 诸函数往这个窗口内写的其它文本。

(2) 被保护的菜单项的短描述文本。

(3) 长描述文本。

(4) 被加亮的菜单项的短描述文本。

(5) 菜单窗口的边界。

(6) 菜单窗口的上标题。

(7) 菜单窗口的下标题。

缺省属性只应用于第 1 组。第 2、3、4 组属性是在建立菜单时分别指定的。第 5、6、7 组属性是在显示菜单窗口时指定的。

菜单窗口与普通窗口的第二大区别就是对用户击键的反应。 用户每敲一个键，其产生的效果可能是下述三种情况的组合：

(1) 移动加亮条;

(2) 产生一次响声;

(3) 返回到调用程序，并同时把所选菜单项的坐标、引起这次返回的击键的键码、状态码等返回给调用程序。

在 Turbo C Tools 中，为了编程方便，为菜单操作所需的各种动作定义了一些符号常数，以便程序员指定敲什么键将引起什么动作。

一般动作(一次击键可以引起多个一般动作)：

符号	值	动作
MN_TRANSMTT	0X0010	从MNREAD或MNLREAD返回，报告所选项的行和列。
MN_BEEP	0X0020	使扬声器发声
MN_DISABLE	0X0080	按键被忽略。
MN_ABORT	0X0100	从MNREAD或MNLREAD返回，报告所选项的行和列，返回一个错误代码。

MN_KBIGNORE	0X0200	忽略并废弃后面的按键，直至下一个认可的鼠标器操作。
MN_HIDE_BAR	0X0800	使亮条关闭，删除说明字符串。
MN_SHOW_BAR	0X1000	使亮条打开(取代MN_HIDE___BAR)。
MN_NOWRAP	0X4000	取代MN_UP、MN_DOWN、MN_RIGHT、MN_LEFT、MN_NEXT或MN_PREVIOUS操作中的回绕动作。

移动加亮条的动作(如果在一般动作中未指明需要 MN_DISABLE):

符号	值	移动亮条动作
MN_NOMOVE	0X000	不移动。
MN_UP	0X001	同列上移。
MN_DOWN	0X002	同列下移。
MN_RIGHT	0X003	同行右移。
MN_LEFT	0X004	同行左移。
MN_NEXT	0X005	移到紧随本项安装之后安装的选项。
MN_PREVIOUS	0X006	移到恰在本项安装之前安装的选项。
MN_FIRST	0X007	移到第1个安装的选项
MN_LAST	0X008	移到最后安装的选项
MN_SELECT	0X009	移到指定位置上的选项
MN_PGUP	0X00A	上移行数为视口高度
MN_PGDN	0X00B	下移行数为视口高度
MN_PGRIGHT	0X00C	右移列数为视口宽度
MN_PGLEFT	0X00D	左移列数为视口宽度

移动加亮条的操作只能选择其中的一个。如果未指明 MN_NOWRAP，则上下方向和左右方向都可以回绕，即到达一端之后，将接着从另一端继续执行。上下左右(UP，DOWN，LEFT，RIGHT)的移动是按屏幕上显示的位置（坐标位置）决定的，而上一个，下一个，第一个，最后一个(PREVIOUS，NEXT，FIRST，LAST)则是按菜单项安装的顺序决定的。因此，如果删掉了某一菜单项而随后又安装了这个菜单项，尽管其坐标位置可能不变，但顺序可能变化，移动加亮条的动作效果可能不一样。

在 Turbo C Tools 中已建立了一张缺省的键分配表，如下所示:

键	ASCII 码	扫描码	一般动作	移动亮条动作
上箭头	0	72		MN_UP
下箭头	0	80		MN_DOWN
右箭头	0	77		MN_RIGHT
左箭头	0	75		MN_LEFT
Home	0	71		MN_FIRST
End	0	79		MN_LAST
Tab	9	15		MN_NEXT
Shift_Tab	0	15		MN_PREVIOUS
PgUP	0	73		MN_PGUP
Pgdn	0	81		MN_PGDN

键	ASCII 码	扫描码	一般动作	移动亮条动作
Ctrl_右箭头	0	116		MN_PGRIGHT
Ctrl_左箭头	0	115		MN_PGLEFT
Ender	13	28	MN_TRANSMIT	MN_NOMOVE
Escape	27	1	MN_ABORT	MN_NOMOVE
增强键盘:				
上箭头	224	72		MN_UP
下箭头	224	80		MN_DOWN
右箭头	224	77		MN_RIGHT
左箭头	224	75		MN_LEFT
Home	224	71		MN_FIRST
End	224	79		MN_LAST
PgUp	224	73		MN_PGUP
PgDn	224	81		MN_PGDN
Ctrl_右箭头	224	116		MN_PGRIGHT
Ctrl_左箭头	224	115		MN_PGLEFT
Ender	13	224		MN_NOMOVE

如果希望建立自己的键分配表，则可以使全局变量 b_mnkeytab 指向自己的表，详见源程序 mndefkey.c。

如果仅仅希望增加、修改、删除某一个键的定义，则可以通过下面将要介绍的 mnkey 函数进行。

由于菜单函数利用了 Turbo C Tools 的窗口，故在内存分配问题上，有关窗口的注意事项同样也适用于菜单。如果要编写常驻内存的程序，为了防止以后产生内存不足的问题，应在程序终止之前建立自己的菜单，设置所有的选项、键分配和鼠标器事件，对于每个菜单都不要忘记执行如下这条语句:

```
wnsetopt(pmenu->pwin, WN_PREV_ALLOC,1);
```

5.4.2 建立、显示和注销菜单

Turbo C Tools 提供了如下几个建立、显示和注销菜单的函数:

```
BMENU    * cdecl  mncreate(int height,int width,int textattr,
                       int hilattr,int proattr,int longattr);
BMENU    * cdecl  mndsplay(BMENU  * pmenu,const WHERE  * pwhere,
                       const BORDER  * pbord);
int      cdecl  mndstroy (BMENU  * pmenu);
BMENU  * cdecl  mnvdisp(BMENU  * pmenu,const WHERE  * pwhere,
                   int view_ht,int view_wid,int org_row,
                   int org_col,const BORDER  * pbord);
```

(1) 函数 mncreate 建立一个菜单，在调用时要求指定这个菜单窗口的高度和宽度，指定 4 组前景／背景显示属性：未被保护的短描述文本，加亮条，被保护的短描述文本，

长描述文本。mncreate 实际上是一个宏，它为结构 BMENU 分配空间并用缺省值填充这个空间。这包括创建一个窗口，用具有指定的未被保护的短描述文本显示属性的空格符，对窗口数据区进行初始化，将菜单窗口的 BWINDOW 结构的地址保存在结构 BMENU 中字段 pwin 内，返回指向结构 BMENU 的指针。

(2) 函数 mndsplay 显示一个菜单，它与函数 wndsplay 极相似，可参考 wndsplay 函数的说明。

(3) 函数 mnvdisp 通过视口显示一个虚拟菜单，它与函数 wnvdisp 极相似，可参考 wnvdisp 函数的说明。

(4) 函数 mndstroy 注销一个菜单。

值得注意的是，菜单函数中不存在一个与普通窗口函数 wnremove 相对应的函数，因此，如果希望暂时从屏幕上关掉一个菜单窗口而不注销这个菜单，则只能通过函数 wnremove 进行，其格式如下：

```
wnremove(pmenu->pwin)
```

5.4.3 定义菜单项和按键鼠标事件

```
BMENU *cdecl mnhilite(BMENU *pmenu,int row,int col,int option);
BMENU *cdecl mnitem (BMENU *pmenu,int row,int col,int option,
                     const char *pstring);
BMENU *cdecl mnitmkey (BMENU *pmenu,int row,int col,int option,
                       const char *pstring,const char *pkeys,
                       int action);
BMENU *cdecl mnkey (BMENU *pmenu,int row,int col,int ch,
                    int key,int act_move,int howchange);
BMENU *cdecl mnlitem(BMENU *pmenu,int row,int col,int option,
                     const char *pstring,int lrow,int lcol,
                     const char *plstring);
BMENU *cdecl mnlitkey(BMENU *pmenu,int row,int col,int option,
                      const char *pstring,int lrow,int lcol,
                      const char *plstring,const char *pkeys,
                      int action);
BMENU *cdecl mnmouse(BMENU *pmenu,unsigned long event,
                     unsigned long ignore,int action,int option);
BMENU *cdecl mnmstyle(BMENU *pmenu,int style,unsigned button);
```

(1) 函数 mnitem 往一个指定菜单窗口内添加、修改或删除一个菜单项。正如前面所述，各个菜单项是根据其坐标来区分的。如果指定的坐标处已经有一项，则意味着对这项进行修改或删除，此时，如果指向新的短描述文本的指针 pstring 为 NIL 或指向的字符串为空字符串，则该菜单项连同它的选择一起被废弃。如果指定的坐标处原来不存在菜单项，则在菜单项表的末尾添加一个新的菜单项，pstring 指向的字符串被写入窗口。

选择项参数 option 具有如下几种作用。

符号	值	意义
MN_NOPROTECT	0	选项可以用亮条选择。
MN_PROTECT	1	选项被保护，亮条不能选择它。
MN_CHARS_ONLY	0	* pstring仅包含字符，以'\0'结束
MN_CHAR_ATTR	2	* pstring包含(char,attr)偶对。第一个包含'\0'字符值的偶对结束这个字符串。

(2) 函数 mnkey 为指定的菜单窗口添加、修改或删除一个键分配。在移动亮条的情况下，按键所引起的移动，应与具体菜单项无关，因此参数 row 和 col 也就不很重要。这个函数常用来说明按某个键所选择的菜单项，此时，动作参数 act_mode 应是 MN_SELECT 或者是 MN_SELECT 与其它常数的逻辑或，坐标参数 row 和 col 必须指明正确的菜单项位置。例如：

```
mnkey(pmenu, 5,2,"I", kbscanof('I'), MN_SELECT, MN_ADD);
```

使用户可以通过按键"I"来选择位于第 5 行第 2 列的那个菜单项。应该注意，这里说的选择只是把亮条移到这个菜单项上，而不是确认这个菜单项。按照 5.4.1 节中的缺省规定，只有回车键才意味着确认。

(3) 函数 mnitmkey 集上两个函数的功能于一体，它向指定的菜单窗口添加一个菜单项并为它分配选择键。因为固定是添加，如果指定的坐标处已有一个菜单项，则此函数不产生什么结果。同一菜单项可以通过多个按键进行选择，例如大写字母"K"和小写字母"k"，所以参数 pkeys 可以是含多个字符的字符串。参数 option 的选择范围与函数 mnitem 相同。

(4) 函数 mnlitem 与函数 mnitem 类似，只是添加、修改或删除的是一个 lotus 风格的菜单项，即带长描述文本的菜单项。因此，在其参数中又增加了三个用来说明显示长描述文本的行列坐标号和文本本身的参数。参数 option 的选择范围与函数 mnitem 相同。

(5) 函数 mnlitkey 与函数 mnitmkey 类似，只是添加的是一个 lotus 风格的菜单项并为它分配选择字符。

(6) 函数 mnmstyle 说明如何通过鼠标来选择菜单项。参数 button 说明用哪一个按钮，可以是下列三种值之一：左按钮 MO_LEFT(1)，右按钮 MO_RIGHT(2)，中按钮 MO_MIDDLE(0)。

参数 style 说明如何通过按钮进行选择，它也可以是下列三种值之一：

符号	值	意义
MN_MOU_CLICK	1	揿方式(揿一次则选择，揿二次则传送，界外揿则取消)。
MN_MOU_DRAG	2	拖动方式(拖动选择，界外拖动取消亮条，释放传送)。
MN_MOU_ALT_DRAG	3	另一种拖动方式(拖动选择，释放传送)。

(7) 函数 mnmouse 添加、修改或删除一个菜单所认可的鼠标器使用事件。上一个函数只说明了最标准的几种鼠标器使用方式，这个函数则可以说明更多的鼠标器事件及对这些事件的反应。

参数 event 说明需处理的鼠标器事件或键盘事件，它可以是下列值的逻辑"或"：

符　号	值	意　义
MO_LEFT	0X0001	左鼠标器按钮
MO_RIGHT	0X0002	右鼠标器按钮
MO_MIDDLE	0X0004	中鼠标器按钮
MO_PRESS	0X0008	按钮按下
MO_RELEASE	0X0010	按钮释放
MO_CLICK	0X0040	按钮快速按下并释放。
MO_DCLICK	0X0080	按钮两次快速按下并释放。
MO_HOLD	0X0800	按钮短暂地处于指定位置。
MO_RSHIFT	0X1000	右 shift 键按下。
MO_LSHIFT	0X2000	左 shift 键按下。
MO_CSHIFT	0X4000	Ctrl 键按下。
MO_ASHIFT	0X8000	Alt 键按下。
MN_ITEMS	0X0200	鼠标位于非保护选项上。
MN_OFF_ITEMS	0X0400	鼠标在视口中但不在任何非保护选项上。
MN_OUT_MENU	0X0100	鼠标在视口数据区之外。

参数 ignore 说明可以忽略的鼠标器事件或键盘事件，它可以是下列值的逻辑"或"：

MO_LEFT　　MO_RIGHT　　MO_MIDDLE
MO_RSHIFT　　MO_LSHIFT　　MO_CSHIFT　　MO_ASHIFT
MO_ITEMS　　MO_OFF_ITEMS　　MO_OUT_MENU

参数 action 说明如何进行处理，见 5.4.1 节关于一般动作和移动亮条动作的说明。

参数 option 的取值是：添加 MN_ADD(0)；修改 MN_CHANGE(1)，删除 MN_DELETE。

例如，下面这条语句将添加一个鼠标器事件，在菜单窗口内（但在所有非保护菜单项之外）揿一下鼠标器的左按钮，就可以使亮条不再被加亮，不管移位键处于什么状态。

```
mnmouse(pmenu,
        MO_LEFT | MO_CLICK | MN_OFF_ITEMS,
        MO_RSHIFT | MO_LSHIFT | MO_CSHIFT | MO_ASHIFT,
        MN_HIDE_BAR,MN_ADD);
```

(8) 函数 mnhilite 与前两个函数不同，它是让程序员来控制加亮条的移动或取消，而不是让用户来控制加亮条的移动。参数 row 和 col 说明对哪一个菜单项进行操作，参数 option 说明如何进行操作，可以是下列值之一：

符　号	值	意　义
MN_UNHIGHLIGHT	0X0000	取消亮条和说明字符串(忽略row和col)。
MN_USE_DESCRIPTION	0X4000	如果选项具有说明字符串，则显示之。
WM_UPDATE	0X0000	除非菜单窗口被"延迟"，否则显示被挂 起的输出。
WN_NO_UPDATE	0X0004	不立即显示被改变的内容。

5.4.4 读取用户的选择

这样的函数有如下几个：

```
int  cdecl  mnlread(BMENU  * pmenu,int srow,int scol,int prow,
                   int pcol,int pch,int pscan,int option);
int  cdecl  mnread(BMENU  * pmenu,int srow,int scol,int  * prow,
                   int  * pcol,int  * pch,int  * pkey,int option);
```

(1) 函数 mnread 读取用户对菜单的响应。执行这个函数以后，按照函数 mnmstyle 和 mnmouse 的说明以及 5.4.1 节中的缺省规定，用户可以移动亮条或做其它一些事情，直至用户敲入一个回车或 ESC 键，才会从 mnread 函数返回。返回时，参数 prow 和 prol 记录着被确认选中的菜单项的坐标，参数 pch 和 pscan 记录着最后一次按键的字符码和扫描码。

应该注意的是，函数 mnread 与函数 wnread 不同，mnread 要求菜单窗口已经显示在屏幕上。所以，在调用 mnread 时不需要再指明在哪儿显示这个菜单窗口,也不需要指明边界特性。参数 srow 和 scol 只是说明最初应加亮哪一个菜单项。

参数 option 可以是下列值的逻辑"或"：

符号	值	意　义
MN__USE-DESCRIPTION	0X4000	显示被显亮选项的Lotus形式的说明(若有的话)。
MN__PREV__BAR	0X2000	显亮原先显亮的选项。
MN__SHOW__BAR	0X1000	如果使用了MN__PREV__BAR,则迫使亮条显示，即使它原来末显示。
MN__HIDE__BAR	0X0800	最初不显亮任何选项，取代MN__SHOW__BAR。
MN__UNKNOWN__BEEP	0X0010	当按下末分配键时鸣叫。
MN__UNKNOWN__TRANSMIT	0X0020	当按下末分配键时返回。
MN__ALL__TRANSMIT	0X0008	每次按键后返回。
MN__HOLD__BEEP	0X0080	返回时制止鸣叫，取代MN__UNKNOWN__BEEP。
MN__KEEP__HIGHLIGHT	0X0004	使最后的选项显亮。
MN__KEEP__DESCRIPTION	0X0040	不消除最后的说明字符串。
MN__REMOVE	0X0001	事后关闭菜单。
MN__DESTROY	0X0003	事后关闭并废弃菜单。

例如，下面这条语句读取用户对菜单的选择，开始时加亮位于第 1 行第 1 列处的菜单项。如果用户按下了一个未定义的键，则发出鸣叫。

```
mnread(pmenu,1,1,&row,&col,&ch,&key,MN__UNKNOWN__BEEP);
```

这个函数的返回值值得特别说明一下。如果没发生任何错误，只是由于用户按了回车键才返回，则返回值为 WN__NO__ERROR，即返回值为 0。如果发生下列情况之一，则出现一个错误，返回一个非 0 值:

菜单末显示;

菜单窗口数据区某一部分被另一个覆盖;

菜单中没有非保护菜单项;

指明了 MN__SHOW__BAR 而末发现可供选择的选项;

指明了 MN__UNKNOWN__TRANSMIT 而一个末分配的键被按下;

发生了具有 MN__ABORT 动作的按键或鼠标器操作;

指明了 MN__REMOVE 或 MN__DESTROY 而不能关闭该菜单;

指明了 MN_DESTROY 而不能废弃该菜单。

(2) 函数 mnlread 与函数 mnread 几乎一样，差别仅在于这个函数是用于 lotus 风格的菜单。其实，mnread 也可以显示 lotus 风格的菜单，只要在参数 option 中选择 MN_USE_DESCRIPTION 即可。因为 mnlread 函数肯定可以读 lotus 风格的菜单，所以在其参数 option 的可选值内也就没有 MN_USE_DESCRIPTION 这一项。

5.4.5 菜单应用举例

5.4.5.1 [例 1]：一个简单演示程序

```
/ * menu1.dem * /
#include <bmenu.h>
#include <conio.h>
char *thankyou   = "Thank you for selecting  ";
char *caption[3] = {"[A]  Alpha.","[B]  Beta。 ","[C]  Gamma."};
char *keys[3]    = {"Aa","Bb","Cc"};
void main( )
{
     int     width,i,row,col,ch,key;
     BMENU   *menu;
     WHERE  where = {SC_COLOR,0,{6,10}};
     BORDER border;
     width  = strlen(thankyou)+strlen(caption[0])+2;
     menu   = mncreate(6,width,NORMAL,REVERSE,NORMAL,NORMAL);
     col    = 11;
     for (i = 0; i < 3; ++i)
          mnitmkey(menu,i+1,col,MN_NOPROTECT,
               caption[i],keys[i],MN_TRANSMIT);
     border.type     = BBRD_SSSS | BBRD_TCT;
     border.attr     = NORMAL;
     border.ch       = 0;
     border.pttitle = " Make a choice! ";
     border.ttattr  = NORMAL | INTENSITY;
     mndsplay(menu,&where,&border);
     row = 1;
     while (mnread(menu,row,col,&row,&col,&ch,&key,MN_UNKNOWN_BEEP)
                                    == WN_NO_ERROR) {
          wncurmov(5,1);
          wnprintf("%s%s",thankyou,caption[row-1]);
     }
     mndstroy(menu);
     scpgcur(0,12,13,0);
     getch( );
}
```

这个例子比较简单，它不是一个实用程序，而只是演示了 Turbo C Tools 菜单处理的基本功能。程序执行以后，屏幕上产生的输出效果可能如图 5-1 所示。

```
--------- Make a choice! ---------

          [A]  Alpha.
          [B]  Beta.
          [C]  Gamma.

  Thank you for selecting  [C]  Gamma.
```

图 5-1 例 1 的输出

用户然后可以对菜单进行操作。按上下光标移动键将使加亮条在同一列的上下方向移动，按左右光标移动键将使加亮条在同一行的左右方向移动。行和列可以回绕。按<tab>和<shift_tab>键也将使加亮条从一个菜单项移到另一个菜单项，只是根据菜单项被安装的顺序，而不是根据菜单项在屏幕上的物理位置顺序，所以移动速度要明显慢一些。当然，几乎所有的应用软件中（包括这个例子），两个顺序是一致的，没有必要人为地造成不便。按<home>和<end>键将使加亮条分别移到第一个菜单项和最后一个菜单项，按回车键将确认某一个菜单项。在这个例子里，确认所引起的动作只是更新菜单窗口最后一行(这一行不属于菜单项)的显示内容，不产生别的结果。按<ESC>键结束这个程序的运行，注销菜单。

在本例中，用户也可以通过按一个字母键来选择相应的菜单项，字母不区分大小写。

Turbo C Tools 的菜单函数有一个特点，就是所有按键分配都属于整个菜单，而不是属于某一个菜单项，不能为某一个菜单项指定特殊的按键动作。以这个程序为例，无论加亮条位于哪一个菜单项上，按上移光标键都将使加亮条上移一个菜单项，按字母"A"或"a"都将选择第 1 个菜单项。在一般情况下，这个特点是合理而方便的，但在某些情况下，这个特点就成了一个限制。

5.4.5.2 [例 2]: 用菜单实现电子表格

```
/ * menu2.dem * /
#include   <math.h>
#include   <bkeybrd.h>
#include   <bmenu.h>
#include   <conio.h>
float   a = 1, b = 1;
float   x,y,u,v,d;
#define K 7
#define L 9
#define M 3
#define N 7
#define col(j)  (j * (K+L+M) + 1 )
#define vcol(j) (col(j)    + K )
typedef struct {
```

```
    int   row;
    int   col;
    int   vcol;
    char  caption[K+L+1];
    char  keys[3];
    float * vptr;
}celltype;
celltype cell[N] =
    {{0,col(0),vcol(0),"A   =             ","Aa",&a },
     {1,col(0),vcol(0),"B   =             ","Bb",&b },
     {0,col(1),vcol(1),"X   =             ","Xx",&x},
     {0,col(2),vcol(2),"U   =             ","Uu",&u},
     {1,col(1),vcol(1),"Y   =             ","Yy",&y},
     {1,col(2),vcol(2),"V   =             ","Vv",&v},
     {3,col(2),vcol(2),"D   =             ","" , &d }};

int cellindex(int row, int col)
{
    int i;
    for (i = 0; i < N; ++i)
        if (cell[i].row == row && cell[i].col == col) return i;
    return -1;
}

void display(celltype c)
{
    char s[80];
    sprintf(s,"% * .2f",L, * c.vptr);
    if (strlen(s) > L) strnset(s,' * ',L);
    wnwrbuf(c.row,c.vcol,L,s,-1,-1,CHARS_ONLY);
}

BMENU   * menu;
WHERE  where = {SC_COLOR,0,{5,5}};
BORDER border;
char    * digitsetc="0123456789+-.";
#define CR  13
#define ESC 27

void main( )
{
    int  i,ch,key,row,col;
    char s[L];
    menu = mncreate(4,col(3)-M+1,NORMAL,REVERSE,NORMAL,NORMAL);
    wnselect(menu->pwin);
    wnwrbuf(3,col(0),0,"Enter   ",-1,-1,CHARS_ONLY);
    for (i = 0; (ch = digitsetc[i]) != 0; ++i)
        mnkey(menu,0,0,ch,kbscanof(ch),MN_TRANSMIT,MN_ADD);
    for (i = 0; i < N; ++i)
        mnlitkey(menu, cell[i].row, cell[i].col,
                (i < 4 ? MN_NOPROTECT : MN_PROTECT),
```

```
                    cell[i].caption, 3, vcol(0), cell[i].caption,
                    cell[i].keys, MN_SELECT);
    border.type    = BBRD_SSSS | BBRD_TCT;
    border.attr    = NORMAL;
    border.ch      = 0;
    border.pttitle = " Distance D : (X,Y) to (U,V) ";
    border.ttattr  = NORMAL;
    mndsplay(menu,&where,&border);
    for (i = 0; i < 2; display(cell[i++]));
    row = 0;
    col = col(0);
    for (;;) {
        mnlread(menu, row, col, &row, &col, &ch, &key,
                MN_UNKNOWN_BEEP | MN_KEEP_HIGHLIGHT
                                | MN_KEEP_DESCRIPTION);
        if (ch == CR || ch == ESC) break;
        kbplace(KB_HEAD,ch,key);
        wncurmov(3,vcol(0)+K);
        wnquery(s,L,&ch);
        i = cellindex(row,col);
        * cell[i].vptr = atof(s);
        if (i <= 2)
            y = b * (1 - x / a);
        if (i == 3) v = u * u;
        d = hypot(x-u,y-v);
        for (i = 0; i < N; ++i)
            display(cell[i]);
    }
    mndstroy(menu);
    scpgcur(0,12,13,0);
}
```

这个例子相当复杂，但很实用，它不但演示了许多菜单函数的使用方法，而且演示了如何把菜单技术应用于电子表格之类的数据处理。

本例能显示 7 个变量的值，让用户输入某一变量的值，计算由此而引起的有关变量的新值，重新显示这些变量的值。这 7 个变量的名字是 A、B、X、Y、U、V、和 D，它们通过如下三个方程式相关联

$$X / A + Y / B = 1$$

$$V = U^2$$

$$D^2 = (U-X)^2 + (V-Y)^2$$

大体上可以把 A 和 B 看作可变的常数，(X,Y)和(U,V)是平面上两个点的坐标，(X,Y)在一条直线上移动，(U,V)在一条抛物线上移动，D 是这两个动点间的距离。用户可以改变变量 A、B、X 和 U 的值，但不可改变 Y、V 和 D 的值，后 3 个变量的值是根据前 4 个变量的值计算出来的。

程序把 7 个变量处理成具有 lotus 风格的菜单项，其屏幕显示效果可能如图 5–2 所示:

Distance D: (X,Y) to (U,V)					
A =	12.34	X =	1234.56	U =	−789.01
B =	−56.78	Y =	5623.80	V =	622536.81
Enter U =				D =	616916.31

图 5-2

菜单窗口的最下面一行用于提示并接收用户输入的变量值。程序通过语句:

```
mnlitkey(menu, cell[i].row, cell[i].col,
        (i < 4 ? MN_NOPROTECT : MN_PROTECT),
        cell[i].caption, 3, vcol(0), cell[i].capll[i].col,
        (i < 4 ? MN_NOPROTECT : MN_PROTECT),
        cell[i].caption, 3, vcol(0), cell[i].caption,
        cell[i].keys, MN_SELECT);
```

把相应于变量 A、B、X、U 的 4 个菜单项设置成未加保护的，以便用户进行选择。把相应于变量 Y、V、D 的 3 个菜单项设置成加以保护的，用户不可进行选择。因为这条语句中把键盘行为设置为 MN_SELECT，所以按"Aa"、"Bb"、"Xx"、"Uu"键就选择相应的变量(菜单项)。再一次强调，选择不等于确认，MN_SELECT 不隐含有 MN_TRANSMIT。但在本例里，也不是用回车键来确认，回车键(和 ESC 键)被用来结束程序的运行。仅当用户真正开始输入变量的值时才表示确认，才在菜单窗口的最下面一行显示相应变量的名字及等号，如"X=　　"，然后进一步接收用户输入的数字字符。

所有键盘输入都移到窗口的最下面一行来接收并回显，显然这使画面简洁，用户操作更容易，更不易出错。这最下面一行的字符"Enter　"是通过普通的窗口写函数 wnwrbuf 实现的，后面的字符则是随用户的选择(确认)而变化的，并作为每个菜单项(每个变量)的长描述文本而显示出来，所以程序中用的是 mnlitkey 而不是 mnitmkey 来设置各个菜单项，并用 mnlread 而不是 mnread 来读取用户的响应。每个菜单项的长描述文本都一样长，并在同一个坐标位置显示出来，其内容都与各自的短描述文本相同。这只要仔细地分析一下 mnlitkey 语句就可以看出来。

更新电子表格中的数据项存在着一定的困难，Turbo C Tools 中并没有提供比较直接的解决此类问题的办法。在这类应用程序中，这些数值变量往往是某些更大的数据结构(如矢量和矩阵)的一部分，它们一般是由其它进行科学计算的函数独立处理的，与用户接口往往没有什么关系。程序员需要一种比较方便的办法，把屏幕的位置（或者菜单项的坐标）和相应的数值变量联系起来。由于 Turbo C Tools 中没有提供这样一种直接办法，所以，在这个例子里用的是一个数组 cell，这个数组的每一个元素都是一个结构 celltype 变量，相应于一个菜单项。结构内含有如下 6 项信息：这个菜单项的行号和列号，开始显示它的数值变量的起始列号，菜单项的短描述文本(同时也是长描述文本)，选择这个菜单

项的字母键，指向它的数值变量的指针。在这些信息中，有些项是为了通过 mnlitkey 函数安装菜单项时所需要的。lotus 风格的长描述文本在这里不用于提供附加的详细解释，而是用于数据输入，减少数据输入过程中的错误。描述文本中含有许多空格，这是为了使加亮条加亮菜单项中整个数值字符串。

数组 cell 使用户接口与数学处理部分相隔离。因为 celltype 结构中含有指向数值变量的指针，所以，可以在屏幕上显示一个数值变量的值而不需要知道变量的名字。在这个例子里，这是通过函数

```
void  display(celltype c )
```

来实现的。这个函数准确计算出显示的屏幕位置，不与其它变量或边界发生冲突。

如果知道用户从键盘上选择的变量在数组 cell 中的索引地址，那么找到这个菜单项也就不难了。遗憾的是，从函数 mnread 和 mnlread 返回的是用户所选菜单项的坐标，而不是这个菜单项在总菜单中的索引，所以，还必须通过函数

```
int  cellindex(int row,int col)
```

进行这种转换。程序中用 Turbo C Tools 的高级键盘输入函数 wnquery 来读取用户敲入的数字字符串，再通过 Turbo C 函数 atof 把这个 ASCII 数字字符串转换成浮点数。然后，程序通过函数 cellindex 找到相应的菜单项，把转换好的浮点值存入相应的变量中。一旦存入了一个新的变量值，程序就更新有关的各个变量值，并通过函数 display 在屏幕上更新整个电子表格。

下面，我们再反过来看看如何在选择(加亮)一个菜单项后，按一个数字键既起到确认的作用，又把敲入的这个数字字符作为 ASCII 数字字符串的第 1 个字符。程序通过语句

```
char    * digitsetc = "0123456789+-.";
```

和语句

```
for (i = 0; (ch = digitsetc[i]) != 0; ++i)
        mnkey(menu,0,0,ch,kbscanof(ch),MN_TRANSMIT,MN_ADD);
```

设定：用户按下的每个数字键，包括 10 个数字以及加号、减号和小数点，都能引起 MN_TRANSMIT 行动，即引起从 mnread 和 mnlread 函数返回。根据 5.4.1 节中的缺省规定，回车键和 ESC 键也能引起从 mnread 和 mnlread 函数返回。所以，程序通过下列语句

```
mnlread(menu, row, col, &row, &col, &ch, &key,
        MN_UNKNOWN_BEEP | MN_KEEP_HIGHLIGHT
                        | MN_KEEP_DESCRIPTION);
if (ch == CR || ch == ESC) break;
```

把回车键和 ESC 键引起的返回作为退出程序处理，而把由数字键(包括加号，减号和小数点)引起的返回作为输入一个数字字符串的开始。通过函数 kbplace 把这个数字字符送回到缓冲区的头部，通过函数 wncurmov 把光标移到菜单窗口的最下面一行，通过函数 wnquery 进一步接收用户的键盘输入。函数 wnquery 是一个高级窗口输入函数，允许在输入过程中通过回退键等进行比较简单的编辑，直至用户敲入回车键。因为此时是由函数 wnquery 接管控制权，所以回车键只是结束字符串的输入，而不是终止程序的运行。

这个程序所要完成的任务应该说是比较复杂的，如果不利用 Turbo C Tools 的菜单处理函数，这个程序写起来就长且麻烦多了。由此可见，如果能熟练地掌握建立菜单、显示菜单、选择菜单项等许多技术，那么编写一个复杂的交互式输入／输出程序不是一件很困难的事。

Turbo C Tools 的菜单函数也还存在一些缺陷:

第 1，同一菜单项内描述文本的显示属性必须一样，不可使用不同的颜色。这个缺陷可以通过把描述文本从只含 ASCII 码的字符串改变成含(ASCII 码，显示属性)偶对来补救，这不很困难。

第 2，前面已经说过，键盘定义对整个菜单是一致的，不可能为某个菜单项指定只属于这个菜单项的键盘定义。因此，也就不大可能用 lotus 风格长描述文本作为子菜单。这个缺陷不容易补救，它几乎要求重写所有菜单处理函数。

第 3，Turbo C Tools 的 mnread 和 mnlread 函数返回的只是用户所选菜单项的坐标，而不返回这个菜单项在菜单中的索引号。如果能够返回这个索引号，这个程序中的 cellindex 函数就不需要了。

5.5 Turbo C Tools 的域编辑

5.5.1 概述

Turbo C Tools 6.0 除加强了各个键盘函数的功能外，还提供了具有域编辑功能的函数，这在一般语言中是没有的。所谓域编辑，就是对屏幕或窗口上的一个矩形区域进行编辑。这就是说，只要调用一个域编辑函数，就可以接受用户的一大串键盘输入，这些输入回显在屏幕或窗口的一个矩形区域内，在输入的过程中可以进行比较简单的编辑，从函数返回时(按缺省规定，按回车键后才返回)得到的是编辑后的结果。

Turbo C Tools 的域编辑功能具有下列特性:

(1) 输入区域中可以有多个字符行;

(2) 保证字不跨行;

(3) 允许左对齐、右对齐或居中;

(4) 有插入和替换两种方式，可以用不同的光标形状来区分;

(5) 通过函数 wnfield 能完全支持 Turbo C Tools 窗口，利用虚拟窗口可以编辑一个比显示数据的视口更大的缓冲区;

(6) 可用初始值预先装载缓冲区;

(7) 支持键控制函数(见 5.3.5 节)，这就可以完全控制每一个按键，也可以在任何时候以任何方式修改输入的数据;

(8) 可以定义每一个按键所引起的动作。

与菜单窗口一样，为了编程方便，Turbo C Tools 为所有要用到的编辑动作定义了相应的符号常数，这些定义放在枚举数据类型 ED_ACTION 中。ED_ACTION 的定义在头文件 bedit.h 中。

```
typedef enum
```

```
{
    ED_NULL,                      没有动作
    ED_ABORT,                     不编辑，退出
    ED_BEEP,                      使扬声器发声
    ED_ATTR,                      改变属性
    ED_ASCII,                     显示字符
    ED_LEFT,                      左移
    ED_RIGHT,                     右移
    ED_UP,                        上移
    ED_DOWN,                      下移
    ED_FRONT,                     移到缓冲区的开头
    ED_END,                       移到缓冲区的末尾
    ED_TEXT_END,                  移到最后非空白字符处
    ED_WRAP,                      使缓冲区整字换行
    ED_REDUCE,                    压缩空白字符
    ED_NEXT_WORD,                 移到下一个字
    ED_PREV_WORD,                 移到上一个字
    ED_BEGIN_WORD,                移到字的开始位置
    ED_END_WORD,                  移到字的结束位置
    ED_NEXT_WHITE,                移到下一个空白字符处
    ED_PREV_WHITE,                移到上--个空白字符处
    ED_NEXT_NONWHITE,             移到下一个非空白字符处
    ED_PREV_NONWHITE,             移到上一个非空白字符处
    ED_DELETE,                    删除当前字符
    ED_RUBOUT,                    左移，删除字符
    ED_DELLEFT,                   删除左边字符
    ED_DEL_WORD,                  删除1个字
    ED_CLEAR,                     删除整个缓冲区
    ED_CLEAREOL,                  删除到本行行尾
    ED_CLEAREOB,                  删除到缓冲区末尾
    ED_INSERT,                    插入方式
    ED_REPLACE,                   替换方式
    ED_INSTOGGLE,                 切换插入／替换方式
    ED_UNDO,                      废除所做的编辑工作
    ED_TRANSMIT,                  完成编辑并返回
    ED_INVALID_ACTION             无效的编辑动作
} ED_ACTION;
```

用户每按一个按键，可以引起一个或多个动作。对于一些常用的编辑键，函数 edinitky 建立了一个缺省的键分配表，如下所示。在这个表中，每一个键只引起一个动作。

缺省的编辑键分配

键		ASCII 码	扫描码	动作
Esc		0X1B	0X01	ED_ABORT
Up	(数字小键盘)	0X00	0X48	ED_UP
Up	(增强键)	0XE0	0X48	ED_UP
Down	(数字小键盘)	0X00	0X50	ED_DOWN
Down	(增强键)	0XE0	0X50	ED_DOWN
Right	(数字小键盘)	0X00	0X4D	ED_RIGHT
Right	(增强键)	0XE0	0X4D	ED_RIGHT
Left	(数字小键盘)	0X00	0X4B	ED_LEFT
Left	(增强键)	0XE0	0X4B	ED_LEFT
Home	(数字小键盘)	0X00	0X47	ED_FRONT
Home	(增强键)	0XE0	0X47	ED_FRONT
End	(数字小键盘)	0X00	0X4F	ED_END
End	(增强键)	0XE0	0X4F	ED_END
Ctrl-End		0X00	0X75	ED_TEXT_END
Ctrl-F		0X06	0X21	ED_NEXT_WORD
Ctrl-A		0X01	0X1E	ED_PREV_WORD
Ctrl-T		0X14	0X14	ED_DEL_WORD
-	(数字小键盘)	0X2D	0X4A	ED_BEGIN_WORD
+	(数字小键盘)	0X2B	0X4E	ED_END_WORD
F2		0X00	0X3C	ED_NEXT_WHITE
F1		0X00	0X3B	ED_PREV_WHITE
F4		0X00	0X3E	ED_NEXT_NOWHITE
F3		0X00	0X3D	ED_PREV_NOWHITE
Del	(数字小键盘)	0X00	0X53	ED_DELETE
Del	(增强键)	0XE0	0X53	ED_DELETE
Backspase		0X08	0X0E	ED_RUBOUT
Ctrl-H		0X08	0X23	ED_DELLEFT
Ctrl-Home	(数字小键盘)	0X00	0X77	ED_CLEAR
Ctrl-Home	(增强键)	0XE0	0X77	ED_CLEAR
Ctrl-Right	(数字小键盘)	0X00	0X74	ED_CLEAREOL
Ctrl-Right	(增强键)	0XE0	0X74	ED_CLEAREOL
Alt-E		0X00	0X12	ED_INSERT
Alt-R		0X00	0X13	ED_REPLACE
Ins	(数字小键盘)	0X00	0X52	ED_INSTOGGLE
Ins	(增强键)	0XE0	0X52	ED_INSTOGGLE
Alt-X		0X00	0X2D	ED_UNDO
Enter		0X0D	0X1C	ED_TRANSMIT
Enter	(数字小键盘)	0X0D	0XE0	ED_TRANSMIT

既然是对域进行编辑，则还需要一个结构来说明域的特征。这个结构的名字是ED__CONTROL，其定义如下：

```
typedef struct
{
    LOC              ul__corner;
    DIM              dimensions;
    int              attribute;
    CUR__TYPE        replace__cursor;
    CUR__TYPE        insert__cursor;
    PKEY__CONTROL    control__function;
    unsigned int     control__flags;
} ED__CONTROL;
```

字段 ul__corner 说明域的左上角在屏幕或窗口中的坐标位置。字段 dimensions 是说明域的高度和宽度。字段 attribute 是说明域的显示属性。如果为 0，则使用屏幕上的现有属性。字段 replace__cursor 和 insert__cursor 分别说明替换状态和插入状态时光标的形状。字段 control__function 指向一个键控制函数，每次查询键盘时该函数就被调用。控制函数的调用参数是指向结构 KB__DATA 的指针，而结构 KB__DATA 的字段 pfunction__data 又是指向结构 ED__BUFFER 的指针，见 edbuffer 函数。字段 control__flags 说明编辑过程中以及退出域编辑函数时的一些其它特性：

符　号	值	意　义
ED__LEFTJUST	0X8000	退出时左对齐
ED__CENTERJUST	0X4000	退出时中对齐
ED__RIGHTJUST	0X2000	退出时右对齐
ED__ZERO__FILL	0X1000	退出时以0填充
ED__BEEP__END	0X0800	在域的末端鸣叫
ED__AUTOSKIP	0X0400	在域的末端传送
ED__INSERT__MODE	0X0200	开始时为插入方式
ED__WORD__WRAP	0X0100	整字换行
RCONTROL	0X0080	退出时删除全部控制字符
TOLOW	0X0040	退出时将大写字母转为小写字母
TOUP	0X0020	退出时将小写字母转为大写字母
NOQUOTE	0X0010	退出时不修改括在单引号或双引号中的字符
REDUCE	0X0008	退出时将一片连续空白字符压缩成一个空格
RTWHITE	0X0004	退出时删除全部尾随空格字符
RLWHITE	0X0002	退出时删除全部前导空格字符
RWHITE	0X0001	退出时删除全部空白

在域编辑的各函数中，还经常用到结构 KEY__SEQUENCE，它用来存放一次按键的 ASCII 码和扫描码，其定义如下：

```
typedef struct {
    unsigned char character__code;
    unsigned char key__code;
```

```
}KEY_SEQUENCE;
```

5.5.2 域编辑

域编辑函数共有下列三个：

```
ED_ACTION cdecl  edbuffer(ED_BUFFER  *pbuffer,const ED_KEY  *pcommand);
  int      cdecl  edfield(const char  *pinitstr,char  *pretstr,
                          int retsize,const ED_CONTROL  *pctrl,
                          KEY_SEQUENCE  *pfinal);
  int cdecl  wnfield(BWINDOW  *pwin,char  *pinitstr,char  *pretstr,
                     int size,ED_CONTROL  *pctrl,
                     KEY_SEQUENCE  *pfinal);
```

(1) 函数 edfield 对屏幕上的一个域进行编辑。参数 pinitstr 指向一个 ASCIIZ 串，编辑开始时在域中显示这个字符串。参数 pretstr 指向一个缓冲区，用来存储编辑后的结果，参数 retsize 说明这个缓冲区的容量，结尾 NUL('\0')占用一个字节。参数 pctrl 指向说明编辑域特性的结构 ED_CONTROL，其定义已在上一小节中介绍。参数 pfinal 指向结构 KEY_SEQUENCE，这个结构用来返回用户最后一次按键的 ASCII 码和扫描码。

调用 edfield 函数后，用户就可以进行编辑了。按照上一小节中的缺省规定，用户可以使用许多编辑键，直到按一个回车键或 ESC 键才返回。返回值可以是下列值之一：

符 号	值	意 义
ED_NO_ERROR	0	成功
ED_NO_MEMORY	1	动态内存不够
ED_ILL_DIM	2	pctrl->dimensions中的域尺寸非法
ED_NO_KEY	300	在队列中未发现对应的按键
ED_KEY_EXISTS	301	键已经存在
ED_USER_ABORT	302	用户放弃编辑，不返回数据

(2) 函数 wnfield 几乎与上一个函数完全一样，只是它是对窗口而不是对屏幕中的一个域进行编辑，所以它的调用参数中多了一个指向结构 BWINDOW 的指针 pwin。

(3) 函数 edbuffer 不处理任何输入／输出操作，它不是让用户对屏幕或窗口中的一个矩形区域进行编辑，而是让程序对内存中的一个缓冲区进行编辑。它有两个调用参数，一个是指向结构 ED_BUFFER 的结构指针，说明被编辑的缓冲区；另一个是指向结构 ED_KEY 的结构指针，提供一张编辑操作表，说明如何进行编辑。一般情况下这个函数用处不大。但是，Turbo C Tools 提供了在查询键盘时自动调用键控制函数的功能。因此，如果在键控制函数中再调用 edbuffer 函数，就相当于在程序中对缓冲区进行编辑，这可以提供比一般所说的键盘宏定义更为强的编辑功能。下面给出的例子就说明了这一点。

结构 ED_BUFFER 说明被编辑的缓冲区的特性，其定义如下：

```
typedef struct
{
    int  h,w;
```

```
}DIM;
typedef struct
{
    char            * pbuffer;
    DIM             dimensions;
    int             attribute;
    int             buffer__size;
    int             edit__pos;
    int             data__end;
    unsigned int    control__flags;
} ED__BUFFER;
```

字段 pbuffer 指向被编辑的缓冲区，buffer__size 说明这个缓冲区的大小，edit__pos 说明从哪一个位置开始编辑，data__end 说明缓冲区中当前有多少个字符。当从函数 edbuffer 返回时，参数 edit__pos 和 data__end 的值都可能被修改了，字段 dimensions 说明域的尺寸，attribute 说明这个域内的显示属性。初看起来，对缓冲区的数据进行编辑不应牵涉到域的问题。但实际上，edbuffer 函数往往是在键控制函数中调用的，而键控制函数又是执行 edfield 或 wnfield 函数时调用的，这时被编辑的缓冲区对应于屏幕或窗口上的一个矩形区域。如果不给定域的宽度和高度，且指定编辑动作是上移 1 行(ED__UP)或下移 1 行(ED__DOWN)，则在缓冲区中就不知道该移多少个字符。缓冲区的大小至少应等于 (dimensions.h * dimensions.w+1)，这样才能放下最后的 NUL('\0')字节。字段 control__flags 与 5.5.1 节的 ED__CONTROL 结构中的字段 control__flags 意义相同，但在 edbuffer 函数中，这个字段只能取 ED__INSERT__MODE 和 ED__WORD__WRAP 这两个值。

结构 ED__KEY 说明编辑动作系列，其定义如下：

```
typedef struct
{
    int             num__actions;
    ED__ACTION      * pactions;
} ED__ACTION__LIST;
typedef struct
{
    unsigned char character__code;
    unsigned char key__code;
}KEY__SEQUENCE;
typedef struct
{
    ED__ACTION__LIST  edit__actions;
    KEY__SEQUENCE     key__sequence;
    int               attribute;
} ED__KEY;
```

字段 edit__actions 是一张编辑操作表，第 1 个整数说明一共要进行多少个操作，其后就是各个操作，每个操作都是枚举数据类型 ED__ACTION 的成员，见 5.5.1 节。如果操作是 ED__ASCII,则字段 key__sequence.character__code 就是加到缓冲区中的字符。如

果操作是 ED__ATTR，则字段 attribute 就是赋予缓冲区的新显示属性。

下面是一个应用 edbuffer 函数的例子。

```
#include <string.h>
#include <bedit.h>
#include <bkeybrd.h>
#include <bscreens.h>
void my__key__ctrl(KB__DATA *);
/*   first section   */
ED__BUFFER my__buffer;
ED__KEY     my__command;
static ED__ACTION my__actions[ ]={ED__FRONT,ED__DEL__WORD,ED__DEL__WORD};
my__command.edit__actions.num__actions=3;
my__command.edit__actions.pactions    =my__actions;
edbuffer(&my__buffer,&my__command);
/*   second section  */
void my__key__ctrl(KB__DATA *pdata)
{
     ED__KEY            my__command;
     static ED__ACTION my__actions[ ]={ED__ATTR};
     if(   pdata->control__action != KB__FLUSH
        && pdata->key__found
        && pdata->key__seq.character__code == KB__C__A__F1
        && pdata->key__seq.key__code == KB__S__A__F1)  {
          strupr(((ED__BUFFER *)pdata->pfunction__data)->pbuffer);
          my__command.edir__actions.num__actiond=1;
          my__command.edit__actions.pactions=my__actions;
          my__command.attribute=REVERSE;
          edbuffer(pdata->pfunction__data,&my__command);
          pdata->key__found=0;
          pdata->returned__action=KB__REMOVE__KEY;
     }
}
```

这个例子分为两部分。第 1 部分包括 include 语句后的 7 条语句，它对缓冲区顺序执行 3 个编辑操作：将光标移到缓冲区的开头，删掉 1 个字，再删掉 1 个字。第 2 部分是键控制函数 my__key__ctrl。它是设计成供函数 edfield 和 wnfield 使用的。前面已经说过，在查询键盘时，键控制函数会获得控制权。在获得控制权后，这个函数检查用户是否敲入了 Alt__F1。如果是，则它首先调用 Turbo C 函数 strupr 把整个缓冲区的小写字母变为大写字母，随后调用这里介绍的 edbuffer 函数对缓冲区执行编辑操作 ED__ATTR，把整个缓冲区置为反显(REVERSE)。键控制函数接收指向结构 KB__DATA 的指针作为参数，在结构 KB__DATA 中还有一个字段 pfunction__data，这个字段又是指向结构 ED__BUFFER 的指针。

5.5.3 编辑键定义

Turbo C Tools 提供了如下 5 个编辑键定义函数：

```
int       cdecl   edchgkey(const KEY_SEQUENCE  * pkey_seq,
                           const ED_ACTION_LIST  * pedits);
int       cdecl   edinitky(void);
int       cdecl   edremkey(const KEY_SEQUENCE  * pkey_seq);
ED_KEY  * cdecl   edretkey(const KEY_SEQUENCE  * pkey_seq);
void      cdecl   edzapkey(void);
```

(1) 函数 edinitky 为函数 edfield 和 wnfield 建立缺省的键分配表，如 5.5.1 节所述。函数 edfield 和 wnfield 会自动调用 edinitky，程序员一般不需要调用这个函数。

(2) 函数 edretkey 返回一个按键所对应的编辑操作。调用参数是指向结构 KEY_SEQUENCE 的指针，说明键的 ASCII 码和扫描码。返回值是一个指向结构 ED_KEY 的指针，这个结构存放着对应于这个键的编辑操作表。下面是在一个键控制函数内应用 edretkey 函数的例子。

```
#include <stdio.h>
#include <bedit.h>
extern int valid_data(const char * );
void my_key_ctrl(KB_DATA  * pdata)
{
     const ED_KEY  * pactive_key;
     int  i;
     if(pdata->control_action != KB_FLUSH && pdata->key_found) {
          pactive_key = edretkey(&pdata->key_seq);
          if(pactive_key != NIL) {
               for(i = 0;i < pactive_key->edit_actions.num_actions;
                                   i++) {
                    if(ED_TRANSMIT ==
                      pactive_key->edit_actions.pactions[i]) {
                         if(!valid_data(((ED_BUFFER  * )
                           pdata->pfunction_data)->pbuffer)) {
                              sctty wrt('\a');
                              pdata->key_found = 0;
                              pdata->returned_action =
                                             KB_REMOVE_KEY;
                         }
                         break;
                    }
               }
          }
     }
}
```

这个键控制函数是供函数 edfield 或 wnfield 调用的。它在获得控制权后，通过传来的数据结构 KB_DATA 的某些字段知道最近一次按了什么键，通过函数 edretkey 知道这个键曾引起一些什么编辑操作，再查这些编辑操作中是否有引起确认传送的操作 (ED_TRANSMIT)。如果有，则说明一个数据项已输入完毕，再通过函数 valid_data

(这个例子里未列出)检查缓冲区中的数据是否非法。如果非法，则产生一声鸣叫，并请求调用者放弃最近的这次按键。

(3) 函数 edchgkey 添加或修改 edfield 和 wnfield 函数所认可的一个键及这个键所引起的编辑操作。

(4) 函数 edremkey 删除 edfield 和 wnfield 函数所认可的一个键。

(5) 函数 edzapkey 删除 edfield 和 wnfield 函数所认可的整个按键操作表。

第6章　基本文件处理

Turbo C 函数库中最令人感到混乱的部分就是它的文件输入／输出。像大部分 C 编译程序一样，Turbo C 提供了两种不同级别的文件存取方法，提供了丰富的文件处理函数。程序员的难处不在于找出一个函数来满足他的需要，而在于被众多的功能相似的函数所迷惑，不知道应该选用哪一个。在这方面，Turbo C 用户手册提供的帮助很少，因为它几乎没有明确说明有两种不同级别的方法，更没说明这两级方法一般不应混合使用。事实上，对整个文件处理，Turbo C 用户手册说得很少。这一章的目的就是帮助读者弄清楚这方面的问题。

文件处理的两级方法就是标准级和系统级。所谓标准级，就是这些函数更符合 C 语言的普通标准，与其它机器上的 C 语言更兼容，更容易移植到其它机器其它操作系统上，对程序员来说标准级函数也更容易使用，更高级，功能更强。另外，由于其内部带有缓冲能力，故磁盘存取的次数较少，效率可能更高。标准级函数有时也称为流式输入／输出函数，因为在使用这些函数时，经常把被操作的文件看作是一个字符流。所谓系统级，就是这些函数往往直接调用操作系统，在 PC 机上就是 DOS。正因为如此，系统级函数速度更快，内存占用更少，但它们的兼容性更差。对程序员来说，系统级函数使用起来更难，因为它们功能较差，最明显的是没有自动缓冲的能力，也没有格式输入／输出的能力。

下面列出对文件处理函数进行分类的表格。

标准级和系统级函数对照表

标准级	系统级	标准级	系统级
–	chsize	–	creattemp
clearerr	–	fprintf	–
–	dup	fputc	–
freopen	dup2	fputs	–
fclose	close	fread	read
–	_close	–	_read
fcloseall	–	freopen	–
fdopen	–	fscanf	–
feof	eof	fseek	lseek
ferror	–	fsetpos	–
fflush	–	fstat	fstat
fgetc	–	ftell	–
fgetchar	–	fwrite	write

标准级	系统级	标准级	系统级
fgetpos	–	–	_write
fgets	–	rewind	–
fileio	–	setbuf	–
flushall	–	setvbuf	–
fopen	open	stat	stat
–	_open	ungetc	–
–	_creat	vfprintf	–
–	creat	vfscanf	–
–	creatnew	strerror	–
		perror	–

标准级函数有时被错误地理解为只提供流式输入／输出，即把文件看作字符流或字符序列。虽然大部分情况下是这样用的，但这些函数的功能却不只是这些。例如，可以通过标准级函数进行格式化的输入／输出，可以对文件进行随机存取，也可以按块或按记录对文件进行存取。虽然标准级函数一般用来处理文本文件，系统级函数一般用来处理二进制文件，然而却没有任何理由认为标准级函数只能处理文本文件。

Turbo C 文件处理函数的引用说明及其有关的定义分散在 7 个头文件中：

io.h	dir.h	dos.h	fcntc.h
stdio.h		errno.h	stat.h

这 7 个头文件中，左边 3 个是最常用的，最重要的。分散为 7 个头文件未必很好，相反，在一定程度上却表现出组织不当。io.h 头文件中包含的大多是系统级函数，stdio.h 头文件中包含的大多是标准级函数，dir.h 头文件中包含的大多是目录处理函数。

本章在第 1 节中对文件输入／输出系统的一些共同问题作了一个概述，为后几节作准备，第 2 节和第 3 节分别详细讨论了主要的系统级和标准级函数，给出了一些小例子，以便比较这两级之间的区别。第 4 节讨论了一些比单个文件操作更加复杂的问题，如目录和卷。

6.1 目录／文件系统概述

6.1.1 文件存取级别

为了应付各种不同的情况，程序员可能需要在不同的级别上来建立和存取文件。粗略地看，大致可以抽象为 4 级，如表 6–1 所示。

最低级 Turbo C 文件处理函数有 ioctrl、biosdisk、absread 和 abswrite 等几个。ioctrl 的某些特性下面还会讨论到，其它 3 个函数就不再叙述了，因为程序员一般都不使用这一级函数。

低级 Turbo C 文件处理函数是 6.2 节讨论的重点。函数 randbrd 和 randbwr 虽然也属于这一级，但它们是基于绝对的 DOS 文件控制块(FCR)机制的，Microsoft 公司建议

不要使用它们，这里也不讨论了。

表 6-1 文件存取级别

	级 别	文件存取方法
最高级	文件由高度结构化的记录序列组成	面向数据处理高级语言。Turbo C 语言中的标准级函数只支持定长记录，不定长记录要求程序员综合运用各种技巧去处理
高级	文件由字节序列组成，具有缓冲、格式化和安全保护特性	Turbo C 语言标准函数
低级	文件由字节序列组成，略具有缓冲和安全保护特性	DOS 的文件柄服务。Turbo C 语言的系统级函数
最低级	文件由字节或字节块组成,直接或通过特殊的硬件进行存取	BIOS 服务。低级的与硬件有关的 DOS 服务

高级 Turbo C 文件处理函数在 6.3 节讨论，也是本章的重点。它是实际编程中使用最广的一级。

最高级 Turbo C 文件处理函数只有两个，也在 6.3 节讨论。对于那些通过函数库内的基本函数不能进行处理的复杂文件存取问题，则放在第 9 章专门讨论.

6.1.2 文件属性

读者对 DOS 的文件组织方法和目录树结构可能是很熟悉的。每个 DOS 文件在其目录项中都包含 1 个属性字节，其各位定义如下：

7	6	5	4	3	2	1	0
不用	不用	档案	目录	卷标	系统	隐藏	只读

应该注意，尽管 DOS 允许一个文件具有多种属性,但这 6 种属性中有些是彼此不兼容的，不能共存。

只读：DOS 将拒绝修改或删除只读文件。

隐藏和系统：在目前的 DOS 下不加区分，所有目录操作将不对具有这些属性的文件进行处理。

卷标：只能在根目录下，表示这个文件名是此磁盘卷的卷标号，而不是一个真正的文件。

子目录：表示这个文件是一个具有特殊格式的文件，文件的内容就是一个个目录项。

档案：DOS 的 Backup 实用程序置这位为 0，一旦对这个文件进行修改,则 DOS 又将这位置为 1。目的是帮助后备程序区别：自上次后备以来，哪些文件已经修改了，哪些文件还未被修改过。

每个目录项中也有一个称为"时间邮戳"的字段，占 4 个字节，记录下文件建立或最后一次被修改的日期和时间。此外，还有一个文件长度字段，也占 4 个字节。应该注意，文件长度的计算办法有多种，彼此不完全一致，适用不同场合。在用 DIR 命令列文件目录

时，看到的是目录项中记载的文件长度，它反映的是实际往文件中写的字节数。在 DOS 分配文件空间时，用的是"簇"的概念。簇是 DOS 文件分配的最小单位，是 512 字节的整数倍，随盘容量的大小及 DOS 版本不同而不同。另外，很多字处理软件以及某些 Turbo C 库函数把<Ctrl-Z> 看作文件结束标志，并依此来计算文件长度，这往往与 DOS 文件目录项中记载的文件长度不一致。

6.2 系统级输入 / 输出

6.2.1 文件柄

系统级文件输入／输出都是通过文件柄来参照一个文件的。文件柄是一个整数，DOS 用它来区别各个已打开的文件。这个整数的大小是从 0～N ，DOS 允许 N 的最大值为 255，N 的具体值由 CONFIG.SYS 文件中的 FILES 命令决定，如果没有 CONFIG.SYS 文件或没有 FILES 命令，则缺省值是 8。因为 DOS 自身占用了 5 个，所以只剩下 3 个给应用程序。对于许多应用场合，同时打开的文件数往往超过 8 个。Turbo C 认为文件柄应在 0～20。所以，为了完全与 Turbo C 兼容，应在 CONFIG.SYS 文件中设置 N 为 20。

打开一个文件意味着给这个文件指定一个文件柄，并初始化各种有关的数据结构，准备进行输入／输出。关闭一个文件意味着保留存放在这些数据结构中的有关信息,释放文件柄，以便指定给另一个文件。

对于程序员来说，最关心的是：通过文件柄能得知哪些文件特性和能对文件进行哪些操作。

DOS 为每一个已打开的文件保存了一份表格，记录着与该文件有关的一些信息。这张表格的结构包含在头文件 sys\stat.h 的结构 stat 内，如下所示：

```
struct stat {
      short   st_dev;
      short   st_ino;
      short   st_mode;
      short   st_nlink;
      int     st_uid;
      int     st_gid;
      short   st_rdev;
      long    st_size;
      long    st_atime;
      long    st_mtime;
      long    st_ctime;
}
```

这个结构的大部分字段是为了与 UNIX 相兼容而提供的，DOS 根本不用。在系统级，这个结构可以通过 Turbo C 函数 fstat 读取到，从而可以得知这个结构中包含的信息。

另外，借助其它函数，程序员还可以通过文件柄存取如下一些文件特性：

(1) 文件本身:字节系列。

(2) 文件指针:当前读写的位置。

(3) 文件指针是否在文件尾。

(4) 文件的日期和时间。

(5) 文件长度。

(6) 文件属性。

(7) 文件在哪一个驱动器上。

(8) 文件是一个普通的磁盘文件还是一个输入／输出设备。

应该注意，文件名是不能通过文件柄来存取的，程序一旦根据文件名打开一个文件并获得文件柄后，就再也不能根据这个文件柄得到文件原来的名字。这有时使程序员很难编写出用户界面友好的实用程序，尤其是在报错时。例如，在读某文件时出错，如果知道文件名，就可以报"某某文件(名字)错"；否则，只能报"某号(只有程序员知道的无意义的号码)文件错"。这个问题虽然可以解决，但要费点周折。

当执行程序时，DOS 自动打开 5 个文件，占用头 5 个文件柄:

文件柄	设备名	一般实际连接的设备	DOS 赋予的文件名
0	标准输入	键盘	CON
1	标准输出	显示	CON
2	标准错误	显示	CON
3	标准辅助	串行口 1	AUX
4	标准打印	并行口 1	PRN

这 5 个文件柄也可以赋给别的设备或文件。事实上，经常把标准输入和标准输出重定向到普通磁盘文件，把标准打印重定向到标准辅助设备，即串行口。应该注意，尽管程序有办法知道标准文件柄已经被重定向到盘文件上，甚至也可以知道文件在哪一个磁盘驱动器上，但却无法知道它的输入或输出文件的名字。

通过文件柄能对文件进行如下操作：移动文件指针，读／写／关闭文件。磁盘文件的输入／输出总是带有缓冲的：DOS 读写磁盘文件总是以扇区为单位。根据 CONFIG.SYS 文件中所设置的缓冲(BUFFERS)大小，最近读入的一些扇区总是存放在内存中，以便较快地重复存取。DOS 规定一个缓冲区(buffer)存储 1 扇，但有些外存设备，例如虚拟盘，可以让用户在安装时选择扇区大小，而不管 CONFIG.SYS 文件中的 BUFFERS 命令。显然，这可能发生冲突。DOS 是如何解决这个冲突的，Microsoft 公司没有给予说明。虽然 DOS 的文件缓冲区与文件柄发生关联，但程序员不可能通过文件柄来直接存取缓冲区的内容，甚至连缓冲区的位置和大小都不可能知道。

6.2.2 文件柄存取字节

对于每一个文件柄，DOS 还建立并维护一个文件柄存取字节，以便控制对文件的存取。应该注意，不要把文件柄存取字节和文件属性字节二者混淆了，也不要和下面将要讲到的文件柄属性字节混淆。在文件被关闭后，文件柄存取字节就不复存在。文件柄存取字

节各位的定义如下表所示。该表中包括了一些宏常数定义，这是为了便于系统级函数对文件柄存取字节进行操作。Turbo C 把某些存取控制位组合的名字，放在头文件 fcntl.h 中，

通过当前文件柄进行存取：：

O_RDONLY	0	只读
O_WRONLY	1	只写
O_RDWR	2	读／写
O_NOINHERIT	0x80	不遗传给子进程

通过其它文件柄进行存取：

O_DENYALL	0x10	拒绝所有的存取
O_DENYWRITE	0x20	拒绝写
O_DENYREAD	0x30	拒绝读
O_DENYNONE	0x00	不拒绝

从这个定义中可以看出，虽然文件柄存取字节也是由 DOS 提供的，但与文件属性字节有很大区别，这也说明了 DOS 在设计之初考虑不周。fcntl.h 头文件中还定义了其它一些常数宏，它们并不完全与文件柄存取字节直接对应。

程序通过文件柄可以对文件进行三种操作中的一种:只读，只写，读写。如果文件属性字节中规定文件是只读的，则将拒绝文件柄存取字节中的写请求。程序还可以请求把它的文件柄遗传给它的子程序(这里的子程序应理解为子进程，它是由当前进程调进内存执行的)。不仅如此，程序还可以规定在通过其它文件柄对该文件进行存取时所应强加的限制。这些限制共有 4 种，但每次只能规定其中的 1 种。为了使这些限制能起作用，必须预先执行 DOS 提供的一个内存驻留程序 share.exe。文件柄存取字节中的兼容方式是应用于网络计算机的，本书对此不再讨论。存取控制字节的缺省设置是 00000010。

6.2.3 文件柄属性字节

除了 DOS 提供的文件柄存取字节外，Turbo C 又为每个文件柄建立和维护了一个文件柄属性字节。Turbo C 最多打开 20 个文件，这 20 个文件的文件柄属性字节构成了一个数组，称为_openfd。文件柄属性字节的定义如下：

O_RDONLY	0x0100	只读文件
_O_EOF	0x0200	检测到<Ctrl_Z>字符
O_RDWR	0x0400	可读可写
O_APPEND	0x0800	在写以前移到文件尾(EOF)
O_CHANGED	0x1000	文件已经被修改
O_DEVICE	0x2000	文件柄是一个设备
O_TEXT	0x4000	文本文件

O__BINARY　　0x8000　　二进制文件

文件柄属性字节各位的含义不言而喻，但对文本方式和二进制方式还要再加以解释。如果文件柄属性字节中说明一个文件为文本方式，那么在输入这个文件时，每遇到<CR>(回车)<LF>(换行)对，舍弃<CR>，只保留<LF>。在输出这个文件时，每输出一个<LF>，在它之前先输出一个<CR>。而且在输入时，一旦检测到字符<ctrl－Z>（文件结束，EOF)，则终止输入过程，设置文件柄属性字节中的__O__EOF位。如果文件柄属性字节中说明一个文件为二进制方式，则输入输出过程中不进行上述转换。在 Turbo C 的头文件 fcntl.h 中，定义了一个全局变量__fmode，如果程序中没有另行说明文件是文本文件还是二进制文件，则大部分函数都是隐含地使用这个全局变量的值来决定是什么文件。

Turbo C 初始化__fmode 的值为 O__TEXT，即置为文本方式，但程序员可以修改这个值。

Turbo C 相应地初始化 5 个标准文件柄的文件柄属性字节，初始化其它文件柄的文件柄属性字节为全 0。

6.2.4 文件出错处理

在详细描述各个系统级函数前，先了解一下这些函数处理错误的办法也许很有帮助。文件处理中的失败可能经常出现，诸如：软盘驱动器未关门，文件找不到，企图往只读文件中写。反映这种操作失败用的办法就是返回一个无效值。但除此之外，所有函数也都通过全局变量 errno 来反映文件操作成功还是失败。如果失败，则给出错误号。这个全局变量的说明包含在头文件 errno.h 和 dos.h 中，0 值代表成功。

各种错误的描述也包含在头文件 errno.h 中。程序员可以用如下几种办法处理错误。

第 1，查阅 errno.h 文件，找到相应的说明。

第 2，执行 stdio 函数

```
void perror ( char *s )
```

这个函数先往标准错误输出设备上输出字符串 s，接着输出一个冒号以及与 errno 的当前值相应的错误信息，最后是换行符。字符串 s 用来帮助进行错误定位。下面的例子就说明了这种用法。

第 3，执行函数

```
char *strerror ( int N )
```

这个函数不是输出错误信息，而是返回指向错误描述的字符串的指针。只要在调用这个函数时，设置 n 的值等于全局变量 errno 的值即可以。strerror 的说明在头文件 string.h 中。

6.2.5 建立文件

文件处理的第 1 步就是建立文件。在为建立文件而提供的各个系统级函数中，下面两个是最常用的：

```
int creatnew(const char *path,int attrib);
```

```
int  creattemp(char  * path,int  attrib);
```

attrib 可以有如下 6 种值，这 6 种值分别对应于 6.1.2 节中所述的文件的 6 种属性，这些属性定义包含在头文件 dos.h 中。

```
FA_RDONLY          FA_SYSTEM          FA_DIREC
FA_HIDDEN          FA_LABEL           FA_ARCH
```

这些属性可以"或"起来，例如，可以通过如下的属性指定来建立一个只读的隐藏文件。

```
attrib=FA_RDONLY | FA_HIDDEN
```

如果指定名字的文件已经存在，creatnew 函数将失败。如果不存在，creatnew 函数将建立并打开这个文件。文件的文件柄存取字节被赋予隐含值 00000010，即只置位了读写控制位，但是奇怪的是，即使指定文件的属性为只读，仍然可以往这个文件上写，直到关闭这个文件。Turbo C 初始化这个文件的文件柄属性字节为 0。

函数 creattemp 与函数 creatnew 类似，差别在于:在调用 creattemp 时，参数 path 是以反斜线结尾的路径名的指针，Turbo C 将在这个目录下建立一个与已有文件不同名的新的临时文件，并把这个临时文件的名字返回在 path 指针指向的字符串中。程序退出之前，最好先关闭这个临时文件，但不要求一定这样做。

如果希望以与缺省值不同的文件柄存取字节和文件柄属性字节来建立和打开文件，则必须先通过 creatnew 或 creattemp 函数以缺省值建立文件，接着关闭它，然后再以所希望的方式通过 open 或_open 函数来打开它。

io.h 中还包含了另外两个建立文件的函数，即 creat 和_creat 。这两个函数与 creatnew 函数类似，但如果同名文件已经存在且是可写的，则这两个函数仍然打开它，删除已有文件的内容，为重写做好准备。此外，保留文件柄存取字节和文件柄属性字节的内容不变，而不管调用时的 fmode 参数值。一般来说，应避免使用 creat 和 _creat 函数，并尽可能把建立文件和打开文件的操作分开。

io.h 中还包含两个其作用与建立文件相类似的函数：

```
int  dup (int  oldhandle);
int  dup2(int  oldhandle,int  newhandle);
```

这两个函数都是复制文件柄，前者由 DOS 指定新的文件柄，后者由程序员指定新的文件柄。如果程序员指定的新文件柄已经赋予某个文件，则将在复制之前先关闭那个文件。复制功能是为已打开的同一文件(或设备)赋予一个新的文件柄，两个文件柄具有同样的文件柄存取字节和文件柄属性字节，具有同样的文件指针位置。dup 和 dup2 函数的作用主要有两个。第 1，用于更新文件而不需要关闭它；第 2，用于输入输出和重定向。10.3 节和 10.4 节将会看到这两方面应用的例子。

6.2.6 打开文件

io.h 中包含的最常用的打开现有文件的函数是:

```
int open(char * filename,int access);
```

参数 access 用来同时说明所希望的 DOS 文件柄存取字节和 Turbo C 文件柄属性字节，其值可以是如下之一:

- O_RDONLY、O_WRONLY、O_RDWR 中的一个
- O_NOINHERIT
- O_DENYALL、O_DENYWRITE、O_DENYREAD、O_DENYNONE 中的一个
- O_BINARY、O_TEXT 中的一个
- O_APPEND

如果成功，则函数返回一个文件柄。如果打开的目的是为了读，则文件指针将指向文件头。如果打开的目的是为了写，则原来的内容将被删除，文件指针将指向文件头。如果打开的目的是为了追加，则文件原来的内容仍将保持不动，在每次写操作前，文件指针都将指向文件尾。如果既没说明 O_BINARY，也没说明 O_TEXT，则使用全局变量 _fmode 中的值来决定文件是二进制文件还是文本文件。

io.h 中还有另一个打开文件的函数 _open，它比 open 函数稍低级一些。 _open 函数不处理 Turbo C 的文件柄属性字节,只用二进制文件方式，也不允许使用 O_APPEND 追加方式，实际中很少使用。

虽然 open 和 _open 函数都可以用来建立和打开文件，但建议不要这样做。

下面这个 sharing.dem 程序是用来解释文件共享存取位的作用的。第 1 次打开 sharing.exe 文件时，指定通过本文件柄可以写，拒绝通过别的文件柄写。假设第 1 次打开后返回的文件柄是 5 。第 2 次打开 sharing.exe 文件时，指定通过本文件柄只能读，通过别的文件柄可以做任何事情。第 2 次打开后，返回的文件柄将是 6。第 3 次打开 sharing.exe 文件时，指定通过本文件柄(如果成功，文件柄将是 7)只能读，拒绝通过别的文件柄进行写。当然，这种拒绝也包括拒绝通过文件柄 5 进行写，但第 1 次打开时已指定通过文件柄 5 可以写，所以第 3 次打开不能成功，返回无效的文件柄 -1。

```
#include    <io.h>
#include <fcntl.h>
#include <stdio.h>
void main( )
{
     int h1,h2,h3;
     h1 = open("Sharing.Exe",O_WRONLY | O_DENYWRITE);
     perror("Open for writing, denying others write access ");
     printf("Handle  %i\n",h1 );
     h2 = open("Sharing.Exe",O_RDONLY | O_DENYNONE);
     perror("Open for reading, with no further restrictions ");
     printf("Handle  %i\n",h2 );
     h3 = open("Sharing.Exe",O_RDONLY | O_DENYWRITE);
```

```
        perror("Open for reading, denying others write access  ");
        printf("Handle  %i\n",h3 );
    }
```

顺便说一句，对共享文件可以进行更精确的控制，可以只对文件的一部分上锁和解锁，而不一定要对整个文件上锁和解锁。io.h 中包含的 lock 和 unloch 函数就可以用来做这种上锁和解锁工作，这里不再举例了。

6.2.7 读取和设置文件的特性

Turbo C 中包含许多函数，用来通过文件名或文件柄读取或设置文件的各种特性，如：文件属性字节、文件长度、文件日期、文件时间，等等。下面这张表概括说明了这些函数及其功能。

特性	读取／设置	文件名／文件柄	函数名
存在?	读取	文件名	access
文件属性字节	读取	文件名	stat
	读取	文件名	_chmod
	设置	文件名	_chmod
只读	读取	文件名	access
	读取	文件柄	fstat
	设置	文件名	chmod
文件时间	读取	文件名	stat
	读取	文件柄	fstat
	读取	文件柄	getftime
	设置	文件柄	setftime
驱动器号	读取	文件柄	fstat
文件长度	读取	文件名	stat
	读取	文件柄	fstat
	读取	文件柄	filelength
是设备吗?	读取	文件名	stat
	读取	文件柄	fstat
文本／二进制?	设置	文件柄	setmode
文件尾?	读取	文件柄	eof
文件指针位置	读取	文件柄	tell
	设置	文件柄	lseek

表中只有 fstat 和 stat 函数的说明是在头文件 stat.h 中，其余都在头文件 io.h 中。对于表中所列的文件特性，为了便于引用，相当一部分都有相应的宏常数定义，这些定义散布在头文件 dos.h、fcntl.h、io.h 和 stat.h 中。这些函数的意义都是相当明确的，这里不再一一解释，读者可参考 Turbo C 的库函数手册。下面稍对 eof 函数做一些说明。

```
int eof(int handle);
```

它首先检查 Turbo C 的文件柄属性字节,看看这个字节中的文件结束标志位。如果这1位表示还没有检测到 1 个<Ctrl-Z>文件结束字符，eof 函数就使用与 tell 函数同样的办法去检查文件指针的位置。如果文件指针在文件尾，则返回 1；否则返回 0；如果出错，则返回-1。

6.2.8 读、写和关闭文件

在 io.h 中提供了两对读文件和写文件的函数：

```
int  _read(int handle,void  * block,int n);
int   read(int handle,void  * block,int n);
int _write(int handle,void  * block,int n);
int  write(int handle,void  * block,int n);
```

调用这些函数时，应给定文件柄、内存缓冲区地址、读写的字节数。函数返回实际读写的字节数，如果失败，则返回-1。因为-1 代表 65535(0XFFFF)，所以读写的最大字节数应是 65534，而不是 65535。函数_read 和_write 直接调用相应的 DOS 系统调用，不受 Turbo C 的文件柄属性字节的控制。即使是读写文本文件，_read 和_write 也不进行< CR >< LF >和< LF >之间的转换。函数 read 和 write 则较高级一些，受 Turbo C 的文件柄属性字节的控制。对于读操作，如果是文本文件，则把读入的< CR > < LF > 对转换成< LF >。在返回的实际读的字节数中，不包括字符< CR >，也不包括文件结束字符。当检测到文件结束字符时，函数 read 会置位文件柄属性字节中的文件结束标志。还没读完指定的字节数就检测到文件结束字符不认为是一个错误，但可能导致实际读入的字节数少于所请求的字节数。对于写操作，如果是文本文件，则在每写入 1 个< LF >字符前先写入 1 个 < CR > 字符，但这个< CR >字符不计在实际写的字节数内。如果文件打开时，已经置位 O_APPEND，则每次写操作都是从文件尾开始，而不是从当前文件指针开始。对于写操作，如果实际写的字节数少于所要求的字节数，则一般意味出了问题。

与 DOS 的 verify 命令相对应，Turbo C 提供的函数

```
int getverify(void);
int setverify(code);
```

分别用来读取和设置校核标志。1 表示要校核，0 表示不要校核。因为校核很费时间，所以缺省值为不校核。

系统级输入输出还包括 chsize、close、_close、lseek、tell 等函数，这些函数意义清楚，容易理解，这里不再详述了。

下面是一个表演程序，它利用系统级文件处理函数把一个文件变为大写字母文件或小写字母文件。执行这个程序时，第 1 个命令行参数应是文件名，第 2 个命令行参数应是 upper 或 lower 。

```
#include <ctype.h>
#include <io.h>
#include <fcntl.h>
#define BLOCKLENGTH 1024
void main(int nparameters,char *parameter[ ])
{
    int optionislower,n,endoffile,i;
    int f;
    long fp;
    char block[BLOCKLENGTH];
    optionislower=(toupper(parameter[2][0])=='L');
    f=open(parameter[1],O_RDWR);
    do {
        fp=tell(f);
        n=read(f,&block,BLOCKLENGTH);
        endoffile=eof(f);
        for (i=0;i<n;++i)
            block[i]=(optionislower ? tolower(block[i]):toupper(block[i]));
        lseek(f,fp,SEEK_SET);
        write(f,&block,n);
    }while(!endoffile);
    close(f);
}
```

6.3 标准级(流式)输入输出

与系统级函数比较，标准级函数具有缓冲功能和安全保护功能，故易于使用，有时效率也更高一些。然而，使用标准级函数的最主要好处还在于它与许多其它 C 语言函数的兼容性。这一节将详细讲述标准级函数，并给出一个演示程序。这个演示程序与上一节演示程序的功能相同，但这里是用标准级函数来写的，以便进行比较。

6.3.1 FILE 数据结构

标准级函数添加的许多功能都是由于它使用了 FILE 这个数据结构。这个结构定义在头文件 stdio.h 中，现摘录如下：

```
typedef struct {
    short           level;
    unsigned        flags;
    char            fd;
    unsigned char   hold;
    short           bsize;
    unsigned char   *buffer;
    unsigned char   *curp;
    unsigned        istemp;
    short           token;
}FILE;
```

每当打开一个文件进行标准输入输出时，就建立了一个 FILE 结构，并返回一个指向这个结构的指针。对于所有随后的操作，都以这个结构指针(下面称为流指针)为参照。程序员不应直接存取这个结构中的任何一个字段，因为它们仅供内部使用。这里列出这个结构只是为了方便读者理解 Turbo C 是怎样维护一个文件的。其它的 C 编译可能用了一个定义完全不同的结构，尽管结构的名字可能仍然是 FILE。

FILE 结构中的字段 fd 就是上一节所讲的文件柄。前面已经说过，一般来说不应混合使用系统级的文件柄和标准级的流指针。这是因为，标准输入输出操作是带缓冲的。假设一开始是进行标准级的输入，后来转入系统级的输入，待到再次使用标准级输入时，缓冲区就可能不再与文件中的数据相一致。而且，有些函数只能通过文件柄调用。为了根据流指针而获得文件柄，Turbo C 在头文件 stdio.h 中定义了一个宏 fileno(f):

```
#define fileno(f) ((f)->fd)
```

最好是通过这个宏定义去取得文件柄，而不要直接存取 FILE 结构。原因是：其它 C 编译可能使用了完全不同的 FILE 结构，Turbo C 也不保证它的 FILE 结构永远不变。

相反的要求也是能够实现的，即给已在系统级打开的文件再附上一个流指针。方法是使用 fdopen。总的来说，混合使用文件柄和流指针是可能的，但建议不要这样做。

结构 FILE 中的 flags 字段各位的含义如下：

符　号	位	意　义
_F_TERM	9	文件是一个终端
_F_OUT	8	输出在进行
_F_IN	7	输入在进行
_F_BIN	6	二进制方式
_F_EOF	5	EOF文件尾标志
_F_ERR	4	错误标志
_F_LBUF	3	行缓冲
_F_BUF	2	fclose 释放缓冲区
_F_WRIT	1	写
_F_READ	0	读

为了便于引用，Turbo C 在头文件 stdio.h 中为各位提供了一个宏常数定义。这些位中的大部分其含义是相当明了的。其中有些位与上一节讲述的 Turbo C 的文件柄属性字节的含义相同。位_F_BUF 表示：除了 DOS 为每个文件柄提供的缓冲外，Turbo C 是否还为每个流提供附加的缓冲。一般是提供附加缓冲的，指向这个附加缓冲的指针就存放在 FILE 结构内的 buffer 字段内。缓冲区大小在 bsize 字段内。缓冲区容量的缺省值为 512 字节，但可以通过函数 setvbuf 加以修改。如果用 setvbuf 函数分配一个缓冲区，则该函数将置位_F_BUF，以便当用 fclose 函数关闭这个文件时，释放这个缓冲区。FILE 结构内的 level 和 curp 字段用来存放有关缓冲的技术信息。

flags 字段内的位_F_LBUF 表示使用行缓冲还是扇区缓冲。无论是哪一种缓冲，输入操作都是从缓冲区中读，除非缓冲区被读空，当缓冲区为空后，输入操作将从盘文件

再读内容填充缓冲区。类似地，输出时也是首先输往缓冲区。如果是扇区缓冲，则当缓冲区满后就倒空到盘文件上。如果是行缓冲，则当遇到 1 个< CR > 字符时，就倒空到盘文件上。如果打开一个流时所关联的是一个终端设备，则这个流将是行缓冲的或不带缓冲的。如果关联的是一个磁盘文件，则是扇区缓冲的。

FILE 结构中的 hold 字段主要用来存储由程序员通过 ungetc 函数返回到输入的 1 个字符。istemp 字段表示该文件是否临时文件。临时文件一般是通过函数 tmpfile 建立的，关闭时必须从磁盘上删除。

上一节已经说过，程序开始运行时，Turbo C 自动打开了 5 个文件柄。为了也能用标准级函数来存取这 5 个标准设备，Turbo C 也自动分配和初始化相应的 5 个标准流及其流指针。

缺省情况下，只要不被重定向，stdin (标准输入)指的是键盘，用的是行缓冲。stdout (标准输出)和 stderr(标准错误)都是指显示器,不带缓冲。stdaux (标准辅助)、stdprn(标准打印)指的是什么与具体机器有关，在 PC 机上一般指串行口和并行口，打开时一般不带缓冲。通过 freopen 函数，可以重定义这些标准流指针，使它们参照一个盘文件或一个不同的设备。freopen 函数关闭当前与流相关联的文件，并把这个流赋予一个新文件。

下面分 7 个部分来阐述标准级文件输入输出操作：

· 建立／打开／关闭／删除文件
· 取文件状态和出错处理
· 控制文件缓冲区
· 移动文件指针
· 字节级的读写
· 字符串级的读写
· 记录级的读写

6.3.2 建立／打开／关闭／删除文件

为了建立／打开／关闭／删除文件，Turbo C 中提供了如下几个函数：

```
int  fclose(FILE *stream);
int  fcloseall(void);
FILE *fdopen(int handle,char *type);
FILE *fopen(char *filename,char *type);
FILE *freopen(char *filename,char *type,FILE *stream);
```

在 3 个打开文件的函数中，fopen 函数是用得最多的，它的调用参数是文件名和文件存取方式。文件存取方式有如下 6 种：

方式	若文件已经存在	若文件未存在
a	打开，只进行追加	建立和打开，只进行追加
a+	打开，读和追加	建立和打开，读和追加
r	打开，只读	报告一个错误

r+	打开，读和写	报告一个错误
w	使文件为空，打开，只写	建立和打开，只写
w+	使文件为空，打开，读和写	建立和打开，读和写

大体上可以这样说，如果希望存取一个已有的文件，且当这个文件不存在时能返回一个错误信息,则应该用"r" 类或"r+"类方式。如果希望建立一个文件，则应该用"w"类或"a"类方式。若文件已存在，"w"类将重写，"a"类将只追加而不重写。" r+ "、" w+ "和 " a+ "之类既可以读也可以写。交错进行读和写时应特别小心。每当从读转为写或从写转为读时，都应先调用 fseck 或 rewind 函数正确地定位文件指针。

为了说明被打开的文件是文本文件还是二进制文件，在 6 种文件存取方式字符串的后面还可以加上字母"t"或"b"，如"rt"、"rb"等。"t"表示文本文件，"b"表示二进制文件。这些函数也允许在"t"或"b"的前面或后面加 1 个"+"号，即"r+t"和"rt+"与"rt"是等效的。如果在打开时不说明"t"或"b"，则如上面所述，缺省方式是由全局变量__fmode 决定的。__fmode 在头文件 fcntl.h 中定义，其本身的缺省值是文本方式。为了避免混淆，最好在调用 fopen 函数时明确指定是"t"还是"b"。

下面是一个简单的例子，说明如何打开一个已有的文件去进行更新。

```
#include <stdio.h>
main( )
{
     FILE  * f;
     if((f=fopen("c:\\mydir\\myfile.dat","r+b"))==NULL) {
          printf("Error opening file, probably no such path\n");
     }
     else {
          printf("File opened for updating in binary mode.\n");
          fclose(f);
          printf("And is now closed.\n");
     }
}
```

注意，在说明路径名时应该用双反斜线"\\"，这是程序员经常容易犯的一个错误。在 DOS 下，反斜线(在 UNIX 操作系统下是用其它办法)实质上是一个 ESC 系列，所以必须用两个反斜线。这类小错误有时花几个小时也查不出来。

与系统级的 creattemp 函数相对应，为了建立临时文件，标准级提供了函数

FILE * tmpfile(void)

这个函数在执行过程中首先调用 tmpname 函数，取得一个在当前目录下不同于任何已有文件名的形为 TMP *.¥¥¥的文件名，然后再调用 fopen 函数，以"w+b"方式建立和打开这个临时文件。函数 tmpfile 返回一个流指针，并设置 FILE 结构中的 istemp 字段，以便当该文件被关闭程序终止时，从盘上删除这个文件。如果程序员需要知道临时文件名，则必须由程序员自己仿照 tmpfile 函数的执行过程去一步步实现，而不能直接调用 tmpfile 函数。

函数 freopen 与系统级函数 dup2 类似，它首先关闭所指定的流,然后以所指定的文件存取方式再重新打开这个流,并使之与所指定的文件名相关联。

函数 fdopen 前面已提到过，它是根据已在系统级打开的文件的文件柄，再辅之以一个流指针。

6.3.3 取文件状态和出错处理

取文件状态和出错处理的函数主要有下面几个：

```
void  clearerr(FILE  * stream);
int   feof(FILE  * stream);
int   ferror(FILE  * stream);
int   fstat(char  * handle,struct stat  * buff);
int   stat(char  * pathname,struct stat  * buff);
void  perror(char  * s);
char  strerror(int errnum);
char  * _strerror(char  * s);
```

前面已经说过，函数 stat 和 fstat 是用来取得 DOS 维护的 stat 结构的。如果在上一次输入输出操作中检测了文件结束标志，则 feof 函数返回非 0 值。如果在流的输入输出过程中发生了任何错误，则函数返回 1 个非 0 值。文件结束标志只能通过函数 clearerr、fseek、rewind 或 fsetpos 来清除。其它错误标志只能通过调用 rewind 或 clearerr 来清除。

当文件发生错误时，DOS 也维护着这些错误信息。Turbo C 函数 strerror、_strerror 和 perror 使得程序员可以打印大部分最近发生的 DOS 错误信息。Tuorbo C 的全局变量 errno 和_doserrno 记录最近发生的错误的号码。变量 errno 是一个与 UNIX 兼容的错误号，函数 perror 和 strerror 就使用了这个全局变量。全局变量_doserrno 被设置为实际的 DOS 错误号。虽然有时这两个变量的值相等，但一般情况下是不相等的，Turbo C 的参考手册上提供了这些错误的说明。全局变量 errno 和 _doserrno 反映的是上一次调用设置的值，Turbo C 的许多输入输出函数都设置这些错误标志(如果出错)，但每个函数都只设置多个标志中的一部分，这在 Turbo C 的函数库手册中都有说明。程序员应该尽可能使用这些标志，以确定错误发生的准确位置。

6.3.4 控制文件缓冲区

Turbo C 提供了 4 个函数，用来控制标准输入输出的缓冲区：

```
int   fflush(FILE  * stream);
int   flushall(void);
void  setbuf(FILE  * stream,char  * buf);
int   setvbuf(FILE  * stream,char  * buf,int type,unsigned size);
```

当打开一个文件时，系统自动为它分配一个缓冲区。这个缓冲区既用来读，也用来写。读文件操作实际上是先到缓冲区去读，当缓冲区中没有内容时，才到盘上去读 。一次从盘上读入的是一整块，而不仅仅是所需要的字节数。写文件操作实际上是先写到缓冲

区，当缓冲区满后才转储到盘上去。如果缓冲区的大小合适，文件存取的速度就会加快。Turbo C 提供的这些函数，使得程序员可以根据应用程序的具体情况，最佳地优化文件存取速度。

缓冲区也会引来一些问题。如果在缓冲区的内容被刷清到盘上去前程序突然流产，数据就可能丢失。另外，当从读转为写或从写转为读时，缓冲区的内容可能变得混乱，这就是为什么在改变输入输出的方向前要求程序员调用 fseek 或 rewind 函数重新设置文件的原因。这两个函数在移动指针到新的位置之前，先把缓冲区内容刷清到盘上。应该注意，当文件被关闭时，Turbo C 会自动刷清缓冲区。另外，如果通过 exit 函数或正常的终止来结束程序的执行，则所有缓冲区都会被刷清，所有文件都会被关闭。然而，文件操作结束后，及时显式地关闭文件是一个良好的编程习惯，这样还可以释放有限的内部文件表，以供其它文件使用。

在程序的任一处，通过调用 fflush 函数可以刷清指定流的缓冲区，调用 flushall 可以刷清所有流的缓冲区。这两个函数可用来帮助维护数据的完整性。例如，在写完一个重要数据后，立即刷清缓冲区，就可以避免由于程序突然流产而导致的数据丢失。

Turbo C 还提供了 setbuf 和 setvbuf 函数，允许程序员自己设置缓冲区的大小，甚至完全不使用缓冲功能。函数 setbuf 用于关闭缓冲区，或使用程序员自己分配的缓冲区而不使用系统分配的缓冲区。缓冲区的大小是固定的，由头文件 stdio.h 中的宏 BUFSIZE 规定，隐含设置为 512 字节。应该注意的是，setbuf 函数必须紧跟在文件打开之后，或紧跟在缓冲区刚被刷清之后被调用，否则，缓冲区的内容就会混乱。下面这个简短程序说明了 setbuf 函数的使用方法。

```
#include <stdio.h>
char mybuff[BUFSIZE];
main ( )
{
     FILE *f, *g, *h;
     f=fopen("file1.dat","r+");
     g=fopen("file2.dat","r+");
     h=fopen("file3.dat","r+");
     setbuf(g,NULL);    /* turn buffering off */
     setbuf(h,mybuff);  /* specify your own buffering */
     :
}
```

这个程序片断执行完以后，文件 f 仍然使用系统为它分配的缓冲区，文件 g 没有缓冲区，文件 h 使用用户定义的缓冲区。

函数 setvbuf 给程序员更多的控制权，它允许程序员说明缓冲区的大小以及其类型(type)。缓冲区类型有如下 3 种：

_IONBF	不用缓冲区，此时其它参数没有作用。
_IOLBF	文件是行缓冲的，只要写1个换行字符，缓冲区就被自动刷清。读操作仍然是全缓冲的。
_IOFBF	全缓冲(正常方式)。

使用函数 setvbuf 的一个程序片断如下：

```
#include <stdio.h>
char mybuff[8192];
main ( )
{
     FILE  * f, * g, * h;
     f=fopen("file1.dat","r+");
     g=fopen("file2.dat","r+");
     h=fopen("file3.dat","r+");
     setvbuf(f,NULL,_IONBF,0);
     setvbuf(g,mybuff,_IOFBF,sizeof(mybuff));
     setvbuf(h,NULL,_IOFBF,1024);
     :
}
```

这个程序片断执行完后，文件 f 将没有缓冲区，文件 g 将用 8K 字节缓冲区 mybuff，文件 h 将用 1K 字节缓冲区，其空间是间接调用 malloc 函数分配的。对文件 h 这种使用缓冲区的办法应特别小心，除调用 fclose 函数外，没有任何别的办法能释放这个缓冲区。这种对 malloc 函数的间接调用会设置 FILE 结构中的 flags 字段中的 _F_BUF 位，以便 fclose 函数正确释放这个缓冲区。

程序员在使用 setbuf 和 setvbuf 函数时最容易产生的一个错误就是把缓冲区分配在某个用户自定义函数的局部变量区内。如果在文件被关闭之前就退出这个用户函数，尽管这个文件的缓冲区也随之消失，缓冲区所占用的堆栈空间可能已移作它用，但随后的文件操作将继续把这块堆栈空间当缓冲区使用，其结果将可能造成系统崩溃。正因为如此，最好的办法是把缓冲区定义在任何函数之外，或把缓冲区说明为静态变量。

在某些情况下，系统不为打开的流自动开辟缓冲区。例如，标准输出流一般就是不带缓冲的，因为它们实际上关联的是显示器，缓冲没有什么意义。如果这些标准流被重定向到真正的盘文件，缓冲区就会自动建立起来。

总之，能够对缓冲区进行控制，正是 C 语言所提供的诸多灵活性中最具代表性的例子。程序员如果使用得好，可以使 C 语言程序执行速度更快。正是由于程序员可以对底层一些基本功能进行控制，进行优化，才使得 C 语言被看作是一种高效的语言。但这些底层功能的控制和优化，掌握起来较困难。

6.3.5 移动文件指针

随机文件存取往往需要移动文件指针，Turbo C 中提供了许多移动文件指针的函数，最常用的有如下 5 个：

```
int   fgetpos(FILE  * stream,fpos_t  * pos);
int   fseek(FILE  * stream,long offset,int origin);
int   fsetpos(FILE  * stream,const fpos_t  * pos);
long ftell(FILE  * stream);
int   rewind(FILE  * stream);
```

函数 rewind 移动指针到文件头，函数 fseek 移动指针到指定偏移量(offset)，偏移的参照点 origin 有如下 3 种：

SEEK_SET	文件头
SEEK_CUR	当前文件位置
SEEK_END	从文件尾开始往前移

应该说明的是，通过 fseek 函数把文件指针移到超过文件尾，并从那个新位置上进行写操作是合法的，这样做很容易扩充文件的长度。但有两点值得注意。第 1，原来文件尾和新位置之间的区域是未被初始化的。第 2，即使可以越过文件尾进行写，但试图读过文件尾时，还是会返回一个错误信息。

函数 ftell 用来读取文件指针当前位置相对于文件起点的偏移量。当一个文件以追加方式打开时，函数 ftell 返回的是由上一次输入输出操作所决定的当前文件指针位置，它不一定是下一次写操作的位置。这是因为，在追加方式下， 即使可以通过 fseek 函数把文件指针定位在任何一个位置，但写操作却总是在文件尾处进行。对这一点应格外小心。

在文本方式下，函数 ftell 还会带来另一个问题。因为<CR> <LF> 对和<LF>之间发生转换，故由 fteel 函数返回的值可能不代表相对于盘文件起点的真正偏移量。然而，fseek 也是同样处理这个转换关系的。因此，配合使用 ftell 和 fseek，先记住文件指针的位置，以后再返回到这个位置，就不会出错了。

函数 fgetpos 和 fsetpos 是新的 ANSI C 标准的一部分，它们分别类似于上面的函数 ftell 和 fseek。这里就不再多述了。

函数 fsetpos 和 ftell 会取消以前通过调用 ungetc 函数所退回的字符，对这一点也应加以小心。同时还应记住，随机存取只对真正的盘文件有意义，对设备(如控制台)不能进行随机存取。

6.3.6 字节级的读／写

字节级的读／写函数主要有如下几个：

```
int  fgetc(FILE * stream);
int  fgetchar(void);
int  fputc(int ch,FILE * stream);
int  ungetc(char * ch,FILE * stream);
int  getc(FILE * stream);
```

这几个函数比较简单，这里只简述函数 ungetc。如果从流中输入了 1 个字符，但决定过一会儿再处理这个字符，或者仅仅预先试探下一次输入的将是 1 个什么字符，则可以通过函数 ungetc 把已读入的字符再退回到流中去。被退回的这个字符可能放在流中，还可能保存在 FILE 结构中的 hold 字段中。如果文件指针被移动了，或者再次调用了 ungetc 函数，原来退回的字符就将丢失。

getc 实际上只是一个宏。

6.3.7 字符串级的读／写

字符串级的读／写函数主要有如下几个：

```
char  * fgets(char  * string,int  n,FILE  * stream);
char  * gets(char  * s);
int  fprintf(FILE  * stream,char  * fmt,...);
int  fputs(char  * string,FILE  * stream);
int  fscanf(FILE  * stream,char  * fmt,...);
int  vfprintf(FILE  * stream,char  * fmt,va_list  param);
int  vfscanf(FILE  * stream,char  * fmt,va_list  argp);
```

除最后两个以外，这些函数都是常用的，读者一般都很熟悉，这里就不多述了。值得一提的是函数 fgets 和 gets 之间的区别。前者用于读普通盘文件，所以在遇到"换行"符时，仍然保留它。后者用于读标准流，一般是键盘，所以在遇到"回车"符时，只保留回车符前的所有字符，而"回车"符则被"空"字符(NULL,'\0')代替。所以，如果一个程序原来是为控制台输入输出而写的，以后又想扩展到可以处理文件输入输出，则必须把 gets 改为 fgets。

函数 vfprintf 和 vfscanf 与 fprintf 的功能完全一样，都能处理可变数目的调用参数，只是 vfprintf 和 vfscanf 使用了比较新式的调用规则（参见第 1 章）。

6.3.8 记录级的读／写

记录级的读／写函数只有如下两个：

```
int fread(void  * ptr,int  size,int  nitems,FILE  * stream);
int fwrite(void  * ptr,int  size,int  nitems,FILE  * stream);
```

所谓记录，只不过是一个没有格式的数据块。记录有两种：一种是定长的，另一种是不定长的，这一节讨论定长记录，不定长记录在第 9 章讨论。值得注意的是，从函数的角度来讲，Turbo C 只提供了定长记录的支持，不定长记录也是通过这两个函数读／写的，但需要的是更多的技巧，更精心构思的算法。

在函数参数中，ptr 是指向内存缓冲区的指针，size 是一个记录的字节数，nitems 是读／写的记录数。下面就是记录级读／写的一个例子，它改写文件 myfile.dat 中的第 6 个记录。

```
#include <stdio.h>
typedef struct my_parts_struct {
      int    part_code;
      int    quantity;
      float  price;
}part;
main ( )
{
      FILE  * f;
```

```
    int  nb;
    part mypart={15,300,1.25};
    if((f=fopen("myfile.dat","w+b"))!=NULL) {
        fseek(f,(long)(sizeof(part) * 6),SEEK_SET);
        nb=fwrite(&mypart,sizeof(part),1,f);
        printf("The number of records written is %d\n",nb);
        fclose(f);
    }
    else {
        printf("Error opening file ...\n");
    }
}
```

这个例子比较简单，但有两点值得注意。第1，调用fseek函数时所指定的偏移量应该是一个长整数。第2，一个记录的字节数应该通过sizeof操作符求得，而不应该由程序员自己计算。这是因为，在C中记录一般都是通过结构实现的，而在许多C编译中具体实现一个结构时，为了边界对齐，往往添加了一些字节。但各个C编译添加的方法和字节数往往不一致，程序员自己计算出来的结构的字节数可能与编译实际为每个结构分配的字节数不一致。在Turbo C编译中，缺省的原则是不添加任何字节，以使结构最紧凑，少占内存。但是，如果程序员确实想以整数为边界对齐，则在编译时可以加一个"_a"选择项。加了这个选择项会进行如下三项对齐：

(1) 结构将总是从偶地址开始。

(2) 结构内的非字符字段也总是从偶地址开始。

(3) 整个结构总是占偶数个字节。

这种对齐使运行速度加快，但除了增加存储量外，还带来另一个问题,即使是同一个Turbo C源程序，如果在编译时一个加了"_a"选择项，另一个没有加"_a"选择项，则两个可执行文件所产生的数据文件也是不能互相共享的。

6.4 基本文件处理工具包

前面已经说过，Turbo C的文件处理函数功能已经较完整了，Turbo C Tools在这方面没有做什么大的改进，本工具包也只是稍微做了一些改进，没有也不可能做什么本质上的改变。本工具包的10个函数基本上做两件事：随时报告文件存取过程中发生的错误和允许用户方便地以记录为单位进行读写。因为结构FILE没有提供足够的信息来做这两件事，所以本工具包建立了一张内部文件表，表中的每一项除包含指向结构FILE的指针外，还包含一个指向文件名字符串的指针和一个记录大小字段。通过本工具包打开的每一个文件都在这个表内占有一项，每一项在表内的索引号也就成了控制这个文件的文件柄。为了与DOS的文件柄相区别，我们不妨称它为包内文件柄。

6.4.1 10个工具函数

本工具包一共提供了如下10个工具函数：

```
extern  void init__files(void);
extern  int  get__file__data(int h,FILE * * f,char * * fname,int * recsize);
extern  int  openfile(char * fname, char * access__type, int recsize);
extern  int  closefile(int h);
extern  long movepos(int h, long recno,int smode);
extern  int  iobytes(int h, long recno, unsigned char * d,
                      int nr,int iodir);
extern  int  rdstr(char * str, int n, int h);
extern  int  wrtstr(char * str, int h);
extern  int  rdfstr(int h, char * format, ...);
extern  int  wrtfstr(int h, char * format, ...);
```

(1) 函数 init__files 初始化包内文件表，使表内每一项为空项，即使表内每一项的 FILE 结构指针和文件名字符串指针都为 NULL。此外，还调用 add__file 函数把 5 个标准输入／输出文件加到文件表中去。内部函数 add__file 在包内文件表中寻找一个空项。如果找到，则增加一项，把 FILE 指针、文件名字符串指针和记录大小填写进去，返回包内文件柄。如果找不到，则返回−1。

内部函数 rmv__file 从包内文件表中删去一项，使该项成为一个空项，释放文件名字符串占用的空间。

内部函数 valid__index 校核所指定项是否为空项。如果不空，则返回 1，否则返回 0。

(2) 函数 get__file__data 读取包内文件表中指定项的信息.如果这一项不空，则通过 3 个参数返回该项的 3 个值，返回值为 0。如果这一项为空，则返回的 FILE 指针为空，文件名为 unknownfile(表示不存在这个文件)，返回值为−1。

(3) 函数 openfile 以所指定的存取方式和记录大小打开／建立一个文件，把有关的信息存入包内文件表中，返回包内文件柄。如果出错，则返回值为−1。

(4) 函数 closefile 关闭所指定的文件，使包内文件表中的相应项为空，返回值为 0。如果出错，返回值为−1。

(5) 函数 movepos 移动文件指针到指定的记录号处，寻找方式可以是 SEEK__SET、SEEK__CUR 和 SEEK__END，其定义与 Turbo C 函数 fseek 的定义相同。如果成功，则返回值为记录号，否则返回值为−1。应该注意的是，在文件字节位置和文件记录号换算时忽略了文本文件中换行符的转换问题。

(6) 函数 iobytes 在指定文件 h 中从指定记录号 recno 开始连续读(iodir＝0)或写(iodir＝1)指定的记录数 nr，缓冲区指针是 d。如果记录号 recno 为−1，则从当前记录号开始。如果成功，则返回值为实际读或写的字节数，否则返回值为−1。

(7) 函数 rdstr 和 wrtstr 分别用来从指定文件的当前文件指针处读和写 1 个字符串。它们分别调用 Turbo C 的 fgets 和 fputs 函数去实现，但为了正确地维护缓冲区，它们首先重新定位了文件指针。如果成功，函数 rdstr 的返回值为 0，函数 wrtstr 的返回值为写入的最后 1 个字符。如果不成功，返回值为−1。

(8) 函数 rdfstr 和 wrtfstr 分别用来从指定文件的当前文件指针处读和写 1 个格式化字符串。它们分别调用 Turbo C 的 vfscanf 和 vfprintf 函数去实现，但为了正确地维护缓冲

区，它们首先定位了文件指针。如果成功，rdfstr 的返回值为扫过的字段数，wrtfstr 的返回值为写出去的字节数。如果不成功，返回值为-1。

基本文件处理工具包头文件 fileio.h:

```
#define inbytes(port,rec,d,nr)     iobytes(port,rec,d,nr,0)
#define outbytes(port,rec,d,nr)    iobytes(port,rec,d,nr,1)
#define MAXFILES 20
typedef struct fh_struct {
        FILE  *fp;
        char  *name;
        int   recsize;
}file_entry;
extern        file_entry ft[ ];
extern  void init_files(void);
extern  int  get_file_data(int h,FILE **f,char **fname,int *recsize);
extern  int  openfile(char *fname, char *access_type, int recsize);
extern  int  closefile(int h);
extern  long movepos(int h, long recno,int smode);
extern  int  iobytes(int h, long recno, unsigned char *d,
                 int nr,int iodir);
extern  int  rdstr(char *str, int n, int h);
extern  int  wrtstr(char *str, int h);
extern  int  rdfstr(int h, char *format, ...);
extern  int  wrtfstr(int h, char *format, ...);
```

基本文件处理工具包源程序 fileio.c:

```
#include    <string.h>
#include    <alloc.h>
#include    <stdio.h>
#include    "popup.h"
#include    "fileio.h"

file_entry  ft[MAXFILES];
static char unknownfile[ ] = "writing";
static char iowstr[ ] = "writing";
static char iorstr[ ] = "reading";
static int  add_file(FILE *f, char *n, int recsize);
static void rmv_file(int h);
static int  valid_index(int h);

void init_files(void)
{
     int h;
     for (h=0; h<MAXFILES; h++) {
          ft[h].fp  = NULL;
          ft[h].name = NULL;
     }
     add_file(stdin, "stdin", 1);
```

```
	add_file(stdout,"stdout",1);
	add_file(stderr,"stderr",1);
	add_file(stdaux,"stdaux",1);
	add_file(stdprn,"stdprn",1);
}

static int add_file(FILE *f, char *n, int recsize)
{
	int h;
	for (h=0; (h<MAXFILES) && (ft[h].fp !=NULL); h++);
	if (h ==MAXFILES)        {
		sayerr(SERR,"FILE table full\r\n");
		return -1;
	}
	ft[h].fp =f;
	ft[h].name = strdup(n);
	ft[h].recsize = recsize;
	return h;
}

static void rmv_file(int h)
{
	ft[h].fp = NULL;
	free(ft[h].name);
	ft[h].recsize = 0;
}

static int valid_index(int h)
{
	if ((h >= 0) && (h < MAXFILES) && (ft[h].fp != NULL)) {
		return 1;
	}
	else {
		sayerr(SERR,"Illegal file index %d\r\n",h);
		return 0;
	}
}

int get_file_data(int h, FILE **f, char **fname, int *recsize)
{
	if (!valid_index(h)) return -1;
	if ((*f = ft[h].fp) == NULL) {
		*fname = unknownfile;
		*recsize = 0;
		return -1;
	}
	else {
		*fname = ft[h].name;
		*recsize = ft[h].recsize;
		return 0;
	}
```

```
}

int openfile(char * fname, char * access_type, int recsize)
{
     FILE * f;
     if ((f = fopen (fname, access_type)) == NULL) {
          sayerr(FERR,"%s\r\n",fname);
          return -1;
     }
     else {
          return add_file(f, fname, recsize);
     }
}

int closefile(int h)
{
     if (!valid_index(h)) return -1;
     if (fclose(ft[h].fp) == -1) {
          sayerr(FERR,"%d - %s\r\n",ft[h].fp,ft[h].name);
          return -1;
     }
     rmv_file(h);
     return 0;
}

long movepos(int h, long recno,int smode)
{
     long newpos;
     if (!valid_index(h)) return -1;
     newpos = recno * ft[h].recsize;
     if ((newpos = fseek(ft[h].fp,newpos,smode)) ==-1)          {
          sayerr(FERR,"file: %s\r\nrecord %ld\r\n",
                    ft[h].name,recno);
          return -1;
     }
     return newpos / ft[h].recsize;
}

int iobytes(int h, long recno, unsigned char * d, int nr,int iodir)
{
     unsigned int recsmoved;
     char * iodirstr;
     long savepos;
     if (!valid_index(h)) return -1;
     if (recno == -1) {
          if (movepos(h, 0L,   SEEK_CUR) == -1L) return -1;
     }
     else {
          if (movepos(h, recno,SEEK_SET) == -1L) return -1;
     }
     if (iodir == 1) {
```

```
        recsmoved = fwrite(d,ft[h].recsize,nr,ft[h].fp);
    }
    else {
        recsmoved = fread (d,ft[h].recsize,nr,ft[h].fp);
    }
    if (recsmoved == -1) {
        if (iodir==1) iodirstr=iowstr;
        else iodirstr = iorstr;
        sayerr(FERR,"%s %s\r\npos'n %ld\r\n", iodirstr,
               ft[h].name, recno);
        return -1;
    }
    else {
        /* savepos += recsmoved * ft[h].recsize;
           if(movepos(h,savepos,SEEK_SET)==-1L) return -1;
        */
    }
    return recsmoved;
}

int rdstr(char *str, int n, int h)
{
    if (!valid_index(h)) return -1;
    if (movepos(h, 0, SEEK_CUR) == -1L) return -1;
    if (fgets(str, n, ft[h].fp) ==NULL) return -1;
    return 0;
}

int wrtstr(char *str, int h)
{
    int c;
    if (!valid_index(h)) return -1;
    if (movepos(h, 0, SEEK_CUR) == -1L) return -1;
    if ((c = fputs(str, ft[h].fp)) ==EOF) return -1;
    return c;
}

int rdfstr(int h, char *format, ...)
{
    va_list arg_ptr;
    int e;
    if (!valid_index(h)) return -1;
    if (movepos(h, 0, SEEK_CUR) == -1L) return -1;
    va_start(arg_ptr,format);
    e = vfscanf(ft[h].fp,format, arg_ptr);
    va_end(arg_ptr);
    if (ferror(ft[h].fp) || feof(ft[h].fp)) {
        sayerr(FERR,"scanning %s\r\n", ft[h].name);
        clearerr(ft[h].fp);
        return -1;
    }
```

```
    return e;
}

int wrtfstr(int h, char *format, ...)
{
    va_list arg_ptr;
    int e;
    if (!valid_index(h)) return -1;
    if (movepos(h, 0, SEEK_CUR) == -1L) return -1;
    va_start(arg_ptr,format);
    e = vfprintf(ft[h].fp, format, arg_ptr);
    va_end(arg_ptr);
    if (ferror(ft[h].fp) || feof(ft[h].fp)) {
        sayerr(FERR,"printing to %s\r\n", ft[h].name);
        clearerr(ft[h].fp);
        return -1;
    }
    return e;
}
```

6.4.2 工具包应用

这个工具包没有引进复杂的新概念，每个函数的物理含义都十分清楚，故不再单独举例介绍了。在第8章和第9章，都会有应用这个工具包的具体例子。

6.5 驱动器和目录操作

本节主要讨论如何对驱动器和目录进行操作，而不是对各个文件进行操作。在Turbo C中，有些函数几乎直接与DOS的命令相对应，有些函数使程序员在程序中能够像DOS一样处理文件名中的"?"号和"*"号。这一节还提供了两个演示程序。第1个演示程序说明如何进行文件和目录的换名和重定向，第2个演示程序是一个搜索程序，统计其名字与指定模式相匹配的文件的总数。

大部分目录操作函数在遇到错误时，都会返回一个非0值或无效值，同时设置全局变量errno为相应的值。程序员可通过perror和strerror函数得知错误的进一步细节，如前两节所述的一样。但也有几个函数是例外的，后面将给予说明。

6.5.1 驱动器和驱动器信息

Turbo C中提供的驱动器操作函数有如下几个：

```
int    biosequip(void);
char   *getcwd(char *buf,int buflen);
int    getcurdir(int drive,char *directory);
void   getdfree(unsigned char drive,struct dfree *dtable);
int    getdisk(void);
int    setdisk(int drive);
```

函数 getdisk 和 setdisk 与 DOS 的 CD 命令或驱动器号命令相并行，是用来读取或设置当前驱动器台号的，它们的说明在头文件 dir.h 中。在这两个函数中，0 对应于驱动器 A，1 对应于驱动器 B,……。这一点与其它目录函数不同，后者的 0 对应于当前驱动器，1 对应于驱动器 A，2 对应于 B，……。这两个函数都几乎没有什么错误信息,setdisk 在遇到无效的 drive 参数时，只是简单地加以忽略。函数 setdisk 返回的是驱动器的总数，但这个数字是不可靠的，并随 DOS 版本的不同而不同。在后期的版本中，返回的值最少是 5。函数 biosequip 也能返回软盘驱动器的总数，但往往也容易被误解。因为单个软盘驱动器也可以起 A 盘和 B 盘的作用，某些虚拟盘驱动程序也可以当作软盘驱动器使用，但某些软盘设备驱动程序却又把自己标志为硬盘，等等。

函数 getfree 可以用来取得有关驱动器的更详细的信息，这些信息存放在一个类型为 dfree 的结构中。函数 getdfree 和结构 dfree 的说明都在头文件 dos.h 中。

```
struct dfree {
      unsigned df__avail;
      unsigned df__total;
      unsigned df__bsec;
      unsigned df__sclus;
}
void getdfree(unsigned char drive,struct dfree  *dtable);
```

如果遇到错误，函数 getdfree 会置结构 dfree 中的 df__sclus 字段为-1。

还有一些其它函数也提供了有关驱动器的信息，但这些信息太专门化，除非是专门面向管理驱动器的软件，否则很少需要这些信息。其中 ioctl 函数的部分功能可能比较常用。

```
int ioctl(int handle,int func[,void  *argdx,int argcx]);
```

调用 ioctl 时，若位置参数 handle 为驱动器号(0 代表缺省盘，1 代表 A 盘，等等)，func 置为 8，argdx 和 argcx 都不需要，则返回 0 值意味着该驱动器是可换介质的，返回非 0 值意味着是不可换介质的。其余的用法可参考 Turbo C 手册。下面是一个说明这些函数用法的例子：

```
#include <stdio.h>
#include <dir.h>
#include <io.h>
#include <dos.h>
void main( )
{
     struct dfree d;
     unsigned long totalbytes,availbytes;
     printf("The medium in drive %c: is %s removable.\n\n",
             getdisk( )+'A',ioctl(0,8) ? " not " : " ");
     getdfree(0,&d);
     totalbytes=(unsigned long)d.df__total * d.df__bsec * d.df__sclus;
     availbytes=(unsigned long)d.df__avail * d.df__bsec * d.df__sclus;
```

```
    printf("Bytes / sector:     %9u     Sectors / cluster:     %2u\n",
            d.df_bsec,d.df_sclus);
    printf("Total clusters:  %9u     Available:          %8u\n",
            d.df_total,d.df_avail);
    printf("Total bytes:      %9lu     Available:          %8lu",
            totalbytes,availbytes);
}
```

6.5.2 目录操作

Turbo C 中提供的目录操作函数有如下几个:

```
int  getcurdir(int drive,char * directory);
int  chdir(const char * path);
int  mkdir(const char * path);
int  rmdir(const char * path);
char * getcwd(char * buf,int buflen);
int  chmod(const char * path,int amode);
int  _chmod(const char * path,int func[,int attrib]);
int  rename(const char * oldname,const char * newname);
```

在这 8 个函数中,前 5 个的意义与 DOS 命令差不多。这里只指出需要注意的几点。第 1,在调用 getcurdir 时,参数 directory 所指向的缓冲区(用来返回当前目录)至少应有 MAXDIR 个字节。MAXDIR 是在头文件 dir.h 中定义的一个常数宏。第 2,在调用 chdir 时,若参数 * path 中不含驱动器名,则指的是当前驱动器;若含驱动器名,则该函数只改变所指定驱动器上的当前目录,不改变当前驱动器。

DOS 存储子目录(只是子目录,不包括根目录) 的办法与存储一个普通文件一样,每个目录也有 1 个文件属性字节,通过函数_chmod 可以读取(func=0)和设置(func=1)这个属性字节。但 chmod 只能改变文件属性字节 6 位中的 3 位,即 FA_RDONLY(只读)、FA_HIDDEN(隐藏)、FA_SYSTEM(系统)。是否为"只读"对目录操作看来没有什么影响,"隐藏"和"系统"可以防止这个目录在搜索和列表时被显示出来。

子目录也有日期/时间,但 Turbo C 的库函数 getftime 和 setftime 却只能读取/设置文件的日期/时间,不能读取/设置子目录的日期/时间。作为下面将要讲到的广义搜索技术(搜索含"*"和"?"的文件名)的副产品,可以读取目录的日期/时间,却仍然不能设置日期/时间。但如果能够设置,则有时是方便的。例如,在准备发行商品软件时,可以使目录的日期/时间与软件版本号相一致。如果想这么做,看来只能先设置所希望的日期/时间,接着建立一个新子目录,再接着拷贝所要的文件到这个新子目录下,然后删除被拷贝的源文件,最后重新设置日期/时间为当前的日期/时间。

通过函数 rename 可以对子目录换名,但却不能将整个子目录从一个目录下移到另一个目录下。如果要想这么做,只能按如下步骤去实现:建立新目录,拷贝所有文件,删除源文件,删除原目录。而且,当原目录下还有子目录时,这个过程还必须递归进行,象下面将要讲到的广义搜索一样。

6.5.3 文件名操作

Turbo C 中对文件名进行操作的函数有如下几个：

```
int fnsplit(const char *path,char *drive,char *dir,
            char *name,char *ext);
void fnmerge(char *path,const char *drive,const char *dir,
             const char *name,const char *ext);
int rename(const char *oldname,const char *newname);
```

前两个函数是互补的，前者用来把全文件名分解为驱动器名、路径名、文件名、扩展名。后者则相反，由 4 个部分名合并成全文件名。用来存储这 5 个名字的空间要求 MAXPATH、MAXDRIVE、MAXDIR、MAXFILE、MAXEXT 定义在 dir.h 中，分别为 80、3、60、9 和 5 个字节。

函数 rename 可以对同一个驱动器下的文件和子目录进行换名。如果是文件，则新名和旧名可以在不同目录下。如果是子目录，则新名和旧名必须在同一目录下。无论是文件或子目录，都必须在同一驱动器下，都不允许使用广义字符“*”和“?”。

6.5.4 目录搜索

Turbo C 中提供的目录搜索函数有如下两个：

```
int findfirst(const char *pathname,struct ffblk *ffblk,int attrib);
int findnext(struct ffblk *ffblk);
```

这两个函数与 DOS 的两个系统调用类似，它们配合使用，以解决含“*”和“?”广义字符的文件名的搜索问题。结构 ffblk 用来存储有关文件的一些信息，基本上就是 DOS 的文件目录项中的信息。这两个函数及结构 ffblk 的说明都在头文件 dir.h 中。

```
struct ffblk {
    char      ff_reserved[21];
    char      ff_attrib;
    unsigned  ff_ftime;
    unsigned  ff_fdate;
    long      ff_fsize;
    char      ff_name[13];
}
```

函数 findfirst 根据当前驱动器和目录以及由参数 pathname 传来的目录名和文件名，决定到哪一个目录上去搜索。pathname 的文件名部分可以含符号“*”和“?”。如果成功了，则返回 1 个 0 值，同时把找到的第 1 个文件的信息存放在结构 ffblk 中。参数 attrib 说明搜索什么类型的文件。类型有 6 种，即文件属性中的 6 位：FA_RDONLY、FA_HIDDEN、FA_SYSTEM、FA_LABEL、FA_DIREC、FA_ARCH。由于 DOS 的一些缺陷，这些属性的解释略为复杂，归纳起来有如下 3 点：

(1) 如果 attrib 等于 0，则只搜索普通文件，包括只读文件和档案文件。

(2) 如果 attrib 设置为隐藏、系统、子目录中的一种或它们的组合，则除了搜索普通文件外，还搜索符合指定属性的文件。

(3) 如果 attrib 设置为卷标，则只搜索卷标。

如果成功，函数 findfirst 把调用时传来的参数 pathname 和 attrib 保存在结构 ffblk 的 ff_reserved 字段内，把文件的属性、长度、日期、时间、名字等信息存放在结构 ffblk 的其它字段内。函数 findnext 接过由 findfirst 借结构 ffblk 传来的信息，继续往下找，并在成功后，按 findfirst 函数同样的方式存储信息，以供再一次调用 findnext 函数去继续搜索。

下面是目录搜索的演示程序，它的输入参数有两个。第 1 个是欲搜索的第 1 个文件的名字，这个名字可以是一个普通的文件名，也可以是子目录名；可以带路径，也可以不带路径。第 2 个参数是文件名和扩展名的模式。这个程序按目录树的结构打印出找到的所有匹配的子目录的名子，统计并打印出匹配文件的总数和总字节数。

整个演示程序的重点是递归函数 process，它比较复杂，下面做一些解释。主函数第一次调用 process 时，传给它的搜索参数就是程序执行时命令行尾的两个参数。filename 是被搜索的起点，可以是子目录名，也可以是一个普通文件名。process 又首先调用 findfirst 去寻找第 1 个匹配。如果 findfirst 返回 1 个非 0 值，一般情况下这意味着找不到相匹配的名字，则退出 process。如果入口时所指定的参数 filename 是根目录(filename 的最后 1 个字符是冒号“:”或反斜线“\”)，则 findfirst 返回的也是非 0 值，即未找到一个匹配。这是因为，每个子目录中的第 1 个子子目录是“.”，它的名字字段包含该子目录自身的名字，而根目录没有这样一个参考其自身的子目录名，所以 findfirst 找不到根目录。但根目录总是存在的，总是认为可以找到的，如果 findfirst 返回 1 个 0 值，则意味着找到一个匹配。如果匹配的是一个文件，则相应地登记文件数和字节数。如果匹配的是一个目录，包括根目录，则保留当前目录名，以备在退出 process 时回到这个目录下。把所找到的目录设为当前目录，然后依次进入两轮循环。第 1 轮循环在当前目录下寻找所有与调用时指定的名字模式相匹配的普通文件，统计文件数和字节数。第 2 轮循环在当前目录下寻找所有下层子目录(不包括当前目录本身和其父目录)。对于每个找到的下层子目录，再次递归调用 process 函数，统计文件数和字节数。在遍历了所有下层子目录后，再重新设置循环开始前所保存的目录为当前目录，退出 process。

```
#include <general.h>
#include <ctype.h>
#include <dir.h>
#include <dos.h>
#include <stdio.h>
int indent = 2;
void process(char *filename,char *quest,int *files,long *bytes)
{
     int            code;
     struct ffblk   info;
     char           ch;
     char           directory[MAXPATH];
```

```
    int             morefiles;
    long            morebytes;
    printf("% * s%s\n",indent,"",filename);
    * files = * bytes = 0;
    code = findfirst(filename,&info,FA_DIREC);
    if (code) {
        ch = filename[strlen(filename)-1];
        if (ch == ':' || ch == '\')
            info.ff_attrib = FA_DIREC;
        else {
            puts("No file found.\n");
            return;
        }
    }
    if (info.ff_attrib != FA_DIREC) {
        * files = 1;
        * bytes = info.ff_fsize;
        return;
    }
    getcwd(directory,MAXPATH);
    chdir(filename);
    code = findfirst(quest,&info,0);
    while (code == 0) {
        ++ * files;
        * bytes += info.ff_fsize;
        code    = findnext(&info);
    }
    code = findfirst(" * . * ",&info,FA_DIREC);
    while (code == 0) {
        if (info.ff_attrib==FA_DIREC && info.ff_name[0]!='.') {
            indent += 2;
            process(info.ff_name,quest,&morefiles,&morebytes);
            indent -= 2;
            * files += morefiles;
            * bytes += morebytes;
        }
        code = findnext(&info);
    }
    chdir(directory);
}

void main ( )
{
    int  drive,files;
    char filename[MAXPATH];
    char quest[MAXPATH];
    long bytes;
    drive = getdisk( );
    printf("Search the directory tree starting at    ");
    gets(filename);
    if (filename[1] == ':')
```

```
        setdisk(toupper(filename[0]) - 'A');
    printf("search for files matching this pattern:  ");
    gets(quest);
    printf("Processing directories:\n");
    process(filename,quest,&files,&bytes);
    printf("Number of files found = %d \n",files);
    printf("Total number of bytes = %ld\n",bytes);
    setdisk(drive);
}
```

对第1轮和第2轮循环的某些条件进行改变就可以使process函数适用于别的应用场合，如统计某年某月某日以后建立的所有文件等等。

应当注意，函数findfirst 和 findnext 会改变一个DOS的指针，使它由指向磁盘传输区(DTA，Disk Transfer Address)改为指向一个结构变量ffblk，把一些目录信息放在那个结构里，在整个目录搜索过程中，这个指针必须保持不变。然而，很多其它DOS服务也用到这个指针。因此，在调用findfirst和findnext前，应把这个指针压入堆栈，从函数返回后，再从堆栈中弹出这个指针。否则，如果你的程序中断了另一个程序，它就可能破坏DOS为那个程序提供的服务。类似地，别的程序也应采取同样的策略。Turbo C中的getdta 和 setdta 函数为读取和设置DTA 指针提供了方便。

在Turbo C 中还有一个很有用的函数，它虽然与广义文件名搜索没有什么关系，但应该在这儿讨论一下。这就是函数

```
char *searchpath(char *filename)
```

它首先在当前目录下，接着在DOS环境 PATH 所指示的各个目录下，寻找文件filename。如果找到了，则返回一个指向全名字符串的指针，否则返回一个空指针。在调用这个函数时，不需要分配存储全名的空间。全名被存放在一个静态变量中，每次调用searchpath 函数都会改写这个静态变量。函数 searchpath 不仅可以用于可执行文件，也可用于任何其它文件。

第 7 章　字符串处理

大部分编程任务都牵涉到字符串处理。即使被处理的数据主要是数值数据或图形数据，在用户接口中以及各部分程序之间的接口中，仍然需要用到字符串处理。

7.1 字　符

7.1.1 字符数据和常数

在 Turbo C 中，字符分为无符号字符和有符号字符两种，缺省情况下认为字符数据类型 char 是有符号字符，除非在编译时明确推翻这个缺省规定。把字符认为是带符号的不符合道理，因为 ASCII / IBM 码是从 0～255，而不是从-128～127。当 ASCII / IBM 码值超过 127 后，这个问题尤其应该注意。

signed char	0......127	-128.....-1
unsigned char	0......127	128.....255
hex	00......7F	80........FF
ASCII 码	0......127	128.....255

ASCII 标准只规定了码 0～127，称为 ASCII 码标准。码 128～255 则在各种不同的输入 / 输出设备上有很大差别，IBM 公司在 PC 机上对码 128～255 做了规定，所以码 0～255 统称为 ASCII / IBM 码。

一般认为，字符 char 占 1 个字节，但实际上字符也是按 16 位整数存储的。对于有符号字符，高字节是低字节的符号扩展；对于无符号字符，高字节是 0。但是在 Turbo C 中，允许两个字节的字符常数，如'AB'，'\n\t'，'\007\007' 等等。这些两字符常数也是按 16 位整数存储的，第 1 个字符在低字节，第 2 个字符在高字节。这种表达方式仅在 Turbo C 中，不能移植到其它 C 编译中。

给 1 个字符变量赋值可以有 8 种办法:

- 十进制常数，以非 0 数字开头
- 十进制常数，以负号开头
- 八进制常数，以数字 0 开头
- 十六进制常数，以 0X 开头，X 可以大写或小写
- 加单引号的字符
- 加单引号的以十六进制常数表示的ESC系列，即一个反斜线后加一个X(大小写均可)再加十六进制字符
- 加单引号的以八进制常数表示的ESC系列，即一个反斜线后加几个八进制数字。
- 加单引号的特殊 ESC 系列，如下表所示。

ESC 系列	值	符号表示	意义
\a	0x07	BEL	响铃
\b	0x08	BS	回退
\f	0x0c	FF	换页
\n	0x0a	LF	换行
\r	0x0d	CR	回车
\t	0x09	HT	水平制表
\v	0x0b	VT	垂直制表
\\	0x5c	\	反斜线
\'	0x27	'	单引号
\"	0x22	"	双引号
\?	0x3f	?	疑问号

因为空格符在书写时容易出错，所以本书在 general.h 中定义了一个宏:

```
#define BLANK ' '
```

7.1.2 字符输入 / 输出

在 Turbo C 中，提供了单个字符的输入／输出函数 getchar 和 putchar。这两个函数实际上都是宏:

```
int getchar(void)
int putchar(char)
```

这两个函数的物理意义很清楚，使用也不复杂，但函数本身存在一些令人易混淆之处。请看下面这个例子:

```
#include <general.h>
void main ( )
{
    char ch;
    for (;;) {
        printf("\nEnter a character:    ");
            fflush(stdin);
        ch=getchar( );
        putchar(ch);
        printf("  in  %%d  format:  %d\n",ch);
        printf("It's   ASCII  %d .\n",(unsigned char)ch);
    }
}
```

如果按下大写字母 A，则产生的输出将如下所示。第 1 行行尾的 A 是由 DOS 回显出来的，第 2 行行头的 A 是由 putchar 函数输出产生的。

```
Enter a character:    A
```

```
A  in  %d  format:   65
It's ASCII 65.
```

值得注意的是，getchar 是标准的 C 输入函数。但标准的 C 输入函数是带缓冲的，故第 1 次调用 getchar 时，尽管其本意是读取 1 个字符，但实际上却是等待用户键入一大串字符，直至按回车键。在按回车键后，才会从缓冲区中取 1 个字符返回。在下一次调用 getchar 前必须通过 fflush 函数先把缓冲区刷清，才能取到下一次输入的第 1 个字符。即使用户每次只键入 1 个字符，也必须这样做。这是因为键入 1 个字符后也必须按回车键才能返回，DOS 会在<CR>后面添加一个<LF>。<CR>被抛弃掉了，而<LF>却留在那儿，如果不刷清缓冲区，下一次 getchar 就会读入这个<LF>字符。

最糟糕的是 DOS 的文件结束字符 Ctrl_Z。在敲入 Ctrl_Z 后，getchar 在其返回值中报告已经检测到 C 的文件结束字符 EOF，但却不从缓冲区中把这个 EOF 去掉。

7.1.3 字符的分类和转换

Turbo C 中提供了如下 12 个对字符进行分类的函数。所有这些函数都接收整数 C 作为参数，返回值也是一个整数。如果为真返回非 0 值，否则返回 0 值。

函数名	字 符 集
isascii	ASCII: 码0～127
isgraph	图形: 码33～126
isprint	可打印: 码32 + 图形
isupper	大写字母: 'A'～'Z'
islower	小写字母: 'a'～'z'
isalpha	字母: 大写字母 + 小写字母
isdigit	数字: '0'～'9'
isalnum	字母数字: 字母 + 数字
iscntrl	控制字符: 码0～31 + 码127
isspace	空格: 码9～13 + 码32
ispunct	标点: 控制字符 + 空格
isxdigit	十六进制字符: 数字+'A'～'F'+'a'～'f'

Turbo C 在头文件 ctype.h 中也提供了一些字符转换函数，如：

int toupper(int c)　把小写字母转换为大写字母

int tolower(int c)　把大写字母转换为小写字母

7.1.4 宏和宏的副作用

在头文件 ctype.h 中也提供了上面两个字符转换函数的宏版本：

```
#define   _toupper(c)    ((c)+'A'-'a')
#define   _tolower(c)    ((c)='a'-'A')
```

这两个宏显然不能正确工作，除非在转换为大(或小)写字母之前，字母一定是小(或大)写。因此，为完整地做好这个转换工作，需要首先调用一个函数来检查被转换的字母是什么字母，然后再决定是否进行转换。于是，可以把这个宏修改成如下的形式：

```
#define    toupper1(c)    (islower(c)?_toupper(c):(c))
```

这个宏一般情况下会正确工作。但是，由于这个宏两次计算参数 C，如果参数 C 是一个带有副作用的函数，则会带来一些麻烦。例如，下面这个例子就是以函数 F(int C) 作为宏的参数的，这个函数是有副作用的，它会显示信息"Hello from f!"。

```
#include <general.h>
#include <ctype.h>
#define  toupper1(c) (islower(c) ? _toupper(c) : (c))
int f(int c)
{
     printf("Hello from f ! ");
     return c;
}
void main ( )
{
     printf("_toupper(f('A')) = %c\n", _toupper(f('A')));
     printf("toupper1(f('A')) = %c\n", toupper1(f('A')));
     printf(" toupper(f('A')) = %c\n",  toupper(f('A')));
}
```

由于函数的副作用，这个程序输出的结果将如下所示：

```
Hello from f !  _toupper(f('A')) = !
Hello from f !  Hello from f !  toupper(f('A')) = A
Hello from f !   toupper(f('A')) = A
```

3 行输出结果分别由 Turbo C 的宏_toupper、这儿定义的宏 toupper1 和 Turbo C 的函数 toupper 产生的。输出结果最前面的"Hello form f !" 都是由于计算参数而产生的。宏 toupper1 两次计算参数，所以产生两个"Hello from f !"。还可以看出：Turbo C 宏_toupper 的输出结果是不对的，它把大写字母'A'转换成了警叹号'!'。

通用的宏不应该多次计算它的参数。避免重复计算参数的方法就是先把参数放在某一个变量内，如下所示：

```
char v;
#define toupper2(c) (islower(v=(c))?_toupper(v):v)
```

因为宏展开的结果必须是一个符合语法的有效表达式，故不可能在一个宏定义之内去使用一个以前未定义的变量。如果希望这个宏在整个程序内都可以使用，则这个宏定义内的变量 v 必须是全局变量。全局变量又带来一个问题：如果 toupper2 在其执行过程中被第 2 个程序所中断，而这第 2 个程序本身又使用了 toupper2，则第 1 次全局变量 v 的值将被第 2 次调用 toupper2 改写，当从第 2 个程序返回时，第 1 次全局变量 v 的值将不复存在，无法使用。这类似于以后将要讲到的 DOS 的不可重入问题。

由此看来，在宏中两次计算参数和在宏中使用全局变量都可能出问题，究竟怎样避免这些问题呢？在某些情况下，可以用如下的办法。它用到的变量 v 是一个局部变量，而不是一个全局变量。

```
#define toupper3(c) {char v=c; if(islower(v)) c=_toupper(v);}
```

这个宏在堆栈上建立一份c的拷贝v，如果必要，再对v进行转换，接着把结果拷回v，最后从堆栈中把v弹出不要。因为toupper3每次被调用时都建立了一个新的堆栈变量，故它的程序是可重入的。其所以说这个宏只能在某些情况下使用，是因为这个宏展开后的结果是两条语句而不是一个函数，这与前面的习惯用法很不一致.

7.2 字符串

常用的表示1个字符串的结构的方法有4种。

第1种是固定字符串的长度。这种方法很简单，但其缺点是，即使浪费了大量存储空间，仍然可能出现字符串放不下的情况。

第2种是在结构中包含字符串长度信息，这个信息放在字符串的最前面。DOS和Turbo Pascal就是这样处理的，它们用1个字节来表示字符串的长度。这种方法的优点是容易取得长度信息，并由此而使某些操作比较简捷。缺点是字符串长度受限制（在Turbo Pascal中是256字节），而且连续的数据存储使得重新安排字符串相当耗费时间。

第3种是采用链表。这种方法容易操作，字符串长度也不受限制，其代价是要为链指针分配空间，从链中逐个结点地寻找某一个字符串也比较耗费时间。下一章将讲到用链表表示动态通用串的方法。

第4种就是C语言中采用的方法，它是在字符串的末尾添加1个表示结束的特殊字符NUL('\0')，即ASCII码0，称为ASCIIZ串。这种方法对字符串的长度也没有限制。但是，不易得到字符串长度信息，必须从头扫到尾，逐个计数。另外，因为ASCII码0已经作为字符串终止符使用，所以在字符串中就不能再使用ASCII码0。尽管ASCII码0一般很少使用，但在用ASCII码字符对数字信息进行编码的情况下（如在某些打印机控制系列中），却要求把ASCII码0也作为1个字符使用，这时就有些困难。

在标准C语言中，字符串是用双引号括起来的，字符串如果在一行内写不下，应在末尾加一个反斜线，然后在下一行内接着写。Turbo C 放松了这个要求，如:

```
main( )
{
    char *p;
    p="This is an example of how Turbo C "
      " will automatically\ndo the concatenation for "
      "you on very long strings, \nresulting in nicer"
      " looking program.";
    printf(p);
}
```

产生的输出结果将如下所示:

```
This is an example of how Turbo C will automatically
do the concatenation for you on very long strings,
resulting in nicer looking program.
```

字符串可以空，即可以是 "" ，但这容易写错，故本书在头文件 general.h 中定义了如下两个宏：

```
#define EMPTYSTR ""
#define STREMPTY(s)  (*(s)==0)
```

前一个宏定义一个空字符串，后一个宏用来判断一个指针所指向的字符串是否为空。

前面已经说过，在 C 中计算 1 个字符串的长度是比较麻烦的，这是就它的本质而言的。对 C 程序员来说，只要调用如下这个函数就可以得到字符串的长度：

```
size_t strlen(const chra *s)
```

数据类型 size_t 与 unsigned int 在使用上没有什么差别。

经常使用的字符串输入／输出函数是：

```
char *gets(char *s)
int puts(const chra *s)
```

这两个函数的说明在头文件 stdio.h 中。gets 在取键盘输入时，直到用户按回车键才会返回。DOS 把这个<CR>字符读入标准输入缓冲区，并在其后添加一个<LF>。当函数 gets 把键盘输入移到字符串 S 中时，它只从缓冲区去除了<CR>，却留下了一个<LF>。所以，在下次输入前，应先调用 fflush(stdin)刷清缓冲区。puts 函数则在输出的 ASCIIZ 串后面补加了<CR><LF>。如果不希望补加<CR><LF>，则需要用到第 4 章讲过的许多其它屏幕处理技术。

经常用到的另一对字符串输入／输出函数是 printf 和 scanf，它们是带格式化的，请参见第 7.7 节。

7.3 字符串的分析

所谓字符串分析，就是取得一些有关字符串的信息，但不改变字符串。这一类工作包括：取字符串长度，判断 1 个字符串是否为空，比较两个字符串，在一个给定的字符串内寻找 1 个字符。在 Turbo C 中，这些函数的名字都以“str”开头，其说明都在头文件 string.h 中。

(1) 取字符串长度的函数是：

```
size_t strlen(const char *s);
```

(2) 判断 1 个字符串是否为空，只要判断字符串的第 1 个字节是否为 ASCII 码 0 即可，如下面这个宏定义：

```
#define STREMPTY(s) (*(s)==0)
```

(3) 经常要对字符串进行比较。对于文本字符串来说，顺序是很重要的。如果两个文本字符串不相等，则还要根据词典顺序，说明哪一个字符串在前(较小)，哪一个字符串在后(较大)。Turbo C 中提供了 4 个这样的函数：

```
int strcmp(char *s1,char *s2);
int stricmp(char *s1,char *s2);
```

```
int strncmp(char *s1,char *s2,unsigned n);
int strnicmp(char *s1,char *s2,unsigned n);
```

在这几个函数中，参数s1和s2是欲进行比较的两个函数。都是按无符号字符进行比较，返回结果的形式也一样:

小于0	如果s1小于s2
等于0	如果s1等于s2
大于0	如果s1大于s2

strcmp区分大小写字母，stricmp不区分字母大小写，一律按大写处理。在ASCII码表中，由于26个大写字母与26个小写字母不是连续排列的，中间插有ASCII码91～96这6个符号，因此，如果被比较字符串中含有这6个符号，则两个函数的比较结果不一样。例如，如果用strcmp，方括号"["(ASCII码91)应在'a'前面，如果用stricmp，方括号则在'a'后面。这是因为，'a'被当作'A'处理了。

strncmp和strnicmp与前两个函数类似，但它们只比较前 n 个字符。

Turbo C 中还有另两个函数strcmpi和strncmpi，它们与stricmp和strnicmp完全一样，只是strcmpi 和 strncmpi 是两个宏，它们的宏定义在头文件string.h中，通过宏再把strcmpi翻译成stricmp，所以引用时还必须加include <string.h> 语句。

Turbo C 的字符串比较函数有一个缺点，它们不报告从哪一个位置开始两个字符串不相等。下面这个程序(函数strequalsf)可以完成这个任务。这个函数有4个调用参数，s和t是指向两个被比较的字符串的指针，ps和pt又是指向s和t的两个指针。函数不但通过返回值表示两个字符串是否相等，还通过ps和pt表示在什么位置不相等。如果相等，返回值为1，返回的指针指向终止符ASCII码0。如果不相等，返回值为0。返回的指针指向第1个不同字符处。如果一个字符串正好是另一个字符串的子字符串，则返回值也为0，返回的指针指向子字符串的终止符处。

值得注意的是，这个程序在while的条件内用了赋值语句，一般不建议这样做，因为可能引起警告信息（请参考Turbo C 参考手册的附录B)。另外，这个程序中的ps和pt是指针的指针。这是因为，根据c的"传值"规则，要想在函数中修改调用者变量的值，必须通过指针进行。而在这个函数中，ps和pt本来已是指向字符串的指针，所以要想修改这两个指针，就只能用指针的指针了。

```
int strequalsf(char *s,char *t,char **ps,char **pt)
{
    int equals;
    *ps = s;
    *pt = t;
    while ((equals = (**ps == **pt)) && (**ps != 0)) {
        ++*ps; ++*pt;
    }
    return equals;
}
```

在某些情况下，也可能需要按从后往前的相反顺序处理字符串。下面这个函数

strequalsb 与函数 strequalsf 实现的是同一功能，但前者是从后往前比较。另外，这个函数也用到 4 个调用参数，s 和 t 也是指向被比较的两个字符串，但另两个参数 pm 和 pn 不是指针的指针，而是调用程序中用来存储两个字符串长度的变量。也就是说，这个函数不是按指针而是按索引(或者说按数组)返回两个字符串第 1 个彼此不等的字符位置。如果相等，返回值为 1，并设置 * pm=0= * pn。如果不相等，返回值为 0，并设置 * pm 和 * pn 为第 1 个不同字符的索引号。如果一个字符串是另一个字符串的子字符串，返回值也为 0，这时 * pm 和 * pn 中有一个为 0，另一个将是子字符串的开始索引号。

```
int strequalsb(char *s,char *t,unsigned *pm,unsigned *pn)
{
    *pm = strlen(s);
    *pn = strlen(t);
    while ((*pm > 0) && (*pn > 0) && (s[*pm] == t[*pn])) {
        --*pm; --*pn;
    }
    return (*pm == 0) && (*pn == 0) && (s[*pm] == t[*pn]);
}
```

(4) 经常需要在 1 个字符串中搜索某一个字符，Turbo C 中的函数 strchr 和 strrchr 可以完成这个任务。

```
char *strchr(const char *s,int c);
char *strrchr(const char *s,int c).
```

函数 strchr 从前往后找，函数 strrchr 从后往前找。两个函数的返回值都是一个指针，如果找到了，这个指针就指向该字符第 1 次出现的位置，如果找不到，这个指针就指向 NULL。这两个函数都是用 C 语言写的，如果知道被搜索的字符串长度，用第 2 章中的 memchr 函数速度会更快一些。

```
void *memchr(void *s,int ch,unsigned n);
```

这两个函数的调用参数都只有一个指向字符串的指针，即一定是从字符串的开头或结尾处开始查找。可是在很多情况下，要求在找到一个以后，下次从这个位置接着往下找，找同一个字符的第 2 次出现。对于从前往后查找，这个问题还好解决，因为可以把函数 strchr 返回的指针作为第 2 次查找的字符串指针。对于从后往前查找，则必须对 strrchr 做些修改，修改后的函数称之为 strchrb。

```
char *strchrb(char *s,char *p,char c)
{
    unsigned i = p-s;
    while (s[i] != c && i != 0) --i;
    return(s[i] == c ? s + i : NULL);
}
```

这几个函数返回的都是指针，Turbo C Tools 提供的一个函数则是返回索引号，即

位置号。

```
int stschind(char check,const char * psearch);
```

参数 check 是待查找的字符，psearch 指向被查找的字符串，返回值是查找到的位置。如果在开头，返回值为 0，如果未找到，返回值为-1。

(5) 实际应用中还经常需要从 1 个字符串中查找某几个字符是否存在，例如在表达式字符中查找是否有"+ － * /"运算符。Turbo C 的函数 strpbrk 就是这样一个函数:

```
char * strpbrk(const char * s1,const char * s2);
```

参数 s1 指向被查找的字符串，s2 指向的字符串是由欲查找的字符组成的。这个函数返回一个指针，指向 s1 中第 1 次出现 s2 中所含字符的位置。

实际应用中也经常需要从 1 个字符串中查找某几个字符是否不存在。Turbo C 中没有提供这样的函数，但下面几条语句可以实现这样一个函数，我们称它为 strcpbrk。

```
char * strcpblk(char * s,char * ignore)
{
    unsigned n = strspn(s,ignore);
    return(n! = strlen(s)?s+n:NULL);
}
```

这个函数(程序)中用到了 Turbo C 的另两个字符串分析函数:

```
size_t  strcspn(const char * s1,const char * s2);
size_t  strspn(const char * s1,const char * s2);
```

这两个函数的返回值类型都是 size_t，实际上是返回 1 个整数，说明字符串 s1 中某一段的最大长度，这一段完全是由 s2 中的字符组成(strspn)，或完全不出现 s2 中的字符(strcspn)。

很容易联想到，函数 strpbrk 和 strcpbrk 也应该有其相应的反向查找函数，但 Turbo C 中没有提供这样的函数。下面利用 Turbo C 的 strrev 函数，编出一个与 strpbrk 相对应的反向查找函数，称为 strpbrkb。

```
char * strpbrkb(char * s,char * t,char * seek)
{
    char * r, * q;
    r = strrev(strdup(s));
    q = strpbrk(r,(char * )seek);
    free(r);
    return (q == NULL ? NULL : t - (q - r));
}
```

这个函数在字符串 s 中进行反向查找，从指针 t 所指的位置开始，直到字符串的开头，查找是否存在 seek 字符串中的字符。如果存在，则返回一个指针指向找到的第 1 个字符位置。如果不存在，则返回一个空指针 。

(6) 最后一个任务就是在 1 个字符串中查找是否存在另一个子字符串。

Turbo C 中有这样一个函数:

```
char *strstr(const char *s1,const char *s2);
```

这个函数在 s1 中查找，看看是否存在子字符串 s2。如果存在，则返回的指针指向 s1 中开始出现 s2 的位置。如果不存在，则返回一个空指针。Turbo C 中没有相应的反向查找函数。

7.4 字符串的综合

所谓字符串的综合，就是讨论字符串的建立、填充、复制、连接、拷贝、插入和对齐。对于这些操作，Turbo C 中已经提供了不少函数，但仍有一些不足，尤其是在字符串的插入和对齐方面。

(1) 在 C 语言中，为建立一个长度为 n 的字符串，第 1 步就是要通过数组或指针为字符串分配(n+1)个字节存储单元:

```
char    s[n+1];
s=(char *)malloc(n+1);
```

第 2 步是初始化这个字符串，包括清除所分配空间原来的内容，建立字符串终止符，但 Turbo C 中没有这样的函数。

(2) Turbo C 提供了如下两个字符串填充函数:

```
char *strset(char *s,char ch);
char *strnset(char *s,char ch,unsigned n);
```

函数 strset 以字符 ch 填充整个字符串 s，函数 strnset 也是以字符 ch 填充字符串 s，但至多填充 n 个，所谓至多，就是在遇到 s 的终止符时也结束填充操作，尽管这时有可能没有填够 n 个字符。从这个执行过程可以看出，这两个函数都要求 s 是已初始化了的字符串，而不仅仅是分配了一个空间。

(3) Turbo C 提供了一个复制函数:

```
char *strdup(const char *s);
```

字符串 s 是源字符串，函数负责为新字符串分配空间，返回指向新字符串的指针。但是，这个函数不负责释放新字符串所占用的空间，这件工作必须由程序员做。

(4) Turbo C 提供了两个字符串连接函数:

```
strcat(char *t,char *s);
strncat(char *t,char *s,unsigned n);
```

连接是指把源字符串连到目标字符串的后面，连接后补 1 个字符串终止符。strncat 最多把源字符串的前 n 个字符连到目标字符串。所谓最多，就是当源字符串长度小于 n 时，则只把源字符串连接过去。

(5) Turbo C 提供了 3 个字符串拷贝函数:

```
strcpy(char *t,char *s);
strncpy(char *t,char *s,unsigned n);
stpcpy(char *t,char *s);
```

函数 strcpy 和 stpcpy 都是把整个源字符串(直到终止符为止)拷贝到目标字符串，只不过 strcpy 返回的指针是指向目标字符串的头，而 stpcpy 返回的指针是指向目标字符串的尾，即终止符。stpcpy 函数的这个特性使程序员可以很容易把 1 个字符串重复多遍，构成 1 个新字符串。保证目标字符串有足够的存储空间是程序员的责任。

函数 strncpy 则不同，它只拷贝源字符串的前 n 个字符。应该注意，与函数 strnset 和 strncat 等不一样，这里不是“最多”n 个字符，而一定是 n 个字符。如果源字符串的长度不够 n 个，则会以 ASCII 码 0 补充，而不是把源窗口字符串后面的内容拷贝过去。如果源字符串的长度等于或大于 n，则只拷贝前 n 个字符，但这时生成的目标字符串没有终止符，这一点应特别注意。

(6) Turbo C 中没有提供在 1 个字符串中插入另一个字符或字符串的方法，本书提供的下一个函数可以解决这个问题。它在字符串 t 的位置 p 处插入字符串 s。但这个函数并不破坏字符串 t，而是重新分配一块内存来存放新字符串，返回的也是指向这个新字符串的指针。

```
char *strinsstr(char *t,char *p,char *s)
{
    unsigned l,m,n;
    char *u;
    l = strlen(t);
    n = strlen(s);
    u = (char *) malloc(l+n+1);
    if (u == MULL) return MULL;
    m = p-t;
    memcpy(u,t,m);
    memcpy(u+m,s,n);
    memcpy(u+m+n,t+m,l-m+1);
    return u;
}
```

(7)Turbo C 中也没有提供字符串对齐函数，Turbo C Tools 中提供了这样一个函数:

```
char *stpjust(char *ptarget,const char *psource,
              char fill,int fldsize,int code);
```

参数 fldsize 说明域宽，ptarget 指向的缓冲区用来存放生成后的结果，它至少应该为(fldsize + 1)字节长。参数 psource 指向源字符串，如果源字符串的长度比 fldsize 小，则用参数 fill 填充。参数 code 指定对齐类型:

JUST_LEFT	-1	左对齐
JUST_CENTER	0	居中
JUST_RIGHT	1	右对齐

7.5 字符串的操作

这里说的字符串操作包括字符串的反转、替换、删除和分解。

(1) Turbo C 中提供了一个把字符串反转的函数:

```
char strrev(char *s);
```

这个函数原地反转，不建立新的拷贝。

(2) 字符替换是字符串替换的最简单情况，Turbo C 中没有这样的函数，Turbo C Tools 中提供了这样一个函数。

```
char *stpxlate(char *source,char *table,char *trans);
```

3 个指针参数分别指向源字符串、被替换字符表、替换字符表。替换是原地进行的，返回的仍是指向源字符串的指针。一般情况下，两张表应该一样长，才能按顺序对应替换。如果替换字符表比被替换字符表还长，则长出的部分不用。如果替换的字符表比被替换的字符表短，则以空格符来代替。例如:

被替换字符表	abc	abac
替换字符表	123	123
源字符串	aaacbbb	aaacbbb
替换后的结果	1113222	111 222

这个函数最常用于把 1 个字符串中的大写字母变为小写字母，或反之。这个工作也可以通过其它函数来完成，见 7.6 节。

(3) 字符串替换是每一个文本编辑软件必须具备的功能，这比字符替换麻烦多了。Turbo C 和 Turbo C Tools 中都没有提供这样的函数，下面的程序(函数)strsubst 可以做这种工作。

```
static unsigned lengths,lengtha,lengthb,numberofas;
static char *pa,*pb;
char *strsubst(char *s,char *a,char *b)
{
    if (strempty(a)) return NULL;
    lengths = strlen(s);
    lengtha = strlen(a);
    lengthb = strlen(b);
    pa = a;
    pb = b;
    numberofas = 0;
    return subst(s);
}
static char * subst(char *ps)
```

```
{
    char  * p, * t;
    unsigned l,m;
    if ((p = strstr(ps,pa)) == NULL) {
        l = lengths + numberofas * (lengthb -lengtha);
        t = (char  * ) malloc(l+1);
        if (t == NULL) return NULL;
        m = strlen(ps);
        t += l - m;
        memmove(t,ps,m+1);
        return t;
    }
    ++numberofas;
    if ((t = subst(p+lengtha)) == NULL)
        return NULL;
    t -= lengthb;
    memmove(t,pb,lengthb);
    m  = p - ps;
    t -= m;
    memmove(t,ps,m);
    return t;
}
```

程序主要由函数 strsubst 和 subst 两部分组成，主要工作是由函数 subst 做的，strsubst 只不过是一个调用接口。

字符串的替换过程本质上是一个递归过程，所以函数 subst 采取的是递归算法。为了便于理解这个函数，不妨看一个实际替换例子。

```
假设:       源字符串                 s="xaaxyaaaxyz"
            被替换字符串             A="aa"
            替换字符串               B="&"
分析过程:   第 1 次  被分析字符串    xaaxyaaaxyz      把指针 1 和 2 压入
                                     ||               堆栈
                     找到第 1 个 A   12
            第 2 次  被分析字符串    xyaaaxyz         把指针 3 和 4 压入
                                     | |              堆栈
                     找到第 2 个 A   3 4
            第 3 次  被分析字符串    axyz
                     没找到 A
生成过程:   拷贝 s 的最后部分        axyz             生成结果
            用指针 3 和 4            xy&axyz
            用指针 1 和 2            x&xy&axyz
```

替换过程由分析过程和生成过程两大步组成。分析过程是从左往右进行的，在分析中，要记录被替换字符串中 A 出现的次数(变量 numberofas)，以便计算出需要分配多大

的空间来存放最后生成的结果。还要记录每次分析的起始点(指针 ps)，所找到的被替换字符串 A 出现的位置(指针 p)，以便于以后生成。生成过程则是从右往左进行的。对于源字符串中不含被替换字符串 A 的尾部，只要拷贝到目标字符串即可。每生成一次，记录下指向部分生成结果的指针，以便下次接着往左生成。这个指针在程序中就是指针 t，也就是递归函数 subst 返回的指针。根据指针 t 以及分析过程中保存的指针 p 和 ps，就可以逐次往左生成了。

(4) 从字符串中删除某些子字符串也是常见的操作，Turbo C 和 Turbo C Tools 都没有提供相应的函数。在实际处理过程中，可以把删除看作是替换的一种特例，即替换字符串是空字符串。函数 strsubst 允许替换字符串是空字符串。

源字符串	Repeal Coopers beeper
被替换字符串s1:	pe
替换字符串s2:	(空字符串)
strsubst(s, s1, s2)的结果:	Real Coors beer

(5) 字符串的分解是按指定的分界符把字符串分解成单个小段，每小段称为一个 token。字符串分解技术在语言编译中经常要用到。Turbo C 中提供了这样一个函数:

```
char *strtok(char *s1,char *s2);
```

参数 s1 指向被分解的字符串，s2 指向的字符串是由分界符组成的，返回的指针指向在 s1 中找到的第 1 个 token。如果没有找到，则返回一个空指针。这个函数不重新分配内存，而是原地分解。在第 1 次找到 1 个 token 后，函数自动在紧随其后的位置用 1 个字符串终止符(ASCII 码 0)代替原来的分界符。为了接着寻找第 2 个 token，第 2 次调用 strtok 函数时 s1 应为空指针 NULL。函数并不真正用这个空指针，而是在其内部有一个静态变量，记录了第 2 次寻找时应从什么位置开始。通过这种办法，就可以逐次把源字符串分析完毕，如下例所示。

```
#include <stdio.h>
#include <string.h>
main( )
{
    char *ptr;
    ptr=strtok("FEB.14,1988",".,-/");
    while(ptr!=NULL) {
        printf("ptr=%s\n",ptr);
        ptr = strtok(NULL,".,-/");
    }
}
```

这个例子产生的输出将是:

```
ptr=FEB
ptr=14
ptr=1988
```

7.6 文本字符串

上面的许多函数也可以用来处理文本字符串，但本小节所述的函数是专门针对文本字符串的，主要包括：大小写字母之间的转换，横向制表符 TAB 与空格符之间的转换，其它转换。

(1) Turbo C 中提供了两个大小写字母之间的转换函数：

```
char * strlwr(char * s);
char * strupr(char * s);
```

前者把指定字符串内的大写字母变为小写字母，后者把小写字母变为大写字母，其它字符不变。

(2) Turbo C Tools 中提供了两个横制表符与空格符之间的转换函数：

```
char * stptabfy(char * psource,int incx);
char * sepexpan(char * ptarget,char * psource,int incr,
                int tarsize);
```

函数 stptabfy 用 tab 字符替换空格，参数 psource 指向源字符串，参数 incr 是两条制表线之间的距离，返回的指针指向替换后的字符串。替换是原地进行的，空间也足够用。

函数 stpexpan 则是把 tab 字符扩展成 1 个或多个空格字符。因为扩展后空间将增大，故不是在原地进行，参数 ptarget 指向目标字符串地址，参数 tarsize 说明目标字符串的容量。如果 tarsize 不够大，则返回的指针指向源字符串中尚未被替换的下一个字符。如果 tarsize 足够大，则替换完成后，返回空指针。

(3) Turbo C Tools 中提供了另一个用途更广泛的字符串转换函数：

```
char * stpcvt(char * psource,int conv);
```

参数 conv 说明如何进行转换，它可以是下列值的逻辑“或”：

符号	值	意　义
RWHITE	1	删除所有空白字符
RLWHITE	2	删除所有前导空白字符
RTWHITE	4	删除所有尾随空白字符
REDUCE	8	将一片连续空白字符压缩成1个空格字符
NOQUOTE	16	括在单引号和双引号中的字符不作转换
TOUP	32	将小写字母转换为大写
TOLOW	64	将大写字母转换为小写
RCONTROL	128	删除所有控制字符

空白字符的定义是 ASCII 码 9～13 和 32，控制字符的定义是 ASCII 码 0～31 和 127。

7.7 数字字符串

数的表示法分内部表示和外部表示。在机器内部，数可以有 4 种表示：短整数，长整数，短浮点数，长浮点数。在机器外部，其表示法有二进制、八进制、十进制、十六进制，还有科学记数法等等。当数从外部（如键盘）读入机器内时，需要把外部表示法变为内部表示法，当数从机器内输出到外部（如显示器）时，需要把内部表示法变为外部表示法。

Turbo C 中提供了 7 对这样的转换函数：scanf 和 printf,cscanf 和 cprintf，fscanf 和 fprintf，sscanf 和 sprintf，vscanf 和 vprintf，vfscanf 和 vfprintf，vsscanf 和 vsprintf。

scanf 和 printf 是最基本的一对，表示标准输入／输出。

加前缀c，　表示控制台(console)，输入来自键盘，输出到屏幕。

加前缀f，　表示流文件(file)。

加前缀s，　表示字符串(string)，即内存中的一个缓冲区。

加前缀v，　表示字符串(variable)参数个数可变，其参数不是一个参数表，而是一个指向参数表的指针。

加前缀vs，综合前缀"v"和"s"的含义。

加前缀vf，综合前缀"v"和"f"的含义。

尽管这 7 对函数中也考虑了其它数据，如字符和字符串的输入输出转换，但主要的还是为了数的转换。Turbo C 中还提供了许多其它数的转换函数，如：strtol、strtoul、strtod、atof、atoi、atol、itoa、ltoa、ultoa、ecvt、fcvt、gcvt，等等。这些函数特别繁琐，用得也不多，这里就不解释了。下面首先集中讨论 printf 函数。

```
int printf(char * format[,parameter,...]);
```

参数 format 是格式字符串，它由两类字符串组成。一类是普通字符串，它被原封不动地输出。另一类是以%开头的转换格式字符串，它说明以什么样的格式输出随后的参数表中的各参数。转换格式字符串的个数应与参数个数相同。

格式字符串本身的格式如下：

```
% [flags] [width] [.prec] [F|N|h|l|L] type
flags:          正负号修正
width:          宽度修正
prec:           精度修正
F,N,h,l,L:      参数类型修正
type:           参数类型
```

在转换格式字符串的 5 部分中，只有类型 type 是必须有的，其余都是可选的。下表列出了在前 4 个可选部分都不存在的情况下 type 的各种字符及其含义。如果前面有可选部分，则其含义可能被修正。

type数值	输入参数	输出格式
d	整数	有符号十进制整数
i	整数	有符号十进制整数
o	整数	无符号八进制整数
u	整数	无符号十进制整数
x	整数	无符号十六进制整数(a,b,c,d,e,f)
X	整数	无符号十六进制整数(A,B,C,D,E,F)
f	浮点数	形为[-]dddd.dddd的有符号值，小数点后的位数等于精度。
e	浮点数	形为[-]d.ddd....e[+／-]dddd的有符号值，小数点后的位数等于精度，指数部分至少两位。
g	浮点数	与e或f格式相同，取决于所给定的值和精度.
E	浮点数	与e格式相同，只是指数用大写字母E。
G	浮点数	与g格式相同，只是指数用大写字母E(如果需用e格式)。
字符		
c	字符	单个字符
s	字串指针	输出整个ASCIIZ串，除非被精度要求所限
%	无	百分号"%"本身被输出
指针		
n	整数指针	在参数所指向的位置存入当前为止已输出的字符数
p	指针	把参数按指针形式打印出，远指针打印为xxxx:yyyy，近指针打印为YYYY

下面说明格式字符串的 4 个可选择部分。

(1) flags

flags 可以是如下 4 种符号:

flags	意　义
-	左对齐，缺省情况下为右对齐。左对齐时，右边补空格符。右对齐时，左边补空格符。
+	带符号数总是以正号(+)或负号(-)开头
空格	如果数大于等于0，则以空格开头。如果数小于0，则仍以负号开头。
#	对type字符进行修正

type	修正
c,s,d,i,u	无影响
o	在非0数前加0。
x,X	在数前加0x或0X
e,E,f	输出结果中总要有一个小数点，即使小数点后面没有数。
g,G	与e和E一样，只是不去掉尾部的0,

(2) width

width 说明输出数占的最小宽度。如果输出数所需要的位数比 width 少，则左对齐或右对齐输出结果，并以空格或数字 0 填充。反之，则扩大宽度，直至满足要求。

width 有两种说明方法，一种是直接在 width 选择项处用十进制数说明，另一种是在 width 选择项处写 1 个星号"＊"，这时实际的宽度来自参数表中的对应参数，这个参数必须是整数，紧随其后的那个参数才是实际输出的数。

width	含　义
n	至少n个字符，无论左对齐还是右对齐，一律用空格符填充
0n	至少n个字符，左对齐时用空格填充，右对齐时用"0"填充

*	用参数表中的参数来说明宽度

(3) precision

precision 总是以小数点开始，以便与 width 相区分。precision 也有两种说明方法。

prec	含　义
没有	精度置为缺省值: 对于d,i,o,u,x,X,缺省为1 对于e,E,f，缺省为6 对于g,G，缺省为全部有效数字 对于s,缺省为整个ASCIIZ串
.0	对于d,i,o,u,x，置为缺省 对于e,E,f，不输出小数点
.n	对于d,i,o,u,x,X,至少n个数字。如果输出值小于n个数字，则右对齐，左边补数字0。 如果输出值大于 n 个数字，则不被截短 对于e,E,f,小数点后输出n个数字，最后1位四舍五入 对于g,G，至多n个有效数字 对于c，没有影响 对于s，至多输出n个字符
.*	精度由参数表中的参数决定

(4) F,N,h,l,L

F 和 N 是对指针类型的参数(%p，%s 和%n)进行修正，分别修正为远指针和近指针，否则按编译模式来解释指针参数的类型。h、L 和 l 是对数值型参数进行修正。

类型修正	含　义
F	按远指针读入参数
N	按近指针读入参数，在huge模式下不可用
h	对于d,i,o,u,x,X,按短整数解释
l	对于d,i,o,u,x,X,按长整数解释 对于e,E,f,g,G,按倍精度double解释
L	对于e,E,f,g,G,按长倍精度long double解释

scanf 函数比 printf 函数复杂，因为它是用于输入，而输入的格式更难于控制。

```
int scanf(char *format[,parameters,...]);
```

参数 format 也是格式字符串，它由 3 类字符组成。

第 1 类是空白字符，包括空格符、制表符和回车符。遇到这类字符时，函数照样从输入设备上读，直至读到下一个非空白字符为止。读入的内容不存入输出设备。

第 2 类是转换格式字符串，这类字符串以%开头，具体格式下面还会讲到。遇到这类字符串时，函数从输入设备上读入一段字符，按要求的格式进行转换，转换后的结果再存入相应参数所指定的地址中。

第 3 类是非空白字符，也就是除上述两类以外的其它字符。遇到这类字符时，函数照样从输入设备上读，且要求读入的字符与格式字符串中的字符相匹配，读入的结果不存入输出设备。

输入设备，例如键盘上敲入的字符，是连续的，怎样才能把它们分隔成段，使它们分

别对应于各参数呢？一般情况下，空白字符，包括空格符、制表符和回车符，可作为段的分界符。但在下列 4 种情况下，非空白字符也可能成为段的分界符。

(1) 遇到了在当前转换格式定义下不能进行转换的字符，这个字符就成为下一段的第 1 个字符。

(2) 已读完了宽度说明中所指定的 n 个字符，下一个字符将成为下一段的第 1 个字符。

(3) 在转换格式%[...]中，遇到了一个未包含在查找字符集中的字符，这个字符将成为下一段的第 1 个字符。

(4) 在转换格式%[^...]中，遇到了一个包含在查找字符集中的字符，这个字符将成为下一段的第 1 个字符。

在下列情况下，scanf 函数将终止它的工作，返回到调用者。

(1) 转换格式说明已经用完。

(2) 遇到了文件结束符 EOF。

(3) 遇到了一个与 format 格式字符串中非空白字符不相匹配的字符。即使在返回后，这个不匹配的字符也没有从输入设备上被读走。

scanf 函数的转换格式字符串的格式如下:

% [*] [width] [F|N] [h|l|L] type

与 printf 函数相比，这里缺了精度修正 prev 和正负号修正 flags，但多了一个抑制修正符*。

修正符	含　义
*	继续按所说明的类型从输入设备上读完当前段，但读入的内容不存入输出设备
width	最多读n个字符，然后转换并存入输出设备。如果在读n个字符之前，已经遇到空白字符或不可转换字符，则不再读入。
N,F	推翻缺省的或说明的大小，参数被解释为远指针(F)或近指针(N)。N不能用于巨编译模式
h	对于d,i,o,u,x类型，把输入转换为short int，存入short目标 对于其它，没有影响
l	对于d,i,o,u,x类型，把输入转换为long int，存入long目标 对于e,f,g类型，把输入转换为double，存入double目标 对于其它，没有影响
L	对于e,f,g类型，把输入转换为long double，存入long double目标。对于其它，没有影响
type	参数的类型，见下面的表格。

在转换格式字符串中，类型说明 type 是必须的，它可以有下表所示的一些格式，表中各项的含义是假设在类型说明前没有加任何别的修正说明。

类型	所期望的输入	参数类型
数值		
d	十进制整数	int * arg
D	十进制整数	int * arg
o	八进制整数	int * arg
O	八进制整数	long * arg
i	十进制、八进制或十六进制整数	int * arg
I	十进制、八进制或十六进制整数	long * arg
u	无符号十进制整数	unsigned int * arg
U	无符号十进制整数	unsigned long * arg
x	十六进制整数	int * arg
X	十六进制整数	long * arg
e	浮点数	float * arg
E	浮点数	float * arg
f	浮点数	float * arg
g	浮点数	float * arg
G	浮点数	float * arg
字符		
s	字符串	char arg [ ]
[字符集]	字符串	char arg [ ]
c	字符	char * arg
%	字符%	不需转换，原样存入
指针		
n		int * arg
p	十六进制形式 YYYY:ZZZZ 或 ZZZZ	far * 或 nera * 由编译模式决定

关于类型转换，需要补充说明几点。

(1) %c，读下一个字符，包括空白字符。如果希望跳过空白字符而读下一个非空白字符，则应该用%1s 。

(2) %wc，这里的 w 表示宽度 width，连续 w 个字符，存入一个数组。

(3) %s，读入一段字符串，存入时自动补 1 个终止符。

(4) %[字符集]，与%s 类似，只是输入的字符串必须由字符集中的字符组成。

(5) %[^字符集]，只是输入的字符串不能包含字符集中的字符。

在这两个说明的字符集中可以用连字符“–”。如:

%[abcd]与%[a–d]是等效的。

%[^0123456789]与%[^0–9]是等效的

(6) %e，%E，%f，%g 和%G 要求输入的形式如下所示:

[+/-]ddddddddd [.]dddd[E|e][+/-]ddd

这里，带方括号的项表示是可选的，d 是十进制或八进制或十六进制字符。

7.8 国家和货币字符串

各个国家的语言一般是不同的，其货币、日期、时间、数据等的书写格式也可能不同。DOS 是通过码页(code page)机制来提供与国家有关的信息格式，它把大于 127 的 ASCII 码用于各种语言的非英文文字符。但是，中文是大字符集文字，DOS 至今没有提供这样的支持，汉化的 DOS 都是建立在美国英文的基础上，所以本书不再详细介绍有关码页的问题，只指出指定给美国的码页是 437。

借助于 CONFIG.SYS 文件中的 COUNTRY，用户可以在启动操作系统时告诉 DOS 希望用哪个国家的信息格式。COUNTRY 命令的格式如下:

COUNTRY=XXX,[YYY]，文件名

XXX 是一个 3 位数字，是国际电话系统为各个国家分配的国家码，即拨国际长途电话时用的那个号码。YYY 是 DOS 赋于的码页。文件名所指明的文件内含有这个国家的信息格式。如果启动时没有这条命令，则隐含指美国，其国家码为 001，码页为 437，文件名为 COUNTRY.SYS。

Turbo C 提供了一个相应的 country 函数，用来读取指定国家的信息:

```
struct country *country(int xcode,struct country *cp);
```

参数 xcode 是国家码，参数 cp 是指向结构 COUNTRY 的指针。这个结构用来存放有关国家的格式信息，返回的指针也指向这个结构。结构的定义在头文件 dos.h 中，如下所示:

```
struct country {
        int  co_date;
        char co_curr[5];
        char co_thsep[2];
        char co_desep[2];
        char co_dtsep[2];
        char co_tmsep[2];
        char co_currstyle;
        char co_digits;
        char co_time;
        long co_case;
        char co_dasep[2];
        char co_fill[10]
};
```

所有的字符数组字段都是 ASCIIZ 格式，即都有终止符 ASCII 码 0。与下面讨论有关的只是字段 co_date、co_time 和 co_currstyle。

co_date 说明日期格式，有如下三种值:

0　　美国格式:月,日,年

1　　欧州格式:日,月,年

2　　中国格式:年,月,日

co__time 是时间格式，可以有如下两种值:

0　　12 小时制

1　　24 小时制

co__currstyle 是货币格式，可以有如下 4 种值:

0　　货币符号在前，货币数在后，中间无空格

1　　货币符号在后，货币数在前，中间无空格

2　　货币符号在前，货币数在后，中间有空格

3　　货币符号在后，货币数在前，中间有空格

对于美国和法国，COUNTRY 结构中各字段的值如下所示。

	美国	法国	说明
co__date	0	1	日期格式
co__curr	&	F	货币符号
co__thsep	,	(空格)	千分符号
co__desep	.	,	小数点符号
co__dtsep	__	/	日期分隔符号
co__tmsep	:	:	时间分隔符号
co__currstyle	0	3	货币格式
co__digits	2	2	货币小数位数
co__time	0	1	12/24 小时制
co__dasep	,	;	数据分隔符号

值得注意的是，虽然 DOS 支持不同国家信息格式，但 DOS 本身并不用它。这就是说，即使当前正在使用法国格式，但通过 Turbo C 的 printf 函数(间接通过 DOS)输出的实数的小数点仍然是一个圆点，而不会是逗号，要想输出的数符合法国格式，还必须由程序员去实现。下面就是这样一个函数，它把一个浮点数转换成货币数形式的字符串。函数接收浮点数和国家码为参数，返回指向所生成的货币数字符串的指针。例如，给定浮点数 1234.56，返回￥1234.56(美国)或 1234,56F(法国)之类的字符串。

```
#include <dos.h>
char *strtomoney(double x,int Code )
{
     struct country c;
     char  *t, *s, *m;
     int  point,sign,l;
     country(Code,&c);
     t = fcvt(x,c.co__digits,&point,&sign);
     l = strlen(t) + 12;
     s = m = (char *) malloc(l);
     if (c.co__currstyle & 1 == 0) s=stpcpy(m,c.co__curr);
```

```
    if (c.co_currstyle == 2) *s++ = BLANK;
    if (sign)                *s++ = '-';
    strncpy(s,t,point);
    s += point;
    t += point;
    if (c.co_digits != 0) *s++ = c.co_desep[0];
    strncpy(s,t,c.co_digits);
    s += c.co_digits;
    if (c.co_currstyle ==3)    *s++ = BLANK;
    if (c.co_currstyle & 1 != 0)  s=stpcpy(s,c.co_curr);
    *s = 0;
    return m;
}
```

7.9 日期和时间字符串

日期和时间看来比较简单，但在 Turbo C 中却提供了多达 15 个这方面的函数。

函数较多的原因之一是系统中存在 5 种不同的格式。这些格式之间的转换需要许多函数，如图 7–1 所示。

(1) 内部的 DOS 格式可以参考 DOS 的技术手册，程序员一般不需要关心这个格式。

(2) 结构 time 和 date 的定义如下:

```
struct time {
    unsigned char ti_min;
    unsigned char ti_hour;
    unsigned char ti_hund;
    unsigned char ti_sec;
}
struct date {
    int  da_year;
    char da_day;
    char da_mon;
}
```

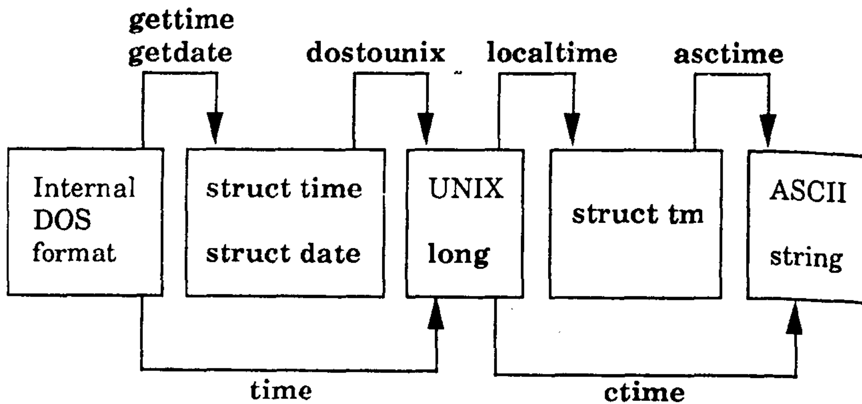

图 7–1　各种日期和时间函数

(3) unix 的格式是一个长整数，单位是秒，其值表示从 1970 年 1 月 1 日以来所走过的秒数。应该注意，DOS(getdate)不报告 1980 年以前的任何日期。

(4) 结构 tm 的定义如下:

```
struct tm {
      int tm__sec;
      int tm__min;
      int tm__hour;
      int tm__mday;
      int tm__mon;
      int tm__year;
      int tm__wday;
      int tm__yday;
      int tm__isdst;
}
```

注意，结构 tm 和结构 date 中关于年和月的格式不同。在结构 date 中，尽管只报告 1980 年以后的年，但字段 da__year 是年的整数值，字段 da__mon 中的月是从 1 开始计数。在结构 tm 中，字段 tm__year 的值是当前年减去 1900 以后的差值，字段 tm__mon 中的月是从 0 开始计数。

(5) 函数 asctime 和 ctime 返回的是一个表示日期和时间的 ASCII 字符串，其格式如下:

Sat Apr 23 23:13:14 1991

最前面的两个字分别是星期和月份的英文缩写，这个例子显示的是 1991 年 4 月 23 日 13 分 14 秒，星期日。

在处理日期和时间时还要考虑到另外两个问题：时区差和夏时制。在 Turbo C 中，是通过两个全局变量来处理这个问题的。

全局变量 daylight 表示夏时制是否起作用，0 表示标准时间，1 表示夏时制。但是，各国的夏时制起止日期并不一样，夏时制和非夏时制之间的时间差也不一样，而全局变量 daylight 只表示是否是夏时制,不反映各国夏时制之间的差别，实际上，隐含的是美国的标准夏时制。由于我国和美国的夏时制规定不一样，故依靠全局变量 daylight 来解决这个问题是困难的。夏时制主要影响从 unix 格式到 tm 格式之间的转换，还影响到计算两个日期之间的时间差(无论是同一时区还是不同时区)。

全局变量 timezone 记录当地时间与格林尼治标准时间之间的差值，单位为秒。东半球的差值应为负，西半球的差值应为正。在美国，这个差值应设置为 8 * 60 * 60。在中国，这个差值应设置为-8 * 60 * 60。时区差值主要影响从 unix 格式到 tm 格式之间的转换，还影响计算世界两个不同时区的时间差。Turbo C 中提供了一个根据 unix 格式计算当地时间的 localtime 函数，一个根据 unix 格式计算格林尼治标准时间的 gmtime 函数，一个计算当地时间与格林尼治标准时间差值的函数 difftime。

下面列出这些函数的说明，以备查阅。

```
void getdate(struct date *datep);
void gettime(struct time *timep);
long dostounix(struct date *datep,struct time *timep);
void unixtodos(long unixtime,struct date *datep,struct time *timep);
long time(long *unixtime);
struct tm *localtime(conat long *unixtime);
char *asctime(conat struct tm *t);
char *ctime(const long *unixtime);
```

下面给出输出美国当地日期和时间的实例，未考虑夏时制。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
main( )
{
    struct tm          *timeptr;
    time_t              secsnow;
    timezone=8*60*60;
    time(&secsnow);
    timeptr=localtime(&secsnow);
    printf("The date is %d-%d-19%02d\n",((timeptr->tm_mon)+1),
            timeptr->tm_mday,timeptr->tm_year);
    printf("Local time is %02d:%02d:%02d\n",timeptr->tm_hour,
            timeptr->tm_min,timeptr->tm_sec);
}
```

虽然 Turbo C 已经提供了国家信息格式的支持，但它的日期和时间函数却不跟这些国家信息格式发生关系。这个缺陷突出表现在函数 asctime 上，这个函数没有提供有关国家码的参数，故转换出来的 ASCII 中日期时间字符串是美国格式。

各国在时间格式上的差别主要表现在 12/24 小时制和时间分隔符上。下面这个程序(函数)strtime，也是根据结构 tm 转换成 ASCII 串，但它包含用于指定这两个差别的参数，故适用于不同的国家。

```
#include <general.h>
char *strtime(char *timestr,const struct tm *t,
              boolean twentyfourhours,char separator)
{
    sprintf(timestr,"%02d%c%02d%c%02d\0",
            (twentyfourhours ? t->tm_hour : t->tm_hour%12),
            separator,t->tm_min,separator,t->tm_sec);
    return timestr;
}
void main( )
{
    long        unixtime;
    struct tm   *t;
    int         xxiv;
```

```
    char        timestr[9];
    time(&unixtime);
    t = localtime(&unixtime);
    for (xxiv = 0; xxiv <=1; ++xxiv)
        printf("%s\n",strtime(timestr,t,xxiv,':'));
}
```

各国在日期格式上的差别主要表现在年月日的顺序以及它们之间的分隔符上。下面这个程序(函数)strday 也是根据结构 tm 转换成 ASCII 串，但它包括用于指定这两个差别的参数，故适用于不同国家。

```
#include <time.h>
char *dayname[7] = {"Sunday", "Monday", "Tuesday", "Wednesday",
                    "Thursday","friday", "Saturday" };
char *monthname[12] = {"January", "February","March", "April",
                       "May",     "June",    "July",  "August",
                       "September","October", "November","December"};
static int f[3][3] = {2,1,0,1,0,2,0,1,2};
char *strday(char *datestr,const struct tm *t,
                      int format,char separator)
{
    int u[3];
    if (format < 3) {
        u[0] = t->tm_mday;
        u[1] = t->tm_mon + 1;
        u[2] = t->tm_year;
        sprintf(datestr,"%02d%c%02d%c%02d\0",u[f[format][0]],
                separator,u[f[format][1]],
                separator,u[f[format][2]]);
    }
    else
        if (format ==3)
            sprintf(datestr,"%s %d, %d\0",monthname[t->tm_mon],
                    t->tm_mday,(t->tm_year)+1900);
        else
            sprintf(datestr,"%d %s, %d\0",t->tm_mday,
                    monthname[t->tm_mon],(t->tm_year)+1900);
    return dayname[t->tm_wday];
}
```

参数 format 说明年月日的顺序，当取值为 0～2 时，与结构 COUNTRY 的定义相同。

format	意义(示例)
0	88-04-20
1	04-20-80
2	20-04-88
3	April 20, 1988
≥4	20 April 1988

7.10 文件名字符串

Turbo C 中提供了一对互补函数，用来处理文件名。

```
void fnmerge(char *pathname,char *drive,char *dir,
             char *name,char *ext);
int  fnsplit(char *pathname,char *drive,char *dir,
             char *name,char *ext);
```

函数 fnmerge 根据给定的驱动器名(drive)、目录名(dir)、文件名和扩展名(ext)拼出一个全文件名(path)。

函数 fnsplit 则相反，它把完整的全文件名分解成驱动器名、目录名、文件名和扩展名。同时，这个函数还返回一个整数，这个整数中的 5 位分别用来表示全文件名中各相应部分是否存在。为了编程方便，Turbo C 在头文件 dir.h 中定义了相应的 5 个符号常数，以便通过简单的“位与”运算决定相应部分是否存在。

EXTENSION	扩展名
FILENAME	文件名
DIRECTORY	目录和子目录
DRIVE	驱动器
WILDCARDS	符号 * 或?

这两个函数都不检查插入的字符是否有效。这就是说，如果输入是无效的，则产生的输出结果也无效。下面列出有效和无效的例子，先由 fnmerge 合成，合成的结果再由 fnsplit 分解。

		有效	无效
fnmerge	驱动器名	=a	=ax
	目录名	=bcd\efg.hij	=bcd>!
	文件名	=klm	=ef.g
	扩展名	=.n*	=hi
	合成结果	=a:bcd\efg.hij\klm.n*	=a:bcd>!ef.ghi
fnsplit	输入	=a:bcd\efg.hij\klm.n*	=a:bcd>!ef.ghi
	驱动器名	=a:	=a:
	目录名	=bcd\efg.hij\	=bcd>!\
	文件名	=klm	=ef
	扩展名	=.n*	=.ghi
	标志	5 部分都存在	不存在 * 或?

7.11 命令行字符串

一个 C 程序在编译并连接后就可以执行了。在执行时，可以键入参数，这些参数就称为命令行参数，它们可以通过函数 main 来读取。函数 main 的完整格式为:

```
type main(int argc, char *argv[ ], char *env[ ])
```

其中，type 是 main 的返回值类型（详见 7.13 节），env[ ]是环境变量（详见下一节）。argc 和 argv[ ]是处理命令行字符串的。

命令行中的各参数以空格符分隔，argc 说明命令行中共有多少参数，包括命令文件名本身。字符串数组存放着各参数，其中 argv[0]就是命令文件名,argv[argc]是空字符串。

值得注意的是，Turbo C 允许命令行参数中出现星号"*"，它的含义与 DOS 的 copy 命令之类的星号一样。遇到星号后，Turbo C 会把它扩展成单个的文件名。如果星号出现在双引号内，则把它当成普通字符看待。另外，为使星号能起扩展作用，连接时必须把 Turbo C 的 wildargs.obj 连接进去。例如，如果有一个 C 文件名为 args.c，执行如下命令:

```
tcc  args  wildargs.obj
args  c:\TC\INCLUDE\*.H  "*.C"
```

则在运行 args.exe 时，第 1 个参数将被扩展成目录 C:\TC\INCLODE 下的所有扩展名为 H 的文件名，第 2 个参数*.C 将不被扩展。

7.12 环境字符串

DOS 的环境是一个由 DOS 设置的特殊存储区，其中存储的变量可供批文件或用户程序使用。可用环境来存放任一信息，或把文件处理信息从一个程序传递到另一个程序。环境中的每一项都是一个 ASCIIZ 字符串，其形式为:

环境变量名=环境变量的内容(一个单词或一段话)

由 DOS 自动建立的环境变量共有 3 个：PROMPT，PATH 和 COMSPEC。PROMPT 是在用户键入或从批文件中输入 PROMPT 命令时建立的。若用户从未使用过 PROMPT 命令，则也会自动建立一个 PROMPT，其内容为￥n￥g。PATH 也是在用户键入或从批文件中输入 PATH 命令时建立的。若用户从未使用过 PATH 命令，则不建立这个变量。COMSPEC 是在用户键入或从批文件中输入"SET COMSPEC=......"时建立的。若用户从未使用过这个命令，则 DOS 也会自动建立 COMSPEC，其内容为"\COMMAND.COM"。

在 DOS 提示符下执行一个程序时，DOS 会把它环境的一份拷贝传给这个程序。当程序由它的父程序装入执行时，父程序会把它环境的一份拷贝传给这个子程序。Turbo C 是通过 main 函数的第 3 个参数 char *env[ ]来读取它的环境变量的。环境变量是一个接一个的 ASCIIZ 串，直至 1 个空字符串。

Turbo C 还提供了两个函数用来读取和设置环境变量:

```
char *getenv(const char *name);
int  putenv(const char *name);
```

getenv 的参数是指向变量名的字符串，返回的是指向变量值的字符串。putenv 的参数是指向“变量名＝变量值”这样的整个字符串。

下面是读取环境变量的一个例子。

```
#include <general.h>
void main(int n, char *comline[ ], char *environ[ ])
{
    int  i;
    char variable[80];
    char *value;
    printf("Program environment:\n");
    for (i = 0; !strempty(environ[i]);
         printf("%s\n",environ[i++]));
    for (;;) {
         printf("\nVariable: ");
         fflush(stdin);
         gets(variable);
         value = getenv(variable);
         if (value ==NULL)
              printf("Not the name of an environment variable.");
         else
              printf("Value = %s",value);
    }
}
```

7.13 错误级字符串

程序员一般不利用 main 函数的返回值设置它的类型为 void，但 Turbo C 中，main 可以有返回值。在 main 内，返回值可通过 return 语句或 exit 语句实现。这个返回值以 DOS 变量 ERRORLEVEL 的值的形式传给 DOS，在退出这个程序后，可通过 DOS 的 IF ERRORLEVEL 命令取得这个值。

在下面这个例子里，C 语言程序 getkey.c 只含函数 main，而 main 函数又只有一条语句，它返回用户击入的一个键的 ASCII 码。

程序 getkey.dem:

```
#include <stdio.h>
char main( )
{
     return getchar( );
}
```

批文件 getkdem.bat 执行程序 getkey，通过 IF ERRORLEVEL 语句取得返回的 ASCII 码，做相应处理。

```
@echo off
echo  Enter a letter of the alphabet.
getkey
if ERRORLEVEL 91 goto lower
echo  Upper case
goto  end
rem (Z is ASCII 90)
:lower
echo  Lower case
:end
```

如果在批文件执行过程中键入一个大写 Z 键，批文件产生的输出结果可能如下：

```
getkdem
Enter a letter of the alphabet.
Z
Upper case
```

关于 ERRORLEVEL，在第 10 章还将详细讲述。

7.14 字符串处理工具包

7.14.1 13 个工具函数

在下面提供的字符串处理工具包中，又提供了 13 个工具函数。这些函数作为前面已经介绍过的函数的补充，使程序员在处理字符串时更为方便。

```
/* strops.h */
int    pos_str(char *str, char *substr, unsigned int start_index);
int    pos_char(char* str, char ch, int start_index);
void   in_insert(char* str, char* substr, unsigned int index);
void   in_overwrite(char* str, char* substr, unsigned int index);
int    in_delete(char* str, unsigned int index,unsigned int count);
int    in_replace_str(char* str,
                      char* find,
                      char* replace,
                      unsigned int start_index,
                      unsigned int freq);
int    in_pos_mid(char* str, unsigned int first, unsigned int last);
int    in_count_mid(char* str, unsigned int index, unsigned int count);
void   pad_left(char* str, char pad_char, int num_chars);
void   pad_right(char* str, char pad_char, int num_chars);
void   trim_left(char* str, char ch);
void   trim_right(char* str, char ch);
void   trim_ends(char* str, char ch);
```

(1) 函数 pos__str 在字符串 str 中从位置 start__index 开始查找子字符串 substr，返回的是所找子字符串在字符串 str 中的索引号。如果未找到，返回值为-1。

(2) 函数 pos__char 在字符串 str 中从位置 start__index 开始查找字符 ch。如果找到，返回值是相对于字符串 str 的索引号。如果未找到，返回值为-1。

(3) 函数 in__insert 在字符串 str 的 index 处插入一子字符串 substr。如果 index 的值超过了字符串 str 的长度，则这个函数把子字符串直接追加到字符串 str 的后面。

(4) 函数 in__overwrite 从字符串 str 的 index 处开始，用新的子字符串 substr 代替原来的字符串。如果 index 的值超过了字符串 str 的长度，则也相当于追加。显然，这个函数可能会扩大 str 的长度。

(5) 函数 in__delete 从字符串 str 的 index 开始连续删掉 count 个字符。如果 index 大于 str 的长度，则什么也不做。如果 count 太大，则只删到 str 的终止符为止。返回值表示成功(0)还是失败(-1)。

(6) 函数 in__replace__str 从字符串 str 的 strat__index 处开始，用子字符串 replace 替换子字符串 find，连续替换 freg 次。返回值表示成功(0)还是失败(-1)。

(7) 函数 int__pos__mid 抽取字符串 str 中从 first 至 last 之间的子字符串，即削去字符串的头尾部分。返回值表示成功(0)还是失败(-1)。

(8) 函数 in__count__mid 抽取字符串 str 中从 index 处开始的 count 个字符。返回值表示成功(0)还是失败(-1)。

(9) 函数 pad__left 在字符串 str 的左边填充 num__chars 个字符 pad__char。

(10) 函数 pad__right 在字符串 str 的右边填充 num__chars 个字符 pad__char。

(11) 函数 trim__left 删掉字符串中字符 ch 左边所有的字符。

(12) 函数 trim__right 删掉字符串中字符 ch 右边所有的字符。

(13) 函数 trim__ends 删掉字符串中字符 ch 两边所有的字符。

```
/*   strops.c   */
int  pos__str(char *str, char *substr, unsigned int start__index)
{
     char*  ptr;
     ptr = strstr((str+start__index), substr);
     if (ptr !=0 )
          return (ptr - str);
     else
          return -1;
}
int pos__char(char*  str, char ch, int start__index)
{
     int k;
     unsigned int n = strlen(str);
     for (k = start__index; ((*(str+k) != ch) && (k< n)); k++);
     if (k == n)
          k = -1;
     return k;
}
```

```
void in_insert(char * str, char * substr, unsigned int index)
{
     unsigned int m = strlen(substr);
     unsigned int n = strlen(str);
     if (index < n) {
          int i;
          memmove((str+index+m), (str+index),n+1-index);
          memmove((str+index),substr,m);
     }
     else
          memmove((str+n),substr,m+1);
}
void in_overwrite(char * str, char * substr, unsigned int index)
{
     unsigned int m = strlen(substr);
     unsigned int n = strlen(str);
     unsigned int first, shift;
          if (index < n) {
          first = index;
          shift = m;
          if ((index+m) >n)
               shift++;
          }
     else {
          first = n;
          shift = m + 1;
     }
     memmove((str+first),substr,shift);
}
int in_delete(char * str, unsigned int index,unsigned int count)
{
     unsigned int n = strlen(str);
     if (index < 0 ) index = 0;
     if (index < n) {
          if ((index+count-1) >= n)
               str[index] = '\0';
          else
               memmove ((str+index), (str+index+count),n-index-count+1);
          return 0;
     }
     else
          return -1;
}
int in_replace_str(char * str,
                   char * find,
                   char * replace,
                   unsigned int start_index,
                   unsigned int freq)
{
     unsigned int findlen = strlen(find);
     unsigned int repl_strlen=strlen(replace);
```

```
	unsigned int sstrlen = strlen(str);
	int match__pos;
	if (((findlen * sstrlen) = =0) || (start__index >= sstrlen))
		return -1;
	match__pos = pos__str(str, find, 0);
	while (match__pos > -1 && freq > 0) {
		freq--;
		in__delete(str, (unsigned int) match__pos, findlen);
		if(repl__strlen>0)
			in__insert(str, replace, (unsigned int) match__pos);
			match__pos = pos__str(str, find, 0);
	}
	return 0;
}
int in__pos__mid(char * str, unsigned int first, unsigned int last)
{
	unsigned int len = strlen(str);
	unsigned count;
		if (len = = 0 || last < first)
			return -1;
		if (first > len) first = len;
		if (last > len) last = len;
		if (first > last)
			return -1;
		count = last - first + 1;
		if (first > 0) {
			in__delete(str, 0, first);
			last -= first;
			len -= first;
		}
	if (last < len)
		in__delete(str, (last+1), (len-last));
	return 0;
}
int in__count__mid(char * str, unsigned int index, unsigned int count)
{
	unsigned int len = strlen(str);
	if (len = =0)
		return -1;
	if (index > len)
		index = len;
	if ((index + count) > len)
		count = len - index + 1;
	if (index > 0) {
		in__delete(str, 0, index);
		len -= index;
	}
	if (count < len)
		in__delete(str, count, (len-count+1));
	return 0;
}
```

```
void pad__left(char * str, char pad__char, int num__chars)
{
    char pad__str[41];
    memset(pad__str, pad__char, num__chars);
    pad__str[num__chars] = '\0';
    in__insert(str, pad__str, 0);
}
void pad__right(char * str, char pad__char, int num__chars)
{
    char pad__str[41];
    memset(pad__str, pad__char, num__chars);
    pad__str[num__chars] = '\0';
    in__insert(str, pad__str, strlen(str)+1);
}
void trim__left(char * str, char ch)
{
    int i, count = 0;
    for (i = 0; ((* (str+i) != '\0') && (* (str+i) == ch)); i++)
        count++;
    in__delete(str, 0, count);
}
void trim__right(char * str, char ch)
{
    int i, count = 0;
    unsigned int len=strlen(str);
    for (i=len-1;((i>=0) && (* (str+i) == ch)); i--)
        count++;
    str[len-count]='\0';
}
void trim__ends(char * str, char ch)
{
    trim__left(str, ch);
    trim__right(str, ch);
}
```

7.14.2 工具包应用举例：PASCAL 程序翻译为 C 程序

下面将介绍一个翻译程序 translan.c。这个程序运行后，先后要求输入 3 个文件名。一个是被翻译的源文件，另一个是说明如何进行翻译的说明文件，最后一个是翻译后生成的输出文件。说明文件也是一个文本文件，它是由一行一行的文本组成的，每一行是一条命令。目前的这个 translan.c 只支持 3 条命令。如果要加强翻译效果，可以修改 translan.c，使它支持更多种类的命令。该程序模块化程度高，添加功能比较容易。

目前支持的 3 条命令是：

(1) ″D|<text>|。这条 D 命令删除文本 text，区分大小写。

(2) ″K|<text>|。这条 K 命令删除文本 text，不区分大小写。

(3) ″R| <old__text>| <new__text>|。这条 R 命令以文本 new__text 代替文本 old__text。

翻译程序 translan.c:

```
/ * traslan.c * /
#include <stdio.h>
#include "conio.h"
#include "string.h"
#include "strops.h"
#define TITLE "MULTI-FILE TEXT MANIPULATION"
#define VERSION "Version 1.0"
#define DEFAULT_SCRIPT_FILE "PAS2C.SRC"
#define MAX_SCRIPT_LINES 45
typedef char STRING[81];
typedef unsigned int word;
typedef unsigned char byte;
enum booleans{FALSE,TRUE};
typedef enum booleans boolean;
struct script_cmd {
            char op_char;
            STRING dtext;
};
main( )
{
    FILE *infile_var, *outfile_var, *script_var;
    STRING infile, outfile, script;
    STRING line, genstr;
    struct script_cmd script_lines[MAX_SCRIPT_LINES];
    word num_script, i;
    char go_on, quit_ch;
    boolean ok, use_same_script, have_read_script;
    strcpy(script, DEFAULT_SCRIPT_FILE);
    have_read_script = FALSE;
    do {
        clrscr( );
        center(TITLE,1);
        center(VERSION,2);
        printf("\n\n\n");;
        do {
            printf("Enter source filename ->");
            gets(infile); printf("\n");
            infile_var = fopen(infile, "rt");
            if (infile_var == NULL)       {
                printf("Cannot open file %s\n\n",infile);
                printf("Exit ? (Y / N)");
                quit_ch = getche( );
                if ((quit_ch == 'Y') || (quit_ch == 'y'))
                    exit(0);
                printf("\n");
            }
        } while (infile_var == NULL);
        do {
            printf("Enter destination filename -> ");
            gets(outfile); printf("\n");
```

```
        outfile_var=fopen(outfile,"wt");
        if (outfile_var == NULL) {
            printf("Cannot open file %s\n\n",outfile);
            printf("Exit ? (Y/N)");
            quit_ch = getche( );
            if ((quit_ch == 'Y') || (quit_ch == 'y'))
                exit(0);
            printf("\n");
        }
    } while (outfile_var == NULL);
    do {
        printf("The default script filename is %s\n", script);
        printf("Enter script filename ");
        printf("press [Enter] for default ->");
        gets(genstr); printf("\n");
        if (strlen(genstr) !=0) {
            strcpy(script,genstr);
            use_same_script = FALSE;
        }
        else
            use_same_script = TRUE;
        script_var = fopen(script, "rt");
        if (script_var == NULL)        {
            printf("Cannot open file %s\n\n",outfile);
            printf("Exit ? (Y/N)");
            quit_ch = getche( );
            if ((quit_ch == 'Y') || (quit_ch == 'y'))
                exit(0);
            printf("\n");
        }
    } while (script_var == NULL);
    if (have_read_script == FALSE ||
        use_same_script == FALSE) {
        printf("Reading and processing the script file ...");
        num_script = 0;
        while ((! feof(script_var)) &&
            (num_script < (MAX_SCRIPT_LINES-1))) {
        fgets(genstr,80,script_var);
        get_script(genstr, &script_lines[num_script], &ok);
        if (ok == TRUE) num_script++;
        }
        have_read_script = TRUE;
        printf("\n");
    }
    fclose(script_var);
    if (num_script > 0) {
        while (!feof(infile_var)) {
            fgets(line,80,infile_var);
            if (strlen(line) > 0) {
                for (i = 0; i <num_script; i++)
                    switch (script_lines[i].op_char) {
```

```
                    case 'D':
                      delstr(line, script_lines[i].dtext);
                      break;
                    case 'K':
                      kilstr(line, script_lines[i].dtext);
                      break;
                    case 'R':
                      translate(line,script_lines[i].dtext);
                      break;
                  }
              }
              printf("%s", line);
              fprintf(outfile_var, "%s",line);
          }
          fclose(infile_var);
          fclose(outfile_var);
      }
      printf("\n");
      printf("Want to process more files? (Y/N) ");
      go_on = getche( );
  } while (!( go_on == 'Y' || go_on == 'y'));
}
int center(char *strng, byte line_num)
{
    gotoxy(40 - strlen(strng) /2, line_num);
    printf("%s",strng);
}
int get_script(char * strng,
               struct script_cmd *script_line,
               boolean *good_line)
{
    char ch;
    int  index1, index2,index3;
    STRING str0;
    ch =  *strng;
    if (ch >='a' && ch <='z') ch += 'A' - 'a';
    *good_line = FALSE;
    if (ch == 'D' || ch == 'K' || ch =='R') {
        script_line->op_char = ch;
        index1 = pos_char(strng, '|', 0);
        if (index1 > -1)  {
            index2 = pos_char(strng, '|', index1+1);
            if (index2 > -1)  {
                strcpy(script_line->dtext,strng);
                in_pos_mid(script_line->dtext, index1+1,index2-1);
                *good_line = TRUE;
                if (ch == 'R') {
                    index3 = pos_char(strng, '|', index2+1);
                    if (index3 > -1) {
                        strcpy(str0, strng);
                        strcat(script_line->dtext,"|");
```

```
                    in_pos_mid(str0,index2+1,index3-1);
                    strcat(script_line->dtext, str0);
                }
                else
                    * good_line = FALSE;
            }
        }
    }
  }
}
int delstr(char * strng, char * substr)
{
    int i;
    unsigned int len = strlen(substr);
    i = pos_str(strng, substr, 0);
    while (i > -1) {
        in_delete(strng,i,len);
        i=pos_str(strng,substr,0);
    }
}
int kilstr(char * strng, char * substr)
{
    int i;
    unsigned int len = strlen(substr);
    STRING strcopy;
    strcpy(strcopy, strng);
    strupr(strcopy);
    strupr(substr);
    i = pos_str(strcopy, substr, 0);
    while (i > -1) {
        in_delete(strng, i, len);
        in_delete(strcopy, i, len);
        i = pos_str(strcopy, substr, 0);
    }
}
int translate(char * strng, char * pattern)
{
    unsigned int findlen, replen;
    int i;
    STRING find, replace;
    i = pos_char(pattern,'|',0);
    strcpy(find,pattern);
    in_delete(find,i,strlen(pattern)-i);
    strcpy(replace,pattern);
    in_delete(replace,0,i+1);
    findlen = strlen(find);
    replen = strlen(replace);
    i = pos_str(strng, find, 0);
    while (i > -1) {
        in_delete(strng, i, findlen);
        in_insert(strng, replace, i);
```

```
            i = pos_str(strng, find, i+replen+1);
        }
}
```

这个程序的说明文件 PAS2C.SRC 如下:

```
R    |Turbo Pascal|C|
R    |(*|/*|
R    |{|/*|
R    |*)|*/|
R    |}|*/|
R    |BEGIN|{|
R    |END|}|
R    |FUNCTION|void|
R    |PROCEDURE|void|
D    |THEN|
D    |DO|
R    |REPEAT|do|
R    |clrscr|clrscr( )|
R    |clreol|clreol( )|
R    |clrScr|clrscr( )|
R    |FOR|for|
R    |SQR(|∧∧(|
R    | = | == |
R    |:=|=|
R    |WRITELN(|printf("\n|
R    |WRITE(|printf(|
R    |READLN(|scanf("%|
R    |READ(|scanf("%|
R    |'|"|
R    |WRITELN|printf("\n");|
R    |UNTIL|while not|
R    |<>|!=|
R    |NOT|!|
R    |AND|&&|
R    |WORD|word|
R    | OR | || |
R    |IF|if|
R    |ELSE|else|
R    |CASE|switch|
R    |WHILE|while|
R    |TO|; <=|
R    |Length(|strlen(|
R    |};|}|
```

假设有如下这样一个 PASCAL 源程序:

```
PROGRAM SIEVE_TEST;
CONST size = 7000;
    MAX_ITER = 100;
```

```
VAR I, prime, k, count, iter : INTEGER;
    flags : ARRAY [0..size] OF BOOLEAN;
BEGIN
    WRITELN('START ',MAX_ITER,' iterATIONS');
    FOR iter := 1 TO MAX_ITER DO BEGIN
        count := 0;
        FOR i := 0  TO size DO flags[i] := TRUE;
        FOR i := 0  TO size DO
            IF flags[i] THEN BEGIN
                prime := k + k + 3;
                k := k + prime;
                WHILE k <= size DO BEGIN
                    flags[i] := FALSE;
                    k := k + prime
                END;
                count := count + 1
            END;
    END;
    WRITELN(∧G,count,' primeS');
END.
```

经 translan 程序翻译以后，所得到的 C 源程序将如下所示：

```
PROGRAM SIEVE_TEST;
CONST size == 7000;
    MAX_ITER == 100;
VAR I, prime, k, count, iter : INTEGER;
    flags : ARRAY [0..size] OF BOOLEAN;
{
    printf("\n"START ",MAX_ITER," iterATIONS");
    for iter = 1 ; <= MAX_ITER  {
        count = 0;
        for i = 0  ; <= size  flags[i] = TRUE;
        for i = 0  ; <= size
            if flags[i]  {
                prime = k + k + 3;
                k = k + prime;
                while k <= size  {
                    flags[i] = FALSE;
                    k = k + prime
                }
                count = count + 1
            }
    }
    printf("\n∧G,count," primeS");
}
* /
```

当然，这个 C 源程序不完全符合 C 的语法，还需要由人工去去进行修改。修改后真正的 C 源程序如下所示，从中也可看出这种机器翻译能起多大作用，还需要多少人工的

干预。

```
/ * Sieve test program * /
#include <stdio.h>
#define  size 7001
#define  MAX_ITER 100
#define  TRUE 1
#define  FALSE 0
main( )
{
    int i, prime, k,count, iter;
    unsigned char flags[size];
    printf("START %d iterATIONS\n",MAX_ITER);
    for (iter = 0 ; iter < MAX_ITER; iter++) {
        count = 0;
        for (i = 0,k = 0; k < size; k++)
            flags[i] = TRUE;
        for (i = 0,k = 0; k < size; k++) {
            if (flags[i] == 1) {
                prime = k + k + 3;
                k = k + prime;
                while (k <= size) {
                    flags[i] = FALSE;
                    k += prime;
                }
                count++;
            }
        }
    }
    printf("\n\007%d primeS\n",count);
}
```

第 8 章　动态通用串处理

动态通用串与一般字符串有两点重要差别：一是“动态”，二是“通用串”。所谓动态，就是串的长度在运行时可以变化。所谓通用串就是串中的每一个基本元素不再只是 1 个字符，而可以是任何一种数据类型。例如，1 个整数，1 个浮点数，等等。由此可以看出，动态通用串基本上不是为了处理字符而引伸的，而是为了处理一些更复杂的数据类型。无论哪一种 C 编译都没有直接提供处理动态通用串的函数，本章将提供一些补充的工具函数并用这些工具函数来实现一个动态通用串应用的典型例子：存储任意增长的多边形的顶点坐标系列。

8.1 动态字符串

C 语言中的字符串是 ASCIIZ 串，即用 ASCII 码 0 来表示 1 个字符串的结尾。这种表示法带来 3 个问题。第 1，不容易计算字符串的长度，这不符合“C 语言是高效语言”的原则。第 2，在字符串内不能出现 ASCII 码 0，这使得不能用字符串来存储任意的二进制数据。第 3，在第 1 次为字符串分配空间后，如果经过多次运算，就不知道字符串内究竟还剩多少可用空间。例如在拷贝时，可能使目标字符串太长，冲毁原来在这个字符串后面的其它内容。

PASCAL 语言采用了另一种表达字符串的办法。每个字符串的最大长度是在编译时指定的，运行时在字符串的最前面用 1 个字节来说明字符串的实际长度，(不能超过 256)。如果用 C 的结构来描述 PASCAL 的字符串，其定义应该如下：

```
struct turbo__pascal__string {
      unsigned char currlen;
      undigned char data[MAXSIZE];
}
```

字段 currlen 是字符串的实际长度，data 是存储字符串的数组。

BASIC 语言的字符串功能比 C 语言强，前者几乎允许字符串的长度任意变化，只要不超过 64K 字节。如果用 C 的结构来描述 BASIC 语言的字符串，则其定义如下：

```
struct basic__string {
      unsigned int dimlen;
      undigned int currlen;
      unsigned char data[CURRDIMSIZE];
}
```

字段 dimlen 是字符串的最大长度，currlen 是字符串的实际长度，data 是存储字符串

的数组。

如果希望改善 C 语言中字符串的处理功能，则应该模仿 BASIC 语言来建立一个结构:

```
typedef struct dynamic__str__struct {
      unsigned int  dimlen;
      unsigned int  currlen;
      unsigned char  * data;
}dynamic__str;
```

这个结构与 BASIC 语言字符串结构的差别在于: 这里的字段 data 不是 1 个字符数组，而是 1 个字符串指针，这样可以把字符串数据和字符串结构本身隔离开，从而可以更方便地为字符串数据分配存储空间。

为了在程序中使用这样一个结构，应在定义了 dynamic__str 型的结构变量后，接着为字符串数据分配空间，把分配的字节数存放在字段 dimlen 中。字符串的当前长度 currlen 初始化为 0，以后则随所存放字符串长度的变化而变化。例如:

```
#include  <alloc.h>
#include  <string.h>
dynamic__str mystr;
char msg[ ]="hello";
main( ) {
      mystr.data=malloc(25);
      mystr.dimlen=25;
      mystr.currlen=0;
      memcpy(mystr.data,msg,sizeof(msg));
      mystr.currlen+=sizeof(msg);
}
```

如果原来为字符串数据分配的空间不够用，则可以通过 Turbo C 的 realloc 函数来扩大字符串的长度. realloc 可以改变一个已分配块的大小。如果变大，则该块原来存放的数据不会被破坏，但整个分配块可能移到一个新的位置。如果变小，则该块原来已存放的数据可能被截短，但位置不会变化。

因为重新分配内存可能引起数据移动，故指向这块数据的指针可能改变其值，即在调用了 realloc 函数后就不能保证这个指针不发生变化。例如，下面这些语句

```
char  *p,  *q;
p=malloc(200);
q=p;
p=realloc(300);
```

在执行完后，指针 q 就可能不再指向有意义的数据。但不要忘记，指针 p 仍是可用的。这就给我们一个提示: 把指向字符串数据的指针存放在一个结构内，始终通过这个指针来存取数据，来分配和重分配空间，就不会出错了。结构 dynamic__str 正是这样做的。例如:

```
#include <alloc.h>
void myfunc(dynamic__str * s);
main( )
{
    dynamic__str mystr;
    mystr.data=malloc(200);
    myfunc(&mystr);
    if(mystr.data[1]=='Y')
    /* then do something useful. */
}
void myfunc(dynamic__str * s)
{
    s->data=realloc(s->data,600);
}
```

即使在 myfunc 函数内通过 realloc 使数据指针 s->data 发生变化，由于在主函数内仍然使用这个指针(尽管名字为 mystr.data)，故仍可以进行正确存取。

8.2 动态通用串

从上面可以看出，有了结构 dynamic__str 后，字符串的处理可以做得比 BASIC 好，进一步利用 C 语言的特性，还可以使串的处理功能更强。C 语言中允许数据类型的强制转换，其中用得最多的就是指针所指数据的类型的强制转换。例如:

```
char *p;
int  *i;
/* ... */
i=(int *)p;
```

指针 p 本来是指向字符的，在强制转换后变成了指向整数，尽管 p 所指存储区的内容实际上没有变化。上一节介绍了用动态字符串结构处理字符数据，其实还可以把这个结构用来处理其它类型的数据，如整数、浮点数等。这只要把结构略作修改即可。

```
typedef struct vstr__struct {
    unsigned int dimlen;
    unsigned int currlen;
    int esize;
    int inc;
    void * data;
}vstr;
```

与结构 dynamic__str 相比，这里的 vstr 结构多了两个字段，一个是 esize，它说明串内每个基本元素占多少字节。另一个是 inc，它只用来说明，如果需要扩大串的长度，应该一次至少扩大多少元素的空间，以免经常调用 realloc 函数。例如，下面这个 init__vstr 函数:

```
#include <alloc.h>
void init__vstr(vstr s)
{
      s.dimlen    =25;
      s.currlen   =0;
      s.esize     =sizeof(int);
      s.inc       =10;
      s.data      =calloc(25,sizeof(int));
}
```

就建立了一个可以容纳 25 个整数的整数串 vstr，每次增加 10 个整数。这个函数中用 sizeof 确定每个元素所占的字节数，这样便于移植。

结构 vstr 与结构 dynamic__str 另一不同之处，就是它的指针 data 不再指向字符，而是指向 void，即所指数据类型是不确定的。使用时根据实际需要进行强制转换。在下面这个例子里，先往整数串中存储了 0～9 这样 10 个整数，然后再依次把它们读出来。

```
#include <stdio.h>
#include <alloc.h>
typedef struct vstr__struct {
      unsigned int dimlen;
      unsigned int currlen;
      int esize;
      int inc;
      void * data;
}vstr;
main( )
{
      vstr s;
      int i, * ip;
      s.dimlen    =25;
      s.currlen   =0;
      s.esize     =sizeof(int);
      s.inc       =10;
      s.data      =calloc(25,sizeof(int));
      for ( i=0; i<10; i++) ((int * )(s.data))[i]=i;
      s.currlen  =10;
      for ( i=0, ip=(int  * )(s.data); i<10; i++,ip++)
            printf("%d\n", * ip);
}
```

在该程序中，存数是用数组方法，即((int *)(s.data))[i]，先把 s.data 强制转换为指向整数的指针，然后再通过下标[i]来取某个元素。取数是用指针，先把 s.data 强制转换为指向整数的指针并赋给指针变量 ip，即 ip=(int *)(s.data),然后再通过 * ip 取出相应的整数值。但这种取数方法违背了前面讲过的原则，如果在 ip=(int *)(s.data)和 * ip 之间某个程序通过 realloc 修改了指针 s.data，则可能失败。因此，最好还是用第 1 种方法。为了书写方便，可以定义如下的宏：

```
#define int__array ((int * ) (s.data))
```

这样，就可以通过 int__array[i]来存取串 s 中第 i 个元素。但这个宏把结构变量的名字固定为 s。如果把宏修改成如下形式，结构变量的名字就不会被固定死。

```
#define int__array(x)((int * )(x.data)
```

这样，就可以通过 int__array(abc)[i]来存取串 abc 中第 i 个元素。

8.3 动态通用串工具包

8.3.1 8 个工具函数

由于动态通用串是建立在结构 vstr 的基础上，无论哪一种 C 编译，都没有提供这样一种数据结构，因此也不提供这样的支持函数。在本工具包中，提供了一些这样的支持函数，所有函数都在成功时返回 1，失败时返回 0。

这些函数所处理的动态通用串结构与前面讲过的结构稍有区别，即在最前面多了一个字段 marker。

```
typedef struct vstr__struct {
      char marker[5];
      unsigned int dimlen;
      unsigned int currlen;
      int esize;
      int inc;
      void * data;
}vstr;
```

marker 占 5 个字节，作为初始化标志。当第 1 次为动态通用串分配空间时，这个标志被设置为“(@)%”。当 vstr 结构作为参数传给别的工具函数时，这些函数会对这个标志进行检查，看是否为“(@)%”。如果不对，则说明 vstr 未被正确初始化，从而给出一个错误信息，停止程序执行。这种安全性检查主要是便于程序调试,调试完毕可以去掉它，但保留它也不会耗费多大开销。

工具包内的 8 个函数如下：

```
extern  int vstr__inited(vstr * v);
extern  int dimvstr(vstr * v,unsigned int dimlen,int esize,int inc);
extern  int redimvstr(vstr * y, unsigned int newdimlen);
extern  int clrvstr(vstr * v,unsigned int dimlen,int esize,int inc);
extern  int delvstr(vstr * v);
extern  int copyvstr(vstr * t, vstr * s);
extern  int vstrdel(vstr * v,int p,int n);
extern  int vstrins (vstr * v, int p, void * s, int n);
```

(1) 函数 vstr__inited 检查动态通用串 v 是否已被初始化。如果没有，则程序流产。

(2) 函数 dimvstr 为动态通用串 v 分配 dimlen 个元素的空间并进行初始化，每个元素的字节数为 esize。以后动态增长时，增量为 inc 个元素。如果 inc=0，则意味着该动态通

用串的长度不允许变化。

(3) 函数 redimstr 使动态通用串 v 的长度变为 newdimlen 个元素。

(4) 函数 clrvstr 与函数 dimvstr 的参数相同，它首先检查指针 v__data 是否为空。如果是，则认为还没有为动态通用串分配空间，于是调用函数 dimvstr，按所给定的参数为动态通用串分配并初始化数据空间。如果不是，则认为已分配了数据空间，置动态通用串的当前长度为 0，意味着它是一个空串，以便重新使用这个串。应该注意，这里空串的概念是串的当前长度为 0，而不是串的数据区指针为 NULL。

有了 clrstr 函数后，就可以在程序的一开始先定义一些 vstr 型的变量，并设置它们的数据区指针为空指针 NULL。以后每当为串分配空间或重新使用它时，就可以调用 clrvstr 去完成。当程序变得越来越大，跟踪已分配的内存变得很困难时，这个技术大大方便程序的调试。然而，不要忘记在程序一开始设置串的数据区指针为 NULL。也就是说，动态通用串的初始值应该如下所示：

```
vstr pstr={
      {0,0,0,0,0},
      0,
      0,
      0,
      0,
      NULL
  };
```

本工具包的头文件 vstr.h 中定义了宏 NULLVSTR：

```
#define NULLVSTR {{0,0,0,0,0},0,0,0,0,NULL}
```

这个宏可以在程序的一开头通过语句

```
#include <stdlib.h>
#include "vstr.h"
vstr pstr=NULLVSTR;
```

使动态通用串 pstr 在编译时就被置为所希望的初始值。应该注意，NULL 是在头文件 stdlib.h(或 stddef.h，mem.h，allo.h)中定义的。为了使用宏 NULLVSTR，必须包括这些文件之一。

(5) 函数 delvstr 释放动态通用串 v 所占用的数据空间，并使串成为空串。

(6) 函数 copyvstr 拷贝源通用串 s 到目标通用串 t。如果目标通用串原来未被分配空间或分配的空间不够用，则函数会自动为它分配必要的空间。

(7) 函数 vstrdel 在动态通用串 v 中从元素 p 开始连续删掉 n 个元素。如果 p 超出范围或 n<0，则什么也不做。

(8) 函数 vstrins 把由指针 s 指向的串中的 n 个元素插入到动态通用串 v 的元素 p 处。如果 p 超过了串 v 的当前长度，则相当于连接两个串。如果插入过程需要扩大存储空间，则函数会自动增加空间，增加的数目等于 n 和 v->inc 中的较大者。应该注意，指针 s 所指数据的类型是 void，这就是说可以把任何类型的数据插到串 v 中去。但 n 指的是元

素个数，不是字节数，要插入多少字节是根据串 v 中定义的元素大小来计算的。一般要求两个数据类型相同。

```
/*  vstr.h  */
#define vstrcat(v,e) vstrins(v,(v)->currlen,e,1)
#define NULLVSTR {{0,0,0,0,0},0,0,0,0,NULL}
typedef struct vatr_struct {
        char            marker[5];
        unsigned int dimlen;
        unsigned int currlen;
        int             esize;
        int             inc;
        void            *data;
}vstr;
extern  int vstr_inited(vstr *v);
extern  int dimvstr(vstr *v,unsigned int dimlen,int esize,int inc);
extern  int redimvstr(vstr *y, unsigned int newdimlen);
extern  int clrvstr(vstr *v,unsigned int dimlen,int esize,int inc);
extern  int delvstr(vstr *v);
extern  int copyvstr(vstr *t, vstr *s);
extern  int vstrdel(vstr *v,int p,int n);
extern  int vstrins (vstr *v, int p, void *s, int n);
/*****************************************************/
/*  vstr.c  */
#include  <stddef.h>
#include  <stdio.h>
#include  <alloc.h>
#include  <conio.h>
#include  <string.h>
#include  <process.h>
#include  "vstr.h"
#define   MAX(a,b)  ((a) > (b) ? (a) : (b))
#define   MIN(a,b)  ((a) < (b) ? (a) : (b))
static char vinitflag[ ] = "(@)%";

int vstr_inited(vstr *v)
{
     if (strcmp(v->marker,vinitflag))  {
          printf("Serious error: vstr not init'ed\n"
                 "Prees return ..");
          getch( );
          exit(1);
     }
     return 1;
}

int dimvstr(vstr *v, unsigned int dimlen, int esize, int inc)
{
     if ((v->data = calloc(dimlen, esize)) != NULL)  {
          v->dimlen = dimlen;
```

```
            v->currlen = 0;
            v->inc = inc;
            v->esize = esize;
            strcpy(v->marker, vinitflag);
            return 1;
        }
        else return 0;
}

int  redimvstr(vstr *v, unsigned int newdimlen)
{
        if (vstr_inited(v)) {
            if ((v->inc) && (v->data =
                    realloc(v->data,newdimlen * v->esize)) != NULL) {
                v->dimlen = newdimlen;
                v->currlen = MIN(v->currlen, newdimlen);
                return 1;
            }
        }
        return 0;
}

int clrvstr(vstr *v, unsigned int dimlen, int esize, int inc)
{
        if (v->data == NULL) {
            return dimvstr(v, dimlen, esize, inc);
        }
        else {
            if (vstr_inited(v)) {
                v->currlen = 0;
                return 1;
            }
            return 0;
        }
}

int delvstr(vstr *v)
{
        if (v->data != NULL) {
            if (vstr_inited(v)) {
                free(v->data);
                v->data = NULL;
                v->currlen = 0;
                return 1;
            }
            else return 0;
        }
        return 1;
}

int copyvstr(vstr *t, vstr *s)
```

```
{
    if (vstr_inited(s)) {
        if (clrvstr(t, s->dimlen, s->esize, s->inc)) {
            memmove(t->data, s->data, s->currlen * s->esize);
            t->currlen = s->currlen;
            return 1;
        }
    }
    return 0;
}

int vstrdel(vstr *v,int p,int n)
{
    int k;
    char *t;
    if (vstr_inited(v)) {
        if ((p>=0) && (n>0) && (p < v->currlen)) {
            if ( (k = p+n) >= v->currlen ) {
                v->currlen = p;
            }
            else {
                t = (char *)v->data;
                memmove(t+p * v->esize, t+k * v->esize,
                        (v->currlen-k) * v->esize);
                v->currlen -=n;
            }
            return 1;
        }

    }
    return 0;
}

int vstrins (vstr *v, int p, void *s, int n)
{
    char *t;
    int addsize;
    if (vstr_inited(v) && p >=0) {
        addsize = v->currlen + n - v->dimlen;
        if (addsize >0) {
            if (v->inc != 0) {
                addsize =MAX(addsize, v->inc);
                if(!redimvstr(v,v->dimlen+addsize)) return 0;
            }
            else {
                if (p < v->currlen) {
                    t = (char *)v->data;
                    if (n > (v->currlen-p)) {
                        n = v->currlen - p;
                    }
                    else {
```

```
                    memmove(t+(p+n) * v->esize,t+p * v->esize,
                            v->currlen-p-n);
                }
                memmove(t+p * v->esize, s, n * v->esize);
                return 1;
            }
            return 0;
        }
    }
    t = (char *)v->data;
    if (p >= v->currlen) {
        p = v->currlen;
    }
    else {
        memmove (t+(p+n) * v->esize, t+p * v->esize,
                (v->currlen-p) * v->esize);
    }
    memmove(t+p * v->esize, s, n * v->esize);
    v->currlen += n;
    return 1;
  }
  else return 0;
}
```

8.3.2 工具包应用举例:多边形表示法

下面这个例子说明如何用动态通用串来存储多边形各顶点的坐标。这些坐标是通过鼠标输入的，每揿一下(释放时起作用) 鼠标的左按钮，就输入一个点，同时在上一个点和这个点之间连一条线。当最后揿一下鼠标的右按钮时(也是释放时起作用)，就把第 1 个点和最后一个点连接起来。同时调用 Turbo C 的 fillpoly 函数用红色的交叉斜线填充这个多边形。

为弄清程序的工作过程，必须先看一下 fillpoly 函数的说明:

```
void far fillpoly(int numpoints,int far *polypoints);
```

参数 numpoints 是顶点数，参数 polypoints 指向的整数数组应含(numpoints * 2)个整数，表示各个顶点的坐标，x 在前，y 在后。为了进行填充，要求最后一个点的坐标等于第 1 个点的坐标。填充后，无论释放鼠标的哪个按钮，都会使显示返回到文本方式，打印出所有顶点的坐标。

程序中用到了宏 vstrcat，它的定义如下所示:

```
#define vstrcat(v,e) vstrins(v,(v)->currlen,e,1)
```

这个宏只允许在串的末尾添加 1 个元素。

下面就是多边形表示法的程序清单。

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

```
#include "mouse.h"
#include "vstr.h"
#define  POINT ((point  * )(v.data))
typedef  struct {
     int x,y;
}point;
void main( )
{
     int   gd=DETECT,gm=0;
     unsigned int k;
     vstr v=NULLVSTR;
     point p;
     initgraph(&gd,&gm,"");
     init_mouse(MOUSE_NEEDED,gd,gm);
     clrvstr(&v,30,sizeof(point),5);
     do {
          while(!(k=mouse_trigger(0)));
          if(k= =LEFT_MOUSE_REL) {
               p.x=mouse_grph_x;
               p.y=mouse_grph_y;
               vstrcat(&v,&p);
               / *  if have more than one point,draw a line  * /
               if(v.currlen>1) {
                    mouse_off(1);
                    line(POINT[v.currlen-1].x,POINT[v.currlen-2].y,
                         POINT[v.currlen-2].x,POINT[v.currlen-2].y);
                    mouse_on(1);
               }
               else {
                    mouse_off(1);
                    putpixel(p.x,p.y,WHITE);
                    mouse_on(1);
               }
          }
     }while(k!=RIGHT_MOUSE_REL);
     vstrcat(&v,v.data);
     setfillstyle(8,4);
     mouse_off(1);
     fillpoly(v.currlen,(int far  * )(v.data));
     mouse_on(1);
     while(!(k=mouse_trigger(0)));
     mouse_reset( );
     closegraph( );
     for (k=0; k<v.currlen; k++) {
          printf("%d  %d\n",POINT[k].x,POINT[k].y);
     }
}
```

8.4 通用串的排序与查找

Turbo C 2.0 版提供了 1 个排序函数和 3 个查找函数。

```
void qsort(void *base,size_t nelem,size_t width,
           int (*fcmp)(const void *,const void *));
void *bsearch(void *key,void *base,size_t nelem,size_t width,
              int (*fcmp)(const void *,const void *));
void *lsearch(void *key,void *base,size_t nelem,size_t width,
              int (*fcmp)(const void *,const void *));
void *lfind  (void *key,void *base,size_t nelem,size_t width,
              int (*fcmp)(const void *,const void *));
```

在这几个函数中，参数 base 都是指向欲进行处理的数组(串)，nelem 是数组中元素的个数，width 是每个元素占的字节数，fcmp 是进行比较的函数的指针，key 指向欲进行查找的数组元素。

进行比较的函数是对两个数组元素进行比较，它应按下列规则返回 1 个整数:

元素 1 小于元素 2	返回值小于 0
元素 1 等于元素 2	返回值等于 0
元素 1 大于元素 2	返回值大于 0

函数 qsort 采用 quicksort 算法对数组进行排序。排序后，值小的在前，值大的在后。值大小是由比较函数（而不是由 qsort 函数）决定的。因此，若调用 strcmp 函数对字符串进行比较，则最后排出的结果为升序。反之，若有一个字符串比较函数认为 abcd 大于 abce，则最后排出的结果为降序。还应注意，不管数组元素是什么数据类型，只要比较函数能对它们比较，就可以应用这些排序和查找函数。如果数组元素是有值的，如整数、浮点数和字符串等，则大小的规则是公认的。如果数组元素是无值的，如结构，则比较函数往往是按结构中某个有值字段来排序。

函数 bsearch 是按二分法在数组中查找 1 个元素，要求数组事先是已排好序的。如果找到，则返回指向这个元素的指针，否则返回空指针 NULL。

函数 lfind 是按线性法在数组中查找 1 个元素，不要求数组事先排好序。如果找到，则返回指向这个元素的指针，否则返回空指针 NULL。

函数 lsearch 与函数 lfind 类似，只是在没有找到的情况下，把元素 key 添加到数组末尾。

下面是对字符串数组进行排序和查找的例子:

```
#include <stdio.h>
#include <stdlib.h>
#include "string.h"
main( )
{
    char str[10][11]={"Robert","Bobbi","Keith","James","David",
                      "Kim","Thomas","Paul","Peter","John" };
```

```
    char      name[11] = "Kim";
    int       i,n=10;
    unsigned found;
    qsort(str, n, 11, strcmp);
    found=((unsigned)lfind(name,str,(size_t *)n,11,strcmp) -
          (unsigned)str) / sizeof(name);
    printf("found %s in index %u\n\n",name,found);
    for ( i = 0; i <n; i++)
        printf("%d %s\n", i, str[i]);
}
```

下面是对结构数组进行排序和查找的例子。结构由 name(名字)、age(年龄)、wt（体重）3 个字段组成且都有值，但它们的大小是按名字排定的。

```
#include <stdio.h>
#include <stdlib.h>
#include "string.h"
struct mail_rec {
    char name[31];
    unsigned int age;
    double wt;
};
typedef struct mail_rec mail;
int struct0_cmp(const void *e1,const void *e2)
{
    return strcmp(((mail *)e1)->name, ((mail *)e2)->name);
}
main( )
{
    mail person[5] = { { "Namir", 34, 180.0 },
                       { "Bobbi", 31, 165.0 },
                       { "Keith", 29, 155.0 },
                       { "James", 35, 190.0 },
                       { "David", 41, 190.0 } };
    mail who = { "Keith", 29, 155.0 };
    unsigned i, n = 5;
    unsigned found;
    qsort(person, n, sizeof(mail), struct0_cmp);
    found=((unsigned)bsearch(&who,person,n,sizeof(mail),struct0_cmp)
          -(unsigned)person) / sizeof(mail);
    printf("Found %s in index %d\n\n", who.name, found);
    for ( i = 0; i < n; i++)
        printf("%d %s is %2d years old and weighs %lg bounds \n",
               i, person[i].name, person[i].age, person[i].wt);
}
```

不同的排序和查找算法各有其优缺点。如 shell 算法比 quicksort 算法速度可能慢些，但没有递归，需要的堆栈空间小得多。下面就是用 shell 算法实现的排序程序 gen_shell_sort，它可以作为一个工具函数引用。

```
void gen__shell__sort(void  * base, int nelem, int elmsize.
                    int ( * fcmp) (const void * , const void * ))
{
    int i, j, jump = nelem;
    unsigned char done;
    unsigned char  * tempo,  * ptr = (unsigned char  * )base;
    tempo = (unsigned char  * ) malloc(elmsize);
    while (jump > 1) {
        jump /= 2;
        do {
            done = 1;
            for (j = 0; j < (nelem - jump); j++) {
                i = j + jump;
                if(( * fcmp)((ptr+i * elmsize),(ptr+j * elmsize))<0){
                    done = 0;
                    memmove(tempo, (ptr+i * elmsize), elmsize);
                    memmove((ptr+i * elmsize),(ptr+j * elmsize),
                            elmsize);
                    memmove((ptr+j * elmsize),tempo, elmsize);
                }
            }
        } while (!done);
    }
    free(tempo);
}
```

编写这类排序和查找函数应注意如下几点:

第 1，为了应用于各种类型的数组，入口参数 base 和 key(本例中未出现 key)都必须是 void 类型的指针，在函数内部则必须把它们强制转换为指向字符的指针。这样就可以处理最小类型的数据，即只占 1 个字节的数据。

第 2，函数内需要动态地临时分配内存，在退出函数前应释放这些临时内存。

第 3，函数内需要用 memmove 函数来拷贝数组元素。

第 4，函数内以下列形式调用比较函数:

```
( * fcmp)((ptr+i * elmsize),(ptr+j * elmsize))
```

为了进行比较，程序员还应提供与此格式相应的比较函数。在下面这个工具包中，提供了对整数、无符号整数、长整数、无符号长整数、浮点数进行比较的函数:

```
/ *   getsort.h   * /
extern int  intcmp(int  * i, int  * j);
extern long longcmp(long  * i, long  * j);
extern int  uintcmp(unsigned int  * i, unsigned int  * j);
extern long ulongcmp(unsigned long  * i, unsigned long  * j);
extern int  doublecmp(double  * x, double  * y);
extern void gen__shell__sort(void  * base, unsigned nelem,
                            unsigned elmsize,
                            int ( * fcmp) (const void * , const void * ));
extern void gen__insert__sort(void  * base, void *  key,
```

```
                         unsigned  * nelem, unsigned elmsize,
                         int ( * fcmp)(const void * , const void * ));
extern int  gen__reverse__array(void * base, unsigned nelem,
                            unsigned elmsize);
extern int  gen__merge__sort(void * base1, void * base2,
                         unsigned * nelem1, unsigned nelem2,
                         unsigned elmsize,
                         int ( * fcmp)(const void * , const void * ));
extern int  bidir__find(void * key, void * base,
                      unsigned nelem, unsigned elmsize,
                      int ( * fcmp) (const void * , const void * ));
extern int  bidir__search(void * key, void * base,
                      unsigned * nelem, unsigned elmsize,
                      int ( * fcmp)(const void * , const void * ));
extern int  set__index__table(void * table, unsigned * index, void * base,
                          unsigned tbl__size,unsigned nelem,
                          unsigned elmsize);
extern int  search__index__table(void * key,
                             void * table,
                             unsigned * index,
                             void * base,
                             unsigned tbl__size,
                             unsigned nelem,
                             unsigned elmsize,
                             int ( * fcmp)(const void * ,const void * ));
/ * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * /
/ *  gensort.c  * /
#include  <stdlib.h>
#include  <string.h>
int  intcmp(int * i, int * j)
{
    return ( * i -  * j);
}
long longcmp(long * i, long * j)
{
    return ( * i -  * j);
}
int  uintcmp(unsigned int * i, unsigned int * j)
{
    return (int)( * i -  * j);
}
long ulongcmp(unsigned long * i, unsigned long * j)
{
    return (long) ( * i -  * j);
}
int  doublecmp(double * x, double * y)
{
    if ( * x <  * y)
        return -1;
    else if ( * x >  * y)
        return 1;
```

```
    else
        return 0;
}

void gen_shell_sort(void *base, unsigned nelem,
                    unsigned elmsize,
                    int (*fcmp) (const void *, const void *))
{
    unsigned i, j, jump = nelem;
    unsigned char done;
    unsigned char *tempo, *ptr = (unsigned char *) base;
    tempo = (unsigned char *) malloc(elmsize);
    while (jump > 1) {
        jump /=2;
        do {
            done =1;
            for (j = 0; j < (nelem - jump); j++) {
              i = j + jump;
              if((*fcmp)((ptr+i*elmsize),(ptr+j*elmsize))<0){
                done = 0;
                memmove(tempo,            (ptr+i*elmsize),elmsize);
                memmove((ptr+i*elmsize),(ptr+j*elmsize),elmsize);
                memmove((ptr+j*elmsize),tempo,           elmsize);
              }
            }
        } while (!done);
    }
    free(tempo); /* restore dynamic memory */
}

void gen_insert_sort(void *base, void* key,
                    unsigned *nelem, unsigned elmsize,
                    int (*fcmp)(const void *, const void *))
{
    unsigned i, found ;
    unsigned char *ptr = (unsigned char *) base;
    unsigned char *kee = (unsigned char *) key;
    if (*nelem >0) {
        for ( i = 0, found = 0; ( i < *nelem && !found); i++)
            if ((*fcmp) ((ptr+i*elmsize),kee) > 0) {
                memmove((ptr+(i+1)*elmsize),
                        (ptr+i*elmsize),
                        (*nelem - i) * elmsize);
                found = 1;
            }
        (*nelem)++;
        if (!found)
            memmove((ptr + *nelem * elmsize), kee, elmsize);
    }
    else {
        memmove(ptr, kee, elmsize);
```

```
            ( * nelem)++;
        }
}
int  gen_reverse_array(void * base, unsigned nelem,
                       unsigned elmsize)
{
     unsigned median = nelem / 2, i, j;
     unsigned char * tempo, * ptr = (unsigned char * ) base;
     if (nelem < 3)
         return 0;
     tempo = (unsigned char * ) malloc(elmsize);
     for ( i = 0, j = nelem-1; i < median; i++, j--) {
         memmove(tempo,              (ptr+i * elmsize), elmsize);
         memmove((ptr+i * elmsize),(ptr+j * elmsize), elmsize);
         memmove((ptr+j * elmsize),tempo,             elmsize);
     }
     free(tempo);
     return 1;
}

int  gen_merge_sort(void * base1, void * base2,
                    unsigned * nelem1, unsigned nelem2,
                    unsigned elmsize,
                    int ( * fcmp)(const void * , const void * ))
{
     unsigned i, j, k;
     unsigned char * ptr0 = (unsigned char * ) base1;
     unsigned char * ptr1 = (unsigned char * ) base1;
     unsigned char * ptr2 = (unsigned char * ) base2;
     if ( * nelem1 < 1 || nelem2 < 1)
          return 0;
     ptr0 = (unsigned char * ) calloc( * nelem1, elmsize);
     memmove(ptr0, ptr1, * nelem1 * elmsize);
     for ( i=0,j=0,k=0; ( i< * nelem1 && j<nelem2); k++) {
          if (( * fcmp)((ptr0+i * elmsize),(ptr2+j * elmsize))<0){
               memmove((ptr1+k * elmsize),(ptr0+i * elmsize),elmsize);
               i++;
          }
          else {
               memmove((ptr1+k * elmsize),(ptr2+j * elmsize),elmsize);
               j++;
          }
     }
     if ( i < * nelem1) / * copy the rest of array1 * /
          memmove ((ptr1+k * elmsize),
                   (ptr0+i * elmsize),
                   ( * nelem1-j) * elmsize);
     else / * copy the rest of array2 * /
          memmove ((ptr1+k * elmsize),
                   (ptr2+j * elmsize),
                   (nelem2-j) * elmsize);
```

```
    *nelem1 += nelem2;
    free(ptr0);
    return 1;
}

int bidir_find(void *key, void *base,
               unsigned nelem, unsigned elmsize,
               int (*fcmp) (const void *, const void *))
{
    int i, j;
    unsigned char *ptr = (unsigned char *) base;
    unsigned char *kee = (unsigned char *) key;
    if (nelem < 2)
        return -1;
    for( i=nelem/2-1,j=i+1;(i>-1 || j <nelem);i--, j++) {
        if ( i > -1) {
            if ( (*fcmp) ((ptr+i*elmsize),kee) == 0)
                return i;
        }
        if (j <nelem) {
            if ( (*fcmp) ((ptr+j*elmsize),kee) == 0)
                return j;
        }
    }
    return -11;
}

int bidir_search(void *key, void *base,
                 unsigned *nelem, unsigned elmsize,
                 int (*fcmp)(const void *, const void *))
{
    unsigned i, j;
    unsigned char *ptr = (unsigned char *) base;
    unsigned char *kee = (unsigned char *) key;
    if (*nelem > 1) {
        for( i= *nelem/2-1,j=i+1;(i>-1 || j< *nelem);i--,j++) {
            if ( i > -1) {
                if ( (*fcmp) ((ptr+i*elmsize),kee) == 0)
                    return i;
            }
            if (j < *nelem) {
                if ( (*fcmp) ((ptr+j*elmsize),kee) == 0)
                    return j;
            }
        }
    }
    memmove((ptr + *nelem * elmsize), kee, elmsize);
    (*nelem)++;
    return (*nelem - 1);
}
```

```
int  set_index_table(void *table, unsigned *index, void *base,
                     unsigned tbl_size,unsigned nelem,
                     unsigned elmsize)
{
    unsigned i, j, offset = nelem / tbl_size;
    unsigned char *ptr = (unsigned char *) base;
    unsigned char *tbl = (unsigned char *) table;
    if (offset < 1)
        return 0;
    for ( i = 0, j =0; j < tbl_size; j++, i += offset) {
        memmove((tbl+j*elmsize),(ptr+i*elmsize),elmsize);
        *(index+j) = i;
    }
    return 1;
}

int  search_index_table(void *key,
                        void *table,
                        unsigned *index,
                        void *base,
                        unsigned tbl_size,
                        unsigned nelem,
                        unsigned elmsize,
                        int (*fcmp)(const void *,const void *))
{
    unsigned i, first, last, found = 0;
    unsigned median;
    unsigned char *ptr = (unsigned char *) base;
    unsigned char *tbl = (unsigned char *) table;
    unsigned char *kee = (unsigned char *) key;
    for ( i = 1; (i < tbl_size && found == 0); i++) {
        if ( (*fcmp)(kee, (tbl+i*elmsize)) <= 0 ) {
            found = 1;
            first = *(index+i-1);
            if ( i < (tbl_size - 1))
                last = *(index+i);
            else
                last = nelem - 1;
        }
    }
    if (found == 0) {
        first = *(index+tbl_size-1);
        last = nelem -1;
    }
    found = 0;
    while ( (first+1) < last && found == 0 ) {
        median = (first + last) / 2;
        if ( (*fcmp)(kee, (ptr+median*elmsize)) < 0)
            last = median;
        else if ( (*fcmp)(kee, (ptr+median*elmsize)) > 0 )
            first = median;
```

```
        else
            found =1;
    }
    if (found == 0) {
        if ( (*fcmp)(kee, (ptr+last*elmsize)) == 0)
            return last;
        else if ( (*fcmp)(kee, (ptr+first*elmsize)) == 0)
            return first;
        else
            return -1;
    }
    return median;
}
```

8.5 动态通用串与链表的比较

从上面的例子可以看出，动态通用串在处理这类问题时非常有效而方便，其结构的定义与函数 fillpoly 的要求容易吻合。

无论用动态通用串还是用链表，对于上面存储顶点坐标的应用，大都应该首先建立结构

```
typedef struct point_struct {
    int x,y;
}point;
```

如果用链表法，则链中的每个结点除包括顶点坐标外，还应包括指向下一结点的指针(单向链表)，每个结点的数据应如下所示：

```
typedef struct point_link_struct {
    point data;
    point_link_struct *next;
}point_list;
```

这就是说，每存放一个顶点坐标，要附带存储一个指针。如果是近指针(微模式，小模式，中模式)，则这个附带开销占开销的三分之一。如果是远指针(其它编译模式)，则这个附带开销占总开销的一半。另外，链表法本质上是有序数据类型，为了查找某个元素，必须逐个由前往后找，非常费时间。

动态通用串法则不一样，不管串有多长，它的附带开销始终是一个结构 vstr 的大小，在这个例子里是 15～17 个字节，如果去掉 5 个标志字节，则只需 10～12 个字节。另外，动态通用串在查找某个元素时，就像计算数组下标一样，无非是做整数乘法和加法。动态通用串法还便于使用 Turbo C 函数库中提供的许多函数（如 qsort 和 bsearch）进行串操作，无需程序员再去编写这些工具函数。

动态通用串的缺点就是插入和删除操作很不方便，需要移动内存中的内容。对于链表法，插入和删除只是引起某些指针操作，串本身不需要移动。但是，80X86之类的微机上都有高效的内存块移动指令，大部分情况下感觉不到对速度有多大影响。

第9章　高级文件处理

这一章将在第6章的基础上，结合上一章的动态通用串技术，讨论变长记录数据文件的处理问题。变长记录数据文件是常见的数据文件。例如在CAD中，所生成的各图形文件不可能一样长，要通过一个指针表把它们连接在一起。再例如联机帮助信息，各条信息文本也不可能一样长。当然，也可以用定长记录的办法来处理这些变长记录的数据文件。但用变长记录的处理办法，所生成的数据文件的长度能大大缩短。

本章介绍了一个变长记录工具函数包。这个工具包的源码中除用到许多文件输入输出函数外，还用到了第4章的pop__up窗口工具包和上一章的vstr动态通用串工具包。本章最后提供了一个把变长记录技术用于生成和放映幻灯片的例子。

9.1　变长记录(VLR)文件

变长记录VLR(Variable__Length Record)文件的处理比定长记录文件更复杂，主要应解决以下几个问题:

· 怎样从变长记录文件中查找一个记录

· 怎样插入和删除一个变长记录

· 怎样处理碎片化

9.1.1　从文件中查找一个记录

变长记录文件一般都应带有索引，索引的每一项对应一条记录，包括该记录的关键字和该记录在数据文件中的位置。也就是说，查找某一记录是通过查找数据文件的索引来进行的。同一数据文件可以有许多不同的索引，以便提供多个关键字或提供不同的检索方法。

索引可以和数据放在不同的文件中，即对于一个数据文件，可以辅之以一个或多个索引文件。这种办法使得关键字的组织可以与数据文件无关，可以易于建立多个索引或删去某些索引。但是，这种办法牵涉的文件数目多，需要同时打开多个文件，而DOS对同时打开的文件数目有一定的限制（见第6章)。

索引也可以和数据放在同一个文件里，这样可以避免同时打开多个文件，但带来了另一个棘手的问题。虽然索引的每一项可以是定长的，但因为数据中的记录数是不定的，所以索引项数也是不定的，整个索引的长度也就是不定的。例如，如果索引在前，数据在后，那么究竟需要为索引留下多大空间呢？当然，如果对最大记录数加以限制，则索引的长度也就固定了，但最好不要强加这样一个限制。

解决的办法之一就是把索引也当作一个变长记录看待。与其它记录不同的是，在打开数据文件时把这个存有索引的记录全部读入内存并始终驻留内存。当然，索引必须能在内

存中装下。

9.1.2 插入和删除记录

无论对定长记录文件还是变长记录文件，插入和删除记录都不易处理。许多定长记录数据库都有一个自由(未用的)记录表。当删除一个记录时，就把这个记录(所占的空间)加到自由记录表中。当插入一个记录时，就从自由记录表中取出一个来用。

同样的思想也可用于变长记录文件。本章介绍的变长记录文件处理中，每个记录可以由一个或多个数据块组成，属于同一记录的各数据块被连成一个单向链表，即一个记录就是一个表，所以下面"记录"和"表"不加区分。各数据块的长度可以不同，最多为256个字节。文件中有一个特殊的表，叫作自由空间表，它包含了所有曾经被使用过但后来又被闲置的数据块。当插入一个记录而需要更多的空间时，就首先使用自由空间表中的各块。倘若这个自由空间表已用完，就往文件尾再添加一些内存。当删除一个记录时，这个记录内的所有各数据块都被加到自由空间表中。

9.1.3 碎片化问题

当一个变长记录被重新使用（即被改写）时，如果新的数据比原来的大，则只需从自由空间表中再取出一些块即可。如果新的数据比原来的小，为了不浪费空间，应把多余空间返回给自由空间表，但这样就可能引起碎片化问题。如果碎片化过于严重，则不但不节省空间，反而可能浪费空间，因为这时需要更多的指针。指针占据空间的比例过大，文件存取的速度也会降低。

碎片化问题在每一个内存管理系统、磁盘管理系统以及DOS的文件系统中都存在。在本章的变长记录文件工具包中，是采取下列办法来处理碎片化问题的。

只要有可能，VLR算法总是试图用完整的数据块。当增加一个新记录时，尽可能使用完整的数据块。对于最后剩下的不足一块的那些字节，则再加一点余度。如果加上余度后仍不够一个整块，则分配加上余度后的字节数。如果加上余度后超过一个整块，则分配一个整块。这样，当记录稍许增多一点时，不需要再分配空间。程序中预先垫的字节数是由全局变量padsize决定的。根据不同的应用情况，合理地选择padsize的大小，可以使碎片化问题得到缓解。

当更新一个记录时，首先将使用它原来的空间。如果新数据较少，则把多余的块返回给自由空间表。为了减少碎片化，数据块一旦被建立，就永远不再被划分。这就是说，不会把原来的块再次划分，而把剩下的部分再返回给自由空间表。只有那些完全不用的块才返回给自由空间表。如果新数据较多，但不超过预先留下的余度，则也不需扩展空间。

预先留有余度带来了一个麻烦，这就是块的长度与实际使用的长度不相等。必须有一种办法来表示实际数据的结尾。例如，ASCIIZ串是以ASCII码0表示字符串结尾，二进制数据可以用某一特殊系列来表示数据的结尾。这一章的工具包采取的是另一种办法：在每一个记录的第1块，记录下这个记录的实际字节数。

9.1.4 VLR 文件格式

这个工具包所用的变长记录文件的格式如下所示:

偏移量	含义
文件头部分	
0～3	自由空间表地址
4～7	文件结束地址
8～11	索引地址
12～16	记录数
块的余下部分	用户定义参数
数据部分	
	第1个数据块
	第2个数据块
	.
	.
	.

文件头占一个数据块，由 5 个部分组成。自由空间表地址是自由空间的起始地址，所有地址都是相对于文件头 0 字节的偏移量。文件结束地址实际上就是文件的长度。应该注意，自由空间表是个特殊记录，因此也是文件内的一部分。如果自由空间表地址等于文件结束地址，那就意味着文件内已没有自由空间了，若再增加记录，就必须扩充文件长度。

索引地址是索引的起始地址。索引也是一个特殊记录。并不是每个数据文件一定要有索引，如何建立和利用索引是应用程序的问题，这个工具包不对这个参数进行操作，也不提供与索引有关的函数。

文件头的最后一个固定参数是文件中的记录数（不是块数）。从文件头的结构可以看出，文件头不能少于 16 个字节，但可以多于 16 个字节，多余的部分保留，供用户使用。用户可以用这个空间存储任何数据。例如，如果要建立多种索引，这个空间就可用来存其它种索引的地址。

9.1.5 VLR 记录格式和数据块格式

每个记录是由一个或多个数据块组成的，这些数据块形成一个单向链表。

每个记录的第 1 个数据块的格式如下:

偏移量	长度	含 义
0	1	该块的长度，不包括该字节本身
1	1	标志字节
2	4	下一块的地址，从文件头算起
6	2	该记录的实际长度，单位是字节
8	其余	数据本身

每个记录其余各数据块的格式与第 1 个数据块稍有差别，即从偏移量 6 开始就是数据本身。只在第 1 个数据块中才存有记录实际长度。

从这个格式可以看出，即使一个数据块内不存有任何数据，也至少要占 6 个字节。一

个完整的数据块究竟多少个字节，是在编译时由宏 BLK__SIZE 定义的，本工具包中暂定为 256 字节。因各数据块长度不一致,故需要指向下一块的指针，即下一块的地址。对于一个记录的最后一个数据块，它的下一块地址应是 0x0000。每个记录第 1 个数据块的起始地址可以当作记录的标志(即记录的“名字”)使用。

标志字节是为了帮助进行检查的，目前它的内容是 0xfd。当检索一个记录时，如果发现同步字节不是 0xfd，则意味着数据已发生混乱。

9.2 VLR 工具包

9.2.1 7 个工具函数

变长记录工具包共包括 7 个工具函数:

```
int openvlr(char *fname, char *access__type, long rs);
int readhdr(int fh, long *fs, long *fe, long *ita, long *nl);
int writehdr(int fh, long fs, long fe, long ita, long nl);
int getvlr(int fh, vstr *s, long pos);
int reuse__vlr(int fh,long locn,vstr *w,int offset,int use__fs);
int delvlr(int fh, long locn);
int addvlr(int fh, vstr *w, long *locn);
```

(1) 函数 openvlr 打开一个变长记录文件。第 1 个参数是文件名，第 2 个参数是文件存取方式，其取值范围和含义与 Turbo C 的 fopen 函数一样，有“r”、“w”、“a”、“r+”、“w+”、“a+”6 种，还可选择文本(t)和二进制(b)方式。这里的工具包，不允许选择追加方式(a,a+)和文本方式，否则可能出现问题。第 3 个参数是说明数据部分从文件的什么位置开始。如果仅打开已有的文件而不是建立新文件，则第 3 个参数没有作用。

如果不成功，函数返回值为-1，如果成功，函数返回值是一个文件柄，以后就可以用这个文件柄来对文件进行操作。应该注意，这个文件柄是由第 6 章基本文件处理工具包定义的文件柄，不是 DOS 定义的文件柄。因此，用这个文件柄只能调用这两个工具包内的函数去对文件进行存取，而不能调用 Turbo C 库内的函数存取文件。

(2) 函数 readhdr 和 writehdr 分别读取和设置变长记录文件头的 4 个固定参数。其中，fh 是文件柄，fs 是自由空间表地址，fe 是文件尾地址，ita 是索引地址，nl 是记录数。文件头中用户自定义的部分必须由用户自己编写工具函数去存取。因为这 4 个参数至关重要，所以每次调用 writehdr 后，都会刷清缓冲区。

(3) 函数 getvlr 是从文件 fh 的位置 pos 处读一个记录，存至动态通用串 s 的数据区。这个工具包的各记录存取函数，认为变长记录文件中的每个记录存放的是一个动态通用串的数据部分。如果不是这样，则这个函数以及下面（除 delvlr 外）的所有函数都必须修改。

(4) 函数 extend__file 是一个内部函数，不作为工具函数对外提供。它被用来扩充文件 fh 的长度，以便容纳动态通用串 s 中从偏移量 offset 开始的数据部分。如果不成功，返回值为-1。如果成功，返回值为 0，且通过参数 locn 返回扩充部分在文件 fh 内的偏移量。这个函数首先使文件增加完整的数据块，对于最后剩下的那些字节，增加一个

padsize。如果增加后超过一整块，则就增加一整块。

(5) 函数 delvlr 从变长记录文件 fh 中删除一个记录 locn。locn 是被删除记录在文件中的起始位置。所谓删除就是把该记录中所有各数据块归还到自由空间表中。在程序中，是把原自由空间表链接到被删除记录的最后一块，而把被删除记录的第 1 块当作自由空间表的新链头。同时，使文件中的记录数减 1。

(6) 函数 addvlr 往文件 fh 中增加一个新记录，用来存放动态通用串 w 中的数据部分。这个函数首先查看文件中是否有自由空间。如果有，则调用 reuse_vlr 函数从自由空间分配一些数据块来满足需要。如果没有，则调用 extend_file 函数扩大文件的长度来满足需要。如果不成功，返回值为-1；如果成功，返回值为 0，且通过参数 locn 返回该新记录在文件 fh 中的位置。

(7) 函数 reuse_vlr 是这个工具包中最复杂的一个函数，它重新使用变长记录文件 fh 中的记录 locn 来存放动态通用串 w 中从偏移量 offset 开始的数据部分。参数 use_fs 说明是否使用自由空间表。如果在调用这个函数时，希望在使用完记录 locn 后立即通过扩展文件来满足空间要求，则应设置 use_fs 为 1。如果在调用这个函数时，希望在使用完记录 locn 后，先通过重新分配自由空间表再通过扩展文件来满足空间要求，则应设置 use_fs 为 0。注意，从用户的应用程序中调用这个函数时，总是应该设置 offset 和 use_fs 为 0，这两个参数是为函数本身递归调用而设置的。

这个函数首先使用指定的记录。通过内部的 getvlrelem 函数，把文件 fh 中位置 locn 中的一块读入缓冲区 datablk，同时在 nxtlocn 中取得下一块的地址。从读得的内容中知道该块的实际长度，于是就把动态通用串 w 中偏移量 offset 处开始的同样长度的内容拷贝到缓冲区 datablk 的数据部分，并修改偏移量 offset 的值。改写完缓冲区 datablk 后，再把这个缓冲区通过 outbytes 函数写回到文件 fh 中原来的位置。然后再次调用 getvlrelem，开始一轮新的循环。

如果原记录足够长，则循环到一定的时候，会有某一数据块的实际长度比通用串 w 中剩下的内容要长。这时，不对这一数据块再次划分，而是把整个这一块供 w 中剩下的字节使用。同时把该记录中未用的各块链接到自由空间表前面，最后退出该函数。

如果原记录不够长，则查看 use_fs 参数。如果这个参数为 1，或虽然这个参数为 0 但已没有自由空间(fs = fe)，则调用 extend_file 函数以满足空间要求。如果参数 use_fs 为 0 且有自由空间(fs≠fe)，则递归调用该函数本身从自由空间来获取空间。递归调用时，参数 use_fs 设置为 1。

自由空间表也是一个记录，符合记录的组织格式，因此可以递归调用。重复前面说过的过程，逐块往下分，直到某一块满足通用串 w 的全部要求为止。因为是在自由空间表这个特殊记录中，故这个记录未用的各块不用再链接到自由空间表中（它们本来就在这个表中），只需改变一下表头位置即可。如果自由空间表不够用，则调用 extend_file 函数扩充文件长度。

这个递归函数最多只递归一次，不会发生堆栈溢出问题。

工具包头文件 vlr.h

```
/*    vlr.h    */
```

```
extern int openvlr(char *fname, char *access_type, long rs);
extern int readhdr(int fh, long *fs, long *fe, long *ita, long *nl);
extern int writehdr(int fh, long fs, long fe, long ita, long nl);
extern int getvlr(int fh, vstr *s, long pos);
extern int reuse_vlr(int fh,long locn,vstr *w,int offset,int use_fs);
extern int delvlr(int fh, long locn);
extern int addvlr(int fh, vstr *w, long *locn);
extern int padsize;
```

工具包程序文件 vlr.c:

```
/* vlr.c */
#include <stddef.h>
#include <io.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include "popup.h"
#include "vstr.h"
#include "fileio.h"
#include "vlr.h"

#define SYNC_BYTE 253
#define NOLINK    0
#define BLK_SIZE  256
#define DATA_SIZE 250

int     padsize = 10;
static  int getvlrelem(int fh, long pos, long *nxtlocn,unsigned char *datablk);
static  int extend_file(int fh,vstr *s,long *locn,int offset);

int openvlr(char *fname, char *access_type, long rs)
{
    int fh;
    long zero = 0L;
    if ((fh = openfile(fname, access_type,1)) != -1) {
        if (*access_type =='w') {
            outbytes(fh, 0L, (unsigned char *)&rs,  4);
            outbytes(fh, 4L, (unsigned char *)&rs,  4);
            outbytes(fh, 8L, (unsigned char *)&zero,4);
            outbytes(fh,12L, (unsigned char *)&zero,4);
        }
    }
    return   fh;
}

int readhdr(int fh, long *fs, long *fe, long *ita, long *nl)
{
    unsigned char datablk[16];
    int  rval;
```

```
    rval=inbytes(fh,0L,datablk,16);
    memcpy((char *)fs, datablk,   4);
    memcpy((char *)fe, datablk+4, 4);
    memcpy((char *)ita,datablk+8, 4);
    memcpy((char *)nl, datablk+12,4);
    if (rval < 0) return -1;
    return 0;
}

int writehdr(int fh, long fs, long fe, long ita, long nl)
{
    unsigned char datablk[16];
    int rval;
    memcpy(datablk,    (char *)&fs, 4);
    memcpy(datablk+4,  (char *)&fe, 4);
    memcpy(datablk+8,  (char *)&ita,4);
    memcpy(datablk+12, (char *)&nl, 4);
    rval =outbytes(fh,0L, datablk, 16);
    if (rval < 0) return -1;
    if (fflush(ft[fh].fp)) {
        sayerr(FERR,"Writing vlr header data in %s\r\n",
               ft[fh].name);
        return -1;
    }
    return 0;
}

static int getvlrelem(int fh, long pos, long *nxtlocn,
                      unsigned char *datablk)
{
    int bytesread;
    inbytes(fh, pos, datablk, 1);
    bytesread = inbytes(fh, -1L, datablk+1, *datablk);
    if (datablk[1] != SYNC_BYTE) {
        sayerr(SERR, "Sync byte error at %ld\r\n",pos);
        return -1;
    }
    else {
        memcpy((char *)nxtlocn, datablk+2, 4);
    }
    return bytesread;
}

int getvlr(int fh, vstr *s, long pos)
{
    unsigned char datablk[256];
    unsigned int offset, actual_len, data_size, data_indx;
    int nblks;
    s->currlen = 0;
    offset = 0;
    nblks = 0;
```

```
        while(pos != 0L) {
            if ((getvlrelem(fh, pos, &pos, datablk)) == -1) {
                s->currlen = offset / s->esize;
                return -1;
            }
            else {
                if (++nblks == 1) {
                memcpy(&actual_len, datablk+6, 2);
                    if (actual_len % s->esize) {
                        sayerr(SERRF,"Data length doesn't jive "
                                "with vstr type.\r\n"
                                "Data len = %u,"
                                "element size = %u\r\n",
                                 actual_len, s->esize);
                        return -1;
                    }
            if ((actual_len / s->esize) >s->dimlen) {
                if (!(redimvstr(s,actual_len / s->esize)))return -1;
            }
            data_indx = 8;
            data_size = *datablk-7;
            }
            else {
                data_indx = 6;
                data_size = *datablk-5;
            }
            memcpy((char *)(s->data)+offset,datablk+data_indx,
                   data_size);
            offset += data_size;
            }
        }
        s->currlen = actual_len / s->esize;
        return actual_len;
}

static int extend_file(int fh, vstr *s, long *locn, int offset)
{
        long fs,fe,ita,nrecs,link;
        unsigned int i,nb,nl,len;
        unsigned char datablk[256];
        len = s->currlen * s->esize - offset;
        if (offset == 0) len += 2;
        if (readhdr(fh, &fs, &fe, &ita, &nrecs) == -1) return -1;
        nb = len / DATA_SIZE;
        nl = len - nb*DATA_SIZE;
        link = fe;
        *locn= fe;
        if (nb)  {
            for (i = 1; i<=nb; i++) {
                if (i==nb && nl==0) link = 0;
                else link += BLK_SIZE;
```

```
                memcpy(datablk+2, (char *)&link, 4);
                datablk[0] = BLK__SIZE-1;
                datablk[1] = SYNC__BYTE;
                if (offset == 0) {
                    len -= 2;
                    memcpy(datablk+6,&len, 2);
                    memcpy(datablk+8,((char *)(s->data)),
                            DATA__SIZE-2);
                    offset += DATA__SIZE - 2;
                }
                else {
                    memcpy(datablk+6,((char *)(s->data))+offset,
                            DATA__SIZE);
                    offset += DATA__SIZE;
                }
                if(outbytes(fh,fe,datablk,BLK__SIZE) == -1)
                    return -1;
                fe += BLK__SIZE;
            }
        }
        if (nl) {
            link = NOLINK;
            nl += padsize;
            if (nl > DATA__SIZE) nl = DATA__SIZE;
            datablk[0] = nl+5;
            datablk[1] = SYNC__BYTE;
            memcpy (datablk+2, (char *)&link,4);
            if (offset == 0) {
                len -= 2;
                memcpy (datablk+6,&len, 2);
                memcpy (datablk+8,((char *) (s->data)), nl-2);
            }
            else {
                memcpy (datablk+6,((char *)(s->data))+offset,nl);
            }
            if (outbytes(fh, fe, datablk, nl+6) == -1) return -1;
            fe += nl + 6;
        }
        if (writehdr(fh, fe, fe, ita, nrecs) == -1) return -1;
        return 0;
}

int reuse__vlr(int fh,long locn,vstr *w,int offset,int use__fs)
{
        unsigned int nb, nu, ls, len;
        unsigned char datablk[256];
        long oldfs,nxtlocn,fs,fe,ita,nrecs,dmy;
        len = w->currlen * w->esize;
        if (offset == 0) len += 2;
        if (readhdr(fh,&fs,&fe,&ita,&nrecs) == -1) return -1;
        do {
```

```
if (getvlrelem(fh, locn, &nxtlocn, datablk) == -1)
        return -1;
nb = *datablk;
if (nb <= 5) {
      locn = nxtlocn;
}
else {
      nu = nb-5;
      ls = len - offset;
      if (nu >= ls) {
            memset(datablk+2, 0, 4);
            if (offset == 0) {
                  len -=2;
                  memcpy(datablk+6,&len, 2);
                  memcpy(datablk+8,((char *)(w->data)),ls-2);
            }
            else {
                  memcpy(datablk+6,((char *)(w->data))+offset,
                        ls);
            }
            if (outbytes(fh,locn,datablk, *datablk+1) == -1)
                  return -1;
            oldfs = fs;
            if (nxtlocn)  {
                  fs = nxtlocn;
            }
            else {
                  if (use_fs) fs = fe;
            }
            if (!use_fs && (nxtlocn != 0)) {
                  do {
                        locn = nxtlocn;
                        if (getvlrelem(fh,locn,&nxtlocn,datablk)
                              ==-1) return -1;
                  } while (nxtlocn);
                  if  (oldfs != fe)  {
                        memcpy(datablk+2, (char *)&oldfs, 4);
                        if(outbytes(fh,locn,datablk, *datablk+1)
                                    ==-1) return -1;
                  }
            }
            writehdr(fh, fs, fe, ita, nrecs);
            break;
      }
      else {
            if (offset ==0) {
                  len -= 2;
                  memcpy(datablk+6,&len, 2);
                  memcpy(datablk+8,((char *)(w->data)),nu-2);
                  offset +=nu-2;
            }
```

```
                    else {
                        memcpy(datablk+6,((char *)(w->data))+offset,
                                    nu);
                        offset += nu;
                    }
                }
                if (nxtlocn) {
                    if (outbytes(fh, locn, datablk, *datablk+1) == -1)
                        return -1;
                    locn = nxtlocn;
                }
            }
        }while (nxtlocn);
        if (nu < ls) {
            if (use_fs) memcpy(datablk+2, (char *)&fe, 4);
                else   memcpy(datablk+2, (char *)&fs, 4);
            if (outbytes(fh,locn,datablk, *datablk+1)==-1) return -1;
            if (use_fs || (fs==fe)) {
                if (extend_file(fh,w,&dmy,offset)==-1) return -1;
            }
            else {
                if (reuse_vlr(fh,fs,w,offset,1) == -1) return -1;
            }
        }
        return 0;
}

int delvlr(int fh, long locn)
{
        unsigned char datablk[256];
        long newfs,fs,fe,ita,nrecs,nxtlocn;
        if (readhdr(fh, &fs, &fe, &ita, &nrecs) == -1) return -1;
        newfs = locn;
        nxtlocn = locn;
        do {
            locn = nxtlocn;
            if((getvlrelem(fh,locn,&nxtlocn,datablk))==-1)return -1;
        } while (nxtlocn);
        if (fs != fe) {
            memcpy(datablk+2, (char *)&fs, 4);
            if(outbytes(fh,locn,datablk,*datablk+1)==-1)return -1;
        }
        if (writehdr(fh, newfs, fe, ita, nrecs-1) == -1) return -1;
        return 0;
}

int addvlr(int fh, vstr *w, long *locn)
{
        long oldfs, fs, fe, ita, nrecs;
        if(readhdr(fh,&oldfs,&fe,&ita,&nrecs) == -1) return -1;
        if (oldfs == fe) {
```

```
        if (extend_file(fh, w, &fs, 0) == -1) return -1;
    }
    else {
        if (reuse_vlr(fh, oldfs, w, 0, 1) == -1) return -1;
    }
    if (readhdr(fh, &fs, &fe, &ita, &nrecs) == -1) return -1;
    if (writehdr(fh, fs, fe, ita,nrecs+1) == -1) return -1;
    *locn = oldfs;
    return 0;
}
```

9.2.2 索引处理

本节举例说明如何为一组变长记录建立一个索引，如何把这个索引本身也作为一个变长记录存储起来。在这个例子中，索引中将不存储任何关键字，而只存储各记录地址。最容易的办法就是把索引当作由长整数组成的动态串。

程序首先建立 5 个动态通用串，前 3 个用来存储普通字符串并被初始化，第 4 个用作临时存储单元，第 5 个用作索引，存储前 3 个记录的起给地址。接着，程序建立一个变长记录文件，把 3 个普通动态字符串作为 3 个记录写入这个文件，每写入一个记录后，把其在文件中的起始地址追加到用作索引的动态通用串中，最后把这个索引动态通用串也写到文件中，更新文件的文件头部分，以便把索引的起始地址加到文件头中去。程序的最后部分是从文件中把各记录读出来。它首先读出文件头部分，找到索引的地址，把索引读入一个动态通用串 index 中，这个动态通用串中的每个串是一个长整数，代表另一个普通动态字符串记录在文件中的起始位置，再据此把这些普通记录读出来。

```
#include <stdio.h>
#include "popup.h"
#include "fileio.h"
#include "vstr.h"
#include "vlr.h"
void main( )
{
    int  fh;
    vstr rec1data  =  NULLVSTR;
    vstr rec2data  =  NULLVSTR;
    vstr rec3data  =  NULLVSTR;
    vstr buffer    =  NULLVSTR;
    vstr index     =  NULLVSTR;
    long locn;
    long fs,fe,ia,nr;
    init_win( );
    init_files( );
    clrvstr(&rec1data,20,sizeof(char),5);
    vstrins(&rec1data,0 ,"This is record one  \0",19);
    clrvstr(&rec2data,20,sizeof(char),5);
    vstrins(&rec2data,0 ,"This is record two  \0",19);
    clrvstr(&rec3data,20,sizeof(char),5);
    vstrins(&rec3data,0 ,"This is record three\0",25);
```

```
    clrvstr(&buffer   ,20,sizeof(char),5);
    clrvstr(&index    ,10,sizeof(long),5);
    if((fh=openvlr("mydata.vlr","w+b",32L))!=-1) {
        mprintf("New vlr file created successfully\r\n");
        addvlr(fh,&rec1data,&locn);
        vstrcat(&index,&locn);
        addvlr(fh,&rec2data,&locn);
        vstrcat(&index,&locn);
        addvlr(fh,&rec3data,&locn);
        vstrcat(&index,&locn);
        addvlr(fh,&index,&locn);
        readhdr(fh,&fs,&fe,&ia,&nr);
        writehdr(fh,fs,fe,locn,nr);
        readhdr(fh,&fs,&fe,&ia,&nr);
        getvlr(fh,&index,ia);
        locn=((long  *)(index.data))[0];
        getvlr(fh,&buffer,locn);
        mprintf("First  record at locn %ld contains: '%s'\r\n",
                locn,buffer.data);
        locn=((long  *)(index.data))[1];
        getvlr(fh,&buffer,locn);
        mprintf("Second record at locn %ld contains: '%s'\r\n",
                locn,buffer.data);
        locn=((long  *)(index.data))[2];
        getvlr(fh,&buffer,locn);
        mprintf("Third  record at locn %ld contains: '%s'\r\n",
                locn,buffer.data);
        mprintf("Free space at %ld\r\n",fs);
        mprintf("File ends  at %ld\r\n",fe);
        mprintf("Number of records is %ld\r\n",nr);
        closefile(fh);
    }
    else {
        mprintf("File not created successfully\r\n");
    }
}
```

这个程序的输出结果可能如下所示:

```
New vlr file created successfully
First  record at locn 32 contains:  'This is record one'
Second record at locn 69 contains:  'This is record two'
Third  record at locn 106 contains:  'This is record three'
free space at 179
File ends at 179
Number of records is 4
```

9.2.3 工具包应用举例：制作和显示幻灯片

这个例子由两部分程序组成：一部分是制作幻灯片并将其存放到当前目录下的文件objects.vlr 中；另一部分是从 objects.vlr 中读回并显示这些幻灯片。objects.vlr 文件是一个变长记录文件，每张幻灯片的数据在文件中占一个记录。

制作幻灯片并存盘的程序：

```
#include    <graphics.h>
#include    <stdio.h>
#include    <string.h>
#include    <process.h>
#include    "mouse.h"
#include    "vstr.h"
#include    "popup.h"
#include    "fileio.h"
#include    "vlr.h"
#define     rectxul 0
#define     rectyul 32
typedef struct {
      int x,y;
}point;
typedef struct {
      int x,y;
      char label[10];
}button;
button draw__menu[ ] = {
      { rectxul,        rectyul-16, "Store" },
      { rectxul+80,     rectyul-16, "Fill"  },
      { rectxul+160,    rectyul-16, "Clear" },
      { rectxul+240,    rectyul-16, "Quit " }
};
int  num__draw__menu__items = sizeof(draw__menu) / sizeof(button);
int  mouse__on__entry(button * menu, int nitems);

void main( )
{
     int gd=DETECT, gm=0, rectxlr, rectylr;
     unsigned int k;
     vstr v      = NULLVSTR;
     vstr index  = NULLVSTR;
     point p     = {0,0};
     int first__time = 1, i, command, done;
     int starting__point;
     int fh;
     long locn, fs, fe, ita, nr;
     init__win( );
     init__files( );
     if ((fh = openvlr("objects.vlr","w+b",32L)) == -1) {
          printf("Error creating objects.vlr\n");
```

```
        exit(1);
    }
    initgraph(&gd,&gm,"");
    init_mouse(MOUSE_NEEDED, gd, gm);
    rectxlr = getmaxx( );
    rectylr = getmaxy( );
    clrvstr(&index,30,sizeof(point),5);
    clrvstr(&index,20,sizeof(long) ,5);
    mouse_off(1);
    for(i=0; i < num_draw_menu_items; i++) {
        outtextxy(draw_menu[i].x,draw_menu[i].y,
                draw_menu[i].label );
    }
    mouse_on(1);
    mouse_off(1);
    rectangle(rectxul, rectyul, rectxlr, rectylr);
    mouse_on(1);
    setviewport(rectxul+1, rectyul+1, rectxlr-1, rectylr-1,1);
    setfillstyle(8,4);
    done = 0;
    do {
        while(!(k = mouse_trigger(0)));
        if (k == LEFT_MOUSE_REL) {
            command=mouse_on_entry(draw_menu,num_draw_menu_items);
            switch(command) {
                case 0:
                    addvlr(fh, &v, &locn);
                    vstrcat(&index,&locn);
                    mouse_off(1);
                    clearviewport( );
                    mouse_on(1);
                    v.currlen=0;
                    first_time = 1;
                    break;
                case 1:
                    vstrcat(&v,((point *)v.data)+starting_point);
                    mouse_off(1);
                    fillpoly(v.currlen, (int far *) (v.data));
                    mouse_on(1);
                    first_time = 1;
                    break;
                case 2:
                    mouse_off(1);
                    clearviewport( );
                    mouse_on(1);
                    v.currlen = 0;
                    first_time = 1;
                    break;
                case 3:
                    done = 1;
                    addvlr(fh, &index, &locn);
```

```
                        readhdr(fh, &fs, &fe, &ita,&nr);
                        writehdr(fh, fs, fe, locn,nr);
                        closefile(fh);
                        break;
                    default:
                        if (first__time) {
                            mouse__off(1);
                            putpixel(mouse__grph__x - rectxul,
                                         mouse__grph__y - rectyul, 15);
                            mouse__on(1);
                        }
                        else {
                            mouse__off(1);
                            line(p.x, p.y,
                                    mouse__grph__x - rectxul,
                                    mouse__grph__y - rectyul);
                            mouse__on(0);
                        }
                        p.x = mouse__grph__x - rectxul;
                        p.y = mouse__grph__y - rectyul;
                        vstrcat(&v,&p);
                        if(first__time)starting__point=v.currlen-1;
                        first__time = 0;
                }
            }
        }while (!done);
        mouse__reset( );
        closegraph( );
}

int  mouse__on__entry(button *menu, int nitems)
{
        int i;
        for(i=0; i < nitems; i++) {
            if(mouse__in__box(1, menu[i].x, menu[i].y,
                            menu[i].x + strlen(menu[i].label)*8 - 1,
                            menu[i].y + 7))
                return i;
        }
        return -1;
}
```

这个程序执行后，将首先在屏幕上画出一个边框，在上框线的外面显示4个选择项“Store Fill Clear Quit”，如图9-1所示。然后，用户就可以在框内用鼠标画图或选择某一选择项。图是由折线组成的，每按一下鼠标的左按钮，就在该点与上一点之间连一条线。第1次按左按钮只画一个点，即起始点。

4个选择项的作用如下：

Store:　把屏幕上已画的内容作为一张幻灯片，即作为一个记录写到盘文件上，在

索引记录中增加一项索引，清除屏幕，为画下一张幻灯片做好准备。

Fill:　在最后一点和起始点之间连一条线，形成封闭多边形并填充之。下一次鼠标左按钮按下的点将成为新的起始点。这样，可以在同一张幻灯片上画出几个彼此孤立的多边形。

Clear:　废除屏幕上已画出的所有内容，清除屏幕，为重画一张幻灯片做准备。

Quit:　把索引写到盘文件上，更新盘文件的文件头，以反映索引的变化。关闭文件，复位鼠标，结束程序的运行。注意，这个选择项不把最后一屏的内容保存到盘上去。

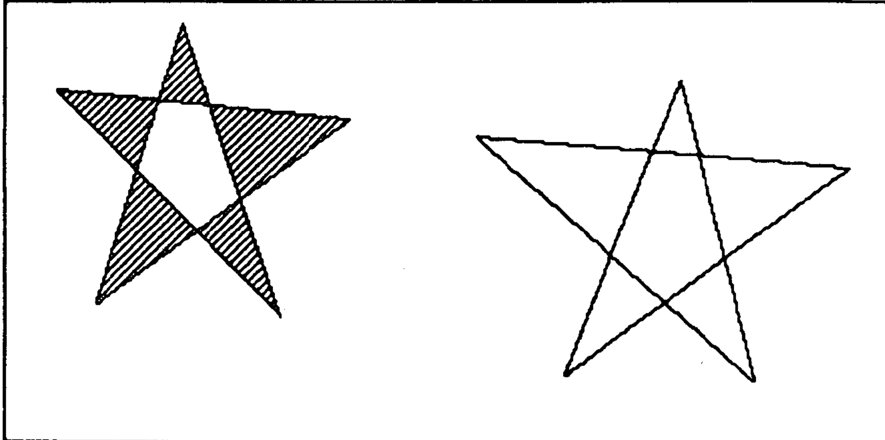

图 9—1　制作幻灯片的程序

读盘文件并显示幻灯片程序：

```
#include  <graphics.h>
#include  <stdio.h>
#include  <string.h>
#include  <process.h>
#include  "mouse.h"
#include  "vstr.h"
#include  "popup.h"
#include  "fileio.h"
#include  "vlr.h"
#define  rectxul 0
#define  rectyul 32
typedef struct {
     int x,y;
     char label[10];
} point;
typedef struct {
     int x, y;
     char label[10];
} button;
button show__menu[ ] = {
     {rectxul,       rectyul-16,    "First"     },
     {rectxul+50,    rectyul-16,    "Last"      },
     {rectxul+100,   rectyul-16,    "Next"      },
```

```
    {rectxul+150,   rectyul-16,     "Prev"       },
    {rectxul+200,   rectyul-16,     "quit"       },
};
int num_show_menu_items = sizeof(show_menu) / sizeof(button);
vstr v      = NULLVSTR;
vstr index = NULLVSTR;
char buffer[20];
int mouse_on_entry(button *menu, int nitems);
char far *bios_video_area = (char far *)0x00400049L;

void main( )
{
    int gd=DETECT, gm=0, i, command, obj, fh, rectxlr, rectylr;
    unsigned int k;
    point p;
    long fs, fe, ita, nr;
    init_win( );
    init_files( );
    if ((fh = openvlr("objects.vlr","rb",32L)) == -1) {
        printf("Error opening file objects.vlr\n");
        exit(1);
    }
    initgraph(&gd,&gm,"");
    init_mouse(MOUSE_NEEDED, gd, gm);
    rectxlr = getmaxx( );
    rectylr = getmaxy( );
    clrvstr(&v,     30,sizeof(point),5);
    clrvstr(&index,20,sizeof(long), 5);
    readhdr(fh, &fs, &fe, &ita, &nr);
    getvlr(fh, &index, ita);
    mouse_off(1);
    for(i=0; i < num_show_menu_items; i++) {
        outtextxy(show_menu[i].x, show_menu[i].y,
                  show_menu[i].label);
    }
    mouse_on(1);
    mouse_off(1);
    rectangle(rectxul, rectyul, rectxlr, rectylr);
    mouse_on(1);
    setviewport(rectxul+1, rectyul+1, rectxlr-1, rectylr-1,1);
    setfillstyle(8,4);
    command = 0;
    obj = 0;
    do {
        setviewport(rectxul-60,rectyul-16,rectxlr,rectylr-8,1);
        mouse_off(1);
        clearviewport( );
        sprintf(buffer, "%d of %d", obj+1, index.currlen);
        outtextxy(0,0,buffer);
        mouse_on(1);
        setviewport(rectxul+1,rectyul+1,rectxlr-1,rectylr-1,1);
```

```
            while(!(k = mouse__trigger(0)));
            if (k == LEFT__MOUSE__REL) {
                command=mouse__on__entry(show__menu,num__show__menu__items);
                if (command !=4) {
                    switch(command) {
                    case 0:
                        obj = 0;
                        break;
                    case 1:
                        obj = index.currlen-1;
                        break;
                    case 2:  / * next object,wrap around to beginning * /
                        if (++obj == index.currlen) obj = 0;
                        break;
                    case 3:
                        if (--obj < 0) obj = index.currlen -1;
                        break;
                    default:
                        obj = 0;
                    }
                    if (command >= 0 && command <= 3) {
                        getvlr(fh,&v,((long  * )(index.data))[obj]);
                        mouse__off(1);
                        clearviewport( );
                        fillpoly(v.currlen, (int far * ) (v.data));
                        mouse__on(1);
                    }
                }
            }
        }while(command!=4);
        mouse__reset( );
        closegraph( );
        closefile(fh);
}

int  mouse__on__entry(button  *menu, int nitems)
{
        int i;
        for(i=0; i < nitems; i++) {
            if (mouse__in__box(1,menu[i].x, menu[i].y,
                            menu[i].x+strlen(menu[i].label) * 8 - 1,
                            menu[i].y + 7))
                return i;
        }
        return -1;
}
```

这个程序执行后，也是首先在屏幕上画出边框，在顶框线的上面显示 5 个选择项："Frist Last Next Prev Quit"，然后等待用户进行选择，如图 9-2 所示。选择是通过鼠标左按钮进行的。前 4 个选择项分别表示先选哪一张幻灯片:第一张，最后一张，下

一张，上一张。第 5 个选择项是结束程序的运行。在显示幻灯片的同时，还在屏幕的左上角报告当前显示的是第几张幻灯片和总共有多少张幻灯片。

这两个程序实际上是存在问题的。变长记录工具包用到了基本文件输入／输出工具包(见第 6 章)，后者又用到了错误报告工具包(见第 4 章)，再后者又用到了 pop_up 弹出窗口工具包(见第 4 章)。但第 4 章介绍的弹出窗口工具包只适用于文本方式。所以，当在图形方式下制作幻灯片时，如果出现错误，报出的错误信息将是不可识别的。修正的办法是重新编写 sayerr 函数，使它工作在图形方式下，并相应地修改头文件。

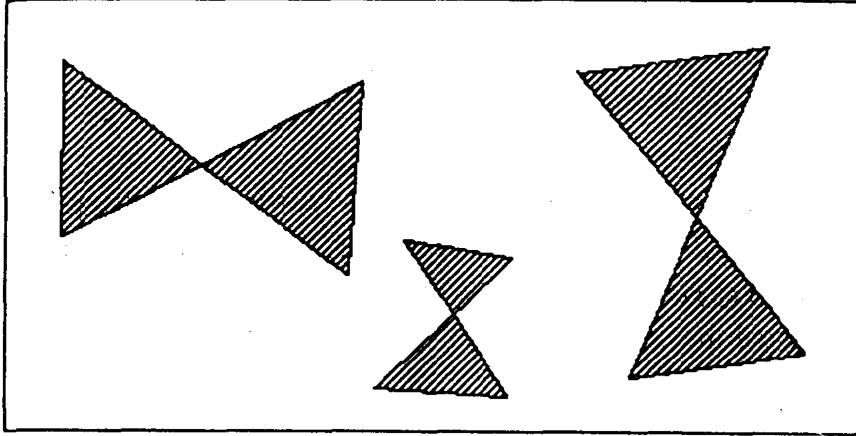

图 9-2 显示幻灯片的程序

第 10 章　内存和程序管理

内存管理和程序执行是紧密相关的，这是因为，当一个程序执行另一个子程序时，一方面要为子程序分配内存，另一方面要通过内存在父程序和子程序之间进行通信，这就是内存管理问题。

10.1　PSP 和环境

10.1.1 PSP

当 DOS 装入一个程序时，它分配两个内存块：一个用于存储程序本身，另一个用于程序存储从它的父程序(当在 DOS 命令行下装入时，这个父程序就是 DOS 的命令处理程序 COMMAND.COM)继承下来的环境变量。环境变量包含了保证程序正确执行的一些重要信息。程序块中最前面的 256 个字节是 PSP，这 256 个字节的分配情况如下：

偏移量	长度	含　义
0	2	指令 INT 20H
2	2	剩余内存的第 1 个段地址
4	1	保留
5	5	长调用指令(CP／M 操作系统用)
a	4	保存父程序的终止程序地址
e	4	保存父程序的 Ctrl__Break 程序地址
12	4	保存父程序的严重错处理程序地址
16	22	DOS 工作区
2c	2	父程序传下来的环境段地址
2e	34	DOS 工作区
50	3	INT 21H 指令和远返回指令
53	2	保留
55	7	第 1 个 FCB 的扩展部分
5c	9	第 1 个 FCB 的标准部分
65	7	第 2 个 FCB 的扩展部分
6c	9	第 2 个 FCB 的标准部分
80	128	命令行参数和盘传输区 DTA

在 PSP 中，有些信息是从 CP／M 操作系统中继承过来的，DOS 已经不用它们了。但有些字段对程序员来说非常重要。Turbo C 在头文件 stdlib.h 中已经定义了一个全局变量：

```
unsigned  __psp
```

程序员可通过它取得他的程序 PSP 的段地址。通过 Turbo C 的 getpid 和 getpsp 函数也可以取得程序 PSP 的段地址。

```
unsigned   getpid(void);
unsigned   getpsp(void);
```

getpid 函数是为了与 unix 相兼容。unix 是多任务操作系统，每个进程都有唯一的标志号 ID。在 DOS 下，PSP 的段地址可以当作进程标志使用。

PSP 中偏移量 2 处记录了该程序装入后剩下空间的起始段地址。剩下空间可能是自由的，也可能后来又被分配给其它程序了。根据这个数字以及 PSP 的起始地址，可以计算出该程序究竟占了多少空间。

PSP 中偏移量 a、e 和 12 处分别存放着 3 个远指针，它们指向 3 个特殊的处理程序：正常终止，Ctrl__Break，严重出错。进入当前程序后，当前程序对这 3 种特殊情况可能有它自己的处理程序，并且把这些处理程序的入口地址放到中断向量 22H、23H、24H 中。一旦出现这些特殊情况，DOS 就会转到这 3 个中断向量去处理。在退出当前程序时，DOS 又会把保存在 PSP 中的这 3 个远指针恢复到相应的中断向量中，从而恢复父程序对这 3 种特殊情况的处理，程序员不应该修改这 3 个远指针。

放在 PSP 中的程序终止地址应理解为：父程序告诉 DOS 在当前程序结束后应如何处理。 至于当前程序的子程序结束后应如何处理，则由当前程序自己去设置中断向量 0x22。如果当前程序从未设置 0x22，则 0x22 应等于 PSP 中的程序终止地址，即当前程序子程序的结束处理办法与当前程序本身一致。如果整个程序链谁也未曾修改过中断向量 0x22，则相当于都沿用 DOS 的程序终止处理办法。

PSP 中的偏移量 2c 处是从父程序继承的环境变量区的段地址，紧随环境变量区的是当前被执行程序的全文件名，但全文件名不是一个环境变量。

PSP 中的偏移量 55、5c、65、6c 处是文件控制块 FCB（这主要用于 DOS2.0 版以前，现在几乎没有人再用 FCB 这个概念了）。即使用 FCB，一旦程序被执行，这两个 FCB 就没有什么用了，所以 DOS 把命令行的头两个参数存放在这儿。如果命令行参数是文件名，则存放的将只是驱动器名和文件名，舍去了目录名，所以仍然用处不大。

进入程序时，从 PSP 中的偏移量 0x81 处开始，存放着命令行参数，包括与程序文件名之间的分隔符。除任何标准输入／输出重定向说明外，命令行参数的所有其它内容都存放在这儿。命令行参数的字节数存放在 PSP 的偏移量 0x80 处。因为这个区域只有 128 字节长，故限制 DOS 的命令行参数不能超过 128 字节。在 7.10 节讨论 main 函数的格式时，曾介绍过可以通过 main 函数的参数取得命令行参数，其实 Turbo C 启动程序就是从这儿取得这些值的。应该注意，如果程序需要从这个区域读取命令行参数，则应在程序一开始就读。一旦执行了任何磁盘输入输出，这个区域就可能已被 DOS 用作盘传输区 (Disk Transfer Area，DTA)。重定向是在程序被装入前，由 DOS 的 command.com 处理的，程序本身没有方便的办法可以得知重定向是否正在起作用。关于重定向的细节，可参看 10.4 节。

10.1.2 环境

7.12 节已介绍过环境变量的含义及其存取方法。现在又提出一些新的方法，归纳如下：

（1）可以通过 main 函数的参数读取环境变量。

(2) 可以通过 Turbo C 的 getenv 和 putenv 函数读取和设置环境变量。应该注意，环境变量的传送是单向的，只能由父程序传给子程序。因此，设置环境变量的目的是为了把新的环境变量传给当前程序的子程序，而不是返回给当前程序的父程序。如果不用 putenv 改写或添加环境变量，当前程序将只把父程序传过来的环境变量再传给子程序(两者已是爷爷和孙子的关系了)。

(3) Turbo C 在其头文件 dos.h 中还提供了全局变量

```
extern char *environ[ ]
```

这个全局变量是一个指针数组，存放的是指向各环境变量的指针。刚进入程序时，environ 与 main 函数的 env 相等，指向的是同一内存区。但是，一旦执行了 putenv 函数，如果原来的环境变量区放不下，则会建立一个新的环境变量区。此时，environ 将指向这个新环境变量区，而 main 函数中的 env 仍然指向原来的环境变量区，如下面的程序所示。

```
#include <general.h>
void displayenv(char *v)
{
    char *value;
    value=getenv(v);
    if(value==NULL)
        printf("String %s is not in the environment array.\n",v);
    else printf("%s=%s is in the environment array.\n",v,value);
}
void main(int param1,char *param2[ ],char *param3[ ])
{
    displayenv("B");
    displayenv("C");
    printf("  parameter3,environ=%p,%p.\n",param3,environ);
    printf("Having set string B,putenv returned %d.\n",
            putenv("B=BANANA  TREES"));
    displayenv("B");
    printf("  parameter3,environ=%p,%p.\n",param3,environ);
    printf("Having set string C,putenv returned %d.\n",
            putenv("C=COCONUT PALMS"));
    displayenv("C");
    printf("  parameter3,environ=%p,%p.\n",param3,environ);
}
```

这个演示程序产生的输出结果如下：

```
String B is not in the environment array.
String C is not in the environment array.
  parameter3,environ=0606,0606.
Having set string B,putenv returned 0.
B=BANANA  TREES is in the environment array.
  parameter3,environ=0606,081E.
Having set string C,putenv returned 0.
```

```
String C is not in the environment array.
  parameter3,environ=0606,081E.
```

(4) 如果知道了程序的 PSP 地址，则可以找到该程序的环境变量区段地址。于是，就可以读取该程序的环境变量。下面这个函数提供了根据一个程序的 PSP 段地址求得它的环境变量区段地址的方法。

```
#include <dos.h>
unsigned envseg(unsigned PSP)
{
     unsigned  *envsegptr;
     envsegptr=MK_FP(PSP,0x20);
     return  *envsegptr;
}
```

显然，如果把第 4 种方法用于求当前程序的环境变量区段地址，则是舍近求远，没有必要。这种方法的优点是可以求得任一程序的环境变量区段地址。

下面这个程序利用 envseg 函数，根据一个程序的 PSP 段地址求得指向该程序全文件名的指针。其中利用了这样一个事实：从继承的环境变量区后的第 3 个字节开始，存放的 ASCIIZ 串是该程序的全文件名（包括驱动器名）。

```
#include <dos.h>
char  *progname(unsigned PSP)
{
     char        *p;
     unsigned    i;
     p=MK_FP(envseg(PSP),0);
     for(i=0;(p[i]!=0)||(p[i+1]!=0);i++);
     return  p+i+4;
}
```

10.2 内存管理

10.2.1 内存块及其控制

DOS 是按块来管理内存的。块是内存中一个连续的存储区，大小不定，但总是 16 字节的整数倍，即是按节(1 节等于 16 字节)来分配的。每个程序至少占有两块，一块用来存放程序本身，其最前面就是 PSP，另一块用来存放环境变量。一般来说，环境变量块在前(低地址)，程序块在后(高地址)，既然两块属于同一个程序，它们的拥有者就应是同一块。

DOS 把内存块分为三类：自由块，已分配块和控制块。每块都是从节边界开始，所以在说明块的地址时只需说明它的段地址。各内存块的段地址是唯一的，故可以当作内存块的名字来使用。控制块则用于对另两类块提供一个说明，它固定为 16 字节长。每个自由块和已分配块的前面都有一个控制块。

控制块的第 1 个字节是一个标志。标志只有两种：字母 M 和字母 Z。最后那个控制

块(在最高地址)的标志为 Z，其余各控制块的标志都是 M。控制块的第 2 和第 3 字节是说明谁拥有它所控制的块，即拥有紧随该控制块之后的那个自由块或已分配块。“拥有者”一般都是程序，该程序的 PSP 所在的那一块的段地址就作为拥有者的“名字”填在第 2 和第 3 字节。控制块的第 4 和第 5 字节说明它所控制的块的大小，单位为节，不包括控制块本身所占的 1 节。控制块其余的 11 个字节暂时未使用，保留供以后扩展。

这就是说，每一个自由块和已分配块的前后都是控制块，DOS 检查控制块是否正确。如果发现第 1 个字节既不是 M 又不是 Z，DOS 就认为内存分配方案已被破坏，从而显示信息：

Memory Control Blocks Destroyed

并使系统停止工作（必须重新引导）。一旦程序往一个已分配块写的信息超过该块的长度，就会冲毁该块后面控制块信息。

当 DOS 装入一个程序时，它先分配一块给环境变量，再把全部可用的内存都分给这个程序的 PSP 块。一个写得较好的程序应该在一开始执行时就把它多余的内存释放给 DOS，否则 DOS 就不可能为当前程序再装入其它子程序。遗憾的是，所有的.COM 程序不可能控制它们的内存分配。当正在运行的程序通过 DOS 正常地终止运行时，DOS 为它分配的两个内存块就会被释放。

10.2.2 内存分布映像程序 memrymap

下面是一个演示程序 memrymap，它打印出所有内存块的信息，包括每个控制块的段地址，紧随控制块之后的内存块长度，拥有某个内存块的程序 PSP 块的地址以及程序的全文件名。

源程序 memrymap.c:

```
#include <general.h>
#include <dos.h>
#define INT20 0x20cd
unsigned envseg(unsigned PSP)
{
    unsigned *envsegptr;
    envsegptr=MK_FP(PSP,0x2C);
    return *envsegptr;
}
char *progname(unsigned PSP)
{
    char        *p;
    unsigned    i;
    p=MK_FP(envseg(PSP),0);
    for(i=0;(p[i]!=0)||(p[i+1]!=0);i++);
    return p+i+4;
}
void memrymap( ) {
    unsigned  s,goods,i,found,question;
    char      *mcb,*name;
```

```
    unsigned  * owner, * size, * ownerspsp;
    s = _psp -1;
    do {
        goods = s;
        for (i = found = 0; (s != 0) && !found; ++i) {
            mcb = (char * ) MK_FP(-s,0);
            owner = (unsigned * ) MK_FP(s,1);
            size  = (unsigned * ) MK_FP(s,3);
            found = (( * mcb == 'M') && ( * size == i));
        }
    }while (s != 0);
    printf("%s\n","      Block       owner               ");
    printf("%s\n","Address  Length    PSP   owner name");
    s = goods;
    do {
        mcb   = (    char * ) MK_FP(s,0);
        owner = (unsigned * ) MK_FP(s,1);
        size  = (unsigned * ) MK_FP(s,3);
        printf("  %04x     %04x     %04x   ",s, * size, * owner);
        question = 0;
        if ( * owner == 0) name = "free";
        else {
            if (envseg( * owner) == 0)
                name = "DOS";
            else {
                name = progname( * owner);
                if (name == EMPTYSTR)
                    name = "DOS";
            }
            ownerspsp = (unsigned * ) MK_FP( * owner,0);
            question  = ( * ownerspsp != INT20);
        }
        printf("%s",name);
        if (question) printf("%c",'?');
        printf("\n");
        s = s + * size + 1;
    }while ( * mcb == 'M');
}

void main( )
{
    unsigned a,b;
    allocmem(0x10,&a);
    allocmem(0x20,&b);
    freemem(a);
    printf("10 and 20 paragraph segments a,b=%x,%x\n"
          "were allocated and a released.\n\n",a,b);
    memrymap( );
}
```

如果只想看看为当前程序以及后续程序分配的内存块，而不想看看为当前程序之前的程序分配的内存块，则 memrymap 可以简化。这是因为，根据全局变量__psp 就可以找到当前程序的 PSP 块，再往前一节就是这个 PSP 块的控制块，然后再逐个控制块地往后移就可以了。但是，要发现所有的内存块则必须往前移。控制块的信息中，没有提供往前移的信息，只能每次往前移 1 节。先看第 1 字节是否含 M 标志字符，再看第 4 和第 5 字节中的大小是否与往前移过的节数相等。如果两个条件都符合，则认为这一节就是一个控制块。但也可能出现两个条件都符合但却不是控制块的情况，尽管发生这种事件的概率很小。这个过程不断循环，直至内存的起点。

找到了第 1 个控制块后，再从前往后移，打印出所需要的信息。

10.2.3 内存管理函数

Turbo C 中提供了三个内存管理函数。

```
int allocmem(unsigned n,unsigned * segment);
int freemem(unsigned segment);
int setblock(int segment,int n);
```

使程序员可以对内存分配过程进行某种控制。

allocmem 函数分配 n 节内存。如果成功，返回值为-1，分得的块的段地址在 * segment 中。如果不成功，返回值为最大可能的节数。

setblock 函数使已分配块的大小改为一个新值。

freemem 函数释放已分配的块。

这三个函数都通过全局变量 errno 来报告错误，errno 的说明在头文件 dos.h 中。这三个函数的说明也在头文件 dos.h 中，而 2.5 节讲的内存分配函数的说明是在头文件 alloc.h 中。应该注意，这三个函数使用的参数是已分配块的段地址，而不是指针，因此与编译模式没有关系，它们只与 DOS 发生关系。

Turbo C Tools 也增加了几个内存管理函数，但实用意义不太大，这里就不介绍了。

10.3 多个程序的执行及通信

10.3.1 程序间的通信

当在一个程序内执行另一个程序时，两个程序之间的通信就成了一个主要问题。按通信数据量的大小，可以把程序间的通信分为大通信和小通信。大通信可以通过内存缓冲区和文件进行。如果子程序的程序块与父程序的程序块不相重叠，则可以在父程序内开辟一个数组来作为通信缓冲区。父程序可以用 allomem 函数向 DOS 申请一个新内存块作为通信缓冲区。无论通过什么办法取得通信缓冲区，父程序都必须把这个缓冲区的段地址告诉子程序。

如果通过文件通信，父程序也有两种处理办法。一种是把信息写入文件内，把文件名传送给子程序。另一种是保持传递信息的文件为打开状态，把文件的文件柄遗传给子程

序。当子程序被装入执行时，它接收父程序所有文件柄的拷贝，除非父程序在打开某个文件时指定这个文件不遗传给子程序。当子程序终止时，将关闭它自己的文件柄，但父程序的文件柄仍保持为打开状态。子程序继承的只是文件柄和有关的 DOS 信息，Turbo C 的文件柄属性或流结构不遗传给子程序。

如果子程序原来是通过 DOS 的标准输入输出与外界传递信息，则父程序可以使子程序的输入输出重定向，以便父程序和子程序借助标准输入输出文件通信。后面将介绍三个这样的输入输出重定向方法。

因为通过标准的内存管理函数建立缓冲区和通过文件处理函数建立与读写文件并不困难，所以大通信剩下的问题就只归结为：输入输出重定向和小通信。小通信是为了让父程序把通信缓冲区的地址或文件柄（也许还有某些格式细节）告诉子程序。

小通信有三个渠道：

命令行参数

环境变量

ERRORLEVEL 变量

在这三个渠道中，前两个只能用于父程序向子程序传递信息。对于第 3 个渠道，main 函数返回的值只能在 DOS 的批命令文件中通过 IF ERRORLEVEL 语句取得。可以把 main 函数说明为返回一个整数 int，即可以返回任何 16 位值，但 DOS 只识别它的低 8 位。

借助命令行参数和环境变量传递信息还有一个缺点：由于它们都是 ASCIIZ 串，如果传递的是数值，则必须以某种方法进行编码，否则字节'\0'就会被解释为字符串终止符。

10.3.2 spawn：调用子进程

在一个程序的运行过程中又去装入和运行另一个子程序是一个较大型软件经常需要用到的技术。习惯上常用"spawn"（生育）来形容这个装入和执行功能。为避免把这里的"子程序"与一般编程的"子程序"相混淆，我们这里把"子程序"称为"子进程"，把"父程序"称为"父进程"。

Turbo C 中一共提供了 8 个生育函数：

```
int spawnl  (int mode,char * path,char * arg0, * arg1,..., * argn,NULL);
int spawnle (int mode,char * path,char * arg0, * arg1,..., * argn,NULL,
                      char * envp[ ]);
int spawnlp (int mode,char * path,char * arg0, * arg1,..., * argn,NULL);
int spawnlpe(int mode,char * path,char * arg0, * arg1,..., * argn,NULL,
                      char * envp[ ]);
int spawnv  (int mode,char * path,char * argv[ ]);
int spawnve (int mode,char * path,char * argv[ ],char * envp[ ]);
int spawnvp (int mode,char * path,char * argv[ ]);
int spawnvpe(int mode,char * path,char * argv[ ],char * envp[ ]);
```

这些函数的说明全部在头文件 process.h 中。为了装入并执行子进程，必须有足够的

可用内存空间。

函数第1个参数mode是方式，在Turbo C 中可以有如下三种值:

P_WAIT	父进程等待子进程完成后才继续执行
P_NOWAIT	在子进程执行过程中父进程也继续执行
P_OVERLAY	覆盖，即子进程装到了原来由父进程占据的空间，子进程执行完毕后不返回父进程。

实际上，值P_NOWAIT当前是不可用的，如果使用，则会产生错误。值P_OVERLAY虽然可以用，但如果用了，就与下一小节将要讲到的exec函数完全一样。

参数path指向被调用的子进程的文件名，文件名的查找顺序遵守下列规则:

- 若不带扩展名也不带圆点符，则按所给文件名查找。若找不到，则增加扩展名EXE再查找;
- 若带扩展名，则只按所给文件名(包括扩展名)查找;
- 若带圆点符，则只按所给文件名去查找，不添加扩展名;
- 遵守DOS的查找规则。即:若在path参数中指定了驱动器或路径名，包括全路径名(从根目录开始)和部分路径名(从当前目录开始)，则按指定路径查找。若在path参数中未指定驱动器名或路径名，则在当前目录中查找。若在指定路径或当前目录中找不到，而函数名又带后缀p，则按从父进程继承来的PATH环境变量指定的路径名查找。

spawn的8个函数是根据它们的后缀来区分的，各后缀符号的意义如下:

l: 命令行参数是以指针表的形式给出的，最后一个指针应NULL。

v: 命令行参数是以指向指针数组的指针的形式给出的。

p: 在查找子进程名时支持父进程遗传下来的PATH环境变量。

e: 表示子进程环境的来源。若有后缀e，则环境来自最后一个参数，这个参数是指向指针数组的指针，这个指针数组指向各环境变量字符串，最后一个是空字符串。 若没有后缀e (或虽有后缀e，但envp为空指针)，则继承父进程的环境。

按照二进制编码原则，8种情况只需要3个字符就可以表达。但因为字符l和v必居其一，所以要用4个字符加以区分。

应该注意，args[0]所指向的字符串与path所指向的字符串都是子进程的文件名。但实际上，args[0]可以指向任一字符串，只要不是空指针。

如果生育成功，函数返回值等于子进程内return语句的值或exit语句的状态码，这个返回值同时也设置在全局变量ERRORLEVEL中。如果不成功，返回值等于-1，全局变量errno被设置成相应的错误号。请参见有关手册中的说明。

下面的例子说明父进程如何调用子进程，父进程和子进程如何借助缓冲区通信，以及父进程如何接收子进程的返回值。为了简化起见，整个例子用的都是大编译模式。

父进程spawn程序如下:

```
#include  <general.h>
```

```
#include  <process.h>
void main( )
{
    char       * param1;
    unsigned  segment;
    char       segmenthex[5];
    char       envstring[80];
    char       * message;
    char       * child;
    int        errorlevel;
    if  (allocmem(0x10,&segment)  != -1)  {
        perror("Funmction allocmen  failed");
        return;
    }
    sprintf(segmenthex, "%04x",segment);
    param1 = "MESSAGEAT";
    strcpy(envstring,param1);
    strcat(envstring,"=");
    strcat(envstring,segmenthex);
    putenv(envstring);
    message = MK_FP(segment,0);
    strcpy(message,"child, you come back here!");
    child = "child.exe";
    printf("Parent has set               \n"
          " message              %s   \n"
          " Command line           %s    \n"
          " Environment string  %s=%s\n"
          "and will now spawn     %s \n\n",
          message,param1,param1,getenv(param1),child);
    errorlevel = spawnle(P_WAIT,child,child,param1,NULL,environ);
    printf("The parent has regained control, and sees\n"
          " message              %s\n"
          " errorlevel          %i\n",
           message,errorlevel);
    getch( );
}
```

父进程首先通过函数 allocmem 分配一块缓冲区，通过函数 sprintf 把返回的段地址值由整型转换为 4 个十六进制字符，再把这个地址字符串以环境变量值的形式存入环境变量区，其变量名为“MESSAGEAT”。接着，往缓冲区内写入信息“Child, you come back here”。最后调用子进程 child，传送环境变量，同时把环境变量“MESSAGEAT”作为第 1 个参数传给子进程。从子进程返回后，再次打印缓冲区中的内容，但这时是由子进程传回信息“Coming, Mother!”，同时打印出返回的全局变量 ERRORLEVEL 的值。

子进程 child 程序如下：

```
#include <general.h>
int main(int paramcount,char * param[ ])
```

```
{
    char      * program;
    char      * segmenthex;
    unsigned segment;
    char      * message;
    int       errorlevel = 12;
    program    = param[0];
    segmenthex = getenv(param[1]);
    sscanf(segmenthex,"%x",&segment);
    message    = MK_FP(segment,0);
    printf("Her child  %s  has seen       \n"
           " Command line        %s   \n"
           " Environment string  %s=%s\n"
           " Message             %s   \n",
           program,param[1],param[1],segmenthex,message);
    strcpy(message,"Coming, Mother!");
    printf("and has now set               \n"
           " Message             %s   \n"
           " ErrorLevel          %i \n\n",
           message,errorlevel);
    return errorlevel;
}
```

子进程把参数 1 当作一个环境变量名，再通过函数 getenv 从父进程传过来的环境中找到这个变量的值。根据父子进程间的约定，这个值是缓冲区的段地址字符串。接着，通过函数 sscanf 把这个十六进制字符串变为整型的段地址值。找到缓冲区的段地址，就可以把缓冲区的信息读出来。最后，往缓冲区写返回给父进程的信息，返回到父进程，返回值设置为 12。

执行这两个进程后，其输出结果可能如下：

```
Parent has set
  message             child, you come back here!
  Command line         MESSAGEAT
  Environment string  MESSAGEAT=0ea4
and will now spawn     child.exe

Her child  C:\WB\A\CHILD.EXE  has seen
  Command line         MESSAGEAT
  Environment string  MESSAGEAT=0ea4
  Message             child, you come back here!
and has now set
  Message             Coming, Mother!
  ErrorLevel          12

The parent has regained control, and sees
  message             Coming, Mother!
  errorlevel          12
```

10.3.3 exec:转到子进程

如果用 spawn 函数来调用一个子进程，并且其方式选为 P_WAIT，则在子进程执行过程中，父进程仍驻留在内存，以便从子进程返回后，父进程接着执行。如果子进程执行完后，不需要或不希望返回父进程，则应该用 exec 函数来代替 spawn 函数，或仍然用 spawn 函数，但方式 mode 选为 P_OVERLAY。在这种情况下，子进程被装在父进程原来占据的内存块(就从 PSP 后开始)。如果原来的块不够大，还会自动扩展，只要有可用的内存空间。控制从父进程转到子进程是通过跳转指令进行的。跳转指令本身在父进程内，在装入子进程时， 这条指令本身也会被冲掉，所以 exec 函数必须先拷贝父进程的某一部分到一个安全区，跳到这个安全区，再从此地装入子进程，最后跳到子进程。子进程继承父进程的 PSP 时，做了某些修改，以反映新的命令行参数和新的环境变量。

Turbo C 一共提供了 8 个 exec 函数，与 8 个 spawn 函数一一对应，只是没有第 1 个参数 mode。

```
int execl  (char *path,char *arg0,*arg1,...,*argn,NULL);
int execle (char *path,char *arg0,*arg1,...,*argn,NULL,char **env);
int execlp (char *path,char *arg0,*arg1,...,*argn,NULL);
int execlpe(char *path,char *arg0,*arg1,...,*argn,NULL,char **env);
int execv  (char *path,char *argv[ ]);
int execve (char *path,char *argv[ ],char **env);
int execvp (char *path,char *argv[ ]);
int execvpe(char *path,char *argv[ ],char **env);
```

10.3.4 system: 执行 DOS 命令

spawn 和 exec 函数不能用来在一个程序内执行 DOS 的内部命令，也不能用来执行批命令文件。Turbo C 提供的另一函数 system 可以弥补这一缺陷:

```
int system(const char *command);
```

这个函数调用 DOS 的 COMMAND.COM(根据环境变量 COMSPEC 找到的)文件去执行一条 DOS 命令，或一个批命令文件，或由参数指定的其它可执行文件。被执行的程序文件必须是当前目录下或 PATH 环境变量所指明的各目录路径中。如:

```
vord main( ) {system("chr d:")}
vord main( ) {system("chkdsk d:")}
vord main( ) {system("command")}
```

最后这个例子执行后将暂时离开当前程序，回到 DOS 提示符下：用户可以执行任何 DOS 命令，直至键入 exit 命令，才返回到当前程序。

通过函数 system 执行内部的 DOS 命令是一个递归调用过程。用户当前可能正在 DOS 提示符下执行一个父进程，如果在父进程内调用函数 system，则可以把所要执行的命令直接传给命令处理程序(一般是 command.com)，而且这个命令处理程序已经在内存。但是，command.com 是不可重入的，它不能递归调用，因此 system 必须装入第 2 份 command.com，由第 2 份 command.com 去处理所要执行的命令。由此可以看出，sys-

tem 函数要求较多的内存。

10.3.5 signal 和 raise 事件处理

signal 字面含义为"信号"，但它在本节的含义更类似于事件。每当某一事件发生时，就向操作系统发出一个警告。Turbo C 提供了一个 signal 函数，让程序员说明在发生各种事件时分别调用什么函数去处理。函数 signal 的说明在头文件 signal.h 中，其一般使用格式是：

```
signal(int__sig, void( * func)(int__sig[int subcode]));
```

格式中，sig 是事件名，func 是指向用户自己的处理该事件函数名字的指针。Turbo C 中预先定义了两个事件处理函数，它们的函数指针及其处理方法如下：

SIG__DFL: 终止当前进程，并将控制权交回MS__DOS。关闭所有由进程打开的文件。

SIG__IGN: 忽略该事件。

在 Turbo C 中，目前可以接受的事件(即 signal 函数的参数 sig 可以取的值) 以及对这些事件的缺省处理办法如下：

SIGABRT: 由于程序非正常终止而产生，缺省的处理办法等效于调用 __exit(3)。

SIGFPE: 由于某些浮点运算出错而引起，如上溢，除数是0，非法操作等。缺省的处理办法等效于调用__exit(1)。

SIGILL: 由于某些非法操作而引起，缺省的处理办法等效于调用 __exit(1)。

SIGINT: 由于 Ctrl__C 键而引起，缺省的处理办法等效于执行 INT 23H。

SIGSEGV: 由于非法存储器存取而引起，缺省的处理办法等效于调用 __exit(1)。

SIGTERM: 由于请求程序终止而引起，缺省的处理办法等效于调用 __exit(1)。

当因外部客观原因或因 raise 函数(下面会讲到)而产生事件时，则按如下顺序处理：如果用户已安装了相应于这个事件的自定义处理函数，则调用这个自定义处理程序，并把所发生的事件传给这个函数。从处理函数返回后，原来被事件中断的进程将继续执行。在调用自定义处理函数前，对同一事件的处理函数指针被修改成 SIG__DFL。因此，下一个这种事件将按 SIG__DFL 进行处理，除非在下一个这种事件发生前，再一次调用了 signal 函数重新设置该事件的用户自定义处理函数。

如果调用成功，signal 函数返回一个指针，指向上一次为该事件设置的处理函数。如果调用不成功，signal 函数的返回值为-1，且置全局变量 errno 的值为 EINVAL。

下面这个例子定义了一个函数 ctrl__break，主函数 main 把这个函数安装为事件 SIGINT 的处理函数。这样，每当按下 Ctrl__Break 键，就进入 ctrl__break 函数，这个函数重新设置事件 SIGINT 的处理函数指针为 SIG__IGN，以便在处理期间忽略 Ctrl__Break 事件。处理很简单，前 4 次只简单地做计数和判断，然后返回，返回前又恢

复设置事件 SIGINT 的处理函数为 ctrl__berak。第 5 次键入 Ctrl__Break 后，程序结束运行。

```
#include <signal.h>
#include <stdio.h>
#include <process.h>
main ( )
{
      int ctrl__break( );
      signal(SIGINT,ctrl__break);
      printf("Press Ctrl__Break 5 times to terminate program\n");
      getchar( );
}
ctrl__break( )
{
      static int count = 1;
      signal(SIGINT,SIG__IGN);
      if(count++ = = 5) {
            printf("Program terminated by Ctrl__Break\n");
            exit( );
      }
      signal(SIGINT,ctrl__break);
}
```

通常用户编写的事件处理函数难以调试，原因之一就是怎样人为地产生这样一些事件。Turbo C 的 raise 函数可以帮助解决这方面的问题：

```
int  raise (int sig);
```

执行函数 raise 后，就会产生相应的事件。如果成功，则返回值为 0，否则返回值为非 0。

```
#include <signal.h>
#include <stdio.h>
main( )
{
      int abort__handler( );
      signal(SIGABRT,abort__handler);
      raise(SIGABRT);
}
abort__handler( )
{
      signal(SIGABRT,SIG__IGN);
      printf("In abort handler - closing all open files\n");
      fcloseall( );
}
```

通过这种方法使用 raise 函数，就可以测试用户自定义的事件处理函数，看看它是否能 响应所发生的事件。调试完毕后，就可以用真正的可能产生这类事件的程序代替 raise 语句。

10.4 标准输入 / 输出重定向

如果子进程的输入输出是通过标准的输入输出文件柄进行的，则父进程与子进程之间的通信就变得很简单。此时，父进程只要把欲传给子进程的信息存入一个文件内，并把这个文件重定向为标准输入文件(通过 freopen 或 dup 函数)，则子进程从标准输入文件取信息时就取到了父进程传过来的信息。同理，父进程只要建立一个用于存放子进程返回信息的文件，把这个文件重定向为标准输出文件，则子进程往标准输出文件写信息时，就写到了父进程中的文件中。

在具体介绍标准输入输出重定向的例子之前，先简单地回顾一下 Turbo C 函数 freopen 和 dup。

```
FILE  * freopen(const char  * filename,const char  * mode,FILE  * stream);
int  dup(int handle);
int  dup2(int oldhandle,int newhandle);
```

函数 freopen 关闭 stream 当前代表的文件，而把 stream 重新赋予由 filename 指定的文件。该函数一般用于把预先打开的文件 stdin、stdout、stderr、stdaux 和 stdprn 重新定义为用户指定的文件。打开的方式 mode 有 r，w，a，r+，w+，a+。值得注意的是，当打开的方式为“r+”、“w+”和“a+”时，虽然读和写文件都允许，但由读操作转为写操作时、或由写操作转为读操作时，必须先执行 fseek 或 rewind 操作。

函数 dup 建立并返回一个新的文件柄，这个新文件柄与老文件柄 handle 在下列三点上完全一样：

· 同一个打开文件或设备；

· 同样的文件指针；

· 同样的文件存取方式。

这就是说，dup 的两个文件柄实际上是同一个文件，只不过用了两个不同的文件柄，即两个不同的名字而已。这一点与 freopen 很不一样，它不是为同一个文件增加一个名字，而是关闭了老文件，把老文件的名字用于一个新文件。

函数 dup2 的功能基本上与 dup 相同，只是它强制新的文件柄一定是 newhandle。如果这个文件柄原已赋予另一个已打开的文件，则先关闭那个文件。

我们还需要回顾一下 mktemp 函数：

```
char  * mktemp(char  * template);
```

这个函数创建一个临时文件，文件的名字是从参数 template 指向的字符串修改而来的。在入口时，这个字符串应该是路径名再加 6 个尾随的字母"X"。新创建的文件名的路径名部分将取自 template 所指向的字符串的路径名，而文件名(两个字母)和扩展名(三个字母)则由系统指定，以保证不会与同一子目录下已建立的其它文件名重复。

10.4.1 [例 1]: freopen.dem

这个例子是说明如何通过函数 freopen 进行输入输出的重定向。程序建立了临时文件 temp1 和 temp2。temp1 用于向子进程写 5 个随机数，然后被重定义为 stdin，使子进程可以通过标准输入设备读到这个文件。temp2 用于接收子进程返回的信息,被重定义为 stdout，使子进程可以通过标准输出设备写到这个文件。由于采取了输入输出重定向的办法，故这个例子无需小通信。父进程是通过 spawnlp 函数调用子进程的。

子进程就是 DOS 的外部命令 SORT，它对父进程传送来的 5 个随机数进行排序，把排序后的结果写到标准输出文件。

从子进程返回父进程后，父进程再通过 freopen 函数使 stdin 和 stdout 恢复为标准输入输出设备，即控制台 CON，然后把排序前后的两组数打印出来。

```
/ * freopen.dem * /
#include <general.h>
#include <process.h>
#define N       5
#define TEMPLATE "d:\XXXXXX"
void main( )
{
    char  temp1[15],temp2[15];
    FILE  *f;
    int   i,x[N],y;
    strcpy(temp1,TEMPLATE);
    mktemp(temp1);
    f = fopen (temp1,"a");
    strcpy(temp2,TEMPLATE);
    mktemp(temp2);
    for (i = 0; i <N; ++i)
        fprintf(f,"%5d\n",x[i] = rand( ));
    fclose(f);
    freopen(temp1,"r",stdin);
    freopen(temp2,"w",stdout);
    spawnlp(P_WAIT,"Sort","Sort",NULL);
    freopen("CON","a",stdout);
    freopen("CON","r",stdin);
    printf("Executed  Sort  with  i/o        \n");
    printf("redirected to temporary files  \n\n");
    printf("Random  sorted    \n");
    printf("Array    Array  \n\n");
    f = fopen(temp2,"r");
    for (i = 0; i < N; ++i) {
        fscanf(f,"%d",&y);
        printf("%5d      %5d\n",x[i],y);
    }
    fclose(f);
    remove(temp1);
    remove(temp2);
```

```
}
```

打印出的结果可能如下所示:

```
Executed  Sort  with  i\o
redirectioned  to  temporary  files

Random    Sorted
Array     array

  346       130
  130       346
10982      1090
 1090     10982
11656     11656
```

注意,程序 freopen.dem 在运行时只接收标准输入输出,不能被重定向。从子进程返回后是通过语句

```
freopen("CON","a",stdout);
freopen("CON","r",stdin);
```

使控制台再度成为标准输入输出设备的。函数 freopen 需要知道文件名(本例中是"CON")。如果 freopen.dem 本身的输入输出被重定向,它就无法知道自己的输入文件名和输出文件名,因此,在执行语句

```
freopen(temp1,"r",stdin);
freopen(temp2,"w",stdout);
```

后,再也无法使 stdin 和 stdout 与它自己的输入文件和输出文件联系起来。

10.4.2 [例 2]:dup.dem

这个例子与[例 1]的功能完全相同,产生的输出也一样。只是实现的方法不同,本例不用 freopen 函数,而用 dup 函数。

```
/*  dup.dem  */
#include <general.h>
#include <io.h>
#include <process.h>
#define  STDIN    0
#define  STDOUT   1
#define  N        5
#define  TEMPLATE "d:\XXXXXX"
void main( )
{
    char   temp1[15],temp2[15];
    FILE   *f1,*f2;
    int    i,x[N],y;
```

```
    int     handle1,oldstdin;
    int     handle2,oldstdout;
    strcpy(temp1,TEMPLATE);
    mktemp(temp1);
    f1 = fopen(temp1,"w+");
    strcpy(temp2,TEMPLATE);
    mktemp(temp2);
    f2 = fopen(temp2,"w+");
    for (i = 0; i < N; ++i)
        fprintf(f1,"%5d\n",x[i] =rand( ));
    rewind(f1);
    handle1    = fileno(f1);
    oldstdin   = dup(STDIN);
    dup2(handle1,STDIN);
    handle2    =fileno(f2);
    oldstdout = dup(STDOUT);
    dup2(handle2,STDOUT);
    spawnlp(P_WAIT,"Sort","Sort",NULL);
    dup2(oldstdin ,STDIN);
    dup2(oldstdout,STDOUT);
    printf("Executed  Sort  with  i/o        \n"
          "redirected to temporary files  \n\n");
    printf("Random  Sorted     \n");
    printf("Array     Array  \n\n");
    rewind(f2);
    for (i = 0; i < N; ++i) {
        fscanf(f2,"%d",&y);
        printf("%5d     %5d\n",x[i],y);
    }
    fclose(f1);
    remove(temp1);
    fclose(f2);
    remove(temp2);
}
```

与[例 1]相比，它有以下 3 点不同:

(1) 它通过函数 fileno 把流文件变成文件柄，因为下面的函数 dup2 需要这个文件柄。如果开始时不是用 stdio.h 中的 fopen 而是用低级的 io.h 中的函数来建立文件，则在建立文件时得到的就是文件柄，这一步也就不需要了。

(2) 在重定向之前执行了 rewind(f1)语句，这是绝对必要的。因为子进程从父进程继承的不仅是已打开文件的状态，还继承了文件的指针和文件的存取方式。如果不执行 rewind，子进程就将存取文件尾而不是文件头。另外，在父进程中对文件 f1 是写，而在子进程中对文件 f1 是读，根据 Turbo C 的要求，在由写转为读或由读转为写时，都必须执行 rewind 或 fseek 函数。同理，从子进程返回后，也必须首先执行 rewind(f2)语句，才对文件 f2 进行存取。

(3) 由于 dup2 函数所要求的输入参数是文件柄而不是文件名，故从子进程返回后，可用 dup2 函数恢复 stdin 和 stdout 与程序自己的输入输出文件的联系。也就是说，程序

dup.dem 本身也可以重定向输入输出。

10.4.3 [例 3]: 利用 system

Turbo C 的 system 函数提供了在程序内执行任一 DOS 内部命令或外部命令或批命令文件的办法。因此，如果用语句

```
strcpy(commsnd,"sort <");
strcat(command,temp1);
strcat(command," >");
strcat(command,temp2);
system(command);
```

代替[例 1]中 freopen.dem 的语句

```
freopen(temp1,"r",stdin);
freopen(temp2,"w",stdout);
spawnlp(P_WAIT,"Sort","Sort",NULL);
freopen("CON","a",stdout);
freopen("CON","r",stdin);
```

则程序执行的效果将完全一样。修改后的 5 条语句实际上是构成了这样一条 DOS 命令:

```
sort <d:\AA.AAA>d:AA.AAB
```

10.5 程序的终止

终止一个程序(子进程)运行的方法共有 7 种: 通过无 return 语句的 main 函数返回，通过有 return 语句的 main 函数返回，通过__exit 函数，通过 exit 函数，通过 abort 函数，通过 assert 函数，通过 keep 函数。这 7 种办法最终都要调用__exit 函数。这些函数的格式是:

```
void __exit(int status);
void exit(int status);
void abort(void);
void assert(int test);
void keep(unsigned char status,unsigned size);
```

(1) 如果子进程最终是通过 main 函数正常终止的，而 main 函数内又无 return 语句，则相当于通过语句

```
return 0
```

终止。

(2) 如果子进程最终是通过 main 函数正常终止的，且在 main 函数内有 return 语句，则相当于调用了函数

```
__exit(返回值)
```

这个返回值不应超过两个字节，因为__exit只传递两个字节。可以通过DOS的全局变量ERRORLEVEL取到这个返回值。

(3) 当调用到__exit函数时，__exit先恢复中断向量0到子进程被装入前的值，因为Turbo C的启动码在装入子进程时已使中断向量0指向浮点运算错处理程序。接着，__exit对数据段起始处的Turbo C拷贝权信息进行检查(通过检查和)。如果这个信息改变了，则可能是通过一个空指针进行了无效的赋值，于是通过标准错误输出设备(一般是显示器)显示出如下信息:

Null pointer assignment

最后，__exit返回到DOS的程序执行服务，由DOS关闭程序的所有已打开文件，从程序的PSP中把程序终止地址(这是由父进程传下来的)拷贝到中断向量0x22处，释放程序占用的所有内存，通过中断向量0x22返回到父进程。返回码被DOS赋给DOS的全局变量ERRORLEVEL。

Turbo C的用户手册中指出，__exit函数不关闭文件，但实际上并不如此。虽然在__exit内部没有关闭文件操作，但退出到DOS时，DOS会关闭所有已打开的文件。

(4) 如果通过exit函数终止，它只是先依次调用预先安装好的各个退出函数，然后调用__exit，并把退出码传给__exit，以后的过程就与__exit相同了。

退出函数是预先通过atexit函数安装的:

```
int atexit(atexit__t func);
```

atexit函数把由func指向的函数登记为一个退出函数。一旦程序通过exit终止，exit就会自动调用这些退出函数。退出函数所接受参数的类型是atexit__t，其定义在头文件stdlib.b中，如下所示:

```
typedef void __Cdecl (* atexit__t)(void);
```

每调用一次atexit，就登记一个退出函数，最多可以登记32个。没有任何办法可以把已登记的退出函数从登记表中删除。一旦执行exit函数，这些已登记的退出函数将按先进后出的原则被一一执行,如下例所示:

```
#include <stdio.h>
#include <stdio.h>
atexit__t exit__fn1(void)
{
     printf("Exit function 1 called\n");
}
atexit__t exit__fn2(void)
{
     printf("Exit function 2 called\n");
}
main( )
{
      atexit(exit__fn1);
      atexit(exit__fn2);
      printf("Main quitting ...\n");
}
```

程序的输出结果可能如下所示:

```
Main quitting ...
Exit function 2 called
Exit function 1 called
```

(5) 如果通过 abort 函数终止程序的运行，则意味着异常终止。abort 首先通过标准错误输出设备输出信息:

```
Abnormal program termination
```

然后调用_exit(3)。函数_exit 和 exit 都是用退出码 0 表示成功结束后的正常终止，用退出码 3 表示失败后的异常终止。

(6) 当通过 assert 函数终止时，如果其参数 test 的值为 0，则首先通过标准错误输出设备输出下列信息，然后调用 abort 函数退出。如果 test 的值不为 0，则什么也不做，也不退出。

输出信息的格式如下:

```
Assertion failed: <test>, file <filename>, line <linenum>
```

其中，filename 是源文件的名字，linenum 是函数 assert 所在的行号。这个函数主要用于调试源程序。如果源文件中在

```
#include     <assent.h>
```

语句之前有一条语句:

```
#define     NDEBUG
```

则 assert 函数将不起作用，只相当于在源文件中留下一个注解。

例如:

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
struct ITEM {
    int key;
    int value;
}
main( )
{
    additem(NULL);
}
void additem(struct ITEM *itemptr)
{
    assert(itemptr! = NULL);
    / * ...add the item ... * /
}
```

假设这个源文件的名字是 assertst.c，且在目录 C:\turboc 下，则可能输出信息如下:

```
Assertion failed: itemptr != NULL,
```

```
file C:\TURBOC\ASSERTST.C, line 12
```

(7) 函数 keep 退出当前程序，返回到 DOS，但当前程序仍驻留在内存。status 是退出码，size 是仍驻留在内存的长度，从程序的起点开始计算，单位是节。所有其余的内存都被释放。keep 函数显然是编写常驻内存程序 TSR 用的，但编写 TSR 牵涉到许多其它复杂的技术问题，这留待中断处理程序一章再详细讨论。

除这些专用于终止程序的函数外，Turbo C 还提供了一些与终止程序功能有关的函数，10.3.5 中讲过的 signal 函数即是其中一例。通过 signal 函数可以设置 Ctrl_Break 键的处理程序。Turbo C 中还提供了另一个专用函数来做这件事:

```
void ctrlbrk(int(*handler)(void));
```

函数 ctrlbrk 设置中断向量 0x23，使它指向中断处理程序 handler。handler 不接收任何参数，可以执行任何操作和发出任何系统调用。handler 本身可以有三种退出办法。第 1 种是 return 0，这将流产当前程序。第 2 种是返回任何其它值，这会回到调用 ctrlbrk 函数的那个位置。第 3 种是调用 longjmp 函数，跳转到(注意不是返回)程序的 setjmp 函数设置的任一点。

例如:

```
#include <stdio.h>
#include <dos.h>
#define  ABORT 0
int c_break(void)
{
      printf("Control-Break hit,
               Program aborting ...\n");
      return(ABORT);
}
main( )
{
      ctrlbrk(c_break);
      /* infinite loop */
      for (;;) printf("Looping ...\n");
}
```

程序的输出结果可能如下所示:

```
Looping ...
Looping ...
Looping ...
^C
Control-Break hit, Program aborting ...
```

一般来说，在一个函数内只能调用另一个函数，不能跳转到另一个函数，更不能跳转到另一个函数内部的某处。但是，Turbo C 的一对 setjmp 和 longjmp 函数却提供了从一个函数内跳转到另一个函数内某一点的办法:

```
int  setjmp(jmp_buf jmpb);
void longjmp(jmp_buf jmpb,int retval);
```

参数 jmpb 实际上是一个缓冲区，执行 setjmp 函数，当前的许多现场信息,包括寄存器 CS、DS、ES、SS、SI、DI、SP、BP 和 FLAGS(标志)，都被保存到这个缓冲区内，以后当通过 longjmp 跳转到这点时，便可以从缓冲区内取得这些保留值，恢复现场，接着往下执行。我们不妨称这样的点为保留点，通过 setjmp 函数可以设置多个保留点，调用 setjmp 时的实参 jmpb 可以当作各个保留点的“名字”看待，以便彼此区分。

第 1 次执行 setjmp 时，返回值一定为 0。以后当由 longjmp 跳转到这点时，setjmp 的返回值则是 longjmp 带过来的，即调用 lonjmp 时的实际参数 retval。retval 的值不应该为 0，如果是 0，则被强行改置为 1。

setjmp 和 longjmp 可能分别处于不同的函数体内，应该注意，当执行 longjmp 时，相应的 setjmp 所在的函数体应为活动的。如果 setjmp 在 main 函数内，则肯定会满足这个要求。下面是应用这两个函数的例子。

```
#include <stdio.h>
#include <setjmp.h>
jmp_buf jumper;
main( )
{
     int value;
     value=setjmp(jumper);
     if(value != 0) {
          printf("Longjmp with value %d\n",value);
          exit(value);
     }
     printf("Abort to call subroutine ... \n");
     subroutine( );
}
subroutine( )
{
     longjmp(jumper,1);
}
```

这个例子产生的输出结果如下所示:

```
About to call subroutine ...
Longjmp with value 1
```

第 11 章　MS C 6.0 的基指针技术

Microsoft C 6.0 版中增加了一种新的指针类型：基(based)指针。基指针综合了近指针和远指针的优点，它在内存中只占两个字节，但却可以寻址到内存中的任一单元。本章在介绍了各种基指针后，提供了一个把基指针应用于链表管理的工具包和两个应用例子。这一章的程序只能用 MSC 6.0 编辑。

11.1　六种基(Based)指针

近指针的段地址隐含地取自程序的数据段(DS 寄存器)，远指针和巨指针的段地址取自于指针本身，在 MSC 中，基指针的段地址，可以有 6 个不同的来源，构成 6 种不同的基指针。

11.1.1 变量值基指针

变量值基指针是以某一个变量的值作为段地址的。假设想以 EGA 或 VGA 的显示缓存的段地址作为基指针的段基址，则应像下面这样书写程序：

```
__segment videoseg=0xb800;
unsigned __based(videoseg) *bpvid=0;
```

这两条语句做如下几件事。第 1，它建立数据类型为__segment 的变量 videoseg。__segment 是 Microsoft C 6.0 新增的数据类型。它类似于整型变量，用来存放一个段地址，供别的指针作为段基址。第 2，建立一个指针 bpvid，关键字__based 说明了这个指针是基指针，它的偏移量被初始化为 0，其段基址存放在 videoseg 中。关键字__based 后面必须跟一个被圆括号括起来的有效基地址表达式，但这个表达式不可以是如下的无符号整常数：

```
unsigned __based((__segment)0xb800) *bpvid;
```

为了对比，下面也列出用远指针寻址同一单元的写法：

```
unsigned __far *fpvid=(unsigned __far *)0xb8000000;
```

虽然基指针 bpvid 和远指针 fpvid 指向的是同一个单元，但它们却有些不一样。假设从地址 A000:0000，B000:0000 等处开始，也有一个与从地址 B800:0000 开始的同样的显示缓存结构。为了在这些段也使用指针 fpvid，则必须在每次使用时重新设置整个指针，至少必须通过宏 FP__SEG 重新设置指针的段地址部分。但是，为了在这些段也使用指针 bpvid，则只需改变变量__videoseg 的值即可。基指针所带来的好处是:若某些数据在内存

中有两份或多份，彼此之间又不连续甚至相距很远，则可以相对于某一个段地址建立起一系列基指针，分别去存取某个数据结构内的各个成员。在存取另一数据结构内的各成员时，只需修改段基址就即可，而不需重新建立一系列基指针。在下面这个例子里，pspptr是指向程序段前缀(PSP)的基指针，初始化为0，它的段地址基于变量psp中的值:

```
#include <dos.h>
__segment psp;
typedef struct __psp
{
      unsigned              int20;
      unsigned              allocblockseg;
      char                reserved0;
      unsigned char       dosfuncdispatch[5];
      unsigned long       int22;
      unsigned long       int23;
      unsigned long       int24;
      char                reserved1[22];
      __segment              envseg;
}PSP
PSP  __based(psp)  * pspptr=0;
```

如果某个应用程序知道了另一应用程序的PSP，它就可以把另一应用程序的段地址赋给变量psp，从而可以存取另一应用程序程序段前缀中的任一成员。为了说明这个过程，请看下面这个例子:

```
/ * m environ.c  * /
#include  <stdio.h>
#include  <dos.h>
#include  <stdlib.h>
#include  <string.h>
__segment psp;
typedef struct __psp
{
      unsigned              int20;
      unsigned              allocblockseg;
      char                reserved0;
      unsigned char       dosfuncdispatch[5];
      unsigned long       int22;
      unsigned long       int23;
      unsigned long       int24;
      char                reserved1[22];
      __segment              envseg;
}PSP
PSP  __based(psp)  * pspptr=0;
void main(void)
{
      psp=__psp;
      printf("Int22 is set to %lp\n
```

```
            Int23 is set to %lp\n
            Int24 is set to %lp\n",
            (void far *)pspptr->int22,
            (void far *)pspptr->int23,
            (void far *)pspptr->int24);
    {
        __segment envseg=pspptr->envseg;
        char __based((__segment)envseg) * envptr=0;
        printf("pspptr->envseg=%04x\n",pspptr->envseg);
        printf("Environment:\n");
        for( ;*((char far *)envptr);
                    envptr+=__fstrlen((char far *)envptr)+1)
            printf("%Fs\n",(char far *)envptr);
    }
}
```

假设这个程序的名字为 ENVIRON.C，则编译这个程序的命令行应该是：

```
cl /Lr -WX -Od -Zpiel environ.c /link /co
```

这个程序建立了一个__segment 型变量 psp，定义了一反映程序段前缀结构的变量类型 PSP，以及一个指向 PSP 型变量的基指针 pspptr。在主程序中，通过一个 psp 取得程序本身的程序段前缀的段地址，并把这个地址作为指针 pspptr 的段基址。然后，通过基指针 pspptr 就可以取到程序段前缀的中断 0X22、0X23、0X24 的中断向量，以及本程序的环境段的段地址。这个段地址又进一步作为段基址赋给另一个基指针 envptr，这个基指针也被初始化为 0，是指向字符型变量的。最后，通过基指针 envptr 打印出环境段的各字符串。

11.1.2 变量地址基指针

变量地址基指针是以某个数据目标所在段的段地址作为基指针的段基址，具体做法是通过__segment 目标的段地址强制转换为段基址。在下面这个小例子里，基指针 cbpsv 的段基址是变量 count 所在段的段地址。基指针 cbpsp 的段基址是包含 cp 的段的段地址。

```
unsigned count;
char *cp;
char __based((__segment)&count) *cbpsv;
char __based((__segment)cp) *cbpsp;
```

注意，__segment 在这里意指“某某所在的段”，这与下面将要讨论的“指针基指针”不是一回事。

11.1.3 不定基指针

不定基指针的段基址是不定的。

在说明一个基指针时，并不总是需要包含基指针的段地址。例如，在说明

```
unsigned __based(void) *ubpv=0;
```

中，基指针 ubpv 就不包含段地址。但基指针总应有某个段基址，所以在引用一个不定基指针时，必须说明段地址值:

```
__segment videoseg=0xb800;
unsigned __based(void) * ubpv=0;
 * (videoseg:>ubpv)=(0x70<<8)+'H';
```

这个小例子用 videoseg 的值来提供 ubpv 的段地址，它在 EGA 或 VGA 屏幕的左上角反显一个字母"H"。程序中用到了一个新的基操作符":>"，它的操作是把存在 videoseg 中的段地址和存在 ubpv 中的偏移量综合起来。但是，基操作符只能用于不定基指针。如果用于其它基指针，则编译程序会报错。

不定基指针与变量值基指针的实际应用场合大致相似。 但不定基指针不需改变 __segment 型变量，只需在每次使用时临时指定一个参考段地址值。

11.1.4 段名基指针

段名基指针是以某个段名所代表的段的段地址作为段基址。

在编写大中型应用软件时，程序员往往为每一个程序模块指定了一个名字，在同一模块内使用近调用和近数据。虽然这是很实用的技术，但在一个模块内要参照另一模块命名的各段几乎是不可能的。这样一些模块一般是在编译时通过命令行开关来指定名字。

在下面这个例子里，建立了基指针 vbpc 和 vbpd，它们的段基址分别是该程序的码段段地址和数据段段地址。

```
void __based(__segment("__CODE")) * vbpc;
void __based(__segment("__DATA")) * vbpd;
```

vbpd 实际上没有必要，因为它等效于下面这样一个近指针:

```
void     * vptr;
```

一般情况下，把段基址设置在程序员所指定的有名段上可能相当有用。__segment 操作符后面必须跟一个圆括号括起来的带引号的段名字符串。这个段名可以是 Microsoft C 预先定义的一些段名，如__CODE、__DATA、__CONST 或__STACK，也可以是用户定义的在编译时生成的一些其它段名。在下面这个例子里，段基址就是建筑在一个新段 __NEWSEG 上的:

```
void __based(__segment("__NEWSEG")) * vbpm;
```

编译程序在遇到这一行时，就将建立一个新段。但是，如果这个新段不包含任何内容，把基指针建立在这样一个段上也就没有任何意义。所以，编译程序也允许程序员往这样一个新段上填写内容。例如，下面这条语句就将在__NEWSEG 段内存放一个字符串数组，并通过 newseg__message 存取这个数组:

```
char __based(__segment("__NEWSEG"))newseg__message[ ]=
        "This string is stored in the segment: __NEWSEG";
```

应该注意，如果 newseg__message 不是一个数组而是一个基指针，则上面这条语句应修改成如下形式：

```
char __based(__segment("__NEWSEG")) * newseg__message=
        (char __based(__segment("NEWSEG")) * )
        "This string is stored in the segment: __NEWSEG";
```

这里进行了类型的强制转换。

在小编译模式下，如果程序不足 64K 字节，而数据又略微超过 64K 字节，则可以通过这种基指针把一部分数据放到程序段中去，从而避免升级到大数据编译模式。下面这条语句就把一个字符串放到__CODE 段数组 codeseg__message 中去了。

```
char __based(__segment("__CODE")) codeseg__message[ ]=
        "This string is stored in the __CODE segment";
```

应该记住，只要是从别的段而不是从程序的隐含数据段存取数据，都可能引起重新加载段寄存器的操作，都会降低速度，因此应尽量避免使用这种方法。

11.1.5 指针基指针

指针基指针是以某个指针所指向的地址作为基指针的段基址。例如：

```
unsigned __near  * ip;
unsigned __far   * fip;
unsigned __based(ip)        * ibpip;
unsigned __based(fip)       * ibpfip;
```

这里定义了两个普通指针和两个基指针，每个基指针的段基址都来自一个普通指针所指向的地址。因此，如果远指针 fip 指向 B800:0000，而基指针 ibpfip 被初始化为 5，则 ibpfip 将指向 B800:0005。对远指针 fip 的任何改变都会影响到 ibpfip，ibpfip 将总是被设置为它的值与 fip 的地址相加后所产生的地址。

11.1.6 自参照基指针

所谓自参照基指针，是说基指针的段基址就是这个基指针所在段的段地址。从这个意义上讲，近指针就是一种基指针。但是，在一个远段内不可以使用近指针，远指针也不是基于它所存放的段的段地址的。例如，考虑下面这样一个链表程序：

```
typedef struct __list LIST;
struct __list {
        void  * item;
        LIST __based((__segment)__self)  * left;
        LIST __based((__segment)__self)  * right;
};
void main(void)
```

```
{
        LIST __based(__segment("LISTSEG")) list;
        ..........
}
```

程序中首先定义了一个目标 LIST，它包括两个自参照基指针，使用了新的__self 关键字。这些基指针将以 LIST 目标所在段的段地址为基地址，在主程序中说明 list 是存放在 LISTSEG 段内，所以 left 和 right 基指针将指向 LISTSEG 段内的偏移量，因为这两个基指针是自参照的。

11.2 基指针应用于链表管理的工具包

11.2.1 基指针应用于链表管理

为了进一步解释基指针的应用特点，本节将较详细地讨论一个链表管理工具包。一般来说，与盘文件相比，链表又简单又快，可以很快地插入一个表项、删除一个表项或对各表项进行排序，开销也比较小。链表的缺点是容易产生碎片，频繁地增加和删除节点会很快地耗尽程序的堆空间。当把链表保存到盘文件中时也比较麻烦，必须把每一个表项写到文件上，每次一项。从盘文件中恢复链表到内存，除了要读回每个表项外，还要读回有关链表的其它信息。这些缺点大大限制了链表的应用。

如果把基指针运用于链表管理，则可以解决这些问题。这里提出的链表管理工具包不是把链表放在调用程序缺省的数据段内，而是把链表存放在另外一个新段内，通过基指针来管理随后在这个新段内所进行的内存分配工作。因为每一个链表都有单独的一段，所以这个链表管理程序可以管理多个链表。每个链表可以多达 64K 字节。即使以小模式或微模式来编译这个链表管理工具包，每个链表在程序的缺省数据段内占用的字节数都不会超过 4 个字节(用来存放一个柄)，也就是说，此时链表不会使程序的堆空间碎片化.

把链表单独放在另一个段内也解决了内存与盘文件之间的输入输出问题。整个段可以作为一个传输单位一次输入输出。因为链表内各表项之间的连接关系也是在这个独立段内维护，所以在读入链表文件时不需要重新建立这种连接关系。如果希望，还可以把链表管理工具包扩展到使用 EMS 内存。

限制每个链表不超过 64K 字节，在某些应用场合似乎会带来一些问题，特别是当应用于像图形处理这样的大型数据目标。但是，每个结点可以再包含一个远指针，这个远指针再指向段外的目标，不占用链表所在段的空间。这样，每个表项所代表的目标又可以多达 64K 字节，甚至还可以存放到 EMS 内存中去。下面这个链表管理工具包中，每个结点还包含了一个指向结点名的指针。如果每个结点所代表的目标较小，就可以用这个指针来存储目标而不需要目标指针了。

基指针使这一切成为可能。用户程序只需用一个远指针作为管理链表段的柄，至于链表各表项本身，它们之间的连接，就全都在一个段内通过基指针进行。

11.2.2 基指针分配函数

把基指针运用于链表管理的办法虽然很好，但它要求在程序的缺省数据段外，在链表段动态分配和释放内存，一般的内存分配函数无法解决这个问题。所以，Microsoft C 6.o 提供了一些新的与基指针配合使用的内存分配函数，这些函数名冠以前缀“__b”，都是在一个远段上进行操作。这些函数不仅允许程序员分配和释放一个远数据段，而且允许在这个远数据段内进行随后的子分配，就像上面所述的链表管理程序所要求的那样。这里，把在远数据段内的分配称作“子分配”，以便与通常的在缺省数据段内的分配区别。

对应于 Microsoft C 中每一个标准内存分配函数，都有一个带“__b”的内存分配函数，在调用__bheapseg 分配一个远数据段后，可以用__bmalloc 和__bfree 去子分配和子释放这个远数据段内的某一部分。通过函数__bheapwalk、__bheapchk 和__bheapset 可以检查堆的连贯性，通过函数__bheapchk 和 bheapmin 可以扩大和缩小一个堆段，已经子分配了的内存也可以通过____brealloc 重新子分配或扩大。最后，通过函数__bfreeseg 可以放弃一个远数据段。

除实现基指针外，带“__b”的内存分配函数还可以让程序员把有关的数据编成组，放在单独一个数据段内，从而简化有关数据的处理。

除__bheapseg 函数外，在调用所有其余函数时，第一个参数必须说明是在哪个基堆上工作。在源程序开头，必须包含#include malloc.h 语句。下面就是这些“__b”内存分配函数及其简短说明。

```
void __based(void) * __bcalloc(__segment seg, size__t num, size__t size);
```

这个函数子分配一块内存供一个数组使用，返回一个指向数组的基指针。

```
void __based(void) * bexpand(__segment seg,
                void __based(void) * memblock, size__t size);
```

这个函数用来子扩大或子缩小一个已子分配的块，失败时返回__NULLOFF。

```
void __bfree(__segment seg, void __based(void) * memblock);
```

这个函数子释放一个已子分配的块。

```
int __bfreeseg(__segment seg);
```

这个函数释放一个基堆，成功时返回 0，失败时返回-1。

```
int __bheapadd(__segment seg,void __based(void) * memblock,size__t size);
```

这个函数扩大一个基堆。

```
int __bheapchk(__segment seg);
```

这个函数检查堆的连贯性，返回一个整数说明堆的状态.

```
int  __bheapmin(__segment seg);
```

这个函数从基堆上释放尚未使用的内存，但它不把基堆退回给操作系统。成功时返回0、失败时返回−1。

```
__segment __bheapseg(size__t size);
```

这个函数分配一个基堆并返回段值供其它函数使用。失败时返回__NULLSEG。

```
int  __bheapset(__segment seg, unsigned fill);
```

这个函数以填充值(fill)填充所有未用的堆空间，返回值与__bheapchk返回值相同。

```
int  __bheapwalk(__segment seg, __HEAPINFO  *entryinfo);
```

这个函数遍历一个堆，取得堆中有关下一项的信息。如果调用时的seg参数值为__NULLOFF。则遍历所有基堆。返回值与__bheapchk返回值相同。

```
void __based(void)  *bmalloc(__segment seg, size__t size);
```

这个函数从堆中子分配一块内存。如果没有足够的内存，则返回__NULLOFF。调用时的seg值必须是以前由函数__bheapseg返回的值。

```
size__t __bmsize(__segment seg, void __based(void)  *memblock);
```

这个函数返回一个以前已子分配的块的大小。

```
void __based(void)  *__brealloc(__segment seg,void __based(void)  *memblock,size__t size);
```

这个函数改变一个以前已子分配的块的大小。

11.2.3 16个工具函数

链表管理程序动态地把链表建立在一个远数据段上，而这个远数据段仅当程序运行后才被分配，所以无法为基于这个段的各基指针预先说明一个段基址。链表头和各结点(表项)的数据结构内也不能使用自参照基指针，因为自参照基指针的目标要求一个特殊的基表达式。这里的解决办法是，链表管理程序维护了一个静态__segment型变量__tempseg，这个变量被用作各基指针的段基址。因为链表管理程序允许使用多个链表，__tempseg总是应该含有当前正被链表管理函数操作的那个链表的段地址。因此，工具包内每个函数都是以宏__listinit开头，这个宏把__tempseg初始化为链表被存储的那个段的段地址。

链表管理程序在表的远数据段内维护了两个数据结构。

第1个是整个表的控制块，这个结构含有链表的段地址、段大小、结点数、指向链表名字符串的指针和指向链表的第一个结点的指针。如下所示：

```
typedef struct __list LIST;
typesef struct __litem LISTITEM;
struct __list {
        __segment seg;
```

```
        unsigned num;
        unsigned segsize;
        char __based(__tempseg) * name;
        LISTITEM __based(__tempseg) * item;
};
```

第 2 个是每个结点的结构，它包括指向该结点名字符串的指针，指向表中位于它前面和位于它后面两个结点的指针，指向该结点目标的远指针以及目标的大小。如下所示：

```
struct __litem {
        void far * object;
        unsigned objectsize;
        char __based(__tempseg) * name;
        LISTITEM __based(__tempseg) * prev;
        LISTITEM __based(__tempseg) * next;
};
```

除指向结点目标的指针外，其余的指针都是以变量__tempseg 的值为段基址，这个变量总是被装入某一特定链表段的段地址。由链表管理程序分配的表的每一个成份都在链表段内，这使得表的维护比较容易，也便于保存到盘上或从盘上恢复。

链表管理工具包中共含有 16 个工具函数。使用这个工具包的应用程序不需要知道 Microsoft C 6.0 的基指针内存分配函数，甚至不需要知道基指针。应用程序只需调用 listinit 函数，用这个函数返回的远指针来参照和进一步操作新建立的链表。在调用了 listinit 后，应用程序就可以使用其余的 10 多个函数对链表进行操作。下面列出所有这些函数的简要说明。

```
void far * listinit(unsigned members, char * listname);
```

这个函数建立一个新链表，估算这个表所需要的内存，分配一个新数据段。可以在调用 listinit 时用 listname 参数给链表起一个名字，也可以在以后调用 listsetname 时给链表补一个名字。listinit 函数返回一个远指针，在随后所有对其它函数的调用中应该用这个远指针作为一个柄来参照链表段。

```
unsigned listadd(void far * listptr, void far * object,
            unsigned objectsize, char * membername);
```

这个函数往链表中增加一个新结点，返回一个参照这个新结点的柄。目标指针 object 可以是 NULL。参数 membername 用来给这个结点取一个名字，或者用作结点所有字符串的指针。

```
unsigned listfind(void far * listptr, char * membername);
```

这个函数根据结点名来寻找一个结点，返回这个结点的柄。

```
void far * listdelete(void far * listptr, char * membername);
```

这个函数从链表中删除指定名字的结点。如果这个结点有指向目标的远指针，则返回

这个指针，否则返回 NULL。

```
void listdestroy(void far *listptr);
```

这个函数放弃一个链表及与其相关的目标，收回它们所占用的内存。

```
void listsetobject(void far *listptr, void far *object,
                   unsigned objectsize, char *membername);
```

这个函数初始化一个结点的目标指针和目标大小。

```
void far *listgetobject(void far *listptr, char *membername);
```

这个函数根据结点名返回这个结点的目标指针。

```
void listdeleteobject(void far *listptr, char *membername);
```

这个函数删除一个结点的目标，收回它占用的内存。

```
void listsetname(void far *listptr, char *listname);
```

这个函数设置链表的名字。

```
unsigned listgetnumitems(void far *listptr);
```

这个函数返回链表中结点的数目。

```
unsigned listgetitem(void far (listptr);
```

这个函数返回链表中第 1 个结点的柄。

```
unsigned listitemgetnextitem(void far *listptr,
                   unsigned litemhdl);
```

这个函数返回链表中下一个结点的柄。

```
void far *listitemgetobject(void far *listptr,
                   unsigned litemhdl);
```

这个函数返回指向结点目标的指针。

```
void listdump(void far *listptr);
```

这个函数在标准输出设备上显示链表信息。

```
unsigned listsave(void far *plist, char *filename);
```

这个函数保存链表到一个指定的盘文件上。它首先往文件上写一个结构，这个结构中包括段的大小和链表柄(指针)的偏移量。然后，把整个链表段写到这个盘文件上，最后写每个结点的目标。

```
void far *listrestore(char *filename);
```

这个函数从盘上把整个链表恢复到内存。它首先打开盘文件，读文件起始部分的结构。根据这个结构中的段大小分配一块内存。当这个函数调用__listinit 建立新段时，返回的LIST 指针可能与文件中原来保留的不一致，因此，LIST 指针的偏移量被设置为等于原先保存在文件起始部分的链表柄(指针)的偏移量。然后，这个函数把整个链表读入新分配的段内，最后读每个结点的目标到它所建立的各远内存段中。

```
/*  list.h  */
void far *listinit(unsigned members, char *listname);
unsigned listadd(void far *listptr, void far *object,
                 unsigned objectsize, char *membername);
unsigned listfind(void far *listptr, char *membername);
void far *listdelete(void far *listptr, char *membername);
static void far *__listdelete(void far *plist, unsigned bplitem);
void listdestroy(void far *listptr);
void listsetobject(void far *listptr, void far *object,
unsigned objectsize,char *membername);
void far *listgetobject(void far *listptr, char *membername);
void listdeleteobject(void far *listptr, char *membername);
void listsetname(void far *listptr, char *listname);
static void far *__listinit(unsigned realsize);
unsigned listgetnumitems(void far *listptr);
unsigned listgetitem(void far (listptr);
unsigned listitemgetnextitem(void far *listptr,
                     unsigned litemhdl);
void far *listitemgetobject(void far *listptr,
                     unsigned litemhdl);
unsigned listsave(void far *plist, char *filename);
void far *listrestore(char *filename);
void error__exit(int err, char *msg);
void listdump(void far *listptr);
void listdumpitem(unsigned temp);

/*  list.c  */
#include  <stdio.h>
#include  <stdlib.h>
#include  <malloc.h>
#include  <string.h>
#include  <dos.h>
#include  <fcntl.h>

static    __segment __tempseg;

typedef   struct    __list     LIST;
typedef   struct    __litem    LISTITEM;

struct    __list {
     __segment  seg;
     unsigned   num;
     unsigned   segsize;
```

```
    char __based(__tempseg) * name;
    LISTITEM __based(__tempseg) * item;
};

struct   __litem {
    void far  * object;
    unsigned  objectsize;
    char __based(__tempseg) * name;
    LISTITEM __based(__tempseg) * prev;
    LISTITEM __based(__tempseg) * next;
};

typedef  struct   __listheader {
    unsigned  segsize;
    unsigned  offset;
} LISTHEADER;

typedef  LIST far                        * PLIST;
typedef  LIST __based(__tempseg)          * BPLIST;
typedef  LISTITEM __based(__tempseg)      * BPLITEM;

#define   LITEMHDL                        BPLITEM
#define   __tlistinit( )           (__tempseg=plist->seg)

PLIST    listinit(unsigned members, char * name);
BPLITEM   listadd(PLIST plist, void far * object, unsigned size, char * name);
LITEMHDL  listfind(PLIST plist, char * name);
void far  * listdelete(PLIST plist, char * name);
static    void far  * __listdelete(PLIST plist, BPLITEM bplitem);
void listdestroy(PLIST plist);
void listsetobject(PLIST plist, void far * object, unsigned size, char * name);
void far * listgetobject(PLIST plist, char * name);
void listdeleteobject(PLIST plist, char * name);
void listsetname(PLIST plist, char * name);
static          PLIST __listinit(unsigned realsize);
unsigned  listgetnumitems(PLIST plist);
LITEMHDL  listgetitem(PLIST plist);
LITEMHDL  listitemgetnextitem(PLIST plist, LITEMHDL bplitem);
void far  * listitemgetobject(PLIST plist, LITEMHDL bplitem);
void error__exit(int err, char * msg);
void listdump(PLIST plist);
void listdumpitem(BPLITEM temp);

PLIST listinit(unsigned members, char * name)
{
    long realsize;
    PLIST plist;

    if(!members)
        members=((0xffff / 2) / sizeof(LISTITEM));
    realsize=sizeof(LISTITEM);
```

```
        realsize * =members;
        if(realsize>65535L) return NULL;
        if((plist=__listinit((unsigned)realsize+sizeof(LIST)))!=NULL) {
                plist->name=__NULLOFF;
                listsetname(plist, name);
                plist->num=0;
        }
        return plist;
}

static    PLIST __listinit(unsigned realsize)
{
        BPLIST plist;
        if((__tempseg=__bheapseg(realsize))==__NULLSEG) return NULL;
        if((plist=(BPLIST)__bmalloc(__tempseg, sizeof(LIST)))==__NULLOFF) {
                __bfreeseg(__tempseg);
                return (PLIST)NULL;
        }
        plist->seg=__tempseg;
        plist->segsize=realsize;
        plist->item=__NULLOFF;
        return    (PLIST)plist;
}

void listsetname(PLIST plist, char * name)
{
        __tlistinit( );
        if(plist->name==__NULLOFF)
                __bfree(plist->seg, plist->name);
        if((plist->name=__bmalloc(plist->seg, strlen(name)+1))!=__NULLOFF)
                __fstrcpy((char far * )plist->name, (char far * )name);
}

BPLITEM listadd(PLIST plist, void far * object, unsigned size, char * name)
{
        BPLITEM plistitem;
        __tlistinit( );
        if((plistitem=__bmalloc(plist->seg, sizeof(LISTITEM)))==__NULLOFF)
                return    __NULLOFF;
        plistitem->object=object;
        plistitem->objectsize=size;
        if((plistitem->name=__bmalloc(plist->seg, strlen(name)+1))!=__NULLOFF)
                __fstrcpy((char far * )plistitem->name, (char far * )name);
        if(plist->item==__NULLOFF) {
                plist->item=plistitem;
                plistitem->prev=__NULLOFF;
        }
        else {
                BPLITEM temp=plist->item;
                for( ;temp->next!=__NULLOFF;temp=temp->next);
                temp->next=plistitem;
```

```
            plistitem->prev=temp;
        }
        plistitem->next=__NULLOFF;
        plist->num++;
        return plistitem;
}

void far *listdelete(PLIST plist, char *name)
{
        BPLITEM temp;
        __tlistinit( );
        if((temp=listfind(plist, name))!=__NULLOFF)
            return __listdelete(plist, temp);
        plist->num--;
        return (void far *)NULL;
}

static void far *__listdelete(PLIST plist, BPLITEM bplitem)
{
        void far *object;
        __tlistinit( );
        object=bplitem->object;
        if(bplitem->prev!=__NULLOFF)
            bplitem->prev->next=bplitem->next;
        else
            plist->item->next=bplitem->next;
        if(bplitem->next!=__NULLOFF)
            bplitem->next->prev=bplitem->prev;
        __bfree(plist->seg, bplitem->name);
        __bfree(plist->seg, bplitem);
        return object;
}

void listsetobject(PLIST plist, void far *object, unsigned size, char *name)
{
        BPLITEM temp;
        __tlistinit( );
        if((temp=listfind(plist, name))!=__NULLOFF) {
            temp->object=object;
            temp->objectsize=size;
        }
}

unsigned listgetnumitems(PLIST plist)
{
        __tlistinit( );
        return plist->num;
}

void listdeleteobject(PLIST plist, char *name)
{
```

```
    BPLITEM temp;
    _tlistinit( );
    if((temp=listfind(plist, name))!=_NULLOFF) {
        _ffree(temp->object);
        temp->object=(void far *)NULL;
    }
}

void listdestroy(PLIST plist)
{
    BPLITEM bplitem;
    _tlistinit( );
    for(bplitem=plist->item;bplitem!=_NULLOFF;bplitem=bplitem->next)
        _ffree(bplitem->object);
    _bfreeseg(plist->seg);
}

LITEMHDL listfind(PLIST plist, char *name)
{
    BPLITEM temp;
    _tlistinit( );
    for(temp=plist->item;temp!=_NULLOFF;temp=temp->next)
        if(!_fstrncmp((char far *)temp->name, (char far *)name,
                                                strlen(name)))
            return (LITEMHDL)temp;
    return 0;
}

void listdump(PLIST plist)
{
    BPLITEM temp;
    _tlistinit( );
    printf("listname=%Fs\n", (char far *)plist->name);
    printf("located @ %lp, occupies %u bytes in %04x:0000"
           " and contains:\n",
           (void far *)plist,
           plist->segsize,
           plist->seg);
    if(!plist->item) {
        printf("no items \n");
        return;
    }
    for(temp=plist->item;temp!=_NULLOFF;temp=temp->next)
        listdumpitem(temp);
}

void listdumpitem(BPLITEM bplitem)
{
    char far *name1;
    char far *name2;
    void far *prev;
```

```
        void far *next;
        if(bplitem->next!=__NULLOFF) {
              next=(void far *)bplitem->next;
              name1=(char far *)bplitem->next->name;
        }
        else {
              next=NULL;
              name1="NULL";
        }
        if(bplitem->prev!=__NULLOFF) {
              prev=(void far *)bplitem->prev;
              name2=(char far *)bplitem->prev->name;
        }
        else {
              prev=NULL;
              name2="NULL";
        }
        printf("item: %Fs is located @ %lp"
                "(stires rekated object @ %lp)"
                "\n\t(prev=%lp \"%Fs\" next=%lp \"%Fs\")\n",
                (char far *)bplitem->name,
                (void far *)bplitem,
                (void far *)bplitem->object,
                prev,
                name2,
                next,
                name1);
}

void far *listitemgetobject(PLIST plist, LITEMHDL bplitem)
{
      __tlistinit( );
      return (bplitem)->object;
}

void far *listgetobject(PLIST plist, char *name)
{
      BPLITEM temp;
      __tlistinit( );
      if((temp=listfind(plist, name))!=__NULLOFF)
            return temp->object;
      return (void far *)NULL;
}

LITEMHDL listgetitem(PLIST plist)
{
      __tlistinit( );
      return (LITEMHDL)plist->item;
}

LITEMHDL listitemgetnextitem(PLIST plist, LITEMHDL bplitem)
```

```
{
      __tlistinit( );
      return (LITEMHDL)(bplitem)->next;
}
unsigned listsave(PLIST plist, char * filename)
{
      int fh;
      unsigned doserror, bytes;
      BPLITEM  temp;
      LISTHEADER lh;
      void far * buf;
      __tlistinit( );
      FP__OFF(buf)=0;
      FP__SEG(buf)=(unsigned)plist->seg;
      lh.segsize=plist->segsize;
      lh.offset=FP__OFF(plist);
      if(doserror=__dos__creat(filename, __A__NORMAL, &fh))
            error__exit(doserror, "unable to create list file");
      if((doserror=__dos__write(fh, (void far * )&lh, sizeof(lh), &bytes))
            || (bytes!=sizeof(lh)))
            error__exit(doserror, "unable to write list header");
      if((doserror=__dos__write(fh, buf, plist->segsize, &bytes))
            || (bytes!=plist->segsize))
            error__exit(doserror, "unable to write list segment");
      for(temp=plist->item;temp!=__NULLOFF;temp=temp->next)
            if(temp->object)
                  if((doserror=__dos__write(fh, temp->object, temp->objectsize,
                        &bytes)) ||(bytes!=temp->objectsize))
                        error__exit(doserror,
                                    "unable to write list file object");
      __dos__close(fh);
      return 0;
}

PLIST listrestore(char * filename)
{
      int fh;
      unsigned doserror, bytes;
      PLIST plist;
      LISTHEADER lh;
      void far * buf;
      BPLITEM temp;
      if(doserror=__dos__open(filename, O__RDONLY, &fh))
            error__exit(doserror, "unable to open list file");
      if((doserror=__dos__read(fh, (void far * )&lh, sizeof(lh), &bytes))
            || (bytes!=sizeof(lh)))
            error__exit(doserror, "unable to read list file header");
      if(!(plist=__listinit(lh.segsize))) {
            __dos__close(fh);
            return (PLIST)NULL;
      }
```

```
        FP_OFF(plist)=lh.offset;
        FP_SEG(buf)=plist->seg;
        FP_OFF(buf)=0;
        if((doserror=_dos_read(fh, buf, lh.segsize, &bytes))
            || (bytes!=lh.segsize))
            error_exit(doserror, "unable to read list file segment");
        plist->seg=(_segment)FP_SEG(buf);
        for(temp=plist->item;temp!=_NULLOFF;temp=temp->next)
            if(temp->object) {
                if(!(temp->object=_fmalloc(temp->objectsize)))
                    error_exit(0, "unable to allocate space for object");
                if((doserror=_dos_read(fh, temp->object, temp->objectsize,
                    &bytes)) || (bytes!=temp->objectsize))
                    error_exit(doserror, "unable to read list object");
            }
        _dos_close(fh);
        return plist;
}

void error_exit(int err, char *msg)
{
        printf("%s\n", msg);
        exit(err);
}
```

11.2.4 工具包应用三例

11.2.4.1 [例 1]

设本例的名字为 testlist.c，编译行命令应该是：

```
cl /Lr -W4 -Zpiel -Od testlist.c list.c /link /co /cp:1
```

本例建立一个链表，链表中增加 n 个结点，然后调用 listdump 函数显示这 n 个结点及它们之间的连接。

```
/*   testlist.c          */
    #include <stdio.h>
    #include <malloc.h>
    #include "list.h"
    #define OBJECTSIZE 10
    void main(void)
    {
        void far *l;
        if (l=listinit(0, "TEST")) {
            listadd(l, _fmalloc(OBJECTSIZE), OBJECTSIZE, "JUNK01");
            listadd(l, _fmalloc(OBJECTSIZE), OBJECTSIZE, "JUNK01");
            listadd(l, _fmalloc(OBJECTSIZE), OBJECTSIZE, "JUNK01");
            listadd(l, _fmalloc(OBJECTSIZE), OBJECTSIZE, "JUNK01");
            listadd(l, _fmalloc(OBJECTSIZE), OBJECTSIZE, "JUNK01");
        }
```

```
        listdump(l);
    }
```

11.2.4.2 [例 2]和[例 3]

[例 2]和[例 3]的名字分别是 makanim.c 和 animate.c，编译行命令应该分别是：

```
cl /Lr -W4 -Zpiel -Od makanim.c list.c /link /co /cp:1
cl /Lr -W4 -Zpiel -Od animate.c list.c /link /co /cp:1
```

[例 2]是把保存在盘上的一系列文本屏幕映像文件在内存中链接起来，形成一个链表，链表中每一结点的目标对应一个文本屏幕映像文件，再把这个综合起来的链表文件以名字"ANIMATE.LST"存到盘上。本例假设这一系列文本屏幕映像文件的名字是：APPART1，APPART2，...，APPART21。本例没有限制这 21 个文件的内容，但假设它们是在屏幕上画了一个小方框，这个小方框从屏幕顶端向底端逐渐移动形成 21 幅画面。但是，链表文件实际上是由 41 个结点构成的，增加的 20 个结点是考虑到小方框再由屏幕底端沿原路弹回到屏幕顶端所形成的 20 幅画面。本例未包含形成这 21 个文件的部分。这 21 个文件可以用任何一种文本屏幕捕俘软件预先形成，与链表管理无关。

```
/* ma.bat */
/* cl/Lr_W4_Zpiel_Od makanim.c list.C/link/co/cp:/ */
/* makanim.c */
#include <stdio.h>
#include <dos.h>
#include <fcntl.h>
#include <malloc.h>
#include <conio.h>
#include "list.h"
#define SCREENSIZE 4000
#define SCREEN_ADDRESS 0Xb8000000
void show_file(char *filename);
void show_object(void far *object);

void show_file(char *filename)
{
    int fh;
    unsigned bytes, doserror;
    void far *screen= (void far *)SCREEN_ADDRESS;
    if(doserror=_dos_open(filemname, O_RDONLY, &fh));
        error_exit(doserror, "Unable to open screen file");
    if((doserror=_dos_read(fh, screen, SCREENSIZE, &bytes)) ||
            (bytes!=SCREENSIZE))
        error_exit(doserroe, "Unable to read screen file");
    _dos_close(fh);
}

void show_object(void far *object)
{
```

```
    unsigned i;
    unsigned far *screen=(void far *)SCREEN_ADDRESS;
    unsigned far *objptr=(unsigned far *)object;
    for(i=0;i<SCREENSIZE/2;i++, screen++, obptr++)
        *screen=*objptr;
}
void main(void)
{
    void far *l;
    void far *object;
    char *screen="APPART";
    char name[20];
    int fh;
    unsigned doserror, bytes;
    if(l=listinit(170, "ANIMATE")) {
        int i;
        for(i=0;i<22;i++) {
            sprintf(name, "%s%02d.ACP", screen, i);
            if(object=_fmalloc(SCREENSIZE)) {
                printf("Adding %s\n", name);
                if(doserror=_dos_open(name, O_RDONLY, &fh))
                    error_exit(doserror,
                               "Unable to open screen file");
                if((doserror=_dos_read(fh, object, SCREENSIZE, &bytes))
                               || (bytes!=SCREENSIZE))
                    error_exit(doserror,
                               "Unable to read screen file");
                _dos_close(fh);
                listadd(l, object, SCREENSIZE, name);
            }
            else {
                printf("Unable to allocate far memory: i=%d\n", i);
                break;
            }
        }
        for(i=20;i>0;i--) {
            sprintf(name, "%s%02d.ACP", screen, i);
            if(object=_fmalloc(SCREENSIZE)) {
                printf("Adding %s\n", name);
                if(doserror=_dos_open(name, O_RDONLY, &fh))
                    error_exit(doserror,
                               "Unable to open screen file");
                if((doserror=_dos_read(fh, object, SCREENSIZE, &bytes))
                               || (bytes!=SCREENSIZE))
                    error_exit(doserror,
                               "Unable to read screen file");
                _dos_close(fh);
                listadd(l, object, SCREENSIZE, name);
            }
            else {
                printf("Unable to allocate far memory: i=%d\n", i);
```

```
                    break;
                }
            }
            listdump(l);
            printf("Saving list ...\n");
            listsave(l, "ANIMATE.LST");
            printf("Destroying list ...\n");
            listdestroy(l);
            printf("Restoring list ...\n");
            if(l=listrestore("ANIMATE.LST")) {
                printf("List restored\n");
                listdump(l);
            }
        }
}
```

[例 3]是把[例 2]形成的链表文件(其文件名应为 ANIMATE.LST)再读进内存，并不断循环显示链表文件中各结点目标的内容，其效果是一个小方框在屏幕顶端和底端之间不断来回跳动，直至按下任一键。启动[例 3]程序(名为 ANIMATE)的命令行格式是:

ANIMATE　ANIMATE.LST　[/B]

选择项 B 是指定用基指针屏幕显示程序，否则将用远指针屏幕显示程序。实践将证明，这两种办法在执行速度上几乎一样。

```
/* an.bat */
/* cl/Lr_w4_Zpiel_Od animate.c list.c/link/co/cp:1 */
/* animate.c */
#include <stdio.h>
#include <dos.h>
#include <conio.h>
#include "list.h"
#define SCREENSIZE 4000
#define SCREEN_ADDRESS 0xb8000000
void fp_show_object(unsigned far *object);
void bp_show_object(unsigned far *object);
void main(int argc, char **argv);
unsigned far *screen=(void far *)SCREEN_ADDRESS;
void fp_show_object(unsigned far *object)
{
    unsigned i;
    unsigned far *sptr=screen;
    for(i=0;i<SCREENSIZE/2;i++, *sptr++=*object++) ;
}
unsigned _based(screen) *bptr;
void bp_show_object(unsigned far *object)
{
    unsigned i;
    bptr=0;
    for(i=0;i<SCREENSIZE/2;i++, *bptr++=*object++) ;
```

```
}
void main(int argc,char * * argv)
{
    void far * l;
    void far * object;
    char * screen="APPART";
    unsigned numobjects,i,based=0;
    unsigned lih;
    if(argc<2||argc>3)
        error_exit(0,"Usage:ANIMATE <filename.LST>
                [/Based pointer display]");
    if(argc==3 && ((*argv[2]=='-' || *argv[2]=='/') &&
                (argv[2][1]=='b' || argv[2][1]=='B')))
        based=1;
    if(l=listrestore(argv[1])) {
        numobjects=listgetnumitems(l);
        while(1) {
            lih=listgetitem(l);
            for(i=0;i<numobjects;i++) {
                if(object=listitemgetobject(l,lih))
                    if(based)
                        bp_show_object((unsigned far *)object);
                    else
                        fp_show_object((unsigned far *)object);
                lih=listitemgetnextitem(l,lih);
            }
            if(kbhit( ))
                break;
        }
    }
}
```

第 12 章　与 BIOS 和 DOS 的接口

本章主要讨论在 C 语言程序中调用 BIOS 和 DOS 服务的方法。其所以要讨论这个问题，主要有如下两点原因:

(1) 程序通过 BIOS 和 DOS 服务可以取得更好的效果，更高的执行速度。

(2) 使读者理解 PC 系列机上中断处理的机制，为编写自己的中断处理程序提供基础。

BIOS 和 DOS 服务不是以函数，而是以中断指令的形式提供的，因此在 C 语言中必须通过函数库中提供的一些专用函数才能得到这些服务。

12.1　中断概述

80X86 微机，最多可支持 256 个中断，包括硬中断和软中断。为了说明是哪一个中断，为每个中断取了一个“名字”，即编了一个号码 (0～255)，称作中断向量。每个中断向量都有一个相应的中断处理程序，这些中断处理程序的入口地址(包括段地址和偏移量)称作中断向量的值，它们依次存放在从绝对内存地址 0 开始的 1K 字节内，每个中断向量占 4 个字节。

在 80X86 机中，产生中断的根源以及它们的优先级顺序如下所示:

8086	80286
软中断,“意外”	软中断,“意外”
不可屏蔽中断	单步中断
可屏蔽中断	不可屏蔽中断
单步中断	处理器扩展
	段溢出
	可屏蔽中断

在这些中断中，除了软中断外，其余全都属于硬中断。软中断一般是由程序员的一条 INT n 中断指令产生的，也可能是由条件中断指令 INTO 产生的。执行 INTO 指令时，如果标志寄存器的溢出标志位 OF 是 1，则执行一条 INT 4 指令。

“意外”是某些指令在遇到错误条件时产生的中断。对于 8086，只有一种“意外”，是由除法得到的商太大(通常是因为除数为 0)引起的，执行一条 INT 0 指令。对于 80286，则产生“意外”的原因有如下一些:

产生“意外”的原因	采取的办法
除法错	INT 0
Bound指令范围超出	INT 5
无效操作码	INT 6

段溢出	INT d
处理器扩展不可用	INT 7
处理器扩展段溢出	INT 9
处理器扩展错	INT 10
LIDT指令错	INT 8

80286对"意外"中断的处理与8086稍有不同，压入堆栈的返回地址不是下一条指令的地址，而是产生"意外"的这一条指令的地址。这样，可以确定是哪一条指令产生了"意外"。在8086中，尽管只有除法指令会产生"意外"，但因各种除法指令的长度不一，故不知道"意外"是哪一条指令产生的。

因为CPU一次只能执行一条指令，而软中断和"意外"中断都是由于执行指令而产生的，即不可能同时产生，所以它们可以排在同一优先级内。

在PC机上不可屏蔽中断是用于报告奇偶校验出错和某些8087数值协处理器出错，在PC／AT机上只用于报告奇偶校验出错。

PC机只用一片8259中断控制器芯片，能接收8个可屏蔽中断请求（IRQ0～IRQ7）。在PC／AT机上，用了两片8259中断控制器，可以接受16个可屏蔽中断，(IRQ0～IRQ15)。因为多个可屏蔽中断可能同时产生，所以有个优先级问题，从IRQ0～IRQ15，优先级是逐次降低的。与这些中断相关连的设备以及中断号如下所示：

PC机和PC / AT机			**只在PC / AT机**		
IRQ	**INT**	**设备**	**IRQ**	**INT**	**设备**
0	8	定时器	8	70	实时时钟
1	9	键盘	9	71	
2	a		10	72	
3	b	COM2:	11	73	
4	c	COM1:	12	74	
5	d		13	75	80287
6	e	磁盘	14	76	磁盘
7	f		15	77	

仅当标志寄存器中的中断允许标志位IF为1时，才允许CPU接收可屏蔽中断。在汇编语言一级，可以通过sti和cli指令来置1和置0这个标志，在Turbo C中，则可以通过下列两个函数完成：

```
void disable(void);
void enable(void);
```

80286的处理器扩展和段溢出错是在80287数值协处理器试图存取一个多字节操作数时产生的，如果这个操作数从段尾回绕到段头，或者从该段尾覆盖到下一段头。

单步中断（即硬中断1）是供调试程序用的。当单步中断标志位TF为1时，执行完当前指令后就产生一次中断1，除非在当前这条指令执行完前发生了另一个优先级更高的中断。在8086机上，任何中断都比单步中断优先级高，但在80286机上只有"意外"中断比单步中断优先级高，这是为了能跟踪和分析中断服务程序本身。

所有软中断指令都自动置0标志位IF，禁止一切可屏蔽中断，以保证中断处理程序在其执行期间不被中断。如果中断处理程序过长，为保证产生的可屏蔽中断能等到及时处

理，则应在中断处理程序内通过 sti 指令重新允许可屏蔽中断。

当多个可屏蔽中断同时发生时，它们的优先级是由 8259 中断控制器确定的，它保证处理一个可屏蔽中断时，只有优先级更高的才可以被 CPU 接收。但是，可屏蔽中断处理程序什么时候结束处理，CPU 无法知道，8259 中断控制器也无法知道，只能由程序员通过指令通知 8259 中断控制器。在 PC 和 PC／AT 机上，这条指令就是往 CPU 的口地址 0x20 写 1 个字节 0x20。

80x86 最多支持 256 个中断向量，为了避免在中断向量的分配上发生冲突，IBM 公司在设计 PC 机时，为这 256 个中断向量划分了用途范围：

中断向量数	用途
12	8086，80286
28	BIOS
32	外部设备
32	DOS
113	BASIC
39	用户软件
256	总计

但是，IBM 公司忽视了 80x86 芯片生产厂商 INTEL 公司的说明。在原来 8086 的芯片设计中，INTEL 公司声明保留中断 0x00～0x1f 供 CPU 使用，尽管它当时只用了 0x00～0x04。IBM 公司在设计 PC 机时用了 INTEL 保留的 32 个中断向量的一部分。而 INTEL 设计 80286 芯片时，它又使 CPU 扩充使用了 0x05～0x09 以及 0x0d、0x10 7 个中断向量。这就与 IBM 和 PC 机对中断的分配产生冲突。解决这种冲突的办法只能是重定向到用户(这里是 IBM)的中断处理程序，由它首先判断产生中断的原因，然后再分别处理。

下面是中断向量的分配表：

00...1f	INTEL公司保留
00...04	INTEL公司已用于8086和80286
05...09，0d，10	INTEL公司已用于80286
05，10...1c	BIOS用于各种软中断
08...09	BIOS用于定时器和键盘
1d...1f	BIOS用作某些重要表格的入口地址
20...3f	DOS为各种软中断保留
20...28，2f	在DOS 3.3中使用
40...5f	为外部设备保留，PC／AT已用其中的一部分作磁盘BIOS
42...44	已被EGA的BIOS使用
4a	PC／AT机用于实时时钟服务
60...66	未分配，供用户使用
67	EMS管理使用

68...6f	未分配，供用户使用
70...77	PC / AT BIOS用于各种外部硬中断
78...7f	未分配，供用户使用
80...85	划归BASIC，但实际未使用
86...fo	BASIC已用
f1...ff	未分配，供用户使用

可以看出，这种分配已部分重叠，但为了保持兼容性，现在只能在这个基础上开发了。

机器引导时，首先是 BIOS 为某些中断向量赋值，即指定中断处理程序。待到 BIOS 把 DOS 引导到内存并在 DOS 初始化后，DOS 又为某些中断向量赋值。还可能修改 BIOS 为中断向量赋的值。待到 DOS 装入并运行某一程序时（包括常驻内存的程序），这个程序又可能增加或修改某些中断向量的值。每个程序都可能这样做，Turbo C 就修改中断向量 0 的处理程序，使之通过标准错误设备输出信息"Divide error"，终止程序的执行，并使 DOS 的全局变量 ERRORLEVEL＝3。

下面这个小程序用于打印出所有 256 个中断向量的值。在装入一个常驻内存程序之前和之后，分别运行这个程序并比较它们的结果，就知道常驻程序用到了哪些中断向量。

```
#include <general.h>
void main( )
{
    int i, j, k;
    void far *l;
    printf("Interrupt vectors\n\n");
    for (i=0;i<64;++i) {
        for (j=0;j<4;++j) {
            k=64*j+i;
            l=MK_FP(0, 4*k);
            printf("%02x  %Fp      ", k, *l);
        }
        printf("\n");
    }
}
```

12.2 与 BIOS 的接口

12.2.1 与 BIOS 的接口函数

为了从程序中直接发一条软中断指令，Turbo C 中提供了如下几个函数。对于那些专门为取得某类 BIOS 服务而提供的函数，在叙述那些服务时再介绍。

```
void geninterrupt(int n);
int  int86x(int n, union REGS *inregs, union REGS *outregs,
            struct SREGS *segregs);
int  int86(int n, union REGS *inregs, union REGS *outregs);
void segread(struct SREGS *segregs);
```

```
void intr(int n, struct REGPACK *regs);
```

(1) 函数 geninterrupt 的功能等效于一条 INT n 指令。但是，在发 INT n 指令的前后，总是要设置和读取某些寄存器的值的，而这个函数没有提供任何存取寄存器值的手段，只能借助于伪变量。

伪变量看起来像变量但实际上却不是变量。对于除标志寄存器和指令计数器外的所有 8086 寄存器，包括：AX、AL、AH、BX、BL、BH、CX、CL、CH、DX、DL、DH、CS、DS、ES、SS、SP、BP、DI、SI，Turbo C 都为它们提供了相应的伪全局变量，其变量名就是在寄存器名前加一个下划线，如伪变量_AX。根据寄存器是 8 位的还是 16 位的，变量类型相应为无符号字符或无符号整数。有了伪变量，就可以使用 geninterrupt 这个函数。下面这个函数就是通过 BIOS 从指定页 page 的当前光标位置读 1 个字符，读得的 ASCII 码放在变量 ch 中，属性放在变量 attr 中。

```
void readchar(unsigned char page, unsigned char *ch,
              unsigned char *attr)
{
    _AH=8;
    _BH=page;
    geninterrupt(0x10);
    *ch=_AL;
    *attr=_AH;
}
```

伪变量只能存放在寄存器中，而寄存器是频繁使用的，故在调用函数 geninterrupt 之前设置的伪变量值和在调用函数之后这些伪变量值能维持多久难以确定。再者，通过伪变量强行设置某些寄存器的值（例如强行设置 DS 的值），可能会破坏程序后来执行的环境，引起意想不到的结果。

(2) 函数 int86x 也是执行一条 INT n 指令，入口时的寄存器值来自变量 inregs，返回时的寄存器值被保存在变量 outregs 中。变量 inregs 和 outregs 可以是同一个变量。变量的类型都是联合 REGS，其在头文件 dos.h 中的定义如下：

```
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;}
```

字段 flags 对应于标志寄存器，字段 cflag 仅对应于标志寄存器中的进位标志位。函数 int86x 在调用时不用这个字段，只在返回时设置这个字段，因为许多 BIOS 服务都是通过进位标志位为 0 还是为 1 来表示成功还是失败。

结构 SREGS 是为了设置和保存段寄存器的值，其在头文件 dos.h 中的定义如下：

```
struct SREGS {unsigned int es, cs, ss, ds; }
```

int86x 函数只用了结构 SREGS 中的段寄存器 DS 和 ES。调用时根据它们的值设置 DS 和 ES，同时把 DS 和 ES 原来的值保存在这个结构中，返回时再恢复 ES 和 DS 为原来的值，这就允许程序使用远指针或大数据编译模式。

int86x 函数置全局变量__doserrno 等于寄存器 AX 中的值，其返回值也等于 AX 中的值。

(3) 函数 segread 是配合 int86x 等函数使用的，它的作用是把当前段寄存器的值保存到 SREGS 型结构变量 * segregs 中。

(4) 函数 int86 的功能与 int86x 类似，但稍微简化了一些，因为它没有参数 segregs，入口和出口时都不处理段寄存器。从某种意义上讲，它的功能减弱了，因为不能跨段。但从另一个角度讲，使用 int86 更安全一些，因为如果在不需要跨段时调用了 int86x，而且参数 segregs 设置不正确或忘记设置，都会产生错误的结果。

(5) 函数 intr 与函数 int86x 类似，它的入口和出口是用同一个结构变量 regs 来传递寄存器的值。在头文件 dos.h 中，结构 REGPACK 的定义如下：

```
struct REGPACK {
     unsigned r__ax, r__bx, r__cx, r__dx;
     unsigned r__bp, r__si, r__di, r__ds, r__es, r__flags;
}
```

与 int86x 函数不同的是，intr 函数在返回时不设置全局变量__doserrno 的值，也没有返回值。

12.2.2 BIOS 提供的部分服务

PC／AT 机上提供的 BIOS 服务如下表所示：

软中断服务		有关的硬中断服务	
中断向量	服务	中断向量	中断源
11，12	系统配置		
15	PC／AT 扩伸内存		
15	PC／AT 保护方式内存移动		
10	显示		
16	键盘	9	IRG1
13	磁盘	e,76	IRQ6和IRQ14
15	磁带(PC／AT 没有)		
15	游戏操纵杆(只在 PC／AT)		
5，17	打印		
1a	定时器	8	IRQ0
15，1C	实时时钟(只在PC／AT)	70,4a	IRQ8
		a,d f,	IRQ2，5，7
		71-74	IRQ9-12
		75	IRQ13
		77	IRQ15
14	串行通信2	b	IRQ3
14	串行通信1	c	IRQ4
		2	NMI
1b	Ctrl__Break		
3	断点	1	单步
4	溢出 INTO		
15	PC／AT 多任务		
		0	除法指令“意外”
		5-10,d	80286指令“意外”

通过 12.2.1 节介绍的几个函数，可以得到所有这些 BIOS 服务。但是，为了使用更加方便，Turbo C 和 Turbo C Tools 为其中的某些服务提供了专门的手段。下面先介绍比较“零散”的几个，一些常用的专项服务将从 12.4 节开始逐项介绍。

(1) 设备服务

通过 Turbo C 的 biosequip 函数可以得知系统当前的某些配置情况，它对应中断 0x11。

```
int biosequip(void);
```

这个函数返回值的含义如下：

位15−14	并行口数目
位13	接有串行打印机(为1时)
位12	接有游戏操纵杆(为1时)
位11−9	串行口数目
位8	没有DMA(为1时)
位7−6	软盘驱动器数减1
位5−4	初始屏幕方式
位3−2	母板上RAM容量
位1	存在数值协处理器(为1时)
位0	只要有1个软盘存在，则为1

这个函数只能在 PC 机上工作，在 PC／AT 机上无多大用处。

(2) 内存服务

通过 Turbo C 的 biosmemory 可以知道系统中常规内存的容量，单位是 K 字节，它对应 BIOS 中断 0x12。

```
int biosmemory(void);
```

通过 BIOS 中断 0x15 的子服务号 0x88 可以知道系统中扩伸内存的容量，单位也是 K 字节。但 Turbo C 和 Turbo C Tools 都没有提供相应的函数，下面这个小程序可以做这件事：

```
#include <dos.h>
unsigned extmem(void)
{
    struct REGPACK r;
    r.r_ds=_DS;
    r.r_es=_ES;
    r.r_ax=0x8800;
    intr(0x15, &r);
    return r.r_ax;
}
```

通过 BIOS 中断 0x15 的子服务号 0x87 可以在常规内存和扩伸内存中互相搬移数据。调用这个服务时要提供 24 位的源地址和目的地址，以及搬移的字节数。这个 BIOS 服务能把 CPU 置为保护方式，搬移数据，然后再返回到实地址方式。遗憾的是，Turbo C

和 Turbo C Tools 都没有提供这样的函数。

(3) IRQ2 和 IRQ9

PC 机上只有一片 8259 中断控制器芯片，它的 IRQ2 是保留的，一般没有使用。PC／AT 机上有了两片 8259 芯片，IRQ2 用于把两片 8259 级联起来，原来接在 IRQ2 上的设备(如果有)现改接到 IRQ9 上。IRQ2 和 IRQ9 的中断号分别是 0x0a 和 0x71。若在 IRQ9 上发生中断，中断号 0x71 的处理程序将会自动转移到中断号 0x0a 的处理程序。

(4) INT 2

不可屏蔽中断 NMI 发生时，会调用 BIOS 中断 2，这个服务程序在显示出“PARITY CHECK 2”信息后就死机了。在 PC 机上，引起 NMI 的原因可能是奇偶校验错，也可能是 8087 数值协处理器错。在 PC／AT 机上，80287 协处理器错将触发 IRQ13，相应的中断向量是 0x75，为了保持软件上的兼容性，中断 0x75 会自动转到中断 2。

(5) INT 1b

当检测到 Ctrl__Break 键时，BIOS 中断 9 会执行 INT 0x1b，并把 0x0000 放在键盘缓冲区。BIOS 中的中断 0x1b 处理程序什么也不做，但其它软件可能已把中断 0x1b 重新定向到它的处理程序。事实上，DOS 已经这样做了。但 DOS 的中断 0x1b 处理程序也只是置位一个标志，表示已检测到 Ctrl__Break 键。以后当执行某些 DOS 系统调用时，DOS 会自动检查这个标志。如果发现这个标志已置位，则调用中断 0x23 处理程序（见本章最后一节)。

(6) INT 1 和 INT 3

中断 1 是由单步中断标志位引起的硬件中断，中断 3 是软中断。与所有其它的软中断指令不同，INT 3 是单字节指令，因此常被 DEBUG 程序用来做断点指令。在调试某个程序时，若设立了断点，DEBUG 会用 INT 3 指令代替断点处的第 1 个字节。这样，当执行到那个断点时，就会执行 INT 3 指令，于是转到断点处理程序。单步中断处理程序和断点处理程序一般都是临时停止程序的运行，报告当时的各寄存器的内容以及其它状态，显示下一条将执行的指令，以便检查被调试程序的运行。从断点处理程序返回后，恢复断点下原来那个字节的内容，接着往下执行。由此可以看出，设置断点的办法只能用于调试 RAM 中的程序，不能用于调试 ROM 中的程序。

(7) INTO

如果溢出标志位 OF 为 1，则执行 INTO 指令会引起调用 INT 4。软件系统必须提供相应的处理程序，但 DOS 提供的极其简单，不显示任何信息就终止。

(8) “意外”

“意外”中断一般是由于程序错误以及各种 CPU 的内部条件而引起的，软件系统应该提供相应的处理程序。8086 机上只有一个“意外”中断（是除法溢出)，触发 INT 0。BIOS 的中断 0 处理程序什么也没做。DOS 改写了中断 0 的处理程序，显示出“Divide overflow”，然后停机。Turbo C 修改了中断 0 的处理程序，把它和 Turbo C 的错误报告系统联系起来。

PC／AT 机的 80286 有 8 个“意外”中断源，但除中断 0 外，中断 5～中断 10 以及中断 0x0d 都与各 BIOS 服务相冲突。BIOS 中已提供了这些中断服务，但不是为这些“意

外"服务，而是为别的事件服务。DOS 没有为这些中断提供服务。因此，如果用户程序不修改这些中断服务程序，则当发生这些意外时，得到的是不需要的服务。

12.3 与 DOS 的接口

12.3.1 与 DOS 接口的函数

为 DOS 分配的中断号 0x20～0x3f 现在的使用情况如下所示：

中断号(HEX)	服务	参考章节
20	程序终止	12.3
21	系统调用	见下表
22	程序终止地址	12.3节
23	Ctrl__Break处理程序地址	第13章
24	严重错误处理程序地址	第13章
25–26	绝对盘地址读／写	一般不用
27	常驻程序	第13章
28	空	第13章
29–2e	保留	
2f	打印机	12.4节
30–3f	保留	

DOS 提供的这许多服务中，最重要的是系统调用服务。系统调用有上百个，大致分类情况如下：

系统调用号(HEX)	简要说明	参考章节
0，26，33，4b–4d，62	程序执行和终止	12.6节
1，2，6–c，	标准输入／输出	12.4节
3，4	串行输入／输出	12.9节
5	打印机	12.7节
d–e.19–1a， 2e–2f，36， 39–47，4e–4f， 54，56–57，5a–5c	文件输入／输出	12.5节
f–17，1b–1c，21–24， 27–29	文件控制块FCB	一般不用
25，35	中断向量	12.3节
2a–2d	定时器	12.6节
30	DOS版本号	12.3节
31，34，50–51	中断服务程序	13章
38，63	国家信息	12.3节
48–4a，58	内存管理	12.6节
59	错误处理	12.10节
5e–5f	网络	
18，1e–20，32，37 52–53，55，5d，60–61	可能被DOS内部使用	

因为 DOS 服务也是通过中断指令调用的，所以 12.2.1 节介绍的几个函数都可以用来

调用 DOS 服务。但在大部分情况下，调用 DOS 服务的目的是为了利用它的“系统调用”，而所有“系统调用”都是通过中断 0x21 进入的，因此 Turbo C 还提供 4 个专用于取得这些系统调用的函数：

```
int bdos(int dosfun, unsigned dosdx, unsigned dosal);
int bdosptr(int dosfun, void *argument, unsigned dosal);
int intdos(union REGS *inregs, union REGS *outregs);
int intdosx(union REGS *inregs, union REGS *outregs,
            struct SREGS *segregs);
```

这 4 个函数的说明都在头文件 dos.h 中。函数 bdos 和 bdosptr 很少使用。函数 intdos 和 intdosx 分别与函数 int86 和 int86x 对应，除参数中少了一个中断号外，其余的使用方法全都一样（见 12.2.1 节）。

12.3.2 DOS 提供的部分服务

下面先介绍 DOS 服务中比较“零碎”的部分，一些重要的专项服务从第 12.4 节开始，再逐项介绍。

可以通过上面的函数来取得这些服务，但 Turbo C 和 Turbo C Tools 提供了一些更为方便的办法。

(1) DOS 版本号

Turbo C 维护了三个全局变量，保存 DOS 的版本号：

```
unsigned int  _version      DOS版本号
unsigned char _osmajor      DOS主版本号
unsigned char _osminor      DOS次版本号
```

(2) 读取和设置中断向量

Turbo C 提供了如下两个函数来读取和设置中断向量：

```
void interrupt(*getvect(int interruptno))( );
void setvect(int interruptno, void interrupt(*isr)( ));
```

函数 getvect 读取中断向量 interruptno 的值，把这个值作为指向中断函数的远指针返回。

函数 setvect 设置中断向量 interruptno 的值为一个新值 isr，isr 是一个指向新中断处理程序的远指针。作为中断处理程序使用的函数必须是 interrupt 类型的函数，见第 13 章。

(3) 国家信息格式

通过 Turbo C 的 country 函数可以读取指定国家或当前国家(若 xcode=0)的一些信息表达格式，如日期、时间、千分符、货币符号等等。

```
struct country *country(int xcode, struct country *cp);
```

12.4 标准输入／输出服务

标准输入设备一般指的是键盘，标准输出设备一般指的是显示器，因此这一节主要介绍显示和键盘，包括 BIOS 和 DOS 提供的服务。

12.4.1 BIOS 提供的显示服务

BIOS 提供的显示服务都是通过中断 0x10 调用的，共有 10 多个子功能，由入口时寄存器 AH 的值来决定。对于所有这些子功能，Turbo C Tools 几乎都提供了相应的函数(详见第 4 和第 5 章)。本节亦未详细列出 BIOS 服务，只把它们的对应关系做一些简短的说明。

AH 值(HEX)	特征	说明	对应的 Turbo C Tools 函数
0	TGE	置显示方式	scnewdev
f	TGE	取显示方式	scmode
5	TGE	置当前显示页	scpage
f	TGE	取当前显示页	scmode
1	TE	置光标形状	scpgcur
3	TE	取光标形状	sccurst
2	TEP	置光标位置	sccurset
3	TEP	取光标位置	sccurst
8	TGEP	从当前光标位置读字符码和属性	scread
9	TGEP	往当前光标位置写字符码和属性	scattrib
a	TGEP	往当前光标位置写字符码	scwrite
e	TGEP	往当前光标位置写 TTY 方式字符	scttywrt
13	TGEP	往当前光标位置写 TTY 方式字符串	
6／7	TGE	上／下滚	viscroll
4	TGE	取光笔位置	
b	TG P	置边界颜色	scborder
b	G	置调色板	scmode4
10	E	置 EGA 调色板寄存器	scblink scborder scpall scpalett
d／c	G	读／写一个点	grptread grptwrit
11	E	选择 EGA 字符集	scnewdev
12	E	返回 EGA 信息	scequip
12	E	选择另一个 EGA 屏幕打印函数	

表中特征栏 4 个字母的含义如下：

T	文本方式
G	图形方式
E	EGA
P	可以指定页号

12.4.2 BIOS 提供的键盘服务

BIOS 提供的键盘服务都是通过调用中断 0x16 而得到的，一共有三个子功能，分别由寄存器 AH 的值来选定。Turbo C Tools 有三个对应的函数，如下所示：

AH 值(HEX)	说明	对应的 Turbo C Tools 函数
0	从键盘读入一个字符	kbgetkey
1	查询是否有字符	kbready
2	查询键盘状态	kbstatus

键盘状态是放在 BIOS 数据区 0000:0417 中的一个字，BIOS 服务只是取回这个字的低字节，Turbo C Tools 函数 kbstatus 返回这个字的所有 16 位。16 位的定义如下所示。Turbo C Tools 的 kbset 函数还可以设置这些状态位。

15	Ins键按下	7	Ins锁在插入状态
14	Capslock键按下	6	Capslock锁定在大写状态
13	Numlock键按下	5	Numlock锁定在数字状态
12	Scrolllock键按下	4	Scrolllock锁定在滚动状态
11	Ctrl_Numlock锁定在暂停状态	3	Alt键按下
10	(PC／AT)SysReg键按下	2	Ctrl键按下
9未用		1	左shift键按下
8未用		0	右shift键按下

事实上，BIOS 还维护了一个键盘缓冲区，但没有为这个缓冲区提供任何服务。Turbo C Tools 为此提供了 4 个函数：kbqueue，kbflush，kbplace，kbstuff。这 4 个函数用到了 BIOS 数据区中如下 4 个地址：

BUFFER_HEAD	0000:041A	存放键盘缓冲区头指针
BUFFER_TAIL	0000:041C	存放键盘缓冲区尾指针
KB_BUFFER	0000:041E	键盘缓冲区起始地址
KB__BUFFER_END	0000:043E	键盘缓冲区结束地址

12.4.3 DOS 提供的标准输入输出服务

DOS 为标准输入输出服务提供了一些系统调用，与 BIOS 提供的服务不一样，DOS 的这些服务可以重定向到其它文件。

系统调用号	寄存器	特征	说明
输入:			
1	AL	B X W	getche 函数的基础
6(DL=FF)	AL, ZF		如果无输入可用，置 If=1
7	AL	W	getch 函数的基础
8	AL	B W	
a	DS:DX	BSXEW	DOS 的带缓冲的键盘字符串输入，cgets 函数的基础
b	AL	B	如果输入已准备好，则置 AL=ff，否则置 AL=0
c			刷清缓冲区，然后执行 AL 中规定的系统调用
输出:			
2	DL	B	
6(DL!=FF)	DL		
9	DS:AX	BS	字符串终止符是"Y"

表中特征栏 5 个字母的含义如下：

B　检查用户是否按了Ctrl_Break键，否则只在DOS的Break开关为ON时才检查。

S　字符串输入／输出，否则为字符输入／输出。

X　回显字符，否则不回显。

E　用户可以进行编辑，否则不可编辑。

W　如果没有输入可用，则等待，否则不等待。

12.5　文件输入／输出服务

文件输入／输出服务是 DOS 的核心，DOS 系统调用的大部分都是用来处理文件的，而 BIOS 中不存在文件输入／输出服务。实际上，Turbo C 和 Turbo C Tools 已提供了许多函数，这些函数或者直接对应某个 DOS 的系统调用，或者综合了多个 DOS 系统调用，不但容易使用，而且功能更强，移植性更好。下面提供的对照表供读者参考。

系统调用号(HEX)	说明	对应的C函数
影响整个系统:		
d	复位整个盘系统	
2e	置位／复位校核标志	setverify
54	取校核标志	getverify
影响某台盘:		
e	选择磁盘	setdisk
19	取当前盘	getdisk
1b	取当前盘分配表信息	
1c	取指定盘分配表信息	
36	取盘剩余空间	getdfree
影响某个目录:		

3b	改变当前目录	chdir
47	取当前目录	getcurdir
39	建立子目录	mkdir
3a	删除子目录	rmdir
影响某个文件:		
56	文件更名	rename
4e	寻找第1个匹配文件	findfirst
4f	寻找下一个匹配文件	findnext
1a	设置磁盘传输区DTA	setdta
2f	取磁盘传输区DTA	getdta
43	改变文件属性	chmod
57	取/置文件的日期和时间	getffime, setftime
3c	建立文件	Creat, _crat
5a	建立唯一临时文件	creattemp
5b	建立新文件	creatnew
41	删除文件	wnlink
3d	打开文件	_open
3e	关闭文件	_close
5c	上锁/开锁文件	lock, unlock
42	移动文件读写指针	lseek
3f	读文件	_read
40	写文件	_write
44	输入/输出控制	ioctl
44	判断标准输入/输出是否已重定向	isatty
45	复制文件柄	dup
46	强制复制文件柄	dup2

12.6 内存管理与程序执行服务

BIOS 中不存在内存管理与程序执行服务，只有 DOS 才会提供。与上一节一样，本节只提供一张 DOS 系统调用与 C 语言函数的对照表。

系统调用号	说 明	对应的C函数
48	分配内存	allocmem
49	释放已分配的内存	freemem
4a	修改已分配的内存块	setblock
中断0x20	终止程序，返回DOS	
0	终止程序，返回DOS	
4c	终止程序	_exit
中断0x22	终止程序后将跳转的地址	
62	取PSP地址	全局变量____PSP
4b	装入并执行子进程，返回父进程	spawn......
4b	装入并执行子进程，返回DOS	exec......
4d	取子进程返回码	DOS变量ErrorLEVEL
33	取Break开关状态	getcbrk
33	设置Break开关	setcbrk

12.7 打印服务

对于 C 语言程序员来说，一般没有必要用 BIOS 和 DOS 提供的低级打印机控制功能，通过文件系统打印就足可以了。因此，本节的重点不是介绍 BIOS 和 DOS 的低级打印功能，而是介绍 DOS 提供的 print.com 实用程序。

BIOS 对打印的支持有如下这些:

BIOS 支持	说　明	相应的 TC 函数	相应的 TCT 函数
中断 0x17，AH=0	打印一个字符	biosprint	prchar
中断 0x17，AH=1	初始化打印机	biosprint	prinit
中断 0x17，AH=2	取打印机状态	biosprint	prstatus
shift__prtsc	打印文本屏幕		
EGABIOS	打印 43 行文本屏幕		
	打印图形屏幕		

关于屏幕打印需要补充几点。当按下 shift__Prtsc 键时，则调用中断 5 处理程序打印出屏幕上的内容，但显示必须在文本方式下，也固定送往第 1 台打印机 LPT1。EGA 可以有 25 行和 43 行两种文本方式，一般情况下，只打印前 25 行。如果希望 43 行全部打印，则必须选择 EGA BIOS 支持的另一种屏幕打印方式。如果希望打印图形屏幕，则应用 DOS 提供的实用程序 graphics.com（这是一个常驻内存的程序），把中断向量 5 修改成它的处理程序。

DOS 对打印的支持有如下几项:

(1) 实用程序 graphics.com。

(2) 把<Ctrl__Prtsc>键当作一个乒乓开关使用，按一下送往打印机，再按一下又不送往打印机。

(3) 使用系统调用号 5，可往标准打印设备送 1 个字符。

(4) 提供了实用程序 print.com（常驻内存），通过它建立打印缓冲队列，并可往队列中增加、删除文件等。为了在程序员编的程序中也可以利用 print.com 提供的打印缓冲队列功能，DOS 还提供了中断 0x2f 处理程序，入口时如果寄存器 AH=1，则处理打印缓冲队列。其下又分 6 个子功能，分别由入口时 AL 寄存器的值来确定。对于这 6 个子功能，Turbo C Tools 提供了相应的支持函数。6 个子功能及其对应的 TCT 函数如下:

子功能号	说　明	相应的 TCT 函数
0	查看是否已安装 print.com	prinstld
1	向打印队列提交一个文件	prspool
2	从打印队列中撤消一个文件	prcancel
3	从打印队列中删除所有文件	prcancel
4	冻结和检查打印队列	prgetq
5	解冻打印队列	

表中的 4 个 Turbo C Tools 函数的说明如下:

```
int prinstld(void);
int prspool(const char *pfile);
int prcancel(char *pfile);
int prgetq(int n, int *psize, char *pfile);
```

函数 prinstld 报告 print.com 是否已驻留在内存。若已驻留，返回值为非 0，否则返回值为 0。

函数 prspool 将文件 * pfile 提交给打印队列。

函数 prcancel 从打印队列中撤消文件 * pfile。若指针 pfile 为空指针或所指向的字符串为空，则撤消所有文件。

函数 prgetq 读取打印队列中第 n 个文件的名字放在缓冲区 * pfile 中，缓冲区至少为 65 字节长。当前文件号是 0，依次往下编。队列中总的文件数目返回在 * psize 中。

后三个函数的返回值都是表示错误码，如下表所示。每个函数只报告其中的一部分错误。

PR_OK(0)	无错误
PR_INSTAL(1)	未安装print.com
PR_EXIST(2)	文件不存在
PR_PATH(3)	未找到路径
PR_HANDLES(4)	文件柄已用完
PR_ACCESS(5)	文件是卷标或目录等，违例
PR_EMPTY(6)	队列为空
PR_RANGE(7)	n超出范围
PR_FULL(8)	队列已满
PR_BUSY(9)	假脱机打印系列忙，无法通信
PR_NAMELEN(12)	全文件名超过64个字符

下面这个小程序报告打印队列中正在打印和没有打印的文件的名字。

```
#include <stdio.h>
#include <bprint.h>
int  i, ercode, qsize;
char filename[65];
void main( )
{
     ercode=prgetq(0, &qsize, filename);
     if(ercode= =PR_EMPTY)
          printf("Print queue is empty.\n");
     else
          if(ercode!=PR_OK)
               printf("Error %d accessing print queue.\n", ercode);
          else {
               printf("%s is currently being printed.\n", filename);
               for (i=1;i<qsize;i++) {
                    if(PR_OK!=prgetq(i, &qsize, filename)) {
                         printf("Error %d accessing print queue.\n",
                                   ercode);
```

```
                    break;
                }
                printf("%s is in queue.\n", filename);
            }
        }
    getch( );
}
```

12.8 时钟／日历服务

早期的 IBM／PC 机，其时钟／日历特性不能满足用户的要求，最明显的缺陷就是没有不丢失的实时时钟。尽管后来许多厂家都在其硬件上弥补了这个缺陷，但这种增强没有标准。BIOS、DOS 和其它子系统在日期／时间方面的数据格式是不兼容的，实时时钟没有任何标准的接口定义。PC／AT 机的 BIOS 使用户程序可以使用实时时钟。但是，日期／时间数据格式不兼容的问题仍然存在。

12.8.1 PC 机上的时钟系统

PC 机上的基础时钟是由主振荡器经 4 分频后再送到 8253 定时器芯片分频而得到的。送往 8253 的脉冲，频率是 1.193182MHz，8253 被设置为 16 位计数器，所以分频后的频率是 1193182／65536＝18.2。即时钟脉冲频率是 18.2Hz，周期约为 55ms，每个时钟脉冲称为一个"tick"。

每个时钟"tick"都会产生一次优先级最高的硬中断，中断请求是 IRQ0，相应的中断向量是 8。PC 机上的 BIOS 提供了中断 8 的处理程序，它主要做三件事。第 1，增量保存在 BIOS 数据区内的"tick"计数器（这个计数器占 4 个字节，每次引导时都初始化为 0）。这个中断 8 处理程序还每隔 24h 把这个计数器清 0，同时记下已经过了 24h 这件事。第 2，如果在预先规定的一个连续时间间隔内(约 2s)没有任何软盘读／写操作，则关掉软盘驱动器的电源。第 3，调用中断向量 0x1c。

BIOS 也提供了中断向量 0x1c，但什么也没做。用户如果希望在每个"tick"到来时执行某个任务，则应编写自己的中断 0x1c 处理程序（见第 13 章）。

BIOS 中还提供了与时钟有关的中断向量 0x1a，通过它可以读取或设置上述的"tick"计数器。为了编程方便，Turbo C 中提供了相应的函数：

```
long biostime(int cmd, long newtime);
```

如果 cmd＝0，则返回当前"tick"计数器的值，否则置"tick"计数器的值为 newtime。但是，这个函数不能反应自上次读计数器以来是否已过了 24h。Turbo C Tools 的函数对此做了修正：

```
int utgetclk(long *pcount)
```

这个函数只能报告"tick"计数器的值，但它通过返回值为 1 还是为 0，报告自上次以来是已经过了还是未过 24h 这个界限。另外，这个函数不是通过中断 0x1a。而是直接从 BIOS 数据区内读取"tick"计数器内容。

如果需要更精密的时钟，则要用 PC／AT 机或者对 8253 定时器芯片重新编程。

每次启动 DOS 时，DOS 要求操作者键入当时的日期和时间。然而，如果根目录下有 AUTOEXEC.BAT 批命令文件，DOS 就会假设这个批命令文件中含有某个可执行文件，由这个文件对机器的扩充功能板编程，从板上取得不丢失的实时时钟信息。简言之，假设有用户自已附加的硬件和软件。但是，这种硬件和软件是没有标准的，其口地址和数据格式可能因厂家不同而不同，它们可能会设置"tick"计数器的值，使"tick"计数器的值与当前的时间联系起来。这样，以后通过 DOS 的时钟服务（如 TIME 命令），才可根据读取的"tick"计数器的值算出正确的时间。

12.8.2 PC / AT 机上的时钟系统

PC／AT 上提供的实时时钟是由电池供电的，用的是 Motorola 的 MC146818 CMOS 芯片。它有两个重要改进：第 1，在启动 DOS 时，DOS 可以从这个芯片上取得日期和时间。第 2，时钟精度可以达到约 1ms。

为了支持这个新的硬件，PC／AT 机的 BIOS 中断 0x1a 除提供前面已讲过的两个服务外，还提供了另外 8 个服务。服务 0 和 1 仍然是读和写 BIOS 的"tick"计数器。服务 2～5 读和写 CMOS 芯片中的时钟和日历。时钟／日历的格式与 DOS 系统调用 2a～2d 中的格式不同。服务 6 设置闹钟的起闹时间(时，分，秒)。一旦到了起闹时间，就调用中断 0x4a，用户必须自已提供这个中断处理程序。服务 7 取消以前设置的闹钟。

PC／AT 机精度更高的定时是通过中断 0x15 的服务 0x83 和 0x86 提供的。服务 0x83 是以 ms 为单位设置闹钟的起闹时间，服务 0x86 是以 ms 为单位设置延迟时间。这两个服务的实现原理都是对 CMOS 芯片编程，使 CMOS 芯片每隔 1ms 产生一次中断请求 IRQ8，相应的中断向量是 0x70。PC／AT 机的 BIOS 中提供了中断 0x70 的处理程序，它使计数器减 1，当减到 0 时就设置一个标志位。

12.8.3 DOS 的时间 / 日历服务

除 Date 和 Time 命令外，DOS 还提供了 4 个系统调用，用来设置／读取日期和时间。相应于这 4 个系统调用，Turbo C 提供了 4 个函数：

系统调用号	功能	相应的 TC 函数
2a	取日期	void getdate(struct date *date)
2b	置日期	void setdate(struct date *date)
2c	取时间	void gettime(struct time *time)
2d	置时间	void settime(struct time *time)

这些 Turbo C 函数的用法以及其数据结构，请参见 7.9 节。

12.8.4 延迟函数

Turbo C 中提供了两个延迟函数：

```
void sleep(unsigned seconds);
void delay(unsigned milliseconds);
```

函数 sleep 的延时单位是 s，用于秒级以上的延时。函数 delay 的延时单位是 ms，但实践证明，它的精度仍然是一个"tick"，即约 55ms。

Turbo C Tools 也提供了一个延迟函数：

```
unsigned utsleep(unsigned ticks);
```

延时过程中该函数可能被中断，可能不能及时从中断返回，它通过返回值报告实际延迟的"tick"数，而不一定是所设定的值。

12.8.5 声音函数

8253 定时器芯片的输出也连到了扬声器上，通过对 8253 编程可以产生声音，遗憾的是，BIOS 和 DOS 都没有提供任何有关声音的服务。Turbo C 提供了两个声音函数：

```
void sound(unsigned frequency);
void nosound(void);
```

函数 sound 使扬声器发出频率为 frequency 的声音，单位是 Hz，直到执行了 no sound 函数。为了控制发声时间长短，必须使用延迟函数。

```
sound (hz);
delay (ms);
nosound;
```

Turbo C 中还提供了另一个发声函数 beep，这个函数实际上是一个宏：

```
#define heep putchra('\a')
```

当 putchar 函数要求控制台设备显示 ASCII 码 7('\a')时，为控制台设备(CON:)提供的 DOS 设备驱动程序就会产生一声响声，以作为一个警告或一个提示。

12.9 串行通信服务

在 PC 机上，硬件对串行口的数目几乎没有限制，但 BIOS 和 DOS 只支持串行口 COM1 和 COM2。DOS 把其中的一个指定为标准辅助设备 AUX，通常这个串行口是 COM1，但也可以用 DOS 的 MODE 命令去改变。

BIOS 对串行口的支持是通过中断 0X14H 进行的，共支持 4 个服务：设置串行口参数，发送 1 个字节，接受 1 个字节，查看串行口的状态。Turbo C 中提供了一个相应的函数：

```
int bioscom(int cmd, char abyte, int port);
```

(1) cmd＝0，设置串行口的参数。此时参数 abyte 可以是下列值的逻辑"或"：

0x02 7个数据位
0x03 8个数据位

0x00 1个停止位

0x00 110波特率
0x20 150波特率
0x40 300波特率
0x60 600波特率

0x04	2个停止位	0x80	1200波特率
		0xA0	2400波特率
0x00	不校验	0xC0	4800波特率
0x08	奇校验	0xE0	9600波特率
0x18	偶校验		

(2) cmd＝1，发送 1 个字节

(3) cmd＝2，接收 1 个字节

(4) cmd＝3，返回通信口的当前状态

对于所有的 cmd，返回值的高 8 位都是反映错误状态；对于无错误的 cmd＝2，低 8 位是所接收的字节，对于所有其它情况也是反映状态。

15	超时错	7	接收线信号检测
14	发送移位寄存器空	6	振铃指示
13	发送保存寄存器空	5	数据设备准备好
12	检测到一个断点	4	清除发送
11	帧错	3	"接收线信号检测"发生变化
10	奇偶校验错	2	"振铃指示"的下降沿
9	溢出错	1	"数据设备准备好"发生变化
8	数据准备好	0	"清除发送信号"发生变化

在启动机器时，DOS 初始化 AUX＝COM1:，如果安装了串行口，则选择 2400 波特率，8 个数据位，不进行奇偶校验，1 个停止位。通过 MODE 命令可以改变这些参数。在 DOS 的系统调用中，只有调用 4(发送 1 个字节)和调用 3(接收 1 个字节)牵涉到串行口，不能报告任何状态和错误信息。也就是说，DOS 对串行口的支持比 BIOS 还少。Turbo C 中也没有相应的函数。

注意，无论是 BIOS 和 DOS，还是 Turbo C 和 Turbo C Tools，都没有提供中断驱动的串行通信功能，都必须采取查询策略进行串行通信。

12.10 错误处理服务

12.10.1 DOS 怎样报告错误

在早期的 DOS 版本中，报告错误的办法是彼此不同的。采用 FCB 的早期版本，通过返回值 AL＝ff 报告发生了一个错误。在 DOS2.0 版中，通过置位进位标志位报告发生了一个错误。这些 DOS 版本的错误码都是通过某些寄存器来报告的。DOS 3.0 以后的版本提供了一个统一的错误报告和诊断系统。一旦检测到一个错误，则可以通过系统调用 0x59 得到许多信息，AX 返回的是扩展的错误码，BH 返回的是错误的分类，BL 返回的是建议采取何种行动，CH 返回的是错误发生的位置.

错误码(AX):

0	没有错误	22	不可识别的命令
1	无效的系统调用号	23	CRC 数据错
2	文件未找到	24	坏的请求结构长度
3	路径未找到	25	寻道错

4 没有文件柄
5 拒绝存取
6 无效的文件柄
7 内存控制块被破坏
8 内存不够
9 无效的内存块地址
10 无效的环境
11 无效的格式
12 无效存取码
13 无效数据
14 内部错(保留)
15 无效的盘号
16 不可能删除当前目录
17 设备名不同
18 没有匹配文件
19 软盘片是写保护的
20 不可识别的设备
21 盘未准备好
26 未知的介质类型
27 扇区未找到
28 没有打印纸
29 写错(盘)
30 读错(盘)
31 一般的失败
32 非法共享
33 非法上锁
34 无效盘改变
35 没有文件控制块
36 ~ 70 保留
80 文件存在
81 保留
82 不可能执行
83 执行 INT24 时失败

错误类型:

1 资源用尽
2 临时情况
3 授权
4 内部
5 硬件失败
6 系统失败
7 应用程序错
8 未找到
9 坏格式
10 已上锁
11 介质
12 已存在
13 未知

建议采取的行动:

1 重试
2 延迟，重试
3 由用户定
4 流产
5 立即退出
6 忽略
7 在用户介入后重试

错误位置码:

1 未知
2 块设备
3 保留
4 串行设备
5 内存

12.10.2 Turbo C 库函数的错误报告特性

当一个 Turbo C 库函数执行一个 DOS 系统调用而报告错误时，库函数将相应地设置全局变量__doserrno。这些库函数及其它不执行 DOS 系统调用的库函数也都设置全局变量 errno，errno 用于模拟 UNIX 操作系统的错误报告系统。__doserrno 和 errno 的某些号码一致，但彼此都包含一些对方不存在的错误号码。下面将要讲到的几个 Turbo C 报告错误的函数用的都是 errno，不是__doserrno。Turbo C 中还提供了另外两个全局变量：

```
char   * sys__errlist[ ]
int    sys__nerr
```

sys_errlist 是字符串数组，sys_nerr 是这个数组中字符串的数目。如果用户不加改变，数组 sys_errlist 则按 errno 来索引。但是，用户可以把数组中某个字符串改成他自己定义的字符串，即修改 errno 错误号的含义。

Turbo C 中提供了三个报告错误的函数：

```
char *strerror(int n);
char *_strerror(char *s);
void perror(char *s);
```

函数 strerror 返回一个字符串指针，这个字符串是相应于 errno 号的错误解释，但后面加了换行符。函数_strerror 返回的字符串是按下列格式组成的：字符串 s，冒号，空格，相应于当前 errno 号的错误解释，换行符。这两个函数的说明是在头文件 string.h 中。函数 perror 的说明是在头文件 stdio.h 中，它是往标准错误文件流(通常是控制台)输出一个字符串，这个字符串的格式与函数_strerror 返回的字符串的格式几乎一样。这两个函数使用户可以先输出他自己定义的字符串 s，例如出错的文件名和行号信息（见第 6 章)，然后再输出系统出错信息。用户自定义的字符串 s 不能超过 94 个字符。

下面这个演示程序说明了这些函数的用法。

```
#include <general.h>
#define call_perror printf("%32c", BLANK); perror(s)
void main( )
{
    char *s;
    errno=3;
    printf("errno              =%d\n", errno             );
    printf("strerror(errno) =%s\n", strerror(errno) );
    printf("S            _strerror(s)            perror(s)\n\n");
    s=NULL;
    printf("NULL       %s", _strerror(s));
    call_perror;
    s=EMPTYSTR;
    printf("EMPTY      %s", _strerror(s));
    call_perror;
    s="SS";
    printf("SS         %s", _strerror(s));
    call_perror;
    sys_errlist[errno]="New message";
    perror("sys_errlist[errno] altered");
    getch( );
}
```

此程序产生的输出如下所示：

```
errno            =3
strerror(errno) =Path not found

S            _strerror(s)            perror(s)
```

```
NULL        Path not found
                               (NULL): Path not found
EMPTY       Path not found
                               : Path not found
SS          SS: Path not found
                               SS: Path not found
sys_errlist[errno] altered: New message
```

12.10.3 致命错

当 DOS 系统调用企图执行输入／输出操作时，如果遇到了像软盘驱动器门未关好或没有打印纸之类的错误，则会当作致命错处理。一般来说，处理这类错误总是需要操作者干预。DOS 遇到这类错误时，它以说明这个错误条件的码装载各寄存器，然后执行中断 0x24。

0x24 处理程序很简单，它在显示出简短的错误描述信息后，询问操作者如何处理：

Abort, Retry, Fail?

如果选择 Retry，DOS 将试图再一次执行那个引起致命错的系统调用。如果选择 Fail，则终止这个系统调用，返回一个错误码。只要对这个错误码的分析和采取的行动是正确的，则不会引起严重后果。如果处理不妥，则软件可能崩溃，更糟糕的是可能导致貌似正确的结果。如果选择 Abort，DOS 将终止程序的运行，但终止处理是很不完善的，甚至未关闭已打开的文件。这种强行流产的做法有时可能是灾难性的。

用户可以用他自己的中断 0x24 处理程序代替 DOS 的中断 0x24 处理程序。但是，如果用户程序终止了，他的中断 0x24 处理程序却仍然留下来起作用，这就会产生问题，因为他的中断 0x24 处理程序不一定适合父程序或 DOS 的情况。为了解决这个问题，DOS 在执行任何程序时，总是把父程序的(当时的)中断 0x24 处理程序地址拷贝到新装入的子程序的 PSP 中。当终止子程序时，DOS 又把保存在子程序 PSP 中这份拷贝再恢复到中断向量 0x24 处。用户自己的中断 0x24 处理程序可以更加仔细地分析错误码，为操作者提供更加明确的选择，且当用户选择 Abort 时可能更妥善地终止程序的运行。

在 DOS 3.3 以前的版本中，致命错处理的第 3 个选择项不是 Fail 而是 Ignore。如果选择 Ignore，DOS 会让程序继续运行，虽然所需要的输入／输出操作并没有完成。这样做的后果很可能丢失数据。

12.10.4 <Ctrl_Break>和<Ctrl_C>

在 PC 机上，按<Ctrl_Break>所引起的结果基本上与按<Ctrl_C>相同。<Ctrl_C>是为了在只具有有限数目键的微机上运行 DOS 而开发的。

当 BIOS 的中断 9 处理程序检测到<Ctrl_C>或<Alt_3(小键盘)>时，它把 ASCII 码 3 放在键盘缓冲区，不采取任何进一步的行动，即这些键对 BIOS 来说没有任何特殊意义。然而，当检测到<Ctrl_Break>键时，它将调用中断 0x1b 处理程序，然后把 ASCII 码 0 放在键盘缓冲区。BIOS 本身也提供了中断 0x1b 的处理程序，但这个处理程序什么事也不做。DOS 在引导时修改了中断向量 0x1b，使之指向它的处理程序，这个程序也只

是设置一个<Ctrl_Break>标志。

当发生下列两种情况之一时，DOS 就会去检查这个<Ctrl_Break>标志，并检查键盘缓冲区内是否有 ASCII 码 3，即检查是否曾经按下了<Ctrl_C>:

· DOS 正执行某些字符输入／输出功能。

· DOS 正执行某些其它功能而 Break 开关为 ON。

Break 开关是通过 DOS 命令或在 CONFIG.SYS 文件中设置的，目的是控制 DOS 检查<Ctrl_Break>的频度。DOS 检查到这一个信号后，就调用中断 0x23 处理程序。

DOS 的 0x23 处理程序关闭所有已打开的文件柄(但不关闭以 FCB 方法打开的文件)，终止当前正在执行的程序。如果被终止的程序留下一些不合适的东西在机器内，例如修改了某些中断向量而这些修改后的中断向量又不适合随后运行的软件的需要，则可能引起灾难性的后果。

用户可以用他自己的中断 0x23 处理程序代替 DOS 的中断 0x23 处理程序。但是如果用户程序终止了，他的中断 0x23 处理程序却仍然留下来起作用，也会产生问题。其解决办法与 0x24 类同。

第 13 章　中断服务程序

本章讨论如何编写用户自己的中断服务程序(Interrupt Service Routines，简写为ISR)。编写中断服务程序需要对机器硬件、操作系统和汇编语言有比较深入的了解。Turbo C Tools 工具包中提供了这样一些函数，使程序员可以避开繁琐的汇编语言的细节，而完全通过 C 语言去实现。即使这样，仍然要求程序员对中断系统、中断与 DOS 和 BIOS 之间的关系有透彻的理解。

本章第 1 节是一般概念,对用任何高级语言写中断服务程序都可能遇到的问题作了一个概述。

第 2 节阐述 Turbo c Tools 是如何解决这些问题的，在这一节里，介绍了一些非常重要的子程序，通过一个最简单的例子说明中断服务程序的大致框架，以后在这个框架的基础上就可以编写出实用的中断服务程序。

本章第 3 节介绍了三个演示例子。第 1 个例子是编写用户自己的“tick”中断服务程序，它周期性地使扬声器发声。中断服务程序本身是很简单的，但因为发声的时间超过一个“tick”的时间长度，所以必须解决这样一个问题：在发声的同时允许中断服务程序本身又被同一中断源所中断，即中断服务程序应是可递归调用的。第 2 个例子是一个键盘中断服务程序，它检测是否同时按下了<A>和<J>键，并对此做出反应。这个例子与第 1 个例子相反，它不可能递归地中断其本身，所以这方面就简单一些，但它的服务却是极其复杂的。键盘中断是程序员们最经常遇到的任务，这个例子提供的有关管理键盘缓冲区等许多技术细节对程序员可能是很有帮助的。第 3 个例子是 prtsc 服务程序，它发送格式化的屏幕硬拷贝到一个文件。这个例子牵涉到 DOS 文件服务。由于 DOS 的单用户单任务性质，所以这个例子会遇到一些严重困难。尽管为克服这些困难已经有一些很吸引人的技术，但解决得仍然不够彻底，仍然不能完全令人满意。

本章第 4 节介绍了 Turbo C Tools 的插入服务技术。所谓插入服务，就是插在别的程序的执行间隙内，且这个间隙是允许利用的，执行某些服务程序。插入服务技术比较彻底地解决了 DOS 的不可重入问题，大大提高了用户中断服务程序运行的安全可靠性。

本章最后一节介绍了 Turbo C 对用户中断服务程序的支持，与 Turbo C Tools 比较起来，这种支持是很简单的，但使用起来更容易一些。

13.1　一般概念

在讲述如何编写中断服务程序之前，先来看看在什么情况下需要编写中断服务程序，这对理解一些基本概念和实现中的困难会有所帮助。编写中断服务程序一般来自如下 4 个方面的要求。

第 1 类，对某些设备提供实时服务，其典型例子就是中断 0xc，它是响应串行口

COM1 在 IRQ4 中断请求线上发出的中断请求而产生的硬件中断。对于这一类中断，效率是最主要的考虑因素。定时器中断 8 和键盘中断 9 也属于这一类中断。用户如果在 PC 机上扩充了其它硬件设备，并且这些设备具有中断机制，则这些中断都属于这一类。

第 2 类，由系统软件产生，使用户有机会去执行某些服务。0x1c 就是这样的中断。BIOS 中的 0x1c 中断处理程序是空的，只有一条 IRET 指令，但是用户也可以修改中断向量，编写自己的 0x1c 中断服务程序，使之每隔一定时间去做某件事。更多的情况是 BIOS 或 DOS 中原来没有的中断（通过这些中断为系统提供服务），如 EMS 用的中断 0x67，XMS 用的中断 0x15,以及用户定义的任何软中断，都应属于这一类。对于这一类中断，效率也是考虑的主要因素。

第 3 类，对已发生的错误进行诊断、补救或终止。这类中断可能是硬中断，也可能是软中断。某些 BIOS 和 DOS 中断归并到这一类更合适一些。例如，中断 9 在响应每次击键行为时，都会去检查是否为<Ctrl__Break>键。如果是，则调用中断 0x1b。BIOS 中的中断 0x1b 处理程序是空的，用户可以编写自己的中断 0x1B 处理程序，以响应<Ctrl__Break>键。属于这一类中断的还有 0x24，每当遇到错误并需要用户干预时就会产生这个中断。如打印机无纸，软盘片写保护等等。

第 4 类，修改和扩充现有的 BIOS 或 DOS 的功能。例如修改中断 0x16，使它返回的信息有所不同。再例如，有的机器上安装的高容量软盘（可高达 20MB），而普通 DOS 和 BIOS 都不支持这类软盘。当用 DOS 系统调用 0x44 子功能 8 去取这类软盘的信息时，返回来的信息可能说明该软盘是不可换介质。当用 BACKUP 去后备文件并且盘已满时，就会造成后备工作终止而不是提示用户插入新的盘片。解决这个问题的办法之一就是修改 DOS 的系统调用，即中断 0x21H。

编写中断服务程序的困难主要来自如下四个方面：

· 过滤问题

· 优先权问题

· 重入问题

· 高级语言问题

过滤问题较易处理。如前所述，有时仅需对现有中断服务程序进行部分修改或补充，而不是完全替换。这只要在安装时把原来的中断服务程序入口地址先保存起来，以后每次中断时，在做完新的处理后，再跳转到原来的中断服务程序继续处理。

处理中断优先权的关键是要弄清楚各中断的优先顺序，什么时候允许可屏蔽中断，什么时候不允许。例如，软中断指令本身是禁止可屏蔽中断的，因此软中断服务程序必须重新打开中断，否则就有可能破坏时钟，丢失通信数据和键盘输入数据。再如，在硬中断服务程序中，最后必须发出一个 EOI 信号(往口地址 0x20 写 1 字节 0x20)，以便告诉中断控制器可以允许同级和更低级的可屏蔽中断源发出中断。另一方面，如果程序的某段必须保证不被其它可屏蔽中断打断，则应该用 CLI 和 STI 指令来关闭和再打开中断。

重入(或者说递归)在某些情况下是必须的。例如，如果通过中断 0x1c 来定时发出闹钟声，由于闹钟叫唤的时间超过一个“tick”，所以在闹钟叫唤时必须允许同一个中断源再次发出中断。更一般的情况是，各中断必须能中断彼此的中断服务程序。为了允许重入，每个中断服务程序都必须有足够的堆栈空间。中断服务程序不能假设被中断的程序可以提

供足够大的堆栈空间，而必须由自己开辟。在另一些情况下，重入应该加以禁止。例如，在打印屏幕的过程中(中断5)不应该再次要求打印屏幕，在解释新的键盘宏定义的过程中不应该再接受键盘输入并解释。

最难解决的是DOS的重入问题。DOS本来是一个单用户单任务系统，是不可重入的。当执行DOS系统调用时，DOS常把一些必要的信息存放在它的全局变量中。如果在执行过程中被一个中断源中断，而这个中断服务程序又使用DOS系统调用，从而改写了那些全局变量，这就会引起问题。

编写中断服务程序的最后一个困难对于任何一种高级语言都存在，而不仅仅是Turbo C。Turbo C是高度模块化的，可执行的基本单位是程序和函数。程序通过调用DOS服务而终止，函数通过return语句(最终是通过ret指令)返回。但是，中断处理程序应该通过iret指令返回，而高级语言一般都不提供这种返回办法。另外，中断处理程序的入口和出口参数一般都是通过寄存器传送的。为了解决这些问题，Turbo C中提供了一个新的函数类型interrupt。Turbo C Tools提供了许多重要函数，这是它相对于Turbo C最重要的改进。本章介绍的技术几乎全是由Turbo C Tools提供的。

13.2 用Turbo C Tools写中断服务程序

13.2.1 工作原理

在Turbo C Tools中，如果一个函数准备用作中断服务程序，则它的格式应该如下(假设函数的名字是isr):

```
void isr(ALLREG *register,ISRCTRL *isrblk,ISRMSG *s);
```

第1个参数是指向结构ALLREG的指针，结构的定义在头文件butil.h中，它综合实现了Turbo C的头文件dos.h中REGS、SREGS和REGPACK三个结构的定义。当中断发生时，这个结构被用来保存所有寄存器的内容。当结束中断服务程序时，再把保存的内容恢复到各寄存器中去，恢复被中断的现场。

```
typedef struct {
      unsigned char l;
      unsigned char h;
}HALFREGS;
typedef union {
      HALFREGS hl;
      unsigned x;
}DOUBLREG;
typedef struct {
      DOUBLREG ax,bx,cx,dx;
      unsigned si,di;
      unsigned ds,es,ss,cs;
      unsigned flags,bp,sp,ip;
}ALLREG;
```

第2个参数是指向结构ISRCTRL的指针。这个结构的定义在头文件bintrupt.h中，

它是一个中断服务程序的控制块，由函数 isinstal 或 isreep 初始化。内部的中断调度程序使用和维护这个结构的各字段，这些信息对于中断服务程序的运行非常重要，但中断服务程序本身不应去修改它们。下面先列出这个结构的定义。

```
struct isr_control
{
      unsigned fcall_opcode;
      void (far *isrdisp)( );
      unsigned iret_opcode;
      char far *isrstk;
      unsigned isrstksize;
      unsigned isrsp;
      unsigned isrds;
      unsigned isres;
      void (far *isr)(ALLREG *,ISRCTRL *,ISRMSG *);
      unsigned isrpsp;
      void far *prev_vec;
      unsigned level;
      unsigned limit;
      unsigned signature;
      unsigned sign2;
      char     ident[16];
      unsigned control;
      unsigned status;
      char     scratch[10];
};
```

第 3 个参数是指向结构 ISRMSG 的指针。这个结构的定义也在头文件 bintrupt.h 中，它用于中断调度程序和中断服务程序之间的通信。下面先列出这个结构的定义。

```
typedef struct {
      int           exit_style;
      unsigned   working_flags;
}ISRMSG;
```

一般的中断服务程序接收到这样 3 个结构后，如果没有必要，则不去对它们进行操作，即忽略它们的存在。但内部的调度程序用到了这 3 个结构，从而保证当返回到被中断程序时，被中断程序各寄存器的值不被改变。

除这 3 个结构外，Turbo C Tools 还有一个内部调度函数：

```
extern void isdispat(void);
```

这个函数是被中断程序和中断服务程序之间的介入者，它是用汇编语言写的，汇编成一个.obj 模块，包含在相应的 Turbo C Tools 库文件内。在安装一个中断服务程序时，安装程序会建立一个中断控制块，其中就包含这个内部调度程序的地址。

从结构 ISRCTRL 的定义中可以看出，它的第 1 个字段是由两条机器指令组成的：空操作指令 NOP 和对调度程序的远调用指令。空操作指令是为了保证整个结构处处都是

偶地址对齐的。远调用指令是由远调用指令操作码后跟内部调度程序 isdispat 的地址组成的。中断安装程序设置相应的中断向量的值，使它指向这个中断控制块。当该中断发生时，中断机制会把当时的标志寄存器内容以及被中断程序的返回地址压入堆栈。因为中断向量已指向这个控制块，所以下一条被执行的指令就是刚说过的那条远调用指令，执行这条指令的结果是把控制块内下一字段的地址压入同一堆栈，然后跳转到内部调度程序。

调度程序开始执行时，它从堆栈中知道被中断程序的返回地址和标志寄存器内容，也知道控制块的地址，从而可找到中断服务程序的地址。因为这个中间作用要求复杂的堆栈操作，还要求对段寄存器操作，所以调度程序先把被中断程序的寄存器值保存起来，再搬到结构 ALLREG 中，之后把这个结构发送到中断服务程序。中断服务程序完成后，调度程序必须再一次确保把结构 ALLREG(此时可能已修改了)的值搬回到保留区，并在行将返回到被中断程序前，将这个保留区的值重新设置到各寄存器中。

因此，调度程序做的第一件事就是把所有寄存器的内容压入堆栈（这个堆栈还是被中断程序的堆栈）。接着，对堆栈的内容稍做一些修改,使得从调度程序返回时不是返回到中断控制块，而是直接返回到被中断程序。然后，调度程序为中断服务程序建立一个新的堆栈（它不能假设被中断程序的堆栈足够满足服务程序的需要）。前面已经说过，中断服务程序需要 3 个入口参数，故在调用中断服务程序之前，还须准备这 3 个参数。为此，调度程序建立起 ALLREG 和 ISRMSG 结构变量 register 和 s，用保存在原堆栈中的各寄存器值设置 register。至于 ISRCTRL 结构变量 isrblk，中断安装程序已经建立并初始化了一份，现在也正在使用，调度程序只是在必要时对这个现有的 isrblk 进行某些修改。然后，按照函数调用的规则，把这 3 个结构变量的指针放到新建立的堆栈中，模拟 Turbo C 函数调用规则，通过如下方式调用中断服务程序：

isr (®ister, &isrblk, &s)

从中断服务程序返回时，调度程序必须对这 3 个结构变量进行处理，废弃掉为中断服务程序建立的堆栈，把 ALLREG 结构变量 register 的值设置到各寄存器中，返回到被中断程序。

中断发生时各程序之间的关系是：

被中断程序→中断控制块→调度程序→中断服务程序。

中断服务结束后各程序之间的关系是：

中断服务程序→调度程序→被中断程序。

此外，还有一个中断安装程序。中断安装程序除了建立和初始化中断控制块以外，还装进了调度程序和中断服务程序。中断安装程序也驻留一部分在内存，并建立和初始化了一些全局变量，通过这些变量与中断服务程序通信。

中断控制块结构 ISRCTRL 中还应包括调度程序的地址和中断服务程序的地址。

中断服务程序在存取中断安装程序建立的全局变量区时，是假设 DS 和 ES 寄存器已为它正确地设置好，并已指向这个变量区。但实际上往往并不如此，如果不加特殊处理，进入中断服务程序时 DS 和 ES 寄存器的值将仍然是被中断程序的 DS 和 ES 的值。因此，中断安装程序必须把全局变量区的段地址值 DS 和 ES 保存到中断控制块中，每次进入中断服务程序之前，由调度程序把这两个保存值设置到 DS 和 ES 寄存器中。一旦安装后，这两个值就应是只读的。

为了对中断进行过滤，每个中断服务程序还要调用在它被安装之前原来的中断服务程序。原来的中断服务程序的地址也保存在控制块的字段 prev__vec 内。从这一点也可看出，中断控制块应是全局变量，应让中断服务程序可以存取到。这就是为什么要把中断控制块作为一个参数传给中断服务程序的原因。如果要撤消已安装的新中断服务程序，也需要用到原来中断服务程序的地址。

为了唯一识别每个中断服务程序，控制块中还包含 3 个识别字段。第 1 个是字段 signature，它是 Blaise 公司的“签名”。第 2 个是字段 sign2，它是字段 signature 的补码。第 3 个是字段 ident，它是由 16 个字符组成的字符串，是这个中断服务程序的名字。凭这些识别字段，用户程序可以知道一个中断向量是否指向一个中断控制块，如果是，它的名字是什么，从而避免重复安装，也可以避免在撤消中断服务程序时弄错“对象”。

控制块的字段 isrpsp 内保存的是中断安装程序的 PSP 段地址，当撤消一个中断服务程序和更新 DOS 的当前执行进程的记录时，就要用到这个段地址。

控制块的字段 iret__opcode、control、status 和 scratch 是从早期的版本中继承过来的，中断安装程序初始化它们，但调度程序和中断服务程序都不使用这些字段。

控制块内剩下的 isrstk、isrsp、isrstksize、level 和 limit 都是与堆栈有关的字段。调度程序必须解决递归问题。中断服务程序执行过程中，有可能又被同一中断源所中断，又要进入中断服务程序。而安装程序只装入了一份中断服务程序，解决的办法是由调度程序为每次对中断服务程序的调用分别建立一个堆栈，字段 isrstksize 说明每层堆栈的大小，单位是字节，limit 是允许的嵌套次数，isrstk 指向为堆栈保留的空间，其大小应大于 isrstksize * limit 字节。字段 level 是当前的嵌套层数，isrsp 是当前的堆栈指针。粗略地讲，这几个字段之间应符合下列关系：

isrsp=isrstk+level * isrstksize

这个等式的准确解释取决于 isrstk 是怎样分配的和当前所用的编译模式，但无论在什么情况下，字段 isrsp 可以根据其它值计算出来，因而可能导致错误的理解。安装程序初始化 level 为 0，并相应的初始化 isrsp。

调用中断服务程序之前，调度程序会检查下列不等式是否成立：

level<limit

如果不成立，则意味着堆栈已用完，不提供任何服务，而直接返回到被中断程序。但是，如果这次中断是硬件中断，则由于未经过中断服务程序，也就没有向 8259 芯片发出 EOI 信号，使得同级及更低级的中断得不到系统的处理，从而可能引起崩溃。

如果不等式成立，说明还有堆栈空间，则调度程序使字段 level 的值加 1，相应调整字段 isrsp 的值，根据 isrstk 和 isrsp 计算出新的堆栈段地址和堆栈指针，并装入寄存器对 SS:SP，建立一个新的堆栈。

每发生一次中断，就要进入调度程序一次，Turbo C Tools 怎样解决调度程序本身的重入问题呢？调度程序的局部变量总是与对中断服务程序的某一特定调用关联的，并保存在相应的一层堆栈上。每次发生中断进入调度程序时(称为第 1 次进入)，它首先把被中断程序的各寄存器值保存在当时正被使用的堆栈上，把指向这个保留区的指针保存在寄存器 BX 中，然后建立起新的一层堆栈，并使它开始起作用。调度程序用 BX 作为指针，把保存在原堆栈上的值拷贝到新堆栈上的 ALLREG 结构内，建立起所要求的 ISRMSG 结

构，把与本次中断有关的所有其它变量和寄存器的值保存到新堆栈上。接着，调度程序把控制块内字段 isrds 和 isres(是由安装程序初始化的)的值装入 DS 和 ES 寄存器。最后调度程序依次把指向结构 ISRMSG、控制块 ISRCTRL、结构 ALLREG 的指针压入堆栈，调用中断服务程序。

当从中断服务程序返回后，再次进入调度程序(称为第 2 次进入)，调度程序从堆栈中弹出所有它保留在那儿的信息，恢复所要求的寄存器和局部变量的值，包括 ISRMSG 结构的内容。用刚恢复的各寄存器内的地址，调度程序从堆栈中弹出 ALLREG 结构的值(也许已被中断服务程序所修改)，并把这些值压回到上一层堆栈中第 1 次进入调度程序时保存各相应寄存器的保留区，然后恢复 SS:SP 寄存器为当初启用新堆栈之前的值。至此，又启用了原来的上一层堆栈。调度程序使字段 level 的值减 1，相应地调整字段 isrsp 的值，从而废弃新的堆栈。最后，弹出原堆栈中的内容，放在相应寄存器中，恢复 SS:SP 为第 1 次进入调度程序时的值，通过执行一条 iret 指令返回到被中断程序。

结构 ISRMSG 用于说明如何返回标志寄存器的内容。调度程序在调用中断服务程序之前，设置 exit__style 为 0，把被中断程序的标志寄存器的内容保存在字段 working__flags 内。从中断服务程序返回后，调度程序检查字段 exit__style 的值是否被改变。如果仍然是 0，则按上一段所述过程进行处理。如果已被中断服务程序修改为 1，则用字段 working__flays 内的值(可能已被中断服务程序修改了)恢复标志寄存器的内容，通过远返回指令 retf（而不是通过中断返回指令 iret）返回到被中断程序，使中断服务程序可通过标志寄存器向被中断程序传回处理信息。下面这个小例子就是从中断服务程序 cbreak 返回时，进位标志位被置为 1。

```
#include <bintrupt.h>
void cdecl cbreak(ALLREG *, ISRCTRL *, ISRMSG *);
void cbreak(pregs,pisrblk,pmsg)
ALLREG  *pregs;
ISRCTRL  *pisrblk;
ISRMSG  *pmsg;
{
    extern unsigned long count;
    count++;
    pmsg->working__flags |= CF__FLAG;
    pmsg->exit__style = IEXIT__RETF;
}
```

13.2.2 安装和驻留

为了便于用户编写自己的中断服务程序，Turbo C Tools 6.0 版提供了下列函数：

```
int       cdecl  iscall(void far *pisr,ALLREG *preg);
unsigned  cdecl  iscurprc(int option,unsigned newproc);
void      cdecl  isgetvec(int intype);
int       cdecl  isinstal(int intype,
                     void (*pfunc)(ALLREG *,ISRCTRL *,ISRMSG *),
                     const char *pident,ISRCTRL *pisrblk,
```

```
                        char  * pstack,int stksize,int stknum);
void        cdecl   isprep(void (* pfunc)(ALLREG  * ,ISRCTRL  * ,ISRMSG  * ),
                        const char  * pident,ISRCTRL  * pisrblk,
                        char  * pstack,int stksize,int stknum);
void        cdecl   isputvec(int intype,void far  * ptr);
int         cdecl   isremove(unsigned psp__seg);
int         cdecl   isreserv(size__t reserve,size__t blksize,
                        char  * * ppblock);
void        isresext(int excode);
ISRCTRL  far  * cdecl   issense(void far  * ptr,const char  * pident);
```

在这些函数中最重要的就是中断安装函数 isinstal。其中，intype 是中断向量号，pfunc 是指向中断服务程序的指针，pident 是 16 字节的标识名字符串，pisrblk 是指向中断控制块的指针，pstack 是指向堆栈空间的指针，其大小至少为 stksize * stknum 字节。调用者必须为中断控制块分配空间，由安装程序对它进行初始化。调用者也必须为堆栈分配空间。这两个空间在中断服务程序工作期间必须是静态的。

下面是一个中断服务程序的例子。

```
/*    prtsc1.dem   * /
#include  <bintrupt.h>
#include  <general.h>
void isr(ALLREG   * registers,ISRCTRL  * isrblk,ISRMSG  * s)
{
     beep;
}
#define STACKSIZE 1024
#define MAXDEPTH      1
char     stacks[STACKSIZE * MAXDEPTH];
ISRCTRL controlblock;
void main(int paramcount,char  * paramstr[ ])
{
     int n = strtol(paramstr[1],(char  * * ) NULL,16);
     printf("Touch  <CR>  to install ISR   %d,");
     printf("or  <Ctrl-Break>  to terminate.",n);
     getch( );
     isinstal(n,isr,paramstr[2],&controlblock,
              stacks,STACKSIZE,MAXDEPTH);
     isresext(0);
}
```

这个例子中的中断服务程序 isr 很简单，它只是调用 Turbo C 的 beep 函数发出声音。这个例子是通过命令行接受中断向量号和标识名的，假设这个演示程序的名字是 prtsc1.dem，中断向量是 5，标识名是"ABCDEFGHIJKLMNO"则执行完下列命令：

prtsc1   5   ABCDEFGHIJKLMNO

后，中断服务程序就安装好了。以后凡在键盘上按<Shift__PrtSc>键，将不再是打印屏幕，而是发生一阵响声。但是，现在不能恢复被 isinstal 所修改的中断向量的值，除非重新引导。以后将讲到如何恢复的问题。

这个演示程序首先调用 Turbo C 的 strtol 函数，把命令行的中断向量号(十六进制字符串)转换为长整数，再转换为整数。演示程序中还用到了 Turbo C Tools 提供的另一个函数：

```
void isresext(int errorlevel);
```

这个函数的功能与 Turbo C 的 keep 函数类似，它关闭所有已打开的文件，终止当前程序，设置退出码，但程序保持驻留在内存中。父程序可以通过 DOS 系统调用 0x4d 取得这个退出码，也可以在批命令文件中通过全局变量 ERRORLEVEL 取得这个退出码，退出码应在 0～255 之间，一般用 0 表示成功。

函数 isprep 与函数 isinstal 的参数几乎相同，只是少了一个中断向量号。这是因为，这个函数只是准备一个中断控制块，而不安装这个中断服务程序，以后可以通过 iscall 函数来安装，如下例所示：

```
#include <stdlib.h>
#include <bintrupt.h>
#define  MY_STK_SIZE       2048
#define  MY_NUM_STKS        1
void my_isr(ALLREG *pregs,ISRCTRL *pisrblk,ISRMSG *pmsg)
{
     static ISRCTRL isrblk;
     char *pstacks;
     ALLREG regs;
     if (NIL == (pstacks=malloc(MY_STK_SIZE*MY_NUM_STKS)))
          exit(3);
     isprep(my_isr,"THIS IS MY ISR.",&isrblk,pstacks,
                    MY_STK_SIZE,MY_NUM_STKS);
     iscall((void far *)&isrblk,&regs);
     free(pstacks);
}
```

函数 isreserv 是为一个中断服务程序预先留出它所需要的堆栈空间和动态内存。参数 reserve 说明预留的动态内存的大小，单位是字节，以后在中断服务程序中可用 malloc 函数对这块内存进行动态分配。参数 blksize 说明预留的堆栈大小，单位也是字节。参数 ppblock 是指针的地址，如下例所示：

```
#include <bintrupt.h>
#define  MY_INTYPE        0X60;
#define  MY_STK_SIZE        2048;
#define  MY_NUM_STKS         5;
#define  MY_RESERVE        20000;
void my_isr(ALLREG *pregs,ISRCTRL *pisrblk,ISRMSG *pisrmsg)
    {
    static ISRCTRL isrblk;
    char              *pstacks;
    if(IS_OK!=isreserv(MY_RESERVE,MY_STK_SIZE*MY_NUM_STKS,
                    pstacks)
         exit(3);
```

```
    isinstal(MY__INTYPE,my__isr,"THIS IS MY ISR.",&isrblk,
             pstacks,MY__STK__SIZE,MY__NUM__STKS);
}
```

函数 isgetvec 返回一个中断向量的值，即一个中断服务程序的入口地址。函数 isputvec 则设置一个中断向量的值。它们与 Turbo C 的 getvect 和 putvect 函数相类似，但不牵涉到 Turbo C 的中断函数类型说明符 interrupt。

13.2.3 过滤

在很多情况下，用户自己的中断服务程序只是对原来的中断服务程序进行一些修改或补充，而不是彻底改写，这时就要用到过滤技术。前面讲过的 isinstal 安装函数只负责把中断向量原来的值放到控制块中的字段 prev__vec 内，但要使用原来的中断服务程序却有点问题，因为它要求用软中断指令进入而不能用 call 指令进入，即不能通过一个 C 语言调用规则进入。当然，可以把 prev__vec 的值存入一个临时中断向量中，然后通过 Turbo C 的 geninterrupt 函数使用这个中断。如果其它软件也需要那个中断向量，特别是在用户程序可以重新获得控制权恢复它原来的值以前，这种办法就不总是可行的。

Turbo C Tools 的函数:

```
int iscall(void far *pisr,ALLREG *preg);
```

提供了一种解决办法，它把各寄存器值存入结构 ALLREG 中，然后通过一个远调用指令进入所指定的中断服务程序，返回后再把结构 ALLREG 中的值赋回给各寄存器。中断和调用之间的差别就由这个函数去处理。如果把 13.2.2 演示程序的 isr 函数修改成

```
void isr(ALLREG *regsters,ISRCTRL *isrblk,ISRMSG *s)
{
    iscall(isrblk->prev__vec,regsters);
    beep;
}
```

则在执行后，每次按<Shift__PritSc>键，不但发出一阵响声，还会得到一份屏幕硬拷贝。

13.2.4 探测和撤消

中断控制块的字段 ident 是每个中断服务程序的唯一标识名字符串。使用这个标识名，可以避免安装同一中断服务程序的多份拷贝，也可从控制块中得到足够的信息来撤消用户的中断服务程序，恢复原来的中断服务程序。

函数 issense 根据所给定的中断向量值和标识名，判断具有这个标识名的中断服务程序是否已安装，如果已安装，则返回指向控制块的指针。否则返回空指针 FARNIL。

函数 isremove 根据所给定中断服务程序的 PSP 段地址，撤消这个中断服务程序。如果撤消成功，则返回值为 0，否则返回一个错误码。这个函数从所给定的 PSP 开始，沿 DOS 的内存控制块链(见 10.2 节)向上(内存高地址方向)寻找所有属于这个 PSP 的内存块

(包括环境块和程序块)，释放这些内存块，下面就是这样一个简短例子：

```
#include <general.h>
#include <bintrupt.h>
int rstorisr(int n,char *id)
{
    ISRCTRL far *isrblk;
    isrblk=issense(isgetvec(n),id);
    if (FARNULL==isrblk) return 2;
    isputvec(n,isrblk->prev_vec);
    return isremove(isrblk->isrpsp);
}
```

这个程序的返回值是：

0 成功。

2 没有这样一个中断服务程序。

7 内存控制块 MCB 被破坏。

9 无效的内存控制块MCB，可能所给定的PSP段地址不是一个驻留程序。

其中 0、7 和 9 是函数 isremove 返回的。

注意，同一安装程序(不是安装函数 isinstal)可能安装了多个用户中断服务程序。在这种情况下，这个例子将是撤消这个安装程序所安装的所有中断服务程序，而不仅是撤消具有指定中断向量和标识名的那个中断服务程序。

13.2.5 其它

对于某些 DOS 服务，要求发出服务请求的程序是“当前执行程序”（或称为“当前进程”)，否则可能得不到正确的服务。DOS 是凭 PSP 段地址来区分各进程的。每当 DOS(或其它父程序)装入或终止一个程序时，会相应地更改“当前进程”的记录，重新认定新的“当前进程”。但是，如果“当前进程”被中断，由中断服务程序接管运行，DOS 却不会自动更改“当前进程”记录。也就是说，当中断服务程序执行时，“当前进程”不是中断服务程序，而是被中断程序，从而可能不能正确执行某些 DOS 服务。为了得到 DOS 的可靠服务，中断服务程序必须：

(1) 拷贝和记住当前 PSP 段地址。

(2) 以它自己的 PSP 段地址作为当前 PSP 段地址。

(3) 获得 DOS 的可靠服务。

(4) 在返回被中断程序前恢复当前 PSP 段地址。

为了帮助进行这些工作，Turbo C Tools 提供了函数 iscurprc。选择项 option 可以是下列两种情况之一：

IS_SETPROC(1) 设置新的“当前进程”，参数newproc就是新的“当前进程”的PSP段地址，返回的是原来的"当前进程"的 PSP 段地址。

IS_RETPROC(0) 仅报告"当前进程"的PSP段地址。

这个函数的说明是在头文件 bintrupt.h 中，它用到了 DOS 系统调用 0x50 和 0x51,

但这两个系统调用在 DOS 技术手册中都没有公布。

13.3 中断服务程序实例

13.3.1 [例 1]: 周期性地发声

这个示例程序每隔 10s 以 440Hz 的频率发声 0.5s。

程序 timerisr.c:

```
#include <general.h>
#include <bintrupt.h>
int rstorisr(int n,char *id)
{
      ISRCTRL far *isrblk;
      isrblk = issense(isgetvec(n),id);
      if (FARNULL==isrblk) return 2;
      isputvec(n,isrblk->prev__vec);
      return isremove(isrblk->isrpsp);
}
int        countdown;
#define  TENSECONDS       182
void timerisr(ALLREG *registers,ISRCTRL *isrblk,ISRMSG *s)
{
    iscall(isrblk->prev__vec,registers);
    if (--countdown)return;
    countdown = TENSECONDS;
    outportb(INTA00,EOI);
    sound(440);
    delay(500);
    nosound( );
}

#define  USERTICKINTERRUPT 0X1C
#define  ID      "10 Sec Sounder"
#define  STACKSIZE        1024
#define  MAXDEPTH            2
char      stacks[STACKSIZE * MAXDEPTH];
ISRCTRL   controlblock;

void main( )
{
    if (rstorisr(USERTICKINTERRUPT,ID) == 2) {
        countdown = TENSECONDS;
        isinstal(USERTICKINTERRUPT,timerisr,ID,&controlblock,
                 stacks,STACKSIZE,MAXDEPTH);
        isresext(0);
    }
}
```

这个程序有一点值得注意。中断服务程序的第1条语句是iscall，它调用原来的中断服务程序，首先保证按原来的处理办法处理，这与一般人用汇编语言写中断服务程序的习惯很不一样。接着，程序对计数器减1，若结果不为0，则立刻返回，否则重置计数器。程序的第4条语句是往口地址0x20输出1个字节0x20，以允许同级及更低优先级的中断可以得到服务。前4条语句的执行时间不会超过一个"tick"，所以下一次"tick"中断一定会得到响应和处理。程序的最后3条语句就是发声。在发声期间，"tick"中断照样可以处理。从分析中可以看出，这个中断服务程序必须是可以递归的，但递归不会超过两层，因而在安装时设置最大层数MAXDEPTH为2。

主函数main中用到了13.2.4中的程序rstorisr，它来检查此中断服务程序是否已经安装。只有在以前没有安装的情况下，才初始化计数器、安装中断服务程序并驻留在内存。

13.3.2 [例2]：检测A和J键的同时按下

在各种汉化DOS中，都需要用键的组合在各种输入法之间进行切换。例如经常用<Alt_F2>切换到五笔字形输入状态，用<Alt_F6>切换到ASCII码输入状态。但是，<Alt_F2>和<Alt_F6>这样键的组合很难做到"盲打"，从而大大影响输入速度。本例用A键和J键的同时按下来作为一个乒乓开关，同时按一次将切换到五笔字型输入状态，再同时按一次将切换到ASCII码输入状态，但不影响任何其他击键。因为A键和J键容易"盲打"，大大加快了输入速度。

```
/*      f2f6.c   */
#include  <bintrupt.h>
#include  <general.h>
boolean   switchh = FALSE;
boolean   adown = FALSE;
boolean   jdown = FALSE;
unsigned  firstkey = 0;
#define   altf2 0x6900
#define   altf6 0x6d00
#define   kbd_int        9
#define   id             "yinyinyinyinyin"
#define   stacksize      1024
#define   maxdepth        1
char      stack[stacksize * maxdepth];
ISRCTRL     controlblock;
void      cdecl kbdisr(ALLREG *, ISRCTRL *, ISRMSG *);
void      getfirstkey(void);
void      putfirstkey(void);
void      putfirstkey1(unsigned );
void      cleanup(int );
#define   kb_data 0x60
#define   kb_ctrl 0x61
#define   prtsc          0x37
#define   scrolllock     0x46
#define   capslock       0x3a
#define   numlock        0x45
```

```
#define   p              0x1e
#define   j              0x24
#define   kbbuffer       0x001e
#define   kbbufferend    0x003c
#define   biosdata       0x0040
#define   bufferhead (unsigned far * )MK_FP(biosdata, 0x001a)
#define   buffertail (unsigned far * )MK_FP(biosdata, 0x001c)
#define   keystates  (unsigned char far * )MK_FP(biosdata, 0x0017)

void main( )
{
     isinstal(kbd_int, kbdisr, id, &controlblock, stack,
               stacksize, maxdepth);
     isresext(0);
}

void cdecl kbdisr(ALLREG * registers,ISRCTRL * isrblk,ISRMSG * s)
{
     unsigned          formertail;
     unsigned char     code1;
     unsigned char     event;
     code1 = inportb(kb_ctrl);
     event = inportb(kb_data);
     if ((event = = prtsc) || (event = = scrolllock))
          cleanup(code1);
     else if (event = = capslock) {
          if ((( * keystates)&0x40) = = 0)
               * keystates = ( * keystates|0x40);
          else  * keystates = ( * keystates&0xbf);
          cleanup(code1);
     }
     else if (event = = numlock) {
          if ((( * keystates)&0x20) = = 0)
               * keystates = ( * keystates|0x20);
          else  * keystates = ( * keystates&0xdf);
          cleanup(code1);
     }
     else {
          if ((event = = p) || (event = = j)) {
               if (event = = p) adown = TRUE;
               else jdown = TRUE;
               if (adown && jdown) {
                    if (switchh = = FALSE){
                         switchh = TRUE;
                         putfirstkey1(altf2);
                    }
                    else {
                         switchh = FALSE;
                         putfirstkey1(altf6);
                    }
                    adown = jdown = FALSE;
```

```
                    firstkey=0;
                    cleanup(code1);
                }
                else {
                    formertail= * buffertail;
                    iscall(isrblk->prev__vec, registers);
                    if (formertail != ( * buffertail)) getfirstkey( );
                }
            }
            else {
                event-=128;
                if ((event= =capslock) || (event= =prtsc) ||
                    (event= =numlock) || (event= =scrolllock))
                    cleanup(code1);
                else {
                    if ((event= =p) || (event= =j)) {
                        if (event= =p) adown=FALSE;
                        else jdown=FALSE;
                        putfirstkey( );
                        cleanup(code1);
                    }
                    else {
                        iscall(isrblk->prev__vec, registers);
                    }
                }
            }
        }
}
void putfirstkey(void)
{
    unsigned         next;
    if (!firstkey) return;
    next= * buffertail+2;
    if (next>kbbufferend) next=kbbuffer;
    if (next= =  * bufferhead) beep;
    else    {
        disable( );
        * (unsigned far * ) MK__FP(biosdata,  * buffertail)=firstkey;
        * buffertail=next;
        enable( );
    }
    firstkey=0;
}
void putfirstkey1(unsigned fkey)
{
    unsigned         next;
    if (!fkey) return;
    next= * buffertail+2;
    if (next>kbbufferend) next=kbbuffer;
    if (next= =  * bufferhead) beep;
    else    {
```

```
            disable( );
            *(unsigned far *) MK_FP(biosdata, *buffertail)=fkey;
            *buffertail=next;
            enable( );
        }
}
void getfirstkey(void)
{
      unsigned        previous;
      previous= *buffertail-2;
      if (previous<kbbuffer) previous=kbbufferend;
      disable( );
      firstkey= *(unsigned far *) MK_FP(biosdata, previous);
      *buffertail=previous;
      enable( );
}
void cleanup(int code1)
{
      outportb(kb_ctrl, code1|0x80);
      outportb(kb_ctrl, code1);
      outportb(INTA00, EOI);
}
```

这个例子中的中断服务程序是函数 kbdisr，因为它只处理<A>和<J>键同时按下的情况，因此也必须对中断进行过滤。

在每次击键事件（包括按下键和释放键）之后，键盘控制器都在中断请求线 IRQ1 上升起一个中断信号，同时把这次事件的事件码放在 CPU 的地址 KB_DATA=0x60 中。对于按下键，这个事件码就是键的扫描码。对于释放键，这个事件码就是键的扫描码再加 128，即最高位置 1。响应这个中断请求的中断向量是 9，通过 Turbo C Tools 的中断调度程序就进入到本例中的中断服务程序 kbdisr 中。虽然 kbdisr 是对中断进行过滤的，即它把大部分工作仍然留给 BIOS 中的键盘中断服务程序去做，但它不能立即调用 BIOS 的中断服务程序。因为一旦经过 BIOS，有关这次事件的信息就会被 BIOS 破坏。

这个中断服务程序 kbdisr 不但应检测键的按下，还要检测键的释放，但 BIOS 的中断服务程序只处理键的按下而忽略键的释放。因此，kbdisr 必须首先检测口地址 KB_ DATA 中的数据，看看它是什么事件。kbdisr 还用到了口地址 KB_CTRL = 0x61，这个口地址是可读可写的，通过其中的一位(保留其它位不动)向键盘控制器发送一个控制信号，告诉它这次事件已处理完毕。

如果事件码表示是按下了一个热键，则 kbdisr 把这个事件记录在全局变量 adown 或 jdown 中，如果这两个变量都为真，则说明<A>键和<J>键同时按下了。此时查看原来是在<Alt_F6>还是<Alt_F2>状态(即 ASCII 码输入还是五笔字型输入)，并改变为相反的状态。然后，置变量 adown 和 jdown 为假，清除 firstkey 全局变量为 0，调用 cleanup 函数做完善后工作后退出。当 kbdisr 检测到一个热键时，它必须记住完整的 16 位键盘码，包括 ASCII 码和扫描码，以便以后在发现这个键不是<A_J>热键对的第 1 个键时，可以恢复这个键变量。firstkey 就是用来存储这完整的 16 位键盘码的，它的初始

值是 0，表示还没有按下键。

kbdisr 在检测到一个热键之后，不能立刻断定它就是<A__J>热键对的第 1 个(也可能是一个普通的键)。究竟是什么，要待到下一键盘事件到来时才能确定。如果下一事件到来时证明这两个键确是一个热键对，则这两个键都不记录在键盘缓冲区，而应该往键盘缓冲区放一个<Alt__F2>或<Alt__F6>。如果下一事件到来时证明第 1 个键是一个普通键，则应该把这个键记录在键盘缓冲区。这就是说，必须把第 1 个热键记住，以备以后可以放到键盘缓冲区中。

乍一看，记住一个键的完整 16 位码是一件很简单的事，其实不然。这是因为，必须结合当前的移位键和锁定键状态，才能确定所按键的 ASCII 码。例如，如果按下的是 A 键，但这可能是大写字母 A 或小写字母 a，也可能是<Alt__A>、<Ctrl__A>或<Shift__A>等。最好的办法是让 BIOS 的键盘中断服务程序去做这个解释工作。程序中是通过调用 iscall 函数实现的。此时，口地址 KB__DATA 和 KB__CTRL 中的数据未被破坏(改变)，以便 BIOS 可以正确地进行解释。BIOS 把解释后的完整 16 位码放在键盘缓冲区中，除非键盘缓冲区已满。因为 BIOS 每次往键盘缓冲区放一个键时，会相应地修改键盘缓冲区尾指针，所以 kbdisr 可以通过检查尾指针是否改变来判断键盘缓冲区是否已满。如果已满，则这个键就丢失了，这种情况很难处理。但这样的情况极少发生，尾指针总是会发生变化的，kbdisr 可以调用 getfirstkey 函数，从键盘缓冲区把这个键的 16 位码取出放在全局变量 firstkey 中。然后，kbdisr 就可以返回，没有必要再调用 cleanup 函数去做善后工作，因为 BIOS 已经做完了这些工作。应该注意的是，不能通过 BIOS 的中断 0x16 从键盘缓冲区取出这个键的 16 位码，因为 BIOS 是从缓冲区的头而不是从缓冲区的尾取这 16 位码的。

如果 kbdisr 检测到的是释放一个热键，则它相应地使全局变量 adown 或 jdown 为假。如果刚释放的这个键在它被按下时曾经作为热键对的第 1 个而保存起来，则就应调用 putfirstkey 函数把这个保存的值送回到键盘缓冲区。然后调用 cleanup 函数做好善后工作。

若用户键入的既不是<A>也不是<J>，则简单地调用 BIOS 的键盘中断服务程序即可。

这个例子在实用过程中可能还会碰到一些问题，其中之一就是在某些机器上将妨碍 Capslock 锁定键的使用，有的还妨碍 Numlock 锁定键的使用。解决办法之一就是在中断服务程序 kbdisr 中对这两个锁定键也进行处理（而不让 BIOS 处理)。另外，这个中断服务程序驻留以后，在 DEBUG 下和不在 DEBUG 下的键盘行为可能稍有不同，在调试时应加以注意。

由于第 1 个热键按下后只是记录在案，并没有立即送往键盘缓冲区，所以按下 1 个热键后，无论是<A>还是<J>，必须释放后这个键才会在屏幕上显示出来，而所有其它键都是刚一按下就会在屏幕上显示。

13.3.3 [例 3]：发送格式化的输出

这个例子与 13.2 节已介绍过的 Prtsc1.dem 类似，也是打印屏幕。程序的名字是 Prtsc2.dem，所用到的中断服务函数是 prtsc2，处理打印屏幕的函数是 Prtsc。

函数 Prtsc2.dem:

```
/ *    prtsc2.dem    * /
#include  <dos.h>
#include  <butil.h>
#include  <general.h>
#include  <bintrupt.h>
#include  <fcntl.h>
#include  <io.h>
#include  <bvideo.h>

#define    PRTSCINTERRUPT         5
#define    ID                     "Screen Hardcopy"
#define    STACKSIZE              1024
#define    MAXDEPTH               2
#define    exists(f) (access((f),0) == = 0)
#define    ESCAT                  "\x1b\x40"
#define    SI                     "\xf"
#define    FF                     "\xc"
#define    CRLF                   "\r\n"
#define    RESET                  ESCAT
#define    COMPRESSED             SI
#define    ROWSPERPAGE            66
#define    COLUMNSPERPAGE         136
#define    MAXROWS                43
#define    MAXCOLUMNS             80
#define    HALFSECOND             9
#define    CONCERTA               440

int          busy=FALSE;
char         * filename;
int          rows,columns;
int          topmargin,leftmargin;
int          ignore;
int          file;
int          row;
char         buffer[MAXROWS * MAXCOLUMNS];
char         blanks[MAXCOLUMNS / 2];
char         stacks[STACKSIZE * MAXDEPTH];
ISRCTRL      controlblock;

void peep (void)
{
     sound(CONCERTA);
     utsleep(HALFSECOND);
     nosound( );
}

int  rstorisr(int n,char * id)
{
```

```
    ISRCTRL far * isrblk;
    isrblk = issense(isgetvec(n),id);
    if (FARNULL = = isrblk) return 2;
    isputvec(n,isrblk-> prev__vec);
    return isremove(isrblk-> isrpsp);
}

void prtsc(void) {
    busy = TRUE;
    rows = scrows( );
    ignore = scmode(&ignore,&columns,&ignore);
    virdrect(0,0,rows-1,columns-1,buffer,CHARS__ONLY);
    file = (!exists(filename)? __creat(filename,0) :
                        open(filename,O__WRONLY|O__APPEND));
    write(file,RESET, 2);
    write(file,COMPRESSED,1);
    topmargin          = (ROWSPERPAGE -rows) / 2;
    leftmargin = (COLUMNSPERPAGE-columns) / 2;
    memset(blanks,BLANK,leftmargin);
    for(row = 0;row < topmargin;++row)
        write(file,"\r\n",2);
    for(row = 0;row < rows;++row) {
        write(file,blanks,leftmargin);
        write(file,&buffer[row * columns],columns);
        write(file,CRLF,2);
    }
    write(file,FF,1);
    write(file,RESET,2);
    close(file);
    busy = FALSE;
}

void prtsc2(ALLREG * registers, ISRCTRL * isrblk, ISRMSG * s)
{
    enable( );
    if (busy)       peep( );
    else            prtsc( );
}

void main(int  paramcount, char * paramstr[ ])
{
    rstorisr(PRTSCINTERRUPT,ID);
    if (paramcount < 2) filename = "PRN";
    else
        if (paramstr[1][0] = = '/') {
            printf("custom  prtsc is no longer installed.");
            return;
        }
        else
            filename = paramstr[1];
    isinstal(PRTSCINTERRUPT,prtsc2,ID,&controlblock,stacks,
```

```
            STACKSIZE,MAXDEPTH);
    printf("prtsc outputs to file %s.",filename);
    isresext(0);
}
```

这个函数除打印屏幕这一基本功能外，还具有许多格式功能。例如，顶上空出一些行，左边留出一些空白，弹出一页纸等等。函数假设所用的打印机是 Epson Mx-100，几个控制码序列都是按此列出的。如果不是这种打印机，则可能要做相应修改。

函数首先置全局布尔变量 busy(忙标志)为真，以保证在打印时不再有别的中断请求打印。然后，通过 Turbo C Tools 函数 scrows 和 scmode 取得当前屏幕的行数和列数，通过函数 virdrect 把屏幕内容读入缓冲区 buffer，通过函数 access 查看屏幕硬拷贝文件是否存在。如果存在，则打开它进行添加；如果不存在，则建立它。之后就往文件中写各种打印控制码和屏幕的内容，最后关闭文件，置全局变量 busy 为假，退出。

中断服务程序(函数)prtsc2 很简单。如果全局变量 busy 为忙，则调用函数 peep 响 0.5s；如果不忙，则调用函数 prtsc 往盘文件写屏幕的内容。但是，无论是发声还是写文件，都需要较长时间，所以程序一开头就通过 enable 函数开启中断，以便不妨碍其它中断的处理。注意，这里之所以通过 peep 函数发声，而不通过 putchar('\a')发声，原因是 putchar 函数要调用 DOS 的系统服务。

当发出响声时，实际上机器正在建立屏幕的硬拷贝文件，已经是第 2 次进入中断服务程序 prtsc2 了。所以，在安装这个中断服务程序时，至少需要两层堆栈。程序中设置 MAXDEPTH 为 2。如果正在发声时，又按<Shift__Prtsc>，则就是第 3 次进入中断服务程序，这可能引起系统崩溃。

Prtsc2.dem 用前面已经介绍过的函数 rstorisr 撤消任何原先安装的 Prtsc2 中断服务程序，通过命令行接受用户键入的盘文件名，如果用户未键入文件名，则缺省的文件名是 PRN，即标准的 DOS 打印输出文件。如果用户键入的文件名仅仅是一条斜线，则意味着撤消用户自己的中断服务程序，恢复以前的中断服务程序。

应该用下面的办法检测 Prtsc2 运行的可靠性：

第 1，在 DOS 提示符下按<Shift__Prtsc>键。

第 2，通过一个单行程序执行中断 5。

第 3，通过一个单行程序执行 Turbo C 的库函数 getch，并在等待按键的过程中先按<Shift__Prtsc>，再按任一其它键。

这个中断服务程序产生的输出比 DOS 的 type 命令产生的输出要漂亮一些，因为 type 命令产生的输出每行可能超过 80 个字符。

Prtsc2 并非在每种情况下都能正确运行。如果在执行一条需要花大量时间进行盘文件输入／输出操作的命令（例如执行 chkdsk 命令或执行 dir 命令但重定向到一个盘文件上）时，按<Shift__Prtsc>键去激活中断服务程序 prtsc2，则系统会崩溃！这是因为，DOS 是不可重入的。

13.4 用 Turbo C Tools 写插入服务程序

13.4.1 DOS 的重入问题

实际上，在大部分执行时间内，DOS 把一些重要信息存放在全局变量中。当 DOS 被 Prtsc2 中断时，Prtsc2 反过来又调用 DOS 服务进行文件输入／输出，这些全局变量又被改写了。第 2 次 DOS 服务执行完毕后会返回到 Prtsc2，Prtsc2 执行后又返回到 DOS，但这时全局变量已被破坏，使 DOS 不能往下正确运行。这就是所谓 DOS 在“致命阶段”被中断。

DOS 经常处于“致命阶段”，例如磁盘文件输入／输出肯定处于“致命阶段”。当 DOS 进入这样一种不稳定状态时，它会设置一个“致命阶段”标志。如果中断服务程序能找到这个标志，并在调用 DOS 服务之前检查这个标志，就可以避免系统崩溃。能够为此提供服务的就是 DOS 的系统调用 0x34，它在寄存器对 ES:BX 中返回一个指针，这个指针指向的就是这个“致命阶段”标志。应该注意，如果在调用 DOS 服务查询“致命阶段”标志时，DOS 正好处于“致命阶段”，则这种查询本身就可能引起系统崩溃。幸好系统调用 0x34 返回的是标志指针而不是标志值，这就可以在某个安全时间查到这个标志指针，以后通过这个指针而得到标志值，用不着总去调用 DOS 服务。Turbo C Tools 为此提供了函数

```
void utcrit(void);
int  utdosrdy(void);
```

函数 utcrit 把“致命阶段”标志地址存放在 Turbo C Tools 的全局变量 b_critadd 中，函数 utdosrdy 检查那个标志，根据 DOS 是否处于“致命阶段”返回 0 或 1。在调用 utdosrdy 前，必须先执行过一次 utcrit。这些函数和全局变量的说明都在头文件 butil.h 中。

假设建立如下这样一个中断服务程序：

```
void isr1(ALLREG  * pregs,ISRCTRL  * pisrblk,ISRMSG  * pmsg)
{
    cputs("Are DOS services abaiable? ");
    cputs((utdosrdy( )?"YES!":"NO!));
}
```

通过中断安装函数 isinstal 把它安装为中断 5 的服务程序，并在安装之前调用一次 utcrit。安装之后，用下列办法激活中断 5，看看在每种情况下 DOS 是否处于“致命阶段”，是否允许服务。

第 1，执行如下程序，通过一条简单语句激活中断 5：

```
#include <dos.h>
void main( )
{
    struct REGPACK r;
    r.r_ds=_DS;
    r.r_es=_ES;
```

```
        intr(0x5,&r);
    }
```

这时将报告 DOS 是可用的。

第 2，执行一个简单程序，这个程序在等待输入时按<Shift__Prtsc>，激活中断 5。如果等待键盘输入用的是 BIOS 调用 0x16 或 Turbo C Tools 的函数 kbgetkey，则将报告 DOS 是可用的。如果等待键盘输入用的是一个 DOS 系统调用或任一 Turbo C 库函数，如 getch，将报告 DOS 是不可用的。

第 3，在 DOS 提示符下按<Shift__Prtsc>，激活中断 5，将报告 DOS 是不可用的。

第 4，执行 DOS 的 chkdsk 命令或执行 dir 命令但重定向到盘文件上，在执行过程中按<Shift__Prtsc>，激活中断 5，将报告 DOS 是不可用的。

从以上所述可以看出，DOS 是经常处于"致命阶段"的。尤其是第 2 和第 3 两种情况，明明是等待键盘输入，却报告 DOS 是不可用的。实践证明，在这两种情况下是可以重入 DOS 的，至少可以运行 13.3.3 节那个 prtsc2.dem 程序。这说明 DOS 的"致命阶段"留的保险系数过大了点。

注意，DOS 的技术手册中没有正式公布"致命阶段"标志这一技术，但许多商品软件、DOS 本身都用到了这一标志。

13.4.2 插入服务技术

所谓插入服务就是插在别的程序的执行间隙内，且这个间隙允许用来执行某些服务程序。既然是可以插入时才插入，插入服务程序就有可能被推迟执行，这就牵涉到调度问题。为了与前几节的(中断)调度程序相区别，这儿把负责调度的程序称为"插入规划程序"或"规划程序"。

"插入"，是从插入服务程序的观点而言的，从被插入程序的观点，与其说是"插入"，倒不如说是"查询"，即当前执行程序在它认为可以安全地让其它程序插入时，才会查询是否有别的服务程序在等待插入运行。如果有，则让它运行，待它运行完后自己接着运行。

用户程序本身难以实现这种插入技术，因为它不知道何时是安全的，谁在等待执行，它们的入口地址在哪儿，等等。为了实现这种插入技术，Turbo C Tools 提供了很多辅助手段和函数。

Turbo C Tools 截取了中断 8、9、0x13、0x21、0x28 这 5 个中断的中断服务程序，加入它自己的过滤程序。任何程序运行时，只要发出 0x13、0x21、0x28 这 3 个软中断指令或发生硬件时钟和键盘中断，就会进入 Turbo C Tools 的过滤程序，这些过滤程序就会决定当时是否允许嵌入其它程序。如果允许，就让处于挂起的其它插入服务程序运行。下面分别介绍 Turbo C Tools 对这 5 个中断的处理（参见下面的图 13-1）。

(1) 中断 9

按下一个键或释放一个键时就会产生中断 9。Turbo C Tools 的中断 9 过滤程序将首先执行原来的中断 9 服务程序(也许这就是 BIOS 的中断 9 服务程序)，然后置位"忙"标志，通过 BIOS 中断 0x16 服务程序的服务 1 读取键盘输入，判断是否一个热键。如果

是，则把它记录在案，以备插入规划程序尽早调入相应插入服务程序为这个热键服务。注意，这里是“尽早”，而不是等着立即执行插入服务程序。最后释放“忙”标志，返回被键盘

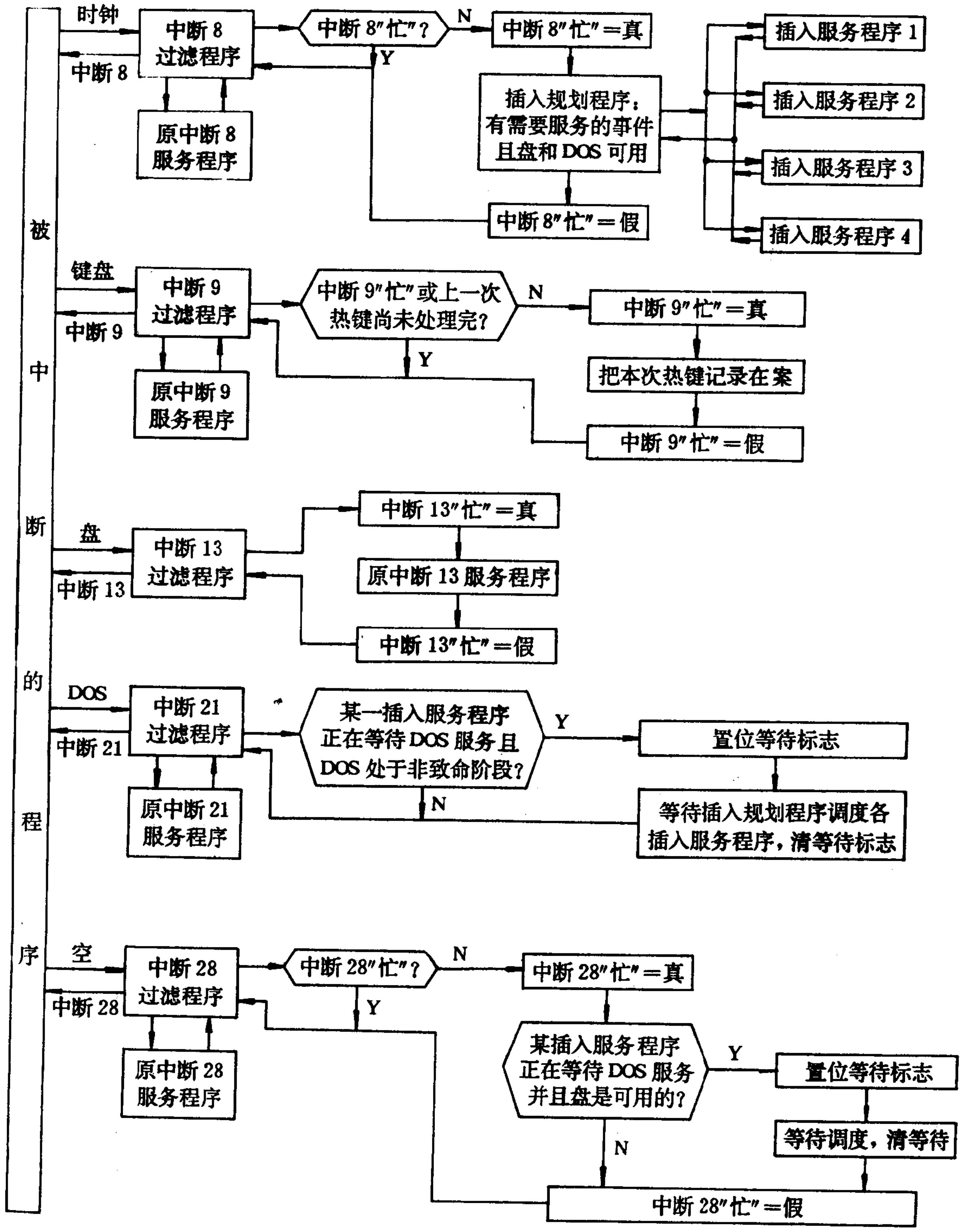

图 13-1　插入服务处理机制

9 中断的程序。但是，如果“忙”标志已经置位，或者插入规划程序正在处理上一次热键引起的插入服务，则不调用 BIOS 中断 0x16 服务程序，而是直接返回到被键盘 9 中断的程序。

由此可以看出，Turbo C Tools 的键盘过滤程序是可重入的，但热键插入请求却至多只有一个被记录在案，即被挂起。这种过滤处理办法存在两个缺点。第 1，既然是通过 BIOS 中断 0x16 取键盘输入，则只能从缓冲区的头而不是从缓冲区的尾取得输入。除非缓冲区空，否则按下的热键不会立即得到处理。第 2，BIOS 中断 0x16 只能取 BIOS 中断 9 认可并放入缓冲区的键和键组合，因此，像<F-J>、<Shift-Prtsc>这样的键组合都不可能产生一次热键插入请求。

(2) 中断 0x13

Turbo C Tools 的中断 0x13 过滤程序用于防止执行 0x13 中断服务程序过程中再被别的插入请求打断。因此，它首先设置“忙”标志，然后再调用原来的 0x13 中断服务程序进行处理，最后撤消忙标志，返回被中断程序。它根本不让插入规划程序和插入服务程序获得执行的机会。实际上，如果所有应用程序都是通过 DOS 的文件系统存取盘上的信息，则这个过滤程序就没有必要，因为下面将要讲到的中断 0x21 过滤程序已经防止了这类重入问题。之所以还要这个过滤程序，只是因为有些应用软件是通过绝对盘地址读写来存取盘上的信息的。

(3) 中断 0x21

这个中断是 DOS 系统调用的总入口。一般来说，程序花在执行 DOS 系统调用方面的时间将占程序执行时间的大部分，因此必须考虑在执行 DOS 系统调用时也能让插入服务程序运行，否则插入服务程序获得运行的机会就会大大减少。Turbo C Tools 的中断 0x21 过滤程序也是首先调用原来的 0x21 中断服务程序去服务这次系统调用。然后，检查是否有一插入服务程序正在等待获得 DOS 系统调用服务，如果有并且 DOS 也处于“非致命阶段”，则置位一个“等待”标志，并不断查询这个标志，一直等到插入服务程序不再需要 DOS 系统调用服务，即等到“插入规划程序”清除了这个“等待标志”，才返回发出 0x21 中断调用的那个程序。如果 DOS 处于“致命阶段”或插入服务程序不需要 DOS 服务，则不进入这个等待循环，直接返回。

(4) 中断 0x28

当 DOS 处于等待键盘输入并可提供盘服务时，它会调用中断 0x28 服务程序。DOS 本身为 0x28 提供的服务程序是空的，什么也设做。Turbo C Tools 的中断 0x28 过滤程序首先执行原来的 0x28 中断服务程序，然后置位一个“忙”标志，查看是否有一个不需要键盘输入／输出(即不需要 DOS 系统调用的子功能号 0～0xc)的插入服务请求。如果没有，则在释放“忙”标志后返回。如果有，则设置“等待”标志，进入一个等待循环，直至这个插入服务请求得到满足，即直到插入规划程序清除这个“等待”标志，才释放“忙”标志，返回发出 0x28 中断调用的那个程序。如果“忙”标志本来就已经置位，则不进入等待循环。这就是说，Turbo C Tools 的 0x28 中断服务程序是可以重入的，但它的等待循环是不可重入的。

(5) 中断 8

这是一个周期性的“tick”硬件时钟中断，频率约为 18.2Hz。Turbo C Tools 的中断 8

过滤程序首先执行原来的中断 8 服务程序。然后，检查它的"忙"标志。如果忙，则立即返回被时钟中断的程序。如果不忙，则置位"忙"标志，启动插入规划程序。插入规划程序检查当时的时钟，看是否有定时事件(也称为闹钟事件) 或周期事件已经到时。如果到时，则把这些事件记录在案。接着，对这记录的事件以及由 0x9 中断过滤程序记录的热键事件统一进行调度。再根据中断 0x13，0x21，0x28 过滤程序所设立的标志，决定 DOS 资源和盘资源是否可用。依次检查各事件，如果某一事件的插入服务程序所需要的 DOS 资源和盘资源可以得到满足，则让这个插入服务程序去执行。如果某一插入服务程序需要 DOS 系统调用，则插入规划程序还会禁止 <Ctrl__Break> 和 <Ctrl__C>，调用 iscurprc 函数使这个插入服务程序成为当前执行进程(见 13.2.5 节)，以便插入服务程序可以打开文件和分配内存。处理完所有插入服务后，最后执行一些必要的握手处理，如清除 0x21 和 0x28 设置的"等待"标志，清除它自己的"忙"标志，返回到过滤程序，最后返回到被时钟中断的程序。从这一过程可以看出，这个过滤程序也是可重入的，但插入规划程序是不可重入的。

总之，插入服务程序很类似于中断服务程序，但又不同于中断服务程序。它不是由硬件中断事件或软中断指令事件引起的，而是由定时事件和周期事件和热键事件引起的。虽然这三个事件与时钟中断和键盘中断紧密相关，但不等于这两个中断。只是在"时间到了"或"热键按下了"才会记录在案，才有可能进入插入服务程序，其余的时钟中断和键盘中断将只进入相应的中断服务程序。由于不能通过软中断指令来激发插入服务程序，所以插入服务程序的适用范围将比中断服务程序小得多，当然也可靠得多。

只有每次时钟中断到来时，插入服务程序才有可能得到执行。这就是说，如果在这次时钟中断到来时插入服务程序没有机会执行，则至少还需要再等 55ms 才有机会执行。究竟是否有机会执行，还要看插入服务程序是否需要磁盘资源和键盘资源以及 DOS 资源，这些资源当时是否可用。

为了便于说明启动插入服务程序的各种事件，Turbo C Tools 在头文件 binterv.h 中定义了如下两个结构：

```
typedef struct
{
    char ch;
    char keycode;
    int  action;
} IV__KEY;
typedef struct
{
    long ticks;
    int  action;
} IV__TIME;
```

结构 IV__KEY 定义热键，ch 和 keycode 分别是热键的 ASCII 码和扫描码，action 是这个热键应引起的动作，可以是下列 4 种值之一：

IV__KY__NONE(0):　　　　没有热键按下

IV__KY__SERVICE(1):　　激活插入服务函数
IV__KY__SLEEP(2):　　临时禁止插入服务程序
IV__KY__WAKE(3):　　使插入服务程序重新起作用

结构 IV__TIME 定义定时事件和周期事件。字段 ticks 说明时间的长短，单位是"tick"，字段 action 是说明何种事件，可以是下列两种值之一：

IV__TM__INTERVAL(1):　　周期事件
IV__TM__MOMENT(2):　　定时事件

因为一个插入服务程序可以响应许多热键，也可以响应许多定时或周期事件，所以在安装一个插入服务函数时，应提供两个结构数组，列出所有激活插入服务程序的事件：

```
IV__KEY     keytable[numkeys]
IV__TIME    timetable[numtimes]
```

但是，具体到某次激活插入服务程序，则总是由于某一具体事件引起的，插入规划程序必须把这一具体事件告诉插入服务程序。为了便于描述这一具体事件，Turbo C Tools 中还定义了结构 IV__EVENT：

```
typedef struct
{
    IV__KEY  key:              热键描述，如果是热键激活的。
    int time__action;          定时事件还是周期事件，如果是这些事件激活的。
    int time__index;           timetable中的索引号。
    long time;                 "tick"计时器的当前值。
}IV__EVENT
```

13.4.3 插入服务函数

Turbo C Tools 6.0 中提供了如下几个插入服务函数：

```
IV__CTRL far *cdecl  ivctrl(void);
int cdecl  ivdetect(const IV__VECTORS far *pvectors,
                    const char far *pident,IV__CTRL far **ppctrl,
                    unsigned far *pfound);
int cdecl  ivdisabl(IV__CTRL far *pctrl);
int cdecl  ivinstal(void (*pfunc)(IV__EVENT *),
                    const char *pident,char *pstack,
                    int stksize,IV__KEY *pkeytable,int numkeys,
                    IV__TIME *ptimetable,int numtimes,int option);
IV__CTRL far *cdecl  ivsense(const IV__VECTORS far *pvectors,
                      const char far *pident);
int cdecl  ivvecs(int option,IV__VECTORS far *pvectors);
```

(1) 函数 ivinstal 是插入安装函数。pfunc 是指向插入服务函数(程序)的指针，该函数应接受指向结构 IV__EVENT 的指针作为参数。参数 pident 是指向 16 个字符(包括最后的空字符)组成的标识名字符串的指针。参数 pstack 和 stksize 是指向堆栈的指针以及这个

堆栈的大小。这个堆栈空间必须是静态的，直至该插入服务函数失效。与中断服务函数不一样，因为插入服务程序没有重入的问题，所以只需要一层堆栈。参数 pkeytable、numkeys、ptimetable、numtimes 就是前面所说的那两个事件数组，说明哪些事件可以激活这个插入服务函数。参数 option 由 4 位组成，可以是下列值的逻辑“或”：

符 号	值	意 义
IV__DOS__NEED	1	需要DOS系统调用
IV__NO__DOS__NEED	0	不需要DOS系统调用
IV__DKEY__NEED	2	需要DOS系统调用1～12
IV__NO__DKEY__NEED	0	不需要DOS系统调用1～12
IV__FLOAT__NEED	4	激活时重新安装浮点向量
IV__NO__FLOAT__NEED	0	不需要重新安装浮点向量
IV__DAILY	8	定时事件应该每日发生

如果选择了 IV__FLOAT__NEED，则在调用插入服务程序之前，插入规划程序总是装入一组浮点模拟程序的中断向量，因为自插入服务程序被安装之后，别的软件可能已经重新设置了这些中断向量。

该函数的返回值是错误码，可能为下列值之一：

IV__NO__ERROR	0
IV__INSTALLED	5

(2) 函数 ivvecs。Turbo C Tools 的插入服务技术自动使用了 5 个插入过滤程序:时钟，键盘，磁盘，DOS 系统调用，DOS 空调用。实际上，Turbo C Tools 还有两个可选的插入过滤程序：通信口 1 和通信口 2。全局变量 b__ivmask 记录了被函数 ivinstal、ivsense 和 ivdisable 所用的插入过滤程序。下面是 b__ivmask 所使用的位值：

IV__F__TIMER	0X0001	INT 08H: ″tick″中断
IV__F__KEYSTROKE	0X0002	INT 09H: 键盘中断
IV__F__DISK	0X0004	INT 13H: BIOS磁盘服务
IV__F__IDLE	0X0010	INT 38H: DOS空闲
IV__F__2COM	0X0020	INT 0CH: 串行口1
IV__F__1COM	0X0040	INT 0BH: 串行口2
IV__STD__FILTERS	(IV__F__TIMER|IV__F__KEYSTROKE|IV__F__DISK| IV__F__DOS|IV__F__IDLE)	
IV__COM__FILTERS	(IV__F__2COM|IV__F__1COM)	

注意，b__ivmask 是一个工作变量。当调用 ivinstal 函数时，b__ivmask 指明要安装哪些插入过滤程序，一旦安装完毕，这个值就记录在插入控制块中的字段 filter__mask 中，如果必要，以后可以对变量 b__ivmask 进行修改而不影响插入服务程序的运行。当调用 ivsense 和 ivdetect 函数时，b__ivmask 指明要检查哪些中断向量的值，仅当当前所有这些中断向量的值都是插入过滤程序的入口地址而不是普通中断服务程序的入口地址时，这两个函数才认为可以摘除插入服务程序，即可以使 b__ivmask 中的所有插入过滤程序同时失效。当调用 ivdisabl 函数摘除一个插入服务程序时,b__ivmask 指明要摘除哪些

插入过滤程序。

函数 ivvecs 设置或读取插入过滤程序所使用的中断向量。如果参数 option 是 IV_SETVEC(1)，则为设置，设置哪一个或哪些中断向量，由 b_ivmask 决定，中断向量的值由结构 IV_VECTORS 给定。如果参数 option 是 IV_RETVEC(0)，则读取 Turbo C Tools 插入过滤程序当时所使用的所有中断向量，不管 b_ivmask 的值如何。读取的中断向量通过结构 IV_VECTORS 返回。

IV_VECTORS 记录了 7 个中断向量，它的定义在头文件 binterv.h 中，如下所示：

```
typedef struct
{
    const void far *ptimer;
    const void far *pkeybd;
    const void far *pdisk;
    const void far *pdos;
    const void far *pidle;
    const void far *pcom1;
    const void far *pcom2;
} IV_VECTORS;
```

注意，ivvecs 读取的是 7 个中断向量的值，但这些值不一定是 Turbo C Tools 插入过滤程序的入口地址。如果先安装了插入服务程序，后安装中断服务程序，则读取的将往往是中断服务程序的地址，而不是插入服务程序的地址。

(3) 函数 ivsense。这个函数的入口参数 pvectors 所指结构 IV_VECTORS 中各中断向量的值应是各插入过滤程序的入口地址，函数根据变量 b_ivmask 中的值，检查相应中断向量的值是否就是参数 pvectors 所指结构 IV_VECTORS 中的值，并且插入服务程序的标识名是否就是参数 pident 所指向的字符串。如果不是，则返回一个空指针 FARNIL。如果是，则说明可以摘除这个插入服务程序，返回一个指向插入控制块 IV_CTRL 的指针。插入控制块 IV_CTRL 的定义在头文件 binterv.h 中，其作用与 13.2 节的中断控制块 ISRCTRL 的作用类似，但它含有更多的字段，程序员不必理解所有这些字段的含义，本书只在必要时介绍其中的部分字段。

注意，这个函数只把当前中断向量的值和结构 IV_VECTORS 中的值进行比较，如果某插入过滤程序的入口地址被后安装的普通中断服务程序入口地址所屏蔽，则该函数找不到这个插入过滤程序。在下面这个例子里，假设在安装插入服务程序之后又安装了中断 8 服务程序，则在调用 ivsense 函数判断一个插入服务程序是否可以被摘除时，不能直接使用函数 ivvecs 返回的 IV_VECTORS 中的值，而要先把其中的中断 8 的向量值修改为插入过滤程序的入口地址，即中断控制块中保存的原来的中断向量值。

```
#include <dos.h>
#include <stdio.h>
#include <binterv.h>
#include <bintrupt.h>
ISRCTRL far *pisr;
IV_VECTORS vectors;
```

```
IV_CTRL far *pinterv;
void main( )
{
    pisr = issense(isgetvec(8),"MY CLOCKTICK ISR");
    if(pisr == FARNIL)
        printf("Can't find my clock tick ISR.\n");
    else {
        ivvecs(IV_RETVEC,&vectors);
        vectors.ptimer=pisr->prev_vec;
        pinterv=ivsense(&vectors,"MY INTERVENTION");
        if(pinterv==FARNIL)
            printf("Can't find my intervention function.\n");
        else if(pisr->isrpsp!= pinterv->psp)
            printf("ISR and intervention lie different programs\n");
        else {
            isputvec(8,pisr->prev_vec);
            if(IV_NO_ERROR != ivdisabl(pinterv))
                printf("Can't disable the intervention code.\n");
            else if (0!=isremove(pinterv->psp))
                printf("Can't remove the resident program.\n");
            else
                printf("Successfully disabled and removed.\n");
        }
    }
}
```

(4) 函数 ivdisabl。它摘除一个插入服务程序，也即使全局变量 b_ivmask 中的所有插入过滤程序失效，恢复原来中断向量的值。参数 pctrl 指向欲被摘除的插入服务程序的控制块。

函数的返回值可能是下列值之一：

```
IV_NO_ERROR          0
IV_NOT_ENABLED       1
IV_NOT_INSTALLED     2
IV_NULL_PTR          3
```

如果希望暂时使一个插入服务程序失效而不将它从内存中摘除，不恢复原来的中断向量，则不应调用 ivdisabl 函数，而只需置插入控制块中的 no_call 字段为 1，这就会在每个"tick"中断时，不调用插入规划程序，使插入服务程序得不到执行机会。如果以后又想让插入服务程序重新起作用，则只需设置插入控制块中的 no_call 字段为 0。

(5) 函数 ivdetect。这个函数与 ivsense 函数类似，但比 ivsense 函数功能更强。它的前两个参数的含义与函数 ivsense 的两个参数一样，给出中断向量的比较值和插入服务程序的标识名。第 3 个参数相当于函数 ivsense 的返回值，返回所发现的插入控制块的地址。因为这个参数是用于返回一个结构的内容，所以其类型是指针的指针。第 4 个参数返回一个无符号整数，其各位的定义与变量 b_ivmask 中各位的定义一样。如果某位为 1，则表明相应中断向量的当前值是一个插入过滤程序的入口地址；如果为 0，则是一个普通

中断服务程序的入口地址。这个函数的返回值如下所示：

```
IV__NO__ERROR          0
IV__NOT__FOUND         7
IV__PART__COVERED      6
```

例如：

```
#include <stdio.h>
#include <bintrupt.h>
#include <binterv.h>
IV__VECTORS vectors;
IV__CTRL far *pctrl;
unsigned found__mask;
void main( )
{
    ivvecs(IV__RETVEC,&vectors);
    b__ivmask=IV__STD__FILTERS|IV__COM__FILTERS;
    ercode=ivdetect(&vectors,"MY__INTERVENTION",&pctrl,&found__mask);
    if(ercode==IV__NOT__FOUND)
        printf("Intervention code not found.\n");
    else {
        if((pctrl->filter__mask!=(found__mask&pctrl->filter__mask)) {
            pctrl->no__call=1;
            printf("Diabling intervention code but unable to
                                        remove it.\n");
        }
        else {
            b__ivmask=pctrl->filter__mask;
            ercode = indisabl(pctrl);
            if(ercode==IV__NO__ERROR)
                printf("Intervention code successflly removed.\n");
            else printf("Error %d from IVDISABL.\n",ercode);
        }
    }
}
```

(6) 函数 ivctrl。这个函数返回指向插入控制块的指针。程序员很少需要对这个控制块进行直接操作。前面的例子中已经介绍了其中的两个字段：filter__mask 和 no__call。在下面的 rstorint 中，还将介绍另一个字段(psp)的用法。然而，一旦插入机制出错，检查一下结构中的下列字段是很有帮助的：awake，cbreak，timer__busy，kb__busy，disk__ busy，idle__ busy，com1__ busy，com2__ busy，idle__ safe，kb__ ext，key__event，req，wait 等（见 Turbo C Tools 的技术手册）。

13.4.4 插入服务程序举例

下面这个例子与 13.2 节介绍的例子 rstorisr 类似。那儿是撤消一个中断服务程序，这儿是撤消一个插入服务程序。应该注意的是，因为程序中实际撤消的是插入安装程序，因

此不仅是所指定标识名的插入服务程序，所有经它安装的插入服务程序都将被撤消。撤消后所有已分配的内存都将返回给 DOS。如果撤消成功，返回值为0，否则返回错误码。

```
#include  <general.h>
#include  <bintrupt.h>
#include  <binterv.h>
int  rstorint (char  * id)
{
     IV_VECTORS       vectors;
     IV_CTRL  far     * ivblock;
     int              code;
     ivvecs(IV_RETVEC,&vectors);
     ivblock  =  ivsense(&vectors,id);
     if (ivblock  = =  FARNULL)
              return  4;
     else
          if (code  =  ivdisabl(ivblock))
              return  code;
          else
              return  isremove(ivblock->psp);
}
```

下面这个例子的作用与 13.3.3 中的[例 3]一样，都是屏幕打印，程序与用户的介面形式也一致，最底层的把屏幕信息格式化后送往盘文件或打印机的函数是同一个 prtsc 函数。不同的只是那儿是采用中断服务程序的办法，而这儿是采用插入服务程序的办法。即使正在执行 DOS 系统调用时激活屏幕打印动作，这个例子也不会引起系统崩溃。

```
/ *    prtsc3.dem    * /
#include      <fcntl.h>
#include          <io.h>
#include   <general.h>
#include  <bintrupt.h>
#include   <binterv.h>
#define   beep( ) (putchar('\a'))
boolean busy    =  FALSE;
boolean wanted  =  FALSE;
void  prtsc(void);
void  prtscisr(ALLREG  * registers,ISRCTRL  * isrblk,ISRMSG  * s)
{
     enable( );
     if (busy)
          beep( );
     else
          wanted  =  TRUE;
}
void  prtsc3(IV_EVENT  * e)
{
     if (!wanted)
          return;
```

```
        else {
            prtsc( );
            wanted = FALSE;
        }
}
#include  <fcntl.h>
#include  <io.h>
#include  <general.h>
#include  <bvideo.h>
#define   exists(f) (access((f),0) == 0)
#define   ESCAT                    "\x1b\x40"
#define   SI                       "\xf"
#define   FF                       "\xc"
#define   CRLF                     "\r\n"
#define   RESET                    ESCAT
#define   COMPRESSED               SI
#define   ROWSPERPAGE              66
#define   COLUMNSPERPAGE           136
#define   MAXROWS                  43
#define   MAXCOLUMNS               80
extern    boolean    busy;
extern    char       * filename;
int  rows,columns;
int  topmargin,leftmargin;
int  ignore;
int  file;
int  row;
char buffer[MAXROWS * MAXCOLUMNS];
char blanks[MAXCOLUMNS / 2];
void prtsc(void) {
     busy = TRUE;
     rows = scrows( );
     ignore = scmode(&ignore,&columns,&ignore);
     virdrect(0,0,rows-1,columns-1,buffer,CHARS_ONLY);
     file=(!exists(filename)? _creat(filename,0) :
                         open(filename,O_WRONLY|O_APPEND));
     write(file,RESET,  2);
     write(file,COMPRESSED,1);
     topmargin  = (ROWSPERPAGE-rows) / 2;
     leftmargin = (COLUMNSPERPAGE-columns) / 2;
     memset(blanks,BLANK,leftmargin);
     for(row=0;row<topmargin;++row)
          write(file,"\r\n",2);
     for(row=0;row<rows;++row) {
          write(file,blanks,leftmargin);
          write(file,&buffer[row * columns],columns);
          write(file,CRLF,2);
     }
     write(file,FF,1);
     write(file,RESET,2);
     close(file);
```

```
        busy=FALSE;
}
int  rstorisr(int n,char  * id)
{
        ISRCTRL far  * isrblk;
        isrblk=issense(isgetvec(n),id);
        if (FARNULL= =isrblk) return 2;
        isputvec(n,isrblk->prev__vec);
        return isremove(isrblk->isrpsp);
}
int  rstorint (char  * id)
{
        IV__VECTORS       vectors;
        IV__CTRL far      * ivblock;
        int               code;
        ivvecs(IV__RETVEC,&vectors);
        ivblock  =  ivsense(&vectors,id);
        if (ivblock  = =  FARNULL)
                return 4;
        else
            if (code  =  ivdisabl(ivblock))
                return code;
            else
                return isremove(ivblock->psp);
}
#define  PRTSCINTERRUPT 5
#define  ID              "Screen hardcopy"
char     * filename;
ISRCTRL  controlblock;
#define  ISRSTACKSIZE    256
#define  MAXDEPTH        2
char     isrstacks[ISRSTACKSIZE * MAXDEPTH];
#define  IVSTACKSIZE     2048
char     ivstack[IVSTACKSIZE];
#define  TIMES           1
IV__TIME  timetable[TIMES]  =  {{1L,IV__TM__INTERVAL}};

void main(int paramcount,char  * paramstr[ ])
{
        rstorisr(PRTSCINTERRUPT,ID);
        rstorint(ID);
        if (paramcount  <2)
            filename  =  "PRN";
        else
            if (paramstr[1][0]  = =  '/') {
                printf("Custom  PrtSc  Is no longer installed.");
                return;
            }
            else
                filename  =  paramstr[1];
        isinstal(PRTSCINTERRUPT,prtscisr,ID,&controlblock,isrstacks,
```

```
                ISRSTACKSIZE,MAXDEPTH);
        ivinstal(prtsc3,ID,ivstack,IVSTACKSIZE,(IV_KEY  * )NULL,0,
                timetable,TIMES,IV_DOS_NEED);
        printf("PrtSc   outputs to file   %s.",filename);
        isresext(0);
    }
```

整个演示程序的名字是 prtsc3.dem，它所用到的中断服务函数的名字是 prtscisr，插入服务函数的名字是 prtsc3，负责打印的函数是 13.3.3 中的 prtsc。主函数 main 首先调用 13.2.4 节介绍过的 rstorisr 函数和 13.4.3 节介绍过的 rstorint 函数，撤消用户原先安装的中断服务程序和插入服务程序，解释命令行参数。如果命令行参数只是一个斜线符，则意味着用户要求撤消服务，因前面已经撤消了，则直接返回即可。否则，建立相应的盘文件或打印文件，安装中断服务程序和插入服务程序，并驻留在内存。

这儿既然是采用插入服务技术来解决问题，为什么还要安装中断服务程序呢？这是因为，Turbo C Tools 的键盘插入过滤程序是首先执行原来的中断 9 服务程序，然后才调用 BIOS 中断 0x16 从缓冲区取字符来判断是否击了一个热键。这样，当用户按 <Shift_Prtsc>键后，将首先进入 BIOS 的中断 9 处理程序，这个处理程序将发出一个中断 5 调用，但不识别<Shift_Prtsc>为 ASCII / IBM 码或扩展的 ASCII 码，更不往键盘缓冲区内存放这个码，所以插入过滤程序将取不到这个<Shift_Prtsc>，也无法激活插入服务程序。解决的办法是截取中断 5 的服务程序，如果当时没正在打印，则设置一个 wanted 标志。以后当每个"tick"中断到来时，由插入规划程序启动插入服务程序，插入服务程序再检查这个 wanted 标志。如果它已置位，则表示曾经在打印机不忙时按过<Shift_Prtsc>键，于是调用 prtsc 函数打印屏幕，打印过后复位 wanted 标志。

这就是说，打印屏幕这个动作表面上是由于按了<Shift_Prtsc>键引起的，但实际上却是由于"tick"中断而引起的，<Shift_Prtsc>并没有通常意义上的热键作用。如果仍说是一个热键，那也是由 BIOS 中断 9 检测的，而不是由用户程序检测的。正因为如此，在安装插入服务程序时，没有指定热键触发事件表，只指定了一个周期触发事件，时间间隔为 1，即每个“tick”都可能触发打印屏幕的动作。

还应注意的是，在打印机忙时仍然是调用 peep 函数发声，而不是调用 putchar('\a') 发声。按理，应该可以使用 putchar('\a')，但使用 putchar('\a')可能引起系统崩溃。总的说来，这个演示程序可以相当安全地运行。

13.5 用 Turbo C 写中断服务程序

与 Turbo C Tools 相比，Turbo C 对中断服务程序的支持显得太简单。但是，如果在中断服务程序中不牵涉到中断链以及 DOS 和其本身的重入问题，则用 Turbo C 的函数也就可以了。

Turbo C 提供了一种新的函数类型 interrupt。如果一个 C 函数准备作为中断服务程序，则必须说明这个函数为 interrupt 类型，如下所示:

```
void interrupt myhandler(bp,di,si,ds,es,dx,cx,bx,ax,ip,cs,flags);
```

所有参数都是同名寄存器的内容，但在函数内部可以像使用任何其它C的无符号整数一样使用CPU的所有寄存器，而不用借助于伪变量。interrupt型函数会自动保留入口时的bp、si、di、ax、bx、cx、dx、es和ds寄存器的内容，在返回时再恢复这些寄存器。，如果在函数内通过C语句显式地修改了这些寄存器的内容，则在返回时这些寄存器的内容也就会被修改，即可以通过寄存器传递返回值。还应注意，interrupt型函数是通过iret指令返回的，所以不应像调用一般的C函数那样来调用interrupt型函数，而应通过软中断指令或Turbo C的geninterrupt函数来调用。

下面是这样一个interrupt型函数，它发出一会儿声音。声音的频率由函数内的循环参数j决定，声音的长短由入口时寄存器bx的值决定。

```
#include <dos.h>
void interrupt mybeep(unsigned bp,unsigned di,unsigned si,
                      unsigned ds,unsigned es,unsigned dx,
                      unsigned cx,unsigned bx,unsigned ax)
{
    int  i,j;
    char originalbits,bits;
    unsigned bcount=bx;
    /*  get the current control port setting */
    bits=originalbits=inportb(0x61);
    for(i=0;i<=bcount;i++) {
        /* Turn off the speaker for awhile */
        outportb(0x61,bits&0xfc);
        for(j=0;j<=100;j++)
            ; /* empty statement */
        /* Now turn it on for some more time */
        outportb(0x61,bits|2);
        for(j=0;j<=100;j++)
            ; /* another empty statement */
    }
    /* Restore the control port setting */
    outportb(0x61,originalbits);
}
```

有了interrupt型函数后，还需要一个安装这个中断服务函数的函数。这个安装函数应接收中断服务函数的地址和指定的中断向量号，如下所示：

```
void install (void interrupt (*faddr)( ), int inum)
{
    setvect (inum, faddr);
}
```

最后，还需要一个激活中断服务程序的函数，每当需要激活中断时，调用这个函数就可以了，如下所示：

```
void testbeep(unsigned bcount, int inum)
{
```

```
    _BX = bcount;
    geninterrupt(inum);
}
```

这个函数中用到了伪变量_BX，有关伪变量的说明请见 12.2.1。做了这些准备工作后，就可以写出如下这样的主函数:

```
main( )
{
    char ch;
    install(mybeep,10);
    testbeep(3,10);
    ch = getch( );
}
```

第14章　图 形 处 理

14.1　Turbo C 图形处理函数

14.1.1　概述

如果按功能对 Turbo C 的库函数进行分类，属于图形处理的库函数是最多的，大约上百个。本章只对其中较难理解的部分函数做了较详细的解释。

这些图形处理函数大约可以分为 7 类：

图形系统控制函数；

画图和图形填充函数；

屏幕和视口管理函数；

图形方式下的文本输出函数；

颜色控制函数；

错误处理函数；

状态查询函数。

14.1.2　图形系统控制函数

图形系统控制函数有如下这些：

```
void          far _Cdecl closegraph(void);
void          far _Cdecl detectgraph(int far *graphdriver,
                                     int far *graphmode);
int           far _Cdecl getgraphmode(void);
void          far _Cdecl getmoderange(int graphdriver,
                                  int far *lomode,int far *himode);
void          far _Cdecl graphdefaults(void);
void          far _Cdecl _graphfreemem(void far *ptr,unsigned size);
void far *    far _Cdecl _graphgetmem(unsigned size);
void          far _Cdecl initgraph(int far *graphdriver,
                                   int far *graphmode,
                                   char far *pathtodriver);
int           far _Cdecl installuserdriver( char far *name,
                                       int huge(*detect)(void));
int           far _Cdecl installuserfont( char far *name );
void          far _Cdecl restorecrtmode(void);
int               _Cdecl registerbgidriver(void (*driver)(void));
unsigned      far _Cdecl setgraphbufsize(unsigned bufsize);
void          far _Cdecl setgraphmode(int mode);
```

为了启动图形系统，必须调用的第1个函数就是initgraph，它对图形系统进行初始化。所谓初始化，就是从盘上装入一个图形驱动程序或校核已登记的驱动程序是否存在，并使 显示系统进入图形方式。如果在编译／连接时已把驱动程序连接进来，并且在调用initgraph函数前已用registerbgidriver对这个驱动程序进行登记，则调用initgraph的目的就只是为了进行校核，否则就是为了在运行时装入驱动程序。

为了装入驱动程序，可以告诉initgraph装入哪一个驱动程序和使用哪一种显示方式，也可以让initgraph(它再去调用detectgraph)根据当时实际使用的显示卡自动选择合适的驱动程序，并采用该卡所支持的最高分辨率的显示方式。函数initgraph也调用graphdefaults函数，复位所有图形系统参数到它们的缺省值，包括设置视口为整个屏幕，当前位置为(0,0)，设置缺省的调色板、背景颜色、画图颜色、填充型式、字库和对齐形式等等。initgraph还初始化用于记录最近一次图形操作错误码的变量graphresult为0，即没有错误。

一般情况下，initgraph调用函数__graphgetmem分配一块内存，然后装入相应的.bgi驱动程序文件。如果不希望动态装入，而预先把用户程序和图形驱动程序连接起来，则应先用实用程序bgiobj.exe把相应的.bgi文件变成目标文件.obj，然后在连接时把这个目标文件和用户的目标文件连接起来。在运行用户程序时，再通过registerbgidriver登记上这个驱动程序，initgraph就不需要重新装入了。当然，调用initgraph时所指定的驱动程序应与已登记的驱动程序一致。

调用initgraph时，参数pathtodriver说明驱动程序的目录路径名。如果在这个目录下找不到或者目录路径名为空，则到当前目录下去找。参数graphdrier是一个整数指针，它说明用哪一种图形驱动程序，其符号与值的对应关系如下表所示。从符号名就可以知道相应的驱动程序适用于哪种显示卡。

驱动程序符号	**值**
DETECT	0
CGA	1
MCGA	2
EGA	3
EGA64	4
EGAMONO	5
IBM8514	6
HERCMONO	7
ATT400	8
VGA	9
PC3270	10
CURRENT__DRIVER	−1

调用registerbgidriver登记一个图形驱动程序时，参数driver指定驱动程序的符号名。随Turbo C的图形包一共提供了6个驱动程序文件，其对应的符号名如下所示：

图形驱动程序文件(.bgi)	**registerbgidriver符号名参数**
CGA	CGA__driver
EGAVGA	EGAVGA__driver

HERC	Herc__driver
ATT	ATT__driver
PC3270	PC3270__driver
IBM8514	IBM8514__driver

参数 graphmode 也是一个整数指针，它说明用哪一种图形显示方式。如果 * graphdriver 指定为 DETECT，则自动选最高分辨率的显示方式。注意，这里的显示方式号不是通常的 BIOS 所定义的显示方式号，而是 Turbo C 定义的显示方式号，如下表所示：

驱动程序号	显示方式号	方式值	分辨率	调色板	总页数
CGA	CGAC0	0	320 * 200	C0	1
	CGAC1	1	320 * 200	C1	1
	CGAC2	2	320 * 200	C2	1
	CGAC3	3	320 * 200	C3	1
	CGAHI	4	640 * 200	2 色	1
MCGA	MCGAC0	0	320 * 200	C0	1
	MCGAC1	1	320 * 200	C1	1
	MCGAC2	2	320 * 200	C2	1
	MCGAC3	3	320 * 200	C3	1
	MCGAMED	4	640 * 200	2 色	1
	MCGAHI	5	640 * 200	2 色	1
EGA	EGALO	0	640 * 200	16 色	4
	EGAHI	1	640 * 350	16 色	2
EGA64	EGA64LO	0	640 * 200	16 色	1
	EGA64HI	1	640 * 350	4 色	1
EGAMONO	EGAMONOHI	3	640 * 350	2 色	1(64K)
	EGAMONOHI	3	640 * 350	2 色	2(256K)
HERC	HERCMONOHI	0	720 * 348	2 色	2
ATT400	ATT400C0	0	320 * 200	C0	1
	ATT400C1	1	320 * 200	C1	1
	ATT400C2	2	320 * 200	C2	1
	ATT400C3	3	320 * 200	C3	1
	ATT400MED	4	640 * 200	2 色	1
	ATT400HI	5	640 * 400	2 色	1
VGA	VGALO	0	640 * 200	16 色	2
	VGAMED	1	640 * 350	16 色	2
	VGAHI	2	640 * 480	16 色	1
PC3270	PC3270HI	0	720 * 350	2 色	1
IBM8514	IBM8514HI	0	640 * 480	256 色	
	IBM8514LO	0	1024 * 768	256 色	

一旦图形驱动程序装入，通过函数 getdrivername 就可以取得这个驱动程序的符号名，通过函数 getgraphmode 取得当前所用显示方式号，通过函数 getmaxmode 取得这个驱动程序所能支持的最大显示方式号，最小显示方式号固定为 0。知道了显示方式号，还可以通过函数 getmodename 得到显示方式的名字，以便用于菜单或用于报告显示状态。通过函数 setgraphmode 可以设置当前显示方式，通过函数 restoremode 使屏幕回到文本方式，并回到执行 initgraph 函数前所在的文本方式显示号，但并不关闭图形系统，驱动程序和字库仍在内存中。这就是说，可以把 setgraphmode 和 restoremode 联合起来使用，构成一对开关，在图形方式和文本方式之间进行切换。注意，不能用下面将要讲到的 textmode 函数来做这种切换工作，textmode 只能在屏幕已处于文本方式下才可以做各种文本方式之间的切换工作。

函数 graphdefaults 使图形系统恢复到它的缺省设置，包括设置视口为整个屏幕、光标位置在(0,0)、缺省的调色板、背景色和画图色、缺省的填充图样、缺省的字库和对齐规则。

在 Turbo C 的图形包中，已提供了下列一些驱动程序文件和字库文件：

att.bgi	cga.bgi	egavga.bgi
hevc.bgi	ibm8514.bgi	pc3270.bgi
goth.chr	litt.chr	sans.chr
trip.chr		

用户可以装入新的驱动程序和新的字库。函数 installuserdriver 就是用来把用户的图形驱动程序装入到内部的 bgi 表格中去的。参数 name 是新的驱动程序文件的名字，参数 detect 是一个函数指针。detect 不需要任何参数，它自动检测显示卡的硬件，决定用户驱动程序所要求的显示卡是否存在。如果不存在，返回值为−11。如果存在，返回所用的显示方式号（整数）。整个函数的返回值也是一个整数，表示赋给所安装的用户驱动程序的号码。显示卡制造厂家不仅应提供相应的驱动程序，还应提供相应的自动检测程序。

注意，installuserdriver 仅把用户驱动程序和所支持的显示方式号登记到内部的表格中，而不是真正使用这个驱动程序。真正使用这个驱动程序要通过 initgraph 函数实现。

假设用户有一种新的显示卡 SGA，其相应的图形驱动程序文件名是 SGA.BGI。使用 SGA.BGI 的办法有两种。其一是先调用 installuserdriver，安装 SGA.BGI，并把它的返回值(驱动程序号)作为参数再去直接调用 initgraph。其二也是先调用 installuserdriver，但不理睬它的返回值，在调用 initgraph 函数时要它去自动检测。因为在调用 installuserdriver 时已给定了驱动程序名和检测函数指针，所以 initgraph 在自动检测时将首先使用用户的自动检测函数进行检测，在这里就是 SGA 自动检测程序。如果用户的检测程序发现他的显示卡并不存在，则返回值为−11，表示错误(grerror)。然后 initgraph 将调用它自己的自动检测函数 detectgraph 继续进行检测，也许在此之前还会再调用另一个用户的检测程序去进行检测。依照这些用户检测程序被安装的先后次序，如果某个用户的检测程序发现他的显示卡确实存在，则返回一个非负的显示方式号，然后 initgraph 找到并装入这个 SGA.BGI 驱动程序，使显示卡硬件处于用户自动检测程序所推荐使用的显示

方式下，最后退出 initgraph。

最多可以一次安装 10 个图形驱动程序。

下面是安装并使用用户驱动程序的一个示意性例子:

```
#include <stdio.h>
#include <stdlib.h>
#include <graphics.h>
int driver,mode;
int huge detectsga(void)
{
      int found,defaultmode;
      if(!found) return (grError);
      defaultmode= ....
      return (defaultmode);
}
main( )
{
      driver=installuserdriver("SGA",detectsga);
      if(grOK != graphresult( ) ) {
      printf("Error installing user driver SGA.\n");
      exit(1);
      }
      initgraph(&driver,&mode,"");
      if(grrOK != graphresult( )) exit(1);
      outtext("Userinstalled crivers supported");
      getchar( );
      closegraph( );
}
```

函数 installuserfont 安装用户的字库文件，入口参数是字库文件的全文件名，返回的整数是这个字库文件的标识号 ID，以后在调用 settextstyle 函数时就可以用这个 ID 作为参数选择相应的字库文件。如果安装不成功，返回值为-11。

用户不仅可以提供自己的图形驱动程序，还可以控制图形驱动程序在内存中的位置。在运行用户程序时，图形系统可能需要分配内存去装入图形驱动程序、字库和内部缓冲区。但图形系统是调用_graphgetmem 而不是 malloc 分配内存，调用_graphfreemem 而不是 free 释放内存。如果用户希望自己控制图形内存的分配和释放，则可以提供自己的_graphgetmem 和_graphfreemem 函数。此时，尽管会报出重名错(duplicate symbols)，但不会有什么问题。如果用户不希望控制图形内存的分配和释放，则不用提供自己的函数，内部已有这两个函数了，但它们只不过分别调用 malloc 和 free 函数而已。

14.1.3 画图和填充函数

Turbo C 提供了丰富的画图和填充函数，通过这些函数可以画彩色的线条、圆弧、圆、椭圆、矩形、饼形、两维条形、三维条形、多边形，以及由这些基本图形组成的图形，可以控制横轴和竖轴的比例。通过这些函数可以填充任何封闭的图形，或填充围绕这个封闭图形的外部。填充图样已经定义了 11 种，用户还可以定义自己的填充图样。通过

这些函数还可以控制画笔的当前位置，控制画线的厚度和风格(实线，虚线等等)。下面是这些函数的名字、参数和简要说明。

```
void far _Cdecl arc(int x, int y, int stangle, int endangle,int radius);
void far _Cdecl circle(int x, int y, int radius);
void far _Cdecl drawpoly(int numpoints, int far *polypoints);
void far _Cdecl ellipse(int x, int y, int stangle, int endangle,int xradius, int yradius);
void far _Cdecl getarccoords(struct arccoordstype far *arccoords);
void far _Cdecl getaspectratio(int far *xasp, int far *yasp);
void far _Cdecl getlinesettings(struct linesettingstype far *lineinfo);
void far _Cdecl line(int x1, int y1, int x2, int y2);
void far _Cdecl linerel(int dx, int dy);
void far _Cdecl lineto(int x, int y);
void far _Cdecl moverel(int dx, int dy);
void far _Cdecl moveto(int x, int y);
void far _Cdecl rectangle(int left,int top,int right,int bottom);
void far _Cdecl setaspectratio( int xasp, int yasp );
void far _Cdecl setlinestyle(int linestyle, unsigned upattern,int thickness);

void far _Cdecl bar(int left, int top, int right, int bottom);
void far _Cdecl bar3d(int left, int top, int right, int bottom,int depth, int topflag);
void far _Cdecl fillellipse(int x,int y,int xradius,int yradius);
void far _Cdecl fillpoly(int numpoints, int far *polypoints);
void far _Cdecl floodfill(int x, int y, int border);
void far _Cdecl getfillpattern(char far *pattern);
void far _Cdecl getfillsettings(struct fillsettingstype far *fillinfo);
void far _Cdecl pieslice(int x, int y, int stangle, int endangle,int radius);
void far _Cdecl sector( int X, int Y, int StAngle, int EndAngle,
                        int XRadius, int YRadius );
void far _Cdecl setaspectratio( int xasp, int yasp );
void far _Cdecl setfillpattern(char far *upattern, int color);
void far _Cdecl setfillstyle(int pattern, int color);
void far _Cdecl setlinestyle(int linestyle, unsigned upattern,int thickness);
```

函数 setlinestyle 和下面将要讲到的函数 getlinesettings 是一对，用来设置和读取线型信息。但设置时用的是三个参数，读取时用的是一个参数。读取时的这个参数是一个结构，其三个字段分别对应于设置时的三个参数。

```
struct linesettingstype {
        int linestyle;
        unsigned upattern;
        int thickness;
};
```

参数 linestyle 的取值范围如下所示：

符号名	值	说明
SOLID__LINE	0	实线
DOTTED__LINE	1	点线
CENTER__LINE	2	中心线
DASHED__LINE	3	虚线
USERBIT__LINE	4	用户自定义线

参数 thickness 的取值范围如下所示:

符号名	值	说明
NORM__WIDTH	1	1个像点宽
THICK__WIDTH	3	3个像点宽

参数 upattern 是 16 位的码型，仅当参数 linestyle 为 USERBIT__LINE 时，这个参数才有意义，其它情况下将被忽略。16 位值中，与 1 相应的位将是前景色，与 0 相应的位将是背景色。

函数 setlinestyle 设置的线型只影响画线、画矩形、画多边形，不影响画圆弧、画圆、画椭圆、画饼形图。

函数 setfillstyle 和下面将要讲到的函数 getfillsettings 构成了另一对，分别用来设置和读取填充线型信息。但设置时用的是两个参数，读取时用的是一个参数。读取时的这个参数是一个结构，其两个字段分别对应于设置时的两个参数。

```
struct fillsettingstype {
        int pattern;
        int color;
};
```

参数 pattern 的取值范围是 0～11，分别对应 12 种填充线型。参数 color 说明填充线的颜色。

函数 setfillpattern 和下面将要讲到的函数 getfillpattern 又构成了一对，分别用来设置和读取用户自定义的填充线型信息。参数 pattern 指向一个 8 字节的数据数组，说明一个 8 * 8 的填充图样。为 1 的点则画出，为 0 的点则不画出。设置时还可指定画点的颜色。

14.1.4 屏幕和视口管理函数

屏幕和视口管理函数有如下这些:

```
void      far __Cdecl cleardevice(void);
void      far __Cdecl setactivepage(int page);
void      far __Cdecl setvisualpage(int page);

void      far __Cdecl clearviewport(void);
void      far __Cdecl getviewsettings(struct viewporttype far * viewport);
void      far __Cdecl setviewport(int left, int top, int right,int bottom,int clip);
void      far __Cdecl getimage(int left, int top, int right,int bottom,void far * bitmap);
unsigned  far __Cdecl imagesize(int left, int top, int right,int bottom);
```

```
void      far __Cdecl putimage(int left,int top,void far * bitmap,int op);
unsigned far __Cdecl getpixel(int x, int y);
void      far __Cdecl putpixel(int x, int y, int color);
```

在IBM PC系列机上，显示缓冲区最多可存储8个屏幕大小的信息，每屏所占的显示缓冲区称为一页。正在屏幕上显示的页称为当前显示页，正在编辑的页称为当前编辑页或当前活动页。当前显示页和当前编辑页不一定是同一页。函数setvisualpage和setactivepage分别设置当前显示页和当前编辑页。函数cleardevice清除整个屏幕，但保留所有其它图形系统参数不变(画图线型，填充图样，字库种类，视口坐标等等)。

与在文本方式下类似，Turbo C虽然提供了对图形窗口的支持，但这种支持是非常有限的。Turbo C对图形窗口的支持是通过其视口函数实现的。视口是屏幕上的一个矩形区域，视口与屏幕间的关系就像文本方式下window函数开辟的窗口与屏幕间的关系。视口只有一个，不是每页一个。因为开辟视口的目的是为了进行读写，所以视口实际上是与当前编辑页相联系的。如果当前编辑页不是当前显示页，视口中的内容也就看不到了。所有图形输出函数，包括画图、填图、文本等等，它们的坐标都是相对于视口左上角的，而不是相对于屏幕左上角。注意，putimage和setimage函数的坐标也是相对于视口的，而文本方式下，函数puttext和gettext函数的坐标是相对于整个屏幕的。另外，getimgae和putimgae所处理的缓冲区中不仅应包含映像本身，还应包含映像区宽度和高度信息，在单独调用putimage函数时尤其应加以注意。正因为缓冲区中包含了映像区尺寸信息，所以putimage的参数只需要左上角坐标和缓冲区指针，不需要右下角坐标。但putimage多了一个参数op，用来说明在把缓冲区内容写到屏幕上去以前，先要和已在屏幕上对应位置的内容做什么操作。可选的值有如下5种：

符号常数	值	说 明
COPY_PUT	0	预先不进行任何操作
XOR_PUT	1	预先进行"异或"操作
OR_PUT	2	预先进行"或"操作
AND_PUT	3	预先进行"与"操作
NOT_PUT	4	预先对缓冲区内容求"反"。

Turbo C的window函数不支持多个窗口，也不提供窗口边界。类似地，Turbo C的setviewport函数不支持多个视口，也不提供视口边界。为了解决这些问题，必须由程序员在这些函数的基础上再去编程实现，详见14.2节的pop_up图形窗口工具包。

函数setviewport和函数getviewsettings构成一对，分别用来设置和读取有关视口的参数。但设置时用的是5个参数，读取时用的是一个参数。读取时的这个参数是一个结构，其5个字段分别对应于设置时的5个参数。参数(left,top)和(right,bottom)指定视口的左上角和右下角坐标。参数clip说明当前视口边界是(非0)否(0)被截断。

14.1.5 图形方式下的文本输出函数

图形方式下的文本输出函数如下：

```
void far __Cdecl gettextsettings(struct textsettingstype far * texttypeinfo);
```

```
void far __Cdecl outtext(char far  * textstring);
void far __Cdecl outtextxy(int x, int y, char far  * textstring);
int       __Cdecl registerbgifont(void  ( * font)(void));
void far __Cdecl settextjustify(int horiz, int vert);
void far __Cdecl settextstyle(int font, int direction, int charsize);
void far __Cdecl setusercharsize(int multx, int divx,int multy, int divy);
int   far __Cdecl textheight(char far  * textstring);
int   far __Cdecl textwidth(char far  * textstring);
```

Turbo C 的图形包提供了一个 8 * 8 的点阵字库和几个向量字库。点阵字库适用于字形较小的情况，向量字库适用于字形较大的情况。缺省情况下，使用的是 8 * 8 的点阵字库。各个向量字库包含在各个.chr 文件中，这些文件既可在运行时装入内存，也可在连接时连接到.exe 文件中(连接前还要用 bgiobj 实用程序转化为.obj 文件)。

函数 settextstyle 用来选择字库、字的书写方向和字形大小。所有其后的 outtext 和 outtextxy 函数产生的输出都将受 settextstyle 函数的影响。Turbo C 的字库有如下这些，用户还可以定义自己的字库，其字库号由 registerbgifont 返回。

字库号常数	值	说明
DEFAULT__FONT	0	8 * 8点阵库
TRIPLEX__FONT	1	triplex向量字库
SMALL__FONT	2	small向量字库
SANSSERIF__FONT	3	sans__serif向量字库
GOTHIC__FONT	4	gothic向量字库

8 * 8 点阵字库嵌在图形系统内，而其它字库却是由 settextstyle 函数从盘上相应的.chr 文件中装入的。如果在连接时已经连接上了，则不须装入，但在调用 settextstyle 函数前，必须先通过 registerbgifont 函数登记这个(或这些)字库文件。registerbgifont 的参数 * font 可以是如下这些:

参数符号名	相应的.chr文件
TRIP	triplex__fond
LITT	small__font
SANS	sansserif__font
GOTH	gothic__font

字的书写方向(direction)可以从左往右(HORIZ__DIR, 0)，也可以反时针转 90°，从下往上(VERT__DIR, 1)。

字的大小(charsize)可以是 1 倍、2 倍、0 倍。0 只用于向量字库，意味着乘以缺省值 4，除非用户通过 setusercharsize 函数重新设置字符大小。用户指定大小时，水平和垂直方向的比例因子可以不同，也不一定是整数倍，可用分数来表示，4 个参数依次是：水平方向的分子和分母，垂直方向的分子和分母。

```
struct textsettingstype {
        int font;
        int direction;
        int charsize;
        int horiz;
        int vert;
};
```

下面将要讲到的函数 gettextsettings 与函数 settextstyle 相反，它是读取有关图形下文本的控制信息。但它返回的是一个结构，这个结构的前 3 个字段就是函数 settextstyle 的 3 个参数，除此之外，它多了两个字段 horiz 和 vert。

函数 settextjustify 用来设置文本的对齐形式。参数 horiz 和 vert 分别说明水平和垂直方向的对齐形式，可以是下列值之一：

常数名	常数值	说明
LEFT_TEXT	0	水平左对齐
CENTER_TEXT	1	中心对齐
RIGHT_TEXT	2	水平右对齐
BOTTOM_TEXT	0	垂直底对齐
TOP_TEXT	2	垂直顶对齐

14.1.6 颜色控制函数

颜色控制函数如下：

```
int  far _Cdecl getbkcolor(void);
int  far _Cdecl getcolor(void);
struct palettetype * far _Cdecl getdefaultpalette( void );
int  far _Cdecl getmaxcolor(void);
void far _Cdecl getpalette(struct palettetype far *palette);
int  far _Cdecl getpalettesize( void );
void far _Cdecl setallpalette(struct palettetype far *palette);
void far _Cdecl setbkcolor(int color);
void far _Cdecl setcolor(int color);
void far _Cdecl setpalette(int colornum, int color);
```

处于图形方式下时，屏幕是由单个的像点组成的。像点值并不直接说明该点的颜色，它只不过是一个索引，根据这个索引到一个称为“调色板”的彩色表中找到的值才代表该点的颜色。这种间接办法有许多优点。虽然显示系统硬件可显示多种颜色，但往往同时在屏幕上显示的颜色种类却是有限的。这个有限数就称为调色板的大小，即 size。CGA 和 EGA 调色板的 size 分别为 4 和 16。像点值的取值范围应是 0~(size−1)。函数 getmaxcolor 返回当前显示方式下最大允许的像点值(size−1)。当讨论图形函数时，我们经常说的“颜色”(如当前画图色，填充色，像点色等)就是像点值，即调色板的索引，而不是实际颜色。通过对调色板进行操作，即使不改变像点值，也能改变实际显示的颜色。

背景色总是相应于像点值 0。

由于 CGA 和 EGA 在硬件上的差别，使得在编程时也稍有差别。这里所说的

CGA，包括 CGAHI、MCGAMED、MCGAHI、ATT400MED 和 ATT400HI。CGA 有两种分辨率：320 * 200 和 640 * 200，分别能显示 4 种颜色和 2 种颜色。

在 320 * 200 分辨下，有 4 种预先定义的调色板，每种调色板有 4 种颜色，如下表所示：

调色板号	像点值 0	像点植 1	像点值 2	像点值 3
0	背景色	浅绿	浅红	黄
1	背景色	浅青	浅品红	白
2	背景色	绿	红	棕
3	背景色	青	口红	浅灰

调色板号是通过 initgraph 函数的显示方式进行选择的，像点值是通过 setcolor 函数选择的。

背景色是通过 setbkcolor 函数选择的，其可取的范围为如下 16 种值之一：

符号	值	颜色
BLACK	0	黑
BLUE	1	蓝
GREEN	2	绿
CYAN	3	青
RED	4	红
MAGENTA	5	品红
BROWN	6	棕
LIGHTGRAY	7	浅灰
DARKGRAY	8	深灰
LIGHTBLUE	9	浅蓝
LIGHTGREEN	10	浅绿
LIGHTCYAN	11	浅青
LIGHTRED	12	浅红
LIGHTMAGENTA	13	浅品红
YELLOW	14	黄
WHITE	15	白

注意，在 CGA 中，与函数 setcolor 不同，setbkcolor 中的参数值不是像点值（调色板的索引），而是直接把这个参数值放到调色板的索引 0 处。

在 640 * 200 分辨率下，像点值只能是 0 或 1。由于 CGA 硬件本身奇怪的设计，前景色实际上被硬件当作了背景色，所以应通过 setbkcolor 来设置前景色。所设置的颜色可以是 16 种颜色中的任何一种，凡像点值为 1 的点都将以这种颜色显示出来。

因为 CGA 上的调色板是预先在硬件上定义好的，所以在 CGA 上不可用 setallpalette 函数，也不可用 setpalette(index，actual__color) 函数，除非 index=0。通过 index=0 来调用 setpalette 是另一种设置背景色的方法。

下面再来谈谈 EGA 和 VGA 的颜色控制方法。

在 EGA 上，总共可以有 64 种颜色，只有一个调色板。但这个调色板可以有 16 项，可同时显示 16 种颜色。缺省的 16 种颜色与 CGA 上的 16 种颜色相同，顺序也一样，只

是用来表示各颜色的符号名前都加了一个"EGA_"，如 EGA_BLACK，EGA_WHITE等。用户可以通过setpalette函数改变整个调色板，还可以通过getpalette函数取得整个调色板的值。

在EGA上，setbkcolor函数中的参数代表的是像点值，即调色板中的索引号，它所做的工作是把调色板中指定项的值拷贝到项0。这与CGA很不一样。

14.1.7 错误处理函数

错误处理函数只有两个:

```
char * far _Cdecl grapherrormsg(int errorcode);
int    far _Cdecl graphresult(void);
```

每当图形操作出现错误时，图形系统内部就会记录下这次错误。通过graphresult可以取得最近发生的错误的错误号码。错误号码是0～−18，0表示正确，负数表示出现某种错误。通过grapherrormsg可以获得相应于某一错误号的错误信息字符串。每调用一次graphresult函数或initgraph函数，就会清除原来的错误号，使错误号复位到0。

14.1.8 状态查询函数

Turbo C提供了如下这些状态查询函数:

```
void      far _Cdecl getarccoords(struct arccoordstype far *arccoords);
void      far _Cdecl getaspectratio(int far *xasp, int far *yasp);
int       far _Cdecl getbkcolor(void);
int       far _Cdecl getcolor(void);
char *    far _Cdecl getdrivername( void );
void      far _Cdecl getfillpattern(char far *pattern);
void      far _Cdecl getfillsettings(struct fillsettingstype far *fillinfo);
int       far _Cdecl getgraphmode(void);
void      far _Cdecl getlinesettings(struct linesettingstype far *lineinfo);
int       far _Cdecl getmaxcolor(void);
int       far _Cdecl getmaxmode(void);
int       far _Cdecl getmaxx(void);
int       far _Cdecl getmaxy(void);
char *    far _Cdecl getmodename( int mode_number );
void      far _Cdecl getmoderange(int graphdriver,int far *lomode,int far *himode);
void      far _Cdecl getpalette(struct palettetype far *palette);
int       far _Cdecl getpalettesize( void );
unsigned  far _Cdecl getpixel(int x, int y);
void      far _Cdecl gettextsettings(struct textsettingstype far *texttypeinfo);
void      far _Cdecl getviewsettings(struct viewporttype far *viewport);
int       far _Cdecl getx(void);
int       far _Cdecl gety(void);
```

14.2 pop__up 图形窗口工具包

14.2.1 图形窗口与文本窗口

4.4 节已经介绍了一个 pop__up 文本窗口工具包。这里所述的工具包借用了 4.4 节中的一些概念，甚至许多结构和函数名都一样，目的就是使两个工具包尽可能相兼容。正因为两个工具包很相似，对其中的工作原理就不介绍了。

但是，Turbo C 中为图形方式提供的支持比文本方式多，程序员能够对图形窗口进行更多的控制，所以许多窗口数据结构定义也有些不同。窗口的颜色结构 wincolors 被修改成如下所示：

```
typedef struct wincolors__struct {
        struct linesettingstype lineinfo;
        struct fillsettingstype fillinfo;
        struct textsettingstype textinfo;
        struct linesettingstype brdinfo;
        struct viewporttype     viewinfo;
        char   border__type;
        unsigned char border__color,text__color,hilite__color,back__color;
}wincolors;
```

与文本方式下的 wincolors 结构相比，这里多了 5 个结构字段和 1 个 back__color 字段，但却少了 1 个 title__color 字段。新增加的 5 个结构字段分别用来说明画图线型、填充图样、字库种类、边界线型、视口位置，所用到的 4 个结构定义就是 Turbo C 库函数中的结构定义（见 14.1 节）。新增加的 back__color 字段不是用来说明整个屏幕的背景颜色，而是指图形窗口被清除后所用的颜色。因为图形窗口没有标题，所以也就没有 title__color 这个字段。

边界类型字段基本上与文本方式下的工具包类似，取值可以是 0,1(NORM__WIDTH)，2(THICK__MIDTH)，分别表示没有边界、1 个像点宽边界和 2 个像点宽边界。

整个窗口结构 windesc 的定义也做了一些修改，如下所示：

```
typedef struct winstruct {
        char     * name;
        void     * image;
        struct   winstruct  * under, * over;
        wincolors wc;
        int      wd,ht;
        int      xsave,ysave;
        enum     windowtype wtype;
}windesc;
```

虽然图形窗口不显示标题，但为了保持与文本窗口相兼容，仍保留了存储窗口标题的

* name 字段。与文本窗口相比，图形窗口结构少了表示窗口坐标的 xul、yul、xlr 和 ylr 4 个字段，这几个坐标已经在 wincolors 结构中的 viewporttype 结构中定义了。为了使文本窗口下的程序能够比较容易地移植到图形窗口下，在图形窗口工具包的头文件中做了如下定义：

```
#define xul wc.viewinfo.left
#define yul wc.viewinfo.top
#define xlr wc.viewinfo.right
#define ylr wc.viewinfo.bottom
```

14.2.2 6 个工具函数

第 4 章的文本窗口工具包共提供了 14 个函数，这里的图形窗口工具包只提供了 6 个函数。其中 5 个函数(init__win，draw__win，view__win，swap__image，clr__win)与前一个工具包的名字是相同的，它们的功能一样，下面仅介绍它们在实现上的差别。图形窗口工具包多了一个 chg__win 函数，但却少了 9 个函数。少了的函数主要集中在文本输出方面，读者可自己补充这些函数。

注意，Turbo C 库函数支持图形视口而不支持图形窗口，这里的工具包函数虽然支持图形窗口，为了利用 Turbo C 函数，实际上每个图形窗口就对应一个 Turbo C 库函数的视口，边框是由工具包函数在视口内部实现的。虽然有时把窗口内部(除边框外)的部分也称为"视口"，但它不是 Turbo C 库函数中视口的含义。

```
extern  void       init__win(void);
extern  windesc    * draw__win(int x, int y, int wd, int ht, char * title,
                          enum windowtype wt,wincolors * wc);
extern  void       clr__win(void);
extern  void       view__win(windesc * this, int select);
extern  void       chgviewport(windesc * this);
extern  void       swap__image(windesc * w);
```

(1) 函数 init__win 用于初始化图形窗口工具包，取得当前屏幕的有关信息,存储在基窗口中。与文本窗口工具包不同，这儿的 init__win 函数用了 3 个新的全局变量，其定义如下：

```
int graphdriver=0;
int graphmode=0;
char * pathdriver=""
```

头两个变量是在初始化图形系统时用来指定图形驱动程序和图形方式(是 Turbo C 的图形方式，不是 BIOS 的显示方式)，缺省值是图形系统自动确定的图形驱动程序和相应的最高分辨率的图形方式。

第 3 个变量是图形驱动程序文件和字库文件的目录路径名，缺省值是空，意味着在当前目录下。这个图形窗口工具包是在运行时才装入图形驱动程序和字库的，如果希望在连接时就把它们连接在一起，则应相应地修改 init__win 函数。

(2) 函数 swap__image 用来把当前显示在屏幕上的窗口内容和保存在窗口结构内的窗口映像进行交换。它是通过 Turbo C 的 getimage 和 putimage 函数实现交换的，但 Turbo C 的这两个函数用的是相对于视口的坐标。为了交换整个窗口的内容，在交换之前要调用 chgviewport 函数转换到基窗口(即整个屏幕)的绝对坐标下，在交换之后再调用 chgviewport 函数转换到本窗口的相对坐标下。

(3) 函数 draw__win 用来弹出一个窗口。这个函数调用了 swap__image，而后者又调用了 putimage 去画出图像。与 gettext 和 puttext 所用的文本缓冲区不同，图像缓冲区不仅应包含图像本身，还应包含图像尺寸信息。如果在缓冲区未初始化之前就调用 putimage 函数，则画出来的图像是杂乱的。为了弥补这个缺陷，draw__win 函数调用 calloc 函数来分配图像缓冲区，因为函数 calloc 会初始化整个缓冲区为 0，这就初始化缓冲区的宽度和高度为 0，以后在调用 getimage 时这个尺寸信息会自动更新。

(4) 函数 clr__win 不仅清除当前的图形窗口,而且重画边界。注意，不可用 setbkcolor 设置窗口的背景颜色，因为这个函数是设置整个屏幕的颜色。也不可用 bar 重画窗口，因为 bar 函数只画出一个条形块，不加边框。只能用 bar3d 函数去做这件事，它不仅画出一个矩形边框，而且可以以指定的颜色填充这个矩形。在调用 bar3d 函数前，已设置填充图样为实心的，填充颜色是窗口的背景色。注意，如果线型是 THICK__WIDTH，即边框线条占 3 个点宽，则在调用 bar3d 函数时所指定的矩形必须稍小一点，以便整个窗口的边框仍显示在为窗口保留的屏幕范围内。

(5) 函数 view__win、chg__win 和 chgviewport 都是在切换到另一个窗口时要用到的，调用时都需要一个指向窗口结构的指针，它们是一个调用一个，功能依次减弱。函数 chgviewport 只负责把原窗口的线型等信息存储到窗口结构中，在切换后还负责根据新窗口结构中的信息设置当前线型，并使新窗口成为当前窗口。函数 view__win 牵涉到复杂的窗口堆栈处理，与第 4 章文本窗口工具包中的同名函数的作用相同。

图形窗口工具包头文件 gpopup.h

```
/*    gpopup.h    */
#define CTRWIN  999
#define xul  wc.viewinfo.left
#define yul  wc.viewinfo.top
#define xlr  wc.viewinfo.right
#define ylr  wc.viewinfo.bottom
enum     windowtype {popup,tile};
typedef  struct wincolors__struct {
         struct linesettingstype lineinfo;
         struct fillsettingstype fillinfo;
         struct textsettingstype textinfo;
         struct linesettingstype brdinfo;
         struct viewporttype      viewinfo;
         char    border__type;
         unsigned char border__color,text__color,hilite__color,back__color;
}wincolors;
typedef  struct winstruct {
         char      * name;
         void      * image;
```

```
        struct   winstruct  * under, * over;
        wincolors wc;
        int      wd,ht;
        int      xsave,ysave;
        enum     windowtype wtype;
}windesc;
extern  windesc   * base__win;
extern  windesc   * curr__win;
extern  wincolors defcolors;
extern  int       graphdriver;
extern  int       graphmode;
extern  char      * pathdriver;
#define rmv__win(w)  (view__win(w,0))
#define slct__win(w) (view__win(w,1))
extern  void      init__win(void);
extern  windesc   * draw__win(int x, int y, int wd, int ht, char * title,
                        enum windowtype wt,wincolors * wc);
extern  void      clr__win(void);
extern  void      view__win(windesc * this, int select);
extern  void      chgviewport(windesc * this);
extern  void      swap__image(windesc * w);
```

图形窗口工具包源文件 gpopup.c

```
/ *    gpopup.c   * /
#include <stddef.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <conio.h>
#include <string.h>
#include <process.h>
#include <graphics.h>
#include "gpopup.h"
#include "mouse.h"
#define MAX(a,b) ((a) > (b) ? (a) : (b))
#define MIN(a,b) ((a) < (b) ? (a) : (b))
windesc * base__win = NULL;
windesc * curr__win = NULL;
int        graphdriver = 0;
int        graphmode   = 0;
char       * pathdriver = "";
wincolors defcolors = {
    {SOLID__LINE, 0, NORM__WIDTH},
    {EMPTY__FILL, 0},
    {DEFAULT__FONT, HORIZ__DIR, 1, LEFT__TEXT, TOP__TEXT},
    {SOLID__LINE, 0, THICK__WIDTH},
    {0,0,0,0,1}.
    1,
```

```
    7,7,7,0
};
static windesc  *top_win;
static void     dispose_window_node(windesc  *w);
static windesc  *push_window_node(void);
static windesc  *make_window_node(void);
static void     chg_win(windesc  *this);
static void     fill_win(windesc  *w);

void init_win(void)
{
    initgraph(&graphdriver, &graphmode, pathdriver);
    base_win = make_window_node( );
    getviewsettings(&base_win->wc.viewinfo);
    base_win->wc.text_color = getcolor( );
    getfillsettings(&base_win->wc.fillinfo);
    getlinesettings(&base_win->wc.lineinfo);
    gettextsettings(&base_win->wc.textinfo);
    base_win->wc.brdinfo = base_win->wc.lineinfo;
    base_win->xsave = getx( );
    base_win->ysave = gety( );
    base_win->wtype = tile;
    base_win->wc.border_type =0;
    top_win  = base_win;
    curr_win = base_win;
}

windesc *draw_win(int x, int y, int wd, int ht, char *title,
                  enum windowtype wt, wincolors *wc)
{
    windesc  *w;
    int maxx, maxy;
    w = push_window_node( );
    maxx = getmaxx( );
    maxy = getmaxy( );
    wd = MAX(wd,3);
    wd = MIN(wd,maxx);
    ht = MAX(ht,3);
    ht = MIN(ht,maxy);
    if (x == CTRWIN) x = (maxx-wd) / 2;
    if (y == CTRWIN) y = (maxy-ht) / 2;
    x = MAX(x,0);
    y = MAX(y,0);
    if ((x+wd) > maxx) x = maxx-wd+1;
    if ((y+ht) > maxy) y = maxy-ht+1;
    w->wc = *wc;
    w->wd = wd;
    w->ht = ht;
    w->xul= x;
    w->yul= y;
    w->xlr= x + w->wd -1;
```

```
    w->ylr= y + w->ht -1;
    w->xsave = 1;
    w->ysave = 1;
    w->wtype = wt;
    w->name = strdup(title);
    if (wt == popup) {
        w->image = calloc(imagesize(w->xul,w->yul,
                              w->xlr,w->ylr),1);
        swap_image(w);
    }
    fill_win(w);
    chg_win(w);
    return w;
}

static void fill_win(windesc *w)
{
    mouse_off(1);
    chgviewport(base_win);
    setcolor(w->wc.border_color);
    setlinestyle(w->wc.brdinfo.linestyle,
                 w->wc.brdinfo.upattern,
                 w->wc.brdinfo.thickness);
    setfillstyle(SOLID_FILL,w->wc.back_color);
    if(w->wc.border_type&&w->wc.brdinfo.thickness==THICK_WIDTH)
        bar3d(w->xul+1,w->yul+1,w->xlr-1,w->ylr-1,0,0);
    else
        bar3d(w->xul,w->yul,w->xlr,w->ylr,0,0);
    chgviewport(curr_win);
    mouse_on(1);
}

void clr_win(void)
{
    fill_win(curr_win);
}

void view_win(windesc *this, int select)
{
    windesc *p;
    if (select && this ==top_win) return;
    mouse_off(1);
    if (this->wtype ==popup) {
        p = top_win;
        while(p != this) {
            swap_image(p);
            p = p->under;
        }
        swap_image(this);
        p = this;
        while(p != top_win){
```

```
                p = p->over;
                swap_image(p);
            }
        }
        if (this == top_win) {
            this->under->over = NULL;
            top_win = this->under;
        }
        else {
            this->under->over = this->over;
            this->over->under = this->under;
        }
        if (select) {
            top_win->over = this;
            this->under = top_win;
            top_win = this;
            swap_image(this);
            chg_win(this);
        }
        else {
            chg_win(top_win);
            dispose_window_node(this);
        }
        mouse_on(1);
}

static void chg_win(windesc * this)
{
        curr_win->xsave = getx( );
        curr_win->ysave = gety( );
        getfillsettings(&curr_win->wc.fillinfo);
        getlinesettings(&curr_win->wc.lineinfo);
        gettextsettings(&curr_win->wc.textinfo);
        chgviewport(this);
        setcolor(this->wc.text_color);
        setfillstyle  (this->wc.fillinfo.pattern,
                       this->wc.fillinfo.color);
        setlinestyle  (this->wc.lineinfo.linestyle,
                       this->wc.lineinfo.upattern,
                       this->wc.lineinfo.thickness);
        settextjustify(this->wc.textinfo.horiz,
                       this->wc.textinfo.vert);
        settextstyle  (this->wc.textinfo.font,
                       this->wc.textinfo.direction,
                       this->wc.textinfo.charsize);
        moveto(this->xsave,this->ysave);
        curr_win = this;
}

void chgviewport(windesc * this)
{
```

```
    int delta;
    if (this->wc.border_type)
        delta = this->wc.brdinfo.thickness;
    else delta = 0;
    setviewport(this->xul+delta, this->yul+delta,
                this->xlr-delta, this->ylr-delta,
                this->wc.viewinfo.clip);
}

void swap_image(windesc *w)
{
    char *temp_image;
    int  nbytes;
    if (w->wtype == popup) {
        nbytes = imagesize(w->xul,w->yul,w->xlr,w->ylr);
        temp_image = malloc(nbytes);
        chgviewport (base_win);
        mouse_off(1);
        getimage(w->xul,w->yul,w->xlr,w->ylr,temp_image);
        putimage(w->xul,w->yul,w->image,COPY_PUT);
        mouse_on(1);
        chgviewport(w);
        memcpy(w->image,temp_image,nbytes);
        free(temp_image);
    }
}

static windesc *make_window_node(void)
{
    /*
    Make_window_node allocates room for a new window structure,
    and initializes it's links to NULL.
    */
    windesc *q;
    q = (windesc *)malloc(sizeof(windesc));
    q->image = NULL;
    q->under = NULL;
    q->over  = NULL;
    return q;
}

static windesc *push_window_node(void)
{
    windesc *q;
    q = make_window_node( );
    top_win->over = q;
    q->under = top_win;
    top_win = q;
    return q;
}
```

```
static void dispose__window__node(windesc *w)
{
    if (w !=NULL) {
        if (w->wtype == popup) free(w->image);
        free(w->name);
        free(w);
    }
}
```

14.2.3 工具包应用举例：移动窗口

在下面这个简单例子中，画出了三个图形窗口，在三个窗口内分别画了一个矩形，圆和椭圆、所画的图形内分别标上了 1、2 和 3，表示窗口号码，如图 14-1 所示。如果使用

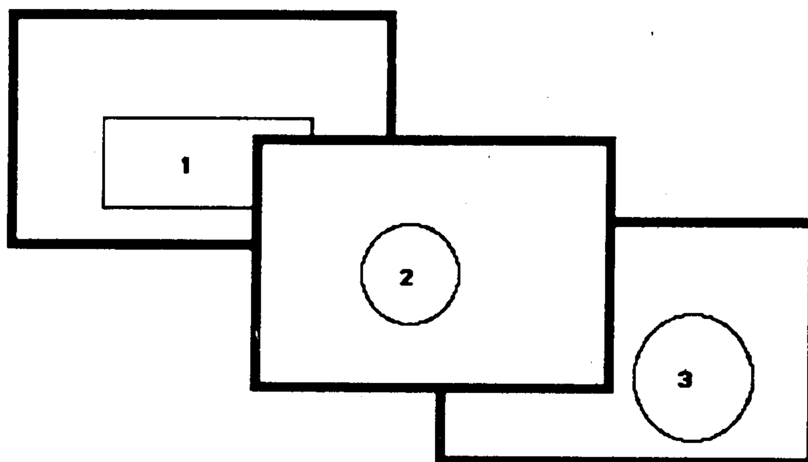

图 14-1 pop_up 图形窗口

键盘，用户可用数字键 1、2 和 3 来选择这三个窗口，用光标移动键来移动所选择的窗口，用 ESC 键退出这个程序。如果使用鼠标，用户可把鼠光标移到窗口的任一边框上，并按下左按钮来选择相应的窗口。如果移动鼠标的同时按下左按钮，则会拖曳当前窗口满屏幕移动。用户可以按右按钮退出这个程序。

如果显示系统是 EGA 或 VGA 的高分辨率彩色方式，则移动窗口的速度将是相当慢的。其原因就在于图形操作是位操作，而且需要更新的显示缓冲区容量也大得多。当 EGA 工作在 640 * 350 分辨率 16 种彩色时，显示缓冲区大约占 112K 字节，而文本方式却只需 4K 字节。解决的办法只能求助于更高速的图形卡。

下面是这个例子的源程序：

```
#include <stdlib.h>
#include <conio.h>
#include <bios.h>
#include <graphics.h>
#include "gpopup.h"
#include "mouse.h"
#define MAX(a,b)  ((a) > (b) ? (a) : (b))
```

```
#define MIN(a,b)  ((a) < (b) ? (a) : (b))
wincolors acolors, bcolors, ccolors;
int  mouse_on_border(windesc * * w, int * i, int * xofs, int * yofs);
void move_curr_win(int x, int y);

void main( ) {
    windesc * w[3];
    int x, y, xofs, yofs, i;
    unsigned int k=0;
    init_win( );
    init_mouse(MOUSE_OPTIONAL, graphdriver, graphmode);
    acolors = defcolors; acolors.back_color = BLACK;
    bcolors = defcolors; bcolors.back_color = BLACK;
    ccolors = defcolors; ccolors.back_color = BLACK;
    w[0] = draw_win(50, 60,200,50,"",popup,&acolors);
    rectangle(50,15,150,35);
    outtextxy(100,25,"1");
    w[1] = draw_win(70, 90,200,60,"",popup,&bcolors);
    circle(100,25,20);
    outtextxy(100,25,"2");
    w[2] = draw_win(90,110,200,50,"",popup,&ccolors);
    ellipse(100,25,0,360,40,15);
    outtextxy(100,25,"3");
    do {
        while (!(k = mouse_trigger(1)));
        if (k == LEFT_MOUSE_PRESS) {
            if (mouse_on_border(w, &i, &xofs, &yofs)) {
                slct_win(w[i]);
                while(button_state( )) {
                    mouse_grph_posn(&x,&y);
                    move_curr_win(x-xofs,y-yofs);
                }
            }
        }
        else {
            xofs = 0; yofs = 0;
            x = curr_win->xul;
            y = curr_win->yul;
            switch(k) {
                case UPKEY:
                    y -= 5;
                    move_curr_win(x,y);
                    break;
                case DOWNKEY:
                    y += 5;
                    move_curr_win(x,y);
                    break;
                case LEFTKEY:
                    x -= 5;
                    move_curr_win(x,y);
                    break;
```

```
                case RIGHTKEY:
                    x += 5;
                    move_curr_win(x,y);
                    break;
                case 0x0231:
                    slct_win(w[0]);
                    break;
                case 0x0332:
                    slct_win(w[1]);
                    break;
                case 0x433:
                    slct_win(w[2]);
                    break;
                default:
                    if (k != RIGHT_MOUSE_PRESS)
                        /* send character to current window */
                        /* mprintf("%c",lo(k)); */
                    break;
            }
        }
    } while (k != ESCKEY && k != RIGHT_MOUSE_PRESS);
    rmv_win(w[2]);
    rmv_win(w[1]);
    rmv_win(w[0]);
    mouse_reset( );
    closegraph( );
}

int  mouse_on_border(windesc **w, int *i, int *xofs, int *yofs)
{
    int x, y, j;
    mouse_grph_posn(&x,&y);
    for (j = 0; j<3; j++) {
        if(x >= w[j]->xul && x <= w[j]->xlr &&
           y >= w[j]->yul && y <= w[j]->ylr)     {
            if(abs(x - w[j]->xul) < 4 ||
               abs(x - w[j]->xlr) < 4 ||
               abs(y - w[j]->yul) < 4 ||
               abs(y - w[j]->ylr) < 4)      {
                *i = j;
                *xofs = x - w[j]->xul;
                *yofs = y - w[j]->yul;
                return 1;
            }
        }
    }
    return 0;
}

void move_curr_win(int x, int y)
{
```

```
    int xsave, ysave;
    if (x != curr_win->xul || y != curr_win->yul) {
        xsave = getx( );
        ysave = gety( );
        swap_image(curr_win);
        curr_win->xul = x;
        curr_win->yul = y;
        curr_win->xul = MAX(curr_win->xul,0);
        curr_win->xul = MIN(curr_win->xul,getmaxx( )-curr_win->wd);
        curr_win->yul = MAX(curr_win->yul,0);
        curr_win->yul = MIN(curr_win->yul,getmaxy( )-curr_win->ht);
        curr_win->xlr = curr_win->xul + curr_win->wd -1;
        curr_win->ylr = curr_win->yul + curr_win->ht -1;
        swap_image(curr_win);
        chgviewport(curr_win);
        moveto(xsave, ysave);
    }
}
```

14.3 图形方式下输出文本的若干问题

在图形方式下输出文本基本上有两种办法。第1种是调用普通的屏幕文本输出函数实现，如 putchar、puts、printf 等。这些标准输入输出函数最终是调用 BIOS 去实现的，而 BIOS 在所有图形方式下(除 Hercules 图形卡外)都支持文本输出。除了方便之外，使用标准输入输出函数使程序既可在图形方式下又可在文本方式下工作(DOS 的 DIR 命令就是这样的一个典型)，可保留输入输出重定向功能，可用 printf 输出格式化的文本。但这些标准输入输出函数对图形的支持是有限的，它们用电传机的滚动方式来输出文本，它们不支持彩色，对不同字库不同大小字符的支持也极其有限。

正是为了弥补这些不足，Turbo C 提供了另一组专用于图形方式下输出文本的函数(见 14.1.5 节)。这些函数支持不同的字库、不同的字形大小、不同的文本对齐方式和不同的颜色，但是它们仍然有许多不足，主要表现在:

- 不支持格式化文本输出。
- 不支持滚动.
- 重写已在屏幕上的文本不能把文本擦除干净。
- 不支持文本的加亮特性，从而不容易实现菜单。

下面分别讨论这些问题的解决办法。

14.3.1 格式输出

为了在图形方式下输出格式化文本，可调用 vsprintf 函数对文本进行格式化，存入一个缓冲区，然后再调用 outtext 或 outtextxy 函数把缓冲区的内容送往屏幕，如下例所示:

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

```
#include <conio.h>
#include <stdarg.h>
#include <graphics.h>
void outfmttext(int x, int y, char * fmt, ...);
void main( ) {
     int gd = DETECT, gm = 0, age = 30;
     initgraph(&gd,&gm,"");
     outfmttext(50,50,"Sally is %d years old",age);
     getch( );
     closegraph( );
}
void outfmttext(int x, int y, char * fmt, ...)
{
     va_list arg_ptr;
     char t[255];
     va_start(arg_ptr,fmt);
     vsprintf(t,fmt,arg_ptr);
     va_end(arg_ptr);
     outtextxy(x,y,t);
}
```

注意，这个例子不支持换行、TAB 等一些文本文件中常用的控制字符，因为 outtextxy 把这些字符当作普通图形字符。为支持这些控制字符，必须对这个例子加以修改。

14.3.2 重写

先看一个例子:

```
#include <conio.h>
#include <graphics.h>
char msg1[ ] = "Baby, come here";
char msg2[ ] = "I am the ape man";
void main( ) {
     int gd = DETECT, gm = 0;
     initgraph(&gd,&gm,"");
     settextstyle(DEFAULT_FONT, HORIZ_DIR, 1);
     outtextxy(0,0,msg1);
     getch( );
     outtextxy(0,0,"I am the ape man");
     getch( );
     closegraph( );
}
```

运行这个例子就会发现，在写第 2 次信息时，它仅与已在屏幕上的第 1 次信息进行逻辑"或"，并未把第 1 次信息擦干净。

解决这个问题的办法有三种。(1) 定义一个视口，使它正好与显示第 1 次信息所占的面积一致，在写第 2 次信息前先调用 clearviewport 函数清除这个视口。(2) 用 bar 函数

画一个条形图，条的大小和位置正好与显示第 1 次信息所用的面积一致，填图颜色与背景色一致，这就把第 1 次信息清除干净了。(3) 通过 getimage 函数取得第 1 次显示信息后屏幕映像的位图，然后再通过 putimage 以逻辑"异或"方式写回到屏幕上。

在这三种办法中，第 2 种办法是最好的，它不影响当前视口的设置(不定义一个新的视口)，不要求为保存位图而分配内存，容易为文本重新设置背景色。所有这三种办法都需要知道显示第 1 次文本所占屏幕的宽度和高度，textheigh 和 textwidth 函数可以帮助实现这一点。

按照这些考虑对上面的例子进行修改，其结果如下：

```
#include <conio.h>
#include <graphics.h>
char msgl[ ] = "Baby, come here";
char msg2[ ] = "I am the ape man";
void main( ) {
     int gd = DETECT, gm = 0, th, tw;
     initgraph(&gd,&gm,"");
     settextstyle(DEFAULT_FONT, HORIZ_DIR, 1);
     outtextxy(0,0,msg1);
     getch( );
     th = textheight(msg1);
     tw = textwidth(msg1);
     setfillstyle(SOLID_FILL, 0);
     bar(0,0,tw,th);
     outtextxy(0,0,"I am the ape man");
     getch( );
     closegraph( );
}
```

如果使用的是向量字库，这样修改以后还会出现问题。虽然 textwidth 和 textheight 能够正确地计算出文本的宽度和高度，但文本的位置却是错误的。当输出小写 g 或 y 之类的字母时，字母的尾部总是写到下一行上。如果上例中的字型设置为

```
settextstyle(TRIPLEX_FONT,HORIZ_DIR,1);
```

则第 1 个信息的字母 y 和逗号都将不会被擦除干净。拖到下一行尾部的高度大约为总高度的 1／4，为了修正这一点，需要修改调用 bar 函数时所用的参数。修改后的结果如下所示：

```
#include <conio.h>
#include <graphics.h>
char msgl[ ] = "Baby, come here";
char mag2[ ] = "I am the ape man";
struct textsettingstype txtinfo;
void main( ) {
     int gd = DETECT, gm = 0, th, tw;
     initgraph(&gd,&gm,"");
     settextstyle(TRIPLEX_FONT, HORIZ_DIR, 0);
     outtextxy(0,0,msg1);
```

```
        getch( );
        th = textheight(msg1);
        tw = textwidth(msg1);
        gettextsettings(&txtinfo);
        setfillstyle(SOLID_FILL, 0);
        if (txtinfo.font) bar(0,th / 4,tw,th+th / 4);
        else bar(0,0,tw,th);
        outtextxy(0,0,"I am the ape man");
        getch( );
        closegraph( );
    }
```

14.3.3 加亮

在上述两个问题解决的基础上，加亮问题也就容易解决了。只需计算出文本的大小，在原来文本的位置画一个实心彩色条，然后再输出文本。当然，要考虑到向量字库的特殊性。下面就是这样一个例子:

```
#include <graphics.h>
#include <conio.h>
#include <stdio.h>
void hilite(int x,int y,char * text,int back_color,int fore_color);
void main( ) {
    int gd = DETECT, gm = 0;
    initgraph(&gd,&gm,"");
    settextstyle(SANS_SERIF_FONT,HORIZ_DIR,0);
    hilite(50,50,"Hello there baby", WHITE, BLACK);
    getch( );
    closegraph( );
}
void hilite(int x,int y,char * text,int back_color,int fore_color)
{
    struct textsettingstype txtinfo;
    int th, tw;
    gettextsettings(&txtinfo);
    th = textheight(text);
    tw = textwidth(text);
    if (txtinfo.font) th++;
    setfillstyle(SOLID_FILL,back_color);
    bar(x,y,x+tw-1,y+th-1);
    setcolor(fore_color);
    if (txtinfo.font) {
        outtextxy(x,y-th / 4,text);
    }
    else {
        outtextxy(x,y,text);
    }
}
```

14.3.4 滚动

滚动问题既牵涉到特殊控制字符，又牵涉到重写，显然更加复杂一些。但是，其解决办法实际上与前面几个问题一样，先用 textheight 函数计算出 1 行文本的高度，然后用 getimage 和 putimage 使图形上移 1 行。

14.4 用 XOR 方式画旋转橡皮筋

当用 putimage 函数往屏幕上画图时，可以与已在屏幕上的图像进行逻辑“或”、“异或”、“与”、“非”等操作。遗憾的是，当通过画线、画矩形、画圆、画椭圆等函数画图时，Turbo C 不提供这些操作。异或操作是一种很重要的操作，尤其对于动画设计，因为一次异或操作可以显示一个图像，再一次异或操作又可以清除这个图像，恢复原来的样子。

解决这个问题较好的办法就是重写 Turbo C 的这些画图库函数，使它像 putimage 函数一样可以接收操作方式参数。对于 EGA 和 VGA，还有另一种更为简捷的办法。EGA 和 VGA 中有数十个可编程的寄存器，用以控制显示卡的行为或读取显示卡的状态。详细介绍这些寄存器的编程不属于本书的范畴，下面只介绍数据移位／功能选择寄存器，其中的位 3 和位 4 与 putimage 函数的参数 op 所起的作用一样，用以控制与已在屏幕上的映像数据做何种操作，共有 4 种选择：

位 4	位 3	操作
0	0	替换
0	1	逻辑″与″
1	0	逻辑″或″
1	1	逻辑″异或″

这个寄存器的其它 6 位用于别的目的，但正常情况下被设置为 0。为了设置为″异或″方式，应该用下列语句：

```
outp(0x3ce,3);
outp(0x3cf,0x18);
```

为了回到″替换″方式，应该用下列语句：

```
outp(0x3ce,3);
outp(0x3cf,0);
```

如果以″异或″方式画一条线，则调用一次画出该线，再调用一次又擦掉了。遗憾的是，如果所画的是一条水平线，则不能正确恢复。这是因为，Turbo C 的 line 函数在画水平线时，它本身在某些点上已经执行了两次″异或″操作，从而使″异或″操作不配对。解决的办法是在画水平线时不用 line 函数，而用 getimage 和 putimage 函数对先保留然后再恢复映像，在恢复时就做了一次″异或″操作。

这又引起另外一些问题。getimage 和 putimage 函数要求所处理的矩形块的宽度和高

度必须至少为两个像点，所以，即使是处理一个像点宽的水平线，也必须扩大成两个像点宽，这样把线附近的这一行也一起处理了。另外，line 函数不要求起点坐标小于终点坐标，而 getimage 和 putimage 却要求左上角坐标小于右下角坐标。

下面这个演示程序首先在屏幕上显示一行提示信息，然后用户移动鼠标，在他选中的位置按一下左按钮，这个点就将成为起点。接着，用户可以继续移动鼠标，在移动过程中，鼠光标所在位置和起点位置之间就会画一条线。鼠光标稍一移动，原来那条线就将消失，新位置又将画一条线。这样可以来回比较和挑选。一旦选中了某一位置，再按一下左按钮，这条线就固定不动，亦不再被擦除，并且鼠光标的这个新位置就成为新的起点(支点)，继续画下一条线，直至用户按右按钮，结束程序的运行。

```
#include <math.h>
#include <dos.h>
#include <conio.h>
#include <graphics.h>
#include <alloc.h>
#include "mouse.h"
void *save_image;
int  been_saved=0;
char msg[ ]="Thisis a test of rubber banding lines";
void rubber_line(int px,int py,int ex,int ey, int drawflag);
void main( )
{
    int gd=EGA,gm=EGAHI;
    int px,py,ex,ey,mx,my;
    unsigned int k;
    initgraph(&gd,&gm,"");
    init_mouse(MOUSE_NEEDED,gd,gm);
    save_image=calloc(imagesize(0,0,getmaxx( ),1),1);
    while(!mouse_trigger(1));
    px=mouse_grph_x;
    py=mouse_grph_y;
    ex=px;
    ey=py;
    mouse_off(1);
    settextstyle(TRIPLEX_FONT,HORIZ_DIR,0);
    outtextxy((getmaxx( )-textwidth(msg))/2,getmaxy( )/2,msg);
    mouse_on(1);
    setlinestyle(DOTTED_LINE,0,1);
    rubber_line(px,py,ex,ey,1);
    do {
        mouse_grph_posn(&mx,&my);
        if(k=mouse_trigger(1)) {
            mouse_off(1);
            rubber_line(px,py,ex,ey,0);
            setcolor(GREEN);
            line(px,py,ex,ey);
            setcolor(WHITE);
            mouse_on(1);
```

```
                px=ex;
                py=ey;
            }
            else {
                if(mx!=ex || my!=ey) {
                    mouse__off(1);
                    rubber__line(px,py,ex,ey,0);
                    ex=mx;
                    ey=my;
                    rubber__line(px,py,ex,ey,1);
                    mouse__on(1);
                }
            }
    }while(k!=RIGHT__MOUSE__PRESS);
    mouse__reset( );
    closegraph( );
}

void rubber__line(int px,int py,int ex,int ey,int drawflag)
{
    int sx,fx;
    if(py=ey) {
        if(px>ex) {
            sx=ex;
            fx=px;
        }
        else {
            sx=px;
            fx=ex;
        }
        if(drawflag) {
            getimage(sx,py,fx,py+1,save__image);
            been__saved=1;
            line(sx,py,fx,py);
        }
        else {
            if(been__saved) putimage(sx,py,save__image,COPY__PUT);
            been__saved=0;
        }
    }
    else {
        outp(0x3ce,3);
        outp(0x3cf,0x18);
        line(px,py,ex,ey);
        outp(0x3ce,3);
        outp(0x3cf,0);
    }
}
```

注意，在这个程序里，setimage 和 putimage 本来应该成对使用，但是，如果在水平

方向上按左按钮的时机或次数不合适，则有可能不成对使用。，程序中引入了一个全局变量 been__saved，以保证成对使用。这个程序所画的线也必须只是一个像点宽。

第 15 章　混合模式和混合语言编程

15.1　混合模式编程

15.1.1　概述

Turbo C 共支持 6 种编译模式：微模式，小模式，中模式，紧凑模式，大模式，巨模式。其中，中模式、大模式和巨模式又合称为大程序模式，因为在这三种模式下，程序指针是 32 位，每个模块都有自己的 CS 段。紧凑模式、大模式和巨模式又合称为大数据模式，因为在这三种模式下，数据指针是 32 位，缺省数据段、堆栈段、堆段三者彼此分开，且在巨模式下，每个模块都有自己的 DS 段(缺省数据段)。

但是，在某些特殊情况下，可能需要混合使用这些模式，即需要对函数和程序数据指针进行修正。Turbo C 中提供了 7 个这样的修饰符：near，far，huge，__cs，__ds，__es，__ss。

near、far 和 huge 可用来对函数和程序数据指针进行修饰。当这 3 个修饰符用于修饰程序指针时，分别说明被修饰的函数在被调用时要求用近(或远)调用指令，在返回时用近(或远)返回指令。huge 与 far 在调用与返回方面是一样的，只是 huge 函数可以设置 DS 为一个新的值。

修饰符__cs、__ds、__es 和__ss 只用于近数据指针，说明该数据指针是和哪一个段寄存器相关联的。例如：

```
char  __ss  *p
```

则使 p 含有相对于堆栈段的 16 位偏移量。

在一个给定的程序内，根据编译模式的不同，数据指针可以是近或远，但不能两者都有。如果是近，数据指针则自动与 DS 寄存器相关联，函数和程序指针则自动与 CS 寄存器相关联，如下表所示：

编译模式	函数和程序指针	数据指针
微	nera，__cs	near，__ds
小	near，__cs	near，__ds
中	far	near，__ds
紧凑	near，__cs	far
大	far	far
巨	far	far

15.1.2 说明一个函数为 near 或 far

在某些情况下，可能需要推翻编译模式缺省规定的函数类型。例如，假设编译时使用的内存模式是大模式，但在源程序中却有如下这样一个递归函数：

```
double power(double x, int exp)
{
    if(exp< =0)
        return(0);
    else
        return(x * power(x, exp-1));
}
```

这个函数计算指数函数值。每次递归调用 power 函数本身时，尽管是同一个函数，在同一个模块内，却必须用远调用，这无疑浪费了堆栈空间，也降低了程序执行的速度。如果显式地说明这个函数为 near，迫使对这个函数的调用都是近调用，那么这些无用的开销就没有了。

```
double near power(double x,int exp)
```

这样说明以后，所有对这个函数的调用都成了近调用。但是，有两点值得注意。第 1，在第一次使用这个函数前，必须先定义这个函数为近函数或说明它是近函数，否则编译程序无法知道对这个函数的调用应使用近调用。第 2，如果使用会产生多个代码段的内存模式，则只有在 power 函数所属的那个模块内的各函数可以调用 power 函数，其它模块有它们自己的段地址，不可能调用这个模块内的近函数。既然如此，不如把函数说明为静态的 (static)，别的模块也就看不到这个函数了。如下所示：

```
slatic doublc near power(double x, int exp)
```

相反的情况也是存在的，这就是所选定的编译内存模式只产生一个代码段(小模式，紧凑模式)，但用 far 显式地说明一个函数是远函数。一个函数是远函数意味着它的返回指令一定是远返回指令，所以调用也必须是远调用。

15.1.3 说明一个指针为 near、far 或 huge

与上面讲的函数调用类似，数据指针有时也需要显式地加以说明。或者是为了避免不必要的开销，从而用显式的 near 去推翻隐式的 far；或者是为了读写缺省数据段以外的某一内存区域，从而用显式的 far 或 huge 推翻隐式的 near。本书的许多程序都有这种例子。但是，如果使用不当，则可能引起问题。例如：

```
void myputs(s)
char * s;
{
    int i;
    for(i=0; s[i]!=0;i++)puts(s[i]);
}
main( )
{
    char near * mystr;
    mystr="Hollo,world\n";
    myputs(mystr);
}
```

如果用小模式编译这个程序，则不会出现问题。事实上，说明 mystr 指针为 near 是

多余的。如果再用大数据模式编译这个程序，则就出问题。由于显式说明为 near，故在 main 函数内 mystr 指针仍然是 near；而由于隐式规定为 far，故在 myputs 函数内所期望的指针却是 far，这显然不一致。

解决这个问题的办法是按新的 C 语言风格定义 myputs 函数，如下所示：

```
void myputs(char * s);
{
    / * body of myputs * /
}
```

采取这种格式后，C 编译程序在进行编译时就会知道 myputs 函数所期望的是一个指向字符的指针。仍以显式说明为 near 而编译模式隐式规定为 far 来说明，因为编译模式为大数据模式，所以编译程序知道指针必须是 far，于是在编译 myputs(mystr)这条语句时，在自动往堆栈压入一个偏移量后，又压入一个 DS 寄存器的值，从而形成一个远指针，保证了指针的一致性。

反之，如果在 main 函数内显式说明 mystr 指针是 far，而编译时指定的编译模式是小数据模式，从而在函数 myputs 内所期望的指针是近指针，这同样会引起问题，解决的办法也是采取新的 C 函数定义格式。

总之，借助于新的 C 函数定义风格，使得在编译时，根据所用编译模式的缺省规定(除非被显式地推翻)，就可以知道被调用函数期望着的调用和指针(远或是近)，从而采用正确的调用法则，不管调用函数本身是远还是近，也不管被传递的指针在调用函数内是远还是近。应该记住：如果某一指针被显式地修饰过了，则所有通过这一指针调用的函数都必须按新的 C 函数定义风格编写。本书的全部例子都是按新的定义风格写的。

15.1.4 使用库文件

Turbo C 提供了三类库文件：启动库文件，数学库文件，其它标准的运行时库文件。对于不同的编译模式，这三类库文件是不同的，如下表所示：

编译模式	启动库文件	数学库文件	标准库文件
微	COT.OBJ	MATHS.LIB	CS.LIB
小	COS.OBJ	MATHS.LIB	CS.LIB
紧凑	COC.OBJ	MATHC.LIB	CC.LIB
中	COM.OBJ	MATHM.LIB	CM.LIB
大	COL.OBJ	MATHL.LIB	CL.LIB
巨	COH.OBJ	MATHH.LIB	CH.LIB

Turbo C 还提供了两个与编译模式无关的库文件：fp87.lib 和 emu.lib，分别用于有和没有硬件协处理器(而又需要浮点运算)的场合。

Turbo C 提供了三种编译和连接的手段。第 1 种是使用集成开发环境 tc.exe。第 2 种是使用命令行程序 tcc.exe 编译和连接。第 3 种是使用命令行程序 tcc.exe 编译，但不连接，然后通过 tlink 进行连接。

对于前面两种办法，只要指定编译模式(缺省为小模式)，它们就会自动决定用哪一个库文件和以什么次序连接这些库文件。实际上，从 TCC 的命令行语法可以看出，在最简单的情况下，只需指明被编译的文件名，而无需加任何选择项，尽管有数十个选择项可供挑选。

第 3 种办法则不然，在使用 tlink 时必须把要用到的所有库文件显式地写出来，并且还要保证次序正确。tlink 的命令行语法如下所示:

```
tlink c0x <myobjs>,<exe>,[map],<mylibs>,[emu|fp87 mathx] cx
```

<myobjs>、<exe>、[map]、<mylibs>分别指用户的目标文件名(可以多个，以空格隔开)、生成的可执行文件名、生成的映像文件名、所用的用户库文件名(可以多个，以空格隔开)。而字段 c0x、[emu|fp87]、[mathx]、cx 就是上面提到的 Turbo C 的几种库文件，其中 x 随所用编译模式而定，可以是 s、c、m、l 和 h，对于 c0x，还可以是 c0t。

一般不会用第 3 种办法编译和连接文件。但是，它提供了把不同编译模式下生成的目标文件连接的办法。

15.1.5 不同编译模式所生成模块的连接

如果不采取特殊处理办法，不同编译模式所生成的各模块显然不能连接到一起。解决的办法是再一次利用新的 C 函数定义风格。假设以大模式编译上面提到的函数 myputs，然后建立一个头文件 myputs.h，头文件中包括如下说明语句:

```
void far myputs(char far *s);
```

现在，再建立一个主函数 main(假设这个文件名为 mymain.c)，在它内部调用函数 myputs，并加了一条#include ″myputs.h″语句，如下所示:

```
#include <stdio.h>
#include <myputs.h>
mymain( )
{
    char near *mystr;
    mystr = "Hello,world\n";
    myputs(mystr);
}
```

不管以什么编译模式编译 mymain.c，它都知道函数 myputs 期望的是远调用，所需要的指针是远指针，从而保证了程序的正确运行。

但是，在这个例子中，main 函数中没有调用 Turbo C 的标准库函数，只有 myputs 用到了 Turbo C 的标准库函数。假如两个模块是用不同编译模式编译的，又都用到了 Turbo C 的库函数，那么就须为各模块中用到的所有 Turbo C 库函数重新写一个说明，说明这些函数一律是远函数，它们用到的指针一律是远指针。例如，可以为 printf 函数写这样一个说明:

```
int far cdecl printf(char far *format,...);
```

把所有这些说明集中在一个头文件，如头文件 fstdio.h 中，在所有各模块的源文件中都加入下面这条语句:

```
#include  <fstdio.h>
```

这样，尽管各模块以不同的编译模式编译，但它们用到的 Turbo C 库函数却都属于大模式的库，在用 tlink 连接模块时，只要用 mathl.lib 和 cl.lib 就可以了。至于用哪一个启动库文件 c0x.lib，则取决于 main 函数所在模块的编译模式。

15.2 C 和汇编语言混合编程

这一节我们将解释怎样从 C 语言内调用汇编语言程序和怎样从汇编语言程序内调用 C 语言程序，我们还将讨论有关“段”和“连接程序”的一些问题，如：全局变量存放在哪里？怎样存取这些变量？段是怎样划分的？又是怎样组合的？等等。

C 和汇编语言混合编程的具体方法随 C 编译和宏汇编的不同而有所不同，本书是根据 Turbo C 和 TASM 来分析的。

本节讲的混合编程是把 C 程序和汇编程序各自单独作为一个模块来处理，只是在连接时才合到一起。Turbo C 还提供了其它混合编程的办法，见 15.3 节。

在 2.3 节，我们曾经介绍过 C 语言的调用规则，但那是针对 C 函数调用另一个 C 函数，C 和汇编语言混合编程除应遵守这些规则外，还要考虑到汇编语言的许多特点。本节将首先介绍混合编程必须解决的 6 个问题，最后以两个例子来说明这个过程。

15.2.1 段的组合

所谓段的组合，简单说来，就是如何让连接程序把两种语言的代码和数据分别合到一起，成为一个可执行文件。连接程序是凭名字来识别代码和数据段的，并决定它们之间的组合关系。为此，我们必须明了 C 语言和汇编语言怎样命名各自的段，然后才能考虑如何组合。

15.2.1.1 汇编语言的段和组

在汇编语言中，各段的格式如下所示：

```
段名     SEGMENT   [对齐类型] [组合类型] ['类别']
         .
         .
         .
         各条语句
         .
         .
         .
段名     ENDS
```

段名是源程序／数据段的名字。程序员必须为每个程序／数据段取一个名字，但允许不同的源程序／数据段有同样的名字。

对齐类型是指这一段的起始地址的边界对齐要求，有 BYTE(字节)、WORD(字)、DWORD(双字)、PARA (节)、PAGE(页，256 字节) 几种。缺省的对齐类型是 PARA。

组合类型指的是对名字相同的段的处理办法，有 PUBLIC、COMMON、STACK、PRIVATE 几种。PUBLIC 是把名字相同的段一个接一个地（仍保证对齐类型的要求）放在连续的内存块里，形成一个"大段"，共用一个段寄存器。偏移量不再是相对于各源程序段段头，而是相对于这个共同的段寄存器。"大段"的长度不应超过 64K 字节。STACK 的处理办法类似，只是把形成的这个"大段"当作堆栈段处理，并且自动初始化 SS 和 SP 寄存器，使之指向这个堆栈段。COMMON 是把各同名段放在从同一地址开始的内存里，实际上是重叠地放在一起，最后形成的大段的长度就等于最长的那个段的长度。"AT 地址"是指定这段从某一固定地址开始。PRIVATE 指明各段不发生关系。缺省的组合类型是 PRIVATE。

类别名是给同一类源码（例如都是代码或都是数据）取的名字。类别名应用单引号括起来，属于同一类别的各段在内存连续存放，但不使用共同的段寄存器。类别名没有固定的名字，由程序员取名，甚至可以不赋予类别名。

宏汇编 TASM 中还提供了另外一种段的组合办法，这就是伪指令 GROUP。GROUP 伪指令的一般使用格式是：

组名　GROUP　段名，…，段名

GROUP 伪指令把数个段组合成一组，这些段具有共同的段起始地址，一个组的长度也不能超过 64K 字节。但是，组不影响各段的装入顺序，同一组内的各段之间可以插入不属于本组的段。

15.2.1.2　Turbo C 的段和组

Turbo C 在编译一个源程序时，如果不加选择项来推翻缺省规定，则按如下规则处理：在小程序模式下，生成的代码的段名为"_TEXT"，类别为'CODE'。在大程序模式下，因为每一个模块都有自己的代码段，所以生成的代码段的段名为文件名再加"_TEXT"，类别为'CODE'。在除巨模式的所有其它模式下，生成的已初始化全局变量段的段名为"_BSS"，类别为'BSS'；生成的堆栈段的段名为"STACK"，类别为'STACK'。在小数据模式下，"_DATA"、"_BSS"和"STACK"合成一个组，组名为 DGROOP，即这三个段合起来不能超过 64K 字节。在大数据模式下，只有"_DATA"和"_BSS"是合在一起的，组名仍为 DGROOP，即全局数据最多可达 64K 字节，堆栈也可为 64K 字节。在巨模式下，因为每个模块都可有自己的全局变量段，所以段名为文件名再加"_DATA"，此时没有"_BSS"段，也没有 DGROOP 这个组。

至于堆，因为它是动态分配的，编译时不会为它分配一个段。在三种小数据模式下，近堆夹在"_BSS"和"STACK"之间，故在 DGROUP 组，也用 DS 做段寄存器。其中，小模式和中模式除了近堆之外，还有远堆。三种大数据模式只有远堆，没有近堆。远堆没有固定的段寄存器，段地址只能存放在远指针内。

下面依照从低地址到高地址的顺序，分别列出 6 种编译模式下各自的内存分配情况。表中"所属物理段"一栏，数字相同的表示在同一物理段内，每一物理段最大可达 64K 字节。

编译模式	逻辑段名	段的类型	组名	所属物理段	
微模式	__TEXT	CODE	DGROUP	1	CS=DS=SS
	__DATA	DATA	DGROUP	1	CS=DS=SS
	__BSS	BSS	DGROUP	1	CS=DS=SS
	(近堆)	——	DGROUP	1	CS=DS=SS
	STACK	STACK	DGROUP	1	CS=DS=SS
小模式	__TEXT	CODE		1	CS
	__DATA	DATA	DGROUP	2	DS=SS
	__BSS	BSS	DGROUP	2	DS=SS
	(近堆)	——	DGROUP	2	DS=SS
	STACK	STACK	DGROUP	2	DS=SS
	(远堆)	——	——	其余	远指针内
中模式	模块名 1__TEXT	CODE	——	1	模块 1 的 CS
	:	:	:	:	:
	:	:	:	:	:
	模块名 n__TEXT	CODE	——	n	模块 n 的 CS
	__DATA	DATA	DGROUP	n+1	DS=SS
	__BSS	BSS	DGROUP	n+1	DS=SS
	近堆	----	DGTOUP	n+1	DS=SS
	STACK	STACK	DGROUP	n+1	DS=SS
	远堆	——	DGROUP	其余	远指针内
紧凑格式	__TEXT	CODE	——	1	CS
	__DATA	DATA	DGROUP	2	DS
	__BSS	BSS	DGROUP	2	DS
	STACK	STACK	——	3	SS
	远堆	——	——	其余	远指针内
大模式	模块名 1__TEXT	CODE	——	1	模块 1 的 CS
	:	:	:	:	:
	:	:	:	:	:
	模块名 n__TEXT	CODE	——	n	模块 n 的 CS
	__DATA	DATA	DGROUP	n+1	DS
	__BSS	BSS	DGROUP	n+1	DS
	STACK	STACK	——	n+2	SS
	远堆	——	----	其余	远指针内

巨模式	模块名 1_TEXT	CODE	——	1	模块 1 的 CS
	:	:	:	:	:
	:	:	:	:	:
	模块名 n_TEXT	CODE	——	n	模块 n 的 CS
	模块名 1_DATA	DATA	——	n+1	模块 1 的 CS
	:	:	:	:	:
	:	:	:	:	:
	模块名 n_DATA	DATA	——	2n	模块 n 的 DS
	STACK	STACK	——	2n+1	SS
	远堆	——	——	其余	远指针内

15.2.1.3 段和组的连接

了解了 C 语言和汇编语言是如何处理段和组后，就可以设法使两种语言产生的模块正确地连接起来。一般的做法是：在编写汇编语言源程序时遵守 C 语言的缺省规定，即使汇编语言程序往 C 语言程序靠拢。这样编出来的汇编语言程序应如下所示：

```
<code>      SEGMENT   BYTE PUBLIC 'CODE'
            ASSUME   CS:<code>,DS:<dseg>
            :
            <code segment>
            :
<code>      ENDS
<dseg>      GROUP     _DATA,_BSS
<data>      SEGMENT   WORD PUBLIC 'DATA'
            :
            <initialized data segment>
            :
<data>      ENDS
_BSS        SEGMENT   WORD PUBLIC 'BSS'
            :
            <uninitialized data segment>
            :
_BSS        ENDS
            END
```

一般来说，"_BSS"段可以不要，而且在巨模式下，没有"_BSS"段，也没有 GROUP 组。

<code>、<data>、<dseg>分别为代码段名、全局变量段名、数据组组名。如果希望遵守 Turbo C 的缺省规定，这些名字应该命名如下：

编译模式	名字规定
微模式、小模式和紧凑模式	<code> = _TEXT
	<data> = _DATA
	<dseg> = DGROUP
中模式和大模式	<code> = 文件名_TEXT
	<data> = _DATA

	<dseg> = DGROUP
巨模式	<code> = 文件名_TEXT
	<data> = 文件名_DATA
	<dseg> = 文件名_DATA

15.2.2 变量和函数名的相互引用

C 语言和汇编语言混合还必须解决函数和变量的相互引用问题。应该特别注意的是，C 编译在其产生的目标文件中自动在名字(包括函数名和变量名) 前加了一道下划线。因此，汇编语言在引用 C 语言的函数和变量时，应该在其名字前加一道下划线，也可以把这下划线看作是名字的一部分。同样，为了使汇编语言程序中的函数和变量名能够被 C 语言所引用，这些名字的前面也必须加下划线。这一点与 PASCAL 语言不同。

为了在汇编语言程序中使用到 C 语言程序定义的变量和函数，在 C 语言一侧无需特殊处理，因为所有 C 函数和全局变量都可供外部使用，而在汇编语言一侧，必须用 EXTRN 关键字加以说明，其格式如下:

```
EXTRN        函数名: 函数类型
EXTRN        变量名: 变量类型
```

其中，函数类型可以是 near 和 far，变量类型可以是 BYTE、WORD、DWORD、QWORD、TBYTE，分别占 1、2、4、8、10 个字节。应该保证两方的类型一致。对于数组，变量类型是由一个数组元素的大小决定的。对于结构，变量类型是由经常使用的大小决定的。例如，为了使用 C 中的三个变量

```
int      i, jarry[10];
char     ch;
long     result
```

在汇编语言中，应有如下的变量定义语句:

```
EXTRN _i: WORD, _jarry:WORD, _ch:BYTE, _result:DWORD
```

为了在 C 语言程序中可以用汇编语言程序定义的变量和函数，在 C 语言一侧应该用 extern 说明，在汇编语言一侧应该用 PUBLIC 说明，其格式如下:

```
PUBLIC    函数名
PUBLIC    变量名
```

注意，这个格式中并没有规定函数和变量的类型，其类型是在定义它们时确定的，必须保证两侧的类型一致。如果变量是用来存储指针，则在定义时应遵守下列原则:

编译模式	**程序指针**	**数据指针**
微和小模式	DW _TEXT:XXX	DW DGROUP:XXX
紧凑模式	DW _TEXT:XXX	DD DGROUP:XXX
中模式	DD XXX	DW DGROUP:XXX
大模式	DD XXX	DD DGROUP:XXX
巨模式	DD XXX	DD XXX

其中，XXX 代表指针的地址。

15.2.3 参数传递规则

C 语言在调用一个函数时，传给函数的参数是按从右到左的顺序压入堆栈的。例如，如果发出调用"callit(a，b，c)"，则先压入 c，接着压入 b，最后压入 a。

有些类型的变量是先转换成另一类型后才压入堆栈的：字符型(char) 转换成整型(int)，无符号字符型(unsigned char)转换成无符号整型(unsigned int)，浮点型(float)转换成双精度型(double)。结构(structure) 是整个压入堆栈的，也是按相反顺序。对于数组，压入堆栈的是指向数组的指针。近指针是 16 位，远指针是 32 位。先压入段寄存器值，后压入偏移量。下面列出压入参数时所占的字节数。

类　型	转换后的类型	在堆栈中的字节数
char	int	2
unsigned char	unsigned int	2
short		2
unsigned short		2
int		2
unsigned int		2
long		4
unsigned long		4
float	double	8
double		8
near pointer		2
far pointer		4
array	指向数组的指针	4或2
structure		n

如果是从 C 中调用汇编，则汇编语言子程序的格式应如下所示：

```
mov    bp，sp
sub    sp，n1                      ；子程序内的自动变量
push   di                          ；留 n1 个字节
push   si
 :
 :
mov    参数 n2,[bp+偏移量]          ；取某一参数
 :
 :
mov    [bp−偏移量],自动变量 n3      ；存某一自动变量
 :
 :
pop    si
pop    di
mov    sp，bp
pop    bp
mov    ax，返回值                  ；送返回值
```

采用这种格式后，根据后面将要讲到的寄存器原则，在函数体内就可以对所有寄存器进行任意修改和使用。这里是借助 BP 作地址寄存器到堆栈中寻找操作数的。[BP+偏移量]用来取出调用者传来的参数，[BP−偏移量]用来存取函数体内开辟的局部变量区，即相当于 C 语言的自动变量区。

如果从汇编中调用 C，则汇编语言调用程序的格式应如下所示：

```
EXTRN     c函数名: near或far
:
:
mov       ax, 参数1
push      ax
mov       ax, 参数2
push      ax
:
:
call      C函数名            ; 从堆栈中去掉已用完的参数,
add       sp, n              ; n由诸参数占的字节数决定。
mov       答案, ax           ; 取返回值
```

当控制权由被调用函数返回到调用函数后，由发出调用的函数从堆栈中把参数清除掉，这一点与 PASCAL 语言不一样。

15.2.4 返回值传递规则

被调用函数的返回值是按如下原则传递给调用者的：如果返回值小于或等于 16 位，则存放在 AX 寄存器内。如果返回值是 32 位，则存放在 AX／DX 两个寄存器内。如果返回值大于 32 位，(如结构、浮点数和双精度数)，则存放在静态变量存储区，指向这个存储区的指针在 AX 寄存器内。下面列出了各类返回值的存放原则。当使用 AX／DX 两个寄存器时，如果是 32 位数，则 DX 存放的是高 16 位；如果是 32 位指针，则 DX 存放的是段地址。

类　　型	存放返回值的位置
char	AX
unsigned char	AX
short	AX
unsigned short	AX
int	AX
unsigned int	AX
long	AX／DX
unsigned long	AX／DX
structure(<4byte)	AX／DX
structure	静态存储区，指针在AX或AX／DX中
flota	静态存储区，指针在AX或AX／DX中
double	静态存储区，指针在AX或AX／DX中
near pointer	AX
far pointer	AX／DX

15.2.5 寄存器规则

8086／8088／80286 共有 14 个寄存器，从调用函数进入被调用函数，以及从被调用函数返回调用函数时，这些寄存器可能含什么值，哪些寄存器的值应保持不变，这也是在混合编程时必须弄清楚的问题。

在 C 编译中，大体上可以把这些寄存器分为三类。

第一类是 CS、IP、SS、SP、DS、ES。只要按照上面已阐述的各项原则正确地把两种语言的模块连接起来，正确进入被调用函数并返回到调用函数，就说明对 CS、IP、SS、SP 四个寄存器的处理是正确的。在被调用函数内，会改变 IP 和 SP 的值(隐式地)，如果必要，也可以改变 CS 和 SS 的值。在某些 C 编译内存模式下(只有一个数据段)，如果 C 语言模块和汇编语言模块连接正确，则从 C 语言进入汇编语言时，DS 寄存器已指向所要求的数据段，无需在汇编语言程序中再初始化 DS。如果所使用的 C 编译内存模式产生多个数据段，并且互相连接的 C 模块和汇编模块的数据不在同一数据段内，则在汇编语言程序中需要修改 DS，使之指向自己的数据段。但是，如果修改了 DS，则应在返回到 C 语言前，恢复入口时的 DS 寄存器值。至于 ES 寄存器，则没有任何约定。

第二类是 BP、SP、DI 寄存器。在汇编语言子程序内，必须一开始就把 BP 寄存器的值压入堆栈，在返回到 C 语言前再把这个值弹回 BP，保证 BP 的值不改变。C 编译一般用 SI 和 DI 寄存器来存放寄存器变量，在汇编语言子程序内也应保持它们不被改变。但是 C 编译也提供了一个编译选择项(–r–)，通过它可以禁止 C 编译使用寄存器变量，此时在汇编语言子程序内可以任意使用 SI 和 DI，而不用担心对 C 语言程序会产生什么影响。

第三类是 AX、BX、CX、DX 寄存器。这四个寄存器在汇编语言子程序内可以任意使用，尤其是 BX 和 CX 寄存器。AX 和 DX 寄存器还担任传递返回值的任务。因此，如果汇编语言子程序有返回值，则应保证 AX 和 DX 中存有正确的返回值。

15.2.6 混合编程示例

15.2.6.1 C 调用汇编

下面是一个最简单的 C 语言程序:

```
main( )
{
      printf("1+2 = %d.\n",add(1,2));
}
```

这个程序调用函数 add 求两数的和，并调用函数 printf 把这个和打印出来。其中，函数 add 就是一个用汇编语言写的子程序，其源程序如下:

```
_TEXT   SEGMENT BYTE PUBLIC 'CODE'
        ASSUME  CS:_TEXT
        :
        :
```

```
            public __add,
            a=4
            b=6
__add       proc near
            push bp
            mov bp,sp
            mov ax,[bp+a]
            add ax,[bp+b]
            pop bp
            ret
__add       endp
__TEXT      ENDS
            END
```

15.2.6.2 汇编调用 C

假设在 C 语言程序内已定义了函数

```
long docalc(9nt * factl, int fact2,int fact3)
```

如果在 C 语言中调用这个函数，则可能的调用语句是:

```
ans=docalc(&xval,imax,421);
```

在这条语句中，ans 是长整数，存放 32 位的返回值，xval 是一个整数，C 函数 docalc 要的是这个整数的地址，421 是一个整常数，imax 是一个整数。

如果这个 C 语言函数 docalc 是在小模式下编译的，则调用函数 docalc 的汇编语言程序的格式可能如下:

```
CONST1  EQU         421
__TEXT  SEGMENT     BYTE     PUBLIC     'CODE'
        ASSUME      CS:__TEXT,      DS:DGROUP
        EXTRN       __docalc:near
        ........
        mov         ax,CONST1
        push        ax
        push        imax
        lea         xval
        push        ax
        call        __docalc
        add         sp,6
        mov         ans,ax
        mov         ans+2,dx
        .........
__TEXT  ENDS
DGROUP  GROUP       __DATA,     __BSS
DATA    SEGMENT     WORD        PUBLIC     'DATA'
        .........
imax    DW          1234
xval    DW          5678
        ........
```

```
_DATA   ENDS
_BSS    SEGMENT         WORD      PUBLIC    "BSS"
        .........
ans     DD              ?
        .........
_BSS    ENDS
        END
```

如果 C 语言函数 docalc 是用大模式编译的，则调用它的汇编语言程序的格式可能如下：

```
CONST1    EQU           421
_TEXT     SEGMENT       BYTE      PUBLIC    'CODE'
          ASSUME        CS:_TEXT,           DS:DGROUP
          EXTRN         _docalc:far
            .......
          mov           ax,CONST1
          push          ax
          push          imax
          les           ax,xval
          push          es
          push          ax
          call          _docalc
          add           sp,8
          mov           ans,ax
          mov           ans+2,dx
            .......
_TEXT     ENDS
DGROUP    GROUP         _DATA,_BSS
_DATA     SEGMENT       WORD      PUBLIC    'DATA'
            .......
imax      DW            1234
xval      DW            5678
            .......
_DATA     ENDS

_BSS      SEGMENT       WORD PUBLIC   'BSS'
            .......
ans       DD        ?
            .......
_BSS      ENDS
          END
```

主要的改变有三点：把 EXTRN _docalc: near 改为 EXTRN _docalc:far；往堆栈中压入的是指向变量 xval 的 far 指针；把 add sp，6 改为 add sp，8。

15.3 行内汇编

汇编语言模块和C语言模块的连接并不是混合编程的唯一方法。Turbo C 还提供了另外3种与汇编语言接口的方法，且都不需把汇编语言单独写成一个模块。第1种方法是使用伪变量（见12.2.1节）。第2种方法是使用interrupt型函数（见13.5节）。第3种方法就是本节将要介绍的行内汇编。

所谓行内汇编，就是不脱离C语言环境，在一条C语言语句里调用汇编语言去做某一件事。为了在C程序内使用行内汇编，可以在编译时加-B选择项。如果没加-B选择项，则在遇到行内汇编语句时会报出一个警告错误信息，并自动以-B选择项重新启动。为了避免发生这样的问题，可在源程序内加一条#pragma inline语句。同时，还必须有一个汇编程序TASM。因为在遇到行内汇编语句时，编译程序会首先自动产生一个汇编语言源文件，然后调用TASM汇编这个源文件，产生.obj文件。

行内汇编语句的格式如下：

asm <操作码> <操作数> <；或换行符>

操作码可以是任何一条有效的8086指令或某些汇编伪指令，如db、dw、dd和extern。操作数可以是C的常数、变量和标号。行内汇编语句可以有两种结束办法，除了普通的C语句结束符号";"以外，还可以用换行符。同一文本行内可以有多条行内汇编语句，但一条行内汇编语句不能跨越多个文本行。注意，不能用分号";"来标识一个注解的开始，注解行应遵循C语言的规定，用／*......*／来标识。

每条行内汇编语句在编译时都当作一条正常C语句看待，如在程序

```
myfunc( )
{
     int i;
     int x;
     if(i>0)
          asm  mov x,4
     else
          i=7;
}
```

中，if语句是合法的C语句，因为行内汇编语句asm mov x,4 可以用换行符结束。

行内汇编的办法不仅简捷，而且有时功能也很强，请看下面这个例子：

```
int min(int v1,int v2)
{
          asm mov ax,v1
          asm cmp ax,v2
          asm jle minexit
          asm mov ax,v2
     minexit:
          return(_AX);
```

```
}
```

这个函数求出两个整数值中较小的一个。不管是以大模式还是以小模式编译，也不管是 PASCAL 调用规则还是 C 调用规则，这个函数都能正确工作。

如果用单独的汇编语言模块写这个函数，则根据 15.2 节的原则，则总要随相连接的 C 语言模块编译模式和调用规则的不同而进行相应地修改，并要仔细计算偏移量的大小和检查标识符拼写是否正确(min 或 MIN)。

对于行内汇编语句内使用的 C 标识符，Turbo C 会自动将其转变成相应的汇编语言操作数，在标识符前加下划线。所有 C 标识符都可以合法地使用，包括自动变量、寄存器变量和函数参数。

一般来说，当汇编语言操作数要求一个寄存器操作数时，只能使用寄存器名或 C 的寄存器变量。寄存器变量最多只能有两个，并总是用 si 和 di 实现。只有整数 int(包括短整数 short，无符号整数 unsigned)或两字节的的指针变量可以指定为寄存器变量。如果在函数内没有指定寄存器变量，则行内汇编语句可以任意使用 si 和 di 寄存器。这是因为，在进入和退出这个函数时，Turbo C 会自动保存和恢复 si 和 di 寄存器的内容，函数内的修改不会影响到调用者的运行。如果在函数内指定了寄存器变量，而行内汇编语句再修改 si 和 di，就会间接修改寄存器变量，是不妥的。

行内汇编语句的操作数也可以是 C 结构中的某个字段。引用办法有两种。一种是结构变量名再加字段名，中间用圆点符"."隔开。另一种是先把结构变量的起始地址送往某个地址寄存器，然后用该寄存器名(加方括号)再加字段名，中间也用圆点符"."隔开，如下所示:

```
struct mystruct {
     int a__a;
     int a__b;
     int a__c;
}myA;
myfunc( )
{
     .
     .
     .
     asm mov ax,myA.a__b
     asm mov bx,[di].a__c
     .
     .
     .
}
```

因为不同的结构变量中允许相同的字段名，所以当用第 2 种办法时，如果存在多个结构变量，则应在圆点符和字段名之间再加一个结构类型(带圆括号)，如:

```
asm  mov  bx, [di].(struct tm)tm__hour
```

行内汇编语句中可以用跳转指令（分直接跳转和间接跳转）。直接跳转只能跳转到函数内的某个标号处。因为行内汇编语句内不能加标号，所以只能跳转到C的标号，且只能近跳，不能远跳。如下所示:

```
int x( )
{
    a:
        .
        .
        .
        asm jmp a
        .
        .
        .
}
```

间接跳转是通过寄存器或变量来跳转的，可以远跳。远跳需要变量为4字节，可能需要加DWORD PTR之类的数据类型修饰符。

附录 1 操作符表

分类	操 作	操作符	结合性	优先级
选择	强制类型，参数表，圆括号	( )	从左到右	1(最高)
	数组下标	[ ]	从左到右	1
	结构字段存取	.	从左到右	1
		->	从左到右	1
单目操作	后加	++	从左到右	2
	后减	—	从左到右	2
	地址	&	从右到左	2
	位反	-	从右到左	2
	逻辑反	!	从右到左	2
	取负	−	从右到左	2
	取正	+	从右到左	2
	前加	++	从右到左	2
	前减	—	从右到左	2
	强制类型转换	(类型名)	从右到左	2
	长度计算	sizeof	从右到左	2
乘,除,模	整数取模	%	从左到右	3
	乘	*	从左到右	3
	除	/	从左到右	3
加,减	加	+	从左到右	4
	减	−	从左到右	4
移位	左移	<<	从左到右	5
	右移	>>	从左到右	5
关系	小于	<	从左到右	6
	小于或等于	<=	从左到右	6
	大于	>	从左到右	6
	大于或等于	>=	从左到右	6
	等于	==	从左到右	7
	不等于	!=	从左到右	7
位操作	与	&	从左到右	8
	异或	∧	从左到右	9
	或	|	从左到右	10
逻辑	与	&&	从左到右	11

	或	||	从左到右	12
条件	条件表达式	? :	从右到左	13
赋值	赋值	=	从右到左	14
	算术操作并赋值	+=	从右到左	14
		-=	从右到左	14
		*=	从右到左	14
		/=	从右到左	14
		%=	从右到左	14
	移位并赋值	>>=	从右到左	14
		<<=	从右到左	14
	位操作并赋值	&=	从右到左	14
		|=	从右到左	14
		∧=	从右到左	14
次序	逗号	,	从左到右	15

附录 2　键盘码表

扫描码	键名	ASCII / IBM 或扩展 ASCII 码				
(键号)		基	shift	Ctrl	Alt	NumLock
1	ESC	27	27	27		
2	1!	49	33		x120	
3	2@	50	64	0	x121	
4	3#	51	35		x122	
5	4$	52	36		x123	
6	5%	53	37		x124	
7	6^	54	94	30	x125	
8	7&	55	38		x126	
9	8*	56	42		x127	
10	9(	57	40		x128	
11	0)	48	41		x129	
12	-_	45	95	31	x130	
13	=+	61	43		x131	
14	<-	8		127		Backspace
15	=><=	9	x15			Tab
16	Q	113	81	17	x16	
17	W	119	87	23	x17	
18	E	101	69	5	x18	
19	R	114	82	18	x19	
20	T	116	84	20	x20	
21	Y	121	89	25	x21	
22	U	117	85	21	x22	
23	I	105	73	95	x23	
24	O	111	79	15	x24	
25	P	112	80	16	x25	
26	[{	91	123	27		
27	]}	93	125	29		
28	Enter	13	13	10		
29	Ctrl					
30	A	97	65	1	x30	
31	S	115	83	19	x31	
32	D	100	68	4	x32	

33	F	102	70	6	x33	
34	G	103	71	7	x34	
35	H	104	72	8	x35	
36	J	106	74	10	x36	
37	K	107	75	11	x37	
38	L	108	76	12	x38	
39	;:	59	58			
40	'"	39	34			
41	-	96	126			
42						Left Shift
43	|	92	124	28		
44	Z	122	90	26	x44	
45	X	120	88	24	x45	
46	C	99	67	3	x46	
47	V	118	86	22	x47	
48	B	98	66	2	x48	
49	N	110	78	14	x49	
50	M	109	77	13	x50	
51	,<	44	60			
52	.>	46	62			
53	/?	47	63			
54						Right Shift
55	* Prtsc	42		x114		
56	Alt					
57	Space	32	32	32	32	
58	Caps Lock					
59	F1	xa9	x84	x94	x104	
60	F2	x60	x85	x95	x105	
61	F3	x61	x86	x96	x106	
62	F4	x62	x87	x97	x107	
63	F5	x63	x88	x98	x108	
64	F6	x64	x89	x99	x109	
65	F7	x65	x90	x100	x110	
66	F8	x66	x91	x101	x111	
67	F9	x67	x92	x102	x112	
68	F10	x68	x93	x103	x113	
69	Num Lock					
70	Break			0		Scroll Lock

71	Home 7	x71	55	x119	7	55
72	8	x72	56		8	56
73	Pgup 9	x73	57	x132	9	57
74	−	45	45			45
75	<−4	x75	52	x115	4	52
76	5		53		5	53
77	−>6	x77	54	x116	6	54
78	+	43	43			43
79	End 1	x79	49	x117	1	49
80	2	x80	50		2	50
81	Pgdn 3	x81	51	x118	3	51
82	Ins 0	x82	48		0	48
83	Del	x83	46			46

附录 3

	函 数 格 式
void	abort(void);
int	abs(int x);
int	absread(int drive, int nsects, int lsect, void *buffer);
int	abswrite(int drive, int nsects, int lsect, void *buffer);
int	access(const char *path, int amode);
double	acos(double x);
int	allocmem(unsigned size, unsigned *segp);
void far	arc(int x, int y, int stangle, int endangle,int radius);
char	*asctime(const struct tm *tblock);
double	asin(double x);
void	assert(int test);
double	atan(double x);
double	atan2(double y, double x);
int	atexit(atexit_t func);
double	atof(const char *s);
int	atoi(const char *s);
long	atol(const char *s);
void far	bar(int left, int top, int right, int bottom);
void far	bar3d(int left,int top,int right,int bottom,int depth,int topflag);
int	bdos(int dosfun, unsigned dosdx, unsigned dosal);
int	bdosptr(int dosfun, void *argument, unsigned dosal);
int	bioscom(int cmd, char abyte, int port);
int	biosdisk(int cmd,int drive,int head,int track,int sector,int nsects, void *buffer);
int	biosequip(void);
int	bioskey(int cmd);
int	biosmemory(void);
int	biosprint(int cmd, int abyte, int port);
long	biostime(int cmd, long newtime);
int	brk(void *addr);
void	*bsearch(const void *key,const void *base,size_t nelem,size_t width, int(fcmp)(const void *, const void *));
double	cabs(struct complex z);
void	*calloc(size_t nitems, size_t size);
double	ceil(double x);
char	*cgets(char *str);

Turbo C 2.0 函数简表

所需头文件	简 要 说 明
stdlib.h	异常终止一个进程
math.h	求整数的绝对值
dos.h	读绝对盘扇区
dos.h	写绝对盘扇区
io.h	确定一个文件的可存取性
math.h	求反余弦值
dos.h	分配 DOS 内存段
graphics.h	画一段圆弧
time.h	把日期和时间转换为 ASCII 串
math.h	求反正弦值
assert.h,stdio.h	测试一个条件,当条件为 0 时流产
math.h	求反正切值
math.h	求 y / x 的反正切值
stdlib.h	登记一个函数为退出时应执行的函数
math.h	把字符串转换为浮点数
stdlib.h	把字符串转换为整数
stdlib.h	把字符串转换为长整数
graphics.h	画两维条形
graphics.h	画三维条形
dos.h	DOS 系统调用
dos.h	DOS 系统调用
bios.h	串行输入 / 输出
bios.h	BIOS 盘服务
bios.h	检查系统设备
bios.h	BIOS 键盘服务
bios.h	返回内存大小(K 字节)
bios.h	BIOS 打印服务
bios.h	读和设置 BIOS 时钟
alloc.h	改变分配给调用程序的数据段大小
stdlib.h	二进制查找一个数组
math.h	求复数的绝对值
stdlib.h	分配主内存
math.h	求不小于 x 的最小整数
conio.h	从控制台读字符串

```
int       chdir(const char *path);
int       chmod(const char *path, int amode);
int      _chmod(const char *path, int func, ...);
int       chsize(int handle, long size);
void far  circle(int x, int y, int radius);
unsigned  int _clear87(void);
void far  cleardevice(void);
void      clearerr(FILE *stream);
void far  clearviewport(void);
clock_t   clock(void);
int       close(int handle);
int      _close(int handle);
void far  closegraph(void);
void      clreol(void);
void      clrscr(void);
unsigned  int _control87(unsigned int new,unsigned int mask);
unsigned  coreleft(void);
double    cos(double x);
double    cosh(double x);
struct    country *country(int xcode, struct country *cp);
int       cprintf(const char *format, ...);
int       cputs(const char *str);
int       creat(const char *path, int amode);
int      _creat(const char *path, int attribute);
int       creatnew(const char *path, int mode);
int       creattemp(char *path, int amode);
int       cscanf(const char *format, ...);
char     *ctime(const time_t *time);
void      ctrlbrk(int (*handler)(void));
void      delay(unsigned milliseconds);
void      delline(void);
void far  detectgraph(int far *graphdriver,int far *graphmode);
double    difftime(time_t time2, time_t time1);
void      disable(void);
div_t     div(int numer, int denom);
int       dosexterr(struct DOSERROR *eblkp);
long      dostounix(struct date *d, struct time *t);
void far  drawpoly(int numpoints, int far *polypoints);
int       dup(int handle);
int       dup2(int oldhandle, int newhandle);
```

dir.h	改变当前目录
dos.h,io.h	改变文件存取方式
sys＼stat.h,io.h	改变文件存取方式
io.h	改变文件大小
graphics.h	画圆
float.h	清除浮点状态字
graphics.h	清除图形屏幕
stdio.h	复位错误指示器
graphics.h	清除当前视口
time.h	确定两次事件间的时间(s)
io.h	关闭文件
io.h	关闭文件
graphics.h	关闭图形系统
conio.h	在文本窗口中清除到本行尾
conio.h	清除文本方式窗口
float.h	操作浮点控制字
alloc.h	返回自由内存的大小
math.h	求余弦值
math.h	计算 hyperbolic 余弦值
dos.h	返回国家信息
conio.h	往屏幕上输出格式化信息
conio.h	往屏幕上输出字符串
dos.h,io.h	建立新文件或重写已存在的文件
sys＼stat.h,io.h	建立新文件或重写已存在的文件
dos.h,io.h	建立新文件
dos.h,io.h	建立临时文件
conio.h	从控制台输入格式化信息
time.h	把日期和时间转换为字符串
dos.h	设置 Ctrl_Break 处理程序
dos.h	延迟一段时间
conio.h	在文本窗口中删除一行
graphics.h	自动确定图形驱动程序和图形方式
time.h	计算两个时间的时间差
dos.h	禁止中断
stdlib.h	两个整数相除
dos.h	取扩展的 DOS 错误信息
dos.h	把日期和时间变为 UNIX 时间格式
graphics.h	画多边形的轮廓线
io.h	复制文件柄
io.h	强制复制文件柄

```
     char    * ecvt(double value, int ndig, int * dec, int * sign);
 void far      ellipse(int x, int y, int stangle, int endangle,int xradius,
                        int yradius);
     void    ____emit____();
     void      enable(void);
      int      eof(int handle);
      int      execl(char * path, char * arg0, ...);
      int      execle(char * path, char * arg0, ...);
      int      execlp(char * path, char * arg0, ...);
      int      execlpe(char * path, char * arg0, ...);
      int      execv(char * path, char * argv[]);
      int      execve(char * path, char * argv[], char * * env);
      int      execvp(char * path, char * argv[]);
      int      execvpe(char * path, char * argv[], char * * env);
     void    __exit(int status);
     void      exit(int status);
   double      exp(double x);
   double      fabs(double x);
 void far    * farcalloc(unsigned long nunits, unsigned long unitsz);
 unsigned      long farcoreleft(void);
     void      farfree(void far * block);
 void far    * farmalloc(unsigned long nbytes);
 void far    * farrealloc(void far * oldblock, unsigned long nbytes);
      int      fclose(FILE * stream);
      int      fcloseall(void);
     char    * fcvt(double value, int ndig, int * dec, int * sign);
     FILE    * fdopen(int handle, char * type);
      int      feof(FILE * stream);
      int      ferror(FILE * stream);
      int      fflush(FILE * stream);
      int      fgetc(FILE * stream);
      int      fgetchar(void);
      int      fgetpos(FILE * stream, fpos__t * pos);
     char    * fgets(char * s, int n, FILE * stream);
     long      filelength(int handle);
      int      fileno(FILE * stream)
 void far      fillellipse( int x, int y, int xradius, int yradius );
 void far      fillpoly(int numpoints, int far * polypoints);
      int      findfirst(const char * path, struct ffblk * ffblk, int attrib);
      int      findnext(struct ffblk * ffblk);
```

stdlib.h	把浮点数转换为字符串
graphics.h	画一段椭圆弧
dos.h	直接往目标码中插入一些值
dos.h	允许中断
io.h	检查文件尾
process.h	装人和运行其它程序
process.h	装人和运行其它程序
process.h	装人和运行其它程序
process.h	装人和运行其它程序
process.h	装人和运行其它程序
process.h	装人和运行其它程序
process.h	装人和运行其它程序
process.h	装人和运行其它程序
process.h	终止程序的执行
process.h	终止程序的执行
math.h	以 e 为底的指数
math.h	求一个浮点数的绝对值
alloc.h	从远堆中分配一块内存
alloc.h	查远堆中还有多少自由内存
alloc.h	从远堆中释放一块内存
alloc.h	从远堆中分配一块内存
alloc.h	调整远堆中一块已分配内存的大小
stdio.h	关闭一个流文件
stdio.h	关闭所有已打开的流文件
stdlib.h	把浮点数转换为一个字符串
stdio.h	辅之一个流以一个文件柄
stdio.h	检测流文件的文件尾
stdio.h	检测流文件上的错误
stdio.h	刷清一个流文件
stdio.h	从流文件中取字符
stdio.h	从标准输入设备取字符
stdio.h	取流文件当前指针
stdio.h	从流文件中取字符串
io.h	取文件大小(字节)
stdio.h	取流文件的文件柄
graphics.h	画并填充一个椭圆
graphics.h	画并填充一个多边形
dir.h,dos.h	搜索一个盘目录
dir.h	接着 findfirst 继续进行搜索

```
void far    floodfill(int x, int y, int border);
double      floor(double x);
int         flushall(void);
double      fmod(double x, double y);
void        fnmerge(char *path,const char *drive,const char *dir,
                    const char *name, const char *ext);
int         fnsplit(const char *path, char *drive, char *dir,char *name,
                    char *ext);
FILE       *fopen(const char *path, const char *mode);
unsigned    FP_OFF(farpointer);
void       _fpreset(void);
int         fprintf(FILE *stream, const char *format, ...);
unsigned    FP_SEG(farpointer);
int         fputc(int c, FILE *stream);
int         fputchar(int c);
int         fputs(const char *s, FILE *stream);
size_t      fread(void *ptr, size_t size, size_t n, FILE *stream);
void        free(void *block);
int         freemem(unsigned segx);
FILE       *freopen(const char *path, const char *mode,FILE *stream);
double      frexp(double x, int *exponent);
int         fscanf(FILE *stream, const char *format, ...);
int         fseek(FILE *stream, long offset, int whence);
int         fsetpos(FILE *stream, const fpos_t *pos);
int         fstat(int handle,struct stat *statbuf);
long        ftell(FILE *stream);
void        ftime(struct timeb *buf);
size_t      fwrite(const void *ptr, size_t size, size_t n, FILE *stream);
char       *gcvt(double value, int ndec, char *buf);
void        geninterrupt(int intr_num);
void far    getarccoords(struct arccoordstype far *arccoords);
void far    getaspectratio(int far *xasp, int far *yasp);
int far     getbkcolor(void);
int         getc(FILE *stream);
int         getcbrk(void);
int         getch(void);
int         getchar(void);
int         getche(void);
int far     getcolor(void);
int         getcurdir(int drive, char *directory);
```

graphics.h	填充一个封闭图形的里面或外面
math.h	求不大于 x 的最大整数
stdio.h	刷清所有流文件
math.h	求 x 对于 y 的模以及余数
dir.h	从部分名建立完整的路径／文件名
dir.h	把完整的路径／文件名分解为部分名
stdio.h	打开一个流文件
dos.h	取一个远指针的偏移量部分
float.h	重新初始化浮点数学包
stdio.h	往流文件上写格式化信息
dos.h	取一个远指针的段地址部分
stdio.h	往流文件上写一个字符
stdio.h	往标准输出设备写一个字符
stdio.h	往流文件上写一个字符串
stdio.h	从一个流文件上读数据
stdlib.h	释放已分配的内存块
dos.h	释放原先分配的 DOS 内存块
stdio.h	以一个新文件替代一个流文件
math.h	把一个 double 数分解为 2 的幂和尾数
stdio.h	从一个流文件读格式化输入信息
stdio.h	重定位一个流文件的指针
stdio.h	定位一个流文件的指针
sys＼stat.h	取一个已打开文件的信息
stdio.h	取当前文件指针
sys＼timeb.h	把当前时间存入结构 timeb 中
stdio.h	往流文件中写
dos.h,stdlib.h	把浮点数变为字符串
dos.h	产生软中断
graphics.h	取上一次调用 arc 函数时的坐标
graphics.h	抽取当前图形方式的比例系数
graphics.h	取当前背景色
stdio.h	从流中取字符
dos.h	取 Ctrl_Break 开关的设置值
conio.h	从键盘取字符,不回显
stdio.h	从标准输入设备取字符
conio.h	从控制台取字符,回显到屏幕上
graphics.h	取当前画笔颜色
dir.h	取指定驱动器上的当前目录

```
char        * getcwd(char * buf, int buflen);
void          getdate(struct date * datep);
struct        palettetype * far getdefaultpalette( void );
void          getdfree(unsigned char drive, struct dfree * dtable);
int           getdisk(void);
char * far    getdrivername( void );
char far    * getdta(void);
char        * getenv(const char * name);
void          getfat(unsigned char drive, struct fatinfo * dtable);
void          getfatd(struct fatinfo * dtable);
void far      getfillpattern(char far * pattern);
void far      getfillsettings(struct fillsettingstype far * fillinfo);
int           getftime(int handle, struct ftime * ftimep);
int far       getgraphmode(void);
void far      getimage(int left, int top, int right, int bottom,void far * bitmap);
void far      getlinesettings(struct linesettingstype far * lineinfo);
int far       getmaxcolor(void);
int far       getmaxmode(void);
int far       getmaxx(void);
int far       getmaxy(void);
char * far    getmodename( int mode_number );
void far      getmoderange(int graphdriver, int far * lomode,int far * himode);
void far      getpalette(struct palettetype far * palette);
int far       getpalettesize( void );
char        * getpass(const char * prompt);
unsigned      getpid(void);
unsigned      far  getpixel(int x, int y);
unsigned      getpsp(void);
char        * gets(char * s);
int           gettext(int left, int top, int right, int bottom, void * destin);
void          gettextinfo(struct text_info * r);
void far      gettextsettings(struct textsettingstype far * texttypeinfo);
void          gettime(struct time * timep);
void          interrupt( * getvect(int interruptno))();
int           getverify(void);
void far      getviewsettings(struct viewporttype far * viewport);
int           getw(FILE * stream);
int far       getx(void);
int far       gety(void);
struct        tm * gmtime(const time_t * timer);
```

dir.h	取当前工作目录
dos.h	取系统日期
graphics.h	取缺省的填充图样
dos.h	取盘自由空间信息
dir.h	取当前驱动器号
graphics.h	取含当前图形驱动程序名的字符串
dos.h	取盘传输区地址
stdlib.h	从环境中取字符串
dos.h	取指定盘的文件分配表信息
dos.h	取文件分配表信息
graphics.h	拷贝用户定义的填充图样到内存
graphics.h	取当前填充图样和填充色
io.h	取文件日期和时间
graphics.h	返回当前图形方式
graphics.h	保存指定区域的位映像到内存
graphics.h	取当前线型信息
graphics.h	返回 setcolor 函数的最大颜色值
graphics.h	返回当前图形所支持的最大方式号
graphics.h	返回最大的屏幕 X 坐标
graphics.h	返回最大的屏幕 Y 坐标
graphics.h	返回含有指定图形方式名的字符串
graphics.h	取指定图形驱动程序的图形方式范围
graphics.h	取当前调色板信息
graphics.h	返回调色板彩色对照表的大小
conio.h	读口令
process.h	取一个程序的进程标志 ID
graphics.h	取指定像点的颜色
dos.h	取程序段前缀
stdio.h	从标准输入设备取字符串
conio.h	从文本方式屏幕拷贝文本到内存
conio.h	取文本方式显示信息
graphics.h	取有关当前图形文本字库的信息
dos.h	取系统时间
dos.h	取中断向量
dos.h	取 DOS 校核标志的状态
graphics.h	取当前视口的信息
stdio.h	从流文件中取整数
graphics.h	返回当前图形位置的 X 坐标
graphics.h	返回当前图形位置的 Y 坐标
time.h	把日期和时间转换为格林尼治标准时间

```
void        gotoxy(int x, int y);
void far    graphdefaults(void);
char * far  grapherrormsg(int errorcode);
void far    _graphfreemem(void far * ptr, unsigned size);
void far    * far _graphgetmem(unsigned size);
int far     graphresult(void);
void        harderr(int ( * handler)());
void        hardresume(int axret);
void        hardretn(int retn);
void        highvideo(void);
double      hypot(double x, double y);
unsigned    far imagesize(int left, int top, int right, int bottom);
void far    initgraph(int far * graphdriver,int far * graphmode,
                      char far * pathtodriver);
int         inport(int portid);
unsigned    char inportb(int portid);
void        insline(void);
int far     installuserdriver( char far * name, int huge ( * detect)(void) );
int far     installuserfont( char far * name );
int         int86(int intno, union REGS * inregs, union REGS * outregs);
int         int86x(int intno,union REGS * inregs,union REGS * outregs,
                   struct SREGS * segregs);
int         intdos(union REGS * inregs, union REGS * outregs);
int         intdosx(union REGS * inregs,union REGS * outregs,struct SREGS * segregs);
void        intr(int intno, struct REGPACK * preg);
int         ioctl(int handle, int func[,void * argdx,int argcx]);
int         isalnum(int c);
int         isalpha(int c);
int         isascii(int c);
int         isatty(int handle);
int         iscntrl(int c);
int         isdigit(int c);
int         isgraph(int c);
int         islower(int c);
int         isprint(int c);
int         ispunct(int c);
int         isspace(int c);
int         isupper(int c);
int         isxdigit(int c);
char        * itoa(int value, char * string, int radix);
```

conio.h	设置文本窗口的光标位置
graphics.h	重新置所有图形参数为它们的缺省值
graphics.h	返回指定错误号的错误信息字符串指针
graphics.h	释放原先 graphgetmem 分配的图形内存
graphics.h	分配图形内存
graphics.h	返回最后一次失败的图形操作的错误码
dos.h	建立一个硬件错误处理程序
dos.h	硬件错误处理程序
dos.h	硬件错误处理程序
conio.h	选择高亮度字符
math.h	计算直角三角形斜边长
graphics.h	求存储一个位映像所需要的字节数
graphics.h	初始化图形系统
dos.h	从一个口地址读一个字
dos.h	从一个口地址读一个字节
conio.h	在文本窗口中插入一个空行
graphics.h	往 BGI 设备驱动程序表中添加新的一项
graphics.h	装入原不在 BGI 系统内的用户字库文件
dos.h	产生 8086 软中断
dos.h	产生 8086 软中断
dos.h	产生 DOS 软中断
dos.h	产生 DOS 软中断
dos.h	产生 8086 软中断
io.h	控制输入／输出设备
ctype.h	是字母和数字吗?
ctype.h	是字母吗?
ctype.h	是标准 ASCII 码(0–127)吗?
ctype.h	文件柄是对照于一个设备吗?
ctype.h	是删除字符或普通的控制字符吗?
ctype.h	是数字吗?
ctype.h	是除空格外的可打印字符(0X21～7E)吗?
ctype.h	是小写字母吗?
ctype.h	是可打印字符(0X21～7E)吗?
ctype.h	是控制字符或空白字符吗?
ctype.h	是空白字符(0X09～0D,0X20)吗?
ctype.h	是大写字母吗?
ctype.h	是十六进制字符吗?
stdlib.h	把整数转换为字符串

```
int         kbhit(void);
void        keep(unsigned char status, unsigned size);
long        labs(long x);
double      ldexp(double x, int exponent);
ldiv_t      ldiv(long numer, long denom);
void       *lfind(const void *key, const void *base,size_t *num, size_t width,
                  int (*fcmp)(/* const void *, const void * */));
void far    line(int x1, int y1, int x2, int y2);
void far    linerel(int dx, int dy);
void far    lineto(int x, int y);
struct tm  *localtime(const time_t *timer);
int         lock(int handle, long offset, long length);
double      log(double x);
double      log10(double x);
void        longjmp(jmp_buf jmpb, int retval);
void        lowvideo(void);
unsigned    long _lrotl(unsigned long val, int count);
unsigned    long _lrotr(unsigned long val, int count);
void       *lsearch(const void *key, void *base,size_t *num, size_t width,
                  int (*fcmp)(/* const void *, const void * */));
long        lseek(int handle, long offset, int fromwhere);
char       *ltoa(long value, char *string, int radix);
void       *malloc(size_t size);
double     _matherr(_mexcep why,char *fun,double *arg1p,double *arg2p,
                  double retval);
int         matherr(struct exception *e);
(type)      max(a,b);
void       *memccpy(void *dest, const void *src, int c, size_t n);
void       *memchr(const void *s, int c, size_t n);
int         memcmp(const void *s1, const void *s2, size_t n);
void       *memcpy(void *dest, const void *src, size_t n);
int         memicmp(const void *s1, const void *s2, size_t n);
void       *memmove(void *dest, const void *src, size_t n);
void       *memset(void *s, int c, size_t n);
(type)      min(a,b);
int         mkdir(const char *path);
void far   *MK_FP(unsigned seg,unsigned ofs);
char       *mktemp(char *template);
double      modf(double x, double *ipart);
void        movedata(unsigned srcseg, unsigned srcoff, unsigned dstseg,
```

conio.h 检查是否有键按下
dos.h 终止程序的执行但驻留内存
math.h 计算长整数的绝对值
math.h 计算 x * 2 的 exp 幂
stdlib.h 两个长整数相除
stdlib.h 线性查找

graphics.h 在指定的两点间画一条线
graphics.h 从当前点到指定点画一条线
graphics.h 从当前点到指定点画一条线
time.h 把日期和时间转换为一个结构
io.h 设置文件共享锁
math.h 计算 x 的自然对数
math.h 计算 x 的常用对数
setjmp.h 执行非局部转移
conio.h 选择低亮度
stdlib.h 左旋无符号长整数指定位
stdlib.h 右旋无符号长整数指定位
stdlib.h 执行线性查找

io.h 移动文件指针
stdlib.h 把长整数转换为字符串
stdlib.h 分配主内存
math.h 浮点错误处理

math.h 用户可修改的数学错误处理程序
stdlib.h 返回两个值中的较大省
math.h 拷贝 n 个字节的内存块,拷完 c 后也停
amth.h 从字符串的头 n 个字节中查找字符 c
math.h 精确比较两个字符串的前 n 个字节
math.h 拷贝 n 个字节的内存块,不许重迭
math.h 比较两个字符串的前 n 个字节,不管大小
math.h 拷贝 n 个字节的内存块,可以重叠
math.h 设置 n 个字节的内存块为字符 c
stdlib.h 返回两个值中的较小者
dir.h 建立一个目录
dos.h 根据段地址和偏移量构造一个远指针
dir.h 创建一个独特的文件名
math.h 把 double 数分成整数和小数两部分
mem.h 拷贝 n 个字节

```
                        unsigned dstoff, size_t n);
void far        moverel(int dx, int dy);
     int        movetext(int left, int top, int right, int bottom,int destleft,
                        int desttop);
void far        moveto(int x, int y);
    void        movmem(void *src, void *dest, unsigned length);
    void        normvideo(void);
    void        nosound(void);
     int        open(const char *path, int access,... /*unsigned mode*/);
     int       _open(const char *path, int oflags);

    void        outport(int portid, int value);
    void        outportb(int portid, unsigned char value);
void far        outtext(char far *textstring);
void far        outtextxy(int x, int y, char far *textstring);
    char       *parsfnm(const char *cmdline, struct fcb *fcb, int opt);
     int        peek(unsigned segment, unsigned offset);
    char        peekb(unsigned segment, unsigned offset);
    void        perror(const char *s);
void far        pieslice(int x, int y, int stangle, int endangle,int radius);
    void        poke(unsigned segment, unsigned offset, int value);
    void        pokeb(unsigned segment, unsigned offset, char value);
  double        poly(double x, int degree, double coeffs []);
  double        pow(double x, double y);
  double        pow10(int p);
     int        printf(const char *format, ...);
     int        putc(int c,FILE *stream);
     int        putch(int c);
     int        putchar(int c);
     int        putenv(const char *name);
void far        putimage(int left, int top, void far *bitmap, int op);
void far        putpixel(int x, int y, int color);
     int        puts(const char *s);
     int        puttext(int left,int top,int right,int bottom,void *source);
     int        putw(int w, FILE *stream);
    void        qsort(void *base,size_t nelem,size_t width,
                     int(*fcmp)(const void *,const void *));
     int        raise(int sig);
     int        rand(void);
     int        randbrd(struct fcb *fcb, int rcnt);
```

graphics.h	把当前位置移动指定距离
conio.h	把屏幕文本从一个区域拷贝到另一区域
graphics.h	移动当前位置到指定点
mem.h	拷贝 length 字节的块,允许重叠
conio.h	选择正常亮度字符
dos.h	关掉 PC 扬声器
fcntl.h,io.h	打开文件,进行读/写
fcntl.h,io.h, sys\stat.h	打开文件,进行读/写
dos.h	往输出口写一个字
dos.h	往输出口写一个字节
graphics.h	在视口显示一个字符串
graphics.h	在指定位置显示一个字符串
dos.h	分析命令行中的文件名
dos.h	返回内存指定单元中的一个字
dos.h	返回内存指定单元中的一个字节
stdio.h	打印系统错误信息
graphics.h	画并填充一个饼图
dos.h	往内存指定单元存一个整数
dos.h	往内存指定单元存一个字节
math.h	画一条多项式曲线
math.h	计算 x 的 y 次幂
math.h	计算 10 的 p 次幂
stdio.h	往标准输出设备写格式化信息
stdio.h	往流文件输出一个字符
conio.h	往屏幕上输出字符
stdio.h	往标准输出设备输出字符
stdlib.h	往环境中添加字符串
graphics.h	往屏幕上输出位映像
graphics.h	在指定位置画一个点
stdio.h	往标准输出流输出一个字符串
conio.h	从内存拷贝文本到文本方式屏幕
stdio.h	往文件流中写一个整数
stdlib.h	用 quicksort 算法排序
signal.h	发送一个软信号到当前正在执行的程序
stdlib.h	随机数发生器
dos.h	随机方式读指定块数

```
int       randbwr(struct fcb  * fcb, int rcnt);
int       random(int num);
void      randomize(void);
int       read(int handle, void  * buf, unsigned len);
int      _read(int handle, void  * buf, unsigned len);
void     * realloc(void  * block, size_t size);
void far  rectangle(int left, int top, int right, int bottom);
int far   registerfarbgidriver(void far  * driver);
int far   registerfarbgifont(void far  * font);
int       remove(const char  * filename);
int       rename(const char  * oldname, const char  * newname);
void far  restorecrtmode(void);
void      rewind(FILE  * stream);
int       rmdir(const char  * path);
unsigned _rotl(unsigned value, int count);
unsigned _rotr(unsigned value, int count);
void     * sbrk(int incr);
int       scanf(const char  * format, ...);
char     * searchpath(const char  * file);
void far  sector(int x,int y,int stangle,int endangle,int xradius,int yradius);
void      segread(struct SREGS  * segp);
void far  setactivepage(int page);
void far  setallpalette(struct palettetype far  * palette);
void far  setaspectratio( int xasp, int yasp );
void far  setbkcolor(int color);
int       setblock(unsigned segx, unsigned newsize);
void      setbuf(FILE  * stream, char  * buf);
int       setcbrk(int cbrkvalue);
void far  setcolor(int color);
void      setdate(struct date  * datep);
int       setdisk(int drive);
void      setdta(char far  * dta);
void far  setfillpattern(char far  * upattern, int color);
void far  setfillstyle(int pattern, int color);
int       setftime(int handle, struct ftime  * ftimep);
unsigned  far setgraphbufsize(unsigned bufsize);
void far  setgraphmode(int mode);
int       setjmp(jmp_buf jmpb);
void far  setlinestyle(int linestyle,unsigned upattern,int thickness);
void      setmem(void  * dest, unsigned length, char value);
```

dos.h	随机方式写指定块数
stdlib.h	随机数发生器
stdlib.h,time.h	初始化随机数发生器
io.h	读文件
io.h	读文件
stdlib.h	重新分配主内存
graphics.h	画一个矩形
graphics.h	登记用户装人或已连人的图形驱动程序
graphics.h	登记一个已连人的向量字库
stdio.h	删除一个文件
stdio.h	文件换名
graphics.h	恢复屏幕到进入图形以前的显示方式
stdio.h	重定位文件指针到流文件的开始
dir.h	删除一个 DOS 文件目录
stdlib.h	左旋一个无符号整数值
stdlib.h	右旋一个无符号整数值
alloc.h	改变数据段空间分配
stdio.h	从标准输入设备流读格式化信息
dir.h	搜索一个文件的路径
graphics.h	画并填充一个椭圆饼形图
dos.h	读段寄存器
graphics.h	设置当前编辑页
graphics.h	改变全部调色板颜色
graphics.h	改变缺省的坐标轴比例系数
graphics.h	设置背景色
dos.h	修改原已分配块的大小
stdio.h	为一个流文件指定缓冲区
dos.h	设置 Ctrl__Break 开关的状态
graphics.h	设置当前画笔的颜色
dos.h	设置日期
dir.h	设置当前盘号
dos.h	设置盘传输区地址
graphics.h	选择用户定义的填充图样
graphics.h	设置填充图样和颜色
io.h	设置文件日期和时间
graphics.h	改变内部图形缓冲区的大小
graphics.h	置系统为图形方式,清屏
setjmp.h	设置非局部转移的目标地址
graphics.h	设置当前线型
mem.h	初始化一片内存为某一值

```
int       setmode(int handle, int amode);
void far  setpalette(int colornum, int color);
void far  setrgbpalette(int colornum,int red, int green, int blue);
void far  settextjustify(int horiz, int vert);
void far  settextstyle(int font, int direction, int charsize);
void      settime(struct time *timep);
void far  setusercharsize(int multx, int divx,int multy, int divy);
int       setvbuf(FILE *stream, char *buf, int type, size_t size);
void      setvect(int interruptno, void interrupt (*isr) ());
void      setverify(int value);
void far  setviewport(int left,int top,int right,int bottom,int clip);
void far  setvisualpage(int page);
void far  setwritemode( int mode );
void (*   signal(int sig, void (*func)(int sig[,int subcode])))(int);
double    sin(double x);
double    sinh(double x);
void      sleep(unsigned seconds);
int       sopen(char *path,int access,int shflag,int mode);

void      sound(unsigned frequency);
int       spawnl(int mode, char *path, char *arg0, ...);
int       spawnle(int mode, char *path, char *arg0, ...);
int       spawnlp(int mode, char *path, char *arg0, ...);
int       spawnlpe(int mode, char *path, char *arg0, ...);
int       spawnv(int mode, char *path, char *argv[]);
int       spawnve(int mode,char *path,char *argv[],char **env);
int       spawnvp(int mode, char *path, char *argv[]);
int       spawnvpe(int mode,char *path,char *argv[],char **env);
int       sprintf(char *buffer, const char *format, ...);
double    sqrt(double x);
void      srand(unsigned seed);
int       sscanf(const char *buffer, const char *format, ...);
int       stat(char *path,struct stat *statbuf);
unsigned  int _status87(void);
int       stime(time_t *tp);
char      *stpcpy(char *dest, const char *src);
char      *strcat(char *dest, const char *src);
char      *strchr(const char *s, int c);
int       strcmp(const char *s1, const char *s2);
int       strcmpi(const char *s1,const char *s2);
```

io.h	设置打开文件的方式
graphics.h	改变一个调色板颜色
graphics.h	允许用户为 IBM8514 定义颜色
graphics.h	设置图形方式的文本对齐方式
graphics.h	设置图形方式下的文本特点
dos.h	设置系统时间
graphics.h	允许用户改变向量字库字符宽度和高度
stdio.h	给流文件指定缓冲区
dos.h	设置中断向量值
dos.h	设置 DOS 校核标志的状态
graphics.h	为图形输出设置当前视口
graphics.h	设置当前图形显示页
graphics.h	设置图形方式下画线的写方式
signal.h	说明信号(事件)处理行为
math.h	计算正弦值
math.h	计算 hyperbolic 正弦值
dos.h	暂停执行(s)
fcntl.h,share.h, sys\stat.h,io.h	打开一个共享文件
dos.h	以指定频率发声
process.h,stdio.h	建立和运行子进程
process.h,stdio.h	建立和运行子进程
process.h,stdio.h	建立和运行子进程
process.h,stdio.h	建立和运行子进程
process.h,stdio.h	建立和运行子进程
process.h,stdio.h	建立和运行子进程
process.h,stdio.h	建立和运行子进程
process.h,stdio.h	建立和运行子进程
stdio.h	写格式化信息到一个字符串
math.h	计算平方根
stdlib.h	初始化随机数发生器
stdio.h	从缓冲区中读格式化输入
sys\stat.h	取文件信息
float.h	取浮点状态
time.h	设置系统日期和时间
string.h	拷贝字符串到另一个字符串
string.h	追加字符串到另一个字符串
string.h	从字符串 S 中找字符 C 的第 1 次出现
string.h	比较两个字符串
string.h	比较两个字符串,不管大小写

```
char      * strcpy(char * dest, const char * src);
size_t      strcspn(const char * s1, const char * s2);
char      * strdup(const char * s);
char      * _strerror(const char * s);
char      * strerror(int errnum);
int         stricmp(const char * s1, const char * s2);
size_t      strlen(const char * s);
char      * strlwr(char * s);
char      * strncat(char * dest, const char * src, size_t maxlen);
int         strncmp(const char * s1,const char * s2,size_t maxlen);
int         strncmpi(const char * s1,const char * s2,size_t n);
char      * strncpy(char * dest, const char * src, size_t maxlen);
int         strnicmp(const char * s1,const char * s2,size_t maxlen);
char      * strnset(char * s, int ch, size_t n);
char      * strpbrk(const char * s1, const char * s2);
char      * strrchr(const char * s, int c);
char      * strrev(char * s);
char      * strset(char * s, int ch);
size_t      strspn(const char * s1, const char * s2);
char      * strstr(const char * s1, const char * s2);
double      strtod(const char * s, char * * endptr);
char      * strtok(char * s1, const char * s2);
long        strtol(const char * s, char * * endptr, int radix);
unsigned    long strtoul(const char * s,char * * endptr,int radix);
char      * strupr(char * s);
void        swab(char * from, char * to, int nbytes);
int         system(const char * command);
double      tan(double x);
double      tanh(double x);
long        tell(int handle);
void        textattr(int newattr);
void        textbackground(int newcolor);
void        textcolor(int newcolor);
int far     textheight(char far * textstring);
void        textmode(int newmode);
int far     textwidth(char far * textstring);
time_t      time(time_t * timer);
FILE      * tmpfile(void);
char      * tmpnam(char * s);
int         toascii(int c);
```

string.h	拷贝一个字符串到另一个字符串
string.h	在串 s1 中寻找完全不包含 s2 中字符的段
string.h	拷贝字符串到一个新建立的位置
string.h	返回错误信息字符串的指针
string.h	返回错误信息字符串的指针
string.h	比较两个字符串,不管大小写
string.h	计算字符串的长度
string.h	把字符串中的大写字母转换为小写
string.h	追加字符串的一部分到另一个字符串
string.h	比较两个字符串的前面一部分
string.h	比较两个字符串的前一部分,不管大小写
string.h	拷贝字符串的部分字节到另一个字符串
string.h	比较两个字符串的前一部分,不管大小写
string.h	拷贝字符 ch 到字符串 s 的前 n 个字节
string.h	搜索字符串 s1,直到 s2 中的任何字符出现
string.h	反向搜索字符串 s1,直到 c 的第一次出现
string.h	反转一个字符串
string.h	初始化字符串 s 为字符 ch
string.h	搜索串 s1 中完全由 s2 中字符构成的段
string.h	在字符串 s1 中找 s2 的出现
stdlib.h	把字符串转换为 double 值
string.h	搜索串 S1,它由 S2 中的字符隔为各 token
stdlib.h	把字符串转换为长整数
stdlib.h	把字符串转换为无符号长整数
string.h	把字符串中的小写字母变为大写字母
stdlib.h	字节交换
stdlib.h	执行一条 DOS 命令
math.h	计算正切值
math.h	计算 hyperbolic 正切值
io.h	取文件指针的当前位置
conio.h	设置文本属性
conio.h	选择新的文本背景颜色
conio.h	选择新的文本方式字符颜色
graphics.h	返回字符串的高度(以像点为单位)
conio.h	置屏幕为文本方式
graphics.h	返回字符串的宽度(以像点为单位)
time.h	取时间
stdio.h	以二进制方式建立和打开一个临时文件
stdio.h	建立一个独特的文件名
ctype.h	把整数转换为 ASCII 格式

```
int    _tolower(int ch);
int     tolower(int ch);
int    _toupper(int ch);
int     toupper(int ch);
void    tzset(void);
char   *ultoa(unsigned long value, char *string, int radix);
int     ungetc(int c, FILE *stream);
int     ungetch(int ch);
void    unixtodos(long time, struct date *d, struct time *t);
int     unlink(const char *path);
int     unlock(int handle, long offset, long length);
void    va_start(va_list ap,lastfix);
type    va_arg(va_list ap,type);
void    va_end(va_list ap);
int     vfprintf(FILE *stream, const char *format, va_list arglist);
int     vfscanf(FILE *stream, const char *format, va_list arglist);
int     vprintf(const char *format, va_list arglist);
int     vscanf(const char *format, va_list arglist);
int     vsprintf(char *buffer, const char *format, va_list arglist);
int     vsscanf(const char *buffer,const char *format,va_list arglist);
int     wherex(void);
int     wherey(void);
void    window(int left, int top, int right, int bottom);
int     write(int handle, void *buf, unsigned len);
int    _write(int handle, void *buf, unsigned len);
```

ctype.h	把字母转换为小写字母,要求原为大写
ctype.h	把字母转换为小写字母
ctype.h	把字母转换为大写字母,要求原为小写
ctype.h	把字母转换为大写字母
time.h	根据环境变量 TZ 设置几个全局变量的值
stdlib.h	把无符号长整数转换为字符串
stdio.h	把字符放回到输入流中
conio.h	把字符放回到键盘缓冲区中
dos.h	把日期和时间由 unix 格式变为 DOS 格式
dos.h	删除一个文件
io.h	释放文件共享锁
stdarg.h	初始化可变参数指针
stdarg.h	指向下一个参数
stdarg.h	帮助执行正常的返回
stdio.h	往流文件写格式化输出
stdio.h	从流文件读格式化输入
stdio.h	往标准输出设备写格式化输出
stdio.h	从标准输入设备读格式化输入
stdio.h	往缓冲区写格式化输出
stdio.h	从缓冲区读格式化输入
stdio.h	取光标在窗口内的水平位置
stdio.h	取光标在窗口内的垂直位置
stdio.h	定义一个活动的文本方式窗口
io.h	写文件
io.h	写文件

附录4

函　数　格　式

```
ED__ACTION   edbuffer(ED__BUFFER  *pbuffer,const ED__KEY  *pcommand);
int          edchgkey(const KEY__SEQUENCE  *pkey__seq,
                   const ED__ACTION__LIST  *pedits);
int          edfield(const char  *pinitstr,char  *pretstr,int retsize,
                  const ED__CONTROL  *pctrl,KEY__SEQUENCE  *pfinal);
int          edinitky(void);
int          edremkey(const KEY__SEQUENCE  *pkey__seq);
ED__KEY     *edretkey(const KEY__SEQUENCE  *pkey__seq);
void         edzapkey(void);
int          fldolock(int handle,unsigned long offset,unsigned long length,
                  int seconds, int option);
int          flflush (int handle);
void far    *flgetdta (void);
int          fllock (int handle,int option,unsigned long offset,
                  unsigned long length);
int          flnorm (const char  *pfile,char  *pnorm, int  *plast);
int          flprompt (const char  *pprompt,char  *presp,int resp__size);
void         flputdta (void far  *pdta);
int          flremvol (int drive);
int          flretvol(int drive, char  *pvol,unsigned  *pfdate,unsigned  *pftime);
int          flsetvol(int drive,const char  *pvol);
int          hlclose(void);
WN__EVENT   *hldisp (const char  *ppath,const char  *pid__string,
                  unsigned long offset,WN__EVENT  *pfinal,int option);
int          hllookup(const char  *ppath,const char  *pid__string,
                   unsigned long offset,HL__WINDOW  *phelp__win,char  * *pptext,
                   int  *plength,int option);
int          hlopen(const char  *ppath);
WN__EVENT   *hlread(BWINDOW  *pwin,const HL__WINDOW  *phelp__win,const char  *ptext,
                int length,WN__EVENT  *pfinal,int option);
int          iscall(void far  *pisr,ALLREG  *preg);
unsigned     iscurprc(int option,unsigned newproc);
void         isgetvec(int intype);
int          isinstal(int intype,void ( *pfunc)(ALLREG  *,ISRCTRL  *,ISRMSG  *),
                  const char  *pident,ISRCTRL  *pisrblk,char  *pstack,
                  int stksize,int stknum);
```

Turbo C Tools 6.0 函数简表

所需头文件	简 要 说 明
bedit.h	对编辑缓冲区执行编辑动作
bedit.h	加入、修改 edfield 和 wnfield 所认可的按键
bedit.h, strings.h	对屏幕上的一个域进行编辑
bedit.h	安装 edfield 和 wnfield 所接受的缺省按键
bedit.h	删去 edfield 和 wnfield 所接受的一个按键
bedit.h	报告一个按键的编辑动作
bedit.h	删去 edfield 和 wnfield 所接受的按键队列
bfiles.h	对已打开文件的一个文件段上锁或解锁
bfiles.h	迫使挂起的文件输出写到磁盘上
bfiles.h	返回磁盘传送地址
bfiles.h	对已打开文件的一个文件段上锁或解锁
bfiles.h	验证一个文件名并把它转化为标准形式
bfiles.h	从标准输入返回一行,可以使用提示信息
bfiles.h	设置磁盘传送地址(DAT)
bfiles.h	从给定的磁盘上删除卷标(如果有的话)
bfiles.h	报告给定磁盘上的卷标
bfiles.h	建立或修改给定磁盘上的卷标
bhelp.h	释放二叉帮助文件的已有索引
bhelp.h	从帮助文件读帮助信息并显示在屏幕上
bhelp.h	从二叉帮助文件中读取一段帮助信息
bhelp.h	对一个二叉帮助文件建立索引
,bhelp.h	在视口中显示帮助信息,供用户浏览
bintrupt.h	对软件中断调用中断服务例程进行模拟
bintrupt.h	返回或设置当前执行的进程
bintrupt.h	返回一个中断向量
bintrupt.h	安装一个中断服务例程(ISR)

```
void          isprep(void ( * pfunc)(ALLREG  * ,ISRCTRL  * ,ISRMSG  * ),
                   const char  * pident,ISRCTRL  * pisrblk,char  * pstack,
                   int stksize,int stknum);
void          isputvec(int intype,void far  * ptr);
int           isremove(unsigned psp__seg);
int          * isreserv(size__t reserve,size__t blksize,char  *  * ppblock);
void          isresext(int excode);
ISRCTRL far  * issense(void far  * ptr,const char  * pident);
IV__CTRL far * ivctrl(void);
int           ivdetect(const IV__VECTORS far  * pvectors,const char far  * pident,
                    IV__CTRL far  *  * ppctrl,unsigned far  * pfound);
int           ivdisabl(IV__CTRL far  * pctrl);
int           ivinstal(void( * pfunc)(IV__EVENT  * ),const char  * pident,char  * pstack
                    int stksize,IV__KEY  * pkeytable,int numkeys,
                               IV__TIME  * ptimetable,int numtimes,int option);
IV__CTRL far * ivsense(const IV__VECTORS far  * pvectors,const char far  * pident);
int           ivvecs(int option,IV__VECTORS far  * pvectors);
int           kbequip (void);
int           kbextend (int selection);
int           kbflush (void);
int           kbgetkey (int  * pkey);
void          kbkcflsh (PKEY__CONTROL pfunc,void  * pdata);
int           kbplace (int at__head, char value, char key);
int           kbpoll (PKEY__CONTROL pfunc,void  * pdata,KEY__SEQUENCE  * pkey__seq,
                   int option);
char          kbquery (char  * presp,int resp__size,int  * pkey,int  * pscrolled);
int           kbqueue (int  * ptotal);
int           kbready (char  * pch,int  * pkey);
int           kbscanof(int ch);
void          kbset (const KEYSTATUS  * pkeybd);
int           kbstatus(KEYSTATUS  * pkeybd);
char         * kbstuff (int at__head,char  * pstring);
KEY__SEQUENCE  kbwait(PKEY__CONTROL pfunc,void  * pdata);
int           mmctrl(unsigned memblk,MEMCTRL  * pctrlblk,unsigned  * pnextblk);
unsigned      mmfirst(void);
unsigned      mmsize(void);
BMENU        * mncreate(int height,int width,int textattr,int hilattr,
                   int proattr,int longattr);
BMENU        * mndsplay(BMENU  * pmenu,const WHERE  * pwhere,const BORDER  * pbord);
int           mndstroy (BMENU  * pmenu);
```

bintrupt.h	预备一个 ISR 控制块
bintrupt.h,dos.h	设置一个中断向量
bintrupt.h	摘除一个驻留程序
bintrupt.h	保留 ISR 所需的动态内存
bintrupt.h,dos.h	中止一个程序但保持驻留
bintrupt.h	探测一个已安装的中断服务例程(ISR)
binterv.h	报告本程序中插入控制块的地址
binterv.h	探测已安装的插入函数，即使它被部分遮蔽
binterv.h	使一个插入函数失效
,binterv.h	安装一个插入函数
binterv.h	探测一个已安装的插入函数是否可摘除
binterv.h	设置或返回插入过滤程序使用的中断向量
bkeybrd.h	探测键盘环境
bkeybrd.h	选用扩展的或一般的 BIOS 键盘服务
bkeybrd.h	废弃所有在键盘缓冲区等待的按键
bkeybrd.h	等待读入下一个按键
bkeybrd.h	通过键控制函数废弃所有等待的按键
bkeybrd.h	在键盘缓冲区中放置一个按键
bkeybrd.h	通过一个键控制函数查看下一个等待按键
bkeybrd.h	从标准 IBM 控制台读取用户的响应
bkeybrd.h	报告键盘缓冲区总容量及剩余容量
bkeybrd.h	检查下一个等待的按键
bkeybrd.h	返回一个字符的键码
bkeybrd.h	设置移位键的当前状态
bkeybrd.h	报告移位键的当前状态
bkeybrd.h	强行将一字符串送入 BIOS 超前键入缓冲区
bkeybrd.h	等待并通过键控制函数读取下一个按键
bkeybrd.h	读取 DOS 内存控制块
bmem.h	报告第一个内存块的地址
bmem.h	报告一个程序的尺寸
bmem.h	建立一个包含选单信息的选单结构和窗口
bmenu.h	在同尺寸视口中显示一个选单
bmenu.h	从屏幕上取消一个选单,废弃其数据结构

```
BMENU   * mnhilite(BMENU  * pmenu,int row,int col,int option);
BMENU   * mnitem (BMENU  * pmenu,int row,int col,int option,
                 const char  * pstring);
BMENU   * mnitmkey (BMENU  * pmenu,int row,int col,int option,
                 const char  * pstring,const char  * pkeys,int action);
BMENU   * mnkey (BMENU  * pmenu,int row,int col,int ch,int key,int act__move,
                 int howchange);
BMENU   * mnlitem(BMENU  * pmenu,int row,int col,int option,
                 const char  * pstring,int lrow,int lcol,const char  * plstring);
BMENU   * mnlitkey(BMENU  * pmenu,int row,int col,int option,
                 const char  * pstring,int lrow,int lcol,
                 const char  * plstring,const char  * pkeys,int action);
   int    mnlread(BMENU  * pmenu,int srow,int scol,int prow,int pcol,
                 int pch, int pscan,int option);
BMENU   * mnmouse(BMENU  * pmenu,unsigned long event,unsigned long ignore,
                 int action,int option);
BMENU   * mnmstyle(BMENU  * pmenu,int style,unsigned button);
   int    mnread (BMENU  * pmenu,int srow,int scol,int  * prow,int  * pcol,
                 int  * pch,int  * pkey,int option);
BMENU   * mnvdisp(BMENU  * pmenu,const WHERE  * pwhere,int view__ht,int view__wid
                 int org__row,int org__col,const BORDER  * pbord);
 int far  moavoid(unsigned u__row,unsigned u__col,unsigned l__row,
                 unsigned l__col);
   int    mobutton(int event,unsigned  * pbuttons,unsigned  * pcount,
                 unsigned  * pvert,unsigned  * phoriz);
 int far  mocheck(unsigned long event,unsigned long ignore,int option,
                 unsigned long  * pfound,unsigned  * pvert,unsigned  * phoriz);
   int    mocurmov(unsigned vert,unsigned horiz);
   int    moequip(void);
   int    mogate(const DOSREG  * pinregs,DOSREG  * poutregs);
   int    mogetmov(int  * pvert,int  * phoriz);
   int    mograph(const unsigned  * pmasks,int hot__row,int hot__col);
   int    mohandlr(PMOHANDLER pfunc,unsigned call__mask,char  * pstack,
                 int stksize,int option);
   int    mohard(int high,int low);
 int far  mohide(int option);
   int    mojump(unsigned speed);
   int    molitpen(int option);
 int far  mopreclk(int option);
   int    morange(int option,unsigned low,unsigned high);
```

bmenu.h	移动或取消选单亮条及选项说明
bmenu.h	加入、修改或删除一个选项
bmenu.h	向选单加入一个选项,为它分配选择字符
bmenu.h	加入、修改或取消一个选单的键分配
bmenu.h	加入、修改或取消一个 Lotus 形式的选项
bmenu.h	加入一个 Lotus 式的选项,分配选择字符
bmenu.h	通过 Lotus 形式的选单读入一个用户响应
bmenu.h,bmouse.h	加入、修改、删除一个选单认可的鼠标事件
bmenu.h,bmouse.h	设立一个标准选单鼠标器风格
bmenu.h	读取来自选单的用户响应
bmenu.h	在视口中显示一个虚拟选单
bmouse.h	在指定区域中隐蔽鼠标
bmouse.h	报告鼠标器按钮的按下/释放历史
bmouse.h	检查最近发生的鼠标器事件
bmouse.h	移动鼠标
bmouse.h	感知鼠标器驱动程序的存在
bmouse.h	鼠标器驱动程序的入口
bmouse.h	报告自上次查询以来物理鼠标器的移动
bmouse.h	设置鼠标器图形方式光标
bmouse.h	安装或摘除中断处理程序
bmouse.h	设置鼠标器硬件字符方式光标
bmouse.h	隐藏或显示鼠标
bmouse.h	设置鼠标器加速阈值
bmouse.h	使鼠标器光笔模拟有效或失效
bmouse.h	安装或摘除 mocheck 所有的内部例程
bmouse.h	设置鼠标器范围界限

```
int         moreset(void);
int         mosoft(unsigned screen__mask,unsigned cursor__mask);
int         mospeed(unsigned vert,unsigned horiz);
int         mostat(unsigned *pbuttons,unsigned *pvert,unsigned *phoriz);
int         prcancel (char *pfile);
int         prchar(int port,char bval);
const char  *prerror (int ercode);
int         prgetq (int n,int *psize,char *pfile);
int         prinit(int port);
int         prinstld (void);
int         prspool (const char *pfile);
int         prstatus(int port);
int         scapage(int page);
int         scattrib(int fore,int back,char ch,unsigned cnt);
int         scblink(int option);
int         scborder(unsigned color);
int         scbox(int u__row,int u__col,int l__row,int l__col,int boxtype,
                 char boxchar,int attrib);
int         scchgdev(int device);
int         scclrmsg(int row,int col,int len);
int         sccurset(int row,int col);
int         sccurst(int *prow,int *pcol,int *phigh,int *plow);
char        scequip(void);
void        scgetvid(ADAP__STATE *pstate);
int         scmode(int *pmode,int *pcolumns,int *papage);
int         scmode4(int palett,int backgrd);
int         scnewdev(int mode,int num__rows);
int         scpage(int page);
int         scpages(void);
int         scpal1(unsigned reg,unsigned value);
int         scpalett(const char *ptable);
void        scpclr(void);
int         scpgcur(int off,int high,int low,int adjust);
char        scread(int *pfore,int *pback);
int         screstpg(const PAGE__STATE *ppage__state);
int         scrows(void);
int         scsavepg(PAGE__STATE *ppage__state);
int         scsetvid(const ADAP__STATE *pstate);
void        scttywin(int u__row,int u__col,int l__row,int l__col,char ch,int fore,
                 int back,int scr__fore,int scr__back);
```

bmouse.h	重置鼠标器驱动程序
bmouse.h	设置鼠标器软件字符方式光标
bmouse.h	设置鼠标器灵敏度
bmouse.h	报告鼠标器位置和按钮状态
bprint.h	删除假脱机打印队列中一个或全部文件
bprint.h	通过 BIOS 向打印机发送一个字符
bprint.h	返回相应打印函数错误代码的字符串指针
bprint.h	报告假脱机打印队列中的一个文件名
bprint.h	通过 BIOS 初给化一个打印口
bprint.h	检查驻留假脱机打印系统 PRINT 是否安装
bprint.h	将一个文件提交给假脱机打印系统
bprint.h	通过 BIOS 报告打印机的状态
bscreens.h	显示(激活一个显示页)
bscreens.h	用指定的显示属性显示一个字符的拷贝
bscreens.h	选择前景闪烁或背景亮度
bscreens.h	设置当前显示屏幕的边界颜色
bscreens.h	用图形字符在屏幕上画一个方框
bscreens.h	切换至彩色或单色显示
bscreens.h	清除屏幕上的消息
bscreens.h	移动当前显示页上的光标
bscreens.h	返回当前显示页上的光标位置和尺寸
bscreens.h	探测显示硬件环境
bscreens.h	记录整个显示状态
bscreens.h	返回屏幕的显示方式
bscreens.h	设置方式 4 色板和背景颜色
bscreens.h	选择和重置显示设备及设置字符行数
bscreens.h	设置当前显示页
bscreens.h	返回显示页的数目
bscreens.h	定义一个 EGA、VGA 或 MCGA 调色板颜色
bscreens.h	定义 EGA、VGA 或 MCGA 颜色的整个调色板
bscreens.h	清除当前显示页
bscreens.h	设置当前页的光标尺寸
bscreens.h	从屏幕读取一个显示字符及其属性
bscreens.h	恢复一个显示页
bscreens.h	返回屏幕的字符行数
bscreens.h	保存一个显示页
bscreens.h	恢复整个显示状态
bscreens.h	以 TTY 方式向矩形区域写入一个字符

```
int             scttywrt(char ch,int fore);
void            scwrap(int u__row,int u__col,int l__row,int l__col,int num__spaces.
                       const char *pbuffer.int fore,int back,int option);
int             scwrite(char ch,unsigned cnt);
char           *stpcvt(char *psource,int conv);
char           *stpexpan(char *ptarget,char *psource,int incr,int tarsize);
char           *stpjust(char *ptarget,const char *psource,char fill,
                       int fldsize, int code);
char           *stptabfy(char *psource,int incr);
char           *stpxlate(char *psource,const char *ptable,const char *ptrans);
int             stschind(char check,const char *psearch);
int             utansi(int option);
int             utchknil(void);
unsigned char   far  *utcrit (void);
int             utctlbrk (int set,int new__state);
int             utdosrdy(void);
int             utgetclk (long *pcount);
int             utintflg (int flag);
char            utmodel(void);
void            utmovmem(const char far *psource,char far *ptarget,
                       unsigned int length);
type__name far *utnorm(void far *ptr,type__name);
unsigned int    utnulchk (void);
unsigned int    utoff(void far *ptr);
unsigned char   utpeekb(ptr);
void            utpeekn(const void far *psource,void far *ptarget.
                       unsigned int length);
unsigned int    utpeekw(const void far *ptr);
unsigned long   utplong(void *ptr);
void            utpokeb(void far *ptr,unsigned char byteval);
void            utpoken(const void *psource.void far *ptarget,unsigned int length
void            utpokew(void far *ptr,unsigned char wordval);
int             utsafcpy(const void far *psource,void far *ptarget,
                       unsigned int length);
unsigned int    utseg(void far *ptr);
unsigned        utsleep (unsigned period);
void            utspkr (unsigned freg);
int             utsqzscn(const char far *psource,char *ptarget,
                       int srcsize, int tarsize);
unsigned long   uttim2tk(int hours,int mins,int secs,int hunds);
```

bscreens.h	以 TTY 方式向屏幕写一个字符
bscreens.h	以 TTY 方式向矩形中写入字符串,整字换行
bscreens.h	在屏幕上显示一个字符的多个拷贝
bstrings.h	常用字符串转换
bscreens.h	将 tab 字符转换为空格
bscreens.h	在域中将一个字符串左右对齐或居中
bscreens.h	把空格转换为 tab 字符
bscreens.h	用翻译表翻译一个字符串
bscreens.h	查找字符串中的一个字符,返回它的位置
butil.h	探测、关闭或重新开放 ANSI.SYS
butil.h,stdio.h	报告无效指针赋值,使程序夭折
butil.h	取得 DOS 临界段标识的地址
butil.h	设置或返回 Ctrl__Break 检查的状态
butil.h	报告 DOS 服务是否可用
butil.h	报告自午夜以来 BIOS 计时脉冲的个数
butil.h	开放或关闭硬件中断
butil.h	报告 IBM 型号和子型号及 BIOS 版本
butil.h	从内存或向内存任何位置拷贝数据
butil.h	使一个指针具有最小的偏移值
butil.h	探测无效的指针赋值
butil.h	返回一个地址的偏移部分
butil.h	从任意地址读取一个字符
butil.h	从任意地址读取多个字节的数据
butil.h	从任意地址读取一个字
butil.h	将指针转换为指向 20 位物理地址的指针
butil.h	在任意地址存放一个字节的数据
butil.h	在任意地址存放多个字节的数据
butil.h	在任意位置写入一个字的数据
butil.h	以确保不跨越段界的方式拷贝数据
butil.h	返回任意地址的段部分
butil.h	暂停处理，直至经过几个计时脉冲
butil.h	打开或关闭扬声器
butil.h	压缩一个屏幕图像
butil.h	将时间转换为计时脉冲计数

```
void        uttk2tim(unsigned long ticks,int *phrs,int *pmins,int *psecs,
                     int *phunds);
type_name far *uttofar(unsigned int seg,unsigned int off,type_name);
void far    *uttofaru(unsigned int seg,unsigned int off);
int         utunsqz(const char *psource,char far *ptarget,
                     int srcsize, int tarsize);
int         viatrect(int u_row,int u_col,int l_row,int l_col,
                     int fore, int back);
int         vidspmsg(int row,int col,int fore,int back,const char *pmsg);
int         vihoriz(int num_cols,int attrib,int u_row,int u_col,int l_row,
                     int l_col,int dir);
char far    *viptr(int row,int col);
int         virdrect(int u_row,int u_col,int l_row,int l_col,char *pbuffer,
                     int option);
int         virdsect(int u_row,int u_col,int l_row,int l_col,char *pbuffer,
                     unsigned gap,int option);
int         viscroll(int num_rows,int attrib,int u_row,int u_col,int l_row,
                     int l_col,int dir);
int         viwrrect(int u_row,int u_col,int l_row,int l_col,
                     const char *pbuffer,int fore,int back,int option);
int         viwrsect(int u_row,int u_col,int l_row,int l_col,
                     const char *pbuffer,unsigned gap,int fore,int back,
                     int option);
BWINDOW     *wnatrblk (BWINDOW *pwin,int r1,int c1,int r2,int c2,int fore,
                     int back,int option);
BWINDOW     *wnatrstr (BWINDOW *pwin,int row,int col,int count,int fore,
                     int back,int option);
BWINDOW     *wnattr(int fore,int back);
int         wnchgevn (WN_EVENT_LIST **ppfirst,const WN_EVENT *pevent,
                     WN_ACTION action);
BWINDOW     *wncreate(int height,int width,int attr);
BWINDOW     *wncurmov (int row,int col);
BWINDOW     *wncurpos (int *prow,int *pcol);
BWINDOW     *wncursor (BWINDOW *pwin);
BWINDOW     *wndsplay(BWINDOW *pwin,const WHERE *pwhere,const BORDER *pbord);
int         wndstroy (BWINDOW *pwin);
int         wnerror(int ercode);
int         wnfield(BWINDOW *pwin,char *pinitstr,char *pretstr,int size,
                     ED_CONTROL *pctrl,KEY_SEQUENCE *pfinal);
BWINDOW     *wngetopt (BWINDOW *pwin,int item,int *pvalue);
```

butil.h	将计时脉冲计数转换为 24 小时制时间
butil.h	用段和偏移构造一个双字指针
butil.h	用一个段和偏移构造一个双字指针
butil.h	还原一个压缩屏幕图像
bvideo.h	改变屏幕上一个矩形的属性
bvideo.h	显示一条消息
bvideo.h	在当前显示页上水平滚动正文列
bvideo.h	将屏幕位置转换成内存地址
bvideo.h	读取屏幕上一个矩形区域的内容
bvideo.h	将屏幕一个域读人内存中更大的矩形域
bvideo.h	垂直滚动当前显示页上的正文行
bvideo.h	向当前显示页上一个矩形区域写入数据
bvideo.h	显示矩形缓冲区中的一个矩形段
bwindow.h	修改窗口中一个矩形块的属性
bwindow.h	改变窗口中一片连续位置的属性
bwindow.h	改变当前窗口的属性
bwindow.h	加入或修改 wnread 认可的一个用户响应
bwindow.h	建立一个窗口结构
bwindow.h	移动当前窗口的光标
bwindow.h	返回当前窗口的光标位置
bwindow.h	激活一个窗口光标
bwindow.h	在同尺寸视口中显示一个窗口
bwindow.h	废弃一个窗口的结构
bwindow.h	记录窗口或选单的系统错误
bwindow.h,bedit.h, bstring.h	对窗口中的一个域进行编辑
bwindow.h	读取窗口信息项或状态

```
BWINDOW   *wnhoriz (int num_cols,int fore,int back,int dir);
    int    wninitev (BWINDOW *pwin);
BWINDOW   *wnorigin (BWINDOW *pwin,int row,int col,int option);
    int    wnprintf ( const char *pfmt, ...);
   char    wnquery (char *presp,int resp_size,int *pkey);
    int    wnrdbuf (int row,int col,int num_spaces,char *pbuffer,int option);
WN_EVENT  *wnread(BWINDOW *pwin,const WHERE *pwhere,int view_ht,int view_wid,
                 int org_row,int org_col,const BORDER *pbord,
                   WN_EVENT *pfinal,int option);
    int    wnredraw (int dev,int page);
    int    wnremevn(WN_EVENT_LIST **ppfirst,const WN_EVENT *pevent);
BWINDOW   *wnremove (BWINDOW *pwin);
BWINDOW   *wnrevupd (void);
BWINDOW   *wnscrblk (BWINDOW *pwin,int r1,int c1,int r2,int c2,int fore,
                   int back,int dir,int count,int option);
BWINDOW   *wnscrlbr (BWINDOW *pwin,int view_ht,int view_wid,unsigned attr,
                   int option);
BWINDOW   *wnscroll (int num_rows,int fore,int back,int dir);
BWINDOW   *wnselect (BWINDOW *pwin);
unsigned   wnsetbuf (unsigned size);
BWINDOW   *wnsetopt (BWINDOW *pwin,int item,int value);
BWINDOW   *wnshoblk (BWINDOW *pwin,const REGION *pblock,const LOC *phot1,
                   const LOC *phot2,int option);
BWINDOW   *wnupdate (BWINDOW *pwin);
BWINDOW   *wnvdisp (BWINDOW *pwin,const WHERE *pwhere,int view_ht,
                 int view_wid,int org_row,int org_col,const BORDER *pbord);
   void    wnwrap (int num_spaces,const char *pbuffer,int fore,int back,
                 int option);
    int    wnwrbuf (int row,int col,int num_spaces,const char *pbuffer,
                   int fore,int back,int option);
BWINDOW   *wnwrrect (BWINDOW *pwin,int u_row,int u_col,int l_row,int l_col,
                   const char *pbuffer,int fore,int back,int option);
   void    wnwrstr(const char *pbuffer,int fore,int back);
   void    wnwrstrn (BWINDOW *pwin,const char *pstring,int num_spaces,
                   int fore,int back,int option);
   void    wnwrtty(char ch,int fore,int back);
   void    wnzapevn(WN_EVENT_LIST **ppfirst);
```

bwindow.h	水平滚动当前窗口
bwindow.h	为 wnread 安装缺省的窗口事件
bwindow.h	在视口中移动窗口
bwindow.h	向当前窗口写入一个格式化的字符串
bwindow.h	返回经窗口得到的来自用户的字符串
bwindow.h	读取当前窗口中一片连续位置的内容
bwindow.h	允许用户在虚拟窗口中浏览
bwindow.h	重现显示在当前显示页上的全部窗口
bwindow.h	删去 wnread 接受的一个用户响应
bwindow.h	从屏幕上取消一个窗口
bwindow.h	用显示的数据更新已保存的窗口图像
bwindow.h	在窗口中以任意方向滚动一个矩形区域
bwindow.h	向窗口加入一个滚动箭头
bwindow.h	垂直滚动当前窗口
bwindow.h	选择用于 I/O 的窗口
bwindow.h	为 wnprintf 分配内部缓冲区
bwindow.h	设置窗口控制项
bwindow.h	在视口间隙中显示一个窗口数据块
bwindow.h	将挂起的输出写入窗口
bwindow.h	在视口中显示一个虚拟窗口
bwindow.h	以 TTY 方式向当前窗口写字符串,整字换行
bwindow.h	向当前视口中的一片连续位置写入字符
bwindow.h	写入窗口中的一个矩形区域
bwindow.h	以 TTY 方式向当前窗口写入一个字符串
bwindow.h	以 TTY 方式向窗口写字符串,带有任选项
bwindow.h	以 TTY 方式向当前窗口写入一个字符
bwindow.h	删除 wnread 认可的窗口事件表

[General Information]
书名=北京科海培训中心系列教材 C语言高级实用教程
作者=
页数=1000
SS号=812625272648
出版日期=
出版社=

前言

目录

前言
第1章　几个重要问题
1.1　数据类型转换
1.1.1　各类整数之间的转换
1.1.2　实数与整数之间的转换
1.1.3　指针之间的转换
1.2　指针
1.2.1　指针说明
1.2.2　指针与地址
1.2.3　指针运算
1.2.4　指针分类
1.2.4.1　近（near）指针
1.2.4.2　远（far）指针
1.2.4.3　巨（huge）指针
1.2.4.4　基（based）指针
1.2.5　各类指针之间的转换
1.3　函数
1.3.1　有返回值的函数
1.3.2　无返回值的函数
1.3.3　修改参数的函数
1.3.4　递归函数
1.3.5　参数个数不定的函数
1.3.6　函数指针及其应用
第2章　编译模式和内存组织
2.1　段与偏移量
2.2　六种编译模式
2.2.1　概述
2.2.2　微模式
2.2.3　小模式
2.2.4　中模式
2.2.5　紧凑模式
2.2.6　大模式
2.2.7　巨模式
2.3　堆栈的组织
2.4　堆的组织
2.5　其它内存操作函数
第3章　鼠标输入
3.1　鼠标驱动程序的基本功能
3.2　与鼠标接口的C函数工具包
3.2.1　14个工具函数
3.2.2　工具包应用举例
3.3　Turbo C Tools的鼠标支持函数
3.3.1　鼠标的初始化
3.3.2　询问鼠标的状态

3.3.3　鼠光标的位置和速度控制
3.3.4　鼠光标的形状和开关控制
3.3.5　对鼠标硬件中断的处理
第4章　文本屏幕输出和文本窗口
4.1　概述
4.2　Turbo C的文本屏幕处理
4.2.1　文本输出与操作
4.2.1.1　TTY输出规则
4.2.1.2　输出文本
4.2.1.3　对屏幕内容和光标的操作
4.2.1.4　屏幕与内存之间文本的移动
4.2.2　窗口和显示方式控制
4.2.3　属性控制
4.2.4　状态查询
4.3　pop__up文本窗口工具包
4.3.1　窗口结构和窗口栈
4.3.2　16个工具函数
4.3.3　工具包应用实例
4.3.3.1　用“瓷砖”式窗口制作菜单
4.3.3.2　移动pop__up窗口
4.3.3.3　pop__up错误信息和正常信息
4.4　Turbo C Tools的文本屏幕处理
4.4.1　Turbo C Tools与 Turbo C的比较
4.4.2　显示设备和显示方式
4.4.3　页控制
4.4.4　清除和滚动
4.4.5　光标控制
4.4.6　显示属性和颜色
4.4.7　屏幕写入和读取
4.4.8　矩形区域写入和读取
4.4.9　各种文本输出函数的速度比较
4.5　Turbo C Tools的窗口处理
4.5.1　概述
4.5.2　建立和注销窗口
4.5.3　显示和关闭窗口
4.5.4　窗口控制和状态
4.5.5　清除和滚动
4.5.6　光标位置查询和控制
4.5.7　属性控制
4.5.8　窗口输出／输入
4.5.9　虚拟窗口
4.5.10　常驻内存的窗口程序
4.6　Turbo C Tools的帮助信息窗口
4.6.1　帮助信息源文件
4.6.2　缺省帮助信息窗口
4.6.3　帮助函数

第5章　键盘输入、菜单和编辑
5.1　Turbo C的键盘输入
5.1.1　概述
5.1.2　控制台级（conio）键盘输入处理
5.1.3　标准文件级键盘输入处理
5.1.4　普通文件级键盘输入处理
5.1.5　BIOS级键盘输入处理
5.1.5.1　中断0×9
5.1.5.2　中断0×16
5.1.5.3　bioskey函数
5.2　〈Ctrl Break〉和〈Ctrl C〉
5.3　Turbo C Tools的键盘处理
5.3.1　键盘输入
5.3.2　缓冲区处理
5.3.3　状态控制键
5.3.4　使用加强键盘
5.3.5　使用键控制函数
5.3.6　取得键码
5.4　Turbo C Tools的菜单处理
5.4.1　概述
5.4.2　建立、显示和注销菜单
5.4.3　定义菜单项和按键鼠标事件
5.4.4　读取用户的选择
5.4.5　菜单应用举例
5.4.5.1　［例1］：一个简单演示程序
5.4.5.2　［例2］：用菜单实现电子表格
5.5　Turbo C Tools的域编辑
5.5.1　概述
5.5.2　域编辑
5.5.3　编辑键定义
第6章　基本文件处理
6.1　目录／文件系统概述
6.1.1　文件存取级别
6.1.2　文件属性
6.2　系统级输入／输出
6.2.1　文件柄
6.2.2　文件柄存取字节
6.2.3　文件柄属性字节
6.2.4　文件出错处理
6.2.5　建立文件
6.2.6　打开文件
6.2.7　读取和设置文件的特性
6.2.8　读、写和关闭文件
6.3　标准级（流式）输入输出
6.3.1　FILE数据结构
6.3.2　建立／打开／关闭／删除文件

6.3.3　取文件状态和出错处理
6.3.4　控制文件缓冲区
6.3.5　移动文件指针
6.3.6　字节级的读／写
6.3.7　字符串级的读／写
6.3.8　记录级的读／写
6.4　基本文件处理工具包
6.4.1　10个工具函数
6.4.2　工具包应用
6.5　驱动器和目录操作
6.5.1　驱动器和驱动器信息
6.5.2　目录操作
6.5.3　文件名操作
6.5.4　目录搜索
第7章　字符串处理
7.1　字符
7.1.1　字符数据和常数
7.1.2　字符输入／输出
7.1.3　字符的分类和转换
7.1.4　宏和宏的副作用
7.2　字符串
7.3　字符串的分析
7.4　字符串的综合
7.5　字符串的操作
7.6　文本字符串
7.7　数字字符串
7.8　国家和货币字符串
7.9　日期和时间字符串
7.10　文件名字符串
7.11　命令行字符串
7.12　环境字符串
7.13　错误级字符串
7.14　字符串处理工具包
7.14.1　13个工具函数
7.14.2　工具包应用举例：PASCAL程序翻译为C程序
第8章　动态通用串处理
8.1　动态字符串
8.2　动态通用串
8.3　动态通用串工具包
8.3.1　8个工具函数
8.3.2　工具包应用举例：多边形表示法
8.4　通用串的排序与查找
8.5　动态通用串与链表的比较
第9章　高级文件处理
9.1　变长记录（VLR）文件
9.1.1　从文件中查找一个记录

9.1.2　插入和删除记录
9.1.3　碎片化问题
9.1.4　VLR文件格式
9.1.5　VLR记录格式和数据块格式
9.2　VLR工具包
9.2.1　7个工具函数
9.2.2　索引处理
9.2.3　工具包应用举例：制作和显示幻灯片
第10章　内存和程序管理
10.1　PSP和环境
10.1.1　PSP
10.1.2　环境
10.2　内存管理
10.2.1　内存块及其控制
10.2.2　内存分布映像程序memrymap
10.2.3　内存管理函数
10.3　多个程序的执行及通信
10.3.1　程序间的通信
10.3.2　spawn：调用子进程
10.3.3　exec：转到子进程
10.3.4　system：执行DOS命令
10.3.5　signal和raise：事件处理
10.4　标准输入／输出重定向
10.4.1［例1］：freopen.dem
10.4.2［例2］：dup.dem
10.4.3［例3］：利用system
10.5　程序的终止
第11章　MSC 6.0的基指针技术
11.1　6种基（Based）指针
11.1.1　变量值基指针
11.1.2　变量地址基指针
11.1.3　不定基指针
11.1.4　段名基指针
11.1.5　指针基指针
11.1.6　自参照基指针
11.2　基指针应用于链表管理的工具包
11.2.1　基指针应用于链表管理
11.2.2　基指针分配函数
11.2.3　16个工具函数
11.2.4　工具包应用三例
11.2.4.1　［例1］
11.2.4.2　［例2］和［例3］
第12章　与BIOS和DOS的接口
12.1　中断概述
12.2　与BIOS的接口
12.2.1　与BIOS接口的函数

12.2.2　BIOS提供的部分服务
12.3　与DOS的接口
12.3.1　与DOS接口的函数
12.3.2　DOS提供的部分服务
12.4　标准输入／输出服务
12.4.1　BIOS提供的显示服务
12.4.2　BIOS提供的键盘服务
12.4.3　DOS提供的标准输入／输出服务
12.5　文件输入／输出服务
12.6　内存管理与程序执行服务
12.7　打印服务
12.8　时钟／日历服务
12.8.1　PC机上的时钟系统
12.8.2　PC/AT机上的时钟系统
12.8.3　DOS的时间／日历服务
12.8.4　延迟函数
12.8.5　声音函数
12.9　串行通信服务
12.10　错误处理服务
12.10.1　DOS怎样报告错误
12.10.2　Turbo C库函数的错误报告特性
12.10.3　致命错
12.10.4　〈Ctrl Break〉和〈Ctrl C〉
第13章　中断服务程序
13.1　一般概念
13.2　用Turbo C Tools写中断服务程序
13.2.1　工作原理
13.2.2　安装和驻留
13.2.3　过滤
13.2.4　探测和撤消
13.2.5　其它
13.3　中断服务程序实例
13.3.1　［例1］：周期性地发声
13.3.2　［例2］：检测A和J键的同时按下
13.3.3　［例3］：发送格式化的输出
13.4　用Turbo C Tools写插入服务程序
13.4.1　DOS的重入问题
13.4.2　插入服务技术
13.4.3　插入服务函数
13.4.4　插入服务程序举例
13.5　用Turbo C写中断服务程序
第14章　图形处理
14.1　Turbo C图形处理函数
14.1.1　概述
14.1.2　图形系统控制函数
14.1.3　画图和填充函数

14.1.4 屏幕和视口管理函数
14.1.5 图形方式下的文本输出函数
14.1.6 颜色控制函数
14.1.7 错误处理函数
14.1.8 状态查询函数
14.2 Pop up图形窗口工具包
14.2.1 图形窗口与文本窗口
14.2.2 6个工具函数
14.2.3 工具包应用举例：移动窗口
14.3 图形方式下输出文本的若干问题
14.3.1 格式输出
14.3.2 重写
14.3.3 加亮
14.3.4 滚动
14.4 用XOR方式画旋转橡皮筋
第15章 混合模式和混合语言编程
15.1 混合模式编程
15.1.1 概述
15.1.2 说明一个函数为near或far
15.1.3 说明一个指针为near、far或huge
15.1.4 使用库文件
15.1.5 不同编译模式所生成模块的连接
15.2 C和汇编语言混合编程
15.2.1 段的组合
15.2.1.1 汇编语言的段和组
15.2.1.2 Trubo C的段和组
15.2.1.3 段和组的连接
15.2.2 变量和函数名的相互引用
15.2.3 参数传递规则
15.2.4 返回值传递规则
15.2.5 寄存器规则
15.2.6 混合编程示例
15.2.6.1 C调用汇编
15.2.6.2 汇编调用C
15.3 行内汇编
附录1 操作符表
附录2 键盘码表
附录3 Turbo C 2.0函数简表
附录4 Turbo C Tools 6.0函数简表
目录