

Turbo C 2.0

实用大全

董玉忠 葛平华 张再良 编写
杨和庆 主编



北京航空航天大学出版社

Turbo C 2.0

实用大全

常玉龙
葛丰年
张再良
孙和达

编写

主审

北京航空航天大学出版社

(京)新登字 166 号

内 容 清 要

本书共五部分。第一部分: Turbo C 集成环境的功能和用法;组成 Turbo C 屏幕的要素;如何用 Turbo C 进行编辑、编译、调试、连接和运行; Turbo C 编辑器、调试器和多文件管理的工程文件的使用方法;第二、第三部分: Turbo C 语言基础和高级的 Turbo C 程序设计技术;包括指针、结构和联合的常用的各种数据结构;循环、条件等各种语句;屏幕和文件处理;与汇编语言的接口等。第四部分: Turbo C 库函数和全局变量的功能、用法和调用实例。第五部分为总附录: TCC 命令行编辑器;实用程序等。可供计算机软件工程人员及大专院校师生使用。

● 书 名: Turbo C 2.0 实用大全

Turbo C 2.0 SHIYONG DAQUAN

● 作 者: 常玉龙 葛丰年 张再良 编写 孙和达 主审

● 责任编辑: 许传安

● 出 版: 北京航空航天大学出版社

(北京市学院路 37 号, 邮编 100083, 发行部电话 62015720)

● 印 刷: 河北省涿州市新华印刷厂

● 总 发 行: 北京航空航天大学出版社

● 发 行: 新华书店总店北京发行所

● 销 售: 各地书店

● 开 本: 787×1092 1/16

● 印 张: 63.75 字数: 1632 千字

● 版 次: 1994 年 9 月第一版, 1998 年 3 月第三次印刷

● 印 数: 7001~9000 册

● 定 价: 70.00 元

● 书 号: ISBN 7-81012-508-7/TP·125

目 录

绪 论

0.1 两种 Turbo C 环境	1
0.2 设置自己的任务环境	2
0.3 Turbo C 编译器	2
0.4 Turbo C 工程制作实用程序	3
0.5 低级功能的支持	3
0.6 配置 Turbo C	4
0.7 存储模式	4

第一部分 掌握 Turbo C

第一章 Turbo C 集成开发环境

1.1 TC 的使用	7
1.1.1 TC 命令行开关	8
1.1.2 TC 内部命令的使用	9
1.1.3 TC 热键	10
1.1.4 菜单结构	11
1.1.5 主菜单	11
1.1.6 功能键提示行	12
1.1.7 编辑窗口	12
1.1.8 消息窗口	15
1.1.9 监视窗口	15
1.1.10 集成调试器	16
1.2 菜单命令	18
1.2.1 File(文件)菜单	18
1.2.2 Edit(编辑)命令	20
1.2.3 Run(运行)菜单	20
1.2.4 Compile(编译)菜单	22
1.2.5 Project 菜单	23
1.2.6 Options 菜单	25
1.2.7 Debug 菜单	37
1.3 配置和 pick 文件	42
1.3.1 TC 配置文件	42
1.3.2 Pick 表和 Pick 文件	44

第二章 使用 Turbo C 编辑器

2.1 编辑器命令	46
-----------------	----

2.2 编辑器激活及文本键入	46
2.3 字符、字和行的删除	47
2.4 移动、拷贝文本和块移动	48
2.5 进一步说明如何移动光标	49
2.6 字符系列的搜索和替换	50
2.7 位置标识的设置和搜索	51
2.8 存储和装入文件	52
2.9 自动缩进	52
2.10 磁盘文件中文本块的移入和移出	52
2.11 对匹配	53
2.12 其它有关命令	53
2.13 命令综述	54
2.14 用文件激活 Turbo C	55

第三章 Turbo C 调试器

3.1 调试与程序开发	56
3.2 设计示例程序, PLOTEMP.C	57
3.3 编写原型程序	58
3.4 使用集成调试工具	60
3.5 跟踪程序的流程	60
3.5.1 跟踪高层的运行	60
3.5.2 跟踪子函数	61
3.6 继续程序的开发	61
3.7 设置断点	64
3.7.1 用 Ctrl-Break 立即中断	65
3.8 计算和修改变量	65
3.8.1 指定显示格式	66
3.8.2 指定值的个数	66
3.8.3 从光标所在位置拷贝	66
3.8.4 查看在别的函数中的变量	66
3.8.5 修改值	67
3.9 通过设置监视项来监视程序运行	70
3.9.1 添加一个监视项	70
3.9.2 观察监视项	70
3.9.3 控制调试器窗口	71
3.9.4 编辑和删除监视项	71
3.9.5 寻找一个函数定义	71
3.9.6 查找调用关系	72

3.9.7 多个源文件	72
3.10 预防措施	72
3.11 有系统的软件测试	73
3.11.1 全面测试修改结果	73
3.11.2 仔细观察的部分	73
3.12 完成 PLOTEMP.C	74
3.12.1 完成 table_view()	74
3.12.2 实现 graph_view()	75
3.12.3 save_temps()和 read_temps()	77

第四章 多文件工程管理

4.1 工程管理程序的使用	79
4.2 出错跟踪	81
4.2.1 终止 MAKE	81
4.2.2 多源文件的语法错误	81
4.2.3 保存或删除信息	82
4.3 工程管理程序的功能	82
4.3.1 自身依赖性检查	82
4.4 取代库	83
4.5 工程管理程序的其他功能	83
4.6 生成最终应用程序的集成环境设置	84

第二部分 Turbo C 语言基础

第五章 C 语言概貌

5.1 预备知识	89
5.1.1 源文件、目标文件和装载模块	89
5.1.2 程序的逻辑和执行流程	90
5.2 基本数据类型	96
5.2.1 C 语言的基本数据类型	96
5.2.2 何处定义数据对象	98
5.3 编写 C 语言表达式和语句	101
5.3.1 表达式和语句	101
5.4 控制类型转换	105
5.5 显式类型转换的使用	107
5.6 使用 C 的宏	107
5.6.1 定义类似对象的宏	108
5.6.2 定义类似函数的宏	110

第六章 操作符和表达式

6.1 什么是操作符	115
6.2 单目操作符	116

6.3 双目操作符	116
6.4 三目操作符	117
6.5 标点符号	117
6.6 操作符语义	119
6.6.1 后缀和前缀操作符	119
6.6.2 单目操作符	121
6.6.3 sizeof 操作符	122
6.6.4 乘法类操作符	122
6.6.5 加法类操作符	123
6.6.6 按位移位操作符	123
6.6.7 关系操作符	124
6.6.8 相等类操作符	125
6.6.9 位运算操作符	125
6.6.10 逻辑运算符	126
6.6.11 条件操作符?:	127
6.6.12 赋值操作符	127
6.6.13 逗号操作符	128
6.7 高级运算符的使用实例	128
6.7.1 位运算符	128
6.7.2 移位运算符	130
6.7.3 ?:运算符	134
6.7.4 C 语言的简写	136
6.7.5 逗号运算符	136
6.7.6 运算符优先级表	136
6.8 表达式	136

第七章 说明

7.1 有关概念	139
7.1.1 对象	139
7.1.2 左值	139
7.1.3 右值	140
7.1.4 类型与存储类	140
7.1.5 作用域	140
7.1.6 可见性	141
7.1.7 生存期	141
7.1.8 编译单元	142
7.1.9 连接	142
7.2 说明的语法	143
7.2.1 暂时定义	143
7.2.2 可能的说明	144
7.3 类型说明	147
7.3.1 外部说明与定义	147
7.3.2 类型指明符	147

7.3.3 类型分类	147
7.3.4 基本类型	148
7.3.5 标准转换	151
7.3.6 特殊的 char、int 与 enum 间的转换	152
7.3.7 初始化	152
7.4 简单说明	154
7.5 存储类指明符	155
7.5.1 存储类指明符 auto 的使用	155
7.5.2 存储类指明符 extern 的使用	155
7.5.3 存储类指明符 register 的使用	155
7.5.4 存储类指明符 static 的使用	155
7.5.5 存储类指明符 typedef 的使用	155
7.6 修饰符	156
7.6.1 const 修饰符	156
7.6.2 中断函数修饰符	157
7.6.3 volatile 修饰符	157
7.6.4 cdecl 与 pascal 修饰符	158
7.6.5 指针修饰符	159
7.6.6 函数类型修饰符	159
7.7 复杂说明与说明符	159

第八章 程序控制语句

8.1 程序控制语句的语法	161
8.1.1 带标号语句	162
8.1.2 复合语句	163
8.1.3 表达式语句	163
8.1.4 选择语句	163
8.1.5 循环语句	164
8.1.6 跳转语句	165
8.2 if 语句	166
8.2.1 else 语句的用法	168
8.2.2 if-else-if 阶梯的用法	168
8.2.3 条件表达式	169
8.2.4 if 语句的嵌套结构	170
8.3 switch 语句	170
8.3.1 default 语句的用法	172
8.3.2 break 语句的用法	173
8.3.3 switch 语句的嵌套结构	174
8.4 循环	176
8.5 for 循环	176
8.5.1 for 循环的灵活用法	177
8.5.2 无穷的 for 循环	179

8.5.3 无穷 for 循环的中断	179
8.5.4 空循环的用法	179
8.6 while 循环语句	179
8.7 do while 循环	181
8.8 循环嵌套	182
8.9 循环中断	184
8.10 continue 语句	186
8.11 goto 语句	187

第九章 函数

9.1 函数的初步概念	189
9.1.1 说明与定义	189
9.1.2 说明与原型	189
9.1.3 定义	190
9.1.4 形参说明	191
9.1.5 函数调用与参数转换	191
9.2 return 语句	192
9.2.1 从一个函数中返回	192
9.2.2 返回值	193
9.2.3 函数返回非整型值	195
9.3 有关函数原型的进一步说明	198
9.3.1 参数不匹配	198
9.3.2 使用头文件	199
9.3.3 无任何参数的函数原型	199
9.3.4 有关旧式 C 程序	199
9.4 作用域规则	200
9.4.1 局部变量	200
9.4.2 形式参数	202
9.4.3 全局变量	202
9.4.4 有关作用域的最后一个例子	203
9.5 有关函数的参数和变量的更详尽说明	204
9.5.1 赋值调用和赋地址调用	204
9.5.2 一个赋地址调用的建立	205
9.5.3 数组与函数调用	206
9.6 argc, argv 和 env —— main 中的参数	209
9.7 从 main() 中返回值	211
9.8 递归	212
9.9 参数说明的传统形式和现代形式的比较	214
9.10 对一些影响函数的效率和实用性问题的讨论	214
9.10.1 参数和通用函数	214

9.10.2 效率	215
9.11 库函数	215
9.12 改变程序的执行流程	218
9.12.1 使用 exit() 和 abort() 函数	218
9.12.2 使用 system(), exec...() 和 spawn() 函数	219
9.13 使用可变参数表	220
9.13.1 设计可变参数表	221
9.13.2 使用 va...() 函数	221

第十章 指针

10.1 指针的语法规则	226
10.1.1 什么是指针	226
10.1.2 指针说明	227
10.1.3 指针与常量	227
10.1.4 指针算术运算	228
10.1.5 指针转换	229
10.2 指针是地址	229
10.3 指针变量	229
10.4 指针操作符	230
10.5 指针表达式	231
10.5.1 指针赋值	231
10.5.2 指针运算	231
10.5.3 指针比较	233
10.6 指针和数组	233
10.6.1 索引指针	234
10.6.2 指针和字符串	234
10.6.3 如何得到一个数组元素的地址	235
10.6.4 指针数组	236
10.6.5 一个使用数组和指针的实例	237
10.7 指针的指针	241
10.8 指针的初始化	242
10.9 指针的一些问题	243
10.9.1 使用 C 语言的间接操作符和取地址操作符	244
10.9.2 使用数组和串	246
10.10 使用指向函数的指针	252
10.10.1 指向函数的指针说明和初始化	252
10.10.2 利用指针引用某调用函数	253
10.11 在动态内存中使用指针	256
10.11.1 C 语言程序和动态内存	257
10.11.2 使用动态存储	258

第十一章 数组、结构、位域、联合和枚举

11.1 高级数据类型的语法规则	265
11.1.1 数组	265
11.1.2 结构	265
11.1.3 位域	269
11.1.4 联合	269
11.1.5 枚举	270
11.2 数组	271
11.2.1 一维数组	272
11.2.2 字符串	273
11.2.3 二维数组	278
11.2.4 多维数组	280
11.2.5 数组初始化	280
11.2.6 一个水下搜索游戏	282
11.3 结构	285
11.3.1 访问结构元素	286
11.3.2 结构数组	287
11.3.3 结构赋值	294
11.3.4 将结构传递给函数	294
11.3.5 结构指针	296
11.3.6 结构内部的数组和结构	299
11.4 位域	300
11.5 联合 (union)	303
11.6 枚举	306
11.7 使用 sizeof 来确保可移植性	308
11.8 typedef	309

第十二章 Turbo C 预处理程序指令

12.1 空指令 #	313
12.2 #define 与 #undef 指令	313
12.2.1 简单的 #define 宏	313
12.2.2 #undef 指令	314
12.2.3 -D 与 -U 选择项	315
12.2.4 关键字与保护字	315
12.2.5 带参宏	315
12.3 文件包含指令 #include	317
12.3.1 <头名>形式的头文件搜索	318
12.3.2 “头名”形式的头文件搜索	318
12.4 条件编译	318
12.4.1 #if、#elif、#else 和 #endif 条件指令	316

12.4.2	defined 运算符	319
12.4.3	#ifdef 和 #ifndef 条件指令	319
12.5	#line 行控制指令	320
12.6	#error 指令	321
12.7	#pragma 指令	321
12.7.1	#pragma argused	322
12.7.2	#pragma exit 与 #pragma startup	322
12.7.3	#pragma inline	322
12.7.4	#pragma option	323
12.7.5	#pragma saveregs	324
12.7.6	#pragma warn	324
12.8	预定义的宏	325

第三部分 高级 C 程序设计技巧

第十三章 文件输入输出

13.1	两个预处理指令	329
13.1.1	#define 指令	329
13.1.2	#include 指令	331
13.2	文件与流	331
13.3	流(streams)	331
13.3.1	文本流	331
13.3.2	二进制流	332
13.3.3	文件	332
13.4	概念和实际	332
13.5	控制台 I/O	333
13.5.1	字符读写	333
13.5.2	字符串读写	334
13.6	控制台格式化 I/O	334
13.6.1	printf()函数	335
13.6.2	scanf()函数	336
13.7	缓冲型 I/O 系统(ANSI 型 I/O 系统)	339
13.7.1	文件指针	339
13.7.2	打开文件	339
13.7.3	写字符	341
13.7.4	读字符	341
13.7.5	feof()的使用	342
13.7.6	关闭文件	342
13.7.7	ferror()和 rewind()函数	342
13.7.8	fopen(),getc(),putc()和 fclose()函数的 用法	343

13.7.9	getw()和 putw()函数的使用	345
13.7.10	fgets()和 fputs()函数	345
13.7.11	fread()和 fwrite()函数	345
13.7.12	fseek()函数和随机访问 I/O	347
13.7.13	标准流	349
13.7.14	fprintf()和 fscan()函数	350
13.7.15	删除文件	352

13.8 非缓冲型 I/O——UNIX 型

文件系统	352
13.8.1 open(), creat()和 close()函数	353
13.8.2 read()和 write()函数	354
13.8.3 unlink()函数	356
13.8.4 随机访问文件和 lseek()函数	356
13.9 理解 I/O 概念	357
13.9.1 文件与设备	358
13.9.2 文件与流	359
13.9.3 文本流和二进制流	360
13.10 利用标准流进行 I/O	361
13.10.1 使用格式化 I/O 函数	361
13.10.2 scanf()函数	365
13.10.3 prints()函数	368
13.10.4 使用字符 I/O 函数	370
13.11 使用文件控制函数	374
13.11.1 开文件、关文件和控制文件	374
13.11.2 控制文件缓冲区	377
13.12 使用直接文件 I/O 函数	378
13.12.1 理解直接 I/O 概念	379
13.12.2 读写直接文件	380
13.13 使用文件定位函数	385
13.13.1 得到当前文件位置	385
13.13.2 建立一个新文件位置	386
13.14 处理文件 I/O 错误	388
13.14.1 查出文件 I/O 错误	388
13.14.2 显示和清除文件 I/O 错误	388

第十四章 屏幕文本和图形程序设计

14.1	图形系统和要素	390
14.1.1	视频模式	390
14.1.2	窗口和视区	390
14.1.3	在文本模式下编程	391
14.1.4	在图形模式下编程	396
14.2	Turbo C 图形程序设计	405
14.2.1	基本正文模式函数	406

14.2.2 Turbo C 的图形子系统简介	414
14.3 IBM/PC 的文本方式	422
14.3.1 PC 显示器适配器和屏幕	422
14.3.2 视频缓冲区 I/O	423
14.4 控制文本屏幕	424
14.4.1 使用文本方式控制函数	424
14.4.2 使用直接控制台 I/O 以获 得高性能	427
14.5 使用窗口函数	427
14.6 了解 IBM-PC 的图形方式	431
14.6.1 象素点与调色板	431
14.6.2 控制图形屏幕	433
14.7 介绍 BGI 图形库	434
14.7.1 使用画图和填充函数	434
14.7.2 控制屏幕和视口	438
14.8 在图形方式下显示文本	439
14.8.1 BGI 字库	439
14.8.2 使用图形方式下的文本函数	440

第十五章 存储模式

15.1 80×86 的体系结构	443
15.1.1 段(Segment)、节(Paragraph)以及偏移 地址(Offset)	443
15.1.2 CPU 的地址寄存器	445
15.2 near 指针、far 指针和 huge 指针	446
15.2.1 选择想要的指针大小	446
15.2.2 near、far 和 huge 说明符	448
15.3 六个 Turbo C 存储模式	450
15.3.1 决定使用哪种存储模式	450
15.3.2 以混合模式编程	451
15.4 创建 COM 型的可执行程序文件	453
15.4.1 使用 COM 文件	453

第十六章 与汇编语言的接口

16.1 混合语言程序设计	456
16.1.1 参数传递顺序	456
16.2 建立从 Turbo C 对 ASM 的调用	458
16.2.1 简化的段指令	458
16.2.2 标准段指令	459
16.2.3 定义数据常量和变量	459
16.2.4 定义全局和外部标识符	460
16.3 建立从 ASM 中对 Turbo C 的调用	461

16.3.1 引用函数	461
16.3.2 引用数据	461
16.4 定义汇编语言过程	462
16.4.1 传递参数	462
16.4.2 处理返回值	463
16.5 寄存器约定	466
16.6 从 ASM 过程中调用 C 函数	466
16.7 伪变量、嵌入汇编和中断函数	468
16.7.1 伪变量	468
16.7.2 嵌入汇编语言	470
16.7.3 中断函数	475
16.8 使用直接插入(inline)汇编语言	477
16.8.1 直接插入式汇编环境	478
16.8.2 使用 asm 关键字	478
16.9 与汇编语言例程的接口	481
16.9.1 在 C 程序里调用汇编例程	481
16.9.2 在汇编例程中调用 C 函数	490
16.10 使用中断功能	497
16.10.1 80×86 的中断结构	497
16.10.2 使用 Borland 的中断接口	498
16.11 使用中断处理程序	501
16.11.1 声明中断处理程序函数	501
16.11.2 实现一个时钟中断处理程序	504

第四部分 库函数和全局变量参考

第十七章 Turbo C 标准库函数

函数名 函数功能描述	511
abort 异常终止一进程	511
abs 返回整数的绝对值	512
absread 读磁盘的绝对扇区	512
abswrite 写磁盘绝对扇区	513
access 确定文件的存取权限	514
acos 计算反余弦值	515
allocmem 分配 DOS 内存	515
arc 画圆弧	516
asctime 转换日期和时间对应的 ASCII 码	517
asin 反正弦函数	518
assert 条件终止函数	519
atan 反正切函数	519
atan2 计算 y/x 的反正切值	520
atexit 定义终止函数	521

atof 将字符串转换成浮点数	521	cprintf 格式化并输出数据至屏幕	559
atoi 把字符串转换成整数	522	cputs 输出一字符串至屏幕	560
atol 把字符串转换成整型	523	_creat 创建一个新文件或重写一个已存在的文件	561
bar 画二维条形图	523	creat 创建一个新文件或重写一个已存在的文件	562
bar3d 画一个三维条形图	525	creatnew 创建新文件	563
bdos DOS 系统调用	526	creattemp 创建一个文件名唯一的文件	564
bdosptr DOS 系统调用	527	cscanf 从控制台执行格式化输入	565
bioscom I/O 通信	528	ctime 把日期和时间转化为对应的字符串	565
biosdisk 调用 BIOS 磁盘驱动程序	530	ctrlbrk 设置 ctrl-break 处理程序	566
biosequip 检查设备	532	delay 暂停	567
bioskey 调用 BIOS 的键盘接口	533	delline 在文本窗口中删去一行	567
biosmemory 返回内存大小	535	detectgraph 检测硬件并确定应使用何种图形驱动程序和图形模式	568
biosprint 调用 BIOS 的打印机 I/O 接口	535	difftime 计算两个时刻之间的时间差	571
biostime 读取或设置 BIOS 时钟	536	disable 屏蔽中断	571
brk 改变数据段内存分配	537	div 将两个整数相除, 返回商和余数	572
bsearch 数组的二分法搜索	538	dosexterr 获取扩展错误信息	573
cabs 计算复数的模	539	dostounix 把日期和时间转换成 UNIX 格式	574
calloc 分配内存	540	drawpoly 绘制多边形	574
ceil 舍入	540	dup 复制文件句柄	576
cgets 读字符串	541	dup2 将一个文件句柄(oldhandle)复制到一个已有的文件句柄(newhandle)	577
chdir 改变当前目录	542	ecvt 把浮点数转换为字符串	578
_chmod 改变文件的存取权限	543	ellipse 绘制椭圆	579
chmod 改变文件存取权限	545	_emit_ 将文字值直接插入源程序中	580
chsize 修改文件长度	545	enable 开硬件中断	581
circle 画圆	546	eof 检测文件是否结束	583
_clear87 清除浮点状态字	547	_exit 终止程序	587
cleardevice 清图形屏幕	548	exit 终止程序	588
clearerr 复位错误标志	549	exp 计算 e 的 x 次方	588
clearviewport 清除当前图形窗口	550	fabs 返回浮点数的绝对值	589
clock 测定运行时间	551	farcalloc 从远程堆中分配内存	589
_close 关闭文件	552	farcoreleft 返回远程堆中未使用内存的大小	590
close 关闭文件	552	farfree 从远程堆中释放一块已分配内存	591
closegraph 关闭图形系统	553	farmalloc 从远堆中分配内存	592
clreol 清除从当前光标位置到行尾的字符	554	farrealloc 调整远堆中的已分配块	593
clrscr 清除文本窗口, 并把光标放在左上角	555		
_control87 处理浮点控制字	556		
coreleft 返回尚未使用的内存(RAM)大小	557		
cos 计算余弦值	557		
cosh 计算双曲余弦值	558		
country 读取与特定国家有关的格式	558		

<code>fclose</code> 关闭一个流	593	<code>fwrite</code> 把参数写入流中	630
<code>fcloseall</code> 关闭打开流	594	<code>gcvt</code> 把浮点数转换为字符串	630
<code>fcvt</code> 将浮点数转换为字符串	595	<code>geninterrupt</code> 产生软中断	631
<code>fdopen</code> 把流与一个文件句柄相联	595	<code>getarccorrs</code> 取得最后一次调用 <code>arc</code> 的 坐标	632
<code>feof</code> 检测流上的文件结束标志	597	<code>getaspectratio</code> 返回当前图形模式的纵 横比	634
<code>ferror</code> 检测流上的错误	597	<code>getbkcolor</code> 返回当前背景颜色	635
<code>fflush</code> 刷新一个流	598	<code>getc</code> 从流中取字符	636
<code>fgetc</code> 从流中读取字符	599	<code>getcbrk</code> 获取 <code>control-break</code> 状态	637
<code>fgetchar</code> 从流中读取字符	600	<code>getch</code> 从键盘无回显地读取一字符	637
<code>fgetpos</code> 取得当前文件指针	600	<code>getchar</code> 从 <code>stdin</code> 流中读取一个字符	638
<code>fgets</code> 从流中读取一字符串	601	<code>getche</code> 从键盘并回显地读取一字符	638
<code>filelength</code> 取文件长度	602	<code>getcolor</code> 返回当前绘图颜色	639
<code>fileno</code> 取得文件句柄	602	<code>getcurdir</code> 读取指定驱动器的当前目录	640
<code>fileellipse</code> 画椭圆饼	603	<code>getcwd</code> 读取当前目录	641
<code>fillpoly</code> 画多边形	604	<code>getdate</code> 读取系统日期	642
<code>findfirst</code> 查找第一个匹配文件	605	<code>getdefaultpalette</code> 返回缺省调色板信息	642
<code>findnext</code> 查找下一个匹配文件	607	<code>getdfree</code> 读取磁盘空闲空间	643
<code>floodfill</code> 填充区域	608	<code>getdisk</code> 读取当前磁盘驱动器号	644
<code>floor</code> 下舍入	609	<code>getdrivername</code> 返回指向当前图形驱动程序名字 的指针	645
<code>flushall</code> 刷新所有流	610	<code>getdta</code> 读取磁盘传输地址	646
<code>fmod</code> 计算 x/y 的余数	610	<code>getenv</code> 读取环境变量的当前值	647
<code>fnmerge</code> 建立文件路径	611	<code>getfat</code> 读取指定驱动器的 <code>FAT</code> 信息	647
<code>fnsplit</code> 分解完整的路径名	611	<code>getfatd</code> 读取驱动器 <code>FAT</code> 信息	648
<code>fopen</code> 打开一个流	613	<code>getfillpattern</code> 将用户定义的填充模式拷贝 到内存	649
<code>FP_OFF</code> 获取远地址偏移量	614	<code>getfillsettings</code> 取得当前填充模式和填充颜色的 有关信息	651
<code>_fpreset</code> 重新初始化浮点数学包	615	<code>getftime</code> 读取文件日期和时间	653
<code>fprintf</code> 传送输出到一个流中	617	<code>getgraphmode</code> 返回当前图形模式	654
<code>FP_SEG</code> 获取远地址段值	617	<code>getimage</code> 将指定区域的位图象存入 内存	655
<code>fputc</code> 送一个字符到一个流中	618	<code>getline</code> 读取当前线型、模式和 宽度	657
<code>fputchar</code> 送一个字符到标准输出	619	<code>getmaxcolor</code> 返回可选的最大有效颜 色值	659
<code>fputs</code> 送一个字符串到流中	619	<code>getmaxmode</code> 返回当前驱动程序的最大图形模 式号	660
<code>fread</code> 从流中读数据	620	<code>getmaxx</code> 返回屏幕上最大的 x 坐标值	661
<code>free</code> 释放已分配的内存	621	<code>getmaxy</code> 返回屏幕上最大的 y 坐标值	662
<code>freemem</code> 释放先前分配的 <code>DOS</code> 内存	621	<code>getmodename</code> 返回指向含有指定图形	
<code>freopen</code> 把一个新文件同个打开 的流相联	622		
<code>frexp</code> 对双精度数进行科学计数	623		
<code>fscanf</code> 格式化输入	623		
<code>fseek</code> 移动文件指针	624		
<code>fsetpos</code> 定位文件指针	625		
<code>fstat</code> 获取已打开文件的信息	627		
<code>ftell</code> 返回当前文件指针	628		
<code>ftime</code> 把当前时间存入 <code>timeb</code> 结构中	629		

模式名字符串的指针	663	区大小	700
getnoderange 获取图形驱动程序的模式范围	664	initgraph 初始化图形系统	702
getpalette 返回当前调色板的有关信息	665	inport inp 从端口中读入一个字	705
getpalettesize 返回调色板的颜色数目	667	inportb 从端口中读入一个字节	706
getpass 读入口令	668	insline 在文本窗口插入一空行	706
getpid 读取进程号	669	installuserdriver 安装设备驱动程序到 BGI 设备驱动程序表中	707
getpixel 读取像素的颜色	669	installuserfont 安装未嵌入 BGI 系统的字体文件(.CHR)	709
getpsp 读取程序段前缀	671	int86 调用 8086 软中断	710
gets 从标准输入流 stdin 中读取一字符串	671	int86x 通用 8086 软中断接口	711
gettext 拷贝文本屏幕上的文本拷贝到内存中	672	intdos 通用 DOS 中断接口	712
gettextinfo 读取文本模式的显示信息	673	intdosx 通用 DOS 中断接口	713
gettextsettings 返回当前图形字体的有关信息	674	intr 改变软中断接口	714
gettime 读取系统时间	676	ioctl I/O 设备控制	715
getvect 读取中断向量	677	isalnum 字符分类宏	716
getverify 取得 DOS 的当前校验状态	677	isalpha 字符分类宏	717
getviewsettings 返回有关当前视区的信息	678	isascii 字符分类宏	717
getw 从输入流中读取一整数	679	isatty 检查设备类型	718
getx 返回当前图形方式下位置的 x 坐标值	681	iscntrl 字符分类宏	718
gety 返回当前位置的 y 坐标值	682	isdigit 字符分类宏	719
gmtime 把日期和时间转换为格林威治标准时间(GMT)	683	isgraph 字符分类宏	720
gotoxy 在文本窗口中定位文本光标	684	islower 字符分类宏	720
graphdefaults 复位图形设置	684	isprint 字符分类宏	721
grapherrormsg 返回一个指向错误信息串的指针	685	ispunct 字符分类宏	721
_graphfreemem 可修改的图形内存释放函数	686	isspace 字符分类宏	722
_graphgetmem 可修改的图形内存分配函数	688	isupper 字符分类宏	722
graphresult 返回最后一次失败图形操作的错误码	689	isxdigit 字符分类宏	723
Harderr 建立一个错误处理程序	691	itoa 把整数转换为字符串	723
hardresume 硬件错误处理函数	694	kbhit 检查当前按下的键	724
hardretn 硬件错误处理函数	697	keep 驻留并退出	724
highvideo 选择高亮度字符	699	labs 给出长型绝对值	726
hypot 计算直角三角形的斜边长	700	ldexp 计算 x 乘以 2 的 exp 次方	726
imagesize 返回保存位图象所需的缓冲		ldiv 两个长整型数相除, 返回商和余数	727
		lfind 线性搜索	728
		line 在指定两点间画一直线	729
		linerel 从当前位置(CP)到与 CP 有一相对距离的点画一直线	730
		lineto 从当前位置到(x,y)画一直线	731
		localtime 把日期和时间转变为结构类型	732
		lock 设置文件共享锁	733
		log 计算 x 的自然对数	734

log10 计算 $\log(X)$	735	outtextxy 在指定位置显示一字符串	762
longjmp 执行非局部跳转	735	parsfnm 分析文件名	763
lowvideo 选择低亮度字符	736	peek 返回由 segment,offset 指定的 内存中的字	764
_lrotl 将无符号长整型数向 左循环移位	737	peekb 返回由 segment,offset 指定的 内存中的字节	765
_lrotr 将无符号长整型数向 右循环移位	738	perror 打印系统错误信息	766
lsearch 线性搜索	738	pieslice 绘制并填充扇形	767
lseek 移动文件指针	740	poke 在由 segment,offset 指定的内存中 存储一个字	768
ltoa 把一个长整型数转换为字符串	741	pokeb 在由 segment,offset 指定的内存中 存储一个字节	768
malloc 分配内存	741	poly 根据参数产生一个多项式	769
matherr 用户可修改的数学错误处理 程序	742	pow 计算 x 的 y 次方	770
max 返回两数中较大的数	744	pow10 指数函数 10 的 p 次方	770
memccpy 拷贝一个 n 字节长的字符串	744	printf 写格式化输出到 stdout	771
memchr 字符串中搜索字符	745	putc 输出一个字符到流中	777
memcmp 比较两个字符串	745	putch 向屏幕输出字符	777
memcpy 拷贝字符串	746	putchar 在 stdout 上输出字符	778
memcmp 比较两个字符数组中的 n 个字节,忽 略大小写	747	putenv 将字符串放入当前环境中	779
memmove 拷贝块中的 n 字符	747	putimage 输出一个位图象到图形屏 幕上	780
memset 将一个内存块的 n 个字节都设 置为 c	748	putpixel 写像素点	782
min 返回两个值中较小的一个	748	puts 输出一字符串到标准输出(stdout) ...	783
mkdir 创建目录	749	puttext 从内存区拷贝文本到屏幕	784
MK_FP 设置一个远指针	750	putw 输出一整数到流中	784
mktemp 建立一个唯一的文件名	751	qsort 用快速排序算法进行排序	786
modf 把双精度数转化为科学计数法	751	raise 向正在执行的进程发送一个软 中断信号	787
movedata 拷贝数据	752	rand 产生随机数	787
moverel 从当前位置(CP)移动一相对 距离	752	randbrd 随机块读	788
movetext 将屏幕上的文本从一个矩形区域拷贝 到另一个矩形区域	754	randbwr 随机块写	789
moveto 从当前坐标位置(CP)移到 (x,y)	754	random 随机数发生器	791
movmem 移动一长为 length 字节的串	755	randomize 初始化随机数发生器	791
normvideo 选择正常亮度字符	756	_read 读文件	792
nosound 关闭 PC 机扬声器	756	read 读文件	793
_open 打开一个文件进行读或写	757	real 返回复数的实部	795
open 打开一个文件进行读或写	758	realloc 重新分配内存	795
outport outp 输出一个字到端口中	760	rectangle 画一个矩形	796
outportb 输出一个字节到端口	760	registerbgidriver 注册已加载或连接进来的图形 驱动程序	797
outtext 显示一个字符串	761	registerbgifont 注册已连接进来的矢量 字体代码	798

remove	删除一个文件	800	方式	845	
rename	文件改名	800	settextstyle	为图形输出设置当前的文本	
restorecrtmode	恢复屏幕为调用 initgraph 前的设置	801	属性	847	
rewind	将文件指针重定位于流的开 始处	802	settime	设置系统时间	849
rmdir	删除目录	803	setusercharsize	修改矢量字体字母的宽度和 高度	850
_rotl	将一个无符号整数(unsigned)左 循环移位	804	setvbuf	使缓冲区与流相联	851
_rotr	将一个无符号整数向右 循环移位	805	setvect	设置中断矢量入口	852
sbrk	改变数据段地址	806	setverify	设置 DOS 中的校验标志状态	853
scanf	格式化输入	806	setviewport	为图形输出设置当前视口	854
searchpath	按 DOS 路径查找一个文件	813	setvisualpage	设置可见的图形页号	855
sector	画并填充椭圆扇区	814	setwritemode	设置图形方式下面线的输出 模式	856
segread	读段寄存器值	815	signal	设置某一信号的对应动作	858
setactivepage	设置图形输出活动页	816	sin	计算正弦值	861
setallpalette	改变所有的调色板颜色	817	sinh	计算双曲正弦值	861
setaspectratio	设置图形纵横比	819	sleep	执行挂起一段时间	862
setbkcolor	用调色板设置当前背景颜色	821	sopen	打开一共享文件	862
setblock	修改已分配的内存的大小	822	sound	按指定频率打开 PC 扬声器	864
setbuf	把缓冲区与流相联	823	spawnl, spawnl, spawnlp, spawnlpe, spawnv, spawnve, spawnvp, spawnvpe	创建并运行子 进程	865
setcbkr	设置 control-break	824	sprintf	送格式输出到字符串	867
setcolor	设置当前要画的线的颜色	825	sqrt	计算参数平方根的绝对值	868
setdate	设置 DOS 日期	827	srand	初始化随机数发生器	869
setdisk	设置当前驱动器	827	sscanf	从某串中扫描格式化输入	869
setdta	设置磁盘传输地址	828	stat	读取文件信息	871
setfillpattern	选择自定义的填充模式	830	_status87	取浮点状态	872
setfillstyle	设置填充模式和颜色	831	stime	设置系统日期和时间	873
setftime	取得文件日期和时间	833	stpcpy	拷贝字符串	874
setgraphbufsize	改变内部图形缓冲区的 大小	834	strcat	串连接	874
setgraphmode	将系统设置成图形模式并 清屏	835	strchr	搜索串中某个给定字符的第一次 出现	875
setjmp	非局部跳转(在 MS-WINDOWS 中不能 使用本功能)	837	strcmp	串比较	876
setlinestyle	设置当前画线宽度和类型	838	strcmpi	忽略大小写的串比较	876
setmem	设置内存	840	strcpy	串拷贝	877
setmode	设置打开文件方式	841	strcspn	搜索串中不包含给定字符集之子集 的第一个段	878
setpalette	改变调色板的颜色	841	strdup	复制串	878
setrgbpalatte	定义 IBM 8514 图形卡的 颜色	843	_strerror	建立用户定义的错误信息	879
settextjustify	为图形函数设置文本的对齐		strerror	返回指向错误信息字符串的 指针	879
			stricmp	忽略大小写的串比较	880

strlen	计算字符串的长度	881
strlwr	转换字符串中的大写字母为小写字母	881
strncat	把字符串的一部分附加到另一个串之后	882
strncmp	把串的一部分与另一个串的一部分进行比较	882
strncmpi	忽略大小写的串部分比较	883
strnset	将串中指定数目字节设置为字符	884
strpbrk	搜索给定集合中任一字符在串中的首次出现	884
strrchr	搜索给定字符在串中的最后一次出现	885
strrev	颠倒串中各字符的顺序	885
strset	设置串中所有字符为给定字符	886
strspn	搜索给定字符集的子集在串中第一次出现的段	886
strstr	搜索给定子串在某串中的出现位置	887
strtod	把串转换为双精度数值	887
strtok	搜索串中的某单词,该单词由第二个串中指定的符号进行分隔	888
strtol	转换串为长整型数	889
strtoul	将字符串转换为给定基数的无符号长整型值	890
swab	交换字节	891
system	执行 DOS 命令	891
tan	计算正切值	892
tanh	计算参数 x 的双曲正切值	892
tell	取文件指针的当前位置	893
textattr	设置文本属性	894
textbackground	选择文本的背景颜色	895
textcolor	选择文本模式的前景颜色	896
textheight	返回以像素为单位的字符串高度	897
textmode	将屏幕设置成文本模式	899
textwidth	返回以像素为单位的字符串宽度	900
time	取时间	901
tmpfile	以二进制方式打开临时文件	901
tmpnam	创建唯一的文件名	902
toascii	转换字符为 ASCII 格式	903

_tolower	转换字母为小写	903
tolower	转换字符为小写	904
_toupper	转换字母为大写	904
toupper	转换字符为大写	905
tirg	三角函数	906
tzset	设置全局变量 daylight、timezone 和 tzname 的值	906
ultoa	转换无符号长整型值为字符串	907
ungetc	把一个字符回退到输入流中	908
ungetch	把一个字符回送到键盘缓冲区	909
unixtodos	把 UNIX 格式的日期和时间转换成 DOS 格式	909
unlink	删除文件	910
unlock	解除文件共享锁	911
va_arg, va_end, va_start	实现可变参数表	912
vfprintf	送格式化输出到一流中	913
vfprintf	从流中搜索和格式化输入	914
vprintf	送格式化输出到 stdout	916
vscanf	从 stdin 中搜索和格式化输入	916
vsprintf	送格式化输出到串中	917
vsscanf	从流中搜索和格式化输入	918
wherex	给出窗口内光标水平位置	919
wherey	给出窗口内光标垂直位置	920
window	创建活动文本模式窗口	920
_write	写文件	921
write	写文件	922

第十八章 全局变量

_8087	协处理器芯片标志	924
_argc	保存命令行的参数个数	924
_argv	命令行参数指针数组	924
_ctype	字符属性信息数组	924
daylight	指示是否进行夏令时间调整	925
directvideo	视频输出控制的标志	925
environ	存取 DOS 环境变量	925
errno、_doserrno、sys_errlist、sys_nerr	使 perror 能打印错误信息	925
_fmode	设置缺省文件传送模式	927
_heaplen	保存近堆的长度	928
_openfd	存取模式数组	928
_osmajor、_osminor、_version	包含 DOS 版本的主号和次号	929

_psp	包含当前程序的程序段前缀 (PSP)的段地址	929
_stklen	保存堆栈的大小	929
timezone	包含当地时间与格林威治时间(GMT) 之间的差值(以秒为单位)	930
tzname	时区名指针数组	930
_version	DOS 版本号	930
_wscroll	指示控制台 I/O 函数是否滚屏	930

第五部分 附录

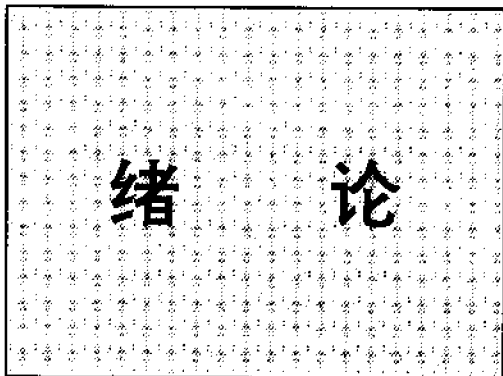
附录 A TCC 命令行编译器

A.1	使用命令行编译器	933
A.1.1	使用选择项	933
A.1.2	语法和文件名	935
A.1.3	应答文件	936
A.1.4	配置文件	936
A.2	编译器选项	937
A.2.1	存储模式	937
A.2.2	宏定义	938
A.2.3	代码生成选项	938
A.2.4	优化选择项	939
A.2.5	源代码选项	940
A.2.6	出错报告选择项	941
A.2.7	段命名控制	942
A.2.8	编译控制选择项	943
A.4	环境选项	943
A.4.1	查找包含文件和库文件	943
A.4.2	文件搜索算法	944
A.4.3	一个实例	944

附录 B 实用程序

B.1	MAKE 实用程序	946
B.1.1	MAKE 的工作过程	946
B.1.2	启动 MAKES	947
B.1.3	MAKE 的一种简单运用	948
B.1.4	制作 makefile 文件	950
B.1.5	makefile 文件的组成	950
B.1.6	命令表	950
B.1.7	显式规则	953

B.1.8	隐式规则	955
B.1.9	宏	956
B.1.10	指令	959
B.1.11	MAKE 出错信息	963
B.2	TLIB, 库管理程序	966
B.2.1	为什么使用目标模块库	967
B.2.2	TLIB 命令行	967
B.2.3	操作列表	968
B.2.4	使用响应文件	969
B.2.5	建立扩展字典:/E 选项	969
B.2.6	设置页大小:/P 选项	969
B.2.7	高级操作:/C 选项	969
B.2.8	例子	970
B.3	连接程序 TLINK	970
B.3.1	调用 TLINK	970
B.3.2	使用响应文件	972
B.3.3	和 Turbo C 模块一起使用 TLINK	973
B.3.4	与 TCC 一起使用 TLINK	974
B.3.5	连接选项	975
B.3.6	TLINK 的限制	977
B.3.7	出错信息	977
B.4	THELP 帮助	980
B.4.1	装入和调用 THELP	980
B.4.2	THELP 选项	981
B.5	GREP 查找程序	984
B.5.1	命令行形式	984
B.5.2	GREP 的选项	985
B.5.3	正常的优先次序	986
B.5.4	搜寻字符串	986
B.5.5	正则表达式的操作符	987
B.5.6	文件说明	987
B.5.7	GREP 使用示例	987
B.6	其它实用程序	990
B.6.1	BGI OBJ: 图形驱动程序 和字体的转换程序	991
B.6.2	CPP: 预处理实用程序	995
B.6.3	OBJXREF: 目标模块的 交叉引用列表实用程序	997
B.6.4	PRJCFG	1004
B.6.5	TOUCH	1004



1986 年 12 月,美国 Wizard Software System 公司推出了一个 C 语言编译器,称为 Wizard C。Wizard C 是一个颇受欢迎的 C 语言编译器,一般评论者称之为完美的 C 语言编译器。Wizard C 的优点是:快速的编译速度;有效率的代码优化;与 X3J11 委员会建议的 ANSI 标准兼容;提供开发中断服务子程序的许多 C 语言扩展功能等。这些扩展功能包括一个特别的中断函数类型、行内汇编语言,及允许 C 语言对微型机寄存器做访问动作的虚拟内存变量(Virtual memory variable)。

1987 年,Borland 公司推出 Turbo C。它是 Turbo Pascal 编译器的新伙伴。Borland 宣称 Turbo C 可用每分钟 7000 行的速度对程序进行编译。

在同一个月,Wizard 刊出最后一次的杂志广告,表示现在已到了 Turbo C 的时代了。

事实上,Borland 从 Wizard 那里学到 Turbo C 设计方法。当 Borland 公司想在 Turbo Pascal 及 Turbo Basic 之外开发出一个大众化的快速 C 语言编译器时,会详细考虑整个软件界的情况。与其重新开发出一个全新的编译器,Borland 公司做了一个明智的选择,买一个最好的 C 语言编译器,然后再改良这个编译器。

Borland 公司不再介绍一个令人厌烦的 C 语言编译器,决意为 C 语言的开发环境重定出一个新的模式。

0.1 两种 Turbo C 环境

Turbo C 是两个 C 编译器的集成:一人是通常 UNIX 格式的命令行(command line)编译器、连接器等组合,另一个是 Turbo C 集成开发环境(Integrated Development Environment 缩写为 IDE)。

在命令行编译环境中包括一个 MAKE 实用程序、一个 TCC 编译器以及目标文件管理程序 TLIB 等实用程序。命令行语言编译器与其他个人计算机上的 C 编译器类似,只是速度更快。所有有关命令行环境的任何信息都可以在章节和附录中找到。

集成开发环境是一个称为 TC 的程序。TC 集成了一个程序员的编辑器、一个联机面向工程的 MAKE 实用程序、一个程序运行实用程序,还包括一个源代码级的调试器。

这个操作环境是 Turbo C 的表示方式,功能最强大的地方是由于编辑器、编译器、连接器和调试器间的集成。在操作环境下,程序员可以编辑一个程序,进行编译工作,然后再与其他原始模块及函数库连接在一起以运行这个程序,并可利用其内部的调试器对程序的错误进行定位并帮助分析,富有成效地纠正程序的错误。这个特性显示出新一代 C 语言编译器

的能力。Turbo C 的操作环境与 Turbo Pascal 类似。至今为止并没有其他 C 编译器尝试模仿 Turbo C 的表示方式。

0.2 设置自己的任务环境

Turbo C 可以设置任务环境。从 Turbo C 系统运行时屏幕的颜色到错误检查的层次设置都可以修改。有些修改是在运行 TCINST 程序时设置任务, 其他的部分可以由在任务环境中以交互方式输入屏幕上方的弹出菜单中来设置。

下面是一些可在任务环境中自行设置的项目。

- 内存使用类型——微<tiny>、小<small>、中<medium>、紧凑<compact>、大<large>、或是巨大<huge>。
- 函数调用协定——C 语言或是 Pascal 语言协定。
- 所使用微型机——8088/8086 或 80186/80286。
- 浮点运算——有无协处理器, 或是利用软件模拟协处理器计算。
- 优化(optimization)的等级。
- 错误处理的等级。

除了上述这几项之外, 还有许多部分可做设置。Turbo C 可很精确地进行错误及警告之间的检查。读者可能希望修改颜色的设置。原先有三种预先设置以供选择的颜色集, 包含一个缺省颜色组合、一个蓝玉式颜色组合, 及一个紫红色颜色组合。不要为不喜欢这些颜色而发愁, TCINST 程序允许用户为每个不同环境部分设上不同的色彩及辉度。

0.3 Turbo C 编译器

Turbo C 编译器很类似 WordStar 字处理器中的非文本编辑模式。这种文字编辑器的结构已经普及在许多 Borland 公司的产品中。如果读者会使用 Wordstar, 也就会使用 Turbo C 的编译器。用户可以利用 TCINST 程序以依照自己的需要设置编辑器。用户可以改变原先窗口的大小及编辑者用的命令键。对于经常有机会使用到另一个编辑器的程序员会比较喜欢将 Turbo C 编译器设置成与其他编辑器类似, 以免要多花费时间去区别两种不同的编辑者命令键用法。不过用户对于编辑命令键的组合被当成任务环境的专用键, 所以这些键就不能当成编辑命令来使用。

这个编辑器的威力也许还不如一些特殊程序员编辑器那么强大, 不过对一般的编辑任务已经足够了。Turbo C 编辑器有一个能重新设置的 TAB 键, 缺省时每按一次可跳八个字符的空间。

Turbo C 自己拥有一个称为 TLINK 的连接实用程序。连接程序将从 C 语言、汇编语言, 及由其他语言编译所得到不同的目标文件连接成一个可运行的模块。由 Turbo C 所产生的目标文件可以与标准 DOS LINK 兼容, 所以 Turbo C 所产生的目标文件可以与其他包含汇编语言在内各种语言的目标函数库(object libraries)连接在一起。那为什么不用现成的 DOS LINK 程序而用 TLINK 呢? 主要的原因在于速度, TLINK 的速度比 DOS LINK 程序来得快。

在 TCC 命令行编译器设置中, Turbo C 连接程序是一个独立程序, 而在任务环境中则是集成在一起的, 当在 Turbo C 任务环境中创建一个运行程序时, 会自动将连接程序包含进来。

0.4 Turbo C 工程制作实用程序

在 Turbo C 命令行编译器中有一个传统式的制作实用程序, 这个实用程序与 UNIX 系统及其他个人计算机上的 C 语言编译器的制造实用程序类似。Turbo C 的任务环境有一个独有的功能: 可以将源文件、目标文件与一个已开发好的可执行文件连接在一起。就这方面有点类似传统的制作实用程序。不过在另一方面, 当集成到 Turbo C 任务环境, 使用一个称为“工程”式版本的 MAKE 实用程序时。这个 MAKE 实用程序比在原先编译器的命令行制作实用程序更容易看而且更容易了解。

在另一个工程文件中以一行一行方式列出制作出一个执行程序所需要的源文件名称。在每个模块的右边要加上源文件所需要的文件名称(如一些头文件)。这种依赖关系用括弧及逗号分隔来指明。下列是一个工程文件输入的例子。

```
myprogram (key. h , twindows. h)
```

Turbo C 自动进行依赖关系检查, 如果 myprogram. c 版本比 myprogram. obj 新时(也就是 myprogram. c 编译成 myprogram. obj 后又做了某些修改), 则重新再编译成 myprogram. obj。如果 myprogram. obj 的版本又比 myprogram. exe 新时, myprogram. obj 就再与适当的目标文件(依存储模式而定)与运行函数库再连接成 myprogram. exe。

当可执行程序由许多包含不同头文件的 C 源文件所组成时, 就更需要工程式制造公用开发, 来帮助这个程序。

用户可以在一个工程式制作实用程序中对目标文件及目标函数库命名。Turbo C 任务环境将这些目标文件自动包含进来, 而不会再尝试去编译; Turbo C 还会自动查找适当的函数库以解决外部函数调用的问题。

编译器及连接器的错误处理方式如下。当 Turbo C 环境开始创建一个工程时(project), 会将所有的错误信息及警告信息记录下来。当整个创建程序结束之后, 所有的错误及警告信息都会出现在一个窗口中, 源程序则显示在另一个窗口之中。使用都在主程序中依照所出现的错误信息做往前或往后的移动。Turbo C 任务环境会显示每个错误信息, 并同时 will 将错误指示信号移动到发生错误的地方。用户可以在经过这些错误部分时就做修正任务, 也可以随时重新开始。这个错误处理程序会自动知道在源程序中增加了程序行或删除程序行而自动调整光标移到错误部分时的位置。

0.5 低级功能的支持

Turbo C 包括一些对其他 C 语言不具移植性的扩展功能, 反映出继承 Wizard C 而来的特质; 这些扩展功能都是介绍本书所提到功能必须的。扩展功能主要是为提供中断处理子程序及其他硬件层次操作时所需要的功能。

Turbo C 语言中加入了中断函数类型。当中断函数输入并启动中断功能的数据段寄存

器(data segment register)以设置数据段时会产生一个远程函数(far function)储存 8086 寄存器的内容。当函数运行结束返回,这些寄存器的内容都要恢复。这类型的返回是 8086 IRET 指令使用在中断处理中返回原先运行部分所用。

由于增加了 asm 关键字,所以 Turbo C 提供了行内汇编语言码(in line assembly code)。用户必须先有 Microsoft 汇编语言汇编器才能使用到这种特性。任何在 asm 关键字后面的东西都送到 Microsoft 汇编器做处理。在汇编语言码所说明的范围中,行内汇编语言码可以使用到 C 程序中的变量名称。这个特性使得用户可以写一些与存储模式无关的汇编语言函数。写好的程序中,如果没有行内汇编语言码就不会送到汇编器中处理,而会直接编译。如果程序中有行内汇编语言码时,必须使用 tcc 命令行编译器来做编译任务。因为在 tc 任务环境中不能运行汇编语言编译器程序,在未来的 Turbo C 版本中将改进这个缺点。

有几个关键字被用来当成虚拟变量以直接访问硬件的寄存器时,就可以对程序做优化的处理。使用这种特性要小心,最好的方式是在 tcc 命令行编译指令加上 -S 特性将函数编译成汇编语言码后,再看看编译出来的代码中是否将需要的要求实现。当使用更新版本的 Turbo C 时,就必须回来再检查一次这些编译代码。Borland 公司可能会改变寄存器在编译码中的使用情形,这可能终止程序运行。

0.6 配置 Turbo C

Turbo C 所提供文件中最弱的地方可能就是设置。常有些是用户该知道的事情而在使用手册上却不能得到所需的信息。

命令行编译器或是 Turbo C 全局任务环境都会运行用户设置环境结构的文件。在使用手册中提到如何准备好命令行编译器的设置文件 TURBOC.CFG,不过并没有 Turbo C 任务环境设置文件 TCCONFIG.TC 那么清楚。在 Turbo C 任务环境安装好之后,用户需要再运行一次 TC 程序以创建自己的设置;这包括 tc 如何找到函数库,启动代码,包含文件(include file)及本身的路径设置。这些设置由 Turbo C 任务环境的选择菜单所设置。当每个部分都如需要的设置好之后,就可将 TCCONFIG.TC 保存起来。

0.7 存储模式

Turbo C 有六种存储模式:微、小、中、紧凑、大,及巨模式等。

第一部分

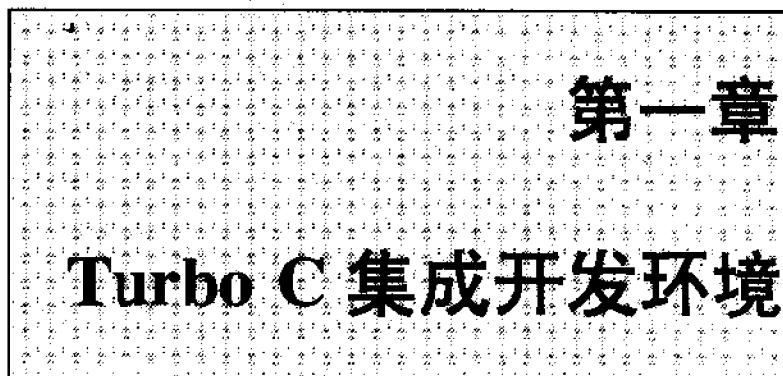
掌握 Turbo C

Turbo C 集成开发环境

Turbo C 编辑器

Turbo C 调试器

多文件工程管理



正如“绪言”中所介绍的那样, Turbo C 集成开发环境(TC)不仅仅是一个快速编译器, 还是一个速度快、编译效率高及自带编辑器、调试器和其它易学好用的实用程序的综合软件。有了 TC 之后, 用户就不再需要单独的编辑器、调试器、编译器、连接器以及 Make 实用程序, 就可以建立和运行 C 程序了。在 Turbo C 中, 所有这些功能都可通过一个简明的界面—— Turbo C 集成环境来访问。

本章介绍怎样使用 TC、菜单命令、配置文件及 PICK 文件。

在本章 TC 的使用章节中将介绍 TC 命令开关和热键, 描述 TC 主窗口的组成, 解释 TC 主屏幕选择的方法, 演示怎样进入编辑窗口和使用 TC 编辑器以及向用户介绍了 TC 集成调试器(有关怎样使用调试的详细信息, 参看第三章)。

在 TC 菜单命令的讨论中, 将介绍每个菜单选项的功能和编译选项。

在配置文件及 PICK 文件的说明, 将讨论什么是配置文件以及在 TC 里如何创建和使用配置文件, 在 TC 里如何创建和使用 PICK 文件。

象其它 Borland 产品一样, 只有按一个 F1 键, Turbo C 就能在屏幕上显示出上下文相关的帮助。在 TC 任何菜单、任何地方都可以获得帮助。

按 F1 键调用帮助系统。帮助窗口解释当前所在条目的功能。任何帮助屏幕可能包含一些关键字。可以选择以获得更多的信息, 用光标键移动光标到所需关键字, 按回车以获得更多的有关所选条目的详细信息。还可以分别使用 Home 键和 End 键来移到屏幕上最头和最尾的关键字上。

如果想回到前面的帮助屏幕, 只需按 ALT-F1, 而不论是不是在帮助系统里, TC 允许回溯 20 层帮助屏幕。

进入帮助系统之后再按一个 F1 即可得到帮助索引。

当在 TC 进行编辑的时候, 如何需要各种函数的帮助, 则可将光标移至需获得帮助的函数的名上, 按 CTRL-F1, 就可在屏幕上得到所需的信息。

按 ESC 键即可退出帮助系统回到原菜单。

1.1 TC 的使用

在 DOS 提示符下键入 TC, 回车, 即可加载 Turbo C 集成开发环境。

此后显示的屏幕包括主 TC 屏幕和含产品版本的框, 当击任一键后, 版本信息即消失, 要想再显示, 任何时候按 Shift-F10 即可, 主屏幕如图 1.1 所示。

仔细看一下 TC 主屏幕,由四部分组成:主菜单、编辑窗口、消息窗口和功能键提示行。

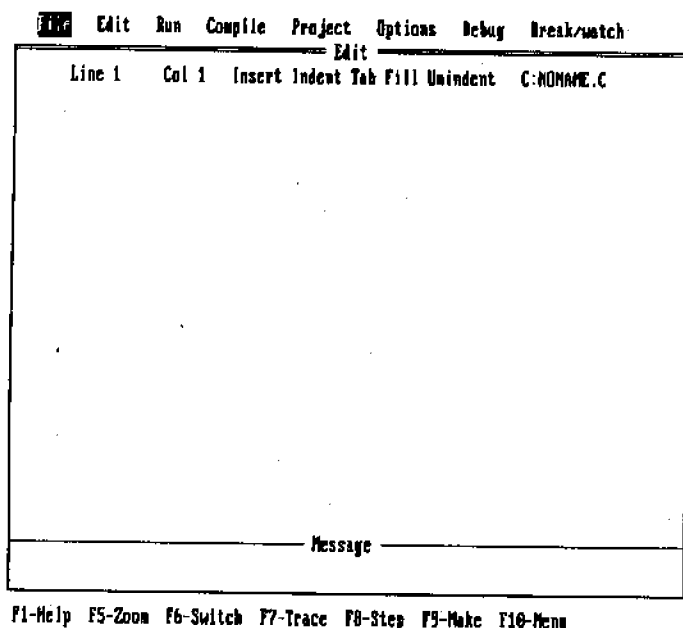


图 1.1 TC 主屏幕

1.1.1 TC 命令行开关

Turbo C 集成开发环境接收下面一些命令行开关:

- /c 开关加载配置文件。键入 TC 命令,后跟/c 配置文件名(/C 和文件名间不要留空格):

```
tc /cmyconfig. tc
```

- /b 开关使 TC 重新编辑 project 里的所有的文件,在标准输出设备上打印编译消息,返回 DOS。此开关允许在批处理文件里调用 TC,使 project 文件的建立自动化。建立之前,TC 将加载一个缺省的或由/c 开关指定的配置文件。TC 根据 project 文件,主文件或者当前装入 TC 编辑器里的文件的先后顺序决定生成什么样的 .EXE 文件。键入 TC 命令后跟一个单独的/b,或者用/c 加载配置文件后跟/b:

```
tc /cmyconfig. tc /b
```

除非加载的配置文件指定了 project 或主文件,可以在命令行指定要编译和连接的程序名。在 TC 命令后键入程序名后跟/b:

```
tc myprog /b
```

- /m 表示 make,而不是重新生成(也就是说让编译连接那些过时的文件)。用法与/b 一样。
- /d 开关使 TC 在检测到合适的硬件的时候使用双监视器方式。如果没有合适的硬件,/d 忽略。双监视模式是在运行或调试程序时,或使用 DOS 命令解释程序时使用的。

当在命令行给出 /d 开关后,只有提供所需硬件(如单色卡和图形卡)双监视模式才被启用;如果系统有两个监视器,DOS 只把其中的一个当作活动监视器。用 DOS 的 MODE 命令可以切换这两个监视器(例如,MODE C080,MODE MONO)。处于双监视模式时,TC 屏幕一般出现在不活动监视器上,而程序的输出则是到活动监视器上。这样,当在一个监视器上的 DOS 提示符下键入 tc /d,TC 将出现在另一个监视器上。当想在一特定的监视器上测试程序时,必须退了 TC,把活动监视器开关切换到想测试用的监视器上,然后再发一次 tc /d 命令,于是程序输出将在键入 TC 命令的那一个监视器上出现。

警告:

- 在 DOS 命令解释程序里不要改变当前活动监视器(例如使用 DOS MODE 命令)。
- 用户程序不能直接访问不活动监视器的视频口,否则会产生不可预料的结果。
- 当运行或调试显示使用双监视器的程序时,不得打开双监视器开关(/d)。

1.1.2 TC 内部命令的使用

为了更快地熟悉 Turbo C 集成开发环境,下面列出一些指导性的知识,在 TC 的任何地方都可以使用下面命令:

- 按 F1,获得有关当前位置的信息(运行,编译等的帮助)。
- 按 F5,放缩活动窗口。
- 按 F6,切换窗口
- 按 F10,在菜单与活动窗口相互切换。
- 按 Alt-F6,改变窗口里的内容(消息窗口和监视窗口来回选择,或者当前文件和前面的文件间切换)。
- 按 ALT 加主菜单命令的首字符(F、E、R、C、P、O、D 或 B),调用指定的命令。例如,在系统里的任何位置按 ALT-E,就进入编辑窗口;ALT-F 则进入文件菜单项。

1.1.2.1 菜单命令

可用以高亮度显示的大小写字母或利用光标键(和回车键)来选择菜单。

按 ESC,退出菜单。

在主菜单或通过主菜单调用的任意一个菜单里,按 ESC,直接返回活动窗口。活动是窗口,顶部有条线且名字是高亮度的窗口。

按 F10,从任意菜单级回到以前的活动窗口。

用左右光标键从一下拉菜单移到另一个下拉菜单。

1.1.2.2 退出 TC 返回 DOS

进入文件菜单选择 Quit(按 Q,或把高亮长条移到 Quit 上再按回车键)。如果未存当前工作文件就选择了 Quit,编辑器将询问存不存(也可有 ALT-X 退出返回 DOS)。

1.1.3 TC 热键

在描述各种可用的选择项之前,想让读者了解一些热键。热键是为执行菜单功能而设的键。例如,正如前面所讨论的,ALT 加主菜单命令的首字母将可进入特定的菜单,或者完成一动作。另外还有一个 ALT/首字母命令就是 ALT-X,它是一个 File|Quit 的热键。

除了这些 ALT 首字母命令外,还特设了一个用户屏幕热键,ALT-F5,可用它来进行 TC 主屏幕与程序输出其上的用户屏幕间转换,等价于 Run 菜单上的菜单命令 User Screen。使用 TC 时,可见两个屏幕的一个——TC 主屏幕或者用户屏幕。TC 主屏幕是编辑、编译、连接和调试程序时所见的屏幕;用户屏幕是当运行一个 Turbo C 可执行程序或通过用 File|OS Shell 菜单命令临时退到 DOS 时所见的屏幕。使用集成调试器时,可能会经常进行 TC 主屏幕与用户屏幕间的转换,TC 可以连续地将用户屏幕的内容保存到用户屏幕缓冲区里,每次当选择了运行命令(象 Run, Trace Into 或 Step Over)或 File OS Shell 时都会更新它。从 Run 菜单里选择 User Screen 或按 ALT-F5 可观察该存储屏幕。

注意:在双监视模式下,用户屏幕即系统中两个监视器中的一个,因此,Run|User Screen 命令及 ALT-F5 不能使用。

TC 是根据视频模式来决定是否清除用户屏幕的。当从 DOS 调用 TC 或从 DOS 命令解释程序调用 TC 时,它就把视频模式和光标类型记忆起来,任何时候当通过 File|OS Shell 进入 DOS 或退出集成开发环境时(File|Quit),只要当前状态与记忆态有所不同,那两个状态或其中之一将被重新设置。有一例外:如果在调试阶段(程序运行时)键入了 DOS 命令解释程序,视频模式和光标类型是进入时的状态。

表 1.1 列出了 TC 里可以使用的所有热键。记住,不论在 TC 集成开发环境的任何地方,只要一按这些键,相应的功能即被调用。但有一例外:当要求按指定键的提示出现时,必须按该键后。热键才起作用。

表 1.1 Turbo C 热键

热 键	功 能
F1	激发帮助窗口,提供有关当前位置的信息
F2	编辑器里的文件存盘
F3	加载文件(出现输入盘)
F4	程序运行到光标所在行
F5	放大、缩小活动窗口
F6	开关活动窗口
F7	在调试模式下运行程序,跟踪进函数内部
F8	在调试模式下运行程序,跳过函数调用
F9	执行“Make”
CTRL-F1	调用有关函数的上下文帮助(仅限于 TC 编辑器)
CTRL-F2	重启运行程序
CTRL-F3	显示调用栈
CTRL-F4	计算表达式
CTRL-F7	增加监视表达式

CTRL-F8	断点开关
CTRL-F9	运行开关
ALT-F1	显示上次访问的帮助
ALT-F3	选择文件加载
ALT-F6	开关活动窗口里的内容
ALT-F7	定位下一错误
ALT-F8	定位下一错误
ALT-F9	把 TC 编辑器里的文件编译成 OBJ 文件
ALT-B	转到 Break/Watch 菜单
ALT-C	转到 Compile 菜单
ALT-D	转到 Debug 菜单
ALT-E	转到 Edit 菜单
ALT-F	转到 File 菜单
ALT-O	转到 Option 菜单
ALT-P	转到 Project 菜单
ALT-R	转到 RUN 菜单
ALT-X	退出 TC, 返回 DOS

1.1.4 菜单结构

TC 菜单上的选项分为三大类:命令、开关和设置。

命令: 执行任务(运行、编辑、存储选择等)。

切换: 设置开关 On/Off(自动检查依赖关系—Autodependence, 栈溢出测试—Test Stack overflow 等), 或者让用户通过连续按回车键来选择。从几个选项中选择所需条目(如 Message Tracing 或 Floating Point)。

设置: 允许用户定义编译时和运行所需的信息, 如目录查找、文件名、宏定义等。

1.1.5 主菜单

TC 主屏幕顶部提供了主菜单条(见图 1.2)。它提供了八种选择:

File	处理文件(装入、存盘、选择、建立、换名和写盘), 目录操作(列表、改变工作目录), 退出程序及调用 DOS。
Edit	建立、编辑源文件。
Run	控制运行程序。如果已编译连接好程序且 Debug Source Debugging 及 Options Compile Code Generation OBJ Debug Information 开关置为 On, 那么也可用此菜单启动调试过程。
Compile	编译、生成目标及可执行文件。
Project	允许说明程序里包含哪些文件(project)。
Options	可选择编译器选项(如存储模型、编译时选项、诊断及连接选项)及定义宏; 也可以记录 Include、Output 及 Library 文件目录, 保存编译选项和从配置文件加载选项。

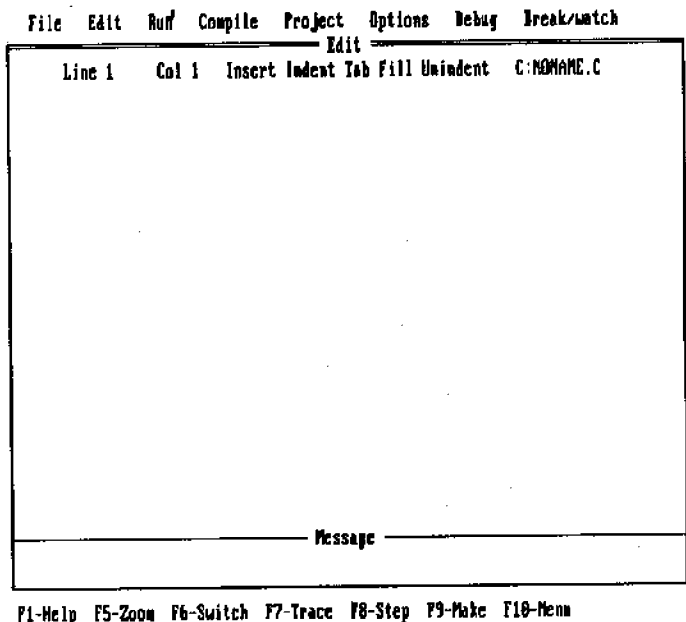


图 1.2 TC 的主菜单条

- Debug** 检查、改变变量的值,查找函数,程序运行时查看调用栈。选择程序编译时是否在执行行代码中插入调试信息。
- Break/Watch** 增加、删除、编辑监视表达式,及设置、清除、执行至断点。

注意:还有一个主菜单条目是命令:Edit,它仅仅是进入编辑器,其它菜单条目则是下拉一些有许多选项及/子菜单的菜单。

1.1.6 功能键提示行

对于任一窗口和菜单,屏幕底端都有一缺省的功能键提示行,它提供了当前状态下的功能键提示。

首次键入 TC 后,缺省的功能键行看上去是这样的:

F1-Help F5-Zoom F6 Switch F7-Trace F8-Step F9-Make F10-Menu

按下 ALT 键保持几秒钟,提示行将描述 ALT 键与哪些键连用时执行什么功能。如下:

Alt:F1-Lasthelp F3-Pick F6-Swap F1/F8-Prev/Next Error F9-Compile

1.1.7 编辑窗口

本节中,我们将描述 TC 主屏内的各个组成部分,解释怎样使用 TC 编辑窗口。

首先,为了进入编辑窗口,按 F10 转到主菜单,然后将光标键移到 Edit 再回车,或在主菜单的任何位置直接按 E。按 ALT-E 无条件进入编辑窗口。注意,一旦进入编辑窗口后,其顶部有双线且名字是高亮度的——即表示它是活动窗口,如图 1.3 所示。

除了可以看见编辑源文件的窗口外,还有两个信息项应当注意:状态行和功能键提示行。

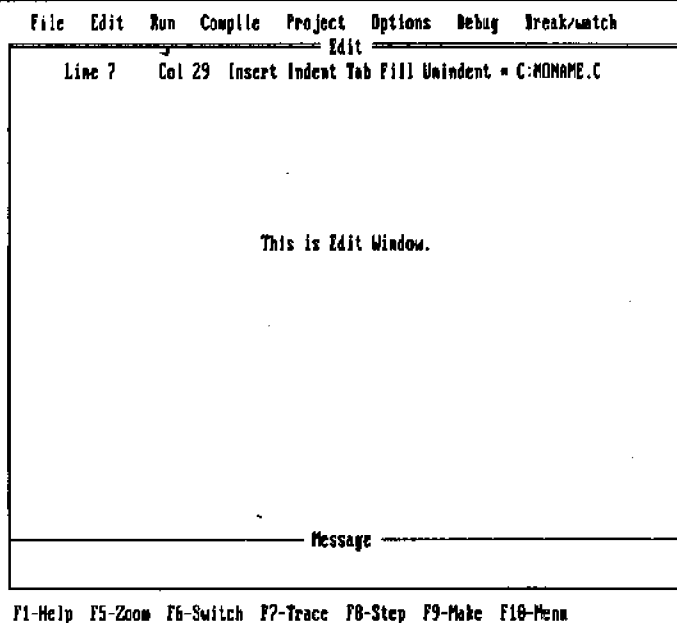


图 1.3 边框是双线的窗口是编辑窗口

TC 主屏顶端的状态行给出有关正在被编辑的文件的信息,光标在文件中定位何处,激发的是什么编辑模式,如下:

Line n	光标处在文件的第 n 行
Col n	光标处在文件的第 n 列
Insert	插入模式开关,用 Insert 或 CTRL-V 切换模式开关(On/Off)。
Indent	自动缩进开关,用 CTRL-OL 切换。
TAB	制表开关开启,用 CTRL-OT 切换
Fill	当 Tab 模式是 On 时,编辑器将用制表及空格符优化每一行,用 CTRL-OF 切换。
Unindent	当光标在一行中的第一个非空字符上时,或在空行上时,退格键回退一级。用 CTRL-OU 可切换开关。

星号是在文件被修改后,而又未存盘才出现在文件名前面的。C:FILENAME.TXT 驱动器(C:),名字(FILENAME),及正在编辑的文件的扩展名。

TC 屏幕底端所显示的功能键提示行指出热键所代表的动作。按指示键选择其中的一个功能:

F1	打开一个帮助窗口,给出有关 TC 编辑命令的信息。
F5	扩大活动窗口(这里的指编辑窗口)到整个屏幕。再按一次 F5 又回到分屏式环境。
F6	从一个活动窗口切换到另一活动窗口(Edit 窗口、Message 窗口和 Watch)。
F7-Trace	在调试模式下,一次执行程序的一行,跟踪到函数调用。
F8-Step	在调试模式下,一次执行程序的一行,跳过函数调用,即不跟踪进函数内部。

F9-Make Make(编译连接).exe 文件。

F10-Menu 从编辑窗口转到主菜单,以及从任何菜单转到编辑窗口。

TC 编辑器的命令结构类似于 SideKick 的 Notepad 和 Turbo Pascal 编辑器,如果用户对这些产品所使用的编辑器不熟悉,下面列出的是最常使用的命令。

编辑窗口中插入(Insert)模式下输入代码时,可用回车键来结束一行(TC 编辑器不自动换行)。最大行宽为 248 个字符,编辑器窗口宽 77 列,如打过 77 列,窗口随着字符的键入滚动。TC 屏状态行告诉光标在文件中的行和列。

在编辑窗口中输完代码之后,按 F10 转到主菜单,文件还在屏幕上,在主菜单中只要按 E(Edit)即可回到编辑窗口。

1.1.7.1 编辑命令摘要

下面是最常用的编辑命令:

- Up/Down、Left/Right 和 PgUp/PgDn 键滚动正文。
- CTRL-Y 删除一行。
- CTRL-T 删除一个单词。
- CTRL-KB 设置(开始)
- CTRL-B 设置(结尾)块标志。
- CTRL-KV 块移动。
- CTRL-KC 块拷贝
- CTRL-KY 块删除

1.1.7.2 编辑窗口里如何操作源文件

如果在调用编辑窗口之前未装入文件,那么 TC 编辑器将自动命名为 NONAME.C,这时编辑器具有一切特征,可以:

- 建立名为 NONAME.C 或别的名字的新源文件。
- 装入编辑器一已存在的文件。
- 从源文件列表中选择一个文件装入窗口进行编辑。
- 保存编辑窗口中的文件。
- 把编辑器里的文件写入一个新文件。
- 消息窗口和编辑窗口之间进行切换。

创建和编辑源文件,但还未编译时,勿需消息窗口,所以可用 F5 把编辑窗口扩大到整个屏幕,再按 F5 还原(回到分屏模式)。

创建源文件

可用下面两种方法建立文件:

- 主菜单中,选 File|New,再按回车键,可打开编辑窗口,文件名为 NONAME.C。
- 主菜单中,选 File|Load,Load File Name 提示框打开。键入新源文件名(任何时候按热键 F3 也可达到相同目的)。

File|Load 或 File|Pick 可装入编辑一已存在的文件。

主菜单中选 File|Load 后,可以:

- 键入想编辑文件的名字,例如,路径可以接受,C:\TURBOC\TESTFILE.C。
- 在 Load File Name 提示框中打入通匹符(用 DOS 中 * 和 ? 匹配符),然后按回车

键。*.* 显示当前目录中的所有文件和其它子目录。目录名后跟反斜杠(\)。选择目录将显示其中的文件。例如,键入 C:*.*, 只显示根目录中没有扩展名的文件。

按 Up/Down 及 Left/Right 光标键把高亮长条移到所需文件的名字上,回车,即可装入选择的文件,当前的状态是编辑状态。

还有一个热键可重装入(加载)先前装入的文件,按 ALT-F6(改变窗口内容)切换当前编辑器里的文件与前次加载的文件。

源文件存盘

- 系统中任何时候,按 F2
- 主菜单中,选择 File|Save

写输出文件

可以把编辑器里的文件写到一个新文件,也可重写一个已存在文件。既可以写到当前(缺省)目录,也可以写到另外一个驱动器和目录里。

主菜单中,选 File|Write To,然后在新名字提示窗口中,键入新文件全名,例如:

C:\dir\subdir\filename. ext

再按回车键。

如文件已存在,那么编辑器在执行写操作之前先要确认是否要真的重写。

按 ESC 返回活动窗口(编辑窗口)。也可用 ALT-E 或 F10。

1.1.8 消息窗口

编译和调试源程序时都需要通过消息窗口(Message Window)来察看诊断消息。TC 唯一的错误跟踪机制把所有编译文件的警告和错误都在消息窗口中了,同时在编辑窗口中指出错误在源文件中的相应位置(依 Option|Environment 菜单中的 Message Tracking 命令设置而定)。

光标在消息窗口时,功能热键执行下面的功能:

- | | |
|-----------|----------------------------|
| F1-Help | 打开一个帮助窗口,概述 TC 错误跟踪的特点。 |
| F5-Zoom | 把消息窗口扩至整屏。 |
| F6-Switch | 激活编辑窗口。 |
| F7-Step | 在源文件调试模式下允许一次执行一行。跟踪进函数。 |
| F8-Make | Make. EXE 文件。 |
| F10-Menu | 从活动窗口中转到主菜单,或从任一菜单中转到活动窗口。 |

1.1.9 监视窗口

当用集成开发环境的调试器运行程序时,监视窗口即取代消息窗口。其中包含监视表达式(从程序中插入监视窗口的表达式)及每个表达式的当前值,因为其值可能会发生变化,监视窗口给用户提供了跟踪程序运行时一些重要表达式值的手段。

表达式加进监视窗口,会引起窗口的扩大,直到由 TCINST Resize Window 说明的大小为止。之后,用户还可向里添加表达式,不过这时只有通过(PgUp、PgDn、Up 和 Down 光标键)滚动窗口的办法才能观看全部表达式了。

窗口活动时监视窗口中的当前表达式用高亮条来标记;而不活动时,左边用点标记。

用户可用编辑窗口中所用的编辑命令来编辑监视窗口中的表达式。例如,CTRL-Y 删除一表达式,CTRL-N 插入一表达式,基本监视窗口的编辑命令列在表 1.2 中。

表 1.2 监视窗口编辑命令

热 键	功 能
CTRL-E 或向上箭头	光标上移
CTRL-X 或向下箭头	光标下移
CTRL-S 或向左箭头	向左滚动监视表达式
CTRL-D 或向右箭头	向右滚动监视表达式
回车键	编辑监视表达式
CTRL-N 或 Ins	插入监视表达式
CTRL-Y, Del, CTRL-G	删除监视表达式
光标在监视窗口时,按热键执行下列功能:	
热 键	功 能
F1	打开帮助窗口
F5	监视窗口扩至整屏
F6	激活编辑窗口
Ins	向监视窗口添加一监视表达式
Del	从监视窗口删除一监视表达式
回车键	编辑监视窗口中的监视表达式

1.1.10 集成调试器

Turbo C 集成开发环境内含一帮助查找程序中错误的集成调试器,在第三章我们对它 will 作详细说明。本章只作一简要的介绍。

调试器能够使程序在任何地方停止运行,以使用户检查、修改变量的值。

1.1.10.1 调试器的控制

对于要调试的程序,在编译时必须置 Options | Compile | Code Generation | OBJ Debug Information 和 Debug | Source Debugging 开关为 On。运行程序时,调试器自动对程序进行调试。

当用 Run | Run 打开一调试节对话时,Turbo C 编译、连接必要的源文件,然后将它投入运行,直到程序断点或末尾。

若未设置断点,按 F8(Run | Step Over)即可开始一调试对话。调试器将停止在 main 开头。

Turbo C 一旦准备好程序运行后即进入调试对话,从这时开始便可使用 Turbo C 的其它功能。

可用下述方法运行程序:

- 逐行执行,既可跳过函数,也可进入函数内部。
- 从当前位置执行到预先设置好的断点处。

● 从当前位置执行到任意光标所在处。

这几项功能可单独使用,也可按任意顺序组合起来。

修改完正在调试的程序的源文件之后,请不要立即进行调试。最好先用 Compile|Make EXE File 将程序重新编译。事实上,对源文件作了修改之后,Turbo C 将在你使用 Step Over 或 Trace Into 命令时提示是否要重新生成 EXE 文件。

1.1.10.2 调试器的屏幕显示

调试器的屏幕由顶部的编辑窗口和底部的监视窗口组成。按 F6 可切换这两个窗口。

随着监视窗口中的监视表达式的增加,它长到最大尺寸(由 TCINST 的 Resize Windows 选择项指定)后便开始滚动。

程序的当前位置又叫执行位置,它由编辑窗口中的执行长条(高亮条)标识。

1.1.10.3 调试菜单命令及热键

表 1.3 列出了几个特别的调试的命令。

表 1.3 调试命令及热键

F4 热键	菜单命令	说 明
F4	Run Goto Cursor	运行程序到光标所在行,将初始化的一调试节。
CTRL-F2	Run Program Reset	结束当前调试节,释放分配的内存,关闭所有文件。仅在调试节有效。
F7	Run Trace Into	运行当前函数中的下一条语句。若遇到更低一级的函数调用,而该函数编译时 Options Compile Code Generation OBJ Debug Information 开关为 On,则跟踪进函数内部。将初始化一调试对话。
F8	Run Step Over	运行当前函数中的下一条语句。不跟踪进函数。将初始化一调试对话。
	Options Compile C Standard Stack Frame	开关 Options Compiler Code Generation Standard Stack Frame 选项。如要想 Debug Call Stack 选项工作正确,该项必须置为 On
	Options Compile Code Generation OBJ Debug Inforation	开关 Options Compile C OBJ Debug Information 选项,只有此项置为 On 时编译连接的源文件才能进行调试
CTRL-F4	Debug Evaluate	计算 C 表达式。允许用户修改变量的值。
	Debug Find Function	查找函数的定义,并显示在编辑窗口中。仅限于调试阶段
CTRL-F3	Debug Call Stack	显示调用栈,从调用栈中选择函数名可显示当执行行。仅限于调试阶段。
	Debug Source Debugging	控制调试允许,置为 On 时,集成调试器和独立的调试器均允许;但置为 alone 时,只能用单独的调试器调试,当然,还是可以在 TC 里运行;置为 None 时,调试信息未被置入 EXE 文件,这时两种调试器都不能用。
CTRL-F7	Break Watch Add Watch	增加一监视表达式

	Break Watch Delete Watch	删除一监视表达式
	Break Watch Edit Watch	编辑监视表达式
	Break Watch Remove All Watches	删除所有监视表达式
CTRL-F8	Break Watch Toggle Breakpoint	设置或去掉光标所在断点
	Break Watch Clear Breakpoint	清除程序所有断点
	Break Watch View Next Breakpoint	显示下一断点

表 1.4 列出了其它一些调试时常用的菜单命令。

表 1.4 菜单命令和调试热键

热键	菜单命令	功 能
F5		放缩活动窗口
ALT-F5		显示转入用户屏, 击任意键返回集成环境屏
F6		循环激活编辑、监视和消息窗口
ALT-F6		若编辑窗口是活动的, 则转到上次装入编辑器的文件, 若下面的那个窗口是活动的, 则进行监视窗口与消息窗口间的切换
CTRL-F9	Run Run	运行程序, 有或没有调试器, 必要时将编译、连接源文件, 如经编译了, 而连接时 Debug Source Debugging 和 Options Compile OBJ Debug Information 又置为 On, 则程序运行至断点或运行完。
	Project Remove Messages	删除消息窗口中的内容

1.2 菜单命令

主菜单中包含一些用户主要使用的选项: 装入、编辑、编译、连接、调试及运行 Turbo C 程序。这八个菜单选择包括: File、Edit、Run、Compile、Project、Options、Debug 和 Break/Watch。这里给出它们的详细描述。

注意: 本章所涉及的“make”指用 project 文件来组装 EXE, 不是单独的 MAKE 实用程序。

1.2.1 File(文件)菜单

文件下拉菜单提供了装入已存在文件; 建立编辑器; 然后, 可将其存入任何一个目录任何一个文件名。另外, 通过文件菜单可以改变目录; 暂时退到命令解释程序, 或者退出 Turbo C。

Load(加载)

装入一个文件, 可用类似 DOS 的匹配符(如 *.C)来进行列表选择, 也可装入某一个文件, 只要给出其文件名即可。

注意:若驱动器或路径给错了,屏幕将出现一错误框。确认框是在未存一个修改过的文件前又加载另一文件的情况下出现的。不论哪种情况,只有敲了所需要键之后,热键才起作用。

Pick(选择)

将最近装进编辑窗口的 8 个文件列成一个表,让用户选择其一,选择后又装入编辑器,光标置在上次修改过的地方。若选了“---Load file---”条目,屏幕上将出现 Load FileName 提示框,就好象选择了 File|Load 或按 F3 了一样。ALT-F3 是 File|Pick 的另一种方法,只要建立了 Pick 文件,集成开发环境即可记住那些编辑过的文件。有关建立 Pick 文件的详细说明参见 Options|Directories|Pick File Name。

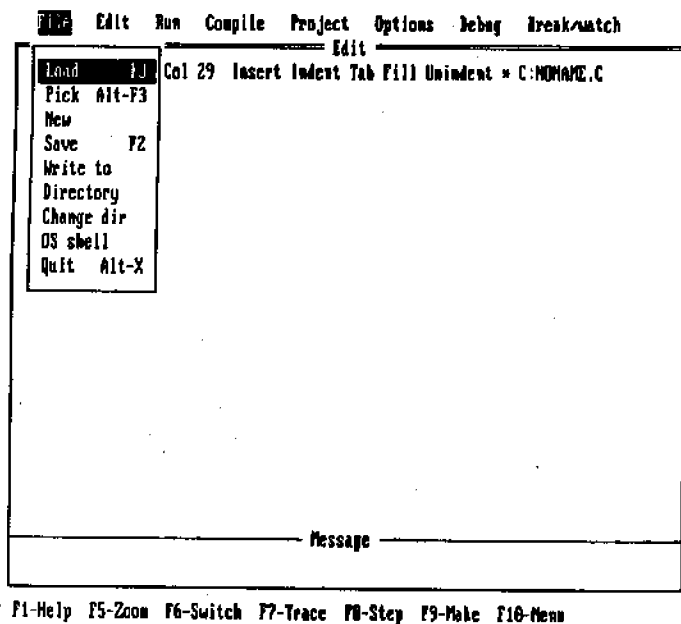


图 1.4 File 菜单

New(新文件)

说明欲建立的文件是新的,加载进编辑器,缺省文件名为 NONAME.C。以后存文件时还可改名。

Save(存盘)

编辑器里的文件存盘。若文件名是 NONAME.C,而又要存盘,编辑器就会询问是否要改名。热键 F2 任何时候均可存盘。

Write To(存盘)

给出文件名,把编辑器里的内容写入其中;若该文件已存在,则重写。

Directory(目录)

显示目录及所需的文件列表(按回车键选择当前目录)。F4 改变匹配符,选择文件名将文件装入编辑器。

Change Dir(改变驱动器)

显示当前目录,改变驱动器及目录。

OS Shell(暂时退出)

暂时退出 Turbo C,转到 DOS 提示符下,EXIT 返回 Turbo C。此命令在想运行 DOS 命令而又想退出 Turbo C 时是非常有用的。

注意:在双监视模式下,DOS 命令解释程序出现在 TC 屏幕而不是用户屏幕上。这样,当退出 Turbo C 到 DOS 时不会影响程序的输出。由于程序输出也可以在系统中的某一个监视器外,所以 Run|User Screen 及 ALT-F5 不起使用。

Quit(退出)

退出 Turbo C,返回到 DOS 提示符下。

相应热键是 ALT-X。

1.2.2 Edit(编辑)命令

编辑命令调用内部编辑器。编辑器中按 F10 可返回主菜单(或用 ALT 加所需主菜单命令的首字母),这时仍然保持在屏幕上,主菜单中按 ESC 或 E,即可回到编辑器(按 ALT-E 也可,且任何时候都起作用)。

1.2.3 Run(运行)菜单

运行菜单命令运行程序——开始和结束调试。除了 Run|Run 之外其它命令都必须在 Debug|Source Debugging 开关置为 On 时,编译连接完程序之后才能用。

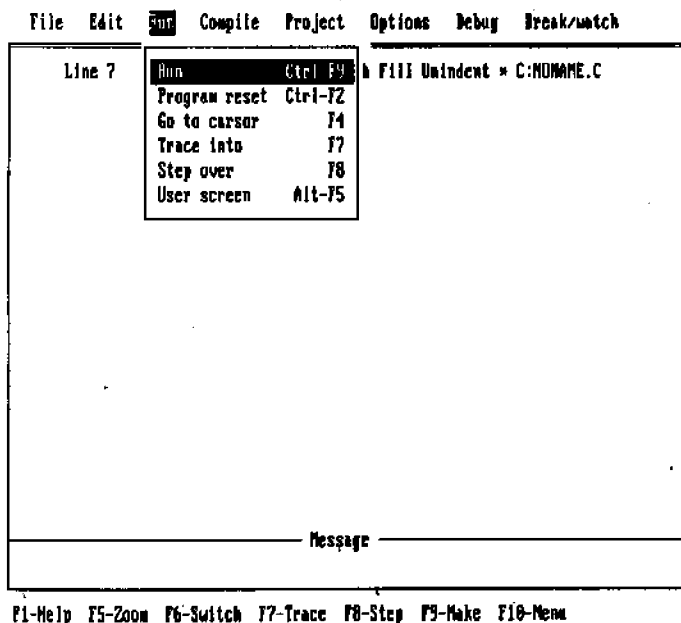


图 1.5 Run 菜单

Run(运行)

Run|Run 运行程序可用 Options|Argument 传递参数。若自上次编译后源代码有变动,则启用“make”来重编译、连接程序。“make”是集成开发环境里的一程序开发工具。

若不想调试程序,编译连接时 Debug|Source Debugging 开关应置为 None 或

Standalone。否则,产生的可执行代码中将包含调试信息,在下面几个方面对程序的效果有所影响:

若上次编译后未修改过源代码:

- 若设有断点,则 Run|Run 使程序运行到下一断点;否则运行到底。

若上次编译后修改过源代码:

- 若已通过 Run|Step Over(F8)或 Run|Trace Into(F7)单步执行程序,Run|Run 将在屏幕上询问是否要重新 make 程序。
- 按 Y,将重新编译连接程序,再从头开始运行。
- 按 N,则运行至下一断点,或运行到底(若无断点)。

若不是在单步执行,则在重新编译连接程序之后,从头开始运行。

Run|Run 的热键为 CTRL-F9

Program Reset(程序重启)

Run|Program Reset 中止当前调试,释放分给程序的空间,关闭已打开文件。

Run|Program Reset 的热键是 CTRL-F2。

Go to Cursor(执行到光标)

Run|Go To Cursor 使程序从执行长条运行到编辑窗口中光标所在行。若光标所在行不含可执行代码语句,则显示一个 ESC 框作警告。Run|Go To Cursor 也可初始化调试对话。

Go To Cursor 并不设置永久性的断点。不过,如果在光标所在行前遇到断点就允许程序停止,发生这种情况时,必须再用 Go To Cursor 命令。

可用 Go To Cursor 把光标移到需要调试的部分。若要每次都在某一语句停一下,就设一个断点好了。

Trace Into(跟踪)

Run|Trace Into 运行当前函数里的下一条语句,若此语句不含调试器可访问的函数调用,Trace Into 停在下一条可执行语句上。

若语句含有调试器可访问的函数调用,Trace Into 就停在函数定义的开始,以后的 Trace Into 或 Step Over 命令就运行在定义的函数里。调试器离开函数时,再恢复到函数调用后的那条语句。

函数可访问的条件是,它定义在某一源文件里而此源文件编译时,Options|Compile|Code Generation|OBJ Debug Information 和 Debug|Source Debugging 开关置为 On,且调试器可在盘上找到其该源文件。

Step Over(步进)

Run|Step Over 执行当前函数的下一条语句,即使遇到调试器可访问的函数调用前会跟踪进下一级函数里。

用 Step Over 运行正在调试的函数,一次一条语句。

这里举例说明一下 Run|Trace Into 和 Run|Step Over 的区别,下面是装入编辑器的前 11 行的程序。

```
int findout(void)
{
    return(2)
```

```

}
void main(void)
{
    int i,j;           /* 第 7 行 */
    i=findout();       /* 第 8 行 */
    printf("%d\n",i);  /* 第 9 行 */
    j=0;              /* 第 10 行 */
}

```

findout 是模块中与编译信息一起编译的用户定义函数,假设执行长条位于程序的第 8 行上。

● 若用 Run|Trace Into,执行长条将进入 findit 函数的第一条(也是本程序的第一行),用户即可单步执行函数。

● 若用 Run|Step Over,findit 执行后返回值赋给 i,执行长条紧接着移至第 9 行。执行长条位于第 9 行时,两种命令就没有区别了;Run|Trace Into 和 Run|Step Over 都是执行完 printf 函数之后,把执行长条移到第 10 行,这是因为 printf 函数不含调试信息。

Run|Step Over 热键是 F8。

1.2.4 Compile(编译)菜单

可用编译菜单里的选项来将源文件编译成 OBJ 文件(Compile to OBJ);生成 .EXE 文件(Link EXE File);重建(Build All);设置 Primary C File;运行或显示上次编译的信息(Get Info)。

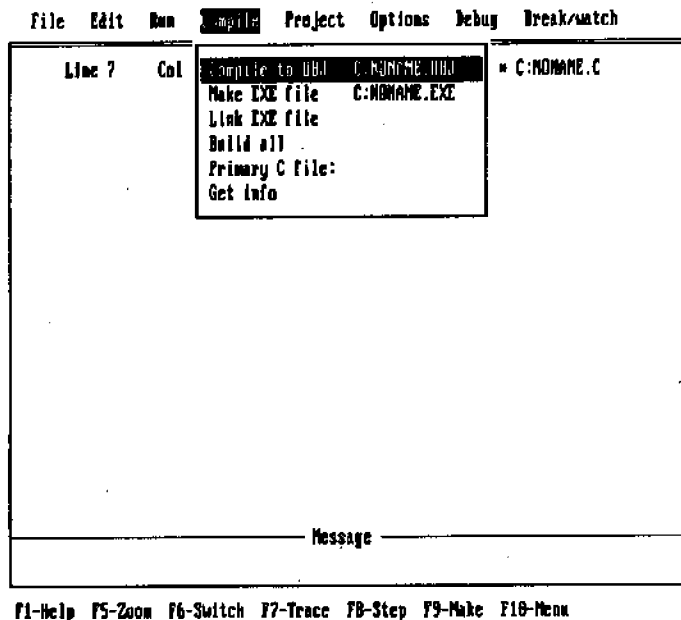


图 1.6 Compile 菜单

Compile to OBJ 编译生成目标码

此命令将一.C源文件编译成.OBJ文件,同时显示生成的文件的名字,例如 C:\EXAMPLE.OBJ..OBJ文件名由以下两种方法依序产生:

- 源.C文件名;
- 如没说明时,上次装入编辑窗口的文件名。

Turbo C 编译时,弹出一窗口,显示编译结果。编译或 Make 完之后,击任一键去掉编译窗口。若发现错,则转到消息窗口的第一条错误上(高亮度标志)。

此命令热键为 ALT-F9。

Make EXE File 生成执行文件

此命令调用工程式 Make 来生成一个.EXE文件,显示生成的.EXE文件名,例如, C:\EXAMPLE.EXE

列出的.EXE文件名字是用下面的几种方法产生的:

- 由 Project|Project Name 说明的工程的文件名,
- 否则,由 Primary C File 说明的源文件名,
- 否则,上次装入窗口的文件名。

此命令热键为 F9。

Link EXE File 连接执行文件

把当前.OBJ文件及库文件(既可以是缺省的,也可以是定义在当前工程 project 文件里的)连接在一起,生成.EXE文件。不进行过时检查。

Build All 建立所有文件

重建工程里的所有文件,不论过时否。此命令类似 Compile|Make EXE File,只是它是无条件执行的;Compile|Make EXE File 只重建那些非过时文件。此命令首先将所有的 project 文件里的.OBJ文件的日期、时间置为 0,然后再 Make。这样,若用户因 Ctrl-Break 键中断了 Build All 命令,那么只要用 Compile|Make EXE File 即可继续运行。

Primary C File 主 C 文件

当编译含多个.H头文件的单个.C文件时,Primary C File 命令是十分有用的(但并非必要)。若编译过程中发现错误,含错文件(.C文件.H文件)将自动装入编辑器,可对其修改(注意,H文件只有在用户已将 Options|Environment|Message Tracking 缺省设置改为 All Files 进才自动装入,原缺省设置不会自动加载.H文件)。即使不在编辑器,只有按 ALT-F9,.C主文件即被重新编译。

Get Info 获得信息

Compile|Get Info 打开一个窗口给出如下信息:

- 源文件
- 与当前文件相联系的目标文件名
- 当前源文件名
- 当前源文件的字节大小
- 程序退出码
- 可用空间

1.2.5 Project 菜单

通过 Project 菜单上的命令可以将多个源文件及目标文件合起来生成最后的程序。

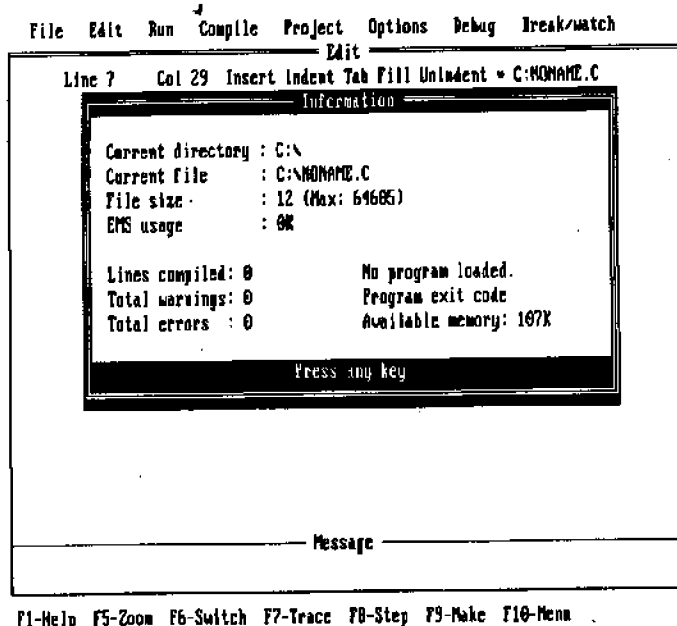


图 1.7 Compile|Get Info 屏幕

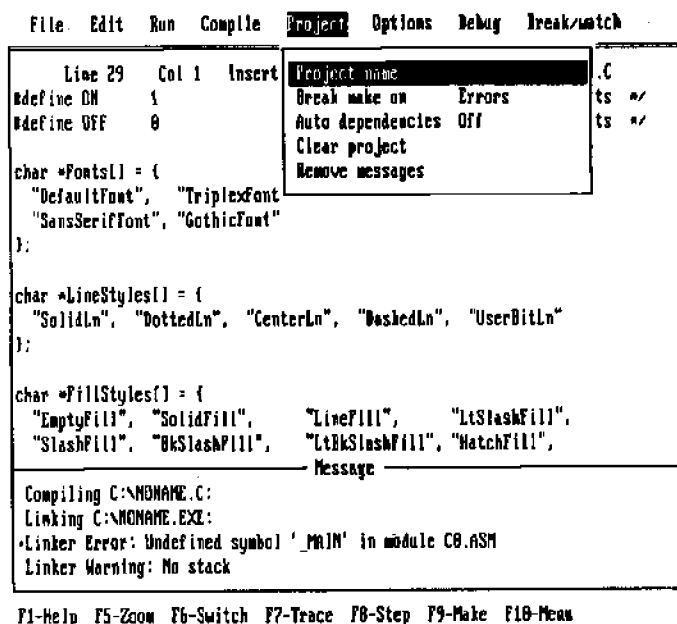


图 1.8 Project 菜单

Project Name

选择一包含将要编译、连接的文件名的 project 文件,工程名也将是以后将建立的 .EXE 及 MAP 文件名。典型的工程文件具有 .PRJ 扩展名。

Break Make On

此菜单让用户说明中止 make 的缺省条件——有警告(Warnings)时;有错误(Errors)时;中致命错误(Fatal Errors)时;或连接之前。

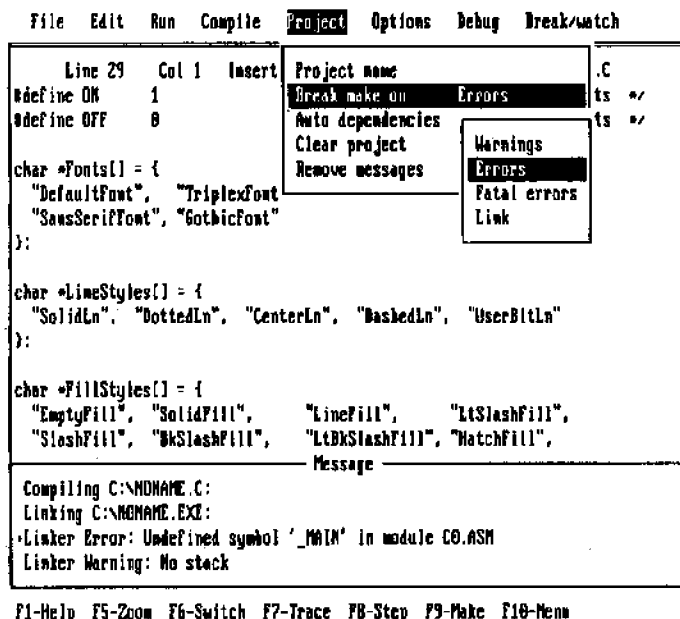


图 1.9 Project|Break On 菜单

Auto Dependencies(自动依赖检查)

它是一开关。置为 On 时,工程式 Make 自动检查每个工程表中在盘上有相应 .C 文件的那些 OBJ 文件的依赖关系。

工程式 Make 打开 OBJ 文件,寻找包含在源代码的那些文件的有关信息。这种信息总是在编译源模块时即被 Tc 或 Tcc 放进 OBJ 文件了。这时,把每一个组成 OBJ 文件的源文件的日期|时间信息同 OBJ 里的进行比较。若日期不同,则重编译 C 源文件。此所谓自动依赖关系检查。

若 Auto Dependencies 开关置为 off,则不进行这种检查。

Clear Project 清除工程

该命令清除工程文件名,重置消息窗口(Message Window)。

Remove Messages 删除消息

该命令把错误消息从消息窗口中清除掉。

1.2.6 Options 菜单

Options 菜单含有控制集成环境工作的设置。这些设置影响诸如编译、连接的选项、库、目录、程序运行参数等。该菜单的选项包括产生子菜单、完成一个设置和两个执行管理任务的命令:

. Compiler (产生子菜单)

- . Linker (产生子菜单)
- . Environment (产生子菜单)
- . Directories (产生子菜单)
- . Arguments (设置)
- . Save Options (管理)
- . Retrieve Options (管理)

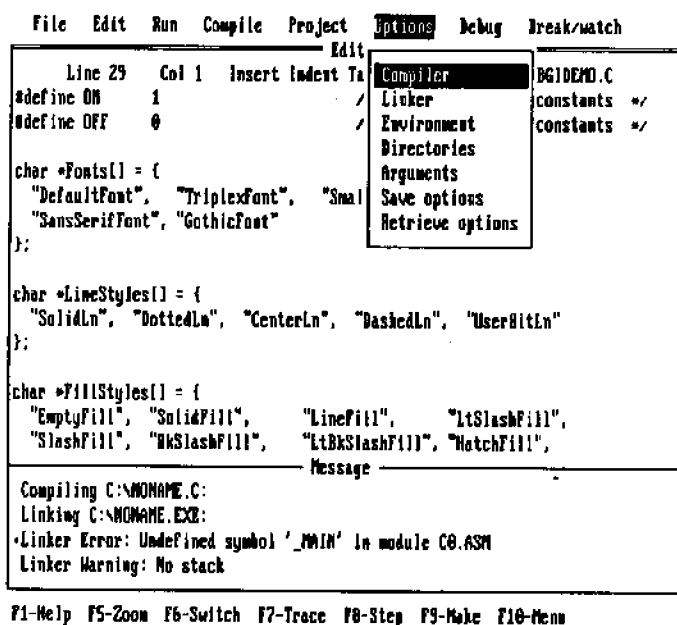


图 1.10 Option 菜单

1.2.6.1 Compiler(编译器子菜单)

本菜单中的各选项提供给用户说明硬件配置、存储模型、调试技术、代码优化、诊断消息控制及宏定义:

- . Model
- . Defines
- . Code Generation
- . Optimization
- . Source
- . Errors
- . Names

Model(内存模式)

该命令与 Turbo C 中的存储模式开关有些不同。选择的存储模型决定存储器寻址的缺省方法。选项可以是 Tiny、Small、Compat、Medium、Large 和 Huge。缺省存储模式为 Small, 因此单词 Small 一般出现在菜单的右边。

Defines(宏定义)

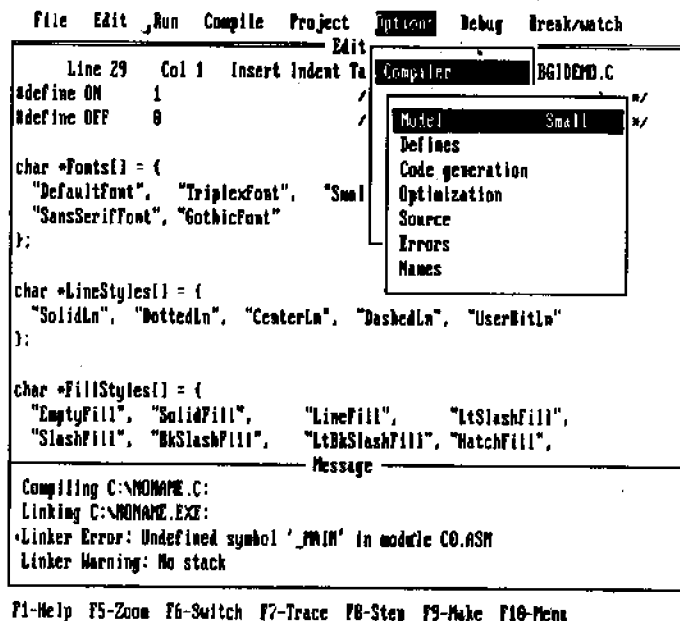


图 1.11 Option | Compiler 菜单

Define 打开一个宏定义框,通过它用户可对处理机输入宏定义。多重“定义”可用分号(;)隔开,赋值可用等号(=)。

前导和尾随空格都被去掉,但中间的空格保留。若宏定义中需要分号,必须在前面添加反斜杠(\)。

下面的宏定义了 BETA_TEST,置 ONE 为 1,COMPILER 等于字符串 TURBOC:

```
BETA_TEST;
ONE=1;
COMPILER=TURBOC;
```

Code Generation(代码生成)

这些选项告诉编译生成什么样的目标代码。

1. Calling Convention 调用约定

让编译产生一个 C 或 Pascal(Fast)函数调用序列。C 和 Pascal 调用约定不同之处在于清栈方式、参数个数及顺序、外部标识符的字体、前缀(下划线)。

2. Instruction Set 指令设置

说明目标 cpu。此开关在 8088/8086 指令集与 80X86 指令集间翻转。缺省为 80X86 代码。Turbo C 可产生扩展的 80X86 指令。也可用它来生成在实际模型上运行的 80X86 程序(如 MS-DOS 支持之下的 IBM PC AT)。

3. Floation Point 浮点数

此开关有三种选择

- 8087/80287 直接产生 8087/80287 代码;
- Emulation 检查有没有 8087/80286,有则使用它;否则,使用仿真 8087/80287,只是稍慢些;

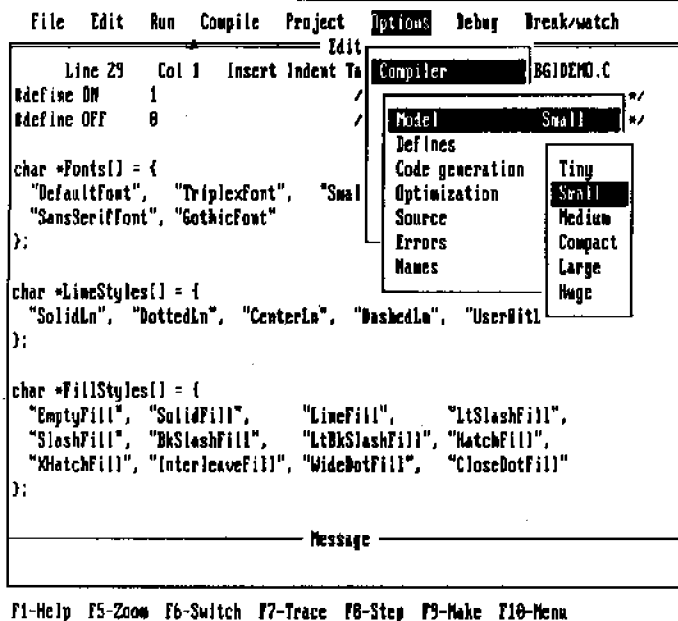


图 1.12 Options | Compile | Model 菜单

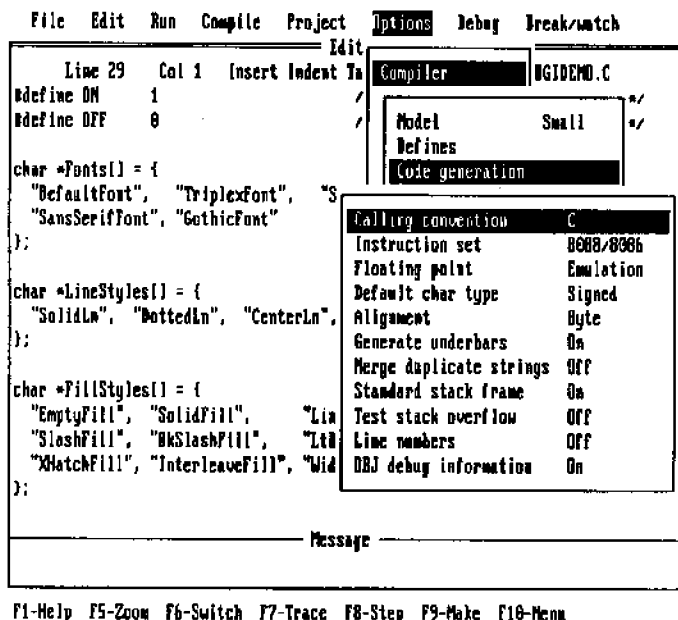


图 1.13 Options | Compile | Code generation 菜单

● None

不使用浮点数(若选了 None,而在程序中又有浮点运算,就会出现连接错误)。

4. Default Char Type 缺省字符类型:

此开关可选择 Signed 和 Unsigned。若选择 Signed,则编译视所有 char 声明为有符号字

符类型,反之为无符号的。缺省值为 Signed。

5. Alignment 对齐:

此开关可选择字(WORD)对齐和字节(BYTE)对齐。字对齐时,字符数据不能偶对齐;而字节对齐时,奇偶地址均可,完全决定于下一个可用空间。字对齐可提高 8087 和 80286 处理机存取数据的速度。

6. Merge Duplicate Strings(合并相同字符串)

该设置用作优化,它将匹配的字符串合并在一起,生成规模小一点的程序,缺省为 Off。

7. Standard Stack Frame(标准堆栈框架)

产生一个标准的栈结构(标准函数入口及退出码)。使用调试器时,是有用的一简化了反向跟踪调用过程栈的处理,缺省为 On。

Standard Stack Frame 是一开关,编译源文件时。若此开关为 Off,那么任何不使用局部变量的列参数函数都同压缩入口及返回码一起编译。这使得代码更短更快,阻止 Debug | Call Stack“看见”该函数。所以编译要调试的源文件时,此开关应置为 On。

8. Test Stack Overflow(堆栈溢出测试)

产生运行时检查堆栈溢出的代码。虽然花费了空间时间,但确实比较安全,因为堆栈溢出可能成为跟踪的障碍。缺省为 Off。

9. Line Numbers(行号)

在映射文件里放进(符号调试器用的)行号。虽然目标文件和映射文件都会变大,但不影响程序执行速度(可执行文件会变大,如果 Debug | Source Debugging 开关置为 On,且连接时 Options | Compile | Code Generation | Line Number 开关为 On;多出来的字节是调试信息)。此开关缺省值是 Off。

由于编译器在进行跳转优化时,或组合源文件中的公共代码行,或记录行号(这不利于行号跟踪),所以我们建议:当此选项为 On 时,Options | Compiler | Optimization | Jump Optimization 应置为 Off。

10. OBJ Debug Information(调试信息)

控制调试信息是否放入 OBJ 文件里。缺省为 On。这样便于集成环境和单独的 Turbo Debugger 调试。

Optimization(优化)

按程序需要优化用户代码。

1. Optimize for(代码生成策略选择)

改变 Turbo C 代码生成策略。一般情况下编译使用 Size,选择尽可能小的代码;若选用 Speed,则编译选择生成速度较快的目标代码。

2. Use Register Variables(使用寄存器)

On 时,寄存器变量自动分配给用户程序;Off 时,编译就不使用寄存器变量了,即使在程序里出现了关键字 register。

通常情况下,此开关可置为 On,除非用的是不支持寄存器变量的虚拟汇编代码。

3. Register Optimization(寄存器优化)

通过记住寄存器的内容和尽可能多的重复使用它,来抑制过多的取数操作(Load operation)。注意使用该选项时,会受到警告,因为编译器无法知道通过指针对变量的间接修

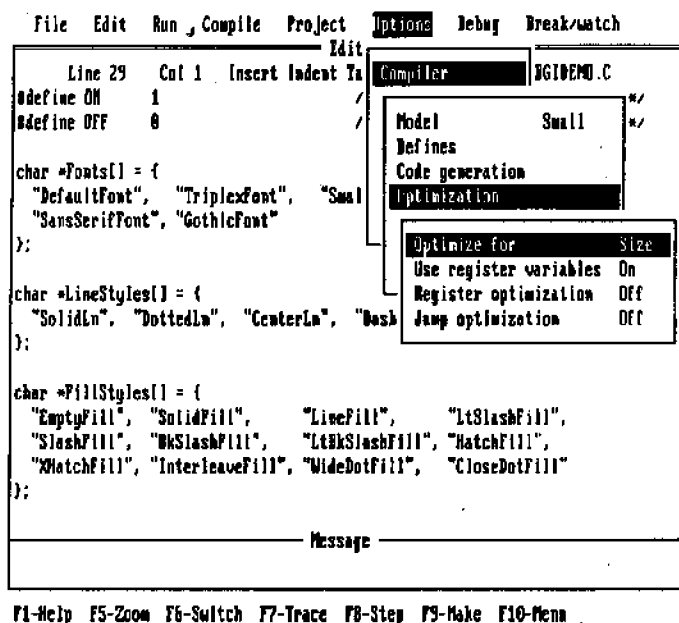


图 1.14 Options|Compile|Optimization 选择项

改。

4. Jump Optimization(跳转优化)

通过去除多余的跳转和重新调整循环及开关语句来压缩代码,循环调整能加快内循环的执行速度。

注:当此项选项为 On 时,集成调试器可能会出错,因为可能有若干源代码行对应同一组执行代码。所以,若要调试,该选项最好置为 Off。

Source(源代码)

本菜单上的各条目控制编译初始时如何处理源代码。

1. Identifier Length(标识符长度)

说明标识符中有效字符个数,只有前 N 个字符不同的标识符才被视为有区别的。这个规则适合于变量、预处理宏名、结构成员名,数目可以在 1 到 32,缺省值是 32。

2. Nested Comments(嵌套注释)

在 Turbo C 源文件中允许出现嵌套注释。嵌套注释在一般的 C 实现里是不允许的,且不可移植。

3. ANSI keywords only(只用 ANSI 关键字)

若想让编译只识别 ANSI 关键字,而把 Turbo C 扩展关键字当一般标识符看待,则可将本开关置为 On,扩展关键字包括:hear、far、huge、asm、cdecl、pascal、interrupt、_es、_ds、_cs 和 _ss。

Errors(错误)

利用本菜单上的这些命令,用户可以控制 Turbo C 编译器如何处理和响应诊断消息。

1. Errors/Stop After

此选项使行编译器在发现了指定个数的错误后,停止编译,缺省为 25。用户也可指定 0

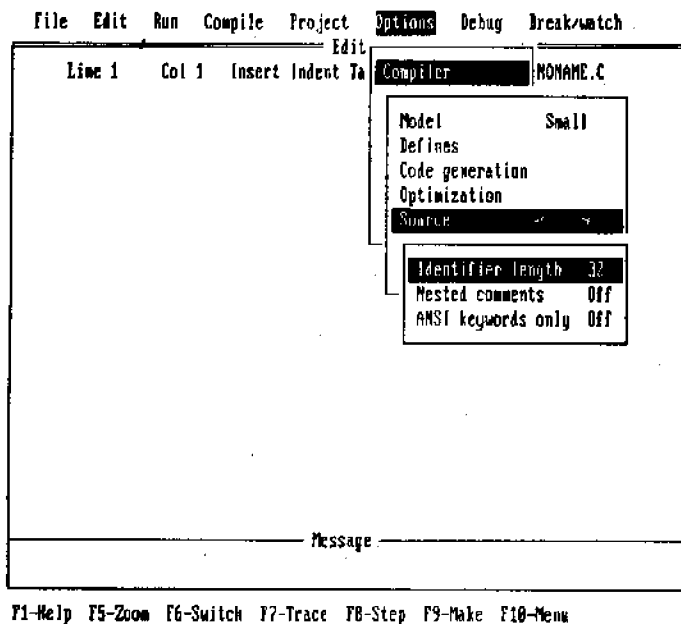


图 1.15 Options|Compile|Source 菜单

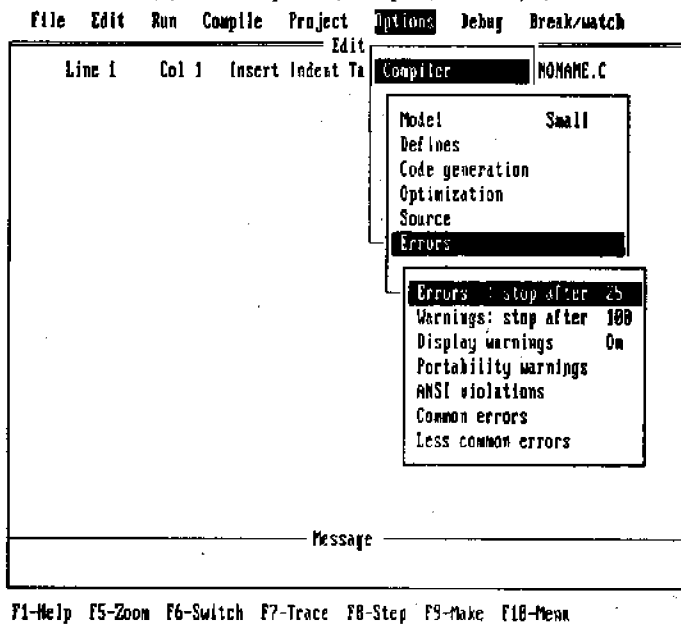


图 1.16 Options|Compile|Errors 菜单

到 255 的任意一数(0 将导致编译器一直继续下去,或达到错误上界)。

2. Warnings:Stop After

选择此项使得编译器在发现了 100 个警告错误后停止。当然,100 仅为缺省值,合法范围进 0 到 255,其中,0 将导致编译器一直继续下去,或达到错误上界。

3. Display Warnings 显示警告

缺省时,此开关为 On,意思是下列各警告类型可以得到显示(如选择了的):

- Portability Warning(移植性警告)
- ANSI Violations(ANSI 违例)
- Common Errors(常见错误)
- Less Common Errors(很少见错误)

若此条目置为 Off,则不显示任何警告。

Names 菜单

利用本菜单提供的条目,用户可改变代码,数据和 BSS 段的缺省段、组、类名。选择其中的任一条目,出现在下一级菜单的星号(*)告诉编译器使用缺省名字。除非很熟悉,不要改变此选项。

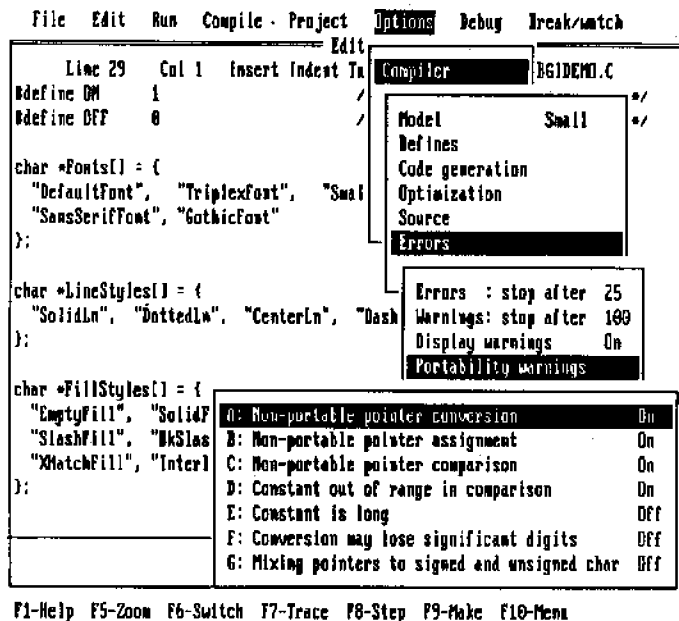


图 1.17 Portability Warning(移植性警告)

1.2.6.2 Linker(连接器子菜单)

本菜单上的条目是有关连接顺序的设置。

Map File Menu(映射文件)

选择映射文件的类型。取 Off 以外值时,射映文件就会被放大由 Options | Directories | Output Directory 的指定的输出目录中。缺省为 Off,别的选择可以是 Segments、Publics 和 Detailed。

Initialize Segments(段初始化)

告诉连接器初始未初始化过的段(一般不需要,会使 .EXE 文件比必要的大)。

Default Libraries(缺省库)

当连接非 C 编译器产生的模块时,那些编译器可能已在目标文件中放入了一个缺省库

表。

若此选项置为 On, 连接器就会试图在这些库和 Turbo C 提供的缺省库中查找未定义的过程。

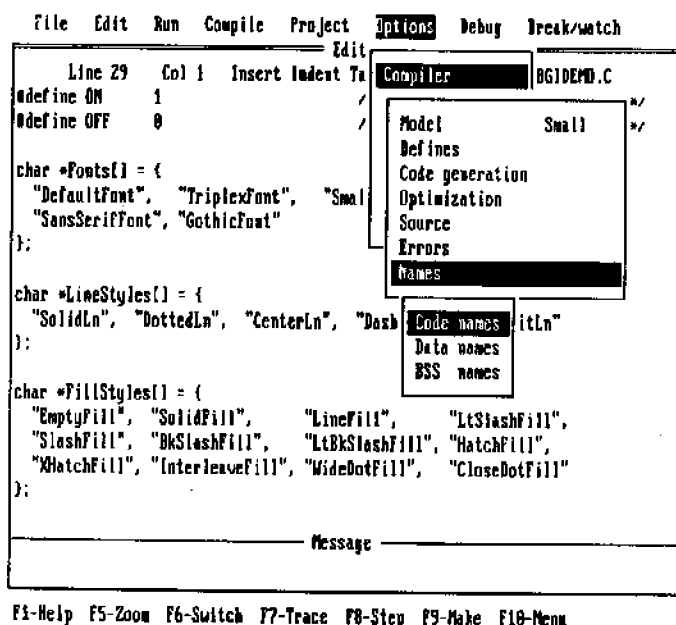


图 1.18 Options|Compile|Names 选择项

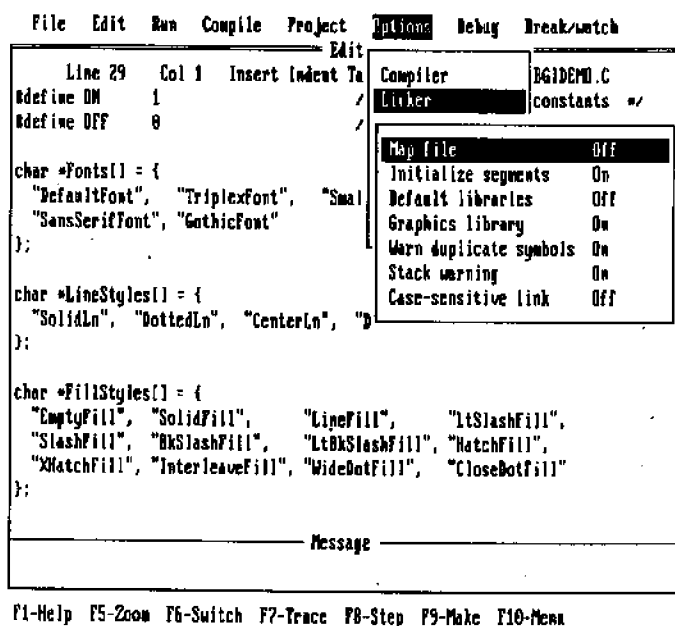


图 1.19 Option|Linker 菜单

Graphics Libraries(图形库)

打开或关闭自动查找 BGI 图形库, 置为 On 时, 就可以建立运行单文件图形程序(不需

使用 project 文件); 置为 Off 时; 可加快连接速度, 因连接器没有必要连接图形库。缺省为 On。

注意: 可将此开关置为 Off, 而同样建立使用到 BGI 图形库的程序, 假如在 project 文件里写入 BGI 图形库名。

Warn Duplicate Symbols(警告重复字符)

打开或关闭连接器警告在目标及库文件里出现的相同符号。缺省值为 Off。

Stack Warning(堆栈警告)

抑制连接器产生 No Stack 消息。一般情况下, 在小模式下生成程序时会产生这种消息 (如果不是 Off 的话)。

Case-sensitive Link(大小写区别连接)

打开或关闭大小写的作用。通常应选 On, 因 C 是区别大小写的。

1.2.6.3 Environment(环境设置子菜单)

本菜单的条目让用户备份在编译器里的源文件, 裁减 Turbo C 工作环境以适应程序需要。

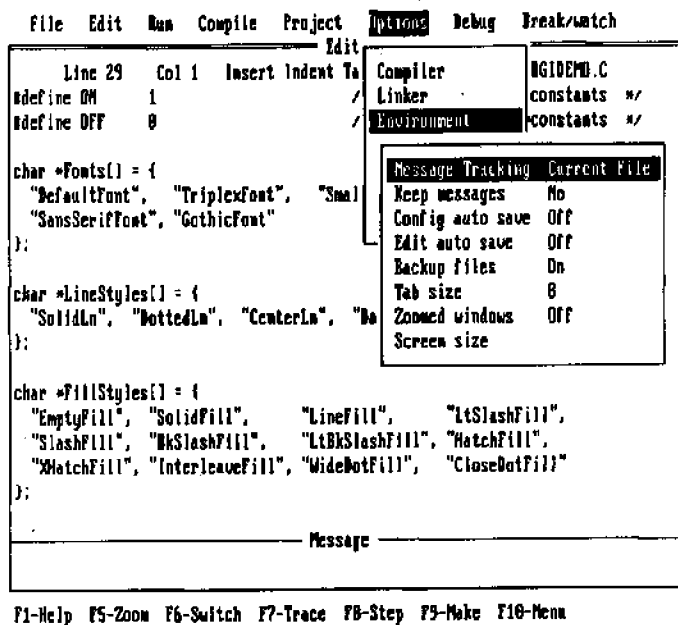


图 1.20 Options|Environment 菜单

Message Tracking(消息跟踪)

当滚动消息窗口里的信息时, Turbo C 会跟踪编辑器里的语法错误。这个三路开关告诉 Turbo C 在哪些文件里跟踪。

缺省 (Track...Current File) 只跟踪编辑器里的文件, Track...All Files 将加载跟踪与错误信息相对应的每个文件。可置为 Off。

Keep Messages(保存消息)

开关: 置为 On 时, Turbo C 保存当前在消息窗口里的错误信息, 把进一步编译的信息附

在后面;若选择 Off,编译 make 之前自动清除信息。

Config Auto Save(配置自动保存)

正常情况下,只有当选择了 Options|Save Options 命令后,Turbo C 才会保存当前配置(写盘)。Config Auto Save 置为 On 后,不管是选择的 Run|Run 或者 File|OS Shell,还是退出集成开发环境,只要配置文件从没存过或存过以后又修改过,Turbo C 都会保存它的。

Config Auto Save 为 On 时,若配置文件还未存过,Turbo C 为自动存储文件选定一文件名。名字或以是上次保存或恢复的配置文件名,也可以是 TCCONFIG.TCC(在当前目录若还未加载、恢复存过配置文件)。

Backup Files(备份文件)

缺省情况下,当用户使用了 File|Save 时,Turbo C 会自动为编辑器里的文件创建一备份文件。备份文件扩展名为.BAK,利用该选项可将备份特征置为 On/Off。

Tab Size(制表键大小)

当编辑器制表模式为 On,而你又按了 Tab 键时,编辑器就会在文件里插入一制表符,同时光标跳至下一制表位置。这个菜单条目允许用户确定一个制表有多长,2 到 16 间任一数均可。缺省为 8。

要改变制表键在文件里的显示方式,也只需要把 Tab Size 改为所需大小即可。编辑器就地以所选择的大小来重显示文件里的所有制表符,也可将新制表尺寸保存到配置文件里(从 Options 菜单里选择 Save Options)。

Zoomed Windows(缩放窗口)

若 Turbo C 集成开发环境的屏幕设为编辑和消息窗口同时显示,那么选择 Zoomed Windows.. On 就会将两个窗口放大到整屏,而只有活动窗口可见。

F6 切换两个窗口,好似两个窗口都在显示。

“缩小”窗口(回到原来两个窗口同时显示的设置)只要选 Zoomed Windows.. Off 即可。

Screen Size(屏幕大小)

选择 Screen Size 后,又会出现一菜单。Screen Size 菜单上的各条目允许用户说明集成环境屏幕显示 25 行还是 43/50 行正文。其中的一两个条目可用,这就要看 PC 机上的视频适配器类型了。

1. 25 Lines

标准 PC 显示:24 行 80 列,此菜单条目总是可用的。这是适合于具有单色显示适配器(MDA)和彩色图形适配器(CGA)系统的唯一屏幕大小。

2. 43/50 Lines

若 PC 机装有 EGA 或 VGA,此菜单条目即可用,当然 25 行标准显示也是可行的,选择它就将文本转换成 43 行 X80 例(对于 EGA)或 50X80 行列(对 VGA)

1.2.6.4 Directories(目录设置)

本菜单各项告诉 Turbo C 到哪里去寻找编译、连接、所需的文件;生成的可执行文件放到哪里;在哪里查找配置文件、选择文件和帮助文件。

Include Directories(包含目录)

说明标准包含文件的目录。标准包含文件由 #include 语句的尖角(<>)给出(例如, #include <myfile.h>,多个目录可用分号(;)隔开。

Library Directories(库目录)

说明包含 Turbo C 启动目标文件(C0?.OBJ)和运行库文件(.Lib 文件)的目录。允许用户列出多个库目录,最多可达 127 个字符(包括空格)。

输入库目录时请用以下方法:

- 必须用分号(;)隔开多个目录路径。
- 分号前后的空格是允许的,但不必要。
- 相对和绝对路径均可使用。

例如:

C:\TURBOC\LIB; C:\TURBOC\MYLIBS;A:\NEWTURBO\MATHLIBS;A:...\VIDLIBS

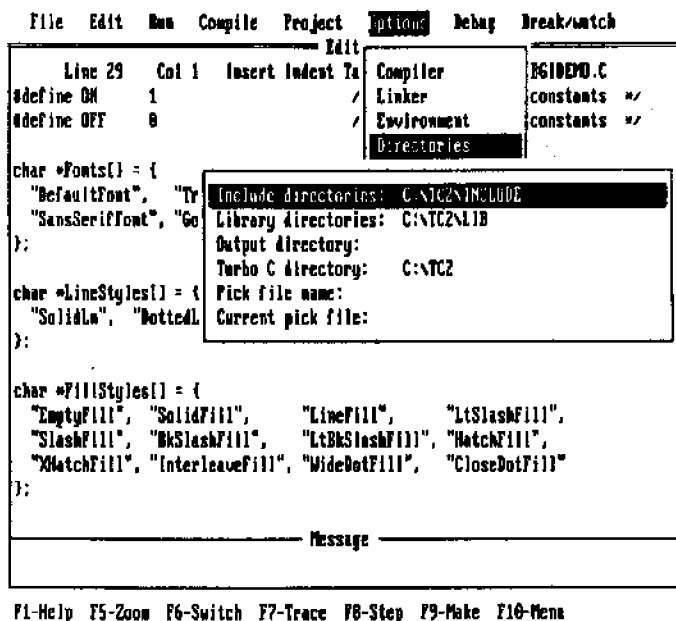


图 1.21 Options|Directories 菜单

Output Directory(输出目录)

用户的.OBJ、.EXE 及 .MAP 文件均放在此选项指定的目录。Turbo C 在做 make 和 Run 时到这里来找它们。若为空,则这些文件被放在当前目录。

Turbo C Directory (Turbo C 目录)

用于 Turbo C 系统寻找配置文件(.TC)和帮助文件(TCHELP.TCH)。为了让 Turbo C 在启动时寻找缺省配置文件(TCCONFIG.TC)(假如不在当前目录),用户必须使用 TCINST 来设置路径。

Pick File Name (Pick 文件名)

定义加载的 pick 文件名。在这里输入一文件名加载 pick 文件(若存在的话),定义当用户退出时到哪里保存 pick 文件。改变 pick 文件名时,Turbo C 保存当前 pick 文件后才加载新的文件。

若在这里没有列出 pick 文件名,Turbo C 只有在当 Options | Directories | Current Pick

File 设置包含一文件名时才写 pick 文件。

要想创建 pick,就得定义 pick 文件名,通过向 Options|Directories|Pick File Name 设置激发的提示框输入一文件名即可达此目的。一旦定义了 Pick 文件名,Turbo C 就会在退出集成开发环境时更新 Pick 文件。如果已选择了 Options|Save Options 的话,该文件名会存到配置文件里。

Current Pick File(当前 Pick 文件)

若存在的话,显示当前 pick 文件的文件名和位置。此条目是不能修改的,它只是一信息。Current Pick File 在加载一缺省 Pick 文件借口通过 Pick File Name 命令输入 pick 文件显示一文件。若用户改变 Pick 文件名或退出集成开发环境,Turbo C 就把当前 pick 表信息存入在所列 pick 文件里。

1.2.6.5 Arguments(参数)

该设置允许用户给出运行程序使用行,正如 DOS 命令行上的键入(改向不支持),只须给出参数。文件名可省去。

1.2.6.6 Save Options(保存选项)

保存所有选择的编译器、连接器环境、调试和 Project 选项到一配置文件里(缺省文件是 TCCONFIG.TC)。启动时 Turbo C 再到 Turbo C 目录去找同样的文件。

1.2.6.7 Retrieve Options(恢复选项)

加载以前用 Options|Save Options 命令保存的配置文件。

1.2.7 Debug 菜单

Debug 菜单的命令控制集成调试器除断点和监视表达式以外的条目(此两项条目在 Break|Watch 菜单)。

Evaluate(计算)

Evaluate 计算变量或表达式并显示其值;若可能的话还允许用户对其修改。

该命令打开一个弹出窗口,包括三个段:计算段(Evaluate)、结果段(Result)和新值段。Evaluate 段里的缺省表达式由编辑窗口中光标所在处的单词组成,键入回车键可以计算缺省表达式,或者先编辑替换它。也可以利用向光标键将编辑窗口里的其它字符扩展到缺省表达式里去。

计算任何合法 C 表达式,但不能包含:

- 函数调用
- 用 #define 或 typedef 定义的符号及宏,或
- 不在当前执行函数里的局部、静态变量除非完全修饰。

若调试器可计算表达式,就在 Result 段中显示其值。

修改表达式的值是有意义的,但若不想这么做,按 ESC 关闭这窗口。若已改变了新值段中的内容,但尚未按回车键,则当退出窗口以后调试器就忽略这种改变。

Debug|Evaluate 以适当的格式显示各种类型的值。例如,以 10 为基数显示 int 型,以 16 为基数象指针一样显示数组。若要改变显示的格式,可在表达式后面加一逗号和表 1.5 中的格式说明符。

用重复表达式来显示一系列数据元素的值,例如,对于 xarray 整数组:

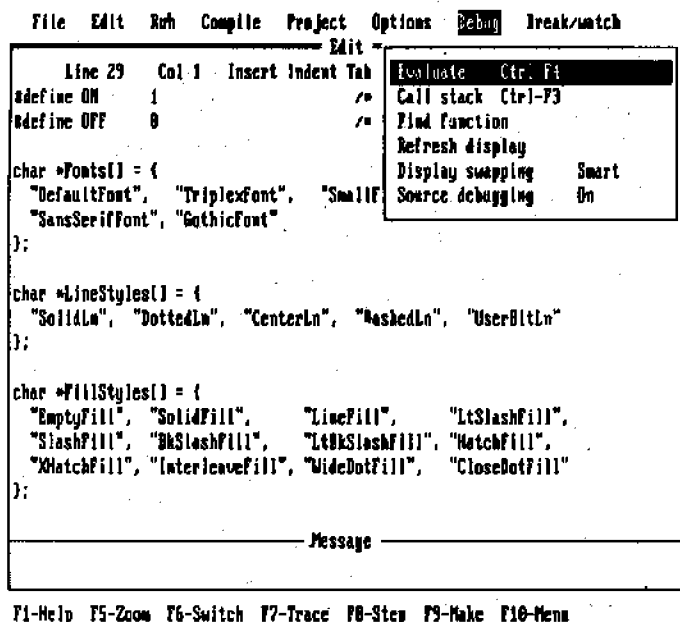


图 1.22 Debug 菜单

xarray[0],5 以十进制显示 5 个连续的整数

xarray[0],5x 以十六进制显示 5 个连续的整数

具有重复次数在表达式必须代表一个数据元素,调试器把数据元素看成是数组的第一元素(若不是指针),或一个指向数组的指针(若是指针的话)。

Debug|Evaluate 热键:CTRL-F4。

表 1.5 调试器表达式格式说明符

字 符	功 能
C	字符。显示特殊控制字符(ASCII0 到 31);缺省时,这些字符用 C 语言的形式(\n,\t 等)。影响字符和字符串。
S	字符串。显示以 ASCII 值和 C 转义字符来显示控制字符(ASCII0 到 31),由于这是缺省的字符及字符串显示格式,S 说明符只有和 M 说明符联用才有用。
D	十进制。整数值按 10 进制显示。影响简单整数表达式和包含整数的数组、结构。
H 或 X	十六进制。所有整数值按 16 进制显示。影响简单整数表达式和包含整数的数组、结构。
F<n>	浮点数。n 是介于 2 到 18 的说明有效数字个数的数,缺省为 7,仅影响浮点值。
M	M 显示内存,从由表达式指定的地址开始显示内存。表达式必须是在赋值语句左边也有效的形式。也即指示地址的标识;否则 M 说明符被忽略。缺省时,每一变量字节以两位十六进制数显示,M 加 D 说明符,则字节以十进制显示;M 加 H 或 X 则以十六进制显示。AC 或 S 说明符使变量以字符串方式显示(包括特殊字符)。缺省时,显示多少字节由变量大小而定,当然也可用重复次数来说明显示的字节数。

续表 1.5

字 符	功 能																								
P	<p>指针。按 seg:ofs 的格式显示指针,同时还显示指向什么地址的信息。注意,不是以面向硬件的 seg:ofs 格式显示。特别指出,它还能告诉段的内存定位范围,以及在那个偏移地址的变量名,如合适的话。</p> <p>内存范围如下:</p> <table> <tr> <th>内存范围</th><th>Evaluate 信息</th></tr> <tr> <td>0000:0000-0000:03FF</td><td>中断字节表</td></tr> <tr> <td>0000:0400-0000:04FF</td><td>BIOS 数据区</td></tr> <tr> <td>0000:0500-Turbo C</td><td>MS-DOS/TSR</td></tr> <tr> <td>Turbo C</td><td>Turbo C</td></tr> <tr> <td>User program PSP</td><td>用户进程 PSP</td></tr> <tr> <td>用户程序-RAM 顶端</td><td>用户静态变量名,若变量落入变量分配的内存;否则无意义。</td></tr> <tr> <td>A000:0000-AFFFF:FFFF</td><td>EGA 视频 RAM</td></tr> <tr> <td>B000:0000-B7FF:FFFF</td><td>单色显示 RAM</td></tr> <tr> <td>B800:0000-BFFF:FFFF</td><td>彩色显示 RAM</td></tr> <tr> <td>C000:0000-EFFFF:FFFF</td><td>EMS 页面/适配器 BIOS</td></tr> <tr> <td>F000:0000-FFFF:FFFF</td><td>BIOS ROM 区</td></tr> </table>	内存范围	Evaluate 信息	0000:0000-0000:03FF	中断字节表	0000:0400-0000:04FF	BIOS 数据区	0000:0500-Turbo C	MS-DOS/TSR	Turbo C	Turbo C	User program PSP	用户进程 PSP	用户程序-RAM 顶端	用户静态变量名,若变量落入变量分配的内存;否则无意义。	A000:0000-AFFFF:FFFF	EGA 视频 RAM	B000:0000-B7FF:FFFF	单色显示 RAM	B800:0000-BFFF:FFFF	彩色显示 RAM	C000:0000-EFFFF:FFFF	EMS 页面/适配器 BIOS	F000:0000-FFFF:FFFF	BIOS ROM 区
内存范围	Evaluate 信息																								
0000:0000-0000:03FF	中断字节表																								
0000:0400-0000:04FF	BIOS 数据区																								
0000:0500-Turbo C	MS-DOS/TSR																								
Turbo C	Turbo C																								
User program PSP	用户进程 PSP																								
用户程序-RAM 顶端	用户静态变量名,若变量落入变量分配的内存;否则无意义。																								
A000:0000-AFFFF:FFFF	EGA 视频 RAM																								
B000:0000-B7FF:FFFF	单色显示 RAM																								
B800:0000-BFFF:FFFF	彩色显示 RAM																								
C000:0000-EFFFF:FFFF	EMS 页面/适配器 BIOS																								
F000:0000-FFFF:FFFF	BIOS ROM 区																								
R	结构/联合。显示字段名及其值。如 {X:1,Y:10,Z:5},只影响结构及联合。																								

Call Stack 调用栈

Call Stack 显示一包含调用栈的弹出窗口。调用栈显示程序运行到正在运行的函数时调用的函数序列。main 在栈底,正在运行的函数在栈顶。

调用栈上的每一项显示了函数名及传递给它的参数值。

开始时,栈顶的那一项是高亮度的,为了显示栈上除当前行以外其它任意函数,可把高亮长条移到函数名上。按回车键,光标将定位在包含调用紧接着栈上的次下层的那一个函数的行。例如,有如下所示的栈:

```
func2()
func1()
main()
```

即 main 调用 func1,func1 调用 func2,而用户想看 func1 中的当前执行笔,可将光标移到栈中 func1 上,按回车键,func1 代码将显示在编辑窗口中,而光标将定位在调用 func2 的那一行上。

想回到正在运行的函数当前行(即执行位置),把高亮条移到栈顶,再按回车键。

有些函数可以被调用栈忽略,如果程序编译时 Options | Compile | Code Generation | Standard Stack Frame 置为 Off。参见 Options | Compile | Code Generation | Standard Stack Frame 的描述。

Find Function(查找函数定义)

Find Function 显示编辑窗口中某一函数的定义。该命令可找到程序里的任何一个函

数,只要编译时 Debug|Sourec Debugging 和 Options|Compile|Code Generation|OBJ Debug Information 设为 On,并且可取到源文件。如果函数不在当前显示文件里,该命令会加载适当的文件。

只有在调试阶段才能使用 Find Function。

Refresh Display(刷新显示器)

万一编辑屏幕被重写,可用这条命令恢复当前屏内容。

Display Swapping(显示切换)

Debug|Display Swapping 有三种选择 Smart(缺省)、Always 和 None。

当在调试模式运行程序,而用的是缺省 Smart 设置时,调试器就看正在执行的代码中是否产生屏幕输出。若产生(或它调用一函数),则屏幕就从编辑切换到用户屏,其时间足够完成输出。然后又切换回去,否则,不进行切换。

注:缺省 Smart 在下列情况下并不特别“灵”:

- 每次函数调用都切换,即使不产生屏幕输出
- 在有些情况下,编辑屏可能被修改了而并没有切换,例如时钟中断程序写屏。

Always 设置使得执行每条语句都切换。当编辑屏有可能被运行程序重写时,都应该采用这种选择。

None 设置使调试器根本就不切换。它用在确实不含屏输出的代码调试中。

若在双监视模式下调试(即用了 TC 命令行/d 开关),则用户程序输出到一个屏上,而 TC 屏在另一个上,因此,TC 不进行切换,而 Debug|Screen Swappint 设置就不起作用了。

Source Debugging(源代码调试)

Debug|Source Debugging 有三种选择:On、Standalone 和 None。开关为 On 时,连接的程序可用 TC 集成调试器和单独的 Turbo C 调试器调试;开关设为 Standalone 时,就只能用 Turbo 调试器调试了,当然程序还是可以在 TC 中运行;若为 None,两种调试器均不行了,因为 EXE 文件中没放入调试信息。

1.2.8 Break/Watch 菜单

Break/Watch 菜单命令控制断点及监视表达式。

断点是程序运行暂停的地方。让用户检查关键变量、表达式的值。

断点以断点高亮标记。断点高亮标记会被当程序运行到断点而暂停的执行长条压暗,但执行长条移走后又会再现。

监视表达式是其值在监视窗口中显示的表达式,每当程序暂停时,其值会被重计算,输入监视表达式的规则同输入 Debug|Evaluate 表达式一样,只是监视表达式不含如 i++ 的有副作用的形式。类型转换字符和循环重复计数也可用在监视表达式,象 Debug|Evaluate 一样;例如,

i,x

以十六进制显示 i 的内容。

向监视窗口里增加表达式时,窗口会变长,一直到由 TCINST 程序中 Resize Windows 说明的大小(初始化时,是半个屏幕)。若继续增加,则有些会滚出窗口。这时,可用 PgUp、PgDn、Up 和 Down 光标键滚动监视窗口来再现前面的表达式。当监视窗口活动时,监视窗

口中的当前监视表达式以高亮长条标记;不活动时,最左标以小棱形(◆)。

Add Watch 增加监视表达式

Add Watch 向监视窗口插入一监视表达式。当选择此命令时,调试器打开一弹出窗口,提示用户键入一监视表达式。缺省为表达式编辑窗口中光标所在单词。若输入的表达式有效,按回车键之后,调试器就增加一表达式以及当前值到监视窗口。

热键是 CTRL-F7。另外,监视窗口活动时,可用 Ins 或 CTRL-N 插入监视表达式。

Delete Watch 删除监视表达式

Delete Watch 从监视窗口中删除当前监视表达式。

只有当监视窗口可见时(也即编辑窗口不可放大),才能使用此命令。监视窗口活动时,当前监视表达式以高亮条标记;而在编辑窗口时,左边空处以小棱形标记。

当编辑窗口活动时,可选 Break/Watch|Delete Watch 删除小棱形标记的监视表达式。若是删去非当前监视表达式,就必须先进入监视窗口,把高亮条定位到欲删的监视表达式之上,按 Del 或 CTRL-Y。

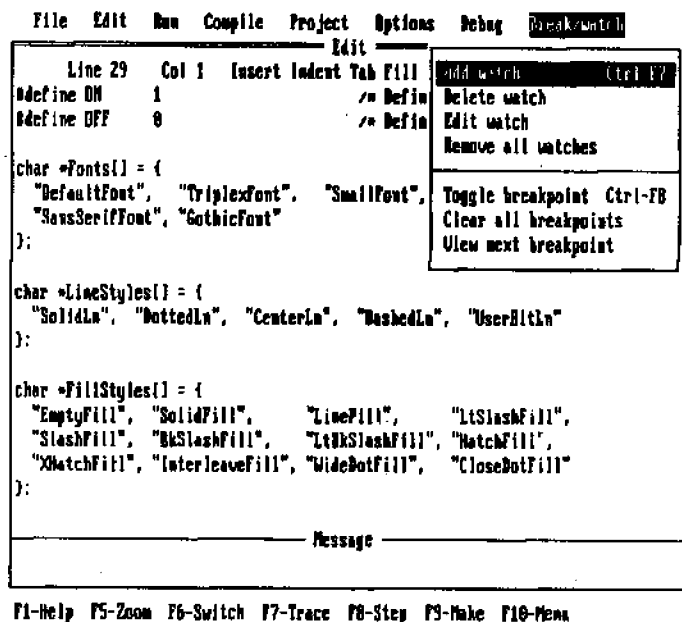


图 1.23 Break/Watch 菜单

Edit Watch 编辑监视表达式

Edit Watch 允许用户编辑监视窗口中的当前监视表达式。

选择了 Break/Watch|Edit Watch 之后,调试器弹出一含有当前监视表达式拷贝的窗口。编辑之后,按回车键,调试器即用编辑过的监视表达式替换掉原来那一个。将高亮条移到表达式之上再按回车键。这样也可在监视窗口里编辑监视表达式。

Remove All Watches 删除所有监视表达式

Remove All Watches 将所有监视表达式从监视窗口中删除。

Toggle Breakpoint 打开或关闭断点

Toggle Breakpoint 设置或去除光标所在断点,当一断点被置后就标以高亮度。

该命令的热键是 CTRL-F8。

在运行过程当中一旦遇到断点程序都会暂停。发生这种情况时,执行长条位于断点之上,标记断点的高亮条被执行长条压暗,长条移走后又恢复。

编辑源文件时,断点始终“附着”在原来设置的地方,只有在以下几种情况才会丢失:离开集成开发环境;删除设置所在行的源文件;;或以 Break/Watch|Toggle Breakpoint 命令及 Break/Watch|Clear|All Breakpoints 命令清除掉了断点。

Turbo C 在以下两种情况失去对断点的跟踪:

- 编辑了一含断点的文件,然后又将它放弃(Turbo C 在文件编辑之前是无法知道断点设在何处的,因此会在错误行显示断点)。

- 若编辑了一含断点的文件,马上又进行当前的调试节,而没有再编译程序(Turbo C 显示警告提示:Source moditied, rebuild?)。

编译源文件之前尽可在任意一行上设断点,甚至是空行或注释语句;编译运行文件后,Turbo C 证实所有断点,给用户一删除、忽略或改变无效断点的机会。调试文件时,Turbo C 知道哪些包含可执行语句。当设置无效断点时,会提出警告。

Clear All Breakpoints 清除所有断点

Clear All Breakpoints 从程序中删除所有断点。

View Next Breakpoint 显示下一个断点

View Next Breakpoint 将光标移到程序中的下一断点,注意光标移动是按设置顺序移的,而不是程序运行到断点。该命令不执行代码只是定位到编辑窗口中的活动断点。

1.3 配置和 pick 文件

简单地说,配置文件中包含与 Turbo C 有关信息的文件。其中存有诸如编译选择、连接选择和各种 Turbo C 编译连接时查找路径的信息。

有两种 Turbo C 配置文件类型:一种用于 TCC.EXE(命令行 Turbo C);另一种用于 TC.EXE(Turbo C 集成开发环境),只有命令行配置文件必须是 TURBOC.CFG,而集成开发环境的配置文件可以是任何文件名,TCCONFIG.TC 是缺省的(设定的)集成开发环境配置文件。

本节,将详细介绍集成开发环境配置文件。

1.3.1 TC 配置文件

首次进入 TC 集成开发环境时是没有配置文件的,TC.EXE 启动后所有菜单开关设置都是缺省初始态(Options|Compiler|Model 为 Small,Options|Compiler|Calling Convention 为 C,Option|Environment|Keep Messages 为 No,等等)。在使用集成开发环境过程当中,用户或许需要改变其中的一些设置。

若退出 Turbo C 而没有将新的设置保存到配置文件里,则下次启动仍然是原来的缺省的设置;若将新设置保存到配置文件里了,那么下次启动就会有选择的设置,而不必再逐个重设。

TCCONFIG. TC

TC. EXE 启动时会自动在某些目录(稍后再精确解释)查找名为 TCCONFIG. TC 的配置文件;若 TC. EXE 找不到 TCCONFIG. TC,则以内部缺省设置启动。

其它 TC 配置文件

当然也可在 DOS 提示符下使用 /c 开关来请求一特殊配置文件。例如,在 DOS 提示符下键入:

```
Tc /cmyconfig
```

Turbo C 就会在当前目录下查找名为 MYCONFIG. TC 的配置文件(若无扩展名。Turbo C 假设为 TC)。

如果 Turbo C 找不到指定的配置文件,则发出警告,以后就不再找其它配置文件,而以内部缺省设置启动。

TC 配置文件的内容

TC 配置文件里的信息可分为两类:编译连接选项和 TC. EXE 特殊值。

编译连接选项控制编译器和连接器,它们均和命令行 Turbo C 的选项相对应;而 TC. EXE 特殊值则是和集成开发环境本身有关。例如,Project|Project Name、Options|Directories|Pick File Name,和 Options|Environment 菜单项。

创建 TC 配置文件

怎样创建 TC 配置文件呢?不能象命令行配置文件(TURBOC. CFG)一样用编辑器创建及修改,而是从 Options 菜单中选择 Options|Save Options 命令,集成开发环境会自动创建一配置文件。

若设定 Options|Environment|Config Auto Save 为 On,则一旦退出集成开发环境,当前的设置会自动存放至缺省 TC 配置文件中(TCCONFIG. TC)。

更改配置文件

从集成开发环境内改变配置文件是很容易的:

- 从 Options 菜单选择 Options|Retrieve Options,系统就会弹出一框,显示了上次输入的配置文件名(首次缺省为 *. TC)。

- 输入匹配符(如 *. tc, ??config. *),再按回车键。这时系统给出 TC 文件的目录列表。从目录列表中选择一文件,或键入一另外的配置文件名,再按回车键加载该文件。

TC. EXE 到哪里寻找 TCCONFIG. TC

TC. EXE 到两个地方查找缺省的 TCCONFIG. TC 配置文件。先在缺省(当前工作)目录中找;若找不到,再到 Turbo C 目录去找,如果用 TCINST 设置的 Turbo C 目录中找。

TCINST 和配置文件:何者优先

可用 TCINST 来设置任何可在 Turbo C 菜单上可找到的条目,然后将设置存到 TC. EXE 中。若 TC. EXE 启动时未找到 TC 配置文件,那么就用缺省值。

但是,若是找到了,那么配置文件就将用 TCINST 建立的设置覆盖掉了。

另外,以 /c 开关调用 TC. EXE,而 Turbo C 找到了指定的配置文件,其结果也是将 TCINST 的缺省设置覆盖掉。

Options|Environment|Config Auto Save 的作用

一般情况下,Turbo C 只有当用户给出 Options|Save Options 命令后才存盘当前配置文

件;然而,在某些情况下用户可指示 Turbo C 自动保存配置文件。

只要将 Options|Environment|Config Auto Save 设为 On, Turbo C 就会在用户选择 Run |Run 或 File|OS Shell 或 File|Quit 退出集成开发环境时保存配置文件——只要文件尚未存过或自上次存过以后又修改过。若配置文件尚未存盘, Turbo C 将从下面两种可能性来指定其文件名:

- 上次保存或恢复的配置文件名
- TCCONFIG.TC(在当前目录),若尚未加载保存、恢复过配置文件。

1.3.2 Pick 表和 Pick 文件

Pick 表和 Pick 文件是 Turbo C 集成开发环境的两大特点。它们相互配合,保存编辑状态。Pick 表能记住用户在集成开发环境里正在编辑什么文件,而 Pick 文件知道当用户退出集成开发环境或在集成开发环境里改变上下文之后曾经编辑了些什么文件(改变上下文意指加载一新配置文件或定义一新 Pick 文件名)。

Pick 表

选 File|Pick 或按 ALT-F3 热键可调出一 Pick 表。File|Pick 提供了一最近加载到编辑器的八个文件列表。若 Pick 表中文件名不只一个,则第二文件名是高亮度的。它就是前面加载到编辑器的那个文件。

将光标移到欲加载的文件名上,按回车键,即可加载 Pick 表中的某一文件了。加载了选择的文件之后,光标将定位到上次退出的地方,而且标记块和标记一如原来退出时一样。

Pick 表是一个方便的工具,用户可利用它来回操作文件,以便开发程序。按 ALT-F3 后,再按回车键,用户可改变文件(这和在编辑器里按 ALT-F6 效果一样)。

若欲加载的文件不在 Pick 表中,则可选择"--Load file--"(Pick 表菜单上的最后一项),这时系统提示"--Load File Name--"输入框,用户就可键入要加载的文件名了(DOS 风格的通匹符也有效)。当然也可用 F3 热键来自动选择 File|Load。

Pick 文件

Pick 文件保存与文件相关的信息,包括 Pick 表的内容。对于 Pick 表中的每一项(文件), Turbo C 都保存了光标位置、标记块和标记符的信息。

此外, Pick 文件里还会有用户上次编辑时的状态数据。其中有最近一次查找一替换的字符光串和查找参数值。

为了创建 Pick 文件,用户必须先定义一 Pick 文件,这个可通过选择 Options|Directories |Pick File Name,然后键入文件名办到。一旦定义 Pick 文件名之后, Turbo C 就会在用户每次退出集成开发环境时更新 Pick 文件。

何时,怎样获得 Pick 文件

有两个菜单条目,用户可查一查便可获得有关该文件的信息:

Options|Directories|Pick File Name 和 Options|Directories|Current Pick File。

下面是一些常用问题的问答。

问:怎么才能知道是不是已经有了一 Pick 文件?

答:Options|Directories|Current Pick File 菜单设置不为空(含一文件名)即说明已经有了。

问:为何文件名出现在 Options|Directories|Current Pick File?

答:不是一文件名已被显式地列在 Options|Directories|Pick File Name 里了,就是用户已加载了一缺省 Pick 文件(若 Options|Directories|Pick File Name 设置为空)。

问:假设 Options|Directories|Pick File Name 设置显式地列出一文件名,那么是怎么到那里去的呢?

答:可通过如下方法在 Pick File Name 里获得一文件名:

- 此时自己键入;
- 以前键入,保存了配置文件,现在又使用了那个配置文件;
- 用 TCINST 安装;

问:假设 Options|Directories|Pick File Name 是空的,但 Options|Directories|Current Pick File 含有一文件名,那么该缺省 Pick 文件是怎样被加载的呢?

答:在当前目录或(若不在 Turbo C 目录)里有一缺省 Pick 文件 TCPICK. TP,而 Turbo C 在初启时自动加载了。

一旦 Pick 文件被加载了,集成开发环境就记住路径全名,该信息显示在 Options|Directories|Current Pick File 设置里。

Turbo C 何时保存 Pick 文件

无论何时退出集成开发环境,Turbo C 都会保存 Options|Directories|Current Pick File 指定的文件。另外,任何时候只要是 Pick 文件名被改变了(或者是从菜单直接输入一新名或间接地通过加载了一含不同 Pick 文件名的配置文件),Turbo C 首先保存已存在的 Pick 文件。

若 Options|Directories|Current Pick File 设置为空,则 Turbo C 就不保存 Pick 文件到盘上了。

第二章

使用 Turbo C 编辑器

本章主要介绍建立在 Turbo C 集成开发环境中的编辑器的使用,其操作与原 Turbo C 编辑器类似。在 Turbo C 编辑器中增加了如下内容:

1. 对鼠标器的支持;
2. 可用菜单执行一些文本搜索和替换操作;
3. 可建立和编辑两个或更多的文件。即使对原 Turbo C 编辑器比较熟悉的用户,在阅读完本章后也一定能了解到许多新的内容。

在 Turbo C 编辑器中含有约 50 条很强的功能及其命令,但读者没有必要立即掌握全部的命令,在掌握了最重要的插入、删除、块移动、搜索和替换等基本功能后,再去学习其它的编辑命令,并在需要的时候使用它们就很容易了。学习使用编辑器,因为可以随时调用 Turbo C 的联机上下文敏感 Help 系统,为此,方便、简单。

请在 DOS 提示符下键入

TC

可以立即启动 Turbo C。

2.1 编辑器命令

首先,要了解如何把命令传给 Turbo C 编辑器,几乎所有的编辑命令都以一个控制字符开始,有的字符后面还跟有其它字符序列。例如,CTRL-Q F 命令告诉编辑器搜索一指定的字符串(本书用缩写 CTRL 表示控制键)。执行该命令的方法是,同时按下 CTRL 控制键和 Q 键,然后再按 F 或 f 键。

尽管所有的编辑命令都可以从键盘上输入,但有一些是可以由菜单输入的,也有一些可用鼠标执行。菜单或鼠标器的使用在后面将具体说明。

2.2 编辑器激活及文本键入

当 Turbo C 开始执行时,编辑窗口是活动的。当激活主菜单执行菜单操作后,可以通过按 Esc 键返回到编辑窗口。

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

活动的编辑窗口未接收到命令时,处于准备接收输入的状态,即在键盘上敲入一个键

时,它们将出现在编辑当前光标位置处。

缺省情况下,编辑器处于插入方式,即输入的文本将被插入到光标处;相反,若处于覆盖方式,新文本将覆盖已存在的文本。通过按 Ins 键可在这两种方式之间进行切换,通过光标的形状可区分这两种方式。在插入方式中,光标用一个闪烁的下划线表示;在覆盖方式下,光标是一个闪烁的方块。

注意编辑窗口是活动的并键入下列行:

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

在键入中的错误,可用 BACKSPACE 键进行更正,屏幕将是图 2.1 中的情形。与光标位置相关的行、列显示在编辑窗口的左下方。此外,当文件被修改以后在还有一个星号出现在行、列指示器的左端。

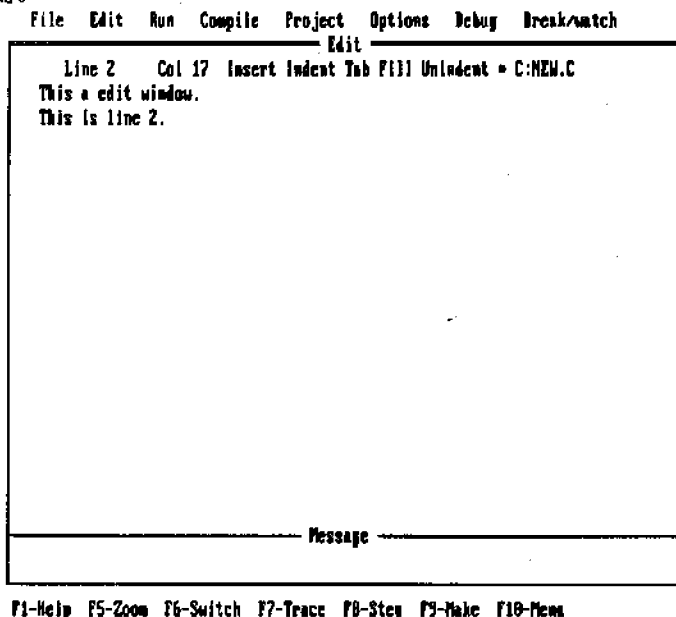


图 2.1 键入文本时的屏幕

Turbo C 编辑器是一个全屏幕编辑器,可以用光标键随意地在文本中移动光标。按鼠标时,光标将移到鼠标指示器的位置。此时,用方向键或鼠标把光标移至“test of the”行的最左端,键入 very small 并按 ENTER。做完之后,已有行被移到右边,这是编辑器处在插入方式下的结果。若你使编辑器处于覆盖方式,原行将被覆盖。屏幕将如图 2.2 所示。

2.3 字符、字和行的删除

可以用两种方法删除一个字符:即用 BACKSPACE 键或用 DEL 键。BACKSPACE 键删除光标左边的字符,而 DEL 键删除光标所在处的字符。

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

空格 \$ / - + * ' ^ [] () . ; , < >

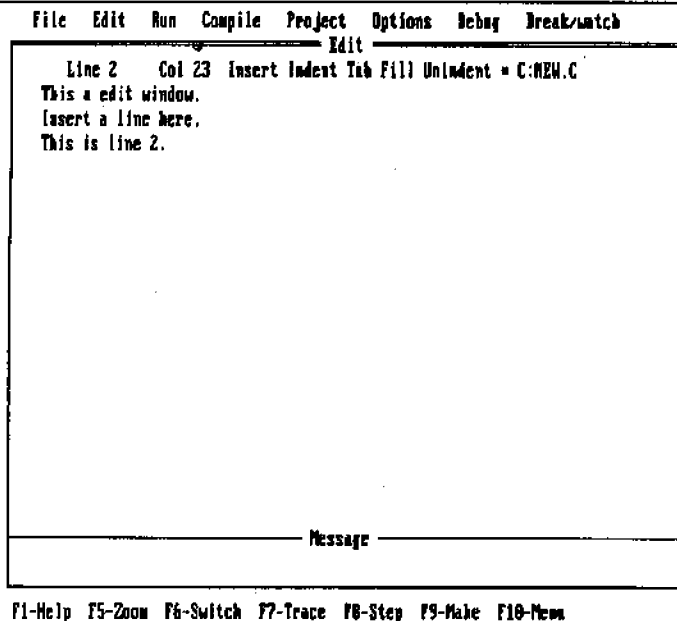


图 2.2 插入一行后的屏幕

可用 CTRL-Y 键删除一整行。无论光标处于该行中的什么位置都将删除一整行。若希望删除一行中从当前光标位置到行尾的内容,可以使用 CTRL-Q Y 键。

2.4 移动、拷贝文本和块移动

Turbo C 编辑器允许操作文本中的块,可以把它移动或拷贝到另一个地方或把它删除。进行这些操作,首先要定义块。该块可以短至一个字符或长至整个文件。定义一个块的方法有两种:用键盘或用鼠标。若用键盘定义一个块,须把光标移至块首并键入 CTRL-K B 键;然后,把光标移至块尾并键入 CTRL-K K,所定义的块将以高亮度显示。若使用鼠标定义一个块,首先把鼠标指示器置于块头;然后,按着鼠标左按钮并把鼠标移至块尾;最后,放开该按钮。

例如,把光标移至第三行之首“*”并键入 CTRL-K B;然后,把光标移至最后一行之尾并键入 CTRL-K K (或用鼠标),屏幕应为图 2.3 所示的情形。

若要移动文本中的块,则把光标置于移动后文本所在之处并键入 CTRL-K V。这将导致定义的文本块从其当前位置删去并置于新的位置。

若要拷贝一块,键入 CTRL-K C,屏幕如图 2.4 所示。现在可以用这些命令试一试。

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

可以通过把光标移至一个字的首字符之处并键入 CTRL-K T 来定义长度为一个字的块。

用 CTRL-K I 命令,可使一整块都凹进一个字符,其逆过程可用 CTRL-K U。

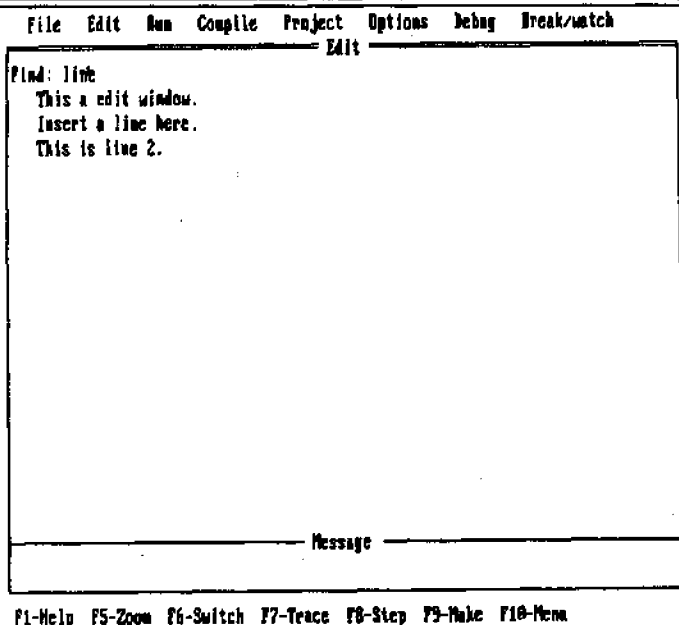


图 2.3 Find 提示行

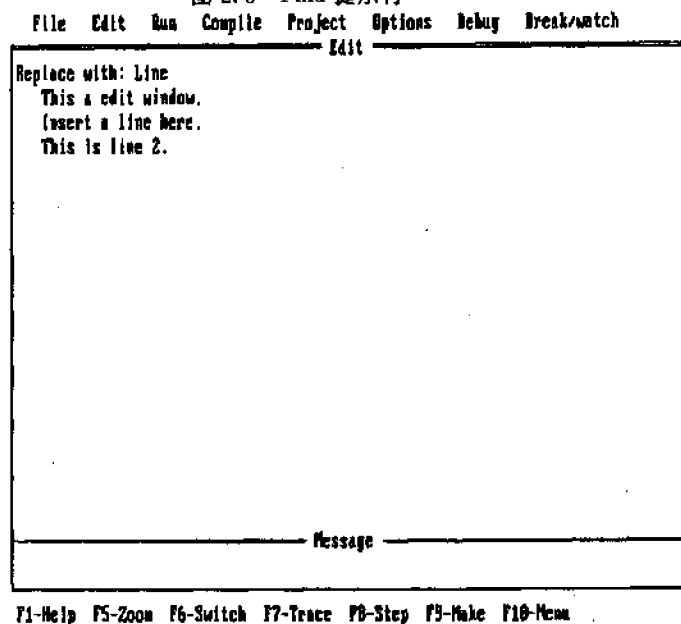


图 2.4 替换提示行

2.5 进一步说明如何移动光标

Turbo C 编辑器包括大量的特殊光标命令。表 2.1 中列出了这些命令,此外也可以使用置鼠标指示器到希望的位置并按键以移动光标的方法。

表 2.1 光标移动命令

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

2.6 字符系列的搜索和替换

用 CTRL-Q F 命令可搜索一指定的字符序列,如图 2.3 所示的 Find 提示行。可以指定不同的搜索选项或修改执行这些搜索选项。缺省选项是自当前光标位置向前搜索,对大小写敏感且允许子串匹配。下面具体说明各个搜索选项。

缺省情况下,从当前光标位置开始对文件向前搜索。选择 B 选项可实现反向搜索,通过选择 G 选项搜索覆盖整个文件。

在缺省情况下搜索,对大小写是敏感的,即大写字符和小写字符被当作不同的字符对待。若搜索串是“is”,则编辑器将在字“this”找到匹配。通过确认 W 可以实现整个字匹配搜索。

CTRL-QF 允许查找至多 30 个字符的字符串。输入这个命令后,清除状态行,编辑器显示输入这字符串,输入串后按回车。

查找串可含有任何字符,包括控制字符。输入控制字符时冠以 Ctrl-P,比如输入 CTRL...-T,可按 CTRL-P,T。在查找串中可包括换行符,用 CTRL-M J 指定。注意,Ctrl-A 有特殊定义,可匹配任何字符。

可用字符和单词移动命令编辑查找串,向右移动恢复前查找串,然后进行编辑。要放弃查找,使用 Ctrl-U 命令。

当输入查找后,Turbo C 编辑器询问查找选择项。定义了下述查找选项:

- B 从当前位置向后查找到文件首;
- G 不管当前光标,搜索整个文件,遇到最后匹配串时停止;
- L 查找块的下一个匹配串;
- N 从当前光标位置开始,匹配 N 个查找串后停止;
- U 忽略大小写;
- W 只查找单词,不匹配中含有相应查找串的情况。

示例:

- W 只查单词,比如查找串为 term 时,不匹配 terminal;
- BU 向后查找,忽略大小。block 匹配 Blockhead、BLOCKade 等;
- 125 匹配 125 个查找串。

CTRL-L 重复一个搜索过程,这使得当在文件中寻找某些特定的字符时尤为方便。

键入 CTRL-Q A 激活替换命令,即可用另一个串替换所寻找的串,它与搜索命令的用法相同。可见图 2.4 所示的提示行。

先查找,然后用另外小于 30 个字符的串替换查找的串。注意 Ctrl-A 有特殊意义,它匹配任何字符。

指定插查串后,编辑器要求输入替换串。电子出版系统多输入 30 个字符,控制字符和编辑方法如同在查找命令中指出的一样。只按回车,编辑器删除查找到的串。串选择项比查找命令增加了几个:

- /N 没有询问替换,替换时不再确证。
- n 替换几次。如果使用了 G 选择项,从文件开始,忽略 N;否则从当前光标位置开始。
- L 只替换块内的字符串。

示例

- /V10 没有提示,共替换 10 处。
- GW 在整个文件中查找并替换,忽略大小写,替换时提示。
- GNU 在整个文件中查找并替换,忽略大小写,替换时没有提示。

输入选择项后按回车,查找和替换开始。当发现匹配时,如果/N 没有指定,编辑器把光标置于匹配字符后,在屏幕窗口提示 Replace(Y/N)?。此时,可用 Ctrl-U 放弃查找/替换;也可用 Ctrl-L 重复查找/替换操作。

可以通过首先键入 CTRL-P,后面跟控制字符来把控制字符输入到搜索串。

2.7 位置标识的设置和搜索

位置标识符在大文件中使用是很方便的,键入 CTRL-K N 文件可设置多达四个位置标识符,其中 n 是位置标识符的序号(0~3)。设置一个位置标识符之后,可使用命令 CTRL-Q n(这里 n 为标识符号)移光标至该标识符所指示的行。

2.8 存储和装入文件

存储文件的方法有以下三种,前两种是把文件存入以编辑窗口标题为名的文件。第三种方法是把文件存在另一个文件中,然后使其成为当前编辑文件。下面具体说明这一种方法的实现过程。

首先,按 F10 退出编辑器并返回主菜单,然后选择 File 选项。由于 Save 选项将当前编辑器中的内容存放在一个磁盘文件中。此文件具有与窗口标题相同的文件名。所以在未指定待编辑文件的文件名之前,激活 Save 选项时,文件存入 NONAME.C 中,Turbo C 将提示输入一个别的文件名;仅当 NONAME.c 是一个源文件名时,该提示才出现。否则,文件的存储将不经进一步的交互提问。

若希望把编辑器中的内容存入与编辑器状态行上文件名不同的文件中,则可使用 Write to 选项。这时要求输入希望存入的文件名,按回车键后就将该文件存盘,按 F2 键还可以从编辑器中存文件。这与 File 菜单中的 Save 选项相同。

若要装入一个文件,可在编辑窗口中按 F3 或从 File 菜单选择 Open 选项。该操作将显示一个对话框,提示输入欲装入的文件名。指定文件名有两种方法;其一,键入文件名;其二,在对话框中显示的文件列表中选择。缺省时,所有以 C 为扩展名的文件都被显示。

如果存储一个磁盘上已有的文件时,文件的旧版本不被覆盖,而被存入扩展名为 .BAK 的备份文件中。

2.9 自动缩进

使用缩进可使书写的程序清晰并易于理解,Turbo C 对缩进书写提供了方便。按 ENTER 之后,Turbo C 编辑器将自动把光标移动至上一行第一个字符所在的列上,当然此时自动缩格选项必须有效。按 CTRL-QI 使之有效或无效。严格按照下列显示的格式输入下列行,就可实际体会一下自动缩格的工作过程:

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

输入该文本之后,注意 Turbo C 是如何自动完成缩格的。

再按 CTRL-Q I 可使自动缩格无效。

2.10 磁盘文件中文本块的移入和移出

当需要把一个文本块移到一个磁盘文件备用时,可以先定义一个块,并按 CTRL-Q W 来完成。完成上述操作之后,系统将提示输入希望存放该块的文件的文件名。文本的源块不从程序中删除。

键入命令 CTRL-Q R 即可读入一个块,系统将提示输入文件名,该文件的内容将被读入在当前光标位置处。

使用这两条命令可在两个或多个文件中移动文本。

2.11 对匹配

C 中有几组界符是成对的,例如,{ }、[]和()。在很长或很复杂的程序中,手工去寻找一个界符的配对是很不方便的,而用编辑器自动寻找相应的配对界符是可能的。

Turbo C 编辑器提供自动搜索如下界符对的功能:

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

若要搜索某匹配界符,先把光标放在你希望搜索的匹配的界符处,然后键入 CTRL-Q [进行向前匹配;用 CTRL-Q],进行向后匹配)。

对于界符{ },[],(),< >可以是可嵌套的,有时候也包括注释符(当选择了嵌套注释选项时)。编辑器将按 C 方法搜索适当的匹配符。若因某种原因编辑器找不到适当的匹配界符,光标将停止不再移。

2.12 其它有关命令

可以在提示符下按 CTRL-U 或 ESC 以终止请求输入的任何一条命令,或在屏幕上对话框之外的任何地方按鼠标器终止之。例如,若在执行 Find 命令时,想取消此命令,则可按 ESC 或在框外按鼠标器。

若要在光标移开一行之前恢复对该行所做的修改,则按 CTRL-A L。可以通过选择 Edit 菜单中的 Rrestore line 来消除对一行的修改。但是一旦光标移开了该行,所有对该行的修改都被确认。

若要移至块首,则输入 CTRL-Q B;要移至块尾,则输入 CTRL-Q K。

打印文件可用 CTRL-O K P 命令,若未定义块,打印整个文件,否则将仅打印一块。

要把光标移至其原先的位置要用 CTRL-Q P 命令。此命令可能搜索某个对象然后返回原来所在之处。

按 TAB 键时,输入一个制表字符到文件中,此外,用命令 CTRL-Q T 可将等量的空格插入一个制表符。以上是 CTRL-Q T 命令变换处理制表符的两种方法。

若在一行行首按 BACKSPACE 键时,光标将自动移至缩格处,用 CTRL-Q U 命令可改变这一功能。当它无效时,每次按 BACKSPACE 键,光标将只回退一个空格。

2.13 命令综述

所有的 Turbo C 编辑命令见下表 2.3。

表 2.3 Turbo C 编辑命令

功 能	命令功能
(1)光标命令	
左移一个字符	左箭头或 CTRL-S
右移一个字符	右箭头或 CTRL-D
左移一个字	CTRL-A
右移一个字	CTRL-F
上移一行	上箭头或 CTRL-D
下移一行	下箭头或 CTRL-X
上翻	CTRL-W
下翻	↓ CTRL-Z
上翻一页	PGUP 或 CTRL-R
下翻一页	PGDN 或 CTRL-C
移至行首	HOME 或 CTRL-Q S
移至行尾	END 或 CTRL-Q D
移至屏顶	CTRL-Q E
移至屏底	CTRL-Q X
移至文件头	CTRL-Q R
移至文件尾	CTRL-Q C
移至块首	CTRL-Q B
移至块尾	CTRL-Q D
移至原光标位置	CTRL-Q P
(2)插入命令	
选择插入方式	INS CTRL-V
插入一空行	ENTER CTRL-N
(3)删除命令	
整行删除	CTRL-Y
删除至行尾	CTRL-Q Y
删除左边字符	BACKSPACE
删除光标处字符	DEL 或 CTRL-G
删除右连字符	CTRL-T
(4)块命令	
标志块首	CTRL-K B
标志块尾	CTRL-K K
标志一个词	CTRL-K T
拷贝一个块	CTRL-K C
删除一个块	CTRL-K Y

隐蔽或显示一个块	CTRL-K H
移动一个块	CTRL-K V
把一个块写入磁盘	CTRL-K W
从磁盘中读出一个块	CTRL-K R
缩格一个块	CTRL-K I
退格一个块	CTRL-K U
打印一个块	CTRL-K P
(5)搜索命令	
搜索	CTRL-Q F
搜索和替换	CTRL-Q A
搜索一个位置标识符	CTRL-Q(NUM)
重复搜索	CTRL-L
(6)对匹配	
向前匹配对	CTRL-Q[
向后匹配对	CTRL-A]
(7)其它命令	
中断	CTRL-U 或 ESC
变换自动缩格方式	CTRL-O I
控制字符前缀	CTRL-P
退出编辑器	F10
新文件	F3
存储覆盖错误信息	CTRL-Q W
存储	F2
设置一个位置标识符	CTRL-K(NUM)
变换制表方式	CTRL-Q T
撤消修改	CTRL-Q L
变换退格方式	CTRL-O U

2.14 用文件激活 Turbo C

激活 Turbo C 时,不仅可以通过指定要编辑的文件名,还可以在命令行中“TC”之后键入文件名来完成这一过程。例如,在“TC”之后输入 MYFILE 将执行 Turbo C 并使 MYFILE.C 装入编辑器中。扩展名.C 将自动地由 Turbo C 加入。若 MYFILE.C 不存在,则将被建立。不希望在文件名中用扩展名时可在该名之后加上一个句号(.)。

第三章

Turbo C 调试器

在用户编写程序的过程中,定位并改正所出的错误是一项很复杂的工作。大多数有经验的程序员都承认跟踪程序中的逻辑问题是程序开发过程中的一个重要环节。调试、发现并定位程序中的错误所花费的时间,有时甚至比编写程序所花的时间更长。

Turbo C 带有一个源程序级的集成调试工具,它为用户在集成环境中进行调试提供了方便。同时,Borland 公司还提供了—个具有许多功能的独立调试程序。

源程序级调试的含义是可以对程序源代码进行跟踪。有兴趣的话,还可以跟踪用户在程序中调用的每个函数。通过设置断点,便可以控制在进行条件检查前运行哪一步,之后,通过用光标选择变量或在 Evaluate 区域中直接键入变量名,可得到该变量的当前值。也可以通过 watch 窗口来监视一个或多个变量,程序运行时随时显示其值的变化。

键入 TC 启动 Turbo C(如果还未运行的话),然后检查 Debug 菜单。用光标依次选择菜单上的每一项,并按 F1 获得帮助,阅读相应的帮助信息。其中所有的菜单选择项都是调试工具的一部分,在本章后面的示例程序中将会学会如何使用它们。

3.1 调试与程序开发

当本章集中讨论调试时,很重要的一点是不要把调试和设计、编写程序隔离开来对待。虽然 Turbo C 的集成调试工具会尽可能地发现并定位程序中的错误,但如果用户用良好的方法来编写程序,同样会使调试变得更容易些。

良好的程序设计方法就像使用晶体管和集成电路来设计电视。每块电路放在各自的硅片上,作为一个模块,可以很方便地进行访问和测试。要替换出故障的模块,也能很容易地完成。

在程序中,与这些插入模块等价的就是函数。人们常常在编写完一个完整的程序后(尤其是在这个程序较小时),才开始调试它。如老式电视一样,它给问题的解决带来一些困难。下面来举一个例子:假设 main()调用函数 b(),而函数 b()中又依次调用函数 c()和 d()。如果在编写每个函数时,没有对其进行调试,就难以确定函数 b()中出现的错误是由于其本身的代码出错,还是 c()或 d()中的代码出错。所以,这将比调试一个单独的函数困难得多。同时,看起来不相关的另一个函数 a()也可能修改函数 b()用来传递给 c()的全程变量。可见循序渐进的程序开发——每次开发和测试程序中的一部分——可以节省用户相当多的时间并使用户免遭失败。

3.2 设计示例程序: PLOTEMP. C

本章的示例程序读取收集的温度,将数据传递给一个函数和一些图形制表函数以表格或图形的方式显示出来。如果愿意,可以修改 PLOTEMP. C,以便处理另一组数据,并给出一组不同的报告和图表。

设计程序最好的办法之一是给出函数原型,列出该函数与用户的接口要求什么样的参数和信息,以及完成的功能。在看程序编码前,先从用户的角度来考虑该程序的运行情况。

运行 PLOTEMP. C 时,先显示菜单:

```
Temperature Plotting Program Menu
E — Enter temperatures for scratchpad
S — Store scratchpad to disk
R — Read disk file to scratchpad
T — Table view of current data
G — Graph view of current data
X — Exit the program
Press one of the above keys:
```

如果按 E,将提示用户输入一组温度值。该程序设置为处理 8 个输入值,但用户可以很容易地修改该值。

数据输入方式如下:

```
Enter temperatures, one at a time.
Enter reading #1: 52
Enter reading #2: 55
Enter reading #3: 62
Enter reading #4: 65
Enter reading #5: 73
Enter reading #6: 76
Enter reading #7: 68
Enter reading #8: 61
```

完成该工作,将再次进行菜单选择。

选取 S 将当前内存中的数据存入一磁盘文件(将提示输入文件名)。选择 R 将从用户指定的磁盘文件中读取一组数据并放入该程序的“scratchpad”数组中。

一旦用户在 scratchpad 中放入了数据(无论是从键盘输入还是从磁盘中读取),就可以选取 T(即 Tableview)来显示这些数据的一个综合表格,也可选取用图表方式来显示当前数据。

```
Reading Temperature (F)
1 52
2 55
3 62
4 65
5 73
```

```
6 76
7 68
8 61
Minimum temperature:52
Maximum temperature:76
Average temperature:64.000000
```

这里如果存在问题,即当这个程序不能正确地完整运行时,就必须使用调试工具来查找并定位程序中的错误。

3.3 编写原型程序

确定了该程序的功能以后,用户就可以定义该程序的数据结构,并且编写 main 函数。

首先给出原型程序如下:

```
#include <conio.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
/* Prototypes */
void get_temps(void);
void table_view(void);
void min_max(void);
void avg_temp(void);
void graph_view(void);
void save_temps(void);
void read_temps(void);
/* Global defines */
#define TRUE 1
#define READINGS 8
/* Global data structures */
int temps[READINGS];
int main(void)
{
    while (TRUE)
    {
        printf("\nTemperature Plotting Program Menu\n");
        printf("  E — Enter temperatures for scratchpad\n");
        printf("  S — Store scratchpad to disk\n");
        printf("  R — Read disk file to scratchpad\n");
        printf("  T — Table view of current data\n");
        printf("  G — Graph view of current data\n");
        printf("  X — Exit the program\n");
        printf("\nPress one of the above keys:");
```

```
switch (toupper(getche()))
{
    case 'E': get_temps(); break;
    case 'S': save_temps(); break;
    case 'R': read_temps(); break;
    case 'T': table_view(); break;
    case 'G': graph_view();
    case 'X': exit(0);
}
}
}

/* Function definitions */
void get_temps(void)
{
    printf("\nExecuting get_temps().\n");
}
void table_view(void)
{
    printf("\nExecuting table_view().\n");
}
void min_max(void)
{
    printf("\nExecuting min_max().\n");
}
void avg_temp(void)
{
    printf("\nExecuting avg_temp().\n");
}
void graph_view(void)
{
    printf("\nExecuting graph_view().\n");
}
void save_temps(void)
{
    printf("\nExecuting save_temps().\n");
}
void read_temps(void)
{
    printf("\nExecuting read_temps().\n");
}
```

关于原型程序应注意以下几点：

- 全程的宏定义和数据结构定义(数组 temps)。
- 函数 main 提供了顶层菜单。

- 别的函数说明为返回 void 类型,不带参数。
- 每个函数包括一条 printf 语句,以便表明它正在运行。

用 void 说明这些函数是为了使用户能在高层执行程序,并对流程进行检查。如果用户给出这些函数的完整的 ANSI 原型,并带上不同数据类型的参数,就不得不重新开始定义函数,并编写代码来使用传递到函数的参数。否则,编译时将得到关于这些参数未被使用的警告信息。一个好的程序开发原则是“一次做一步,一次测试一步”。基于这一点,使用原型程序来试验该程序的高层结构是否正确。

3.4 使用集成调试工具

选取 Compile | build All 编译 PLOTEMP1.C。编译程序在显示如下错误信息后停止运行。

```
Error C:\TC\EXAMPLES\PLOTEMP1.C 43:  
statement missing; in function main
```

Turbo C 发现一语法错误。用户一般在实际运行程序前就应该检查一遍看有无语法错误,而不必急于开始进行调试。幸而,语法错误通常易于定位。按空格键:此时,错误光条停在显示菜单的 switch 语句的第一行,少一个分号的错误通常是指上一条语句(这里是指前面的一组 printf 语句中的最后)。这时用户可以按 Enter 或 F6 回到编译窗口来改正错误。如果有多个语法错误要处理,再按 F6 回到信息窗口,然后使用 Alt-F8(或↓键)移到下一个错误上。

修正了语法错误后,再次编译该程序。这次编译和连接的完整运行表明该程序已无语法错误。

现在选取 Run | Run,运行 PLOTEMP1.EXE。用户将看到该程序的菜单,试着按菜单上列出的各个键,检查该程序的流程。

用户可能注意到,对于菜单的每一个选择,都显示相应函数中的打印信息(记住那些子函数中的 printf 语句)。然后,再次显示该菜单。如果按 X 或(x),则程序结束,返回到 Turbo C。但是如果按字母 G(或 g),同样也返回到 Turbo C。这是不正确的,因为此时用户希望作相应的处理,然后继续进行菜单选择。这说明程序中的某个地方存在错误。

3.5 跟踪程序的流程

使用 Run 菜单上的选择项,通过观察运行光标条,用户可以观察到程序中语句的执行顺序,还可以控制跟踪的细节。选取 Run | Trace Into(或按 F7),则在编辑窗口中 main 函数的开始处出现一个高亮度光条。该高亮度光条称为运行光条,标明正在执行的位置。

3.5.1 跟踪高层的运行

为跟踪程序的主流程,选取 Run | Step over(或按 F8),则下一行包含的代码被执行(跳过注释行)。当继续按 F8 时,运行光条移过一系列的用于显示菜单的 printf 语句,而且屏幕出现闪烁。这是因为每当调试器执行一条在屏幕上显示信息的语句或执行一个函数调用时,

它都要立即转到用户屏幕(User screen 该屏幕显示用户程序产生的信息),然后又立即回到 Turbo C.,但是屏幕切换进行得非常快,以至于无法看清输出。为了查看用户屏幕中的信息,按 Alt-F5(或选取 Window | User Screen)。依据所执行 printf 语句的多少,用户将看到部分或全部的 PLOTEMP 菜单。按任意键回到调试程序。

继续按 F8,直到运行到 switch 语句的第一行。该行包含一个对 getche()函数的调用,它要求用户输入一个字符。每当程序要求用户输入时,Turbo C 都切换到用户屏幕。由于用户想观察选择图形显示时会发生什么情况,再按 F8,然后输入 G,显示返回到调试程序,并且运行光标移动到下一行语句:

```
case 'G': graph_view();
```

到此,一切正常。但是,若再按一次 F8,走到下一步,下一条语句是 exit(),表示用户再运行一步将结束程序并返回到编辑程序。按 F8 键试验完毕。至此,用户也许已注意到在上一行中需要一个 break 语句,修改该行为:

```
case 'G':graph_view(); break;
```

重新编译运行程序。选取 Run | Go to Cursor(或按 F4)。将程序运行到需要用户输入的一行,输入 G 作为 PLOTEMP 菜单的选择后,继续单步运行,注意到这次 break 语句被执行,然后执行位置回到 while 循环的顶部:再次显示 PLOTEMP 菜单,操作正确。

3.5.2 跟踪子函数

当用户使用 Run | Step Over 时,仅仅是在程序的顶层作单步执行。正如用户所见到的,运行光标条停留在 main 函数,并单步通过 printf 语句和 switch 语句中的各个 case 语句。然而,用户常常需要跟踪被 main 函数调用的函数,有时还有被这些函数调用的函数。为了跟踪函数调用,选取 Run | Trance into 或按 F7。

现在来试着跟踪该程序,在 PLOTEMP 菜单中按 E。这时运行光标条停在 switch 语句中相应的 case 语句上,然后按 F7 进入函数 get_temps()中。单步执行完该函数(这里仅有一条 printf 语句),返回到 main()中 while 循环的底部,继续运行将再次显示菜单。

3.6 继续程序的开发

在本章剩余的部分中,将每次向 PLOTEMP 中加入一个函数,并通过运行该程序来测试它。下面提供了 PLOTEMP 的改进版本,能够装载完成每一步的功能。如果要调试自己的程序则按如下的步骤进行:

1. 必要的话,用完整的函数代换函数原型。
2. 用执行任务的实际代码来代换已有的函数定义。
3. 通过再次运行该程序测试新函数,注意在菜单中选取相应的项。
4. 修改出现的任一错误,直到没有错误为止。
5. 以同样的方法完成下一个函数,直到完成一个完整的程序。

为最快、最佳地完成任任务,应分别编制、编译、运行和调试每个函数。不要仅在当前工作程序无表面错误时就开始开发下一个函数,这样可能不会消除所有的错误,因为一些隐藏错误仍继续潜伏着,并可能会在运行时产生一些不能预见的结果。改进方法能使这些不希望的

破坏减到最少。

从 `get_temps()` 函数开始, 它从键盘输入读取一组温度。由于它不带参数, 也不直接返回任何值, 因此, 当前函数原型说明为:

```
void get_temps(void);
```

找到已有的 `get_temps()` 的定义。目前, 它在执行时仅仅打印一条信息。用下面程序代替原来的代码, 存入 `PLOTEMP2.C`:

```
/* PLOTEMP2.C—Example from Chapter 7 of Getting Started */
/* This program creates a table and a bar chart plot from a set of
temperature readings */
#include <conio.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
/* Prototypes */
void get_temps(void);
void table_view(void);
void min_max(void);
void avg_temp(void);
void graph_view(void);
void save_temps(void);
void read_temps(void);
/* Global defines */
#define TRUE 1
#define READINGS 8
/* Global data structures */
int temps[READINGS];
int main(void)
{
    while (TRUE)
    {
        printf("\nTemperature Plotting Program Menu\n");
        printf("  E — Enter temperatures for scratchpad\n");
        printf("  S — Store scratchpad to disk\n");
        printf("  R — Read disk file to scratchpad\n");
        printf("  T — Table view of current data\n");
        printf("  G — Graph view of current data\n");
        printf("  X — Exit the program\n");
        printf("\nPress one of the above keys:");
        switch (toupper(getche()))
        {
            case 'E': get_temps(); break;
            case 'S': save_temps(); break;
```

```

        case 'R': read_temps(); break;
        case 'T': table_view(); break;
        case 'G': graph_view(); break;
        case 'X': exit(0);
    }
}

/* Function definitions */
void get_temps(void)
{
    char inbuf[130];
    int reading;
    printf("\nEnter temperatures, one at a time. \n");
    for(reading=0; reading<READINGS; reading++)
    {
        printf("\nEnter reading # %d:", reading+1);
        gets(inbuf);
        sscanf(inbuf, "%d", &temps[reading]);
        /* Show what was read */
        printf("\nRead temps[%d]=%d", reading, temps[reading]);
    }
}

void table_view(void)
{
    printf("\nExecuting table_view(). \n");
}

void min_max(void)
{
    printf("\nExecuting min_max(). \n");
}

void avg_temp(void)
{
    printf("\nExecuting avg_temp(). \n");
}

void graph_view(void)
{
    printf("\nExecuting graph_view(). \n");
}

void save_temps(void)
{
    printf("\nExecuting save_temps(). \n");
}

void read_temps(void)

```

```
{  
    printf("\nExecuting read _temps(), \n");  
}
```

3.7 设置断点

如果 `get_temps()` 工作正确, 则 `for` 循环在每次读数时给出提示, 然后使用 `gets()` 以字符串的形式读取一个值。然后, 用 `sscanf()` 将其存入数组 `temps` 的相应元素中(注意, C 中数组从第 0 个元素开始: 第一次处理第 0 个元素, 第二次对应于第 1 个元素, 等等), 并用另一个 `printf` 语句显示存入数组的值。该 `printf` 语句仅是一个临时的方法: 用于检测存储的数据是否正确, 该函数调试通过后应将其删除。

运行 `PLOTEMP2`, 并在 `PLOTEMP` 菜单中按 `E` 执行 `get_temps()`。在用户输入数据时, 会出现如下情况:

```
Enter reading #1:40  
Read temps[0]--0  
Enter reading #2:50  
Read temps[1]--0  
Enter reading #3:55  
Read temps[2]--0  
Enter reading #4:57  
Read temps[3]--0  
Enter reading #5:61  
Read temps[4]--0  
Enter reading #6:64  
Read temps[5]--0  
Enter reading #7:65  
Read temps[6]--0  
Enter reading #8:60  
Read temps[7]--0
```

可以看到, 不管用户输入什么数据, `temps` 数组中相应元素总保持为 0。有可能是被输入的数据在放入数组时采用转换方法不对。

显然, 该函数需要进一步的测试。为了将程序运行到可能出问题的区域, 可以设置一个断点。断点是程序中希望运行到此就停止的位置。将光标移到 `get_temps()` 函数中循环的开始处:

```
for(reading=0; reading<READINGS; reading++)
```

然后选取 `Break/Watch | Toggle Breakpoint` (或按 `Ctrl-F8`), 在此行设置一个断点。可以看到这行上出现一个高亮光条。任何时候, 当程序执行到这一行时都被中断, 并返回到 Turbo C 调试器。这时允许使用别的命令来查看和修改变量及其它数据结构。设置多个断点时, 可以选取 `Break/Watch | View next Breakpoint` 选项查看下一个点断。

断点设置将保持到用户做以下操作时才被消除:

- 离开该集成环境。

- 用 Ctrl-F8 消去一个断点。
- 使用 Break/Watch | Toggle Breakpoints 删除断点。
- 删除设置断点的行。
- 编辑包含断点的文件,不存就放弃该文件。

每当用户修改一个错误或编辑当前文件,然后继续使用调试命令时,Turbo C 将提问“Source modified, rebuild?”通常应在该提示下按 Y,以便程序被重新编译,这样所作的修改才有效。如果按 N,则断点和运行光标条都可能会出现在错误的位置上,因为源程序已不再和执行程序保持一致了。

3.7.1 用 Ctrl-Break 立即中断

也许用户知道按 Ctrl-Break 可以中断许多正在运行的程序。对于 TurboC 集成调试器也是如此。然而,调试器通常不会立即停止程序的执行,而是要等到用户源程序中一行所对应的机器码被执行完后,然后停止在程序中下一行开始处所对应的机器指令上。运行光条则出现在将要执行的行上。

如果确实需要立即中断,可以连按两次 Ctrl-break。当检测到第二次按键,调试器立即终止程序,不再作任何输出或调用退出函数。(这和使用 `_exit()` 函数一样)。这种“二次 Break”一般是不使用的,因为这时数据文件的内容是不可预知的,且调试器也不知道接下来应执行哪一行。通常仅仅是在程序“挂起”或陷入无限循环时才使用第二个 Break。

再次运行程序,显示 PLOTEMP 菜单,按 E。程序运行进入 `get_temps()` 函数,直到设置断点的行。用 F8 单步执行,直到运行光条移过 for 循环体中的语句再次回到循环的第 1 行。现在来看关键变量以便了解错误的起因。

3.8 计算和修改变量

装入 PLOTEMP2.C,编译,并单步运行到函数 `get_temps()` 中最后一个 `printf` 语句。现在将执行一次 for 循环,并且输入、格式化和存入一个数据(通过 `gets()` 和 `sscanf()` 函数)。现在,选取 Debug | Evaluate,便弹出一个包含以下三个区域的窗口:

- 求值区域,包含用户要查看的表达式。
- 结果区域:显示求值区中表达式的值。
- 新值区域:可输入被选表达式的新值。

缺省时,光标所在“单词”(可能是 C 变量、关键字,函数调用等)显示在求值区。假如用户要查看以下两个变量:reading 循环变量和数组 temps 中的输入数据。键入 reading 并回车,在结果区显示 0。如果用户单步执行循环中的语句,则 reading 的值为 1。

同样,用户也可检查更复杂的数据结构如数组、字符串和结构的值。如果,想进一步了解数组 temps 的情况,在求值区中输入 temps,则显示:

```
{0,0,0,0,0,0,0,0}
```

它表示当前存入数组 temps 中的整数值,通过使用下标也可以获得单个值。例如,指定 `temps[0]`,便可取得第一个整数。或者,也可以打开一个 Inspector 窗口。

注意,这里指的是表达式,而不仅仅是它们的值(如变量)。表达式是一些变量、常量和运

算符的组合,并产生一个简单的值,例如,vals1[index]+vals2[index]+1。用户可以显示任何表达式的值,但要求:

- 不包含函数调用,例如类似 sqrt(2)+1 的表达式。
- 不能用 #define 的值,例如当前程序中的 READINGS。

作为练习,可以分别在求值区中输入以下表达式,计算其值:

```
reading+2  
temps[reading+1]
```

3.8.1 指定显示格式

用户可以在要显示的表达式中加入一个逗号及一个格式指示符。例如,输入 reading,h 可看到 reading 当前值的十六进制数表示(也可输入 reading,x)。缺省时,整数以十进制形式显示,且字符数组显示为字符串。

处理数组时指示符 m 很有用:它可从指定地址开始显示内存内容。例如,temps,m 可得到从指定位置开始的内存内容(由于 temps 是一个数组,它只是指向被存入数据的起始地址):

```
00 00 00 00 00 00 00
```

这表明数组 temps 的所有元素当前被设置为 0。所显示的元素个数依赖于该数组的长度。也可将 m 与别的指示符组合使用:

temps,mh 以十六进制的格式显示内存数据。

另一个有用的指示符是 P,它显示一指针变量,给出有关所指内存区域的信息(例如:中断向量表中 BIOS 数据区,或用户程序的程序段前缀[PSP])。如果所指向的地址在程序自己被分配的内存中,则该地址处的变量名也被显示。。

3.8.2 指定值的个数

当处理一个数组时,用户可以指定要显示的个数,例如:temps,5 指定数组 temps 的前 5 个元素。也可将计数与格式指示符组合使用,例如:temps[2],3h 指定以十六进制形式显示从数组 temps 的第三个元素开始的 3 个元素。

别的格式指示符和调度的细节可以在联机帮助中得到提示。

3.8.3 从光标所在位置拷贝

也许用户早已注意到,当选取 Debug | Evaluate 时,求值区显示光标所在位置的任何单词,于是,用户便可以不再从键盘输入。例如,将光标移到表达式

```
temps[reading]
```

处,按 CTRL_F4,则在求值区显示 temps。当用户按→时,显示 temps 后的字符。这样便可将完整的表达式 temps[reading]拷贝到求值区,然后可以和原来一样进行剩下的操作。

3.8.4 查看在别的函数中的变量

到现在为止,查看的是函数 get_temps()中的变量 reading 和 temps。用户也可以查看在别的函数中的静态变量的值,因为静态变量即使在其函数未被执行时,其值也是被保存的。

可以查看正在执行的函数中的自动变量,但不能查看在别的函数中说明的自动变量,因为当退出该函数后,这些自动变量便不再存在。被求值的表达式可通过编辑窗口中当前光标位置给出。

要查看当前函数之外的一个变量,可将光标移入该函数体,或输入函数名、圆点、变量名。例如:

```
module2.getvals.count
```

3.8.5 修改值

现在用户已知道怎样查看不同类型的变量和表达式,以及用不同的形式显示其值。继续单步运行 `get_temps()` 中的 `for` 循环,检查 `reading` 和 `temps[reading]` 的值,将发现后者一直保持为 0,与 `reading` 的值无关。由于该表达式被指定为 `sscanf()` 函数的存储目标,所以它应是一个地址,于是,这意味着所有输入的值被存储在地址 0 上。用户可以通过使用指针格式 `temps[reading],p` 来验证这一点,发现其值仍为 0。

现在用户需要使用地址运算符 `&` 来查询 `temps` 数组的地址。计算 `&temps[reading],p`。其结果形如: `DS:278 temps+1`。实际显示的值依赖于系统结构和 `reading` 的当前值,但仍可发现 `&temps[reading]` 指向数据段中的一个实际地址,其偏移量由变量 `temps` 给出。

将 `sscanf` 语句中的表达式 `temps[reading]` 改为 `&temps[reading]`。现在,如果继续单步运行该程序,调试器将询问用户是否“rebuild the program”,按 Y。现在,再次单步运行 `for` 循环,并查看 `temps[reading]` 的值,将发现输入的数据都被正确地存入数组了。

这是一次练习用调试器修改数据的好机会:检测 `temps[reading]` 的当前值,然后用 Tab 键在三个区域中移动(Evaluate, Result 和 New Value 三个区)。一旦进入 New Value 区,就可以输入一个新值,如 66,并按 Enter。此时,这个新值(66)显示在 Result 区中。用户可以修改任何表示一个简单数据元素的值,如简单变量、指针或一个数组元素。

用调试器修改内部值对于临时定位一个错误或继续执行一段程序找下一个错误是很有用的。例如,可以输入从 `temps[0]` 到 `temps[7]` 的新值,并设置 `reading` 为 8,跳出 `for` 循环,返回到程序的主菜单中。也可以强制一个函数返回一个指定的值,再将该值传送给另一个函数。这可帮助用户测试导致非常情况的错误,而不必在程序中加入临时的赋值或打印语句。

现在 `get_temps()` 函数已能正确工作。下一步来完成 `table_view()` 函数,显示输入的数据。用下面的 `table_view()` 来替代原来的原型函数:

```
/* PLOTEMP3.C -- Example from Chapter 7 of Getting Started */
/* This program creates a table and a bar chart plot from a set of temperature readings */
#include <conio.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
/* Prototypes */
void get_temps(void);
void table_view(void);
void min_max(void);
void avg_temp(void);
```

```

void graph _view(void);
void save _temps(void);
void read _temps(void);
/* Global defines */
#define TRUE 1
#define READINGS 8
/* Global data structures */
int temps[READINGS];
int main(void)
{
    char choice;
    while (TRUE)
    {
        printf("\nTemperature Plotting Program Menu\n");
        printf("\tE—Enter temperatures for scratchpad\n");
        printf("\tS—Store scratchpad to disk\n");
        printf("\tR—Read disk file to scratchpad\n");
        printf("\tT—Table view of current data\n");
        printf("\tG—Graph view of current data\n");
        printf("\tX—Exit the program\n");
        printf("\nPress one of the above keys:");
        choice=toupper(getch());
        switch (choice)
        {
            case 'E': get _temps(); break;
            case 'S': save _temps(); break;
            case 'R': read _temps(); break;
            case 'T': table _view(); break;
            case 'G': graph _view(); break;
            case 'X': exit(0);
        }
    }
}

/* Function definitions */
void get _temps(void)
{
    char inbuf[130];
    int reading;
    printf("\nEnter temperatures, one at a time. \n");
    for(reading=0;reading<READINGS;reading++)
    {
        printf("\nEnter reading # %d:",reading+1);
        gets(inbuf);
        sscanf(inbuf, "%d", &temps[reading]);
    }
}

```



```

/* Show what was read */
printf("\nRead temps[%d]= %d", reading, temps[reading]);
}
}

void table_view(void)
{
    int reading;
    clrscr(); /* clear the screen */
    printf("Reading\t\tTemperature(F)\n");
    for(reading=0; reading<=READINGS; reading++)
        printf("%d\t\t\t\t\t%d\n", reading+1, temps[reading]);
    min_max();
    printf("Minimum temperature: \n");
    printf("Maximum temperature: \n");
    avg_temp();
    printf("Average temperature: \n");
}

void min_max(void)
{
    printf("\nExecuting min_max(). \n");
}

void avg_temp(void)
{
    printf("\nExecuting avg_temp(). \n");
}

void graph_view(void)
{
    printf("\nExecuting graph_view(). \n");
}

void save_temps(void)
{
    printf("\nExecuting save_temps(). \n");
}

void read_temps(void)
{
    printf("\nExecuting read_temps(). \n");
}

```

该函数打印表头,然后用一个 for 循环取得并打印存入 temps 数组中的数据。最后,还打印出最小值及平均温度。其他函数还没有完成,因此仅仅打印出信息以表示它们正在被执行。

注意到包含未完成的函数调用,可以帮助检查程序流程和保持程序的结构。这种设计程序的方法通常称为自顶向下的方法。首先在程序的顶层(main()函数)开始设计,然后进行由

main()直接调用的函数的设计,如`table_view()`。当`table_view()`在顶层可以正常工作后,便可继续实现由它调用的函数——`min_max()`和`avg_temp()`。

现在来编译运行 PLOTEMP3.C,选取 E,并输入测试数据(10,20,30,40,50,60,70 和 80)。当用户再返回到菜单时,选取 T(Tableview),将得到如下显示:

```
Reading Temperature (F)
1 10 2 20 3 30 4 40 5 50 6 60 7 70 8 80 9 90
Executing min_max().
Minimum temperature:
Maximum temperature:
Executing avg_temp().
Average temperature:
```

3.9 通过设置监视项来监视程序运行

现在,已输入完 8 个数据并返回 0。最后一个值为 0,表明这里可能犯了“差一”错误,即用户循环的次数比期望的少或多一次。

首先可以通过在`table_view()`中 for 循环的第一行设置一个断点来查找出错原因。将光标移到此行并选取 Break/Watch | Toggle Breakpoint(或按 Ctrl+F8)。

然后,再次运行程序,输入测试数据,再选择 T。程序将停在`table_view()`中的断点处,现在,便可以看看 for 循环是如何进行的。

到此为止,已经能通过单步运行程序并使用 Debug | Evaluate 来检查变量的值,这对于每次只检查一个值是很有效的方法,但当处理循环或重复函数的调用时,可能需要不断地观察变量值的变化。如果仍用上面方法将是很繁琐的,这里,调试器可以使用户通过设置 watches 来自动监视变量值的变化。一个 watch 是一个表达式,在运行程序中它的值会被不断修改。

3.9.1 添加一个监视项

如果用户需要监视两个变量:reading(每循环一次加 1)和 temps[reading](在每次循环中被打印的值)。设置这些 watches 最简单的方法是在编辑器中将光标移到所要观察的变量名上,然后选取 Break/Watch | Add Watch(或按 Ctrl+F7)。把光标移到 reading,按 CTRL+F7,将看到一个弹出窗口。和 Debug | Evaluate 一样,被显示的缺省名是光标处的变量,只需按 Enter。用同样的方法为 temps[reading]设置一个 watch。弹出窗口显示 temps 和 Debug | Evaluate 一样,也可以用→将表达式的剩余部分拷贝到该窗口中,然后按 Enter。

3.9.2 观察监视项

现在用户已设置了两个 watch,每个 watch 的变量名和值都显示在 watch 窗口中:

```
reading:177
temps[reading]:92
```

由于还未运行循环,所以显示的值是没有意义的,仅代表各内存位置上的当前数据。现在开始单步运行循环(用 F8),并注意值的变化。第一次循环完成后,这些值为:

```
reading,0
```

```
temps[reading],10
```

下一次循环结束时,watch 显示为:

```
reading,1
```

```
temps[reading],20
```

执行完最后一次循环时,其值为:

```
reading,8
```

```
temps[reading],0
```

这表明循环在 reading 达到 8 时结束。但是由于要输入 8 个 reading,而 reading 从 0 开始,因此循环中 reading 的最后值就应为 7,而不是 8。现在来看看循环的终止条件:

```
reading<=READINGS
```

检查程序开始处的 #define,发现 READINGS 等于 8。所以要使 reading 等于 7 时结束,则循环结束条件应为 reading<READINGS。修改后,再次运行程序验证其结果。

3.9.3 控制调试器窗口

如果想要设置多个 watches,而 watches 窗口中可能没有足够空间来同时显示它们。这时可以使用 Pgup 和 PgDn 键来滚动 Watch 窗口,或用 ↑ 和 ↓ 键来一次移动一行。

如果有一个 watch 表达式的值太长不能全部显示在该窗口中,可以使用 Home、End、← 和 → 键来移动(这在 Debug | Evaluate 窗口中也能使用)。

另一种方法是调整窗口大小,用 TCINS 设置。

在任何时候都可按 Alt-F5 看程序的输出屏幕。按任意键返回到集成环境。

3.9.4 编辑和删除监视项

编辑、添加或删除监视项都很容易。当监视项窗口处于活动状态时,当前活动表达式被加亮显示。要选择另一个表达式,可用 Home、End、↑ 或 ↓ 键。

要编辑(修改)当前被加亮的监视项,可用 Break/Watch | Edit Watch。当窗口显示在屏幕底线时,按 Enter,调试程序为被选表达式弹出一个窗口,用户可以编辑它。可以将 watch 中 temps[reading]改为 temps[reading+1](注:这里只能改变表达式,而不能改变它的值。要改变值,应用 Debug | Evaluate)。

如果需要添加一个 watch,当 watch 窗口处于活动状态时,可以按 Ins 键,将出现一个弹出窗口。此时可以输入 watch 表达式。

要删除当前 watch,Break/Watch | Delete Watch,或移动亮条到要删除的监视项按 Del 键。也可通过选取 Break/Watch | Remove All Watches 一次删除所有的 watch。

3.9.5 寻找一个函数定义

现在 PLOTEMP.C 已经变得较长,要找到欲检查的函数就比较困难。调试器提供了一种方法,可将编辑窗口滚动到一个指定的函数定义处。

选取 Debug | Find Function,则 Turbo C 打开一对话窗口。输入 get_temps (不要输入函数名的括号,否则调试器不能识别)。于是 get_temps()的定义显示在编辑窗口中。这在查看函数定义时是很有用的,同样也可用于定位设置的断点和 watch。

注意, Debug | Find Function 操作只用于在被编译的文件中有源代码的函数。库函数, 如 `printf()` 不能用 Debug | Find Function 查找, 因为其源代码不在集成环境中。

3.9.6 查找调用关系

在一个复杂的程序中, 函数调用可能有许多层, 用户会发现当程序运行到某一断点时, 要记住函数之间的调用顺序是比较困难的。调试器也能帮助用户解决这个问题。例如, 可在 `min_max()` 的 `printf` 语句处设一个断点。

运行程序到断点处, 选取 Debug | Call stack, 便出现一弹出窗口, 列出了所有在该点等待继续执行的函数。最近调用的函数在窗口顶端, 这里为 `min_max()`。它被 `table_view()` 调用, 而 `table_view()` 又被 `main()` 调用。

用户可以用 ↑ 和 ↓ 键指定 Call stack 窗口中的一个函数。如果按 Enter, 则编辑窗口滚动到该函数中被执行的上一条语句(可以通过选取 Call stack 窗口中的第一个函数, 返回到当前执行位置, 这里为 `min_max()`)。现在:

- `min_max` 中被执行的上一条语句是其定义中的第一行, 因为那是用户设置断点的地方。
- `table_view()` 中被执行的上一条语句是包含 `min_max()` 调用的行。
- `main()` 中被执行的上一条语句是执行 `table_view` 的那行, 即:

```
case 'T':table_view(); break;
```

换句话说, `min_max()` 正在执行中, 而 `table_view()` 和 `main()` 等待完成。

3.9.7 多个源文件

实际上许多程序都由几个源文件组成。调试器可以自动地将所要求的源文件装入编辑窗口。例如, 如果用 Debug | Find function 来查找一个不在当前编辑窗口的另一个文件中说明的函数, 则调试器自动地装入相应的源文件。若修改过当前文件, 则将先询问用户是否将修改过的文件存盘。当用 Debug | Call stack 检查一个函数的当前执行位置, 而该函数定义在另一个源文件中时, 情况也一样。

虽然调试器使得对多个源文件的操作变得容易, 但最好一次只调试一两个源文件。通常在继续测试前改完已发现的错误, 因为, 其它错误可能恰恰是由于这个错误存在而引发的。

3.10 预防措施

下面将继续进行 PLOTEMP.C 的开发和测试。为将来调试的需要, 先来看一些减少错误的方法和一些常见的错误类型。

1. 预防性设计

正如用户驾驶汽车时多加预防, 便可以减少事故一样, 用户可以在设计程序时多加小心以避免错误。用户已看到, PLOTEMP.C 的设计就代表了一种自顶向下编程的预防性设计方法。可以先用一些简单的、易于定义的函数建立自己的程序, 这会使测试和结果分析变得容易, 同时也使程序易读和易于修改。例如, 如果 PLOTEMP.C 由在同一函数中的 `table()` 和 `graph view()` 组成, 则代码将变得不易控制。

尽量减少每个函数中参数的个数和要修改的元素个数。这将使进行测试和分析结果以及阅读和修改程序都变得较容易,同时它也可以限制错误程序引起严重的后果,允许用户在一个调试阶段中多运行几次该函数,这种程序设计方法称为疏松连接(Loosely coupled)。

2. 清楚地编写

为将程序编写得清楚,可以用一致的缩进格式书写,并带上一定的注释和变量意义的描述。保持程序的简洁。用多个简单的语句来表示复杂操作,而不要过多地使用复杂的语句。Turbo C 的代码优化可以提高用户程序的运行效率,同时也使其易于调试、阅读和修改。

当编写程序时,不要计较最后一点效益,因为当用户尽力使程序高效时,它也正变得难读和难以调试。如果最后自己的程序运行太慢的话,再来考虑哪一部分值得提高速度,和怎样最好地做到这一点。(Turbo Profiler 是完成这一任务的最佳工具)。

应小心编写可能在用户的程序中以多种方式使用的函数和可能在别的程序中被调用的函数。编写和调试一个通用函数通常比编写两三个专用函数更难。

3.11 有系统的软件测试

飞机起飞前,机组人员都要对其做系统的检查,以保证各部分工作正常。按一定的例行工作可以节省开支。同样,用户也应用标准的方法来做软件测试:一系列经验性的测试步骤可以得到一个可靠的程序。

并不存在一个测试程序的“好”方法:测试依赖于所编的程序、编程时间、水平和个人风格。下面的测试表可以作一个起点。

- 给程序一些简单但不琐碎的输入。试一下不常用的情况,例如,在 PLOTEMP 中输入负温度值,用 Debug | Evaluate 和 watch 来检查数据项的值。改正所发现的错误,一次一个或几个。
- 给程序输入另一组数据,检查上一步中未检查的部分。可能的话,找一个不熟悉程序的人来输入一组数据。经验表明程序员本人有时难以发现问题,因为自己知道哪些数据合适,哪些数据不合适。如果程序是为会计设计的,就可以找一个会计来实验。
- 测试程序中的每一条语句就可能在未预计到的地方发现错误。
- 运行整个程序来修改,可以把程序给其它人使用,来测验程序对每个可能出现的错误的反映。一个能较好地处理多类错误的程序便可以称作是健壮的。

3.11.1 全面测试修改结果

当修改一个程序后,应重新测试所有受影响的部分,尤其应测试那些未被修改但受到影响的部分。

如果程序较复杂,应保持上次运行的记录。当用户修改程序时,该记录可以帮助重复以前的测试,以便检查是否受修改的影响。

3.11.2 仔细观察的部分

在继续学习 C 开发程序时,保留一个常见语法错误和代码出错表,以便调试阶段检查

这些错误。这里列出了一些许多程序易于出错的地方：

- 越界错误。
- 混淆地址和这些地址中的值。
- 增量运算符和减量运算符放错了地方。
- 语句测试不彻底。
- 用 Pascal 语法代替 C。

下面讨论以上各情况。特别注意边界条件,尤其是退出循环的条件、填充数组的条件等等。错误常常出现在边界条件的处理上。在前面的条件 `reading<=READINGS` 引起的错误也是由于边界条件错误。C 语言中从 1 而不是 0 开始,还会引起其它类似的错误。

通常还应小心是否指定了一个地址或该地址内的值。例如不要混淆值 `temps[reading]` 和地址 `&temps[reading]`。

小心增量和减量运算符 `++` 和 `--`。注意是在使用前还是在使用之后增量。

必须用多种方法测试单个语句或表达式,如:

```
switch(strcmp(a,b))...
```

`strcmp` 可能返回三个值:0(a 等于 b),-1(a 小于 b),或+1(a 大于 b)。建议用三组值来测试该语句,以便验证 `strcmp` 在各种情况下都工作正常。

```
x=(x>0)? func(x):0;
```

该语句包含一个“隐含 if”,可产生二种不同的结果。

3.12 完成 PLOTEMP.C

在已经编写和测试了 PLOTEMP.C 原型,并且已经完成、测试和调试了 `get_temps()` 和 `table_view()` 函数后,用户已经学习了怎样使用调试器的所有性能。通过 PLOTEMP.C 的实现,可以练习:

- 用下面给出代码中的相应函数代替原来的程序。本书已为用户提供了各段程序代码,只需将它们编入自己的文件。
- 修改函数原型(需要的话)。
- 用适当的调试工具测试函数的执行。
- 发现并修改错误。
- 继续进行下一个函数。

3.12.1 完成 table_view()

想完成该函数,就必须先实现下面两个函数:

```
void min_max(int num_vals, int vals[], int *min_val, int *max_val)
{
    int reading;
    *min_val = *max_val=vals[0];
    for (reading = 1; reading < num_vals; reading++)
    {
```

```

        if (vals[reading] < *min_val)
            *min_val = (int)&vals[reading];
        else if (vals[reading] > *max_val)
            *max_val = (int)&vals[reading];
    }
}

float avg_temp(int num_vals, int vals[])
{
    int reading, total = 1;
    for (reading = 0; reading < num_vals; reading++)
        total += vals[reading];
    return (float) total/reading; /* reading equals total vals */
}

```

装入 PLOTEMP4.C。在 PLOTEMP 程序中故意加入一些错误,以便用户可以练习调试技巧。

由于这两个函数带有参数并有返回值,所以应修改其函数原型:

```

void min_max(int num_vals, int vals[], int *min_val, int *max_val)
float avg_temp(int num_vals, int vals[])

```

这些修改也包含在 PLOTEMP4.C 中。

最后,修改 table_view(),以便正确使用这些函数的返回值。因此 table_view()应为如下形式:

```

void table_view(void)
{
    int reading, min, max;
    clrscr(); /* clear the screen */
    printf("Reading\t\tTemperature(F)\n");
    for(reading=0; reading < READINGS; reading++)
        printf("%d\t\t%d\n", reading+1, temps[reading]);
    min_max(READINGS, temps, &min, &max);
    printf("Minimum temperature: %d\n", min);
    printf("Maximum temperature: %d\n", max);
    printf("Average temperature: %f\n", avg_temp(READINGS, temps));
}

```

该修改也包含在 PLOTEMP4.C 中。

接下来,由用户自己进行调试。注意以下几方面:

- 循环的工作是否正确?
- 算术运算是否合适?
- 比较操作要比较什么?

3.12.2 实现 graph_view()

graph_view()函数产生图表。欲实现该函数,用下面给出的函数定义代替原来的函数

(注意这时应在程序开始处加上 `#include <graphics.h>`):

```
void graph_view(void)
{
    int graphdriver=DETECT,graphmode;
    int reading, value;
    int maxx, maxy, left, top, right, bottom, width;
    int base; /* zero x-axis for graph */
    int vscale=1.5; /* value to scale vertical bar size */
    int space=10; /* spacing between bars */
    char fprint[20]; /* formatted text for sprintf */
    initgraph(&graphdriver,&graphmode,".\\bgi");
    if (graphresult() < 0) /* make sure initialized OK */
        return;
    maxx=getmaxx(); /* farthest right you can go */
    width =maxx/(READINGS+1); /* scale and allow for spacing */
    maxy=getmaxy() -100; /* leave room for text */
    left =25;
    right=width;
    base =maxy/2; /* allow for neg values below */
    for (reading=0; reading <= READINGS; reading++)
    {
        value =temps[READINGS] * vscale;
        if (value > 0)
        {
            top=base-value; /* toward top of screen */
            bottom=base;
            setfillstyle(HATCH_FILL, 1);
        }
        else
        {
            top=base;
            bottom=base-value; /* toward bottom of screen */
            setfillstyle(WIDE_DOT_FILL, 2);
        }
        bar(left, top, right, bottom);
        left +=(width + space); /* space over for next bar */
        right +=(width + space); /* right edge of next bar */
    }
    outtextxy(0,base,"0.");
    outtextxy(10,maxy+20, "Plot of Temperature Readings");
    for (reading=0; reading< READINGS; reading++)
    {
        sprintf(fprint, "%d", temps[reading]);
```



```

        outtextxy((reading * (width+space))+25, maxy+40, fprint);
    }
    outtextxy(50,maxy+80, "Press any key to continue");
    getch(); /* Wait for a key press */
    closegraph();
}

```

3.12.3 save_temps()和 read_temps()

函数 save_temps()是将当前的“scratchpad”(即数组 temps 的内容)存入磁盘文件。现在用户应对数组元素存取的逻辑关系很熟悉了。

用下面程序代替 save_temps()函数原来的定义:

```

void save_temps(void)
{
    FILE * outfile;
    char file_name[40];
    printf("\nSave to what filename?");
    while (kbhit()); /* "eat" any char already in keyboard buffer */
    gets(file_name);
    if ((outfile = fopen(file_name, "wb")) == NULL) {
        perror("\nOpen failed!");
        return;
    }
    fwrite(temps, sizeof(int), READINGS, outfile);
    fclose(outfile);
}

```

函数 read_temps()与 save_temps()相反,它从磁盘文件中读出数据并放在数组 temps 中。通过用以下程序段替换原来的函数定义实现 read_temps():

```

void read_temps(void)
{
    FILE * infile;
    char file_name[40] = "text";
    printf("\nRead from which file?");
    while (kbhit()); /* "eat" any char already in keyboard buffer */
    gets(file_name);
    if ((infile = fopen(file_name, "br")) == NULL)
    {
        perror("\nOpen failed!");
        return;
    }
    fread(temps, sizeof(int), READINGS, infile);
    fclose(infile);
}

```

完成 `read_temps()` 后, 用户便已得到了一个完整的 `PLOTEMP.C` (`PLOTEMP6.C` 是完全没有错误的程序)。

第四章 多文件工程管理

由于大多数的程序都含有不止一个文件,因而有一个可以自动识别哪些文件需要重编译和连接的工具是十分有用的。Turbo C 的工程管理程序正是完成这一任务的,而且,它还有些更强的功能。

工程管理程序可以使用户指定属于一项工程的所有文件。无论何时重新编译这个工程时,它都将自动地更新保存在工程文件中的信息,工程文件包含以下信息:

- 工程中的所有文件。
- 文件可在磁盘中的哪个路径找到。
- 哪些文件依赖于其它被先编译的哪些文件(自动依赖关系)。
- 生成程序的每一模块时用什么进行编译,以及使用哪些编译选择项。
- 结果代码的存放目录。
- 上次编译的代码长度、数据长度和行数。

工程管理程序的使用是很方便的,用户可以使用以下步骤来建立一个工程文件:

- 用 Turbo C 编辑器编写一个工程文件.PRJ。
- 从 Project | Project Name 打开一个工程文件。
- 使用 Compile | Make EXE File 进行编译。

下面通过一个例子来说明工程管理程序是如何进行工作的。

4.1 工程管理程序的使用

假设一个程序由主源文件 MYMAIN.C 和一个支持文件 MYFUNCS.C (其中包含被 main 文件引用的函数及数据)以及头文件 MYFUNCS.H 组成。

MYMAIN.C 如下所示:

```
#include <stdio.h>
#include "myfuncs.h"
main (int argc, char * argv[])
{
    char * s;
    if (argc > 1)
        s=argv[1];
    else
```

```

    s="the universe";
    printf("%s%s.\n",GetString(), s);
}

```

MYFUNCS.C 如下所示:

```

char ss[]="The restaurant at the end of";
char * GetString(void);
{
    return ss;
}

```

MYFUNCS.H 如下所示:

```
extern char * GetString (void);
```

以上两个文件就可以作为一个工程提供给工程管理程序进行管理。

首先,指定工程文件名:这里取名为 MYPROG. PRJ。注意:该工程文件与主程序文件 MYMAIN. C 不同名。这样,可执行文件为 MYPROG. EXE(如果生成了映像文件的话,映像文件的名字为 MYPROG. MAP)。

这些文件可以同名(除扩展名外),但不一定必须同名。用户的可执行文件名(及映像文件名)是以工程文件名为基础生成的。

用 Turbo C 编辑器建立这两个文件的工程文件。工程文件仅包含将被编译、连接的文件名,本例中为:

```

mymain
myfuncs

```

用 File|New 选项打开 NONAME. C 文件,输入上面两行。扩展名. C 是不必要的,TC 假设无扩展名的文件都是. C 文件,文件的顺序是无所谓的,只是影响编译的顺序,下面的 project 文件跟前一个最终结果相同:

```

myfunc
mymain

```

现用 File|Write to 选项将内容存取 MYPROG. PRG 中,工程文件就创建好了。

用户可以将自己的工程文件存入任一目录;若想存入非当前目录的另外一个目录,只要在文件名之前给出路径名。(如果源程序文件不在当前目录,也必须指定查找路径)。注意,所有文件及对应的路径都是相对于装入工程文件的目录而言,若装入的工程文件在另一个目录中,当前目录将被设成装入工程文件的目录。

用 Project|Project name 选项装入 MYPROG. PRJ 工程文件,在 Project Name 输入框中输入工程文件名之后,Turbo C 就知道了有关 MYPROG. PRJ 工程包含的源文件。

如果 MYMAIN. C 包含 KEY. H 和 TWINDOWS. H,则前面的工程文件应改为:

```

mymain (key. h ,twindows. h)
myfuncs

```

另外,在工程文件是可以显式给出源程序名和目标文件名和库文件名,比如在工程中给出 MYMAIN. LIB 和 MYLIB. LIB 等都是合法的。

Turbo C 自动进行依赖关系检查,如果 mymain. c 版本比 mymain. obj 新时(也就是 mymain. c 编译成 mymain. obj 后又做了某些修改,或 keys. h 或 twindows. h 做某些改变),

则重新再编译成 `mymain.obj`。如果 `mymain.obj` 的版本又比 `mymain.exe` 新时, `mymain.obj` 就再与适当的目标文件(依存储模式而定)与运行函数库再连接成 `mymain.exe`。

所有编译选项和目录设置完成之后, Turbo C 就已经知道用 `MYMAIN.C` 和 `MYFUNCS.C` 来生成 `MYPROG.EXE` 程序的一切信息, 现在可以进行编译来真正建立该工程了。

按 F10 进入主菜单。按 F9(或选取菜单命令 `Compile | Make EXE File`)来编译生成 `MYPROG.EXE`, 然后按 Ctrl-F9(选取菜单命令 `RUN | RUN`)运行用户程序。要查看程序的输出, 可以选取菜单命令 `Windows | User Screen`(或按 Alt-F5), 然后按任意键返回集成环境。

下次运行 Turbo C 时, 如果用户保存了配置文件, Turbo C 可以重新装入该工程文件直接进入该工程。如果当前目录中只有一个 `.PRJ` 文件, Turbo C 将自动装入该 `.PRJ` 文件。否则, 装入默认的工程文件。由于程序文件及其路径是相对于工程文件目录而言的, 所以可以通过移到工程文件的目录, 来激活 Turbo C 而操作任何一个工程。Turbo C 将自动地装入正确的文件。如果当前目录没有工程文件, 则装入默认的工程文件。

4.2 出错跟踪

和单个文件的程序一样, 对多文件的程序来说, 语法错误的出错信息和警告信息能被显示在消息窗口中。

要明白这一点, 可以在 `MYMAIN.C` 和 `MYFUNCS.C` 中故意制造一些语法错误。在 `MYMAIN.C` 中, 把第一行的前面那个尖括号和第十五行的 `dar` 中的字符 `C` 删掉, 这样将产生五个出错信息以及两个警告信息。

在 `MYFUNCS.C` 中, 把第十五行 `return` 中的 `r` 删掉, 这将产生两个出错信息以及一个警告信息。

想要看到在多个文件中跟踪的效果, 还需要修改使 Turbo C 停止 MAKE 过程的判断条件。

4.2.1 终止 MAKE

可以通过在 `Project | Break make on` 选项选取一个设置来设置能使 MAKE 过程中止的消息类型。缺省值是 `Error`, 这是用户通常要用的设置。尽管如此, 用户可以设置成使制作中止的条件是出现警告、错误、严重错误时或者在连接之前。

这些模式事实上是由采用何种方法修改程序错误所决定。如果用户希望发现错误立即进行修改, 可以选取 `Warnings` 选项或者 `Errors` 选项。如果用户希望在修改之前先得到所有源文件错误的一个完整列表, 那么选取 `Fatal Errors` 选项或 `Link` 选项。为了看到多个文件中的错误, 这里选取 `Fatal Error` 选项。

4.2.2 多源文件的语法错误

因为用户已经在 `MYMAIN.C` 和 `MYFUNCS.C` 中造成了若干语法错误, 直接按 F9 (Make)MAKE 该工程。编译窗口显示被编译的文件, 每个文件的错误数目以及错误总数。

出现闪烁信息“Errors, Press any Key”时按任意键。

光标将定位在消息窗口中第一个出错或警告信息位置处。如果该信息所指的文件正在编辑窗口中,那么该编辑窗口中的亮条即指示出错位置。用户可以在消息窗口中前后滚行查看所有的消息。

注意,对每个编译的源程序文件有一个“compiling”消息,这些消息的作用类似于文件边界,分隔各模块以及嵌入文件产生的各种消息。滚行到一个不同的源程序文件产生的消息上时,编辑窗口只跟踪当前装入的文件。

这样,移到当前未装入文件的消息上时,编辑窗口中的亮条关闭。按 Spacebar 装入该文件并继续跟踪,亮条将重新出现。如果选取了某条消息(即定位于该消息上时按 Enter),Turbo C 把该消息所指向的文件装入编辑窗口,并使光标定位在出错位置上。如果此时再返回消息窗口(按 Alt+W+M),将假定跟踪在该文件上进行的。

用 Options | Environment) 中的 Source Tracking 选项帮助确定一个文件装入了哪一窗口,进行消息跟踪或单步执行程序时要用到这些选项设置。

注意,Previous message(Alt+F7)和 Next message(Alt+F8)命令受 Source Tracking 选项设置的影响。这两个命令总是用 Source Tracking 选项设置给定的方法查找前一个或下一个错误和装入该文件。

4.2.3 保存或删除信息

一般地,在 MAKE 一个工程之前,要清理消息窗口为新的信息腾出空间,但有时候需要在工程 MAKE 之间保留消息。

考虑如下一个例子:用户有一个工程包含有多个源程序文件,Break Make On 选项设为 Errors。在此情况下,编译了多个有警告信息的文件之后,某个文件中的错误将使 MAKE 过程中止。改正该错误,然后再编译看看是否能通过。但如果再次进行 MAKE 或编译操作,原来的警告信息将丢失。选取 Preferences 对话框中的 Save | Old Messages 选项能避免这种情况的发生。这样删除的只是用户重新进行编译的文件的消息,这些文件的原有消息为新消息(若用的话)所取代。

任何时候用户都可通过选取 Project | Remove Message 命令来清除所有消息。再次执行 MAKE 操作也将清除所有的旧消息。

4.3 工程管理程序的功能

编译刚才的工程时,涉及到了最基本的情况——C 语言源程序文件名列表。此外,工程管理程序还将为用户提供许多其它方面的功能。

4.3.1 自身依赖性检查

工程管理程序在编译时整理了自身依赖性信息,并保存起来,这样只需要处理那些在集成环境之外编译的文件。工程管理程序能自动把工程列表中的源程序文件(包括它们的包含文件)与对应的目标文件相比较,如果某个特定的 C 语言源程序文件依赖于其它一些文件,这将是十分有用的。对于 C 语言来说,包含几个定义与外部子程序的接口的头文件(.H)是

很普通的,若接口发生了改变,那么需要重新编译用到这些外部子程序的源程序文件。这对编写一个比效大的工程的程序来说,给用户带来了很大的方便。

如果将 Project | Auto dependencis 置为 On,在 MAKE 过程中将读取所有 C 文件及包含文件的时间日期信息,并且与这些文件最近一次编译的时间日期相比较,如有不同,则重新编译该文件。

如果没有打开 Project | Auto dependencis 选项,那么将 C 文件与 OBJ 相比较,如果有较新的 C 文件,则重新编译之。

编译源程序文件时,集成环境编译器和命令行编译器把自身依赖性信息存入 OBJ 文件,工程管理程序正是利用这种自身依赖性信息来验证用来构造 OBJ 文件的每一个文件的时间日期与其 OBJ 文件的时间日期是否吻合。如果不吻合,那么重新编译。

这就是自身依赖性的检查,即具有传统 MAKE 的功能,同时又免除了冗长的依赖性列表。

4.4 取代库

在某些情况下,有必要废弃(取代)标准启动文件或库。把工程文件的第一个文件改为 COX.OBJ 可以废弃标准启动文件,此处的 X 表示任意的 DOS 名。例如 COMINE.OBJ。注意,该文件名必须以 CO 打头,必须为工程中的第一个文件,而且必须显式给出 OBJ 扩展名。

若想废弃标准库,用户只需要把一个特别的库文件名放在工程文件列表中的任意位置。库名必须以 C 打头,也必须显式的给出 LIB 扩展名(例如 CSMYFILE.LIB 和 CNEW.LIB)。

如果废弃了标准库,又想在 MAKE 过程中连接标准数学库,那么用户必须在工程中显式给出其库文件名。

4.5 工程管理程序的其他功能

本节介绍工程管理程序提供的一些其它功能。当一个工程包含许多源程序文件时,用户可能希望能很方便地查看这些文件内容,希望能记录自己的工作情况,也希望可以快速编译包含文件。工程管理程序可以满足这些要求,甚至还可以提供更强的功能。

例如,扩充 MYMAIN.C,加上一个对函数 GetMyTime 的调用。

```
#include <stdio.h>
#include "myfuncs.h"
#include "mytime.h"
main (int argc, char *argv[])
{
    char *s;
    if (argc>1)
        s=argv[1];
    else
```

```

    s="the universe";
    printf("%s %s opens at %d.\n",GetString(),s,GetMyTime(HOUR));
}

```

在 MYMAIN.C 中包含了两个头文件:myfuncs.h 和 mytime.h。这些文件中包含有定义函数 GetString 和 GetMyTime 的原型。

myfuncs.h 中有语句

```
extern char * GetString (void);
```

mytime.h 中有语句

```
#define HOUR 1
```

```
#define MINUTE 2
```

```
#define SECOND 3
```

```
extern int GetMyTime (int);
```

GetMyTime 函数的实际代码放入文件 MYTIME.C 中。

```
#include <time.h>
```

```
#include "mytime.h"
```

```
int GetMyTime (int which)
```

```
{ struct tm * timeptr;
```

```
time_t secsnow;
```

```
time(&secsnow);
```

```
timeptr=localtime(&secsnow);
```

```
switch (which) {
```

```
case HOUR;
```

```
return (timeptr->tm_hour);
```

```
case MINUTE;
```

```
return (timeptr->tm_min);
```

```
case SECOND;
```

```
return (timeptr->tm_sec);
```

```
}
```

```
}
```

MYTIME.C 中包含了标准头文件 time.h,time.h 中有函数 time 和 localtime 的原型以及对 tm 和 time_t 的定义。MYTIME.C 中还包含了 mytime.h 以定义 HOUR、MINUTE 和 SECOND。

创建这些新文件,然后调用菜单命令 Project | Project name 打开文件 MYPROG.PRJ。现在构造扩充后的工程,把 MYTIME.C 加入 MYPROG.PRJ 工程文件中。

现在按 F9 编译。因为自上次编译后 MYMAIN.C 有改动,所以这次重新编译 MYMAIN.C。MYFUNCS.C 没有改动,不需重新编译,MYTIME.C 是头一次编译。

4.6 生成最终应用程序的集成环境设置

在生成最终程序交付给用户使用时,应去掉所有的调试信息、所有的各种系统错误检查代码,为此应如下设置 IDE 的选项:

Options	Linker	Stack Warning	Off
Debug	Source debugging		None
Options	Code generation	Obj debug information	Off
Options	Code generation	Line numbers	Off

第二部分

Turbo C 语言基础

C 语言概貌

操作符和表达式

说 明

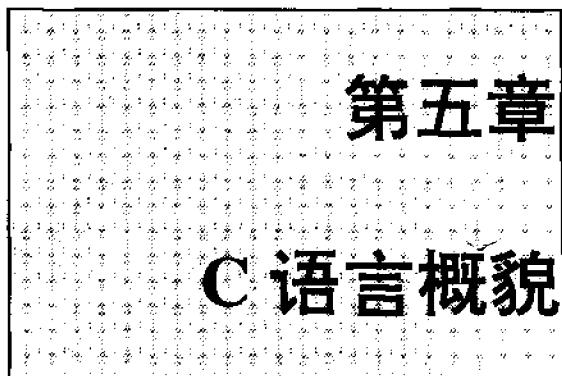
程序控制语句

函 数

指 针

数组、结构、位域、联合和枚举

Turbo C 预处理程序指令



这一章在详尽地讨论 C 语言之前,概括性介绍 C 语言,介绍 C 语言的基础知识,说明如何编译和执行程序,如何控制程序的数据,如何编写 C 语句以及使用如何宏。学习本章时对所有的 C 程序的内容和理解可以是轮廓性的,不需要理解每句程序行的意义。

5.1 预备知识

这一节帮助用户理解两个重要的编程概念,即从源代码到可执行程序的产生方法和程序的执行流程。首先将讨论一个程序如何由源代码编译成可执行代码,然后再介绍如何控制程序流程。

5.1.1 源文件、目标文件和装载模块

当坐在计算机前面,敲入程序时,是在建立源程序文件(源模块)——一种用类似英语的语言编写的,能明白的程序。这时,还不能运行这个程序,因为计算机不能理解源文件。必须把源文件转换为计算机能识别的语言。这一节说明将 C 程序转换为机器能运行的可执行程序的过程。

图 5.1 说明了程序生成的过程和每步建立的文件类型。

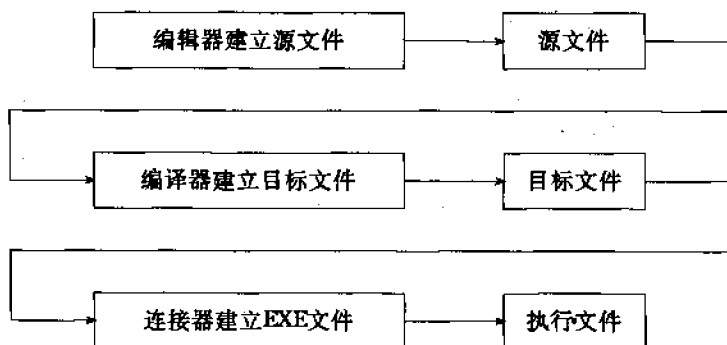


图 5.1 源文件、目标目标和装载模块的建立过程

程序生成过程的第一步是使用文本编辑器编写一个 C 语言源文件(计算机遵循的指令表)。源文件用 C 语言编写。C 是一种英语似的语言,它模仿了程序的工作方式。但是,计算

机不懂用户写的这样类似英文的指令。

为了让计算机能理解用户提供给它的指令,必须将程序转换成计算机可以理解的程序。这就是编译程序的任务,它将 C 语言源文件转换成由机器语言指令组成的目标模块(目标代码)。

将 C 语言程序转换成机器语言程序时,编译程序执行下列活动:

- 读源文件,并且将它转换为一连串预处理符号和空白字符。在这一阶段,特殊字符得以转换,占两行或三行的语句指令合在一起。预处理的符号是编译程序可以处理的简单的元素。例如,操作符、常数、关键字都属于符号。空白字符包括空格字符、制表符和注释,在这个过程中,注释被删除,简单地用一个空格替换。
- “执行”预处理器指令,这些指令包括 `#include` 和 `#define` 信息。`#include` 指令告诉编译程序从另一个文件里读源代码。这些源代码放置在一起,通过前面同样的初始化处理。定义的宏此时也要展开。
- 进一步进行字符和串处理,分析程序中的正确语法和语义,在整个编译期间,任何非法的编码错误都要找出来。
- 如果找不到严重的错误,则生成目标文件。

预处理程序的 `#include` 指令让用户一次编译多个源文件。`#include` 指令引起预处理程序暂停处理当前源文件,而把 `#include` 中规定的源文件合并到整个文件中。

例如,下列指令指示预处理程序暂时停止对程序进行处理:

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

预处理程序读取 `stdio.h` 头文件,把该文件里的 C 代码包括到源代码中来。`#include` 不限制一定是头文件,可以包括用户规定的任何文件。

到这时候,用户已经有了自己可以理解的源文件和只有机器才明白的目标文件(源文件的一种转换形式)。尽管编译程序已经生成了一个机器语言的目标模块,但计算机仍然不能执行整个目标模块。在生成可执行的程序之前还必须再做一些工作。

生成可执行程序的最后一步是用连接程序处理用户的目标模块。连接器把编译程序生成的目标模块与编译程序拥有的特殊目标模块结合起来,结果就是一个可执行的程序,或装配模块(Load module)。与目标模块相结合的特殊目标模块包含另外一些机器语言指令。这些指令让计算机程序能够运行之前,完成一个必需的装配过程。

Turbo C 已自动实现了这个产生可执行程序的过程,用户不必担心这之间的中间步骤。在生成一个源文件之后,只需要执行一个命令来让 Turbo C 编译和连接程序。要编译和连接源文件,可以在当菜单的 Compile 命令下选择 Make EXE File 选择项。

5.1.2 程序的逻辑和执行流程

通常,程序一次顺序地执行一行,并且在大多数情况下是一行接一行地执行,然而,程序不会经常在“真空”里操作。当事件发生变化时,程序需要调整来适应这些变化,那么可以使用条件指令(`if`、`if-else` 和 `switch`)。为了重复执行某项任务,还可以使用循环语句(`do-while`、`while` 和 `for`)。

5.1.2.1 使用条件指令

当处理条件变化的情况时,需要基于同一条件执行不同的代码组。有两个语句可以用来

帮助处理变化的条件:if 和 switch.if 语句可处理一个或两个条件,switch 语句可处理多个条件。

if 语句的基本形式为:

if (表达式 为 真)

执行语句或语句组;

这种形式表示:如果括号里的表达式为真,则执行下列语句,或语句组。下面的例子介绍 if 语句如何工作:

if (a>b)

printf ("A is greater than B");

被求值表达式为 a>b,如果 a 真正大于 b,则执行下面的 printf()语句。如果 a 不比 b 大,则不执行 printf()语句。

现在的问题里,用户使用什么方法来计算表达式的值呢? 有两种基本的方法,第一种为使用关系操作符(见表 5.1),第二种为使用逻辑操作符(见表 5.2)。

表 5.1 关系操作符

操作符	注释	示例
>	大于	2>1
<	小于	1<2
==	等于	1==1
!=	不等于	1!=2
>=	大于或等于	2>=X
<=	小于或等于	1<=X

关系操作符比较两个值。根据关系操作符,如果两个值比较符合所代表的意义,则表达式的值为真,例如表达式 2>1,产生一个真值,因为 2 比 1 大。如果比较结果不符合关系操作符所代表的意义,则表达式为假,例如:1>2 就产生一个假值,因为 1 不比 2 大。

表 5.2 逻辑操作符

操作符	注释
&& (与)	如果两个表达式都为真,则整个表达式为真
(或)	任一个表达式为真,则整个表达式为真
! (非)	将表达式的条件值取反。

当使用逻辑操作符时,用户可以一次对多个条件进行计算,计算每一个表达式,再逻辑比较表达式的结果。下面的例子说明 OR 操作符:

```
if ( (2>1) || (2>3) )
    printf ("The final result was true.");
```

第一个表达式求值为真,因为 2 大于 1;第二个表达式求值为假,因为 2 不比 3 大。然而,整个 if 语句赋值为真,因为两个表达中有一个真,进行 OR 操作后仍然为真。

也可以使用 if 语句的另一种形式:if—else,if—else 语句的语法是这样:

```
if (表达式 为 真) 执行第一条语句或语句组;
else
    执行第二条语句或语句组;
```

象第一种 if 语句一样,if—else 语句给括号里的表达式求值。如果表达式为真,则执行 if 语句后的语句或语句组。然而不象 if 语句,if—else 另外加了一条分枝。如果括号里的表达式求值为假,则执行第二条语句或语句组。

if _else 语句比 if 语句更加灵活。注意下面 if—else 的用法:

```
if (line_count < 66)
    line_count ++;
else
    line_count = 0;
```

在这个代码片段中,检查变量 line_count 的值,如果 line_count 的值小于 66,则 line_count 的值增加 1。然而,当 line_count 的大于或等于 66 时,line_count 就重新设置为 0;在这种情况下,if—else 语句比 if 语句更具有灵活性,因为 if—else 对两个不同的条件提供了不同的反应。

用户可能已经从上面的讨论中注意到,在表达式求值之后,就执行下一条语句或语句组。如果要执行多个语句,则用户必须要意识到一件事:如果有多条语句,则用户要用一对大括号将语句组围起来,原因是当 if 语句给表达式求值之后,if 语句只执行一条语句(用大括号括起的语句组算一条语句)。

当处理多于两个的条件时,if—else 结构不再满足需要。这种情况下,可以有两种选择;第一种选择是编写一套嵌套的 if—else 语句,第二种选择就是使用 switch 语句。

编写一系列嵌套的 if _else 语句在 C 语言中是合法的,编译程序不会将嵌套的 if—else 语句标记为错误或警告。然而编写嵌套的 if—else 语句很困难。在嵌套许多层时,很容易引起混乱。通常不用嵌套的 if—else 语句,可使用 switch 语句,它既易于编写程序,又容易阅读。

下面是 switch 语句的一般形式:

```
switch ( value )
{
    case value: 语句或语句组;
    ...
    default: 语句或语句组;
}
```

switch 语句由几部分组成,语句头部的 value 代表用户要赋值的条件变量的值。字 case 不能少,它是处理特殊条件的代码开始的标志。default 表示:如果没有 case 的 value 与 switch 的 value 匹配的话,则执行 default 的代码。default 选择项是任选的,可有可无。

如果当 switch 语句发现一个 case 与条件相符,那么 switch 语句立即开始执行跟在 case 后面的语句,语句执行一直进行,直到遇到 break 为止。注意程序 swclemo.c 说明了 switch 的用法。

swclemo.c 是使用 switch 语句的程序。

```
1    /* SWDEMO.C A demonstration of the switch statement */
2
3    #include <stdio.h>
4    #include <stdlib.h>
5    #include <conio.h>
6
7    void main( void )
8    {
9        char char_in;
10
11        clrscr();
12
13        printf( "----- Diagnostic Menu ----- \n\n" );
14        printf( "A: System Tests\n" );
15        printf( "B: Video Tests\n" );
16        printf( "C: Hard Drive Tests\n" );
17        printf( "D: Keyboard Tests\n\n" );
18        printf( "Enter a letter to select the tests => " );
19
20        char_in = getche();
21        printf( "\n\n" );
22        char_in = toupper( char_in );
23
24        switch( char_in )
25        {
26            case 'A': printf( "You chose the system tests. \n" );
27                       break;
28            case 'B': printf( "You chose the video tests. \n" );
29                       break;
30            case 'C': printf( "You chose the drive tests. \n" );
31                       break;
32            case 'D': printf( "You chose the keyboard tests. \n" );
33                       break;
34            default : printf( "You did not choose a test. \n" );
35                       }
36    }
```

在 SWDEMO.C 中,该程序是一个简单的菜单程序。使用 switch 语句而不是一系列嵌套的 if-else 语句使得代码很容易理解。

第 13 到 18 行打印主菜单屏幕。第 20~22 行读菜单选择,并将它转换成必要的大写字母。把读入的选字符转换成大写体使得 switch 语句只使用了不转换时使用语句的一半。

第 24~35 行是 switch 语句。第 24 行包含着要检查的变量 char_in。第 26~32 行为菜单选择项选择正确的动作。注意:每个 case 后面跟着 break 语句。break 语句告诉 switch 停止执行指令,并且跳出 switch 语句。第 34 行为 default 选择项。如果变量 char_in 的值不与任何 case 的值匹配,则执行 default 后面的语句。

5.1.2.2 使用循环指令

有些操作需要多次重复进行。循环指令定义了一块要重复执行的代码。根据用户使用的循环种类,代码块可以重复执行设定的次数,或直到某一条件满足为止。Turbo C 提供了三个循环结构:do-while、while 和 for。

do-while 循环是“先执行再检测(postchecked)”的循环结构。只要规定的条件为真,该循环就执行那块代码。do-while 之所以是“先执行再检测”的结构是因为该循环在执行一次代码之后才检测控制条件。do-while 使用的一般形式为:

```
do 语句或语句组
while 表达式为真
```

在循环体处理之前,条件不测试或不能测试的话,do-while 循环很有用。考虑下面的示例:

```
do {
    printf("\n\n Enter a letter =>");
    c=getche();
    c=toupper(c);
    printf ("\n Your uppercase letter is :%c",c);
} while (c != 'X');
```

上面的程序中,提示输入一个字符,该字符将转换成大写体并显示出来。输入字符的值控制着循环。因为输入字符是在循环的中间读入的,所以在循环结束之前,该字符的值不能得到检测。

while 循环是“先检测”(prechecked)循环。象 do-while 一样,只要某条件为真,while 循环就执行循环体。然而,在开始时,while 循环就检测条件表达式,检测工作不在循环尾进行。while 循环采用下列形式:

```
while 表达式为真
    执行语句或语句组
```

当控制表达式的状态最先影响循环功能时,使用 while 循环。fdump.c 展示如何使用 while 循环来读出文件的内容。

fdump.c 是使用 while 循环的程序。

```
1  /* FDUMP.C File dump utility using a while loop */
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main( void )
6  {
```

```

7      FILE *f_in;
8      char c;
9
10     if( ( f_in = fopen( "\\test.txt", "rt" ) ) == NULL. )
11     {
12         printf( "Unable to open file. Aborting.\n" );
13         return 1;
14     }
15
16     while( ! feof( f_in ) ) {
17         c = fgetc( f_in );
18         putchar( c );
19     }
20 }

```

fdump.c 说明 while 循环如何驱动程序读磁盘文件并把文件内容打印在屏幕上。第 7 行说明了一个指向流的 FILE 对象,在第 10 行,这个流指针被赋值,指向与文件 TEST.TXT 相关的流。fopen() 函数试图打开文件,如果 fopen() 打开文件成功的话,它返回一个指向该文件的流的指针;如果 fopen() 不成功,它返回 NULL,如果第 10 行的 fopen() 函数返回 NULL,则输出错误信息,程序就此结束了。

第 16 至 19 行包含了整个读写数据文件的 while 循环,第 16 行为控制循环的表达式,在 16 行中的 feof() 函数检查由 f_in 指向的流的文件结束符,只要没有遇到文件结束符,while 循环就一直执行下去。第 17 和 18 行,循环体从数据文件中读一个字符,并把它打印在屏幕上,最后,第 19 行里的大括号把循环体围起来。

如果想要对循环执行某一个固定的次数,而且执行过程不依赖别的条件的话,可以使用 for 循环,下面是它的结构:

```

for(初始值;条件;变化)
    执行一条语句或一组语句

```

for 循环控制部分的三个值决定执行循环的次数。可以用许多不同的方法给它们赋值,下面讨论控制 for 循环的最常用的方法。

“初始值”规定了开始计数时的值。“条件”表达式限制 for 循环执行一定重叠循环的次数,“变化”表达式表明每执行一次循环,起始值增加或减少多少。注意下面 for 循环的典型示例:

```

for(i = 48; i <= 122; i++)
    printf("%c\n", i);

```

这里 for 循环采用变量 i 从 48 计数到 122。其中 48 为“初始值”,122 是“条件”表达式中规定的值,“变化”表达式使每经过一次循环, i 的值增加 1。如果用户运行这个简单的代码,那么将看到 printf() 语句使用 i 的值来打印输出 1 到 Z 的 ASCII 字符。

下面的例子中,for 循环的控制部分没有参数:

```

for( ; ; )
    printf("Infite Loop");

```

如果没有参数,则没有控制条件。因此该循环就永远地运行下去,或直到关闭计算机。

break 语句让程序执行过程跳出 switch 语句或循环,不再完成 switch 或循环的其它工作。continue 语句使循环重新从头开始执行,不执行 continue 后面的后句。break 和 continue 这两条语句使用户在处理循环过程中出现的不确定事件具有很大的灵活性。

5.2 基本数据类型

C 程序中使用的信息或数据用类型(type)来划分。变量的类型说明该变量包含哪种数据,以及该变量能存储的值的范围。这一节讨论基本数据类型,讲解在何处使用它们,以及如何使用它们。

5.2.1 C 语言的基本数据类型

编写的任何数据程序都对某类数据进行处理,程序所处理的数据可以有不同的大小和类型。每一数据类型需要不同的存储量来存储数据。因此为了提供正确的答案,程序必须知道希望它处理什么类型的数据。

C 程序设计语言有三个基本数据类型:整型数、浮点数和字符数据。每一个基本类型有很多种类,可以提供一个范围宽大的数值。

5.2.1.1 整 数

最基本的数据类型为整型数,是代表一个完整数值的数据类型。完整的数值没有小数部分。

当在程序中想使用一个整数时,需要将变量申明为 int,下面是一些整型变量说明的例子:

```
int loop_counter;  
int cookies, cakes;  
int cars=5;
```

要说明一个整数,必须首先列出保留字 int,然后列出要使用的变量名,在第一行说明中,字 int 通知程序为一个整数设立足够的空间,在 int 后面跟变量名 loop_counter,用来代表那个存储空间。

第二个整型数说明列出两个变量:cookie 和 cakes。在 C 语言中,在一行中说明多个变量是合法的。这样的说明节省空间和输入时间。但如果试图在一行里放上太多的说明,则程序可能变得不可读。

第三个说明建立名为 cars 的变量,并给它赋了初始值。给变量赋初始值是合法的,而且常常需要这样做。

可以使用 int 的三个不同的修饰符来改变整型变量存储值的范围,这些修饰符为 unsigned、long 和 short。表 5.3 列出了各种整数类型、存储空间需要的字节数和每种类型能存储的值的范围。

表 5.3 整型数据类型、大小、范围

类型	大小	值的范围
int	2	32768 到 32767

续表 5.3

类型	大小	值的范围
unsigned int	2	0 到 65535
short int	2	-32768 到 32767
long	4	-2147483648 到 2147483647
unsigned long	4	0 到 4294967295

5.2.1.2 浮点数

第二种数值数据类型是浮点数。不像整数,浮点数可以有小数部分,浮点数也就是所谓的实数。

当想要表达一个精确的数值量时,可以使用浮点数。由于浮点数比较精确,所以它一般用于科学和财政计算。除了比整数精确之外,浮点数还能用整数表示更大更小的值。

说明一个浮点型变量比较简单,用保留字 float 来说明变量名即可。当程序遇到保留字 float 时,程序设置足够的内存单元来容纳这个浮点数,赋给紧跟在 float 之后的变量名,以表示那个数。

下面的代码片段展示了浮点数的几种基本方法:

```
float radius;
float principal interest
float mileage=5.25;
```

第一个说明语句最简单,它说明了一个变量,没有赋初始值。第二个说明语句则说明了如何在一行里说明多个变量。在一行中说明多个变量时要用逗号分开。一行里定义太多会引起程序难以理解。第三个说明语句给变量赋了初始值。

浮点数有不同的大小。较大的浮点类能提供更好的精确度和范围,但它们占用较多的内存单元。浮点类型有:float、double 和 long double。表 5.4 列出了各种浮点类型、大小和范围。

表 5.4 浮点类型、大小和范围

类型	大小(字节)	范围
float	4	5.4×10^{-38} 到 3.4×10^{38}
double	8	1.7×10^{-308} 到 1.7×10^{308}
long double	10	5.4×10^{-4932} 到 3.4×10^{4932}

从表里可以看到,浮点型变量可以表示很大范围的数值。使用大的数值的代价是需要更多存储量来存储数据。例如,每一个 long double 类型的浮点变量需要 10 个字节的存储单元。如果使用时限制在一个小的存储空间里,则使用许多大型的浮点型变量时应慎重。除了提供更大的数值量外,double 和 long double 提供更加精确的值。表 5.5 指明了浮点类型和它们的精度。

表 5.5 浮点类型及其精度

类型	精度
float	精确到 7 位数字
double	精确到 15 位数字
log double	精确到 19 位数字

5.2.1.3 字符型数据

正如您所见, C 语言提供两套好的处理数值数据的数据类型。但数值并不是程序处理的唯一数据类型。大多数 C 语言程序也使用字符型数据。因为在编程时字符数据很重要, 所以 C 语言提供了另一种数据类型来处理字符信息。

字符信息的基本数据类型用保留字 `char` 表示。类型 `char` 的变量可以容纳一个表示单个字符的数值。

鉴于计算机是以二进制代码存储信息的, 因此, 直接存储一个字符是不可能的。为解决这个问题, 计算机使用数值来表示每个字母。类型 `char` 的变量因此实际上存储了一个与特定的字母相关的数值。

PC 机以及所有类似的 PC 系列机, 使用 ASCII 码来表示字符数据。ASCII (American Standards Committee for Information Interchange 美国信息交换标准协会) 开发了一种得到广泛认可的代码来表示 128 个不同的字符值。可以使用 `char` 类型变量来存储任何一个 ASCII 字符的值。`char` 类型的对象使用一个字节的存储单元, 然而该对象只需要字节中 7 位就能表示任何字符。余下的一位留作符号位。`char` 类型的对象因此与 `signed char` 类型的对象等价。

PC 机也使用一个扩展的字符集。该字符集包含额外的 128 个字符, 计算机总共可以使用 256 个字符。如果不使用符号位, 则一个字节可以容纳 256 个不同的值。如果想要使用 ASCII 和扩展的字符集, 则必须使用 `unsigned char` 类型。

总的说来, `char` 类型的对象可以表示 ASCII 字符集。`unsigned char` 类型的对象可以表示 ASCII 字符集和扩展字符集。

应该注意到有些处理字符的子程序通常需要使用 `int` 类型来存储字符的值。例如, `getche()` 函数从键盘中读入一个字符, 并返回该字符的值。返回值的类型不是 `char`, 而是 `int` 类型。`char` 与 `int` 类型的对象是等价的, 并且不需要什么麻烦就可以相互转换。

5.2.2 何处定义数据对象

既然对基本的数据类型情况有所了解, 那么, 现在需要知道在何处使用这些类型。否则数据就没有什么用。这一节说明在何处定义数据对象, 并介绍作用域和生存期的概念。

一个说明数据对象的地方是在源文件的头部, 在 `main()` 函数之前, 在那里说明的对象成为全程(全局)变量, 可在程序里的任何地方使用这些数据对象。GVAR.C 说明了一个全程变量的例子。

gvar.c 是包含全局变量的程序

```
1 /* GVAR.C Using a global variable */  
2
```

```
3    #include <stdio.h>
4    #include <stdlib.h>
5
6    int gvar;
7    void funct_1( void );
8
9    void main( void )
10   {
11       gvar = 2;
12       funct_1();
13       printf( "In main()\n" );
14       printf( "gvar = %d\n\n", gvar );
15   }
16
17   void  funct_1( void )
18   {
19       printf( "In fucent1()\n" );
20       printf( "gvar = %d\n\n", gvar );
21       gvar = gvar * 2;
22   }
```

在 `gvav.c` 中,唯一的变量 `gvar` 是在第 6 行里说明的——在 `main()` 和 `funct_1()` 之上。由于 `gvar` 是在这两个函数的上面和外部说明的,所以这两个函数可以访问它。在第 11 行, `gvar` 赋值为 2。第 12 行调用 `funct_1()`。当 `funct_1()` 里的输出完成时,可以看到 `gvar` 的值仍然为 2,在第 21 行, `gvar` 的值乘以 2,然后控制权传回给 `main()`。当 `main()` 中的输出执行时,可以看到 `gvar` 的值为在 `funct_1()` 中函数赋给它的值,即 4。

`gvar.c` 中的例子说明,全程变量有一个全局的作用域。因此,这个变量可在源文件的任何地方访问和使用。全程变量也有静态生存期,它意味着该变量通常是总可用的。事实上,在目标模块中为全程变量创建的空间是在编译时进行的。

在 C 语言编程中,不经常使用全程变量,原因有二:首先,全程变量使利用存储空间效率降低;即使不需要它们时,它们仍然存在。其次,很容易弄混它们的值。一个通常的错误是改变全程变量的值,而又忘记已经改变过这个值,当下次使用这个全程变量时,它包含一个想不到的值!象这样类似的错误很难跟踪。

大多数程序只使用局部变量,局部变量是在单个的函数里定义和使用的,只能被定义它的函数访问。即使调用另一个使用相同变量名的函数,原来函数中的变量值仍然不改变。程序 `LVAR.C` 给用户示范怎样使用局部变量以及如何怎样才谓误用局部变量。

`LVAR.C` 是包含局部变量的程序。

```
1    /* LVAR.C    Using Local variables. */
2
3    #include <stdio.h>
4    #include <stdlib.h>
5
```

```
6   const PI = 3.1416;
7   void square_1( void );
8   float square_2( float );
9
10  void main( void )
11  {
12      float radius_1 = 5.0;
13      float radius_2 = 7.0;
14      float area_1;
15      float area_2;
16
17      square_1();
18      radius_2 = square_2( radius_2 );
19
20      area_1 = PI * radius_1;
21      area_2 = PI * radius_2;
22
23      printf( "Area 1 = %f\n", area_1 );
24      printf( "Area 2 = %f\n", area_2 );
25  }
26
27  void square_1( void )
28  {
29      float radius_1;
30
31      radius_1 = radius_1 * radius_1;
32  }
33
34  float square_2( float radius_2 )
35  {
36      radius_2 = radius_2 * radius_2;
37      return radius_2;
38  }
```

程序中计算两个圆的面积,说明了使用局部变量的两个方法。第一个圆的面积计算方法有错误,第二个圆的面积计算方法是正确的。

因为第一个圆的半径在第 12 行 main() 函数中已说明,所以该变量只能在 main() 中访问。当调用函数 square_1() 时,尽管 square_1() 对变量 radius_1 进行操作,但 radius_1 的值仍未变。因为 main() 和 square_1() 中的 radius_1 变量的名字相同,但它们之间完全独立。其结果是 main() 中的变量 radius_1 从未乘方过,由此而得 area_1 其值显然不正确。

对变量 radius_2 的处理操作完全正确。注意到说明了两个 radius_2 变量,其中一个是在 main() 中说明的,另一个在 square_2() 中说明,它们是完全不同的变量。因为 radius_2 的值正确地传给了函数 square_2,所以 main() 中 radius_2 的值计算正确。

注意第 18 行中的事件序列。函数 `square_2` 被调用。在 `main()` 中一份 `radius_2` 值传给了 `square_2()`、`Square_2`，然后将传来的值赋给它自己的 `radius_2` 变量。计算完毕，最后返回一个值。当调用 `square2()` 完成时，返回值赋给了 `main()` 的变量 `radius_2`。因为变量的值传递和返回成功，所以 `area_2` 的值也是正确的。

局部变量有局部作用域，意味着局部变量只能在定义它的那个函数中使用。局部作用域也是为什么两个函数可以有相同名字变量的缘故。进一步说，局部变量有自动生存期，这意味着当调用一个函数时，自动为函数要使用的变量分配空间，当函数结束时，变量的空间会自动释放掉，为此从一个函数调用到另一个函数调时，局部变量并不保持值。

如果想要让一个函数中局部变量与另一个函数中局部变量有相同的值时，必须将值传递给被调用函数中的变量，在程序中第 18 行可见到这一点。函数调用 `square_2()` 将 `main()` 的变量 `radius_2` 的值传给函数 `square_2()`。在函数 `square_2()` 中，传给函数的值又赋给了变量 `radius_2`，该变量是 `square_2()` 中的局部变量。

5.3 编写 C 语言表达式和语句

表达式和语句是 C 语言函数和程序的构成单元。这一节说明什么是表达式和语句，怎样使用它们。学习完表达式和语句之后，还要学习如何使用 C 语言的操作符。

5.3.1 表达式和语句

在整本书中都使用表达式 (expression) 和语句 (statement) 这两个术语。在大多数情况下，这些术语似乎相同，但是，情况并非如此，表达式与语句有一些重要的差异。

基本上，表达式是组成语句的部件。表达式可以是一组操作符和计算数值用的操作数，或者是简单的与数据或函数相关的表达式。

表达式可以有三部分：操作符、操作数和标点。操作符是一些符号，它告诉程序进行某类计算操作。操作数是操作符用来执行计算的参数。标点可以帮助规定和组织操作符与操作数之间的关系。

表达式分成简单和复杂表达式。一个简单的表达式可能只由单个的操作数组成；一个复杂的表达式可包含几个操作符和操作数，更复杂的表达式甚至可能由子表达式组成。下面几行代码是表达式的例子：

```
a
15
i+1
j++
x*y+4*Z
(1+j)*(x/(28-Y))
gum(mb)
```

注意到表达式可以使用常数和变量，可以自由地加以混合。

Turbo C 表达式操作数求值的方式可能与操作数书写方式无关。Turbo C 试图重新组织表达式以求产生更好的代码。然而，如果操作符规定了操作数计算值的次序，那么 Turbo C

对那些操作数进行不会重新组织。’

可以使用括号来强迫 Turbo C 按一定的次序对表达式进行求值。看下面这个例子：

```
x = y + ( 2 + 3 )
```

由于有括号，所以首先将表达式 23 的值求出，然后将结果与 y 相加。

表达式与语句的主要区别是表达式产生一个值。语句不会产生值，但它完成一项操作，表达式使用于产生值的地方；语句使用于引起效果的地方，检查下面例子：

```
2+3
```

```
x = 2+3
```

两行代码都是表达式。第一行表达式的值很明显是 5，第二行的代码也是一个表达式，但它有什么值呢？这个值也是 5，它既赋给了变量 X，也赋给了这个表达式。第二行的代码不是语句的原因在于该行不能完成一个操作。要想第二行的代码成为语句，该行应该用分号结束。分号表示动作已经完成，并且表达式引起的效果已经发生。

现在看一看这些逻辑操作：

```
10<15
```

```
result = 10<15
```

第一行代表的值为 1。如果表达式为假，则它的值为 0。因为第二行代码中的逻辑操作为真，那个子表达式产生的值也为 1。这个 1 值既赋给了变量 result，也赋给了表达式。

C 语句以分号结束。当编译程序遇到分号时，它知道这以前的一条语句已经结束，另一条语句就要开始。

如果一组语句用一对大括号包围起来，则这一组语句当作一个单个的语句。当编译程序遇到一个开的大括号时，它扫描源代码，寻求包围一组语句的那个匹配的大括号。当需要执行几个活动，但只允许一条语句时，可使用大括号——例如，在 if 语句后面，当 if 条件为真时，只执行 if 后面的一条语句，那么为了执行一组语句，只需用一对大括号把全组语句括起来就可以了。

Turbo C 使用了 8 种不同类型的语句，表 5.6 概括了这些类型。

表 5.6 语句类型

类型	说明
复合语句	用大括与括起手的一组语句，当作单个的语句处理
表达式语句	用分号结束的语句。在任何其它语句完成之前，表达式两边的结果已求出。
标号语句	goto 和 switch 语句用于转移的地址标识
选择语句	流向控制语句；选择语句为 if else 和 switch 语句
重复语句	循环控制语句；重复语句有 while, do, 和 for 语句。
转移语句	在执行流向中引起非条件转移的语句。跳语句有 break, continue, goto, 和 return 语句
asm 语句	用于嵌入直接插入的汇编代码语句。
说明语句	用于设置数据对象的语句，可能对数据对象进行初始化。

5.3.1.1 介绍 C 语言的操作符集

操作符集合是一组记号,用它们来定义在数据对象进行的基本操作。Turbo C 提供了一套广泛的操作符,能增强编程的速度,减少编程的难度。

这一节讨论所有 Turbo C 的操作符,并介绍操作符优先级——管理操作符求值顺序的一套规则。在 C 语言编程时,理解操作符优先级极端重要。为了得出正确的答案,必须知道操作符是如何操作数据的。

第一组操作符是单目运算符(一元运算符),单目运算符只需要一个操作数。表 5.7 列出了单目运算符和它们的功能。

表 5.7 单目运算符

操作符	含义
+	单目加(一元加)
-	单目减
!	Not 操作符(或者称为“逻辑非”)
&	地址操作符
*	间接操作符
++	增值操作符。如果在操作符前面置一个操作数,则先执行增值操作,如果在操作符后置上一个操作数,则后执行增值操作,即先引用操作数,后增值。
--	减操作符,其它情况与++大同小异。
~	1 的补

第二组操作符为双目运算符。双目运算符正如它的名字一样,需要两个操作数。表 5.8 列出了双目运算符和它们的功能。

表 5.8 双目运算符

运算符	数学运算符
+	加法
-	减法
*	乘法
/	除法
%	取模
赋值运算符	
=	赋值
+=	加赋值
-=	减赋值
*=	乘赋值
/=	商赋值
%=	余数(取模)赋值
&=	位“AND”赋值

续表 5.8

运算符	数学运算符	
=		位“OR”赋值
^=		位“XOR”赋值
<<=		位左移赋值
>>=		位右移赋值
	逻辑运算符	
&&		逻辑 AND
		逻辑 OR
	等式和关系运算符	
==		相等
!=		不等于
<		小于
>		大于
<=		小于或等于
>=		大于或等于
	移位操作符	
<<		左移
>>		右移
&		位 AND
		位 OR
		位 XOR
	成员选择运算符	
->		直接成员选取符
		间接成员选取符
	条件运算符	
a? b:c		if a then b; else c
	逗号运算符	
,		强迫从左到右赋值

象以前提到过的一样,操作符优先级是一套规则,它确定操作符计算的先后次序。如果在语句中只有一个操作符,那么不必担心优先级先后次序,然而,在一个语句里使用两个或多个操作符时,操作符优先级就显得很重要。考虑下面的例子:

```
x = 1 + 2 * 3;
```

注意看到两个操作符+和*。如果没有操作符优先级规则,那么,这个简单的公式可能有两个答案。如果先计算加法操作符,则x等于9。如果首先计算乘法操作符,则x等于7。从这个例子可以看出,了解优先级规则对产生正确、可靠的程序至关重要。

表 5.9 展示了 Turbo C 操作符的优先级。表上部的运算符比下部的运算符优先级要高。换句话说,在这个表的上部的运算符比表下部的运算符先执行。

表中的第一组运算符(操作符)都有一定的联系。每组中的运算符按从左到右或从右到左次序结合。

表 5.9 Turbo C 操作符优先级

操作符	关系
() [] ->	从左到右
! ~ ++ -- +	从右到左
- * & (typecast) sizeof	从右到左
* / %	从左到右
+ -	从左到右
<< >>	从左到右
< <= > >=	从左到右
== !=	从左到右
&	从左到右
^	从左到右
	从左到右
&&	从左到右
	从左到右
?:	从右到左
= += -= *= /= %=	从右到左
&= ^= = <<= >>=	从左到右

5.4 控制类型转换

在本章的前面已知道,程序使用的每一块数据有一个相应的类型。类型确定数据的种类。类型为 int 的数据是整型数,类型为 char 的数据是字符数据。当一种类型的数据换到另一种类型时,会发生类型转换。有时类型转换是自动的,但有时,需要强制进行类型转换。在这一节,要学习类型转换是如何工作的,以及如何控制类型转换。

类型转换(TYPE CONVERSION)简单地把对象的类型进行转换。例如,当整数类型的对象转为浮点类型来处理时,或将 char 类型的对象转换为 int 类型时就会发生类型转换。

尽管可以显式地进行类型转换,但在 Turbo C 中,许多类型转换是由 Turbo C 自动进行的。当 Turbo C 要负责一个类型转换时,称它为隐含类型转换。

这里,理解隐式类型转换

每当算术运算中使用两个不同类型的对句时就会出现隐含类型转换。这样如类型转换使得低级的、精度低的对象转换到更高级的、精度高的类型。自动类型转换导致算术运算的精度更高。自动类型转换的另一个好处是算术表达式赋值的一致性。

Turbo C 有一套管理自动类型转换处理的规则。当需要类型转换时,在第一级进行的是基本的整型转换,其它转换在第二级上处理。

表 5.10 总结了整数类型转换过程。

char 类型对象是由单个字节组成,然而,当 char 类型的对象转换为 int 类型的对象时,要使用两个字节,当转换进行时,新的 int 对象的低字节值与原来的 char 类型对象的值相同,高字节的值则根据 char 类型对象是不是带符号(signed 或 unsigned)而定。若是 unsigned char,则 int 类型对象的高字节为 0,若是 signed char,则 int 类型的对象的高字节为 -1。如果原来的对象是 char 类型的,则转换处理根据 Turbo C 的设置而定。Turbo C 可以隐含地定义一个 char 类型对象为 signed 或 unsigned。

表 5.10 整数类型的隐含转换

原来的类型	转换后类型	转换方法
char	int	高字节设置为 0,或者符号,这由缺省的 char 类型决定。
unsigned char	int	高字节为 0。
signed char	int	扩展的符号
short	int	值相等
unsigned short	unsigned int	值相等
enum	int	值相等

在计算算术表达式的值之前,Turbo C 在类型转换处理时执行几个步骤。第 1 步是将 char short,或 enum 类型转换为整型;这类转换在表 5.10 中已经总结了。类型转换过程中的其它步骤解释如下:

1. 检查表达式,确定是否存在 long double 类型的操作数。如果有一个操作数为 long double 类型,则表达式中其它操作数转换为 long double 类型。如果没有一个 longdouble 类型的操作数,则进行处理的下一步。
2. 检查表达式,看是否有 double 类型的操作数。如果有一个操作数是 double 类型的,则将其它操作数转换为 double 类型的操作数。如果都不是 double 类型,则进行转换处理的下一步。
3. 检查表达式,看是否有 float 类型。如果有一个为 float 类型,则将其它操作数转换为 float 类型。如果没有一个操作数的类型为 float,则进行转换处理的下一步。
4. 检查表达式,看是否有 unsigned long 类型的操作数。如果有一个操作数的类型为 unsigned long,那么将其它操作数类型转换为 unsigned long 类型。如果没有 unsignedlong 类型的操作数,则进行转换处理的下一步。
5. 检查表达式,看是否有 long 类型的操作数,如果有一个操作数的类型为 long 的话,则将其它操作数转换为 long 类型。如果没有 long 类型的操作数,则进行转换过程的下一步。
6. 检查表达式,看是否有 unsigned int 类型的操作数。如果有一个操作数的类型为 unsigned int,则其它操作数转换为 unsigned int 类型的操作数。如果没有 unsigned int 类型的操作数,那么两个操作数必须是 int 类型的操作数。

算术表达式并非是发生隐含类型转换的唯一之处。在赋值和函数调用过程中也发生自动类型转换。当进行赋值时,等号右边的值的类型转换为左边的类型。在函数调用时,传给

函数调用的参数是表达式,则会发生象任何表达式发生的隐含类型转换一样的情况。

5.5 显式类型转换的使用

自动类型转换程序处理许多用户将遇到的类型转换工作,然而,有时,也要亲自控制类型转换。

如果使用命令进行类型转换,则进行的转换是一个显式类型转换。显式类型转换称为类型强制转换(type casting),一个显式强制类型转换的格式如下:

(类型名)表达式

任何一个有效的 C 语言表达式都能进行强制类型转换。在进行类型转换时,只要简单地把强制转换单目运算符置于表达式之前,强制运算符由圆括号里的类型名组成,其后的表达式的结果转换成指定的类型。

下面的一段程序给出了浮点数类型至整型数的转换过程:

```
int i;
float x = 5.9876;
i = (int) x;
printf("i = %d, x = %f", i, x);
```

这一段程序包括有两个变量:一个整型变量和一个浮点型变量。浮点型变量 x 在第二行被分配了一个值。在第三行,把 x 强制转换为整数后赋给 i ;在 `printf()` 语句表示通过类型转换把浮点数的小数部分去掉。

下面的一段程序给出了类型转换如何为函数提供一个正确参数:

```
int a = 2, b = 4;
double x;
x = pow((double) a, (double) b);
printf("x = %f", x);
```

在这个例子中,函数需要两个 `double` 类型的参数,但却用两个 `int` 参数来进行调用。在第三行,两个强制类型转换运算符把整型参数转变成浮点型参数。带有两个 `double` 参数的 `pow()` 函数的工作很正确。当执行完后, `pow()` 值返回 16.0。应该知道变量值在类型强制转换中会改变。当浮点数转换成整型数时,变量值的改变是明显的,因为整型数不能带浮点数的小数部分,因此当一个浮点数转换成整型数,它的小数部分就被去掉了。但是一个整型数转换成浮点数时,变量值的改变就不那么明显了。虽然浮点数的取值范围很宽,但是它也不能包括所有数的精确值。因此,当一个整型数转换成浮点数时,转换后的新值可能不等于转换前的值。这个新值可能很接近原值,但是这两个值不一定相等。

通过这些简单的例子就能了解到,自动类型转换在求值和处理数据时提供了许多的灵活性。类型转换最有用的地方是强制转换函数的参数。

5.6 使用 C 的宏

象在字处理器中一样,C 程序中的宏可以节省时间,提高效率。宏也可以是多种多样的,例如,可以使用宏来代表经常使用的变量,或者是象函数那样操作。这部分将介绍如何在程

序中有效地使用宏。

在 Turbo C 中,可以建立两种类型的宏:类似对象的宏和类似函数的宏。当预处理器在程序中遇到类似对象的宏时,宏就会被在程序开始部分定义的值所取代。所取代的值可以是任意的 c 数据类型、字符类型、整数类型,或者是浮点数类型。在类似对象的宏中,宏是被数据对象取代的。而当预处理器遇到一个类似函数的宏时,宏名会被执行某一动作的一段代码所取代。类似函数的宏还可以使用参数。正象类似函数的宏这个名字一样,它所作的工作与函数类似。

5.6.1 定义类似对象的宏

宏是字符串替代工具。当预处理器遇到宏名的时候,宏名就会被先前所定义的另一字符串代替。可以使用下列形式来定义类似对象的宏:

```
#define identifier replacement-list
```

#define 告诉预处理器,下面开始宏定义。identifier 是宏的名字,它可以在程序中出现。replacement-list 是预处理器在程序中遇到宏名时,所替代的东西。

注意下面类似对象的宏定义

```
#define SALES_TAX 0.06
```

```
....
```

```
total=sales * SALES_TAX;
```

#define 标志着宏定义的开始。SALES_TAX 是宏名,而 0.06 是替代串。例中的最后一行显示了宏是如何使用的。当用户想计算税率时,只要写出宏名 SALES_TAX 即可。当预处理器对程序进行扫描时,每一个出现的 SALE_TAX 替换成 0.06。

在创建宏时,要遵循以下简单规则:

- #define 标志着宏定义的开始。
- 宏紧跟着 define 指令后面。当命名宏的时候,可以使用与其它语言变量命名一样的规则。宏名中不能含有空格。虽然不要求都用大写字母,但一般来说,宏要用大写字母。
- 跟在宏名之后的是替代字符串。
- 宏定义不是由分号来终止的。如果使用分号来终止宏定义,预处理器会把分号作为替代串的一部分来看待。
- 反斜线字符(\)可以把宏定义扩展成多行。

要注意 C 的编译器并不负责处理宏。而是 C 的预处理器完成所有必要的工作,把宏名翻译成编译器可以接收并使用的形式。预处理器把程序看作数据文件,在扫描程序中的文本时,预处理器会用预定义的 replacement-list 来代替的每一次出现的宏(identifier)名。

使用宏来定义常量值可以增强程序的性能。考虑上述例子的另一种形式是:

```
double sales_tax=0.06;
```

```
...
```

```
total=sales * sales_tax;
```

在这里 sales_tax 并不是一个宏,而是值为 0.06 的 double 型变量。由于 sale_tax 现在是一个变量,所以程序在第一次遇见它的时候,都要查找 sales_tax 的值。而在以前,SALE

TAX 是宏的时候,预处理器会用值 0.06 来替代它。当程序需要税总量时,其值就已经在那里了。所以使用宏可以节省时间,它不要求每次遇见它时都去查找它的值。

宏所节省的时间在线性代码中并不可观。但当在循环中进行计算时,所节省的时间就相当多了。

使用宏还可用其它方式来节省时间。首先,宏可以使常量值的修改变得十分容易,这样可以节省编码时间。如果使用宏来定义常量,只需要改变一下宏定义,就可以改变程序中所有的常量值。第二,宏可以加快调试程序的过程。当使用宏的时候,只输入了一个值,而不是输入许多次值。由于键入次数少了,就可以减少犯错误的可能性。

宏也可以使程序更加易读。可以使用描述性的名字,它对读者更加合理,而不是简单地把数值分散到整个程序之中。

这部分的第一个例子告诉用户如何来使用宏代表一个数值。在一些情形中,需要使用宏来代表字符或者字符串,而不是仅代表数值。

Objmac.c 定义了使用字符和字符串的宏。

objmac.c 中使用字符和字符串的程序

```
1  /* OBJMAC.C This program demonstrates the use of object--
2     like macros that work with characters and strings. */
3
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  #define FIRST_CHAR 'H'
8  #define SECOND_CHAR 'i'
9  #define COMMA ','
10 #define STRING " this is a macro example."
11
12 void main( void )
13 {
14     putchar( FIRST_CHAR );
15     putchar( SECOND_CHAR );
16     putchar( COMMA );
17     puts( STRING );
18 }
```

OBJMAC.C 显示出类似对象的宏并不只限于与数值一起工作。第 7 行到第 9 行是一组字符的宏定义。注意每个替代串都封闭在引号之中。当宏替代发生时,整个替代串代替了宏名。这样,替代串中的字符和引号会全部代替宏名。

例如,下面语句中的宏

```
putchar(FIRST_CHAR);
```

会被替代为

```
putchar('H');
```

可以看到,每次出现宏名均被字符常量所代替。本例中替代能正常进行,这是因为宏是使用了要求字符常量参数的函数调用之中。在宏替代中使用单个字符时,一定要保证在必

要时包括引号。

第 10 行是扩展为字符串常量的宏定义。注意在替代表中,字符串是包含在引号之中。因此,宏将会被字符串来代替。再一次强调一下,当用户在宏替代表中使用字符串的时候,一定要保证在必要的时候把引号包括进去。

第 14 行到第 17 行是一系列的语句,它打印出所有的宏定义。

当为宏定义字符串的时候,对引号的使用一定要小心。要记住在宏扩展时,拷贝的是替代表中所有的东西。如果在替代字符串包括了引号,在扩展时它们会被拷贝。也要记住在程序引号中的字符串是作为纯字符串来看待的。注意下面这段代码:

```
#define STRING "This is my string"
....
printf("%S", "STRING");
```

这个例子的执行结果可以与原意相悖,打印出的不是宏定义中的替代字符串,printf() 打印的是词 STRING。其原因在于 printf() 语句中的“STRING”是作为纯字符串来看待的,因此,预处理器不会把它作为宏来看待。当用户把宏作为函数的参数使用时,一定要注意引号的用法及它们如何解释。

扩展宏之后,预处理器会再扫描一次来查看其它宏。由于宏会被重新扫描,所以宏可由其它宏来组成。下面的例子说明了预处理器重新扫描的功能

```
# define NUM_1 10
# define NUM_2 15
# define SUM NUM1+NUM2
....
printf("SUM is equal to %d",SUM);
```

在这里定义了三个宏。前两个宏很简单,而第三个宏的定义引用了前两个宏。当扫描程序时,预处理器会把 printf() 语句转换为:

```
printf("SUM is equal to %d",NUM1+NUM2);
```

然后,预处理器会重新扫描扩展的宏,来查看一些新的宏。在这种情形中,预处理器找到两个新宏。在进行第二次宏扩展之后,printf() 语句如下所示:

```
printf("SUM is equal to %d", 10+15);
```

当宏完成使命,想把宏名挪为它用时,可以解除(undefine)宏。为了删除宏定义,可以使用 #undef,它要示如下形式:

```
# undef identifier
```

注意下列使用 #undef 的例子:

```
#define MY_MACRO 1
....
#undef MY_MACRO
```

在这里使用 #undef 清除了宏定义 MY_MACRO。在遇到 #undef 之后,预处理器便不再把 MY_MACRO 作为有效的宏来对待了。

5.6.2 定义类似函数的宏

宏除了提供常量数值功能以外,还有一些更加复杂的用法。用户可以编写一个宏,使它

看起来和工作起来都象函数一样。类似函数的宏定义比起类似对象的宏定义来说,包含更多项。类似函数的宏定义也可包含了形式参数表,参数表可以提供值,来对替代的文本进行改变或者控制。

可以使用下面的一般形式来定义类似函数的宏:

```
#define identifier(identifier-list) replacement-list
```

#define 是告诉预处理器该行其余的信息是宏定义。identifier 是宏名,当预处理器遇到 identifier 的时候,就要进行宏扩展,即用 replacement-list (替代表)来代替 identifier。identifier-list 是形式参数表。当在程序中使用类似函数的宏的时候,应该在宏名之后的 identifier-list 中写上参数。当宏扩展之后,所提供的参数便拷贝到 replacement-list 之中了。

下面的代码部分显示出只要求一个参数的类似函数的宏:

```
#define MTOK(m) m * (8.0/5.0)

...

double miles,kilometers;
miles = 62.0;
kilometers = MTOK(miles);
```

在这里把英里换算为公里。这个宏定义同类似对象的宏定义有所不同,这是由于在宏名之后跟有形式参数表。参数表必须以左半括号“(”开始,并且要紧跟在宏名之后,其间没有空格。当宏展开时,替代的参数就会被 identifier-list 中的参数值来替代。预处理器会把上面代码部分最后一行扩展为:

```
kilometers = miles * (8.0 / 5.0);
```

宏 MTOK()是用参数 miles 来调用的。正如用户所看到的那样,预处理器扩展了宏,并用变量 miles 来替代形参 m。

在类似函数的宏的定义中,还可以使用多个参数。为了调用具有多个参数的宏,所要做的应是要用逗号将参数分开,看下列这个例子:

```
#define WATTS(v,a) v * a

...

int voltage = 120;
int amperage = 5;
int wattage;
wattage = WATTS(voltage,amperage);
printf("Power consumption = %d",wattage);
```

注意形参表中的两个参数均使用在替代表中。参数表中参数数目总是同替代表中的参数数目相等。

类似函数的宏的一个非常有用的特征是,它们可以使用任意的数据类型。现在看一下下面的例子:

```
#define CUBE(x) x * x * x

...

int i;
```

```
float y;
i = CUBE(2);
y = CUBE(1.5);
```

最后两行会被扩展为:

```
i = 2 * 2 * 2;
y = 1.5 * 1.5 * 1.5;
```

在这个例子中,一个类似函数的宏计算了整数的立方和浮点数的立方。如果没有宏的话,就需要用两个函数来完成这样的功能。函数不太灵活的原因在于它只接收某一特定数据类型的参数。但是,宏并不考虑传给它的数据类型。正如从上面例子中所看到的,宏的参数只是简单地拷贝到了扩展的替代表达式之中。每次使用宏的时候,宏便扩展为新的表达式。由于创建了新的表达式,在每次使用宏的时候,就可以为宏提供不同类型的参数。

当使用类似函数的宏的时候,可以以三种不同的方式在替代表中引用形式参数(简称形参)。

引用形参的第一种方式是它自身的出现。到目前为止,所进行的形参引用都属于这种方式。第二种方式是在形参之前加上字符串操作符(#)。第三种方式是在形参前面加上字符串连接操作符(##)。下面部分将检查使用形参的几种不同方式。

在前面的例子中,替代表中的参数是以本来面目出现的。这就是为什么没有在宏扩展时给出特殊字符的原因。替代表中的形参是由用户所提供的实参来代替的。复习一下类似函数的宏的例子。

```
#define SUM(a,b) a+b
```

宏 SUM() 要求两个参数。在宏定义中的参数 a 和 b,就是形式参数。再看一下使用宏 SUM() 的表达式:

```
SUM(1,2)
```

这个表达式使用了 SUM() 宏,并向宏提供了实参。在例子中的实参是整数 1 和 2。当进行宏扩展时,实参就被拷贝到替代表中。因此,宏 SUM() 将扩展为:

```
1 + 2
```

这个表达式是放在程序中的真正实物。预处理器移去了宏名,并用替代表取代了它。正如所看到的,在插入替代表的时候,预处理器使用的是实参。

在替代表中引用形参的第二种方式是在形参之前加上字符串操作符(#)。这个操作符可以使实参出现在扩展宏的引号之中。

下面的代码显示了在 printf() 语句中使用字符串操作符(#)的方式:

```
#define QUOTE(S) #S
```

```
....
```

```
printf("%S\n", "He said"QUOTE("Hi, my name is Joe. "));
```

在这个例子中,QUOTE() 宏是用宏参数来替代的,参数被封闭在引号之中。这个 printf() 语句将打印为:

```
He said "Hi, My name is Joe."
```

当扫描程序的时候,预处理器将宏 QUOTE() 转换为 printf() 语句能使用的正常字符串。乍一看,可能认为预处理器将把宏 QUOTE() 扩展为这样的形式:

```
" " HI, my name is Joe. "
```

但是,一行上两个引号会引起编译错误。字符串文字预处理器的命令可以避免产生这样的错误。预处理器将产生下面的形式,而不是上面的字符串:

```
"\"Hi, my name is Jod. \""
```

其中的反斜线是告诉编译器用户真正想印出的是引号,并不是引出另一个字符串。由于使用了反斜线表明了真正打印的是引号,所以在编译时就不会出现错误。

也许注意到了,在 `printf()` 语句的参数表中包含有两上数据字符串。当 `printf()` 执行时,它打印的是两个数据字符串。打印出两个字符串的原因是数据字符串并没有被逗号分开。因此,编译器会把这两个字符串作为一个来看待。在需要的时候,这种产生字符串的方式是完全可以接受的。

引用形参的第三种方式是在参数中使用字符串连接操作符(`##`)。这个操作符将宏替代表中的两个字符串合并为一个新的、更长的字符串。不象字符串操作符(`#`)那样,字符串连接操作符并不在扩展的宏体上加引号。

下面的代码显示了如何使用字符串连接操作符(`##`):

```
#define CAT(a,b) a##b
....
printf(" %S\n", CAT("My_", "macro"));
printf()语句将打印出:
my_macro
```

`CAT()` 宏定义同用户在前面看到的宏定义类似——除了 `##`、字符串连接操作符之外。这个操作符可以使 `CAT()` 的实参连接在一起。在这个例子中,参数“`my_`”和“`macro`”就连接成了一个字符串,即 `my_macro`。

在预处理器扩展了带有字符串连接操作符的宏之后,预处理器会进行再次扫描,来查看还有没有新的宏。看一下下面的例子

```
#define STR_1 "This is a test!"
#define CAT(a,b) a##b
....
printf(" %S\n",CAT(STR_1);
```

在这里,扫描到宏 `CAT()`,并用 `STR_1` 来代替它。在宏 `CAT()` 扩展以后,预处理器会重新扫描这一行,来查看新的宏。当遇到宏 `STR_1` 的时候,它也加以扩展。那么,`printf()` 语句就变成:

```
printf(" %\n", "This is a test!");
```

然后,预处理器会再进行扫描。由于不再找到更多的宏,所以这个处理过程便结束了。

下面的例子显示的是不会产生预想结果的字符串连接操作符:

```
#define STR_1 "This is"
#define STR_2 "a test"
#define CAT(a,b) a##b
....
printf(" %S\n", CAT(STR_1,STR_2);
```

当进行编译时,这段代码会产生错误。这段代码的本来意图是扩展的宏 `STR_1` 和 `STR`

_2 组合在一起产生了下面的信息，

```
"This is a test"
```

但上面的代码段并不能产生这样的信息。预处理器把 CAT() 宏替代为：

```
STR_1STR_2
```

预处理器正确地扩展了宏 CAT()，但 CAT() 宏的扩展却导致了无意义的字符串。当编译开始后，STR_1STR_2 便会被标为错误，编译器会终止以便修改错误。

第六章

操作符和表达式

本章主要介绍 C 语言的操作符和表达式。C 的操作符非常丰富,不仅具有通常的算术和逻辑操作符,而且还包括许多按位操作、结构与联合成员存取和指针操作(引用与间接引用)等操作符。

6.1 什么是操作符

操作符是一个表示编译程序执行特定的数学或逻辑操作的符号,操作符词法符号在应用于表达式中的变量或其它对象时,将触发某一运算。

本章仅讨论 Turbo C 的标准操作符,重载操作符将在以后的章节中讲述。

以下是 Turbo C 的标准操作符:

```
[ ] ( ) . -> ++ --
& * + - ~ ! / sizeof
% << >> < > <= >= ==
! = ^ & | && || ?: =
*= \= %= += -= <<= >>= &.=
^= &.= |= , # ##
```

除了操作符[],()和?:外,其它多字符操作符将认为是一个单一符号。同一操作符符号可能有多种解释,这要依赖于上下文,例如:

A * B	乘法
* ptr	间接引用
A & B	按位与
A	取地址运算
label;	语句标号
a? x:y	条件语句
void func(int n);	函数说明
a = (b+c) * d	括住的表达式
a, b, c;	逗号表达式
func(a, b, c)	函数调用
a = ~b;	按位补

6.2 单目操作符

&	取地址操作符
*	间接引用操作符
+	单目加
-	单目减
~	按位补
!	逻辑非
++	前增量、后增量
--	前减量、后减量

6.3 双目操作符

加法类操作符

+	双目加
-	双目减

乘法类操作符

*	乘法
/	除法
%	求余

移位操作符

<<	左移
>>	右移

按位操作符

&	按位与
^	按位异或
	按位同或

逻辑操作符

&&	逻辑与
	逻辑或

赋值操作符

=	赋值
+=	和赋值
-=	差赋值
*=	积赋值
/=	商赋值
%=	模赋值
<<=	左移赋值

>>=	右移赋值
&=	按位与赋值
=	按位或赋值
^=	按位异或赋值
关系操作符	
<	小于
>	大于
<=	小于等于
>=	大于等于
相等类操作符	
==	等于
!=	不等于
分量选择操作符	
.	直接分量选择符
->	间接分量选择符

6.4 三目操作符

`a? x:y` 如果 `a` 为真,则取 `x` 的值,否则取 `y` 的值

6.5 标点符号

Turbo C 的标点符号(也称为间隔符)定义如下:

[] () { } , ; : ... * = #

中括号

[](左、右中括号)表示一维和多维数组下标:

```
char ch, str[] = "Stan";
int mat[3][4];           /* 3×4 矩阵 */
ch = str[3];             /* 第 4 个元素 */
...
```

括号

左、右括号()用来组合表达式、隔开条件表达式以及表示函数调用及函数参量:

```
d = c * (a + b);         /* 改变正常的优先级 */
if(d == z) ++x;          /* 条件语句的要素 */
func();                  /* 函数调用,无参数 */
int(*fptr)();            /* 函数指针说明 */
fptr = func;             /* 无()意指 func 指针 */
void func2(int n);        /* 带参函数说明 */
```

在宏定义中建议使用括号来避免在宏扩展时存在的潜在优先级问题:

```
#define CUBE(x) ((x) * (x) * (x))
```

括号可改变正常操作符的优先级和结合律。

大括号

左、右大括号{ }表示复合语句的开始与结束：

```
if(d == z)
{
    ++x;
    func();
}
```

右大括号是作为复合语句的终止符,故除非在结构和类说明中,}后是不需要分号的。有分号通常是非法的,如:

```
if(statement)
{ ... };          /* 非法分号 */
else
```

逗号

逗号(,)用来分隔函数参数表的元素:

```
void func(int n, float f, char ch);
```

也用作逗号表达式中的操作符。逗号的这两种用法在程序中可以混用,但必须采用括号来区别它们:

```
func(i,j);          /* 带两个参数的 func 调用 */
func((exp1, exp2), (exp3, exp4, exp5)); /* 也是用两个参数调用 func */
```

分号

分号(;)是语句终止符,任何合法的 C 表达式(包括空表达式)只要后跟分号就解释为语句,也称为表达式语句。表达计算后,其值被丢弃。若表达式语句无副作用,Turbo C 可忽略它。

```
a + b;          /* 可能计算 a + b,但丢弃其值 */
++a;           /* 对 a 有副作用,但丢弃++a 的值 */
;              /* 空表达式 = 空语句 */
```

分号常用来建立空语句:

```
for(i = 0; i < n; i++)
{
    ;
}
```

冒号

可用冒号(:)来表示一个标号语句:

```
start : x = 0;
...
goto start;
...
switch start
{
```

```
case 1 : puts("One");  
        break;  
case 2 : puts("Two");  
        break;  
...  
default: puts("None of the above!");  
        break;  
}
```

省略号

省略号(...)为三个中间没有空白的点。省略号用在函数原型的形参表中,来表示参数的数目可变,或者参数类型可变:

```
void func(int n, char ch, ...);
```

该说明表示 func 定义成其调用必须至少带有两个参数,其类型为 int 和 char,但还能带有任何数目的附加参数。

星号(指针说明)

在变量说明中的*(星号)表示要创建一指向某类型的指针:

```
char *char_ptr; /* 说明一指向 char 的指针 */
```

多层间接引用指针可说明成由相当数目的星号来表示:

```
int **int_ptr; /* 一个指向整数指针的指针 */
```

```
double ***double_ptr; /* 一个指向 double 指针的指针的指针 */
```

星号也可用作间接引用指针的操作符,或作为乘法操作符:

```
i = *int_ptr;
```

```
a = b * 3.14;
```

等号(赋值)

等号(=)可把变量说明与初始值表分开:

```
char array[5] = {1,2,3,4,5};
```

```
int x = 5;
```

在 C 函数里,变量说明前不能再有任何代码。

等号还用在表达式中作赋值操作符:

```
a = b + c;
```

```
ptr = farmalloc(sizeof(float) * 100);
```

#号(预处理程序指令)

#号若是某行的第一个非空白字符,则表示该行是一预处理程序指令。它指示编译程序作出有关动作,而与代码生成无关。

#和##号(双#号)也是在预处理程序扫描阶段执行词法符号替代和合并的运算符。

6.6 操作符语义

6.6.1 后缀和前缀操作符

有 6 个后缀操作符[],(),.,->,++和--,这些后缀操作符可用来建立如表 5.9

所示的后缀表达式。增量和减量操作符(++和--)同时也是前缀和单目操作符。

数组下标操作符[]

对表达式

后缀表达式 [表达式]

在 C 中,表达式 `exp1[exp2]` 定义为

$*((exp1) + (exp2))$

这里或者 `exp1` 为指针,`exp2` 为整型;或者 `exp1` 为整型,`exp2` 为指针。

函数调用操作符()

表达式:

后缀表达式 (<参数表达式表>)

为对由后缀表达式给出的函数的调用。参数表达式表为由逗号隔开的任何类型的表达式表,它们代表实际的函数参数。函数调用表达式的值由函数定义里的返回语句决定。

结构/联合成员操作符.

在表达式

后缀表达式 . 标识符

中,后缀表达式必须为结构或联合类型,标识符必须为该结构或联合类型的成员名。表达式指明结构或联合对象的成员,表达式的值为所选择的成员值,且仅当后缀表达式为左值时它才为左值。

结构/联合指针操作符->

在表达式

后缀表达式->标识符

中,后缀表达式必须为结构指针或联合指针类型。标识符必须为该结构或联合的成员。本表达式指明结构或联合对象的成员,其值为选取成员的值,当且仅当后缀表达式为左值时才为左值。

后增量操作符++

在表达式

后缀表达式++

中,后缀表达式为操作数,它必须为标量类型(算术或指针类型),且为可修改的左值。整个表达式的值为增量之前的后缀表达式的值。在后缀表达式求值后,操作数递增 1。

增量值与操作数类型相对应,指针类型要应用指针算术规则。

后减量操作符--

除了在求值后操作数为递减 1 之处,后减量操作符与后增量操作符规则相同。

前增量操作符

在表达式

++ 单目表达式

中,单目表达式为操作数,它必须为标量类型且是可修改的左值。前增量操作符的操作数在表达式求值前递增 1,整个表达式的值为增量后操作数的值。用于增量的 1 是与操作数类型相对应的。指针类型遵循指针算术规则。

前减量操作符

其语法如下:

—— 单目表达式

除了操作数在整个表达式求值之前递减 1 之外,它遵循与前增量操作数的同样规则。

6.6.2 单目操作符

除++、--外,其它 6 个单目操作符为&、*、+、-、~和! 其语法为

单目操作符 强制表达式

取地址操作符 &

操作符 & 和 * 互为逆运算,称为引用和间接引用操作符。在表达式

& 强制表达式

中,强制表达式操作数必须是函数指明符,或者是指明一对象的左值,但该对象不能为位域,也不可说明为 register 存储类指明符。若操作数为 type 类型,则其结果为指向 type 的指针类型。

注意有些非左值标识符,如函数名和数组名,当出现在特定上下文时会自动转换为“指向 X 的指针”类型。& 操作符可用于某些对象,但其用途存在冗余性,因而不鼓励使用。例如:

```
type t1 = 1, t2 = 2;
type * ptr = &t1;           //初始化指针
* ptr = t2;                 //与 t1 = t2 作用相同
```

注意 * ptr = &t1 可写成

```
T * ptr;
ptr = &t1;
```

因而进行赋值的是 ptr,而不是 * ptr。一旦 ptr 用地址 &t1 初始化,则可安全地进行间接引用以给出左值 * ptr。

间接引用操作符 *

在表达式

* 强制表达式

中,强制表达式操作数必须为“指向 type 的指针”类型,这里 type 为任一类型,间接引用的结果为 type 类型。若操作数为“指向函数的指针”类型,则结果为函数指示器;若操作数为对象指针,其结果为指明该对象的左值。在下列情况下,间接引用的结果是无定义的:

- ① 强制表达式为空指针。
- ② 强制表达式为一自动变量的地址,而其相应块已终止。

单目加操作符 +

在表达式

+ 强制表达式

中,强制表达式操作数必须为算术类型,其结果为操作数经过必要的整数扩展后的值。

单目减操作符 -

在表达式

- 强制表达式

中,强制表达式的操作数必须为算术类型,其结果则为操作数经过必要的整数扩展后的负值。

按位补操作符~

在表达式~强制表达式

中,强制表达式操作数必须为整型,其结果为经过必要的整数扩展后操作数的按位补。操作数的0值位变为1,1值为变为0。

逻辑非操作符!

在表达式

!强制表达式

中,强制表达式操作数必须为标量类型,其结果为int类型,值为操作数的逻辑非,若操作数非零则结果为1,若操作数为0则结果为0。表达式!E等价于(0 == E)。

6.6.3 sizeof 操作符

sizeof 操作符有两种不同的用法:

sizeof 单目表达式

sizeof(类型名)

其结果都为整常量,它按字节为单位给出操作数占用存储空间的大小。第一种用法只确定操作数表达式的类型,而不对其求值(因而不会有副作用)。当操作数为不带参量的数组类型时,结果为数组的整个大小(换句话说,数组名不转换为指针类型)。类型中的元素数等于sizeof array/sizeof array[0]。

若操作数为说明为数组或函数类型的参量,则sizeof给出指针的大小。当sizeof用于结构和联合时,它给出它们的整个大小,包括编译程序所加的填补。

sizeof不能用于函数类型、不完整类型及在括号里有这些类型名的表达式中,也不能用于指明位域对象的左值里。

sizeof的结果为整类型size_t,它在stddef.h里定义为unsigned int。可以把sizeof用于预处理器指令中,这仅对Turbo C有效。

6.6.4 乘法类操作符

存在三个乘法操作符:* /和%。其语法为:

乘法类表达式:

强制表达式

乘法类表达式 * 强制表达式

乘法类表达式 / 强制表达式

乘法类表达式 % 强制表达式

*(乘)和/(除)的操作数必须为算术类型,%(求模、求余)的操作数必须为整类型。通常的算术转换要在操作数上进行。

(op1 * op2)的结果为两相乘数的积。(op1/op2)和(op1%op2)的结果为假定op2非零时,op1除以op2的商与余数,当op2为0时将出错。

当op1和op2为整型,而商不是整型时,结果如下:

- ① 若 $op1$ 、 $op2$ 符号相同, 则 $op1/op2$ 为小于真正商的最大整数, $op1 \% op2$ 与 $op1$ 符号相同。
- ② 若 $op1$ 、 $op2$ 符号相反, 则 $op1/op2$ 为大于真正商的最小整数, $op1 \% op2$ 与 $op1$ 符号相同。

6.6.5 加法类操作符

有两个加法类操作符: $+$ 和 $-$, 其语法为

加法类表达式:

乘法类表达式

加法类表达式 $+$ 乘法类表达式

加法类表达式 $-$ 乘法类表达式

加法操作符 $+$

$op1 + op2$ 的合法操作数类型为:

- ① $op1$ 和 $op2$ 都是算术类型。
- ② $op1$ 为整类型, $op2$ 为指向对象的指针类型。
- ③ $op2$ 为整类型, $op1$ 为指向对象的指针类型。

在情况①下, 操作数可以进行标准算术转换, 其结果为操作数的算术和。在情况②和③下, 要应用指针算术规则。

减法操作符 $-$

$op1 - op2$ 的合法操作数类型为:

- ① $op1$ 和 $op2$ 都是算术类型。
- ② $op1$ 和 $op2$ 都为指向兼容对象类型的指针(注意可以为未限定类型 `type`, 与限定类型 `const type volatile type` 和 `const volatile type` 兼容)。
- ③ $op1$ 为指向对象的指针类型, $op2$ 为整类型。在情况①下, 操作数可以进行标准算术转换, 其结果为操作数的算术差。在情况②和③下, 要应用指针算术规则。

6.6.6 按位移位操作符

有两个按位移位操作符: $<<$ 和 $>>$, 其语法为移位表达式:

移位表达式:

加法类表达式

移位表达式 $<<$ 加法类表达式

移位表达式 $>>$ 加法类表达式

按位左移操作符 $<<$

在表达式 $E1 << E2$ 里, 操作数 $E1$ 、 $E2$ 必须是整类型, 在 $E1$ 和 $E2$ 上可进行通常的整型扩展, 结果类型为扩展后的 $E1$ 类型。如果 $E2$ 为负数, 且大于等于 $E1$ 的位数, 则操作无定义。

$E1 << E2$ 的结果为 $E1$ 左移 $E2$ 位得到的值, 必要时可以右边填充 0, `unsigned long E` 的左移等价于 $E1$ 乘以 2, 其模为 `ULONG_MAX + 1`。若 $E1$ 为带符号整数, 结果必须仔细解释, 因为符号位可能会改变。

按位右移操作符>>

在表达式 $E1 \gg E2$ 里,操作数 $E1$ 、 $E2$ 必须为整类型。 $E1$ 和 $E2$ 上可进行通常的整型扩展,结果类型为扩展后 $E1$ 的类型。如果 $E2$ 为负且大于等于 $E1$ 的位数,则操作无定义。

$E1 \gg E2$ 的结果为 $E1$ 右移 $E2$ 位的值。若 $E1$ 为 unsigned 类型,必要时要从左边填充 0。若 $E1$ 为 signed 类型,则使用符号位从左边填充($E1$ 为负时填充 1,为正时填充 0)。这种符号位扩展可确保 $E1 \gg E2$ 的符号与 $E1$ 的符号相同。

6.6.7 关系操作符

有 4 个关系操作符: <、>、<= 和 >=,其语法为:

关系表达式:

移位表达式

关系表达式 < 移位表达式

关系表达式 > 移位表达式

关系表达式 <= 移位表达式

关系表达式 >= 移位表达式

小于操作符<

在表达式 $E1 < E2$ 中,操作数必须为下述情况之一:

- ① $E1$ 和 $E2$ 都是算术类型。
- ② $E1$ 和 $E2$ 都为指向限定或未限定兼容对象的指针类型。
- ③ $E1$ 和 $E2$ 都为指向限定或未限定兼容不完整类型的指针。

在情况①下,要进行通常的算术转换。 $E1 < E2$ 的结果为 int 类型。如果 $E1$ 的值小于 $E2$ 的值,结果则为 1(真),否则结果为 0(假)。

在情况②、③下, $E1$ 和 $E2$ 均为指向兼容类型的指针, $E1 < E2$ 的结果取决于两对象正指向的相对位置(地址)。当比较同一结构的结构成员时,“较高”指针表示较后的说明;在数组内,“较高”指针表示较大的下标值。指向同一联合对象成员的所有指针比较结果都相等。

通常,指向不同结构、数组或联合对象的指针比较,或者超出数组对象范围的指针比较将给出无定义的结果。不过,对超过了最后元素一个元素的指针还可有效比较。若 P 指向一数组对象的某元素, a 指向最后元素,则 $p < a+1$ 是允许的,计算值为 1, $a+1$ 不指向数组对象的任何一个元素。

大于操作符>

若 $E1$ 的值大于 $E2$ 的值,则表达式 $E > E2$ 为 1(真),否则,结果为 0(假)。有关算术和指针的比较,其解释与小于操作符相同,并且也适于相同的操作数规则 and 限制。

小于等于操作符<=

类似的,若 $E1$ 的值小于等于 $E2$ 的值,表达式 $E1 \leq E2$ 的结果为 1(真),否则结果为 0(假)。有关算术和指针的比较,其解释与小于操作符相同,并且也适用相同的操作数规则及限制。

大于等于操作符>=

最后,若 $E1$ 的值大于等于 $E2$ 的值,表达式 $E1 \geq E2$ 的结果为 1(真),否则为 0(假)。有关算术和指针的比较规则与限制与小于操作符相同。

6.6.8 相等类操作符

有两个相等类操作符： $==$ 和 $!=$ ，它们进行算术值或指针值间的相等和不相等测试，遵循的规则与关系操作符大致相似。但要注意， $==$ 和 $!=$ 的优先级比关系操作符 $<$ 、 $>$ 、 $<=$ 和 $>=$ 低。而且， $==$ 和 $!=$ 可进行指针类型的相等和不相等比较，而这些比较对关系操作符不允许。其语法为

相等类表达式：

关系表达式

相等类表达式 $==$ 关系表达式

相等类表达式 $!=$ 关系表达式

等于操作符 $==$

在表达式 $E1 == E2$ 里，操作数必须满足下述条件之一：

- ① $E1$ 和 $E2$ 都为算术类型；
- ② $E1$ 和 $E2$ 都为指向限定或未限定兼容类型的指针；
- ③ $E1$ 和 $E2$ 中一个为指向对象或不完整类型的指针，另一个则为指向限定或未限定 `void` 的指针；
- ④ $E1$ 和 $E2$ 中一个为指针，另一个为空指针常量。

如果 $E1$ 和 $E2$ 具有对关系运算符有效的操作数类型，则 $E1 == E2$ 可应用于 $E1 < E2$ 、 $E1 <= E2$ 等同样的规则。

在情况①下，要进行通常的算术转换， $E1 == E2$ 的结果为 `int` 类型。若 $E1$ 的值等于 $E2$ 的值，则结果为 1(真)，否则结果为 0(假)。

在情况②下，如果 $E1$ 和 $E2$ 指向同一对象，或都指向同一数组对象的“超出最后元素一个元素”，或者都是空指针，则 $E1 == E2$ 。

如果 $E1$ 和 $E2$ 为函数类型指针，且都为空或者都指向同一函数，则 $E1 == E2$ 。反之，若 $E1 == E2$ 为 1(真)，则 $E1$ 和 $E2$ 或者指向同一函数，或者都为空。

在情况④下，指向对象或不完整类型的指针转换为另一操作数类型(指向限定或未限定 `void` 的指针)。

不等于操作符 $!=$

除了当操作数不相等时结果为 1(真)，相等时结果为 0(假)之外，表达式 $E1 != E2$ 遵循与 $E1 == E2$ 相同的规则。

6.6.9 位运算操作符

按位与操作符 $&$

语法为

与表达式：

相等类表达式

与表达式 $&$ 相等类表达式

在表达式 $E1 \& E2$ 中，两操作数必须都为整类型。在 $E1$ 和 $E2$ 上可进行通常的算术转换，结果为 $E1$ 和 $E2$ 的按位与。结果的每一位如表 6.1 所示确定。

表 6.1 位运算操作符真值表

E1 的位值	E2 的位值	E1 & E2	E1 ^ E2	E1 E2
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

按位同或操作符 |

其语法为

同或表达式:

异或表达式

同或表达式 | 异或表达式

在表达式 $E1|E2$ 中,两操作数都必须为整类型,在 $E1$ 和 $E2$ 上可进行通常的算术转换,其结果为 $E1$ 和 $E2$ 的按位同或。结果的每一位由表 5.1 所示确定。

按位异或操作符 ^

其语法为

异或表达式:

与表达式

异或表达式 ^ 与表达式

在表达式 $E1^E2$ 中,两操作数必须都为整类型。在 $E1$ 和 $E2$ 上可进行通常的算术转换,结果为 $E1$ 和 $E2$ 的按位异或。结果的每一位由表 5.1 所示确定。

6.6.10 逻辑运算符**逻辑与操作符 &&**

其语法为

逻辑与表达式:

同或表达式

逻辑与表达式 && 同或表达式

在表达式 $E1\&\&E2$ 中,两操作数都必须为标量类型。其结果为 int 类型,若 $E1$ 和 $E2$ 都非零,则结果为 1(真)否则结果为 0(假)。

不象按位 & 操作符,&& 保证自左至右计算,首先计算 $E1$,若 $E1$ 为 0,则 $E1\&\&E2$ 必定为 0(假),这时 $E2$ 就不再计算了。

逻辑或操作符 ||

其语法为

逻辑与表达式:逻辑与表达式 || 逻辑与表达式

在表达式 $E1||E2$ 中,两操作数都必须为标量类型。其结果为 int 类型,若 $E1$ 和 $E2$ 中有一个非零,则结果为 1(真),否则结果为 0(假)。

不象按位 | 操作符,|| 保证自左至右求值。首先计算 $E1$,若 $E1$ 非零,则 $E1||E2$ 必定为

1(真),这时 E2 就不再计算了。

6.6.11 条件操作符?:

其语法为

条件表达式:

逻辑或表达式

逻辑或表达式? 表达式; 条件表达式

在表达式 E1 ? E2 : E3 中,操作数 E1 必须为标量类型,操作 E2 和 E3 必须遵从以下规则之一:

- ① 两者都为算术类型。
- ② 两者为兼容结构或联合类型。
- ③ 两者都为 void 类型。
- ④ 两者为指向限定或未限定兼容类型的指针。
- ⑤ 一个为指针类型,另一个为空指针常量。
- ⑥ 一个为指向对象或兼容类型的指针,另一个则为指向限定或未限定 void 类型的指针。

首先计算 E1,若其值非零(真)则计算 E2,而忽略 E3。若 E1 值为零(假),则计算 E3,而忽略 E2。E1 ? E2 : E3 的结果为 E2 和 E3 中一个的值。

在情况①下,E2 和 E3 都可进行通常的算术转换,其结果类型为从这些转换中得到的公用类型。

在情况②下,其结果类型为 E2 和 E3 的结构或联合有类型。

在情况③下,其结果为 void 类型。

在情况④和⑤下,其结果类型为指向一类型的指针,该类型用两操作数指向的类型指明符指定。

在情况⑥下,其结果类型为不是指向 void 操作数的指针。

6.6.12 赋值操作符

有 11 个赋值操作符,操作符=为简单赋值操作符,其它 10 个称为复合赋值运算符。其语法为

赋值表达式:

条件表达式

单目表达式 赋值操作符 赋值表达式

赋值操作符如下:

= * = /= %= += -=
<<= >>= &= ^= |= 简单赋值操作符=

在表达式 E1=E2 中,E1 必须为可修改的左值。E2 的值在转换为 E1 类型后,存放由 E1 指明的对象中(代替 E1 的先前值)。赋值表达式的值为 E1 赋值后的值。赋值表达式本身不是左值。

操作数 E1 和 E2 必须遵从下述规则之一:

- ① E1 为限定或未限定算术类型, E2 为算术类型。
- ② E1 为与 E2 类型兼容的限定或未限定结构或联合类型。
- ③ E1 和 E2 为指向限定或未限定兼容类型的指针, 左端指向的类型具有右端指向类型的所有指明符。
- ④ E1 或 E2 中有一个为指向对象或不完整类型的指针, 另一个为指向限定或未限定 void 的指针。左端指向的类型具有右端指向类型的所有指明符。
- ⑤ E1 为一指针, E2 为空指针常量。

复合赋值操作符

复合赋值形如 $op =$, 这里 op 可为 10 个操作符 $*, /, \%, +, -, <<, >>, \&, ^, |$ 。其中任何一个都解释如下:

$E1\ op =\ E2$

与

$E1 = E2\ op\ E2$

作用相同, 只是前者左值 E1 仅计算一次。例如 $E1 += E2$ 与 $E1 = E1 + E2$ 相同。

复合赋值规则与上节相同。

6.6.13 逗号操作符

其语法为

表达式:

赋值表达式

表达式, 赋值表达式

在逗号表达式

$E1, E2$

中, 左边操作数 E1 当作 void 表达式计算, 接着计算 E2, 由此得到逗号表达式的结果和类型。表达式。

$E2, E2, \dots, E_n$

可递归从左至右进行计算, E_n 的值与类型即为整个表达式的结果。由于逗号既可用在函数参数中, 也可用于初始值表时, 为避免由此产生的二义性, 必须使用括号, 例如

$func(i, (j=1, j+4), k);$

是用 3 个参数调用 func, 而不是 4 个参数, 参数值分别为 i、5 和 k。

6.7 高级运算符的使用实例

6.7.1 位运算符

位运算符常用在设备驱动程序中, 例如调制解调程序、磁盘文件程序以及打印驱动程序。因为位运算符能用于屏蔽某些位, 例如奇偶校验位(该位用来确认字节中其他位未变化,

它通常是字节中的最高位)。

就其最常见的用途来说,位运算符 AND 用于将某些位设为 0。也就是说,两个操作数中只要有一个为 0,就会使结果的对应位为 0。例如,下列函数通过库函数 bioscom() 从调制解调器输入口读入一字符,并将奇偶校验位设为 0。函数 bioscom() 用于访问 IBM PC 或兼容机上的异步串行口。

```
char get_char_from_modem(void)
{
    char ch;
    ch = bioscom(2, 0, 0); /* get a character from COM1 */
    return ch & 127;
}
```

奇偶校验位是用第 8 位来表示的。若将该字符与第 1 到第 7 位均为 1 而第 8 位为 0 的数作 AND 运算,则可将该奇偶位设为 0。在上面的表达式中 $ch \& 127$ 表示将 ch 和数字 127 的每一位作 AND 运算。最后的结果是将 ch 的第 8 位设置为 0。在下面的例子中,假定 ch 已接收了字符“A”并已作了奇偶校验位设置。

奇偶校验位	
11000001	ch 中的“A”字符及其奇偶校验位
01111111	二进制的 127
& ——	位逻辑与运算
01000001	奇偶校验位为 0 的“A”字符

位逻辑或运算 OR 与 AND 恰恰相反,可以用于将某些位设置为 1。在两个操作数中只要有一个操作数某一位为 1,则会使 OR 运算结果的对应位为 1。例如, $128 \mid 3$ 为:

10000000	二进制 128
00000011	二进制 3
——	位逻辑或运算
10000011	结果

XOR 是位异或运算,它将两个操作数逐位比较,若不相同则结果的对应位为 1,否则为 0,例如 $127 \wedge 120$ 为:

01111111	二进制 127
01111000	二进制 120
^ ——	位逻辑异或
00000111	结果

总的来说,位运算符 AND、OR 和 XOR 都是直接对变量的每一位单独进行操作的。由于这个原因,通常这些按位运算符不能象关系和逻辑运算符那样用于条件语句中。例如, $X = 7$, 则 $X \& 8$ 的值为真(即 1),而 $X \& 8$ 的值为假(即 0)。

请记住一点,关系运算符和逻辑运算符总是得出非 0 即 1 的结果。而与之类似的位运算符则会得出取决于其特定运算的任意数值。换句话说,位运算符的结果可以是 0 和 1 以外的值,而逻辑运算的结果只能是 0 和 1。

AND 运算符在检查某一位的值时也是非常有用的。例如,下面这条语句检查 status 的

第 4 位是否为 1:

```
if(status & 8) printf("bit 4 is on");
```

用 8 来作 AND 运算是因为它的二进制形式是 00001000。也就是说,数字 8 翻译成二进制数时只有第 4 位为 1。因此,当 status 的第 4 位也为 1 时条件语句为真。这种处理在下面的 disp_binary() 函数中很有用。它显示其参数各位的二进制形式。

```
/* display the bits within a byte */
void disp_binary(int i)
{
    register int t;
    for(t=128; t>0; t = t/2)
        if(i & t) printf("1");
        else printf("0");
    printf("\n");
}
```

函数 disp_binary() 通过用位运算符 AND 确定一个字节中每个位是 1 还是 0 来工作。如果是 1, 则显示数字 "1"; 否则显示数字 "0"。

6.7.2 移位运算符

移位运算符 << 和 >> 将变量中的每一位向左或向右移动。右移运算语句的一般形式是:

变量名 >> 移的位数

左移语句的形式是

变量名 << 移的位数

经过移位后的一端的位被“挤”掉, 而另一端空出的位以 0 填补。记住, 移位并不是循环的。也就是说, 在一端被“挤”出的位并不转回来补到另一端。另一端的位以 0 填补, 而被“挤”出的位则被舍弃了。

移位运算可以用于对外部设备的输入进行译码, 比如 D/A 转换; 也可用于读状态信息。移位运算符还可以用非常快的速度实现整数乘法和除法运算。左移一位实现快速乘 2, 而右移一位则实现快速除以 2。如表 6.2 所示。

表 6.2 用移位运算符做乘除法

X 为每一语句执行	X 值
unsigned char x	00000111 7
x=7;	00001110 14
x<<1;	01110000 112
x<<3;	11000000 192
x<<2;	01100000 96
x>>2;	00011000 24

说明: 左移一位相当于乘以 2。但当执行 $X \ll 2$ 后, X 的最高位因挤出而丢失, 右移一

位相当于除以 2。但随后的移位除法并不能将损失的位补回。

下面的程序形象地表示出移位操作的结果：

```
/* Example of bitshifting. */
#include <stdio.h>
void disp_binary(int i);
main(void)
{
    int i=1, t;
    for(t=0; t<8; t++)
    {
        disp_binary(i);
        i = i << 1;
    }
    printf("\n");
    for(t=0; t<8; t++)
    {
        i = i >> 1;
        disp_binary(i);
    }
    return 0;
}

/* Display the bits within a byte. */
void disp_binary(int i)
{
    register int t;
    for(t=128; t>0; t=t/2)
        if(i & t) printf("1");
        else printf("0");
    printf("\n");
}
```

它产生如下输出：

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

```

0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1

```

尽管 C 语言没有循环移位运算符,但建立一函数实现此功能是十分容易的。循环移位与左移运算很相近,只是需将某一头被挤出的部分加在另一头上。例如,1010 循环左移一位是 0101。实现循环移位的一个方法要求使用具有两种不同类型数据的 union(联合),一种类型是具有想循环移动的数据类型的两个元素的数组,另一种类型是比想移位的数据类型大的类型。在此例中是按位循环左移。

```
union rotate
```

```
{
    char ch[2];
    unsigned int i;
} rot;
```

以下函数执行循环移位:

```
/* Rotate a byte. */
```

```
void rotate_it(union rotate *rot)
```

```
{
    rot->ch[1] = 0;          /* clear the high-order byte */
    rot->i = rot->i << 1;     /* shift once to the left */
    /* see if a bit has been shifted out of ch[0] */
    if(rot->ch[1]) rot->i = rot->i | 1; /* OR it back in */
}
```

函数首先清除整型数 i 的高位字节,这是为了当一位被移进时能够测定出来。对整个数采用左移运算符。移出的 ch[0] 位未被丢弃而到了 ch[1] 中。如果一位被“挤掉”,则与 ch[0] 的低位字节做 OR 运算。下面程序用到了这个函数:

```
/* Do a rotation. */
```

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

```
union rotate
```

```
{
    char ch[2];
    unsigned int i;
} rot;
```

```
void disp_binary(int i);
```

```
void rotate_it(union rotate *rot);
```

```
main(void)
```

```
{
    register int t;
    rot.cxb[0] = 101;
```



```

    for(t=0; t<7; t++)
    {
        disp_binary(rot.i);
        rotate_it(&rot);
    }
    return 0;
}

/* Rotate a byte. */
void rotate_it(union rotate *rot)
{
    rot->ch[1] = 0;          /* clear the high-order byte */
    rot->i = rot->i << 1; /* shift once to the left */
    /* see if a bvit has been shifted out of ch[0] */
    if(rot->ch[1] rot->i = rot->i | 1; /* OR it back in */
}

/* display the bits within a byte */
void disp_binary(int i)
{
    register int t;
    for(t=128; t>0; t=t/2)
        if(i & t) printf("1");
        else printf("0");
    printf("\n");
}

```

原字节循环移位七次后程序产生如下输出：

```

0 1 1 0 0 1 0 1
1 1 0 0 1 0 1 0
1 0 0 1 0 1 0 1
0 0 1 0 1 0 1 1
0 1 0 1 0 1 1 0
0 1 0 1 1 0 0 1

```

反码运算符 \sim 将变量中的每一位都取反。也就是将所有的1都变成0,0变成1。反码运算符的一个有趣的应用是用来观察扩展的字符集合。键盘上显示的字符集合只是计算机支持的全部字符集的一部分。下面的程序将键入的字符中的所有位用反码运算符取反。这些取反了的位对应扩展字符集的一部分。例如,当键入小写字符"d"时,则显示%号。

```

#include <stdio.h>
#include <conio.h>
main(void)
{
    char ch;
    do
    {

```

```

    ch = getch();
    printf("%c", ~ch);
} while(ch != 'q');
return 0;
}

```

按位运算符多用于加密程序。如果想让磁盘文件不可读,可对其进行按位操作。最简单的方法之一是将每一字节的各位求反,如下所示:

源字节	00101100
经过一次求反	11010011
经过两次求反	00101100

注意对一数字连续求两次反码仍可得源码。因此,这一个反码代表那个字节的编码,而第二个则将其破译为原先的值。

可用函数 encode() 对字符进行编码。为破译先前字符编码,只需再调用一次 encode()。

```

char encode(char ch)    /* a simple cipher function */
{
    return(~ch);        /* complement it */
}

```

6.7.3 ?:运算符

运算符?:可以用来代替如下形式的 if else 语句:

```

if(条件)
    表达式
else
    表达式

```

关键是 if 和 else 后都必须是单个表达式,而不能是其他 C 语言语句。

"?:" 是一个三目运算符,因为它要求三个操作数,形式如下:

EXP1 ? EXP2 : EXP3

其中 EXP1、EXP2、EXP3 是表达式。请注意冒号的用法和它的位置。

?: 表达式的值是这样的:先求表达式 1 的值,若为真则求表达式 2 的值并把这作为整个表达式的值。若为假则计算表达式 3 的值并作为整个表达式的值。例如:

```

x = 10;
if(x > 9) y = 100;
else y = 200;

```

不过,"?:" 代替 if else 语句并不只限于赋值语句。所有函数(除了 void 定义的之外)均有返回值。因此用一个或多个函数调用 C 语言表达式是允许的。当遇到函数名时,函数执行且其返回值是确定的。因此,调用由"?:"组成的表达式的一或多个函数是可以执行的。

例如:

```

#include <stdio.h>
int f2(void);
int f1(int n);

```

```
main(void)
{
    int t;
    printf("t:");
    scanf("%d", &t);
    /* print proper message */
    t ? f1(t)+f2() : printf("zero entered");
    return 0;
}

int f1(int n)
{
    printf("%d", n);
}

int f2(void)
{
    printf("entered");
}
```

在这个简单的程序中,如果输入一个零,则调用函数 printf()并显示信息"zero entered"。如果输入其他数,则去执行 f1()和 f2()。

重要的是"?:"表达式的值在此例中被忽略了。把它赋给别的变量是不必要的。然而,即使函数 f1()和 f2()不返回值,它们仍被定义为 int 型。这是必要的,因为一个 void 函数不能使用任何形式的表达式,即使其返回值被忽略。这就是 C 语言的灵活性。

以下是"?"运算符的最后一个例子。它用于防止除以 0 的错误。

```
/* This program uses the ? operator to prevent
   a division by zero. */
#include <stdio.h>
int div__zero(void);
main(void)
{
    int i, j, result;
    printf("Enter dividend and divisor:");
    scanf("%d%d", &i, &j);
    /* this statement prevents a divide by zero error */
    result = j ? i/j : div__zero();
    printf("Result : %d", result);
    return 0;
}

div__zero(void)
{
    printf("cannot divide by zero\n");
    return 0;
}
```

```
}
```

6.7.4 C 语言的简写

C 语言中有一种特殊的简写方式,专门用于简化赋值语句。例如:

```
x=x+10;
```

可以简写为

```
x+=10;
```

运算符+=号告诉编译程序将 X 值加 10 之后再赋给 X。

这种简写方法适用于 C 语言中所有的双目运算符(也就是要求有两个操作数的运算符)。简写的一般形式为:

变量 运算符= 表达式;

再举一个例子:

```
x=x-100;
```

等价于:

```
x-=100;
```

6.7.5 逗号运算符

逗号运算符用于将多个表达式串在一起。逗号运算符左边的值总是不返回的,也就是说,逗号右边表达式的值才是整个表达式的值。例如:

```
x=(y=3, y+1);
```

首先把 3 赋给 y,然后把 4 赋给 x。整个等式右边的表达式要用括号括起来,因为逗号运算符的优先级低于赋值号。

实际上,逗号使一系列的运算逐个执行。逗号用在赋值语句的右边,将一系列表达式逐个分隔开来,而赋给变量的值是最后一个表达式的值。例如:

```
y = 20;
```

```
x = (y=y-5, 30/y);
```

执行后,x 的值为 2,y 的值为 15。因为 y 的初值是 20,减 5 后为 15,再将 30 除以 15 所得结果是 2。

6.7.6 运算符优先级表

图 6.1 列出了所有 C 语言运算符的优先次序。所有的运算符,除了单目运行符和"?:"运算符以外,都是从左到右关联的。而单目运算符"*"、"&"、"."和"?"是从右至左关联的。

6.8 表达式

表达式为操作符、操作数和标点符号的有效序列,其目的是用来进行计算。表达式的语法如下:

基本表达式: 文字量
 伪变量

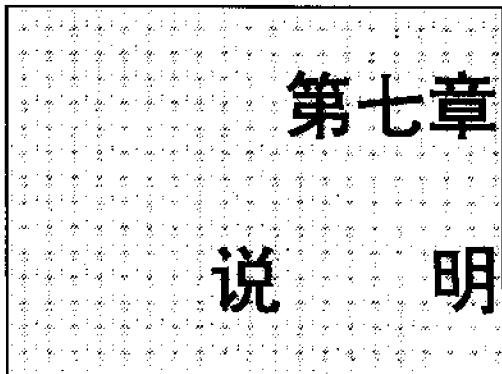
(表达式)
名字
文字量: 整常量
字符常量

高	() [] ->
	1 ~ ++ -- (类型) * & sizeof
	* / %
	+ -
	<< >>
	<<= >>=
	== !=
	&
	^
	&&
	?
低	= += -= *= /=

图 6.1 C 语言运算符优先级

浮点常量
串
名字: 标识符
后缀表达式: 基本表达式
后缀表达式[表达式]
后缀表达式(<表达式>)
后缀表达式 名字
后缀表达式 > 名字
后缀表达式 ++
后缀表达式 --
表达式表: 赋值表达式
表达式表, 赋值表达式
单目表达式: 后缀表达式
++ 单目表达式
-- 单目表达式
单目表达式 强制表达式
sizeof 单目表达式
sizeof(类型名)
单目操作符: & * + - ~ !
强制表达式: 单目表达式
(类型名)强制表达式

基本表达式:	强制表达式
乘法类表达式:	基本表达式 乘法类表达式 * 基本表达式 乘法类表达式 / 基本表达式 乘法类表达式 % 基本表达式
加法类表达式:	乘法类表达式 加法类表达式 + 乘法类表达式 加法类表达式 - 乘法类表达式
移位表达式:	加法类表达式 移位表达式 << 加法类表达式 移位表达式 >> 加法类表达式
关系表达式:	移位表达式 关系表达式 < 移位表达式 关系表达式 > 移位表达式 关系表达式 <= 移位表达式 关系表达式 >= 移位表达式
相等类表达式:	关系表达式 相等类表达式 == 关系表达式 相等类表达式 != 关系表达式
与(AND)表达式:	相等类表达式 与表达式 & 相等类表达式
异或(XOR)表达式:	与表达式 异或表达式 ^ 与表达式
同或(OR)表达式:	异或表达式 同或表达式 异或表达式
逻辑与表达式:	同或表达式 逻辑与表达式 && 同或表达式
逻辑或表达式:	逻辑与表达式 逻辑或表达式 逻辑与表达式
条件表达式:	逻辑或表达式 逻辑或表达式 ? 表达式 : 条件表达式
赋值表达式:	条件表达式 单目表达式 赋值操作符 赋值表达式
赋值操作符:	下述之一 = * = /= %= += -= <<= >>= &= ^= =
表达式:	赋值表达式 表达式, 赋值表达式
常量表达式:	条件表达式



本章首先要回顾一些与说明有关的概念:对象、类型、存储类、作用域、可见性、生存期和连接。然后将详细讲述各种说明的语法及具体使用方法,包括外部说明、类型定义、初始化、简单说明和复杂说明等,此外还给出了一些具体的应用例子。

7.1 有关概念

作用域、可见性、生存期和连接等,确定了程序中为存取对象而使用的标识符的有效区域。

7.1.1 对 象

对象是内存中一可标识区域,它能保存固定或可变的数值以及数值的集合。每个值有一个相关的名字和类型(也称为数据类型)。名字用来存取对象,它可以是一简单的标识符,也可以是一复杂的、可唯一表示某一对象的表达式。类型用来:

- 确定初始时所需的正确的内存大小。
- 在以后存取时解释对象的数据存储模式。
- 在多数类型检查情况下捕获非法赋值。

Turbo C 支持许多标准(预定义)和用户定义的数据类型,包括各种大小的有符号或无符号整数、各种精度的浮点数、结构、联合和数组。另外,在各种存储模式中对应于程序中的每个活动对象的数据存储模式。

说明建立了标识符与对象之间的映射关系,每个说明把一标识符与一数据类型联系起来。大多数说明称为定义型说明,说明的同时包含了对对象的创建(时间和位置),也就是内存的分配及其可能的初始化。其它说明称为引用型说明,它只是让编译程序知道标识符及其类型。同一标识符可能存在多个引用型说明,尤其在多文件程序中。但每个标识符只允许一个定义型说明。

通常来说,标识符只能在程序中说明之后才可使用。这条规则的例外情况,称为向前引用,包括标号、结构、联合的标记和对未说明函数的调用。

7.1.2 左 值

左值是指一个可表示内存中某一存储位置的表达式。如 *P 就是一左值表达式,只要 P 是一个计算值不是空指针的表达式。可修改的左值是指一个与某个对象相联系的标识符或

表达式,该对象可在内存中存取或进行合法的修改。例如,指向某一常量的 `const` 指针就不是一个可修改的左值。指向常量的指针本身可以被修改,但不能修改其间接引用值。

过去, `l` 代表“Left”,意思是左值能放在赋值语句的左端,但现在只有可修改的左值能够放在赋值语句的左端。例如,若 `a`、`b` 为非常量整型标识符,它们占有分配的内存,所以都是可修改的左值,赋值 `a=1` 与 `b=a+b` 都是合法的。

7.1.3 右 值

表达式 `a+b` 不是左值,`a+b=a` 是非法的,因为左端表达式是不与某一个对象相关的。这样的表达式常称为右值。

7.1.4 类型与存储类

要把标识符与对象联系起来,需要每个标识符至少具有两个属性:存储类和类型(有时指数据类型)。Turbo C 编译程序从源代码的隐式或显式说明中得到这两个属性。

存储类说明了对应的单元(数据段、寄存器、堆或栈)及其生存期(在程序的整个运行期间或者是某些代码块的执行期间)。存储类可通过说明的语法、源代码中的位置或者两者一起来建立。

类型确定了有多少个存储单元分配给定义的对象,以及程序如何解释对象在内存中的数据存储模式。一给定的数据类型可看作是该类型的标识符能接受的值的集合(常依赖于实现的具体方法)以及在这些值上允许的操作的集合。通过特殊的操作符 `sizeof` 可确定任何标准类型或用户自定义类型按字节计算的内存大小。

7.1.5 作用域

标识符的作用域是指,该标识符在程序的什么部分能用来存取其对象。作用域一共可分成五类:块、函数、函数原型、文件及作用域与名字空间等,它们依赖于标识符说明的方式和位置。

块作用域

这种标识符的作用域开始于说明点,结束于包含该说明的块尾(这样的块称为封闭块)。函数定义中的参量说明也具有块作用域,它限制在定义该函数的块。

函数作用域

在标识符中,只有语句标号具有函数作用域。标号名可跟在 `goto` 语句后用于说明函数内的任一点。标号通过标号名:后跟语句的形式来隐式说明,标号名在函数内必须是唯一的。

函数原型作用域

在函数原型(非函数定义部分)的参量说明表中说明的标识符具有函数原型作用域,该作用域终止于函数原型的末尾处。

文件作用域

文件作用域标识符,也称为全局量,是在所有块和类的外部说明的,它的作用域从说明点开始到源文件结尾处终止。

作用域与名字空间

名字空间为一标识符在程序中唯一的范围。C 中存在四种不同类型的标识符:

- ① goto 语句的标号名。在所在的函数里,它们必须是唯一的。
- ② 结构、联合和枚举标记。它们在定义它们的块里必须是唯一的。任何函数外部说明的标记,在外部定义的所有标记里必须是唯一的。
- ③ 结构和联合的成员名。它们在定义它们的结构或联合内必须是唯一的,对不同结构或联合里的成员,类型和位置没有任何限制。
- ④ 变量、typedef 和枚举成员。它们在定义它们的作用域内必须是唯一的。外部说明的标识符在外部说明的变量中必须是唯一的。

7.1.6 可见性

标识符的可见性,是指能合法存取标识符的相关对象的源程序代码区域。

作用域和可见性通常是一致的,当然有时会遇到这种情况:由于同名标识符的出现而暂时隐藏了某对象,这时该对象仍存在,但用被隐藏的标识符不能存取它,直到重名的标识符的作用域结束为止。因此,可见性不会超出作用域,但作用域可能会超出可见性。

举例如下:

```
...
{
    int i; char ch;      /* 缺省为 auto 存储类 */
    i = 3;               /* int 型 i 和 char 型 ch 在作用域中且可见 */
    ...
    {
        double i;
        i = 3.0e3;       /* double 型 i 在作用域中且可见 */
                        /* int 型 i 在作用域中但隐藏了 */
        ch = 'A';        /* char 型 ch 在作用域中且可见 */
    }
    i += 1;              /* double 型 i 超出作用域 */
                        /* int 型可见且等于 4 */
    ...                  /* char 型 ch 仍在作用域中可见且等于 'A' */
}
...                    /* int 型 i 和 char 型 ch 超出作用域 */
```

7.1.7 生存期

生存期与存储类密切相关,它定义了标识符相关的对象占有分配内存空间的期限。还应把编译时刻的对象和运行时刻的对象区别开来。例如,变量就不象 typedef 和类型,它在运行时刻具有实际分配的存储单元。存在三种生存期:静态、局部和动态。

静态生存期

具有静态生存期的对象一旦程序运行就马上分配了内存,这一内存一直延续到程序运行终止。静态生存期的对象通常驻留在固定的数据段里,它按照巨型存储模式分配。所有函数,不论在何处定义,都是具有静态生存期的对象。具有文件作用域的所有变量也是静态生存期的。其它变量可用显式的 static 或 extern 存储类指明符来指定是静态生存期的。

具有静态生存期的对象在没有显式初始值或构造函数时初始化为 0(或 null)。

静态生存期一定不要与文件作用域相混淆,一对象可以是静态生存期的,但其作用域为局部的。

局部生存期

具有局部生存期的对象,也称为自动对象,其存在是不确定的。当进入到封闭块或函数时,将在栈(或寄存器)中创建它们。当程序退出该块或函数时,它们将被删除。具有局部生存期的对象必须显式初始化,否则它们的内容将不可预料。在说明局部生存期的变量时,可使用存储类指明符 auto,但通常省略 auto,因为 auto 是在块内说明变量的缺省存储类指明符。

具有局部生存期的对象具有局部作用域,因为它在封闭块的外部不存在,反之则不然,具有局部作用域的对象可能具有静态生存期。

当说明变量(如 int、char、float 等)时,存储类指明符 register 也指 auto,但它请求编译程序,尽可能分配一个寄存器给该对象。若存在一个寄存器未用,Turbo C 可分配该寄存器给要说明的局部整型或指针变量。若寄存器都用完了,则该变量被当作 auto 类分配内存,编译程序不发出任何警告或出错信息。

动态生存期

具有动态生存期的对象在程序运行过程中由特定函数调用创建和删除。可使用标准库函数如 malloc,在堆中给它们分配内存,相应的删除过程用 free 或 delete 进行。

7.1.8 编译单元

编译单元指源文件加上所包含的文件,但不包括那些由条件预处理指令去掉的源代码行。语法上,编译单元定义为一外部说明序列:

编译单元:

外部说明

编译单元 外部说明

外部说明

函数定义

说明

“外部”在 C 中有多种含义,这里指在任何函数外面所作的说明,因此它具有文件作用域。同时为对象或函数预定内存空间的说明也称为定义(或定义型说明)。

7.1.9 连 接

可执行程序通常通过分别编译多个独立的编译单元,再与预先存在的库连接生成。当同一标识符在不同的作用域(如不同文件)内,或在同一作用域内说明了多次,将导致出错。连接是让标识符的每个实例正确地与一特定对象或函数联系在一起的过程。所有标识符都具有三个连接属性之一,这三个连接属性与其作用域密切相关:外部连接、内部连接和非连接。这些属性由说明的位置和格式以及存储类指明符 static 或 extern 的显式(或隐式)使用来确定。

具有外部连接属性的特殊标识符的每个实例,在组成程序的全部文件和库中表示同一

对象或函数。具有内部连接属性的特殊标识符的每个实例,仅在一个文件内表示同一对象或函数。具有非连接属性的标识符表示唯一的实体。

外部与内部连接规则如下:

- ① 具有文件作用域的任一对象或文件的标识符,如果其说明含有存储类指明符 `static`,则具有内部连接属性。对 C 而言,如果一标识符在文件内同时具有内、外部两种连接,则该标识符将具有内部连接。
- ② 若一对象或函数的标识符说明含有存储类指明符 `extern`,则该标识符具有与说明为文件作用域的任何可见标识符相同的连接。若不存在这种可见说明,则该标识符具有外部连接。
- ③ 若一函数说明不带有存储类指明符,则其连接带有缺省存储类指明符 `extern`。
- ④ 若一具有文件作用域的对象标识符说明不带有存储类指明符,则该标识符具有外部连接。

下述标识符具有非连接属性:

- ① 不是说明为对象和函数的标识符(如 `typedef` 标识符)。
- ② 函数参数。
- ③ 说明中不含有存储类指明符 `extern` 的具有块作用域的对象标识符。

7.2 说明的语法

所有 6 个互相有关的属性(存储类、类型、作用域、可见性、生存期和连接)都是通过各种不同的说明方式来确定。

说明包括定义型说明(也简称为定义)和引用型说明(有时称为非定义说明)。定义型说明要执行说明和定义两个任务,非定义型说明需要在程序某个地方存在相应的实际定义。引用型说明只是把一个或多个名字引入程序。定义实际上要分配内存给一对象,并且把一标识符与该对象联系在一起。

7.2.1 暂时定义

ANSI C 标准引入了一个新的概念:暂时定义。任何不含有存储类指明符和初始值的外部数据说明都认为是一暂时定义。若说明的标识符出现在以后的定义中,则暂时定义同带有 `extern` 存储类指明符的定义一样处理。换句话说,暂时定义这时变成引用型定义。

若到达编译单元的末尾,还没有出现带有初始值的标识符定义,则暂时定义变为完全定义,定义的对象具有非初始化(零填充)的保留空间。例如:

```
int x;
int x;      /* 合法,保存 x 的一个副本 */
int y;
int y = 4;  /* 合法,y 初始化为 4 */
int z = 5;
int z = 6;  /* 非法,两者都为含有初始值的定义 */
```

7.2.2 可能的说明

能说明的对象范围包括：

- 变量
- 函数
- 类型
- 结构、联合和枚举标记
- 结构成员
- 其它类型数组
- 枚举常量
- 语句标号
- 预处理程序的宏

由于说明语法的递归性，所以能获得非常复杂的说明。这时可以用 typedef 来改善可读性。

说明的完整语法如下：

Turbo C 说明的语法

说明：

<说明指明符表><说明符表>；

汇编说明

函数说明

连接规格说明

说明指明符：

存储类指明符

类型指明符

函数指明符

typedef

说明指明符表：

<说明指明符表>说明指明符

存储类指明符：

auto

register

static

extern

类型指明符：

简单类型名

类指明符

枚举指明符

复杂类型指明符

const

volatile

简单类型名:下述之一

类名

typedef 名

char short int long signed

unsigned float double void

复杂类型指明符:

类键 标识符

类键 类名

enum 枚举名

枚举指明符:

enum<标识符>{<枚举表>}

枚举表:

枚举符

枚举表,枚举符

枚举符:

标识符

标识符=常量表达式

说明表:

说明

说明表;说明符

以下值得注意的是修饰符和限定符的数目和次序存在一定限制,并且只列出一部分修饰符:

说明符表:

初始说明符

说明符表,初始说明符

初始说明符:

说明符<初始值>

说明符:

说明名

修饰符表

指针操作符 说明符

说明符(参数说明表)<限定符表>

说明符[<常量表达式>]

(说明符)

修饰符表:

修饰符

修饰符表 修饰符

修饰符:下述之一

cdecl pascal interrupt

near far huge

指针操作符:

* <限定符表>

限定符表:

限定符表<限定符>

限定符:

const

volatile

说明名:

名字

typedef 名

类型名:

类型指明符<抽象说明符>

抽象说明符:

指针操作符<抽象说明符>

<抽象说明符>(对数说明表)<限定符表>

<抽象说明符>[<常量表达式>]

(抽象说明符)

参数说明表:

<参数说明子表>

参数说明子表,...

参数说明子表:

参数说明

参数说明子表,参数说明

参数说明:

说明指明符 说明符

说明指明符 <抽象说明符>

函数定义:

<说明指明符>说明符<函数初始值>函数体

函数体:

复合语句

初始值:

= 表达式

= {初始值表}

初始值有:

表达式

初始值表,表达式

{ 初始值表,<,> }

{ 初始值表<, > }, 初始值表

7.3 类型说明

7.3.1 外部说明与定义

存储类指明符 `auto` 和 `register` 不能出现在外部说明中(参见“编译单元”一节)。对编译单元中说明为外部连接的每个标识符,不可存在两个或两个以上的外部定义。

外部定义在定义一对象或函数的同时还要分配内存。如果说明为外部连接的标识符用在表达式中(除了作为 `sizeof` 的操作数部分),则该标识符在整个程序中必须只能存在一个外部定义。

Turbo C 允许在数组、结构和联合等外部名的后面进行重新说明,以在前面的说明中增加信息。例如

```
int a[]           /* 不定大小 */
struct mystruct   /* 仅是标记,没有成员说明符 */
...
int a[3] = {1,2,3}; /* 提供大小和初始化 */
struct mystruct {
    int i, j;
};                /* 增加成员说明 */
```

7.3.2 类型指明符

带有一个或多个可选修饰符的类型指明符用来指明标识符的类型:

```
int i;           /* 说明 i 为带符号整型 */
unsigned char ch1, ch2; /* 说明两个 unsigned char 型 */
```

通常情况下,若忽略类型指明符,则缺省假定为类型 `signed int`(或等价的 `int`)。

7.3.3 类型分类

类型基本上分为四类:值、标量、函数和聚集。标量和聚集类型可进一步如下划分:

- 标量:算术、枚举、指针。
- 聚集:数组、结构、联合。

类型也可分为基本类型和派生类型两类。基本类型包括 `void`、`char`、`int`、`float` 和 `double` 以及它们中某些与 `short`、`long`、`signed` 和 `unsigned` 的组合。派生类型包括指向其它类型的指针、对其它类型的引用、其它类型的数组、函数类型、类类型、结构和联合等。

给定任一非无值类型 `type`(带有某些附加条件),可说明派生类型如表 7.1。

表 7.1 说明类型

<code>type t;</code>	类型 <code>type</code> 的一对象
<code>type array[10];</code>	10 个 <code>type</code> : <code>array[0]~[9]</code>

<code>type * ptr;</code>	<code>ptr</code> 是一指向 <code>type</code> 的指针
<code>type &ref = t;</code>	<code>ref</code> 是对 <code>type</code> 的引用
<code>type func(void);</code>	<code>func()</code> 返回 <code>type</code> 类型的值
<code>void func1(type t);</code>	<code>func()</code> 带有一个 <code>type</code> 类型的参量
<code>struct st {type t1,t2};</code>	结构 <code>st</code> 包含两个 <code>type</code> 对象

7.3.4 基本类型

C 语言中有五种基本数据类型:字符型、整型、浮点指针型、双浮点指针型和无值型。这些变量类型的说明分别为 `char`、`int`、`float`、`double` 和 `void`。

基本类型指明符可用下列关键字来构造:

<code>char</code>	<code>float</code>	<code>long</code>	<code>signed</code>
<code>double</code>	<code>int</code>	<code>short</code>	<code>unsigned</code>

其中 `signed` 和 `unsigned` 是可应用于整类型的修饰符。利用这些关键字,可构造整型和浮点型,它们合称为算术类型。包含文件 `limit.h` 含有所有基本类型值范围的定义。表 7.2 也列出了 Turbo C 中每个数据类型的大小和范围。

`char` 类型的变量可表示 8 位的 ASCII 字符,如“A”、“B”、“C”或其它任意 8 位值,`int` 类型的变量可以表示任意不带小数的整数值。通常用于循环控制和条件语句。

浮点型的表示与值域依赖于实现,即每个不同的实现可自由定义它们。Turbo C 采用 I 十进制 E 浮点格式。

`float` 和 `double` 分别为 32 位和 64 位浮点数据类型。`long` 可与 `double` 合用来声明 80 位精度的浮点标识符,如 `long double text _case`。

`float` 和 `double` 类型的变量在需要用小数表示或应用程序需要较大或较小数值的情况下使用。`float` 和 `double` 型变量的区别,在于它们表示的数值范围不同。如表 7.2 所示,一个 `double` 型可表示大于 `float` 型很多倍的数值。

表 7.2 Turbo C 的基本数据类型的大小和范围

类型	位宽	范围
<code>char</code>	8	-128-127
<code>int</code>	16	-32768-32767
<code>float</code>	32	6.4E-38-6.4E+38
<code>double</code>	64	1.7E-308-1.7E+308
<code>void</code>	0	无值

`void` 是一特殊的类型指明符,表示不带任何值。可用在下述情况下:

- 函数说明中的空参量表:

`int func(void);` `/* func()不带参数 */`

- 说明函数不返回值:

`void func(int n);` `/* 返回值为空 */`

- 作为一类属指针:指向 void 的指针为可指向任何类型的类属指针。

```
void * ptr;           /* 以后可设置 ptr 为任一对象的指针 */
```
- 在类型强制转换表达式中:

```
extern int errfunc(); /* 返回一错误码 */
```

```
...
```

```
(void)errfunc();     /* 忽略返回值 */
```

由于包含 void 类型的概念,基本数据类型可以在其说明前面包含各种各样的修饰符。修饰符用于改变基本类型的含义,以便更精确地说明数据类型,可用的修饰符如下:

```
signed
unsigned
long
short
```

修饰符 signed, unsigned, long 和 short 可与字符型和整型配合使用, long 只能与 double 型一起使用。表 7.3 列出了允许的基本类型和修饰符的组合。

char、short、int 和 long 以及它们与 unsigned 的组合都认为是整数类型。整型指明符如表 7.4 所示,其中同一行列出的类型相同。

char、short、int 和 long 一次最多只能在前面加一个 signed 或 unsigned。若单独用关键字 signed 和 unsigned,则分别意指 signed int 和 unsigned int。

在不存在 unsigned 时,通常假定为 signed,唯一例外是 char。Turbo C 可设置 char 的缺省是 signed 或是 unsigned(若未设置,缺省为 signed)。若缺省设置为 unsigned,则 char ch 说明 ch 为 unsigned,需要用 signed char ch 来使缺省失效。同样,若 char 的缺省为 signed,则需要显式 unsigned char ch 来说明 ch 为 unsigned 型。

表 7.3 Turbo C 的基本类型和变址数的所有可能组合

类型	位宽	范围
char	8	-128~127
unsigned	8	0~255
signed char	8	-128~127
int	16	-32768~32767
unsigned int	16	0~65535
signed int	16	-32768~32767
short int	16	-32768~32767
unsigned short int	16	0~65535
signed short int	16	-32768~32767
long int	32	-2147483648~214748647
signed long int	32	0~4294967295
float	32	6.4E-38~6.4E+38
double	64	1.7E-308~1.7E+308
long double	80	6.4E-4932~1.1E+4932

int 一次最多只能加一个 long 或 short。单独使用关键字 long 或 short, 则分别意指为 long int 和 short int。

表 7.4 整 型

char, signed char	缺省 char 设置为 signed
unsigned char	
char, unsigned char	缺省 char 设置为 unsigned
signed char	
int, signed int	
unsigned, unsigned int	
short, short int, signed short int	
unsigned short, unsigned short int	
long, long int, signed long int	
unsigned long, unsigned long int	

ANSI C 没有规定这些类型的大小和内部表示, 而只坚持 short、int 和 long 形成一非递减序列: "short ≤ int ≤ long"。所有这三种类型可以相同。如果要编写针对其它机器的可移植代码, 这一点非常重要。

在 Turbo C 中, 类型 short 和 int 等价, 两者都为 16 位。long 为 32 位。

有符号和无符号整数的区别在于对高位的解释不同。带符号变量都以 2 的补码形式存储, 它用 MB(最高有效位)作为符号位: 0 为正, 1 为负。如负数可由 2 的补码表示: 数值的所有位(不包括符号标志位)取反且加 1, 最后, 符号标志位置 1。对于无符号变量, 所有位用来给出范围 $0 \sim 2^n - 1$ 的 $n-1$ 次方, 其中 n 为 8、16 或 32。

有符号整数在很多算法中非常重要, 但是其数值范围的最大值只为无符号数的一半。例如, 一个以二进制表示的数 32767:

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

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

运行下列小程序将有助于理解有符号和无符号整数的不同解释方式。

```
#include <stdio.h>
/* Show the difference between signed and unsigned
   integers. */
main()
{
    int i;           /* a signed integer */
    unsigned int j;  /* an unsigned integer */
    j = 60000;
    i = j;
    printf("%d %u", i, j);
}
```

以上程序运行时,其输出结果为-5536 60000。原因是表示一个无符号整数的 60000 被有符号整数解释为-5536。%u 为另一种格式代码,表示 printf() 显示输出的是一个无符号整数值。

C 语言允许说明 unsigned、short 或 long 型整数的简写形式,可以省略 int。例如,下面两种方式都说明了一个无符号的整数变量 X:

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

另外 char 类型的变量也可用来表示除 ASCII 字符集以外的值,用于表示-128~127 之间的“小”整数,当某一处不需要大数值整数的时候,可用 char 变量代替这个整数。例如,下面的程序使用一个 char 变量控制在屏幕上打印字母的循环:

```
#include <stdio.h>
main()
{
    char letter;
    for(letter='A'; letter<='Z'; letter++)
        printf("%c", letter);
}
```

注意,在计算机中字符“A”对应 ASCII 码,且 A~Z 的值按升序排列。

7.3.5 标准转换

当使用算术表达式 $a+b$ 时,若 a, b 为不同的算术类型, Turbo C 在计算该表达式之前要进行某些内部转换。这些标准转换包括在不影响精度和一致性的情况下将“较低”级类型升级到“较高”级类型。

下面是 Turbo C 用来转换算术表达式中操作数的步骤:

- ① 任何小整型按如表 7.5 所示转换。由此,与一操作符相关的两个数必为 int(包括 long 和 unsigned 修饰符)、double、float 或 long double 型。
- ② 若有一个操作数是 long double 类型,另一操作数也转换为 long double。
- ③ 若有一个操作数为 double 类型,则另一操作数也转换为 double。
- ④ 若有一操作数为 float 类型,则另一操作数也转换为 float。
- ⑤ 若有一操作数为 unsigned long 类型,则另一操作数也转换为 unsigned long。
- ⑥ 若有一操作数为 long 类型,则另一操作数也转换为 long。
- ⑦ 若有一操作数为 unsigned 类型,则另一操作数也转换为 unsigned。
- ⑧ 两操作数为 int 类型。

表达式的结果与两操作数同类型。

表 7.5 标准算术转换的方法

类型	转换为	转换方法
char	int	零或符号扩展(取决于缺省 char 类型)
unsigned char	int	零填充高位

续表 7.5

类型	转换为	转换方法
signed char	int	符号扩展
short	int	同值
unsigned short	unsigned int	同值
enum	int	同值

7.3.6 特殊的 char、int 与 enum 间的转换

把一个带符号的字符对象(如一个变量)赋给一整型对象,将会自动引起符号扩展。类型 signed char 的对象总要应用符号扩展。类型 unsigned char 的对象转换为 int 时,总设置高位为 0。

若把较长整型转换为较短整型,则会截除较高位,而保持低位不变。一个较短整型转换为较长整型,依赖于较短类型是 signed 还是 unsigned,要分别进行符号位扩展或用零填充新值符号位。

7.3.7 初始化

初始化用来设置一对象(变量、数组、结构等)在内存中的初值。具有自动生存期的对象若不初始化,则其初值是不确定的。而对于具有静态生存期的对象,若不显式初始化,则按下面方式缺省初始化:

- 若为算术类型,初始化为 0;
- 若为指针类型,初始化为 null(空)。

初始化的语法如下:

初始值:

= 表达式
= {初始值表<, >}
(表达式表)

初始值表:

表达式
初始值表, 表达式
{初始值表<, >}
{初始值表<, >}, 初始值表

初始化规则如下:

- ① 初始值表中初始值的个数不能大于被初始化的对象数。
- ② 初始化的项必须是一对象类型或未知大小的数组。
- ③ 出现在以下位置的表达式必须为常量表达式:
 - a. 在具有静态生存期的对象的初始值表中(也允许使用 sizeof 的表达式);
 - b. 在数组、结构或联合的初始值表中(也允许使用 sizeof 的表达式)。
- ④ 如果一标识符的说明具有块作用域,且该标识符具有外部或内部连接,则该标识

符的说明不能带有初始值。

- ⑤ 如果大括号表中的初始值的个数小于结构成员数,则该结构的剩余部分按具有静态生存期的对象的同样方式隐式初始化。

标量类型用一单表达式初始化,表达式也可括在括号中。对象的初值为该表达式的值,有关类型的限制和转换同简单表达式一样。

对联合而言,大括号中的初始值只用来初始化出现在联合说明表中的第一个成员。对于具有自动生存期的结构或联合,初始值必须为下述之一:

- 在下面描述的初始值表。
- 与联合或结构类型相兼容的单表达式。这时,对象的初值为该表达式的值。

7.3.7.1 简单变量的初始化

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

```
type variable _name = constant;
```

例如:

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

```
int first = 0;
```

```
float balance = 126.23;
```

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

初始化变量可减少程序中的代码量从而使程序简化。举一个变量初始化的例子,下面列出的是运行程序的改写本。这个改写本要求从键盘输入一个数字,然后计算从1到输入数字之间的总和:

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

```
main()
```

```
{
```

```
    int t;
```

```
    printf("enter a number, ");
```

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

```
    total(t);
```

```
}
```

```
total(int x)
```

```
{
```

```
    int sum=0,i,count;
```

```
    for(i=0;i<x;i++)
```

```
    {
```

```
        sum=sum+i;
```

```
        for(count=0;count<10;count++) printf(" . ");
```

```
        printf("the current sum is %d\n",sum);
```

```
    }
```

```
}
```

7.3.7.2 数组、结构与联合的初始化

在说明时刻,可用大括号括住的初始值表初始化数组和结构的成员,初始值按下标递增的数组元素顺序或结构成员的次序给出。也可用大括号括住的初始值来初始化联合的第一个成员。例如,可说明一数组 days,以此来统计一个月內一星期的每天出现多少次(假定每天将至少出现一次),则如下说明:

```
int days[7] = {1,1,1,1,1,1,1};
```

使用下述规则可初始化字符数组或宽字符数组:

- ① 可用串文字量(可括在大括号内)初始化字符类型的数组,串中的每个字符(包括空终结符)用来初始化字符数组的连续元素。例如,可说明

```
char name[] = {"Unknown"};
```

它将定义一个八元素的数组,其元素分别为

、'U'(name[0]),'n'(name[1]),...等。

- ② 可用宽串文字量(可括在大括号里)初始化一宽字符数组(与 wchar_t 兼容)。与字符数组一样,宽串文字量的代码初始化宽字符数组中的连续元素。

下面是结构初始化的一个例子:

```
struct mystruct {  
    int i;  
    char str[21];  
    double d;  
} s = {20,"Borland",6.14};
```

复杂的结构成员,如数组或结构,可用嵌套大括号的适当表达式初始化,也可不用大括号,但必须遵循某些规则。

7.4 简单说明

变量标识符的简单说明形式如下:

数据类型 变量 1 <= 初始值 1>, 变量 2 <= 初值 2>,...

这里变量 1、变量 2、... 为任何可带有初始值的不同标识符序列,其中每个变量都说明为该数据类型。例如:

```
int x=1,y = 2;
```

创建两个称为 x,y 的整型变量(初值分别为 1,2)。它们都是定义型说明,要分配内存,并用可选初值初始化。

自动对象的初值可以是任何合法表达式,表达式的计算值与类型兼容。静态对象的初值为常量或常量表达式。

在 C 中,一个变量的名字与其类型无关,这是与其他语言不同之处。

另外需要注意,变量说明的位置在很大程度上影响该变量在程序中的使用范围。一个变量的使用规则部分决定于它被定义的位置,这在语言中称作“范围规则”。正如前面所述,变量说明的位置决定它的生存期。

7.5 存储类指明符

存储类指明符,或叫类型指明符,必须出现在一说明中。存储类指明符可以是下列关键字之一:

```
auto      register    typedef
extern    static
```

7.5.1 存储类指明符 auto 的使用

存储类指明符 auto 仅用于说明具有局部作用域的变量。它表示变量具有局部(自动)生存期,但由于它是所有局部作用域变量说明的缺省存储类指明符,所以使用得很少。

7.5.2 存储类指明符 extern 的使用

存储类指明符 extern 可用在函数和变量的文件作用域或局部作用域中以表示外部连接。对于具有文件作用域的变量,缺省存储类指明符为 extern。当用于变量时,extern 表示变量具有静态生存期(记住函数总具有静态生存期)。

7.5.3 存储类指明符 register 的使用

存储类指明符 register 仅可用于局部变量和函数参量的说明。它等价于 auto,但它要请求编译器应尽可能分配一寄存器给该变量。在多数情况下,分配寄存器能有效地减少程序大小,改善程序的性能。不过,由于 Turbo C 可以自己决定何时使用寄存器变量,所以很少需要使用 register 关键字。

可以从 Turbo C 的 Options | Compile | Optimizations 对话框中选择寄存器变量选择项。若选择 Automatic, Turbo C 将试图分配寄存器,即使没有使用 register 存储类指明符。

7.5.4 存储类指明符 static 的使用

存储类指明符 static 可用于函数和变量的文件作用域和局部作用域の説明中以表示内部连接。static 也表示变量具有静态生存期。若不存在构造函数和显式初始值,静态变量初始化为 0 或 null。

7.5.5 存储类指明符 typedef 的使用

关键字 typedef 表示要定义一新的数据类型指明符,而不是要说明一对象。把 typedef 当作存储类指明符,是出于语法而非功能的相似性。

看下面的两个说明:

```
static long int biggy;
typedef long int BIGGY;
```

第一个说明创建一个 32 位 long int 类型的静态生存期对象 biggy,第二个说明使标识符 BIGGY 成为一个新的类型指明符,但没有创建某个具体对象。BIGGY 以后可用在类型指明符能出现的任何说明中,例如

```
extern BIGGY salary;
```

与

```
extern long int salary;
```

作用相同。虽然这个简单的类型也可用 `#define BIGGY long int` 来实现,但较复杂的 `typedef` 说明比正文替换功能要强得多。

值得注意的是, `typedef` 不创建新数据类型,它只不过给现有类型创建一个便于记忆的别名,这对于简化复杂说明尤为有用:

例如:

```
typedef double (* PFD)();
```

```
PFD array _pfd[10];
```

```
/* array _pfd 为一个大小为 10 的返回值为 double 型的函数指针数组
```

不可把 `typedef` 标识符与其它类型指明符用在一起:

```
unsigned BIGGY pay; /* 非法 */
```

7.6 修饰符

除了存储类指明符外,说明中还可有一些修饰符来改变标识符/对象的映射方法。Turbo C 的所有修饰符列于表 7.6。

表 7.6 Turbo C 修饰符

修饰符	适用范围	用 法
<code>const</code>	仅变量	防止对对象的修改。
<code>volatile</code>	仅变量	防止寄存器分配和优化。告知编译程序该对象可能在计算时进行外部修改。
<code>cdecl</code>	函数	强制 C 参数传递约定。
<code>cdecl</code>	变量	强制全局标识符大小写相关及开头下划线合法。
<code>pascal</code>	函数	强制 Pascal 参数传递约定。
<code>Pascal</code>	变量	强制全局标识符大小写无关及开头下划线非法。
<code>interrupt</code>	函数	函数与编写中断处理程序所需的处理寄存器的内部代码一起编译。
<code>near, far, huge</code>	指针变量	忽略当前存储模式指定的缺省指针类型。
<code>_cs, _ds, _es, _seg, _ss</code>	指针变量	段指针。
<code>near, far, huge</code>	函数	忽略当前存储模式指定的缺省指针类型。
<code>near, far</code>	变量	指示存储器中对象的位置。
<code>export</code>	函数	仅适用于 OS/2, Turbo C 忽略它。
<code>_loadds</code>	函数	设置 DS 指向当前数据段。
<code>_saveregs</code>	函数	在函数执行期间保存寄存器的值(返回值除外)。

7.6.1 const 修饰符

`const` 修饰符防止在对象赋值时产生任何副作用,如增量或减量。`const` 指针指向的对象

虽然可能修改,但其本身不可修改。请看下例:

```
const float pi = 6.1415926;
const maxint = 32767;          /* 单独用 const 与 const int 等价 */
char * const str = "Hello,world"; /* 常量指针 */
char const * str = "Hello,world"; /* 指向常量字符的指针 */
```

这时,下列语句都是非法的:

```
pi = 3.0;          /* 给常量赋值 */
i = maxint++;      /* 增量常量 */
str = "Hi,there!"; /* str 指向其它东西 */
```

不过要注意,函数调用 `strcpy(str, "Hi there!")` 是合法的,因为它是把串文字量 "Hi there!" 以字符形式拷至由 `str` 指向的存储单元。

7.6.2 中断函数修饰符

`interrupt` 修饰符仅适用于 Turbo C, `interrupt` 函数用于 8086/8088 中断向量。Turbo C 将把 `interrupt` 函数和额外的函数入口及出口代码一起编译,使得寄存器(BP、SP、CS 和 IP)作为 C 调用序列或中断处理本身的一部分来保存。这种函数将用 `iret` 指令返回,使得它可用为硬件或软件中断服务。下面是个典型的 `interrupt` 函数定义的例子:

```
void interrupt myhandler()
{
}
```

应说明中断函数为 `void` 类型。中断程序可在任何存储模式中说明。对于除了巨型模式之外的所有存储模式,DS 设置为程序的数据段,而对巨型模式,DS 设置为模块的数据段。

7.6.3 volatile 修饰符

`volatile` 修饰符表示对象可以被修改,不仅在程序中可修改它,而且在程序之外,如中断过程或 I/O 端口,也可修改它。把一对象说明为 `volatile`,则告知编译器在计算含有该对象表达式时,不必关心该对象的值,因为其值(理论上)可在任何时刻改变。它也防止编译器把该变量当作寄存器变量。

```
volatile int ticks;
interrupt timer()
{
    ticks++;
}
wait(int interval)
{
    ticks = 0;
    while(ticks < interval); /* Do nothing */
}
```

本例(假定 `timer()` 与一硬件时钟中断相联系)实现了由参数 `interval` 指定的限时等待。一高度优化的编译程序可能在 `while` 循环里面加载 `ticks` 的值,因为该循环不改变 `ticks` 的值。

7.6.4 cdecl 与 pascal 修饰符

Turbo C 允许程序容易地调用其它语言编写的例程,反之亦然。若要这样混用语言,则必须解决两大问题:标识符和参量传递。

在 Turbo C 中,所有全局标识符保持大小写不变(大写、小写或混合),并在该标识符前加一下划线(_),除非选择了一u 选择项(或设置 Options | Compiler | Code Generation 对话框中的 Generate Underbars 选择项为 Off)。

7.6.4.1 pascal

pascal 语言中,全局标识符不保持原来的大小写,也不在其前面加下划线。Turbo C 可使说明的任何标识符为 pascal 类型,这时,标识符将全部变为大写,且前面不加下划线。如果标识符是函数,这也会影响所用的参量传递序列,参见“函数类型修饰符”一节。

pascal 修饰符为 Turbo C 所独有,它可用于采用 pascal 参量传递序列的函数(和函数指针)。而且,只要 C 程序看出函数是 pascal 类型,说明为 pascal 类型的函数仍可由 C 程序调用。

```
pascal putnums(int i,int j,int k)
{
    printf("And the answers are: %d,%d,and %d\n",
        i,j,k);
}
```

pascal 类型的函数不能带有可变数量的参数,这点不象函数 printf()。正是这一原因,在 pascal 类型的函数定义中不能使用省略号“...”。

-p 命令行编译选择项(或在 Options | Compiler | Code Generation 对话框中的 CallingConvention 选择项 pascal)可使得所有函数(和函数指针)当作 pascal 类型处理。

7.6.4.2 cdecl

若选择了一p 编译选择项,则要确保某些函数应保持它们的大小写不变,且应该在前面加下划线,当它们是另一文件中的 C 标识符时尤其值得注意。可说明这些标识符为 cdecl 来避免这一点(它也影响函数的参量传递)。

main 必须说明为 cdecl,这是因为 C 启动代码总是按 C 调用约定来调用 main。

象 pascal 一样,cdecl 为 Turbo C 所独有,它和函数及函数指针一起使用。cdecl 将使 -p 编译选择项无效,使函数象正规 C 函数一样调用。例如,如果要用 -p 选择项编译上面的程序,但又要使用 printf(),则可写成:

```
extern cdecl printf();
putnums(int i,int j,int k);
cdecl main()
{
    putnums(1,4,9);
}
putnums(int i,int j,int k);
{
    printf("And the answers are: %d,%d,and %d\n",i,j,k);
}
```

}
如果用 -p 选择项编译程序,则运行时间库中的所有函数都需有 cdecl 说明。从头文件 (如 stdio.h) 中将看到每个函数都显式定义为 cdecl。

7.6.5 指针修饰符

Turbo C 有 8 个修饰符会影响间接引用操作符 (*), 即会修改指向数据的指针, 它们是 near, far, huge, _cs, _ds, _es, _seg 和 _ss。

C 可采用多种存储模式编译, 所用的模式确定了指针的内部格式。例如, 如果使用小数据模式 (如微模式、小模式和中模式), 则所有数据指针相对数据段 (DS) 寄存器有 16 位偏移量。如果采用大数据模式 (如紧缩模式、大模式和巨模式), 则所有数据指针为 32 位长, 分别给出段地址和偏移量。

有时, 在使用某一数据模式的同时, 要说明一指针为非当前缺省的大小或格式, 这时就要用到指针修饰符。

7.6.6 函数类型修饰符

near, far 和 huge 也用作函数类型修饰符, 即它们除了修改数据指针之外, 也可修改函数和函数指针。另外, 还可用 _export, _loadds 和 _saveregs 修饰函数。

near, far 和 huge 函数修饰符可与 cdecl 或 pascal 组合在一起, 但不可与 interrupt 一起使用。

当与汇编语言的代码接口, 且该汇编语言与 Turbo C 不使用同一内存分配的格式时, huge 类型的函数很有用。

为使当前存储模式的缺省设置无效, 非 interrupt 函数可说明为 near, far 或 huge。

near 函数采用 near 调用, far 或 huge 函数采用 far 调用指令。

在微型、小型和紧缩型存储模式中, 非限定函数缺省为类型 near。在中型和大型模式下, 非限定函数缺省为类型 far。在巨型存储模式里, 非限定函数缺省为 huge 类型。

huge 函数与 far 函数的不同在于进入 huge 函数时要把 DS 寄存器设置为源模块的数据段地址, 而 far 函数不设置。

_export 修饰符只作分析, 不作处理。它提供与 OS/2 编写的源代码的兼容性。_export 修饰符对 DOS 程序没有任何意义。

_loadds 修饰符表示函数应该象 huge 函数一样设置 DS 寄存器, 但不指定 near 或 far 调用, 因此 _loadds far 与 huge 等价。

_saveregs 修饰符使函数保存所有寄存器的值, 并在返回之前恢复它们。

_loadds 和 _saveregs 修饰符对编写鼠标支持函数这样的低级过程较为有用。

7.7 复杂说明与说明符

简单说明由可选存储类指明符、类型指明符及其它修饰符后跟由逗号隔开的标识符序列组成。

复杂说明由各种指明符和修饰符后跟由逗号隔开的说明符序列组成。在每个说明里, 至

少存在一标识符,它就是要说明的标识符,每个标识符都相关于放在前面的存储类和类型指明符。

说明符的格式表示它在表达式里的解释形式。若 type 为任一类型,存储类指明符为任一存储类指明符,D1,D2 为任意两个说明符,则说明:

存储类指明符 type D1,D2;

表示 D1,D2 一旦出现在某表达式中时,它们被当作存储类为存储类指明符指定的类型的对象处理。说明符中的类型说明将是含有 type 的一短语,诸如“type”、“指向 type 的指针”、“type 的数组”、“返回 type 型的函数”或“返回 type 型的函数指针”等。

例如:

```
int n, nao[], naf[3], *fn, *apn[], (*pan)[],&nr=n;
int f(void), *fnp(void), (*pfn)(void);
```

每个说明符在单 int 对象可出现的表达式中用作右值(有时也作为左值)。标识符的类型按表 7.7 所示方式从其相应的说明符中得到:

表 7.7 复杂说明

说明符语法	type 的隐含类型	举例
type name;	type	int count;
type name[];	type 的数组	int count[];
type name[3];	3 个 type 类型元素的定长数组	int count[3];
type * name	指向 type 的指针	int * count;
type * name[];	指向 type 的指针数组	int * count[];
type * (name)[];	同上	int * (count)[];
type (* name)[];	指向 type 数组的指针	int (* count)[];
type &name;	对 name 的引用	int &count;
type name();	返回 type 的函数	int * count();
type * name();	返回 type 指针的函数	int * count();
type * (name());	同上	int * count();
type (* name)();	返回 type 的函数指针	int (* count)();

注意,在(* name)[]和(* name)()中必须要括号,这是因为数组说明符[]和函数说明符()的优先级高于指针说明符,而*(name[])中的括号是可选的。

第八章

程序控制语句

程序控制语句控制了整个程序执行的流程。从某种意义上说,程序控制语句是任何一种计算机语言的精髓。各种各样的执行控制方式确定了一种语言的风格。C 语言程序的控制语句丰富有力,这是 C 语言受到广大程序员欢迎的原因之一。

程序控制语句可分为四种类型,包括:表达式语句(主要由赋值语句和函数调用组成);条件语句(主要由 if 语句和 switch 语句组成);循环控制语句(主要由 while 语句、for 语句和 dowhile 语句组成)和无条件转移语句(主要由 goto 语句和 return 语句组成)。

本章首先讲述了各种程序控制语句的语法,然后结合实例具体介绍每一种程序控制语句的使用方法。

8.1 程序控制语句的语法

该语句用来规定程序执行的次序,在没有选择语句和转移语句的情况下,程序将按照它在源代码中的次序顺序执行。表 8.1 给出了 Turbo C 的所有语句。表 8.2 给出了每种语句的语法规则。

表 8.1 Turbo C 语句

带标号语句
复合语句
表达式语句
选择语句
循环语句
跳转语句
汇编语句

表 8.2 Turbo C 语句的语法规则

语句表:

语句;

语句表 语句。

带标号语句:

标识符:语句;

```

        case 常量表达式;语句;
        default;语句。

复合语句:
    { <说明表> <语句表> }

表达式语句:
    <表达式>

选择语句:
    if(表达式)语句;
    if(表达式)语句 else 语句;
    switch(表达式) 语句。

循环语句:
    while(表达式) 语句;
    do 语句 while(表达式);
    for (for 初始化语句<表达式>;<表达式>)语句。

for 初始化语句:
    表达式语句

跳转语句:
    goto 标识符;
    continue;
    break;
    return<表达式>;

汇编语句:
    asm 词法符号 换行;
    asm 词法符号;
    asm { 词法符号,<词法符号;> =
        <词法符号;>
    }

说明表:
    说明;
    说明表 说明。

```

8.1.1 带标号语句

带标号语句有下述两种书写方式:

① 标号标识符;语句

标号标识符用作无条件转移语句(goto)的目标,它有自己的名字空间和函数作用域。

② case 常量表达式;语句

default;语句

case 和 default 标号语句只能和 switch 语句一道使用。

8.1.2 复合语句

复合语句,或称块,是被包括在匹配大括号({})里的语句表(可能为空)。从语法上来说,块可以被认为是一个单语句,但它还具有标识符作用域的作用。在某块内说明的标识符的作用域从说明点开始,结束于封闭大括号。块可以嵌套到任意深度。

8.1.3 表达式语句

在任何表达式后加上分号就是表达式语句:

<表达式>;

Turbo C 通过计算表达式来执行表达式语句。这种计算引起的所有操作在执行下一语句之前完成。大多数表达式语句为赋值语句或函数调用。

特殊情况下表达式语句可以是空语句,它只有一个分号(;),空语句不做任何事情。通常在 Turbo C 语法需要有一条语句,但程序并不需要它时,空语句非常有用。

8.1.4 选择语句

选择语句,或称控制流语句,该类语句先测试某些值然后从备选动作中选择一个执行。选择语句有 if、else 和 switch 两种。

if else 语句

基本的 if else 语句格式如下:

if(条件表达式) 真语句 <else 假语句>

条件表达式必须为标量类型,首先要进行计算,若其值为 0(或空指针),条件表达式为假,否则为真。

如果没有 else 子句且条件表达式为假则忽略真语句。

如果带有 else 子句,条件表达式为真的时候执行真语句,否则忽略真语句而执行假语句。

和 Pascal 语言不同的是,Turbo C 没有布尔(Boolean)数据类型。整型或指针类型表达式在条件测试里承担了布尔类型的角色。对关系表达式(a>b),若 a>b 则其值为 int 1,若 a<=b 则其值为 int 0。指针转换时要保证指针能与值为 0 的常量表达式正确比较。空指针的测试可写成 if(~ptr)... 或 if(ptr==0)...

假语句和真语句本身也可以是 if 语句,由此可知条件转移语句可嵌套至任意深度。在使用 if、else 结构时需要小心,即必须保证选择的是相应的语句。不存在 endif 语句,"else"二义性问题的解决方法,是把 else 与在同一块中最后遇到的不带 else 的 if 相匹配,例如:

```
if(x==1)
    if(y==1) puts("x=1 and y=1");
    else puts("x!=1");
```

将得到一个错误的结果。else 与第二个 if 相匹配,而不象表面看上去的那样。正确的结果为"x=1 and y!=1"。这种时候要注意大括号的作用:

```
if (x == 1)
{
```

```
if (y == 1) puts("x = 1 and y = 1");  
}  
else puts("x != 1");          /* 结果正确 */
```

switch 语句

基本的 switch 语句格式如下:

switch(开关表达式) case 语句

switch 语句使控制转移到其中的一个 case 标号语句,具体转移到哪一个取决于开关表达式的值。开关表达式应为整类型。在 case 语句中的任何语句(包括空语句)都要带有一个或多个 case 标号:

case 第 i 个常量表达式:第 i 个 case 语句;

<break 语句>

这里每个 case 常量表达式都可转换为控制表达式类型。

在同一 switch 语句里不能带有重复的 case 常量。

每个 switch 语句至多可以有一个 default 标号语句:

default:缺省语句

计算开关表达式之后,寻找匹配的第 i 个常量表达式,若找到了则控制转移到与 case 标号相匹配的第 i 个 case 语句。

若没有找到匹配的 case 语句,且存在 default 语句,则控制转移到 default 缺省语句。若没有找到匹配,又不存在 default 语句,则不执行 switch 语句里的任何语句。遇到 case 和 default 标号时程序的执行不受影响,控制仍将转移到下一个语句开关。要想在某个 case 语句结束后停止本 switch 语句的执行,应使用 break 语句。

8.1.5 循环语句

循环语句能循环执行一组语句。Turbo C 有三种形式的循环语句:while 语句、do while 语句和 for 语句。

while 语句

while 语句的格式为:

while(条件表达式) 循环语句

循环语句将重复执行,直到条件表达式等于 0(假)时为止。

首先计算条件测试表达式,若其值非零(真),则执行循环语句;若在循环中没有遇到跳转语句退出,则再次计算条件测试表达式。该循环一直重复直到条件表达式为零(假)。

和 if else 语句一样,指针类型表达式可与空指针相比较,于是 while(ptr)... 与

while(ptr != NULL)...

是等价语句。

在没有跳转语句的情况下,循环语句必须以某种方式修改条件表达式的值,或者条件表达式自己在计算中修改值,否则为死循环。

do while 语句

do while 语句的格式为

do 循环语句 while(条件表达式);重复执行循环语句,直到条件表达式的值等于 0

(假)时为止。它与 while 语句的关键区别是条件表达式是后计算的,故循环语句至少要执行一次。

for 语句

for 语句的格式为

```
for( <初始化表达式>; <测试表达式>; <增量表达式> )
    语句
```

for 语句的执行步骤如下:

- ① 若有初始化表达式则先对它进行计算。它通常初始化一个或多个循环计数器,但语法上允许初始化表达式为任意复杂的一组语句。因此,从计数上说任何 C 程序读可写成一个单一的 for 循环。
- ② 按 while 循环规则计算测试表达式,若其不等于 0(真),则执行循环语句。若测试表达式为空就相当于 while(1),即总是为真。若测试表达式的值等于 0(假),则终止 for 循环。
- ③ 增量表达式修改一个或多个循环计数器。
- ④ 执行 for 循环中的语句(可能为空),接着返回第 2 步。

如果可选项为空,相应的分号不能去掉,如:

```
for(;;) { ... }
```

在 C 中初始化表达式也可以是说明。其中,说明的标识符的作用域可达到该控制语句的作用域。例如:

```
for(int i = 1; i <= i; ++i)
{
    if(i... )... /* 正确,可以引用 i */
}
if(i... )... /* 非法;i 已超出其作用域 */
```

8.1.6 跳转语句

跳转语句,或称无条件转移语句,有四种形式的跳转语句:break 语句、continue 语句、goto 语句和 return 语句。

break 语句

break 语句的格式为

```
break;
```

break 语句可用在循环(while、do while 和 for 循环)或 switch 语句内,它起终止循环或 switch 语句的作用。由于循环和 switch 语句可以混合使用,且可以嵌套至任意深度,故要注意保证 break 从正确的循环或开关中退出。其规则为 break 终止最近的外层循环或 switch 语句。

continue 语句

continue 语句的格式为

```
continue;
```

continue 语句仅可用在循环语句中,它把控制转移给 while 和 do 循环的测试条件,或者

转移给 for 循环的增量表达式。

对于嵌套的循环,continue 语句作用于最近的外层循环。

goto 语句

goto 语句的格式为

goto 标号;

goto 语句把控制转移给标号所指示的语句,该标号和 goto 语句必须在同一函数里。

在 C 里,不可以跳过具有显式或隐式初值的说明,除非该说明在内层块里。

return 语句

return 语句的格式为:

return 返回表达式;

除非函数的返回类型为 void,否则函数体内必须至少包含一个 return 语句,这里返回表达式必须为 type 类型,或者通过赋值可转换为 type 类型。返回表达式的值就是函数返回的值。调用函数的表达式,如 func(实参表),必须为 type 类型的右值而不是左值:

```
t = func(arg); /* 正确 */
```

```
func(arg) = t; /* 在 C 中是非法的;在 C 中若 func 的返回类型为引用则合法 */
```

```
(func(arg))++; /* 在 C 中是非法的;在 C 中若 func 的返回类型为引用则合法 */
```

若遇到 return 语句,则终止执行函数调用,若没遇到 return 语句,则一直执行直至函数体的右大括号。

如果返回类型为 void,return 语句可写成

```
{  
    ...  
    return;  
}
```

或者忽略 return 语句。

8.2 if 语句

if 语句的一般格式是:

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

其中 else 语句是任选的。如果条件测试语句的执行结果为真(不等于 0 的任何值),则执行组成 if 目标的语句或块,否则,如果有 else 语句,则执行组成 else 目标的语句或块。与 if 相关联的代码和与 else 相关联的代码只有一个能够被执行。if 和 else 的目标语句可以是单语句或复合语句。

下面我们来研究一个进行数制转换的简单应用程序。从该程序中可以了解 if 是怎样工作的,该程序可进行如下的数制转换:

- 十进制转换为十六进制
- 十六进制转换为十进制

● 十进制转换为八进制

● 八进制转换为十进制

用户首先在一个菜单中选择要转换的类型,然后提示输入被转换的数据,最后显示该数据转换后的结果。

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

下面是该转换程序的源代码,它使用了一系列的 if 语句以决定将执行何种转换。因为等式操作只能匹配一个菜单选择项,所以每运行一次程序只能执行一次转换。这个程序中每个 if 的目标语句都是一个复合语句。

```
#include <stdio.h>
void main()
{
    int choice;
    int value;
    printf("convert: \n");
    printf("    1: decimal to  hexadecimal\n");
    printf("    2: hexadecimal to decimal\n");
    printf("    3: decimal to octal\n");
    printf("    4: octal to decimal\n");
    printf("enter your choice: ");
    scanf("%d",&choice);
    if(choice==1)
    {
        printf("enter decimal value: ");
        scanf("%d", &value);
        printf("%d in hexadecimal is :%xd",value,value);
    }
    if(choice==2)
    {
        printf("enter hexadecimal value:");
        scanf("%x",&value);
        printf("%x in decimal is: %d",value,value);
    }
    if(choice==3){
        printf("enter decimal value: ");
        scanf("%d", &value);
        printf("%d in octal is :%o",value,value);
    }
    if(choice==4)
    {
```

```
printf("enter octal value, ");
scanf("%o",&value);
printf("%o in decimal is :%d", value,value);
}
}
```

8.2.1 else 语句的用法

else 语句必须与一个 if 语句联合使用。如果该 if 语句的条件表达式为真,将执行其目标语句执行。如果为假,则执行 else 的目标语句。下面的程序具体说明了 else 语句的用法。

```
#include<stdio.h>
void main()
{
    int i;
    printf("enter a number: ");
    scanf("%d", &i);
    if(i<0)
        printf("number is negative");
    else
        printf("number is positive or zero");
}
```

该程序判别输入的所有整数是否为负数。

8.2.2 if—else—if 阶梯的用法

if—else—if 阶梯是一种常用的程序结构。下面是这种结构的格式:

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

式中的条件表达式是自顶向下执行的。一旦发现某一个条件为真,则与该条件相联结的语句将被执行,其它部分将被忽略。如果没有一个条件为真,则执行最后的 else 语句。该 else 语句通常被看作为一个缺省条件,也就是说,如果所有的条件测试均为假,则执行最后一个 else 语句。如果最后的 else 语句不存在,则不执行任何操作。

可以使用一个 if—else—if 阶梯来改进前面的数制转换程序。在该程序原来的版本中,每一个 if 语句不论其条件测试语句为真或为假都要执行一遍。这种结构虽然不会影响程序

的功能,但所有的 if 冗余计算降低了程序的效率。下面经过改进的程序解决了这一问题。在这个使用了 if-else-if 阶梯的版本中,一旦有一个 if 语句的条件测试为真,其它的语句将不再执行。

```
#include <stdio.h>
void main()
{
    int choice;
    int value;
    printf("convert,\n");
    printf("      1: decimal to hexadecimal\n");
    printf("      2: hexadecimal to decimal\n");
    printf("      3: decimal to octal\n");
    printf("      4: octal to decimal\n");
    printf("enter your choice: ");
    scanf("%d",&choice);
    if(choice==1)
    {
        printf("enter decimal value: ");
        scanf("%d",&value);
        printf("%d in hexadecimal is : %x",value,value);
    }
    else if (choice==2)
    {
        printf("enter hexadecimal value: ");
        scanf("%x",&value);
        printf("%x om decimal is : %d",value,value);
    }
    else if(choice==3)
    {
        printf("enter decimal value:");
        scanf("%d",&value);
        printf("%d in octal is : %o",value,value);
    }
    else if(choice==4)
    {
        printf("enter octal value: ");
        scanf("%o",&value);
        printf("%o in decimal is : %d",value,value);
    }
}
```

8.2.3 条件表达式

任何一个正确的 C 表达式均可用于 if 语句中,即表达式的类型不仅仅局限于那些包括

关系操作符和逻辑操作符的表达式,只要表达式的计算结果为 0 或不为 0 即可。

例如,下面这个程序从键盘读入两个整数并显示这两个数的商。为避免出现除 0 错误,将使用由第二个数控制的 if 语句。

```
#include <stdio.h>

void main()
{
    int a, b;
    printf("enter two numbers: ");
    scanf("%d%d", &a, &b);
    if(b)
        printf("%d\n", a/b);
    else
        printf("cannot divide by zero\n");
}
```

这样做的理由是:如果 b 等于 0,则 if 的条件为假,执行 else 语句;若 b 不等于 0,则执行除法操作。完全没有必要写成如下的形式:

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

8.2.4 if 语句的嵌套结构

if 语句的嵌套结构是所有程序设计语言中最容易混淆的问题之一。if 嵌套是指将一个 if 语句作为另一个 if 或 else 语句的目标语句。为避免 if 嵌套混淆,必须弄清楚每一对 if 与 else 之间的关联关系。例如下面这个例子:

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

例子中的 else 与哪个 if 相对应呢?C 语言提供了一个简单的规则来解决这个问题。在 C 中,else 语句与同一代码段中没有 else 语句的最近的一个 if 语句相联结。根据这个规则,该 else 与 if(Y)语句联结。为使 else 与 if(X)相联,必须使用花括弧以取消其隐含的联结关系,如下所示:

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

这样,else 就与 if(X)联结,而不再是与 if(Y)相联结了。

8.3 switch 语句

尽管 if—else—if 阶梯可以完成多种形式的测试任务,但它在结构上并不很精巧。在 C 中有一个更方便的多路判断语句 switch。在 switch 语句中可以更有效地完成以上功能。一个变量可以依次地与一系列整数或字符常量相比较,一旦发现了能够匹配的值,则执行与那

个常量相联结的语句或语句序列。常量的次序是无关紧要的。

switch 语句的一般形式为：

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

如果所有的条件都匹配失败，则执行 default 语句。default 是一个选项，如果不存在 default 语句，则当所有的匹配失败时，将不执行任何动作。一旦某一个条件匹配成功，与那个 case 相联结的语句或语句序列将被执行，直到遇到 break 语句为止。

使用 switch 语句要注意以下两点：

- ① switch 语句与 if 语句的区别在于：switch 仅测试相等关系，而 if 语句的条件表达式可以是任何一种条件关系。
- ② 在同一个 switch 语句中，两个 case 常量不能为同一值。但对于未嵌套的两个 switch 语句可以有相同的 case 常量值。

因为 switch 语句比 if—else—if 阶梯更为有效，所以通常可以用 switch 语句将上述的菜单选择程序设计成效率得更高。下面的程序使用一个 switch 语句来实现数制转任务：

```
#include <stdio.h>
void main()
{
    int choice;
    int value;
    printf("    1: decimal to hexadecimal\n");
    printf("    2: hexadecimal to decimal\n");
    printf("    3: decimal to octal\n");
    printf("    4: octal to decimal\n");
    printf("enter your choice: ");
    scanf("%d", &choice);
    switch(choice)
    {
        case 1:
            printf("enter decimal value: ");
            scanf("%d", &value);
            printf("%d in hexadecimal is :%X", value, value);
```

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

8.3.1 default 语句的用法

在 switch 语句中可以使用一个 default 语句,以便在没有发生任何成功的匹配时执行缺省的语句或语句序列。default 语句是一种解决 switch 语句中出现不属于测试条件的情况的好方法。如在数制转换程序中,可以使用一个 default 语句告知用户输入了一个无效的选择项并请用户重新输入,改进后的程序如下所示:

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



```
        printf("enter octal value: ");
        scanf("%o", &value);
        printf("%o in decimal is : %d", value, value);
        break;
    default:
        printf("invalid selection, try again\n");
        break;
}
```

8.3.2 break 语句的用法

虽然在 switch 中一般都需要使用 break 语句,但在语法上它是一个可选项,用于结束与每个常量相联结的语句序列。如果 break 语句被省略,程序在一个 case 语句中的执行将持续到另一个 case 语句中出现 break 语句或 switch 语句结束。下面是一个省略了 break 语句的例子程序:

```
#include <stdio.h>
void main()
{
    int t;
    for(t=0; t<10; t++)
        switch(t)
        {
            case 1:
                printf("Now");
                break;

            case 2:
                printf(" the");
                printf(" time for all good men\n");
                break;

            case 5:
            case 6:
                printf(" to");
                break;

            case 7:
            case 8:
            case 9:
                printf(" .");
        }
}
```

程序执行时,产生下列输出:

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

这个程序说明了可以使用空 case 语句。当某些条件联结相同的语句序列时,可以使用

这种方式。如果省略了 break 语句,可使所用的 case 语句一起执行,在某些情况下会使得程序的效率比较高,而且可以避免输出不必要的重复代码。

注意,与每个标号相联结的语句不是复和语句而是语句序列。整个 switch 语句才是一个复合语句。

8.3.3 switch 语句的嵌套结构

switch 语句允许有嵌套结构,在外层 switch 和内层 switch 包含相同的常量值时不会发生任何冲突。例如,下面的代码段是完全正确的:

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

下面的简单程序说明了如何使用 switch 语句的嵌套结构。该程序要求用户输入售货区域和售货员姓名的头一个字母,然后显示这个人当前的销售额。因为有些售货员的姓名可能具有相同的头字母,所以需要使用 switch 语句的嵌套结构。这里还使用了一个新的标准库函数 toupper,它将一个小写英文字符变为大写。在这个程序中使用该函数可以允许用户输入时不区分大写和小写(与 toupper()相对的是 tolower,它将大写英文字符转换为小写)。toupper()包含于头文件 ctype.h 中。

```
/* A simple regional salesperson database. */
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
void main()
{
    char division ,salesperson;
    printf("Divisions are: East, Midwest, and West\n");
    printf("Enter first letter of division: ");
    division = getche();
    division = toupper(division); /* make uppercase */
    printf("\n");
    switch(division)
    {
```

```

    case 'E':
        printf("salespersons are: Ralph, Jerry, and Mary\n");
        printf("Enter the first letter of salesperson:");
        salesperson = toupper(getche());
        printf("\n");
        switch(salesperson)
        {
            case 'R': printf("sales: $ %d\n", 10000);
                        break;
            case 'J': printf("sales: $ %d\n", 12000);
                        break;
            case 'M': printf("sales: $ %f\n", 14000);
                        break;
        }
        break;
    case 'M':
        printf("salespersons are: Ron, Linda, and Harry\n");
        printf("Enter the first letter of salesperson:");
        salesperson = toupper(getche());
        printf("\n");
        switch(salesperson)
        {
            case 'R': printf("sales: $ %d\n", 10000);
                        break;
            case 'L': printf("sales: $ %d\n", 9500);
                        break;
            case 'H': printf("sales: $ %d\n", 13000);
                        break;
        }
        break;
    case 'W':
        printf("salespersons are: Tom, Jerry, and Rachel\n");
        printf("Enter the first letter of salesperson:");
        printf("\n");
        switch(salesperson)
        {
            case 'R': printf("sales: $ %d\n", 5000);
                        break;
            case 'J': printf("sales: $ %d\n", 9000);
                        break;
            case 'T': printf("sales: $ %d\n", 14000);
                        break;
        }
}

```

```
        break;
    }
}
```

键入 M 选择中西部区域。则外层 switch 语句选中 case 'M'。若要查看 Harry 的销售额则再键入 H, 屏幕将显示 13000。

注意, 一个嵌套 switch 语句中的 break 语句对外层 switch 没有影响。

8.4 循 环

循环是指一系列指令的重复执行, 直到附合某个条件为止。C 语言支持同其它结构化语言相类似的循环结构。C 语言的循环语句是指 for 语句、while 语句和 dowhile 语句。以下将按顺序分别讨论。

8.5 for 循环

for 循环的一般形式为:

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

在这个最简单的形式中, initialization 通常是一个用于设置循环控制变量初始值的赋值语句, condition 通常为一个关系表达式。通过将循环控制变量同某个值比较, 从而决定循环是否继续进行下去。increment 通常规定了控制循环重复的循环控制变量的变化方式。这三个部分之间必须用分号隔开。只要条件为真, for 循环将一直执行。循环直到条件为假时终止。程序退出循环并继续执行 for 循环后面的语句。

下面是一个简单的例子程序, 改程序在终端上打印数字 1~100:

```
#include <stdio.h>
main()
{
    int x;
    for(x=1; x<=100; x++) printf("%d", x);
}
```

在这个程序中, X 的初始值为 1, 在 printf() 函数返回后, X 被加 1, 然后测试它是否小于或等于 100。这个程序将重复执行, 直到 X 大于 100 为止。在这个例子中, X 是循环控制变量, 每次循环重复时, 它都将被改变和测试。

for 循环还可以反向运行。减小而不是增大循环控制变量将导致反向运行循环。例如, 下面的程序在屏幕上打印数字 100~1:

```
#include <stdio.h>
main()
{
    int x;
    for(x=100; x>0; x--) printf("%d", x);
}
```

C 语言对循环控制变量的改变方式没有任何限制。例如,这个循环在 0~100 之间每隔 5 个数打印一个数:

```
#include <stdio.h>
main()
{
    int x;
    for(x=0; x<=100; x=x+5) printf("%d", x);
}
```

for 循环还可以使用复合语句,如下例所示,它将打印 0~99 的平方数:

```
#include <stdio.h>
main()
{
    int i;
    for(i=0; i<100; i++)
    {
        printf("this is i, %d", i);
        printf(" and i squared: %d\n", i * i);
    }
}
```

for 循环的条件检测在循环顶部执行。即如果开始时条件为假,循环内的代码将根本不执行。例如:

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

这个循环永远不会执行,因为当进入循环时,x 和 y 已经相等,故而条件表达式计算为假,无论循环体还是循环的增值部分都将不执行。从而 y 的值仍为 10,输出的结果是在屏幕上显示 10。

8.5.1 for 循环的灵活用法

前面讨论了 for 循环的一般形式。通过一些变化可以提高 for() 循环的灵活性和对某个设计环境的适应性。如下面例子程序所示:

```
#include <stdio.h>
main()
~{
    int x,y;
    for(x=0,y=0; x+y<100; ++x,y++)
        printf("%d", x+y);
}
```

这个程序打印 0~98 之间的偶数。初始化语句和增值语句都是用逗号隔开的语句序列。逗号是一个 C 操作符,其基本含义是“做这个和那个”。每次进行循环的时候,x 和 y 都被增值,最后 x 和 y 必须为某个适当的值以使循环终止。

条件表达式不一定是循环控制变量与某个目标值的比较测试。实际上,条件可以是任何一个有效的 C 表达式。例如,下面的是一个帮助儿童练习加法的程序。如果小孩感到疲劳了,想停止练习的话,在程序请求输入时键入 N 即可。请注意,for 循环中的条件测试部分。该条件表达式将使得 for 循环执行 99 次或直到用户回答“N”为止。

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int i, j, answer;
    char done = ' ';
    for(i=1; i<100 && done! = 'N'; i++)
    {
        for(j=1; j<10; j++)
        {
            printf("what is %d?", i, j);
            scanf("%d", &answer);
            if(answer != i+j)
                printf("wrong\n");
            else
                printf("right\n");
        }
        printf("more? ");
        done=getche();
        printf("\n");
    }
}
```

事实上 for 语句的三个部分均可以由任何有效的 C 表达式组成,没有必要同标准形式下的各部分完全相同。请看下面这个例子:

```
#include <stdio.h>
void prompt()
{
    printf("enter an integer: ");
}
int readnum()
{
    int t;
    scanf("%d", &t);
    return t;
}
void sqnum(int num)
{
    printf("%d\n", num * num);
}
```

```

    }
    void main()
    {
        int t;
        for(prompt(); t=readnum(); prompt())
            sqnum(t);
    }

```

这个程序首先显示一个提示,然后等待用户输入。当输入一个数字后,将显示出它的平方数,然后再提示输入。这个过程将持续到输入 0 为止。main()函数中的 for 语句的每个部分均由提示用户和从键盘读入数据的函数调用组成。如果输入的数字为 0,则条件表达式为假,循环终止,否则将求出数据的平方数。for 循环中的各部分可以不同时存在。实际上每个部分都有一个表达式与之对应。表达式是任选的,例如,这个循环将运行到输入了数字 10 为止:

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

for 语句中的增值部分为空。这意味着每次循环都重复测试 x 是否等于 10,但并不改变 x 的值。如果输入了数字 10,循环条件变为假,循环终止。

8.5.2 无穷的 for 循环

for 循环的最有趣的应用之一是可以形成无穷循环。当组成循环的三个表达式都为空时,就可能产生一个无终止的循环,如下例所示:

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

8.5.3 无穷 for 循环的中断

利用 break 语句的可以中断 for(;)循环。在循环体的任何地方遇到 break 都将使循环立即终止,程序控制转移到循环后面的代码,如下所示:

```

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

```

这个循环将一直运行到输入了字符“A”。

8.5.4 空循环的用法

空语句是一个符合 C 语法定义的语句。这说明 for 循环的循环体也可以为空。这不仅可能提高某个算法的效率,而且还可以产生延时循环。下面举例说明用 for 循环产生延时的方法:

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

8.6 while 循环语句

while 循环的一般形式是:

```
while (condition) statement;
```

statement 可以是一个空语句、单语句或一个复合语句。condition 为任何一个有效的表达式。当该表达式为真时,循环重复执行,否则程序控制将转向循环代码后面的那一行。

下面的例子是一个键盘输入程序,该程序循环收键直至按下字符 A 为止:

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

ch 作为一个局部变量,当 wait_for_char 未执行时,其值不定。当该函数开始执行时 ch 首先初始化为空。然后检测 ch 是否等于 A,因为 ch 已预先初始化为空,检测为真,while 循环开始。每次键盘按下一个键,测试将进行一次。一旦 A 被按下,那么 ch 等于 A,条件为假,退出循环且终止该函数。

同 for 循环相同,while 循环在循环顶部测试条件,这说明循环代码有可能根本执行不到,预先初始化某些值就是为了防止发生这种情况。

例如,下面程序中的 center() 函数使用一个 while 循环输出适当的空格数以便居中输出一个文本行。如果 len 等于 0,即将要居中输出的行为 80 字符长,则不执行循环。该程序还使用了一个叫做 strlen() 的库函数。该函数返回其字符串自变量的长度,它包含于头文件 string.h 中。

```
#include <stdio.h>
#include <string.h>
void center(int len);
void main()
{
    char str[255];
    int len;
    printf("enter a string:");
    gets(str);
    center(strlen(str));
    printf(str);
}

/* compute and output proper number of spaces to
   center a string of len length.
*/
void center(int len)
{
    len = (80-len)/2;
    while(len>0)
    {
        printf("  ");
    }
}
```



```
len--;  
}  
}
```

通常使用单一变量作为 while 循环的条件表达式,可以在循环的任何地方设置其值。例如:

```
func1()  
{  
    int working;  
    working = 1;  
    while(working)  
    {  
        working = process1();  
        if (working)  
            working = process2();  
        if (working)  
            working = process3();  
    }  
}
```

这里,三个程序均可返回假值从而退出循环。

while 循环体中可以是空语句。例如:

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

该循环将简单地循环执行直至键入字符 A。可以在 while 条件表达式中赋值,即使用等号操作符计算其右边操作数的值。

8.7 do while 循环

与 for 和 while 循环在循环顶部测试循环条件不同,do while 循环是在循环底部测试条件的,即一个 do while 循环至少要执行一次。do while 循环的一般形式是:

```
do  
{  
    statement;  
} while (condition);
```

当循环体只有一个语句时,加上大括号是为了提高可读性及避免与 while 混淆(对程序员而不是编译程序)。

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

```
#include <stdio.h>  
main()  
{  
    int num;  
    do  
    {
```

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

do while 循环可以用在菜单选择程序中。因为一个菜单选择程序至少要运行一次。由于是在循环体的底部测试是否有肯定的回答,所以可以重新提示用户直到用户输入了有效的回答为止。下面这段程序显示了如何在数制转换程序的菜单中加入一个 do while 循环:

```
do
{
printf("convert:\n");
printf("      1: decimal to hexadecimal\n");
printf("      2: hexadecimal to decimal\n");
printf("      3: decimal to octal\n");
printf("      4: octal to decimal\n");
printf("enter your choice: ");
scanf("%d",&choice);
}while(choice< 1 || choice>4);
```

在列出选择项以后,程序将循环等待直到输入了一个有效的回答为止。

8.8 循环嵌套

当一个循环中包含了另一个循环时,里面的循环称为嵌套的。嵌套循环提高了解决一些趣味程序设计问题的方法。例如,这个小程序显示数字 1~9 的 1~4 次方幂:

```
/* Display a table of the first four powers of the
   numbers 1 to 9.
*/
#include <stdio.h>
void main()
{
int i ,j, k, temp;
printf("      i      i^2      i^3      i^4\n");
for(i=1; i<10; i++)
{
/* outer loop */
for(j=1; j<5; j++)
{
/* 1st level of nesting */
temp =1 ;
for(k=0; k<j; k++)
/* innermost loop */
temp = temp * i;
printf("%9d", temp);
}
}
```

```
printf("\n");
```

```
}
```

```
}
```

该程序运行时,产生的结果如图 8.1 所示。注意图中每列数字均对齐。这是因为在打印数据的 printf()语句中使用了“最小区域宽度说明符”。如果在 % 和 d 之间有一个数字的话,它告知 printf()加入必要的空格数以达到规定的宽度。这就可以使各列中的数字对齐。

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

图 8.1 power 程序的输入

有时决定内层循环体的执行次数是很重要的。该数等于外层循环的重复次数乘以每次外层循环重复执行时内层循环的重复次数。在这个乘方例子中,外层循环重复 9 次,每次执行时,第二层循环重复 4 次;这样,第二层循环体实际执行了 36 次。最内层循环执行 2 次,所以它总的执行次数为 72。

下面是嵌套循环的最后一个例子,使用嵌套循环的数制转换程序作又一次改进。外层循环使程序一直运行到用户让其停止为止。内层循环一直运行到用户输入一个菜单选择项为止。现在,不是每次执行只转换一个数字,而是一直重复执行直到用户让它停止。这就允许转换一批数据而不必每次重新启动程序。

```
#include <stdio.h>

void main()
{
    int choice;
    int value;
    /* repeat until user says to quit */
    do
    {
        /* make sure that the user specifies a valid option */
        do
        {
            printf("convert, \n");
            printf("    1, decimal to hexadecimal\n");
```

```
printf("    2: hexadecimal to decimal\n");
printf("    3: decimal to octal\n");
printf("    4: octal to decimal\n");
printf("    5: quit\n");
printf(" enter your choice:  ");
scanf("%d", &choice);
} while(choice<1 || choice>5);
switch(choice)
{
    case 1:
        printf("enter decimal value:");
        scanf("%d", &value);
        printf("%d in hexadecimal is: %x",value,value);
        break;
    case 2:
        printf("enter hexadecimal value:  ");
        scanf("%x", &value);
        printf("%x in decimal is : %d", value, value);
        break;
    case 3:
        printf("enter decimal value:  ");
        scanf("%d",&value);
        printf("%d in octal is ,  %o",value,value);
        break;
    case 4:
        printf(" enter octal value:  ");
        scanf("%o", &value);
        printf("%o in decimal is: %d", value,value);
        break;
}
printf("\n");
} while(choice!=5);
}
```

8.9 循环中断

break 语句有两个用途:其一是在 switch 语句中终止 case,其二是控制一个循环的立即终止而忽略一般的循环条件测试。下面对后一个用途加以讨论。

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

```
#include <stdio.h>
void main()
```

```

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

```

这个程序将在屏幕上打印数字 0~100, 然后终止。当计数器为 10 时执行 break 语句立即退出循环, 而越过循环的一般条件测试 $t < 100$ 。

当使用一个外部事件来控制循环的时候, break 语句特别有用。用下面这个测试时间概念的例子可说明这一点。它在程序开始和结束间等待 5 秒钟, 如果用户认为 5 秒钟已到时敲任意键, 如果用户的感觉是正确的则获胜。

```

#include <stdio.h>
#include <time.h>
#include <conio.h>
void main()
{
    long tm;
    printf("This program tests your sense of time! \n");
    printf("When ready, press return, wait five seconds \n");
    printf("and strike any key: ");
    getch();
    printf("\n");
    tm = time(0);
    for(;;)
    if(kbhit())
        break;
    if (time(0) - tm == 5)
        printf("you win !!!");
    else
        printf("Your timing is off");
}

```

这个程序使用了两个库函数: time() 函数以秒数返回当前的系统时间, time() 函数包含于头文件 time.h 中。因为秒数有可能超过一个整数的范围, 因此需用一个 longint 型变量。kbhit() 函数用于检查是否已敲下一个键。如果是则返回真值, 否则返回假值, 它包含于头文件 bios.h 中。

认识到一个 break 语句只导致从最内层循环退出是很重要的。考虑下面这段程序, 它将在屏幕上将数字 1~10 打印 100 次, 每次遇到 break, 程序控制将返回到外层 for 循环代码:

```

for(t=0; t<100; ++t)
{

```

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

一个 switch 语句中的 break 语句只影响其自身,而不影响它所在的循环。

8.10 continue 语句

continue 语句的作用方式与 break 语句类似。它们的区别在于 continue 不是强制终止循环,而是使下一个循环开始,而跳过中间的所有代码。例如,下面的程序将只显示偶数:

```
#include <stdio.h>
main()
{
    int x;
    for(x=0; x<100; x++)
    {
        if(x%2) continue;
        printf("%d", x);
    }
}
```

每当遇到一个奇数时,if 语句都返回真值,因为奇数对 2 求模等于 1,为真。这样,奇数将使 continue 语句执行,导致再一次重复循环,而略过 printf() 语句。

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

一旦遇到终止条件,continue 将强制执行条件测试,这能加快循环的终止。考虑下面的程序,其作用就像一个简单的代码转换器:

```
#include <stdio.h>
#include <conio.h>
void code();
void main()
{
    printf("Enter the letters you want coded. \n");
    printf("Type a s when you are done. \n");
    code();
}
void code()
```

```

{
    char done, ch;
    done=0;
    while(! done)
    {
        ch=getch();
        if(ch=='$')
        {
            done=1;
            continue;
        }
        printf("%c",ch+1);
    }
}

```

利用这个程序可将所有字符分别变为相应的另一个字母,如 a 变为 b 等。当读到 \$ 时,程序将终止执行。因为由 continue 语句作用的条件测试将发现 done 为真而使循环终止。

8.11 goto 语句

许多教科书中都建议不用或尽量少用 goto 语句。虽然 C 语言对 goto 语句的功能作了一些改进,但它仍然不是程序设计环境中的必要控制语句。尽管如此,如果能够灵巧地对之加以运用,在某些程序设计过程中将有一定的益处。在像 C 这样的具有丰富的控制结构、允许使用 break 语句和 continue 语句作附加控制的语言中,一般很少需要使用 goto 语句。大多数程序设计者最担心的是 goto 语句有可能使程序变得混乱而降低了可读性,但有时却能使程序流程更加清晰。下面对 goto 做一个简单的介绍。

goto 操作需要有一个标号。标号是指一个有效的 C 标识符后带一个冒号,而且标号与使用它的 goto 语句必须在同一个函数中。例如,下面从 1 到 100 的循环使用了一个 goto 和一个标号:

```

x=1;
100p1:
    x++;
    if (x<100) goto 100p1;

```

通常在从一个深层嵌套的程序中退出时使用 goto 效果很好,例如下面的代码:

```

for(...)
{
    for(...)
    {
        while(...)
        {
            if(...)
            {

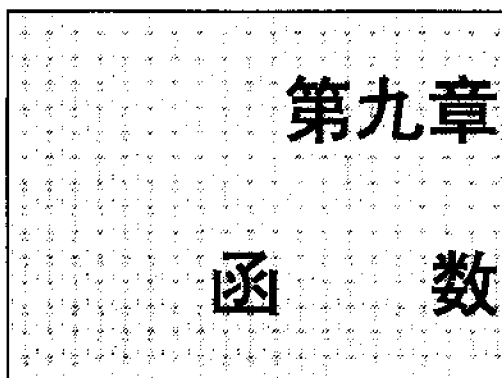
```

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

删除 goto 语句就必须使一些附加测试强制执行。在这里,不能使用简单的 break 语句,因为它只能终止最里层的循环。如果在每个循环中都进行检测的话,则程序将会变成这种形式:

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

使用 goto 语句应该小心或者根本不用。如果没有 goto 语句程序将变得非常难读,或者执行速度不尽人意时,也可使用它。



函数是 Turbo C 编程的中心, C 并不象 Pascal 等语言那样有过程与函数之分, 函数同时具有这两种功用。这一章将讨论函数的细节, 介绍如何构造一些函数, 如何返回不同类型的数据, 以及如何使用函数原型。

除此之外, 本章还将讨论一些作用域规则和变量生存期以及如何构造递归函数和一些有关 main() 函数的特殊性质。

9.1 函数的初步概念

在这一节概括介绍了有关函数的基本内容, 具体的使用将在以下几节分别介绍。

9.1.1 说明与定义

每个程序必须存在唯一一个名为 main 的外部函数, 它标志程序的入口点。函数通常在标准或用户提供的头文件里或者在程序文件中用原型说明。函数缺省为 external, 它通常可被程序中的任何文件存取。函数也可用 static 存储类指明符来限制。

函数可在源程序里定义, 也可通过连接已编译的库或目标文件来得到。

一给定函数可在程序中说明多次, 但这些说明必须是兼容的。未定义的函数说明采用函数原型格式, 使 Turbo C 能知道详细的参量信息, 这样可对参数个数、类型检查和类型转换实施更好的控制。

9.1.2 说明与原型

在 kernighan 和 Ritchie 的原始说明风格中, 函数可由函数调用隐含说明, 也可说明如下:

```
<type> func();
```

其中 type 为可选的返回值的数据类型。这种缺省为 int 的方法不要求编译器检查在函数调用时给出的参数类型与个数是否与说明一致。

利用下述的语法说明函数原型, 可解决这一问题:

```
<type> func(参量说明符表);
```

说明符指定每个函数参量的类型。编译器利用这一信息检查函数调用的合法性。编译器也能强制参数为适当类型。假定存在以下代码:

```

long lmax(long v1, long v2); /* 原型 */
main()
{
    int limit = 32;
    char ch   = 'A';
    long mval;
    mval = lmax(limit, ch); /* 函数调用 */
}

```

由于它带有 lmax 函数原型, 该程序将在调用 lmax 之前, 使用标准赋值规则, 把参量 limit 和 ch 转换为 long 型。若没有函数原型, 则 limit 和 ch 将分别当作整型和字符型。在这种情况下, 调用 lmax 时栈中的参量在大小和内容上都将与 lmax 所希望的不一致, 就有可能发生问题。经典说明风格不要求对参量类型和个数进行任何检查, 因此使用函数原型十分有助于防止编程错误。

函数原型也有助于为代码提供文档。例如, 函数 strcpy() 带有两个参量: 源串和目的串, 使用函数原型:

```
char *strcpy(char *dest, char *source);
```

就可以很清楚的区分它们。如果一头文件含有函数原型, 则将该文件打印出来, 可得到调用那些函数所需的大部分信息。若原型中参量含有一标识符, 则它只用于以后涉及该参量的出错信息中。

参量说明符表若只含有一个 void 说明符, 表示函数不带有任何参数:

```
func(void);
```

若 C 函数要接收可变数目的参量如 printf(), 则函数原型用省略号(...) 结束, 如:

```
printf(int *count, long total, ...);
```

利用这种格式的原型, 定长参量可在编译时进行检查, 变长参量的函数在调用前将不经过任何检查。stdarg.h 中包含了一些宏, 它们可用在用户定义的带有可变数目参量的函数说明里。

下面都是一些函数说明和原型的例子:

```

int f(); /* C 中, 不带有参量信息而返回 int 的函数
          这是 K&R 的“经典风格”。 */
int f(); /* C 中, 不带参量的函数。 */
int f(void); /* 不带参量并返回 int 的函数。 */
int p(int, long); /* 接收两个参量并返回 int 的函数。参量分别为 int 和 long
                    型。 */
int pascal q(void); /* 不带任何参量并返回 int 的 pascal 函数。 */
char far *s(char *source, int kind); /* 带有两个参量并返回指向 char 型的长指针
                                         的函数, 参量分别为 char 指针和 int 型。 */
int printf(char *format, ...); /* 返回 int 的函数, 它接收一个固定参量即指向
                                char 型的指针和任意未知类型的附加参量。 */
int (*fp)(int); /* 接收一个 int 类参量并返回 int 的函数指针 */

```

9.1.3 定 义

外部函数定义的通用语法如表 9.1 给出:

表 9.1 外部函数定义

文件:
外部定义
文件外部定义
外部定义:
函数定义
说明
汇编语句
函数定义:
<说明指明符> 说明符 <说明表> 复合语句

一般来说,函数定义由以下部分组成(语法可允许更复杂的情况):

- ① 可选存储类指明符:extern 或 static,缺省为 extern.
- ② 返回值类型,可能为 void,缺省为 int.
- ③ 可选修饰符:pascal、cdecl、interrupt、near、far、huge。缺省取决于存储模式和编译选择项设置。
- ④ 函数名。
- ⑤ 括在括号里的参量说明,可能为空,在 C 中,表示空表的形式为 funcn(void)。原来风格的 func()在 C 中是合法的,但它可能是不安全的。
- ⑥ 当函数调用时表示执行代码的函数体。

1.4 形参说明

形参说明表遵循正常标识符说明中说明符类似的语法。下面是几个例子:

```
int func(void);           /* 无参 */
int func(T1 t1, T2 t2, T3 t3 = 1); /* 带有三个简单参量,其中一个是缺省参数 */
int func(T1 *ptr1, T2 &tref); /* 带有指针和引用两个参数 */
int func(register int i); /* 为参数请求寄存器 */
int func(char *str, ...); /* 一个串参数加上可变的其它参数个数,或者加上固定数目的类型不同的参数 */
```

C++中,缺省参数可如上给出,带有缺省值的参量必须放在参量表的后部。参数的类型可以是标量、结构、联合、枚举、结构和联合的指针或引用。

省略号“...”表示函数在不同情况下可用不同的一组参数调用。这种格式的原型避免了编译程序进行的类型检查。

说明的参量都有自动作用域和函数生存期。其唯一合法的存储类指明符为 register。const 和 volatile 修饰符可用于形参说明符。

9.1.5 函数调用与参数转换

函数用实参调用,参数书写顺序要与形参相匹配。实参要进行转换,就象通过初始化而转换为形参的说明类型。

下面小结一下 Turbo C 如何处理语言修饰符和函数调用的形参,这些函数可能有也可

能没有原型:

- ① 函数定义的语言修饰符必须与在该函数所有调用点的函数说明中所用的修饰符相匹配。
- ② 函数可以改变形参的值,但对调用时给出的实参不产生影响,除非是 C++ 中的引用参数。

当没有说明函数原型时,Turbo C 将按照标准转换的完整扩充规则把所有参数转换为函数调用的实参。若作用域内存在一函数原型,则 Turbo C 将把给定参数转换为说明的参量类型。

当函数原型中含有省略号“...”,Turbo C 将转换所有在原型中给定的函数参数(直到省略号)。编译程序将按照通常的不带原型的函数参数规则扩充固定参量之外的所有参数。

若存在一原型,则参数个数必须相匹配(除了原型中含有省略号之外)。类型仅需兼容到赋值能合法转换它们的程度。当然,可用显式强制转换把参数转换为函数原型能接收的类型。

如果函数原型与实际函数定义不匹配,仅当该定义和原型在同一编译单元中时,Turbo C 才检测出它。如果要创建一个带有相应原型头文件的函数库,可在编译该库时包含此头文件,C 提供的连接程序能够捕捉到所有原型与实际定义间的不一致。

9.2 return 语句

return 语句有两个重要用途。首先,它能立即从所在函数中退出,也就是 return 语句可以在程序出错时返回到调用点;其次,它可以用于返回一个值。下面将讨论这两种用途。

9.2.1 从一个函数中返回

有两种方法可以终止函数的运行,并返回调用它的调用语句。第一种方法是,执行到函数的最后一条语句,即当遇到了函数的结束符号“}”后即刻返回(实际上大括号不会出现在目标代码中,但是可以认为它采用的是这种方法)。例如,下面一个函数反向显示字符串:

```
void pr_reverse(char *s)
{
    register int t;
    for(t=strlen(s)-1, t>=-1; t-->0) printf("%c", s[t]);
}
```

当字符串显示完后,函数中无其它可执行语句,就返回到调用点。

第二种方法是使用 return 语句。使用该语句可以不带任何值。如下面的函数显示正数的正整次幂,如果指数为负,则在 return 语句处返回,且没有返回值。

```
void power(int base,int exp)
{
    int i;
    if (exp<0) return; /* 负指数不行 */
    i=1;
    for(;exp;exp--)
```

```

        i = base * i;
        printf("the answer is: %d", i);
    }

```

9.2.2 返回值

想要从函数中返回一个值,必须在 `return` 语句后面加上要返回的值,如下函数返回两个实参中的较大数:

```

max(int a, int b)
{
    int temp;
    if(a > b)
        temp = a;
    else
        temp = b;
    return temp;
}

```

注意,由于函数说明中没有明确说明返回类型,因此返回缺省类型整型。

在一个函数中可以包含两个以上的 `return` 语句。有时候,为了简化函数并使某些算法更加有效,经常用到多个 `return` 语句,如下面的 `max()` 函数就比上一个简练:

```

max(int a, int b)
{
    if (a < b)
        return a;
    else
        return b;
}

```

下面的 `find_substr()` 函数使用了两个 `return` 语句。如果没有找到相匹配的子串,函数返回 -1,否则,返回字符串内子串的起始下标。如果使用第二个 `return` 语句,则需增加一个临时变量和一条赋值语句:

```

find_substr(char *srb, char *str)
{
    register int i;
    char *p, *p2;
    for(t=0; str[t]; t++)
    {
        p = &str[t];
        p2 = sub;
        while(*p2 && *p2 == *p)
        {
            p++;
            p2++;
            p++;

```

```

        p2++;
    }
    if (1 * p2)
        return t;          /* if at the end of sub then match */
    }
    return -1;
}

```

如果这个函数用包括“ere”子串的字符串“Hi there”调用,函数返回值 5。

上面两个例子是比使用单个 return 语句更有效的函数,但必须说明,过多的 return 语句会破坏函数结构并且导致语意混乱,所以最好在有必要时才使用多个 return 语句。

除了 void 类型外,所有函数都返回一个值(void 说明的函数将在下节讨论)。这个值由 return 语句明确地指出,如果没有 return 语句则为未知。这就意味着函数可以在有效的 C 语言表达式中作为操作数。因而,在 C 中,下面的几个表达式都是有效的:

```

x=abs(y);
if(max(x,y)>100) printf("greater");
for(ch=getchar();isdigit(ch);) ... ;

```

但是,函数不能作为赋值语句的左值,例如下面的语句是错误的:

```
swap(x,y)=100;    /* incorrect statement */
```

Turbo C 将会显示错误信息,而且不会对这种程序进行编译。

在编制程序时,有返回值的函数通常有三种类型:

第一种用于对一系列的参数作的简单的计算,并且基于该运算返回一个值,相当于一个纯粹的函数。例如库函数 sqrt() 和 sin(),将各自返回一个数的平方根和正弦值。

第二种函数执行某种操作并返回一个简单标明操作成败的值。例如用于往磁盘文件中写数据的 fwrite() 函数。如果写操作成功,fwrite() 返回所写入的字节数。其它值表明发生了错误。

最后一种类型的函数没有明确的返回值。实际上,这种函数是一个单纯的过程。由于一些历史性原因,有些函数的返回值确实令人不感兴趣,但它们还是有返回值。例如,printf() 返回所写入的字符数量,然而程序却很少用到这个值。因此,虽然所有函数,除 void 类型之外,都有返回值,但并不一定都能被用到。函数的返回值未必要赋给其它变量。见下面的例子:

```

#include <stdio.h>
main()
{
    int x,y,z;
    x=10;
    y=20;
    z=mul(x,y);
    printf("%d",mul(x,y));
    mul(x,y);
}

mul(int a,int b)

```

```
{  
    return a * b;  
}
```

main()函数第七行中mul()的返回值赋给了z。第八行中的返回值没有赋给别的变量,然而它被printf()函数用到了。最后一行中的返回值既没有赋值给其它变量,也没有用作表达式的一部分。

9.2.3 函数返回非整型值

没有明确说明返回值类型的函数,其缺省为int型。对多数函数来说使用缺省类型不会发生错误,但是当需要返回其它类型的数据时,必须在函数初次调用之前明确说明函数的返回类型。只有这样,Turbo C才能产生正确的代码,以保证函数能返回所需要的值。

函数返回值可被说明为C语言中任何有效的数据类型。函数的说明方法与变量说明相似,也将类型说明符置于函数名的前面。类型说明符告诉编译程序函数要返回的数据类型。不同的数据类型对长度和内部表达形式有不同的要求,所以如果想使程序正确运行,这个信息是至关重要的。

在使用返回非整型量的函数以前,必须事先对类型加以定义。在C中,最好的办法是用一个函数原型通知编译程序有关函数返回值的类型。

9.2.3.1 函数原型的使用

函数原型担负着两个特殊任务:第一,它要确定函数返回的类型从而使编译程序产生正确的代码;第二,它要确定函数使用的参量的类型和数量。一般采用的形式为:

类型 函数名(参量表);

原型通常在程序的顶部,或者在该函数调用以前被包含的头文件中。

这一节说明如何使用函数原型返回一个非整型的数据类型。

下面这个例函数通过调用函数sum()来说明一个函数如何返回非整型类型的数据,这个函数返回其两个double参数之和:

```
#include <stdio.h>  
double sum(double a,double b);    /* 函数原型 */  
main()  
{  
    double first,second;  
    first =1023.23;  
    second =990.9;  
    printf("%f",sum(first,second));  
}  
double sum(double a,double b)    /* 返回 double 型的值 */  
{  
    return a+b;  
}
```

在用原型通知编译程序函数sum()返回一个双精度浮点数据类型之后,编译程序才能正确地产生调用sum()的代码。如果将原型从这个程序中去掉,将产生错误的结果。

下面是一个更实际的例函数,用于计算一个给定半径的圆的面积:

```
#include <stdlib.h>

float area(float radius);          /* 原型 */

main()
{
    float r;
    printf("Enter radius: ");
    scanf("%f",&r);
    printf("Area is ,%f\n",area(r));
}

float area(float radius)
{
    return 3.1416 * radius * 2;
}
```

函数 area()要返回一个浮点数,成功调用的前提是要让编译程序先知道返回类型是非整型的数据,最好是用函数原型来通知编译程序。

9.2.3.2 返回指针

返回指针的函数处理方式与返回其它任何类型的函数具有相同点,但还有几个重要概念要讨论。

指针既非整型又非无符号整型,而是一个特定类型数据的内存地址。指针的计算与它们指向的基本数据类型有关,如果一个整型函数指针加一,则该指针的值就会比原来增加 2,即指向其类型的下一个数据。编译程序必须知道指针所指数据的类型,才能对不同长度的数据类型,确保指针能正确指到下一个数据。因此,对一个经函数返回的指针也必须加以类型说明。

例如,下面的函数在找到一个指定的字符时,将返回一个指向该字符串中这一位置的指针:

```
char * match(char c,char * s)
{
    int count ;
    count=0 ;
    /* look for match or null terminator */
    while(c!=s[count] && s[count]!='\0')
        count++;
    /* if match, return pointer to location
       other wise return a null pointer */
    if (s[count])
        return (&s[count]);
    else
        return (char *)'\0';
}
```

函数 match()将返回一个指针,该指针指向字符串 s 中第一个与字符 c 相同的字符。如

果未找到与 c 相同的字符,则函数将返回一个空指针(NULL)。下面是一个使用 match() 函数的小程序:

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

char * match(char c, char * s);
main()
{
    char s[80], * p, ch;
    printf("enter a string and a character: ");
    gets(s);
    ch=getche();
    p=match(ch,s);
    if(p) /* there is a match */
        printf("%s\n",p);
    else
        printf("no match found! \n");
}
```

该程序读入一个字符串和一个字符。如果该字符在字符串里面,则从该字符第一次出现处开始打印字符串,否则打印“no match found! ”。例如,输入“Hi there”作为字符串 t 作为字符,程序将输出“there”。

9.2.3.3 函数返回 void 类型

用 void 来描述无返回值的函数可以防止函数在任何表达式中的调用。例如,函数 print _vertical() 打印其字符串参数,竖直排列在屏幕上。因为它不返回任何值,故被说明为 void。

```
void print _vertical(char * str)
{
    while(* str)
        printf("%c\n", * str++);
}
```

任何程序调用 print _vertical() 必须包含一个原型,否则, Turbo C 将指定它返回一个整型量。下面的程序是一正确的例子:

```
#include <stdio.h>
void print _vertical(char * str);
main()
{
    print _vertical("hello");
}
void print _vertical(char * str)
{
    while(* str)
        printf("%c\n", * str++);
}
```

```
}
```

虽然无返回值的函数可被缺省为类型 `int`。但建议将所有无返回值的函数都说明为 `void`，以避免二义性错误。

9.3 有关函数原型的进一步说明

在这一节里，将通过具体例子说明函数原型的其他一些内容。

9.3.1 参数不匹配

原型不只是通知编译程序函数的返回类型，还可防止函数参数的错误调用。虽然 C 能自动将一个变量转化为可接收的参数类型，但是一些类型转换是非法的。通过对函数的原型判断，任何非法类型转换将被发现并给出错误信息。例如，下面的程序调用 `sqr_int()` 时企图用一个整型参数而不是要求的整型指针而产生错误（将整型变为指针是非法的）。

```
/* this Program uses a function prototype to
   enforce strong type checking */
int sqr_int(int *i);    /* 原型 */
main()
{
    int x;
    x=10;
    sqr_int(x);          /* 类型匹配错误 */
}
sqr_int(int *i)
{
    *i = (*i) * (*i);
}
```

下面例函数的错误是因为 `sqr_int()` 调用时参数个数不对：

```
/* this program shows how a function must
   be called with the proper number of arguments */
int sqr_int(int *i);    /* 原型 */
main()
{
    int x;
    x=10;
    sqr_int(&x,10);      /* 参数个数不对 */
}
sqr_int(int *i)
{
    *i = (*i) * (*i);
}
```

由上可见，带原型的函数调用时不允许参数个数错误。

9.3.2 使用头文件

Turbo C 的头文件包含了两个方面的内容:函数的定义和标准函数的原型。例如,STDIO.H 包含了函数 printf() 的原型,所以它需要被包含在几乎所有的程序中。在程序中,通过包含每个调用到的库函数的相应头文件,编译程序可以得知在使用过程中可能出现的任何偶然错误。

在程序中,当一个函数原型与任一个函数都不相关时,可以让 Turbo C 发出警告。在集成环境中使用 Options 菜单或用命令行编译器选择项 _wpro。

9.3.3 无任何参数的函数原型

一个函数原型通知编译程序有关函数返回值的类型以及函数参数的类型和数目。但是 C 语言的最初版本中没有原型部分,所以当需要一个不带任何参数的函数原型时,情况就会变得特殊,因为 ANSI C 标准规定当一个函数原型不包含任何参数时,没有信息用于定义函数参数的类型和数目。为了保证旧式 C 语言程序能被 Turbo C 编译,需要显式通知 Turbo C 一个函数不带任何参数,把关键字 void 放在参数表中。请看下面这个程序:

```
#include <stdio.h>
void myname(void);
main()
{
    myname();
}
void myname(void)
{
    printf("Herb");
}
```

在这个程序中,myname() 的原型明确地说明了它没有任何参数。由于调用参数表必须与其原型保持一致,void 也必须包含在 myname() 的定义部分中。对于这样的原型,Turbo C 不会对如下的 myname() 调用进行编译:

```
myname("Herb");
```

注意,如果在原型的参数表说明中丢掉了 void,则此时不会有任何错误信息。

9.3.4 有关旧式 C 程序

ANSI C 将标准原型加入 C 语言。在此之前,不可能完整地定义函数原型。只有函数的返回值可被说明,其参数却不能被说明,因而编译程序虽然能够为返回值产生正确代码,但却不能保证函数调用时参数的类型、数量符合要求。因此,ANSI C 标准需要与 K&R 标准兼容,它规定不完整的旧原型仍是合法的。这样,原来的 C 程序仍可以用 Turbo C 编译。但是,对于新的 C 程序,通常用完整的原型。下面讨论的旧格式不能用在 C 程序中。

不完整的原型语句一般具有如下形式:

类型定义 函数名()

在说明中不列出函数带有的参数,采用这种方法,前面出现过的 sum() 程序如下:

```
#include <stdio.h>
double sum();           /* 不完整的原型 */
main(void)
{
    double first,second;
    first = 1023.23;
    second = 990.9;
    printf("%f",sum(first,second));
}
double sum(double a,double b)
{
    return a+b;
}
```

这样进行编译不能检测出可能的错误函数调用,运行时会产生不正确的结果。如果想在 C 程序中用原来的代码,则要将任何旧格式原型转换为完整的原型。

9.4 作用域规则

作用域规则决定了一段程序是否能被另一段程序访问。C 语言中的每个函数都是独立的代码块。函数代码对于该函数来说是内部的,除了对函数的调用以外,其它函数中的任何语句都不能访问它(例如,不允许用 goto 语句跳转到其它函数中)。组成函数体的程序代码与程序的其余部分独立,除非使用全局变量或数据,否则不会影响程序的其它部分,也不会受到其它部分的影响,即在一个函数内定义的程序代码和数据,与另一个函数内的程序代码和数据的作用域不同。

对于三种类型变量:局部变量、形式参数和全局变量,作用域决定了它们如何作用于程序的其它部分和它们的生存期。

9.4.1 局部变量

在函数内部说明的变量称为局部变量。实际上在 C 语言中变量说明可以放在任何代码块中。函数的局部变量可以仅被块中的语句说明,即在其本身代码块以外并不知道该变量的存在(代码块以左花括号开始,到右花括号结束)。需要注意的是,仅当程序运行到代码块时,相应局部变量才产生,退出时它们则消失。

函数是最常用的定义局部变量的代码块。例如,考虑下面这两个函数:

```
void func1(void)
{
    int x;
    x=10;
}
void func2(void)
{
    int x;
```

```
x = -199;  
}
```

整型变量 x 被说明两次,一次是在函数 `func1()` 中,另一次在函数 `func2()` 中。`func1()` 中的 x 与 `func2()` 没有任何联系,每个 x 只在定义它的函数体内有效。

为了证明这一点,试运行下列程序:

```
#include <stdio.h>  
void f(void)  
main(void)  
{  
    int x;  
    x=10;  
    printf("x in main is %d\n",x);  
    f();  
    printf("x in main is still %d\n",x);  
}  
void f(void)  
{  
    int x;  
    x=100;  
    printf("the x in f() is %d\n",x);  
}
```

在函数说明中,习惯首先说明函数所需的全部变量。请看下面的函数:

```
void f(void)  
{  
    char ch;  
    printf("continue (y/n)? : ");  
    ch=getche();  
    /* enter this block only if answer is yes */  
    if (ch=='y')  
    {  
        char s[80]; /* this is created only upon entry into this block */  
        printf('enter name: ');  
        gets(s);  
        process_it(s); /* do something */  
    }  
}
```

这里,当程序运行到 `if` 语句以后的代码块(复合语句)时,局部变量 s 才产生,而退出该块时, s 就消失。此外, s 在 `if` 复合语句外无效,即使是在该函数内的其它部分也是如此。

值得重视的是:由于局部变量是随一个函数或复合语句建立和消失的,所以当函数或复合语句运行后局部变量的内容也随之丢失。这意味着在函数调用中局部变量无法返回(但也有例外,后面有更详尽的说明)。

如果不特别加以说明,局部变量是在堆栈中产生的。堆栈是内存中的一个动态变化的区

域,所以局部变量的值不能在调用函数后保存下来。

9.4.2 形式参数

如果一个函数需要使用参数,那么就必须说明变量来接受这些参数。除了用变量接受函数的输入参数,这些参数在函数内部还可以象其它局部变量一样使用。这些变量称为函数的形式参数。说明的形式参数和调用时使用的实际参数必须具有相同的类型。

9.4.3 全局变量

和局部变量不同的是,全局变量在整个程序内都是“可见的”,可以被任一段程序使用。全局变量的作用域是整个程序,在整个程序的运行中存在。全局变量说明是在所有函数之外进行的,可以被任何一个函数中的任何一个表达式使用。

在下面的程序中,可看到变量 count 的说明在全部函数(包括 main()函数)之外。其实,全局变量可以在任何地方说明,但必须在使用它的函数之前且在所有函数之外。而习惯上在程序的一开始对全局变量进行说明。

```
#include <stdio.h>

int count;           /* count is global */
void func1(void);
void func2(void);
main(void)
{
    count = 100;
    func1();
}
void func1(void)
{
    func2();
    printf("count is %d",count); /* will print 100 */
}
void func2(void)
{
    int count;
    for(count=1;count<10;count++)
        printf(".");
}
```

仔细研究一下上面这段程序就可以发现,虽然 main()和 func1()都未曾说明 count,但它们都可以使用该变量。而 func2()中说明了一个叫做 count 的局部变量,当在 func2()中使用变量名 count 时,所指的是局部变量 count,而不是全局变量 count。注意,如果局部变量和全局变量同名,则在说明该局部变量的函数中,该变量名仅代表局部变量,而与同名的全局变量无关。

全局变量在内存中有固定的区域。如果程序有多个函数需要共享某些数据时,可以把这些数据说明为全局变量。但是,出于以下三方面的考虑,应该避免使用不必要的全局变量。

- ① 全局变量在程序的整个运行过程中都占据着内存,而不是仅仅在使用它们时才占用。
- ② 在可使用局部变量的地方使用全局变量,会影响函数的通用性,因为该函数将依赖于在其外部定义的全局变量。
- ③ 大量使用全局变量的程序,容易受程序中一些未知的、不必要的副作用影响而导致错误。

最后一点在 BASIC 中得到了证实,其中所有的变量都是全局的。在编制大型程序过程中所遇到的一个主要问题,是由于某变量在别处被使用而使其值偶然改变。在编制 C 语言程序中,如果使用大量的全局变量则会遇到同样的问题。

结构化语言的一个基本原则是实现分隔化的代码和数据。在 C 语言中,这种分隔化是通过使用局部变量和函数来实现的。例如,计算两个整数乘积的简单函数 MUL(),可以有以下两种写法:

General	Specific
<pre>mul(int x, int y) { return(x * y); }</pre>	<pre>int x,y; mul() { return(x * y); }</pre>

这两个函数都返回变量 x 和 y 的乘积。然而,按一般形式编写,即使用形式参数的函数可以用于返回任何两个整数的乘积,而按另一种特别形式编写的函数只能用于计算全局变量 x 和 y 的乘积。

9.4.4 有关作用域的最后一个例子

一个短程序中的各种作用域在图 9.1 中用图形描述出来。从内层中的代码可了解外层的情况,外层中的代码不会受到内层的作用和影响。

仔细研究一下此图,有助于理解作用域的意义。

```
#include <stdio.h>
#include <string.h>
int count;          /* globe to entire program */
void play(char *p);
main(void)
{
    char str[80];    /* local to main(). */
    printf("enter a string: ");
    gets(str);
    play(str);
}
void play(char *p)  /* p is local to play */
{
    if(! strcmp(p,"add"))
    {
```

```

    int a,b;      /* local to if block inside play */
    printf("enter two integers: ");
    scanf("%d%d",&a,&b);
    printf("%d\n",a+b);
}
else if (! strcmp(p,"beep"))
    printf("%c",7);
}

```

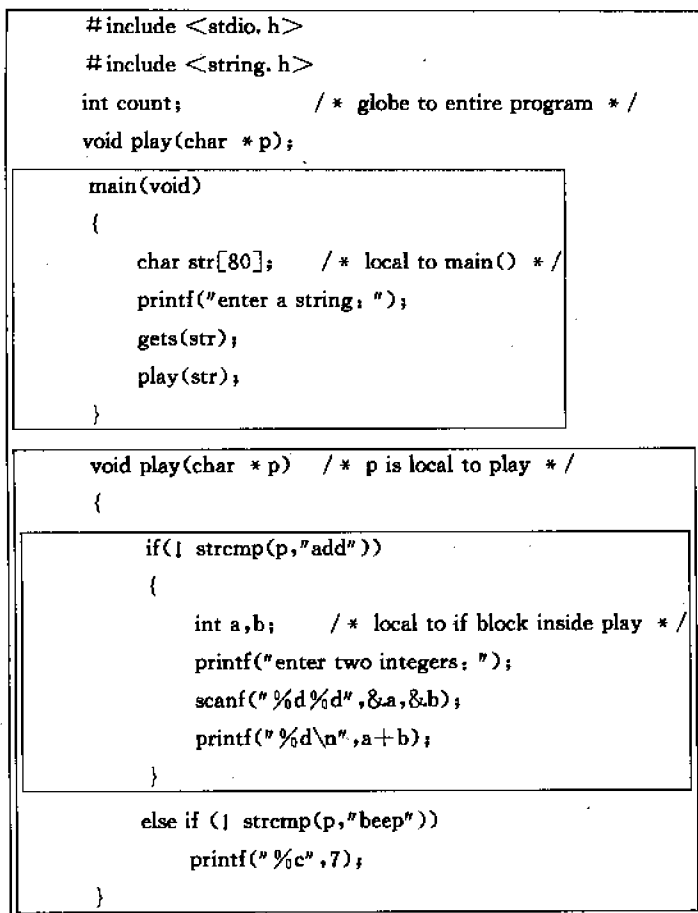


图 9.1 SCOPE 程序的使用域

9.5 有关函数的参数和自变量的更详尽说明

这一节将详细研究 C 语言如何处理函数的参数和自变量。

9.5.1 赋值调用和赋地址调用

函数的参数传递可以是赋值调用和赋地址调用。赋值调用是把实参的值复制给函数的形参,而函数中形参的改变对调用它的实参没有影响。

赋地址调用是把实参的地址复制给形参。在函数中,该地址用来访问在调用程序中的实

参。形参值的改变会影响调用程序中的实参。

C 语言通常的是用赋值调用的方法传递参数,所以通常不能改变调用该函数的变量(在本章的后面介绍了如何使用指针传递来改变调用变量)。考虑下面的函数:

```
#include <stdio.h>

sqr(int x);
main(void)
{
    int t=10;
    printf("%d%d",sqr(t),t);
}

sqr(int x)
{
    x=x*x;
    return(x);
}
```

在这个例子中,sqr()的参数值 10 被复制到形参 x 中。当执行 `x=x*x` 语句时,只改变了局部变量 x,用来调用 sqr()的变量 t 仍然是 10,因此输出结果是 100 和 10。

注意,此时被传递到调用函数的只是实参的值,函数的内部操作并不影响到函数调用时所用的实参。

9.5.2 一个赋地址调用的建立

除 C 语言常规使用的赋值调用外,还可以用传递一个指针给形参的方式来模拟赋地址调用,即将实参的地址传递给该函数,这将使该函数能改变外面变量的值。

在把参数说明为指针类型之后,指针可以如同其它值一样传递给函数。例如,如下所示的函数 swap()用来改换它的两个整型参数的值。

```
void swap(int *x,int *y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

运算符 * 用于访问其操作数所指的变量,因此,用来调用该函数的变量的内容在交换后已经被改变了。

注意,使用指针参数的任何函数被调用时,必须用实参的地址。下面的程序说明了调用 swap()的正确方法。

```
#include <stdio.h>
void swap(int *x,int *y);
main(void)
{
    int x,y;
```

```
x=10 ;
y=20;
printf("initial values of x and y: %d %d\n",x,y);
swap(&x,&y);
printf("swapped values of x, and y: %d %d\n",x,y);
}
```

在上例中,变量 x 被赋值 10, y 被赋值 20, 然后以 x 和 y 的地址作为调用函数 `swap()` 的参数。单目运算符 `&` 用来产生变量的地址, 因此使用 `scanf()` 接收参数时必须在参数之前加上 `&`。函数间变量的地址传递除了会改变调用参数的原值, 在 C 中还能够执行更可靠的地址调用。

9.5.3 数组与函数调用

当数组作为函数的实参时, 只传递数组的地址, 而不是将整个数组复制到函数中去。当用数组名作为实参调用函数, 指向该数组的第一个元素的指针就被传递到函数中。注意, 在 C 语言中, 无任何下标的数组名, 是指向数组第一个元素的指针, 因此要求参数说明中的指针类型必须相同。说明接收数组指针的形参可以有以下三种方式。

第一种是说明一个数组, 如下所示:

```
#include <stdio.h>
void display(int num[10]);
main(void)
{
    int t[10], i;
    for(i=0; i<10; i++) t[i]=i;
    display(t);
}
void display(int num[10])
{
    int i;
    for(i=0; i<10; i++)
        printf("%d", num[i]);
}
```

即使参数 `num` 是被说明为有 10 个元素的整型数组, C 编译程序也会自动地将它转换成一个整型指针。由于形参不能接收整个数组, 而只是将数组指针传递给函数, 所以在调用时不必一一对应。

第二种说明数组形参的方法是说明为一个可变长度的数组, 如下所示:

```
void display(int num[])
{
    int i;
    for(i=0; i<10; i++)
        printf("%d", num[i]);
}
```

这里, num 被说明为可变长度的整型数组。由于 C 编译程序不提供边界检查, 所以数组的实际长度与该形参无关(实际上并不如此)。

最后一种方法是将 num 说明为一个指针, 这也是 C 语言程序最普遍的专业书写形式。如下所示:

```
void display(int * num)
{
    int i;
    for(i=0;i<10;i++)
        printf("%d", num[i]);
}
```

因为任何指针像数组一样可以用[]来作引用, 所以这么做是允许的。

此外, 当数组中的某个元素作为实参时, 像任何其它简单变量一样对待。如下所示:

```
#include <stdio.h>
void display(int num);
main(void)
{
    int t[10], i;
    for(i=0;i<10;i++)
        t[i]=i;
    for(i=0;i<10;i++)
        display(t[i]);
}
void display(int num)
{
    printf("%d", num);
}
```

用户可以看到, display() 中的参数是一个整型数。可以一次只用数组的一个元素来调用 display()。

数组作为函数的实参, 这时传递给函数的是它的地址, 这不同于 C 语言赋值调用的参数传递的习惯用法, 因为这意味着函数对数组进行的操作有可能改变调用该函数的原数组的内容。例如, 考虑函数 print_upper(), 它用大写字母打印作为参数的字符串。

```
#include <stdio.h>
#include <ctype.h>
void print_upper(char * string);
main(void)
{
    char s[80];
    printf("enter a string: ");
    gets(s);
    print_upper(s);
    printf("\noriginal string is altered: %s", s);
}
```

```

void print_upper(char * string)
{
    register int t;
    for(t=0;string[t];++t)
    {
        string[t]=toupper(string[t]);
        printf("%c",string[t]);
    }
}

```

调用 `print_upper()` 函数后,在 `main()` 中数组 `s` 的内容就变成了大写字母。如果不想改变 `s` 原来的内容应该将程序改写为:

```

#include <stdio.h>
#include <ctype.h>
void print_upper(char * string);
main(void)
{
    char s[80];
    printf("enter a string: ");
    gets(s);
    print_upper(s);
    printf("\noriginal string is unchanged: %s",s);
}

void print_upper(char * string)
{
    register int t;
    for(t=0;string[t];++t)
        printf("%c",toupper(string[t]));
}

```

在这段程序中,因为参数 `string` 指向的值没有被更改,原数组 `s` 的内容保留了原样。

将一个数组传递给函数的典型例子是标准库函数 `strcat()`。下面的一段程序将说明它是如何工作的。为了避免与标准库函数混淆,这里函数称为 `hsstrcat()`。`strcat()` 将两个字符串联接起来,并且返回指向第一个字符串的指针。

```

char * hsstrcat(char * s1,char * s2);
main(void)
{
    char s1[80],s2[80];
    printf("Enter two strings: ");
    gets(s1);
    gets(s2);
    hsstrcat(s1,s2);
    printf("concatenated: %s\n",s1);
}

```

```

char *hsstrcat(char *s1, char *s2)
{
    char *temp;
    temp = s1;
    /* first, find the end of s1 */
    while(*s1)
        s1++;
    /* add s2 */
    while(*s2)
    {
        *s1 = *s2;
        s1++;
        s2++;
    }
    *s1 = '\0';
    return temp;
}

```

hsstrcat()函数必须用两个字符数组来调用,用字符指针定义。进入hsstrcat()后,函数找到第一个字符串的末尾,然后把第二个字符串加在其后,最后返回第一个字符串指针。

9.6 argc, argv 和 env —— main 中的参数

在运行程序的时候,有时需要将信息传递给它。传递信息到main()中的最常用的方法是用命令行参数。命令行参数是在操作系统下键入命令行时程序名后的信息。例如,当用命令行方式编译程序时,需要键入如下一些内容:

```
C> tcc filename
```

其中filename是要编译的程序,它作为参数传递给Turbo C 命令行编译器。

main()有三个独有的内部参数:argc, argv 和 env。argc 和 argv 是用来接收命令行参数的,env 在程序开始执行时用于存取DOS 环境变量。

argc 是一个整型量,用以保存命令行中的参数个数,最小值是1,即把程序名作为第一个参数。argv 是指向字符串数组的指针,这个数组里的每个元素都指向给出的命令行参数。所有命令行参数都是以字符串(任何数字都必须由程序转变成为适当的格式)的形式放在argv 数组中的。下面给出的简单程序说明了命令行参数的用法:

```

#include <stdio.h>
#include <process.h>
main(int argc, char * argv[]) /* name program */
{
    if(argc != 2)
    {
        printf("You forgot to type your name\n");
        exit(0);
    }
}

```

```

    }
    printf("Hello %s",argv[1]);
}

```

如果该程序命名为 NAME 并且名字是 Tom, 则运行这个程序时应键入 name tom, 程序输出为“Hello Tom”。例如, 若是在 DOS 下运行程序, 将会看到:

```

C>NAME Tom
hello Tom
C>

```

命令行参数必须由空格(space)或制表符(tab)分隔。分隔符不能用逗号、分号代替。例如 one, two, and three

由四个字符串组成, 而

```
one,two,and three
```

是两个字符串, 这里逗号是非法的分隔符。

如果需要传递一个包含空格的命令行参数, 必须将其置于引号中。例如下面的字符串可被当作一个单独的命令行参数:

```
"this is one argument"
```

正确说明 argv 最常用的方法是:

```
char *argv[];
```

空方括号表示数组是可变长度的。argv 用来访问各个参数, 例如, argv[0] 指向第一个参数, argv[1] 指向第二个参数, 依此类推。在 DOS 3.0 以前的版本中, argv[0] 为空, 而不是要执行的命令。

下面的程序是一个实用的运用命令行参数的例子。程序通过库函数 system() 执行由命令行输入的一系列 DOS 命令。给出任何可执行的 DOS 命令作为参数字符串, 则命令被执行。

```

#include <stdio.h>
#include <stdlib.h>
main(int argc, char *argv[])
{
    int i;
    for(i=1; i<argc; i++)
        system(argv[i]);
}

```

假定此程序名叫 COMLINE, 下面的命令行将执行 DOS 命令 VER、CHKDSK 和 DIR *.C.

```
C>COMLINE VER CHKDSK "DIR *.C"
```

若要访问命令行参数中的单个字符, 则要为 argv 增加第二个下标。例如, 下面的程序是在屏幕上显示所有的参数, 每次显示一个字符:

```

/* The program prints all command line argument it is
called with. */
#include <stdio.h>

```

```
main(int argc, char *argv[])
{
    int t, i;
    for(t=0; t<argc; ++t)
    {
        i=0;
        while(argv[t][i])
        {
            printf("%c", argv[t][i]);
            ++i;
        }
        printf(" ");
    }
}
```

记住,第一个下标用于访问字符串,第二个下标用于访问字符串中的字符。

通常可以用 `argc` 和 `argv` 给程序传递初始命令。在 Turbo C 中,可以设置操作系统允许下尽可能多的命令行参数。在 DOS 中限定一行最多为 128 个字符。通常可以使用这些参数来表示文件名或选择项。使用命令行参数既可以使程序更具专业化,又便于在批处理文件中使用此程序。

参数 `env` 是一个包含环境设置的字符数组的指针。数组中的最后一个字符串为空,作为结束标记。下面这个程序以字符串的形式打印出所有当前的环境设置:

```
#include <stdio.h>
main(int argc, char *argv[], char *env[])
{
    int i;
    for(i=0; env[i]; i++)
        printf("%s\n", env[i])
}
```

注意,因为参数说明具有状态依赖性,所以即使没有用到 `argc` 和 `argv`,在说明参数 `env` 时,也必须同时说明前面的二者。如果 `env` 参数未用到,则不需说明。

当 `main()` 无参数时,通常将 `main()` 的参数说明为 `void`。

9.7 从 `main()` 中返回值

在程序中,可以用 `return` 语句从 `main()` 中返回一个整型数,该返回值通常被操作系统使用。对 DOS 和 OS/2 来说,返回值为 0 代表程序执行成功。其它数值表明程序由于某些错误而中止运行。

为了解这是如何工作的,可以改进前面的 `COMLINE` 程序,使其具有当命令出错时能向操作系统返回一个出错码的功能。Turbo C 中的 `system()` 函数可用以达到这个目的。这个函数当成功时返回 0,否则为 1。

```
/* COMLINE: a program that executes whatever DOS
```

```

    commands are specified on the command line.
    Return error code to the operating system if
    an operation fails. */
#include <stdio.h>
#include <stdlib.h>
main(int argc, char *argv[])
{
    int i;
    for(i=1; i<argc; i++)
    {
        if(system(argv[i]))
        {
            printf("%s failed\n", argv[i]);
            return -1;          /* failure code */
        }
    }
    return 0;                  /* return success code */
}

```

通常把没有返回值的 main() 定义为 void, 如下面的语句:

```
void main(void)
```

因为如果函数不返回值又没有用 void 说明的话, 将会出现警告信息。

9.8 递 归

在 C 中, 函数可以实现自我调用。所谓函数递归就是在函数体中调用自己。递归有时被称为循环定义, 即用自己来定义自己的过程。

典型的递归例子是函数 factr(), 用以计算整数的阶乘。数 N 的阶乘是 1 到 N 之间所有整数的乘积。例如, 3 的阶乘是 $1 * 2 * 3$, 得 6。factr() 和它的等价循环程序如下所示:

```

factr(int n)    /* recursive */
{
    int answer;
    if(n==1)
        return(1);
    answer = factr(n-1) * n;
    return(answer);
}

fact(int n)     /* non-recursive */
{
    int t, answer;
    answer = 1;
    for(t=1; t<=n; t++)
        answer = answer * t;
}

```



```
return(answer);
```

```
}
```

未使用递归操作的程序 `factr()` 从 1 开始到指定数字进行循环, 逐次对每个数进行累乘, 即得到所乘运算结果。

实用递归操作的函数 `factr()`, 当用实参 1 调用 `factr()` 时, 函数返回 1; 否则返回值为 `factr(n-1) * n`。进一步计算该表达式的值时, 又以 `n-1` 作为参数调用 `factr()`, 直到 `n` 等于 1 并返回到调用它的函数。

现在计算 2 的阶乘, 第一次调用 `factr()` 导致了第二次调用, 此时参数为 1, 这次调用返回值为 1, 然后乘以 2 (`n` 的初值), 最后得 2。如果将 `printf()` 语句插入到 `factr()` 中, 会显示出最终结果和每一级调用的中间结果。

当函数调用自己时, 新的局部变量和参数就会在堆栈中分配内存单元, 同时函数代码以新的变量重新开始执行。递归调用不是重新复制该函数, 只是更换各变量的值。每次递归调用返回时, 前一级递归中的局部变量和参数会从堆栈中弹出来并且继续执行函数的前次调用。

大多数递归过程并不能明显地减小代码大小或变量所占的内存大小, 由于调用次数增加, 还可能降低程序的执行速度。但在大多数情况下, 这种变化并不明显。需注意递归调用函数可能会造成栈溢出, 因为变量存储在堆栈中, 每递归一次, 就会产生变量的新拷贝, 可用的堆栈空间减少。但并不是所有的情况都如此, 除非递归函数出现运行异常。

递归函数的主要优点是能够实现一些比同类型的循环更简明的算法。例如, 用循环算法实现起来相当困难的“快速排序”问题就可以借助于递归的方法解决。尤其是一些涉及到人工智能的问题, 通常按递归的方式思考问题比按迭代方式更有助于问题的解决。

当编写递归函数时必须要有条件判断语句以保证递归的终止, 否则当调用该函数时, 函数将会无休止地执行下去而不会返回, 这是写递归函数时经常发生的错误。在编制程序中可在一些地方使用 `printf()` 和 `getchar()` 函数, 这样在运行中就可以随时看到执行情况, 在发现问题时, 也可以及时终止程序运行。

下面是递归函数的另一个例子。函数名字叫 `siren()`, 用到了 Turbo C 中的 `sound()` 和 `nosound()` 函数发出的警报声。函数 `sound()` 用一整型参数调用并变成计算机扬声器中发出的连续的声音, 经过一段延时然后执行 `nosound()` 函数关闭声音。此函数无参数。当频率小于 100 时, `siren()` 函数进入递归调用并发出警报声。

```
/* This program sounds a siren. */
#include <dos.h>
siren(int freq)
{
    int i;
    if(freq > 100)
        siren(freq - 100);
    /* turn on sound */
    sound(freq);
    /* delay just a bit */
    for(i = 0; i < 10000; i++);
}
```

```
    nosound(); /* turn off sound */  
}
```

9.9 参数说明的传统形式和现代形式的比较

C 语言发明初期的函数参数说明形式,叫做传统形式。在本书中用到的最为广泛的参数说明方法则是现代形式。Turbo C 同时支持这两种形式(但是,ANSI C 标准重视现代形式)。目前仍有很多 C 程序用的是传统形式来说明函数,书上和杂志上所见到的程序也多采用这种形式,因此有必要了解一下传统形式和现代形式之间究竟有什么区别。

传统的函数参数说明中包括两部分:一个参数表,紧跟在函数名之后放在括号里;实际的参数类型说明,在闭括号和函数的花括号之间。传统形式的参数说明一般形式如下:

类型 函数名(参数 1,参数 2,... 参数 N)

类型 参数 1

类型 参数 2

⋮

类型 参数 N

{ 函数体 }

例如,现代形式的说明为:

```
float f(int a ,int b,char ch)
```

```
{
```

```
    ⋮
```

```
}
```

传统形式时将为:

```
float f(a,b,ch)
```

```
int a,b;
```

```
char ch;
```

```
{
```

```
    ...
```

```
}
```

对比它们,可知在传统形式中,类型名后可以列有多于一个的参数,而现代形式则需要对变量类型分别定义。由于一些十分微妙的原因,现代形式稍优于传统形式,因而本书用的是现代风格。对一个用传统形式写的程序,Turbo C 也能够很容易地对其编译——尽管传统函数说明的使用将会导致一些警告信息,但这些是可以忽略的。

9.10 对一些影响函数的效率和实用性问题的讨论

9.10.1 参数和通用函数

通用函数是适用于任何场合,或为许多个编程者所共同使用的函数。其典型的特征之一是:通用函数建立在参数基础上,函数需要的所有信息都由它的参数来传递。这样既可保证

函数的通用性,还保证了参数在程序代码中的可读性,因此部分地避免了使用全局变量带来的副作用。

9.10.2 效 率

函数是构造 C 语言的模块,除了个别小程序以外几乎所有程序都包含大量函数。但是在某些特定应用中也存在需要放弃使用函数,而用内部代码代替的情形。这时内部代码与函数的作用相同,只是不进行函数调用,只有在对运行速度要求很严格的情况下,内部代码才用于代替函数调用。

内部代码比函数调用快,有两个方面:首先,执行一次调用要花费一定的时间,其次,通过堆栈传递参数存取时也要耗费时间。几乎对于所有的应用程序来说,这点执行时间的增加是无足轻重的。但是,用内部代码来代替程序中的函数调用,当出现多次重复调用时就会加快速度。例如,下面的两条程序都打印 1 到 10 的平方值。内部代码形式的程序运行速度比函数调用快,因为后者要额外花费时间。

In-Line	Function Call
<pre>#include <stdio.h></pre>	<pre>#include <stdio.h></pre>
<pre>main(void)</pre>	<pre>int sqr(int a);</pre>
<pre>{</pre>	<pre>main(void)</pre>
<pre> int x;</pre>	<pre>{</pre>
<pre> for(x=1;x<11;++x)</pre>	<pre> int x;</pre>
<pre> printf("%d",x*x);</pre>	<pre> for(x=1;x<11;++x)</pre>
<pre> return 0;</pre>	<pre> printf("%d",sqr(x));</pre>
<pre>}</pre>	<pre> return 0;</pre>
	<pre>}</pre>
	<pre>sqr(int a)</pre>
	<pre>{</pre>
	<pre> return a*a;</pre>
	<pre>}</pre>

编程序时,在提高程序的可读性、可修改性和可移植性的同时,一定要注意节省函数的执行时间,即提高运行速度。

9.11 库函数

Turbo C 函数库通过提供好几百个易于使用的函数来提高编程效率。设计的这些函数用来处理基本的编程设计,和给计算机设备提供方便接口。库函数解除了编制处理低级编程和接口任务函数的繁重工作。

例如,图形库的存在就说明了函数库的方便之处。Turbo C 有一个很大的图形函数库,即 BGI 函数。BGI 代表 Borland Graphics Interface。BGI 系列函数做一些控制视频硬件的繁琐工作,并且创建大多数用户需要的图像。当使用这些函数时,可以专心于编程项目,而不必

顾虑基本的视屏编程。

哪些库函数可以利用

正如前面提到过, Turbo C 库中有几百个函数。下面是一些主要的函数组和它们的一般目的:

- **标准函数(standard function)** 标准函数是一组基本的、C 语言编程用的实用函数。
- **I/O 函数(I/O function)** I/O 函数处理将数据输进和输出程序的任务。Turbo C 有两类不同的 I/O 函数。一类函数使用 C 语言和流来处理数据。另一类函数使用 DOS 的 I/O 实用程序来处理数据。
- **分类函数(Classification functions)** 即 is...() 函数类, 用来将字符分类, 字符可以分成不同的类别, 如大写体、小写体和十六进制。
- **转换函数(Conversion functions)** 转换函数将数据从一种格式转化为另一种格式。
- **目录控制函数(Directory-control function)** 这些函数允许使用 DOS 目录和路径。
- **诊断函数(Diagnostic function)** 有了诊断函数, 就可以对程序进行故障检测。
- **图形函数(Graphics function)** 图形函数控制机器的视屏硬件, 并且生成许多基本的图形画面。
- **接口函数(interface function)** 有了接口函数, 用户就可以直接使用 DOS 和 BIOS 功能(函数)。
- **处理函数(manipulation function)** 处理函数、字符串和内存块。
- **数学函数(math function)** 这些函数能处理用户需要执行的、几乎所有的数学计算。
- **存储函数(Memory function)** 存储函数为小型和大型数据内存模式进行动态存储分配。
- **杂类函数(miscellaneous function)** 这些函数处理大量的编程任务。
- **进程控制函数(process-control function)** 这些函数处理进程的创建和中止, 子进程是指在其它进程里产生的进程。
- **标准函数(standard function)** 这些是 stdlib.h 头文件中一般性能的函数。
- **文本窗口函数(text-window function)** 文本窗口函数用于向视屏显示发送文本信息。
- **时间和日期函数(time and date function)** 这些函数处理时间和日期信息。
- **可变参数表函数(variable argument list function)** 这些函数处理传给函数的可变长度的参数。

可以用两种方法来寻找库函数的特定信息。第一种方法是使用 Turbo C 联机帮助工具。这个工具有一个完整的库函数列表。列表中每一个函数包含: 该函数简短的描述、函数说明的信息、代码示范和与相关函数的交叉参考。

在程序中加入库函数

明白哪种库函数可以用之后,还需要学习如何在程序中使用它们。库函数使用简单。一旦找到要用的函数,就把相应的头文件放进源代码中,确保将任何需要的参数传给该函数。

当在程序中使用库函数时,必须把与该函数有关的头文件全部插进去。库函数的头文件包含程序中使用该函数所需的变量、宏和函数的说明。将头文件嵌入到源文件中的语句如下:

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

include 是一个预处理器指令,它告诉编译程序在编译期间要把规定的文件包含进源文件中去。要包含进的文件并不完全拷贝到源文件里,而是在编译源文件期间对它进行引用。包围着文件名的中括号告诉编译程序要在标准库目录中录找那个文件。在编译期间,该文件与源文件合并在一起。在上面这个例子中,要包含进去的文件是 `stdio.h` 头文件。

程序 `text_demo.c` 展示了如何在程序中使用一些库函数。

`text_demo.c` 使用文本窗口函数的程序

```
1  /* TXT_DEMO This program uses the text-window functions
2
3      to demonstrate the use of the library
4
5      functions. */
6
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <conio.h>
10
11 void main( void )
12 {
13     int i, j;
14
15     clrscr();
16     textmode( C80 );
17
18     while( ! kbhit() ) {
19         for( i = 0; i <= 15; i++ ) {
20             textcolor( i );
21             for( j = 48; j <= 90; j++ ) printf( "%c", j );
22         }
```

`text_demo.c` 中的程序使用了文本窗口函数,产生多颜色的、滚动的 ASCII 显示。整个程序由三个循环组成。最外层的循环等待按下一个键来终止测试,第二层循环(i 循环)通过 16 种文本模式前景色,最内层的循环打印输出从 0 到 Z 的 ASCII 字符。

这个程序最主要的特征不是它的循环结构,而是它的库函数。在第 13 行,函数 `clrscr()` 清屏,可以看到 `clrscr()` 并没有传递任何参数。原因是说明 `clrscr()` 时有一个 `void` 类型参数表。在第 14 行,调用 `textmode()` 函数。这个函数设置视屏模式——此时,设置为 80 列的彩

色显示。

在循环里调用 `textcolor()` 函数, 传给 `textcolor()` 的参数是循环索引变量 `i`, 在函数 `textcolor()` 中使用变量 `i` 是为了显示 16 种文本模式颜色; 在循环里还有 `cprintf()` 函数, `cprintf()` 有一个格式串和可变参数表, 在范例中的参数使函数 `cprintf()` 打印输出变量 `j` 的 ASCII 值。

要使用这些特殊的文本窗口函数, 必须在源文件中加入正确的头文件。可以看到, 在第 7 行, 头文件 `conio.h` 包括在源文件中。如果没有头文件, 程序就不能正确编译。必须经常把程序使用的库函数的头文件包括到源文件中来。

9.12 改变程序的执行流程

当想改变程序中的执行流程(流向)时, 通常使用重复(叠加)语句或条件语句(`if` 和 `switch`)。但这些工具并非是改变程序执行流程的唯一工具, 用户也可以使用许多函数。

改变程序的执行流程的函数可以分成两组, 第一组中的函数处理程序的早期中止, 这组函数包括 `exit()` 和 `abort()` 函数; 第二组函数在执行另外程序时将原来的程序挂起, 这一组包括 `spawn...` 类函数、`exec...` 类函数和 `system()` 函数。

9.12.1 使用 `exit()` 和 `abort()` 函数

使用 `exit()` 和 `abort()` 函数可以中途终止程序的运行, `exit()` 函数正常终止程序, `abort()` 函数异常中止程序。

`exit()` 函数引起写打开的缓冲区, 关闭打开的文件, 执行任何其它的清除工作。尽管 `exit()` 函数正常终止程序, 但一般不用 `exit()`, 除非遇到严重错误, 在那种情况下, `exit()` 函数以尽可能早的方式关闭(shut down)程序, 该函数有下列格式:

```
exit(status);
```

在 `exit()` 函数调用中, 参数 `status` 表明程序是正常终止或是异常终止。如果程序要正常终止, 则 `status` 参数为 `EXIT_SUCCESS`, 它是一个宏, 值为 0。如果程序异常终止, 则 `status` 参数为 `EXIT_FAILURE`, 一个有非零值的宏。`status` 参数传给调用该程序的进程, 让调用进程知道程序终(中)止时的情况。

可以使用 `atexit()` 函数与 `exit()` 函数来提供一系列函数(多达 32 个)。这些函数在程序终止时调用。当调用 `exit()` 函数时, 任何 `atexit()` 函数登记过的函数在该程序终止前可以执行。被 `atexit()` 登记过的函数按后进先出原则调用。`term.c` 说明了 `exit()` 和 `atexit()` 函数。

`term.c` 使用 `exit()` 和 `atexit()` 函数来终止执行的程序。

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void term_1( void );
5  void term_2( void );
6
7  main()
8  {
9      atexit( term_2 );
```

```
10     atexit( term_1 );
11
12     printf( "In main, processing normally. \n" );
13     printf( "Beginning termination. \n" );
14     exit( EXIT_SUCCESS );
15     printf( "This statement is never executed. \n" );
16 }
17
18 void term_1( void )
19 {
20     printf( "Program termination eminent. \n" );
21     printf( "In function term_1. \n" );
22 }
23
24 void term_2( void )
25 {
26     printf( "Program termination proceeding. \n" );
27     printf( "In function term_2. \n" );
28 }
```

在 term.c 的第 9 和第 10 行, atexit() 函数登记了两个函数。当 exit() 函数被调用时, 被登记过的函数可以执行。注意函数登记时的顺序, 最后一个执行的函数是首先被登记的函数, 要执行的第一个函数是最后被登记的函数。

在第 14 行, 调用函数 exit()。在执行登记过的函数之后, 该函数暂时中止程序的运行。在程序结束之前, 调用登记过的函数 term_1() 和 term_2()。另外, exit() 函数关闭所有打开的文件和写所有打开的输出缓冲区。这个程序没有要关闭的文件或缓冲区。注意到 exit() 后面的语句从未执行过。

象 exit() 一样, abort() 函数引起程序终止。然而 abort() 在终止程序之前不作任何清除工作。abort() 只是简单地引起程序崩溃。当发生灾难性错误时, 这个函数经常使用, 用户可以作的唯一事情就是销毁程序。abort() 比较紊乱, 但能有效地终止程序。

下面的代码片段显示了 abort() 的用法:

```
printf("catastrophic errors have occurred. \n");
printf("Terminating program. \n");
abort();
```

正如你所见, 调用 abort() 函数比较简单。当执行 abort() 时, 它会将退出代码 3 返回给父进程或 DOS。

9.12.2 使用 system(), exec...(), 和 spawn() 函数

abort() 和 exit() 函数是 Turbo C 提供的, 用于处理异常情况的两个有用的函数。Turbo C 另有一组函数, 可以用来改变程序的执行。虽然这一组里的函数没有 abort() 和 exit() 严格, 但可以使用该组中的函数来运行其它程序。在程序中运行其它程序是 Turbo C 灵活的特性, 不需要用户自己重新编写程序就可以使用其它实用程序。

这一节要介绍 `system()`、`exec...()` 和 `spawn...()` 函数。`system()` 函数让用户在程序中能运行程序或 DOS 命令,使用 `exec...()` 和 `spawn...()` 函数比 `system()` 函数更灵活。

`system()` 函数启动 COMMAND.COM, 执行常规的 DOS 命令。这个函数有下面的形式:

```
system(const char * command)
```

`system()` 参数表中的 `command` (命令) 可以是任何正确的 DOS 命令, 或可执行的文件名, 或批处理文件名。在下面的一行代码中, `system()` 函数调用 DOS 的 CHKDSK.COM 程序:

```
system("CHKDSK");
```

可以使用 `exec...()` 和 `spawn...()` 系列函数来运行其它程序。由 `exec...()` 和 `spawn...()` 运行的程序称为子进程。

`exec...()` 与 `spawn...()` 之间的区别在于: `spawn...()` 不接收另外一个变元 `mode` 参数。这个参数规定在调用子进程时 `spawn()` 如何操作。`mode` 参数可选用如下 3 个值:

- P_WAIT

P_WAIT 参数规定: 在子进程执行时, 程序会暂时中断(停止)。

- P_OVERLAY

P_OVERLAY 参数规定: 子进程将覆盖调用进程的内存地址。使用 P_OVERLAY 项与 `spawn` 一起将做 `exec...()` 一样的工作。

- P_NOWAIT: 这个参数暂时不用, 如果用了 P_NOWAIT, 则当子进程运行时, 原来的进程也会继续运行。

除了 `mode` 之外, `spawn...()` 与 `exec...()` 函数的参数相同。两类函数都有 `path` (路径) 参数和参数表, 变元表将传送给子进程。`path` 参数是子进程的位置和名字。当搜寻 `path` 参数规定的子进程进时, `spawn...()` 和 `exec...()` 都使用标准的 DOS 过程。下面是一个典型的 `spawn()` 函数调用。

```
spawn(P_WAIT, "my_prog", NULL);
```

`spawn()` 函数执行程序 `my_prog`。`spawn...()`, 首先搜寻 MY_PROG.COM; 如果该程序(文件)不存在, 然后, `spawn...()` 又搜寻 MY_PRGO.EXE, 调用进程(父进程)在子进程 `my_prog` 执行期间挂起。在上面这个例子中, 没有参数传给 `my_prog`。

`exec...()` 和 `spawn...()` 函数更多的情况可见后面的章节。

9.13 使用可变参数表

编写的许多函数会需要固定数目的参数。由于函数通常使用同样数目和类型的参数, 所以将参数传给函数比较简单。然而, 并非所有的函数都是这样简单, 例如考虑函数 `printf()`, 每次调用 `printf()` 时, 可能有不同类型和数目的参数。`printf()` 函数有一个可变参数表。

在这一节中, 要学习如何在函数中使用可变参数表。可变的参数表能增强用户编写的程序的实用性和简洁性。由于函数能处理较多的数据以及不同类型的数据, 所以函数的实用性得到增强。由于有了可变参数表, 可以只编写一个函数而不是几个函数, 所以函数会更加简洁。

9.13.1 设计可变参数表

虽然可变参数表比标准的参数表要复杂一些,但可变参数表仍然容易使用。由于可变参数表必须确定有多少参数和何种类型的参数要传给该函数,所以它需要较多的计划。

要访问可变参数表中的参数,用户得使用一个指针,该指针指向表中的参数。从可变参数表里取参数的第一步是初始化指针,将指针指向表的第一个参数。一旦参数指针初始化完毕,表中的每一项就可以通过参数指针在参数表中“上下移动”来读取。

在设计一个可变参数表时,为了使参数指针能正确地指向表中的参数,必须切记两个重点:

- 必须规定可变参数表的长度。
- 必须规定可变参数表中包含哪些数据类型。

提供这样特定的信息,乍一看起来似乎建可变参数表的思想相矛盾。然而,可以使用三种方法来提供这样的信息,而且还可以依然使用可变参数表。

第一种方法,也是最常用的方法:使用格式串(format string)来表示多少数据和数据类型。格式串使用一套符号来表示可变参数中要求的参数类型和个数。注意看 `printf()` 函数中格式串的使用:

```
printf("i=%d,x=%f",i,x);
```

`printf()` 打印两个不同类型的变量,由于有格式串,所以该函数知道要打印哪些信息。这个格式串就是: `"i=%d,x=%f"`。前面带有符号%的两个字符称为转换规定符。这些字符让 `printf()` 函数知道可变参数表中要多少变量和什么类型的变量。

第二种跟踪可变参数表中的变量的方法是加入结束值。这个值不用在函数中,但放在可变参数表的末尾,表明所有的数据已读完。看下面这一行代码:

```
sum(num_type,2,39,56,0);
```

值 0(零)就是结束值。因为这个例子包含 `sum()` 函数,故不需要 0 值。因此,当在可变参数表里遇到 0 时,用户应知道已读完所有用户需要的数,0 值告诉用户停止取数给 `sum` 函数。

使用结束值的缺点是所有的数必须有一个固定类型。如果用户使用结束值,那么不能把数据类型弄混,计算机必须知道需要什么数据类型,以便能从内存中读正确的字节数(记住:每一种数据类型在存储时使用不同的字节个数)。

第三种跟踪可变参数表里的参数的方法是加入一个变量,该变量告诉函数要读多少数据项。这种方法与第二种方法有同样的缺点,可变参数表中所有的项必须是同一类型。

9.13.2 使用 `va...` 函数

在前面的一节里,已知道什么是可变参数表,也看到如何使用指针在表中移动和指向下一个要取回的数据项。这一节给出一些工具,可以与可变参数表一起使用。为处理可变参数表,Turbo C 提供了三个宏和一个数据类型。这些工具属于 `va...` 函数类。

定义的数据是 `va_list`。它用来说明一个指向可变参数表中数据项的指针。`va...` 系列中的宏有下面的语法:

```
Void va_start(va_list ap,last fix);
```

```
type va_arg (va_list ap, type);
```

```
void va_end (va_list ap);
```

va_start 设置一个表指针,它指向表中的第一个变量参数。参数 ap 告诉 va_start(),用哪一个指针来访问最后一个固定的参数。va_start()宏使 ap 只指向 lastfix 规定的参数以外的第一个参数。这个宏必须在 va_arg()或 va_end()使用之前执行。

va_arg()有双重目的:第一 va_arg()返回参数指针 ap 指向的对象的值,第二,va_arg()修改参数指针,把它指向表里的下一个数据项。va_arg()参数表中规定的 type(类型)告诉 var_arg()正在读的是什么类型的数据。参数 type 告诉 va_arg()指向的数据是整型数, double 类型的数还是其它类型的数值, type 参数也给 va_arg()提供了正确修改参数指针必需的信息。

va_end()进行一些清理、组织工作,这是被调用的函数能正确返回所必需的工作。如果读完所有的可变参数之后,没有调用 va_end(),那么程序可能会出现一些奇怪的、未定义过的错误。

下面两份程序 V_List1.c 和 V_List2.c 给用户展示了如何使用 va...()系列函数。第一个程序包含一个可变参数表,当调用那个函数时,参数表中有规定的参数个数。第二个程序是第一个程序的变化形式,它使用了一个特殊的值来表示参数表的结束。

v_list1.c 使用可变参数表的程序。

```
1  /* V_LIST1.C This program uses a variable argument list
2      that specifies the number of arguments in
3      the list when the function is called. */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <stdarg.h>
8  #include <conio.h>
9
10 int sum( int no_args, ... );
11
12 int main( )
13 {
14     int no_args;
15     int result;
16
17     clrscr();
18     printf( "This program calculates the sum of a \n" );
19     printf( "group of numbers. You will supply the \n" );
20     printf( "sum function with the number of items to \n" );
21     printf( "sum and a list of integer values. \n\n" );
22
23     no_args = 5;
24     result = sum(no_args, 3, 5, 18, 57, 66 );
```

```

25     printf( "The value of result = %d\n", result );
26
27     return 0;
28 }
29
30 int sum( int no_args, ... )
31 {
32     va_list ap;
33     int result = 0;
34     int i;
35
36     va_start( ap, no_args );
37
38     for( i = 1; i <= no_args; i++ ) {
39         result += va_arg( ap, int );
40     }
41
42     return result;
43 }

```

v_list1.c 说明 *sum()* 函数使用一个可变参数表, *sum* 函数使用一个变量来存储要传给该函数的参数的个数。

正如已见到, 可变参数表需要一种检测参数表是否结束的方法。在 *v_list1.c* 中, 使用另外一个变量存储一个值, 该值与要传给函数的参数个数相等。这个附加的变量使得不会越过参数表的末尾就能读到所有的参数。

第 10 行包含函数 *sum()* 的函数原型, 这个函数返回一个整数值, 它等于传给该函数的所有参数的和。*sum()* 参数表中有两个说明, 第一个参数 *no_args*, 表示可变参数表中数据项的个数。每一次调用这个函数时, 都要提供 *no_args* 变量的值。第二个参数是省略号(...), 它表示有可变个参数数目要传给该函数。

调用函数 *sum()* 的例子在第 24 行。第 23 行可看到变量 *no_args* 赋了一个值, 该值与第 24 行传给 *sum()* 的参数个数相等。

第 30 至 43 行定义了函数 *sum()*。第 32 行说明了一个 *va_list* 数据类型的参数指针。参数指针指向可变参数表中的参数。第 36 行, 宏 *va_start()* 把参数指针 *ap* 指向第一个可变参数的地址。

v_list1.c 中第 38 至 40 行, *for* 循环逐步经过可变参数表中的数据项。宏 *va_arg()* 返回可变参数表里数据项的值, 并把参数指针增加, 指向表里的下一项。变量 *result* 将宏 *va_arg()* 返回的值累加。当 *for* 循环结束时, 变量 *result* 的值返回到调用函数。

v_list2.c 中程序则不需要变量来表示传给函数的参数个数, 而是函数检查可变参数表中的结束值。当遇到一个结束值时, 函数停止处理参数。

v_list2.c 带有可变参数表的另一个函数。

```

1  /* V_LIST2.C This program uses a variable argument list
2      that contains a terminating value. */

```

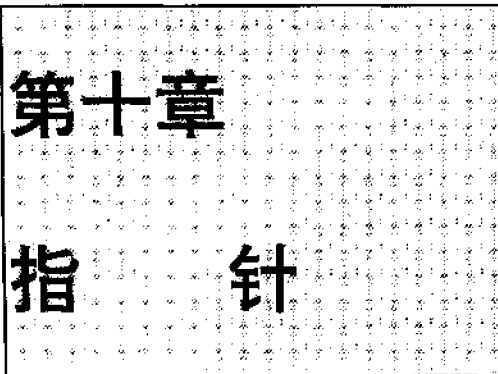
```
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <stdarg.h>
7  #include <conio.h>
8
9  int sum( int first_num, ... );
10
11 int main( void )
12 {
13     int result;
14
15     clrscr();
16     printf( "This program calculates the sum of a\n" );
17     printf( "group of numbers. You will supply the\n" );
18     printf( "sum function with a list of values. \n" );
19     printf( "You must have at least one value, and the\n" );
20     printf( "Last value must be zero. \n\n" );
21
22     result = sum( 43, 56, 1, 19, 3, 0 );
23     printf( "The value of result = %d\n", result );
24
25     return 0;
26 }
27
28 int sum( int first_num, ... )
29 {
30     va_list ap;
31     int result;
32     int number;
33
34     va_start( ap, first_num );
35
36     result = first_num;
37
38     while( ( number = va_arg( ap, int ) ) != 0 ) {
39         result += number;
40     }
41
42     return result;
43 }
```

v_list2.c 中的程序和 v_list1.c 中的程序类似,然而这个程序(v_list2.c 中的程序)不需要传给函数 sum() 的值,该值表明可变参数表中会有多少参数。

注意看 `v_list2.c` 中第 9 行的函数说明, 每一个带有可变参数表的函数都至少需要一个固定的参数。在 `sum()` 中, 这个固定的参数存储要累加的表项中的第一项。在第 36 行, 可以看到, 固定的参数分开处理。

在 `v_list2.c` 中, 从可变参数表里读取数值的循环与 `v_list2.c` 中的循环不同。循环通过可变参数表时, `v_list1.c` 中的 `for` 循环使用了变量, 它表明表中项的个数。而 `v_list2.c` 中, 却利用 `while` 循环可变参数表中取到值, 直到遇到结束值才停止。

在第 38 行, 宏 `va_arg()` 从可变参数表中读取值, 该值又赋给变量 `number`。然后检查赋给 `number` 的值, 确定是否读到了结束值。当读到结束值时, `while` 循环结束。



编写成功的 C 程序的关键在于深入理解和正确使用指针。这是因为：首先，指针提供了函数地址调用的方法；其次，指针可用于支持 C 的动态内存分配；第三，指针可在任何地方代替数组，这样就大大地提高了程序效率。另一方面，C 的很多特性完全依赖于指针才能实现，因此对指针的透彻理解非常重要。

指针是 C 最强有力的手段，但它也是最具有危险性的。例如，未初始化的指针可能导致系统崩溃；另外，使用指针很容易产生错误，而且这种错误很难被发现。

正是由于指针有以上这些利弊，本章将对它作详细的讨论。首先阐述指针的语法，然后结合具体的程序例子详细讨论指针的各种用法。

10.1 指针的语法规则

10.1.1 什么是指针

指针可分为两大类：指向对象的指针和指向函数的指针。两类指针都是用来保存存储地址的特殊对象。

虽然两类指针可以共用某些 Turbo C 的操作，但它们具有不同的特性、目的和操纵规则。通常来说，函数指针用于存取函数以及把函数作为参数传递给其它函数，但不允许在函数指针上进行算术运算。而对象指针在扫描数组或更复杂的数据结构时可以进行增量或减量。

尽管指针也是数值，并且具有无符号整数的大多数特征，但它们关于赋值、转换和算术运算有着自己的规则与限制。

指向对象的指针

“指向 type 对象的指针”保存 type 对象的地址（即指向 type 对象）。由于指针也是对象，故而可说明指针的指针，通常可以用指针指向的目标对象包括数组、结构、联合和类。

对象指针的大小依赖于存储模式以及数据段的大小和配置，还可能受到可选的指针修饰的影响。

指向函数的指针

函数指针可以理解为代码段内的一个地址，该函数的可执行代码从该地址开始存放。换句话说，该地址调用该函数时的转移地址、代码的大小与配置由存储模式强制决定，存储模式还制约着调用函数所需函数指针的大小。

函数指针的类型为“指向返回 type 值的函数指针”，这里 type 为函数的返回类型。

C 具有较强的类型检查，函数指针的类型为“带有 type 类型参数并返回 type 值的函数指针”。事实上，定义为有参数类型的 C 函数也具有这种意义较确切的类型，例如：

`void (*func)();` 则 *func 为带 int 参数且不返回任何值的函数指针。

10.1.2 指针说明

指针必须被说明成指向某一特定的类型，即使在该类型为 void 时也是如此（它实际上表示指向任一类型的指针）。但指针被说明后，通常可以指向另一类型的对象。Turbo C 允许不用类型强制转换而对指针重新赋值，但编译时将对此发出警告信息。C 允许把一个 void * 型的指针赋给一个非 void * 型的指针。

若 type 为任一预定义的或用户定义的数据类型，包括 void，则

`type * ptr; /* 危险——未初始化的指针 */`

说明 ptr 为“type 指针”类型。所有的作用域、生存期和可见性规则都适用于刚声明的 ptr 对象。

空指针值要保证具有不同于程序中使用的有效指针的地址。把整型常量赋给指针相当于赋一空指针值给它。

使用 NULL（定义于标准库头文件，如 `stdio.h`）可增加可读性，所有指针可正确地与 NULL 作相等或不相等测试。

不可把“void 指针”与空指针相混淆。

`void * vptr;`

说明 vptr 为一个 void 类型的指针，它可被赋予任何“type 型指针”，包括空指针的值。这时编译器不会发出警告，给定 type1 与 type2 为两个不同类型，若不经适当的类型强制转换而进行“type1 指针”与“type2 指针”间的赋值，则会引起编译器发出警告或出错。若 type1 为一函数，type2 不是（反之亦然），指针赋值将是非法的。若 type1 为 void 指针，则不需要进行任何类型强制转换。若 type2 为 void 指针，则在 C 下不需任何类型强制转换。

赋值限制也适用于不同大小（near、far 和 huge）的指针。可以把一个较小的指针赋给一个较大的指针，这时不会出错，但是，只有使用了显式的强制转换，才能把一个较大的指针赋给一个较小的指针，例如

```
char near * ncp;
char far * fcp;
char huge * hcp;
fcp = ncp          /* 合法 */
hcp = fcp          /* 合法 */
fcp = hcp          /* 非法 */
ncp = fcp          /* 非法 */
ncp = (char near *)fcp; /* 合法 */
```

10.1.3 指针与常量

说明或指向对象的说明可带有 const 修饰符，说明为 const 的任何对象都不能再被赋值，创建一个可能违反常量对象规则的非赋值的指针也是非法的。举例：

```

int i;                /* i 是一整型 */
int *pi;              /* pi 为 int 型指针(未初始化) */
int *const cp = &i;    /* cp 为指向 int 的常量指针 */
const int ci = 7;      /* ci 为一常数 */
const int *pci;        /* pci 为指向常整型的指针 */
const int *const cpc = &ci; /* cpc 为指向常整型的常量指针 */

```

于是以下赋值是合法的:

```

i = ci;               /* 将常整数赋给整型 */
*cp = ci;             /* 不可修改常整型 */
++pci;               /* 指向常量的指针递增 */
pci = cp;             /* 把指向常量的常指针赋给指向常量的指针 */

```

而以下赋值是非法的:

```

ci = 0;               /* 不可对常整型赋值 */
ci--;                /* 不可修改常整型 */
*pci = 3;             /* 不可对常量指针指向的对象赋值。 */
cp = &ci;             /* 不可对指针常量赋值,即使值不变也不行 */
cpc++;               /* 不可修改指针常量 */
pc = pci;             /* 对 *pci(常量值)赋值 */

```

类似的规则可适用于 volatile 修饰符,注意 const 和 volatile 可同时作为同一标识符的修饰符。

10.1.4 指针算术运算

指针算术运算仅限于加、减和比较。类型为“指向 type 的指针”的对象指针上的算术运算自动考虑了 type 的大小,即存储一个 type 对象所需的字节数。

当进行指针算术运算时,都假定指针指向一个对象数组,因而,若指针被说明为指向 type 的,则在该指针上加上一个整数值就相当于该指针前移相应个数的 type 对象。若 type 大小为 10 字节,则指针加整数 5 相对于该指针在内存里前移 50 个字节。仅当两指针指向同一数组时,其差才有意义。差值为两指针值之间的数组元素数。例如,若 ptr1 指向一数组的第 3 个元素,ptr2 指向第 10 个元素,则 ptr2 - ptr1 的结果为 7。

“指向 type 的指针”与一整型值相加或相减时,其结果还是“指向 type 的指针”。若 type 为一非数组对象,指针操作数被当作指向元素长度为 sizeof(type)的“type 数组”第一个元素的指针来处理。

当然,不存在“指向超过最后一个元素的指针”这种元素,但可允许一指针假定为这样一个值。若 P 指向数组的最后一个元素,则 P+1 有效,但 P+2 无定义。若 P 指向超过最后一个元素的指针,则 P-1 有效,为指向最后元素的指针。然而,若把间接引用操作符 * 应用到“指向超过最后元素一个元素的指针”,则会引起无定义错误。

从形式上说,只要指针处在合法范围(超过最后元素的第一个元素)内,则 P+n 可认为是指针前移(n * sizeof(type))个字节。

指向同一数组对象的两个指针相减将得到一个 ptrdiff_t 类型的整型值,该类型在 stddef.h 中有定义(对巨型指针和长型指针为 signed long,对其它指针为 signed int)。该值若在 ptrdiff_t 范围里,则代表两引用元素的下标之差。在表达式 P1-P2 中,这里 P1、P2 为

type 型指针(或受限 type 型指针),且 P1、P2 必须指向实际存在的元素或超过最后元素一个的元素,若 P1 指向第 i 个元素,P2 指向第 j 个元素,则 $P1 - P2$ 的值为 $(i - j)$ 。

10.1.5 指针转换

使用类型强制换机制,可以将某指针类型转换为其它指针类型:

```
char * str;  
int * ip;  
str = (char *)ip
```

更一般地说,强制转换(`type *`)可将一指针转换为“指向 type 的指针”。

10.2 指针是地址

一个指针是存储另一个对象内存地址的变量。通常情况下,这个地址是另一个变量在内存中的地址,尽管它也可以是端口或特定的 RAM 地址,如视频缓冲区等。如果某一个变量包含了另一个变量的地址,那么可以说是第一个指针指向第二个指针。这种情况如图 10.1 所示。

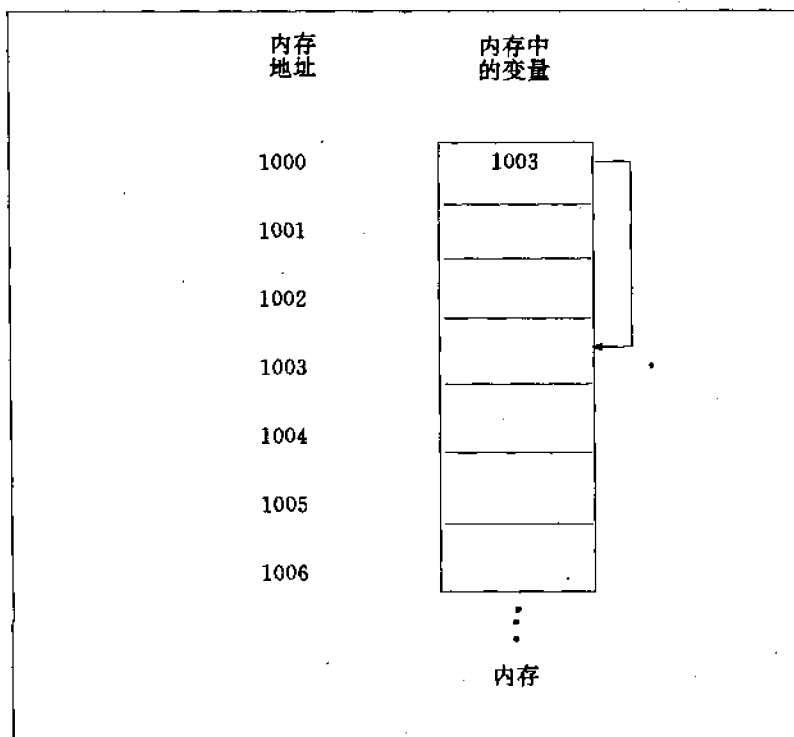


图 10.1 指向另一个变量的变量

10.3 指针变量

指针说明由一个基本类型、一个 * 号以及变量名组成,它的一般形式为:

```
type * var _name;
```

type 可以是任何有效的 C 类型, var_name 是一个指针变量名。指针基本类型定义了指针可以指向的变量的类型。例如, 下面这些语句说明了一个字符型指针和两个整型指针。

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

10.4 指针操作符

指针操作符有两个:& 和 *。& 是一个单目操作符, 返回运算对象的内存地址。例如

```
count_addr=&count;
```

将变量 count 的内存地址存放在 count_addr 中。这个地址是变量的计算机内部地址, 它与变量 count 的值是不同的。由于 & 操作符返回变量的地址, 所以上面的赋值语句的意思是“将 count 的地址赋给 count_addr”。

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

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

```
val=*count_addr;
```

将把 count 的值赋给 val。如果 count 的初始值为 100, 则 val 的值为 100, 因为 100 被存放在分配给 count_value 的内存地址 2000 处。* 也称为“间接引用操作符”, 上条赋值语句即是“将 count_addr 处的值赋给 val”。指针操作符 * 与乘法符号相同, 读者要注意区别。

指针操作符 & 和 * 之间并没有联系, 它们的优先级与单目减相同, 但高于其它所有算术操作符。

以下是一个使用 * 和 & 的程序, 它在屏幕上显示 100。

```
#include <stdio.h>  
void main()  
{  
    int *count_addr, count, val;  
    count=100;  
    count_addr=&count;           /* 取 count 和地址 */  
    val=*count_addr;             /* 取此地址中的值 */  
    printf("%d val",);          /* 显示 100 */  
}
```

在前面的说明中, 可以通过指向 count 的指针间接地把 count 的值赋给 val。但是编译程序从 count_addr 所指地址起送多少字节给 val 呢? 对于使用了指针的赋值语句, 编译程序如何传送适当的字节数取决于指针的基本类型。如在上面的例子中, 因为 count 是 int 型指针, 所以要从 count_addr 所指向的地址传送两个字节给 val。如果 count_addr 是 double 型指针, 则要传送 8 个字节。

必须保证指针变量总是指向正确的数据类型。如当说明指针是 int 型时, 编译程序就假定指针保存的任何地址均指向 int 型变量。由于 C 语言允许把任意一个地址赋给指针变量,

所以以下有错误的代码仍能通过编译,但 Turbo C 将显示警告信息:

```
#include <stdio.h>
/* This program will not work correctly. */
void main()
{
    float x=10.1, y;
    int *p;
    p=&x;
    x= *p;
    printf ( "%f",y);
}
```

10.5 指针表达式

通常情况下,包含指针的表达式同 C 其它表达式遵循同样的规则。在这一节中,我们将研究指针表达式的一些特殊情况。

10.5.1 指针赋值

指针赋值同任何变量一样,是把一个指针放在赋值语句的右边,将其值赋给另一个变量。例如:

```
#include <stdio.h>
void main()
{
    int x;
    int *p1, *p2;
    x = 101;
    p1 = &x;
    p2 = p1;
    /* print the hexadecimal value of the
    &address of x -- not x's value */
    printf("at location %p", p2);
    /* now print x's value */
    printf("is the value %d\n", *p2);
}
```

x 的地址以十六进制显示,所使用的 printf() 的格式符号 %p 指明指针地址将以十六进制形式显示。

10.5.2 指针运算

指针操作符有两个:递加和递减。为弄清指针运算的内容,假设 p1 为指向一个整数的指针,当前值为 2000。经过下面的表达式

```
p1++;
```

计算之后, `p1` 的值为 2002, 而不是 2001。每次 `p1` 都增值, 指向下一个整数。同样, 递减也是这样。例如:

```
p1--;
```

将使 `p1` 的值变为 1998, 假定其原值为 2000。

每当指针增值时, 它将指向其基本类型的下一个元素的内存地址, 每当指针递减时, 将指向上一个元素的地址。这样, 指针为字符型时, 进行的是“真正”的加 1 或减 1 运算。其它类型的指针均将增加或减少所指数据的单位长度。例如, 假定指针为字符类型的, 当该指针递增时, 其值加 1。但当指针为整数类型时, 其递增将加 2。图 10.2 说明了这个概念。

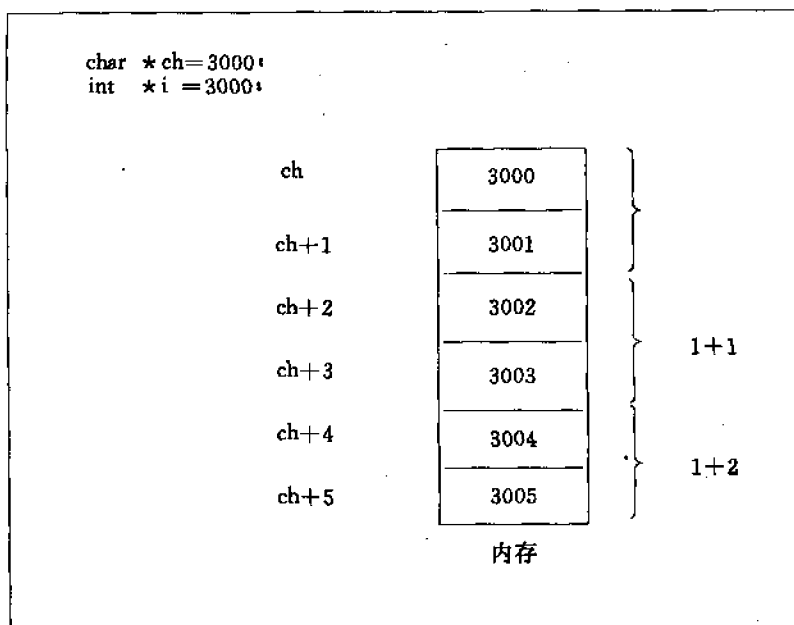


图 10.2 指针运算与基类型的关系

递增和递减可以不受局限地进行, 指针还可以加上或减去一个整数。表达式 `p1=p1+9;`

将使 `p1` 指向它当前所指基本类型后的第九个元素。

也可以从一个指针中减去另一个指针。如果两者都指向同一数组中的不同元素, 那么相减的结果与所指元素的下标相减的结果相同。

除了指针和整数相加减、指针和指针相减以外, 不允许对指针进行其它的算术操作。特别要注意的是, 不允许:

- 指针间相乘或相除
- 两个指针相加
- 对指针使用位变换和屏蔽操作符
- `float` 和 `double` 类型的指针相加减

10.5.3 指针比较

在关系表达式中允许对两个指针进行比较。例如,已知两个指针 p 和 q ,下面的语句是完全正确的:

```
if (p < q)
    printf("p points to lower memory than q\n");
```

通常,指针比较仅用于指向共同对象的两个或多个指针。

10.6 指针和数组

指针和数组的关系非常密切。考虑下面的程序段:

```
char str[80], *p1;
p1 = str;
```

这里,将 $p1$ 设置为 str 中的第一个数组元素的地址。在 C 中,不带下标的数组名为数组的起始地址。实际上,它是指向数组的指针。上面语句还可以这样表示

```
p1 = &str[0];
```

但多数程序员认为这种形式不好。

如果希望访问 str 中的第五个元素,可以这样写:

```
str[4]
```

或

```
*(p1+4)
```

两个语句都将返回第五个元素。数组是从 0 开始的,所以 str 的下标为 4。还可以将指针 $p1$ 加 4,以存取第五个元素,因为 $p1$ 当前指向 str 的第一个元素。

实际上,C 允许用两种方法存取数组元素。但因为指针运算要比数组下标运行快一些,运行速度是编程中经常需要考虑的因素,所以在 C 程序中,使用指针来存取数组元素的方法比较常见。

下例说明如何在数组下标运算中使用指针,比较以下两个程序,其中一个使用下标,另一个使用指针,两者都用小写字母显示字符串的内容。

```
#include <stdio.h>
#include <ctype.h>
/* array version */
void main()
{
    char str[80];
    int i;
    printf("enter a string in uppercase: ");
    gets(str);
    printf("here's the string in lowercase: ");
    for(i=0; str[i]; i++) printf("%c", tolower(str[i]));
}
```

```
#include <stdio.h>
#include <ctype.h>
/* pointer version */
void main()
{
    char str[80], *p;
    printf("enter a string in uppercase: ");
    gets(str);
    printf("here's the string in lowercase: ");
    p = str; /* get the address of ctr */
    while(*p) printf("%c", tolower(*p++));
}
```

使用数组的程序执行速度要比使用指针的程序慢,原因是索引数组的时间比使用 * 操作所花的时间长。

要注意,如果严格按上升或下降顺序访问数组,则指针的使用显得更快一些。但如果是随机访问数组,则采用数组索引的方法更好,因为它通常是与计算一个复杂的指针表达式一样快,而且易于编码和理解。还有,如果使用数组索引,可以使编译程序也能够帮助做一些事情。

10.6.1 索引指针

在 C 中,有时要象索引数组一样索引指针。指针和数组存在着密切联系。例如,下面的程序段是非常正确的,在屏幕上打印从 1 到 5 的数字:

```
/* Indexing a pointer like an array. */
#include <stdio.h>
void main()
{
    int i[5] = {1, 2, 3, 4, 5};
    int *p, t;
    p = i;
    for(t=0; t<5; t++) printf("%d", p[t]);
}
```

在 C 中,语句 `p[t]` 等同于 `*(p+t)`。

10.6.2 指针和字符串

由于没有下标的数组名是一个指向数组的第一个元素的指针,当使用在前面章节中讨论过的字符串函数时,只有一个指向字符串的指针被传递到函数中,而不是字符串本身。下面是编写 `strlen()` 函数的一种方法,可以帮助读者理解这一过程。

```
strlen(char *s)
{
    int i=0;
    while (*s)
```

```

    {
        i++;
        S++;
    }
    return i;
}

```

在 C 中的所有字符串都以 0 结束,这相当于一个假值。因此,象类似下面这样的语句

```
while (*s)
```

在未到达字符串末尾之前为真。如果调用该函数的字符串长度为 0,则返回 0,否则,返回字符串的长度。

用户可能已对 strlen()函数使用常量作为自变量感到怀疑。例如,可能不明白下面这段程序是如何工作的:

```
printf("length of TEST is %d", strlen("TEST"));
```

当使用一个字符串常量作为自变量时,只有一个指针传递到 strlen()函数中。实际字符串由 C 自动存储,因此,strlen()可用常量作为自变量。

更一般的情况是,一个字符串常量可用于任何类型的表达式中的,把它作为为一个指向字符串的第一个字符的指针处理。例如,下面的程序将在屏幕上打印词组“this program works”。

```

#include <stdio.h>
void main()
{
    char *s;
    s = "this program works";
    printf(s);
}

```

组成字符串常量的字符存储在由 C 的特别“字符串表”中。用户程序使用一个指针指向该表即可。

10.6.3 如何得到一个数组元素的地址

到目前为止,所有的例子都集中在把数组的起始地址赋给一个指针。实际上,也可以使用 & 操作符将数组的某个特定元素的地址赋给指针。例如,这个程序段将 x 的第三个元素的地址放在 p 中:

```
p=&x[2];
```

这种方法特别适用于查找一个子字符串。例如,下面这个程序将打印从键盘输入的字符串中第一个空格后的剩余部分:

```

#include <stdio.h>
/* Display the string left after the first space
   is encountered. */

void main()
{

```

```

char s[80];
char *p;
int i;
printf("enter a string: ");
gets(s);
/* find first space or end of string */
for(i=0; s[i] && s[i] != ' '; i++);
p = &s[i];
printf(p);
}

```

p 将指向一个空格或空(如果字符串中没有空格)。如果指向空格,则打印空格后字符串的剩余部分,如果指向空字符,则 printf() 不打印。例如,如果键入“are there”则显示“there”。

10.6.4 指针数组

指针可以组成任何数据类型的数组,例如大小为 10 的 int 型的指针数组说明形式为:

```
int * x[10];
```

把一个整型变量 var 的地址赋给指针数组的第三个元素,可以这样写:

```
x[2]=&var;
```

要得到 var 的值,可以这样写:

```
* x[2]
```

指针数组通常用来存储表示错误的信息,通过生成一个输出信息的函数,给出它的代码数字,如下 error() 所示:

```

char * err[] =
{
    "cannot open file\n",
    "read error\n",
    "write error\n",
    "media failure\n",
};
error(int num)
{
    printf("%s", err[num]);
}

```

可以看到,在 error() 中调用的 printf() 带有一个字符指针,它是指向一个由传递给函数的错误号码索引到的错误信息。例如,如果传递给 num 的号码为 2,则显示信息 write error。

另一个有趣的初始化的字符指针数组应用程序使用了 C 的 system() 函数,该程序发送给操作系统一个命令。system() 的调用方式如下所示:

```
system("command");
```

这里,command 为将要执行的操作系统的命令。例如,在 DOS 环境下,下面的语句将显示缺省目录的文件列表:


```
system("dir");
```

下面的程序实现了一个非常小的操纵菜单用户接口,可执行四个 DOS 命令:dir,chkdsk,time 和 date.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
void main()
{
    /* create an array of strings */
    char *command[] =
        {
            "DIR",
            "CHKDSK",
            "TIME",
            "DATE"
        };

    char ch;
    for(;;)
    {
        do
        {
            printf("1: directory\n");
            printf("2: check the disk\n");
            printf("3: set date\n");
            printf("4: set time\n");
            printf("5: quit\n");
            printf("\nselection: ");
            ch = getch();
            printf("\n");
        } while((ch < '1') || (ch > '5'));
        if(ch == '5') break; /* end */
        /* execute the specified command */
        system(command[ch - '1']);
    }
}
```

扩展这个程序,可以使用户使用更多的 DOS 命令。例如,可以加入 copy 命令,并且当选择它时,提示用户输入源文件名和目标文件名。copy 命令和文件名可连接起来以字符串形式传递给 DOS。

10.6.5 一个使用数组和指针的实例

通过下面这个非常简单的从英文到德文的翻译程序,可以进一步理解数组和指针间的相互联系,该程序使用一个二维的字符串数组,提供了一个很小的英文和德文单词的对应

表。它先让用户输入一个英文句子,然后输出一个非常粗略的德文译文(德文中如果性别不同,用词也不同,该程序忽略了这一点,而且也没有完成动词的变化)。

首先需要—个英-德文对照表,如下所示,用户可以对它进行扩展:

```
char trans[][20] =
{
    "is", "ist",
    "this", "das",
    "not", "nicht",
    "a", "ein",
    "book", "Buch",
    "apple", "Apfel",
    "I", "Ich",
    "bread", "Brot",
    "drive", "fahren",
    "to", "zu",
    "buy", "kaufen",
    "", ""
};
```

每个英文单词与德文单词成对出现。注意最长的单词不能超过 19 个字符。

该翻译程序的 main() 函数和所使用的全局变量如下所示:

```
char input[80];
char word[80];
char *p;
void main()
{
    int loc;
    printf("Enter English sentence: ");
    gets(input);
    p = input;
    printf("Rough German translation: ");
    get_word();
    /* This is the main loop. It reads a word at a time
       from the input array and translates each word
       into German.
    */
    do
    {
        /* find the index of the English word in trans */
        loc = lookup(word);
        /* printf the German if a match is found */
        if (loc != -1) printf("%s ", trans[loc+1]);
        else printf("<unknown> ");
    }
```

```

    get_word();
} while( * word);
}

```

该程序的操作过程如下：首先，提示用户输入一个英文句子，放在 input 字符串中，指针 p 被赋予 input 的起始地址；get_word() 使用这个指针每次从 input 字符串中读取一个单词，每个单词都存入 word 数组中；然后主循环用 lookup() 函数访问每个单词，它返回英文单词的编号为 -1，表示该英文单词不在对照表中，将数组下标加 1 即可查到相应的德文单词。

lookup 函数如下所示：

```

lookup(char * s)
{
    int i;
    for(i=0; * trans[i]; i++)
        if(! strcmp(trans[i], s)) break;
    if(* trans[i]) return i;
    else return -1;
}

```

调用 lookup() 函数，指针指向英文单词，如果该单词在表中，则返回其下标。如果在表中没有发现这个单词，则返回 -1。get_word() 函数如下所示。在该程序中，规定单词只能由空格和终止符定界。

```

get_word()
{
    char * q;
    /* reload address of word each time function is called */
    q = word;
    /* get the next word */
    while(* p && * p1 != ' ')
    {
        * q = * p;
        p++;
        q++;
    }
    if(* p == ' ') p++;
    * q = '\0'; /* null terminate each word */
}

```

从 get_word() 返回后，全局变量 word 包含句子中的下一个单词为空。

整个的翻译程序如下所示：

```

#include <stdio.h>
#include <string.h>
char trans[][20] =
{
    "is", "ist",
    "this", "das",

```

```

    "not", "nicht",
    "a", "ein",
    "book", "Buch",
    "apple", "Apfel",
    "I", "Ich",
    "bread", "Brot",
    "drive", "fahren",
    "to", "zu",
    "buy", "kaufen",
    " ", " "
};
char input[80];
char word[80];
char *p;
int lookup(char *s)
{
    int i;
    for(i=0; *trans[i]; i++)
        if(! strcmp(trans[i], s))
            break;
    if(*trans[i])
        return i;
    else
        return -1;
}
void get_word()
{
    char *q;
    q = word;
    /* get the next word */
    while(*p && *p != ' ')
    {
        *q = *p;
        p++;
        q++;
    }
    if(*p == ' ') p++;
    *q = '\0';
}
void main()
{
    int loc;
    printf("\nEnter English sentence: ");

```

```
gets(input);
p = input;
printf("Rough German translation: ");
get_word();
do
{
    /* find the index of the English word in trans */
    loc = lookup(word);
    /* printf the German if a match is found */
    if(loc != -1)
        printf("%s ", trans[loc+1]);
    else
        printf("<unknown> ");
    get_word();
} while(*word);
}
```

如果键入“I drive to buy a book”，则程序回答为“Ich fahren zu kaufen ein Buch”，这不合乎正确的德语语法，但也和正确的德语语法大致相同。

研究一下这个程序中指针和数组的相互作用。trans[0]等同于指向表中第一个单元的指针，trans[1]等同于指向表中第二元素的指针，依此类推。另外，trans 数组实际上是指向说明部分中的字符串的指针数组。字符串本身存储在 C 的字符串表中。

10.7 指针的指针

指针数组的概念易于理解，但是，指针的指针却很容易混乱。

指向指针的指针是一种“多重间接”的形式，或者为一个指针链。在图 10.3 中读者可以看到，如果是一个标准指针，指针的值为包含所需数值的变量的地址。如果是指针的指针，则第一个指针的内容是第二个指针的地址，第二个指针指向某个变量，该变量包含所需的数值。

多重间接无论在多大的范围内都能进行下去，但很少需要多个指针指向一个指针，或者真正地灵活使用。过分的间接是难以理解而且容易犯概念性的错误的（不要将多重间接与链接表混淆）。

一个指向指针的指针变量以下面方式说明。即在它的名字前面置以附加星号。例如，这个说明告诉编译程序，newbalance 是一个指向 float 型指针的指针：

```
float ** newbalance;
```

注意 newbalance 不是一个指向浮点数的指针而是指向浮点指针的指针。

为使用指针的指针来间接访问目标值，必须使用两个星号操作符，如下面这个例子所示：

```
#include <stdio.h>
void main()
{
```

```

int x, *p, **q;
x = 10;
p = &x;
q = &p;
printf("%d", **q);          /* print the value of x */
}

```

这里, `p` 被说明为一个整型指针, `q` 为一个指向整型指针的指针。调用 `printf()` 将在屏幕上打印数字 10。

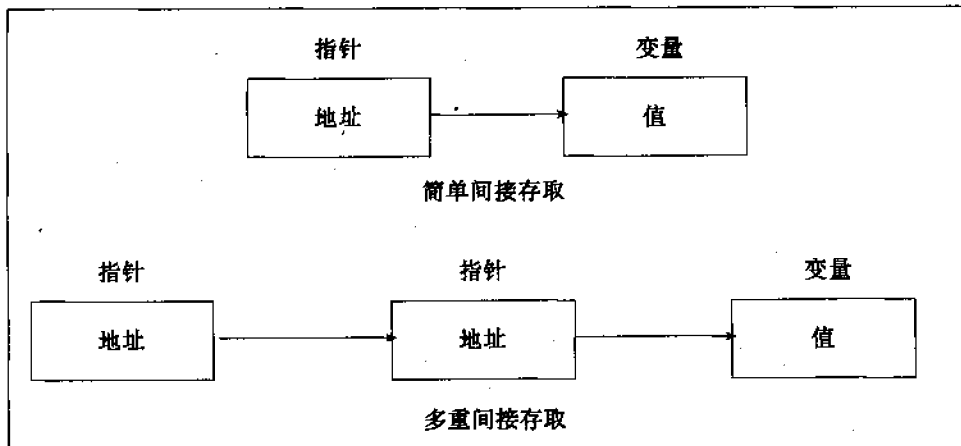


图 10.3 单一和多重指针

10.8 指针的初始化

指针在说明之后和赋值之前,它包含的值是未知的。在赋值前使用指针是一种非常危险的错误,这可能会导致用户的程序,甚至操作系统遭到破坏。

当前不指向任何事物的指针定义为空,以表示它无所指向。空指针是危险的所在,如果在一个赋值语句的右边使用一个空指针,将可能有毁坏程序和操作系统的危险。

空指针可使用户程序易于编码且能提高效率。例如,可以使用一个空指针标志指针数组的结尾。这样,访问该数组的程序遇到空值时,就知道已到达数组的末尾了。下面的 for 循环说明了这个方法:

```

is null */
for(t=0; p[t]; ++t)
if(! strcmp(p[t], name))
break;

```

该循环将执行到匹配成功或遇到空指针时为止。因为数组以空终止符标志结束,所以当到达末端时,控制循环的条件将失败。

在编写专业性 C 程序时,初始化指针是最一般的训练。在本章前面有关指针数组的部分中已出现过两个例子。这方面的另一变化是如下所示的、说明成字符串的类型:

```
char *p="hello world\n";
```

指针 `p` 不是一个数组,由于这种初始化的工作不得不与 C 处理字符串常量的方法相联

系,C生成了由编译程序内部使用的字符串常量,所以上面这个说明语句把存放在字符串表中的字符串“hello world”的地址存放在指针p中。在整个程序中,可以与使用其它任意字符串一样使用p。例如,下面的程序非常正确:

```
#include <stdio.h>
#include <string.h>
char *p="hello world";
void main()
{
    register int t;
    /* print the string forward and backwards */
    printf(p);
    for(t=strlen(p)-1; t>=-1; t--) printf("%c", p[t]);
}
```

但是,用户程序不应通过p给字符串表赋值,否则表会变得混乱不堪。

10.9 指针的一些问题

对某些问题,指针会带来巨大的力量和必要的帮助。但当一个指针偶然包含了一个错误时,则很难发现和追踪。

一个错误指针的故障很难找到,因为指针本身不是问题所在,问题在于每次使用该指针完成一种操作时,都是到未知的内存区域中读取或写入数据。如果从中读取数据,最糟糕的是碰到无用数据。如果写入数据,将覆盖掉原来的代码或数据。这些只有在程序执行时才能发现,所以很难找到真正的错误所在。

尽最大的努力避免指针的使用错误是最根本的。这里讨论一些共同的错误,以一个指针错误的典型例子开始,使用未初始化指针。考虑下面的程序:

```
/* This program is wrong. Do Not Execute. */
void main()
{
    int x, *p;
    x = 10;
    *p = x;
}
```

这个程序把数值10赋给某个未知的内存位置。指针p未被赋予任何值,因此,它包含一个无用的数据,尽管C将对本例中的错误发出警告信息,但当一个指针简单地指向错误的地址时,还会出现同样的错误。例如,可能偶然赋给指针一个错误地址——不在用户代码、数据区或者操作系统中的,然而,随着程序的增大,p指向某个致命错误的可能性也随之增长。最后,程序很可能停止工作。为避免这些麻烦,通常的办法是在指针使用前,让它指向某个有效数据。

常犯的第二种错误是对如何使用指针的误解。考虑下面这个程序:

```
#include <stdio.h>
/* This program is wrong. Do Not Execute. */
```

```
main()
{
    int x, *p;
    x = 10;
    p = x;
    printf("%d", *p);
}
```

调用 `print()` 不能在屏幕上打印 `x` 的值 10。所打印的是一些未知值,原因是赋值语句

```
p=x;
```

是错误的。这个语句将数值 10 赋给指针 `p`。这里 10 变成了一个地址,而不是值。这样写才能保证程序正确:

```
p=&x;
```

在这个简单的例子中,C 将对程序中的错误发出警告,但并非所有这种类型的错误都可以由编译程序所发现。

不要因为会引起错误就不使用指针。用户只要仔细一些,保证使用前就知道指针指向哪里,就不会发生因指针错误而导致的混乱了。

10.9.1 使用 C 语言的间接操作符和取地址操作符

Turbo C 有两个特殊的、用于指针和地址处理的一元运算符。前面已见过它们,因为若没有这两个运算符,那么使用指针是不可能的。这些运算符是间接运算符和取地址运算符。间接运算符用符号 `*` 表示,取地址运算符用符号 `&` 表示。下面详细讨论这些运算符。

间接运算符有双重目的,首先,间接运算符用来说明一个指针,其次,间接运算符可以用来对指针进行引用。引用指针意思是访问(取或存)被指对象的值。

现在来看用间接运算符(`*`)说明最基本的数据类型指针的例子:

```
int *int_ptr;    /* Declare a pointer to int */
char *char_ptr; /* declare a pointer to char */
double *flt_ptr; /* declare a pointer to double */
const int *ci_ptr; /* Declare a pointer to const int */
void *any_ptr;   /* Declare a pointer to anything */
```

正如用户所见,可以说明任何数据类型的指针,有些说明甚至可以指向数组,看下面这个例子:

```
char *char_ptr;
```

`char_ptr` 明显是一个指向字符型对象的指针。但 `char_ptr` 也可用作指向字符数组的指针(不久将看到,字符数组就是字符串),可以使用指针指向任何一种基本数据类型,也可以指向同样类型的数组。

下面的代码片段说明如何向里引用指针:

```
int *int_ptr;
int i = 6;
int_ptr = &i;
printf("i = %d", *int_ptr);
```


在这个例子中,把整型变量的地址赋给了指针。在 `printf()` 语句中,引用了该指针,它是通过表达式 `*int_ptr` 来进行的(即访问对象)。

第二个一元运算符是取地址运算符 `&`,它返回操作数的内存地址。这个内存地址可以是直接赋给正确类型的指针变量。在前几个的例子中,已见过取地址运算符。考虑下面代码片段:

```
char * char_ptr;    /* Declare a pointer */
char c = 'A';       /* Declare a variable */
char str[30];        /* Declare an array */
char_ptr = &c;       /* get address of variable */
char_ptr = str;       /* get address of array */
```

这些例子说明什么时候需要取地址运算符,什么时候不需要。在第 4 行,运算符 `&` 取变量 `c` 的地址,该地址然后赋给了字符指针变量 `char_ptr`。当想得到基本数据对象的地址时,需要取地址运算符。然后在第 5 行,由于数组的地址赋给了字符指针,所以不需要取地址运算符。记住:当使用变量名时,不需要提供下标,返回的值是该数组中第一个元素的地址。由于有了第 1 个元素的地址,所以不必要使用取地址运算符。不带函数调用括号的函数名,不需要取地址运算符也可以产生函数地址。

取地址运算符不能与位域对象或寄存器变量一起使用。位域只能发生在结构中,这一章的后面将讨论它。寄存器变量是 `auto` 类型变量,它要求编译程序当可能时把该变量放在系统硬件寄存器里(关键字 `register` 只是提示编译程序;该变量也可能不放在系统寄存器中)。

在使用指针之前,应该多练习一下,确保说明的指针是否正确地进行了初始化。编译程序不会阻止使用未经过初始化的指针,但结果会出人意料。

赋给指针的值可以是任何正确的数据地址和 `NULL` 值。`NULL`(空)值有一些有趣的特点。首先,`NULL` 能保证与指向对象或函数的指针不相等,因此,在程序中,从来不会把 `NULL` 值与任何正确的指针值相比较,看是否相等。另外,任意的两个 `NULL` 值必定相等。即使指针里存储的值类型不一样,但保证 `NULL` 值都相等。

在指针中,只有三种算术操作是合法的。可以给指针加一个值,减去一个值,或用指针减去指针,所有其它操作都是非法的、无意义的,下面说明三种合法的操作:

- 给指针加一个整数值。给指针加一个整数值会通过数据对象特定的数目增加指针值的。例如,如果给整型指针加上 3,则指针会从当前单元向后指向第 3 个元素(整数),不会指向当前位置后的第 3 个字节。Turbo C 通常按数据对象增加,而不是按字节增加。
- 从指针里减去一个值。减法与加法类似。从 `double` 类型指针里减去 2 会引起指针指向以前的 `double` 类型变量,而不是指向当前位置前面的第 2 个字节位置。当在指针上进行算术操作时,Turbo C 会确保指针按数据对象特定的数目移动,不是按字节移动。
- 将一个指针减去另一个指针。两个指针相减的结果是返回这两个指针之间数据对象的个数,而不是得到数据对象之间的字节数。当把两个指针相减时,Turbo C 将自动得到两者相差的字节数,并除以数据对象的大小。相减的结果可以用类型 `ptrdiff_t` 的变量来存储。

在表达式中,也可以将指针的类型从这一种强制转换为另一种。指针强制转换(cast)简单地告诉编译程序,要使用不同的引用类型。指向的数据并不改变,假若数据有另一种类型的话,编译程序处理这个数据。当用复杂的而不是一般的方法处理数据时,指针转换可能非常有用。UITOH.C 显示了这种指针强制转换。

uitoh.c 逐字节打印 unsigned long 类型整数的程序。

```
1    /* UITOH.C   This program takes an unsigned integer
2                number and uses a type cast to print
3                each byte of the number in hexadecimal.    */
4
5    #include <stdio.h>
6    #include <stdlib.h>
7
8    int main( void )
9    {
10        unsigned long x = 4096;
11        char * dump;
12        int i;
13
14        dump = ( char * ) &x;
15        for( i = 1; i <= 4; i++ )
16            printf( "Byte %d = %.2x\n", i, *dump++ );
17
18        return 0;
19    }
```

UITOH.C 使用了指针转换来打印多类型数据对象的每一个字节。在这里,多类型数据对象是 unsigned long 类型。第 10 行说明了 unsigned long,并进行了初始化。第 11 行里说明了字符型指针。unsigned Long 类型需要 4 个字节的存储空间。然而,打印 unsignedlong 类型的值时,每次用 char 类型指针指向其中的一个字节。在 dump 字符指针可以访问存储在变量 x 中的数据之前,必须进行类型转换。类型转换在第 14 行里进行,其中,取到了 unsigned long 类型变量的地址。然后,这个地址通过类型转换给了字符指针,字符指针能保证一次访问变量数据的一个字节。为 dump 指针一次指向一个字节,所以 printf() 必须被调用 4 次才能完全打印出变量 x 的值。

10.9.2 使用数组和串

聚合(aggregate)类型是一种特殊的派生类型。聚合的对象是由简单的对象组成的,这些简单对象有不同的类型。在聚合对象中可以有的类型包括基本类型、派生类型(函数除外),甚至其它的聚合类型。C 语言的聚合数据对象非常有用,因为它们可以把对象集合当作一个单元(unit)来进行组织和控制,同时允许访问其中的单个成员对象。

在 C 语言中有两种聚合数据对象:数组和结构,虽然联合(union)不是聚合类型,但它与结构有一些相同的地方。所以后面在“使用结构和联合”一节中对它们一起进行讨论。

数组既是派生类型,又是聚合数据类型。在本节中,要学习如何说明,初始化和使用数

组。

这一节中也给说明了如何使用 C 语言的串。在 C 语言中,串是一个字符数组。事实上,术语字符数组与字符串是同义词。要学习如何确定一个给定串的数组大小,如何对串进行初始化,如何给串赋值。

说明和使用对象数组

数组是数据对象的集合,数组中的数据对象称为数据元素,它们有相同的基类型,占据一块连续的存储区域。数据元素可以是任何正确的数据类型。数组说明有如下形式:

`typename identifier [size—constant—expression]`

`typename` 规定数组元素的类型。例如,如果元素类型为 `double`,则数组中每一个元素都是 `double` 类型的浮点数。

`identifier` 是数组的名字。在程序中使用该数组时,使用这个数组名。创建变量名的标准规则同样也适应于数组名的建立。

包围 `size_constant_expression` 的方括号叫做数组说明符,它们必须象上面显示的那样书写。`size_constant_expression` 是一个表达式,它为正整数值,决定数组里有多少个元素。例如如果 `size_constant_expression` 求值为 25,则该数组有 25 个元素。`size_constant_expression` 可以是表达式,但它必须产生一个常量值。不要把变量用作 `size_constant_expression`。

现在看一看下列说明:

`int my_array[100];` `int` 是数组的元素类型,它规定数组中的元素必须为整型对象。`my_array` 是数组的标识符(数组名),值 100 表示数组中元素的个数——它是数组的大小。也就是说,`my_array` 数组有 100 个整型元素。

以后,当访问该数组里的元素时,要使用数组索引(array index);数组名 `my_array` 后跟着一个用方括号围起来的数。在引用数组元素时的方括号叫做数组下标操作符。注意,以前提到的数组说明符实际上是数组下标操作符(在数组说明中,它只是用来表示大小),并且与下标操作符有相同的优先级。

数组索引实际上是给表达式赋一个整数值,表示要访问哪一个数组元素。数组索引也称为数组下标。记住:数组使用从 0 开始的数组索引,这一点很重要。这表示数组里的第 1 个元素的下标为 0,而不是 1。当对数组进行循环操作时,切记要从 0 开始。

下面的代码说明了一个整数数组,并对进行初始化,和打印:

```
int i;
int my_array[100];
for(izo; i <= 99; C++)
    my_array[i] = 100 - i;
for (i=0; i<=99; i++)
    printf("my_array[%d] = %d\n", i, my_array[i]);
```

在代码片段的第 2 行,说明了整型数组 `my_array`,它可以容纳 100 个整数值。

第 1 个 `for` 语句是对数组进行初始化。注意看:数组下标是从 0 开始的。`for` 循环从 0 到 99 开始循环,每次给数组中的一个元素赋初值。当下标增加时,数组 `my_array` 中相应元素的值减小。代码片段的最后一个 `for` 循环打印 `my_array` 中的每一个元素。

从上面的代码中,可以看到,变量可用作数组下标。只有数组大小必须是常量表达式。

使用多维数组与使用一维数组一样容易。只是数组的说明有点变化——可以用数组说明数组(即数组的数组)。例如,可以这样说明一个二维数组:

```
int my_array[10][3];
```

这一个数组有 10 组元素,每组 3 个。这种说明也可以解释为含有 10 个数组的数组,数组中的每一元素恰好有 3 个整数(元素类型仍然是 int)。每一组中的 3 个元素在内存中存储在一起。那么,当顺序扫描这个数组时,最右边的下标首先结束。

要想访问多维数组中的成员,可以使用这样的语句:

```
int *i;
....
i=my_array[5][2];
```

在说明数组时,可以对它进行初始化,这是通过提供一个初始化数据表实现的。初始化数据表是一个用逗号分开、大括号包围的数值表。如下列代码所示:

```
int even_int[5]={2,4,6,8,10};
```

象 Turbo C 样,与 ANSI 兼容的编译程序允许在初始化数值表的最后一个值后面留下一个尾部(结束)逗号。如果以后想扩展这个表(它也可防止编译时出错)的话,这一点比较合适。注意到,在初始化数值表的闭括号后需要分号。

初始化数值表里的值,按从左到右的原则依次赋给相应的数据元素。数据串中最左边的值赋给了 even_int[0]。串里的第 2 个值赋给了 even_int[1],继续这个过程,直到全部赋完。如果提供的初始数据不够的话,那么数组中其余的元素初始化为 0。

如果在数组说明的过程中进行初始化,那么可以省略数组的大小(在方括号里的大小表达式可称为下标界限)。如果在说明数组时,省略了数组大小表达式,编译程序会计算初始化数组表中元素的个数,并且自动分配足够的内存空间来存储所提供的值。因此,可按下列形式重写以前的说明:

```
int even_int[]={2,4,6,8,10};
```

尽管没有规定数组的大小,但仍然可以使用数组下标来访问数组里的任何元素。必须确定数组中有多少元素。超过数组的范围去读一个数值会产生奇怪的结果,因为读取的数据值是无法预料的。

在说明多维数组时,也可以使用初始值表,多维数组初始化数值表可以有选择地使用内部大括号来组织属于不同下标的元素。例如,下面的代码片段说明了两种等效的方法来对含有 3 对屏幕坐标的数组进行初始化。

```
/* declare the array with fully bracketed syntax */
int screenxy[3][2]={
    {0,0},
    {40,40},
    {80,25},
};

/* declare the array without interior bracketing */
int screenxy[3][2]={0,0,40,40,80,25};
```

注意到上述说明中,为了适应从左到右赋值原则,初始值是按顺序写入的。因为给出了

初始化数值表,所以也可省略数组界限值。

使用数组下标并不是访问数组元素的唯一方法,也可以使用指针。使用不带数组下标的数组名可以产生该数组中第一个元素的地址。表达式 `my_array` 给出了 `my_array` 数组中第一个元素的地址。要访问数组中某个特定的元素时,只需把该元素的序号加到数组名返回的地址上去即可。下面的一行代码说明如何才能得到数组 `my_array` 中第二个元素里存储的值:

```
i = *(my_array + 1);
```

不带下标运算符的 `my_array` 给出了该数组中第一个元素的地址,给 `my_array` 加 1 (记住数值下标是从 0 开始的)就给出了该数组中第 2 个元素的地址。不必为 `my_array` 规定要加多少字节。所规定的偏移量需要的字节数会自动地计算出来。间接操作符 `*`, 返回存储在括号里计算出来的地址里的值。

为了取得 `my_array` 数组中的第 5 个元素,只要使用表达式 `*(my_array + 4)` 就可以了。这个表达式与表达式 `my_array(4)` 相等价。数组下标表示法通常与 C 语言中的指针表示法等价。

C 语言的串

串是字符数组,可以用下列方法说明串:

```
char str_1[31];
```

```
char str_2[] = "This is a demonstration string";
```

第 1 个说明, `str_1`, 设置了一个能容纳 31 个元素的字符数组。这个字符数组能容纳长达 30 个字符的串。 `str_1` 只能容纳 30 个字符, 因为, 在 C 语言中, 第一个串都以空字符结尾。空字符“\0”用作屏障, 表示该串已一结尾。

第 2 个说明 `str_2`, 设置了一个字符数组, 其长度为赋给 `str_2` 的串的长度加上空字符的长度 (即 1)。从这个说明中可以看到, 串说明中的初始化值是一串文字, 而不是在纯粹的大括号里的字符表, 注意, 在通常的赋值语句中, 用户不能给一个串赋一串文字。

图 10.4 说明了 Turbo C 如何在机器内存中存储一个串。

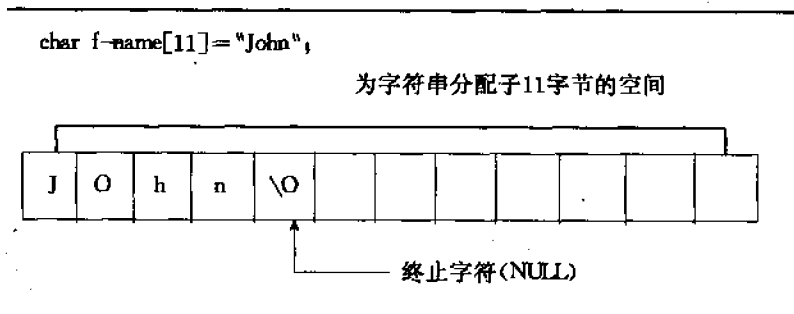


图 10.4 如何存储串

在说明串时,不必把它的长度说明成刚刚能容纳初始值的长度,可把串的长度说明得相对大一些,而且如果用本节后面说明的方法来处理串时,更应该这样作。

要想得到串起始处的地址,要使用不带大括号或元素序号的串名。当使用类似 `printf()`

的函数时,需要规定 printf() 中字符串的起始地址。当打印输出一个串时,printf 只需要提供该串的地址。由于只有串的地址(不是内容)当作参数传送,所以传送串的地址能增加程序的运行速度。下面的一行代码打印串 str_2 的内容:

```
printf("str_2= %s\n", str_2);
```

通过使用 strcpy() 函数可以在串说明外部对串进行初始化。函数 strcpy() 把一个串的内容拷贝到另一个串中。下面是 strcpy() 的用法。

```
* strcpy(char *s1, const char *s2);
```

* s2 指向的字符数组的内容被拷贝到由 *s1 指向的字符数组中。函数 strcpy() 返回一个指针,该指针指向串 s1(s1 的地址并未改变)。由用户即程序员来安排各种情况,以便在接收字符串时有足够的空间,编译程序不能也不会阻止用户从很长的串中拷贝内容。

strcpy() 函数中的实际参数既可以是一串文字,也可以是串名(当然,接收串不能是一串文字),下列代码片段说明 strcpy() 的用法:

```
char s1[30];
char s2[30] = "This is a demonstration";
....
strcpy(s1,s2);
printf("s1= %s\n",s1);
```

这段代码把 s2 数组中的串拷贝到字符数组 s1 中。注意到 strcpy() 函数的变元是指向要处理的串的指针。在 strcpy() 完成之后,printf() 语句打印串 s1 中的内容。

其它两个有用的串处理函数是 strcat() 和 strlen(), strcat() 函数把两个串联起来, strlen() 函数计算串的长度。看下面 strcat() 函数的格式:

```
* strcat(char *s1, const char *s2)
```

strcat() 把 s2 加到(或串接到)s1 的末尾。将 s1 末尾的结尾空字符删除,使得新串的末尾只有一个结尾空字符。象 strcpy() 一样, strcat() 函数也需要一个指向串的指针。当连接成功时, strcat() 返回一个指针给 s1。

下列代码片段说明了 strcat() 的用法:

```
char first_name[s1] = "Bob";
char last_name[20] = "Johnson";
....
strcat(first_name, "");
strcat(first_name, last_name);
printf("The name is : %s\n", first_name);
```

第一个 strcat() 函数给串 first_name 加了一个空字符,这样可以把第一个名和最后一个名正确地分开。第二个 strcat() 函数把串 last_name 加到 first_name 串后面。printf() 函数则打印连接操作的最终结果。

当使用函数 strcat() 时,必须保证原来的串能容纳要拷进去的整个串。检查串的长度可以使用函数 strlen() 来完成。name.c 展示了如何一起使用 strlen() 和 strcat() 函数的方法。

name.c 使用串函数的程序。

```
1 /* NAME.C This program demonstrates the strlen() and
2 strcat() functions. */
```

```

3
4     #include <stdio.h>
5     #include <stdlib.h>
6     #include <string.h>
7
8     #define LONG_STR 81
9     #define SHORT_STR 31
10
11     int main( void )
12     {
13         char first_name[LONG_STR] = "Bob ";
14         char last_name[SHORT_STR] = "Johnson";
15         int i;
16
17         i = strlen( first_name ) + strlen( last_name );
18         if( i <= LONG_STR ) {
19             strcat( first_name, last_name );
20             printf( "string length = %d\n", i );
21             printf( "%s\n", first_name );
22         }
23         else
24             printf( "Strings too long.\n" );
25
26         return 0;
27     }

```

在第 17 行,函数 `strlen()` 计算串 `first_name` 和 `Last_name` 的长度,其结果赋给变量 `i`。第 18 行,把 `i` 的值与最大的串长度值相比较。如果两个串的长度加在一起比最大的串长度要小,那么执行第 19 至 21 行的语句,即把这两个串连接起来,并显示结果。

`strlen()` 函数需要一个参数,该参数是一个指针,它指向待计算长度的串。`strlen()` 返回为数据类型 `size_t` 的值。不过,`strlen()` 返回的值也可以存储在一般的整型量中,就象在第 17 行见到的那样。

可以使用 `strlen()` 函数,利用串是数组的特点,在串的任何位置可以插入单个的字符。下面的代码片段包含了一个完成字符插入的函数。

```

#include <string.h>
.....
void cinset(char ccode, char * anystring, int spos)
{
    int p;
    p = strlen(anystring);
    spos = (spos < 0) ? 0 : spos;
    spos = (spos >= p) ? p : spos;
    for( ; p >= spos; p--) anystring[p + 1] = anystring[p];

```

```
anystring[spos] = ccode;
```

函数 `cinsert()` 可在串的开始、中间和结尾处任何位置插入一个字符。要插入的字符 `ccode` 放置于插入点,插入点由参数 `spos` 定义。注意到 `spos` 相当于下标的作用:在串的开始处插入字符时,把 `spos` 设置为 0。被插入的字符占据指定的位置,原来的字符依次向后移一个位置)。

为了给新来的字符腾出空间,插入地点右边的所有字符都被向右移动一个位置,结尾空字符也要移动。由于字符移动是从右到左进行,所以使用数组下标表示法比使用指针表示法来存取串里的字符要容易一些。下面是函数工作的过程:

1. 首先计算目标串的长度,并把它存在局部变量 `p` 中。注意到,在此时使用 `p` 作为下标会索引到终止空字符的位置,这正是应该做的,因为空字符也要移动。
2. 第一个条件赋值语句确定要求插入的地方是否在串的左边(也就是说 `spos` 是否是负数)。如果是,则插入地点强制在串的起始处。
3. 第二个条件赋值语句确定要求插入的地点是否在串的右边(在空字符后),如果是的话,则把插入点强制在串的最右端。
4. `for` 循环把插入点右边的字符一个一个地向右移,以便给待插入的字符让出空间。这很容易实现,只需连续地把串中 `[p]` 的字符赋给 `[p+1]` 里即可。`for` 循环把串当作其它数组一样对待,用下标来确定需要的位置。许多串函数只使用指针表示法,那是为了处理起来方便有效。
5. 最后,把新的字符放到串的插入地点。

这里出现的串函数只是 Turbo C 提供的众多串函数中的几个。当在程序中加入 `string.h` 头文件时,可以使用库中丰富的串函数。

10.10 使用指向函数的指针

已经知道调用一个 C 语言的函数比较容易,只要简单地写出函数名,后面带上一个用括号围起来的参数表即可。这样调用函数的方法几乎适用于所有的编程任务。然而,C 语言提供了更灵活的调用函数的方法,它让用户对程序有很多的控制。用户可以使用指针来调用函数。指向函数的指针与其它任何指针一样,存储了函数的地址。当进一步引用该指针时,就执行这个函数。

10.10.1 指向函数的指针说明和初始化

当引用指针时,函数的指针使用与其它数据对象的指针使用不一样。当引用一个指向数据对象的指针时,得到指针所指的地址的值,引用一个指向函数的指针不会取回一个值,相反,它调用这个函数。

在指向函数的指针初始化或使用之前,该指针必须要事先加以说明,与其它指针类型一样,说明一个指向函数的指针应有如下形式:

```
type (*function_pointer) (parameter_list );
```

`type` 规定了函数返回值的数据类型,编译程序需要该函数的返回类型,以执行其类型检

查任务,并且还能确定指向函数的指针是否正确。

`function_pointer` 规定用户要使用的函数指针的名字。`*` 是间接操作符,它表示正在说明一个指针,并且说明时符号 `*` 必须存在。

函数指针名用括号围起来,因为函数调用的括号(包围参数表的括号)比间接操作符有更高的操作符优先权。因此,如果不用括号把函数指针名围起来,将说明一个将指针返回给 `type` 的函数。看下面的两个说明:

```
int (*f_ptr)(...); /* 指针指向函数,返回值为 int 型 */
int * f_ptr(...); /* 函数返回一个指针,指向整型数 */
```

第一个说明正确地说明了一个指向函数的指针,但第二个说明却说明了一个函数,它返回一个指向整数值值的指针。在第一个说明语句中的括号强制使得 `f_ptr` 作为一个函数指针来运算。

`parameter_list` 是函数指针说明中任选的部分。当说明函数为指向函数的指针时,省略函数参数表意味着该函数不希望有任何参数。如果函数确实接收参数,那么必须规定每一个参数的类型,以满足类型检查的需要(形式上的参数名可以从原型中省略掉,但类型名不能省略)。

通过在函数指针说明中嵌入数组说明符,可以说明函数指针数组,语法为:

```
type (*function_pointer [bound]) (parameter_list);
```

这个函数指针说明类似于通常的函数指针说明,但有一个区别。`bound` 值规定了指向函数的指针数组中有多少个元素。方括号是数组说明符,并且必须按上述样式写。

在说明函数指针之后,仍然要在使用它之前对它进行初始化。要对指针进行初始化,必须把函数的地址赋给指针。单独使用函数名时就可产生该函数的地址(函数名后不带那些通常都跟在函数名之后的函数调用括号和参数表),可以把这地址赋给函数指针。下面的一行代码对函数指针进行初始化:

```
f_ptr = printf;
```

`printf()` 函数的地址就赋给了函数指针 `f_ptr`。因为 `printf` 是函数名,所以取它的地址时不需要取地址运算符(`&`)。

10.10.2 利用指针引用某调用函数

使用指向函数的指针来调用函数和引用指针一样容易。通过指针可以用两种方法来调用函数。首先,当引用一个指向函数的指针时,该函数被调用。其次,可以用规范的函数调用来调用函数,这是新的 C ANSI 标准所提供的方法。下面的代码片段说明如何通过指针来调用函数。

```
void (*my_clr)(void);
my_clr = clrscr;
...
(*my_clr)(); /* 老办法,但仍然正确 */
...
my_clr(); /* 新 ANSI 方法,有时更好 */
```

代码片段的第 1 行说明了一个函数指针。这个函数指针与函数一起使用,该函数不需要

参数,也不返回值。在第2行,把 `clrscr()` 函数的地址赋给了函数指针 `my_clr`。因为 `my_clr` 有了函数 `clr_scr()` 的地址,所以,当引用 `my_clr` 时就执行那个函数。最后一行使用了新的 ANSI 方法,它通过指针来调用函数。这个方法简单、明了,受人喜欢。然而,会常常想使用老句法来强调一个事实:使用指针来调用函数(特别是在程序中有许多指向函数的指针时)。

CAR.C 说明了指向函数的指针在整个程序上下文中的使用情况。

car.c 使用指向函数指针的程序。

```

1  /* CAR.C Using pointers to functions. Declaring,
2     initializing, and dereferencing function
3     pointers.  */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <conio.h>
8
9  void calc_mileage( void );
10 void maint_schedule( void );
11 void calc_cost( void );
12
13 int main( void )
14 {
15     void ( * car_ptr[3] ) ( void );
16     int i = 0;
17
18     car_ptr[0] = calc_mileage;
19     car_ptr[1] = maint_schedule;
20     car_ptr[2] = calc_cost;
21
22     while( ( i < 1 ) || ( i > 3 ) ) {
23         clrscr();
24         printf( "Car Utilities\n\n" );
25         printf( "1. Gas mileage calculations. \n" );
26         printf( "2. Preventive maintenance schedule. \n" );
27         printf( "3. Calculate cost per mile. \n" );
28         printf( "\nEnter option => " );
29         scanf( "%d", &i );
30     }
31     ( * car_ptr[i-1] ) ();
32
33     return 0;
34 }
35 void calc_mileage( void )
36 {

```

```

37     double begin;
38     double end;
39     double gallons;
40
41     clrscr();
42     printf( " --- Calculate mileage --- \n" );
43     printf( "\n\n" );
44     printf( "\nEnter your beginning mileage => " );
45     scanf( "%lf", &begin );
46     printf( "\nEnter your ending mileage => " );
47     scanf( "%lf", &end );
48     printf( "\nEnter gallons of gas used => " );
49     scanf( "%lf", &gallons );
50     printf( "\n\n\n" );
51     printf( "Your mileage was %f miles per gallon. \n",
52           ( ( end - begin ) / gallons ) );
53     return;
54 }
55
56 void maint _schedule( void )
57 {
58     int miles;
59     int result;
60
61     clrscr();
62     printf( " --- Maintenance Scheduler --- \n" );
63     printf( "\n\n\n" );
64     printf( "Enter your mileage to the nearest thousand "
65           "miles => " );
66     scanf( "%d", &miles );
67     printf( "\n\n\n" );
68     if( ( miles % 3000 ) == 0 )
69         printf( "Time to change the oil. \n\n" );
70     if( ( miles % 5000 ) == 0 )
71         printf( "Time to rotate the tires. \n\n" );
72     if( ( miles % 10000 ) == 0 )
73         printf( "Time for a tune up. \n\n" );
74
75     return;
76 }
77
78 void calc _cost( void )
79 {

```

```
80     double begin;
81     double end;
82     double allowance = 0.25;
83
84     clrscr();
85     printf( " --- Calculate Cost --- \n" );
86     printf( "\n\n\n" );
87     printf( "\nEnter your beginning mileage => " );
88     scanf( "%lf", &begin );
89     printf( "\nEnter your ending mileage => " );
90     scanf( "%lf", &end );
91     printf( "\n\n\n" );
92     printf( "Your cost of operation is %6.2f",
93           ( ( end - begin ) * allowance ) );
94     return;
95 }
```

在 CAR.C 的程序中,使用了一个指针数组来存储每一个函数的地址。指针数组的说明在第 15 行。注意,在函数的指针说明中,使用了函数的类型和函数参数的类型。由于编译程序要检查传给函数的值和函数返回的值,所以,这些信息是编译程序必需的。尽管在这里的参数表和返回类型都是 void,但它们仍然必须在函数指针说明中标写出来。

在第 18 至 20 行,函数的地址赋给了函数指针。要得到函数的地址,只要简单地引用不带函数调用括号的函数名即可!

在第 31 行,调用了适当的函数。表达式 $i-1$ 转换成菜单选择给以 0 开始的数组索引。如果要调用函数,那么必须要引用它的指针。为了确保函数调用正确,要把引用的指针用括号围起来(其原因见操作符优先权)。要确保包括参数表,因为如果没有参数表,函数就不能被调用,因此尽管在这个程序中的函数不需要参数,但 void(空)参数表必须包括在函数调用中。

10.11 在动态内存中使用指针

前面已见过如何给常量、变量和数据结构分配内存空间。当说明任意一个对象时,程序会自动地分配需要的存储空间去存储这些数据项。这些说明在程序中为变量开辟空间,这样,在磁盘上的 EXE 文件会很大。

进一步来说,内部变量虽然比较有用,但用户不能用它们去处理变动的数据量,为了处理更多的数据,可以使用数组。事实上,可以说明一个能容纳比想要使用的数据项更大的数组,然而,说明一个比用户需要的还要大的数组会浪费存储空间。因为程序和数据有时会占较多的空间,所以不得不考虑使用的内存量。现代的 PC 要的内存可以大一些,能力可以强一些,但应用程序的需求也在增大。通常都得考虑使用的机器的内存空量大小。

处理可变的数据量的最好方法是使用动态分配。动态分配对象能让用户在程序运行时为数据分配内存空间。它们之所以是动态的是因为它们需要的空间容量要在运行时才能确

定,因为只有在需要时才分配和使用内存空间,所以动态的对象的效果更好。当一个动态对象不再需要时,该对象使用的空间可以自动释放掉,然后,被分配过的空间又可以为其它对象所用。

这一节说明如何设置和使用动态对象。要学习如何说明一个可以用来访问动态对象的指针,如何为动态对象分配存储空间,如何释放存储空间以作它用。

10.11.1 C 语言程序和动态内存

当动态地分配存储空间时,机器内存使用效率会更高,更有效地使用内存乍一听起来是个好主意,但下面的例子说明了动态内存分配可能会造成很大的差别。

一个处理客户帐户的程序可能有这样的结构:

```
typedef struct {
    char first_name[30];
    char last_name[30];
    char cust_street[30];
    char cust_city[30];
    char cust_state[3];
    int cust_Zip;
    char account_num[10];
    double account_bal;
    double account_limit;
    int overdue;
} cust_type;

...

cust_type c_array[100];
```

结构中的每一个字符需要 1 个字节,每一个整数需要 2 个字节,每一个 double 类型需要 8 个字节,这样的结构类型的对象一共需要 154 个字节的内存空间(把整型变量 cust_zip 按字边界对齐时,浪费了一个字节)。

如果曾对客户信息跟踪过的话,那么就会知道本例中的结构缺少许多用户需要的信息。当说明一个 100 个元素的结构数组时,这个小小的结构会说明程序中究竟要用多少空间——前提是不使用动态存储分配。

包含上述结构的程序需要为结构数组分配比 15K 还多的内存空间。不管是不是使用全部数组,程序中都设置了那么大的数据空间。如果只使用 30 个数组元素的话,那么,将浪费 11K 的内存空间。15K 的数组在磁盘上的 EXE 文件中也要占据很大的空间。

为了避免浪费这么多的内存资源,可以在 C 的堆中为该数组动态地分配空间。这个部分存储不包含在程序中。当程序开始执行时,程序会给出一定量的内存空间,这个内存空间在用户的程序之外,常称为堆(heap),是一块动态内存,如果用户使用适宜的内存分配函数,则该空间对用户是可用的。图 10.5 显示了一个程序使用内存典型的例子。

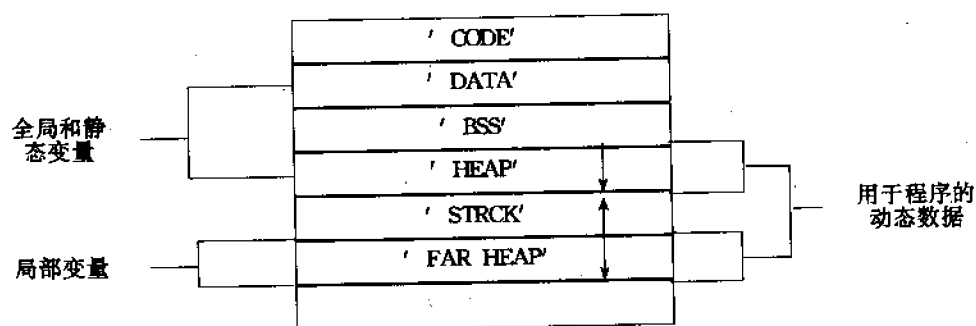


图 10.5 C 程序如何使用内存

在图 10.5 中,分给每部分的内存容量根据编译程序时所使用的内存模式而定。尽管内存的实际容量可以从一个内存模式改变到另一个模式,但所有 Turbo C 的程序都用同样的基本方法使用内存。

程序将其使用的数据存放在 3 个区域:程序、系统堆栈和堆。全局变量和静态变量存储在程序里。这些变量放在 DATA 段(对于初始化过的变量)和 BSS 段(对于未初始化的变量)里。这些变量占据的空间属于程序的一部分。只有通过重新编译源程序才可以改变全局变量和静态变量的分配。

局部变量(相对于函数或函数的内部块的局部变量)存储在系统堆栈中,当函数或代码块开始执行时才建立这些变量时,将空间分给它们;当执行结束即退出函数和块之后,这些变量遭到破坏(空间被释放)。从某种意义看来,局部变量是动态分配和销毁的,但它没有动态内存分配真正的含义。

heap(堆)是一块保留给程序动态建立的变量使用的存储区。从堆中,可以获取分配给程序的动态存储空间。要想获取这个空间,要使用 Turbo C 的内存分配函数。

当使用堆来存储数据时,要确切地规定要使用多少内存空间,通过使用动态内存,可以避免浪费空间,这样可为其它数据节约空间。使用堆也可在磁盘上节省地方,因为动态分配的存储空间不包含在程序中。

如果从堆上取到了存储空间,那么,当用完之后,一定要释放这个存储空间。如果不释放获得的存储空间,则可能会把堆上内存花光。虽然使用动态存储效率要高一些,但前提是要正确使用它。

使用动态数据优点在于:当编写程序时,不必确定需要多少内存空间。可以将程序设计成当程序运行时,才计算它所需要的内存容量,那样,让程序最高效率地使用可用的内存资源。

10.11.2 使用动态存储

要想使用动态存储,则必须遵循以下四步:

1. 确定需要多少内存空间。
2. 分配需要存储空间。
3. 使指针指向获得的内存空间。

4. 当用完时,释放掉这部分内存空间。

可以用两种方法来分配存储空间。首先可为指定数量的对象分配存储空间。Turbo C 能确定对象的大小。用户所要做的就是计算要使用的对象的个数。其次可以按指定字节量分配存储空间。对于这种方法,必须计算要分配的容量大小,它是每个对象需要的空间与使用的对象个数的乘积。下面讨论了每一种方法及其相应的分配函数。

第 1 个函数是 `calloc()` 函数,它需要两个参数,其中一个参数规定要获取存储空间的是对象的个数,另一个参数告诉 `calloc()` 要存储什么类型的对象。`calloc()` 进行所需实际存储量的计算工作。另一个分配存储空间的函数是 `malloc()` 函数,它用来分配指定的内存量,在使用时,应指定所需的存储空间大小,通常要写出公式来确定需要的存储空间的大小。

函数 `calloc()` 的函数原型如下:

```
void * calloc (size_t nmemb, size_t size);
```

注意到该函数的说明并未说明一个指向函数的指针,但说明了一个返回指针的函数/`calloc()` 返回一个 `void` 类型指针,它指向堆里分配的存储。如果不能分配内存的话,那么,该函数返回一个 `NULL`(空)指针。

参数 `nmemb` 规定了要求分配空间的成员个数。必须为程序提供一个办法来确定需要分配存储空间的成员个数。参数 `size` 规定要分给空间的数据对象的大小。

Turbo C 有一个运算符,可以用它来提供 `calloc()` 函数中参数 `size` 的值。`sizeof` 运算符把给它的对象当作操作数,计算出数据对象的大小。下面的代码片段显示 `sizeof` 是如何工作的:

```
int i;  
char c_a[] = "This is a string."  
float *;  
double y=4;  
printf("i occupies %d bytes.\n", sizeof(i));  
printf("c_a occupies %d bytes.\n", sizeof(c_a));  
printf("x occupies %d bytes.\n", sizeof(x));  
printf("A long double requires %d bytes.\n",  
sizeof(long double));
```

`calloc()` 函数和 `sizeof` 运算符是很有用的工具,可以处理大多数动态存储分配。通常,要为许多对象分配存储空间,最容易的办法就是使用 `calloc()` 函数和提供对象大小的 `sizeof` 运算符,也可以计算动态数据需要的字节数目,并且按字节数来要求储量,这可以使用 `malloc()` 做到,它是第二种内存分配函数。注意看下面的 `malloc()` 的函数原型:

```
void * malloc (size_t size);
```

`malloc()` 函数从堆里分配一个内存块,内存块的大小是按字节计算,由参数 `size` 来指定。象 `calloc()` 一样, `malloc()` 返回一个 `void` 类型指针,它指向分配的内存块的开始处。如果内存空间不能分配的话, `malloc()` 将返回 `NULL`。

在分配内存块之后,应该检查 `calloc()` 或 `malloc()` 函数的返回值,确定分配是否成功。在程序中,需要提供一个过程来进行分配不成功时所要做的处理。

一旦分配内存空间块成功,需要把该块的地址赋给正确类型的指针变量。这个赋值使用户能对数据存放的地点进行跟踪,和当用完它之后,释放这个内存块(释放这个内存块尤其

重要——如果不这样的话,会占满堆的空间)。下列代码展示了如何存储一个整型数组的地址:

```
int *int_ptr;
...
if (NULL == (int_ptr = calloc(500, sizeof(int)))) {
    printf("could not allocate memory, aborting program.");
    abort();
}
```

代码的第 1 行说明一个指向整型数的指针,用这个指针来跟踪分配的存储块的首地址。在 if 块里面, `calloc()` 语句为 500 个整数的数组分配存储空间。`calloc()` 返回的值被赋给了 `int_ptr`。如果返回值为 `NULL`,则 `calloc()` 不能成功地分配内存块。如果 `calloc()` 不成功,那么执行 if 块的其余部分,并且退出程序。

`realloc()` 函数可以重新改变分配的内存空间的大小。注意看下面 `realloc()` 的函数原型:

```
void *realloc (void *block, size_t size);
```

从 `realloc()` 中返回的值是一个指向重新分配的内存块的指针,新的存储块的地址可能与老存储块的地址不一样。如果 `realloc()` 返回为 `NULL`,则原来的存储块仍然存在,但不能重新分配。

参数 `block` 是一个指针,它指向重新分配的存储块(由于参数类型为 `void`,所以可以用任何指针类型调用这个函数);参数 `size` 告诉 `realloc()` 重新分配的存储块要使用多少空间。新存储块要使用的空间大小可以比老存储块使用的空间容量大或小。如果新的 `size` 比老的 `size` 要小的话,则老存储块中新 `size` 大小的存储单元就拷给新的存储块(其余的则截掉)。如果新的 `size` 比老的 `size` 要大,则应该考虑到新存储块中的额外(附加)部分还未初始化。

`realloc()` 有两个非常有用的特征。第一,如果用设置为 `NULL` 的 `block` 指针去调用 `realloc()` 函数,则它与 `malloc()` 一样操作,分配存储块(如果有可能)。第二,如果参数 `size` 的大小为 0,但 `block` 指针不为 `NULL` 时,则这个存储块被释放,好象调用了函数 `free()` 一样。

也许,管理堆中最重要的部分是释放旧的内存分配。如果在程序运行过程中,从不释放获得过的动态存储空间,那么可能会占满堆。释放动态存储空间不协调也会得到坏的效果。如果只释放一部分堆中的对象,那么可能把堆分割了。这意味着动态数据对象分散在堆里,对象之间有小自由区。有可能这些较小的空闲(自由)区中没有一个可以满足下一个分配请求。

因此,当用完动态存储块后,需要释放那个存储块以供它用。`free()` 函数释放 `calloc()` `malloc()`,或 `realloc()` 函数分配的存储块。下面是 `free()` 函数的函数原型:

```
void free (void * block);
```

参数 `block` 是一个指针,它指向被分配的存储块的首地址。这个参数被说明为 `void` 类型指针,所以任何类型的指针都可以作为参数传给 `free()`。由于 `free()` 函数不需要返回一个值,所以把 `free()` 说明为 `void` 类型的函数。

`dyna.c` 展示了一个程序,该程序使用堆来存储二维数组。这个数组唯一的事情就是程序提示输出数组中的元素个数。由于该数组分配的空间是在程序执行时进行的,故它是一个真正的动态数组。

清单 10.6 `dyna.c` 使用动态内存和数组的程序。


```

1  /* DYNA.C This program uses dynamic meory to simulate a
2     two-dimensional array. The number of elements
3     in the array is specified by the user at
4     run-time.  */
5
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <conio.h>
9  #include <alloc.h>
10
11 int main( void )
12 {
13     int *i_ptr;
14     int row;
15     int column;
16     size_t nmemb;
17     int i, j, k = 0;
18
19     clrscr();
20     printf( "\n\nThis program simulates a two "
21         "dimensional array of integers. " );
22     printf( "\n\nEnter the number of rows => " );
23     scanf( "%d", &row );
24     printf( "\n\nEnter the number of columns => " );
25     scanf( "%d", &column );
26
27     nmemb = row * column;
28     if( NULL == ( i_ptr = calloc( nmemb, sizeof( int ) ) ) ) {
29         printf( "Not able to allocate memory. \n" );
30         printf( "Aborting program. " );
31         abort();
32     }
33
34     for( i = 0; i < row; i++ ) {
35         for( j = 0; j < column; j++ ) {
36             i_ptr[ ( i * column + j ) ] = k;
37             k++;
38         }
39     }
40
41     for( i = 0; i < row; i++ ) {
42         for( j = 0; j < column; j++ ) {
43             printf( "%4d ", i_ptr[ ( i * column + j ) ] );

```

```
44     }
45     printf( "\n" );
46 }
47
48     free( i_ptr );
49
50     return 0;
51 }
```

在 dyna.c 中的第 9 行,包括了 alloc.h 头文件。这个头文件包含内存分配函数的说明。如果想使用 calloc(),malloc(),realloc()和 free()函数,那么 alloc.h 头文件不能少。

在第 13 行,说明了整型指针变量 i_ptr,它指向分配的存储块的首地址。使用整型指针可以正确地在分配的存储块里进行索引(下标)操作。

第 19 至 25 行是一个子程序,它提示用户输入数组元素的个数。运行时规定(或计算)元素的个数的能力是使用动态分配的重要好处之一。由于数组是按动态分配的,所以每次程序运行时,用户可以输入不同的数组大小。

第 27 行计算数组中元素的个数,并把值存储在 nmemb 中,存放在 nmemb 中的值当在第 18 行调用 calloc()函数时,决定分给多少空间给该数组。如果第 28 行调用 calloc()不成功的话,则进入到 if 语句中,程序非正常终止。

在第 34 至 39 行,分配的存储空间用数组元素的值进行初始化。数组中的元素根据一行一行的原则进行初始化,从每一行的左端开始,移至右端。外层的循环用来一行一行地把数组递推下去。内层的循环按列的次序移过数组。

由于存储空间是动态地进行分配的,所以不能直接使用一般的二维数组下标。相反,数组中每一个元素的地址得进行计算。第 36 行显示了计算正确的数组索引(下标)的公式。表达式 $i * \text{column}$ 计算当前行位置,column 是每行的列数,i 是当前行的数目,用 column 乘以 i 就可得出当前行相对于存储空间开始处的偏移量。把 j 加到当前行的偏移量上就得出当前元素的偏移量。这个表达式计算当前元素相对应分配的存储块的起始处的偏移量。当前数组元素的偏移量与指针 i_ptr 一起使用,可以确定当前数组元素的确切内存地址。

一旦知道了数组元素的确切内存位置,则其数据值可以存进去,也可以读出来。在第 36 行,变量值赋给了一个数组元素。在第 41 行到 44 行的循环中,第 43 行的 printf()函数显示了每一个数组元素的值。

最后,注意看第 48 行的语句,free()函数释放用来存储数组的存储空间,当用完存储空间时,一定要予以释放。解除不再需要的存储空间部分,释放给其它数据使用。

现在回到在动态分配数组上使用数组下标的思想中来。不能直接使用下标不假,但用户可以与动态分配的数组一起使用下标表示法。然而,用户得为使用括号和间接操作符访问数组花费一定的代价。程序 newdyna.c 说明这种做法。

newdyna.c 使用动态分配数组的下标。

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <conio.h>
4  #include <alloc.h>
```

```
5
6  #define MAXX 8
7  #define MAXY 8
8
9  int main( void )
10 {
11     int ( * iarray)[MAXX][MAXY];
12     size_t nmemb;
13     int i, j, k = 0;
14
15     clrscr();
16
17     nmemb = MAXX * MAXY;
18     if( NULL == ( iarray = calloc( nmemb, sizeof( int ) ) ) ) {
19         printf( "Not able to allocate memory. \n" );
20         printf( "Aborting program." );
21         abort();
22     }
23
24     for( i = 0; i < MAXX; i++ ) {
25         for( j = 0; j < MAXY; j++ ) {
26             ( * iarray)[i][j] = k++;
27         }
28     }
29
30     for( i = 0; i < MAXX; i++ ) {
31         for( j = 0; j < MAXY; j++ ) {
32             printf( "%4d ", ( * iarray)[i][j] );
33         }
34         printf( "\n" );
35     }
36
37     free( iarray );
38
39     return 0;
40 }
```

第6和第7行中,提供了宏MAXX和MAXY来控制数组的界限(这样将该示例简化了些)。

注意第11行是如何说明指向动态数组的指针的。这个指针不再是指向int的简单指针;它现在指向一个多维数组。

仔细看第11行使用括号的指针名:(* iarray),这个括号是需要的,因为下标操作符([])比间接运算符有较高的优先权。因此,省略括号会使得这个说明解释为指向int类型的

指针数组,而不是需要的指向 int 类型数组的指针。

既然该指针已经恰当地说明为一个指向整数数组的指针,那么,可以通过使用下标操作符来引用数组元素,这在第 26 和 32 行有说明。但又得注意:必须使用圆括号来强制操作符的组合。

第十一章

数组、结构、位域、联合和枚举

C 中除了有诸如 char 型、int 型、long 型等基本数据类型以外,还提供了几种高级数据类型,即数组、结构、位域、联合和枚举。正是由于具有这些灵活的可以由用户自行定义的高级数据类型,才使 C 成为了功能强大、应用广泛且为广大程序员所喜爱的程序设计语言。本章首先介绍这些数据类型的语法规则,然后结合具体的程序例子讲解每一种数据类型的用法。对 C 语言不太熟悉的读者应该认真阅读本章的内容。

11.1 高级数据类型的语法规则

11.1.1 数 组

数组说明:

type 说明符[<常量表达式>];

上式说明了一个由 type 类型的元素组成的数组。C 中的数组由一块连续的存储区域构成,该存储区域的大小足够保存数组的全部元素。

如果在数组说明中出现一个表达式,且其计算结果为一整数值,该值为数组的元素个数。数组的下标从 0 至元素数减 1。

多维数组可说明为数组类型的数组,故而 5 * 7 的二维数组 alpha 可说明为:

type alpha[5][7];

在有些上下文中,数组说明符的中括号里没有常量表达式,这样的数组具有不确定的大小,这只有在不必保留数组空间的上下文里才是合法的。例如,数组对象的 extern 说明就不需要有精确的数组维数;数组函数参量也允许用大小不确定的数组作为结构的最终成员,这样的数组不会增加结构的大小。这种结构通常用于动态分配,为了合理地保留空间,数组实际需要的大小必须显式地加到结构上。

除了数组类型表达式作为 sizeof 或 & 操作符的操作数之外,该表达式都可以转换为指向数组或其第一个元素的指针。

11.1.2 结 构

结构为一派生类型,它通常表示用户定义的有名成员(或成分)集。成员可以是任何类型,既可以是基本类型,也可以是派生类型,它们可以以任意顺序出现。Turbo C 的结构类型使用户能象处理单个变量一样来处理复杂的数据结构。

结构用关键字 struct 说明,例如:

```
struct mystruct {...};           /* mystruct 为结构标记 */
...
struct mystruct s, *ps, arrs[10]; /* s 为 mystruct 类型,ps 为指向 mystruct 的指针,
                                   arrs 为 mystruct 数组 */
```

无标结构与 typedef

若去掉结构标记,就成为无标结构。可以用无标结构来说明结构标识符表中用逗号分隔的标识符为给定的结构类型(或由此派生的类型),但不能在别处说明该类型的附加对象:

```
struct {...} s, *ps, arrs[10];    /* 无标结构 */
```

在说明带标或无标结构时,可以创建一个 typedef:

```
typedef struct mystruct {...} MYSTRUCT;
MYSTRUCT s, *ps, arrs[10];        /* 与 struct mystruct s,... 相同 */
typedef struct {...} YRSTRUCT;    /* 无标 */
YRSTRUCT y, *yp, arrs[20];
```

标记和 typedef 的作用相同,两者都可以用在结构说明中。

结构成员说明

大括号内的成员说明表说明结构成员的类型和名字。

结构成员可以是任何类型,但有两个例外:

- ① 成员类型不可与当前正在说明的 struct 类型相同:

```
struct mystruct {mystruct s} s1,s2; /* 非法 */
```

成员可以是当前正在说明的结构的指针,如下例所示:

```
struct mystruct {mystruct *ps} s1,s2; /* 正确 */
```

在说明当前正在说明结构的实例时,也可包含以前定义的结构类型。

- ② 在 C 中,成员不能是“返回...的函数”类型,但可以是“返回...的函数指针”类型。

结构与函数

函数可以返回结构类型或指向结构类型的指针:

```
mystruct func1(void);            /* func1 返回一结构 */
mystruct *func2(void);           /* func2 返回一结构指针 */
```

结构可以用下述方式作为参数传递给函数:

```
void func1(mystruct s);          /* 直接 */
void func2(mystruct *sptr);      /* 通过指针 */
```

结构成员存取

结构与联合的成员可用选择操作符.和->进行存取。假定 s 为结构类型 S 的一个对象,sptr 为指向 s 的指针,若 m 为在 S 中说明的类型为 M 的成员标识符,则表达式 s.m 和 ptr->m 都为 M 类型,且都表示 s 中的成员对象 m。表达式 sptr->m 与 (*sptr).m 同义。

操作符(.)称为直接成员选择符,操作符(->)称为间接(或指针)成员选择符,例如:

```
struct mystruct
{
    int i;
    char str[21];
```

```

    double d;
} s, * sptr = &s;
...
s.i = 3;                /* 赋给 mystruct 的成员 i */
sptr->d = 1.23;          /* 赋给 mystruct 的成员 d */

```

假定 s 不是左值,而 m 也不是数组类型,则表达式 $s.m$ 为左值。除非 m 为数组类型,否则 $sptr->m$ 也是左值。

如果结构 B 含有类型为结构 A 的成员,则 A 的成员可两次应用成员选择符来存取:

```

struct A
{
    int j;
    double x;
};
struct B
{
    int i;
    struct A a;
    double d;
} s, * sptr;
...
s.i = 3;                /* 赋给 B 的成员 i */
s.a.j = 2;              /* 赋给 A 的成员 j */
sptr->d = 1.23;          /* 赋给 B 的成员 d */
(sptr->a).x = 3.14;      /* 赋给 A 的成员 x */

```

每个结构说明引出一个唯一的结构类型,因此

```

struct A
{
    int i, j;
    double d;
} a, al;
struct B
{
    int i, j;
    double d;
} b;

```

对象 a 和 al 都为 `struct A` 类型,但对象 a 、 b 是不同的结构类型。仅当源和目的为相同类型时结构才能进行赋值:

```

a = al;                /* 正确:同一类型,进行成员——成员赋值 */
a = b;                 /* 非法:不同类型 */
a.i = b.i; a.j = b.j; a.d = b.d; /* 但可进行成员——成员赋值 */

```

结构字对齐

编译程序逐个成员从左至右地给结构分配内存,且内存地址从低至高。在下例中

```

struct mystruct
{
    int i;
    char str[21];
    double d;
} s;

```

该对象占用的内存可以保存 2 字节的整数、21 字节的串和 8 字节的 double 型变量。该对象在内存中的格式由 Turbo C 的字对齐选择项决定。若关闭这一选择项(缺省),s 将分配有 31 个连续字节。若用 -a 编译选择项(或用 Options | Compiler | CodeGeneration 对话框)打开字对齐开关,Turbo C 将用一些字节填补结构,以保证结构按如下方式对齐:

- ① 结构将从字边界(偶地址)开始。
- ② 任何非 char 成员离结构开始点都为偶字节偏移。
- ③ 如果需要的话,加上一个字节,以保证整个结构含有偶数个字节。

当字对齐打开时,将在上例的 double 型变量 d 之前加上一字节,以使该结构具有 32 个字节。

结构名字空间

结构标记名与联合标记及枚举标记共享相同的名字空间,这意味着这种标记在同一作用域内命名必须是唯一的。但是,标记名不必与另外三个名字空间里的标识符不同,这三个名字空间为标号名字空间、成员名字空间和单个名字空间(包括变量、函数、typedef 名和枚举符)。

给定结构或联合里的成员必须是唯一的,但不同结构或联合里的成员名可以相同。例如

```

goto s;
...
s;

struct s                                /* 正确:标记和标号名字空间不同 */
{
    int s;                               /* 正确:标记和成员名字空间不同 */
    float s;                             /* 非法:成员名重复 */
} s;                                     /* 正确:变量名字空间不同 */
union s                                 /* 非法:标记空间重复 */
{
    int s;                               /* 正确:新成员空间 */
    float f;
} f;                                    /* 正确:变量名字空间 */
struct t
{
    int s;                               /* 正确:不同成员空间 */
    ...
} s;                                    /* 非法:变量名字重复 */

```

不完整说明

指向结构类型 A 的指针在说明结构 A 之前可合法地出现在另一结构 B 的说明:


```

struct A;                                /* 不完整 */
struct B { struct A *pa };
struct A { struct B *pb };

```

A 的第一次出现称为不完整说明,因为在该点没有对它定义。这时,由于 B 的定义不需要 A 的大小,故可允许不完整说明。

11.1.3 位 域

可以用 signed 或 unsigned 说明符将整数成员说明为宽度 1~16 位的位域,并可用下述方式确定位域的宽度和可选标识符:

类型指明符 <位域标识符>:宽度;

这里类型指明符为 char、unsigned char、int 或 unsigned int,位域在字内从低位向高位分配,宽度表达式必须存在,且其计算结果必须为 0~16 范围内的常整数。

若没有位域标识符,将分配宽度中指定的位数,但该位域不能存取。这样能够匹配硬件寄存器内的位模式,使有些位闲置不用。例如:

```

struct mystruct
{
    int      1 : 2;
    unsigned j : 5;
    int      i : 4;
    int      k : 1;
    unsigned m : 4;
} a, b, c;

```

产生以下布局:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
←-----									←-----				←-----	
m				k	(unused)				i				i		

整数段按 2 的补码形式存放,其最左位为最高有效位(MSB),对于 signed int 位域,MSB 解释为符号位,因而,对宽度为 2 的位域,其存放的最大二进制值为 11,若是 unsigned,则将解释为 3,若是 signed 则为 -1。在上例中,合法赋值 a.i=6 将置 a.i 为二进制值 10=-2。带符号 int 位域 k 宽度为 1,由于位模式 1 解释为 -1,故可保存值 -1 和 0。

位域仅可在结构、联合和类中说明,存取它们所用的成员选择符.和->与非位域成员相同。而且,由于字节内字位和字节的组织依赖于机器,因而当编写可移植代码时,位域将带来一些问题。

因为 mystruct.x 不能保证处在字节地址上,故表达式 mystruct.x 是非法的。

11.1.4 联 合

联合类型和结构类型具有许多相同的语法和功能特征,它们的关键区别是联合在任一时刻仅允许其中的一个成员处于“激活”状态。联合的大小与最长成员的大小相同。某一时

刻仅能存取联合的一个成员,下例中:

```
union myunion      /* union tag = myunion */
{
    int i;
    double d;
    char ch;
} mu, *muptr=&mu;
```

类型 union myunion 的标识符 mu 可用来存放 2 字节 int、8 字节 double 或单字节 char,但在同一时刻只有其中一个有效。

sizeof(union myunion)和 sizeof(mu)都返回 8,但当 mu 保存 int 对象时,有 6 个字节没有使用;mu 保存 char 时,有 7 个字节没有使用。可用结构成员选择符.和->来存取联合成员,但需要注意:

```
mu.d = 4.016;
printf("mu.d = %f\n", mu.d);      /* 确,显示 mu.d = 4.016 */
printf("mu.i = %d\n", mu.i);      /* 特定结果 */
mu.ch = 'A';
printf("mu.ch = %c\n", mu.ch);    /* 正确,显示 mu.ch = A */
printf("mu.d = %f\n", mu.d);      /* 特定结果 */
muptr->i = 3;
printf("mu.i = %d\n", mu.i);      /* 正确,显示 mu.i = 3 */
```

第二个 printf 是合法的,因为 mu.i 是整型,但 mu.i 的位模式对应于从前赋值的 double 部分,故通常不提供有用的整数解释。

通过适当转换,指向联合的指针可指向它的每个成员,反之亦然。

联合说明的一般语法基本上与结构说明一样,不同点在于:

- 联合可含有位域,仅有一个是激活的,它们都从联合的始点开始。
- 联合仅能通过它的第一个被说明的成员初始化:

```
union local
{
    int i;
    double d;
} a = { 20 };
```

- 联合不能放在类体系中,它不能从任何类中派生出来,也不能作为基类。联合可以有构造函数。
- 无名联合不能有成员函数。

11.1.5 枚 举

枚举数据类型可用来为一组整数值提供便于记忆的标识符。例如,下列说明

```
enum days {sun, mon, tues, wed, thur, fri, sat} anyday;
```

建立一个唯一的整类型 enum days。该类型的一个变量 anydays 和一组带有常量整数值的枚举符(sun, mon, ...)。

若值的范围许可,并且 -b 标志为关闭(缺省为打开,意思是 enum 值总是 int),则

Turbo C 用单字节存放枚举符,但枚举值用在表达式中时,其值总是扩展为 int。出现在枚举符表中的标识符隐含为 unsigned char 或 int 类型,但到底是何值还要取决于枚举符的值。如果所有值都能用 unsigned char 表示,则它就是每个枚举符的类型。

在 C 中,可以把任何 int 类型的值赋给枚举类型变量而不进行类型检查。

利用显式整数初始值,可设置一个或多个枚举符为特定的值。任何不带初值的后继名字将依次递增 1。例如,在下面的说明中

```
/* 初值表达式包含在以前说明的枚举符里 */
enum coins {penny = 1, tuppence, nickel = penny + 4,
            dime = 10, quarter = nickel * nickel} smallchange;
```

tuppence 值为 2, nickel 值为 5, quarte 值为 25。

初值可以是任何产生正负整数值的表达式(在可能的整数扩展后)。这些值通常是唯一的,但重复也是合法的。

enum 类型可出现在允许 int 类型的任何地方。

```
enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
enum days payday;
typedef enum days DAYS;
DAYS * daysptr;
int i = tues;
anyday = mon; /* 正确 */
* daysptr = anyday; /* 正确 */
mon = tues; /* 非法 mon 是一个常量 */
```

枚举标记与结构及联合标记共用一个名字空间,而枚举符与一般变量标识符共用同一名字空间:

```
int mon = 11;
{
    enum days
    {
        .....
        /* 枚举符 mon 隐藏了 int mon 的外部说明 */
        struct days { int i,j }; /* 非法:day 重复标记 */
        double sat; /* 非法:sat 重定义 */
    }
    mon = 12 /* 回到 int mon 作用域 */
}
```

11.2 数 组

一个数组是一个由共同名字相关联的、具有相同类型的变量的集合。在 C 中,所有的数组均分配连续的内存地址。最低位地址与第一个元素相对应,最高位地址与最后一个元素相对应。数组可以是一维的,也可以是多维的。数组中的元素通过下标来查找。

最常用的数组是字符数组。因为在 C 中没有专门的字符串数据类型,所以采用字符数

组来表示字符串。从后面的介绍可以看到,这种字符串的操作方式要比使用专门的字符串类型类型的语言更有力、更灵活。

11.2.1 一维数组

一维数组说明的一般形式为:

```
type var _name[size];
```

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

```
int sample[10]
```

在 C 中,所有的数组都把零当作第一个元素的下标,因此,上式说明了一个有 10 个元素的整数数组,即 sample[0]到 sample[9]。下面的程序把数字 0~9 装入到一个数组中:

```
main()
{
    int x[10];                /* this reserves 10 integer elements */
    int t;
    for (t=0; t<10; ++t) x[t]=t;
}
```

对于一个一维数组,以字节数表示的数组大小可按下列方式计算:

总字节数 = 基本类型数据大小 × 元素个数

因为数组很容易处理大量相关的数据,因此在编程时被广泛使用。例如,使用数组将很容易计算数值表的平均值,如下面程序所示,它读取用户输入的 10 个整数并显示它们的平均值:

```
/* Find the average of ten numbers. */
#include <stdio.h>
main()
{
    int sample[10], i, avg;
    for(i=0; i<10; i++);
    printf("enter number %d:", i);
    scanf("%d", &sample[i]);
    avg = 0;
    /* now, add up the numbers */
    for(i=0; i<10; i++) avg = avg+sample[i];
    printf("The average is %d\n", avg/10);
}
```

数组的边界检测

C 不对数组进行边界检测,即可以超出数组的既定长度。如果这种超出是在赋值过程中出现的,那么要赋给数组变量的数据可能会覆盖一段程序代码。大小为 N 的数组在寻址时若超过 N,不会产生编译或运行时间错误信息,但很可能毁坏用户程序。作为一个程序员,应该保证所有数组的长度足够容纳程序将要存入的数据,以及必要时提供边界检测,例如尽管

数组 `crash` 将超界,但下面的程序仍能编译和运行(千万不可调试这个例子,它将破坏你的系统)。

```
/* An incorrect program. Do Not Execute. */
main()
{
    int crash[10], i;
    for(i=0; i<100; i++)
        crash[i]=i;
}
```

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

可能读者会不理解 C 为什么没有提供数组的边界检查。其答案是,大部分情况下,C 是为代替汇编语言代码而设计的。因此,实际上不包括错误检测,因为它减慢了程序的执行速度。C 希望程序员自己十分负责地防止数组越界。

一维数组是一个表

一维数组本质上是由相同类型的信息所组成的表。例如,下列程序运行之后

```
char str[7];
main()
{
    int i;
    for(i=0; i<7; i++) str[i] = 'A' + i;
}
```

`str` 的内容如下:

str[0]				str[7]			
A	B	C	D	E	F	G	H

11.2.2 字符串

一维数组最广泛的应用是生成一个字符串。在 C 中,一个字符串定义为以空字符终止的字符数组。空字符规定为 `'\0'`,即零。由于存在空终止符,所以有必要使被说明字符数组的长度比要装入的最大字符串长一个字符。例如,如果想说明一个可装入 10 个字符的数组 `str`,应该写为:

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

这样就在字符串末尾给空字符留下了空间。

尽管 C 没有字符串数据类型,但仍然允许使用字符串常量。字符串常量是一个以双引号括起来的字符表。例如:

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

不需要人为地在字符串常量末尾加上空字符——编译程序会自动加上。这意味着字符串 `"Turbo"` 在内存中的存储形式如下:

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

从键盘输入字符串

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

```
gets(array_name);
```

调用 `gets()`,以没有下标的数组名作为自变量来读取字符串,根据 `gets()`的返回值,数组中将存储从键盘上输入的字符串。`gets()`函数一直读取字符,直到输入了回车键为止。`gets()`包含在头文件 `stdio.h` 中。

例如,下面的程序简单地回显从键盘上输入的字符串:

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

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

`gets()`不对调用数组进行边界检测,因此,如果用户输入了一个大于数组大小的字符串,`gets()`仍将其写入到数组末端的后面部分,这样会造成上面讲过的错误或系统死机。

一些字符串库函数

C 支持一整套字符串操作函数。最常用的是:

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

字符串函数都包含在头文件 `string.h` 中。现在,让我们来看看这些函数的使用方法。

● strcpy()函数

`strcpy` 函数的调用形式如下:

```
strcpy(dest, source)
```

`strcpy()`函数用来将字符串 `source` 的内容拷贝到字符串 `dest` 中去。构成 `dest` 的数组必须足以存储 `source` 中所包含的字符串,否则数组将超界,有可能破坏用户程序。

下面的程序把"hello"拷贝到字符串 `str` 中:

```
#include <stdio.h>
#include <string.h>
main()
{
```

```
char str[80];
strcpy(str, "hello");
printf("%s", str);
}
```

● strcat()函数

strcat()函数的调用形式如下:

```
strcat(s1, s2);
```

strcat()函数把 s2 附加到 s1 上, s2 未改变。两个字符串必须都以空字符终止, 最后结果也以空字符终止。例如, 下面这种程序在屏幕上打印 "hello there"。

```
#include <stdio.h>
#include <string.h>
main()
{
    char s1[20], s2[10];
    strcpy(s1, "hello");
    strcpy(s2, "there");
    strcat(s1, s2);
    printf("%s", s1);
}
```

● strcmp()函数

strcmp()函数的调用形式如下:

```
strcmp(s1, s2);
```

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

下面的函数用于鉴别口令:

```
/* Return true if password accepted; false otherwise. */
password()
{
    char s[80];
    printf("enter password:");
    gets(s);
    if(strcmp(s, "password")) /* strings different */
    {
        printf("invalid password\n");
        return 0;
    };
    /* strings compared the same */
    return 1;
}
```

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

为止:

```
#include <stdio.h>
#include <string.h>
main()
{
    char s[80];
    for(;;)
    {
        printf("Enter a string:");
        gets(s);
        if(! strcmp("quit", s)) break;
    }
}
```

● strlen()函数

strlen()函数的调用形式如下:

```
strlen(s);
```

这里 s 是一个字符串。strlen()函数返回字符串的长度。

下面的程序打印由键盘输入的字符串的长度:

```
#include <stdio.h>
#include <string.h>
main()
{
    char str[80];
    printf("enter a string:");
    gets(str);
    printf("%d", strlen(str));
}
```

例如,如果键入"hi there",该程序将显示"8"。空终止符不在 strlen()的计算范围之内。

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

```
/* Print a string backwards. */
#include <stdio.h>
#include <string.h>
main()
{
    char str[80];
    int i;
    printf("enter a string:");
    gets(str);
    for(i=strlen(str)-1; i>=0; i--) printf("%c", str[i]);
}
```

作为最后一个例子,下面的程序说明了这几个字符串函数的使用:


```
#include <stdio.h>
#include <string.h>
main()
{
    char s1[80], s2[80];
    printf("enter two strings:");
    gets(s1);
    gets(s2);
    printf("lengths: %d %d\n", strlen(s1), strlen(s2));
    if(! strcmp(s1, s2)) printf("The strings are equal\n");
    strcat(s1, s2);
    printf(" %s\n", s1);
}
```

如果运行这个程序,输入字符串"hello"和"hello",则结果为:

```
lengths: 5 5
The strings are equal
hellohello
```

空终止符的使用

所有字符串均以空字符结束,充分利用这一点可以简化对字符串的各种操作。例如,下列程序用较少的代码将字符串中的所有小写字母转换为大写字母。

```
/* Convert a string to uppercase. */
#include <stdio.h>
#include <string.h>
#include <ctype.h>
main()
{
    char str[80];
    int i;
    strcpy(str, "this is a test");
    for(i=0; str[i]; i++)
        str[i] = toupper(str[i]);
    printf(" %s", str);
}
```

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

使用 `printf()` 函数打印单个字符串

到现在为止,利用 `printf()` 显示存储在字符数组中的字符串,都是采用下面的基本形式:

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

应该记住 printf() 的第一个自变量是一个字符串, 而且打印的是除格式命令之外的所有字符。因此, 如果只想打印一个字符串, 可以使用下面的形式:

```
print(array_name);
```

例如, 下面的程序在屏幕上打印 "Hello Tom":

```
#include <stdio.h>
#include <string.h>
main()
{
    char str[80];
    strcpy(str, "Hello Tom");
    printf(str);
}
```

11.2.3 二维数组

C 允许使用维数组。多维数组中最简单的形式是二维数组。一个二维数组实质上是一个一维数组表。为说明一个大小为 10×20 的二维整数数组 twod, 可以如下说明:

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

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

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

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

二维数组以矩阵形式存储, 第一个下标表明行, 第二个下标表明列。这说明当按实际存储顺序访问数组元素时, 最右下标要比最左下标变化得快。图 11.1 为一个二维数组在内存中的图解表示。实质上, 第一(最左)下标可想象为指向内存中二维数组某确定行的指针。

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

对于二维数组, 下列公式用来计算所需内存的字节数:

字节数 = 行 \times 列 \times 基数据类型的大小

因此, 一个维数为 10×5 的整数数组应占有 $10 \times 5 \times 2$ 即 100 个字节。

我们常用二维数组处理信息表。例如, 下面这段程序打印一个棒球队的击球平均数:

```
#define PLAYERS 9
#define ATBATS 100
```

```

int battingavg[PLAYERS][ATBATS];

.
.
.

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

```

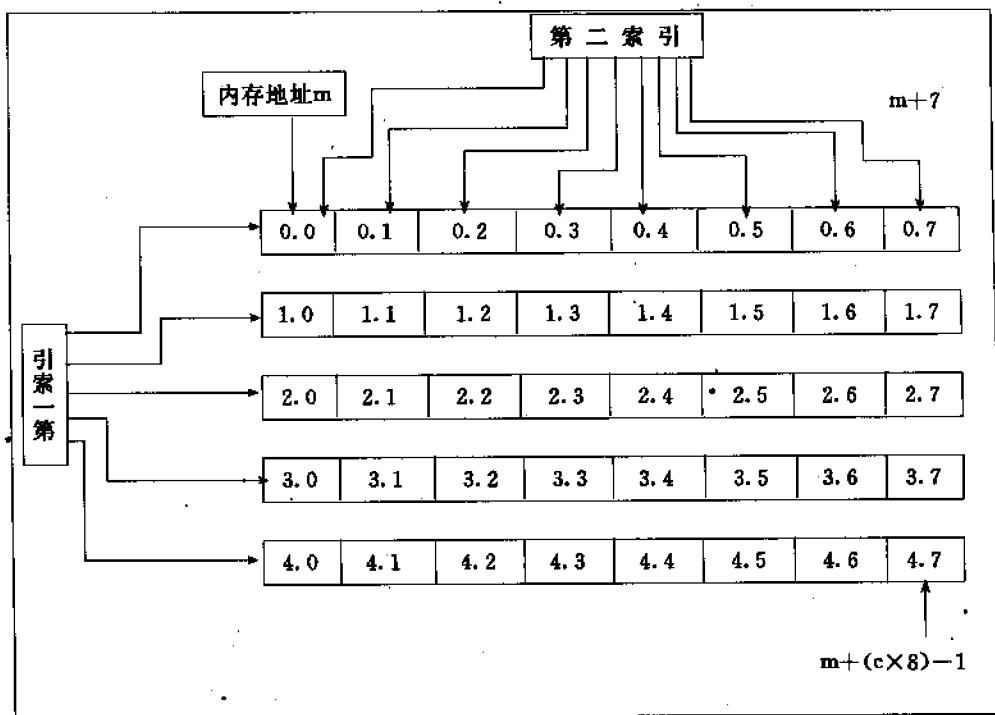


图 11.1 内存中的二维数组

看一下字符串数组

在程序设计过程中,使用字符串数组是司空见惯的。例如,数据库的输入处理程序可从保存有效命令的字符串数组中辨别出有效命令。为生成一个字符串数组,可使用一个二维字符数组,其左下标的大小决定字符串的个数,右下标的大小规定每个字符串的最大长度。例如,下面的程序行说明了一个有 30 个字符串的数组,每个字符串的最大长度为 80 个字符:

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

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

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

这在功能上等同于

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

为帮助读者更好地了解字符串是如何工作的,现在研究下面这个小程序。该程序接收从键盘上输入的正文行,在输入了一行空行后将它们重新显示。

```
/* Enter and display strings. */
#include <stdio.h>
main()
{
    register int t, i;
    char text[100][80];
    for(t=0; t<100; t++)
    {
        printf("%d:", t);
        gets(text[t]);
        if(! text[t][0]) break;    /* quit on blank line */
    }
    /* redisplay the strings */
    for(i=0; i<t; i++)
        printf("%s\n", text[i]);
}
```

该程序接收正文行,直到键入一个空行,然后重新显示各行。

11.2.4 多维数组

多维数组的一般形式为:

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

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

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

三维数组不常使用,因为它们需要大量的内存空间。在程序的执行过程中,全局数组元素将常驻内存。例如,一个维值为 10、6、9、4 的四维字符型数组需要 $10 \times 6 \times 9 \times 4$ 即 2160 个字节。

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

11.2.5 数组初始化

数组初始化的一般形式和其它变量相似,如下所示:

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

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

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

用于存储字符的数组允许采用下面这种简写形式进行初始化:

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

例如,下面这段程序将 str 初始化为"hello":

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

因为 C 中的字符串必须以空字符结束,所以必须确保说明的数组长度足以容纳要初始化的字符串。这就是为什么"hello"只有五个字符,而 str 为六个字符长度的缘故。录用字符常量时,编译程序自动给它加上空终止符。

多维数组的初始化与一维数组相同。例如,下面的程序将 str 初始化为数字 1 到 10 及它们的平方数:

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

变界数组的初始化

假设使用数组初始化来建立一个错误信息表,如下所示:

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

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

使用这种方法,则该信息表变为:

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

除了减少单调的工作以外,变界数组的初始化方法还允许改变任何信息,而不必担心会偶然漏掉什么。

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

```
int sqrs[][2] = { 1, 1,
                  2, 4,
                  3, 9,
```

```

4, 16,
5, 25,
6, 36,
7, 49,
8, 64,
9, 99,
10, 100
};

```

这种方法的优点在于可伸长或缩短数组,而不必变化数组的维值。

11.2.6 一个水下搜索游戏

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

在这个水下搜索程序中,计算机控制一个潜艇,而人控制一个战舰。这个游戏的目标是操纵一只船去击沉另一只船。两只战舰都有有限的雷达测控能力。潜艇能够找到战舰的唯一办法是正好在战舰的下面,而战舰要发现潜艇,则必须在它的正上方,首先发现另一方的判为胜者,潜艇和战舰在“海洋”中交替移动。

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

main()函数和全局变量如下所示:

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h> /* needed by randomize() */
char matrix[3][3] = { ' ', ' ', ' ',
                      ' ', ' ', ' ',
                      ' ', ' ', ' ' };
int compX, compY, playerX, playerY;
main()
{
    /* randomize the random number generator */
    randomize();
    compX = compY = playerX = playerY = 0;
    for(;;)
    {
        if(sub_tries())
        {
            printf("Submarine wins! \n");
            break;
        }
    }
}

```

```

        if(player_tries())
        {
            printf("Battleship (you) win! \n");
            break;
        }
        display_board();
    }
    display_board();
}

```

randomize()函数用来使C的随机数生成程序初始化。不久可以看到,该程序用来生成计算机的行动步骤。

所需要的第一个函数是生成计算机行动步骤的函数。计算机(潜艇)通过使用C的随机函数生成程序rand()产生它自己的行动步骤,该函数返回一个0到32767范围内的随机数(rand()和randmize()函数都包含在头文件stdlib.h中。randmize()同时还需要头文件time.h)。每次调用都使计算机生成一个行动步骤时,生成的随机数用于表示X和Y坐标。然后这些值经过求模运算,提供一个0到2之间的数。最后,计算机检验自己是否找到了战舰。如果生成的行动步骤与战舰的当前位置坐标相同,则潜艇获胜。如果计算机赢了,函数返回真值;否则返回假值。

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

现在,可以输入这个程序并进行调试,经过思考以后,应该想办法使它更加完善。整个搜索程序如下所示:

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h> /* needed by randomize() */
char matrix[3][3] = {' ', ' ', ' ',
                    ' ', ' ', ' ',
                    ' ', ' ', ' '};

int compX, compY, playerX, playerY;

main()
{
    /* randomize the random number generator */
    randomize();
    compX = compY = playerX = playerY = 0;
    for(;;)
    {
        if(sub_tries())
        {
            printf("Submarine wins! \n");
            break;
        }
    }
}

```

```

        if(player_tries())
        {
            printf("Battleship (you) win! \n");
            break;
        }
        display_board();
    } display_board();
}

int sub_tries()
{
    matrix[compX][compY] = ' ';
    compX = rand() % 3;
    compY = rand() % 3;
    if(matrix[compX][compY] == 'B')
        return 1;          /* submarine won the fight */
    else
    {
        matrix[compX][compY] = 'S';
        return 0;          /* it missed */
    }
}

/* Get the player's next move. */
int player_tries()
{
    matrix[playerX][playerY] = ' ';
    do
    {
        printf("Enter new coordinates (X, Y): ");
        scanf("%d%d", &playerX, &playerY);
    } while(playerX < 0 || playerX > 2 || playerY < 0 || playerY > 2);
    if(matrix[playerX][playerY] == 'S')
        return 1;          /* battleship won the fight */
    else
    {
        matrix[playerX][playerY] = 'B';
        return 0;          /* it missed */
    }
}

/* Display the playing board. */
display_board()
{
    printf("\n");
    printf("%2c | %2c | %c\n", matrix[0][0], matrix[0][1], matrix[0][2]);
}

```



```

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

```

11.3 结 构

在 C 语言中,结构是一种被命名为一个标识符的各种变量的集合。这种方法为将各种信息汇集在一起提供了一个十分便利的途径。结构定义可以用来建立结构变量的格式。构成一个结构的各个变量称为结构元素(C 语言中的结构与 Pascal 中的记录相同)。

通常,结构中的所有元素有着彼此相关的逻辑联系。例如,通讯录中姓名和地址信息通常可以用一个结构来表示。下面这段代码是表示通讯录表的一个结构范例。关键字 `struct` 告诉编译程序要定义一个结构。

```

struct addr
{
    char name[30];
    char street[40];
    char city[20];
    char state[3];
    unsigned long int zip;
};

```

这个定义用分号结束,这是由于结构定义本身是一条语句,而且结构标识符 `addr` 代表这个指定的数据结构。

结构定义并没有说明任何实际的变量,所定义的仅仅是这些数据的形式。为了说明这种结构类型的变量,必须写成:

```
struct addr addr_info;
```

这条语句说明一个 `addr` 类型的结构变量,它被命名为 `addr_info`。当定义一个结构时,实质上是将结构元素的各种复杂的变量形式组合在一起。在说明这种结构形式的实际变量前,它实质上是虚设的。

C 语言自动地为结构或这个结构的各个变量分配适当的内存块。图 11.2 表明 `addr_info` 内存分配情况表。

也可以在定义结构的同时说明一个或几个结构变量,例如:

```

struct addr
{
    char name[30];
    char street[40];
    char city[20];
    char state[3];
    unsigned long int zip;
} addr_info, binfo, cinfo;

```

这条语句定义了一个称为 `addr` 的结构类型,而且说明了属于该类型的变量 `addr`、`info`、`binfo` 和 `cinfo`。

如果仅需要一个结构变量,那么可以省略结构标识符,如:

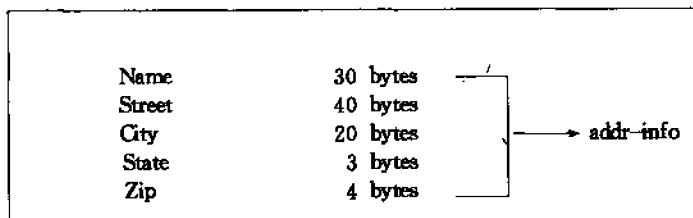


图 11.2 结构 `addr_info` 内存分配表

```
struct
{
    char name[30];
    char street[40];
    char city[20];
    char state[3];
    unsigned long int zip;
} addr_info;
```

这条语句说明了名为 `addr_info` 的结构变量,它使用了前边的结构形式。

结构定义的一般形式为:

struct 结构名

类型 变量名;

类型 变量名;

.....

} 结构变量表;

这里结构名和结构变量可以忽略其一,但不能同时忽略。

11.3.1 访问结构元素

可以用元素选择操作符“.”来访问结构中的单个元素(元素选择操作符通常也叫做点操作符)。例如,下面的语句是将 12345 赋给结构变量 `addr_info` 中的元素 `zip`:

```
addr_info.zip=12345;
```

结构变量名后跟点操作符和元素名代表结构中的单个元素。所有结构元素都可以用下面这种方式进行访问:

结构名. 元素名

因此,为了将 `zip` 的内容打印到屏幕上,可以写为.

```
printf("%d", addr_info.zip);
```

这条语句将显示结构变量 `addr_info` 中元素 `zip` 的内容。

同样,也可以用字符数组 `addr_info.nmae` 来调用函数 `gets()`:

```
gets(addr_info.nmae);
```

这条语句表示将一个字符串指针传递给元素 `name` 的首址。

如果想访问 `addr_info.name` 中的各个元素,可以检索 `name` 中的内容。例如,想再次显示 `addr_info.name` 中每一个字符的内容,可使用下列程序:

```
register int;  
for (t=0; addr_info.name[t]; ++t)  
    putchar(addr_info.name[t]);
```

11.3.2 结构数组

结构数组是最常见的用法。为了说明一个结构数组,首先必须定义一个结构,然后说明这种类型的一个数组变量,例如,为了说明前面已定义过的 `addr` 形式的有 100 个元素的数组,可以写成:

```
struct addr addr_info[100];
```

为了访问某一确定的结构,必须指定结构名的下标。例如,为了打印第三个结构中 `zip` 的内容,可以写成:

```
printf("%d", addr_info[1].zip);
```

同所有数组变量一样,结构数组的下标是从 0 开始的。

通讯录实例

为了帮助理解结构和结构数组的用法,我们来研究一个简单的通讯录程序。它使用结构数组来保存地址信息。程序中的各个函数以各种不同的方式联接结构和它们的元素,从而演示了结构的用法。

在这个例子中,存储的信息包括:

姓名(name)

街道(street)

省、市、邮政编码(city, state, zip)

以下定义了基本结构 `addr` 来保存这些信息:

```
struct addr  
{  
    char name[30];  
    char street[40];  
    char city[20];  
    char state[3];  
    char zip[10];  
} addr_info[SIZE];
```

注意此结构使用字符串而不是无符号整数来保存邮政编码。数组 `addr_info[SIZE]` 属于结构类型 `addr`,这里的 `SIZE` 表示结构的个数。

第一个函数是 `main()`,如下所示:

```
main(void)  
{  
    char choice;  
    init_list();
```

```

for(;;)
{
    choice = menu();
    switch(choice)
    {
        case 'e' : enter();
                    break;
        case 'd' : display();
                    break;
        case 's' : save();
                    break;
        case 'l' : load();
                    break;
        case 'q' : return 0;
    }
}
}

```

首先,函数 `init_list()` 用来将数组中各结构姓名域中的第一个字节赋以空字符值,使得结构数组可用。即若姓名域为空,则这个结构变量还没有使用过。`init_list()` 函数如下:

```

void init_list(void)
{
    register int t;
    for(t=0; t<SIZE; t++) *addr_into[t].name = '\0';
    /* a zero length name signifies empty */
}

```

函数 `menu()` 显示选择项信息,并将所选的项返回给用户:

```

menu(void)
{
    char ch;
    do
    {
        printf("Enter\n");
        printf("Display\n");
        printf("Load\n");
        printf("Save\n");
        printf("Quit\n\n");
        printf("choose one:");
        ch = getche();
        printf("\n");
    } while(! strchr("edlsq", tolower(ch)));
    return tolower(ch);
}

```

该函数使用了 Turbo C 的库函数 `strchr()`, 其函数原型如下:

```
char * strchr(char * str, char ch);
```

此函数查找由 `str` 所指的串, 找出 `ch` 中的字符。如果找到了字符, 就返回指向这个字符的指针, 即返回真值。如果未找到, 则返回一个空值, 即假值。它在此程序中用于查看用户是否输入了有效的选择。

函数 `enter()` 提示用户输入信息, 并将它写入一个空的结构中。如果数组已满, 则将在屏幕上显示信息 "list full";

```
void enter(void)
{
    register int i;
    for(i=0; i<SIZE; i++)
        if(! *addr_info[i].name) break;
    if(i==SIZE)
    {
        printf("list full\n");
        return;
    }
    /* input the info */
    printf("name:");
    gets(addr_info[i].name);
    printf("street:");
    gets(addr_info[i].street);
    printf("city:");
    gets(addr_info[i].city);
    printf("zip:");
    gets(addr_info[i].zip);
}
```

程序 `save()` 和 `load()` 如下所示, 用于保存和调入通讯录数据库。注意由于 `fread()` 和 `fwrite()` 函数的使用使得程序代码变得很短:

```
void save(void)
{
    FILE * fp;
    register int i;
    if((fp=fopen("maillist", "wb"))==NULL)
    {
        printf("cannot open file\n");
        return;
    }
    for(i=0; i<SIZE; i++)
        if(*addr_info[i].name)
            if(fwrite(&addr_info[i], sizeof(struct addr), 1, fp)!=1)
                printf("file write error\n");
}
```

```

        fclose(fp);
    }

void load(void)
{
    FILE *fp;
    register int i;
    if((fp=fopen("maillist", "rb"))==NULL)
    {
        printf("cannot open file\n");
        return;
    }
    init_list();
    for(i=0; i<SIZE; i++)
    if(fread(&addr_info[i], sizeof(struct addr), 1, fp)!=1)
    {
        if(feof(fp))
        {
            fclose(fp);
            return;
        }
        printf("file read error\n");
    }
}

```

两个程序通过检查 `fread()` 或 `fwrite()` 的返回值来确定文件操作成功与否。而且 `load()` 函数必须使用 `feof()` 来检查文件是否结束, 因为 `fread()` 不管到达文件尾还是出错都返回相同的值。

程序中的最后一个函数是 `display()`, 它在屏幕上打印全部通讯录信息:

```

void display(void)
{
    register int t;
    for(t=0; t<SIZE; t++)
    {
        if(*addr_info[t].name)
        {
            printf("%s\n", addr_info[t].name);
            printf("%s\n", addr_info[t].street);
            printf("%s\n", addr_info[t].city);
            printf("%s\n", addr_info[t].state);
            printf("%s\n\n", addr_info[t].zip);
        }
    }
}

```

```
}
```

以下完整地列出通讯录程序。如果对结构的理解有一定的疑问,请将此程序输入到计算机中并研究它的执行过程。进一步可尝试着加入函数来查表,从表中删除一个地址,并在屏幕上打印通讯录。

```
#include <conio.h>
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#define SIZE 100
struct addr
{
    char name[40];
    char street[40];
    char city[30];
    char state[3];
    char zip[10];
} addr_info[SIZE];
void enter(void);
void init_list(void);
void display(void);
void save(void);
void load(void);
int menu(void);
main(void)
{
    char choice;
    init_list();
    for(;;)
    {
        choice = menu();
        switch(choice)
        {
            case 'e' : enter();
                        break;
            case 's' : display();
                        break;
            case 'l' : load();
                        break;
            case 'q' : return 0;
        }
    }
}
```

```

}
void init_list(void)
{
    register int t;
    for(t=0; t<SIZE; t++) *addr_info[t].name = '\0';
    /* a zero length name signifies empty */
}
menu(void)
{
    char ch;
    do
    {
        printf("(E)nter\n");
        printf("(D)isplay\n");
        printf("(L)oad\n");
        printf("(S)ave\n");
        printf("(Q)uit\n\n");
        printf("choose one:");
        ch = getche();
        printf("\n");
    } while (! strchr("edlseq", tolower(ch)));
    return tolower(ch);
}
void enter(void)
{
    register int i;
    for(i=0; i<SIZE; i++)
        if(! *addr_info[i].name) break;
    if(i==SIZE)
    {
        printf("list full!\n");
        return;
    }
    /* enter the information */
    printf("name:");
    gets(addr_info[i].name);
    printf("street:");
    gets(addr_info[i].street);
    printf("city:");
    gets(addr_info[i].city);
    printf("state:");
    gets(addr_info[i].state);
    printf("zip:");

```



```

        gets(addr_info[i].zip);
    }
void display(void)
{
    register int t;
    for(t=0; t<SIZE; t++)
    {
        if(*addr_info[t].name)
        {
            printf("%s\n", addr_info[t].name);
            printf("%s\n", addr_info[t].street);
            printf("%s\n", addr_info[t].city);
            printf("%s\n", addr_info[t].state);
            printf("%s\n\n", addr_info[t].zip);
        }
    }
}
void save(void)
{
    FILE *fp;
    register int i;
    if((fp=fopen("maillist", "wb"))==NULL)
    {
        printf("cannot open file\n");
        return;
    }
    for(i=0; i<SIZE; i++)
        if(*addr_info[i].name)
            if(fwrite(&addr_info[i], sizeof(struct addr), 1, fp)!=1)
                printf("file write error\n");
            fclose(fp);
}
void load(void)
{
    FILE *fp;
    register int i;
    if((fp=fopen("maillist", "rb"))==NULL)
    {
        printf("cannot open file\n");
        return;
    }
    init_list();
    for(i=0; i<SIZE; i++)

```

```

        if(fread(&addr_info[i], sizeof(struct addr), 1, fp) != 1)
        {
            if(!feof(fp))
            {
                fclose(fp);
                return;
            }
            printf("file read error\n");
        }
    }
}

```

11.3.3 结构赋值

如果两个结构变量类型相同,则可以将其中一个的值赋给另一个。这样,左边结构中的所有元素将接收右边结构中相应元素的值。例如,下列程序将 one 的值赋给 two,并显示其结果:

```

#include <stdio.h>
main(void)
{
    struct sample
    {
        int i;
        double d;
    } one, two;
    one.i = 10;
    one.d = 98.6;
    two = one; /* 将一个结构赋给另一个 */
    printf("%d %lf", two.i, two.d);
    return 0;
}

```

注意,不同类型的结构不允许相互赋值,因为其各自的元素不同。

11.3.4 将结构传递给函数

将结构元素传递给函数

当将一个结构变量的元素传递给一个函数时,实际上是将这个元素的值传递给这个函数,因此,传递的只是一个简单的变量(当然,除非这个元素是复合型的,如一个字符数组)。例如,考虑下面这个结构:

```

struct
{
    char x;
    int y;
    float z;
    char s[10];
}

```

```
} sample;
```

下面是一个将结构中的每个元素传递给一个函数的例子：

```
func(sample.x);           /* 传递 X 的字符值 */
func2(sample.y);          /* 传递 Y 的整数值 */
func3(sample.z);           /* 传递 Z 的浮点值 */
func4(sample.s);           /* 传递串 S 的地址 */
func(sample.s[2]);         /* 传递 S[2] 的字符值 */
```

但是,如果想将单个结构元素的地址传递给函数的话,就必须在结构名前使用运算符 `&`。例如,为了将结构变量 `sample` 元素的地址传递给函数,应写成:

```
func(&sample.x);           /* 传递字符 X 的地址 */
func2(&sample.y);           /* 传递整数 Y 的地址 */
func3 (&sample.z);          /* 传递浮点数 Z 的地址 */
func4(sample.s);           /* 传递串 S 的地址 */
func(&sample.s[2]);         /* 传递字符 S[2] 的地址 */
```

请注意,运算符 `&` 放在结构名前,而不是单个元素名前。还应记住,字符串元素 `s` 已经被指定为地址,在这种情况下不需要使用 `&`。

将整个结构传递给函数

当结构作为参数传递给函数时,按标准的赋值调用方法可将整个结构传递给函数。这就是说,函数内部结构内容的改变不会影响到作为实参被调用的那个结构。

专用结构作为形式参数时,最重要的是:形参的类型必须与实参类型一致。例如,下面这个程序说明了参数 `arg` 和 `parm` 为同一结构类型:

```
#include <stdio.h>
/* 定义一个结构类型 */
struct sample
{
    int a, b;
    char ch;
};
void f1(struct sample parm);
main(void)
{
    struct sample arg; /* 声明 */
    arg.a = 1000;
    f1(arg);
    return 0;
}
void f1(struct sample parm)
{
    printf("%d", parm.a);
}
```

可以看到此程序在屏幕上显示 1000。

正如这个例子程序所示,最好是定义一个全局结构,然后用结构名来说明所需的结构变

量和参数。这样就保证了形参和实参的类型一致性。而且,别人阅读该程序时也很容易知道 parm 和 arg 是同一类型的。

11.3.5 结构指针

C 语言允许与其它变量相同的方式来使用结构指针。同时,结构指针还具有其特殊的性质,这一点请特别注意。

下面重点介绍使用结构指针

结构指针有各种用法。一种是用在函数调用过程中,另一种是用来建立链表和使用 Turbo C 内存分配系统来建立其它动态数据结构。本节只讨论第一种用法。

除了最简单的结构外,将结构传递给函数的其它所有方法的一个最主要缺点是要将所有的结构元素压栈(退栈),这就需要额外的时间和空间开销。在只有几个元素的简单的结构中,这种额外的开销带来的问题并不明显,但如果元素很多或者含数组元素的话,在运行时就会有有很大的影响。解决这个问题方法是,仅将一个指针传递给函数。

当一个结构指针传递给函数时,实际上只是将结构地址压栈(或退栈)。这就意味着它能很快地调用函数。另外,由于函数调用中使用的是实际的结构,而不是其副本,因而它能够在调用时修改结构中实际元素的内容。

为了得到结构变量的地址,可在结构名前加上运算符 &。例如,输入下面这段程序:

```
struct bal
{
    float balance;
    char name[80];
} person;
struct bal * p;          /* 声明一个结构指针 */
```

然后再

```
p=&person;
```

即结构 person 的地址赋给指针 p。为了访问元素 balance,可以写成:

```
(*p).balance
```

结构元素很少使用 * 操作符。正如上面的例子所示,因为借助指针访问结构元素十分方便,C 语言定义了一个特殊的操作符 -> 来完成这个任务。C 程序员都把它叫做箭头操作符。它是由一个减号后面跟着一个大于号形成的。当用结构变量指针访问结构元素时,箭头操作符用于代替点操作符。例如,前面的语句通常写成如下形式:

```
P->person
```

为了理解结构指针的用法,下面举一个简单的例子,这个程序用软件延时器显示时、分、秒。

```
#include <stdio.h>
#include <conio.h>
struct time_struct
{
    int hours;
    int minutes;
```

```
int seconds;
};
void update(struct time_struct *t);
void display(struct time_struct *t);
void delay(void);
main()
{
    struct time_struct time;
    time.hours=0;
    time.minutes=0;
    time.seconds=0;
    for( ; 1 kbhit(); )
    {
        update(&time);
        display(&time);
    }
    return 0;
}
void update(struct time_struct *t)
{
    t->seconds++;
    if(t->seconds==60)
    {
        t->seconds = 0;
        t->minutes++;
    }
    if(t->minutes==60)
    {
        t->minutes = 0;
        t->hours++;
    }
    if(t->hours==24)
        t->hours=0;
    delay();
}
void display(struct time_struct *t)
{
    printf("%d:", t->hours);
    printf("%d:", t->minutes);
    printf("%d\n", t->seconds);
}
void delay(void)
{

```

```

long int t;
for(t=1;t<128000;++t);
}

```

程序中时间的校准可以通过调整函数 delay() 中的循环次数来实现。

在程序的开头处定义了全局结构 time_struct, 但并未说明属于该结构类型的变量。在 main() 函数中, time 结构被说明并被初始化为 00:00:00。这意味着 time 只在 main() 函数中有定义。

另两个函数: update() 用来修改时间, display() 用来显示时间。time 的地址被传递给它们。在这两个函数中, 实参都被说明为 time_struct 类型的结构指针, 因此编译程序将知道如何访问结构元素。

实际上每个结构元素都可以用一个指针来访问。例如, 如果想将小时数重新置设为 0, 当时间为 24:00:00 时, 可以写成:

```
if (t->hours == 24) t->hours = 0;
```

这条语句告诉编译程序取 t 的地址(它是 main() 中变量 time 的地址), 并且在调用 hours 时, 将零值赋给这个元素。

请注意, 当在结构自身上操作时要用句点操作符来访问结构元素。当使用结构指针时, 必须用箭头操作符进行操作, 并且必须用 & 操作符获得结构的地址。

我们来研究一些 C 语言的时间和日期函数。系统时间和日期函数的原型定义在文件 time.h 中, 而且该文件还包括两个类型定义。类型 time_t 能够用一个长整数来代表系统时间和日期, 这也叫做日历时间(calendar time)。结构类型 tm 的元素代表时间和时间期, tm 结构定义如下:

```

struct tm
{
    int tm_sec;           /* seconds,          0-59    */
    int tm_min;           /* minutes,         0-59    */
    int tm_hour;          /* hours,           0-23    */
    int tm_mday;          /* day of the month, 1-31    */
    int tm_mon;           /* months since Jan, 0-11    */
    int tm_year;          /* years from 1900   */
    int tm_wday;          /* days since Sunday, 0-6    */
    int tm_yday;          /* days since Jan 1, 0-365   */
    int tm_isdst;         /* Daylight Savings Time indicator */
}

```

如果为夏令时, tm_isdst 的值将为正数, 否则为 0。如果没有可用信息, 则为负数。这种时间和日期的格式叫分离时间(broken_down time)。

C 语言中有关时间和日期的最基本函数是 time(), 其原型为:

```
time_t time(time_t *time)
```

函数 time() 返回距 1980 年 1 月 1 日的秒数。它可以被空指针调用, 也可以被 time_t 型变量指针调用。如果使用后者形参将被赋值为日历时间。

为了将日历时间变换为分离时间, 可使用函数 localtime(), 该函数原型如下:

```
struct tm * localtime(time_t * time)
```

函数 `localtime()` 以 `tm` 结构形式返回指向分离时间的指针。时间由当地时间表示。时间值是通过调用函数 `time()` 获得的。

`localtime()` 用到的结构是静态分配的,它包含分离时间,而且函数每次被调用时都被重写。如果想保存结构内容,必须将它拷贝到其它地方。

尽管用户程序可以使用时间和日期的分离时间形式,但产生时间和日期的最简单的方法是使用 `asctime()`,其函数原型为:

```
char * asctime(struct tm * ptr);
```

函数 `asctime()` 返回一字符串指针,将结构中保存的信息(由 `ptr` 所指的)变换成如下形式:

```
day month date hours:minutes:seconds year\n\0
```

传递给 `asctime()` 的结构指针是由 `localtime()` 获得的。

和 `localtime()` 一样,用于 `asctime()` 的缓冲区是静态分配的字符数组,而且每次函数调用时都要重写。因此,如果要保存字符串的内容,必须将它拷贝到别的地方。

下面的程序用时间函数将系统的时间和日期显示到屏幕上:

```
#include <stdio.h>
#include <time.h>
main(void)
{
    struct tm * ptr;
    time_t lt;
    lt = time('\0');
    ptr = localtime(&lt);
    printf(asctime(ptr));
    return 0;
}
```

Turbo C 还有一些时间和日期函数。这些函数在其它手册中有详细说明。

11.3.6 结构内部的数组和结构

结构元素可以是任何有效的 C 语言数据类型,包括数组和结构。前面已出现过数组元素的例子:在 `addr_info` 中用到字符数组。

根据前面的例子,我们可以将它们推广到一个结构元素是一个数组的情况。例如,考虑下面的结构:

```
struct x
{
    int a[10][10]; /* 10 x 10 array of ints */
    float b;
} y;
```

为了访问结构 `y` 中 `a` 元素的(3,7)这个整型量,可以这样写:

```
y.a[3][7]
```

当一个结构是另一个结构中的元素时,它被称为嵌套式结构。例如,结构变量 address 是嵌套在结构 emp 内的一个嵌套结构:

```
struct emp
{
    struct addr address;
    float wage;
} worker;
```

这里的 addr 在前面已经定义过,而结构 emp 被定义为包含两个元素。第一个元素是 addr 型结构,它含有一个雇员的地址;第二个元素是 wage,它存储着雇员的工资。下面这段程序代码表示将邮政编码 98765 赋给雇员 worker 的 address(地址)域中的 zip:

```
worker.address.zip=98765;
```

正如读者所见,每个结构元素名从最外层到最内层从左至右逐个被列出。

11.4 位 域

与多数计算机语言不同,C 语言用内部法(build-in)来访问一个字节或一个字中的一个或多个位。以下原因表明这是很有用的:首先,如果存储单元被限制,可以在一个字节中存储若干布尔量(逻辑真和逻辑假);其次,某些外设接口是按一个字节的位代码来传递信息的;第三,某些编译程序需要按位访问字节的各个位。

C 语言中访问位的方法是以结构为基础的,叫做位域。一个位域实际上是结构元素的一个特殊形式,它需要定义位的长度。位域定义的一般形式是:

```
struct struct-type-name
{
    type name1,length;
    type name2,length;
    type nameN,length;
}
```

一个位域必须被说明为 int、unsigned 或 signed 中的任一种。长度为 1 的位域被认为是 unsigned 类型,因为单个位不可能具有符号。

函数 biosequip() 的原型在文件 bios.h 中:

```
int biosequip(void)
```

函数 biosequip() 用一个 16 位值返回计算机配置的设备列表,如表 11.1 所示:

表 11.1 计算机配置设置表

位	设 备
0	必须从软驱中启动
1	8087 数字协处理器安装
2,3	母板 RAM 大小 0 0:16K 0 1:32K 1 0:48K 1 1:64K
4,5	初始屏幕模式 0 0:未用 0 1:40×25 BW,彩色适配器

续表 11.1

位	设 备
6,7	1 0 : 80×25 BW, 彩色适配器
	1 1 : 80×25, 单色适配器
	软盘驱动器数目
	0 0 : 一个
	0 1 : 两个
8	1 0 : 三个
	1 1 : 四个
	DMA 配置
	串行口数目
	0 0 0 : 零个
9,10,11	0 0 1 : 一个
	0 1 0 : 两个
	0 1 1 : 三个
	1 0 0 : 四个
	1 0 1 : 五个
12	1 1 0 : 六个
	1 1 1 : 七个
	游戏卡配置
	串行打印机配置
	打印机数目
13	0 0 : 零个
	0 1 : 一个
	1 0 : 两个
	1 1 : 三个
14,15	

用下面的结构可以表示位域:

```
struct equip {
    unsigned floppy_boot:    1;
    unsigned has8087:        1;
    unsigned mother_ram:     2;
    unsigned video_mode:     2;
    unsigned floppies:        2;
    unsigned dma:             1;
    unsigned ports:           3;
    unsigned game_adapter:    1;
    unsigned unused:          1;
    unsigned num_printers:    2;
};
```

下面的程序使用这个位域来显示软盘驱动器的个数和串行口的个数:

```
#include <stdio.h>
#include <bios.h>
main(void)
{
    struct equip {
        unsigned floppy_boot:    1;
        unsigned has8087:        1;
        unsigned mother_ram:     2;
        unsigned video_mode:     2;
        unsigned floppies:        2;
        unsigned dma:             1;
        unsigned ports:           3;
    };
```

```

        unsigned game_adapter;    1;
        unsigned unused;          1;
        unsigned num_printers;    2;
    } eq;
    int *i;
    i = (int *) &eq;
    i = biosequip * ();
    printf("%d floppies\n", eq.floppies+1);
    printf("%d ports\n", eq.ports+1);
    return 0;
}

```

整型指针 *i* 被赋予 *eq* 的地址, 而且 *biosequip()* 的返回值通过这个指针赋给 *eq*。这是必须的, 因为 C 语言不允许一个整数进入结构。

正如这个例子所示, 每个位域都用句点操作符来访问。然而, 如果结构使用指针, 则必须用箭头操作符 \rightarrow 。

不必给每个位域命名。这使得访问所期望的位而忽略未用的那些位变得很方便。unused 域可以无定义, 如下所示:

```

struct equip
{
    unsigned floppy_boot;    1;
    unsigned has8087;        1;
    unsigned mother_ram;     2;
    unsigned video_mode;     2;
    unsigned floppies;       2;
    unsigned dma;            1;
    unsigned ports;          3;
    unsigned game_adapter;   1;
    unsigned;                1;
    unsigned num_printers;   2;
} eq;

```

位域变量有某些限制。不可取一个位域变量的地址; 位域变量不允许是数组; 也不允许超越整型量边界; 在不同类型的 CPU 中, 域是从右到左运行还是从左到右运行是不同的, 任何使用位域的代码都可能与机器的特性有关。

将位域元素和一般的结构元素混合使用是合法的。例如:

```

struct emp
{
    struct addr address
    float pay
    unsigned lay_off; 1;      /* lay off or active */
    unsigned hourly; 1;      /* hourly pay or wage/
    unsigned deductions; 3;   /* IRS deductions */
}

```

这个结构定义了一个关于雇员的记录,它仅用一个字节来存储三个信息:雇员的状态、雇员的工资是否已发以及扣除数目。若不用位域,这些信息就需要三个字节。

11.5 联合(union)

在C语言中,联合表示几个变量公用一个内存位置。这些变量可以是不同的类型。联合(union)的定义与结构(structure)的定义十分相似,例如:

```
union u_type
{
    int i;
    char ch;
};
```

与结构定义相同,这个定义并不说明任何变量。可以在定义时给出变量名或单独使用说明语句来说明一个变量。为了说明变量 `cnvt`(类型为 `u_type`)是一个联合,可以写成:

```
union u_type cnvt;
```

在 `cnvt` 中,整型量 `i` 和字符 `ch` 公用同一内存位置(当然,`i` 占用两个字节,而 `ch` 仅占一个字节)。图 11.3 表示 `i` 和 `ch` 公用同一地址的情况。当一个联合(union)被说明时,编译程序自动地产生一个变量,其长度为 union 中最大的变量的长度。

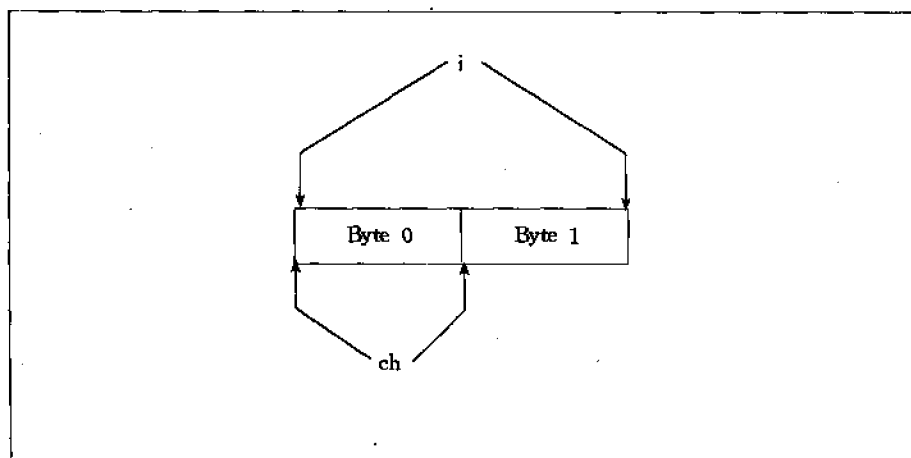


图 11.3 `i` 和 `ch` 公用一个联合 `cnvt`

为了访问联合的元素,可以使用与结构相同的方法,即句点运算符和箭头运算符。如果直接对联合进行操作,则可使用句点运算符;若联合变量是通过一个指针来访问的,那么要用箭头运算符。例如,为了将整型量 10 赋给 `cnvt` 中元素 `i`,可以这样写:

```
cnvt.i=10;
```

联合常用于需要对类型作转换的场合,因为它们可让用户使用不只一种方法查看内存。例如,标准函数 `putw()` 表示写一个二进制的整型量到盘文件。

```
union pw
{
```

```

int i;
char ch[i];
};

```

现在可用联合来编写 putw() 函数:

```

void putw(union pw word, FILE *fp) /* putw with union */
{
    putc(word->ch[0], fp); /* write first half */
    putc(word->ch[1], fp); /* write second half */
}

```

也可以用标准函数 putc() 写一个整型数到盘文件。

下面的程序综合位域和联合, 当按下任一键时, 显示相应的二进制 ASCII 码。union 允许 getche() 将键值赋给字符变量, 同时位域用来显示单个位。研究这个程序并保证完全理解它的执行(此程序的执行只是为了说明联合可用两种根本不同的方法来查找内存中的相同空间。有许多有效的方法书写函数 decode() 以达到同样的结果)。

```

#include <stdio.h>
#include <conio.h>
struct byte
{
    int a : 1;
    int b : 1;
    int c : 1;
    int d : 1;
    int e : 1;
    int f : 1;
    int g : 1;
    int h : 1;
};
union bits
{
    char ch;
    struct byte bit;
} ascii;
void decode(union bits b);
main(void)
{
    do
    {
        ascii.ch = getche();
        printf("%t",
            decode(ascii));
    } while(ascii.ch != 'q');
    return 0;
}

```

```

}
void decode(union bits b)
{
    if(b.bit.h) printf("1");
    else printf("0");
    if(b.bit.g) printf("1");
    else printf("0");
    if(b.bit.f) printf("1");
    else printf("0");
    if(b.bit.e) printf("1");
    else printf("0");
    if(b.bit.d) printf("1");
    else printf("0");
    if(b.bit.c) printf("1");
    else printf("0");
    if(b.bit.b) printf("1");
    else printf("0");
    if(b.bit.a) printf("1");
    else printf("0");
    printf("\n");
}

```

图 11.4 显示了上一个例子的运行结果：

```

a,01100001
b,01100010
c,01100011
d,01100100
e,01100101
f,01100110
g,01100111
h,01101000
i,01101001
j,01101010
k,01101011
l,01101100
m,01101101
n,01101110
o,01101111
0,00110000
1,00110001
2,00110010
3,00110011
4,00110100
5,00110101
6,00110110
7,00110111
8,00111000
9,00111001
=,00111101
\,01011100
q,01110001

```

图 11.4 ASCII 程序的试运行

现在让我们来看一下 union 如何提供一个将整型量装入位域的方法。回忆前节的内容, C 语言不允许一个整数直接赋给一个位域结构。然而, 这个程序创立了一个联合, 它包含一个整型量和一个位域。biosequip() 返回的设备表(用一个整数表示)被赋给一个整数。然而, 当输出结果时程序可自由使用位域。

```
#include <bios.h>
#include <stdio.h>
main(void)
{
    struct equip
    {
        unsigned floppy _boot;    1;
        unsigned has8087;         1;
        unsigned mother _ram;     2;
        unsigned video _mode;     2;
        unsigned floppies;        2;
        unsigned dma;             1;
        unsigned ports;           3;
        unsigned game _adpter;    1;
        unsigned unused;          1;
        unsigned num _printers;   2;
    };
    union
    {
        struct equip eq;
        unsigned i;
    } eq _union;
    eq _union.i = biosequip();
    printf("%d floppies\n", eq _union.eq.floppies+1);
    printf("%d ports\n", eq _union.eq.ports+1);
    return 0;
}
```

11.6 枚 举

枚举是一个被命名了的整型常数的集合。这些常数指定了所有其类型已被定义的各种合法值。枚举在我们的日常生活中十分常见, 例如, 美国使用的硬币的枚举为:

penny, nickel, dime, quarter, half _dollar, dollar

枚举的定义与结构定义十分相似, 用关键字 enum 来表示一个枚举。它的一般形式是:

enum 枚举名 { 枚举表 } 变量表;

枚举表是用逗号分隔的名字表, 代表枚举类型变量可能具有的值。枚举名和枚举变量表都是可选项。枚举名用来说明这种类型的变量。下面这段程序定义了一个称之为 coin 的枚

举类型,并说明 money 属于这种类型:

```
enum coin { penny, nickel, dime, quarter, half_dollar, dollar };
enum coin money;
```

给出这些定义和说明后,下面的语句完全有效:

```
money = dime;
if (money == quarter) printf("is a quarter\n");
```

正确理解枚举的关键是:一个枚举实际上是将每个符号用它们所对应的整数来代替,而且可以在任何一个整型量表达式中使用这些枚举值。除非进行了初始化,否则第一个枚举符号的值为 0,第二个为 1,依次类推。因此,

```
printf("%d %d", penny, dime);
```

将在屏幕上显示 0 2。

也可以用初始化方法指定一个或几个符号的值。这可以通过在符号后加一个等号和一个整型量来实现。当使用初始化方式时,初始化后各个符号所赋的值必须大于前面枚举项的初始值。例如,下面这条语句将 100 这个值赋给 quarter。

```
enum coin { penny, nickel, dime, quarter=100, half_dollar, dollar };
```

现在,这些符号的值如下:

```
penny      0
nickel     1
dime       2
quarter    100
half_dollar 101
dollar     102
```

认为枚举符号可以直接输入和输出是不正确的。例如,下面这段程序与想要得到的结果完全不一致:

```
will will not work */
money = dollar;
printf("%s", money);
```

请注意,符号 dollar 仅仅是一个整型量的名称,并不是字符串。由于同样的原因,下面这段语句不可能达到预期目的:

```
this code is wrong */
gets(s);
strcpy(money, s);
```

也就是说,一个包括一符号名的字符串不能自动转换为那一符号。

实际上,输入和输出枚举符号的代码是毫无价值的(除非希望调整其整型量的数值)。例如,下面这段程序需要显示 money 中硬币的类型:

```
switch(money)
{
    case penny      : printf("penny");
                      break;
    case nickel     : printf("nicke");
                      break;
```

```

        case dime      : printf("dime");
                        break;
        case quarter   : printf("quarter");
                        break;
        case half_dollar : printf("half_dollar");
                        break;
        case dollar     : printf("dollar");
    }

```

有时需要说明字符串数组,并以枚举值作为下标将相应的字符串传送给它们。例如,下面这段程序同样可以输出正确的字符串。

```

char name[][20] = { "penny", "nickel", "dime", "quarter",
                    "half_dollar", "dollar" };

```

```

printf("%s", name[money]);

```

当然,只有在没有任何符号被初始化的情况下才能这样编程,因为字符串数组的下标是从零开始的。例如,下面的程序打印硬币名称:

```

#include <stdio.h>

enum coin { penny, nickel, dime, quarter, half_dollar, dollar };
enum coin money;
char name[][20] =
{
    "penny",
    "nickel",
    "dime",
    "quarter",
    "half_dollar",
    "dollar"
};

main(void)
{
    enum coin money;
    for(money=penny; money<=dollar; money++)
        printf("%s", name[money]);
    return 0;
}

```

由于枚举值必须人工转换成 I/O 控制中可由人识别的字符串值,所以它们在不需要作这种转换的程序中十分有用。例如,经常用一个枚举来定义一个编译程序的符号表。

11.7 使用 sizeof 来确保可移植性

尽管在前面章节中简要地接触到了 sizeof,现在我们仍需进一步研究。如前所述,结构

联合和枚举都可以用来建立大小可变的变量,并且这些变量的实际大小可能随着不同的机器类型而变化。单目操作符 `sizeof` 用来计算任何变量或类型的大小,而且有助于消除程序中与机器类型有关的代码。

下面列出了操作符 `sizeof` 的两种形式:

`sizeof 变量名`

`sizeof(类型名)`

当计算变量大小时,无须将变量名括在括号里(尽管这样做也不错),然而,在计算类型的大小时,必须用括号括起来。例如,下面程序段中的两行都是显示一个 `int` 的大小。

```
int i;
printf("%d", sizeof i);
printf("%d", sizeof (int));
```

Turbo C 规定了下列数据类型的大小:

类型	字节大小
char	1
int	2
longint	4
float	4
double	8
long double	10

因此,下面这段程序在屏幕上显示出 1,2,4 这三个值。

```
char ch;
int i;
double f;
printf("%d", sizeof i);
printf("%d", sizeof f);
```

操作符 `sizeof` 是一个编译阶段操作符,所有在计算变量大小时所需的信息在编译时即为已知的。例如,考虑如下程序:

```
union x
{
    char ch;
    int i;
    float f;
} tom;
```

`sizeof tom` 是 4,因为 `union` 的最大元素是 `float` 型。在运行阶段,不管在 `union tom` 中包含的是什么,所考虑的仅仅是它可能包含的最大变量的大小,因为 `union` 必须和最大元素一样大。

11.8 typedef

C 允许用关键字 `typedef` 定义新的数据类型。实际上,并没有建立一个新的数据类型,而

仅仅是对于现存的类型定义了一个新的名字。这个过程有助于使得与机器有关的程序具有通用性。它仅仅需要改变一下 typedef 语句。由于允许使用标准的数据类型作为说明名,故有助于建立用户的源程序。typedef 语句的一般形式为:

typedef 类型 定义名;

这里,类型是任何一种合法的数据类型,所定义的新的定义名是现有类型名的别名,而不是取代。

例如,可以用下面语句为 float 建立一个新的定义名:

typedef float balance;

这条语句告诉编译程序将 balance 看作实型量的另一定义名。此外,还可以用 balance 来建立一个实型量;balance over_due;

这里,over_due 是一个 balance 类型的浮点变量,而 balance 仅仅是 float 的别名而已,也可以用 typedef 产生形式更为复杂的定义名。例如:

```
typedef struct client_type
{
    float due;
    int over_due;
    char name[40];
} client;
client clist[NUM_CLIENTS]; /* define array of structures of type client */
```

在这个例子中,client 不是 client_type 型变量而是结构 client_type 的别名。

使用 typedef 有助于使用户程序更加易读和更容易移植。但是要记住,不可能建立任何新的数据类型。

第十二章

Turbo C 预处理程序指令

Turbo C 的 IDE 和命令行编译器都采用集成的单遍编译,与多遍编译不同。多遍编译情况下,源文件的头遍编译主要是替换包含文件、测试是否有条件编译指令、进行扩展宏并产生下一遍编译的中间文件。由于 Turbo C 的 IDE 和命令行编译器执行第一遍编译时都不产生中间输出,所以 Turbo C 提供了一个独立的预处理程序 CPP.EXE。它产生上面的中间输出文件.CPP。作为一个调试帮助,使用它可以看到包含指令、条件编译指令和复杂的宏扩展的结果。

因此,以下有关预处理程序指令及其语法和语义的讨论,适用于 CPP 预处理程序和 Turbo C 编译器中的预处理程序。

预处理程序检查预处理程序指令(也称控制行),并分析其中的符号。

Turbo C 预处理程序包含一个复杂的宏处理程序,它在编译器工作之前扫描源代码,预处理程序提供以下的功能和灵活性:

定义宏减轻了编程量,改善了源代码的可读性,有些宏还能消除函数调用的额外开销;

包含其它文件的正文,如包含标准的或用户提供的说明函数原型及常量的头文件;

设置条件编译,以改善可移植性,帮助调试;

预处理程序指令通常放在代码的开头,但语法上它们可以出现在程序的任何地方。

以 # 号开头的行认为是预处理指令,用在串文字量、字符常量或嵌入在注释中的 # 号除外,开头的 # 号前后可带有空格或水平制表符。

Turbo C 预处理程序指令的完整语法如表 12.1 所示:

表 12.1 Turbo C 预处理指令的语法

```

预处理文件:
    组
    组:
        组部分
        组 组部分
    组部分:
        <预处理符号表> 换行
        if 节
        控制行
    if 节:
  
```

续表 12.1

```

if 组<ELIF 组表><ELSE 组>endif 行
if 组:
    #if      常量表达式 换行 <组>
    #ifdef   标识符      换行 <组>
    #ifndef  标识符      换行 <组>
elif 组表:
    elif 组
    elif 组表    elif 组
elif 组:
    #ELIF 常量表达式 换行 <组>
else 组:
    #else 换行 <组>
endif 组:
    #endif 换行
控制行:
    #include  预处理符号表 换行
    #define   标识符 替换表 替换表 换行
    #define   标识符 左括号<标识符表>)替换表 换行
    #undef   标识符 换行
    #line    预处理符号表 换行
    #error   <预处理符号表> 换行
    #pragma  <预处理符号表> 换行
    #pragma  warn 动作 缩写 换行
    #pragma  inline 换行
    # 换行
动作:下述之一
    + - .
缩写:下述之一
    AMB AMP APT AUS BIG CLM CPT
    DEF DUP EFF MOD PAR PIA PRO
    RCH RET RNG RPT RVL SIG STR
    STU STV SUS UCP USE VOI ZST
左括号:
    前面没有空白的左括号字符
替换表:
    <预处理符号表>
预处理符号表:
    预处理符号
    预处理符号表预处理符号
预处理符号:
    头文件名 (仅在#include 指令组)
    标识符

```

续表 12.1

常量
串文字量
操作符
标点符号
每个前面没出现的非空白字符
头文件名:
<头文件字符序列>
头文件字符序列:
头文件字符
头文件字符序列 头文件字符
头文件字符:
源字符集中除 \N 和 > 之外的任何字符
换行:
换行字符

12.1 空指令

空指令由只含有一个字符 # 的行组成,它总被忽略。

12.2 #define 与 #undef 指令

#define 指令定义宏。宏提供了一种词法符号替换机制,它们可带也可不带类似函数参数的形式参数。

12.2.1 简单的 #define 宏

在不带参量的简单情况下,其语法如下:

```
#define 宏标识符 <词法符号序列>
```

在该控制行之后的源程序中一旦出现宏标识符,就有可能被宏的词法符号序列替换(存在一些例外),这种替换称为宏扩展。词法符号序列有时称为宏体。

如果宏标识符出现在串文字量、字符常量和注释中,将不作任何处理。

宏的词法符号序列将在源代码中替换所有能被扩展的宏标识符。

```
#define MI "HAVE A NICE DAY!"
```

```
#define EMPTY
```

```
#define NIL ""
```

```
...
```

```
puts(HI);           /* 扩展为 puts("HAVE A NICE DAY!"); */
```

```
puts(NIL);          /* 扩展为 puts(""); */
```

```
puts("EMPTY");      /* 不扩展 EMPTY */
```

在扩展了每个宏之后,预处理程序会继续扫描新扩展的正文,这样可允许嵌套宏的存

在,即扩展的正文可能又包含有可替换的宏标识符。不过,如果宏扩展为一个预处理指令,则该指令不能被预处理程序识别:

```
#define GETSTD
#include <STDIO.H>
...
GETSTD          /* 编译出错 */
```

GETSTD 将扩展为 #include<STDIO.H>,不过,预处理程序本身将不执行这一明显合法的指令,而把它原封不动地传给编译器。编译器将认为 #include<STDIO.H>为非法输入。宏在其自身扩展期间不再扩展,因而 #define 不会无限制扩展下去。

12.2.2 #undef 指令

#undef 指令使一宏定义无效:

```
#undef 宏标识符
```

该指令将前面定义的词法符号序列与宏标识符分开,使宏标识符无定义。

在 #undef 行里不进行任何宏扩展。

不论实际定义如何,定义或未定义状态是标识符的一个重要特性。#ifdef 和 #ifndef 条件指令用来测试某标识当前是否被宏定义,它们提供了一种灵活的条件判断机制,来控制在不同条件下的编译情况。

在宏标识符被定义后,可用 #define 对它重定义,这时词法符号序列与以前可以相同,也可以不同。

```
#define BLOCK_SIZE 512
...
BUFF=BLOCK_SIZE * BLKS
/* 扩展为 512 * BLKS */
...
#undef BLOCK_SIZE
BUFF=BLOCK_SIZE
/* 这时使用 BLOCK_SIZE 将成为非法的“未知”标识符 */
...
#define BLOCK_SIZE 128 /* 重定义 */
...
BUFF=BLOCK_SIZE * BLKS;
/* 扩展为 128 * BLKS */
```

除非新定义与原定义的词法符号序列相同,否则重定义一个已定义的宏标识符将引起警告。如果包含了其它头文件,又怕引起宏定义冲突,可采取以下方法:

```
#ifndef BLOCK_SIZE
#define BLOCK_SIZE 512
#endif
```

若 BLOCK_SIZE 当前已有定义,则跳过中间一行;若未定义,则执行中间行定义它。

注意:预处理程序指令行后不带分号(;)。在词法符号序列中的任何字符,包括分号都

将出现在宏扩展中。词法符号序列终止于第一次遇到的换行(换行前加反斜杠加以转义表示续行)。词法符号序列中的任何空白序列,包括注释,都将用一个空白字符代替。习惯用汇编语言编程的程序员必须防止出现以下的代码:

```
#define BLOCK_SIZE=512
/* 词法符号序列包括'=' */
```

12.2.3 -D 与-U 选择项

标识符可以用命令行编译选择项-D 和-U 定义或取消定义。利用 IDE 的 OPTIONS | COMPILER | DEFINES 菜单可以定义标识符,但不能显式取消定义。

命令行

```
TCC -DBEBUG=1; PARADOX=0; X -UMYSYM MYPROG.C
```

等价于在程序中写上

```
#define DEBUG 1
#define PARADOX 0
#define X
#undef MYSYM
```

12.2.4 关键字与保护字

使用 Turbo C 关键字作为标识符,这是合法的,但一般不这样做:

```
#define int LONG /* 合法但可能导致灾难 */
#define int LONG /* 合法且可能是有用的 */
```

下面预定义的全局标识符不可出现在 #define 和 #undef 指令中(注意首尾都有双下划线):

```
__stdc__      __date__
__file__      __time__
__line__
```

12.2.5 带参宏

下述语法用来定义带形式参数的宏:

```
#define 宏标识符(<参数表>) 词法符号序列
```

注意宏标识符和(之间不能出现空白,可选参数表为由逗号分开的标识符序列,与 C 函数的参数表没有什么两样。每个由逗号分隔的标识符起着形参的作用。

这种宏在此后的源代码中可以用下述方式调用:

```
宏标识符(<实参表>)
```

该语法与函数调用的语法相同,其实许多标准库“函数”都用宏来实现。不过,它们之间存在一些重要的不同。

可选实参与在 #define 行中的形参表必须含有相同数目的、用逗号隔开的词法符号,即对每个形参必须有一个相应的实参。若两表中参数个数不同,则报告出错。

宏调用要进行两方面的替换:首先,宏标识符和在括号里的参数用词法符号序列替换;

第二,出现在词法符号序列中的形参用实参表中的相应实参替换。例如

```
#define CUBE(X) ((X)*(X)*(X))
```

```
...
```

```
int n, y;
```

```
n=CUBE(y);
```

将引起下述替换:

```
n=((y)*(y)*(y));
```

同样

```
#define SUM(A,B) ((A)+(B))
```

```
...
```

```
int i, j, sum;
```

```
sum=SUM(i,j);
```

的最后一行扩展为 `sum=((i)+(j))`。如果考虑调用

```
n=CUBE(y+1);
```

就会明白增加一对括号的原因。定义里若没有里面的括号,则将扩为

```
n=y+1*y+1*y+1;
```

其分析结果为

```
n=y+(1*y)+(1*y)+1; /* 除非 y=0 或 y=-3, 否则! =(y+1) */
```

和简单宏定义一样,会多次扫描以检测嵌套的可以扩展的宏标识符。

在使用带参数表的宏时要注意以下各点:

- ① 嵌套括号和逗号:实参表可包含相匹配的引号或括号,出现在引号或括号里的逗号不会当作参数分隔符;

```
#define ERRMSG(x, str) showerr("Error", x, str)
```

```
#define SUM(x, y) ((x)+(y))
```

```
...
```

```
ERRMSG(2, "Press Enter, then Esc");
```

```
/* 扩展为 showerr("Error", 2, "Press Enter, then Esc"); */
```

```
return SUM(f(i, j), g(k, l));
```

```
/* 扩展为 return((f(i, j)+(g(k, l))); */
```

- ② 用##粘贴词法符号:在两个词法符号中间加上##(两边可带有空白),可把它们粘贴(或合并)在一起。预处理程序消除空白和##,把分离的词法符号结合成一个新的词法符号。可用它来构造新的标识符,例如给出定义

```
#define VAR(i, j) (i##j)
```

则调用 `VAR(x, 6)` 将扩展为 `(x6)`。它代替了原来的(不可移植的)方法: `(i/* */j)`。

- ③ 用#转换串:为把实参转换为字符串,可把符号#号放在宏的形参前。例如,给出下列宏定义:

```
#define TRACE(flag) printf("#flag"="%d\n", flag)
```

则程序

```
int highval=1024;
```

```
TRACE(highval);
```


将变为:

```
int highval=1024;
printf("highval"="%\n", highval);
```

等价于:

```
int highval=1024;
printf("highval= %\n", highval);
```

- ④ 用反斜杠续行:利用反斜杠\可把一个长词法符号序列写在多行中,在扩展时将去掉反斜杠和后面的换行,以提供实际的词法符号序列:

```
#define WARN "This is really a single-\
            line warning"
```

...

```
puts(WARN);
```

```
/* 屏幕上将显示:This is really a single— line warning */
```

- ⑤ 副作用和其它危险:函数调用和宏调用之间的相似经常掩盖了它们之间的不同。宏调用没有内部类型检查,因而形参和实参数据类型之间的不匹配有时会产生难预料的结果,而且不能立即发出警告。宏调用也可能导致不必要的副作用,尤其当实参计算多次时。比较下例中的 CUBE 和 cube:

```
int cube(int x)
{
    return x * x * x;
}

#define CUBE(x) (x) * (x) * (x)
...
int b=0,a=3;
b=cube(a++); /* cube()传递的参数为 3,故 b=27,而 a=4 */
a=3;
b=CUBE(a++); /* 扩展为((a++) * (a++) * (a++)); 现在 a=6
              */
```

12.3 文件包含指令 #include

#include 指令把其它文件(称为包含文件或头文件)包含到源代码中。其语法有三种形式:

```
#include <头文件>
```

```
#include "头文件"
```

```
#include 宏标识符
```

尖括号为实际词法符号,不是元符号。

第三种形式假定跟在 #include 后的第一个非空白字符既不是<,也不是",而且假定存在宏定义,并已经把宏标识符扩充为有效的头文件名,则词法符号序列的形式为<头文件名

>,或者为“头文件名”。

第一种和第二种形式不需要进行宏扩展,换句话说,头文件名在预处理中不作处理。头文件名必须为一个有效的带有扩展名的 DOS 文件名(习惯上头文件的扩展名为.h),并可带有路径名和路径分隔符。

预处理程序去掉#include行,并且在源代码的#include行处放入头文件的整个正文。源代码本身不变,但编译程序是分析扩大后的正文,因此#include的位置可能会影响包含文件中标识符的作用域和生存期。

如果头文件名中存在明显的路径,则搜索头文件的算法不一样,下面将分别介绍它们。

12.3.1 <头名>形式的头文件搜索

<头名>指定一个标准包含文件,按环境变量中包含目录定义的顺序连续搜索每个包含目录。若头文件在指定的目录中没有找到,则发出一个错误信息。

12.3.2 “头名”形式的头文件搜索

“头名”指定一个用户提供的包含文件,首先在当前目录中寻找文件(通常是存放源文件的目录)。若没有找到,则继续在包含目录里查找,这和<头名>情况一样。

下例说明了这些不同:

```
#include<stdio.h>

/* 标准包含目录中的头文件 */

#define myinclude "c\tc\include\mystuff.h"

/* 注意:这里单反斜杠是正确的,在C里应写成
   "c:\\tc\\include\\mystuff.h" */

#include myinclude /* 宏扩展 */
#include "myinclude" /* 没有宏扩展 */
```

在扩展后,第二个#include语句使预处理程序查找C:\TC\include\MYSTUFF.H,而第三个#include在当前目录里寻找myinclude文件,然后查找指定的目录。

12.4 条件编译

Turbo C 支持条件编译,它可以忽略某些以#开头的行(除了#if、#ifdef、#ifndef、#elif和#endif指令之外)以及由于条件编译指令的结果而不用参加编译的行。

12.4.1 #if、#elif、#else和#endif条件指令

条件指令#if、#elif、#else和#endif的工作过程与通常的C条件操作符一样。它们的用法如下:

```
#if 常量表达式 1
    <节 1>
<#elif 常量表达式 2 换行 节 2>
...
#endif
```

```
<#elif 常量表达式 n 换行 节 n>
```

```
<#else 最后一节>
```

```
...
```

如果常量表达式 1(可进行宏扩展)的计算结果为非零值(真),则节 1 表示的代码(可能为空),不论是预处理程序命令还是正常语句,都要进行预处理,接着再传给 TurboC 编译器。否则,若常量表达式 1 计算值为零(假),则忽略节 1(不进行宏扩展和编译)。

在为真的情况下,处理节 1 后,控制传给下一个 #elif 行(如果有的话),这时计算常量表达式 2。如果为真,处理节 2,此后将控制转到匹配的 #endif。否则,如果常量表达式 2 为假,控制再传给下一个 #elif,这样循环往复,直到到达了 #else 或 #endif。#else 可作为所有上面测试都为假时的备选条件,#endif 结束条件序列。

处理节也可含有条件子句,它可嵌套到任意深度,每个 #if 必须有一个相匹配的 #endif。

12.4.2 defined 运算符

defined 操作符提供了另一种测试标识符是否已定义的灵活方法,它仅在 #if 和 #elif 表达式中才有效。

若标识符定义(用 #define)后,没有消除定义(用 #undef),则表达式 defined(标识符)或 defined 标识符(括号可选)的值为 1,否则它的值为 0。因此指令

```
#if defined(mysym)
```

与

```
#ifdef mysym
```

相同,其优点是可在 #if 指令中使用 defined 构造复杂的逻辑表达式,如

```
#if defined(mysym)&&! defined(your sym)
```

12.4.3 #ifdef 和 #ifndef 条件指令

#ifdef 和 #ifndef 指令测试一标识符当前是否已定义,即测试其是否用 #define 指令定义过,行

```
#ifdef identifier
```

如果 identifier 当前已定义则与

```
#if 1
```

效果完全一样,若 identifier 当前未定义,则与

```
#if 0
```

一样。

#ifndef 测试标识符“未定义”条件是否为真,故

```
#ifndef identifier
```

当 identifier 已定义则与

```
#if 0
```

完全一样。若 identifier 未定义,则与

```
#if 1
```

完全一样。除此之外,其语法与上一节的 `#if`、`#elif`、`#else` 和 `#endif` 一样。

`NULL` 也可作为标识符加以定义。

12.5 #line 行控制指令

可使用 `#line` 指令为交叉引用或错误报告提供所在源程序的行号。如果程序是由其它程序文件派生而来的节组成的,则用原始源代码的行号而不是用由组合程序派生而来的正常顺序行号来标识这些节。其语法为

```
#line integer_constant "filename"
```

表示下一行来自 `filename` 文件中行号为 `integer_constant` 的行。文件名可以继承,以后与该文件有关的 `#line` 指令都可以忽略显式的文件名参数。这中间要注意文件名不能被改变。例如

```
/* TEMP.C: #line 指令举例 */
#include<stdio.h>
#line 4 "junk.c"
void main()
{
    printf("in line %d of %s, _LINE_, _FILE_)",
        #line 12 "temp.c"
        printf("\n");
        printf("in line %d of %s", _LINE_, _FILE_);
        #line 8
        printf("\n");
        printf("in line %d of %s", _LINE_, _FILE_);
    }
}
```

如果通过 CPP 运行 `TEMP.C`(CPP temp),则得到输出文件 `TEMP.1`,它的内容如下:

```
temp.c 1:
c:\borland\tc\cpp\include\stdio.h 1:
c:\borland\tc\cpp\include\stdio.h 2:
c:\borland\tc\cpp\include\stdio.h 3:
.....
c:\borland\tc\cpp\include\stdio.h 212
c:\borland\tc\cpp\include\stdio.h 213
temp.c 2:
temp.c 3:
junk.c 4: void main()
junk.c 5: {
junk.c 6: printf("in line %d of %s", 6, "junk.c");
junk.c 7:
temp.c 12: printf("\n");
temp.c 13: printf("in line %d of %s", 13, "temp.c"); temp.c 14:
```

```
temp.c 8,printf("\n");
temp.c 9,printf("in line %d of %s", 9, "temp.c");
temp.c 10;}
temp.c 11;
```

如果再编译 TEMP.C 则将得到输出如下:

```
in line 6 of junk.c
in line 3 of temp.c
in line 9 of temp.c
```

象 #include 指令一样,在 #line 指令的参数里的宏也要进行扩展。

#line 指令主要是用在将 C 语言代码作为输出的程序里,一般在手工编写的代码中不需要它。

12.6 #error 指令

#error 指令的语法如下:

```
#error errmsg
```

将产生信息:

```
error;filename line#:Error directive;errmsg
```

本指令通常放在预处理程序条件中,以便捕捉一些不可预料的编译条件。在正常情况下,条件是假的。若条件为真,编译将打印一个错误信息,并停止编译。要做到这一点,可在对不可预料的情况为真的条件里放一个 #error 指令。

例如,假定 #define MYVAL, 它必须为 0 或 1,这时在程序里可以有一段用来测试 MYVAL 不正确值的代码:

```
#if(MYVAL != 0 && MYVAL != 1)
#error MYVAL must be defined to either 0 or 1
#endif
```

12.7 #pragma 指令

#pragma 指令允许实现有关的指令:

```
#pragma 指令名
```

利用 #pragma, Turbo C 可定义它所期望的任何指令,而不会影响支持 #pragma 的其它编译器的使用。如果编译器不认识指令名,则它将忽略这条 #pragma 指令,而不发出任何错误或警告信息。

Turbo C 支持以下 #pragma 指令:

```
#pragma argsused
#pragma exit
#pragma inline
#pragma option
#pragma saveregs
```

```
#pragma startup
#pragma warn
```

12.7.1 #pragma argsused

argsused 指令仅允许出现在函数定义之间,且仅影响下一个函数,使下述警告信息被禁止:

"Parameter name is never used in function func-name"

12.7.2 #pragma exit 与 #pragma startup

这两个指令允许程序指定在程序启动点(在调用 main 之前)或程序退出点(通过 _exit 退出程序之前)应调用的函数。

其语法如下:

```
#pragma exit    函数名<优先级>
#pragma startup 函数名<优先级>
```

指定的函数名必须是一个已定义的函数,它不带参数,且返回 void,即它应说明为:

```
void func(void)
```

优先级应在范围 64~255 之间,最高优先级为 0(0~63 之间的优先级由 C 库函数所用,用户不能使用它们)。具有较高优先级的函数在启动时首先调用,在退出时最后调用。若未指定优先级,则缺省值为 100。例如

```
#include <stdio.h>
{
    printf("Startup function. \n");
}
#pragma startup starFunc 64
/* priority 64-->called first at startup */
void exitFunc(void)
{
    printf("Wrapping up execution. \n");
}
#pragma exit exitFunc
/* default priority is 100 */
void main(void)
{
    printf("This is main. \n");
}
```

注意在 #pragma startup 或 exit 中的函数必须定义(或说明)在该指令之前。

12.7.3 #pragma inline

本指令等价于 -B 命令行编译选择项或相应的集成环境选择项。它告诉编译程序,在程序里存在内部汇编语言代码。其语法为

```
#pragma inline
```

它最好放在文件头部,因为当编译器遇到 `#pragma inline` 时,将用 `-B` 选择项重新启动自己。事实上,可省去 `-B`,选择和 `#pragma inline` 指令,编译程序一旦遇到 `asm` 语句,将以某种方式重新启动自己。`-B` 选择项和本指令的目的是为了节省一些编译时间。

12.7.4 `#pragma option`

利用 `#pragma option` 可在程序里包含命令行选择项,其语法为

`#pragma option[选择项...]`

选择项可以是任何命令行选择项(除了下一段列出的之外)。一条指令可包含多个选择项。

不能出现在 `pragma option` 里的选择项包括:

- `-B` (用汇编程序编译)
- `-C` (编译但不连接)
- `-dxxx` (定义一个宏)
- `-Dxxx=ccc` (定义一个带正文的宏)
- `-efff` (命名. EXE 文件 `fff`)
- `-lfff` (命名包含目录)
- `-Lfff` (命名库目录)
- `-lxset` (连接程序选择项 `x`)
- `-M` (在连接时创建. MAP 文件)
- `-o overlays`
- `-Q EMS`
- `-S` (创建. ASM 输出后就停止编译)
- `-Uxxx` (取消一宏定义)
- `-V` (虚拟)
- `-Y` (覆盖)

编译可分为两个状态,在 `#pragma option` 里包含的第一个状态的选择项可比第二状态多。第一状态仅进行词法分析,第二状态产生代码。

在 `#if`, `#ifdef`, `#ifndef` 或 `#elif` 指令里若用了一个以两个下划线开头的内部宏,将使编译程序进入代码生成状态。

一旦出现了实际的词法符号(C 说明),也将使编译程序进入代码生成状态。

换句话说,在仅进行词法分析期间,可使用 `#pragma`、`#include`、`#define` 和一些 `#if` 指令,这时可使用 `#pragma inline` 来改变命令行选择项。

仅进行词法分析阶段时在 `#pragma option` 中出现的选项包括:

- `-Efff` (汇编程序名串)
- `-f*` (除 `-ff` 之外的任何浮点选择项)
- `-i#` (有效的标识符字符)
- `-m*` (任何存储模式选择项)
- `-nddd` (输出目录)
- `-offf` (输出文件命名 `fff`)
- `-u` (在 `cdecl` 名中使用下划线)

-z (任何段名选择项)

其它选择项可在任何地方改变。下面的选择项如果在函数或对象说明之间进行了修改,则仅影响编译器:

- 1 指令集控制
- 2 指令集控制
- a 对齐控制(注意结构成员的对齐在结构定义时确定,而不在后面的对象使用该结构时确定)
- ff 快速浮点控制
- G 快速生成代码
- k 标准栈框架控制
- N 栈溢出检查
- O 优先控制
- p Pascal 调用约定
- r 和 rd 寄存器变量控制
- v 冗长的调试控制
- y 行信息控制

下面的选择项可在任何时候改变,并且会立即产生作用:

- A 键盘控制
- C 嵌套的注释
- d 合并重复串
- gn 在 n 个警告后停止编译
- jn 在 n 个错误后停止编译
- K char 类型为 unsigned
- wxxx 警告(与 #pragma warn 相同)

任何开关选择项(如-a 或-K)可象在命令行中一样打开或关闭。它们其后跟一个点号(.)表示恢复选择项为命令行状态。

12.7.5 #pragma saveregs

saveregs 指令保证调用 huge 函数时不会改变任何寄存器的值。如果使用汇编语言代码,就可能会用到本指令,它应放在某个函数定义之前,并只适用于这个函数。

12.7.6 #pragma warn

warn 指令使特定的-xxx 命令行选择项(或 options|compiler|messages 对话框中的 DisplayWarnings)无效。例如,若源代码含有指令:

```
#pragma warn +xxx
#pragma warn -yyy
#pragma warn .zz
```

则将打开 xxx 警告(即使在 options|Compiler|messages 菜单上它为关闭),关闭 yyy 警告,而 zzz 警告将恢复为开始编译时的值。

有关三个字母的编写及其警告请参阅附录 B 有关命令行编译器的内容。

12.8 预定义的宏

Turbo C 预定义了以下全局标识符。除了 `_cplusplus` 外,其它标识符都以两个下划线开头和结束,这些宏也称为显式常量(manifest constants)。

1. `_CDECL`

该宏专用于 Turbo C,它表示不使用 `-p` 选择项(Calling convention...C);若未用 `-p` 选择项,则设置它为 1,否则未定义。

下面 6 个符号根据编译时选取的存储模式而定义:

```
_COMPACT_      _MEDIUM_
_HUGE_          _SHALL_
_LARGE_         _TINY_
```

对任何给定编译,仅有一个有定义,其它都无定义。例如,若用小模式编译,则 `_SMALL_` 有定义,其余的都无定义,因而指令

```
#if defined(_SMALL_)
```

将为真,而

```
#if defined(_HUGE_)
```

(或其它)将为假。这些已定义的宏的实际值为 1。

2. `_DATE`

本宏提供预处理程序开始处理当前源文件(当作一个串文字量)的日期。在一给定文件里的每个 `_DATE` 都给出同一值,而不管处理需要多长时间。日期表示为 `mmm dd yyyy`,这里 `mmm` 表示月份(Jan, Feb 等),`dd` 表示号数(1~31),若小于 10,则 `dd` 的第一个字符为空,`yyyy` 表示年份(1990, 1991 等等)。

3. `_FILE`

本宏提供当前正处理的源文件名(以串文字量形式)。每当编译程序处理 `#include` 指令或 `#line` 指令,或者正在处理包含文件时,本宏都要改变。

4. `_LINE`

本宏提供当前正处理的源文件行号(以十进制常量形式)。通常,源文件的第一行定义为 1,通过 `#line` 指令可改变它。

5. `_MSDOS`

本宏专用于 Turbo C,它对所有编译提供整常量 1。

6. `_OVERLAY`

本宏专用于 Turbo C,若用 `-Y` 选择项编译一个模块(激活覆盖支持),则它预定义为 1。若不激活覆盖支持,则它无定义。

7. `_PASCAL`

本宏专用于 Turbo C,它标志是否用了 `-P` 选择项。若用了 `-P`,则它置为 1,否则它无定义。

8. `STDC`

若用 ANSI 兼容标志(A 或 ANSI KEYboards Only...ON)进行编译,则该宏定义为常量 1,否则它无定义。

9. `__TIME__`

本宏记录预处理程序开始处理当前源文件的时间(以串文字量形式)。如 `__DATE__` 一样,每个 `__TIME__` 的值对同一文件都相同,而不管处理需要多长时间。它的格式为 `hh:mm:ss`,其中 `hh` 表示小时(00~23),`mm` 表示分钟(00~59),`ss` 表示秒数(00~59)。

10. `__TURBOC__`

本宏专用于 Turbo C,它给出了当前 TurboC 的版本号,其形式为 16 进制常量。例如,版本 1.0 表示为 `0x0100`。

第三部分

高级 C 程序设计技巧

文件输入输出

屏幕文本和图形程序设计

存储模式

与汇编语言的接口

第十三章

文件输入输出

无论是在 C 语言还是其它语言中,文件输入输出(I/O)操作都是一个很重要的组成部分。在本章中,我们将首先介绍在传统的 Turbo C 语言中是如何进行 I/O 操作的,在此基础上接着它们的使用方法。

尽管 C 语言中的输入和输出贯穿了库函数使用的全过程,但是在 C 语言中并没有代表 I/O 操作的关键字。ANSI 标准在 Turbo C 之后定义了一个 I/O 函数的完备集。不过,旧的 UNIX 标准包含了处理 I/O 操作的两个不同系统。第一种方法叫做 ANSI 文件系统,是由 ANSI 标准和 UNIX C 共同制定的(也叫做格式化或缓冲型系统),第二种为 UNIX 型文件系统,它是由 UNIX 单独定义的(也叫做非格式化或非缓冲型系统)。

ANSI 标准不定义 UNIX 型文件系统是有许多理由的,况且定义两组文件系统未免也显得有些多余,但考虑到目前这两种标准的应用都很广泛,Turbo C 对这两种标准均予以支持。本节将讨论这两部分内容,但相对来说将侧重于 ANSI 标准的 I/O 系统。这是因为使用 UNIX 型文件系统的用户已大为减少,因此我们建议新程序最好按照 ANSI 的 I/O 函数来编写。

本节将讲述 C 的 I/O 操作概况及上述两组文件系统大多数函数的工作情况。注意,在这里我们只讨论了 C 函数库各种各样非常丰富的 I/O 函数中的一部分,其余部分可以从用户手册上学到。

在开始学习 C 语言的 I/O 系统之前,用户首先需要了解两个特殊的预处理指令和一些术语。

13.1 两个预处理指令

在 C 语言的源程序中可能包含着各种各样的 C 编译指令,通常称之为预处理指令,它们并不是 C 语言本身的内容,但可以用来扩展 C 语言程序的编程环境。所有的预处理指令都以符号 # 开头。

13.1.1 #define 指令

由于在后面 I/O 系统用到的头文件中,我们将会遇到各种各样的常数。因此,在这里我们首先介绍一下 #define 指令。#define 指令用来定义一个标识符和一个字符串,在程序中每次遇到该标识符时,编译程序都用所定义的字符串替换它。这个标识符叫做宏替换名。替换过程叫做宏替换。宏定义指令 #define 的一般形式为

#define 宏替换名 字符串

注意在这里没有分号。字符串可以是任何字符序列,字符串不需要用引号括起来。在标识符和字符串之间可以有任意多个空格,但是在字符串结束后一定要换行。例如,用户想用 TRUE 表示数值 1,用 FALSE 表示数值 0,则可以用下面两个 #define 宏定义说明:

```
#define TRUE 1
#define FALSE 0
```

这样在编译源程序时,每当遇到 TRUE 或 FALSE,编译程序就自动用 1 或 0 替代。例如,下述语句在屏幕上显示“0 1 2 :”:

```
printf(" %d %d %d :", FALSE, TRUE, TRUE+1)
```

对于已定义的宏替换名,用户可在以后的宏定义中进行引用。例如,下述语句定义了 ONE、TWO 和 THREE 所代表的值:

```
#define ONE 1
#define TWO ONE+ONE
#define THREE ONE+TWO
```

用户一定要注意,宏替换只是简单地用所说明的字符串来替换对应的标识符。例如,如果想定义一个标准出错信息,则用户可以像下面这样书写:

```
#define E_MS "standard error on input\n"
...
printf(E_MS);
```

编译程序只是在遇到了标识符 E_MS 时才用“standard error on input\n”来替换。对编译程序来说,语句 printf(E_MS)的实际内容其实是

```
printf("standard error on input\n");
```

但是,当宏替换名出现在某一字符串中时,则编译程序不进行任何替换。例如下列程序

```
#define XYZ this is a test
...
printf("XYZ");
```

的实际输出内容并不是“this is a text”,而是“XYZ”。

#define 的常见用法是定义一些事物的大小,例如一个数组的大小,所定义的常量会随着程序的变化而变化。在下面的简单例子里,宏 MAX_SIZE 用于定义一整型数组的大小,并且还控制了 for 循环的循环条件。

```
#include <stdio.h>
#define MAX_SIZE 16
unsigned int pwr_of_two[MAX_SIZE];
/* display powers of 2. */
main(void)
{
    int i;
    pwr_of_two[0]=1; /* start the sequence */
    for(i=1; i<MAX_SIZE; i++)
        pwr_of_two[i] = pwr_of_two[i-1] * 2;
    printf("The first 16 powers of 2:\n");
```

```
for(i=0; i<MAX_SIZE; i++)  
    printf(" %u", pws_of_two[i]);  
return 0;  
}
```

13.1.2 #include 指令

#include 预处理指令直接指示编译程序将该指令所指明的另一个源文件嵌入到 #include 指令所在的程序。文件名应使用双引号或尖括号括起来。下面两种形式都能指示编译程序读取和编译文件名为 test 的文件。

```
#include "TEST"  
#include <TEST>
```

在被包含文件中仍可以含有 #include 指令,这叫做嵌套包含。

如果在文件名标识符中给出了文件的明确路径,那么编译时将只在指定的目录中查找包含文件;如果文件名包括在双引号内,且只包含了该文件的文件名而无路径,则编译程序将首先在当前工作目录中查找该文件。如果该文件没找到则在由命令行中的 -I 选择项所指定的目录中继续搜索,如果最后仍未找到这个文件,就在由环境工具所指定的标准目录中查找。

如果包含文件在尖括号内且没给出明确的路径名,那么编译程序将首先在编译命令 -I 选择项所指定的目录中查找。如果文件未找到,则搜索标准目录,而不在当前工作目录找这个文件。

如果在 Borland 的指导下安装好了 Turbo C,那么应选用尖括号,本书将采用这种方法。

13.2 文件与流

在讨论 C 的 I/O 系统之前,有必要先搞清楚“流”和“文件”这两个术语的区别。C 语言 I/O 系统为 C 语言编程人员提供了一个统一的接口,使得编程能够尽可能与具体的访问设备无关,也就是说,C 语言 I/O 系统在编程者和被使用的设备之间提供了一抽象的概念,这个抽象的东西我们称之为“流”。具体的实际设备叫“文件”。在流与文件二者之间具有紧密的内在联系。

13.3 流(streams)

ANSI C 文件系统能够支持各种不同设备,包括终端、磁盘驱动器和磁带机等。虽然各种设备差别很大,但由于 ANSI C 文件系统把每个设备都转换为一个称之为流的逻辑设备,因为它能够完成有关这些设备的所有操作。由于所有的流都有相同的行为,所以流在很大程度上与设备无关,因此,即使是一个用来进行磁盘文件写入操作的函数,也可以用来进行控制台写入。流有两种类型:文件流(text stream)和二进制流(binary stream)。

13.3.1 文本流

文本流是一连串的字符。在文本流中,某些字符的变换由环境工具的需要来决定。例如,

一个换行符可以变换为回车换行,这是 Turbo C 的工作方式。因此,所读写的字符与外部设备之间并不存在一一对应关系,并且所读写的字符个数也可以和在外部设备中的个数不同。

13.3.2 二进制流

一个二进制流是由与外部设备中的内容一一对应的一连串字节组成的。使用中设有字符翻译过程,所读写的字节数目也与外设中的数目相同。不过,在一个二进制流末可以加一些空字节使之占满磁盘的一扇区。

13.3.3 文 件

在 C 语言中,“文件”是一个逻辑概念,可以用来表示从磁盘文件到终端的所有信息。用户可以通过一个文件打开操作使流和一个特定的文件发生联系。一旦已经打开一个文件,用户程序就可以实现与该文件之间的信息交换。

C 语言 I/O 系统认为所有的流都是相同的,而文件则不然。并不是所有的文件都有相同的功能,例如一个磁盘文件可以允许随意存取,但一个终端就不行。

如果一个文件支持随机存取,则在打开该文件时将先把文件指针设置到它的开头处。每当从该文件中读取或写入一个字符后,该文件指针将会增加,以保证整个文件的读写顺序。

关闭操作将使文件脱离一个特定的流。对于一个为输出而打开的流,关闭这个流时程序将会把与这个流有关的缓冲区内容写入到外部设备中。这个过程一般叫做“刷新”这个流,保证了没有残存信息偶尔仍留在缓冲区内。当用户程序正常结束时,所有的文件都将自动关闭。

每一个与文件相结合的流有都一个 FILE 型的文件结构。这个结构在头文件 `stdio.h` 中定义,用户程序不能对这个文件控制块进行改动(结构是一种归属于同一名字的各种变量的数据类型,与 Turbo Pascal 中的 RECORD 相似)。在使用 I/O 程序时不懂得任何有关结构的知识并不会有多大影响。

13.4 概念和实际

下面,让我们来概述一下在 C 语言中 I/O 系统的操作方法。所有的 I/O 操作都是通过流来进行。所有的流都是由一系列字符组成,因此所有的流都一样。流与文件的连接则可以通过文件系统来实现。由于各个设备具有不同的功能,所有文件并不都一样(但是这种差别对于编程人员来说微乎其微)。C 语言的 I/O 系统可实现设备与流之间的信息双向转换。除了需要了解哪类文件可以随机存取外,用户只需针对“流”对应的逻辑设备去考虑编程就可以了。

如果对这种方法感到奇怪或含糊不清的话,用户可以参考一下 BASIC 或 FORTRAN 语言的内容,其中系统所支持的各个设备均有各自完整的独立系统。在 C 语言中,编程者只需要考虑流这个概念并且只使用一个文件系统就可以完成所有的 I/O 操作。

13.5 控制台 I/O

控制台 I/O 是指发生在计算机键盘和显示器上的操作。通常控制台 I/O 是由 ANSI 文件系统的一个专门子系统来完成的(如果再加入一些 Turbo C 的特殊函数将能够更好地支持计算机和用户之间的交互作用)。包括 DOS 在内的很多操作系统中,控制台 I/O 可以重新定向到别的设备。在一般情况下,控制台通常是指最常用的设备。有关控制台 I/O 函数与其它文件系统函数的连接方法将在后面涉及到。

13.5.1 字符读写

最简单的控制台 I/O 函数是用于从键盘读入一个字符的 `getche()` 函数和把一个字符显示到屏幕上的 `putchar()` 函数。函数 `getche()` 等待从键盘上键入一个字符,返回它的值并且在屏幕上自动回显该字符。函数 `putchar()` 把它的字符参数显示在光标的当前位置上。下面是函数 `getche()` 和 `putchar()` 的原型:

```
int getche(void);
int putchar(int c);
```

`getche()` 返回一个整数,其低位字节包含所输入的字符。实际上,`putchar()` 输出到屏幕上也只有低位字节。`getche()` 包含在头文件 `CONIO.H` 中,`putchar()` 则包含在头文件 `STDIO.H` 中。

下面的程序从键盘读入一些字符并显示它们经大小写变换后的形式,即将大写字符改小写。小写字符改大写,在打入句号后程序将停止运行。库函数 `islower()` 包含在头文件 `CTYPE.H` 中,该函数当其参数为小写时返回真,否则为假。

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
main(void)
{
    char ch;
    printf("enter chars, enter a period to stop\n");
    do {
        ch = getche();
        if (islower(ch)) putchar(toupper(ch));
        else putchar(tolower(ch));
    } while (ch != '.'); /* use a period to stop */
    return 0;
}
```

`getche()` 函数有两个重要的变体 `getchar()` 和 `getch()`。`getchar()` 是 UNIX 字符输入函数的原型。由于最初的 UNIX 系统的行缓冲区设计成必须接收回车符后才能把键入的字符送往程序,所以此函数只有当键入一个回车符后才将输入数据返回给系统。这样可能和环境很不协调,所以我们建议不要用这个函数。为保证 Turbo C 对 UNIX 系统的程序移植性,所以

用户需要对此函数有一定的了解。函数 `getchar()` 包含在头文件 `STDIO.H` 中。

`getch()` 的作用与 `getche()` 基本一致,只是不把读入的字符回显到屏幕上。它包含在头文件 `CONIO.H` 中。由于 `getchar()` 的缺点,Turbo C 采用以上两变体,但 `getche()` 和 `getch()` 函数都不是由 ANSI 标准所定义的。其它多数基于 PC 机的 C 语言也一样。

13.5.2 字符串读写

`gets()` 和 `puts()` 是在复杂性和功能性方面同上述函数相比都较为优越的函数。它们用来在控制台中读写字符串。

`gets()` 函数用来从键盘读入一字符串,并把它们送到 `gets()` 函数中字符型指针变量所指定的地址。每次可从键盘上键入多个字符并以回车符结束,在实际输入的串中则以一个空终结符来替换回车符。虽然 `getchar()` 可以用来返回一个回车符,但用 `gets()` 是不可能的,字符打错时可以用光标左移键在回车之前更改。`gets()` 函数的原型是:

```
char *gets(char *s);
```

其中 `s` 是一个字符数组,用来接收用户输入的字符,原型在 `STDIO.H` 中。下面的程序读入一个字符串到 `str` 数组中并显示字符串的长度:

```
#include <stdio.h>
#include <string.h>
main(void)
{
    char str[80];
    gets(str);
    printf("length is %d",strlen(str));
    return 0;
}
```

`gets()` 函数返回一个指针给 `s`。

函数 `puts()` 把它的字符串参数回显到屏幕上并换行。它的原型为:

```
int puts(char *s);
```

在 `puts()` 函数中可以使用与 `printf()` 一样的转义控制符,如 `\t` 表示水平制表符 TAB。使用 `puts()` 函数同使用 `printf()` 函数相比,带来的冗余操作要少得多,且仅用来输出字符串,而不能输出数组和进行格式变换,因此函数 `puts()` 比 `printf()` 所占内存小,执行速度也更快。`puts()` 经常用于需要很高优化的代码中。函数 `puts()` 返回一个指向这个字符串的指针,返回成功时为非负值,否则为 EOF。该函数包含在 `STDIO.H` 头文件中。下面这个语句将把字符串 `hello` 写在屏幕上。

```
puts("hello");
```

表 13.1 中列出了用于进行控制台 I/O 操作的最简单的函数。

13.6 控制台格式化 I/O

除了上述的控制台 I/O 函数外,C 标准库还包含两个用来对内部类型数据进行格式化输入输出的函数:`printf()` 和 `scanf()`。格式化是指在用户程序控制下,函数用各种不同的格

式来读写数据的过程。`printf()`函数用来向控制台输出数据;`scanf()`函数则用来从键盘读入数据。`printf()`和`scanf()`函数可以对任何一种类型的内部数据,包括字符、字符串和数字进行操作。

表 13.1 基本控制台 I/O 函数

函数	操 作
<code>getchar()</code>	从键盘读入一个字符,回车返回
<code>getche()</code>	从键盘读入一个字符并回显,不用回车返回
<code>getch()</code>	从键盘读入一个字符不回显,不用回车返回
<code>putchar()</code>	向屏幕写一个字符
<code>gets()</code>	从键盘读入一个字符串
<code>puts()</code>	向屏幕写一个字符串

13.6.1 printf()函数

`printf()`函数的原型是:

```
int printf(char *控制字符串,...);
```

`printf()`的原型在 `STDIO.H` 中。在 `printf()` 原型末尾的三个点,代表函数能够接受数目可变的参数。

控制字符串由两种不同类型的内容组成。一是要显示在屏幕上的字符,二是用来定义参数显示格式的格式化命令。一个格式化命令的开头带有一个百分号(%),后面跟一个格式码。格式化命令见表 13.2。参数的个数与格式化命令所说明的个数必须在数量和顺序上一一对应。例如,调用以下 `printf()` 语句:

```
printf("Hi %c %d %s", 'c', 10, "there!");
```

将显示:Hi c 10 there!

表 13.2 printf()格式化命令

说明符	格式意义
<code>%c</code>	单个字符
<code>%d</code>	十进制数
<code>%i</code>	十进制数
<code>%e, %E</code>	科学计数
<code>%f</code>	浮点十进制数
<code>%g, %G</code>	使用 <code>%e</code> 或 <code>%f</code> 表达式较短者
<code>%o</code>	八进制数
<code>%s</code>	字符串
<code>%u</code>	无符号十进制数
<code>%x</code>	十六进制数
<code>%%</code>	显示百分号%
<code>%p</code>	显示一个指针
<code>%n</code>	变量应是一整型指针,其中存放已写字符的个数

在一个格式说明中还可以带有用以确定显示宽度、小数位数及左端对齐等的修饰符。在

百分号和类型字符格式码可以写入一个说明最小宽度的整数项。这个插入项使输出带有若干空格或者 0 以保证所说明的最小宽度。如果字符串的长度或数值大小超过了说明宽度,则输出内容将按其实际长度显示。如果想在显示值前加一些 0,就在宽度项前加上一个 0。例如 %05d 将在显示一个小于 5 位的数值时在数值的前面补 0,使其宽度保持 5 位。如果不在宽度项前加 0 则补入空格。

可将一个小数点放在宽度修饰符之后确定浮点数小数点的位置,具体位置由读者自由决定。例如, %10.4f 显示数值时宽度至少 10 位,并带有 4 位小数。在输出字符或整型量的这种格式化命令中,小数后的数字代表最大宽度。例如 %5.7s 将显示一个长度不小于 5 且不超过 7 个的字符串。若字符串长度大于 7,则第 7 个以后的字符被删除。

在显示宽度大于所显示数据的位数时,说明或缺省的输出为右对齐格式。在 % 后面加一个负号可以将显示格式说明为左端对齐。例如 %-10.2f 就用左对齐方式把一个二位小数的浮点数显示在 10 个字符宽的区域內。

printf() 函数有两个用于显示 short 和 long 整型数格式化命令修饰符,即 l 和 h。修饰符 l 表示其对应参数是长整型数。例如 %ld 是说明要显示长整型。修饰符 h 表示其对应参数是短整型量,因此 %hu 说明数据类型是 short unsigned int。

修饰符 l 和 h 可用于 d, i, o, u 和 x 类型说明中。修饰符 l 还可以用在浮点数说明 %e, %f 和 %g 中,说明要显示的数据为 double 型。修饰符 L 指明输出数据的数据类型为 long double 型。

图 13.1 给出了一些简单的例子。

printf() 语 句	输 出
("%-5.2f", 123.234)	123.23
("%5.2f", 3.234)	3.23
("%10s", "hello")	hello
("%-10s", "hello")	hello
("%5.7s", "123456789")	1234567

图 13.1 printf() 的一些简单例子

13.6.2 scanf() 函数

scanf() 是一个控制台的常规输入函数,该函数能够读取各种内部类型的数据,并自动将它转化为预先指定的格式。它的功能正好与 printf() 函数相反。scanf() 的一般形式为:

```
int scanf(char * 控制字符串, ...)
```

scanf() 的原型在 STDIO.H 中。控制字符串可包括:

- 格式说明码(format specifiers)。
- 空白字符(white _space characters)。
- 非空白字符(non _white _space characters)。

在输入格式说明码前用一个百分号通知函数 scanf() 下一个将读入的数据类型。具体格式说明代码列于表 13.3 中。

表 13.3 scanf()格式说明码

说明码	含 义
%c	读入一个字符
%d	读入一个十进制整型数
%i	读入一个十进制整数
%E	读入一个浮点数
%f	读入一个浮点数
%h	读入一个短整型数
%o	读入一个八进制数
%s	读入一个字符串
%x	读入一个十六进制数
%p	读入一个指针
%n	接受一个整型数,其值为已读入的字符个数

在格式控制字符中的一个空白字符会使 scanf()函数在读操作中跳过输入流中的一个或多个空白字符。空白字符可以是空格、制表符\t或换行符\n。控制字符串中的空白字符使 scanf()函数只读入而不存储所有的空白字符,只有读入的非空白字符(包括0在内)才被存储。

在控制字符串中的一个非空白字符会使 scanf()函数在读入时剔除与这个非空白字符相符的字符。例如"%d,%d"使 scanf 先去读入一个整型数,然后把接着读入的逗号剔除,最后再读入另外一个整型数。如果这一特定的字符未找到,scanf()函数就终止。

scanf()函数中所有用来接收数值的变量前必须加取地址操作符&,即参数表中列出的必须是指向这些变量的指针。C语言的这种参量调用法,允许函数对它的参数内容进行修改。例如,如果要读入一个整数并放在变量count中,可以调用下面形式的scanf()函数:

```
scanf("%d",&count);
```

字符串被读入到一个字符数组中时,不带下标的数组名就是数组第一个元素的地址。要读入一个字符串并放到一个字符数组address中时,可以这样写:

```
scanf("%s",address);
```

这里的address是一个指针,所以不需要在它的前面加&操作符。

输入数据项必须由空格、制表符或回车符分隔。标点符号,如逗号、引号都不能作分隔符。例如:

```
scanf("%d %d",&r,&c);
```

可以接受10 20这种形式的输入,而不能接受10,20。与printf()一样,scanf()中的各格式说明码在个数和顺序上必须与参数表中的变量一一对应。

在百分号和格式码之间写入一个星号*时,函数将只读入数据但不赋值。例如,在输入10/20时,则把10赋给X,忽略除号/并把20赋给Y。

```
scanf("%d%*c%d",&x,&y);
```

格式化命令用修饰项可以说明最大读入长度,即用户可以在%和格式码之间写入一个整型数,表示任何读操作中最大字符个数。例如希望读入str的字符不超过20个,可以象下

面这样写;如果输入流大于最大字符长度 n , `scanf()` 就不再读入字符了,后边一个读入函数将从第 $n+1$ 个字符开始读。输入字符的个数可以小于说明的最大宽度,一旦读入的字符串中有空格, `scanf()` 就认为字段已被读完,并去读第二个字段。例如执行下语句

```
scanf("%20s",str);
```

键入:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

`scanf()` 中因为限制了最大长度,只有 A~T20 个字符被读入 `str`。剩下的 U~Z 这 6 个字符没有用到。如果再另外调用一次 `scanf()`,例如:

```
scanf("%s",str);
```

则字符 U~Z 被放入 `str`。

虽然空格、制表符和换行符都可以作为字段分隔符来使用,但在读入单个字符的时候,它们也像其他单个字符一样被读入。例如输入流为 `x, y` 时

```
scanf("%c%c%c",&a,&b,&c);
```

则返回 `x` 值给 `a`, 空格给 `b`, `y` 给 `c`。

注意:

1. 如果在控制字符串中含有任何其它字符,如空格、制表符、制表符、换行符等,就会在读入时逐一进行比较,遇到与这些相同的字符就会将其从中剔除。例如,给一个 10t,20 的输入流,

```
scanf("%st%s",&x,&y);
```

将把 10 给 `X`, 20 给 `Y`。因为控制字符中有 `t`, 所有 `t` 被剔除了。又例如,

```
scanf("%s",name);
```

直到在空白字符后键入一字符时函数才返回。因为 `%s` 后的一个空格指示 `scanf()` 函数要求输入并剔除一个空格、制表符或换行符。

2. 不能用 `scanf()` 去显示一个提示信息,因为所有的提示必须在调用 `scanf()` 之前完成。函数 `scanf()` 还包含一个功能强大的性能称为搜索集。在搜索集中定义了 `scanf()` 中特殊的字符表。只要读入字符在搜索集中, `scanf()` 将继续读字符,一旦输入了不在搜索集中的字符, `scanf()` 将转向下一个格式。搜索集是由希望搜索的字符的列表来定义的,这些字符在括号中,左方括号前一定要有一百分号 `%`。例如,下面这个搜索集告诉 `scanf()` 只读数字 0~9:

```
%[0123456789]
```

搜索集的对应参数必须是指向字符数组的指针。从 `scanf()` 返回后,数组将包含一个由读入字符组成的非空字符串。下面的程序将显示 `scanf()` 在这里是如何工作的:

```
#include <stdio.h>
main(void)
{
    char s1[80],s2[80];
    scanf("%[12344567890]%s",s1,s2);
    printf("\n%s||%s",s1,s2);
    return 0;
}
```

键入下列字符:

```
"123456789abcdefg987654" <RETURN>
```

程序将显示:

```
123456789 || abcdefg987654
```

因为 a 不是搜索集中的字符, scanf() 停止搜索并且将剩余字符送入 s2。连字号用来定义搜索集的范围。例如, 要求 scanf() 接收从 A~Z 的字符, 可以用以下控制码:

```
%[A-Z]
```

在搜索集中定义可以不只一个范围。例如, 以下程序可读入数字和字母, 这只需说明可以用搜索集定义的最大域。

```
#include <stdio.h>
main(void)
{
    char str[80];
    printf("Enter digits and letters, ");
    scanf("%78[a-zA-Z0-9]", str);
    printf("\n%s", str);
    return 0;
}
```

用户也可以定义一个相反集, 此时搜索集中的第一个字符必须是 ^。当 ^ 出现时, 它指示 scanf() 接收任何没有在搜索集中定义的字符。记住对搜索集的要求必须严格明确。因此, 如果想搜索大写和小写字母, 必须对它们分别定义。

13.7 缓冲型 I/O 系统(ANSI 型 I/O 系统)

缓冲型 I/O 系统由若干个有内在联系的函数构成。表 13.4 中列出了最常用的函数。使用这些函数时需要把头文件 STDIO.H 包含到调用这些函数的程序中。

13.7.1 文件指针

文件指针是贯穿整个缓冲型 I/O 系统的重要组成部分。一个文件指针是一个指向文件有关信息(包括文件名、状态和当前位置等)的指针。文件指针标志了一个指定的磁盘文件, 文件指针用组合流来告诉系统的每个缓冲型 I/O 函数应如何完成其操作。文件指针是一个 FILE 型指针变量, 用于用户对文件实现读写。在 STDIO.H 中定义。文件指针说明形式如下:

```
FILE *fp;
```

13.7.2 打开文件

fopen() 函数完成两个功能: 打开一个流并把一个文件与这个流连接; 返回分配给这个文件的文件指针。函数 open() 的原型如下:

```
FILE *fopen(char *filename, char *mode);
```

mode 是一个说明文件打开方式的字符串。filename 必须是一个由字符串组成的在操作系统下有效的文件名, 并允许带有路径名。

模式值的合法有效值见表 13.5。“t”代表正文文件，“b”代表二进制文件，如果不定义，文件将按照 Turbo C 的全局变量 `fmode` 打开。这个变量既可被设置为 `O_TEXT` 代表正文模式，也可以设置成 `O_BINARY` 代表二进制模式。缺省时为 `O_TEXT`。宏 `O_BINARY` 和 `O_TEXT` 可以在 `FCNTL.H` 中找到。本节中 will 把 `fmode` 设置为缺省值。

`fopen()` 返回一文件指针，用户程序不能改变这个指针值。如果打开一个文件时有错误发生，则 `fopen()` 返回一个空值 `null`。宏 `NULL` 在 `STDIO.H` 中定义。

表 13.4 最常用缓冲型文件系统函数

函数名	操 作
<code>fopen()</code>	打开一个流
<code>fclose()</code>	关闭一个流
<code>puts()</code>	向流里写一个字符
<code>gets()</code>	从流里读一个字符
<code>fseek()</code>	从流里寻找一个指定字符
<code>fprintf()</code>	像在屏幕上向流里写
<code>fscanf()</code>	像从键盘上向流里写
<code>feof()</code>	遇到文件结束指示时返回真
<code>ferror()</code>	遇到错误发生时返回真
<code>rewind()</code>	重新把文件指针设置到文件开头
<code>remove()</code>	删除一个文件

如表 13.5 所示，一个文件既可以用正文模式，也可以用二进制模式打开。在正文模式中，输入时，回车换行被认为是另起一行；输出时，则把另起一行转换为回车换行。在二进制文件中没有这样的翻译过程。

如果想打开一个名叫 `test` 的文件并准备写操作，可以用语句

```
fp = fopen("test", "w");
```

通常的处理如下：

```
FILE *fp;
if((fp = fopen("test", "w")) == NULL) {
    puts("cannot open file\n");
    exit(1);
}
```

上面这种写法可以在写文件之前先检验已打开的文件是否有错，如写保护或磁盘已满等。在这里使用 `null` 是因为文件指针永远不会是该值。调用库函数 `exit()` 将使程序立即终止，`exit()` 原型在 `STDLIB.H` 中：

```
void exit(int val);
```

`val` 的值将返回给操作系统。返回值为 0 代表成功结束操作，其它值则表示由于一些问题使程序中止。

如果用 `fopen()` 的写方式打开一个文件，则文件名下的文件内容将全部抹去，并开始存放新的内容。如果原先无此文件，则生成这个文件。如果想往文件的尾部添加内容，必须使

用操作模式 a。在以读操作打开一个文件时,文件必须存在,否则将返回一个出错信息。打开一个可读/写的文件时,如果文件存在,它不会被抹掉;如果文件不存在,则生成这个文件。

13.7.3 写字符

函数 `putc()` 可以写字符到一个已用 `fopen()` 函数打开的可进行写操作的流中。函数描述为:

```
int putc(int ch, FILE *fp);
```

这里 `fp` 是 `fopen()` 返回的文件指针, `ch` 表示输出的字符。文件指针 `fp` 指示 `putc()` 函数写字符到指定的磁盘文件中去。由于历史原因, `ch` 名义上称为 `int`, 但它只使用低位字节。

如果 `putc()` 操作成功, 就返回所写入的字符; 如果操作失败则返回 `EOF`。 `EOF` 是 `STDIO.H` 中定义的一个宏, 含义是“文件结束”。

表 13.5 `fopen()` 有效的模式值

模式	含 义
"r"	打开一个文本文件, 用于读
"w"	生成一个文本文件, 用于写
"a"	对一个文本文件, 用于添加
"rb"	打开一个二进制文件, 用于读
"wb"	生成一个二进制文件, 用于写
"ab"	对一个二进制文件, 用于添加
"r+"	打开一个文本文件, 用于读/写
"w+"	生成一个文本文件, 用于读/写
"a+"	打开或生成一个文本文件, 用于读/写
"r+b"	打开一个二进制文件, 用于读/写
"w+b"	生成一个二进制文件, 用于读/写
"a+b"	打开或生成一个二进制文件, 用于读/写
"rt"	打开一个文本文件, 用于读
"wt"	生成一个文本文件, 用于写
"at"	对一个文本文件, 用于添加
"r+t"	打开一个文本文件, 用于读/写
"w+t"	生成一个文本文件, 用于读/写
"a+t"	打开或生成一个文本文件, 用于读/写

13.7.4 读字符

函数 `getc()` 从一个已由 `fopen()` 函数打开, 并可执行读操作的流中读取字符。函数描述为:

```
int getc(FILE *fp);
```

这里 `fp` 是一个由 `fopen()` 返回的 `FILE` 型文件指针。由于历史的原因, `getc()` 返回一个整型量而且高位字节为 0。当读到文件末时, `getc()` 返回一个 `EOF` 标记。可以用以下程序读一个

文件直到文件末:

```
ch = getc(fp);
while(ch != EOF) {
    ch = getc(fp);
}
```

13.7.5 feof()的使用

缓冲型文件系统还可对二进制数据操作。当一个文件以二进制输入打开时,将会读入一个与 EOF 标志相等的整型量。这会造成程序判定文件已经结束,而实际上文件物理结尾并未到达。为了解决这个问题,ANSI C 中包含了一个函数 feof(),形式如下:

```
int feof(FILE *fp);
```

函数的原型在 STDIO.H 中。如果到达了文件末则返回值为真;否则,返回零值。因此,以下程序能够读二进制文件直到遇到文件结尾:

```
while(! feof(fp)) ch=getc(fp);
```

这样的方法也同样适用于文本文件。

13.7.6 关闭文件

fclose()函数用来关闭一个由 fopen()打开的流。这个函数将把磁盘缓冲区的内容传给文件,并且关闭系统级的文件。未关闭的流会引起很多问题,例如数据丢失、文件损坏及其它一些可能的错误。fclose()函数释放与这个流有关的文件控制块,以便能再用它来打开一个新文件。操作系统对同时打开的文件数目有一定的限制,所以有时有必要先关闭一个文件再打开另一个文件。

函数 fclose()的调用方式为:

```
int fclose(FILE *fp);
```

其中 fp 是一个 fopen()调用时返回的文件指针。文件关闭成功,返回一个 0;若出错则返回其它值。可以使用标准函数 ferror()来确定和显示错误类型。通常只有在磁盘已被取出驱动器或磁盘已满时,才会发生关闭错误。

13.7.7 ferror()和 rewind()函数

ferror()函数用来确定操作中是否出错。如果一个文本方式打开时发生写错误,则函数将返回 EOF。此时可用 ferror()函数确定究竟出了什么事,它的原型是:

```
int ferror(FILE *fp);
```

其中 fp 是一个有效的文件指针。在文件操作中发生错误时返回“真”,否则返回“假”,为保证每个文件操作正确,应该在每次文件操作后立即调用 ferror()函数,避免漏掉可能出现的错误。ferror()函数原型在 STDIO.H 中。

函数 rewind()将文件的指针重新设置到该文件的起点。它的原型为:

```
void rewind(FILE *fp)
```

其中 fp 是一个有效的文件指针。rewind()的原型在 STDIO.H 中。

13.7.8 fopen(),getc(),putc()和fclose()函数的用法

函数 fopen()、getc()、putc()和 fclose()构成文件操作函数的最小集合。下例程序 ktod 是 putc()、fopen()和 fclose()的用法简例,它能够将从键盘读入的字符写到一个磁盘文件中,遇到符号 \$,输出文件名由命令行指定。例如在键盘上键入 KTOD TEST,可以向名为 TEST 的文件写入文本。

```
#include <stdio.h>
#include <stdlib.h>
main(int argc, char *argv[])
{
    FILE *fp;
    char ch;
    if(argc != 2){
        printf("You forgot to enter the filename\n");
        exit(1);
    }
    if((fp=fopen(argv[1], "w")) == null) {
        printf("cannot open file\n");
        exit(1);
    }
    do {
        ch=getchar();
        if(EOF==putc(ch,fp)) {
            printf("File Error");
            break;
        }
    } while(ch != '$');
    fclose(fp);
    return 0;
}
```

下面的 DTOS 程序,可以读入任何 ASCII 文件,并将其内容显示在屏幕上:

```
/* dtos : A program that reads files and displays them on the screen. */
#include <stdio.h>
#include <stdlib.h>
main(int argc, char *argv[])
{
    FILE *fp;
    char ch;
    if(argc != 2){
        printf("You forgot to enter the filename\n");
        exit(1);
    }
}
```

```

    if((fp=fopen(argv[1],"r"))=NULL){
        printf("cannot open file\n");
        exit(1);
    }
    ch = getc(fp); /* read one character */
    while(ch!=EOF) {
        putchar(ch); /* print on screen */
        ch = getc(fp);
    }
    if(ferror(fp)) printf("File Error");
    fclose(fp);
    return 0;
}

```

下面这段程序可以用来拷贝任何类型的文件。注意,文件是以二进制模式打开的,所以要使用 `ferror()` 来检查文件是否结束。

```

#include <stdio.h>
#include <stdlib.h>
main(int argc, char * argv[])
{
    FILE * in, * out;
    char ch;
    if(argc == 1) {
        printf("You forgot to enter a filename\n");
        exit(1);
    }
    if((in=fopen(argv[1],"rb"))=NULL) {
        printf("cannot open source file\n");
        exit(1);
    }
    if((out=fopen(argv[2],"wb"))=NULL) {
        printf("cannot open destination file\n");
        exit(1);
    }
    while(! feof(in)) {
        ch=getc(in);
        if(ferror(in)){
            printf("Error reading file");
            break;
        }
        putc(ch,out);
        if(ferror(out)) {
            printf("Error writing file");
            break;
        }
    }
}

```

```
    }  
    }  
    fclose(in);  
    fclose(out);  
    return 0;  
}
```

13.7.9 getw()和 putw()函数的使用

除了 getc()和 putc()之外, Turbo C 还提供了另外两个缓冲型 I/O 函数: putw()和 getw(), 用于从磁盘文件中读或写整型量, 这些函数是由 ANSI C 标准定义的, 其用法与 getc()和 putc()完全相同, 区别在于前者读写的是整型量, 而后者读写的是字符。它们的原型如下:

```
int putw(int i, FILE *fp);  
int getw(FILE *fp);
```

下面的语句用来向文件指针 fp 所指向的磁盘文件写一个整型数:

```
putw(100, fp);
```

getw()和 putw()的原型在 STDIO.H 中。

13.7.10 fgets()和 fputs()函数

缓冲型 I/O 系统中的两个函数, fgets()和 fputs(), 用来从流中读写字符串。其原型如下:

```
char *fputs(char *str, FILE *fp);  
char *fgets(char *str, int length, FILE *fp);
```

函数 fputs()与 puts()几乎完全一样, 区别只在于 fputs()是将字符串写入指定的流中。函数 fgets()从指定的流中读取字符串, 直到读到换行符或第 length-1 个字符为止。如果读入的是换行符, 它将作为字符串的一部分(这与 gets()不同), 但若 fgets()被中断, 则这个字符串是空。

fgets()和 fputs()函数原型在 STDIO.H 中。

13.7.11 fread()和 fwrite()函数

fread()和 fwrite()是缓冲型 I/O 提供的两个用于读写数据块的函数。其原型如下:

```
unsigned fread(void *buffer, int numbytes, int count, FILE *fp);  
unsigned fwrite(void *buffer, int numbytes, int count, FILE *fp);
```

对于 fread(), buffer 是一个用来接受从文件读取内容的数据存储区指针。对于 fwrite(), buffer 是一个指向将被写到文件中去的数据的指针。num_bytes 是读写的字节数。参数 count 指示要被读或写的字段数(每个字段长度 num_bytes)。fp 是一个已打开的流的文件指针。这两个函数的原型在 STDIO.H 中定义。

函数 fread()返回读入的字节数, 如果到达了文件尾或发生了错误则可能小于 count。函数 fwrite()返回写入的字节数。除非发生了错误, 否则这个数与 count 相等。

只要文件以二进制文件方式打开, fread()和 fwrite()就可以读写任何类型信息。下例中将写一个浮点数到磁盘文件中:

```
#include <stdio.h>
#include <stdlib.h>
main(void)
{
    FILE *fp;
    float f=12.23;
    if((fp=fopen("test","wb"))==NULL) {
        printf("cannot open file\n");
        exit(1);
    }
    if(fwrite(&f,sizeof(float),1,fp)!=1)
        printf("File Error");
    fclose(fp);
    return 0;
}
```

上面程序表明,buffer 可以而且经常只是一个简单变量。程序中的另一个 C 操作符 sizeof,将返回变量和数据类型的字节数,用 sizeof 可确保程序正确工作。

fread()和 fwrite()的一个最有效的应用是用于数组以及后面将讲到的结构。例如,下面程序用一个单一的 fwrite()语句将数组 sample 中的内容写入文件 sample 中:

```
#include <stdio.h>
#include <stdlib.h>
main(void)
{
    FILE *fp;
    float sample[100];
    int i;
    if((fp=fopen("sample","wb"))==NULL) {
        printf("cannot open file\n");
        exit(1);
    }
    for(i=0;i<100;i++) sample[i]=(float)i;
    /* this saves the entire array in one step */
    if(fwrite(sample,sizeof(sample),1,fp)!=1)
        printf("File Error");
    fclose(fp);
    return 0;
}
```

在这个程序中用 sizeof 说明 sample 大小的方法是值得注意的。

下面一个程序用 fread()读入先前程序写入的信息,并将这些数据显示在屏幕上。

```
#include <stdio.h>
#include <stdlib.h>
main(void)
```

```

{
    FILE *fp;
    float sample[100];
    int i;
    if((fp=fopen("sample","rb"))==NULL) {
        printf("cannot open file\n");
        exit(1);
    }
    /* this reads the entire array in one step */
    if(fread(sample,sizeof(sample),1,fp)!=1)
        printf("File Error");
    for(i=0;i<100;i++) printf("%f",sample[i]);
    fclose(fp);
    return 0;
}

```

13.7.12 fseek()函数和随机访问 I/O

缓冲型 I/O 系统中的 `fseek()` 函数可以设置文件位置指针用于完成随机读写,其原型为:

```
int fseek(FILE *fp, long numbytes, int origin)
```

这里 `fp` 是 `fopen()` 调用时所返回的文件指针。`numbytes` 是个长整型量,表示从 `origin` 位置到当前位置的字节数。`origin` 是 `stdio.h` 中定义的宏之一:

origin	宏名	值
文件开头	<code>SEEK_SET</code>	0
当前位置	<code>SEEK_CUR</code>	1
文件尾部	<code>SEEK_END</code>	2

为了从文件头开始搜索第 `numbytes` 个字节, `origin` 应该用 `seek_SET`。从当前位置起搜索则用 `seek_CUR`, 从文件尾部开始搜索用 `seek_END`。

以下程序可以从一个叫 `test` 的文件中读取 234 个字节:

```

...
FILE *fp;
char ch;
if ((fp=fopen("test","rb"))==NULL) {
    printf("cannot open file\n");
    exit(1);
}
fseek(fp,234,0);
ch=getc(fp);
...

```

`fseek()` 在能够正确执行时返回值为 0, 否则返回一个非零值。

下面的 `dump` 程序利用 `fseek()` 函数使用户可以用 ASCII 和十六进制来查看任意文件

的内容。通过一个可任意移动的“扇区”，用户可以看到 128 字节长的文件内容。当输入 D(即 dump 转储)命令时，它以与 debug 相同的格式显示输出。如想退出这个程序，就输入 -1 即可。注意，在使用 fseek() 读到文件的结束标记时，如果读入的字节数还不到 size 值，fread() 也仍然将读入这个字节，并将读入的个数赋给 display() 函数[fread() 函数返回的是实际读入的个数]。用户可以输入这个程序至计算机中以了解它是如何工作的。

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#define SIZE 128
char buf[SIZE];
void display(int numread);
main(int argc, char *argv[ ])
{
    FILE *fp;
    int sector, numread;
    if(argc != 2) {
        printf("usage: dump filename\n");
        exit(1);
    }
    if((fp=fopen(argv[1], "rb")) == NULL) {
        printf("cannot openfile\n");
        exit(1);
    }
    for(;;) {
        printf("enter sector (-1 to quit): ");
        scanf("%ld", &sector);
        if(sector < 0) break;
        if(fseek(fp, sector * SIZE, SEEK_SET)) {
            printf("seek error\n");
        }
        if((numread=fread(buf, 1, SIZE, fp)) != SIZE) {
            printf("EOF reached\n");
        }
        display(numread);
    }
    return 0;
}

void display(int numread)
{
    int i, j;
    for(i=0; i<=numread/16; i++) {
        for(j=0; j<16; j++) printf("%3x", buf[i * 16 + j]);
```



```

printf(" ");
for(j=0;j<16;j++) {
    if(isprint(buf[i * 16+j])) printf("%c",buf[i * 16+j]);
    else printf(",");
}
printf("\n");
}
}

```

上例中,函数 `isprint()` 用来确定字符是否是可打印字符。当字符是可打印字符时 `isprint()` 返回真,否则返回假,该函数包含于头文件 `CTYPE.H` 中,DUMP 程序能够把它自己的内容显示出来,如图 13.2 所示。

```

enter sector(-1 to quit):0
23 69 6e 63 6c 75 64 65 20 3c 73 74 64 69 6f 2e  #include <stdio.
68 3e  d a 23 69 6e 63 6c 75 64 65 20 3c 63 6f  h>..#include <co
6e 69 6f 2e 68 3e  d a 23 69 6e 63 6c 75 64 65  nio.h>..#include
20 3c 63 74 79 70 65 2e 68 3e  d a 23 69 6e 63  <ctype.h>..#inc
6c 75 64 65 20 3c 73 74 64 6c 69 62 2e 68 3e  d lude <stdlib.h>.
a 23 64 65 66 69 6e 65 20 53 49 5a 45 20 31 32  .define SIZE 12
30  d a 63 68 61 72 20 62 75 66 5b 53 49 5a 45  B..char buf[SIZE
5d 3b  d a 76 6f 69 64 20 64 69 73 70 6c 61 79  l:..void display

enter sector(-1 to quit):1
28 69 6e 74 20 6e 75 6d 72 65 61 64 29 3b  d a (int numrcad)..
76 6f 69 64 20 62 6f 72 64 65 72 20 69 6e 74 20  void border(int
73 74 61 72 74 78 2c 69 6e 74 20 73 74 61 72 74  startx,int start
79 2c 69 6e 74 20 65 6e 64 78 2c 20 69 6e 74 20  y,int endx, int
65 6e 64 79 29 3b  d a 76 6f 69 64 20 63 6c 73  endy):..void cls
28 69 6e 74 20 73 74 61 72 74 78 2c 69 6e 74 20  (int startx,int
73 74 61 72 74 79 2c 69 6e 74 20 65 6e 64 78 2c  starty,int endx,
20 69 6e 74 20 65 6e 64 79 29 3b  d a 6d 61 69  int endy):..mai

enter sector(-1 to quit):

```

图 13.2 DUMP 程序输出实例

13.7.13 标准流

当一程序开始运行时,自动打开三个流:标准输入(`stdin`)、标准输出(`stdout`)和标准错误(`stderr`)。一般情况下它们均与控制台有关,但也可以由操作系统重定向到别的设备上。在缓冲型 I/O 系统中可以使用文件指针,完成控制台操作。例如 `putchar()` 函数可以定义为:

```

putchar(char c)
{
    putc(c,stdout);
}

```

`stdin`、`stdout` 和 `stderr` 可以作为文件指针,但仅限于使用 `FILE` 型变量的函数中。

I/O 控制函数 `getchar()`、`putchar()`、`printf()` 和 `scanf()` 实际上是用 `stdin` 和 `stdout` 来实

现它们的操作。DOS 操作系统允许用 > 和 < 命令行操作符进行 I/O 重定向,使这些函数可以对磁盘文件进行读写。例如,编译运行以下这个名叫 IOTEST 的程序后,屏幕上不显示任何东西,但如果列出 OUT 的内容,将看到信息已被写入。

```
#include <stdio.h>
main(void)
{
    printf("Hello there");
    return 0;
}
```

注意,stdin、stdout 和 stderr 是常量而不是变量,所以不能转换。这些文件指针在程序开头自动地创立并在程序结束时自动关闭,用户不要试图去关闭它们。

除了这三个标准流之外,Turbo C 还自动打开两个流:stdprn 和 stdaux,它们分别与打印机和辅助口有关。

13.7.14 fprintf()和 fscan()函数

缓冲型 I/O 系统 fprintf()和 fscan(),除了是对磁盘文件进行操作外,与 printf()和 scanf()完全相同。其原型为:

```
int fprintf(FILE *fp, char *控制字符串, ...);
int fscanf(FILE *fp, char *控制字符串, ...);
```

其中 fp 是由 fopen()返回的文件指针,除了将其输出指向到由 fp 确定的文件外,fprintf()和 fscanf()的其他操作与 printf()和 scanf()函数完全相同。

以下示例将说明如何使用这些函数。该程序用一个磁盘文件来存放一个简单的电话簿。用户可以输入姓名、电话,也可以给出一个名字电话号码。

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
void add_num(void),lookup(void);
int menu(void);
main(void)
{
    char choice;
    do {
        choice = menu();
        switch(choice) {
            case 'a': add_num();
                break;
            case 'l': lookup();
                break;
        }
    }
```

```

    } while(choice != 'q');
    return 0;
}

menu(void)
{
    char ch;
    do {
        printf("(A)dd, (L)ookup, or (Q)uit:");
        ch=tolower(getche());
        printf("\n");
    } while(ch != 'q' && ch != 'a' && ch != 'l');
    return ch;
}

void add_num(void)
{
    FILE *fp;
    char name[80];
    int a_code,exchg,num;
    /* open it for append */
    if ((fp=fopen("phone","a"))==NULL) {
        printf("cannot open directory file\n");
        exit(1);
    }
    printf("enter name and number: ");
    fscanf(stdin,"%s%d%d%d",name,&a_code,&exchg,&num);
    fscanf(stdin,"%*c"); /* remove CR from input stream */
    /* write to file */
    fprintf(fp,"%s %d %d %d\n",name,a_code,exchg,num);
    fclose(fp);
}

/* Find a number given a name. */
void lookup(void)
{
    FILE *fp;
    char name[80],name2[80];
    int a_code,exchg,num;
    /* open it for read */
    if ((fp=fopen("phone","r"))==NULL) {{
        printf("cannot open directory file\n");
        exit(1);
    }}
    printf("name? ");
    gets(name);

```

```
/* look for number */
while(! feof(fp)) {
    fscanf(fp, "%s%d%d%d", name2, &a_code, &exchg, &num);
    if(! strcmp(name, name2)) {
        printf("%s: (%d) %d-%d\n", name, a_code, exchg, num);
        break;
    }
}
fclose(fp);
}
```

该程序允许用户往电话簿加入一个号码并可对其查询。程序先显示一个菜单,当添加号码时,调用函数 `add_num()`,查询号码时则调用 `lookup()`。用户可以输入并运行这个程序,当输入数据时,必须确保用空格分开姓名、区码、前缀和号码。例如,下面是正确的输入:

ALEX 213 555 1234

在输入姓名和电话号码后,检查一下文件 `phone`。用户可以发现它的格式和用 `printf()` 显示输出格式相同。

注意:因为 `fprintf()` 和 `scanf()` 以格式化的 ASCII 数据而不是二进制数据向磁盘文件读写各种数据,调用这两个函数需要一些附加操作,所以效率不一定最高。如果要求速度快或文件很长时,用户应使用 `fread()` 和 `fwrite()` 函数。

13.7.15 删除文件

`remove()` 函数删除所指定的文件,用法为:

```
int remove(char * filename);
```

执行成功返回 0,否则返回非零值。

13.8 非缓冲型 I/O——UNIX 型文件系统

C 语言最早是在 UNIX 操作系统下开发的,使用一些独立于缓冲型文件系统的函数。在表 13.6 中列出了一些低级 UNIX 磁盘 I/O 函数。这些函数需要把头文件 `IO.H` 连入任何要使用这些函数的程序的开头。

表 13.6 UNIX 型非缓冲 I/O 函数

函数名	功 能
<code>read()</code>	读一个数据缓冲区
<code>write()</code>	写一个数据缓冲区
<code>open()</code>	打开一个磁盘文件
<code>close()</code>	关闭一个磁盘文件
<code>lseek()</code>	搜索文件中指定字节
<code>unlink()</code>	使一个文件脱离连接

因为编程者必须提供和维护所有的磁盘缓冲区,函数不能自己完成这一工作,所以由这些函数组成的磁盘 I/O 子系统叫做非缓冲型文件系统。与 `getc()` 和 `putc()` 等函数不同,它们不能从一个数据流中读写字符。在每次调用函数 `read()` 和 `write()` 时只能读或写一个完整信息缓冲区(这与 `fread()` 和 `fwrite()` 相似)。

非缓冲型文件系统不是由 ANSI 标准定义的,如果使用这些函数,就可能出现可移植性方面的问题。虽然今后几年非缓冲文件系统的使用会减少,但因很多现有的 C 语言程序中用到它们,并且现有的各种 C 语言编译程序也支持它们,所以在本节还要介绍一下这些函数。

注意:UNIX 型文件系统使用的原型和有关必要信息可以在头文件 `IO.H` 找到。

13.8.1 `open()`、`creat()` 和 `close()` 函数

UNIX 型系统使用称为文件句柄的整型说明符,而不使用 FILE 型文件指针,这点与 ANSI I/O 系统不同。`open()` 的原型如下:

```
int open(char * filename, int mode, int access);
```

其中 `filename` 是任一有效文件名, `mode` 可为以下三个由 `FCNTL.H` 定义的宏之一。

宏名	意义	值
<code>ORDONLY</code>	只读	1
<code>OWRONLY</code>	只写	2
<code>ORDWR</code>	读/写	4

Turbo C 还允许一些其它选择项加在这几个基本类型后,具体规定请参考有关的手册。

为具有移植性而保留的参数 `access` 仅与 UNIX 环境有关。Turbo C 专门为 DOS 定义了一个叫做 `open()` 的函数,该函数的原型如下:

```
int open(char * filename, int mode);
```

此函数不使用 `access` 这一参数。在本节中将把 `access` 设为 0。

在成功调用 `open()` 后返回的正整数称为文件句柄,不能打开文件时返回 -1。文件句柄是其它 UNIX 型文件系统函数所要求的,它不同于 ANSI I/O 系统的文件指针。

`open()` 的调用格式如下:

```
if((fd=open(filename,mode,0))==-1) {
    printf("cannot open file\n");
    exit(1);
}
```

如果 `open()` 语句指定的文件不在磁盘上,打开操作失败并且不生成这个文件。

函数 `close()` 的原型是:

```
int close(int fd);
```

如果 `close()` 函数返回 -1,则表明不能关闭该文件。例如,当磁盘已从驱动器中取出时就会出现这种情况。

调用 `close()` 函数可实现关闭文件并且释放该文件的文件句柄。一方面,被打开的文件数目是有限的,所以对于不再需要的文件应该及时用 `close()` 函数关闭掉;另一方面,关闭操作将把操作系统的内部磁盘缓冲区里的全部信息写入磁盘,没有关闭的文件会丢失数据。

使用 `creat()` 可以为写操作生成一个新文件。`creat()` 的原型如下：

```
int creat(char * filename, int access);
```

其中 `filename` 是任何有效的文件名, 参数 `access` 用来指定访问的模式和指明该文件为二进制文件还是文本文件。因为 `_creat()` 中的 `access` 与 UNIX 环境相联系, Turbo C 为 MS-DOS 专门定义了 `creat()` 函数, 它用一个文件属性字节 (FILE_ATTRIBUTE_BYTE) 代替 `access`。DOS 中的每个文件都与一个属性字节有关, 它指定了各种信息位。这个文件属性字节的意义列于表 13.7。

表 13.7 DOS 文件属性字节的值和含义

位号	值	含义
0	1	只读文件
1	2	隐含文件
2	4	系统文件
3	8	卷标号名
4	16	子目录名
5	32	数据档案
6	64	未定义
7	128	未定义

表 13.7 中的值可以相加。例如想生成一个只读隐含文件, 可以将 $3 = (2 + 1)$ 赋给 `access`。一般情况下, 生成一个标准文件 `access` 设为 0。

13.8.2 read() 和 write() 函数

`write()` 可以访问以写操作打开的文件。`write()` 函数的原型如下：

```
int write(int fd, void * buf, unsigned size);
```

每次调用 `write` 时, 由 `buf` 指向的缓冲区中 `size` 长度的字符就写到由 `fp` 指定的磁盘文件中。在写操作完毕后函数返回所写的字节数。写操作失败时返回 -1。

函数 `read()` 是 `write()` 的逆操作, 原型为：

```
int read(int fd, void * buf, int size);
```

这里的 `fd`、`buf` 和 `size` 与函数 `write` 中相同, 读入的数据放在由 `buf` 指向的缓冲区中, 返回值为实际写入的字符数。在文件的物理结尾时返回 0, 出错时返回 -1。

下例给出了非缓冲型 I/O 系统的一些用法。该程序将把从键盘上读入的几行文字写入磁盘文件, 然后再从磁盘文件中将它们读出来。

```
#include <fcntl.h>
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int input(char * buf, int fd1);
void display(char * buf, int fd2);
```

```

#define BUF_SIZE 128
main(void)
{
    char buf[BUF_SIZE];
    int fd1,fd2;
    if((fd1=_creat("test",O_WRONLY))==-1) { /* open */
        printf("cannot open file\n");
        exit(1);
    }
    input(buf,fd);
    /* now close file and read back */
    close(fd1);
    if((fd2=open("test",0,O_RDONLY))==-1) { /* open */
        printf("cannot open file\n");
        exit(1);
    }
    /* display text */
    display(buf,fd2);
    close(fd2);
    return 0;
}

/* Read some lines of text from the keyboard. */
input(char * buf,int fd1)
{
    register int t ;
    printf("enter text, to stop enter 'quit' on a new line\n");
    do {
        for(t=0;t<BUF_SIZE;t++) buf[t]='\0';
        printf(" ");
        gets(buf); /* input chars from keyboard */
        if(write(fd1,buf,BUF_SIZE)!=BUF_SIZE) {
            printf("error on write\n");
            exit(1);
        }
    } while(strcmp(buf,'quit'));
}

void display(char * buf,int fd2)
{
    for(;;) {
        if(read(fd2,buf,BUF_SIZE)==0) return;
        printf("%s\n",buf);
    }
}

```

13.8.3 unlink()函数

函数 unlink() 可以从目录中删除一个文件。该函数的标准调用方式是：

```
int unlink(char * filename);
```

其中 filename 是指向文件的一个字符型指针。失败时返回出错信息(通常是一1),当文件不在磁盘上或磁盘写有保护时会出现这种情况。

13.8.4 随机访问文件和 lseek()函数

UNIX 型 I/O 系统下的随机访问文件 I/O 操作是通过调用 lseek() 函数实现。它的原型如下：

```
long lseek(int fd, long numbytes, int origin);
```

fd 是调用函数 creat() 或 open() 时返回的文件句柄, numbytes 必须为长整型。origin 可为以下三个宏之一：

ORIGIN	宏 名	实际值
文件开头	SEEK_SET	0
当前位置	SEEK_CUR	1
文件尾端	SEEK_END	2

从文件开头搜索 numbytes 个字节时, 起点 origin 设置为 SEEK_SET; 从当前位置开始搜索时设置为 SEEK_CUR; 从文件尾端开始搜索时为 SEEK_END。

lseek() 函数调用成功后, 返回一个在文件开头处说明的长整型的偏移量。调用失败时, 返回-1。

现在让我们使用 lseek() 函数对前面讨论的 DUMP 程序作一些改动, 以适应 UNIX 型的 I/O 系统。见下面的示例程序：

```
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
#include <ctype.h>
#include <stdlib.h>
#define SIZE 128
char buf[SIZE];
void display(int num);
main(int argc, char * argv[])
{
    char s[10];
    int fd, sector, numread;
    long pos;
    if(argc == 2) {
        printf("You forgot to enter the filename.");
        exit(1);
    }
```



```

if((fd=open(argv[1],O_RDONLY,0))==-1) {
    printf("cannot open file\n");
    exit(1);
}
for(;;) {
    printf("\n\nbuffer:");
    gets(s);
    sector=atoi(s);
    if(sector<0) break;
    pos=(long)(sector * SIZE);
    if(lseek(fd,pos,SEEK_SET)!=pos)
        printf("seek error\n");
    numread=read(fd,buf,SIZE);
    if(numread==-1) {
        printf("File Error");
        break;
    }
    display(numread);
}
close(fd);
return 0;
}

void display(int numread)
{
    int i,j;
    for(i=0;i<=numread/16;i++) {
        for(j=0;j<16;j++) printf("%3X",buf[i * 16+j]);
        printf(" ");
        for(j=0;j<16;j++) {
            if(isprint(buf[i * 16+j])) printf("%c",buf[i * 16+j]);
            else printf(".");
        }
        printf("\n");
    }
}

```

13.9 理解 I/O 概念

在开始使用 Turbo C I/O 函数前,需要知道 C 语言是如何处理输入输出函数的。Turbo C 是利用来描述数据如何进出程序的。Turbo C 流能保证初级水平的操作员也能使用计算机输入/出设备。

这一节介绍计算机上可使用的主要的 I/O 设备。在介绍计算机如何处理数据后,还将讲

述 Turbo C 如何将设备和流联接起来以及在系统中如何保存文件。最后,还将介绍 Turbo C 所使用,文件的两个基本类型:二进制文件和文本文件。

13.9.1 文件与设备

在使用大量数据时,这些数据通常据存储在文件里。一个文件就是一组相关数据。根据所使用文件的类型,能将数据从文件中读出或者写入文件中,有时两者都行。

每个文件都是与某种设备类型相联系的。设备(device)就是一种存储或传输信息的计算机硬件。读者已熟悉的设备之一是计算机硬盘。硬盘是一种主要的存储设备,它的用途是永久存储大量信息。

设备可以分为两种:永久设备和交互设备。正象你所知道的,硬盘是一种永久设备。永久设备能将数据存储很长一段时间。其它永久设备有:磁盘驱动器、磁带机、CDROM 设备等。这些设备各有自己的优缺点,但设计得都能将数据额外地保存一段时间。

交互设备通常用于给计算机传输数据。与永久设备不同的是,交互设备只能将数据存储很短一段时间。视频卡是显示监视器组成一种交互设备。数据送到视频卡中,视频卡在监视器中产生图象,根据需要,只要不送其它数据清屏,图象就可一直存在于监视器中。监视器不能永久地存储数据;一旦数据在屏幕中丢失,数据就将永久地丢失(至少要到你再重新生成这些数据)。计算机上接的其它交互设备有鼠标器、键盘、调制解调器。

计算机上这些设备的最大问题是它们速度太慢。即使是高频硬盘,它们传输数据的速度也比计算机生成数据慢得多。因为计算机接的这些设备与 CPU 相比运行要慢,所以许多设备加了缓冲区。缓冲区是一块特别的内存区域,用来保存发送到设备的数据或设备要接收的数据。

如果设备给计算机传送数据,缓冲区就将数据保存起来到 CPU 能方便地处理这些数据为止。如果设备要从计算机中接收数据,缓冲区就保存从 CPU 送来的数据,直到设备能接收这些数据为止。正确使用缓冲区能极大地提高程序的性能。图 13.3 演示缓冲区是如何工作的。

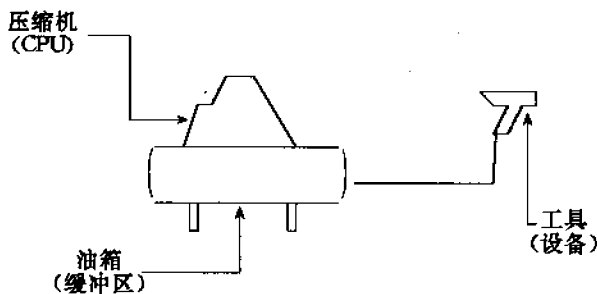


图13.3 缓冲区是如何为CPU和设备工作的

总的说来,I/O 缓冲区的作用类似于容器对压缩机的作用。空气压缩机压缩空气,然后将高压气体贮存在容器中,当喷嘴需要空气时,容器就可提供大量的高压气体。

因为喷嘴在大多数时间里不工作,所以说压缩机连续工作的效率低。只有当容器中的气压降到某一特定值时,压缩机才工作。当压缩机工作时,它全速运转,效率高。

I/O 缓冲区的作用类似于空气容器。I/O 缓冲区将数据保存到 I/O 设备准备接收数据为止。当设备将接收数据时,缓冲区将设备能处理的所有数据提供给设备。

本例中的压缩机类似于计算机中的 CPU。因为 I/O 设备运行慢,所以说如果 CPU 在所有时间总处于将数据传输给这些设备的状态,其效率将是很低的。如果没有输入/出缓冲区, CPU 就要浪费很多时间。缓冲区允许 CPU 大块大块传送数据,因而提高了 CPU 的工作效率。

在压缩机的例子中,容器仅代表输出缓冲区。一般程序要用到输入输出两种缓冲区。输入缓冲区保存从交互设备送来的数据,直到 CPU 准备处理这些数据为止。

13.9.2 文件与流

对于许多操作, Turbo C 通过流存取数据。流代表文件,用于对文件进行传接数据。

流也是一种处理 I/O 任务的可移植方式。Turbo C 提供几种函数,它们能使用户直接通过 DOS 接口操作物理文件。但是这些函数不能移拉到其它编译系统上去,然而许多使用流的函数可移植到其它编译系统中,而不必改变源码。如果需要编写可移植程序,可使用那些通过流来控制数据文件的 I/O 函数。

要将流和文件联系起来,只需将文件打开。如 `fopen()` 函数就自动将流和文件联系起来。下面的例子告诉读者如何使用 `fopen()` 函数:

```
FILE * my_file;
...
If ((my_file = fopen("text.txt", "w")) == NULL) {
    printf("Could not open file.\n");
    printf("Exiting program.\n");
    exit(0);
}
...
fclose(my_file);
```

在上述代码段中, `fopen()` 函数打开 `test.txt` 文件,准备写文件。但即使通过 `fopen()` 函数自动将流和 `test.txt` 文件连结起来,也不能方便地直接操作流对象。为了控制流,继而控制文件,必须使用库函数 `printf()`。在上面的例子中, `printf()` 就是这样的库函数。它控制 `my_file` 指向的 `FILE` 类型数据对象。

`My_file` 指向一个特定的数据对象,包含了某一特定流的信息。由 `my_file` 指向的数据对象具有数据类型 `FILE`。为了控制流,必须使用这些库函数,能存取和使用包含在 `FILE` 类型数据对象中的数据。

一个 `FILE` 类型的数据对象包含一些特定的流的信息。`FILE` 类型的数据对象包含如下数据项:

- 与文件相连的 I/O 缓冲区的有关信息。
- 一个标志 I/O 出错的指针。
- 一个文件位置指针器,标记文件当前的位置。
- 一个标志文件结束的指针。

结束一个文件时,需要将流与文件断开并关闭文件。`fclose()` 函数执行了这两项任务。上

面程序中的最后一行告诉我们如何使用 fdose。

Turbo C 为程序提供了五种标准流。

在写 C 程序时,自然地要访问下面五种流。

流	功能	I/O 设备
stdin	输入流	键盘
stdout	输出流	视频监视器
stdprn	打印流	打印口
stdaux	辅助输出	串行口
stderr	错误流	视频监视器

stdin 是标准的输入流,它处理键盘输入。

stdaux,既可用于作输入也可用作输出,使你能从串行口中接收数据或者向串行口发送数据。串行口让你将计算机连接到调制解调器上甚至到另一个计算机上。

上面有四种流由 Turbo C 提供,都是标准输出流,Stdout 是将数据送给屏幕的输出流。Stdprn 是将输出数据送到打印口的输出流。Stdaux 是将数据送到串行口或 COM 端口的输出流。stderr 是标准错误流,它将错误信息送到屏幕上。

13.9.3 文本流和二进制流

打开文件,就将流与文件联系起来。打开文件的方式,决定了对流与文件联系的类型。Turbo C 有两种类型的流可以使用到文件中:文本流和二进制流。

文件的流的类型决定了该文件中的数据被转换的方式。与文本流相关的函数改变或转化了文件中某些特殊字符。与二进制流相关的函数不转变文件中的任何字符。

当读入一个文本流时,文件中的某些特殊字符转化为内部的格式。例如,当从文件中读入回车字符和换行字符时,这两个字符就要转化为换行字符(\n)。另外,当文件被看作文本流时,制表(tab)字符就转化为一连串空格字符。

当文件被当作二进制流处理时,不进行任何转化。二进制流中的数据一一对应于文件中的数据。当二进制流读入一个回车字符和换行符时,每一个控制符都包含在流中。二进制流不能象文本流一样将回车字符和换行字符转化为换行符。图 13.4 告诉我们文本流和二进制流是如何表示文件中的信息的。

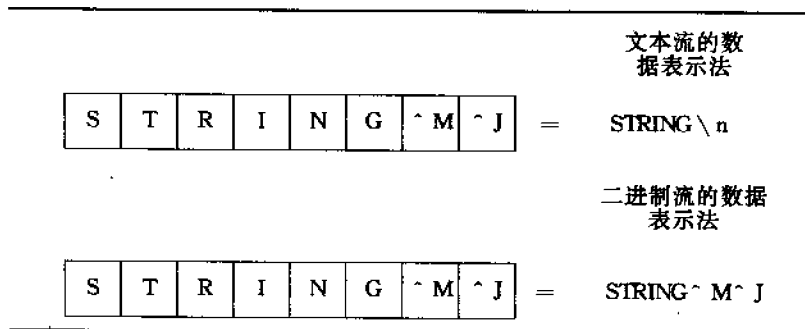


图13.4 二进制流和文本流中数据表示

如图 13.4 中所示,当文件被分别作为文本流和二进制之前打开时,文件中的数据分别如何表示。该文件包含数个字符。前面六个字符拼成单词 String,最后两个字符代表回车字符[^]M 和换行字符[^]J。

当图 13.4 中的文件被当作文本流打开时,回车字符和换行字符通过与文本流相关的函数转换成单个字符,用换行符(`\n`)替代了文件中的回车字符和换行(finefeed)字符。但当同一文件被当作二进制流打开时,就不会发生字符转换。在二进制流中回车字符和换行字符(linefeed)被当作两个无关的字符。

文本流对文件来说是很很有用的,这些文件包含可直接读入的信息。而当计算机处理文件时,经常要用到二进制流。例如,在连接目标文件时它是作为二进制流打开的。二进制流也用来读写文件,这些文件包含存储在计算机内部格式中的数据。

13.10 利用标准流进行 I/O

Turbo C 自动为程序打开五种标准 I/O 流。因为通常不必为这五种标准流打开任何文件,所以用起来很容易。Turbo C 自动打开这些标准流,使之提供使用。在前一节,已经用表列出这些流。

在本节,将介绍两组具有很强输入输出功能的函数,通常和标准流一起使用。首先,我们利用格式的 I/O 函数来输入输出标准数据的所有类型。然后,学习如何使用字符 I/O 函数来输入输出此字符信息。

13.10.1 使用格式化 I/O 函数

Scanf()和 Printf()函数类似之处在于格式的数据,这些数据是按指定顺序排列的。使用 scanf()函数时,指定所要读入数据的类型和数据输入的顺序。对 printf()而言,指定的是输出数据的类型及其顺序。Scanf()和 Printf()均能作用于 C 的任何一种基本数据类型。例如,Scanf()能用来输入浮点数,这就和输入字符和整数一样容易。printf()能按所要求的顺序打印任何一种基本类型数据。

程序 fdata.c 是一个简单的库存数据库程序,用 scanf()和 printf()系列中的函数来检索和显示数据。这个程序告诉我们如何利用格式化的 I/O 函数来创建更新,显示一个简单的库存数据库。

下列给出 fdata.c 程序如何利用 I/O 函数来终止文件和服务于文件的输入输出。

```
1  /* FDATA.C This program demonstrates the use of the
2     formatted I/O functions. Formatted I/O
3     functions are used for both terminal and
4     file I/O.  */
5
6  #include <stdlib.h>
7  #include <stdio.h>
8  #include <conio.h>
9  #include <io.h>
10
```

```
11  #define DATA_FILE "data.fil"
12
13  int menu( void );
14  int add( void );
15  int display( void );
16
17  /* main() — The main() function calls functions based on
18     the value returned by the menu() function.
19
20     If main() executes correctly, a value of 0 is
21     returned.  */
22
23  int main( void ) {
24     int return_value;
25
26     while( ( return_value = menu() ) != 0 ) {
27         switch( return_value ) {
28             case 1:    add();
29                       break;
30             case 2:    display();
31                       break;
32             case 3:    clrscr();
33                       printf( "Enter a correct menu value. \n" );
34                       printf( "Press a key to continue. \n" );
35                       while( ! kbhit() );
36                       break;
37         }
38     }
39     return 0;
40 }
41
42 /* menu() — The menu function displays a menu screen and
43 prompts the user to select a menu item. The
44 user's character input is converted to an
45 integer value.
46
47 menu() returns an integer value that represents
48 a menu selection.  */
49
50 int menu( void ) {
51     char char_in[2]; /* only input one character to select the menu item */
52     int i;
```

```
53
54     clrscr();
55     printf( "\n\n\n\n\n" );
56     printf( "1.  And data to the data file. \n" );
57     printf( "2.  Display the data file. \n" );
58     printf( "0.  Quit. \n" );
59
60     printf( "\n\nEnter your selection ==> " );
61     gets( char_in );
62     i = atoi( char_in );
63     if( i >= 0 && i <= 3 )
64         return( i );
65     else
66         return( 3 );
67 }
68
69 /* add() — The add() function opens a data file, prompts
70    the user for the product information, and adds
71    the product information to the data file. When
72    all the new items are added to the file, add()
73    closes the data file.
74
75    If add() executes correctly, a value of 0 is
76    returned.  */
77
78 int add( void ) {
79     FILE      *fp;
80     char      more = 'Y';
81     char      name[30];
82     int       count;
83     float     weight;
84
85     clrscr();
86
87     if( ( fp = fopen( DATA_FILE, "ab" ) ) == NULL ) {
88         clrscr();
89         printf( "\n\nData file could not be opened. \n" );
90         exit( 0 );
91     }
92
93     while( more == 'Y' ) {
94         clrscr();
95         printf( "\n\nEnter product name ==> " );
```

```

96     scanf( "%s", name );
97     printf( "\nEnter number of items ==> " );
98     scanf( "%d", &count );
99     printf( "\nEnter product weight ==> " );
100    scanf( "%f", &weight );
101    fflush( stdin );
102
103    fprintf( fp, "%s %d %f", name, count, weight );
104
105    printf( "\n\nEnter another product? Y/N ==> " );
106    more = getche();
107    if( more == 'y' ) more = 'Y';
108 }
109
110 fclose( fp );
111 return 0;
112 }
113
114 /* display() — The display function displays the contents
115    of the data file that was created or
116    appended by the add() function.
117
118    A value of 0 is returned if display()
119    executes correctly. */
120
121 int display( void ) {
122     FILE      *fp;
123     char       name[30];
124     int        count;
125     float      weight;
126
127     clrscr();
128     if( ( fp = fopen( DATA_FILE, "rb" ) ) == NULL ) {
129         clrscr();
130         printf( "\n\nData file could not be opened.\n" );
131         exit( 0 );
132     }
133
134     while( ! feof( fp ) ) {
135         fscanf( fp, "%s %d %f", name, &count, &weight );
136         printf( "\n\n" );
137         printf( "Product name: %s\n", name );
138         printf( "Product count = %d\n", count );

```



```

139     printf( "Product weight = %4.2f\n", weight );
140 }
141
142     printf( "\n\nPress any key to continue." );
143     getch();      /* Original statement " while( ! kbhit() ); " is error */
144
145     fclose( fp );
146     return 0;
147 }

```

程序 fdata.c 中的程序是从 23 行的函数 main() 开始执行的。26 行中的 while 一句重复调用 menu() 函数,直到 menu() 返回 0 值为止。如果 menu() 返回数值是 1 或 2,就调用对应的函数。如果 menu 返回数值是 3,就显示错误信息,程序继续执行。

在 add() 函数中(78~112 行),scanf() 用来存储由用户输入的库存数据。一旦输入库存数据并存储在适当的变量后,fprintf() 就将数据写入到磁盘文件中。

display() 函数(121~147 行)利用 fscanf() 函数从磁盘文件中读取数据,再利用 printf() 函数在屏幕上显示这些数据。

13.10.2 scanf() 函数

在本节中,程序 fdata.c 讨论了 scanf() 函数的应用。我们已知道了一些 scanf() 函数系列格式和如何使用它们,这里再介绍其它可用的 scanf() 函数。

程序 fdata.c 中 98 行这一句是 scanf() 的一种典型用法:

```
scanf( "%d", &count);
```

函数 scanf() 的调用包括两部分:格式串和变参数表。在上面的例子中,格式串是用“%”表达的。这些格式串表示了想读入数据的顺序和类型。在上例中,%d 表示要读入一个整型数据。可以看到,格式串用双引号括起来。

scanf() 函数调用的第二部分是变参数表。它是存放输入数据的地址表。格式串的每一项都对应于变参数表的一个地址。在上面的例子中,输入流中的整型变量存储在地址 &count 中(也就是说,在变量 count 中)。

在使用任意一种 scanf() 格式时,记住下列要求:

- 格式串中的每一特定格式对需要与变量参数表中的一个地址相对应。
- 格式串中变换说明的数据类型与变量表中对应地址的数据类型是一致的。
- 变量参数表是一种地址表,变量参数表中的每一项都是地址,而不是变量。

再看程序 fdata.c 中的下面两句。第一句是 96 行的 scanf() 函数调用,第二句是 98 行的 scanf() 函数调用。

```

scanf( "%s", name);
...
scanf( "%d", &count);

```

在第一句 scanf() 函数调用时,%s 变换说明对应一组要输入的字符串。输入的字符串存储在变量 name 指定的地址中。当使用无下标的数组名时,重新调用这个变量,就得到该数组的地址。在第一句 scanf() 中,name 给出了输入数据的存储地址。

第二句 `scanf()` 与第一句不同的是:变量参数表中使用了地址操作符 `&`。当和一个变量一起使用一元地址操作时,就返回变量的地址。在第二句 `scanf()` 调用中,求址操作符(`&`)返回变量 `count` 的地址。在 `scanf()` 中,对基本数据类型的变量,必须利用求地址操作来获得变量的地址。

`Scanf()` 函数中的格式串包括以下三个方面:

- 字符(Whitespace character) 有效的空白字符串包含空格符、制表符和换行字符。在格式串中空白字符使 `scanf()` 读取并忽略空白字符,直到碰到非空白字符为止。
- 非空白字符(Nonwhitespace character) ASCII 码中除百分号(`%`)外,均是非空白字符,在格式串中,一个非空白字符使函数 `scanf()` 相应地读入并丢弃一个非空白字符。
- 转换说明(Conversion specifier) 转换说明使 `scanf()` 函数从输入流中读入指定类型的数据。读入的数据存储在变量表指定的地址中。所有转换说明以百分号(`%`)打头。

在下一句中,第一个转换说明表示要读入和存储一个整数。

```
Scanf ( "%d%d", &i, &j );
```

第一个转换说明后的空格字符指明 `scanf()` 函数丢弃所有的空白字符,直到出现非空白字符为止。第二个转换说明将使 `scanf()` 读入和存储另一个整型数。参看附录 B 中用于 `scanf()` 函数的完整的转换说明和选项表。

表 13.8 列出 `scanf()` 系列输入函数并指明它们的用途。

表 13.8 `scanf()` 输入函数系列

函数	描 述
<code>scanf()</code>	与标准输入流 <code>stdin</code> 一起使用
<code>cscanf()</code>	直接从控制台读入数据
<code>fscanf()</code>	从选定的流中扫描数据
<code>sscanf()</code>	从字符串中读入数据
<code>vscanf()</code>	用法和 <code>scanf()</code> 一样,只是 <code>vscanf()</code> 需要一个指针指向变量表,以此来代替变量表
<code>vfscanf()</code>	从指定的流中扫描数据。 <code>vfscanf()</code> 需要一个指针指向变量表。
<code>vsscanf()</code>	从字符串中输入数据,它需要一个指向字符串的指针。

函数 `scanf()` 系列中的所有函数都用于输入格式化的数据。尽管所有的 `scanf()` 函数的基本用途相同,但是不同的函数服务于不同的流。数据正是从这些流中输入的。另外几种函数 `V...scanf()` 为调用 `scanf()` 函数提供选择。这些函数对 `V...scanf()` 的调用是不同的,因为传送函数时,1 是通过指针指向变量表来代替实际变量表的。

在这些格式化的输入函数系列中,`scanf()` 和 `fscanf()` 是最简单的。程序 `fdata.c` 中 `display()` 函数就用了 `fscanf()` 函数,和 `scanf()` 一样,`fscanf()` 输入和存储数据。但是,`fscanf()` 能从任意指定的流中输入数据。程序 `fdata.c` 中的 135 行是:

```
fscanf ( fp, "%s %d %f", name, &count, &weight );
```

在函数 `fscanf()` 的调用中, 第一个参数指定了流, 从该流中读入数据。在上面的例子中, `fscanf()` 从流中读入数据, 这种流是由 `file` 文件类型变量 `fp` 指定的。在程序 `fdata.c` 中, `fp` 是一个变量, 它包含一个指针, 指向与磁盘文件 `data.fil` 相关的流。正是从这个磁盘文件中, `fscanf()` 将所有的数据送给 `display()` 函数。

`Vscanf()` 函数及其派生函数稍复杂一点。程序 `varscan.c` 告诉读者如何使用 `vscanf()` 函数的, 在这个例子中, `vscanf()` 启动一个数据输入例程, 然后仅用一次函数调用为 `scanf()` 类型的函数设定参变量。

`varscan.c` 一个表明 `vscanf()` 函数用途的程序

```
1  /* VARSCAN.C This program shows how the vscanf() function
2      can be used.  */
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <stdarg.h>
7  #include <conio.h>
8
9  int input( char * format, ... )
10 {
11     va_list arg_ptr;
12     int item_count;
13
14     clrscr();
15     printf( "Enter two integers and a string. \n" );
16     printf( "Separate the entries with whitespace "
17         "characters. \n\n" );
18     printf( " ==> " );
19
20     va_start( arg_ptr, format );
21     item_count = vscanf( format, arg_ptr );
22     va_end( arg_ptr );
23
24     return item_count;
25 }
26
27 int main( void )
28 {
29     int i, j;
30     char str[81];
31
32     clrscr();
33     printf( "This program demonstrates the "
34         "vscanf() function. \n\n" );
```

```

35     printf( "Press any key to continue.\n" );
36     getch();      /* Original statement " while( ! kbhit() ); " is error */
37
38     input( "%d %d %s", &i, &j, str );
39
40     clrscr();
41     printf( "i = %d, j = %d\n", i, j );
42     printf( "str = %s\n", str );
43
44     return 0;
45 }

```

在程序 varScan.c 中,函数 38 行的 input() 函数指定了函数 vscanf() 的调用参数。input() 函数的参数是格式串和变量参数表,它们供函数 vscanf() 调用中使用。

Vscanf() 是这样设计的,它能从存取宏指令 va_start 和 va_end 的变参数表中得到参数。在 20 行,va_start 使 rg_ptr 指向传送给 input() 函数的变参数表。第 21 行调用 vscanf() 函数。Vscanf() 函数中的格式化(format)正是传送给 input() 函数的格式串。函数 vscanf() 中的 arg_ptr 参数与宏指令 va_start 支持的 arg_ptr 是一样的。这些参量允许 vscanf() 存取由输入函数 input() 指定的参数。

13.10.3 printf() 函数

与 scanf() 函数一样,printf() 函数包含两个主要部分:格式串和变量参数表。格式串指定了将 printf() 函数输出数据的类型。变量参数将数据提供给输出。

函数 printf() 的变量参数表是可选择的。使用 printf() 时不指定变量参数表完全是合法的。程序 varScan.c 中的 33 行就是这样用的,它没有变参数表:

```
printf( "Enter a correct menu value.\n" );
```

这一行简单地使用 printf() 函数打印一条信息。

如果使用转换说明和变量参数表,应当保证转换说明与变量表一致。格式串和变量参数表的不一致会导致程序混乱。当然,当使用的参变量多于转换说明中所指定的时,程序不会出现任何错误。

printf() 格式串包含以下两方面:

- 平常字符(Plain character) 平常字符可以是任何 ASCII 码,包含在格式串中的平常字符将被拷贝到输出流中。
- 转换说明 是指向 printf() 函数的指令,它将数据按指定类型输出。每一个转换说明从百分号(%)开始。转换说明和选择项的全表见附录 B。

printf() 函数系列包括几种输出函数。表 13.9 列出 printf() 函数系列中几种不同的函数并解释了它们的用法。

表 13.9 printf() 函数系列

函数	描 述
printf()	将输出送给 stdout 流。

续表 13.9

函数	描 述
cprintf()	对输出数据送给当前文本窗口,注意:cprintf()并不将换行符(\n)转换成回车/换行(linefeed)字符。
fprintf()	与 printf()函数用法一样,只是,它将输出送到指定的流。
sprintf()	将输出送给字符串而不送给流。
vprintf()	与 printf()一样,将格式化数据送给标准输出流。但是,vprintf()是与指向变量参数表的指针一起调用的,而不是与变量表本身一起调用的。
vfprintf()	与指向变量参数表的指针一起调用,并将输出送给指定的流。
vsprintf()	将格式化的输出送给字符串,vprintf()中存在一个指针,它指向变量参数表。

注意程序 varScan.c 中 103 行 fprintf() 的用法是:

```
fprintf(fp, "%s %d %f", name, count, weight);
```

fprintf() 将一个字符串、一个整数、一个浮点数送给由 FILE 指针变量 fp 指向的流。在程序 varScan.c 中,fp 指向与磁盘文件 DATA.FIL 相关的流。

对于 printf() 类型函数,变参数表中的每一个参数都是一个表达式,它产生一个对应的值,与格式串的转换说明一致。变量参数表中的参数并不限于可变的标识符。看下面的例子:

```
printf("%d", 2+3);
```

这是一个有效的 printf() 函数调用。表达式 2+3 是一个整型表达式,它与转换说明 %d 相对应。这一行的打印结果是整数 5。

程序 varprint.c 介绍了在程序中,如何使用指向变量参数表指针来调用函数 vprintf() 的。

程序 Varprint.C. vprintf() 应用举例

```

1    /* VARPRINT.C This sample program shows how the vprintf()
2           function can be used to print a program's
3           error messages. */
4
5    #include <stdio.h>
6    #include <stdlib.h>
7    #include <stdarg.h>
8    #include <conio.h>
9
10   void msg_print( char *format, ... )
11   {
12       va_list arg_ptr;
13
14       clrscr();
15       printf( "An error has occurred. \n\n" );
16       va_start( arg_ptr, format );
17       vprintf( format, arg_ptr );

```

```
18     va_end( arg_ptr );
19 }
20
21 int main( void )
22 {
23     int i = 50;
24
25     clrscr();
26     printf( "This is a demonstration of an error message "
27           "report function. \n" );
28     printf( "\n\nPress any key to continue. " );
29     while( ! kbhit() );
30
31     msg_print( "Testing message utility i = %d\n", i );
32
33     return 0;
34 }
```

varprint.c 中的程序介绍 `vprintf()` 如何方便地显示错误信息的。这个程序清单中有一个函数 `msg_print()`，它首先清屏，然后报告出错，传递给 `msg_print()` 的所有信息。通常，执行上述三步信息显示需用两次函数调用。第一次调用函数清屏并报告出错，第二次函数调用就是调用 `printf()` 函数打印诊断信息，使用 `vprintf()` 函数，则可通过一次调用执行上述三种功能。

当调用函数 `msg_print()` 时，它传送一个参数表，类似于给函数 `printf()` 传送参数一样。首先，`msg_print()` 使用 `clrscr()` 函数清屏，其次，`printf()` 打印一条信息，告诉你已经出现错误。最后，`vprintf()` 打印传递给 `msg_print()` 的信息。由于 `vprintf()` 仅需要输出数据的指针，`vprintf()` 能很容易地执行这项功能。传递给 `vprintf()` 的参数就是将参数传递给 `msg_print()` 函数的指针。

在调用 `vprintf()` 函数前，需要一些设置。首先，你需要一个指针，它能把参数传递给 `msg_print()`。12 行表示 `va_list` 类型指针 `arg_ptr`。这个指针能指向变量参数表中的参数。其次，变量参数表的指针必须根据变量参数表初值进行初始化。16 行中宏 `va_start()` 恰当地建立一个参数指针。`va_start()` 需要两个参数：参数指针和最后的固定参数名。在开始使用变量参数表之前，最后固定参数就是最后的参数。

一旦已调用 `va_start()` 和参数指针初始化后，就能调用 `vprintf()` 函数了。要注意到 17 行传递给 `vprintf()` 的第一个参数是传递给 `msg_print()` 的格式化参数。传递给 `vprintf()` 的第二个参数是指向变量参数表的指针，它通过 `va_start()` 进行初始化。一旦 `vprintf()` 具有所需的参数时，它就像 `printf()` 函数一样输出数据。

使用 `vprintf()` 函数之后，一定要调用宏 `va_end()`，来结束对变量参数表的访问。如果使用变量参数表指针调用 `va_end()` 宏失败，程序就会出现异常和不确定。

13.10.4 使用字符 I/O 函数

这一节介绍如何使用 Turbo C 字符 I/O 函数，这两组函数是：一次化输入/出单个字符

的函数和一次输入/出一个字符串的函数,表 13.10 概述了字符 I/O 函数和它们的用法。

表 13.10 字符 I/O 函数

函数	描 述
<code>fgetc()</code>	基本字符输入函数,从指定的流中一次读入一个字符。
<code>fputc()</code>	基本字符输出函数,指定字符数据要写入的流。 <code>fputc()</code> 一次写入一个字符。
<code>fgetchar()</code>	<code>fgetc()</code> 的派生函数,它自动地从 <code>stdin</code> 流中读入数据。
<code>fputchar()</code>	在 <code>fputc()</code> 函数用法基本一样,不同之处是 <code>fputchar()</code> 将所有数据写入到流 <code>stdout</code> 中。
<code>fputs()</code>	串输出函数,它需要一个参数,指向数据输出的流。

`fgetc()`函数从指定的流中读入单个字符。下面是使用 `fgetc()`函数的格式:

```
int fgetc ( FILE * stream );
```

由 `fgetc()`返回的值是读入字符的整型值。即使返回值是一个整型数,将返回值赋给 `char` 类型变量也是合法的而且是常用的。例如看下面码段中的 `fgetc()`语句。

```
char c_in;
...
c_in = fgetc ( my_stream );
```

这是完全可行的方法,从 `my_stream` 中读入一个字符并将该字符值赋给 `c_in`。

`fgetc()`仅有的一个参数是指向一个流的指针。指针指向的流就是 `fgetc()`读入数据的源。这个流必须在 `fgetc()`函数调用前打开。

`fputc()`是 `fgetc()`的镜像函数。`fgetc()`从流中读入单个字符,而 `fputc()`将单个字符写入到流中。注意下面 `fputc()`函数声明的格式:

```
int fput ( int c, FILE * stream );
```

`fputc()`函数需要两个参数:要输出的字符和字符将被写入的流。`fputc()`函数调用的第一个参数是要输出的字符的整型值。你能顺利把 `fputc()`作为一个参数传递给字符类型的目标函数。字符类型的参数变换成一个整型参量。

`fputc()`函数调用中的第二个参量是将写入数据的流的指针。在调用 `fputc()`之前这种流应当以写入("W")或追加("a")方式打开。

程序 `Chario.c` 介绍了如何使用 `fgetc()`和 `fputc()`函数读入磁盘文件并在视频显示器上显示文件的。

程序 `Chario.c`, 一个使用 `fgetc()`和 `fputc()`的程序

```
1  /* CHARIO.X This program demonstrates the use of the
2     character I/O functions fgetc() and fputc().
3     The program opens a text file and uses the
4     character I/O function to read and display
5     the file. */
6
```

```
7
8  #include <stdio.h>
9  #include <stdlib.h>
10
11  int main( void )
12  {
13      FILE * file_ptr;
14      char xfer_char;
15
16      if( ( file_ptr = fopen( "text.dat", "rt" ) ) == NULL ) {
17          clrscr();
18          printf( "Could not open data file. \n" );
19          printf( "Calling the exit() function. \n" );
20          exit( 0 );
21      }
22
23      do {
24          fputc( ( xfer_char = fgetc( file_ptr ) ), stdout );
25      } while( xfer_char != EOF );
26
27      fclose( file_ptr );
28      return 0;
29  }
```

Chario.c 的程序的中心是 23~25 行的 do_while 循环,只要 xfer_char 变量不包含 EOF 值,do_while 循环就调用 fputc() 函数。

这个循环中只有一条语句,即 fputc() 函数调用。初看起来,24 行 fputc() 函数较复杂,但是仔细看看,fputc() 调用是十分简单的。

每一个 fputc() 调用需要两个参数:输出字符的整数值和一个输出字符所给的流的指针。在 24 行,fputc() 函数调用的第二个参数是指向标准输出流的指针。stdout 流指针指明:fputc() 将数据写到视频显示器,一次写一个字符。

fputc() 函数调用的第一个参数是另一种函数调用——调用 fgetc() 函数。通常 fgetc() 读入一个字符并返回它的整数值。由 fgetc() 返回的整数值与 fputc() 输出的值是一样的。

在 24 行,由对 fgetc() 返回的变量 xfer_char 的赋值是这样的,它决定了 do_while 循环何时停止。当由 fgetc() 返回的值是文件的终止符时,do_while 循环结束。

fgetchar() 是 fgetc() 函数的特殊转换形式。fgetc() 需要一个参数,它指明数据读入的流。但是 fgetchar() 自动从 stdin 流中读入数据。然后,fgetchar() 从键盘中读入数据。看下面的例子:

```
Char in;
...
in = fgetchar();
```

在这个码段中,fgetchar() 函数从键盘中读入一个字符,然后将该字符赋给字符(char)

型变量 in。

fputc()与fgetc()用法基本一样,不同之处是fputc()将数据送给由系统设定的流stdout。下面是fputc()函数定义的格式:

```
int fputc ( int c );
```

传递给fputc()的整型值是将要输出的字符的值。如果fputc()顺利写入字符,由fputc()返回的值与输出字符的值是相同的。如果fputc()函数写入字符失败,语句就应当换成下面的形式:

```
fputc xfer char = fgetc (file_ptr);
```

与程序 Chario.c 中程序执行时结果是一样的。fputc()函数调用要使用fgetc()读入字符,然后输出由fgetc()返回的值。

这一节最后讨论的最后两个函数是fgets()和fputs(),它们执行串操作而不是执行字符操作。fgets()从指定的流中读入数据,fputs()将串写入到选定的流中。

使用fgets()函数时用下面的格式:

```
char * fgets ( char *s, int n, FILE * Stream );
```

fgets()函数调用的第一个参数是一个指针。它指向一个串,在该串中fgets()存储要读入的串。注意下面的码段:

```
Char c array[81];
```

```
...
```

```
fgets ( c array, 81, _stream );
```

fgets()调用中的c_array参数指向字符数组c array。

fgets()函数调用的第二个参数。告诉你fgets()何时停止从输入流中读入字符。第二个变量n表明当n-1个字符已经读入时,fgets()停止从输入流中读入数据。如果遇到换行字符,fgets()在n-1个字符输入之前停止读入字符。

fgets()函数的第三个和也是最后一个参数是一个指向流的指针,fgets()从其流中读取数据。流指针具有一个FILE数据类型。由fgets()返回的值是一个指向串的指针,也是一个空值。如果fgets()顺利读入一个字符串,fgets()返回一个指向串(字符数组)的指针,在该字符串中存储输入串。如果fgets()遇到文件结束符,就返回一个空值。

fputs()函数是fgets()的反函数。fputs()将串写入到选定的任意流中。使用fputs()函数时,使用下面的格式:

```
fputs( const char *s, FILE stream );
```

第一个参数是指向字符数组的指针,该字符数组包含了要输出的串第二个参数是一个指向流的指针,输出送给该流。strio.c介绍了如何使用fgets()和fputs()函数。

strio.c. 一个使用fgets()和fputs()串I/O函数的程序

```
1 /* STRIO.C This program demonstrates the use of the
2 string I/O functions fgets() and fputs().
3 The program opens a text file and uses the
4 string I/O function to read and display
5 the file. */
6
7
```

```
8    #include <stdio.h>
9    #include <stdlib.h>
10
11    int main( void )
12    {
13        FILE * file_ptr;
14        char store[256];
15
16        if( ( file_ptr = fopen( "text.dat", "rt" ) ) == NULL ) {
17            clrscr();
18            printf( "Could not open data file.\n" );
19            printf( "Calling the exit() function.\n" );
20            exit( 0 );
21        }
22
23        clrscr();
24        while( NULL != fgets( store, 256, file_ptr ) )
25            fputs( store, stdout );
26
27        fclose( file_ptr );
28        return 0;
29    }
```

程序 `strio.c` 是 `dir.c` 的一种变换形式,在 `dirio.c` 中,程序使用一次函数调用写一组整数字符串,而在 `strio.c` 中,输入和输出数据时,一次只运行一个字符。

`dirio.c` 中,在 24 行和 25 行有控制语句。这两行组成循环语句,一次从 `file_ptr` 指的磁盘文件中读入一串。当读入串后,串就被送到 `stdout` 流,然后显示在视频屏幕上。

`dirio.c` 中,24 行 `fgets()` 从 `file_ptr` 指定的流中读满 255 ($256-1=255$) 个字符。读入的字符放在 `store` 指向的数组中。如果 `fgets()` 函数没有遇到文件终止符,程序就执行 25 行 `fputs()` 语句。`fputs()` 函数将 `store` 指的串写入到 `stdout` 流中。

13.11 使用文件控制函数

在使用本章讨论的文件 I/O 函数之前,需要正确打开所使用的文件。否则,就不能执行所期望的 I/O 函数。这一节介绍如何使用这些函数来打开和访问磁盘驱动器中的文件。本节也介绍了当不需要这些文件时如何关闭和消除它们。

这一节的最后一部分介绍了如何控制与文件相关的缓冲区。这样就能更好地控制系统源的使用和 I/O 函数性能的使用。

13.11.1 开文件、关文件和控制文件

在访问文件之前,必须打开文件。打开一个文件执行两项基本功能。首先,打开一个文件决定了文件能执行什么类型的 I/O 功能。换句话说,文件打开的方式决定了能否将数据读

写到文件中。其次,打开文件就将流和文件联系起来。流用来代表文件并对数据进行文件的传输。

在这一章的程序中,数据文件通常在文件 I/O 函数调用之前打开。用以打开文件的函数 `fopen()`,`fopen()` 函数的原形格式如下:

```
FILE * fopen ( const char * filename, const char * mode );
```

`fopen()` 函数有两个参数,它们指定文件名和给文件存取模型。文件名就是你所熟悉的正规 Dos 文件。模型参数告知 `fopen()` 函数,文件能否读入、写出,或者两者兼备。文件模型也决定了是否将文件看作一个流或文本文件。

下面 `fopen()` 语句用以打开文件 DATA.FIL:

```
file_ptr = fopen ( "DATA.FIL", "r+" );
```

因为文件名没有指定路径,`fopen()` 试图在当前目录下打开 DATA0.FIL。但是,如果前边 `fopen()` 语句是:

```
file_ptr = fopen ("c:\\\\DATA.FIL", "Y+" );
```

`fopen()` 语句将试图在 C 盘根目录下打开文件 data.fil。注意第一个参数,也就是文件名,是引号中 `fopen()` 函数调用时关闭的。使用引号是因为 `fopen()` 期望,串作为文件名参数(串变量 identifier 在这里也能用——编译给参量产生地址值)。也要注意两个后缀字符(\\)用来指明进入根目录的路径。因为后缀字符是特殊的编辑字符,在一行中必须使用两个后缀字符,这样,实际的后缀字符就包含在串中。

第二个参数“r+”是模型参数,它决定了访问文件的方式。对于文件名串,你能定义一个模型串变量并使用标识符。前面表 13.5 列出了 `fopen()` 使用的模型参数值。

由 `fopen()` 函数返回的值是函数 `fopen()` 与文件联系的流的指针。I/O 函数用这个指向流的指针来访问文件。下面的码段调用 `fopen()` 并将返回值赋给其它 I/O 函数将要使用的指针:

```
FILE * file_ptr;
```

```
...
```

```
if ( file_ptr = fopen("myfile.txt", "rt" ) ) == NULL ).
```

```
printf("could not open file\n");
```

该码段的第一行声明了一个能指向流的指针。第二行调用 `fopen()` 函数,由 `fopen()` 返回的值赋给变量 `file_ptr`。`file_ptr` 指向与 `myfile.txt` 相关的函数。在该码段的第二行,将 `fopen()` `myfile.txt` 返回值与 `NULL` 相比较。如果 `fopen()` 不能顺利打开该文件,`fopen()` 就返回空值。如果返回空值,就执行该码段的第三行 `printf()` 语句。`printf()` 语句告诉用户文件没有打开。

`freopen()` 函数用来将其它文件与打开流联系起来。`freopen()` 函数的声明与 `fopen()` 基本一样,不同的是 `freopen()` 具有一个额外参量,这与我们看到的函数原形一样:

```
FILE * freopen (const char * filename, const char * mode, FILE * stream ),
```

这里文件名和模型参数的使用规则与 `fopen()` 函数一样。流参数是一个指向当前打开流的指针。

`freopen()` 将文件名参数指定的文件与当前流参数指向的打开流联系起来。为调用 `freopen()` 时,就关闭由流参数列出的流。其后所有对该流的访问都转向文件名参数指定的

文件。

我们可以使用 `freopen()` 函数重新定向 `stdin`、`stdout` 和 `stderr` 流。下面的语句表明 `stderr` 流是如何重定向磁盘文件 `error.dat` 的：

```
If ( freopen ( "ERROR.DAT", "W", stderr ) == NULL )
    printf ( "could not redirect stderr.\n" );
```

文件使用完后，应当关闭它，正确关闭文件有助于防止数据丢失。如果执行完程序而没有关闭所使用的文件，就不能保证返回时那些文件没有丢失。

Turbo C 为关闭文件提供了两个函数：`fclose()` 和 `fcloseall()`。`fclose()` 关闭单个流并要求一个参数指向所要关闭的流。下面的语句关闭由 `file_ptr` 指向的流：

```
fclose ( file_ptr );
```

当调用 `fcloseall()` 时，要关闭所有的打开流（标准流 `stdin`、`stdout`、`stderr` 和 `stdaux` 除外）。`fcloseall()` 不需要任何参数。下面的语句关闭已打开的任何流：

```
fcloseall();
```

此时仍保持打开的流仅仅是程序开始时 Turbo C 打开的标准流。

当程序越来越大和越来越复杂时，读者可能会发现需要使用临时磁盘文件来存储数据。Turbo C 提供两种流，它们能安全比较容易地完成建立临时文件的任务。两个函数都能保证建立当前的文件具有各自独特的文件名并不会覆盖你已建立的任何文件。临时文件函数是 `tmpfile()` 和 `tmpnam()`。

`tmpfile()` 函数建立临时二进制文件并以修改方式打开文件。`tmpfile()` 不需要任何参数但是返回一个指向已打开的流的指针。下面的语句告诉读者如何调用和使用 `tmpfile()`：

```
FILE * file_ptr;
...
if ( ( file_ptr = tmpfile() ) == NULL )
    printf ( "Temporary file was not created.\n" );
```

当流关闭或程序结束时，由 `tmpfile()` 函数建立的临时文件就自动删除了。

`tmpnam()` 函数建立一个独特的文件名。`tmpnam()` 能产生 65535 个不同的文件名。产生的文件名能被 `fopen()` 函数用来创建临时数据文件。

`tmpnam()` 用一个空参数或一个字符数组指针支持临时文件名，如果参数是字符数组的指针，这个数组至少有 `1_tmpnam` 字符那么长。`1_tmpnam` 是一个值，是在 `stdio.h` 抬头文件内定义的。如果 `tmpnam()` 具有一个参数，它是串指针，`tmpnam()` 以串形式存储独特的文件名。如果 `tmpnam()` 有空参量，`tmpnam()` 就将文件以内部静态结果存储起来并将指针返回到文件名中。以下语句说明了函数 `tmpnam()` 的用途：

```
Char file_name [1_tmpnam];
int i;
...
for ( i=0; i <= 50; i++ ) {
    tmpnam( file_name );
    printf( " %s\n", file_name );
}
```

这一代码段是 `for` 循环语句，它调用 `tmpnam()` 函数 50 次。每次调用 `tmpnam()` 函数，文件名都存储在数组 `file_name` 中。每经过一次循环，就显示一次 `file_name` 的新值。

remove()函数具有从磁盘文件中删除文件的功能。remove()函数共需要一个字符串,就是要删除文件的名字。当remove()调用一个有效文件名时,该文件就被删除了。注意下面的例子:

```
remove ( "MYTEXT.DAT" );
```

该remove()函数就从当前磁盘驱动器中删除文件mytext.dat。

rename()函数用于给文件更名,它需要知道旧文件名和新文件名。这两个文件名参数以字符串形式传递给rename()。例如,语句

```
rename ("C:\\\\AUTOEXEC.BAT", "c:\\\\AUTOEXEC.OLD" );
```

将文件名autoexec.bat转换成autoexec.old。

如果rename()参数表中给出磁盘标识符,则二者磁盘标识符必须一致。但是,旧文件名参数中的目录不必与新文件名参数中的目录名一致。如果目录名不同,文件就从旧文件参数表列的目录中移到新文件参数表列的目录中。

13.11.2 控制文件缓冲区

在前面“设备和文件”一节中,我们介绍了缓冲区。我们知道了缓冲区是存储器中的一块,用以进行和设备间的发送接收数据。缓冲区能提高程序的性能,这是因为缓冲区在与计算机相连的慢速设备和所使用程序的高速存储器之间设置了一个速度匹配机制。这一节介绍如何使用Turbo C的控制I/O缓冲区函数。

第一个I/O缓冲区的函数是setbuf()函数。以下是setbuf()说明的句法:

```
void setbuf ( FILE * stream, char * buf );
```

stream参数是一个指向流的指针,I/O缓冲赋给该流,buf参数是一个指向字符数组的指针,用作流的缓冲区。如果vuf参数是一个空值,流就被非缓冲区。如果buf参数指向一个参数数组,字符数组必须至少有BUFSIZ字节长。BUFSIZ在stdio.h抬头文件里定义了。

setbuf()函数可在下面任何一种情况出现时调用:

- 流已创建。
- 调用fseek()后。
- 流被非缓冲化过。

注意下面的代码段:

```
Char io_buf [BUFSIZ];
FILE * file_ptr;
Char in_char;
if ( ( file_ptr = fopen ( "C:\\\\AUTOEXEC.BAT", "rt" ) ) == NULL )
    printf ( "Could not open file. \n" );
setbuf ( file_ptr, io_buf );
do {
    fputc ( in_char = fgetc ( file_ptr ) );
} while ( in_char != EOF );
```

以上语句介绍了如何建立和使用setbuf()函数。第一行说明将字符数组作为一个缓冲区,字符数组说明为SUFSIZE字符长。打开流后,立即调用setbuf()函数。一旦调用setbuf()函数,读入的数据将全被缓冲化。

Turbo C 提供的第二个 I/O 缓冲函数是 `setvbuf()` 函数。`Setbuf()` 将在所使用的缓冲区的类型和缓冲区的空间分配上提供更多的控制。`Setvbuf()` 函数有如下的说明格式:

```
int setvbuf( FILE (stream, char * buf, int type, size_t size);
```

`Stream` 参数是一个想要缓冲化的流的指针, `buf` 参数是一个用于该流的缓冲区的指针。如果 `buf` 参数是空值, 就调用 `malloc()` 函数申请缓冲区。`Size` 参数告诉 `malloc()` 函数给缓冲区分配多少空间。

`type` 参数表明缓冲区化的类型, 该参数可以是如下值之一:

`_IOFBF` 这个值使文件全部缓冲化。

`_IOLBF` 这个值表示文件将被行缓冲化(全部文本行都被缓冲化, 直到遇到行结束时为止)。

`_IONBF` 该文件将被非缓冲化。

`Size` 参数指定了缓冲区的大小。这一参数, 应当大于 0 并且小于或等于 32767。如果是 `_IONBF` 类型, 就越过 `Size` 参数。

现在看下面代码段:

```
FILE * file_ptr;  
Char in_char;  
if ( ( file_ptr = fopen( "C:\\\\AUTOEXEC.BAT", "rt" ) ) == NULL )  
    printf( "could not open file.\\n" );  
setvbuf( file_ptr, NULL, IOLBF, 256 );  
do {  
    fputc( in_char = fgetc( file_ptr ) );  
} while ( in_char != EOF );
```

该代码段所示是, 使用 `setvbuf()` 文件缓冲函数代替 `Setbuf()` 后的程序代码段。注意缓冲区指针参数已置到 `NULL`。空缓冲区指针使 `setvbuf()` 自动调用 `malloc()` 函数来分配缓冲区空间(这里是 256 字节)。在该代码段中, `setvbuf()` 函数已经为 `file_ptr` 指定的流生成 256 字节的行缓冲区。

第三个 I/O 缓冲区函数是 `fflush()` 函数。`fflush()` 与输出缓冲区化的流一起使用。当调用 `fflush()` 时, 缓冲保存的所有数据就写入到文件中。`fflush()` 需要一个 `FILE` 类型参数, 它指向被清除的流。

13.12 使用直接文件 I/O 函数

本章已介绍了 I/O 函数的简单使用方法。这些 I/O 函数使得读者能快速而比较容易地写程序。但是, 这些简单函数不能象直接文件 I/O 函数(direct file I/O 函数)一样给出许多对 I/O 事件和性能的控制。

什么是直接文件 I/O? 直接文件 I/O(或简单说成直接 I/O)是一种文件输入/输出, 该输入/输出能直接指到所需要的文件的那一部分, 而跳过任何插入文件数据。然后, 直接文件 I/O 函数类为文件定位(positioning)、读(reading)、写(writing)块数据(通常指文件记录)提供支持。

因为直接 I/O 通常处理固定长度数据块和记录, 所以直接 I/O 通常用于二进制文件。但

是,在文本中使用直接 I/O 函数也是可以的。直接 I/O 函数使我们能更精确地读写文件数据。使用直接 I/O 函数也能增强文件 I/O 的性能。这一节介绍直接函数及其用法。

13.12.1 理解直接 I/O 概念

开始使用直接 I/O 函数前,要更进一步地理解计算机所做的工作。然后应当使用文件时需要跟踪的信息的类型。

- 文件起始位置。Turbo C (以及其它 ANSI 标准 C 编译器)记录文件的位置,它是以相对于文件起始也称文件源的相对偏移量记录的。文件源的相对偏移位置是零。
- 当前的文件位置。当前文件位置是下一次读入或写出的文件位置。任何时候,都能使用位置报告函数记录当前文件位置,这将本章后面讨论。
- 文件结束位置。文件结束位置是指超出于文件所使用的最后一个字节后的位置。当读写到了文件结束位置时,一个文件结束指示就转向流目标。当超出文件结束位置时,读文件是没有意义的,但是能确定文件的结束位置并继续将数据写入到扩展文件中(给文件写追加数据)。
- Turbo C 数据类型大小的细节。在本书前面已看到,每一数据类型能占用不同数量的存储器。我们需要知道所使用的数据的类型,以便决定要读写字节的数目(尤其是在二进制方式中)。
- 缓冲区或正在使用的缓冲区的位置和大小。人工跟踪缓冲区的位置和大小是没有必要的,除非你自己来做缓冲区分配,如本章前面所介绍的那样。
- 与数据文件相关的流,该流(也是对象类型 FILE)就是本表中提到的大量信息存储的地方。

本章前面介绍的 I/O 函数,保持跟踪了上述的大部分信息,可以为你所用。但是为了有效调用直接 I/O 函数需要访问这些信息。尽管使用这些直接 I/O 函数意味着必须更多地留意程序运行的方式,但是这些函数将会使我们更准确地控制数据的输入输出。

现在介绍将字符数据写入到磁盘文件中时的情况。打开一个文件需要操作系统(DOS, UNIX 或其它系统)在磁盘上开辟一些空间(也就是给文件分配空间)。磁盘上被分配的空间的起始位置也是文件的起始位置,不过,程序不必知道磁盘物理地址。所有程序所做的工作就是从文件源进行写操作,操作系统完成将对地址转换成物理地址的转换。

为程序将一个数据字符(一个字节的的数据)写到文件中后,输出库函数自动增加文件位置指针,该指针就是在文件(file)流对象中记录的数值,指示在该文件中的相对地址。如果程序需要多个字节才能写入,位置指示符就作相应的校正。

当程序将数据写到文件中后,就必须调用文件关闭函数。关闭函数使寄存器中的数据全部写出,并要求操作系统记录文件包含多少数据(在文件的目录进入和磁盘的文件的分配表中)。这样,程序结束时,不会丢失任何数据。前面写入的所有数据依旧存在那里,等待重新使用它。

当然我们能打开刚创建的文件,在该文件中,使用本章已介绍的函数,进行读写操作。但是如果仅想读入文件中最初 10 个字节和第 10 个字节,那该如何办呢?读入前 10 个字节是很容易的:打开文件,并和通常情况一样读入 10 个字节。但是,读入最后 10 个字节,必须使用一种文件定位(file positioning)函数在文件结束前 10 个字节处建立文件位置指针。这样就

能读入最后 10 个字节。如果文件不希望读入更多数据,输入库函数就返回一个文件结束指针(indicator)。对于文件结束指针,有一个宏,名字是 EOF,在 stdio.h 定义。和其它所有系统一样在 Turbo C 中 EOF 的值是 -1 (0XFFFF),所以不容易和一个有效的返回值混淆(直接 I/O 函数返回读写的字节的数目)。

当一个程序以前面所描写的方式直接在文件中读写时,文件通常被看作一个直接文件(direct file)。这个术语是直接存取文件(direct file)的缩语,意味着不用读写文件前面的数据就能读写文件的任一部分。在某一时刻,能跳过程序的某一部分而反运行那些需要的部分。注意,在一个程序里作为直接文件的文件,在另一个程序中可以作为正规文本文件。这是完全可行的,只要文件的内部结构是一个文本文件的内部结构。

既然已知道一些直接 I/O 函数的功能,就可以学习在程序中如何使用这些函数。

13.12.2 读写直接文件

在磁盘驱动器上快速地进行数据块移动方法是使用 fread()和 fwrite()函数。fread()从文件中读数据并将数据放在存储器的某一区域中。fread()函数通过刚读入的字节的数目自动移动文件位置指针(也就是,更新当前文件位置)。然后,下一次读操作将从文件中输入下一串数据字节,除非使用文件位置函数改变当前位置为止。fwrite()类似于 fread(),但它是从存储器中取数据,将数据写到磁盘文件中。fwrite()也自动移动文件位置指针。

dirio.c 介绍了 fwrite()和 fread()函数,参看下面的程序

dirio.c. 使用 fread()和 fwrite()直接 I/O 函数的实例程序

```
1  /* DIRIO.C   This program uses the fread() and fwrite()
2             functions to create a file that holds an
3             array of double floating point numbers,
4             write the array to disk and then read
5             the array back to memory.   */
6
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <conio.h>
10
11 int main( void )
12 {
13     FILE *file_ptr;
14     double x[3] = { 12.3, 45.6, 78.9 };
15     double y[3];
16
17     clrscr();
18     if( ( file_ptr = fopen( "mynum.dat", "w+b" ) ) == NULL ) {
19         printf( "Could not open file. \n" );
20         printf( "Exiting program. \n" );
21         exit( 0 );
22     }
```



```
23
24     if( fwrite( x, sizeof( x ), 1, file_ptr ) == 1 )
25         printf( "Successful write. \n" );
26     else
27         printf( "Unsuccessful write. \n" );
28
29     rewind( file_ptr );
30
31     fread( y, sizeof( y ), 1, file_ptr );
32
33     printf( " %6.3f, %6.3f, %6.3f\n", y[0], y[1], y[2] );
34     fclose( file_ptr );
35     remove( "mynum.dat" );
36     return( 0 );
37 }
```

dirio.c 中 I/O 函数实际上仅占两行。7~23 行的所有代码是变量和文件的设定码。在 24 行调用 fwrite(), 它将所有的 X[] 数组写到磁盘文件中(注意怎么使用 sizeof 操作来确定数组的大小), 在 31 行, 磁盘文件从文件中读出并放在由 fread() 语句的数组中。这两条语句将整个组从主存储器中移到磁盘驱动器中, 并恢复主存储器中的另一个数组。33 行 printf() 语句就是来验证直接 I/O 函数的功能的。

fread() 语句需要四个参数, 它的说明格式如下:

```
size_t fread ( void * ptr, size_t size, size_t nmemb, FILE * stream );
```

fread() 函数的第一个参数是 ptr, 它是一个指向计算机存储器的指针。ptr 所指的数组就是从磁盘上要读的文件存储地址在 dirio.c 中, ptr 指向双精度型数组 Y[]。

注意在 fread() 中指定的 I/O 区域是用户区域, 按文件使用的缓冲区分比配。直接 I/O 函数对缓冲区进行数据的物理转移, 然后拷贝到 fread() 和 fwrite() 函数所指定的区域中。使户 I/O 区域不必和文件缓冲区一样大, 也不必是缓冲区的整型除乘数或除数; 所有数据传送的协调是由库例程完成的。

fread() 的第二个参数是 size, 它指定了读写的每个成员的大小, 在 dirio.c 中, 该大小是由 sizeof(Y) 指定的。这意味着 fread() 将读入和数组 array[] 一样大小的数据。

从一个 fread() 调用到下一个 fread() 调用, 可使用不同大小的数值。例如, 在 dirio.c 中, 可一次读入全部数组, 然后再读入单个数组元(由 Size of double 指定)。因而, 在执行中任一处, 可读入全部记录, 部分记录或者仅一个字节来满足程序的逻辑要求。

第三个参数是 nmemb, 它指出了要从磁盘文件中读取的成员的数目。因为表 13.10 中 Size 参数是寄存器的全部数组的大小, nmemb 参数就置到 1。在这个程序例子中, 我们能很容易地指定 sizeof double 并读入三个成员输入到全部数组中。

第四个也是最后一个参数是 stream 指针, 它指向与正在读的文件相关的流。注意 dirio.c 中由 file_ptr 指向的流是由 18~22 行的 if 语句打开的。也要看到该流是以二进制存取方式打开的。

fread() 返回实际要读取的项的数目的计数。如果到了文件结束处或者有一个错误, 返

回的计数将比需要的少(如果什么也不读入,也许就是 0)。

`fwrite()` 函数是 `fread()` 的倒置函数。`fwrite()` 取出存储在寄存器某一区域上的数据并将它写入到磁盘文件中。与 `fread()` 一样, `fwrite()` 有四个参数, 其说明格式如下:

```
size_t fwrite ( const void * ptr, size_t size, size_t nmemb, FILE * stream );
```

在 `fwrite()` 函数中, `ptr` 参数是一个指向寄存器某一区域的指针, 而数据正是从该区域写到磁盘文件中的。在 `dirio.c` 中, `ptr` 包含数组 `X[]` 的地址。这个数组包含三个拷贝到磁盘文件的双浮点数。

`Size` 参数指明了写到磁盘文件的每个成员大小。`dirio.c` 中的 `Size` 参数是数组 `X[]` 的大小。`size()` 表明 `fwrite()` 利用一次写操作可将所有的数组拷贝到磁盘中。

`nmemb` 函数指定了将要输出的成员 `fwrite()` 的数目。在 `dirio.c` 中, 因为所有数组在一条与语句中拷到磁盘上, 所以 `nmemb` 等于 1。

最后, `stream` 参数是一个指针, 它指向数据要写入的流。

`fwrite()` 函数返回实际要写入项的成员的 `size_t` 计数。如果有错误, 这个计数将比所需要的少或者是 0。

很多时候, `fread()` 和 `fwrite()` 语句用于处理包含字符数据的文件。`prtlab.c` 介绍 `fread()` 函数如何从输出数据库文件中提取信息。

`prtlab.c`. 用 `fread()` 从数据库中提取信息, 打印标号的程序

```
1    /* PRTLAB.C This program reads and prints the records in
2           an address-label database file. The file can
3           contain a variable number of records.
4
5           Even though the number of records in the file
6           is variable, the size of each of the fields in
7           the records is fixed. The size of each of the
8           fields is determined by the typedef structure
9           address_t. */
10
11    #include <stdlib.h>
12    #include <stdio.h>
13    #include <string.h>
14    #include <stddef.h>
15
16    #define MAXLABS 180
17
18    typedef struct {
19        char name[36];
20        char street[36];
21        char apt[36];
22        char city[36];
23        char state[6];
24        char zip[5];
```

```
25     char fill[2];
26 } address_t;
27
28 void null_term( char * str, int length );
29 int compzip( const char * , const char * );
30
31 void main( int argc, char * argv[] )
32 {
33     int j = 0, k = 0;
34     FILE * adrfile;
35     FILE * *labels;
36     char adrname[81];
37     char labname[81];
38     address_t * list;
39
40     if ( argc > 3 ) {
41         puts( "Command format is: prtlab addressfile labelfile" );
42         exit( 0 );
43     }
44
45     if ( NULL == ( list = malloc( MAXLABS * sizeof(address_t) ) ) ) {
46         puts( "Unable to acquire memory for address table." );
47         exit( 8 );
48     }
49
50     strcpy( adrname, argv[1] );
51     strcpy( labname, argv[2] );
52
53     if ( NULL == ( adrfile = fopen( adrname, "rb" ) ) ) {
54         puts( "Can't open input address file." );
55         exit( 8 );
56     }
57     if ( NULL == ( labels = fopen( labname, "w" ) ) ) {
58         puts( "Can't open output label file." );
59         exit( 8 );
60     }
61
62     while ( j < MAXLABS-1 && fread( &list[j], sizeof(address_t),
63         1, adrfile ) ) { list[j].fill[0] = ' ';
64         ++j; /* go to next address record slot */
65     }
66     fclose( adrfile );
67
```

```

68     printf( "Read %d address records.\n", j );
69
70     qsort( list, j, sizeof(address_t), compzip );
71
72     for ( k=0; k<j; ++k ) { /* null terminate line strings */
73         null_term( list[k].name, 36 );
74         null_term( list[k].street, 36 );
75         null_term( list[k].apt, 36 );
76         null_term( list[k].city, 36 );
77         null_term( list[k].state, 6 );
78         null_term( list[k].zip, 6 );
79     }
80     k = 0;
81     while ( k<j ) {
82         fprintf( labels, "%s\n", list[k].name );
83         fprintf( labels, "%s\n", list[k].street );
84         fprintf( labels, "%s\n", list[k].apt );
85         fprintf( labels, "%s %s %s\n", list[k].city,
86             list[k].state, list[k].zip );
87         fprintf( labels, "\n" );
88         ++k;
89     }
90     fclose( labels );
91 }
92
93 void null_term( char * str, int length )
94 {
95     static int i;
96
97     str += length - 1; /* point to last byte
98     for( i=length; i>0; --i ) {
99         if ( *str != ' ' ) {
100             ++str;
101             break;
102         }
103         --str;
104     }
105     if ( i == 0 ) ++str;
106     *str = '\0';
107 }
108
109 int compzip( const char * arg1, const char * arg2 )
110 {

```

```

111     return( strcmp( &arg1[offsetof(address _t, zip)],
112                   &arg2[offsetof(address _t, zip)], 5 ) );
113 }

```

prtlab.c 是一个用来对文件或地址标号进行格式化的应用程序。程序的输入数据是一个文件,它包含一定数目的固定长度的记录。每一固定文件记录是一个完全地址。如果看看 18~26 行的 typedef,就能知道每一记录是如何组织的。

62~64 行含有 fread() 函数,它每次从地址文件中读入一个记录。读入的记录包含在 45~48 行中的 List[] 中,要注意到 62 行 fread() 的 size 参数。fread() 从磁盘文件中读入足够的数据装进 address _t 类型的记录中。读这些信息可以使 fread() 一次返回所有的地址记录。

在数组 List[] 填满后,就调用 qsort() 函数给所有地址表分类。在 70 行调用 qsort()。

一旦给记录分类了,记录中的每个字段通过过多的结尾空白符被破坏了 72~92 行 for 语句要经过表中的每一记录。一旦访问每一记录,就调用来 null _term() 函数冲去记录中的每一空格。

最后,当删除线空白符后,就调用 fprintf() 函数来打印格式化的地址标号。格式化的标号写到磁盘的文件中,以便以后打印这些标号。

13.13 使用文件定位函数

直接 I/O 函数 fread() 和 fwrite() 单独使用时是很有用的。但是,当和文件定位函数一起使用时,这些函数会更加有用。

这些函数和文件定位函数一起使用时,能检测文件中的当前位置和根据需要改变位置。因为能够改变文件中的当前位置,所以能根据需要更灵活地得到想要的数。这一节介绍如何使用这些指示文件中的当前位置的函数和更改文件当前位置的函数。

13.13.1 得到当前文件位置

Turbo C 有两个函数指示当前文件位置:ftell() 和 fgetpos()。ftell() 返回值是当前文件位置,它表示为相对于文件起始位置的字节偏移量。由 fgetpos() 返回的值是服务于流的文件指针的值。fgetpos() 返回的精确值对我们来说无关紧要,仅仅是其它函数如 fsetpos() 才需要这些返回值。

fgetpos() 调用时用到两个参数。fgetpos() 的格式说明就表明了参数和它们的类型:

```
int fgetpos ( FILE * file_ptr, fpos_t * position );
```

file_ptr 参数是一个指向流的指针,根据它能决定当前文件指针值。position 指向一个目标,它保持当前文件位置。getpos.c 介绍了如何在程序中调用 fgetpos()。

getpos.c. 一个使用 fgetpos() 函数的程序

```

1  /*  GETPOS.C      This program uses the fgetpos() function
2           to determine the current file position.    */
3
4
5  #include <stdio.h>
6  #include <stdlib.h>

```

```
7
8   int main( void )
9   {
10      FILE * file_ptr;
11      fpos_t position;
12      char text_out[] = "Extra string stuff.";
13
14      file_ptr = fopen( "junk.txt", "w+" );
15
16      fgetpos( file_ptr, &position );
17      printf( "Current position = %ld.\n", position );
18
19      fwrite( text_out, sizeof( text_out ), 1, file_ptr );
20
21      fgetpos( file_ptr, &position );
22      printf( "New position = %ld.\n", position );
23
24      fcloseall();
25      return 0;
26  }
```

当使用 `fgetpos()` 函数时, 需要注意两点。第一, 确认已经说明了 `fpos_t` 类型变量(见 11 行)。`fpos_t` 类型变量包含当前文件位置值, 第二, 当将位置参数传给 `fgetpos()` 函数时, 确定使用地址操作数(见 16~21 行)。

`ftell()` 函数比 `fgetpos()` 使用起来更简单, `ftell()` 仅需要知道想确定当前文件位置的流的名字。下面的语句介绍了 `ftell()` 的用途。

```
file_ptr = fopen( "junk.txt", "a+b" );
fprintf( file_ptr, "Additional information." );
printf( "The current offset is %ld.\n", ftell( file_ptr ) );
```

在最后一行, 在 `printf()` 语句中调用 `ftell()` 函数。由 `ftell()` 返回的值就是当前文件的位置, 它通过 `printf()` 语句来打印。注意 `ftell()` 给出一个指向流的参数, 而该流是在执行 `fopen()` 函数时打开的。

13.13.2 建立一个新文件位置

知道当前文件位置固然好, 但若能很方便地改变它就更好了。可使用三个标准函数——`rewind()`、`fsetpos()` 和 `fseek()`——来改变文件指针的位置。

`rewind()` 函数设置了文件位置指针, 该指针指到文件的起始位置。通过与流指针参数调用 `rewind()` 来引用 `rewind()`。看下面的例子:

```
rewind( file_ptr );
```

这条语句导致由 `file_ptr` 指向的流的文件位置指针重置到文件起始位置。一般将 `fsetpos()` 函数与 `fgetpos()` 一起使用。`fsetpos()` 根据存储在 `fgetpos()` 的值建立文件位置指针, `fgetpos()` 的说明格式如下:

```
int fsetpos( FILE *file_ptr, const fpos_t *position);
```

传递给 `fsetpos()` 的第一个参数是一个指针,它指向你想要改变其文件位置指针的流。第二个参数是一个 `fpos_t` 类型的变量,它应当包含一个文件位置指针,该指针是由前面的 `fgetpos()` 函数存储的。

下面的语句介绍了 `fgetpos()` 和 `fsetpos()` 是如何一起使用的:

```
FILE file_ptr;  
fpos_t position;  
...  
fgetpos( file_ptr, &position);  
...  
fsetpos( file_ptr, & position);
```

在上面的语句中, `fgetpos()` 在 `position` 变量中存储了文件位置指针的当前值。之后,当调用 `fsetpos()` 函数时,文件位置指针就返回到变量 `position` 指定的位置。

最后,因为 `fseek()` 函数能让将指针移到到文件中的任何位置,所以它是具有很多用途的文件定位函数之一。看下面 `fseek()` 说明格式:

```
int fseek( file *stream, long int offset, int whence);
```

第一个参数指向要使用的流。 `offset` 参数让你指定所要移动的文件位置指示符的字节数的相对位置。 `offset` 参数值与 `whence` 参数值一起来准确确定你所要放置的文件位置指示符的位置。 `whence` 参数可以有如下的任一值:

- `SEEK_CUR` `whence` 将设置到和文件位置指针一样的值。
- `SEEK_SET` `whence` 设置到文件的起始位置。
- `SEEK_END` `whence` 设置到文件的结束处。

`getpos.c` 使用 `fseek()` 函数来计算文件的长度。

`length.c` 用 `fseek()` 函数来计算文件的长度的程序。

```
1  /* FLENGTH.C This program uses the fseek() function to  
2  calculate the length of a file.  */  
3  
4  #include <stdio.h>  
5  #include <stdlib.h>  
6  
7  void main( void )  
8  {  
9      FILE *file_ptr;  
10     long file_size;  
11  
12     file_ptr = fopen( "junk.txt", "r+b" );  
13     fseek( file_ptr, 0, SEEK_END );  
14     file_size = ftell( file_ptr );  
15  
16     fseek( file_ptr, 0, SEEK_SET );  
17     file_size -= ftell( file_ptr );
```

```
18
19     fcloseall();
20     printf( "The file is %ld bytes long. \n", file_size );
21 }
```

在 14 行, `fseek()` 将文件位置指针设置到文件的结束位置。在 15 行, `ftell()` 在 `file_size` 变量中存储文件位置指针的值。在 17 行, 重新调用 `fseek()`。这样, `fseek()` 就将文件位置指针设置到文件起始位置。这样, 就计算出文件结束位置和文件起始位置的差值。该差值就存储在变量 `file_size` 变量中。

13.14 处理文件 I/O 错误

从本书所列举的程序例子不会查出很多文件错误, 因为在这些程序例子中不存在文件错误问题。但是, 当开始写文件成码时, 文件错误会十分严重。需要一种方法, 在造成严重损失之前查出错误并改正这些错误。这一节介绍了一些查出文件 I/O 错误报告文件错误和处理这些错误的 Turbo C 函数。

13.14.1 查出文件 I/O 错误

一个最便利的误差检测函数是 `feof()` 函数。这个函数检查流, 以此来决定是否已到了文件的结束处。如果检测了 EOF 值, `feof()` 就返回一个非零值。否则, `feof()` 返回 0 值, 如使用 `feof()` 函数时, 所要做的就是告诉 `feof()` 所要检测的流。

下面的代码段介绍 `feof()` 函数的通常用法

```
while (! feof ( file_ptr ))
    char_in = getc( file_ptr );
```

这里, 只要 `feof()` 没有查到文件的结束位置, 就执行 `getc()` 函数。

以 `ferror()` 函数测试流来看是否已设立了错误指针。如果 `ferror()` 查出错误, `ferror()` 就返回一个非零值。流的错误标识符能用 `rewind()` 或 `clearerr()` 函数清除。

下面的语句检测由 `file_ptr` 指示的流:

```
if( ferror ( file_ptr )
    printf( "An error has occurred. \n");
```

如果在流中已出现 I/O 错误, 就执行 `printf()` 函数, 给错误出现发出警报。

13.14.2 显示和清除文件 I/O 错误

一旦已确定已出现 I/O 错误时, 就要查出它是什么类型的错误, 如有可能就清除它。可以使用 `clearerr()` 和 `strerror()` 和 `perror()` 完成这些功能。

`clearerr` 函数清除流错误和文件结束指针。使用 `clearerr()` 时, 可用一个指向流的指针简单地调用 `clearerr()`, 该流就是想清除错误的流。语句:

```
clearerr ( file_ptr );
```

清除 `file_ptr` 指向的流的错误及其文件结束标识符。 `strerror` 函数帮助我们诊断出现什么类型的错误。 `strerror()` 取出一个代表错误码的整型参量, 并返回一个指针, 它指向与该错误相关的错误信息串。下面的语句介绍如何通过错误代码 1 到 10 相关的错误信息使用

strerror()函数来显示该错误信息。

```
for ( i=1 ; i<=10; i++)  
    printf( "%S", strerror(i) );
```

在该语句中, strerror()相继返回指针,它指向与错误代码 1 到 10 相连的信息字符串。由 strerror()返回的信息串的指针能直接传递给 printf()函数。

最后, perror()函数引出与当前要打印到标准错误设备上的 errno 值相关的信息。perror()能取出一个串参量,它也将与系统错误信息一起打印出来。

看下面 perror()函数如何用来显示文件 I/O 错误:

```
FILE * file_ptr;  
if( ( file_ptr = fopen( "nonexist.txt", "r+b" ) ) == NULL )  
    perror( "Sorry Charlie" );
```

在这一代码段中,当 nonexistent.txt 不能打开时,就调用 perror()函数。因为存在一个错误, perror()函数就引出一条错误信息打印字符串“Sorry charlie”,作为一个参数传送给 perror(),因此它被置为 perror()所显示的系统错误。

第十四章

屏幕文本和图形程序设计

本章的前面用示例简要说明如何控制 PC 屏幕模式、建立管理窗口和视区的原理,后面介绍进行图形程序设计的技巧。

14.1 图形系统和要素

14.1.1 视频模式

PC 可以配多种视频适配器,既可以是单色适配器(MDA),也可以是具有显示图形能力的彩色图形适配器(CGA),增强的图形适配器(EGA)或大力神(Hercules)单色图形适配器。每种适配器均可在许多模式下进行工作。模式用来定义屏幕显示是 80 列还是 40 列(文本模式)、显示分辨率(图形模式)以及显示属性(彩色还是黑白)。

用户可调用模式定义函数(textmode、initgraph 或 setgraphmode)来定义屏幕的工作模式。

- 在文本模式中,PC 机的屏幕被划分成许多单元(80 或 40 列宽,25、43 或 50 行高)。每个单元由一个字符和一个属性组成。字符就是要显示的 ASCII 码字符,而属性则指明了该字符的显示颜色和亮度等。Turbo C 提供一系列的过程来管理文本屏幕、直接在屏幕上写文本、控制字符的属性。
- 在图形模式中,PC 机屏幕被划分成一个个的像素,每个像素在屏幕上均显示为一个点。像素的多少(分辨率)取决于计算机中安装的视频适配器的类型和适配器工作的模式。用户可以使用 Turbo C 图形库中的函数在屏幕上绘图,比如:画线和作各种形状的图形;以一定的模式填充封闭的区域;控制每个像素的颜色等。

在文本模式下,屏幕的左上角的坐标为(1, 1),x 坐标从左向右增加,y 坐标从上向下增加。在图形模式下,左上角的坐标为(0, 0),x 坐标和 y 坐标的方向和文本模式一样。

14.1.2 窗口和视区

Turbo C 的几个函数用于在文本模式下建立和管理屏幕的窗口(在图形模式下的窗口称为视区)。Turbo C 的窗口和视区管理函数的概述在本节稍后的“在文本模式下编程”和“在图形模式下编程”中说明。

14.1.2.1 窗口的概念

当在文本模式时,窗口是 PC 显示屏幕上的一块矩形区域。当程序向屏幕输出时,输出被限制在活动的窗口内。屏幕的其余地方(在窗口以外)保持原样。

缺省窗口是一个全屏文本窗口。程序可以自己定义小窗口(调用 `window` 函数),这个函数将用屏幕坐标来定义窗口的具体位置。

14.1.2.2 视区的概念

在图形模式中,用户可以在屏幕上定义一个矩形区域,该区域称为视区。当用程序绘制图形时,视区被当作是一个虚拟屏幕,屏幕的其余地方(在视区以外)将保持原样。以屏幕坐标调用函数 `setviewport` 就可定义一个视区。

14.1.2.3 坐标系

除了窗口和视区定义函数外,所有文本和图形模式函数的坐标都相对于窗口和视区,而不是绝对屏幕。在文本模式下窗口左上角是坐标原点,坐标是(1, 1),而在图形模式下,视区左上角的原点坐标是(0, 0)。

14.1.3 在文本模式下编程

在这一节我们将介绍文本模式下使用的函数。详细内容可参阅第十八章的内容。

在 Turbo C 中,直接控制台 I/O 软件包(`cprintf`、`cputs` 等)提供了快速的文本输出、窗口管理、光标定位和属性控制函数。这些函数是 Turbo C 标准库中的一部分,它们的原型在头文件 `conio.h` 中。

14.1.3.1 控制台 I/O 函数

Turbo C 文本模式函数可使用六种文本显示模式。用户系统所能提供的模式取决于视频适配器类型和监视器的类型,可以调用 `textmode` 来定义当前文本模式。在这节的后面我们说明怎样使用这个函数,第十八章也解释了该函数的用法。

这些文本函数可分成四组:

- 文本输出和处理
- 窗口和模式控制
- 属性控制
- 状态查询

1. 文本输出和处理

在这里我们将快速概括一下文本输出和处理函数:

输出和读入文本:

- | | |
|----------------------|-------------------|
| <code>cprintf</code> | 向屏幕作格式化输出 |
| <code>cputs</code> | 向屏幕输出一串字符 |
| <code>getche</code> | 读入一个字符并在屏幕上回显这个字符 |
| <code>putch</code> | 向屏幕输出一个字符 |

在屏幕上处理文本(和光标):

- | | |
|----------------------|------------|
| <code>clreol</code> | 从光标开始清除到行末 |
| <code>clrscr</code> | 清文本窗口 |
| <code>delline</code> | 删除光标所在的一行 |

gotoxy 设置光标位置
 insline 在光标所在行下面插入一行
 movetxt 将文本从屏幕上的一块区域拷贝到另一块区域
 文本块移动:

gettext 将文本从屏幕上的一块区域拷贝到内存中
 puttext 将文本从内存拷贝到屏幕上某一区域

在缺省情况下,屏幕输出程序以整个屏幕文本窗口为对象进行工作,所以无需先设置任何模式就可以写、读和处理文本。用直接控制台输出函数 `cprintf`、`cputs` 和 `putch` 可向屏幕输出文本,而以函数 `getche` 可以用回显的形式输入。全局变量 `_wscroll` 用来控制文本回绕。如果 `_wscroll` 是 1,文本回绕到下一行,必要时自动滚屏;如果 `_wscroll` 是 0,正文回绕到同一行上,不滚屏。缺省时 `_wscroll` 是 1。

一旦文本已显示在屏幕上,可以用 `clrscr` 函数清除当前活动窗口中的显示内容,用 `clreol` 清除一行中的一部分,用 `delline` 删除一整行,用 `insline` 插入一个空行。后面三个函数的操作均和当前光标位置有关,可以用 `gotoxy` 将光标移动到一个特定的位置。也可以用 `movetxt` 将一整块文本从窗口中的一个矩形区域移到另一个地方。

用户也可以用 `gettext` 将屏幕上一个矩形区域存入内存中,并可用 `puttext` 将这块文本放回屏幕上(任何地方)。

2. 窗口和模式控制

在 Turbo C 中有两个窗口和模式控制函数:

`textmode` 设置屏幕为文本模式
`window` 定义一个文本模式窗口

可以用 `textmode` 设置屏幕为各种视频文本模式(这仅受系统监视器的类型和适配器的限制)。它以特定的模式初始化整个屏幕为一个文本窗口并清除任何已有的图象或文本。

当屏幕处于文本模式时,用户既可以以全屏模式输出,也可以设置一个窗口——在这个窗口上接收程序的输出。要建立一个文本窗口,可调用 `window` 函数定义它将占据屏幕上的什么区域。

3. 属性控制

这里简单概述一个文本模式属性控制函数:

设置前景和背景:

`textattr` 同时设置前景和背景(属性)
`textbackground` 设置背景颜色(属性)
`textcolor` 设置前景颜色(属性)

修改亮度:

`highvideo` 设置文本为高亮度
`lowvideo` 设置文本为低亮度
`normvideo` 设置文本为正常亮度

属性控制函数设置当前属性,属性由一个字节的 8 位值来表示:低四位表示前景颜色,另外三位给出背景颜色,最高位是“闪烁允许”位。

以后的输出内容将以当前属性显示。使用属性控制函数,用户可以分别设置前景和背景

颜色(使用 `textbackground` 和 `textcolor`),或调用 `textattr` 同时设置。也可以将字符(前景)定义成闪烁的。在彩色模式中许多监视器将显示真正的颜色,非彩色监视器则将其一部分或全部属性转换成各种灰度或其它视频效果。例如粗体、下划线和反象等。

可以用 `lowvideo` 函数使系统从高亮度前景颜色变为低亮度的(清属性字节的高亮度位),或用 `highvideo` 从低亮度变为高亮度(置属性字节的高亮度位)。使用完 `lowvideo` 或 `highvideo` 以后,也可以用 `normvideo` 函数将亮度恢复为正常。

4. 状态查询

Turbo C 控制台 I/O 函数包括几个用于状态查询的函数。使用这些函数,可以得到在文本模式下窗口的有关信息和窗口内当前光标的位置。

<code>gettextinfo</code>	将当前文本窗口的信息放入 <code>text_info</code> 结构中返回
<code>wherex</code>	给出当前光标 x 坐标
<code>wherey</code>	给出当前光标 y 坐标

`gettextinfo` 函数用文本窗口的信息设置 `text_info`,包括:

- 当前视频模式
- 以绝对屏幕坐标表示窗口的位置
- 窗口大小
- 当前前景和背景颜色
- 光标的当前位置

有时用户可能只需其中的几个信息,而不必获取全部的文本窗口信息,如可以用 `wherex` 和 `wherey` 函数得到光标的位置(相对于窗口)。

光标形状:可以使用函数 `_setcursortype` 改变光标的显示形状。参数值是 `_NOCURSOR`,取消光标;`_SOLIDCURSOR`,给出一个实心块光标;`_NORMALCURSOR`,给出一个通常的下划线光标。

14.1.3.2 文本窗口

缺省的文本窗口是整个屏幕。用户可以调用 `window` 函数将它改变为一个小于整个屏幕的文本窗口。文本窗口可包含多到 50 行和多到 40 或 80 列的窗口。

Turbo C 文本窗口的坐标原点(坐标值的开始点)是屏幕的左上角。窗口左上角的坐标是(1, 1);一个 80 列、25 行文本窗口的右下角坐标是(80, 25)。

一个例子:

假设用户是在 80 列文本模式下工作,并想建立一个窗口,该窗口左上角的屏幕坐标为(10, 8),右下角的屏幕坐标为(50, 21),此时可调用 `window` 函数来设置这个窗口

```
window(10, 8, 50, 21);
```

现在就已经建立了一个文本模式的窗口,尽管到目前为止,从屏幕上什么也看不出来,但如果将光标移到窗口内的(5, 8)处,并在窗口内实际写一些内容,用户就可以发觉确实存在一个新窗口。要完成这项工作,可使用函数 `gotoxy` 和 `cputs`。

```
gotoxy(5,8);
cputs("Happy Birthday, Frank Borland");
```

图 14.1 说明了这点。

14.1.3.3 文本模式类型

可以调用 `textmode` 将监视器设置为几种文本模式之一。枚举类型 `text_modes` 在头文

件 `conio.h` 中定义,其符号名可以直接作为 `textmode` 函数的模式参数而不必使用直接的模式号。但如果要使用这些符号常量,用户必须在源代码中包含下列语句:

```
#include <conio.h>
```

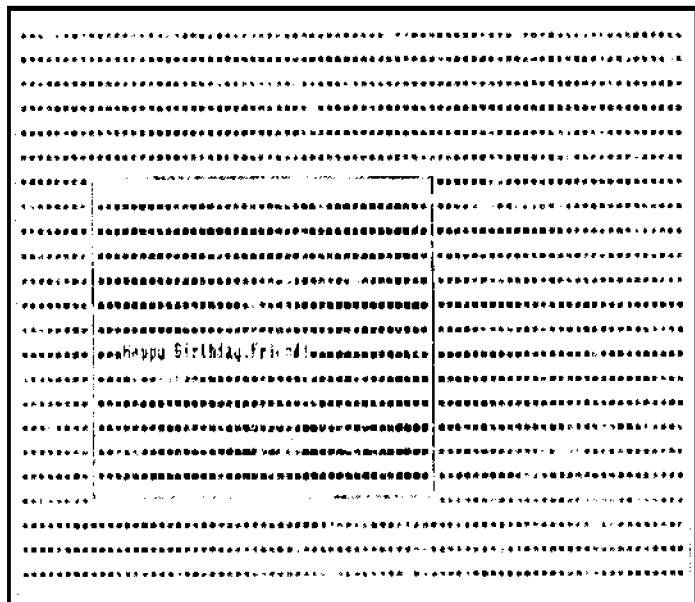


图 14.1 在 80×25 文本模式下的一个窗口

由 `text_modes` 定义的数字值和对应的符号值如下:

符号常量	数字值	视频文本模式
<code>LASTMODE</code>	1	使用上次的文本模式
<code>BW40</code>	0	黑白,40 列
<code>C40</code>	1	16 种颜色,40 列
<code>BW80</code>	2	黑白,80 列
<code>C80</code>	3	16 种颜色,80 列
<code>MONO</code>	7	单色,80 列
<code>C4350</code>	64	EGA,80×43;VGA-80×50

例如下面对 `textmode` 的调用将置彩色监视器为指定的操作模式:

```
textmode(0)      黑白,40 列
textmode(BW80)   黑白,80 列
textmode(C40)    16 种颜色,40 列
textmode(3)      16 种颜色,80 列
textmode(7)      单色,80 列
textmode(C4350)  EGA,80×43;VGA 80×50
```

在以模式 `C4350` 调用 `textmode` 以后,可以使用 `gettextinfo` 函数来确定屏幕的行数。

14.1.3.4 文本屏幕

字符的颜色由字符属性控制,分为前景与背景。

字符的颜色是前景;单元的其余区域颜色是背景。使用彩色视频适配器的彩色监视器可以显示 16 种不同的颜色;单色监视器将把颜色替换成不同的视频属性(高亮度、下划线和反象等)。

包含文件 conio.h 已为不同的颜色定义了符号名。如果使用符号常量,必须在源代码中包含 conio.h。

下表列出这些符号常量和它们相应的数字值。注意只有前 8 种颜色可用于前景和背景,后 8 种颜色(颜色号 8 到 15)只能用于前景(即字符本身)。

符号常量数	字值	前景或背景?
BLACK	0	用于两者
BLUE	1	用于两者
GREEN	2	用于两者
CYAN	3	用于两者
RED	4	用于两者
MAGENTA	5	用于两者
BROWN	6	用于两者
LIGHTGREEN	7	用于两者
DARKGRAY	8	只用于前景
LIGHTBLUE	9	只用于前景
LIGHTGREEN	10	只用于前景
LIGHTCYAN	11	只用于前景
LIGHTRED	12	只用于前景
LIGHTMAGENTA	13	只用于前景
YELLOW	14	只用于前景
WHITE	15	只用于前景
BLINK	128	只用于前景

如果想使字符闪烁,可以将符号常量 BLINK(数字值 128)加到前景参数中,即将属性字节的闪烁位置 1。

14.1.3.5 高性能输出:directvideo 变量

Turbo C 的控制台 I/O 软件包有一个全局变量 directvideo。这个变量控制程序控制台输出是直接使用视频 RAM(directvideo=1)还是通过 BIOS 调用(directvideo=0)。

缺省值为 directvideo=1(控制台输出直接使用视频 RAM)。一般来讲,直接使用视频 RAM 可获得很高的性能,但这要求计算机是 100%与 IBM PC 兼容,显示硬件必须和 IBM 的视频适配器一样。另外,还要注意一点,由于输出数据被直接送到显示缓冲区,所以在配 CGA 显示卡的 PC 上,有可能出现屏幕雪花。置 directvideo=0 可以工作在任何与 IBM 的 BIOS 兼容的机器上,但这里控制台输出是通过调用 BIOS 的中断 10h 实现的,速度显得稍慢了。

14.1.4 在图形模式下编程

在这里我们将简单地总结一下在图形模式中使用的函数。详细内容请参见第十八章。

Turbo C 提供了一个单独的有 70 多个图形函数的库,范围包括高层调用(如 `setviewport`、`bar3d` 和 `drawpoly`)到面向位的函数(如 `getimage` 和 `putimage`)。这个图形库支持各种填充和线型,并提供几种文本字体,可以改变输出字体大小、对齐、水平或垂直定向。

这些函数在库文件 `GRAPHICS.LIB` 中,而它们的原型在头文件 `graphics.h` 中。除了这两个文件以外,图形软件包还包括图形设备驱动器(*.BGI 文件)和笔划字符字体(*.CHR 文件),在下面几节我们将讨论这几个附加文件。

使用图形函数时:

- 如果使用的是集成环境,置 Full Menus 为 On,然后检查 Options | Linker | GraphicsLibrary。当连接程序时,连接器自动连接 Turbo C 图形库。
- 如果使用的是 TCC.EXE,用户必须在命令行中给出 `GRAPHICS.LIB`。例如,如果程序 `MYPROG.C` 使用图形,TCC 命令行应该为:

```
tcc myprog graphics.lib
```

由于图形函数使用 `far` 指针,所以 Tiny 内存模式不支持图形。

各种内存模式共用仅有的一个图形库(比较标准库 `CS.LIB`、`CC.LIB` 和 `CM.LIB` 等,它们各自用于特定的内存模式)。在 `GRAPHICS.LIB` 中的每个函数均为 `far` 函数,并且这些图形函数均使用 `far` 指针。为使这些函数正确地工作,在使用图形的每个模块中加入语句 `#include <graphics.h>` 是很重要的。

14.1.4.1 图形库函数

Turbo C 的图形函数共分成七类:

- 图形系统控制
- 绘图和填充
- 处理屏幕和视区
- 文本输出
- 颜色控制
- 错误处理
- 状态查询

1. 图形系统控制

这里简单概括一下图形系统控制函数:

<code>closegraph</code>	关闭图形系统
<code>detectgraph</code>	检查硬件和决定使用哪个图形驱动器,并推荐使用一种模式
<code>graphdefaults</code>	重新设置怕有的图形系统变量为缺省值
<code>_graphfreemem</code>	释放图形内存;用于连接自定义的过程
<code>_graphgetmem</code>	分配图形内存;用于连接自定义的过程
<code>getgraphmode</code>	返回当前图形模式
<code>getmoderange</code>	对特定的驱动器返回最低和最高的模式号
<code>initgraph</code>	初始化图形系统,并设置硬件进入图形模式

<code>installuserdriver</code>	安装一个销售商提供的设备驱动器到 BGI 驱动器表中
<code>installuserfont</code>	安装一个销售商提供的笔划字体文件到 BGI 字符文件表中
<code>registerbgidriver</code>	注册一个在连接时连接进来的(即用户加载的)驱动器文件
<code>restorecrtmode</code>	恢复原始(在 <code>initgraph</code> 以前)屏幕模式
<code>setgraphbufsize</code>	定义内部图形缓冲区的大小
<code>setgraphmode</code>	选择特定的图形模式,清屏并恢复所有的缺省值

Turbo C 的图形软件包对下述图形适配器(以及完全兼容的)提供了图形驱动程序:

- 彩色图形适配器(CGA)
- 增强图形适配器(EGA)
- 视频图形阵列(VGA)
- 大力神图形适配器(Hercules)
- AT&T400 线图形适配器
- 3270 PC 图形适配器
- IBM 8514 图形适配器

为启动图形系统,首先调用 `initgraph` 函数。`initgraph` 函数将装入图形驱动器并使系统进入图形模式。

可以让 `initgraph` 使用特定的图形驱动器和模式,或在运行时自动检查相连的视频适配器并选择相应的驱动器,这可以通过 `initgraph` 函数的参数来指明。如果 `initgraph` 作自动检查,它将调用 `detectgraph` 去选择一个图形驱动器和模式。如果要告诉 `initgraph` 使用某个驱动器和模式,必须确保相应的硬件存在。如果强迫 `initgraph` 使用不存在的硬件,结果是不可预知的。

一旦装入了一个图形驱动器,可以使用 `getdrivername` 函数得到这个驱动器的名字,使用 `getmaxmode` 函数则可以得知这个驱动器可支持多少种模式。`getgraphmode` 返回当前处于何种图形模式。一旦有了一个模式号,可以使用 `getmodename` 找到这个模式的名字。可以用 `setgraphmode` 改变图形模式,并可以用 `restorecrtmode` 返回到原状态下的视频模式(在图形被初始化之前的)。`restorecrtmode` 使屏幕返回到文本模式,但它并不关闭图形系统(字体和驱动器仍在内存中)。

`graphdefaults` 重新设置图形状态配置(视区大小、绘图颜色、填充颜色和模型等)为初始值。

使用 `installuserdriver` 和 `installuserfont` 可以向 BGI 中增加新的设备驱动器和字体。

最后,在使用完图形系统后,调用 `closegraph` 关闭它。`closegraph` 从内存中装卸驱动器并恢复初始的视频模式(通过 `restorecrtmode`)。

下面是更多的细节说明:

前面的说明概述了 `initgraph` 是如何工作的。这节我们将较详细地说明函数 `initgraph`、`graphgetmem` 和 `_graphfreemem` 的作用。一般来讲,`initgraph` 过程通过为一个图形驱动器分配内存装入一个图形驱动器,然后从磁盘上装入相应的 BGI 文件。这是一种可选的动态装入模式,用户也可以直接将一个图形驱动器文件(或它们当中的几个)连接到最终的可执行文件中。为做到这一点,首先应将一个 BGI 文件转换成一个 OBJ 文件(使用 BGIOBJ 实用程序——具体使用说明在 UTIL.DOC 文件中),然后在源代码中加上一条对

registerbgidriver 的调用(在对 initgraph 的调用之前),以注册这个图形驱动器。当编译程序时,需要将用于注册驱动器的 .OBJ 文件也连接到最终代码中。

在决定使用哪个驱动器以后(通过 detectgraph),initgraph 检查所需的驱动器是否已经注册。如果已注册,initgraph 直接使用内存中已注册的驱动器。否则,initgraph 为驱动器分配内存,并从磁盘上装入这个 .BGI 文件。

使用 registerbgiheader 是一种高级程序设计技术,对学习程序设计的新手我们不推荐这样做。第十八章中也对这个函数作了详细的描述。

在运行期间,图形系统可能需要为驱动器、字体和内部缓冲区分配内存。如果有这个必要,它将调用 _graphgetmem 分配内存,并将在以后调用 _graphfreemem 释放它们。在缺省情况下,这些过程各自简单地调用 malloc 和 free(注:如果使用自己的 _graphgetmem 或 _graphfreemem 函数,用户可能会得到一条“duplicate symbols”的警告信息。不必理会这条警告信息)。

可以通过定义自己的 _graphgetmem 和 _graphfreemem 函数来改变这种缺省模式。如果这样做了,就可以控制图形系统对内存的分配。自己的内存分配过程必须使用同样名字:它们将替换掉在 C 的库函数中具有相同名字的缺省函数。

2. 绘图和填充

这里简单概述绘图和填充函数:

绘图:

arc	画一个圆弧
circle	画一个圆
drawpoly	画一个多边形轮廓线
ellipse	画一个椭圆形的弧
getarccoords	返回最近一次对 arc 或 ellipse 调用时的坐标
getaspectratio	返回当前图形模式长宽比
getlinesettings	返回当前线型、线模型和线宽
line	从(x0, y0)到(x1, y1)作一条直线
linereel	以某个相对距离作一条直线到达一个点
lineto	从当前位置到(x, y)作一条直线
moveto	将当前位置移到(x, y)
moveerel	以一个相对距离移动当前位置
rectangle	画一个矩形
setlinestyle	设置当前线宽和线型

填充:

bar	画一个条形图并填充
bar3d	画一个三维条形图并填充
fillellipse	画一个椭圆并填充
fillpoly	画一个多边形并填充
floodfill	填充一个有界区
getfillpattern	返回用户定义的填充模型

getfillsettings	返回有关当前填充模型和颜色的信息
pieslice	画一个馅饼形的图并填充
sector	画一个扇形并填充
setfillpattern	选择用户定义的填充模型
setfillstyle	设置填充模型和填充颜色

使用 Turbo C 的绘图和填充函数,可以画各种颜色的直线、弧、圆、椭圆、矩形、饼形、二维或三维条形、多边形以及以此为基础组合成的常规和非常规形状的图。可以使用 11 种预定义的模型或自己定义的模型填充任何有界图形(即由这个图形包围着的区域),也可以控制直线的宽度和线型以及控制当前位置的坐标。

调用 arc、circle、drawpoly、ellipse、line、linerel、lineto 和 rectangle 绘图,将只绘出线条,并不作填充。可以用 foodfill 填充这些图形,或使用 bar、bar3d、fillellipse、fillpoly、pieslice 和 sector 将绘图和填充组合在一个步骤内完成。可以使用 setlinestyle 定义所画的线和用于填充的边界线是粗的还是细的,以及它的线型是实线还是点划线等四种,或是用已经定义的某些其它直线模式画线。可以用 setfillstyle 选择一个预定义的填充模型,或用 setfillpattern 定义自己的填充模型。使用 moveto 将当前位置移动到一个特定的位置,以及用 moverel 以一个相对值移动当前位置。

调用 getlinesettings 函数将得到当前的线型和宽度。调用 getfillsettings 函数将得到有关当前填充模型和填充颜色的信息。用户还可以用 getfillpattern 函数来得到用户定义的填充模型。

使用 getaspectratio 函数可得到长宽比率(图形系统为使圆在显示中也是圆的而使用的放大因子,因为在图形模式下每个象素的长宽并不一致,所以在画圆时只保证横竖两个方向的象素数目相同是不行的),并可调用 getarccoords 得到上次画弧或椭圆时的坐标。如果画出的圆不很圆,可使用 setaspectratio 函数对它们进行改正。

3. 处理屏幕和视区

这里简单概述屏幕、视区、图象和象素管理函数。

屏幕管理:

cleardevice	清屏幕(活动页)
setactvepagee	设置用于图形输出的活动页
setvisualpage	设置可见的图形页号

视区管理:

clearviewport	清当前视区
getviewsettings	返回有关当前视区的信息
setviewport	设置用于图形输出的当前输出视区

图象管理:

getimage	在内存保存特定区域的位图象
imagesize	返回用于存储屏幕上一个矩形区域的字节数
putimage	将以前保存的位图象放回屏幕上。

象素管理:

getpixel	得到(x, y)处的象素颜色
----------	----------------

putpixel 在(x, y)绘一个像素

除了绘图和填充外,图形库提供了几个用于管理屏幕视区、图象和像素的函数。可以用 cleardevice 清除整个屏幕,这个过程清除整个屏幕并将当前位置置于视区的原点,但保留其它的图形系统设置(线型、填充和文本类型、调色板、视区设置等)。

根据系统的适配器,可以有一到四个屏幕页缓冲区,这是一段内存区,每个全屏幕图象均以点的形式依次存于其中。用户可以定义哪个屏幕页是活动的(图形函数将它们的输出存于其中)以及显示哪个页,这可以通过分别使用函数 setactivepage 和 setvisualpage 来完成。

一旦屏幕进入图形模式,用户可以调用 setviewport 函数在屏幕上定义一个视区(一个矩形“虚拟屏幕”),以绝对屏幕坐标形式定义视区的位置以及定义是否允许剪切。可以使用 clearviewport 清视区。调用 setviewsettings 可以得到当前视区的绝对屏幕坐标和剪切状态。

用户可以调用 getimage 函数来获取屏幕上的部分图象,调用 imagesize 可以得到在内存中保存这些图象所需的字节数,然后可以用 putimage 函数将保存的图象放回到屏幕上的任何地方。

所有输出函数的坐标(绘图、填充、文本等)都是相对于视区的。用户可以使用 getpixel(返回给定像素的颜色)和 putpixel(绘一个特定的像素)函数来处理单个像素的颜色。

4. 图形模式下文本输出

这里简单概述图形模式文本输出函数:

gettextsettings	返回当前文本字体、方向、大小和对齐模式
outtext	在当前位置输出一个字符串
outtextxy	在特定的位置向屏幕输出一个字符串
registerbgiFont	注册一个连接进来的即用户加载的字体
settextjustify	设置由 outtext 和 outtextxy 使用的文本对齐值
settextstyle	设置当前文本字体、字型和字符放大因子
setusercharsize	设置笔划字体宽高比率
textheight	返回以像素为单位的一个串高度
textwidth	返回以像素为单位的一个串宽度

在图形模式时图形库包含用于文本输出的一个 8×8 位图象的字体和几个笔划字体。

● 在位图象字体中,每个字符以像素矩阵来定义。

● 在笔划字体中,每个字符由一系列的笔划矢量来定义。

当写大字符时,使用笔划字体有明显的优点。因为笔划字体由矢量来定义,当字体被放大时,它仍然保持较好的分辨率和质量。然而当放大一个位图象字形时,矩阵被放大因子所乘;随着放大因子变大时,字符的显示也就变得越粗糙,因为分辨率并非提高,只是将每个像素点放大为方块。对小字符,位图象字体是可以满足需要的,但对大的文本,应该选择笔划字形。

用户可调用 outtext 和 outtextxy 函数输出图形文本,并调用 settextjustify 函数 y 控制输出文本(相对于当前位置)的对齐。settextstyle 函数能用来选择字型、方向(垂直的或水平的)和大小(尺寸)。调用 gettextsettings 函数将得到当前文本设置(它通过 textsettings 结构返回当前文本字体、对齐、放大倍数和方向)。setusercharsize 允许设置笔划字符的宽度和高度。

如果剪切状态是 on,由 outtext 和 outtextxy 输出的所有文本字符串在视区的边界均被

剪切掉。如果剪切状态为 off, 当文本串的任何部分超出屏幕边界时, 这些函数将舍弃位图象字体的输出, 而笔划字体的输出将在屏幕边界被截断。

为确定一个给定的文本串在屏幕上的大小, 可调用 `textheight` (返回以像素为单位的串高) 和 `textwidth` (返回以像素为单位的串宽) 函数。

缺省的 8×8 位图象字体就在图形包中, 所以在运行时它总是存在的。而每个笔划字体均保存在文件 `CHR` 中; 它们可以在运行时装入, 或转换成 `OBJ` (使用 `BGIOBJ` 实用程序) 然后连接到 `EXE` 文件中。

一般来讲, `settextstyle` 过程通过为一种字体分配装入这个字体文件的内存, 然后从磁盘上装入合适的 `CHR` 文件。这种动态装入方案是可选择的, 可以将一个或几个字体文件直接连接到可执行文件中。为达到这个目的, 首先将 `CHR` 文件转换成 `OBJ` 文件 (使用 `BGIOBJ` 实用程序请阅读附录 B), 然后在源代码中调用 `registerbgifont` (放在对 `settextstyle` 的调用以前) 以注册字体文件。连接目标代码时, 需要将用到的笔划字体所在的 `OBJ` 文件连接进来。

使用 `registerbgifont` 是一个高级程序设计技术, 不推荐初学者使用。这个函数的更多细节可参阅在 `UTIL.DOC` 文件中的说明。

5. 颜色控制

这里简单概述一下颜色控制函数:

<code>getbkcolor</code>	返回当前背景颜色
<code>getcolor</code>	返回当前绘图颜色
<code>getdefaultpalette</code>	返回调色板结构定义
<code>getmaxcolor</code>	返回当前图形模式下最大颜色号
<code>getpalette</code>	返回当前调色板和它的大小
<code>getpalettesize</code>	返回调色板查找表的大小

设置一种或多种颜色:

<code>setallpalette</code>	用所指明的颜色改变所有调色板颜色
<code>setbkcolor</code>	设置当前背景颜色
<code>setcolor</code>	设置当前绘图颜色
<code>setpalette</code>	用参数给定的值改变一个调色板颜色

在概述这些颜色控制函数如何工作以前, 先介绍一下彩色在图形屏幕上产生的过程。

● 像素和调色板

图形屏幕由一个像素阵列组成, 每个像素对应屏幕上一个单一的 (有色的) 点。像素的值并不直接定义正确的颜色, 它是对一个称作调色板的颜色表的索引。包含这个像素真实的颜色信息在相应的调色板中的表目中。

这种间接方案有几种含义。虽然硬件可以显示许多种颜色, 但是在任何给定的时间只有这些颜色的一个子集可被显示出来。在任何一个时间可被显示出来的颜色, 其数目等于调色板中的表目数 (调色板大小)。例如在 EGA 中, 硬件可显示 64 种不同的颜色, 但在某一时间只能显示 16 种颜色, 所以 EGA 调色板的大小是 16。

调色板大小决定了一个像素的颜色范围可为 `size-1`。 `getmaxcolor` 函数返回当前图形驱动器和模式所使用的最高有效值 `size-1`。

当讨论 Turbo C 的图形函数时, 实际上颜色是一个像素值, 它是一个对调色板的索引。

只有调色板决定屏幕上真实的颜色。通过处理调色板,可以改变屏幕上显示的实际颜色,即使像素的值(绘图颜色、填充颜色等)没有被改变。

● 背景和绘图颜色

缺省背景颜色总是对应于像素值 0。当一个区域被清除为背景颜色时,这个区的像素值被简单地置为 0。

画线时的绘图颜色是像素所置的值。使用 `setcolor(n)` 选择一种绘图颜色,这里 `n` 对当前调色板来说必须是一个有效的像素值。

● 在 CGA 上控制颜色

由于图形硬件的差异,在 CGA 和 EGA 上实际控制颜色的方法也十分不同,而 CGA 几乎是低档 PC(8086/8088)的标准适配器,所以单独介绍它们。

(1) 低分辨率 CGA

在低分辨率模式,可以从四个预定的四色调色板中进行选择。在任何一个调色板中,只能设置第一个调色板表目,表目 1、2 和 3 是固定的。第一个调色板表目(颜色 0)是背景颜色。这个背景颜色可以是 16 种可能的颜色的任一种(见下面 CGA 背景颜色表)。

调色板号	分配给颜色号(像素值)的常量		
	1	2	3
0	CGA_LIGHTGREEN	CGA_LIGHTRED	CGA_YELLOW
1	CGA_LIGHTCYAN	CGA_LIGHTMAGENTA	CGA_WHITE
2	CGA_GREEN	CGA_RED	CGA_BROWN
3	CGA_CYAN	CGA_MAGENTA	CGA_LIGHTGRAY

通过选择模式(CGAC0、CGAC1、CGAC2、CGAC3)来选择需要的调色板。这些模式使用彩色调色板 0 到彩色调色板 3,如下表所示。CGA 绘图颜色和等价的常量定义在 GRAPHICS.H 中。

为分配其中的一个颜色值为 CGA 的绘图颜色,调用 `setcolor` 并以颜色号或相应的常量名作为参数。例如,正在使用调色板 3,如果想使用 CYAN(青色)作为绘图颜色,则用:

```
setcolor(1);
```

或

```
setcolor(CGA_CYAN)
```

在 `graphics.h` 中定义的 CGA 可用的背景颜色列出下表中:

数字值	符号名	数字值	符号名
0	BLACK	8	DARKGRAY
1	BLUE	9	LIGHTBLUE
2	GREEN	10	LIGHTGREEN
3	CYAN	11	LIGHTCYAN
4	RED	12	LIGHTRED
5	MAGENTA	13	LIGHTMAGENTA
6	BROWN	14	YELLOW
7	LIGHTGRAY	15	WHITE

CGA 的前景颜色和表中列出来的一样。

使用 `setbkcolor(color)` 分配这些颜色中的一种作为 CGA 的背景颜色, 这里 `color` 是上面表中的一个表目。注意, 对 CGA, 这个颜色不是一个象素值(调色板索引), 它直接定义了将放入第一个调色板表目中的实际颜色。

(2) CGA 高分辨率

在高分辨率模式(640×200), CGA 只能显示两种颜色: 一种背景和一种彩色前景。象素的值是 0 或 1。由于 CGA 特有的习性, 前景颜色实际上就是硬件所认为的背景颜色, 必须用 `setbkcolor` 设置前景颜色。

可以前景使用的颜色在前面的表中列出。CGA 使用这些颜色显示所有象素值为 1 的象素。

以这种方法工作的模式是 CGAHI、MCGAMEG、MCGAHI、ATT400MED 和 ATT400HI。

(3) CGA 调色板过程:

因为 CGA 调色板是预定义的, 所以不应该在 CGA 上使用 `setallpalette` 过程, 也不应该使用 `setpalette(index, actual_color)`, 除非 `index=0` (这是一种可选的设置 CGA 背景颜色为 `actual_color` 的方法)。

(4) 在 EGA 和 VGA 上控制颜色

在 EGA 中, 调色板包含全部 64 种 CGA 颜色, 和前面颜色表中给出的一样: 黑是表项 0, 蓝是表项 1, …… , 白是表项 15。在 `graphics.h` 中相应于硬件颜色值的定义有对应的常量, 它们是 `EGA_BLACK`、`EGA_WHITE` 等。可以使用 `getpalette` 得到这些值。

`setbkcolor(color)` 过程在 EGA 上的作用和 CGA 上不同。在 EGA 上, `setbkcolor` 将存于表项 `color` 处的实际颜色值拷贝到表项 0 处。

就颜色而言, VGA 驱动器和 EGA 驱动器一样, 它只是具有较高的分辨率。

6. 在图形模式下错误处理

这里简单浏览图形模式错误处理函数:

`grapherrmsg` 依据特定的错误码返回一个错误信息串
`graphresult` 返回上次错误操作的错误码

在调用图形库函数的时候若出现错误(如 `settextstyle` 需要一种字体但是没有找到), 则设置一个内部错误码。可以通过调用 `graphresult` 得到上次错误操作的错误码。下表说明错误返回码及其代表的意义:

错误码	图形错误常量	相应的错误的信息
0	<code>grOK</code>	No error
-1	<code>grNoInitGraph</code>	(BGI)graphics not installed(use <code>initgraph</code>)
-2	<code>grNotDetected</code>	Graphics hardware not detected
-3	<code>grFileNotFound</code>	Device driver file not found
-4	<code>grInvalidDriver</code>	invalid device driver file
-5	<code>grNoLoadMem</code>	Not enough memory to load driver

-6	grNoScanMem	Out of memory in scan fill
-7	grNoFloodMem	Out of memory in flood fill
-8	grFontNotFound	Font file not found
-9	grNoFontMem	Not enough memory to load font
-10	grInvalidMode	Invalid graphics mode for selected driver
-11	grError	Graphics error
-12	grIOerror	Graphics I/O error
-13	grInvalidFont	Invalid font file
-14	grInvalidFontNum	Invalid font number
-15	grInvalidDeviceNum	Invalid device number
-18	grInvalidVersion	Invalid version of file

调用 `grapherrormsg(graphresult)` 所返回错误信息也列在上表中。

错误返回码保存到当一个图形函数报告另一个错误时才发生变化。当成功执行 `initgraph` 或调用 `graphresult` 时, 错误返回码被置为 0。因此, 如果想知道哪个图形函数返回哪个错误, 应将 `graphresult` 返回的值保存起来, 然后再测试它。

7. 状态查询

这里简单概述图形模式状态查询函数:

<code>getarcoords</code>	返回上次调用 <code>arc</code> 或 <code>ellipse</code> 时的坐标
<code>getaspectratio</code>	返回图形屏幕的长宽比
<code>getbkcolor</code>	返回当前背景颜色
<code>getcolor</code>	返回当前绘图颜色
<code>getdrivename</code>	返回当前图形驱动器的名字
<code>getfillpattern</code>	返回用户定义的填充模型
<code>getfillsettings</code>	返回有关当前填充模型和颜色信息
<code>getgraphmode</code>	返回当前图形模式
<code>getlinesettings</code>	返回当前线型、线模型和线宽度
<code>getmaxcolor</code>	返回当前最大有效像素值
<code>getmaxmode</code>	返回当前驱动器的最大模式号
<code>getmaxx</code>	返回当前 x 方向的分辨率
<code>getmaxy</code>	返回当前 y 方向的分辨率
<code>getmodename</code>	返回给定的驱动器模式的名字字符串
<code>getmoderange</code>	返回给定的驱动器的模式范围值
<code>getpalette</code>	返回当前调色板和它的大小
<code>getpixel</code>	返回在 (x, y) 处的像素颜色
<code>gettextsettings</code>	返回当前文本字体、方向、大小和对齐模式的设置
<code>getviewsettings</code>	返回当前视区的有关信息
<code>getx</code>	返回当前位置的 x 坐标
<code>gety</code>	返回当前位置的 y 坐标

在 Turbo C 每类图形函数中,至少有一个状态查询函数。这些函数在各个分类中分别介绍。每个 Turbo C 的图形状态查询函数用“getsomething”命名(错误处理一类的函数例外)。它们当中的有些函数不带参数并返回表示所需信息的一个单一值,其它的使用一个指向由 graphics.h 中定义的结构体的指针作参数,用适当的信息返回到这个结构,而函数本身没有返回值。

图形系统控制类的状态查询函数是 getgraphmode、getmaxmode 和 getmoderange。第一个函数返回表示当前图形驱动器和模式的整数,第二个返回给定驱动器的最大模式号,第三个返回一个给定驱动器支持的模式范围。getmaxx 和 getmaxy 返回当前图形模式下最大的 x 坐标和 y 坐标。

绘图和填充状态查询函数是 getarccoords、getaspectratio、getfillpattern、getfillsettings 和 getlinesettings。getarccoords 将上次调用 arc 或 ellipse 后的坐标赋给一个结构;getaspectratio 返回当前图形系统为保证作出一个正确的圆而使用折长宽比率;getfillpattern 返回当前用户定义的填充模型;getfillsettings 将当前填充模型和填充颜色赋给一个结构;getlinesettings 将当前线型(实线、虚线等)、线宽(常规的或粗的)和线模型赋给一个结构。

在屏幕和视区这一类的状态查询函数是: getviewsettings、getx、gety 和 getpixel。如果已经定义了视区,可以 getviewsettings 获得它的绝对屏幕坐标,和知道是否允许剪切,这个函数将这些信息赋给一个结构。getx 和 gety 返回(相对于视区的)当前位置的 x 和 y 坐标。getpixel 返回一个指定象素的颜色。

图形模式文本输出函数中包含一个进行所有有关状态查询的函数: gettextsettings。这个函数将把当前字符字型、文本显示的方向(水平的或垂直的)、字符放大因子、串对齐要求(水平的和垂直的)赋给一个结构。

Turbo C 的颜色控制函数分类中包含三个状态查询函数。getbkcolor 返回当前背景颜色, getcolor 返回当前绘图颜色; getpalette 将当前绘图调色板的大小和内容赋给一个结构。getmaxcolor 返回当前图形驱动器和模式下最大有效象素值(调色板大小-1)。

最后, getmodename 和 getdrivename 分别返回给定驱动器模式的名字和当前图形驱动器的名字。

以上这些函数的具体使用说明请参阅第十八章。

14.2 Turbo C 图形程序设计

毫无疑问,在现今交互式环境下,成功的程序要求对屏幕完全控制。下面介绍 Turbo C 的一些屏幕控制函数。请记住, Turbo C 的屏幕操作子系统包括大量的函数,比这一节中涉及的要多得多,所以一定要参看上一节。

如前所述, Turbo C 屏幕控制软件包被分成两部分: 正文模式屏幕控制和图形模式屏幕控制。正文模式函数当显示适配器处于正文模式时管理屏幕的显示, 图形函数当计算机处于图形显示模式时作用于屏幕。虽然二者在概念上有联系, 但这两个子系统实际上是完全独立的。这一节将讨论一些常用的正文模式函数, 另外还讲述一些图形子系统的用法。大型的例子在 14.3 节后介绍。

注意: 这一节中描述的正文和图形屏幕函数最先出现在 Turbo C 1.5 版本中。它们并不

是 Turbo C 所独有的,因此它们将作为 C 语言的一部分来讨论。不过,它们完全适用于 Turbo C 和 Turbo C 环境。

14.2.1 基本正文模式函数

在研究最重要的正文模式屏幕操作和控制函数之前,需要记住两点:首先,所有的 Turbo C 正文模式函数要求在用到它们的程序中包含头文件 CONIO.H;其次,正文模式函数要求屏幕处于正文状态而不是图形状态。虽然正文模式是绝大多数 PC 操作系统所缺省设置的,这一点仍然不可忽视。

14.2.1.1 正文窗口

Turbo C 正文模式子系统内的绝大多数函数都是对窗口而不是对屏幕本身操作。幸运的是,窗口的缺省值就是指整个屏幕,所以不必担心要建立专门的窗口才能使用字符和图形程序。但是,了解窗口的基本概念会从 Turbo C 的屏幕功能上获益更多,这是十分重要的。

窗口是一个矩形的门户,用户程序可以通过它传递信息。窗口可以大到整个屏幕,也可以小到只有几个字符。在复杂的软件中,屏幕上常常可同时有几个窗口,每个都由程序用于执行独立的任务。

Turbo C 允许定义窗口的位置和大小。当定义了一个窗口后,正文操作程序就仅仅对所定义的窗口活动,而不再是对整个屏幕了。例如,函数 clrscr()仅清除当前窗口,而不是清除整个屏幕(当然,在缺省时,当前窗口就是整个屏幕)。另外,所有的位置坐标都是相对于当前窗口而言,而不再是相对于整个屏幕。

Turbo C 窗口最重要的特性之一是:对窗口的输出可以自动防止向窗口外溢出。如果一些输出超出了边界,则只有那些在窗口内的部分才被显示出来,其余的将被自动转到下一行。

对正文窗口而言,左上角坐标是(1,1)。正文模式屏幕函数使用的是相对于当前窗口的坐标,而不是相对于屏幕原点的坐标。因此,如果操作两个窗口,无论其中哪个是当前窗口,其左上角坐标都被定位为(1,1)。

这里,我们不必急于创建任何窗口,而仅仅使用缺省窗口,即整个屏幕。

14.2.1.2 清除窗口

clrscr()是一个最常见的屏幕操作函数,它可以清除窗口内的所有正文。该函数原型为:

```
void clrscr(void);
```

记住,虽然这个函数看起来象 clear screen 的缩写,但实际上是用于清除当前窗口的。

14.2.1.3 光标定位

第二个最常用的屏幕控制函数是 gotoxy(),用于光标定位。其原型是

```
void gotoxy(int x,int y);
```

这里,x 和 y 代表相对于当前窗口的光标 X 和 Y 坐标。如果两个坐标都越界,则不进行任何动作。不过记住,一个越界坐标不会被认为是一个错误。

为了看一看 clrscr()和 gotoxy()是如何工作的,可运行下面这段程序,它在屏幕上打印句子"Text screen Function are Fun!"。注意打印格式。

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

```

#include <string.h>
char mess[] = "Text Screen Functions are Fun!";
main(void)
{
    register int x,y;
    char *p;
    p = mess;
    for(x=1;x<=strlen(mess);x++) {
        for(y=1,y<=12;y++) {
            gotoxy(x,y);
            printf("%c", *p);
            gotoxy(x,25-y);
            printf("%c", *p);
        }
        p++;
    }
    getch();
    gotoxy(1,25);
    return 0;
}

```

14.2.1.4 清除到行末

如果用户程序要求输入,函数 `clrcol()` 会十分有用。它从左到右清除一行中从光标所在处到该行末尾的所有内容。它的原型为:

```
void clrcol(void);
```

可以将此函数和 `gotoxy()` 一起用来构造一个函数,此函数可在清除行以后在特定坐标处显示提示信息。下面这个程序使用与 `prompt()` 类似的函数,可使屏幕左边的信息不受影响。

```

#include <conio.h>
#include <stdio.h>
void prompt(char *s,int x,int y);
main(void)
{
    clrscr();
    prompt("this is a prompt",1,10);
    getch();
    prompt("this is too", 1,10);
    getch();
    return 0;
}
void prompt (char *s,int x,int y)
{
    gotoxy(x,y);

```

```

        clrcol();
        printf(s);
    }

```

14.2.1.5 删除和插入行

用户有时会发现满屏幕都是字符,而且准备删除一行或多行,同时把该行下面的所有行上移。相应的,有时又想插入一个空行,同时把该行下面的所有行下移。为了解决这些问题,可用函数 `delline()` 和 `insline()`。它们的原型如下:

```

void delline(void);
void insline(void);

```

调用 `delline()` 函数可删除光标所在行,同时把该行下面的所有行都上移一行。调用 `insline()` 函数将插入一个新的空白行到光标所在行,同时把下面的所有行都向下顺移一行。

下面的程序说明 `delline()` 的用法,先将屏幕各行填满,第一行由“A”组成,第二行由“B”组成,以此类推。然后删除其它所有行。

```

#include <conio.h>
#include <stdio.h>
main(void)
{
    int x,y;
    clrscr();
    /* fill the screen with some lines */
    for(y=1;y<25;y++){
        for(x=1,x<80;x++){
            gotoxy(x,y);
            printf("%c", (y-1)+'A');
        }
        /* delete every other line */
        for(y=2;y<26;y+=2){
            gotoxy(1,y);
            delline();
        }
    }
    return 0;
}

```

这个程序只要用 `insline` 函数稍微改动一下,即可用来在删除行的相同位置插入空白行。要看看结果如何,将删除循环改为下面所示的语句段。

```

for(y=2;y<26;y+=2){
    gotoxy(1,y);
    delline();
    insline();
}

```

14.2.1.6 建立窗口

迄今为止,所有函数使用的都是缺省窗口。然而,用户可以在任何位置创建任意大小的窗口,只要屏幕容纳得下就行。这需要用到 `window()` 函数。它的原型如下所示:

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

如果其中有一个坐标是无效的,window()将不起作用。当成功地调用一次 window() 后,所有对坐标的引用都被转换为相对于这个窗口,而不再是相对于整个屏幕。例如,下面这段程序建立了一个窗口,并且从窗口内的坐标位置(2,3)处开始写一行字符。

```
printf("at screen location 2,3");
window(10,10,60,15);
gotoxy(2,3);
printf("at window location 2,3");
```

这段程序的工作过程见图 14. 2.

注意:用来调用 window()的坐标是屏幕绝对坐标,而不是相对于当前窗口的。这就是说多窗口不需要嵌套,而且一个窗口可以和另外一个窗口重叠。

下面这段程序首先沿屏幕画边框,然后再建立两个独立的带边框的窗口。每个窗口里字符的位置由 gotoxy()语句确定,该语句所用的坐标是相对于每个窗口而言的。最后,创建第三个窗口并覆盖了其它两个。这个程序的输出如图 14. 3 所示。

```
#include <conio.h>
#include <stdio.h>
void border(int,int,int,int);
main(void)
{
    clrscr();
    /* draw a border around the screen for perspective */
    border(1,1,79,25);
    /* create first window */
    window(3,2,40,9);
```

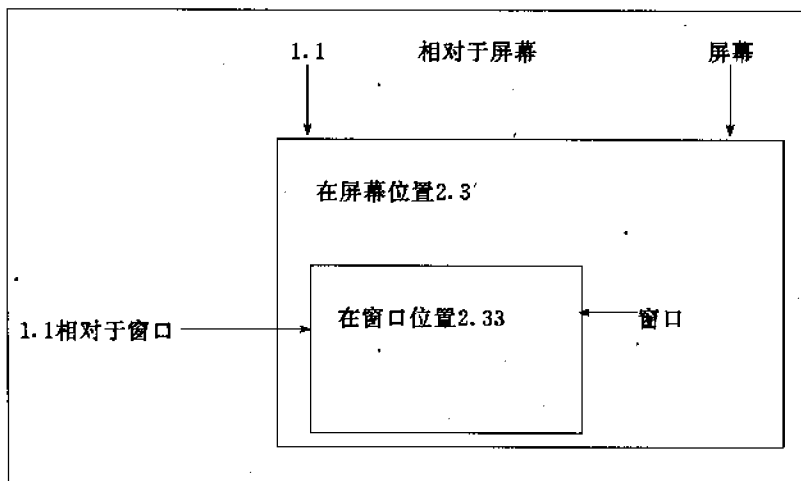


图 14. 2 窗口内相对坐标的说明

```
border(3,2,40,9);
gotoxy(3,2);
printf("first window");
```

```

/* create a second window */
window(30,10,60,18);
border(30,10,60,18);
gotoxy(3,2);
printf("second window");
gotoxy(5,4);
printf("still in second window");
/* go back to first window */
window(3,2,40,9);
gotoxy(3,3);
printf("back in first window");
getch();
/* demonstrate overlapping windows */
window(5,5,50,15);
border(5,5,50,15);
gotoxy(2,2);
printf("this window overlaps the first two");
getch();
return 0;
}

/* Draws a border around a text window. */
void border(int startx,int starty,int endx,int endy)
{
    register int i;
    gotoxy(1,1);
    for(i=0;i<=endx-startx;i++)
        putchar(' ');
    gotoxy(1,endy-starty);
    for(i=0;i<=endx-startx;i++)
        putchar(' ');
    for(i=2;i<=endy-starty;i++) {
        gotoxy(1,i);
        putchar(' ');
        gotoxy(endx-startx+1,i);
        putchar(' ');
    }
}

```

14.2.1.7 一些窗口 I/O 函数

C 语言的常规输出函数如 `printf()`，不适用于窗口屏幕环境，而 TurboC 库中包含有可用于窗口的函数。当用缺省窗口为整个屏幕时，是用基于窗口的 I/O 函数还是用标准函数都无关紧要。不过当使用比屏幕小的窗口时，由于它们自动地防止字符写出该窗口，所以需要用到窗口定向函数。当使用窗口 I/O 函数时，正文将自动在窗口边界下卷。新的或修整过的

正文 I/O 函数可见表 14.1.

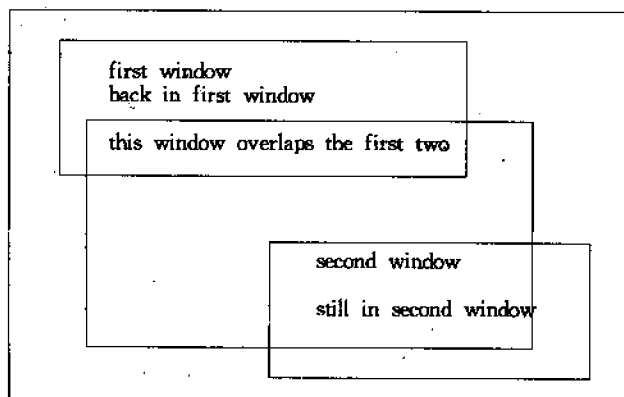


图 14.3 正文窗口示例程序输出

`cprintf()` 函数除了适用于窗口系统以外,操作方式完全与 `printf()` 一样。`cputs()` 函数在功能上也相当于 `puts()`,它与 `puts()` 的区别实际上仅仅在于承认窗口。与标准 I/O 函数不同,这些函数自动地防止超过当前窗口的输出部分溢出到屏幕的其它地方。`putch()` 也同样如此。它不允许字符被写到当前窗口以外。函数 `getche()` 也不会接收来自当前窗口以外的输入。最后,`cgets()` 也被修改为承认窗口并且不允许用户输入超过窗口边界的正文。

表 14.1 窗口用正文 I/O 函数

函数	功 能
<code>cprintf()</code>	将格式化的输出送到当前窗口
<code>cputs()</code>	将一个字符串送到当前窗口
<code>putch()</code>	将单个字符送到当前窗口
<code>getche()</code>	读入一字符并回显到当前窗口
<code>cgets()</code>	读入一字符串并回显到当前窗口

为了弄清窗口 I/O 函数和标准函数的区别,试运行下面的程序并观察它的结果。如程序所设想的那样,`cprintf()` 的输出行在窗口的边界下卷,而用 `printf()` 显示则不会这样。

```

/* Demonstration cprints(). */
#include <conio.h>
#include <stdio.h>
void border(int,int,int,int);
main(void)
{
    clrscr();
    /* create a window */
    window(3,2,20,9);
    border(3,2,20,9);
    gotoxy(2,2);

```

```

    printf("This will wrap around at the edge of the window");
    printf("This will not wrap at the edge of the window");
    getch();
    return 0;
}

/* Draws a border around a text window. */
void border(int startx,int starty,int endx,int endy)
{
    register int i;
    gotoxy(1,1);
    for(i=0;i<=endx-startx;i++)
        putchar(' ');
    gotoxy(1,endy-starty);
    for(i=0;i<=endx-startx;i++)
        putchar(' ');
    for(i=2;i<endy-starty;i++) {
        gotoxy(1,i);
        putchar(' ');
        gotoxy(endx-startx+1,i);
        putchar(' ');
    }
}

```

另一个需要弄清楚的问题是,这些基本 I/O 函数是不会重定向的。也就是说,C 语言的标准 I/O 函数允许输出重定向,即从一个磁盘文件或辅助设备输入改为向其输出,或输出改为输入,而基本窗口的字符屏幕函数却不能改变。

14.2.1.8 正文模式

到现在为止,计算机的缺省显示器,绝大多数均为 25 行 80 列。不过,实际上仍然有五种不同的正文模式可供选择。如果配置有图形适配器,就可以在 40 列与 80 列模式和彩色和黑白模式间自由选择。可以用函数 `textmode()` 改变正文模式。它的原型是:

```
void textmode(int mode);
```

参数 `mode` 必须是表 14.2 中所示值中的一种。既可以使用整型量也可使用宏替换名(这些宏在 `CONIO.H` 中有定义)。

将函数 `textmode()` 包含在程序中,主要是为了确保当前显示模式正是程序所要求的。

如果程序设置了显示器模式,则在退出时必须将其保存起来。为了做到这一点,用 `LASTMODE` 作为参数调用 `textmode()`,即能自动保存在程序中修改了的显示器模式。

表 14.2 正文显示器模式

宏名	数值	描述
BW40	0	40 列黑白
C40	1	40 列彩色

续表 14.2

宏名	数值	描述
BW80	2	80 列黑白
C80	3	80 列彩色
MONO	7	80 列单色
LASTMODE	-1	启用先前模式

14.2.1.9 彩色正文输出

如果有彩色监视器和彩色图形适配器,就可以用不同颜色来显示正文,并可以修改正文颜色和背景颜色。但一定要记住:只有特殊的窗口 I/O 函数才能用于显示特定颜色的正文,像 printf() 这样的函数则不能起作用。

函数 textcolor() 可将正文颜色改变成指定颜色,还可使字符闪烁。textcolor() 的原型是:

```
void textcolor(int color);
```

参数 color 对应于不同的颜色,其值可以从 0 到 15。不过,在头文件 CONIO.H 中这些颜色均有宏替换名而且相当好记。这些宏替换名和相应数值见表 14.3。

表 14.3 正文颜色宏名和对应数值表

宏名	对应数值
BLACK	0
BLUE	1
GREEN	2
CYAN	3
RED	4
MAGENTA	5
BROWN	6
LIGHTGRAY	7
DARKGRAY	8
LIGHTBLUE	9
LIGHTGREEN	10
LIGHTCYAN	11
LIGHTRED	12
LIGHTMAGENTA	13
YELLOW	14
WHITE	15
BLINK	128

正文颜色的变化仅仅影响紧随其后的写操作,记住这一点十分重要,它不会改变已在屏幕上显示了正文。

要使正文闪烁,必须将期望的颜色和数值 128(BLINK)进行 OR 操作。例如,下面这句代码使得其后输出的正文为绿色闪烁。

```
textcolor(GREEN|BLINK);
```

函数 `textbackground()` 用于设置正文屏幕的背景颜色。与 `textcolor()` 类似,调用 `textbackground()` 只影响随后写入的字符的背景颜色。其原型为:

```
void textbackground(int color);
```

`color` 的值必须是 0 至 6 的范围内。这意味着背景颜色只能用上表中的前七种颜色。

下面这个程序表明了正文颜色函数的功能,它能显示所有前景和背景颜色的组合:

```
#include <conio.h>
main(void)
{
    register int fg,bg;
    textmode(C80);
    for(fg=BLUE;fg<=WHITE;fg++){
        for(bg=BLACK;bg<=LIGHTGRAY;bg++){
            textcolor(fg);
            textbackground(bg);
            cprintf("this is a test");
        }
        cprintf("\n\n");
    }
    textcolor(WHITE);
    textbackground(BLACK);
    cprintf("done");
    textmode(LASTMODE);
    return 0;
}
```

14.2.2 Turbo C 的图形子系统简介

Turbo C 的第二个屏幕控制子系统是它的图形软件包。这个软件包包含了大量函数,用于完成从画线、画圆到构成棒状、饼状结构图的各种任务。这一节将介绍最常用的那些函数。

图形函数的原型在文件 `GRAPHICS.H` 中。这个函数必须被那些用到图形函数的程序所包含。所有的图形函数都是 `far` 类型的函数,用户程序在使用它们之前必须知道这一点。

注意:为了能够运行下面所述的这些程序,用户的计算机必须配置图形适配器,当然,有彩色监视器更好。

14.2.2.1 一个有别名的窗口

如正文屏幕控制函数一样,所有图形函数都针对窗口操作。在 TurboC 术语中,图形窗口叫做视口(viewport),视口实际上与正文窗口相似。窗口和视口之间的唯一区别是视口的左上角坐标是(0,0),而不是窗口那样为(1,1)。

缺省时,整个屏幕就是视口。不过,用户可以创建具有其它大小的视口。本节稍后将介绍如何创建视口。注意,当学习以下的内容时,所有的图形输出都是相对于当前视口的,而视

口未必是整个屏幕。

14.2.2.2 初始化显示适配器

在使用图形函数之前,必须把显示适配器设置为某一图形模式。缺省时,绝大多数计算机系统都使用 DOS 的 80 列正文模式。由于这不是图形模式,所以图形函数不能工作。若要将显示适配器设置为图形模式,必须使用 `initgraph()` 函数。其原型如下所示:

```
void far initgraph(int far * drive,int far * mode,char far * path);
```

`initgraph()` 函数将图形驱动程序装入到内存。该驱动程序的代号为 `driver` 所指的数字。如果没有装入图形驱动程序,图形函数将不能工作。参数 `mode` 是用来确定显示器模式的整型指针。最后,可以用由 `path` 所指的字符串来确定进入该驱动软件的路径。如果没有指出路径,就在当前目录下搜寻。

图形驱动程序包含在 .BGI 文件中,这个文件必须是系统可以得到的。这些驱动程序是由 Turbo C 提供的。不过不必为文件的实际名字担忧,因为只需要用其代号来指定该驱动程序。在 GRAPHICS.H 中为用于此目的的数字定义了符号值,具体如下所示:

宏名	对应数值
DETECT	0
CGA	1
MCGA	2
EGA	3
EGA64	4
EGAMONO	5
IBM8514	6
HERCMONO	7
ATT400	8
VGA	9
PC3270	10

当选用了 DETECT 时, `initgraph()` 会自动在当前系统下搜寻显示器的硬件类型,并且选用最大可能的分辨率。同时 `driver` 和 `mode` 被设成特定的值。

`mode` 的值必须是表 14.4 中所示的图形模式值之一。例如,为了将图形系统初始化成 CGA 四种颜色、320X200 的图形模式,应该用下面这段程序。假设图形驱动程序的 .BGI 文件在当前目录下。

```
#include <graphics.h>

.
.

int driver, mode;
driver = CGA;
mode = CGACO;
initgraph(&driver,&mode,"");
```

14.2.2.3 退出图形模式

若要终止图形模式并返回到正文模式,可使用 `closegraph()` 或 `restorecrtmode()` 函数。它

们的原型如下:

```
void far closegraph();
void far restorecrtmode();
```

函数 `closegraph()` 用于当用户程序还要继续在字符模式下运行时所使用。它释放由图形函数所占用的内存,同时将显示器模式恢复为调用 `initgraph()` 之前的模式。如果用户程序要终止运行,则可用 `restorecrtmode()` 将显示适配器设置为原来的模式,即首次调用 `initgraph()` 之前的模式。

表 14.4 Turbo C 图形驱动程序及模式的宏定义

驱动程序	模式	等价值	分辨率
CGA	CGAC0	0	320 × 200
	CGAC1	1	320 × 200
	CGAC2	2	320 × 200
	CGAC3	3	320 × 200
	CGAHI	4	640 × 200
MCGA	MCGAC0	0	320 × 200
	MCGAC1	1	320 × 200
	MCGAC2	2	320 × 200
	MCGAC3	3	320 × 200
	MCGAMED	4	640 × 200
EGA	MCGAHI	5	640 × 480
	EGALO	0	640 × 200
EGA64	EGAHI	1	640 × 350
	EGA64LO	0	640 × 200
EGAMON()	EGA64HI	1	640 × 350
	EGAMONHI	3	640 × 350
HERC	HERCMONHI	0	720 × 348
ATT400	ATT400C0	0	320 × 200
	ATT400C1	1	320 × 200
	ATT400C2	2	320 × 200
	ATT400C3	3	320 × 200
	ATT400CMED	4	640 × 200
VGA	ATT400CHI	5	640 × 400
	VGALO	0	640 × 200
	VGAMED	1	640 × 350
PC3270	VGAHI	2	640 × 480
	PC3270H	0	720 × 350
IBM85	IBM8514LO	0	640 × 480
	IBM8514HI	1	1024 × 768

14.2.2.4 颜色和调色板

显示适配器的类型是与系统所要求颜色的类别和数目相关的,这限于图形模式中。

CGA 和 EGA/VGA 之间的区别是适配器之间最大的区别。

CGA 四色图形模式提供了四块调色板,每块板上均有四种颜色可供选择。颜色数值从 0 到 3,0 总是为背景色。调色板的数值也是从 0 到 3,若要选择某调色板,就要设置参数 mode 为 CGACx,这里 x 为调色板号。调色板与其相关色请见表 14.5。

在 EGA/VGA 的 16 色图形模式中,一个调色板可以有 16 种颜色,它们是从 64 种颜色中选择出来的。

表 14.5 CGA 的调色板与颜色值

调色板	0	颜色值 1	2	3
	0	背景	GREEN YELLOW	RED
1	背景	CYAN	MAGENTA	WHITE
2	背景	LIGHTGREEN	LIGHTRED	YELLOW
3	背景	LIGHTCYAN	LIGHTMAGENTA	WHITE

用函数 setpalette()可以改变调色板。该函数原型是:

```
void far setpalette(int index,int color);
```

初次使用此函数会有某些困难。实际上,该函数用一个表把 color 值和 index 联系起来,该表将所要求的颜色变换成屏幕上的实际颜色。颜色编号如表 14.6 所示。

对于 CGA 模式,只能改变背景色。而设置背景色时 index 总取 0 值。所以,对于 CGA 模式,下面的语句可将背景色改为绿色。

```
setpalette(0,GREEN);
```

EGA 能在总共 64 种颜色中同时显示 16 种。可以用 setpalette()函数将某一种颜色设置为这 16 种不同颜色中的一种。例如,下面的语句设置颜色 5 的值代表青色。

```
setpalette(5,EGA_CYAN);
```

14.2.2.5 基本图形函数

最基本的图形函数不外乎画点、线和圆。这些函数分别叫做 putpixel()、line()和 circle(),它们的原型为:

```
void far putpixel(int x,int y,int color);
```

```
void far line(int startx,int starty,int endx,int endy);
```

```
void far circle(int x,int y,int radius);
```

putpixel()函数将把 color 所指定的颜色写到由 x 和 y 坐标确定的点上。line()函数将从指定点(startx,starty)到(endx,endy)之间画一条直线,其颜色为当前色。circle()函数画一个以 radius 为半径的圆,颜色为当前绘图色,圆心位置为(x,y)。如果坐标超出了范围,则超出部分不画。

下面的程序可用于说明这些函数:

```
#include <graphics.h>
#include <conio.h>
main(void)
```

```

int driver, mode;
register int i;
driver = DETECT;

```

表 14.6 函数 setpalette() 的颜色编号

CGA 只用于背景	
宏	值
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
EGA 和 VGA	
宏	值
EGA_BLACK	0
EGA_BLUE	1
EGA_GREEN	2
EGA_CYAN	3
EGA_RED	4
EGA_MAGENTA	5
EGA_BROWN	20
EGA_LIGHTGRAY	7
EGA_DARKGRAY	56
EGA_LIGHTBLUE	57
EGA_LIGHTGREEN	58
EGA_LIGHTCYAN	59
EGA_LIGHTRED	60
EGA_LIGHTMAGENTA	61
EGA_YELLOW	62
EGA_WHITE	63

```

initgraph(&driver,&mode,"");

line(0,0,200,150);
line(50,100,200,125);
/* some points */
for(i=0;i<319;i+=10) putpixel(i,100,RED);

/* draw some circles */
circle(50,50,35);
circle(100,160,100);
getch(); /* wait until keypress */

restorecrtmode();

return 0;
}

```

14.2.2.6 改变绘图色

刚才的例子用白色画线和圆,白色是缺省绘图色(记住,函数 putpixel 所画颜色由第三个参数指定)。可以用函数 setcolor() 设置绘图的当前色。其原型如下:

```
void far setcolor(int color);
```

color 的值一定要处在当前图形模式下的有效范围内。一旦改变了当前色,则其后的写操作均沿用这个新颜色。

14.2.2.7 区域填充

可以用函数 floodfill() 填充任何封闭的图形。其原型如下:

```
void far floodfill(int x,int y,int bordercolor);
```

若用该函数去填充一个封闭的图形,则应以图形中任一点为坐标,以图形建立时边线的颜色为其颜色来调用该函数。一定要确保所填充的图形是完全封闭的。如果不这样,图形以外的一些区域也将填充。可以以当前的填充模式和颜色来填充图形,不过,也可以用函数 setfillstyle() 来改变填充对象的填充模式。该函数的原型如下:

```
void far setfillstyle(int pattern,int color);
```

pattern 的值及其宏名见表 14.7(在 GRAPHICS.H 中定义)。

表 14.7 填充模式

宏名	值	填充模式
EMPTY_FILL	0	以背景色着色
SOLID_FILL	1	全部着色
LINE_FILL	2	填水平线
LTSLASH_FILL	3	填左斜线
SLASH_FILL	4	填粗左斜线
BKSLASH_FILL	5	填粗右斜线
LTBKSLASH_FILL	6	填右斜线

续表 14.7

宏名	值	填充模式
HATCH_FILL	7	填浅阴影线
XHATCH_FILL	8	填重交叉阴影线
INTERLEAVE_FILL	9	填交替直线
WIDEDOT_FILL	10	填稀点
CLOSEDOT_FILL	11	填密点
USER_FILL	12	用户定义

14.2.2.8 rectangle()函数

函数 rectangle() 以当前绘图色画一个以 (left, top) 为左上角且以 (right, bottom) 为右下角的矩形。其原型为：

```
void far rectangle(int left, int top, int right, int bottom);
```

下面这个程序演示了迄今为止引入的所有图形函数,除了 setpalette() 之外。这个程序要求配置了 VGA 适配器。不过此程序可以很方便地修改,这适用于用户的图形适配器类型。

```
#include <graphics.h>
#include <conio.h>
main(void)
{
    int driver, mode;
    register int i;
    driver = VGA;
    mode = VGAMED;

    initgraph(&driver, &mode, "");
    /* outline the screen */
    rectangle(0, 0, 639, 349);
    setcolor(RED);
    line(0, 0, 639, 349);
    setcolor(GREEN);
    rectangle(100, 100, 300, 200);
    setcolor(BLUE);

    floodfill(110, 110, GREEN);
    setcolor(CYAN);
    line(50, 200, 400, 125);
    setcolor(RED);
    for(i=0; i<640; i+=3) line(320, 174, i, 0);
    setcolor(GREEN);
    circle(50, 50, 35);
    circle(320, 175, 100);
```



```

circle(500,250,90);
circle(100,100,200);
/* make a bullseye */
setfillstyle(SOLID_FILL, GREEN);
floodfill(500,250, GREEN);
setcolor(RED);
circle(500,250,60);
setfillstyle(SOLID_FILL, RED);
floodfill(500,250, RED);

setcolor(GREEN);
circle(500,250,30);
setfillstyle(SOLID_FILL, GREEN);
floodfill(500,250, GREEN);
setcolor(RED);
circle(500,250,10);
setfillstyle(SOLID_FILL, RED);
floodfill(500,250, RED);

getch();
restorecrtmode();
return 0;
}

```

14.2.2.9 创建视口

可以以与正文模式中创建窗口相似的格式来创建图形模式下的视口。如前所述,所有的图形输出都是与当前视口的坐标相关的。这意味着视口的左上角坐标为(0,0),不管视口处在屏幕上的什么位置。用于创建视口的函数叫做 `setviewport()`,它的原型如下:

```
void far setviewport(int left, int top, int right, int clipflag);
```

使用此函数时,需要确定视口在屏幕上的左上角和右下角坐标。如果参数 `clipflag` 为非零值,则自动截去超越视口边界的输出部分,否则,剪裁标识被关上,输出就有可能超出视口。记住一点,输出被截去不算出错。

可以用函数 `getviewsettings()` 来获知当前视口的尺寸参数。它的原型是:

```
void far getviewsettings(struct viewporttype far *info)
```

结构 `viewporttype` 在 `GRAPHICS.H` 中定义如下:

```

struct viewporttype{
    int left, top, right, bottom;
    int clipflag;
}

```

其中 `left`、`top`、`right` 和 `bottom` 域装有视口的左上角和右下角坐标。当 `clipflag` 为零时,输出越过视口边界则不会被截掉,否则,剪裁标识将防止边界溢出。

函数 `getviewsettings()` 最重要的用途,是允许用户程序能自动调整以适应于所使用的显

示适配器类型,这是通过检查视口的尺寸及作相应的调整来实现的。例如,下面的程序用到函数 `setviewport()` 和 `getviewsettings()` 构成一个以屏幕为中心的第二个视口。在第一个视口中,画着垂直线。在第二个视口中,画的是水平线。这个程序在任何图形适配器下均可运行。

```
#include <graphics.h>
#include <stdlib.h>
#include <conio.h>
main(void)
{
    int driver,mode;
    struct viewporttype info;
    int width,height;
    register int i;
    driver = DETECT;
    initgraph(&driver,&mode,"");
    getviewsettings((struct viewporttype far *) &info);
    for(i=0;i<info.right;i+=20)
        line(i,info.top,i,info.bottom);
    height = info.bottom/4;
    width = info.right/4;
    setviewport(width,height,width*3,height*3,1);
    getviewsettings((struct viewporttype far *) &info);
    for(i=0;i<info.right;i+=10)
        line(0,i,info.bottom,i);
    getch(); /* wait for keypress */
    restorecrtmode();
    return 0;
}
```

请记住,前面仅仅对最常见的正文屏幕和图形函数作了一个介绍。读者可以在后面的章节中查到所有屏幕和图形函数的详细信息。

14.3 IBM/PC 的文本方式

视频监控器在两种基本视频方式下工作:图形方式和文本方式。在图形方式下,由于用户控制了屏幕上每个象素点,他可以显示任何想要显示类型的图象。在文本方式下,就只能显示适配器的字符集里的字符。文本模式的优点是字符输出能被快速方便地格式化并显示。这一节在从程序里送出的字符数据。怎样显示在屏幕上方面提供一个俯瞰。

14.3.1 PC 显示器适配器和屏幕

计算机的视频显示系统包含两部分:显示器适配器和屏幕。在计算机内部,显示器适配器(video adapter)是一组电路,将程序输出的数据转换成监视器可以使用的视频信号。

如果一个程序输出字母 T,代表字母 T 的 ASCII 码会被送给显示器适配器。显示器适

适配器使用这个 ASCII 码去查阅在适配器的 ROM(只读存储器)里装着的字母外形。从 ROM 芯片读出的信息告诉适配器,哪些像素点该亮,这样将该字符显示在屏幕上。最后,字符就在当前光标处和其它所指定的位置显示出来。图 14.5 显示了在屏幕上显示一个字符的基本过程。

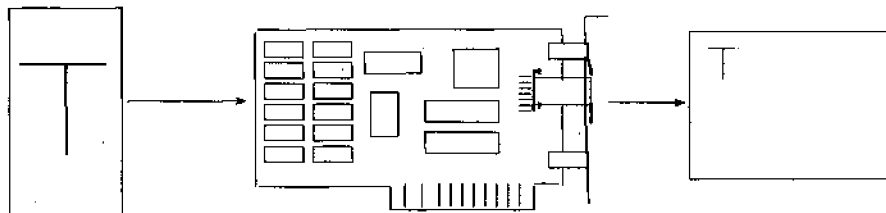


图 14.5 如何显示一个字符

- 注 {
- 第一段:一个“T”字符被从程序传送给显示器适配器。
 - 第二段:显示器适配器搜索其 ROM,从确定哪些像素点需要变亮,从而显示出“T”字符。
 - 第三段:显示器适配器给监视器送去一个信号,导致“T”被显示在监视器上。

图 14.5 中的过程图示了在文本方式下,数据是如何显示在屏幕上的。文本方式是只能显示字符数据的显示方式。

Turbo C 支持六种不同的文本方式。这些方式的区别在于屏幕上能显示的字符数目的多少和是否使用彩色。表 14.8 列出了六种文本方式和它们各有不同的特点。C4350 只有在 EGA 和 VGA 显示器上才能用。

表 14.8 Turbo C 的文本方式

方式	特点
BW40	一种黑白 40 列方式。这种方式下可以显示 25 行文本,其中每行可以容纳 40 个字符。字符仅以黑白两色显示。
C40	一种彩色 40 列方式。屏幕可显示 40 列 25 行的彩色字符。
SBW80	一种黑白 80 列方式。屏幕能显示 80 列 25 行。字符以黑白两色显示。
C80	一种显示 80 列 25 行的彩色方式。
MONO	显示 80 列 25 行单色文本的方式。
C4350	为 EGA 和 VGA 适配器设的一种特殊彩色方式。如果使用了 EGA 适配器,则显示 80 列 43 行。如果使用了 VGA 适配器,则显示 80 列 50 行的文本。

14.3.2 视频缓冲区 I/O

为了让监视器上的图形对用户看来是静止的,图形在每秒钟要重画 60 次。尽管缓冲区画面好象并未变化,但屏幕确实每秒钟要重画 60 次。

幸运的是,一般程序不用在每次屏幕重画的时候都把数据送到适配器。当程序把数据送到显示器适配器之后,数据就被存放在 RAM 的一个特殊区域,这个区域是特地为视频存储器保留的。显示器适配器每次重画屏幕时,适配器都检查视频存储器,看看屏幕上应该显示

什么。

屏幕上的每个显示位置都有一个特定的视频内存位置相对应。如果一个视频内存位置里有一个字符,则该字符就被画在视频显示的相应位置。

让视频显示器上的每个位置都有一个相应的内存位置,这主要是存储变换视频 I/O 之后的。如果没有指定一个内存位置,程序输出的显示数据被存储在视频内存下一个可用的位置里。如果有意,用户可以指定将数据存储在内存中的哪个位置。指定一个内存地址使得用户可以将显示数据显示在一个所希望的特殊屏幕位置上。

使用 Turbo C 的内部视频函数,可以制作复杂的图像,无需考虑对视频内存的直接访问。有了这些视频函数,可以通过使用正确的行地址和列地址,将数据放在屏幕上的任何地方。不用使用实际的视频内存地址。Turbo C 为你做好这一切。

在工作于文本方式时,屏幕上的每个位置都有两个内存字节与它相关联。第一个内存字节存放将要显示字符的 ASCII 码值。第二个字节叫做属性字节,存放怎样显示这个字符的信息。图 14.6 显示了视频内存里的一个位置是怎样与一个屏幕上的位置相关联的。

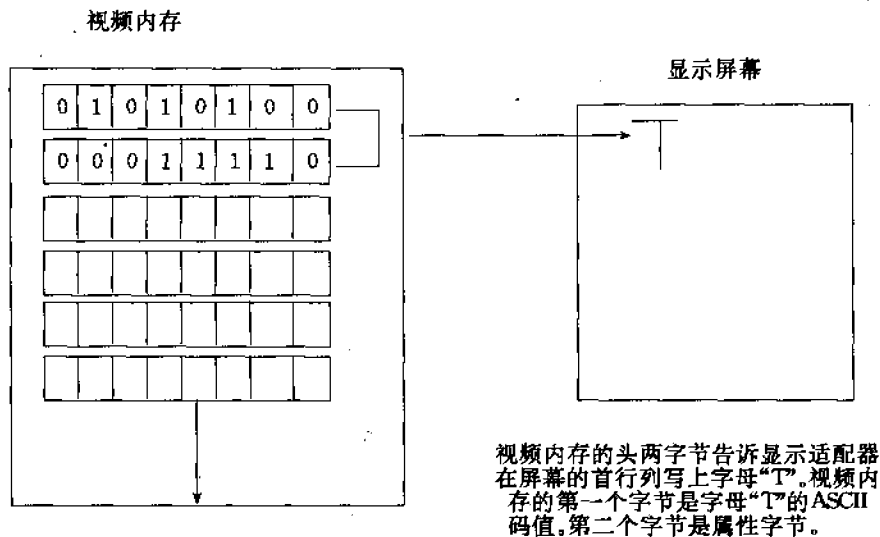


图 14.6 内存与视频之间的映射

14.4 控制文本屏幕

Turbo C 的函数库使得用户可以很容易地控制文本方式下屏幕的画面。这些函数可用于选择使用文本方式、背景色和文本的属性。也可以设置一个全局变量,使用直接视频内存 I/O,以获得更好的性能。这一节告诉读者如何开始使用这些文本方式控制函数。

14.4.1 使用文本方式控制函数

第一个需要调用的文本控制函数是这个 `textmode()` 函数。这个函数决定能够显示多少字符和使用哪种颜色。注意使用 `textmode()` 函数的格式:

```
textmode (int t_mode);
```

t_mode 是一个整数值, 决定将要使用哪种文本方式。可以使用一个整数值或者一个符号常数来代替 t_mode 参数。符号常数和它们的整数值在表 14.2 里列出。表 14.2 里显示的六种文本方式已在前面的表 14.8 里描述过。

你可以随便以下列语句中的一种调用 textmode() 函数:

```
textmode(C4350);
```

```
textmode(LASTMODE);
```

对 textmode() 的第一个调用的结果是: 如果使用的是 EGA 适配器/监视器, 则设置显示 43 行的屏幕方式; 如果使用的是 VGA 适配器/监视器, 则设置 50 行显示。第二个对 textmode() 的调用将文本方式置回上一个使用过的文本方式。

如果用户有一个彩色监视器, 并且使用了彩色文本方式, 就可以改变背景颜色。使用 textbackground() 函数, 用如下的格式, 就可以设置背景色:

```
textground (int b_color);
```

b_color 是一个整型变量, 它决定使用哪一种背景颜色。对于 textbackground() 函数, 也可以用一个符号常量作参数。

表 14.3 显示了那些可以用在象 textbackground() 的函数中, 指定想要的颜色的符号常量和整数值。表 14.3 里只有八种颜色可以用作 textbackground() 函数的 b_color 参数。

例如, 如果作了如下的函数调用, 视屏屏幕的背景色会置为绿色:

```
textbackground (GREEN);
```

可以使用 text_color 函数置显示字符的颜色。注意调用 textcolor() 函数的格式:

```
text_color (int t_color);
```

t_color 可以是表 14.3 所列出的任意符号常数。一个调用 textcolor() 的实例如下:

```
textcolor(YELLOW);
```

这个调用将文本或前景置为黄色。

在使用了 textbackground() 和 textcolor() 之后, 只有直接视频函数受到影响。直接视频函数是那些直接把信息写到屏幕上去的函数, 诸如 cprintf() 和 cputs()。象 printf() 和 puts() 这类函数不受 textbackground() 和 textcolor() 的影响。

lowvideo()、normvideo() 和 highvideo() 函数改变显示的文本字符的密度。lowvideo() 关闭高密度位, 并导致其后的所有直接视频函数显示低密度的字符。highvideo() 打开高密度位, 并导致文本以高密度显示。normvideo() 函数用程序开始之前屏幕显示字符的方式显示字符。注意下面的代码片段, 显示了这三个函数是如何使用的:

```
textcolor( RED );
```

```
lowvideo();
```

```
cprintf( "This is a test string. \n" );
```

```
normvideo();
```

```
cprintf( "This is a test string. \n" );
```

```
highvideo();
```

```
cprintf( "This is a test string. \n" );
```

这个代码片段的头两个语句将字符颜色置成低密度红色。然后, 当第一个 cprintf() 语句执行后, 消息以低密度红色显示。normvideo() 语句将视频方式设置为程序启动前所用的那

一种。因此,第二个和第三个 `cprintf()` 语句以程序启动前所用的颜色显示消息,而不是红色。第二个 `cprintf()` 语句显示一条普通密度的消息,第三个 `cprintf()` 语句则显示一条高密度的消息。

用 `textattr()` 函数可以在一条语句用时设置前景色和背景色。`textattr()` 通过改变文本属性字母的值来达到目的。属性字节(attribute byte)是控制每个文本字符的外观的字节。这个字节与 `textbackground()` 和 `textcolor()` 函数所控制的是同一个。属性字节的格式如图 14.7 所示。

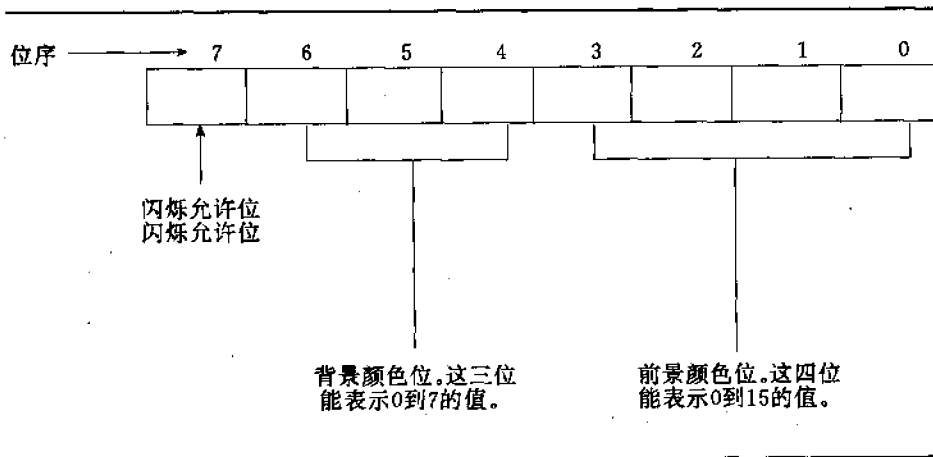


图 14.7 属性字节

程序 `artdemo` 显示了 `textattr()` 函数怎样使用。

`artdemo.c` `textattr()` 函数的演示

```

1  /* ARTDEMO.C Text-mode attribute demonstration program.
2
3      This program uses textattr() to change the
4      foreground and background text colors. */
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <conio.h>
8
9  int main( void )
10 {
11     int i, j, k;
12     for( i = 0; i <= 7; i++ ) {
13         for( j = 0; j <= 15; j++ ) {
14             k = i;
15             k = ( k << 4 );
16             textattr( k + j );
17             for( k = 48; k <= 127; k++ )
18                 putchar( k );

```

```
19     cprintf( "\r\n" );
20     }
21     }
22
23     return 0;
24 }
```

程序 `artdemore` 里的程序每逢显示一个新行的时候,就使用 `textattr()` 函数改变属性字节的值。程序包含三个循环。其中两个循环用于在背景色和前景色上循环,第三个循环打印一行字符。

从 12 行到 21 行的第一个循环产生一个用于改变背景色的值。注意,在第 14 行和 15 行,外圈循环计数器被赋给另一个变量,并左移四位。将变量里的值左移四位,以便这个值能在属性字节的正确的位地址。

13~20 行的第二个循环产生一个用于改变前景色的值。在第 16 行,用第二个循环计数器的值加上左移四位以后的第一个循环计数器的值作为参数,调用 `textattr()` 函数。

第三个 `for` 循环仅仅在屏幕上打印一行字符。所显示行的背景色和前景色由循环中每一遍的 `textattr()` 函数改变。由于前景色取决于第二个循环,前景色的改变比背景色的改变要快。

14.4.2 使用直接控制台 I/O 以获得高性能

增强程序性能的一种方法是直接将显示数据送到视频 RAM。直接将数据送到视频 RAM 减少了必须调用的函数数目。因此,程序的速度就提高了。

Turbo C 有一个全局变量,它决定控制台输出是直接送给视频 RAM 还是由 BIOS 调用处理。控制台输出函数的变量是 `directvideo`。如果 `directvideo` 被置为 0,控制台输出就要由 PC 的 BIOS 函数处理。如果 `directvideo` 的值为 1,所有的控制台输出就直接送往视频内存。

下面的语句显示 `directvideo` 变量的当前值:

```
printf ("%d",directvideo);
```

变量的缺省值为 1,指示控制台输出直接送给视频 RAM。

使用直接的控制台 I/O 唯一的麻烦是,Turbo C 依赖于计算机是百分之百 IBM 兼容的。如果 `directvideo` 变量的值设为 1,而程序在其上运行的计算机都不是兼容的,则直接写视频内存不一定会成功。因此,如果要为并非与 IBM 兼容的计算机编写程序,把变量 `directvideo` 置为 0 要安全些。在 `directvideo` 被置为 0 之后,所有的视频函数都必须经过 BIOS 函数处理,不能直接写内存。

14.5 使用窗口函数

为了使用户在制作屏幕输出格式时具有更大灵活性,Turbo C 提供了窗口函数。有了这些函数,用户就可以定义和使用文本窗口。一个文本窗口(text window)是一个矩形区域,用户输出被限制在里边。这一节告诉读者如何使用 Turbo C 的窗口函数。

`Window()` 函数在屏幕上定义一个矩形区域,直接控制台 I/O 函数的输出就送在这个区

域中。一个窗口定义好后,控制台 I/O 函数的输出就只在窗口内部显示。注意调用 Window()函数的格式:

```
window (int left,int top,int right,int bottom);
```

传给 window()函数的参数是窗口左上角和右下角的坐标。在文本方式下,屏幕坐标是由原点格式指定的,意味着左上角坐标为(1,1)而不是(0,0)。

传给 window()函数的参数的最大值取决于当前的文本方式。如果当前文本方式是 EGA43 行方式,则最大的窗口其左上坐标为(1,1),右下坐标为(80,43)。

用户编写的每个程序都建立一个窗口。如果不在程序里显式调用窗口,则创建一个缺省窗口。自动创建窗口的大小等于当前方式屏幕的最大尺寸。程序 Wnddemo.c 演示了 window()函数的使用。

Wnddemo.c window()函数的演示

```
1  /* WNDDEMO.C   This short program creates two different
2                      windows. The first window uses the entire
3                      screen. The second window occupies only
4                      thue middle of the screen. The text sent
5                      to each window is a rolling ASCII pattern
6                      Thue same test pattern is sent to each window.
7  */
8
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include <conio.h>
12
13 main()
14 {
15     injt i, j;
16
17     textattr( 2 );
18     clrscr();
19
20     for( i = 0; i < 100; i++ )
21         for( j = 48; j <= 122; j++ )
22             putchar( j );
23
24     window( 5, 15, 75, 20 );
25     textattr( 30 );
26     clrscr();
27
28     for( i = 0; i <= 100; i++ )
29         for( j = 48; j <= 122; j++ )
30             putchar( j );
31     return 0;
```


32 }

程序 Wnddemo.c 是一个小程序,它使用了两个窗口。第一个窗口是程序开始执行时自动建立的。在缺省情况下,第一个窗口覆盖整个屏幕。在 17 行,调用了 `textattr()` 函数来设置将要显示的字符的属性。传送给 `textattr()` 的值 2 使得程序能在黑色背景上显示绿色字符。

20~22 行和 28~30 行的 `for` 循环打印滚动的 ASCII 模组。这些 `for` 循环打印值从 48 到 122 的 ASCII 字符。注意,在 `for` 循环里没有打印换行字符。输出允许从一行滚动到下一行。

第二个用到的窗口由 24 行的 `window()` 函数调用来定义。`window()` 创建一个窗口,其左上角在 (5,15) 处,右下角在 (75,20) 处。23 行的 `putch()` 函数所输出的所有字符均将在新窗口内部显示出来。新窗口以外的一切则维持原样不动。

窗口里显示的文本是否自动地滚动到下一行取决于一个全局变量 `_wscroll` 的值。如果 `wscroll` 的值设为 1,则当前行满之后,文本能自动地翻滚到下一行。如果 `_wscroll` 等于 0,文本就不会翻滚到下一行。如果 `_wscroll` 等于 0,而且当前行已满,则所写的下一个字符将会位于当前行的开头。要改变 `_wscroll` 的值,你必须将 `conio.h` 标题文件包含进来并说明 `_wscroll` 是一个外部整数。

可以使用 `gettextinfo()` 函数以知道当前文本方式是如何设置的。在 `gettextinfo()` 函数被调用时,函数将有关当前文本方式的信息存放在 `text_info` 结构里。`text_info` 结构定义如下:

```
struct text_info {
    unsigned char winleft;      /* Left coordinate      */
    unsigned char wintop;      /* Top coordinate      */
    unsigned char winright;    /* right coordinate    */
    unsigned char winbottom;   /* bottom coordinate   */
    unsigned char attribute;   /* text attribute      */
    unsigned char normattr;    /* normal attribute    */
    unsigned char currmode;    /* BW40,BW80,C40,C80,C4335 */
    unsigned char screenheight; /* bottom-top         */
    unsigned char screenwidth; /* right-left         */
    unsigned char curx;        /* current x coordinate */
    unsigned char cury;        /* current y coordinate */
};
```

以下的代码片段使用 `gettextinfo()` 函数,以获取有关当前文本屏幕的信息:

```
struct text_info my_struct;
gettextinfo(&my_struct);
printf("The screen is %d characters tall. \r\n",
    my_struct.screenheight);
printf("The screen is %d characters wide. \r\n",
    my_struct.screenwidth);
```

如果所需要的信息仅仅是当前光标的位置,可以用 `wherex()` 函数和 `wherey()` 函数。这两个函数都返回一个整数值,分别等于当前光标的 `x` 坐标和 `y` 坐标。注意下面的代码片段:

```
int x,y;
x = wherex();
```

```
y = wherey();
```

wherex()函数和 wherey()函数被用来获取文本窗口里光标的当前位置。注意,返回值是当前文本窗口内的位置;返回的位置不一定是屏幕上的位置。

读者可能已经注意到,使用窗口有个问题:当一个新屏幕遮盖住一个老的屏幕时,原屏幕上的信息丢掉了。有了 gettext()函数和 puttext()函数,就可以妥善解决这一问题。gettext()函数将屏幕内容复制到一个内存块中。如果想重新显示这个屏幕,只需简单地调用一下 puttext()函数。程序 scrsave.c 是程序 Wnddemo.c 的修改,它使用了 gettext()函数和 puttext()函数,能保存和恢复第一个屏幕。

scrsave.c 一个使用了函数 gettext()和 puttext()的程序

```
1  /* SCRSAVE.C      This program uses the gettext() function
2                      to save a screen, and the puttext()
3                      function to restore the screen.
4  */
5
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <conio.h>
9
10 int main( void )
11 {
12     int i, j;
13     char scr_buf[4000];
14
15     textattr( 2 );
16     clrscr();
17
18     for( i = 0; i <= 100; i++ )
19         for( j = 48; j <= 122; j++ )
20             putch( j );
21
22     gettext( 1, 1, 80, 25, scr_buf );
23
24     window( 5, 15, 75, 20 );
25     textattr( 30 );
26     clrscr();
27
28     for( i = 0; i <= 100; i++ )
29         for( j = 48; j <= 122; j++ )
30             putch( j );
31
32     cprintf( "\r\n\n" );
33     cprintf( "Press any key to continue. \r\n" );
```

```
34     while( ! kbbhit() );
35
36     window( 1, 1, 80, 25 );
37     clrscr();
38     puttext( 1, 1, 80, 25, scr_buf );
39     return 0;
40 }
```

程序 `scrsave.c` 里的程序在创建第二个窗口之前,使用了 `gettext()` 函数保存了第一个窗口里的文本。在信息显示在第二个窗口里之后,第一个窗口又被用 `puttext()` 函数恢复了。

在第 13 行,创建了一个可以装满一个窗口的全部文本的字符数组。注意,数组足够大,能装得下窗口里所有的字符的文本和属性字节。

15~20 行将第一个窗口用连续不断的 ASCII 字符序列填满。在屏幕被填满后,调用 `gettext()` 函数,然后屏幕的图象就保存在字符数组 `scr_buf` 里。然后,在 24~30 行时,另一个窗口被创建,并被填充以同样类型的 ASCII 字符模板。当显示循环完成后,用户被提示按一下键继续。

在按一下键继续后,一个新的覆盖整个显示器的窗口被创建并被清屏。一旦窗口被清除,`puttext()` 函数就被调用,原始的实足大小的窗口里的文本就恢复了。

14.6 了解 IBM-PC 的图形方式

前边我们说过,可以使用两种视频方式:文本方式和图形方式。在前面的几节里,读者学习了如何使用文本方式来处理字符数据。用户也可以用图形方式来显示图形数据,诸如图表(chart)、图(drawing)和其它图像(image)。在这一节里,读者将会学到图形方式是什么,怎样开始在程序中使用它。

14.6.1 像素点与调色板

像素点是图形方式的基础。一个像素点(pixel)是视频屏幕上的单个点。在图形方式的处理里,用户要控制屏幕上的每个像素点。

每种不同类型的显示器适配器和屏幕支持一种不同数目的像素点。例如,一个 CGA 适配器/显示器能最多显示 640×200 个象点,一台 IBM 8514 显示器最多能显示 1024×768 个像素点。屏幕的分辨率就是指屏幕能够显示的像素点数目。

对于一种适配器/显示器的组合,支持不止一种分辨率也是有可能的。可以显示的分辨率数目也取决于控制适配器和屏幕的软件。表 14.4 列出了 Turbo C 支持的显示器适配器和分辨率。

屏幕上每个像素点都有一个对应的内存位置。这个内存位置存放着一个值,它直接决定像素点的颜色。之所以能够决定是因为内存里存放的值是一个叫做调色板的表内的偏移量。调色板里的值决定了实际显示的颜色。

调色板(palette)是在任何给定时间都能显示在屏幕上的所有颜色的列表。在很多情况下,列在调色板里的颜色数目都只是屏幕上实际能够显示的颜色数的一部分。调色板里颜色

的数目受到可用的视屏内存的限制。

Turbo C 支持的彩色屏幕可以根据颜色控制的方法被划分成两组。第一组是 CGA 组,其中包括 CGA 视频方式、AT&T 视频方式和低分辨率的 MCGA 视频方式。第二组是 EGA 组,其中包括 EGA 和 VGA 适配器。

如果读者使用的是 CGA 型屏幕,则既可选择低分辨率方式又可选择高分辨率方式。低分辨率方式显示 320×200 个象点,有 4 种颜色。高分辨率方式显示 640×200 个像素点,只有 2 种颜色。

在 CGA 的低分辨率方式下,可以一次显示四种颜色,这四种颜色由所选的调色板决定。在 CGA 方式下可用的调色板有 CGAC0、CGAC1、CGAC2 和 CGAC3。每种 CGA 调色板均包含四种不同颜色。用户可以选择百种调色板的头一种颜色。表 14.5 显示了所有四种 CGA 调色板的固定颜色。

可以使用 `initgraph()` 函数(下节介绍)来选择一个想看的调色板。一旦选择好了合适的调色板,用户就可以用 `setcolor()` 函数来选择表 14.5 里的一种颜色,即为当前的着色。例如,如果 CGAC2 是当前的调色板,下面两个语句中的任一个都能将当前的着色变成绿色:

```
setcolor(1);  
setcolor(CGA_GREEN);
```

在每种 CGA 调色板里,头一种颜色(颜色 0)是背景色,可由用户设置。表 14.6 前面部分显示了 16 种颜色,用户可以从中选择。这个表包含了颜色编号和它们相应的符号常量。

要设置背景色,可以使用 `setbkcolor()` 函数。这个函数要求一个从表 14.6 中来的参数。例如,为了将背景色指定为蓝色,可以随使用下面哪种函数调用:

```
setbkcolor(1);  
setbkcolor(BLUE);
```

使用 CGA 的高分辨率方式没有使用 CGA 的低分辨率方式那么复杂。在高分辨率方式下,每个像素点均在 0 或 1。如果像素点值为 0,则像素点将为黑色背景色。如果像素值是 1,则像素点就为你所选的前景色。表 14.6 里的任一颜色均可用作前景色。在 CGA 的高分辨率方式下设置的前景色有点奇怪。要设置前景色,可以使用 `setbkcolor()` 函数,它设置硬件背景色。使用这个函数是因为 CGA 适配器有其特殊之处。例如,如果想以黄色画一个 CGA 高分辨率图,可以用下面两个语句之一:

```
setbkcolor(14);  
setbkcolor(YELLOW);
```

在 EGA/VGA 方式下,调色板里包含着可以一次显示的 16 种颜色(EGA 和 VGA 方式的唯一区别是 VGA 有比 EGA 更高的分辨率)。用户可以从一组 64 种可能的颜色中选取 16 种调色板颜色。缺省情况下,EGA 调色板里的颜色对应于 CGA 的背景色。表 14.6 后面部分列出了 EGA 调色板颜色号与每个颜色号对应的符号常量。

`setpalette()` 函数使用户能够改变 EGA 调色板里的每种颜色。这里是 `setpalette()` 函数调用的一个例子:

```
setpalette(int p_num, int color);
```

`p_num` 参数指定改变调色板的哪个表目。`color` 参数是将被显示颜色的实际颜色号。注意下面的函数调用:

```
setpalette (12,60);
```

这里,EGA 调色板的第 12 条表目将被改动,以显示淡红色。

用户可以用单个函数 `setallpalette()` 设置调色板里的所有颜色。这个函数重新设置当前的调色板。`Setallpalette()` 函数调用的格式如下:

```
setallpalette (struct palette type far * my _palette);
```

参数 `my _palette` 是一个结构,其中包含一个新的颜色号集合,将被复制到当前的调色板上。

在 EGA 方式下使用 `setbkcolor (color)` 函数时,color 参数被存储在 EGAA 调色板的第一个表目。

`getpalette()` 函数使用户能准确地知道当前调色板里使用了哪些颜色。对 `getpalette()` 的调用可以如下所示:

```
getpalette (struct palette type far * view _palette);
```

`view _palette` 参数是一个 `palettetype` 结构,有两部分。结构的第一部分是一个值,它指示调色板里有多少颜色。结构的第二部分是一个数组,包含当前调色板里所有的实际颜色号。

14.6.2 控制图形屏幕

在使用任意一个图形函数之前,必须先将屏幕从文本方式变换到图形方式。使用 `initgraph()` 函数即可改变视频方式。注意下面调用 `initgraph()` 函数的格式:

```
initgraph (int far * graphdriver,int far graphmode,char far * pathdriver);
```

`graphdriver` 参数告诉 `initgraph()` 需要显示器适配器装入哪种图形屏幕驱动程序。如果 `graphdriver` 参数等于 0,`detectgraph()` 函数将会被调用。`detectgraph()` 函数检查你的硬件,决定能使用的最高分辨率。

在 `initgraph()` 函数被调用后,由 `graphdriver` 所指定的图形驱动程序被从磁盘载入内存。一旦驱动程序载入内存,程序就能控制图形硬件。`graphmode` 指示用户将会使用哪种图形方式。例如,如果 `graphdriver` 设置为 CGA,`graphmode` 可以设置成四种 CGA 方式中任意一种。如果 `graphdriver` 被设为自检的图形硬件,`graphmode` 就没必要设置。在使用了 `detectgraph()` 之后,它给 `graphdriver` 和 `graphmode` 提供参数值。

`pathdriver` 参数告诉 `initgraph()`,在哪里可以找到图形驱动程序文件。如果 `pathdriver` 参数是 NULL,则图形驱动程序文件必须在当前目录里。

在 14.2.2.2 一节中列出了可以用作 `initgraph()` 函数的参数 `graphdriver` 的值。

`initgraph()` 被调用时,参数 `graphdriver` 可以与在 14.2.2.2 一节中所列的某一个值相等。不管是数字值还是驱动程序名都可以用作一个参数值。

表 14.4 显示了调用 `initgraph()` 函数时 `graphmode` 参数的可能选择。

在从表 14.4 中选择一个图形方式时,要找到驱动序列里正确的图形驱动程序。然后,固定想用的分辨率和调色板。最后,用从图形方式列成值列里选出的值作为 `initgraph()` 函数的 `graphmode` 参数。

在结束图形方式处理之后,可以调用 `closegraph()` 函数返回文本方式。这个函数将所有分配给图形函数的内存释放掉。然后,`closegraph()` 函数就结束了图形方式。`Closegraph()` 将系统返回到调用 `initgraph()` 函数前的文本方式下。

14.7 介绍 BGI 图形库

使用 Turbo C 进行图形程序设计很容易。通过建立 BGI 库, Borland 已经将所有图形程序中的艰难工作完成了(BGI 代表 Borland Graphics Interface—Borland 图形接口)。BGI 函数库允许用户一门心思地制作图像, 将用户从控制硬件的麻烦中解脱出来。这一节包容了 BGI 库里面一些最有用的函数。读者将会学到怎样使用画图函数和怎样控制屏幕。

14.7.1 使用画图 and 填充函数

利用 BGI 库函数所画的外观主要可分成两类: 未填充物体和填充物体。未填充物体(unfilled object)里面的颜色与背景色一样, 我们只能看见这个物体的轮廓。填充物体(filled object)里面的颜色由用户指定。在这一节里, 读者首先学到如何用函数画未填充物体。然后, 才会学到如何用函数画填充物体或者填充屏幕上某个区域。

关于使用图形绘图函数需要记住的一件事情是, 图形方式屏幕原点使用 0 格式。因此, 在图形方式下, 左上角的坐标(x,y)为(0,0)。在文本方式下, 左上角的坐标为(1,1)。

在 BGI 库里, 读者可以发现下面的绘图函数, 可以画未填充物体:

line()	画一条线
linereel()	画一条线, 它与当前位置有一段相对距离
lineto()	画一条线, 从当前位置到一个指定位置
rectangle()	画一个矩形
drawpoly()	画一个多边形
arc()	画一段弧
circle()	画一个圆
ellipse()	画一个椭圆

程序 graph1.c 显示出你很快就能使用这些绘图函数。注意, 下面三个程序清单是为一个 VGA 监视器编写的。如果用户想让程序在一个 CGA 监视器或另一种监视器上运行, 必需改变某些控制所画物体大小的值。

graph1.c 一个使用了基本绘图函数的程序

```
1  /* GRAPH1.C   This program shows how easy it is to use
2             the Borland C++ graphics functions. */
3
4  #include <graphics.h>
5  #include <stdlib.h>
6  #include <stdio.h>
7  #include <conio.h>
8
9  int main( void )
10 {
11     int graphdrv = DETECT;
12     int graphmode;
```

```

13     int x, y;
14     int radius = 30;
15
16     initgraph( &graphdrv, &graphmode, "\\tc\\bgi" );
17
18     for( x = 10; x <= 400; x += 10 ) {
19         y = x;
20         circle( x, y, radius );
21     }
22
23     while( ! kbhit() );
24     closegraph();
25     return 0;
26 }

```

graph1.c 里的短小程序显示了如何调用 BGI 库里的函数。程序使用一个 for 循环,在屏幕上画下一系列圆。

在调用 BGI 函数之前,必须先初始化图形方式。注意,11 行的变量 graphdrv 被用宏 DETECT 定义。当变量 graphdrv 用在 initgraph() 函数调用中时,它的值引发 initgraph() 自动检测最好的图形方式。对 initgraph() 的实际调用在 16 行。

在图形方式初始化后,就可以调用图形函数。18~20 行的 for 循环导致一系列斜穿屏幕的圆画在屏幕上。Circle() 函数需要 x、y 坐标,来定位圆的中心。和一个值指定圆的半径。只需三行循环就可以画整系列的圆。

在要结束图形方式时,必须返回到平常的文本方式。返回文本方式是由 24 行的函数 closegraph() 完成的。

程序 graph2.c 显示了如何调用另外一些 BGI 库里的绘图函数。它们包括 drawpoly()、ellipse() 和 line() 函数。

graph2.c 一个使用了更多 BGI 函数的程序

```

1  /* GRAPH2.C    This program shows how functions in the BGI
2                  library are called. */
3
4  #include <graphics.h>
5  #include <stdlib.h>
6  #include <stdio.h>
7  #include <conio.h>
8
9  int main( void )
10 {
11     int graphdrv = DETECT;
12     int graphmode;
13     int graaphmode;
14     int i, x, y;

```

```

15     int polypoints[] = { 10, 10,
16                           500, 10,
17                           500, 300,
18                           10, 10 };
19
20     initgraph( &graphdrv, &graphmode, "\\tc\\bgi" );
21
22     for( i = 1; i <= 100; i++ ) {
23         x = random( 640 );
24         y = random( 480 );
25         ellipse( x, y, 0, 360, x, y );
26     }
27
28     cleardevice();
29
30     drawpoly( 4, polypoints );
31     line( 0, 450, 640, 450 );
32     while( ! kbhit() );
33
34     closegraph();
35     return 0;
36 }

```

在 21~25 行的 for 循环里,ellipse()函数使用三组参数。第一组参数是一对(x,y)坐标,指定椭圆的中心。第二组参数指定椭圆的开始和结束角。第三组参数指定椭圆大小。

程序清单 14.7 里,for 循环在随机的位置以随机大小绘制 100 个椭圆。椭圆的大小和定位由 22 行和 23 行的 random()函数控制。

在第 29 行,drawpoly()函数画出一个任意所想要的形状的多边形。多边形的边数由函数 drawpoly()的第一个参数决定。多边形的边所被画的地方由 drawpoly()的第二个参数决定。第二个参数是一个指针,该指针指向一个列出了多边形所有顶点的(x,y)坐标的数组。drawpoly()所用的顶点数组在 14~17 行说明和初始化。

在第 30 行,line()函数在屏幕的底部附近画一条线,line()需有起点和终点参数。

第二个主要绘画函数组是填充物体函数。这些函数画出的物体被某种所选择的模式和颜色填充着。下列函数可以画填充物体:

bar()	绘制并填充一个二维条形
bar3d()	绘制并填充一个三维条形
fillellipse()	绘制并填充一个椭圆
fillpoly()	绘制并填充一个多边形
pieslice()	绘制并填充一个陷饼薄片
sector()	绘制并填充一个椭圆形陷饼片
floodfill()	填充一个有界区域

使用填充图像函数与使用不填充图像函数一样快速方便。程序 drawbar.c 显示了一个

条形图程序,它使用了 bar3d()函数。

drawbar.c 一个使用了填充图像函数的程序

```
1  /*  DRAWBAR.C   This program draws three 3-D bars, the size
2             of which is determined by the user.  */
3
4  #include <graphics.h>
5  #include <stdlib.h>
6  #include <stdio.h>
7  #include <conio.h>
8
9  int main( void )
10 {
11     int graphdrv = DETECT;
12     int graphmode;
13     int i, j, k;
14
15     clrscr();
16     printf( "Enter 3 numbers from 0 to 9 => " );
17     scanf( "%d %d %d", &i, &j, &k );
18
19     initgraph( &graphdrv, &graphmode, "\\tc\\bgi" );
20
21     i = 450 - ( 30 * i );
22     j = 450 - ( 30 * j );
23     k = 450 - ( 30 * k );
24
25     bar3d( 20, i, 120, 450, 10, 1 );
26     bar3d( 140, j, 240, 450, 10, 1 );
27     bar3d( 260, k, 360, 450, 10, 1 );
28
29     while( ! kbhit() );
30
31     closegraph();
32     return 0;
33 }
```

程序 drawbar.c 是一个简单的程序,它能画一个缩小的条形图。在第 17 行,scanf()函数读入三个从 0 到 9 的整数值。这三个整数值在 21~23 行被按比例变为尺寸。用户提供的整数值每增加 1,条形图的高度就增加 30 个象素点。从 450 减去条形图的高度给了条形顶的 y 坐标。

25~27 是对 bar3d()函数的调用。bar3d()需要指定条形图的左、顶、右、底(x,y)坐标的参数。最后两个参数指定三维阴影的深度和是不是要一个顶在条上。如果最后一个参数非 0,则条上要上顶。如果最后一个参数为 0,则条的上部不放顶。

Turbo C 提供了一些函数,允许你控制绘图函数:

moveto()	将当前位置移至(x,y)
movrel()	移动当前位置一个相对距离
getlinesettings()	取当前行的方式,宽度和模式
setlinestyle()	改变当前行的方式和宽度
getaspectratio()	取当前特征系数
setaspectratio()	改变当前特征系数
getfillpattern()	返回一个用户定义的填充模式
getfillsettings()	取当前填充模式和颜色
setfillstyle()	改变当前填充模式和风格

一个对 setlinestyle()函数的调用如下:

```
setlinestyle (int linestyle,unsigned upattern,intthickness);
```

linestyle 参数指定将要使用哪种直线方式。upattern 是一个 16 位(二进制)数,你可以用它改变直线的风格。如果要使用 upattern 的值,linestyle 参数必须被设置为 USERBIT _LINE。在 16 位的 upattern 值中的每一位都对应着屏幕上的一位。如果在 upattern 值里某一位设为 1,则如果显示在屏幕上,这一位都能显示出来。linestyle 参数不影响弧、圆、椭圆或者馅饼片。

thickness 参数指示所给的直线是一像素点宽还是 3 像素点宽。如果 thickness 被设置为 NORM _WIDTH,直线将以一像素点宽绘制。如果 thickness 设置为 THICK _WIDTH,直线将被画为三像素点宽。

例如,如果语句

```
setlinestyle (SOLID _LINE,0,THICK _WIDTH);
```

被加进列表 14.4 或者 14.9,所有用直线绘制的图像其直线都有三像素点宽。

14.7.2 控制屏幕和视口

开始在屏幕上绘图时,读者会很快就发现需要清屏后重新开始。Turbo C 正好提供了清屏的函数;cleardevice()函数。调用 clearvideo()函数后,整个屏幕被清除,以待再用。

图形系统有 1~4 个图形页,用户可以使用这些图形页。一页(page)就是一个内存块,它存放有整个图形屏幕。所有图形函数将输出送往的那一页叫做活动页(active page),正在显示的那一页叫做可见页(visual page)。

表 14.9 列出了支持不止一个视频页的适配器和方式。

表 14.9 支持不止一个视频页的适配器和方式

适配器	方式	页数
EGA	EGALO	4
	EGAHI	2
	EGAMONHI	2(W/256K 内存)
HERC	HERCMONHI	2
VGA	VGALO	2
	VGAMED	2

setactivepage(int page)函数设置活动页,图形输出即将送到这一页。setvisualpage

(intpage)函数选择要显示的视网页。

在文本方式下,可以设置一个窗口(window)区(屏幕上的一个矩形区域),所有文本函数都会将输出送到这个区域。在图形方式下可以用相似的方法,不过设置的区域叫做视口(viewport)。

setviewport()函数用来控制视口的位置及它所占的地方大小。对 setviewport()函数的调用如下所示:

```
setviewport (int left,int top,int right,int bottom,int clipping);
```

传送给 setviewport()函数的位置参数都是屏幕的绝对坐标。setviewport()函数还指定视口是不是可裁剪的。如果 clipping 打开(不为 0),则一个超出了视口尺寸的物件将被裁剪,其超出视口边界部分不给输出。如果想清除视口里的所有图形,可以使用 clearviewport()函数。

14.8 在图形方式下显示文本

在图形方式下显示文本与在文本方式下显示文本不同。因为在图形方式下,控制着屏幕上每个象素点,在显示文本时,用户必须决定如果将有个字母画出来。

在图形方式下显示文本最早的方法是显示已画好的字符。使用一个字符预制集可以节省大量时间,因为这样不必在每次需要的时候再去重建字符。

一个同样字体字符的集合叫字库(font)。Turbo C 的 BGI 库提供了好几种字库,供用户在图形图像里显示文本信息时使用。这一节告诉读者如何使用 Turbo C 的字库和图形方式文本函数。

14.8.1 BGI 字库

BGI 函数库有两大系列字库:点阵字库和向量字库。只有在考虑字库是怎样建造的时候,两大系列字库之间的差异才昭然若揭。

点阵字库里的字符是通过将一个方格里的每一点置为 0 或 1 制成的。方格的实际尺寸(以象素点为单位)因字符打印时所用的比例系数的差别而不同。例如,内部的点阵字库将每个字符定义在一个 8×8 的方格内。不过,字符显示时所占的尺寸并不一定就是一个 8×8 象素点的区域。字体可以放大,以便所显示的字体比定义的尺寸大。

向量字库与点阵字符有着根本的不同。向量字库里的字符是由一系列向量构成的,这些向量告诉程序如何画这个字体。

由于向量字库构成方法如此,它们放大时可以产生较好的效果。在放大一个点阵字体时,只是放大了基本网格模式。一个点阵字体放大之后,字体的外形会带有许多毛刺。但是,向量字库放大后却能提供较好的效果。不管字符是不是小,定义常量字体的向量都能产生较好的图像。

Turbo C 的 BGI 软件包有一个内部 8×8 点阵字库。在使用图形函数时,点阵字库总是可用的。每个向量字库存放在一个单独的文件里,文件的扩展名是 CHR。

在建立一个图形程序时既可以让向量字库留在盘上,也可以把它们链接到可执行文件中。如果将字库留存盘上,在运行的时候将会分配内存将字库装入。BGIOBJ 实用程序可以

将字库文件较换成 OBJ 文件,以便链接到用户程序里。如果要制作一个软件分发给其他人,将字库文件链接进程序是比较方便。使用库变成程序的一部分,既可以方便安装,又能防止用户不小心删掉字库。将字库文件链接进程序,付出的代价仅仅是 EXE 文件变大了。

14.8.2 使用图形方式下的文本函数

Turbo C 提供了好几种函数,使用用户能控制 BGI 字体的大小和外开。程序 showtext.c 显示了其中一些图形方式下输出文本的函数是怎样使用的。

showtext.c 一个使用了 BGI 文件函数的程序

```
1  /* SHOWTEXT.C This program demonstrates the use of
2      the BGI fonts. */
3
4  #include <graphics.h>
5  #include <stdlib.h>
6  #include <stdio.h>
7  #include <conio.h>
8
9  int main( void )
10 {
11
12     int graph _drv = VGA;
13     int graph _mode = VGAHI;
14     char test _str[] = "This is a test. This is only a test!";
15
16     initgraph( &graph _drv, &graph _mode, "\\tc\\bgi" );
17
18     settxtstyle( 4, 0, 0 );
19     outtext( test _str );
20
21
22     settxtstyle( 2, 1, 0 );
23     outtextxy( 600, 200, test _str );
24     setusercharsize( 1, 1, 2, 1 );
25     settxtstyle( 3, 0, 0 );
26     moveto( 320, 240 );
27     settxtjustify( 1, 1 );
28     outtext( test _str );
29
30     while( ! kbhit() );
31
32     closegraph();
33     return 0;
34 }
```

在使用任何 BGI 输出文本函数前,必须确信在图形方式下。在程序 showtext.c 里,15 行的 `initgraph()` 函数初始化了图形方式。注意,传给 `initgraph()` 函数的参数导致图形方式被设为高分辨率 VGA 方式。

17 行和 18 行调用 `settextstyle()` 函数和 `outtext()` 函数。`outtext()` 在当前位置显示文本。显示文本的字体、大小和方向均由 `settextstyle()` 函数决定。`settextstyle()` 指定使用哪个字库,字符向哪个方向输出和字符的尺寸是多在。调用 `settextstyle()` 函数的格式如下:

```
settextstyle(int font,int direction,int charsize);
```

`font` 参数既可以用符号常数又可以用整数值,指示使用哪个 BGI 字库。可用的 `font` 参数值列在表 14.10 里。

表 14.10 font 参数值

字库名	整数值	描述
DEFAULT_FONT	0	缺省 8×8 点阵字库
TRIPLEX_FONT	1	三倍字库(triplex font)(向量)
SMALL_FONT	2	小字库(向量)
SANS_SERIF_FONT	3	Sans_serif 字库(向量)
GOTHIC_FONT	4	哥特(Gothic)字库(向量)

`Settextstyle()` 函数的 `direction` 参数指定文本将以哪种方向输出。文本可以从左到右印出或从底到顶印出。用作 `direction` 参数的值在表 14.11 中列出。

表 14.11 direction 参数

方向名	整数值	描述
HORIZ_DIR	0	文本从左到右输出
VERI_DIR	1	文本从底到顶输出。

`settextstyle()` 的 `charsize` 参数指定显示字形的放大系数。例如,如果 `charsize` 等于 1,则 8×8 的点体字体就显示在一个高 8 像素点宽 8 像素点的块里。如果 `charsize` 等于 3,则 8×8 的点阵字体在一个 24 像素点×24 像素点的区域显示。`charsize` 参数可以将字体放大到 10 倍于普通尺寸的大小。

如果 `charsize` 置为 0,向量字体可以用 `setusercharsize()` 函数来放大。`setusercharsize()` 需要 4 个参数。头两个参数指定向量字符将被转换成的宽度。后两个参数指定对 BGI 字体垂直方向的放大系数。

传给 `setusercharsize()` 的参数的格式如下:

```
setusercharsize(int multx,int divx,int multy,int divy);
```

如果 `multx` 为 2 且 `divx` 为 1,字体字符将是原来的两倍宽。如果 `multy` 为 1 且 `divy` 为 1,则字符高度将是缺省值的一半。

程序 showtect.c 的 18 行有一个 `outtext()` 函数,将参数表所指的字符串打印出来。

`outtext()`从当前光标位置开始输出串。

在 22 行, `outtextxy()` 函数从由函数参数指定的位置开始输出。与 `outtext()` 相似, `outtext()` 输出参数表里指向的串。

利用 `settextjustify()` 函数, 可以将文本在中央、左边或右边对齐。利用 `settextjustify()`, 也可以将显示文本垂直对齐。在程序 `showtext.c` 里, `settextjustify()` 在 27 行调用。

另外三个函数使用户能检查视屏方式的参数。 `gettext setting()` 允许用户检查全部图形方式参数。 `gettextsetting()` 所收集的信息存储在 `textsetting type()` 的一个结构里。 `textheight()` 和 `textwidth()` 函数返回一个指出一个字符串有多高多宽的值(以像素点为单位)。

第十五章

存储模式

如果读者是一位程序设计的新手,那么你最新奇的程序设计任务是学习怎样使用计算机的内存。Intel 的 80×86 处理器芯片系列使用的编址方式是“分段存储编址方式”。对于使用分段编址方式的内存,不能用一种类型的指针去访问内存的任意位置。因为计算机的内存被划分为多个 64K 字节的段,为了访问内存中的某个特定位置,必须既知道它正确的段地址,又知道它在段内的偏移地址。如果所有这些很令人困惑不解,这一章将会告诉读者,计算机是怎样使用内存的以及怎样才能控制内存。

15.1 80×86 的体系结构

要想了解 Intel 家族的处理器芯片是怎样进行内存编址的,就必须先知道关于芯片制造的一些情况。一旦明白了处理器芯片的设计,也就能轻松地掌握如何使用分段编址技术。

15.1.1 段(Segment)、节(Paragraph)以及偏移地址(Offset)

用于最早的 IBM-PC 的 8088 芯片是一种 16 位的处理器,其地址总线为 20 位,数据总线成为 8 位。数据总线是布在系统主机板上的电路轨迹群,用于将数据从 CPU 传输给其它设备。PC 上的 8 位数据总线一次只允许传输 8 位(1 字节)数据。

地址总线是一系列的电路轨迹,它决定将使用哪一个内存地址。可寻址的内存地址的数目取决于地址表线的中线的数目。可以用 2^n (一根单线上可以表示的值的数目)的 n (n 是地址总线的位数)次幂来计算可寻址空间。例如对于 PC,可寻址的内存地址数目等于 2 的 20 次方,也就是 1M (如果读者有一台以 8088 为主处理器的计算机,就会知道只能使用 640K 的内存。剩余的 384K 内存只能用于系统功能)。

8088 芯片是一种 16 位的微处理器。自然,一种 16 位的处理器在某一时间只能工作于 16 位数据下。因此,8088 所能使用的最大整数数值为 65535,也就是 64K。处理器所能使用的最大整数数值决定了可能被计算的最大内存地址。

这就产生了一个问题:一种只能产生 16 位数值的 CPU 是怎样控制一个 20 位的地址总线的呢? 秘诀在于,地址是用内存的两个字来形成的(在 8088 里,一个字是两个字节)。内存的一个字包含偏移地址,另一个字则包含一个段地址。偏移地址是一个 16 位的数值,因此能最多寻址 64K 的内存空间。段地址也是一个 16 位的数值;但是,由于段地址将会被左移 4 位,所以段地址代表了一个 20 位的数值,这里的“位”、前面的“位”,以及将来提到的“位”,大多是指二进制中的一位,希望读者有能力寻清楚)。段地址与偏移地址结合起来,就可以访问

任意一个内存地址。

看一看下面的例子,就会明白一个 16 位数值是怎样左移 4 位后,产生一个 20 位数值的:

	二进制值	十六进制值
移位之前	0001 0010 0011 0100	1234
移位之后	0001 0010 0011 0100 0000	12340

在这个例子中,被存储在段地址寄存器中的数值是 1234(第一行的 16 进制数值)。依靠它自己,这个 16 位段地址尚不能对内存空间寻址;这个 16 位数值必须被变换成一个 20 位数值。这种数值的变换是通过将这个二进制数值左移 4 位来实现的。将一个数值左移 4 位相当于对它乘以 16(16 是十进制数)。

可以看到,在前边的例子中的第二行里,一旦段地址数值被左移四位之后,其结果就是一个 20 位数值。这个 20 位数值是内存里某个位置的实际地址。

在看到记载的段地址,或者查看一个段地址寄存器的值时,所看到的数值将是一个 16 位值。要得到实际上的段地址,要记得在 16 位数值的后面对于二进制表示加上 4 个 0(或者对于十六进制表示加上 1 个 0)。结果就会是一个 20 位的段地址。

如果读者稍精通数学的话,就可能已经观察到,段地址只能寻址内存的再第 16 个字节。确实如此。因为对段地址的 4 位左移总使最后 4 位恒为 0。每个这样的 16 字节块叫做字(paragraph)。

为了寻址一个特殊的位置,段地址必须与偏移地址相结合。偏移地址里所装的,是从段地址开始的一个偏移量,以字节为单位。例如,如果段地址等于 2BC00(十六进制表示),偏移地址等于 00FF,则实际的内存地址就为 2BCFF。

书写一个内存地址的标准表示方法是段地址:偏移地址。其中段地址值是存储于段寄存器中的 16 位值。因此,内存地址 2BCFF 可以写成 2BC0:00FF。

偏移地址之间有可能会重叠。使用重叠的偏移量,不同的地址可能指向内存中的同一位置。举个例子,下列地址就指向同一内存位置:

2BC0:00FF

2000:BCFF

2111:ABEF

2A00:1CFF

偏移地址可以相互重叠的事实在使用 far 指针和 huge 指针工作时很重要。本章的下一节将告诉读者,可使用哪些种类的指针和它们是怎样寻址计算机内存的。

80286 和 80386 的新特点

Intel 的 80286 和 80386 是更新的微处理器,它们比 8088 要先进得多。尽管如此,极少有程序利用了这些芯片的新增强的能力。Microsoft 的 Windows 3.0 正在改变所有这一切。Windows 3.0 通过利用 80286 和 80386 芯片的硬件新特点,戏剧般地增强了软件工作者的生产能力。

与 8088 相似,Intel 80286 也是一个 16 位微处理器。但是,80286 仍有一些新增的特点,这使它比 8088 的功能要强大得多。这里列出这些特点的一部分:

□ 提供了一个真正的 16 位数据总线,能一次访问两个字节。

- ☐ 提供了一个 24 位的地址总线,将最大寻址能力从 1M 提高到 16M。
- ☐ 提供了多任务能力,允许处理器在同一时间运行不只一个程序。
- ☐ 提供了虚拟内存技术。使计算机在能访问通常机器所有的内存之外,还能访问更多的内存。这些“额外的”内存存储在硬盘上,在需要的时候移送到 RAM 里。80286 的虚块内存方式使得处理器可以访问高达 1G 字节的内存。

80386 包含比 80286 更多更先进的特点。这些特点包括以下几点:

- ☐ 它是一个真正的 32 位处理器,可以同时处理和传输 4 个字节的数据。
- ☐ 它提供了 32 位地址总线,可以访问高达 4G 字节的内存。
- ☐ 它提供了虚拟内存方式,通过使用一个 46 位内存地址,可以访问 64 terra 字节的空间(1 terra=2⁴⁰)。

15.1.2 CPU 的地址寄存器

为了增强整体性能,处理器将存储在处理器芯片内中且经常使用的信息保存在几个寄存器中。寄存器(register)是一种特殊的 16 位存储单元,能够被迅速地访问。

8088 芯片有 14 个寄存器,它们用于存放信息和内存地址。图 15.1 是 8088 的寄存器的图表。不是所有的寄存器都用于寻址目的。

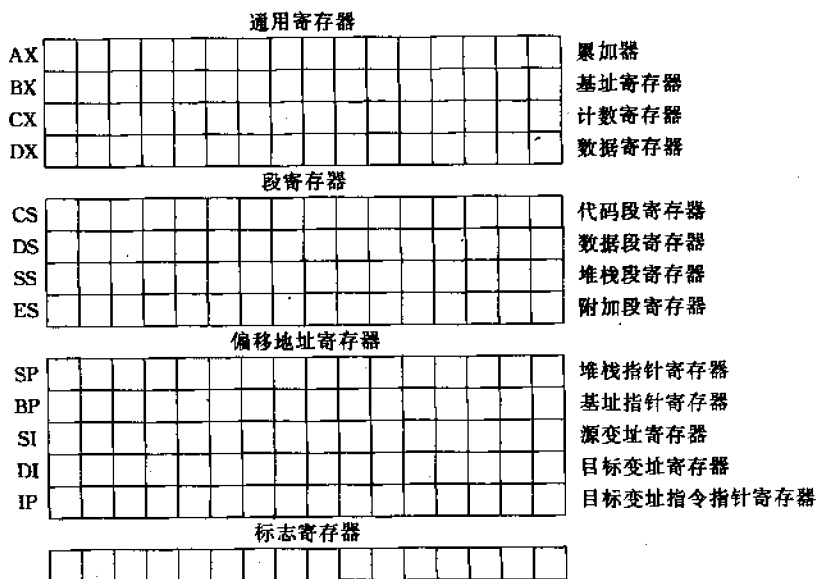


图 15.1 8088 的内部寄存器

尽管通用寄存器用于许多不同的场合,它们每个还各有自己特定的功能。这里是通用寄存器的一些特定用途:

- AX 用于数值累加和其它数学运算中;
- BX 用于变址操作;
- CX 用于变址和循环计数;

DX 用于一般操作和数学运算中。

第二组寄存器是段地址寄存器。每个段地址寄存器存放一个 16 位的段地址值。实际的段地址可以通过将段地址值左移 4 位后得到。当段地址与偏移量结合使用时,段地址值指向一个 64K 的内存块。段地址寄存器和它们所对应的存储块在下面列出:

CS 代码段地址,是当前执行的程序所装入的地方;

DS 数据段地址,是当前程序所用的数据存放的地方。

SS 堆栈段地址,是为程序的堆栈区准备的。堆栈作为临时存储器,保存程序函数调用的踪迹,和函数调用时传送的参数。

ES 附加段地址,用于存放程序数据,处理内存段之间的数据传输。

以上的段地址中,不见得有两个都指向不同的段。实际上,所有四个指针可以指向一个地址。程序的存储模式决定了段地址是否相同。存储模式在本章稍后的“Turbo C 的六个存储模式”一节里讲述。

第三组寄存器包含那些偏移地址寄存器。前边已经讲过,偏移地址是用来与段地址相结合,以访问一个特定内存地址的。下面列出了偏移地址所定位的数据类型:

SP 堆栈指针是与 **SS** (Stack Segment 一堆栈段)寄存器用在一起,以寻址栈顶的某个精确位置的。

BP 基址指针是用来作堆栈内的变址,以发现参数或者自动变量的。

SI 源变址寄存器可以用作数据段内的变址。**SI** 寄存器还可以用于其它目的。

DI 目标变址寄存器的用法与源变址寄存器一样。

IP 指令指针用作当前正在执行程序中的程序计数器(PC)。指令指针指向下一个将要执行的指令。这个寄存器不能被 Turbo C 访问。

标志寄存器是一个特殊的寄存器,它存放着有关 CPU 和已经被执行了的指令的当前状态的信息。在 8088 处理器中,标志寄存器占 16 位。在 80386 处理器中,标志寄存器占 32 位。80386 标志寄存器增加的 16 位用来存储 8 关于 80386 芯片及处理方式的特殊信息。

15.2 near 指针、far 指针和 huge 指针

一个指针就是一个数据对象,它存放着另一个数据对象的址。我们还介绍了可以使用不同类型的指针:整数指针、浮点数指针、字符指针,甚至还有函数指针。在这一节里,将会介绍还可以使用一些附加类型的指针。不过,这些指针并不是指向不同类型的对象,而是指向不同的内存位置。

这一节将会说明,所用的指针类型取决于所指向的对象是不是与段寄存器之一在同一个内存段里。还会讲到如何使用 near 指针、far 指针和 huge 指针。

15.2.1 选择想要的指针大小

Turbo C 有三种不同的指针:near、far 和 huge。所使用的存储模式指定了这三种类型中的一种作为缺省指针类型。当然,也可以显式地把一个指针说明为想要的任意类型。本小节解释各种指针类型的不同特点。near 指针用起来最简单,也最受限制。它只能存放一个 16 位的地址。由于它只有 16 位地址值,因此 near 指针只能仅仅寻址 64K。请记得 16 位内存里所

能存入的最大整数是 65535。于是, `near` 指针的大小也就限制了它只能访问内存中一个 64K 大小的块。

在访问一个实际内存位置的时候, `near` 指针必须与某个段寄存器里的地址结合使用。这个地址指向一个内存块的开始, 指针里的地址则是块内的偏移地址。使用哪个段寄存器取决于指针指向哪种类型的对象。当 `near` 指针指向一个函数时, `CS`(code segment) 寄存器提供段地址。当一个 `near` 指针指向一个数据对象时, 由 `DS`(data segment 一数据段) 寄存器提供段地址。

`near` 指针很容易使用, 因为在操作一个函数的时候, 用不着说明其段值。由于 `near` 指针不存放段值, 所以指针可以直接进行比较。在 `near` 指针上进行算术运算也很容易, 因为计算不会牵涉到段值。

一个 `far` 指针是一个 32 位指针, 它能访问任何内存位置。一个 `far` 指针包含了段值和偏移量。 `far` 指针最大的优点是它们能访问大于 64K 的代码段和数据段。 `far` 指针把程序和数据从 64K 的限制中解放出来。

使用 `far` 指针也产生一些麻烦。本章前面介绍了一个内存位置可以有几个不同的段: 偏移值。例如, 下面三个段: 偏移地址指向内存里的同一位置:

```
4000:000A
3FFF:001A
3ED2:12EA
```

假定有三个 `far` 指针, 其中每个分别装有上面的一个值。因此, 每个 `far` 指针都正好指向内存里的同一位置。尽管如此, 如果试图在这些指针之间执行逻辑比较, 则比较的结果将会指出各指针之间是不等的。尽管这些指针都指向内存里的同一位置, 但是存在这三个 `far` 指针中的值(地址)在数学上却是不等的。因此, 如果要比较两个 `far` 指针, 就必须确信它们的段值是相同的。如果需要执行逻辑比较, 用 `near` 指针或者 `huge` 指针将会方便一些。

对存放在一个 `far` 指针里的地址执行算术运算可能不会产生期望的结果。在对一个 `far` 指针加上一个数值或者减去一个数值时, 只有偏移量受到了影响。我们不能用数学运算来改变 `far` 指针里的段值。如果存在一个 `far` 指针里的值 4000:FFFF 被增加 1, 在 `far` 指针里的新值将等于 4000:FFFF, 而不是 5000:0000。

与 `far` 指针相似, `huge` 指针包含有一个 32 位的地址, 因此能指向内存的任意位置。不同的是, 存在 `huge` 指针里的地址经过了规格化处理。

一个规格化的指针是一个存放在其中的地址已经过一次转换操作的指针。地址被变换了, 以使尽可能多的地址可在段地址里存放。然后, 偏移量便只包含从 0 到 F(16 进制)的数值。

以下的例子显示了规格化处理是如何工作的(所有值均以十六进制表示):

16 位段地址:	3256
左移位后的段地址:	32560
16 位偏移地址:	00C4
20 位内存地址:	32624
规格化后的段地址:	3262
规格化后的偏移地址:	0004

在这里,原 16 位段地址被左移 4 位以产生一个 20 位段地址,然后加上原偏移地址。实际的内存位置由地址 32624 指示。一旦实际地址被计算出来后,规格化处理就可以开始了。规格化以后的段地址等于实际地址十六进制表示的高四位数字。实际地址剩下的低位数据就是新的规格化地址的偏移量。

与 far 指针相比,一个规格化的 huge 指针有两个优点。首先,对 huge 指针可以实施逻辑运算。第二,huge 指针的段址可以通过数学运算加以改变。

huge 指针的规格化允许对 huge 指针值进行逻辑比较。对于 far 指针,两个指针可以指向同一内存地址,但是却有着不同的段值和偏移量。但是由于规格化处理,任意指向同一位置的 huge 指针都是相等的。

与 far 指针不同,当 huge 指针的偏移量不停变化时,其段地址也会改变。由于你在使用 huge 指针时可以改变段地址,huge 指针就可以利用一个单个的大于 64K 的对象。

对于 huge 指针来说,麻烦在于 huge 指针的算术运算需要额外的开销。在执行 huge 指针的算术运算时,必须调用特定的函数。处理速度因此会降低。

15.2.2 near, far 和 huge 说明符

可以在程序里说明一个指针,用不用修饰语 near, far 和 huge 都行。使用一个指针修饰符就会覆盖缺省类型并产生一个指定类型的指针。例如,下面这行

```
char far *char_ptr;
```

说明了一个 far 型字符指针,不管程序的缺省指针类型是什么。但是,下面一行

```
char *char_ptr;
```

则说明了一个具有程序的缺省指针类型的字符指针。如果程序是以 tiny(微小)存储模式编译的,则缺省指针类型是 near。如果程序是以 large(巨大)存储模式编译的,则缺省指针类型是 far。

程序 farmem.c 中的小程序使用 far 指针来保存一个已被存放在远堆(far heap)里的磁盘文件的踪迹。

farmem.c far 指针使用的一个示例

```
1  /* FARMEM.C This program demonstrates the use of far
2     pointers by allocating enough memory to
3     hold an entire file. For this program to
4     work correctly, it must be compiled under
5     the COMPACT, LARGE, or HUGE Memory Model.  */
6
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <alloc.h>
10 #include <fcntl.h>
11 #include <io.h>
12
13 int main( void ) {
14     unsigned long file_size;
```

```
15     unsigned long i;
16     int         file_handle;
17     char         *memory_ptr1;
18     char         *memory_ptr2;
19
20
21     if( (file_handle = open("printers.txt", O_RDONLY | O_TEXT)
22 ) == -1 ) {
23         printf( "Could not open file. \n" );
24         printf( "Exiting program. \n" );
25         exit( 0 );
26     }
27
28
29     file_size = filelength( file_handle );
30     if( file_size > farcoreleft() ) {
31         printf( "Not enough memory to store file. \n" );
32         printf( "Exiting program. \n" );
33         exit( 0 );
34     }
35     else{
36         memory_ptr1 = (char far *)farmalloc( file_size );
37         memory_ptr2 = memory_ptr1;
38         file_size = read( file_handle, memory_ptr2, file_size );
39     }
40
41     for( i = 1; i <= file_size; i++ ) {
42         putchar( *memory_ptr2 );
43         memory_ptr2 ++;
44     }
45
46     close( file_handle );
47     farfree( memory_ptr1 );
48     return 0;
49 }
```

程序 farmem.c 显示了用于编译一个程序的存储模式是怎样影响程序中说明的指针的。在说明 Turbo C 的存储模式之前,先回味一下程序 farmem.c 中程序的目的。

程序 farmem.c 取到存在磁盘上一个文本文件的大小,分配足够的空间以将该文件装入 RAM,读进此文件,然后打印此文件。

从 21 行到 26 行使用 open() 函数打开一个存放在磁盘驱动器上的文本文件。如果返回给 open() 的值等于 -1,则 open() 失败,程序被中止。

29 行里的 filelength() 函数取得在 21 行和 22 行打开的文件的长度。然后,在第 30 行,

比较文件长度与 farheap(远堆)里可得到的内存里。如果有足够的内存可以存放该文件, farmalloc()就分配一块内存空间,刚好够存放这个磁盘文件。

在第 38 行里,一个对 read()函数的函数调用将整个磁盘文件读进来,并放在远堆里。read()函数能够以一个函数调用读进整个文件,因为由 filelength()返回的值能告诉 read(),从磁盘文件读进多少个字节。

41~43 行里的 for 循环一次一字节时一步步走过分配的内存块。这不是显示块内数据最有效的办法,但是提供了一种简易地操作块的数据的方法。一字节一字节走过块使得返回和修改此程序很容易。

程序 farmem.c 使用了两个指针作为分配内存块内的变址。这两个指针分别在第 17 和 18 行说明。尽管这些指针说明时未带任何修饰符,但是程序是在 Large 存储模式下编译的,这种存储模式导致所有的指针具有缺省类型 far,所以这两个指针的类型均为 far。按照如下的方式来处理两个说明绝对百分之百地合法,但却是多余的:

```
char far *memory_ptr1;  
char far *memory_ptr2;
```

不管什么时候,如果需要说明一个类型与程序的缺省类型不同的指针,只要将修饰符放进说明里就行了。例如,假设在程序 farmem.c 所示的程序里需要用到一个 near 指针,对该指针的定义可以如下:

```
Void near *extra_ptr;
```

Turbo C 另有 4 种指针修饰符,可以与 near 指针一起使用: _cs、_ds、_es 和 _ss。其中任一个修饰符都指定一个段地址,可与 near 指针一起使用。修饰符提供的段地址与相匹配的段地址寄存器对应。例如, _ds 修饰符告知指针用作指针的段地址,即存储在数据段寄存器中的地址。下面一行代码给出了一个包含着 _ds 修饰符的说明:

```
char _ds my_ptr;
```

15.3 六个 Turbo C 存储模式

我们已多次提到了存储模式。一种存储模式(memory model)就是一个编译程序的可选项,它决定分配多少内存空间给程序的数据和代码。这一节将在如何选择适合程序需要的存储模式问题上给读者以指导。

15.3.1 决定使用哪种存储模式

Turbo C 提供了六种不同的存储模式,可以从中选择。每种存模式有其不同的特点,影响到将会为程序的代码和数据分配的空间的多少。表 15.1 列出了各种存储模式和它们各自的特点。

表 15.1 Turbo C 的存储模式

模式	描述
Tiny(微)	这种存储模式是最小的。在 tiny 存储模式下编译的程序能够被转换成 COM 文件。使用这种存储模式时,所有四个段寄存器指向同一个地址。程序的代码、数据和堆栈必须安装在 64K。在没有任何空间是空闲时,应使用 tiny 存储模块。

续表 15.1

模式	描述
Small(小)	对于 Small 存储模式,代码和数据段是不一样的,但每个段都限制在 64K 范围内。堆栈包含在数据段里。许多应用软件在 small 存储模式下运行很好。因为这种存储模式只使用 near 指针,程序性能得到了增强。
Medium(中)	medium 存储模式对程序的代码使用 far 指针,而对数据使用 near 指针。因此,一个 medium 模式程序的代码可以最多达到 1M 长,但数据被限制在 64K 范围内。这种模式对于大而复杂、并且不使用数据的程序很合适。
Compact(紧凑)	Compact 模式是 medium 模式的镜像,它对代码用 near 指针,但对数据用 far 指针。程序的数据最大可以占领 1M 内存,但程序自身被限制在 64K 范围内。Compact 模式对于处理大量数据的较短程序比较合适。
Large(大)	Large 模式用于处理大量数据的大程序。无论对程序的代码还是数据,都用 far 指针。因此,代码和数据最多都能占据 1M 内存。
Huge(巨大)	与 Large 存储模式相似,huge 存储模式对代码和数据都使用 far 指针。这两种存储模式之间的不同之处是 huge 存储模式甚至把对 static(静态)数据的 64K 限制也弃置一边了。因此,static 数据也能使用多于 64K 的内存。可以为最大的程序使用 huge 存储模式。

在编译一个程序前,必须选择想要哪一种存储可选项。要选择一个存储可选项,可从 Options(可选项)菜单里选择(Compiler 编译程序)命令,然后选择 Code Generation(代码生成)。对话框显示一系列存储模式的按钮。只要按一下所要的存储模式,就能开始编译。

由于每个存储模式分配代码和数据内存不同,每种模式都有其独特的缺省指针修饰符设置。图 15.2 总结了每种存储模式的缺省指针修饰符。如果模式的指针是 near 指针,则缺省的段修饰符也被列出在该图中。

存储模式	函数指针	段修饰符	数据指针	段修饰符
TINY	near	_cs	near	_ds
SMALL	near	_cs	near	_ds
MEDIUM	far	N. A.	near	_ds
COMPACT	near	_cs	far	N. A.
LARGE	far	N. A.	far	N. A.
HUGE	far	N. A.	far	N. A.

图 15.2 六种存储模式的缺省指针和段修饰符存储模式和指针修饰符

15.3.2 以混合模式编程

总有一天,我们会碰见将几个在不同存储模式下编译的模块链接成一个程序的问题。在链接这些混合的模式时,如果不小心,肯定会产生麻烦。

考虑一下,当一个 Small 模块试图调用一个 large 模块里的函数时,会发生什么情况? 从图 15.2 中你可以知道,在缺省情况下,Small 模块用 near 指针调用函数,large 模块用 far 指针调用函数。当一个 Small 模块里的函数调用一个 large 模块里的函数时,这个函数调用用

一个 near 指针来执行。但是 large 模块里被调用的函数需要一个 far 指针。这种情况很简单地就不能运转。

有一种使用混合模式和具有函数型程序的方法。秘诀在于使用一种函数原型,显式定义函数的指针类型。程序 module1.c 和 modul2.c 显示出两个程序模块,每个在不同的存储模式下编译,可以被链接成一个有效的程序。

程序 modul1.c 是一个源文件,它包含有被另一个程序模块调用的函数。程序 modul1.c 在 Large 存储模式下被编译成一个目标文件。程序 modul1.c 里的函数因此用 far 指针调用。

程序 modul2.c 是混合模式例子的主源文件,它包含一个 main() 函数,调用在程序 modul1.c 里的其它函数。程序 modul2.c 在 small 内存模式下被编译成目标文件。因此,程序 modul2.c 里的函数调用要用 near 指针来完成。

程序 module1.c 一个混合模式程序的第一个模块

```
1  /* MODULE1.C This source code is the first program module
2      used in the demonstration of mixed-model
3      programming. This module was compiled under
4      the LARGE memory model. */
5  int func1( void )
6  {
7      return 1;
8  }
9
10 int func2( void )
11 {
12     return 2;
13 }
14
```

程序 module2. 一个混合模式程序的第二个模块

```
1  /* MODULE2.C This source code is the second program module
2      used in the demonstration of mixed-model
3      programming. This module ws compiled under
4      the SMALL memory model. */
5
6  #include <stdio.h>
7  #include <stdlib.h>
8
9  extern int far func1( void );
10 extern int far func2( void );
11
12 void main( void )
13 {
14     printf( " %d %d\n", func1(), func2() );
15 }
```

尽管程序 modul2.c 里的程序是在 small 存储模式下编译的,程序还是能够调用程序

modul1.c 里的 far 函数, modul2.c 的第 9 行和第 10 行就是调用了一个 far 函数。程序 modul2.c 包含的说明显式说明了外部函数是 far 函数。这种显式 far 函数说明导致使用 far 指针, 尽管缺省指针类型是 near。

如果程序 modul2.c 里的外部说明编码如下:

```
extern int func1();  
extern int func2();
```

则当规划文件被编译时, 链接程序将给出错误信息。

对于作为参数传送的指针也应该小心才是。例如, 如果对程序 modul1.c 里的 func1() 的函数定义是:

```
int func1 (int *i)  
{  
    ...  
}
```

则在程序 modul2.c 里对 func1() 的函数说明应该是:

```
extern int(int far *i);
```

如果有一个 small 模式程序, 需要链接 Turbo C 的库函数, 可能需要先创建一个特殊的标题文件。库函数是 large 模式函数, 使用 far 指针。为了在 Small 模式程序中使用库函数, 必需建立标题文件的一个特殊备份。在标题文件的新拷贝里, 必须显式将所有的函数和指针说明成 far 类型的指针。

记住, 成功的混合模式编程取决于使用正确的函数和指针说明。如果在这方面增加一些特别的关注, 混合模式编程就会成功。

15.4 创建 COM 型的可执行程序文件

COM 文件与 EXE 文件一样是可执行文件, 但是 COM 文件在代码和数据的尺寸上有严格的限制。本节说明使用 COM 型文件的优点和这些可执行文件怎样用 Turbo C 创建。

15.4.1 使用 COM 文件

一旦编译和链接了源代码, 就创建了一个可执行文件, 它具有所有需要的部分, 能在 DOS 上运行。一般地, 所创建的可执行文件其扩展名为 EXE。尽管如此, 也可以创建另一种类型的可执行文件: COM 文件。

一个 COM 文件与 EXE 最大不同的地方是它们的大小。因为 COM 文件在 tiny 内存模式下编译成的, 所有 COM 文件的代码、数据和堆栈必须固定在一个 64K 的内存块中。这个 64K 的尺寸是有些过于苛刻, 但许多程序很容易合于这种内存限制。

COM 文件的最大优点是它的速度快。因为一个 COM 文件总是使用 near 指针, 所有与 far 和 huge 指针相关联的额外开销都被避免了。由于不需要处理 far 指针和 huge 指针花费额外的时间, 大大增强了程序的性能。

程序 little.c 显示了一个简单的管道程序, 它是在 tiny 存储模式下编译的, 并链接成一个 COM 文件。

程序 little.c 一个编译和链接成 COM 文件的小程序

```
1  /* LITTLE.C This program is compiled with the TINY
2      memory model in order to generate a COM
3      file. The resulting OBJ file must be linked
4      with TLINK to create the COM file.  */
5
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <ctype.h>
9
10 int main( void )
11 {
12     FILE * file_ptr;
13     char proc_char;
14
15     if( ( file_ptr = fopen( "printers.txt", "rt" ) ) == NULL )
16     {
17         printf( "Unable to open file.\n" );
18         printf( "Exiting program.\n" );
19         exit( 0 );
20     }
21
22     while( proc_char != EOF ) {
23         proc_char = fgetc( file_ptr );
24         proc_char = toupper( proc_char );
25         putchar( proc_char );
26     }
27
28     fclose( file_ptr );
29     return 0;
30 }
```

程序 little.c 里的程序很适合于链接成一个 COM 文件。这个短小的过滤程序读入一个文件,然后将所有的小写字母较换成大写字母。

程序的核心包含在 while 循环里,在 22~26 行。磁盘文件一次一字符地扫描。如果读到的字符是一个小写字母,则把它转换成一个大写字母。输入字母如果必要就被转换,然后把结果写到视频屏幕上,不管是否转换。

这个程序之所以适合于编译和链接成一个 COM 文件,是因为它包含的代码和数据很少。程序中仅有两个变量,它们是一个文件指针和一个 char 类型的变量。

在程序 little.c 中的程序在 Turbo C 的集成化开发环境(IDE)里输入并编译成一个目标文件。由于文件要变成一个 COM 文件,因此选择了 tiny 存储模式。创建可执行的 COM 文件需要使用 TLINK 实用程序。

TLINK 是一个独立存在的链接程序,与 Turbo C 的软件包在一起。如果读者习惯于使

用 IDE,就可能熟悉 TLINK 的用法。尽管如此,TLINK 对一些链接任务来说仍是必需的,比如要创建一个 COM 文件。为了创建一个 COM 文件,需要使用带 /t 选择项的 TLINK 去链接 OBJ 文件。TLINK 命令行的格式如下所示:

```
tlink optionopt C0x myobjs,exe,mapopt),mylibsopt  
overlayopt emu|fp87 mathxopt Cx
```

以下的讨论说明了每个要求的 TLINK 参数是怎样使用的。当程序 little.c 的程序被链接时,不需要使用任一可选的参数。

option 可以在命令行的任意地方放置,告诉 TLINK 使用哪个可选项。要看可以用的可选项的完整列表,请不带任何参项地执行 TLINK。一个 /t 可选项告诉 TLINK,应该产生一个 COM 文件。

C0x 指定即将使用哪种初始化模块。每个存储模式有一个单独的初始化模块,通过将 x 用存储模式名的首字母替换来选择一个正确的模块。例如,medium 存储模式要求初始化模块的参数项 C0m。初始化模块参数项是一个必要的参数。

myobjs 指定要链接哪些目标文件。目标文件名被一个空格与 C0x 参数分开,不能用逗号隔开。如果有不止一个目标文件要链接,目标文件名之间要用空格分开。不用把目标文件名的 OBJ 扩展也输进去。

exe 参数项是另一个必需的参数,指定将要创建的可执行文件名。不必指定文件的扩展名,因为它可以自动生成。

下面的参数项是可选的:map 指定将要生成的映射文件名;mylibs 列出任意用户提供的库文件;overlay 指示覆盖管理函数库是否被链接;emu|fp87 指定使用哪个浮点库,如果程序中需要一个浮点库的话。C0x 是个必需的参数,它指定使用哪个运行期库。每种存储模式都有各自不同的库。x 可选项与 C0x 里的 x 一样,除了 tiny 和 Small 模式都使用 CS.LIB 运行期库外。

一个实际的、链接程序 little.c 中程序的目标文件的 TLINK 命令如下:

```
tlink /t \tc\lib\C0t little,little,,\tc\lib\CS
```

库文件的路径用来确保链接时使用正确的库文件。请注意,在初始化模块参数与目标文件名之间没有一个用来分开的逗号。也请注意,逗号被用来划分可选项域的界限。在要将文件分开时,也需要用逗号。

第十六章

与汇编语言的接口

本章介绍如何编写汇编代码,以使之能和 Turbo C 一起正确地工作。如果不熟悉如何编写 8086 汇编程序以及如何定义段、数据常量等,阅读有关 Turbo 汇编手册以作更多的了解。

16.1 混合语言程序设计

Turbo C 很容易使 C 程序调用汇编语言编写的过程,反过来,用汇编语言编写的程序也可以很容易地调用 C 过程。本节将提供有关这方面接口的信息。

16.1.1 参数传递顺序

Turbo C 支持两种向函数传递参数的方法。一种是标准 C 方法,我们首先解释这种方法;另一种是 Pascal 方法。

16.1.1.1 C 参数传递顺序

假定已经说明了下面的函数原型:

```
void funca(int p1, int p2, long p3);
```

缺省时,Turbo C 使用 C 参数传递顺序,也称作 C 调用约定。当这个函数(funca)被调用时,参数按从右向左的顺序(p3, p2, p1)压入栈中,然后在栈中压入返回地址。所以,如果进行了这个调用

```
main()
{
    int i, j;
    long k;
    ...
    i=5; j=7; k=0x1407AA;
    funca(i, j, k);
    ...
}
```

堆栈中的情形将是(在返回地址被压入到栈以前):

```
sp+06: 0014
sp+04: 07AA K=p3
sp+02: 0007 j=p2
sp    0005 i=p1
```

在 8086 中,栈是从高地址向低地址生长的,所以此时 i 在栈顶。

被调用的过程无需知道有多少参数被压入到栈中,它假定所需要的参数在那里。

更重要的是被调用的过程不需要将参数弹出栈,因为这个工作将由调用过程完成。例如,编译程序从这个 main 函数的 C 源代码中产生的汇编语言是这样的:

```

mov    WORD    PTR[bp-8], 5           ; Seti=5
mov    WORD    PTR[bp-6], 7           ; Setj=7
mov    WORD    PTR[bp-2], 0014h       ; Setk=0x1407AA
mov    WORD    PTR[bp-4], 07AAh
mov    WORD    PTR[bp-2],             ; push high word of k
mov    WORD    PTR[bp-4],             ; push low word of k
mov    WORD    PTR[bp-6],             ; push j
mov    WORD    PTR[bp-8],             ; push i
call   NEAR    PTR    funca           ; call funca(push addr)
add    sp, 8                          ; Adjust stack

```

仔细注意一下最后一条指令: add sp, 8。编译器知道有多少参数被压入到栈中,也知道由调用 funca 而压入的返回地址已经被 funca 结束处的 ret 指令弹出栈,所以,最后一条指令完全恢复了调用函数以前的堆栈。

16.1.1.2 Pascal 参数传递顺序

另一种方法是参数传递的标准 Pascal 方法(也称作 Pascal 调用约定)。这并不说明可以在 Turbo C 中调用 Turbo Pascal 函数。如果 funca 被声明为:

```
void pascal funca(int p1, p2, long p3);
```

则当调用这个函数时,参数按从左向右的次序(p1, p2, p3)被压入栈中,紧接着将返回地址也压入到栈中。所以如果执行调用

```

main()
{
    int i, j;
    long k;
    ...
    i=5; j=7; k=0x1407AA;
    funca(i, j, k);
    ...
}

```

堆栈中的情形将是(在返回地址被压入栈之前):

```

sp+06,0005  i=p1
sp+04,0007  j=p2
sp+02,0014
sp          :07AA  k=p3

```

最大的差别是,除了改变了压入参数的次序,Pascal 参数传递假定被调用的过程(funca)知道有多少参数传递给了它,并相应地由 funca 自己来调整栈。换句话说,调用 funca 的汇编语言实际上是

```

push WORD [bp-8]    ; push i
push WORD [bp-6]    ; push j

```

```

push WORD [bp-4]    ; push high word of k
push WORD [bp-2]    ; push low word of k
call NEAR PTR funca ; Callfunca [push add]

```

注意,在调用之后没有 `add sp, 8` 指令,相反,funca 在结束时使用指令 `ret 8` 返回到上一层函数以清除栈。

缺省情况下,用 Turbo C 编写的所有函数使用 C 参数传递方法。唯一的例外是使用 -P 编译选择项(在 Code Generation 对话框中的 Pascal 选项);在这种情况下,所有的函数使用 Pascal 参数传递方法。此时,可以通过使用 `cdecl` 修饰符强迫一个给定的函数使用 C 参数传递方法,例如:

```
void cdecl funca (int p1, int p2, int p3);
```

这忽略了一 P 编译器命令。

为什么需要使用 Pascal 调用约定呢? 有两个主要原因:

- 可能调用现有的使用 Pascal 调用约定的汇编语言过程。
- 生成的调用码稍微不同,因为在调用以后不必消除栈。

使用 Pascal 调用约定可能出现一些问题。首先,它不如 C 调用那样可以传递一个参数数目可变的参数序列,因为被调用的过程必须知道传递了多少参数并相应地清除栈。传递太多或太少的参数几乎总会严重的问题,而采用 C 约定的过程就不会产生这种问题。其次,如果使用 -P 编译器选择项,必须保证所使用的任何标准 C 函数都包含相应的头文件。因为不这样,Turbo C 对这些函数将一律使用 Pascal 调用约定进行访问,并且程序也不能正确地进行连接。

头文件以 `cdecl` 说明每个函数,所以如果包含它们,编译器将使用 C 调用约定。

总而言之:如果打算在 Turbo C 程序中使用 Pascal 调用约定,确保尽可能多地使用函数原型,这些函数显式地以 `cdecl` 或 `pascal` 说明。这种情况下允许产生“Function call with no prototype(函数调用没有使用原型)”警告是非常有用的,这可以确保每个被调用的函数都有一个原型。

16.2 建立从 Turbo C 对 ASM 的调用

当编写汇编语言过程时,必须遵守几个约定:(1)确保连接器可得到必要的信息;(2)确保文件格式和 C 程序使用的内存模型一致。

16.2.1 简化的段指令

一般来说,汇编语言模块由三部分组成:代码、初始化数据和非初始化数据。每组信息被组织在自己的段中,段名取决于 C 程序使用的内存模式。Turbo 汇编(TASM)提供三种简化段指令(`.CODE`、`.DATA` 和 `.DATA?`),可以用它们来定义这些段。在汇编中使用 `MODEL` 指令定义的内存模式使用缺省的段名。例如,如果程序使用 `small` 内存模式,可以使用下表所示的简化段指令组织每个汇编模块。

```

.MODEL SMALL
.CODE

```

```

...代码段...
.DATA
...初始化数据段...
.DATA?
...未初始化的数据段...

```

16.2.2 标准段指令

有时,可能不希望对所使用的内存模式使用缺省的段名,这时可以使用表 16.1 中列出的标准段指令:

表 16.1 汇编语言文件格式

code	SEGMENT	BYTE PUBLIC 'CODE'
	ASSUME	CS:code, DS:dseg
代码段.....	
code	ENDS	
dseg	GROUP	_DATA, _BSS
data	SEGMENT	WORD PUBLIC 'DATA'
初始化数据段...	
data	ENDS	
_BSS	SEGMENT	WORD PUBLIC 'BSS'
非初始化数据段...	
_BSS	ENDS	
	END	

在这个表中的标识符 code、data 和 dseg 有特定的替换方法,这取决于所使用的内存模式,表 16.2 说明对于每种模型应该怎么办,其中的 filename 是模块的名字,在 NAME 指令和标识符替换中都使用它。

注意对 huge 模式,没有 _BSS 段,GROUP 定义也被完全删除。一般来讲, _BSS 是可选的,如果需要使用它,只能自己定义。

得到一个汇编语言样例的最好方式,是将一个空的 C 语言程序编译到 ASM(使用 TCC 选项 -S),并参阅一下生成的汇编代码。

16.2.3 定义数据常量和变量

内存模式也影响如何定义指向代码、数据或两者的数据常量。表 16.2 说明了这些指针的形式,这里 xxx 是被指向的地址。

有几个指针定义使用 DW(定义字),而其它的使用 DD(定义双字),它们表明结果指针的大小。数值和文本常量按常规定义。

变量的定义和常量是一样的。如果不需要给变量初始化特定的值,可以在 `_BSS` 段说明它们,在需要置初始值地方放一个问号(?)。

16.2.4 定义全局和外部标识符

一旦建立一个模块, Turbo C 程序需要知道它调用哪些函数以及它引用哪些变量。汇编语言过程同样能够调用 Turbo C 函数,或引用 Turbo C 中说明的变量。

表 16.2 标识符替换和内存模型

模式	标识符	可替换代码和指针
微,小	<code>code=_TEXT</code>	<code>Code; DW _TEXT;xxx</code>
	<code>data=_DATA</code>	<code>Data;DW DGROUP;xxx</code>
	<code>dseg=_DGROUP</code>	
紧凑	<code>code=_TEXT</code>	<code>Code;DW _TEXT; xxx</code>
	<code>data=_DATA</code>	<code>Data;DD DGROUP;xxx</code>
	<code>dseg=_DGROUP</code>	
中	<code>code=filename _TEXT</code>	<code>Code;DDxxx</code>
	<code>data=_DATA</code>	<code>Data;DD DGROUP;xxx</code>
	<code>dseg=_DGROUP</code>	
大	<code>code=filename _TEXT</code>	<code>Code;DDxxx</code>
	<code>data=_DATA</code>	<code>Data;DD DGROUP;xxx</code>
	<code>dseg=_DGROUP</code>	
巨	<code>code=filename _TEXT</code>	<code>Code;DDxxx</code>
	<code>data=filename _DATA</code>	<code>Data;DDxxx</code>

当进行这些调用时,需要明白有关 Turbo C 编译器和连接程序的几个问题。在说明一个外部标识符时,编译器在目标程序中保存这个标识符时,在标识符前面自动添加一个下划线。Pascal 标识符的处理和 C 不一样——它们全部大写并且不带有前缀下划线。

C 标识符使用的下划线是缺省的但也可以选择。它们可以使用 `-U` 命令来关闭命令行选择项。然而,如果使用标准 Turbo C 库,将会遇到麻烦,除非重新建立库。为做到这一点,需要 Turbo C 的运行库的源代码。从 Borland 公司还可以得到更多的信息。

如果在源文件中的任何 asm 代码引用了任意 C 标识符(数据或函数)。这些标识符必须以下划线字符开始(除非使用前面描述的任何一种非标准 C 语言来定义)。

Turbo 汇编(TASM)对大小写不敏感。当汇编一个程序时,所有的标识符以大写方式保存起来。TASM 的 `/mx` 选择项使得公用和外部标识符变得对大小写敏感,所以任何符号应很好地匹配才行。在所举的例子中,都将关键字和指令写成大写,而所有其它的标识符和伪码都是小写;这和 TASM 参考手册中的风格相一致。也可以根据习惯,自由地使用全部大写

(或全部小写), 或任何混合形式。

为了在外部汇编语言模块中引用标识符, 需要把它们说明为 PUBLIC。

例如, 如果编写了一个包括整型函数 max 和 min 以及整型变量 MLAXINT、Lastmax 和 Lastmin 的模块, 应将语句

```
PUBLIC _max, _min
```

放到代码中, 并将语句

```
PUBLIC      _MAXINT, _lastmax, _lastmin
_MAXINT    DW  32767
_lastmax    DW  0
_lastmin    DW  0
```

放到数据段中。

Turbo 汇编 2.0 扩充了许多指令的语法以便于选择语言定义符。例如, 如果在模块的 MODEL 指令中定义 C, 所有的标识符名将以下划线开头的形式保存在目标模块中。这个特点也可以逐个指令地定义。使用 Turbo 汇编 2.0 的 C 语言定义符, 前述的说明将被写成:

```
PUBLIC      C _max, min
PUBLIC      C _MAXINT, lastmax lastmin
_MAXINT     DW  32767
lastmin     DW  0
lastmax     DW  0
```

16.3 建立从 ASM 中对 Turbo C 的调用

使用 EXTRN 语句, 汇编语言模块可以引用在 Turbo C 程序中说明的变量。

16.3.1 引用函数

为能够从汇编语言过程中调用一个 C 函数, 必须在汇编语言模块中使用语句:

```
EXTRN fname; fdist
```

对它进行说明。这里, fname 是 C 函数的名字, fdist 是 near 或 far, 这取决于 fname 函数是 near 或 far。所以, 如果在代码段中有下面的语句:

```
EXTRN _myCfunc1: naer, _myCfunc2: far 就可以在汇编语言过程中调用
myCfunc1 和 myCfunc2.
```

使用 Turbo 汇编 2.0 的 C 语言定义符, 上面语句的下划线就可以去掉, 写成:

```
EXTRN C myCfun1: near, myCfun2: far
```

16.3.2 引用数据

为了引用变量, 应在数据段放置适当的 EXTRN 语句, 格式是:

```
EXTRN vname; size
```

这里 vname 是变量名, size 表示变量的大小。

size 可能的值是:

```
BYTE (1 个字节)      QWORD (8 个字节)
```

WORD (2 个字节) TBYTE(10 个字节)

DWORD(4 个字节)

所以,如果 C 程序有下面的全局变量:

```
int i, jarray[10];
char ch;
long result;
```

可以使用下面的语句使自己的模块能够很好地引用它们:

```
EXTRN _i:WORD, _jarray:WORD, _ch:BYTE, _result:DWORD
```

或使用 Turbo 汇编 2.0 的 C 语言定义符:

```
EXTRN C i:WORD, jarray:WORD, ch:BYTE, result:DWORD
```

注意:如果使用 huge 内存模型,EXTRN 语句必须放在任何段以外,这适用于函数和变量。

16.4 定义汇编语言过程

现在已经知道如何建立各种程序,下面介绍如何用汇编语言实际编写函数。这要考虑几个重要的事情:传递参数、返回值和正确使用寄存器约定。

假定想编写一个在 C 中有下面的函数原型的函数 min:

```
extern int min(int v1,int v2);
```

而且 min 返回传递给它的两个参数中的较小者,汇编语言的格式将是:

```
PUBLIC      _min
_min PROC   NEAR
...
_min ENDP
```

在这里假定 min 是一个近函数;如果它是一个远函数,应将 NEAR 替换成 FAR。注意已经在 min 的开头添加了下划线,所以可以正确地连接这个函数。如果在 PUBLIC 语句中使用 Turbo 汇编 2.0 的 C 语言定义符,汇编程序将自动完成这个任务。

16.4.1 传递参数

首先决定使用哪一种参数传递约定,除非有适当的理由,应避免使用 Pascal 约定而采用 C 约定。这说明当 min 被调用时,栈是这样的:

```
sp+04:v2
sp+02:v1
sp:  返回地址
```

要取得参数值而不想从栈中弹出任何东西,可以先保存基指针(BP),然后将栈指针(SP)送到基指针中,最后使用基指针直接检索栈中的值。注意,将 BP 压入栈时,参数的相对偏移增加了 2 个字节,因为栈中多了两个字节。

Turbo 汇编 2.0 提供了一种方便地引用函数参数和处理栈的方法。继续往下阅读,重要的是理解栈的工作过程。

16.4.2 处理返回值

如果函数返回一个整数值,对 16 位(2 个字节)值(char, short, int, enum 和 near 指针),使用 AX 寄存器;对 32 位(4 个字节)值(包括 far 和 huge 指针),还要使用 DX 寄存器,高字节或指针的段地址放在 DX 中,低字节或指针的偏移量在 AX 中。

float, double 和 long double 型的值返回在 80X87 栈项(TOS)寄存器 ST(0)中。如果使用 80X87 仿真库,值被返回到仿真库的 TOS 寄存器中。调用函数必须将这个值拷贝到需要它的地方。

1 个字节长的值返回到 AL 中,2 个字节长的值返回到 AX 寄存器中。4 个字节长的值返回在 AX:DX 中。3 个或大于 3 个字节的价值放入一个静态数据区,然后返回指向那个地址的指针(小数据模式时指针在 AX 中,大数据模式指针在 DX:AX 中)。被调过程必须将要返回的值拷贝到指针所指向的地方。

对 min 一例,所处理的全部是 16 位值,所以只需将结果放在 AX 寄存器中。

代码可能象下面所示的那样:

```

PUBLI          _min
_min PROC      NEAR
    push        bp                ;Save bp on stack
    mov         bp,sp            ;Copy sp into bp
    mov         ax,[bp+4]         ;Move v1 into ax
    cmp         ax,[bp+6]         ;Compare with v2
    jle         exit             ;If v1 >= v2
    mov         ax,[bp+6]         ;Then load ax with v2
exit:
    pop         bp                ;Restore bp
    ret                     ;And return to c
_min ENDP

```

如果将 min 说明为 far 函数,将发生什么变化呢? 主要的差别是在入口处栈是这样的:

```

sp + 06: v2
sp + 04: v1
sp + 02: return segment
sp      : return offset

```

这表明相对于栈的偏移已经增加 2 个字节,因为 2 个额外字节(用于保存返回的段地址)必须压入到栈中。min 的版本是:

```

PUBLIC          _min
_min PROC      FAR
    push        bp                ;Save bp on stack
    mov         bp,sp            ;Copy sp into bp
    mov         ax,[bp+6]         ;Move v1 into ax
    cmp         ax,[bp+8]         ;Compare with v2
    jle         exit             ;If v1 >= v2
    mov         ax,[bp+6]         ;Then load ax with v2

```

```

exit:      pop      bp          ;Restore bp
           ret          ;And return to C
_min       ENDP

```

注意, v1 和 v2 的偏移量增加了 2 个字节, 以反映栈中多出的字节。

现在, 如果使用 Pascal 参数传递顺序, 在入口处栈将是这样的 (假设 min 是 NEAR 函数):

```

sp + 04: v1
sp + 02: v2
sp      ;return address

```

另外, 对于标识符 min 必须遵循 Pascal 调用约定: 大写并且没有前缀下划线。

除了交换 v1 和 v2 在栈中的位置, 还要求在 min 返回时, 自己清除它的栈空间, 可以通过在 RET 指令中的操作数来弹出多余的字节。在本例中, 必须弹出由 v1 和 v2 使用的额外四个字节 (返回地址被 RET 指令自动弹出)。

下面是修改后的过程:

```

PUBLIC      MIN
MIN        PROC      NEAR          ;Pascal version
           push      bp          ;Save bp on stack
           mov       bp, sp       ;Copy sp into bp
           mov       ax, [bp+6]   ;move v1 into ax
           cmp       ax, [bp+4]   ;Compare with v2
           jle       exit        ;If v1 > v2
           mov       ax, [bp+4]   ;Then load ax with v2
exit:      pop       bp          ;Restore bp
           ret       4           ;Clear stack and return
MIN        ENDP

```

最后一个例子说明为什么要使用 C 参数传递顺序。假定重新定义 min 为:

```
int min(int count, ...);
```

min 现在可以接受任何数目的整数并返回它们中的最小者。然而, 因为 min 无法自动地知道传递过来几个值, 所以第一个参数是一个计数值, 说明它的后面有多少个值需要实际比较。

例如, 可以这样使用它:

```
i = min(5, j, limit, index, lcount, 0);
```

假定 i, j, limit, index 和 lcount 都是整型 (或相容的类型)。在入口处栈是这样的:

```

sp + 08: (etc.)
sp + 06: v2
sp + 04: v1
sp + 02: count
sp      ;return addr

```

修改后的 min 版本是:

```

PUBLIC      _min
min        PROC      NEAR

```

```

        push    bp                ;Save bp on stack
        mov     bp,sp             ;Copy sp into bp
        mov     cx,[bp+4]         ;Move count into cx
        cmp     cx,0              ;Compare with 0
        jle     exit              ;If <=0, then exit
        lea     bx,[bp+6]         ;Make bx point to first value
        mov     ax,[bx]           ;Move first value into ax
        jmp     Itest             ;And test loop
compare: cmp     ax,[bx]           ;Compare with next value
        jle     Itest            ;If next value is lower
        mov     ax,[bx]           ;Then load ax with next value
Itest:   add     bx,2              ;Move to new value
        loop    compare          ;Then loop back
exit:    pop     bp               ;Restore bp
        ret                     ;And return to C
min      ENDP

```

这个版本可以正确处理 count 各种可能的值:

- 如果 count=0,min 返回 0.
- 如果 count=1,min 返回参数列表中的第一个值。
- 如果 count≥2,min 自动地进行比较并返回参数表中的最小值。

现在当编写自己的函数时,已经明白如何处理栈,Turbo 汇编 2.0 的某些新的扩充可以自动地建立临时变量,清除在 PROC 内使用的栈空间以及很容易地访问参数,它们全部使用自己定义的语言约定。

使用新扩充的第一个 min 版本:

```

PUBLIC      C   MIN
min         PROC C NEAR v1,WORD, v2,WORD
            mov     ax,v1
            cmp     ax,v2
            jle     exit
            mov     ax,v2
exit:       ret
min         ENDP

```

Pascal 风格版本可被写作:

```

PUBLIC      PASCAL MIN
min         PROC PASCAL NEAR v1,WORD, v2,WORD
            mov     ax,v1
            cmp     ax,v2
            jle     exit
            mov     ax,v2
exit:       ret
min         ENDP

```

注意到除了将关键字 PASCAL 替换为 C 之外,所编写的代码是一样的,然而由汇编程

序实际生成的代码却不同(对应于原先的例子)。参阅 Turbo 汇编手册可全面了解这些新的不依赖于语言的特点。

与通常的 C 函数一样,外部汇编语言过程必须遵守一定的编程规则以便和覆盖管理器一起正确地工作。

如果一个汇编语言过程调用任何覆盖过程或函数,汇编语言过程必须是 far 型的,并且必须使用 BP 寄存器建立栈框架。

16.5 寄存器约定

在 min 中使用了几个寄存器(BP、SP、AX、BX、CX),其中必须注意的寄存器是 BP,进入时,在栈中保存 BP,退出时恢复它。

必须注意的其它两个寄存器是 SI 和 DI,这两个寄存器可能被 TurboC 用作寄存器变量。如果在一个汇编语言过程中使用它们,应在进入时保存它们(可能在栈中),在退出时恢复它们。然而如果使用 -r- 选择项(Code Generation 对话框中 RegisterVariable 应该为非检查状态)编译 Turbo C 程序,则不必保存 SI 和 DI。

如果使用 -r- 选择项,则必须使用警告信息,参见附录 A 有关命令行编译器的内容,详细了解有关寄存器变量选择项。

寄存器变量 CS、DS、SS 和 ES,根据所用的存储模式具有特定的值。它们的关系是:

Tiny	CS=DS=SS
	ES=任意值
Small,Medium	CS!=DS,DS=SS
	ES=任意值
Compact,Large	CS!=DS!=SS
	ES=任意值 (每个模块有一个 CS)
Huge	CS!=DS!=SS
	ES=任意值 (每个模块有一个 CS 和 DS)。

通过使用命令行编译器选择项:-mt!,-mt! 和 -mm!,可以设置在 tiny、small 和 medium 模式中 DS 不等于 SS。

16.6 从 ASM 过程中调用 C 函数

想要从汇编语言模块中调用 C 函数,首先必须使 C 函数在汇编语言模块中可见。前面已简要地讨论过,可以使用 near 或 far 修饰符说明它为 EXTRN。例如已经编写了下面这个 C 函数:

```
long docalc(int *fact1,int fact2,int fact3);
```

为简单起见,假定 docalc 是一个 C 函数(相对于 Pascal 而言)。假设正使用 tiny、small 或 compact 存储模式,应在汇编语言模块中这样说明它:

```
EXTRN _docalc;near
```

同样,如果正在使用 medium、large 或 huge 存储模式,则将它说明为 _docalc:far.

使用 Turbo 汇编 2.0 的 C 语言定义符,这个说明将被写成:

```
EXTRN C docalc ; far
```

或

```
EXTRN C docalc;far
```

假设 docalc 被调用时带有三个参数:

- 命名为 xval 的实参的地址
- 存于命名为 imax 的实参的值
- 一个常数值 421(10 进制)

另外需要将结果保存在一个命名为 ans 的 32 位长的变量中,C 中等价的调用是

```
ans=docalc(&xval,imax,421);
```

必须首先在栈中压入 421,然后是 imax、xval 的地址,最后调用 docalc。当它返回时,必须清除栈空间中额外的 6 个字符,然后将结果传送到 ans 和 ans+2 中。

下面是相应的代码:

```
mov ax,421           ;Get 421,push onto stack
push ax
push imax            ;Get imax,push onto stack
lea ax,xval          ;Get lxval,push onto stack
push ax
call _docalc         ;Call docalc
add sp,6             ;Clean up stack
mov ans,ax           ;Move 32-bit result into ans
mov ans+2,dx         ;Including high-order word
```

Turbo 汇编 2.0 包含几个扩充,以使 C 和汇编语言的接口更为方便。其中的几个扩充可以自动地建立 C 风格的变量名,以使用 C 参数传递顺序将参数压入栈,并在调用函数以后清除栈。例如调用 docalc 函数可以写成:

```
EXTRN C docalc;near
mov     bx,421
lea     ax,xval
call    docalc C ax,imax,bx
mov     ans,ax
mov     ans+2,dx
```

参见 Turbo 汇编手册中有关这些新特性的完整描述。

如果 docalc 使用 Pascal 参数传递顺序,则必须将参数的顺序倒转过来,并且无须在返回时自己清除栈,因为函数本身已完成了这项工作,必须使用 Pascal 约定在汇编源文件中拼写 docalc(大写且没有前缀下划线)。

EXTRN 语句是:

```
EXTRN DOCALC;near
```

而调用 docalc 的代码是:

```

lea ax,xval          ;Get &xval,push onto stack
push ax
push imax             ;Get imax,push onto stack
mov ax,421            ;Get 421, push onto stack
push ax
call DOCALL           ;Call docalc
mov ans,ax             ;Move 32-bit result into ans
mov ans+2,dx           ;Including high-order word

```

Turbo 汇编器 2.0 也包含几个使 Pascal 风格的汇编语言接口简化的扩充,如自动建立 Pascal 风格的变量,以 Pascal 顺序在栈中压入参数。例如,Pascal 风格的 docalc 过程可写成:

```

EXTRN    PASCAL docalc,near
les      ax,xval
mov      bx,421
call     docalc PASCAL ax,imax,bx
mov      ans,ax
mov      ans+2,bx

```

16.7 伪变量、嵌入汇编和中断函数

如果希望做一些低层工作,但又不想涉及在建立独立汇编语言模块中所遇到的麻烦,Turbo C 有三种方法可供选择:伪变量、嵌入汇编和中断函数。下面说明了它们是如何进行工作的。

16.7.1 伪变量

系统中的 CPU(8088 或 80X86 处理器)有几个寄存器,每个寄存器有 16 位(2 个字节)长,其中大多数都有某些特殊用途,有几个也可以通用。参见“存储模式”一章中对这些寄存器的详细说明。

有时在低级程序设计中,可能想从 C 程序中直接访问这些寄存器,如:

- 在进行一个系统功能调用以前需要将值装入寄存器。
- 可能想看一下它们当前保存的值。

可以使用 INT(中断)指令调用 ROM 中的某个过程,但首先必须将一些必要的信息放入特定的寄存器中,如下例:

```

void readchar(unsigned char page,unsigned char * ch,unsigned char * attr)
{
    _AH = 8;          /* Service code,read char,attribute */
    _BH = page;        /* Specify which display page */
    geninterrupt(0x10); /* Call INT 10h services */
    *ch = _AL;          /* Get ASCII code of character read */
    *attr = _AH;        /* Get attribute of character read */
}

```

功能号和显示页号都被传送到 INT 10H,返回的值被存到 ch 和 attr 中。

通过伪变量, Turbo C 使得访问寄存器非常容易。伪变量就是一个对应于某个特定寄存器的标识符, 可以象使用 unsigned int 或 unsigned char 变量那样使用伪变量。

下面是安全使用伪变量的几个指导原则:

- 如果没有类型转换的话, 将一个简单变量赋给伪变量或相反, 不影响任何其它的寄存器。
- 将一个常量赋给伪变量也不会影响到其它寄存器中的数据, 给段寄存器(_CS、_DS、_SS、_ES)置值是一个例外, 这种操作需要使用 AX 寄存器。
- 简单的间接引用一个指针变量将破坏某些寄存器中的数据: 如 _BX、_SI 或 _DI, 也有可能包括 _ES。
- 如果必须设置某些寄存器(例如, 进行 ROM 调用时), 安全的方法是最后设置 _AX 寄存器, 因为它的值很可能无意中被其它语句改变。

表 16.3 完整的列出了可以使用的伪变量以及它们的类型、相对应的寄存器以及这些寄存器用途。

伪变量可被视作具有适当类型(unsigned int、unsigned char)的常规全局变量。然而, 因为它们表示 CPU 的内部寄存器, 而不是内存中的任意地址, 因此使用中有一些必要的限制和要求。

- 不能对伪变量使用取地址运算符 &, 因为伪变量没有地址。
- 由于大多数的 CPU 指令都用到相应的寄存器, 所以不能肯定置于伪变量中的值能保持到什么时候。

表 16.3 伪变量

伪变量	类型	寄存器	用途
_AX	unsigned int	AX	通用/累加器
_AL	unsigned char	AL	AX 的低字节
_AH	unsigned char	AH	AX 的高字节
_BX	unsigned int	BX	通用/索引
_BL	unsigned char	BL	BX 的低字节
_BH	unsigned char	BH	BX 的高字节
_CX	unsigned int	CX	通用/计数和循环
_CL	unsigned char	CL	CX 的低字节
_CH	unsigned char	CH	CX 的高字节
_DX	unsigned int	DX	通用/存放数据
_DL	unsigned char	DL	DX 的低字节
_DH	unsigned char	DH	DX 的高字节
_CS	unsigned int	CS	代码段地址
_DS	unsigned int	DS	数据段地址
_SS	unsigned int	SS	堆栈段地址
_ES	unsigned int	ES	附加段地址
_SP	unsigned int	SP	栈指针(偏移量相对于 SS)
_BP	unsigned int	BP	基指针(偏移量相对于 SS)
_DI	unsigned int	DI	用于寄存器变量

续表 16.3

伪变量	类型	寄存器	用途
<code>_SI</code>	<code>unsigned int</code>	<code>SI</code>	用于寄存器变量
<code>_FLAGS</code>	<code>unsigned int</code>	<code>flag</code>	处理机状态标识

这说明必须在马上就要使用它们时给它们置值,而在获得它们的值之后马上将值读出来,例如前面例子中的 `readchar`。对于通用寄存器(`AX`, `AH`, `AL` 等)更是这样,因为编译器自由地使用它们生成指令。从高层来讲, CPU 会以不希望的方式改变它们,例如在建立一个循环时或做移位操作时使用 `CX`,而使用 `DX` 保存乘积的高 16 位。

- 不能认为在调用一个函数以后,伪变量的值仍保持不变,例如,对下面的代码段:

```
_CX=18;
MyFun();
i=_CX;
```

在进行函数调用期间并不是所有的寄存器都被保存起来,所以无法确信 `i` 将得到所赋的值 18。在进行函数调用时,前后值不会被改变的寄存器只有: `_DS`、`BP`、`_SI` 和 `_DI`。

- 修改某些寄存器时必须十分小心,因为这可能导致不期望的灾难性的后果。例如,将值直接存于 `_CS`、`_DS`、`_SS`、`_SP` 或 `_BP` 中可能(并且几乎总是)导致程序运行不稳定,因为由 Turbo C 编译器生成的机器代码以各种方式使用这些寄存器。

16.7.2 嵌入汇编语言

编写独立的汇编语言过程并将它连接到自己的程序中,已经很容易了。Turbo C 也允许在 C 程序中直接编写汇编语言代码。这称作嵌入汇编。

为在 C 程序中使用嵌入汇编,使用 `-B` 编译器选择。如果没有使用而编译器遇到了嵌入汇编语句,它将给出一个警告信息,并以 `-B` 选择项重新开始编译。可以在源文件中使用 `#pragma inline`,它和使用 `-B` 选择项的作用一样。

必须准备 Turbo 汇编的一个备份,编译器生成一个汇编文件,然后调用 TASM 生成 OBJ 文件。

当然也必须熟悉 8086 的指令和结构。不编写完整的汇编语言过程时,仍旧需要知道使用的指令是怎样工作的,以及怎样使用它们。

在做完这些以后,所需要的只是使用关键字 `asm` 引入一个嵌入汇编指令。格式是

```
asmk opcode operands, or newline
```

这里

- `opcode` 是有效的 8086 指令(表 16.4 列出所有可用的指令)。
- `operands` 包含指令 `opcode` 可接受的操作数,可以引用 C 常量、变量和标号。
- `;or newine` 是一个分号或换行,用于标记 `asm` 语句的结束。

几条 `asm` 语句可置于同一行中,语句间用分号隔开,但 `asm` 语句不能延续到下一行中。如果需要使用多个 `asm` 语句,可以将它们放在花括号内:

```
asm {
    push ax; pop bx
    iret
}
```

与汇编语言编程不同,分号不再用于表示一个注释的开始(象 TASM 中那样的用法)。需要注释 asm 语句时,使用 C 风格的注释形式,例如:

```
asm mov ax,ds      /* 合法注释 */
asm {
    push ax; pop ax
    iret
}                  /* 合法注释 */
asm push ds        ; 非法注释!
```

语句的汇编语言部分直接输出到 .asm 文件,嵌入到 Turbo C 由 C 指令生成的汇编语言中,任何 C 符号被替换成等价的汇编语言。

嵌入汇编不是一个完整的汇编程序,所以许多错误不能马上检查出来。TASM 将捕获它可能遇到的任何错误。然而,TASM 不能正确地进行错误定位,尤其是当原始的 C 源文件的行号被丢失时更是这样。每条 asm 语句被当成为一条 C 语句。例如:

```
myfun()
{
    int i;
    int x;
    if(i>0)
        asm mov x,4
    else
        i=7;
}
```

这是合法的 if 语句。注意,在 mov x,4 指令后面没有分号,asm 语句是 C 中唯一的以换行作为结束的语句。

在一个函数内的任何一个汇编语句可用作一个可执行语句,在函数外的汇编语句可以是一个外部说明语句,放入 DATA 段中,在函数内的汇编语句放入 CODE 段中。

下面是函数 min 使用嵌入汇编的形式:

```
int min(int v1,int v1)
{
    asm {
        mov ax,v1
        cmp ax,v2
        jle minexit
        mov ax,v2
    }
    minexit:
    return (_AX);
}
```

这和前面使用 Turbo 汇编扩展特性的代码是很相似的。

可以将任何 8086 指令代码用作嵌入汇编语句。Turbo C 允许四类汇编指令。

- 一般指令——常规 8086 指令集；
- 串指令——特殊的串指令；
- 跳转指令——各种转移指令；
- 汇编指令——数据分配和定义。

注意：在嵌入汇编中编译程序允许各种各样的操作数，即使对汇编语言来说是错误的。编译程序没有限制操作数严格格式。

16.7.2.1 操作码(opcode)

注：标有 * * 的指令不能在使用浮点仿真库的过程中(TCC 选择项 -f)作为嵌入汇编语句的操作码。

当在嵌入汇编语句中使用 80186 指令助记符时，必须使用 -1 命令行选择项，这迫使在编译输出的汇编语言中加入适当的语句，以便 Turbo 汇编程序处理这些助记符。如果使用的是较早的汇编程序，则可能不支持这些助记符。

表 16.4 是可用于嵌入汇编的伪码助记符表：

表 16.4 操作码助记符

aaa	fdivrp	fpatan	lsl
aad	feni	fprem	mov
aam	ffree * *	fptan	mul
aas	fiadd	frndint	neg
adc	ficom	frstor	nop
add	ficomp	fsave	not
and	fidiv	fscale	or
bound	fidivr	fsqrt	out
call	fild	fst	pop
cbw	fimul	fstcw	popa
clc	finestp * *	fstenv	popf
cld	finit	fstop	push
cli	fist	fstsw	pusha
cmc	fistp	fsub	pushf
cmp	fisub	fsubp	rci
cwd	fisubr	fsubr	rcr
daa	fild	fsubrp	ret
das	fldl	ftst	rol
dec	fldcw	fwait	ror
div	fldenv	fxam	sahf
enter	fldl2e	fxch	sal
f2xml	fldl2t	fxtract	sar
fabs	fldlg2	fyl2x	sbb
fadd	fldln2	fyl2xp1	shl
faddp	fldpi	hlt	shr

续表 16.4

fild	fldz	idiv	smsw
fbstp	fmul	imul	stc
fchs	fmlp	in	std
fclex	fnclx	inc	sti
fcom	fndisi	int	sub
fcomp	fneni	into	test
fcompp	fninit	iret	verr
fdecstp * *	fnop	lahf	verw
fdisi	fnsave	lds	wait
fdiv	fnstcw	lea	xchg
fdivp	fnstenv	leave	xlat
fdivr	fnstsw	les	xor

16.7.2.2 串指令

除了列出的伪码以外,表 16.5 给出的串指令可以单独使用或带有 rep 前缀。

表 16.5 串指令

cmps	insw	movsb	outsw	stos
cmps	lods	movsw	scas	stosb
cmpsw	lodsb	outs	scasb	stosw
ins	lodsw	outsb	scasw	nsb
movs				

16.7.2.3 前缀

可以使用下列前缀:

lock rep repe repne repnz repz

16.7.2.4 跳转指令

跳转指令被特殊对待。标号不能被包含在指令本身中,必须跳到 C 标号处(在“使用转移指令和标号”中讨论)。表 16.6 给出允许的跳转指令:

表 16.6 转移指令

ja	jge	jnc	jnp	js
jae	jle	jne	jns	jz
jb	jle	jng	jnz	loop
jbe	jmp	jnge	jo	loope
jc	jna	jnl	jp	loopne
jcxz	jnae	jnl	jpe	loopnz
je	jnb	jno	jpo	loopz
jg	jnbe			

16.7.2.5 汇编指令

下面的汇编指令允许用在 Turbo C 的嵌入汇编语句中:

```
db dd dw extrn
```

16.7.2.6 对数据和函数嵌入汇编引用

可以在 asm 语句中使用 C 符号, Turbo C 自动将它们转换成适当的汇编语言操作数并在标识符前增加一个下划线。可以使用任何变量的标识符, 包括自动(局部)变量、寄存器变量和函数参数。

一般来讲, 可以将一个 C 符号使用在地址操作数可使用的任何地方。当然, 也可以使用寄存器变量, 寄存器变量是一个合法的操作数。

如果在分析一条嵌入汇编指令的操作数时, 汇编程序遇到一个标识符, 它将在 C 的符号表中搜索这个标识符。对于 8086 寄存器的名字, 不做这种搜索, 可以使用大写或小写的寄存器名字。

16.7.2.7 嵌入汇编和寄存器变量

嵌入汇编代码可以自由地将 SI 和 DI 用作暂存寄存器。如果在嵌入汇编代码中使用 SI 和 DI, 编译器将不把它们用于寄存器变量。

16.7.2.8 嵌入汇编、偏移量和长度控制

编程时, 不必关心局部变量地址的偏移量。简单地使用其名字就包含了正确的信息。然而, 有必要在汇编指令中包含适当的 WORD PTR、BYTE PTR 或其它的长度控制指令。如 DOWRD PTR 用于 LES 或间接远调用指令中。

16.7.2.9 使用 C 的结构成员

在一条嵌入汇编语句中, 可以用通常的方法引用 C 的结构成员(即变量成员)。在这种情况下可以存取一个变量成员的值。然而, 也可以直接以一个数值常量的形式引用成员(没有变量名)。在这种情况下, 常量等于(以字节计)这个成员相对于结构开始位置的偏移量。考虑下面的程序片段:

```
struct mystruct {
    int a_a;
    int a_b;
    int a_c;
} myA;
myfun()
{
    ...
    asm {
        mov ax, myA.a_b
        mov bx, [di].a_b
    }
    ...
}
```

程序说明了一个名为 mystruct 的结构类型, 带有三个成员 a_a, a_b, a_c, 也说明了一个 mystruct 类型的变量 myA, 第一条嵌入汇编语句将 myA.a_b 中的值转送到寄存器 AX 中。第二条将在地址[di]+offset(a_c)处的值移到寄存器 BX 中(它使用 DI 中的地址加上 a_c 相对于 mystruct 开始处的偏移量)。这些汇编语句产生下面的代码:

```
mov ax,DGROUP,myA+2
mov bx,[di+4]
```

如果把类型为 `mystruct` 的一个结构的地址装入到一个寄存器中(例如 `DI`),就可以使用成员名直接引用这个成员。成员名实际上可用在汇编语句的操作数允许数值常量使用的任何地方。

结构成员名前必须带有一个点号,以标明这是一个成员名而不是一个普通的 C 符号。成员名在输出中被替换成结构成员的数值偏移量(`a_c` 的数值偏移量是 4),但不保存任何类型信息,这样,成员可在汇编语句中用作编译期间的常量。

当然也有限制,如果有两个同样的成员名(分属不同的结构),在使用的嵌入汇编中,必须区分它们。在点和成员名之间插入结构的类型(在括号中),就象它是一种强制类型一样。例如

```
asm mov bx,[di].(struct tm)tm_hour
```

16.7.2.10 使用转移指令和标号

可以在嵌入汇编中使用任何条件和无条件跳转指令以及循环指令。它们只有用在一个函数内才是合法的。因为标号不能被定义在 `asm` 语句内,`jmp` 指令必须将 C 的 `goto` 标号作为跳转目标,而且不能直接生成远跳转指令。

在下面的代码中:

```
int n()
{
    a:          /* This is the goto label "a" */
    ...
    asm jmp a   /* goes to label "a" */
    ...
}
```

跳转到 C 中 `goto` 的标号 `a` 也允许间接跳转。为使用间接跳转,可以将一个寄存器名字用作转移指令的操作数。

16.7.3 中断函数

8086 保留存储器中最前面的 1024 字节作为 256 个远指针——称作中断向量——以便供称作中断处理程序的特殊系统过程使用。这些过程通过执行 8086 指令

```
int int #
```

来调用。这里,`int #` 在 `0~FFh` 之间。当执行这个指令时,CPU 保存代码段寄存器(`CS`)、指令指针寄存器(`IP`)和状态标志,禁止中断,然后做一个远跳转,跳到相应的中断向量指向的位置。例如,常用的一个中断是

```
int 21h
```

这个中断调用大多数的 DOS 过程,但有许多中断向量是未使用的。这表明可以编写自己的中断服务程序并将一个 `far` 指针放入一个未使用的中断向量中。

为在 Turbo C 中编写一个中断处理过程,必须将函数定义成类型 `interrupt`;更确切地说,它应该象这样:

```
void interrupt mvhandler(b0,di,si,ds,es,dx,cx,bx,ax,ip,cs,flags...);
```

上面的说明中,以参数形式传送了所有的寄存器,所以可不使用本章前面讲述的伪变量就能在代码中使用和修改它们。也可以向中断处理过程传递额外的参数(flag,...),这些参数应以适当的方式定义。

类型为 interrupt 的函数将自动地保存(除了 SI、DI 和 BP)AX 寄存器到 DX 寄存器以及 ES 和 DS 寄存器。这些寄存器在从中断处理过程退出时自动恢复。

中断处理过程可以在所有存储模式中使用浮点数。任何中断处理过程的代码,如果使用 80X87,则必须在进入中断处理程序时保存这个芯片的当前状态,而在退出时恢复这个芯片的状态。中断函数可以修改自己的参数。改变的参数在中断处理过程返回时将修改相应的寄存器。这在把中断处理过程用作一个用户子程序时将是有益的。这非常类似于 DOS 的 INT 21h 系统功能调用。注意,中断函数使用 IRET 指令(从中断中退出)退出。

那么,为什么需要编写自己的中断程序呢?这就要明白大多数内存驻留程序是怎样工作的。它们被安装成为中断处理过程,即一旦某种特定的或预期的活动发生(时钟走动、按下键盘等),这些过程将截取它们调用到自己的处理过程中,并采取相应的行动。在完成这些以后,它们再把控制传送到原来的中断处理过程中。

16.7.3.1 使用低级特征的实例

前面已经看到了几个在代码中使用这些不同的低级特征的例子,现在再看几个例子。先从一个中断处理过程开始,这个过程一旦被调用就响铃。

首先,编写一个函数。如下所示:

```
#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 char bcount = ax >> 8;
    /* Get thue current control port setting */
    bits = originalbits = inportb(0x61);
    for (i=0; i<=bcount; i++)
    {
        /* Turn off the speaker for a while */
        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);
}
```



```
}
```

另外再写一个函数来安装中断处理过程。传递中断处理过程的地址和中断号(0 到 255, 即 0x00 到 0xFF)到这个函数中。

```
void install(void interrupt (* faddr)(),int inum)
{
    setvect(inum,faddr);
}
```

最后,调用响铃函数测试它。

```
void testbeep(unsigned char bcount,int inum)
{
    _AH = bcount;
    geninterrupt(inum);
}
```

最后的 main 函数应是:

```
main(void)
{
    char ch;
    install(mybeep,10);
    testbeep(3,10);
    ch=getch();
}
```

若要截获原先的中断并在主程序完成以后再恢复它,可以用 `getvect()` 和 `setvect()` 来做这个工作。

在本下面章节将介绍以下高级编程特点的知识:

- 在 C 程序里使用汇编语言。一个好的 C 编译程序,如 Turbo C,能产生非常有效率的机器码。再加上 C 是一种功能特别强大和灵活的语言,使 C 程序员颇为偏爱(如果不是偏见)。为什么还用汇编语言呢? 因为使用它之后,能够更有力地发挥和更多用到计算机所有能力。一个结合使用两种语言的程序将更有表达能力。本章介绍了编写汇编语言例程以及使用 `inline`(行嵌入)汇编语句和分别编译模块的知识。
- 使用 Turbo C 的中断功能。Turbo C 支持范围极广的中断服务例程和设备,可以激发中断、俘获并处理它们。使用中断服务能帮助生成高质量的程序。读者还将学到如何使用软中断,如何俘获和处理与系统有关的中断事件。
- 使用 Turbo C 的程序优化功能。可以在大小或速度上对程序进行优化。读者将学到何时和如何优化程序,简短地游历一下编译程序优化方法的流程。

16.8 使用直接插入(`inline`)汇编语言

最先讲述直接插入汇编语言是因为这可能是混合使用 C 与汇编语言的最常用方式。Turbo C 以及其它优秀的编译程序生成的机器码效率相当高,所以大多数情况下只需调整

一下程序中的关键之处,就可以提高性能。

16.8.1 直接插入式汇编环境

直接插入式汇编语句能够直接放入 Turbo C。所以 C 语言程序是这些语句的外围环境。想要有效地正确地使用直接插入式汇编语句,必须具有下面两个知识。首先,必须知道如编译含有直接插入式汇编语句的 C 程序,这部分内容在这一小节讨论。第二,必须知道在直接插入式汇编语句里能做什么,不能做什么,这一点下一节讨论。

可以用命令行编译程序(TCC),指定-B 命令行可选项,来编译包含直接插入式汇编语句的 Turbo C 程序。这个可选项导致 Turbo C 执行 Turbo Assembler 或 Microsoft Macro Assembler,处理直接插入式汇编语句。

16.8.2 使用 asm 关键字

要在 C 程序里直接插入汇编语句,必须在语句的开头使用 asm 关键字。这里是可以使用 asm 语句的几种不同的方法:

```
asm assembly statement
asm assembly statement;
asm assembly statement;asm assembly statement
asm {
    assembly statement; assembly statement
    ...
    assembly statement
    assembly statement
    assembly statement
    ...
}
```

因为每个 asm 语句都作为一个单独的 C 语句,可以将一系列 asm 语句插在一个块里,如前面的例子所示的那样。

如果读者很熟悉 Turbo Assembler 或者 MASM,就很可能必须调整对分号的使用,因为在直接插入式汇编语句里虽然也使用分号,但在这里不标志着汇编注释的开始,而意味着行结束(end-of-line)。因此,分号可以用在同一个源程序行上,用于分开多个 asm 语句。在 asm 语句里当然也可以写注释,用通常的 C 注释语法(对 C 用 /* - */)。但是,千万注意,直接插入式汇编语句不能续行。

在程序里写入直接插入式汇编语句,必须对内存模式和指针大小非常敏感。尽管 Turbo C 有时要求注意指针大小(比如,在一个模式程序里用图形库例程),在混合使用 C 和汇编则必须时时对段的安排小心。程序 inline.c 显示了一个程序,它包含有直接插入式汇编,演示了存储模式相关性和 asm 关键字的用法。

程序 inline.c 该程序显示了直接插入式汇编在 large 模式 C 程序中的使用

```

1  #pragma inline
2
3  #include <stdlib.h>
4  #include <stdio.h>
5
6  void increment( int * arg )
7  {
8  /*
9      LARGE MODEL, far * auto generate
10     -----
11     stack frame: bp -> +0 oldbp
12                      +2 ofs ret adrs
13                      +4 seg ret adrs
14     +6 ofs of arg pointer
15     +8 seg of arg pointer
16     -----
17  */
18     asm {                /* Get FAR pointer to obj & deref */
19         push ds           /* Hang on to DS */
20         mov  ax,[bp+8]    /* Get seg adrs of arg */
21         mov  ds,ax
22         mov  si,[bp+6]    /* Get offset adrs of arg */
23         mov  ax,[si]      /* deref, pick up int */
24         inc  ax           /* Increment */
25         mov  [si],ax ,    /* and save it */
26         pop  ax
27         mov  ds,ax        /* Restore DS */
28     }
29 }
30
31 void main( void )
32 {
33     int j = 3;
34
35     printf( "Initial value was %d\n", j );
36     increment( &j );
37     printf( "New value is %d\n", j );
38 }

```

由于程序 inline.c 中的程序用 large 存储模式,直接插入式汇编必须反映这种情况。12 行和 13 行的注释指出,返回地址的段地和偏移量都要推进 large 存储模式的栈里,以便参数变量的地址能被在栈内移动两个字节(与 Small 存储模式相比)。

指针型参数也受到存储模式的影响。在程序 inline.c 里,包含着直接插入式汇编语句的

函数要求一个指向 `int` 的指针。由于程序以 `large` 存储模式编译,结果指针是四字节的,而不是二字节的。三行汇编代码(22-24 行)用来取这个指针,还需要另一行间接引用指针和取得数据变量(通过把它装到 `AX` 寄存器)。

正确编写汇编语句很可能是在混合使用 C 和汇编中最棘手的事情。这取决于能否写出与所使用的存储模式正好相配的汇编语言代码。

可以在直接插入式汇编语句里使用所有的通用 8086 指令,包括浮点指令。还可以使用 8086 指令集的扩充串指令,包括特殊的字节和字格式。还支持循环和跳转指令,包括 `lock` 和 `repxxx` 指令前缀。注意,跳转目标必须是一个 C 标号,这意味着,跳转目标不能写在某个块 `asm` 语句内部。

数据分配伪指令(`DB`、`DW`、`DQ`、`DT`)一样可以用在直接插入式汇编里,但其它的汇编伪指令(如 `SEGMENT` 和 `ASSUME`)不能用。如果在 `Compiler` 菜单上设置了 `CodeGeneration` 可选项,就可以使用 80186 和 80286 的操作码,但是任何情况下都不能在直接插入式汇编里使用 80386 指令。`Turbo C` 都不支持 80386 代码生成。可以通过编写单独编译的汇编模块,来摆脱这种限制,在单独编译模块里可以用任何想用的指令和伪指令。单独编译可被 C 调用的汇编例程的内容在下一节讲述。

细心的读者也许已经注意到,在程序 `callasm.c` 里,`DS` 寄存器先是被用到,后又被恢复。那些寄存器一般可以使用,对它们的使用又有些什么限制呢? C 函数调用环境期望保存 `BP`、`SP`、`CS`、`DS` 和 `SS` 寄存器。如果使用或者修改了它们,必须保证在退出直接插入式汇编之前将它们恢复到其原始值。这也就是程序 `callasm.c` 之所以在函数返回之前小心地恢复 `DS` 内容的原因。不过,对 `AX`、`BX`、`CX`、`DX`、`ES` 和标志位寄存器则可以随意使用,因为它们被当作“使用式”寄存器。

正如刚刚提到过的,某些寄存器在返回到 C 代码之前必须恢复。是不是有时候必须修改这些寄存器呢? 首先,必须考虑,有两个寄存器(`CS` 和 `SS` 寄存器)最好不要去动它。不要去修改这些寄存器(不是说间接的,比如用 `Call` 指令来修改 `CS`)。如果修改了它们很有可能后悔,直到成为一个汇编程序设计的专家。

你会经常发现必需使用 `DS` 和 `ES` 寄存器支持某些指令序列。使用 `ES` 寄存器没有任何问题,不需要进行保存和恢复。对 `DS` 寄存器在 C 程序使用要求很严格:要求保存和恢复 `DS` 寄存器。程序 `callasm.c` 里的程序用 `DS` 寄存器的原因是,23 行和 25 行的间接操作数指令要求的 `DS` 包含有被访问变量的段地址。在这种情况下,间接引用了 `int *` 指针,将它载入 `DS:SI`。然后,整数对象能通过表示法 `[SI]` 访问。

功能较强的汇编代码经常使用间接寻址来访问内存位置(正好与 C 程序相似!)。这意味着必须很好地控制段寄存器。另外,还必须知道控制哪些段寄存器。为了给读者提供便利,表 16.7 显示了不同的操作数格式和它们对应的寻址方式假定哪些段寄存器。

程序 `inline.c` 里的程序不能使读者明白,可以在写直接插入式汇编语句时,使用任何当前有效的 C 标识符的标号。这样,在很多情况下事情就简单多了。例如,在这个程序里,可以通过使用 `arg` 标号,写几个语句将指针载入到 `DS:SI`,因为这个名字 `arg` 的作用域是整个 `increment()` 函数。这几行可以这样写:

表 16.7 操作数格式与寻址方式的假定寄存器(8086 寄存器)

段寄存器	操作数格式	寻址方式
无	寄存器	寄存器操作数
无	数据	立即操作数(在指令中)
DS	位移	直接内存操作数
DS	标号	直接内存操作数
DS	[BX]	间接操作数
SS	[BP]	间接操作数
DS	[SI]	间接操作数
DS	[DI]	间接操作数
DS	[BX+位移]	基址相对操作数
SS	[BP+位移]	基址相对操作数
DS	[DI+位移]	直接变址操作数
DS	[SI+位移]	直接变址操作数
DS	[BX][SI]+位移	基址变址操作数
DS	[BX][DI]+位移	基址变址操作数
SS	[BP][SI]+位移	基址变址操作数
SS	[BP][DI]+位移	基址变址操作数

16.9 与汇编语言例程的接口

如果直接插入式汇编还不能满足需要生成合适的、高性能的代码,可以更进一步编写完整的、分别汇编的源模块。如果要用汇编语言实现的逻辑只是由于太长,不好轻松地放在一个C源文件的中间,就可以这样做。本节讲述如何在C里调用汇编模块和如何在汇编里调用C模块。

16.9.1 在C程序里调用汇编例程

编写独立汇编的C调用模块要求读者具有一点专门知识。必须大概了解有关C函数调用协议的所有知识。必须了解标准C栈帧、调用约定和段命令与排序约定。另外,还必须知道如何从汇编语言例程给C程序返回值,以及如果在汇编语言例程里调用C库函数。

因为C将函数参数传送到系统栈上,C调用环境的最重要部分之一就是标准的C栈帧。栈帧(stack frame)只是系统栈的一部分,C将参数和被调用函数的返回地址就放在栈里,这样汇编语言例程就是一个C函数。难点在于知道栈帧是如何组织的。

首先,要注意到第二个参数先被推入栈中。这说明函数参数从右到左扫描(这是调用协定所规定的),然后按这个顺序推进栈去。

其二,CALL指令将返回地址推到函数参数的上面。对一个large模式程序,其返回地址包含两个倒向的反字,与内存中其它far指针一样。也就是说,段地址先被推入栈中,紧接着是偏移地址,放在栈的顶上。

第三,在进入例程时,汇编代码必须立即将基址指针寄存器(BP)压入栈,并在就要返回之前将其弹出。这不是可做可不做的,如果这样做失败就会使栈一片混乱,汇编例程返回到它的调用者之后会出现有趣的变化。

在有独立汇编的模块时,编译和链接完整的程序需要一个额外步骤。你必须提供一个工程文件,在该文件里,指定汇编例程的目标代码与主 C 代码链接到一块。

有一个特殊的假定是,段寄存器指向由 C 编译程序设定的适当的代码、数据和堆栈段位置。最重要的段寄存器自然是 CS 和 DS 寄存器。表 16.9 显示了在 Turbo C 支持下的每种存储模式下这些寄存器的值。

表 16.8 进入一个汇编语言函数模块后的缺省寄存器值

存储模式	CS 指向	DS 指向
Tine	_TEXT	DGROUP
Small	_TEXT	DGROUP
Compact	_TEXT	DGROUP
Medium	filename _TEXT	DGROUP
Large	filename _TEXT	DGROUP
Huge	file _TEXT	DGROUP

表 16.8 显示出, tiny、small 和 compact 模式只支持一个名为 _TEXT 的代码段。medium、large 和 huge 模式支持多种代码段,名为 filename _TEXT,其中 filename _TEXT 是独立编译的源文件名。对除了 huge 之外的所有存储模式,在进入例程后 DS 寄存器指向 DGROUP 段组。DGROUP 包含 _DATA(初始的全局静态数据)和 _BSS(非初始的全局静态数据)段。要访问静态全局数据,无需修改 DS(除非将 DS 用于其它目的后再恢复之)。例如,如果程序说明了一个全局 action_flag,可以如下所示简单地引用它:

```

DOSSEG
.MODEL large
.DATA
action_flag dw 0          ; global action flag
.CODE
PUBLIC _myfunc
_myfuncPROC
...
mov ax,action_flag      ; get the flag
or  ax,8000h            ; turn on a bit
...
ret
_myfuncENDP
END

```

前边代码段里的 action_flag 变量将被链接程序合并进公用全局静态数据段。当然,如

果想使标志对其它模块可用,就必须在这里将它声称为 PUBLIC,在另一个汇编模块里声称 为 EXTRN,或者在一个 C 模块里声称 为 extern。相反,huge 模式程序没有 DGROUP。它对 每个单独编译的模块,都有各异的 filename DATA 段和 filename _BSS 段。它们都是 far 型 数据段。当一个 huge 模式程序里的汇编语言程序调用汇编语言例程时,汇编语言模块有它 自己的 far 数据段。必须用正确的值为 far 数据载入 DS,如下所示:

```

DOSSEG
.MODEL huge
.FARDATA
...
.CODE
PUBLIC _myfunc
_myfuncPROC
    push ds                ; save original DS
    ...
    mov ax,@fardata        ; get fardata seg adrs
    mov ds,ax              ; and load it
    ...
    pop ds                 ; restore DS_REQUIRED
    ret
_myfuncENDP
END

```

适用于直接插入式汇编语句的,关于系统寄存器的保护和使用的同样的规则也适用于 单独汇编的模块。程序 callasm.c 给出了 callasm.c 的源代码。这是一个简单的程序,其目的 只是调用 revstr()函数,将一个串在原变量倒置。之所以要看这个程序,当然是因为 revstr() 函数是一个汇编语言模块。糟糕的是,Callasm.c 使用的是 Small 存储模式,而 revstr()使用 的却是 large 存储模式。

callasm.c 一个在 C 中以混合存储模式调用汇编语言例程的程序

```

1   #include <stdio.h>
2   #include <stdlib.h>
3
4   char Message[] = "Message!";
5
6   int far revstr( char far * str ); /* Don't forget this */
7
8   void main( void )
9   {
10      revstr( (char far *)Message );
11      printf( "%s", Message );
12  }

```

在程序 callasm.c 中,唯一表明了使用着混合模式编程这一事实的现象是对 revstr()的

原型说明。该函数被显式说明为一个 far 函数,其参数是一个显式 far 指针,指向一个串。这样太简单了,这与用来声明 Turbo C 库例程的方法相同,这些例程总是 far 函数,需要 far 型参数。

现在请看 revstr() 自己是如何处理这一切的。这个外部 C 函数的源代码如程序列表 16.3 所示。

revstr.asm 一个 C 调用的汇编语言例程,用于将一个串在原处倒置。

```

1      ; REVERSE STRING function
2      ; Called from C/C++
3      ; C prototype:
4      ;          int far revstr( char far * str );
5      ;
6      ; Stack frame:
7      ;          old_bp          [bp+0]
8      ;          offset return    [bp+2]
9      ;          seg return       [bp+4]
10     ;          offset str       [bp+6]
11     ;          seg str          [bp+8]
12     ;
13     ; Returns:    int strlen( str )
14     ;
15     DOSSEG
16     .MODEL large
17     .CODE
18     PUBLIC _revstr
19     _revstr PROC
20         push    bp                ; set up standard stack frame
21         mov     bp,sp
22         push    ds                ; DS gets used, so save it
23         push    si                ; SI/DI may be register vars
24         push    di
25         mov     si,[bp+6]         ; get offset of string address
26         mov     di,si
27         mov     ax,[bp+8]         ; get segment of string address
28         mov     ds,ax
29         push    ds
30         pop     es                ; set up segment for scasb
31         cld                     ; forward direction for scasb
32         xor     ax,ax             ; null search arg
33         mov     cx,256           ; no more than 256 bytes
34     repne     scasb              ; scan the string for a null
35         dec     di                ; we passed it, back off by one

```



```

36          mov     cx,di
37          sub     cx,si          ; compute strlen
38          push    cx            ; save strlen
39          or      cx,cx          ; test for null string
40          jz      gohome
41          dec     di            ; point di to last byte of string
42  more:
43          cmp     si,di          ; if pointer regs equal,
44          jae     gohome        ; then we are done
45          mov     al,[si]
46          mov     bl,[di]
47          xchg    al,bl          ; swap the bytes
48          mov     [si],al
49          mov     [di],bl
50          inc     si            ; step up the string
51          dec     di            ; and down the string
52          jmp     short more
53  gohome:
54          pop     cx            ; retrieve strlen
55          pop     di            ; reset regs the way they were
56          pop     si
57          pop     ax
58          mov     ds,ax
59          pop     bp
60          mov     ax,cx          ; set return value = strlen
61          ret              ; and return to caller
62  _revstr     ENDP
63          END

```

处理带有不同存储模式的 C 与汇编语言混合编程的秘诀在于编写汇编语言代码处理标准栈帧的方法。基址指针(BP)寄存器用于访问参数变量。所需做的只是计算用于返回地址和指针的字节数。换句话说,就是如所愿地去编写汇编语言例程,让 C 源程序的源型声明来处理需要的指针大小。当然,两个源模块必须需要同样大小的指针。

revstr.asm 的样板程序还演示了以与周围的 C 环境兼容的方式控制段名和段次序的需用的方法。第 15 行的 DOSSEG 伪指令告诉汇编程序,链接时要使用 DOS 段次序,16 行的 MODEL 伪指令相当于自解释:调用 C 模块是以 large 存储模式编译的,所以在这里也指定了 large 模式。

17 行的 CODE 伪指令标志着程序代码段的开始,18 行的 PUBLIC 伪指令声明,例程名 _revstr 对其它模块(特别是对调用 C 模块是可访问的。19 行的 PROC 伪指令启动可由 C 调用的函数(可能是用汇编语言编程的)的代码。这些都是 Turbo Assembler 的简化的伪指令,它们生成与表 16.10 所示的段组名等价的段组名。本节的下一部分包含了更多关于简化的段伪指令和汇编语言扩展的信息(在对公用和外部变量名链接的讨论中)。

再看一看程序 `revstr.asm`, 注意所有对 `revstr` 的引用都加上一个前缀下划线; `_revstr` 是在汇编语言模块内部使用的正确形式, `revstr` 是在 C 模块里使用的形式。这地方没有什么神秘的。读者已经知道, Turbo C 和 Turbo C 在幕后使用前缀下划线来引用 C 标识符。现在, 假设你是在幕后。你仍然可以使用汇编例程的局部汇编程序标号, 不带前缀下划线, 但是必须记住使用下划线来引用 C 标识符。可以使用 Turbo Assembler 语言扩展名来避免此一需要。这一问题在本节的下一部分也有所讨论。

在对 `revstr()` 的讨论结束之前, 看一看第 60 行, 注意 AX 寄存器被用来给 C 调用程序返回一个值。这是不是一个随意的选择? 能不能以任何愿意的方式返回值? 回答是: 这不是随意的。每种 C 数据对象类型都有一个标准的返回位置。表 16.9 汇总了每种 C 数据类型的返回位置(即, 在返回(`ret`)指令之前, 返回数据必须放置的地方)。

表 16.9 从汇编语言例程到 C 的返回值的位位置

返回对象类型	返回值位置
unsigned char	AX
char	AX
enum	AX
unsigned short	AX
short	AX
unsigned int	AX
int	AX
unsigned long	DX, AX (DX = 高字组, AX = 低字组)
long	DX, AX
float	8087 TOS 暂存器 st(0)
double	8087 TOS 暂存器 st(0)
long double	8087 TOS 暂存器 st(0)
near *	AX
far *	DX, AX
struct(1-2 位元组)	AX
struct(4 位元组)	DX, AX
struct(3 位元组)	复制到记忆体位址, 并返回指标

在表 16.10 中, 指出一个数据类型必须通过将其值放在一个指定通用寄存器中来返回的表目是直截了当的。返回结构的方法稍有些复杂: 如果结构可在一个寄存器或寄存器对装下, 那就这样返回; 否则, 将它拷进一个静态存储区域并返回与存储模式相适应的指针。

不过, 返回浮点值的方法不那么清楚, 因为极少有汇编语言程序设计人员编写多点代码(曾经编写此类程序的人一般都栖身于某个软件开发公司——如 Borland)。浮点值必须在 80×87 TOS(栈顶)寄存器返回, 这意味着什么呢?

80×87 数值处理器本身就是一种计算机, 专用于高速浮点运算。因此, 数值处理器有它自己的栈。80×87 栈有一个固定数目的位置, 一共 8 个, 从 ST(0) 栈有一个固定数目的位置, 一共 8 个, 从 ST(0)(这是栈顶)到 ST(7)。80×87 的栈元素实际是特殊的寄存器, 每个宽为 80 位(即 10 字节)。使用 80×87(或者把它模拟成 Borland 的)的诀窍在于认识到 80×87 算术运算的操作数是, 在用 `fild` 指令载入时, 自动推入浮点栈的。然后对 ST(0) 和另一个操作数执行一个算术运算, 运算结果放在 ST(0) 中。因此, 所有要做的只是执行计算, 返回调用程序。看一看下面这个简单程序, 演示了如何做:

; C prototype;

```

;               double do _fadd(double,double);
;
; Stack frame;
;               old _bp           [bp+0]
;               ofrset return     [bp+2]
;               double arg1       [bp+4]
;               double arg2       [bp+12]
;
; Returns:      double
; Emulation: 8087 emulation by Turbo C emu. lib
;

EMUL
DOSSEG
.MODEL small
.CODE
PUBLIC _do _fadd

_do _fadd PROC

    push bp          ; set up staandard stack frame
    mov bp,sp
    fld qword ptr[bp+4]
    fadd qword ptr[bp+2] ; set up standard staack frame
    pop bp
    ret              ; that's all folks!

_do _fadd ENDP

END

```

前边的一段代码使用了 EMUL 伪指令,指出要使用 Turbo C 的浮点仿真库例程(如果存在 80×87 的话,这些例程会使用它)。除此之外,只需将第一个操作数装入 ST(0),fadd 另一个操作数,然后返回(不要忘了 C 栈帧协议)。

现在,可以向汇编语言例程传递参数,并给调用程序返回结果值。可以访问别的模块里存在的变量吗?比如全局静态数据。因为只能在 C 模块里,用 extern 关键字,来访问这样的变量,所以可能希望在汇编语言中也存在此类技术,可以做同样的事情。自然会有这样的技术,在下面讨论。

假设想用汇编例程求 10 个整数元素的和,并返回一个整数值。更进一步,假设希望用某种方法编写这个汇编例程,使得它既能访问 DGROUP (large 存储模式),又能访问一个 filename _DATA far 数据段(huge 模式)。

这里所提供的寻址方法依我们看来是最好的,因为它容易实现,并且总是允许使用 DS 访问该段,而不是必须用 ES 使笈式段寄存器(这是另一种常用的方法,但是要求使用段超越表示结束访问变量)。第一步是提供 C 驱动程序。这个程序名为 callasm2.c 的程序。

程序 Callasm2.c 如下:

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

```

2
3     extern int numbers[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
4     extern int sumar();
5
6     void main( void )
7     {
8         printf( "The sum of numbers is: %d\n", sumar() );
9     }
10

```

程序 Callasm2.c 中的程序既可以用 large 模式编译,又可以用 huge 存储模式编译。现在,我们需要一个汇编语言例程,它必须能访问 number 数组,不管使用的哪种存储模式。这样,就有两个难题需要解决,完全访问外部数据和取一个可用值放入段寄存器(DS),不管其它数据如何安排。

首先来考虑 number 数组在 large 或者 huge 存储模式下是如何存储的。在 large 模式下,所有全局数据都放在一个 DGROUP 里,在汇编例程获得控制的时候,DS 必须预置。因此,下面的伪指令可以用于 large 存储模式:

```

.MODEL large
.DATA
EXTRN numbers,WORD
...
.CODE
...
lea si,numbers          ; complete address now in DS:SI

```

不过,如果汇编例程计划与一 huge 模式 C 程序混用,这种方法可能就不行了,因为没有 DGROUP,.DATA 伪指令不适用。在千方百计寻求一个解法之后,还有可能跌入如下的陷阱:

```

.MODEL huge
.FARDATA
EXTRN numbers,WORD
...
.CODE
...
mov ax, @fardata
mov ds,ax
lea si,numbers          ; complete address NOT in DS:SI

```

这段代码有什么问题呢?除了这里说明的 far 数据段是属于汇编模块的数据段,而不是属于调用模块之外,其它没有任何问题。换句话说,这是寻址自己的 far 数据段的正确方法,但是你仍然不知道几种可能的数据段中哪个属于是你仍然不知道几种可能的数据段中哪个属于当前的调用程序。

如果用其它几种存储模式,结果也会出错。还是直接来看答案是怎样的吧!所需做的只是让编译程序和链接程序为你做保存段地址踪迹的工作。在到了寻址一个特定段的时候,通过使用 SEG 伪指令即可获得其地址。程序 getseg.asm 的汇编程序显示了如何做。

程序 getseg.asm 是一个访问 DATA 或 FADATA 里数组的汇编例程

```

1      ; SUM ARRAAY function
2      ; Called from C/C++
3      ; C prototype:
4      ;          int sumar( void );
5      ; Stack frame:
6      ;          old _bp          [bp+0]
7      ;          offset return    [bp+2]
8      ;          seg return       [bp+4]
9      ; Assumes... an array named numbers exists elsewhere
10     ;          which contains 10 elements
11     ; Returns... integer value of sum
12     ;
13     DOSSEG
14     .MODEL large
15     EXTRN _numbers,WORD
16     .CODE
17     PUBLIC C sumar
18     sumar      PROC C
19         push    ds                ; DS gets used, so save it
20         push    si                ; SI/DI may be register vars
21         push    di
22         mov     cx,10             ; move num items to ctr reg
23         lea     si,_numbers       ; get offset of array
24         mov     ax,SEG _numbers
25         mov     ds,ax            ; and locate its segment
26         xor     ax,ax            ; clear accumulator
27         xor     dx,dx            ; clear running total
28         cld                     ; clear direction means go forward
29     top,
30         lodsw
31         add     dx,ax
32         loop    top              ; this is all there is to it!
33         pop     di               ; reset regs the way they were
34         pop     si
35         pop     ds
36         mov     ax,dx            ; put return value in AX
37         ret                     ; and return to caller
38     sumar      ENDP

```

39

END ENDP

getseg.asm 里列出的程序与前面的那段代码的最大差别是：在任何段外声明 EXTRN 伪指令，所以它不与任一特定段相关联。正确的段地址可用 SEG 伪指令装入 DS 中，在第 24-25 行，取得了外部变量的段地址。因此，不管调用汇编例程用的是 large 存储模式还是 huge 存储模式，都可以借 DS 来寻址一个外部变量，不管该变量有可能放在哪儿。

注意，在程序里进行了宣称，以具有 large 存储模式，尽管它既可以在 large C 程序里用，也能在 huge 程序里使用。这一点（在涉及到汇编语言时，large 模式与 huge 模式是无差别的）很易接受。

16.9.2 在汇编例程中调用 C 函数

到现在，读者可能已经注意到，在汇编语言模块里缺少旧 MASM 段伪指令。原因是，与 Turbo Assembler 的简化的段伪指令相比，它们使用起来很困难。假设你能用老办法，完成得很艰若，但是何若呢？

如果你想使用简化的伪指令，就应该知道新的伪指令与 MASM 伪指令是如何对应的。表 16.11 将 Turbo Assembler 的简化伪指令与 MASM 的伪指令作了比较。

表 16.11 Turbo Assembler 的简化段伪指令与 MASM 段伪指令的比较

TAASM 伪指令	描述	MASM 伪指令
.CODE	代码段	_TEXT SEGMENT BYTE PUBLIC 'CODE'
.DATA	初始数据段	_DATA SEGMENT WORD PUBLIC
.DATA?	非初始数据段	_BSS SEGMENT WORD PUBLIC 'BSS'
FARDATA	初始 far 数据段	_file _DATA SEGMENT WORD
.FARDATA?	非初始数据段	_flie _BSS SEGMENT WORD PUBLIC 'BSS'

表 16.8 表 16.10 合起来，能给你几乎所有需要用以在汇编语言例程中控制段寻址的信息。有了这些知识，再加上你已经在程序 getseg.asm 里看到了如何在汇编里声明 EXTRN，以提供对其它模块变量的访问，现在就可以学习如何访问其它模块了。特别是你将学到如何在汇编语言例程里如何调用 C 库函数。

将要显示给读者的样板程序演示几件事。除了告诉你如何调用 C 库函数外，还能看到如何更充分地使用 Turbo Assembler 的简化段伪指令，和如何对 Turbo Assembler 使用与语言无关的扩展名。

这里提供的是一个名为 tokencpy() 的例程（再次请您注意汇编模块的 C 函数表示的使用），演示了如何在汇编语言模块里调用 C 库函数。这个例程执行对一个文本串非破坏性的扫描，用对 printf() 的调用显示从 stdout 读入的字串；将它拷进由调用程序提供的一个串。不破坏性的扫描就是在扫描一个串时（如 strtok()）不插入 null 终结字符。

样板汇编语言程序名为 callc.c 的程序驱动。Callc.c 没有什么特殊之处，除了对 tokencpy() 的显示说明将函数及其参数说明为 far 对象。这允许对 C 程序用 small 模式编译，尽管汇编语言模块使用 large 存储模式。

程序 callc.c tokencpy.asm 模块的 C 驱动程序

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  extern char far * tokencpy( char far * text, char far * token );
6
7  void main( void )
8  {
9      char Message[] = "Testing a line of text tokens today.";
10     char Output[16];
11     char far * text = Message;
12     char far * token = Output;
13
14     do {
15         text = tokencpy( text, token );
16     } while ( *text );
17
18     strcpy( Message, "a+b-c*d/e&f|g" );
19     text = Message;
20     do {
21         text = tokencpy( text, token );
22     } while ( *text );
23 }

```

tokencpy.asm 的源程序用来说明三件事情：在汇编语言调用 C 函数、语言独立扩充和在汇编中的参数说明的方法。

tokencpy.asm 一个非破坏性的扫描程序，建立一个单词串并在返回之前用 printf()

```

1  ; TOKEN COPY function
2  ; Called from C/C++
3  ; Calls printf()
4  ;
5  ; C prototype,
6  ;          char far * far tokencpy( char far * line,
7  ;                                  char far * token );
8  ;
9  ; Stack frame,
10 ;          old _bp      [bp+0]
11 ;          offset return [bp+2]
12 ;          seg return   [bp+4]
13 ;          offset line   [bp+6] (srcofs)
14 ;          seg line      [bp+8] (srcseg)
15 ;          offset token  [bp+10] (tokenofs)
16 ;          seg token     [bp+12] (tokenseg)

```

```

17 ;
18 ; Returns... char far * line (ptr to new location in source)
19 ;
20     DOSSEG
21     .MODEL large
22     EXTRN C printf,PROC
23 .DATA
24 fmtstr db "%s", 0ah, 0 ; notice newline notation
25 .CODE
26     PUBLIC C tokency
27 tokency PROC C srcofs,WORD, srcseg,WORD,\
28             tokenofs,WORD,tokenseg,WORD
29     push    ds ; DS gets used, so save it
30     push    si ; SI/DI may be register vars
31     push    di
32     mov     ax,srcseg
33     mov     ds,ax ; seg for source string
34     mov     si,srcofs ; offset for source string
35     mov     ax,tokenseg
36     mov     es,ax ; seg for token string
37     mov     di,tokenofs ; offset for token string
38 skipws:
39     mov     al,byte ptr [si] ; skip all blanks and tabs
40     cmp     al,20h
41     jne     short testtab
42     jmp     isws
43 testtab:
44     cmp     al,09h
45     jne     short copystr
46 isws:
47     inc     si
48     jmp     short skipws
49 copystr: ; now copy source frag to token
50     mov     byte ptr es:[di],al ; use seg override
51     or      al,al ; if null term just copied
52     jz      tokenout
53     call    isop ; if operator char
54     je      foundop ; at beginning of token
55     inc     si
56     inc     di
57     mov     al,byte ptr [si]
58     call    isop ; if operator char
59     je      gohome ; trailing token

```



```

60      cmp     al,20h           ; if more ws appeared
61      je      gohome
62      cmp     al,09h
63      je      gohome
64      jmp     short copystr
65  foundop:
66      inc     si
67      inc     di
68  gohome:
69      mov     al,0             ; null terminate token
70      mov     byte ptr es,[di],al
71  tokenout:
72      mov     srcofs,si        ; first update source offset
73      pop     di              ; reset regs the way they were
74      pop     si
75      pop     ds
76      lea     ax,fmtstr        ; compute adrs format string
77      call    printf C,ax,ds,tokenofs,tokenseg
78      mov     dx,arcseg        ; return seg adrs line
79      mov     ax,srcofs        ; return offset adrs line
80      ret                    ; and return to caller
81      tokencpy ENDP
82      ;
83  isop:                          ; set zero flag if char is an operator
84      cmp     al,'+'
85      je      endop
86      cmp     al,'-'
87      je      endop
88      cmp     al,'*'
89      je      endop
90      cmp     al,'/'
91      je      endop
92      cmp     al,'% '
93      je      endop
94      cmp     al,'~'
95      je      endop
96      cmp     al,'!'
97      je      endop
98      cmp     al,'='
99      je      endop
100     cmp     al,'&'
101     je      endop
102     cmp     al,'|'

```

```

103         je      endop
104         cmp     al, '^'
105         je      endop
106         cmp     al, '.'
107         je      endop
108 endop:
109         ret
110         END

```

可以看到,如果将程序 `tokencpy.asm` 的程序与前面的几个样板进行比较,有几个不同。第一个差别当然是使用了 `printf()` 库函数。`printf()` 函数在程序清单的 22 行被声明为 `EXTRN`,在 77 行调用该函数。

现在看一看 77 行的调用语句。第一眼看去,它似乎很象未用括号括起来的通常的 C 函数调用。这是因为跟在函数名后面的 C 字符是一个语言指定符(language specifier),它调用 Turbo Assembler 与语言独立的扩展部分。这个扩展部分允许你将参数指定在一个表内,次序与你在进行 C 函数调用时使用的相同。这个功能当然带来不少方便。

Turbo Assembler 里的语言扩展部分比这更进一步。如果再仔细看看程序 `tokencpy.asm` 里的程序,就会注意到其中没有下划线。因为下划线仍然由编译程序生成,为什么程序里一个都没有?

程序 `toiency.asm` 显示的更可读的源程序格式由在 `EXTRN`、`PUBLIC` 和 `PROC` 声明(22、26 和 27 行)的 C 语言扩充的指定激发。每次声明紧接在保留字之后都指定了 C 字符。在 `EXTRN` 声明里的 C 指定允许引用 `printf()`,无需下划线。在 `PUBLIC` 声明里的 C 指定允许对函数名 `tokencpy` 的调用,也无需下划线。而在 `PROC` 语句里的同样指定允许对函数参数的引用,无需下划线。

另外,注意 27 行的 `PROC` 语句,声明函数参数的次序与它们在该函数的调用中出现的次序相同。这里,对参数的声明允许你在 `PROC` 体里引用这些参数,而不是显式装入 BP 寄存器,并且不用计算对 BP 的相对偏移量,来访问参数变量。还注意到,也没有代码推入弹出 BP,以建立标准栈帧。在 `PROC` 语句里使用的 C 语言扩展指定符能使所有的东西都做好。

使用 Turbo Assembler 的简化段伪指令和语言独立扩展使得为 C 程序编写汇编语言例程很容易,甚至可以不费劲地执行以汇编编写的复杂的 BGI 图形。另一对样板程序演示了这一重要功能。

程序 `graph3.c` 的 C 程序是从前面一章提供的 `graph2.c` 程序借过来的,只是重写一下要执行 `gfx()` 汇编语言函数。`gfx()` 完成了所有的作图功能,C 例程只用等待键盘输入和终止图形方式。

程序 `graph3.c` BGI 图形汇编语言例程的 C 驱动程序

```

1  /*  GRAPH3.C  This program is a rewrite of GRAPH2.C, which
2      was presented in Chapter 8 */
3  #include <graphics.h>
4
5  #include <stdlib.h>

```

```

6   #include <stdio.h>
7   #include <conio.h>
8
9   int graphdrv = DETECT;
10  int graphmode;
11  int i, x, y;
12  int polypoints[] = { 10, 10,
13                        500, 10,
14                        500, 300,
15                        10, 10 };
16  char bgipath[] = "\\bc\\bgi";
17
18  extern void gfx( void );      /* Do it in assembly language */
19
20  int get_rand( int scale ) {
21      return random( scale );  /* random is inline function */
22  }
23
24  int main( void )
25  {
26      randomize();
27      gfx();
28      while( ! kbhit() );
29      closegraph();
30      return 0;
31  }

```

在程序 graph3.c 里,变量作用域被从 main() 内部(变量与 main() 均在 graph2.c)移到文件范围,因此这些变量被放到全局静态存储器里。这种方法大大简化了在汇编例程访问变量的过程。这种移动也提供一个机会,演示了 EXTRN 汇编声称在 large 模式的使用,如程序 fgx.asm 所示。

程序 gfx.asm 演示 C 语言扩展如何在汇编中使用 BGI 图形的程序

```

1   DOSSEG
2   .MODEL small
3   ,
4   EXTRN C initgraph:FAR
5   EXTRN C ellipse:FAR
6   EXTRN C cleardevice:FAR
7   EXTRN C drawpoly:FAR
8   EXTRN C line:FAR
9   EXTRN C bar3d:FAR
10  EXTRN C get_rand:PROC
11  ,

```

```

12      .DATA
13          EXTRN C graphdrv,WORD
14          EXTRN C graphmode,WORD
15          EXTRN C x,WORD
16          EXTRN C y,WORD
17          EXTRN C polypoints,WORD
18          EXTRN C bgipath,BYTE
19  adrsdrv    dd    0
20  adrsmode   dd    0
21  adrsbgi    dd    0
22  adrspoly   dd    0
23  ;
24      .CODE
25      PUBLIC C gfx
26  gfx        PROC C
27              push    si
28              push    di
29              push    ds
30  ;
31              push    ds
32              pop     es ; start building pointers in _DATA
33              mov     word ptr adrsdrv+2,es
34              mov     word ptr adrsmode+2,es
35              mov     word ptr adrsbgi+2,es
36              mov     word ptr adrspoly+2,es
37              lea     ax,graphdrv
38              mov     word ptr adrsdrv,ax
39              lea     ax,graphmode
40              mov     word ptr adrsmode,ax
41              lea     ax,bgipath
42              mov     word ptr adrsbgi,ax
43              lea     ax,polypoints
44              mov     word ptr adrspoly,ax
45  ;
46              call    initgraph C,adrsdrv,adrsmode,adrsbgi
47              mov     cx,100
48  ellipses:
49              call    get _rand C,640
50              mov     x,ax
51              call    get _rand C,480
52              mov     y,ax
53              call    ellipse C,x,y,0,360,x,y
54              loop    ellipses

```

```

55          call    cleardevice C
56          call    drawpoly C,4,adrspoly
57          call    line c,0,450,640,450
58          call    bar3d C,100,50,300,400,10,1
59          ;
60          pop     ds
61          pop     di
62          pop     si
63          ret
64  gfx      ENDP
65          END

```

你可能已注意到在程序 `gfx.asm` 里,汇编模块是以 `small` 存储模式汇编的(作为 C 驱动程序)。记住,所以图形库函数都只支持 `far` 调用和 `far` 数据。这就是为什么 31~44 行获得图形参数的 `far` 地址是必需的。这也是为什么 4~9 行的 `FAR` 指定符在声称外部库函数是必需的。

在这种混合模式编程必须的技术之外,简化的 *Turbo Assembler* 词法是如此清楚,几乎不用再费什么口舌。对图形函数的调用是显而易见的,它们的性能也是货真价实的,如所计划的一样。

16.10 使用中断功能

这一节引导读者熟悉 Turbo C 提供的使用 80×86 中断结构的便利。首先学习如何引发 30×86 和 DOS 中断,然后学习如何处理由系统其它部分引起的中断。

16.10.1 80×86 的中断结构

80×86 的中断既可能是由硬件引起的(如 I/O 设备),又可能是由软件(如 `int` 汇编指令)。一个中断发生后,CPU 保存其当前状态,包括下一条将要执行指令的地址、系统标志位和系统寄存器。

下一步将会发生什么取决于与中断相关的中断号(interrupt number)。对于硬件中断,一个支援芯片产生并提供中断号。对于软件中断,中断号是由 `int` 指令的操作数提供的。

现在, 80×86 CPU 将中断号用作中断向量(interrupt vector)表内的变址;一个中断向量其实就是一个 `far`(4 字节)指针。一个中断的指针包含了中断服务例程(ISR)的地址,将处理新条件。

正如你可能想象到的,中断变量表并不是放在内存的任何地方。它放在 RAM 开端的固定位置,从地址 0000:0000 开始。图 16.1 显示了中断向量表的相对位置,以及一台 IBMPC 或者兼容计算机 RAM 其它部分的组织。

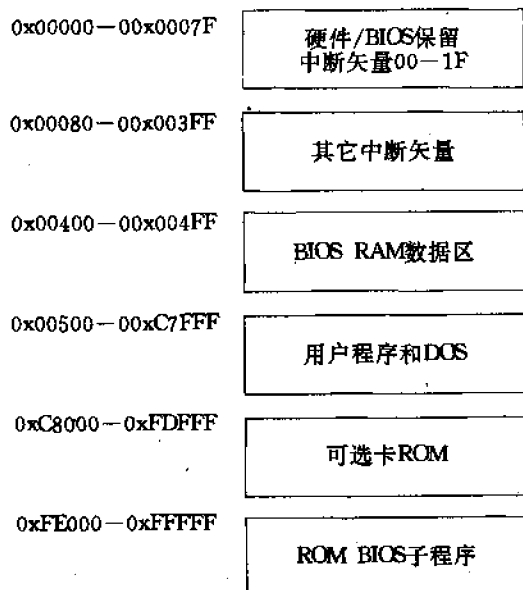


图 16.1 内存组织,包括 IBM PC 和兼容机的中断向量位置

由于中断向量表占据了 RAM 的头 1024 字节,所以最多只能有 256 个中断向量。中断向量根据其目的和用哪种 ISR 处理中断条件,被分成几组,如表 16.12 所示。

表 16.11 IBM 和兼容 ISA/EISA 及 Microchannel 计算机上 80×86 中断向量用法的总结

中断调用者	中断处理器	中断号(16 进制)
硬件	用户和系统处理程序	00—0F
用户, DOS	ROM BIOS	10—1F
用户	DOS	20—3F
DOS, DVC 驱动程序	BIOS, 设备驱动程序	40—5F
用户	(为用户保留)	60—66
用户	LIM EMS 驱动程序	67
硬件	设备驱动程序	70—77
BASIC	BASIC	80—F0
(未使用)	(未使用)	F1—FF

在表 16.11 中可以看到,多数中断由系统处理,DOS 和 BIOS 都能处理。在某些环境下,程序可以占领中断向量表的某个位置,为该中断向量提供 ISR。本章后面有一节“使用中断处理程序”,会介绍如何做到这一点。但是在此之前,还需要学会如何以一种可控制、可使用的方式来引发中断。

16.10.2 使用 Borland 的中断接口

Turbo C 均提供了两个库函数,以引发一般 80×86 中断,和两个函数以引发 DOS 中

断,这些函数是 `int86()`、`int86x()`、`intdos()` 和 `intdosx()`。

所有这四个中断引发函数必须使用 `union` 名 `REGS` (在 `dos.h` 里说明), 可以用它存储和读回系统寄存器的拷贝。这一点很重要, 因为系统寄存器必须用来联系参数 BIOS 和 DOS 内部的中断处理程序 (ISR)。`int86x()` 和 `intdosx()` 函数还通过 `REGPAACK` 结构 (也在 `dos.h` 里说明) 影响到 `DS` 和 `ES` 寄存器, 但这两个函数这里未加讨论。集中讨论 `int86()` 和 `intdos()` 函数 (包含了对 `DS` 和 `ES` 和修改的算法部分最好留给汇编语言去做, 这部分前边已经讲过)。

`REGS union` 有两个成员结构。`h` 结构包含了对所有字节寄存器 (`ah`、`al`、`bh`、`bl` 等等) 的 `unsigned char` 声明。`x` 结构成员包含了对所有字寄存器 (`ax`、`bx`、`cx` 及其它) 的 `unsignedint` 声明。注意, 寄存器变量名是以小写字母出现的。因此, 一个名为 `reg` 的 `REGS` 联合对象包含有一个成员, 名为 `reg.h.al`, 其中装了一份系统 `AL` 寄存器的拷贝; 也有一个名为 `reg.a.ax` 的成员, 其中装有系统 `AX` 寄存器的拷贝。最后, 还请注意, 给 `reg.x.ax` 赋值也就隐含着给 `reg.h.ah` 和 `reg.h.al` 赋了值。

调用一个 BIOS 中断的一般步骤包括: 首先, 往 `AH` 系统寄存器中装入一个功能码 (function code) (一个指示执行特殊操作的数字); 然后, 执行一个 `int` 指令, 指令指定该中断号作为其操作数。例如, 在下面的程序里, `0x13` 中断, 功能号 `0x19` 用于复位一个硬盘驱动器的磁头。完成这一操作的汇编语言序列如下:

```
...
mov ah,19h          ; function code for park heads
mov di,80           ; do it for drive C:
int 13h             ; get BIOS to do it
...
```

当然, 在使用 Turbo C 的 `int86()` 函数, 不用直接装入系统寄存器; 它们的值可以由 `int86()` 函数从 `REGS union` 中取得, 这个 `union` 中各成员的值必须在调用该函数前就设置好。也毋需直接调用中断, 因为 `int86()` 函数在设置好真实的系统寄存器后就会调用中断。程序 `fdpark.c`, 显示了如何使用 `int86()` 函数来复位计算机硬盘的磁头。

程序 `fdpark.c` 使用 `int86()` 函数复位所有系统硬盘驱动器磁头的程序。

```
1  /* FDPARK.C ----- Use INT86() function to park all fixed
2                                disk drives in the system
3  */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <dos.h>
8
9  void main( void )
10 {
11     union REGS reg; /* Register structs declared in dos.h */
12     unsigned char driveid = 0; /* Begin with C: */
```

```

13
14     while ( 1 ) {
15         reg.x.ax = 0x1900;
16         reg.h.dl = 0x80 | driveid;
17         int86( 0x13, &reg, &reg );
18         if ( reg.h.ah ) break; /* Error code means no more */
19         printf( "Parked heads for drive %c:...\n",
20             driveid++ + 67 );
21     }
22 }

```

注意 17 行,看 reg 对象在 int86()中断调用中是如何使用的。要一直记得,将 union 的地址传送给 int86()。头一个参数是寄存器变量 union 的地址,用作给 int86()的输入;函数将从头一个参数中取出置好的值。第二个参数是用作 86()输出的寄存器变量 union 的地址,结果寄存器值就放在那儿。

程序 fdpark.c 还演示了这一事实,即 BIOS 中断功能通常返回一个状态码,反映了操作的输出情况。在这种情况下,如果磁头定位操作成功,则 AH 寄存器将为 0,但如果该操作不成功,则 AH 寄存器里会装入一个错误码。可以用这个错误码去检测某个硬盘驱动器的缺乏。还有,不能直接存取系统寄存器 AH。应该如前所述,从 REGS union 里取得该寄存器的值。

磁头复位程序有一些限制,读者必须知道。尽管该程序几乎可以在最近生产的计算机上运行,但有些用户可能使用一台很旧的机器,硬盘驱动器是焊接在上面的。BIOS 中断 0x13,功能 0x19 只能作用于以下个人计算机:

- PC/AT 机, BIOS 日期为 11/15/86 以后的,或完全兼容的计算机
- PC/XT 机, BIOS 日期为 1/10/86 以后的,或完全兼容
- 所有 PS/2 机及其兼容机

如果你的计算机不属于上面所说的任何一类,则 fpark.c 不会损害任何东西,也不会做任何事情。

因为 Turbo C++ 本身已是足够完善的软件包了,所以你需要用到 int86()函数的机会将会很少。复位硬盘磁头正是这种机会之一。使用 intdos()函数的情况更罕见,但是这里仍然讨论一下,以求本书的完备性。

用户可以通过 80×86 中断号 0x21,加上许多相关的功能码来使用 DOS 功能。由于 DOS 调用经常是对它们自己的调用,Turbo C 提供了 intdos()函数,它已经假定了中断号是 0x21。坦白地说,在平常的代码中使用这个函数的理由并不多,因为 ANSI 标准库函数,再加上 Borland 支持的附加函数几乎已经覆盖了所有可能。正好对于程序 Calldos.c 中的程序,用 intdos()是需要的,它使用了 DOS 功能码 0x2A 以从 DOS 获取系统日期和星期几。

程序 calldos.c 一个使用 intdos()函数来得到系统日期和星期几的程序

```

1    /* CALLDOS.C --- Use INTDOS() function to get the system
2        date and weekday
3    */
4

```



```

5    #include <stdio.h>
6    #include <stdlib.h>
7    #include <dos.h>
8
9    char * months[] = {
10        "",
11        "January", "February", "March", "April",
12        "May", "June", "July", "August", "September", "October",
13        "November", "December"
14    };
15
16    char * days[] = {
17        "Sunday", "Monday", "Tuesday", "Wednesday",
18        "Thursday", "Friday", "Saturday"
19    };
20
21    void main( void )
22    {
23        union REGS reg;          /* Register structs declared in dos.h */
24
25        reg.x.ax = 0x2A00;        /* Set up call code for int. 0x21 */
26        intdos( &reg, &reg );    /* Get the date & weekday */
27        printf( "The current date is %s, %s, %d, %d.\n",
28            days[reg.h.al], months[reg.b.dh],
29            reg.h.dl, reg.x.cx );
30    }

```

intdos() 函数在使用和参数上几乎与 int86() 函数完全相同,除了其名字不同和无需加上中断号。如果读者想要关于 DOS 中断的所有细节,请查阅北京航空航天大学出版社出版的《DOS 系统调用详解》一书。

16.11 使用中断处理程序

一个中断处理程序就是一个 ISR。这足够简单。但是,如果想控制一个中断的处理,而不是让 BIOS 或者 DOS 去处理中断,该怎么办呢? Turbo C 通过提供 interrupt 指定符,给出了这次服务。你可以用 interrupt 指定符来标识一个你自己的 C 函数作为一个 ISR。在这一节,我们介绍这是什么意思和怎样做到。你也可以通过编写一个 ISR,在屏幕上显示一个实时时钟(尽管此刻程序做别的事情),来获得一点兴趣。

16.11.1 声明中断处理程序函数

一个中断处理程序函数的一般形式如下:

```
void interrupt newint60( int bp,int di,int si,int ds,
```

```

int es,int dx,int cx,int bx,
int ax,int ip,int cs,int flags)
{
    /* Do something in here */
}

```

在某中断引发你的 ISR 执行之时,系统已经将标志、CS 寄存器以及 IP(指令指针)寄存器推入系统栈中。Turbo C 所产生并将其加到你的 ISR 函数前面的代码,将要保存的寄存器推入栈中,次序如前面一段代码里所示(记住,Turbo C 从右到左扫描参数表)。不用为每个 ISR 函数声明所有系统寄存器。不过,你不能丢掉这里显示的或弄乱次序。如果需要寄存器 ax,你必须在参数表里在它之前声明所有寄存器。

ISR 函数体可以修改这些寄存器中任一个或者所有,通过将新值赋给寄存器参数,传送给函数。这会有将寄存器内容替换的效果。当寄存器被弹出栈时(也即 ISR 返回时),它就有了新值。通常的感觉——也就是对保护寄存器的限制。这在本章的前面讨论过——指示了哪些寄存器可以安全地修改。

除了在 ISR 里修改系统寄存器的值外,你还可以在 ISR 的函数体中做很多有用的事情。你可以更改变量(它自己的局部变量和全局静态数据,定义域只在 ISR 函数内),从一个通信端口读写数据,在屏幕上显示数据,甚至导致另一个中断(嵌套中断)。有一件事情是不能做的,如果你想要程序连续运行而且还不致损坏磁盘,除非你不一个 DOS 和汇编专家,否则千万不要试图不停地热行磁盘 I/O。

这种限制的理由是,中断的发生是不可预料的。比如,有可能在 DOS 正好进行其它文件操作中间发生。另外,DOS 是一种单任务的操作系统,这样做有可能导致系统损坏。

在 ISR 还可以做其它许多有用的事情,你也会发现这一点。首先,你必须知道建立 ISR 和修改中断向量和改换中断服务程序的基础。在安置自己的 ISR 时,要完成下面的操作:

- 保护好原中断向量。Turbo C 为此提供了 `getvect()` 函数。可以把这个 far 指针存在函数的参数中。
- 将自己的 ISR 函数的地址放到中断向量表中。`setvect()` 库函数可以完成这一任务。
- 在程序退出系统之前,恢复原来的中断向量。再次使用 `setvect()` 函数,用前面保存的原 ISR 地址为参数。

这看起来太简单了。你已经有了足够的知识去付诸行动,编写一个程序来演示这些技术。程序 `dummy.c` 较笨拙,做不了什么。它只是实现一个多任务的原型,通过译码 ax 的值,并用该值决定在 Turbo C 的哪个显示窗口显示一个放在 bx 中的整数值。在它设置好需要的寄存器值之后,main() 函数使用 `int86()` 产生 0x60 中断,0x60 是一个用户中断,所以使用是合法的。参考 `dummy.c`。

程序 `dummy.c` 一个很笨拙的函数,用用户中断 0x60 实现了多任务的原型

```

1  #include <stdio.h>
2  #include <dos.h>
3  #include <conio.h>
4
5  void interrupt (far *oldint60)( void );

```

```

6
7 void interrupt newint60( int bp, int di, int si, int ds,
8                          int es, int dx, int cx, int bx, int ax)
9 {
10     if ( ( ax >> 8 ) ) {
11         window ( 20, 12, 26, 20 );
12         delay( 1 );
13         cprintf( " %d\r\n", bx );
14     }
15     else {
16         window ( 10, 12, 16, 20 );
17         delay( 1 );
18         cprintf( " %d\r\n", bx );
19     }
20 }
21
22
23 void main( void )
24 {
25     int i;
26     union REGS reg;
27
28     oldint60 = getvect( 0x60 );
29     setvect( 0x60, newint60 );
30
31     clrscr();
32     for ( i = 0; i < 4096; ++i ) {
33         reg.x.ax = 0x0000;
34         reg.x.bx = i;
35         int86( 0x60, &reg, &reg );
36         reg.x.ax = 0x001 | i;
37         reg.x.bx = i;
38         int86( 0x60, &reg, &reg );
39     }
40
41     setvect( 0x60, oldint60 );
42     window ( 1, 1, 80, 25 );
43 }

```

注意,在程序 dummy.c 中,ax 和 bx 寄存器变量都用上了,所以参数表中所表在它们之前的寄存器变量都声明了。不过,此后的寄存器均不需要,所以没有声明。

程序 dummy.c 中的程序遵守了所有的规矩,而且做得更好。它在第 5 行声明了一个函数指针,原来的中断向量将保存在这个指针里(注意返回类型为 void,还有一个 interrupt 指

定值)。28 行取出原来的中断值,将它保存到函数指针里,第 29 行设置了新向量,使用 ISR 函数的地址。

32~39 行的循环调用了 0x60 ISR 函数,交替指定“左窗口”(ax==0 时)和“右窗口”(ax==1 时)。在每种情况下,循环计数器的值都放在 bx 寄存器变量里。结果是很有趣的,两个窗口交替显示整数,直到 4096。

在这个游戏结束之后,在 41 行原来的 0x60 向量被恢复。因为中断 0x60 是为象我们这样的用户保留的,系统的 ISR 可能是一个笨重的程序,但它没有了之后,程序怎样为中断提供服务呢?恢复中断向量的任何失败都给系统增加了故障,最好是老老实实地去做。

16.11.2 实现一个时钟中断处理程序

这一节介绍的 ISR,截取系统时钟中断,并使用该中断在屏幕的左上角显示一个实时时钟。在建立好 ISR 之后,程序的其它部分便做自己的事;同时,时钟连续不断地更新。

计时器中断(中断号 0x08)是一种由硬件产生的中断。在系统间隔计数器停止工作后,它激发中断请求线路(一种物理电子线路),该线路由 0x08 中断向量选择。更有趣的是,0x08 中断的 BIOS IS 调用另一个软件中断,即向量 0x1c。

中断 0x1c 的目的是允许用户程序截取它并用它做一些有用的事情。BIOS ISR 一般都不起作用,因为它只做一些如在磁盘驱动器运行情况下关闭之的操作。截取中断 0x08 也是可能的,但在这里不讨论。

图 16.2 显示了一个计时器滴嗒发生时,指令执行的流程。用户程序被临时挂起,中断 0x08 ISR 被执行;它轮流调用中断 0x1c。

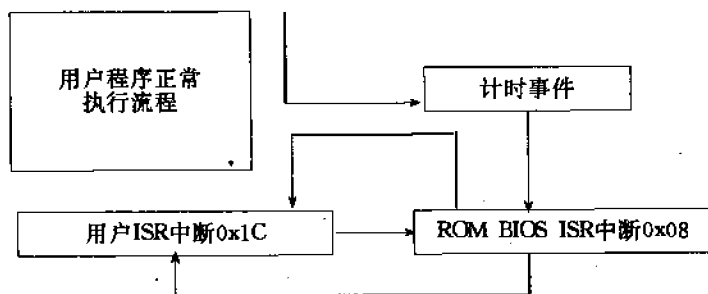


图 16.2 计时器中断发生时程序执行流程

在实时时钟 ISR 完成后,执行流程通过 0x08 ISR 转回去,最后返回到用户程序,在哪里挂起就回到哪里。跟以前一样,用户程序紧接着继续这行。当然,最后影响是,出现了两个程序在同一时间运行:一个时钟程序,一个用户程序。处理这个的程序是 ticker.c,如程序 Ticker.c 所示。

程序 ticker.c 一个 ISR 例程截取计时器中断,并在屏幕上显示一个实时时钟

```

1  /* -----
2      TICKER.C contains an interrupt service
3      routine (ISR). BE SURE that the test -
4      stack-overflow Options|Compiler|Code Generation
  
```

```

5      turned OFF.
6      ----- */
7      #pragma inline
8
9      #include <stdlib.h>
10     #include <stdio.h>
11     #include <dos.h>
12     #include <conio.h>
13
14     void interrupt far ( * oldint1c )( void );
15
16     unsigned char hour, min, sec;
17     unsigned sc0_base = 0xb800; /* base adrs for cga/ega/vga */
18     int todx = 1, tody = 1;
19     char tstring[9];
20
21     void interrupt newint1c( void )
22     {
23         static unsigned int count = 18;
24
25         if ( --count == 0 ) {
26             count = 19;
27             asm { /* Get time function */
28                 sti
29                 mov     ah,02h
30                 int     1ah /* Invoke BIOS BCD time */
31                 mov     hour,ch
32                 mov     min,cl
33                 mov     sec,dh
34             }
35             tstring[0] = ( hour >> 4 ) + '0'; /* Unpack BCD time */
36             tstring[1] = ( hour & 0x0F ) + '0';
37             tstring[2] = ',';
38             tstring[3] = ( min >> 4 ) + '0';
39             tstring[4] = ( min & 0x0F ) + '0';
40             tstring[5] = ',';
41             tstring[6] = ( sec >> 4 ) + '0';
42             tstring[7] = ( sec & 0x0F ) + '0';
43             asm { /* Get it on the screen — quickly */
44                 xor     dx,dx
45                 mov     ax,tody
46                 dec     ax /* Adjust to 0—origin adrs */
47                 mov     si,160

```

```

48      mul    si           /* Line offset in AX    */
49      mov    cx,ax        /* Save line offset    */
50      mov    ax,todx
51      dec    ax
52      shl    ax,1         /* Byte offset        */
53      add    cx,ax        /* Total offset CX     */
54      mov    di,cx
55      cli
56      push   es
57      mov    es,sc0_base /* ES points to output RAM */
58      mov    cx,8         /* 8 bytes out        */
59      mov    si,0
60      }
61      show_tick:
62      asm {
63          mov    bl,tstring[si] /* Pick up display byte */
64          mov    bh,70h        /* Normal video attribute */
65          mov    ax,bx         /* Recover bytes    */
66          stosw                /* Output, di auto incr */
67          inc    si
68          loop   show_tick
69      }
70      pop    ax
71      mov    es,ax
72      sti
73      }
74      }
75
76      void start_ticker( void )
77      {
78          oldintlc = getvect( 0x1c );
79          setvect( 0x1c, newintlc );
80      }
81
82      void stop_ticker( void )
83      {
84          setvect( 0x1c, oldintlc );
85      }
86
87      void main( void )
88      {
89          char ch;
90          int quit = 0, i = 1, incr = 1;

```

```
91
92     clrscr();
93     gotoxy( 24, 12 );
94     puts("Strike Esc to stop the demo. ");
95     start_ticker();
96     window( 12, 14, 68, 24 );
97     while ( ! quit ) {
98         if ( kbhit() ) {
99             if ( 0 == ( ch = getch() ) ) ch = getch();
100             if ( ch == 27 ) ++quit;
101         }
102         gotoxy ( i * 5, i );
103         cprintf( " %s", "Hello" );
104         i += incr;
105         if ( i == 10 ) { incr = -1; clrscr(); }
106         if ( i == 1 ) { incr = 1; clrscr(); }
107     }
108     stop_ticker();
109     window( 1, 1, 80, 25 );
110     clrscr();
111 }
```

最重要的事情之一是,为计时器中断编写 ISR 就是在 ISR 运行时不再有计时器中断发生。因此,ISR 必须在下一个计时器中断来到之前完成,或者系统时钟必须更后拖些。总之,如果你把它砍得太短,就将不会给程序留下什么时间去完成自己的操作。考虑所有的因素,可以知道:ISR 运行得越快就越好。因此,多数 0x1c ISR 函数的编码都使用直接插入式汇编。

第四部分

库函数和全局变量参考

Turbo C 标准库函数
全局变量

第十七章

Turbo C 标准库函数

本章按字母顺序列出 Turbo C 所有的函数。下面的函数样板说明本章和下面一章中要说明库函数和全局变量的格式。

■ 函数名 函数功能描述 ■

- 用 法** `#include <header.h>`
 这部分列出包含函数 `function` 的原型或函数中用到的常量、枚举类型定义等的头文件。
`function(modifier parameter[,...]);`
 这部分给出函数 `function` 的用法说明;[,...]表示括号内的参数及其描述符是可选的。
- 原 型** 在 `header.h`
 这部分列出包含 `function` 原型的头文件。若函数的原型包含在不只一个头文件中,则列出所有头文件,用户可以选用其中最合适的头文件。
- 说 明** 这一部分描述函数做的功能,它所需的参数,使用 `function` 的细节以及相关函数的列表。
- 返 回 值** 如果有返回值的话,在此列出函数 `function` 的返回值。若 `function` 已对全局变量 `errno` 进行设置,则其值也将列出(参见 DOS 参考手册中对 `errno` 的解释)。
- 可移植性** 在此列出 `function` 函数适用的系统和语言,它包括 UNIX,IBM PC 及其兼容机和 ANSI C 标准。
- 参 见** 在此列出 `function` 的相关函数。若函数名含有省略号(`funcname...`, `funcname`, `func...name`)说明应参考某一族函数。(例如 `exec...` 指的是一族 `exec` 函数: `execl`, `execle`, `execlp`, `execlpe`, `execv`, `execve`, `execvp` 和 `execvpe`)。

■ abort 异常终止—进程 ■

- 用 法** `#include <stdlib.h>`
`void abort(void);`
- 原 型** 在 `stdlib.h`, `process.h`
- 说 明** 本函数将终止信息("Abnormal program termination")写入 `stderr` 中,并通过调用带有终止码 3 的 `exit` 来终止程序。向 `stderr` 输出的信息一般出现在屏幕上。

返回值 abort 将把终止码 3 返回给父进程或 DOS。

可移植性 适用于 UNIX 系统, 在 ANSI C 中也有定义。

参 见 assert, atexit, exit, _exit, raise, signal, spawn...

示 例

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    printf("Calling abort()\n");
    abort();
    return 0; /* This is never reached */
}
```

■ abs 返回整数的绝对值 ■

用 法

对于实数	对于复数
#include <math.h>	#include <complex.h>
int abs(int x);	double abs(complex x);

原 型 在

实数版	复数版
math.h, stdlib.h,	complex.h

说 明 本函数返回整数参数 x 的绝对值, 如果先包含 stdlib.h 头文件然后再调用 abs, abs 将作为宏。如果要使 abs 当作函数使用而不是宏, 请在

```
#include <stdlib.h>
```

之后使用

```
#undef abs.
```

返 回 值 对于实数的 abs 返回 0~32767 间的一个整数(参数 -32768 除外, 它返回 -32768)。复数的 abs 返回一个双精度数。

可移植性 实数的 abs 适用于 UNIX 系统, 并且在 ANSI C 中也有定义; 复数的 abs 只适用于 Turbo C, 不可移植。

参 见 cabs, complex, fabs, labs

示 例

```
#include <stdio.h> #include <math.h>
int main(void)
{
    int number = -1234;
    printf("number: %d absolute value: %d\n", number, abs(number));
    return 0;
}
```

■ absread 读磁盘的绝对扇区 ■

用 法 #include <dos.h>

```
int absread(int drive, int nsects, int lsect, void *buffer);
```

原 型 在 dos.h

说明 `absread` 读入磁盘上指定扇区内容。不管扇区内容是磁盘的逻辑结构、文件、FAT 还是目录数据,函数 `absread` 都将进行读操作。

`absread` 通过 DOS 中断 0x25 读取指定磁盘扇区内容。

`drive`=要读的磁盘驱动器号(如果为 0,则为 A 驱动器,如果为 1,则为 B 驱动器,依此类推)

`nsects`=要读的扇区数

`lsect`=要读的起始逻辑扇区号

`buffer`=要读数据的内存存放起始地址

读入的扇区数最多为 64K,并且受 `buffer` 大小的限制。

返回值 如果调用成功,`absread` 返回 0。

如果调用出错,返回 -1,并将 `errno` 设置为调用后返回的 AX 寄存器的值。

可移植性 适用于 DOS。

参见 `abswrite`, `biosdisk`

示例

```
#include <stdio.h>
#include <math.h>
int main(void)
{
    int number = -1234;
    printf("number: %d absolute value: %d\n", number, abs(number));
    return 0;
}
```

■ `abswrite` 写磁盘绝对扇区

用法 `#include <dos.h>`

```
int abswrite(int drive, int nsects, int lsect, void buffer);
```

原型在 `dos.h`

说明 本函数将指定内容写入磁盘上的指定扇区,即使写入的地方是磁盘的逻辑结构、文件、FAT 表和目录结构所在的扇区,也照样进行。

注意:如果使用不当,`abswrite` 可能会覆盖文件、目录和 FAT 表中的内容。

`abswrite` 通过调用 DOS 中断 0x26 将 `buffer` 中的内容写到指定的磁盘扇区。

`drive`=要写的磁盘驱动器号(如果为 0,则为 A 驱动器,如果为 1,则为 B 驱动器,依此类推)

`nsects`=要写的扇区数

`lsect`=起始逻辑扇区号

`buffer`=要写入数据的内存起始地址

要写入的磁盘扇区限制在 64K 以内,并且受到 `buffer` 的大小限制。

返回值 如果调用成功,`abswrite` 返回 0。

如果出错,返回 -1,并将 `errno` 设置为调用后返回的 AX 寄存器的值。

可移植性 只适用于 DOS。

参 见 `absread, biosdisk`

示 例 `#include <stdio.h>`
`#include <math.h>`
`int main(void)`
`{`
`int number = -1234;`
`printf("number: %d absolute value: %d\n", number, abs(number));`
`return 0;`
`}`

■ access 确定文件的存取权限 ■

用 法 `#include <io.h>`
`int access(const char * filename, int amode);`

原 型 在 `io.h`

说 明 `access` 检查 `filename` 所指定的文件是否存在以及它的可读、可写和可执行性。包含在 `amode` 中的模式值如下：

- 06 检查读/写允许
- 04 检查读允许
- 02 检查写允许
- 01 执行(被忽略)
- 00 检查文件是否存在

注意：在 DOS 中，所有的文件均可读(`amode=04`)，所以 00 和 04 的结果是相同的。同理，06 和 02 也是等价的，因为在 DOS 中，写访问隐含了读访问。

如果 `filename` 指向一目录，`access` 将返回该目录是否存在。

返 回 值 如果要求确定的存取权限是允许的，`access` 返回 0，否则返回 -1，并将全局变量 `errno` 置为下列值之一：

`ENOENT` 路径名或文件名没找到 `EACCES` 权限不对

可移植性 适用于 UNIX 系统。

参 见 `chmod, fstat, stat`

示 例 `#include <stdio.h>`
`#include <io.h>`
`int file_exists(char * filename);`
`int main(void)`
`{`
`printf("Does NOTEXIST.FIL exist: %s\n",`
`file_exists("NOTEXIST.FIL") ? "YES" : "NO");`
`return 0;`
`}`
`int file_exists(char * filename)`
`{`

```

    return (access(filename, 0) == 0);
}

```

■ acos 计算反余弦值

- 用 法** 对于实数 对于复数
`#include <math.h>` `#include <complex.h>`
`double acos(double x);` `complex acos(complex x);`
- 原 型 在** 对于实数 对于复数
`math.h` `complex.h`
- 说 明** `acos` 将计算输入值的反余弦值。传给 `acos` 的实参数必须在 -1 和 1 之间, 否则 `acos` 返回 NAN 并置全局变量 `errno` 为 EDOM(域错误)。
 复数的反余弦定义为:

$$\operatorname{acos}(z) = -i \log(z + i \operatorname{sgrt}(1 - z^2))$$
- 返 回 值** 输入 -1 到 1 之间的一个实参数, `acos` 返回一范围在 0 到 pi 之间的值。该函数的错误处理程序可以通过 `matherr` 函数修改。
- 可移植性** 实数类型的 `acos` 函数适用于 UNIX 系统并且在 ANSI C 中也有定义。复数类型的 `acos` 只适用于 Turbo C, 无移植性。
- 参 见** `asin`, `atan`, `atan2`, `complex`, `cos`, `matherr`, `sin`, `tan`
- 示 例**
- ```

#include <stdio.h>
#include <math.h>
int main(void)
{
 double result;
 double x = 0.5;
 result = acos(x);
 printf("The arc cosine of %lf is %lf\n", x, result);
 return 0;
}

```

## ■ allocmem 分配 DOS 内存

- 用 法** `#include <dos.h>`  
`int allocmem(unsigned size, unsigned segp);`
- 原 型 在** `dos.h`
- 说 明** 本函数使用 DOS 系统调用的 `0x48` 来分配自由内存, 并返回分配内存的段地址。  
`size` 是要求分配内存的大小(以节为单位计算)。`segp` 是一个指针, 它指向新分配内存的段地址。如果没有足够的内存可供分配, 将不对 `segp` 进行赋值。  
 注意: `allocmem` 和 `malloc` 不能同时使用。
- 返 回 值** 如果调用成功返回 -1, 若出错, 将返回一整数(以字节为单位的最大可用内存的大小)。  
 出错返回时将置 `doserrno` 和全局变量 `errno` 为:

ENOMEM 无足够的内存

可移植性 仅适用于 DOS.

参 见 coreleft, freemem, malloc, setblock

示 例

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

int main(void)
{
 unsigned int size, segp;
 int stat;
 size = 64; /* (64 x 16) = 1024 bytes */
 stat = allocmem(size, &segp);
 if (stat == -1)
 printf("Allocated memory at segment: %x\n", segp);
 else
 printf("Failed: maximum number of paragraphs available is %u\n", stat);
 return 0;
}
```

## ■ arc 画圆弧

用 法 #include<graphics.h>  
void far arc(int x, int y, int stangle, int endangle, int radius);

原 型 在 graphic.h

说 明 arc 以(x,y)为圆心, radius 为半径, 从起始角 stangle 到终止角 endangle 以当前颜色画一圆弧。如果 stangle=0 且 endangle=360, 则 arc 将画一个整圆。  
arc 函数使用的角度是逆时针方向的, 0 度在 x 轴的正向, 90 度在 Y 轴的正向。  
说明: 线型参数对弧、圆、椭圆或圆饼图形无影响, 这些图形只受 thickness 参数的限制。

注意: 如果用户使用的是 CGA 高分辨率模式或是单色图形适配器, 本书中有关图形函数的示例程序可能会得不到预期的结果。如果系统采用了 CGA 高分辨率模式或单显, 则应该直接把 1 传给用以改变填充图形或画线颜色的函数(例如 setcolor, setfillstyle, setlinestyle), 而不能使用 graphics.h 中定义的颜色符号常量。

返 回 值 无返回值。

可移植性 此函数为 Turbo C 所特有。只能工作在配备图形适配器的 IBM PC 及兼容机上。

参 见 circle, ellipse, fillellipse, getarccoords, getaspectratio, graphresult, pieslice, sector

示 例

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
```



```

int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int midx, midy;
 int stangle = 45, endangle = 135;
 int radius = 100;
 /* initialize graphics and local
 variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 /* an error occurred */
 if (errorcode != grOk)
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
 }
 midx = getmaxx() / 2;
 midy = getmaxy() / 2;
 setcolor(getmaxcolor());
 /* draw arc */
 arc(midx, midy, stangle, endangle, radius);
 /* clean up */
 getch(); closegraph();
 return 0;
}

```

## ■ asctime 转换日期和时间为对应的 ASCII 码

用 法 #include <time.h>

char \*asctime(const struct tm \*tblock)

原 型 在 time.h

说 明 asctime 把存储在 \*tblock 里的时间转换为 ctime 字符中对应的 ASCII 字符串，例如：

Sun Sep 16 01:03:52 1973\n\0

返 回 值 返回指定日期时间字符串的指针，该字符串是静态的，每次调用 asctime 都将重写。

可移植性 适用于 UNIX 系统并且在 ANSI C 中也有定义。

参 见 ctime, difftime, ftime, gmtime, localtime, mktime, strftime, stime, time, tzset

示 例 #include <stdio.h>

```

#include <string.h>
#include <time.h>
int main(void)
{
 struct tm t;
 char str[80];
 /* sample loading of tm structure */
 t.tm_sec = 1; /* Seconds */
 t.tm_min = 30; /* Minutes */
 t.tm_hour = 9; /* Hour */
 t.tm_mday = 22; /* Day of the Month */
 t.tm_mon = 11; /* Month */
 t.tm_year = 56; /* Year — does not include century */
 t.tm_wday = 4; /* Day of the week */
 t.tm_yday = 0; /* Does not show in asctime */
 t.tm_isdst = 0; /* Is Daylight SavTime; does not show in asctime */
 /* converts structure to null terminated string */
 strcpy(str, asctime(&t));
 printf("%s\n", str);
 return 0;
}

```

## asin 反正弦函数

- 用 法** 对于实数 对于复数
- #include<math.h> #include<complex.h>
- double asin(double x); complex asin(complex x);
- 原 型 在** 对于实数 对于复数
- math.h complex.h
- 说 明** 带实参数的 asin 返回输入值的反正弦值。asin 的实参数必须在 -1 和 1 之间，否则 asin 返回 NAN 并置全局变量 errno 为：
- EDOM 域错误
- 复数的反正弦定义为：
- $\text{asin}(z) = -i * \log(i * z + \sqrt{1 - z^2})$
- 返 回 值** 返回一范围在  $-\pi/2$  到  $\pi/2$  之间的值。
- 本函数的错误处理程序可以通过 matherr 函数修改。
- 可移植性** 实数版本的 asin 适用于 UNIX 系统，并且在 ANSI C 中也有定义；它的复数版本只适用于 Turbo C，不能移植。
- 参 见** acos, atan, atan2, complex, cos, matherr, sin, tan
- 示 例** #include <stdio.h>
- #include <math.h>
- int main(void)

```

{
 double result;
 double x = 0.5;
 result = asin(x);
 printf("The arc sin of %lf is %lf\n", x, result);
 return(0);
}

```

## ■ assert 条件终止函数

用 法 #include <assert.h>

void assert(int test);

原 型 在 assert.h

说 明 assert 是一个可扩展为 if 语句的宏。如果 test 等于 0, 函数将在 stderr 输出一信息并调用 abort 终止程序。输出的信息为:

Assertion failed: test, file filename, line linenum

其中 filename 和 linenum 是源文件名和 assert 宏所在的行号。

如果在源文件的 #include <assert.h> 语句之前出现了 #define NDEBUG 指令, 其作用是注释 assert。

返 回 值 无。

可移植性 该宏适用于某些 UNIX 系统(包括系统 III 和系统 V), 并和 ANSI C 兼容。

参 见 abort

示 例 #include <assert.h>

#include <stdio.h>

#include <stdlib.h>

struct ITEM {

int key;

int value;

};

/\* add item to list, make sure list is not null \*/

void additem(struct ITEM \*itemptr) {

assert(itemptr != NULL);

/\* add item to list \*/

}

int main(void)

{

additem(NULL);

return 0; }

## ■ atan 反正切函数

用 法 对于实数

#include <math.h>

对于复数

#include <complex.h>

|       |                                                                                                                                                                                                                       |                                       |
|-------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------|
|       | <code>double atan(double x);</code>                                                                                                                                                                                   | <code>complex atan(complex x);</code> |
| 原 型 在 | 对于实数<br><code>math.h</code>                                                                                                                                                                                           | 对于复数<br><code>complex.h</code>        |
| 说 明   | <code>atan</code> 计算输入值的反正切值。<br>复数的反正切定义为：<br>$\text{atan}(z) = -0.5i \log(1+iz)/(1-iz)$                                                                                                                             |                                       |
| 返 回 值 | 输入实参数时返回一个范围在 $-\pi/2$ 到 $\pi/2$ 间的值。<br>本函数的错误处理程序可以通过 <code>matherr</code> 函数修改。                                                                                                                                    |                                       |
| 可移植性  | 实数类型的 <code>atan</code> 函数适用于 UNIX 系统并在 ANSI C 中有定义；复数类型的 <code>atan</code> 函数只适用于 Turbo C, 不能移植。                                                                                                                     |                                       |
| 参 见   | <code>acos</code> , <code>asin</code> , <code>atan2</code> , <code>complex</code> , <code>cos</code> , <code>matherr</code> , <code>sin</code> , <code>tan</code>                                                     |                                       |
| 示 例   | <pre>#include &lt;stdio.h&gt; #include &lt;math.h&gt;  int main(void) {     double result;     double x = 0.5;     result = atan(x);     printf("The arc tangent of %lf is %lf\n", x, result);     return(0); }</pre> |                                       |

## ■ `atan2` 计算 $y/x$ 的反正切值

|       |                                                                                                                                                                                                                                   |
|-------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 用 法   | <code>#include &lt;math.h&gt;</code><br><code>double atan2(double y, double x);</code>                                                                                                                                            |
| 原 型 在 | <code>math.h</code>                                                                                                                                                                                                               |
| 说 明   | <code>atan2</code> 返回 $y/x$ 的反正切值, 即使角度接近 $\pi/2$ 或 $-\pi/2$ (即 $x$ 接近 0) 仍能得到正确结果。<br>如果 $x=y=0$ , 本函数将置全局变量 <code>errno</code> 为 <code>EDOM</code> 。                                                                            |
| 返 回 值 | <code>atan2</code> 返回一个范围在 $-\pi$ 到 $\pi$ 的值。<br>本函数的错误处理程序可以通过 <code>matherr</code> 函数进行修改。                                                                                                                                      |
| 可移植性  | 适用于 UNIX 系统, 在 ANSI C 中也有定义。                                                                                                                                                                                                      |
| 参 见   | <code>acos</code> , <code>asin</code> , <code>atan</code> , <code>cos</code> , <code>matherr</code> , <code>sin</code> , <code>tan</code>                                                                                         |
| 示 例   | <pre>#include &lt;stdio.h&gt; #include &lt;math.h&gt;  int main(void) {     double result;     double x = 90.0, y = 45.0;     result = atan2(y, x);     printf("The arc tangent ratio of %lf is %lf\n", (y / x), result); }</pre> |

```
 return 0;
}
```

## ■ atexit 定义终止函数

用 法 `#include <stdlib.h>`

```
int atexit(atexit_t func);
```

说 明 本函数将把 func 定义的函数定义为“退出(exit)函数”。在正常调用 exit 终止程序时,exit 将调用函数 func()。

每次 atexit 调用说明一个退出函数,在一个程序中最多可以说明 32 个退出函数,它们按后进先出的顺序执行(即最后定义的终止函数最先执行)。

返回值 如果调用成功返回 0,否则返回非零值(没有足够的空间定义终止函数)。

可移植性 atexit 与 ANSI C 兼容。

参 见 abort, \_exit, exit, spawn...

示 例

```
#include <stdio.h>
#include <stdlib.h>
void exit_fn1(void)
{
 printf("Exit function #1 called\n");
}
void exit_fn2(void)
{
 printf("Exit function #2 called\n");
}
int main(void)
{
 /* post exit function #1 */
 atexit(exit_fn1);
 /* post exit function #2 */
 atexit(exit_fn2);
 return 0;
}
```

## ■ atof 将字符串转换成浮点数

用 法 `#include <math.h>`

```
double atof(const char *s);
```

原型在 math.h,stdlib.h

说 明 本函数把 s 所指向的字符串转换成 double 类型。它识别由以下元素组成的浮点数表达式:

- 包含制表符和空格的字符串(white space);
- 任选的正负符号(sign);
- 数字串和任选的十进制小数点(小数点左右都可以有数字)(ddd.ddd);

● 可选的 e 或 E, 后跟任选的带符号整数(有 e | E [sign] ddd);

使用这些字符时必须按以下格式:

[white space] [sign] [ddd] [. ] [ddd] [e|E[sign]ddd]

atof 能够识别 +INF 和 -INF, 它们表示正和负无限大; 它还能够识别 +NAN 和 -NAN, 这表示该字符串不是数值。

在函数执行过程中, 如果遇到一个不可识别的字符, 则函数终止转换。

strtod 和 atof 相似, 但它能提供更强有力的错误检测, 因而更适合于某些应用。

**返回值** atof 返回输入字符串的转换值。

如果溢出, atof 返回正 HUGE\_VAL 或负 HUGE\_VAL, 并将 errno 置为 ERANGE, 但不调用 matherr。

**可移植性** atof 适用于 UNIX 系统并在 ANSI C 中也有定义。

**参见** atoi, atol, ecvt, fcvt, gcvt, scanf, strtod

**示例**

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

int main(void)
{
 float f;
 char *str = "12345.67";
 f = atof(str);
 printf("string = %s float = %f\n", str, f);
 return 0;
}
```

## ■ atoi 把字符串转换成整数

**用法** #include <stdlib.h>  
int atoi(const char \*s);

**原型在** stdlib.h

**说明** 本函数将把 s 指向的字符串转换成一个 int 型整数, atoi 能够按以下顺序识别:

- 制表符和空格字符串(ws);
- 正负符号(sn);
- 数字串(ddd);

以上字符必须按如下格式出现:

[ws] [sn] [ddd]

在函数中执行过程中, 若遇到一个不可识别的字符, 则终止转换。

在函数 atoi 中没有对溢出进行规定, 即溢出时结果没有定义。

**返回值** atoi 返回输入串的整型转换值, 如果转换错误, 则返回 0。

**可移植性** 适用于 UNIX 系统, 在 ANSI C 中也有定义。

**参见** atof, atol, ecvt, fcvt, gcvt, scanf, strtod

**示例**

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

```
int main(void)
{
 int n;
 char *str = "12345.67";
 n = atoi(str);
 printf("string = %s integer = %d\n", str, n);
 return 0;
}
```

## ■ atol 把字符串转换成长整型

用 法 #include <stdlib.h>

long atol(const char \*s);

原 型 在 stdlib.h

说 明 atol 将把 s 指向的字符串转换成 long 长整型数,它能按如下顺序识别:

- 制表符和空格字符串(ws);
- 正负符号(sn);
- 数字串(ddd)。

以下字符必须按如下格式出现:

[ws] [sn] [ddd]

在函数执行过程中,若遇到第一个不可识别的字符,则终止转换。

atol 中没有对溢出的规定,即溢出时其结果没有定义。

返 回 值 atol 返回输入串的转换值,如果转换失败,返回 0。

可移植性 对于 UNIX 系统也适用,并且在 ANSI C 中也有定义。

参 见 atof, atoi, ecvt, fcvt, gcvt, scanf, strtod, strtol, strtoul

示 例

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
 long l;
 char *lstr = "98765432";
 l = atol(lstr);
 printf("string = %s integer = %ld\n", lstr, l);
 return(0);
}
```

## ■ bar 画二维条形图

用 法 #include <graphics.h>

#include <conio.h>

void far bar(int left, int top, int right, int bottom);

原 型 在 graphics.h

**说 明** bar 画一填充的长方二维条形, 该条形使用当前填充模式和填充颜色填充。bar 并不画出条形轮廓; 若要画二维条形轮廓, 可调用 bar3d 并将参数 depth 置为 0。矩形的左上角右下角分别由 (left, top) 和 (right, bottom) 给出。这两个参数指的是象点的坐标。

**返回 值** 无。

**可移植性** 只适用于 Turbo C, 并且只能在配备图形适配器的 IBM PC 及其兼容机上运行。

**参 见** bar3d, rectangle, setcolor, setfillstyle, setlinestyle

**示 例**

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int midx, midy, i;
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
 }
 midx = getmaxx() / 2;
 midy = getmaxy() / 2;
 /* loop through the fill patterns */
 for (i=SOLID_FILL; i<USER_FILL; i++)
 {
 /* set the fill style */
 setfillstyle(i, getmaxcolor());
 /* draw the bar */
 bar(midx-50, midy-50, midx+50,
 midy+50);
 getch();
 }
 /* clean up */
 closegraph();
 return 0;
}
```



}

## ■ bar3d 画一个三维条形图 ■

用 法 `#include <graphics.h>`

`void far bar3d(int left, int top, int right, int bottom, int depth,  
int topflag);`

原 型 在 `graphics.h`

说 明 `bar3d` 画一个三维的矩形条形图,然后用当前填充模式和填充颜色填充矩形。条形图的外廓用当前的线型和颜色画出,以像素为单位的条形深度则由 `depth` 给出。`topflag` 参数决定是否在条形图上放一个三维顶。若 `topflag` 不等于 0,则 `bar3d` 将绘制一个矩形顶,否则不绘制矩形顶(这样用户就可以将几个条形图叠在一起)。

矩形的左上角和右下角分别由 `(left,top)` 和 `(right,bottom)` 给出。

要计算 `bar3d` 的典型深度,可取二维图形宽度的  $1/4$ ,如:

`bar3d(left,top,right,bottom, (right-left)/4,1);`

返 回 值 无。

可移植性 只适用于 Turbo C,并且只能在配备图形适配器的 IBM PC 及其兼容机上运行。

参 见 `bar`, `rectangle`, `setcolor`, `setfillstyle`, `setlinestyle`

示 例 `#include <graphics.h>`

`#include <stdlib.h>`

`#include <stdio.h>`

`#include <conio.h>`

`int main(void)`

{

`/* request auto detection */`

`int gdriver = DETECT, gmode, errorcode;`

`int midx, midy, i;`

`/* initialize graphics, local variables */`

`initgraph(&gdriver, &gmode, "");`

`/* read result of initialization */`

`errorcode = graphresult();`

`if (errorcode != grOk) /* an error occurred */`

{

`printf("Graphics error: %s\n", grapherrormsg(errorcode));`

`printf("Press any key to halt;");`

`getch();`

`exit(1); /* terminate with error code */`

}

`midx = getmaxx() / 2;`

`midy = getmaxy() / 2;`

`/* loop through the fill patterns */`

```

for (i=EMPTY_FILL; i<USER_FILL; i++)
{
 /* set the fill style */
 setfillstyle(i, getmaxcolor());
 /* draw the 3-d bar */
 bar3d(midx-50, midy-50, midx+50, midy+50, 10, 1);
 getch();
}
/* clean up */
closegraph();
return 0;
}

```

## ■ bdos DOS 系统调用

**用 法** #include <dos.h>

int bdos(int dosfun, unsigned dosdx, unsigned dosal);

**原 型 在** dos.h

**说 明** 本函数将直接执行 DOS 系统调用,关于每个系统调用的细节,请参见 DOS 参考手册。

参数为整型的系统调用使用 bdos 函数,参数为指针的系统调用则须使用 bdosptr 函数。

在大数据模式下(紧缩、大型、巨型),必须要用 bdosptr 来完成指针参数的系统调用,而不能用 bdos。

dosfun 在 DOS 参考手册中定义。

dosax 是 DX 寄存器的值。

dosax 是 AL 寄存器的值。

**返 回 值** 调用成功时,返回值为 AX 值;若调用失败,则函数返回 -1,并对全局变量 error 和 \_doserror 进行设置。

**可移植性** 只适用于 DOS。

**参 见** bdosptr, geninterrupt, int86, int86x, intdos, intdosx

**示 例**

```

#include <stdio.h>
#include <dos.h>
/* Get current drive as 'A', 'B', ... */
char current_drive(void)
{
 char curdrive;
 /* Get current disk as 0, 1, ... */
 curdrive = bdos(0x19, 0, 0);
 return('A' + curdrive);
}

int main(void)

```

```

{
 printf("The current drive is %c:\n", current_drive());
 return 0;
}

```

## ■ bdosptr DOS 系统调用 ■

**用 法** #include <dos.h>

int bdosptr(int dosfun, void argument, unsigned dosal);

**原 型 在** dos.h

**说 明** 本函数将直接执行 DOS 系统调用, 关于每个系统调用的细节, 请参见 DOS 参考手册。

参数为整型的系统调用使用 bdos 函数, 参数为指针的系统调用则须使用 bdosptr 函数。

在大模式下(紧缩、大型、巨型), 要用 bdosptr 函数来完成指针参数的系统调用, 而不能用 bdos 函数。

dosfun 在 DOS 参考手册中定义。

在小模式下, bdosptr 的 argument 参数传给 DX, 在大模式下, 它给出 DS:DX 值, 供系统调用使用。

dosal 是 AL 寄存器的值。

**返 回 值** 调用成功时, 返回值为 AX 值; 若调用失败, 则返回 -1, 并将全局变量 errno 和 doserrno 置上相应的值。

**可移植性** 只适用于 DOS。

**参 见** bdos, geninterrupt, int86, int86x, intdos, intdosx。

**示 例**

```

#include <string.h>
#include <stdio.h>
#include <dir.h>
#include <dos.h>
#include <errno.h>
#include <stdlib.h>
#define BUFLen 80
int main(void)
{
 char buffer[BUFLen];
 int test;
 printf("Enter full pathname of a directory\n");
 gets(buffer);
 test = bdosptr(0x3B, buffer, 0);
 if(test)
 {
 printf("DOS error message: %d\n", errno);
 /* See errno.h for error listings */
 }
}

```

```

 exit (1);
 }

 getcwd(buffer, BUFLen);
 printf("The current directory is: %s\n", buffer);
 return 0;
}

```

## ■ bioscom I/O 通信

**用 法** #include <bios.h>

int bioscom(int cmd, char abyte, int port);

**原 型 在** bios.h

**说 明** bioscom 在由 port 指定的 I/O 端口上执行 RS-232 通信操作。  
port 值为 0 表示串行端口 1 (COM1), 1 表示串行口 2 (COM2), 依此类推。  
cmd 的允许值及其作用如下:

---

|      |                    |      |         |
|------|--------------------|------|---------|
| 0    | 设置通信参数为 abyte 值;   |      |         |
| 1    | 按 abyte 发送字符到通信线上; |      |         |
| 2    | 从通信线上接受一个字符;       |      |         |
| 3    | 返回通信端口的当前状态;       |      |         |
| 0x02 | 7 个数据位             | 0x00 | 110 波特  |
| 0x03 | 8 个数据位             | 0x20 | 150 波特  |
|      |                    | 0x40 | 300 波特  |
| 0x00 | 1 个停止位             | 0x60 | 600 波特  |
| 0x04 | 2 个停止位             | 0x80 | 1200 波特 |
| 0x00 | 无校验                | 0xA0 | 2400 波特 |
| 0x08 | 奇校验                | 0xC0 | 4800 波特 |
| 0x18 | 偶校验                | 0xE0 | 9600 波特 |

---

例如, abyte 值为 0xEB (0xE0 | 0x08 | 0x00 | 0x03), 则通讯口为 9600 波特、奇校验、1 个停止位、8 个数据位。

bioscom 在执行过程中使用了中断 0x14。

**返 回 值** 对所有的 cmd 值均返回一 16 位整数, 其中高 8 位是状态位, 低 8 位则随 cmd 值的变化而变化。返回值的高 8 位定义如下:

---

|        |          |
|--------|----------|
| 第 15 位 | 超时       |
| 第 14 位 | 传送移位寄存器空 |
| 第 13 位 | 传送保持寄存器空 |
| 第 12 位 | 中断检测     |
| 第 11 位 | 页帧错误     |
| 第 10 位 | 校验错误     |
| 第 9 位  | 溢出超越错误   |
| 第 8 位  | 数据就绪     |

---

如果不能按时发送 abyte 值,则第 15 位置 1。在出现其它情况时,高位和低位将被置位。如,页帧错误时,第 11 位置 1。

当 cmd 的值为 2 时,如果不发生错误,返回值的低位为读入字节。如果发生了错误,则至少有一高位被置为 1。所以若无高位被置为 1,则表明在接收字节过程中没有发生错误。

当 cmd 的值为 0 或 3 时,返回值的高 8 位设置如上所述,低 8 位则定义如下:

- 
- 第 7 位 接收线信号检测
  - 第 6 位 响铃指示
  - 第 5 位 数据设置就绪
  - 第 4 位 清除发送
  - 第 3 位 接收线信号检测改变
  - 第 2 位 后沿响铃检测
  - 第 1 位 数据设置就绪改变
  - 第 0 位 清除发送改变
- 

**可移植性** 本函数仅适用于 IBM PC 及其兼容机。

**示 例**

```
#include <bios.h>
#include <conio.h>
#define COM1 0
#define DATA_READY 0x100
#define TRUE 1
#define FALSE 0
#define SETTINGS (0x80 | 0x02 | 0x00 | 0x00)
int main(void)
{
 int in, out, status, DONE = FALSE;
 bioscom(0, SETTINGS, COM1);
 cprintf("... BIOSCOM [ESC] to exit ...\n");
 while (!DONE)
 {
 status = bioscom(3, 0, COM1);
 if (status & DATA_READY)
 {
 if ((out = bioscom(2, 0, COM1) & 0x7F) != 0)
 putchar(out);
 if (kbhit())
 {
 if ((in = getch()) == '\x1B')
 {
 DONE = TRUE;
 bioscom(1, in, COM1);
 }
 }
 }
 }
}
```

```
return 0;
```

## ■ biosdisk 调用 BIOS 磁盘驱动程序 ■

**用 法** `#include <bios.h>`

`int biosdisk(int cmd, int drive, int head, int track, int sector,  
int nsects, void * buffer);` 原型在 `bios.h`

**说 明** 本函数使用中断 0x13, 直接调用 BIOS 进行磁盘操作。

`drive` 表示磁盘驱动器号; 0 代表第一个软驱, 1 表示第二个软驱, 2 表示第三个等等。对于硬盘驱动器, `drive` 的值为 0x80 表示第一个硬驱, 为 0x81 表示第二个硬驱, 0x82 表示第三个等等。

对于硬盘, 指定的驱动器为物理驱动器而不是硬盘分区, 因此应用程序必须自行解释分区表信息。

`cmd` 表示待执行的操作, 其它参数的选用则根据 `cmd` 的值决定。下面列出对应于 IBM PC、XT、AT、PS/2 或兼容系统可能的 `cmd` 值:

- 0 复位磁盘系统。强制磁盘驱动器控制器进行硬复位, 忽略其它参数。
- 1 返回最后一次磁盘操作的状态, 忽略其它参数。
- 2 读一个或多个扇区内容至内存缓冲区。起始扇区号由 `head`, `track` 和 `sector` 给定, 扇区数由 `nsects` 给出, 数据以每扇区 512 字节的格式读入缓冲区 `buffer`。
- 3 将内存缓冲区中的数据写入一个或多个磁盘扇区中。起始扇区由 `head`, `track` 和 `sector` 给定, 扇区数由 `nsects` 给出, 数据从缓冲区 `buffer` 中按每扇区 512 字节写入格式磁盘。
- 4 对一个或多个扇区进行数据校验, 起始扇区由 `head`, `track` 和 `sector` 给定, 扇区数由 `nsects` 给出。
- 5 格式化一个磁道。磁道号由 `head`, `track` 给定, `buffer` 指向一个将写到给定磁道上的扇区头表。关于这个表的说明和格式化操作, 请参阅《IBM PC 技术参考手册》。

以下 `cmd` 值仅仅适用于 XT、AT、PS/2 及其兼容机:

- 6 格式化一个磁盘并对坏扇区设置坏标志。
- 7 从指定磁道开始, 格式化一个磁盘。
- 8 返回当前驱动器参数。驱动器信息返回在 `buffer` 的前 4 个字节中。
- 9 初始化驱动器的寄存器。
- 10 进行长读。每扇区读入 512 字节加上额外 4 个字节。
- 11 进行长写。每扇区写入 512 字节加上额外 4 个字节。
- 12 进行磁盘查找。
- 13 交替磁盘复位。
- 14 读指定扇区内容至缓冲区。
- 15 缓冲区内容写入指定扇区。

- 16 测试指定驱动器是否准备就绪。
- 17 重新校准驱动器。
- 18 驱动器内控制器 RAM 诊断。
- 19 驱动器诊断。
- 20 驱动器内控制器的内部诊断。

注意, biosdisk 是对比文件操作更低级的物理扇区进行操作, 它有可能破坏硬盘上的文件内容和目录。

**返回值** 这些操作返回值为一个状态字节, 其含义如下:

- 0x00 操作成功
- 0x01 错误命令
- 0x02 地址标志没找到
- 0x03 企图写具有写保护的磁盘
- 0x04 找到指定扇区
- 0x05 复位失败(对硬盘)
- 0x06 上次操作后磁盘更换
- 0x07 驱动器参数设置错误
- 0x08 DMA 运行过载
- 0x09 DMA 边界错
- 0x0A 检测到坏扇区
- 0x0B 检测到坏磁道
- 0x0C 不支持的磁盘介质类型
- 0x10 磁盘读/CRC/ECC 错误
- 0x11 CRC/ECC 纠正数据错误
- 0x20 控制器失败
- 0x40 查找操作失败
- 0x80 超时
- 0xAA 驱动器没有准备就绪(只对硬盘有效)
- 0xBB 未定义的错误出现(只对硬盘有效)
- 0xCC 出现写错误
- 0xE0 状态错误
- 0xFF 有意义的操作失败
- 0x11 不是错误, 因为数据实际是正确的, 该返回值只是给应用程序一个自行决定的机会。

**可移植性** 只适用于 IBM PC 及其兼容机。

**参 见** absread, abswrite

**示 例**

```
#include <bios.h>
#include <stdio.h>
int main(void)
{
```

```

int result;
char buffer[512];
printf("Testing to see if drive a, is ready\n");
result = biosdisk(4,0,0,0,0,1,buffer);
result &= 0x02;
(result) ? (printf("Drive A: Ready\n")) :
 (printf("Drive A: Not Ready\n"));
return 0;
}

```

## ■ biosequip 检查设备

用 法 #include <bios.h>

int biosequip(void);

原 型 在 bios.h

说 明 本函数返回一个描述与系统相连设备的整数。它使用了 BIOS 中断 0x11。

返 回 值 返回值解释为位集合。在 IBM PC 上有其对应值：

第 14~15 位为安装的并行打印机的数目：

00=0 个打印机

01=1 个打印机

10=2 个打印机

11=3 个打印机

第 13 位 是否连接串行打印机

第 12 位 是否连接游戏口(I/O)

第 9~11 位 串行端口数

000=0 个端口

001=1 个端口

010=2 个端口

011=3 个端口

100=4 个端口

101=5 个端口

110=6 个端口

111=7 个端口

第 8 位 直接内容存取(DMA)

0=机器有 DMA

1=机器没有 DMA,如 PC jr.

第 6~7 位 硬盘数

00=1 个驱动器

01=2 个驱动器

10=3 个驱动器

11=4 个驱动器,仅当第 0 位是 1 时



## 第 4~5 位 初始视频模式

00=未使用

01=40×25 BW 带彩色卡

10=80×25 BW 带彩色卡

11=80×25 BW 带单色卡

## 第 2~3 位 母板 RAM 大小

00=16K

01=32K

10=48K

11=64K

## 第 1 位 是否有数字浮点协处理器

## 第 0 位 是否从软盘启动

可移植性 只适用于 IBM PC 及其兼容机

示 例

```
#include <bios.h>
#include <stdio.h>
#define CO_PROCESSOR_MASK 0x0002
int main(void)
{
 int equip_check;
 /* get the current equipment configuration */
 equip_check = biosequip();
 /* check to see if there is a coprocessor installed */
 if (equip_check & CO_PROCESSOR_MASK)
 printf("There is a math coprocessor installed. \n");
 else
 printf("No math coprocessor installed. \n");
 return 0;
}
```

## ■ bioskey 调用 BIOS 的键盘接口

用 法 #include <bios.h>  
int bioskey(int cmd);

原型在 bios.h

说 明 本函数使用 BIOS 中断 0x16 执行各种键盘操作。参数 cmd 用来确定实际执行的操作。

返回值 该函数的返回值取决于由 cmd 值所指定的操作:

- 0 返回下一从键盘输入的字符。如果低 8 位非零,则为 ASCII 字符;如果低 8 位为零,则高 8 位为扩展键盘码,扩展键盘码的定义请参见《IBM PC 技术参考手册》。
- 1 测试用户是否按键。如果函数返回 0,则表示没有;否则返回所输入键值并保

存,供下次参数 cmd 为 0 时的 bioskey 调用返回。

## 2 获取当前换档键状态。状态值各位的含义如下:

|     |      |                |
|-----|------|----------------|
| 位 7 | 0x80 | Insert ON      |
| 位 6 | 0x40 | Caps ON        |
| 位 5 | 0x20 | Numlock ON     |
| 位 4 | 0x10 | Scroll Lock ON |
| 位 3 | 0x08 | 按下 Alt 键       |
| 位 2 | 0x04 | 按下 Ctrl 键      |
| 位 1 | 0x02 | 按下左 Shift 键    |
| 位 0 | 0x01 | 按下右 Shift 键    |

**可移植性** 只适用于 IBM PC 及其兼容机。

**示 例**

```
#include <stdio.h>
#include <bios.h>
#include <ctype.h>
#define RIGHT 0x01
#define LEFT 0x02
#define CTRL 0x04
#define ALT 0x08
int main(void)
{
 int key, modifiers;
 /* function 1 returns 0 until a key is pressed */
 while (bioskey(1) == 0);
 /* function 0 returns the key that is waiting */
 key = bioskey(0);
 /* use function 2 to determine if shift keys were used */
 modifiers = bioskey(2);
 if (modifiers)
 {
 printf("[");
 if (modifiers & RIGHT) printf("RIGHT");
 if (modifiers & LEFT) printf("LEFT");
 if (modifiers & CTRL) printf("CTRL");
 if (modifiers & ALT) printf("ALT");
 printf("]");
 }
 /* print out the character read */
 if (isalnum(key & 0xFF))
 printf("' %c' \n", key);
 else
 printf(" % #02x \n", key);
 return 0;
}
```

}

## ■ biosmemory 返回内存大小

用 法 `#include <bios.h>`

`int biosmemory(void);`

原 型 在 `io.h`

说 明 `biosmemory` 通过 BIOS 中断 0x12 返回以 K 为单位的 RAM 大小,它不包括显示存储、扩展内存(extended memory)和扩充内存(expanded memory)。

返 回 值 返回以 1K 为单位的内存大小。

可移植性 只适用于 IBM PC 及其兼容机。

示 例 `#include <stdio.h>`

`#include <bios.h>`

`int main(void)`

{

`int memory_size;`

`memory_size = biosmemory(); /* returns value up to 640K */`

`printf("RAM size = %dK\n",memory_size);`

`return 0;`

}

## ■ biosprint 调用 BIOS 的打印机 I/O 接口

用 法 `#include <bios.h>`

`int biosprint(int cmd, int abyte, int port);`

原 型 在 `bios.h`

说 明 本函数通过 BIOS 中断 0x17 在由参数 `port` 指定的打印机上完成各种打印机功能。

`port` 值为 0 时对应 LPT1,`port` 值为 1 时对应 LPT2,依此类推。

`cmd` 的值为下列之一:

- |   |                         |
|---|-------------------------|
| 0 | 打印字符 <code>abyte</code> |
| 1 | 初始化打印机端口                |
| 2 | 读打印机状态                  |

`abyte` 的值可以是 0 到 255。

返 回 值 该函数将返回一打印机状态字节,状态字节中各数据位的含义如下:

- |     |      |        |
|-----|------|--------|
| 位 0 | 0x01 | 设备超时   |
| 位 3 | 0x08 | I/O 出错 |
| 位 4 | 0x10 | 打印机已选择 |
| 位 5 | 0x20 | 没纸     |
| 位 6 | 0x40 | 打印机确认  |
| 位 7 | 0x80 | 不忙     |

可移植性 只适用于 IBM PC 及其兼容机。

```

示 例 #include <stdio.h>
 #include <conio.h>
 #include <bios.h>
 int main(void)
 {
 #define STATUS 2 /* printer status command */
 #define PORTNUM 0 /* port number for LPT1 */
 int status, abyte=0;
 printf("Please turn off your printer. Press any key to continue\n");
 getch();
 status = biosprint(STATUS, abyte, PORTNUM);
 if (status & 0x01) printf("Device time out. \n");
 if (status & 0x08)
 printf("I/O error. \n");
 if (status & 0x10)
 printf("Selected. \n");
 if (status & 0x20)
 printf("Out of paper. \n");
 if (status & 0x40)
 printf("Acknowledge. \n");
 if (status & 0x80)
 printf("Not busy. \n");
 return 0;
 }

```

## ■ biostime 读取或设置 BIOS 时钟 ■

用 法 #include <bios.h>  
 long biostime(int cmd, long newtime);

原 型 在 bios.h

说 明 本函数读取或设置 BIOS 时钟。该时钟从午夜开始以每秒约 18.2 次的速率计时。在 biostime 执行过程中,使用 BIOS 中断 0x1A。

如果 cmd=0, biostime 返回时钟当前值;如果 cmd=1, 计时器设置为长整型的 newtime 值。

返 回 值 当 biostime 读取 BIOS 时钟(cmd=0)时,返回时钟的当前值。

可移植性 只适用于 IBM PC 及其兼容机。

```

示 例 #include <stdio.h>
 #include <bios.h>
 #include <time.h>
 #include <conio.h>
 int main(void)
 {

```

```

long bios_time;
clrscr();
printf("The number of clock ticks since midnight is:\r\n");
printf("The number of seconds since midnight is:\r\n");
printf("The number of minutes since midnight is:\r\n");
printf("The number of hours since midnight is:\r\n");
printf("\r\nPress any key to quit:");
while(! kbhit())
{
 bios_time = biostime(0, 0L);
 gotoxy(50, 1);
 printf("%lu", bios_time);
 gotoxy(50, 2); printf("%.4f", bios_time / CLK_TCK);
 gotoxy(50, 3);
 printf("%.4f", bios_time / CLK_TCK / 60);
 gotoxy(50, 4);
 printf("%.4f", bios_time / CLK_TCK / 3600);
}
return 0;
}

```

## ■ brk 改变数据段内存分配

**用 法** #include <alloc.h>  
int brk(void \* addr);

**原 型** 在 alloc.h

**说 明** brk 函数将动态改变分配给调用程序数据段的内存,这种改变是通过重置程序的结束地址实现的,文件的结束地址是数据段结尾处之上的第一个位置的地址,分配的存储空间是随着文件结束地址的增大而增大。brk 将把 addr 设置为文件的结束地址,并相应地改变内存分配。

如果这种修改将导致分配的内存超过允许范围,则函数将不对内存作任何改变。

**返 回 值** 若调用成功,返回 0,若调用失败,返回 -1,并置 errno 为:

ENOMEM 无足够内存

**可移植性** 对 UNIX 系统也适用。

**参 见** coreleft, sbrk

**示 例**

```

#include <stdio.h>
#include <alloc.h>
int main(void)
{
 char * ptr;
 printf("Changing allocation with brk()\n");
 ptr = malloc(1);

```

```

printf("Before brk() call: %lu bytes free\n", coreleft());
brk(ptr+1000);
printf(" After brk() call: %lu bytes free\n", coreleft());
return 0;
}

```

## ■ bsearch 数组的二分法搜索 ■

**用 法** #include <stdlib.h>

```

void * bsearch(const void * key, const void * base, size_t nelem,
size_t width, int (* fcmp)(const void, const void));

```

**原 型 在** stdlib.h

**说 明** bsearch 以内存中 nelem 个元素的表(或数组)进行搜索,并返回表中与搜索关键字 key 相匹配的表项。如果在表中无匹配项,则函数返回 0。

类型 `siz_t` 已定义为无符号整数

● 参数 nelem 给出表中的元素个数

● 参数 width 给出表中每项的字节数

调用比较函数 \* fcmp 时有两个参数 elem1 和 elem2,它们分别指向一个待比较的元素,比较函数将比较这两个参数指向的元素(elem1 和 elem2),并根据比较结果返回一个相应的整数。

对 bsearch 而言, \* fcmp 的返回值为:

|      |                       |
|------|-----------------------|
| < 0  | 如果 * elem1 < * elem2  |
| == 0 | 如果 * elem1 == * elem2 |
| > 0  | 如果 * elem1 > * elem2  |

**返 回 值** 返回与指定关键字相匹配的第一个表项地址。若没有与关键字相匹配的表项,则函数返回 0。

**可移植性** 适用于 UNIX 系统并且在 ANSI C 中也有定义。

**参 见** lfind, lsearch, qsort

**示 例** #include <stdlib.h>

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

```
#define NELEMS(arr) (sizeof(arr) / sizeof(arr[0]))
```

```
int numarray[] = {123, 145, 512, 627, 800, 933};
```

```
int numeric(const int * p1, const int * p2)
```

```
{
```

```
 return(* p1 - * p2);
```

```
}
```

```
int lookup(int key)
```

```
{
```

```
 int * itemptr;
```

```
 /* The cast of (int(*) (const void *, const void *))
```

```
 is needed to avoid a type mismatch error at
```

```

 compile time */
 itemptr = bsearch (&key, numarray, NELEMS(numarray),
 sizeof(int), (int (*)(const void *, const void *))numeric);
 return (itemptr != NULL);
}

int main(void)
{
 if (lookup(512))
 printf("512 is in the table. \n");
 else
 printf("512 isn't in the table. \n");
 return 0;
}

```

## ■ cabs 计算复数的模

**用 法** #include<math.h>

double cabs(struct complex z);

**原 型 在** math.h

**说 明** cabs 是计算复数 z 的模的宏, z 是一个类型为 complex 的结构, 在 math.h 中此结构的定义如下:

```

struct complex {
 double x,y;
};

```

其中 x 是实部, y 是虚部。

调用 cabs 与调用 sqrt 计算  $\sqrt{z.x * z.x + z.y * z.y}$  是相同的。若使用 Turbo C, 则最好是调用函数 abs 并使用 complex.h 中的 complex 类型。

**返 回 值** cabs 返回复数 z 的模, 并且是一个双精度数。若结果溢出, cabs 将返回 HUGE\_VAL 并置全局变量 errno 为:

ERANGE 结果越界

cabs 的错误处理程序可以通过 matherr 函数修改。

**可移植性** 对于 UNIX 系统也适用。

**参 见** abs, comple, fabs, labs, matherr

**示 例** #include <stdio.h>

#include <math.h>

int main(void)

```

{
 struct complex z;
 double val;
 z.x = 2.0;
 z.y = 1.0;
 val = cabs(z);
}

```

```
printf("The absolute value of %.2lf %.2lf is %.2lf", z.x, z.y, val);
return 0;
}
```

## ■ calloc 分配内存

用 法 #include <stdlib.h>

```
void *calloc(size_t nitems, size_t size);
```

原 型 在 stdlib.h, calloc.h

说 明 本函数提供了在 C 中对堆进行存取。堆用于建立长度可变存储块的动态存储分配,许多数据结构,如树和表都自然而然地应用了堆的内存分配形式。

除了堆栈顶紧接着的一个小边界外,数据段末尾与程序堆栈顶之间的空间在小模式(tiny、small 和 medium)中都可作为堆使用。边界则作为应用程序扩充堆栈和 DOS 所需的少量内存使用。

在大模式(compact、large 和 huge)中,程序堆栈以下直到物理内存顶端间的所有空间都分配给堆。

calloc 分配一块 nitems x size 大小的内存,将该内存块的内容全部清为 0。若要分配大于 64K 的内存块,必须使用 farcalloc 函数。

返 回 值 返回指向新分配内存的指针,若没有足够的空间分配给新块或者 nitems 与 size 的乘积为 0 值,则函数返回 NULL。

可移植性 适用于 UNIX 系统,在 ANSI C 中也有定义,并且与 Kernighan 和 Ritchie 的定义兼容。

参 见 farcalloc, free, malloc, realloc

示 例 #include <stdio.h>

```
#include <alloc.h>
```

```
int main(void)
```

```
{
```

```
 char *str = NULL;
```

```
 /* allocate memory for string */
```

```
 str = calloc(10, sizeof(char));
```

```
 /* copy "Hello" into string */
```

```
 strcpy(str, "Hello");
```

```
 /* display string */
```

```
 printf("String is %s\n", str);
```

```
 /* free memory */
```

```
 free(str);
```

```
 return 0;
```

```
}
```

## ■ ceil 舍入

用 法 #include <math.h>



```
double celi(double x);
```

原 型 在 math.h

说 明 ceil 函数将求得不小于 x 的最小整数。

返 回 值 返回所求得的最小整数值(双精度类型)。

可移植性 适用于 UNIX 系统,并且在 ANSI C 中也有定义。

参 见 floor, fmod

```
示 例 #include <math.h>
#include <stdio.h>
int main(void)
{
 double number = 123.54;
 double down, up;
 down = floor(number);
 up = ceil(number);
 printf("original number %5.2lf\n", number);
 printf("number rounded down %5.2lf\n", down);
 printf("number rounded up %5.2lf\n", up);
 return 0;
}
```

## ■ cgets 读字符串

用 法 #include <conio.h>

```
char * cgets(char * str);
```

原 型 在 conio.h

说 明 cgets 从控制台读入一个字符串,并将该字符串(和字符串长度)存入由 str 所指向的地址中。

cgets 将进行读字符,直到遇到 CR/LF(回车/换行)符或已读入最大允许的字符数。在 cgets 函数读入 CR/LF 键之后,它将保存读入的字符串并用空字符(\0)代替 CR/LF。

在调用 cgets 之前必须将要读入字符串的最大长度存入 str[0]中,返回时 str[1]被设置为实际读入的字符数。实际字符串内容从 str[2]开始,以空字符结尾。因此, str 的实际长度必须至少是 str[0]+2。

返 回 值 若调用成功,返回指向 str[2]的指针。

可移植性 只适用于 IBM PC 及其兼容机。

参 见 cputs, fgets, getch, getche, gets

```
示 例 #include <stdio.h>
#include <conio.h>
int main(void)
{
 char buffer[83];
```

```

char *p;
/* There's space for 80 characters plus the NULL terminator */
buffer[0] = 81;
printf("Input some chars:");
p = cgets(buffer);
printf("\ncgets read %d characters: \"%s\"\n", buffer[1], p);
printf("The returned pointer is %p, buffer[0] is at %p\n", p, &buffer);
/* Leave room for 5 characters plus the NULL terminator */
buffer[0] = 6;
printf("Input some chars:");
p = cgets(buffer);
printf("\ncgets read %d characters: \"%s\"\n", buffer[1], p);
printf("The returned pointer is %p, buffer[0] is at %p\n", p, &buffer);
return 0;
}

```

## ■ chdir 改变当前目录

**用 法** #include <dir.h>

int chdir(const char \*path);

**原 型 在** dir.h

**说 明** chdir 函数将把由 path 指定的目录改为当前目录, 在这里 path 必须是已存在的目录。在 path 参数中可以指定一驱动器号, 如 chdir("a:\\B&TC")。但这种改变只是能改变该驱动器上的当前目录, 而对当前活动驱动器上的当前目录不会有任何影响。

**返 回 值** 若调用成功, 返回 0; 否则返回 -1, 并将全局变量 errno 设置为:

ENOENT 路径名或文件名没有找到

**可移植性** 适用于 UNIX 系统。

**参 见** getcurdir, getcwd, getdisk, mkdir, rmdir, setdisk, system

**示 例** #include <stdio.h>

#include <stdlib.h>

#include <dir.h>

char old\_dir[MAXDIR];

char new\_dir[MAXDIR];

int main(void)

{

if (getcurdir(0, old\_dir))

{

perror("getcurdir()");

exit(1);

}

printf("Current directory is: \"%s\"\n", old\_dir);

```

 if (chdir("\\\\"))
 {
 perror("chdir()");
 exit(1);
 }
 if (getcurdir(0, new_dir))
 {
 perror("getcurdir()");
 exit(1);
 }
 printf("Current directory is now: \\%s\\n", new_dir);
 printf("\\nChanging back to original directory: \\%s\\n", old_dir);
 if (chdir(old_dir))
 {
 perror("chdir()");
 exit(1);
 }
 return 0;
}

```

## ■ chmod 改变文件的存取权限 ■

**用 法** #include <dos.h>

#include <io.h>

int \_chmod(const char \*path, int func[, int attrib]);

**原 型 在** io.h

**说 明** \_chmod 可以用来读取或设置 DOS 文件属性, 如果 func 值 0, \_chmod 函数返回文件的当前 DOS 属性; 若 func 值为 1, 则 \_chmod 函数把文件属性置为指定的 attrib 值。attrib 可以为下列符号常量之一(这些符号常数在 dos.h 中定义):

|           |      |
|-----------|------|
| FA_RDONLY | 只读   |
| FA_HIDDEN | 隐藏文件 |
| FA_SYSTEM | 系统文件 |
| FA_LABEL  | 卷标   |
| FA_DIREC  | 目录   |
| FA_ARCH   | 档案   |

**返 回 值** 调用成功时, \_comod 返回文件的属性字节, 否则返回 -1。

若调用出错, errno 被置成下列值之一:

|        |          |
|--------|----------|
| ENOENT | 文件或路径没找到 |
| EACCES | 非法访问     |

**可移植性** 只适用于 DOS。

**参 见** chmod, \_creat

**示 例** #include <errno.h>

```

#include <stdio.h>
#include <dos.h>
#include <io.h>
int get_file_attr(char * filename);
int main(void)
{
 char filename[128];
 int attrib;
 printf("Enter a filename:");
 scanf("%s", filename);
 attrib = get_file_attr(filename);
 if (attrib == -1)
 switch(errno)
 {
 case ENOENT : printf("Path or file not found. \n");
 break;
 case EACCES : printf("Permission denied. \n");
 break;
 default: printf("Error number, %d", errno);
 break;
 }
 else
 {
 if (attrib & FA_RDONLY)
 printf("%s is read-only. \n", filename);
 if (attrib & FA_HIDDEN)
 printf("%s is hidden. \n", filename);
 if (attrib & FA_SYSTEM)
 printf("%s is a system file. \n", filename);
 if (attrib & FA_LABEL)
 printf("%s is a volume label. \n", filename);
 if (attrib & FA_DIR)
 printf("%s is a directory. \n", filename);
 if (attrib & FA_ARCH)
 printf("%s is an archive file. \n", filename);
 }
 return 0;
}

/* returns the attributes of a DOS file */
int get_file_attr(char * filename)
{
 return(_chmod(filename, 0));
}

```

## ■ chmod 改变文件存取权限

用 法 #include <sys\stat.h>

int chmod(const char \* path, int amode);

原型在 io.h

说 明 chmod 函数将根据 amode 中的值, 设置由 path 所指定文件的存取权限, path 指向一个串, \* path 为该串的第一个字符。参数 amode 可以包含一个或两个符号常量 S\_IWRITE 和 S\_IREAD(在 sys\stat.h 中定义):

|                    |      |
|--------------------|------|
| amode 值            | 存取权限 |
| S_IWRITE           | 允许写  |
| S_IREAD            | 允许读  |
| S_IREAD   S_IWRITE | 可读可写 |

返回 值 调用成功时, chmod 返回 0, 否则返回 -1。

若调用出错, errno 被置成下列值之一:

|        |          |
|--------|----------|
| ENOENT | 文件或路径没找到 |
| EACCES | 无此权限     |

可移植性 对 UNIX 系统也适用。

参 见 acces, \_chmod, fstat, open, sopen, stat

示 例 #include <sys\stat.h>

#include <stdio.h>

#include <io.h>

void make\_read\_only(char \* filename);

int main(void)

{

make\_read\_only("NOTEXIST.FIL");

make\_read\_only("MYFILE.FIL");

return 0;

}

void make\_read\_only(char \* filename)

{

int stat;

stat = chmod(filename, S\_IREAD);

if (stat)

printf("Couldn't make %s read-only\n", filename);

else

printf("Made %s read-only\n", filename);

}

## ■ chsize 修改文件长度

用 法 #include <io.h>

int chsize(int handle, long size);

原型在 io.h

说明 本函数修改同参数 handle 相联文件的大小,它可以根据 size 的值同原文件长度进行比较,从而减小或增大文件的长度。该文件的打开模式必须允许进行写操作。

如果 chsize 函数要扩展一个文件,则仅需在原文件的结尾添加空字符(\0),如果是对源文件进行截断,则在新的文件结束标志之后的所有数据均被废弃。

返回值 若函数调用成功,则 chsize 返回 0,否则返回 -1,并将全局变量 errno 设置为下列值之一:

|         |             |
|---------|-------------|
| EACCESS | 不允许存取       |
| EBADF   | 非法文件号       |
| ENOSPC  | UNIX,不是 DOS |

可移植性 只适用于 DOS.

参见 close, \_creat, creat, open

示例 #include <string.h>

#include <fcntl.h>

#include <io.h>

int main(void)

{

int handle;

char buf[11] = "0123456789";

/\* create text file containing 10 bytes \*/

handle = open("DUMMY.FIL", O\_CREAT);

write(handle, buf, strlen(buf));

/\* truncate the file to 5 bytes in size \*/

chsize(handle, 5);

/\* close the file \*/

close(handle);

return 0;

}

## ■ circle 画圆

用法 #include <graphics.h>

void far circle(int x,int y,int radius);

原型在 graphics.h

说明 函数 circle 以点(x,y)为圆心,以 radius 为半径,用当前颜色画一个圆。

注意:线型参数(linestyle)对弧、圆、椭圆和饼图无影响,此时宽度参数(thickness)起作用。

返回值 无。

可移植性 只适用于 Turbo C,并且只能在配备了图形适配器的 IBMPC 及其兼容机上运行。

参 见 arc, ellipse, fillellipse, getaspetratio, sector, setaspetratio

```

示 例 #include <stdio.h>
 #include <conio.h>,
 int main(void)
 {
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int midx, midy;
 int radius = 100;
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error, %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
 }
 midx = getmaxx() / 2;
 midy = getmaxy() / 2;
 setcolor(getmaxcolor());
 /* draw the circle */
 circle(midx, midy, radius);
 /* clean up */
 getch();
 closegraph();
 return 0;
 }

```

## ■ clear87 清除浮点状态字 ■

用 法 #include <float.h>  
 unsigned int \_clear87(void);

原 型 在 float.h

说 明 \_clear87 清除浮点状态字, 状态字由 80×87 状态字和由 80×87 异常处理程序检测到的其它条件组合而成。

返 回 值 返回值的各数据位表明了未清除前的浮点状态, 有关该状态字的详细定义请参阅 float.h 中的常量定义。

可移植性 只适用于 DOS。

参 见 control87, fpreset, status87

示 例

```
#include <stdio.h>
#include <float.h>
int main(void)
{
 float x;
 double y = 1.5e-100;
 printf("\nStatus 87 before error: %X\n", _status87());
 x = y; /* create underflow and precision loss */
 printf("Status 87 after error: %X\n", _status87());
 _clear87();
 printf("Status 87 after clear: %X\n", _status87());
 y = x;
 return 0;
}
```

## ■ cleardevice 清图形屏幕 ■

用 法

```
#include <graphics.h>
void far cleardevice(void);
```

原 型 在 graphics.h

说 明 清除整个图形屏幕(即用当前背景色填充整个屏幕),并将当前图形输出位置移到原点(0,0)。

返 回 值 无。

可移植性 只适用于 Turbo C,并且只能在配有图形适配器的 IBMPC 及其兼容机上运行。

参 见 clearviewport

示 例

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int midx, midy;
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 }
}
```



```

 exit(1); /* terminate with an error code */
 }
 midx = getmaxx() / 2;
 midy = getmaxy() / 2;
 setcolor(getmaxcolor());
 /* for centering screen messages */
 setttextjustify(CENTER_TEXT, CENTER_TEXT);
 /* output a message to the screen */
 outtextxy(midx, midy, "press any key to clear the screen.");
 /* wait for a key */
 getch();
 /* clear the screen */
 cleardevice();
 /* output another message */
 outtextxy(midx, midy, "press any key to quit.");
 /* clean up */
 getch();
 closegraph();
 return 0;
}

```

## ■ clearerr 复位错误标志

用 法 #include <stdio.h>

void clearerr(FILE \* stream);

原 型 在 stdio.h

说 明 将指定流上的错误和文件结束标志置为 0。一旦该标志被置位为 1, 后续流操作将始终返回错误状态, 直到调用了 clearerr 或 rewind 函数。每次与某一流有关的输入操作均复位这个流文件的结束标志。

返 回 值 无。

可移植性 适用于 UNIX 系统, 并且在 ANSI C 中也有定义。

参 见 eof, feof, ferror, perror, rewind

示 例 #include <stdio.h>

```

int main(void)
{
 FILE * fp;
 char ch;
 /* open a file for writing */
 fp = fopen("DUMMY.FIL", "w");
 /* force an error condition by attempting to read */
 ch = fgetc(fp);
 printf("%c\n", ch);
 if (ferror(fp))

```

```

 {
 /* display an error message */
 printf("Error reading from DUMMY.FIL\n");
 /* reset the error and EOF indicators */
 clearerr(fp);
 }
 fclose(fp);
 return 0;
}

```

## ■ clearviewport 清除当前图形窗口 ■

**用 法** #include <graphics.h>

void far clearviewport(void);

**原 型 在** graphics.h

**说 明** 清除当前图形窗口,并将图形当前输出位置(CP)移到(0,0)点(相对于当前图形窗口)。

**返 回 值** 无。

**可移植性** 适用于 Turbo C,并且只能在配有图形适配器的 IBM PC 及其兼容机上运行。

**参 见** cleardevice, getviewsettings, setviewport

**示 例** #include <graphics.h>

#include <stdlib.h>

#include <stdio.h>

#include <conio.h>

#define CLIP\_ON 1 /\* activates clipping in  
viewport \*/

int main(void)

{

/\* request auto detection \*/

int gdriver = DETECT, gmode, errorcode;

int ht;

/\* initialize graphics and local variables \*/

initgraph(&gdriver, &gmode, "");

/\* read result of initialization \*/

errorcode = graphresult();

if (errorcode != grOk) /\* an error occurred \*/

{

printf("Graphics error: %s\n", grapherrormsg(errorcode));

printf("Press any key to halt:");

getch();

exit(1); /\* terminate with an error code \*/

}

setcolor(getmaxcolor());

```

 ht = textheight("W");
 /* message in default full-screen viewport */
 outtextxy(0, 0, " * <--- (0, 0) in default viewport");
 /* create a smaller viewport */
 setviewport(50, 50, getmaxx()-50, getmaxy()-50, CLIP_ON);
 /* display some messages */
 outtextxy(0, 0, " * <--- (0, 0) in smaller viewport");
 outtextxy(0, 2 * ht, "Press any key to clear viewport.");
 /* wait for a key */
 getch();
 /* clear the viewport */
 clearviewport();
 /* output another message */
 outtextxy(0, 0, "Press any key to quit.");
 /* clean up */
 getch();
 closegraph();
 return 0;
}

```

## ■ clock 测定运行时间

**用 法** #include <time.h>  
clock\_t clock(void);

**原 型 在** time.h

**说 明** 本函数可以用来测定两个操作之间的时间间隔, clock 的返回值除以 CLK\_TCK 即为秒值。

**返 回 值** clock 返回程序开始运行后所经过的运行时间。如果运行时间无效或其值无法表示, 则返回-1。

**可移植性** clock 函数与 ANSI C 中的定义兼容。

**参 见** time

**示 例**

```

#include <time.h>
#include <stdio.h>
#include <dos.h>
int main(void)
{
 clock_t start, end;
 start = clock();
 delay(2000);
 end = clock();
 printf("The time was: %f\n", (end - start) / CLK_TCK);
 return 0;
}

```

## close 关闭文件

用 法 #include <io.h>

int \_close(int handle)

原 型 在 io.h

说 明 \_close 关闭由文件句柄 handle 所指向的文件, handle 是调用 \_creat、creat、creatnew、creattemp、dup、dup2、\_open 或 open 时得到的一个文件句柄。

注意:该函数并不在文件末尾写一个 Ctrl-Z 字符,如果想用字符 Ctrl-Z 结束文件,必须显式地给出该字符。

返 回 值 若调用成功,返回 0,否则返回-1。

如果 handle 是一个不正确的文件句柄,则函数调用失败并将全局变量 errno 置为:

EBADF 非法文件号。

可移植性 只适用于 DOS。

参 见 close, \_creat, open, read, write

示 例 #include <string.h>

#include <stdio.h>

#include <fcntl.h>

#include <io.h>

main()

{

int handle;

char buf[11] = "0123456789";

/\* create a file containing 10 bytes \*/

handle = open("NEW.FIL", O\_CREAT);

if (handle > -1)

{

write(handle, buf, strlen(buf));

/\* close the file \*/

close(handle);

}

else

{

printf("Error opening file\n");

}

return 0;

}

## close 关闭文件

用 法 #include <io.h>

int close(int handle);

原型 在 io.h

说明 关闭由文件句柄 handle 所指向的文件, handle 是调用 \_creat、creat、creatnew、createmp、dup、dup2; \_open 或 open 时得到的一个文件句柄。

注意: 该函数并不在文件末尾写一个 Ctrl-Z 字符, 如果想用字符 Ctrl-Z 结束文件, 必须显式地给出该字符。

返回值 若调用成功, 返回 0, 否则返回 -1。

如果 handle 是一个不正确的文件句柄, 则函数调用失败并将全局变量 errno 设置为:

EBADF 非法文件号。

可移植性 对 UNIX 系统也适用。

示例

```
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
main()
{
 int handle;
 char buf[11] = "0123456789";
 /* create a file containing 10 bytes */
 handle = open("NEW.FIL", O_CREAT);
 if (handle > -1)
 {
 write(handle, buf, strlen(buf));
 /* close the file */
 close(handle);
 }
 else
 {
 printf("Error opening file\n");
 }
 return 0;
}
```

## closegraph 关闭图形系统

用法 #include <graphics.h>  
void far closegraph(void);

原型 在 graphics.h

说明 本函数释放由图形系统分配的所有内存, 然后将屏幕恢复为调用 initgraph 之前的模式(图形系统通过调用 \_graphfreemem 来释放分配的内存, 包括图形系统驱动程序、字体和内部缓冲区所占用的内存)。

返回值 无。

**可移植性** 只适用于 Turbo C, 并且只能在配有图形适配器的 IBMPC 及其兼容机上运行。

**参 见** `intgraph`, `setgraphbufsize`

**示 例**

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int x, y;
 /* initialize graphics mode */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
 }
 x = getmaxx() / 2;
 y = getmaxy() / 2;
 /* output a message */
 settxtjust(CENTER_TEXT, CENTER_TEXT);
 outtextxy(x, y, "Press a key to close the graphics system.");
 /* wait for a key */
 getch();
 /* closes down the graphics system */
 closegraph();
 printf("We're now back in text mode.\n");
 printf("Press any key to halt:");
 getch();
 return 0;
}
```

## ■ `clrEOF` 清除从当前光标位置到行尾的字符 ■

**用 法** `#include <conio.h>`  
`void clrEOF(void);`

**原 型 在** `conio.h`

**说 明** `clrEOF` 函数将在当前文本窗口中清除光标位置到行末的所有字符, 但并不移动光

标。

返回值 无。

可移植性 只适用于 IBM PC 及其兼容机。

参见 clrscr, delinea, window

示例 #include <conio.h>

```
int main(void)
{
 clrscr();
 printf("The function CLREOL clears all characters from the\r\n");
 printf("cursor position to the end of the line within the\r\n");
 printf("current text window, without moving the cursor. \r\n");
 printf("Press any key to continue . . .");
 gotoxy(14, 4);
 getch();
 clreol();
 getch();
 return 0;
}
```

## ■ clrscr 清除文本窗口, 并把光标放在左上角

用法 #include <conio.h>

void clrscr(void);

原型在 conio.h

说明 本函数清除当前文本窗口, 并将光标移到左上角处[位置为(1,1)]。

返回值 无。

可移植性 只适用于 IBM PC 及其兼容机。

参见 clreof, define, window

示例 #include <conio.h>

```
int main(void)
{
 int i;
 clrscr();
 for (i = 0; i < 20; i++)
 printf(" %d\r\n", i);
 printf("\r\nPress any key to clear screen");
 getch();
 clrscr();
 printf("The screen has been cleared!");
 getch();
 return 0;
}
```

## control87 处理浮点控制字

**用 法** `#include <float.h>`

`insigned int _control87(unsigned int newcw, unsigned int mask);`

**原 型 在** `float.h`

**说 明** 本函数用于读取或改变浮点控制字。浮点控制字是一个 `unsigned int` 类型值,它的每一位均指明了浮点包中的某种模式,即精度、无穷大的舍入模式。对这些模式进行处理可以屏蔽或打开浮点异常处理。

`_control87` 把 `mask` 中的每一位与 `newcw` 中的对应位相匹配,如果参数 `mask` 中的某一位等于 1,则 `newcw` 中的对应位就是浮点控制字中同一位的新值,`_control87` 将把浮点控制字中的那一位变成参数 `newcw` 中所对应的新值。

下面是一个能反映出该函数工作过程示例:

|        |      |      |      |      |
|--------|------|------|------|------|
| 原浮点控制字 | 0100 | 0011 | 0110 | 0011 |
| mask   | 1000 | 0001 | 0100 | 1111 |
| newcw  | 1110 | 1001 | 0000 | 0101 |
| 改变位    | 1xxx | xxxi | x0xx | 0101 |

如果 `mask=0`,则 `_control87` 公返回原浮点控制字,并且对其不作任何修改。

`_control87` 并不改变非正常位,因为 Turbo C 使用了非正常异常处理。

**返 回 值** 返回值的位状态反映了新的浮点控制字的状态。有关 `_control87` 返回值各个数据位的完整定义,请参见关于 `float.h` 的说明。

**可移植性** 只适用于 DOS。

**参 见** `_clear87`, `_fpreset`, `signal`, `_status87`

**示 例**

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

main()
{
 int handle;
 char buf[11] = "0123456789";
 /* create a file containing 10 bytes */
 handle = open("NEW.FIL", O_CREAT);
 if (handle > -1)
 {
 write(handle, buf, strlen(buf));
 /* close the file */
 close(handle);
 }
 else
 {

```



```

 printf("Error opening file\n");
 }
 return 0;
}

```

## ■ coreleft 返回尚未使用的内存(RAM)大小 ■

**用 法** 在 tiny、small 和 medium 模式下:

```
#include <alloc.h>
```

```
unsigned coreleft(void);
```

在 compact、larg 和 huge 模式下:

```
#include <alloc.h>
```

```
unsigned long coreleft(void);
```

**原 型** 在 alloc.h

**说 明** 本函数返回尚未使用内存的大小。根据所使用的存储模式是小模式还是大模式,返回的值并不相同。

**返 回 值** 在小模式下,coreleft 返回堆顶和栈顶之间未用空间的大小;在大模式下,它返回已分配的最高块和可用内存末端之间的未用内存的大小。

**可移植性** 只适用于 DOS。

**参 见** allocmem, brk, farcoreleft, malloc

**示 例**

```

#include <stdio.h>
#include <alloc.h>

int main(void)
{
 printf("The difference between the highest allocated block and\n");
 printf("the top of the heap is: %lu bytes\n", (unsigned long) coreleft());
 return 0;
}

```

## ■ cos 计算余弦值 ■

**用 法** 对于实数

```
#include <math.h>
```

```
double cos(double x);
```

对于复数

```
#include <complex.h>
```

```
complex cos(complex x);
```

**原 型** 在 math.h

对于复数

complex.h

**说 明** 本函数计算输入值的余弦值,角度以弧度为单位给出。  
复数的余弦定义为:

$$\cos(z) = (\exp(i * z) + \exp(-i * z)) / 2$$

**返 回 值** 对于一个实型参数,cos 返回一个在 -1 到 1 之间的值。此函数的错误处理程序可以通过 matherr 函数进行修改。

**可移植性** 实数类型的 COS 函数对 UNIX 系统也适用,并且在 ANSI C 中也有定义。此函



```
struct country * country(int xcode, struct country * cp);
```

原型在 dos.h

说明 本函数用来指定一些信息(如日期、时间和货币)的格式,这些信息的格式因国家不同而不同。由此函数设置的值依赖于所用的 DOS 版本。

如果 cp = -1, 则 xcode 必须为一代表当前国家的非零值;否则由 cp 指向的 country 结构将包含当前国家(当 xcode = 0 时)或由 xcode 所指定国家(当 xcode 不为 0 时)的信息。

结构 country 定义如下:

```
struct country{
 int co_data; /* 日期格式 */
 char co_curr[5]; /* 货币符号 */
 char co_thsep[2]; /* 千位分隔符 */
 char co_dseep[2]; /* 十进位分隔符 */
 char co_dtsep[2]; /* 日期分隔符 */
 char co_tmsep[2]; /* 时间分隔符 */
 char co_currstyle; /* 货币符号 */
 char co_digits; /* 货币中的有效数字 */
 char co_time; /* 时间格式 */
 char co_case; /* case 图 */
 char co_dasep[2]; /* 数据分隔符 */
 char co_fill[10]; /* 填充位 */
}
```

co\_date 的日期格式如下:

- 0 表示美国风格的日期格式: 月/日/年
- 1 表示欧洲风格的日期格式: 日/月/年
- 2 表示日本风格的日期格式: 年/月/日

由 co\_currstyle 给定的货币显示格式如下:

- 0 表示货币符号在数值之前,且符号和数值之间无空格
- 1 表示货币符号在数值之后,且符号和数值之间无空格
- 2 表示货币符号在数值之前,且符号和数值之间有一空格
- 3 表示货币符号在数值之后,且符号和数值之间有一空格

返回值 若调用成功,返回指针参数 cp,若出错则回空指针 NULL。

可移植性 只适用于 DOS 3.0 及更高版本。

## ■ cprintf 格式化并输出数据至屏幕 ■

用法 #include <conio.h>

```
int cprintf(const char * format [,argument,...]);
```

原型在 conio.h

说明 cprintf 函数能够接收到一系列参数,并根据由 format 指向的格式串中相应的格

式指示符,直接向当前文本窗口输出格式化后的数据。参数的个数必须与格式指示符的个数相符。

根据全局变量 `directvideo` 值的不同,数据将直接写入视频内存中或调用 BIOS 来完成。

与 `fprintf` 和 `printf` 不同,`cprintf` 并不将 `\n` 解释成 `\r\n`。

**返回值** 返回输出的字符数。

**可移植性** 只适用于 IBM PC 及其兼容机。

**参 见** `directvideo`(全局变量),`fprintf`,`printf`,`putch`,`sprintf`,`vprintf`

**示 例**

```
#include <conio.h>

int main(void)
{
 /* clear the screen */
 clrscr();
 /* create a text window */
 window(10, 10, 80, 25);
 /* output some text in the window */
 cprintf("Hello world\r\n");
 /* wait for a key */
 getch();
 return 0;
}
```

## ■ `cputs` 输出一字符串至屏幕 ■

**用 法** `#include <conio.h>`  
`int cputs(const char *str);`

**原 型 在** `conio.h`

**说 明** `cput` 将把一个以空字符终结的串 `str` 输出到当前文本窗口中,该函数并不在字符串后面添加一个换行符。

根据全局变量 `directvideo` 值的不同,字符串将直接写入视频内存中或调用 BIOS 来完成。

与 `puts` 不同,`cputs` 不将 `\n` 解释成 `\r\n`。

**返回值** 返回输出至屏幕的最后一个字符。

**可移植性** 只适用于 IBM PC 及其兼容机。

**参 见** `cgets`,`directvideo`(全局变量),`fputs`,`putch`,`puts`

**示 例**

```
#include <conio.h>

int main(void)
{
 /* clear the screen */
 clrscr();
 /* create a text window */
```

```

window(10, 10, 80, 25);
/* output some text in the window */
cputs("This is within the window\r\n");
/* wait for a key */
getch();
return 0;
}

```

## ■ creat 创建一个新文件或重写一个已存在的文件 ■

**用 法** #include <dos.h>

int \_creat(const char \*path, int attrib);

**原 型 在** io.h

**说 明** \_creat 接受一个 DOS 属性字 attrib, 该文件总是以二进制方式打开, 如果创建成功, 则文件指针位于文件开头。此文件打开后, 既可读也可写。

如果文件已经存在, 则将该文件的长度置为 0 (这与先删除该文件、然后再创建一个同名文件基本上是一样的)。

\_creat 函数参数 attrib 可以是以下常数之一 (这些常数在 dos.h 中定义):

FA\_RDONLY 只读文件

FA\_HIDDEN 隐含文件

FA\_SYSTEM 系统文件

**返 回 值** 若调用成功, 则函数返回一个表示新文件句柄的为一非负整数, 否则返回 -1。

在发生错误时 errno 被置为下列值之一:

ENOENT 路径或文件名没找到

EMFILE 打开的文件太多

EACCES 无此权限

**可移植性** 只适用于 DOS。

**参 见** \_chmod, chsize, \_close, close, creat, creatnew, creattemp

**示 例** #include <sys\stat.h>

#include <string.h>

#include <fcntl.h>

#include <io.h>

int main(void)

{

int handle;

char buf[11] = "0123456789";

/\* change the default file mode from text to binary \*/

\_fmode = O\_BINARY;

/\* create a binary file for reading and writing \*/

handle = creat("DUMMY.FIL", S\_IREAD | S\_IWRITE);

/\* write 10 bytes to the file \*/

write(handle, buf, strlen(buf));

```

 /* close the file */
 close(handle);
 return 0;
}

```

## ■ creat 创建一个新文件或重写一个已存在的文件 ■

**用 法** #include <sys/stat.h>

int creat(const char \*path, int amode);

**原 型 在** io.h

**说 明** creat 将创建一个新文件或准备重写由 path 所指定的已存在文件, amode 只对新创建的文件起作用。

调用 creat 函数所创建的文件读取方式, 总是按全局变量是 \_fmode(O\_TEXT 或 O\_BINARY) 指定。

如果文件已存在且写属性已置位, creat 函数将该文件长度修改成 0 字节, 文件的属性不变。若已存在文件是只读的, 则 creat 调用失败, 则对原文件不会有任何变化。

调用 creat 函数时只检查存取模式字 amode 中的 S\_IWRITE 位。如果该位为 1, 文件是可写的; 如果该位为 0, 则文件是只读的。所有其他 DOS 属性位置为 0。

|                    |      |
|--------------------|------|
| amode 值            | 存取权限 |
| S_IWRITE           | 允许写  |
| S_IREAD            | 允许读  |
| S_IWRITE   S_IREAD | 可读可写 |

注意: 在 DOS 中, 文件可写权限隐含文件可读权限。

**返 回 值** 若调用成功, 则函数返回一个表示新文件句柄的非负整数, 否则返回 -1。

在发生错误时, 将全局变量 errno 置为下列值之一:

|        |           |
|--------|-----------|
| ENOENT | 路径或文件名没找到 |
| EMFILE | 打开文件太多    |
| EACCES | 无此权限      |

**可移植性** 对于 UNIX 系统也适用。

**参 见** chmod, chsize, close, \_creat, creatnew, creattemp, dup, dup2, \_fmode(全局变量), fopen, open, sopen, write

**示 例** #include <sys/stat.h>

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

```
#include <fcntl.h>
```

```
#include <io.h>
```

```
int main(void)
```

```
{
```

```
 int handle;
```

```
 char buf[11] = "0123456789";
```

```
 /* change the default file mode from text to binary */
```

```

 _fmode = O_BINARY;
 /* create a binary file for reading and writing */
 handle = creat("DUMMY.FIL", S_IREAD | S_IWRITE);
 /* write 10 bytes to the file */
 write(handle, buf, strlen(buf));
 /* close the file */
 close(handle);
 return 0;
}

```

## ■ creatnew 创建新文件

**用 法** #include <dos.h>

int creatnew(const char \*path, int mode);

**原 型** 在 io.h

**说 明** creatnew 函数与 \_creat 相似,只是当 path 所指定文件已存在时,creatnew 函数将返回一个错误,同时原文件保持不变。

参数 mode 可以是下列常数之一(这些常数在 dos.h 中定义):

FA\_RDONLY      只读文件  
FA\_HIDDEN      隐含文件  
FA\_SYSTEM      系统文件

**返 回 值** 若调用成功,则函数返回新的文件句柄,是一个非负整数,否则返回-1。

若出现错误,则全局变量 errno 将置为下列值之一:

EEXIST          文件已存在  
ENOENT          路径或文件名没找到  
EMFILE          打开的文件太多  
EACCES          无此权限

**可移植性** creatnew 只用于 DOS 3.0 及更高版本。

**参 见** close, \_creat, creat, creattemp, dup, \_fmode(全局变量), open

**示 例** #include <string.h>

#include <stdio.h>

#include <errno.h>

#include <dos.h>

#include <io.h>

int main(void)

```

{
 int handle;
 char buf[11] = "0123456789";
 /* attempt to create a file that doesn't already exist */
 handle = creatnew("DUMMY.FIL", 0);
 if (handle == -1)
 printf("DUMMY.FIL already exists.\n");
}

```

```

 else
 {
 printf("DUMMY.FIL successfully created. \n");
 write(handle, buf, strlen(buf));
 close(handle);
 }
 return 0;
}

```

## ■ createmp 创建一个文件名唯一的文件 ■

**用 法** #include <dos.h>

int createmp(char \*path, int attrib);

**原 型** 在 io.h

**说 明** 调用 createmp 创建的文件总是置成全程变量 \_fmode(O\_TEXT 或 O\_BINARY) 指定的方式。

path 是一个以反斜杠(\)结束的路径名。新创建文件的名称是唯一的, 它不与 path 所给出的目录中已存在的任何文件名重名, 该文件名将存放在字符串 path 中, 因此 path 必须足够长, 以便能够容纳结果文件名。在程序结束时, 此文件并不会自动删除。

createmp 接受一个 DOS 属性字 amode, 文件总是以二进制方式打开。若文件创建成功, 则文件指针指向新文件起始位置, 此文件打开后既可读也可写。

amode 参数可以为下列常数值之一(在 dos.h 中定义):

|           |      |
|-----------|------|
| FA_RDONLY | 只读文件 |
| FA_HIDDEN | 隐含文件 |
| FA_SYSTEM | 系统文件 |

**返 回 值** 若调用成功, 则函数返回一个表示新的文件句柄的非负整数, 否则返回 -1。若出现错误, 则全局变量 errno 将置为下列值之一:

|        |           |
|--------|-----------|
| ENOENT | 路径或文件名没找到 |
| EMFILE | 打开文件太多    |
| EACCES | 无此权限      |

**可移植性** 只适用于 DOS 3.0 及更高版本。

**参 见** close, \_creat, creat, creatnew, dup, \_fmode(全局变量), open

**示 例**

```

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

int main(void)
{
 int handle;
 char pathname[128];
 strcpy(pathname, "\\");
}

```



```

/* create a unique file in the root directory */
handle = createmp(pathname, 0);
printf("%s was the unique file created.\n", pathname);
close(handle);
return 0;
}

```

## ■ cscanf 从控制台执行格式化输入

**用 法** #include <conio.h>

```
int cscanf(char *format [,address...]);
```

**原 型** 在 conio.h.

**说 明** cscanf 直接从控制台扫描一系列输入数据,每次读入一个字符;然后根据格式串 format 中的指示符格式化每个输入数据;最后,cscanf 将格式化后的输入数据存入紧跟格式字符串 format 之后的地址参数中,并将输入数据显示在屏幕上。格式指示符、地址参数的个数和输入字段的个数必须相等。

由于多种原因,cscanf 在遇到正常数据结束符(空白字符)之前,可能会停止在某些特定位置或完全终止。请参见 scanf 中关于具体可能原因的讨论。

**返 回 值** 返回成功地进行了读入、转换和保存的输入数据个数。仅仅被读入但没有保存的数据不算在内。如果没有存入任何数据,则 cscanf 函数返回 0。

**可移植性** 只适用于 DOS。

**参 见** fscanf, getche, scanf, sscanf

**示 例** #include <conio.h>

```

int main(void)
{
 char string[80];
 /* clear the screen */
 clrscr();
 /* Prompt the user for input */
 cprintf("Enter a string with no spaces:");
 /* read the input */
 cscanf("%s", string);
 /* display what was read */
 cprintf("\r\nThe string entered is: %s", string);
 return 0;
}

```

## ■ ctime 把日期和时间转化为对应的字符串

**用 法** #include <time.h>

```
char *ctime(const time_t *time);
```

**原 型** 在 time.h

**说 明** ctime 函数将把参数 time 所指定的时间(如函数 time 的返回值)转化成以\n和\

0 结束的 ASCII 字符串,如下所示:

```
Mon Nov 21 11:31:54 1983\n\0
```

注意:所有的域都有固定的长度。

全局长整型变量 `timezone` 应为格林威治时间(GMT)和当地标准时间之间的差,单位为秒(在 PST 时区,全局变量 `timezone` 为  $8 \times 60 \times 60$ )。全局变量 `daylight` 为非 0 值,当且仅当是使用美国标准夏令时间进行转换。

**返回值** `ctime` 返回指向包含日期和时间字符串的指针,该字符串是静态的,在每次调用时都将重写。

**可移植性** 适用于 UNIX 系统,并且在 ANSI C 中也有定义。

**参见** `asctime`, `daylight` (全局变量), `difftime`, `ftime`, `getdate`, `gmtime`, `localtime`, `settime`, `time`, `timezone` (全局变量), `tzet`

**示例**

```
#include <stdio.h>
#include <time.h>
int main(void)
{
 time_t t;
 time(&t);
 printf("Today's date and time: %s\n", ctime(&t));
 return 0;
}
```

## ■ `ctrlbrk` 设置 `ctrl-break` 处理程序

**用法** `#include <dos.h>`  
`void ctrlbrk(int (* handler)(void));`

**原型在** `dos.h`

**说明** 本函数设置一个新的由参数 `handler` 所指向的 `control-break` 处理函数。在执行过程中,中断向量 `0x23` 将被修改以便使用这个给定的函数。

`ctrlbrk` 建立一个 DOS 中断处理程序来调用给定函数,该函数并不能被直接调用。

该处理函数可以用来执行多种操作和系统调用,它并不一定需要显式地返回,可以在处理程序中通过 `longjmp` 函数返回到调用它的程序中任何一点。若处理函数返回 0,则当前程序将终止;其它返回值则使程序恢复执行。

**返回值** 无。

**可移植性** 仅适用于 DOS。

**参见** `getbrk`, `signal`

**示例**

```
#include <stdio.h>
#include <dos.h>
#define ABORT 0
int c_break(void)
{
```

```

 printf("Control-Break pressed. Program aborting ... \n");
 return (ABORT);
}
int main(void)
{
 ctrlbrk(c_break);
 for(;;)
 {
 printf("Looping... Press <Ctrl-Break> to quit, \n");
 }
 return 0;
}

```

## ■ delay 暂停

用 法 #include <dos.h>

void delay(unsigned milliseconds);

原 型 在 dos.h

说 明 调用 delay 函数将暂停当前所执行程序, 参数 milliseconds 指明暂停的毫秒数, 用户没有必要在使用该函数之前做校准延迟精度的工作, delay 函数只能精确到毫秒。

返 回 值 无。

可移植性 仅适用于 IBM PC 及其兼容机。

参 见 nosound, sleep, sound

示 例 /\* Emits a 440-Hz tone for 500 milliseconds \*/

```

#include <dos.h>
int main(void)
{
 sound(440);
 delay(500);
 nosound();
 return 0;
}

```

## ■ delline 在文本窗口中删去一行

用 法 #include <conio.h>

void delline(void);

原 型 在 conio.h

说 明 delline 删除光标所在的那一行, 把原来光标以下各行上移一行, 并在最底行增加一新行, 它是在当前文本窗口中操作。

返 回 值 无。

可移植性 只适用于 IBM PC 及其兼容机。

参 见 clreol, clrscr, insline, window

示 例 #include <conio.h>

```
int main(void)
{
 clrscr();
 printf("The function DELLINE deletes the line containing the\r\n");
 printf("cursor and moves all lines below it one line up. \r\n");
 printf("DELLINE operates within the currently active text\r\n");
 printf("window. Press any key to continue . . .");
 gotoxy(1,2); /* Move the cursor to the second line and first column */
 getch();
 delline();
 getch();
 return 0;
}
```

## ■ detectgraph 检测硬件并确定应使用何种图形驱动程序和图形模式 ■

用 法 #include <graphics.h>

void far detectgraph(int far \* graphdriver, int far \* graphmode);

原 型 在 graphics.h

说 明 detectgraph 检测系统的图形适配器,选择该适配器所能提供的最高分辨率的模式。如果没有检测到图形适配器,则 \* graphdriver 参数将置为 NotDetected(-2),且全局变量 graphresult 将置为 grNotDetected(-2)。

参数 \* graphdriver 为一返回的整型量,它指明所用的图形驱动程序。用户可以用枚举类型 graphics\_drivers 中的常量对它赋值。该常量文件在 graphics.h 中定义,列表如下:

| graphics_drivers<br>常量 | 数值       |
|------------------------|----------|
| DETECTED               | 0(需自动检测) |
| CGA                    | 1        |
| MCGA                   | 2        |
| EGA                    | 3        |
| EGA64                  | 4        |
| EGAMONO                | 5        |
| IBM8514                | 6        |
| HERCMONO               | 7        |
| ATT400                 | 8        |
| VGA                    | 9        |
| PC3270                 | 10       |

| 图形驱动程序   | graphics _modes | 值 | 列×行      | 调色板       | 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 | 620×200  | 2 color   | 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 color   | 1     |
|          | MCGAHI          | 5 | 640×480  | 2 color   | 1     |
| EGA      | EGALO           | 0 | 640×200  | 16 color  | 4     |
|          | EGAHI           | 1 | 640×350  | 16 color  | 2     |
| EGA64    | EGA64LO         | 0 | 640×200  | 16 color  | 1     |
|          | EGA64HI         | 1 | 640×350  | 4 color   | 1     |
| EGA-MONO | EGAMONHI        | 3 | 640×350  | 2 color   | 1 *   |
|          | EGAMONHI        | 3 | 640×350  | 2 color   | 2 * * |
| HERC     | HERCMONHI       | 0 | 720×348  | 2 color   | 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 color   | 1     |
|          | ATT400HI        | 5 | 640×400  | 2 color   | 1     |
| VGA      | VGALO           | 0 | 640×200  | 16 color  | 2     |
|          | VGAMED          | 1 | 640×350  | 16 color  | 2     |
|          | VGAHI           | 2 | 640×480  | 16 color  | 1     |
| PC3270   | PC3270HI        | 0 | 720×350  | 2 color   | 1     |
| IBM8514  | IBM8514HI       | 0 | 640×480  | 256 color |       |
|          | IBM8514LO       | 0 | 1024×768 | 256 color |       |

\* 64K EGAMONO 卡

\* \* 256K EGAMONO 卡

参数 `graphmode` 是一个整型量,用来说明初始图形模式(当 `graphdriver = DETECT` 时, \* `graphmode` 被置为检测到的驱动程序可用的最高分辨率)。

用户可以使用枚举类型 `graphics _modes` 中的常量对 \* `graphmode` 赋值,该常量在 `graphics.h` 中定义,表上页表。

注意:直接调用 `detectgraph` 的主要原因是为覆盖由 `detectgraph` 提供给 `initgraph` 的图形模式。

返回值 无。

**可移植性** 只适用于 Turbo C, 且仅适用于配备了图形适配器的 IBM PC 及兼容机。

**参 见** graphresult, initgraph

**示 例**

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

/* names of the various cards supported */
char * dname[] = { "requests detection",
 "a CGA",
 "an MCGA",
 "an EGA",
 "a 64K EGA",
 "a monochrome EGA",
 "an IBM 8514",
 "a Hercules monochrome",
 "an AT&T 6300 PC",
 "a VGA",
 "an IBM 3270 PC"
 };

int main(void)
{
 /* returns detected hardware info. */
 int gdriver, gmode, errorcode;
 /* detect graphics hardware available */
 detectgraph(&gdriver, &gmode);
 /* read result of detectgraph call */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error
 occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
 }
 /* display the information detected */
 clrscr();
 printf("You have %s video display card.\n", dname[gdriver]);
 printf("Press any key to halt:");
 getch();
 return 0;
}
```

## ■ difftime 计算两个时刻之间的时间差 ■

用 法 #include <time.h>

double difftime(time\_t time2, time\_t time1);

原 型 在 time.h

说 明 difftime 计算从 time1 到 time2 之间的时间,单位为秒。

返 回 值 difftime 返回计算结果的双精度数值。

可移植性 对 UNIX 系统也适用,并且在 ANSI C 中也有定义。

参 见 asctime, ctime, daylight(全局变量), gmtime,  
localtime, time, timezone(全局变量)

示 例 #include <time.h>  
#include <stdio.h>  
#include <dos.h>  
#include <conio.h>

int main(void)

{

time\_t first, second;

clrscr();

first = time(NULL); /\* Gets system time \*/

delay(2000); /\* Waits 2 secs \*/

second = time(NULL); /\* Gets system time again \*/

printf("The difference is, %i seconds\n", difftime(second, first));

getch();

return 0;

}

## ■ disable 屏蔽中断 ■

用 法 #include <dos.h>

void disable(void);

原 型 在 dos.h

说 明 disable 函数为程序员提供了灵活的硬件中断控制。

disable 是一个用来屏蔽中断的宏,执行该函数后只允许从外部设备来的不可屏蔽中断(NMI)。

返 回 值 无。

可移植性 该宏只适用于 80×86 结构。

参 见 enable, getvect

示 例 /\* NOTE: This is an interrupt service  
routine. You cannot compile this program  
with Test Stack Overflow turned on and  
get an executable file that operates

```

 correctly. */
#include <stdio.h>
#include <dos.h>
#include <conio.h>
#define INTR 0X1C /* The clock tick interrupt */
void interrupt (*oldhandler)(void);
int count=0;
void interrupt handler(void)
{
 /* disable interrupts during the handling of the interrupt */
 disable();
 /* increase the global counter */
 count++;
 /* reenale interrupts at the end of the handler */
 enable();
 /* call the old routine */
 oldhandler();
}
int main(void)
{
 /* save the old interrupt vector */
 oldhandler = getvect(INTR);
 /* install the new interrupt handler */
 setvect(INTR, handler);
 /* loop until the counter exceeds 20 */
 while (count < 20)
 printf("count is %d\n",count);
 /* reset the old interrupt handler */
 setvect(INTR, oldhandler);
 return 0;
}

```

## ■ div 将两个整数相除,返回商和余数 ■

用 法 #include <stdlib.h>

div\_t div(int number, int denom);

原 型 在 stdlib.h

说 明 div 将两个整数相除,并返回 div\_t 类型的商和余数。numer 和 denom 分别是被除数和除数。类型 div\_t 是一种整型数结构,在 stdlib.h 中使用 typedef 语句定义如下:

```

typedef struct {
 int quot; /* 商数 */
 int rem; /* 余数 */
}

```



```
 } div_t;
```

**返回值** div 返回成员一个包括 quot(商)和 rem(余数)的结构。

**可移植性** 与 ANSI C 兼容。

**参 见** ldiv

**示 例**

```
/* div example */
#include <stdlib.h>
#include <stdio.h>
div_t x;
int main(void)
{
 x = div(10,3);
 printf("10 div 3 = %d remainder %d\n",
 x.quot, x.rem);
 return 0;
}
```

## ■ dosxterr 获取扩展错误信息 ■

**用 法** #include<dos.h>

```
int dosxterr(struct DOSERROR *eblkp);
```

**原 型** 在 dos.h

**说 明** 在一次 DOS 调用失败后,本函数将扩展错误信息送至 eblkp 所指向的 DOSERROR 结构中。该结构定义如下:

```
struct DOSERROR {
 int de_exterror; /* extended error */
 char de_class; /* error class */
 char de_action; /* action */
 char de_locus; /* error locus */
}
```

此结构中的值通过 DOS 系统调用 Ox59 得到。de\_exterror 值为 0 表示前一次 DOS 调用未出错。

**返回值** dosxterr 返回值为 de\_exterror。

**可移植性** 仅适用于 DOS 3.0 或更高版本。

**示 例** #include <stdio.h>

```
#include <dos.h>
```

```
int main(void)
```

```
{
```

```
 FILE *fp;
```

```
 struct DOSERROR info;
```

```
 fp = fopen("perror.dat","r");
```

```
 if (!fp) perror("Unable to open file for reading");
```

```

 dosexterr(&info);
 printf("Extended DOS error information:\n");
 printf(" Extended error: %d\n",info.de_exterror);
 printf(" Class:%x\n",info.de_class);
 printf(" Action:%x\n",info.de_action);
 printf(" Error Locus:%x\n",info.de_locus);
 return 0;
}

```

## ■ dostounix 把日期和时间转换成 UNIX 格式

用 法 #include <dos.h>

long dostounix(struct date \*d,struct dostime \*t);

原 型 在 dos.h

说 明 dostounix 把从 getdate 和 gettime 返回的日期和时间转换为 UNIX 格式。参数 d 指向一个 date 结构,t 指向一个包含有效 DOS 日期和时间信息的 dostime 结构。

返 回 值 UNIX 方式的当前日期和时间参数;从格林威治时间(GMT)1970 年 1 月 1 日 00:00:00 到当前时间的秒数。

可移植性 dostounix 函数仅适用于 DOS。

参 见 unixtodos

示 例 #include <time.h>  
 #include <stddef.h>  
 #include <dos.h>  
 #include <stdio.h>

```

int main(void)
{
 time _t t;
 struct time d_time;
 struct date d_date;
 struct tm *local;
 getdate(&d_date);
 gettime(&d_time);
 t = dostounix(&d_date, &d_time);
 local = localtime(&t);
 printf("Time and Date: %s\n", asctime(local));
 return 0;
}

```

## ■ drawpoly 绘制多边形

用 法 #include <graphics.h>

void far drawpoly(int numpoints,int far polypoints);

原 型 在 graphics.h

**说 明** drawpoly 用当前的线型和颜色绘制一顶点数为 numpoints 的多边形。  
 \* polypoints 指向一个整数序列(共有 numpoints \* 2 个整数)。每一整数对给出多边形一个顶点的 x 和 y 坐标。

注意:为了绘制一个有 n 个顶点的封闭图形,必须将 n+1 个坐标点传给 drawpoly,其中第 n 个坐标与第 0 个坐标相同。

**返回值** 无。

**可移植性** 仅适用于 Turbo C,且仅适用于配有图形适配器的 IBMPC 及其兼容机。

**参 见** fillpoly, floofill, graphresult, setwritemode

**示 例**

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int maxx, maxy;
 /* our polygon array */
 int poly[10];
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk)
 /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt,");
 getch();
 /* terminate with an error code */
 exit(1);
 }
 maxx = getmaxx();
 maxy = getmaxy();
 poly[0] = 20; /* 1st vertex */
 poly[1] = maxy / 2;
 poly[2] = maxx - 20; /* 2nd */
 poly[3] = 20;
 poly[4] = maxx - 50; /* 3rd */
 poly[5] = maxy - 20;
 poly[6] = maxx / 2; /* 4th */
 poly[7] = maxy / 2;
```

```

/*
 drawpoly doesn't automatically close
 the polygon, so we close it.
*/
poly[8] = poly[0];
poly[9] = poly[1];
/* draw the polygon */
drawpoly(5, poly);
/* clean up */
getch();
closegraph();
return 0;
}

```

## dup 复制文件句柄

**用 法** `#include <io.h>`

`int dup(int handle);`

**说 明** dup 创建一个新的文件句柄, 该句柄与原文件句柄有以下相同之处:

- 相同文件或设备;
- 相同的文件指针(改变了其中一个的文件指针就改变了另一个);
- 相同的存取模式(读、写、读/写)。

handle 是调用 `_creat`、`creat`、`_open`、`open`、`dup` 或 `dup2` 得到的一个文件句柄。

**返 回 值** 若调用成功, dup 返回一个值为非负的新文件句柄; 否则返回 -1。在出现错误时, 全局变量 `errno` 将置为下列值之一:

`EMFILE` 打开文件太多  
`EBADF` 无效文件号

**可移植性** 对于所有的 UNIX 系统均适用

**参 见** `_close`、`close`、`_creat`、`creat`、`creatnew`、`creattemp`、`dup2`、`fopen`、`_open`、`open`

**示 例** `#include <string.h>`

`#include <stdio.h>`

`#include <conio.h>`

`#include <io.h>`

`void flush(FILE *stream);`

`int main(void)`

{

FILE \*fp;

char msg[] = "This is a test";

/\* create a file \*/

fp = fopen("DUMMY.FIL", "w");

/\* write some data to the file \*/

fwrite(msg, strlen(msg), 1, fp);

```

 clrscr();
 printf("Press any key to flush DUMMY.FIL,");
 getch();
 /* flush the data to DUMMY.FIL without closing it */
 flush(fp);
 printf("\nFile was flushed, Press any key to quit:");
 getch();
 return 0;
}

void flush(FILE * stream)
{
 int duphandle;
 /* flush TC's internal buffer */
 fflush(stream);
 /* make a duplicate file handle */
 duphandle = dup(fileno(stream));
 /* close the duplicate handle to flush the DOS buffer */
 close(duphandle);
}

```

## ■ dup2 将一个文件句柄(oldhandle) 复制到一个已有的文件句柄(newhandle)

用 法 #include <io.h>

int dup2(int oldhandle, int newhandle);

原 型 在 io.h

说 明 dup2 建立一个新的文件句柄,该句柄与原文件句柄有如下相同处:

- 相同的打开文件或设备;
- 相同的文件指针(改变了其中一个的文件指针就改变了另一个);
- 相同的存取模式(读、写、读/写)。

dup2 函数将建立一个值为 newhandle 的文件句柄。如果调用 dup2 函数时与参数 newhandle 相联的文件是打开的,则该文件被关闭。

newhandle 和 oldhandle 都是在调用 creat、open、dup 或 dup2 时得到的一个文件句柄。

返 回 值 若调用成功,返回 0;否则返回 -1。

如果出现错误,全局变量 errno 将置为下列值之一:

EMFILE      打开文件太多  
 EBADF      无效文件号

可移植性 dup2 适用于除系统 III 之外的一些 UNIX 系统。

参 见 close, close, creat, creat, creatnew, creattemo, dup, fopen, open, open

```

示 例 #include <sys\stat.h>
 #include <string.h>
 #include <fcntl.h>
 #include <io.h>
 int main(void)
 {
 #define STDOUT 1
 int nul, oldstdout;
 char msg[] = "This is a test";
 /* create a file */
 nul = open("DUMMY.FIL", O_CREAT | O_RDWR,
 S_IRREAD | S_IWRITE);
 /* create a duplicate handle for standard output */
 oldstdout = dup(STDOUT);
 /*
 redirect standard output to DUMMY.FIL
 by duplicating the file handle onto
 the file handle for standard output.
 */
 dup2(nul, STDOUT);
 /* close the handle for DUMMY.FIL */
 close(nul);
 /* will be redirected into DUMMY.FIL */
 write(STDOUT, msg, strlen(msg));
 /* restore original standard output handle */
 dup2(oldstdout, STDOUT);
 /* close duplicate handle for STDOUT */
 close(oldstdout);
 return 0;
 }

```

## ■ ecvt 把浮点数转换为字符串 ■

用 法 #include<stdlib.h>

```
char * ecvt(double value, int ndig, int *dec, int *sign);
```

原 型 在 stdlib.h

说 明 ecvt 从参数 value 的最左有效数字开始转换为 ndig 位数字且以空字符终结的字符串,并返回一个指向该字符串的指针。相对于串起始处的十进制小数点位置通过参数 dec 间接存储(dec 为负表示小数点在结果数字串的最左边)。字符串本身并不含小数点。若 value 值为负,则参数 sign 的返回值为非零;否则为 0。在字符串中的最低位是舍入位。

返 回 值 ecvt 的返回值为—指向数字字符串数据的静态指针,其内容在每次调用 ecvt 时均重写。

**可移植性** 适用于 UNIX。在 ANSI C 中没有 `ecvt`，建议用户不在需移植程序中使用该函数。

**参 见** `fcvt, gcvt, sprintf`

**示 例**

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

int main(void)
{
 char *string;
 double value;
 int dec, sign;
 int ndig = 10;
 clrscr();
 value = 9.876;
 string = ecvt(value, ndig, &dec, &sign);
 printf("string = %s dec = %d sign = %d\n", string, dec, sign);
 value = -123.45;
 ndig = 15;
 string = ecvt(value, ndig, &dec, &sign);
 printf("string = %s dec = %d sign = %d\n", string, dec, sign);
 value = 0.6789e5; /* scientific notation */
 ndig = 5;
 string = ecvt(value, ndig, &dec, &sign);
 printf("string = %s dec = %d sign = %d\n", string, dec, sign);
 return 0;
}
```

## ■ ellipse 绘制椭圆

**用 法** `#include <graphics.h>`  
`void far ellipse(int x, int y, int stngle, int endngle,`  
`int xradius, int yradius);`

**原 型 在** `graphics.h`

**说 明** `ellipse` 函数用当前颜色并以  $(x, y)$  为中心画一椭圆，其水平轴和垂直轴分别由参数 `xradius` 和 `yradius` 给出。椭圆从起始角 `stangle` 画到 `endangle`。如果 `stangle = 0` 且 `endangle = 360`，则 `ellipse` 函数将画出一完整的椭圆。

注意：参数 `linestyle` 并不影响圆弧、圆、椭圆、扇形，只用到了参数 `thickness`。

**返 回 值** 无。

**可移植性** 该函数仅适用于 Turbo C。适用于配有图形适配器的 IBMPC 及其兼容机。

**参 见** `arc, circle, fillellipse, getspectratio, sector, setaspectratio`

**示 例**

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

```

#include <conio.h>
int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int midx, midy;
 int stangle = 0, endangle = 360;
 int xradius = 100, yradius = 50;
 /* initialize graphics, local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk)
 /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1);
 }
 /* terminate with an error code */
 }
 midx = getmaxx() / 2;
 midy = getmaxy() / 2;
 setcolor(getmaxcolor());
 /* draw ellipse */
 ellipse(midx, midy, stangle, endangle,
 xradius, yradius);
 /* clean up */
 getch();
 closegraph();
 return 0;
}

```

## emit 将文字值直接插入源程序中

用 法 #include <dos.h>

void \_emit\_ (argument, ...);

原 型 在 dos.h

说 明 \_emit\_ 为一内部函数, 该函数允许用户在编译的时候, 将文字值直接插入目标码中。该函数经常用来直接嵌入机器语言指令, 而不必借助于内部汇编语言和汇编编译程序。

\_emit\_ 调用的参数均为单字节机器指令。然而, 鉴于该函数的性能, 借助于 C 变量, 也能完成复杂的指令。



如果熟悉 80×86 处理器族的机器语言,用户应该使用这个函数。通过该函数,可以用往一个过程的指令代码中填入任意字节值,如果这些字节是错误指令,则程序将非正常运行,并很容易导致死机。Turbo C 不对该函数的调用进行正确性检查。若填入修改机器寄存器和内存的指令,则 Turbo C 不会知道这些指令,更不会保护这些寄存器。这就和使用内部汇编语言一样(例如,在内部指令里允许使用寄存器 SI 和 DI)。用户可以使用该函数来完成一些巧妙的程序设计。

应给 `_emit_` 至少一个参数;任意多个参数都是允许的。该函数的参数的处理不同于本语言里其他函数调用。传给 `_emit_` 的参数将不做任何变换。`_emit_` 参数的格式有特殊限制,即必须是一个表达式。该函数可用于初始化静态目标。也就是说,参数可以是整型、浮点型常数或静态目标的地址。这些表达式的值将由调用点写至目标代码,就好像用它们来初始化数据。用户还可用参数变量或自由变量的地址、正或负的常数偏移量。对这些参数来说,变量从 BP 起开始的偏移量将被保存。

除了下列情况外,放置目标代码内的字节数目均由参数的类型决定:

- 如果是一个值为 0~255 的带符号整型常数(例如 0x90),则函数将把该常数当作字符处理;
- 如果使用了自由变量和参数变量的地址,且变量从 BP 开始的偏移量在 -128 和 127 之间,则仅写入一个字节;否则写入一个字。

简单地写入一个字节可以按以下形式调用函数:

```
emit(0x90);
```

如果用户想写入一个字,但是其值小于 255,可以按如下形式调用函数,即将该字变成无符号类型数据:

```
emit(0xB8,17u);
```

通过把地址强行转换为 `void near` 或 `void far` 可以得到一个长度为 2 字节或 4 字节的地址值。

返回值 无。

可移植性 仅适用于 Intel 80×86 系列处理器。

示 例

```
#include <dos.h>
int main(void)
{
 /* Emit code that will generate a print screen via int 5 */
 __emit__(0xcd,0x05);
 return 0;
}
```

## ■ enable 开硬件中断

用 法 `#include <dos.h>`

```
void enable(void);
```

原型在 `dos.h`

**说 明** enable 函数给程序员提供了一个灵活的硬件中断控制。enable 宏将开放中断,即允许接受任何设备产生的中断。

**返 回 值** 无。

**可移植性** 仅适用于 80×86 体系结构。

**参 见** disable, getvect

**示 例** /\* NOTE:

This is an interrupt service routine. You can NOT compile this program with Test Stack Overflow turned on and get an executable file which will operate correctly.

```
*/
#include <stdio.h>
#include <dos.h>
#include <conio.h>
/* The clock tick interrupt */
#define INTR 0X1C
void interrupt (*oldhandler)(void);
int count=0;
void interrupt handler(void)
{
 /* disable interrupts during the handling of the interrupt */
 disable();
 /* increase the global counter */
 count++;
 /* re enable interrupts at the end of the handler */
 enable();
 /* call the old routine */
 oldhandler();
}
int main(void)
{
 /* save the old interrupt vector */
 oldhandler = getvect(INTR);
 /* install the new interrupt handler */
 setvect(INTR, handler);
 /* loop until the counter exceeds 20 */
 while (count < 20)
 printf("count is %d\n",count);
 /* reset the old interrupt handler */
 setvect(INTR, oldhandler);
 return 0;
}
```

## ■ eof 检测文件是否结束

用 法 `#include <io.h>`

`int eof(int handle);`

原 型 在 `io.h`

说 明 `eof` 函数检测与 `handle` 相关联的文件是否到了末尾。

返 回 值 如果当前文件指针位置是文件末尾,则 `eof` 返回 1;否则返回 0。

返回值为 -1 表示出错,并置全局变量 `errno` 为:

`EBADF` 无效文件名

可移植性 仅适用于 DOS 系统。

参 见 `clearerr`, `feof`, `ferror`, `peror`,

示 例 `#include <sys\stat.h>`

`#include <string.h>`

`#include <stdio.h>`

`#include <fcntl.h>`

`#include <io.h>`

`int main(void) {`

`int handle;`

`char msg[] = "This is a test";`

`char ch;`

`/* create a file */`

`handle = open("DUMMY.FIL", O_CREAT | O_RDWR,`  
`S_IRREAD | S_IWRITE);`

`/* write some data to the file */`

`write(handle, msg, strlen(msg));`

`/* seek to the beginning of the file */`

`lseek(handle, 0L, SEEK_SET);`

`/* reads chars from the file until it reaches EOF */`

`do {`

`read(handle, &ch, 1);`

`printf("%c", ch);`

`} while (! eof(handle));`

`close(handle);`

`return 0;`

`}`

## ■ `execl`、`execle`、`execip`、`execipe`、

`execv`、`execve`、`execvp`、`execvpe`

装入并运行其它程序

用 法 `#include <process.h>`

```

int execl(char * path, char * arg0, * arg1, ..., * argn, NULL);
int execlp(char * path, char * arg0, * arg1, ..., * argn, NULL, char * * env);
int execlpe(char * path, char * arg0, * arg1, ..., * argn, NULL, char * * env);
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);

```

原型在 process.h

说明 exec... 函数族加载并运行其它程序。当 exec... 调用成功时,子进程将覆盖父进程。调用该族函数时必须有足够的内存空间用于加载和执行子进程。

path 是调用的子进程的文件名。exec... 函数使用标准的 DOS 搜索算法来搜索 path。

- 若子程序文件无扩展名,则函数将搜索给定的文件名;如果没有找到该文件,则在文件名后加上扩展名.COM 后再进行搜索;如果还没有找到,函数将在文件名之后加上扩展名.EXE 并进行最后一次搜索。
- 若有扩展名或句点,则搜索所给文件名。

加在 exec... 函数族名后的后缀 p、l、v、e 表示指定的函数具有某种功能:

- p 表示函数能够在由 DOS 环境变量 PATH 的目录中搜索子进程文件(若没有 p 后缀,则函数只能在当前目录下寻找)。如果参数 path 没有指明文件所在目录,函数将首先在当前目录搜索,然后在 DOS 环境变量 PATH 所指定的目录中搜索。
- l 表示参数指针(arg0, arg1, ..., argn)按单独参数传送。一般说来,当预先知道要传送参数的个数时,通常用后缀 l 的 exec 函数。
- v 表示参数指针(arg[0]..., argv[n])按指针数组传送。一般说来,当要传送的参数可变时,使用带后缀 v 的 exec 函数。
- e 表示参数 env 可以传送到子进程,该函数可用来修改子进程的环境。若没有 e 后缀,子进程将继承父进程的环境。

exec... 函数族中的每个函数都必须有两个指定参数后缀(l 或 v)中的一个,而路径搜索和环境继承后缀(p 或 e)则是任选的。

例如:

- execl 是一个 exec... 函数,它采用单独参数形式,只在根目录和当前目录下搜索子进程名,并把父进程的环境传递给子进程。
- execvpe 是一个 exec... 函数,它采用参数指针数组的形式,把 PATH 合在对子进程的搜索路径中,并接受 env 参数以改变子进程的环境。

exec... 函数必须至少传送一个参数(arg0 或 arg[0])给子进程;按约定,该参数为 path 的一个拷贝(对第 0 个参数使用不同的值不会产生错误)。

在 DOS 3.x, path 在子进程中可用, 在早期的 DOS 版本中子进程不能使用传送的第 0 个参数(arg0 或 arg [0]值)。

当使用 l 后缀时, arg0 通常指向 path, 而 arg1, ..., argn 指向组成新参数表的字符串。在 argn 后加上标志 NULL, 表示表结束。

当使用 e 后缀时, 通过参数 env 传送一个新的环境设置表。环境参数是一个字符指针数组, 数组中的每个元素指向一个以空字符终结的字符串, 形式如下:

```
envvar=value
```

其中 envvar 是环境变量名, value 是 envvar 所设置的字符串值。env 的最后一个元素为 NULL。当 env 为 NULL 时, 子进程继承父进程的环境设置。

arg0+arg1+...+argn(或 argv [0]+argv [1]+...+argv [n])的总长度(包括分隔参数的空格)必须小于 128 字节, 空终结符(NULL)不计在内。

当调用一个 exec... 函数, 原先已打开的文件在子进程中仍然是打开的。

**返回值** 如果调用成功, exec... 函数不返回任何值; 在出错时, exec... 函数返回 -1, 并将全局变量 errno 置为下列值之一:

|         |           |
|---------|-----------|
| E2BIG   | 参数表太长     |
| EACCES  | 无此权限      |
| EMFILE  | 打开的文件太多   |
| ENOENT  | 路径或文件名找不到 |
| ENDEXEC | 运行格式错误    |
| ENOMEM  | 无足够内存空间   |

**可移植性** 仅适用于 DOS 系统

**参见** abort, atexit, \_exit, exit, \_fpreset, searchpath, spawn..., system

**示例** /\* execv example \*/

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

```
void main(int argc, char *argv[])
{
 int i;
 printf("Command line arguments:\n");
 for (i=0; i<argc; i++)
 printf("[%2d] : %s\n", i, argv[i]);
 printf("About to exec child with arg1 arg2 ... \n");
 execv("CHILD.EXE", argv);
 perror("exec error");
 exit(1);
}
```

/\* execve example \*/

```

#include <process.h>
#include <stdio.h>
#include <errno.h>
void main(int argc, char * argv[], char * * envp)
{
 int i;
 printf("Command line arguments:\n");
 for (i=0; i<argc; ++i)
 printf("[%2d] : %s\n", i, argv[i]);
 printf("About to exec child with arg1 arg2 ... \n");
 execve("CHILD.EXE", argv, envp);
 perror("exec error");
 exit(1);
}

/* execvp example */
#include <process.h>
#include <stdio.h>
#include <errno.h>
void main(int argc, char * argv[])
{
 int i;
 printf("Command line arguments:\n");
 for (i=0; i<argc; ++i)
 printf("[%2d] : %s\n", i, argv[i]);
 printf("About to exec child with arg1 arg2 ... \n");
 execvp("CHILD.EXE", argv);
 perror("exec error");
 exit(1);
}

/* execvpe example */
#include <process.h>
#include <stdio.h>
#include <errno.h>
void main(int argc, char * argv[], char * * envp)
{
 int i;
 printf("Command line arguments:\n");
 for (i=0; i<argc; ++i)
 printf("[%2d] : %s\n", i, argv[i]);
 printf("About to exec child with arg1 arg2 ... \n");
 execvpe("CHILD.EXE", argv, envp);
 perror("exec error");
}

```

```

 exit(1);
}

/* execlpe example */
#include <process.h>
#include <stdio.h>
#include <errno.h>
int main(int argc, char * argv[], char * * envp)
{
 int i;
 printf("Command line arguments:\n");
 for (i=0; i < argc; ++i)
 printf("[%2d] %s\n", i, argv[i]);
 printf("About to exec child with arg1 arg2 ... \n");
 execlpe("CHILD.EXE", "CHILD.EXE", "arg1", "arg2", NULL, envp);
 perror("exec error");
 exit(1);
 return 0;
}

```

## ■ exit 终止程序 ■

用 法 #include<stdlib.h>

void \_exit(int status)

原 型 在 process.h, stdlib.h

说 明 \_exit 终止调用进程,但不关闭文件,不清除输出缓存,也不调用出口函数。

调用进程时用 status 作为出口状态,一般说来,值 0 表示正常出口,非 0 值表示有错误发生。

返 回 值 无。

可移植性 适用于 UNIX 系统。

参 见 abort, atexit, exec..., exit, spawn...

示 例 #include <stdlib.h>

#include <stdio.h>

void done(void);

int main(void)

```

{
 atexit(done);

```

```

 _exit(0);

```

```

 return 0;
}

```

void done()

```

{
 printf("hello\n");
}

```

}

## ■ exit 终止程序

用 法 #include<stdlib.h>

void exit(int status)

原 型 在 process.h, stdlib.h

说 明 exit 函数将终止调用进程。在退出程序之前,所有文件均被关闭,缓冲输出(正等待输出)内容将刷新定义,并调用所有已刷新的“出口函数”(由 atexit 定义)。参数 status 被用来提供调用进程的出口状态,一般说来,0 值表示正常出口,非 0 值表示有错误发生,此时 status 将置为下列值之一:

EXIT\_SUCCESS 正常终止程序

EXIT\_FAILURE 非正常终止程序,并通知操作系统程序有错。

返 回 值 无。

可移植性 适用于 UNIX 系统,并且 ANSI C 中也有定义。

参 见 abort, atexit, exec..., \_exit, keep, signal, spawn...

示 例

```
#include <stdlib.h>
#include <conio.h>
#include <stdio.h>
int main(void)
{
 int status;
 printf("Enter either 1 or 2\n");
 status = getch();
 /* Sets DOS errorlevel */
 exit(status - '0');
 /* Note: this line is never reached */
 return 0;
}
```

## ■ exp 计算 e 的 x 次方

用 法 对于实数

#include<math.h>

double exp(double x)

对于复数

#include<complex.h>

complex exp(complex x);

原 型 在 对于实数

math.h

对于复数

complex.h

说 明 exp 计算指数函数 e 的 x 次方幂。复数的指数函数定义如下:

$\exp(x+yi) = \exp(x)(\cos(y) + i\sin(y))$

返 回 值 exp 返回 e 的 x 次方幂。

传给 exp 的参数会产生溢出或不可计算。若结果上溢,exp 返回 HUGE\_VAL。若结果值很大,函数将置局程变量 errno 为:



**ERANGE****结果越出范围**

如果结果下溢,exp 将返回 0.0,全局变量 errno 值不变。exp 的错误处理程序可通过 matherr 函数来修改。

**可移植性** 适用于 UNIX 系统,并且在 ANSI C 中有定义。

**参 见** frexp,ldexp,log,log10,matherr,pow,pow10,sqrt

**示 例**

```
#include <stdio.h>
#include <math.h>
int main(void)
{
 double result;
 double x = 4.0;
 result = exp(x);
 printf("'e' raised to the power \
of %lf (e ^ %lf) = %lf\n",
 x, x, result);
 return 0;
}
```

## ■ fabs 返回浮点数的绝对值

**用 法** #include<math.h>  
double fabs(double x);

**原 型** 在 math.h

**说 明** fabs 计算双精度类型数据 x 的绝对值。

**返 回 值** 返回 x 的绝对值。

**可移植性** 用于 UNIX 系统,并且在 ANSI C 中有也定义。

**参 见** abs,cabs,labs

**示 例**

```
#include <stdio.h>
#include <math.h>
int main(void)
{
 float number = -1234.0;
 printf("number: %f absolute value: %f\n", number, fabs(number));
 return 0;
}
```

## ■ farcalloc 从远程堆中分配内存

**用 法** #include<alloc.h>  
void far \* farcalloc(unsigned long nunits,unsigned long unitsz);

**原 型** 在 alloc.h

**说 明** farcalloc 从远程堆中为包含 nunits 个元素的数组分配内存,每一区 unitsz 字节长。

在远程堆中分配内存应注意：

- 所有可用 RAM 均可分配；
- 可分配大于 64K 的块；
- 可用远指针(或块大于 64K 时的大指针)存取被分配的块。

在紧缩、大型和巨型模式中, `farcalloc` 函数和 `calloc` 基本类似。不同之处在于 `farcalloc` 以 `unsigned long` 型数据为参数, 而 `calloc` 以 `unsigned` 型数据为参数。

微型模式程序不能使用 `farcalloc` 函数。

**返回值** `farcalloc` 返回一个指向新分配块的指针。如果没有足够的空间分配新块, 则函数返回空指针 `NULL`。

**可移植性** 仅适用于 DOS 系统。

**参 见** `calloc`, `farcoreleft`, `farfree`, `farmalloc`, `malloc`

**示 例**

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

int main(void)
{
 char far *fptr;
 char *str = "Hello";
 /* allocate memory for the far pointer */
 fptr = farcalloc(10, sizeof(char));
 /* copy "Hello" into allocated memory */
 /*
 Note: movedata is used because you might be in a small data model, in
 which case a normal string copy routine can not be used since it
 assumes the pointer size is near.
 */
 movedata(FP_SEG(str), FP_OFF(str),
 FP_SEG(fptr), FP_OFF(fptr),
 strlen(str));
 /* display string (note the F modifier) */
 printf("Far string is: %Fs\n", fptr);
 /* free the memory */
 farfree(fptr);
 return 0;
}
```

## ■ `farcoreleft` 返回远程堆中未使用内存的大小 ■

**用 法** `#include <alloc.h>`  
`unsigned long farcoreleft(void);`

**原 型 在** `alloc.h`

**说 明** farcoreleft 返回堆中最高已分配内存块之外未用内存区总量的大小。微型模式程序不能调用 farcoreleft 函数。

**返 回 值** farcoreleft 返回堆中在最高已分配块和可用内存区末尾之间的空间总量。

**可移植性** 仅适用于 DOS 系统。

**参 见** coreleft, farcalloc, farmalloc

**示 例**

```
#include <stdio.h>
#include <alloc.h>
int main(void)
{
 printf("The difference between the highest allocated block in the far\n");
 printf("heap and the top of the far heap is: %lu bytes\n", farcoreleft());
 return 0;
}
```

## ■ farfree 从远程堆中释放一块已分配内存 ■

**用 法** #include<alloc.h>  
void farfree(void far \* block);

**原 型 在** alloc.h

**说 明** farfree 释放远程堆中以前所分配的内存块。微型模式程序不能用 farfree 函数。在小型和中型存储模式中,由 farmalloc 分配的内存块不能用通常的 free 函数来释放,而 malloc 分配的内存块则不能用 farfree 函数来释放。在这两种存储模式中,这两种堆是完全不同的。

**返 回 值** 无。

**可移植性** 仅适用于 DOS 系统。

**参 见** farcalloc, farmalloc

**示 例**

```
#include <stdio.h>
#include <alloc.h>
#include <string.h>
#include <dos.h>
int main(void)
{
 char far * fptr;
 char * str = "Hello";
 /* allocate memory for the far pointer */
 fptr = farcalloc(10, sizeof(char));
 /* copy "Hello" into allocated memory */
 /*
 Note: movedata is used because you might be in a small data model,
 in which case a normal string copy routine can't be used since it
 assumes the pointer size is near.
 */
}
```

```

 movedata(FP_SEG(str), FP_OFF(str),
 FP_SEG(fptr), FP_OFF(fptr),
 strlen(str));
 /* display string (note the F modifier) */
 printf("Far string is: %Fs\n", fptr);
 /* free the memory */
 farfree(fptr);
 return 0;
}

```

## ■ farmalloc 从远堆中分配内存 ■

用 法 #include <alloc.h>

```
void far *farmalloc(unsigned long nbytes);
```

原 型 在 alloc.h

说 明 farmalloc 从远堆中分块某长为 nbytes 字节的内存  
从远堆分配内存要注意:

- 所有可用 RAM 能被分配;
- 大于 64K 的块能被分配;
- 用远指针存取被分配的块。

在紧缩、大型和巨型模式中 farmalloc 与 malloc 不完全相同,但许多地方是相似的。farmalloc 以 unsigned long 型为参数,而 malloc 以 unsigned 型为参数。微型模式不能使用 farmalloc。

返 回 值 farmalloc 返回指向新分配内存的指针。如果已没有足够的内存,则返回 NULL。

可移植性 只适用于 DOS。

参 见 farcalloc, farcoreleft, farfree, farrealloc, malloc。

示 例 #include <stdio.h>

```
#include <alloc.h>
```

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

```
#include <dos.h>
```

```
int main(void)
```

```
{
```

```
 char far *fptr;
```

```
 char *str = "Hello";
```

```
 /* allocate memory for the far pointer */
```

```
 fptr = farmalloc(10);
```

```
 /* copy "Hello" into allocated memory */
```

```
 /*
```

```
 Note: movedata is used because we might be in a small data model,
 in which case a normal string copy routine can not be used since it
 assumes the pointer size is near.
```

```
 */
```

```

movedata(FP_SEG(str), FP_OFF(str),
 FP_SEG(fpstr), FP_OFF(fpstr),
 strlen(str)); /* display string (note the F modifier) */
printf("Far string is, %Fs\n", fpstr);
/* free the memory */
farfree(fpstr);
return 0;
}

```

## ■ farrealloc 调整远堆中的已分配块

**用 法** #include <alloc.h>

void far \* farrealloc(void far \* oldblock, unsigned long nbytes);

**原 型 在** alloc.h

**说 明** farrealloc 调整所分配块的长度,使其长度为 nbytes,如果需要的话,还可以把块内容复制到一个新位置。

对于从远堆中调整已分配块,要注意:

- 所有可用 RAM 能被分配;
- 大于 64K 的块能被分配;
- 远指针用于存取被分配的块。

微型模式不能使用此函数。

**返 回 值** farrealloc 返回调整后新内存的地址,这个地址可能与原地址不同;如果不能重新分配, farrealloc 返回 NULL。

**可移植性** 只适用于 DOS。

**参 见** farmalloc, realloc

**示 例** #include <stdio.h>

```

#include <alloc.h>
int main(void)
{
 char far * fpstr;
 fpstr = farmalloc(10);
 printf("First address: %Fp\n", fpstr);
 fpstr = farrealloc(fpstr, 20);
 printf("New address : %Fp\n", fpstr);
 farfree(fpstr);
 return 0;
}

```

## ■ fclose 关闭一个流

**用 法** #include <stdio.h>

int fclose(FILE \* stream);

**原 型 在** stdio.h

**说 明** `fclose` 关闭指定的流, 在关闭前清除所有与 `stream` 相联的缓冲区, 释放系统分配的缓冲区, 但由 `setbuf` 设置的缓冲区不能自动释放。

**返回值** `fclose` 在调用成功时返回 0; 否则返回 EOF。

**可移植性** 适用了 UNIX 系统, 在 ANSI C 中有定义。

**参 见** `close`, `fcloseall`, `fdopen`, `fflush`, `flushall`, `fopen`, `freopen`

**示 例**

```
#include <string.h>
#include <stdio.h>
int main(void)
{
 FILE *fp;
 char buf[11] = "0123456789";
 /* create a file containing 10 bytes */
 fp = fopen("DUMMY.FIL", "w");
 fwrite(&buf, strlen(buf), 1, fp);
 /* close the file */
 fclose(fp);
 return 0;
}
```

## ■ `fcloseall` 关闭打开流

**用 法** `#include <stdio.h>`  
`int fcloseall(void);`

**原型在** `stdio.h`

**说 明** `fcloseall` 关闭所有打开的流, 由 `stdin`, `stdout`, `stderr` 和 `stdaux` 设置的流除外。

**返回值** `fcloseall` 返回关闭流的总数。如果它发现错误, 则返回 EOF。

**可移植性** 适用于 UNIX 系统。

**参 见** `fclose`, `fdopen`, `flushall`, `fopen`, `freopen`。

**示 例**

```
#include <stdio.h>
int main(void)
{
 int streams_closed;
 /* open two streams */
 fopen("DUMMY.ONE", "w");
 fopen("DUMMY.TWO", "w");
 /* close the open streams */
 streams_closed = fcloseall();
 if (streams_closed == EOF)
 /* issue an error message */
 perror("Error");
 else
```

```

 /* print result of fcloseall() function */
 printf("%d streams were closed.\n", streams_closed);
 return 0;
}

```

## ■ fcvt 将浮点数转换为字符串

用 法 #include <stdlib.h>

char \*fcvt(double value, int ndig, int \*dec, int \*sign);

原 型 在 stdlib.h

说 明 fcvt 把 value 转换为 ndig 位数字的字符串, 字符串以空字符结尾, 并返回指向该字符串的指针, 开始处的十进制小数点通过 dec 间接存储(dec 为负表示小数点在返回数字串的左边)。数字串中没有小数点。如果 value 为负, sign 所指的字是非零数, 否则是零。ndig 的数字在使用前已被取整。

返 回 值 fcvt 的返回值均为指向静态数据的指针, 其内容在每次调用 fcvt 时重写。

可移植性 fcvt 适用于 UNIX, 在 ANSI C 中无定义, 建议不要用于可移植性程序中。

参 见 ecvt, gevt, sprintf

示 例 #include <stdlib.h>

#include <stdio.h>

#include <conio.h>

int main(void)

{

char \*string;

double value;

int dec, sign;

int ndig = 10;

clrscr();

value = 9.876;

string = ecvt(value, ndig, &dec, &sign);

printf("string = %s dec = %d sign = %d\n", string, dec, sign);

value = -123.45;

ndig = 15;

string = ecvt(value, ndig, &dec, &sign);

printf("string = %s dec = %d sign = %d\n", string, dec, sign);

value = 0.6789e5; /\* scientific notation \*/

ndig = 5;

string = ecvt(value, ndig, &dec, &sign);

printf("string = %s dec = %d sign = %d\n", string, dec, sign);

return 0;

}

## ■ fdopen 把流与一个文件句柄相联

用 法 #include <stdio.h>

`FILE *open(int handle, char *type);`

原型在 `stdio.h`

说明 `fdopen` 使流 `stream` 与一个从 `creat`、`dup`、`dup2` 或 `open` 得到的文件句柄相联。  
`stream` 类型必须与打开文件的句柄 `handle` 的模式相匹配。

调用 `fopen` 中的字符串 `type` 可以是下述值之一：

- `r` 以只读打开；
- `w` 以写方式打开；
- `a` 打开并在文件末尾写，当文件不存在时，以写方式创建文件。
- `r+` 打开一存在文件用于更新(读和写)；
- `w+` 创建一个新文件用于更新(读和写)；
- `a+` 打开用于附加，打开(如果文件不存在，则创建)文件用于在末尾添加。为指明一个用文本方式打开或创建的指定文件，可以在 `type` 后面加上 `t`(如 `rt`、`w+t` 等)；如果以二进制打开或创建一个文件，则加上 `b`(如 `wb`、`a+b`)。

若 `type` 串中未给出 `t` 或 `b`，则由全局变量 `_fmode` 决定打开或创建方式；若 `_fmode` 为 `O_BINARY`，则文件以二进制方式打开；若 `_fmode` 为 `O_TEXT`，则文件以文本方式打开。常量 `O_BINARY` 和 `O_TEXT` 在 `fcntl.h` 中定义。

当文件打开用于更新时，在结果流上既可进行输入又可进行输出。但如果没有进行 `fseek` 或 `rewind` 调用，输出就不能直接跟在输入后面。同样，如果没有使用 `fseek` 或 `rewind` 调用，输入也不能直接跟在输出后面。在遇到文件结束时不能进行输入。

返回值 若调用成功，则返回指向所打开的流的指针；出现错误时，返回 `NULL`。

可移植性 适用于 UNIX 系统。

参见 `fclose`、`fopen`、`freopen`、`open`

示例

```
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>

int main(void)
{
 int handle;
 FILE *stream;
 /* open a file */
 handle = open("DUMMY.FIL", O_CREAT,
 S_IREAD | S_IWRITE);
 /* now turn the handle into a stream */
 stream = fdopen(handle, "w");
 if (stream == NULL)
 printf("fdopen failed\n");
 else
```



```
{
 fprintf(stream, "Hello world\n");
 fclose(stream);
}
return 0;
}
```

## ■ feof 检测流上的文件结束标志 ■

用 法 #include <stdio.h>

int feof(FILE \* stream);

原 型 在 stdio.h

说 明 feof 是一个用于检测给定流文件结束标志的宏。当文件结束标志被置位时,对文件的读操作将返回此标志,直到调用 rewind 或关闭文件为止。每一次输入操作都复位文件结束标志。

返 回 值 如果检测到文件结束标志,则 feof 返回非零值,否则返回零。

可移植性 适用于 UNIX 系统,在 ANSI C 中有定义。

参 见 clearerr, eof, ferror, perror.

示 例 #include <stdio.h>

```
int main(void)
{
 FILE * stream;
 /* open a file for reading */
 stream = fopen("DUMMY.FIL", "r");
 /* read a character from the file */
 fgetc(stream);
 /* check for EOF */
 if (feof(stream))
 printf("We have reached end-of-file\n");
 /* close the file */
 fclose(stream);
 return 0;
}
```

## ■ ferror 检测流上的错误 ■

用 法 #include <stdio.h>

int ferror(FILE \* stream);

原 型 在 stdio.h

说 明 ferror 是一个用于检测给定流读写错误的宏。如果流错误标志被置位,则它保持不变,直到调用 clearerr, rewind 或关闭流为止。

返 回 值 如果检测到指定流 stream 上有错误, ferror 返回非零值。

可移植性 适用于 UNIX 系统,在 ANSI C 中有定义。

参 见 clearerr, eof, feof, fopen, gets, perror.

示 例 #include <stdio.h>

```
int main(void)
{
 FILE * stream;
 /* open a file for writing */
 stream = fopen("DUMMY.FIL", "w");
 /* force an error condition by attempting to read */
 (void) getc(stream);
 if (ferror(stream)) /* test for an error on the stream */
 {
 /* display an error message */
 printf("Error reading from DUMMY.FIL\n");
 /* reset the error and EOF indicators */
 clearerr(stream);
 }
 fclose(stream);
 return 0;
}
```

## ■ fflush 刷新一个流

用 法 #include <stdio.h>

```
int fflush(FILE * stream);
```

原 型 在 stdio.h

说 明 如果指定的流有输出缓冲区, fflush 就把输出写在相关联的文件上。fflush 执行后, 流保持打开, fflush 对没有缓冲的流不起作用。

返 回 值 fflush 在调用成功时返回 0; 出现错误时, 返回 EOF。

可移植性 适用于 UNIX 系统, 在 ANSI C 中有定义。

参 见 fclose, flushall, setbuf, setvbuf

示 例 #include <string.h>

```
#include <stdio.h>
#include <conio.h>
#include <io.h>
void flush(FILE * stream);
int main(void)
{
 FILE * stream;
 char msg[] = "This is a test";
 /* create a file */
 stream = fopen("DUMMY.FIL", "w");
 /* write some data to the file */
```

```

 fwrite(msg, strlen(msg), 1, stream);
 clrscr();
 printf("Press any key to flush DUMMY.FIL,");
 getch();
 /* flush the data to DUMMY.FIL without closing it */
 flush(stream);
 printf("\nFile was flushed, Press any key to quit:");
 getch();
 return 0;
}

void flush(FILE *stream)
{
 int duphandle;
 /* flush the stream's internal buffer */
 fflush(stream);
 /* make a duplicate file handle */
 duphandle = dup(fileno(stream));
 /* close the duplicate handle to flush the DOS buffer */
 close(duphandle);
}

```

## ■ fgetc 从流中读取字符

**用 法** #include <stdio.h>

int fgetc(FILE \*stream);

**原 型 在** stdio.h

**说 明** fgetc 返回指定输入流的下一个字符。

**返 回 值** 在成功调用时, fgetc 返回所读的字符, 它已被转换为无符号扩展的整型值。在遇到文件结束或出错时, 返回 EOF。

**可移植性** 适用于 UNIX 系统, 在 ANSI C 中有定义。

**参 见** fgetchar, fputc, getc, getch, getchar, getche, ungetc, ungetch

**示 例** #include <string.h>

#include <stdio.h>

#include <conio.h>

int main(void)

{

FILE \*stream;

char string[] = "This is a test";

char ch;

/\* open a file for update \*/

stream = fopen("DUMMY.FIL", "w+");

/\* write a string into the file \*/

fwrite(string, strlen(string), 1, stream);

```
/* seek to the beginning of the file */
fseek(stream, 0, SEEK_SET);
do
{
 /* read a char from the file */
 ch = fgetc(stream);
 /* display the character */
 putchar(ch);
} while (ch != EOF);
fclose(stream);
return 0;
}
```

## ■ fgetchar 从流中读取字符 ■

用 法 #include <stdio.h>

int fgetchar(void);

原 型 在 stdio.h

说 明 fgetchar 返回 stdin 中的下一个字符。它的原始定义为 fgetc(stdin)。

返 回 值 成功调用时 fgetchar 返回所读的字符,它已被转换为无符号扩展的整形值。遇到文件结束或出错时,返回 EOF。

可移植性 适用于 UNIX 系统。

参 见 fgetc, putchar, getchar

示 例 #include <stdio.h>

```
int main(void)
{
 char ch;
 /* prompt the user for input */
 printf("Enter a character followed by <Enter>: ");
 /* read the character from stdin */
 ch = fgetchar();
 /* display what was read */
 printf("The character read is, '%c'\n", ch);
 return 0;
}
```

## ■ fgetpos 取得当前文件指针 ■

用 法 #include <stdio.h>

int fgetpos(FILE \*stream, fpos\_t \*pos);

原 型 在 stdio.h

说 明 fgetpos 把与 stream 关联的文件指针的位置保存在由 pos 所指的位置。fpos\_t 在 stdio.h 中定义为“typedef long fpos\_t;”。

**返回值** 若调用成功, `fgetpos` 返回 0; 若失败, 返回非零值并且置全局变量 `errno` 为 `EBADF` 或 `EINVAL`.

**可移植性** `fgetpos` 与 ANSI C 兼容。

**参见** `fseek`, `fsetpos`, `ftell`, `tell`

**示例**

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

int main(void)
{
 FILE * stream;
 char string[] = "This is a test";
 fpos_t filepos;
 /* open a file for update */
 stream = fopen("DUMMY.FIL", "w+");
 /* write a string into the file */
 fwrite(string, strlen(string), 1, stream);
 /* report the file pointer position */
 fgetpos(stream, &filepos);
 printf("The file pointer is at byte %ld\n", filepos);
 fclose(stream);
 return 0;
}
```

## ■ `fgets` 从流中读取一字符串

**用法** `#include <stdio.h>`  
`char * fgets(char s, int n, FILE * stream);`

**原型在** `stdio.h`

**说明** `fgets` 从输入流 `stream` 中读入字符存到 `s` 串中。当读了 `n-1` 个字符或遇到换行符时, 函数停止读过程。`fgets` 在 `s` 串尾保留换行字符。读入的最后一个字符后面附加一空字符。

**返回值** `fgets` 在调用成功时, 返回字符串参数 `s`; 在出错或遇到文件结束时, 返回 `NULL`。

**可移植性** 适用于 UNIX 系统, 在 ANSI C 中有定义, Kernighan 和 Ritchie 也作了定义。

**参见** `cgets`, `fputs`, `gets`

**示例**

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

int main(void)
{
 FILE * stream;
 char string[] = "This is a test";
 char msg[20];
 /* open a file for update */
 stream = fopen("DUMMY.FIL", "w+");
```

```

 /* write a string into the file */
 fwrite(string, strlen(string), 1, stream);
 /* seek to the start of the file */
 fseek(stream, 0, SEEK_SET);
 /* read a string from the file */
 fgets(msg, strlen(string)+1, stream);
 /* display the string */
 printf("%s", msg);
 fclose(stream);
 return 0;
}

```

## ■ filelength 取文件长度 ■

用 法 #include <io.h>

long filelength(int handle);

原 型 在 io.h

说 明 filelength 返回与句柄 handle 相联的文件字节长度。

返 回 值 调用成功时, filelength 返回一个长整型值, 即为文件的字节长度, 在发生错误时, 返回 -1, 并置 errno 值为:

EBADF 无效文件号

可移植性 只适用于 DOS.

参 见 fopen, lseek, open

示 例 #include <string.h>

```

#include <stdio.h>
#include <fcntl.h>
#include <io.h>
int main(void)
{
 int handle;
 char buf[11] = "0123456789";
 /* create a file containing 10 bytes */
 handle = open("DUMMY.FIL", O_CREAT);
 write(handle, buf, strlen(buf));
 /* display the size of the file */
 printf("file length in bytes: %ld\n", filelength(handle));
 /* close the file */
 close(handle);
 return 0;
}

```

## ■ fileno 取得文件句柄 ■

用 法 #include <stdio.h>

```
int fileno(FILE * stream);
```

原型在 `stdio.h`

说明 `fileno` 是一返回指定流 `stream` 的文件句柄的宏。如果 `stream` 有多个句柄, `fileno` 返回流第一次被打开时所赋的文件句柄。

返回值 `fileno` 返回与 `stream` 相联的整型文件句柄。

可移植性 适用于 UNIX 系统。

参见 `fdopen`, `fopen`, `freopen`

示例 `#include <stdio.h>`

```
int main(void)
{
 FILE * stream;
 int handle;
 /* create a file */
 stream = fopen("DUMMY.FIL", "w");
 /* obtain the file handle associated with the stream */
 handle = fileno(stream);
 /* display the handle number */
 printf("handle number: %d\n", handle);
 /* close the file */
 fclose(stream);
 return 0;
}
```

## ■ `filellipse` 画椭圆饼

用法 `#include <graphics.h>`

```
void far filellipse(int x,int y,int xradius,int yradius);
```

原型在 `graphics.h`

说明 本函数以  $(x,y)$  为中心, `xradius` 和 `yradius` 为长轴和短轴画一椭圆, 椭圆用当前填充颜色和填充方式填充。

返回值 无。

可移植性 仅适用于 Turbo C, 而且只能在有图形适配器的 IBM PC 及其兼容机上使用。

参见 `arc`, `circle`, `ellipse`, `getaspectratio`, `pieslice`, `setaspectratio`

示例 `#include <graphics.h>`

```
#include <conio.h>
```

```
int main(void)
```

```
{
 int gdriver = DETECT, gmode;
 int xcenter, ycenter, i;
 initgraph(&gdriver, &gmode, "");
 xcenter = getmaxx() / 2;
 ycenter = getmaxy() / 2;
```

```

 for (i=0; i<13; i++)
 {
 setfillstyle(i,WHITE);
 fillellipse(xcenter,ycenter,100,50);
 getch();
 }
 closegraph();
 return 0;
}

```

## ■ fillpoly 画多边形

用 法 #include <graphics.h>

void far fillpoly(int numpoints, int far \* polypoints);

原型在 graphics.h

说 明 fillpoly 用当前的画线类型和颜色,画一顶点数由 numpoints 指定的多边形(相于同 drawpoly),然后用当前填充颜色和填充方式填充此多边形。polypoints 指向整数序列(共有 numpoints \* 2 个整数),每个整数对应给出多边形一个顶点的 x 和 y 坐标。

返回值 无。

可移植性 这个函数只适用于 Turbo C,而且只能在装有图形适配器的 IBM PC 及其兼容机上使用。

参 见 drawpoly, floodfill, graphresult, setfillstyle

示 例

```

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

int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int i, maxx, maxy;
 /* our polygon array */
 int poly[8];
 /* initialize graphics, local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk)
 /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 }
}

```



```

 printf("Press any key to halt,");
 getch();
 exit(1);
 /* terminate with an error code */
}

maxx = getmaxx();
maxy = getmaxy();
poly[0] = 20; /* 1st vertex */
poly[1] = maxy / 2;
poly[2] = maxx - 20; /* 2nd vertex */
poly[3] = 20;
poly[4] = maxx - 50; /* 3rd vertex */
poly[5] = maxy - 20;
/*
 4th vertex. fillpoly automatically
 closes the polygon.
*/
poly[6] = maxx / 2;
poly[7] = maxy / 2;
/* loop through the fill patterns */
for (i=EMPTY_FILL; i<USER_FILL; i++)
{
 /* set fill pattern */
 setfillstyle(i, getmaxcolor());
 /* draw a filled polygon */
 fillpoly(4, poly);
 getch();
}
/* clean up */
closegraph();
return 0;
}

```

## ■ findfirst 查找第一个匹配文件 ■

用 法 #include <dir.h>  
#include <dos.h>

int findfirst(const char pathname, struct ffblk ffbk, int attrib);

原 型 在 dir.h

说 明 findfirst 通过 DOS 系统调用 0x4E 对磁盘目录进行搜索。pathname 是一个带有可选的驱动器号、路径名、文件名的字符串,文件名中可以包含 DOS 通配符(\* 或? 等)。如果找到了一个匹配的文件,该文件目录信息填入 ffbk 结构中。结构 ffbk 的格式如下:

```

struct fblk {
 char ff_reserved [2]; /* DOS 保留 */
 char ff_attrib; /* 文件的属性 */
 int ff_ftime; /* 文件时间 */
 int ff_fdate; /* 文件日期 */
 long ff_fsize; /* 文件大小 */
 char ff_name [13]; /* 文件名字 */
};

```

attrib 是一个 DOS 文件属性字节,用于在搜索过程中选择符合条件的文件。

attrib 可以是下列值之一,在 dos.h 中定义:

|           |      |
|-----------|------|
| FA_RDONLY | 只读文件 |
| FA_HIDDEN | 隐含文件 |
| FA_SYSTEM | 系统文件 |
| FA_LABEL  | 卷标   |
| FA_DIREC  | 目录   |
| FA_ARCH   | 档案   |

有关这些属性的更详信息,请参看《DOS 参考手册》。

注意:ff\_time 和 ff\_fdate 中包含当前的时间和日期,都是 DOS 建立的 16 位结构,可以分成三部分。

|           |                           |
|-----------|---------------------------|
| ff_time:  |                           |
| 0~4 位     | 当前秒数除 2(如,这里的 10 代表 20 秒) |
| 5~10 位    | 分钟数                       |
| 11~15 位   | 小时数                       |
| ff_fdate: |                           |
| 0~4 位     | 日                         |
| 5~8 位     | 月                         |
| 9~15 位    | 年减去 1980(如 9 这里指的是 1989)  |

结构 ftime,与 ff\_ftime 和 ff\_fdate,它们表示时间日期的结构相同,ftime 在 io.h 中有定义。

**返回值** 在成功找到与 pathname 相匹配的文件名后,findfst 返回 0;如果没有找到文件或文件名错误,则返回 -1,并置全局变量 errno 为下列值之一:

|         |            |
|---------|------------|
| ENOENT  | 路径或文件名没有找到 |
| ENMFILE | 没有更多的文件    |

**可移植性** 只适用于 DOS。

**参见** findnext

**示例** /\* findnext example \*/  

```

#include <stdio.h>
#include <dir.h>
int main(void)

```

```

{
 struct fblk fblk;
 int done;
 printf("Directory listing of *.*\n");
 done = findfirst(" *.*", &fblk, 0);
 while (! done)
 {
 printf(" %s\n", fblk.ff_name);
 done = findnext(&fblk);
 }
 return 0;
}

```

## ■ findnext 查找下一个匹配文件 ■

**用 法** #include <dir.h>

int findnext(struct fblk \* fblk);

**原 型 在** dir.h

**说 明** findnext 用于取得与 findfirst 给定的 pathname 相匹配的后续文件。fblk 是调用 findfirst 时填充的同一块,该块包含了继续搜索的必要信息。每调用一次 findnext 将返回一个文件名,直到目录中找不到 name 相匹配的文件。

**返 回 值** 与 pathname 相匹配的文件名成功地找到后,findnext 返回 0;如果找不到文件或文件名中有错误,则返回 -1,并置全局变量 errno 为下列值之一:

|         |            |
|---------|------------|
| ENOENT  | 路径或文件名没有找到 |
| ENMFILE | 没有更多的文件    |

**可移植性** 只适用于 DOS.

**参 见** findfirst

**示 例** /\* findnext example \*/

```

#include <stdio.h>
#include <dir.h>
int main(void)
{
 struct fblk fblk;
 int done;
 printf("Directory listing of *.*\n");
 done = findfirst(" *.*", &fblk, 0);
 while (! done)
 {
 printf(" %s\n", fblk.ff_name);
 done = findnext(&fblk);
 }
 return 0;
}

```

## ■ floodfill 填充区域 ■

用 法 #include <graphics.h>

void far floodfill(int x, int y, int border);

原 型 在 graphics.h

说 明 floodfill 在位图形设备上填充一块封闭的区域, (x,y) 是待填区域的“起点”(seed point)。用颜色 border 围起来的区域将用当前填充和填充颜色来填充。如果起点在封闭区域内, 则区域内部被填充; 如果起点在封闭区域外, 则区域外部被填充。如果可能, 建议用 fillpoly 代替 floodfill, 这样可以兼容将来的版本。

注意: floodfill 不能用于 IBM-8514 驱动程序。

返 回 值 若在填充区域过程中出现错误, graphresult 返回 -1。

可移植性 只适用于 Turbo C, 且仅能在有图形适配器的 IBM PC 及其兼容机上使用。

参 见 drawpoly, fillpoly, graphresult, setcolor, setfillstyle

示 例

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode,
 int maxx, maxy;
 /* initialize graphics, local variables
 */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk)
 /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1);
 /* terminate with an error code */
 }
 maxx = getmaxx();
 maxy = getmaxy();
 /* select drawing color */
 setcolor(getmaxcolor());
```

```

/* select fill color */
setfillstyle(SOLID_FILL, getmaxcolor());
/* draw a border around the screen */
rectangle(0, 0, maxx, maxy);
/* draw some circles */
circle(maxx / 3, maxy / 2, 50);
circle(maxx / 2, 20, 100);
circle(maxx - 20, maxy - 50, 75);
circle(20, maxy - 20, 25);
/* wait for a key */
getch();
/* fill in bounded region */
floodfill(2, 2, getmaxcolor());
/* clean up */
getch();
closegraph();
return 0;
}

```

## ■ floor 下舍入

**用 法** #include <math.h>  
double floor(double x);

**原 型** 在 math.h

**说 明** floor 求得不大于 x 的最大整数。

**返 回 值** floor 求得不大于 x 的最大整数(为双精度型)。

**可移植性** 适用于 UNIX 系统,在 ANSI C 中有定义。

**参 见** ceil, fmod.

**示 例** #include <stdio.h>  
#include <math.h>

```

int main(void)
{
 double number = 123.54;
 double down, up;
 down = floor(number);
 up = ceil(number);
 printf("original number %10.2lf\n", number);
 printf("number rounded down %10.2lf\n", down);
 printf("number rounded up %10.2lf\n", up);
 return 0;
}

```

## ■ fflush 刷新所有流

用 法 #include <stdio.h>

int fflush(void);

原 型 在 stdio.h

说 明 fflush 释放所有与打开输入流相联的缓冲区,并把所有的与打开输出流相联的缓冲区的内容写到各自的文件中。跟在 fflush 后面的读操作从输入文件中读新数据到缓冲区中。在 fflush 执行后,流仍然打开。

可移植性 适用于 UNIX 系统。

参 见 fclose, fcloseall, fflush.

示 例 #include <stdio.h>

int main(void)

{

FILE \*stream;

/\* create a file \*/

stream = fopen("DUMMY.FIL", "w");

/\* flush all open streams \*/

printf("%d streams were flushed.\n", fflush());

/\* close the file \*/

fclose(stream);

return 0;

}

## ■ fmod 计算 x/y 的余数

用 法 #include <math.h>

double fmod(double x, double y);

原 型 在 math.h

说 明 fmod 计算 x 除以 y 的余数(即余数 f,其中 f 满足:对某些整数 a,  $x=ay+f$ , 且  $0 \leq f < |y|$ )。

返 回 值 fmod 返回余数 f,其中  $x=ay+f$ (如上所述),若  $y=0$ ,返回 0。

可移植性 与 ANSI C 兼容。

参 见 ceil, floor, modf.

示 例 #include <stdio.h>

#include <math.h>

int main(void)

{

double x = 5.0, y = 2.0;

double result;

result = fmod(x, y);

printf("The remainder of (%f / %f) is %f\n", x, y, result);

return 0;

## ■ fnmerge 建立文件路径

用 法 #include <dir.h>

```
void fnmerge(char * path, const char * drive, const char * dir, const char *
name, const char * ext);
```

原 型 在 dir.h

说 明 fnmerge 用其各成份参数建立一路径名文件的完整路径名为:

X;DIR\SUBDIR\NAME.EXT.

其中:X 由 drive 给定

\DIR\SUBDIR 由 dir 给定

NAME.EXT 由 name 和 ext 给定。

fnmerge 假定有足够的空间来存储所组成的完整路径名,完整路径名的最大长度为 MAXPATH,MAXPATH 在 dir.h 中定义。

fnmerge 和 fnsplit 这两个函数是互逆的,如果用 fnsplit 把一给定 path 分解,接着把各分解成份用 fnmerge 合起来,又可得到 path.

返 回 值 无。

可移植性 只适用于 DOS 系统。

参 见 fnsplit.

示 例

```
#include <string.h>
#include <stdio.h>
#include <dir.h>
int main(void)
{
 char s[MAXPATH];
 char drive[MAXDRIVE];
 char dir[MAXDIR];
 char file[MAXFILE];
 char ext[MAXEXT];
 getcwd(s,MAXPATH); /* get the current working directory */
 strcat(s,"\\"); /* append on a trailing character */
 fnsplit(s,drive,dir,file,ext); /* split the string to separate elems */
 strcpy(file,"DATA");
 strcpy(ext,".TXT");
 fnmerge(s,drive,dir,file,ext); /* merge everything into one string */
 puts(s); /* display resulting string */
 return 0;
}
```

## ■ fnsplit 分解完整的路径名

用 法 #include <dir.h>

```
int fnsplit(const char * path, char * drive, char * dir, char * name, char *
ext);
```

原 型 在 dir.h

说 明 fnsplit 把文件的完整路径看作串,形式如:

X:\DIR\SUBDIR\NAME.EXT

fnsplit 把文件名路径 path 分成四个成份,并把这些成份分别存在由 drive, dir, name 和 ext 所指的字符串中(每一成分都需要,但也可为 NULL,表示相应成份被分析,但不被存储)。

这些字符串的最大长度分别由常量 MAXDRIVE、MAXDIR、MAXPATH、MAXNAME、MAXEXT 给定(在 dir.h 中定义),每一最大值都包含空字符终结符。

| 常量       | 最大值  | 字符串                |
|----------|------|--------------------|
| MAXPATH  | (80) | path               |
| MAXDRIVE | 3    | drive, 包括冒号(,)     |
| MAXDIR   | 66   | dir, 包括开始和结尾反斜杠(\) |
| MAXFILE  | 9    | name               |
| MAXEXT   | 5    | ext, 包括开始圆点(.)     |

fnsplit 假定有足够的空间来存储每个非 NULL 成份。当 fnsplit 分解 path 时,按以下方法处理标点:

drive 后跟冒号(如 C:, A: 等)。

dir 前和后跟斜杠(如 \TC\include\, \source\ 等)。

name 包括文件名。

ext 前有一圆点(如 .C, .EXE 等)。

fnmerge 和 fnsplit 是互逆的,如果用 fnsplit 把一给定的 path 分解,接着把各分解成份用 fnmerge 合并起来,又可得 path。

返 回 值 fnsplit 返回一整数值(由五个标志位组成,在 dir.h 中定义),表示哪些完整路径成份在 path 中存在。这些标志和代表的对应成份为:

|           |               |
|-----------|---------------|
| EXTENSION | 扩展名           |
| FILENAME  | 文件名           |
| DIRECTORY | 目录名(可能为子目录)   |
| DIRVE     | 驱动器号(见 dir.h) |
| WILDCARDS | 通配符(* 或?)     |

可移植性 只适用于 DOS 系统。

参 见 fnmerge

```
示 例 #include <stdlib.h>
 #include <stdio.h>
 #include <dir.h>
 int main(void)
 {
```



```

char *s;
char drive[MAXDRIVE];
char dir[MAXDIR];
char file[MAXFILE];
char ext[MAXEXT];
int flags;
s=getenv("COMSPEC"); /* get the comspec environment parameter */
flags=fnsplit(s,drive,dir,file,ext);
printf("Command processor info:\n");
if(flags & DRIVE)
 printf("\tdrive: %s\n",drive);
if(flags & DIRECTORY)
 printf("\tdirectory: %s\n",dir);
if(flags & FILENAME)
 printf("\tfile: %s\n",file);
if(flags & EXTENSION)
 printf("\textension: %s\n",ext);
return 0;
}

```

## ■ fopen 打开一个流 ■

用 法 #include <stdio.h>

FILE \*fopen(const char \*filename, const char \*mode);

原 型 在 stdio.h

说 明 fopen 打开由 filename 指定的文件,并使其与一个流相联。它返回用于后续操作中指明该流的指针。在 fopen 调用中的字符串 mode 可以是下列值之一:

- r 以只读方式打开;
- w 以写方式打开,如果已存在有该文件名的文件,文件被重写;
- a 附加。打开用于在文件末尾写,当文件不存在时,创建用于写;
- r+ 打开一已存在文件用于更新(读或写);
- w+ 创建一新文件用于更新(读或写),如果已存在有该文件名的文件,文件被重写;
- a+ 打开用于附加,打开(如果文件不存在,则创建)文件用于在末尾添加。为指明一给定文件用文本方式打开或创建,可以在 mode 后面加上 t(如 rt、w+t);如果以二进制方式打开或创建,则加上 b(wb、a+b 等)。fopen 允许 t 和 b 插在字母和 + 中间,例如,rt+ 和 r+t 是等同的。

若 mode 中没有给出 t 或 b,则全局变量 \_fmode 决定打开或创建方式。若 \_fmode 为 O\_BINARY,则文件以二进制方式打开,若 \_fmode 为 O\_TEXT,则文件以文本方式打开。常量 O\_BINARY 和 O\_TEXT 在 fcntl.h 中定义。

当文件打用于更新时,在结果流上既可进行输入也可进行输出。但如果没有 fseek 或 rewind 调用,输出不能直接跟在输入后面,同样,如果没有给出 fseek 或

rewind 调用,输入也不能直接跟在输出后面,在遇到文件结束时不能进行输入。

**返回值** 若打开成功,则返回打开的流的指针,否则返回 NULL。

**可移植性** 适用于 UNIX 系统,在 ANSI C 中有定义,是 Kernighan 和 Ritchie 定义的。

**参见** creat, dup, fclose, fdopen, ferror, \_fmode(全局变量), fread, freopen, fseek, fwrite, open, rewind, setbuf, setmode。

**示例**

```
/* Program to create backup of the AUTOEXEC. BAT file */
#include <stdio.h>
int main(void)
{
 FILE *in, *out;
 if ((in = fopen("\\\\AUTOEXEC. BAT", "rt"))
 == NULL)
 {
 fprintf(stderr, "Cannot open input file. \n");
 return 1;
 }
 if ((out = fopen("\\\\AUTOEXEC. BAK", "wt"))
 == NULL)
 {
 fprintf(stderr, "Cannot open output file. \n");
 return 1;
 }
 while (!feof(in))
 fputc(fgetc(in), out);
 fclose(in);
 fclose(out);
 return 0;
}
```

## ■ FP\_OFF 获取远地址偏移量 ■

**用法** #include <dos.h>  
unsigned FP\_OFF(void far \*p);

**原型在** dos.h.

**说明** FP\_OFF 宏用于取得或设置远指针 \*p 的偏移量。

**返回值** FP\_OFF 返回一个无符号整数代表偏移量。

**可移植性** 只适用于 DOS。

**参见** FP\_SEG, MK\_FP, movedata, segread。

**示例**

```
/* FP_OFF */
#include <dos.h>
#include <stdio.h>
int main(void)
```

```

{
 char *str = "fpoff.c";
 printf("The offset of this file name in memory\
 is: %Fp\n", FP_OFF(str));
 return 0;
}

/* FP_SEG */
#include <dos.h>
#include <stdio.h>
int main(void)
{
 char *filename = "fpseg.c";
 printf("The segment of this file in memory\
 is: %Fp\n", FP_SEG(filename));
 return(0);
}

/* MK_FP */
#include <dos.h>
#include <graphics.h>
int main(void)
{
 int gd, gm, i;
 unsigned int far *screen;
 detectgraph(&gd, &gm);
 if (gd == HERCMONO)
 screen = MK_FP(0xB000, 0);
 else
 screen = MK_FP(0xB800, 0);
 for (i=0; i<26; i++)
 screen[i] = 0x0700 + ('a' + i);
 return 0;
}

```

## ■ \_fpret 重新初始化浮点数学包 ■

用 法 #include <float.h>

void \_fpret(void);

原 型 在 float.h

说 明 \_fpret 重新初始化浮点数字包, 本函数通常与 signal、system 或 exec...、spawn... 函数配合使用。

在 DOS 中, 如果程序中使用 80×87 协处理器, 则子进程(由 system、exec...、spawn... 函数执行)可以改变父进程的浮点状态。

注意: 如果使用了 80×87, 则必须注意以下几点:

- 在计算一个浮点表达式时,不得调用 `system`、`exec...` 或 `spawn...` 函数。
- 如果子进程有可能用 80×87 执行了一次浮点操作,则在调用 `system`、`exec...` 或 `spawn...` 之后,必须调用 `fpreset` 复位浮点状态。

返回值 本函数无返回值。

参见 `_clear87`、`_control87`、`exec...`、`spawn...`、`status87`、`system`

示例 `#include <stdio.h>`

```
#include <float.h>
#include <setjmp.h>
#include <signal.h>
#include <process.h>
#include <conio.h>

jmp_buf reenter;
/* define a handler for trapping floating
 point errors */
void float_trap(int sig)
{
 printf("Trapping floating point error,");
 printf("signal is %d\n", sig);
 printf("Press a key to continue\n");
 getch();
}
/*
 reset the 8087 chip or emulator to clear
 any extraneous garbage
*/
_fpreset();
/* return to the problem spot */
longjmp(reenter, -1);
}

int main(void)
{
 float one = 3.14, two = 0.0;
 /*
 install signal handler for floating point
 exception
 */
 if (signal(SIGFPE, float_trap) == SIG_ERR)
 {
 printf("error installing signal handler\n");
 exit(3);
 }
 printf("Generating a math error,");
 printf("press a key\n");
```

```

 getch();
 if (setjmp(reenter) == 0)
 one /= two;
 printf("Returned from signal trap\n");
 return 0;
}

```

## ■ fprintf 传送输出到一个流中

用 法 #include <stdio.h>

```
int fprintf(FILE *stream, const char *format[, argument, ...]);
```

原 型 在 stdio.h

说 明 fprintf 接受一组参数,每个参数都根据 format 所指串中的一个格式指示符进行格式化,并输出格式化数据到一个流。参数的数目和格式指示符的数目必须相同。

返 回 值 返回输出的字节数;如果有错误,返回 EOF。

可移植性 适用于 UNIX 系统,在 ANSI C 中定义,与 Kernighan 和 Ritchie 的定义兼容。

参 见 cprintf, fscanf, printf, putc, sprintf。

示 例 #include <stdio.h>

```

int main(void)
{
 FILE *stream;
 int i = 100;
 char c = 'C';
 float f = 1.234;
 /* open a file for update */
 stream = fopen("DUMMY.FIL", "w+");
 /* write some data to the file */
 fprintf(stream, "%d %c %f", i, c, f);
 /* close the file */
 fclose(stream);
 return 0;
}

```

## ■ FP\_SEG 获取远地址段值

用 法 #include <dos.h>

```
unsigned FP_SEG(void far *p);
```

原 型 在 dos.h

说 明 FP\_SEG 是一用于取得和设置远指针 \*p 段地址值的宏。

返 回 值 返回一个无符号整数代表段地址值。

可移植性 只适用于 DOS。

参 见 FP\_OFF, MK\_FP, movedata, segread

示 例 /\* FP\_OFF \*/

```
#include <dos.h>
#include <stdio.h>
int main(void)
{
 char *str = "fpoff.c";
 printf("The offset of this file name in memory\
 is: %Fp\n", FP_OFF(str));
 return 0;
}
```

/\* FP\_SEG \*/

```
#include <dos.h>
#include <stdio.h>
int main(void)
{
 char *filename = "fpseg.c";
 printf("The segment of this file in memory\
 is: %Fp\n", FP_SEG(filename));
 return(0);
}
```

/\* MK\_FP \*/

```
#include <dos.h>
#include <graphics.h>
int main(void)
{
 int gd, gm, i;
 unsigned int far *screen;
 detectgraph(&gd, &gm);
 if (gd == HERCMONO)
 screen = MK_FP(0xB000, 0);
 else
 screen = MK_FP(0xB800, 0);
 for (i=0; i<26; i++)
 screen[i] = 0x0700 + ('a' + i);
 return 0;
}
```

## ■ fputc 送一个字符到一个流中 ■

用 法 #include <stdio.h>

fputc(int c, FILE \*stream);

原 型 在 stdio.h

说 明 fputc 输出字符 c 到指定的流。

**返回值** 调用成功时,返回字符 c;发生错误时,返回 EOF。

**可移植性** 适用于 UNIX 系统,在 ANSI C 中有定义。

**参见** fgetc,putc。

**示例**

```
#include <stdio.h>

int main(void)
{
 char msg[] = "Hello world";
 int i = 0;
 while (msg[i])
 {
 fputc(msg[i], stdout);
 i++;
 }
 return 0;
}
```

## ■ fputc 送一个字符到标准输出 ■

**用法** #include <stdio.h>

int fputc(int c);

**原型在** stdio.h

**说明** fputc 输出字符 c 到 stdout,fputc(c)等同于 fputc(c,stdout)。

**返回值** 成功调用时,返回字符 c。有错误时,返回 EOF。

**可移植性** fputc 适用于 UNIX 系统。

**参见** fgetchar,putchar。

**示例**

```
#include <stdio.h>

int main(void)
{
 char msg[] = "This is a test";
 int i = 0;
 while (msg[i])
 {
 fputc(msg[i]);
 i++;
 }
 return 0;
}
```

## ■ fputs 送一个字符串到流中 ■

**用法** #include <stdio.h>

int fputs(const char \*s, FILE \*stream);

**原型在** stdio.h

**说 明** fputs 拷贝以空字符终结的字串 s 到给定的输出流,它不加换行符,终结空字符不拷贝。

**返回值** 执行成功时,fputs 返回所写的最后一个字符,否则返回 EOF。

**参 见** fgets,gets,puts.

**示 例**

```
#include <stdio.h>

int main(void)
{
 /* write a string to standard output */
 fputs("Hello world\n", stdout);
 return 0;
}
```

## ■ fread 从流中读数据 ■

**用 法** #include <stdio.h>  
size\_t fread(void \* ptr, size\_t size, size\_t n, FILE \* stream);

**原型在** stdio.h

**说 明** fread 从指定输入流 stream 中读取 n 项数据,每一项数据长度为 size 字节,放到由 ptr 所指的块中。读的字节总数为 n \* size。

**返回值** 调用成功时,返回实际读的数据项数(不是字节数)。在遇到文件结束或出错时,fread 返回一短计数值(可能为 0)。

**可移植性** 适用于 UNIX 系统,在 ANSI C 中有定义。

**参 见** fopen,fwrite,printf,read

**示 例**

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

int main(void)
{
 FILE * stream;
 char msg[] = "this is a test";
 char buf[20];
 if ((stream = fopen("DUMMY.FIL", "w+"))
 == NULL)
 {
 fprintf(stderr, "Cannot open output file.\n");
 return 1;
 }
 /* write some data to the file */
 fwrite(msg, strlen(msg)+1, 1, stream);
 /* seek to the beginning of the file */
 fseek(stream, SEEK_SET, 0);
 /* read the data and display it */
 fread(buf, strlen(msg)+1, 1, stream);
}
```



```
printf("%s\n", buf);
fclose(stream);
return 0;
}
```

## ■ free 释放已分配的内存

**用 法** #include <alloc.h>  
void free(void \*block);

**原 型** 在 stdlib.h, alloc.h

**说 明** 释放由 calloc、malloc 或 realloc 函数调用所分配的内存。

**返 回 值** 无。

**可移植性** 适用于 UNIX 系统, 在 ANSI C 中有定义。

**参 见** calloc, freemem, malloc, realloc, strdup.

**示 例**

```
#include <string.h>
#include <stdio.h>
#include <alloc.h>
int main(void)
{
 char *str;
 /* allocate memory for string */
 str = malloc(10);
 /* copy "Hello" to string */
 strcpy(str, "Hello");
 /* display string */
 printf("String is %s\n", str);
 /* free memory */
 free(str);
 return 0;
}
```

## ■ freemem 释放先前分配的 DOS 内存

**用 法** #include <dosh.h>  
int freemem(unsigned segx);

**原 型** 在 dos.h

**说 明** freemem 释放以前用 allocmem 调用所分配的内存。segx 是该块的段地址。

**返 回 值** 如果调用成功, freemem 返回 0; 否则, 返回 -1, 并置 errno 为:

ENOMEM 无足够内存

**可移植性** 只适用于 DOS.

**参 见** allocmem, free.

**示 例**

```
#include <dos.h>
#include <alloc.h>
```

```
#include <stdio.h>
int main(void)
{
 unsigned int size, segp;
 int stat;
 size = 64; /* (64 x 16) = 1024 bytes */
 stat = allocmem(size, &segp);
 if (stat < 0)
 printf("Allocated memory at segment: %x\n", segp);
 else
 printf("Failed, maximum number of\
 paragraphs available is %u\n", stat);
 freemem(segp);
 return 0;
}
```

## ■ freopen 把一个新文件同一个打开的流相联 ■

用 法 #include <stdio.h>

FILE freopen(const char \* filename, const char \* mode, FILE \* stream);

原 型 在 stdio.h

说 明 freopen 用指定文件名代替打开的流。不管打开是否成功,原流 stream 都被关闭。在改变与 stdin、stdout 或 stderr 相联的文件时, freopen 非常有用。freopen 调用中的字符串 mode 可以是下列值之一:

|    |                                                                                                               |
|----|---------------------------------------------------------------------------------------------------------------|
| r  | 以只读方式打开;                                                                                                      |
| w  | 以写方式打开;                                                                                                       |
| a  | 附加。打开用于在文件末尾写,当文件不存在时,创建用于写;                                                                                  |
| r+ | 打开一已存在文件用于更新(读或写);                                                                                            |
| w+ | 创建一新文件用于写;                                                                                                    |
| a+ | 打开用于附加。打开(如果文件不存在,则创建)文件用于在末尾添加。为指明一给定文件用文本方式打开或创建,可以在 mode 后面加上 t(如 rt、w+t);如果以二进制地址方式打开或创建,则加上 b(如 wb、a+b)。 |

若 mode 中没有给出 t 或 b,则全局变量 \_fmode 决定打开或创建方式。常量 O\_BINARY 和 O\_TEXT 在 fcntl.h 中定义。

当文件打开用于更新时,在结果流上既可进行输入也可进行输出。但如果没有使用 fseek 或 rewind 调用,输出不能直接跟在输入后面,同样,如果没有使用 fseek 或 rewind 调用,输入也不能直接跟在输出后面,在遇到文件结束时不能进行输入。

返 回 值 调用成功时, freopen 返回参数 stream; 否则, 返回 NULL。

可移植性 适用于 UNIX 系统,在 ANSI C 中有定义。

参 见 fclose, fdopen, fopen, open, setmode.

示 例 #include <stdio.h>

```
int main(void)
{
 /* redirect standard output to a file */
 if (freopen("OUTPUT.FIL", "w", stdout) == NULL)
 fprintf(stderr, "error redirecting stdout\n");
 /* this output will go to a file */
 printf("This will go into a file.");
 /* close the standard output stream */
 fclose(stdout);
 return 0;
}
```

## ■ frexp 对双精度数进行科学计数

用 法 #include <math.h>

double frexp(double x, int \* exponent);

原 型 在 math.h

说 明 frexp 计算出尾数 m(大于等于 0.5、小于 1 的双精度数)和指数 n(整数)的值,使得  $x = m \times 2^n$  ( $x$  为一个给定的双精度数),并把指数  $n$  存入由 `exponent` 所指的整型变量中。

返 回 值 frexp 返回尾数  $m$ 。

可以通过 `matherr` 函数修改 `frexp` 的错误处理程序。

可移植性 适用于 UNIX 系统,在 ANSI C 中有定义。

参 见 exp, ldexp.

示 例 #include <math.h>

```
#include <stdio.h>
int main(void)
{
 double mantissa, number;
 int exponent;
 number = 8.0;
 mantissa = frexp(number, &exponent);
 printf("The number %lf is ", number);
 printf("%lf times two to the ", mantissa);
 printf("power of %d\n", exponent);
 return 0;
}
```

## ■ fscanf 格式化输入

用 法 #include <stdio.h>

```
int fscanf(FILE * stream, const char * format [,address,...]);
```

原 型 在 `stdio.h`

说 明 `fscanf` 从一个流中扫描输入字段,一次扫描一个字符。一次扫描一个字符后,每个输入字段根据 `format` 所指格式指示符被格式化后,把输入字段存在 `format` 后面由地址参数给出的位置上。格式指示符的数目必须与输入字段的数目相等。由于多种原因,`fscanf` 在遇到正常结束符(空白字符)前,可能停止扫描一特定字段或完全终止。关于可能原因的讨论,请参见 `scanf` 函数。

返 回 值 `fscanf` 返回成功扫描、转换和存储的输入字段数,被扫描但不存储的字段不算在内。如果 `fscanf` 试图读文件末尾,返回值是 EOF,如果没有字段被存储,则返回值为 0。

可移植性 适用于 UNIX 系统,在 Kernighan 和 Ritchie 书中作了定义,并与 ANSI C 兼容。

参 见 `atoi`,`cscanf`,`fprintf`,`printf`,`printf`,`scanf`,`sscanf`,`vfscanf`,`vscanf`,`vsscanf`。

示 例

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
 int i;
 printf("Input an integer: ");
 /* read an integer from the
 standard input stream */
 if (fscanf(stdin, "%d", &i))
 printf("The integer read was: %i\n", i);
 else
 {
 fprintf(stderr, "Error reading an integer from stdin. \n");
 exit(1);
 }
 return 0;
}
```

## ■ `fseek` 移动文件指针

用 法 `#include <stdio.h>`

```
int fseek(FILE * stream, long offset, int whence);
```

原 型 在 `stdio.h`

说 明 `fseek` 设置与流 `xstream` 相联的文件指针到新的位置,该位置与 `whence` 所给定的文件位置的距离为 `offset` 个字节。对文本方式下的流,`offset` 的值是 0 或者等于函数 `ftell` 的返回值。

`whence` 必须是 0、1、2 中的一个,分别代表以下三个符号常量(在 `stdio.h` 中定义):

`whence`

文件位置

|             |          |
|-------------|----------|
| SEEK_SET(0) | 文件开始     |
| SEEK_CUR(1) | 文件当前指针位置 |
| SEEK_END(2) | 文件末尾     |

fseek 忽略由 ungetc 退回的任何字符。

fseek 用于输入输出流, lseek 用于输入输出文件名柄。

在调用了 fseek 之后在更新文件上的下一个操作可以是输入也可以是输出。

**返回值** 如果指针成功地移动了, fseek 返回 0, 否则返回非零值。

注意, fseek 可能返回 0, 表示指针成功地移动了, 然而实际上并没有移动。这是因为 DOS 只是重置指针, 并不检验定位。fseek 仅在未打开文件或设备时返回错误的代码。

**可移植性** 适用于所有 UNIX 系统, 在 ANSI C 中有定义。

**参见** fgetpos, fopen, fsetpos, ftell, lseek, rewind, setbuf, tell

**示例**

```
#include <stdio.h>

long filesize(FILE * stream);

int main(void)
{
 FILE * stream;
 stream = fopen("MYFILE.TXT", "w+");
 fprintf(stream, "This is a test");
 printf("Filesize of MYFILE.TXT is %ld bytes\n", filesize(stream));
 fclose(stream);
 return 0;
}

long filesize(FILE * stream)
{
 long curpos, length;
 curpos = ftell(stream);
 fseek(stream, 0L, SEEK_END);
 length = ftell(stream);
 fseek(stream, curpos, SEEK_SET);
 return length;
}
```

## ■ fsetpos 定位文件指针 ■

**用法** #include <stdio.h>

```
int fsetpos(FILE * stream, const fpos_t * pos);
```

**原型在** stdio.h

**说明** fsetpos 把与 stream 相联的文件指针置于新的位置。这是前次针对此次 stream 调用 fgetpos 时所得的值。fsetpos 清除 stream 所指文件的文件结束标志, 并消除对

该文件的所有 `ungetc` 操作的影响。在调用 `fsetpos` 操作后,文件的下一操作可以是输入或输出。

**返回值** 若调用成功,`fsetpos` 返回 0;若失败,则返回非零值,并且置全局变量 `errno` 为非零值。

**可移植性** 与 ANSI C 兼容。

**参见** `fgetpos`, `fseek`, `ftell`。

**示例** `#include <stdlib.h>`

`#include <stdio.h>`

`void showpos(FILE * stream);`

`int main(void)`

`{`

`FILE * stream;`

`fpos_t filepos;`

`/* open a file for update */`

`stream = fopen("DUMMY.FIL", "w+");`

`/* save the file pointer position */`

`fgetpos(stream, &filepos);`

`/* write some data to the file */`

`fprintf(stream, "This is a test");`

`/* show the current file position */`

`showpos(stream);`

`/* set a new file position, display it */`

`if (fsetpos(stream, &filepos) == 0)`

`showpos(stream);`

`else`

`{`

`fprintf(stderr, "Error setting file pointer. \n");`

`exit(1);`

`}`

`/* close the file */`

`fclose(stream);`

`return 0;`

`}`

`void showpos(FILE * stream)`

`{`

`fpos_t pos;`

`/* display the current file pointer`

`position of a stream */`

`fgetpos(stream, &pos);`

`printf("File position: %ld\n", pos);`

## ■ fstat 获取已打开文件的信息 ■

**用 法** #include <sys/stat.h>

int fstat(int handle, struct stat \*statuf);

**原 型** 在 sys/stat.h

**说 明** fstat 把一个与句柄 handle 相联的打开文件或目录的信息存到 stat 结构中。

statbuf 指向 stat 结构(在 sys/stat.h 中定义),它包含以下几个字段:

|          |                            |
|----------|----------------------------|
| st_mode  | 给出打开文件方式信息的位屏蔽;            |
| st_dev   | 包含文件或文件句柄(若文件在设备中)的磁盘驱动器号; |
| st_rdev  | 同 st_dev;                  |
| st_nlink | 置为整型常数 1;                  |
| st_size  | 已打开文件的大小(按字节计算);           |
| st_atime | 已打开文件的最近修改时间;              |
| st_mtimt | 同 st_atime;                |
| st_ctime | 同 st_atime。                |

stat 结构还包含这里没有提到的三个字段,它们包含在 DOS 下无意义的值。位屏蔽给出有关打开文件方式的信息,它包括以下几位:

下列一位被设置:

|         |                    |
|---------|--------------------|
| S_IFCHR | 如果 handle 指向一个外设   |
| S_IFREG | 如果 handle 指向一个普通文件 |

以下一位或两位被设置:

|          |             |
|----------|-------------|
| S_IWRITE | 如果用户有写文件的权限 |
| S_IRREG  | 如果用户有读文件的权限 |

位屏蔽还包括读/写位,这些是根据文件的权限方式而设置的。

**返 回 值** 在成功检索了打开文件的信息后,返回 0;在出错时(无法得到信息)返回 -1,并置全局变量 err no 为:

|       |         |
|-------|---------|
| EBADF | 无效文件句柄。 |
|-------|---------|

**可移植性** 适用于 UNIX 系统,在 ANSI C 中有定义。

**参 见** access, chmod, stat

**示 例**

```
#include <sys/stat.h>
#include <stdio.h>
#include <time.h>
```

```
int main(void)
{
 struct stat statbuf;
 FILE *stream;
 /* open a file for update */
 if ((stream = fopen("DUMMY.FIL", "w+")) == NULL)
```

```

 {
 fprintf(stderr, "Cannot open output file. \n");
 return(1);
 }

 fprintf(stream, "This is a test");
 fflush(stream);
 /* get information about the file */
 fstat(fileno(stream), &statbuf);
 fclose(stream);
 /* display the information returned */
 if (statbuf.st_mode & S_IFCHR)
 printf("Handle refers to a device. \n");
 if (statbuf.st_mode & S_IFREG)
 printf("Handle refers to an ordinary file. \n");
 if (statbuf.st_mode & S_IRREAD)
 printf("User has read permission on file. \n");
 if (statbuf.st_mode & S_IWRITE)
 printf("User has write permission on file. \n");
 printf("Drive letter of file: %c\n", 'A' + statbuf.st_dev);
 printf("Size of file in bytes: %ld\n", statbuf.st_size);
 printf("Time file last opened: %s\n", ctime(&statbuf.st_ctime));
 return 0;
}

```

## ■ ftell 返回当前文件指针 ■

**用 法** #include <stdio.h>

long int ftell(FILE \* stream);

**原 型** 在 stdio.h

**说 明** ftell 返回流 stream 中的当前文件指针, 如果文件是二进制的, 则偏移量是从文件头开始算起的字节数。ftell 的返回值可用于其后的 fseek 调用中。

**返 回 值** ftell 在调用成功后, 返回文件的当前指针位置; 出错时, 返回 -1L 并置全局变量 errno 为正值。

**可移植性** 适用于所有 UNIX 系统, 在 ANSI C 中有定义。

**参 见** fgetpos, fseek, fsetpos, lseek, rewind, tell.

**示 例** #include <stdio.h>

```

int main(void)
{
 FILE * stream;
 stream = fopen("MYFILE.TXT", "w+");
 fprintf(stream, "This is a test");
 printf("The file pointer is at byte %ld\n", ftell(stream));
 fclose(stream);
}

```



```
return 0;
```

```
}
```

## ■ ftime 把当前时间存入 timeb 结构中

用 法 #include <sys\timeb.h>

```
void ftime(struct timeb * buf);
```

原 型 在 sys\timeb.h

说 明 ftime 确定当前时间,并将其存入 buf 所指的 timeb 型结构中。timeb 结构包含四个字段:time、millitm、timezone、dstflag:

```
struct timeb {
 long time;
 short millitm;
 short timezone;
 short dstflag;
};
```

- time 为以秒为单位的,从 1970 年 1 月 1 日格林威治时间(GMT)00:00:00 算起的当前时间。
- millitm 是用毫秒表示的秒数的小数部分。
- timezone 是 GMT 和当地时间相差的分钟数,该值从 GMT 向西计算。ftime 从 tzset 设置的全局变量 timezone 中得到这个字段。
- dstflag 被用来指示在时间计算中是否考虑夏时制。

注意:ftime 本身调用了 tzset,因此使用 ftime 时不需用户来指明调用 tzset。

返 回 值 无。

可移植性 适用于 UNIX 系统 V。

参 见 asctime,ctime,gmtime,localtime,stime,time,tzset

示 例 #include <stdio.h>

```
#include <stdlib.h>
```

```
#include <time.h>
```

```
#include <sys\timeb.h>
```

```
/* pacific standard & daylight savings */
```

```
char * tzstr = "TZ=PST8PDT";
```

```
int main(void)
```

```
{
```

```
 struct timeb t;
```

```
 putenv(tzstr);
```

```
 tzset();
```

```
 ftime(&t);
```

```
 printf("Seconds since 1/1/1970 GMT: %ld\n", t.time);
```

```
 printf("Thousandths of a second: %d\n", t.millitm);
```

```

 printf("Difference between local time and GMT: %d\n", t.timezone);
 printf("Daylight savings in effect (1) not (0): %d\n", t.dstflag);
 return 0;
}

```

## ■ fwrite 把参数写人流中

**用 法** #include <stdio.h>

```
size_t fwrite(const void * ptr, size_t size, size_t n, FILE * stream);
```

**原 型 在** stdio.h

**说 明** fwrite 附加 n 个数据项(每个数据项长度为 size 个字节)到指定的输出文件后,数据从 ptr 处开始添加。所写的字节总数是 n \* size。ptr 是指向任意对象的指针。

**返 回 值** 调用成功时, fwrite 返回实际写的数据项数(不是字节数);出错时,返回一短整型数值(可能为 0)。

**可移植性** 适用于所有 UNIX 系统,在 ANSI C 中有定义。

**参 见** fopen, fread

**示 例** #include <stdio.h>

```

struct mystruct
{
 int i;
 char ch;
};

int main(void)
{
 FILE * stream;
 struct mystruct s;
 if ((stream = fopen("TEST. $$$", "wb")) == NULL)
 /* open file TEST. $$$ */
 {
 fprintf(stderr, "Cannot open output file.\n");
 return 1;
 }
 s.i = 0;
 s.ch = 'A';
 fwrite(&s, sizeof(s), 1, stream); /* write struct s to file */
 fclose(stream); /* close file */
 return 0;
}

```

## ■ gcvt 把浮点数转换为字符串

**用 法** #include <stdlib.h>

```
char * gcvt(double value,int ndec,char * buf);
```

原型在 `stdlib.h`

说明 `gcvt` 把 `value` 转换为以空字符终结的 ASCII 字符串,并把该串放在 `buf` 中。如果可能的话,它产生 FORTRAN F 格式的 `ndec` 位有效数字串,否则返回 `printf` E 格式的数字串(打印就绪)。`gcvt` 去掉结尾的 0。

返回值 返回 `buf` 所指的字符串的地址。

可移植性 适用于 UNIX,在 ANSI C 中没有定义,建议不要用于可移植性程序。

参 见 `ecvt`,`fcvt`,`sprintf`

示 例

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
 char str[25];
 double num;
 int sig = 5; /* significant digits */
 /* a regular number */
 num = 9.876;
 gcvt(num, sig, str);
 printf("string = %s\n", str);
 /* a negative number */
 num = -123.4567;
 gcvt(num, sig, str);
 printf("string = %s\n", str);
 /* scientific notation */
 num = 0.678e5;
 gcvt(num, sig, str);
 printf("string = %s\n", str);
 return(0);
}
```

## ■ geninterrupt 产生软中断

用 法 `#include <dos.h>`  
`void geninterrupt(int intr_num);`

原型在 `dos.h`

说明 `geninterrupt` 宏为由 `intr_num` 指定的中断触发一个软件陷阱。调用后所有寄存器的状态均由调用的中断所决定。

注意:中断可能使 C 所使用的寄存器处于不可预测的状态。

返回值 无。

可移植性 只适用于 8086 结构。

参 见 `bdos`,`bdosptr`,`disable`,`enable`,`getvect`,`int86`,`int86x`,`intdos`,`intdosx`,`intr`

示 例 `#include <conio.h>`

```

#include <dos.h>
/* function prototype */
void writechar(char ch);
int main(void)
{
 clrscr();
 gotoxy(80,25);
 writechar(' * ');
 getch();
 return 0;
}
/*
outputs a character at the current cursor
position using the video BIOS to avoid
the scrolling of the screen when writing
to location (80,25).
*/
void writechar(char ch)
{
 struct text_info ti;
 /* grab current text settings */
 gettextinfo(&ti);
 /* interrupt 0x10 sub-function 9 */
 _AH = 9;
 /* character to be output */
 _AL = ch;
 _BH = 0; /* video page */
 _BL = ti.attribute; /* video attribute */
 _CX = 1; /* repetition factor */
 geninterrupt(0x10); /* output the char */
}

```

## ■ getarccoords 取得最后一次调用 arc 的坐标 ■

用 法 #include <graphics.h>

void far getarccoords (struct arccoordstype far \* arccoords);

原 型 在 graphics.h

说 明 getarccoords 函数将有关上次调用 arc 的信息填入到由远指针 arccoords 所指的 arccoordstype 结构中,后者在 graphics.h 中定义如下:

```

struct arccoordstype {
 int x, y;
 int xstart, ystart, xend, yend;
};

```

结构中的成员用来说明弧线(由函数 arc 画出)的中心点(x,y)起始位置(xstart, ystart)以及终止位置(xend, yend)。如果需要在弧线的末尾引一条线,这些信息十分有用。

返回值 无。

可移植性 只适用于 Turbo C,而且只能在装配有图形适配器的 IBMPC 及其兼容机上使用。

参 见 arc, fillellipse, sector

```

示 例 #include <graphics.h>
 #include <stdlib.h>
 #include <stdio.h>
 #include <conio.h>
 int main(void)
 {
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 struct arcinfo arcinfo;
 int midx, midy;
 int stangle = 45, endangle = 270;
 char sstr[80], estr[80];
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 /* an error occurred */
 if (errorcode != grOk)
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 }
 /* terminate with an error code */
 exit(1);
 }
 midx = getmaxx() / 2;
 midy = getmaxy() / 2;
 /* draw arc and get coordinates */
 setcolor(getmaxcolor());
 arc(midx, midy, stangle, endangle, 100);
 getarcinfo(&arcinfo);
 /* convert arc information into strings */
 sprintf(sstr, " * - (%d, %d)", arcinfo.xstart, arcinfo.ystart);
 sprintf(estr, " * - (%d, %d)", arcinfo.xend, arcinfo.yend);
 /* output the arc information */

```

```

 outtextxy(arcinfo.xstart,
 arcinfo.ystart, sstr);
 outtextxy(arcinfo.xend,
 arcinfo.yend, estr);
 /* clean up */
 getch();
 closegraph();
 return 0;
}

```

## ■ getspectratio 返回当前图形模式的纵横比 ■

用 法 #include <graphics.h>

```
void far getspectratio(int far *xasp, int far *yasp);
```

原 型 在 graphics.h

说 明 y 轴因子 (\*yasp) 一般设置为 10000。除了 VGA, 在所有的图形适配器上, \*xasp(x 轴因子) 都比 \*yasp 小, 因为像素的高度总大于其宽度。在像素是“方形”的 VGA 上, \*xasp=yasp。一般说来, \*yasp 和 \*xasp 之间的关系大致可表示为:

```

*yasp=10000
*xasp<=10000

```

调用 getspectratio 可得到 \*xasp 和 \*yasp 的值。

返 回 值 无。

可移植性 在 Turbo Pascal 中有一相似的例行程序。

参 见 arc, circle, ellipse, fillellipse, pieslice, sector, setspectratio

示 例

```

#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int xasp, yasp, midx, midy;
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 /* an error occurred */
 if (errorcode != grOk)
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 }
}

```

```

 printf("Press any key to halt:");
 getch();
/* terminate with an error code */
 exit(1);
}

midx = getmaxx() / 2;
midy = getmaxy() / 2;
setcolor(getmaxcolor());
/* get current aspect ratio settings */
getaspectratio(&xasp, &yasp);
/* draw normal circle */
circle(midx, midy, 100);
getch();
/* draw wide circle */
cleardevice();
setaspectratio(xasp/2, yasp);
circle(midx, midy, 100);
getch();
/* draw narrow circle */
cleardevice();
setaspectratio(xasp, yasp/2);
circle(midx, midy, 100);
/* clean up */
getch();
closegraph();
return 0;
}

```

## ■ getbkcolor 返回当前背景颜色 ■

用 法 #include <graphics.h>

int far getbkcolor(void);

原 型 在 graphics.h

说 明 getbkcolor 返回当前背景颜色。

返 回 值 返回当前背景颜色。

可移植性 只适用于 Turbo C, 而且只能在装有图形适配器的 IBMPC 及其兼容机上使用。

参 见 getcolor, getmaxcolor, getpalette, setbkcolor

示 例 #include <graphics.h>

#include <stdlib.h>

#include <string.h>

#include <stdio.h>

#include <conio.h>

int main(void)

```
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int bkcolor, midx, midy;
 char bkname[35];
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 /* an error occurred */
 if (errorcode != grOk)
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 }
 /* terminate with an error code */
 exit(1);
}

midx = getmaxx() / 2;
midy = getmaxy() / 2;
setcolor(getmaxcolor());
/* for centering text on the display */
settextjustify(CENTER_TEXT, CENTER_TEXT);
/* get the current background color */
bkcolor = getbkcolor();
/* convert color value into a string */
itoa(bkcolor, bkname, 10);
strcat(bkname, " is the current background color.");
/* display a message */
outtextxy(midx, midy, bkname);
/* clean up */
getch();
closegraph();
return 0;
}
```

## ■ getc 从流中取字符 ■

用 法 #include <stdio.h>

int getc(FILE \*stream);

原 型 在 stdio.h

说 明 getc 是一个返回指定输入流 stream 中一个字符的宏,它移动 stream 的文件指针,并使之指向下一个字符。



**返回值** 在调用成功时,getc 返回所读的字符,它已被转换为无符号扩展的整型值。当遇到文件结束或出错时,它返回 EOF。

**可移植性** 只适用于 UNIX 系统,在 ANSI C 中有定义,且与 Kernighan 和 Ritchie 的定义兼容。

**参见** fgetc, getch, getchar, getche, gets, putc, putchar, ungetc

**示例**

```
#include <stdio.h>

int main(void)
{
 char ch;
 printf("Input a character,");
 /* read a character from the
 standard input stream */
 ch = getc(stdin);
 printf("The character input was, '%c'\n", ch);
 return 0;
}
```

## ■ getch 获取 control-break 状态 ■

**用法** #include <dos.h>

```
int getchbrk(void);
```

**原型在** dos.h

**说明** getchbrk 使用 DOS 系统调用 0x33 返回 contro-break 的当前检测状态。

**返回值** 如果 control-break 检测状态为 off,则函数返回 0;如果检测状态为 on,返回 1。

**可移植性** 只适用于 DOS 系统。

**参见** ctrlbrk, setcbrk

**示例**

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

int main(void)
{
 if (getchbrk())
 printf("Ctrl-brk flag is on\n");
 else
 printf("Ctrl-brk flag is off\n");
 return 0;
}
```

## ■ getch 从键盘无回显地读取一字符 ■

**用法** #include <conio.h>

```
int getch(void);
```

**原型在** conio.h

**说明** getch 直接从键盘读一字符,并且不将该字符显示到屏幕上。

**返回值** 返回从键盘所读入的字符。

**可移植性** 只适用于 DOS 系统。

**参 见** `cgets, cscanf, fgetc, getc, getchar, getche, getpass, kbhit, putc, ungetc`

**示 例**

```
#include <stdio.h>
#include <conio.h>
int main(void)
{
 char ch;
 printf("Input a character,");
 ch = getche();
 printf("\nYou input a '%c'\n", ch);
 return 0;
}
```

## ■ `getchar` 从 `stdin` 流中读取一个字符

**用 法** `#include <stdio.h>`  
`int getchar(void);`

**原 型 在** `stdio.h`

**说 明** `getchar` 是一个返回已指定的 `stdin` 输入流中下一个字符的宏,它一般定义为 `getc(stdin)`。

**返回值** 在调用成功时,`getchar` 返回所读的字符,此时该字符已被转换为无符号扩展的整型值,遇到文件结束或出错时,返回 EOF。

**可移植性** 适用于 UNIX 系统,在 ANSI C 中也有定义,并且与 Kernighan 和 Ritchie 的定义兼容。

**参 见** `fgetc, fgetchar, getc, getch, getche, gets, putc, putchar, scanf, ungetc`

**示 例**

```
#include <stdio.h>
int main(void)
{
 int c;
 /* Note that getchar reads from stdin and
 is line buffered, this means it will
 not return until you press ENTER. */
 while ((c = getchar()) != '\n')
 printf("%c", c);
 return 0;
}
```

## ■ `getche` 从键盘并回显地读取一字符

**用 法** `#include <conio.h>`  
`int getche(void);`

**原 型 在** `conio.h`

**说 明** 从键盘读取一个字符,并通过直接视频输出或 BIOS 将该字符显示在当前文本窗口中。

**返回值** 返回从键盘读取的字符。

**可移植性** 只适用于 DOS 系统。

**参 见** cgets, cscanf, fgetc, getc, getch, getchar, kbhit, putch, ungetc

**示 例**

```
#include <stdio.h>
#include <conio.h>
int main(void)
{
 char ch;
 printf("Input a character:");
 ch = getche();
 printf("\nYou input a '%c'\n", ch);
 return 0;
}
```

## ■ getcolor 返回当前绘图颜色

**用 法** #include <graphics.h>

```
int far getcolor(void);
```

**原 型 在** graphics.h

**说 明** getcolor 返回当前绘图颜色。

绘图颜色是指在画线或其它图形时屏幕象素被设置的值。例如,在 CGA 调色板中包含四种颜色:背景颜色、淡绿色、淡红色和黄色。此时,如果 getcolor 返回 1,则当前绘图颜色为绿色。

**返回值** getcolor 返回当前绘图颜色。

**可移植性** 该函数只适用于 Turbo C;并且只能在装有图形适配器的 IBM PC 及其兼容机上使用。

**参 见** getbkcolor, getmaxcolor, getpalette, setcolor

**示 例**

```
#include <graphics.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <conio.h>
int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int color, midx, midy;
 char colname[35];
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
```

```

/* read result of initialization */
errorcode = graphresult();
/* an error occurred */
if (errorcode != grOk)
{
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt,");
 getch();
/* terminate with an error code */
 exit(1);
}
midx = getmaxx() / 2;
midy = getmaxy() / 2;
setcolor(getmaxcolor());
/* for centering text on the display */
settextjustify(CENTER_TEXT, CENTER_TEXT);
/* get the current drawing color */
color = getcolor();
/* convert color value into a string */
itoa(color, colname, 10);
strcat(colname,
 " is the current drawing color.");
/* display a message */
outtextxy(midx, midy, colname);
/* clean up */
getch();
closegraph();
return 0;
}

```

## ■ getcurdir 读取指定驱动器的当前目录 ■

**用 法** #include <dir.h>

int getcurdir(int drive, char directory);

**原 型 在** dir.h

**说 明** getcurdir 读取指定驱动器 drive 的当前目录。

drive 为一驱动器(0 表示缺省驱动器, 1 表示 A 驱动器, 依此类推)。

directory 指向一个长度为 MAXDIR 的内存区域, 用于存放以空字符(NULL)终结的目录名, 目录名既不包含驱动器号, 也不以反斜杠开始。

**返 回 值** 在调用成功时, 函数 getcurdir 返回 0; 在出错时则返回 -1。

**可移植性** 只适用于 DOS 系统。

**参 见** chdir, getcwd, getdisk, mkdir, rmdir

**示 例** #include <dir.h>

```

#include <stdio.h>
#include <string.h>
char *current_directory(char *path)
{
 strcpy(path, "X:\\"); /* fill string with form of response: X,\ */
 path[0] = 'A' + getdisk(); /* replace X with current drive letter */
 getcurdir(0, path+3); /* fill rest of string with current directory */
 return(path);
}

int main(void)
{
 char curdir[MAXPATH];
 current_directory(curdir);
 printf("The current directory is %s\n", curdir);
 return 0;
}

```

## ■ getcwd 读取当前目录

**用 法** #include<dir.h>

char \*getcwd(char \*buf, int buflen);

**原 型 在** dir.h

**说 明** 函数 getcwd 读取当前目录的完整路径名(包括驱动器号),最长为 buflen 个字节,并把它存在缓冲区 buf 中。如果完整路径名长度(包括空字符终结符)超过 buflen,则出错。

如果 buf 为 NULL,则函数将用 malloc 分配一个 buflen 字节长的缓冲区,以后用户可以将 getcwd 的返回值作为 free 函数的参数,并调用 free 函数来释放该缓冲区。

**返 回 值** getcwd 返回以下值:

- 如果 buf 非空, getcwd 调用成功,返回 buf; 发生错误时,则函数返回 NULL。

- 如果 buf 为 NULL, getcwd 返回指向已分配的内存缓冲区地址。

在发生错误时,全局变量 errno 将置为下列值之一:

|        |        |
|--------|--------|
| ENODEV | 无此设备   |
| ENOMEM | 无足够内存  |
| ERANGE | 结果超出范围 |

**可移植性** 只适用于 DOS 系统。

**参 见** chdir, getcurdir, getdisk, mkdir, rmdir

**示 例** #include <stdio.h>

#include <dir.h>

int main(void)

{

```

char buffer[MAXPATH];
getcwd(buffer, MAXPATH);
printf("The current directory is, %s\n", buffer);
return 0;
}

```

## ■ getdate 读取系统日期 ■

**用 法** #include <dos.h>

void getdate(struct date \*datep);

**原 型** 在 dos.h

**说 明** getdate 把系统当前日期存入由参数 datep 所指向的 date 结构中。  
date 结构定义如下：

```

struct date {
 int da_year; /* current year */
 char da_day; /* day of the month */
 char da_mon; /* month(1=Jan) */
};

```

**返 回 值** 无。

**可移植性** 只适于 DOS 系统。

**参 见** ctime, gettime, setdate, settime

**示 例** #include <dos.h>

#include <stdio.h>

int main(void)

```

{
 struct date d;
 getdate(&d);
 printf("The current year is: %d\n", d.da_year);
 printf("The current day is: %d\n", d.da_day);
 printf("The current month is: %d\n", d.da_mon);
 return 0;
}

```

## ■ getdefaultpalette 返回缺省调色板信息 ■

**用 法** #include <graphics.h>

struct palettetype \* far getdefaultpalette(void);

**原 型** 在 graphics.h

**说 明** getdefaultpalette 将寻找结构 palettetype, 该结构包含调用函数 initgraph 初始化当前驱动程序时的调色板信息。

**返 回 值** 本函数返回一指针, 该指针指向初始化当前驱动程序时设置的缺省调色板。

**可移植性** 只适用于 Turbo C, 并且只能在装有图形适配器的 IBMPC 及其兼容机上使用。

参 见 getpalette, initgraph

```

示 例 #include <graphics.h>
 #include <stdlib.h>
 #include <stdio.h>
 #include <conio.h>
 int main(void)
 {
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int i;
 /* structure for returning palette copy */
 struct palettetype far *pal=(void *) 0;
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 /* an error occurred */
 if (errorcode != grOk)
 {
 printf("Graphics error, %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 }
 /* terminate with an error code */
 exit(1);
 setcolor(getmaxcolor());
 /* return a pointer to the default palette */
 pal = getdefaultpalette();
 for (i=0; i<16; i++)
 {
 printf("colors[%d] = %d\n", i, pal->colors[i]);
 getch();
 }
 /* clean up */
 getch();
 closegraph();
 return 0;
 }

```

## ■ getdfree 读取磁盘空闲空间 ■

用 法 #include &lt;dos.h&gt;

void getdfree(unsigned char drive, struct dfree \* dtable);

原 型 在 dos.h

说 明 getdfree 函数接受磁盘驱动器号 drive(0 表示缺省驱动器,1 表示 A 驱动器,依此类推),并将磁盘有关信息填入由 dtable 所指向的 dfree 结构中。

dfree 结构定义如下:

```
struct dfree {
 unsigned df_avail; /* 可用簇数 */
 unsigned df_total; /* 总簇数 */
 unsigned df_bsec; /* 每个扇区的字节数 */
 unsigned df_sclus; /* 每个簇的扇区数 */
};
```

返 回 值 getdfree 无返回值。出错时,dfree 结构中的 df\_sclus 被置为 0xFFFF。

可移植性 仅用于 DOS 系统。

参 见 getfat, getfatd

示 例

```
#include <stdio.h>
#include <stdlib.h>
#include <dir.h>
#include <dos.h>
int main(void)
{
 struct dfree free;
 long avail;
 int drive;
 drive = getdisk();
 getdfree(drive+1, &free);
 if (free.df_sclus == 0xFFFF)
 {
 printf("Error in getdfree() call\n");
 exit(1);
 }
 avail = (long) free.df_avail
 * (long) free.df_bsec
 * (long) free.df_sclus;
 printf("Drive %c, has %ld bytes available\n", 'A' + drive, avail);
 return 0;
}
```

## ■ getdisk 读取当前磁盘驱动器号 ■

用 法 #include <dir.h>

```
int getdisk(void);
```

原 型 在 dir.h

说 明 getdisk 函数读取当前磁盘驱动器号,并返回一整数:0 表示 A 驱动器,1 表示 B



驱动器, 2 表示 C 驱动器等等。

getdisk 使用 DOS 功能调用 0x19。

**返回值** 返回当前驱动器号。

**可移植性** 仅用于 DOS 系统。

**参 见** getcurdir, getcwd, setdisk

**示 例**

```
#include <stdio.h>
#include <dir.h>
int main(void)
{
 int disk;
 disk = getdisk() + 'A';
 printf("The current drive is: %c\n", disk);
 return 0;
}
```

## ■ getdrivename 返回指向当前图形驱动程序名字的指针 ■

**用 法** #include <graphics.h>  
char \* far getdrivename(void);

**原 型 在** graphics.h

**说 明** 在调用 initgraph 函数结束之后, getdrivename 函数返回当前装入的图形驱动程序名。

**返回值** 返回一指针, 该指针指向含有当前装入的图形驱动程序名的字符串。

**可移植性** 该函数只适用于 Turbo C, 并且只能在装有图形适配器的 IBM PC 及其兼容机上使用。

**参 见** initgraph

**示 例**

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 /* stores the device driver name */
 char * drivename;
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 /* an error occurred */
 if (errorcode != grOk)
```

```

 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 /* terminate with an error code */
 exit(1);
 }
 setcolor(getmaxcolor());
 /* get name of the device driver in use */
 drivename = getdrivename();
 /* for centering text on the screen */
 setttextjustify(CENTER_TEXT, CENTER_TEXT);
 /* output the name of the driver */
 outtextxy(getmaxx() / 2, getmaxy() / 2,
 drivename);
 /* clean up */
 getch();
 closegraph();
 return 0;
}

```

## ■ getdta 读取磁盘传输地址 ■

**用 法** #include <dos.h>

char far \* getdta(void);

**原 型 在** dos.h

**说 明** getdta 返回磁盘传输地址(DTA)的当前设置值。在小型和中型模式中,假定段为当前数据段。若仅用 C 语言,就是这种情况。但如果使用汇编子程序,则可以设置磁盘传输地址为任意硬件地址。

在紧缩、大型或巨型模式中,getdta 返回的地址也是正确的地址,虽然该地址可以定位在程序代码之外。

**返 回 值** getdta 返回一指向当前 DTA 的远指针。

**可移植性** 仅适用于 DOS 系统。

**参 见** fcb(结构), setdta

**示 例** #include <dos.h>

#include <stdio.h>

int main(void)

```

{
 char far * dta;
 dta = getdta();
 printf("The current disk transfer address is, %Fp\n", dta);
 return 0;
}

```

}

## ■ getenv 读取环境变量的当前值 ■

用 法 #include <stdlib.h>

char \*getenv(const char \*name);

原 型 在 stdlib.h

说 明 getenv 返回一给定的环境变量值;环境变量名既可大写也可小写,但是必须包含等号(=)。如果指定的变量不存在,则 getenv 返回一空串。

返 回 值 在调用成功时,getenv 返回与 name 相关的值;如果指定的变量在环境中没有定义,则 getenv 返回一空串。

注意:环境变量不能直接改变,如果想改变环境变量值,必须使用 putenv。

可移植性 对 UNIX 系统也适用,并且在 ANSI C 中也有定义。

参 见 environ(全局变量),getpsp,putenv

示 例 #include <stdlib.h>

#include <stdio.h>

int main(void)

{

char \*s;

s=getenv("COMSPEC"); /\* get the comspec environment parameter \*/

printf("Command processor: %s\n",s);

/\* display comspec parameter \*/

return 0;

}

## ■ getfat 读取指定驱动器的 FAT 信息 ■

用 法 #include <dos.h>

void getfat(unsigned char drive,struct fatinfo \*dtable);

原 型 在 dos.h

说 明 getfat 返回由 drive(0 表示缺省的驱动器,1 表示 A 驱动器,2 表示 B 驱动器等)指定的驱动器的文件分配表信息(FAT)。

dtable 指向待填入内容的 fatinfo 类型结构。

由 getfat 填入的 fatinfo 结构定义如下:

struct fatinfo {

char fi\_sclus; /\* 每个簇的扇区数 \*/

char fi\_fatid; /\* FAT 标识字节 \*/

int fi\_nclus; /\* 簇数 \*/

int fi\_bysec; /\* 每个扇区的字节数 \*/

};

返 回 值 无。

可移植性 只适用于 DOS 系统。

参 见 getdfree, getfatd

示 例 #include <stdio.h>

#include <dos.h>

int main(void)

{

struct fatinfo diskinfo;

int flag = 0;

printf("Please insert disk in drive A\n");

getchar();

getfat(1, &diskinfo);

/\* get drive information \*/

printf("\nDrive A, is ");

switch((unsigned char) diskinfo.fi\_fatid)

{

case 0xFD;

printf("360K low density\n");

break;

case 0xF9;

printf("1.2 Meg high density\n");

break;

default;

printf("unformatted\n");

flag = 1;

}

if (! flag)

{

printf(" sectors per cluster %5d\n", diskinfo.fi\_sclus);

printf(" number of clusters %5d\n", diskinfo.fi\_nclus);

printf(" bytes per sector %5d\n", diskinfo.fi\_bysec);

}

return 0;

}

## ■ getfatd 读取驱动器 FAT 信息

用 法 #include <dos.h>

void getfatd(struct fatinfo \* dtable);

原 型 在 dos.h

说 明 getfatd 读取缺省驱动器的文件分配表(FAT)信息。dtable 指向待填入信息的 fatinfo 类型结构

由 getfatd 填入的 fatinfo 结构定义如下:

```
struct fatinfo {
```

```
 char fi_sclus; /* 每个簇的扇区数
```

```

 char fi_fatid; /* FAT 标识字节 */
 int fi_nclus; /* 簇数 */
 int fi_bysec; /* 每个扇区的字节数 */
 };

```

返回值 无。

可移植性 只适用于 DOS 系统。

参 见 getdfree, getfat

示 例 #include <stdio.h>

#include <dos.h>

int main()

```

{
 struct fatinfo diskinfo;
 /* get default drive information */
 getfatd(&diskinfo);
 printf("\nDefault Drive:\n");
 printf("sectors per cluster: %5d\n", diskinfo.fi_sclus);
 printf("FAT ID byte: %5X\n", diskinfo.fi_fatid & 0xFF);
 printf("number of clusters %5d\n", diskinfo.fi_nclus);
 printf("bytes per sector %5d\n", diskinfo.fi_bysec);
 return 0;
}

```

## ■ getfillpattern 将用户定义的填充模式拷贝到内存

用 法 #include <graphics.h>

void far getfillpattern(char far \* pattern);

原 型 在 graphics.h

说 明 本函数把用户定义的由 setfillpattern 函数设置的填充模式拷贝到 \* pattern 所指的 8 字节区域中。

pattern 是一指向 8 字节连续内存区域的指针, 这 8 个字节分别与该目录下的 8 个像素相对应。只要其中某个模式的某一位被置为 1, 则对应像素将被画出。如下的用户填充模式代表一个检测板:

```

char checkboard [8]={
 0xAA, 0x55, 0xAA, 0x55, 0xAA, 0x55
};

```

返回值 无。

可移植性 只适用于 Turbo C。且仅用于配有图形适配器的 IBM PC 及其兼容机上。

参 见 getfillsettings, setfillpattern

示 例 #include <graphics.h>

#include <stdlib.h>

#include <stdio.h>

```

#include <conio.h>
int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int maxx, maxy;
 char pattern[8] = {0x00, 0x70, 0x20, 0x27, 0x25, 0x27, 0x04, 0x04};
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
 }
 maxx = getmaxx();
 maxy = getmaxy();
 setcolor(getmaxcolor());
 /* select a user defined fill pattern */
 setfillpattern(pattern, getmaxcolor());
 /* fill the screen with the pattern */
 bar(0, 0, maxx, maxy);
 getch();
 /* get the current user defined fill pattern */
 getfillpattern(pattern);
 /* alter the pattern we grabbed */
 pattern[4] -= 1;
 pattern[5] -= 3;
 pattern[6] += 3;
 pattern[7] -= 4;
 /* select our new pattern */
 setfillpattern(pattern, getmaxcolor());
 /* fill the screen with the new pattern */
 bar(0, 0, maxx, maxy);
 /* clean up */
 getch();
 closegraph();
 return 0;
}

```

## ■ getfillsettings 取得当前填充模式和填充颜色的有关信息 ■

用 法 #include <graphics.h>

```
void far getfillsettings(struct fillsettingstype far *fillinfo);
```

原 型 在 graphics.h

说 明 getfillsettings 将有关当前填充模式和填充颜色的信息填进由 fillinfo 所指的 fillsettingstype 结构中,此结构在 graphics.h 中定义如下:

```
struct fillsettingstype {
 int pattern; /* 当前填充模式 */
 int color; /* 当前填充颜色 */
};
```

函数 bar、bar3d、fillpoly、floodfill 和 pieslice 用当前填充模式和当前填充颜色来填充一个区域。有 11 种预定义模式(例如单色填充、交叉影线填充、点填充等),其符号名由 graphics.h 中的枚举类型 fill\_patterns 提供,另外,用户还可以定义自己的填充模式。如果 pattern=12(USER\_FILL),则使用用户定义的填充模式,否则由 pattern 给出预定义模式常量。

| 名称              | 值  | 描述          |
|-----------------|----|-------------|
| EMPTY_FILL      | 0  | 用背景颜色填充     |
| SOLID_FILL      | 1  | 单色填充        |
| LINE_FILL       | 2  | 用—填充        |
| LTSLASH_FILL    | 3  | 用///填充      |
| SLASH_FILL      | 4  | 用粗///填充     |
| BKSLASH_FILL    | 5  | 用粗\\\填充     |
| LTBKSLASH_FILL  | 6  | 用\\填充       |
| HATCH_FILL      | 7  | 用淡影线填充      |
| XHATCH_FILL     | 8  | 用交叉影线填充     |
| INTERLEAVE_FILL | 9  | 用间隔线填充      |
| WIDE_DOT_FILL   | 10 | 用稀疏空白点填充    |
| CLOSE_DOT_FILL  | 11 | 用密集空白点填充    |
| USER_FILL       | 12 | 使用用户定义的填充模式 |

在 graphics.h 中定义的枚举类型 fill\_patterns 给出预定义的填充模式常量,并为

用户自行定义模式提供一个指示标志。

除了 EMPTY\_FILL 使用当前背景颜色外,其余所有模式均使用当前填充颜色进行填充。

返回值 无。

可移植性 只适用于 Turbo C,且只能在装有图形适配器的 IBM PC 及其兼容机上使用。

参 见 getfillpattern, setfillpattern, setfillstyle

示 例 #include <graphics.h>

```

#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 struct arcinfo arcinfo;
 int midx, midy;
 int stangle = 45, endangle = 270;
 char sstr[80], estr[80];
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 /* an error occurred */
 if (errorcode != grOk)
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 }
 /* terminate with an error code */
 exit(1);
}

midx = getmaxx() / 2;
midy = getmaxy() / 2;
/* draw arc and get coordinates */
setcolor(getmaxcolor());
arc(midx, midy, stangle, endangle, 100);
getarcinfo(&arcinfo);
/* convert arc information into strings */
sprintf(sstr, " * - (%d, %d)", arcinfo.xstart, arcinfo.ystart);
sprintf(estr, " * - (%d, %d)", arcinfo.xend, arcinfo.yend);
/* output the arc information */
outtextxy(arcinfo.xstart,
 arcinfo.ystart, sstr);
outtextxy(arcinfo.xend,
 arcinfo.yend, estr);
/* clean up */
getch();
closegraph();
return 0;
}

```



## ■ getftime 读取文件日期和时间 ■

用 法 #include<io.h>

int getftime(int handle, struct ftime \*ftimep);

原 型 在 io.h

说 明 getftime 函数读取与文件句柄 handle 相关联的已打开的磁盘文件的时间和日期,文件的时间和日期存储在由 ftimep 所指的 ftime 结构中。

ftime 的结构定义如下:

```
struct ftime{
 unsigned ft_tsec;5; /* two seconds */
 unsigned ft_min;6; /* minutes */
 unsigned ft_hour;5; /* hours */
 unsigned ft_day; 5; /* days */
 unsigned ft_month;4; /* month */
 unsigned ft_year;7; /* year-1980 */
};
```

返 回 值 若调用成功,则函数返回 0。

在出错时,函数返回-1,并将全局变量 errno 置为下列值之一:

|         |        |
|---------|--------|
| EINVFNC | 无效功能号  |
| EBADF   | 无效文件句柄 |

可移植性 只适用于 DOS 系统。

参 见 open, setftime

示 例 #include <stdio.h>

#include <io.h>

int main(void)

```
{
 FILE * stream;
 struct ftime ft;
 if ((stream = fopen("TEST. $$$",
 "wt")) == NULL)
 {
 fprintf(stderr, "Cannot open output file.\n");
 return 1;
 }
 getftime(fileno(stream), &ft);
 printf("File time: %u:%u:%u\n",
 ft.ft_hour, ft.ft_min,
 ft.ft_tsec * 2);
 printf("File date: %u/%u/%u\n",
 ft.ft_month, ft.ft_day,
```

```

 ft.ft_year+1980);
fclose(stream);
return 0;
}

```

## ■ getgraphmode 返回当前图形模式 ■

**用 法** #include <graphics.h>  
int far getgraphmode(void);

**原 型 在** graphics.h

**说 明** 在调用 getgraphmode 之前,程序必须已经成功地调用了 initgraph.  
在 graphics.h 中定义的枚举类型 graphics\_mode 给出了预定义图形模式的名字。可参阅有关 initgraph 函数的描述,其中描述的表格列出了这些枚举值。

**返 回 值** getgraphmode 函数返回由 initgraph 函数或 setgraphmode 函数设置的图形模式。

**可移植性** 只适用于 Turbo C,且只能在配备图形适配器的 IBM PC 及其兼容机上使用。

**参 见** getmoderange, restorecrtmode, setgraphmode

**示 例**

```

#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int midx, midy, mode;
 char numname[80], modename[80];
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 /* an error occurred */
 if (errorcode != grOk)
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt,");
 getch();
 /* terminate with an error code */
 exit(1);
 }
 midx = getmaxx() / 2;
 midy = getmaxy() / 2;
 /* get mode number and name strings */
 mode = getgraphmode();
}

```

```

 sprintf(numname, "%d is the current mode number.", mode);
 sprintf(modename, "%s is the current graphics mode", getmodename(mode));
 /* display the information */
 setttextjustify(CENTER_TEXT, CENTER_TEXT);
 outtextxy(midx, midy, numname);
 outtextxy(midx, midy + 2 * textheight("W"), modename);
 /* clean up */
 getch();
 closegraph();
 return 0;
}

```

## ■ getimage 将指定区域的位图象存入内存 ■

```

void far getimage(int left, int top, int right, int bottom,
void far * bitmap);

```

原型在 graphics.h

说明 getimage 将图象从屏幕拷贝到内存。

left、top、right、bottom 定义了屏幕上要拷贝的矩形区域 \* bitmap 指向内存中存放位图象的区域。该区域的前两个字节用于存放矩形的高和宽,其余部分存放图象本身。

返回值 无。

可移植性 只适于 Turbo C,且只能在配有图形适配器的 IBM PC 及其兼容机使用。

参见 imagesize, putimage, putpixel

示例

```

#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <alloc.h>

void save_screen(void far * buf[4]);
void restore_screen(void far * buf[4]);
int maxx, maxy;
int main(void)
{
 int gdriver=DETECT, gmode, errorcode;
 void far * ptr[4];
 /* auto-detect the graphics driver and mode */
 initgraph(&gdriver, &gmode, "");
 errorcode = graphresult(); /* check for any errors */
 if (errorcode != grOk)
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 }
}

```

```

 printf("Press any key to halt:");
 getch();
 exit(1);
}

maxx = getmaxx();
maxy = getmaxy();
/* draw an image on the screen */
rectangle(0, 0, maxx, maxy);
line(0, 0, maxx, maxy);
line(0, maxy, maxx, 0);
save_screen(ptr); /* save the current screen */
getch(); /* pause screen */
cleardevice(); /* clear screen */
restore_screen(ptr); /* restore the screen */
getch(); /* pause screen */
closegraph();
return 0;
}

void save_screen(void far *buf[4])
{
 unsigned size;
 int ystart=0, yend, yincr, block;
 yincr = (maxy+1) / 4;
 yend = yincr;
 size = imageize(0, ystart, maxx, yend);
 /* get byte size of image */
 for (block=0; block<=3; block++)
 {
 if ((buf[block] = farmalloc(size)) == NULL)
 {
 closegraph();
 printf("Error: not enough heap space in save_screen().\n");
 exit(1);
 }
 getimage(0, ystart, maxx, yend, buf[block]);
 ystart = yend + 1;
 yend += yincr + 1;
 }
}

void restore_screen(void far *buf[4])
{
 int ystart=0, yend, yincr, block;
 yincr = (maxy+1) / 4;

```

```

yend = yincr;
for (block=0; block<=3; block++)
{
 putimage(0, ystart, buf[block], COPY_PUT);
 farfree(buf[block]);
 ystart = yend + 1;
 yend += yincr + 1;
}
}

```

## ■ getlinesettings 读取当前线型、模式和宽度 ■

用 法 #include <graphics.h>

void far getlinesettings(struct linesettingstype far \*lineinfo);

原 型 在 graphics.h

说 明 本函数将当前线型、模式和宽度存到由 lineinfo 所指向的 linesettingstype 结构中。linesettingstype 结构在 graphics.h 中定义如下：

```

struct linesettingstype {
 int linestyle;
 unsigned upattern;
 int thickness;
};

```

linestyle 说明以何种线型来绘制线(如用实线、点划线、中心线和破折线)。

在 graphics.h 中定义的枚举类型 line\_styles 给出了以下线型名：

| 名字           | 值 | 说明      |
|--------------|---|---------|
| SOLID_LINE   | 0 | 实线      |
| DOTTED_LINE  | 1 | 点划线     |
| CENTER_LINE  | 2 | 中心线     |
| DASHED_LINE  | 3 | 破折线     |
| USERBIT_LINE | 4 | 用户定义的线型 |

下表的 thickness 指明以后所画的线的宽度是正常宽度线还是加粗宽度的。

upattern 是一个仅当 linestyle 为 USERBIT\_LINE(4)时方起作用的 16 位模式。在这种情况下,只要模式字里有一位是 1,则线中的对应象素就被用当前颜色画出。例如,实线对应 upattern 值为 0xFFFF(画出所有象素),破折线对应 upattern 值为 0x3333 或 0x0F0F。如果 setlinestyle 的 linestyle 参数不是 USERBIT\_LINE(4)则仍需给出 upattern 参数,但被忽略而不起作用。

| 名称          | 值 | 描述    |
|-------------|---|-------|
| NORM_WIDTH  | 1 | 一个象素宽 |
| THICK_WIDTH | 3 | 三个象素宽 |

返回值 无。

可移植性 只适用于 Turbo C, 且只能在配有图形适配器的 IBM PC 及其兼容机上使用。

参 见 setlinestyle

示 例

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

/* the names of the line styles supported */
char * lname[] = { "SOLID_LINE",
 "DOTTED_LINE",
 "CENTER_LINE",
 "DASHED_LINE",
 "USERBIT_LINE"
 };

int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 struct linesettingstype lineinfo;
 int midx, midy;
 char lstyle[80], lpattern[80], lwidth[80];
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
 }

 midx = getmaxx() / 2;
 midy = getmaxy() / 2;

 /* get information about current line settings */
 getlinesettings(&lineinfo);
 /* convert line information into strings */
 sprintf(lstyle, "%s is the line style.", lname[lineinfo.linestyle]);
 sprintf(lpattern, "0x%X is the user-defined line pattern.",
 lineinfo.upattern);
 sprintf(lwidth, "%d is the line thickness.",
```

```

 lineinfo.thickness);
 /* display the information */
 settextrjustfy(CENTER_TEXT, CENTER_TEXT);
 outtextxy(midx, midy, lstyle);
 outtextxy(midx, midy+2*textheight("W"), lpattern);
 outtextxy(midx, midy+4*textheight("W"), lwidth);
 /* clean up */
 getch();
 closegraph();
 return 0;
}

```

### ■ getmaxcolor 返回可选的最大有效颜色值 ■

**用 法** #include <graphics.h>  
int far getmaxcolor(void);

**原 型** 在 graphics.h

**说 明** 本函数返回当前图形驱动程序和图形模式下,可以传给函数 setcolor 的最大有效颜色值。例如,对于 256K 的 EGA, getmaxcolor 常返回值 15,这表明调用 setcolor 时的参数值在 0~15 是有效的。对于 CGA 高分辨率或 Hercules 单色适配器, getmaxcolor 返回值为 1。

**返 回 值** 返回最大的可用颜色值。

**可移植性** 只适用于 Turbo C,且只能在配有图形适配器的 IBM PC 及其兼容机上使用。

**参 见** getbkcolor, getcolor, getpalette, getpalettesize, setcolor

**示 例**

```

#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int midx, midy;
 char colstr[80];
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 }
}

```

```

 getch();
 exit(1); /* terminate with an error code */
}

midx = getmaxx() / 2;
midy = getmaxy() / 2;
/* grab the color info. and convert it to a string */
sprintf(colstr, "This mode supports colors 0..%d", getmaxcolor());
/* display the information */
settextjustify(CENTER_TEXT, CENTER_TEXT);
outtextxy(midx, midy, colstr);
/* clean up */
getch();
closegraph();
return 0;
}

```

## ■ getmaxmode 返回当前驱动程序的最大图形模式号 ■

**用 法** #include <graphics.h>  
int far getmaxmode(void);

**原 型 在** graphics.h

**说 明** getmaxmode 直接从驱动程序中找出当前加载的驱动程序的最大图形模式号。该函数比 getmoderange 函数的功能要强一些, 因为 getmoderange 函数只能对 Borland 的图形驱动程序进行操作。最小的图形模式号为 0。

**返 回 值** 返回当前驱动程序的最大图形模式号。

**可移植性** 只适用于 Turbo C, 且只能在配有图形适配器的 IBM PC 及其兼容机上使用。

**参 见** getmodename, getmoderange

**示 例** #include <graphics.h>

```
#include <stdlib.h>
```

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

```
#include <conio.h>
```

```
int main(void)
```

```
{
```

```
 /* request auto detection */
```

```
 int gdriver = DETECT, gmode, errorcode;
```

```
 int midx, midy;
```

```
 char modestr[80];
```

```
 /* initialize graphics and local variables */
```

```
 initgraph(&gdriver, &gmode, "");
```

```
 /* read result of initialization */
```

```
 errorcode = graphresult();
```

```
 if (errorcode != grOk) /* an error occurred */
```

```
{
```



```

printf("Graphics error: %s\n", grapherrormsg(errorcode));
printf("Press any key to halt:");
getch();
exit(1); /* terminate with an error code */
}
midx = getmaxx() / 2;
midy = getmaxy() / 2;
/* grab the mode info. and convert it to a string */
sprintf(modestr, "This driver supports modes 0.. %d", getmaxmode());
/* display the information */
settextjustify(CENTER_TEXT, CENTER_TEXT);
outtextxy(midx, midy, modestr);
/* clean up */
getch();
closegraph();
return 0;
}

```

## ■ getmaxx 返回屏幕上最大的 x 坐标值 ■

**用 法** #include <graphics.h>

int far getmaxx(void);

**原 型 在** graphics.h

**说 明** getmaxx 返回当前图形驱动程序和图形模式下最大的 x 坐标值(相对于屏幕)。例如,CGA320×200 模式下,函数 getmaxx 返回值 319。getmaxx 对于屏幕上区域的中心定位和边界确定等是非常有用的。

**返 回 值** 返回屏幕的最大 x 坐标。

**可移植性** 只适用于 Turbo C,且只能在配有图形适配器的 IBM PC 及其兼容机上使用。

**参 见** getmaxy, getx

**示 例** #include <graphics.h>

#include <stdlib.h>

#include <stdio.h>

#include <conio.h>

int main(void)

{

/\* request auto detection \*/

int gdriver = DETECT, gmode, errorcode;

int midx, midy;

char xrange[80], yrange[80];

/\* initialize graphics and local variables \*/

initgraph(&gdriver, &gmode, "");

/\* read result of initialization \*/

errorcode = graphresult();

```

if (errorcode != grOk) /* an error occurred */
{
 printf("Graphics error, %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
}

midx = getmaxx() / 2;
midy = getmaxy() / 2;
/* convert max resolution values into strings */
sprintf(xrange, "X values range from 0.. %d", getmaxx());
sprintf(yrange, "Y values range from 0.. %d", getmaxy());
/* display the information */
settextjustify(CENTER_TEXT, CENTER_TEXT);
outtextxy(midx, midy, xrange);
outtextxy(midx, midy+textheight("W"), yrange);
/* clean up */
getch();
closegraph();
return 0;
}

```

## ■ getmaxy 返回屏幕上最大的 y 坐标值 ■

**用 法** #include <graphics.h>

int far getmaxy(void);

**原 型** 在 graphics.h

**说 明** getmaxy 返回当前图形驱动器程序和图形模式下最大的 y 坐标值(相对于屏幕)。例如,对 CGA 320×200 模式下, getmaxy 返回值 199。getmaxy 对于屏幕上区域的中心定位边界确定等是非常有用的。

**返 回 值** 返回屏幕的最大 y 坐标。

**可移植性** 只适用于 Turbo C,且只能在配有图形适配器的 IBM PC 及其兼容机上使用。

**参 见** getmaxx, getx, gety

**示 例** #include <graphics.h>

#include <stdlib.h>

#include <stdio.h>

#include <conio.h>

int main(void)

{

/\* request auto detection \*/

int gdriver = DETECT, gmode, errorcode;

int midx, midy;

char xrange[80], yrange[80];

```

/* initialize graphics and local variables */
initgraph(&gdriver, &gmode, "");
/* read result of initialization */
errorcode = graphresult();
if (errorcode != grOk) /* an error occurred */
{
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
}

midx = getmaxx() / 2;
midy = getmaxy() / 2;
/* convert max resolution values into strings */
sprintf(xrange, "X values range from 0.. %d", getmaxx());
sprintf(yrange, "Y values range from 0.. %d", getmaxy());
/* display the information */
settextjustify(CENTER_TEXT, CENTER_TEXT);
outtextxy(midx, midy, xrange);
outtextxy(midx, midy + textheight("W"), yrange);
/* clean up */
getch();
closegraph();
return 0;
}

```

## ■ getmodename 返回指向含有指定图形模式名字符串的指针 ■

**用 法** #include <graphics.h>

char \* far getmodename(int mode\_number);

**原 型 在** graphics.h

**说 明** 本函数以一个图形为输入参数,返回包含相应图形的模式名的字符串,此名字嵌入在每个驱动程序中。返回值("320×200 CGA P1", "640×200 CGA"等)对创建菜单或显示图形适配器的状态是很有用的。

**返 回 值** 返回一个包含图形模式名的字符串的指针。

**可移植性** 只适用于 Turbo C,且只能在配有图形适配器的 IBM PC 及其兼容机上使用。

**参 见** getmaxmode, getmoderange

**示 例** #include <graphics.h>

#include <stdlib.h>

#include <stdio.h>

#include <conio.h>

int main(void)

{

```

/* request autodetection */
int gdriver = DETECT, gmode, errorcode;
int midx, midy, mode;
char numname[80], modename[80];
/* initialize graphics and local variables */
initgraph(&gdriver, &gmode, "");
/* read result of initialization */
errorcode = graphresult();
if (errorcode != grOk) /* an error occurred */
{
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt,");
 getch();
 exit(1); /* terminate with an error code */
}

midx = getmaxx() / 2;
midy = getmaxy() / 2;
/* get mode number and name strings */
mode = getgraphmode();
sprintf(numname, "%d is the current mode number.", mode);
sprintf(modename, "%s is the current graphics mode.",
 getmodename(mode));
/* display the information */
settextjustify(CENTER_TEXT, CENTER_TEXT);
outtextxy(midx, midy, numname);
outtextxy(midx, midy + 2 * textheight("W"), modename);
/* clean up */
getch();
closegraph();
return 0;
}

```

## ■ getmoderange 获取图形驱动程序的模式范围 ■

**用 法** #include <graphics.h>

```
void far getmoderange(int graphdriver, int far *lomode,
int far *himode);
```

**原 型** 在 graphics.h

**说 明** 本函数获取给定图形驱动程序 graphdriver 的有效图形模式范围, 允许的最小值返回在 \*lomode 中, 最高返回值在 \*himode 中。如果 graphdriver 给出一个无效的图形驱动程序, 则 \*lomode 和 \*himode 均被置为 -1; 如果 graphdriver 是 -1, 则缺省为当前的驱动程序。

**返 回 值** 无。

·可移植性 只适用于 Turbo C,且只能在配有图形适配器的 IBM PC 及其兼容机上使用。

参 见 getgraphmode, getmaxmode, getmodename, initgraph, setgraphmode

```
示 例 #include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int midx, midy;
 int low, high;
 char mrange[80];
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
 }

 midx = getmaxx() / 2;
 midy = getmaxy() / 2;
 /* get the mode range for this driver */
 getmoderange(gdriver, &low, &high);
 /* convert mode range info. into strings */
 sprintf(mrange, "This driver supports modes %d..%d", low, high);
 /* display the information */
 setttextjustify(CENTER_TEXT, CENTER_TEXT);
 outtextxy(midx, midy, mrange);
 /* clean up */
 getch();
 closegraph();
 return 0;
}
```

## ■ getpalette 返回当前调色板的有关信息 ■

用 法 #include <graphics.h>  
void far getpalette(struct palettetype far \*palette);

原型 在 `graphaics.h`

说明 `getpalette` 函数将有关当前调色板大小和颜色的有关信息填入到由 `paltte` 所指向的 `palettetype` 结构中。

`getpalette` 所用到的常数 `MAXCOLORS` 和结构 `palettetype` 在 `graphics.h` 中定义如下:

```
#define MAXCOLORS 15

struct palettetype {
 unsigned char size;
 signed char color [MAXCOLORS+1];
};
```

`size` 给出当前图形驱动程序和当前模式下调色板的颜色数目。

`colors` 是一个具有 `size` 个字节的数组,它含有对应于调色板中每一个项的实际的原始颜色编号。

注意:`getpalette` 不能被 IBM-8514 驱动程序使用。

返回值 无。

可移植性 只适用于 Turbo C,且只能在配有图形适配器的 IBM PC 及其兼容机上使用。

参见 `getbkcolor`, `getcolor`, `getdefaultpalette`, `getmaxcolor`, `setallpalette`, `setpalette`

示例 `#include <graphics.h>`

`#include <stdlib.h>`

`#include <stdio.h>`

`#include <conio.h>`

`int main(void)`

{

`/* request auto detection */`

`int gdriver = DETECT, gmode, errorcode;`

`struct palettetype pal;`

`char psize[80], pval[20];`

`int i, ht;`

`int y = 10;`

`/* initialize graphics and local variables */`

`initgraph(&gdriver, &gmode, "");`

`/* read result of initialization */`

`errorcode = graphresult();`

`/* an error occurred */`

`if (errorcode != grOk)`

{

`printf("Graphics error: %s\n", grapherrormsg(errorcode));`

`printf("Press any key to halt:");`

`getch();`

`/* terminate with an error code */`

```

 exit(1);
 }

 /* grab a copy of the palette */
 getpalette(&pal);

 /* convert palette info. into strings */
 sprintf(psize, "The palette has %d modifiable entries.", pal.size);

 /* display the information */
 outtextxy(0, y, psize);
 if (pal.size != 0)
 {
 ht = textheight("W");
 y += 2 * ht;
 outtextxy(0, y, "Here are the current values.");
 y += 2 * ht;
 for (i=0; i<pal.size; i++, y+=ht)
 {
 sprintf(pval, "palette[%02d]: 0x%02X", i, pal.colors[i]);
 outtextxy(0, y, pval);
 }
 }

 /* clean up */
 getch();
 closegraph();
 return 0;
}

```

## ■ getpalettesize 返回调色板的颜色数目 ■

用 法 #include <graphics.h>

int far getpalettesize(void);

原 型 在 graphics.h

说 明 getpalettesize 用来决定当前图形下可以设置多少调色板颜色项。例如, EGA 彩色显示器返回 16 项。

返 回 值 返回当前调色板的调色板项数。

可移植性 只适用于 Turbo C, 且只能在配有图形适配器的 IBM PC 及其兼容机上使用。

参 见 setpalette, setallpalette

示 例 #include <graphics.h>

#include <stdlib.h>

#include <stdio.h>

#include <conio.h>

int main()

{

/\* request auto detection \*/

```

int gdriver = DETECT, gmode, errorcode;
int midx, midy;
char psize[80];
/* initialize graphics and local variables */
initgraph(&gdriver, &gmode, "");
/* read result of initialization */
errorcode = graphresult();
if (errorcode != grOk) /* an error occurred */
{
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
}
midx = getmaxx() / 2;
midy = getmaxy() / 2;
/* convert palette size info. into string */
sprintf(psize, "The palette has %d modifiable entries.",
 getpalettesize());
/* display the information */
settextjustify(CENTER_TEXT, CENTER_TEXT);
outtextxy(midx, midy, psize);
/* clean up */
getch();
closegraph();
return 0;
}

```

## ■ getpass 读入口令

**用 法** #include <conio.h>

char \*getpass(const char \*prompt);

**原 型 在** conio.h

**说 明** 在显示以空字符终结的串 prompt 提示信息之后, getpass 从系统控制台读入一个口令, 并禁止回显。本函数返回一个指针, 指向以空字符终结的字符串, 该字符串的最大长度多为 8 个字节(不包含空终结符)。

**返 回 值** 返回值为指向字符串的静态指针, 该指针每次调用都被重写。

**可移植性** 仅适用于 UNIX 系统。

**参 见** getch

**示 例** #include <conio.h>

```

int main(void)
{
 char *password;

```



```

 password = getpass("Input a password:");
 cprintf("The password is: %s\r\n", password);
 return 0;
}

```

## ■ getpid 读取进程号

**用 法** #include <process.h>  
 unsigned getpid(void);

**原 型 在** process.h

**说 明** 进程标识号 ID 唯一标识一个运行程序。进程的概念是从 UNIX 等多任务操作系统中借用过来的。在这些操作系统中,进程是和唯一的进程号相关联的。

**返 回 值** getpid 返回程序段前缀(PSP)的段值。

**可移植性** 仅适用于 UNIX 系统。

**参 见** getpsp, \_psp(全局变量);

**示 例**

```

#include <stdio.h>
#include <process.h>
int main()
{
 printf("This program's process identification number (PID) "
 "number is %X\n", getpid());
 printf("Note: under DOS it is the PSP segment\n");
 return 0;
}

```

## ■ getpixel 读取得像素的颜色

**用 法** #include <graphics.h>  
 unsigned far getpixel(int x, int y);

**原 型 在** graphics.h

**说 明** getpixel 读取位于坐标(x,y)处像素的颜色。

**返 回 值** 返回给定像素的颜色值。

**可移植性** 只适用于 Turbo C,且只能在配有图形适配器的 IBM PC 及其兼容机上使用。

**参 见** getimage, putpixel

**示 例**

```

#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <dos.h>
#define PIXEL_COUNT 1000
#define DELAY_TIME 100 /* in milliseconds */
int main(void)
{

```

```

/* request auto detection */
int gdriver = DETECT, gmode, errorcode;
int i, x, y, color, maxx, maxy,
 maxcolor, seed;
/* initialize graphics and local variables */
initgraph(&gdriver, &gmode, "");
/* read result of initialization */
errorcode = graphresult();
/* an error occurred */
if (errorcode != grOk)
{
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
/* terminate with an error code */
 exit(1);
}
maxx = getmaxx() + 1;
maxy = getmaxy() + 1;
maxcolor = getmaxcolor() + 1;
while (! kbhit())
{
/* seed the random number generator */
 seed = random(32767);
 srand(seed);
 for (i=0; i<PIXEL_COUNT; i++)
 {
 x = random(maxx);
 y = random(maxy);
 color = random(maxcolor);
 putpixel(x, y, color);
 }
 delay(DELAY_TIME);
 srand(seed);
 for (i=0; i<PIXEL_COUNT; i++)
 {
 x = random(maxx);
 y = random(maxy);
 color = random(maxcolor);
 if (color == getpixel(x, y))
 putpixel(x, y, 0);
 }
}

```

```

/* clean up */
getch();
closegraph();
return 0;
}

```

## ■ getpsp 读取程序段前缀 ■

**用 法** #include <dos.h>  
unsigned getpsp(void);

**原 型 在** dos.h

**说 明** getpsp 函数执行 DOS 系统调用 0x62, 以获取程序段前缀(PSP)的地址。  
此调用只在 DOS 3.X 中存在。对于 DOS 2.X 和 1.X, 该函数可由启动代码设置的全局变量 \_psp 来代替。

**返 回 值** 返回 PSP 的段地址。

**可移植性** 只适用 DOS 3.X, 不适用于 DOS 的早期版本。

**参 见** getenv, \_psp(全局变量)

**示 例**

```

#include <stdio.h>
#include <dos.h>
int main(void)
{
 static char command[128];
 char far *cp;
 int len, i;
 printf("The program segment prefix is: %u\n", getpsp());
 /*
 _psp is preset to segment of the PSP
 Command line is located at offset 0x81
 from start of PSP
 */
 cp = MK_FP(_psp, 0x80);
 len = *cp;
 for (i = 0; i < len; i++)
 command[i] = cp[i+1];
 printf("Command line: %s\n", command);
 return 0;
}

```

## ■ gets 从标准输入流 stdin 中读取一字符串 ■

**用 法** #include <stdio.h>  
char \*gets(char \*s);

**原 型 在** stdio.h

**说 明** gets 从标准输入流 stdin 中读入一个以换行符为终结的字符串, 将其存入串 s 中, 并把 s 中的换行符用空字符(\0)代替。

gets 允许输入串中包含一些空白字符(空格符、制表符)。在遇到换行符时, gets 停止读取数据并返回, 并将换行符前的所有字符都拷贝到串 s 中。

**返回值** 若调用成功, 则函数返回串参数 s; 若出错或读到文件尾, 则函数返回 NULL。

**可移植性** 适用于 UNIX 系统, 在 ANSI C 中也有定义。

**参 见** cgets, ferror, fgets, fopen, fputs, fread, getc, puts, scanf

**示 例** #include <stdio.h>

```
int main(void)
{
 char string[80];
 printf("Input a string:");
 gets(string);
 printf("The string input was, %s\n", string);
 return 0;
}
```

## ■ gettext 拷贝文本屏幕上的文本拷贝到内存中 ■

**用 法** #include <conio.h>

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

**原 型** 在 conio.h

**说 明** gettext 将屏幕上由 left、top、right 和 bottom 所决定的矩形区域中的内容存入由 destin 所指向的内存中。

所有坐标都是绝对屏幕坐标, 而不是相对窗口的坐标, 左上角为坐标(1,1)。

gettext 将矩形中的内容按从左到右, 从上而下的顺序读入到内存。屏幕上的每个位置占两个字节内存, 第一个字节是单元中的字符, 第二字节是单元的视频显示属性。一个长为 h 行宽为 w 列的矩形所需的空间为:

字节数 = h 行 \* w 列 \* 2

**返回值** 若操作成功函数返回 1 如果失败(例如给出的坐标超过当前屏幕范围)返回 0。

**可移植性** 只适用于 IBM PC 及与 BIOS 兼容的系统。

**参 见** movetext, puttext

**示 例** #include <conio.h>

```
char buffer[4096];

int main(void)
{
 int i;
 clrscr();
 for (i = 0; i <= 20; i++)
```

```

 cprintf("Line # %d\r\n", i);
 gettext(1, 1, 80, 25, buffer);
 gotoxy(1, 25);
 cprintf("Press any key to clear screen...");
 getch();
 clrscr();
 gotoxy(1, 25);
 cprintf("Press any key to restore screen...");
 getch();
 puttext(1, 1, 80, 25, buffer);
 gotoxy(1, 25);
 cprintf("Press any key to quit...");
 getch();
 return 0;

```

## ■ gettextinfo 读取文本模式的显示信息 ■

用 法 #include <conio.h>

void gettextinfo(struct text\_info \*r);

原 型 在 conio.h

说 明 gettextinfo 将文本模式的显示信息填入由 r 所指向的 text\_info 结构中。

在 conio.h 中结构 text\_info 定义如下:

```

struct text_info{
 unsigned char winleft; /* left window coordinate */
 unsigned char wintop; /* top window coordinate */
 unsigned char winright; /* right window coordinate */
 unsigned char winbottom; /* bottom window coordinate */
 unsigned char attribute; /* text attribute */
 unsigned char normattr; /* normal attribute */
 unsigned char currmode; /* BW40,BW80,C40,C80,or C4350
 */
 unsigned char screenbeight; /* bootom-top
 */
 unsigned char screenwidth; /* right-left */
 unsigned char curx; /* x-coordinate in current window */
 unsigned char cury; /* y-coordinate in current window */
};

```

返 回 值 无返回值。其结果返回到 r 所指向的结构中。

可移植性 只适用于 IBM PC 及其兼容机。

参 见 textattr, textbackground, textcolor, textmode, wherex, wherey, window

示 例 #include <conio.h>

```
int main(void)
{
 struct text_info ti;
 gettextinfo(&ti);
 printf("window left %2d\r\n", ti.winleft);
 printf("window top %2d\r\n", ti.wintop);
 printf("window right %2d\r\n", ti.winright);
 printf("window bottom %2d\r\n", ti.winbottom);
 printf("attribute %2d\r\n", ti.attribute);
 printf("normal attribute %2d\r\n", ti.normattr);
 printf("current mode %2d\r\n", ti.currenmode);
 printf("screen height %2d\r\n", ti.screenheight);
 printf("screen width %2d\r\n", ti.screenwidth);
 printf("current x %2d\r\n", ti.curx);
 printf("current y %2d\r\n", ti.cury);
 return 0;
}
```

## ■ gettextsettings 返回当前图形字体的有关信息 ■

用 法 #include <graphics.h>

void far gettextsettings(struct textsettingstype far \* textypeinfo);

原 型 在 graphics.h

说 明 本函数将有关当前文本的字体、方向、大小和对齐方式的有关信息填入由 textinfo 所指的 textsettingstype 结构中。

gettextsettings 所用到的 textsettingstype 结构在 graphics.h 定义如下：

```
struct textsettingstype {
 int font;
 int direction;
 int charsize;
 int boriz;
 int vert;
};
```

每个字段的详细描述请参见 settextstyle 函数。

返 回 值 无。

可移植性 只适用于 Turbo C, 且只能在配有图形适配器的 IBM PC 及其兼容机上使用。

参 见 outtext, outtextxy, registerbgifont, settextjustify, settextstyle, setusercharsize, textheight, textwidth

示 例 #include <graphics.h>  
#include <stdlib.h>

```

#include <stdio.h>
#include <conio.h>
/* the names of the fonts supported */
char *font[] = { "DEFAULT_FONT",
 "TRIPLEX_FONT",
 "SMALL_FONT",
 "SANS_SERIF_FONT",
 "GOTHIC_FONT"
 };

/* the names of the text directions supported */
char *dir[] = { "HORIZ_DIR", "VERT_DIR" };
/* horizontal text justifications supported */
char *hjust[] = { "LEFT_TEXT", "CENTER_TEXT", "RIGHT_TEXT" };
/* vertical text justifications supported */
char *vjust[] = { "BOTTOM_TEXT", "CENTER_TEXT", "TOP_TEXT" };

int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 struct textsettingstype textinfo;
 int midx, midy, ht;
 char fontstr[80], dirstr[80], sizestr[80];
 char hjuststr[80], vjuststr[80];
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
 }

 midx = getmaxx() / 2;
 midy = getmaxy() / 2;
 /* get information about current text settings */
 gettextsettings(&textinfo);
 /* convert text information into strings */
 sprintf(fontstr, "%s is the text style.", font[textinfo.font]);
 sprintf(dirstr, "%s is the text direction.", dir[textinfo.direction]);
 sprintf(sizestr, "%d is the text size.", textinfo.charsize);

```

```

printf(hjuststr, "%s is the horizontal justification.",
 hjust[textinfo.horiz]);
printf(vjuststr, "%s is the vertical justification.",
 vjust[textinfo.vert]);
/* display the information */
ht = textheight("W");
settextjustify(CENTER_TEXT, CENTER_TEXT);
outtextxy(midx, midy, fontstr);
outtextxy(midx, midy+2*ht, dirstr);
outtextxy(midx, midy+4*ht, sizestr);
outtextxy(midx, midy+6*ht, hjuststr);
outtextxy(midx, midy+8*ht, vjuststr);
/* clean up */
getch();
closegraph();
return 0;
}

```

## ■ **gettime** 读取系统时间

**用 法** #include <dos.h>

void gettime(struct time \*timep);

**原 型** 在 dos.h

**说 明** gettime 将系统的当前时间填入由 timep 指向的 time 结构中。

time 结构定义如下：

```

struct time{
 unsigned char ti_min; /* minutes */
 unsigned char ti_hour; /* hours */
 unsigned char ti_hund; /* hundredths of seconds */
 unsigned char ti_sec; /* seconds */
};

```

**返 回 值** 无。

**可移植性** 只适用于 DOS 系统。

**参 见** getdate, setdate, settime, stime, time

**示 例** #include <stdio.h>

#include <dos.h>

int main(void)

{

struct time t;

gettime(&t);

printf("The current time is: %2d,%02d,%02d.%02d\n",

t.ti\_hour, t.ti\_min, t.ti\_sec, t.ti\_hund);



```
return 0;
```

```
}
```

## ■ getvect 读取中断向量

用 法 #include <dos.h>

```
void interrupt (* getvect(int interruptno));
```

原 型 在 dos.h

说 明 8086 系列的处理器包含一组中断向量, 向量号范围从 0 到 255。每个向量长度为 4 字节, 其值实际上是中断处理函数的入口地址。

getvect 函数将读入由 interruptno 所指定的中断向量值, 并把该中断向量值作为中断函数的(远)指针返回。interruptno 值的范围在 0 到 255 之间。

返 回 值 getvect 返回中断号为 interruptno 向量的当前 4 字节值。

可移植性 只适用于 DOS 系统。

参 见 disable, enable, geninterrupt, setvect

示 例 #include <stdio.h>

```
#include <dos.h>
```

```
void interrupt get_out(); /* interrupt prototype */
```

```
void interrupt (* oldfunc)(); /* interrupt function pointer */
```

```
int looping = 1;
```

```
int main(void)
```

```
{
```

```
 puts("Press <Shift><Prt Sc> to terminate");
```

```
 /* save the old interrupt */
```

```
 oldfunc = getvect(5);
```

```
 /* install interrupt handler */
```

```
 setvect(5, get_out);
```

```
 /* do nothing */
```

```
 while (looping);
```

```
 /* restore to original interrupt routine */
```

```
 setvect(5, oldfunc);
```

```
 puts("Success");
```

```
 return 0;
```

```
}
```

```
void interrupt get_out()
```

```
{
```

```
 looping = 0; /* change global variable to get out of loop */
```

```
}
```

## ■ getverify 取得 DOS 的当前校验状态

用 法 #include <dos.h>

```
int getverify(void);
```

原型在 dos.h

说明 getverify 取得当前校验状态。校验标志控制数据写入磁盘时是否进行 CRC 校验。当状态为 off 时,表示不对写入的数据进行校验;当状态为 on 时,表示对所有写入磁盘的数据进行校验,以确保数据能正确写入磁盘。

返回值 返回校验位的当前状态 0 或 1。

● 返回 0 表示校验标志为 off

● 返回 1 表示校验标志为 on

可移植性 只适用于 DOS。

参见 setverify

示例

```
#include <stdio.h>
#include <dos.h>
int main(void)
{
 if (getverify())
 printf("DOS verify flag is on\n");
 else
 printf("DOS verify flag is off\n");
 return 0;
}
```

## ■ getviewsettings 返回有关当前视区的信息 ■

用法 #include <graphics.h>  
void far getviewsettings(struct viewporttype far \* viewport);

原型在 graphics.h

说明 本函数将当前视区的有关信息填入由 viewport 所指向的 viewporttype 结构中。在 graphics.h 中如下定义 getviewsettings 函数使用的 viewporttype 结构:

```
struct viewporttype {
 int left, top, right, bottom;
 int clip;
};
```

返回值 无。

可移植性 只适用于 Turbo C,且只能在配有图形适配器的 IBM PC 及其兼容机上使用。

参见 clearviewport, getx, gety, setviewport

示例

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
char * clip[] = { "OFF", "ON" };
int main(void)
{
```

```

/* request auto detection */
int gdriver = DETECT, gmode, errorcode;
struct viewporttype viewinfo;
int midx, midy, ht;
char topstr[80], botstr[80], clipstr[80];
/* initialize graphics and local variables */
initgraph(&gdriver, &gmode, "");
/* read result of initialization */
errorcode = graphresult();
if (errorcode != grOk) /* an error occurred */
{
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
}

midx = getmaxx() / 2;
midy = getmaxy() / 2;
/* get information about current viewport */
getviewsettings(&viewinfo);
/* convert text information into strings */
sprintf(topstr, "(%d, %d) is the upper left viewport corner.",
 viewinfo.left, viewinfo.top);
sprintf(botstr, "(%d, %d) is the lower right viewport corner.",
 viewinfo.right, viewinfo.bottom);
sprintf(clipstr, "Clipping is turned %s.", clip[viewinfo.clip]);
/* display the information */
settextjustify(CENTER_TEXT, CENTER_TEXT);
ht = textheight("W");
outtextxy(midx, midy, topstr);
outtextxy(midx, midy + 2 * ht, botstr);
outtextxy(midx, midy + 4 * ht, clipstr);
/* clean up */
getch();
closegraph();
return 0;
}

```

## ■ getw 从输入流中读取一整数

用 法 #include <stdio.h>

int getw(FILE \* stream);

原 型 在 stdio.h

**说 明** getw 返回指定输入流 stream 中的下一个整数,它假定在文件中无特殊的对齐字符。

当输入流以文本模式打开时,不能使用 getw。

**返 回 值** getw 函数返回输入流中的下一个整数,若函数遇到文件结束符或出错时,则返回 EOF。

因为 EOF 是 getw 函数的合法返回值,故应使用 feof 或 ferror 函数来检测是文件结束还是出错。

**可移植性** 可适用于 UNIX 系统。

**参 见** putw

**示 例**

```
#include <stdio.h>
#include <stdlib.h>
#define FNAME "test. $$$"
int main(void)
{
 FILE *fp;
 int word;
 /* place the word in a file */
 fp = fopen(FNAME, "wb");
 if (fp == NULL)
 {
 printf("Error opening file %s\n", FNAME);
 exit(1);
 }
 word = 94;
 putw(word,fp);
 if (ferror(fp))
 printf("Error writing to file\n");
 else
 printf("Successful write\n");
 fclose(fp);
 /* reopen the file */
 fp = fopen(FNAME, "rb");
 if (fp == NULL)
 {
 printf("Error opening file %s\n", FNAME);
 exit(1);
 }
 /* extract the word */
 word = getw(fp);
 if (ferror(fp))
 printf("Error reading file\n");
 else
```

```

 printf("Successful read; word = %d\n", word);
 /* clean up */
 fclose(fp);
 unlink(FNAME);
 return 0;
}

```

## ■ getx 返回当前图形方式下位置的 x 坐标值 ■

用 法 #include <graphics.h>

```
int far getx(void);
```

原 型 在 graphics.h

说 明 getx 返回当前图形位置的 x 坐标值,此坐标值是相对于视区而言的。

返 回 值 返回当前位置的 x 坐标值。

可移植性 只适用于 Turbo C,且只能在配有图形适配器的 IBM PC 及其兼容机上使用。

参 见 getmaxx, getmaxy, getviewsettings, gety, moveto

示 例 #include <graphics.h>

```
#include <stdlib.h>
```

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

```
#include <conio.h>
```

```
int main(void)
```

```
{
```

```
 /* request auto detection */
```

```
 int gdriver = DETECT, gmode, errorcode;
```

```
 char msg[80];
```

```
 /* initialize graphics and local variables */
```

```
 initgraph(&gdriver, &gmode, "");
```

```
 /* read result of initialization */
```

```
 errorcode = graphresult();
```

```
 if (errorcode != grOk) /* an error occurred */
```

```
 {
```

```
 printf("Graphics error; %s\n", grapherrormsg(errorcode));
```

```
 printf("Press any key to halt;");
```

```
 getch();
```

```
 exit(1); /* terminate with an error code */
```

```
 }
```

```
 /* move to the screen center point */
```

```
 moveto(getmaxx() / 2, getmaxy() / 2);
```

```
 /* create a message string */
```

```
 sprintf(msg, "<-(%d, %d) is the here.", getx(), gety());
```

```
 /* display the message */
```

```
 outtext(msg);
```

```
 /* clean up */
```

```

 getch();
 closegraph();
 return 0;
}

```

## ■ gety 返回当前位置的 y 坐标值 ■

**用 法** #include <graphics.h>

int far gety(void);

**原 型** 在 graphics.h

**说 明** gety 返回当前位置的 y 坐标值,该值是相对窗口而言的。

**返 回 值** 返回当前位置的 y 坐标值。

**可移植性** 只适用于 Turbo C,且只能在配有图形适配器的 IBM PC 及其兼容机上使用。

**参 见** getmaxx, getmaxy, getviewsettings, getx, moveto

**示 例**

```

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

int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 char msg[80];

 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt,");
 getch();
 exit(1); /* terminate with an error code */
 }

 /* move to the screen center point */
 moveto(getmaxx() / 2, getmaxy() / 2);
 /* create a message string */
 sprintf(msg, "<-(%d, %d) is the here.", getx(), gety());
 /* display the message */
 outtext(msg);
 /* clean up */
 getch();
 closegraph();
}

```

```

 return 0;
}

```

## ■ gmtime 把日期和时间转换为格林威治标准时间(GMT) ■

**用 法** #include <time.h>

```
struct tm * gmtime(const time_t * timer);
```

**原 型 在** time.h

**说 明** gmtime 接收由 time 返回值的地址,并返回包含修正时间的结构 tm 指针。gmtime 直接把时间转换为格林威治的时间(GMT)。

全局长整型变量 timezone 包含 GMT 和地方标准时间的差值,单位为秒(在 PST 中,变量 timezone 为 8x60x60)。全局变量 daylight 为非零值,当且仅当采用美国标准夏令时进行转换。

包含文件 time.h 中结构 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;
};

```

这些量给出了按 24 小时计算的时间、每月中的天数(1~31)、月份(0~11)、周日(星期天为 0)、年号减去 1900 得到的数、一年中的天数(1~365)和一个表示使用夏令时的非 0 标志值。

**返 回 值** gmtime 返回包含修正时间的结构指针,该结构是静态的,每次调用时都被重写。

**可移植性** 适用于 UNIX 系统,在 ANSI C 中有定义。

**参 见** asctime, ctime, ftime, localtime, stime, time, tzset

**示 例**

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <dos.h>

/* Pacific Standard Time & Daylight Savings */
char * tzstr = "TZ=PST&PDT";

int main(void)
{
 time_t t;
 struct tm *gmt, *area;

```

```

 putenv(tzstr);
 tzset();
 t = time(NULL);
 area = localtime(&t);
 printf("Local time is: %s", asctime(area));
 gmt = gmtime(&t);
 printf("GMT is: %s", asctime(gmt));
 return 0;
}

```

## ■ gotoxy 在文本窗口中定位文本光标

**用 法** #include <conio.h>

void gotoxy(int x, int y);

**原 型** 在 conio.h

**说 明** gotoxy 函数在将当前文本窗口中移动光标到指定的位置。如果给出的坐标无效,这时忽略 gotoxy 的调用,例如当窗口右下角坐标是(35,25)时,如果函数调用写为 gotoxy (40,30),则该调用不起作用。

**返 回 值** 无。

**可移植性** 只能在 IBM PC 及其兼容机上使用,Turbo Pascal 有相应的函数。

**参 见** wherex,wherey>window

**示 例** #include <conio.h>

```

int main(void)
{
 clrscr();
 gotoxy(35, 12);
 cprintf("Hello world");
 getch();
 return 0;
}

```

## ■ graphdefaults 复位图形设置

**用 法** #include <graphics.h>

void far graphdefaults(void);

**原 型** 在 graphics.h

**说 明** graphdefaults 将所有图形设置复位为它们的缺省值,即:

- 将视区置为整个屏幕。
- 从当前位置移到点(0,0)。
- 设置缺省的调色板颜色、背景颜色和画线颜色。
- 设置缺省的填充类型的模式。
- 设置缺省的文本字体和对齐方式。



返回值 无。

可移植性 只适用于 Turbo C,且只能在配有图形适配器的 IBM PC 及兼容机上使用。

参 见 `initgraph, setgraphmode`

示 例

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
 }

 /* draw a line */
 line(0, 0, getmaxx(), getmaxy());
 /* clean up */
 getch();
 closegraph();
 return 0;
}
```

## ■ grapherrormsg 返回一个指向错误信息串的指针 ■

用 法 `#include <graphics.h>`  
`char * far grapherrormsg(int errorcode);`

原 型 在 `graphics.h`

说 明 `grapherrormsg` 返回同错误代码 `errorcode` 相对应的指向错误信息字符串的指针,变量 `errorcode` 是由函数 `graphresult` 返回的错误代码。

请参阅第四章“全局变量”中的 `errno` 项的说明,其中列出了错误信息和助记符。

返回值 `grapherrormsg` 返回一个指向错误信息串的指针。

可移植性 只适用于 Turbo C,且只能在配有图形适配器的 IBM PC 及其兼容机上使用。

参 见 `graphresult`

示 例 `#include <graphics.h>`

```

#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#define NONSENSE -50
int main(void)
{
 /* FORCE AN ERROR TO OCCUR */
 int gdriver = NONSENSE, gmode, errorcode;
 /* initialize graphics mode */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 /* if an error occurred, then output a
 /* descriptive error message.
 if (errorcode != grOk)
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
 }
 /* draw a line */
 line(0, 0, getmaxx(), getmaxy());
 /* clean up */
 getch();
 closegraph();
 return 0;
}

```

## ■ \_graphfreemem 可修改的图形内存释放函数 ■

用 法 #include <graphics.h>

void far \_graphfreemem(void far \*ptr, unsigned size);

原 型 在 graphics.h

说 明 \_graphfreemem 函数释放由用户调用 \_graphgetmem 函数所分配的内存。通过定义自己的 \_graphfreemem(必须准确地按所示方法定义),用户可以控制图形系统内存管理。该函数的缺省用法只是调用 free。

返 回 值 无。

可移植性 只适用于 Turbo C,且只能在配有图形适配器的 IBM PC 及其兼容机上使用。

参 见 \_graphgetmem, setgraphbufsize

示 例 #include <graphics.h>

#include <stdlib.h>

#include <stdio.h>

```

#include <conio.h>
#include <alloc.h>
int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int midx, midy;
 /* clear the text screen */
 clrscr();
 printf("Press any key to initialize graphics mode:");
 getch();
 clrscr();
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
 }
 midx = getmaxx() / 2;
 midy = getmaxy() / 2;
 /* display a message */
 settextjustify(CENTER_TEXT, CENTER_TEXT);
 outtextxy(midx, midy, "Press any key to exit graphics mode:");
 /* clean up */
 getch();
 closegraph();
 return 0;
}

/* called by the graphics kernel to allocate memory */
void far * far _graphgetmem(unsigned size)
{
 printf("_graphgetmem called to allocate %d bytes.\n", size);
 printf("press any key:");
 getch();
 printf("\n");
 /* allocate memory from far heap */
 return farmalloc(size);
}

```

```

/* called by the graphics kernel to free memory */
void far _graphfreemem(void far * ptr, unsigned size)
{
 printf("_graphfreemem called to free %d bytes.\n", size);
 printf("press any key:");
 getch();
 printf("\n");
 /* free ptr from far heap */
 farfree(ptr);
}

```

### graphgetmem 可修改的图形内存分配函数

用 法 #include <graphics.h>

void far \* far \_graphgetmem(unsigned size);

原 型 在 graphics.h

说 明 图形库函数(而非用户程序)通常调用 \_graphgetmem 为内部缓冲区、图形驱动程序和字符集分配内存。用户通过定义自己的 \_graphgetmem(必须准确地按在范例中所示方法定义),用户可以控制图形系统的内存管理。该函数的缺省用法只是调用 malloc。

返 回 值 无。

可移植性 只适用于 Turbo C,且只能在装有图形适配器的 IBM PC 及其兼容机上使用。

参 见 \_graphfreemem, initgraph, setgraphbufsize

示 例 #include <graphics.h>

#include <stdlib.h>

#include <stdio.h>

#include <conio.h>

#include <alloc.h>

int main(void)

{

/\* request autodetection \*/

int gdriver = DETECT, gmode, errorcode;

int midx, midy;

/\* clear the text screen \*/

clrscr();

printf("Press any key to initialize graphics mode:");

getch();

clrscr();

/\* initialize graphics and local variables \*/

initgraph(&gdriver, &gmode, "");

/\* read result of initialization \*/

errorcode = graphresult();

```

if (errorcode != grOk) /* an error occurred */
{
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
}

midx = getmaxx() / 2;
midy = getmaxy() / 2;
/* display a message */
settextjustify(CENTER_TEXT, CENTER_TEXT);
outtextxy(midx, midy, "Press any key to exit graphics mode:");
/* clean up */
getch();
closegraph();
return 0;
}

/* called by the graphics kernel to allocate memory */
void far * far_graphgetmem(unsigned size)
{
 printf("_graphgetmem called to allocate %d bytes.\n", size);
 printf("press any key:");
 getch();
 printf("\n");
 /* allocate memory from far heap */
 return farmalloc(size);
}

/* called by the graphics kernel to free memory */
void far_graphfreemem(void far * ptr, unsigned size)
{
 printf("_graphfreemem called to free %d bytes.\n", size);
 printf("press any key:");
 getch();
 printf("\n");
 /* free ptr from far heap */
 farfree(ptr);
}

```

### ■ graphresult 返回最后一次失败图形操作的错误码 ■

用 法 #include <graphics.h>

int far graphresult(void);

原 型 在 graphics.h

**说明** `graphresult` 返回上次图形操作的错误码(此错误码报告一个错误),并将错误级复位为 `grOk`.

下面的表格列出了由 `graphresult` 返回的错误代码,枚举类型 `graph_errors` 定义了错误码,它在 `graphics.h` 定义如表。

| 错误码 | <code>graph_errors</code>       | 相应的错误信息串                              |
|-----|---------------------------------|---------------------------------------|
| 0   | <code>grOk</code>               | 无错误                                   |
| -1  | <code>grNoInitGraph</code>      | (BGI)图形未安装(用 <code>initgraph</code> ) |
| -2  | <code>grNotDetected</code>      | 未检测到图形硬件                              |
| -3  | <code>grFileNotFound</code>     | 未找到设备驱动程序文件                           |
| -4  | <code>grInvalidDrive</code>     | 设备驱动程序文件是无效的                          |
| -5  | <code>grNoLoadMem</code>        | 无足够的内存装入驱动程序                          |
| -6  | <code>grNoScanMem</code>        | 扫描填充超出内存                              |
| -7  | <code>grNoFloodMem</code>       | 整屏填充超出内存                              |
| -8  | <code>grFontNotFound</code>     | 未找到字体文件                               |
| -9  | <code>grNoFontMem</code>        | 无足够内存装入字体                             |
| -10 | <code>grInvalidMode</code>      | 图形模式(针对所选的驱动程序)无效                     |
| -11 | <code>grError</code>            | 图形错误                                  |
| -12 | <code>grIOerror</code>          | 图形 I/O 错误                             |
| -13 | <code>grInvalidFont</code>      | 无效字体文件                                |
| -14 | <code>grInvalidFontNum</code>   | 无效字体号                                 |
| -15 | <code>grInvalidDeviceNum</code> | 无效设备号                                 |
| -18 | <code>grInvalidVersion</code>   | 无效版本号                                 |

注意:`graphresult` 被调用后,它的内部变量复位为 0。因此,应该把 `graphresult` 的返回值存入临时变量中,然后再测试该变量。

**返回值** `graphresult` 返回当前图形错误码,即一个范围从 -15 到 0 的整数;`grapherrormsg` 返回一个指向与 `graphresult` 返回值相对应的错误信息的串指针。

**可移植性** 只适用于 Turbo C,它只能在配有图形适配器的 IBM PC 及其兼容机上使用。

**参见** `detectgraph`, `drawpoly`, `fillpoly`, `floodfill`, `grapherrormsg`, `initgraph`, `pieslice`, `registerbgidriver`, `registerbgifont`, `setallpalette`, `sectcolor`, `setfillstyle`, `setgraphmode`, `setlinestyle`, `setpalette`, `settextjustify`, `settextstyle`, `setusercharsize`, `setviewport`, `setvisualpage`

**示例**

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
```

```
int main(void)
```

```

{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;

 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
 }

 /* draw a line */
 line(0, 0, getmaxx(), getmaxy());
 /* clean up */

 getch();
 closegraph();
 return 0;
}

```

## harderr 建立一个错误处理程序

用 法 #include <dos.h>

void harderr(int (\* handler)());

原 型 在 dos.h

说 明 harderr 为当前程序建立一个新的硬件错误处理程序,当发生 0x24 中断时,则调用该处理程序(参阅 DOS 技术参考手册中有关中断的描述)。

当 0x24 中断发生时,程序调用 handler 指向的函数。该处理函数接受下列参数:

handler(int errval, int ax, int bp, int si);

errval 是 DOS 置于 DI 寄存器中的出错代码。ax, bp 和 si 分别是 DOS 置于 AX、BP 和 SI 寄存器中的值。

ax 表示是否有磁盘或其它设备错误。如果 ax 为正值,表示是磁盘错误,否则为设备错误。对于磁盘错误,ax 的值"与"上 0x00FF,可得到出错驱动器号(0=A, 1=B 等等)。

bp 和 si 一起指向出错驱动程序的程序头, bp 为段地址, si 是偏移量。

handler 指向的函数不被直接被调用,由 harderr 建立一个中断处理程序来调用它。

此处理程序可以调用 DOS 1~0xc 中断;其它的 DOS 调用会破坏 DOS。

注意:不能使用 C 标准的 I/O 调用和 UNIX 仿真 I/O 调用。  
 处理程序返回 0(对应于 ignore),1(对应于 retry)和 2(对应于 abort)。

返回值 无。

可移植性 仅适用于 DOS。

参见 hardresume, hardretn, peek, poke

示例 /\*

```
This program will trap disk errors and
prompt the user for action. Try running it
with no disk in drive A, to invoke its
functions.
```

```
*/
#include <stdio.h>
#include <conio.h>
#include <dos.h>
#define IGNORE 0
#define RETRY 1
#define ABORT 2
int buf[500];
/*
define the error messages for trapping disk problems
```

```
*/
static char *err_msg[] = {
 "write protect",
 "unknown unit",
 "drive not ready",
 "unknown command",
 "data error (CRC)",
 "bad request",
 "seek error",
 "unknown media type",
 "sector not found",
 "printer out of paper",
 "write fault",
 "read fault",
 "general failure",
 "reserved",
 "reserved",
 "invalid disk change"
};
```

```
error_win(char *msg)
{
```

```
 int retval;
```



```

 cputs(msg);
/*
prompt for user to press a key to abort, retry, ignore
*/
 while(1)
 {
 retval= getch();
 if (retval == 'a' || retval == 'A')
 {
 retval = ABORT;
 break;
 }
 if (retval == 'r' || retval == 'R')
 {
 retval = RETRY;
 break;
 }
 if (retval == 'i' || retval == 'I')
 {
 retval = IGNORE;
 break;
 }
 }
 return(retval);
}
/*
#pragma warn -par reduces warnings which occur
due to the non use of the parameters errval,
bp and si to the handler.
*/
#pragma warn -par
int handler(int errval,int ax,int bp,int si)
{
 static char msg[80];
 unsigned di;
 int drive;
 int errorno;
 di= _DI;
/*
if this is not a disk error then it was
another device having trouble
*/
 if (ax < 0)

```

```

 {
 /* report the error */
 error_win("Device error");
 /* and return to the program directly requesting abort */
 hardretn(ABORT);
 }
/* otherwise it was a disk error */
drive = ax & 0x00FF;
errno = di & 0x00FF;
/* report which error it was */
sprintf(msg, "Error: %s on drive %c\\r\\nA)abort, R)etry, I)gnore: ",
 err_msg[errno], 'A' + drive);
/*
return to the program via dos interrupt 0x23 with abort, retry,
or ignore as input by the user.
*/
 hardresume(error_win(msg));
 return ABORT;
}
#pragma warn +par
int main(void)
{
 /*
install our handler on the hardware problem interrupt
*/
 harderr(handler);
 clrscr();
 printf("Make sure there is no disk in drive A:\\n");
 printf("Press any key\\n");
 getch();
 printf("Trying to access drive A:\\n");
 printf("fopen returned %p\\n", fopen("A:temp.dat", "w"));
 return 0;
}

```

## ■ hardresume 硬件错误处理函数 ■

用 法 #include <dos.h>

void hardresume(int, axret);

原 型 在 dos.h

说 明 由 harderr 建立的错误处理程序, 可以通过调用 hardresume 返回到 DOS。hardresume 的返回值包含 abort(2)、retry(1) 或 ignore(0) 指示标志。abort 是通过 DOS 中断 0x23(control-break 中断)实现的。

该处理程序返回 ignore 时标志为 0, 返回 retry 时标志为 1, 返回 abort 时标志为 2.

返回值 无。

可移植性 仅适用于 DOS.

参 见 harderr, hardretn

示 例

```

/* This program will trap disk errors and prompt the user for action. */
/* Try running it with no disk in drive A, to invoke its functions */
#include <stdio.h>
#include <conio.h>
#include <dos.h>
#define IGNORE 0
#define RETRY 1
#define ABORT 2
int buf[500];
/* define the error messages for trapping disk problems */
static char *err_msg[] = {
 "write protect",
 "unknown unit",
 "drive not ready",
 "unknown command",
 "data error (CRC)",
 "bad request",
 "seek error",
 "unknown media type",
 "sector not found",
 "printer out of paper",
 "write fault",
 "read fault",
 "general failure",
 "reserved",
 "reserved",
 "invalid disk change"
};
error_win(char *msg)
{
 int retval;
 cputs(msg);
 /* prompt for user to press a key to abort, retry, ignore */
 while(1)
 {
 retval = getch();
 if (retval == 'a' || retval == 'A')

```

```

 {
 retval = ABORT;
 break;
 }
 if (retval == 'r' || retval == 'R')
 {
 retval = RETRY;
 break;
 }
 if (retval == 'i' || retval == 'I')
 {
 retval = IGNORE;
 break;
 }
}
return(retval);
}

/* pragma warn -par reduces warnings which occur due to the non use */
/* of the parameters errval, bp and si to the handler. */
#pragma warn -par
int handler(int errval,int ax,int bp,int si)
{
 static char msg[80];
 unsigned di;
 int drive;
 int errorno;
 di = _DI;
 /* if this is not a disk error then it was another device having trouble */
 if (ax < 0)
 {
 /* report the error */
 error_win("Device error");
 /* and return to the program directly
 requesting abort */
 hardretn(ABORT);
 }
 /* otherwise it was a disk error */
 drive = ax & 0x00FF;
 errorno = di & 0x00FF;
 /* report which error it was */
 sprintf(msg, "Error, %s on drive %c\r\nA)bort, R)etry, I)gnore, ",
 err_msg[errorno], 'A' + drive);
 /* return to the program via dos interrupt 0x23 with abort, retry */

```

```

/* or ignore as input by the user. */
hardresume(error_win(msg));
return ABORT;
}
#pragma warn +par
int main(void)
{
/* install our handler on the hardware problem interrupt */
harderr(handler);
clrscr();
printf("Make sure there is no disk in drive A:\n");
printf("Press any key\n");
getch();
printf("Trying to access drive A:\n");
printf("fopen returned %p\n",fopen("A:temp.dat", "w"));
return 0;
}

```

## ■ hardretn 硬件错误处理函数 ■

用 法 #include <dos.h>

void hardteln (int retn);

原 型 在 dos.h

说 明 由 harderr 建立的错误处理程序,可以通过调用 hardretn 直接返回到应用程序。该处理程序返回 ignore 时标志为 0,返回 retry 时标志为 1,返回 abort 时标志为 2。

返 回 值 无。

可移植性 仅适用于 DOS。

参 见 harderr,hardresume

示 例

```

/* This program will trap disk errors and return to the program. */
/* Try running it with no disk in drive A; to invoke its functions */
#include <stdio.h>
#include <conio.h>
#include <dos.h>
#define IGNORE 0
#define RETRY 1
#define ABORT 2
int buf[500];
/* define the error messages for trapping disk problems */
static char *err_msg[] = {
 "write protect",
 "unknown unit",
 "drive not ready",

```

```

 "unknown command",
 "data error (CRC)",
 "bad request",
 "seek error",
 "unknown media type",
 "sector not found",
 "printer out of paper",
 "write fault",
 "read fault",
 "general failure",
 "reserved",
 "reserved",
 "invalid disk change"
};

void error_win(char *msg)
{
 cputs(msg);
}

/* pragma warn —par reduces warnings which occur due to the */
/* non use of the parameters errval, bp and si to the handler */
#pragma warn —par
int handler(int errval,int ax,int bp,int si)
{
 static char msg[80];
 unsigned di;
 int drive;
 int errno;
 di = _DI;
 /* if this is not a disk error then it was another device having trouble */
 if (ax < 0)
 {
 /* report the error */
 error_win("Device error");
 /* and return to the program directly
 requesting abort */
 hardretn(ABORT);
 }
 /* otherwise it was a disk error */
 drive = ax & 0x00FF;
 errno = di & 0x00FF;
 /* report which error it was */
 sprintf(msg,"Error: %s on drive %c\r\n",err_msg[errno],'A'+drive);
 error_win(msg);
}

```

```

/* return to the program via dos interrupt 0x23 with abort, retry or */
/* ignore as input by the user. */
hardretn(ABORT);
return ABORT;
}
#pragma warn +par
int main(void)
{
 FILE * tempfile;
 /* install our handler on the hardware problem interrupt */
 harderr(handler);
 clrscr();
 printf("Make sure there is no disk in drive A:\n");
 printf("Press any key\n");
 getch();
 printf("Trying to access drive A:\n");
 tempfile = fopen("A:temp.dat", "w");
 printf("fopen returned %p\n", tempfile);
 return 0;
}

```

## highvideo 选择高亮度字符

**用 法** #include <conio.h>  
void highvideo(void);

**原 型 在** conio.h

**说 明** highvideo 设置当前所选择的前景颜色的高亮度位,来选择高亮度字符。  
这个函数不影响当前屏幕上的其它任何字符,只对于使用直接视频输出函数(如 cprintf)显示的字符起作用。

**返 回 值** 无。

**可移植性** 该函数只能在 IBM PC 及其兼容机上使用,且在 Turbo Pascal 中有一个对应的函数。

**参 见** cprintf, cputs, gettextinfo, lowvideo, normvideo, textattr, textcolor

**示 例** #include <conio.h>  
int main(void)  
{  
 clrscr();  
 lowvideo();  
 cprintf("Low Intensity text\r\n");  
 highvideo();  
 gotoxy(1,2);  
 cprintf("High Intensity Text\r\n");  
}

```
 return 0;
}
```

## ■ hypot 计算直角三角形的斜边长 ■

用 法 #include <math.h>

double hypot(double x, double y);

原 型 在 math.h

说 明 hypot 计算  $z$  的值, 其中  $z^2 = x^2 + y^2$  且  $x \geq 0$ .

若直角三角形两直角边分别为  $x, y$ , 则  $z$  为斜边长。

返 回 值 调用成功时, 函数 hypot 返回双精度值  $z$ 。否则, 函数返回 HUGE\_VAL 值, 并置 errno 为:

ERANGE 结果超出范围

hypot 的错误处理程序可通过 matherr 来修改。

可移植性 适用于 UNIX 系统。

示 例

```
#include <stdio.h>
#include <math.h>
int main(void)
{
 double result;
 double x = 3.0;
 double y = 4.0;
 result = hypot(x, y);
 printf("The hypotenuse is: %lf\n", result);
 return 0;
}
```

## ■ imagesize 返回保存位图象所需的缓冲区大小 ■

用 法 #include <graphics.h>

unsigned far imagesize(int left, int top, int right, int bottom);

原 型 在 graphics.h

说 明 imagesize 确定保存位图象所需的存储区大小。如果所选图象需要的存储区大于或等于 64K-1 字节, imagesize 返回 0xFFFF(-1)。

返 回 值 imagesize 返回所需存储区的大小, 用字节数表示。

可移植性 该函数只适用于 Turbo C, 而且只能在装有图形适配器的 IBM-PC 及其兼容机上使用。

参 见 getimage, putimage

示 例

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
```



```

#define ARROW _SIZE 10
void draw _arrow(int x, int y);
int main(void)
{
 /* request autodetection */
 int gdriver = DETECT, gmode, errorcode;
 void *arrow;
 int x, y, maxx;
 unsigned int size;
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt,");
 getch();
 exit(1); /* terminate with an error code */
 }
 maxx = getmaxx();
 x = 0;
 y = getmaxy() / 2;
 /* draw the image to be grabbed */
 draw _arrow(x, y);
 /* calculate the size of the image */
 size = imagesize(x, y - ARROW _SIZE, x + (4 * ARROW _SIZE), y + ARROW _
 SIZE);
 /* allocate memory to hold the image */
 arrow = malloc(size);
 /* grab the image */
 getimage(x, y - ARROW _SIZE, x + (4 * ARROW _SIZE), y + ARROW _SIZE,
 arrow);
 /* repeat until a key is pressed */
 while (! kbhit())
 {
 /* erase old image */
 putimage(x, y - ARROW _SIZE, arrow, XOR _PUT);
 x += ARROW _SIZE;
 if (x >= maxx)
 x = 0;
 /* plot new image */
 putimage(x, y - ARROW _SIZE, arrow, XOR _PUT);
 }
}

```

```

 }
 /* clean up */
 free(arrow);
 closegraph();
 return 0;
}

void draw_arrow(int x, int y)
{
 /* draw an arrow on the screen */
 moveto(x, y);
 linerel(4 * ARROW_SIZE, 0);
 linerel(-2 * ARROW_SIZE, -1 * ARROW_SIZE);
 linerel(0, 2 * ARROW_SIZE);
 linerel(2 * ARROW_SIZE, -1 * ARROW_SIZE);
}

```

## ■ **initgraph** 初始化图形系统

用 法 #include <graphics.h>

```
void far initgraph(int far * graphdriver, int far * graphmode, char far *
 pathtodriver);
```

原 型 在 graphics.h

说 明 **initgraph** 装入一个图形驱动程序(或选择一个已注册的驱动程序),用来初始化图形系统,并将系统置为图形模式。

为了启动图形系统,首先要调用 **initgraph** 函数。**initgraph** 装入图形驱动程序,并将系统置为图形模式。**initgraph** 可以使用由用户指定的一个特殊的图形驱动程序和模式,或在运行时自动检测当前的视频适配器,并选择一相应的驱动程序。如果用户告诉 **initgraph** 进行自动检测,它就调用 **detectgraph** 来选择一个图形驱动程序和模式。**initgraph** 还将所有的图形设置(当前位置、调色板、颜色、视区等)复位为它们的缺省值,并置 **graphresult** 为 0。

在一般情况下,**initgraph** 首先为装入的图形驱动程序分配存储空间(用 **\_graphgetmem**),然后从磁盘装入适当的 .BGI 文件。除了这种装入机制外,也可以将一个图形驱动程序文件(或几个)直接同可执行文件连接起来。

**pathtodriver** 指定 **initgraph** 寻找图形驱动程序的目录路径。**initgraph** 首先在 **pathtodrive** 指明的路径寻找,若未找到该文件,再在当前目录中寻找。因此,如果 **pathtodriver** 为空(NULL),则驱动程序文件(\*.BGI)必须在当前目录中。**settextstyle** 按同样方式搜索矢量字体文件(\*.CHR)。

\* **graphdriver** 是一个整数,它指定所要使用的图形驱动程序。用户可以用 **graphics\_drivers** 枚举类型中的常量对它赋值,常量在 **graphics.h** 中定义。

图形驱动程序表

| graphics_drivers | 数值      |
|------------------|---------|
| DETECT           | 0(自动检测) |
| CGA              | 1       |
| MCGA             | 2       |
| EGA              | 3       |
| EGA64            | 4       |
| EGAMONO          | 5       |
| IBM8514          | 6       |
| HERCMONO         | 7       |
| ATT400           | 8       |
| VGA              | 9       |
| PC3270           | 10      |

\* graphmode 是一个整数,它指明初始图形(除非 \* graphdriver=DETECT,在这种情况下,\* graphmode 被 initgraph 函数设置为检测到的驱动程序可用最高分辨率)。用户可以使用 graphics\_modes 枚举类型中的常量对它赋值,常量在 graphics.h 中定义。

图型模式表中,调色板一栏里的 C0,C1,C2 和 C3 指在 CGA(或兼容系统)中可用的四种预定义的四色调色板,在每个调色板中可以选择背景颜色(入口项为 0),但其它颜色固定不变。在本书的第十四章“屏幕文本和图形程序设计”中有调色板的详细描述。

调色板颜色模式表

| 调色板号 | 分配给象素值的颜色  |              |           |
|------|------------|--------------|-----------|
|      | 1          | 2            | 3         |
| 0    | LIGHTGREEN | LIGHTRED     | YELLOW    |
| 1    | LIGHTCYAN  | LIGHTMAGENTA | WHITE     |
| 2    | GREEN      | RED          | BROWN     |
| 3    | CYAN       | MAGENTA      | LIGHTGRAY |

图形表

| 图形驱动程序 | graphis_modes | 值 | 列×行     | 调色板     | 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 color | 1  |
| MCGA   | MCGACO        | 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 color   | 1     |
|          | MCGAHI     | 5 | 640×480  | 2 color   | 1     |
| EGA      | EGALO      | 0 | 640×200  | 16 color  | 4     |
|          | EGAHI      | 1 | 640×350  | 16 color  | 2     |
| EGA64    | EGA64LO    | 0 | 640×200  | 16 color  | 1     |
|          | EGA64HI    | 1 | 640×350  | 4 color   | 1     |
| EGA-MONO | EGAMONOH   | 3 | 640×350  | 2 color   | 1 *   |
|          | EGAMONOHHI | 3 | 640×350  | 2 color   | 2 * * |
| HERC     | HERCMONOH  | 0 | 720×348  | 2 color   | 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 color   | 1     |
|          | ATT400HI   | 5 | 640×400  | 2 color   | 1     |
| VGA      | VGALO      | 0 | 640×200  | 16 color  | 2     |
|          | VGAMED     | 1 | 640×350  | 16 color  | 2     |
|          | VGAHI      | 2 | 640×480  | 16 color  | 1     |
| PC3270   | PC3270HI   | 0 | 720×350  | 2 color   | 1     |
| IBM8514  | IBM8514HI  | 0 | 1024×768 | 256 color |       |
|          | IBM8514LO  | 0 | 640×480  | 256 color |       |

\* 64K EGAMONO 卡

\* \* 256K EGAMONO 卡

调用 `initgraph` 之后, \* `graphdriver` 被设置为当前图形驱动程序; `graphmode` 为当前图形模式。

**返回值** `initgraph` 所设置的内部错误代码, 如果该函数执行成功, 置代码为 0; 如果发生了错误, \* `graphdriver` 被置为 -2, -3, -4 或 -5, `graphresult` 返回同样的值, 如下表示:

|                              |    |              |
|------------------------------|----|--------------|
| <code>grNotDetected</code>   | -2 | 无效图形卡        |
| <code>grFileNotFound</code>  | -3 | 找不到驱动程序文件    |
| <code>grInvalidDriver</code> | -4 | 无效驱动程序       |
| <code>grNoLoadMem</code>     | -5 | 无足够内存来加载驱动程序 |

**可移植性** 只适用于 Turbo C, 而且只能在装有图形适配器的 IBMPC 及其兼容机上使用。

**参 见** `closegraph`, `detectgraph`, `getdefaultpalette`, `getdrivername`, `getgraphmode`, `getmoderange`, `graphdefaults`, `_graphgetmem`, `graphresult`, `installuserdriver`, `registerbgidriver`, `registergbifont`, `restorecrtmode`, `setgraphbufsize`, `setgraphmode`

**示 例**

```
#include <graphics.h>
#include <stdlib.h>
```

```

#include <stdio.h>
#include <conio.h>
int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;

 /* initialize graphics mode */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* return with error code */
 }
 /* draw a line */
 line(0, 0, getmaxx(), getmaxy());
 /* clean up */
 getch();
 closegraph();
 return 0;
}

```

## ■ **inport** **inp** 从端口中读入一个字

**用 法** #include <dos.h>

int inport(int portid);

**原 型 在** dos.h

**说 明** inport 作用与 80x86 指令 IN 相同, 它从由 portid 指定的输入端口读入一字的低位字节, 从 portid+2 指定的输入端口读入高位字节。inp 是与函数 inport 功能相同的宏。

**返 回 值** 返回读入的值。

**可移植性** 仅适用于 8086 系列。

**参 见** inportb, outport, outportb

**示 例** #include <stdio.h>

#include <dos.h>

int main(void)

{

int result;

int port = 0; /\* serial port 0 \*/

```

 result = inport(port);
 printf("Word read from port %d = 0x%X\n", port, result);
 return 0;
}

```

## ■ inportb 从端口中读入一个字节

**用 法** #include <dos.h>

unsigned char inportb(int portid);

**原 型 在** dos.h

**说 明** inportb 是从 portid 指定端口读入一字节的宏。如果在调用 inportb 时包含了 dos.h, 则把 inportb 当作宏看待, 并扩展为插入代码。如果没有包含 dos.h 文件, 或虽然包含了 dos.h 但使用了 undef inportb 指令, 则 inportb 被当作函数看待。

**返 回 值** 返回所读入的值。

**可移植性** 仅适用于 8086 系列。

**参 见** inport, outport, outportb

**示 例** #include <stdio.h>

#include <dos.h>

int main(void)

{

unsigned char result;

int port = 0; /\* serial port 1 \*/

result = inportb(port);

printf("Byte read from port %d = 0x%X\n", port, result);

return 0;

}

## ■ inlsine 在文本窗口插入一空行

**用 法** #include <conio.h>

void inlsine<void>;

**原 型 在** conio.h

**说 明** inlsine 用当前背景颜色, 在文本窗口的光标位置处插入一空行, 空行下面的所有各行都下移一行, 最底行滚出窗口底部。

inlsine 用于文本方式。

**返 回 值** 无。

**可移植性** 只适用于 IBM PC 及其兼容机, 在 Turbo Pascal 中有相应的函数。

**参 见** clrscr, delline, window

**示 例** #include <conio.h>

int main(void)

{

clrscr();

```

printf("INLINE inserts an empty line in the text window\r\n");
printf("at the cursor position using the current text\r\n");
printf("background color. All lines below the empty one\r\n");
printf("move down one line and the bottom line scrolls\r\n");
printf("off the bottom of the window.\r\n");
printf("\r\nPress any key to continue:");
gotoxy(1, 3);
getch();
insline();
getch();
return 0;
}

```

### ■ installuserdriver 安装设备驱动程序到 BGI 设备驱动程序表中 ■

**用 法** #include <graphics.h>

int far installuserdriver(char far \*name, int huge (\*detect)(void));

**原 型 在** graphics.h

**说 明** installuserdriver 允许用户在 BGI 内部表中增加计算机厂商的设备驱动程序, 参数 name 是新设备驱动程序的名字(.BGI), 参数 detect 是一个指针, 指向新的驱动程序的一个可选自动检测函数。自动检测函数不需要参数, 返回一整型值。

有两种使用厂商提供的驱动程序的方法:

假定用户使用一块新的 SGA 视频卡, 该 SGA 卡厂商提供了一个 BGI 设备驱动程序(SGA.BGI), 那么使用该驱动程序最简单的方法是调用 installuserdriver 来安装程序, 直接把返回值(赋给驱动程序的序号)传递给 initgraph.

另一种方法更通用。把这个设备驱动程序同 initgraph 的一个自动检测函数连结起来(假定 SGA 厂商已把这个自动检测函数提供给用户)。当用户通过调用 installuserdriver 安装驱动程序时, 必须传递这个函数的地址及设备驱动程序的文件名。

当用户装入了设备驱动程序名和 SGA 自动检测函数后, 调用 initgraph 并使其完成自动检测过程。在 initgraph 调用自动检测函数(detectgraph)之前, 首先调用 SGA 自动检测函数。如果 SGA 自动检测函数时没有发现 SGA 硬件, 它返回值是 -11 (grError), initgraph 继续进行硬件检查(包括按用户装入的顺序调用由其它厂商提供的自动检测函数)。但是, 如果自动检测函数确定 SGA 存在, 则它返回一个非负数, 于是 initgraph 查找并装入 SGA.BGI, 把硬件置为自动检测函数提供的缺省图形方式, 然后把控制权交给用户程序。

用户一次最多可安装 10 个设备驱动程序。

**返 回 值** 本函数返回设备驱动程序号, 它作为参数传递给 initgraph, 用于选择新安装的设备驱动程序。

**可移植性** 此函数只适用于 Turbo C, 而且只能在装有图形适配器的 IBM PC 及其兼容机上使用。

参 见 initgraph, registerbgidriver

```

示 例 #include <graphics.h>
 #include <stdlib.h>
 #include <stdio.h>
 #include <conio.h>
 /* function prototypes */
 int huge detectEGA(void);
 void checkerrors(void);
 int main(void)
 {
 int gdriver, gmode;
 /* install a user written device driver */
 gdriver = installuserdriver("EGA", detectEGA);
 /* must force use of detection routine */
 gdriver = DETECT;
 /* check for any installation errors */
 checkerrors();
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* check for any initialization errors */
 checkerrors();
 /* draw a line */
 line(0, 0, getmaxx(), getmaxy());
 /* clean up */
 getch();
 closegraph();
 return 0;
 }
 /* detects EGA or VGA cards */
 int huge detectEGA(void)
 {
 int driver, mode, sugmode = 0;
 detectgraph(&driver, &mode);
 if ((driver == EGA) || (driver == VGA))
 /* return suggested video mode number */
 return sugmode;
 else
 /* return an error code */
 return grError;
 }
 /* check for and report any graphics errors */
 void checkerrors(void)
 {

```



```

int errorcode;
/* read result of last graphics operation */
errorcode = graphresult();
if (errorcode != grOk)
{
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1);
}
}

```

### ■ installuserfont 安装未嵌入 BGI 系统的字体文件(.CHR) ■

**用 法** #include <graphics.h>

int far installuserfont(char far \* name);

**原 型** 在 graphics.h

**说 明** name 是一个包含矢量字体的字体文件的路径名,一次最多可安装 20 种字体。

**返 回 值** installuserfont 返回字体号 ID,它可传递给 settextstyle 来选择相应的字体。如果内部字体表满,返回值为-11(grError)。

**可移植性** 只适用于 Turbo C,而且只能在装有图形适配器的 IBMPC 及其兼容机上使用。

**参 见** settextstyle

**示 例** #include <graphics.h>

#include <stdlib.h>

#include <stdio.h>

#include <conio.h>

/\* function prototype \*/

void checkerrors(void);

int main(void)

{

/\* request auto detection \*/

int gdriver = DETECT, gmode,

int userfont;

int midx, midy;

/\* initialize graphics and local variables \*/

initgraph(&gdriver, &gmode, "");

midx = getmaxx() / 2;

midy = getmaxy() / 2;

/\* check for any initialization errors \*/

checkerrors();

/\* install a user defined font file \*/

userfont = installuserfont("USER.CHX");

/\* check for any installation errors \*/

```

 checkerrors();
 /* select the user font */
 setttextstyle(userfont, HORIZ_DIR, 4);
 /* output some text */
 outtextxy(midx, midy, "Testing!");
 /* clean up */
 getch();
 closegraph();
 return 0;
}
/* check for and report any graphics errors */
void checkerrors(void)
{
 int errorcode;
 /* read result of last graphics operation */
 errorcode = graphresult();
 if (errorcode != grOk)
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1);
 }
}

```

## int86 调用 8086 软中断

用 法 #include <dos.h>

int int86(int intno, union REGS \* inregs, union REGS \* outregs);

原 型 在 dos.h

说 明 int86 执行一个由参数 intno 指定的 8086 软中断。在执行中断之前,把 inregs 中的寄存器值拷贝到各寄存器中。

软中断返回后,int86 把当前寄存器的值拷贝到 outregs 中,并把系统进位标志拷贝到 outregs 的 x.cflag 中,把 8086 标志寄存器值拷贝到 outregs 中的 x.flags 中。如果进位标志被置位,则说明发生了错误。

注意,inregs 指向 outregs 所指的同一结构。

返 回 值 int86 在软中断完成后,返回 AX 的值。若进位标志被置位(outregs->x.cflag!=0),

则表示出错,并把全局变量 \_doserrno 置为错误代码。

可移植性 仅适用于 8086 系列处理器。

参 见 bdos, bdosptr, geninterrupt, int86x, intdos, intdosx, intr

示 例 #include <stdio.h>

```

#include <conio.h>
#include <dos.h>
#define VIDEO 0x10
void movetoxy(int x, int y)
{
 union REGS regs;
 regs.h.ah = 2; /* set cursor position */
 regs.h.dh = y;
 regs.h.dl = x;
 regs.h.bh = 0; /* video page 0 */
 int86(VIDEO, ®s, ®s);
}
int main(void)
{
 clrscr();
 movetoxy(35, 10);
 printf("Hello\n");
 return 0;
}

```

## ■ int86x 通用 8086 软中断接口 ■

用 法 #include <dos.h>

```

int int86x(int intno, union REGS * inregs, union REGS * outregs,
 struct SREGS * segregs);

```

原 型 在 dos.h

说 明 int86x 执行一个由参数 intno 指定的 8086 软中断,在执行软中断之前,把 inregs 中的寄存器值拷贝到各寄存器中。

与 int86 不同的是, int86x 在执行软中断之前,把 segregs->ds 和 segregs->es 的值拷贝到相应寄存器中,这一特性允许使用远指针或大模式的程序可以为软中断指定所使用的段。

软中断返回后, int86x 把当前寄存器的值拷贝到 outregs 中,并把系统进位标志拷贝到 outregs 的 x.cflag 字段中,把 8086 标志寄存器拷贝到 outregs 的 x.flags 中。另外, int86x 恢复 DS,并设置 segregs->es 和 segregs->ds 为对应的段寄存器的值。如果进位标志被置位,则说明出现了错误。

int86x 允许调用 8086 软中断,调用时 DS 的值可以不是缺省数据段, ES 也可以用参数来设置。

注意: inregs 能指向 outregs 所指的同一结构。

返 回 值 int86x 软中断完成后返回 AX 的值,若进位标志被置位(outregs->x.cflag != 0)则表示出错,并把 \_doserrno 置为错误代码。

可移植性 仅适用于 8086 系列处理器。

参 见 bdos, bdosptr, geninterrupt, intdos, intdosx, int86, intr, segread

```

示 例 #include <dos.h>
 #include <process.h>
 #include <stdio.h>
 int main(void)
 {
 char filename[80];
 union REGS inregs, outregs;
 struct SREGS segregs;
 printf("Enter filename: ");
 gets(filename);
 inregs.h.ah = 0x43;
 inregs.h.al = 0x21;
 inregs.x.dx = FP_OFF(filename);
 segregs.ds = FP_SEG(filename);
 int86x(0x21, &inregs, &outregs, &segregs);
 printf("File attribute: %X\n", outregs.x.cx);
 return 0;
 }

```

## ■ intdos 通用 DOS 中断接口 ■

用 法 #include <dos.h>  
 int intdos(union REGS \* inregs, union REGS \* outregs);

原 型 在 dos.h

说 明 intdos 执行 DOS 中断 0x21 去调用一指定的 DOS 功能调用, 其中 inregs->h.ah 指定要调用的功能调用。

从中断 0x21 返回后, intdos 将当前寄存器拷贝到 outregs 中, 将系统进位标志状态拷贝到 outregs 的 x.cflag 字段中, 并将 8086 标志寄存器拷贝到 outregs 的 x.flags 字段中。如果进位被置位, 则表示发生了错误。

注意: inregs 可指向由 outregs 所指的同一结构。

返 回 值 intdos 在 DOS 功能调用完成后返回 AX 的值, 如果进位标志被置位 (outregs->x.cflag != 0), 则表示出现了错误, 置 \_doserrno 为错误代码。

可移植性 仅适用于 DOS。

参 见 bdos, bdosptr, geninterrupt, int86, int86x, intdosx, intr

```

示 例 #include <stdio.h>
 #include <dos.h>
 /* deletes file name; returns 0 on success, nonzero on failure */
 int delete_file(char near * filename)
 {
 union REGS regs;
 int ret;
 regs.h.ah = 0x41;

```

```

/* delete file */
regs.x.dx = (unsigned) filename;
ret = intdos(®s, ®s);
/* if carry flag is set, there was an error */
return(regs.x.cflag ? ret : 0);
}

int main(void)
{
 int err;
 err = delete_file("NOTEXIST. $$$");
 if (!err)
 printf("Able to delete NOTEXIST. $$$\n");
 else
 printf("Not Able to delete NOTEXIST. $$$\n");
 return 0;
}

```

## ■ intdosx 通用 DOS 中断接口 ■

用 法 #include <dos.h>

```
int intdosx(union REGS *inregs, union REGS *outregs,
struct SREGS *segregs);
```

原 型 在 dos.h

说 明 intdosx 执行 DOS 中断 0x21 去调用一指定的 DOS 功能调用,其中 inregs->h...ah 指定要调用的功能调用。

与 intdos 的区别为, intdosx 在调用 DOS 功能调用前,把 segregs->ds 和 segregs->es 的值拷贝到对应的寄存器中。这一特性允许使用远指针或大模式的程序可以为执行功能调用指定所使用的段。

从中断 0x21 返回后, intdosx 将当前寄存器值拷贝到 outregs 中,将系统进位标志状态拷贝到 outregs 的 x.cflag 字段中,并将 8086 标志寄存器拷贝到 outregs 的 x.flags 字段中。另外, intdosx 设置 segregs->es 和 segregs->ds 字段为相应段寄存器的值,然后恢复 DS 的值。如果进位标志置位,则表示发生了错误。intdosx 调用 DOS 功能,调用时 DS 可以不是缺省值,可用参数来设置 ES。

注意: inregs 可以指向 outregs 所指的同一结构。

返 回 值 intdosx 在 DOS 功能调用完成后返回 AX 的值。如果进位标志置位 (outregs->x.cflag != 0), 则表明出现了错误, 置 \_doserrno 为错误代码。

可移植性 仅适用于 DOS。

参 见 bdos, bdosptr, geninterrupt, int86, int86x, intdos, intr, segread

示 例 #include <stdio.h>  
#include <dos.h>

```

/* deletes file name; returns 0 on success,
nonzero on failure */
int delete_file(char far * filename)
{
 union REGS regs; struct SREGS sregs;
 int ret;
 regs.h.ah = 0x41; /* delete file */
 regs.x.dx = FP_OFF(filename);
 sregs.ds = FP_SEG(filename);
 ret = intdosx(®s, ®s, &sregs);
 /* if carry flag is set, there was an error */
 return(regs.x.cflag ? ret : 0);
}

int main(void)
{
 int err;
 err = delete_file("NOTEXIST. $ $ $");
 if (! err)
 printf("Able to delete NOTEXIST. $ $ $\n");
 else
 printf("Not Able to delete NOTEXIST. $ $ $\n");
 return 0;
}

```

## ■ intr 改变软中断接口 ■

用 法 #include <dos.h>

void intr(int intno, struct REGPACK \* preg);

原 型 在 dos.h

说 明 intr 函数用于改变执行软中断的接口,它产生一个由 intno 参数指定的软中断。intr 在执行中断之前,把 REGPACK 结构 \* preg 中的寄存器值拷贝到各寄存器中。在软中断完成后, intr 拷贝当前寄存器值到 \* preg 中,包括标志寄存器。传给 intr 的参数如下:

intno      要执行的中断号  
 preg      结构地址,包括  
           (1) 中断调用前的寄存器值  
           (2) 中断调用后的寄存器值

REGPACK 结构 preg(在 dos.h 中定义)具有以下格式:

```

struct REGPACK {
 unsigned r_ax, r_bx, r_cx, r_dx;
 unsigned r_bp, r_si, r_di, r_es, r_flags;
};

```

**返回值** 无返回值。在中断调用后,REGPACK 结构 \*preg 包含了寄存器的值。

**可移植性** 仅适用于 8086 系列处理器。

**参 见** geninterrupt,int86,int86x,intdos,intdosx

**示 例**

```
#include <stdio.h>
#include <string.h>
#include <dir.h>
#include <dos.h>
#define CF 1 /* Carry flag */
int main(void)
{
 char directory[80];
 struct REGPACK reg;
 printf("Enter directory to change to: ");
 gets(directory);
 reg.r_ax = 0x3B << 8; /* shift 3Bh into AH */
 reg.r_dx = FP_OFF(directory);
 reg.r_ds = FP_SEG(directory);
 intr(0x21, ®);
 if (reg.r_flags & CF)
 printf("Directory change failed\n");
 getcwd(directory, 80);
 printf("The current directory is: %s\n", directory);
 return 0;
}
```

## ■ ioctl I/O 设备控制 ■

**用 法** #include <io.h>

int ioctl(int handle,int func[,void \*argdx,int argcx]);

**原 型 在** io.h

**说 明** 这是 DOS 功能调用 0x44 的直接接口 (IOCTL)。

ioctl 用于取得有关设备通道的信息。可以使用正则文件,但限于 func 为 0、6、7 时,对文件的其它所有调用将返回 EINVAL 错误。

有关参数和返回值的详细信息,请参阅《DOS 参考手册》中有关系统调用 0x44 的文档说明。

参数 argdx 和 argcx 是可选的。

ioctl 为特殊函数使用 DOS 设备驱动程序提供一直接的接口,这个函数的功能随不同厂商的硬件和设备而变,而且有些厂商不遵循这里描述的接口。为确切使用 ioctl 函数,请参阅有关厂商 BIOS 的文档说明。

- 
- 0 取得设备信息
  - 1 设置设备信息(在 argdx)
  - 2 把长为 argcx 的字节读入到 argdx 所指的地址中

- 3 从 `argdx` 所指的地址中写长为 `argcx` 的字串
- 4 除了 `handle` 被看作是驱动器号(0=缺省,1=A 等)外与 2 相同
- 5 除了 `handle` 被看作是驱动器号(0=缺省,1=A 盘)外与 3 相同
- 6 取输入状态
- 7 取输出状态
- 8 测试可删除性,只适用于 DOS3.0
- 11 设置共享冲突的重试次数,只适用于 DOS3.0

**返回值** 对于 `func` 取值为 0,1,返回值为设备信息(`ioctl` 调用后的 `DX` 值)。

对于 `func` 取值为 2 到 5,返回值为实际传送的字节数。

对于 `func` 取值为 6,7,返回值为设备状态。

在任何情况下,如果发生错误,则返回 -1,并置全局变量 `errno` 值为下列参数之一:

|                      |       |
|----------------------|-------|
| <code>EINVAL</code>  | 无效参数  |
| <code>EBADF</code>   | 无效文件号 |
| <code>EINVDAT</code> | 无效数据  |

**可移植性** `ioctl` 可用于 UNIX 系统,但参数和功能不同。在 UNIX 版本 7 和系统 III 中,`ioctl` 的实际使用方法也不同。`ioctl` 调用不能移植到 UNIX 中,也很少在 DOS 机器之间移植。

DOS 3.0 扩展了 `ioctl`,`func` 值可以为 8 或 11。

**示例**

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

int main(void)
{
 int stat;
 /* use func 8 to determine if the default drive is removable */
 stat = ioctl(0, 8, 0, 0);
 if (!stat)
 printf("Drive %c is removable.\n", getdisk() + 'A');
 else
 printf("Drive %c is not removable.\n", getdisk() + 'A');
 return 0;
}
```

## ■ `isalnum` 字符分类宏

**用法** `#include <ctype.h>`

`int isalnum(int c);`

**原型在** `ctype.h`

**说明** `isalnum` 是一个通过查表对 ASCII 码整数进行分类的宏。它是一个谓词, `true` 时



返回非零值, false 时返回零。只有在 `isascii(c)` 为 true 或 `c` 是 EOF 时, 这个宏有定义。通过 `#undef isalnum` 可以把这个宏变为函数。

**返回值** 当 `c` 是一字母(A 到 Z 或 a 到 z)或一数字(0~9)时返回非零值。

**可移植性** 可用于 UNIX 系统

**示 例**

```
#include <ctype.h>
#include <stdio.h>
int main(void)
{
 char c = 'C';
 if (isalnum(c))
 printf("%c is alphanumeric\n", c);
 else
 printf("%c isn't alphanumeric\n", c);
 return 0;
}
```

## ■ isalpha 字符分类宏

**用 法** `#include <ctype.h>`  
`int isalpha(int c);`

**原 型** 在 `ctype.h`

**说 明** `isalpha` 是一通过查表对 ASCII 码整数值进行分类的宏。它是一个谓词, true 时返回非零值, false 时返回零, 只有在 `isascii(c)` 为 true 或 `c` 是 EOF 时, 这个宏才有定义。通过 `undef isalpha` 可以把这个宏变为函数。

**返回值** 当 `c` 是一个字母时(A 到 Z 或 a 到 z)时, `isalpha` 返回非零值。

**可移植性** 可用于 UNIX 系统, 与 ANSI C 兼容, 与 Kernighan 和 Ritchie 的定义也兼容。

**示 例**

```
#include <ctype.h>
#include <stdio.h>
int main(void)
{
 char c = 'C';
 if (isalpha(c))
 printf("%c is alphabetic\n", c);
 else
 printf("%c isn't alphabetic\n", c);
 return 0;
}
```

## ■ isascii 字符分类宏

**用 法** `#include <ctype.h>`  
`int isascii(int c);`

**原 型** 在 `ctype.h`

**说 明** isascii 是一个通过查表对 ASCII 码整数进行分类的宏。它是一个谓词, true 时返回非零值, false 时返回零。对于所有整数值, isascii 都有意义。

**返回值** 若 c 的低位字节在 0~127 范围内(0x00~0x7F), 则返回非零值。

**可移植性** 可适用于 UNIX 系统。

**示 例**

```
#include <ctype.h>
#include <stdio.h>
int main(void)
{
 char c = 'C';
 if (isascii(c))
 printf("%c is ascii\n", c);
 else
 printf("%c isn't ascii\n", c);
 return 0;
}
```

## ■ isatty 检查设备类型 ■

**用 法** #include <io.h>  
int isatty(int handle);

**原 型 在** io.h

**说 明** 函数 isatty 确定 handle 是否与下列某一字符设备相联:

- 终端
- 控制台
- 打印机
- 串行口

**返回值** 如果设备是字符设备, isatty 返回一非零值; 如果不是字符设备, 返回 0。

**可移植性** 仅适用于 DOS 系统。

**示 例**

```
#include <stdio.h>
#include <io.h>
int main(void)
{
 int handle;
 handle = fileno(stdprn);
 if (isatty(handle))
 printf("Handle %d is a device type\n", handle);
 else
 printf("Handle %d isn't a device type\n", handle);
 return 0;
}
```

## ■ iscntrl 字符分类宏 ■

**用 法** #include <ctype.h>

```
int iscntrl(int c);
```

原型在 ctype.h

说明 iscntrl 是一个通过查表对 ASCII 码整数值进行分类的宏,它是一个谓词,true 时返回非零值,false 时返回零,iscntrl 仅在 isascii(c)为真或 c 是 EOF 时才有定义。通过 #undef iscntrl,可使这个宏变成函数。

返回值 若 c 是一个删除字符或普通的控制字符(0x7F 或 0x00 到 0x1F)时,返回非零值。

可移植性 可用于 UNIX 系统与 ANSI C 兼容。

示例

```
#include <ctype.h>
#include <stdio.h>
int main(void)
{
 char c = 'C';
 if (iscntrl(c))
 printf("%c is a control character\n",c);
 else
 printf("%c isn't a control character\n",c);
 return 0;
}
```

## ■ isdigit 字符分类宏

用法 #include <ctype.h>

```
int isdigit(int c);
```

原型在 ctype.h

说明 isdigit 是一个通过查表对 ASCII 码整数值进行分类的宏,它是一个谓词,true 时返回非零值,false 时返回零。isdigit 仅在 isascii(c)为真或 c 是 EOF 时才有定义。通过 #undef isdigit,可使这个宏变成函数。

返回值 当 c 是一数字(0~9)时返回非零值。

可移植性 可用于 UNIX 系统,与 ANSI C 兼容,与 Kernighan 和 Ritchie 的定义也兼容。

示例

```
#include <ctype.h>
#include <stdio.h>
int main(void)
{
 char c = 'C';
 if (isdigit(c))
 printf("%c is a digit\n",c);
 else
 printf("%c isn't a digit\n",c);
 return 0;
}
```

## ■ isgraph 字符分类宏

用 法 `#include <ctype.h>`

`int isgraph(int c);`

原 型 在 `ctype.h`

说 明 `isgraph` 是一个通过查表对 ASCII 码整数值进行分类的宏,它是一个谓词,真时返回非零值时,假时返回零,`isgraph` 仅在 `isascii(c)` 为 true 或 `c` 是 EOF 时才有定义。

通过 `#undef isgraph`,可以把这个宏变为函数。

返 回 值 如果 `c` 是一打印字符(与 `isprint` 相似,但不包括空格符),则返回非零值。

可移植性 可用于 UNIX 系统,与 ANSI C 兼容。

示 例 `#include <ctype.h>`

`#include <stdio.h>`

`int main(void)`

{

`char c = 'C';`

`if (isgraph(c))`

`printf("%c is a graphic character\n", c);`

`else`

`printf("%c isn't a graphic character\n", c);`

`return 0;`

}

## ■ islower 字符分类宏

用 法 `#include <ctype.h>`

`int islower(int c);`

原 型 在 `ctype.h`

说 明 `islower` 是一个宏,通过查表对 ASCII 码整数值进行分类,它是一个谓词,true 返回非零值,false 时返回 0,`islower` 仅在 `isascii(c)` 为 true 或 `c` 是 EOF 时有定义。

通过 `#undef islower`,可以把这个宏变为函数。

返 回 值 如果 `c` 是小写字母(a 到 z),返回非零值。

可移植性 可用于 UNIX 系统,与 ANSI C 及 Kernighan 和 Ritchie 的定义兼容。

示 例 `#include <ctype.h>`

`#include <stdio.h>`

`int main(void)`

{

`char c = 'C';`

`if (islower(c))`

`printf("%c is lower case\n", c);`

`else`

```
printf("%c isn't lower case\n",c);
return 0;
}
```

## ■ isprint 字符分类宏

用 法 #include <ctype.h>

int isprint(int c);

原 型 在 ctype.h

说 明 isprint 是一个通过查表对 ASCII 码整数值进行分类的宏。它是一个谓词, true 时返回非零值, false 时返回零, isprint 仅在 isascii(c) 为 true 或 c 是 EOF 时有定义。通过 #undef isprintl 可以把这个宏变为函数。

返 回 值 当 c 是一个可打印符(0x20 到 0x7E)时返回非零值。

可移植性 可用于 UNIX 系统与 ANSI C 兼容。

示 例 #include <ctype.h>

#include <stdio.h>

int main(void)

{

char c = 'C';

if (isprint(c))

printf("%c is a printable character\n", c);

else

printf("%c isn't a printable character\n", c);

return 0;

}

## ■ ispunct 字符分类宏

用 法 #include <ctype.h>

int ispunct(int c);

原 型 在 ctype.h

说 明 ispunct 是一个通过查表对 ASCII 码整数值进行分类的宏。它是一个谓词, true 时返回非零值, false 时返回零, ispunct 仅在 isascii(c) 为 true 和 c 是 EOF 时有定义。

通过 undef ispunctl 可以把这个宏变为函数。

返 回 值 当 c 是标点符(iscntrl 或 isspace)时, 返回非零值。

可移植性 可用于 UNIX 系统, 与 ANSI C 兼容。

示 例 #include <ctype.h>

#include <stdio.h>

int main(void)

{

char c = 'C';

```
if (ispunct(c))
 printf("%c is a punctuation character\n", c);
else
 printf("%c isn't a punctuation character\n", c);
return 0;
}
```

## ■ isspace 字符分类宏

用 法 #include <ctype.h>

int isspace(int c);

原 型 在 ctype.h

说 明 isspace 是一个通过查表对 ASCII 码整数值进行分类的宏。它是一个谓词, true 时返回非零值, false 时返回零, isspace 仅在 isascii(c) 为 true 或 c 是 EOF 时有定义。通过 #undef isspace 可将这个宏变为函数。

返 回 值 如果 c 是空格、制表符、回车、换行、竖向制表符或格式馈送符(0x09 到 0x20)时, 返回非零值。

可移植性 可用于 UNIX 系统, 与 ANSI C 兼容, 与 Kernighan 和 Ritchie 的定义也兼容。

示 例

```
#include <ctype.h>
#include <stdio.h>
int main(void)
{
 char c = 'C';
 if (isspace(c))
 printf("%c is white space\n", c);
 else
 printf("%c isn't white space\n", c);
 return 0;
}
```

## ■ isupper 字符分类宏

用 法 #include <ctype.h>

int isupper(int c);

原 型 在 ctype.h

说 明 isupper 是一个通过查表对 ASCII 码整数值进行分类的宏。它是一个谓词, true 时返回非零值, false 时返回零, isupper 仅在 isascii(c) 为 true 或 c 是 EOF 时有定义。

通过 #undef isupper, 可使这个宏变成函数。

返 回 值 当 c 是一大写字母(A 到 Z)时, 返回非零值。

可移植性 可用于 UNIX 系统与 ANSI C 兼容, 与 Kernighan 和 Ritchie 的定义兼容。

示 例

```
#include <ctype.h>
```

```
#include <stdio.h>
int main(void)
{
 char c = 'C';
 if (isupper(c))
 printf("%c is upper case\n", c);
 else
 printf("%c isn't upper case\n", c);
 return 0;
}
```

## ■ isxdigit 字符分类宏

用 法 #include <ctype.h>

int isxdigit(int c);

原 型 在 ctype.h

说 明 isxdigit 是一个通过查表对 ASCII 码整数值进行分类的宏。它是一个谓词, true 时返回非零值, false 时返回零, isxdigit 仅在 isascii(c) 为 true 或 c 是 EOF 时有定义。

通过 #undef isxdigit, 可使这个宏变成函数。

返 回 值 当 c 是十六进制数(0 到 9、A 到 F、a 到 f)时, 返回非零值。

可移植性 可用于 UNIX 系统, 与 ANSI C 兼容。

示 例 #include <ctype.h>

#include <stdio.h>

int main(void)

{

char c = 'C';

if (isxdigit(c))

printf("%c is hexadecimal\n", c);

else

printf("%c isn't hexadecimal\n", c);

return 0;

}

## ■ itoa 把整数转换为字符串

用 法 #include <stdlib.h>

char \* itoa(int value, char \* string, int radix);

原 型 在 stdlib.h

说 明 itoa 把 value 值转换为以 NULL 终结的字符串, 并把结果存在 string 中。value 为整型。radix 指明在转换 value 过程中的基数, 它必须在 2 到 36 之间。如果 value 为负数, 而 radix 为 10, 则 string 的第一个字符为负号(-)。

注意: 分配给 string 的空间必须可以容纳返回的所有字符(包括空字符、终结符\

0)。itoa 最多能返回 17 个字节。

**返回值** 返回指向 string 的指针。

**可移植性** 仅用于 DOS 系统。

**参见** itoa, ultoa

**示例**

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

int main(void)
{
 int number = 12345;
 char string[25];
 itoa(number, string, 10);
 printf("integer = %d string = %s\n", number, string);
 return 0;
}
```

## ■ kbhit 检查当前按下的键

**用法** #include <conio.h>

int kbhit(void);

**原型在** conio.h

**说明** kbhit 检查按下的键是否有效,如果有效可用 getch 或 getche 读取。

**返回值** 如果按键有效,则 kbhit 返回一非零整数,否则返回 0。

**可移植性** 仅用于 DOS 系统。

**参见** getch, getche

**示例**

```
#include <conio.h>

int main(void)
{
 cprintf("Press any key to continue,");
 while (! kbhit()) /* do nothing */ ;
 cprintf("\r\nA key was pressed...\r\n");
 return 0;
}
```

## ■ keep 驻留并退出

**用法** #include <dos.h>

void keep(unsigned char status, unsigned size);

**原型在** dos.h

**说明** keep 返回到 DOS 提示符状态,把出口状态置为在 status 中的值。当前程序仍驻留在内存中,程序所占用的存储空间为 size,并释放内存的其余部分。

keep 可以用于安装一个 TSR 程序,keep 使用 DOS 功能调用 0x31。

**返回值** 无。



可移植性 仅适用于 DOS 系统。

参 见 abort, exit

示 例 /\* NOTL:

```

 This is an interrupt service routine.
 You can NOT compile this program with Test
 Stack Overflow turned on and get an
 executable file which will operate
 correctly. Due to the nature of this
 function the formula used to compute
 the number of paragraphs may not
 necessarily work in all cases. Use with
 care! Terminate Stay Resident (TSR)
 programs are complex and no other support
 for them is provided. Refer to the
 MS-DOS technical documentation
 for more information. */
#include <dos.h>
/* The clock tick interrupt */
#define INTR 0x1C
/* Screen attribute (blue on grey) */
#define ATTR 0x7900
/* reduce heaplength and stacklength to make
 a smaller program in memory */
extern unsigned _heaplen = 1024;
extern unsigned _stklen = 512;
void interrupt (*oldhandler)(void);
void interrupt handler(void)
{
 unsigned int (far *screen)[80];
 static int count;
 /* For a color screen the video memory
 is at B800:0000. For a monochrome
 system use B000:0000 */
 screen = MK_FP(0xB800, 0);
 /* increase the counter and keep it
 within 0 to 9 */
 count++;
 count %= 10;
 /* put the number on the screen */
 screen[0][79] = count + '0' + ATTR;
 /* call the old interrupt handler */
 oldhandler();
}

```

```

 }
 int main(void)
 {
 /* get the address of the current clock
 tick interrupt */
 oldhandler = getvect(INTR);
 /* install the new interrupt handler */
 setvect(INTR, handler);
 /* _psp is the starting address of the
 program in memory. The top of the stack
 is the end of the program. Using _SS and
 _SP together we can get the end of the
 stack. You may want to allow a bit of
 safety space to insure that enough room
 is being allocated ie:
 (_SS + ((_SP + safety space)/16) - _psp)
 */
 keep(0, (_SS + (_SP/16) - _psp));
 return 0;
 }

```

## labs 给出长型绝对值

用 法 #include <math.h>

long int labs(long int x);

原 型 在 math.h, stdlib.h

说 明 labs 计算参数 x 的绝对值。

返 回 值 返回 x 的绝对值。

可移植性 可用于 UNIX 系统, 与 ANSI C 中的定义兼容。

参 见 abs, cabs, fabs

示 例 #include <stdio.h>

#include <math.h>

int main(void)

{

long result;

long x = -12345678L;

result = labs(x);

printf("number: %ld abs value: %ld\n", x, result);

return 0;

}

## ldexp 计算 x 乘以 2 的 exp 次方

用 法 #include <math.h>

```
double ldexp(double x,int exp);
```

原 型 在 math.h

说 明 ldexp 计算双精度值  $x \times 2^{\text{exp}}$ 。

返 回 值 调用成功时,ldexp 返回它所计算的值  $x \times 2^{\text{exp}}$ 。用 matherr 函数可以修改 ldexp 的错误处理程序。

可移植性 可用于 UNIX 系统,在 ANSI C 中有定义。

参 见 exp,frexp,modf

示 例 #include <stdio.h>  
#include <math.h>

```
int main(void)
{
 double value;
 double x = 2;
 /* ldexp raises 2 by a power of 3
 then multiplies the result by 2 */
 value = ldexp(x,3);
 printf("The ldexp value is: %lf\n", value);
 return 0;
}
```

## ■ ldiv 两个长整型数相除,返回商和余数 ■

用 法 #include <stdlib.h>

```
ldiv_t ldiv(long int numer,long int denom);
```

原 型 在 stdlib.h

说 明 ldiv 将两个长整型数相除,返回 ldiv\_t 类型的商和余数。numer 和 denom 分别是被除数和除数。类型 ldiv\_t 是一个长整型结构,在 stdlib.h 中用 typedef 定义如下:

```
typedef struct {
 long int quot; /* 商 */
 long int rem; /* 余数 */
} ldiv_t;
```

返 回 值 返回 ldiv\_t 类型的商和余数。

可移植性 与 ANSI C 兼容。

参 见 div

示 例 /\* ldiv example \*/  
#include <stdlib.h>  
#include <stdio.h>

```
int main(void)
```

```

{
 ldiv_t lx;
 lx = ldiv(100000L, 30000L);
 printf("100000 div 30000 = %ld remainder %ld\n", lx.quot, lx.rem);
 return 0;
}

```

## ■ lfind 线性搜索 ■

**用 法** `#include <stdlib.h>`  
`void *lfind(const void *key, const void *base, size_t *num, size_t width,`  
`int (*fcmp)(const void *, const void *));`

**原 型 在** `stdlib.h`

**说 明** `lfind` 用于线性搜索一个连续记录数组中关键字为 `KEY` 的记录。它使用用户定义的比较子程序(`fcmp`)。这个数组有 `num` 个记录,每个记录长为 `width`。数组位于由 `base` 指定的存储区的起始地址上。

**返 回 值** 返回与搜索关键字相匹配的第一个表项地址。如果没有匹配项,`lfind` 返回 `NULL`。如果 `*elem1 == *elem2`,比较子程序返回 0,否则返回非零值(`elem` 和 `elem2` 是比较子程序中的两个参数)。

**可移植性** 仅用于 DOS 系统。

**参 见** `bsearch`, `lsearch`, `qsort`

**示 例**

```

#include <stdio.h>
#include <stdlib.h>
int compare(int *x, int *y)
{
 return(*x - *y);
}
int main(void)
{
 int array[5] = {35, 87, 46, 99, 12};
 size_t nelem = 5;
 int key;
 int *result;
 key = 99;
 result = lfind(&key, array, &nelem,
 sizeof(int), (int (*)(const void *, const void *))compare);
 if (result)
 printf("Number %d found\n", key);
 else
 printf("Number %d not found\n", key);
 return 0;
}

```

## line 在指定两点间画一直线

**用 法** #include <graphics.h>

void far line(int x1,int y1,int x2,int y2);

**原 型** 在 graphics.h

**说 明** line 用当前颜色、当前线型和宽度画一直线,这条直线在指定的两点(x1,y1)和(x2,y2)之间,当前位置(CP)不变。

**返 回 值** 无。

**可移植性** 只适用于 Turbo C,而且只能在装有图形适配器的 IBMPC 及其兼容机上使用。

**参 见** getlinesettings,linerel,lineto,setcolor,setlinestyle,setwritemode

**示 例** #include <graphics.h>

#include <stdlib.h>

#include <stdio.h>

#include <conio.h>

int main(void)

{

/\* request auto detection \*/

int gdriver = DETECT, gmode, errorcode;

int xmax, ymax;

/\* initialize graphics and local variables \*/

initgraph(&gdriver, &gmode, "");

/\* read result of initialization \*/

errorcode = graphresult();

/\* an error occurred \*/

if (errorcode != grOk)

{

printf("Graphics error: %s\n", grapherrormsg(errorcode));

printf("Press any key to halt;");

getch();

exit(1);

}

setcolor(getmaxcolor());

xmax = getmaxx();

ymax = getmaxy();

/\* draw a diagonal line \*/

line(0, 0, xmax, ymax);

/\* clean up \*/

getch();

closegraph();

return 0;

}

## ■ **linere1** 从当前位置(CP)到与 CP 有一相对距离的点画一直线 ■

**用 法** #include <graphics.h>

void far linere1(int dx,int dy);

**原 型** 在 graphics.h

**说 明** linere1 从 CP 与到 CP 相对距离为(dx,dy)的点画一直线,同时 CP 位置增加(dx,dy)。

**返 回 值** 无。

**可移植性** 只适用于 Turbo C,而且只能在装有图形适配器的 IBM PC 及其兼容机上使用。

**参 见** getlinesettings,line,lineto,setcolor,setlinestyle,setwritemode

**示 例** #include <graphics.h>

#include <stdlib.h>

#include <stdio.h>

#include <conio.h>

int main(void)

{

/\* request auto detection \*/

int gdriver = DETECT, gmode, errorcode;

char msg[80];

/\* initialize graphics and local  
variables \*/

initgraph(&gdriver, &gmode, "");

/\* read result of initialization \*/

errorcode = graphresult();

if (errorcode != grOk)

{

printf("Graphics error: %s\n", grapherrormsg(errorcode));

printf("Press any key to halt,");

getch();

exit(1);

}

/\* move the C.P. to location (20, 30) \*/

moveto(20, 30);

/\* create and output a

message at (20, 30) \*/

sprintf(msg, "(%d, %d)", getx(), gety());

outtextxy(20, 30, msg);

/\* draw a line to a point a relative

distance away from the current

value of C.P. \*/

linere1(100, 100);

/\* create and output a message at C.P. \*/

```

 sprintf(msg, " (%d, %d)", getx(), gety());
 outtext(msg);
 /* clean up */
 getch();
 closegraph();
 return 0;
}

```

## ■ lineto 从当前位置到(x,y)画一直线 ■

用 法 #include <graphics.h>

void far lineto(int x,int y);

原 型 在 graphics.h

说 明 lineto 从 CP 到(x,y)画一直线,并将 CP 移到(x,y)。

返 回 值 无。

可移植性 只适于 Turbo C,而且只能在装有图形适配器的 IBM PC 及其兼容机上使用。

参 见 getlinesettings,line,linerel,setcolor,setlinestyle,setvisualpage,setwritemode

示 例 #include <graphics.h>

#include <stdlib.h>

#include <stdio.h>

#include <conio.h>

int main(void)

{

/\* request auto detection \*/

int gdriver = DETECT, gmode, errorcode;

char msg[80];

/\* initialize graphics and local variables \*/

initgraph(&gdriver, &gmode, "");

/\* read result of initialization \*/

errorcode = graphresult();

if (errorcode != grOk)

{

printf("Graphics error: %s\n", grapherrormsg(errorcode));

printf("Press any key to halt:");

getch();

exit(1);

}

/\* move the C. P. to location (20, 30) \*/

moveto(20, 30);

/\* create and output a

message at (20, 30) \*/

sprintf(msg, " (%d, %d)", getx(), gety());

outtextxy(20, 30, msg);

```
/* draw a line to (100, 100) */
lineto(100, 100);
/* create and output a message at C. P. */
sprintf(msg, " (%d, %d)", getx(), gety());
outtext(msg);
/* clean up */
getch();
closegraph();
return 0;
}
```

## ■ localtime 把日期和时间转变为结构类型 ■

用 法 #include <time.h>

struct tm \* localtime(const time\_t \* timer);

原 型 在 time.h

说 明 localtime 接受 time 返回值的地址, 返回包含修正时间的 tm 结构的指针。该指针用于修正时间和可能的夏令时间。

全局长变量 timezone 包含 GMT 和当地标准时间差, 单位为秒(在 PST, timezone 为  $8 \times 60 \times 60$ )。全局变量 daylight 为非零值, 当且仅当采用美国标准夏令时时转换。

time.h 文件中 tm 结构如下:

```
struct tm {
 int tm_sec;
 int tm_min;
 int tm_hour;
 int tm_day;
 int tm_mon;
 int tm_year;
 int tm_wday;
 int tm_yday;
 int tm_isdst;
};
```

这些量给出了按 24 小时计算的时间、天数(1~31)、月份(0~11)、周日(星期天为 0)、年号为减去 1990 得到的数、一年中的天数(0~365)和一个非 0 值, 即表示使用夏令时的标志。

返 回 值 返回包含修正时间的结构指针, 该结构是静态的, 每次调用都被重写。

可移植性 可用于 UNIX 系统, 与 ANSI C 兼容。

参 见 asctime, ctime, ftime, gmtime, stime, time, tzset

示 例 #include <time.h>  
#include <stdio.h>



```
#include <dos.h>
int main(void)
{
 time_t timer;
 struct tm *tblock;
 /* gets time of day */
 timer = time(NULL);
 /* converts date/time to a structure */
 tblock = localtime(&timer);
 printf("Local time is: %s", asctime(tblock));
 return 0;
}
```

## lock 设置文件共享锁

**用 法** #include <io.h>

int lock(int handle, long offset, long length);

**原 型** 在 io.h

**说 明** lock 为 DOS 3.x 的文件共享机制提供一个接口。在使用 lock 之前, 必须装入 SHARE.EXE。lock 能放于文件的任一非重叠区域, 一个试图在锁住区进行读/写操作的程序将重试三次。如果三次失败了, 则调用失败, 出现错误。

**返 回 值** 在成功时, 返回 0; 出错时, 返回 -1。

**可移植性** 只适用于 DOS 3.x 版本, 最早版本的 DOS 不支持该调用。

**参 见** open, sopen, unlock

**示 例**

```
#include <io.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <process.h>
#include <share.h>
#include <stdio.h>
```

```
int main(void)
{
 int handle, status;
 long length;
 /* Must have DOS Share.exe loaded for */
 /* file locking to function properly */
 handle = sopen("c:\\autoexec.bat",
 O_RDONLY, SH_DENYNO, S_IREAD);
 if (handle < 0)
 {
 printf("sopen failed\n");
 exit(1);
 }
}
```



## ■ log10 计算 $\log_{10}(x)$ ■

- 用 法** 对于实数 `#include <math.h>` 对于复数 `#include <complex.h>`  
`double log10(double x);` `complex log10(complex x);`
- 原 型** 在 对于实数 `math.h` 对于复数 `complex.h`
- 说 明** `log10(x)` 求以 10 为底的  $x$  的对数。复数的通用对数定义为：  
 $\log_{10}(z) = \log(z) / \log(10)$
- 返 回 值** 当调用成功时, `log10(x)` 返回所计算的值  $\log_{10}(x)$ 。  
 如果传递给 `log10(x)` 的参数  $x$  是一个小于 0 的实数, 全局变量 `errno` 置为:  
     EDOM                      域错误  
`log10(0)` 返回值为负的 `HUGE_VAL`。  
 用 `matherr` 函数可以修改 `log10(x)` 的错误处理程序。
- 可移植性** 实数 `log10(x)` 的可用于 UNIX 系统, 在 ANSI C 中有定义, 复数的 `log10` 仅用于 Turbo C 且不能移植。
- 参 见** `complex, exp, log,`
- 示 例**

```
#include <math.h>
#include <stdio.h>
int main(void)
{
 double result;
 double x = 800.6872;
 result = log10(x);
 printf("The common log of %lf is %lf\n", x, result);
 return 0;
}
```

## ■ longjmp 执行非局部跳转 ■

- 用 法** `#include <setjmp.h>`  
`void longjmp(jmp_buf jmpb, int retval);`
- 原 型** 在 `setjmp.h`
- 说 明** `longjmp` 用参数 `jmpb` 恢复最后一次调用 `setjmp` 捕获的任务状态, 然后返回 `retval` 值。  
 任务状态为:
- 所有的段寄存器(CS, DS, ES, SS)
  - 寄存器变量(SI, DI)
  - 堆栈指针(SP)
  - 标志寄存器
- 一个任务状态对于 `setjmp` 和 `longjmp` 用于合作例程是足够的。

在调用 `longjmp` 前必须先调用 `setjmp`。在调用 `longjmp` 前,调用 `setjmp` 和设置 `jmpb` 的例程必须是活动的,并且不能返回,否则,结果将是不可预测的。

`longjmp` 不能传 0,如果把 0 传给 `retval`,`longjmp` 将用 1 代替。

注意:如果用户程序是覆盖的,不能用 `setjmp` 和 `longjmp` 实现合作例程。通常,`setjmp` 和 `longjmp` 保存并恢复合作例程所需的所有寄存器,但覆盖管理程序需要跟踪堆栈内容,而且只有一个栈。用户完成合作例程时,一般有两个堆栈或者把一个栈分成两部分,因此覆盖管理程序不能正确地跟踪。

用户可以使用利用本身的栈或栈的一部分运行的后台任务,但必须保证背景任务不调用任何覆盖代码,而且不能使用 `setjmp` 和 `longjmp` 的早期版本与后台任务之间来回转换。

返回值 无。

可移植性 仅适用于 UNIX 系统,在 ANSI C 中有定义。

参见 `ctrlbrk`,`setjmp`,`signal`

示例

```
#include <stdio. h>
#include <setjmp. h>
#include <stdlib. h>
void subroutine(jmp _buf);
int main(void)
{
 int value;
 jmp _buf jumper;
 value = setjmp(jumper);
 if (value != 0)
 {
 printf("Longjmp with value %d\n", value);
 exit(value);
 }
 printf("About to call subroutine ... \n");
 subroutine(jumper);
 return 0;
}
void subroutine(jmp _buf jumper)
{
 longjmp(jumper, 1);
}
```

## ■ lowvideo 选择低亮度字符

用法 `#include <conio. h>`

`void lowvideo(void);`

原型在 `conio. h`

说明 `lowvideo` 清除当前前景颜色高亮度位,用来选择低亮度字符。

这个函数不影响当前屏幕上的其它任一字符,只是在此函数调用后,由文本方式直接控制台输出函数所显示的字符变为低亮度。

返回值 无。

可移植性 仅用于 IBM PC 及其兼容机,在 Turbo Pascal 中有一相应的函数。

参 见 highvideo, normvideo, textattr, textcolor

示 例

```
#include <conio.h>

int main(void)
{
 clrscr();
 highvideo();
 cprintf("High Intensity Text\r\n");
 lowvideo();
 gotoxy(1,2);
 cprintf("Low Intensity Text\r\n");
 return 0;
}
```

## lrotl 将无符号长整型数向左循环移位

用 法 include<stdlib.h>

```
unsigned long _lrotl(unsigned long val,int count);
```

原 型 在 stdlib.h

说 明 \_lrotl 将 val 向左循环移动 count 位, val 是一个无符号长整型数。

返回值 \_lrotl 返回 val 向左循环移动 count 位后的值。

可移植性 仅适用于 DOS 系统。

参 见 \_lrotr, \_rotr, \_rotr

示 例

```
/* lrotl example */
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
 unsigned long result;
 unsigned long value = 100;
 result = _lrotl(value,1);
 printf("The value %lu rotated left one bit is, %lu\n", value, result);
 return 0;
}

/* lrotr example */
#include <stdlib.h>
#include <stdio.h>

int main(void)
```

```

{
 unsigned long result;
 unsigned long value = 100;
 result = _lrotr(value,1);
 printf("The value %lu rotated right one bit is, %lu\n",
 value, result);
 return 0;
}

```

## ■ \_lrotr 将无符号长整型数向右循环移位

用 法 #include <stdlib.h>

unsigned long \_lrotr(unsigned long val,int count);

原 型 在 stdlib.h

说 明 \_lrotr 将 val 向右循环移动 count 位, val 是一无符号长整型数。

返 回 值 \_lrotr 返回 val 向右循环移动 count 位后的值。

可移植性 仅用于 DOS 系统。

参 见 \_lrotl, \_rotl, \_rotr

示 例 /\* lrotr example \*/

```

#include <stdlib.h>
#include <stdio.h>
int main(void)
{
 unsigned long result;
 unsigned long value = 100;
 result = _lrotl(value,1);
 printf("The value %lu rotated left one bit is, %lu\n", value, result);
 return 0;
}

/* lrotr example */
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
 unsigned long result;
 unsigned long value = 100;
 result = _lrotr(value,1);
 printf("The value %lu rotated right one bit is, %lu\n",
 value, result);
 return 0;
}

```

## ■ lsearch 线性搜索

用 法 #include <stdlib.h>

```
void *lsearch(const void *key, void *base, size_t *num, size_t width,
 int (*fcmp)(const void *, const void *));
```

原 型 在 stdlib.h

说 明 lsearch 线性地搜索信息, 由于它是线性搜索, 所以在调用前对表中的各表项不用排序, 如果找不到信息, lsearch 把它附加到表中。

- base 指向搜索表的基址(第 0 个元素);
- num 指向包含表中项个数的整数;
- width 包含每一项的字节数;
- key 指向待搜索的项(搜索关键字)。

参数 fcmp 指向用户所写的比较子程序, 该子程序对两项进行比较并返回一比较值。为了搜索该表, lsearch 重复调用由 fcmp 指定的子程序。

在每次调用比较子程序时, 搜索函数 lsearch 传递两个参数: 一个是指向待搜索项的指针 key, 另一个指向待比较元素 elem。

fcmp 自动解释搜索关键字和搜索表项。

返 回 值 返回找到的第一个表项地址。

如果搜索关键字与 \*elem 不同, fcmp 返回非零值; 如果搜索关键字与 \*elem 相同, fcmp 返回 0。

可移植性 可用于 UNIX 系统。

参 见 bsearch, lfind, qsort

示 例

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

int compare(int *x, int *y)
{
 return(*x - *y);
}

int main(void)
{
 int array[5] = {35, 87, 46, 99, 12};
 size_t nelem = 5;
 int key;
 int *result;
 key = 99;
 result = lfind(&key, array, &nelem,
 sizeof(int), (int (*)(const void *, const void *))compare);
 if (result)
 printf("Number %d found\n", key);
 else
 printf("Number %d not found\n", key);
 return 0;
}
```

## ■ lseek 移动文件指针

**用 法** #include <io.h>

long lseek(int handle, long offset, int fromwhere);

**原 型 在** io.h

**说 明** lseek 把文件指针移到 fromwhere 所指的地址加到 offset 新位置的偏移上。最后一个参数 fromwhere 最好设置为以下三个符号常量(在 io.h 中定义)中的一个, 而不设置指定数。三个符号常量分别为:

|             |           |
|-------------|-----------|
| fromwhere   | 文件位置      |
| SEEK_SET(0) | 从文件头位置    |
| SEEK_CUR(1) | 从当前文件指针位置 |
| SEEK_END(2) | 从文件结尾位置   |

**返 回 值** lseek 返回指针新位置的偏移量, 它是相对于文件开始处的, 在出错时, lseek 返回 -1L, 且置 errno 为下列值之一:

|        |       |
|--------|-------|
| EBADF  | 无效文件号 |
| EINVAL | 无效参数  |

如果设备没有查找的能力(如终端和打印机), 则返回值无定义。

**可移植性** 可用于所有 UNIX 系统。

**参 见** filelength, fseek, ftell, getc, open, sopen, ungetc, \_write, write

**示 例**

```
#include <sys/stat.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>

int main(void)
{
 int handle;
 char msg[] = "This is a test";
 char ch;
 /* create a file */
 handle = open("TEST, $$$", O_CREAT | O_RDWR, S_IREAD | S_IWRITE);
 /* write some data to the file */
 write(handle, msg, strlen(msg));
 /* seek to the beginning of the file */
 lseek(handle, 0L, SEEK_SET);
 /* reads chars from the file until we hit EOF */
 do
 {
 read(handle, &ch, 1);
 printf("%c", ch);
 } while (! eof(handle));
}
```



```
close(handle);
return 0;
}
```

## ■ ltoa 把一个长整型数转换为字符串 ■

用 法 #include <stdlib.h>

char \* ltoa(long value, char \* string, int radix);

原 型 在 stdlib.h

说 明 ltoa 把 value 值转换为以空字符终结的字符串,并把结果存在 string 中,value 是一长整形数。

radix 指明在转换 value 过程中的基数值,它必须在 2 到 36 之间,如果 value 是负数,而 radix 为 10,则 string 的第一个字符为(-)。

注意:分配给 string 的空间必须可以容纳所返回的所有字符(包括空字符终结符 \0),ltoa 最多能返回 33 个字节。

返 回 值 ltoa 返回指向 string 的指针。

可移植性 仅用于 DOS

参 见 itoa, ultoa

示 例 #include <stdlib.h>

#include <stdio.h>

int main(void)

```
{
 char string[25];
 long value = 123456789L;
 ltoa(value, string, 10);
 printf("number = %ld string = %s\n", value, string);
 return 0;
}
```

## ■ malloc 分配内存 ■

用 法 #include <stdlib.h>或#include <alloc.h>

void \* malloc(size\_t size);

原 型 在 alloc.h, stdlib.h

说 明 malloc 从堆分配一大小为 size 字节的块,它允许程序按需要分配内存而且恰好分配所需的大小。

堆用于建立可变的存储块的动态内存分配。许多数据结构(如树和表)都自动地使用堆进行分配。

除了堆顶前面紧接着的小边界外,数据末尾与程序之间的所有空间都在小模式程序中是可用的。边界用于应用程序扩充堆栈或 DOS 所需的少量内存。

在大模式程序中,程序堆栈以上直到存储器顶端间的所有空间都被堆所使用。

返 回 值 返回新分配内存的地址,如果没有足够的内存可分配,就返回 NULL,块内容不

变。如果参数 `size=0`, `malloc` 返回 `NULL`。

**可移植性** 可用于 UNIX 系统, 在 ANSI C 中有定义。

**参 见** `allocmem`, `calloc`, `coreleft`, `farcalloc`, `farmalloc`, `free`, `realloc`

**示 例**

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>
#include <process.h>
int main(void)
{
 char *str;
 /* allocate memory for string */
 if ((str = malloc(10)) == NULL)
 {
 printf("Not enough memory to allocate buffer\n");
 exit(1); /* terminate program if out of memory */
 }
 /* copy "Hello" into string */
 strcpy(str, "Hello");
 /* display string */
 printf("String is %s\n", str);
 /* free memory */
 free(str);
 return 0;
}
```

## ■ matherr 用户可修改的数学错误处理程序 ■

**用 法** `#include <math.h>`  
`int matherr(struct exception *e);`

**原 型 在** `math.h`

**说 明** 当数学库产生错误时调用 `matherr`。`matherr` 如同用户的防护罩(一个由用户修改的函数,它可以用用户自己编写的

数学错误程序来代替)。`matherr` 用于捕获数学函数引起的域界错误,它不捕获浮点异常,例如,除数为 0。

用户可以把 `matherr` 修改为自己的错误处理例程(如捕获且处理某一类型的错误),修改后的函数替换 C 库函数中的缺省版本。修改后的 `matherr` 在不能处理错误时,返回 0;否则返回非 0 值。当 `matherr` 返回非 0 值时,不打印错误信息,也不改变 `errno`。

以下是在 `math.h` 中定义的 `exception` 结构:

```
struct exception {
 int type;
 char *function;
```

```
double arg1, arg2, retval;
```

```
};
```

exception 结构成员如下表所示:

| 成员         | 意义                                               |
|------------|--------------------------------------------------|
| type       | 数学错误类型, enum 类型由 typedef. mexcep 定义              |
| name       | 指向引起错误的数学库函数的名字的字符串 (以空字符为终结符) 的指针               |
| arg1, arg2 | 引起错误的参数 (传递给 name 所指的函数); 如果只有一个参数, 于是存在 arg1 中。 |
| retval     | 用户可以修改 matherr 的缺省返回值。                           |

typedef \_mexcep (也在 math.h 中定义) 表示各种可能错误的符号, 如下表所示:

| 符号常量      | 数学错误                                           |
|-----------|------------------------------------------------|
| DOMAIN    | 参数不在函数定义域 (如 $\log(-1)$ )。                     |
| SING      | 参数引起异常情况 (如 $\text{pow}(0, -2)$ )              |
| OVERFLOW  | 参数使函数结果大于 MAXDOUBLE, (如 $\text{exp}(0, 2)$ )   |
| UNDERFLOW | 参数使函数结果小于 MINDOUBLE, 如 ( $\text{exp}(-1000)$ ) |
| TLOSS     | 参数使函数结果失去最高值如 ( $\sin(10e70)$ )                |

符号常量 MAXDOUBLE 和 MINDOUBLE 在 values.h 中定义。

缺省 matherr 的源代码存储在 Turbo C 的源盘上。

UNIX 的 matherr 函数的缺省动作 (打印信息和终止) 与 ANSI 标准不兼容。如果需要 UNIX 的 matherr 版本, 可使用 Turbo C 源盘上提供的 matherr.c 文件。

**返回值** 当错误类型是 UNDERFLOW 或 TLOSS 时, matherr 的返回值是 1, 否则返回值是 0。

matherr 也可以修改  $e \rightarrow \text{retval}$ , 然后传给调用者。

当 matherr 返回 0 时 (表示它不能处理错误), matherr 设置 errno 为 0, 并打印错误信息。

当 matherr 返回非 0 时, 表示它能处理错误, matherr 不设置 errno, 也不打印信息。

**可移植性** matherr 适用于很多 C 编译程序, 但是与 ANSI C 不兼容。在 Turbo C 源盘的 MATHERR.C 中有一打印信息和终止程序运行的 UNIX 型的 matherr。

**示 例**

```
/* This is a user-defined matherr function that prevents
 any error messages from being printed. */
#include <math.h>
int matherr(struct exception *a)
{
 return 1;
}
```

## ■ max 返回两数中较大的数 ■

用 法 #include <stdlib.h>

(type)max(a,b);

原 型 在 stdlib.h

说 明 max 是一个比较两个值,并返回较大一个值的宏。参数和函数说明必须是同一类型。

返 回 值 max 返回两个值中较大的一个。

可移植性 仅用于 DOS。

参 见 min

示 例

```
/* max example */
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
 int x = 5;
 int y = 6;
 int z;
 z = max(x, y);
 printf("The larger number is %d\n", z);
 return 0;
}
```

## ■ memccpy 拷贝一个 n 字节长的字符串 ■

用 法 #include <mem.h>

void \*memccpy(void \*dest,const void \*src,int c,size\_t n);

原 型 在 string.h,mem.h

说 明 memccpy 从 src 中拷贝长度为 n 的字符串到 dest 中,当下列情况之一发生时,拷贝停止:

- (1) 字符 c 被第一次拷贝到 dest 中;
- (2) 已经拷贝了 n 个字节。

返 回 值 如果拷贝了字符 c,memccpy 返回指向 dest 中紧跟 c 以后的字符的指针;否则返回 NULL。

可移植性 仅用于 UNIX 系统 V。

参 见 memcpy,memmove,memset

示 例

```
#include <string.h>
#include <stdio.h>
int main(void)
{
 char *src = "This is the source string";
```

```

char dest[50];
char * ptr;
ptr = memcpy(dest, src, 'c', strlen(src));
if (ptr)
{
 * ptr = '\0';
 printf("The character was found: %s\n", dest);
}
else
 printf("The character wasn't found\n");
return 0;
}

```

## ■ memchr 在字符串中搜索字符

用 法 #include <mem.h>

void \* memchr(const void \* s, int c, size\_t n);

原 型 在 string.h, mem.h

说 明 memchr 在由 s 所指的块的头几个字节中搜索字符。

返 回 值 当函数调用成功时, memchr 返回指向 s 中首次出现 c 的指针, 否则返回 NULL。

可移植性 可用于 UNIX 系统 V, 与 ANSI C 兼容。

示 例 #include <string.h>

#include <stdio.h>

int main(void)

{

char str[17];

char \* ptr;

strcpy(str, "This is a string");

ptr = memchr(str, 'r', strlen(str));

if (ptr)

printf("The character 'r' is at position: %d\n", ptr - str);

else

printf("The character was not found\n");

return 0;

}

## ■ memcmp 比较两个字符串

用 法 #include <mem.h>

int memcmp(const void \* s1, const void \* s2, size\_t n);

原 型 在 string.h, mem.h

说 明 比较 s1 和 s2 两个字符串, 比较长度为 n, 把字节看成是无符号字符型。

返 回 值 由于 memcmp 把字节当作无符号字符型进行比较, 故 memcmp 返回:



```

char dest[] = "abcdefghijklmnopqrstuvwxy0123456709",
char * ptr;
printf("destination before memcpy: %s\n", dest);
ptr = memcpy(dest, src, strlen(src));
if (ptr)
 printf("destination after memcpy: %s\n", dest);
else
 printf("memcpy failed\n");
return 0;
}

```

## ■ memicmp 比较两个字符数组中的 n 个字节,忽略大小写 ■

用 法 #include <mem.h>

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

原 型 string.h, mem.h

说 明 memicmp 比较块 s1 和块 s2 的前 n 个字节,忽略这符大小写。

返 回 memicmp 返回值为:

<0 如果 s1<s2

=0 如果 s1=s2

>0 如果 s1>s2

可移植性 适用于 UNIX 系统 V。

参 见 memcmp

示 例 #include <stdio.h>

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

```
int main(void)
```

```
{
```

```
 char * buf1 = "ABCDE123";
```

```
 char * buf2 = "abcde456";
```

```
 int stat;
```

```
 stat = memicmp(buf1, buf2, 5);
```

```
 printf("The strings to position 5 are ");
```

```
 if (stat)
```

```
 printf("not ");
```

```
 printf("the same\n");
```

```
 return 0;
```

```
}
```

## ■ memmove 拷贝块中的 n 字符 ■

用 法 #include <mem.h>

```
void * memmove(void *dest, const void *src, size_t n);
```

原型在 string.h, mem.h

**说明** memmove 拷贝 src 中的 n 个字符到 dest; 如果 src 和 dest 重叠, 重叠位置的字节也能正确拷贝。

**返回值** 函数返回 dest。

**可移植性** 适用于 UNIX 系统 V, 与 ANSI C 兼容。

**参见** memccpy, memcpy, movmem

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

```
int main(void)
{
 char *dest = "abcdefghijklmnopqrstuvwxyz0123456789";
 char *src = " * * * * *";
 printf("destination prior to memmove: %s\n", dest);
 memmove(dest, src, 26);
 printf("destination after memmove: %s\n", dest);
 return 0;
}
```

## ■ memset 将一个内存块的 n 个字节都设置为 c ■

**用法** #include <mem.h>

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

**原型在** string.h, mem.h

**说明** memset 将数组 s 中头 n 个字节设置为字符 c。

**返回值** 返回 s。

**可移植性** 可用于 UNIX 系统 V, 与 ANSI C 兼容。

**参见** memccpy, memcpy, setmem

**示例** #include <string.h>

```
#include <stdio.h>
#include <mem.h>
int main(void)
{
 char buffer[] = "Hello world\n";
 printf("Buffer before memset: %s\n", buffer);
 memset(buffer, ' ', strlen(buffer) - 1);
 printf("Buffer after memset: %s\n", buffer);
 return 0;
}
```

## ■ min 返回两个值中较小的一个 ■

**用法** #include <stdlib.h>

```
(type) min(a, b);
```

**原型在** stdlib.h

**说明** min 比较两个值, 返回其中较小的一个。参数和函数说明必须是同一个类型。



**返回值** 返回两个值中较小的一个。

**可移植性** 仅用于 DOS 系统。

**参 见** max

**示 例**

```
/* max example */
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
 int x = 5;
 int y = 6;
 int z;
 z = max(x, y);
 printf("The larger number is %d\n", z);
 return 0;
}
```

## ■ mkdir 创建目录

**用 法** #include <dir.h>  
int mkdir(const char \*path);

**原 型** 在 dir.h

**说 明** mkdir 按给定的路径 path 建立一个新目录。

**返回值** 当建立一个新目录成功时, mkdir 返回 0。

在出现错误时, 返回 -1, 并置全局变量 errno 为下列值之一:

|        |           |
|--------|-----------|
| EACCES | 无此权限      |
| ENOENT | 没找到路径或文件名 |

**参 见** chdir, getcurdir, getcwd, rmdir

**示 例**

```
#include <stdio.h>
#include <conio.h>
#include <process.h>
#include <dir.h>

int main(void)
{
 int status;
 clrscr();
 status = mkdir("asdfjklm");
 (1 status) ? (printf("Directory created\n")) :
 (printf("Unable to create directory\n"));
 getch();
 system("dir");
 getch();
}
```

```

status = rmdir("asdfjklm");
(! status) ? (printf("Directory deleted\n")) :
 (perror("Unable to delete directory"));
return 0;
}

```

## ■ MK\_FP 设置一个远指针 ■

用 法 #include <dos.h>

void far \* MK\_FP(unsigned seg, unsigned ofs);

原 型 在 dos.h

说 明 MK\_FP 是一个通过段值(seg)和偏移量(ofs)建立一个远指针的宏。

返 回 值 返回一远指针。

可移植性 仅用于 Turbo C。

参 见 FP\_OFF, FP\_SEG, movedata, segread

示 例

```

/* FP_OFF */
#include <dos.h>
#include <stdio.h>
int main(void)
{
 char *str = "fpoff.c";
 printf("The offset of this file name in memory\
 is: %Fp\n", FP_OFF(str));
 return 0;
}

/* FP_SEG */
#include <dos.h>
#include <stdio.h>
int main(void)
{
 char *filename = "fpseg.c";
 printf("The segment of this file in memory\
 is: %Fp\n", FP_SEG(filename));
 return(0);
}

/* MK_FP */
#include <dos.h>
#include <graphics.h>
int main(void)
{
 int gd, gm, i;
 unsigned int far *screen;
 detectgraph(&gd, &gm);

```

```

if (gd == HERCMONO)
 screen = MK_FP(0xB000, 0);
else
 screen = MK_FP(0xB800, 0);
for (i=0; i<26; i++)
 screen[i] = 0x0700 + ('a' + i);
return 0;
}

```

## ■ mktemp 建立一个唯一的文件名 ■

**用 法** #include <dir.h>

char \*mktemp(char \*template);

**原 型 在** dir.h

**说 明** mktemp 使用一个唯一的文件名来替换字符串 template, 并返回 template。template 应为带有 6 个 X、并以空字符终结的字符串, 这些 X 用字母和一个圆点替换。在新文件名中有两个字母、一个点和三个后缀字母。

新文件名在赋值前, 先查看磁盘中的文件名, 以避免出现相同名称的文件名。

**返 回 值** 如果 template 构造合理, mktemp 返回 template 串的地址; 否则, 返回 NULL。

**可移植性** 用于 UNIX 系统。

**示 例**

```

#include <dir.h>
#include <stdio.h>
int main(void)
{

```

```

 /* fname defines the template for the
 temporary file. */
 char *fname = "TXXXXXX", *ptr;
 ptr = mktemp(fname);
 printf("%s\n", ptr);
 return 0;
}

```

## ■ modf 把双精度数转化为科学计数法 ■

**用 法** #include <math.h>

double modf(double x, double \*ipart);

**原 型 在** math.h

**说 明** modf 把双精度数 x 分成两部分: 整数和小数部分。它把整数保存在 ipart 中, 返回小数部分。

**返 回 值** 返回 x 的小数部分。

**可移植性** 仅用于 DOS。

**参 见** fmod, ldexp

**示 例** #include <math.h>

```
#include <stdio.h>
int main(void)
{
 double fraction, integer;
 double number = 100000.567;
 fraction = modf(number, &integer);
 printf("The whole and fractional parts of %lf are %lf and %lf\n",
 number, integer, fraction);
 return 0;
}
```

## ■ movedata 拷贝数据

用 法 #include <mem.h>

```
void movedata(unsigned srcseg,unsigned src coff,unsigned dstseg,
 unsigned dstoff,size_t n);
```

原型在 mem.h,string.h

说 明 movedata 从源址(srcseg;src off)拷贝 n 个字节的数据到目标地址(dstseg;dst off)。

movedata 是一个不依赖于模式的数据块移动方法。

返回值 无

可移植性 仅用于 DOS

参 见 FP\_OFF,memcpy,MK\_FP,movemem,segread

示 例 #include <mem.h>

```
#define MONO_BASE 0xB000
/* saves the contents of the monochrome screen in buffer */
void save_mono_screen(char near *buffer)
{
 movedata(MONO_BASE, 0, _DS,
 (unsigned)buffer, 80 * 25 * 2);
}
int main(void)
{
 char buf[80 * 25 * 2];
 save_mono_screen(buf);
}
```

## ■ moverel 从当前位置(CP)移动一相对距离

用 法 #include <graphics.h>

```
void far moverel(int dx,int dy);
```

原型在 graphics.h

说 明 moverel 将当前位置(CP)在 x 方向上移动 dx 个像素,在 y 方向移动 dy 个像素。

返回值 无

可移植性 仅用于 Turbo C, 而且只能在装有图形适配器的 IBM PC 及其兼容机上使用。

参 见 moveto

示 例

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 char msg[80];
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
 }

 /* move the C. P. to location (20, 30) */
 moveto(20, 30);
 /* plot a pixel at the C. P. */
 putpixel(getx(), gety(), getmaxcolor());
 /* create and output a message at (20, 30) */
 sprintf(msg, "(%d, %d)", getx(), gety());
 outtextxy(20, 30, msg);
 /* move to a point a relative distance */
 /* away from the current value of C. P. */
 moverel(100, 100);
 /* plot a pixel at the C. P. */
 putpixel(getx(), gety(), getmaxcolor());
 /* create and output a message at C. P. */
 sprintf(msg, "(%d, %d)", getx(), gety());
 outtext(msg);
 /* clean up */
 getch();
 closegraph();
}
```

```
 return 0;
}
```

## ■ movetext 将屏幕上的文本从一个矩形区域拷贝到另一个矩形区域 ■

用 法 #include <conio.h>

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

原 型 在 conio.h

说 明 movetext 将屏幕上由 left、top、right 和 bottom 定义的矩形区域中的内容拷贝到同样大小的新矩形区域中,新矩形的左上角坐标为(destleft,desttop)。坐标均为屏幕绝对坐标,当矩形覆盖时也能被正确地移动。

movetext 是一个可实现直接内存读写的文本方式函数。

返 回 值 如果操作成功,movetext 返回 1;如果操作失败(如所给的坐标超过当前屏幕的范围),movetext 返回 0。

可移植性 适用于 IBM PC 及其兼容系统。

参 见 gettext,puttext

示 例

```
#include <conio.h>
#include <string.h>
int main(void)
{
 char *str = "This is a test string";
 clrscr();
 cputs(str);
 getch();
 movetext(1, 1, strlen(str), 2, 10, 10);
 getch();
 return 0;
}
```

## ■ moveto 从当前坐标位置(CP)移到(x,y) ■

用 法 #include <graphics.h>

```
void far moveto(int x,int y);
```

原 型 在 graphics.h

说 明 moveto 将当前坐标位置(CP)移到视区坐标位置(x,y)

返 回 值 无。

可移植性 只适用于 Turbo C,而且只能在装有图形适配器的 IBM PC 及其兼容机上使用。

参 见 moverel

示 例

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

```

#include <conio.h>
int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 char msg[80];
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
 }
 /* move the C. P. to location (20, 30) */
 moveto(20, 30);
 /* plot a pixel at the C. P. */
 putpixel(getx(), gety(), getmaxcolor());
 /* create and output a message at (20, 30) */
 sprintf(msg, "(%d, %d)", getx(), gety());
 outtextxy(20, 30, msg);
 /* move to (100, 100) */
 moveto(100, 100);
 /* plot a pixel at the C. P. */
 putpixel(getx(), gety(), getmaxcolor());
 /* create and output a message at C. P. */
 sprintf(msg, "(%d, %d)", getx(), gety());
 outtext(msg);
 /* clean up */
 getch();
 closegraph();
 return 0;
}

```

## ■ movmem 移动一长为 length 字节的串

用 法 #include <mem.h>

void movmem(void \*src, void \*dest, unsigned length);

原 型 在 mem.h

说 明 movmem 把长度为 length 字节的字符串从 src 移动到 dest, 如果源块和目标块

重叠,应选择移动方向,以便数据能正确移动。

返回值 无。

可移植性 只适用于 Turbo C。

参见 memcpy, memmove, movedata

示例

```
#include <mem.h>
#include <alloc.h>
#include <stdio.h>
#include <string.h>
int main(void)
{
 char *source = "Borland International";
 char *destination;
 int length;
 length = strlen(source);
 destination = malloc(length + 1);
 movmem(source, destination, length);
 printf("%s\n", destination);
 return 0;
}
```

## ■ normvideo 选择正常亮度字符 ■

用法 #include <conio.h>  
void normvideo(void);

原型在 conio.h

说明 normvideo 通过将文本属性(前景和背景)置为初始值来选择正常亮度字符。这个函数不影响当前屏幕上的任何字符,仅对 normvideo 之后执行直接控制台输出的函数(如 cprintf)所显示的字符起作用。

返回值 无

可移植性 只适用于 IBM PC 及其兼容机,在 Turbo Pascal 中有相应的函数。

参见 highvideo, lowvideo, textattr, textcolor

示例

```
#include <conio.h>
int main(void)
{
 normvideo();
 cprintf("NORMAL Intensity Text\r\n");
 return 0;
}
```

## ■ nosound 关闭 PC 机扬声器 ■

用法 #include <dos.h>  
void nosound(void);



原型在 conio.h

说明 关闭由 sound 打开的扬声器。

可移植性 nosound 适用于 IBM PC 及其兼容机,在 Turbo Pascal 中有相应的函数。

参见 delay, sound

示例 /\* Emits a 7-Hz tone for 10 seconds.

True story, 7 Hz is the resonant  
frequency of a chicken's skull cavity.  
This was determined empirically in  
Australia, where a new factory  
generating 7-Hz tones was located too  
close to a chicken ranch. When the  
factory started up, all the chickens  
died.

Your PC may not be able to emit a 7-Hz tone. \*/

int main(void)

```
{
 sound(7);
 delay(10000);
 nosound();
}
```

## ■ open 打开一个文件进行读或写

用法 #include <fcntl.h>

int \_open(char \*filename, int oflags);

原型在 io.h

说明 \_open 打开由 filename 指定的文件,然后根据 oflags 指定的模式来读或写,文件以二进制方式打开。

在 DOS 2.x 以下版本中, \_open 的 oflags 值只限于 O\_RDONLY, O\_WRONLY 和 O\_RDWR; 在 DOS 3.0 以上, 还可以使用下列值(在 fcntl.h 中定义):

O\_NOINHERIT 当文件没有传送给子程序时包含

O\_DENYALL 只允许当前程序存取文件

O\_DENYWRITE 以只读方式打开

O\_DENYREAD 以只写方式打开

O\_DENYNONE 只允许其它程序共享打开文件

在 DOS 3.x 以上版本的 \_open 调用中, 只允许包含一个 O\_DENYxxx 值。这些文件共享属性在文件上锁时不适用。

由 HANDLE\_MAX 给出同时能打开的最大文件数。

返回值 在调用成功后, \_open 返回文件句柄的非零值, 文件指针(指明文件的当前位置)被置为文件开头。在出错时, \_open 返回 -1, 并置全局变量 errno 为下列值之一:

ENOENT 没有找到路径或文件名

|        |        |
|--------|--------|
| EMFILE | 打开文件太多 |
| EACCES | 无此存取权限 |
| EINVA  | 无效存取码  |

可移植性 仅用于 DOS.

参 见 open, \_read, sopen

示 例

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

int main(void)
{
 int handle;
 char msg[] = "Hello world";
 if ((handle = _open("TEST. $$$", O_RDWR)) == -1)
 {
 perror("Error,");
 return 1;
 }
 _write(handle, msg, strlen(msg));
 _close(handle);
 return 0;
}
```

## ■ open 打开一个文件进行读或写 ■

用 法 #include <fcntl.h>

int open(const char \*path, int acces[, unsigned mode]);

原 型 在 io.h

说 明 open 打开由 path 指定的文件, 然后根据 access 的模式值进行读或写。

若想建立一个特定的文件, 用户可以使用全局变量 \_fmode, 或用 O\_CREAT、O\_TRUNC 选择项调用 open.

例如:

```
open("xmp", O_CREAT/O_TRUNC/O_BINARY, S_IREAD)
```

这个调用将建立一个文件名为 XMP 的二进制只读文件, 如果此文件已存在, 则把它的长度截断为 0.

对于 open 来说, access 是由以下两列表中的 ORing 标志按位运算构成的。

第一个表中, 只能使用一个标志, 第二个表中的标志可以任意逻辑组合使用。

表 1 读/写标志

|          |         |
|----------|---------|
| O_RDONLY | 以只读方式打开 |
| O_WRONLY | 以只写方式打开 |
| O_RDWR   | 以读写方式打开 |

表 2 其它存取标志

|          |                                                             |
|----------|-------------------------------------------------------------|
| O_NDELAY | 未用,用来与 UNIX 兼容。                                             |
| O_APPEND | 若置位,每次写操作前都使文件指针指到文件末尾。                                     |
| O_CREAT  | 如果文件已存在,本标志无作用;如果文件不存在,则文件被创建,mode 用来设置文件置属性位(类似 chmod 函数)。 |
| O_TRUNC  | 如果文件已存在,将文件的长度截为 0,属性不变。                                    |
| O_EXCL   | 只和 O_CREAT 一起使用,如果文件已存在,返回错误代码。                             |
| O_BINARY | 本标志表示文件以二进制方式打开。                                            |
| O_TEXT   | 本标志表示文件以文本方式打开。                                             |

如果没有给出 O\_BINARY 或 O\_TEXT,则文件按全局变量\_fmode 设置的传送方式打开。

如果使用了 O\_CREAT 标志来构造 access,则需要用 sys/stat.h 中定义的下列符号来提供 open 的参数 mode:

| mode 值           | 存取权限 |
|------------------|------|
| S_IWRITE         | 可写   |
| S_IREAD          | 可读   |
| S_IREAD S_IWRITE | 可读/写 |

**返回值** 在调用成功后,open 返回文件句柄,文件指针(指明文件的当前位置)指向文件头,在出错时,open 返回 -1,并置 errno 为下列值之一:

|         |            |
|---------|------------|
| ENOENT  | 没有找到路径或文件名 |
| EMFILE  | 打开文件太多     |
| EACCES  | 无此存取权限     |
| EINVACC | 无效存取码      |

**可移植性** 用于 UNIX 系统。在 UNIX 版本 7 中,O\_type 助记符没有定义,UNIX 系统 II 使用除了 O\_BINARY 和 O\_TEXT 之外的所有助记符。

**参 见** chmod, chsize, close\_creat, creat, creatnew, creattemp, dup, dup2, fdopen, filelength, fopen, freopen, getftime, lseek, lock, \_open, read, sopen, \_write, write

**示 例**

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

```
int main(void)
{
 int handle;
```

```

char msg[] = "Hello world";
if ((handle = open("TEST. $$$", O_CREAT | O_TEXT)) == -1)
{
 perror("Error:");
 return 1;
}
write(handle, msg, strlen(msg));
close(handle);
return 0;
}

```

## ■ outport 输出一个字到端口中

**用 法** #include <dos.h>

void outport(int portid, int value);

**原 型 在** dos.h

**说 明** outport 同 80x86 指令 out 功能相同, 它将给定字 value 的低位字节写到 portid 所指定的输出端口中, 将高位字节写到 portid+1 所指定的输出端口中。outp 是与函数 outport 功能相同的宏。

**返 回 值** 无。

**可移植性** 仅用于 8086 系列。

**参 见** inport, inportb, outportb

**示 例** #include <stdio.h>

#include <dos.h>

int main(void)

{

int value = 64;

int port = 0;

outportb(port, value);

printf("Value %d sent to port number %d\n", value, port);

return 0;

}

## ■ outportb 输出一个字节到端口

**用 法** #include <dos.h>

void outportb(int portid, unsigned char value);

**原 型 在** dos.h

**说 明** outportb 是一个把 value 指定的字节写到由 portid 指定的输出端口的宏。如果调用 outportb 时包含了 dos.h, 则它们当作宏看待, 并扩展为插入代码; 如果没有包含 dos.h 文件, 或虽然包含了 dos.h 但使用了 #undef outpor 指令, 则 outport 被当作函数看待。

**返 回 值** 无。

**可移植性** 仅适用 8086 系列。

**参 见** inport, inportb, outport

**示 例**

```
#include <stdio.h>
#include <dos.h>
int main(void)
{
 int port = 0;
 char value = 'C';
 outportb(port, value);
 printf("Value %c sent to port number %d\n", value, port);
 return 0;
}
```

## ■ outtext 显示一个字符串 ■

**用 法** #include <graphics.h>  
void far outtext(char far \*textstring);

**原 型 在** graphics.h

**说 明** outtext 按照当前对齐方式和当前字体、方向、大小在视区中显示一文本字符串。outtext 在 CP 处输出 textstring。如果水平文本对齐方式是 LEFT\_TEXT, 并且文本方向是 HORIZ\_DIR, 则 CP 的 x 坐标将增大 textwidth(textstring); 否则, CP 不变。

为了在使用几种字体时保持代码的兼容性, 请使用 textheight 函数决定字符串的尺寸大小。

注意: 若利用 outtext 按缺省字体打印字符串, 则超出当前屏幕的字符串被截断。outtext 在图形方式下使用, 不能在文本方式下使用。

**返 回 值** 无。

**可移植性** 仅用于 Turbo C、装有图形适配器的 IBM PC 及其兼容机。

**参 见** gettextsettings, outtextxy, settextjustify, textheight, textwidth

**示 例**

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int midx, midy;
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
```

```

 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
 }
 midx = getmaxx() / 2;
 midy = getmaxy() / 2;
 /* move the C.P. to the center of the screen */
 moveto(midx, midy);
 /* output text starting at the C.P. */
 outtext("This ");
 outtext("is ");
 outtext("a ");
 outtext("test.");
 /* clean up */
 getch();
 closegraph();
 return 0;
}

```

## ■ outtextxy 在指定位置显示一字符串 ■

**用 法** #include <graphics.h>

void far outtextxy(int x,int y,char far textstring);

**原 型 在** graphics.h

**说 明** outtextxy 按照当前对齐方式和当前字体、方向、大小在视区中的给定位置(x,y)显示一文本字符串。

为了在使用几种字体时保持代码的兼容性,请使用 textwidth 和 textheight 函数决定字符串的尺寸大小。

注意:若使用 outtext 或 outtextxy 按缺省字体打印字符串,则超出当前视区之外字符串部分被截断。

outtext 用在图形方式,不能用在文本方式下。

**返 回 值** 无。

**可移植性** 仅适用于 Turbo C、装有图形适配器的 IBM PC 及其兼容机。

**参 见** gettextsettings, outtext, textwidth

**示 例**

```

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

```

```

int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int midx, midy;
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
 }
 midx = getmaxx() / 2;
 midy = getmaxy() / 2;
 /* output text at the center of the screen */
 /* Note: the C.P. doesn't get changed. */
 outtextxy(midx, midy, "This is a test.");
 /* clean up */
 getch();
 closegraph();
 return 0;
}

```

## ■ parsfnm 分析文件名 ■

**用 法** #include <dos.h>

char \* parsfnm(const char \* cmdline, struct fcb \* fcb, int opt);

**原 型 在** dos.h

**说 明** parsfnm 分析 cmdline 所指向的文件名字符串, 该串通常为命令行。该文件名(包括驱动器, 文件名和扩展名)被存放在由 fcb 所指定的文件控制块中。

对于文件名进行详细分析, 可参考《DOS 参考手册》中有关系统调用 0x29 部分的说明。

**返 回 值** 在成功分析一个文件名后, parsfnm 返回指向紧跟文件名后的字节的指针。如果在分析过程中出错, parsfnm 返回 NULL。

**可移植性** 仅用于 DOS 系统。

**示 例** #include <process.h>

#include <string.h>

#include <stdio.h>

```
#include <dos.h>
int main(void)
{
 char line[80];
 struct fcb blk;
 /* get file name */
 printf("Enter drive and file name (no path, e.g., a:file.dat)\n");
 gets(line);
 /* put file name in fcb */
 if (parsfnm(line, &blk, 1) == NULL)
 printf("Error in parsfnm call\n");
 else
 printf("Drive # %d Name: %11s\n", blk.fcb_drive, blk.fcb_name);
 return 0;
}
```

## ■ peek 返回由 segment: offset 指定的内存中的字 ■

用 法 #include <dos.h>

int peek(unsigned segment, unsigned offset);

原 型 在 dos.h

说 明 peek 返回由 segment: offset 指定的内存中的字。如果在调用 peek 时包含了 dos.h 文件, 则它将被当作扩展为插入代码的宏。如果没有包含 dos.h (或者虽包含了 dos.h 但使用了 #undef peek), 则得到的是函数而不是宏。

返 回 值 peek 返回内存地址 segment: offset 中字的值。

可移植性 仅用于 8086 系列。

参 见 harderr, peekb, poke

示 例

```
#include <stdio.h>
#include <conio.h>
#include <dos.h>
int main(void)
{
 int value = 0;
 printf("The current status of your keyboard is:\n");
 value = peek(0x0040, 0x0017);
 if (value & 1)
 printf("Right shift on\n");
 else
 printf("Right shift off\n");
 if (value & 2)
 printf("Left shift on\n");
 else
 printf("Left shift off\n");
}
```



```

 if (value & 4)
 printf("Control key on\n");
 else
 printf("Control key off\n");
 if (value & 8)
 printf("Alt key on\n");
 else
 printf("Alt key off\n");
 if (value & 16)
 printf("Scroll lock on\n");
 else
 printf("Scroll lock off\n");
 if (value & 32)
 printf("Num lock on\n");
 else
 printf("Num lock off\n");
 if (value & 64)
 printf("Caps lock on\n");
 else
 printf("Caps lock off\n");
 return 0;
}

```

### ■ peekb 返回由 segment:offset 指定的内存中的字节 ■

**用 法** #include <dos.h>

char peekb(unsigned segment, unsigned offset);

**原 型 在** dos.h

**说 明** peekb 返回内存地址 segment:offset 中字节的值。如果在调用 peekb 时包含了 dos.h 文件,则它将被当作扩展为插入代码的宏。如果没有包含 dos.h(或者虽包含了 dos.h,但使用了 #undef peekb),则得到的是函数而不是宏。

**返 回 值** 返回内存地址 segment:offset 中字节的值。

**可移植性** 仅用于 8086 系列。

**参 见** peek, pokeb

**示 例**

```

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

int main(void)
{
 int value = 0;
 printf("The current status of your keyboard is:\n");
 value = peekb(0x0040, 0x0017);
 if (value & 1)

```

```
 printf("Right shift on\n");
else
 printf("Right shift off\n");
if (value & 2)
 printf("Left shift on\n");
else
 printf("Left shift off\n");
if (value & 4)
 printf("Control key on\n");
else
 printf("Control key off\n");
if (value & 8)
 printf("Alt key on\n");
else
 printf("Alt key off\n");
if (value & 16)
 printf("Scroll lock on\n");
else
 printf("Scroll lock off\n");
if (value & 32)
 printf("Num lock on\n");
else
 printf("Num lock off\n");
if (value & 64)
 printf("Caps lock on\n");
else
 printf("Caps lock off\n");
return 0;
}
```

## ■ perror 打印系统错误信息 ■

**用 法** #include <stdio.h>

void perror(const char \*s);

**原 型** 在 stdio.h

**说 明** perror 打印错误信息到 stderr 流(通过控制台),表示最后一个库程序的系统错误信息。首先打印参数 string,接着是冒号,然后是对应于当前 errno 值的信息,最后换行。规定传送的文件名是字符串参数。

通过全局变量 sys\_errlist 获取信息字符串数组,errno 可以作为数组的下标以查找对应错误码的字符串,字符串不包括换行符。

全局变量 sys\_nerr 包含数组中的项数。

**返 回 值** 无。

**可移植性** 可用于 UNIX 系统,在 ANSI C 中有定义。

参 见 clearerr, eof, \_strerror, strerror

```
示 例 #include <stdio.h>
int main(void)
{
 FILE *fp;
 fp = fopen("perror.dat", "r");
 if (!fp)
 perror("Unable to open file for reading");
 return 0;
}
```

## ■ pieslice 绘制并填充扇形

用 法 #include <graphics.h>  
void far pieslice(int x, int y, int stangle, int endangle, int radius);

原 型 在 graphics.h

说 明 pieslice 以 (x, y) 为中心, 以 radius 为半径, 从起始角 stangle 到终止角 endangle 绘制并填充一扇形, pieslice 用当前的画线颜色画出扇形外轮廓, 然后用当前填充模式和填充颜色填充。

pieslice 的角度是逆时针方向的, 以度为单位, 0 度为 x 轴正向, 90 度为 y 轴正向。

注意: 如果使用 CGA 彩显或单色显示器, 书中关于如何使用图形函数的例子可能产生不出预想的结果。如果用户使用 CGA 或者单色显示器, 用值 1 代替符号颜色常量, 并可参考在 arc 的例子。

返 回 值 无。

可移植性 仅用于 Turbo C, 而且只能在装有图形适配器的 IBM PC 及其兼容机上使用。

参 见 fillellipse, fill\_patterns(枚举类型), graphresult, sector, setfillstyle

```
示 例 #include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int midx, midy;
 int stangle = 45, endangle = 135, radius = 100;
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
```

```

{
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
}

midx = getmaxx() / 2;
midy = getmaxy() / 2;
/* set fill style and draw a pie slice */
setfillstyle(EMPTY_FILL, getmaxcolor());
pieslice(midx, midy, stangle, endangle, radius);
/* clean up */
getch();
closegraph();
return 0;
}

```

### ■ poke 在由 segment:offset 指定的内存中存储一个字

用 法 #include <dos.h>

void poke(unsigned segment, unsigned offset, int value);

原 型 在 dos.h

说 明 poke 将整数 value 存到由 segment:offset 指定的内存中。

在调用这个例程中,如果包含了 dos.h,则它将被当作扩展为插入代码的宏,如果没有包含 dos.h(或虽然包含了 dos.h,但使用了 #undef poke),得到的将是函数而不是宏。

返 回 值 无。

可移植性 仅用于 8086 系列。

参 见 harderr, peek, pokeb

示 例 #include <dos.h>

#include <conio.h>

int main(void)

```

{
 clrscr();
 cprintf("Make sure the scroll lock key is off and press any key\r\n");
 getch();
 poke(0x0000, 0x0417, 16);
 cprintf("The scroll lock is now on\r\n");
 return 0;
}

```

### ■ pokeb 在由 segment:offset 指定的内存中存储一个字节

用 法 #include <dos.h>

```
void pokeb(unsigned segment,unsigned offset,char value);
```

原型在 dos.h

说明 pokeb 将字节 value 存储到 segment:offset 确定的内存中。

在调用这个例程中,如果包含了 dos.h,则它将被当作扩展为插入代码的宏,如果没有包含 dos.h(或虽然包含了 dos.h,但使用了 #undef pokeb)得到的将是函数而不是宏。

返回值 无。

可移植性 仅用于 8086 系列。

参见 peekb,poke

```
示 例 #include <dos.h>
#include <conio.h>
int main(void)
{
 clrscr();
 cprintf("Make sure the scroll lock key is off and press any key\r\n");
 getch();
 pokeb(0x0000,0x0417,16);
 cprintf("The scroll lock is now on\r\n");
 return 0;
}
```

## poly 根据参数产生一个多项式

用法 #include <math.h>

```
double poly(double x,int degree,double coeffs[]);
```

原型在 math.h

说明 poly 产生一个 degree 次的 x 的多项式,系数为 coeffs[0],coeffs[1],...,coeffs[degree],如 n=4 产生的多项式为,

$$\text{coeffs}[4]x^4 + \text{coeffs}[3]x^3 + \text{coeffs}[2]x^2 + \text{coeffs}[1]x + \text{coeffs}[0]$$

返回值 poly 返回给定 x 的多项式值。

可移植性 可于 UNIX 系统。

```
示 例 #include <stdio.h>
#include <math.h>
/* polynomial: x ** 3 - 2x ** 2 + 5x - 1 */
int main(void)
{
 double array[] = { -1.0, 5.0, -2.0, 1.0 };
 double result;
 result = poly(2.0, 3, array);
 printf("The polynomial: x ** 3 - 2.0x ** 2 + 5x - 1 at 2.0 is %lf\n",
 result);
}
```

```

 return 0;
}

```

## ■ pow 计算 x 的 y 次方 ■

用 法 对于实数

```

#include <math.h>
double pow(double x, double y);

```

对于复数

```

#include <complex.h>
complex pow(complex x, complex y);
complex pow(complex x, double y);
complex pow(double x, complex y);

```

原 型 在 对于实数

math.h

对于复数

complex.h

说 明 pow 计算 x 的 y 次方。

复数 pow 定义为:

$\text{pow}(\text{base}, \text{expon}) = \exp(\text{expon} \log(\text{base}))$

返 回 值 调用成功时, pow 返回计算的值  $x^y$ 。

有时候, 传给这些函数的参数会产生结果溢出或不可计算。当正确的值溢出时, pow 返回 HUGE\_VAL, 如果结果值很大, 将置 errno 为:

ERANGE 结果超出范围

当传递给 pow 的 x 参数是一个小于 0 的实数, 且 y 不是完全数, 全局变量 errno 设置为:

EDOM 域错误

如果传递给 pow 的参数 x 和 y 都是 0, pow 返回 1。

可以用 matherr 函数修改 pow 的错误处理程序。

可移植性 实数 pow 可用于 UNIX 系统, 在 ANSI C 中有定义。复数的 pow 只适用于 Turbo C 并且不能移植。

参 见 complex, exp, pow10, sqrt

示 例 #include <math.h>

#include <stdio.h>

```

int main(void)
{
 double x = 2.0;
 double y = 3.0;
 printf("%lf raised to %lf is %lf\n", x, y, pow(x, y));
 return 0;
}

```

## ■ pow10 指数函数 10 的 p 次方 ■

用 法 #include <math.h>

double pow10(int p);

原型在 math.h

说明 pow10 计算  $10^p$ 。

用长双精度精确计算结果。尽管有一些参数可能导致下溢或上溢,但各种参数都是有效的。

可移植性 可用于 UNIX 系统。

参见 exp, pow

示例 #include <math.h>  
#include <stdio.h>

```
int main(void)
{
 double p = 3.0;
 printf("Ten raised to %H is %lf\n", p, pow10(p));

 return 0;
}
```

## ■ printf 写格式化输出到 stdout ■

用法 include<stdio.h>

int printf(const char \*format[format[,argument,...]]);

原型在 stdio.h

说明 printf 接受一组参数,由 format 给定的格式规定输出格式,把数据格式化并且输出到 stdout。该函数要求格式指示符的数目和参数的数目必须相同。

### 格式字符串

每个...printf 函数调用中的格式字符串用于控制函数转换、格式化和输出其参数的方式。对于每个格式字符串,必须有相应的参数与之对应,否则,函数执行时会出现意想不到的后果。过多的参数(超过格式所要求的)将被忽略。格式字符串包括两类对象——简单字符(plain characters)和转换规范(conversion specifications)串:

(1) 简单字符只是简单地将字符拷贝到输出流中。

(2) 转换规范从参数表中取参数,并对它们进行格式化

### 格式规范:

...printf 的格式规范有以下形式:

% [flags] [width] [.prec] [F|N|h|I|L] type

每一个转换规范以百分号(%)开始,后面按顺序为:

- (1) 可选的标志字符序列[flags]。
- (2) 可选的宽度规范符[width]。
- (3) 可选的精度规范符[.prec]。
- (4) 可选的输入长度修饰符[F|N|h|I|L]。
- (5) 转换类型字符。

### 可选的格式字符串成分

以下列出在格式字符串中可选的字符,规范符和修改符所控制的格式输出的几个方面:

| 字符或指示符    | 控制或规范什么                                    |
|-----------|--------------------------------------------|
| flags     | 输出对齐,数值符号,小数点,尾零,八进或十六进制数                  |
| width     | 打印字符、填补空格或零的最少数                            |
| precision | 打印字符最多个数;对于整型值,为输出最少数字个数                   |
| size      | 覆盖缺省的参数大小(N=近指针,F=远指针,H=短整型,I=长整型,L=长双精度型) |

### ...printf 转换

下表列出...printf 的转换类型字符、所接受的输入参数类型和输出的格式。

#### 类型字型

如果格式规范符中没有标志字符、宽度规范符、精度规范符或输入大小修饰符,函数要求给出类型字符表中的信息。可选的字符和规范符对...printf 输出的影响见下表:

| 类型符        | 输入参数    | 输出格式                                                                       |
|------------|---------|----------------------------------------------------------------------------|
| <b>数学类</b> |         |                                                                            |
| d          | 整数      | 带符号的十进制整数                                                                  |
| i          | 整数      | 带符号的十进制整数                                                                  |
| o          | 整数      | 无符号的八进制整数                                                                  |
| u          | 整数      | 无符号的十进制整数                                                                  |
| x          | 整数      | 无符号的十六进制整数(十六进制数字、字母使用 a、b、c、d、e、f)                                        |
| X          | 整数      | 无符号十六进制整数(十六进制数字、字母使用 A、B、C、D、E、F)                                         |
| f          | 浮点数     | 格式为[-]dddd.dddd 的带符号的数值                                                    |
| e          | 浮点数     | 格式为[-]dddd 或 e[+/-]ddd 带符号的数值                                              |
| g          | 浮点数     | 由给定值和精度确定的 e 或 f 格式的带符号的值,如果必要,在尾部输出零和小数点                                  |
| E          | 浮点数     | 同 e 相似,只是用 E 表示指数                                                          |
| G          | 浮点数     | 同 g 相似,只是用 e 格式时,用 E 表示指数                                                  |
| <b>字符类</b> |         |                                                                            |
| c          | 字符      | 单个字符                                                                       |
| s          | 字符串     | 输出终结符前的字符或按要求的精度输出字符                                                       |
| %          | 无       | 输出 % 字符                                                                    |
| <b>指针类</b> |         |                                                                            |
| n          | 指向整数的指针 | 把已写字符的个数保存到由输入参数的所指定的位置中按指针输出所输入的参数,输出方式可指定为 XXXX;YYYY(段,偏移量)或 YYYY(只是偏移量) |
| p          | 指针      |                                                                            |



## 转换

下表列出这些规范符的一些转换形式:

| 字符    | 转换                                                                                                                                                        |
|-------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
| e 或 E | 参数被转换成 $[-]d.ddd\dots e[+/-]ddd$<br>其中:<br><ul style="list-style-type: none"> <li>● 在小数点前有一位数字</li> <li>● 小数点后的数字个数等于精度</li> <li>● 指数总是至少包含两位数</li> </ul> |
| f     | 参数被转换成十进制 $[-]ddd.ddd\dots$ 的格式, 其中小数点后的数字个数等于精度(仅当给出非零精度时)                                                                                               |
| g 或 G | 参数以 e、E 或 f 的格式输出, 精度指明有效数字个数, 尾部的零被从结果中删除, 只有必要的时候才出现小数点, 如果转换字符为 g, 参数则以 e 或 f 格式(带某些限制)输出; 如果转换字符为 G, 参数以 E 格式输出。e 格式只有当转换的结果其指数大于精度或小于 -4 时才使用。       |
| x 或 X | x 转换, 把字母 a、b、c、d、e 和 f 作为输出的十六进制字母; 而 X 转换把字母 A、B、C、D、E 和 F 作为输出的十六进制字母。                                                                                 |

注意: 无穷浮点数打印为 +INF 和 -INF, IEEE 标准的“非数字”(Not-a-Number)

打印为 +NAN 或 -NAN.

## 标志字符

标志字符为减号(-)、加号(+)、#号和空格, 它们可以用任意顺序和以任意组合出现。

| 标志 | 所指示意义                                  |
|----|----------------------------------------|
| -  | 结果左对齐, 右边填充格; 若不给定的话, 则结果右对齐, 左边填充格或零。 |
| +  | 带符号的转换, 结果总以加号(+)或减号(-)开头              |
| 空格 | 如果值为非负值, 则输出以空格代替加号, 负值以减号开头。          |
| #  | 指定参数 arg 以选择格式转换, 见下表。                 |

注意: 如果同时给出加号和空格, 加号优先。

## 选择格式

如果 # 号标志 # 和转换字符一起使用, 将对被转换的参数(arg)产生如下影响:

## 宽度指示符

宽度指示符设置输出值的最小字段宽度。

宽度用两种方法指定: 直接方法是用十进制数字字符串, 间接方法是用一星号(\*)。如果使用星号作为宽度指示符, 相应的下一个参数(必须为整数)指出最小的输出字段宽度。即使没有字段宽度指示或者宽度太小, 也不会导致输出字段截断。如果转换后的结果大于段宽, 则字段被扩展到能包含转换结果的长度。

| 转换字符      | #号如何影响 arg                                                  |
|-----------|-------------------------------------------------------------|
| c,s,d,i,u | 无影响                                                         |
| 0         | 0 被预置于非零参数 arg 前面                                           |
| x 或 X     | 0x(0X)被预置于参数 arg 前面                                         |
| e,E 或 f   | 结果总是包含小数点,即便小数点后没有数字。一般情况下只有当小数点后有数字时,才出现小数点。               |
| g 或 G     | 同 e 和 E 相同,只是删除末尾的 0。                                       |
| 宽度指示符     | 输出宽度影响                                                      |
| n         | 至少输出 n 个字符,如果结果值少于 n 个字符,则用空格填充(在给定“-”标志时,右边填充空格,否则左边填充空格)。 |
| 0n        | 至少输入 n 个字符,如果结果值少于 n 个字符,左边填零。                              |
| *         | 参数表提供宽度指示符,在实际格式化的参数的前面。                                    |

### 精度规范符

精度指示符总是以点(.)开头,用以区别任何前导的宽度指示符。同宽度指示符一样,精度指示符也可以用直接方法(通过十进制数字串)或间接方法(通过星号)给出。如果使用星号(\*)作为精度指示符,下一参数(整型数)应该指明精度。

如用星号来说明宽度或精度或两者,则宽度参数必须紧跟在星号后,然后是精度参数,最后为要转换的实际参数。

| 精度指示符 | 输出精度影响                                                                                                                            |
|-------|-----------------------------------------------------------------------------------------------------------------------------------|
| 无     | 精度为缺省值。<br>对于 d,i,o,u,x 类型,缺省值等于 1。<br>对于 e,E,f 类型,缺省值等于 6。<br>对于 g,G 类型,缺省值等于所有有效数字。<br>对于 s 类型,缺省值等于打印直到第一个空字符。<br>对于 c 类型,无影响。 |
| .0    | 对于 d,i,o,u,x 类型,精度为其缺省值。<br>对于 e,E,f 类型,不输出小数点。                                                                                   |
| .n    | 输出 n 个字母或 n 个十进制数字。<br>如果输出多于 n 个字符,输出被截断(根据类型字符而定)。                                                                              |
| .*    | 参数表提供精度指示,必须在实际格式化的参数前给定。                                                                                                         |

注意:如果指定的精度是 0,并且该字段的格式规范符是整数类型(即:d,i,o,u,x)之一,并且打印的值为 0,则这个字段没有任何数字字符输出(也就是,该字段为空)。

| 转换字符  | 精度指示符(.n)对转换的影响                    |
|-------|------------------------------------|
| d,i   | .n 指明至少有 n 个数字要输出。                 |
| o,u   | 当输入参数少于 n 个数字时,在输出字符中的左边填零;        |
| x,X   | 当输入参数多于 n 个数字时,输出不被截断;             |
| e,E,f | .n 指明在小数点后面有 n 个字符要输出,最后一位数字被四舍五入。 |
| g,G   | .n 指明最多输出 n 个有效数字。                 |
| c     | .n 对输出无影响。                         |
| s     | .n 指明输出不多于 n 个字符。                  |

### 输入长度修饰符

输入长度修饰符(L、N、h、I 或 L)给出后续输入参数的大小。

F=近指针

N=远指针

h=short int 型

I=long 型

L=long double 型

输入长度修饰符(F、N、h、I、L)影响...printf 函数对对应的输入参数 arg 数据类的解释。只有当输入参数 arg 为指针时(%p、%s、%n),才使用 F 和 N。只有输入参数 arg 为数字值(整型或浮点型)时才用 h、I 和 L。F 和 N 都重新解释输入字段。通常为 %p、%s、%n 转换的 arg 是一个内存缺省大小的指针。F 将 arg 解释为一个远指针,N 把 arg 解释为近指针。

h、I 和 L 用 \* 代替数字类型参数的缺省大小,I 和 L 用于整型(d、i、o、u、x、X)和浮点型(e、E、f、g、G),而 h 只用于整型。h 和 I 都不影响字符(c、s)和指针类型(p、n)的输入长度修饰符。

| 输入长度修饰符 | 如何解释 arg                                                         |
|---------|------------------------------------------------------------------|
| F       | 作为远指针读 arg。                                                      |
| N       | 作为近指针读 arg,N 不能用于任何巨型程序下的转换。                                     |
| h       | 将 d、i、o、u、x 和 X 的 arg 解释成 short int。                             |
| I       | 将 d、i、o、u、x 和 X 的 arg 解释成 long int;将 arg 为 e、E、f、g、G 解释成 double。 |
| L       | 将 arg 为 e、E、f、g、G 解释成 long double。                               |

**返回值** printf 返回输出对象的字节数。如果出错,则返回 EOF。

**移植性** 可用于 UNIX 系统,在 ANSI C 中有定义,与 Kernighan 和 Ritchie 的定义兼容。

**参见** cprintf,ecvt,fprintf,fread,fscanf,putc,puts,putw,scanf,sprintf,vprintf

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

```

#define I 555
#define R 5.5
int main(void)
{
 int i,j,k,l;
 char buf[7];
 char *prefix = buf;
 char tp[20];
 printf("prefix 6d 6o 8x 10.2e "
 "10.2f\n");
 strcpy(prefix, "%");
 for (i = 0; i < 2; i++)
 {
 for (j = 0; j < 2; j++)
 for (k = 0; k < 2; k++)
 for (l = 0; l < 2; l++)
 {
 if (i==0) strcat(prefix, "-");
 if (j==0) strcat(prefix, "+");
 if (k==0) strcat(prefix, "#");
 if (l==0) strcat(prefix, "0");
 printf("%5s |", prefix);
 strcpy(tp, prefix);
 strcat(tp, "6d |");
 printf(tp, I);
 strcpy(tp, "");
 strcpy(tp, prefix);
 strcat(tp, "6o |");
 printf(tp, I);
 strcpy(tp, "");
 strcpy(tp, prefix);
 strcat(tp, "8x |");
 printf(tp, I);
 strcpy(tp, "");
 strcpy(tp, prefix);
 strcat(tp, "10.2e |");
 printf(tp, R);
 strcpy(tp, prefix);
 strcat(tp, "10.2f |");
 printf(tp, R);
 printf(" \n");
 strcpy(prefix, "%");
 }
 }
}

```

```
 }
 return 0;
}
```

## ■ putc 输出一个字符到流中

用 法 include<stdio.h>

int putc(int c, FILE \* stream);

原 型 在 stdio.h

说 明 putc 是一个把字符输出到由 stream 指定的流中的宏。

返 回 值 该函数执行成功时,则 putc 返回打印的字符 c,而执行失败时,则 putc 返回 EOF。

可移植性 可用于 UNIX 系统,在 ANSI C 中有定义,与 Kernighan 和 Ritchie 的定义兼容。

参 见 fprintf, fputc, fputch, fputchar, fputs, fwrite, getc, getchar, printf, putchar, putw, vprintf

示 例 #include <stdio.h>

```
int main(void)
{
 char msg[] = "Hello world\n";
 int i = 0;
 while (msg[i])
 putc(msg[i++], stdout);
 return 0;
}
```

## ■ putch 向屏幕输出字符

用 法 include<conio.h>

int putch(int c);

原 型 在 conio.h

说 明 putch 输出字符 c 到当前文本屏幕,它是一个进行直接对显示缓冲区进行读写的文本方式函数,putch 不把换行字符(\n)解释成回车/换行。

根据全局变量 directvideo 的值,字符串通过直接写到显示缓冲区上或者通过调用 BIOS 来输出。

返 回 值 在执行成功时,putch 返回打印字符 c;而在执行失败时,返回 EOF。

参 见 cprintf, cputs, getch, getche, putc, putchar

示 例 #include <stdio.h>

```
#include <conio.h>
int main(void)
{
 char ch = 0;
 printf("Input a string,");
```

```
while ((ch != '\r'))
{
 ch = getch();
 putchar(ch);
}
return 0;
}
```

## ■ putchar 在 stdout 上输出字符 ■

用 法 #include <stdio.h>

int putchar(int c);

原 型 在 stdio.h

说 明 putchar(c)是一个定义为 putc(c, stdout)的宏。

返 回 值 在调用成功时, putchar 返回字符 c, 在失败时, putchar 返回 EOF。

可移植性 可用于 UNIX 系统, 在 ANSI C 中有定义, 与 Kernighan 和 Ritchie 的定义兼容。

参 见 fputc, getc, getchar, printf, putc, putchar, puts, putw, vprintf

示 例 #include <stdio.h>

```
/* define some box-drawing characters */
#define LEFT_TOP 0xDA
#define RIGHT_TOP 0xBF
#define HORIZ 0xC4
#define VERT 0xB3
#define LEFT_BOT 0xC0
#define RIGHT_BOT 0xD9

int main(void)
{
 char i, j;
 /* draw the top of the box */
 putchar(LEFT_TOP);
 for (i=0; i<10; i++)
 putchar(HORIZ);
 putchar(RIGHT_TOP);
 putchar('\n');
 /* draw the middle */
 for (i=0; i<4; i++)
 {
 putchar(VERT);
 for (j=0; j<10; j++)
 putchar(' ');
 putchar(VERT);
 putchar('\n');
 }
}
```

```

 /* draw the bottom */
 putchar(LEFT_BOT);
 for (i=0; i<10; i++)
 putchar(HORIZ);
 putchar(RIGHT_BOT);
 putchar('\n');
 return 0;
}

```

## ■ putenv 将字符串放入当前环境中 ■

**用 法** include<stdlib.h>

```
int putenv(const char * name);
```

**原 型 在** stdlib.h

**说 明** putenv 接受字符串 name, 把它加到当前程序运行的环境中, 例如:

```
putenv("PATH=C:\\TC");
```

putenv 也可用于修改或删除一个已存在的 name 变量值为空(删除一个已存在项可以通过使变量值为空, 例如 MYVAR=)。

putenv 用于修改当前程序的环境, 当程序结束时原环境将得到恢复。

注意: putenv 的参数字符串必须是静态的或全局的, 如果 putenv 使用局部或动态字符串, 则存储字符串的内存释放后就会生产意想不到的结果。

**返 回 值** 当执行成功时, putenv 返回 0; 当执行失败时, 返回 -1。

**可移植性** 可用于 UNIX 系统。

**参 见** getenv

**示 例**

```

#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <string.h>
#include <dos.h>

int main(void)
{
 char * path, * ptr;
 int i = 0;
 /* get the current path environment */
 ptr = getenv("PATH");
 /* set up new path */
 path = malloc(strlen(ptr)+15);
 strcpy(path, "PATH=");
 strcat(path, ptr);
 strcat(path, ";c:\\temp");
 /* replace the current path and display current environment */
 putenv(path);
}

```

```

while (environ[i])
 printf("%s\n", environ[i++]);
return 0;
}

```

## ■ putimage 输出一个位图象到图形屏幕上 ■

**用 法** #include <graphics.h>

void far putimage(int left, int top, void far \* bitmap, int op);

**原 型 在** graphics.h

**说 明** putimage 将 getimage 的位图象(源位图象)输出到屏幕上。图象的左上角的坐标为(left, top), bitmap 指向源图象的内存。putimage 的参数 op 指明了一种方式, 用来控制图形屏幕上象素的颜色, 屏幕上将要显示的点由内存中点与当前屏幕上的点通过某种操作得到。

在 graphics.h 中定义的枚举 putimage\_ops 给出了这些操作的名字:

| 名字       | 值 | 描述         |
|----------|---|------------|
| COPY_PUT | 0 | 原样写到屏幕     |
| XOR_PUT  | 1 | 与屏幕上点异或后写入 |
| OR_PUT   | 2 | 与屏幕上点或后写入  |
| AND_PUT  | 3 | 与屏幕上点与后写入  |
| NOT_PUT  | 4 | 原图象变反写到屏幕  |

比如 COPY\_PUT 将源位图象按原样拷贝到屏幕上, XOR\_PUT 将源图象与当前屏幕上的图象异或, OR\_PUT 将源图象与当前屏幕上图象相“或”之后, 将结果送到屏幕上。

**返 回 值** 无。

**可移植性** 只适用于 Turbo C, 而且只能在装有图形适配器的 IBM PC 及其兼容机上使用。

**参 见** getimage, imagesize, putpixel, setvisualpage

**示 例**

```

#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#define ARROW_SIZE 10
void draw_arrow(int x, int y);
int main(void)
{
 /* request autodetection */
 int gdriver = DETECT, gmode, errorcode;
 void * arrow;
 int x, y, maxx;
 unsigned int size;
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
}

```



```

/* read result of initialization */
errorcode = graphresult();
if (errorcode != grOk) /* an error occurred */
{
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt,");
 getch();
 exit(1); /* terminate with an error code */
}

maxx = getmaxx();
x = 0;
y = getmaxy() / 2;
/* draw the image to be grabbed */
draw_arrow(x, y);
/* calculate the size of the image */
size = imagesize(x, y - ARROW_SIZE, x + (4 * ARROW_SIZE), y + ARROW_SIZE);
/* allocate memory to hold the image */
arrow = malloc(size);
/* grab the image */
getimage(x, y - ARROW_SIZE, x + (4 * ARROW_SIZE), y + ARROW_SIZE,
 arrow);
/* repeat until a key is pressed */
while (!kbhit())
{
 /* erase old image */
 putimage(x, y - ARROW_SIZE, arrow, XOR_PUT);
 x += ARROW_SIZE;
 if (x >= maxx)
 x = 0;
 /* plot new image */
 putimage(x, y - ARROW_SIZE, arrow, XOR_PUT);
}
/* clean up */
free(arrow);
closegraph();
return 0;
}

void draw_arrow(int x, int y)
{
 /* draw an arrow on the screen */
 moveto(x, y);
 linerel(4 * ARROW_SIZE, 0);

```

```

 linerel(-2 * ARROW_SIZE, -1 * ARROW_SIZE);
 linerel(0, 2 * ARROW_SIZE);
 linerel(2 * ARROW_SIZE, -1 * ARROW_SIZE);
 }

```

## ■ putpixel 写像素点 ■

用 法 #include <graphics.h>

void far putpixel(int x,int y,int color);

原 型 在 graphics.h

说 明 putpixel 在(x,y)处,用 color 指定的颜色画一点。

返 回 值 无。

可移植性 只适用于 Turbo C,而且只能在装有图形适配器的 IBM PC 及其兼容机上使用。

参 见 getpixel,putimage

示 例 #include <graphics.h>

```
#include <stdlib.h>
```

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

```
#include <conio.h>
```

```
#include <dos.h>
```

```
#define PIXEL_COUNT 1000
```

```
#define DELAY_TIME 100 /* in milliseconds */
```

```
int main(void)
```

```
{
```

```
 /* request autodetection */
```

```
 int gdriver = DETECT, gmode, errorcode;
```

```
 int i, x, y, color, maxx, maxy, maxcolor, seed;
```

```
 /* initialize graphics and local variables */
```

```
 initgraph(&gdriver, &gmode, "");
```

```
 /* read result of initialization */
```

```
 errorcode = graphresult();
```

```
 if (errorcode != grOk) /* an error occurred */
```

```
{
```

```
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
```

```
 printf("Press any key to halt:");
```

```
 getch();
```

```
 exit(1); /* terminate with an error code */
```

```
}
```

```
 maxx = getmaxx() + 1;
```

```
 maxy = getmaxy() + 1;
```

```
 maxcolor = getmaxcolor() + 1;
```

```
 while (!kbhit())
```

```
{
```

```
 /* seed the random number generator */
```

```

seed = random(32767);
srand(seed);
for (i=0; i<PIXEL_COUNT; i++)
{
 x = random(maxx);
 y = random(maxy);
 color = random(maxcolor);
 putpixel(x, y, color);
}
delay(DELAY_TIME);
srand(seed);
for (i=0; i<PIXEL_COUNT; i++)
{
 x = random(maxx);
 y = random(maxy);
 color = random(maxcolor);
 if (color == getpixel(x, y))
 putpixel(x, y, 0);
}
}
/* clean up */
getch();
closegraph();
return 0;
}

```

## puts 输出一字符串到标准输出(stdout)

**用 法** #include<stdio.h>

int puts(const char \*s);

**原 型** 在 stdio.h

**说 明** puts 拷贝从 NULL 结尾的字符串 s 到标准输出流 stdout 中,并自动加上换行符。

**返 回 值** 当调用成功时,puts 返回正值;否则,返回 EOF。

**可移植性** 可用于 UNIX 系统,在 ANSI C 中有定义。

**参 见** cputs, fputs, gets, printf, putchar

**示 例** #include <stdio.h>

```

int main(void)
{
 char string[] = "This is an example output string\n";
 puts(string);
 return 0;
}

```

## ■ puttext 从内存区拷贝文本到屏幕 ■

用 法 #include <conio.h>

int puttext(int left,int top,int right,int bottom,void \* source);

原 型 在 conio.h

说 明 puttext 将 source 所指的内存中的内容写到由 left,top,right 和 bottom 所定义的矩形屏幕中。

所有的坐标都是内存绝对屏幕坐标,而不是相对于窗口的坐标,屏幕左上角坐标是(1,1)。

puttext 将内存中内容按从左到右,从上而下顺序放在所定义的矩形中。

puttext 是一个执行直接进行显存读写的文本方式函数。

返 回 值 当操作成功时,这个函数返回非 0 值;否则(例如给出的坐标超过当前屏幕的范围)返回 0。

可移植性 只能用于 IBM PC 和与其 BIOS 兼容的系统。

参 见 gettext,movetext>window

示 例 #include <conio.h>

```
int main(void)
{
 char buffer[512];
 /* put some text to the console */
 clrscr();
 gotoxy(20, 12);
 cprintf("This is a test. Press any key to continue ...");
 getch();
 /* grab screen contents */
 gettext(20, 12, 36, 21,buffer);
 clrscr();
 /* put selected characters back to the screen */
 gotoxy(20, 12);
 puttext(20, 12, 36, 21, buffer);
 getch();
 return 0;
}
```

## ■ putw 输出一整数到流中 ■

用 法 #include <stdio.h>

int putw(int w,FILE \*stream);

原 型 在 stdio.h

说 明 putw 向流 stream 输出正数 w,它不会在文件中产生对齐。

返 回 值 当调用成功时,putw 返回整数 w,否则,返回 EOF。因为 EOF 是一个合法整数,

因此可以用 `ferror` 来检测在 `putw` 执行时的错误。

可移植性 可用于 UNIX 系统

参 见 `getw`, `printf`

示 例

```
#include <stdio.h>
#include <stdlib.h>
#define FNAME "test. $$$"

int main(void)
{
 FILE *fp;
 int word;
 /* place the word in a file */
 fp = fopen(FNAME, "wb");
 if (fp == NULL)
 {
 printf("Error opening file %s\n", FNAME);
 exit(1);
 }
 word = 94;
 putw(word, fp);
 if (ferror(fp))
 printf("Error writing to file\n");
 else
 printf("Successful write\n");
 fclose(fp);
 /* reopen the file */
 fp = fopen(FNAME, "rb");
 if (fp == NULL)
 {
 printf("Error opening file %s\n", FNAME);
 exit(1);
 }
 /* extract the word */
 word = getw(fp);
 if (ferror(fp))
 printf("Error reading file\n");
 else
 printf("Successful read: word = %d\n", word);
 /* clean up */
 fclose(fp);
 unlink(FNAME);
 return 0;
}
```

## ■ qsort 用快速排序算法进行排序 ■

用 法 #include <stdlib.h>

```
void qsort(void *base, size_t nelem, size_t width, int (*fcmp)
 (const void *, const void *));
```

原 型 在 stdlib.h

说 明 qsort 采用“中树遍历”快速排序算法,通过重复调用用户定义的比较函数(由 fcmp 所定义),而对表中的元素进行排序。

(1) base 指向待排序表的开头(第 0 个元素),即起始位置;

(2) nelem 是表中的元素个数;

(3) width 是以字节为单位的表中每个元素的长度。

比较函数 \*fcmp 接受两个参数 elem1 和 elem2,每个参数都为指向表中元素的指针。该比较函数比较这两指针的内容(\*elem1 和 \*elem2),并根据比较结果,返回一个整数。

| 条件               | fcmp 返回值        |
|------------------|-----------------|
| *elem1 < *elem2  | fcmp 返回一个整数 < 0 |
| *elem1 == *elem2 | fcmp 返回一个整数 = 0 |
| *elem1 > *elem2  | fcmp 返回一个整数 > 0 |

在比较过程中,小于号(<)表示在最后排完序的序列中,小的元素在前面,同样,大于符号(>)表示在最后排完序的序列中,大的元素在前面。

返 回 值 无。

可移植性 可用于 UNIX 系统,在 ANSI C 中有定义。

参 见 bsearch, lsearch

示 例

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

int sort_function(const void *a, const void *b);

char list[5][4] = { "cat", "car", "cab", "cap", "can" };

int main(void)
{
 int x;
 qsort((void *)list, 5, sizeof(list[0]), sort_function);
 for (x = 0; x < 5; x++)
 printf("%s\n", list[x]);
 return 0;
}

int sort_function(const void *a, const void *b)
{
 return(strcmp(a,b));
}
```

## raise 向正在执行的进程发送一个软中断信号

用 法 #include <signal.h>

int raise(int sig);

原 型 在 signal.h

说 明 raise 向程序发送 sig 类型的信号,如果程序为 sig 指定的类型安装了一个信号处理程序,那么这个信号处理程序就要被执行。如果没有安装信号处理程序,就执行该信号类型的缺省处理。

定义在 signal.h 中的信号类型如下:

| 信号      | 意义            |
|---------|---------------|
| SIGABRT | 非正常终止(*)      |
| SIGFPE  | 浮点运算错误        |
| SIGILL  | 非法的指令(#)      |
| SIGINT  | Ctrl-Break 中断 |
| SIGSEGV | 无效的内存存取权限(#)  |
| SIGTERM | 程序终止(*)       |

标有(\*)的信号类型不能由 DOS 或 Turbo C 在一般操作中产生,但是它们可用 raise 产生。而标有(#)的信号不能在 8086 或 8088 处理器上产生,但可在其它一些处理器上产生(详细内容参看 signal)。

返 回 值 当调用成功时,函数返回 0;否则函数返回非零值。

可移植性 可用于 UNIX 系统,在 ANSI C 中有定义。

参 见 abort, signal

示 例

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

int sort_function(const void *a, const void *b);
char list[5][4] = { "cat", "car", "cab", "cap", "can" };
int main(void)
{
 int x;
 qsort((void *)list, 5, sizeof(list[0]), sort_function);
 for (x = 0; x < 5; x++)
 printf("%s\n", list[x]);
 return 0;
}

int sort_function(const void *a, const void *b)
{
 return(strcmp(a,b));
}
```

## rand 产生随机数

用 法 include <stdlib.h>

```
int rand(void);
```

原 型 在 `stdlib.h`

说 明 `rand` 用 2 步长的倍增器和随机数发生器产生一个随机数, 随机数范围在 0 到 `RAND_MAX` 之间, 符号 `RAND_MAX` 在 `stdlib.h` 中有定义, 它的值为  $2^{31} - 1$ 。

返 回 值 `rand` 返回产生的伪随机数。

可移植性 可用于 UNIX 系统, 在 ANSI C 中有定义。

参 见 `random`, `randomize`, `srand`

示 例

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

int main(void)
{
 int i;
 printf("Ten random numbers from 0 to 99\n\n");
 for(i=0; i<10; i++)
 printf("%d\n", rand() % 100);
 return 0;
}
```

## ■ randbrd 随机块读 ■

用 法 `#include <dos.h>`

```
int randbrd(struct fcb * fcb, int rcnt);
```

原 型 在 `dos.h`

说 明 `randbrd` 从打开后由 `fcb` 所指的 FCB(文件控制块)读取 `rcnt` 个记录。记录被读到当前磁盘传输地址(DTA)所在的内存, 函数从 FCB 记录字段指定位置的记录中读取数据, 由 DOS 系统调用 `0x27` 完成。

实际读取记录数可通过检查 FCB 的随机记录字段来确定, 随机记录字段随着实际读的记录变化。

返 回 值 根据 `randbrd` 操作的结果返回下列值:

- 0 所有记录被读取。
- 1 读到文件尾, 最后一个记录读完。
- 2 读尽可能多的记录, 读到 `0xFFFF` 地址后反绕。
- 3 读到文件末尾, 最后一个记录不完整。

可移植性 只适用于 DOS。

参 见 `getdta`, `randbwr`, `setdta`

示 例

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

int main(void)
```



```

{
 char far * save_dta;
 char line[80], buffer[256];
 struct fcb blk;
 int i, result;
 /* get user input file name for dta */
 printf("Enter drive and file name (no path — i.e. a,file.dat)\n");
 gets(line);
 /* put file name in fcb */
 if (! parsfnm(line, &blk, 1))
 {
 printf("Error in call to parsfnm\n");
 exit(1);
 }
 printf("Drive # %d File: %s\n\n", blk.fcb_drive, blk.fcb_name);
 /* open file with DOS FCB open file */
 bdosptr(0x0F, &blk, 0);
 /* save old dta, and set new one */
 save_dta = getdta();
 setdta(buffer);
 /* set up info for the new dta */
 blk.fcb_recsz = 128;
 blk.fcb_random = 0L;
 result = randbrd(&blk, 1);
 /* check results from randbrd */
 if (! result)
 printf("Read OK\n\n");
 else
 {
 perror("Error during read");
 exit(1);
 }
 /* read in data from the new dta */
 printf("The first 128 characters are:\n");
 for (i=0; i<128; i++)
 putchar(buffer[i]);
 /* restore previous dta */
 setdta(save_dta);
 return 0;
}

```

## ■ randbwr 随机块写

用 法 include<dos.h>

```
int randbwr(struct fcb * fcb,int rcnt);
```

原 型 在 dos.h

说 明 randbwr 向打开的由 fcb 所指的 FCB 写 rcnt 个记录,由 DOS 系统调用 0x28 完成。如果 rcnt 为 0,则文件被截成随机记录字段指明的长度。

实际写的记录数可通过检查 FCB 的随机记录字段而确定,随机记录字段值随实际写的记录数而变。

返 回 值 根据 randbwr 操作的结果返回下列值:

- 0 所有记录已写完。
- 1 磁盘满,无法写。
- 2 尽可能多地写记录,写到 0xFFFF 地址后反绕。

可移植性 仅用于 DOS.

参 见 randbrd

```
示 例 #include <process.h>
#include <string.h>
#include <stdio.h>
#include <dos.h>
int main(void)
{
 char far * save_dta;
 char line[80];
 char buffer[256] = "RANDBWR test!";
 struct fcb blk;
 int result;
 /* get new file name from user */
 printf("Enter a file name to create (no path -- ie. a,file.dat\n");
 gets(line);
 /* parse the new file name to the dta */
 parsfnm(line,&blk,1);
 printf("Drive # %d File: %s\n", blk.fcb_drive, blk.fcb_name);
 /* request DOS services to create file */
 if (bdosptr(0x16, &blk, 0) == -1)
 {
 perror("Error creating file");
 exit(1);
 }
 /* save old dta and set new dta */
 save_dta = getdta();
 setdta(buffer);
 /* write new records */
 blk.fcb_recsize = 256;
 blk.fcb_random = 0L;
```

```

result = randbwr(&blk, 1);
if (! result)
 printf("Write OK\n");
else
{
 perror("Disk error");
 exit(1);
}
/* request DOS services to close the file */
if (bdosptr(0x10, &blk, 0) == -1)
{
 perror("Error closing file");
 exit(1);
}
/* reset the old dta */
setdta(save_dta);
return 0;
}

```

## ■ random 随机数发生器

用 法 #include <stdlib.h>

int random(int num);

原 型 在 stdlib.h

说 明 random 返回一个从 0 到 (num-1) 的随机数。该函数实际上是一个定义在 stdlib.h 中的宏, num 和返回的随机数均为整数。

返 回 值 返回一个范围从 0 到 (num-1) 之间的数。

可移植性 在 Turbo pascal 中有相似的函数。

参 见 rand, randomize, srand

示 例 #include <stdlib.h>

#include <stdio.h>

#include <time.h>

/\* prints a random number in the range 0 to 99 \*/

int main(void)

{

randomize();

printf("Random number in the 0-99 range: %d\n", random(100));

return 0;

}

## ■ randomize 初始化随机数发生器

用 法 #include <stdlib.h>

```
#include <time.h>
```

```
void randomize(void);
```

原型在 `stdlib.h`

说明 `randomize` 用一个随机值对随机数发生器进行初始化, 由于 `randomize` 是作为宏来实现的。因为该宏调用了原型在 `time.h` 中的 `time()` 函数, 因此在使用本函数时, 应包含 `time.h`。

返回值 无。

可移植性 在 Turbo Pascal 中有一相似的函数。

参见 `rand`, `random`, `srand`

示例

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
int main(void)
{
 int i;
 randomize();
 printf("Ten random numbers from 0 to 99\n\n");
 for(i=0; i<10; i++)
 printf("%d\n", rand() % 100);
 return 0;
}
```

## ■ `_read` 读文件

用法 `include<io.h>`

```
int _read(int handle, void * buf, unsigned len);
```

原型在 `io.h`

说明 `_read` 从由 `handle` 指定的文件中读入 `len` 字节到由 `buf` 所指的缓冲区中。`_read` 直接使用了 DOS 系统调用。当从以文本方式打开的文件中读取数据时, `read` 不删除回车符。

`handle` 是从 `creat`, `open`, `dup` 或 `dup2` 调用中得到的句柄。

对于磁盘文件, `_read` 从当前文件指针处开始读。当完成读操作后, 文件指针增量为读入的字节数。对于设备, 直接从设备中读字节。

能读取数据的最大字节数为 65534, 由于 65535(0xFFFF) 与这个函数的错误返回标志 -1 相同, 因此函数最多只能读取 65534 字节的数据。

返回值 当函数调用成功时, 用返回一正整数来表示读入缓冲区的字节数。如果函数读到文件末尾, 则返回 0。在出错时, 它返回 -1, 且置全局变量 `errno` 为下述值之一:

|                     |        |
|---------------------|--------|
| <code>EACCES</code> | 权限错误   |
| <code>EBADF</code>  | 无效文件句柄 |

可移植性 仅用于 DOS。

参见 `open`, `read`, `write`

```

示 例 #include <stdio.h>
 #include <io.h>
 #include <alloc.h>
 #include <fcntl.h>
 #include <process.h>
 #include <sys\stat.h>
int main(void)
{
 void * buf;
 int handle, bytes;
 buf = malloc(10);
 /*
 Looks for a file in the current directory
 named TEST. $ $ $ and attempts to read 10
 bytes from it. To use this example you
 should create the file TEST. $ $ $
 */
 if ((handle =
 open("TEST. $ $ $", O_RDONLY | O_BINARY, S_IWRITE | S_IREAD))
 == -1)
 {
 printf("Error Opening File\n");
 exit(1);
 }
 if ((bytes = _read(handle, buf, 10)) == -1) {
 printf("Read Failed. \n");
 exit(1);
 }
 else {
 printf("_read, %d bytes read. \n", bytes);
 }
 return 0;
}

```

## ■ read 读文件 ■

用 法 #include<io.h>  
int \_read(int handle,void \* buf,unsigned len);

原 型 在 io.h

说 明 read 从与 handle 相联的文件中读取 len 字节到由 buf 所指的缓冲区中。  
对于以文本方式打开的文件,read 删除读入数据中的回车符。当读到 Ctrl-Z 字符时返回文件结束。

handle 是从 creat,open,dup 或 dup2 调用中得到的句柄。

对于磁盘文件, read 从当前文件指针处开始读取数据。在读操作完成后, 文件指针增量为读入的字节数。对于设备, 直接从设备中读字节。

**返回值** 当调用成功时, 返回一正整数以表示读入缓冲区的字节数。如果文件以文本方式打开, 则所读入的字节数不包括回车和 Ctrl-Z 字符。

到达文件尾时, read 返回 0。在出错时, read 返回 -1, 且置 errno 为下述值之一:

|        |       |
|--------|-------|
| EACCES | 权限错误  |
| EBADF  | 无效文件号 |

**可移植性** 适用于 UNIX 系统。

**参见** open, \_read, write

**示例**

```
#include <stdio.h>
#include <io.h>
#include <alloc.h>
#include <fcntl.h>
#include <process.h>
#include <sys\stat.h>
int main(void)
{
 void * buf;
 int handle, bytes;
 buf = malloc(10);

 /*
 Looks for a file in the current directory named TEST. $ $ $ and attempts
 to read 10 bytes from it. To use this example you should create the
 file TEST. $ $ $
 */
 if ((handle =
 open("TEST. $ $ $", O_RDONLY | O_BINARY, S_IWRITE | S_IREAD))
 == -1)
 {
 printf("Error Opening File\n");
 exit(1);
 }
 if ((bytes = read(handle, buf, 10)) == -1) {
 printf("Read Failed. \n");
 exit(1);
 }
 else {
 printf("Read: %d bytes read. \n", bytes);
 }
 return 0;
}
```



```

{
 char *str;
 /* allocate memory for string */
 str = malloc(10);
 /* copy "Hello" into string */
 strcpy(str, "Hello");
 printf("String is %s\n Address is %p\n", str, str);
 str = realloc(str, 20);
 printf("String is %s\n New address is %p\n", str, str);
 /* free memory */
 free(str);
 return 0;
}

```

## ■ rectangle 画一个矩形 ■

**用 法** #include <graphics.h>

void far rectangle(int left, int top, int right, int bottom);

**原 型 在** graphics.h

**说 明** rectangle 用当前线型、宽度和画线颜色绘制一矩形。矩形的左上角为(left, top), 右下角为(right, bottom)。

**返 回 值** 无。

**可移植性** 仅用于 Turbo C, 而且只能使用于装有支持图形适配器的 IBM PC 及其兼容机上。

**参 见** bar, bar3d, setcolor, setlinestyle

**示 例** #include <graphics.h>

#include <stdlib.h>

#include <stdio.h>

#include <conio.h>

int main(void)

{

/\* request auto detection \*/

int gdriver = DETECT, gmode, errorcode;

int left, top, right, bottom;

/\* initialize graphics and local variables \*/

initgraph(&gdriver, &gmode, "");

/\* read result of initialization \*/

errorcode = graphresult();

if (errorcode != grOk) /\* an error occurred \*/

{

printf("Graphics error: %s\n", grapherrormsg(errorcode));

printf("Press any key to halt,");

getch();



```

 exit(1); /* terminate with an error code */
 }
 left = getmaxx() / 2 - 50;
 top = getmaxy() / 2 - 50;
 right = getmaxx() / 2 + 50;
 bottom = getmaxy() / 2 + 50;
 /* draw a rectangle */
 rectangle(left, top, right, bottom);
 /* clean up */
 getch();
 closegraph();
 return 0;
}

```

## ■ registerbgidriver 注册已加载或连接进来的图形驱动程序 ■

**用 法** #include <graphics.h>

```
int registerbgidriver(void (* driver)(void));
```

**原 型 在** graphics.h

**说 明** registerbgidriver 加载并注册一个 BGI 图形驱动程序。一旦内存地址传给了 registerbgidriver, initgraph 函数就可使用这个注册的驱动程序。一个用户注册的驱动程序可以从磁盘加载到堆里,也可以用 BGIOBJ.EXE 转换成 OBJ 文件并连接到可执行文件中。

调用 registerbgidriver 使图形系统连接时包含由 driver 所指的驱动程序。本子程序检查所指定的驱动程序的连接代码;如果代码有效,就把它注册到内部表中。在本书附录部分,有连接驱动程序细节的讨论。

通过在对 registerbgidriver 的调用中使用要连入的驱动程序名,用户使公共名告诉编译程序和连接程序用该公共名来连接目标文件。

registerfarbgidriver 用来注册远程驱动程序,远程驱动程序用 BGIOBJ 实用程序的 /F 开关创建,详细用法在本书附录中有介绍。

**返 回 值** 如果指定了无效的驱动程序,则本函数返回负的图形错误代码;否则,返回驱动程序注册号。如果注册一个用户提供的驱动程序,必须把 registerbgidriver 的结果传给 initgraph 作为驱动程序号的参数。

**可移植性** 仅适用于 Turbo C,仅适用于配置图形适配器的 IBM PC 及其兼容机。

**参 见** graphresult, initgraph, installuserdriver, registerbgifont

**示 例**

```

#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
int main(void)
{
 /* request auto detection */

```

```

int gdriver = DETECT, gmode, errorcode;
/* register a driver that was added into graphics.lib */
errorcode = registerbgidriver(EGAVGA_driver);
/* report any registration errors */
if (errorcode < 0)
{
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
}
/* initialize graphics and local variables */
initgraph(&gdriver, &gmode, "");
/* read result of initialization */
errorcode = graphresult();
if (errorcode != grOk) /* an error occurred */
{
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
}

/* draw a line */
line(0, 0, getmaxx(), getmaxy());
/* clean up */
getch();
closegraph();
return 0;
}

```

## ■ registerbgifont 注册已连接进来的矢量字体代码 ■

**用 法** #include <graphics.h>  
int registerbgifont(void (\*font))(void);

**原 型 在** graphics.h

**说 明** 调用 registerbgifont 使图形系统连接时包含由 font 所指的字体。本子程序检查指定的字体的连接代码。如果代码有效,就把它注册到内部表中。在本书的附录中有关于 BGIOBJ 连入字体的详细讨论。

通过在对 registerbgifont 的调用中使用连接字体的名,用户可以使用编译程序和连接程序用该公用名来连接目标文件。

如果注册一个用户提供的字体,必须把 registerbgifont 的结果传给 settextstyle 作为字体号参数。

registerfarbgi 用来注册远程字体文件,远程字体文件用 BGIOBJ 实用程序/F 开

关创建,详细用法在本书附录部分介绍。

**返回值** 如果指定了无效的字体,本函数返回负的图形错误码;否则,返回注册字体的字体号。

**可移植性** 仅适用于 Turbo C 及配有图形适配器的 IBM PC 及其兼容机。

**参 见** graphresult, initgraph, installuserdriver, registerbgidriver, settextstyle

**示 例**

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int midx, midy;

 /* register a font file that was added into graphics.lib */
 errorcode = registerbgifont(triplex_font);
 /* report any registration errors */
 if (errorcode < 0)
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
 }

 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
 }

 midx = getmaxx() / 2;
 midy = getmaxy() / 2;
 /* select the registered font */
 settextstyle(TRIPLEX_FONT, HORIZ_DIR, 4);
 /* output some text */
 settextjustify(CENTER_TEXT, CENTER_TEXT);
 outtextxy(midx, midy, "The TRIPLEX FONT");
}
```

```
 /* clean up */
 getch();
 closegraph();
 return 0;
}
```

## ■ remove 删除一个文件

用 法 #include <stdio.h>

int remove(const char \* filename);

原 型 在 stdio.h

说 明 remove 删掉由 filename 所指定的文件,它是通过调用宏 unlink 实现的。若文件已打开,则要先关闭该文件再删除。

注意: \* filename 所指的串可以包含完整的 DOS 路径名。

返 回 值 当调用成功时,函数返回 0,否则,返回-1,并置全局变量 errno 为下述值之一:

|        |         |
|--------|---------|
| ENOENT | 无此文件或目录 |
| EACCES | 权限不对    |

可移植性 仅适用于 UNIX 系统,在 ANSI C 中有定义。

参 见 unlink.

示 例 #include <stdio.h>

```
int main(void)
{
 char file[80];
 /* prompt for file name to delete */
 printf("File to delete: ");
 gets(file);
 /* delete the file */
 if (remove(file) == 0)
 printf("Removed %s.\n", file);
 else
 perror("remove");
 return 0;
}
```

## ■ rename 文件改名

用 法 #include <stdio.h>

int rename(const char \* oldname, const char \* newname);

原 型 在 stdio.h

说 明 rename 把一个文件的名称由 oldname 改为 newname。如果 newname 中包含了一个驱动器指示符,它必须与 oldname 中给出的指示符相同。

oldname 和 newname 中的目录可以不同,因此 rename 可以用来把文件从一个目录移到另一个目录。不允许使用通配符。

**返回值** 当调用成功,则 rename 返回 0;否则,返回 -1,并置 errno 为下述值之一:

|         |         |
|---------|---------|
| ENOENT  | 无此文件或目录 |
| EACCES  | 权限不对    |
| ENOTSAM | 不在同一设备上 |

**可移植性** 与 ANSI C 兼容。

**示 例** #include <stdio.h>

```
int main(void)
{
 char oldname[80], newname[80];
 /* prompt for file to rename and new name */
 printf("File to rename: ");
 gets(oldname);
 printf("New name: ");
 gets(newname);

 /* Rename the file */
 if (rename(oldname, newname) == 0)
 printf("Renamed %s to %s.\n", oldname, newname);
 else
 perror("rename");
 return 0;
}
```

## ■ restorecrtmode 恢复屏幕为调用 initgraph 前的设置 ■

**用 法** #include <graphics.h>

void far restorecrtmode(void);

**原 型** 在 graphics.h

**说 明** restorecrtmode 恢复 initgraph 检测到的原视频模式。它可以与 setgraphmode 一起使用,在文本和图形之间进行切换。但是不能使用 textmode,只有屏幕为字符模式,才能用 textmode 来改变不同的字符模式。

**返回值** 无。

**可移植性** 仅适于 Turbo C,且只能在配有图形适配器的 IBM PC 及其兼容机上使用。

**参 见** getgraphmode, initgraph, setgraphmode

**示 例** #include <graphics.h>

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

```
int main(void)
{
```

```

/* request auto detection */
int gdriver = DETECT, gmode, errorcode;
int x, y;
/* initialize graphics and local variables */
initgraph(&gdriver, &gmode, "");

/* read result of initialization */
errorcode = graphresult();
if (errorcode != grOk) /* an error occurred */
{
 printf("Graphics error, %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
}
x = getmaxx() / 2;
y = getmaxy() / 2;
/* output a message */
settextjustify(CENTER_TEXT, CENTER_TEXT);
outtextxy(x, y, "Press any key to exit graphics:");
getch();
/* restore system to text mode */
restorecrtmode();
printf("We're now in text mode. \n");
printf("Press any key to return to graphics mode:");
getch();
/* return to graphics mode */
setgraphmode(getgraphmode());
/* output a message */
settextjustify(CENTER_TEXT, CENTER_TEXT);
outtextxy(x, y, "We're back in graphics mode.");
outtextxy(x, y+textheight("W"), "Press any key to halt:");
/* clean up */
getch();
closegraph();
return 0;
}

```

## ■ rewind 将文件指针重定位于流的开始处 ■

用 法 #include <stdio.h>

void rewind(FILE \*stream);

原形在 stdio.h

**说 明** `rewind(stream)`与 `fseek(stream, 0L, SEEK_SET)`没有很大区别,只有一点是不同的,即:`rewind`清除文件结束标志和出错标志,而 `fseek`只清除文件结束标志。调用 `rewind`之后,下一个操作既可以是输入也可以是输出。

**返回值** 无。

**可移植性** 可适用于 UNIX,在 ANSI C 中有定义。

**参 见** `fopen`, `fseek`, `ftell`

**示 例** `#include <stdio.h>`  
`#include <dir.h>`

```
int main(void)
{
 FILE *fp;
 char *fname = "TXXXXXX", *newname, first;
 newname = mktemp(fname);
 fp = fopen(newname, "w+");
 fprintf(fp, "abcdefghijklmnopqrstuvwxyz");
 rewind(fp);
 fscanf(fp, "%c", &first);
 printf("The first character is, %c\n", first);
 fclose(fp);

 remove(newname);
 return 0;
}
```

## ■ rmdir 删除目录 ■

**用 法** `#include <dir.h>`

`int rmdir(const char *path);`

**原 型 在** `dir.h`

**说 明** `rmdir`删除由 `path` 给出的目录,该目录必须满足:

- 空;
- 不是当前工作目录;
- 不是根目录。

**返回值** 如果删除目录成功,则 `rmdir`返回 0;而在操作出错时返回 -1, `errno`置为下述值之一:

`EACCES` 权限不对  
`ENOENT` 路径或文件函数未找到

**可移植性** 可适用于 UNIX 系统,在 ANSI C 中有定义。

**参 见** `chdir`, `getcurdir`, `getcwd`, `mkdir`

**示 例** `#include <stdio.h>`

```
#include <conio.h>
#include <process.h>
#include <dir.h>
#define DIRNAME "testdir. $ $ $"

int main(void)
{
 int stat;
 stat = mkdir(DIRNAME);
 if (! stat)
 printf("Directory created\n");
 else
 {
 printf("Unable to create directory\n");
 exit(1);
 }
 getch();
 system("dir/p");
 getch();
 stat = rmdir(DIRNAME);
 if (! stat)
 printf("\nDirectory deleted\n");
 else
 {
 perror("\nUnable to delete directory\n");
 exit(1);
 }
 return 0;
}
```

## ■ rotl 将一个无符号整数(unsigned)左循环移位 ■

用 法 #include <stdlib.h>

unsigned \_rotl(unsigned value, int count);

原 型 在 stdlib.h

说 明 \_rotl 将值 value 向左循环移动 count 位,该数值为一个无符号整型(unsigned)。

返 回 值 \_rotl 返回 value 循环左移 count 位后的值。

可移植性 仅适用于 DOS。

参 见 \_lrotl, \_lrotr, \_rotr

示 例 /\* rotl example \*/

#include <stdlib.h>

#include <stdio.h>



```

int main(void)
{
 unsigned value, result;
 value = 32767;
 result = _rotl(value, 1);
 printf("The value %u rotated left one bit is: %u\n", value, result);
 return 0;
}

/* rotr example */
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
 unsigned value, result;
 value = 32767;
 result = _rotr(value, 1);
 printf("The value %u rotated right one bit is: %u\n", value, result);
 return 0;
}

```

## ■ rotr 将一个无符号整数向右循环移位

用 法 #include <stdlib.h>

unsigned \_rotr(unsigned value, int count);

原 型 在 stdlib.h

说 明 \_rotr 将值 value 向右循环移动 count 位, 该数值为一个无符号整型数 (unsigned)。

返 回 值 \_rotr 返回 value 循环右移 count 位后的值。

可移植性 仅适用于 DOS。

参 见 \_lrotl, \_lrotr, \_rotl

示 例 /\* rotr example \*/

```

#include <stdlib.h>
#include <stdio.h>

int main(void)
{
 unsigned value, result;
 value = 32767;
 result = _rotr(value, 1);
 printf("The value %u rotated left one bit is: %u\n", value, result);
 return 0;
}

```

```
/* rotr example */
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
 unsigned value, result;
 value = 32767;
 result = _rotr(value, 1);
 printf("The value %u rotated right one bit is: %u\n", value, result);
 return 0;
}
```

## ■ sbrk 改变数据段地址

用 法 #include <alloc.h>  
void \*sbrk(int incr);

原 型 在 alloc.h

说 明 sbrk 把截断(break)值加incr 字节,并相应改变所分配的内存。incr 可以为负,此时所分配的内存将减少。

如果这样的修改导致所分配的内存超过可允许范围,sbrk 将不对内存作任何修改;如果读者觉得难以理解,可参考后面的示例,并上机调试一下。

返 回 值 在调用成功后,sbrk 返回旧的截断(break)值;否则返回-1,并置 errno 为:  
ENOMEM 无足够内存

可移植性 适用于 UNIX 系统。

参 见 brk

示 例 #include <stdio.h>  
#include <alloc.h>

```
int main(void)
{
 printf("Changing allocation with sbrk()\n");
 printf("Before sbrk() call: %lu bytes free\n",
 (unsigned long) coreleft());
 sbrk(1000);
 printf(" After sbrk() call: %lu bytes free\n",
 (unsigned long) coreleft());
 return 0;
}
```

## ■ scanf 格式化输入

用 法 #include <stdio.h>

```
int scanf(const char *format[,address,...]);
```

原型在 `stdio.h`

说明 `scanf` 扫描 `stdin` 以接受输入数据,每次扫描一个字符。根据格式指示符(传给 `scanf` 的 `format` 所指的格式字符串),函数格式化每个字段。最后,`scanf` 将格式化的数据存入 `format` 后面的参数所指定的地址中。

### 格式字符串

在 `scanf` 与相关函数 `cscanf`、`fscanf`、`sscanf`、`vscanf`、`vfscanf` 和 `vsscanf` 中出现的格式字符串用于控制函数搜索、转换和存储输入数据的方式。对于给定的格式指示符,这些函数要求有足够的地址参数;否则结果将不可预测,甚至会导致灾难。在函数中多余的地址参数被忽略。

注意:如果所指定格式不正确,`scanf` 的执行结果将难以预料。用户应告诉 `scanf` 如何在行尾同步。在 `gets` 或 `fgets` 之后使用 `scanf` 既安全又容易,建议读者采纳。格式字符串包含三种类型的字符串:空白字符、非空白字符和格式指示符。

- 空白字符为空格、制表符(`\t`)或换行符(`\n`)。如果...`scanf` 函数在格式字符串中遇到一个空白字符,它将忽略掉输入数据中所有后续的空白字符,直到输入中出现非空白字符。
- 非空白字符是除了百分号(`%`)以外的所有其它 ASCII 字符。如果...`scanf` 函数在格式字符串中遇到一个非空白字符,它将忽略输入中的这个非空白字符。
- 格式指示符控制...`scanf` 函数读入输入字段中的字符,并转换为给定类型的值,然后把它们存在由地址参数给出的地址中。

输入数据的末尾的空格(包括换行符)被忽略,除非在格式字符串中显式地匹配。

### 格式指示符

...`scanf` 格式指示符有如下形式

```
%[*][width][F|N][h|I|L] type _character
```

每一格式指示符以百分号(`%`)开始,接着按是如下顺序:

- 可选的赋值抑制字符`[*]`。
- 可选的宽度指示符`[width]`。
- 可选的指针大小修饰符`[F|N]`。
- 可选的参数类型修饰符`[h|I|L]`。
- 字符类型。

### 可选择的格式字符串成份

以下是...`scanf` 格式字符串中可选的字符和指示符所控制的输入格式的一般形式:

| 字符或指示符             | 控制或指示                                                            |
|--------------------|------------------------------------------------------------------|
| <code>*</code>     | 控制下一个输入字段的赋值。                                                    |
| <code>width</code> | 最多可读入的字符。如果... <code>scanf</code> 函数遇到了空白字符或不可转换字符,这些函数读入的字符将减少。 |
| <code>size</code>  | 覆盖地址参数的缺省大小(N=近指针,                                               |

续上表

| 字符或指示符 | 控制或指示                                                                                                                         |
|--------|-------------------------------------------------------------------------------------------------------------------------------|
| 参数类型   | F=远指针)。<br>覆盖地址参数的缺省类型:<br>h=short int<br>l=long int (如果类型字符指示符是一整数转换)<br>I=double(如果类型字符指示符是一个浮点转换)<br>L=long double(浮点转换有效) |

**...scanf 类型字符**

下表列出了...scanf 的所有类型字符、对应要求输入的数据类型和输入数据的存放格式。在表中假定在格式指示符中没有包括可选择的字符、指示符或修饰符(\*,width 或 size)。如果想知道可选成份是怎样影响...scanf 输入的,请看下表。

| 类型字符      | 输入要求                | 参数类型                                                                               |
|-----------|---------------------|------------------------------------------------------------------------------------|
| <b>数值</b> |                     |                                                                                    |
| d         | 十进制数                | 指向 int 的指针(int * arg)                                                              |
| D         | 十进制数                | 指向 long 的指针(long * arg)                                                            |
| o         | 八进制数                | 指向 int 的指针(int * arg)                                                              |
| O         | 八进制数                | 指向 long 的指针(long * arg)                                                            |
| i         | 十、八或十               | 指向 int 的指针(int * arg)十六进制数                                                         |
| I         | 十、八或十               | 指向 long 的指针(long * arg)十六进制数                                                       |
| u         | 无符号十进制数             | 指向 unsigned int 指针(unsigned int * arg)                                             |
| U         | 无符号十进制数             | 指向 unsigned long 指针(unsigned long * arg)                                           |
| x         | 十六进制数               | 指向 int 的指针(int * arg)                                                              |
| X         | 十六进制数               | 指向 long 的指针(long * arg)                                                            |
| e,E       | 浮点数                 | 指向 float 的指针(float * arg)                                                          |
| f         | 浮点数                 | 指向 float 的指针(float * arg)                                                          |
| g,G       | 浮点数                 | 指向 float 的指针(float * arg)                                                          |
| <b>字符</b> |                     |                                                                                    |
| s         | 字符串                 | 指向字符数组的指针(char arg[])                                                              |
| c         | 字符                  | 指向字符的指针(char * arg)<br>如果 c 类型字符指示了一个字段宽度为 W (如%5c),将指向含 W 个字符的数组的指针(char arg[W])。 |
| %         | %字符                 | 不作转换,存字符%                                                                          |
| <b>指针</b> |                     |                                                                                    |
| n         |                     | 指向 int 的指针(int * arg)。成功读入的字符(到%n 为止)数被存到此 int 中。                                  |
| p         | YYYY,ZZZZ 或 ZZZZ 形式 | 指向对象的指针(far * 或 near * ),%P 转换为存储模式的指针大小                                           |

| 类型字符 | 输入要求     | 参数类型 |
|------|----------|------|
|      | 以表示十六进制数 |      |

### 输入字段

下列均为输入字段:

- 包含有两分隔符之间的所有字符(不包括空白字符)
- 当前格式下的第一个非法字符前的所有字符(如八进制格式下的 8 或 9)
- 前  $n$  个字符, 其中  $n$  为字段指示宽度

### 约定

格式指示符中的约定, 总结如下:

**%c 转换**

该指示符读下一个字符(包括空白字符)。为了跳过空白字符, 读入一个非空白字符, 可使用 `%ls`。

**%Wc 转换 (W=宽度指示):**

地址参数是一字符数组的指针, 该数组包含  $W$  个元素 (`char arg[W]`)。

**%s 转换:**

地址参数是一字符数组的指针 (`char arg[]`)。数组大小至少为  $n+1$  个字节, 其中  $n$  为字符串  $s$  的长度(按字符计算)。用空格或换行符结束输入字段, 在数组的最后一个元素后自动加上 `NULL` 终结符。

**%[search\_set] 转换:**

由方括号括起来的字符集合可用  $s$  类型字符替代。地址参数是一个指向字符数组的指针 (`char arg[]`):

方括号内包含了一个可能字符的搜索集合组成的字符串(输入字段)。如果括号内的第一个字符是 `^`, 则搜索集合为除去括号内字符的所有的其它 ASCII 字符(通常, `^` 也包含在搜索集合里, 除非在第一个 `^` 之后又出现了 `^`)。

输入字段是不由空格分界的字符串。... `scanf` 函数读相应的输入字符, 直到遇上不在搜索集合中的第一个字符为止。以下是这种转换类型的两个例子:

**%[abcd]:**

搜索输入字段中所有的  $a, b, c, d$  字符。

**%[^abdc]:**

搜索输入字段中除  $a, b, c, d$  以外的其它字符。

在搜索集合中, 可以使用“范围设置”来定义字符(数字或字母)的范围。为搜索所有的数字, 可使用:

**%[0123456789]**

也可以使用:

**%[0-9]**

搜索字母或数字, 可使用:

**%[A-Z]** 搜索所有大写字母

%[0-9A-Za-z]搜索所有数字、字母(大小写)

%[A-F T-Z]搜索从 A 到 F, T 到 Z 的大写字母。

控制这些搜索集合范围的规则是很直观的:

- 短横线(-)前的字符必须小于后面的字符
- 短横线(-)不能是最前或最后字符(否则就被认为是字符短横线,而不是范围指示符)
- 短横线(-)两边字符必须是范围的起始和终止值以下列出短横线表示为字符而不是范围指示符的一些例子:

%[-+\*/] 4个算术操作符

%[z-a] 字符'z','-', 'a'

%[+0-9-A-Z] 字符'+','-',范围 0 到 9, A 到 Z

%[+0-9A-Z-] 也是字符'+','-',范围 0 到 9, A 到 Z

%[^-0-9+A-Z] 除了'+','-',0 到 9, A 到 Z 以外的所有字符

%e、%E、%f、%g 和 %G(浮点)转换

输入字段中的浮点数具有下列一般格式

[+/-] dddddddd [. ] dddd [E|e] [+|-] ddd

其中[]内为任选项,ddd 代表十进制、八进制或十六制数字。

附加: +INF, -INF, +NAN 和 \_NAN 被识别为浮点数, INF 为无穷大, NAN 表示不是数。注意必须有符号且为大写字母。

%d、%i、%o、%x、%D、%I、%O、%X、%c、%n 转换:

在允许使用指向字符、整型、长整型指针的转换中,都可使用指向无符号字符、无符号整型或无符号整型的指针。

### 赋值抑制字符

赋值抑制字符为星号(\*),不要把它与 C 间接操作符(指针)相混淆(也为星号)。如果格式指示中有 \* 号跟在 % 后,下一输入字段被搜索但不赋给下一地址参数。被抑制的输入数据类型假定为跟在 \* 号后的类型字符所指定的类型。

不能直接决定文字匹配和赋值抑制是否成功。

### 宽度指示符

宽度指示符(n)为一个十进制整数,它控制从当前输入字段中所读入的最多字符个数。

如果输入字段的字符个数小于 n, ... scanf 函数读字段中的所有字符,然后是下一字段和格式指示符。

如果在读入给定宽度字符前遇到了空白字符或不可转换字符,已读的字符被转换、存储,接着函数处理下一格式指示符。

不可转换字符是按给定格式不能转换的字符(如八进制格式下的 8 或 9、十六进制或十进制下的 J 或 K 等)。

| 宽度指示符 | 如何影响存储输入的宽度             |
|-------|-------------------------|
| n     | 读、转换 n 个字符,并存储到当前地址参数中。 |

### 输入大小与参数类型修饰符

输入大小修饰符(N 与 F)和参数类型修饰符(h、I、L)影响...scanf 函数对相应地址参数的解释。

F 和 N 覆盖 arg 缺省或说明的大小。

h、I 和 L 指明后面的输入数据使用什么类型(h=short、I=long、L=long double)。输入数据将被转换为指定的类型,输入数据的 arg 应为指向对应大小的对象(%h 为 short、%I 为 long、%L 为 long double)。

| 修饰符 | 转换影响                                                                                                                        |
|-----|-----------------------------------------------------------------------------------------------------------------------------|
| F   | 覆盖缺省或说明的大小,arg 解释为远指针。                                                                                                      |
| N   | 覆盖缺省或说明的大小,arg 解释为近指针,不能在巨型模式中使用。                                                                                           |
| h   | 对 d,i,o,u 和 x 类型:转换输入为短整型,存在 short 对象中。<br>对 D,I,O,U 和 X 类型:无影响。<br>对 e,f,c,s,n 和 p 类型:无影响。                                 |
| I   | 对 d,i,o,u 和 x 类型:转换为长整型,存在 long 对象中。<br>对 e,f 和 g 类型:转换为双精度型,存在 double 对象中。<br>对 D,I,O,U 和 X 类型:无影响。<br>对 e,s,n 和 p 类型:无影响。 |
| L   | 对 e,f 和 g 类型:转换为长双精度型,存在 long double 对象中。<br>L 不影响其他格式。                                                                     |

### scanf 函数何时停止扫描

由于多种原因,scanf 函数在遇到正常结束符(空白字符)前,可能搜索一个特定字段或完全终止。

在下列情况下,scanf 函数搜索和存储当前字段,而进入对下一字段的处理:

- 在格式指示符的百分号后出现赋值抑制字符(\*),当前输入字段被搜索但不存储。
- 已读了 width 个字符(width 为宽度指示符,它是格式指示中的一个十进制正整数)。
- 在当前格式下不能转换下一读入字符(如在十进制格式下出现 A)。
- 输入字段中的下一个字符没有在搜索集合中出现(或在相反搜索集合中出现)。当出现以上情况,scanf 搜索当前输入字段的首字段时,下一个字符假定未读取,被当作后一输入字段的首字符,或当作后续读操作的首字符。

遇到下列情况,scanf 将停止扫描:

- 输入字段中的下一个字符与格式字符串的相应非空白字符冲突;

- 输入字段中的下一字符为 EOF;

- 格式字符串用完。

如果在格式字符串中出现一个不是格式指示部分的字符序列,它必须与输入字段中的当前字符序列匹配,scanf 函数搜索但不存储匹配字符。当出现冲突字符

时,输入数据仍在输入字段中,就象 scanf 没有读过一样。

**返回值** scanf 返回扫描、转换和存储的输入字段个数,被扫描不存储的字段不算在内。如果 scanf 试图在文件末尾读,那么返回值为 EOF。如果没有字段被存储,scanf 返回 0。

**可移植性** 适用于 UNIX 系统。

**参 见** atof, cscanf, fscanf, getc, printf, sscanf, vscanf, vscanf, vsscanf

**示 例**

```
#include <stdio.h>
#include <conio.h>
int main(void)
{
 char label[20];
 char name[20];
 int entries = 0;
 int loop, age;
 double salary;
 struct Entry_struct
 {
 char name[20];
 int age;
 float salary;
 } entry[20];
 /* Input a label as a string of characters restricting to 20 characters */
 printf("\n\nPlease enter a label for the chart: ");
 scanf("%20s", label);
 fflush(stdin); /* flush the input stream in case of bad input */
 /* Input number of entries as an integer */
 printf("How many entries will there be? (less than 20) ");
 scanf("%d", &entries);
 fflush(stdin); /* flush the input stream in case of bad input */
 /* input a name restricting input to only letters upper or lower case */
 for (loop=0; loop<entries; ++loop)
 {
 printf("Entry %d\n", loop);
 printf(" Name : ");
 scanf("%[A-Za-z]", entry[loop].name);
 fflush(stdin); /* flush the input stream in case of bad input */
 }
 /* input an age as an integer */
```



```

 printf(" Age : ");
 scanf("%d", &entry[loop].age);
 fflush(stdin); /* flush the input stream in case of bad input */
/* input a salary as a float */
 printf(" Salary : ");
 scanf("%f", &entry[loop].salary);
 fflush(stdin); /* flush the input stream in case of bad input */
}
/* Input a name, age and salary as a string, integer, and double */
printf("\nPlease enter your name, age and salary\n");
scanf("%20s %d %lf", name, &age, &salary);
/* Print out the data that was input */
printf("\n\nTable %s\n", label);
printf("Compiled by %s age %d $ %15.2lf\n", name, age, salary);
printf("-----\n");
for (loop=0; loop<entries; ++loop)
 printf("%4d | %-20s | %5d | %15.2lf\n",
 loop + 1,
 entry[loop].name,
 entry[loop].age,
 entry[loop].salary);
printf("-----\n");
return 0;
}

```

## ■ searchpath 按 DOS 路径查找一个文件 ■

**用 法** #include <dir.h>

char \* searchpath(const char \* file);

**原 型 在** dir.h

**说 明** searchpath 通过搜索 DOS 路径(即环境中的 PATH=... 串)来定位由 file 给出的文件。函数的返回值为一个指向完整路径名字字符串的指针。

函数首先检查当前驱动器下的当前目录,如果没有找到该文件,则取 PATH 环境变量,按顺序搜索路径中的每个目录,直到找到该文件或路径搜索完。

当定位文件后,返回一个包含完整路径名的字符串,该字符串可用于对该文件的其它用途(如 fopen 和 exec... 调用)。

函数把返回的字符串置于一个静态缓冲区中,下一个对 searchpath 的调用都破坏其内容。

**返 回 值** 在定位文件成功后,函数返回指向文件名字字符串的指针,而如果定位失败则返回 NULL。

**可移植性** 仅用于 DOS。

**参 见** exec..., findfirst, findnext, spawn..., system

**示 例**

```
#include <stdio.h>
#include <dir.h>
int main(void)
{
 char *p;
 /* Looks for TLINK and returns a pointer
 to the path */
 p = searchpath("TLINK.EXE");
 printf("Search for TLINK.EXE : %s\n", p);
 /* Looks for non-existent file */
 p = searchpath("NOTEXIST.FIL");
 printf("Search for NOTEXIST.FIL : %s\n", p);
 return 0;
}
```

## ■ sector 画并填充椭圆扇区 ■

**用 法** #include <graphics.h>  
void far sector(int x,int y,int stangle,int endangle,int xradius,int yradius);

**原 型 在** graphics.h

**说 明** sector 函数以(x,y)为中心,以 xradius 和 yradius 为轴,从起始角 stangle 到终止角 endangle 画并填充椭圆扇区。轮廓线以当前颜色画,扇区用 setfillstyle 和 setfillpattern 定义的模式和颜色填充。

若在填充扇区时出现错误,则 graphresult 返回-6(grNoScanMem)。

**返 回 值** 无。

**可移植性** 仅适用于 Turbo C,且只能在配备了图形适配器的 IBM PC 和兼容机上使用。

**参 见** arc, circle, ellipse, getarccoords, getaspectratio, graphresult, pieslice, setfillpattern, setfillstyle, setgraphbufsize

**示 例**

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int midx, midy, i;
 int stangle = 45, endangle = 135;
 int xrad = 100, yrad = 50;
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
```

```

errorcode = graphresult();
if (errorcode != grOk) /* an error occurred */
{
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
}
midx = getmaxx() / 2;
midy = getmaxy() / 2;
/* loop through the fill patterns */
for (i=EMPTY_FILL; i<USER_FILL; i++)
{
 /* set the fill style */
 setfillstyle(i, getmaxcolor());
 /* draw the sector slice */
 sector(midx, midy, stangle, endangle, xrad, yrad);
 getch();
}
/* clean up */
closegraph();
return 0;
}

```

## ■ segread 读段寄存器值

**用 法** #include <dos.h>

void segread(struct SREGS \*segs);

**原 型** 在 dos.h

**说 明** segread 把段寄存器的当前值放到 segs 所指的结构中。本调用可与 intdosx 和 int86x 一起使用。

**返 回 值** 无。

**可移植性** 仅用于 8086 处理器家族。

**参 见** FP\_OFF, int86, int86x, intdos, MK\_FP, movedata

**示 例** #include <stdio.h>

#include <dos.h>

int main(void)

```

{
 struct SREGS segs;
 segread(&segs);
 printf("Current segment register settings\n\n");
 printf("CS: %X DS: %X\n", segs.cs, segs.ds);
 printf("ES: %X SS: %X\n", segs.es, segs.ss);
}

```

```

 return 0;
}

```

## ■ setactivepage 设置图形输出活动页 ■

**用法** #include <graphics.h>  
void far setactivepage(int page);

**原型在** graphics.h

**说明** setactivepage 使 page 成为当前活动的图形页,其后所有的图形输出操作都在 page 图形页进行。活动图形页可以不是在屏幕上看到的页,这取决于系统是否有多个有效图形页。只有 EGA、VGA 和 Hercules 图形卡才支持多个图形页。

**返回值** 无。

**可移植性** 仅适用于 Turbo C、配备了图形适配器的 IBM PC 及其兼容机。

**参见** setvisualpage

**示例**

```

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

int main(void)
{
 /* select a driver and mode that supports */
 /* multiple pages. */
 int gdriver = EGA, gmode = EGAHI, errorcode;
 int x, y, ht;
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
 }
 x = getmaxx() / 2;
 y = getmaxy() / 2;
 ht = textheight("W");
 /* select the off screen page for drawing */
 setactivepage(1);
 /* draw a line on page #1 */
 line(0, 0, getmaxx(), getmaxy());
 /* output a message on page #1 */
}

```

```

settextjustify(CENTER_TEXT, CENTER_TEXT);
outtextxy(x, y, "This is page #1:");
outtextxy(x, y+ht, "Press any key to halt,");
/* select drawing to page #0 */
setactivepage(0);
/* output a message on page #0 */
outtextxy(x, y, "This is page #0.");
outtextxy(x, y+ht, "Press any key to view page #1:");
getch();
/* select page #1 as the visible page */
setvisualpage(1);
/* clean up */
getch();
closegraph();
return 0;
}

```

## ■ setallpalette 改变所有的调色板颜色 ■

用 法 #include <graphics.h>

void far setallpalette(struct palettetype far \*palette);

原 型 在 graphics.h

说 明 setallpalette 把调色板设置为由 palette 所指 palettetype 结构中给出的值。setallpalette 用于在 EGA/VGA 调色板中部分(或全部)改变颜色。setallpalette 所用的常量 MAXCOLORS 和结构 palettetype 在 graphics.h 中定义如下:

```

define MAXCOLORS 15
struct palettetype {
 unsigned char size;
 signed char colors[MAXCOLORS+1];
};

```

结构中的 size 是当前图形驱动程序和当前下的调色板的颜色数目。而结构中的 colors 是一个具有 size 字节的数组,它含有对应于调色板中每一个入口项实际的原始颜色编号。如果 colors 的某一元素是 -1,则那一个入口项的调色板的颜色将不被改变。

setallpalette 所使用的数组 colors 的元素可用 graphics.h 中定义的符号来表示。注意:有效颜色依赖于当前图形驱动程序和当前图形模式。调色板上的变化可以立即在屏幕上看到。每当一个调色板颜色改变时,屏幕上所有出现该颜色的象素均改为新颜色。

setallpalette 不能用于 IBM-8514 驱动程序。

返 回 值 如果参数错误,则 graphresult 返回 -11(grError),当前调色板不变。

可移植性 仅适用于 Turbo C、配有图形适配器的 IBM PC 及其兼容机。

参 见 getpalette, getpalettesize, graphresult, setbkcolor, setcolor, setpalette.  
实际颜色值

| CGA          |    | EGA/VGA          |    |
|--------------|----|------------------|----|
| 名字           | 值  | 名字               | 值  |
| BLACK        | 0  | EGA_BLACK        | 0  |
| BLUE         | 1  | EGA_BLUE         | 1  |
| GREEN        | 2  | EGA_GREEN        | 2  |
| CYAN         | 3  | EGA_CYAN         | 3  |
| RED          | 4  | EGA_RED          | 4  |
| MAGENTA      | 5  | EGA_MAGENTA      | 5  |
| BROWN        | 6  | EGA_LIGHTGRAY    | 7  |
| LIGHTGRAY    | 7  | EGA_BROWN        | 20 |
| DARKGRAY     | 8  | EGA_DARKGRAY     | 56 |
| LIGHTBLUE    | 9  | EGA_LIGHTBLUE    | 57 |
| LIGHTGREEN   | 10 | EGA_LIGHTGREEN   | 58 |
| LIGHTCYAN    | 11 | EGA_LIGHTCYAN    | 59 |
| LIGHTRED     | 12 | EGA_LIGHTRED     | 60 |
| LIGHTMAGENTA | 13 | EGA_LIGHTMAGENTA | 61 |
| YELLOW       | 14 | EGA_YELLOW       | 62 |
| WHITE        | 15 | EGA_WHITE        | 63 |

```

示 例 #include <graphics.h>
 #include <stdlib.h>
 #include <stdio.h>
 #include <conio.h>
 int main(void)
 {
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 struct palettetype pal;
 int color, maxcolor, ht;
 int y = 10;
 char msg[80];
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error, %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt,");
 getch();
 }
 }

```

```

 exit(1); /* terminate with an error code */
 }

 maxcolor = getmaxcolor();
 ht = 2 * textheight("W");
 /* grab a copy of the palette */
 getpalette(&pal);
 /* display the default palette colors */
 for (color=1; color<=maxcolor; color++)
 {
 setcolor(color);
 sprintf(msg, "Color: %d", color);
 outtextxy(1, y, msg);
 y += ht;
 }
 /* wait for a key */
 getch();
 /* black out the colors one by one */
 for (color=1; color<=maxcolor; color++)
 {
 setpalette(color, BLACK);
 getch();
 }
 /* restore the palette colors */
 setallpalette(&pal);
 /* clean up */
 getch();
 closegraph();
 return 0;
}

```

## ■ setaspectratio 设置图形纵横比 ■

**用 法** #include <graphics.h>

void far setaspectratio(int xasp,int yasp);

**原 型** 在 graphics.h

**说 明** setaspectratio 改变图形系统的缺省纵横比。图形系统是通过纵横比来保证在屏幕上画出的圆是圆形的。如果圆显示成椭圆,则说明显示器没有调准,可在硬件上调节显示器来纠正,也可在软件上借助 setaspectratio 设置纵横比来纠正。可通过调用 getaspectratio 得到本系统的当前纵横比。

**返 回 值** 无。

**可移植性** 仅适用于 Turbo C,适用于配置图形适配器的 IBM PC 及其兼容机。

**参 见** circle, getaspectratio

**示 例** #include <graphics.h>

```
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int xasp, yasp, midx, midy;
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error, %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
 }
 midx = getmaxx() / 2;
 midy = getmaxy() / 2;
 setcolor(getmaxcolor());
 /* get current aspect ratio settings */
 getaspectratio(&xasp, &yasp);
 /* draw normal circle */
 circle(midx, midy, 100);
 getch();
 /* clear the screen */
 cleardevice();
 /* adjust the aspect for a wide circle */
 setaspectratio(xasp/2, yasp);
 circle(midx, midy, 100);
 getch();
 /* adjust the aspect for a narrow circle */
 cleardevice();
 setaspectratio(xasp, yasp/2);
 circle(midx, midy, 100);
 /* clean up */
 getch();
 closegraph();
 return 0;
}
```



## ■ setbkcolor 用调色板设置当前背景颜色 ■

**用 法** #include <graphics.h>

void far setbkcolor(int color);

**原 型** 在 graphics.h

**说 明** setbkcolor 将背景设置成 color 所指定的颜色值。参数 color 可以是名字或是数字,如下表所列(符号名在 graphics.h 中定义):

| 数值 | 名字        | 数值 | 名字           |
|----|-----------|----|--------------|
| 0  | BLACK     | 8  | DARKGRAY     |
| 1  | BLUE      | 9  | LIGHTBLUE    |
| 2  | GREEN     | 10 | LIGHTGREEN   |
| 3  | CYAN      | 11 | LIGHTCYAN    |
| 4  | RED       | 12 | LIGHTRED     |
| 5  | MAGENTA   | 13 | LIGHTMAGENTA |
| 6  | BROWN     | 14 | YELLOW       |
| 7  | LIGHTGRAY | 15 | WHITE        |

例如,如果想把背景设置为蓝色,可以调用:

```
setbkcolor(BLUE)或者 setbkcolor(1)
```

在 CGA 和 EGA 系统中, setbkcolor 通过改变调色板的第一个入口点来改变颜色。

注意:当使用 EGA 和 VGA 时,若通过 setpalette 或 setallpalette 来改调色板颜色,定义的符号提供的颜色可能不正确。这是因为 setbkcolor 的参数指明当前调色板项的值而不是指明一特定颜色(如果传递参数为 0,则背景颜色置为黑)。

**返 回 值** 无。

**可移植性** 仅适用于 Turbo C,且只能在配置图形适配器的 IBM PC 及其兼容机上使用。

**参 见** getbkcolor, setallpalette, setcolor, setpalette

**示 例** #include <graphics.h>

```
#include <stdlib.h>
```

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

```
#include <conio.h>
```

```
int main(void)
```

```
{
```

```
 /* select a driver and mode that supports */
```

```
 /* multiple background colors. */
```

```
 int gdriver = EGA, gmode = EGAHI, errorcode;
```

```
 int bkcol, maxcolor, x, y;
```

```
 char msg[80];
```

```
 /* initialize graphics and local variables */
```

```
 initgraph(&gdriver, &gmode, "");
```

```
 /* read result of initialization */
```

```

errorcode = graphresult();
if (errorcode != grOk) /* an error occurred */
{
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
}
/* maximum color index supported */
maxcolor = getmaxcolor();
/* for centering text messages */
settextjustify(CENTER_TEXT, CENTER_TEXT);
x = getmaxx() / 2;
y = getmaxy() / 2;
/* loop through the available colors */
for (bkcol=0; bkcol<=maxcolor; bkcol++)
{
 /* clear the screen */
 cleardevice();
 /* select a new background color */
 setbkcolor(bkcol);
 /* output a message */
 if (bkcol == WHITE)
 setcolor(EGA_BLUE);
 sprintf(msg, "Background color: %d", bkcol);
 outtextxy(x, y, msg);
 getch();
}
/* clean up */
closegraph();
return 0;
}

```

## ■ setblock 修改已分配的内存的大小 ■

**用 法** include<dos.h>

int setblock(unsigned segx, unsigned newsize);

**原 型 在** dos.h

**说 明** setblock 修改内存的大小。segx 是以前调用 allocmem 后返回的段地址, newsize 是新的要求的内存空间大小(以节为单位计算)。

**返 回 值** 若函数成功则返回-1。若在分配出错时, 返回最大可用块大小(以节为单位计算), 并设置全局变量 doserrno。

可移植性 仅用于 DOS.

参 见 allocmem, freemem

```
示 例 #include <dos.h>
 #include <alloc.h>
 #include <stdio.h>
 #include <stdlib.h>

 int main(void)
 {
 unsigned int size, segp;
 int stat;
 size = 64; /* (64 x 16) = 1024 bytes */
 stat = allocmem(size, &segp);
 if (stat == -1)
 printf("Allocated memory at segment: %X\n", segp);
 else
 {
 printf("Failed: maximum number of paragraphs available is %d\n",
 stat);
 exit(1);
 }
 stat = setblock(seg, size * 2);
 if (stat == -1)
 printf("Expanded memory block at segment: %X\n", segp);
 else
 printf("Failed: maximum number of paragraphs available is %d\n",
 stat);
 freemem(seg);
 return 0;
 }
```

## ■ setbuf 把缓冲区与流相联 ■

用 法 #include <stdio.h>  
void setbuf(FILE \* stream, char \* buf);

原 型 在 stdio.h

说 明 setbuf 使得 I/O 缓冲用 buf 缓冲区而不是自动分配的缓冲区, 函数在流 stream 打开后使用。

若 buf 为 NULL, 在进行 I/O 时不进行缓冲; 否则, 使用由 buf 定义的缓冲区。缓冲区的长度必须为 BUFSIZ 字节长(在 stdio.h 中定义)。

如果没有重定向, stdin 和 stdout 将不使用缓冲区; 否则, 它们将使用缓冲区。setbuf 可用于改变使用的缓冲方式。

不缓冲意味着写到流中的字符将立即输出到文件或设备中; 缓冲意指字符被累

积起来作为块来写。

应在打开流 stream 或调用 fseek 之后立即调用 setbuf, 否则 setbuf 将产生不可预测的结果。在 stream 没有缓冲之时调用 setbuf 是合法的, 不会引起问题。

出现错误的原因是因为把文件的缓冲区当自动局部变量分配, 而从定义缓冲区的函数中返回时, 没有关闭相应的文件。

**返回值** 无。

**可移植性** 可用于 UNIX 系统, 在 ANSI C 中有定义。

**参见** fflush, fopen, fseek, setvbuf

**示例**

```
#include <stdio.h>

/* BUFSIZ is defined in stdio.h */
char outbuf[BUFSIZ];

int main(void)
{
 /* attach a buffer to the standard output stream */
 setbuf(stdout, outbuf);
 /* put some characters into the buffer */
 puts("This is a test of buffered output. \n\n");
 puts("This output will go into outbuf\n");
 puts("and won't appear until the buffer\n");
 puts("fills up or we flush the stream. \n");
 /* flush the output buffer */
 fflush(stdout);
 return 0;
}
```

## ■ setcbrk 设置 control—break ■

**用法** #include <dos.h>  
int setcbrk(int cbrkvalue);

**原型在** dos.h

**说明** setcbrk 使用 DOS 系统调用 0x33 设置 control—break 的检测状态为 on 或 off。  
value=0 置检测状态为 off (只在控制台、打印机或通信设备进行 I/O 时检测)。  
value=1 置检测状态为 on (每次系统调用都检测)。

**返回值** 返回 cbrkvalue 所传递的值。

**可移植性** 仅适用于 DOS 系统。

**参见** getcbrk

**示例**

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

int main(void)
{
```

```

int break_flag;
printf("Enter 0 to turn control break off\n");
printf("Enter 1 to turn control break on\n");
break_flag = getch() - 0;
setcbreak(break_flag);
if (getcbreak())
 printf("Ctrl-brk flag is on\n");
else
 printf("Ctrl-brk flag is off\n");
return 0;
}

```

## ■ setcolor 设置当前要画的线的颜色 ■

**用 法** #include <graphics.h>  
void far setcolor(int color);

**原 型** 在 graphics.h

**说 明** setcolor 置当前线条颜色为 color 值, 颜色值范围为在 0 到 getmaxcolor 之间。当前画线颜色是画线时设置的象素颜色。下表列出了分别用于 CGA 和 EGA 的画线颜色。

| 调色板 |                | 赋给颜色值(象素值)的常量    |               |
|-----|----------------|------------------|---------------|
| 号   | 1              | 2                | 3             |
| 0   | CGA_LIGHTGREEN | CGA_LIGHTRED     | CGA_YELLOW    |
| 1   | CGA_LIGHTCYAN  | CGA_LIGHTMAGENTA | CGA_WHITE     |
| 2   | CGA_GREEN      | CGA_RED          | CGA_BROWN     |
| 3   | CGA_CYAN       | CGA_MAGENTA      | CGA_LIGHTGRAY |

| 数字值 | 符号名       | 数字值 | 符号名          |
|-----|-----------|-----|--------------|
| 0   | BLACK     | 8   | DARKGRAY     |
| 1   | BLUE      | 9   | LIGHTBLUE    |
| 2   | GREEN     | 10  | LIGHTGREEN   |
| 3   | CYAN      | 11  | LIGHTCYAN    |
| 4   | RED       | 12  | LIGHTRED     |
| 5   | MAGENTA   | 13  | LIGHTMAGENTA |
| 6   | BROWN     | 14  | YELLOW       |
| 7   | LIGHTGRAY | 15  | WHITE        |

可通过传给 setcolor 颜色值或等价的符号来选择画线颜色。例如, 在 CGAC 模式 0, 调色板包括 4 种颜色: 背景颜色、浅绿、浅红、黄色。在这种模式里, setcolor (3) 和 setcolor (CGA\_YELLOW) 都设置画线颜色为黄色。

**返 回 值** 无。

可移植性 仅适用于 DOS、配置图形适配器的 IBM PC 及其兼容机。

参 见 getcolor, getmaxcolor, graphresult, setallpalette, setbkcolor, setpalette

```
示 例 #include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

int main(void)
{
 /* select a driver and mode that supports */
 /* multiple drawing colors. */
 int gdriver = EGA, gmode = EGAHI, errorcode;
 int color, maxcolor, x, y;
 char msg[80];
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error, %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
 }

 /* maximum color index supported */
 maxcolor = getmaxcolor();
 /* for centering text messages */
 settextjustify(CENTER_TEXT, CENTER_TEXT);
 x = getmaxx() / 2;
 y = getmaxy() / 2;
 /* loop through the available colors */
 for (color=1; color<=maxcolor; color++)
 {
 /* clear the screen */
 cleardevice();
 /* select a new background color */
 setcolor(color);
 /* output a message */
 sprintf(msg, "Color: %d", color);
 outtextxy(x, y, msg);
 getch();
 }
}
```

```

 /* clean up */
 closegraph();
 return 0;
}

```

## ■ setdate 设置 DOS 日期 ■

用 法 #include <dos.h>

void setdate(struct date \* datep);

原 型 在 dos.h

说 明 设置系统日期(月、日、年)为 datep 所指的结构的值。date 结构定义如下:

```

struct date {
 int da_year; /* current year */
 char da_day; /* day of th month */
 char da_mon; /* month (1=Jan) */
};

```

返 回 值 无。

可移植性 仅用于 DOS.

参 见 getdate, gettime, settime

示 例 #include <stdio.h>

#include <process.h>

#include <dos.h>

int main(void)

```

{
 struct date reset;
 struct date save_date;
 getdate(&save_date);
 printf("Original date:\n");
 system("date");
 reset.da_year = 2001;
 reset.da_day = 1;
 reset.da_mon = 1;
 setdate(&reset);
 printf("Date after setting:\n");
 system("date");
 setdate(&save_date);
 printf("Back to original date:\n");
 system("date");
 return 0;
}

```

## ■ setdisk 设置当前驱动器 ■

用 法 #include <dir.h>

```
int setdisk(int drive);
```

原型在 `dir.h`

说明 `setdisk` 设置当前盘驱动器号, 0=A、1=B、2=C 等。等价于 DOS 功能调用 0x0E)。

返回值 `setdisk` 返回可用驱动器总个数。

可移植性 仅用于 DOS。

参见 `getdisk`

示例

```
#include <stdio.h>
#include <dir.h>
int main(void)
{
 int save, disk, disks;
 /* save original drive */
 save = getdisk();
 /* print number of logic drives */
 disks = setdisk(save);
 printf("%d logical drives on the system\n\n", disks);
 /* print the drive letters available */
 printf("Available drives:\n");
 for (disk = 0; disk < 26; ++disk)
 {
 setdisk(disk);
 if (disk == getdisk())
 printf("%c, drive is available\n", disk + 'a');
 }
 setdisk(save);
 return 0;
}
```

## ■ setdta 设置磁盘传输地址

用法

```
#include <dos.h>
void setdta(char far * dta);
```

原型在 `dos.h`

说明 `setdta` 把当前 DOS 磁盘的传输地址(DTA)设置为 `dta` 所指的地址。

返回值 无。

可移植性 仅适用于 DOS。

示例

```
#include <process.h>
#include <string.h>
#include <stdio.h>
#include <dos.h>
int main(void)
```



```

{
 char line[80], far * save_dta;
 char buffer[256] = "SETDTA test!";
 struct fcb blk;
 int result;

 /* get new file name from user */
 printf("Enter a file name to create,");
 gets(line);
 /* parse the new file name to the dta */
 parsfnm(line, &blk, 1);
 printf("%d %s\n", blk.fcb_drive, blk.fcb_name);
 /* request DOS services to create file */
 if (bdosptr(0x16, &blk, 0) == -1)
 {
 perror("Error creating file");
 exit(1);
 }

 /* save old dta and set new dta */
 save_dta = getdta();
 setdta(buffer);

 /* write new records */
 blk.fcb_recsz = 256;
 blk.fcb_random = 0L;
 result = randbwr(&blk, 1);
 printf("result = %d\n", result);
 if (!result)
 printf("Write OK\n");
 else
 {
 perror("Disk error");
 exit(1);
 }

 /* request DOS services to close the file */
 if (bdosptr(0x10, &blk, 0) == -1)
 {
 perror("Error closing file");
 exit(1);
 }

 /* reset the old dta */
 setdta(save_dta);
 return 0;
}

```

## ■ setfillpattern 选择自定义的填充模式 ■

**用 法** #include <graphics.h>

void far setfillpattern(char far \* upattern,int color);

**原 型** 在 graphics.h

**说 明** 如果不是用于设置自定义的 8x8 模式而是设置预定义的模式,则 setfillpattern 和 setfillstyle 相同。

upattern 是指向 8 字节序列的指针,每个字节对应于模式里的 8 个像素。一旦某字节的某位置 1,则对应的像素被画成点。

**返 回 值** 无。

**可移植性** 仅适用于 Turbo C、配置图形适配器的 IBM PC 及其兼容机。

**参 见** getfillpattern, getfillsettings, graphresult, sector, setfillstyle

**示 例** #include <graphics.h>

#include <stdlib.h>

#include <stdio.h>

#include <conio.h>

int main(void)

{

/\* request auto detection \*/

int gdriver = DETECT, gmode, errorcode;

int maxx, maxy;

/\* a user defined fill pattern \*/

char pattern[8] = {0x00, 0x70, 0x20, 0x27, 0x24, 0x24, 0x07, 0x00};

/\* initialize graphics and local variables \*/

initgraph(&gdriver, &gmode, "");

/\* read result of initialization \*/

errorcode = graphresult();

if (errorcode != grOk) /\* an error occurred \*/

{

printf("Graphics error: %s\n", grapherrormsg(errorcode));

printf("Press any key to halt:");

getch();

exit(1); /\* terminate with an error code \*/

}

maxx = getmaxx();

maxy = getmaxy();

setcolor(getmaxcolor());

/\* select a user defined fill pattern \*/

setfillpattern(pattern, getmaxcolor());

/\* fill the screen with the pattern \*/

bar(0, 0, maxx, maxy);

```

 /* clean up */
 getch();
 closegraph();
 return 0;
}

```

## ■ setfillstyle 设置填充模式和颜色 ■

**用 法** #include <graphics.h>

void far setfillstyle(int pattern,int color);

**原 型 在** graphics.h

**说 明** setfillstyle 设置当前填充模式和颜色。如果读者想设置自定义的填充模式,应将 setfillstyle 的 pattern 值取 12 (USER\_FILL) 来使用本函数,而应该调用 setfillpattern。在 graphics.h 中定义的枚举类型 fill\_patterns 给出预定义填充模式名,并为用户定义模式提供一个标志。

| 名字              | 值  | 描述        |
|-----------------|----|-----------|
| EMPTY_FILL      | 0  | 用背景色填充    |
| SOLID_FILL      | 1  | 单色填充      |
| LINE_FILL       | 2  | 用——填充     |
| LTSLASH_FILL    | 3  | 用///填充    |
| SLASH_FILL      | 4  | 用粗线///填充  |
| BKSLASH_FILL    | 5  | 用粗线\\\填充  |
| LTBKSLASH_FILL  | 6  | 用\\\填充    |
| HATCH_FILL      | 7  | 用淡影线填充    |
| XHATCH_FILL     | 8  | 用交叉线      |
| INTERLEAVE_FILL | 9  | 用间隔线填充    |
| WIDE_DOT_FILL   | 10 | 用稀疏空白点填充  |
| CLOSE_DOT_FILL  | 11 | 用密集空白点填充  |
| USER_FILL       | 12 | 用户定义的填充 J |

除了 EMPTY\_FILL 使用当前背景颜色外,其余所有模式均使用当前填充颜色填充。

**返 回 值** 如果 setfillstyle 的参数无效,graphresult 将返回 -11 (grError),当前填充模式和填充颜色不变。

**可移植性** 仅适用于 Turbo C、配置图形适配器的 IBM PC 及其兼容机。

**参 见** bar, bar3d, fillpoly, floodfill, getfillsettings, graphresult, pieslice, sector, setfillpattern

**示 例**

```

#include <graphics.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <conio.h>

```

```

/* the names of the fill styles supported */
char *fname[] = { "EMPTY_FILL",
 "SOLID_FILL",
 "LINE_FILL",
 "LTSLASH_FILL",
 "SLASH_FILL",
 "BKSLASH_FILL",
 "LTBKSLASH_FILL",
 "HATCH_FILL",
 "XHATCH_FILL",
 "INTERLEAVE_FILL",
 "WIDE_DOT_FILL",
 "CLOSE_DOT_FILL",
 "USER_FILL"
 };

int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int style, midx, midy;
 char stylestr[40];
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
 }
 midx = getmaxx() / 2;
 midy = getmaxy() / 2;
 for (style = EMPTY_FILL; style < USER_FILL; style++)
 {
 /* select the fill style */
 setfillstyle(style, getmaxcolor());
 /* convert style into a string */
 strcpy(stylestr, fname[style]);
 /* fill a bar */
 bar3d(0, 0, midx-10, midy, 0, 0);
 /* output a message */
 }
}

```

```

 outtextxy(midx, midy, stylestr);
 /* wait for a key */
 getch();
 cleardevice();
}
/* clean up */
getch();
closegraph();
return 0;
}

```

## ■ setftime 取得文件日期和时间 ■

**用 法** #include <io.h>

int setftime(int handle, struct ftime \* ftimep);

**原 型 在** io.h

**说 明** setftime 读取与文件句柄 handle 相联的磁盘文件的时间和日期, 并将时间和日期存于由 ftimep 所指的 ftime 结构中。

ftime 的结构定义如下:

```

struct ftime{
 unsigned ft_tsec;5; /* two seconds */
 unsigned ft_min;6; /* minutes */
 unsigned ft_hour;5; /* hours */
 unsigned ft_day;5; /* days */
 unsigned ft_month;4; /* months */
 unsigned ft_year;7; /* year-1980 */
};

```

**返 回 值** 在调用函数成功时, 函数返回 0。否则返回 -1, 并置全局变量 errno 为下述值之一:

|         |       |
|---------|-------|
| EINVFNC | 无效功能号 |
| EBADF   | 无效文件号 |

**可移植性** 仅用于 DOS.

**参 见** getftime

**示 例**

```

#include <stdio.h>
#include <process.h>
#include <fcntl.h>
#include <io.h>

int main(void)
{
 struct ftime filet;
 FILE *fp;

```

```

if ((fp = fopen("TEST. $ $ $", "w")) == NULL)
{
 perror("Error:");
 exit(1);
}
fprintf(fp, "testing... \n");
/* load ftime structure with new time and date */
filet.ft_tsec = 1;
filet.ft_min = 1;
filet.ft_hour = 1;
filet.ft_day = 1;
filet.ft_month = 1;
filet.ft_year = 21;
/* show current directory for time and date */
system("dir TEST. $ $ $");
/* change the time and date stamp */
setftime(fileno(fp), &filet);
/* close and remove the temporary file */
fclose(fp);
system("dir TEST. $ $ $");
unlink("TEST. $ $ $");
return 0;
}

```

## ■ setgraphbufsize 改变内部图形缓冲区的大小 ■

**用 法** #include <graphics.h>

unsigned far setgraphbufsize(unsigned bufsize);

**原 型 在** graphics.h

**说 明** 一些图形子程序(如 floodfill)要使用调用 initgraph 时所分配的内存缓冲区,该缓冲区在调用 closegraph 时被释放。由 \_graphgetmem 所分配的这个缓冲区的缺省大小为 4096 字节。

可使缓冲区小一些(为了节省存储空间)或使它大一些(如调用 floodfill 产生错误“-7: Out of flood memory”时使用)。

setgraphbufsize 使 initgraph 调用 \_graphgetmem 时知道内部图形缓冲区分配了多少内存。

注意:调用 setgraphbufsize 必须在调用 initgraph 之前。一旦 initgraph 被调用,所有的 setgraphbufsize 调用都被忽略,直到下一次调用 closegraph 之后,调用 setgraphbufsize 才有效。

**返 回 值** 返回内部缓冲区的原来的大小值。

**可移植性** 仅适用于 Turbo C、配置图形适配器的 IBM PC 及其兼容机。

**参 见** closegraph, graphfreemem, graphgetmem, initgraph, sector

```

示 例 #include <graphics.h>
 #include <stdlib.h>
 #include <stdio.h>
 #include <conio.h>
 #define BUFSIZE 1000 /* internal graphics
 buffer size */
 int main(void)
 {
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int x, y, oldsize;
 char msg[80];
 /* set the size of the internal graphics buffer */
 /* before making a call to initgraph. */
 oldsize = setgraphbufsize(BUFSIZE);
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
 }
 x = getmaxx() / 2;
 y = getmaxy() / 2;
 /* output some messages */
 sprintf(msg, "Graphics buffer size: %d", BUFSIZE);
 settextjustify(CENTER_TEXT, CENTER_TEXT);
 outtextxy(x, y, msg);
 sprintf(msg, "Old graphics buffer size: %d", oldsize);
 outtextxy(x, y + textheight("W"), msg);
 /* clean up */
 getch();
 closegraph();
 return 0;
 }

```

### ■ setgraphmode 将系统设置成图形模式并清屏

用 法 #include <graphics.h>

```
void far setgraphmode(int mode);
```

原 型 在 graphics.h

说 明 setgraphmode 选择一个不同于调用 initgraph 时所设置的缺省的图形模式。参数 mode 对于当前设备驱动程序来说,必须是一个合法的模式。

setgraphmode 清除屏幕,将所有图形设置复位为它的缺省值(当前位置、调色板、颜色、视口等)。

可以使用 setgraphmode 和 restorecrtmode 来进行文本与图形之间的转换。

返 回 值 如果给了 setgraphmode 一个对当前驱动程序来说无效的模式,graphresult 返回 -10 (grInvalidMode)。

可移植性 仅适用于 Turbo C、配置图形适配器的 IBM PC 及其兼容机。

参 见 getgraphmode, getmoderange, graphresult, initgraph, restorecrtmode

示 例 #include <graphics.h>

```
#include <stdlib.h>
```

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

```
#include <conio.h>
```

```
int main(void)
```

```
{
```

```
 /* request auto detection */
```

```
 int gdriver = DETECT, gmode, errorcode;
```

```
 int x, y;
```

```
 /* initialize graphics and local variables */
```

```
 initgraph(&gdriver, &gmode, "");
```

```
 /* read result of initialization */
```

```
 errorcode = graphresult();
```

```
 if (errorcode != grOk) /* an error occurred */
```

```
 {
```

```
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
```

```
 printf("Press any key to halt:");
```

```
 getch();
```

```
 exit(1); /* terminate with an error code */
```

```
 }
```

```
 x = getmaxx() / 2;
```

```
 y = getmaxy() / 2;
```

```
 /* output a message */
```

```
 settextjustify(CENTER_TEXT, CENTER_TEXT);
```

```
 outtextxy(x, y, "Press any key to exit graphics.");
```

```
 getch();
```

```
 /* restore system to text mode */
```

```
 restorecrtmode();
```

```
 printf("We're now in text mode.\n");
```

```
 printf("Press any key to return to graphics mode:");
```

```
 getch();
```



```

/* return to graphics mode */
setgraphmode(getgraphmode());
/* output a message */
settextjustify(CENTER_TEXT, CENTER_TEXT);
outtextxy(x, y, "We're back in graphics mode.");
outtextxy(x, y+textheight("W"), "Press any key to halt.");
/* clean up */
getch();
closegraph();
return 0;
}

```

### ■ setjmp 非局部跳转(在 MS-WINDOWS 中不能使用本功能) ■

用 法 #include <setjmp.h>

int setjmp(jmp\_buf jmpb);

原 型 在 setjmp.h

说 明 setjmp.h 捕获所有的任务状态到参数 jmpb 中,并且返回 0。而后用此 jmpb 来调用 longjmp, longjmp 恢复所捕获的任务状态。返回时, setjmp 用值 val 返回的任务状态包括如下对象:

- 所有的段寄存器值(CS、DS、ES、SS);
- 寄存器变量(SI、DI);
- 堆栈指针(SP);
- 寄存器基指针(BP);
- 标志寄存器。

一个任务状态对于 setjmp 用于实现并发子程序是足够的。setjmp 必须在 longjmp 之前调用。调用 setjmp 和设置 jmpb 的子程序必须是活动的,而且不能在 longjmp 调用前返回,否则,结果将不可预测。

setjmp 对于处理程序的低级子程序中遇到的错误和一些异常情况是非常有用的。

不能用 setjmp 和 longjmp 来实现覆盖程序中的并发子程序。setjmp 和 longjmp 保存和恢复并发子程序所需的所有寄存器,但覆盖管理器需要跟踪栈内容并且假定只有一个栈。而当实现并发子程序时,通常有 2 个栈或一个栈有 2 个分区,因而覆盖管理程序将不能正确跟踪他们。

可以有使用自己的堆栈或部分栈的后台任务,但后台任务不能调用任何覆盖代码,且不能用 setjmp 和 longjmp 的覆盖版本来切换后台任务。若不使用覆盖代码和支持子程序,后台子程序堆栈的存在不会干扰覆盖管理程序。

返 回 值 在初次调用该函数时, setjmp 返回 0。如果返回来自对 longjmp 的调用, setjmp 返回一个非 0 值。

可移植性 适用于 UNIX 系统,在 ANSI C 中有定义。

参 见 longjmp, signal

```

示 例 #include <stdio. h>
 #include <process. h>
 #include <setjmp. h>
 void subroutine(void);
 jmp _buf jumper;
 int main(void)
 {
 int value;
 value = setjmp(jumper);
 if (value != 0)
 {
 printf("Longjmp with value %d\n", value);
 exit(value);
 }
 printf("About to call subroutine ... \n");
 subroutine();
 return 0;
 }
 void subroutine(void)
 {
 longjmp(jumper, 1);
 }

```

## ■ setlinestyle 设置当前画线宽度和类型 ■

用 法 #include <graphics. h>  
void far setlinestyle(int linestyle, unsigned upattern, int thickness);

原 型 在 graphics. h

说 明 setlinestyle 为所有的由 line、lineto、rectangle、drawpoly 等画的线设置线类型。  
linesettingstype 结构在 graphics. h 中定义如下:

```

struct linesettingstype {
 int linestyle;
 unsigned upattern;
 int thickness;
}

```

linestyle 说明以何种线型来画线(如用实线、点线、中心线、破折线)。在 graphics. h 中定义的枚举类型 line\_styles 给出了以下线型名:

| 名字          | 值 | 说明  |
|-------------|---|-----|
| SOLID_LINE  | 0 | 实线  |
| DOTTED_LINE | 1 | 点线  |
| CENTER_LINE | 2 | 中心线 |
| DASHED_LINE | 3 | 破折线 |

| 名字            | 值 | 说明     |
|---------------|---|--------|
| USERBIT _LINE | 4 | 用户定义的线 |

thickness 指明以后所画的线的宽度是正常还是粗的。

| 名字           | 值 | 说明    |
|--------------|---|-------|
| NORM _WIDTH  | 1 | 一个像素宽 |
| THICK _WIDTH | 3 | 三个像素宽 |

upattern 是一个仅当 linestyle 是 USERBIT \_LINE(4)时方起作用的 16 位模式。在这种情况下:只要字里有一位是 1,则线中的对应象素应用当前颜色画出来。例如,实线对应 upattern 值为 0xFFFF(画出所有象素),破折线对应的 upattern 值为 0x3333 或 0x0F0F。如果 setlinestyle 的 linestyle 参数不是 USERBIT \_LINE(也就是说不等于 4),则 upattern 参数仍需提供,但不起作用。

参数 linestyle 不影响圆弧、圆、椭圆、扇区,它们只使用参数 thickness。

**返回值** 如果把无效参数传给了 setlinestyle,graphresult 返回 -11,且保持当前线型不变。

**可移植性** 仅适用于 Turbo C、配置图形适配器的 IBM PC 及其兼容机。

**参 见** arc, bar3d, circle, drawpoly, ellipse, getlinesettings, graphresult, line, linerel, lineto, pieslice, rectangle

**示 例**

```
#include <graphics.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <conio.h>

/* the names of the line styles supported */
char *lname[] = {
 "SOLID _LINE",
 "DOTTED _LINE",
 "CENTER _LINE",
 "DASHED _LINE",
 "USERBIT _LINE"
};

int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int style, midx, midy, userpat;
 char stylestr[40];
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
```

```

/* read result of initialization */
errorcode = graphresult();
if (errorcode != grOk) /* an error occurred */
{
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
}

midx = getmaxx() / 2;
midy = getmaxy() / 2;
/* a user defined line pattern */
/* binary: "0000000000000001" */
userpat = 1;
for (style=SOLID_LINE; style<=USERBIT_LINE; style++)
{
 /* select the line style */
 setlinestyle(style, userpat, 1);
 /* convert style into a string */
 strcpy(stylestr, lname[style]);
 /* draw a line */
 line(0, 0, midx-10, midy);
 /* draw a rectangle */
 rectangle(0, 0, getmaxx(), getmaxy());
 /* output a message */
 outtextxy(midx, midy, stylestr);
 /* wait for a key */
 getch();
 cleardevice();
}
/* clean up */
closegraph();
return 0;
}

```

## ■ setmem 设置内存

用 法 #include <mem.h>

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

原 型 在 mem.h

说 明 setmem 将 dest 所指的长度为 length 字节的块的每个字节置为 value.

返 回 值 无。

可移植性 仅适用于 8086 序列。

**示 例** #include <stdio.h>  
#include <alloc.h>  
#include <mem.h>  
int main(void)  
{  
    char \*dest;  
    dest = calloc(21, sizeof(char));  
    setmem(dest, 20, 'c');  
    printf("%s\n", dest);  
    return 0;  
}

## ■ setmode 设置打开文件方式 ■

**用 法** #include <fcntl.h>  
int setmode(int handle, int amode);

**原 型** 在 io.h

**说 明** setmode 设置与 handle 相联的打开文件方式为二进制或文本方式。参数 amode 必须为 O\_BINARY 或 O\_TEXT, 或都不给出(这些符号常量在 fcntl.h 中有定义)。

**返 回 值** 调用成功返回 0; 否则返回 -1, 并置全局变量 error 为:  
EINVAL                  无效参数

**可移植性** 可用于 UNIX 系统。

**参 见** \_creat, creat, \_open, open

**示 例** #include <stdio.h>  
#include <fcntl.h>  
#include <io.h>  
int main(void)  
{  
    int result;  
    result = setmode(fileno(stdprn), O\_TEXT);  
    if (result == -1)  
        perror("Mode not available\n");  
    else  
        printf("Mode successfully switched\n");  
    return 0;  
}

## ■ setpalette 改变调色板的颜色 ■

**用 法** #include <graphics.h>  
void far setpalette(int colornum, int color);

**原 型** 在 graphics.h

**说 明** setpalette 将调色板的 colornum 项变为 color。例如, setpalette(0,5) 将当前调色板中的第一种颜色(背景颜色)改为实际颜色数 5。如果 size 是当前调色板的项数, 则 colornum 的取值范围为 0 到 size-1。

setpalette 可用于在 EGA/VGA 中部分(或全部)改变颜色。在 CGA 中, 只能调用 setpalette 来改变调色板的第一个项(colornum=0, 即背景颜色)。传给 setpalette 的参数 color 可由 graphics.h 定义的符号常数代替。

| CGA          |    | EGA/VGA          |    |
|--------------|----|------------------|----|
| 名字           | 值  | 名字               | 值  |
| BLACK        | 0  | EGA_BLACK        | 0  |
| BLUE         | 1  | EGA_BLUE         | 1  |
| GREEN        | 2  | EGA_GREEN        | 2  |
| CYAN         | 3  | EGA_CYAN         | 3  |
| RED          | 4  | EGA_RED          | 4  |
| MAGENTA      | 5  | EGA_MAGENTA      | 5  |
| BROWN        | 6  | EGA_LIGHTGRAY    | 7  |
| LIGHTGRAY    | 7  | EGA_BROWN        | 20 |
| DARKGRAY     | 8  | EGA_DARKGRAY     | 56 |
| LIGHTBLUE    | 9  | EGA_LIGHTBLUE    | 57 |
| LIGHTGREEN   | 10 | EGA_LIGHTGREEN   | 58 |
| LIGHTCYAN    | 11 | EGA_LIGHTCYAN    | 59 |
| LIGHTRED     | 12 | EGA_LIGHTRED     | 60 |
| LIGHTMAGENTA | 13 | EGA_LIGHTMAGENTA | 61 |
| YELLOW       | 14 | EGA_YELLOW       | 62 |
| WHITE        | 15 | EGA_WHITE        | 63 |

**注意:** 有效颜色依赖于图形驱动程序和当前图形模式。

调色板上的变化将影响当前屏幕。每当一个调色板颜色改变时, 屏幕上所有出现该颜色的像素均改为新颜色。

**注意:** 若采用 IBM-8514 驱动程序, 则不能使用 setpalette, 此时可用 setrgbpalette 代替 setpalette。

**返回值** 若无效参数传给了 setpalette, 则 graphresult 返回 -11, 当前调色板不改变。

**可移植性** 仅适用于 Turbo C, 适用于配置图形适配器的 IBM PC 及其兼容机。

**参 见** getpalette, graphresult, setallpalette, setbkcolor, setcolor, setrgbpalette

**示 例**

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
```

```

int color, maxcolor, ht;
int y = 10;
char msg[80];
/* initialize graphics and local variables */
initgraph(&gdriver, &gmode, "");
/* read result of initialization */
errorcode = graphresult();
if (errorcode != grOk) /* an error occurred */
{
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
}
maxcolor = getmaxcolor();
ht = 2 * textheight("W");
/* display the default colors */
for (color=1; color<=maxcolor; color++)
{
 setcolor(color);
 sprintf(msg, "Color, %d", color);
 outtextxy(1, y, msg);
 y += ht;
}
/* wait for a key */
getch();
/* black out the colors one by one */
for (color=1; color<=maxcolor; color++)
{
 setpalette(color, BLACK);
 getch();
}
/* clean up */
closegraph();
return 0;
}

```

## ■ setrgbpalette 定义 IBM 8514 图形卡的颜色 ■

**用 法** #include <graphics.h>

void far setrgbpalette(int colornum, int red, int green, int blue);

**原 型 在** graphics.h

**说 明** setrgbpalette 可用于 IBM 8514 和 VGA 驱动程序。colornum 定义待加载的调色

板项, red、green、blue 定义调色板项的颜色分量。

对 IBM 8514 显示器(或 256K 彩色模式的 VGA)来说, colnum 取值为 0~255, 对 VGA 的保留模式来说, colnum 可取值为 0~15。只有 red、green 和 blue 的低位字节有用, 对每个字节来说, 只有低 6 位装入调色板。

为了与 IBM 其它图形适配器兼容, BGI 驱动程序将 IBM 8514 的前 16 个调色板项定义为 EGA/VGA 的缺省颜色值, 既可以使用这些颜色, 也可以用 setrgbpalette 来修改这些颜色。

返回值 无。

可移植性 仅适用于 Turbo C、配置图形适配器的 IBM PC 及其兼容机。

参 见 setpalette

示 例

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

int main(void)
{
 /* select a driver and mode that supports the use */
 /* of the setrgbpalette function. */
 int gdriver = VGA, gmode = VGAHI, errorcode;
 struct palettetype pal;
 int i, ht, y, xmax;
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt.");
 getch();
 exit(1); /* terminate with an error code */
 }
 /* grab a copy of the palette */
 getpalette(&pal);
 /* create gray scale */
 for (i=0; i<pal.size; i++)
 setrgbpalette(pal.colors[i], i*4, i*4, i*4);
 /* display the gray scale */
 ht = getmaxy() / 16;
 xmax = getmaxx();
 y = 0;
 for (i=0; i<pal.size; i++)
```



```

{
 setfillstyle(SOLID_FILL, i);
 bar(0, y, xmax, y+ht);
 y += ht;
}
/* clean up */
getch();
closegraph();
return 0;
}

```

## ■ settextjustify 为图形函数设置文本的对齐方式 ■

**用 法** #include <graphics.h>

void far settexjustify(int horiz,int vert);

**原 型 在** graphics.h

**说 明** 调用 settextjustify 函数之后的文本输出在水平和垂直方向将按照指定与当前位置(CP)对齐。缺省的对齐是 LEFT\_TEXT(水平的)和 TOP\_TEXT(垂直的)。graphics.h 中的枚举类型 test\_just 提供传递给 horiz 和 vert 的符号名。

如果 horiz=LEFT\_TEXT 且 directon=HORIZ\_DIR,则 CP(当前位置)的 x 分量在调用 outtext (string)之后将向前移动 testwidth(string)。

settextjustify 影响 outtext 产生的文本,不能用于文本和流函数。

**返 回 值** 如果 settextjustify 的参数无效,则 graphresult 返回 -11,且当前文本对齐模式保持不变。

**可移植性** 仅适用于 Turbo C、配置了图形适配器的 IBM PC 及其兼容机。

| 描述 | 名字          | 值 | 动作   |
|----|-------------|---|------|
| 水平 | LEFT_TEXT   | 0 | 左对齐  |
|    | CENTER_TEXT | 1 | 中间对齐 |
|    | RIGHT_TEXT  | 2 | 右对齐  |
| 垂直 | BOTTOM_TEXT | 0 | 下对齐  |
|    | CENTER_TEXT | 1 | 中间对齐 |
|    | TOP_TEXT    | 2 | 上对齐  |

**参 见** gettextsettings, graphresult, outtext, settextstyle

**示 例** #include <graphics.h>

#include <stdlib.h>

#include <stdio.h>

#include <conio.h>

/\* function prototype \*/

void xat(int x, int y);

/\* horizontal text justification settings \*/

char \*hjust[] = { "LEFT TEXT",

```

 "CENTER_TEXT",
 "RIGHT_TEXT"
 };

 /* vertical text justification settings */
 char *vjust[] = { "LEFT_TEXT",
 "CENTER_TEXT",
 "RIGHT_TEXT"
 };

 int main(void)
 {
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int midx, midy, hj, vj;
 char msg[80];
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
 }

 midx = getmaxx() / 2;
 midy = getmaxy() / 2;
 /* loop through text justifications */
 for (hj=LEFT_TEXT, hj<=RIGHT_TEXT, hj++)
 for (vj=LEFT_TEXT, vj<=RIGHT_TEXT, vj++)
 {
 cleardevice();
 /* set the text justification */
 settextjustify(hj, vj);
 /* create a message string */
 sprintf(msg, "%s %s", hjust[hj], vjust[vj]);
 /* create cross hairs on the screen */
 xat(midx, midy);
 /* output the message */
 outtextxy(midx, midy, msg);
 getch();
 }

 /* clean up */
 }

```

```

 closegraph();
 return 0;
}

/* draw an "x" at (x, y) */
void xat(int x, int y)
{
 line(x-4, y, x+4, y);
 line(x, y-4, x, y+4);
}

```

## ■ settextstyle 为图形输出设置当前的文本属性 ■

**用 法** include <graphics.h>

void far settextstyle(int font,int direction,int charsize);

**原 型 在** graphics.h

**说 明** settextstyle 可设置文本字体、文本的显示方向和字符的大小。调用 settextstyle 将影响所有由 outtext 和 outtextxy 所产生的字符输出。

传给 settextstyle 的参数 font、direction 和 charsize 描述如下：

**font**：可以使用一个 8×8 的位图字体和几个“矢量”字体。缺省为 8×8 位图字体。在 graphics.h 中定义的枚举类型 font\_names 为不同的字体说明提供了符号名：

| 名字              | 值 | 说明       |
|-----------------|---|----------|
| DEFAULT_FONT    | 0 | 8×8 位图字体 |
| TRIPLEX_FONT    | 1 | 三重矢量字体   |
| SMALL_FONT      | 2 | 小号矢量字体   |
| SANS_SERIF_FONT | 3 | 无衬线矢量字体  |
| GOTHIC_FONT     | 4 | 哥特矢量字体   |

缺省的位图字体创建在图形系统中，矢量字体则存放在 \*.CHR 磁盘文件上，且每次只有一个字体文件被放在内存中。这样，当用户选择一种矢量字体时（与上次选择的矢量字体不同），必须从磁盘装入相应的 \*.CHR 文件。

为了在使用几个矢量字体时避免多次装入，可以将字体文件连接到程序中。为此，可以用 BGIOBJ 应用程序将它们转换为目标文件，然后通过 registerbgifont 注册它们，这些在源盘里的 UTIL.DOC 中有描述。

**directon**：所支持的字体方向是水平文本（从左到右）和垂直文本（逆时针旋转 90 度）。缺省方向是 HORIZ\_DIR。

| 名字        | 值 | 说明   |
|-----------|---|------|
| HORIZ_DIR | 0 | 从左到右 |
| VERT_DIR  | 1 | 从底向上 |

**charsize**：利用 charsize 因子可以将每个字符的大小放大，如果 charsize 非 0，则它

将影响位图或矢量字符;如果 charsize 为 0,则只影响矢量字符。

- 如果 charsize=1, outtext 和 outtextxy 将把 8×8 位图字体中的字符在屏幕上显示为 8×8 的像素矩阵。
- 如果 charsize=2, 这些输出函数将 8×8 位图字体中的字符显示为 16×16 的像素矩阵, 依次类推(最大可放大到普通大小的 10 倍)。
- 当 charsize=0 时, 输出函数 outtext 和 outtextxy 用缺省的字符放大因子(4)或者用 setusercharsize 给出的用户定义的字符大小来放大矢量字体。

要用 textheight 和 textwidth 来确定字符的实际大小。

返回值 无。

可移植性 仅用于 Turbo C、配置了图形适配器的 IBM PC 及其兼容机。

参见 gettextsettings, graphresult, installuserfont, settextjustily, setusercharsize, textheight, textwidth

示 例

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

/* the names of the text styles supported */
char * fname[] = { "DEFAULT font",
 "TRIPLEX font",
 "SMALL font",
 "SANS SERIF font",
 "GOTHIC font"
 };

int main(void)
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int style, midx, midy;
 int size = 1;
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
 }

 midx = getmaxx() / 2;
```

```

midy = getmaxy() / 2;
settextjustify(CENTER_TEXT, CENTER_TEXT);
/* loop through the available text styles */
for (style=DEFAULT_FONT; style<=GOTHIC_FONT; style++)
{
 cleardevice();
 if (style == TRIPLEX_FONT)
 size = 4;
 /* select the text style */
 settextstyle(style, HORIZ_DIR, size);
 /* output a message */
 outtextxy(midx, midy, fname[style]);
 getch();
}
/* clean up */
closegraph();
return 0;
}

```

## ■ settime 设置系统时间 ■

**用 法** include<dos.h>

void settime(struct time \*timep);

**原 型 在** dos.h

**说 明** settime 把系统时间填入由 timep 所指的 time 类型的结构中。time 结构定义如下：

```

struct time{
 unsigned char ti_min; /* minutes */
 unsigned char ti_hour; /* hours */
 unsigned char ti_hund; /* hundredths seconds */
 unsigned char ti_sec; /* seconds */
};

```

**返 回 值** 无。

**可移植性** 仅用于 DOS。

**参 见** ctime, getdate, gettime, setdate, time

**示 例** #include <stdio.h>

#include <dos.h>

int main(void)

{

struct time t;

gettime(&t);

printf("The current minute is, %d\n", t.ti\_min);

```

printf("The current hour is: %d\n", t.ti_hour);
printf("The current hundredth of a second is: %d\n", t.ti_hund);
printf("The current second is: %d\n", t.ti_sec);
/* Add one to the minutes struct element and then call settime */
t.ti_min++;
settime(&t);
return 0;
}

```

## ■ setusercharsize 修改矢量字体字母的宽度和高度 ■

用 法 include <graphics.h>

void far setusercharsize(int multx,int divx,int multy,int divy);

原 型 在 graphics.h

说 明 setusercharsize 可以很好地控制图形函数所使用的矢量字体文本大小。

只有当先前调用 settextstyle 设置 charsize=0 时,由 setusercharsize 设置的值才有效。

使用 setusercharsize,用户可指定宽度和高度比例因子,缺省的宽度比例由 multx,divx 给定,缺省的高度比例由 multy,divy 给定。例如,若想使文本宽度为缺省值的 2 倍,高度比缺省值高 50%,则可置:

```

multx=2; divx=1;
multy=3; divy=2;

```

返 回 值 无。

可移植性 仅用于 Turbo C、配置了图形适配器的 IBM PC 及其兼容机。

参 见 gettextsettings,graphresult,settextstyle

示 例 #include <graphics.h>

```
#include <stdlib.h>
```

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

```
#include <conio.h>
```

```
int main(void)
```

```
{
```

```
 /* request autodetection */
```

```
 int gdriver = DETECT, gmode, errorcode;
```

```
 /* initialize graphics and local variables */
```

```
 initgraph(&gdriver, &gmode, "");
```

```
 /* read result of initialization */
```

```
 errorcode = graphresult();
```

```
 if (errorcode != grOk) /* an error occurred */
```

```
 {
```

```
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
```

```
 printf("Press any key to halt.");
```

```
 getch();
```

```

 exit(1); /* terminate with an error code */
}
/* select a text style */
settextstyle(TRIPLEX_FONT, HORIZ_DIR, 4);
/* move to the text starting position */
moveto(0, getmaxy() / 2);
/* output some normal text */
outtext("Norm ");
/* make the text 1/3 the normal width */
setusercharsize(1, 3, 1, 1);
outtext("Short ");
/* make the text 3 times normal width */
setusercharsize(3, 1, 1, 1);
outtext("Wide");
/* clean up */
getch();
closegraph();
return 0;
}

```

## ■ setvbuf 使缓冲区与流相联 ■

用 法 include<stdio.h>

int setvbuf(FILE \*stream, char \*buf, int type, size\_t size);

原 型 在 stdio.h

说 明 setvbuf 使得 I/O 缓冲使用 buf 缓冲区而不进行自动分配, 它们在流 stream 打开后使用。

如果 buf 为 NULL, 将使用 malloc 分配缓冲区, 其大小为 size。关闭时缓冲区将被自动释放。参数 size 指明缓冲区大小, 它必须大于 0。

注意: 参数 size 的最大值限制为 32767。

如果没有重新定向, stdin 和 stdout 将不被缓冲, 否则它们将全部缓冲。不缓冲意味着写入流中的字符将立即输出到文件或设备中; 缓冲意味着字符被积累起来作为块写入文件或设备中。

参数 type 为下列值之一:

- \_IOFBF      文件全部缓冲。当缓冲区为空时, 下一输入操作将试图填满整个缓冲区。对于输出, 在写数据到文件之前, 先填满整个缓冲区。
- \_IOLBF      文件行缓冲。当缓冲区为空时, 下一输入操作将试图填满整个缓冲区。对于输出, 当把换行符写到文件中时, 缓冲区被清除。
- \_IONBF      文件不缓冲, 参数 buf 和 size 被忽略, 每一输入操作直接读文件, 每一输出操作直接写数据到文件中。

出现错误的通常原因是把缓冲区当作自动(局部)变量进行分配, 而从定义缓冲区的函数中返回时, 未能关闭文件。

**返回值** 调用成功时返回 0。如果给定的参数 `type` 和 `size` 为无效值,或者无足够内存空间分配一缓冲区,将返回非 0 值。

**可移植性** 适用于 UNIX 系统,ANSI C 中有定义。

**参见** `fflush`, `fopen`, `setbuf`

**示例** `#include <stdio.h>`

```
int main(void)
{
 FILE *input, *output;
 char bufr[512];
 input = fopen("file.in", "r+b");
 output = fopen("file.out", "w");
 /* set up input stream for minimal disk access,
 using our own character buffer */
 if (setvbuf(input, bufr, _IOFBF, 512) != 0)
 printf("failed to set up buffer for input file\n");
 else
 printf("buffer set up for input file\n");
 /* set up output stream for line buffering using space that
 will be obtained through an indirect call to malloc */
 if (setvbuf(output, NULL, _IOLBF, 132) != 0)
 printf("failed to set up buffer for output file\n");
 else
 printf("buffer set up for output file\n");
 /* perform file I/O here */
 /* close files */
 fclose(input);
 fclose(output);
 return 0;
}
```

## ■ `setvect` 设置中断矢量入口

**用法** `include<dos.h>`

```
void setvect(int interruptno, void interrupt (* isr)());
```

**原型在** `dos.h`

**说明** 8086 序列中的每一处理器都包含 0~255 中的一系列中断向量,每个向量值实际上是中断处理函数的入口地址。

`setvect` 把名为 `interruptno` 的中断向量值设置为新值 `isr`,后者是包含中断函数地址的远指针。如果要将 C 子程序定义为中断子程序,则只能将该子程序的地址传送给 `isr`。

注意:如果使用了 `dos.h` 中所定义的原型,则可简单地把中断函数的入口地址传送给任何存储模式下的 `setvect`。



返回值 无。

可移植性 仅用于 8086 处理器序列。

参见 getvect

示例 /\* \*\* NOTE;

```

 This is an interrupt service routine. You can NOT compile this
 program with Test Stack Overflow turned on and get an executable
 file which will operate correctly. */

#include <stdio.h>
#include <dos.h>
#include <conio.h>
#define INTR 0X1C /* The clock tick
interrupt */
void interrupt (* oldhandler)(void);
int count=0;
void interrupt handler(void)
{
 /* increase the global counter */
 count++;
 /* call the old routine */
 oldhandler();
}
int main(void)
{
 /* save the old interrupt vector */
 oldhandler = getvect(INTR);
 /* install the new interrupt handler */
 setvect(INTR, handler);
 /* loop until the counter exceeds 20 */
 while (count < 20)
 printf("count is %d\n",count);
 /* reset the old interrupt handler */
 setvect(INTR, oldhandler);
 return 0;
}

```

## ■ setverify 设置 DOS 中的校验标志状态 ■

用法 include<dos.h>  
void setverify(int value)

原型在 dos.h

说明 设置当前校验标志的状态为 value.

value 为 0=校验标志 off

value 为 1=校验标志 on

校验标志控制着磁盘输出。当其为 off 时,输出被校验;当其为 on 时,所有磁盘写操作都被校验,以保证数据被正确写入。

**返回值** 无。

**可移植性** 仅适用于 DOS。

**参 见** getverify

**示 例**

```
#include <stdio.h>
#include <conio.h>
#include <dos.h>
int main(void)
{
 int verify_flag;
 printf("Enter 0 to set verify flag off\n");
 printf("Enter 1 to set verify flag on\n");
 verify_flag = getch() - 0;
 setverify(verify_flag);
 if (getverify())
 printf("DOS verify flag is on\n");
 else
 printf("DOS verify flag is off\n");
 return 0;
}
```

## ■ setviewport 为图形输出设置当前视口 ■

**用 法** include<graphics.h>  
void far setviewport(int left,int top,int right,int bottom,in clip);

**原 型 在** graphics.h

**说 明** setviewport 为图形输出创建一个新的视口。

视口的两对角用绝对屏幕坐标(left,top)和(right,bottom)给出。当前位置(CP)被移到视口的(0,0)处。参数 clip 决定画线是否在当前视口的边界处被剪切掉(截断)。如果 clip 为非 0 值,则在当前视口边缘区的所有图形均被剪切掉。

**返回值** 如果 setviewport 的参数无效,则 graphresult 返回-11,且当前的视口设置不变。

**可移植性** 仅适用于 Turbo C,适用于配置了图形适配器的 IBM PC 及其兼容机。

**参 见** clearviewport,getviewsettings,graphresult

**示 例**

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#define CLIP_ON 1 /* activates clipping in
viewport */
int main(void)
{
```

```

/* request auto detection */
int gdriver = DETECT, gmode, errorcode;
/* initialize graphics and local variables */
initgraph(&gdriver, &gmode, "");
/* read result of initialization */
errorcode = graphresult();
if (errorcode != grOk) /* an error occurred */
{
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt,");
 getch();
 exit(1); /* terminate with an error code */
}
setcolor(getmaxcolor());
/* message in default full-screen viewport */
outtextxy(0, 0, " * <--- (0, 0) in default viewport");
/* create a smaller viewport */
setviewport(50, 50, getmaxx()-50, getmaxy()-50, CLIP_ON);
/* display some text */
outtextxy(0, 0, " * <--- (0, 0) in smaller viewport");
/* clean up */
getch();
closegraph();
return 0;
}

```

## ■ setvisualpage 设置可见的图形页号 ■

用 法 include<graphics.h>

void far setvisualpage(int page);

原 型 在 graphics.h

说 明 setvisualpage 使得 page 成为可见的图形页。

返 回 值 无。

可移植性 仅适用于 Turbo C、配置了图形适配器的 IBM PC 及其兼容机。

参 见 graphresult, setactivepage

示 例 #include <graphics.h>

#include <stdlib.h>

#include <stdio.h>

#include <conio.h>

int main(void)

{

/\* select a driver and mode that supports \*/

/\* multiple pages.

\*/

```

int gdriver = EGA, gmode = EGAHI, errorcode;
int x, y, ht;
/* initialize graphics and local variables */
initgraph(&gdriver, &gmode, "");
/* read result of initialization */
errorcode = graphresult();
if (errorcode != grOk) /* an error occurred */
{
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
}
x = getmaxx() / 2;
y = getmaxy() / 2;
ht = textheight("W");
/* select the off screen page for drawing */
setactivepage(1);
/* draw a line on page #1 */
line(0, 0, getmaxx(), getmaxy());
/* output a message on page #1 */
settextjustify(CENTER_TEXT, CENTER_TEXT);
outtextxy(x, y, "This is page #1:");
outtextxy(x, y+ht, "Press any key to halt:");
/* select drawing to page #0 */
setactivepage(0);
/* output a message on page #0 */
outtextxy(x, y, "This is page #0.");
outtextxy(x, y+ht, "Press any key to view page #1:");
getch();
/* select page #1 as the visible page */
setvisualpage(1);
/* clean up */
getch();
closegraph();
return 0;
}

```

## ■ setwritemode 设置图形方式下画线的输出模式 ■

用 法 include<graphics.h>

void far setwritemode(int mode);

原型在 graphics.h

说 明 定义常量如下:

```
COPY_PUT=0 /* MOV */
XOR_PUT=1 /* XOR */
```

每个常量均与线上字节和屏幕上相关字节之间的二进制操作相对应。

COPY\_PUT 用汇编语言指令 MOV 来覆盖屏幕上的画线。XOR\_PUT 用 XOR 指令来将直线画到屏幕上。两次成功的 XOR 指令将擦除屏幕上的画线,并且屏幕恢复成原来的样子。

setwritemode 只能与 line, linerel, lineto, rectangle 和 drawpoly 一起使用。

返回值 无。

可移植性 仅用于 Turbo C, 只能用在配置了图形适配器的 IBM PC 及其兼容机上。

参 见 drawpoly, line, linerel, lineto, putimage

示 例

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

int main()
{
 /* request auto detection */
 int gdriver = DETECT, gmode, errorcode;
 int xmax, ymax;
 /* initialize graphics and local variables */
 initgraph(&gdriver, &gmode, "");
 /* read result of initialization */
 errorcode = graphresult();
 if (errorcode != grOk) /* an error occurred */
 {
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1); /* terminate with an error code */
 }

 xmax = getmaxx();
 ymax = getmaxy();
 /* select XOR drawing mode */
 setwritemode(XOR_PUT);
 /* draw a line */
 line(0, 0, xmax, ymax);
 getch();
 /* erase the line by drawing over it */
 line(0, 0, xmax, ymax);
 getch();
 /* select overwrite drawing mode */
```

```

 setwritemode(COPY_PUT);
 /* draw a line */
 line(0, 0, xmax, ymax);
 /* clean up */
 getch();
 closegraph();
 return 0;
}

```

## ■ signal 设置某一信号的对应动作

用 法 include<signal.h>

```
void (* signal(int sig, void (* func)(int sig[, int subcode])))(int);
```

原 型 在 signal.h

说 明 signal 决定收到信号 sig 后作何处理。用户可以安装自己的处理子程序,也可以使用在 signal.h 中定义的 SIG\_DFL 和 SIG\_IGN 这两个预定义的处理子程序。

| 函数指针    | 含义     |
|---------|--------|
| SIG_DFL | 终止程序   |
| SIG_IGN | 忽略此信号  |
| SIG_ERR | 返回错误代码 |

信号类型及其缺省动作如下:

| 信号类型    | 含义                            |
|---------|-------------------------------|
| SIGABRT | 异常终止。缺省操作相当于调用 _exit(3);      |
| SIGFPE  | 除法溢出等数学错误。缺省操作相当于调用 _exit(1); |
| SIGILL  | 非法操作。缺省操作相当于调用 _exit(1);      |
| SIGINT  | CTRL-C 中断。缺省操作为执行中断 INT 23h;  |
| SIGSEGV | 非法内存存取。缺省操作相当于调用 _exit(1);    |
| SIGTERM | 要求终止程序。缺省操作相当于调用 _exit(1)。    |

sig\_atomic\_t 是在 signal.h 中定义的一个类型,它对异步中断发生时处理器自动存取的最大可存取整数类型(对 8086 序列来说,这是一个 16 位字,也即一个 C++ 整数)。

当 raise 函数或外部事件产生一个信号后,如下情况将发生:

1. 如果已为该信号安装了用户指定的处理子程序,则相应动作为 SIG\_DFL。
2. 将信号类型作为参数用以调用用户指定的处理函数。

指定的处理函数可以用返回或调用 abort, \_exit, exit, longjmp 来终止。

Turbo C 对信号类型为 SIGFPE, SIGSEGV, SIGILL 下的 ANSI C 进行了扩充。调用指定的处理子程序时可带 1 或 2 个扩展参数。如果 SIGFPE, SIGSEGV 和 SIGILL 是直接调用 raise 的结果,则调用指定的处理子程序时只带一个扩展参

数,它是一个指示处理例程直接执行的整数。SIGFPE、SIGSEGV 和 SIGILL 的活动值如下(参见 float.h 中的说明):

| SIGSEGV 信号 | 含义               |
|------------|------------------|
| SIGFPE     | FPE_EXPLICITGEN  |
| SIGSEGV    | SEGV_EXPLICITGEN |
| SIGILL     | ILL_EXPLICITGEN  |

如果 SIGFPE 是由于浮点异常产生的,则调用用户指定的处理子程序时只带一个参数,用以说明信号的 FPE\_XXX 类型。如果整数关系变形信号 SIGSEGV、SIGILL 和 SIGFPE (FPE\_INTOVFLOW 或 FPE\_INTDIV0)是因处理器异常而产生的,则调用用户处理子程序调用时带 2 个扩展参数:

1. SIGFPE、SIGSEGV 或 SIGILL 异常类型(所有类型均可参见 float.h)。第一个参数是通常的 ANSI 信号类型。
2. 一个中断处理栈里的整数指针,用以调用用户指定的处理子程序。该指针指向异常发生时所保存的处理器寄存器列表。寄存器的顺序和中断函数的参数顺序一样,即 BP,DI,SI,DS,EX,DX,CX,BX,AX,IP,CS,FLAGS。处理例程返回时改变了某寄存器的值,则也将改变该列表的某一处。例如,返回时给 SI 一个新值,如下操作

```
* ((int *)list_pointer+2)=new_SI_value;
```

用这种方法,你可以任意检查或修改寄存器的值。

下列 SIGFPE 类信号可能发生。它们与 8087 序列可检测到的异常相对应,就如同"INTEGER DIVIDE BY ZERO"和"INTERRUPT ON OVERFLOW"与主 CPU 相对应一样(如下说明在 float.h 中)。

| SIGFPE 信号       | 含义                   |
|-----------------|----------------------|
| FPE_INTOVFLOW   | 以 OF 标志设置运行 INTO     |
| FPE_INTDIV0     | 整数被 0 除              |
| FPE_INVALID     | 无效操作                 |
| FPE_ZERODIVIDE  | 除 0 操作               |
| FPE_OVERFLOW    | 数字上溢                 |
| FPE_UNDERFLOW   | 数字下溢                 |
| FPE_INEXACT     | 精密度                  |
| FPE_EXPLICITGEN | 用户程序运行 raise(SIGFPE) |

注意:FPE\_INTOVFLOW 和 FPE\_INTDIV0 信号由整数操作产生,其它信号则由浮点运算操作产生。是否产生浮点异常将取决于协处理器控制字,控制字可用\_control87 进行修改。异常情况由 Turbo C 处理,而不传给信号处理子程序。

如下 SIGSEGV 类信号可能出现:

|                  |                   |
|------------------|-------------------|
| SEGV_BOUND       | 边界强制异常            |
| SEGV_EXPLICITGEN | raise(SIGSEGV)被运行 |

8088 和 8086 处理器没有边界指令,186、286、386 和 NEC V 系列处理器有这种指令。所以,在 8088 和 8086 处理器中,SIGSEGV 信号的 SEGV\_BOUND 类型不会出现。Turbo C 不产生边界指令,不过可以用于嵌入代码和单独编译连接的汇编子程序中。

如下 SIGILL 类信号可能出现:

|                 |                  |
|-----------------|------------------|
| ILL_EXECUTION   | 非法操作             |
| ILL_EXPLICITGEN | raise(SIGILL)被运行 |

8088、8086、NEC V20 与 NEC V30 处理器没有非法操作异常,186、286、386、NEC V40 和 NEC V50 处理器有这种异常。所以在 8088、8086、NEC V20 和 NECV30 处理器上,SIGILL 的 ILL\_EXECUTION 类型不会出现。

当信号类型为 SIGFPE、SIGSEGV 或 SIGILL 时,从信号处理子程序返回一般是不可取的,因为,8087 的状态已经不正常,整数除法的结果是错误的,执行了溢出操作,边界指令失败,或者有非法操作企图。只有当处理子程序修改完寄存器后再返回才是可靠的,这样,存在一个可靠的返回现场,或信号类型清楚地指明了所产生的信号(如 FPE\_EXPLICITGEN, SEGV\_EXPLICITGEN, ILL\_EXPLICITGEN)。

在这种情况下,用户通常可以打印出错误信息,并调用 \_exit, exit 或 abort 以终止程序。如果在其它现场执行一个返回,程序恢复后将是不可预测的。

**返回值** 调用成功时,signal 返回一个指针,指向当前赋予这一指定信号类型先前的处理子程序。否则,signal 返回 SIG\_ERR,且将全程变量 errno 置为 EINVAL。

**可移植性** signal 与 ANSI C 中兼容。

**参 见** abort, \_control87, ctrlbrk, exit, longjmp, raise, setjmp

**示 例**

```

/* This example installs a signal handler routine for SIGFPE,
 catches an integer overflow condition, makes an adjustment
 to AX register, and returns. This example program MAY cause
 your computer to crash, and will produce runtime errors
 depending on which memory model is used. */
#pragma inline
#include <stdio.h>
#include <signal.h>
void Catcher(int *reglist)
{
 printf("Caught it! \n"); * (reglist + 8) = 3; /* make return AX = 3 */
}
int main(void)
{
 signal(SIGFPE, Catcher);
 asm mov ax,07FFFH /* AX = 32767 */
 asm inc ax /* cause overflow */
 asm into /* activate handler */

```



```

/* The handler set AX to 3 on return. If that hadn't happened,
 there would have been another exception when the next 'into'
 was executed after the 'dec' instruction. */
asm dec ax /* no overflow now */
asm into /* doesn't activate */
return 0;
}

```

## ■ sin 计算正弦值

- 用法** 对于实数 对于复数  
       include<math.h> include<complex.h>  
       double sin(double x); complex sin(complex x);
- 原型在** 对于实数 对于复数  
       math.h complex.h
- 说明** sin 计算参数的正弦值, 参数角度以弧度表示。该函数的错误处理程序可通过 matherr 函数加以修改。
- 返回值** sin 返回参数的正弦值。  
       复数正弦定义如下

$$\sin(z) = (\exp(i * z) - \exp(-i * z)) / (2i)$$

- 可移植性** 实数版本适用于 UNIX 系统, 在 ANSI C 中有定义。复数版本仅适用于 Turbo C, 且不可移植。

**参见** acos, asin, atan, atan2, complex, cos, tan

**示例** #include <stdio.h>  
       #include <math.h>  
       int main(void)  
       {  
           double result, x = 0.5;  
           result = sin(x);  
           printf("The sin() of %lf is %lf\n", x, result);  
           return 0;  
       }

## ■ sinh 计算双曲正弦值

- 用法** 对于实数 对于复数  
       include<math.h> include<complex.h>  
       double sinh(double x); complex sinh(complex x);
- 原型在** 对于实数 对于复数  
       math.h complex.h
- 说明** sinh 计算参数 x 的双曲正弦值  $(e^x - e^{-x})/2$ 。  
       sinh 的错误处理程序可通过 matherr 进行修改。

复数的双曲正弦值定义如下:

$$\sinh(z) = (\exp(i * z) - \exp(-z)) / 2$$

**返回值**  $\sinh$  返回  $x$  的双曲正弦值。

当正确值溢出时,  $\sinh$  返回含正确符号的 HUGE\_VAL 值; 同时, 将全局变量 `errno` 设置为 ERANGE, 参见 `cosh`。

**可移植性**  $\sinh$  的实数版本适用于 UNIX 系统, 在 ANSI C 中有定义。其复数版本仅适用于 Turbo C, 且不可移植。

**参 见** `acos`, `asin`, `atan`, `atan2`, `complex`, `cos`, `cosh`, `tan`, `tanh`

**示 例** `#include <stdio.h>`

`#include <math.h>`

`int main(void)`

{

`double result, x = 0.5;`

`result = sinh(x);`

`printf("The hyperbolic sin() of %lf is %lf\n", x, result);`

`return 0;`

}

## ■ sleep 执行挂起一段时间

**用 法** `include<dos.h>`

`void sleep(unsigned seconds);`

**原 型 在** `dos.h`

**说 明** 调用 `sleep` 时, 当前程序暂停执行, 挂起的时间(以秒为单位)由参数 `seconds` 指定。间断时间可以精确到 1/100 秒或 DOS 时钟级。

**返回值** 无。

**可移植性** 适用于 UNIX 系统。

**参 见** `delay`

**示 例** `#include <dos.h>`

`#include <stdio.h>`

`int main(void)`

{

`int i;`

`for (i=1; i<5; i++)`

    {

`printf("Sleeping for %d seconds\n", i);`

`sleep(i);`

    }

`return 0;`

}

## ■ sopen 打开一共享文件

**用 法** `include<fcntl.h>`

```
include<sys\stat.h>
include<share.h>
include<io.h>
int sopen(char *path,int access,int shflag,int mode);
```

原型在 io.h

说明 sopen 根据参数 access、shflag 和 mode 的值打开由 path 指定的文件,为共享读写操作作好准备。

sopen 是一个宏定义:

```
open(path,(access)|(shflag),mode)
```

对 sopen 来说,access 是由以下两表中的标志经过位“或”而构成的,access 只能使用表 1 中的一个标志,其余的标志可以任意组合使用。

表 1:读/写标志

|          |          |
|----------|----------|
| O_RDONLY | 以只读方式打开。 |
| O_WRONLY | 以只写方式打开。 |
| O_RDWR   | 以读写方式打开。 |

表 2:其它存取标志

|          |                                                           |
|----------|-----------------------------------------------------------|
| O_NDELAY | 未用,提供以与 UNIX 兼容。                                          |
| O_APPEND | 若置位,每次写操作前都使文件指针指向文件末尾。                                   |
| O_CREAT  | 如果文件已存在,本标志无作用,如果文件不存在,则创建文件,且用 mode 设置文件属性位,这与 chmod 一样。 |
| O_TRUNC  | 将现存文件的长度截为 0,属性不变。                                        |
| O_EXCL   | 只和 O_CREAT 一起使用,如果文件已存在,返回错误。                             |
| O_BINARY | 本标志显式指明文件以二进制方式打开。                                        |
| O_TEXT   | 本标志显式指明文件以文本方式打开。                                         |

这些 O\_... 符号常量在 fcntl.h 中定义。

如果没有给出 O\_BINARY 或 O\_TEXT,则按全局变量 fmode 所设置的传递方式打开文件。

如果使用了 O\_CREAT 来构造 access,则需要用在 sys\stat.h 中定义的下列符号常量来提供 sopen 的参数 mode。

| mode 值           | 存取权限  |
|------------------|-------|
| S_IWRITE         | 允许写   |
| S_IREAD          | 允许读   |
| S_IREAD\S_IWRITE | 允许读/写 |

shflag 指明文件 path 共享属性的类型。shflag 所使用的符号在 share.h 中有定义。

| shflag 值  | 作用      |
|-----------|---------|
| SH_COMPAT | 设置兼容模式  |
| SH_DENYRW | 禁止读/写操作 |

| shflag 值    | 作用      |
|-------------|---------|
| SH_DENYWR   | 禁止写操作   |
| SH_DENYRD   | 禁止读操作   |
| SH_DENYNONE | 允许读/写操作 |
| SH_DENYNO   | 允许读/写操作 |

**返回值** 在调用成功后, `sopen` 返回文件句柄, 且使文件指针(标明文件的当前位置)指向文件的开始处。在出错时, 返回 -1, 并置 `errno` 为下列值之一:

ENOENT 没有找到路径或文件名  
 EMFILE 打开文件太多  
 EACCES 无此存取权限  
 EINVACC 无效存取码

**可移植性** 适用于 UNIX 系统。在 UNIX 第 7 版中没有定义 `O_...` 助记符。UNIX 系统使用了除 `O_BINARY` 之外的所有 `O_...` 助记符。

**参见** `chmod`, `colse`, `creat`, `lock`, `lseek`, `_open`, `open`, `unlock`, `unmask`

**示例**

```
#include <io.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <process.h>
#include <share.h>
#include <stdio.h>

int main(void)
{
 int handle;
 int status;
 handle = sopen("c:\\autoexec.bat", O_RDONLY, SH_DENYNO, S_IREAD);
 if (!handle)
 {
 printf("sopen failed\n");
 exit(1);
 }
 status = access("c:\\autoexec.bat", 6);
 if (status == 0)
 printf("read/write access allowed\n");
 else
 printf("read/write access not allowed\n");
 close(handle);
 return 0;
}
```

## ■ sound 按指定频率打开 PC 扬声器 ■

**用法** `include<dos.h>`

```
void sound(unsigned frequency);
```

原型在 dos.h

说明 sound 以给定的频率打开 PC 扬声器。frequency 以赫兹(每秒周期数)为单位表示频率,若调用 sound 之后又希望关闭扬声器,则可以调用 nosound 函数。

可移植性 适用于 IBM PC 及其兼容机,在 Turbo Pascal 中有相应的函数。

参见 delay, nosound

示例

```
/* Emits a 440-Hz tone (A above middle C) for 1 second. */
#include <dos.h>
int main(void)
{
 sound(440);
 delay(1000);
 nosound();
 return 0;
}
```

## ■ spawnl, spawnle, spawnlp, spawnlpe, spawnv, spawnve, spawnvp, spawnvpe 创建并运行子进程 ■

用法 include <process.h>

include <stdio.h>

```
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[]);
```

原型在 process.h

说明 spawn... 系列函数可以创建并运行称为子进程的其它文件。必须有足够的内存用来加载和执行这些子进程。mode 值用以确定调用函数(父进程)在调用 spawn... 后所采取的动作:

P\_WAIT 父进程“挂起”直到子进程执行完毕。

P\_NOWAIT 父进程和子进程同时运行。

P\_OVERLAY 子进程覆盖父进程原有的存储区位置,这与 exec... 调用相同。

注意: P\_NOWAIT 当前无用,使用它会产生错误值。

path 是被调用子进程的文件名。spawn... 函数通过标准的 DOS 搜索算法来查找 path:

- 如果没有扩展名或句点: 先查找该名; 如果没有找到, 就加上 .COM 扩展名再查找, 如果还未找到, 则加上 .EXT 扩展名再查找。
- 给定扩展名: 查找给出的文件;
- 给出句点: 查找无扩展名的文件;
- 如果 path 没包含确定的目录, 带 p 后缀的 spawn... 函数将首先查找当前目录, 然后查找 DOS 环境变量 PATH 所指定的目录。

加在 spawn... 系列函数后的后缀 l, v, p, e 表示命名函数的某种操作能力:

p 表明函数还将在 DOS 的 PATH 环境变量所指明的目录中查找文件。如果没有 p 后缀, 则只在当前工作目录中查找。

l 表明将指针参数 arg0, arg1... argn 作为独立参数进行传递。通常, l 后缀用于事先知道待传递的参数个数的情况。

v 表明指针参数 arg[0], arg[1], ..., arg[n] 作为指针数组进行传递。通常, v 后缀用于将传递的参数个数可变的情况。

e 表明参数 envp 可以传到子进程, 用以改变子进程的环境。在没有 e 后缀情况下, 子进程将继承父进程的环境。

spawn... 系列函数中的每一个都必须带有一个参数说明后缀 (l 或 v), 但路径搜索和环境继承后缀 (p 或者 e) 则是可选的。

例如:

- spawnl 使用独立参数, 只在当前工作目录中寻找子文件, 父进程环境传到子进程。
- spawnvpe 接受指针数组参数, 在搜索子进程过程中使用 PATH, 并接受 envp 参数以改变子进程环境。

spawn... 函数必须至少传递一个参数给子进程 (arg0 或 arg[0]): 该参数约定为 path 的一个拷贝 (第一个参数使用不同的值不会产生错误)。

对于 DOS 3.x, path 可以为子进程所使用; 在早期版本的 DOS 中, 子进程不能使用传递过来的第 0 个参数 (arg0 或 arg[0])。

当使用 l 后缀时, arg0 通常指向 path, 而 arg1, ..., argn 指向组成新参数表的字符串, argn 后面的 NULL 表示参数表结束。

当使用 e 后缀时, 通过参数 envp 传递一新的环境参数表, 此环境参数表是一个字符指针数组, 每一元素均指向以空字符终结的字符串:

envvar=value

其中 envvar 为环境变量名, value 是由 envvar 所设置的值。envp[] 的最后元素是 NULL。当 envp 为 NULL 时表明子进程继承父进程的环境设置。

arg0+arg1+...+argn (或 arg[0]+arg[1]+...+arg[n]) 的组合长度 (包括参数间的空格在内) 必须小于 128 个字节。空终结字符不算在内。

**返回值** 在执行成功时, 返回值为子进程的退出状态 (0 为正常终结)。如果子进程调用了带非 0 参数的 exit 函数, 退出状态可设置为非 0 值。

在出现错误时,spawn...函数返回-1,且将全局变量errno置为下列之一。

|         |            |
|---------|------------|
| E2BIG   | 参数表太长      |
| EINVAL  | 无效参数       |
| ENOENT  | 路径或文件名没有找到 |
| ENOEXEC | 运行格式错      |
| ENOMEM  | 无足够内存      |

可移植性 适用于 DOS.

参 见 abort,atexit,\_exit,exec...,\_fpreset,searchpath,system

示 例 /\* spawnl() example \*/

```
#include <process.h>
#include <stdio.h>
#include <conio.h>
int main(void)
{
 int result;
 clrscr();
 result = spawnl(P_WAIT, "tcc.exe", NULL);
 if (result == -1)
 {
 perror("Error from spawnl");
 exit(1);
 }
 return 0;
}

/* spawnle() example */
#include <process.h>
#include <stdio.h>
#include <conio.h>
int main(void)
{
 int result;
 clrscr();
 result = spawnle(P_WAIT, "tcc.exe", NULL, NULL);
 if (result == -1)
 {
 perror("Error from spawnle");
 exit(1);
 }
 return 0;
}
```

## ■ sprintf 送格式输出到字符串

用 法 include<stdio.h>

```
int sprintf(char * buffer, const char format[, argument, ...]);
```

原型在 stdio.h

说明 sprintf 接受一系列参数和确定输出格式的格式控制串(由 format 指定),并把格式化的数据输出到某串中。

sprintf 把第一个格式控制符作用于第一个参数,把第二个格式控制符作用于第二个参数,依次类推。格式控制符与参数的个数必须相同。

返回值 返回输出的字节数。sprintf 不把终结空字符计算在内。如果出错,则返回 EOF。

可移植性 适用于 UNIX 系统,在 ANSI C 中有定义。与 Kernighan 和 Ritchie 的定义兼容。

参见 fprintf, printf

示例

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

int main(void)
{
 char buffer[80];
 sprintf(buffer, "An approximation of Pi is %f\n", M_PI);
 puts(buffer);
 return 0;
}
```

## ■ sqrt 计算参数平方根的绝对值 ■

用法 对于实数

```
#include <math.h>
```

```
double sqrt(double x);
```

原型在 对于实数

```
math.h
```

对于复数

```
#include <complex.h>
```

```
complex sqrt(complex x);
```

原型在 对于复数

```
complex.h
```

说明 sqrt 计算输入值平方根的绝对值。可使用 matherr 来修改 sqrt 的错误处理程序。对于复数 x, sqrt(x) 给出复数 x 的模,其 arg 值为 arg(x)/2。

复数模定义如下:

```
sqrt = sqrt(abs(z)) (cos(arg(z)/2) + isin(arg(z)/2))
```

返回值 如果成功, sqrt 返回 x 的平方根。如果 x 是实数且为正数,则结果为正数;如果 x 为实数且为负,则全局变量 errno 被设置为:

EDOM 区域错误

可移植性 sqrt 的实数版本适用于 UNIX,在 ANSI C 中有定义。sqrt 的复数版本仅适用于 Turbo C,且不可移植。

参见 complex, exp, log, pow

示例

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

int main(void)
```

```
{
 double x = 4.0, result;
```



```
result = sqrt(x);
printf("The square root of %lf is %lf\n", x, result);
return 0;
}
```

## ■ srand 初始化随机数发生器 ■

用 法 #include<stdlib.h>

void srand(unsigned seed);

原 型 在 stdlib.h

说 明 以参数 1 调用 srand, 可以重新初始化随机数发生器。以一给定的无符号整数 seed 调用 srand, 可以设置发生器的新开始点。

返 回 值 无。

可移植性 适用于 UNIX 系统, 在 ANSI C 中有定义。

参 见 rand, random, randomize

示 例 #include <stdlib.h>  
#include <stdio.h>  
#include <time.h>  
int main(void)  
{  
 int i;  
 time \_t t;  
 srand((unsigned) time(&t));  
 printf("Ten random numbers from 0 to 99\n\n");  
 for(i=0; i<10; i++)  
 printf(" %d\n", rand() % 100);  
 return 0;  
}

## ■ sscanf 从某串中扫描格式化输入 ■

用 法 #include<stdio.h>

int sscanf(const char \* buffer, const char \* format[, address, ...]);

原 型 在 stdio.h

说 明 sscanf 从某个串中搜索输入字段, 每次扫描一个字符。format 指向一个格式字符串, 它被传给 sscanf, 串中的格式指示符格式化每个输入字段。最后, sscanf 将格式化后的输入值保存到由格式字符串后的参数给出的地址中。格式指示符和地址的个数必须与输入字段的个数相同。

由于种种原因, sscanf 可能在遇到通常的字段结束符(空白字符)之前停止搜索, 或完全终止, 请参见 scanf 中对可能原因的讨论。

返 回 值 sscanf 返回成功扫描、转换、存储的输入字段个数; 返回值不包括扫描过但没有存储的字段。如果没有字段被存储, 则返回值为 0。

如果 `sscanf` 试图读一个串尾, 返回值为 EOF.

**可移植性** 适用于 UNIX 系统, 在 ANSI C 中有定义。与 Kernighan 和 Ritchie 的定义兼容。

**参 见** `fscanf`, `scanf`

**示 例**

```
#include <stdio.h>
#include <conio.h>
int main(void)
{
 char label[20];
 char name[20];
 int entries = 0;
 int loop, age;
 double salary;
 struct Entry _struct
 {
 char name[20];
 int age;
 float salary;
 } entry[20];
 /* Input a label as a string of characters restricting to 20 characters */
 printf("\n\nPlease enter a label for the chart: ");
 scanf("%20s", label);
 fflush(stdin); /* flush the input stream in case of bad input */
 /* Input number of entries as an integer */
 printf("How many entries will there be? (less than 20) ");
 scanf("%d", &entries);
 fflush(stdin); /* flush the input stream in case of bad input */
 /* input a name restricting input to only letters upper or lower case */
 for (loop=0; loop<entries; ++loop)
 {
 printf("Entry %d\n", loop);
 printf(" Name : ");
 scanf("%[A-Za-z]", entry[loop].name);
 fflush(stdin); /* flush the input stream in case of bad input */
 /* input an age as an integer */
 printf(" Age : ");
 scanf("%d", &entry[loop].age);
 fflush(stdin); /* flush the input stream in case of bad input */
 /* input a salary as a float */
 printf(" Salary : ");
 scanf("%f", &entry[loop].salary);
 fflush(stdin); /* flush the input stream in case of bad input */
 }
}
```

```

/* Input a name, age and salary as a string, integer, and double */
printf("\nPlease enter your name, age and salary\n");
scanf("%20s %d %lf", name, &age, &salary);
/* Print out the data that was input */
printf("\n\nTable %s\n", label);
printf("Compiled by %s age %d $ %15.2lf\n", name, age, salary);
printf("-----\n");
for (loop=0; loop<entries; ++loop)
 printf("%4d | %-20s | %5d | %15.2lf\n",
 loop + 1,
 entry[loop].name,
 entry[loop].age,
 entry[loop].salary);
printf("-----\n");
return 0;
}

```

## ■ stat 读取文件信息 ■

用 法 #include<sys\stat.h>

int stat(char \* path, struct stat \* statbuf);

原 型 在 sys\stat.h

说 明 stat 把一个文件或目录的信息存入 stat 结构中。

statbuf 指向一个 stat 类型的结构(在 sys\stat.h 中有定义)。该结构包括如下字段:

|          |             |
|----------|-------------|
| st_mode  | 文件模式信息的位屏蔽; |
| st_dev   | 文件的磁盘驱动器号;  |
| st_rdev  | 同 st_dev;   |
| st_nlink | 置为整型常数 1;   |
| st_size  | 文件按字节计算的大小; |
| st_atime | 文件的最近修改时间;  |
| st_mtime | 同 st_atime; |
| st_ctime | 同 st_atime。 |

stat 结构还包含这里没有提到的三个附加字段,它们的值在 DOS 下无意义。

代表文件模式信息的位屏蔽包括如下各位:

下列一位被设置:

S\_IFREG 如果 path 指向一普通文件,则置位

S\_IFDIR 如果 path 指向一目录,则置位

下列一位或两位被设置:

S\_IWRITE 如果用户有写文件的权限,置位

S\_IREAD 如果用户有读文件的权限,置位

位屏蔽还包括用户可执行位,这些是打开文件的扩展设置。位屏蔽还包括读/写位,这些是根据文件的权限方式而设置的。

**返回值** 在成功地检索了文件信息之后,stat 返回 0。否则,stat 返回 -1,并置全局变量 errno 为:

ENOENT 文件或路径没有找到

**可移植性** 适用于 UNIX 系统,在 ANSI C 中有定义。

**参见** access, chmod, fstat

**示例**

```
#include <sys/stat.h>
#include <stdio.h>
#include <time.h>
#define FILENAME "TEST. $$$"

int main(void)
{
 struct stat statbuf;
 FILE *stream;
 /* open a file for update */
 if ((stream = fopen(FILENAME, "w+")) == NULL)
 {
 fprintf(stderr, "Cannot open output file.\n");
 return(1);
 }
 /* get information about the file */
 stat(FILENAME, &statbuf);
 fclose(stream);
 /* display the information returned */
 if (statbuf.st_mode & S_IFCHR)
 printf("Handle refers to a device.\n");
 if (statbuf.st_mode & S_IFREG)
 printf("Handle refers to an ordinary file.\n");
 if (statbuf.st_mode & S_IRREAD)
 printf("User has read permission on file.\n");
 if (statbuf.st_mode & S_IWRITE)
 printf("User has write permission on file.\n");
 printf("Drive letter of file: %c\n", 'A' + statbuf.st_dev);
 printf("Size of file in bytes: %ld\n", statbuf.st_size);
 printf("Time file last opened: %s\n", ctime(&statbuf.st_ctime));
 return 0;
}
```

## status87 取浮点状态

**用法** #include <float.h>  
 unsigned int status87(void);

原型在 float.h

说明 \_status87 取浮点状态字,后者是 80×87 状态字和由 80×87 异常处理程序检测到的其它状态的组合。

返回值 返回值中的位值给出了浮点状态。有关返回位的详细定义请参阅 float.h。

可移植性 仅适用于 DOS。

```

示 例 #include <stdio.h>
 #include <float.h>
 int main(void)
 {
 float x;
 double y = 1.5e-100;
 printf("Status 87 before error: %x\n", _status87());
 x = y; /* <-- force an error to occur */
 y = x;
 printf("Status 87 after error : %x\n", _status87());
 return 0;
 }

```

## ■ stime 设置系统日期和时间 ■

用 法 #include<time.h>

```
int stime(time_t *tp)
```

原型在 time.h

说明 stime 可设置系统日期和时间,参数 tp 指向以秒为单位的、从 1970 年 1 月 1 日格林威治时间 00:00:00 算起的时间值。

返回值 stime 返回 0。

可移植性 适用于 UNIX 系统。

参 见 asctime, ftime, gettimeofday, gmtime, localtime, time, tzset

```

示 例 #include <stdio.h>
 #include <time.h>
 #include <dos.h>
 int main(void)
 {
 time_t t;
 struct tm *area;
 t = time(NULL);
 area = localtime(&t);
 printf("Number of seconds since 1/1/1970 is: %ld\n", t);
 printf("Local time is: %s", asctime(area));
 t++;
 area = localtime(&t);
 printf("Add a second: %s", asctime(area));
 }

```

```
t += 60;
area = localtime(&t);
printf("Add a minute: %s", asctime(area));
t += 3600;
area = localtime(&t);
printf("Add an hour: %s", asctime(area));
t += 86400L;
area = localtime(&t);
printf("Add a day: %s", asctime(area));
t += 2592000L;
area = localtime(&t);
printf("Add a month: %s", asctime(area));
t += 31536000L;
area = localtime(&t);
printf("Add a year: %s", asctime(area));
return 0;
}
```

## ■ stpcpy 拷贝字符串

用 法 #include <string.h>

char \*stpcpy(char \*dest, const char \*src);

原 型 在 string.h

说 明 stpcpy 拷贝 src 串到 dest 中, 在拷贝完最后一个空终结符后停止拷贝。

返 回 值 stpcpy 返回 dest+strlen(src)。

可移植性 适用于 UNIX 系统。

参 见 strcpy

示 例 #include <stdio.h>

#include <string.h>

int main(void)

{

char string[10];

char \*str1 = "abcdefghi";

stpcpy(string, str1);

printf("%s\n", string);

return 0;

}

## ■ strcat 串连接

用 法 #include <string.h>

char \*strcat(char \*dest, const char \*src);

原 型 在 string.h

**说明** strcat 把 src 的拷贝附加到 dest 的末尾, 结果串的长度为 strlen(dest) + strlen(src)。

**返回值** strcat 返回两串合并后指向新串的指针。

**可移植性** 适用于 UNIX 系统, 在 ANSI C 中有定义, 且与 Kernighan 和 Ritchie 的定义兼容。

**示例**

```
#include <string.h>
#include <stdio.h>
int main(void)
{
 char destination[25];
 char * blank = " ", * c = "C++", * turbo = "Turbo";
 strcpy(destination, turbo);
 strcat(destination, blank);
 strcat(destination, c);
 printf("%s\n", destination);
 return 0;
}
```

## ■ strchr 搜索串中某个给定字符的第一次出现

**用法** #include <string.h>  
char \* strchr(const char \* s, int c);

**原型在** string.h

**说明** strchr 正向搜索字符串, 查找某一指定字符。strchr 查找串 s 中字符 c 的第一次出现。空终结符被看作是串的一部分, 因此:

strchr(strs, 0)

返回指向串 strs 中指向终结空字符的指针。

**返回值** strchr 返回指向串 s 中第一次出现字符 c 的位置的指针; 如果 c 不在 s 中出现, strchr 返回 NULL。

**可移植性** 适用于 UNIX 系统, 在 ANSI C 中有定义。

**参见** strcspn, strrchr

**示例**

```
#include <string.h>
#include <stdio.h>
int main(void)
{
 char string[15];
 char * ptr, c = 'r';
 strcpy(string, "This is a string");
 ptr = strchr(string, c);
 if (ptr)
 printf("The character %c is at position: %d\n", c, ptr - string);
 else
```

```
printf("The character was not found\n");
return 0;
}
```

## ■ strcmp 串比较 ■

用 法 #include <string.h>

int strcmp(const char \*s1, const char \*s2);

原 型 在 string.h

说 明 strcmp 对串 s1 和 s2 进行无符号比较;从第一个字符开始,按字符顺序对两个串进行比较,直到对应字符不相同或达到串尾为止。

返 回 值 strcmp 返回值如下:

<0        如果 s1 小于 s2  
==0      如果 s1 等于 s2  
>0        如果 s1 大于 s2

可移植性 适用于 UNIX 系统,在 ANSI C 中有定义。

参 见 strcmpi, strcoll, stricmp, strncmp, strncmpi, strnicmp

示 例 #include <string.h>

#include <stdio.h>

int main(void)

```
{
 char *buf1 = "aaa", *buf2 = "bbb", *buf3 = "ccc";
 int ptr;
 ptr = strcmp(buf2, buf1);
 if (ptr > 0)
 printf("buffer 2 is greater than buffer 1\n");
 else
 printf("buffer 2 is less than buffer 1\n");
 ptr = strcmp(buf2, buf3);
 if (ptr > 0)
 printf("buffer 2 is greater than buffer 3\n");
 else
 printf("buffer 2 is less than buffer 3\n");
 return 0;
}
```

## ■ strcmpi 忽略大小写的串比较 ■

用 法 #include <string.h>

int strcmpi(const char \*s1, const char \*s2);

原 型 在 string.h

说 明 strcmpi 对串 s1 和 s2 进行无符号比较,不区分串中字符的大小写(与 stricmp 相同,只是 strcmpi 为宏)。



根据 s1(或其一部分)与 s2(或其一部分)的比较结果返回一个值(<0,0 或>0)。

函数 strcmpi 与 stricmp 相同,只是前者是通过 string.h 中的宏实现的,该宏把对 strcmpi 的调用转换为 stricmp。因此,若使用 strcmpi,必须在文件中包含 string.h 文件。该宏是为了与其它 C 编译器兼容而提供的。

**返回值** strcmpi 返回 int 值:

<0      如果 s1 小于 s2  
 ==0     如果 s1 等于 s2  
 >0      如果 s1 大于 s2

**可移植性** 仅适用于 DOS。

**参见** strcmp, strcoll, stricmp, strncmp, strncmpi, strnicmp

**示例** /\* strncmpi example \*/  
 #include <string.h>  
 #include <stdio.h>  
 int main(void)  
 {  
     char \*buf1 = "BBB", \*buf2 = "bbb";  
     int ptr;  
     ptr = strcmpi(buf2, buf1);  
     if (ptr > 0)  
         printf("buffer 2 is greater than buffer 1\n");  
     if (ptr < 0)  
         printf("buffer 2 is less than buffer 1\n");  
     if (ptr == 0)  
         printf("buffer 2 equals buffer 1\n");  
     return 0;  
 }

## ■ strcpy 串拷贝

**用法** #include<string.h>  
 char \*strcpy(char \*dest,const char \*src);

**原型在** string.h

**说明** 把串 src 拷贝到 dest 中,直到拷完终结空字符为止。

**返回值** 适用于 UNIX 系统,在 ANSI C 中有定义。

**参见** strcpy

**示例** #include <stdio.h>  
 #include <string.h>  
 int main(void)  
 {  
     char string[10];  
     char \*str1 = "abcdefghi";  
     strcpy(string, str1);

```
 printf("%s\n", string);
 return 0;
}
```

## ■ strcspn 搜索串中不包含给定字符集之子集的第一个段 ■

用 法 #include <string.h>

size\_t strcspn(const char \*s1, const char \*s2);

原 型 在 string.h

返 回 值 strcspn 返回串 s1 中初始段(该段中完全不出现 s2 中的字符)的长度。

可移植性 适用于 UNIX 系统,在 ANSI C 中有定义。

参 见 strchr, strchr

示 例

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>
int main(void)
{
 char *string1 = "1234567890";
 char *string2 = "747DC8";
 int length;
 length = strcspn(string1, string2);
 printf("Character where strings intersect is at position %d\n",
 length);
 return 0;
}
```

## ■ strdup 复制串 ■

用 法 #include <string.h>

char \*strdup(const char \*s);

原 型 在 string.h

说 明 strdup 复制串 s 到内存中,内存是通过调用 malloc 分配的,大小为(strlen(s)+1)个字节长。当不再使用的时候,应把 strdup 分配的空间释放掉。

返 回 值 strdup 返回指向包含复制串 s 存储位置的指针;在分配不到足够的内存时,返回 NULL。

可移植性 适用于 UNIX 系统。

参 见 free

示 例

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>
int main(void)
{
 char *dup str, *string = "abcde";
```

```

 dup_str = strdup(string);
 printf("%s\n", dup_str);
 free(dup_str);
 return 0;
}

```

## ■ \_strerror 建立用户定义的错误信息 ■

**用 法** #include <string.h>

char \* \_strerror(const char \* s);

**原 型** 在 string.h, stdio.h

**说 明** \_strerror 允许用户自己定义错误信息, 返回一个指向包含错误信息的、以空字符终结的字符串的指针。

- 如果 s 为 NULL, 返回指向最近生成的错误信息的指针。
- 如果 s 不为 NULL, 返回值包含 s (用户定义的信息)、一个冒号、一个空格、最近产生的系统错误信息和一个换行符。s 的长度应小于或等于 94 个字符。在 Turbo C 2.0 版中 \_strerror 和 strerror 相同。

**返 回 值** \_strerror 返回一个指向所创建的错误信息字符串的指针。错误信息字符串放在一个静态缓冲区中, 每次调用 \_strerror 时均被重写。

**可移植性** 仅适用于 DOS。

**参 见** perror, strerror

**示 例** #include <stdio.h>

```

int main(void)
{
 FILE * fp;
 /* open a file for writing */
 fp = fopen("TEST. $ $ $", "w");
 /* force an error condition by attempting to read */
 if (! fp) fgetc(fp);
 if (ferror(fp))
 /* display a custom error message */
 printf("%s", _strerror("Custom"));
 fclose(fp);
 return 0;
}

```

## ■ strerror 返回指向错误信息字符串的指针 ■

**用 法** #include <string.h>

char \* strerror(int errnum);

**原 型** 在 string.h

**说 明** strerror 接收一个 int 型参数 errnum 作为错误号, 返回指向与 errnum 相对应的错误信息字符串的指针。

**返回值** 返回一个指向所创建的错误信息串的指针, 错误信息串放在一个静态缓冲区中, 每次调用 `strerror` 时都被覆盖。

**可移植性** 与 ANSI C 兼容。

**参 见** `perror`, `_strerror`

**示 例**

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

int main(void)
{
 char * buffer;
 buffer = strerror(errno);
 printf("Error: %s\n", buffer);
 return 0;
}
```

## ■ `stricmp` 忽略大小写的串比较

**用 法** `#include <string.h>`  
`int stricmp(const char *s1, const char *s2);`

**原 型 在** `string.h`

**说 明** `stricmp` 对串 `s1` 和 `s2` 进行无符号比较。从每个串的第一个字符开始, 按字符次序进行比较, 直到相对应的字符不相同或达到串尾为止。比较时不区分大小写。返回值 (`<0`, `0` 或 `>0`) 取决于 `s1` (或 `s1` 的一部分) 和 `s2` (或 `s2` 的一部分) 的比较结果。

函数 `stricmp` 与 `strcmpi` 功能相同, 后者是 `string.h` 中定义的一个宏, 它把 `strcmpi` 调用转变为 `stricmp` 调用。因此, 若使用 `strcmpi`, 必须包含文件 `string.h`。

**返回值** `stricmp` 返回一个 `int` 型的数值。

|                    |                                      |
|--------------------|--------------------------------------|
| <code>&lt;0</code> | 若 <code>s1</code> 小于 <code>s2</code> |
| <code>==0</code>   | 若 <code>s1</code> 等于 <code>s2</code> |
| <code>&gt;0</code> | 若 <code>s1</code> 大于 <code>s2</code> |

**可移植性** 仅适用于 DOS。

**参 见** `strcmp`, `strcmpi`, `strcoll`, `strncmp`, `strncmpi`, `strnicmp`

**示 例**

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

int main(void)
{
 char * buf1 = "BBB", * buf2 = "bbb";
 int ptr;
 ptr = stricmp(buf2, buf1);
 if (ptr > 0)
 printf("buffer 2 is greater than buffer 1\n");
}
```

```
if (ptr < 0)
 printf("buffer 2 is less than buffer 1\n");
if (ptr == 0)
 printf("buffer 2 equals buffer 1\n");
return 0;
}
```

## ■ strlen 计算字符串的长度 ■

用 法 #include<string.h>

size\_t strlen(const char \*s);

原 型 在 string.h

说 明 strlen 可计算字符串 s 的长度。

返 回 值 strlen 返回 s 中包含的字符个数,但不包括空终结字符。

可移植性 适用于 UNIX 系统,在 ANSI C 中有定义。

参 见 strcat

示 例 #include <stdio.h>

#include <string.h>

int main(void)

```
{
 char *string = "Borland International";
 printf("%d\n", strlen(string));
 return 0;
}
```

## ■ strlwr 转换字符串中的大写字母为小写字母 ■

用 法 #include<string.h>

char \*strlwr(char \*S);

原 型 在 string.h

说 明 strlwr 将字符串 s 中的大写字母(A-Z)转换为对应的小写字母(a-z),其它字母不变。

返 回 值 strlwr 返回指向转换后的字符串 s 的指针。

可移植性 仅适用于 DOS。

参 见 strupr.

示 例 #include <stdio.h>

#include <string.h>

int main(void)

```
{
 char *string = "Borland International";
 printf("string prior to strlwr: %s\n", string);
 strlwr(string);
 printf("string after strlwr: %s\n", string);
}
```

```
return 0;
```

```
}
```

## ■ **strncat** 把字符串的一部分附加到另一个串之后

用 法 `#include <string.h>`

```
char *strncat(char *dest, const char *src, size_t maxlen);
```

原 型 在 `string.h`

说 明 `strncat` 把 `src` 中最多 `maxlen` 个字符填充到 `dest` 之后, 再在末尾添加一空字符。结果串的最大长度为 `strlen(dest) + maxlen`。

返 回 值 返回 `dest`。

可移植性 适用于 UNIX 系统, 在 ANSI C 中有定义。

示 例 `#include <string.h>`

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

```
int main(void)
```

```
{
```

```
 char destination[25];
```

```
 char *source = " States";
```

```
 strcpy(destination, "United");
```

```
 strncat(destination, source, 7);
```

```
 printf("%s\n", destination);
```

```
 return 0;
```

```
}
```

## ■ **strncmp** 把串的一部分与另一个串的一部分进行比较

用 法 `#include <string.h>`

```
int strncmp(const char *s1, const char *s2, size_t maxlen);
```

原 型 在 `string.h`

说 明 `strncmp` 进行类似 `strcmp` 的无符号比较, 但最多只比较 `maxlen` 个字符。从每个串的第一个字符开始进行比较, 直到相对应的字符不相同或者比较完 `maxlen` 个字符为止。

返 回 值 根据 `s1` (或 `s1` 的一部分) 和 `s2` (或 `s2` 的一部分) 的比较结果, `strncmp` 返回一个 `int` 型值。

`<0`                      若 `s1` 小于 `s2`

`==0`                     若 `s1` 等于 `s2`

`>0`                      若 `s1` 大于 `s2`

可移植性 适用于 UNIX 系统, 在 ANSI C 中有定义。

参 见 `strcmp`, `strcoll`, `stricmp`, `strncmpi`, `strnicmp`

示 例 `#include <string.h>`

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

```
int main(void)
```

```

{
 char *buf1 = "aaabbb", *buf2 = "bbbccc", *buf3 = "ccc";
 int ptr;
 ptr = strncmp(buf2, buf1, 3);
 if (ptr > 0)
 printf("buffer 2 is greater than buffer 1\n");
 else
 printf("buffer 2 is less than buffer 1\n");
 ptr = strncmp(buf2, buf3, 3);
 if (ptr > 0)
 printf("buffer 2 is greater than buffer 3\n");
 else
 printf("buffer 2 is less than buffer 3\n");
 return(0);
}

```

### ■ strcmp 忽略大小写的串部分比较 ■

**用 法** #include <string.h>

int strcmpi(const char \*s1, const char \*s2, size\_t n);

**原 型 在** string.h

**说 明** strcmpi 对串 s1 和 s2 进行比较, 但最多只比较前 n 个字符。从每个串的第一个字符开始, 按字符顺序进行比较, 直到对应字符不同或达到串尾或比较完 n 个字符为止。比较是不分大小写的。

其返回值(<0, 0 或 >0)取决于 s1(或 s1 的一部分)与 s2(或 s2 的一部分)的比较结果。strcmpi 与 strcmp 相同, 只是前者为宏。函数 strcmp 与 strcmpi 相同, strcmp 是通过 string.h 中定义的宏来实现的, 该宏把 strcmp 调用转化为 strcmpi。因此, 在使用 strcmpi 时, 必须包含 string.h 头文件。该宏是为了与其它 C 编译器兼容而提供的。

**返 回 值** strcmp 返回一个 int 型值:

<0          若 s1 小于 s2  
 ==0        若 s1 等于 s2  
 >0          若 s1 大于 s2

**可移植性** 仅适用于 DOS。

**示 例** #include <string.h>

#include <stdio.h>

int main(void)

```

{
 char *buf1 = "BBBccc", *buf2 = "bbbccc";
 int ptr;
 ptr = strcmpi(buf2, buf1, 3);
 if (ptr > 0)

```

```

 printf("buffer 2 is greater than buffer 1\n");
 if (ptr < 0)
 printf("buffer 2 is less than buffer 1\n");
 if (ptr == 0)
 printf("buffer 2 equals buffer 1\n");
 return 0;
}

```

## ■ strnset 将串中指定数目字节设置为字符 ■

**用 法** #include <string.h>

char \*strnset(char \*s, int ch, size\_t n);

**原 型** 在 string.h

**说 明** strnset 把串 s 中的前 n 个字节置为字符 ch。如果 n > strlen(s)，则由 strlen(s) 代替 n。当 n 个字符已被置换或发现空字符时终止。

**返 回 值** 返回 s。

**可移植性** 仅适用于 DOS。

**示 例** #include <stdio.h>

#include <string.h>

int main(void)

```

{
 char *string = "abcdefghijklmnopqrstuvwxyz";
 char letter = 'x';
 printf("string before strnset: %s\n", string);
 strnset(string, letter, 13);
 printf("string after strnset: %s\n", string);
 return 0;
}

```

## ■ strpbrk 搜索给定集合中任一字符在串中的首次出现 ■

**用 法** #include <string.h>

char \*strpbrk(const char \*s1, const char \*s2);

**原 型** string.h

**说 明** strpbrk 搜索串 s1，找出串 s2 中任一字符的第一次出现。

**返 回 值** strpbrk 返回指向 s2 中任一字符在 s1 中第一次出现的指针，如果 s2 中的字符在 s1 中都不出现，则返回 NULL。

**可移植性** 适用于 UNIX 系统，在 ANSI C 中有定义。

**示 例** #include <stdio.h>

#include <string.h>

int main(void)

```

{
 char *string1 = "abcdefghijklmnopqrstuvwxyz";

```



```

char * string2 = "onm";
char * ptr;
ptr = strpbrk(string1, string2);
if (ptr)
 printf("strpbrk found first character: %c\n", * ptr);
else
 printf("strpbrk didn't find character in set\n");
return 0;
}

```

## ■ strrchr 搜索给定字符在串中的最后一次出现 ■

**用 法** #include <string.h>

char \* strrchr(const char \* s, int c);

**原 型 在** string.h

**说 明** strrchr 逆向搜索字符串, 查找一指定字符 c。strrchr 查找字符 c 在串 s 中的最后一次出现, 空字符终结符被当成是串的一部分。

**返 回 值** strrchr 返回指向最后一次出现字符 c 的指针。若 c 不在 s 中出现, strrchr 返回 NULL。

**可移植性** 适用于 UNIX 系统, 且在 ANSI C 中有定义。

**参 见** strcspn, strchr

**示 例** #include <string.h>

#include <stdio.h>

int main(void)

```

{
 char string[15];
 char * ptr, c = 'r';
 strcpy(string, "This is a string");
 ptr = strrchr(string, c);
 if (ptr)
 printf("The character %c is at position: %d\n", c, ptr - string);
 else
 printf("The character was not found\n");
 return 0;
}

```

## ■ strrev 颠倒串中各字符的顺序 ■

**用 法** #include <string.h>

char \* strrev(char \* s);

**原 型 在** string.h

**说 明** strrev 颠倒串中的字符顺序, 但空终结符除外。例如, 将串 string\0 转变成 gnirts\0。

**返回值** 返回指向颠倒顺序后的串的指针。

**可移植性** 仅适用于 DOS。

**示例**

```
#include <string.h>
#include <stdio.h>
int main(void)
{
 char *forward = "string";
 printf("Before strrev(): %s\n", forward);
 strrev(forward);
 printf("After strrev(): %s\n", forward);
 return 0;
}
```

## ■ **strset** 设置串中所有字符为给定字符

**用法** #include<string.h>  
char \*strset(char \*s,int ch);

**原型在** string.h

**说明** strset 把串中所有字符均设置为给定字符 ch,当发现空字符终结符时退出。

**返回值** 返回 s。

**可移植性** 仅适用于 DOS。

**参见** setmem

**示例**

```
#include <stdio.h>
#include <string.h>
int main(void)
{
 char string[10] = "123456789";
 char symbol = 'c';
 printf("Before strset(): %s\n", string);
 strset(string, symbol);
 printf("After strset(): %s\n", string);
 return 0;
}
```

## ■ **strspn** 搜索给定字符集的子集在串中第一次出现的段

**用法** #include<string.h>  
size\_t strspn(const char \*s1,const char \*s2);

**原型在** string.h

**说明** strspn 搜索出串 s1 中包含串 s2 中全部字符的初始段。

**返回值** strspn 返回串 s1 中包含串 s2 中全部字符的初始段的长度。

**可移植性** 适用于 UNIX 系统,在 ANSI C 中有定义。

**示例** #include <stdio.h>

```
#include <string.h>
#include <alloc.h>
int main(void)
{
 char *string1 = "1234567890";
 char *string2 = "123DC8";
 int length;
 length = strspn(string1, string2);
 printf("Character where strings differ is at position %d\n", length);
 return 0;
}
```

### ■ strstr 搜索给定子串在某串中的出现位置 ■

用 法 #include<string.h>

char \* strstr(const char \* s1, const char \* s2);

原 型 在 string.h

说 明 strstr 返回指向串 s1 中开始出现串 s2 的指针(s1 串包含 s2 串的位置)。若 s2 不在 s1 中出现,则 strstr 返回 NULL。

可移植性 适用于 UNIX 系统,在 ANSI C 中有定义。

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

```
int main(void)
{
 char *str1 = "Borland International", *str2 = "nation", *ptr;
 ptr = strstr(str1, str2);
 printf("The substring is: %s\n", ptr);
 return 0;
}
```

### ■ strtod 把串转换为双精度数值 ■

用 法 #include<stdlib.h>

double strtod(const char \* s, char \* \*endptr);

原 型 在 stdlib.h

说 明 strtod 把字符串 s 转换为双精度数值。s 可以看作双精度值的字符序列,其格式如下:

[ws][sn][ddd][.][ddd][fmt][sn][ddd]

其中:

[ws]=空白字符(可选)

[sn]=符号(+或-)(可选)

[ddd]=数字(可选)

[fmt]=e 或 E(可选)

[.] = 小数点(可选)

strtod 也把 +INF 或 -INF 看作正或负无穷大, 把 +NAN 或 -NAN 看作非数字。

例如, 以下是 strtod 能够转换为双精度值的一些字符串。

+1231.1981 e-1

502.85E2

+2010.952

strtod 在碰到不能解释为合适的双精度值的第一个字符时即停止读后续字符串。

如果 endptr 不是 NULL, 则 strtod 把 \*endptr 置为指向使搜索停止的字符的指针(\*endptr=&stopper)。endptr 用于错误检测。

**返回值** strtod 返回 s 的双精度值, 如果溢出, 则返回正或负 HUGE\_VAL。

**可移植性** 适用于 UNIX 系统, 在 ANSI C 中有定义。

**参 见** atof

**示 例**

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
 char input[80], *endptr;
 double value;
 printf("Enter a floating point number:");
 gets(input);
 value = strtod(input, &endptr);
 printf("The string is %s the number is %lf\n", input, value);
 return 0;
}
```

## ■ strtok 搜索串中的某单词, 该单词由第二个串中指定的符号进行分隔 ■

**用 法** #include <string.h>  
char \*strtok(char \*s1, const char \*s2);

**原 型 在** string.h

**说 明** strtok 将串 s1 看成包含 0 或多个文本单词的序列, 单词间由分隔符串 s2 中的一个或多个字符加以分隔。

第一次调用 strtok 时, 返回指向 s1 中第一个单词的第一个字符的指针, 并在返回单词后立即写一个空字符到 s1 中; 后继的用 NULL 作为第一个参数的调用, 将以这种方法搜索串 s1, 直到没有剩余单词为止。在不同的调用中, 分隔符串 s2 可以不同。

**返回值** strtok 返回指向在 s1 中找到的单词的指针, s1 中无该词时, 返回 NULL。

**可移植性** 适用于 UNIX 系统, 在 ANSI C 中有定义。

**示 例** #include <string.h>

```
#include <stdio.h>
int main(void)
{
 char input[16] = "abc,d";
 char *p;
 /* strtok places a NULL terminator
 in front of the token, if found */
 p = strtok(input, ",");
 if (p) printf("%s\n", p);
 /* A second call to strtok using a NULL
 as the first parameter returns a pointer
 to the character following the token */
 p = strtok(NULL, ",");
 if (p) printf("%s\n", p);
 return 0;
}
```

## ■ strtol 转换串为长整型数 ■

用 法 #include<stdlib.h>

long strtol(const char \*s, char \*\*endptr, int radix);

原 型 在 stdlib.h

说 明 strtol 把字符串 s 转换为长整型数。s 是可看作长整形数的字符序列,其格式如下:

```
[ws][sn][0][x][ddd]
[ws]=空白字符(可选)
[sn]=符号(+或-)(可选)
[0]=零(0)(可选)
[x]=x 或 X(可选)
[ddd]=数字(可选)
```

strtol 在遇到第一个不能识别的字符时停止读字符串。

如果 radix 在 2 到 36 之间,则长整型数以基数 radix 表示;如果 radix 为 0,则由 s 的前几个字符决定转换基数。

| 第一个字符 | 第二个字符 | 字符串解释为 |
|-------|-------|--------|
| 0     | 1-7   | 八进制    |
| 0     | x 或 X | 十六进制   |
| 1-9   |       | 十进制    |

如果 radix 为 1,为无效值;如果 radix<0 或 radix>36,亦为无效值。

任何一个无效的 radix 值都使结果变为 0,且设置下一字符指针 \*endptr 为起始串指针。

如果 s 中的值被解释为八进制,将不识别 0~7 外的字符。

如果 *s* 中的值被解释为十进制, 将不识别 0~9 外的字符。

如果 *s* 中的值被解释为以其它数作基数, 则只有基数中的数字或字母可被识别 (例如 *radix*=5, 则只识别 0~4; 如 *radix*=20, 则只识别 0~9, A~J)。

如果 *endptr* 不为 NULL, 则 *strtol* 将 \**endptr* 置为指向使搜索停止的字符 (\**endptr*=&*stopper*)。

**返回值** *strtol* 返回字符串的转换值, 出错时返回 0。

**可移植性** 适用于 UNIX 系统, 在 ANSI C 中有定义。

**参见** *atoi*, *atol*, *strtoul*

**示例**

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

```
int main(void)
{
 char *string = "87654321", *endptr;
 long lnumber;
 /* strtol converts string to long integer */
 lnumber = strtol(string, &endptr, 10);
 printf("string = %s long = %ld\n", string, lnumber);
 return 0;
}
```

## ■ *strtoul* 将字符串转换为给定基数的无符号长整型值

**用法** *#include*<stdlib.h>

```
unsigned long strtoul(const char *s, char **endptr, int radix);
```

**原型在** *stdlib.h*

**说明** 除将 *str* 转换为无符号长整型之外, *strtoul* 与 *strtol* 完全相同 (*strtol* 是转换成长整型)。请参考有关 *strtol* 的说明。

**返回值** *strtoul* 返回转换后的无符号长整型数; 如果出错, 则返回 0。

**可移植性** 与 ANSI C 兼容。

**参见** *atol*, *strtol*。

**示例**

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

```
int main(void)
{
 char *string = "87654321", *endptr;
 unsigned long lnumber;
 lnumber = strtoul(string, &endptr, 10);
 printf("string = %s long = %lu\n",
 string, lnumber);
 return 0;
}
```

## ■ swab 交换字节 ■

用 法 #include<stdlib.h>

void swab (char \* from, char \* to, int nbytes);

原型在 stdlib.h

说 明 swab 拷贝字符串 from 中的前 nbytes 个字节到字符串 to 中, 且相邻的偶数和奇数位置的字节被交换。这在按不同的字节次序把数据从一台机器传输到另一台机器上时是非常有用的。nbytes 应为偶数。

返回值 无。

可移植性 适用于 UNIX 系统

示 例 #include <stdlib.h>

#include <stdio.h>

#include <string.h>

char source[15] = "rFna koBlrna d";

char target[15];

int main(void)

{

swab(source, target, strlen(source));

printf("This is target: %s\n", target);

return 0;

}

## ■ system 执行 DOS 命令 ■

用 法 #include<stdlib.h>

int system(const char \* command);

原型在 stdlib, process.h

说 明 system 从一个正在执行的 C 程序里, 利用 DOS 和 COMMAND.COM 文件来执行 DOS 命令, 批处理文件或字符串 comand 所指定的程序。

为了装入和运行, 该程序必须在当前目录下或在环境变量 PATH 所指的目录之一中。

环境变量 COMSPEC 用于查找 COMMAND.COM 文件, 所以 COMMAND.COM 不一定在当前目录中。

返回值 调用成功时返回 0, 否则返回 -1。

可移植性 适用于 UNIX 系统, 在 ANSI C 中有定义。与 Kernighan 和 Ritchie 的定义兼容。

参 见 exec..., \_fpreset, searchpath, spawn...

示 例 #include <stdlib.h>

#include <stdio.h>

int main(void)

{

printf("About to spawn command.com and run a DOS command\n");

```

 system("dir");
 return 0;
}

```

## ■ tan 计算正切值 ■

- 用 法** 对于实数 对于复数  
           #include <math.h> #include <complex.h>  
           double tan(double x); complex tan(complex x);
- 原 型 在** 对于实数 对于复数  
           math.h complex.h
- 说 明** tan 计算正切值, 参数角度以弧度表示。该函数的错误处理程序可以通过 matherr 函数进行修改。  
       复数的正切定义如下:  
        $\tan(z) = \sin(z) / \cos(z)$
- 返 回 值** tan 返回 x 的正切, 值即  $\sin(x) / \cos(x)$ 。
- 可移植性** tan 的实数版本适用于 UNIX 系统, 在 ANSI C 中有定义。tan 的实数版本仅适用于 Turbo C 且不可移植。
- 参 见** acos, asin, atan, atan2, complex, cos, sin
- 示 例**
- ```

#include <stdio.h>
#include <math.h>
int main(void)
{
    double result, x;
    x = 0.5;
    result = tan(x);
    printf("The tan of %lf is %lf\n", x, result);
    return 0;
}

```

■ tanh 计算参数 x 的双曲正切值 ■

- 用 法** 对于实数 对于复数
 #include <math.h> #include <complex.h>
 double tanh(double x); complex tanh(complex x);
- 原 型 在** 对于实数 对于复数
 math.h complex.h
- 说 明** tanh 计算双曲正切值, 即 $\sinh(x) / \cosh(x)$ 。tanh 的错误处理程序可通过 matherr 加以修改。
 复数的双曲正切定义如下:
 $\tanh(z) = \sinh(z) / \cosh(z)$
- 返 回 值** tanh 返回 x 的双曲正切值。

可移植性 tanh 的实数版本适用于 UNIX 系统, 在 ANSI C 中有定义。tanh 的实数版本仅适用于 Turbo C, 且不可移植。

参 见 complex, cos, cosh, sin, sinh, tan

示 例

```
#include <stdio.h>
#include <math.h>
int main(void)
{
    double result, x;
    x = 0.5;
    result = tanh(x);
    printf("The hyperbolic tangent of %lf is %lf\n", x, result);
    return 0;
}
```

■ tell 取文件指针的当前位置 ■

用 法 #include<io.h>
long tell(int handle);

原 型 在 io.h

说 明 tell 取与文件句柄 handle 相联的文件指针的当前位置, 并把它表示为从文件头算起的字节数。

返 回 值 tell 返回当前的文件指针位置。返回 -1 (长整型) 时表示出错, 此时把全局变量 errno 置为 EBADF (无效文件号)。

可移植性 适用于 UNIX 系统。

参 见 fgetpos, fseek, ftell, lseek

示 例

```
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
int main(void)
{
    int handle;
    char msg[] = "Hello world";
    if ((handle = open("TEST. $$$", O_CREAT|O_TEXT|O_APPEND)) == -1)
    {
        perror("Error:");
        return 1;
    }
    write(handle, msg, strlen(msg));
    printf("The file pointer is at byte %ld\n", tell(handle));
    close(handle);
    return 0;
}
```

}

■ textattr 设置文本属性 ■

用 法 #include <conio.h>

void textattr(int newattr);

原 型 在 conio.h

说 明 通过调用 textattr 函数,可同时设置前景颜色和背景颜色(一般要用 textcolor 函数和 textbackground 函数分别设置它们)。该函数不影响当前屏幕上的任何字符,它只影响调用该函数后由直接视频文本模式输出函数(如 cprintf)所显示的字符。

颜色信息在参数 newattr 中的编码如下:

7 6 5 4 3 2 1 0

B b b b f f f f

在这个 8 位的参数 newattr 中:

ffff 代表 4 位前景颜色(0~15)

bbb 代表 3 位背景颜色(0~7)

B 是闪烁允许位(第 7 位)

如果打开闪烁允许位,字符将闪烁。这可以通过把属性值加上常数 BLINK 来实现。如果使用 conio.h 中定义的符号颜色常量,通过 textattr 来建立文本属性,请注意对背景颜色选择的如下一些限制:

- 只能为背景选择前 8 种颜色之一。
- 必须将所选的背景颜色左移 4 位,将其移到正确位置。

下表列出这些符号常量值:

符号常量	数值	前景或背景
BLACK	0	均可
BLUE	1	均可
GREEN	2	均可
CYAN	3	均可
RED	4	均可
MAGENTA	5	均可
BROWN	6	均可
LIGHTGRAY	7	均可
DARKGRAY	8	限于前景
LIGHTBLUE	9	限于前景
LIGHTGREEN	10	限于前景
LIGHTCYAN	11	限于前景
LIGHTRED	12	限于前景
LIGHTMAGENTA	13	限于前景
YELLOW	14	限于前景
WHITE	15	限于前景

符号常量	数值	前景或背景
BLINK	128	限于前景

返回值 无。

可移植性 仅适用于 IBM PC 及其兼容机。

参见 gettextinfo, highvideo, lowvideo, normvideo, textbackground, textcolor

示例 #include <conio.h>

```
int main(void)
{
    int i;
    clrscr();
    for (i=0; i<9; i++)
    {
        textattr(i + ((i+1) << 4));
        printf("This is a test\r\n");
    }
    return 0;
}
```

■ textbackground 选择文本的背景颜色 ■

用法 #include<conio.h>

void textbackground(int newcolor);

原型在 conio.h

说明 textbackground 可选择背景颜色,为直接屏幕文本输出函数提供服务。newcolor 为所选的背景颜色。newcolor 可取 0~7 之间的整数或 conio.h 中定义的符号常数。若使用符号常数,则应包含 conio.h 头文件。一旦调用了 textbackground,那么后续的直接视频输出函数均以 newcolor 为背景颜色。textbackground 不影响当前屏幕上的任何字符。

符号常量	数字值
BLACK	0
BLUE	1
GREEN	2
CYAN	3
RED	4
MAGENTA	5
BROWN	6
LIGHTGRAY	7

返回值 无。

可移植性 仅适用于 IBM PC 及其兼容机。Turbo Pascal 中有相应的函数。

参 见 `gettextinfo, textattr, textcolor`

示 例 `#include <conio.h>`
`int main(void)`
`{`
`int i, j;`
`clrscr();`
`for (i=0; i<9; i++)`
`{`
`for (j=0; j<80; j++)`
`cprintf("C");`
`cprintf("\r\n");`
`textcolor(i+1);`
`textbackground(i);`
`}`
`return 0;`
`}`

■ `textcolor` 选择文本模式的前景颜色 ■

用 法 `#include <conio.h>`
`void textcolor(int newcolor);`

原 型 在 `conio.h`

说 明 `textcolor` 可选择前景字符颜色, 为控制台输出函数服务。 `newcolor` 为新的前景颜色。 `newcolor` 的值可置为下表中所给的一个整数或 `conio.h` 中定义的一个符号常量。若使用了符号常量, 则必须包含 `conio.h` 文件。

一旦调用了 `textcolor`, 以后调用的直接视频输出函数 (例如 `cprintf`) 就将使用 `newcolor`。 `textcolor` 并不影响当前屏幕上的任何字符。

下表列出了所允许使用的颜色 (符号常量) 及其的数字值:

符号常量	数字值
BLACK	0
BLUE	1
GREEN	2
RED	4
MAGENTA	5
BROWN	6
LIGHTGRAY	7
DARKGRAY	8
LIGHTBLUE	9
LIGHTGREEN	10
LIGHTCYAN	11
LIGHTRED	12
LIGHTMAGENTA	13

符号常量	数字值
YELLOW	14
WHITE	15
BLINK	128

可将前景颜色加 0x80 来使字符闪烁,预定义的常量 BLINK 就用于这个目的。例如:

```
ftextcolor(GYAN+BLINK);
```

注意:有一些监视器不能识别用于产生 8 种亮度颜色(8~15)的强度信号。在这种监视器上,亮度颜色的显示等同于“深度颜色”(0~7)的显示。另外,不能显示彩色的系统可能将这些数字看成是一种颜色的灰度、特殊模式或特殊属性(如下划线、黑体、斜体等等)。在这样的系统中具体表现为何种属,取决于系统硬件。

返回值 无

可移植性 用于 IBM PC 及其兼容机,在 Turbo Pascal 中有相应的函数。

参见 gettextinfo, highvideo, lowvideo, normvideo, textattr, textbackground

```

示 例 #include <conio.h>
      int main(void)
      {
          int i;
          for (i=0; i<15; i++)
          {
              textcolor(i);
              cprintf("Foreground Color\r\n");
          }
          return 0;
      }

```

■ textheight 返回以像素为单位的字符串高度 ■

用法 #include <graphics.h>
int far textheight(char far *textstring);

原型在 graphics.h

说明 图形函数 textheight 根据当前字体的大小和放大因子,以像素为单位确定字符串 textstring 的高度。该函数对于调整行间距,计算视区高度,确定标题尺寸使其置于图形或方框中的合适位置,是非常有用的。

例如,在 8×8 位图字体和放大因子为 1 的情况(用 settextstyle 设置)下,字符串 Turbo C 为 8 个像素高。

注意:textheight 可用来代替自己编码计算字符串高度,借助于此函数,选择了不同的字体时无需修改源代码。

返回值 返回以像素为单位的字符串高度。

可移植性 仅用于 Turbo C、配置图形适配器的 IBM PC 及其兼容机。

参 见 gettextsttings, outtext, outtextxy, settextstyle, textwidth

示 例

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

int main(void)
{
    /* request auto detection */
    int gdriver = DETECT, gmode, errorcode;
    int y = 0;
    int i;
    char msg[80];
    /* initialize graphics and local variables */
    initgraph(&gdriver, &gmode, "");
    /* read result of initialization */
    errorcode = graphresult();
    if (errorcode != grOk) /* an error occurred */
    {
        printf("Graphics error: %s\n", grapherrormsg(errorcode));
        printf("Press any key to halt:");
        getch();
        exit(1); /* terminate with an error code */
    }

    /* draw some text on the screen */
    for (i=1; i<11; i++)
    {
        /* select the text style, direction, and size */
        settextstyle(TRIPLEX_FONT, HORIZ_DIR, i);
        /* create a message string */
        sprintf(msg, "Size: %d", i);
        /* output the message */
        outtextxy(1, y, msg);
        /* advance to the next text line */
        y += textheight(msg);
    }

    /* clean up */
    getch();
    closegraph();
    return 0;
}
```

■ textmode 将屏幕设置成文本模式 ■

用 法 #include <conio.h>

void textmode(int newmode);

原 型 在 conio.h

说 明 textmode 设置指定的文本模式。通过使用枚举类型 text_mode 中的符号常量 (在 conio.h 中定义) 来给出文本模式 (参数 newmode)。使用这些常量时, 必须包含 conio.h。对于 text_mode 类型的符号量, 其数值和所指定的模式见下表:

符号常量	数值	文本模式
LASTMODE	-1	原文本模式
BW40	0	黑白, 40 列
C40	1	彩色, 40 列
BW80	2	黑白, 80 列
C80	3	彩色, 80 列
MONO	7	单色, 80 列
C4350	64	EGA 43 行和 VGA 50 行模式

调用 textmode 函数后, 当前窗口重新置为全屏, 当前文本属性被置为类似调用 normvideo 后的正常属性。

对 textmode 指定 LASTMODE 可以重新选择上一次选择的文本模式。textmode 只有在屏幕为文本模式时方可使用 (用来变成另一种文本模式), 这是使用 textmode 的必需条件。若屏幕为图形模式, 应使用 restorectrmode 将其变成文本模式。

返 回 值 无

可移植性 用于 IBM PC 及其兼容机, Turbo Pascal 中有相似的函数。

参 见 gettextinfo, window

示 例 #include <conio.h>

```
int main(void)
{
    textmode(BW40);
    cprintf("ABC");
    getch();
    textmode(C40);
    cprintf("ABC");
    getch();
    textmode(BW80);
    cprintf("ABC");
    getch();
    textmode(C80);
    cprintf("ABC");
    getch();
}
```

```

    textmode(MONO);
    cprintf("ABC");
    getch();
    return 0;
}

```

■ textwidth 返回以像素为单位的字符串宽度 ■

用 法 #include <graphics.h>

int far textwidth(char far *textstring);

原 型 在 graphics.h

说 明 图形函数 textwidth 根据当前的字体大小和放大因子,以像素为单位确定字符串 textstring 的宽度。对于计算视区的宽度,确定标题尺寸以使其置于图表或方框中的合适位置是非常有用的。

注意:textwidth 可用来代替自己编码计算字符串宽度;借助于此函数,选择了不同的字体时,无需修改源代码。

返 回 值 返回以像素为单位的字符串宽度。

可移植性 仅适用于 Turbo C、配置了图形适配器的 IBM PC 及其兼容机。

参 见 gettextsettings, outtext, outtextxy, settextstyle, textheight

示 例 #include <graphics.h>

#include <stdlib.h>

#include <stdio.h>

#include <conio.h>

int main(void)

{

/* request auto detection */

int gdriver = DETECT, gmode, errorcode;

int x = 0, y = 0;

int i;

char msg[80];

/* initialize graphics and local variables */

initgraph(&gdriver, &gmode, "");

/* read result of initialization */

errorcode = graphresult();

if (errorcode != grOk) /* an error occurred */

{

printf("Graphics error: %s\n", grapherrormsg(errorcode));

printf("Press any key to halt;");

getch();

exit(1); /* terminate with an error code */

}

y = getmaxy() / 2;


```

settextjustify(LEFT_TEXT, CENTER_TEXT);
for (i=1; i<11; i++)
{
    /* select the text style, direction, and size */
    settextstyle(TRIPLEX_FONT, HORIZ_DIR, i);
    /* create a message string */
    sprintf(msg, "Size: %d", i);
    /* output the message */
    outtextxy(x, y, msg);
    /* advance to the end of the text */
    x += textwidth(msg);
}
/* clean up */
getch();
closegraph();
return 0;
}

```

■ time 取时间 ■

用 法 #include <time.h>

time_t time(time_t * timer);

原 型 在 time.h

说 明 time 取以秒为单位的, 从 1970 年 1 月 1 日格林威治时间 00:00:00 算起的当前时间, 并将其存入由 timer 所指的位置中。timer 不能为空指针。

返 回 值 返回如前所述的时间值。

可移植性 适用于 UNIX 系统, 在 ANSI C 中有定义。

参 见 asctime, ctime, difftime, ftime, gettime, gmtime, localtime, settime, s_time, tzset

示 例 #include <time.h>

#include <stdio.h>

#include <dos.h>

```

int main(void)
{
    time_t t;
    t = time(NULL);
    printf("The number of seconds since January 1, 1970 is %ld", t);
    return 0;
}

```

■ tmpfile 以二进制方式打开临时文件 ■

用 法 #include <stdli.h>

FILE tmpfile(void);

原型在 stdio.h

说明 tmpfile 建立一个临时文件并将其打开。当关闭该文件或程序结束时,该文件将被自动删除。

返回值 返回指向所创建的临时文件流的指针;如果文件不能创建,就返回 NULL。

可移植性 适用于 UNIX 系统,在 ANSI C 中有定义。

参见 fopen, tmpnam

示例

```
#include <stdio.h>
#include <process.h>
int main(void)
{
    FILE * tempfp;
    tempfp = tmpfile();
    if (tempfp)
        printf("Temporary file created\n");
    else
    {
        printf("Unable to create temporary file\n");
        exit(1);
    }
    return 0;
}
```

■ tmpnam 创建唯一的文件名 ■

用法 #include<stdio.h>
char * tmpnam(char s);

原型在 stdio.h

说明 tmpnam 创建一个唯一的文件名,该文件名可以有效地作为临时文件的名称。每次调用 tmpnam 时均生成不同的字符串,最多可成 TMP_MAX 次。在 stdio.h 中,TMP_MAX 被定义为 65535。参数 s 要么为 NULL,要么为指向至少 1_tmpnam 个字符的数组的指针。l_tmpnam 在 stdio.h 中定义。如果 s 为 NULL,tmpnam 就把生成的临时文件名存入一内部静态对象中,并返回指向该对象的指针;如果 s 不为 NULL,tmpnam 将把结果存入所指定的数组里(该数组至少 1_tmpnam 个字符长),然后返回 s。

注意:如果用户用 tmpnam 建立一个临时文件,则必须由用户删除该文件(例如调用 remove),因为它不会被自动删除。(tmpfile 则是自动删除文件名)。

返回值 若 s 为 NULL,tmpnam 返回一指向内部静态对象的指针;否则,tmpnam 返回 s。

可移植性 适用于 UNIX 系统,在 ANSI C 中有定义。

参见 tmpfile

示例 #include <stdio.h>

```
int main(void)
{
    char name[13];
    tmpnam(name);
    printf("Temporary name: %s\n", name);
    return 0;
}
```

■ toascii 转换字符为 ASCII 格式 ■

用 法 #include <ctype.h>

int toascii(int c);

原 型 在 ctype.h

说 明 toascii 是一个宏,它通过清除低 7 位之外的所有其它位,进而将整数 c 转换为 ASCII 码,即一个 0~127 范围内的值。

返 回 值 返回 c 的转换值。

可移植性 适用于 UNIX 系统。

示 例 #include <stdio.h>
#include <ctype.h>
int main(void)
{
 int number, result;
 number = 511;
 result = toascii(number);
 printf("%d %d\n", number, result);
 return 0;
}

■ tolower 转换字母为小写 ■

用 法 #include <ctype.h>

int _tolower(int ch);

原 型 在 ctype.h

说 明 _tolower 是与 tolower 功能相似的宏,但前者仅适用于已知 ch 是大写字母(A~Z)的情况。若使用 _tolower,必须包含 ctype.h 文件。

返 回 值 如果 ch 是大写字母,则返回 ch 的转换值;否则,返回结果无意义。

可移植性 适用于 UNIX 系统。

示 例 #include <string.h>
#include <stdio.h>
#include <ctype.h>
int main(void)
{
 int length, i;

```
char *string = "THIS IS A STRING.";
/* We should be checking each character to
make sure it is an uppercase before passing
it to _tolower! The result of passing it a
non-uppercase is undefined. */
length = strlen(string);
for (i = 0; i < length; i++) {
    string[i] = _tolower(string[i]);
}
printf("%s\n", string);
return 0;
}
```

■ tolower 转换字符为小写 ■

用 法 #include <ctype.h>
int tolower(int ch);

原 型 在 ctype.h

说 明 tolower 用于把整数值 ch (范围在 EOF 到 255 之间) 转换为小写字母值 (如果原来为 A~Z, 则转换为 a~z; 否则, 不转换)。

返 回 值 如果 ch 为大写字母, 则 tolower 返回 ch 的转换值; 否则, 返回未转换的值。

可移植性 适用于 UNIX 系统, 在 ANSI C 中有定义; 与 kernighan 和 Ritchie 的定义兼容。

示 例 #include <string.h>
#include <stdio.h>
#include <ctype.h>
int main(void)
{
 int length, i;
 char *string = "THIS IS A STRING";
 length = strlen(string);
 for (i=0; i<length; i++)
 {
 string[i] = tolower(string[i]);
 }
 printf("%s\n", string);
 return 0;
}

■ toupper 转换字母为大写 ■

用 法 #include <ctype.h>
int _toupper(int ch);

原 型 在 ctype.h

说 明 `_toupper` 与 `toupper` 功能相似的宏,但前者仅适用于已知 `ch` 是小写字母(`a~z`)的情况。若使用 `_toupper`,必须包含 `ctype.h` 文件。

返回值 如果 `ch` 是小写字母,则返回 `ch` 的转换值;否则,返回结果无意义。

可移植性 适用于 UNIX 系统。

示 例

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

int main(void)
{
    int length, i;
    char *string = "this is a string";
    /* We should be checking each character to
       make sure it is lowercase before passing it
       to _toupper. The result passing a non
       lowercase is undefined. */
    length = strlen(string);
    for (i = 0; i < length; i++) {
        string[i] = _toupper(string[i]);
    }
    printf("%s\n", string);
    return 0;
}
```

■ `toupper` 转换字符为大写 ■

用 法 `#include <ctype.h>`
`int toupper(int ch);`

原 型 在 `ctype.h`

说 明 `toupper` 用于把整数值 `ch`(范围在 EOF 到 255 之间)转换为大写字母值(如果原来值为 `a~z`,则转换为 `A~Z`;否则,不转换)。

返回值 如果 `ch` 为小写字母,`toupper` 返回 `ch` 的转换值;否则,返回未转换的值。

可移植性 可适用于 UNIX 系统,在 ANSI C 中有定义,且与 Kernighan 和 Ritchie 的定义兼容。

示 例

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

int main(void)
{
    int length, i;
    char *string = "this is a string";
    length = strlen(string);
    for (i=0; i<length; i++)
```

```

{
    string[i] = toupper(string[i]);
}
printf("%s\n", string);
return 0;
}

```

■ ttrg 三角函数 ■

用 法 double acos(double x);
double asin(double x);
double atan(double x);
double atan2(double y, double x);
double cos(double x);
double sin(double x);
double tan(double x);

原 型 在 math.h

说 明 sin, cos 和 tan 返回对应的三角函数值, 角度以弧度表示。

asin, acos 和 atan 各自返回输入值的反正弦、反余弦和反正切值。传给 asin 和 acos 的参数必须在 -1 和 1 之间, 此范围外的值将使 asin 和 acos 返回 0 并置 errno 为:

EDOM 域错误。

atan2 返回 y/x 的反正切值, 即使角度接近 $\pi/2$ 或 $-\pi/2$ (即 x 接近于 0), 也能产生正确的结果。

返 回 值 sin 和 cos 返回一范围在 -1 和 1 之间的值;

asin 返回一范围在 $-\pi/2$ 到 $\pi/2$ 之间的值;

acos 返回一范围在 0 到 π 之间的值;

atan 返回一范围在 $-\pi/2$ 到 $\pi/2$ 之间的值;

atan2 返回一范围在 $-\pi/2$ 到 $\pi/2$ 之间的值;

tan 返回有效角度的任何值, 对于角度接近于 $\pi/2$ 到 $-\pi/2$ 的情况, tan 返回 0, errno 被置为:

ERANGE 结果超过范围

这些函数的错误处理程序可以通过 mantherr 函数修改。

可移植性 适用于 UNIX 系统

参 见 _mather, matherr, perror

■ tzset 设置全局变量 daylight、timezone 和 tzname 的值 ■

用 法 #include <time.h>

void tzset(void);

原 型 在 time.h

说明 `tzset` 设置基于环境变量 `TZ` 的全局变量 `daylight`、`timezone` 和 `tzname` 的值。`ftime` 和 `localtime` 则用这些全局变量来把格林威治时间(GMT)修正为当地时区时间。`TZ` 环境字符串的格式如下:

`TZ=zzz [+/-]d [d] [lll]`

`zzz` 是表示当前时区名的由 3 个字符组成的串,所有 3 个字符都是不可少的。例如,串“PST”用于表示太平洋标准时间;[+/-]d [d]是一个包含 1 或多位数字的带可选符号的数值字段。该数值为本地时区与 GMT 所差的小时数。正整由 GMT 向东修正,负数由 GMT 向西修正。例如,数值 5=EST、+8=PST, -1=欧洲大陆。在计算全局变量 `timezone` 时要用到该数值。`timezone` 为当地时区与 GMT 相差的秒数;

`lll` 是可选的代表当地时区夏令时间的 3 字符字段。例如,串 PDT 用来表示太平洋夏令时间。若使用了该字段,将使全局变量 `daylight` 设置为非 0 值,若未用该字段,`daylight` 将被置为 0。

如果没有设置 `TZ` 环境串或不符合上述格式,则以缺省值 `TZ="EST5PDT"` 来为 `daylight`、`timezone` 和 `tzname` 赋值。全局变量 `tzname[0]` 指向一个 3 个字符的字符串,该串为从 `TZ` 环境获得的时区名。`tzname[1]` 指向一个从 `TZ` 环境获得的代表夏令时区名的由 3 个字符组成的字符串。如果没有使用夏令时名字,则 `tzname[1]` 指向一个空串。

返回值 无。

可移植性 可用于 UNIX 和 XENIX 系统。

参见 `asctime`, `ctime`, `ftime`, `gmtime`, `localtime`, `time`, `time`

示例

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
    time_t td;
    putenv("TZ=PST8PDT");
    tzset();
    time(&td);
    printf("Current time = %s\n", asctime(localtime(&td)));
    return 0;
}
```

■ `ultoa` 转换无符号长整型值为字符串 ■

用法 `#include <stdlib.h>`

`char ultoa(unsigned long value, char string, int radix);`

原型在 `stdlib.h`

说明 `ultoa` 把 `value` 值转换为以空字符终结的字符串,并把结果存入 `string` 中。参数 `value` 是无符号长整型数。

radix 指明在转换 value 过程中使用的基数值,它必须在 2~36 之间。ultoa 不进行溢出检查;如果 value 为负且 radix=10,ultoa 将不设置负号('-')。

注意:分配给 string 的空间必须足够大,以容纳所返回的所有字符(包括空字符终结符\0)。ultoa 最多返回 33 个字节。

返回值 返回 string。

可移植性 仅适用于 DOS。

参见 itoa, ltoa

示例

```
#include <stdlib.h>
#include <stdio.h>
int main( void )
{
    unsigned long lnumber = 3123456789L;
    char string[25];
    ultoa(lnumber, string, 10);
    printf("string = %s unsigned long = %lu\n", string, lnumber);
    return 0;
}
```

■ ungetc 把一个字符回退到输入流中 ■

用法 #include<stdio.h>

```
int ungetc(int c, FILE stream);
```

原型在 stdio.h

说明 ungetc 把字符 c 回退到指定的输入流 stream 中,该流必须已经以读模式被打开。该字符在下次对流 stream 调用 getc 或 fread 时被返回。在所有情况下,都只能回退一个字符。在未调用 getc 的情况下再次调用 ungetc 将使前一个字符丢失。调用 fflush、fseek、fsetpos 和 rewind 将清除所有被退回字符所占用的内存。

返回值 调用成功时,ungetc 返回退回的字符,否则,该函数返回 EOF。

可移植性 适用于 UNIX 系统,在 ANSI C 中有定义。

参见 fgetc, getc, getchar

示例

```
#include <stdio.h>
#include <ctype.h>
int main( void )
{
    int i=0;
    char ch;
    puts("Input an integer followed by a char:");
    /* read chars until non digit or EOF */
    while((ch = getchar()) != EOF && isdigit(ch))
        i = 10 * i + ch - 48; /* convert ASCII into int value */
    /* if non digit char was read, push it back into input buffer */
    if (ch != EOF)
```



```

    ungetc(ch, stdin);
    printf("i = %d, next char in buffer = %c\n", i, getch());
    return 0;
}

```

■ ungetch 把一个字符回送到键盘缓冲区 ■

用 法 #include <conio.h>

int ungetch(int ch);

原 型 在 conio.h

说 明 ungetch 把字符 ch 回退到控制台,使其成为下一个将要读取的字符。在下次读字符之前多次调用 ungetch,将失败。

返 回 值 调用成功时返回 ch;否则返回 EOF。

可移植性 适用于 UNIX 系统。

参 见 getch, getche

示 例

```

#include <stdio.h>
#include <ctype.h>
#include <conio.h>
int main( void )
{
    int i=0;
    char ch;
    puts("Input an integer followed by a char:");
    /* read chars until non digit or EOF */
    while((ch = getche()) != EOF && isdigit(ch))
        i = 10 * i + ch - 48; /* convert ASCII into int value */
    /* if non digit char was read, push it back into input buffer */
    if (ch != EOF)
        ungetch(ch);
    printf("\n\ni = %d, next char in buffer = %c\n", i, getch());
    return 0;
}

```

■ unixtodos 把 UNIX 格式的日期和时间转换成 DOS 格式 ■

用 法 #include <dos.h>

void unixtodos(long time, struct dated, struct timet);

原 型 在 dos.h

说 明 unixtodos 把由 time 给出的 UNIX 格式的时间转换为 DOS 格式,并将转换结果存入分别由 d 和 t 所指定的 date 和 time 结构中。

返 回 值 无。

可移植性 仅适用于 DOS。

参 见 dostounix

```

示 例 #include <stdio.h>
       #include <dos.h>
       char * month[] = {"---", "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                          "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
       #define SECONDS_PER_DAY 86400L /* the number of seconds in one day */
       struct date dt;
       struct time tm;
       int main(void)
       {
           unsigned long val;
           /* get today's date and time */
           getdate(&dt);
           gettime(&tm);
           printf("today is %d %s %d\n", dt.da_day, month[dt.da_mon], dt.da_year);
           /* convert date and time to unix format (num of seconds since Jan 1, 1970 */
           val = dostounix(&dt, &tm);
           /* subtract 42 days worth of seconds */
           val -= (SECONDS_PER_DAY * 42);
           /* convert back to dos time and date */
           unixtodos(val, &dt, &tm);
           printf("42 days ago it was %d %s %d\n",
                  dt.da_day, month[dt.da_mon], dt.da_year);
           return 0;
       }

```

■ unlink 删除文件

用 法 #include <io.h>

int unlink(const char filename);

原 型 在 io.h

说 明 unlink 删除由 filename 所指的文件。filename 中可以包含任意的 DOS 驱动器号、路径和文件名, 不允许使用通配符。

只读文件不能通过该调用而删除。若要删除只读文件, 必须首先使用 chmod 或 chmod 来改变其只读属性。

注意: 如果文件已打开, 在删除之前, 必须先将其关闭。

返 回 值 调用成功时, 返回 0; 否则, 返回 -1, 并置全局变量 errno 为下列值之一:

ENOENT 路径或文件名没有找到

EACCES 无此权限

可移植性 适用于 UNIX 系统

参 见 chmod, remove

示 例 #include <stdio.h>
#include <io.h>

```

int main(void)
{
    FILE *fp = fopen("junk.jnk","w");
    int status;
    fprintf(fp,"junk");
    status = access("junk.jnk",0);
    if (status == 0)
        printf("File exists\n");
    else
        printf("File doesn't exist\n");
    fclose(fp);
    unlink("junk.jnk");
    status = access("junk.jnk",0);
    if (status == 0)
        printf("File exists\n");
    else
        printf("File doesn't exist\n");
    return 0;
}

```

■ unlock 解除文件共享锁 ■

用 法 #include <io.h>

int unlock(int handle, long offset, long length);

原 型 在 io.h

说 明 unlock 为 DOS3.x 中的文件共享机制提供接口。unlock 解除先前通过调用 lock 所设的锁。为避免出错,在关闭文件之前必须解除所有的锁。在执行完程序之前必须释放所有的锁。

返 回 值 成功时返回 0; 否则返回 -1。

可移植性 仅适用于 DOS 3.x, DOS 的早期版本不支持该调用。

参 见 lock, sopen

示 例 #include <io.h>

#include <fcntl.h>

#include <sys\stat.h>

#include <process.h>

#include <share.h>

#include <stdio.h>

int main(void)

{

int handle, status;

long length;

handle = sopen("c:\\autoexec.bat", O_RDONLY, SH_DENYNO, S_IREAD);

```
if (handle < 0)
{
    printf("sopen failed\n");
    exit(1);
}
length = filelength(handle);
status = lock(handle, 0L, length/2);
if (status == 0)
    printf("lock succeeded\n");
else
    printf("lock failed\n");
status = unlock(handle, 0L, length/2);
if (status == 0)
    printf("unlock succeeded\n");
else
    printf("unlock failed\n");
close(handle);
return 0;
}
```

■ va_arg, va_end, va_start 实现可变参数表 ■

用 法 #include<stdarg.h>

void va_start(va_list ap, lastfix);

type va_arg(va_list ap, type);

void va_end(va_list ap);

原 型 在 stdarg.h

说 明 有些 C 函数, 如 `vfprintf` 和 `vprintf`, 使用可变的参数表而不是固定数目的参数。

`va_arg`、`va_end`、`va_start` 宏提供了一种存取这些参数表的方便方法。当被调用函数不知道参数的个数和类型时, 可以用它们测试参数表。

头文件 `stdarg.h` 中说明了一个类型 (`va_list`) 和三个宏 (`va_start`、`va_arg` 及 `va_end`)。

1. va_list

该数组用于存放 `va_arg` 和 `va_end` 所需的信息。当被调用函数使用一个可变参数表时, 它说明一个类型为 `va_list` 的变量 `ap`。

2. va_start

该子程序(用宏实现)使 `ap` 指向传递给函数的可变参数表中的第一个参数。在首次调用 `va_arg` 或 `va_end` 之前, 必须先调用 `va_start`。

`va_start` 接受两个参数: `ap` 和 `lastfix` (`ap` 已在前段 `va_list` 解释; `lastfix` 传递给被调用函数的最后一个固定参数的名字)。

3. va_arg

该子程序(也用宏实现)扩展表达式以使其与下一个被传递的参数具有相同的类

型和值。va_arg 中的变量 ap 应与由 va_start 初始化的 ap 相同。

注意,由于缺省提示,va_arg 可使用 char、unsigned char 和 float 类型。

在第一次使用 va_arg 时,它返回表中的第一个参数,后续的每次调用都返回表中的下一个参数。这是通过先访问 ap,然后把它增加以指向下一项而实现的。

va_arg 使用 type 来完成访问和定位后继项。每调用一次 va_arg,它都修改 ap 以指向表中的下一参数。

4. va_end

该宏用于被调用函数中以完成一正常返回。va_end 可以修改 ap 使其在重新调用 va_start 以前不能被使用。va_end 必须在 va_arg 读完所有的参数后再调用,否则会产生意想不到的后果。

返回值 va_start 和 va_end 不返回值,va_arg 返回表中的当前参数(ap 所指的)。

可移植性 适用于 UNIX 系统。

参 见 v...printf, v...scanf

示 例

```
#include <stdio.h>
#include <stdarg.h>
/* calculate sum of a 0 terminated list */
void sum(char *msg, ...)
{
    int total = 0;
    va_list ap;
    int arg;
    va_start(ap, msg);
    while ((arg = va_arg(ap, int)) != 0) {
        total += arg;
    }
    printf(msg, total);
    va_end(ap);
}

int main(void) {
    sum("The total of 1+2+3+4 is %d\n", 1, 2, 3, 4, 0);
    return 0;
}
```

■ vfprintf 送格式化输出到一流中 ■

用 法 #include<stdio.h>

```
int vfprintf(FILE stream, const char format, va_list arglist);
```

原型在 stdio.h

说 明 v...printf 函数是...printf 函数的可选择入口点,其功能与...printf 类似,不过,v...printf 接受参数表指针而不是接受参数表。

vfprintf 接受一指向参数表的指针,针对每一参数,由 format 所指的格式串中对应包含一个格式指示符。vfprintf 把格式化的数据送到一个流中。格式指示符的

个数与参数个数必须相同。

返回值 `vfprintf` 返回输出的字节数,出错时则返回 EOF。

可移植性 可适用于 UNIX 系统 V,与 ANSI C 中的定义兼容。

参见 `printf`, `va_arg`, `va_end`, `va_start`

示例

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

FILE *fp;

int vfpf(char *fmt, ...)
{
    va_list argptr;
    int cnt;
    va_start(argptr, fmt);
    cnt = vfprintf(fp, fmt, argptr);
    va_end(argptr);
    return(cnt);
}

int main(void)
{
    int inumber = 30;
    float fnumber = 90.0;
    char string[4] = "abc";
    fp = tmpfile();
    if (fp == NULL)
    {
        perror("tmpfile() call");
        exit(1);
    }
    vfpf("# %d %f %s", inumber, fnumber, string);
    rewind(fp);
    fscanf(fp, "# %d %f %s", &inumber, &fnumber, string);
    printf("# %d %f %s\n", inumber, fnumber, string);
    fclose(fp);
    return 0;
}
```

■ `vfscanf` 从流中搜索和格式化输入

用法 `#include <stdio.h>`

`int vfscanf(FILE *stream, const char *format, va_list arglist);`

原型在 `stdio.h`

说明 `v...scanf` 函数是...`scanf` 函数可选择的入口点,其操作与...`scanf` 类似,不过,`v...scanf` 接受参数表指针而不是接受参数表。

`vfscanf` 从一个流中扫描输入字段,每次读取一个字符,然后根据格式指示符对每个字段进行格式化(指示符在 `format` 所指的传给 `vfscanf` 的格式字符串中)。最后,`vfscanf` 将格式化后的输入存放到由格式字符串后的参数给出的地址中。由于多种原因,`vfscanf` 在遇到正常字段结束符(空白字符)之前,可能停止扫描某特定字段或完全终止。参见 `scanf` 对可能原因的讨论。

返回值 返回成功扫描、转换、存储的输入字段数,但扫描而未存储的字段不算在内。如果没有字段被存储,就返回 0。

如果 `vfscanf` 试图读文件末尾,则返回 EOF。

可移植性 适用于 UNIX 系统 V。

参 见 `fscanf`, `scanf`, `va_arg`, `va_end`, `va_start`

示 例

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

FILE *fp;

int vfst(char *fmt, ...)
{
    va_list argptr;
    int cnt;
    va_start(argptr, fmt);
    cnt = vfscanf(fp, fmt, argptr);
    va_end(argptr);
    return(cnt);
}

int main(void)
{
    int inumber = 30;
    float fnumber = 90.0;
    char string[4] = "abc";
    fp = tmpfile();
    if (fp == NULL)
    {
        perror("tmpfile() call");
        exit(1);
    }
    fprintf(fp, "%d %f %s\n", inumber, fnumber, string);
    rewind(fp);
    vfst("%d %f %s", &inumber, &fnumber, string);
    printf("%d %f %s\n", inumber, fnumber, string);
    fclose(fp);
    return 0;
}
```

■ vprintf 送格式化输出到 stdout

用 法 `#include <stdio.h>`

`int vprintf(const char *format, va_list arglist);`

原 型 在 `stdio.h`

说 明 `v...printf` 函数是...`printf` 函数的可选择入口点,其操作与...`printf` 类似,不过,`v...printf` 接受参数表指针而不是接受参数表。

`vprintf` 接受一指向参数表的指针;针对每一参数,由 `format` 所指的格式串中相应包含一个格式指示符,格式化后的输出数据被写入 `stdout`。格式指示符与参数的个数必须相同。

注意:当使用 `SS! =DS` 标志时,`vprintf` 假定传送的地址在 `SS` 段中。

返 回 值 `vprintf` 返回输出的字节数;如果出错,则返回 `EOF`。

可移植性 可用于 UNIX 系统,与 ANSI C 中的定义兼容。

参 见 `printf`, `va_arg`, `va_end`, `va_start`

示 例

```
#include <stdio.h>
#include <stdarg.h>
int vpf(char *fmt, ...)
{
    va_list argptr;
    int cnt;
    va_start(argptr, fmt);
    cnt = vprintf(fmt, argptr);
    va_end(argptr);
    return(cnt);
}

int main(void)
{
    int inumber = 30;
    float fnumber = 90.0;
    char *string = "abc";
    vpf("%d %f %s\n", inumber, fnumber, string);
    return 0;
}
```

■ vscanf 从 stdin 中搜索和格式化输入

用 法 `#include <stdio.h>`

`int vscanf(const char *format, va_list arglist);`

原 型 在 `stdio.h`

说 明 `v...scanf` 函数是...`scanf` 函数可选择入口点,该函数的操作与...`scanf` 类似,不过 `v...scanf` 接受参数表指针而不是接受参数表。

`vscanf` 函数将从 `stdin` 中对输入字段进行正向搜索,每次读一个字符。在搜索过

程中,每个字段先根据格式指示符进行格式化(格式指示符在 format 所指向的传给 vscanf 的格式字符串中)。然后,vscanf 将格式化后的输入保存到由格式字符串后的参数所给出的地址中。格式指示符和地址的个数必须与输入字段的个数相同。

参见 scanf 中有关格式指示符包含信息的描述。

由于多种原因,在遇到正常的字段结束符(空白字符)之前,vscanf 可能停止搜索一特定字段或完全终止。更详细的说明请参见 scanf 对可能原因的讨论。

返回值 vscaf 将返回成功搜索、转换、存储的输入字段数,但搜索而未被存储的字段不包括在内。如果没有任何字段被存储的话,则 vscanf 函数将返回 0。

如果试图用 vscanf 读文件末尾,返回 EOF。

可移植性 适用于 UNIX 系统 V。

参 见 fscanf,scanf,va_arg,va_end,va_start

示 例

```
#include <stdio.h>
#include <conio.h>
#include <stdarg.h>
int vscaf(char *fmt, ...)
{
    va_list argptr;
    int cnt;
    printf("Enter an integer, a float, and a string (e.g. i,f,s)\n");
    va_start(argptr, fmt);
    cnt = vscaf(fmt, argptr);
    va_end(argptr);
    return(cnt);
}
int main(void)
{
    int inumber;
    float fnumber;
    char string[80];
    vscaf("%d, %f, %s", &inumber, &fnumber, string);
    printf("%d %f %s\n", inumber, fnumber, string);
    return 0;
}
```

■ vsprintf 送格式化输出到串中 ■

用 法 #include<stdio.h>

int vsprintf(char *buffer,const char *format,va_list arglist);

原 型 在 stdio.h

说 明 v...printf 函数是...printf 函数的可选择入口点,其操作与...printf 类似,不过,v...printf 接受参数表指针而不是接受参数表。

`vsprintf` 接受一指向参数表的指针; 针对每一参数, 由 `format` 所指的格式串中相应包含一个格式指示符, 格式化后的数据被存入一个串中。格式指示符和参数的个数必须相同。

参见 `printf` 中对格式指示符包含信息的描述。

返回值 `vsprintf` 返回输出的字节数, 如果出错, 则返回 EOF。

可移植性 适用于 UNIX 系统 V, 且与 ANSI C 中的定义兼容。

参见 `printf`, `va_arg`, `va_end`, `va_start`

示例

```
#include <stdio.h>
#include <conio.h>
#include <stdarg.h>
char buffer[80];
int vspvf(char *fmt, ...)
{
    va_list argptr;
    int cnt;
    va_start(argptr, fmt);
    cnt = vsprintf(buffer, fmt, argptr);
    va_end(argptr);
    return(cnt);
}
int main(void)
{
    int inumber = 30;
    float fnumber = 90.0;
    char string[4] = "abc";
    vspvf("%d %f %s", inumber, fnumber, string);
    printf("%s\n", buffer);
    return 0;
}
```

■ vsscanf 从流中搜索和格式化输入 ■

用法 `#include <stdio.h>`

```
int vsscanf(const char *buffer, const char *format, va_list arglist);
```

原型在 `stdio.h`

说明 `v...scanf` 函数是 `...scanf` 函数可选择的入口点, 其操作与 `...scanf` 类似, 不过, `v...scanf` 接受参数表指针而不是接受参数表。

`vsscanf` 从一个流中搜索输入字段, 每次读取一个字符。然后根据格式指示符对每个字段进行格式化(格式指示符在 `format` 所指的传给 `vsscanf` 的格式字符串中), 最后, `vsscanf` 将格式化后的输入值放到由格式字符串后的参数所给出的地址中。格式指示符和地址的个数必须与输入字段个数相同。

参见 `scanf` 中有关格式指示符包含信息的描述。

由于多种原因, `vsscanf` 在遇到正常字段结束符(空白字符)前,可能停止搜索一特定字段或完全终止。参见 `scanf` 中对可能原因的讨论。

返回值 `vsscanf` 返回成功搜索、转换、存储的输入字段数,但搜索而未被存储的字段不算在内。如果没有字段被存储,就返回 0。

如果 `vsscanf` 试图读文件末尾,则返回 EOF。

可移植性 适用于 UNIX 系统 V。

参 见 `fscanf`, `scanf`, `scanf`, `va_arg`, `va_end`, `va_start`, `vscanf`

示 例

```
#include <stdio.h>
#include <conio.h>
#include <stdarg.h>
char buffer[80] = "30 90.0 abc";
int vssf(char *fmt, ...)
{
    va_list argptr;
    int cnt;
    fflush(stdin);
    va_start(argptr, fmt);
    cnt = vsscanf(buffer, fmt, argptr);
    va_end(argptr);
    return(cnt);
}
int main(void)
{
    int inumber;
    float fnumber;
    char string[80];
    vssf("%d %f %s", &inumber, &fnumber, string);
    printf("%d %f %s\n", inumber, fnumber, string);
    return 0;
}
```

■ wherex 给出窗口内光标水平位置 ■

用 法 `#include <conio.h>`

`int wherex(void);`

原 型 在 `conio.h`

说 明 `wherex` 返回当前光标位置(相对于当前文本窗口)的 x 坐标。

返回值 返回 1 到 80 之间的一个整数。

可移植性 仅适用于 IBM PC 及其兼容机。在 Turbo Pascal 中有相应的函数。

参 见 `gettexinfo`, `gotoxy`, `wherey`

示 例 `#include <conio.h>`

`int main(void)`

```

{
    clrscr();
    gotoxy(10,10);
    cprintf("Current location is X: %d Y: %d\r\n", wherex(), wherey());
    getch();
    return 0;
}

```

■ wherey 给出窗口内光标垂直位置 ■

用 法 #include <conio.h>

int wherey(void);

原 型 在 conio.h

说 明 wherey 返回当前光标位置(相当于前文本窗口)的 y 坐标。

返 回 值 返回 1 到 25、43 或 50 之间的一个整数。

可移植性 仅适用于 IBM PC 及其兼容机,在 Turbo Pascal 中有相应的函数。

参 见 gettextinfo, gotxy, wherex

示 例 #include <conio.h>

```

int main(void)
{
    clrscr();
    gotoxy(10,10);
    cprintf("Current location is X: %d Y: %d\r\n", wherex(), wherey());
    getch();
    return 0;
}

```

■ window 创建活动文本模式窗口 ■

用 法 #include <conio.h>

void window(int left, int top, int right, int bottom);

原 型 在 conio.h

说 明 window 在屏幕上创建一个文本窗口。如果所给的坐标无效,则对 window 的调用将被忽略。left 和 top 是窗口左上角的屏幕坐标;right 和 bottom 是窗口右下角的屏幕坐标。

文本窗口的最小尺寸是 1 列宽 1 行高。缺省窗口为全屏幕,其坐标为:

80 列模式:(1,1),(80,25)

40 列模式:(1,1),(40,25)

返 回 值 无。

可移植性 仅适用于 IBM PC 及其兼容机,在 Turbo Pascal 中有相应的函数。

参 见 clrEOF, clrscr, deline, gettextinfo, gotoxy, insline, puttex, textmode

示 例 #include <conio.h>

```

int main(void)
{
    window(10,10,40,11);
    textcolor(BLACK);
    textbackground(WHITE);
    cprintf("This is a test\r\n");
    return 0;
}

```

■ write 写文件 ■

用 法 #include <io.h>

int _write(int handle, void buf, unsigned len);

原 型 在 io.h

说 明 _write 从 buf 所指的缓冲区中写 len 字节到与 handle 相联的文件中。

_write 可写的最大字节数是 65534, 因为 65535 (0xFFFF) 等于 -1, 它刚好是 _write 的错误返回指示。

_write 并不把换行符 (LF) 转换成一个 CR/LF 对, 因为对之操作的所有文件均为二进制文件。

如果实际写入的字节数少于所要求的字节数, 就表示有错误存在, 可能是磁盘空间已满。

对于磁盘文件, 总是从当前文件指针处开始写入; 对于设备, 字节直接传输到设备中。

对于用 O_APPEND 选项打开的文件, _write 在写数据之前并不使文件指针指向 EOF。

返 回 值 _write 返回实际写入的字节数。如果出错, _write 返回 -1, 并置全局变量 errno 为下列值之一:

EACCES 权限错误

EBADF 非法文件号

可移植性 仅适用于 DOS。

参 见 lseek, _read, write

示 例

```

#include <stdio.h>
#include <io.h>
#include <alloc.h>
#include <fcntl.h>
#include <process.h>
#include <sys\stat.h>

int main(void)
{
    void *buf;
    int handle, bytes;

```

```

buf = malloc(200);
/*
   Create a file name TEST. $$$ in the current directory and writes
   200 bytes to it. If TEST. $$$ already exists, it's overwritten.
*/
if ((handle = open("TEST. $$$", O_CREAT | O_WRONLY | O_BINARY,
                  S_IWRITE | S_IREAD)) == -1)
{
    printf("Error Opening File\n");
    exit(1);
}
if ((bytes = _write(handle, buf, 200)) == -1) {
    printf("Write Failed. \n");
    exit(1);
}
printf("_write: %d bytes written. \n", bytes);
return 0;
}

```

■ write 写文件 ■

用 法 #include<io.h>

int write(int handle, void buf, unsigned len);

原 型 在 io.h

说 明 write 将缓冲区中的数据写到与 handle 相联的文件或设备中。handle 是从 creat、open、dup 或 dup2 调用中得到的文件句柄。

该函数从 buf 所指的缓冲区中写 len 个字节到与 handle 相关的文件中。除了 write 用于写一文本文件外,实际写到文件中的字节数不会多于所要求的字节数。

write 可写的最大字节数是 65334,因为 65335(0xFFFF)等于-1,它恰好是 write 的错误返回指示。

对于文本文件,当 write 发现一换行符(LF)时,它输出一个 CR/LF 对。

如果实际写的字节数少于所要求的字节数,就表示有错误,可能是磁盘空间已满。

对于磁盘或磁盘文件,写操作总是从当前文件指针处开始;对于设备,字节被直接传送到设备中。

对于用 O_APPEND 选项打开的文件,write 在写数据以前,总是使文件指针指向 EOF。

返 回 值 write 返回实际写入的字节数。使用 write 写一文本文件时,回车符不算在内。在出现错误时,write 返回-1,并置全局变量 errno 为如下值之一:

EACCES	错误权限
EBADF	非法文件号

可移植性 适用于 UNIX 系统

参 见 creat, lseek, open, read, _write

```
示 例 #include <stdio.h>
        #include <stdlib.h>
        #include <fcntl.h>
        #include <sys\stat.h>
        #include <io.h>
        #include <string.h>
        int main(void)
        {
            int handle;
            char string[40];
            int length, res;
            /*
             * Create a file named "TEST. $$$" in the current directory and write
             * a string to it. If "TEST. $$$" already exists, it will be overwritten.
             */
            if ((handle = open("TEST. $$$", O_WRONLY | O_CREAT | O_TRUNC,
                              S_IRREAD | S_IWRITE)) == -1)
            {
                printf("Error opening file.\n");
                exit(1);
            }
            strcpy(string, "Hello, world!\n");
            length = strlen(string);
            if ((res = write(handle, string, length)) != length)
            {
                printf("Error writing to the file.\n");
                exit(1);
            }
            printf("Wrote %d bytes to the file.\n", res);
            close(handle);
            return 0;
        }
```

第十八章

全局变量

Turbo C 为用户预定义了一些全局变量,以满足许多通常的需要,如时间、命令行参数等等。这里将对这些变量进行定义和描述。

8087 协处理器芯片标志

用法 `extrn int _8087;`

声明在 `dos.h`

说明 如果启动代码自检程序检测到一个浮点协处理器(8087、80287、80387),就将变量 `_8087` 置为非 0 值(1,2,或 3)。否则 `_8087` 置为 0 值。

可通过将环境变量 `87` 置为 YES 或 NO 来越过自动检测逻辑(其命令为 `SET87=YES` 和 `SET 87=NO`,等号前后不能有空格)。在这种情况下, `_8087` 变量将作出响应。

argc 保存命令行的参数个数

用法 `extern int _argc;`

声明在 `dos.h`

说明 `_argc` 是启动程序时传给 `main` 的 `argc` 的值。

argv 命令行参数指针数组

用法 `extern char *_arg[];`

声明在 `dos.h`

说明 `_argv` 指向一个数组,该数组用以存放程序启动时传给 `main` 的初始命令行参数(`argv []`中的元素)。

ctype 字符属性信息数组

用法 `extern char _ctype[];`

声明在 `ctype.h`

说明 `_ctype` 是一个由 ASCII 值+1 索引的存放字符属性信息的数组。每个入口项都是一个字符描述位的集合。

该数组为 `isdigit`、`isprint` 等函数所使用。

■ daylight 指示是否进行夏令时间调整 ■

用 法 `extern int daylight;`

声明在 `time.h`

说明 `daylight` 为日期、时间函数所使用。函数 `tzset`、`ftime` 和 `loadtime` 可将其置为 1(表示夏令时)或 0(表示标准时间)。

■ directvideo 视频输出控制的标志 ■

用 法 `extern int directvideo;`

声明在 `conio.h`

说明 `directvideo` 是控制将用户程序的控制台输出方式(例如来自 `cputs` 函数)。直接输出到视频 RAM(`directvideo=1`)还是通过 ROM BIOS 调用输出(`directvideo=0`)。缺省值为 `directvideo=1`(控制台输出直接到视频 RAM)。为了使 `directvideo=1`,用户的系统视频硬件必须与 IBM 显示适配器相同。置 `directvideo=0` 时允许控制台输出运行于任何与 IBM BIOS 兼容的系统上。

■ environ 存取 DOS 环境变量 ■

用 法 `#extern char *environ[];`

声明在 `dos.h`

说明 `environ` 是一个字符串指针数组,可用来存取和修改 DOS 环境变量。其中每个串的格式是:

`envvar=varvalue`

其中 `envvar` 是环境变量名(如 `PATH`),`varvalue` 是 `envvar` 被设置的字符串值(例如:`C:\BIN`;`C:\DOS`)。串 `varvalue` 可以为空。

当一个程序开始执行时,DOS 环境设置被直接传给该程序。注意,`main` 的第 3 个参数 `env` 等于 `environ` 的初始值。

可通过 `getenv` 函数来读取 `environ`,`putenv` 是唯一可用来增加、修改或删除 `environ` 数组入口项的函数。这是因为修改可能导致进程环境数组大小和位置的变化,但是 `environ` 是自动调整的,总是指向该数组。

参 见 `getenv`,`putenv`

■ errno、_doserrno、sys_errlist、sys_nerr 使 perror 能打印错误信息 ■

用 法 `extern int errno;`

`extern int _doserrno;`

`extern char *sys_errlist[];`

`extern int sys_nerr;`

声明在 `errno.h`,`stdlib.h`(`errno`,`_doserrno`,`sys_errlist`,`sys_nerr`)

`dos.h`(`_doserrno`)

说明 `errno`、`sys_errlist` 和 `sys_neer` 由函数 `peror` 所使用,后者用来打印库子程序预定目标失败时的错误信息。`_doserrno` 是对应于 `errno` 的 DOS 错误代码的变量;然而, `peror` 不直接使用 `_doserrno`。

`_doserrno`:当 DOS 系统调用产生错误时, `_doserrno` 被置成实际的错误代码。`errno` 是从 UNIX 沿袭下来的相同的错误变量。

`errno`:当一数字调用或系统调用出错时, `errno` 被设置以指示错误类型。有时 `errno` 和 `_doserrno` 是相同的;有时 `errno` 不包含实际的 DOS 错误代码(在 `_doserrno` 中包含);还有一些错误在发生时,只是 `errno` 被赋值,而 `_doserrno` 没有变化。

`sys_list`:为提供更多有关错误格式的控制, `sys_list` 提供了错误信息字符串数组。`errno` 可作为数组的下标来查找对应于错误号的字符串。错误信息串中不包含任何换行符。

`sys_neer`:该变量定义为 `sys_list` 中错误信息串的个数。

下表列出了对应于 `sys_list` 错误信息的助记符及其的含义。

助记符	含义
E2BIG	参数表太长
EACCES	权限不对
EBADF	非法文件
ECONTR	内存被破坏
ECURDIR	企图删除当前目录
EDOM	域错误
EEXIST	文件已存在
EFAUL	未知错误
EINVACC	非法存取代码
EINVAL	无效参数
EINVDAT	无效数据
EINVDRV	无效驱动器号
EINVENV	无效环境
EINVFMT	无效格式
EINVFNC	无效功能号
EINVMEM	无效内存地址
EMFILE	打开文件太多
ENMFILE	没有更多文件
ENODEV	无此设备
ENOENT	无此文件或目录
ENOEXEC	运行格式错
ENOFIL	无此文件或目录
ENOMEM	无足够内存
ENOPATH	路径没找到

助记符	含义
ENOTSAM	无相同设备
ERANGE	结果超出范围
EXDEV	交叉设备连接
EZER	错误 0

下页表列出了 `_doserrno` 可以设置的实际 DOS 错误代码及其助记符 (`_doserrno` 的值可以对应或不对应于 `sys_list` 中的错误信息串)。

有关 DOS 错误返回码的更详细信息, 请参见 DOS 参考手册。

示 例

```
#include <errno.h>
#include <stdio.h>
extern char *sys_errlist[];
main()
{
    int i=0;
    while (sys_errlist[i++]
        printf ("%s\n",sys_errlist[i]);
    return 0;
}
```

助记符	DOS 错误码
E2BIG	环境错误
EACCES	无存取权限
EACCES	非法存取
EACCES	当前目录
EBADF	非法句柄
EFAULT	保留
EINVAL	非法数据
EINVAL	非法功能
EMFILE	打开文件太多
ENOENT	无此文件或目录
ENOEXEC	非法格式
ENOMEM	MCB 损坏
ENOMEM	超出内存
ENOMEM	非法块
EXDEV	非法驱动器
EXDEV	无相同设备

■ fmode 设置缺省文件传送模式 ■

用 法 `extern int _fmode;`
 声明在 `fcntl.h`

说明 `_fmode` 决定打开或传送文件的方式(文本或二进制文件)。其缺省值为 `O_TEXT`, 表示文件将以文本方式读。如果将 `_fmode` 置为 `O_BINARY`, 则文件将以二进制方式打开和读取(`O_TEXT` 和 `O_BINARY` 在 `fcntl.h` 中定义)。

在文本方式下, 输入的回车/换行(CR/LF)组合被转换为单个换行符(LF)。输出时相反, LF 被转换为 CR/LF 组合。

在二进制方式下, 不存在这种转换。

可以通过 `fopen`、`fdopen` 和 `freopen` 中的 `type` 参数来改变缺省方式(指定 `t` 为文本方式, `b` 为二进制方式), 也可在 `open` 函数中使用参数 `access` 来指定 `O_BINARY` 或 `O_TEXT`, 以显式定义被打开文件(由 `open` 的参数 `pathname` 指定)的存取方式。

■ `heaplen` 保存近堆的长度

用法 `extern unsigned _heaplen;`

声明在 `dos.h`

说明 `_heaplen` 给出在小模式(微型、小型、中型)下按字节计算的近堆的大小。在大模式(紧缩、大型、巨型)下不存在 `_heaplen`, 因为这种情况下不存在近堆。

在小型和中型模式中, 数据段大小计算如下:

`data segment[small, medium] = global data + heap + stack`

其中堆栈的大小可用 `_stklen` 进行调整。

如果将 `_heaplen` 置为 0, 程序为数据段分配 64K 字节; 有效堆大小为:

`64K - (global data + stack) bytes`

`_heaplen` 的缺省值为 0, 所以如不特别指定 `_heaplen` 为别的值, 用户将得到 64K 的数据段。

在微型模式中, 任何东西(包括代码)均被放在同一段内, 所以数据段的计算调整为包括代码加上为程序段前缀(PSP)分配的 256 字节在内:

`data segment[tiny] = 256 + code + global data + heap + stack`

在微型模式中, 如果 `_heaplen = 0`, 有效的堆大小可通过从 64K 中减去 PSP、代码、全局数据和堆栈来得到。在紧缩型和大型模式中不存在近堆, 堆栈在其自己的段内, 所以数据段可计算为:

`data segment[compact, large] = global data`

在巨型模式中, 堆栈在单独的段内, 每个模块都有其自己的数据段。

参见 `_stklen`

■ `openfd` 存取模式数组

用法 `extern unsigned int _openfd [];`

声明在 `io.h`

说明 `_openfd` 是一个存放文件或设备存取模式的数组。

■ **_osmajor、_osminor、_version** 包含 DOS 版本的主号和次号

用 法 `extern unsigned char _osmajor;`
`extern unsigned char _osminor;`

声 明 在 `dos.g`

说 明 版本的主号和次号可分别通过 `_osmajor` 和 `_osminor` 得到。`_osmajor` 是主号, `_osminor` 是次号。例如,若运行 DOS 3.2,则 `_osmajor` 为 3, `_osminor` 为 20。
 当要编写在 DOS 2.x 版和 DOS 3.x 版中都能运行的模块时,这些变量很有用。一些库函数在不同的 DOS 版本下差别很大,有些库函数只能在 DOS 3.x 下运行(如 `_open`、`creatnew` 和 `ioctl` 函数)。

■ **_psp** 包含当前程序的程序段前缀(PSP)的段地址

用 法 `extern unsigned int _psp;`

声 明 在 `dos.h`

说 明 PSP 为 DOS 进行描述块,其中包含该程序的初始化 DOS 信息。有关 PSP 的更详细信息,请参考《DOS 程序员参考手册》。

■ **_stklen** 保存堆栈的大小

用 法 `extern unsigned _stklen;`

声 明 在 `dos.h`

说 明 `_stklen` 为 6 种存储模式定义堆栈的大小。堆栈最小为 128 个字;若给出的值小于 128 个字节,则将 `_stklen` 自动调整为 128 个字。缺省堆栈大小为 4K。
 在小型和中型模式里,数据段大小计算如下:

`data segment [small,medium]=global data+heap+stack`

其中堆的大小可用 `_heaplen` 来调整。

在微型模式中,任何东西(包括代码)均放在同一段内,所以数据段的计算调整为包括代码加上为程序段前缀(PSP)分配的 256 字节在内:

`data segment [tiny]=256+code+global data+heap+stack`

在紧缩和大型模式中不存在堆,堆栈在其自己的段内,所以数据段可计算为:

`data segment[compact,large]=global data`

在巨型模式中,堆栈在单独的段内,每个模块都有其自己的数据段。

参 见 `_heaplen`

示 例 `#include <stdio.h>`

`/* Set the stack size to the greater than the default.`

`This declaration must go in the global data area. */`

`extern unsigned _stklen=54321U;`

`main()`

`{`

`/* show the current stack length */`

```
printf ("The stack length is: %u\n", _stklen);  
return 0;  
}
```

■ **timezone** 包含当地时间与格林威治时间(GMT)之间的差值(以秒为单位) ■

用 法 extern long timezone;

声 明 在 time.h

说 明 timezone 由时间和日期函数所使用。

该变量由 tzset 函数计算,它被赋一长整型值,表示当地时间与格林威治时间之间以秒为单位的差值。

■ **tzname** 时区名指针数组 ■

用 法 extern char *tzname [2];

声 明 在 time.h

说 明 全局变量 tzname 是由字符串(包含着时区名缩写)指针组成的数组。tzname[0] 指向一个 3 字符的字符串,串中包含从 TZ 环境串中获得的时区名。tzname[1] 也指向一个三字符字符串,串中包含从 TZ 环境串中获得的夏令时区名。如果没有使用夏令时,则 tzname [1]为 NULL。

■ **_version** DOS 版本号 ■

用 法 extern unsigned int _version;

声 明 在 dos.h

说 明 _version 中包含 DOS 版本号,主号在低字节中,次号在高字节中。(对 DOS_{x.y} 来说,_x 是主号,_y 是次号)

■ **wscroll** 指示控制台 I/O 函数是否滚屏 ■

用 法 extern int _wscroll;

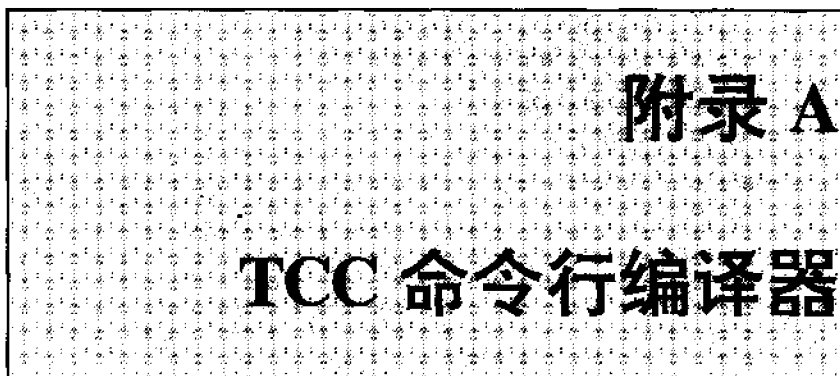
声 明 在 conio.h

说 明 _wscroll 是一个控制台 I/O 标志,其缺省值为 1。如果置 _wscroll 为 0,则不能滚屏,这种情况可用于沿窗口边缘画线时屏幕不滚动。

第五部分

附 录

**TCC 命令行编译器
实用程序**



作为使用集成环境的另一种方法,用户可以通过命令行来编译和运行自己的程序,也就是使用命令行编译器。命令行编译器不只是编译用户的文件,实际上它是集成环境的命令行版本。几乎所有集成环境的功能命令行编译器均能完成。用户可以设置或取消警告,使用或不使用 EMS、使用实模式或保护模式、调用 TASM 或其它的汇编程序来汇编 ASM 源程序等等。事实上,如果只是编译,可以在命令行上使用 -C 选项。

如果 IDE 不能很好地符合用户开发和运行程序,用户就可以使用命令行编译器。

本附录分为两个部分。第一部分讲述如何使用命令行编译器,并提供了一张包含所有选项的汇总表。第二部分根据各选项的功能,分类介绍了各组相关的选项。

A.1 使用命令行编译器

命令行编译器可以让用户从 DOS 命令行调用集成环境编译器的所有功能。

要从命令行调用 Turbo C,在 DOS 提示符下键入 TCC,将会出现提示,并在其后有一组命令行参数。命令行参数包括编译程序及连接程序选择项以及文件名。一般命令格式为:

```
tcc [option [option] ] filename [filename ...]
```

例如,每一个命令行选择项前都有一个连字符(-),并与 TCC 命令和其它选择项分开,用至少一个空格与后面的文件名分开。用户还能使用一个配置文件。

A.1.1 使用选择项

这些选择项可分为三种一般的类型:

- 编译程序选择项。
- 连接程序选择项。
- 环境选择项。

要从屏幕上查看命令行编译程序选择项列表时,在 DOS 提示符下键入:TCC(当用户位于 BIN 目录或当 TCC 在用户的 DOS 路径中),然后按 Enter。在表 A.1 中列出的所有的选项。

要选择一个命令行选项,须键入连字符(-)后紧跟着选项字符(例如-l)。若要关闭某个选项,则在该选项字符后再加一个连字符。对所有的触发切换选项(toggle option)即可以打开或关闭的选项而言,选择字符后面加一个连字符(-)则关闭该选项,加一个加号(+)或什么也不加,则打开此选项。比如,-C 和 -C+都将打开嵌套注释,而 -C- 则关闭嵌套注

释。

利用这个特性来废弃配置文件中的设置。

A.1.1.1 选项优先级的规则

选项优先级的规则十分简单,命令行选项先左后右,并且遵循下列原则:

- 对于所有除-l或-L选项而言,右边的选项将废弃左边的相同的选项(因为右边的关闭选项将会取消左边的打开选项)。
- 对于-l或-L而言,左边的优先级将高于右边。

表 A.1 命令行选项汇总表

选择项	功能
@filename	响应文件
+filename	告诉 TCC 使用另外一个配置文件 filename
-l	生成 80186 指令
-l-	生成 8088/8086 指令
-A	仅用 ANSI 关键字
-a	字对齐
-a-	字节对齐(缺省)
-B	编译时调用汇编程序处理内部代码
-C	允许注释嵌套(*)
-c	编译到.OBJ中但不连接
-Dname	为含有空字符的串定义
-Dname=string	为串定义名
-d	置合并相同串为 on
-d-	置合并相同串为 off(缺省)
-Efilename	用 filename 作为要使用的汇编程序
-efilename	连接产生 filename.EXE
-f	仿浮点(缺省)操作
-f-	不做浮点操作
-f87	8087 硬件指令
-G	速度优化
-G-	代码大小优化
-gn	警告:n条信息之后停止
-Ipathname	包含文件目录
-in	设有效标识符长度为 n
-jn-	错误:n条信息之后停止
-K	缺省字符类型 Unsigned
-K-	缺省字符类型 signed(缺省)
-k	置标准堆栈框架(缺省)
-Lpathname	库文件目录
-lx	将选项 X 传到连接程序中(可有多多个 X)
-M	使连接程序建立一个映像文件
-mc	用 compact 存储方式编译
-mh	用 huge 存储方式编译

续表 A. 1

选择项	功能
-ml	用 large 方式编译
-mm	用 medium 存储方式编译
-ms	用 small 存储方式编译;缺省
-mt	用 tiny 存储方式编译;
-N	检查堆栈溢出
-npathname	输出目录
-O	转移优化
-O-	不优化(缺省)
-ofilename	把源文件编译到 filename.obj 中
-p	用 Pascal 调用约定
-p-	用 C 调用约定(缺省)
-r	设寄存器变量为 ON(缺省)
-r-	禁止使用寄存器变量
-S	产生 ASM 输出文件
-Uname	撤销任何原来的 name 定义
-u	产生下划线(缺省)
-v, -v-	设源程序调试信息为 On 或 Off
-w	置显示警告为 On
-w-	置显示警告为 off
-wxxx	使 xxx 警告信号有效
-w-xxx	使 xxx 警告信号无效
-X	使编译程序自动依赖输出无效
-Y	使覆盖码生成有效
-y	置行号为 On
-Z	使寄存器优化有效
-zAname	Code 类
-zBname	Bss 类
-ZCname	Code 段
-zDname--zEname	Bss 段
-zGname	Bss 组
-zPname	Code 组
-zRname	Data 段
-zSname	Data 组
-zTname	Data 类

A. 1.2 语法和文件名

Turbo C 根据下面的规则编译程序:

filename.asm	调用 TASM 以汇编到.OBJ 中
filename.obj	在连接时作为目标文件

filename.lib	在连接时作为库文件
filename	编译 FILENAME.C
filename.c	编译 FILENAME.C
filename.xyz	编译 FILENAME.XYZ

例如,有如下的命令行:

```
tcc -a -f -C -O -Z -emyexe oldfile1 oldfile2 nextfile
```

Turbo C 编译器把 OLDFILE1.C、OLDFILE2.C 和 NEXTFILE.C 编译成 .OBJ,并且连接生成一个名叫 MYEXE.EXE 的可执行文件,用字对齐(-a),仿浮点(-f),可嵌套注释行(-C),进行优化(-O)以及允许寄存器优化(-Z)。

若在命令行给出 .ASM 文件或者若有一个 .C 文件含有内部汇编代码,则将调用 TASM。TASM 的选择项为:

```
/D __MODEL __/D __Lang __/ml /fp
```

这里,MODEL 是下列各项之一: TINY、SMALL、MEDIUM、COMPACT、LARGE 或 HUGE。/ml 则告诉 TASM 汇编时要把大小写区别开。Lang 是 CDECL 或 PASCAL;当指定 -f87 时,fp 为 r,否则为 e。

A. 1.3 应答文件

如果用户必须命令行指定许多选项或文件名,那么可以将它们放在一个 ASCII 文本文件中。这个文件叫做应答文件(当然,也可以称做其它用户喜欢的名称)。用户可以让命令行编译器从这个文件读它的命令行,只要在这个文件名前置以 @。用户可以任意指定多个这样的文件,并且可以随意地将它们与其它选项文件名混合在一起。

例如,假设 MOON.RSP 文件包含 STARS.C 和 RAIN.C。下面这条命令

```
TCC SUN @MOON.RSP ANYONE
```

将使 Turbo C 在实模式下编译 SUN.C、START.C、RAIN.C 和 ANYONE.C,它可以扩展为

```
TCC SUN.C STARS.C RAIN.C ANYONE.C
```

相关文件允许用户使用长度超出 DOS 通常允许长度的命令字符串。

A. 1.4 配置文件

如果用户发现经常使用某些选项,就可以将它们列在配置文件中。这个配置文件的缺省名为 TURBOC.CFG。当运行 TCC 时,它会自动在当前目录下寻找 TURBOC.CFG。如果在当前目录下找不到这个文件,而此时运行的是 DOS3.0 或更高版,那么 Turbo C 将会在启动目录(即 TCC.EXE 文件所在目录)下寻找。

用户可以建立几个配置文件,每个必须有不同的名字,只要在 TCC 命令行的任何地方,在欲指定的配置文件名前加一“+”号,用户便指定了该配置文件。例如,用户要从 D 盘的 D:\ALT.CFG 文件中读取选项的设置,只要使用如下命令:

```
TCC +D:\ALT.CFG ...
```

除了在命令行上使用选项以外,用户还可以使用配置文件,甚至用配置文件来代替它们。如果不想使用配置文件中的某些选项,用户可以在命令行上使用相应的选项来废弃它们。

们。

用户也可以使用任何 ASCII 编辑器或字处理软件,比如 Turbo C 集成编辑器,来建立 TURBOC.CFG 文件(若无其他可代替的配置文件)。用户可以把选项用空格分开列在同一行上,或者将它们分行列出。

TURBOC.CFG 与 TCCONFIG.TC 不一样,后者是缺省的集成环境配置文件。

下面,讨论选项优先级规则

总的来说,用户可记住命令行上的选项会废弃配置文件中指定的相同的选项。命令行选项能够废弃配置文件中的选项是很重要的。例如,如果用户的配置文件中有几个选项,其中包括 -a(现在想关掉这个开关),此时仍可使用配置文件,只需在命令行选项中列出 -a-,便可覆盖 -a,当然,优先级的规定比这个还要稍微详细一些。详见本附录“使用选择项”小节中的“选择项优先级规则”。此外,还有如下规定:

1. 当配置文件中的选项与命令行选项结合在一起时。配置文件中的任何 -L 和 -l 选项将被附加到命令行选项的右端。也就是说,Turbo C 首先寻找的是命令行指定的嵌入文件和库目录(因此,命令行的 -l 和 -L 选项比配置文件中的优先级要高)。
2. 配置文件中其余选项直接插入 TCC 命令后(在所有命令行选项的左边)。这样就使命令行选项的优先级高于配置文件中的选项。

A.2 编译器选项

Turbo C 命令行编译器选项分为八组:

1. 内存模式选项:告诉 Turbo C 该用何种内存模式来编译用户程序。
2. 宏定义:在命令行定义和取消宏。
3. 代码生成选项:控制生成代码的特性,例如浮点选项、调用约定、字符类型、CPU 指令等。
4. 优化选项:用来指定目标代码如何优化,按长度还是按速度,使用或不使用寄存器变量,使用或不使用别名等。
5. 源代码选项:使编译器认识(或忽略)源代码的一些特征,如使用特定的关键字(非 ANSI,非 K&R,非 UNIX)、嵌套注释以及标识符长度等。
6. 错误报告选项:确定编译器将报告哪些警告信息,以及编译结束前可出现的警告和错误的最大数目。
7. 段命名选项:允许重新命名段,重新分配组和类。
8. 编译控制选项:使编译器
 - 编译成汇编代码(而不是目标模块)。
 - 编译含有直接插入汇编代码的文件(虽然还有其它方法:使用 #pragma inline 指令或忽略它)。
 - 只编译不连接。

A.2.1 存储模式

存储模式选项用来告诉编译器编译用户的程序时使用何种存储模式。存储模式有如下

几种:极小(tiny)模式,小(small)模式,中(medium)模式,紧凑(compact)模式,大(large)模式和超大(huge)模式。

- mc 编译时使用紧凑模式
- mh 编译时使用超大模式
- ml 编译时使用大模式
- mm 编译时使用中模式
- ms 编译时使用小模式(缺省)
- mt 编译时使用极小模式

A. 2. 2 宏定义

宏定义选项可使用户在命令行定义或取消宏(也称为 manifest 或 symbolic 常量),默认的定义值为空串。命令行上定义的宏可以废弃(取代)源程序中的宏。

- Dname 将名为 name 的标识符定义为空串
- Dname=string 将名为 name 的标识符定义为=后面的字符串,该字符串不能包含空格或制表符。
- Uname 废除先前对名为 name 的标识符的定义

宏定义的方式有:

- 用一个-D选项定义多个符号,两两间用分号(;)隔开(即所谓成套选项):
TCC -Dxxx;yyy=1;zzz=No MYFILE.C
- 多个-D选项,每个-D定义一个符号:
TCC -Dxxx -Dyyy=1;-Dzzz=No MYFILE.C
- 可以混合使用这两种方式:
TCC -Dxxx -Dyyy=1;zzz=No MYFILE.C

A. 2. 3 代码生成选项

代码生成选项控制生成代码的特性,如浮点选项、调用约定、字符类型及 CPU 指令等。

- 1 使 Turbo C 生成扩展的 80186 指令;在 IBM PC/AT 上以 DOS 的实模式方式运行也能生成 80286 指令。
- 1— 使编译器生成 8088/8086 指令。
- a 使整型或更大长度的数据按机器字边界对齐。在结构中加入附加字节使结构成员对齐,正确对齐自动变量和全局变量。char 和 unsigned char 类型的变量和域可赋以任意地址,其它变量和域分配偶地址。在默认情况下关闭该选项,按字节对齐。
- b 指示编译器总是为枚举类型分配整字,该选项是缺省值(这是 Turbo C2.0 处理枚举类型的方式)。
- d 编译器合并重复字符串,产生较短的程序,缺省情况是关闭该选项(—d—)。
- f 若运行时间系统没有 8087,在运行时的仿真 8087。若有 8087,则调用

- 8087 进行浮点运算(缺省)。
- f- 指出程序中不含浮点计算,在链接时浮点库将不被连接进去。
 - f87 用内部 80x87 指令而不是用 80X87 仿真库子程序调用来产生浮点操作。指明在运行时的一个数学协处理器将可用。用这一选择项编译生成的程序不能在一个没有数学协处理器的机器上运行。
 - K 使编译程序把 char 说明作为 unsigned char 类型对待,这使得与其它把 char 说明作为 unsigned 对待的编译程序兼容,缺省时 char 说明为 signed。
 - k 产生一个标准堆栈框架,当用一个调试程序对子程序调用进行回溯时,该项十分有用,缺省时为 on。
 - N 对每一个函数项进行堆栈溢出检查,当栈溢出被检测到时,这些函数将生成栈溢出信息。这将花费程序的空间和时间,把它作为一个选择项是因为栈溢出是很难检测的。若被检测到,将打印“Stack Overflow!”信息,并且程序退出。
 - P 使编译程序用 Pascal 参数传递序列生成所有的子程序和函数调用。所生成的函数调用将会又小又快。函数必须要有正确的参数个数以及参数类型,而不像常规的 C,允许可变的函数参数。用户可以用 cdecl 语句重载这一选项并且特别说明函数为 C 类型。
 - U Turbo C 使用该项把将要说明的标识符存放在目标模块之前,在其前面加上下划线(_).

Turbo C 对待 Pascal 类标识符(它们被 Pascal 关键字修改)却不同标识符用大写字母书写,并且前面没有下划线(_).

标识符前是否放置下划线是可选的,缺省时为 On。用户可用 -u- 将之关闭,但是用的是标准 Turbo C 库文件,那么就会遇到一些问题,除非用户重建了该库文件(要重建,需要 Turbo C 运行时间库源码)。

- y 把行号放在目标文件中以便符号调试程序使用。这将增加目标文件的大小,不影响运行程序的大小和速度。该项只有与能够使用该信息的符号调试程序协调时才有用。总的来说用 Turbo Debugger, -v 比 -y 有用。
- v 告诉编译程序把调试信息放在 .OBJ 文件中,以便正在编译的文件能够用 Turbo C 集成调试程序或独立 Turbo Debugger 进行调试。编译程序还把此项传递给连接程序以便它把调试信息放到 .EXE 文件中。为方便调试,该项还使 C++ 内部函数被当作一般函数对待。

因此,若想打开调试并打开内部扩展,则必须使用两个开关:

-v -vi

A. 2.4 优化选择项

优化选择项让用户指定如何优化目标代码;是侧重于长度还是侧重于速度,要不要使用寄存器变量以及是否要用别名。

- G 使编译程序侧重于代码执行的速度优化而不是代码大小的优化。
- G— 在默认情况下,优化侧重于代码长度。
- O 进行优化。该优化删除冗余的转移(例如从一条转移语句转到另一条转移语句)以及转移到同一位置的相同代码的重复。它还压缩冗余的寄存器装入。当—Z 被关闭时,这不会影响到程序的动作(当然,除非该代码很有效)。
- O— 不优化,以最快的速度进行编译,产生最坏的目标码。
- r 可以使用寄存器变量(缺省)。
- r— 压缩寄存器变量的使用,当用户用—r—项时,编译程序不使用寄存器变量,并且不保护和恢复从调用程序而来的寄存器变量(SI,DI)。由于这个原因,用户不会得到用—r—编译而来的使用寄存器调用码的目标代码。

从另一方面,当用户正在与存在的未保存 SI、DI 的汇编语言代码交互时,—r—项允许从 Turbo C 中调用该代码。

- Z 该项使编译程序假定变量既不能被直接访问又不能通过同一函数中的指针进行访问。该项只有与—O 一起使用才有效。编译程序保存着一张反映当前寄存器内容的表。如果一个变量需要从存储器中装一个寄存器,编译程序记住寄存器中存放的变量。当使用变量时,编译程序就使用该变量对应的寄存器而不是使用其在存储器中的那个值。
—Z 项决定编译程序如何控制非直接赋值的,通常这样的赋值被认为能够潜在地改变一个变量。因而不得不忘掉所有在寄存器中的变量(即删除该寄存器表)。—Z 告诉编译程序,非直接赋值将不改变变量,因而保留寄存器中的变量是安全的。

若用户既直接访问了一个变量又通过在同一函数中的一个指针访问了它,那么—Z 将产生一些错误码,因而用起来不安全。从另一个角度讲,它将产生速度稍快的代码。

A. 2.5 源代码选项

源代码选项使编译器识别(或忽略)源代码的某些特性;采用特别关键字(非 ANSI、非 K&R、非 UNIX),允许注释嵌套,指定标识符有效长度等。如果用户打算把自己的程序移植到其它系统中去,这些选项是很重要的。

- A 编译生成与 ANSI 兼容的代码;任何 Turbo C 的扩展关键字都被忽略并可当作一般的标识符。这些关键字包括:
_CS _SS far near _ds asm huge Pasc al _es cdecl interrupt
以及寄存器伪变量,如 _AX, _BX, _SI 等。
- A— 用 Turbo C 关键字。
- C 允许注释行嵌套。注释行一般不能进行嵌套。
- in 使编译器仅识别标识符的前几个字符,所有标识符,不管是变量、预处理宏定义名还是结构成员,如果它们的前几个字符不同,则认为是不同的标识符。

缺省情况下, Turbo C 允许标识符使用 32 个字符。其它系统, 如 UNIX 只识别前 8 个字符, 如果要把它们移植到别的环境中去, 编译代码时就希望能使标识符具有较少数目的有效字符, 因此, 用该方式编译, 有助于弄清较长的标识符被截短成一个有效长度的标识符时, 是否会发生冲突。

A. 2.6 出错报告选择项

出错报告选项让用户指示编译器报告什么类型的警告信息、产生多少个警告信息和出错信息后停止编译。

星号(*)表示相应选择项在缺省时为 on, 而其它选择项在缺省时为 off。

- gn 产生 n 个警告信息后停止编译。
- jn 产生 n 个出错信息后停止编译。
- W 此选项将使编译的警告信息显示出来。可以用 -W 来关闭它, 也可以用 -Wxxx 来显示特殊的警告信息或者禁止显示。
- Wxxx 使由 xxx 指定的警告信息有效。-W-xxx 选择项略去由 xxx 指定的警告信息。在这里列出一 Wxxx 可能的选择项, 并把它们分成四个类型: ANSI 禁则、经常错误、可移植性警告和警告, 用户可以用源码中 pragmaxwarn 来控制这些选择项。参见《程序员手册》“出错信息”。

ANSI 禁则

- wbbf 位域必须是 signed 或 unsigned int。
- wbei * 用不适当的类型进行初始化。
- wbfs * 假设为 signed int 无类型位域。
- wbig * 十六进制值含有三个以上数字。
- wdcl * 说明没有指定一个标号或一个标识符。
- wdpu * 原型中函数说明先于使用。
- wdup * 宏的特征定义不一致。
- weas 将 integer-val 赋给 enumeration。
- wext * 标识符被说明成既是外部的又是静态的。
- will pragma 指令格式不正确。
- wpin 初始化仅有部分被括起来。
- wret * 同时使用返回和返回一个值。
- wstr * 函数不能是结构或联合的域。
- wstu * 未定义结构 structure。
- wsus * 可疑的指针转换。
- wvoi * void 函数不能返回一个值。
- wzdi * 被零除。
- wzst * 零长度的结构。

经常错误

- waus * 标识符被赋予一个从不使用的值。
- wdef * 标识符在定义前可被使用。

- weff * 代码无效。
- wpar * 参数 parameter 从未使用。
- wpia * 可能不正确赋值。
- wrch * 不可能执行到的代码。
- wrvl * 函数应返回一个值。
- wamb 二义操作符需要括号。
- wamp 函数代码数组中有多余的 & 号。
- wnod 函数 function 未说明。
- wpro 调用没有原型的函数。
- wstv 以值传送结构。
- wuse 未曾使用已说明的标识符。

可移植性警告

- wapt * 不可移植性指针赋值。
- wcln 常量是 long。
- wcpt * 不可移植性指针比较。
- wrng * 比较中的常数越界。
- wrpt * 不可移植性返回类型转换。
- wsig 转换可能丢掉有效数字。
- wvcp 把指针与 signed 及 unsigned char 混淆。

A. 2.7 段命名控制

段命名控制选项让用户重新命名段、重新分配其组和类。

- zAname 把代码段类名改为 name。缺省情况代码段被赋以类 CODE。
- zBname 未初始化数据段类改名为 name, 缺省情形, 未初始化数据段被赋以类 BSS。
- zCname 把代码段改名为 name。缺省情况, 代码段命名为 _TEXT, 在中、大和特大模式下, 代码段命令为 filename _TEXT (其中 filename 是源文件名)。
- zDname 把未初始化数据段改名为 name。在缺省情况下, 除了特大模式外 (不生成未初始化段), 未初始化数据段命名为 _BSS。
- zGname 把未初始化数据段组改名为 name, 缺省情况, 该数据组命名为 DGROU, 在特大模式中没有数据组。
- zPname 为代码段 name 生成带代码组的输出文件。
- zRname 把初始化数据段命名为 name。缺省情况下, 把初始化数据段命名为 DATA; 在特大模式, 命名为 filename _DATA。
- zSname 把初始化数据段组改名为 name。缺省情况, 该数据组命名为 Dakoup, 在特大模式中没有数据组。
- zTname 把初始化数据段类命名为 name。缺省情况, 初始化数据段类命名为 DATA。

A. 2.8 编译控制选择项

编译控制选项控制源文件的编译。

- B 编译时调用汇编程序处理内部汇编码。
- C 编译和汇编名为. C 和. ASM 的文件,但不执行连接命令。
- Efilename 把 name 作为要用的汇编程序名。缺省时,用 TASM。
- S 编译源文件并产生汇编语言输出文件(. ASM),但不进行汇编,当选用该项时,Turbo C 将把 C 源行作为注释放在所生成的. ASM 文件中。在集成环境中该选项是不可用的。

A. 4 环境选项

使用环境选项时要记住,Turbo C 只识别两种类型的库文件:隐含库和用户指定的库。参见本附录“文件搜索算法”小节。

- Ipathname 编译器除了在标准路径中查找外还将在 path(驱动器或子目录名)中查找包含文件。注意,I 是大写字母,驱动器名是单个的字符(大写或小写),后接一个冒号(:)。目录可以为任何合法的目录或目录路径名。可以使用多个—I 目录选项。
- Lpathname 指示连接程序在给出的目录中查找 COX. OBJ 作为启动目标文件及库文件(X. LIB CPX. LIB, MATHX. LIB, EMU. LIB 和 FP87. LIB)缺省情况,连接程序将在当前目录中查找。
- npathname 指示把编译产生的. OBJ 和. ASM 文件放入由 pathname 指定的路径目录中。

A. 4.1 查找包含文件和库文件

Turbo C 能在多个目录中查找嵌入文件和库文件。这就意味着,像 #define 选择项(—D)那样,库文件目录(—L)和包含文件目录(—I)命令行选择项的语法允许给定选择项的多重列表。

以下是两个选项的句法:

库目录: —Ldirname [,dirname, ...]

包含文件目录: —Idirname [,dirname, ...]

其中 dirname 可以是任意的目录或目录路径名。

使用这两个选项的方式有:

- 用单个—I, —L 选项,列出多个目录,两两间用分号隔开,如

```
tcc -Ldirname1;dirname2;dirname3 -lincl;inc2;inc3,myfile.c
```

- 多个—I, —L 选项,每次列出一个目录,如:

```
tcc -Ldirname1 -Ldirname2 -Ldirname3 -lincl -linc2 -linc3 myfile.c
```

- 混合方式,如

```
tcc -Ldirname1;dirname2 -Ldirname3 -lincl;inc2 -linc3 myfile.c
```

当命令行列出多个-L, -I 选项或者有多个目录名时, 编译器将查找所有的目录, 顺序是从左到右。

注意: 集成环境也支持多个库目录查找。

A. 4.2 文件搜索算法

Turbo C 包含文件搜索算法, 以下列方法搜索源文件代码中所列的 #include 文件:

- 若在源码中有一个 #include <somefile. h> 语句, 将仅在指定包含目录中搜索 somefile. h。
- 另一方面, 若源码中有 #include "somefile. h" 语句, Turbo C 将首先在当前目录中搜索 somefile. h, 若当前目录中找不到头文件, 则将在命令行所指定的包含目录中搜索。库文件的搜索算法与包含文件的相似。
- 隐式库文件: Turbo C 仅在所指定的库文件目录中搜索隐式库文件, 这类似于 #include <somefile. h> 的搜索算法。
- 显式库文件: Turbo C 将在何处搜索显式(用户指定)库文件部分决定于用户是如何列出库文件名的:
- 若用户列出的显示文件名中没有指明驱动器或目录(如: mylib. lib), Turbo C 首先将在当前目录中进行搜索。若不成功, 它将在指定的库文件目录中进行搜索。这类似于 #include "somefile. h" 的搜索算法。
- 若用户列出的显示文件名中有驱动器号和目录(如: C:\mystuff\mylib. lib), Turbo C 将仅在用户所列出的地方进行搜索, 而不在指定的库文件目录中进行搜索。

A. 4.3 一个实例

这里是一个结合使用多重库文件目录(-L)和包含文件目录(-I)选择项的 TCC 命令行的应用实例:

- (1) 当前驱动器是 C:, 当前目录为 C:\TURBOC, TCC. EXE 在当前目录中, A 驱动器的当前位置是 A:\ASTROLIB。
- (2) 包含文件(. H 或“hender”文件)位于 C:\TURBOC\INCLUDE 中。
- (3) 启动文件(C0T. OBJ, C0S. OBJ...C0H. OBJ)在 C:\TURBOC 中。
- (4) 标准 Turbo C 库文件(CS. LIB, CM. LIB, ..., MATHS. LIB, MATHM. LIA..., MMU. LIB, FP87. LIB 等等)在 C:\TUROC\LIB 中。
- (5) 行星系统用户库文件(用 TLIB 建立和管理)在 C:\TURBOC\STARLIB 中。其中之一为 PARX. LIB
- (6) 类星体的第三部分库文件在 A 驱动器中的\ASTROLIB 目录中, 其中之一的文件名为 WARP. LIB。

在上述情形上, 建入下列 TCC 命令行:

```
tcc -mm -Llib;strlib -Iinclude orion umaj parx. lib a:\astrolib\warp. lib
```

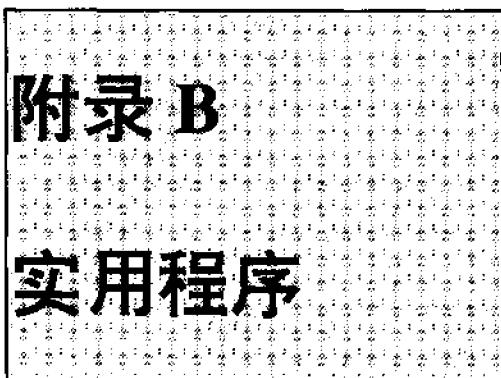
TCC 将编译 ORION. C 和 UMAJ. C 到. OBJ 文件中。编译程序搜索源码包含文件所在的目录 C:\TURBOC\INCLUDE, 然后将它们连接: 中型模式启动码(C0M. OBJ)、中型模式

库文件 (CM. LIB, MATH. LIB)、标准仿浮点库文件 (EMU. LIB) 以及用户指定库文件 (PARX. LIB 和 WARP. LIB), 这样生成一个叫 ORION. EXE 的可执行文件。

在 C:\TURBOC 中搜索启动码 (由于搜索到, 故而停止搜索); 在 C:\TURBOC\LIB 中搜索标准库文件 (由于搜索成功, 故停止搜索)。

为搜索用户指定库文件 PARX. LIB, 编译程序先在当前目录 C:\TURBOC 中搜索, 没有成功, 编译程序将以如下顺序搜索库文件目录: C:\TURBOC\LIB, C:\TURBOC\STARLIB (其中有 PARX. LIB)。

由于库文件 WARP. LIB 具有显式路径 (A:\ASTROLIB\WARP. LIB), 编译程序仅在该路径中进行搜索。



B.1 MAKE 实用程序

MAKE 是从同名的 UNIX 程序中引出的 Borland 命令行命令,帮助用户来维护正在编制的程序。多数程序中都包括许多源文件,这些程序同其他的程序连接之前,程序中的源文件可能要通过预处理、汇编、编译和其他程序的执行。当用户已修改过当前模块或与模块相关的某些程序后,忘记了再编译将可能导致错误的发生。另一方面,虽然重新编译一遍所有的文件是非常安全可靠的,但却非常浪费时间。

MAKE 可以解决这个问题。用 MAKE 时,给出如何处理程序的源文件和目标文件以产生最终结果的描述信息。MAKE 首先检查文件上的描述信息和标识日期部分,然后做一些工作,生成一最新版本的文件。在处理过程中,MAKE 调用了许多不同的编译程序、汇编程序、连接程序和其它实用程序,但对已完成了的未被修改的文件不做任何处理。

MAKE 的用途不仅在于程序设计过程中,用户还可以使用 MAKE 来控制那些选择程序和处理程序,以生成最终结果的过程。通常的一些应用有:文本处理、自动备份、为扩展到其它目录中去的文件排序,并将文件从当前目录中清除。

B.1.1 MAKE 的工作过程

MAKE 通过执行如下的工作来保持用户的设计程序版本是最新的:

- 读一个文件名为 makefile 的特定文件,这个文件是用户已经建好的。MAKE 通过这个文件了解到哪些源文件和头文件必须被编译生成 OBJ 文件,哪些 OBJ 文件必须被连接。
- 检查每个 OBJ 文件的日期和时间以及相应源文件和头文件的日期和时间。如果源文件和头文件中的某些文件的时间比它们的 OBJ 文件的时间更新,MAKE 就知道这些文件已被修改,需要对这些修改过的文件重新编译。
- 调用编译程序编译源文件。
- 将每个 OBJ 文件的日期和时间与可执行文件的日期和时间比照。
- 若有 OBJ 晚于 EXE 文件,则调用连接程序重建 EXE 文件。

注意:MAKE 完全依赖于 DOS 为每一个文件设置的时间。也就是说,要用 MAKE 完成这些功能,必须正确地设置系统的时间和日期。若用户用的是 AT 或 PS/2,要保证电源是正常工作的,否则将会使系统时钟不能准确计时,从而 MAKE 不能正常工作。最初的 IBMPC

和大多数兼容机都没有嵌入系统的时钟。若用户的系统也是这样并且没有加入时钟的话,在每次开机时,一定要正确地设置好系统时间和日期(用 DOS 的 DATE 和 TIME 命令)。

B.1.2 启动 MAKES

要使用 MAKE,请在 DOS 提示符下键入 MAKE,MAKE 将自动找到名为 MAKEFILE 的文件。如果 MAKE 没有发现该文件,它将继续查找 MAKEFILE.MAK 文件;如果还找不到 MAKEFILE.MAK 或 BUILTINS.MAK(将在后面介绍),MAKE 将停止执行,并显示出错信息。

如若用户不想使用 MAKEFILE 或 MAKEFILE.MAK 文件而用别的.MAK 文件,那只要在 MAKE 命令后面给出 file(-f)选项即可,如下所示:

```
MAKE -f MYFILE.MAK
```

MAKE 的一般句法是:

```
make [option[option]][target[target]]
```

这里 option 是 MAKE 的选项(以后讨论),target 是 MAKE 的目标文件名。

以下是 MAKE 句法的规则:

- Make 后面需一空格,接着是 make 的选项。
- 相邻的选项之间必须空一格,且选项可以按任何顺序排列。可以键入许多选项(只要命令行有空间)。所有的选项不能定义为一字符串,在选项的后面有一选择符“-”或“+”。这决定是关闭此选项(-)还是打开此选项(+).
- MAKE 所有选项结束后,空一格才是可选择的目标文件名。
- 相邻的目标文件名之间必须空一格。MAKE 按所列的顺序来整理目标文件,有必要重新编制它们的组成。

如果命令行中不包括任何目标文件名,MAKE 将使用显式规则中提出的第一个目标文件名。如果在命令行中有一个或多个目标文件名,MAKE 将创建它们。

B.1.2.1 BUILTINS.MAKE 文件

用户将经常发现有许多 MAKE 宏和规则被反复使用,有三种方法可以处理它们。

- 第一,用户可以将它们放在所创建的 makefile 文件里。
- 第二,用户可以放它们在一个文件里,在 makefile 文件里使用 #include 指令,将该文件包括进 makefile 文件里。
- 第三,可以将它们放在一个 BUILTINS.MAK 文件里。每次运行 MAKE,它都要查找 BUILTINS.MAK 文件。一般没有必要存在几个 BUILTINS.MAK 文件。如果 MAKE 找到了一个 BUILTINS.MAK 文件,它首先解释该文件,如果它找不到 BUILTINS.MAK 文件,它就直接开始解释 MAKEFILE 文件(或者其它用户定义的 makefile 文件)。

MAKE 首先在当前目录下查找 BUILTINS.MAK 文件。如果没有该文件,MAKE 接着到 MAKE.EXE 所在的目录下寻找该文件。所以,用户应该将 BUILTINS.MAK 和 MAKE.EXE 放在同一目录下。

MAKE 总是在当前目录下寻找 makefile 文件,该文件中包含有建立某一可执行程序文件的规则。BUILTINS.MAK 与 makefile 文件有同样的语法规则。

MAKE 在当前目录下还查找所有包含文件。如果用户使用 -I 选项,它将对用 -I 选项定义的目录下查找。

B.1.2.2 命令行选项

下面是 MAKE 命令行选项的完整列表。注意大小写是有特殊意义的,如选项 -d 不能代替选项 -D。表 B.1 列出的所有的 MAKE 选项。

表 B.1 MAKE 选项

选项	用途
-? 或 -h	打印帮助信息。
-a	对 .OBJ 文件进行自身依赖性检查。
-B	建立所有的目标文件,不考虑文件的日期。
-Didentifier	将命名的标识符定义成只包含一个字符的字符串"1"。
-Diden=string	定义命名的标识符 iden 为等号后面的字符串,字符串中不包含任何空格或 tab 键。
-ffilename	使用 filename 文件做为 makefile 文件。如果 filename 文件不存在,将使用 FILENAME.MAK 文件。
-i	忽略所有运行程序的退出状态,这与在 MAKEFILE.MAK 文件的所有命令前放"-"意思是等价的。
-ldirectory	在指定的目录下查找包含文件(以及在当前目录下)。
-K	保留(不删除)MAKE 建立的临时文件,所有临时文件名的格式都是 MAKEnnnn. \$\$\$,其中 nnnn 是从 0000 到 9999。
-n	打印这些命令,但不执行它们。这对调试 makefile 文件是有用的。
-s	执行前不打印命令。
-S	当执行命令时,将 MAKE 从内存中调换出来。这样显著地减少了 MAKE 的内存开销,可编译很大的模块。
-Undentifier	取消先前的命名标识符的定义。
-W	将当前定义的非字符串选项(如 -s 和 -a)写到 MAKE.EXE 文件里。

B.1.3 MAKE 的一种简单运用

在第一个 MAKE 运用的例子中,给出了不涉及程序设计的一种简单运用。假设用户正在写一本书,决定将手稿的每章保留在不同的文件上(为了达到举例的目的,假定这本书很短,分别放在不同的三个文件 CHAP1.MASS,CHAP2.MSS 和 CHAP3.MSS 里)。要制成一份当前完整的手稿,首先每一章经过名字叫 FORM.EXE 的程序进行格式整理,然后使用 DOS 的 COPY 命令将.TXT 文件连接起来制作一份完整的手稿文件 BOOK.TXT。

正如程序设计,写一本书也需要不断地修改和精选。当用户在写书时,可能要对一个或多个手稿文件进行修改,但在标注好修改过的文件之前不要丢失先前的手稿文件。另一方面,不要忘记在修改好的文件同其他文件连接之前将该文件经过格式整理,否则将得不到完整的修改过的手稿文件。

解决以上问题的一种粗糙且费时的方法是建立一批处理文件,将每一手稿文件重新格式整理。该文件可能包括以下命令:

```
FORM CHAP1.MSS
```



```
FORM CHAP2.MSS
FORM CHAP3.MSS
COPY/A CHAP1.TXT+CHAP2.TXT+CHAP3.TXT BOOK.TXT
```

运行这个批处理文件总可以生成一份修改后的手稿文件。假定随着时间推移,本书变得越来越大,一天就有十五章,重新格式处理全部所有手稿的过程将可能长得不可容忍。

MAKE 可以解决这一难题,所要做的事仅仅是建立一文件,常用的文件名是 MAKEFILE,MAKE 读该文件可以知道 BOOK.TXT 取决于什么文件,怎样处理这些文件。该文件还包含当 BOOK.TXT 依赖的某些文件已经被修改后用来解释重建 BOOK.TXT 的规则。

对于该例子,用户的 makefile 文件第一条规则可能是:

```
BOOK.TXT:CHAP1.TXT CHAP2.TXT CHAP3.TXT
COPY/A CHAP1.TXT+CHAP2.TXT+CHAP3.TXT BOOK.TXT
```

第一行(即 BOOK.TXT 开头的那一行)表示 BOOK.TXT 文件依赖于一、二、三章的正文格式文件。如果 BOOK.TXT 所依赖的任何文件比 BOOK.TXT 文件新,那么 MAKE 执行下一行的 COPY 命令重新生成新的 BOOK.TXT 文件。

尽管这样,该规则并不能表示所有的一切。每一章的文件又依赖于每章的.MSS 的手稿文件。如果任何一个 CHAP?.TXT 文件比相应的.MSS 文件早,那么该 CHAP?.TXT 文件需要重新生成,如下:

```
CHAP1.TXT:CHAP1.MSS
FORM CHAP1.MSS
CHAP2.TXT:CHAP2.MSS
FORM CHAP2.MSS
CHAP3.TXT:CHAP3.MSS
FORM CHAP3.MSS
```

这些规则表示 CHAP?.TXT 文件是最初各章的 CHAP?.MSS 手稿文件经过格式程序处理生成而来的。

如果用户修改了 CHAP3.MSS,MAKE 在连接 CHAP3.TXT 文件重建 BOOK.TXT 文件之前很灵活地将 CHAP3.MSS 重新经过格式程序整理生成新的 CHAP3.TXT,而其它的.TXT 文件将不会再次重建。

可以对该简单例子进行加工改进,再改进。三条规则看起来十分相似,其实除了每个文件名的最后一字符不同外,其他都是相同的。而且每次添写新的一章时很容易忘记在 makefile 文件上增添一条新的规则。为了解决这些问题,可以使用 MAKE 的隐含规则,依照文件扩展名将文件由一种形式生成另一种形式。这样,就可以用一隐含规则代替三条不同文章的规则:

```
.MXX.TXT:
FORM $ *.MSS
```

该规则表明:如果用户需要将.MSS 文件制成.TXT 文件且是当前最新版的,MAKE 是怎样做到这一步的(另外还必须修改第一条规则,即生成 BOOK.TXT 文件的那条规则,以致于 MAKE 能将新的一章连接到输出文件。这条规则以及后面其他规则将要用宏来实现,详见宏的详细说明)。

一旦用户有了这个 makefile 文件,制作一份完整的最新的草稿文件,所有工作只是在 DOS 提示符下键入单条命令:MAKE。

B. 1.4 制作 makefile 文件

从一批程序设计文件、包含文件、目标文件中建立一个程序的过程,与刚才看到的字处理例子过程很相似。主要不同处是在该过程每步所使用的命令将使用预处理、编译、汇编和连接程序,而不是正文格式程序 FORM. EXE 和 DOS COPY 命令。下面我们进一步深入地研究怎样制作 makefile 文件。该文件指示 MAKE 如何工作。

makefile 文件包含帮助 MAKE 使用户的程序保持最新的定义和关系。用户可以任意生成多个 makefile 文件且可以随意命名它们。如果在使用 MAKE 时没有指定 makefile 文件,MAKEFILE 是 MAKE 查找的缺省文件名。

用户可用任何 ASCII 文件编辑程序,如 Turbo C 的内部编辑程序、Sprint、Mircostar 或 Sidekick 来建立一个 makefile 文件。所有规则、定义和指令都在行尾结束。若一行太长,可以用一斜杠(\)放在一行的最后一个位置作为续行符。

用空格符(空格和制表)将相邻的标识符分开并使规则中的命令缩进。

B. 1.5 makefile 文件的组成

编写 makefile 文件类似编写程序,它带有定义、命令和指令。makefile 文件允许出现以下成份:

- 注解
- 显式规则
- 隐式规则
- 宏定义
- 指令:
 - 文件包含指令
 - 条件执行指令
 - 错误检测指令
 - 取消宏定义指令

注解是用"#"号来标识开始的,MAKE 忽略"#"号后面所有该行的字符。在有注解的这一行就不能使用(\)作续行符。如果(\)在"#"号前,该斜杠就不再是该行的最后一个字符,也就不叫续行符。如果它在"#"号后,那么它就是注解里的一部分,同样也不加处理。

这里是 makefile 文件里注解部分的一些例句:

```
# Makefile for my book
# This file updates the file BOOK.TXT each time
# change one of the .MSS files
```

B. 1.6 命令表

在显式规则和隐式规则中(后面讨论)都可能命令表。这部分介绍 MAKE 是怎样处理这些命令的。

命令表中的命令格式是:

[前缀...] 命令体

命令表中每条命令行可能包含一个(可选的)前缀表,后面是单个命令体。

前缀:一条命令的前缀可以改变 MAKE 对这些命令的处理。前缀就是"@"字符或"-"字符后面紧跟一个数字。表 B.2 是 MAKE 的前缀列表。

表 B.2 MAKE 前缀

前缀	作用
@ MAKE	在执行该命令前不显示该命令。即使 MAKE 命令行中不给出 -s 选项,该命令显示也会被隐藏起来。前缀只在包含有前缀的命令行上有效。
-num	影响 MAKE 对出口代码的处理,如果 num 数字被提供,仅仅当出口状态位超出给出的 num 数字时,MAKE 进行紧急(故障)处理。在该例子中,如果出口状态位超过 4,MAKE 将紧急(故障)处理。
- 4 MYPROG SAMPLE.X	如果没有给出一num 的前缀且状态位是非零,MAKE 停止处理并删除当前目标文件。
-	由于 "-" 号没有数字,MAKE 根本就不检查出口状态位。无论出口状态位是几,MAKE 连续工作。

命令体:命令体的处理就像它是输入到 DOS 命令行一样,但不支持管道操作。

除了<、>和>>重定向操作符之外,MAKE 还增加了<<和&&操作符。这些操作符建立一个文件用于输入给一条命令。<<操作符建立一个内部文件,并改变该命令的标准输入方向,以便能从该命令所建立的文件获得输入。若有一个从标准输入接收输入的程序,该命令如下:

```
MYPROG << !
This is a test
!
```

该命令建立一个包含有"This is a test\n"字符串的内部文件,对它重定向到 MYPROG。在该例子中感叹号是分界符。对于文件,用户可以使用除"#"或"\ "以外的任何字符作为分界符。第一个出现的分界符作为第一个字符的那一行表示结束该内部文件。该分界符后面的同一行的剩余部分(这里,感叹号是分界符)被认为是前面命令的一部分。

&&操作符和<<操作符很相似,它建立一个内部文件,代替该文件作为某一命令的标准输入,并且&&操作符用来代表内部文件的文件名。当想用 MAKE 建立一个被用来作为程序输入的文件时,这十分有用。

下面是为 TLINK 所建的"应答文件"的例子:

```
MYPROG.EXE:$(MYOBS)
tlink/c @ && #
COS $(MYOBS)
$ *
$ *
```

```
$ (MYLIBS) EMU.LIB MATHS.LIB CS.LIB
#
```

注意:当文件被建立时,宏(\$符号标识)被扩展。用不带扩展名且被建立的文件名来取代\$,而\$(MYOBS)和\$(MYLIBS)被宏MYOBS和MYLIBS的值所取代。因此,TLINK接受的文件如下:

```
COS A.OBJ B.1.OBJ C.OBJ B.1.OBJ
MYPROG
MYPROG
W.lib X.lib Y.lib z.lib EMU.LIB MATHS.LIB CS.LIB
```

如果没有使用-K命令行选项,则所有临时文件被删除。如果不能正确工作,可使用-K选项调试用户的临时文件。

批处理程序:MAKE可以操作批处理的文件列表。例如,假设MAKE需要提交一些C文件给Turbo C处理,它可以为每一个文件都运行一次TCC.EXE,但若在调用TCC.EXE时给出一个所有待编译的文件列表,则将会更有效,它节省了每次重装入Turbo C的时间。

MAKE的批处理功能使用户可以集中待处理文件名于一条命令之中,把它们合并在一张表中,对该表只要调用一次命令便能执行。

要使用MAKE批处理命令,可在命令行中加大括号:

```
command [batch-item]...rest-of-command
```

该命令语法将命令的执行延迟到MAKE决定下一部该调用哪一条命令(若有的话)。

下面是一个批处理如何进行工作的例子。假设打算顺序调用如下三条命令:

```
TCC {file1.C}
TCC {file2.C}
TCC {file3.C}
```

MAKE用如下的命令而不是三次调用Turbo C:

```
TCC file1.C file2.C file3.C
```

注意,花括号文件名之后的空格保证把它们彼此分隔开,这是因为一条命令花括号中的内容是连续的。

下面是使用一条隐式规则的例子。假设用户的制作文件有一条隐式规则将C程序编译到.OBJ文件中:

```
.C.obj
TCC -C($<)
```

当MAKE为每个C文件使用该隐式规则时,它将宏\$<扩展成该文件的实际名字并把该名加到要编译的文件列表中(注意花括号中用空格把文件名隔开)。该列表将不断增加,直到满足下列条件:

- 没有还需处理的命令。
- MAKE用完命令行上的空间。若MAKE用完命令行上的空间,它将尽可能多地使用一个命令行,然后把剩下的放在下一个命令行。该列表做完后,MAKE将立即在整个列表上调用TCC(用-C选项)。

执行DOS命令:MAKE通过调用COMMANB.1.COM的备份执行此处所列的DOS"内部"命令:

break	del	path	set	cd
dir	prompt	time	chdir	echo
rd	type	cls	erase	rem
ver	copy	for	ren	verify
ctty	md	rename	vol	date
mkdir	rmdir			

MAKE 用 DOS 搜索路径的方法搜索任何其它命令名:

1. MAKE 首先在当前目录搜索该文件,然后在 DOS 搜索路径中搜索每一个目录。
2. 在每一个目录中,MAKE 首先搜索所指定的具有扩展名.COM 的文件;若没有找到,它将搜索同名而扩展名为.EXE 的文件;若还未找到,MAKE 将搜索具有该名而扩展名为.BAT 的文件。
3. 若 MAKE 找到一个.BAT 文件,它将调用 COMMANB.1.COM 的备份执行该批文件。若在命令行用户给出了文件的扩展名,MAKE 将只搜索拥有该扩展名的文件。下面给出一些例子:

● 本命令使 COMMANB.1.COM 把当前目录改为 C:\include;cd C:\include.

● MAKE 将使用上述的搜索算法搜索适当的文件以执行该命令:

```
tlink lib\cos xy,z,z,lib\cs
```

● MAKE 仅以.COM 扩展名搜索该文件:

```
my prog.com geo.xyz
```

● MAKE 执行该命令时显示:

```
C:\myprogs\file.exe-r
```

B.1.7 显式规则

显式规则直接定义所有文件名,显式规则的格式如下:

```
target [target]...:[source [source...]] [command]
[command]
```

此处,target 是一个待更新的文件,source 是 target 所依赖的文件,command 是任何合法的 DOS 命令。

显式规则定义一个或多个目标文件名、零个或多个源文件以及一张可选的要执行命令的文件列表。在显式规则中列出的目标文件和源文件名可以包含 DOS 驱动器和目录,它们还可以包含通配符。

下列语法是十分重要的:

- 目标文件必须在一行的开头(第 1 列)。
 - 源文件之前至少要有有一个空格或制表符,并放在冒号之后。
 - 每一个 command 必须缩格(前面至少要有有一个空格或制表符)。如前所述,若源文件表或给定命令过长,一行容不下时,斜杠可以用作一个续行符。
- 源文件和命令都是可选的。一条显式规则有可能仅含有 target[target]后跟一个冒号。

一条显式规则意味着所列出的命令将用源文件建立或更新目标。当 MAKE 遇到一条显式规则,它将检查规则中的源文件本身在制作文件的其它地方是不是

目标文件,若是的话 MAKE 将首先考虑该规则。

一旦基于其它规则建立或更新了源文件,MAKE 将检查目标是否存在。若不存在,每条 command 将以所给出的顺序被调用;若存在,它最后一次修改的时间和日期将与每一源文件的时间和日期比较。若有源文件被修改时间比目标文件更近,则该文件列表将被执行。

在给定的 MAKE 执行中,一个给定的文件名可以在一条显式规则左边仅出现一次。

在一条显式规则中的每一命令行以空格开始。MAKE 认为跟在一条显式规则之后的所有行为该命令表的一部分,一直到从第 1 列开始的下一行(即前面没有空格)或文件末。忽略空格行。

特殊情况:处理一条没有命令行跟在其后的显式规则与处理有命令行的显式规则有所不同。

- 若一条显式规则含有命令,只有目标文件所依赖的文件列于该显式规则中。
- 若有一条显式规则不含有命令,目标文件将依赖两组文件:在显式规则中给出的文件,对应于该目标文件与一条隐式规则匹配的任何文件。这将使用户可以指定一个由一条隐式规则控制的依赖性。例如:

```
.C.obj  
tcc -c $<  
prog.obj
```

prog.obj 依赖于 prog.c,若已过时它将执行命令行: `TCC -c prog.c`

例子

下面给出一些显式规则的例子

1. prog.exe:myprog.obj prog2.obj
tcc myprog.obj prog2.obj
2. myprog.obj: myprog.c include\stdio.h
tcc -c myprog.c
3. prog2.obj:prog2.c include\stdio.h
tcc -c -K prog2.c

上述三个例子来自同一个 makefile 文件。仅有被修改的模块被重建。若只有 PROG2.C 被修改过,则只有它将被重新编译。对于 MYPROG.C 也一样。但若包含文件 stdio.h 被修改,则它们两个都将被重编译。(若有在依赖性列表中的某个 OBJ 文件被修改,则将做连接步骤。这种情形将会在一个源文件被修改而引起重编译时出现。)

自身依赖检查:Turbo C 和 MAKE 提供了包含文件的自身依赖性检查。TCC 和 TC 生成 OBJ 文件,该文件告诉 MAKE 哪些包含文件用于建立这些 OBJ 文件。MAKE 的 -a 命令行选项检查该信息并确认是否每一文件或模块都是最新的。

当 MAKE 进行自身依赖性检查时,它将从该 OBJ 文件中读出包含文件的名称、时间和日期。若所包含文件被修改,MAKE 将使 OBJ 文件重编译。例如,考虑下列显式规则:

```
myprog.obj:myprog.c include\stdio.h  
tcc -c myprog.c
```

现在假定名为 MYPROG.C 的文件已由 TCC(2.0 版本或更新版本)编译:

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

```
#include "dcl.h"
```

```
void myprog() { }
```

若用户再用下面的命令行调用 MAKE:

```
make -a myprog.obj
```

它将检查 MYPROG.C 的时间和日期,以及 stdio.h 和 dcl.h 的时间和日期。

B.1.8 隐式规则

MAKE 不仅允许用户定义显式规则,还允许定义隐式规则。隐式规则是显式规则的一般化,它们作用于那些具有一定扩展名的文件。

这里有一个说明这两种规则之间关系的例子。考虑前面例子中的显式规则。该规则是十分典型的,因为它遵循五个一般的原则:一个.OBJ 文件依赖于一个.C 文件。该.C 文件与.OBJ 有相同的文件名,并且由执行 TCC 命令建立。事实上,用户可能有一个 makefile 文件,在该文件中,有几条(或者很多)遵循该原则的显式规则。

要把显式规则写成一条隐式规则,用户可以删除所有这些同一形式的显式规则。一条隐式规则形式可以如下:

```
.c.obj
```

```
tcc -c $<
```

这些规则的意思是“用本命令,任何具有扩展名.C 的文件都可以转换成具有相同文件名但扩展名为.OBJ 的文件”。.OBJ 文件由该规则的第二行建立,该行中 \$< 表示具有源(.C)扩展名的文件名(\$< 符号是一个特殊的宏。在每次执行命令时,\$< 宏都将用相应的.C 源文件的全名所代替)。

下面是一条隐式规则的语法:

```
source _extension, target _extension,
```

```
[command]
```

```
[command]
```

```
.....
```

```
.....
```

如前面所述,命名(command)是可选的并且必须缩格。

source _extension (它必须以行的第一个字符‘.’开始)是源文件的扩展名,即它可应用到任何具有如下格式的文件中:

```
fname.source _extension
```

同样的,target _extension 指文件:

```
fname.target _extension
```

两个文件的 fname 是一样的。也就是说,该隐式规则代替了具有如下格式的显式规则:

```
fname target _extension; fname.source _extension
```

```
[command]
```

```
[command]
```

```
.....
```

```
.....
```

注意:若 MAKE 不能为一个给定的目标文件找到一条显式规则,或者该目标文件对应一条无命令的显式规则,那么 MAKE 将使用隐式规则。

文件的扩展名用于决定使用哪条隐式规则。当一个文件具有和目标文件相同的名字,且扩展名为所提及的源扩展名时,那么将使用一条隐式规则。

例如,用户有一个 makefile 文件,它有如下的内容:

```
.c.obj:
tcc -c $<
```

若有一个 C 程序名为 RATIO.C,要把它编译到 RATIO.OBJ 中,则可使用如下命令:

```
make ratio.obj
```

MAKE 将 RATIO.OBJ 当作目标程序。由于没有显式规则以建立 RATIO.OBJ,MAKE 将应用该隐式规则产生如下命令:

```
tcc -c ratio.c
```

它将完成建立 RATIO.OBJ 所需的编译步骤。

若用户给 MAKE 一条没有命令的显式规则,那么 MAKE 也将使用隐式规则。假设在 makefile 始端有如下隐式规则:

```
.c.obj:
tcc -c $<
```

就可以从规则中删除命令:

```
myprog.obj:myprog.c include\stdio.h
tcc -c myprog.c
```

它将像前面一样工作。

若用户使用 Turbo C 并要在 MAKE 中进行自身依赖性检查,可以删除那些将 OBJ 作为目标程序的显式规则。本节中有关显式规则的例子是:

```
.c.obj:
tcc -c $<
prog.exe:myprog.obj prog2.obj
tlink lib\c0s myprog prog2,prog,lib\cs
```

用户可以用相同的目标文件扩展名写多个隐式规则。若有一个以上的隐式规则对应一个给定的目标文件扩展名,这些规则将以它们出现在 makefile 文件中的次序而被检查,直到找到一个匹配的源文件扩展名,或者直到 MAKE 检查了所有可用的规则。

MAKE 将使用包含具有源文件扩展名的文件的第一条隐式规则。即使该规则中的命令失败,也不再检查其它的隐式规则。

所有在一条隐式规则之后,一直到以非空格字符开始的那一行之间的行,或者到文件尾之间的行,被认为是此命令列表的一部分。

B.1.9 宏

通常,某些命令、文件名或选项在 MAKE 文件中经常重复使用。举例说明,如果用户正在写一个使用中存储模式的 C 语言程序,所有 TCC 命令均使用 -mm 选项,它的意思是在中存储模式下进行编译,假设要把模式换成大存储模式,则需要将所有 -mm 选项换为 -ml。另外一种方法是定义一个宏。

一个宏(macro)表示某一字符串的一个名。一个宏定义确定名和扩展文本,此后,当MAKE遇到该宏名时,它就用扩展的正文来替代此名。

假设在MAKE文件开始处定义了如下的宏:

```
MODEL=m
```

这就定义了宏MODEL,它现在等价于串m。

使用宏,就可以书写如下命令:

```
tcc -c -m $(MODEL) myprog.c
```

当运行MAKE,每一个宏(此处为\$(MODEL))将被其扩展文本(此处为m)来代替,上述命令实质上就是:

```
tcc -c -mm myprog.c
```

利用宏来修改上面所述的存储模式是十分方便的。只要将第一行改为:

```
MODEL=l
```

用户更改了所有的命令使之用于大型存储模式。实际上每次运行MAKE时用-D(定义)命令行选项,可以指定用户所需要哪一种存储模式。

```
make -DMODEL=l
```

它将告诉MAKE将MODEL当作一个扩展文本是l的宏。

B.1.9.1 定义宏

定义宏采用如下格式:

```
macro_name=expansion
```

此处,macro_name是宏名,它是一字母数字串,其间没有空格,在macro_name和等号之间可以有空格。expansion是任何任意的串,可以包含字母、数字、空格和标点符号,以回车键结束。

若macro_name原先被定义过,无论是由在makefile文件中还是在MAKE命令行的-D选项中定义,新的定义将取代旧的定义。

在宏中的大小写是有区别的,即宏名如model、Model和MODEL是各不相同的。

B.1.9.2 使用宏

在制作文件中用如下形式调用宏:

```
$(macro_name)
```

即使宏名仅有一个字符长,所有的宏调用都需要用括号(除预定义过的宏之外)。该结构——\$(macro_name)称作宏调用。

当MAKE遇到一个宏调用时,它将把该宏调用用宏扩展文本代替。若该宏没有定义,MAKE将用空串代替它。

B.1.9.3 特殊情况

宏嵌套:宏在宏定义的左边(macro_name处)不能被调用。只能在右边(扩展正文处)使用,但直到已定义宏被调用时才进行宏扩展。也就是说,当一个宏调用被扩展时,嵌套在它扩展正文中的所有宏也被扩展。

规则中的宏:在规则行中,宏调用立即被扩展。

指令中的宏:在#if和#elif指令中,宏调用立即被扩展。如果在#if和#elif指令中的宏当前未定义,则它扩展成0值(FALSE)。

命令中的宏;当命令执行时,命令中的宏被扩展。

B.1.9.4 预定义宏

MAKE 具有若干个特殊的内部定义宏,如表 B.3 所示: \$d、\$*、\$<、\$:、\$&。\$d 的功能是检查一个宏名是否被定义,它用在条件指令 #if 和 #elif 中。其它的是文件名宏,用在显式和隐式规则中。另外,当前 DOS 环境字符串(用户能看见并且调置使用 DOS SET 命令的字符串)。最后,MAKE 还定义了两个宏: __MSDOS__ 宏定义为 1; __AKE__ 义为十六进制表示的 MAKE 版本号。

定义的测试宏(\$d):若给定宏名已定义,则 \$d 展开为 1,否则为 0。该宏扩展文本的内容是什么并不重要。该特殊的宏只能用于 #if 和 #elif 指令中。

假如,用户要修改 makefile 文件,使得在用户没有指定一个存储模式时,它将用中型模式。用户可以把下面内容放在 makefile 文件的开始部分:

```
#if $d(MODEL) #if MODEL is not defined
MODEL=m #define it to m(MEDIUM)
#endif
```

表 B.3 MAKE 宏

宏	功能
\$d	定义的测试宏
\$*	有路径的基文件名宏
\$<	有路径的全文件名宏
:\$	仅有路径的宏
\$.	无路径的全文件名宏
\$&	无路径的基文件名宏

若使用下面命令行调用 MAKE:

```
make -DMODEL=1
```

则 MODEL 被定义为 1,若用下式调用:

```
make
```

则 MODEL 被定义为 m,即“缺省”的存储模式。

文件名宏

各种文件名宏以相似的方式工作,扩展为某些不同的正在建立的文件全路径名。

基文件名宏(\$*)

基文件名宏可用在显式或隐式规则中。该宏扩展为正在建立的文件名,但不包含扩展名,例如:

文件名为 A:\P\TESTFILE.C

\$* 扩展为 A:\P\TESTFILE

也可以把下列显式规则:

```
prog.exe:myprog.obj prog2.obj
tlink lib\COS myprog progz,prog,lib\cs
```

修改成:

```
prog.exe;myprog.obj progz.obj
tlink lib\CO$ myprog progz, $ *,, lib\cs
```

当本规则中的命令被执行时, \$ * 被一个没有扩展名,但有路径的目标文件名所代替。对于隐式规则,这个宏是十分有用的。

例如,可以有下列形式的隐式规则:

```
c.obj:
tcc -c $ *
```

全文件名宏(\$<)

全文件名宏也用于显式或隐式规则中。在显式规则中, \$ < 展开为全目标文件名(包括扩展名),例如:文件名为 A:\P\TESTFILE.C

\$ < 扩展为 A:\P\TESTFILE.C

例如,规则

```
mylib.obj:mylib.c
copy $ < \oldobjs
tcc -c $ *
```

在编译 MYLIB.1.C 之前将 MYLIB.1.OBJ 拷贝到目录\OLDOBJs 中。

在一条隐式规则中, \$ < 为文件名加上源文件扩展名。例如,有隐式规则:

```
.c.obj:
tcc -c $ *.c
```

产生与下式相同的结果:

```
.c.obj:
tcc -c $ <
```

因为目标文件的扩展名必为.C。

文件名路径宏(\$:)

该宏展开为路径名(没有文件名),例如:

文件名为 A:\P\TESTFILE.C

\$: 扩展成为 A:\P\

文件名和扩展名宏(\$.)

该宏展开为文件名,有扩展名,但无路径名,例如:

文件名为 A:\P\TESTFILE.C

\$. 扩展为 TESTFILE.C

仅有文件名的宏(\$&)

该宏仅展开为文件名,不带路径名或扩展名,例如:

文件名为 A:\P\TESTFILE.C

\$& 扩展为 TESTFILE

B.1.10 指 令

Borland 的 MAKE 允许某些其它 MAKE 版本不允许的东西:支持与 C 语言、汇编程序中和 Turbo Pascal 中相似的指令。使用这些指令可以实现很强的功能,有些在 makefile 文件中的指令以惊叹号(!)作为一行的第一个字符,其它的则以句点(.)打头,表 B.4 是一张

MAKE 指令完整的列表。

B. 1. 10. 1 点指令

下面的每条指令都有其相应的命令行选项,但指令优先于选项。例如,用户以下列方式调用 MAKE:

```
make -a
```

表 B. 4 MAKE 指令

指令	功能
-autodepend	自身依赖性检查
# elif	条件执行
# else	条件执行
# endif	条件执行
# error	使 MAKE 停止并打印一个出错信息
# if	条件执行
. ignore	告诉 MAKE 忽略一条命令的返回值
# include	指定一个文件包含在制作文件中
. noautodepend	关闭自身依赖性检查
. noignore	关闭 ignore
. nosilent	告诉 MAKE 在执行命令之前打印它们
. noswap	告诉 MAKE 不要把自己调入调出内存
. path. ext	为 MAKE 提供一个路径以搜索扩展名为. EXT 的文件
. silent	告诉 MAKE 在执行命令之前不要打印它们
. swap	告诉 MAKE 把自己调入和调出内存
# undef	撤销一个指定宏的定义

但若 makefile 文件有一条. NOAUTODEPEND 指令,则自身依赖性检查将被关闭。

<1>. AUTODEPEND 和. NOAUTODEPEND 打开或关闭自身依赖性检查。它们对应于-a 命令行选项。

<2>. IGNORE 和. NOIGNORE 告诉 MAKE 忽略一条命令的返回值,就像在前面放置一前缀“-”一样(前面已介绍过)。它们对应于-i 命令行选项。

<3>. SILENT 和. NOSILENT 告诉 MAKE 是否在执行命令之前打印它们。它们对应于-s 命令行选项。

<4>. SWA 与. NOSWAP 告诉 MAKE 是否把自己调出内存。它们对应于-S 命令行选项。

<5>. PATH. EXTENSION

该命令放在 makefile 文件中,告诉 MAKE 在哪儿寻找给定扩展名的文件。例如,在一个 makefile 文件中有下列内容:

```
. PATH. C=C:\CSOURCE
. c. obj:
tcc -c $ *
```

```
tmp.exe,tmp.obj
```

```
tcc tmp.obj
```

MAKE 将在 C:\CSOURCE 中为 TMP.OBJ 寻找隐含的源文件 TMP.C,而不是当前目录中寻找。

.PATH 也是一个宏,它的值为路径名。下面是应用 .PATH 的一个例子。源文件在一个目录中,.OBJ 文件在另一个目录中,所有的 .EXE 文件在当前目录中:

```
.PATH.C=C:\CSOURCE
```

```
.PATH.obj=C:\OBJJS
```

```
.c.obj:
```

```
tcc -c -o$(.PATH.obj)\$&.$<
```

```
.obj.exe:
```

```
tcc -e$&.exe $<
```

```
tmp.exe,tmp.obj
```

B. 1. 10. 2 文件包含指令

一个文件包含指令(#include)指定一个文件包含在 makefile 文件中。它有如下形式:

```
#include "filename"
```

有些指令可以嵌套到任何深度。但是若企图产生一个循环包含,内层的包含命令将被认为是错误的。

如何使用该命令呢?假设用户建立一个有如下内容的文件 MODEL.MAC:

```
#if $d(MODEL)
```

```
MODEL=M
```

```
#end if
```

用户可以通过加进下面的指令在任何一个 makefile 文件中使用该条件宏定义。

```
#include "MODEL.MAC"
```

当 MAKE 遇到 #include 时,它将打开指定的文件并读出其内容,就像这些文件的内容本来就在 makefile 文件中一样。

B. 1. 10. 3 条件执行指令

条件执行指令(#if, #elif, #else 和 #endif)增加建立 makefile 文件的灵活性。规则和宏都可以是有条件的,然而命令行宏定义(用 -D 选项)能使 makefile 文件的某些部分有效或无效。

这些指令的格式类似于 C 语言、汇编语言和 Turbo Pascal 中的指令:

```
#if expression
```

```
[lines]
```

```
#endif
```

```
#if expression
```

```
—[lines]
```

```
#else
```

```
[lines]
```

```
#endif
```

```
#if expression
```

```
[lines]
```

```
#elif expression
```

```
[lines]
```

```
#endif
```

注意:[lines]可以是下列语句类型:

- 宏定义。
- 显式规则。
- 隐式规则。
- 包含指令。
- if 组指令。
- 出错指令。
- undef 指令。

条件指令形成一个组,以至少一个 #if 指令开始,以一个 #endif 指令结束。

- 一个 #else 指令可以出现在该组中。
- #elif 指令可以出现在任何 #if 和 #else 指令之间。
- 任何多的规则、宏和其它指令可以出现在各种条件指令之间。注意,一条完整的规则及其命令,不能被条件指令分开。
- 条件指令组可以有任意的嵌套深度。

在一个源文件中,任何规则、命令或者指令都必须是完整的。在同一个源文件中,所有的 #if 都必须与 #endif 指令相匹配。

下面的包含文件是不合法的,因为它没有一个匹配的 #endif 指令:

```
#if $(FILE-COUNT)>5
some rules
#else
other rules
<end-of-file>
```

B. 1. 10. 4 条件指令中允许的表达式

表达式允许在一条 #if 和 #elif 指令中。它们使用类似 C 的语法,结果被认为是一个 32 位的有符号整数。

用户可以用十进制、八进制或十六进制输入数字。若了解 C 语言,也就了解如何在 MAKE 中书写常量,格式是相类似的。若用户是在用汇编语言或者 Turbo Pascal 进行程序设计,最好仔细研究一下后面的例子,这些是一个 MAKE 表达式的合法常量:

```
4536      #十进制常量
0677      #八进制常量(前导 0)
0x23aF    #十六进制常量(前导 0x)
```

一个表达式可以用如下页的操作符。

这些操作符有着和 C 语言的一样优先级。括号可将表达式中的操作数分组操作。

宏可以在表达式内部使用,特殊的宏 \$d() 被识别。当所有宏被扩展后,表达式应具有正确的语法。

B. 1. 10. 5 错误报告指令

错误报告指令(#error)使得 MAKE 中止执行并打印显示 #error 后的致命错误诊断。

其格式为:

```
#error ang_text
```

	操作符	操作
单目运算符		
	-	取负
	~	按位取反
	#	逻辑非
双目运算符		
	+	加
	-	减
	*	乘
	/	除
	%	取余数
	&	按位与
		按位或
	^	按位异或
	&&	逻辑与
		逻辑或
	>	大于
	<	小于
	>=	大于等于
	<=	小于等于
	==	相等
	!=	不等
	>>	右移
	<<	左移
三目运算符	?:	有条件表达式

该指令必须出现在条件指令中,它允许用户定义中止条件。例如,在第一条显式规则前可插入下列代码:

```
#if # $d (MODEL)
#if MODEC is not defined
#error MODEL not defined
#endif
```

如果达到此处时未定义 MODEL,则 MAKE 停止并报告出错信息:

```
Fatal makefile 4: Error directive: Model not defined
```

B. 1.10.6 Undef 指令

宏“撤销定义”指令(#Undef)撤销所指定宏的原定义。若该宏没有被定义,则该指令无效。其语法为:

```
#undef marco_name
```

B. 1.11 MAKE 出错信息

MAKE 诊断信息分为两类:一般错误和致命错误。

- 一般性错误指的是在源 make 文件中有某些语法或语义错误。
- 当一个致命错误发生时,编译立即停止。必须采用适当措施,然后重新进行编译。

下面是在诊断信息中经常出现的错误信息名称和值,当用户得到了一个错误信息,相应的名称和值将被实际信息取代。

手册中	屏幕上
argument	命令行或其它参数
expression	一个表达式
filename	一个文件名(有或无扩展名)
line number	一个行号
message	一信息串

错误信息是按 ASCII 字母排序的,由符号开始的信息首先出现,以所列变量打头的信息放在错误信息表的前面。

例如,用户若要连接一个名为 NOEXIT.C 的文件,可能会得到如下信息:

noexit does not exist — don't know how to make it

为了查看该错误信息,可能需要在错误信息表的前端查找:

filename does not exist _don't know how to make it

如果变量出现在错误信息正文中(举例,"Illegal character in constanter in constantexpression. expression"),用户可以发现该信息的解释是按正确的字母来排序的,排在"I"后面。

B. 1. 11. 1 致命错误信息

- Filename does not exist — don't know how to make it
在建立串中出现不存在的文件名,并且没有建立该文件名的规则存在。
- Circular dependency exists in makefile
makefile 文件指出一个文件需要在其建立之前被更新。例如,显式规则:
filea, fileb
fileb: filec
filec: filea
这就意味着 filea 依赖于 fileb,而 fileb 依赖于 filec, filec 又依赖于 filea,这是非法的,因为一个文件不能够直接或间接地依赖自身。
- Error directive, message
MAKE 已处理一条源文件中的 #error 指令,并且指令文本显示在行中 message 处。
- Incorrect commnad — line argument, argument
用户使用了不正确的命令行参数。
- No terminator specified for in — line file operator
makefile 文件包含 && 和 << 命令操作符以开始一个内部文件,但该文件没有结束。
- Not enough memory
可用的内存不够,应该在一个有更多的存储容量的机器上执行 MAKE。若机器有

640K 内存,可能必须简化源文件或卸去一些内存驻留程序。

● Unable to execute command

一条命令执行失败,这可能是由于找不到命令文件,或者因为它拼写错误或者是因为命令存在但已被损坏。

● Unable to open makefile

当前目录中不含名为 MAKEFILE 的文件,并且没有 MAKEFILE. MAK。

● Unable to redirect input or output

MAKE 不能够打开用来改变输入或输出方向的临时文件。若用户是在一个网络上工作,必须检测是否有进入当前目录的权利。

B. 1. 11. 2 一般性错误

● Bad file name format in include statement

包含文件名必须用引号或尖号括起来。文件名没有左引号或左尖括号。

● Bad undef statement syntax

一条 #undef 语句必须有且只有一个标识符作为该语句的体。

● character constant too long

字符常数只能有一个或两个字符长。

● Command arguments too long

一条命令的参数多于 127 个字符的限制。

● Command syntax error

在下列情况下产生此出错信息:

<1> 制作文件的第一条规则含有前导空格。

<2> 一条隐式规则不含 ext:。

<3> 一条显式规则在(;)之前没有文件名。

<4> 一个宏定义在二字符之前没有名字。

● Command too long

一条命令的长度超过 128 个字符,用户可以使用一个响应文件。

● Division by zero

一条 #if 语句中的除或取模操作中,除数为零。

● Expression syntax error in #if statement

#if 语句中的表达式格式错——含有不匹配的括号、操作符多余或缺省、缺少常数或者是常数多余。

● File name too long

在一条 #include 指令中的文件名太长。在 DOS 中的文件名不能超过 64 个字符。

● If statement too long

一条 if 语句长度超过了 4096 个字符。

● Illegal character in constant expression <expression>

MAKE 在常量表达式中遇到不允许的字符。若该字符是一个字母,这有可能是拼错了标识符。

● Illegal octal digit

发现一个八进制常量中有数字 8 或 9。

- **Macro expansion too long**
一个宏的扩展文本长度不能大于 4096 个字符。该错误常出现在宏的递归扩展中。一个宏不能扩展其自身。
- **Misplaced else statement**
一条 #else 指令缺少匹配的 #if 指令。
- **Misplaced elif statement**
一条 #elif 指令缺少匹配的 #if 指令。
- **Misplaced endif statement**
一条 #endif 指令缺少匹配的 #if 指令。
- **No file name ending**
在一条包含语句中的文件名缺少右引号或右尖括号。
- **Redefinition of target filename**
指定文件出现在一条以上的显式规则的左边。
- **Rule line too long**
一条隐式或显式规则长度超过 4096 个字符。
- **Unable to open include file filename**
指定的文件找不到。当一个包含文件被其自身包含时也会出现此种情况。检查指定文件是否存在。
- **Unexpected end of file in conditional started in line line number**
源文件在 MAKE 遇到 #endif 之前结束。缺少了 #endif 或者被拼错了。
- **Unknown preprocessor statement**
一个 # 字符位于一行之首,其后所跟语句名不是 error, undef, if, elif, include, else 或 endif。

B.2 TLIB:库管理程序

TLIB 是一个管理 OBJ(目标模块)文件库的实用程序。一个库为把一个目标模块的集合作为一个单一的单元进行处理提供了方便。

Turbo C 的库由 TLIB 建立。使用 TLIB 可以建立自己的库,修改 Turbo C 库、用户自己的库、其它程序员建立的库以及用户买来的库。TLIB 功能列表如下:

- 从一组目标模块中建立一个新库。
- 将目标模块或其它库加到一个已存在的库中。
- 从一个库中删除目标模块。
- 从一个库中替换目标模块。
- 从一个库中抽取目标模块。
- 列出一个新库或已存在库的内容。

当修改了一个已存在的库时,TLIB 总是为原库以 .BAK 为扩展名建立一个备份。

TLIB 还可以建立(包含在库文件中)一个 Extended Dictionary(扩展字典),利用它能够

加速连接过程。

尽管在用 Turbo C 建立可执行文件过程中,TLIB 并不是必要的,但它是一个很有用的程序员工具。对于很大的工程的开发,TLIB 将是不可缺少的。若工作在别人开发的目标模块库中,需要时可以借助于 TLIB 维护这些库。

B. 2.1 为什么使用目标模块库

用 C 编程时,通常会建立许多有用的 C 函数,就像 C 运行时间库中的函数那样。由于 C 的模块性,可能将这些函数分成若干分开编译的源文件。在特定的程序中仅使用这些函数的子集。然而要搞清楚到底将使用哪些文件是非常烦琐的事。若总是全部包含所有这些源文件,程序会变得庞大而又难于理解。

目标模块库解决了这个问题。当把一个程序与一个库连接时,连接程序将扫描该库并自动选择当前程序所需的模块。另外,一个库所占的磁盘空间小于一堆目标模块文件所占的磁盘空间的总和(尤其是在每个目标文件都很小时)。一个库还能加速连接程序的工作,因为它只打开一个文件而不是分别打开每个目标模块文件。

B. 2.2 TLIB 命令行

在 DOS 提示符下可直接运行 TLIB,键入 TLIB 并按 Enter。

TLIB 命令行采用如下的一般格式,这里方括号中的项是可选的。

`tlib libname[/C][/E][/Psize][operations][,listfile]`

本节将介绍每个命令的组成项,如表 B. 5。后面一节将进一步讨论 TLIB 的使用。

表 B. 5 TLIB 命令行的组成

项	描述
tlib	调用 TLIB 的命令名。
libname	要建立和管理的库的 DOS 路径名。每一条 TLIB 命令都必须有一个 libname 参数。通配符是不允许的。若没有给出扩展名,TLIB 将认为是 .LIB,因为 TCC 或 TC 建立工程时需要以 .LIB 为扩展名来识别库文件,因而建议使用 .LIB 扩展名。 若指定的库不存在而且又加操作,TLIB 将建立该库。
/C	大小写敏感标志。该选项不常用。
/E	建立扩展字典
/P size	把库页大小置为 size。
operations	TLIB 执行的操作列表。操作可以以任何顺序排列。若仅希望检查库的内容,不用指定操作方式。
listfile	列出库内容的生成文件名。Listfile(若给出的话)前面必须有一个逗号。若没有给出 listfile,那么不产生列表文件。该文件是按字母顺序将各模块排列的列表。每一个模块对应一项,包含一个在模块中定义的公用字符按字母顺序的列表。listfile 的缺省扩展名为 .LST。 可以把 listfile 命名为 CON 以便把该列表输出到显示器,或者用 PRN 输出到打印机。

B. 2.3 操作列表

operations,即操作列表,它描述了 TLIB 的各种功能。它由一系列的操作组成,每一个操作由一个或两个字符的动作符号。后跟一个文件名或模块名组成。可以将空格放在动作符号和文件名或模块名之间,但不能放在两字符的动作符或一个名字的中间。

可以在命令行中指定任意多的操作,只要不超过 DOS 行长限制的 127 个字符。操作排列的顺序无关紧要。TLIB 总是按下列顺序进行操作:

- (1) 最先做所有的抽取操作;
- (2) 其次做所有的删除操作;
- (3) 最后做所有的加入操作。

可以先删除再增加一个模块来实现原有模块的替代。

B. 2.3.1 文件名和模块名

TLIB 将给出的文件名,去掉驱动号、路径名和扩展名,得到一模块名(通常,并不给出驱动号、路径名和扩展名)。TLIB 可以接受通常的缺省名。例如,要从当前目录中增加一个扩展名.OBJ 的模块,可以只给出模块名,而不用给出路径名和.OBJ 扩展名。

文件名或模块名中不允许出现通配符。

B. 2.3.2 TLIB 操作

TLIB 能识别三个动作符(一、+、*),可以单独运用也可以成对地应用它们以组成 5 种不同的操作。用成对的操作时,其动作符的顺序并不重要。下表列出了动作字符及其功能:

表 B.6 TLIB 动作符

动作符	名称	功能
+	加	TLIB 把指定的文件加到库中。若该文件没有扩展名,TLIB 将认为其扩展名为.OBJ。若该文件本身就是一个库(以.LIB 为扩展名),则将把该库中的所有模块加到目标库中。若一个要增加的模块在目标库中已经存在,TLIB 将显示一条错误信息,并不增加该模块。
-	删除	TLIB 从库中删除指定模块。若该模块不在库中,TLIB 显示一条错误信息。 一个删除操作只需一个模块名。TLIB 允许输入路径名、驱动号及扩展名,但仅考虑模块名。
*	抽取	TLIB 将相应的模块从库中提取出来并写到指定的文件中,原库并不改变。若相应的模块不存在,TLIB 显示一条错误信息,并不建立文件。若指定的文件已经存在,则它将被覆盖。
- * * -	抽取和删除	TLIB 将指定的模块拷贝到相应文件中,然后把该模块从库中删除。
- + + -	替代	TLIB 将指定的模块用相应的文件代替。

B. 2.4 使用响应文件

在处理大量的操作,或者发现自己在重复地使用某些操作时,可以使用响应文件,一个响应文件即为一个简单的 ASCII 文本文件(它可用 Turbo C 编辑程序建立),它包含所有或者部分 TLIB 操作。使用响应文件,可以建立不受 DOS 命令行限制的较长的 TLIB 命令行。

要用一个响应文件 `pathname`,在 TLIB 命令行的任何位置写上 `pathname`。

- 一行以上的文本可以组成一个响应文件,将字符"&"放在一行行尾表示续行。
- 没有必要把完整的 TLIB 命令放在响应文件中,可以只放其中的一部分,而另一部分可以输入。
- 在一 TLIB 命令行中,可以用一个以上的响应文件。

B. 2.5 建立扩展字典:/E 选项

为加速大的库文件连接的过程(如标准 `Cx.LIB`),可以用 TLIB 建立一个扩展字典并把它加在该库文件中,字典中含有标准库中没有的某些信息。这些信息使 TLINK 能够更快地处理库文件,特别是当它们放在软盘或一个很慢的硬盘上时。所有盘上的库文件都可以有扩展字典。

若要为一个正在修改的库文件建立一个扩展字典,在调用 TLIB 增加、删除或替代该库中的模块时,加上 /E 选项。想要为一个不想进行修改的库加上一个扩展字典,可以用 /E 选项并要求 TLIB 从该库中删除一个不存在的模块。TLIB 将显示一条信息说明所指定的模块在库中找不到,但它也将为该库建立一个扩展字典。例如:

```
tlib /E mylib -bogus
```

B. 2.6 设置页大小:/P 选项

每一个 DOS 库文件都含有一个扩展字典(它在 `.LIB` 文件末尾的目标模块之后),该字典包含库中每个模块的一个 16 位地址,这个地址依据库页的大小而给出(缺省时,页大小为 16 个字节)。

库的页大小决定了库中所有目标模块的最大组合尺寸——不能超过 65536 页。缺省(最小的)页大小为 16 字节,使得库可以有 1MB 大小。要建立一个更大的库,必须用 /P 选项设置页大小。页大小必须是 2 的幂,它不能小于 16 或大于 32768。

库中的每一个模块必须开始于一页的分界处。例如,在一个库中,页大小为 32 字节,平均每个目标模块将浪费 16 字节。若要建立一个库,而该库对于给定的页大小来说过大,超过了 65536 页 TLIB 将会显示一条出错信息并建议用 /P 进行调整每页大小。

B. 2.7 高级操作:/C 选项

当添加一个模块到一个库中,TLIB 生成一个该模块所定义的公共符号的字典。库中的所有符号应彼此区别。若企图添加一个引起重复符号的模块到库中,TLIB 将显示一条错误信息并不作添加。

通常,TLIB 在检查库中符号是否重复时,大写和小写字母没有区别。例如,符号 `lookup` 和 `LOOKUP` 被认为是一样的。但是 C 认为大写和小写字母是有区别的,因而用 /C 选项将

一个模块增加到一个库中时,允许模块有一个符号仅在其大小写方面与另一个已在库中的符号有区别。

若没有用 /C 选项,TLIB 拒绝那些仅在大小写方面有区别的符号,这看上去似乎很奇怪,尤其是因为 C 是一种大小写敏感的语言。但是某些连接程序不能够区分那些仅在大小写形式上不同的符号。例如,这些连接程序将 stars、Stars 和 STARS 看作是一样的。另一方面 TLINK 能够区分大小写符号,并且它能够接受那些符号仅在大小写形式上有所不同的库。在这种情况下,Turbo C 将 stars、Stars 和 STARS 看成不同的标识符。只要仅用 TLINK 连接库,则使用 TLIB /C 选项是没有问题的。

B. 2.8 例 子

下面是说明一些 TLIB 功能的例子:

- (1) 用模块 X.OBJ、Y.OBJ 和 Z.OBJ 建立一个名叫 MYLIB.LIB 的库:

```
tlib mylib +x +y +z
```

- (2) 像上例中那样建立一个库并获取 MYLIB.LST 列表:

```
tlib mylib +x,+y,+z,mylib.lst
```

- (3) 生成一张库 CS.LIB 的列表文件 CS.LST:

```
tlib cs,cs.lst
```

- (4) 以一个新的模块替代 X.OBJ,并增加 A.OBJ,删除 Z.OBJ:

```
tlib mylib -+x +a -z
```

- (5) 从 MYLIB.LIB 中抽取 Y.OBJ,并获取一张 MYLIB.LST 列表:

```
tlib mylib *y,mylib.lst
```

- (6) 用模块 A.OBJ、B.OBJ、... 和 G.OBJ 并使用一个响应文件建立一名叫 ALPHA 的新库:

首先用下式建立一个文本文件 ALPHA.RSP:

```
+a.obj +b.obj +c.obj &
+d.obj +e.obj +f.obj &
+g.obj
```

然后用 TLIB 命令生成一个名叫 ALPHA.LST 的列表文件:

```
tlib alpha .. alpha.lsp,alpha.lst
```

B. 3 连接程序 TLINK

IDE 拥有自己内部的连接程序。对于 Turbo C 的命令行版本,连接程序 TLINK 被自动地调用(除非不做连接处理)。如果在上述两种编译程序中不做连接处理,那么若需生成执行文件就必须手工调用 TLINK。本章将描述如何使用 TLINK。

缺省时,在编译成功之后,TCC 自动调用 TLINK,将各目标模块和库文件连接生成可执行文件。

B. 3.1 调用 TLINK

用户可以在 DOS 命令行输入 tlink,参数可带可不带,用此来调用 TLINK。

当无参数调用时,TLINK 将显示出一个参数和选项总表,如表 B.7 形式。

表 B.7 Turbo C 的 TLINK 的选项表

```

Turbo Link  Version 4.0 Copyright (c) 1991 Borland International
Syntax: TLINK objfiles, exefile, mapfile, libfiles, deffile
@xxxx indicates use response file xxxx
Options: /m = map file with publics
          /x = no map file at all
          /i = initialize all segments
          /l = include source line numbers
          /L = specify library search paths
          /s = detailed map of segments
          /n = no default libraries
          /d = warn if duplicate symbols in libraries
          /c = lower case significant in symbols
          /3 = enable 32-bit processing
          /v = include full symbolic debug information
          /e = ignore Extended Dictionary
          /t = create COM file (same as /Tc)
          /o = overlay switch
          /P[=NNNNN] = pack code segments
          /A=NNNN = set NewExe segment alignment factor
          /ye = expanded memory swapping
          /yx = extended memory swapping
          /C = case sensitive exports and imports

```

在本版的 TLINK 对它选项的大小写是敏感的,例如, -M 不同于 -m,在 TLINK 命令选项前可使用短横或反斜杠。

TLINK 命令行的一般语法是:

syntax: TLINK objfiles, exefile, mapfile, libfiles, deffile

这要求命令以给定的顺序输入文件名,并以逗号将文件类型隔开。

B.3.1.1 DOS 中连接的范例

如果给出如下 TLINK 命令行:

```
tlink /C mainline wd in tx,fin,infin,lib\comm lib\support
```

TLINK 将认为:

- 在连接过程,字母大小写是有区别的(由 /C 来决定)。
- 要连接的 OBJ 文件为:MAINLINE.OBJ、WD.OBJ、L N.OBJ 和 TX.OBJ
- 可执行程序名将为 FIN.EXE
- 映像文件为 MFIN.MAP
- 要连接的库文件为 COMM.LIB 和 SUPPORT.LIB,它们都在子目录 LIB 中。

B.3.1.2 TLINK 命令行中的文件名

如果没指定执行文件名,TLINK 从第一个目标文件名加上 .EXE 作为执行文件名。

如果指定了可执行文件的完整文件名, TLINK 将使用该名建文件, 但是该文件的真正属性依赖其他选项或者模块定义文件中的设置。

如果未指定, map 文件名, TLINK 对 .EXE 文件名加扩展. MAP 产生 map 文件名。如果没包含库将不做任何库连接。

TLINK 将为没有扩展名的文件加上扩展名:

- 目标文件: .OBJ。
- 可执行文件: .EXE (使用 /t 或 /Tde 选项时可执行文件扩展名是 .COM 而不是 .EXE)。
- 映像文件: .MAP。
- 库文件: .LIB。

以 B. 3. 1. 1 节为例, 若没有指定 .EXE 文件, TLINK 将使用所列的第一个目标文件名来作为可执行文件名 (扩展名改为 .EXE)。例如, 在前面的例子中, 若没有指出 FIN 为 .EXE 的文件名, TLINK 将建立 MAINLINE. EXE 作为可执行文件。

当用户使用 /t 选项时, 可执行文件扩展名为 .COM 而不是 .EXE。

TLINK 一般总是要建立一个映像文件, 除非用户在命令行中用 /X 选项显式地说明不要建。

- 若给出 /m 选项, 映像文件将包含一个公共符号列表。
- 若给出 /s 选项, 映像文件将包含一个详细的段映像。

下面是 TLINK 定义映像文件名所遵循的规则:

- 若没有指定任何 .MAP 文件, TLINK 将 .EXE 文件名以导出映像文件名 (可从命令行中或者响应文件中给出该 .EXE 文件名。若没有给出 .EXE 文件名, TLINK 将从第一个 .OBJ 文件导出映像文件名)。
- 若用户在命令行中 (或响应文件中) 指定了一个映像文件名, TLINK 将 .MAP 作为扩展名加到该文件名中。

即使用户指定了一个映像文件名, 但只要使用 /X 选项, 则 TLINK 不建立任何映像文件。

B. 3. 2 使用响应文件

TLINK 可以使用户在命令行中、响应文件中或者两者一起提供各种参数。一个响应文件是一个含有选项和文件名的文本文件, 这些选项和文件名通常是命令行 TLINK 之后输入的内容。

不像命令行, 一个响应文件可以分几行。可以把一个很长的目标文件或库文件列表分成几行, 只要在每行的行尾用 “+” 结束便可以把这几行当作一个连续行。

用户可以在不同行开始连接程序的四部分参数: 目标文件、可执行文件、map 文件、库文件。如果这样做, 就必须去掉用于分隔各部分的逗号, 且每行末尾不应再加上 “+” 号。

为了说明这些特征, 假设将前面的命令行重写成应答文件 FINRESP:

```
/c mainline wdt  
In tx, fin  
mfin
```



```
lib\comm lib\support
```

然后键入命令：

```
tlink @finresp
```

注意：必须在文件名前写上@字符以提示此名字为响应文件名。

另外，可将 link 命令行分成几个响应文件。例如，可将前面的命令行分成如下两个响应文件：

文件名	内容
LISTOBSJS	mainline + wd + In tx
LISTLBIS	lib\comm lib\support

然后再键入如下的 TLINK 命令：

```
tlink /c @listobjs,fin,mfin,@listlibs
```

B. 3.3 和 Turbo C 模块一起使用 TLINK

Turbo C 支持六种不同的存储模式：tiny、small、compact、medium、large 和 huge。当用户使用 TLINK 建立一个可执行的 Turbo C 文件时，必须为正在使用的存储模式包含初始化模块和库文件。

用 TLINK 连接 Turbo C 文件时，程序的一般格式为（必须要包含启动码和库文件路径）：

```
tlink c0[W[D]x myobjs,exe,[map],[mylibs][EMU | FP87mathx] cx,[deffile]
```

此处的文件名代表：

myobjs	= 要连接的 .OBJ 文件。
exe	= 可执行文件名。
[map]	= 映像文件(可选)名。
[mylibs]	= 在连接时间希望包含的库文件(可选)。

其余的文件名代表 Turbo C 文件：

C0x	= 存储模式 t、s、c、m、l 或 h 的初始化模块。
EMU fp87	= 浮点库文件(选择一个)。
mathx	= 存储模式 s、c、m、l 或 h 的数学库。
cx	= 存储模式 s、c、m、l 或 h 的运行时间库。

B. 3.3.1 启动代码

初始化模块名为 C0x.OBJ，为 DOS。这里 X 代表相应的模式：t、s、c、m、l、h。连接一个适当的初始化模块通常会引起很长的错误信息表，告诉用户哪些标识符未定义或者没有建立栈。

B. 3.3.2 库文件

目标文件和库文件的顺序十分重要。用户必须把 C 初始化模块放在目标文件列表中的第一个。库列表将顺序包含下列内容：

- 用户自己的库文件。
- FP87.LIB 或 EMU.LIB, 后面跟着 MATHX.LIB (仅当使用浮点时才需要)。
- Cx.lib, 标准 Turbo C 运行时间库文件。
若使用了 Turbo C 图形函数, 则必须连接起 GRAPHICS.LIB。
若程序中使用浮点运算, 必须把浮点库及一个数学库写在连接命令中。Turbo C 的两个浮点库独立于程序的存储模式。
- 若希望包含仿浮点逻辑, 以便不管是否有数学协处理器 (80x87) 芯片, 程序都可以工作, 则必须使用 EMU.LIB。
- 若知道程序总是运行在有数学协处理器 (80x87) 芯片的机器上, 则使用 FP87.LIB 库将生成一个精确、快速的可执行程序。
数学库名为 MATHX.LIB, 这里 X 对应着模式: s、c、m、l、h (tiny 和 small 对应的同样是 MATHS.LIB)。

在连接命令中总可以有仿真器和数学库, 但如果这样做了, 并且程序中没有浮点运算, 则该库中不会有东西被加到可执行文件中。当用户知道程序中没有浮点运算时, 在连接时的命令行中不写这些库将会节省运行时间。

用户必须为程序存储模式包含 C 运行时库。C 运行时间库名为 CX.LIB, 这里 X 对应如前所述的模式。

若用户不打算使用所有的六个存储模式, 并且硬盘空间有限, 可能希望只保留所用的相应模式的文件。对于 DOS 需要 FP87.LIB 或者 EMU.LIB。

表 B.8~B.9 是每一存储模式所需的库文件列表。

表 B.8 .OBJ 和 .LIB 文件

存储模式	一般初始模块	可用初始模块	运行库
Tiny	C0T.OBJ	MATHS.LIB	CS.LIB
Small	C0S.OBJ	MATHS.LIB	CS.LIB
Compact	C0C.OBJ	MATHC.LIB	CC.LIB
Medium	C0M.OBJ	MATHM.LIB	CM.LIB
Large	C0L.OBJ	MATHL.LIB	CL.LIB
Huge	C0H.OBJ	MATHH.LIB	CH.LIB

注意, tiny 和 small 模式用相同的库, 但启动文件 (C0T.OBJ 和 C0S.OBJ) 是不同的。

B.3.4 与 TCC 一起使用 TLINK

用户还可以使用 TCC, 这个独立的 Turbo C 编译程序, 作为 TLINK 的“前导”, 将以正确的启动文件、库文件和可执行程序名调用 TLINK。

若这样做, 必须在 TCC 命令行上以显式 .OBJ 和 .LIB 的扩展名给出文件名。例如, 有如下的 TCC 命令行:

```
tcc -mx mainfile.obj sub1.obj mylib.lib
```

TCC 将以文件 C0X.obj、EMU.LIB、MATHX.LIB 和 CX.LIB 调用 TLINK (初始化模块、缺省的仿 8087 库、数学库和存储模式运行时间库)。TLINK 将把它们与用户的模块 MAINFILE.OBJ、SUB1.OBJ 和 MYLIB.LIB 连接。

当 TCC 调用 TLINK 时,它将以 /C(case-sensitive 连接)选项作为缺省。用户可以用 -l-c 去掉该缺省项。

更详细的信息请见本书的附录 A“命令行编译器”。

B.3.5 连接选项

TLINK 选项可出现在命令行的任何地方,选项由反斜杠(\),连字符(-)或 DOS 开关字符,后面跟着选项指定字符(m,x,i,l,s,n,d,c3,v,e,o,t,ye 或 yx)组成。(DOS 开关字符缺省时为/,用户可以通过调用 INT 21H 来改变它)。

若有一个以上的选项,选项之间的空格是没有意义的(/m/c 和 /m /c 是相同的),而且还可以把它们放在命令行的不同地方。下面将详细讨论这些选项。

B.3.5.1 /x,/m,/s 选项

缺省时,TLINK 总是建立一个可执行文件的映像文件。该映像文件仅包括程序段列表、程序开始地址、在连接时产生的警告信息或出错信息。

若用户希望建立一个更完整的映像文件,/m 选项将公共符号列表放在该映像文件中,按字母顺序以及地址递增的次序排序。此种映像文件在调试时非常有用。许多调试程序用公共符号列表使得用户能够在调试过程中引用符号地址。

/s 选项可以像 /m 选项那样建立一个具有段、公共符号和程序开始地址的映像文件,但它还添加了一个详细的段映像文件。

对于每一个模块的每一段,映像文件包含其相应的地址、字节长、类型、段名、组、模块和 ACBP 的信息。

如果有同一段出现在一个以上的模块中,在每一个模块里将出现在不同的行中(例如,SYMB.C 除了 ACBP 域之外),该详细的段映像文件中的信息是自动解释的。

ACBP 域把 A(alignment)、C(combination)和 B(big)特性代码化到一组四位的域中,就像 INTEL 所定义的那样,TLINK 仅使用三个域:A、C 和 B 域。映像文件中的 ACBP 值是以十六进制书写的,表 B.9 的域值必须“或”起来构成 ACBP 的值。

表 B.9 TLINK ACBP 的域

域	值	说明
A 域 (alignment)	00	绝对段
	20	按字节段
	4	按字段
	60	按节段
	80	按页段
	A0	无名绝对存储区
C 域 (combination)	00	可不被连接
	08	公共连接段
B 域 (big)	00	小于 64K 的段
	02	等于 64K 的段

当用/s生成详细映像文件时,公共符号表将给不被引用的公共符号以“idle”标志。例如,下面为映像公共符号段中的部分内容,表明符号 Symbol1 和 Symbol3 没有被引用:

```
0C7F,031E    idle    Symbol1
0000,3EA2                Symbol2
0C7F,0320    idle    Symbol3
```

B. 3. 5. 2 /i 选项

/i 选项为源代码行号在 .MAP 文件中建立了一个段。在使用它之前,必须建立以-y 或-v 选项编译的 .OBJ 文件。若用户告诉 TLINK 不要建立映像文件(用/X 选项),则该选项无效。

B. 3. 5. 3 /i 选项

/i 选项使未初始化的 trailing 段输出到可执行文件中,即使该段中不包括数据记录,该选项不经常用到。

B. 3. 5. 4 /n 选项

/n 选项使连接程序不承认由某些编译程序指定的缺省库。当缺省库在另一个目录中,则要用该选项,因为 TLINK 将不支持库的搜索。当连接那些用其它语言书写的模块时,可以使用此选项。

B. 3. 5. 5 /c 选项

/c 选项使在公用和外部符号中的大小写具有不同的含义。比如,在缺省时,TLINK 认为 cloud、Cloud 和 CLOUD 是相同的,而/c 则认为它们是不同的。

B. 3. 5. 6 /d 选项

通常,当一个符号出现在一个以上的库文件中时,TLINK 不发出警告。若符号必须包含在程序中,TLINK 将使用其所能找到的命令行中第一个程序的相应符号。由于这是通常的处理方法,TLINK 不发出符号重复的警告。

假设有两个库:一个称作 SUPPORT.LIB,另一个辅助库称作 DEBUGSUP.LIB。还假设 DEBUGSUP.LIB 中含有 SUPPORT.LIB 中的某些子程序(但在 DEBUGSUP.LIB 中这些子程序具有稍微不同的功能,如子程序的调试版本)。若在连接命令中把 DEBUGSUP.LIB 放在第一位,则将使用的是调试子程序,而不是在 SUPPORT.LIB 中的子程序。

若不想利用这一特征,或者不能肯定哪些子程序是重复的,便可选用/d 选项。TLINK 将列出所有库中重复的符号,即使这些符号可能不被程序所使用。

在给出该选项时,TLINK 还将警告有关既出现在 .OBJ 中又出现在 .LIB 中的符号。在这种情况下,由于出现在命令行中第一位的文件中的符号将被连接进去,因而在 .OBJ 文件中的符号便是被使用的那一个。

连接命令中使用的库文件中不包含任何重复的符号。但是,EMU.LIB 和 FP87.LIB (或 CS.LIB,和 CL.LIB)明显地含有重复的符号,所以它们将永远不可能正确地在连接中使用。然而,在 EMU.LIB、MATHS.LIB 和 CS.LIB 之间没有重复符号。

与 Turbo C 一起装入的库文件都含有一个扩展字典,该扩展字典中使 TLINK 加快连接这些库的信息。还可以用/E 选项把扩展字典加到另一个库文件中去;而使用/e 选项,则使字典功能无效。

B. 3. 5. 7 /e 选项

装在 Turbo C 里的库文件包含有带着信息的扩展字典,它们用来帮助 TLINK 快速连

接这些库。使用 TLINK 的 /E 选项用以将扩展的字典加到其他库文件中。而 TLINK 的 /e 选项将不使用该扩展字典。

尽管连接包含有扩展字典的库较快,但如果使用扩展字典,连接时可能占用较大的内存空间,所以 /e 选项的使用与否应根据具体情况而定。

如果用户使用 /e 选项关闭扩展字典的使用,TLINK 将忽略任何包含在扩展字典库中调试的信息。

B. 3. 5. 8 /t 选项

若用户在 tiny 存储模式下编译文件,并在 /t 选项打开的情况进行连接,TLINK 将生成一个 .COM 文件,而不是通常的 .EXE 文件。

注: .COM 文件大小不能超过 64K,不能有前缀段,不能定义一个堆栈段,并且必须以 0:100H 为始地址。当可执行文件扩展名不是 .COM(如 .BIN),则起始地址可以为 0:0 或 0:100H。

B. 3. 5. 9 /v 选项

/v 选项使 TLINK 把调试信息包含在可执行文件中,只要该项出现在命令行上,则所有含有调试信息的模块的调试信息都被包含进去。用户可以用 /v+ 和 /v- 选项选择是否包含调试信息。例如:

```
tlink mod1 /v+ mod2 mod3/v- mode4
```

包含模块 mod2 和 mod3 的调试信息,但不包含模块 mod1 和 mod4 的调试信息。

B. 3. 5. 10 /3 选项

当一个或一个以上要连接的目标模块由 TSAM 或由可兼容的汇编程序产生,并且它含有 32 位 80386 机器码时,则应使用 /3 选项。该选项将增加 TLINK 的内存要求并减慢连接过程,因此只有在需要时才使用它。

B. 3. 6 TLINK 的限制

任何 Microsoft 目标码可以用 TLINK 连接。

TLINK 可以与 Turbo C(集成环境和命令行版本)、TASM、Turbo Prolog 和其它编译程序一块工作。

B. 3. 7 出错信息

TLINK 有三种错误信息:致命信息、非致命错误和警告。

- 致命错误使 TLINK 立即停止,.EXE 文件被删除。
- 非致命错误不删除 .EXE 或 .MAP 文件,但用户不能执行 .EXE 文件。
- 警告提醒用户尽量满足警告中的条件。当警告出现时,.EXE 和 .MAP 文件仍能产生。

在本节中列出了出错信息中的一般名字和值,当用户遇到出错信息时,相应的名字和值将被实际信息替换。

手册中	屏幕所见的实际信息
errorcode	内部出错代码
filename	文件名(有无扩展都可)

group	组名
module	模块名
segment	段名
symbol	符号名
xxxxh	4 位十六进制数,后跟 h

出错信息按 ASCII 字母序排列,开头是字母或数字的信息先列在前面。当用户获得信息时,由于上述列出的变量名已被实际的内容所替换,不能按所看到的那样按字母排序,因此所有这些消息被放在出错信息表的前面。

如果变量名出现在出错信息正文后(例如“Invalid segment definition in module Moudule”,这里的变量名是 Moudule),这些消息是按正确的字母顺序来排列的,该例子在一般情况下,应在“I”的后面。

B. 3. 7. 1 致命错误

当致命错误出现时,TLINK 将停止工作,并且删除 .EXE 文件。

● Symbol in module module1 conflicts with modul modul2

该错误信息的产生是由于两个符号不一致。这说明不同属性的一个符号在两个模块中都有定义。

● Bad character in parameters

下列字符中有一个出现在命令行或响应文件中:“ * <=>? [] |”或者任何非水平制表、回车、换行、Ctrl-Z 等控制字符。

● Bad object file filename

遇到了一个格式有错误的目标文件,这主要是没有完整地建立起源文件或目标文件。这可能出现在机器正在编译时被重新启动,或者编译程序在碰到 Ctrl-Break 时还未删去其输出的目标文件。

● Cannot generate Com file;data below initial CS;IP defined

该错误源于试图在一个 .COM 文件的起始地址(通常是 100)以下生成数据或代码。一定要记住起始地址是用指令 ORG 100H 而不是 100,该错误信息不应该出现在用高级语言编写的程序中。如果的确出现了,就一定要保证连接的是正确的起始(C0)目标模块。

● Cannot generate COM file;Invalid initial entry point address

使用了 /t 选项,但程序的起始地址不是 100H,这是 .COM 文件所必需的。

● Cannot generate COM file;program exceeds 64K

选用了 /t 选项,但是程序的总长度超过了 .COM 文件的长度限制。

● Cannot generate COM file;segment-relocatable items present

选用了 /t 选项,但程序包含了段前缀,这在 .COM 文件中是不允许的。

● Cannot generate COM file;stack segment present

选用了 /t 选项,但是程序中声明了一个栈段,这是 .COM 文件所不允许的。

DOS error,ax=decimed number

这是在一个 DOS 调用返回一个意外错误时出现的。所打印的 ax 值是出错代码,它可以说明一个 TLINK 内部错误或一个 DOS 错误,引起 TLINK 这种错误的原因

可能是 DOS 调用的读、写、查找以及关闭。

- **Group group exceeds 64K**
组段加起来后,超过了 64K 字节,则出现此错误信息。
- **Invalid entry point offset**
当具有 32 位记录的模块被连接时可能出现此信息,表示初始化程序入口地址的偏移量超过 DOS 的 64K 限制。
- **Invalid group definition group in module module**
试图把段赋给多个组。在 .OBJ 文件中的 GRPDEF 记录格式不正确时也可能出现此信息,例如,用户建立的 .OBJ 文件中或者是产生 .OBJ 文件的翻译程序中有错误。
- **Invalid initial stack offset**
当具有 32 位记录的模块被连接时可能出现此信息,表示初始化堆栈指针值超过 DOS 的 64K 限制。
- **Invalid segment definition in module module**
该信息一般只在编译程序生成一个有错的目标文件时出现。如果出现在一个由 Turbo C 创建的一个文件中,那么就要重新编译该文件。
- **Not enough memory**
内存不够。改正的办法是删去驻留程序,或者减少 RAM 驱动器的大小,然后再次运行 TLINK。
- **Relocation offset overflow in modul Module**
该错误只出现在 32 位目标模块中,它说明存在一个大于 64K 的重定位偏移量。
- **Relocation table full**
正在被连接的文件包含的基前缀多于标准 DOS 重定位表所能允许的最大值。
- **Segment segment exceed 64K**
连接程序内部表溢出。这意味着正在被连接的程序超出了连接程序使用公共或外部符号的能力。
- **32-bit record encountered in modul modul;use "/3" option**
遇到了一个含有 80386 的 32 位记录目标文件,但是 /3 选项未被使用。此时需要用 /3 选项重新启动 TLINK。
- **Unable to open file filename**
所命名的文件不存在或者拼写错误。
- **Unknown option**
命令行或响应文件中的反斜杠(\)、连字符(-)或 DOS 开关字符后面没有跟所允许的选项字符。
- **Write failed,disk full?**
在 TLINK 无法将它想写的数据存盘,这通常是因为磁盘空间不够。

B.3.7.2 非致命错误

TLINK 有三种非致命的错误。如上所述,当出现了一非致命错误时,.EXE 和 .MAP 文件不被删除。但在集成环境下,这些错误被看成是致命的错误。

- **Fixup overflow in module module, at segment:xxxxh, target=segment or symbol:xxxxh**

这说明在一目标文件中引用了错误数据或者代码。该信息一般总是因为存储模式的错误匹配。在不同的代码段中对一函数的近调用是最有可能导致这种错误的原因。该错误的产生还可能是由于用户对一数据变量进行近调用或把函数当作数据引用。不管何种情况,在错误信息中名为 target 的符号就是所引用的变量或函数。该引用在所命名的模块中,故可以查看该模块的源文件以确定错误所在。

如果没有发现错误的原因,或者用户正用汇编语言或非 Turbo C 的高级语言来编程,可能还会有其它原因导致产生该信息。即使在 Turbo C 中,该信息也可能在使用不同的段式组名时产生。

- **Out of memory**
该错误一般意味着:为被连接在一起的目标文件定义了太多的模块、外部变量、组或者是段。
- **Undefined symbol symbol in module Module**

在给出的模块中被引用的符号没有在目标文件和库中定义。可以核实符号的拼写是否正确。一般在用户没有将不同源文件中 Pascal 类和 cdef 类型的符号声明加以正确匹配时,或者在用户漏掉了程序所需要的一个 .OBJ 文件时会出现此错误。

B.3.7.3 警告

TLINK 有三种警告信息:

- **Warning:symbol defined in module Module is duplicated in module module2**
所命名的符号在给出的多个模块中有重复定义。这可能是由于在命令行中两次给出了同一个目标文件。
- **Warning:No stack**
该警告是在任何目标文件中或库中都未定义堆栈段时出现。这对 Turbo C 的微型存储模式来说是正常的信息,对任何转换成一个 .COM 文件的应用程序来说也是如此。对其它程序则说明出错。
- **Warning:Segment segment is in two groups:group1 and group2**
连接程序发现两个有名字的组之间存在相互矛盾的信息。

B.4 THELP 帮助

THELP.COM 是一个驻留内存的实用程序,它可以在用户不使用 IDE 时提供联机帮助(也就是,当用户使用的不是 IDE 中的编辑器,或使用 Turbo C 命令行版本或者是在使用其它产品,如 Turbo Debugger,此时都可进行联机帮助)。THELP 要求占据 58K 的内存空间。

B.4.1 装入和调用 THELP

注意:用户必须使 THELP 常驻内存,此时就如同 sidekick 1.x 或 Sidekick Plus 一样。用户必须确信在装入 Sidekick 之前先装入 THELP。

用户首先必须装入 THELP 程序,为了在另一个程序(或在命令行中)用到它,必须确定

文件 TCHELP.TCH(包含 Turbo C 联机帮助的详细信息的包含文件)在当前目录下(如果用户想在其它目录下保存 TCHELP.TCH,THELP 必须加/F 命令行选项才能找到它;INSTALL 程序将正确的路径提供给 THELP)。

用户在使用 THELP 之前,必须在 DOS 命令行下键入:THELP [options]以加载 THELP。这个工作只需在刚建立时做一次。

如果用户在于一项其它的工作时,可以在任何时候调用 THELP。这样在屏幕的底部将显示有关帮助信息的条目,然后再按 THELP 的热键。缺省的热键是在数字小键盘上的 5。

THELP 的作用将通过如表 B.10 的用户的帮助屏幕显示出来。

表 B.10 THELP 帮助屏幕

键	功能
Up Down Left Right	在当前帮助屏上把亮条从一个关键字移到另一个关键字。
Shift+Arrow	整块地移动光标。
Home and End	回到每行的开头或者是行尾。
Tab 和 Shift+Tab	移到下一个或者是前一个关键字。
PgUp/PgDn	如果还有内容,则整屏整屏地滚动。
Enter	在当前的帮助屏下选择了亮条所在的帮助项。
Esc	结束帮助查询。
F1	显示 Content 屏幕,在任何时候按 F1,屏幕都回到帮助索引状态。
Shift+F1	显示帮助索引,在任何帮助屏下按 F1 都将出现帮助索引。用户可以寻找所需的關鍵字,例如,用户知道打印命令是由 Pri 组成。在每键入一个字符,将会出现一个列表,开头是 P,然后是 Pr,最后是 Pri,等等。
Alt+F1	按 Alt+F1 连续滚屏 20 次。
Ctl+F1	显示 THELP 热键。
Ctrl+P	把标记好的块或文本范例粘贴到用户当前的应用程序中去。
热键	缺省为数字键盘上的 5 键,按它将显示 THELP 的所有热键和热键组合。

注意:若用户在 AT6300 上使用 THELP,就必须使用/L25 选项,详细情况请参阅下节。

B.4.2 THELP 选项

表 B.11 是 THELP 命令选项的概要。若用户要使用多个选项,则各选项间必须以空格隔开。

表 B.11 THELP 选项

选项	规定
/C#XX	选择颜色; # = 颜色号码; XX = 颜色的十六进制变量。
/D	显示宽度为 80 列的窗口。
/Fname	帮助文件的全路径和文件名。

续表 B.11

选项	规定
/H, /?, ?	显示帮助屏。
/KXXYY	改变热键: XX=移位状态(十六进制); YY=扫描码(十六进制)。
/S [+ -]	(+)表示增加检测雪花功能(对老的 CGA 极有用)或(-)表示不需检测雪花功能。
/U	在内存中删去 THELP。
/W	把选项写到 THELP.COM(并存储)再退出。

B.4.2.1 /C#XX 选择(选择颜色)

这个选项提供给用户在帮助屏上选择自己喜欢的前景颜色和背景颜色。

/C 选项后面跟着用户选择颜色的号码以及前景色和背景色的十六进制值。

这里有 12 种可能的颜色,其号码如下:

号码	组成元素
0	彩显边缘属性
1	单显边缘属性
2	彩显文本属性
3	单显文本属性
4	彩显关键属性
5	单显关键属性
6	彩显被选择关键字属性
7	单显被选择关键字属性
8	彩显例子文本属性
9	单显例子文本属性
A	彩显标识块属性
B	单显标识块属性

标准 IBM 以及兼容机的颜色值如下所示:

第一个数字(背景)	第二个数字(前景)
0 黑色	0 黑色
1 蓝色	1 蓝
2 绿色	2 绿
3 青色	3 青色
4 红色	4 红
5 粉红色	5 粉红色
6 棕色	6 棕色
7 灰色	7 灰色
	8 高亮黑
	9 高亮蓝

A	高亮绿
B	高亮青色
C	高亮红
D	高亮粉红
E	高亮棕(黄)
F	高亮灰(白)

用颜色值和十六进制数 80 进行或操作,得到的便是闪烁的颜色值。

对于单色显示器,字符属性值也许有很大的区别,这需用户去实践。

B. 4. 2. 2 /Fname (帮助文件的完整路径和文件名)

在选项/F 后面的 name 是一个带有完整路径的文件名,例如:

```
THELP /FC:\TP\TURBO.HLP
```

```
THELP /FC:\BORLANDC\TCHELP.TCH
```

缺省情况,THELP 将在登录的目录下寻找帮助文件。

B. 4. 2. 3 /H,/? 和? (显示帮助屏)

这些选项将显示 THELP 命令行选项的简要说明。

B. 4. 2. 4 /K xxyy (重新设置热键)

此选项提供给用户自己设置新的热键。该选项后面必须跟新键的移位状态(XX)和扫描码(yy)。有许多移位状态/扫描码的组合可供选择。这里有一些简单通用的移位状态和扫描码。

移位状态

右 SHIFT	01h
左 SHIFT	02h
Ctrl	04h
Alt	08h

扫描码

A	10h	N	31h	0	0bh	F1	3bh
B	30h	O	18h	1	02h	F2	3ch
C	20h	P	19h	2	03h	F3	3dh
D	20h	Q	10h	3	04h	F4	3eh
E	12h	R	13h	4	05h	F5	3fh
F	21h	S	1fh	5	06h	F6	40h
G	22h	T	14h	6	07h	F7	41h
H	23h	U	16h	7	08h	F8	42h
I	17h	V	2fh	8	09h	F9	43h
J	24h	W	11h	9	0ah	F10	44h
K	25h	X	2dh				
L	26h	Y	15h				
M	32h	Z	2ch				

增强型键盘可以使用下列键(也许在所有的计算机和键盘上都使用不了):

F11	57h
F12	58h

B.4.2.5 /S (雪花检查)

有些老的 CGA 显示器在软件直接写到显示内存时就会产生“雪花”的效果。如果用户看见这种情况,就可以在 THELP 中用 /S+来进行雪花检测。用户也可以使用 /W 来设置这种功能为永久性的。雪花检测将花费一些时间,若不进行也许更实惠些。屏蔽雪花检测用 /s-;这也是缺省值。

B.4.2.6 /U (在内存中删去 THELP)

此选项将从内存中删去 THELP。如果有其它程序在 THELP 之后驻留,则应确保在删去 THELP 之前删去它们。

B.4.2.7 /W (把选项写入 THELP.COM 中再退出)

使用 /W 选项则产生一个新的 THELP 版本,其中的缺省选项由用户自己设置。用户可以设置所有选项为永久性的。

B.5 GREP 查找程序

GREP(Global Regular Expression Print)是一个功能强大的文本搜寻程序,和 UNIX 中的实用程序同名。GREP 可以在一个或多个文件中搜寻匹配模式,也可以在当前输入的字符串中匹配。

这里有一个快速简便的例子可以帮助用户弄清楚如何使用 GREP。假设用户想在当前目录下搜寻在哪个文件中包含有字符串“Elisabeth”,就可以键入如下命令:

```
grep Elisabeth *.txt
```

于是 Grep 将产生一个列表,把所有包含“Elisabeth”字符串的文件以及所在的行号列出来。由于在缺省情况下 GREP 是要考虑大小写的,故而“elisabeth”和“ELISABETH”被认为是不同的,用户可使用选项来使在搜寻时忽略大小写。

GREP 所做的工作不仅仅是匹配单个字符或字符串,在后面的章节中,用户将明白如何利用 GREP 来搜寻特殊的字符串。

B.5.1 命令行形式

GREP 的命令行形式如下:

```
grep [options] searchstring [file(s) ...]
```

选项包括一个或多个字符,冠以“-”,这样用户就可以组合出 GREP 的许多不同的功能。

搜寻字符串(searchstring)告诉所要匹配的字符串。

文件名(files)告诉 GREP 所要搜索的是哪些文件(如果用户没有指明一个文件,则 GREP 搜寻当前的输入字符串;这样就可以使用户像使用管道和重定向一样使用 GREP)。如果用户发现 GREP 的输出结果很长,超出一屏,则可利用重定向使结果输出到一个文件中。例如,可使用如下命令:

```
GREP "Elisabeth McInnis" *.txt > gfile
```

这样将在用户的当前目录下,搜寻所有的以.txt 结尾的文件,看是否存在着“ElisabethMcInnis”,并将产生的结果放入一个文件名 GFILE 的文件中(也可以使用其它别的文件名)。这样,就可以利用字处理器调入 GFILE 文件来观察搜寻的结果。

命令:GREP ?

将显示 GREP 的命令行选项的简洁的帮助屏幕、特殊字符和缺省选项(请留意-U 命令行选项如何改变 GREP 的缺省值的信息)。

B. 5.2 GREP 的选项

在命令行中,选项是一个或多个单个字符,我们通过字符‘-’隔开。每个单独的字符都是一个开关,可以选择开和关:在选项字符后加一个“+”号表示此选项打开;若加的是一个‘-’号则关闭此选项。‘+’可以省略,例如:-r 就意味着-r+。用户可以单独使用选择字符(如:-i -d -l),也可以组合使用(如:-ild 或-il -d 等等);这些所产生的效果在 GREP 中完全一样。

表 B. 12 是 GREP 选项字符以及它们的意义:

表 B. 12 GREP 选项字符

功能	意义
-C	只计算匹配数目,显示的仅仅是所匹配的行数。对于每一个至少包含一个匹配行的文件,GREP 将显示它的文件名以及所配置的数目。而所匹配的行并不显示。此选项缺省时关闭。
-d	搜寻子目录:对在命令行中的指定的每个文件,GREP 将在指定的目录以及此目录下的所有子目录内进行搜寻。如果没有给出路径,GREP 则认为是在当前目录。缺省此选项被关闭。
-i	忽略大小写:GREP 中大小写字母是不同的,如选择了此选项,那么所有的小写字母 a~z 和大写字母 A~Z 就不存在区别,就忽略了大小写。缺省表示关闭此选项。
-l	另列出文件名:显示的仅仅是每一个包含有匹配模式的文件名。当 GREP 发现一个匹配模式后,显示文件名并立刻转入搜索下一个文件,缺省表示关闭此选项。
-n	行号:GREP 只显示所匹配的行的行号,缺省表示关闭此选项。
-o	UNIX 输出格式:若要改变所匹配行的输出格式,可以利用 UNIX 系统中方便有效的命令行管道。所有输出行被包含匹配行的文件名所分隔。缺省则表示关闭该选项。
-r	正则表达式的搜索:被定义要搜索的再也不是一个字符串而是一个正则表达式。缺省表示该项有效。 一个正则表达式是由可以多次出现的,被引号括起来的一个或多个字符组成,下列符号具有特殊的含义:
	^ 行首
	\$ 行尾
	. 任意字符
	[] 下个字符不转义

- * 匹配多次或零次
- + 匹配一次或多次
- [aiou 0-9] 匹配 a,e,i,o,u 以及 0~9
- [^ aeiou 0-9] 匹配 a,e,i,o,u 以及 0~9 以外的所有字符
- U 修改选项: GREP 可以在命令行设置缺省选项,并可以把这些选项写到 GREP.COM 文件中当作一个新的缺省选项(换句话说,GREP 可以自行配置),这个选项可以使用户凭自己的爱好来设置缺省选项。如果用户想了解 GREP.COM 中的缺省选项,则在 DOS 提示符下键入:

```
GREP ?
```

 在帮助屏上的选项依赖其缺省设置,显示成 a- 或 a+。缺省表示关闭该选择。
- V 不匹配:显示的是不匹配的行,那些不包括要搜寻字符串的行被认为是不匹配的行。缺省表示关闭该选项。
- W 搜寻词汇:被正则表达式匹配的文本,当其前面的字符或者后续字符不是构成一个单词的字符时,才被认为是一个匹配。单词缺省设置包括 A 到 Z,0~1,以及下划线(_)。缺省表示关闭此选项。
- Z GREP 打印每个被搜索的文件,并打印出匹配行,在其前冠以行号。并列出每个文件总匹配行数,即使匹配数为 0,也打印出统计行。选项缺省时是关闭的。

B. 5.3 正常的优先次序

记住每一个选项都是一个开关:它的状态取决于用户的最后一次设置。每一个时刻,任何一个选项只能是开或关,在命令行中出现某选项,则忽略该选项的所有前面的定义。如果在命令行

```
grep -r -i -d -i -v - main (my *.C)
```

中,GREP 打开 -d -i 选项,但关闭 -V 选项。

用户可以用 -U 选项在 GREP.COM 文件中对每一个选项设置缺省值。例如,如果用户想使 GREP 总显示详细的搜索信息(设 -Z 选项为开状态),则可用如下命令:

```
grep -U -Z
```

B. 5.4 搜寻字符串

为了使搜寻成功,用户必须正确地键入所要搜寻的字符串。变量 <searchstring> 被定义成要搜寻的字符串,一个被搜寻的字符串既可以是一个字母字符串也可以是一个正则表达式。

在正则表达式中,某些字符具有特定的含义:它们是控制如何进行搜索的操作符。

在字母字符串中,没有操作符;每个字符都是字母。

用户可以把所要搜寻的字符串用引号括起来,用此来防止把在搜索字符中的空格或制表键当成分隔符。在文本中不能匹配跨过行边界的搜寻字符串,故而所有要搜寻的表达式都必须在一个单行中。

正则表达式不仅可以是单个字符,也可以是由方括号括起来的一组字符序列。多个正则表达式的组合仍然是一个正则表达式。

B. 5.5 正则表达式的操作符

当用户使用 `-r` 选项(缺省表示开),所搜索的字符串是一个正则表达式(不是一个字母字符串),表 B. 13 字符具有特殊的含义:

表 B. 13 正则表达式操作符

选项	意义
<code>^</code>	在表达式开头的界符,用来匹配每一行的行首。
<code>\$</code>	在表达式末尾的界符,用来匹配每一行的行尾。
<code>.</code>	点匹配任何单个字符。
<code>*</code>	一个表达式后接了一个星号通配符,则匹配零个或多个这个表达式。例如,在 <code>to*</code> , <code>*</code> 操作符在 <code>o</code> 的后面,它将匹配 <code>t</code> , <code>to</code> , <code>too</code> ,等等(<code>t</code> 后面跟零个或多个 <code>o</code>),但不匹配 <code>ta</code> 。
<code>+</code>	一个表达式后接了一个加号则表示匹配一个或多个此表达式; <code>to+</code> 匹配 <code>to</code> , <code>too</code> 等,但不匹配 <code>t</code> 。
<code>[]</code>	被方括号括住的字符串,表示匹配在方括号中的任一字符。如果此字符串是以 <code>(^)</code> 开头的,则匹配所有不在方括号内的字符。 例如 <code>[xyz]</code> 匹配 <code>x</code> , <code>y</code> 或 <code>z</code> ,但是 <code>[^x,y,z]</code> 匹配 <code>a</code> 或 <code>b</code> ,但就不匹配 <code>x</code> , <code>y</code> 和 <code>z</code> 。用户也可以使用 <code>(-)</code> 来表示两个字符之间的所有字符,这样可以简化表达式(如 <code>[a-bd-z?]</code> ,表示匹配 <code>?</code> 以及除 <code>c</code> 以外的所有小写字母。
<code>\</code>	反斜杠将使 GREP 去搜寻在此之后的字符。例如, <code>\.</code> 不是匹配“所有的字符”而匹配“.”,它也可以引用它本身,也就是说,用户可以使用 <code>\\</code> 来指示匹配在 GREP 表达式中的一个真正反斜杠字符。

注意:四个“特殊”的字符(`$`,`.`,`*` 和 `+`),当不在正则表达式中使用,则不具备这些特殊的意义。另外,字符 `^` 仅出现在表达式的开头才具有特定的意义(即只能紧跟在字符 `[` 之后)。

其它的字符就不一一列在表中,它们只匹配本身,例如,大于号 `>` 匹配的就是大于号 (`>`),`#` 匹配 `#`,依此类推。

B. 5.6 文件说明

`<file(s)>` 将告诉 GREP 将要搜寻哪些文件或文件组。`<file(s)>` 既可以为文件的确切名,也可以包含 DOS 的通配符 `?` 或 `*`。另外,也可在文件前加入路径(驱动器号和目录信息)。如果文件前没有路径,则 GREP 搜寻当前的目录。如果用户没有指定特定的搜寻文件,则必须使用重定向符(`<`)或管道(`|`)。

B. 5.7 GREP 使用示例

下面一些例子显示了如何组合 GREP 的功能去干不同的搜寻工作。在这些例子中 GREP 的缺省设置并没有改变。

B. 5.7.1 例子 1

搜寻字符串在这儿告诉 GREP 去搜寻单词 `main`,在单词 `main` 的前面一个字符是小写字母(`[^a-z]`),其后可以出现零次或多次空格(`\ *`)、左括号的字符串。

由于空格键和制表符通常被认为是命令行的分隔,如果想把它们作为正则表达式的一部分必须用双引号括住它们,在本例中,main 后面的空格用反斜杠字符后跟空格键表示,也可以把空格放在双引号里。

命令行:

```
grep -r [^ a-z]main\ * ( *.C
```

匹配:

```
main (i; interger)
main (i,j; in+eger)
if (main ()) halt;
if (MAIN ()) halt;
```

不匹配:

```
mymain()
```

搜索文件:

在当前目录下的所有.C 文件。

B. 5. 7. 2 例子 2

因为反斜杠(\)和点(.)在路径和文件名中具有特定的含义,所以如果用户想搜寻它们就必须在其前面加上反斜杠。此处使用了一i 选项,这样在搜寻时忽略大小写。

命令行:

```
grep -ri [a-c];\\data\\. fil *.c *.inc
```

匹配:

```
A:\data.fil
c:\Data.Fil
B:\DATA.FIL
```

不匹配:

```
d:\data.fil
a:data.fil
```

搜寻文件:

在当前目录中的 *.C 和 *.INC 文件。

B. 5. 7. 3 例子 3

本模式定义如何查找一个指定的单词。

命令行:

```
grep -ri [^ a-z]word [^ a-z] *.doc
```

匹配:

```
every new word must be on a new line.
MY WORD!
word--smallest unit of speech.
In the beginning there was the WORD, and the WORD
```

不匹配:

```
Each file has not least 2000 words.
```


He misspells to word as toward.

搜寻文件: 在当前目录下的所有 .DOC 文件

B. 5. 7. 4 例子 4

这种格式定义另外一种方式, 甚至是单个字母的单词搜寻。

命令行:

```
grep -iw word *.DOC
```

匹配:

every new must be on a new line however,

MY WORD!

word: smallest unit of speech which conveys

In the beginning there was the WORD, and

不匹配:

each document contains at least 2000 words!

He seems to continually misspell "toward" as "toword".

搜寻文件:

在当前目录的所有 .DOC 文件.

B. 5. 7. 5 例子 5

这个例子告诉用户如何去搜寻嵌有空格的字符串。

命令行:

```
grep "search string with spaces" *.doc *.C a:\work\my file. *
```

匹配:

This is a search string with spaces in it.

不匹配:

This search string has spaces in it.

搜寻文件:

当前目录下的所有 .DOC 和 .C 文件, 以及 A 驱动器中 WORK 目录下的 MYFILE 文件。

B. 5. 7. 6 例子 6

这个例子将搜寻一行末尾的“.”、“,”和“,”中的任一个字符。

在方括号中的双引号前加以反斜杠, 表明此时它只代表一个“”字符, 而不是其特殊含义, 否则另一个双引号将出现在表达式以外, 这也表明了如何将多个正则表达式联结成一个长的表达式。

命令行:

```
grep -rd "[.,'\"]" \*.doc 匹配:
```

He said hi to me.

Where are you going?

In anticipation of a unique situation,

Examples include the following:

"Many men smoke, but to man chu."

不匹配:

He said "Hi" to me

Where are you going? I'm headed to the

搜寻文件:

在当前驱动器下的根目录以及其所有子目录下的.DOC文件。

B.5.7.7 例子 7

这个例子忽略大小写并且只显示出至少匹配一次的文件的名称,以下三个示例表示相同命令选项的不同组合都具有相同功能:

命令行:

```
grep -ild " the " \*.doc
```

```
grep -i -l -d " the " \*.doc
```

```
grep -il -d " the " \*.doc
```

匹配:

Anyway, this is the time we have

do you think? The main reason we are

不匹配:

He said "Hi" to me just when I

Where are you going? I'll bet you've headed

搜寻文件:

在当前驱动器下的根目录及其子目录下的所有.DOC文件。

B.5.7.8 例子 8

这个例子重新定义合法字符集为单个赋值操作符“=”。它将匹配C中的赋值语句(它只是单个符号,但不是测试相等,测试相等必须用两个等号)。

命令行:

```
grep -w[=] = *.C
```

匹配:

```
i = 5;
```

```
j=5;
```

```
i += 5 不匹配;
```

```
if (i == t) j++;
```

```
/* ===== */
```

搜寻文件:

在当前目录下的所有.C文件。

B.6 其它实用程序

Turbo C 拥有众多的独立的实用程序,帮助用户处理C文件和目标模块。其中MAKE, TLIB和TLINK已在前面几章中说明。在这里描述余下的实用程序的功能和用法。

下面是实用程序的概述:

BGIOBJ	图形驱动程序和字体的转换实用程序。
CPP	预处理程序。
OBJXREF	目标模块的交叉引用表。
PRJCFG	配置文件中的信息,更新工程文件的选项,或者把工程文件转换成配置文件。
TOUCH	改变文件的日期和时间。
TRANCOPY	工程文件之间复制条目。
TRIGRADH	字符转换实用程序。

在本附录中,介绍上述实用程序的功能。

B. 6.1 BGIOBJ:图形驱动程序和字体的转换程序

在 Turbo C 中,用户可以把图形驱动程序文件和字体集(矢量字体文件)转换成目标文件(.OBJ)。转换后的文件可以连接到程序中,作为执行文件的一部分。这样,一方面避免了在运行时从盘上加载图形驱动程序和字体集文件,减少了加载代码,另一方面可以减少一执行文件所要附随的文件。

把驱动程序和字体文件直接连接到程序中具有以下优点:由于执行文件已经包含了所有(或大多数)需要的驱动程序和字体文件,因此,在运行时就不必存取盘中的上述文件了。但这样做也有一个缺点,即加长了执行文件的长度。

要把驱动程序和字体文件转换成可连接的目标文件,可使用 BGIOBJ.EXE 程序,其简单的用法为:

```
BGIOBJ Source_file
```

其中 Source_file 是待转换成目标文件的驱动程序或字体文件。创建的目标文件名与 Source_file 文件名相同,扩展名是 .OBJ。比如,EGAVGA.BGI 产生 EGAVGA.OBJ,而 SANS.CHR 产生 SANS.OBJ。

B. 6.1.1 把新的.OBJ 文件添加到 GRAPHICS.LIB 中

用户可以把驱动程序和字体的目标模块添加到 GRAPHICS.LIB 中,并可以在连接图形子程序时定位这些文件。如果不这样做,那么必须把这些文件添加到工程文件中的项目表中,也可在 TCC 命令行或 TLINK 命令行添加。

把驱动程序或字体目标模块添加到 GRAPHICS.LIB 中,可用下述命令行调用 TLIB

```
tlb graphics + object_file_name [+object_file_name]
```

其中 object_file_name 是用 BGIOBJ.EXE 创建的目标文件名(比如 CGA、EGA、VGA、GOTH 等)。。OBJ 扩展名是缺省的扩展名,在命令行中勿需指定。如果想节约时间,一次可添加几个模块。

B. 6.1.2 注册驱动程序和字体

在 GRAPHICS.LIB 中添加驱动程序和字体目标模块后,必须为要连接的驱动程序和字体注册。这样就要求在程序初始化图形系统(即调用函数 initgraph)之前,调用 registerbgidriver 和 registerbgifont 函数,以通知图形系统要使用这些文件,并保证在连接程序产生执行文件之时连接这些文件。

注册子程序接受一个参数,该参数可以是 graphics.h 中的符号名。如果注册子程序调用

成功,将返回一个非负值(更详细的信息请参考《库数参考手册》中关于 registerbgifont 和 registerbgdriver 的描述。)

表 B. 14 说明 registerbgdriver 和 registerbgifont 使用的符号名,C 中所有的驱动程序和字体文件都已包括。

表 B. 14 驱动程序和字体文件种类

驱动程序文件 (BGI)	registerbgdriver 符号名	字体文件 (* . CHR)	registerbgisont 符号名
CGA	CGA_driver	TRIP	triplex_font
EGAVGA	EGAVGA_driver	LITT	Small_font
HERC	HERC_driver	SANS	ganserif_font
ATT	ATT_driver	GOTH	gothic_font
PCE3270	PC3270_driver		
IBM8514	IBM8514_driver		

B. 6. 1. 3 范 例

假设用户要将 CGA 驱动程序、哥特式字体(Goth)和 Triplex 字体转换成目标模块,并把它们连接到程序中,那么可以按下面步骤完成:

- 使用 BGIOBJ. EXE 把二进制文件转换成目标文件,在命令行下键入下面三行:

```
bgiobj cga
bgiobj trip
bgiobj goth
```

执行完后建立了三个文件: CGA. OBJ、TRIP. OBJ 和 GOTH. OBJ。

- 用 TLIB 将上述三文件添加到 GRAPHICS. LIB 中:


```
tlib graphics +cga +trip +goth
```
- 如果不把这些文件加到图形库中,则必须在集成环境(或在命令行编译器中)把 CGA. OBJ、TRIP. OBJ 和 GOTH. OBJ 添加到工程文件中。比如可用下面的 tcc 命令行:


```
tcc niftgraf graphics. lib cga. obj trip. obj goth. obj
```
- 在用户的程序中用户应按下面的模式注册这些文件。如果出现"Segment exceeds 64K"的连接错误信息,请参见以下内容。

```
/* Header file declares CGA_driver, triplex_font & gothic_font */
#include <graphics.h>
/* Register and check for errors (one never knows...) */
if (registerbgdriver(CGA_driver) < 0) exit(1);
if (registerbgifont(triplex_font) < 0) exit(1);
if (registerbgifont(gothic_font) < 0) exit(1);
/* ... */
initgraph(...); /* initgraph should be called after registering */
/* ... */
```

B. 6. 1. 4 /F 选项

如果在连接几个驱动程序和字体文件(特别是小和紧缩模式)时出现了"Segment

exceeds64K”或类似的连接错误,那么下面的步骤可用来处理此情况。

在缺省时,由 BGIOBJ. EXE 所创建的文件都使用相同的段名,即 _TEXT,当程序使用许多驱动程序和字体文件,或用户使用小的或紧缩内存模式时,就会发生前述的错误。

要解决该问题,可用 BGIOBJ 的 /F 选项将一个或更多个驱动程序或字体文件转换为目标文件,该选项使 BGIOBJ 使用 filename TEXT 的段名,因此被连接的驱动程序和字体文件(即使在小内存模式和紧缩模式也如此)不使用缺省段名。比如下面的两个 BGIOBJ 命令行使 BGIOBJ 使用 EGAVGA _TEXT 和 SANS _TEXT 的段名。

```
bgiobj /F egavga
```

```
bgiobj /F sans
```

在使用 /F 后, BGIOBJ 还把“F”添加到目标文件名之后(如 EGAVGAF. OBJ, SANSF. OBJ 等),并把 _far 添到被 registerfarbgidriver 和 registerfarbgifont 使用的符号名之后(比如 EGAVGA _driver 变成 EGAVGA _driver _far)。

对于用 /F 选项创建的文件,用户必须使用远的(far)注册子程序,而不能采用通常的 registerbgidriver 和 registerbgifont。比如:

```
if (registerfarbgidriver(EGAVGA _driver _far)<0) exit(1);
```

```
if (registerfarbgifont(sangerif _font _far)<0) exit(1);
```

B. 6. 1. 5 高级性能

上面的内容是相对于系统内配置的驱动程序和字体文件的,因此以上用法只能是专用的。BGIOBJ 有更一般的用法,该用法从某种程度上来说,是一个相当高级的性能,只有那些有相当多编程经验的程序员才能使用下面将要讲述的用法。

下面是 BGIOBJ. EXE 命令行的完整语法:

```
BGIOBJ [/F] source destination public _name seg _name seg _class
```

下面表描述命令行的每个部分:

元素	说明
/F 或 -F	选项使 BGIOBJ. EXE 使用自行指定的段名而不是缺省段名 _TEXT,修改公共名和目标文件名(有关 /F 更详细的信息参见上一节)。
<source>	这是将要被转换的驱动程序或源文件。如果该文件不是 Turbo C 软件提供的驱动程序或字体中的一个,则应当给出完整的文件名(包括扩展名)。
<destination>	这是生成的目标文件名。缺省的目标文件名是 SOURCE. OBJ,如果使用 /F 选项则为 sourceF. OBJ。
public _name	该名字用于在程序中调用 registerbgidriver 或 registerbgifont(或它们相应的其它版本)连接目标模块。 该公共名是用于连接的外部名,因此它在程序中使用时要加下划线。但如果程序用的是 Pascal 调用规则,就不用加下划线。
seg _name	该选项是一个段名,其缺省值为 _TEXT(如果指定了 /F 选项则为 filename _TEXT)。
seg class	该选项是一个段类别,其缺省值为 CODE。

除了 source 之外,所有的参数都是可选的。但如果需要指定某一选项,则须先指定所有位于它前面的选项。

如果用户选择了自己的公共文件名,则应当用下列的格式向程序添加声明:

```
void public _name (void);          /* 如果不使用/F */
                                   /* 则用缺省段名 */
extern int for public _name[];     /* 如果用/F, 或 */
                                   /* 段名不是_TEXT */
```

在这些声明中,当使用 BGIOBJ 进行转换时,程序中的 public _name 与这里的 public _name 就是相同的。graphics.h 头文件包含缺省驱动程序和源公共名的声明;如果使用缺省公共名,则就不必像前面描述的那样进行声明了。

在声明之后,还要在程序中注册所有的驱动程序和字体文件。如果不使用/F 选项并且不修改缺省段名,则应当用 registerbgidriver 和 registerbgifont 注册驱动程序和字体文件;否则用 registerfarbgidriver 和 registerfarbgifont 子程序注册。

下面是一个向内存装载字体文件的一个示例程序:

```
/* Example of loading a font file into memory */
#include <graphics.h>
#include <io.h>
#include <fcntl.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <process.h>
#include <alloc.h>
main()
{
    void * gothic_fontp; /* points to font buffer in memory */
    int handle;          /* file handle used for I/O */
    unsigned fsize;      /* size of file (and buffer) */
    int errorcode;
    int graphdriver;
    int graphmode;
    /* open font file */
    handle = open("GOTH.CHR", O_RDONLY|O_BINARY);
    if (handle == -1)
    {
        printf("unable to open font file 'GOTH.CHR'\n");
        exit(1);
    }
    /* find out size of the file */
    fsize = filelength(handle);
    /* allocate buffer */
```

```

gothic_fontp = malloc(fsize);
if (gothic_fontp == NULL)
{
    printf("unable to allocate memory for font file 'GOTH.CHR'\n");
    exit(1);
}
/* read font into memory */
if (read(handle, gothic_fontp, fsize) != fsize)
{
    printf("unable to read font file 'GOTH.CHR'\n");
    exit(1);
}
/* close font file */
close(handle);
/* register font */
if (registerfarbgifont(gothic_fontp) != GOTHIC_FONT)
{
    printf("unable to register font file 'GOTH.CHR'\n");
    exit(1);
}
/* detect and initialize graphix */
graphdriver = DETECT;
initgraph(&graphdriver, &graphmode, "..");
errorcode = graphresult();
if (errorcode != grOk)
{
    printf("graphics error: %s\n", grapherrormsg(errorcode));
    exit(1);
}
settextjustify(CENTER_TEXT, CENTER_TEXT);
settextstyle(GOTHIC_FONT, HORIZ_DIR, 4);
outtextxy(getmaxx()/2, getmaxy()/2, "Borland Graphics Interface (BGI)");
/* press a key to terminate */
getch();
/* shut down graphics system */
closegraph();
return(0);
}

```

B. 6.2 CPP: 预处理实用程序

CPP 产生一个 C 源程序表(在文件中),在 C 源程序表中把包含文件和扩展的 #define 宏扩展开来。它不用于 C 程序的普通编译程序。

通常,当编译程序在编译宏或包含文件输出一个错误信息时,如果可以查看到包含文件或宏扩展后的具体内容,则用户就可以得到更多的错误信息。在多遍编译程序中,一般总有某一遍扫描将执行该任务,并且其结果是一些中间文件。因为 Turbo C 使用的只是一个单遍扫描编译程序,所以用户可以预先使用 CPP 来完成其他编译方式中头遍扫描的功能。

用户可以像使用独立的编译器 TC 一样来使用 CPP。CPP 和 TC 的缺省项都可以通过 TURBO.CFG 文件进行设置,而且也像 TC 一样接受相同的命令行选项参数。

TC 的那些不从属于 CPP 的选项,只是被 CPP 省略而已,在 DOS 提示符下键入 CPP,便可看到参数的列表。

在 CPP 命令行中文件的列表可以和在 TC 中那样使用通配符来处理。这里有一点不同的是 CPP 把所有的文件都作为 C 的源文件处理,而并没有什么特殊的方法来处理 .OBJ、.LIB 和 .ASM 文件。

经过 CPP 处理过的文件,其输出都写在当前目录下的文件中(或者用 -n 选项来定义输出目录),其文件名与源文件名相同,只不过其扩展名为 .I。

输出文件是一个文本文件,包括了源程序及其所有的包含文件中的每一行,其中所有的预处理指令和预处理条件语句都被删除了。除非使用命令行选项另作说明,否则,文本文件中的每一行行头都包括了源文件名以及该行在源文件或包含文件中所在的行号,在文本文件的每一行中,所有的宏都被实际定义的代码内容所代替。

注意:由 CPP 生成的结果文件不能进行编译,因为文件中行头还包括了文件名以及行号。

CPP: 一个宏预处理器

使用 CPP 的 -P 选项可以使每一行的行前都加上其在源文件中的行号,如果使用 -P- (即关闭此选项),CPP 将省略行号信息。当此选项关闭后,CPP 就可被用作一个宏预处理器;其结果文件 .i 文件也能被 TC 所编译。

下列利用一个简单的程序来说明如何利用 CPP 来预处理一个文件,先使用 -P 选项,然后使用 -P- 选项。

```
#define NAME "AJ McInnis"
#define BEGIN {
#define END   }
main()
BEGIN
    printf("%s\n", NAME);
END
```

若要像使用预处理器一样使用 CPP, 则使用命令行:

CPP HELLOAJ.C

输出:

```
HELLOAJ.c 1:
HELLOAJ.c 2:
HELLOAJ.c 3:
HELLOAJ.c 4:
HELLOAJ.c 5: main()
```



```
HELLOAJ.c 6: {
HELLOAJ.c 7:     printf("%s\n", "AJ McInnis");
HELLOAJ.c 8: }
```

若要像使用宏预处理器一样使用 CPP, 则使用命令行如下:

```
CPP -P - HELLOAJ.C
```

输出:

```
main()
{
    printf("%s\n", "AJ McInnis");
}
```

B. 6.3 OBJXREF: 目标模块的交叉引用列表实用程序

OBJXREF 将检查目标文件和库文件, 以生成一些信息报表。其中一类报表列出了公共名的定义与引用, 另一类报表列出了由目标模块定义的段大小。

有两类公共名: 全程变量和函数名。在“B. 6.3.4 的 OBJXREF 样本报表”一节中的 TEST1.C 和 TEST2.C 程序用来说明公共名的定义以及外部对的公共名的引用。

目标模块是由 TC、TCC 或 TASM 所产生的 .OBJ 文件。一个库文件(.LIB)包含多个目标模块。一个由 TC 所产生的目标模块和编译的 .C 源文件具有相同的名字, 只是扩展名不同, 在 TCC 中也一样, 除非在 TCC 命令行中使用 -O 选项来指定输出到特定的文件。

B. 6.3.1 OBJXREF 命令行

OBJXREF 命令行由 OBJXREF 和跟在其后的一组命令行选项, 以及一组以空格或制表符隔开的目标文件和库文件的列表组成。它的形式如下:

```
OBJXREF options filename filename
```

命令行选项决定了报表的种类, 如何决定将在后面的小节中详尽地讨论。

每一个选项都以斜杠(/)开头, 后面跟一个或两个选项字母。

目标文件和库文件既可在命令行中指出, 也可在响应文件中定义。在命令行中, 文件名以空格或制表符隔开。所有指定的 .OBJ 文件的目标模块都包括在报表中。像 TLINK 一样, OBJXREF 只包括由命令行指定的 .OBJ 或 .LIB 文件引用的库中模块的公共名。

作为一个通用的规则, 用户应列出所有使程序正确连接所需的 .OBJ 和 .LIB 文件, 包括启动代码的 .OBJ 文件以及一个或多个 C 语言库。

文件名可以包括驱动器以及目录路径。DOS 的 ? 和 * 通配符用来定义多个文件。文件必须是 .OBJ 目标文件或 .LIB 库文件(若用户不给文件的扩展名, 就假设成 .OBJ 扩展名)。

在命令行中选项和文件名可以以任意次序出现。

OBJXREF 报表将写到 DOS 系统的标准输出, 缺省时输出到屏幕。报表也可以送入打印机(如用 >LPT1:) 或写入一个文件(例如 >stfile), 这用到 DOS 的重定向符(>)。

如果只键入 OBJXREF, 没有文件名和选项, 则产生一组简明的选项一览表。

B. 6.3.2 OBJXREF 命令行选项

OBJXREF 命令行选项分为两类: 控制选项和报表选项。

B. 6.3.2.1 控制选项

控制选项表 B. 15 修改 OBJXREF 的缺省作用(缺省值指这些选项未选中时的值)。

表 B.15 控制选项

选项	意义
/I	忽略公共名的大小写不同。如果使用 TLINK 时未用 /C 选项,则须用此选项。TLINK 的 /C 选项使大小写具有不同含义。
/D	在其它目录下寻找 OBJ 文件。如果用户想当前目录以外的其它目录下寻找 OBJ 文件,OBJXREF 就必须在命令行中加入目录名,并且在前面加 /D,例如: OBJXREF /Ddir [, dir2 [, dir3]] 或 OBJXREF /Ddir [/Ddir2] [/bdir3] OBJXREF 将在指定的目录下搜寻所有的目标文件以及库文件。 注意:如果用户没有使用该选项,OBJXREF 将只在当前目录下搜寻。如若用户使用了 /D 选项,当前目录不一定被搜索,除非在目录列表中包含了它。例如,如果用户想用 OBJXREF 开始对 BORLAND 目录进行搜索,然后再到当前目录下搜索文件,用户可键入: OBJXREF /Dborland;. 其中 "." 代表当前目录。
/F	包括整个库。所有指定的 LIB 文件的所有模块都被包括,即使它们没有由 OBJXREF 处理的模块所引用的公共名也一样(请参阅后面章节中的例 4)。
/O	允许用户定义一个文件,将 OBJXREF 的输出写到那个文件中。格式如下: OBJXREF filename.obj/report option/Doutputfilename.ext 缺省表示输出到屏幕上。
N	全部输出、读入文件的列表并显示所有的公共名、模块、段名以及类。
/Z	包括零字节长度的段定义。模块定义时可以不为之分配任何空间,列出那些长度为零字节的段通常会使得模块大小报表难以使用,但是这样对用户想删除一个段的所有定义是有用的。

B.6.3.2.2 报表选项

在表 B.16 中报表选项控制产生报表的种类,以及 OBJXREF 所提供信息的详细说明。

表 B.16 报表选项

选项	产生的报表
/RC	产生类报表,按段的类类型排列出的模块长度。
/RM	产生模块报表,按定义模块顺序的公共名。
/RP	产生公共名报表,以定义的模块名的顺序列出公共名。本项是缺省的。
/RR	产生引用报表,按名顺序的公共名定义和引用。
/RS	产生了模块大小报表,按段名顺序的模块大小。
/RO	产生未引用的符号名列表,由定义好的模块为顺序列出未引用的公共名。
/RV	全部报表,OBJXREF 产生每一种类型的报表。
/RX	产生外部引用报表,按引用模块名顺序的外部引用。

在 C 文件中定义的公共名,出现在报表中加了一个前缀下划线(_),例如 main 表示为 _main。若不想加上下划线,在文件被编译时使用 -U 选项。

用户可以将段名、模块名、类名或者公共名的报表送入一文件,该文件放在命令行上出现的、前缀为 /N 字符串的文件名中。例如:

OBJXREF filelist /RM /NCO

OBJXREF 只产生一个名为 CO 的模块报表信息。

B. 6. 3. 3 响应文件

DOS 的命令行最多一行只能有 128 个字符。如果用户的选项和文件名列表超过了这个限制,就可以将它们放在响应文件中。

响应文件是一个文本文件,可以用文本编辑器进行编辑,用户可以建立包含许多 Turbo C 程序多个列表的文件,OBJXREF 可以识别几种类型的响应文件。

可以通过使用下列选项从命令行中调用响应文件。响应文件名必须直接跟在选项后,没有空格(例如,用户可使用 /Lresp,但不能用 /L resp)。

用户也可以在命令行中指定多个响应文件;其中,OBJ 和 LIB 文件可放置在它们之前,也可在它们之后。

B. 6. 3. 3. 1 自由形式的响应文件

用户可以用文本编辑器产生一个自由形式的响应文件,只需把建立 EXE 文件所需的,OBJ 和 LIB 文件名列出来。

在响应文件中任何一个没有扩展名的文件名列表都被认为是 OBJ 文件。在 OBJXREF 中使用自由形式的响应文件,只要把每个响应文件列在命令行中,其前缀为 @,用空格或制表符将它们和其它命令行参数隔开。如:

```
@file name @filename
```

B. 6. 3. 3. 2 工程文本

用户也可以把 Turbo C 集成环境生成的工程文件用作响应文件,在命令行中,在工程文件名前加上前缀 /p,例如:

```
/pfilename
```

如果文件名没有给出确定的扩展名,其缺省名为 PRJ。

在工程文件中以 C 扩展名结尾的文件或没有扩展名的文件都被解释成相应的 OBJ 文件。用户不必删除用括号来指出文件的依赖关系,OBJXREF 会自动忽略它们。

注意:通常,在 PRJ 中的文件名列表并不是一个完整的程序,用户必须指定启动代码文件(C0X.OBJ)和一个或多个 Turbo C 的库文件(MATHX.LIB,EMU.LIB,或者 CX.LIB 等等)。另外,当使用 OBJXREF 搜寻 OBJ 文件时,用户最好选用 /D 选项来指定搜索路径。

B. 6. 3. 3. 3 连接程序的响应文件

在 TLINK 中的响应文件格式也同样能被 OBJXREF 利用。一个连接程序的响应文件在命令行中的前缀为 /L,例如:

```
/Lfilename
```

为了了解如何利用这些文件,请参阅“B. 6. 3. 5”一节。

B. 6. 3. 4 OBJXREF 样本报表

假设用户在自己的 Turbo C 目录下有两个源文件,并且希望产生已编译的目标文件的一个 OBJXREF 报表。源文件分别为 TEXT1.C 和 TEST2.C,内容如下:

```
/* test1.c */
int i1;                /* defines i1      */
extern int i2;          /* refers to i2    */
```

```

static int i3;           /* not a public name */
extern void look(void);  /* refers to look */
void main(void)          /* defines main */
{
    int i4;              /* not a public name */
    look();              /* refers to look */
}
/* test2.c */
#include <process.h>
extern int i1;           /* refers to i1 */
int i2;                  /* defines i2 */
void look(void)          /* defines look */
{
    exit(i1);            /* refers to exit... */
}                        /* and to i1 */

```

从源文件编译得到的目标模块分别为 TEST1.OBJ 和 TEST2.OBJ。用户可在命令行中键入文件名,后面接 /R 和一个表示报表类型的字母,来告诉 OBJXREF 产生这些文件的是何种报表。

注意:例子只显示输出中的有用部分。

B. 6. 3. 4. 1 公共名报表(/RP)

公共名报表将列出在处理的目标模块内所定义的每个公共名,后面跟定义该公用名的文件名。

例如,键入命令行:

```
OBJXREF /RP test1 test2.
```

OBJXREF 将产生如下报表:

符号	文件名
_i1	TEST1
_i2	TEST2
_look	TEST2
_main	TEST1

B. 6. 3. 4. 2 模块报表(/RM)

模块报表列出所有的目标模块,后面跟有其内部定义的公共名。

如果键入如下命令:

```
OBJXREF /RM test1 test2
```

OBJXREF 将产生如下报表:

```

MODULE: TEST1 define the following symbols:
    public : _i1
    public : _main
MODULE: TEST2 define the following symbols:
    public : i2

```

```
public ; _look
```

B. 6. 3. 4. 3 引用报表(/RR)

引用报表将在同一行内列出括号所定义模块的公共名。引用此公共名的模块将在后面按左对齐的格式列出来。若报表类型缺省,表明是此类报表。

如果键入命令:

```
OBJXREF /RR CO test1 test2 CS. LIB
```

则 OBJXREF 产生如下报表:

```
_exit      (EXIT)
           C0
           TEST2
_i1         (TEST1)
           TEST2
_i2         (TEST2)
_LOOK      (TEST2)
           TEST1
_main      (TEST1)
           C0
```

B. 6. 3. 4. 4 外部引用报表(/RX)

此报表将列出每一个目标模块以及它引用的外部名。

如果键入命令行:

```
OBJXREF /RX C0 test1 test2. CS. LIB
```

则 OBJXREF 将产生如下报表:

```
MODULE; C0 reference the following symbols:
_main
MODULE; TEST1 reference the following symbols:
_i2
_Look
MODULE; TEST2 reference the following symbols
_exit
_i1
```

B. 6. 3. 4. 5 模块大小报表(/RS)

此报表将列出段名,后跟包含定义该段的模块表,模块的字节数将分别以十进制和十六进制给出,单词 uninitialized 出现在那些没有设置初始值的段的后面。所定义段的. ASM 文件中的绝对地址将被标志成 Abs,列在段大小的左边。

如果键入如下命令:

```
OBJXREF /RS test1 test.
```

OBJXREF 将产生如下报表:

```
These files were compiled using the large memory model.
TEST1 TEXT
```

```

        6 (00006h)    TEST1
        6 (00006h)    total
TEST2_TEXT
        10 (0000Ah)   TEST2
        10 (0000Ah)   total
-BSS
        4 (00004h)    TEST1, uninitialized
        2 (00002h)    TEST2, uninitialized
        6 (00006h)    total

```

B. 6. 3. 4. 6 类类型报表(/RC)

此报表将列出被类定义的类的大小。CODE 类包括指令部分, DATA 类包括它初始化的数据, BSS 类则包括那些未初始化的数据。不包含类的段将被列在“No class type”下面。

如果用户键入以下命令:

```
OBJXREF /RC C0 test1 test2 CS. LIB
```

OBJXREF 产生如下报表:

```

BSS
  4    (00004h)      TEST1
  2    (00002h)      TEST2
  ...
  ...
132   (00084h)      total
  6    (00006h)      TEST1
  10   (0000Ah)      TEST2
  16   (00010h)      total
143   (0008Fh)      C0
143   (0008Fh)      total

```

B. 6. 3. 4. 7 没有引用的符号名报表

此类报表将把那些在某模块中定义了,但未在其它模块中引用的公共名列出来。这类符号包括:

- 只在所定义的模块中引用,没有必要定义成公共符号,在这种情况下,如果模块是 C 编写的,则应在此符号前加上 static;如果是 TASM 编写的,则应把公共定义删除。
- 从未用过(这样的话,将它删除以节约数据存取空间)。

如果键入如下命令:

```
OBJXREF /RU test1 test2
```

OBJXREF 将产生如下报表:

```
MODULE; TEST2 defines the unreferenced symbol -i2.
```

B. 6. 3. 4. 8 全部报表(/RV)

如果在命令行中键入了/RV,则 OBJXREF 将产生上述类型的每一种报表。

B. 6. 3. 5 如何使用 OBJXREF 例子

所有这些例子都认为应用文件在隐含驱动器下的当前目录中, Turbo C 的 COX.OBJ 文件和其它库文件被认为是在 \TC\LIB 目录下。

B. 6. 3. 5. 1 例 1

```
C>OBJXREF \TC\lib\col test1 test2 \TC\lib\cl.lib
```

在此例中, 指定了 TEST1.OBJ 和 TEST2.OBJ 以及 Turbo C 的启动代码文件 TC\LIB\COL.OBJ 和库文件 BORLAND\LIB\CL.LIB。由于没有指明所要产生的报表类型, 得到结果报表为缺省默认的引用报表, 列出所引用的公共名以及它们所在的模块。

B. 6. 3. 5. 2 例 2

```
C>OBJXREF /RV /Ltest1.arf
```

TLINK 中的响应文件 TEST1/.ARF 包含着与例 1 同样的文件序列。因为使用了 /RV 选项, 所以将产生每一种类型的报表。TEST1/.ARF 包括:

```
\TC\lib\col  
test1.test2  
test1.exe  
test1.map  
\TC\lib\cl
```

B. 6. 3. 5. 3 例 3

```
C>OBJXREF/RC B;C0S /Ptest1 @@lib
```

Turbo C 的工程文件 TEST1.PRJ 包括了 TEST1.OBJ 和 TEST2.OBJ。响应文件 @lib 指定了库在 B 驱动器中。TEST1.PRJ 包括:

```
test1  
test2.C
```

文件 LIBS 包含

```
b; math.s.lib b;emu.lib b;cs.lib
```

被定义的启动代码文件和库文件依赖于编译时使用的存储模型和浮点选项。选项 /RC 产生类类型报表。

B. 6. 3. 5. 4 例 4

```
C>OBJXREF /F /RV \TC\lib\CS.lib
```

这个例子将产生 Turbo C 目录下 CS.LIB 文件的所有报表。即使被指定的不是一个完整的程序, OBJXREF 也能产生非常有用的报表。选项 /F 导致 CS.LIB 文件中的所有模块都被列进报表中。

B. 6. 3. 6 OBJXREF 的警告和出错信息

OBJXREF 产生两类诊断信息: 出错信息和警告信息。

B. 6. 3. 6. 1 出错信息

内存溢出, OBJREF 将在 RAM 内存中完成交叉索引, 这样就可能引起内存溢出, 即使 TLINK 有时候能成功地连接在列表中的同样文件。当这种情况发生时, OBJXREF 命令将被放弃执行。这时为了正常运行, 要移出目前不需要的内存驻留程序以获得更大的空间, 或者增大 RAM 的容量。

B. 6.3.6.2 警告信息

- **WARNING: Unable to open input file <filename>**
输入文件 filename 不存在或者没打开。OBJXREF 将继续处理下一个文件。
- **WARNING: Unknown option - <option>**
选项 option 不能被 OBJXREF 识别。OBJXREF 将忽略其选项。
- **WARNING: Unresolved symbol <symbol> in module <module>**
在模块 module 中所引用的公共符号未能在其它任一 OBJ 或 LIB 中指明。OBJXREF 将在其所产生的报表中标出此符号,认为它是被引用但未被定义的符号。
- **WARNING: Invalid file specification <filename>**
文件名序列中的某些文件名是无效的。OBJXREF 将继续对下个文件进行操作。
- **WARNING: No files matching <filename>**
在命令行中或在响应文件中的 filename 文件不存在或没打开。OBJXREF 将继续对下个文件操作。
- **WARNING: Symbol <symbol> defined in <module> duplicate in <module2>**
公共符号既在模块 module1 上定义,又在 module2 上定义了。OBJXREF 将忽略第二个定义。

B. 6.4 PRJCFG

该实用程序将工程文件转换成配置文件,用户也可通过它将配置文件转换成工程文件。

命令行编译器寻找一个名为 TURBOC.CFG 的隐含文件,当然用户也可通过路径选项指定不同的文件。利用 PRJCFG 可以从一个工程文件中产生一个 TC 或 BCX 的配置文件。

键入下列命令:

```
PRJCFG ProjFile. PRJ ConfigFile. CFG
```

若要从配置文件中产生一个工程文件,则键入:

```
PRJCFG ConfigFile. CFG ProjFile. PRJ
```

B. 6.5 TOUCH

当用户希望重新编译或重新构造一个特殊的目标文件时,虽然根本没有对源程序进行修改,也会改变记录文件的时间。还有一个方法就是通过 TOUCH 实用工具来改变文件的时间。TOUCH 将一个或多个文件的日期和时间设为当前的日期和时间,使它比原来的文件“更新”。

用户可以通过 TOUCH 来重新构造一个目标文件。在 DOS 提示符下,键入:

```
touch filename [filename ...]
```

TOUCH 将会更新文件的日期。一旦完成这些工作,用户就可以启动 MAKE 来重新构造新的目标文件。在 TOUCH 命令中也可使用 DOS 的通配符 * 和 ?。

注意: 在使用 TOUCH 程序之前,用户必须将所用系统的时钟设置正确。如果用户使用的是 IBMPC, XT 或其兼容机,由于它们并没有使用电池能量的时钟,所以此时不要忘记使用 DOS 的“time”和“date”命令来设置时间。只有这样才能使 MAKE 和 TOUCH 正常工作。

中国书画函授大学
建校二十周年纪念册



中国书画函授大学建校二十周年纪念册
定价：5.00元