



TURBO C

程序设计技术

潘金贵 沈默君 袁峰 等编写

陈世福 主审

南京大学出版社

TP312

P13

(2)

353073

TURBO C 程序设计技术

潘金贵 沈默君 袁 峰

谢俊元 王锡江 陈兆乾

编写

陈世福 主审



南京大学出版社

1991·南京

内 容 简 介

本书以 Turbo C 的最新版本为背景,详细介绍了美国 Borland 国际公司在 IBM PC 机上实现的一个高速、强功能的C语言系统。Turbo C 语言不仅完全支持 Kernighan 和 Ritchie 的C语言定义,而且与 ANSI 最新的C语言标准兼容并有所扩充。它提供了 450 多个库函数以及与 Turbo Prolog、Turbo Pascal 和汇编等多种语言的接口。

本书题材丰富,内容深入浅出,叙述较为系统而全面,详细介绍了 Turbo C 语言的功能,使用方法和各种程序开发技术,对与 Turbo Pascal、Turbo Prolog 和宏汇编的接口、以及正文窗口、图形处理等高级程序设计技术都作了专门介绍。

本书是学习 Turbo C 语言的一本较为实用的教材和教学参考书,适合广大从事计算机教学、研究和应用的教师、大学生、研究生以及计算机用户的程序人员阅读和参考。

JS194/11



TURBO C 程序设计技术

潘金贵 沈默君 袁 峰 编写
谢俊元 王锡江 陈兆乾
陈世福 主审

南京大学出版社出版
(南京大学校内)

江苏省新华书店发行 江苏省阜宁印刷厂印刷
开本: 787×1092 1/16 印张: 22 字数: 644千

1990年1月第2版 1991年5月第4次印刷

印数: 14001—24000

ISBN 7-305-00265-8

TP·19

定价: 7.00元

责任编辑: 顾其兵

前 言

C 语言是一种结构化、模块化、可编译的通用程序设计语言,被广泛地用于系统程序 和应用程序的开发。例如,著名的 UNIX 操作系统就是用 C 语言书写的。C 语言有着良好的可移植性,用 C 语言书写的程序在不同的计算机系统之间很容易实现转换。因此,几乎所有的程序设计任务均可使用 C 语言来完成。

Turbo C 是美国 Borland 国际公司在 IBM PC 机上实现的一个高效、优化的 C 编译程序。编译速度快,Turbo C 1.5 版每分钟可编译 10000 行源程序,编译时利用 RAM 存放中间数据结构,一趟扫描产生内部代码,只需要一次性对盘上的源文件读入和写出目标码(其他 C 语言编译程序则要 4-5 次读盘,每次执行一种功能),目标模块的格式与 PC-DOS 连接程序一致,能够和汇编程序连接。支持极小、小、中、紧凑、大和特大等六种存储模式。可以使用近、远指针的混合模式。

Turbo C 支持 IEEE 浮点标准,当没有 8087 或 80287 协处理器时,可使用系统提供的仿真 8087 或 80287 协处理器的实用程序进行高速的浮点运算。

Turbo C 提供了一个完整的交互式集成开发环境:

- 友善的用户接口。包括下拉菜单和多窗口,能够从集成开发环境中产生或运行一个可执行文件。

- 高效能的全屏幕编辑程序。当编译过程中出现错误时,编辑程序自动地用彩色光标指出源程序中适当的出错位置。

- 强有力的“Make”功能。使得 Turbo C 程序开发的管理十分容易。

- 快速的内部 Turbo 连接程序和内含式上下文敏感的帮助功能等。

Turbo C 提供了与 Turbo Prolog, Turbo Pascal 和汇编语言等多种语言的接口,能把用不同风格程序设计语言建立的目标模块连接成单一的程序。Turbo C 实现了美国国家标准局(ANSI)建议的 C 语言标准,完全支持 Kernighan 和 Ritchie 的 C 定义。提供了 450 个库函数和 6 个可独立运行的实用程序,支持 CGA、EGA、VGA、Hercules 和 IBM8514 等的高质量图形库。此外还包括了一些混合模式程序设计任选的扩充,进一步挖掘了 PC 机的能力。

Turbo C 提供了由用户选择的常规命令行版本和集成开发环境版本。只要有 384KB 以上内存,具有单软盘驱动器或双软盘驱动器,或者一个硬盘驱动器和一个软盘驱动器的 IBM PC 机,长城 0520 系列机或其他兼容机,在 DOS 2.0 版本 或其后版本的支持下,就可运行 Turbo C 语言,并且使得这台 PC 机工作起来就像是一台 80286 机。

南京大学计算机科学系已对 Turbo C (1.0 和 1.5) 两个版本进行了成功的汉化,为开发汉化应用软件提供了有力的工具。

本书根据美国 1987 年出版的几种关于 Turbo C 的书籍和配套软件编译和改写而成,共 12 章和 4 个附录。

各章安排和主要内容如下:

第一章: Turbo C 的安装和启动, 描述组成 Turbo C 系统的五张软盘的文件组织, 介绍如何在具体的计算机系统上安装 Turbo C 文件和库。

第二章: Turbo C 集成开发环境, 描述 Turbo C 的菜单系统和菜单命令, 以及如何使用交互式编辑程序建立和修改源程序文件。

第三章: Turbo C 程序的编译和运行, 介绍在集成开发环境下编译和运行 Turbo C 程序的步骤和方法, 以及如何用 “make” 命令重组一个程序。

第四章: Turbo C 程序设计初步, 介绍建立和运行 Turbo C 程序所涉及的一些基本步骤, 以及程序设计的基本元素及使用。

第五章: Turbo C 进一步的程序设计技术, 介绍了另外一些 C 程序设计元素, 包括数组, 指针, 指令和语句。

第六章: Turbo Pascal 与 Turbo C 的异同、转换和连接, 使用了一些例子, 对 Turbo Pascal 和 Turbo C 进行了比较, 描述和概括了这两种语言的异同、转换和连接。给出了避免程序设计易犯错误的某些提示, 可使已经掌握其中一种语言的读者, 很快熟悉另一种语言。

第七章: Turbo C 与 Turbo Prolog 的接口技术, 介绍 Turbo C 的模块如何与 Turbo Prolog 程序接口, 并提供了一些例子来说明这一技术。

第八章: Turbo C 高级程序设计技术, 描述了有关启动不同存储模式代码的内存组织, 指针计算, 汇编语言接口和浮点用法, 以及窗口和图形管理设施及其用法。

第九章: Turbo C 交互式编辑程序, 详细介绍了编辑程序命令和用法。

第十章: Turbo C 命令行, 介绍了 Turbo C 命令行编译程序版本提供的每一个命令行选择项及其用法。

第十一章: Turbo C 的用户定做, 说明定做程序 (TCINST) 的使用, 以及如何定做编辑程序命令, 修改缺省值和改变屏幕的颜色等。

第十二章: Turbo C 语言参考, 描述了与 Kernighan 和 Ritchie 提出的 C 语言定义不一致的特征和所有方面的列表, 详解了 Turbo C 对目前 ANSI C 标准中没有给出的一些扩充的细节。

附录 A: Turbo C 语法的 BNF 描述, 用改进的 BNF 形式描述了 Turbo C 的语法。

附录 B: Turbo C 字符屏幕管理和图形处理库函数, 给出了这两类库函数的详细描述和索引, 包括: 语言定义, 包括文件, 相关联的函数, 操作描述, 返回值等其他信息。

附录 C: Turbo C 实用程序及其使用, 讨论包含在 Turbo C 中六种实用程序。CPP: 预处理程序, 总结了 C 语言的预处理程序的工作方式; MAKE: 工程构造程序, 介绍了怎样使用 MAKE 来重建程序文件; TLINK: Turbo 连接程序, 介绍了如何使用 Turbo C 的命令行版本的内部 Turbo 连接程序。此外, 还介绍了 TLIB: Turbo 库管理程序; GREP: 文件搜索实用程序; BGI OBJ: 图形驱动程序和字体转换实用程序的功能和使用。

附录 D: 编译出错信息, 列出并解释每个出错信息, 并且指出了可能引起出错的各种原因。

本书适合广泛的读者对象, 例如: C 语言的初学者, 富有经验的 C 程序员, 使用 Turbo Pascal 和 Turbo Prolog 的程序员都可以从中各取所需地利用本书。我们建议:

C 语言的初学者, 可先阅读第四章和第五章, 也许能提供一个入门的向导, 并可熟悉建立

和编译 C 语言程序的过程。如果对集成开发环境还不太清楚,可以再回过头来阅读第二章,当准备运行程序时,第三章指出了具体的步骤。

富有经验的 C 程序员,可先读第八章“Turbo C 高级程序设计技术”。如果企图使用 Turbo C 移植和建立 C 程序,那就要读第三章和第十二章,弄清 Turbo C 与 Kernighan 和 Ritchie 以及 ANSI C 标准的差别。

已经熟悉 Turbo Pascal 的程序员,为了提高学习 Turbo C 语言的速度,最好先阅读第六章,那里提供了一些和 Turbo Pascal 程序等价的 Turbo C 程序的例子,并且详细地描述了这两种语言之间的异同和转换。此外,还需要阅读第二,四,五章。若使用过 Borland 的其他菜单驱动软件,则第二章只需快速浏览一下便可。

使用 Turbo Prolog 的程序员,如急于知道 Turbo C 模块怎样与 Turbo Prolog 接口,那么请先阅读第七章。

可配合本书使用的软件有最新版本的 Turbo C (含纯西文和中西文版本)以及 Turbo C 工具库盘片共十多枚,可与本书的作者联系获得。

本书由潘金贵(前言、第一、六、七、九、十一章)、谢俊元(第二章)、沈默君(第三、四、十章)、王锡江(第五章)、袁峰(第八章)、陈兆乾(第十二章)同志分工合作完成。附录由潘金贵、王锡江、陈兆乾、沈默君等同志共同完成。潘金贵对全书进行了仔细的修改。

本书的编写工作是在陈世福副教授的直接指导下进行的,并由他主审;袁峰博士协助校阅了有关章节和附录,谨致谢忱。

编写者还十分感谢谢琪、陆庆文等同志以及研究生陈彬、陈树权、姚威力、王军为本书编写所做的大量辅助工作。

限于水平和时间仓促,书中若有错误或不妥之处,恳请读者批评指正。

编者 1988年3月

于南京大学计算机科学系

目 录

前 言

第一章 Turbo C 的安装和启动

- 1.1 Turbo C 系统文件配置.....(1)
- 1.2 在不同配置的系统中建立 Turbo C(1)
 - 1.2.1 在只有单软盘系统上使用 Turbo C(1)
 - 1.2.2 在只有双软盘系统上使用 Turbo C(2)
 - 1.2.3 在带硬盘的系统上使用 Turbo C(3)
- 1.3 关于中西文 Turbo C(3)

第二章 Turbo C 集成开发环境

- 2.1 Turbo C 菜单系统及其使用(8)
 - 2.1.1 基本导航操作.....(9)
 - 2.1.2 Turbo C 的“热键”.....(10)
 - 2.1.3 菜单中的命令、开关及命名约定.....(10)
 - 2.1.4 主菜单.....(10)
 - 2.1.5 快速参考行.....(11)
 - 2.1.6 编辑窗口.....(11)
 - 2.1.7 信息窗口.....(13)
- 2.2 菜单命令.....(14)
 - 2.2.1 文件菜单.....(14)
 - 2.2.2 编辑命令.....(15)
 - 2.2.3 运行命令.....(15)
 - 2.2.4 编译菜单.....(15)
 - 2.2.5 工程菜单.....(16)
 - 2.2.6 选择项菜单.....(16)
 - 2.2.7 调试菜单.....(21)

第三章 Turbo C 程序的编译和运行

- 3.1 在集成开发环境中编译和连接 Turbo C 程序.....(23)
- 3.2 建立单个源文件的可执行程序.....(23)
- 3.3 调试.....(25)
 - 3.3.1 信息窗口.....(25)
 - 3.3.2 纠正语法错误.....(26)
- 3.4 使用多个源文件.....(26)
 - 3.4.1 建立多源文件的可执行程序.....(27)
 - 3.4.2 出错跟踪.....(27)
 - 3.4.3 Project-Make 的功用.....(29)

3.5 Make 的其他一些特性.....	(30)
3.5.1 外部目标文件和库文件.....	(30)
3.5.2 标准文件的取代.....	(31)
3.6 MAKE 实用程序.....	(31)

第四章 Turbo C 程序设计初步

4.1 建立第一个 Turbo C 程序.....	(32)
4.1.1 编译.....	(32)
4.1.2 运行.....	(33)
4.1.3 浏览产生的文件.....	(33)
4.2 修改第一个 Turbo C 程序.....	(34)
4.3 建立第二个 Turbo C 程序.....	(34)
4.3.1 程序记盘.....	(35)
4.3.2 运行 SUM.C.....	(35)
4.4 程序设计的基本元素.....	(35)
4.4.1 输出.....	(36)
4.4.2 数据类型.....	(37)
4.4.3 基本运算.....	(40)
4.4.4 输入.....	(42)
4.4.5 条件语句.....	(44)
4.4.6 循环.....	(46)
4.4.7 函数.....	(50)
4.4.8 注解.....	(53)

第五章 Turbo C 进一步的程序设计技术

5.1 数据结构.....	(54)
5.1.1 指针.....	(54)
5.1.2 数组.....	(58)
5.1.3 结构.....	(62)
5.2 switch 语句.....	(63)
5.3 控制流命令.....	(65)
5.3.1 return 语句.....	(66)
5.3.2 break 语句.....	(66)
5.3.3 continue 语句.....	(67)
5.3.4 goto 语句.....	(67)
5.3.5 条件表达式 (? ,).....	(68)
5.4 C 程序设计风格.....	(68)
5.4.1 使用函数原型和全函数定义.....	(68)
5.4.2 使用 enum 定义.....	(69)
5.4.3 使用 typedef.....	(69)
5.4.4 说明 void 函数.....	(70)
5.4.5 扩充的使用.....	(70)
5.5 C 程序设计中的常见问题.....	(71)
5.5.1 使用 C 字符串的路径名.....	(71)

5.5.2	指针的使用和误用	(71)
5.5.3	赋值号 (=) 和等号 (==) 的混淆	(73)
5.5.4	switch 语句中忘记 break 语句	(73)
5.5.5	数组下标	(73)
5.5.6	忘记传送地址	(74)
第六章 Turbo Pascal 与 Turbo C 的异同、转换和连接		
6.1	Turbo Pascal 与 Turbo C 的比较	(76)
6.1.1	程序结构	(76)
6.1.2	程序设计成份	(78)
6.1.3	数据结构	(90)
6.1.4	编程问题	(96)
6.1.5	Pascal 程序人员使用 C 时的常见错误	(101)
6.2	Turbo Pascal 程序到 Turbo C 的转换	(103)
6.2.1	把 Turbo Pascal 循环转换为 C 循环	(103)
6.2.2	case 和 if 语句	(104)
6.2.3	结构和记录	(105)
6.2.4	一个手工转换的例子	(105)
6.2.5	实现自动转换的一个试验原型	(107)
6.3	Turbo C 与 Turbo Pascal 的连接	(116)
第七章 Turbo C 与 Turbo Prolog 的接口技术		
7.1	Turbo C 与 Turbo Prolog 连接的步骤	(120)
7.1.1	对程序模块进行编译	(120)
7.1.2	对程序模块进行连接	(120)
7.1.3	其他注意事项	(121)
7.2	Turbo C 与 Turbo Prolog 的连接示例	(121)
7.2.1	示例之一: 两个整数相加	(122)
7.2.2	示例之二: 使用数学库	(123)
7.2.3	示例之三: 使用流模式和存储分配	(126)
7.2.4	示例之四: 画三维条形图	(129)
第八章 Turbo C 高级程序设计技术		
8.1	存储模式	(135)
8.1.1	8086 寄存器	(135)
8.1.2	内存分段及地址计算	(136)
8.1.3	近指针、远指针和特大指针	(137)
8.1.4	Turbo C 的六种存储模式	(139)
8.1.5	混合模式程序设计: 地址修饰符	(140)
8.2	多语言混合程序设计: 和其他语言接口	(145)
8.2.1	C 语言和 Pascal 语言的参数传递顺序	(146)
8.2.2	汇编语言接口	(148)
8.2.3	从汇编语言调用 Turbo C	(150)
8.2.4	定义汇编语言子程序	(151)
8.2.5	寄存器使用约定	(154)
8.2.6	从汇编子程序调用 C 函数	(154)

8.3 程序设计的低级支撑	(155)
8.3.1 伪变量	(155)
8.3.2 直接插入汇编代码	(157)
8.3.3 中断函数	(162)
8.3.4 使用低级支撑的例子 (BIOS 和低级接口模块)	(163)
8.4 浮点库的使用	(164)
8.4.1 仿真8087/80287芯片	(165)
8.4.2 8087/80287数学协处理器	(165)
8.4.3 不使用浮点数	(166)
8.4.4 87环境变量	(167)
8.4.5 寄存器和8087	(167)
8.4.6 浮点出错处理	(167)
8.5 警告和提示	(168)
8.5.1 Turbo C RAM的使用	(168)
8.5.2 要慎用Pascal调用约定	(168)
8.5.3 在DOS3.2和有浮点协处理器下使用Turbo C	(168)
8.6 Turbo C的字符屏幕管理	(169)
8.6.1 基本概念	(169)
8.6.2 显示方式控制	(171)
8.6.3 字符输出	(171)
8.6.4 程序例	(172)
8.7 Turbo C的图形功能	(173)
8.7.1 基本概念	(173)
8.7.2 图形系统控制	(175)
8.7.3 色彩控制	(175)
8.7.4 绘图和着色	(177)
8.7.5 图形屏幕管理和视区设置	(182)
8.7.6 图形模式下的正文输出	(183)
8.7.7 图形模式下的错误处理	(184)
8.7.8 状态询问	(184)

第九章 Turbo C 交互式编辑程序

9.1 快速进入和退出编辑程序	(187)
9.2 编辑窗口状态行	(187)
9.3 编辑命令	(188)
9.3.1 基本光标移动命令	(189)
9.3.2 快速光标移动命令	(180)
9.3.3 插入和删除命令	(190)
9.3.4 块命令	(191)
9.3.5 其他编辑命令	(192)
9.4 Turbo C 编辑程序与Word Star之比较	(197)

第十章 Turbo C 命令行

10.1 编译选择项	(198)
------------	-------

10.1.1 存储模式选择项	(198)
10.1.2 定义	(200)
10.1.3 处理器选择项	(200)
10.1.4 源选择项	(201)
10.1.5 代码选择项	(201)
10.1.6 出错选择项	(203)
10.1.7 命名选择项	(204)
10.1.8 编译控制选择项	(204)
10.2 连接择选项	(204)
10.3 环境选择项	(204)
10.3.1 隐式库文件和显式库文件	(205)
10.3.2 库文件的搜索算法	(205)
10.4 从命令行直接编译和连接Turbo C程序	(206)
10.4.1 命令行的一般格式	(206)
10.4.2 可执行文件的产生	(206)
10.4.3 有关命令行的一些例子	(206)
10.5 TURBOC. CFG文件	(208)
10.6 在DOS下直接运行Turbo C程序	(208)
第十一章 Turbo C的用户定做	
11.1 定做程序TCINST的功用	(209)
11.2 运行TCINST	(209)
11.2.1 Turbo C 目录选择项	(210)
11.2.2 编辑命令选择项	(211)
11.2.3 设置环境选择项	(213)
11.2.4 显示模式选择项	(214)
11.2.5 彩色定制选择项	(215)
11.2.6 改变窗口大小选择项	(216)
11.3 从TCINST 程序退出	(216)
第十二章 Turbo C 语言参考	
12.1 注解	(217)
12.2 标识符	(217)
12.3 关键字	(218)
12.4 常量	(218)
12.5 字符串	(220)
12.6 硬件特性	(220)
12.7 类型转换	(221)
12.8 运算符	(222)
12.9 类型与类型修饰符	(222)
12.10 结构和联合	(225)
12.11 语句	(226)
12.12 外部函数定义	(226)
12.13 作用域规则	(230)

12.14 编译程序控制行	(231)
12.15 过时成份	(234)
附录A Turbo C语法的BNF描述	(235)
附录B Turbo C 字符屏幕管理和图形处理库函数	
B.1 库函数索引.....	(241)
B.2 按字母顺序组织的库函数描述.....	(245)
附录C Turbo C 实用程序及其使用	
C.1 Turbo C 预处理程序 CPP.....	(279)
C.2 独立运行的MAKE 程序.....	(280)
C.3 Turbo 连接程序TLINK.....	(294)
C.4 Turbo库管理程序TLIB.....	(301)
C.5 文件搜索程序 GREP	(304)
C.6 图形驱动程序和字体转换程序 BGI OBJ	(307)
附录D 编译出错信息	(319)
主要参考文献	(321)

第一章 Turbo C的安装和启动

本章首先描述组成 Turbo C 系统的文件在软盘上的配置,然后介绍在不同配置的个人计算机上如何安装和启动 Turbo C 系统。

1.1 Turbo C系统文件配置

组成 Turbo C(1.5)版系统的全部文件分别存放在五张 5 吋软磁盘上,对存于磁盘上的这些文件的组织作了精心安排,这使得在使用的计算机系统上建立 Turbo C 所需要的磁盘交换最少。

各张软磁盘上的文件配置列在表1.1中。

1.2 在不同配置的系统上建立Turbo C

Turbo C 系统实际上包括两个不同的编译程序版本(集成开发环境版本和单独的命令行版本)所需要的所有文件和程序,支持六个存储模式的启动代码和库,以及8087协处理器的仿真程序。在所使用的计算机系统上建立 Turbo C 时必须把有关文件从配给的软磁盘复制到工作软盘或硬盘上。究竟复制哪些文件取决于为有关应用所选择的编译程序版本和存储模式。

前言中曾提到, Turbo C 可以在只有单个软盘驱动器或双软盘驱动器以及一个硬盘驱动器带一了软盘驱动器配置的 IBM PC 机及兼容机上运行。

本节分别介绍在上述不同配置的计算机系统上建立 Turbo C 的方法。

1.2.1 在只有单软盘系统上使用 Turbo C

在只有一个软盘驱动器或者有两个软盘驱动器的计算机系统上都可以运行 Turbo C。当然,带双软盘驱动器时用起来要方便些。本节先介绍在只有单个软盘驱动器的计算机上运行 Turbo C 的具体操作。

对仅有一个软盘驱动器的计算机系统,应该使用 Turbo C 的集成开发环境版本(启动配给软盘 1 号盘上的 TC.EXE 文件)。即使只有一个软盘驱动器,也需要产生两个独立的软盘:一个程序盘和一个工作盘。

程序盘应包括下列文件:

TC.EXE

TCHELP.TCH

工作盘应包括源程序、目标程序和可执行文件。此外,最好在该盘上建立如下两个独立的子目录(如何建立子目录请参见DOS手册的有关说明):

INCLUDE 用作用户的嵌入文件

LIB 用作用户的库文件。

如果建立这两个子目录,那么使用TCC时,必须使用-I选择项指定包括目录,用-L选择项指定库目录(更详细的说明可参阅第十章)。如果使用TC,那么必须在Options/Directories菜单中设置包括目录和库目录。

从配给软盘的3号盘上把嵌入文件拷贝到INCLUDE子目录中(*.H的文件和SYS\STAT.H文件),而把下列文件拷贝到LIB子目录中:

C0x.OBJ
EMU.LIB
FP87.LIB
MATHx.LIB
Cx.LIB

需要说明的是:在这些文件中出现的x代表要使用的存储模式的第一个字母。例如,若需要使用大(Large)存储模式,那么就用L代替相应的x,亦即,上述的C0x.OBJ, MATHx.LIB和Cx.LIB就变成C0L.OBJ, MATHL.LIB和CL.LIB

在只有单个软盘驱动器的计算机系统上运行Turbo C,应按下述步骤执行:

- (1)把程序盘插到软盘驱动器。
- (2)在DOS提示符下打入TC并按Enter键,这就启动了Turbo C的集成开发环境版本。
- (3)一旦程序已装入到内存,那么取下程序盘,并插入工作盘。
- (4)如果需要联机帮助的话,应在按下F1功能键之前再插入程序盘。

1.2.2 在只有双软盘系统上使用Turbo C

如果使用的PC机带有两个软盘驱动器,使用Turbo C要较仅有单个软盘的系统方便。此外,不仅可以使Turbo C的集成开发环境版本,还可以使用常规的命令行版本,不过需要两套不同的软盘来启动不同的编译程序版本。具有方法如下:

(1)运行集成开发环境版本

启动Turbo C的集成开发环境版本(TC.EXE)时,建立程序盘和工作盘的步骤同1.2.1所述。但是,这时两个软盘是不可互相替换的。程序盘必须在A驱动器、工作盘必须在B驱动器中。

(2)运行命令行版本

运行Turbo C命令行版本(TCC.EXE),需要为驱动器A和驱动器B分别建立一个。新的磁盘。并将下列文件复制到在驱动器A的程序盘上:

TCC.EXE
TLINK.EXE
LIB子目录
INCLUDE子目录

而把下列文件复制到LIB子目录中。

C0x OBJ
EMULIB
FP87LIB

MATHx.LIB

Cx.LIB

同样, 这些文件名中x的含义如前所述, 它将替换成要使用的存储模式的第一个字母。

把所有的嵌入文件复制到 INCLUDE 子目录中(H文件可在配给软盘的3号盘上找到)。

把所有的.C和.OBJ以及由 Turbo C 命令行版本建立的.EXE文件复制到在驱动器B的工作盘上。

1.2.3 在带硬盘的系统上使用 Turbo C

在带硬盘的系统上使用 Turbo C, 可以很容易地从集成开发环境版本转换到命令行版本。

将 Turbo C(1.5 版)系统装配到硬盘时, 可运行装配程序 INSTALL.BAT (该程序在配给软盘的1号盘上)。该程序首先在硬盘上建立特定的目录, 然后将配给软盘上的文件按类拷贝到各个目录中。

运行该程序时, 首先将含有该文件的软盘置于A驱动器中, 以C盘为缺省盘符, 在DOS命令行中键入:

```
C>A:install A:C:\TURBOC
```

并按Enter键, 则INSTALL程序自动在C盘上建立如下目录:

C:\TURBOC	Turbo C主目录(编译程序、实用程序等)
C:\TURBOC\INCLUDE	Turbo C系统定义的包括\嵌入文件
C:\TURBOC\INCLUDE\SYS	Turbo C UNIX相容的包括\嵌入文件
C:\TURBOC\LIB	Turbo C所有存储模式的库文件

建立目录后, 装配程序将把配给软盘上的文件按类逐一拷贝到对应的目录中。由于配给软盘不止一张, 故在此过程中, 装配程序提示用户更换软盘。

该程序执行完毕, Turbo C 系统即已装配到硬盘上, 用户可方便地使用。

注意, 键入命令时, 若含有 INSTALL.BAT 的软盘未置于A驱动器中, 则以所在驱动器名替代A。此外, Turbo C 主目录也不一定用 C:\TURBOC, 用户可根据需要, 建立所希望的目录名, 只需用希望的目录名替代C:\TURBOC即可。

若硬盘上已经建立了上述目录, 则执行命令

```
A:install A:C:\TURBOC
```

时, 装配程序直接将各个文件拷贝到相应目录中。

1.3 关于中西文 Turbo C

笔者等在使用 Turbo C 的基础上, 分别对其1.0版和1.5版进行了汉化。

为了适应不同机型的特点, 针对具有不同显示特性的机种, 设计和实现了两个中西文 Turbo C 版本, 即 IBM PC 版(适合IBM PC/XT、AT、286 及兼容机)和长城版(适合长城 0520 CH 及兼容机)。

两个中西文版本的共同特征是:

(1) 保持西文版的全部特点, 并兼容。

(2) 能编辑和调试带有中文输入输出的中西文 Turbo C 程序, 包括:

① 汉字可以作为字符串进行操作。

② 全屏幕编辑系统可以对汉字进行输入、显示及编辑。所有原来的编辑功能, 都能用来对汉字操作。

③ 部分提示信息和帮助信息的汉化。

(3) 中西文 Turbo C 能对带汉字的 Turbo C 源程序进行正确编译, 并生成正确的执行代码。

(4) 长城版所有窗口彩色特性与西文版保持一致。

(5) 中西文 Turbo C 系统以及其编译后的程序能在 CCDOS 下运行。

汉化工作主要是对 Turbo C 的集成开发环境版本进行的, 从而使得 Turbo C 倍受国内广大程序人员的青睐。

表 1.1 Turbo C 系统文件在软盘上的配置

盘 号	文 件 名	内 容
1 集 成 开 发 环 境	README .COM	README.COM 文件
	TC .EXE	Turbo C 编译程序的集成环境版本
	INSTALL .BAT	硬盘装配批处理文件
	INSTALLH .BAT	由 INSTALL.BAT 调用的硬盘装配批处理文件
	TCINST .EXE	TC.EXE 的定做程序
	GRAPHICS .DOC	关于 BGI 图形的补充文件
	HELLO .C	一个 Turbo C 程序例子
	CPASDEMO .PAS	演示 Turbo Pascal4.0 和 Turbo C 接口的 Pascal 程序
	CPASDEMO .C	演示 Turbo Pascal4.0 和 Turbo C 接口的 C 模块例子
	CTOPAS .TC	用于 TC.EXE 的配置文件, 以便按正确的格式建立 与 Turbo Pascal4.0 程序连接的 Turbo C 模块
	BAR .C	Turbo Prolog 程序 PBAR.PRO 使用的 C 例子函数
	PBAR .PRO	演示 Turbo Prolog 与 Turbo C 接口的 Turbo Prolog 程序例子
	READM	README.COM 应用的正文
2 命 令 行 及 实 用 程 序	TCC .EXE	Turbo C 编译程序的命令行版本
	TLINK .EXE	Turbo 连接程序
	BGIOBJ .EXE	字体和图形驱动程序转换程序
	CCP .EXE	Turbo C 预处理程序
	MAKE .EXE	工程管理程序
	TCCONFIG .EXE	转换配置文件的程序
	TLIB .EXE	Turbo C 库管理程序
	RULES .ASI	同 Turbo C 接口的汇编程序嵌入文件
	Co .ASM	用作启动代码的汇编源程序
	SETARGV .ASM	用作命令行语法分析的汇编源程序
	SETENV .ASM	用于准备环境的汇编源程序

续表1.1

盘 号	文 件 名	内 容
2	BUILD-CO.BAT MAIN .C GREP .COM TOUCH .COM	用于建立启动代码模块的批处理文件 Turbo GREP 程序 修改文件日期和时间的程序
3 嵌 入 文 件 和 库 I	???????? .H C0T .OBJ C0S .OBJ CS .LIB MATH .LIB C0L .OBJ CL .LIB MATHL .LIB EMU .LIB GRAPHICS.LIB FP87 .LIB SYS 目录 STAT .H	嵌入文件(详见表1.2) 极小模式启动代码 小模式启动代码 小模式运行时刻库 小模式数学库 大模式启动代码 大模式运行时刻库 大模式数学库 8087仿真程序库 图形库 8087库 Turbo C嵌入文件(含文件状态/目录函数)
4 库 II	C0C .OBJ CC .LIB MATHC .LIB C0M .OBJ CM .LIB ATMHM .LIB C0H .OBJ CH .LIB MATHH .LIB	紧凑模式启动代码 紧凑模式运行时刻库 紧凑模式数学库 中模式启动代码 中模式运行时刻库 中模式数学库 特大模式启动代码 特大模式运行时刻库 特大模式数学库
5 例 和 子 字 、图 体 形 体 驱 文 动 文 程 件 序	TCHelp .TCH MCALC .C MCINPUT .C MCOMAND .C MCPARSER.C MCUTIL .C MCDISPLY.C MCALC .H MCALC .PRJ BGIDEMO .C BGIDEMO .PRJ MATHERR.C GETOPT .C	Turbo C帮助文件 MicroCalc主程序源代码 MicroCalc输入子程序源代码 MicroCalc命令源代码 MicroCalc输入语法分析程序源代码 MicroCalc实用程序源代码 MicroCalc屏幕显示源代码 用于MicroCalc嵌入文件 MicroCalc工程文件 图形演示程序 图形演示工程文件 处理数学库例外的源代码 命令行中语法分析程序选择项

续表1.1

盘 号	文 件 名	内 容
5	CPINIT .OBJ	用于连接Turbo C和Turbo Prolog的初始化代码
	ATT .BGI	关于ATT400图形卡的图形驱动程序
	CGA .BGI	CGA的图形驱动程序
	EGAVGA .BGI	EGA和VGA的图形驱动程序
	HERC .BGI	Hercules的图形驱动程序
	IBM8514 .BGI	IBM8514图形卡的图形驱动程序
	PC3270 .BGI	PC3270的图形驱动程序
	GOTH .CHR	关于黑体字字符集的字体
	LITT .CHR	关于小写字母字符集的字体
	SANS .CHR	关于无衬线字符集的字体
	TRIP .CHR	三倍字符集的字体
	MCALC .DOC	补充的MicroCalc文件

表 1.2 Turbo C 嵌入文件一览

文 件 名	作 用
ALLOC .H	说明存储管理函数(allocation和dcallocation等)
ASSERT .H	定义assert调试宏
BIOS .H	说明调用 IBM-PC ROM BIOS 例行程序所用的各种函数
CONIO .H	说明调用 DOS 控制台 I/O 例行程序所用的各种函数
CTYPE .H	包含字符分类和字符转换宏(如isalpha和toascii)使用的信息
DIR .H	包含为使用目录和路径名进行工作的结构、宏和函数
DOS .H	定义各种常量,并给出为满足MS-DOS和特殊的8086及调用需要的说明
ERRNO .H	为出错代码定义常量助记码
FCNTL .H	定义同库子程序 open 连接所需要的符号常量
FLOAT .H	包含浮点例行程序的参数
IO .H	包含了低级I/O例行程序的结构和说明
LIMITS .H	包含环境参数及编译时刻的限制信息和整型量的上下界
MATH .H	为数学函数说明原型,还定义了宏HUGE-VAL说明由matherr和-matherr函数使用的例外结构
MEM .H	说明存储操作函数(这些函数中有许多在string.h中定义)
PROCESS .H	包含为了spawn...和exec...函数的结构和说明
SETJMP .H	定义由longjmp和setjmp函数使用的一个类型jmp-buf,并且说明函数longjmp和setjmp
SHARE .H	定义用于使用共享文件的函数的参数
SIGNAL .H	定义常量SIG-IGN和SIG-DFL,并且说明ssignal和gsignal函数

续表 1.2

文 件 名	作 用
STDARG .H	定义读被说明为可接受可变参数的参数表所使用的宏
STDDEF .H	定义几种公用的数据类型和宏
STDIO .H	定义标准的I/O程序包所需的类型和宏,这个程序包在K&R中有定义,并且在UNIX系统V中得到扩充。定义标准的I/O的预定义流stdin, stdout和stderr,并且说明流级(stream-level)I/O子程序。
STDLIB .H	说明几个公用的例行程序,转换子程序,查找/排序子程序和其他的杂函数
STRING .H	说明一些串操作和存储操作的库函数
SYS\STAT .H	定义用于打开和建立文件的符号常量
TIME .H	定义被时间转换函数asctime、函数localtime和gmtime填好的一个结构,定义由函数ctime、difftime、gmtime、localtime和stime使用的一个类型;还提供了这些函数的原型。
VALUES .H	定义重要的常量,包括依赖于机器的值;提供了与UNIX系统V的兼容性。

第二章 Turbo C集成开发环境

Turbo C 不仅是一个快速、高效的编译程序，同时还带有一个易学、易用的集成开发环境。使用 Turbo C，无需独立的编辑、编译和连接程序，就能建立并运行 C 程序。所有这些功能都组合在 Turbo C 的集成环境内，并且可以通过一个简单清晰的主屏幕使用这些功能。

本章由以下两节组成。

第一节介绍菜单系统及其使用，主要内容有：

- 描述 Turbo C 主屏幕的组成。
- 说明如何选择 Turbo C 主菜单。
- 叙述如何进入编辑窗口并使用编辑功能。

第二节描述菜单命令，主要内容有：

- 阐述每个菜单项的功能。
- 概述编译时刻的各种选择项。

如果读者对菜单驱动软件不甚了解，建议先阅读第一节。如果对菜单驱动软件如 Sidekick 或 Turbo Prolog 较为熟悉，可以跳过第一节，直接阅读第二节。

2.1 Turbo C 菜单系统及其使用

在 DOS 提示符下键入 TC 并按 Enter 键，即可装入 Turbo C。此时初启屏幕包括主菜单和版本信息（程序运行的任何时刻按 Alt-F10 均可调出版本信息）。按任意键，版本信息消失，留下如图 2.1 所示的主屏幕。主屏幕由四部份组成：主菜单，编辑窗，信息窗和快速参考行。

File	Edit	Run	Compile	Project	Options	Debug
Edit						
Line 1 Col 1 Insert Indent Tab C: EXAMPLE1.C						
Message						
F1--Help F5--Zoom F6--Message F9--Make F10--Main menu						

图2.1 Turbo C的主屏幕

2.1.1 基本导航操作

为了使读者熟悉 Turbo C 系统, 首先介绍一些基本导航操作:

(1) 在菜单内

- 用高亮度的大写字母选择一个菜单项, 或使用箭头键移动到相当的选择项按 Enter 键。
- 按 Esc 退出一层菜单。
- 按 Esc 从主菜单进入前次激活窗。(当某窗口被激活时, 其顶部为双线, 窗口名以高亮度显示)。
- 按 F6 可从任意菜单层进入前激活窗。
- 使用左、右箭头键可从一个下拉菜单项移到另一个。

(2) 在 Turbo C 的任何地方

按 F1 取得当前位置的帮助信息。

像其他 Borland 产品一样, Turbo C 提供了上下文敏感的联屏帮助。在 Turbo C 的任何一个菜单处只要按 F1 功能键就可以得到帮助。

帮助窗详细描述当前位置的各个功能项。每个帮助屏幕可能包含一个关键词(高亮度项), 由此可以得到更详细的信息。只要使用箭头键将光标移到关键词位置, 按 Enter 键就可以得到该选择项的更详细的帮助信息。当然也可以更用 Home 和 End 键来选择屏幕的第一个或最后一个关键词。

按 Alt-F1 可以回到前次帮助屏幕(总共有20个帮助屏幕)。在进入帮助系统后, 按 F1 可以得到帮助索引, 按 Esc (或其他“热键”, 热键的含义稍后介绍), 可退出帮助系统返回原来的菜单选择。

- 按 F10 调用主菜单。
- 按 Alt 和主菜单命令的首字母 (F, E, B, C, P, O, D) 可直接调出该命令。例如, 在系统的任何地方, 按 Alt-E 可立即进入编辑窗; Alt-F 可进入文件菜单。
- 按 Alt-F5 可以交换存储的屏幕。使用 Turbo C 的 TC 版本时, 用户可看到两个屏幕中的一个——集成开发环境屏幕或输出屏幕。集成开发环境用于编辑、编译连接或调试程序, 输出屏幕用于运行 Turbo C 可执行程序或通过 File/OS shell 菜单命令暂时退回到 DOS。一些例外的情况下, Turbo C 可在一个“存储输出屏幕”缓冲区中保存输出屏幕的内容, 每当用户选择 Run 或 File/OS shell 时, 缓存被更新。为显示该保存的屏幕, 可按 Alt-F5 (“存储屏幕”热键)。

在某些条件下, Turbo C 可保存屏幕的内容。这样当用户选择 File/OS shell 或运行一个程序时, 屏幕记录了现场。无论何时用户从集成开发环境中运行一个程序或选择 File/OS shell 时, 系统将显示模式恢复为从 DOS 中启动 TC 时的模式 (“启动模式”)。一般有两种情况使得 Turbo C 删除保存输出屏幕缓冲区内容:

- 第一, 当编译或连接时, 编译程序或连接程序都使用了屏幕缓冲区。
- 第二, 当启动 TC 时, 屏幕的显示模式与存储屏幕模式不兼容。

(3) 在编辑或信息窗内

- 按 F5 可以放大/缩小窗口。
- 按 F6 可选择窗口。

文件菜单下的Quit (按Q或将选择亮条移到Quit按Enter键), 可退出Turbo C 返回DOS。此时, 当前修改过的工作文件若没有存盘, 则系统询问是否需要存盘。

2.1.2 Turbo C 的“热键”

在介绍各种菜单选择项之前, 先介绍一些“热键”。所谓热键就是能够立即完成某一功能的键。如前所述, 按Alt和主菜单命令的首字母将直接进入所选的菜单或执行动作。

表2.1列出了Turbo C的所有热键, 请记住: 无论在Turbo C环境的什么地方, 一旦按了这些键就立即执行相应的功能。

表2.1 Turbo C 的热键

键	功 能	键	功 能
F1	显示当前位置的帮助信息	Alt-F5	交换存储的屏幕
F2	保存编辑器的当前文件	Alt-F9	将编辑程序内的文件编译成.OBJ
F3	装入一个文件	Alt-F10	显示版本屏幕
F5	放大/缩小活动窗	Alt-C	进入编译菜单
F6	转换活动窗	Alt-D	进入排错菜单
F7	指向前一个错误	Alt-E	进入编辑窗
F8	指向下一个错误	Alt-F	进入文件菜单
F9	执行make	Alt-O	进入选择菜单
F10	调主菜单	Alt-P	进入工程菜单
Alt-F1	调出所参考的最后一个帮助屏幕	Alt-R	运行程序
Alt-F3	选取文件并装入	Alt-X	退出Turbo C 返回 DOS

2.1.3 菜单中的命令、开关及命名约定

在Turbo C菜单中有二种类型的项:

- 命令: 执行一个任务(运行, 编译, 保存等等)。
- 开关: 控制Turbo C的一些开关特性(如Link map, Test stack overflow等), 或通过反复按Enter键循环选择某项的值(如Instruction set或Calling convention)。

本书对所有的窗口项都用一个缩写表示。一给定的窗口项的缩写是这样形成的: 从主菜单下开始进入到该菜单名的首字母序列。例如:

(1) 在主菜单下, 称选择(Options)菜单中编译项(Compile)的出错处理(Errors)为O/C/E(按OCE)。

(2) 在主菜单下要选择包括目录的名: Options/Directories/Include directories, 只要按ODI。

2.1.4 主菜单

在主屏幕的顶部是Turbo C的主菜单窗, 如下所示:

File	Edit	Run	Compile	Project	Options	Debug
------	------	-----	---------	---------	---------	-------

现分别介绍主菜单的七个选择项。

- File** 包括对文件的装入、保存、选取、建立、写入等操作；以及显示、修改目录；退出程序；调用 DOS 等。
- Edit** 建立并编辑源文件。
- Run** 自动编译、连接并运行程序。
- Compile** 将源程序编译并装配为目标文件或执行文件。
- Project** 标识组成程序的文件，处理工程。
- Options** 选择编译开关(如：存储模式、编译选择、诊断和连接选择)；可以定义宏，也可以记录输出文件、库文件和嵌入文件等目录，保存或装入编译选择配置。
- Debug** 跟踪排错。

其中，Edit 和 Run 是菜单命令；Edit 调用编辑程序；Run 运行程序。而其他一些菜单项都是带有很多选择项的下拉菜单或上托菜单。

2.1.5 快速参考行

无论在窗口或菜单，屏幕的底部都有一个相应的快速参考行。该行是当前位置可用功能键的帮助一览。若要观察在不同设置下其他复合键的功能，只要按住 Alt 键数秒，当再按其他键时，快速参考行将显示相应信息。

主菜单默认的快速参考行形式如下：

F1-Help F5-Zoom F6-Edit F9-Make F10-Main menu

若按住 Alt 键，就显示 Alt 的组合键功能表：

Alt-F1—Last Help Alt-F3—Pick Alt-F5—Saved screen Alt-F9—Compile Alt-X—Exit

2.1.6 编辑窗口

本节将介绍 Turbo C 编辑窗口的组成及其使用。

进入编辑窗，只要将主菜单的选择亮条移到 Edit 并按 Enter (或直接按 E)。在系统的任何地方(包括信息窗内)按 Alt-E 也能直接进入编辑窗。(注意 Alt-E 是 F10-E 的“热键”)。一旦进入编辑窗，窗口顶部为双线并高亮度显示活动窗口名。

除了可以编辑源程序的编辑窗口体外，还有两个信息行：编辑状态行和快速参考行。

在编辑窗顶部的编辑状态行给出了正在编辑文件的有关信息(如文件光标位置、编辑模式)。编辑状态行的形式如下：

Line Col Insert Indent Tab C:FILENAME.EXT

其中：

- Line n** 光标所处文件的行号
- Col n** 光标所处文件的列号
- Insert** 插入模式，使用 Insert 或 Ctrl-V 可以改变插入模式。有关插入和重写模式参见第九章。
- Indent** 自动缩进，该开关可以用 Ctrl-O I 来控制。关于自动缩进的介绍参阅第九章。
- Tab** 制表符，该开关可以用 Ctrl-O T 来控制。
- C:FILENAME.EXT** 表示所编辑文件的驱动器(C:)、文件名(FILENAME)和

扩展名(EXT)。

屏幕底部的快速参考行显示有关热键的功能,此时为:

F1-Help F5-Zoom F6-Message F9-Make F10-Main menu

这些键的功能是:

F1-Help	打开帮助窗,它提供关于Turbo C编辑命令的信息。
F5-Zoom	使活动窗口在全屏幕和分割式屏幕之间转换。
F6-Message	F6可以在编辑窗和信息窗之间转换。
F9-Make	产生EXE文件。
F10-Main menu	调用主菜单。

编辑程序的命令和Sidekick或Turbo Pascal的相似,将在第九章中详细介绍。

编辑程序处于插入模式时,按Enter可以换行。每行最大宽度为248个字符。编辑窗有77列宽。若输入时超过77列,窗口自动滚列。编辑窗口的状态行给出了文件行、列的光标位置。

当源程序输入完毕时,可以按F10调出主菜单。原文件将继续留在屏幕上。按F键时,可从主菜单进入编辑窗。

2.1.6.1 编辑命令简介

以下是一些最常用的编辑命令:

- 正文的光标滚动用Up/Down, Left/Right和PgUp/PgDn键
- 删除一行用Ctrl-Y
- 删除一个字用Ctrl-T
- 设置块标志用Ctrl-KB(开始)和Ctrl-KK(结束)。
- 移动一块用Ctrl-K V
- 复制一块用Ctrl-K C
- 删去一块用Ctrl-K Y

有关编辑命令的详细说明参见第九章。

2.1.6.2 如何在编辑窗编辑源文件

Turbo C编辑窗在装入某一文件前,自动地以NONAME.C命名。在此可以使用编辑程序的所有功能:

- 建立一个新的源文件,以NONAME.C或其他名命名。
- 装入并编辑存在的文件。
- 从编辑文件表中挑选一个文件,然后将之装入编辑窗。
- 保存编辑窗中的文件。
- 将编辑窗中的文件写入一新文件。
- 在编辑窗和信息窗之间转换,查找相应的编译错误。

建立或编辑程序而不进行编译,并不需要信息窗。因此,可按F5将编辑窗扩展成全屏幕。

(1) 建立一个新源文件

建立一个新源文件可用以下方法:

- 在主菜单,选择File/New,按Enter键,此时打开编辑窗且文件名为NONAME.C。
- 在主菜单,选择File/Load后,打入源文件名(在编辑窗内按F3也可完成同样的功能)。

(2) 装入一个存在的源文件

装入并编辑已存在的文件有两种选择：**File/Load**或**File/Pick**。

① 在主菜单下选择**File/Load**

- 打入要编辑文件的路径和文件名，例如：

`C:\TURBOC\TESTFILE.C`

• 在文件名提示处打入“*.*”，显示当前目录的所有文件，目录后跟一斜线(\)，可选择该目录下的所有文件。例如，打入 `C:*.C` 将仅显示根目录下的带有 `C` 扩展名的所有文件。

按 **Up/Down** 和 **Left/Right** 箭头键将光标移到要选择的文件名处，然后按 **Enter** 即可装入该文件。

• 在主菜单，可按 **E** 调用编辑程序，此时将打开带有被编辑文件的编辑窗（如果没有任何文件装入的话，则使用缺省文件名 `NONAME.C`）。

② 如果要选择**File/Pick**，可以

- 按 **Alt-F** 和 **P**。
- 用 **Up** 和 **Down** 箭头键来移动选择亮条到相应文件名处。

Pick 可以让用户快速地选取以前装入过的文件。

(3) 保存源文件

- 在系统的任何地方可按 **F2**。
- 在主菜单可以选择 **File/Save**。

(4) 写一个输出文件

可以将编辑窗中的文件写到新文件中或覆盖一个已存在的文件，可以将它写到当前目录或指定目录中。

在主菜单下，选择 **File/Write to**，然后在新文件名提示处键入路径名和文件名并按 **Enter** 键。

`C:\DIR\SUBDIR\FILENAME.EXT`

其中：

`C:` 是驱动器符（可省）
`\DIR\SUBDIR\` 表示所选的目录
`FILENAME.EXT` 输出文件名及其扩展名。

按一次 **Esc** 可返回到主菜单，再次按 **Esc** 或 **F6** 或 **Alt-E** 即回到活动窗。

若 `FILENAME.EXT` 已存在，那么编辑程序将核准是否将原来的冲掉。

2.1 7 信息窗口

在编译和调试源程序的时候可以使用信息窗观察诊断信息。Turbo C 在信息窗列出被编译文件的每一个警告或出错信息，同时在编辑窗以高亮度条指出源程序中相应的位置。

当光标在信息窗时，快速参考行如下所示：

F1-Help **F5-Zoom** **F6-Edit** **F9-Make** **F10-Main menu**

相应的功能说明如下：

F1-Help 打开介绍如何使用 Turbo C 的出错处理的帮助窗口
F5-Zoom 将信息窗扩展为全屏幕

F6-Edit	激活编辑窗
F9-Make	生成一个EXE文件
F10-Main menu	调用主菜单

2.2 菜单命令

主菜单主要包括装入、编辑、编译、连接、调试和运行等功能。七个选择项包括：**F**ile（文件操作）、**E**dit（编辑）、**R**un（运行）、**C**ompile（编译）、**P**roject（工程）、**O**ptions（选择）和**D**ebug（调试）。

下面将逐项讨论其功能和使用。有些在高级程序设计时才用到的选择项将在第三章中详细介绍。

2.2.1 文件菜单

文件下拉菜单涉及装入文件、建立新文件、保存文件等多种选择（见图2.2）。当装入文件时，自动进入编辑窗。当编辑结束时，可以将它保存到任何目录下的任何文件名中。甚至可以在下拉菜单中改变到另一目录，暂时进入DOS系统，或退出Turbo C。

装入（Load）

装入一个文件，可以装入一个指定的文件，也可以用类似DOS中的屏蔽得到一个文件选择表。当试图装入另一个文件时，对于修改过而没有保存的文件，系统将提示保存，此时热键失去作用。

选取（Pick）

从以前装入的至多8个文件的表中找一个并装入编辑窗，将光标定在上次最后位置。如果选择“—load file—”项，效果和选择**F**ile/**L**oad或F3一样，Alt-F3是取得该表的热键（详见第十章“Turbo C 命令行”）。

Load	F3
Pick	Alt-F3
New	
Save	F2
Write to	
Directory	
Change dir	
Os shell	
Quit	Alt-X

图2.2 文件菜单选择项

新文件（New）

进入编辑窗，编辑新的文件。文件名默认为NONAME.C（可以在保存文件时改名）。

保存（Save）

将编辑窗中的文件保存到磁盘上。如果此时文件名是NONAME.C，则编辑程序将询问是否要改名，在系统的任何地方，按F2可以完成同样的操作。

写（Write to）

将一个文件写到新的名下或覆盖一个已存在的文件。

目录（Directory）

显示目录及所需的文件（要进入当前目录只要按Enter键）。

改变目录（Change dir）

显示当前目录，并允许将其修改为指定的盘驱动器和目录。

进入DOS系统（Os shell）

暂时离开Turbo C，进入DOS系统。按Exit可返回Turbo C。当需要运行DOS命令

而又不想退出Turbo C时可使用这一功能。

退出 (Quit)

退出Turbo C并返回到DOS提示符。

2.2.2 编辑命令

编辑命令调用内部全屏幕编辑程序。按F10可以在编辑窗内调出主菜单。此时源文件的显示还残留在屏幕上。在主菜单下按Esc或E或在任何地方按Alt-E键可重返编辑窗。

2.2.3 运行命令

先进行工程制作(**Project-Make**, 有关它的功能, 请参见第三章), 然后使用**Options/Args**中给定的参数运行当前程序。程序运行结束后, 将显示“Press any key”的信息。此时若按Alt-V就可得到主函数返回的值。

应该指出的是, 在DOS提示符下, 直接打入可执行文件名, 也可运行相应的程序。

2.2.4 编译菜单

在编译菜单下有编译到OBJ文件、生成EXE文件、连接EXE、建立所有的工程和设置初始C文件等功能(见图2.3)。

编译到OBJ (Compile to OBJ)

本菜单项是一个命令, 它总是显示要产生的文件名, 例如: C: EXAMPLE.OBJ。当选择时, 就编译该文件, 所显示的OBJ文件名来自于下述两个文件名:

- 初始C文件名, 或
- 装进编辑窗口的最后一个文件名。

生成EXE文件 (Make EXE file)

本菜单项是一个调用工程制作的命令。它总是显示产生的EXE文件名, 例如C: EXAMPLE.EXE。一旦选择就生成此文件。该EXE文件名取自下述三个文件名中的一个:

- 在Project/Project name菜单项中指定的工程文件,
- 初始C文件名, 或
- 装进编辑窗口最后一个文件的名。

需要说明的是, 本章所涉及到的“Make”是指Project-Make而不是指独立的MAKE实用程序。Project-Make是功能类似MAKE的程序内部工具(参见第三章)。

连接EXE (Link EXE)

连接当前的OBJ文件和LIB文件生成新的EXE文件。

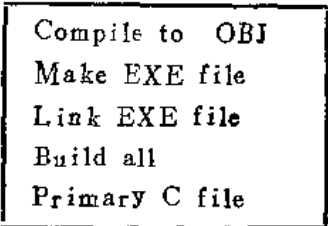
构造所有工程 (Build all)

重构工程的所有文件。

初始C文件 (Primary C file)

当选择**Compile/Compile to OBJ**命令(Alt-F9)时, 可使用本选择项标识将被编译的文件。

初始C文件选择项用在编译包含多个嵌入文件(.H)和初始C文件时使用。如编译时发现错误, 就自动调入那个包含错误的文件以供修改(可能是.C,也可能是.H文件)。当按



```
Compile to OBJ
Make EXE file
Link EXE file
Build all
Primary C file
```

图2.3 编译菜单选择项

Alt-F9时,就重新编译那个初始C文件(无论它是否在编辑窗内)。

编译时,将出现一个反映编译结果的窗口,编译完成后,按任意键即消去该窗口。若有错误出现,则光标自动停在信息窗中的第一个错误处。编译命令及其选择项在第三章中详细说明。

2.2.5 工程菜单

本下拉菜单的选择项(见图2.4,可以将多个源文件和目标文件组合成最后的程序。有关工程的详细内容参见第三章。

工程名 (Project Name)

定义一个包含工程文件名。工程文件名将作为 .EXE和 .MAP文件的名。

打断编译 (Break make on)

本下拉菜单允许指定在编译时,出现以下情形是否要打断编译,这些情况是警告(Warning),出错(Errors),严重出错(Fatalerrors)和连接开始(Link)。

清除工程 (Clear Project)

清除工程名并清除信息窗口。

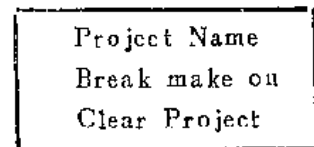


图2.4 工程菜单选择项

2.2.6 选择项菜单

选择项菜单可设置集成开发环境的工作方式。主要包括:编译程序和连接程序的选择项,库和包括目录,程序运行时刻的参数等(参见图2.5)。

2.2.6.1 编译程序

本下拉菜单允许进行以下选择:存储模式(Model),宏定义(Defines),代码生成(Code generation),优化(Optimization),源代码处理(Source),出错处理(Errors)和命名(Name)等。

存储模式 (Model)

这些选择项用来设置 Turbo C 的各种存储模式;存储模式决定了存储编址方式。默认的存储模式是小模式(Small)。如果选择了Compiler/Model则出现以下选择项:极小模式(Tiny),小模式(Small),紧凑模式(Compact),中模式(Medium),大模式(Large)和特大模式(Huge)。有关存储模式的详细内容参见第八章。

宏定义 (Defines)

选择Defines后,就出现一个宏定义窗,它能把宏定义传给预处理器,宏定义之间用分号(;)隔开,值用等号(=)相连。

宏定义中前后的空格均略去,但中间的空格是保留的,若宏定义中包含分号,则必须在分号前放一个斜线(\)。

下面是一些宏定义的例子:

BETA-TEST; ONE = 1; COMPILER = TURBOC

代码生成 (Code Generation)

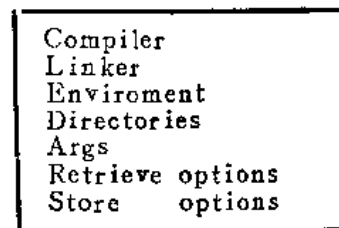


图2.5 选择项菜单

这些选择项用来告诉编译程序生成目标代码的方式。

这些方式包括：调用约定 (Calling convention)，指令集 (Instruction set)，浮点运算方式 (Floating point)，缺省字符类型 (Default char type)，编址方式 (Alignment)，下线符生成 (Generate underbar)，重复串合并 (Merge duplicate strings)，标准栈框架 (Standard stack frame)，栈溢出测试 (Test stack overflow) 和行号 (Line numbers) 等，下面分别介绍其功能和用法。

调用约定 (Calling convention)

该选择项使编译程序对一个函数调用产生C调用序列或Pascal调用序列。两种调用序列的区别在于栈的处理、参数的数目和次序、外部标识符的大小写和前缀情况。如果读者对这些尚不熟悉或还未阅读第八章的高级程序设计技术，请不要改变原来的选择。

指令集 (Instruction set)

允许指定不同的CPU；可以在8088/8086指令和80186/80286之间选择。缺省的指令集是8088/8086。Turbo C可以生成扩充的80186指令，当生成80286程序在IBM PC/AT的MS-DOS 3.0以上版本运行时，也可以使用这个选择。

浮点运算方式 (Floating point)

允许有以下三种选择：

- 8087/80287 直接产生8087在线代码。
- 模拟 判断是否有8087芯片，若有就直接使用，否则就模拟8087，但速度稍慢。
- 无 此时不允许使用浮点运算（若使用浮点运算，将在连接时出错）。

缺省字符类型 (Default char type)

选择带符号和无符号字符说明。若选择带符号，则编译程序视所有字符为signed char类型；选择无符号则相反，缺省值为有符号。

合并重复串 (Merge duplicate strings)

合并两个重复串的优化，使程序更紧凑。

编址方式 (Alignment)

它允许选择字编址和字节编址。若是字编址，非字符数据编址均为偶地址。若是字节编址，奇、偶均可。字编址在8086和80286处理器存取数据时可提高速度。

标准栈框架 (Standard stack frame)

生成一个标准栈框架（标准函数入口和出口代码）调试程序时很有用，它可以简化调用子程序时进、出栈的处理。

栈溢出测试 (Test stack overflow)

生成运行时刻检查栈溢出的代码。这样虽然花费一些时间和空间，却更安全可靠，因为栈溢出是很难检查出来的。

生成下线符 (Generate underbar)

本选择开关默认打开。未仔细阅读第八章关于高级程序设计技术，对其尚不熟悉之前请不要改变它。

行号 (Line numbers)

目标文件中包括行号（用作符号调试程序），它增大了目标文件，但并不影响执行程序的大小和运行速度。由于编译程序在进行跳转优化时可能将源程序的多行组成共用的代码，或重新按排行号，所以建议在使用本选择时关闭“跳转优化” (Jump optimization)。

优化 (Optimization)

本下拉子菜单主要提供下列优化方式:优化策略(Optimization for),使用寄存器变量(Use register variables),寄存器优化(Register)和跳转优化(Jump optimization)等,现分别叙述之。

优化策略 (Optimization for)

改变Turbo C的代码生成策略。编译程序一般使用“空间优化”,即选择尽可能小的代码序列。当选择“速度优化”时,编译程序将对给定任务生成尽可能快的代码。

使用寄存器变量 (Use register variables)

允许使用寄存器变量,若打开此开关,寄存器变量将自动赋值。否则,即使用了register关键字,编译程序也不使用寄存器变量(详见第十章)。

一般地。可以打开该选择开关,除非和该程序接口的汇编代码不支持寄存器变量。

寄存器优化 (Register optimization)

用记住寄存器的内容和尽可能多的重复使用来防止多余的存取操作。

编译程序不知道寄存器是否被某个指针间接地修改过,所以使用该选择项必须特别注意。有关这方面的限制和说明参见第十章。

跳转优化 (Jump optimization)

用消除多余的跳转、重新组织循环和开关语句来减少代码的大小。重组循环可以提高速度。

源代码处理 (Source)

本下拉子菜单可确定编译程序如何处理源代码。主要包括下列内容:标识符长度(Identifier length),嵌套注解(Nested comments)和仅用ANSI关键字(ANSI keyword only)等。下面分别介绍之。

标识符长度 (Identifier length)

指定标识符的有效字符数目。所有标识符用前面N个字符来区别。这包括变量、宏名和结构成员名。数目可以是1—32的任意值。缺省值是32。

嵌套注释 (Nested Comments)

允许在Turbo C的源文件中嵌套注释。嵌套注释在其他C的实现中通常是不允许的。

仅用ANSI关键字 (ANSI keywords only)

当打开此选择开关时,编译程序只辨认ANSI关键字,而Turbo C扩充的关键字被看作一般的标识符。这些关键字包括: near, far, huge, asm, cdecl, pascal, interrupt, -es, -ds, -cs, -ss, 和伪寄存器变量(-AX, -BX, ...)。在编译时这一选择还定义符号—STDC—。

出错处理 (Errors)

本下拉菜单控制Turbo C编译程序发现某些诊断信息后的反应和处理。下面将分别介绍之。

出错后停止 (Errors; stop after)

编译程序检查出25个错误后停止。25是默认值,可以输入0—255中的任何一个数。输入0,使编译程序不停地进行。

警告后停止 (Warnings; stop after)

编译程序检查出100个警告后停止。100是默认值。其合法范围是0—255。输入0表

示编译不停止，直到超过出错界限。

显示警告 (Display warnings)

该选择开关的默认状态是打开的，这意味着下面的警告信息一旦查出则显示其内容：可容忍警告，违犯ANSI，公共错误和非公共错误。若关闭该选择项，则所有警告均不显示。警告信息的详细讨论参见第十章和附录。

命名 (Names)

为代码、数据和BSS段改变缺省的段名、组名和类名，在选择这些项时，星号(*)表示使用缺省名。除非是熟练者，一般情况下不要改变这个选择项。

2.2.6.2 连接程序 Linker

本下拉菜单处理关于连接程序的选择项，有关选择开关的意义下面分别介绍，更详细的内容参阅第十章。

Map 文件 (Map file)

选择要产生的map文件的类型。对于不是off的选择项，map文件被放置在由O/E/O定义的输出目录上。缺省值是关闭(off)，其他的选择是：段(Segment)、公共(Publics)、细节(Detailed)。

初始化段 (Initialize segments)

告诉连接程序要对一些段进行初始化(通常没有必要)。

缺省库 (Default libraries)

当连接那些不是Turbo C编译程序生成的模块时，其他编译程序可能已经在目标文件放置了一个缺省的库表。当这一选择开关打开时，连接程序将在这些库和Turbo C提供的库中找没有定义的例行子程序。否则，连接程序仅在Turbo C提供的库中查找，而忽略在OBJ文件中默认的库。

警告重复符号 (Warn duplicate symbols)

控制目标文件和库文件中有重复符号时，是否要显示警告信息。缺省值为不显示(即为off)。

栈警告 (Stack warning)

取消连接程序的'No stack specified'的信息。

大小写敏感连接 (Case-sensitive link)

连接时，控制大小写敏感开关。通常本选择开关是打开的(区分)，因为C是大小写敏感的语言。

2.2.6.3 环境

这一子菜单下有六个选择项(1.0版本只有前三项)，允许改变Turbo C的工作环境，以适合程序设计的需要。现分别介绍各个菜单选择项。

编辑自动保存 (Edit auto save)

当使用Run或Os shell时，自动保存编辑文件(如果它曾被修改过)，以防止丢失。

后备源程序 (Backup source file)

保存源文件的时候，Turbo C自动建立一个备份。备份使用相同的文件名，扩展名为BAK。本选择开关可以打开或关闭，缺省状态为打开。

窗口扩大 (Zoomed windows)

本选择项打开可将编辑或信息窗扩大为全屏幕。两个窗口仍然可以互相转换，但在任何

时刻只能看到一个窗口。关闭本选择开关可恢复包含编辑和信息窗口的分割式屏幕环境。

自动保存配置 (Config auto save)

通常, 只有当选择**Options/Store**时, Turbo C才保存当前配置(将它写入磁盘)。当该自动保存配置为打开状态时, 在选择**Run**、**File/Os shell**, 或退出集成开发环境时(如果配置文件从未被保存或自上次保存后已被全部修改), Turbo C将保存配置文件。

该项打开时, 如果配置文件仍未被存储, Turbo C将为自动存储的文件选择一个文件名。该名为用户存储或检索的最后一个配置文件的名; 如果用户从未装入、检索、存储过配置文件, 则对配置文件命名为TCCONFIG.TC(在当前目录中)。

制表宽度 (Tab size)

编辑程序Tab模式为打开状态时, 按Tab键, 编辑程序将在该文件中插入一个制表字符且光标跳到下一个制表符处。这个菜单项可用来决定制表符间距的大小; 可以为2—16的任一数字(缺省值为8)。

当要改变制表宽度时, 选择Tab size, 键入所希望的大小并按Enter键, 编辑程序用所选择的大小重新显示所有的制表符, 用户可在配置文件中存储新的制表宽度(从选择项菜单中选择Store选择项)。

屏幕大小的选择 (Screen size)

选择屏幕大小时, 将出现一个下拉子菜单, 该菜单允许用户说明集成开发环境屏幕是否以25行、43行或50行显示正文。可使用其中的一项或两项, 取决于PC机显示适配器的类型。

25行标准显示(25行×80列)。这是配有单色显示适配器(MDA)或彩色图形适配器(CGA)系统唯一可用的屏幕选择。

43行EGA显示(43行×80列)。如果所用PC机里有EGA卡, 则可选择本选择项(也可选择25行标准显示), 选择了本选择项则正文显示转换成43行80列。

50行VGA显示(50行×80列)。如果PC机配有VGA卡, 则可选择本选择项(也可选择25行标准显示), 一旦选择了本选择项, 正文显示将转换成50行80列。

2.2.6.4 目录

该菜单项告诉Turbo C在哪里寻找要编译、连接的文件和帮助文件, 以及设置输出文件所在的目录。

该子菜单下有下列选择项:

包括目录 (Include directories)

可以指定标准包括文件的目录。标准包括文件是在尖括号(< >)内给定的文件。例如,

```
#include <Myfile.h>
```

多个目录用分号隔开。

库目录 (Library directories)

该选择项允许用户列出多个库目录, 输入最多为127个字符(包括空白)。

进入该选择项时遵守下列原则:

- 多个目录路径名须用分号(;)隔开;
- 分号(;)前后可以有空白, 但并不一定需要;
- 可以有相对或绝对的路径名, 包括相对于运行记录位置的路径名, 而不是当前路径名。

下面是一个例子:

c:\turbo\lib;c:\turbo\mylibs; a:\newturbo\mathlibs; a:\vidlibs

输出目录 (Output directory)

.OBJ、.EXE和.MAF文件都存放在输出目录中, 执行**M**ake和**R**un命令时, Turbo C 就在该目录下寻找相应的文件。若不指定输出目录, 文件就存放在当前目录中。

Turbo C 目录 (Turbo C directory)

这个目录是Turbo C系统用来寻找配置文件(.TC)和帮助文件(TCHLP.TCH)的, 为使Turbo C在启动时能找到缺省的配置文件(TCCONFIG.TC), 必须在本目录下执行装配程序TCINST。

选取文件名 (Pick file name)

该选择项定义了一个要装入的选取文件的名。这里输入的名意味着装入该选取文件(如果存在), 并且定义了退出时Turbo C在何处保存该选取文件。如果要改变选择文件名, Turbo C在装入新文件前保存当前选取文件。

如果未给出选取文件名, 那么Turbo C在当前选取文件菜单项包含文件名时, 写一个选取文件。

当前选取文件 (Current pick file)

该菜单项显示了当前选取文件的名和所在位置(如果存在的话)。该项不能由用户选择使用, 仅用于提供信息。当装入缺省的选取文件或用户选取文件名菜单项输入一个选取文件时, 当前选取文件将显示一个文件名。如果用户改变选取文件名或退出集成开发环境, 则Turbo C在该选取文件中存储当前选取表信息。

2.2.6.5 参量

允许给定运行程序的命令行参量。确切地说就如在DOS命令行打入的一样。这里仅需给定参量, 程序名省略。

2.2.6.6 搜索选择项配置

装入曾用保存选择项命令(Store options)保存的配置文件。

2.2.6.7 保存选择项配置

在配置文件中保存所有有关编译程序、连接程序、调试、环境和工程的选择项(配置文件默认的名叫TCCONFIG.TC)。启动时, Turbo C在当前目录寻找TCCONFIG.TC, 若找不到, 就在Turbo目录中寻找。

2.2.7 调试菜单

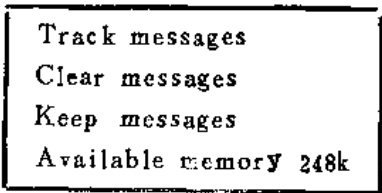
调试菜单的选择项允许对编译后程序进行调试。这里仅作简单介绍, 有关出错跟踪等详细内容参阅第八章。

跟踪信息 (Track messages)

在信息窗口内滚屏时, Turbo C在编辑窗口内跟踪错误。它有以下三个选择项:

- (Track...Current file) 为缺省的情况, 跟踪编辑窗口内的当前文件。

- (Track...All file) 表示装入并跟踪每个产生出错或警告信息的文件。



Track messages
Clear messages
Keep messages
Available memory 248k

图2.6 调试菜单选择项

- 关闭跟踪开关。

清信息 (Clear messages)

清除信息窗的出错信息。

保留信息 (Keep messages)

这是一个选择开关。打开时, Turbo C 将在信息窗保留当前出错信息, 以后编译出现的任何信息将在信息窗追加显示。当某文件编译结束时, 有关该文件的任何信息就从信息窗移去, 新的信息加在后面。当关闭本选择开关时, 在编译或产生 (Make) 操作前所有信息均自动清除。

可用内存量 (Available memory)

指出留给编译用的所有内存容量。

Turbo C 不仅提供了本章所介绍的集成开发环境编译程序版本, 还提供了常规的命令行版本。这将在第十章中详细介绍。

第三章 Turbo C 程序的编译和运行

Turbo C 为开发 C 语言程序提供了一个灵活的环境,启动时的参数是系统按默认选择项设置的,用户亦可方便地改变默认值以更好地满足程序开发的需要。Turbo C 还为程序开发提供了各种支撑工具,如出错跟踪和文件系统管理等。

Turbo C 的集成开发环境是一个完整的软件包,极其有效且易于使用,读者一定首先想了解它,故本章首先讨论在集成开发环境中如何编译及连接 Turbo C 源文件以产生可执行文件,并且描述对调试、完善程序很有帮助的 Turbo C 交互式出错跟踪的特性。然后介绍在集成开发环境中怎样运行程序,同时介绍 Turbo C 的内部工程设施(Project-Make)及其使用。

对 Borland 公司的集成开发环境尚不熟悉的读者,在学习本章之前 首先应了解 第二章介绍的 Turbo C 菜单系统。

3.1 在集成开发环境中编译和连接 Turbo C 程序

在集成开发环境下,建立一个新的可执行文件的一般步骤为:

- (1) 设置工作环境以指示编译程序和连接程序在何处寻找和保存有关的文件。
- (2) 把相应的源程序装入编辑窗中(注意:如果可执行程序由多个模块组成,则需建立一个由所有模块名组成的工程文件)。
- (3) 建立可执行程序文件。

建立可执行文件的步骤取决于源文件是单个文件还是多个文件。

3.2 建立单个源文件的可执行程序

现以 HELLO.C 为例,先介绍由单个源文件建立可执行文件的方法,然后介绍如何对由多个模块组成的程序产生可执行文件。从启动 Turbo C 到运行程序,有以下六个必要的步骤:

- (1) 装入 Turbo C

在 DOS 提示符下键入 tc 便可装入 Turbo C。

用户可指出 Turbo C 程序所在的目录:

- TC.EXE 可能在当前目录下。
- TC.EXE 可能在 DOS 路径上另一个目录下。
- TC.EXE 可能在 DOS 3.X 版本的另一个目录下,可直接在 DOS 命令行打入路径名直到 TC.EXE,例如:

\TURBOC\TC

在 tc 后接受两种命令行参数:一是需编辑的文件名,二是一个 "/c" 开关紧跟着一个配置文件名。这两种参数的次序无关紧要。例如:

`tc hello/cmyconfig`

就将HELLO.C置于编辑窗口并且装入配置文件MYCONFIG.TC(注意:在“/c”开关和文件名之间没有空格,被编辑文件扩展名的默认值假定为c,配置文件扩展名的默认值假定为TC)。

(2) 选择实例程序

选择含有实例程序的驱动器号及目录,按Alt-F进入文件菜单(或用第二章中描述的另一种方法),选择Change dir,打入含有实例程序的目录名,使该目录成为当前目录。

(注意,新目录提示框出现时,列出了当前目录,用于检查现在正处于哪个目录下,简单地按Esc键则返回菜单而不改变当前目录)。

(3) 设置工作环境

要设置和保存工作环境可按Alt-O直接进入Options菜单,选择Directories,则出现一个子菜单,在此子菜单中,选择两项:Include directories和Library directories。

选择Include directories,键入含有Turbo C标准嵌入文件(.H文件)的盘符和目录名并按Enter键。

再选择Library directories,键入含有Turbo C库文件的和启动文件的盘符和目录名。

此外,如果愿意,也可在此子菜单中用同样的方法设置Output directory,一旦确定了一个输出子目录,那么编译程序和连接程序所有的输出文件都写入此目录下,而不再写入当前目录下。

对大多数简单情况而言,为建立C语言程序,所有这些设置是必不可少的。

设置的参数可保存在配置文件TCCONFIG.TC中,当启动时,Turbo C在当前目录中寻找TCCONFIG.TC文件,如果找到则装入。这样,运行一个特定的程序时,在同一个目录中有一个隐含的配置文件与之匹配。如果在当前目录中未找到配置文件,则Turbo C在自己所在目录中寻找。所以,可在Turbo C所在目录中保存一个通用的配置文件,而在不同的源文件目录中保存各种不同的配置文件。(注意:必须用TCINST安装程序设置Turbo C目录)。

按Esc键返回Options菜单,选择Store保存当前设置,默认的配置文件TCCONFIG.TC即被写入当前目录。如果要给配置文件重新命名,只要在配置文件名提示框中键入新文件名并按Enter即可。但请注意:对配置文件重新命名后,必须用tc命令加上“/c”开关或用Options菜单中Retrieve options命令显式装入此配置文件。

(4) 装入实例程序进行编辑

按F3键,装入实例程序。有两种装入程序的方法:直接输入文件名或利用*.c得到一文件名清单,然后将光标移至所需文件名处,按Enter键。文件装入后,编辑(Edit)窗口即成为活动窗口,可编辑源文件(在此例中,不需编辑源文件)。文件形式为:

```
#include <stdio.h>
main( )
{
    printf("Hello,world\n");
}
```

(5) 建立可执行文件

Compile 菜单给出了即将被编译的文件名和即将建立的可执行文件名。如果没有定义工程文件名, Turbo C 默认用户已提供了建立可执行文件所需的信息。Turbo C 首先查看 **Primary C** 文件, 再查看最后装入到编辑程序中的文件来决定可执行文件的名字。然后, 就像已定义了一个只含有这个文件名的工程文件一样进行工作(有关工程文件将在后面讨论)。

当可执行程序只对应一个源文件时, 不建立工程文件就能够编译、连接和运行程序。为建立一个可执行文件, 必须编译源文件产生目标文件, 再将目标文件与标准启动文件、库文件进行连接。有多种方法完成上述工作, 最简单的方法是按 **F9(Make)** 键, 或选择 **Compile** 菜单, 然后, 在 **Make EXE file** 处按 **Enter** 键。

(6) 运行程序

至此, 已建立了一个 **EXE** 可执行文件, 按 **Alt-R** 或主菜单中选择 **Run** 即能运行。(注意: 也可在 **DOS** 提示符下直接打入该可执行程序的名。)

3.3 调 试

一般的程序设计环境, 未能提供有效的跟踪和改错手段。Turbo C 集成开发环境提供了良好的调试手段, 它的编译程序能确定源程序的语法错误且能给出警告信息。Turbo C 把编译程序和连接程序产生的信息集中到一个缓冲区, 并在信息窗口显示出来。这样, 一方面能立即看到有关信息, 同时能直接观察源文件。

现在不妨一试, 在实例程序中增加一些语法错误: 把第一行 **include** 前的 **#** 去掉, 再去掉第四行 **printf** 函数中的后一个双引号, 则文件为:

```
include <stdio.h>
main
{
    printf("Hello world\n");
}
```

按 **Alt-F9** 编译该文件, 编译窗口显示出错信息和警告信息(在这里, 有三个出错信息, 没有警告信息)。

3.3.1 信息窗口

需观察信息时, 在编译窗口按任意键则转入信息窗口。这时, 信息窗口中有一亮条位于第一个错误(或警告)信息上。因为这个错误(或警告)是由编辑窗中的源文件产生, 故在编辑窗口也出现一亮条, 指出错误(或警告)在源文件中的位置。

用光标键(**↑**, **↓**)上下移动信息窗口中的亮条, 以观察其他的信息, 注意编辑窗口中亮条的踪迹, 可以发现: 随着信息窗口亮条的移动, 编辑窗口的亮条总是置于源文件产生相应错误(或警告)处。当信息窗口的亮条置于“**compiling**”处时, 编辑窗口的光标恢复到原来的位置。

如果信息窗口中的信息较长, 超过了窗口长度, 可用 **←** 键和 **→** 键水平地滚动信息。按 **F5** 键(**Zoom**)还可放大信息窗口以便能观察更多的信息。但是, 信息窗口放大后, 将看不到编辑窗口, 就不能进行跟踪, 此时, 窗口不再是分割式屏幕模式。

3.3.2 纠正语法错误

为纠正语法错误, 可将信息窗口的亮条移至第一个错误信息处, 并按 Enter 键, 光标即移至编辑窗口中相应错误的位置上。注意编辑窗口的状态行, 它显示出用户选择的信息(这在放大/缩小(即Zoom)模式下工作非常有效)。现在便可纠正错误(在上面的例子中, 给第一行加上#)。

若源文件不止一个语法错误, 有两种方法可继续纠正下面的错误:

一种方法是按F6键返回信息窗口, 选择下一个要纠正的错误。

另一种方法不需返回信息窗口, 只要简单地按F8键(下一个错误), 编辑程序就把光标置于源文件中对应信息窗口下一个错误的位置上。请注意, 选择“下一个错误”后, 状态行的信息和信息窗口亮条的位置都发生了变化, 如按F7(前一个错误), 则亮条又移回到原先的位置。

这两种方法各有优点。有时, 源文件中一个含糊的错误会引起编译程序的连锁反应, 产生许多出错信息, 在这种情况下, 只需纠正第一个出错信息对应的地方即可。此时, 使用第一种方法较为有效——纠正错误后返回信息窗口, 选择下一个有意义的出错信息。其他情况下, 一般要顺序地检查出错信息, 按F8(下一个错误)较为有效。

F8和F7(下一个错误和前一个错误)都是“热键”, 亦即在任何情况下它们都起作用。这样, 如果信息窗口是活动窗口, 按F8键, 就不能选择当前亮条所指出的出错信息, 而选择的是下一个出错信息(如果想选择当前出错信息, 则应按Enter键)。后面没有出错信息时, F8键无效。注意: 连接时产生的错误是无法进行跟踪的。

在纠正语法错误时, 经常需要增加和删除文字, 编辑程序对此能够保持跟踪: 当处理下一个错误时, 它能正确地将光标置于出错处, 用户不需要记住增加和删除的行数及其位置。

3.4 使用多个源文件

Turbo C 强有力的功能之一是它能够对多源文件分别编译。它的Project-Make设施使之功能更强。

前面的例子中, 可执行程序是仅由一个源文件产生的; 故不需要定义工程文件, 仅用Compile/Make就能产生可执行文件。当需要产生由多个源文件构成的可执行程序时, 必须明确地告诉Turbo C, 这个可执行程序包含了哪些源文件, 即要建立一个工程文件。

一个工程可汇集许多功能, 而建立一个工程文件就像列出C语言源文件名清单一样简单, 现在, 从仅有两个源文件开始对工程文件进行讨论。

最基本的情况是: 两个源文件一个是主程序文件, 另一个是支撑文件。支撑文件定义了主文件所需用的函数或数据。例如, 主文件MYMAIN.C如下:

```
#include <stdio.h>
char *Getsring(void);
main(int argc, char *argv[])
{
    char *s;
```

```

    if(argc>1)
        s=argv [1] ;
    else
        s='the univers';
    printf("%s %s.\n", getstring( ), s);
}

```

支撑文件MYFUNCS.C定义如下:

```

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

```

建立一个名为MYPROG.PRJ的工程文件,其形式为:

```

mymain
myfuncs

```

Turbo C 假定工程文件中源文件的扩展名是.C(亦可给出扩展名.C),源文件排列的次序并不重要,除非是要决定文件被编译的次序。下面的工程文件与上述工程文件产生同样的结果:

```

myfuncs
mymain

```

注意:工程文件名(MYPROG.PRJ)不同于主文件名(MYMAIN.C)。这两个文件名在扩展名前的部分可以相同(但有不同的扩展名),但一般不这样做。这是因为可执行文件名及连接过程中产生的map文件名是以工程文件名为基础的。在这个例子中,可执行文件名为MYPROG.EXE,可能产生的map文件名为MYPROG.MAP。

对工程文件中的文件可以冠以路经名。于是,可执行文件所对应的源文件可分布在不同的目录下。

(为了以后学习的方便,可将上述三个文件MYMAIN.C, MYFUNCS.C和MYPROG.PRJ输入并保存。)

3.4.1 建立多源文件的可执行程序

为了建立工程文件的可执行程序,首先需在工程(Project)菜单中输入工程文件名。按Alt-P显示Project菜单,选择Project name,此时可明确输入工程文件名或使用统配符(*.PRJ)列出目录从中挑选(未保存过的工程文件,不能列入目录中)。工程文件名输入后,按F9(Make)即产生可执行程序,按Alt-R(Run)即运行程序。

运行一个程序(按Alt-R)能直接对工程文件中的源文件进行编译和连接,产生可执行程序并运行,即可省略用F9(Make)产生可执行程序的操作。

3.4.2 出错跟踪

和单源文件一样,信息窗口也处理编译多源文件时产生的错误。

为了说明怎样纠正错误,在 MYMAIN.C 和 MYFUNCS.C 中故意制造一些错误,在 MYMAIN.C 中,将第一行的第一个尖角括号去掉,再去掉第四行 char 中的 c,这两个改变将使 MYMAIN.C 产生三个错误和三个警告信息。

装入 MYFUNCS.C,去掉第四行“return”的第一个 r,这一改变将导致两个错误和两个警告信息。

源文件重新编辑后,原来的目标文件作废,需重新进行编译。为了跟踪多源文件,需要修改 Project-Make 用于决定何时终止产生可执行文件的参考准则,这可通过在 Project 菜单中设置开关“Break make on”的值来进行。

3.4.2.1 产生(可执行文件)过程的终止

Turbo C 终止产生可执行文件有几种可能性。例如,当一个可执行文件产生完毕时,或要报告某些出错信息时,Project-Make 都将停止。

再如,Project-Make 找不到列入工程文件中的某个源文件(或某个从属文件,后面将要讨论)时也要终止,按 Ctrl-Break 也可强行终止 Project-Make。

编译过程产生出错信息时,也可以终止产生过程。可以在 Project 菜单用 Break make on 指定一类信息,这样,当出现这类信息时,产生过程即终止。Break make on 开关指定的信息类可以是“Warnings”,“Errors”,“Fatal Errors”或“Link”,分别表示编译时出现警告信息,错误信息,致命错误或执行连接程序前终止产生过程。Break make on 的默认值是“Errors”。

上述四种模式的使用,取决于纠正源文件的方法。如果想一看到有警告信息或出错信息就立即停下来修改源文件,则可将开关置成“Warnings”或“Errors”。如果想在修改源文件前得到全部的出错信息,则可将开关置成“Fatal Errors”或“Link”。

3.4.2.2 多源文件中的语法错误

为说明多源文件中的错误,将 Break make on 开关置成“Fatal errors”。

前面已在 MYMAIN.C 和 MYFUNCS.C 中引入了一些语法错误。请按 F9(Make)以产生可执行文件,这时编译窗口显示出被编译的源文件和各源文件中错误、警告信息的总数。按任意键(如空格键),则编译窗口消失,信息窗口显示出警告和出错等信息。

此时,有一亮条被置于信息窗口第一个出错或警告信息处,如果该错误信息对应的源文件正处于编辑窗口,则编辑窗口有一亮条置于产生相应错误的位置上。注意,每编译一个源文件,信息窗口都对应有“Compiling”信息,这个信息不是出错或警告信息,而是为了把各个源文件产生的各种信息分开来。

当下移(信息窗口)亮条跨越“文件边界”时,编辑窗口根据 Debug 菜单中 Track messages 开关的值决定是否跟踪下一个源文件。Track messages 开关的默认值是仅跟踪当前源文件。

当信息窗口中亮条处的信息所对应的源文件不在编辑程序中时,编辑窗口中的亮条消失。这时,如果按 Enter 键,则它所对应的源文件被装入编辑程序且光标被置于出错位置上,如果按 F6,则返回信息窗口,不再跟踪所对应的源文件。

如果将 Track messages 开关设置成 All files,就能够不受文件边界限制地跟踪源文件。这意味着,当亮条在信息窗口上下移动(滚动)时,Turbo C 自动装入相应的源文件。

把 Track messages 开关设置成 off,则不产生跟踪,在这种情况下,可简单地 在信

息窗口选择有关信息,选中后按Enter键,这时,选中信息对应的源文件被装入到编辑窗口,光标置于出错位置上。

F7和F8(上一个错误和下一个错误)不受Track messages开关设置的影响,这两个键总是指出上一个错误或下一个错误,并能装入相应的源文件。

3.4.2.3 信息的保留和丢弃

通常,产生一个工程的可执行文件时,信息窗口总是被清除,用来显示新的信息,然而,有时我们希望在新的产生过程中,保留原有的信息。

假设有一个包含许多源文件的工程,Break make on开关置成Errors,在这种情况下,可能会得到来自不同源文件的警告信息,但是,当某个源文件含有一个错误时,产生可执行文件的过程即终止,用户可纠正错误并可证实编译程序已接受了这个纠正,但在重新执行产生(Make)或编译时,将丢失原来的警告信息,这些警告信息有可能还需查看,怎样才能避免这种情况呢?只要将Debug菜单的Keep messages开关置成Yes即可。

如果Keep message开关置成Yes,执行产生(Make)时信息窗口不被清除,仅仅是那些纠正过的信息才被除去。这样,一个文件原有的信息就被重新编译时产生的新信息所替代。

选择Debug菜单中的Clear message,将清除所有当前信息,将Keep message开关置成Off,重新执行产生(make),也可清除所有旧的信息。

更换一个工程时,清除信息窗口是一个好习惯。Project菜单的Clear project既能清除工程文件名,又能清除当前信息。选择Clear project后,可定义一个新的工程文件,或装入单源文件进行编辑、运行,或定义Primary C文件名。

3.4.3 Project-Make的功用

在产生工程的可执行程序时,所涉及到的最基本的信息是工程中的一些C源文件名。Project make在此基础上提供了许多较为复杂的信息,要了解这一点,首先需要了解make的工作过程。

Make对源文件的更新时间和编译程序产生的目标文件的更新时间进行比较,由此对简单的工程文件定义了一些隐含的依赖关系,如在例子MYPROG.PRJ中就有如下的依赖关系:

```
MYMAIN.OBJ 依赖于 MYMAIN.C
MYFUNCS.OBJ 依赖于 MYFUNCS.C
MYPROG.EXE 依赖于 MYMAIN.OBJ, MYFUNCS.OBJ和
                MYPROG.PRJ
```

即目标文件MYMAIN.OBJ在MYMAIN.C更新之后就作废,须重新编译MYMAIN.C以产生新的MYMAIN.OBJ,可执行程序总是依赖于工程文件本身和工程中所有的目标文件,这就是说,如果工程中任何一个目标文件(MYMAIN.OBJ, MYFUNCS.OBJ)或工程文件本身(MYPROG.PRJ)被更新了,则须重新产生可执行程序(MYPROG.EXE)。这些隐含的依赖关系是根据列入工程文件的C文件名产生的。

产生较复杂的工程文件的可执行程序,有时需明确地描述这种依赖关系,这在C语言源文件依赖于其他文件时非常有用。如含有嵌入文件(即,H文件,它定义了与外部程序的接口)

的C语言源文件, 如果与外部程序的接口发生了变化, 那么, 使用外部程序的源文件需重新编译, 这就要明确描述依赖关系来完成。

例如, 有一个主程序文件MYMAIN.C, 它含有一个嵌入文件MYFUNCS.H, 如果在工程文件中说明了如下依赖关系:

```
MYMAIN.C(MYFUNCS.H)
MYFUNCS(MYFUNCS.H)
```

则一旦 MYFUNCS.H 发生变化, 产生可执行程序时将重新编译 MYMAIN.C 和 MYFUNCS.C

注意: 工程文件使得 MYFUNCS.C 依赖于 MYFUNCS.H, 有利于进行一致性检查, 于是, 隐含的及明确的依赖关系为:

```
MYMAIN.OBJ    依赖于 MYMAIN.C和MYFUNCS.H
MYFUNCS.OBJ    依赖于 MYFUNCS.C和MYFUNCS.H
MYPROG.EXE     依赖于 MYMAIN.OBJ, MYFUNCS.OBJ
                和 MYPROG.PRJ
```

列入工程文件中的C源文件可含有多个依赖关系, 只要简单地把它所依赖的文件依次列于圆括号中, 互相之间用空格或逗号(,)或分号(;)隔开即可。例如, MYMAIN.C 依赖于 MYFUNCS.H, YOURS.H 和 OTHER.H, 则可说明成:

```
MYMAIN.C(MYFUNCS.H, YOURS.H, OTHER.H)
```

这种方法无需复杂的语法就能按传统的产生方式产生可执行程序。

3.5 Make 的其他一些特性

Make 功能还具有两个特性, 一是能指明工程文件所需连接的外部目标文件和库文件, 二是能取代标准启动文件和标准库文件。

3.5.1 外部目标文件和库文件

工程文件有时要使用其他语言(如汇编语言或其他语言)编译程序产生的目标文件。有时, 工程文件也会使用标准库文件中未提供的、具有特殊功能的外部库文件。这种情况下, 需将外部目标文件名和外部库文件名分别附以明确的扩展名.OBJ和.LIB, 列入工程文件中(文件名排列的次序无关紧要)。

如:

```
MYMAIN(MYFUNCS.H)
MYFUNCS(MYFUNCS.H)
SPECIAL.OBJ
OTHER.LIB
```

Project-Make遇到具有扩展名为 OBJ 的文件时, 就将它列入需连接的文件之列, 这个过程不需对它编译或寻找它的源文件。同样, 对于扩展名为 .LIB 的文件, 则列入连接程序搜索的库文件之列, 这个过程同样不需对它编译, 也不需要建库。

注意, 这类文件(.OBJ文件和.LIB文件)不能带有明确的依赖关系(如果带有, 则略而不

管)。但C语言源文件却可依赖于这类文件,例如:

```
MYMAIN(MYFUNCS.H,SPECIAL.OBJ)
MYFUNCS(MYFUNCS.H,OTHER.LIB)
SPECIAL.OBJ
OTHER.LIB
```

这意味着,当.OBJ或.LIB文件更新时,依赖它的C语言源文件需重新编译。

3.5.2 标准文件的取代

某些时候,必须避免使用标准启动文件和标准库文件。一般在非常特殊的情况下才使用这个功能,初学者不易掌握,现作简要的说明。

要取代标准启动文件,必须把一个名为C0*.OBJ的文件作为第一个文件列入工程中,在这里星号(*)可以被任何DOS文件名替代(如C0MINE.OBJ),这样,在工程中,第一个文件名开头两字符必是C0,它的扩展名必是.OBJ。

取代标准库文件,只需在工程文件中列入一个特殊的库文件名(不一定是在第一个)。这个库文件名开头字符必须是C,且库文件名只能有两个字符,它的扩展名必须是.LIB,如: CX.LIB或C1.LIB。

不使用标准库文件时,产生过程不再根据O/C/Code generation菜单中Floating-point的值来连接数学库。如果在取代标准库文件时需使用这些库文件,则需把库文件名明确地列入工程文件中。

3.6 MAKE 实用程序

Turbo C有一独立的实用程序MAKE,它的功能比集成开发环境中Project-Make的功能更强,该实用程序基于UNIX MAKE,允许用户描述源文件和目标文件的依赖关系,它还能检验这些依赖关系,以确保文件能正确地编译和连接。

使用MAKE有什么优点呢?使用Project-Make时,不保留工程中某个组成部分发生了变化的信息,因为只要有一个源文件更新,就要重新编译工程中所有的源文件。而独立的MAKE实用程序在连接复杂程序的目标文件之前,仅重新编译那些更新过的源文件,然后简单地把重新编译得到的目标文件和不需重新编译的目标文件连接,产生一个新的可执行程序。

有关MAKE更多的信息可参看附录C。

至此,读者已学会在集成开发环境中如何编译、连接和运行Turbo C程序了。

希望知道在标准命令行中如何编译、连接和运行Turbo C程序的读者,可阅读第十章。

第四章 Turbo C 程序设计初步

本章介绍 C 语言的基本元素以及它们在程序设计中的应用。

在学习本章前,应首先阅读第二章“Turbo C 集成开发环境”,学会使用 Turbo C 的菜单和正文编辑,还应按第一章中介绍的方法安装好 Turbo C 系统。确定已建立了文件 TCCONFIG.TC。否则 Turbo C 不知道在何处寻找库文件(\LIB)和包括(\INCLUDE)文件。

完成上述工作,便可开始学习 Turbo C 程序设计技术。

4.1 建立第一个 Turbo C 程序

习惯上,我们总是把 Kernighan 和 Ritchie 所著的《程序设计语言 C》中的 Hello, World 程序作为第一个例子。

在 DOS 提示符下键入命令:

```
tc hello.c
```

便进入 Turbo C 集成开发环境。创建一个文件名 HELLO.C (如果盘上没有此文件)并激活 Turbo C 编辑程序窗口,在编辑窗口键入以下程序:

```
main( )
{
    printf( "Hello, World\n" );
}
```

程序输入完毕,进行下一步工作前,最好能将程序存入磁盘,否则,如果出现意外情况,将丢失所有输入的信息。要保存上述第一个 Turbo C 程序,请按 F2 键 或按 F10、F、S 键,即从 File 菜单中选择 Save。

4.1.1 编 译

保存源程序后,要运行该程序,须完成两件事:对它进行编译和连接。

编译源程序,只需按 Alt-F9 (将 Alt 键和 F9 键同时按下)即可,也可返回主菜单(按 F10 键),选择 Compile, 显示出 Compile (编译)菜单,再从中选择 Compile to OBJ。

编译完毕(编译只需一到两秒钟),屏幕上即会有一闪烁的信息: Success, press any key. 这时按任意键,编译窗口即消失,光标返回主菜单。

如果编译时产生警告或出错信息,则这些信息均显示在屏幕下部的信息窗口中,必须纠正这些错误。请检查输入机器的程序,确认它与给出的源程序一致后,重新进行编译。

4.1.2 运 行

完成编译后(没有任何错误),就准备运行程序。如果此时主菜单处于活动状态,按 **R**(Run)即可运行。如果当前状态是编辑状态,则可按 **Alt-R**,或者先返回主菜单(按 **F10** 键),然后再选择 **Run** 命令。这时,屏幕上会出现一个连接窗口,显示 Turbo C 正在连接的程序所需的库函数。

如果在上述执行过程产生出错信息,则可能是因为没有说明库文件在何处,如果此时光标仍停留在连接窗口,按任意键使 Turbo C 继续工作。按 **F10** 回到主菜单,选择 **Options** 显示 **Options** 菜单(或直接使用 **Alt-O** 显示 **Options** 菜单)。

选择 **Directories**,再从中选 **Library directories**,则出现一个 **Library directories** 窗口,输入库文件所在子目录的完全路径名,这个路径名可能是 **A:\TURBOC\LIB** 或 **C:\TURBO\LIB**。然后,按 **Enter** 键。

按 **Esc** 退出 **Directories** 菜单,再选择 **Store** (表示保存选择项),将出现一个含有 **TCCONFIG.TC** 配置文件的提示框,按 **Enter**,如果 Turbo C 询问是否要重写 **TCCONFIG.TC**,则打入 **"Y"**。

最后,按 **Esc** 退出 **Options** 菜单,按 **R** 运行程序。

要运行程序, Turbo C 需从有关库中复制出必须的库函数,因存储模式的默认值是 **Small**(有关存储模式的详细情况请看第八章),所以, Turbo C 将连接 **CS.LIB** 中的库函数。

连接完毕 Turbo C 即运行程序,屏幕被清除,顶部显示出 **Hello, world**,底部则出现了 **"press any key to return to Turbo C ..."**,按任意键,则返回到 Turbo C 主菜单。

这样,就建立了第一个 Turbo C 程序。

4.1.3 浏览产生的文件

现在 **File** 菜单中选择 **Quit** 命令退出 Turbo C,观察一下所产生的文件。

在 DOS 提示符下键入 **dir hello.***,并按 **Enter** 键,则屏幕上出现:

HELLO.C	42	1-01-80	9:25P
HELLO.OBJ	221	1-01-80	9:26P
HELLO.EXE	4486	1-01-80	9:26P

第一个文件 **HELLO.C** 是源文件(源代码)文本,在 DOS 提示符下键入 **type hello.c** 命令,可在屏幕上显示该文件的内容。可以看到,该程序只有 42 个字节。

第二个文件, **HELLO.OBJ** 是 Turbo C 编译程序产生的二进制机器指令(目标码),如果用 DOS 命令 **type** 显示该文件,则屏幕上出现混乱的信息。

最后的文件 **HELLO.EXE**,是 Turbo C 连接程序产生的实际可执行文件,它不仅包含了 **HELLO.OBJ** 目标文件,而且还包含了连接程序从库文件中拷贝出来的必要的库函数(如 **printf** 等),运行可执行文件只要在 DOS 提示符下键入可执行文件的文件名即可(不需打入扩展名 **.EXE**)。

在 DOS 提示符下键入 **hello** 并按 **Enter** 键,屏幕上出现信息 **Hello, world**,DOS 提示符返回到行首,这表示 **HELLO.EXE** 是一个可用的执行程序。

4.2 修改第一个 Turbo C 程序

在 DOS 下键入 `tc hello.c`, 则重新调入 Turbo C, 且第一个源文件 `hello.c` 被装入 Turbo C 编辑窗口, 修改该源文件, 使之具有一定的交互对话能力, 形式如下:

```
main( )
{
    char name [30];
    printf("What's your name? ");
    scanf("%s", name);
    printf("Hello, %s\n", name);
}
```

这样, 在原来的 `hello.c` 源程序中增加了 3 行: 第一行 (`char name [30]`); 说明了一个变量 `name`, 它是可容纳最长为 29 个字符的字符串 (字母、数学、标点符号等), 第二行是调用 `printf` 输出 `What's your name?` 第三行是调用 `scanf` 将输入的名字赋值给变量 `name`。

按 F2 键或 F10、F、S 将修改过的程序保存, 按 F10 返回主菜单, 然后按 **R** (**R**un) 运行。注意, Turbo C 能自动判断出源程序已经修改, 所以, 在运行之前 Turbo C 首先编译源程序。

运行程序, 将出现三种情况: 清除屏幕; 信息 `What's your name?` 出现在屏幕顶端; 光标停留在问号之后, 输入姓名并按 Enter, 屏幕上即显示出 `Hello, <your-name>`。注意, 在这里仅仅只读入了一个姓, 后面将解释产生这种现象的原因。现在, 按任意键返回 Turbo C。

编写程序过程中, 可能会产生错误或引起警告, 程序产生错误将阻止 Turbo C 生成目标文件, 而警告信息则指出可能发生问题 (但不一定影响程序运行), 错误信息和警告信息都在信息窗口显示, 有各种类型的错误信息和警告信息。详细情况请参看附录 D。

4.3 建立第二个 Turbo C 程序

修改第一个 Turbo C 程序以建立一个新程序, 将第一个源程序调入编辑窗口 (按 F10, E 或 Alt-E), 修改成如下形式:

```
main( )
{
    int a, b, sum;
    printf("Enter two numbers: ");
    scanf("%d %d", &a, &b);
    sum = a + b;
    printf("The sum is %d\n", sum);
}
```

实际上,对原来的源程序作了下述五个改动:

- 定义 name 改成了定义其他变量(a, b 和 sum, 均为整型);
- 改变了 printf 语句的输出内容;
- 改变了 scanf 语句中的格式串和变量;
- 增加了赋值语句 $sum = a + b$;
- 在最后的 printf 语句中改变了格式串和参数。

注意别混淆百分号(%)和符号(&)及反斜杠(\), 它们的含义将在下面解释。

4.3.1 程序记盘

现在,不能按F2键保存上述程序。如果按F2键,则上述程序仍以 HELLO.C 保存(该程序需用另一个名字来保存)。

按F10返回主菜单,再按F、W执行File/Write to命令,Turbo C则提示用户输入新的文件名,键入sum.c并按Enter,则程序以文件名SUM.C存入磁盘。按F10返回主菜单。

4.3.2 运行 SUM.C

按R选择Run命令,Turbo C首先编译该程序,如果产生出错信息,则请返回到编辑窗口,仔细检查一下,输入机器的程序是否与给出的例子一致。

编译无错误,则Turbo C将目标程序和必要的库函数连接,然后运行程序。屏幕将被清除,且在顶端显示出如下信息:

Enter two numbers;

程序等待用户输入两个整型值。输入时,两整型值用空格或tabs或回车键分开。在第二个值输入后,按Enter,这时,程序输出两个值的和。按任意键则返回Turbo C。

到目前为止,已利用C语言的一些基本元素编写出两个完全不同的Turbo C程序。那么,什么是C语言的基本元素呢?

4.4 程序设计的基本元素

编程的目的是为了解决问题。程序是通过处理信息或数据来解决问题的。编写程序时应注意以下几点:

- 程序应获取的信息
- 提供适当的空间保存信息
- 给出处理信息的语句
- 给用户输出结果

用户可组织程序语句以便能做到:

- 某些语句仅当一特定条件(或一组特定条件)为真时才能执行;
- 某些语句重复执行若干次;
- 某些语句能组成子程序在不同的地方执行。

本章仅描述程序设计的七个基本元素,即:输入,数据类型,运算,输出,条件执行,循环和子程序。仅这七种基本元素是不很全面的,但它们确是程序设计的基础。

大多数程序设计语言都具有这七个基本元素,同时许多语言,包括C语言,也都各自有增加的特性。要想很快掌握一种新的语言,首先要了解这七个基本元素在这一语言中是如何实现的,然后在此基础上学习更进一步的知識,下面先对每一个基本元素作一简短的介绍。

输出(Output):表示将信息写到屏幕、磁盘或输入/输出端口。

数据类型(Data Type):有常量、变量和结构。结构包含数(整型数和浮点数),文字(字符和字符串),地址(变量和结构的地址)。

运算(Operations):运算有赋值运算,算术运算(加法、除法等)和比较运算(相等、不等,等等)。

输入(Input):表示从键盘、磁盘、输入/输出端口读入信息。

条件执行(Conditional Execution):一组语句只有当一个特定条件为真时才执行(如果条件为假,则跳过这组语句)。

循环(Loops):一组语句重复执行固定的次数,或在某一条件为真时执行该组语句直到条件为假止。

子程序(subroutines):子程序是独立的有名的一组语句。它的名可以在程序的任何位置上出现,运行程序时,碰到子程序名,就执行该子程序。

下面介绍在Turbo C中如何使用这些基本元素。

4.4.1 输出

首先讨论输出似乎有点不恰当,但一个程序如果没有输出信息,则这个程序不会有多大用处。输出一般是指将一定形式的信息(文字和图形)写到屏幕、存储设备(软盘或硬盘)或输入/输出端口(串行口,打印机端口)等。

4.4.1.1 printf函数

在前面的程序编写过程中,已用到C语言最常用的输出函数printf,printf的功能是将输出信息在屏幕上显示出来,它的格式既简单又灵活。

printf(<格式串>, <项>, <项>, ...);

其中,格式串是首尾用双引号(")括起来的一个字符串,printf的目的是把格式串显示在屏幕上。显示时,printf按照格式串中的格式说明来显示格式串后面的项,例如:

printf("The sum is %d\n", sum)

格式串中的%d就是格式说明。格式说明由百分号(%)加一转换字符组成,用以指明数据类型以及数据格式。

对应每个格式说明都有一个确切的项,换句话说,printf中项的个数由格式串中格式说明的个数来决定,如果项的数据类型与格式说明的类型不一致,Turbo C在可能范围内把项的值自动转换成格式说明的类型。项本身可以是常量、变量、表达式和函数调用,简单地说,项应能产生一个与格式说明相容的值。

%d表示十进制整型量,表4.1给出一些常用格式说明的含义。

在百分号(%)和转换字符之间可设置域宽。例如,4位十进制整型数的域可写为%4d,输出时,以右对齐方式(输出值不足4位时,前面填充空格),整个域宽为4。

要输出一个百分号,只需插入%%即可。

格式串中的\n不是格式说明,它作为一个转义序列,表示在格式串中插入一个特殊控制字符,\n表示插入一个换行字符。故在输出格式串后,光标移动到下一行行首。

第十二章中列出了所有转义序列，表4.2给出几个常用控制字符的转义序列。

表 4.1 常用格式说明及其含义

格 式 说 明	含 义
%u	无符号整型量
%p	指针值
%f	小数形式的实型量
%e	科学记数形式的实型量
%c	字 符
%s	字符串
%x或%X	十六进制整型量

表 4.2 常用控制字符及其含义

控 制 字 符	含 义
\f	换页或清屏
\t	水平制表 tab
\b	退格符
\xhhh	ASCII 码为十六进制 hhh 的字符

如果要输出反斜杠，则插入\\。要详细了解printf的功能请阅读参考文献[13]。

4.4.1.2 其他输出函数：puts和putchar

puts的功能是在屏幕上输出一个字符串并换行。例如，把hello.c修改成：

```
main( )
{
    puts("Hello, world");
}
```

注意，在字符串后面没有了\n。在这里不需要写\n，因为puts自动换行。

putchar的功能是在幕上输出一个字符，语句putchar(ch)等价于printf("%c", ch)。

为什么要用puts和putchar代替printf呢？这是因为实现printf的程序非常庞大。除非有特殊情况，如数值输出或特别的格式输出，需使用printf，一般情况下，使用puts和putchar可使程序占用较小的空间，并可获得较高的运行速度。例如，使用puts的HELLO.C编译连接所得的.EXE文件比使用printf的HELLO.C所得的.EXE文件占用空间要小。

4.4.2 数据类型

编写程序时，要处理各种类型的信息，大多数信息可归结为四种基本类型：整型、浮点

型、文字和指针。

整型量表示可数的整数(如1, 5, -21, 725)。

浮点型量包括小数(例如3.14159)和幂指数形式数(例如 2.579×10^{24})。

文字是指字符(a, 2, !, 3)和字符串("This is only a test.")。

指针本身不是处理对象, 指针变量的值是另一个变量(对象变量)的地址。

4.4.2.1 浮点型

C语言以各种形式支持上述四种基本类型, 前面介绍的两个程序中已使用两种基本类型: 整型(int)和字符型(char)。现修改程序, 应用第三种基本类型: 浮点型(float), 在Turbo C编辑窗口将程序修改为:

```
main( )
{
    int a,b;
    float ratio;
    printf("Enter two numbers, ");
    scanf("%d %d", &a, &b);
    ratio = a/b;
    printf("The ratio is %f\n", ratio);
}
```

选择File/Write to命令将程序以名RATIO.C保存。按R编译、运行程序, 输入两个值10和3, 得到结果3.000000。

也许所预期的结果应是3.333333, 为什么得到的回答却是3.000000? 因为a和b均为整型量, 故a/b的结果亦是整型, 赋值给ratio时, 被转换成浮点型, 但赋值是在除法之后, 不是之前, 故得结果为3.000000。

将程序中的a, b改成浮点型(float), scanf格式串中的"%d %d"改成"%f %f", 保存程序且编译运行, 则此时的结果为3.333333。

浮点型量又可分为单精度浮点型(float)和双精度浮点型(double), 双精度浮点型变量所占内存字节数是单精度浮点型变量占内存字节数的两倍。Turbo C浮点型变量的取值范围可查阅第十二章。

4.4.2.2 三种整型

C语言除支持整型(int)外, 还支持短整型(short int)和长整型(long int), 通常简写成short和long。short、int、long的实际取值范围依赖于具体实现, 所有C语言都保证short变量所占字节数不会比long变量所占字节数多。在Turbo C中, short变量, int变量均为16位, long变量为32位。

4.4.2.3 无符号类型

C语言允许用户将某些类型(char, short, int, long)说明成无符号类型。这表示被说明的类型不再具有负值, 仅包含了非负值(大于或等于零)。

无符号类型变量的取值比有符号类型变量的取值要大。例如, Turbo C整型量的取值范围是-32768—32767, 而无符号整型量(unsigned int)的取值范围0—65535, 两者都占有相同的空间(Turbo C中是16位), 但使用上有很大的差别, 详细情况参见第十二章。

4.4.2.4 字符串定义

字符串不是C语言中独立的数据类型，但C语言提供了两种不同的方法定义字符串。一种是利用字符数组，另一种是用一个字符指针。

使用字符数组

在File菜单中选择Load命令，把HELLO.C调入编辑窗口，修改程序，使之如下：

```
main( )
{
    char msg [30] ;
    strcpy(msg, "Hello, world");
    puts(msg);
}
```

msg后面的[30]说明了msg是一个由29个字符组成的字符数组，占据29个字符所占的空间（第30个字符是空字符——\0——表示字符串的结束）。变量msg本身不含有字符值，它是这29个字符变量的首地址（内存中的某个位置）。

编译程序遇到语句strcpy(msg, "Hello, world")时要做两件事：

- 在目标代码文件中建立一个字符串"Hello, world"，串后紧跟一个空字符(\0)（它的ASCII码为0）。

- 调用子程序strcpy，每次从字符串中拷贝一个字符到msg指出的内存空间，直到"Hello, world"后的空字符为止。

执行puts(msg)，即把msg的值（字符串首址）传送给puts，puts检查该地址中字符是否为空字符。若是，则puts执行完毕；否则，输出字符，然后将地址加1，重复上述过程。

由于上述工作均依赖于空字符(\0)，故可认为C语言中的字符串是以空字符为结束符的，即：一串字符紧接着一个空字符。这个方法消除了对字符串长度的任何限制，在内存空间足够的情况下，字符串可任意长。

使用字符指针

定义字符串的第二种方法是利用字符指针，现修改程序HELLO.C呈成如下形式：

```
main( )
{
    char *msg;
    msg = "Hello, world";
    puts(msg);
}
```

msg前的星号(*)说明msg是一个指向字符的指针，换句话说，msg可包含一些字符的地址。不过，编译程序不保留空间存储字符，也不给msg任何初始值。

当编译程序遇到语句msg="Hello, world"时，做两件事：

- (1) 像字符数组一样，编译程序在目标代码文件中建立一个字符串"Hello, world"，紧跟一个空字符(\0)。

- (2) 把字符串首地址——字符H的地址赋给msg。

puts(msg)与前面一样，逐个输出字符，直到遇到空字符为止。

用数组和用指针定义字符串，两种方法之间有一些微小的差别，将在下一章中讨论。

4.4.2.5 标识符

到目前为止,我们已经定义过变量名,但没有考虑定义变量名有什么限制。现在,就讨论对变量名的限制。

给予常量、变量、数据类型、函数的名即是标识符,目前使用的一些标识符包括:

`char, int, float` 预先定义的数据类型

`main` 程序主函数

`name, a, b, sum, msg, ratio` 用户定义的变量

`scanf, printf, puts` 预先说明的函数

Turbo C 关于标识符有下述几条规则:

(1) 标识符必须以字母(a...z或A...Z)或下横线(_)开头。

(2) 标识符除首字符外的其余部分可以是字母,下横线、斜杠(/)或数字(0...9)。其他字符不允许出现在标识符中。

(3) 标识符与字母大小写有关。也就是说,小写字母(a...z)不同于大写字母(A...Z),例如,标识符`indx`、`Indx`和`INDX`是不相同的。

(4) 标识符的有效长度是32个字符(超过32个字符时,前32个字符有效)。

4.4.3 基本运算

一旦程序中有了数据(变量有了值),接着就要对数据进行各种运算。C语言提供了较为丰富的运算符。

4.4.3.1 赋值运算符

最基本的运算就是赋值运算,如 `ratio = a/b` 或 `ch = getch()`。在C语言中,赋值运算符是单个等于号(=),赋值运算即把等号右边的值赋给等号左边的变量。

此外,还可以实现多重赋值,如 `sum = a = b`,在这种情况下,赋值的次序是从右到左,故首先是b的值赋给a,再赋给sum,使三个变量具有相同的值(b的值是原始值)。

4.4.3.2 单目运算符和双目运算符

C语言支持的双目算术运算符如下:

乘(*)

除(/)

取模(%)

加(+)

减(-)

Turbo C 支持单目减(`a+(-b)`),即对运算分量取负。作为ANSI的扩充,Turbo C还支持单目加(`a+(+b)`)。

4.4.3.3 增1(++)和减1(--)运算符

C语言有一些较为特殊的单目运算符和双目运算符。其中最主要的单目运算符是增1运算符(++)和减1运算符(--)。分别表示把运算分量加1和减1,增1运算符(++)和减1运算符(--)可以前置,也可以后置,但含义不完全相同,考虑下面两个表达式:

`sum = a + b ++;`

`sum = a ++ + b;`

前者的意义为:a加b,赋值给sum,b再增加1。

后者的意义为：b增加1，再a加b，然后将值赋给sum。

这些运算符非常有效，但是必须在正确理解之后才能使用。现修改SUM.C如下，在运行程序前设想一下它输出的结果。

```
main ( )
{
    int a, b, sum;
    char*format;
    format="a=%d b=%d sum=%d\n";
    a=b=5;
    sum=a+b; printf(format, a, b, sum);
    sum=a++ + b; printf(format, a, b, sum);
    sum=++a + b; printf(format, a, b, sum);
    sum=--a + b; printf(format, a, b, sum);
    sum=a-- + b; printf(format, a, b, sum);
    sum=a+b; printf(format, a, b, sum);
}
```

4.4.3.4 按位运算符

C语言提供下列按位运算符：

左移 (<<)	按位或OR ()
右移 (>>)	按位异或XOR (^)
按位与AND (&)	求反NOT (~)

按位运算使得用户能执行更低一级的运算。现以下的程序来考察按位运算符的作用。

```
main ( )
{
    int a, b, c;
    char*format1, *format2;
    format1="%04X %s %04X=%04X\n";
    format2="%c %04X=%04X\n";
    a=0xFF0; b=0xFF0;
    c=a<<4; printf(format1, a, "<<", 4, c);
    c=a>>4; printf(format1, a, ">>", 4, c);
    c=a&b; printf(format1, a, "&", b, c);
    c=a|b; printf(format1, a, "|", b, c);
    c=a^b; printf(format1, a, "^", b, c);
    c=~a; printf(format2, "~", a, c);
    c=-a; printf(format2, "-", a, c);
}
```

同样，在运行这个程序前，请设想一下它的输出结果。注意，在格式说明中使用了域宽说明，%04X表示输出结果是以0为首的四位十六进制数。

4.4.3.5 组合运算符

书写含有多个运算符的表达式时，C语言提供了一种简写方式。可以把赋值运算符和其他运算符组合在一起写，一般的表达式形式为：

〈变量〉 = 〈变量〉 〈运算符〉 〈表达式〉

可写成：

〈变量〉 〈运算符〉 = 〈表达式〉

下面给出一些例子：

a = a + b;	可写成	a += b;
a = a - b;	可写成	a -= b;
a = a * b;	可写成	a *= b;
a = a / b;	可写成	a /= b;
a = a % b;	可写成	a %= b;
a = a << b;	可写成	a <<= b;
a = a >> b;	可写成	a >>= b;
a = a & b;	可写成	a &= b;
a = a b;	可写成	a = b;
a = a ^ b;	可写成	a ^= b;

4.4.3.6 地址运算符

C语言支持两个特殊的运算符：取地址运算符(&)和间接运算符(*)。

取地址运算符&返回指定变量的地址，如果sum是一个整型变量，则&sum为该变量的存储地址(内存中的位置)；如果msg是一个字符指针，则*msg是msg所指的字符。不妨输入下面程序，观察所得结果。

```
main( )
{
    int sum;
    char * msg;
    sum = 5 + 3;
    msg = "Hello, there\n";
    printf("sum = %d  &sum = %p\n", sum, &sum);
    printf("*msg = %c  msg = %p\n", *msg, msg);
}
```

第一行输出两个值：sum的值(8)和sum的地址(编译程序给出)，第二行也输出两个值：msg所指的字符(H)和msg的值，msg的值即是字符H的地址(由编译程序给出)。

4.4.4 输入

C语言具有多种输入方式，可以从一个文件输入，也可以从键盘等设备输入，有关Turbo C的输入功能，可参看scanf, read和第十二章。

4.4.4.1 scanf 函数

scanf通常用于交互式输入，它的功能与printf的功能相当，其格式为：

scanf(〈格式串〉, 〈地址〉, 地址, ...)

`scanf` 的格式说明基本上与 `printf` 的格式说明相同,如: `%d` 表示整型, `%f` 表示浮点型, `%s` 表示字符串等。

然而, `scanf` 与 `printf` 的一个重要差别是: `scanf` 中, 格式串后的项必须是一个地址, 而不是一个值。程序 `SUM.C` 中有一个函数调用:

```
scanf("%d %d", &a, &b);
```

这个调用表明, 需输入两个十进制(整型)数, 且中间用空格分开, 第一个赋给 `a`, 第二个赋给 `b`。注意, 它使用了取地址运算(`&`)把 `a` 和 `b` 的地址传递给 `scanf`。

(1) 空白

在两个格式说明 `%d` 之间的空格有特殊的意义, 它表示在两个值之间可有任意多个空白, 即空格、制表符(`tab`)和新行的任意组合。在一般情况下, C 语言编译程序忽略空白。

如果不用空格而用逗号(`,`)来分隔两个数, 应怎样做呢? 只要把 `scanf` 的格式串改成:

```
scanf("%d, %d", &a, &b);
```

即可在输入两值之间输入逗号(`,`)把两值分隔开来。

(2) 传送地址给 `scanf`

怎样才能输入一个字符呢? 输入并运行下面的程序:

```
main( )
{
    char name [30] ;
    printf("what is your name: ");
    scanf("%s", name);
    printf("Hello, %s\n", name);
}
```

因为 `name` 是一个字符数组, 故 `name` 的值就是数组本身的地址。所以, 在 `name` 前不需加运算符 `&`, 仅用 `scanf("%s", name)` 即可。

注意, 这里使用了数组方法(`char name[30];`)而没用指针方法(`char *name`), 这是因为数组说明能明确指出为字符串保留的空间, 而指针说明却不能。如果使用指针方法 `char *name`, 则需要明确地为 `*name` 分配内存空间。

4.4.4.2 用 `gets` 和 `getch` 输入

用 `scanf` 输入字符串会引起一些问题, 再一次运行上述程序, 输入姓名。请注意, 输出的结果只是姓而没有名。这是因为 `scanf` 认为姓后面的空格标志着字符串结束。

解决这个问题有两种办法, 第一种是:

```
main( )
{
    char first[20], middle[20], last[20];
    printf("What is your name: ");
    scanf("%s %s %s", first, middle, last);
    printf("Hello, Dr. %s, or should I say %s? \n", first, last);
}
```

在这个例子中, 假设姓名是三个字, `scanf` 在接收到三个字符串以后才继续工作。但怎样才能把这完整的姓名作为一个字符串(包括空格在内)读入呢? 请看第二种解决办法。

```

main( )
{
    char name [60] ;
    printf("What is your name: ");
    gets(name);
    printf("Hello, %s\n", name);
}

```

gets能读入所有输入的字符，直到 Enter 为止。输入字符串中不包括Enter，但在字符串最后加上了一个空字符(\0)。

getch的功能是从键盘上读入一个字符，但不回显(这与 **scanf** 和 **gets**不同)，注意 **getch**不是以ch为参数的，它是一个char类型的函数，它的值可直接赋给ch。

4.4.5 条件语句

在讨论条件语句时，还将涉及一些运算符(关系运算符和逻辑运算符)和一些有关表达式的复杂情况。

4.4.5.1 关系运算符

关系运算就是进行两个值的比较，根据比较结果是真还是假产生结论。如果比较结果是假，则结论值为0，如果是真，则结论值为1。C语言中有下列关系运算符：

>	大于	<=	小于或等于
>=	大于或等于	=	等于
<	小于	!=	不等于

为什么要关心某些情况是真还是假呢？请装入并运行程序 **RATIO.C**，使第二个数的输入值为0，观察发生什么现象。这时，屏幕上会出现一个 Divide by zero 的出错信息且停止运行程序。请对程序作如下修改并重新运行之。

```

main( )
{
    float a,b,ratio;
    printf("Enter two numbers: ");
    scanf("%f %f", &a, &b);
    if (b == 0.0)
        printf("The ratio is undefined\n");
    else
    {
        ratio = a/b;
        printf("The ratio is %f\n", ratio);
    }
}

```

在调用**scanf**之后的语句即是条件语句。读作：“如果表达式(b=0.0)的值为真，则立即执行**printf**，如果表达式的值为假，则把a/b赋给 ratio，然后调用**printf**。”

现在，如果输入0作为第二个值，则程序输出信息：

The ratio is undefined

按任意键，返回到Turbo C，若第二个值不等于0，则程序进行计算且输出比率并等待按键返回到Turbo C——这都是if语句所起的作用。

4.4.5.2 逻辑运算符

C语言提供了三种逻辑运算符：AND(&&)、OR(||)和NOT(!)，注意不要和前面讨论的按位运算符(&, |, ~)混淆，逻辑运算符的运算分量是逻辑值(真和假)和关系表达式。

逻辑运算符与相应的按位运算符的区别在于：

(1) 逻辑运算符总是产生一个值0(假)或1(真)，而按位运算符却是逐位地运算。

逻辑运算符&&和||将能缩短运算周期，假设有一个表达式为exp1 && exp2，如果exp1为假，则整个表达式的值肯定为假，不需再计算exp2。同样，对exp1 || exp2，如果exp1为真，则exp2就不需再进行计算。

4.4.5.3 关于表达式的进一步说明

在讨论循环语句之前，有必要对表达式作进一步的说明，考虑表达式如(b == 0.0)和(a <= q * r)是非常直观简单的。C语言允许用户书写比上述表达式复杂得多的表达式，由于无法说明表达式究竟可以复杂到何种程度，所以仅给出一些例子来说明。

(1) 赋值表达式

用圆括号括起来的赋值语句是一个表达式，这个表达式具有与被赋值的变量相同的值。

例如，表达式(sum = 5 + 3)的值是8，故表达式((sum = 5 + 3) <= 10)的值为真(因为8 <= 10)。还有更奇异的例子，例如：

```
if ( (ch = getch()) == 'q'
    puts("Quitting, huh? \n");
else
    puts("Good move, we'll have another go at it.\n")
```

可以想象出上述语句完成什么动作。当程序运行到表达式(ch = getch()) == 'q'时，等待输入一个字符且赋给ch，然后将ch与字母q进行比较。如果输入字符等于q，则输出信息"Quitting, huh?"，否则，输出信息"Good move..."。

(2) 逗号运算符

应用逗号运算符(,)，可将多个表达式置于一个圆括号中成为一个表达式，括号内表达式赋值的次序是从左到右，整个表达式的值规定是最后被赋值变量的值。例如，如果变量oldch和ch均是字符类型，则表达式

```
(oldch = ch, ch = getch())
```

的赋值次序是：把ch赋给oldch，再从键盘上读入一个字符赋给ch，并且整个表达式的值是ch的值，例如：

```
ch = 'a';
if ( (oldch = ch, ch = 'b') == 'a' )
    puts("aye");
else
    puts("bee");
```

4.4.5.4 if 语句

再观察一下前面例子中的 `if` 语句, 就可知道, `if` 语句的一般格式为:

```
if(值)
```

```
    语句1;
```

```
    else
```

```
    语句2;
```

此处的值是任何可以得出整型值(或可以转换成整型值)的表达式, 如果值非零(即真), 则执行语句1; 否则, 执行语句2。

对 `if-else` 语句的一般形式还有两点必要的说明:

(1) `else` 语句2部分是任选的, 也就是说, 形式为

```
if(值)
```

```
    语句1;
```

的 `if` 语句也是有效的。

(2) 当一个特定表达式的值为真(或假)时, 要执行多个语句, 就需要将这多个语句构成复合语句。复合语句是用一对花括号括住的一组语句, 其中每一个语句以分号(;)结束; 即:

```
{语句1; 语句2; ...}
```

在求比率的例子中, `if` 子句中的语句1是单个语句,

```
if (b == 0.0)
```

```
    printf("The ratio is undefined.\n");
```

而 `else` 子句中的语句2是一个复合语句:

```
else
```

```
{
```

```
    ratio = a/b;
```

```
    printf("The ratio is %f\n", ratio);
```

```
}
```

细心的读者也许注意到, 整个程序体(函数 `main`)就是一个复合语句。

4.4.6 循环

正如一些语句(或一组语句)的执行是有条件的, 有一些语句需要重复执行, 这种结构就称为循环。

C 语言中有三种基本的循环方式: `while` 循环, `for` 循环和 `do...while` 循环。下面分别对它们进行讨论。

4.4.6.1 while 循环

`while` 循环是最一般的循环, 它可以代替其他两种循环, 也就是说 `while` 循环总能满足用户的需要, 而其他两种循环只是在某些情况下使用起来较为方便。装入 `HELLO.C` 并修改成如下形式:

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

```
main( )
```

```
{
```

```
    char ch;
```

```

int len,
len = 0;
puts("Type in a sentence, then press <Enter>"),
while ( (ch = getch()) != '\n' ) {
    putchar(ch);
    len++;
}
printf("\nYour sentence was %d characters long\n", len);
}

```

程序允许用户输入一串字符，并计数键入的次数，直到输入Enter(\n)为止。最后，程序输出用户输入的字符个数(不包括Enter)。程序中使用putch是为了回显用户输入的字符，因为getch不作回显。

注意，程序中使用了赋值表达式，故程序能读入一个字符且判断它是否为Enter，当读入字符不是Enter时，则回显该字符并使len加1。

while语句的一般格式为：

```

while(表达式)
    语句

```

在此，表达式应能产生一个为零或非零的值，语句可以是单个语句，或者是一个复合语句。

while循环首先判断表达式的值是否为真，若为真，则执行语句并再对表达式进行判断，若表达式为假，**while**循环结束，程序继续执行下面的语句。请看以HELLO.C为基础的另一个**while**循环的例子，

```

main( )
{
    char* msg;
    int indx;
    msg = "Hello, world";
    indx = 1;
    while (indx <= 10)
    {
        printf("time # %2d: %s\n", indx, msg);
        indx++;
    }
}

```

编译并运行上述程序，输出结果为：

```

time #1: Hello, world
time #2: Hello, world
time #3: Hello, world
:
time #10: Hello, world

```

程序中的printf语句执行了十次，而在这执行过程中indx的值从1增到10。

细想一下，若程序写成如下形式可使循环紧凑一些。

```
indx = 0
while(indx++ < 10)
    printf("time # %2d: %s\n", indx, msg);
```

在继续学习for循环前，请读者仔细研究上述while循环语句，弄明白为什么它的功能和第一种写法是一样的。

4.4.6.2 for循环

许多程序设计语言都提供了for循环，C语言也不例外，C语言的for循环非常灵活，非常有效。

for循环的基本思想是，由一个变量（称为循环控制变量或索引变量，它在某个值域范围内每次改变一定的量）控制重复执行一组语句的次数。例如，把前面装入的程序修改成：

```
main( )
{
    char * msg;
    int indx;
    msg = "Hello, world";
    for (indx = 1; indx <= 10; indx++)
        printf("time # %2d: %s\n", indx, msg);
}
```

运行程序可以发现程序输出结果与原来用while循环时一样。for循环语句的一般格式为：

```
for(表达式1; 表达式2; 表达式3)
    语句
```

和while循环一样，for循环仅重复执行一个语句，但这个语句可以是一个复合语句（{...}）。

for后面圆括号内，有三个用分号隔开的表达式，它们的含义为：

- 表达式1 给循环变量赋初值；
- 表达式2 是循环测试条件；
- 表达式3 用于修改循环变量。

for循环等价于：

```
表达式1；
while(表达式2) {
    语句；
    表达式3；
}
```

for循环的三个表达式均可以缺省，但分号必须保留。表达式2缺省时，其值恒为1（即真），表示不停地循环（称为无限循环）。

利用逗号，可用多个表达式（各表达式之间用逗号隔开）替代表达式1、表达式2和表达式3。例如，将HELLO.C修改成：

```

main( )
{
    char*msg;
    int up, down;
    msg="Hello,world";
    for (up=1,down=9;up<=10;up++, down--)
        printf("%s,  %2d down,%2d to go\n", msg,up,down);
}

```

上述程序的for循环中的表达式1和表达式3均是由两个表达式组成，分别对变量up和down进行初始化和修改处理。C语言对for循环中表达式的复杂程度不加限制。

4.4.6.3 do...while循环

最后一种循环是do...while循环，现把RATIO.C修改成如下形式：

```

main( )
{
    float a,b,ratio;
    char ch;
    do
    {
        printf("Enter two numbers, " );
        scanf("%f %f", &a, &b);
        if (b == 0.0)
            printf("The ratio is undefined\n");
        else
        {
            ratio=a/b;
            printf("The ratio is %f\n",ratio);
        }
        printf("Press 'q' to quit,any other key to continue");
    }
    while ( (ch = getch()) != 'q' );
}

```

该程序计算比率，等待用户输入一个键。如果输入字符为q，则while中表达式为假，循环即结束；如果是其他非q的键，则表达式为真，重复循环。

do...while循环的一般格式为：

do 语句 while (表达式);

while循环和do...while循环的主要差别在于do...while循环中的语句至少能执行一次。这与Pascal语言中的repeat...until循环相似，但两者也有差别：repeat循环总是在条件为假时执行语句，而do...while则是在条件为真时执行。

4.4.7 函 数

前面介绍了怎样应用条件语句和循环语句,现在考虑这样的问题:怎样使同样的一组语句能够处理不同的数据呢?以及怎样使同样一组语句能够出现在程序的不同位置上?回答是:把这一组语句组成一个子程序,必要时对它进行调用。

在C语言中,所有的子程序都称为函数,理论上说,每个函数需有一个返回值,事实上,许多函数的返回值都可以忽略,近来推出的C语言(包括draft ANSI C标准和Turbo C)允许用户说明void类型的函数,表示这类函数不需有返回值。

C语言可说明和定义函数,一个函数被说明后,程序中的其他函数(包括主函数)就能调用之。定义一个函数时,必须给出函数体。例如,把RATIO.C重写为:

```
/*函数说明*/
void get_parms(float *p1,float *p2),
float get_ratio(float dividend,float divisor),
void put_ratio(float quotient);
const float INFINITY=3.4E+38;
/*主函数: 程序开始*/
main( )
{
    float a, b, ratio;
    char ch;
    do
    {
        get_parms( &a, &b);          /*取参数*/
        ratio=get_ratio(a,b);        /*计算比率*/
        put_ratio(ratio);            /*输出结果*/
        printf("press q to quit,any other key to continue.*");
    }while((ch=getch()) != 'q');
}
/*主函数结束*/
/*函数定义*/
void get_parms(float * p1,float *p2)
{
    printf("Enter two numbers,");
    scanf("%f %f",p1,p2);
}
float get_ratio (float dividend, float divisor)
{
    if (divisor == 0.0)
        return(INFINITY);
    else
```

```

    return(dividend/divisor);
}
void put_ratio(float ratio)
{
    if(ratio == INFINITY)
        printf("The ratio is undefined\n");
    else
        printf("The ratio is %f\n", ratio);
}

```

4.4.7.1 程序分析

程序开始的三行是函数说明，函数说明的目的是为了说明函数的类型以及参数的类型和参数个数，说明参数类型和参数个数是为了便于出错检查。

第四行定义了一个浮点常量INFINITY（在C语言中，常量名规定用大写字母）。该常量有一个非常大的正值——大约是浮点型数最大正值——该值常用来标志除数为0时所得的结果。注意，因为该常量在此处说明，故它在所有函数（包括主函数）中都可以出现。

下面就是主函数，它是程序的主体，任何一个C语言程序均有主函数(main())。程序从主函数开始执行，且执行完主函数，程序就结束并返回Turbo C（如果从DOS下开始运行，则返回DOS）。

主函数可设置在程序的任何部位，一般地，它总是第一个函数，接着是一些函数定义或其他全程量说明。

在主函数后是三个函数说明的实际定义，下面依次进行讨论。

(1) get_parms函数

get_parms不返回固定类型的值，故把它说明成void类型，它的功能是读入两个值并保存它们。为此，必读给**get_parms**两个参数，作为保存值的地址。注意，这两个参数不是浮点型量而是浮点型指针，也就是说，它们是浮点型变量的地址。

请注意，主函数中调用**get_parms**时，参数是&a和&b，而不是a和b。而**get_parms**调用scanf时，在p1和p2之前没有取地址运算符，因为p1和p2已经是地址，分别是a和b的地址。

(2) get_ratio函数

get_ratio函数返回两个参数(dividend和divisor)的值(浮点型)运算的结果(浮点型)。返回的值依赖于除数(divisor)是否为0，如果除数为0，**get_ratio**返回INFINITY。如果除数不等于0，**get_ratio**返回计算出的比率。请注意return语句的格式。

(3) put_ratio函数

put_ratio没有返回值，故类型为void，它只有一个参数——ratio——用于决定输出信息。如果ratio为INFINITY，则认为结果未定义，否则，输出比率ratio。

4.4.7.2 全程量说明

在任何函数(包括主函数)外说明的常量、数据类型和变量都被认为是从那里开始的全程量。也就是说，它们只有在被说明之后才可以使用。例如，如果把INFINITY的说明放在程序结尾处，则在编译时会得到两个出错信息，一个在**get_ratio**中，另一个在**put_ratio**中，理由是使用了未说明的标识符。

4.4.7.3 函数说明

函数说明有两种不同的风格：“传统的”和“现代的”。

传统的风格在许多C语言文本和程序中都能找到，其格式为：

类型 函数名()；

它描述了函数的名(函数名)和函数返回值的类型(类型)，但它不给出任何参数信息，故不能进行出错检查或类型转换。如果用这种风格重写RATIO.C中的函数说明，即为：

```
void get_parms( );
float get_ratio( );
void put_ratio( );
```

现代的风格使用了一个来源于ANSI的扩充，称为函数原型(function prototype)的结构，在说明中增加了参数信息：

类型 函数名(形式参数，形式参数，...)

形式参数的格式是下列形式之一：

```
类型
类型 参数名
...
```

换句话说，每一个形式参数可描述成仅有一个数据类型或一个数据类型加一个参数名，如果函数的参数个数不确定，可用省略号(...)表示最后的参数。

这是一种较好的风格，因为它能使编译程序检查在实际调用函数时，形式参数和实际参数的个数及类型是否一致，并能使编译程序在可能的情况下进行参数的类型转换。RATIO.C中的函数说明使用了这种风格，有关函数原型更详细的信息请参看第八章和第十二章。

4.4.7.4 函数定义

与函数说明一样，函数定义也有两种写法：传统的和现代的。传统的函数定义格式为：

```
类型 函数名(参数名)
参数说明
{
    局部量说明；
    语句；
}
```

现代的函数定义格式把参数的说明移到了函数名后的圆括号中：

类型 函数名(形式参数；形式参数，...)

在这里，形式参数包含了参数的类型和参数名，这就使得函数定义的第一行与函数原型非常相似，仅有一点区别：在函数定义第一行后没有分号(;)，而函数原型总是以分号(;)结束。函数get_parms传统的定义方法：

```
void get_parms(p1,p2)
float *p1,float *p2;
{...}
```

现代的定义方法为：

```
void get_parms(float *p1, float *p2)
{...}
```


注意，在一个函数内说明的局部量(常量、变量、数据类型)仅在此函数内部有效。还要注意，C语言不允许在函数体内再定义函数。

程序中的函数定义可按任意次序排列，它们总被认为是整个程序的全程量(即可以先使用后定义)。在定义或说明函数前使用函数要注意：当编译程序遇到一个新的函数时，便假定它的类型是int，如果在后面又定义该函数的类型不为int(如char*)，则会得到一个出错的信息。

4.4.8 注 解

有时，在程序中插入一些对变量、函数或语句的解释，能增加程序的可读性，这些解释就称为注解，C语言与其他程序设计语言一样，也允许用户在程序中插入注解。

注解是以/*开头，*/结束的一个字符序列，编译时，编译程序忽略其中的内容。

注解可以不止一行。如：

```
/* This is a long  
comment, extending  
over several lines.*/
```

请观察一下最后的RATIO.C例子中的注解。

本章学习了建立、编译和运行Turbo C程序，并且学习了C语言的七个基本元素以及它们在Turbo C程序设计中的初步应用。

下一章将对每一个基本元素的应用作进一步的介绍。

第五章 Turbo C 进一步的程序设计技术

在前一章中，给出了一个使用 Turbo C 编程的教学例子，现在介绍 C 程序设计中的其他程序设计技术。

本章包含下列内容：

- 数据结构，包括指针、数组和结构。
- switch 语句。
- 控制流命令，包括 return, break, continue, goto 和条件表达式操作符 (?:)。
- C 程序设计风格，特别是有关 C 的一些新特征。
- C 程序员关心的一些公共问题。

5.1 数据结构

前一章介绍了基本的数据类型，例如整型、浮点型、字符型和它们的变形。这里介绍怎样使用这些元素来建立数据结构，即数据元素的集合。首先介绍 C 语言中的一个重要概念：指针。

5.1.1 指针

至此所介绍的绝大多数变量都用来保存数据，即程序所处理的实际信息。但是有时想要跟踪一些数据存放的位置，而不是它的值，这时就需要指针。

计算机把程序和有关的数据保存在存储器中（常称为 RAM，意思是随机存取存储器），在它的最低层，计算机的存储器由位组成，即一种微小的电子电路，它能够记忆 0 或 1。

8 位组成一个字节，两个字节可称为字。四个字节称为长字；在 IBM PC 上，16 字节称为一段（paragraph）。

在计算机的存储器中，每个字节有一个唯一的地址，就像街道上的每所房屋的门牌号码一样。但不同的是，在存储器中连续的字节有连续的地址。如果某个字节的地址为 N ，那么它前面字节的地址为 $N-1$ ，而它后面字节的地址为 $N+1$ 。

指针是保存数据地址的变量，并不保存数据本身。指针的用处是：第一，可以使用指针指向不同的数据和不同的数据结构，改变指针所包含的地址，就可以处理（赋值、读取和更改）在各种位置的信息。例如，仅使用一个指针，就可处理一个结构的链表。第二，在程序执行时，使用指针可建立新的变量。C 允许用户程序申请少量的存储器（使用字节数），然后返回一个地址，该地址存储在一个指针中。这就是存储器动态分配的概念。利用这种技术，用户程序可有效地使用存储器。第三，使用指针可存取数据结构中的不同位置。例如：数组、字符串或结构。一个指针实际上只能指向存储器中的一个位置（即一个段：位移量地址），而移动指针，则可存取存储器中的任何连续的字节。

由上可知，指针是非常方便的。但在 C 语言中怎样使用指针呢？首先，必须对指针进行说明，如下面的程序：

```

main ( )
{
    int ivar, *iptr;
    iptr = &ivar;
    ivar = 421;
    printf("Location of ivar: %p\n", &ivar);
    printf("Contents of ivar %d\n", ivar);
    printf("Contents of iptr %p\n", iptr);
    printf("Value pointed to: %d\n", *iptr);
}

```

这个 main 函数说明了两个变量: ivar 和 iptr。第一个变量 ivar 是一个整型变量, 它用来保存类型为 int 的值; 第二个变量 iptr 是一个指向整型变量的指针, 它保存类型为 int 的值的地址。因为在说明 iptr 时, iptr 的前面有一个星号 (*), 所以可知 iptr 是一个指针。在 C 语言中, * 符号作为间接操作符。

在 main 函数中, 赋值语句的含义如下:

ivar 的地址赋给 iptr;

整数值 421 赋给 ivar。

在前面章节中介绍的地址操作符 (&) 取 ivar 变量的地址。

输入并且运行这个程序, 可得到如下的输出结果:

Location of ivar: 166E

Contents of ivar: 421

Contents of iptr: 166E

Value pointed to: 421

前面两行显示 ivar 变量的地址和内容。第三行显示 iptr 包含的地址, 很明显它就是变量 ivar 的地址, 即程序决定建立 ivar 变量在存储器中的位置。输出的最后一个值是存储在该地址的数据, 该数据已经赋给变量 ivar。

注意, 对 printf 函数的第三次调用使用了表达式 iptr 取它的内容, 即 ivar 的地址; 然后最后一次 printf 函数的调用使用了表达式 *iptr 取存储在该地址的数据。

现对上面的程序稍作改动, 使其变为:

```

main ( )
{
    int ivar, *iptr;
    iptr = &ivar;
    *iptr = 421;
    printf("Location of ivar: %p\n", &ivar);
    printf("Contents of ivar: %d\n", ivar);
    printf("Contents of iptr: %p\n", iptr);
    printf("Value pointed to: %d\n", *iptr);
}

```

该程序仍旧把 ivar 的地址赋给 iptr, 但是改动之处将 421 赋给 *iptr, 而不是 ivar。

其结果是什么呢？实际上该程序的结果和前面程序的结果一样。为什么结果相同呢？因为语句 `*iptr = 421` 和语句 `ivar = 421` 的作用一样，`ivar` 和 `*iptr` 都指向相同的存储器位置，这两个语句都把值 421 赋给该位置。

5.1.1.1 动态分配

下面是上述程序的另一种版本：

```
#include <alloc.h>
main ( )
{
    int *iptr;
    iptr = (int*) malloc ( sizeof ( int ) );
    *iptr = 421;
    printf ( "Contents of iptr: %p\n", iptr );
    printf ( "Value pointed to: %d\n", *iptr );
}
```

这种版本取消了变量 `ivar` 的说明，而将函数 `malloc` 的返回值赋给 `iptr`，`malloc` 函数在 `ALLOC.H` 文件中说明（因此在函数的开头要使用预处理指令 `#include`）。然后，把值 421 赋给 `*iptr`，即赋给 `iptr` 指向的地址。如果运行这个程序，则 `iptr` 中的值不同于前面程序，但是 `*iptr` 仍然是 421。

语句 `iptr = (int*) malloc (sizeof (int))` 完成什么样的功能呢？下面将该语句分解予以说明：

（1）表达式 `sizeof (int)` 返回类型为 `int` 的变量所需要的字节数，使用 IBM PC 上的 Turbo C，则该表达式返回的值是 2。

（2）函数 `malloc (num)` 分配 `num` 个连续字节的有效（即未使用的）存储器空间，并返回这些字节的起始地址。

（3）表达式 `(int*)` 表示该起始地址作为类型为 `int` 的一个指针，这称为强置类型转换（type casting），在这种情况下，Turbo C 不需要它；但是很多其他的 C 编译程序确实需要它。如果不使用它，则会得到错误信息：Non-portable pointer assignment。

（4）最后，这个地址存储在 `iptr` 中，这表示已经动态地建立了一个整型变量，可表示成 `*iptr`。

综上所述，整个语句的含义为：在存储器中为类型 `int` 的变量分配足够的存储空间，然后将该存储空间的起始地址赋值给 `iptr`，即一个类型为 `int` 的指针。

所有这一切都是必须的吗？回答是肯定的。因为不这样就不能保证 `iptr` 指向没有使用的存储器区域，或许 `iptr` 中有一些值，即它是使用的地址，该存储器段可能用于其他目的。使用指针的规则是简单的，在使用指针之前，总是将地址赋给指针，而且在没有赋地址给 `iptr` 时，不能赋整型值给 `*iptr`。

5.1.1.2 指针和函数

上一章，解释了怎样说明函数的参数。或许有助于理解现在为什么使用指针作为函数的形式参数，这些参数可能需要改变。例如，给出下面的函数：

```
void swap (int *a, int *b)
```

```

{
    int temp;
    temp = *a; *a = *b; *b = temp;
}

```

这个函数**swap**说明了两个形式参数a和b，这两个参数都是**int**型的指针。这表示该函数要求传送整型变量的地址（而不是它的值），任何更改是针对该地址中的数据。下面是调用**swap**函数的**main**函数：

```

main ( )
{
    int i, j;
    i = 421;
    j = 53;
    printf ( "before: i = %4d j = %4d\n", i, j );
    swap ( &i, &j );
    printf ( "after: i = %4d j = %4d\n", i, j );
}

```

注意，这个程序确定交换了i和j的值，它等价于下列程序：

```

main ( )
{
    int i, j;
    int *a, *b, temp;
    i = 421;
    j = 53;
    printf ( "before: i = %4d j = %4d\n", i, j );
    a = &i;
    b = &j;
    temp = *a; *a = *b; *b = temp;
    printf ( "after: i = %4d j = %4d\n", i, j );
}

```

当然，这个程序产生相同的结果：函数调用 **swap (&i, &j)** 把两个实际参数（&i和&j）的值赋给两个形式参数（a和b），然后执行**swap**函数中的语句。

5.1.1.3 指针运算

如果使iptr 指针指向三个整数，而不是一个，怎样修改程序呢？下面介绍一种方法：

```

#include <alloc.h>
main ( )
{
    #define NUMINTS 3
    int *list, i;
    list = (int*) calloc ( NUMINTS, sizeof ( int ) );
    *list = 421;
}

```

```

*(list + 1) = 53;
*(list + 2) = 1806;
printf("list of addresses: ");
for (i = 0; i < NUMINTS; i++)
    printf("%4p", (list + i));
printf("\n list of values: ");
for (i = 0; i < NUMINTS; i++)
    printf("%4d", *(list + i));
printf("\n");
}

```

这个函数使用 `calloc` 代替 `malloc`，它有两个参数：分配存储空间的项数和每项的大小（字节数）。这样，`list` 指针指向了一块存储空间，6 个字节长（ $3 * 2$ ），用来存放类型为 `int` 的三个变量。

注意下面三个语句的区别：第一个语句是熟悉的，`*list = 421`，表示“把 421 存储到 `list` 指向的地址中去”。第二个语句 `*(list + 1) = 53` 较难理解，初看上去好像是“把 53 存储到 `list` 指向的下一个字节中”。如果这样，就错了。因为这个地址恰好是在前面 `int` 变量（两个字节长）的中间，这将会破坏前面存储的值。但是 C 编译程序能够正确处理这类问题，它知道 `list` 指针指向 `int` 类型的变量，所以表达式 `(list + 1)` 等价于 `(list + 1 * sizeof(int))`，这样 53 这个值就不会破坏 421 值。同样地，`(list + 2)` 等价于 `(list + 2 * sizeof(int))`，1806 这个值能正确存储，不会影响前面两个值。通常，`ptr + i` 表示存储器地址 `ptr + (i * sizeof(int))`。

输入和运行这个程序，输出结果如下：

```

list of address: 06AA 06AC 06AE
list of values : 421   53   1806

```

注意，这里的地址是两个字节长，不是一个字节，并且三个值是分开保存的。

由此可以看出：如果使用 `ptr`，一个指向 `type`（变量类型）的指针，那么表达式 `(ptr + i)` 表示存储器地址（`ptr + i * sizeof(type)`），这里 `sizeof(type)` 返回 `type` 类型变量所需要的字节数。

1.1.2 数 组

绝大多数高级语言，包括 C 语言都允许定义数组。实际上，数组是一张可索引的表，而表中的元素具有给定的数据类型。例如，可重写上面的程序如下：

```

main ( )
{
    #define NUMINTS 3
    int    list[NUMINTS], i;
    list[0] = 421;
    list[1] = 53;
    list[2] = 1806;
    printf("list of addresses: ");
}

```

```

for (i = 0; i < NUMINTS; i++)
    printf("%p", &list[i]);
printf("\n list of values:");
for (i = 0; i < NUMINTS; i++)
    printf("%4d", list[i]);
printf("\n");

```

表达式 `int list[NUMINTS]` 说明 `list` 是一个整型 (`int`) 数组, 该数组的大小实际上为 3 个整型变量。第一个变量约定为 `list[0]`, 第二个变量为 `list[1]`, 第三个变量为 `list[2]`。

数组的一般说明形式如下:

```
type name [size];
```

这里, `type` 是某种数据类型, `name` 是数组名, `size` 是数组 `name` 包含的 `type` 类型的元素个数。在该数组中的第一个元素是 `name[0]`, 而最后一个元素是 `name[size-1]`, 数组的大小用字节数表示, 即为 `size * (sizeof(type))`。

5.1.2.1 数组和指针

数组和指针之间具有密切的关系, 事实上如果运行这个程序, 其输出结果和前面的非常相似:

```

list of addresses: 163A 163C 163E
list of values    : 421  53  1806

```

可看出起始地址不同, 但这是唯一区别。事实上, 可以使用数组名作为指针, 即使是一个数组, 也可以用指针操作它。考虑下面两个重要的表达式:

```

(list+i) == &(list[i])
*(list+i) == list[i]

```

在这两种情况下, 左边的表达式等价于右边的表达式。它们两者可互相替换, 而不管 `list` 是说明为指针, 还是数组。

说明 `list` 是指针和数组的唯一区别是存储空间的分配: 如果 `list` 是一个数组, 则程序自动地为它分配所需要的存储空间; 如果说明 `list` 是一个指针, 则必须使用 `calloc` 或类似的函数调用为它明确地分配存储空间, 或者必须赋给它一些已经分配的内存地址。

5.1.2.2 数组和字符串

在前面章节中已经介绍了字符串, 以及说明字符串的两种不同方法: 即指向字符的指针和字符数组。这里, 更详细地说明这种区别。如果把一个字符串说明为一个字符 (`char`) 型数组, 则为该字符串分配存储空间。如果说明一个字符串作为一个字符 (`char`) 型指针, 则不分配存储空间, 用户必须自己使用 `malloc` 或类似函数调用为它分配存储空间, 或者赋给它一个已存在字符串的地址。

有关的例子在本章“C程序设计的常见问题”一节中给出。

5.1.2.3 多维数组

C语言也具有多维数组, 其一般说明形式如下:

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

例如, 下面的程序首先对一个两维的数组初始化, 然后完成这些数组的矩阵乘法运算。

```

main( )
{
    int a[3][4]={{5, 3, -21, 42},
                  {44, 15, 0, 6},
                  {97, 6, 81, 2}};
    int b[4][2]={{22, 7},
                  {97, -53},
                  {45, 0},
                  {72, 1}};
    int c[3][2], i, j, k;
    for (i = 0; i < 3; i++)
        {for (j = 0; j < 2; j++)
            {c[i][j] = 0;
             for (k = 0; k < 4; k++)
                 c[i][j] += a[i][k] * b[k][j];
            }
        }
    for (i = 0; i < 3; i++)
        {for (j = 0; j < 2; j++)
            printf("c[%d][%d] = %d", i, j, c[i][j]);
            printf("\n");
        }
}

```

对上面的程序应注意两件事，初始化一个二维数组的语法由嵌套的{...}表组成，每个表用逗号分开；方括号[]用来表示每个下标变量。

有些语言表示二维数组的下标使用语法[i, j]，这在C中是非法的，但是对于一维数组[j]是相同的。

多维数组按行一列的次序进行存储，这表示最后的下标变化最快。若给出数组arr[3][2]，则数组arr中的元素以下列次序进行存储：

```

arr[0][0]
arr[0][1]
arr[1][0]
arr[1][1]
arr[2][0]
arr[2][1]

```

相同的原则适用于其他任意维数的数组。

5.1.2.4 数组和函数

当把一个数组传送给函数时，情况又怎样呢？请看下面的函数，它返回一个整型数组中最小值的下标。

```

int lmin(int list[ ], int size)

```



```

int i, minindx, min;
minindx = 0;
min = list[minindx];
for (i = 1; i < size; i++)
    if (list[i] < min)
    {
        min = list[i];
        minindx = i;
    }
return (minindx);
}

```

从中可以看到C的一个最大优点：用户在编译时刻不需要知道 list[] 数组的大小，这是因为编译程序仅关心 list[] 作为数组的起始地址，而不关心该数组在哪儿结束。对函数lmin的调用形式如下面程序：

```

main ( )
{
    #define VSIZE 22
    int i, vector [VSIZE];
    for (i = 0; i < VSIZE; i++) {
        vector[i] = rand ( );
        printf ("vector[%2d] = %6d\n", i, vector[i]);
    }
    i = lmin (vector, VSIZE);
    printf ("minimum, vector[%2d] = %6d\n", i, vector[i]);
}

```

实际上传送给lmin函数的是vector数组的起始地址。这表示如果在lmin函数中对list数组进行操作，则这些操作同样作用于vector数组。例如，给出下面的程序：

```

void setrand (int list[], int size)
{int i;
  for (i = 0; i < size; i++)
    list[i] = rand ();
}

```

然后，可在main函数中调用setrand (vector, VSIZE) 函数对vector数组初始化。

怎样把多维数组传送给函数呢？是否具有与一维数组相同的灵活性？看下面的例子，修改 setrand 函数作用于二维数组，则该函数如下：

```

void setrand (int matrix[] [CSIZE], int rsize)
{int i, j;
  for (i = 0; i < rsize; i++) {
    for (j = 0; j < CSIZE; j++)

```

```
matrix[i][j] = rand( );
```

CSIZE是一个全程常量,它用来固定该数组的第二维大小。任何传送给 `setrand` 函数的数组都应该具有CSIZE的第二维数。

然而,还有另一种解决方法。假设有一个数组 `matrix[15][7]`, 并且想把它传送给 `setrand` 函数, 而且使用原来的说明形式 `setrand(int list[], int size)`, 则调用形式如下:

```
setrand(matrix, 15*7);
```

然后, 该数组 `matrix` 对于 `setrand` 函数来说, 好像是一个大小为105 (即 15×7 个元素) 的一维数组, 其操作类似于一维数组。

5.1.3 结 构

数组和指针允许建立相同数据类型项的表, 然而如果想要建立不同数据类型项的表时, 则可使用结构。

5.1.3.1 结构的定义和使用的例

结构是一种混合的数据结构, 它可包含不同的数据类型。例如, 假设想要跟踪关于星座的信息: 星的名字, 光谱的种类, 坐标等等。则可说明如下:

```
type def struct {
    char name[25];
    char class;
    short subclass;
    float decl, RA, dist;
} star;
```

这就定义了一个类型为 `star` 的结构, 以后就可用它来说明 `star` 类型的结构变量和使用该结构变量, 如下例:

```
main( )
{
    star mystar;
    strcpy(mystar.name, "Epsilon Eridani");
    mystar.class = 'k';
    mystar.subclass = 2;
    mystar.decl = 3.5167;
    mystar.RA = -9.633;
    mystar.dist = 0.303;
    /* Rest of function main( ) */
}
```

在结构变量名的后面跟一个句点(.), 表示存取结构变量的成员。其一般形式为: `varname.memname`; 它等价于相同类型的变量名 `memname`, 并且可对之完成所有相同的操作。

5.1.3.2 结构和指针

C语言中允许说明指针指向结构, 就像说明指针指向其他数据类型一样。这种能力对于建立链表和其他的动态数据结构是最基本的。事实上, 在C语言中经常使用结构指针来存取结构成员。下面使用指针改写前面的程序:

```
#include <alloc.h>

main ( )
{
    star *mystar;
    mystar = ( star* ) malloc ( sizeof ( star ) );
    strcpy ( mystar->name, "Epsilon Eridani" );
    mystar->class    = 'k';
    mystar->subclass = 2;
    mystar->decl     = 3.5167;
    mystar->RA       = -9.633;
    mystar->dist      = 0.303
    /*Rest of function main ( ) */
}
```

改写后的程序说明mystar作为类型为star的指针, 而不是作为类型为star的变量。它使用函数调用malloc为mystar分配存储空间, 以后就可使用 ptrname->memname形式存取mystar结构的成员; 符号->表示由指针指向的结构成员, 它是(*ptrname).memname的缩写形式。

5.2 switch 语句

下面的函数给出了多层if...else if...else if...嵌套结构:

```
#include <ctype.h>

do_main_menu ( short *done )
{
    char cmd;
    *done = 0;
    do{
        cmd = toupper ( getch ( ) );
        if ( cmd == 'F' ) do_file_menu ( done );
        else if ( cmd == 'R' ) run_program ( );
        else if ( cmd == 'C' ) do_compile ( );
        else if ( cmd == 'M' ) do_make ( );
        else if ( cmd == 'P' ) do_project_menu ( );
        else if ( cmd == 'O' ) do_option_menu ( );
        else if ( cmd == 'E' ) do_error_menu ( );
        else handle_others ( cmd, done );
    }
```

```
}while (! *done);
```

这在程序设计中是非常一般的，C有一种特殊的控制结构可完成相同的功能，即**switch** 语句。下面是使用**switch**语句重新改写过的相同函数：

```
#include <ctype.h>
do_main_menu ( short *done )
{
    char cmd;
    *done = 0
    do {
        cmd = toupper ( getch ( ) );
        switch ( cmd ) {
            case 'F': do_file_menu ( done ); break;
            case 'R': run_program ( ); break;
            case 'C': do_compile ( ); break;
            case 'M': do_make ( ); break;
            case 'P': do_project_menu ( ); break;
            case 'O': do_option_menu ( ); break;
            case 'E': do_error_menu ( ); break;
            default: handle_others ( cmd, done );
        }
    }while ( ! *done );
}
```

这个函数进入一个循环，该循环读入一个字符，把它转换为大写且存储在变量cmd中，然后根据cmd的值执行**switch**语句。假设在函数 do_file_menu 或 handle_others 中可得到赋值 0，这个循环一直要执行到变量 done 得到赋值0为止。

switch语句将变量cmd的值和每个case标号进行比较，如果匹配，则从该标号处开始执行，直到遇到**break**语句或**switch**语句的结尾为止。如果没有匹配的情况，则执行default标号处语句。在**switch**语句中，如果既没有匹配的情况，也没有default标号，则跳过整个**switch**语句。

switch语句所使用的值value必须是整型值或与整型兼容的值；也就是说，这些值能够很容易地转换成一个整型值。例如：char型、任何enum型、int型和所有整型的变形。不能够使用浮点型（例如float和double）、指针、字符串或其他的数据结构（但是可以使用数据结构中的整型或与整型兼容的元素）。

虽然（value）可以是任何表达式（常数、变量、函数调用或它们的任何组合），但是情况标号本身必须是常数，每个case关键字仅能够给出一个值。如果 do_main_menu 不能够利用函数 toupper 来把变量 cmd 值转换为大写，那么**switch**语句可以是下列形式：

```
switch ( cmd )
{
    case 'F':
```

```

    case 'F': do_file_menu (done);
                break;
    case 'r':
    case 'R': run_program ();
                break;
    ...

```

如果cmd是小写字母f或大写字母F，这个语句执行函数do_file_menu，其它情况类推。

记住，在执行完每一种情况后，都必须使用break语句，否则将执行后继语句（当然一直执行到遇到一个break语句）。如果在调用函数do_file_menu的后面漏掉了break语句，则输入字母F后，将导致连续调用函数do_file_menu和run_program。

下面的程序将给出这样的情况：

```

typedef enum {sun, mon, tues, wed, thur, fri, sat} days;
main ()
{
    days today;
    ...
    switch ( today )
    {
        case mon,
        case tues,
        case wed,
        case thur,
        case fri: puts ("go work! "); break;
        case sat: printf ("clean the yard and");
        case sun: puts ("relax! ");
    }
    ...
}

```

在上述switch语句中，标号值mon到fri执行相同的puts语句，后面的break语句跳出switch语句。然而，如果today等于sat，那么执行printf语句和puts("relax! ")语句；如果today等于sun，仅仅执行最后一个puts语句。

5.3 控制流命令

有一些命令可用于控制结构中或模拟其他的控制结构。return语句可提前终止函数。break和continue语句通常和循环语句配合使用，并且帮助跳出循环语句。goto语句实现程序中的跳转。条件表达式(?:)可把某些if...else语句压缩成一行。

注意，在使用这些语句之前要三思（当然除了return语句）。它们应该使用在最合适的地方，特别应尽量不要使用goto语句。

5.3.1 return语句

return语句有两种主要的用途。第一,如果一个函数返回一个值,则必须使用它传送一个值回到调用程序。例如:

```
int imax (int a, int b),
{
    if (a > b) return (a);
    else return (b);
}
```

这里,该函数使用return语句传送所接收的两个值中的最大值回到调用程序。

return语句的第二种主要用途是在某些点终止函数,而不是函数的结尾。例如,一个函数可在需要结束时检测一个条件,在if语句的里面,可使用return语句终止函数。如果函数的类型是void,则可使用不带任何值的return语句返回。下面给出lmin函数的修改版本:

```
int lmin (int list[ ], int size)
{
    int i, minindex, min;
    if (size <= 0) return (-1);
    ...
}
```

在这种情况下,如果参数size小于或等于0,那么表示在list表中无任何内容,因此调用return终止该函数。注意,返回的错误值为-1,因为-1永远不是数组的有效下标,调用程序知道发生了错误。

5.3.2 break语句

在到达循环的结尾之前,有时需要提前终止循环,这时可使用break语句。如下面的函数:

```
#define LIMIT 100
#define MAX 10
main ( )
{
    int i, j, k, score;
    int scores[LIMIT][MAX];
    for (i = 0; i < LIMIT; i++) {
        j = 0;
        while (j < MAX - 1) {
            printf ("please enter score # %d: ", j);
            scanf ("%d", &score);
            if (score < 0)
                break;
        }
    }
}
```

```

        scores[i][++j]=score;
    }
    scores[i][0] = j;
}
}

```

请注意，语句 `if (score < 0) break;` 表示如果用户为 `score` 输入一个负值，则终止 `while` 循环语句。变量 `j` 既用作 `scores` 数组的下标，也用来跟踪每一行得分的总数，计数值存储在每行的第一个元素中。可以回忆一下前面 `switch` 语句中的 `break` 语句的作用，在那种情况，它使得程序终止 `switch` 语句；而这里，它使得程序终止循环语句并且使得程序继续工作下去。`break` 语句能够用于三种循环语句（`for`，`while` 和 `do...while`），以及 `switch` 语句；然而，它不能够使用在 `if...else` 语句中，也不能直接用于主函数体中。

5.3.3 continue语句

有时，不想完全跳出循环，只想跳出循环的其余部分，并且又重新从循环的头开始执行。这种情况，可使用 `continue` 语句完成上述功能。观察下面的程序。

```

#define LIMIT 100
#define MAX 10
main ( )
{
    int i, j, k, score;
    int scores[LIMIT][MAX];
    for (i = 0; i < LIMIT; i++) {
        j = 0;
        while (i < MAX - 1) {
            printf ("please enter score # %d: ", i);
            scanf ("%d", &score);
            if (score < 0)
                continue;
            scores[i][++j] = score;
        }
        scores[i][0] = j;
    }
}

```

当执行 `continue` 语句时，程序跳过循环的剩余部分，重新从循环的测试开始执行。很明显，这个程序的结果就不同于前面的程序，当用户输入 -1 得分时，它不是终止内层循环，而是假设出现一个错误，并且转到 `while` 循环的开头重新执行。因为 `j` 没有增量，所以它以相同的得分重新要求用户输入。

5.3.4 goto语句

C中也有 `goto` 语句，其形式很简单：`goto 标号`；这里标号是一些标识符，用以标识所

给出的语句。然而，goto语句不能滥用，比较明智的使用仅对于上述三种语句，要非常小心地将它用在确实需要的地方。

5.3.5 条件表达式(?:)

有时也需要根据某些条件，选择两个表达式（和其结果值）。通常可使用if...else语句完成，例如：

```
int imin(int a, int b)
{
    if(a < b) return(a);
    else return(b);
}
```

然而，C中有一种特殊的结构可完成这种类型的选择，其一般形式为：

```
expr1 ? expr2 : expr3
```

其意义为：如果expr1为真，那么执行expr2并且整个表达式的值就是它的值；否则执行expr3并且整个表达式的值就是它的值。使用这种结构，能够重写imin函数如下：

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

要做得更好一点，则可重写imin函数作为一个一行的宏：

```
#define imin(a, b)((a < b) ? a : b)
```

至此，不管何时当程序遇到表达式imin(e1,e2)时，则使用((e1<e2)? e1:e2)替换它并且继续编译。实际上，这是一种最通用的方法；因为a和b不再限制为int型，它们能够是小于(<)关系所允许的任何数据类型。

5.4 C程序设计风格

当前的一些趋势是，在C中嵌入某些技术，以使得C更容易使用。虽然很多的这些趋势与传统的C程序设计方法相抵触，但根据ANSI C标准委员会所定义的语言扩充，绝大多数已成为可能。本节将介绍过去的情况和现在新的标准以帮助读者写出更好的C程序。

当然，Turbo C既支持传统的程序设计风格也支持现代的程序设计风格。

5.4.1 使用函数原型和全函数定义

按传统的C程序设计风格，说明函数仅需指出函数名和返回值的类型。例如，定义swap函数如下：

```
int swap();
```

它没有给出参数信息，即参数的个数和参数的类型。这个函数的传统定义如下：

```
int swap(a, b)
int *a, *b;
{
    /* 函数体 */
}
```


这种风格使得错误检查困难，也难以理解和跟踪错误，应尽量避免它。

现代风格包括使用函数原型（对于函数说明）和参数表（对于函数定义）。下面使用函数原型重新说明swap函数：

```
int swap(int *a, int *b);
```

这样，当程序编译时，它具有所有的信息，便于对swap函数调用作完全的错误检查。当定义这个函数时，可使用相同的格式：

```
int swap(int *a, int *b)
{
    /* 函数体 */
}
```

现代风格增加了错误检查，即使未使用函数原型来实现也一样。如果确实使用了函数原型，编译程序将保证说明和定义一致。

5.4.2 使用enum定义

现在能够使用关键字enum说明枚举数据类型，例如，以前使用#define指令定义的如下值表：

```
#define sun    0
#define mon    1
#define tues   2
#define wed    3
#define thur   4
#define fri    5
#define sat    6
```

可写成：

```
typedef enum{sun, mon, tues, wed, thur, fri, sat} days;
```

其作用相同于传统方法，依次建立sun = 0和sat = 6。然而，现代方法隐藏了更多的信息并且比#define指令所定义的长表难理解。这样定义后，变量的类型可以说明成days类型。

5.4.3 使用typedef

在C的传统风格中，用户定义的数据类型很少取名，除了结构和联合例外。即使使用结构和联合，也必须使用关键字struct或union放在任何说明的前面。

在C的现代风格中，当使用typedef指令时，隐藏了另一层信息。这允许对所给出的数据类型（包括struct和enum）命名，然后可说明那种类型的变量。下面是一些使用变量说明的样本类型定义：

```
typedef int *intptr;
typedef char namestr[30];
typedef enum{male, female, unknown} sex;
typedef struct{
    namestr last, first;
```

```

    char    ssn[9];
    sex     gender;
    short   age;
    float   gpa;
}student;
typedef student class[100];
class hist104, ps102;
student valedictorian;
intptr iptr;

```

使用typedef使得程序更容易阅读；它也允许改变某个位置（定义类型的地方），而使得这个改变传播到整个程序。

5.4.4 说明void函数

在C的原始定义中，每个函数都返回某一些类型的值；如果不说明类型，则假设函数为int类型。用类似的方式，返回“普通”（无类型）指针的函数通常被说明成返回char型的指针，仅仅因为它们必须返回一些东西。

现在，有一种标准类型void，它作为一种“NULL”类型。不明确返回一个值的函数应该说明成void类型。注意，很多动态分配存储器的函数（例如malloc）被说明为void*类型，这表示它们返回一个无类型的指针，然后（在Turbo C中）能够赋给任意类型的指针而没有强制类型（对于保持可移植性，还是应该强制类型）。

5.4.5 扩充的使用

C语言中的某些扩充有助于增加程序的可读性，替换一些过时的成份，可使得程序员不断进步。下面是一些扩充的简单介绍。

5.4.5.1 字符串文字

在C语言中，为了在程序中具有大字符串的文字，必须使用连续字符或一些连结运算。这在现代风格的C中，能够很容易扩展一个跨越数行的大字符串文字，如下例。

```

main ( )
{
    char *msg
    msg = "Four score and seven year ago, our fathers"
          "brought forth upon\n this continent a new"
          "nation, dedicated to the ideal that all"
          "men\n are created equal";
    printf ("%s", msg);
}

```

5.4.5.2 十六进制字符常数

在传统C语言中，表示特殊ASCII代码的转义序列都使用八进制（基数8）实现。这是因为早期开发C的机器上，二进制数通常用八进制形式表示。

现在，绝大多数计算机使用十六进制（基数16）来表示二进制数。由于这个原因，现代

C允许使用十六进制表示说明字符常数,一般形式为'\XDD',这里DD表示一个或两个十六进制数字(0..9, A..F)。这些转义序列能够直接赋给char型变量,或者它们能够被嵌在字符串中,例如:ch = '\X20'。

5.4.5.3 signed类型

传统C语言假设所有基于整数的类型都带符号,这样就增加了类型修饰符unsigned(无符号),以便于指定其他类型。根据标准约定,char类型的变量被认为是signed(带符号),这意味着值的基本范围是-128到127。

然而,在今天的微型计算机上,char类型经常被认为是unsigned(无符号),并且Turbo C有一个编译程序开关允许设置这种标准约定。在这种情况下,仍然能够说明一个signed char类型变量。在现代C中,signed被看作为一个有效的修饰符。

5.5 C程序设计中的常见问题

很多程序员在刚开始使用C语言编程时,会遇到很多一般的错误。下面列出这些错误,并且给出怎样避免这些错误的建议。

5.5.1 使用C字符串的路径名

众所周知在MS-DOS中的反斜线(\)表示一个目录名,然而在C中,反斜线(\)在字符串中表示转义字符。如果使用C字符串表示路径名,这种冲突就会引起问题。例如,如果输入:

```
"c:\new\tools.c"
```

希望得到C驱动器上的NEW目录中的TOOLS.C文件,实际上不能。因为在C语言字符串中,\n表示新行(LF)字符的转义序列,\t表示制表(tab)字符的转义序列。其结果是文件名嵌入了新行和制表字符,DOS不能识别该字符串作为合适的文件名,文件名中不可以有新行或制表字符。正确的字符串应该是:

```
"c:\\new\\tools.c"
```

5.5.2 指针的使用和误用

对于C程序设计的初学者来说,下面这些问题可能是最难掌握的。什么时候使用指针,什么时候不能使用指针?什么时候使用间接操作符(&)?什么时候使用地址操作符(&)?在程序运行时,怎样避免破坏操作系统?

5.5.2.1 使用没有初始化的指针

将一个值赋给一个没有初始化的指针是非常危险的,例如:

```
main()
{
    int *iptr;
    *iptr = 421;
    printf(" *iptr = %d\n", *iptr);
}
```

这会引起什么样的问题呢?危险主要是无法控制它。在上面的例子中,指针iptr可能有一个

随机地址, 即值421被存储的地方。这个程序非常小, 可以容易地修改它, 但在一个大程序中, 危险性就大了, 因为可能有其他的信息存储在 `iptr` 碰巧包含的地址中。如果使用小存储模式, 这里代码和数据占据相同的空间, 就会有破坏机器代码的危险。

5.5.2.2 字符串

回忆一下, 字符串可以说明为 `char` 型的指针或 `char` 型的数组, 这两者基本是相同的, 但有一个非常重要的区别: 如果使用 `char` 型的指针, 则对于字符串不分配存储空间, 如果使用 `char` 型数组, 则分配存储空间, 并且数组变量保存该存储空间的地址。

错误地理解这种区别, 将会引起两种类型错误。看下面的程序:

```
main ( )
{
    char *name;
    char msg[10];
    printf ( "What is your name? " );
    scanf ( "%s", name );
    msg = "Hello, ";
    printf ( "%s %s", msg, name );
}
```

乍看上去, 这似乎很好, 虽有点笨拙, 但仍然允许。但是这会引入两种错误。第一种错误是语句 `scanf ("%s", name)` 引起的, 这个语句本身是合法和正确的, 因为 `name` 是一个 `char` 型的指针, 不需要在它的前面使用地址操作符 (`&`)。然而, 程序没有为 `name` 分配任何存储空间, 输入的姓名将会存储在 `name` 指针碰巧具有的随机地址中, 这时会出现警告信息: (possible use of 'name' before definition), 但不是错误。

第二种引起错误的问题是语句 `msg = "Hello, "`, 编译程序认为用户想试图改变 `msg` 为常数字符串 "Hello, " 的地址。不能够这样做, 因为数组名是不能修改的常数 (就像 `7` 是常数一样, 不能写成 `7 = i`)。编译程序将给出一个错误信息:

"Lvalue required"

怎样纠正这些错误呢? 最简单的方法是将 `name` 和 `msg` 的说明互换一下, 即

```
main ( )
{
    char name[10];
    char *msg;
    printf ( "What is your name? " );
    scanf ( "%s", name );
    msg = "Hello, ";
    printf ( "%s %s", msg, name );
}
```

这样就没有问题了。变量 `name` 具有保存所输入姓名的存储空间, 而 `msg` 允许赋给它常数字符串 "Hello, " 的地址。然而, 如果不想改变前面的说明形式, 那么需要改变程序如下:

```
#include <alloc.h>
main ( )
{
```

```

char *name;
char msg[10];
name = (char*) malloc(10);
printf("What is your name? ");
scanf("%s", name);
strcpy(msg, "Hello, ");
printf("%s %s", msg, name);

```

函数调用**malloc**分配10个字节的存储器空间,并且将该存储器的地址赋给指针name,这样防止了第一种错误。函数调用**strcpy**完成字符串复制的功能,它将常数字符串"Hello,"复制到数组msg中。

5.5.3 赋值号(=)和等号(==)的混淆

在Pascal和BASIC语言中,对于相等的比较使用表达式if (a = b) 完成。在C中,也有同等的结构,但它表示不同的含义。如下面的程序段:

```

if (a = b) puts("Equal");
else puts("Not equal");

```

对于Pascal或BASIC程序员来说,可能认为如果a和b的值相等,则输出"Equal"信息;否则输出"Not equal"。但是在C中不是这个意思,表达式a = b表示"把b的值赋给a",并且整个表达式的值就是b的值。这样,上面的程序段将b的值赋给a,如果b是非零值,则输出"Equal";否则,输出"Not equal"。如果真正需要比较操作,则程序段的实现如下:

```

if (a == b) puts("Equal");
else puts("Not equal");

```

5.5.4 switch语句中忘记break语句

break语句使用在switch语句中,以终止每一种情况。记住,不要忘记在每一种情况的结尾放上break语句,这样后面的情况才能正确执行。

5.5.5 数组下标

不要忘记数组下标从[0]开始,而不是从[1]开始。写程序的一般错误如下:

```

main( )
{
    int list[100], i;
    for (i = 1; i <= 100; i++)
        list[i] = i * i;
}

```

这个程序留下了list中的第一个位置,即list[0]没有使用,并且它把一个值存储在list的一个不存在的位置list[100]。在程序运行过程中,可能还会覆盖其他的数据。正确的程序应该是:

```

main( )

```

```

    {
        int list[100], i
        for (i = 0; i < 100; i++)
            list[i] = i * i;
    }

```

5.5.6 忘记传送地址

看下面的程序，并且指出错在哪里：

```

main( )
{
    int a, b, sum;
    printf("Enter two values: ");
    scanf("%d %d", a, b);
    sum = a + b;
    printf("The sum is %d\n", sum);
}

```

错误在语句`scanf("%d %d", a, b)`中，注意`scanf`函数需要传送地址，而不是值。形式参数是指针的任何函数都具有的相同的问题。上面的程序编译和运行时，`scanf`函数将接收随机值在`a`和`b`变量中，并且将它们作为地址以存储所输入的值。

正确的语句应该是`scanf("%d %d", &a, &b)`；使用这种方法，`a`和`b`变量的地址能传送给`scanf`函数，并且所输入的值能正确地存储在这些变量中。这种相同的问题也会发生在用户自己的函数中；例如前面所定义的函数`swap`使用指针，如果调用形式如下：

```

main( )
{
    int i, j;
    i = 421;
    j = 53;
    printf("before, i = %4d j = %4d\n", i, j);
    swap(i, j);
    printf("after, i = %4d j = %4d\n", i, j);
}

```

这将发生什么情况呢？在函数`swap`调用之前和之后，变量`i`和`j`都具有相同的值，即它们的值没有改变。然而，在数据地址 421 和 53 之处的值被交换，这将引起一些故障和难以跟踪的问题。

怎样避免这种问题呢？使用函数原型和全函数定义可避免这种问题。实际上，如果`swap`函数像前面所定义的那样，则在`main`的版本中将会得到编译程序错误。然而，如果使用下面的方式定义`swap`函数，则程序将正确编译。

```

void swap(a, b)
    int *a, *b;
{

```

```
...  
}
```

将 a 和 b 的定义移出圆括号，将不能够进行错误检查而继续执行，这是不使用函数定义的传统风格的最好理由。

至此，我们已经对 Turbo C 作了简要的介绍，并且有了良好的开始。为了全面了解 Turbo C 语言以及开发满足应用所需要的 C 语言程序，还应该继续阅读本书的以下章节。

第六章 Turbo Pascal与Turbo C 的异同、转换和连接

在进一步学习有关内容之前,可对第三章和第五章作一简要的回顾。掌握C是怎样实现程序设计的基本成分的。在本章中还将涉及一些相同的问题,但不再重复在第三章和第五章中的许多细节。

总的来说,Pascal是一种相当严谨的结构化语言,而C则相当灵活,故使用C语言要相当小心,否则很容易出错,真所谓水能载舟,亦能覆舟。同C相比,Pascal比较多地考虑到程序设计人员的使用。因此,对学习程序设计基础知识的人来说Pascal是一种比较合适的语言。

Turbo C和Turbo Pascal都更为靠近C—Pascal语言谱系的中心:Turbo C在C的基础上增加了一些结构,而Turbo Pascal在Pascal基础上增加了灵活性。

本章不准备对C及其许多良好特征作详细的讨论,主要讨论两种语言的异同、转换和连接,其目的在于帮助对Turbo Pascal语言熟悉的程序设计人员,充分了解Turbo C,使其能很快用Turbo C来编写程序。随着时间的推移,通过不断的练习,将会熟能生巧,逐步加深了解,编写出较大的实用程序。

6.1 Turbo Pascal与Turbo C的比较

6.1.1 程序结构

回顾一下Turbo Pascal程序的结构,它呈如下形式:

```
program 程序名;  
  <说明:  
  const  
  type  
  var  
  procedure和function>  
  begin    {程序的主体}  
  <语句>  
  end.     {程序结束}
```

程序从主程序体开始执行。如果它要调用另外的过程和函数,那么相应的过程和函数也被执行。所有的标识符——常量名、类型名、变量名、过程名和函数名——都必须在被使用之前说明。说明的先后次序无关紧要。过程和函数的结构几乎同程序一样。

C程序的结构稍微灵活些,即为:

```
<预处理命令>  
<类型定义>  
<函数原型>
```


<变量>

<函数>

其中,函数的结构如下:

<类型> 函数名(<参数说明>)

{<局部说明>

<语句>

}

在所说明的所有函数中,必须有一个名为main的函数,它是主程序体。也就是说当 Turbo C程序开始执行时,首先调用main,而main能依次调用其他的函数。C程序完全由函数组成。然而,有些函数的类型是void,即没有返回值。这些函数有些类似于Pascal过程。另外,与Pascal不同的是对函数的返回值可以不用。

[例6.1] Turbo Pascal 与 Turbo C 程序结构比较

下面两个程序,一个是用 Turbo Pascal 写的,另一个是用 Turbo C 写的,用以说明两种语言在程序结构方面的异同。

Turbo Pascal	Turbo C
<pre>Program Myprog; var I,J,K : Integer function Max(A,B : Integer) : Integer; begin if A>B then Max := A else Max := B end; { End of func Max } procedure Swap(var A,B : Integer); var Temp : Integer; begin Temp := A; A := B; B := Temp end; { End of proc Swap } begin { Main body of Myprog } I := 10; J := 15; K := Max(I,J); Swap(I,K); Write('I = ',I:2,'J = ',J:2); Writeln('K = ',K:2) end. { End of program Myprog }</pre>	<pre>int i,j,k; int max(int a, int b) { if (a>b) return(a); else return(b); } /*End of max()*/ void swap(int *a,int *b) { int temp; temp = *a; *a = *b; *b = temp; } /*End of swap()*/ main() { i = 10; j = 15; k = max(i,j); swap(&i, &k); printf("i = %2d j = %2d",i,j); printf("k = %2d\n",k); } /*End of main*/</pre>

我们也可在 main 内部对 i, j, k 作出说明,而不把它们作为全程变量。在许多情况下,这是比较好的程序设计风格。因为这样减少了在函数内部直接修改全程量的机会和诱惑,同时避免这些变量在整个程序运行过程中均存在。

右侧的 C 程序现在也许看起来稀奇古怪,但学完这一章后就会对其了如指掌。以后,读者自己会写出更奇妙的程序。

6.1.2 程序设计成分

在第四章中,已经讨论了程序设计中七种基本的成分——输出、数据类型、运算符、输入、条件执行、循环和子程序。现从这些方面比较一下 Pascal 和 C 的相似之处和不同之处。

6.1.2.1 输出

Turbo Pascal 中主要的输出命令是 write 和 writeln,而 Turbo C 根据输出的内容没有许多不同的命令。最常使用的命令,同时额外开销也最大的是 printf,其格式为:

```
printf(<格式串>, <项>, <项>, ...);
```

这里<格式串>是字符串直接量或字符串变量(C使用双引号),<项>是任选变量或表达式等。各输出项需和格式串的内容匹配。具体细节见第四章。在C中要换行(等价于writeln)只需在格式串的末端插入转义序列\n(换行符)。

下面有几个 Turbo Pascal 语句及其等价(或大致等价)的 C 语句。

[例6.2] Turbo Pascal 与 Turbo C 输出成份比较。

Turbo Pascal	Turbo C
var	
A,B,C: Integer;	int a,b,c;
Amt : Real;	float amt;
Name : string [20];	char name [21]; (或*name)
Ans : Char;	char ans;
Writeln('Hello, world.');	printf("Hello, world.\n");
Write('What's your name?');	printf("What's your name?");
Writeln('Hello,' said John');	printf("\nHello,\n said John\n");
Writeln(A, ' + ', B, ' = ', C);	printf("%d + %d = %d\n",a,b,c);
Writeln('You owe us \$', Amt:6:2);	printf("You owe us \$%6.2f\n",amt);
Writeln('Your name is', Name, '?');	printf("Your name is %s?\n",name)
Writeln('The answer is', Ans);	printf("The answer is %c\n",ans);
Write('A = ', A:4);	printf("a = %4d",a);
Writeln('A * A = ', (A * A):6);	printf("a * a = %6d\n",a*a);

另外两个值得注意的 C 输出子程序是 puts 和 putchar。puts 接收一字符串为参数并输出之,自动添加一换行符。因此以下的语句是等价的:

writeln(Name);	puts(Name);
writeln('Hi, there! '); writeln;	puts("Hi, there!\n");

putchar就更加简单了,它只输出一个字符。比如

write(ch);	putchar(ch);
------------	--------------

6.1.2.2 数据类型

大多数 Turbo Pascal 的数据类型在 Turbo C 中均有等价数据类型。实际上 C 的数据类型更多，它有不同的长度的整型值和浮点数值，还可以用 signed 和 unsigned 来修饰。下表给出了 Pascal 和 C 数据类型间的大致等价关系：

Turbo Pascal	Turbo C
char (1 字节) chr(0—255)	char (1 字节) -128—127
byte (1 字节) 0—255	unsigned char (1 字节) 0—255
integer (2 字节) -32768—32767	short (2 字节) -32768—32767
	int (2 字节) -32768—32767
	unsigned int (2 字节) 0—65535
	long (4 字节) -2^{31} — $2^{31}-1$
	unsigned long (4 字节) $0-(2^{32}-1)$
real (6 字节) $1E-38-1E+38$	float (4 字节) $\pm 3.4E\pm 308$
	double (8 字节) $\pm 1.7E\pm 308$
boolean(1 字节) false, true	0 = false, 非零 = true

需要注意的是 C 中没有布尔类型。当表达式的结果用作布尔值时，就用 0 作为布尔值 false，其他任意的非 0 值作为布尔值 true。

除表中所列出的数据类型外，Turbo C 还有枚举类型，但不同于 Pascal 的枚举类型，它只是预定意义的整型常量。因此，它与所有整型完全相容。

Turbo Pascal	Turbo C
type	
Days = (Sun, Mon, Tues, Wed, Thurs, Fri, Sat);	enum days = { Sun, Mon, Tues, Wed, Thurs, Fri, Sat };
var	
Today : Days;	enum days today;

6.1.2.3 运算符

Turbo C 包含了 Turbo Pascal 中所有的运算符，还有一些是 Turbo Pascal 中没有的。

这两种语言基本差别之一是对赋值的处理。在 Pascal 中，赋值(:=)是个语句。在 C 中赋值(=)是个运算符，可以用于表达式中。

表 6.1 把 Turbo Pascal 和 Turbo C 中的运算符逐个进行了比较。其中运算符按优先级先后列出，放在一组中的运算符优先级相同。

表6.1 Turbo Pascal 和 Turbo C 的运算符

	Turbo Pascal	Turbo C
单目减	$A := -B;$	$a = -b;$
单目加	$A := +B;$	$a = +b;$
逻辑非	$\text{not Flag};$	$! \text{flag};$
按位取补	$A := \text{not } B;$	$a = \sim b;$
求地址	$A := \text{Addr}(B);$	$a = \&b;$
指引元引用	$A := \text{Intptr}^+;$	$a = * \text{intptr};$
长 度	$A := \text{sizeof}(B);$	$a = \text{sizeof}(b);$
增 一	$A := \text{succ}(A);$	$a++$ 或 $++a;$
减 一	$A := \text{pred}(A);$	$a--$ 或 $--a;$
乘 法	$A := B * C;$	$a = b * c;$
整数除法	$A := B \text{ div } C;$	$a = b / c;$
浮点数除法	$X := B / C;$	$x = b / c;$
取 模	$A := B \bmod C;$	$a = b \% c;$
加 法	$A := B + C;$	$a = b + c;$
减 法	$A := B - C;$	$a = b - c;$
右 移	$A := B \text{ shr } C;$	$a = b >> c;$
左 移	$A := B \text{ shl } C;$	$a = b << c;$
大 于	$A > B$	$a > b$
大于等于	$A \geq B$	$a \geq b$
小 于	$A < B$	$a < b$
小于等于	$A \leq B$	$a \leq b$
等 于	$A = B$	$a = b$
不 等	$A <> B$	$a \neq b$
按位与	$A := B \text{ and } C;$	$a = b \& c;$
按位或	$A := B \text{ or } C;$	$a = b c;$
按位异或	$A := B \text{ xor } C;$	$a = b \wedge c;$
逻辑与	Flag1 and Flag2	$\text{flag1} \&\& \text{flag2}$
逻辑或	Flag1 or Flag2	$\text{flag1} \text{flag2}$
赋 值	$A := B;$	$a = b;$
	$A := A \langle \text{op} \rangle B;$	$a \langle \text{op} \rangle = b;$

Turbo C 和 Turbo Pascal 有一些很重要的不同之处。

第一，减一(--)和增一(++)运算符可以放在变量名的前面或后面，如果是放在变量前面，那么在计算表达式其余部分之前就对变量先做 减一 或 增一 操作，如果是放在变量后

面, 那么先计算表达式其余部分, 然后再对变量做减一或增一操作。

第二, C 中的逻辑运算符(&&, ||)是“短路”运算符。也就是说如果第一项得到的结果能确定表达式的值, 那么第二项就不再计算了。因此, 同 Pascal 不一样, C 允许程序员安全地编写:

```
while (i <= limit && list[i] != 0)...
```

这里 limit 是数组 list 中允许的最大下标。

如果第一项($i \leq \text{limit}$)是 false, 那么 C 就知道整个表达式的值就是 false, 就不再计算第二项($\text{list}[i] \neq 0$)了。第二项会引起下标越界错。

第三, C 允许把如下形式的表达式

$$A = A \langle \text{op} \rangle B$$

替换为,

$$A \langle \text{op} \rangle = B$$

这里 $\langle \text{op} \rangle$ 可以是任意的双目运算符(除 && 和 ||)。

因此, 可以把 $A = A * B$ 写作 $A * = B$, 等等。

6.1.2.4 输入

在 Turbo Pascal 中, 只有一条基本的输入命令 Readln, 还有一些变体 (Readln(), Read(f,), 等等)。Turbo C 中, 用于键盘输入的主要函数是 scanf, 它有如下格式:

```
scanf(<格式串>, <地址1>, <地址2>, ...);
```

这里 $\langle \text{格式串} \rangle$ 是带有格式指示符的字符串(同 printf), 而每个 $\langle \text{地址} \rangle$ 是用来存放 scanf 输入数据的地址。这就意味着要经常用求地址运算符(&)。还有一些命令也要经常用到, 如 gets, 读入一字符串直到按下 Enter 键; getch, 直接从键盘上读一字符, 不回显。

以下是一些 Pascal 输入命令及其对应 C 命令:

Turbo Pascal	Turbo C
Readln(A, B);	scanf("%d %d", &a, &b);
Readln(Name);	scanf("%s", name);
	/*或 gets(name); */
Readln(X, A);	scanf("%f %d", &x, &a);
Readln(Ch);	scanf("%c", &ch);
Read(Kbd, Ch);	ch = getch();

注意, 这两种读字符串的方式 (scanf 和 gets) 间有一重要差别, 即 scanf 读入所有的字符直到空白(空格, 制表符, 换行)出现, 而 gets 则读入所有的字符(包括空格和制表符)直到按下 Enter 键。

6.1.2.5 复合语句

Pascal 和 C 中都有复合语句的概念(它是一组语句, 能出现在任何单条语句所能出现的地方)。Pascal 中, 复合语句呈如下的形式:

```
begin <语句>; <语句>; ... <语句> end;
```

C 取如下的形式:

{<语句>, <语句>, ..., <语句>, }

虽然两种形式很相似, 但有以下两个重要差别:

(1) 在 Pascal 中不要在最后一个 <语句> 后加上分号, 但 C 却要;

(2) 在 C 中不需在右花括号后加上分号, 而在 Pascal 中却要。

6.1.2.6 条件执行

Pascal 和 C 中都有两种条件执行语句: 一种是 if-then-else 语句, 另一种是 case 语句。两种语言中的 if-then-else 语句很类似:

if	<布尔表达式>	if (<表达式>)
then	<语句>	<语句>;
else	<语句>	else <语句>;

在 Pascal 和 C 中, else 子句是任选的。<语句> 部分可以用前述复合语句代替, 但仍有以下一些差别:

C 中 <表达式> 不必是布尔型, 只要能区分成 0 或非 0 即可。这里 0 被看作是 false, 非 0 值就被看作是 true。

C 中 <表达式> 必须用括号括起来。

C 中没有 then。

C 中每个语句后面都要加上分号, 除非是用复合语句。

[例6.3] 下面是用两种语言描述的同含义的条件语句:

Turbo Pascal	Turbo C
<pre>if B = 0 then Writeln('C is undefined') else begin C := A div B; Writeln('C = ', C) end;</pre>	<pre>if (B == 0) puts("c is undefined"); else { c = a/b; printf("c = %d\n", c); }</pre>
<pre>C := A * B; if C <> 0 then C := C + B; else C := A</pre>	<pre>if ((c = a * b) != 0) c += b; else c = a;</pre>

Pascal 和 C 中都提供了 case 语句(在 C 中称为 switch 语句), 但有一些差别, 下面先列出该语句在 Pascal 和 C 中的一般格式:

```
case < 表达式 > of
```

```
< 表 > : < 语句 >;
```

```
< 表 > : < 语句 >;
```

```
...
```

```
< 表 > : < 语句 >;
```

```
else < 语句列 >
```

```
end;
```

```
switch(< 表达式 >){
```

```
case < 项 > : < 语句列 >
```

```
case < 项 > : < 语句列 >
```

```
...
```

```
case < 项 > : < 语句列 >
```

```
default : < 语句列 >
```

```
}
```

除了形式的差别外，还存在着一些非常重要的区别：

(1) Pascal 中，< 表 > 可以包含若干个值，在 Turbo Pascal 中还可以包括类似于 A..Z 的值域。在 C 中每个 < 项 > 只能有一个值。两种语言对值都加以了限制，即只能是整型、字符型、枚举型这些有序类型的常量。

(2) 也是很重要的一点，就是在 Pascal 中，< 语句 > 是单条语句或复合语句。一旦某个语句被执行了，那么 case 语句中其余的语句就跳过不再执行了。在 C 中，< 语句列 > 包含零个或多个语句，每个语句都以分号结束。然而一旦某个 < 语句列 > 执行完了，控制并不转到 switch 语句的结尾处，而继续执行以后的语句，直到遇到了 break 语句。那时，也只有到了那时，switch 语句的剩余部分才略过不执行。每个 case < 项 > 可以看作是一个标号，根据 switch(< 表达式 >) 语句决定跳到何处。这种看法对加深理解是很有帮助的，下面有几个例子。

[例6.4] 用 Turbo Pascal 和 Turbo C 描述的两个 case 结构：

Turbo Pascal

Turbo C

```
case Ch of
```

```
'C' : DoCompile;
```

```
'R' : begin
```

```
if not Compiled
```

```
then DoCompile
```

```
Runprogram;
```

```
end;
```

```
'S' : SaveFile;
```

```
'E' : EditFile;
```

```
'Q' : begin
```

```
if not Saved
```

```
then SaveFile
```

```
end;
```

```
switch(ch)
```

```
{
```

```
case 'C' : DoCompile(); break;
```

```
case 'R' :
```

```
if (!compiled)
```

```
DoCompile();
```

```
Runprogram();
```

```
break;
```

```
case 'S' : SaveFile(); break;
```

```
case 'E' : EditFile(); break;
```

```
case 'Q' :
```

```
if (!saved)
```

```
SaveFile();
```

```
break;
```

续表

Turbo Pascal

Turbo C

<pre> end, case Today of Mon..Fri: WriteLn('go work! '); Sat,Sun : begin if Today = Sat then begin Write('clean the yard'); Write('and') end; WriteLn('relax! ') end end, </pre>	<pre> } switch (today) { case Mon : case Tue : case Wed : case Thur: case Fri : puts("go work! "); break; case Sat : printf("%s", "clean the yard and"); case Sun : puts("relax! "); } </pre>
---	---

注意第二个例子，switch 语句中的 case<项>标明了要处理的各种情况。在从 Mon 到 Thur 的情况下，<语句>部分为空，控制就继续往下找，直到找到了用 case Fri 标明的语句，然后 break 语句就使控制跳过下面的语句到达 switch 语句的末端。然而这个程序在处理 Sat 和 Sun 时也利用了这一特性。当执行完 case Sat 后的语句后，控制转到下一语句 puts。

6.1.2.7 循环

同 Pascal 一样，C 有三类循环：while，do..while 和 for。与此相对应的三种 Pascal 循环是：while、repeat..until 和 for。现在依次来介绍它们。

(1) while 循环

在这三种循环中，两种语言中的 while 循环是最相似的。这里分别列出其格式：

while <布尔表达式> do	while (<表达式>)
<语句>;	<语句>;

在两种语言中，都可以在循环中<语句>的位置使用复合语句以加入多个语句。唯一的区别是 C 所要求的<表达式>比较灵活。比较下列两个循环：

Read (Kbd, ch);	while ((ch = getch()) != 'q')
while ch <> 'q' do	
begin	putchar(ch);
write(ch);	
Read (Kbd, ch)	
end	

(2) do..while 循环

do..while 循环与 Pascal 中的 repeat..until 循环很类似，这里列出其格式：

repeat	do
<语句列>	<语句>;
until <布尔表达式>;	while(<表达式>);

但这两种循环之间有两个很大的不同, 这就是:

① do..while 循环是在<表达式>值为 true 时循环, 而 repeat..until 循环是不断执行, 直到<布尔表达式>值为 true。

② repeat..until 语句并不要求用复合语句来代替多个语句, 而 do..while 中必须这样。

[例6.5] 这里对两种语言各提供一个例子:

Turbo Pascal	Turbo C
<pre>repeat Write('Enter a value: '); Readln(A) until (Low <= A) and (A <= High);</pre>	<pre>do { printf("Enter a value: "); scanf("%d", &a); } while (a < low a > high);</pre>

注意, C 和 Pascal 还有一个很大的差别就是: 在 C 中, 关系运算符(>、<, 等等)的优先级比逻辑运算符(&&, ||)要高。这就是说并不需要像在 Pascal 中那样把每个关系运算表达式都用括号括起来。

(3) for 循环

在 Pascal 和 C 的三种循环中, for 循环的差别最大。Pascal 中, for 循环是相当固定和不灵活的。而在 C 中, for 循环太灵活了, 所允许的某些结构容易使其失去 for 循环的特征。

下面是两种语言的 for 循环的格式:

<pre>for<循环变量>= <初值> to <终值> do <语句>;</pre>	<pre>for(<表达式1>; <表达式2>; <表达式3>) <语句>;</pre>
---	--

在 C 中(实际上在 Pascal 中也是这样), for 语句只是 while 语句的一个特例。上述 for 循环格式与下列语句是等价的:

```
<表达式1>;
while(<表达式2>)
{ <语句>
  <表达式3>;
}
```

<表达式1>用作初始值, <表达式2>用作检测循环结束条件, <表达式3>增量或用其他方法修改循环变量。

[例6.6] 下面的几个例子中, 有些用了 Pascal 的 while 循环:

Turbo Pascal	Turbo C
<pre>for I := 1 to 10 do begin Write('I = ', I:2); Write(' I * I = ', (I * I):4);</pre>	<pre>for (i = 1; i <= 10; i++) { printf("i = %2d ", i); printf("i * i = %4d", i * i);</pre>

续表	Turbo Pascal	Turbo C
	Writeln('I**3=',(I*I*I):6)	printf("i**3=%6d\n",i*i*i);
	end,	}
	I:=17; K:=I;	
	while (I>-450) do begin	for (i=17,k=i; i>-450;
	K:=K+I;	k+=i,i-=15)
	Writeln('K=',K,'I=',I);	printf("k=%d i=%d\n",k,i);
	I:=I-15	
	end,	
	X:=D/2.0;	for (x=d/2; abs(x*x-d)>0.01;
	while (Abs(X*X-D)>0.01) do	x=(x+d/x)/2)
	X:=(X+D/X)/2.0;	; /*空语句*/

6.1.2.8 子程序

Pascal 和 C 都有子程序。Pascal 中有过程和函数，而 C 中只有函数。但函数可以说明成 void 类型，这就是说可以根本没有返回值。也可以不使用函数的返回值。

下面是两种语言函数的格式：

Turbo Pascal	Turbo C
function 函数名(< 参数说明 >) : < 类型 >; < 局部说明 > begin < 语句列 > end,	< 类型 > 函数名 (< 参数说明 >) { < 局部说明 > < 语句列 > }

在 Pascal 中，< 参数说明 > 的形式为 < 参数名 > : < 类型 >；在 C 中的形式是 < 类型 > < 参数名 >，。

[例6.7] 现用例子来说明这两种语言子程序定义间的另外一些重要差别。

Turbo Pascal	Turbo C
function Max(A,B: Integer): Integer; begin if A>B then Max:=A else Max:=B end,	int max(int a,int b) { if (a>b) return(a); else return(b); }

注意 C 中的返回语句是用以在函数中返回一个值，而在 Pascal 中是把这个值赋给函数名。

Turbo Pascal	Turbo C
<pre>procedure Swap(var X,Y: Real); var Temp :Real; begin Temp:= X; X:= Y; Y:= Temp end;</pre>	<pre>void swap(float *x, float *y) { float temp; temp = *a; *a = *b *b = temp; }</pre>

在 Pascal 中，参数有两种类型：var (传送地址)和 value (送传值)。在 C 中，只能使用传传值的参数。如果要用传送地址的参数，那么就必须传送地址，并定义形式参数为指针。swap 就是这样做的。下面为调用这两个子程序的例子。

Turbo Pascal	Turbo C
<pre>Q := 7.5; R := 9.2; Writeln('Q = ',Q:5:1,'R = ',R:5:1); Swap(Q,R); Writeln('Q = ',Q:5:1,'R = ',R:5:1);</pre>	<pre>q = 7.5; r = 9.2; printf("q = %5.1f r = %5.1f\n",q,r); swap(&q, &r); printf("q = %5.1f r = %5.1f\n",q,r);</pre>

请注意在把 q 和 r 传递给 swap 时，C 代码是如何使用求地址运算符(&)的。

6.1.2.9 函数原型

Pascal 和 C 使用函数方面有一个很大的差别就是，Pascal 是通过出错检查来保证在函数中说明的参数个数和类型与函数调用时用到的参数个数和类型是匹配的。比如已定义了一个 Pascal 函数：

```
function Max(I, J: Integer): Integer;
```

接着就来调用它，参数用实型值 (A := Max(B, 3.52);)。结果会出现一个编译错误，指出参数的类型不匹配，因为 3.52 是一个实型数，而 J 要求代入一个整型数。

在 C 中，就不是这样的。C 中，在缺省情况下不对函数调用做错误检查。C 不检查参数的个数，参数的类型，甚至函数返回值的类型。这就提供了一定程序上的灵活性，如在定义函数前就能调用函数。但同时也会带来许多麻烦，详见 6.1.5 节。

为了避免这一麻烦，Turbo C 提供了函数原型，函数原型某种程度上类似于 Pascal 中的 forward 说明。一般的做法是在调用这些函数之前，在文件的开始地方说明函数原型。亦即，函数原型必须出现在函数进行实际调用之前。函数原型格式如下：

<类型>函数名(<类型><参数名>,<类型><参数名>,...);

这同 Pascal 中说明函数的方法很类似,但有一些差别。逗号(不是分号)被用来分隔各参数说明,并且一个<类型>不能用于多个<参数名>。下面是一些函数原型的例子,其对应子程序有的前面已给出,有的将在 6.1.3 节给出。

```
int max(int a, int b);
void swap(float *x, float *y);
void swapitem(listitem *i, listitem *j);
void sortlist(list l, int c);
```

与 Pascal 中 forward 语句不同, C 函数原型并不要求在实际调用时作任何改动,也就是说定义函数完全和没有函数原型一样(也可以用现代 C 风格来定义函数)。事实上,如果定义的函数与函数原型不匹配,那么 Turbo C 会给出编译错误。Turbo C 同时兼有传统风格和现代风格。由于 C 正在趋向现代风格,我们建议使用函数原型和原型风格的函数定义。

使用函数原型可以避免大量的问题。特别是开始编译 C 子程序库时,必须建立一个独立的文件,用以存放库中所用函数的头,如果要使用库中任一个子程序,只要在用户程序中包括嵌入文件即可(用 #include 指令)。在这种方法下,错误检查就能在编译时刻进行,这就能避免许多麻烦。

6.1.2.10 一个例子

本节介绍一个大例子。这个完整的程序使用了大多数已经介绍过的知识,还有一些尚未介绍。它定义了一个数组 mglist, 它的长度用常量 LMax 来定义,基类型用 ListItem 来定义(这里就是整型)。该程序给数组赋以一组递减的初始值,用 DumpList 子程序显示之,用 SortList 对它们进行递增排序,然后再显示。

注意这个程序的 C 描述,可能不是最好的,主要为了尽可能与 Pascal 描述相对应,有些地方不能对应就恰恰说明了 C 和 Pascal 间的一些不同之处。

这里有几点必须注意:

(1) 在 Pascal 描述中,过程 SwapItem 嵌套在过程 SortList 内部;在 C 中函数是不能嵌套的。因此只能把 swapitem 移到 sortlist 之外来。

(2) 在 C 中,数组下标通常从 0 开始,不断增大直到长度-1。比如说,数组 myList 中第一个项是 myList[0],而最后一项是 myList[LMAX-1]。所以 C 程序中各 for 循环和 Pascal 程序中的 for 循环下标不同。

(3) 在把 myList[] 传给 sortlist 时不必使用取地址和指针运算符。这是因为 C 通常传送数组的地址作参数,而不是数组本身。因为它不能传递数组中所有的值,否则会引起严重的问题。同样,在 dumpList 和 sortlist 中说明形式参数 list l 时, C 就知道在该地址处是一个数组,此时就不一定要用指针运算符了。

(4) 在这个例子中不需函数原型,因为每个函数在使用它之前都已定义了,如果想用函数原型,可以把它们放在程序中定义数据类型 item 和 list 之后的任何位置。这些函数原型为:

```
void swapitem(item *i, item *j)
void sortlist(list l, int c)
void dumpList(list l, int c)
```

注意,使用函数原型并不改变函数定义。

```

Program DoSort;
const
  LMax=100;
type
  Item= Integer;
  List=array [1..LMax] of Item;
var
  myList :List;
  Count,I: Integer;
  Ch      :Char;
Procedure SortList(var L : List;
                   C : Integer);
var
  Top,Min,K : Integer;
Procedure SwapItem(var I,J : Item);
var
  Temp : Item;
begin
  Temp := I; I := J; J := Temp
end; { of proc SwapItem }

begin { Main body of SortList }
  for Top :=1 to C-1 do begin

    Min := Top;
    for K := Top+1 to C do
      if L[K]<L[Min]
      then Min := K;
    SwapItem(L[Top],L[Min])
  end
end; { of proc SortList }
procedure DumpList(L : List;
                   C : Integer);
var
  I : Integer;
begin
  for I :=1 to C do
    Writeln('L[' ,I:3, ']= ',L[I]:4)
end; { End of proc DumpList }

begin { Main body of DoSort }
  for I :=1 to LMax do
    myList[I]:= Random(1000);
  Count := LMax;
  DumpList(myList,Count);
  Read(Kbd,Ch);
  SortList(myList,Count);
  DumpList(myList,Count);
  Read(Kbd,Ch)
end. { of DoSort }

```

```

#define LMAX 100

typedef int  item;
typedef item list [LMAX] ;

list myList;
int count, i;
char ch;
void swapitem(item *i,item *j)
{
  item temp;
  temp = *i; *i = *j; *j = temp;
} /*swapitem */

void sortlist(list l, list c)
{
  int top,min,k;
  for (top=0; top<c-1; top++)
  {
    min=top;
    for (k=top+1; k<=c; k++)
      if (l[k]<l[min])
        min=k;
    swapitem( &l[top],&l[min]);
  }
} /*sortlist*/
void dumplist (list l,int c)
{
  int i;
  for (i=0; i<=c; i++)
    printf("l[%3d] %4d\n",i,l[i]);
} /* dumplist */
main()
{
  for (i=0; i<LMAX; i++)
    myList[i] = rand() % 1000;
  count=LMAX;
  dumplist(myList,count);
  ch=getch();
  sortlist(myList,count);
  dumplist(myList,count);
  ch=getch();
} /*main*/

```

6.1.3 数据结构

在这一节中将概述Turbo C和Turbo Pascal在数据结构方面的异同。讨论的数据结构有：指针类型、数组类型、字符串类型、结构类型和联合类型。

6.1.3.1 指针类型

用Pascal编制程序可以长时间不用指针类型。但在C中不行。这是为什么？因为如前所述，在C中函数只有传递值的参数。如果要通过修改形式参数而改动实在参数，用户必须自己传送地址，并说明形式参数是实在数据类型的指针。而且C中的字符串是作为字符指针实现的，因此任何字符串处理都要用到指针。

〔例6.8〕 Pascal和C中的指针类型说明的简短比较例子：

Turbo Pascal		Turbo C	
说明:			
< 参数名 >	: ^ < 类型 >;	< 类型 >	* < 参数名 >;
Intptr	: ^ Integer;	int	*intptr;
Buff1	: ^ Intarray;	int	buff1[];
Buff2	: array[0..N] of Intptr;	int	*buff2[];
PHead	: ^ Node;	node	*phead;
Head	: Node;	node	head;
使用:			
< 参数名 > ^ := < 值 >;		* < 参数名 > = < 值 >;	
Intptr ^ := 22;		*intptr = 22;	
Buffer ^ [152] := 0;		buff1[152] = 0;	
PHead ^ .Next := nil;		(*phead).next = NULL;	
		/*或phead->next = NULL; */	

请注意在C中最后一个例子(phead)是怎样使用括号的，还有phead的第二方案中用到的特殊符号(→)。这里再举几个例子：

- | | |
|---------------------------|---------------------------|
| (1) Buff1 ^ [152] := 0; | buff1[152] = 0 |
| (2) Buff2[152] ^ := 0; | *buff2[152] = 0; |
| (3) Head.Data ^ := 0; | *head.data = 0; |
| (4) Head.Next := nil; | head.next = NULL; |
| (5) PHead ^ .Next := nil; | (*phead).next = NULL; |
| | /*或phead->next = NULL; */ |

第一个例子假定 buff1 指向一个整数数组；

第二个例子表明 buff2 是个整数指针的数组，因此在指针引用之前，先要变址；

第三个例子假设 head 是带有 next 域的记录（C 中的结构），next 是一个整数指针；

第四个例子假设 head 也有一个 next 域，它是一个指针（未说明指向什么）。

最后一个例子表明 phead 是一个记录指针，记录中包含 next 域：它是一个指针。

符号→是缩写记号。亦即，表达式

```
pname->fname = value;
```

表示 pname 是一个指向某类记录的指针, fname 是记录中某个域的名, value 将被赋给 pname 所指出的那个记录中的 fname 域。

6.1.3.2 数组

C 同 Pascal 相比, C 中的数组类型是相当简单的。C 中的数组下标可以是整数型, 字符类型或者是枚举类型, 而 Pascal 允许使用一切有序类型。C 中所有数组下标从 0 开始到 $n-1$ (n 为数组的长度)。这点上与 Pascal 很不同。Pascal 允许选择任意开始下标和任意结束下标。C 中, 数组下标计算有点像指针运算, 并且 Pascal 中的 $a[i]$ 同 C 中的 $a[i]$, $*(a+i)$ 是等价的。

两种语言中数组的一般格式是:

```
<数组名>: <array>[<下界>.. $\infty$ <上界>] <类型> <数组名>[<长度>];  
of <类型>;
```

这里 <长度> 等于 $(1 + \text{<上界>} - \text{<下界>})$ 。

C 中多维数组的说明和 Pascal 差不多, 要么类型本身就是某种数组, 要么在数组末尾添加另外的 <长度>:

```
<类型> <数组名>[<长度1>][<长度2>][<长度3>],
```

注意与 Pascal 不同的是 C 中不能用逗号来代替方括号对 $[]$ 作为省略表示法(见 6.1.5.5)。

在 C 中划出一块足以存放 <长度> 个 <类型> 实例的存储空间, <数组名> 就为指向块的首址常量指针。这便于把数组传送给函数, 更重要的一点是 (和 Pascal 不同) 函数在编译时刻不需要知道数组会占用多大空间。所以可以把不同长度 (但同一类型) 的数组传送给一给定函数。

比如如下函数, 它接收一个整数数组, 返回数组中的最小值:

```
int amin (int a[], int n),  
{  
    int min, i;  
    min = a[0];  
    for (i = 1; i < n; i++)  
        if (a[i] < min)  
            min = a[i];  
    return (min);  
}
```

把任意长度的整数数组传送给该函数, 会求出开始 n 个元素的最小值。C 程序设计员在使用 Pascal 时最大的不满就是这种灵活性不再存在了。

6.1.3.3 字符串类型

标准 Pascal 并没有把字符串类型定义为一种独立的数据类型。而 Turbo Pascal 中定义了, 并且提供了一些过程和函数处理之。C (包括 Turbo C) 并没有定义独立的字符串类型。相反, 把任意一个字符串定义为字符数组或字符指针, 这样定义的效果几乎是一样的。现在对两种语言的字符串类型用法作一比较:

< 变量名 >, string[<长度>];	char <变量名>[<长度>;
type	
BigStr = string[255];	typedef char bigstr[256];
StrPtr = ^BigStr;	typedef char *strptr;
var	
Line : string[80];	char line[81];
Buffer : BigStr;	bigstr buffer;
Word : string[35];	char word[36];
Ptr : StrPtr;	strptr ptr;

Turbo Pascal 和 Turbo C 字符串类型的主要区别和两种语言在数组方面的差别紧密相联。

在 Turbo Pascal 中,说明 S: string[N] 等价于 S: array of[0..N] of char。这个串最多可含 N 个字符,当前长度存储在 S[0] 中,而实际串本身从存储单元 S[1] 开始。可以直接把字符串字面常量或常量赋给字符串变量。Pascal 系统将负责传送各字节并设置长度字节。

在 Turbo C 中,可以用 char strarr[N] 或是 char *strptr 来说明一个字符串。第一种说明要划出 N 个字节来存放字符串,然后把这 N 个字节的首址存放在 strarr 中。第二种说明只要划出几个字节给 strptr,它指向字符类型。

在 C 中,字符串的长度不是单独存放的,而是用字符串的终止符来表示字符串结束。终止符是个空字符(ASCII 代码 0)。这就要求在字符串末端再用另外一个字节来表示结束,而字符串本身就从 strarr[0] 开始。

正由于这样,字符串 strarr 只能存放 N-1 个真正有用的字符,因另外一个字节用来保存空终止符。这就是为什么 C 的说明长度总比相应的 Pascal 说明长度大 1。

进一步说,由于 strarr 不是实际一组字节,因此不能直接赋以字符串字面常量,而要用子程序 strcpy(或其某一派生子程序)把一个字符串按字节传送到另一字符串,如 strcpy(strarr, "Hello, world!"). 也可以用 scanf 或 gets 直接把字符串值读到 strarr 中。

字符串说明的另一种方法是 char *strptr,使用时要特别小心。在这种情况下, strptr 只是一个字符指针,并没有给字符串分配空间,而只是给指针本身分配了几个字节。

可以直接把字符串字面常量赋给 strptr。因为这些字面常量编译后成为目标代码本身的一部分,仅需把它们的地址赋给 strptr。如果把 strarr 赋给 strptr,那么 strarr 和 strptr 都指向同一字符串。如果把其他字符串指针赋给 strptr,那么该指针和 strptr 也指向同一字符串。

那么,怎样才能使 strptr 指向自己私有的字符串而不是其他地方呢?这只要这样来分配空间:

```
strptr = (char*) malloc(N);
```

这就利用子程序 malloc 划出了 N 个字节的可用存储区域,并把首址送给 strptr。然后即可利用 strcpy 把字符串内容(常量或变量)复制到所分配的字节中去。

和(strptr, ptr)对应的Pascal形式只和它大致等价。这时需定义strptr为↑BigStr, 而不是↑char。这样 Turbo Pascal 中就把Ptr作为一个串, 还可以避免下标范围检查的问题。注意下面例子所要求空间仅是实际分配给Ptr的。

下面程序粗略比较了两语言字符串功能, 其中的类型说明已在前面出现过。

Turbo Pascal	Turbo C
<pre> var Line, Name : BigStr; First, Temp : string [80]; ptr : StrPtr; I, Len, Err : Integer; begin Write('Enter name: '); Readln(Name); I := pos(' ', Name); if I = 0 then First := Name else First := Copy(Name, I, I-1); Len := Length(First); Writeln('Len = ', Len); Temp := Concat('Hi, ', Name); Writeln(Temp); if Name <> First then Name := First; I := 823; Str(I, Temp); Val(Temp, I, Err); GetMem(Ptr, 81); Ptr^ := 'This is a test.'; Writeln('Ptr = ', Ptr^); FreeMem(Ptr, 81); end </pre>	<pre> main() { bigstr line, name; char first [81], temp [81]; char* ptr; int i, len; extern char* str chr(char*s, char ch); printf("Enter name: "); gets(name); ptr = strchr(name, ' '); if(ptr == NULL) strcpy(first, name); else strncpy(first, name, ptr-name-1); len = strlen(first); printf("len = %d\n", len); strcpy(temp, "Hi, "); strcat(temp, name); puts(temp); if(strcmp(name, first)) strcpy(name, first); i = 823; sprintf(temp, "%d", i); i = atoi(temp); ptr = (char*)malloc(81); strcpy(ptr, "This is a test."); printf("ptr = %s\n", ptr); free(ptr); } </pre>

Pascal源代码中完全可以不用Ptr。这里只是用于体会一下等价的C代码做些什么。

最后一点: 这个例子中调用的C子程序的函数原型已经列入嵌入文件中了。因此为了做正确的出错检查, 就应该把下面的#include语句放在Turbo C程序的开始处。

```

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

```

```
#include <stdlib.h>
#include <alloc.h>
```

6.1.3.4 结构类型

Pascal 和 C 中都允许定义不同类型数据的聚合体。在 Pascal 中称为记录，在 C 中则为结构。下面分别列出了它们的格式：

Turbo Pascal	Turbo C
<pre>type <记录名> = record <域名> : <类型>; <域名> : <类型>; ... <域名> : <类型> end, var <变量名> : <记录名>;</pre>	<pre>typedef struct { <类型> <域名>; <类型> <域名>; ... <类型> <域名>; } <记录名>; <记录名> <变量名>;</pre>

在 C 中还有一个简单的格式来直接说明结构变量，Pascal 也一样：

Turbo Pascal	Turbo C
<pre>var <变量名> : record <域名> : <类型>; ... <域名> : <类型> end;</pre>	<pre>struct <记录名> { <类型> <域名>; ... <类型> <域名>; } <变量名>;</pre>

在这种情况下，结构中的 <记录名> 是任选的，如果要说明其他变量为 <记录名> 类型，则一定要写上。除了这一点，Pascal 中的记录和 C 中的结构是非常相似的，现举例如下：

Turbo Pascal	Turbo C
<pre>type Student = record Last, First : string[20]; SSN : string[11]; Age : Integer; Tests : array[1..5] of Integer; GPA : Real end; var Current : Student;</pre>	<pre>struct student { char last[20], first[20]; char ssn[11]; int age; int tests[5]; float gpa; } current; main() {</pre>

begin	
Current.Last = 'Smith';	strcpy(current.last, "Smith");
Current.Age = 21;	current.age = 21;
Current.Test[1] = 97;	current.test[0] = 97;
Current.GPA = 3.94;	current.gpa = 3.94;
end.	}

Pascal 和 C 的唯一主要差别是 Pascal 中有 with 语句, 而 C 中没有。若把上面的 Pascal 代码用 with Current do 改写, 那么域前面就不要再写 Current 了。在 C 中, 域前面必须都有 current。C 中还有成员访问运算符 (→), 它用于运算符左边的标识符是一个结构指针而不是结构。比如, 假设 pstudent 是 struct 的一个指针, 那么

```
strcpy(pstudent→last, "Jones");
```

把 Jones 这个字符串复制到 "last" 域。

6.1.3.5 联合类型

Pascal 和 C 中都有类似的概念。在 Pascal 中称为自由联合变体记录, C 中就称为联合。现分别写出其定义, 并举例如下:

Turbo Pascal	Turbo C
type	
< 联合名 > = record	union< 联合名 > {
< 域 表 >	< 类型 > < 域名 >;
case< 类型 > of	< 类型 > < 域名 >;
< 变体表 > : (< 域表 >);	...
< 变体表 > : (< 域表 >);	< 类型 > < 域名 >;
...	}
< 变体表 > : (< 域表 >)	
end,	

在 Pascal 中, < 域表 > 通常是 < 域名 > : < 类型 > 的序列, 根据需要重复若干次。

在这点上, Pascal 和 C 有两个主要差别:

(1) Pascal 要求把联合部分放在一般记录的末端, 而 C 则不是, 但可以先说明联合, 然后再说明结构中某个域类型是该联合类型。

(2) Pascal 中允许联合类型中的每个变体有多个类型。而 C 中虽允许多个域 但类型都是相同的。

[例6.9] 现在研究的这个例子, 写法上尽量使 Pascal 和 C 的描述一致 (即使这样, 也应该承认它们并不是完全等价的)。

Turbo Pascal	Turbo C
type	typedef union {
trick.word = record	int w;
case integer of	struct {
0: (w : integer);	char lob;

<code>1: (lob,hib: byte);</code>	<code>char hib;</code>
<code>end;</code>	<code>} b;</code>
<code>var xp:trick_word;</code>	<code>} trick_word;</code>
	<code>trick_word xc;</code>

注意 Pascal 和 C 的 `trick_word` 定义都是不可移植的，它们都依赖于 8086 的字节序列。

在 C 的联合中，同结构一样，可以在右花括号和分号之间插入 <变量名> 域来直接说明该类型的变量。在这种情况下，如果不再说明其他这样的变量，可以不写 <联合名>。上述 Pascal 记录可引用的域是 `xp.w`, `xp.hib` 和 `xp.lob`，C 则为 `xc.w`, `xc.b.hib` 和 `xc.b.lob`。

6.1.4 编程问题

作为一个 Pascal 程序设计人员，在学习 Turbo C 时不会遇到很大困难。在两种语言的某些程序设计方面，其实现是有些不同的。在这一节中将讨论程序设计的一些主要问题。

6.1.4.1 字母大小写

Pascal 不区分字母的大小写，而 C 区分，这是说在 Pascal 中标识符 `indx`, `Indx` 和 `INDX` 是同一变量，而在 C 中是指三个不同的变量。

注意：由于函数调用，在连接 C 程序时才和库子程序匹配，标识符不同要那时才能发现。因此对 C 中大小写要特别注意。

6.1.4.2 类型转换

Pascal 中一般只允许有限的类型转换。函数 `ord()` 把任意有序类型转为整型，`chr()` 由整型（或相关类型）转为字符型。Turbo Pascal 中允许另外一些有序类型之间的类型转换（如整型、字符型、布尔型和枚举型）。C 就很自由了，允许把任一种类型转到任一种类型，而结果并不总是正确的。

下面是两种语言的类型转换的标准格式和一些例子：

Turbo Pascal	Turbo C
<code><变量> := <类型> (<表达式>);</code>	<code><变量> = (<类型>) <表达式>;</code>
<code>var Ch: Char;</code>	<code>char ch;</code>
<code>I := Integer(Ch);</code>	<code>i = (int) ch;</code>
<code>Ch := Char(Today);</code>	<code>ch = (char) today;</code>
<code>Today := Days(3);</code>	<code>today = (days) 3;</code>

另外，Turbo C 还会作许多自动类型转换，主要是用于整型兼容类型（这些类型的基本表示是整型）。因此，上面三个语句都可删去显式类型转换，而写成：

```
i = ch; ch = today; today = 3;
```

6.1.4.3 常量、变量的存储和初始化

Turbo Pascal 并不对说明的变量初始化。在调用子程序间也不保留局部变量的值。主要例外是类型常量都要初始化，并且在调用定义这些常量的子程序时保存其值（包括在执行中对它们的赋值）。在 C 中所有的全局变量在缺省情况下都有初值 0，除非在初始化时用户给它一个其他的值。

Turbo C提供了两种类型的常量, 允许对任意变量进行预初始化, 并且允许在函数内部说明静态变量。

(1) 常量类型

两种常量类型格式如下:

```
#define <常量名> <值>
```

```
const <类型> <常量名> = <值>;
```

第一种类型 (#define...) 和 Pascal 的常量说明 (const) 较相像。这时只要找到 <常量名>, 就直接用 <值> 替换。

除非实际上不能改变 <常量名>, 第二种类型 (const...) 和 Turbo Pascal 中的带类型常量较相像, 如果要修改或赋一个新值都会引起编译错误。

(2) 变量初始化

Turbo C 允许对任意变量初始化, 这种方法和 Turbo Pascal 的带类型常量很像, 它有如下格式:

```
<类型> <变量名> = <值>;
```

若要求的值不止一个 (如数组、结构) 就必须用括号括起来, 相互之间用逗号隔开 (如 {like_this, and_this, and_this_too})。

```
int x = 1, y = 2
```

```
char name[] = "Frank";
```

```
char answer = "Y";
```

```
char key = 3;
```

```
char list[2][10] = { "First", "Second" };
```

(3) 变量存储

C 给变量定义了一些存储类, 最重要的两个是外部变量和自动变量 (局部变量)。全程变量 (在包括 main 的所有函数外部说明的变量) 都默认为外部变量, 也就是说在程序开始执行时它们的初值都是 0, 除非是由用户自己赋初值。

在函数内部 (包括在 main 内) 说明的变量都默认为自动变量, 它们不进行初始化, 除非用户对它初始化。在调用这个函数时, 其值将丢失。然而, 可以说明这些变量为静态的, 在这种情况下, 将给它们赋初值 0 (一次性的, 在程序开始执行时), 并且在调用函数期间, 这些变量保留其值。

在下面的例子中:

```
init test(void)
{
    int i;
    static int count;
    ...
}
```

变量 i 存在堆栈, 在函数每次被调用时都要由函数 test 来初始化。静态变量 count 却存在全程数据区中, 在程序开始执行时得到初值 0。在每次调用函数 test 时, count 保留它以前的值。

6.1.4.4 动态存储分配

在 Turbo Pascal 中, 有几种不同的方法来管理堆。假设有如下 Turbo Pascal 说明,

```
type
  ItemType = Integer;
  ItemPtr = ^ItemType;
var
  p: ItemPtr;
```

有三种不同的方法分配和去配动态存储空间:

```
/*New 和 Dispose*/
New(p);          { 自动分配所需内存 }
...
Dispose(p);      { 自动去配所分配内存 }
...
/*New, Mark 和 Release*/
New(p);          { 自动分配所需内存 }
...
Mark(p);
Release(p);      { 自动去配从 P ^ 到堆底的动态内存 }
/*FreeMem 和 GetMem*/
GetMem(p, Sizeof(ItemType)); { 必需指定需分配内存 }
...
FreeMem(p, Sizeof(ItemType)); { 必需指定去配内存数量 }
```

在 Turbo C 中分配和去配动态存储空间是用子程序来完成的, 它们同 Turbo Pascal 中的 GetMem 和 Dispose 十分类似:

```
<类型> * <ptr>;
<ptr> = (<type> *) calloc (<数目>, <长度>);
/*或 <ptr> = (<类型> *) malloc (<总的长度>); */
/*或 <ptr> = (<类型> *) realloc (<op>, <unsz>); */
free (<ptr>);
typedef int ItemType;
ItemType *p;
p = (ItemType *) malloc (sizeof(ItemType));
...
free(p);
```

所有三个 C 子程序都返回一个通用指针, 它可以转换成适当的类型。如果堆中没有足够的可用存储空间, 那么三个子程序就返回 NULL。

函数 calloc 接收要创建项的数目和每个项的长度 (字节数), calloc 把各个项都置成 0, 并返回整个块的指针。这对动态创建数组是非常方便的。

malloc 接受要分配的字节数。

free 用于释放 <ptr> 所指向的存储空间。

6.1.4.5 命令行参数

用 Turbo Pascal 生成 com 文件时, 程序中可以用 ParamCount 和 ParamStr 函数来读入命令行上的所有参数。例如, 程序名为 DUMPIT.COM, 如下执行:

```
A>dumpit myfile.txt other.txt 72
```

ParamCount 会返回 3, ParamStr 会返回如下的值:

```
ParamStr(1)  myfile.txt
```

```
ParamStr(2)  other.txt
```

```
ParamStr(3)  72
```

同样, Turbo C (根据标准 C 约定) 允许说明标识符 argc, argv, env 作为 main 的参数。

```
main(int  argc, char *argv[ ], char *env[ ], )
```

```
{
```

```
... main 函数体 ...
```

```
}
```

这里 argc 是参数的个数, argv[] 是一个字符串数组, 用来记录参数。在同样的程序下, argc 产生 4, argv[] 将指向如下值:

```
argv[0]  A: \DUMPIT.EXE
```

```
argv[1]  myfile.txt
```

```
argv[2]  other.txt
```

```
argv[3]  72
```

```
argv[4]  (空)
```

在 C 中, 在 MS-DOS 3.X 版本下, argv[0] 有定义 (而 ParamStr(0) 没有定义) 并含有要执行的程序名。对于 MS-DOS 2.X, argv[0] 指向空串 (""). 注意, argv[4] 实际上也是空串。

第 3 个参数 env[] 是一个字符串数组, 所存各字符串形式如下:

```
envvar = value
```

这里 envvar 是一个环境变量名, value 是赋给 envvar 的串值。

6.1.4.6 文件输入/输出

在标准 Pascal 中, 有两种类型的文件: 正文文件和数据文件。打开、修改和关闭文件的序列几乎相同。Turbo Pascal 还提供了第三种文件类型 (无类型文件), 它和 Turbo C 中的二进制文件类似。

C 文件通常看作是字节流, 通过调用 fopen 时给出的 t(text) 和 b(binary) 可以区分正文文件和数据文件。

表 6.2 给出 Turbo Pascal 和 Turbo C 在文件输入输出方面的大致等价关系。

表 6.2 文件输入/输出相似性

Turbo Pascal		Turbo C	
var			
I	:Integer;	int	i;
X	:Real;	float	x;
Ch	:Char;	char	ch;

Turbo Pascal	Turbo C
Line :string[80]; MyRec:RecType; buffer:array[1..1024] of char	char line[80]; struct rectype myrec; char buffer[1024]
F1 :text; F2 :file of RecType; F3 :file;	FILE *f1; FILE *f2; FILE *f3;
Assign(< fvar >,< fname >); Reset(< fvar >), Reset(< untyped fvar >,< blocksize >);	< fvar >=fopen(< fname >,"r"); /*或< fvar >=fopen(< fname >,"r+"); */ /*或f1 =fopen(< fname >,"r+t"); */ /*或f2 =fopen(< fname >,"r+b"); */
Assign(< fvar >,< fname >); Rewrite(< fvar >); Rewrite(< untyped fvar >,<blocksize>);	< fvar >=fopen(< fname >,"w"); /*或< fvar >=fopen(< fname >,"w+"); */ /*或f1=fopen(< fname >,"w+t"); */ /*或f2=fopen(< fname >,"w+b"); */
Assign(< fvar >,< fname >); Append(< text fvar >);	< fvar >=fopen(< fname >,"a"); /*或< fvar >=fopen(< fname >,"a+t"); */ /*或< fvar >=fopen(< fname >,"a+b"); */
Read(F1,Ch); Readln(F1,Line); Readln(F1,I,X); Read(F2,MyRec); BlockRead(F3,buffer,SizeOf(buffer));	ch=getc(f1); fgets(f1,80,line); fscanf(f1,"%d %f",&i,&x); fread(&myrec,sizeof(myrec),1,f2); fread(&buffer,1,sizeof(buffer),f3);
Write(F1,Ch);	fputc(ch,f1); /*或fprintf(f1,"%c",ch); */
Write(F1,Line);	fputs(line,f1); /*或fprintf(f1,"%s",line); */
Write(F1,I,X);	fprintf(f1,"%d %f",i,x);
Writeln(F1,I,X);	fprintf(f1,"%d %f\n",i,x);
Write(F2,MyRec);	fwrite(&myrec, sizeof(myrec),1,f2);
Seek(F2,< rec# >);	fseek(f2,< rec# >,_sizeof(rectype),0);
Flush(< fvar >);	fflush(< fvar >);
Close(< fvar >);	fclose(< fvar >);
BlockWrite(F3,buffer,SizeOf(buffer));	fwrite(&buffer,1,sizeof(buffer),f3);

关于这些 Turbo C 输入输出子程序的细节可参阅参考文献[2]。

下面的短程序用来在屏幕上显示一个正文文件(文件名在命令行给出)。

<pre> program DumpIt; var F : Text; Ch : Char; begin Assign(F, ParamStr(1)); / \$I- } Reset(F); { \$I+ } if IOResult <> 0 then begin Writeln('Cannot open ', ParamStr(1)); Halt(1); end; while not EOF(F) do begin Read(F, Ch); Write(Ch) end; Close(F) end.</pre>	<pre> #include <stdio.h> main(int argc, char *argv[]) { FILE *f; int ch; f = fopen(argv[1], "r"); if (f == NULL) { printf("Cannot open %s\n", argv[1]); return(1); } while ((ch = getc(f)) != EOF) putchar(ch); fclose(f); }</pre>
--	--

6.1.5 Pascal 程序人员使用 C 时的常见错误

由于 Pascal 和 C 有许多地方很相似，所以经常会犯某些错误。这里列出了一些要注意的地方以避免错误。这些错误按其出现的可能性，Pascal 程序人员易发觉的程度以及编译程序能否查出有一大致次序。（第三章也讨论了一些常见的错误）。

6.1.5.1 赋值和比较的混淆

在 Pascal 中， $A = B$ 是一个布尔表达式，得到值 true 或 false。在 C 中， $A = B$ 是个赋值语句，把 B 的值给 A，然而（这点对理解很重要）这个表达式也要得到一个值，就是 B 的值（这个值赋给了 A）。对 Pascal 程序员来说最不习惯的是如下形式的语句，

if ($A = B$) <语句>;

这在 C 中是完全合法的，并按如下步骤执行：

- (1) B 的值赋给 A
- (2) 表达式 $A = B$ 的值就是 B 的值
- (3) 如果它的值为非 0 数（在 C 中表示 true），就执行 <语句>。

但实际想写的可能是：

if ($A == B$) <语句>;

按本来的想法它应该这样执行：如果 A 和 B 相等，就执行 <语句>。记住：在 C 中比较相等是用两个等号 ($==$)，而不是一个等号 ($=$)。单个等号在 C 中看作是赋值运算符。

6.1.5.2 忘了传送地址（尤其是在使用 scanf 时）

如前所述，C 只允许对函数的参数传送值，如果要给参数传送地址，程序员必须显式地

传送地址。假如写了一个函数 swap(见6.1.2.8), 也许会这样错误地调用它: swap(q,r); 其中 q 和 r 都是浮点类型。在这种情况下, swap 要取出 q 和 r 的值, 把它们解释成地址, 然后根据这些地址交换值。

怎样避免这个错误? 最好的方法是使用函数原型。这样, Turbo C 在编译时可以做相应的出错检查。对 swap 来说, 应当把如下的原型放在源文件开始处。

```
void swap(float *x, float *y);
```

现在, 如果编译到语句 swap(q, r); 就会得到一个出错信息指出在调用 swap 时参数 x 的类型不匹配。

6.1.5.3 在函数调用时漏写括号

在 Pascal 中, 无参过程的调用就使用过程名本身:

```
Anyprocedure;  
i := AnyFunction;
```

在 C 中, 对函数的调用——即使函数没有参数——必须有一个左括号一个右括号。程序员很容易写出这样的代码:

```
AnyFunction; /*代码无效*/  
i = AnyFunction; /*将AnyFunction的地址存在i中*/
```

而实际上想做的却是:

```
AnyFunction(); /*调用AnyFunction*/  
i = AnyFunction(); /*调用 AnyFunction, 结果存在 i 中*/
```

6.1.5.4 忽视警告信息

除了产生错误信息外, Turbo C 还给出非致命性警告。对上述不正确的函数调用, Turbo C 会报告这些警告:

```
Warning test.c 5: Code has no effect in function main
```

```
Warning test.c 6: Non-portable pointer assignment in function main
```

这两个语句都是合法的, 并且由于没有错误出现, 会产生 .OBJ 文件。注意! 这些警告在 Turbo Pascal 中通常是致命错误。不要对 Turbo C 中的警告信息掉以轻心。

6.1.5.5 不习惯多维数组下标的写法

假设有一个二维数组 matrix, 要引用位置(i,j), 作为一个 Pascal 程序员, 很可能会写出如下的语句:

```
x = matrix[i, j];
```

这在 C 中编译是完全对的, 然而其功能并不像所想像的。

在 C 中, 一串用逗号隔开的表达式是合法的, 在这种情况下, 整个表达式的值就是最后一个表达式的值。因此前面的语句就等于:

```
x = matrix[j];
```

这肯定不是所希望的, 但在 C 中却又是合法的。只会得到一个警告。因为 C 认为是想要把 matrix[j] 的地址——就是说 matrix[] 的第 j 列赋给 x。

在 C 中, 必须显式地把每个数组下标用方括号括起来, 因此应写成:

```
x = matrix[i][j];
```

记住: 对多维数组来说, 必须把每个下标放在各自的方括号内。

6.1.5.6 忘记了字符数组和字符指针的区别

假如有如下语句:

```
char *str1, str2[30];  
str1 = "This is a test";  
str2 = "This is another test";
```

第一个赋值是可以的,但第二个不行。为什么?因为 str1 是指向字符串的指针。当编译程序看到这个赋值语句时,就在目标文件某处存放一个串 This is a test 并把它的地址赋给 str1。

而 str2 是个常量指针,它指向某处 30 个字节的区域,不能改变它的地址,应当这样来写:

```
strcpy(str2, "This is another test");
```

常量串 This is another test 就按字节复制到 str2 所指向的地址中去。

6.1.5.7 忘记了 C 区分大小写

在 Pascal 中,标识符 indx, Indx 和 INDx 都是一样的,即大写和小写字母被认为是一样。在 C 中就不是这样的。因此如果说明了:

```
int INDx;
```

然后写语句:

```
for(indx = 1; indx < 10; indx++) <语句>
```

编译程序就会给出一个出错信息,说它不认识 indx。

6.1.5.8 复合语句中最后一个语句漏了分号

如果程序员是一个地道的 Pascal 主义者,只是在要求的地方而不是允许的地方才写上分号,开始使用 C 时就要遇到麻烦。幸运的是,编译程序能发现这一错误,并清楚地给出出错信息。而在 C 中,除两个例外,每个语句都要跟上分号。一个主要例外是函数语句:

<类型>函数名(<参数名列>)

它后面不要有分号。这点上不能同函数原型混淆:

<类型>函数名(<类型><参数名>, <类型><参数名>, ...);

这是用来说明函数而不是真正去定义它。这有些像 Pascal 中的 forward 说明。

另一个主要例外是预处理命令(#<cmd>),比如:

```
#include <stdio.h>  
#define LMAX 100
```

如果忘记了这一点,并打入了#define LMAX 100,那么预处理程序会将所有 LMAX 置换成 100;记住在 C 中对程序员来说有许多陷阱,必须非常仔细,它不像 Pascal 那样宽容。

6.2 Turbo Pascal 程序到 Turbo C 的转换

本章前面部分以对比的方式,讨论了 Turbo Pascal 和 Turbo C 语言的差别,并且已经看到了一些等价程序的例子。本节专门讨论如何把 Turbo Pascal 程序转换成 Turbo C 程序。首先介绍手工转换,然后讨论使用计算机辅助转换的翻译程序的设计。

6.2.1 把 Turbo Pascal 循环转换为 C 循环

循环是大多数程序的基本控制结构,现举例说明 Turbo Pascal 和 Turbo C 三种循环

之间的转换。

(1) For 循环

例如, `for x:=10 to 100 do writeln(x);` 可翻译为如下 C 语句

```
for (x=10; x<=100; ++x) printf("%d\n", x);
```

(2) while 循环和 repeat 循环

Turbo Pascal 与 Turbo C 的 while 循环实际上是一样的。可是, Turbo Pascal 的 repeat-until 循环和 Turbo C 的 do-while 循环需要运用不同的关键字。而且, 循环测试条件必须“相反”。

Turbo Pascal	Turbo C
<pre>while x<5 do begin writeln(x); read(x); end; repeat read(x); writeln(x); until x>5;</pre>	<pre>while(x<5) { printf("%d\n", x); x = getnum(); } do { x = getnum(); printf("%d\n", x); } while(x<=5);</pre>

请仔细观察从 repeat-until 到 do-while 的翻译。一定要颠倒测试条件的方向。

6.2.2 case 和 if 语句

绝大多数情况下, Pascal 的 case 语句可以直接翻译为 C 的 switch 语句。例如, 下列这两个程序段在功能上是等价的:

Turbo Pascal	Turbo C
<pre>case choice of 'E': enter; 'D': display; 'Q': quit; end;</pre>	<pre>switch(choice) { case 'E': enter(); break; case 'D': display(); break; case 'Q': quit(); }</pre>

只是当 case 语句用了一个子值域时, 才会出现问题。C 的 switch 语句不能接受子值域。例如, 下面的 case 语句就不能直接翻译成一个 C 的 switch 语句:

```
case time of
    0..6: sleep;
    7..8: getready;
    9..17: work;
    18..20: rest;
```

```

21..24: sleep;
end,

```

要翻译这种语句，必须运用一系列 if 语句。

Turbo Pascal 的 if 语句可直接翻译成 Turbo C 的 if 语句，不会有错。

6.2.3 结构和记录

除带变体的记录以外，可以把一个 Turbo Pascal 记录翻译成 Turbo C 结构。把一个带变体的记录翻译为一个结构，需要先建立一个含有记录的各个不同部分的联合，然后把这个联合变成结构的一个元素。例如，下面这个带变体的记录：

```

type
  PayType = (salaried, hourly, Laidoff);
  employee = record
    name: string[40];
    age: integer;
    case PayMethod: PayType of
      salaried: (MonthlyWage: real);
      hourly: (HourlyRate: real);
      Laidoff: (Nowage: boolean);
    end;
end;

```

被翻译成 Turbo C 时，成如下形式：

```

union PayMethod {
  float MonthlyWage, HourlyWage;
  char NoWage;
};
enum PayType {Hourly, Monthly, InActive};
struct employee {
  char name[40];
  int age;
  enum PayType method;
  union PayMethod Pay;
}

```

运用这个结构的程序必须人工地将 method 域放到与实际处于 pay 联合中的内容相一致的地方。

变体记录不是一种好的程序设计习惯，因为这样做非常容易产生错误，最好避免使用它。

6.2.4 一个手工转换的例子

为使读者能亲身体会一下转换过程，请将下列简单的 Turbo Pascal 程序转换为 Turbo C 程序：

```

program test (input,output);
var qwerty: real;

procedure func2 (x:integer);
begin
    writeln(x*2);
end;

function func1 (w:real):real;
begin
    func1:= w/3.1415;
    qwerty:= 23.34
end;

begin
    qwerty:= 0;
    writeln(qwerty);
    writeln('hello there');
    func2(25);
    writeln(func1(10));
    writeln(qwerty*2*4);
end.

```

四
内

Turbo Pascal 程序说明了一个函数和一个过程。因为在 C 中函数和过程一样,除了适当地返回数值,可不必顾虑它们的差别,func2 过程变成如下情况:

三

```

func2(x)
int x;
{
    printf("%d",x*2);
}

func1 函数变成:
float func1(w)
float w;
{
    qwerty= 23.34;
    return w/3.1415;
}

```

要注意到,因为 func1()将回送一个浮点数,必须精确地说明它,这可通过将类型说明 float 放置在 func1 名字之前。

另外,程序代码(即不包含在另一个函数或过程里面的以第一个begin开始的代码)必须转化为main()函数。它变为:

```

main()
{

```

```

    qwerty = 0;
    printf("%f",qwerty);
    printf("hello there\n");
    func2(25);
    printf("%f\n",func1(10));
    printf("%2.4f\n,qwerty);
}

```

最后需做的事是把全程变量 `qwerty` 说明为一个浮点数,然后把各个部分连结起来,就可以得到已翻译为 Turbo C 的下列程序:

```

float qwerty;
main()
{
    qwerty = 0;
    printf("%f",qwerty);
    printf("hello there\n");
    func2(25);
    printf("%f\n",func1(10));
    printf("%2.4f\n,qwerty);
}

func2(x)
int x,
{
    printf("%d",x*2);
}

float func1(w)
float w;
{
    qwerty = 23.34;
    return w/3.1415;
}

```

6.2.5 实现自动转换的一个试验原型

我们能够编制一个计算机程序,使其输入某一种语言的源代码而输出另一种语言的等价代码。编制程序的最佳方法是真正地为目标语言提供一个完整的语言分析程序。这个程序不是产生可执行的机器代码,而是输出目标语言的源程序。

这种计算机辅助翻译程序的基本思想是,它将接受源语言形式的 Turbo Pascal 程序作为输入,然后自动地实施一对一的转换,产生作为目标语言的 Turbo C 源程序,而把比较困难的转换留给程序员。例如,在 Turbo Pascal 中,给 `count` 赋值以 10,可写下述语句:

```
count:=10
```

在 Turbo C 中,除没有冒号以外,语句是一样的。计算机辅助程序可以把 Turbo Pascal

赋值语句的“:=”变化为 Turbo C 赋值语句的“=”。但 Turbo Pascal 和 C 存取磁盘文件的方法是不同的, 没有自动实施这种转换的简易方法, 不易进行的翻译只能留给程序员。

翻译程序首先需要的是—次能从输入文件中返回二个词法单位的函数。为此, 我们定义下述的 get_token() 函数:

```
get_token()
{
    register char *temp;
    tok_type = 0; tok = 0;
    temp = token;

    if(*prog=='\n') {
        *temp++ = '\r';
        *temp++ = '\n';
        *temp = '\0';
        prog++;
        tok_type = DELIM;
        return;
    }

    if(*prog=='\0') {
        *temp = '\0';
        tok_type=DELIM;
        return;
    }

    while(isspace(*prog)) ++prog; /* skip over white space */

    /* relational equals */
    if(*prog=='=') {
        prog++;
        strcpy(token, "==");
        tok_type = OP;
    }
}
```

(紧接下页)


```

    return;
}

/* assignment */
if(*prog=='') {
    prog++;
    if(*prog=='=')
    {
        *temp++ = '=';
        prog++;
    }
    else *temp++ = ',';
    *temp = '\0';
    tok_type = OP;
    return;
}

/* string */
if(is_in(*prog, "\"")) {
    *temp++ = '"'; prog++;
    while(!is_in(*prog, "\"")) *temp++ = *prog++;
    *temp = '"'; temp++; *temp = '\0'; prog++;
    tok_type = STRING;
    return;
}

/* other operators */
if(is_in(*prog, "+-*,./^%()")) {
    *temp = *prog;
    prog++; /* advance to next position */
    if(*temp=='.' || *temp=='/') *temp = ' ';
    temp++;
    *temp = '\0';
    tok_type = OP;
    return;
}

/* variables */
if(isalpha(*prog)) {
    while(isalpha(*prog)) *temp++ = *prog++;
    *temp = '\0';
    tok_type = IDENTIFIER;
    return;
}

/* numbers */
if(isdigit(*prog)) {
    while(!isalpha(*prog)) *temp++ = *prog++;
    tok_type = NUMBER;
    *temp = '\0';
    return;
}
prog++; /* unknown character */
}

```

Turbo Pascal 赋值号 “:=” 被转换为 Turbo C 的 “=”, 然后在 get_token() 中 “=” 被转换为 C 的等价符号 “==”。这种转换简化了程序其他部分的代码生成。

第二个重要的子程序是将 Turbo Pascal 关键字和一些函数翻译为 C 中的对应成分, 下面所示的 Translate() 是运用二维数组 trans 查找 Turbo Pascal 标识符, 然后返回它们对应的 C 成分:

```
char * trans[][2] = {
    "and", "&&",
    "begin", "{",
    "case", "switch",
    "div", "/",
    "do", "do",
    "else", "else",
    "end", "}",
    "forward", "extern",
    "for", "for",
    "function", "\n",
    "goto", "goto",
    "if", "if",
    "then", " ",
    "mod", "%",
    "nil", "\0",
    "not", "!",
    "procedure", "\n",
    "record", "struct",
    "repeat", "do",
    "until", "while",
    "while", "while",
    "write", "printf",
    "writein", "printf",
    "read", "scanf",
    "readln", "scanf",
    "real", "float",
    "integer", "int",
    "char", "char",
    "", ""
};

/* translate Turbo Pascal identifiers into Turbo C */
translate(s)
char *s;
{
    register int i;

    for(i=0; *trans[i][0]; i++)
        if(strcmp(s, trans[i][0])) {
            strcpy(s, trans[i][1]);
            return;
        }
}
```

可以很方便地把新标识符加到表中以扩充这个程序翻译新标识符。这个函数的改进需将一个已排序的标识符表放置在 trans 中，然后，应用二分查找法搜索适当的表目。如果读者想实施这种改进，要注意一些词（例如 program）在 C 中没有对应词，在这种情况下，就用换行代替。空串不能用，因为它被用来指示文件的结束。

完整的翻译程序如下所示：

```
/* computer assisted Turbo Pascal to C converter */

#include "stdio.h"
#include "ctype.h"

#define OP          1
#define IDENTIFIER  2
#define VAR         3
#define NUMBER      4
#define DELIM       5
#define STRING      6

char token[80];
int tok_type;
int tok;

char s[10000]; /* holds source file */
char *prog;

char *trans[][2] = (
    "and", "&&",
    "begin", "{",
    "case", "switch",
    "div", "/",
    "do", "do",
    "else", "else",
    "end", ")",
    "forward", "extern",
    "for", "for",
    "function", "\n",
    "goto", "goto",
    "if", "if",
    "then", " ",
    "mod", "%",
    "nil", "\0",
    "not", "!",
    "procedure", "\n",
    "record", "struct",
    "repeat", "do",
    "until", "while",
    "while", "while",
    "write", "printf",
    "writeln", "printf",
    "read", "scanf",
    "readln", "scanf",
    "real", "float",
    "integer", "int".
```

```

    "char", "char",
    "", ""
}

main(argc, argv)
int argc;
char *argv[];
{
    FILE *fp1, *fp2;
    char *p;
    int indent=0, i;

    prog = s;

    if(argc==3) {
        printf("usage: input output");
        exit(1);
    }

    if((fp1=fopen(argv[1], "r"))==0) {
        printf("cannot open input file\n");
        exit(1);
    }

    if((fp2=fopen(argv[2], "w"))==0) {
        printf("cannot open output file\n");
        exit(1);
    }

    while((*prog=getc(fp1))!=EOF)
        prog++; /* read in source */

    *prog = '\0';
    prog = s;

    for (;;) {
        get_token();
        if(*token) break; /* end of input file */
        p = token;
        /* if token is an identifier then translate it */
        if(tok_type==IDENTIFIER) translate(token);

        while(*p) putc(*p++, fp2); /* write it */
        /* put a space between tokens */
        if (*token!='\r') putc(' ', fp2);

        /* indent code to proper level */
        if(*token=='\r') {
            for(i=0; i<indent; i++) {
                putc(' ', fp2);
                putc(' ', fp2);
            }
        }
    }
}

```

```

        if(*token=='}') indent--;
        if(*token=='{') indent++;
    }
    fclose(fp1); fclose(fp2);
}

```

```

get_token()
{

```

```

    register char *temp;
    tok_type = 0; tok = 0;
    temp = token;

```

```

    if(*prog=='\n') {
        *temp++ = '\r';
        *temp++ = '\n';
        *temp = '\0';
        prog++;
        tok_type = DELIM;
        return;
    }

```

```

    if(*prog=='\0') {
        *temp = '\0';
        tok_type=DELIM;
        return;
    }

```

```

    while(isspace(*prog)) ++prog; /* skip over white space */

```

```

    /* relational equals */

```

```

    if(*prog=='=') {
        prog++;
        strcpy(token, "==");
        tok_type = OP;
        return;
    }

```

```

    /* assignment */

```

```

    if(*prog==',') {
        prog++;
        if(*prog=='=')
        {
            *temp++ = '=';
            prog++;
        }
        else *temp++ = ',';
        *temp = '\0';
        tok_type = OP;
        return;
    }

```

```

/* string */
if(is_in(*prog, "\"")) {
    *temp++ = '"'; prog++;
    while(!is_in(*prog, "\"")) *temp++ = *prog++;
    *temp = '\0'; temp++; *temp = '\0'; prog++;
    tok_type = STRING;
    return;
}

/* other operators */
if(is_in(*prog, "+-*,./%()")) {
    *temp = *prog;
    prog++; /* advance to next position */
    if(*temp == '.') *temp = ' ';
    temp++;
    *temp = '\0';
    tok_type = OP;
    return;
}

/* variables */
if(isalpha(*prog)) {
    while(isalpha(*prog)) *temp++ = *prog++;
    *temp = '\0';
    tok_type = IDENTIFIER;
    return;
}

/* numbers */
if(isdigit(*prog)) {
    while(!isdlim(*prog)) *temp++ = *prog++;
    tok_type = NUMBER;
    *temp = '\0';
    return;
}
prog++; /* unknown character */
}

isdlim(c)
char c;
{
    if(is_in(c, " , + - * ^ % ( ) " ) || c == 9 || c == '\r' || c == 0)
        return 1;
    return 0;
}

is_in(ch, s)
char ch, *s;
{
    while(*s) if(*s++ == ch) return 1;
    return 0;
}

```

```

/* translate Turbo Pascal identifiers into Turbo C */
translate(s)
char *s;
{
    register int i;

    for(i=0; *trans[i][0]; i++)
        if(!strcmp(s, trans[i][0])) {
            strcpy(s, trans[i][1]);
            return;
        }
}

```

实质上,把 Turbo Pascal 程序翻译为 Turbo C 的转换辅助程序先读入 Turbo Pascal 程序的全部源代码,然后一次从中获取一个词法单位,实施它所能做的翻译,并且写到一个 Turbo C 的文件中。除了一些运算符变化,标准的 strcmp() 函数可被用来探查可翻译的词法单位,strcpy() 函数把它们转换为适当的 Turbo C 词法单位。书写程序时对大小写是敏感的,且希望 Turbo Pascal 标识符用小写字母。也可以改变这一约定,但要与 Turbo Pascal 代码的书写一致。翻译程序对每个新的层次都自动把翻好的代码缩进两个空格。

读者可以利用下面的 Turbo Pascal 程序运行翻译程序:

```

program test (input,output);
procedure f1(x: integer);
begin
    writeln(x*2);
end;
function f2(w:real):real;
begin
    if w=100 then writeln('w is 100 inside f2');
    f2:=w/3.1415;
end;
begin
    writeln('hello there');
    f1(25);
    writeln(f2(10));
end.

```

这个 Turbo Pascal 程序将转化为如下的伪 C 程序输出。

```

test(input, output);
f1(x:int);
{
    printf(x*2);
};
f2(w:float):float;
{

```

```

    if w == 100 printf("w is 100 inside f2");
    f2 = w/3.1415;
};
{
    printf("hello there");
    f1(25);
    printf(f2(10));
}

```

之所以说它是伪C代码,是因为它还不是真正的C代码,但节省了相当多的键盘输入工作。程序所要做的就是用编辑程序作点编辑以及调整并纠正有出入的地方。

6.3 Turbo C与Turbo Pascal的连接

Turbo C和Turbo Pascal是两个功能强又各具特色的语言。以上,我们已经比较了两个语言的差异及其转换。本节讨论Turbo C与Turbo Pascal的连接,从而实现在一个程序中融合两种语言的特长。

现举例说明Turbo Pascal和Turbo C的接口,用于演示的两个例子程序是cpasdemo.pas和cpasdemo.c。Turbo C用来生成.OBJ文件(CPASDEMO.OBJ),在Turbo Pascal程序中用L编译指令连接这个.OBJ文件。

需要注意的是:

(1) 在Turbo C模块中说明的数据不能在Turbo Pascal程序中存取。共享数据必须在Pascal中定义。

(2) 如果C函数只用在部件(unit)实现部分,用near说明它们,如果在部件的接口部分说明,则用far。编译Turbo C模块时都用小存储模式。

(3) 不能使用Turbo C运行库函数,因为它们的模块没有正确的段名。但是,如果有Turbo C运行库的源文件,在用CTOPAS.BAT重编译后可以使用单个库模块,如果确已重编译,则在C模块中包含了所有需使用的C库函数的原型。

(4) Turbo C生成的某些代码是对内部子程序的调用。在没有对相应的Turbo C运行库源代码重编译前,它们不能使用。要运行该演示程序,需要以下几个文件:

TCC.EXE 和 TURBO.CFG 或者

TC.EXE 和 CTOPAS.TC

运行演示程序CPASDEMO.EXE的步骤是:

(1) 用Turbo C创建一个和Turbo Pascal 4.0相容的CPASDEMO.OBJ文件。

① 如果使用Turbo C集成开发环境(TC.EXE),则在DOS提示符下执行:

TC/CCTOPAS.TC CPASDEMO.C

然后按Alt-F9建立.OBJ文件。

② 如果使用Turbo C命令行版本(TCC.EXE),则在DOS提示符下执行:

TCC CPASDEMO.C

注意:为在Turbo Pascal 4.0中使用Turbo C模块,要用相同的配置文件(Turbo.CFG或CTOPAS.TC)。

(2) 编译并执行 Turbo Pascal 程序 CPASDEMO. PAS。

这个简单程序调用 Turbo C 模块中定义的每一个函数。这些函数通过调用 Turbo Pascal 过程 setcolor 改变当前显示颜色。

在配书软盘的 1 号盘上提供了这两个例子程序。为便于学习，下面介绍给读者。

CPASDEMO. PAS 程序：

```
program CPASDEMO;
uses Crt;
var
    Factor: Word;
{ $L CPASDEMO. OBJ }    {连接Turbo C生成的OBJ模块}
function Sqr (I: Integer): Word; external;
    {改变正文颜色，并回送I的平方}
function HiBits (W: Word): Word; external;
    {改变正文颜色，并回送W的高位字节}
function Suc (B: Byte): Byte; external;
    {改变正文颜色，并回送B + 1}
function Upd (S: Char): Char; external;
    {改变正文颜色，并回送C的大写字母}
function Prd (S: ShortInt): ShortInt; external;
    {改变正文颜色，并回送S - 1}
function LoBits (L: LongInt): LongInt; external;
    {改变正文颜色，并回送L的小写字母}
procedure StrUpd (var S: string); external;
    {改变正文颜色，并回送S的大写字母，注意，
    Turbo C子程序必须跳过串的长字节}
function BoolNot (B: Boolean): Boolean; external;
    {改变正文颜色并回送B的非}
function MultByFactor (W: Word): Word; external;
    {改变正文颜色并回送W * Factor,
    注意Turbo C可存取Turbo Pascal中全程变量}
Procedure SetColor (NewColor: Byte);
    {这个过程通过改变CRT变量TextAttr来改变当前显示颜色}
begin
    TextAttr := NewColor;
end { SetColor }
var
    s: string;
begin
    Writeln (Sqr(10));    {调用每一个定义的函数。传递适当的信息}
    Writeln (HiBits(30000));
```

```

    Writeln(Sus(200));
    Writeln(Upr('x'));
    Writeln(Prd(-100));
    Writeln(LoBits(100000));
    S:= 'abcdefg';
    StrUpr(S);
    Writeln(S);
    Writeln(S);
    Writeln(BoolNot(False));
    Factor:= 100;
    Writeln(MultbyFactor(10));
    SetColor ( LightGray )
end

```

下面是 cpasdemo.c 模块。该模块演示如何书写与 Turbo Pascal 程序连接的 Turbo C 程序。该模块中的子程序调用在 cpasdemo.pas 中的 Turbo Pascal 子程序。

```

typedef unsigned int word;
typedef unsigned char byte;
typedef unsigned long longword;
extern void setcolor(byte newcolor); /*在 Turbo Pascal 程序中定义的过程*/
extern word factor;                  /*在 Turbo Pascal 程序中定义的变量*/
word sgr ( int i )
{
    setcolor ( 1 );
    return ( i*i );
} /*sgr*/
word hibits ( word w )
{
    setcolor( 2 );
    return ( w>>8 );
} /*hibits*/
byte suc ( byte b )
{
    setcolor( 3 );
    return ( ++b );
} /*suc*/
byte upr ( byte c )
{
    setcolor( 4 );
    return ( (c>='a') &&(c<='z') ? c-32: c);
} /*upr*/
char prd ( char s )
{
    setcolor( 5 );
    return ( --s );
} /*prd*/

```

```

long lobits ( long l )
{ setcolor(6);
  return ( (longword) l & 65535 );
} /* lobits */

void strupr ( char *s )
{ int counter; /* 该子程序跳过 Turbo Pascal 的长字节 */
  for ( counter = 1; counter <= s[0]; counter++ )
    s[counter] = upr ( s[counter] );
  setcolor(7);
} /* strupr */

byte boolnot ( byte b )
{ setcolor(8);
  return ( b == 0 ? 1 : 0 );
} /* boolnot */

word multbyfactor ( word w )
{ setcolor(9) /* 该函数存取 Turbo Pascal 程序说明的变量 factor */
  return ( w * factor );
} /* multbyfactor */

```

第七章 Turbo C 与 Turbo Prolog 的接口技术

通过引入 Turbo C, 就能把两种对 PC 机都很适用的语言结合在一起。Turbo C 模块和 Turbo Prolog 模块的连接就能使人工智能 (AI) 和 Turbo C 的应用相结合, 正如已经看到的那样, 在某些方面 Turbo C 是优于 C 的其他实现的。本章将介绍 Turbo C 与 Turbo Prolog 的连接, 并给出演示编译和连接过程的若干例子。

Turbo C 是过程式语言, 而 Turbo Prolog 是一种基于逻辑程序设计的语言。Turbo C 应用程序和 Turbo Prolog 连接为人工智能领域的研究提供了有利条件, 也为 Turbo C 应用提供了人工智能能力, 这样需要解决比较高级的问题时只要给出问题的简单描述并让 Turbo Prolog 推理机来执行推理。连接 Turbo C 应用程序和 Turbo Prolog 程序, 可以显著地缩短软件开发时间, 提高代码清晰度和程序的灵活性。

7.1 Turbo C 与 Turbo Prolog 连接的步骤

Turbo C 模块与 Turbo Prolog 模块连接后, 两者的程序融为一体并能一气呵成地做下去。编译和连接的主要步骤是:

7.1.1 对程序模块进行编译

(1) 被 Turbo Prolog 调用的 C 函数必须用后缀 .O (参见 7.2.1 例 1 的 C 程序 CSUM.C)。

(2) 用 Turbo Prolog 主模块 (这个模块中包含一个目标), 代替 C 主模块。

(3) 在 Turbo Prolog 主模块中必须把 C 函数说明成全局谓词 (参见 7.2.1 例 1 的 Prolog 程序例子 PROSUM.PRO)。

(4) 所有的程序模块必须在大存储模式 (这是 Turbo Prolog 编译时使用的唯一存储模式) 下编译。

(5) 如果程序中调用 Turbo Prolog 1.1 版库, 对模块进行编译时, 必须把寄存器分配开关置成 off (-r-)。

(6) 产生下线符开关必须置成 off (-u-)。

7.1.2 对程序模块进行连接

(1) INIT.OBJ 必须是第一个连接的目标文件 (这是 Turbo Prolog 的初始化模块, 可在 Turbo Prolog 库磁盘中找到)。

(2) 第二个连接的目标文件是 CPINIT.OBJ (这是个特殊的初始化模块, 以保持 Turbo

C和Turbo Prolog 在存储分配上的相容性。CPINIT.OBJ 存于 Turbo C 磁盘上)。

(3) 如果要用Turbo C库函数,可使用CL.LIB;若有浮点运算,可用EMU.LIB和MATH.LIB。

连接命令行必须呈如下形式且必须是单个命令行,

```
tlink init cpinit <T_Prolog_Main> Other_files  
      <T_Prolog_Main.sym> . [exename] , [your_libs]  
      Prolog [emulib mathl] cl
```

7.1.3 其他注意事项

(1) Turbo Prolog函数调用Turbo C书写的函数类似于调用Turbo Prolog 内部谓词(函数)。然而,目前Turbo C不能这样调用Turbo Prolog模块。

(2) 所有要调用的 Turbo C库函数必须前缀以下线符(-)。注意:用户定义的函数则无需加下划线符。

(3) malloc, calloc, free 和其他存储分配函数用 pallocc, malloc-heap 和 release-heap 代替。pallocc, malloc-heap 和 release-heap 可利用 CPINIT.OBJ 进行存储分配,因而可在 Turbo C 函数内使用。pallocc 对全程堆栈进行存储分配,调用形式是 char*pallocc(int size)。mellocc-heap 对 Prolog 堆进行存储分配,调用形式是 char *malloc-heap(int size)。release-heap 是用来释放 Prolog 堆所占的存储空间,调用形式是 release-heap(char *ptr, int size)。在使用 pallocc 时,只有当出现了失败,存储器才会自动释放,并使 Turbo Prolog 回溯越过存储分配。

(4) Turbo C 和 Turbo Prolog 连接后, printf,putc 以及与屏幕输出有关的函数不再具有原有的功能。然而,wrch 可以写一字符到 Prolog 窗口,zwf 具有 writef 的相同功能,它类似于受限的 printf,

```
zwf (FormatString, Arg1, Arg2, ...)
```

其中,FormatString 是一个 printf 类型的格式串。

zwf 和 wrch 都在 PROLOG.LIB 中。

(5) 被 Turbo Prolog 调用的 C 函数不应有返回值,并且被定义为 void。参量的流模式在 Turbo Prolog 全局谓词说明中指定。例如:

```
factorial (integer, real) - (i, o) language C
```

使得 Turbo Prolog 知道 factorial 是个函数,并且它有两个变量——第一个是个整型数,第二个是实数(浮点数)。(i, o)是指第一个参量(整型数)是要被传送进来的,而第二个参量是一个指向浮点数的指针,它将在 factorial 内被赋值。C 告诉 Turbo Prolog 这里是使用 C 调用协定(参见7.2节例3的 DUBLIST.C 和 PLIST.PRO)。

注意,值都是通过引用返回的。关于流模式进一步的信息,参看例3中改变流模式的讨论。

7.2 Turbo C 与 Turbo Prolog 的连接示例

本节中,将提供四个实例,演示 Turbo C 和 Turbo Prolog 程序的编译和连接过程。

7.2.1 示例之一：两个整型数相加

这个例子是 Turbo C 函数（两个整型数相加）和 Turbo Prolog 模块结合，把 C 函数的结果输出到当前窗口中。

Turbo C 源文件：CSUM.C

```
/* 输出函数 zwf 与 C 语言输出函数 printf 相似。它把输出结果显示在当前窗口上 */
extern void zwf(char *format,...);
void sum_0(int parm1, int parm2, int *res_p)
{
    zwf("This is the sum function : parm1 = %d,
        parm2 = %d", parm1, parm2);
    *res_p = parm1 + parm2;
}
/*End of sum_0*/
```

编译 CSUM.C 产生 CSUM.OBJ

对 CSUM.C 编辑和保存后，就要选择编译选择项。Turbo C 提供了两种方法：

(1) 从 Turbo C 菜单上选择以下的编译时的选择项：

```
Options/Compiler/Model/Large/(-ml)
Options/Compiler/Optimization/Jump Optimization/(-o)
Options/Compiler/Code generation/Generate underbars Off/(-u-)
Options/Compiler/Optimization/Use register variables Off/(-r-)
```

一旦选定了这些选择项，需从 Turbo C 主菜单中选择 Options/Store 项，以保存这些设置的参数，然后选择 Compile。Turbo C 根据选择项对 CSUM.C 进行编译，产生目标模块 CSUM.OBJ。

(2) 如果不使用 Turbo C 的菜单，而使用标准 C 命令行对 CSUM.C 编译，就必须 在 DOS 提示符下打入以下信息：

```
tcc -ml -o -c -u- -r- csum
```

注意：Turbo Prolog 仅在大存储模式下编译，因此要把 Turbo C 和 Turbo Prolog 连接起来，必须使用 -ml 编译选择项（大存储模式）。

Turbo Prolog 源文件：PROSUM.PRO

```
global predicates
cpinit language C /*在 Turbo Prolog 主模块中要把 CPINIT 说明为全局量*/
sum(integer, integer, integer) -(i,i,o) language c
/*sum 的流模式说明为 (i, i, o)，表示第三参量是返回值，而前二个参数是输入的*/
goal cpinit, sum(7, 6, x), write("SUM=", x).
/*cpinit 必须在第一个 C 函数被调用之前调用*/
```

编译 PROSUM.PRO 产生 PROSUM.OBJ

对 PROSUM.PRO 编辑保存后，需把它编译成目标文件（OBJ），以便连接 Turbo C 目标模块。因此首先在 Turbo Prolog 主菜单上选择 Options/Obj，然后再选择 Compile。在 Turbo Prolog 编译源文件生成目标文件后，就能进行连接和运行了。

连接 CSUM.OBJ 和 PROSUM.OBJ

为了连接Turbo Prolog模块和Turbo C模块,可以使用Turbo C集成开发环境、Turbo Link(Turbo C软件包中独立的连接程序)或者相容的连接程序(Microsoft linker2.2或更高版本)。除了tlink和link命令,连接命令行参量还包括Turbo Prolog主模块,各种其他模块,输出文件和库。除非有特别说明,一般情况下,它们要按如下次序出现:

① Turbo Prolog初始化模块:

INIT.OBJ(Turbo Prolog初始化模块)。

② Turbo C初始化模块:

CPINIT.OBJ(Turbo C初始化模块,使Turbo C与Turbo Prolog相容)。

③ Turbo Prolog主模块:

一个Turbo Prolog主模块,它包含一个目标。

④ 相配的模块(这些模块出现的顺序无关紧要),

- 汇编语言目标模块
- Turbo C目标模块
- Turbo Prolog目标模块

⑤ 符号表模块:

Turbo Prolog主符号表文件名(这是必须具有的且在最后出现)。

⑥ 输出文件名:

要产生的可执行文件名。

⑦ 库:

列出包含相配模块需要函数的所有库。必须严格按下列次序:

- 用户定义的库
- PROLOG.LIB
- EMU.LIB和MATHL.LIB(如果需要的话)
- CL.LIB

在这个例子中,使用了Turbo Link(tlink命令)并给出了如下的参量:

- Turbo Prolog程序INIT.OBJ和PROSUM.OBJ
- Turbo C目标模块CSUM.OBJ
- 符号表PROSUM.SYM和可执行文件TEST.EXE
- 库PROLOG.LIB, CL.LIB(EMU.LIB和MATHL.LIB用于进行浮点运算)

需要注意的是,PROSUM.SYM是包含了程序PROSUM.OBJ中的变量的名和类型的文件。

下面是第一个例子的连接命令行:

```
tlink init cpinit prosum csum prosum.sym, test.exe, prolog+cl
```

7.2.2 示例之二: 使用数学库

第二个例子类似于第一个例子,它说明了如何写两个Turbo C函数以及如何把它们与一个Turbo Prolog程序连接,每个Turbo C函数都用其各自独立的源文件表示,CSUM1.C把两个浮点数相加并返回其和,FACTRI.C计算一个整型数的阶乘。Turbo Prolog程序FACTSUM.PRO将程序的结果写在两个Prolog窗口中。这个例子用到了Turbo C大存储模式数学库MATHL.LIB。

Turbo C 源文件: CSUM1.C

```
extern void zwf(char *format, ...);  
void sum.o(double parm1, double parm2, double *res.p)  
{  
    *res.p = parm1 + parm2;  
    zwf("This is the sum function, parm1 = %f, parm2 = %f, result = %f",  
        parm1, parm2, *res.p);  
}
```

Turbo C 源文件: FACTRL.C

```
void factorial.o(int top, double *result)  
/*产生阶乘序列*/  
{  
    double x;  
    int i;  
    if (top < 1) {  
        *result = 0.0;  
        return;  
    }  
    for (x = 2.0, *result = 1.0; top > 1; top --, x = x + 1.0)  
        *result = *result * x;  
} /*End of factorial.o*/
```

编译CSUM1.C和FACTRL.C产生.OBJ文件

像第一个例子一样, 在与其他模块或 Turbo Prolog 主程序连接之前, 必须把这两个 Turbo C 模块编译成目标文件(.OBJ)。可以从 Turbo C 主菜单中挑选和设置编译时的选择项, 然后为每个 C 源文件选择编译命令。也可在标准 C 命令行中使用 tcc 命令来编译两个 .C 源文件。不管是哪种情况, 至少都要从如下的编译时刻选择项中挑选一个:

```
Options/Compiler/Model/Large/(-ml)  
Options/Compiler/Optimization/Jump Optimization/(-o)  
Options/Compiler/Code generation/Generate underbars Off/(-u-)  
Options/Compiler/Optimization/Use register variables Off/(-r-)
```

Turbo Prolog 源文件: FACTSUM.PRO

FACTSUM.PRO 是个主 Turbo Prolog 程序, 它开辟了两个窗口: 一个显示 Turbo C 模块的输出, 另一个显示 Turbo Prolog 程序的输出。模块和程序应按如下次序出现:

① FACTSUM.PRO 提示用户输入一个整型数 Int, 由 Turbo Prolog 程序把这个数传送给 FACTRL.C。

② FACTRL.C 中的 Turbo C 函数 factorial 返回 Result(Int 的阶乘)到 FACTSUM.PRO。

③ FACTSUM.PRO 在一个窗口上写 Result 并再次提示用户输入一个数 (这次是个浮点数)。

④ FACTSUM.PRO 把第二次输入的数 Real 和上次计算的阶乘 Result 传送给模块 CSUM1.C。

⑤ CSUM1.C 中的 Turbo C 函数 sum 把 Real 和 Result 相加, 并把结果 sum 送给

FACTSUM.PRO.

⑥ FACTSUM.PRO 把 sum 写在一窗口上, 这时程序结束。

下面是这个 Turbo Prolog 程序的清单:

```
/*Turbo C 模块的说明须放在 Turbo Prolog 领域定义和数据库说明后(如果什么都出现的话)。所有的全程模块在 Turbo Prolog 中均作为全局谓词调用, 并且要有流模式和语言说明。*/
```

```
global predicates
```

```
sum(real, real, real) - (i, i, o) language c.
```

```
factorial(integer, real) - (i, o) language c
```

```
cpinit language c /*调用 Turbo C 初始化程序*/
```

```
/*这个例子很简单, 只用了一个外部句子(Turbo C 模块), 因此只要有一个目标段。
```

```
但是在实际应用中也需要一个子句段。*/
```

```
goal cpinit
```

```
makewindow(1,49,31,
```

```
    "A Turbo Prolog window to the Turbo C program", 0, 0,15, 80),
```

```
makewindow(2, 47, 3,
```

```
    "A Turbo Prolog window to the Turbo Prolog program",15, 0,10,80),
```

```
/*提示用户输入第一个数据*/
```

```
write("Enter an integer; Turbo C will calculate the factorial, "),
```

```
    readint(Int), nl,
```

```
    shiftwindow(1), /*把输出窗口换到 Turbo C 窗口*/
```

```
    /*调用 Turbo C factrl 模块, 计算阶乘*/
```

```
    factorial(Int, Result),
```

```
    shiftwindow(2), /*把输出窗口转换到 Turbo Prolog 窗口*/
```

```
/*提示用户输入第二个数据*/
```

```
write("Enter a real number to add to the factorial,"),
```

```
    readreal(Real), nl,
```

```
    shiftwindow(1), /*把输出窗口转到 Turbo C 窗口*/
```

```
    /*调用 Turbo C csum1 模块, 求和*/
```

```
    sum(Result, Real, sum),
```

```
    shiftwindow(2), /*把输出窗口换到 Turbo Prolog 窗口*/
```

```
/*把第一个计算结果写到窗口上*/
```

```
write("The factorial of", Int, "is", Result), nl,
```

```
/*把第二个计算结果写到窗口上*/
```

```
write("The result, ", Result, "+", Real, "=", sum), nl.
```

编译 FACTSUM.PRO 产生 FACTSUM.OBJ

像第一个例子那样, 在同其他模块连接之前必须把 Turbo Prolog 源文件编译成一个目标文件(.OBJ)。从 Turbo Prolog 主菜单上选择 Option/Obj, 然后再编译这个程序。

连接 CSUM1.OBJ, FACTRL.OBJ 和 FACTSUM.OBJ

在这个例子中连接用到的参数为:

- ① Turbo Prolog 目标模块是 INIT.OBJ 和 FACTSUM.OBJ;
- ② Turbo C 目标模块是 CSUM1.OBJ, FACTRL.OBJ 和 CPINIT.OBJ;
- ③ 输出文件名是 FACTSUM.SYM (符号表) 和 SUM.EXE (可执行文件);

④ 所用到的库有PROLOG.LIB, EMU.LIB, MATH.LIB和CL.LIB。

连接这些模块的命令是:

```
tlink init cpinit factsum factrl csum1 factsum.sym,,sum,,prolog+emu+  
mathl+cl
```

7.2.3 示例之三: 使用流模式和存储分配

这个例子给出了这样一些代码用来在 Turbo C 中创建 Turbo Prolog 的 functor 和 list, 并把这些新的结构返回给 Turbo Prolog。同时, 该例也说明了 Turbo Prolog 的全程堆栈是如何进行存储分配的。List 是三个元素的递归结构, functor 是两个元素的 C 结构 (在这个例子后面有完整的描述)。

(1) Turbo C 模块 DUBLIST.C 包含了三个函数。前两个函数可以是输入一张整型数表, 返回包含了表中第一个整型数的一种结构; 也可以是输入一种带有整型数的结构, 返回一张包含了这个整型数的表。第三个函数输入一个整型数 n , 产生有两个整型数的表, 第一个数是 n , 第二个数是 $2n$ 。

② 必须注意每个 Turbo Prolog 全局谓词中可以有变化的流模式, 并且每个流模式要求一个变化 Turbo C 函数。如在下例中 `clist_0` 必须对应第一个流模式 (i, o) , 而 `clist_1` 则对应第二个流模式 (o, i) 。

global predicates

```
clist(ilist, ifunc) - (i, o)(o, i) language c
```

(i, o) 说明 `ilist` 是要送入 Turbo C 函数 `clist_0` 的, 而 `ifunc` 是一个结构指针, 它在 Turbo C 函数 `clist_0` 中定义。 (o, i) 说明 `ifunc` 要被送入 `clist_1`, 而 `ilist` 是表结构指针, 它在 `clist_1` 中定义。

(3) 如果在 Turbo Prolog 全局领域中还有其他的流模式, 那么就需要由 `clist_2` 来处理这个流模式。

Turbo C 源文件: DUBLIST.C

```
struct ilist  
{    char functor;                /*表元素类型*/  
                                           /*1 = 一个表元素*/  
                                           /*2 = 表尾*/  
    int val;                      /*实际元素*/  
    struct ilist *next;           /*指向下一个节点*/  
}  
struct ifunc  
{ char type;                     /*functor的类型*/  
  int value;                      /*functor的值*/  
}  
void clist_0(struct ilist *in, struct ifunc **out)  
{  
    if(in->functor != 1) fail_cc(); /*如果表空则失败*/  
    *out = (struct ifunc*) malloc(sizeof(struct ifunc));  
    (*out)->value = in->val;        /*把这送到f(x)中*/  
}
```

```

(*out)→type = 1;          /*置functor类型*/
}
void clist.1(struct ilist **out, struct ifunc *in)
{
    int temp=0;
    struct ilist *endlist=(struct ilist *) palloc(sizeof(char));
    endlist→functor = 2;
    temp = in→value;
    temp +=temp;
    *out = (struct ilist *) palloc( sizeof(struct ilist));
    (*out)→val = temp;      /*以表的形式返回 [2, X] */
    (*out)→functor = 1;     /*置表类型, 如果不这样做, 返回的值就毫无意义*/
    (*out)→next = endlist;
}

void dublist.0 (int n, struct ilist **out)
{
    /*这个函数创建表 [n, n+n] */

    struct ilist *temp;
    struct ilist *endlist=(struct ilist*) palloc(sizeof(char));
    endlist→functor = 2;
    temp=(struct ilist *) palloc(sizeof(struct ilist));
    temp→val = n;           /*把表中的第一个元素置为n*/
    temp→functor = 1;
    *out = temp;

    /*现在分配第二个表元素*/
    temp=(struct ilist *) palloc(sizeof(struct ilist));
    temp→val = n + n;      /*把值n+n赋给第二个表元素*/
    temp→functor = 1;
    temp→next = endlist;   /*把第二个元素后的节点送到表尾节点中*/
    (*out)→next = temp;    /*在第一个元素之后*/
}

```

Turbo Prolog 中的表和函子都是 Turbo C 中的结构(见DUBLIST.C)。

表是有三个元素的递归结构。第一个元素是类型,如果是表元素,则值为1,如果是表尾,则值是2。第二个元素是个实际值,它必须与Turbo Prolog 中的元素的类型相同。例如,一张浮点数表是:

```

struct alist {
    char funct;
    double elem;          /*表元素是实数*/
    struct alist *next;
}

```

第三个元素是指向下一个节点的指针。每个节点都是一个结构,它可能是类型1,表示指向表的下一个元素,也可能是类型2,说明已是表尾或是空表。

Turbo Prolog的函子是两个元素的C结构。第一个元素对应于Turbo Prolog领域说明(有关Turbo Prolog领域说明的具体细节请参阅参考文献[11])。比如:

```
domains
    func = i(integer);s(string)
predicates
    call(func)
goal
    call(x)
```

Turbo Prolog中的函子func有两种类型:一种是整型,另一种是字符串。所以在这个例子中,Turbo C结构的类型元素可能是1或2,1对应第一种类型,2对应第二种。

Turbo C结构的第二个元素是函子中这个元素的实际值,并定义为参量的可能类型的联合:

```
union val {
    int ival;
    char *svar;
};
struct func {
    char type;           /*类型可能为1或2对应Turbo Prolog的领域说明*/
    union val value;     /*函子元素的值*/
}
```

注意:在存储管理中必须要用到函数palloc, malloc_heap和release_heap。

这些函数在下列情况下使用:

- (1) 给存放在Turbo Prolog堆或堆栈中的Turbo C结构分配存储空间;
- (2) 释放Turbo Prolog堆的存储空间。

使用palloc时如果出现失败,则自动释放存储空间,并使Turbo Prolog回溯越过存储分配。下面给出Turbo C调用时的语法。

```
char *palloc (size)           /*在栈中分配存储空间*/
char *malloc_heap(size)       /*在堆中分配存储空间*/
release_heap(ptr,size)        /*释放堆空间*/
```

下面给出Turbo Prolog主模块PLIST.PRO,它调用DUBLIST.C中的函数,并打印结果。

```
domains
    ilist = integer*
    ifunc = f(integer)
global predicates
    cpinit language c
    clist(ilist, ifunc) - (i,o)(o,i) language c
    dublist(integer, ilist) - (i, o) language c
goal cpinit
```

```

clearwindow,
clist( [3] , X),          /*X约束为f(3)*/
write("X=" X), nl,
clist(Y, X),              /*Y约束为 [6] */
write("Y=", Y), nl,
dublist(6, Z),            /*Z约束为 [6, 12] */
write(Z), nl.

```

编译DUBLIST.C

就像前二个例子一样，在与Turbo Prolog主模块PLIST.PRO连接之前必须将Turbo C模块DUBLIST.C编译成目标文件(.OBJ)。

连接命令是：

```
tlink init cpinit plist dublist plist.sym,dublist,,prolog+emu+mathl+cl
```

7.2.4 示例之四：画三维条形图

这个例子将说明如何编译、连接C和Prolog模块使之成为一个统一的、混合语言的程序，并且既有AI的灵活性又有C的图形处理能力。此例中，提供了下述程序模块。

- 能画条形图的Turbo C模块BAR.C，它的输入来自另一文件。
- Turbo Prolog主模块PBAR.PRO，它要求从用户那里得到输入。

Turbo C源文件BAR.C

```

/***** BAR.C *****/
/*
/* C program for interfacing Turbo C with Turbo Prolog */
/*
/*****

#include <dos.h>
int errno;
videodot(int x, int y, int color)
{
    union REGS intr, outr;

    intr.h.ah = 12;          /* write pixel          */
    intr.h.al = color;
    intr.x.cx = x;
    intr.x.dx = y;
    int86(16, &intr, &outr); /* call video intr */
}

/* Draws a line on the screen from (x1, y1)
   to (x2, y2) in a selected color */
line(int x1, int y1, int x2, int y2, int color)
{
    int xdelta,              /* The change in x coordinates */
    int ydelta,              /* The change in y coordinates */
    int xstep;               /* The change to make in the x
                               coordinate in each step */
}

```

```

int ystep;      /* The change to make in the y
                  coordinate in each step */
int change;     /* The amount that the x or y
                  coordinate has changed */

xdelta = x2 - x1; /* Calculate the change in
                  x coordinates */
ydelta = y2 - y1; /* Calculate the change in
                  y coordinates */
if (xdelta < 0)
{
    /* The line will be drawn from
       right to left */
    xdelta = -xdelta;
    xstep = -1;
}
else
    /* The line will be drawn from
       left to right */
    xstep = 1;
if (ydelta < 0)
{
    /* The line will be drawn from
       bottom to top */
    ydelta = -ydelta;
    ystep = -1;
}
else
    /* The line will be drawn from
       top to bottom */
    ystep = 1;
if (xdelta > ydelta) /* x changes quicker than y */
{
    change = xdelta >> 1;
    /* change set to twice the
       value of xdelta */
    while (x1 != x2)
    /* Draw until the terminating
       dot is reached */
    {
        videodot(y1, x1, color);
        /* Draw a dot on the screen */
        x1 += xstep;
        /* Update x coordinate */
        change += ydelta;
        /* Update change */
        if (change > xdelta)
        /* If change is large enough to
           update the y coordinate */
        {
            y1 += ystep;
            /* Update the y coordinate */
            change -= xdelta;
            /* Reset change */
        }
    }
}

```

```

    }
}
else
{
    /* y changes quicker than x */
    change = ydelta >> 1;
    /* change set to twice the
       value of ydelta */
    while (y1 != y2)
    /* Draw until the terminating
       dot is reached */
    {
        videodot(y1, x1, color);
        /* Draw a dot on the screen */
        y1 += ystep;
        /* Update y coordinate */
        change += xdelta;
        /* Update change */
        if (change > ydelta)
        /* If change is large enough
           to update the x coordinate */
        {
            x1 += xstep;
            /* Update the x coordinate */
            change -= ydelta;
            /* Reset change */
        }
    }
}
} /* End */

bar_0(int x1, int y1, int width, int height, int color)
{
    int count; /* Counter variable used
                in filling bar */
    int x2, y2, x3, y3, x4, y4;
    /* Additional points on the bar */
    int wfactor; /* The number of pixels to
                  shift the bar to the right */
    int hfactor; /* The number of pixels to
                  shift the bar up */

    wfactor = width / 5; /* figure out wfactor
                           and hfactor */
    hfactor = height / 12;
    x2 = x1 + wfactor; /* compute the location
                           of the points on the bar */
    x3 = x1 + width;
    x4 = x3 + wfactor;
    y2 = y1 - hfactor;
    y3 = y1 + height - hfactor;
    y4 = y1 + height;
}

```

```

line(x1, y1, x3, y1, 2);
/* draw front of the bar */
line(x3, y1, x3, y4, 2);
line(x3, y4, x1, y4, 2);
line(x1, y4, x1, y1, 2);
line(x3, y1, x4, y2, 2);
/* draw side of the bar */
line(x4, y2, x4, y3, 2);
line(x4, y3, x3, y4, 2);
line(x1, y1, x2, y2, 2);
/* draw top of the bar */
line(x2, y2, x4, y2, 2);
line(x1, y4, x2, y3, 3);
line(x2, y3, x4, y3, 3);
/* draw back of the bar */
line(x2, y3, x2, y2, 3);
for (count = y1 + 1; count < y4; count++)
/* fill in the bar */
line(x1 + 1, count, x3, count, color);
} /* bar_0 */

```

编译BAR.C

像前三个例子一样，在与Turbo Prolog主模块PBAR.PRO连接之前必须先将Turbo C模块BAR.C编译成目标文件(.OBJ)。

Turbo Prolog 程序PBAR.PRO

这个Turbo Prolog程序提示终端用户来生成、保存、装入或画一个条形图。

如果用户想要生成一个条形图，那么这个程序就会接收关于图的说明信息，在窗口上给每个条形定位，并调用C模块画每个条形。当所有的条形都画好后，就把它送入数据库中。

终端用户可能需要保存这个条形图，PBAR即把当前条形图的描述保存到一个文件中以备后用。

如果终端用户选择装入选择项，PBAR就删除当前条形图的描述，从一个文件中装入用户说明的条形图描述。

如果是最后一个选择即画条形图，PBAR就使用数据库中的描述递归调用Turbo C的BAR模块，这个模块根据数据库中的当前描述画出一个条形图。

```

/****** PBAR.PRO *****/
/*
/* PROLOG program for interfacing Turbo C with Turbo Prolog */
/*
/*
/******

database
  barchart(integer)
  chartbar(integer, integer, integer, integer, integer, integer)

global predicates
  _bar(integer, integer, integer, integer, integer)
      -(i, l, i, i, i) language c

```



```

predicates
  process(char)
  repeat
    retractall
    drawbar(integer, integer, integer, integer)

goal
  graphics(1, 0, 0),
  asserta(barchart(0)),
  makewindow(1, 1, 1, "Bar", 0, 0, 20, 40),
  makewindow(2, 1, 1, "", 20, 0, 4, 40),
  repeat,
  write("Press any key to continue"), readchar(_),
  clearwindow,
  makewindow(1, 1, 1, "Bar", 0, 0, 20, 40),
  write("m. Make a bar Chart.\n"),
  write("s. Save your bar Chart.\n"),
  write("l. Load your bar Chart.\n"),
  write("d. Draw your bar Chart.\n"),
  write("e. End\nEnter Choice, "),
  readchar(Choice),
  clearwindow,
  shiftwindow(2),
  process(Choice),
  bios(16, reg(3, 0, 0, 0, 0, 0, 0, 0), _).

clauses
  process('m'),-
    barchart(Nu),
    retract(barchart(Nu)),
    NewNu = Nu+1,
    asserta(barchart(NewNu)),
    write("Enter the number of bars, "),
    readint(NuOfBars), clearwindow,
    write("Enter max value on Y scale, "),
    readint(Max), clearwindow,
    YScale = 140/Max,
    Width = 265/NuOfBars,
    Start = 305-Width,
    clearwindow,
    drawbar(NuOfBars, Width, Start, Yscale),
    fail.
  process('s'),-
    write("Enter file name of chart, "),
    readln(Name),
    concat(Name, ".CHT", FName), save(FName), fail.
  process('l'),
    retractall,
    write("Enter file name of chart, "), readln(Name),
    concat(Name, ".CHT", FName), consult(FName), fail.
  process('d'),
    barchart(Char),

```

```

        charibar(Char,XPosition,Ypos,Width,YBarSize,1),
        _bar(XPosition,Ypos,Width,YBarSize,1), fail.
process('e').

drawbar(0,_,_,_),-!.
drawbar(Nu,Width,XPosition,Yscale),
    XNuNew=Nu-1,clearwindow,
    write("Enter the value associated bar ",
        Nu," "),
    readln(Y),
    YBarSize=Y*YScale,
    Ypos=140 - YBarSize,
    _bar(XPosition,Ypos,Width,YBarSize,1),
    barchart(Char),
    assert(charibar(Char,XPosition,Ypos,
        Width,YBarSize,1)),
    XNewPos=XPosition-Width-5,
    drawbar(XNuNew,Width,XNewPos,Yscale).

repeat.
repeat;- repeat.
retractall;- retract(_),fail.
retractall.

```

编译 PBAR.PRO 生成 PBAR.OBJ

像例一那样，在连接 Turbo C 模块之前必须将 Turbo Prolog 主模块源文件编译成目标文件(.OBJ)。

连接 PBAR.OBJ 和模块 BAR.OBJ

利用下述连接命令，就把 PBAR.OBJ 和已经编译过的 Turbo C 目标模块 BAR.OBJ 连接起来。这个连接命令的组成是：

- Turbo Prolog 目标模块 INIT.OBJ 和 PBAR.OBJ;
- Turbo C 目标模块 BAR.OBJ;
- 符号表 PBAR.SYM 和输出文件 BARCHART.EXE(可执行文件);
- 库 PROLOG.LIB 和 CL.LIB.

连接命令是：

```
tlink init cpinit pbar bar pbar.sym, barchart,,prolog+cl
```

通过这四个例子，读者已经看到了如何把 Turbo Prolog 模块和 Turbo C 程序连接起来。对于有经验的 Turbo Prolog 程序人员来说，如果要想更多地了解 C 语言程序设计，可以参阅本书的第四、第五章。对于有经验的 C 程序人员来说，如果想要了解更多的 Turbo Prolog 知识，可以参考一些有关 Turbo Prolog 的书籍，如参考文献[11]和[12]。

第八章 Turbo C 高级程序设计技术

本章将介绍五个主要问题：首先将介绍存储模式，从极小模式到特大模式，包括什么是存储模式，如何选择存储模式，为什么要使用或不用某种存储模式。在第七章介绍 Turbo C 和 Turbo Prolog 语言混用时，已经使用了存储模式的概念。本章将介绍和汇编语言的混合使用。接着将介绍 Turbo C 低级程序设计的三个方面：直接插入汇编代码，伪变量和中断处理。然后将介绍浮点处理问题。最后介绍的两方面的内容是：Turbo C 的字符屏幕管理和图形处理功能。

8.1 存储模式

什么是存储模式，这里为什么要讨论它呢？要回答这个问题，首先要考察一下我们所用的计算机系统。其中央处理器（CPU）为 Intel iAPX 86 系列微处理器，可能是 8088，也可能是 8086，80186 或 80286。现在暂且认为它是 8086。

8.1.1 8086 寄存器

要弄清引进存储模式的概念的原因，还要从寄存器说起。图 8.1 列举了 8086 处理器的寄存器，并简单介绍了其各自功能。另外还有两个寄存器：IP（指令指针）和标志寄存器，因 Turbo C 不能访问之，故从略。

通用寄存器

AX	AH	AL	累加器（数学运算）
BX	BH	BL	基地址寄存器（变址）
CX	CH	CL	计数器（循环计数等）
DX	DH	DL	数据寄存器（保存数据）

段地址寄存器

CS		代码段指针
DS		数据段指针
SS		栈段指针
ES		附加段指针

图 8.1 8086 寄存器

专用寄存器

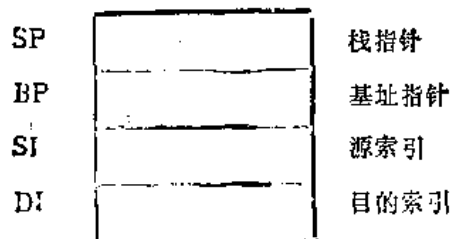


图8.1 8086寄存器(续)

8.1.1.1 通用寄存器

通用寄存器主要用于存放和处理数据。各通用寄存器均有一些特殊功能。例如，

- 许多数学运算只能用 AX 进行；
- BX 可用于存放远指针的位移部分；
- CX 可用于某些 8086 循环指令；
- 有些指令用 DX 保存数据。

有许多操作各寄存器均可使用；许多场合下，可自由换用寄存器。

8.1.1.2 段地址寄存器

段地址寄存器用于存放四个段的始地址。以下将谈到，段地址寄存器的16位值左移四位后（乘以16）才得到真正的20位段地址。

8.1.1.3 专用寄存器

8086 还有一些专用寄存器：

• SI 和 DI 寄存器可起许多通用寄存器的作用，此外还可用作变址寄存器。Turbo C 还用它们作寄存器变量。

• SP 指向当前栈顶，是相对于栈段的位移。

• BP 为辅助栈指针，用于取栈中参数。

基址寄存器（BP）在 C 函数实现时作为参数和自动变量区的基地址。参数相对于 BP 的位移是正的，大小随存储模式和函数入口时保存的寄存器个数而变。BP 总是指向保存的旧 BP 值。未定义变量的无参函数无需使用和保存 BP。

自动变量相对于 BP 的位移为负的，而第一个自动变量位移的绝对值最大。

8.1.2 内存分段及地址计算

Intel 8086 微处理器采用分段内存结构。其总地址空间为 1 兆字节。但在设计上同时只能直接访问 64K 字节。64K 内存区域称为一个段。分段内存结构由此得名。

那么，有多少不同的段呢？它们在什么地方？8086 又是如何知道它们的位置的？

• 8086 记录四个不同的段：代码、数据、栈和附加段。代码段存机器指令；数据段存数据信息；栈段当然是供栈使用；附加段通常存放附加数据。

• 8086 有四个 16 位段寄存器（每段各一），名为 CS，DS，SS 和 ES，分别指向代码段、数据段、栈段和附加段。

• 段可以位于内存中任何位置，至少是几乎所有位置。以后将会知道，段的首地址必须能被 16（十进制）整除。

那么, 8086 是如何使用段寄存器计算地址的呢?

8086 的完整地址由两个 16 位值、段地址和位移组成。假设数据段地址是 2F84 (十六进制), 即 DS 寄存器值, 若想知道数据段中位移为 0532 (十六进制) 的数据的真实地址, 如何计算呢?

地址可这样计算: 将段寄存器的值左移四位 (即一个十六进制位), 然后加上位移。所得 20 位结果即为该数据的真正地址。这一过程可表示为:

$$\begin{array}{rcl} \text{DS 寄存器 (移位后):} & 0010\ 1111\ 1000\ 0100\ 0000 & = 2\text{F}840 \\ \text{位移} & & \\ \text{地址} & \begin{array}{r} 0000\ 0101\ 0011\ 0010 = 00532 \\ \hline 0010\ 1111\ 1101\ 0111\ 0010 = 2\text{F}\text{D}72 \end{array} & \end{array}$$

段的开始地址均为 20 位数, 但段寄存器只能存 16 位数。所以这里总是假设其最低四位全为零。这表明, 在内存中每隔 16 个字节, 才有一地址可作为段开始地址, 该地址最后 4 位为零。

所以, 如果 DS 寄存器值为 2F84, 数据段实际上从 2F840 开始, 另外, 连续 16 个字节称为一小节, 所以段总是从小节边界开始。

地址的标准写法为段:位移。例如, 上述地址可写为 2F84:0532。注意, 由于位移可能重叠, 所给出的 (段:位移) 对不是唯一的。如下地址均指向同一内存单元:

0000:0123
0002:0103
0008:00A3
0010:0023
0012:0003

最后请注意, 段可能会重叠, 但不强求重叠。例如, 所有四个段 (CS, DS, SS, ES) 可有同一开始地址。这意味着整个程序所占空间不能超过 64K。这包括所有代码、数据和栈所需空间。

8.1.3 近指针、远指针和特大指针

指针和存储模式及 Turbo C 有何关系呢? 关系很大。存储模式的选择决定了代码和数据的默认指针类型。但也可不管所用存储模式, 显式说明一指针 (或一函数) 为某一指定类型。下面分别介绍三类指针: 近指针 (16 位)、远指针 (32 位) 及特大指针 (32 位)。

8.1.3.1 近指针

16 位近指针的地址计算依赖于某一段寄存器。例如, 函数指针的地址是将其 16 位值加以代码段 (CS) 寄存器移位后的值; 同理, 近数据指针是相对于数据段 (DS) 寄存器的位移。近指针使用方便, 因为算术运算 (如加法) 能不考虑段而进行。

8.1.3.2 远指针

32 位的远指针不仅包括段内位移, 也包括 (另一 16 位) 段地址, 段地址左移后和位移相加。使用远指针可访问多个代码段, 这就允许程序大于 64K。同样, 使用远数据指针可存取多于 64K 的数据。

当使用远数据指针时, 须注意指针操作上的几个潜在的问题。在讨论地址计算时, 谈到过同一地址可表示为许多不同的地址对 (段:位移)。例如, 远指针 (0000:0120), (0010:0020) 和 (0012:0000) 最终指向同一 20 位地址。但是, 如果有三个不同的远指针

变量 a、b、c，分别含有这三个值，则如下表达式均为假：

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

当用 >，>=，<，<= 运算比较两远指针时，也有同样问题。在这些场合，只有位移（作为 unsigned 类型）参加了比较。在 a、b、c 取上述值时，以下表达式均为真：

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

相等（==）和不等（!=）运算将 32 位值作为 unsigned long 处理，而不是作为一个完整的内存地址。而比较运算（<=，>=，<和>）只使用位移。

因为 == 和 != 运算需要 32 位数据，所以可和空指针（0000：0000）比较。如相等检查只使用位移，则任何位移为 0000 的指针将被认为和空指针相等，这并非我们所要求的。

还需注意，如果加一数到远指针，仅仅改变位移。如加的数足够大，使位移超过 FFFF（最大可能值），指针又回到段的开始位置。例如，5031：FFFF 加 1 的结果为 5031：0000，而不是（6031：0000）。同理 5031：0000 减 1 为 5031：FFFF，而不是（5030：FFFF）。

如果使用指针比较，最安全的方法是使用近指针，或使用下面介绍的特大指针。

8.1.3.3 特大指针

特大指针和远指针一样，为 32 位长，含有段地址和位移。但和远指针不同的是，特大指针是规格化的，可以解决远指针的问题。

所谓规格化指针就是一个 32 位指针，其段地址部分为最大可能值。每隔 16 个字节就可有一地址作为段开始地址，所以这表明指针的位移值在 0 至 15 之间（十六进制的 0 到 F）。

指针的规格化只需将其转化为 20 位地址，取其右边 4 位为位移，左边 16 位为段地址。例如，指针 2F84：0532 的绝对地址为 2FD72，规格化后得 2FD7：0002。下面在给出几个指针及其规格化后的等价形式。

0000：0123	0012：0003
0040：0056	0045：0006
500D：9407	594D：0007
7418：D03F	811B：000F

现在已经知道特大指针总是规格化的，但为什么要这样呢？因为这意味着对给定内存地址，只有唯一的特大指针。所以 == 和 != 对特大指针回送结果正确。

此外，>，>=，<和<= 运算对特大指针是对全部 32 位比较的。规格化保证了其结果的正确无误。

由于规格化的缘故，特大指针的位移部分每隔 16 个值重复一次，而段地址将作相应修改。例如，将 811B：000F 加 1，结果为 811C：0000；同样，811C：0000 减 1，结果为 811B：000F。特大指针的这一特性使得可处理大于 64K 的数据结构。

使用特大指针是要付出额外开销的。特大指针处理需调用专用子程序。所以特大指针处理比远指针或近指针处理要慢。

8.1.4 Turbo C 的六种存储模式

Turbo C 为了帮助程序员减少额外开销, 提供了六种存储模式: 极小模式、小模式、中模式、紧凑模式、大模式和特大模式。模式的选择取决于程序员的需求。下面简要介绍一下各种模式。

(1) 极小模式: 可以想象, 这是最小的存储模式。各段寄存器 (CS, DS, SS, ES) 均取相同的地址值 (即 $CS = DS = SS = ES$)。所以代码、数据和数组的总空间为 64K。极小模式全部用近指针。当内存奇缺时使用该模式。极小模式的程序可转换成 .COM 格式。

(2) 小模式: 其代码和数据段不同且不重叠, 故可有 64K 代码, 64K 静态数据。栈段 (SS)、附加段 (ES) 和数据段 (DS) 取相同值 (即 $DS = SS = ES$)。总使用近指针。适用于一般应用问题。

(3) 中模式: 代码用远指针, 但数据不用远指针。所以静态数据最多为 64K, 但代码可多达 1 MB。适用于内存数据量小的大程序。

(4) 紧凑模式: 和中模式相反, 数据用远指针而代码不用。代码最多为 64K, 而数据可达 1 MB。适用于处理大量数据小程序。

(5) 大模式: 代码和数据均用远指针, 均可达 1 MB。只有非常大的应用问题需用此模式。

(6) 特大模式: 代码和和数据均用远指针。Turbo C 一般限制静态数据为 64K。特大模式可打破这一限制, 使静态数据多于 64K。

图 8.2—8.7 展示了 Turbo C 各种存储模式下 8086 内存分配情况。

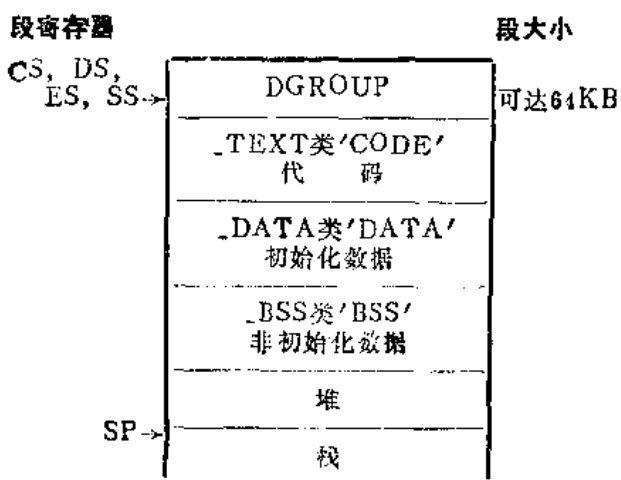


图 8.2 极小模式内存分配

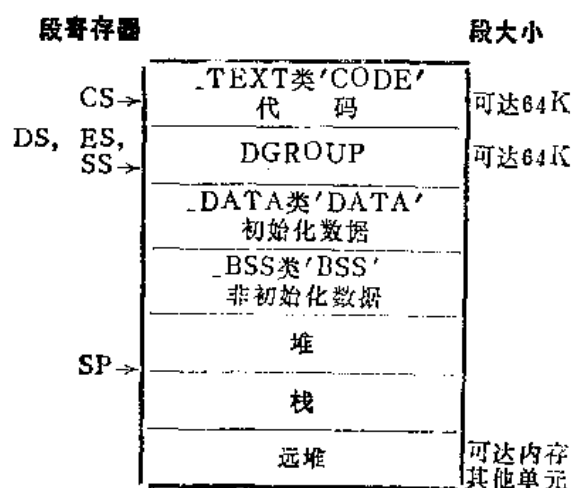


图 8.3 小模式内存分配

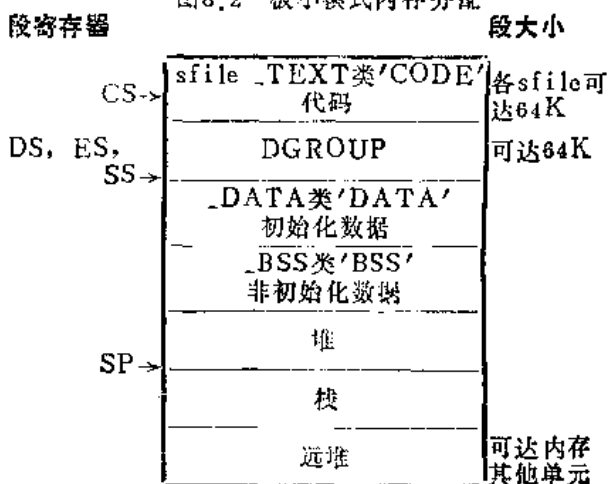


图 8.4 中模式内存分配

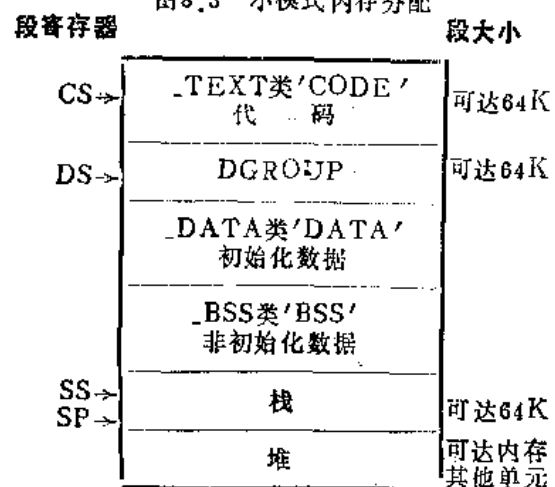


图 8.5 紧凑模式内存分配

段寄存器

段大小

段寄存器

段大小

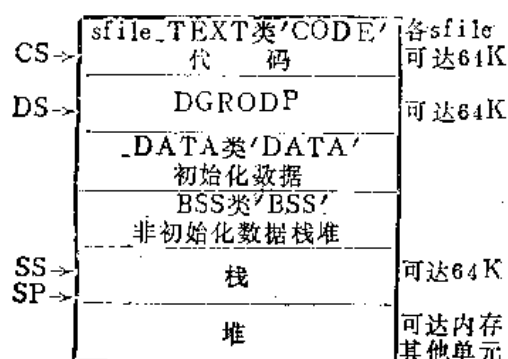


图8.6 大模式内存分配

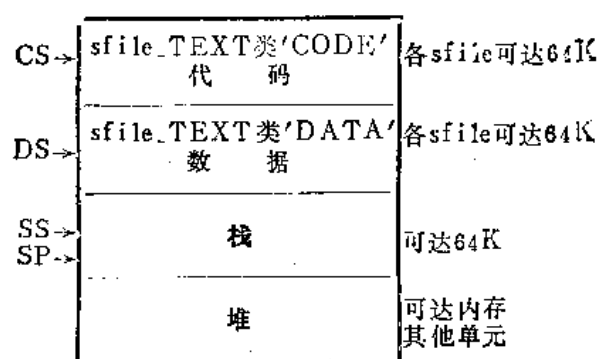


图8.7 特大模式内存分配

表8.1总结比较了各种存储模式的特性，其中存储模式是按其大(1MB)小(64K)组织的，这构成了表8.1的行和列。例如：极小模式、小模式和紧凑模式称为小代码模式，因其默认代码指针为近指针；类似地，紧凑模式、大模式和特大模式称为大数据模式，因其默认数据指针为远指针。对特大模式也是这样，即默认情况下指针为远指针，而非特大指针。如需特大数据指针，需显式定义其为特大指针。

表8.1 存储模式

数据大小	代码大小	
	64K	1MB
64K	极小(代码数据重叠, 总长64K) 小(代码数据不重叠, 总长128K)	中模式(小数据, 大代码)
1MB	紧凑模式(大数据, 小代码)	大模式(大数据, 大代码) 特大模式(同大模式, 但静态数据大于64K)

请注意，当编译一模块(带有一些子程序的一源文件)时，所产生的代码不能超过64K，因为它将装入同一代码段。即使用大代码模式(中模式、大模式或特大模式)也是这样。如果模块太大，无法装入一64K代码段，可将其分为几个源代码文件，分别编译之，然后将其连接在一起。同样，虽然特大模式允许静态数据总数超过64K，各模块数据仍然须少于64K。

8.1.5 混合模式程序设计：地址修饰符

Turbo C引入了C语言标准(《K&R》及ANSI标准)中没有的七个新关键字：near, far, huge, _cs, _ds, _es和_ss(参见第十二章)。这些关键字可作为指针(有时也可作为函数)的修饰符使用，但有一些限制和注意事项。

在Turbo C中可用near, far或huge修饰函数或指针。本章前面部分已介绍了数据近指针，远指针和特大指针。near函数可采用近调用激活并用近返回退出。类似地，far函数可采用远调用激活并用远返回退出。huge函数和far函数类似，不同的只是huge函数能将DS置成新值，而far函数则不能。

还有四个特殊的近数据指针：_cs, _ds, _ss和_es。它们均为16位指针，分别对

应于各段寄存器。例如，若定义如下指针 p：

```
char    _ss *p;
```

p 将为相对于栈段的 16 位移位量。

根据所选存储模式，函数和指针在默认情况下或为远指针或为近指针。如函数或指针为近指针，它们将自动和 CS 或 DS 寄存器相联系。

表 8.2 表明了这种联系情况。其中指针大小由其访问空间为 64K（近指针）或 1 M（远指针）决定。

表 8.2 指 针 结 果

存储模式	函数指针	数据指针
极小	近, _cs	近, _ds
小	近, _cs	近, _ds
紧凑	远	近, _ds
中	近, _cs	远
大	远	远
特大	远	远

8.1.5.1 说明远函数和近函数

有时，读者也许想要修改如表 8.1 所示某存储模式下的默认函数种类。例如，在大模式程序中有如下递归函数：

```
double power(double x,int exp)
{
    if ( exp <= 0 )
        return(0);
    else
        return(x * power(x,exp - 1));
}
```

每次 power 自己调用自己时，必须使用远调用，这将用较多栈空间和时间。如将 power 说明成近函数，可将所有调用改为近调用以减少一些开销：

```
double near power (double x,int exp)
```

这将保证 power 只能在其当前所在代码段被调用，且其所有调用均为近调用。

这意味着如在大代码模式（中模式、大模式或特大模式）下，只能在 power 本身所在模块调用 power。其他模块有其自身的代码段。故不能调用另一模块的 near 函数。而且，near 函数必须在第一次调用前被说明或定义，否则编译程序将无法知道需产生近调用。

相反，说明一 far 函数表明需产生远调用。在小代码模式中，far 函数需在首次使用前定义或说明，以保证产生远调用。

回头再看一下 power 这个例子。最好将 power 说明为静态函数，因其只能在本模块内部被调用。如果说明为静态函数，其名在模块外将不可见。再由于 power 总接收固定个数的参数，还可将其说明为 pascal 以作进一步优化；

```
static double near pascal power (double x, int exp)
```

关于修饰符 pascal 的含义请参阅第十二章中有关的描述。

8.1.5.2 说明近指针、远指针或特大指针

至此已经知道为什么要将一函数说明为和程序其他部分具有不同的模式。对指针为什么也要这么做呢？其理由是一样的：或是为了减少额外开销（当默认为 far 时说明 near），或是为了访问外段内存（当默认为 near 时说明 far 或 huge）。

当然，将函数或指针说明为非默认种类有着潜在的危险。例如，设有如下小程序：

```
void myputs (s)
char *s;
{
    int i;
    for (i=0; s[i] != 0; i++) putc (s[i]);
}
main()
{
    char near *mystr;
    mystr = "Hello, world\n";
    myputs (mystr);
}
```

该程序能正常运行。而且事实上，mystr 函数的 near 说明是多余的，因为所有代码、数据指针均为近指针。

如果用紧凑存储模式（或大模式或特大模式）编译之，结果如何呢？主函数中的 mystr 指针仍然是近的，而 myputs 函数中的 s 现在是远的，这是默认情况。也就是说，myputs 在构造远指针时将从栈中弹出两个字，得到的显然不会是 mystr 的地址。

要解决这一问题，可用现代 C 语言风格定义 myputs：

```
void myputs (char*s)
{
    /*myputs函数体*/
}
```

当 Turbo C 翻译这一程序时，翻译程序知道 myputs 参数为字符指针，且因为是在大模式下编译，指针须为远指针，Turbo C 将把数据寄存器（DS）的值和 mystr 的 16 位位移一同压栈，形成一远指针。

在相反情况下，即说明 myputs 参数为远指针而在小数据模式下编译，情况又是怎样呢？同样，不同函数模式会引起问题，因为主函数会将段地址和位移均压入栈中，而 myputs 只期望得到位移。而用原型风格的函数定义时，主函数只将位移压栈。

从中可得到这样的经验，如需显式将指针说明为远指针或近指针，则必须在所有使用它的函数中使用函数原型。

8.1.5.3 构造适当的说明符

说明符是 C 语言的一种说明函数、变量、指针和数据类型的语句。C 语言允许非常复杂的说明符。本节将介绍一些说明符，使读者在设计和阅读说明符方面能得到一些训练，同时

还介绍了需避开的一些陷阱。

传统的C语言程序设计方法通常是即席构造整个说明符，需要时还将用嵌套定义，不幸的是，这种方法使程序难写难读。例如，在小存储模式（小代码，小数据）下编译如下说明符：

<code>int f1();</code>	回送整数的函数
<code>int *p1;</code>	整型指针
<code>int *f2();</code>	回送整型指针的函数
<code>int far *p2;</code>	整型远指针
<code>int far *f3();</code>	回送整型远指针的近函数
<code>int *far f4();</code>	回送整型近指针的远函数
<code>int (*fp1)(int);</code>	接收整型参数，回送整型的函数的指针
<code>int (*fp2)(int *ip);</code>	接收整型指针，回送整型的函数的指针
<code>int (far *list[5])</code> <code>(int far *ip);</code>	五个函数远指针的数组，其中函数接受整型远指针，回送整型
<code>int (far *gopher(int far *fp[5])</code> <code>(int far *ip))</code> <code>(int far *ip);</code>	近函数，它接受五个远指针的数组为参数，回送整型远指针。其参数指向接受整型远指针的整型函数。

以上均为合法说明符，只是一个比一个难懂。但如能明智地使用typedef机制，说明符的可读性可望改善。以下是用typedef定义改写的说明符：

<code>int f1();</code>	回送整型的函数
<code>typedef int *intptr;</code>	
<code>intptr p1;</code>	整型指针
<code>intptr f2();</code>	回送整型指针的函数
<code>typedef int far *farptr;</code>	
<code>farptr p2;</code>	整型远指针
<code>farptr f3();</code>	回送整型远指针的近函数
<code>intptr far f4();</code>	回送整型近指针的远函数
<code>typedef int (*fncptr1)(int);</code>	
<code>fncptr1 fp1;</code>	接受整型参数，回送整型的函数的指针
<code>typedef int (*fncptr2)(intptr);</code>	
<code>fncptr2 fp2;</code>	接受整型指针，回送整型的函数的指针
<code>typedef int (far *ffptr)(farptr);</code>	
<code>ffptr fp3;</code>	接受整型远指针，回送整型的函数的远指针
<code>typedef ffptr ffplist[5];</code>	
<code>ffplist list;</code>	五个函数远指针的数组，其中函数接受整型远指针，回送整型
<code>ffptr gopher(ffplist);</code>	近函数，它接受五个远指针的数组为参数，回送整型远指针。其参数指向接受整型远指针的整型函数。

可以看出, `gopher` 函数用 `typedef` 机制的说明和原有说明在可读性和整洁性上有很大差别。如能巧妙地使用 `typedef` 机制及函数原型, 则将大大方便程序的编写、调试和维护。

8.1.5.4 库文件的连接

Turbo C 对六种存储模式各提供一个标准库例行程序版本。在集成开发环境 (TC) 运行时, Turbo C 可根据所选存储模式, 以适当的次序连接适当的库; 同样, 编译程序的 TCC 版本当被告知作自动连接时, Turbo C 能正确连接。

但如直接使用 TLINK (即 Turbo C 连接程序), 用户必须指定要连接的库。如果只需使用一部分存储模式, 则只需将相应模式所需文件拷贝至工作盘或硬盘。各模式所需文件如表 8.3 所示。

表 8.3 六种存储模式及其对应的库文件

存 储 模 式	库 文 件
极小模式	C0T.OBJ, MATHS.LIB, CS.LIB
小模式	C0S.OBJ, MATHS.LIB, CS.LIB
紧凑模式	C0C.OBJ, MATHC.LIB, CC.LIB
中模式	C0M.OBJ, MATHM.LIB, CM.LIB
大模式	C0L.OBJ, MATHL.LIB, CL.LIB
特大模式	C0H.OBJ, MATHH.LIB, CH.LIB

请注意极小模式和小模式使用同样的库, 但启动文件不同 (C0T.OBJ 和 C0S.OBJ)。如系统拥有 8087 数学协处理器, 则需文件 FP87.LIB; 如需仿真 8087, 则需文件 EMU.LIB。

下面给出几个 TLINK 命令的例子:

```
tlink c0m a b c, prog, mprog, fp87 mathm cm
```

```
tlink c0c d e f, plan, emu mathc cc
```

第一个命令行使用中模式库和 8087 支撑库, 产生的可执行程序名为 PROG.EXE。第二条命令行产生的程序名为 PLAN.EXE, 使用紧凑模式, 若运行时没有协处理器, 将利用仿真 8087 的浮点运算例行子程序。

代码文件和库在命令行中的次序是非常重要的。C 语言的启动模块 (C0X.OBJ) 必须在代码文件列的第一个位置。包含的库文件列必须遵照以下次序:

- (1) 用户自用库 (如果有的话);
- (2) FP87.LIB 或 EMU.LIB, 后接 MATH_x.LIB (如使用浮点运算);
- (3) C_x.LIB (标准 Turbo C 运行时刻库) (C0_x, MATH_x, C_x 中的 _x 表示代表相应存储模式的首字母: t, s, m, c, l 和 h)。

8.1.5.5 混合模式的连接

如果编译的一模块使用小存储模式, 另一模块使用大模式, 而后又想将两者连接在一起, 情况将会怎样呢?

这两文件确能很好地连接在一起, 但会遇到和 8.1.5.1 节“说明远函数和近函数”中遇到的同样问题。如一小模式模块中的函数调用了一大模式模块中的函数, 它将使用近调用指令, 这可能引起麻烦。而且, 在指针方面, 将会遇到 8.1.5.2 节“说明近指针、

袁

远指针或特大指针”中所遇到的同样的问题，因为小模式模块中的函数将期望接收和传送近指针，而大模式模块中的函数将期望接收远指针。

解决方法同样是使用函数原型。假设将函数 `myputs` 放入其所在模块，并用大存储模式编译之。就必须构造一嵌入文件 `MYPUTS.H`（或以 `.H` 为扩展名的其他文件），其中将包括如下函数原型：

```
void far myputs (char far *s);
```

然后，若将 `main` 函数放在其相应模块中（名为 `MYMAIN.C`），则需这样编写：

```
#include <stdio.h>
#include "myputs.h"
main ( )
{
    char near *mystr;
    mystr="Hello, world\n";
    myputs ( mystr );
}
```

当编译该程序时，Turbo C 从 `MYPUTS.H` 中读入函数原型，知道了 `myputs` 是一个期望远指针的远函数。则即使在小存储模式下，也能据此生成适当的调用代码。

除此之外，库例行子程序连接时，最好的方法是使用某个大存储模式库，并将所有指针和函数都说明为 `far`。这可以这样做，拷贝一些常用嵌入文件（如 `STDIO.H`），将其改成适当的名（如 `FSTDIO.H`），然后编辑备份中的各函数原型，使其显式说明为 `far` 方式。例如：

```
int far cdecl printf (char far *format, ...);
```

按此办理，不但函数调用将用远调用，而且指针也将用远指针方式传递。最后，修改程序，使其使用新的嵌入文件：

```
#include <fstdio.h>
main ( )
{
    char near *mystr;
    mystr="Hello, world\n";
    printf ( mystr );
}
```

用 TCC 编译该程序，然后指定一大存储模式库用 TLINK 作连接。模式混用较为复杂，但能办到。如使用不当，会遇到难查的错误。

8.2 多语言混合程序设计：和其他语言接口

Turbo C 为在 C 语言程序中调用用其他语言书写的子程序，以及为其他语言程序调用 C 语言子程序提供了方便。第七章，我们已经介绍了 Turbo C 与 Turbo Prolog 的接口，本节将介绍同汇编等语言接口的有关问题。Turbo C 和其他语言接口是相当容易的。

以下先介绍参数传递的两种主要顺序，然后介绍如何编制汇编语言模块。

8.2.1 C语言和Pascal语言的参数传递顺序

Turbo C支持两种函数参数传递方法：标准C语言方法和Pascal语言方法。现分别叙述之。

8.2.1.1 C语言参数传递方法

假设有如下函数原型：

```
void funca(int p1, int p2, int p3);
```

Turbo C默认使用C语言参数传递顺序，或称C调用约定。当函数funca被调用时，参数以从右到左的次序压栈(p3, p2, p1)，后接返回地址。如有下列调用：

```
main ( )
{
    int i, j;
    long k;
    ...
    i = 5; j = 7; k = 0x1407AA;
    funca(i, j, k);
    ...
}
```

栈的内容如下(在返回地址压栈前)：

```
sp+06:0014
sp+04:07AA      k = p3
sp+02:0007      j = p2
sp      :0005      i = p1
```

请记住，8086的栈从内存高区向内存低区生长，所以i在当前栈顶。

被调用的子程序无需知道(实际上也无法知道)栈上所压参数的个数。它只需假定所需参数均在栈中。

另外很重要的一点是，被调用子程序无需将参数弹出栈。因为这是由调用者做的。例如，上述主函数编译后产生的汇编代码为：

```
mov word ptr [bp-8], 5      ; 置 i = 5
mov word ptr [bp-6], 7      ; 置 j = 7
mov word ptr [bp-2], 0014h   ; 置 k = 0x1407AA
mov word ptr [bp-4], 07AAh
push word ptr [bp-2]         ; 压入 k 的高字
push word ptr [bp-4]         ; 压入 k 的低字
push word ptr [bp-6]         ; 压入 j
push word ptr [bp-8]         ; 压入 i
call near ptr funca          ; 调用 funca (压入地址)
add sp, 8                    ; 调整栈
```

请注意最后一条指令：add sp, 8。编译程序在这时知道栈上压了多少个参数，也知道调用funca时返回地址已压在栈上，且它已由funca结束时的ret返回指令弹出。

8.2.1.2 Pascal语言参数传递方法

另一种参数传递方法是 Pascal 语言的参数传递方法。但这并不意味着可以从 Turbo C 中调用 Turbo Pascal 函数, 实际上用户不能这样做。

Pascal 调用方法是将参数从左至右压栈, 若定义 funca 如下:

```
void pascal funca (int p1, int p2, int p3);
```

则调用该函数时, 参数将从左至右压栈(p1, p2, p3), 然后压入返回地址。所以, 若有如下调用:

```
main ( )
{
    int i, j;
    long k;
    ...
    i = 5; j = 7; k = 0x1407AA;
    funca (i, j, k);
    ...
}
```

在返回地址压入前, 栈的内容如下:

```
sp+06: 00 0 5   i = p1
sp+04: 00 0 7   j = p2
sp+02: 00 1 4
sp      : 07AA   k = p3
```

那么, 这两种调用次序的差别是什么呢? 除了参数压入的次序不同外, Pascal 调用次序假设被调用的子程序 (funca) 知道向其传送的参数个数, 以便对栈作适当调整。换言之, 调用 (funca) 时的汇编程序呈如下形式:

```
push word ptr [bp-8]      ; 压入 i
push word ptr [bp-6]      ; 压入 j
push word ptr [bp-2]      ; 压入 k 的高字
push word ptr [bp-4]      ; 压入 k 的低字
call near ptr funca       ; 调用 funca (压入地址)
```

请注意, 其中调用指令后无 add sp, 8 指令。而 funca 须在其结束处使用 ret 8 指令清除栈中内容, 然后再返回。

Turbo C 中所有函数, 默认使用 C 语言参数调用方法。唯一的例外是使用 -p 编译选择项, 这时所有函数采用 Pascal 调用方法。在这种情况下, 可用 cdecl 修饰符强行使用 C 语言参数传递方法, 如:

```
void cdecl funca (int p1, int p2, int p3);
```

可避开 -p 编译选择项。

到底为什么要使用 Pascal 调用约定呢? 主要有以下三个理由:

- (1) 可调用使用这种参数传递方法的汇编语言子程序;
- (2) 可调用其他语言编写的子程序;
- (3) 这样产生的调用代码较短, 因为这时无需调用后清栈。

那么, 使用 Pascal 调用方法会产生什么问题呢?

首先, 这种调用方法的坚定性不及 C 语言调用方法。不可以象 C 语言调用方法中那样传

递可变个数参数，因为它要求被调用的子程序知道传给它的参数的个数，以便对栈作相应处理。传递的参数太少或太多几乎必定会引起严重的问题。而在C语言调用方法下这样做一般没有不良作用（除可能得到错误的结果）。

其次，如使用 -p 编译选择项，一定要引用所有被调用的标准 C 函数相应的嵌入文件。否则，Turbo C 将对这些函数使用 Pascal 调用约定。程序就注定要出错，因为参数的次序将出错，且栈未清除。

嵌入文件中将每个函数定义为 cdecl 方式，所以如引用了这些嵌入文件，编译程序将使用 C 语言调用方法。但 C 标识符带有下划线符，而 Pascal 标识符不带下划线符。除非使用无下划线符选择项，且在不区分大小写情况下连接，否则会产生大量连接错误。

从这里可得到结论：如需在 Turbo C 程序中使用 Pascal 调用约定，请尽量使用函数原型，而其中函数都须显式说明成 cdecl 或 Pascal。这时使用“需用原型”警告选择项很有用，可保证每个函数都有函数原型。

8.2.2 汇编语言接口

至此已知道各种调用方法的含义。被调用的子程序将做什么呢？这里先看一下如何编写 Turbo C 调用的汇编语言子程序。

本节将假设读者知道如何编写 8086 汇编语言子程序，知道如何定义段、数据常量等。

8.2.2.1 从 Turbo C 调用汇编程序

C 语言程序调用的汇编语言程序必须编写成模块，而且必须遵守某些约定：

- (1) 保证连接程序能得到必要的信息；
- (2) 保证文件格式符合 C 语言程序所用的存储模式。

汇编程序的一般格式如表 8.4 所示。

表 8.4 汇编程序的一般格式

标识符	名	文 件 名
<text>	SEGMENT ASSUME <.....代码段.....>	BYTE PUBLIC 'CODE' CS: <text>, DS: <dseg>
<text>	ENDS	
<dseg>	GROUP	_DATA, _BSS
<data>	SEGMENT <.....初始化数据段.....>	WORD PUBLIC 'DATA'
<data>	ENDS	
_BSS	SEGMENT <.....非初始化数据段.....>	WORD PUBLIC 'BSS'
_BSS	ENDS	
	END	

表 8.4 中 <text>、<data> 和 <dseg> 标识符根据所选存储模式，可换成相应符号。表

8.5 列举了各种存储模式下的情况，其中 filename 为模块名；它在 NAME 指示和标识符替换中必须一致。

注意在特大存储模式中，无 -BSS 段，GROUP 定义完全可不用。一般，-BSS 是任选项，只在需要用时才需定义。

建立汇编语言样板的最佳方案是将一空程序编译成 .ASM 程序（使用 TCC 的 -s 编译选择项），并查看产生的汇编代码。

表8.5 标识符替换及存储模式

模 式	标 识 符 替 换	代 码 和 数 据 指 针
极小，小	<code><code> = _TEXT</code> <code><data> = _DATA</code> <code><dseg> = DGROUP</code>	代码: DW _TEXT, xxx 数据: DW DGROUP, xxx
紧凑	<code><code> = _TEXT</code> <code><data> = _DATA</code> <code><dseg> = DGROUP</code>	代码: DW _TEXT, xxx 数据: DD DGROUP, xxx
中	<code><code> = filename_TEXT</code> <code><data> = _DATA</code> <code><dseg> = DGROUP</code>	代码: DD xxx 数据: DW DGROUP, xxx
大	<code><code> = filename_TEXT</code> <code><data> = _DATA</code> <code><dseg> = DGROUP</code>	代码: DD xxx 数据: DD DGROUP, xxx
特大	<code><code> = filename_TEXT</code> <code><data> = filename_DATA</code> <code><dseg> = filename_DATA</code>	代码: DD xxx 数据: DD xxx

8.2.2.2 定义数据常量和变量

存储模式也影响到如何定义作为代码、数据指针的数据常量。表 8.5 给出了这些指针的格式，其中 xxx 是指向的地址。

注意有些定义使用 DW（定义字），而另一些则使用 DD（定义双字），以表明产生的指针的大小。数值和文本常量可按通常方法定义。

变量的定义方法当然和常量一样。若想说明非初始化变量，可在 -BSS 段进行说明，并在通常放值的位置写上问号（？）。

8.2.2.3 定义全局和外部标识符

假如建立了一模块，除非相应的 Turbo C 程序知道它能调用什么函数，访问什么变量，这样的模块用处不大。同样，可能也会要求从汇编语言程序中调用 Turbo C 函数，或引用 Turbo C 程序中说明的变量。

进行这些调用和访问时，需对 Turbo C 编译程序和连接程序有所了解。当定义一外部

标识符时，编译程序自动在其首部加上下线符（`_`），然后再将其存入目标模块。所以在C语言程序调用的汇编语言模块中，所有标识符前需加上下线符。Pascal标识符的处理方法和C不同，它们是全大写的，且不冠以下线符。

C标识符的下线符（`_`）是任选的，但默认情况有下线符。可用`-u-`命令行选择项去掉下线符。但使用标准C库时，除非重构整个库，否则会遇到麻烦（如果这样做，尚需运行库的源代码）。

如果汇编代码源文件访问了C标识符（数据或函数），这些标识符必须以下线符开始。

Microsoft汇编程序（MASM）对大小写并不敏感。亦即，当汇编一程序时，所有的标识符仅以大写保存。MASM的`/mx`开关将使之对外部量大小写敏感。Turbo C连接程序对`extern`标识符也作同样处理，所以标识符能较好匹配。读者可能已注意到，在这里的例子中，关键字和指示字是大写的，而其他标识符和操作符是小写的；这种风格和MASM参考手册中的风格一致。读者编程序可自由选择大写、小写或大小写混用。

为使标识符（子程序和变量名）在汇编模块外部可见，须将它们定义为PUBLIC。例如，如需写含整型函数`max`和`min`，整型变量`MAXINT`，`lastmax`和`lastmin`的模块，需在代码段中加入语句：

```
PUBLIC max, min
```

且在数据段加入

```
PUBLIC _MAXINT, _lastmax, _lastmin
_MAXINT DW 32767
_lastmin DW 0
_lastmax DW 0
```

8.2.3 从汇编语言调用 Turbo C

使用和上节类似的方法，可用EXTRN语句从汇编语言模块中访问Turbo C程序中的函数和变量。

8.2.3.1 函数引用

为了能从汇编语言子程序中调用C函数，需在其中写上语句：

```
EXTRN <fname> : <fdist>
```

其中<fname>为函数名。<fdist>为`near`或为`far`。这取决于相应的C函数为近函数还是远函数。如<fdist>为`near`，则EXTRN需在模块代码段内；否则，EXTRN需在所有段的外部。故代码段可有如下语句：

```
EXTRN _mycfunc1, near, _mycfunc2, far
```

这样即可从汇编语言子程序中调用`mycfunc1`和`mycfunc2`。

8.2.3.2 引用数据

为引用数据，需在代码段中加入适当的EXTRN语句。格式如下：

```
EXTRN <vname> : <size>
```

其中<vname>为变量名，<size>表示变量大小，它可取如下值：

- BYTE (1字节)
- WORD (2字节)
- DWORD (4字节)

- QWORD (8字节)
- TBYTE (10字节)

对数组来说, <size> 为其元素大小; 结构的 <size> 为其最常使用的大小。

例如, 若C语言程序有如下全程变量:

```
int    i, jarray[10];
char   ch;
long   result;
```

可用如下语句, 使其在汇编模块中可见:

```
EXTRN _i: WORD, _jarray: WORD, _ch: BYTE, _result: DWORD
```

最后请注意, 如使用特大存储模式。EXTRN语句必须出现在所有段外部。子程序和变量均须这样。

8.2.4 定义汇编语言子程序

已经知道了如何设置各种引用关系, 现在考虑如何编写汇编语言函数。这里, 需认真考虑的是: 参数传递、返回值及寄存器的使用约定。

假设要编写的函数为 min, 其C语言的函数原型为:

```
int extern min(int v1, int v2);
```

该函数返回两个参数的最小值。其大体格式为:

```
        PUBLIC      _min
_min    PROC        near
        :
        :
```

其中假设 min 为近函数; 如其为远函数, 则需用 far 代替 near。注意 min 之前加有下划线。这样 Turbo C 连接程序能正确处理量的引用。

8.2.4.1 参数传递

首先需考虑的是使用哪种参数传递方法; 除非有合适的理由, 我们可用C语言约定而不用Pascal语言约定。这意味着, 当 min 被调用时, 栈的内容如下:

```
sp+04: v2
sp+02: v1
sp    : 返回地址
```

如要不退栈而取得参数, 则需保存基指针(BP), 将栈指针(SP)送给基指针, 然后直接用参数下标取得值。注意, 如将BP压栈, 参数的相对位移将增加2, 这是因为栈上又多了2个字节。

8.2.4.2 处理返回值

函数 min 返回一整数。返回值的位置是这样规定的: 16位值(双字节)(char, short, int, enum 和近指针), 使用 AX 寄存器; 32位值(4字节)(包括 far 和 huge 指针), 还需使用 DX 寄存器。其中高字(指针的段地址)在 DX 中, 而低字在 AX 寄存器中。

float 和 double 类型值回送在 8087 栈顶(TOS)寄存器中, 即 ST(0); 如使用 8087 仿真程序, 则回送在仿真程序的 TOS 寄存器中。

结构值的回送方法是将其放入静态数据区，然后返回其指针（小数据模式在AX中，大数据模式在DX:AX中）。

调用函数需将其拷贝至所需位置。1字节或2字节的结构返回在AX寄存器中（和通常int一样），而4字节结构返回在AX和DX中。

min函数中只有16位值，故结果在AX中回送。整个代码为：

```
        PUBLIC  _min
_min    PROC    near
        push    bp                ; 将 bp 保存在栈上
        mov     bp, sp            ; 将 sp 复制到 bp
        mov     ax, [bp+4]        ; 将 v1 送入 ax
        cmp     ax, [bp+6]        ; 和 v2 比较
        jle     exit              ; 如 v1 > v2
        mov     ax, [bp+6]        ; 则 v2 装入 ax
exit:    pop     bp                ; 恢复 bp
        ret                     ; 返回到 C
_min    ENDP
```

如将 min 说明为远函数，情况会有所不同。最大的差别在于调用时的栈为：

```
sp+06:    v2
sp+04:    v1
sp+02:    返回段
sp        : 返回位移
```

也就是说参数在栈中位置增加了2，因为栈上多压了两个字节（返回地址的段地址）。这时的 min 函数为：

```
        PUBLIC  _min
_min    PROC    far
        push    bp                ; 将 bp 保存在栈上
        mov     bp, sp            ; 将 sp 复制到 bp
        mov     ax, [bp+6]        ; 将 v1 送入 ax
        cmp     ax, [bp+8]        ; 和 v2 比较
        jle     exit              ; 如 v1 > v2
        mov     ax, [bp+6]        ; 则 v2 装入 ax
exit:    pop     bp                ; 恢复 bp
        ret                     ; 返回到 C
_min    ENDP
```

注意 v1, v2 的位移都加了2，以反映栈中增加的内容。

如果由于某种原因，需说明 min 为 Pascal 方式的函数，情况又怎么样呢？调用时的栈将为（假设 min 仍为近函数）：

```
sp+04:    v1
sp+02:    v2
sp        : 返回地址
```

除此之外, 标识符也需符合 Pascal 约定, 即大写和无下划线符。

除了 v1 和 v2 的位置交换了, Pascal 参数传递方式还要求函数 min 返回前清栈, 这可在 ret 指令中指明需从栈中弹出的字节数。本例中, 需弹出 v1, v2 所占的 4 个字节(返回地址由 ret 指令自动弹出)。修改后的子程序为:

```

        PUBLIC MIN
MIN PROC     near                , Pascal 版本
        push     bp              , 将 bp 保存在栈上
        mov      bp, sp         , 将 sp 复制到 bp
        mov      ax, [bp+6]      , 将 v1 送入 ax
        cmp      ax, [bp+4]      , 和 v2 比较
        jle      exit           , 如 v1 > v2
        mov      ax, [bp+4]      , 则 v2 装入 ax
exit:     pop     bp             , 恢复 bp
        ret      4              , 调整栈且返回到 C

```

MIN ENDP

下面再举一个例子, 看看为什么要使用 C 语言参数传递方法。假设将 min 重新定义为:

```
int extern min (int count, int v1, int v2, ...);
```

这时, min 可接受任意多个参数, 并返回其最小值。但 min 无法自己知道传给它的参数个数。故需将第一个参数用作计数, 表示后面参数的个数。

例如, 可这样调用 min:

```
i = min ( 5, j, limit, indx, locount, 0 );
```

假设 i, j, limit, indx 及 locount 均为 int 类型 (或相容类型)。调用时栈为:

```

sp+08:      (其他参数)
sp+06:      v2
sp+04:      v1
sp+02:      count
sp   :      返回地址

```

修改后的 min 代码为:

```

        PUBLIC _min
_min PROC   near
        push     bp              , 将 bp 压栈
        mov      bp, sp         , 将 sp 复制到 bp
        mov      ax, 0          , 置 ax 为 0
        mov      cx, [bp+4]      , 将 count 送入 cx
        cmp      cx, ax         , 和 0 比较
        jle      exit           , 若小于等于 0 则退出
        mov      ax, [bp+4]      , 将第一个数据送入 ax
        jmp      ltest          , 循环检查
compare:     cmp      ax, [bp+6]  , 和下一值比较
        jle      ltest          , 如下一值小

```

```

        mov     ax, [bp+6]    ; 则下一值装入 ax
ltest:   add     bp, 2         ; 移至新数据
        loop   compare       ; 循环
exit:    pop     bp           ; 恢复 bp
        ret                     ; 返回到 C
_min     ENDP

```

请注意, 该 min 函数能处理各种 count 值:

- 如 count ≤ 0, min 返回 0。
- 如 count = 1, min 返回参数表第 1 元素。
- 如 count ≥ 2, min 连续比较, 找出参数表最小值。

8.2.5 寄存器使用约定

min 中使用了多个寄存器 (BP, SP, AX, BX, CX)。本节将介绍如何安全地使用寄存器, Turbo C 将用哪些寄存器。

上述 min 程序是正确的。其中使用的寄存器只有 BP 需注意, 在子程序入口处 BP 已压栈, 在出口处已弹出。

另外两个需考虑的寄存器是 SI 和 DI, Turbo C 将这两个寄存器用作寄存器变量, 如需在汇编语言中使用之, 需在子程序入口处保存它们 (可保存在栈中), 并在离开前恢复。如果用 -r- 选择项 (不使用寄存器变量) 编译 Turbo C 程序, 则无需考虑保存 SI 和 DI。

使用 -r- 选择项时需小心。请参阅第十章中有关寄存器变量选择项的内容。

根据所选存储模式, CS, DS, SS 和 ES 寄存器有相应的值。这一关系可表示为:

极小模式	CS = DS = SS, ES = 暂存
小、中模式	CS ≠ DS, DS = SS, ES = 暂存
紧凑、大模式	CS ≠ DS ≠ SS, ES = 暂存 (各模块有一 CS)
特大模式	CS ≠ DS ≠ SS, ES = 暂存 (各模块有一 CS 和 DS)

8.2.6 从汇编子程序调用 C 函数

要从汇编语言模块内调用 C 语言子程序, 第一步工作是要使 C 语言函数在汇编语言模块中可见。前面已简单介绍过这一问题: 将该函数说明为 EXTRN, 并带上修饰符 near 或 far。设有如下 C 函数:

```
long docalc (int *fact1, int fact2, int fact3);
```

为简单起见, 设 docalc 为 C 参数传递方式 (而非 Pascal 方式) 函数, 所用存储模式为极小、小或紧凑模式。汇编模块中将需如下说明:

```
EXTRN _docalc : near
```

如用中、大或特大存储模式, 则需说明为: _docalc : far。

docalc 有 3 个调用参数:

- (1) xval 的地址
- (2) imax 的值
- (3) 常量 421 (十进制)

假设调用结果存在名为 ans 的 32 位单元, 相应的 C 语言为:

```
ans = docalc ( &xval, imax, 421 );
```

首先压栈的是421, 然后是imax, 再是xval的地址。然后调用docalc。当其返回时, 必须清除栈中6个多余字节。然后将结果送至ans和ans+2。整个代码如下:

```
mov    ax, 421                ; 取421, 压栈
push   ax
push   imax                   ; 取imax, 压栈
lea    ax, xval               ; 取&xval, 压栈
push   ax
call   -docalc                ; 调用docalc
add    sp, 6                  ; 清栈
mov    ans, ax                ; 将32位结果送入ans
mov    ans+2, dx              ; 高位字一起送
```

如若docalc用Pascal参数传递方法, 这时需反序传送参数; 返回后不必清栈, 因为该子程序已经消过。而且docalc将按Pascal约定拼写(大写、无下划线)。

EXTRN语句如下:

```
EXTRN DOCALC, near
```

调用docalc的代码为:

```
lea    ax, xval               ; 取&xval, 压栈
push   ax
push   imax                   ; 取imax, 压栈
mov    ax, 421                ; 取421, 压栈
push   ax
call   DOCALC                 ; 调用docalc
mov    ans, ax                ; 将32位结果送入ans
mov    ans+2, dx              ; 包括高位字
```

关于Turbo C和其他语言的接口就介绍到这里。

8.3 程序设计的低级支撑

如果需要执行某些低层的工作, 而又不想涉及采用单独的汇编语言模块的种种麻烦。可利用Turbo C提供的三种低级支撑: 伪变量, 直接插入汇编代码和中断函数。以下分别介绍这些内容。

8.3.1 伪变量

机器内的CPU(8088/8086/80186/80286处理器)有几个寄存器, 用于数据处理。各寄存器均为16位(双字节)。大多数寄存器有专门用途, 而其中几个可用作通用寄存器。本章开始处, 介绍存储模式时, 谈及了这些寄存器的细节。

在进行低层程序设计时, 有时要求从C语言程序中直接访问这些寄存器, 例如:

- (1) 可能在调用系统子程序前将某些值装入寄存器;
- (2) 可能要查看寄存器的值。

用 Turbo C 的伪变量可方便地取接这些寄存器。伪变量实际上就是对应于寄存器的标识符，其类型可看为 unsigned int 或 unsigned char。

表 8.6 列出了所有的可用伪变量、其类型和相应的寄存器及其用途。

表 8.6 Turbo C 伪变量

伪变量	类 型	寄存器	通 常 用 途
_AX	unsigned int	AX	通用寄存器/累加器
_AL	unsigned char	AL	AX低字节
_AH	unsigned char	AH	AX高字节
_BX	unsigned int	BX	通用寄存器/变址器
_BL	unsigned char	BL	BX低字节
_BH	unsigned char	BH	BX高字节
_CX	unsigned int	CX	通用寄存器/计数和循环
_CL	unsigned char	CL	CX低字节
_CH	unsigned char	CH	CX高字节
_DX	unsigned int	DX	通用寄存器/存放数据
_DL	unsigned char	DL	DX低字节
_DH	unsigned char	DH	DX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	栈段地址
_ES	unsigned int	ES	附加段地址
_SP	unsigned int	SP	栈指针 (SS位移)
_BP	unsigned int	BP	基指针 (SS位移)
_DI	unsigned int	DI	用于寄存器变量
_SI	unsigned int	SI	用于寄存器变量

那么，为什么要存取寄存器变量呢？

可能有必要在调用低层系统程序前将寄存器设置成一定的值。例如，可用 INT 中断指令调用计算机只读存储器中的中断子程序，这时需首先设置寄存器如下：

```
void readchar(unsigned char page,
              unsigned char *ch,
              unsigned char *attr),
{
    _AH = 8,                /*功能号：读字符和属性*/

```



```

    _BH = page;           /*指定显示页号*/
    geninterrupt(0x10);   /*调用10h号中断*/
    *ch = _AL;            /*取读出字符*/
    *attr = _AH;          /*取读出属性*/
}

```

从中可以看到，中断号和显示页号都传递给了 10h 号中断子程序，而其回送的 值 被复制 到 ch 和 attr。

伪变量也可看作相应类型的 (unsigned int, unsigned char) 的全局变量。但因为它们是 CPU 中的寄存器，而在内存中无相应单元，使用者要了解其限制和使用方法：

(1) 伪变量不能使用取地址操作 (&)，因为伪变量无地址；

(2) 编译程序在不断产生代码，这些代码都将使用寄存器，故无法保证放在伪变量中的值能保存那怕很短一段时间。

也就是说，赋值必须仅在使用前进行，在寄存器取得所需值后马上就应读出，如前例 readchar 所示。特别是通用寄存器 (AX, AH, AL 等)，编译程序经常用这些寄存器临时存数据。更严重的是，CPU 对它们的改变无法预测。例如 CX 寄存器用于循环控制和移位，DX 寄存器用于 16 位乘法积的高字。

(3) 经过一函数调用，不能假设寄存器的值不变。假如，设有如下代码段：

```

    _CX = 18;
    myFunc( );
    i = _CX;

```

函数调用时并非所有寄存器都被保存，所以不能保证 i 取值为 18。在过程调用前后保证不变的寄存器只有 _CS, _BP, _SI 和 _DI。

(4) 修改某些寄存器要非常小心，这可能会引起一连串的预想不到的后果。例如，直接将值存入 _CS, _SS, _SP 或 _BP 几乎注定会使程序出错。因为 Turbo C 编译程序产生的代码将以各种方法使用这些寄存器。

8.3.2 直接插入汇编代码

前面已介绍过如何编写单独的汇编语言模块，并将其与 Turbo C 程序连接。Turbo C 还允许在 C 程序中直接写汇编语言代码。这称作直接插入汇编代码 (in-line assembly)。

在 C 程序中直接插入汇编代码，需用 _B 编译选择项。如不使用，则编译程序遇到直接插入代码时，将给出警告信息，并以 _B 选择项重新开始。也可以在源程序中用语句 #pragma inline 代替之，该语句实际上在编译程序遇到其时选用了 _B 选择项。

用户还必须复制一份 Microsoft 宏汇编程序 (MASM) 4.0 版或以上版本。编译程序处理时先产生一汇编代码文件，然后调用 MASM 将其翻译成 .OBJ 文件。

当然，对 8086 指令集和机器体系结构必须有所了解。即使不是编写完整的汇编语言子程序，还是需要了解所用指令集。

直接插入汇编语言指令以关键字 asm 开始，格式为：

```
asm <opcode> <operands> <; 或换行符>
```

其中：

- <opcode> 为合法 8086 指令 (下面将给出操作码表)。

• `<operand>` 为 `<opcode>` 可接收的操作数，可访问 C 中常量、变量和标号。

• `< ; 或换行符 >` 为分号或换行符，表示一条 asm 语句结束。

多条 asm 语句可在同一行上，用分号隔开。但一条 asm 语句不能跨行。

分号不能用于表示注解的开始(MASM中可以这样做)。asm 语句要用注解，需用 C 语言格式：

```
asm mov ax, ds; /*本注解正确*/
asm pop ax; asm pop ds; asm iret; /*本注释也正确*/
asm push ds; /*本注释不正确!*/
```

但其中最后一行会产生错，因其中注解有错。

`<opcode>` `<operand>` 将直接复制到输出文件，形成一汇编语言文件。其中的 C 语言符号将由等价的汇编形式代替。

直接插入汇编代码机制并不是完整的汇编程序，故许多错不能直接查出。MASM 将查出所有的错，但不能正确指出其出错位置，因为原有 C 语言程序行号已丢失。

每条 asm 语句对应于一条 C 语句。例如：

```
myfunc( )
{
    int i;
    int x;
    if (i > 0)
        asm mov x, 4
    else
        i = 7;
}
```

其中的 if 语句为合法条件语句。注意 mov x, 4 指令后无需分号。asm 语句是 C 语言中唯一依靠于换行的语句。这和 C 语言的其他部分不吻合。但有几个基于 UNIX 系统的编译程序也这样做。

下面是前面介绍过的 min 函数的直接插入汇编式版本：

```
int min (int v1, int v2)
{
    asm mov ax, v1
    asm cmp ax, v2
    asm jle minexit
    asm mov ax, v2
    minexit;
    return(_AX);
}
```

该例表明了使用直接插入汇编代码比调用汇编子程序更方便，功能更强。上述直接插入代码适用于大代码和小代码模式，适用于 Pascal 调用方式或 C 调用方式。在 min 的 ASM 版本中，必须顾及参数位移和标识符的拼法；而在直接插入版本中就无需考虑这些细节。

直接插入汇编语句可使用所有 8086 指令代码。Turbo C 允许的指令可分为四类：

- (1) 一般指令——通常 8086 指令集
- (2) 串指令——特殊的串处理指令
- (3) 转跳指令——各种转跳指令
- (4) 汇编指令——数据分配及定义

编译程序允许各式各样的操作数，可能其中有些是错的，不能被汇编程序接受。操作数的确切格式编译程序不作限制。

8.3.2.1 操作码

表 8.7 为一般指令操作码助记符。

表 8.7 操作码助记符

aaa	fcom	fldl2t	fsub	or
aad	fcomp	fldlg2	fsubp	out
aam	fcompp	fldln2	fsubr	pop
aas	fdecstp**	fldpi	fsubrp	popa
adc	fdisi	fldz	ftst	popf
add	fdiv	fmul	fwait	push
and	fdivp	fmulp	fxam	pusha
bound	fdivr	fnclx	fxch	pushf
call	fdivrp	fn disi	fxtract	rcl
cbw	feni	fneni	fyl2x	rcr
cld	ffree**	fninit	fyl2xpl	ret
cld	fiadd	fnop	hit	rol
cli	ficom	fn save	idiv	ror
cmc	ficom p	fnstcw	imul	sahf
cmp	fidi v	fnstenv	in	sal
cwd	fidivr	fnstsw	inc	sar
daa	fild	fpatan	int	sbb
das	fimul	fprem	into	shl
dec	fincstp**	fptan	iret	shr
div	finit	frndint	lahf	stc
enter	fist	frstor	lds	std
f2xmi	fistp	fsave	lea	sti
fabs	fisub	fscale	leave	sub
fadd	fisubr	fsqrt	les	test
faddp	fld	fst	mov	wait
fbld	fldl	fstcw	mul	xchg
fbstp	fldcw	fstenv	neg	xlat
feh s	fldenv	fstp	not	xor
fclex	fldl2e	fstsw		

注意：如在直接插入的汇编语句中使用了 80186 指令助记符，则需用 -l 命令行选择项 (80186/80286 指令集)。这将使编译程序在产生的汇编文件加入适当语句，以使 Microsoft 汇编程序能识别这些助记符。但如使用低版本汇编程序，将不支持这些助记符。

还要注意：如在使用浮点仿真程序(TCC编译选择项-f)的子程序中用直接插入汇编代码，编有“**”记号的指令操作码不能用。

串指令

除上述操作码外，还可使用如下串指令。串指令可单独使用。也可加重复前缀。

表 8.8 串 指 令

cmps	insw	movsb	outsb	scasw
cmps	lods	movsw	outsw	stos
cmps	lods	msb	scas	stos
ins	lodsw	outs	scas	stos
ins	movs			

重复前缀

可用的重复前缀为：

rep repe repne repnz repz

转跳指令

转跳指令将作特殊处理。因为指令本身不能加标号，转跳的目标需为 C 语言标号（在“转跳指令和标号的使用”中介绍）。可用的转跳指令如表 8.9 所列。

表 8.9 转 跳 指 令

ja	jge	jnc	jnp	js
jae	jl	jne	jns	jz
jb	jle	jng	jnz	loop
jbe	jmp	jnge	jo	loope
jc	jna	jnl	jp	loopne
jcxz	jnae	jnle	jpe	loopnz
je	jnb	jno	jpo	loopz
jg	jnbe			

汇编指令

Turbo C 直接插入汇编代码语句中，允许如下汇编指令：

db dd dw extrn

8.3.2.2 引用数据和函数

在 asm 语句中可使用 C 语言符号，Turbo C 将自动将其转换成等价的汇编语言操作数，并将下划线符加到标识符名前。各种符号均可使用，这包括局部变量，寄存器变量和函数参数。

通常，C 语言符号可用在地址操作数可用的场合。当然，寄存器变量可用于所有寄存器能用的地方。

当汇编程序在分析一指令的操作数时遇到一标识符，它在 C 符号表中搜寻该标识符。

8086 寄存器名不进行这一搜索。寄存器名大小写均可用。

寄存器变量

函数中只有两个最常用的寄存器说明被作为寄存器变量处理，其余的寄存器说明均作为自动(局部)变量。如说明中含关键字 `register`，而该说明不能作为寄存器，该关键字被忽略。

寄存器中只能存放 `short`, `int` (或相应 `unsigned` 类型)或双字节指针变量。寄存器变量被存放在 `SI` 和 `DI` 寄存器中。如函数中无寄存器说明，则直接插入汇编代码中可自由使用 `SI` 或 `DI` 作暂存寄存器。C 函数入口和出口点将自动保存和恢复调用者的 `SI` 和 `DI`。

如函数中有寄存器说明，直接插入代码可通过使用 `SI` 和 `DI` 以改变或引用寄存器变量，但最好是使用 C 语言符号，以防寄存器变量的内部实现有变。

位移和大小更改

当用直接插入方式编程时，无需顾及局部变量的确切位移，只需使用相应的名就会产生正确的位置。

但有时可能有必要引入适当的 `WORD PTR`, `BYTE PTR` 或其他大小更改字。例如，`LES` 指令或间接远调用指令需用 `DWORD PTR` 更改字。

8.3.2.3 使用 C 结构成员

在直接插入汇编代码中，当然可以按通常方式引用结构的成员，格式为 `<variable>.<member>`。这样处理的是一个变量，可以存取数据。也可以直接引用成员名(而不用变量)，将其作为一数值常量。这时，该常量为在相应结构中该成员的位移(字节数)。设有如下程序：

```
struct myStruct {
    int    a_a;
    int    a_b;
    int    a_c;
}myA;
myfunc( )
{
    ...
    asm mov ax, myA.a_b
    asm mov bx, [di].a_c
    ...
}
```

结构 `myStruct` 的三个成员为 `a_a`、`a_b` 和 `a_c`；其中变量 `myA` 为 `myStruct` 类型。第一条汇编语句将 `myA.a_b` 中的值传至 `AX` 寄存器。第二条将 `[DI]+offset(a_c)` 的值传至寄存器 `BX` (从 `DI` 中取出地址，加上 `myStruct` 中 `a_c` 的位移得到该地址)。结果，产生的代码为：

```
mov ax, DGROUP: myA+2
mov bx, [di+4]
```

为什么要这样做呢？如果在寄存器(如 `DI`)中装入了结构类型 `myStruct` 的变量地址，则可用成员位移直接访问成员。事实上，成员名可在汇编语句中用在任何数值常量可用的位

置。

结构成员需以句号(。)开始,表示其为结构成员,而非通常 C 符号。在输出的汇编文件中,成员名将以相应成员位移代替(a_c 的位移为4),但不保留类型信息。即成员在汇编语句中可作编译常量之用。

但在目前的版本中有一个限制,各结构的成员名不能相同。

8.3.2.4 转跳指令和标号的使用

在直接插入代码中,可用各种条件、无条件转跳指令和循环指令,但只在一个函数体中有效。因为 asm 语句中无法给标号,转跳指令的转移目标必须为 C 转跳标号。不能产生直接远转跳。

间接转跳也可使用。这时转跳指令的操作数可为寄存器名,或可将转跳地址括在方括号中。

在下面代码中,第一条转跳转移到 C 标号 a。第二条转移到整数 a 中所存地址。

```
int x( )
{
    int a;
    ...
    a;                      /*转跳标号 a*/
    ...
    asm jmp a                /*转跳至标号 a*/
    asm jmp [a]              /*转跳至整数 a 所存地址*/
    ...
}
```

8.3.3 中断函数

8086 处理器将内存的前 1024 个字节保留作为 256 个远指针,这些指针称为中断向量,指向称为中断处理子程序的系统子程序。执行如下的 8086 指令:

```
int < 中断号 >
```

将调用这些子程序,其中 < 中断号 > 取值为 0h 至 FFh。中断处理时,计算机保存代码段(CS)、指令指针(IP)以及状态标志的内容,关闭中断,然后通过一远转跳转至中断向量所指向的单元。例如,常用的中断调用(又称为系统功能调用)是:

```
int 21h
```

这可调用大多数 DOS 子程序。中断向量中许多位置未被使用,所以用户可编写自己的中断处理程序。将相应的远指针作为某个未用中断向量。

用 Turbo C 编写中断处理程序时,要求函数类型说明为 interrupt; 更具体地说,中断处理程序形为:

```
void interrupt myhandler (bp, di, si, ds, es, dx, cx, bx, ax, ip,
                           cs, flags, ...);
```

从中可看出,所有的寄存器均为其参数。所以可以不使用本章前面讨论过的伪变量,而使用和修改这些寄存器。处理程序还有另外的参数(flags, ...),这些参数须适当定义。

interrupt 类型的函数将自动保存(除 SI, DI 和 BP 外) AX 至 DX 以及 ES 和 DS。

这些寄存器在中断处理程序退出时将被恢复。

中断处理子程序可使用各种存储模式下的浮点算术运算。使用 8087 的中断处理程序须保存 8087 的状态，并在退出前恢复。

中断处理函数可修改参数。改变参数将在中断处理程序返回时修改相应的寄存器。当将中断处理程序作为用户服务子程序时这非常有用，这和 DOS 的 INT 21 相似。中断函数退出时需用 IRET (中断返回) 指令。

编写自己的中断处理程序，其一是用于大多数常驻内存子程序。这些子程序将自身变为中断处理子程序。这样，当某一特定事件或周期性事件发生时(时钟，键盘等)，这些子程序可截取处理中断的调用，作适当处理。然后将控制转给原有中断处理程序。

8.3.4 使用低级支撑的例子(BIOS和低级接口模块)

上面已介绍了如何在用户程序中使用低级设施的几个例子。现在再看几个例子。初学者最好编写一个实际的中断处理程序，该程序可做图形显示或产生音响。下面的例子，每次调用时响铃一次。

首先要编写函数本身，

```
#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,
    /*取当前控制端口设置*/
    bits=originalbits=inportb(0x61);
    for (i=0; i<=bcount; i++){
        /*关一会儿扬声器*/
        outportb(0x61, bits & 0xfc);
        for (j=0; j<=100; j++);
        /*空语句*/
        /*开一会儿扬声器*/
        outportb(0x61, bits | 2);
        for (j=0; j<=100; j++);
        /*又一空语句*/
    }
    /*恢复控制端口设置*/
    outportb(0x61, originalbits);
}
```

然后还须写一函数设置中断处理程序,其参数为函数地址和中断号(0..255或0x00..0xFF),该函数完成三项工作:

- (1) 关闭中断,这样修改中断向量表时无异常情况;
- (2) 将得到的函数地址传至适当单元;
- (3) 打开中断,恢复原状。

该设置程序如下:

```
void install(void interrupt(*faddr)(), int inum)
{
    setvect(inum, faddr);
}
```

最后,需要调用一下该中断子程序,以检查其正确性。下面函数即做此事:

```
void testbeep(unsigned char bcount, int inum)
{
    _AH=bcount;
    geninterrupt(inum);
}
```

主程序为:

```
main()
{
    char ch;
    install(mybeep, 10);
    testbeep(3, 10);
    ch=getch();
}
```

8.4 浮点库的使用

C语句中有两类数:整型数(int, short, long等)和浮点数(float, double)。计算机处理器处理整型数非常方便,但处理浮点数时间较长。但iAPX86系列处理器有一系列协处理器8087和80287。

8087和80287(通称为8087)是特殊的硬件数值处理器,可装在PC上。8087执行浮点指令非常快。如大量使用浮点运算,可能要用协处理器。计算机CPU通过一特殊中断和8087连接。

Turbo C可帮助程序员使程序适应于不同计算机和不同需求。

(1) 如根本不用浮点值,可告诉编译程序。

(2) 如需用浮点值,但计算机无数学协处理器,可告知Turbo C连接特殊的子程序模拟8087。这种情况下,若程序在有协处理器的系统上运行,将自动用协处理器(这时程序的运行要快得多)。

(3) 如程序只在有协处理器的系统上运行,可告知Turbo C编译程序产生总是用8087/80287的代码。

下面例子中假设已按第一章所说对工作盘作了适合设置。特别是,TCC和TLINK例

子假设文件 TURBOC.CFG 存在, -L 和 -I 路径设置正确, 库文件和启动代码文件存于名为 LIB 的子目录中。

8.4.1 仿真8087/80287芯片

若要用浮点数, 而计算机无数学协处理器, 或者程序运行的计算机可能有, 也可能没有协处理器, Turbo C 都能处理这些情况。

当使用仿真选择项时, 编译程序将产生 8087 仿真代码, 连接的是仿真库(EMU.LIB)。当程序运行时, 如有 8087 则将使用之; 如无 8087, 则程序将用特殊软件仿真 8087。

仿真库工作方式为:

- (1) 程序开始运行时, C 启动代码将确定有无 8087。
- (2) 若有协处理器, 程序将允许 8087 特殊中断直接传给 8087 芯片。
- (3) 若无协处理器, 程序将截取 8087 中断并转至仿真子程序。

假设将 RATIO.C 修改如下:

```
main( )
{
    float a, b, ratio;
    printf("Enter two values, ");
    scanf("%f %f", &a, &b);
    ratio = a/b;
    printf("The ratio is %0.2f\n", ratio);
}
```

若用 TC(集成开发环境版本), 需进入选择菜单, 选择 Compiler 和 Code generation, 然后选择浮点项直至遇到 Emulation。当编译和连接程序时, Turbo C 将自动选择适当的选择项和库。

若用 TCC(命令行版本), 编译命令行应为:

```
tcc -mx ratio
```

若手工连接产生的代码, 必须指定适当的数学库(根据存储模式)和 EMU.LIB 文件。仿真选择项(-f)缺省值为有, 所以除非 TURBOC.CFG 文件中有其他浮点开关(-f-或-f87)。TLINK 将按如下方式调用:

```
tlink lib\c0x ratio, ratio, ratio, lib\emu.lib lib\mathx.lib lib\cx.lib
```

其中 x 为表示适当存储模式的字母。

注意: tlink 命令需放在一行上, 而且库的次序非常重要。

8.4.2 8087/80287数学协处理器

若能完全确信程序所运行的系统有 8087 或 80287, 则可产生利用这一处理器优势的程序。同时将产生较小的 .EXE 文件, 而无需引用 8087 仿真子程序(EMU.LIB)。

若用 TC 编译程序, 需经过 Option 菜单, 先选择 Compiler, 再选 Code generation, 然后再选择 Floating point 直至遇到 8087/80287。当编译和连接程序时, Turbo C 将自动选择适当的选择项和库。

若用 TCC 编译程序, 命令行需用 -f87 选择项,

```
tcc -f87 -mx ratio
```

这将告诉 Turbo C 产生直接调用 8087/80287 的代码。当调用 TLINK 时, 将自动连接 FP87.LIB 和 MATHx.LIB。

对手工连接生成的代码, 必须指定适当的数学库(根据存储模式)和 FP87:

```
tlink lib\c0x ratio, ratio, ratio, lib\fp87.lib lib\mathx.lib lib\cx.lib
```

其中 x 亦为表示适当存储模式的字母。

8.4.3 不使用浮点数

若程序不使用浮点数, 即使在命令行中列出它们, 连接时连接程序也不连浮点库(与 MATHx.LIB 一起的 EMU.LIB 或 FP87.LIB)。若从连接程序命令行略去这些库名, 则可优化连接步骤。

假设要编译和连接的程序如下(保存在名为 RATIO.C 的文件中):

```
main( )
{
    int a, b, ratio;
    printf("Enter two values: ");
    scanf("%d %d", &a, &b);
    ratio = a/b;
    printf("The ratio is %d\n", ratio);
}
```

该程序未用浮点子程序, 编译时可打开浮点仿真开关, 或根本不用浮点设施。

若用 TC 编译程序版本, 并选择编译时带仿真, 只需在编译菜单中选择 Compile to OBJ。连接程序将在连接时包括浮点库, 但并不真正连入。

若需加快连接过程, 可指定“无浮点”。在转到 Options 菜单时, 选择 Compiler 和 Code generation, 然后选择 Floating point 项。

在这一命令周期中, 反复按 Enter 键可选择三个选择项: None, Emulation 和 8087/80287。可选 None(无浮点)选择项, 然后按 Esc 三次或仅按 F10 便可回到菜单亮条。

当设置了 None, 而编译和连接的程序有浮点运算时, Turbo C 将不连接任何浮点数学库。

若用 TCC 编译程序版本, 命令行需用 -f- 选择项, 例如:

```
tcc -f- -mx ratio.c
```

它告知 Turbo C 程序 ratio.c 中无浮点指令, 同时指出所用的存储模式。这里的 x 所指的是所希望的存储模式的首字母(t=极小, s=小, c=紧凑, m=中, l=大, h=特大)。

因为 RATIO.C 是一个单独的程序, 所以 TCC 将自动调用 TLINK, 连接 C0x.OBJ 和 Cx.LIB, 并生成 RATIO.EXE。

如果对 TCC 命令行仅用了“仅作编译”选择项(-c), 产生的代码需手工连接。这时无需(也不应该)指定数学库。TLINK 命令行为:

```
tlink lib\c0x ratio, ratio, ratio, lib\cx.lib
```

这将把 C0x.OBJ 和 RATIO.OBJ 连接在一起, 使用库为 Cx.LIB, 产生的文件为 RATIO.EXE 和 RATIO.MAP。

8.4.4 87环境变量

若用 8087 仿真方式产生程序, 亦即从菜单中选择了 Floating point...Emulation 或在 TCC 的命令行中包括了 -f 选择项。当程序运行时, C0x.OBJ 启动模块将自动地检查有无 8087。

若有 8087, 程序将用之; 若无, 程序将使用仿真程序。

有些情况下可能不希望自动检查。例如, 所使用的计算机系统上有 8087, 但希望检查程序在无协处理器的系统上能正确运行。或者程序需在 PC 兼容机上运行, 而该机自动检查功能有错 (如有 8087 而说不存在, 或者相反)。

Turbo C 提供了一种选择项, 可改变启动代码中的自动检查。这就是 87 环境变量。

可在 DOS 提示符下用 SET 命令设置 87 环境变数:

```
C> SET 87 = N
```

或

```
C> SET 87 = Y
```

将 87 环境变量置为 N (表示 NO), 告诉启动代码无需检查有无 8087 (虽然系统中可能有)。

相反, 将 87 环境变量置为 Y (表示 Yes) 表示有协处理器, 并希望程序能使用之。请注意, 如设置 87 = Y, 而系统无 8087, 程序将出错。

87 环境变量可改变自动检查方式, 因为程序运行时, 启动代码将首先检查 87 变量有无定义:

(1) 如定义了 87 变量, 则停止检查, 按预定义模式运行程序。

(2) 如未定义 87 变量, 则自动检查有无 8087, 程序根据检查结果运行。

若已定义 87 变量, 当需去除定义时, 可在 DOS 提示符下输入:

```
C> SET 87 =
```

(在键入 = 后按回车)。

8.4.5 寄存器和 8087

使用浮点数时, 有关寄存器使用要注意几点:

(1) 在 8087 仿真方式下, 不支持寄存器周转 (wrap-around)。

(2) 如浮点数和直接插入代码混用, 使用寄存器要特别注意。因为 Turbo C 调用函数前要清除 8087 寄存器集。调用使用协处理器的函数之前, 要弹出和保存 8087 寄存器, 除非能确信空闲寄存器够用。

8.4.6 浮点出错处理

程序执行时, 浮点子程序若检测到错误, 将自动调用 _matherr, 并传送几个参数。_matherr 然后用参数填写在 math.h 中定义的一中断结构中, 然后调用 _matherr, 并使用该结构指针。

`matherr` 是一钩子(hook), 可挂上自编出错处理程序。缺省情况下, `matherr` 不干任何事, 只回送 0。但可改变 `matherr` 使之能按用户要求处理浮点错误。对于修改后的 `matherr`, 如错误已处理则回送非零值, 否则回送 0。

详细情况可参看 Turbo C 附录 B 中关于 `matherr` 的描述。

8.5 警告和提示

8.5.1 Turbo C RAM 的使用

Turbo C 在编译时不在磁盘上产生任何中间数据结构 (Turbo C 仅将 `.obj` 文件写到磁盘上); 而使用 RAM 保存各趟间的中间数据结构。所以编译程序内存不够时将出现信息 OUT OF MEMORY...

解决的办法是将函数变小, 将含有大函数的文件拆开。如已装入常驻内存程序, 还需清除这些程序。

8.5.2 要慎用 Pascal 调用约定

在未阅读过本章, 对其内容尚未真正理解之前, 最好不用 Pascal 调用约定。若按 Pascal 调用方法编译主文件, `main` 需说明为:

```
cdecl main(int argc, char *argv[], char *envp[])
```

8.5.3 在 DOS 3.2 和有浮点协处理器下使用 Turbo C

DOS 3.2 在处理浮点例外时有一个错误。当例外出现时, 系统为例外处理程序分配一个局部的堆栈。但是在系统栈分配后, 它未进行释放。栈默认最多有 8 个。当 9 个浮点例外出现后, 系统挂起, 需重新冷启动。典型的例外情况是溢出和除数为 0。在程序终止时, 栈并没有再释放。因此如果一个程序有一次除数为 0, 经过运行 9 次后就要破坏系统。为了避免这个问题, 可以这样做:

(1) 不要用 DOS 3.2。

(2) 从 IBM 或 Microsoft 那里获得一块用来修补这个错误的补丁, 并把它装到 DOS 备份上。

(3) 不用浮点协处理器。

(4) 不在 Turbo C 程序中捕获例外。只要调用 `_clear87()` 库函数就能很容易地做到这一点。例如:

```
#include <float.h>
unsigned int fpstatus;
/*stdlib.h用于exit,stdio.h用于puts*/
#include <stdlib.h>
#include <stdio.h>
main()
{
    _clear87();
```

```

/*这里进行浮点操作*/
/*...*/
/*退出前检查屏蔽的例外*/
fpstatus=_status87( );
if (fpstatus & SW_INVALID)
    puts("Floating point error: invalid operation.");
if(fpstatus & SW_ZERODIVIDE)
    puts("Floating point error: zero divide.");
if(fpstatus & SW_OVERFLOW)
    puts("Floating point error: overflow.");
exit(fpstatus & (SW_INVALID | SW_ZERODIVIDE
                | SW_OVERFLOW));
}

```

以上介绍的 Turbo C 的低级编程设施的三个方面：伪变量、直接插入汇编代码和中断函数；和其他语言（包括汇编语言）的接口；浮点程序的使用和各种存储模式。希望读者能很好地使用这些技术，充分挖掘 PC 机的潜力。

此至，本章前三个问题已经作了完整的阐述。

以下将进一步介绍 Turbo C 的字符屏幕管理和图形处理功能，以及在正文模式和图形模式下的编程技术。

8.6 Turbo C 的字符屏幕管理

当今软件系统的设计和实现，除功能、性能等常规质量指标要求更高外，在用户友善性方面又提出了越来越高的要求。用户友善性的一个方面是设计并实现美观、清晰的显示画面，以使用户易于了解软件的功能，乐于使用软件。

IBM PC, PC/XT, PC/AT 的显示器为优美的显示画面提供了良好的硬件基础，它在彩色字符方式下，可以 16 种颜色显示字符，且字符底色可换用八种颜色，另外还可控制字符的闪烁。可是，如此丰富的字符屏幕控制功能，在现有条件下，一般只能通过汇编语言进行控制。这就使广大计算机用户望而却步，使很多高性能的显示器大材小用。

在这种情况下，美国 Borland 公司在其推出的 Turbo C 1.5 版中提供了一组较为简单，但功能较强的函数，使广大计算机用户可方便地控制 PC 的字符显示。

本节将对 Turbo C 1.5 版的字符屏幕处理作一简单介绍。

8.6.1 基本概念

IBM PC 系列机可支持多种显示设备，有单色的、彩色的、高分辨率的、低分辨率的。各种显示器在机器内部都需有相应的显示控制卡。常用的控制卡有：单色显示卡(MDA)，彩色图形卡(CGA)，增强图形卡(EGA)等。各显示卡可工作在多种显示模式下，显示模式决定了屏幕上字符显示的列数(80列或40列)和行数(25行，43行或50行)，和显示类型(彩色、单色及黑白)。Turbo C 1.5 共支持五种字符显示模式(未支持 43 行、50 行显示模式)，见表 8.10。

表8.10 字符显示模式

符 号 常 量	数 字 值	含 意	显 示 卡
BW40	0	黑白, 25行, 40列	CGA, EGA
C40	1	彩色, 25行, 40列	CGA, EGA
BW80	2	黑白, 25行, 80列	CGA, EGA
C80	3	彩色, 25行, 80列	CGA, EGA
MONO	7	单色, 25行, 80列	MDA

在各种显示模式下,屏幕分成若干个单元(例如在C80模式下,屏幕分成 $25 \times 80 = 2000$ 个单元)。各单元以行、列编号。行为1至25,列为1至40或80。例如屏幕左上角为(1,1)。各单元由一字符和一属性组成。字符可以是任意ASCII字符,或扩充字符;而属性则用于控制字符显示的显示色、背景色和是否闪烁。各单元的属性值和字符可分别控制。

属性值由八位组成,各位的意义如下:

7	6	5	4	3	2	1	0
闪	烁	字	符	底	色	字	符
闪	烁	字	符	底	色	字	符

其中闪烁位取值为0或1,底色取值为0至7,显示色取值为0至15。

Turbo C 1.5 版定义了如表8.11所示的颜色常量。

表8.11 颜色常量

符 号 值	数 字 值	用 法	含 意
BLACK	0	前景或背景	黑
BLUE	1	前景或背景	兰
GREEN	2	前景或背景	绿
CYAN	3	前景或背景	青
RED	4	前景或背景	红
MAGENTA	5	前景或背景	洋红
BROWN	6	前景或背景	棕
LIGHTGRAY	7	前景或背景	淡灰
DARKGRAY	8	前景	深灰
LIGHTBLUE	9	前景	淡兰
LIGHTGREEN	10	前景	淡绿
LIGHTCYAN	11	前景	淡青
LIGHTRED	12	前景	淡红
LIGHTMAGENTA	13	前景	淡洋红
YELLOW	14	前景	黄
WHITE	15	前景	白

另外还定义 BLINK = 128 用于控制闪烁位。

8.6.2 显示方式控制

对各种字符显示模式, 各种显示属性, Turbo C 都提供了相应的函数加以控制。

使用字符屏幕操作首先需设置显示模式。这可用函数 textmode, 其原型为:

```
void textmode (int mode);
```

其中 mode 可为 BW40, C40, BW80, C80, MONO 或 LAST(-1)。调用 textmode 后, 显示屏幕将清屏, 属性采用常规属性; 若 mode 为 LAST, 则将重新选择最近使用过的字符模式, 一般用于从图形模式回到字符模式。

例如: textmode(C80); 将屏幕置为 80 列×25 行彩色字符方式。

显示属性控制方面, Turbo C 1.5 提供的功能较为简单。程序员可控制的是当前属性值。这一属性值将决定以后所有字符输出的属性(直到再次改变属性值)。这组函数的原型为:

含 意	原 型	参数取值范围
显示色	void textcolor (int color);	BLACK(0) 至 WHITE(15)
背景色	void textbackground (int color);	BLACK(0) 至 LIGHTGRAY(7)
属 性	void textattr (int attribute);	0 .. 255

例如, 将当前显示属性置为兰底黄字可调用:

```
textcolor (YELLOW); textbackground(BLUE);
```

或 textattr(BLUE<<4 | YELLOW);

8.6.3 字符输出

输出字符首先要确定输出的区域, 通常这一区域是整个屏幕。而 Turbo C 允许在屏幕上划出一矩形区域作为输出区域。这一区域称为一窗口, 当然这只是一个非常简单的窗口。一旦设置了一窗口, 则以后所有输出只影响该窗口内部(包括滚屏), 不影响屏幕其他部分。窗口设置用函数 window, 其原型为:

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

其参数分别表示窗口的第一列, 第一行, 最后一列, 最后一行。

通常我们需要在某一固定位置输出信息, 这一位置可由函数 gotoxy 设置, 其原型为:

```
void gotoxy (int x, int y);
```

其参数表示输出位置在当前窗口中的列和行。

设置了这一切后, 字符信息本身的输出非常方便, 并且和原有输出函数有简单对应关系:

int cprintf (char *format, a1, ...);	/*窗口格式化输出*/
int cputs (char *string);	/*窗口字符串输出*/
int putch (int ch);	/*窗口字符输出*/
int getche ();	/*输入字符、窗口回显*/

例如:

```
window(5, 5, 75, 20);           /*设置窗口*/
gotoxy(20, 5);                   /*窗口中第20列, 第5行*/
cputs("Isn't Turbo C powerful!"); /*输出一字符串*/
```

8.6.4 程序例

假设需编制一桥牌程序, 其最基本的要求是要以直观的形象表示一手牌。PC 字符集中有四个专用字符可表示黑桃(chr(6)), 红桃(chr(3)), 方块(chr(4))和草花(chr(5))。另外可用 Turbo C 的字符屏幕控制函数, 以黑色显示黑桃和草花, 以红色显示红桃和方块。这样即可清晰地表示一手牌:

下面的程序将在屏幕上显示如下牌型:

黑桃 A Q 10 5 4 3

红桃 7 5

方块 10 6 4

草花 Q 3

```
#include <conio.h>
#include <stdio.h>
enum suit {spade, heart, diamond, club};
int color[4] = {DARKGRAY, LIGHTRED, LIGHTRED, DARKGRAY};
char symb[4] = {0x06, 0x03, 0x04, 0x05};
void showcards(int line, int suit, char *faces)
{
    gotoxy(8, line);
    textcolor(color[suit]);
    putchar(symb[suit]); cputs(" ");
    cputs(faces);
}
main()
{
    textmode(C40);
    textcolor(YELLOW);
    textbackground(CYAN);
    window(8, 8, 35, 20); clrscr();
    gotoxy(1, 1); cputs("A Hand of cards in card game");
    showcards(6, spade, "A Q 10 5 4 3");
    showcards(8, heart, "7 5");
    showcards(10, diamond, "10 6 4");
    showcards(12, club, "Q 3");
}
```

本节简要介绍了 Turbo C 1.5 版的字符屏幕控制功能, 介绍了其中部分函数及其用法。其他未介绍的函数, 可在窗口中作插入、删除、清除操作, 可将屏幕上一区域拷贝至另一区

域,可进行屏幕信息和内存缓冲区信息的交换,可查询当前屏幕的状态。细节可参见附录B和参考文献[3]。

8.7 Turbo C 的图形功能

在许多计算机应用领域中,如计算机辅助设计、计算机辅助制造、计算机辅助教学,图形是一种形象直观的信息显示方法。

IBM PC 系列机上,除BASIC语言外,通常高级语言的图形功能均较弱,有的根本不支持图形功能。然而,Turbo C (1.5版)提供了一个功能较强的图形库。它支持CGA,EGA,VGA等多种显示卡的各种显示模式,提供了画点、线、圆、椭圆、多边形,输出各种字体、大小的字符等各种原语。这就为开发高质量的含有图形显示的软件提供了方便的工具。本节对 Turbo C 1.5 版图形处理的设施及其使用方法作一简单介绍。

8.7.1 基本概念

IBM PC 系列微机及其兼容机提供了多种图形控制卡控制图形的显示。Turbo C 1.5支持如下八种常用图形卡:

- 彩色图形卡(CGA),
- 多色图形阵列(MCGA),
- 增强图形适配器(分为EGA, EGA64, EGAMONO三种)
- IBM 8514 图形适配器
- 视频图形阵列(VGA)
- 大力神图形适配器(HERCMONO)
- AT & T 400 线图形适配器(ATT400)
- 3270PC 图形适配器(PC3270)

和字符显示一样,图形的显示按其能力可分为多种显示模式。显示模式决定了显示的分辨率(屏幕上点的个数),可同时显示的颜色多少,调色板的设置方式,以及存储图形的页数。例如CCDOS使用的通常是CGA板支持的高分辨率模式,可显示200线 \times 640点,2种颜色,存储1页图形。所以如用16 \times 16汉字库,CCDOS在一屏上可显示12.5行 \times 40字汉字。Turbo C 所支持的图形模式的情况如表8.12所示。

在图形模式中,整个屏幕是一个点组成的阵列。每一点在屏幕上产生一个有颜色的光点,这种点称为像素(或称图素)。每个像素均有一个值表示当前显示的颜色。例如在EGA卡的EGAHI模式中,屏幕可显示640 \times 350个像素,每个像素的值为0至15,以区别16种颜色。

但是,像素的值并不直接表示像素的颜色,它只是一个色彩表的下标。这一色彩表称为调色板。对应于某一像素值的调色板项确定该像素的实际颜色。调色板的大小对应于当前可同时显示的颜色数。例如EGA板硬件允许显示64种颜色(编号为0..63),而EGAHI模式中同时只可显示16种颜色(编号为0..15)。则调色板有16项,每项的值为0..63。如当前屏幕上有一像素值为3,而调色板第3项为EGA_BROWN(20),则该像素的实际颜色为棕色。

采用调色板的方式控制彩色显示不仅使色彩更加丰富,而且修改图形较为方便。例如,

表8.12 Turbo C支持的图形模式

图 形 卡	图 形 模 式	分 辨 率	颜色数	页 数
CGA	CGAC0—CGAC3	320×200	4	1
	CGAHI	640×200	2	1
MCGA	MCGAC0—MCGAC3	320×200	4	1
	MCGAMED	640×200	2	1
	MCGAHI	640×480	2	1
EGA	EGALO	640×200	16	4
	EGAHI	640×350	16	2
EGA64	EGA64LO	640×200	16	1
	EGA64HI	640×350	4	1
EGAMONO	EGAMONOH1	640×350	2	1或2
HERCMONO	HERCMONOH1	720×348	2	2
ATT400	ATT400C0-ATT400C3	320×200	4	1
	ATT400MED	640×200	2	1
	ATT400HI	640×400	2	1
VGA	VGALO	640×200	16	2
	VGAMED	640×350	16	2
	VGAHI	640×480	16	1
PC3270	PC3270HI	720×350	2	1

只要将调色板第3项改为 EGA_YELLOW(62), 则上例中所有棕色点均改为黄色。

彩色图形卡(CGA)调色板的控制限制较大。在中分辨率方式下(CGAC0, CGAC1, CGAC2, CGAC3), 硬件可显示16种颜色, 屏幕可同时显示4种颜色, 故调色板有4项。但其调色板中只有第0项可作修改, 其余3项只有4种固定组合, 如表8.13所示。

表 8.13

模 式	调色板第0项	调色板第1项	调色板第2项	调色板第3项
CGAC0	可 变	浅 绿	浅 红	黄
CGAC1	可 变	浅 洋 红	浅 青	白
CGAC2	可 变	绿	红	棕
CGAC3	可 变	洋 红	青	浅灰

这四组固定的调色板可在模式中直接加以区别,例如CGAC 2模式表示CGA卡中分辨率模式,且调色板的1至3项为(绿、红、棕)。MCGA卡的MCGAC 0-MCGAC 3模式以及ATT400卡的ATTC 0至ATTC 3模式的含义同上。

在CGA的高分辨率模式CGAHI下,同时可显示2种颜色。但其中背景色(0)是固定的,总是黑色,只有前景色(1)可作修改,显示16种颜色中的一种。和这类似的还有MCGAMED, MCGAHI, ATT400MED和ATT400HI模式。

下面将具体介绍Turbo C图形功能。

8.7.2 图形系统控制

Turbo C图形系统分为两部分。一部分是和机器无关的图形库GRAPHICS.LIB,其嵌入文件为GRAPHICS.H。任何使用图形的C程序必须引用GRAPHICS.H并连接GRAPHICS.LIB。图形系统的另一部分为和具体显示卡有关的图形设备驱动程序。图形设备驱动程序以文件形式存于磁盘,其扩展名为BGI(Borland图形接口)。例如,EGA卡的设备驱动程序为EGA.BGI。

在使用Turbo C图形系统之前,必须对图形系统进行初始化。这包括分配内存,从盘中调入当前图形卡对应的设备驱动程序,将当前显示模式置为某一图形模式等工作。初始化可通过initgraph函数完成:

```
void far initgraph(int far* graphdriver,
                  int far* graphmode,
                  char far* pathtodriver);
```

其中graphdriver代表图形卡,graphmode为图形模式,pathtodriver为查找图形设备驱动程序所用的路径名。

和初始化对应的是结束处理,当图形系统用完后,可调用closegraph函数,释放所有的内存,并将屏幕恢复为调用initgraph之前的模式。closegraph的函数原型为:

```
void far closegraph(void);
```

[例8.1] 在配有EGA卡的系统中将屏幕置为高分辨率模式。

```
#include <graphics.h>
main( )
{
    int driver, int mode;
    driver = EGA;
    mode = EGAHI;
    initgraph(&driver, &mode, "");
    .....
    closegraph();
}
```

8.7.3 色彩控制

图形系统初始化后,在真正绘图之前,还必须对绘图所用色彩进行配置。这包括设置背景色、绘图色和整个调色板。

调色板设置确定了调色板中每一项对应的实际颜色，这可用 setpalette 函数完成。

```
void far setpalette(int index, int actual_color);
```

其中第一个参数为调色板下标，第二个参数为实际颜色。颜色编号如表8.14所示。

表8.14 调色板实际颜色编号

颜 色	CGA		EGA或VGA	
	符 号 名	值	符 号 名	值
黑	BLACK	0	EGA-BLACK	0
兰	BLUE	1	EGA-BLUE	1
绿	GREEN	2	EGA-GREEN	2
青	CYAN	3	EGA-CYAN	3
红	RED	4	EGA-RED	4
洋红	MAGENTA	5	EGA-MAGENTA	5
棕	BROWN	6	EGA-BROWN	20
浅灰	LIGHTGRAY	7	EGA-LIGHTGRAY	7
深灰	DARKGRAY	8	EGA-DARKGRAY	58
浅兰	LIGHTBLUE	9	EGA-LIGHTBLUE	57
浅绿	LIGHTGREEN	10	EGA-LIGHTGREEN	58
浅青	LIGHTCYAN	11	EGA-LIGHTCYAN	59
浅红	LIGHTRED	12	EGA-LIGHTRED	60
浅洋红	LIGHTMAGENTA	13	EGA-LIGHTMAGENTA	61
黄	YELLOW	14	EGA-YELLOW	62
白	WHITE	15	EGA-WHITE	63

调色板中第0项颜色通常称为背景色，这是清屏操作所用的颜色。背景色可用专门的函数设置。

```
void far setbkcolor(int actual_color);
```

其参数表示实际颜色。

如前所述，CGA卡的调色板，在中分辨模式下只有第0项可修改，其余3项只有四种组合。第0项也可用 setbkcolor 修改。而在高分辨率模式下，背景为黑色，只有前景色可修改。由于CGA卡硬件设计的特殊约定，用 setbkcolor 在 CGAHI 模式下实际修改的是前景色。

绘图色是画线时各点的颜色，可用 setcolor 设置。

```
void far setcolor(int color);
```

其中 color 为调色板的下标。

[例8.2] 在 EGAHI 模式下置调色板为--彩虹。

```
#include <graphics.h>
main()
{ int driver=EGA, mode=EGAHI,
  initgraph(&driver, &mode, "");
  setpalette(0, EGA-RED);
  setpalette(1, EGA-BROWN);
  setpalette(2, EGA-YELLOW);
  setpalette(3, EGA-GREEN);
  setpalette(4, EGA-CYAN);
  setpalette(5, EGA-BLUE);
  setpalette(6, EGA-MAGENTA);
  :
  closegraph();
}
```

[例8.3] 在 CGAHI 模式下反复修改前景色。

```
#include <graphics.h>
main()
{ int driver=CGA, mode=CGAHI, bkcolor,
  initgraph(&driver, &mode, "");
  for(bkcolor=0; bkcolor<16; bkcolor++)
  { setbkcolor(bkcolor);
    :
  }
  closegraph();
}
```

8.7.4 绘图和着色

Turbo C 的绘图和着色功能相当强, 可以画点、直线、圆弧、圆、椭圆、长方形、圆饼图(或称扇形图)、二维或三维直方图(或称条形图)、多边形等, 可以控制画线的颜色、粗细、方式, 可以将有限区域以11种预定义模式着色, 也可以自定义着色模式。用这些功能可方便地画出丰富多彩的图形。

直线、圆、长方形等图形是由线段构成的。在绘制前需设置画线方式。画线方式包括画线颜色、画线类型、画线粗细。线颜色可用前述setcolor函数设置。画线类型有四种预定义类型, 用户也可以自定义, 如表8.15所示。

用户定义画线类型用一16位字定义。其中某位为1时, 线中的相应点即以画线颜色画出。例如, 实线对应于0xFFFF, 而短横线对应于0xF0F0。画线粗细有两种选择: 1个像素宽(NORM-WIDTH)或3个像素宽(THICK-WIDTH)。画线类型和粗细可由如下函数设置:

```
void far setlinestyle(int linestyle, unsigned upattern, int thickness);
```

表8.15 线段类型

名	含 义	值
SOLID-LINE	直线	0
DOTTED-LINE	虚线	1
CENTER-LINE	中心线	2
DASHED-LINE	短横线	3
USERBIT-LINE	用户自定义	4

常用的画图函数有:

(1) 直线

```
void far line (int x0, int y0, int x1, int y1);
```

画直线(x0,y0)一(x1,y1);

(2) 圆弧

```
void far arc(int x, int y, int stangle, int endangle, int radius);
```

以(x,y)为圆心,画半径为radius的圆弧,其起始角为stangle度,终止角为endangle度,其中角度以反时针时数,0度为3点钟位置,90度为12点钟位置。

(3) 圆

```
void far circle (int x, int y, int radius);
```

以(x,y)为圆心,画半径为radius的圆。

(4) 椭圆

```
void far ellipse (int x,int y,int stangle,int endangle, int xradius,
int yradius);
```

以(x,y)为圆心,画x,y方向半径为xradius,yradius的椭圆圆弧,其起角为stangle度,终止角为endangle度。

(5) 长方形

```
void rectangle (int left, int top, int right, int bottom);
```

画一左上角为(left,top),右下角为(right,bottom)的长方形。

(6) 多边形

```
void far drawpoly (int numpoints, int far* polypoints);
```

给定numpoints个点的坐标,画一多边形,其中polypoints指向numpoints*2个整数的数组。若画一n边形,则需给n+1个点,使第n+1个点和第1点重合。

计算机作图的另一基本要求是着色,即将某一区域全部以某一模式涂成某一颜色。Turbo C提供了12种预定义着色模式,并允许用户自定义着色模式。着色模式如表8.16所示。

着色模式和着色颜色可用setfillstyle设置:

```
void far setfillstyle (int pattern, int color);
```

其中pattern为模式,color为颜色。若pattern为USER-FILL则,用户必须用8个字节定义一个8×8点阵的着色模式。再调用setfillpattern设置该模式:

表8.16 着色模式

名	值	含 意
EMPTY-FILL	0	以背景色着色
SOLID-FILL	1	全部着色
LINE-FILL	2	水平线
LTSLASH-FILL	3	左斜线
SLASH-FILL	4	左斜线, 粗
BKSLASH-FILL	5	右斜线, 粗
LTBKSLASH-FILL	6	右斜线
HATCH-FILL	7	浅阴影线
XHATCH-FILL	8	重交叉阴影线
INTERLEAVE-FILL	9	交替直线
WIDEDOTFILL	10	稀 点
CLOSEDOT-FILL	11	密 点
USER-FILL	12	用户定义

```
void far setfillpattern (char far * upattern, int color);
```

例如, 类似国际象棋棋盘的着色模式可定义为:

```
color chessboard[8] = {0xAA, 0x55, 0xAA, 0x55,
                      0xAA, 0x55, 0xAA, 0x55
                      }
```

常用的着色函数有:

(1) 直方图

```
void far bar (int left, int top, int right, int bottom);
```

以当前着色方法, 画一左上角为(left, top), 右下角为(right, bottom)矩阵。可用于构造直方图。

(2) 立体直方图

```
void far bargd (int left, int top, int right, int bottom, int depth,
               int topflag);
```

以当前着色方法, 画一左上角为(left, top), 右下角为(right, bottom), 深度为 depth 的长方体, 若 topflag 非零, 则画上长方体的顶部, 否则不画。可用于构造立体直方图。其中 topflag 用于控制在重叠式直方图中, 只对顶部长方体画顶部。

(3) 多边形

```
void far fillpoly (int numpoints, int far * polypoints);
```

类似于 drewpoly, 不同的是画完后着色。

(4) 圆饼图

```
void far pieslice (int x, int y, int stangle, int endangle, int radius);
```

类似于 arc, 不同的是用所画圆弧和圆弧两顶点和圆心的连线画一扇形, 以当前着色方法着色。

(5) 不规则图形着色

```
void far floodfill (int x, int y, int border);
```

将一个以 border 为颜色的点构成的有界区域, 以当前方法着色。其中(x,y)为该区域内任一点。如(x,y)不在一有界区域中, 则整个屏幕都将被着色。

[例8.4] CGA图形的一些例子。

```
#include <graphics.h>
main()
{
    int graphdriver = DETECT, graphmode;
    struct arcinfo arcinfo;
    int xasp, yasp;
    long xlong;
    initgraph(&graphdriver, &graphmode, "");
    /*画一半径为50的90度的圆弧*/
    arc(100, 120, 0, 89, 50);
    /*取弧的坐标, 并连接两端点*/
    getarcinfo(& arcinfo);
    line(arcinfo.xstart, arcinfo.ystart, arcinfo.xend, arcinfo.yend);
    /*画一圆*/
    circle(100, 120, 80);
    /*在圆中画一椭圆*/
    ellipse(100, 120, 0, 359, 80, 20);
    /*画圆饼图*/
    setfillstyle(HATCH_FILL, 1);
    pieslice(200, 50, 0, 134, 49);
    setfillstyle(SLASH_FILL, 1);
    pieslice(200, 50, 135, 225, 49);
    setfillstyle(WIDEDOT_FILL, 1);
    pieslice(200, 50, 225, 360, 49);
    /*画一“正”方形*/
    petaspectratio(& xasp, & yasp);
    xlong = (50l * (long) yasp) / (long) xasp;
    rectangle(0, 0, (int) xlong, 50);
    getch();
    closegraph();
}
```

[例8.5] 画棋盘。该程序在配有EGA卡的系统中, 能显示一棕色底, 黑框, 19×19画棋盘, 标上星位, 并放上黑白各一子。

程序先用自动检测功能初始化图形系统,也可设置 `graphdriver = EGA`, `graphmode = EGAHI`, 然后置调色板前三项为白、棕、黑。以棕色画出棋盘底板,以黑色粗实线画出棋盘外框,以黑色细实线画出棋盘上 19×19 个格点,再标上9个星位。最后在左上角星位放一白子,在右下角星位放一黑子。

```
#include <graphics.h>
#define posx(i) (i*15+10)
#define posy(i) (i*15+10)
void putstone(int i, int j, int color)
{
    setstone(color);
    setfillstyle(SOLID_FILL,color);
    pieslice(posx(i), posy(j), 0, 359, 5);
}
main()
{
    int graphdriver=DETECT, graphmode;
    int i,j;
    initgraph (& graphdriver, & graphmode,"");
    setpalette(0,EGA_WHITE);
    setpalette(1,EGA_BROWN);
    setpalette(2,EGA_BLACK);
    setfillstyle(SOLID_FILL,1);
    bar(posx(1)-10,posy(1)-10,posx(19)+10,posy(19)+10);
    setlinestyle(SOLID_LINE,3); setcolor(2);
    rectangle(posx(1)-5,posy(1)-5, posx(19)+5,posy(19)+5);
    setlinestyle (SOLID_LINE,1);
    for(i=1,i<=19,i++)
    {
        line(posx(1),posy(i),posx(19),posy(i));
        line(posx(i),posx(1),posy(i),posy(19));
    };
    setlinestyle(SOLID_LINE,3);
    for (i=4; i<=16; i=i+6)
    for (j=4; j<=16; j=j+6)
    {
        line(posx(i)-5,posy(j),posx(i)+5,posy(j));
        line(posx(i),posy(j)-5,posx(i),posy(j)+5);
    }
    putstone(4,4,0);
    putstone(16,16,2); getch();
    closegraph();
}
```

8.7.5 图形屏幕管理和视区设置

除了绘图和着色, 图形库还提供了一些函数用来管理屏幕、视区、图形和象素。本节先给出一些有关的函数(见表8.17), 然后介绍其作用。

表8.17 图形屏幕管理和视区设置函数

	函 数 名	功 能
屏幕管理	cleardevice	清屏
	setactivepage	设置图形输出活动页
	setvisualpage	设置可见图形页数
视区管理	clearviewport	清除当前视区
	getviewsettings	返回关于当前视区的信息
	setviewport	为图形输出置当前输出视区
图形管理	getimage	把指定区域的位图保存到内存
	imagesize	返回要存放屏幕上的一矩形区域所要求的字节数
	putimage	把以前保存的位图送回到屏幕上
象素管理	getpixel	取(x,y)处象素颜色
	putpixel	在(x,y)处置一个象素

通过调用 cleardevice 可一次性清除整个屏幕内容, 这个例行程序擦除了全部屏幕, 但不改变视区中的当前位置(cp), 且不改变所有的其他图形系统的设置(线条、着色和正文类型、调色板、视区设置等)。

根据所用的图形适配器, 系统可有 1 至 8 个屏幕页缓冲, 这是在内存的区域, 其中屏幕上的图形按点存储。可以指定哪个屏幕页是活动的(图形函数把它们的输出放在什么地方)和哪页是可见的(在屏幕上显示的一页), 这由函数 setactivepage 和 setvisualpage 完成。

一旦屏幕处于某个图形模式, 就能用 setviewport 在屏幕上定义一个视区。视区是图形模式下程序设计的一个术语, 是指在 PC 显示屏幕上定义的一个矩形区域。在图形程序输出图形时, 它起着虚拟屏幕的作用, 屏幕的其余部分(视区外部分)不改变。

定义视区位置时采用绝对屏幕坐标, 并说明裁剪是打开还是关闭。用 clearviewport 可清除视区。要得到当前视区的绝对屏幕坐标和裁剪状态, 可调用 getviewsettings 函数。

可利用 getimage 来得到现行屏幕上图形的一部分, 调用 imagesize 来计算要存储所得图形到内存所需的字节数, 然后把存储的图形送回屏幕(可在任何地方), 这由 putimage 实现。

所有输出函数的坐标(图形、着色、正文等)都是相对于视区的。

还可以通过函数 getpixel (返回给定象素的颜色)和 putpixel (置某颜色于某象素)操纵各象素的颜色。

8.7.6 图形模式下的正文输出

本节先对图形模式正文输出函数作一个简介。

gettextsettings	返回当前正文字体、方向、大小和对齐信息
outtext	把一个字符串送到屏幕当前位置(cp)上
outtextxy	把一个字符串送到屏幕某位置上
registerbgifont	注册一连入或用户装入字体,以便在连接时加入
settextjustify	为 outtext 和 outtextxy 置正文对齐值
settextstyle	置当前正文字体、类型和字符放大因子
setusercharsize	为笔划字体置宽度和高度
textheight	返回串高像素数
textwidth	返回串宽像素数

为在图形模式下输出正文,图形库包括 8×8 位图字体和几个笔划字体。

在位图字体中,每一个字符用一像素矩阵定义。

在笔划字体中,每个用一字符串向量定义,它告诉图形系统怎样画出这个字符。

画大字符时,使用笔划字体的好处是显然的。由于笔划字体是由向量定义的,当字体放大时仍能保持好的分辨率和质量。而当放大一个位图字体时,矩阵要乘上一比例因子,随着比例因子的增大,字符分辨率就较差,对于小字符来说,位图字体较有效,但对于大正文应选择笔划字体。

输出图形正文要调用 outtext 或者 puttextxy,并用 settextjustify 来控制输出正文的对齐方式(相对于cp)。要选择字符字体、方向(水平或垂直)、大小比例,这由 settextstyle 实现。利用 gettextsettings 得到当前正文设置,这个函数返回当前正文字体、对齐、放大、方向,它们保存在 textsettings 结构中。setusercharsize 允许修改笔划字体字符的宽度和高度。

如果裁剪是打开,那么所有 outtext 和 outtextxy 输出的正文串将在视区边界被裁剪。如果裁剪是关闭的,一旦有一部分正文串超出屏幕边界,这些函数放弃位图字体输出,而笔划字体输出将在屏幕边界上截断。

在图形软件包中有标准 8×8 位图字体,因此它在运行时刻总可用。笔划字体每个都分别保存在 .CHR 文件中,它们可以在运行时刻装入或转换为 .OBJ 文件(用 BGIOBJ 实用程序),并连接到用户 .EXE 文件上。

要决定给定正文串在屏幕上的长度,可调用 textheight (用像素表示串高度)和 textwidth (用像素表示宽度)。

一般地, settextstyle 为字体分配内存,然后从磁盘上装入适当的 .CHR 文件;同动态加载方案不同的是可以把一字符字体文件(或它们中的一些)直接连接到用户可执行程序文件中。要完成它,首先把 .CHR 文件转换为 .OBJ 文件(利用 BGIOBJ 实用程序),然后在源代码中(在调用 settextstyle 之前)设置对 registerbgifont 的调用来注册字符字体。用户在构筑程序时,对注册的笔划字体需用 .OBJ 文件连接。

注意:使用 registerbgifont 是一种高级的程序设计技术,初学者不宜使用。在本书附录 C.6 中有关于该函数的详细描述。

8.7.7 图形模式中的错误处理

这里仅对图形模式的错误处理函数作一个简介:

`grapherrormsg` 返回指定 `errorcode` 的出错信息串

`graphresult` 返回最后一次出错的图形操作的错误代码

如果在调用图形库函数时出现错误(如 `settextstyle` 要求的字体没有找到),就要置一内部错误代码。用 `graphresult` 来检索最后报告出错的图形操作的出错代码。表8.18是对出错返回代码的定义。

表8.18 出错代码及其相应的信息串

错 误 代 码	graphics_errors 常 量	相 应 出 错 信 息 串
0	<code>grOk</code>	无错
- 1	<code>grNoInitGraph</code>	(BGI) 图形没有安装(用 <code>initgraph</code>)
- 2	<code>grNotDetected</code>	图形硬件未找到
- 3	<code>grFileNotFound</code>	没找到设备驱动程序文件
- 4	<code>grInvalidDriver</code>	非法设备驱动程序文件
- 5	<code>grNoLoadMem</code>	没有足够的空间加载驱动程序
- 6	<code>grNoScanMem</code>	在扫描着色时超出内存
- 7	<code>grNoFloodMem</code>	在不规则着色时超出内存
- 8	<code>grFontNotFound</code>	没找到字体文件
- 9	<code>grNoFontMem</code>	没有足够的内存加载字体
- 10	<code>grInvalidMode</code>	选择驱动程序的非法图形模式
- 11	<code>grError</code>	图形错误
- 12	<code>grIOerror</code>	图形I/O错误
- 13	<code>grInvalidFont</code>	非法字体文件
- 14	<code>grInvalidFontNum</code>	非法字体号
- 15	<code>grInvalidDeviceNum</code>	非法设备号

调用 `grapherrormsg(graphresult)` 将返回上表所列的出错信息串。

出错返回代码只有在图形函数报告出错时才累计数目。当 `initgraph` 执行正确或调用 `graphresult` 时,出错返回代码复位为0。因此如果要知道哪个图形函数返回哪个错误,应把 `graphresult` 的值存到临时变量中,然后再测试它。

8.7.8 状态询问

本节先对图形模式的状态询问函数作一简介:

getarccoords	返回最后一次调用 arc 或 ellipse 时坐标的信息。
getaspectratio	返回图形屏幕的特征率。
getbkcolor	返回当前背景颜色
getcolor	返回当前绘图颜色
getfillpatten	返回用户定义的着色模式。
getfillsettings	返回当前着色模式和颜色的信息
getgraphmode	返回当前图形模式
getlinesettings	返回当前线类型、线模式和线宽度
getmaxcolor	返回当前最高的有效像素值
getmaxx	返回当前 x 分辨率
getmaxy	返回当前 y 分辨率
getmoderange	返回驱动器的模式范围
getpalette	返回当前调色板及其大小
getpixel	返回在 x, y 处像素的颜色
gettextsettings	返回当前正文字体、方向、大小和对齐方式
getviewsettings	返回当前视区的信息
getx	返回当前位置 cp 的 x 坐标
gety	返回当前位置 cp 的 y 坐标

在每一类 Turbo C 图形函数中，至少有一个状态询问函数。每个 Turbo C 图形状态询问函数取名为 get<something>（除了出错处理类）。有些没有参数，只是返回一个值来表示所要的信息，其他接受一指针作为参数，它指向 GRAPHICS.H 中定义的一个结构，用适当的信息填入结构，而不返回值。

图形系统控制类的状态询问函数是 getgraphmode 和 getmoderange，前者返回一整型值来表示当前图形驱动器和模式，后者返回由已知图形驱动器支持的模式的变化范围。

getmaxx 和 getmaxy 返回当前图形模式的最大 x 和 y 屏幕坐标。

绘图和着色状态查询函数为 getarccoords, getaspectratio, getfillpattern, getfillsettings 和 getlinesettings。getarccoords 用最后一次对 arc 和 ellipse 的调用中的坐标充填一个结构；getaspectratio 给出了当前模式的特征比率，图形系统用其来保证画圆的比率。getfillpattern 返回当前用户定义的着色模式。getfillsettings 用当前着色模式和着色颜色充填一结构。getlinesettings 用当前线类型充填一结构（实线、虚线），线宽（正常的或厚的）和线模式。

在屏幕和视区类中，状态查询函数为 getviewsettings, getx, gety 和 getpixel。当用户已定义了一个视区，则可以通过调用 getviewsettings 了解它的绝对屏幕坐标和裁剪是否激活，getviewsettings 用这些信息填充一个结构。getx 和 gety 返回了 cp（当前位置）的 x 和 y 坐标（视区相对值）。getpixel 返回指定像素的颜色。

图形模式正文输出函数类的状态查询函数：gettextsettings 函数用当前字符字形，正文显示的方向（水平或由底向上的垂直方向），字符放大因子和文本串对齐方式（水平和垂直的）填充一个结构。

Turbo C 的颜色控制种类包括三个状态查询函数。getbkcolor 返回当前的背景颜色，getcolor 返回当前绘图颜色。getpalette 用当前绘图调色板的大小以及调色板的内容填

充一个结构。getmaxcolor为当前图形驱动器和模式(调色板size-1)返回最高有效像素值。

本章最后介绍了Turbo C图形处理功能的基本概念和大部分函数的使用,附录B列出了这部分图形函数的详细信息。

此外,当在Turbo C程序中使用图形时,还必须连接graphics.lib,有关的内容可参看附录C.4的例子。

第九章 Turbo C 交互式编辑程序

Turbo C 编辑程序专用于产生程序文本。如果读者熟悉 Turbo Pascal, SideKick 编辑程序, 或 MicroPro WordStar 程序, 那么使用 Turbo C 编辑程序不会有困难, 因为 Turbo C 的编辑命令几乎与它们相同。

在第二章中, 我们简单介绍了 Turbo C 编辑程序的主要功能, 以便使读者开始时把精力集中在了解 Turbo C 系统本身和学习 Turbo C 的编程技术上。本章详细介绍 Turbo C 交互编辑程序的各种功能。最后一节将总结 Turbo C 与 WordStar 编辑命令的不同之处。

建立和修改源文件, 经常要用到编辑程序, Turbo C 的编辑程序灵活方便, 使用者应熟练掌握。

9.1 快速进入和退出编辑程序

要调用编辑程序, 可从 Turbo C 的主菜单中选择 **E**dit, 使编辑窗口变成活动窗口, 编辑窗口的标题呈高亮度显示, 光标置于编辑窗口中。

如要输入正文, 可像打字员一样敲键, 要结束一行按 Enter 键即可。

要从编辑程序中调主菜单, 按 F10, 此时编辑窗口的数据仍然留在屏幕上。

9.2 编辑窗口状态行

编辑窗口的状态行指出正在编辑文件的信息, 如文件中光标的位置, 编辑模式等。该状态行的形式如下:

Line Col Insert Indert Tab X:Filename.typ

各项的含义如下:

Line 指出光标所在的行号

Col 指出光标所在的列号

Insert 指示编辑程序正处在“插入模式”, 键盘上输入的字符被插到光标处。光标右面的正文向右移。使用 Ins 键或 Ctrl-V 可使编辑程序在“插入模式”与“重写模式”间选择。

在重写模式下, 键盘输入的正文在光标位置上重写, 而不是插在已有的正文之前。

Indert 指明自动缩进功能为开。可用命令 Ctrl-OT 或 Ctrl-QI 将其置为开或关。

Tab 指出是否可以插入制表键。使用 Ctrl-OT 可将其置成开或关。

X:Filename.typ 注明正在编辑的文件的驱动器名(X:), 文件名(Filename)和扩展名(.typ)。若为 NONAME.C, 说明还未指定文件名(NONAME.C 是 Turbo C 的默认文件名)。

9.3 编辑命令

Turbo C编辑程序大约有50条命令,用以移动光标,按页查看正文,查找并替换字符串等等。这些命令可分成以下四类:

(1) 光标移动命令(分为基本的和扩展的两组)

(2) 插入与删除命令

(3) 块命令

(4) 其他命令

表9.1摘要列出这些命令,表中的每一行包括命令定义,用于触发该命令的默认键。

表9.1 编辑程序命令

类别	功能	默认键
基本光标移动命令	字符左	Ctrl-S或Left
	字符右	Ctrl-D或Right
	字左	Ctrl-A
	字右	Ctrl-F
	上行	Ctrl-E或Up
	下行	Ctrl-X或Down
	上滚	Ctrl-W
	下滚	Ctrl-Z
	上一页	Ctrl-R或PgUp
	下一页	Ctrl-C或PgDn
快速光标移动命令	行头	Ctrl-QS或Home
	行尾	Ctrl-QD或End
	窗口头	Ctrl-QE
	窗口底	Ctrl-QX
	文件头	Ctrl-QR
	文件尾	Ctrl-QC
	块头	Ctrl-QB
	块尾	Ctrl-QK
	上次光标位置	Ctrl-QP
插入与删除命令	插入模式	Ctrl-V或Ins
	插入行	Ctrl-N
	删除行	Ctrl-Y
	删除至行尾	Ctrl-QY
	删除光标左边的字符	Ctrl-H或Backspace
	删除光标处的字符	Ctrl-G或Del
	删除光标右边的字符	Ctrl-T

块命令	标记块开始	Ctrl-KB
	标记块结束	Ctrl-KK
	标记单个字	Ctrl-KT
	复制块	Ctrl-KC
	删除块	Ctrl-KY
	隐藏/显示块	Ctrl-KH
	移动块	Ctrl-KV
	从盘中读块	Ctrl-KR
	将块写回磁盘	Ctrl-KW
其他	异常结束操作	Ctrl-U
	制表模式	Ctrl-OT
	制 表	Ctrl-L或Tba
	自动缩进和最佳填充	Ctrl-OI或Ctrl-QI
	定界符配对	Ctrl-Q, Ctrl-[或Ctrl-Q, Ctrl-]
	控制字符前缀	Ctrl-P
	查 找	Ctrl-QF
	查找并替换	Ctrl-QA
	查找标记	Ctrl-QN
	调用主菜单	F10
	装入文件	F3
	退出编辑, 不保存文件	Ctrl-KD或Ctrl-KQ
	重复上次查找	Ctrl-J
	恢复行	Ctrl-QL
	保存并编辑	Ctrl-KS或F2
	置标记	Ctrl-KN

下面详细介绍这些命令。

9.3.1 基本光标移动命令

编辑程序使用控制键在屏幕上、下、左、右移动光标, 下列命令控制光标在屏幕上的移动。

命 令 键	光 标 动 作
Ctrl-A	移到光标左边字的第一个字符
Ctrl-S	光标左移一格
Ctrl-D	光标右移一格
Ctrl-F	移到光标右边字的第一个字符
Ctrl-E	移上一行
Ctrl-R	移上一屏
Ctrl-X	移下一行
Ctrl-C	移下一屏
Ctrl W	屏幕下滚一行, 光标不动
Ctrl-Z	屏幕上滚一行, 光标不动

9.3.2 快速光标移动命令

编辑程序提供下列六条命令快速移动光标到行尾，文件的头或尾，上次光标位置。

命令键	光标动作
Ctrl-QS或Home	移到当前行的第一列
Ctrl-QD或End	移到当前行的尾端
Ctrl-QE	移到屏幕的顶部
Ctrl-QX	移到屏幕的底部
Ctrl-QR	移到文件第一个字符
Ctrl-QC	移到文件最后一个字符
(前缀Ctrl-Q加上字母B, K, P可以使光标跳到正文中的一些特殊点。)	
Ctrl-QB	将光标移到由Ctrl-KB设置的块开始标记;
Ctrl-QK	将光标移到由Ctrl-KK设置的块结束标记;
	这两个命令即使在该块未显示或块结束/开始标志未设置时也有效(参见“隐藏/显示块”)。
Ctrl-QP	移到上一命令前光标的最后位置，在查找或查找/替代操作执行后，希望光标回到执行前的最后位置时，这个命令特别有用。

9.3.3 插入和删除命令

建立程序，不仅要知道如何移动光标。同时，还应知道如何插入与删除正文。下列命令用来插入、删除字符、字和行。

插入模式开/关(Ctrl-V或者Ins)

输入正文时，使用Ctrl-V或Ins可选择两种基本的输入模式，即“插入”与“重写”中的一种模式。当前的模式显示在编辑程序的状态行中。

Turbo C编辑程序的默认模式是插入模式，它的功能是将新的字符插入到旧正文中。输入时，光标右面的正文向右移。

使用重写模式可用新的正文替代旧的正文，输入的字符将替代光标处的字符。

删除光标左面的字符(Ctrl-H或者Backspace)

将光标左移一格，并删除此处的字符，光标右面的字符均左移一格。可以使用这个命令清除换行符。

删除光标处字符(Ctrl-G或Del)

删除光标处字符且将光标右边的所有字符左移一格，这个命令不能跨行使用。

删除光标右面的字(Ctrl-T)

删除光标右边的字。一个字定义为以下列字符之一隔开的字符串。

空隔 < > , ; . ()

[] ^ _ * + - / \$

这个命令可跨行使用，而且可以删除换行符。

插入行(Ctrl-N)

在光标处插入一个换行符。

删除行(Ctrl-Y)

删除光标所在行，以下所有行上移一行。注意被删行无法恢复，所以，用这个命令时应小心。

从光标处一直删除到行尾(Ctrl-QY)

删除从光标位置到行尾的所有正文。

9.3.4 块命令

块命令同样要求一个控制字符命令串，正文块大小可以任意，可从一个字符到上百行，块应由特殊的块标记字符括起，一个文件中某一时刻只可以有一块。

标记一个块的时候，在块第一个字符前置一个块开始符，在块的最后一个字符后置一块结束符。一个块一旦被标识过，就可以复制、移动或删除，或将这块写到文件中去。

标记块开始(Ctrl-KB)

标记一个块的开始，该标记本身不可见，而且块也只有块结束符标上以后才可见。被标识的正文块显示时使用不同的辉度。

标记块结束(Ctrl-KK)

标记一个块的结束，该标记本身不可见，而且块只在块开始符也标上了才可见。

标记单个字(Ctrl-KT)

标记一个字作为一块，以替代块开始/块结束命令序列。若光标在一个字中，则这个字将被标记成一块；若光标不在一个字中，则光标左边的字将被标记成一块。

复制块(Ctrl-KC)

复制已标记的块到当前光标处，以前的块不变，标记移到新复制的块上，若没有标记的块或者光标在标记的块内，该命令无效。

删除块(Ctrl-KY)

删除已标记的块，删除的块不可恢复，所以，使用这个命令时应小心。

隐藏/显示块(Ctrl-KH)

置块标记为可见或不可见。块操作命令(复制、移动、删除和写入文件)仅当块显示时才起作用。和块有关的光标移动命令(跳到块的头/尾)不论块显示/隐藏均起作用。

移动块(Ctrl-KV)

将已标记的块从原来的位置移到光标处，块从原处消失，块标记留在新块中。若没有被标记，这个命令无效。

从磁盘中读块(Ctrl-KR)

将一磁盘文件作为一块读到当前正文的光标处，读入的部分标记为一个块，以不同辉度显示。

使用这个命令，Turbo C的编辑程序提示输入文件名，可以使用DOS通配字符选择一个文件，屏幕上将显示出一个目录。指定的文件可以是任何合法的文件名，若未指定文件类型(.C,.TXT,.BAK等)，编辑程序假设隐含指定了.C。读一个无扩展名的文件，需在文件名后加一句点。

将块写入磁盘(Ctrl-KW)

将已标记的块写入文件中，当前块不变，块标记也不变。若无被标记的块，这个命令无效。

使用这个命令时，Turbo C的编辑程序提示输出文件名。使用DOS通配符，可以选择一个文件进行重写，屏幕上的一个小窗口将显示出目录，若指定的文件已存在，则编辑程序发出一个警告并提示在重写前确认文件名，默认的扩展名为.C。写入一个无扩展名的文件，需在文件名后加一句点。

9.3.5 其他编辑命令

本节将按字母序介绍不能划入上述各类的命令。

异常终止操作(Ctrl-U)

可以中止任何正在等待输入的命令，例如当查找/替代命令询问 Replace Y/N?，或输入搜索串或文件名(读块和写块)时，可用该命令。

制表模式(Ctrl-OT)

制表模式为开时，在正文中按制表键大小分隔，若将制表模式置为关，则按上一行中各字的第一个字母加空格。

制表(Ctrl-I或Tab)

Options/Environment 菜单中的菜单项 Tab size 可用来决定编辑程序制表间隔的大小，其值可为2到16中的任一数字(缺省时为8)。

要改变一个文件中制表符被显示的形式，只需将制表宽度改为所希望的值，则编辑程序将在那个文件中，按所选择制表大小重新显示所有的制表符。新的制表宽度设置可保存在配置文件中(从Options菜单中选择 Store options 即可)。

当编辑程序Tab模式被关闭时，按Tab键插入足够的空格字符将光标移到下一个“软”制表符处。“软”制表符与当前行上面一行正文的每个单词的首字母对齐。

另外，当使用Ctrl-KW 指令从编辑程序向一个文件(或向PRN)送入一块有标记的正文时，编辑程序将视所有的制表字符为“硬”制表符。通常是每8列为一个制表符。然而，当用户用Ctrl-KP指令从编辑程序向打印机发送正文时，编辑程序视所有的制表字符为“软”制表符，并以适当数目的空格字符输出它们(等于使用Tab宽度所设置的制表大小)。

自动缩进和最佳填充(Ctrl-OI或Ctrl-QI)

在以后行中提供缩进。自动缩进为开时，按Enter键，光标不回到第一列，而回到刚结束行的开始列。

若想改变缩进长度，使用空格和←键可以移到新的列。当自动缩进状态为开时，在状态行显示有信息Indent。状态为关时，这个信息消失。自动换行状态默认为开(制表状态为开时，自动缩进不起作用)。

可用下列两种方式中的一种将编辑程序的自动缩进特征打开或关闭：

- (1) 当在编辑窗口时，键入Ctrl-OI或Ctrl-QI(同时按下Ctrl键和O或Q，再按I)；
- (2) 当在TCINST(Turbo C定做程序，见第十一章)时，选择建立环境，然后置自动缩进模式为打开状态。

当自动缩进模式和插入模式都被打开时，编辑程序自动地将新行缩进到与上面一行的首字符对齐。

在某些条件下,编辑程序将新缩进行的前面空白用制表字符和空格字符的最佳结合填充(一个最佳结合是指使用最少数目的字符数)。下面是最佳填充的必要条件:

(1) 自动缩进模式、插入模式和制表模式都为打开状态。

(2) 按下Enter键使光标从一个缩进行移到一个新的空行(编辑程序插入足够的空格字符使光标与上行的首字符对齐)。

(3) 光标仍在新行时,用户可以使用Left(左)和Right(右)光标键,Tab键,退格键和空格键在新行上水平地移动光标。

(4) 键入一个字符或命令,或移向另一行。

当上述情况出现时,编辑程序对新的一行前面的空白(或空格和制表字符)填以制表字符和空格字符的组合,以期用较少的字符产生同样数目的空白。

为了便于理解,下面举例说明。

[例1] Options/Environment菜单中Tab宽度被置为8(制表间隔列为1,9,17,25,...);自动缩进,制表和插入模式为打开状态;光标位于从第27列开始的行尾端。

- 按Enter键插入新行,编辑程序将光标定位于新行的第27列。

- 在不移动光标的情况下,在新的一行上输入字符。

- 编辑程序用三个制表字符(填到25列)和两个空格字符(填到27列)组成的五个填充字符填充新行的开始部分。

[例2] 如果在上例中,Tab宽度被置为5(即制表间隔列分别在1,6,11,16,21,26...),编辑程序将用五个tab字符(填到26列)和一个空白字符填充。

[例3] 如果Tab宽度被置为6(制表间隔列为1,7,13,19,25...)且用户在输入第一个字符前已将光标移至第18列,编辑程序将用两个制表字符(填到13列)和五个空格字符(填到第18列)来填充。

当自动缩进模式和插入模式都为打开状态(但Tab模式已被关闭)时,编辑程序仍然缩进新的一行并与前一行的开头对齐,但它只用填充空格字符(无制表符)来完成。

定界符配对(Ctrl-Q Ctrl-[或Ctrl-Q Ctrl-)]

当调试源文件时,源文件含有许多函数、括号表达式、嵌套注解和大量使用定界符的结构。包括如下符号:

花括号: { 和 } 注解号: /* 和 */

尖括号: < 和 > 双引号: " 和 "

圆括号: (和) 单引号: ' 和 '

方括号: [和]

对一个特殊的花括号寻找其配对是复杂的。假设现有一个含有许多嵌套子表达式的复合表达式,若用户想检查所有的括号是否已配对,或者当位于一个覆盖几个屏幕的函数的开始部分时,而想跳到函数的结束处,如果使用Turbo C的配对指令,可以很快得到结果。只需做下面几点:

(1) 将光标置于问题中的定界符上(如一些覆盖几个屏幕的函数的开始的花括号)。

(2) 对被选择的定界符的配对定位,只需按Ctrl-Q Ctrl-[(在上述例子中,配对应位于函数的结束处)。

(3) 编辑程序立即将光标移向所选择的定界符配对的那一个。如果它移到了所希望的配对处,则表明被调试代码并不包含那种类型的未配对定界符。如果它高亮度显示了错误的

定界符，则在程序中一定有错，这时须根据问题寻找错误。

关于配对的一些细节是：

(1) 实际上有两条“配对”编辑指令：一条用于向前配对，另一条用于向后配对，两指令为：

Ctrl-Q Ctrl-[配对(向前)

Ctrl-Q Ctrl-] 配对(向后)

(2) 编辑程序寻找注解定界符(/·和·/)的方式与其他寻找略有不同。¹

(3) 如果所选择的定界符没有配对，编辑程序将不移动光标。

为什么需要方向配对指令呢？因为既有一些有向的定界符，又有一些无向定界符，所以“方向配对”指令是必需的。例如，如果让编辑程序寻找一个开花括号({)或一个开方括号([)的配对，编辑程序知道配对定界符不会在所选择定界符的前面，因此它将向前寻找配对。开花括号和开方括号是有向的；编辑程序知道寻找配对的方向，因此它并不取决于是哪条“配对”指令。无论哪一条指令，编辑程序都将沿正确方向搜索。

同样地，如果让编辑程序寻找一个闭花括号(})或一个闭括号(])的配对，它将知道配对不会出现在所选择定界符的后面，因此它自动地向后搜索配对。因为这些定界符是有向的，所以它并不取决于所给的是哪一条“配对”指令；编辑程序总会沿正确的方向寻找。

然而，如果让编辑程序寻找双引号(")或单引号(')的配对，它并不知道搜索的方式。用户必须通过给出正确的“配对”指令说明寻找的方向。如果所给的是指令Ctrl-Q Ctrl-[，编辑程序将向前寻找配对；若为指令Ctrl-Q Ctrl-]，它将向后寻找配对。

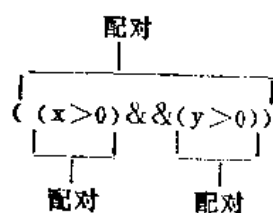
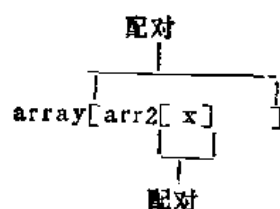
下面的表格总结了配对的定界符的搜索方向和嵌套情况(嵌套定界符在表格后面介绍)。

定 界 符 对	有 向	可 嵌 套
{ }	是	是
()	是	是
[]	是	是
< >	是	是
/* */	是	是和否
" "	否	否
' '	否	否

上面提到了嵌套定界符，什么是嵌套？简单地说，当为一有向定界符寻找配对时，编辑程序记下了搜索过程中所进入和离去的“定界符层”的路径。

可以用下列例子说明：

(1) 寻找方括号或圆括号的配对



(2) 搜索注解定界符

因为注解符是两个字符的定界符，当寻找配对并高亮度显示时须小心。在下列任何一种情

况下,编辑程序都只识别两个字符中的第一个字符:/*注解定界符的/部分或*/的*部分。如果将光标置于注解定界符的第二个字符上,编辑程序不知道要寻找什么,因此它根本不寻找。

另外,如前面表格中所示,注解符有时是可嵌套的,有时是不可嵌套的。(“是”和“否”)。这并不是不可捉摸或无法判定的,它取决于多条件的一个测试,与ANSI兼容的C程序不允许嵌套注解,但Turbo C提供了一个选择“嵌套注解”特征(Options/Compiler/Source菜单中的菜单项目嵌套注解),用户可以置为打开或关闭状态。在配对时它将影响注解定界符的嵌套性:

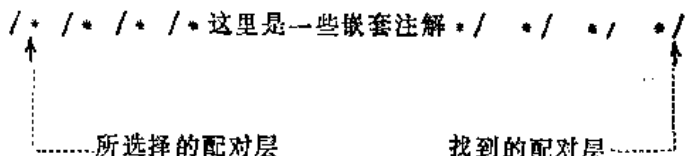
①如果嵌套注解为打开状态,编辑程序认为注解定界符是可嵌套的,并在寻找配对的过程中记下它所进入和离去的定界符层的路径。

②如果嵌套注解为关闭状态,编辑程序将不认为注解定界符是可嵌套的,当/*对被选择时,编辑程序所找到的第一个*/为其配对(反之亦同)。

如果在注解、引号或复合条件段中配对的定界符中出现同类型不配对的定界符,这将影响搜索。

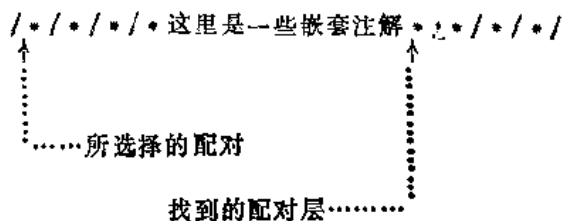
从下面的例子中可看出这些区别:

嵌套注解为打开状态——用^Q^[向前寻找

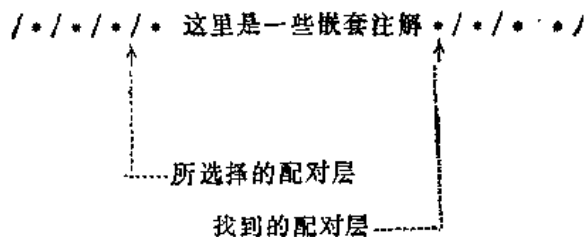


当嵌套注解为打开状态时,从“找到的”*/向后寻找将会得到所“选择”的/*。

嵌套注解为关闭状态——用^Q^[向前寻找



嵌套注解为关闭状态——用^Q^[向后寻找



控制字符前缀(Ctrl-P)

在控制字符前按Ctrl-P,可以将控制字符输入到文件中,方法是,先按Ctrl-P,然后再键入所需的控制字符,控制字符以低亮度大写在屏幕上显示(或者相反,由屏幕设置而定)。

查找(Ctrl-QF)

用于查找30个字符以下的串。键入这个命令以后,状态行清除,编辑程序提示输入被查找串。键入欲查找的串,然后按Enter键。

查找串中可以包含任何字符,包括控制字符,在键入查找串中的控制字符时,应加上前缀^P,例如,输入Ctrl-T时应在按P的同时按下Ctrl键,然后按T,还可以在查找串中放

入换行符，只要按下 Ctrl-MJ(回车/换行)。注意，Ctrl-A有特殊意义：它可以代表任何字符，可作为查找串中的通配符。

可以使用字符左、字符右、字左、字右命令编辑查找串，字右将恢复以前的查找串，然后编辑。欲中止查找操作可以使用异常终止命令(Ctrl-U)。

指定了查找串后，Turbo C编辑程序还要求输入查找选择项，可以使用下列查找选择项：

B 从当前光标位置向后搜索，直到正文的开头。

G 不管光标处于何处，全面搜索正文，它停在最后一次出现串的位置上。

N 从光标位置起，搜索下一个该串出现的位置，当N与G同时使用时，G选择项有效，N选择项无效。

n 从光标位置起，搜索第 n 个该串出现的位置。

u 忽略大/小写的区别。

W 只查整个字的匹配，跳过在其他字中的匹配模式。

例子：

W 只查整个字的匹配，例如，查找串 term 只与 term 匹配，不与 terminal 匹配。

BU 向后搜索，忽略大小写的区别。Block 将与 blockhead 和 BLOCKADE 等匹配。

125 找到查找串的第 125 次出现。

键入 Enter 可结束查找选择项表(若有的话)，于是查找开始。若正文包含有与查找匹配的目标，编辑程序将光标放在该目标处。查找操作可以重复，只要使用重复上次查找命令(Ctrl-L)。

查找并替换(Ctrl-QA)

这个操作与查找命令一样，只不过可以用任意的至多 30 个字符的字符串来替代找到的串。注意 Ctrl-A 在查找串中是作为通配符使用，它在替代串中没有任何特殊含义。

一旦指定了查找串，编辑程序将要求输入替代串；可键入至多 30 个字符，控制字符的输入和编辑与查找命令中一样。若仅键入 Enter，编辑程序删除找到的目标。

它的选择项除与查找命令相同外，还包括：

N 替换时不用用户确认，在每一个查找串的出现处均直接替换。

n 替换查找串的 n 次出现，若使用 G 选择项，查找从文件头开始，否则，从当前光标处开始。

例如：

N10 查找下 10 个查找串的出现。不用用户确认。

GW 在全部正文中查找并替代整个字的出现，忽略大小写之分，并请求确认。

GNU 在全文中查找查找串的每一次出现，不区分大小写，不用用户确认。

同样，按 Enter 键结束选择项表(若有的话)，查找/替换操作开始。若编辑程序发现了目标(若选择项中没有 N)，编辑程序将光标置于目标一端，并在屏幕顶部的提示行问 Replace (Y/N)? 这时可使用 Ctrl-U 中止查找/替换操作，也可以使用重复上一查找命令(Ctrl-L)以重复查找/替换操作。

查找标记(Ctrl-QN)

在正文中查找四处标记(0—3)，可将光标移到以前置的标记，只要按 Ctrl-Q 与标记序号 n。

装入文件(F3)

用于编辑一已有文件或产生新文件。

退出编辑, 不保存文件(Ctrl-KD或Ctrl-KQ)

退出编辑返回到主菜单。如果有文件, 可在文件命令下使用主菜单的 Save 选择, 显式地将已编辑的文件存于磁盘, 也可在编辑时使用 Ctrl-KS 或 F2。

重复上一次查找(Ctrl-L)

重复上一次查找或查找/替换操作。

恢复行(Ctrl-QL)

只要还未离开本行, 可以恢复该行以前的状态。不管该行作过多少修改。

保存文件(Ctrl-KS或者F2)

存文件, 不退出编辑程序。

设置标记(Ctrl-KN)

在正文中可设置四处标记, 只要先按Ctrl-K再按一个数字n(0—3)。标记设置后, 在其他地方编辑时, 均可用Ctrl-QN命令方便地返回到标记的位置。

9.4 Turbo C编辑程序与WordStar之比较

Turbo C的一些编辑命令与WordStar稍有不同。尽管Turbo C编辑程序仅为WordStar命令的一个子集, 但它仍增加了几个WordStar中没有的成份以方便源代码的编辑, 主要的差别是:

自动缩进 Turbo C的编辑命令 Ctrl-QI 可以置自动缩进特征为开或关。

回 车 在Turbo C中, 重写模式下不能在文件的末尾键入回车(若插入模式为关时在一行的尾按Enter, 编辑程序不会插入回车符或将光标移到下一行)。为了插入回车符, 可以打开插入模式或在重写模式下使用Ctrl-N。

光标移动 Turbo C的光标移动命令——Ctrl-S, Ctrl-D, Ctrl-E 和Ctrl-X可以在屏幕上自由地移动而不跳到空行的第一列, 这并不意味着屏幕上充满空格, 相反, 所有行尾空格均自动被清除。这种移动光标的方法对程序的编辑十分有用, 例如, 当匹配锯齿形的语句时就很有用。

左 删 除 Word Star命令 Ctrl-Q Del 从光标处删除到行开始处, Turbo C中无此命令。

标记字为块 Turbo C允许使用Ctrl-KT标记一个字为一块, 这比WordStar中的方法更方便(WordStar中仍采用两步标识字的头和尾)。

移动穿过换行符 Ctrl-S 与 Ctrl-D 不能穿过换行符, 从一行到另一行必须使用 Ctrl-E, Ctrl-X, Ctrl-A或Ctrl-F。

退出编辑 Turbo C中的 Ctrl-KQ 与 WordStar 中的 Ctrl-KQ(退出编辑)不一样。在Turbo C中, 修改后的正文没有丢弃, 而是放在内存中, 可用于编译与存储。

恢 复 Turbo C的Ctrl-QL命令当光标尚未离开一行时可恢复该行编辑以前的内容。

更改磁盘文件 由于Turbo C编辑过程全在储存器中进行, Ctrl-KD命令不改变磁盘上的文件。必须在文件菜单中显式地使用选择Save, 或在编辑程序中使用 Ctrl-K或F2, 以修改磁盘中的文件。

第十章 Turbo C 命令行

第二章介绍的Turbo C集成开发环境为程序开发和运行提供了一个极为友善的用户界面，这也是Turbo C能够吸引用户的魅力之一。

Turbo C不仅提供了集成开发环境，还提供了一个标准的命令行编译程序版本。

本章将首先按选择项类型的字母顺序介绍Turbo C命令行选择项，并描述每个选择项的功能。然后介绍在命令行下如何编译和连接C程序。

命令行选择项可分为以下三个基本类型：

- (1) 编译选择项
- (2) 连接选择项
- (3) 环境选择项

10.1 编译选择项

编译选择项又分为：

- (1) 存储模式选择项
- (2) 定义(宏定义)选择项
- (3) 处理器(CPU)选择项
- (4) 源选择项
- (5) 代码选择项
- (6) 出错报告选择项
- (7) 命名(段命名)选择项
- (8) 编译控制选择项

大多数命令行选择项在Turbo C集成开发环境中有所对应的选择项。通过表10.1可以看到命令行选择项与菜单选择项的对应关系。

10.1.1 存储模式选择项

- mc 编译时采用紧凑(compact)存储模式
- mh 编译时采用特大(huge)存储模式
- ml 编译时采用大(large)存储模式
- mm 编译时采用中(medium)存储模式
- ms 编译时采用小(small)存储模式(系统默认值)
- mt 编译时采用极小(tiny)存储模式，采用此模式所产生的代码几乎与小(small)模式下的代码相同，但在连接时，使用C0T、OBJ产生一个极小(tiny)模式的可执行程序。

表10.1 命令行选择项和菜单选择项的对应关系

命令行选择项		菜单选择项
-A		O/C/Source/ANSI Keyword only...On
-a		O/C/Code generation/Alignment...Word
-a-	**	O/C/Code generation/Alignment...Byte
B		没有对应的菜单选择项
-C		O/C/Source/Nested Comments...On
-c		Compile/Compile to OBJ
-Dname		O/C/Defines
-Dname=string		O/C/Defines
-d	**	O/C/Code generation/Merge duplicate strings
-efilename		Project/Project name
-f	**	O/C/Code generation/Floating point...Emulation
-f-		O/C/Code generation/Floating point...None
-f87		O/C/Code generation/Floating point...8087
-G		O/C/Optimization/Optimize for...Speed
-g#		O/C/Errors/Warnings,stop after...#
-lpathname		O/D/Include directories
-i#		O/E/Identifier length...#
-j#		O/C/Errors/Errors,stop after...#
-K		O/C/Code generation/Default char type...Unsigned
-K-	**	O/C/Code generation/Default char type...Signed
-Lpathname		O/D/Library directories
-M		O/L/Map file
-mc		O/C/Model...Compact
-mh		O/C/Model...Huge
-ml		O/C/Model...Large
-mm		O/C/Model...Medium
-ms	**	O/C/Model...Small
-mt		O/C/Model...Tiny
-N		O/C/Code generation/Test stack overflow...On
-npathname		O/D/Output directory
-O		O/C/Optimization/Optimize for...Size
-ofilename		没有对应的菜单选择项
-P		O/C/Code generation/Calling convention...Pascal
-P-	**	O/C/Code generation/Calling convention...C
-r	**	O/C/Optimization/Use register variables...On
-S		没有对应的菜单选择项
-Uname		没有对应的菜单选择项
-w		O/C/Errors/Display warnings...On
-w-		O/C/Errors/Display warnings...Off
-wxxx		O/C/Errors/Portability warnings or ANSI violations or Common errors or Less common errors...On

-w-xxx	O/C/Errors/Portability warnings or ANSI violations or Common errors or Less common errors...Off
-Y	O/C/Code generation/Standard stack frame...On
-y	O/C/Code generation/Line numbers...On
-Z	O/C/Optimization/Register optimization...On
-zAname	O/C/Names/Code/Class
-zBname	O/C/Names/Data/Class
-zCname	O/C/Names/Code/Segment
-zDname	O/C/Names/BSS/Segment
-zGname	O/C/Names/Data/Group
-ZPname	O/C/Names/Code/Group
-zRname	O/C/Names/Data/Segment
-zSname	O/C/Names/BSS/Group
-zTname	O/C/Names/BSS/Class
-l	O/C/Code generation...80186/80286
-l- **	O/C/Code generation...8088/8086
O = Options C = Compiler E = Environment D = Directories ** = 默认为ON	

七

10.1.2 定 义

-Dxxx 把标识符xxx, 定义成为只有单个空格字符()的字符串。

-Dxxx=string 把标识符xxx定义为等号后的字符串。字符串不包含空隔和列表字符。

该选择项可以定义多个符号, 它在命令行中的语法为:

-Dsymbol [= string][, symbol [= string], ...]

其中, 参数symbol是一个标识符, 用户可以为它定义一个值(例如: -Dtime = year)若, 未给symbol赋值, 如-Dxxx, 则Turbo C将它定义为只有单个空格字符的字符串。

在命令行中, 使用该选择项时可有以下几种方式:

(1) 单个-D选择项, 定义多个符号, 两两间用分号隔开。如

-Dxxx, yyy = 1, zzz = NO

(2) 多个-D选择项, 如-Dxxx, -Dyyy = 1 -Dzzz = NO

(3) 混合方式, 如-Dxxx, -Dyyy = 1 -Dzzz = NO

编译程序将搜索所有列出的符号定义, 搜索顺序从左到右。

-Uxxx 取消标识符xxx原有的定义。

五

10.1.3 处理器选择项

-1 使Turbo C产生扩充的80186指令。当在未保护模式(如IBM PC AT的MS DOS3.0)下运行生成的80286程序时, 也可使用该选择项。

-a 将整型区域项调整到一个机器字边界, 多余字节被插入到一个结构中以确保字段调整的安全性。自动变量和全局变量也作适当调整。char型、unsigned char型的变量和字段可放在任何地址处, 其他变量和字段必须放在偶数地址处。

- f87 用内部8087指令而不是通过8087仿真库函数进行浮点操作。这就指明浮点协处理器在运行时是必须的。所以根据此选择项编译的程序在没有浮点处理器芯片的机器上将不能运行。
- f 如果运行系统没有8087芯片,则在运行时就仿真8087调用。如果有8087芯片,则调用8087进行浮点运算(系统默认)。
- f- 指明程序不包含浮点运算,因此连接时不需与浮点库文件连接。

10.1.4 源选择项

- A 建立与ANSI兼容的代码,Turbo C扩充的关键字视为一般的标识符,这些关键字包括:
near far huge cdecl asm pascal interrupt _cs _ds _es _ss
以及伪寄存器变量如-AX, -BX, -SI等等。
- C 允许注解嵌套,通常注解是不可嵌套的。
- i# 使编译程序仅识别标识符的前#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#个字符不同,则认为是不同的标识符。该选择项缺省时,Turbo C允许标识符使用32个字符。其他系统,如UNIX,只识别前8个字符,如果要把它们移植到别的环境,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。
- K 使编译程序在处理字符说明时,认为它们是unsigned char类型。这就与其他把字符说明作为unsigned char类型处理的编译程序相容,该选择项缺省时,字符说明是signed类型(即是有符号的)。

10.1.5 代码选择项

- d 当两个字符串匹配时,把两者合并,产生较小的程序(系统默认)。
- G 使编译程序在可能的情况下,建立较快的可执行程序,但不一定是较小的程序。
- N 在每个函数的入口处,产生栈溢出逻辑,即一旦发现栈溢出就产生溢出信号。这将使程序长度增加,速度变慢。但该选择项在调试程序时非常有用,因为栈溢出是很难发现的。一旦发现栈溢出,则显示信息“Stack overflow”,程序以出口码1退出。
- y 指出目标文件中符号调试程序使用的行号,这会增加目标文件的长度,但不影响可执行程序的大小和运行速度。该选择项仅与符号调试程序结合使用才有效。
- O 通过消去多余的转移而优化程序长度,并重新组织循环和switch语句。
- P 使编译程序产生的所有子程序调用和函数调用均使用pascal参数传递序列,这使函数调用既短又快。函数调用时,传递参数的数量和类型必须正确,不能像一般C语言允许函数的参数个数是可变的。可用cdecl语句来取代该选择项,把函数调用明确地说明成C参数传递序列。
- r- 禁止使用寄存器变量。使用该选择项时,所有register关键字一概无效,如果有一些汇编语言代码,它们没有保护SI和DI寄存器的值,-r-选择将允许用从Turbo C调用此代码。禁止用寄存器变量虽然降低了所生成代码的效率,但在使

用已有的子程序时是必要的。注意,当使用 `-r-` 选择项时,以 `-r-` 为选择项编译的文件能调用未用 `-r-` 为选择项编译的库文件中的代码,例如调用库文件中的子程序(代码)。反之却不成立。因此带 `-r-` 选择项编译的文件只能被带 `-r-` 选择项编译的文件调用。

- r 可以使用寄存器变量(系统默认)。
- Y 产生标准栈框架,这在使用调试程序了解栈的情况时非常有用。
- Z 通过记住寄存器内容并尽可能多使用它们,以避免多余的存取操作。

注意要谨慎使用该选择项,因为编译程序无法知道是否已通过一个指针间接改变了变量的值而使寄存器的值无效。

例如,变量A装入DX寄存器,它就被保存。如果A后来被赋予一个值,DX中的值就清除并指出其值不再是当前值。遗憾的是。如果A的值间接地被修改(通过一个指向A的指针赋值),Turbo C将不会得到修改后值,仍继续记住DX中A的值(已经是旧值)。

-Z选择项用于避免重复装入已在寄存器里的值,这可以减少指令并且能把访问存储器的指令转变成访问寄存器的指令。下面的例子描述了使用该选择项的好处和缺陷,并说明使用该选择项时为什么要谨慎从事。

C源代码	优化汇编程序
func()	
{	
int A,*P,B;	
A = 4;	mov A,4
...	
B = A;	mov ax,A
	mov B,ax
P = &A;	lea bx,A
	mov P,bx
*P = B+5;	mov dx,ax
	add dx,5
	mov [bx],dx
printf("%d\n",A);	push ax
}	

首先注意语句 `*P = B + 5`,产生的代码使用了一个从ax移到dx的动作,若不使用 `-Z` 选择项优化,则将从B移到dx,产生一个较长较慢的指令。

其次,从对P的赋值看出,*P已经在bx中,因而在加法指令后,从P到bx的移动就被除去,这些改进不仅无害而且很有作用。

然而,对 `printf` 的调用是不正确的。Turbo C发现ax包含了A的值,因而把它送入栈中,而没有把存储器中正确的结果送入栈。`printf` 输出结果是4而不是正确结果9,A的值已通过P的间接赋值发生了变化。

如果语句 `*P = B + 5` 写成 `A = B + 5`,则Turbo C就能知道A的值发生了变化。

当调用函数或者指针转移到某个地方(例如标号,情况语句,循环的开始和结束),寄

寄存器的内容都认为丢失。由于这种限制和8086处理器中的寄存器数量很少，多数程序使用这种优化是不会出现错误的。

10.1.6 出错选择项

- g 产生#个信息后停止编译（#代表警告和出错信息）
- i# #个错误信息发生后停止编译。
- wxxx 给出由xxx指出的警告信息。选择项-w-xxx禁止产生xxx指出的警告信息。
- w-xxx可取下列值，

违背ANSI

- wdup 'xxxxxxxx'的重新定义不唯一
- wret 既使用返回又使用值的返回
- wstr 'xxxxxxxx'不是结构的部分
- wstu 未定义结构'xxxxxxxx'
- wsus 可疑指针转换
- wvoi void类型的函数不可以返回一个值
- wzst 结构长度为零

非常见错误

- waus 'xxxxxxxx'被赋予一个未使用过的值
- wclef 在'xxxxxxxx'定义前可能使用了'xxxxxxxx'
- weff 代码无效
- wpar 参数'xxxxxxxx'从未使用
- wpia 可能不正确赋值
- wrch 不可达代码
- wrvl 函数应该返回一个值

常见错误

- wamb 二义性运算符需要括号
- wamp 函数或数组中有不必要的
- wnod 函数'xxxxxxxx'未说明
- wpro 调用无原型函数
- wstv 结构按值传送
- wuse 'xxxxxxxx'有说明但从未使用

移植性警告

- wapt 对不可移植指针赋值
- wcln 常量太长
- wcpt 不可移植指针比较
- wdgn 比较时常量越界
- wrpt 不可移植返回类型转换

- wssig 转换时可能丢失了有效数字
- wucp 混淆了指向有符号字符和无符号字符的指针

10.1.7 命名选择项

- zAname 改变代码段类名为 name,缺省时,代码段被赋予类 CODE。
- zBname 改变未初始化数据段类名为 name,缺省时,未初始化数据段被赋予类 BSS。
- zCname 改变代码段名为 name,缺省时,除了中、大、特大存储模式外,代码段名为 TEXT,在中、大、特大模式下,代码段名为 filename-TEXT (其中 filename 是一个源文件名)。
- zDname 改变未初始化数据段名为 name,缺省时,除了在特大 (huge) 模式中不生成未初始化数据段外,其它模式下未初始化数据段命名为 BSS。
- zGname 改变未初始化数据段组的名为 name,缺省时,数据组命名为 DGROUP,这不包括特大模式中无数据组的情况。在特大模式中该选择项不起作用。
- zPname 使得产生带代码组的输出文件的代码段名为 name,该选择项不应在极小模式中使用。
- zRname 把初始化数据段名置为 name,缺省时,初始化数据段命名为 DATA,而在特大模式中该段命名为 filename_DATA。
- zSname 把初始化数据段组名改为 name,缺省时,数据组命名为 DGROUP。因在特大模式中无数据组,因此,该选择项在特大模式中不起作用。
- zTname 把初始化数据段类命名为 name,缺省时初始化数据段类命名为 DATA。
- zX* 对 X 使用默认名:例如, -ZA* 使用默认名 CODE 作为代码段名。

10.1.8 编译控制选择项

- B 编译和调用汇编程序处理插入汇编代码。
- c 编译和汇编名为 .C 和 .ASM 的文件,但不执行连接命令。
- ofilename 把给定的源文件编译成指定的目标文件 filename.obj。
- S 编译指定的源文件并产生汇编语言输出文件 (.ASM),但不进行汇编。

10.2 连接选择项

filename 根据文件名 filename 加 .EXE 生成可执行程序名 (程序名即变为 FILENAME.EXE), 文件名必须紧跟在 -e 后, 不允许有间隔, 该选择项缺省时, 用文件名表中的第一个源文件或目标文件名作为 filename。

- M 使连接程序产生一个完全连接 map 文件, 缺省时不产生连接的 map 文件。

10.3 环境选择项

- Idirectory 搜索 directory, 即指出含有包括文件的驱动器或子目录路径名, 驱动器说明是一个大写的或小写的单个字母, 后跟一个冒号 (:), 目录是

目录文件的任何有效路径名。

- Ldirectory 使连接程序从给定目录中取得C0X.OBJ启动目标文件和Turbo C库文件(CX.LIB, MATHX.LIB, EMU.LIB及FP87.LIB)。缺省时, 连接程序在当前目录查找这些文件。
- nxxx 把由编译程序建立的.OBJ或.ASM文件放入由路径xxx指出的目录或驱动器中。

值得注意的是, -L, -I选择项可以支持多个库目录和包括文件目录, 它们在tcc命令行中的语法为:

库目录 -Ldirname[, dirname, ...]

包括目录 -Idirname[, dirname, ...]

其中参数dirname可为任意的有效目录路径名。

用户在命令行中输入多目录时可以有以下几种方式:

- (1) 用单个-L, -I选择项, 列出多个目录, 两两间用分号隔开, 如

-Ldirname1, dirname2, dirname3 -linc1, inc2, inc3

- (2) 多个-L, -I选择项, 每次列出一个目录, 如

-Ldirname1 -Ldirname2 -Ldirname3 -linc1 -linc2 -linc3

- (3) 混合方式, 如

-Ldirname1, dirname2 -Ldirname3 -linc1, inc2 -linc3

当命令行中列出多个-L, -I选择项或有多个目录名时, 编译程序和连接程序将搜索所有列出的目录, 搜索顺序从左到右。

10.3.1 隐式库文件和显式库文件

Turbo C识别两种类型的库文件, 隐式库文件和用户说明的库文件, 后者也称为显式库文件。

隐式库文件是Turbo C自动进行连接的库文件。它们是CX.LIB文件, EMU.LIB或FP87.LIB, MATHX.LIB和启动目标文件(C0X.OBJ)。

用户说明的库文件是显式列在命令行或工程文件中的库文件, 这些库文件均以.LIB为扩展名。

10.3.2 库文件的搜索算法

Turbo C搜索库文件的方法是:

- (1) 隐式库 Turbo C只在指定的库目录中寻找隐式库。

- (2) 显式库 Turbo C搜索显式(用户指定的)库的方法取决于所列出库文件名的方式。

- 如果列出的显式库文件未指定驱动器或目录(如mylib.lib), 则Turbo C首先在当前目录中寻找, 然后(如果第一次搜索不成功)在指定的库目录中寻找。

- 如果用户说明库指定了驱动器和目录(如: C:\mystuffmylib.lib), 则Turbo C仅在显式列出的路径下搜索, 而不在指定的库目录中寻找。

10.4 从命令行直接编译和连接Turbo C程序

Turbo C 程序不仅可在集成开发环境中运行,也可在命令行方式下运行。虽然集成开发环境很适合于程序的开发和运行,但人们有时愿意使用命令行。在一些高级程序中,命令行方式也许是解决错综复杂问题的唯一方法。例如, Turbo C 程序中含有插入汇编代码时,就需要用 Turbo C 的命令行版本 TCC,而不能使用集成开发环境版本 TC。

TCC 可编译 C 语言源文件,并把其连接成可执行文件。它的工作方式与 UNIX CC 命令相似。TCC 也可调用 MASM 汇编程序,以汇编 .ASM 源文件。如果仅仅是编译源文件,只须在命令行中使用 -c 选择项。

10.4.1 命令行的一般格式

用命令行方式执行 Turbo C, 只需在 DOS 提示符下打入 tcc 和命令行参数即可, 命令行参数包含了编译和连接的选择项以及文件名。一般格式为:

tcc[选择项 选择项 选择项 ……] 文件名 文件名 ……

10.4.1.1 TCC命令行的选择项

键入一个短横线 (-) 紧跟一个选择项字符就可设置命令行开关 (如 -I), 每一个选择项应与 tcc 命令、其他选择项及选择项之后的文件名之间用空格隔开。在选择项字符后再加一个短横线就关闭这个选择项, 例如: -A 把 ANSI 关键字选择项设置成“开”(on), -A- 就将此选择项设置成“关”(off)。这种特性对于要把某个任选项设置成 on 或 off 非常有效。利用此特性可在命令行中设置一些选择项, 取代配置文件中相应的设置。

10.4.1.2 TCC命令行的文件名

在一系列编译和连接的选择项之后是文件名。

编译程序按下列规则编译源文件:

filename 编译 filename.c
filename.c 编译 filename.c
filename.xyz 编译 filename.xyz
filename.obj 连接时的目标文件
filename.lib 连接时的库文件
filename.asm 调用 MASM 将之汇编成 OBJ 文件

编译程序启动连接程序并为连接程序提供适当的 C 语言启动文件名和标准 C 语言库文件名。

10.4.2 可执行文件的产生

一般地, 编译程序根据命令行中提供的第一个源文件名或目标文件名加扩展名 .EXE 组成可执行文件名。

如果要另取可执行文件名, 需使用 -e 选择项, 在 tcc 命令之后文件名之前, 输入 -e, 接着输入希望的可执行文件名 (在 e 和文件名之间不可有空格), 这样就能给执行文件重新命名。

10.4.3 有关命令行的一些例子

下例说明了在DOS命令行中执行tcc时的语法。

```
tcc -IB; \include -LB; \lib -etest start.c body.obj end
```

其中, 命令tcc在DOS提示符下调用Turbo C, Turbo C对各命令行选择项作如下解释:

- include子目录为B; \INCLUDE (-IB; \include)
- 库文件在B; \LIB子目录中 (-LB; \lib)
- 产生的可执行文件命名为TEST.EXE (-etest)

Turbo C 在产生可执行文件过程中对所列文件名的解释为:

- 编译源文件START.C;
- 连接时要包括的目标文件为BODY.OBJ,
- 另一个要编译的源文件为END.C。

下面, 再看一个Turbo C命令行的例子:

```
tcc -IB; \include -LB; \lib2 -mm -c -k s1 s2.c z.asm mylib.lib
```

执行时命令行告诉Turbo C:

- 在B; \INCLUDE目录下寻找包括文件 (-IB; \include)
- 在B; \LIB2目录下寻找库文件 (-LB; lib2)
- 使用中存储模式 (-mm)
- 允许嵌套的注解 (-c)
- 字符为unsigned类型 (-k)

Turbo C 对所列出的文件名解释为:

- 需编译的源文件是 S1.C 和 S2.C
- 用 MASM 汇编程序汇编文件 Z.ASM
- 生成的可执行文件命名为 S1.EXE
- 连接时, 需连接的库文件是 MYLIB.LIB

下面是另外一个含有多库目录(-L)和包括目录(-I)的例子。

(1) 建立TCC.EXE于C:\TURBOC目录下, A驱动器的当前目录为A:\ASTROLIB。

(2) 包括文件所在目录为C; \TURBOC\INCLUDE。

(3) 标准启动文件(C0T.OBJ, C0S.OBJ, ..., C0H.OBJ)在C; \TURBOC\STARTUPS目录中。

(4) 标准Turbo C库文件(CS.LIB, CM.LIB, ..., MATHS.LIB, MATHM.LIB, ..., EMU.LIB, FP87.LIB, 等等)在C; \TURBOC\LIB目录中。

(5) 用于星系统的用户库文件用(TLIB建立和管理的)位于C; \TURBOC\STARLIB目录中, 其中一个库为PARX.LIB。

(6) 为类星体生成的第三部分文件位于A驱动器的A; \ASTROLIB目录中, 其中一个库为WARP.LIB。

在该配置下, 可输入下列 tcc 命令行:

```
tcc -mm -lstarups, lib, tarlib -linclude orion vmaj parx.lib  
a:\astrolib\warp.lib
```

tcc 将 ORION.C 和 VMAJ.C 编译成 .OBJ 文件, 然后与中模式启动代码 (C0M.OBJ) 中模式库 (CM.LIB, MATHM.LIB)、标准的浮点仿真库 (EMU.LIB) 以及用户说

明的库 (PARX.LIB和WARP.LIB) 进行连接, 产生一个名为ORION.EXE的可执行文件。

编译程序将在C:\TURBOC\INCLUDE目录中搜索用户源代码中的包括文件; 在C:\TURBOC\STARTUPS目录中搜索启动代码(当找到时停止); 在C:\TURBOC\STARTUPS目录中寻找标准库, 若在此目录中未找到, 则再在C:\TURBOC\LIB目录中寻找(在此找到, 停止搜索)。

当搜索用户指定的库PARX.LIB时, 编译程序首先在当前目录C:\TURBOC中寻找, 若未找到, 则编译程序继续搜索的次序是: 首先是C:\TURBOC\STARTUPS, 然后是C:\TURBOC\LIB, 然后是C:\TURBOC\STARLIB(在此可找到PARX.LIB)。

对于库WARP.LIB, 给出了一个显式路径(A:\ASTROLIB\WARP.LIB), 故编译程序仅在那个目录下搜索。

10.5 TURBOC.CFG文件

在配置文件TURBOC.CFG中设置一系列选择项, 可增加命令行中选择项的个数, 配置文件中选择项的使用就像从命令行中打入它们一样。

可用任何标准ASCII字符编辑程序或文字处理程序建立TURBOC.CFG文件。文件中的选择项可列成一行(用空格分开)也可列成多行。这样, 当在命令行下编译程序时, Turbo C把TURBOC.CFG中选择项与命令行中选择项结合在一起使用。

启动TCC时, 首先在当前目录下寻找TURBOC.CFG, 如果没找到且是在DOS3.X下运行, 则到启动目录(即TCC.EXE所在目录)下寻找。注意, 该配置文件不同于TCCONFIG.TC文件, TCCONFIG.TC是集成开发环境版本内含的配置文件。

应该注意的是, 命令行中的选择项覆盖TURBOC.CFG中相同的选择项。例如, 配置文件中含有几个选择项, 其中有-a选择项(该选择项又不需使用), 这时可继续使用配置文件, 但必须在命令行中列入-a-取代-a选择项。

命令行选择项和TURBOC.CFG中选择项是怎样组合和覆盖的呢? 一般地, 把TURBOC.CFG中所列选择项分成两部份, 一是-I和-L选择项, 二是文件中其他选择项, -I和-L选择项加入到命令行选择项的右边, 其余的TURBOC.CFG选择项都插入到命令行选择项的左边(紧接着TCC)。

命令行选择项是从左到右进行检验的。因此选择项重复出现时, 左边的选择项总是重复左边的选择项。根据命令行选择项和TURBOC.CFG选择项的组合方法, TURBOC.CFG中-I和-L选择项在最右边, 而命令行中描述的include目录和库目录是Turbo C寻找包括文化和库文件时首先使用的目录, 因此, 命令行中-I和-L指出的目录先于配置文件中的-I和-L选择项指出的目录。

10.6 在DOS下直接运行Turbo C程序

在DOS下运行Turbo C可执行程序, 只要在提示符下打入可执行程序文件名即可, 不必打入扩展名.EXE。例如, 要运行程序TEST.EXE, 只需在DOS提示符下打入test并按Enter键就可运行TEST.EXE程序。

第十一章 Turbo C 的用户定做

本章讨论Turbo C软件包中的Turbo C定做程序(TCINST.EXE)的功能和使用。

11.1 定做程序 TCINST 的功用

TCINST是Turbo C的定做程序(或“装配”程序),用它来组装TC.EXE,即Turbo C的集成开发环境。可以用TCINST来改变TC操作环境中的各种缺省设置值,如屏幕大小、编辑模式、菜单颜色和缺省目录等。TCINST可以改变用户操作Turbo C的环境,它可以直接修改TC.EXE副本中某些缺省值。

使用TCINST可以完成下列操作:

- 建立嵌入文件、库文件、配置文件、帮助文件、选取文件和输出文件的目录路径。
- 定制编辑程序的命令键。
- 设置Turbo C编辑程序的缺省值和屏幕配置。
- 设置缺省屏幕显示模式。
- 改变屏幕颜色。
- 改变Turbo C编辑窗口和信息窗口的大小。

若要运行Turbo C,只需按第一章中所述方式从配给软盘上将文件复制到工作软盘(或硬盘)上,即可运行。但如果要直接改变TC.EXE中的缺省值则需运行TCINST。

11.2 运行 TCINST

在DOS提示符下,打入TCINST并给出一些参数(如果需要的话)即可运行用户定做程序,并显示如图11.1所示的主菜单。TCINST的完整的语法为:

`tcinst[/c][路径名]`

其中,路径名和/c均是任选的。若提供路径名,则使用给出的路径名,否则,TCINST在当前目录中查找TC.EXE。通常,即使在彩色监视器上,运行TCINST,屏幕也变成黑白的。若欲使TCINST的运行具有彩色,应给出/c选择项。

注意:可使用TCINST一种版本在系统中定做若干不同的Turbo C副本。这些不同的TC.EXE副本可以有不同的可执行程序名,用户所需做的仅是启动TCINST,并给出定做的TC.EXE副本的路径名,例如:

```
tcinst tc.exe  
tcinst - \... \bwtc.exe  
tcinst/c c: \borland \colortc.exe
```

这样便可在系统中定做具有不同编辑程序命令键、不同菜单颜色等的Turbo C副本。

从TCINST主菜单中可以选择Turbo C目录、编辑程序命令、设置环境、显示模式、

颜色、改变窗口大小、退出/存储等功能（参见图11.1）。

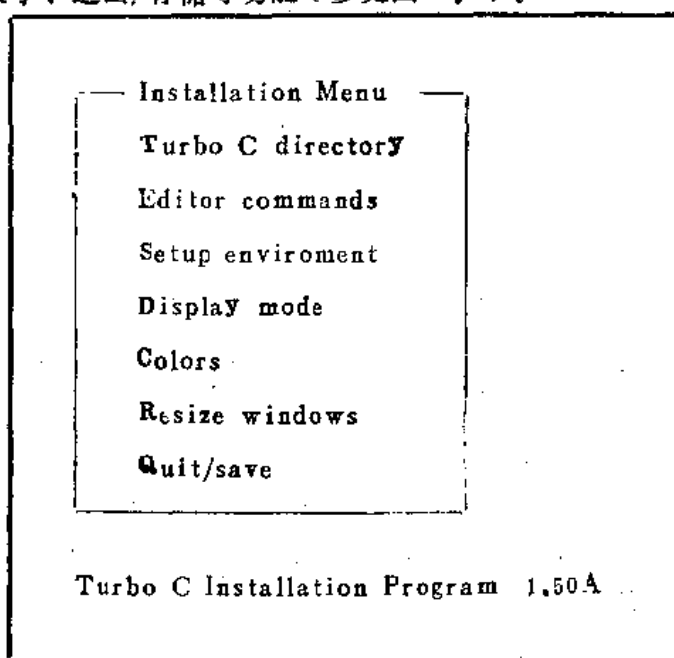


图 11.1 TCINST 主装配菜单

可以用两种途径选择定做程序的功能:

- (1) 用上、下箭头键将选择亮条移到要选的菜单项，然后按Enter键。
- (2) 直接按欲选择的菜单项的第一个发亮的大写英文字母。

例如，要修改 TC 集成环境的颜色，可以按字母 C 键，便进入相应的子菜单。

若要从子菜单返回主菜单，按 Esc 键（如需要可按多次）即可。

以下分别介绍各种选择项的功用。

11.2.1 Turbo C 目录选择项

如果想把路径名（运行 TC 时所用的各种不同目录）直接存到 TC.EXE 中，则需使用 Turbo C 目录选择项。

使用 Turbo C 目录选择项,可以为每个 TC.EXE 的缺省目录说明一个路径。当查找可选择的配置文件、帮助 (Help) 文件、嵌入文件、库文件以及存放输出文件的目录时, Turbo C 就搜索这些目录。

当选择 Turbo C 目录时, TCINST 给出一个子菜单, 子菜单中的项目包括,

- **I**nclude directories (包括目录)
- **L**ibrary directories (库目录)
- **O**utput directory (输出目录)
- **T**urbo C directory (Turbo C目录)
- **P**ick file name (选取文件名)

与 TC.EXE 中相应菜单项类似，用户可为每个项设置一个名字。若不清楚每个项的语法，请参阅本书第二章。

包括目录和库目录 在包括目录和库目录中可以输入多个目录。目录的路径名必须用分号(;)分隔, 每个菜单项最多可输入127个字符, 可以输入绝对或相对路径名。

输出目录和 Turbo C 目录 输出目录和 Turbo C 目录菜单均有一个目录路径名。每

个项最多可有64个字符。

若 TC 在当前目录中找不到帮助 (Help) 文件和 TCCONFIG. TC (缺省 配置文件), 则到 Turbo C 目录中查找。

选取文件名 当选择该菜单项时, 出现一个输入窗口, 在其中用户可输入希望 Turbo C 装入或建立的选取文件的路径名。无缺省的 安装选取文件名。

在为设置 (Setup) 环境菜单项输入一个路径名 (或名) 后, 按下 Enter 键接收之, 再按 Esc 键返回 TCINST 主菜单。退出程序时, TCINST 提示用户是否希望 保存这 些改变。一旦保存了 Turbo C 目录路径, 则该位置 被 写到磁盘上作为 TC.EXE 的缺省设置值的一部分。

11.2.2 编辑命令选择项

Turbo C 的交互式编辑程序提供了许多编辑功能和命令, 包括:

- 光标移动
- 正文插入和删除
- 块和文件操作
- 字符串搜索 (搜索和替换)

这些编辑命令均赋以某些键 (或键的组合), 这在第九章已作了详细说明。

定做程序 TCINST 的菜单之一允许用户将 Turbo C 编辑功能赋以其他的键 (即 所谓的 “重约束键”)。

改变发布 Turbo C 编辑程序命令的键时, 须遵循下列过程:

(1) 从 TCINST 程序的主菜单中选择 编辑 程序命令 (Editor commands) 菜单项, 将出现 Install Editor 屏幕, 显示下述三列正文:

- 第一列 (左边) 描述了可用的编辑功能。
- 第二列列出了主键, 即启动某个编辑程序命令的单键或键组合, 按下这些键可进入特殊的编辑功能。
- 第三列列出次键, 这些是可选择键, 按下这些键可进入同样的编辑程序功能 (次键的优先级要高于主键)。

(2) 在该屏幕 正文的底部几行给出了用于改变在 Primary (主) 和 Secondary (次) 列中的项目的键的简要描述。

键	作用	功 能
Left, Right, Up和 Down光标键	select	选择欲重输入的编辑程序命令
Page Up和 PageDown光标键	page	上滚或下滚一屏
Enter	modify	输入键修改模式
R	restore factor defaults	将所有的编辑程序命令复位到原 缺省输入键定义
Esc	exit	离开装配编辑程序屏幕并返回到 TCINST主菜单
F4	Key Modes	在三种键组合方式之间进行转换

可以按 Enter 键进入编辑键模式，再使用左和右光标键将高亮度光标移向 Primary (主) 或 Secondary (次) 列。

(3) 使用向上 (up) 和向下 (down) 光标键高亮度显示所要重定义键的编辑命令。

(4) 按 Enter 键选择高亮度显示的编辑命令，将出现一个上托窗口，为所选择命令列出当前定义的输入键。在屏幕的底行给出了用于改变这些键的简要描述。

键	作用	功能
Backspace	backspace	删除光标左边的输入键
Enter	accept	对所选择编辑程序命令接受新定义的输入键
Esc	abandon changes	废除对当前选择的修改恢复命令的原输入键并返回到装配编辑屏幕 (准备好选择另一个编辑程序命令)
F2	restore	废除对当前选择的修改，恢复命令的原输入键，但保存当前所选择的命令以便重定义
F3	clear	清除当前选择输入键的定义，但保存当前所选择的命令以便重定义
F4	Key Modes	在三种键组合方式 Wordstarlike, Ignore case 和 Verbatim 之间进行转换

注意，为了把 F2、F3 和 F4 作为编辑程序命令键序列的一部分，需先按下反引号 (') 键，再按下适当的功能键。

(5) 按退格键在上托窗口中从右向左删除各别的键，或按 F3 从窗口中清除所有定义的键。

(6) 键的组合有三种形式：Wordstar-like, Ignore case 和 Verbatim。当前选择的方式在屏幕底行以高亮度给出。不管哪种方式，键组合的第一个字符必须是一个特殊键或控制字符。组合方式决定了如何处理后续字符。

① **Wordstar-like** 在这种方式下，若输入一个字母或字符 [、]、|、^、或 -)，则输入自动作为控制字符组合。例如：

输入 a 或 A 或 Ctrl A	产生	<Ctrl A>
输入 y 或 Y 或 Ctrl Y	产生	<Ctrl Y>
输入 [产生	<Ctrl [>

例如在 Wordstar-like 模式下，若欲定义编辑程序命令作为 <Ctrl A><Ctrl B>，可以输入 TC 编辑程序中下列任一种以激活命令：

<Ctrl A> <Ctrl B>

<Ctrl A> B

<Ctrl A> b

② **Ignore case** 在这种模式下,输入的所有字母键均被转换成相应的大写字母,但输入的字母不自动转换为控制字符组合。若输入的键是控制键和字母组合,则必须同时按下字母和 Ctrl 键。例如,在该模式下,〈Ctrl A〉B与〈Ctrl A〉b相同,但与〈Ctrl A〉〈Ctrl B〉不同。

③ **Verbatim** 若在这种模式下输入字母,则不作任何变化,例如,〈Ctrl A〉〈Ctrl B〉,〈Ctrl A〉B和〈Ctrl A〉b均是不同的。

按 F4 在这些选择中循环,直到所希望的方式被高亮度显示在屏幕的底行。

(7) 为编辑功能键入新定义的键(最多为 6 个键)。如果想删除正赋值的最后一个键,按退格键,如果希望废弃对那个功能的新的键赋值,按 F2 可恢复原来赋值的键或按 Esc 键恢复初值并退出编辑键模式。

(8) 只有当对于一个给定功能的新的(或恢复的)键赋值满意时,按 Enter 键确认。

(9) 当完成键赋值时(已接受了最后一次的修改),按 Esc 键从 Install Editor 屏幕退出并返回到 TCINST 的主菜单。

从以上介绍的改变编辑命令键的过程可知,TCINST 在为用户自己定义 Turbo C 编辑程序命令方面提供了很大的灵活性。但是,用户定义输入键序列时仍要遵循一定的规则。下述的规则有些适用于任何输入键定义,有些则只作用于某些输入键模式。

(1) 对任何给出的编辑程序命令,最多只能定义六个输入键。有些键组合等价于二个输入键:包括 Alt(任何有效键),光标移动键(Up, Page, Down, Del 等),和所有功能键或功能键组合(F4, Shift-F7, Alt-F8 等)。

(2) 第一个输入键必须是非字母数字和非标点符号,即必须是控制键或特殊键。

(3) 为了输入 Esc 作为命令输入键,输入 Ctrl [。

(4) 为了输入 Backspace 作为命令输入键,输入 Ctrl H。

(5) 为了输入 Enter 作为命令输入键,输入 Ctrl M。

(6) Turbo C 的预定义帮助(Help)功能键(F1和Alt F1)不能作为 Turbo C 编辑命令键定义,但其他功能键则可以。若输入一个 Turbo C 热键作为编辑命令键序列的一部分,TCINST 将发出企图覆盖编辑程序热键的警告,并核实用户是否确定要覆盖该键。关于 Turbo C 预定义的热键,可参看第二章。

11.2.3 设置环境选择项

设置环境选择项用于设置与缺省编辑模式和 TC 集成环境显示配置有关的值。用户可以通过该选择项装配几个编辑程序的缺省操作模式。该菜单中有八项是可触发的,第九项给出一个子菜单。

下面说明设置环境菜单的各个选择项及其意义。

Backup source files (后备源文件)

该项为打开状态时(默认为打开),若执行 File/Save 操作,则 Turbo C 自动为源文件建立一个备份,使用原文件名,并加上.BAK 为扩展名。例如,filename.c 的后备文件是 filename.BAK。若该项为关闭状态,则不创建.BAK 文件。

Edit auto save (编辑自动保存)

若该项为打开状态(默认为打开),则在执行 Run 或 File/Os shell 时,Turbo C 自动保存编辑程序中的文件(若上次保存后又修改过)。该选择项有助于防止意外事件发生时源文件

的丢失。若该项为关闭状态，则不进行自动保存。

Config auto save (配置自动保存)

若该项为打开状态（默认为打开），则执行 **Run** 或 **File/Os shcl** 或 **File/Quit** 时，Turbo C 自动保存配置文件。

Zoom state (缩放状态)

若该项为打开状态，则 Turbo C 启动时编辑窗口占全屏幕，当转换到信息窗口时，信息窗口也占全屏幕。若该项为关闭状态，则编辑窗口在信息窗口的上部。（可以使用 TCINST 主装配菜单中的重定义窗口大小选择项来改变窗口的大小）。

Insert mode (插入方式)

若该项为打开状态（默认为打开），则编辑程序在光标位置插入从键盘上输入的信息，并将已有的正文推到光标右边或右边更远处。若插入模式关闭，则可以在光标处覆盖正文。

Autoindent mode (自动缩进模式)

若该模式打开时（默认为打开），按 Enter 键，光标退回到与前行的初始列位置对齐的列。若该模式关闭，光标总是回到第一列。

Use tabs (使用制表符)

若该项为打开状态（默认打开），按 Tab 键时，编辑程序在正文中放入一个制表字符 (^I)，该制表字符的大小由 tab size 说明。若该项关闭，当按 Tab 键时，编辑程序插入足够的空格与前行每个单词的第一个字母的光标对齐。

Screen size (屏幕大小)

一旦选择了该选择项，便出现一个包含三个项目的子菜单。可以为 Turbo C 集成环境的屏幕设置这三种尺寸（25、43或50行）中的一种显示。选用的尺寸依赖于机器硬件：任何机器均可用25行显示，带 EGA 的系统可用43行显示，装有 VGA 的系统可用50行显示。

可以参照快速参考行来选择这些选择项。可以根据需要（及监视器）来改变操作环境缺省值，并作为 Turbo C 的一部分保存。当然，仍可以在 Turbo C 编辑程序内部来改变这些设置值（或通过 **Options/Environment** 菜单项）。

注意：执行 install 时设置的选择项，如果也是 TC.EXE 的菜单项，那么当装入的配置文件中的设置不同于 install 的设置时，则 TC.EXE 的设置覆盖 install 的设置。

11.2.4 显示模式选择项

显示模式选择项可以设置 TC 使用的视屏显示模式，并告知系统视屏适配器是否有“雪花”。

通常，Turbo C 能够正确地检查系统的显示模式。若有以下情况时，需改变显示模式：

- (1) 希望选择一个与当前显示模式不同的模式。
- (2) 彩色图形适配器没有“雪花”。
- (3) 认为 Turbo C 不能正确检测硬件。
- (4) 系统或环境具有一种合成屏幕（与 CGA 类似，只有一种颜色），这时应选择黑白。

键入 D，在主装配菜单中选择显示模式，则出现一个子菜单，从该菜单中选择 Turbo C 将使用的屏幕模式。选择项有：缺省 (Default)，彩色 (Color)，黑白 (Black and white) 和单色 (Monochrome)。

Default (缺省)：选择缺省模式则以启动 Turbo C 时的方式操作。

Color (彩色): 不管启动 TC.EXE 时是什么模式, Turbo C 均采用 80 列彩色模式, 退出时转为原来的模式。

Black and white (黑白): 不管原模式是什么, Turbo C 均采用 80 列黑白模式字符, 退出时转为原模式。具有复合监视器时使用这种模式。

当在前面三种选择项中选择某一种时, 程序对屏幕作视屏检测, 并在快速参考行中给出应做什么的指导。按任意键, 则出现含有一个询问的窗口:

Was there snow on the screen?

这时可选择:

- Yes 若屏幕有“雪花”。
- No 总是关闭雪花检测。
- Maybe 检查硬件。

参照快速参考行可以了解更多一些关于 Maybe 的情况。按 Esc 键返回到主装配菜单。

11.2.5 彩色定制选择项

利用彩色定制选择项, 可为 TC 集成环境的各部分定义所满意的颜色。

从主菜单中按下 C, 允许用户扩充 Turbo C 版本中的颜色。按下 C 后, 就出现包含下列选择项的子菜单。

Customize colors	(定制彩色)
Default color set	(默认颜色集)
Turquoise color set	(青兰颜色集)
Magenta color set	(品红颜色集)

由于屏幕颜色组合近五十种, 因此选择一组预置成的颜色就比较方便。

有三种预置颜色值可供选择, 按 **D**、**T** 或 **M**, 且用 PgUp 和 PgDn 键卷动 Turbo C 屏幕项颜色。若预置颜色值不适用, 则可自己定义。

按 **C** 可定制颜色。Turbo C 中有 12 个项可用于定制颜色, 其中一些是正文项, 一些是屏幕行及窗口, 键入从 A 到 L 的一个字母来选择这些项。

一旦选择了定制颜色的屏幕项后, 就会出现一个上托菜单和一个观察窗口。观察窗口中是所选屏幕项的例子, 而菜单则列出选择的内容。观察窗口同时反应选择调色板时的颜色变化。

例如, 选择 **H** 定制 Turbo C 错误窗口的颜色时, 出现带有错误窗口四个不同部分的上托菜单, 它们是: 标题, 边框, 普通正文和高亮度的正文。

接着从菜单中选择内容, 键入适当的高亮度字母, 就得到选定项的一个调色板。用箭头键从调色板中选择一个所喜欢的颜色, 从观察窗口检查该种颜色下的项, 按 Enter 键记录所选定的项。

欲对每个屏幕项设定颜色, 则重复以上过程。操作结束时, 按 Esc, 返回主装配菜单。

注意: Turbo C 还提供三张内部颜色表, 分别对应彩色、黑白和单色。TCINST 根据当前屏幕模式, 允许用户每次改变一种颜色集。例如, 若欲改到黑白颜色表, 则应在 DOS 下将屏幕模式设置为 BW80, 然后运行 TCINST。

11.2.6 改变窗口大小选择项

如果希望重新设置 Turbo C 的编辑和信息窗口大小, 则可在主装配菜单中按 **R** 来选择重定义窗口大小选择项 (**R**esize windows) 来完成。

使用 Up 和 Down 键可以移动将编辑窗口和信息窗口隔开的光带, 两窗口都不得少于三行。完成了重定义窗口大小后, 按 Enter 键。如需放弃所做出的修改, 按 Esc 返回到主装配菜单。

11.3 从 TCINST 程序退出

一旦完成所希望的各种修改后, 选择主装配菜单中的 **Q**uit/**S**ave 选择项, 屏幕底行将显示:

Save changes to TC.EXE? (Y/N)

- 输入 **Y**, 表示所作的修改将永久地存入 Turbo C 中 (当然, 如果还想对它们进行修改可以重新运行 TCINST)。

- 输入 **N**, 表示所作的修改作废, Turbo C 各缺省值和初始状态均不改变, 并返回到操作系统。

如果恢复原 Turbo C 的缺省值, 仅需把 TC.EXE 从配给磁盘复制到工作磁盘上。也可以通过下述方式恢复编辑命令, 即从主菜单中选 **E**, 然后按 **R** (恢复原缺省值) 以及 Esc 键。

第十二章 Turbo C 语言参考

传统的 C 语言参考手册是 Brian W. Kernighan 和 Dennis M. Ritchie 所著的《C 程序设计语言》(以后简称为《K & R》)。但该书并未为 C 语言定义一个完整的标准。这项工作就留给了美国国家标准局(ANSI)。而《K & R》只提供了一个最小的标准,以便在支持《K & R》定义的 C 语言实现中,可以使用《K & R》中的 C 语言成份编制程序。

Turbo C 不仅支持《K & R》中的定义,而且实现了大多数 ANSI 扩充。Turbo C 通过增加新的成份,改进和扩充传统的 C 语言的功能和灵活性,改善并扩充了 C 语言。限于篇幅,本书不可能提供《K & R》和 ANSI 的 C 定义。不过,本章将注明 Turbo C 对《K & R》定义的扩充,说明哪些来自 ANSI 标准,哪些是 Turbo C 自己增加的成份。

为方便交叉引用,本章将按《K & R》中附录 A “C 语言参考手册”的结构介绍。本章并未引用该附录的所有章节。未在这里介绍的内容可以认为 Turbo C 与《K & R》定义没有重大的区别。为了更清楚地介绍 ANSI 标准中的内容和 Turbo C 扩充的成份,本章有部分内容将用 ANSI C 标准的顺序而不是《K & R》的顺序介绍(在本章各节标题后,将给出对应的《K & R》附录 A 章节号)。

12.1 注 解 (《K & R》2.1)

《K & R》定义的 C 不允许注解的嵌套,例如

```
/*试图将函数 myfunc ( ) 变为注解*/
/*
myfunc ( )
{printf("This is my function\n"); /*The only line*/}
*/
```

该结构将被看作一个注解,该注解在短语 The only line 之后结束,这样,右边的花括号与注解结束符将引起语法错误。Turbo C 中在缺省情况下,也不允许注解嵌套。不过,可以使用 -c 编译选择项(在 O/C/Source 菜单中,设置嵌套注解开关为“开”,使编译程序能正确编译带有嵌套注解的程序(如上面的例子)。一个移植性较好的方法是用 #if 与 #endif 代替注解括号。

注解在宏扩展后用单个空格代替,在其他实现中,注解或被完全删除,或用作词法符号的连接(参见本章中的“词法单位替换”部分)。

12.2 标识符 (《K & R》2.2)

标识符仅用作命名一个变量、函数、数据类型或其他用户定义的对象。在 C 语言中,标识符可以包含字母(A...Z, a...z)、数字(0...9)和下线符(_)。Turbo C 还允许使用美元符(\$)。然而,标识符只能以字母或下划线符开始。

字母的大小写是有意义的,这意味着标识符indx与lndx是不同的。在 Turbo C中,程序中标识符的前32个字符是有效的。但也可以改变,方法是用 -i # 编译选择项。其中 # 是有效字符的个数(这是菜单选择)。

同样,从其他模块移入的全局标识符也是前32个字符有效。可以设定是否区分标识符的大小写,方法是在 Options/Linker 子菜单下设置“连接时区分大小写”(Case-sensitive link...ON)或在 TLINK 命令行置 /c 选择项。但类型为 pascal 的标识符在连接时不区分大小写。

12.3 关键字 (《K&R》2.3)

表12.1 中列出了 Turbo C 保留的关键字,它们不能用作标识符的名字,前面冠以 AN 的是 ANSI 对《K & R》的扩充。前面冠以 TC 的是 Turbo C 的扩充。在《K & R》中提到的关键字 entry 与 fortran 在 Turbo C 中不再作为保留字使用。

表12.1 Turbo C 保留字

TC asm	extern	return	TC -CS	TC -DH
auto	TC far	short	TC -DS	TC -DL
break	float	AN signed	TC -ES	TC -DX
case	for	sizeof	TC -SS	TC -BP
TC cdecl	goto	static	TC -AH	TC -DI
char	TC huge	struct	TC -AL	TC -SI
AN const	if	switch	TC -Ax	TC -SP
continue	int	typedef	TC -BH	
default	TC interrupt	union	TC -BL	
do	long	unsigned	TC -CH	
double	TC near	AN void	TC -CL	
else	TC pascal	AN volatile	TC -CX	
AN enum	register	while		

12.4 常 量 (《K&R》2.4)

Turbo C 支持《K & R》中定义的所有常量类型,并略有增强。

12.4.1 整型常量 (《K & R》2.4.1)

常量的值可以从 0 到 4294967295(十进制)(负常数只是无符号常数加上单目减号)。八进制与十六进制的表示法都是合法的。

任何常量后缀以 L(或 l),均将以长整数形式表示之。类似地,后缀以 U 或 u 表示无正负号数,并且若该常量的值大于 65535,则不论它基数为何,都看成是无正负号长整数。当然也可以在一个常量后既用 L 又用 U。

表12.2 不带 L 或 U 的 Turbo C 整型常量

十 进 制 常 量	
0—32767	int

32768—2147483647
2147483648—4294967295
>4294967295

long
unsigned long
将产生无警告的溢出, 产生的常量
值将是正确值的低位有效部分

八 进 制 常 量

00—077777
0100000—0177777
01000000—01777777777
0100000000000—0377777777777
>0377777777777

int
unsigned int
long
unsigned long
将会溢出(如上所述)

十 六 进 制 常 量

0x0000—0x7FFF
0x8000—0xFFFF
0x10000—0x7FFFFFFF
0x80000000—0xFFFFFFFF
>0xFFFFFFFF

int
unsigned int
long
unsigned long
将会溢出(如上所述)

12.4.2 字符常量(《K & R》2.4.3)

Turbo C 提供双字符常量, 例如 'An', '\n\t', 和 '\007\007'。这些常量在机内表示成16位整型值, 第一个字符在低位字节, 第二个字符在高位字节。注意, 这些常量并不适用于其他的 C 编译程序。

单字符常量, 例如 'A', '\t' 及 '\007' 同样表示成十六位整型值。在这种情形下, 低位字节的正负号被扩充到高位字节, 那就是说, 若值大于127(十进制), 高位字节将被置成-1 [= 0xFF]。若说明默认字符类型是无正负号的(使用 -k 编译选择项 或者在 Options\Compiler\Source 子菜单下选择 Default char type...Unsigned), 则不论低位字节的值为多少, 高位字节均将置成0。

Turbo C 支持 ANSI 允许用十六进制表示字符码的扩充。例如 '\X1F', '\X82' 等等。这里, x 或 X 都可以用, 并可有一到三个数字。

Turbo C 同样支持 ANSI 扩充的转义序列。转义序列(或换码序列)以反斜线(\)开始, 可插入字符常量或字符串常量。表 12.3 列出了所有合法的转义序列, 星号*表示 ANSI 对《K & R》的扩充。

表12.3 Turbo C 转义序列

序 列	值	字 符	含 义
*\a	0x07	BEL	响铃
\b	0x08	BS	退格
\f	0x0C	FF	换页
\n	0x0A	LF	换行
\r	0x0D	CR	回车
\t	0x09	HT	跳格(横向)
\v	0x0B	VT	垂直跳格
\\	0x5C	\	反斜线
\'	0x2C	'	单引号(省略符)

续表

序 列	值	字 符	含 义
\"	0x22	"	双引号
\?	0x3F	?	问号
\DDD		任意	DDD 1到3个八进制数
\xHHH	0xHHH	任意	HHH 1到3个十六进制数

注意：由于 Turbo C 允许双字符常量，若八进制转义序列中数字少于 3 个，而后又接一数字，就会产生歧义。在这种情况下，Turbo C 规定该数字此时不允许出现。例如，因为数字 8，9 在八进制值中是不允许的，常量 \258 将被看成由 \25 与 8 组成的双字符。

12.4.3 浮点常量(《K & R》2.4.4)

按《K & R》规定所有浮点常量都被定义成 double 类型。但可在常量后加后缀 F，从而强制类型为 float 类型的浮点常量。

12.5 字符串(《K & R》2.5)

根据《K & R》，字符串常量仅含一个串单位，它由双引号、正文、双引号组成(如 "like this")。当需要将字符串常量换行时，必须使用反斜线(\)。

Turbo C 允许一个字符串常量包含多个串单位，Turbo C 将对它们进行连接。例如，下述程序：

```
main( )
{
    char *p;
    p= "This is an example of how Turbo C "
        "will automatically\ndo the concatenation for "
        "you on very long strings,\nresulting in nicer "
        "looking programs. ";
    printf(p);
}
```

将输出：

```
This is an example of how Turbo C will automatally
do the concatenation for you on very long strings,
resulting in nicer looking programs.
```

12.6 硬件特性(《K & R》2.6)

《K & R》认为基本数据类型(及其各种组合)的大小及数值范围与实现系统的特性紧密相关，而且通常是由所用的计算机主机的体系结构决定的。对 Turbo C 和所有其他的 C 编译程序而言也是如此。表 12.4 列出了 Turbo C 中各种数据类型的大小及数值范围。Turbo C 中类型 long double 是合法的，但处理与 double 一样。

表12.4 Turbo C数据类型、大小与取值范围

类 型	大 小(位)	范 围
unsigned char	8	0—255
char	8	-128—127
enum	16	-32768—32767
unsigned short	16	0—65535
short	16	-32768—32767
unsigned int	16	0—65535
int	16	-32768—32767
unsigned long	32	0—4294967295
long	32	-2147483648—2147483647
float	32	3.4E-38—3.4E+38
double	64	1.7E-308—1.7E+308
long double	64	1.7E-308—1.7E+308
pointer	16	(near, _cs, _ds, _es, _ss指针)
pointer	32	(far, huge指针)

12.7 类型转换 (《K & R》 6)

Turbo C 提供了从一种数据类型自动转换到另一种数据类型的标准机制。下面几节将介绍对《K & R》的扩充以及和实现有关的部分。

12.7.1 字符、整数与枚举(《K & R》 6.1)

由于单字符和双字符常量都表示为16位的值，将字符常量赋给一个整型对象将是一个16位的赋值(参见《K & R》 2.4.3)。将一个字符对象(例如一变量)赋值给一个整型对象，结果将会自动扩充正负号，除非已设置(使用-k编译选择项)默认char类型无正负号。类型为signed char的对象总要扩充符号，类型为unsigned char的对象转换成int时高位置0。

枚举类型的值可以不加任何修改而转换成整型值。同样，整型值也可以直接转换成枚举类型值。枚举值与字符的转换和整型值与字符的转换相同。

12.7.2 指针(《K & R》 6.4)

Turbo C 中，程序中的不同指针变量大小可以不同，这依赖于存储模式或所使用的指针类型修饰符。例如，在一个特殊的存储模式中编译程序时，源代码中的地址修饰符(near, far, huge, _cs, _ds, _es, _ss)可以完全不顾存储模式规定的指针变量大小。

指针变量必须说明为指向某个特定类型，哪怕那个类型为void(这意味着可以指向任何类型)。一旦说明了以后，指针变量就可以指向任何类型的对象。Turbo C 允许这样重指定指针，当重指定指针发生时编译程序会发出警告，除非这个指针原来被说明为指向void类型。但是，指向数据的指针不能转化成指向函数的指针，反之亦然。

12.7.3 算术转换(《K & R》 6.6)

《K & R》引用了常见的算术转换，指明了算术表达式(运算分量 运算符 运算分

量)中值的转换。下面是Turbo C转换算术表达式中运算分量的步骤:

(1) 任何非整数和非双精度浮点型量的转换如表 12.5 所示。转换后,运算符连接的两个运算分量或是整型 int (带 long 和 unsigned 修饰符)或是双精度浮点型 double。

(2) 若有一运算分量是双精度浮点型 double, 另一运算分量也被转换成 double 类型。

(3) 若有一运算分量为 unsigned long 类型, 另一个运算分量也被转换成 unsigned long 类型。

(4) 若有一运算分量类型为 long 类型, 另一个运算分量也转换成 long 类型。

(5) 若有一运算分量类型为 unsigned 类型, 另一个运算分量也转换成 unsigned 类型。

(6) 两个运算分量均为 int 类型。

表达式的结果类型与变换后的两运算分量的类型相同。

表12.5 常见算术运算类型转换

类 型	转 换 成	方 法
char	int	正负号扩充
unsigned char	int	高位填 0 (全部情况)
signed char	int	正负号扩充(全部情况)
short	int	若无正负号就转换为无正负号整型
enum	int	不变
float	double	尾数填 0

12.8 运算符 (《K & R》 7.2)

Turbo C 提供单目运算符 +, 是对《K & R》的一个扩充。通常, Turbo C 将重新组织表达式重新安排可交换运算符 (例如 * 与双目 +) 使编译产生一个高效的表达式。但 Turbo C 不重组单目 + 附近的表达式。这意味着可以使用单目运算符 + 控制易于产生精度错误和溢出的浮点表达式。而不必将其分成几个独立的表达式并使用临时变量。

例如, 若 a、b、c、f 均为 float 类型的变量, 在计算表达式 $f = a + (b + c)$ 时, 先计算表达式 (b+c) 然后再与 a 相加。

12.9 类型与类型修饰符 (《K & R》 8.2)

Turbo C 提供了下列《K & R》中没有的基本类型:

(1) unsigned char (4) long double

(2) unsigned short (5) enum

(3) unsigned long (6) void

前三种基本类型含义自明, 第四个同类型 double 一样。类型 int 与 short 在 Turbo C 中一样, 都是 16 位。有关各种类型实现的细节, 可参阅“硬件特性”那一节。

12.9.1 枚举类型(enum)

Turbo C 实现的枚举类型与 ANSI 标准一样。枚举类型常用来描述整数值的离散集合。例如, 可以定义,

```
enum day {sun,mon,tues,wed,thur,fri,sat};
```

列在 days 中的名字都是整数值, 第一个(sun)自动转成 0, 后面的比前面依次大 1, (mon=1, tues=2, 等等)。但也可以对一名字赋予一特定的值。没有特定值的名的值是它前一名的值加 1。例如:

```
enum coins {penny=1, nickle=5, dime=10, quarter=25};
```

枚举类型的变量可以赋以任何整型值, 这时不进行类型检查。

12.9.2 void 类型

按《K & R》的定义, 每一个函数均有一个返回值, 如若没有定义类型, 函数将被认为是整型。与 ANSI 标准定义一样, Turbo C 提供了 void 类型, 用于显式表示函数并不返回值。而且, 保留字 void 还可表示空参数表, 例如

```
void putmsg(void)
{
    printf("Hello, world\n");
}

main()
{
    putmsg();
}
```

类似地, 还有一个特殊的结构, 可以定义一个表达式为空类型, 以显式指出将忽略函数返回的值。例如, 若想暂停运行直到用户在键盘上按下一个键, 输入的内容可以忽略, 可以这样编写:

```
(void) getch();
```

最后, 可以说明指向 void 的指针。这并不是建立一个什么也不指的指针, 而是建立一个可以指向各种数据对象的指针。数据对象的类型无须知道。可以把任何指针赋给 void 指针(反之亦然), 而不需要用类型转换。不过, 对 void 指针不可以使用运算符(*), 因为它没有任何对象类型。

12.9.3 带正负号修饰符 signed

作为对《K & R》中定义的三种类型修饰符 long, short 及 unsigned 的扩充, Turbo C 还提供了在 ANSI 标准中定义的另外三种类型修饰符: signed, const 和 volatile。

signed 与 unsigned 相反, 它明确表示值以通常正负号的形式存储(二进制补码)。这样做主要是为了文档编制和完整性。不过, 若用默认的无正负号 char 类型进行编译时, 必须使用 signed 修饰符以便定义一个 signed char 类型的变量或函数。unsigned 单独使用含义同 unsigned int 一样, 同样 signed 单独使用含义为 signed int。

12.9.4 常量修饰符 const

const 修饰符, 按 ANSI 标准的定义, 禁止对对象的赋值或任何其他的副作用如增加或减少。const 指针不能被修改, 但它指向的对象可以修改。const 单独使用同 const int 等价。考虑下例,

```
const float pi    = 3.1415926;
const maxint     = 32767;
const *char str   = "Hello, world";
```

作了上述定义, 下列语句就是非法的:

```
pi = 3.0           /*对一常量赋值*/
i  = maxint - -;   /*常量减一*/
str = "Hi, there! "; /*将str指向他处*/
```

注意: 函数调用 strcpy (str, "Hi, there! ") 是合法的, 它将串 "Hi, there!" 逐个字符拷贝到 str 指向的存储位置。

12.9.5 易变修饰符 volatile

volatile 修饰符在 ANSI 标准中也有定义, 它与 const 几乎相反, 它指出对象可以在程序中, 也可以在程序以外 (例如中断程序或 I/O 端口) 来修改。定义 volatile 的对象将提醒编译程序在计算包含这个对象的表达式时不要作任何假设, 因为这个值 (理论上) 在任何时候均可改变。这同样也阻止编译程序将变量置成寄存器变量。

```
volatile int ticks;
interrupt timer()
{
    ticks++;
}
wait(int interval)
{
    ticks = 0;
    while(ticks < interval);
}
```

这些子程序 (假设 timer 与硬件时钟中断正确地相联系) 将实现由参数 interval 指定的时间等待。一个高度优化的编译程序也不可在 while 循环内部 直接用外部 ticks 的值。这是因为循环体并不改变 ticks 的值。

12.9.6 修饰符 cdecl 和 pascal

Turbo C 允许在程序中方便地调用其他语言书写的程序, 反之亦然。当这样混合编程时, 必须处理好两个重要问题: 标识符与参数传递。

编译一个 Turbo C 程序时, 程序中的所有全局标识符, 即函数名、全局变量, 都存储在编译产生的目标代码程序中, 以用于连接。在缺省情况下, 标识符都是以原来的大小写存储的 (小写、大写或大小写混用)。此外, 下划线符 (_) 加至标识符前, 除非选择 -u- 选项 (Generate underbars...OFF)。

同样, 在程序中说明的任何外部标识符都有同样的格式。连接 (缺省时) 对大小写是敏感的。所以, 用在各个源程序中的同一标识符, 其拼写和大小写均应相同。

(1) pascal

在有些情形 (例如, 引用其他语言的代码), 保存标识符的这一缺省方法可能会引起问题。

Turbo C 提供一个解决这个问题的方法。可以定义标识符为 pascal 类型,这意味着这个标识符将会转化成大写的,且开头没有下划线。(若标识符是一个函数,这也同样影响到参数传递序列,更详细的内容参考“函数类型修饰符”)。源代码中用大写还是用小写就无关紧要了。连接时全被认为是大写的。

(2) cdecl

也可以使用编译选择项 -p (Calling convention...pascal), 使源程序中的全部标识符均变成 pascal 类型。然而,这时需要保证其中某些标识符仍然保持它们原先的大小写和开头的下划线,尤其是其他文件中定义的 C 标识符。

为做到这一点,可以定义这些标识符为 cdecl 类型(同样对函数的参数传递有影响)。

读者将会注意到,例如,嵌入文件(STDIO.H等)中的所有函数均为 cdecl 类型。这保证了即使使用 -p 选择项编译,也可以和库子程序进行连接。

进一步的细节已在第八章中讨论过,读者还可以参阅本章的12.12.1。

12.9.7 修饰符 near, far 和 huge

Turbo C 有三个影响间接操作符(*)的修饰符 near, far 和 huge, 即它们影响数据指针。这些关键词的含义已在第八章作了详细的介绍,这里再作一个简短的概述。

Turbo C 允许使用多个存储模式中的一个进行编译。所使用的模式决定了数据指针的格式(及其他东西)。若使用一个小数据模式(极小、小、中),所有的数据指针为16位,且为数据段寄存器(DI)的位移。如果使用一个大数据模式(紧凑、大、特大),所有数据指针均为32位,且给出段地址与位移。

有时,使用某种规格的数据模式时,又想定义不同于默认大小的指针,就可以用修饰符 near, far 和 huge。

near 指针为16位,它使用数据段寄存器的内容作为段地址。小数据模式下指针默认为 near。使用 near 指针将会限制所用的数据在当前64K数据段。

far 指针32位,它包括段地址与位移。对大模式而言,指针默认为 far。使用 far 指针可以在 Intel 8088/8086 处理器中的1MB存储空间中的任何地方引用数据。

huge 指针也是32位长,同样也含一个段地址和一个位移。不过,和 far 指针不一样, huge 指针总是规格化的。这些细节已在第八章中讨论了,这里只给出几点结论:

(1) 关系运算将(==, !=, <, >, <=, >=)都能正确用于 huge 指针,而不能正确用于 far 指针。

(2) huge 指针是规格化的,所以,施于 huge 指针的算术运算符对段地址与位移均起作用;而对 far 指针,只对位移起作用。

(3) huge 指针可以通过不断增加1而寻址1MB地址空间,而 far 指针将在64K段的头折叠。

(4) 使用 huge 指针需要额外的时间,因为算术运算后必须调用规格化子程序。

12.10 结构与联合(《K & R》8.5)

Turbo C 除实现了《K & R》定义的结构与联合,还增加了下列特征:

12.10.1 字边界

如果使用编译选择项 `-a(Alignment...word)`, Turbo C 将用其所需的字节来调节结构(或联合)以对齐字边界。从而实现:

- (1) 结构将开始于字边界(偶数地址);
- (2) 任何非字符域相对于结构头都有一个偶数位移;
- (3) 结构末尾可能加上一个字节以保证整个结构包含偶数个字节。

12.10.2 位域

在 Turbo C 中, 一个位域可以占 1—16 位, 可为 signed int 又可为 unsigned int。位域在字内从低位到高位分配。例如:

```
struct mystruct {  
    int      i: 2;  
    unsigned j: 5;  
    int      k: 4;  
    int      m: 1;  
    unsigned n: 4;  
};
```

产生下列记录格式

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	(未用)				j				i		

整数域中的数以 2 进制补码形式存储, 最左一位为正负号。例如, 一个 1 位宽 signed int 位域(例如 k)只能表示值 -1 与 0, 因任何非 0 的数都将被解释成 -1。

12.11 语 句(《K & R》9)

Turbo C 原原本本地实现了《K & R》中描述所有的语句, 没有例外和修改。

12.12 外部函数定义(《K & R》10.1)

在 Turbo C 中, 函数内的 extern 说明遵守分程序作用域规则。在定义它们的分程序作用域之外这个说明无效。不过, Turbo C 将记下这些说明以便与以后相同对象的说明进行比较。

Turbo C 实现大部分 ANSI 标准对《K & R》函数定义的扩充。这包括附加的函数修饰符和函数原型。Turbo C 本身也有一些扩充, 例如类型为 interrupt 的函数。

12.12.1 函数类型修饰符(《K & R》10.1.1)

除了 `extern` 与 `static` 之外, Turbo C 还有一些专用于函数定义的类型修饰符: `pascal`, `cdecl`, `interrupt`, `near`, `far` 和 `huge`。

(1) 函数修饰符 `pascal`

`pascal` 修饰符为 Turbo C 所特有, 用于指定使用 `pascal` 参数传递序列的函数(和指向函数的指针)。这允许从用其他语言写的程序中调用 C 语言函数。同样, 也允许 C 程序调用其他语言写的外部子程序。连接时函数名全部转换成大写。

使用编译选择项 `-p(Calling convention...pascal)` 将使所有函数(和指向这些函数的指针)当作 `pascal` 类型处理。而且, 只要 C 子程序能知道函数的类型为 `pascal`, 那么类型为 `pascal` 的函数将可以被 C 子程序调用。例如, 若已定义和编译了下列函数:

```
pascal putnums(int i, int j, int k)
{
    printf("And the answers are: %d, %d, and %d\n", i, j, k);
}
```

只要给出以下说明, 另一个 C 程序就可以连接并调用它。

```
pascal putnums(int i, int j, int k);
main( )
{
    putnums(1,4,9);
}
```

与 `printf` 函数不同, 类型为 `pascal` 的函数不能有可变个数参数。由此, 不可以在一个 `pascal` 函数定义中使用省略号(`...`)(参考以下对“函数原型”的说明)。

(2) 函数修饰符 `cdecl`

`cdecl` 也是 Turbo C 特有的。和 `pascal` 修饰符一样, 它用于函数和指向函数的指针。它的功能是摆脱编译选择项 `-p` 的约束, 把要调用的函数作为一个正常的 C 函数。例如, 如果使用了 `-p` 编译选择项编译上述的程序, 但又想使用 `printf`, 便可这样写,

```
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);
}
```

如果使用编译选择项 `-p` 来编译一个程序, 使用的运行时刻库中的所有函数均有 `cdecl` 说明。读者只要看一下嵌入文件(例如 `STDIO.H`), 将会发现每一个函数均显式定义成 `cdecl` 类型。注意: `main` 也必须说明成 `cdecl`, 因为 C 启动代码总按照 C 参数传递方法调用 `main`。

3) 函数修饰符 interrupt

interrupt 是 Turbo C 的另一特有修饰符。interrupt 函数需和 8086/8088 中断向量联合使用, Turbo C 编译 interrupt 函数时将产生附加的函数入口和出口代码以保护寄存器 AX, BX, CX, DX, SI, DI, ES 与 DS。其他的寄存器 SP、BP、SS、CS 与 IP 作为 C 调用序列的一部分或作为中断处理一部分来保护。下面是一个典型的 interrupt 函数定义。

```
void interrupt myhandler( )
{
...
}
```

必须将中断函数的类型说明为 void。中断函数可在任何存储模式中定义。对 huge 以外的所有存储模式, DS 设置成程序数据段。对 huge 模式, DS 设置成模块的数据段。

12.12.2 函数原型(《K & R》10.1.2)

《K & R》只允许函数说明包含函数名、函数的类型和一对空括号。参数(若有的话)仅在真正定义函数体时才可以定义。

ANSI 标准以及 Turbo C 都允许使用函数原型定义一个函数。这时的函数说明包含有关函数参数的信息。编译程序可利用这一信息检查函数调用的合法性。编译程序同时利用这个信息强制实参为特定的类型。假设有下列程序段:

```
long lmax(long v1, long v2);
main( )
{
    int limit = 32;
    char ch = 'A'
    long mval;
    mval = lmax(limit, ch);
}
```

它为 lmax 给定了函数原型, 这个程序在调 lmax 前使用标准赋值规则将 limit 和 ch 转化为 long 类型, 然后将 limit 与 ch 放入栈中。没有函数原型, limit 和 ch 将分别作为整数和字符放在栈中, 在这种情况下, 调用的实参表和 lmax 的形参表在大小、内容上均不符合, 就会出现错误。在《K & R》中, 没有设施检查参数的类型及数目。函数原型的使用可以解决这一问题且查出其他的编程错误。

函数原型也使程序易读。例如, 函数 strcpy 有两个参数: 一个源串, 一个目的串, 问题是如何区分? 如下的函数原型:

```
char *strcpy(char *dest, char *source);
```

就很清楚地表示出来了。若嵌入文件包含有函数原型, 可以通过打印该文件以得到书写调用那些函数的程序所需的大部分信息。

带有括有单词 void 的函数说明表明这个函数没有任何参量。

```
f(void)
```

否则, 括号内应有用逗号分开的说明元表, 说明元可含类型转换, 在如下说明中,

```
func (int*, long);
```


也可以包含有标识符, 如

```
func(int *count, long total);
```

在上面提及的两个说明元表中, 函数 func 均带有两个参数。一个为名 count 对象类型为 int 的指针, 另一个名为 total 类型为 long integer。包含有标识符时, 该标识符只在参数类型不匹配错误发生时用于诊断信息。

函数原型通常定义一个函数有固定个数的参数。因为 C 函数可带有可变个数的参数(例如 printf), 函数原型的参数表可以省略号(...)结尾, 例如:

```
f(int *count, long total, ...)
```

在这种情形下, 在编译时刻检查固定的参数, 其他参数的处理和无函数原型一样。下面是一些函数原型的例子:

```
int f(); /*该函数回送整型, 未说明参数信息, 是传统的《K & R》风格*/
int f(void); /*该函数回送整型, 无参数*/
int p(int, long); /*该函数回送整型, 接受两个参数,
                  一为 int 类型, 另一为 long 类型*/
int pascal q(void); /*回送整型的 pascal 无参数函数*/
char far*s(char*source, int kind); /*该函数回送字符远指针, 接受两个参
                                     数, 其一为字符指针, 其二为整型*/
int printf(char*format, ...); /*该函数回送整型, 接收固定参数*/
                               /*为一字符指针, 可变参数个数和类型任意*/
int(*fp)(int); /*函数指针, 该函数回送整型, 接收一整型参数*/
```

下面简单总结一下 Turbo C 在有和没有函数原型情况下对函数调用时, 是怎样处理语言修饰符和形式参数的。

规则 1: 在对函数的调用中函数定义的语言修饰符应与函数说明中所用的修饰符相匹配。

规则 2: 一个函数可以修改其形参的值, 但对调用子程序的实参不起作用, 中断函数例外。更详细的内容在第八章的“中断函数”中已作了叙述。

当没有预先说明函数原型时, Turbo C 将根据 12.7.3 节中描述的整数类扩充规则, 在函数调用中将其转化成整数。当作用域内有函数原型时, Turbo C 将给定参数的类型转换为说明的参数的类型。

当函数原型包含省略符时, Turbo C 先按其他原型的方法转换已说明的函数参数(直到省略号), 编译程序将根据没有原型的函数参数的一般处理规则, 处理固定参数后的任何参数。

若存在原型, 参数的个数必须匹配(除非原型中使用了省略号)。类型兼容应达到赋值语句可以合法转换的程度。可以显式地将参数转换成函数原型可接受的类型。

下面的例子可以说明这几点:

```
int strcmp(char *s1, char *s2); /*全原型*/
char *strcpy(); /*无原型*/
int sampl(float, int, ...); /*全原型*/
sampl2();
```

```

{
char *sx,*cp;
double z;
long a;
float q;
if (strcmp(sx, cp)) /* 1.正确*/
    strcpy(sx, cp, 44); /* 2.Turbo C中可以,但不可移植*/
    sample(3, a, q); /* 3.正确*/
    strcpy(cp); /* 4.运行时出错*/
    sample(2); /* 5.编译时出错*/
}

```

在该例中,五个调用(注释中编号)说明了函数调用与函数原型使用的各种情况:

调用1中,strcmp的使用与原型完全匹配,一切正常。

调用2中,strcpy的调用中多了一个参数(strcpy定义时只有两个参数,不是三个)。这种情况下,Turbo C将费一点时间和代码将额外参数压栈。但这里还没有语法错误,因为编译程序还未被告知strcpy的参数信息,这个调用是不可移植的。

调用3中,原型指定了sample的第一个参数转换成float,第二个转换成int。编译程序将发出警告可能丢失有效位,因为从long到int的转换将去掉高位(可以显式将其转换成int消除这个警告)。第三个变元q,对应于原型中的省略号。根据通常的算术转换规则将它转换成double型,整个调用正确。

调用4中,strcpy又一次被调用,不过只有一个参数。这会引起运行错误且会中止程序。编译程序不给出任何信息(即使参数的个数与前一次对同一函数的调用不同)。因为strcpy没有函数原型。

调用5中,sample的调用参数个数不足。因为sample至少要求两个参数,所以这个语句有错。编译程序将给出调用参数个数不足的信息。

需要说明的是,如果函数原型与实际函数定义不匹配,Turbo C将会发现这种情况(当且仅当定义与原型在同一文件中)。若建立了一个带有相应函数原型的嵌入文件的子程序库编译该库时应包括该嵌入文件。这样,原型与实际定义间的差异才可以查出来。

12.13 作用域规则(《K & R》11)

Turbo C在允许标识符不唯一方面,比《K & R》的规定更为宽松。Turbo C实现中有四类不同的标识符。变量名、类型名、枚举值在定义它们的分程序中必须是唯一的。外部标识符在外部说明变量中必须唯一。

结构、联合和枚举标识符在定义它们的分程序内部只能定义一次,在所有函数外定义的标识符在外部定义的所有标识符之内必须唯一。

结构和联合成员名在定义它们的结构和联合内必须唯一。不同结构中定义的不同成员名的类型和位置均没有限制。

转跳标号在定义它们的函数中必须是唯一的。

12.14 编译程序控制行(《K&R》12)

Turbo C 支持《K&R》中的所有控制命令。预处理指令以#开头, #的前后可以有空白。

12.14.1 词法单位替换(《K&R》12.1)

Turbo C 实现了《K&R》中定义的# define和# undef, 并有下列扩充:

(1) 下列标识符不可出现于# define与# undef指令中:

—STDC—

—FILE—

—LINE—

—DATE—

—TIME—

(2) 在宏定义中, 两个词法单位可以拼到一起, 方法是将它们用##分开(两边可有空白)。预处理程序将消除空白及##, 将两个词法单位连起来。这可用作构造标识符。例如:

```
#define VAR(i, j)    (i    ## j)
```

于是VAR(x, 6)将扩展成x6, 这种方法替代了一度使用过的(但不可移植)方法, 即使用(i/**/j)。

(3) 宏定义串中涉及的嵌套宏命令仅当宏命令本身扩展时扩展, 而不是在宏命令定义时扩展。这对# undef与嵌套宏的相互影响极大。

(4) #符号可用在宏参数前以使该参数字符串化(将它转化成字符串)。宏指令扩展以后, #<形参>被“<实参>”代替。例如给出如下宏定义:

```
#define TRACE(flag) printf(#flag " = %d\n", flag)
```

则如下宏调用:

```
highval = 1024;
```

```
TRACE(highval);
```

将扩展为:

```
highval = 1024;
```

```
printf("highval" " = %d\n", highval);
```

(5) 和其他实现不同, Turbo C 不在串及字符常量内部扩展宏参数。

12.14.2 文件嵌入(《K&R》12.2)

Turbo C 实现了《K&R》的# include 指令。但有如下附加的功能: 若预处理程序在缺省目录中不能找到嵌入文件(假设使用形式# include "filename"), 它将在由编译选择项-I指定的目录中查找。若使用# include <filename>形式, 就只搜索由-I指定的那些目录。(在菜单选择项 O/Environment/Include directives 下列的目录表与由-lpathname 命令行选择项给出的目录作用相同)。可以用宏扩展建立# include 路径名, 包括其界线符。若关键字的下一行以标识符开头, 预处理程序将在正文中查找宏。若串括在引号或尖括号中, Turbo C 将不检查其中嵌入的宏。例如,

```
#define myinclude "c:\tc\include mystuff.h"
```

```
#include <myinclude>
#include "myinclude.h"
```

第一个 `#include` 语句将使预处理程序查找 `C:\TC\INCLUDE\MYSTUFF.H`；而第二个将在缺省目录中查找 `MYINCLUDE.H`。

`#include` 语句中的宏不能使用字符串常量并置和词法单位拼合。宏扩展必须产生像通常 `#include` 编译指令所读的正文。

12.14.3 条件编译(《K & R》12.3)

Turbo C 支持《K & R》中定义的条件编译，条件编译可用全为空白的行替代某些行。被忽略的行是以 `#if`，`#ifdef`，`#ifndef`，`#else`，`#elif` 和 `#endif` 编译指令开头的行以及由于该编译指令而不被编译的行。所有的条件编译指令在它们所处的源文件或包括文件中都必须是完整的。

Turbo C 也支持 ANSI 运算符 `defined(symbol)`。若 `symbol` 以前已被定义（使用 `#define`），且未使用 `#undef` 定义过，其值将为 1 (true)；否则，其值作为 0 (false)。于是 `#if defined(mysym)` 和 `#ifdef mysym` 是相同的。这样做的好处是可以在一个 `#if` 指令后的复杂表达式中反复使用 `defined`。例如：

```
#if defined(mysym) || defined(yoursym)
```

最后，Turbo C 和 ANSI 不同的一点是，Turbo C 允许在预处理表达式中使用 `sizeof` 运算符。例如：

```
#if(sizeof(void*) == 2)
#define SDATA
#else
#define LDATA
#endif
```

12.14.4 行控制(《K & R》12.4)

Turbo C 支持《K & R》中定义的 `#line`，`#line` 中的宏定义和 `#include` 中的宏定义一样扩展。

12.14.5 出错指令(ANSI C3.8.5)

Turbo C 支持 ANSI 标准中提出的 `#error` 指令，但 ANSI 标准并未明确定义。格式为：

```
error errmsg
```

如果源程序的条件编译中包含此指令，当条件测试失败时，预处理程序将读 `#error` 指令，然后立即停下，报告下列信息：

```
Fatal: filename line# Error directive: errmsg.
```

预处理程序将不查找嵌套宏调用，只扫描正文，消去其中注释，显示余下的正文。

12.14.6 pragma 编译指令 (ANSI C 3.8.6)

Turbo C 实现了在 ANSI 标准中仅作含混定义的 `#pragma` 编译指令（和 `#error` 类

似)。其目的是允许和实现有关的指令，其形式为：

```
#pragma <directive name>
```

用 #pragma, Turbo C 可定义它需要的任何指令而不影响其他支持 #pragma 的编译程序。为什么呢？因为根据定义，若编译程序不认识编译指令名字，它将忽略 #pragma 指令。

(1) pragma inline

Turbo C 识别两种 #pragma 指令，第一个为

```
#pragma inline
```

这个指令等同于 -B 编译选择项，它告诉编译程序现行程序中有汇编语言代码（参考第八章），这最好放在程序的顶部，因为编译程序遇到 #pragma inline 时，它将以 -B 选择项重新启动。实际上，也可以不用 -B 选择项与 #pragma inline 编译指令，编译程序一遇到 asm 语句就会重新开始。使用选择项与指令的目的是节省编译时间。

(2) #pragma warn

另一个 #pragma 编译指令为

```
#pragma warn
```

这个指令允许忽略特定的 -wxxx 命令行选择项（或特定的 Display warnings...On 选择项）。例如，如果源代码包含以下编译指令：

```
#pragma warn +xxx
```

```
#pragma warn -yyy
```

```
#pragma warn .zzz
```

xxx 警告将开放（即使在 O/C/Errors 子菜单中它被置成关闭），yyy 警告将变成关，zzz 警告将保留其在程序编译前的值。

三个字母缩写和与其有关的警告的完整内容在第十章中已作了描述。

12.14.7 空编译指令(ANSI C 3.7)

为了完整的缘故，ANSI 标准与 Turbo C 均能识别空指令（为仅有一字符 # 的行），这个指令总被忽略掉。

12.14.8 预定义宏名(ANSI C 3.8.8)

ANSI 标准要求实现 5 个预定义宏，Turbo C 全部实现了。注意每一个名的开始和结尾均有两个下划线（__）

__LINE__ 当前正处理的程序行号——十进制常量，源程序的第一行定义为 1。

__FILE__ 当前处理的源程序的名——字符串常量，这个宏在编译程序处理 #include 指令、#line 指令或嵌入文件处理完时改变。

__DATE__ 预处理程序开始处理当前源程序时的日期——一个字符串常量，不论程序执行时间有多长，程序中的每一个 __DATE__ 值均是相同的。

日期的格式为：mmm dd yyyy，其中：

mmm 代表月份（如 Jan, Feb, 等）

dd 代表日期（1..31，这个值小于 10 时 dd 的第一个字符为空。）

yyyy 代表年份（如 1987, 1988 等）

- `__TIME__` 预处理程序开始处理当前源程序的时间——一个字符串常量，不论程序执行时间长短，程序的所有`__TIME__`值均是相同的，其格式为 hh:mm:ss。
 hh 代表小时(00..23)
 mm 代表分(00..59)
 ss 代表秒(00..59)
- `__STDC__` 使用ANSI兼容标志(-A)(ANSI keywords only...ON)，其值为常量1；否则，无定义。

12.14.9 Turbo C 预定义宏

Turbo C 预处理程序定义了几个其他的宏功能。同 ANSI 预定义的宏一样，每一个宏的开头和结尾也均有两个下线符。

- `__TURBOC__` 给出当前 Turbo C 的版本号——十六进制常量。版本1.0是0x0100，版本1.5是0x0105，依次类推。
- `__PASCAL__` 测试-p标记，若-p标记已用，其值为常量1，否则没有定义。
- `__MSDOS__` 对所有的编译均为整型常量1。
- `__CDECL__` -p标记未使用信号(Calling convention...C)。若-p未用，置成整型常量1，否则没有意义。

下面六个符号依据编译时刻选择的存储模式而定义。对某一个编译过程而言，只有一个定义，其他均未定义。例如，若在小模式下编译，`__SMALL__`被定义且其它均未定义，于是 `#if defined(__SMALL__)` 将为true，`#if defined(__HUGE__)` (或其他的) 将为false，已定义过的宏值为1。

- `__TINY__` 极小存储模式的选择项
- `__SMALL__` 小存储模式的选择项
- `__MEDIUM__` 中存储模式的选择项
- `__COMPACT__` 紧凑存储模式的选择项
- `__LARGE__` 大存储模式的选择项
- `__HUGE__` 特大存储模式的选择项

12.15 过时成份(《K&R》17)

Turbo C 中没有《K & R》中提及的过时成份。

通读本章，读者会感到很乏味。所以，编者将它从原文中的第八章移到本章来，以供需要时查阅。

关于 Turbo C 完整语法的 BNF 描述将在附录A中给出。

附录A Turbo C语法的BNF描述

本附录将用变型的BNF (Backus—Naur Form)描述 Turbo C 的语法结构。所谓变型的BNF形式,即能够方便地使用BNF形式表达时,尽量采用之,若表达起来较为啰嗦,或用其他方法更直观时,则用变通的方法。本附录将按下述层次来描述:

- (1) 词法:词法单位、关键字、标识符、常量、字符串常量、运算符、标点字符。
- (2) 语法:表达式、说明、语句、外部定义。
- (3) 预处理指令。

有关的元语言符号的含义说明如下:

< > 一对尖括号括住的成份是元语言变量(即非终结符号),当出现在规则右部时,表示该变量的定义尚需进一步展开。

::= 读作“定义为”,该符号右部的成份是对左部变量的定义。

| 读作“或者”,用于分隔并列的成份。

opt 附在某个元语言变量后的opt,表示其紧前的成份是任选的。

A.1 词 法

A.1.1 词法单位

<词法单位> ::= <关键字> | <标识符> | <常量> | <字符串常量> | <运算符> |
<标点符号>

A.1.2 关键字

<关键字> ::= 下列符号之一,

asm	continue	float	long	sizeof	void
auto	default	for	near	static	volatile
break	do	goto	pascal	struct	while
case	double	huge	register	switch	_cs
cdel	enum	if	return	typedef	_ds
char	extern	int	short	union	_es
const	far	interrupt	signed	unsignd	_ss

A.1.3 标识符

<标识符> ::= <非数字字符> | <标识符><非数字字符> | <标识符><数字>

<非数字字符> ::= <小写字母> | <大写字母> | <下划线符>

<小写字母> ::= a—z

<大写字母> ::= A—Z

<下划线符> ::= _

<数字> ::= 0—9

A.1.4 常 量

<常量> ::= <浮点常量> | <整型常量> | <枚举常量> | <字符常量>

<浮点常量> ::= <小数常数><指数部分>_{opt}<浮点后缀>_{opt} |

<数字序列><指数部分><浮点后缀>_{opt}

<小数常数> ::= <数字序列>_{opt}<数字序列> | <数字序列>.

<指数部分> ::= e<正负号>_{opt}<数字序列> | E<正负号>_{opt}<数字序列>

<正负号>::= + | -
 <数字序列>::= <数字> | <数字序列><数字>
 <浮点后缀>::= f | F | L
 <整型常量>::= <十进制常量><整型后缀>_{opt} |
 <八进制常量><整型后缀>_{opt} |
 <十六进制常量><整型后缀>_{opt}
 <十进制常量>::= <非零数字> | <十进制常量><数字>
 <八进制常量>::= 0 | <八进制常量><八进制数字>
 <十六进制常量>::= 0X<十六进制数字> |
 0x<十六进制数字> |
 <十六进制常量><十六进制数字>
 <非零数字>::= 1—9
 <八进制数字>::= 0—7
 <十六进制数字>::= 0—9 | a—f | A—F
 <整型后缀>::= <无正负号整数后缀><长整数后缀>_{opt} |
 <长整数后缀><无正负号整数后缀>_{opt}
 <无正负号整数后缀>::= u | U
 <长整数后缀>::= l | L
 <枚举常量>::= <标识符>
 <字符常量>::= <C字符序列>
 <C字符序列>::= <C字符> | <C字符序列><C字符>
 <C字符>::= 源字符集中除单引号('), 反斜线(\)以及换行符之外的任何字符 |
 <转义序列>
 <转义序列>::= 下列符号之一:
 \ ' \ b \ v \ xhh
 \ " \ f \ o \ xhhh
 \ ? \ n \ oo \ Xh
 \\ \ r \ ooo \ Xhh
 \ a \ t \ xh \ Xhhh

A.1.5 字符串常量

<字符串常量>::= "<S字符序列>"_{opt}
 <S字符序列>::= <S字符> | <S字符序列><S字符>
 <S字符>::= 源字符集中除双引号("), 反斜线(\)和换行符之外的任何字符 | <转义序列>

A.1.6 运算符

<运算符>::= 为下列符号之一:
 [] () . -> ++ --
 & * + - ~ !
 sizeof / % << >> <
 > <= >= == != ##
 ^ | && || ? : =
 *= /= %= += -= <<= >>=
 >>= &= ^= |= . #

A.1.7 标点字符

<标点字符>::= 为下列符号之一:
 [] () { } . , ;
 = ! ... #

A.2 语 法

A.2.1 表达式

<初等表达式> ::= <标识符> | <常量> | <伪变量> |

<字符串常量> | (<表达式>)

<伪变量> ::= 下列符号之一:

_AX _AL _AH _SI ES

_BX _BL _BH _DI _SS

_CX _CL _CH _BP _CS

_DX _DL _DH _SP _DS

<后缀表达式> ::= <初等表达式> | <后缀表达式> [<表达式>] |

<后缀表达式> (<变元表达式表> , p) |

<后缀表达式> * <标识符> | <后缀表达式> → <标识符> |

<后缀表达式> + + |

<后缀表达式> - -

<变元表达式表> ::= <赋值表达式> | <变元表达式表> , <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> | <单目运算符> <变类型表达式>

| sizeof <单目表达式> | sizeof (<类型名>)

<单目运算符> ::= & | * | + | - | ~ | !

<变类型表达式> ::= <单目表达式> | (<类型名>) <变类型表达式>

<乘除表达式> ::= <变类型表达式> |

<乘除表达式> * <变类型表达式> |

<乘除表达式> / <变类型表达式> |

<乘除表达式> % <变类型表达式>

<加减表达式> ::= <乘除表达式> |

<加减表达式> + <乘除表达式> |

<加减表达式> - <乘除表达式>

<移位表达式> ::= <加减表达式> |

<移位表达式> << <加减表达式> |

<移位表达式> >> <加减表达式>

<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>

<判等表达式> ::= <关系表达式> |

<判等表达式> == <关系表达式> |

<判等表达式> != <关系表达式>

<与表达式> ::= <判等表达式> |

<与表达式> & <判等表达式>

<异或表达式> ::= <与表达式> | <异或表达式> ^ <与表达式>

<同或表达式> ::= <异或表达式> | <同或表达式> | 注 <异或表达式>

<逻辑与表达式> ::= <同或表达式> |

注: 这里的 | 和下页的 || 是运算符。

<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> <赋值运算符> <赋值表达式>
 <赋值运算符> ::= 下列符号之一:
 = * = / = % = + = - =
 < < = > > = & = ^ = | =
 <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>

A.2.2 说明

<说明> ::= <说明指示符> <初始化说明表>_{opt}
 <说明指示符> ::= <存储类别指示符> <说明指示符>_{opt} |
 <类型指示符> <说明指示符>_{opt}
 <初始化说明符表> ::= <初始化说明符> |
 <初始化说明符表>, <初始化说明符>
 <初始化说明符> ::= <说明符> |
 <说明符> = <初始化元>
 <存储类别指示符> ::= typedef | extern | static | auto | register
 <类型指示符> ::= void | char | short | int | long | float | double | signed | unsigned |
 const | volatile | <结构或联合指示符> | <枚举指示符> <typedef名>
 <结构或联合指示符> ::= <结构或联合标识符>_{opt} { <结构说明表> } |
 <结构或联合标识符>
 <结构或联合标识符> ::= struct | union
 <结构说明表> ::= <结构说明> | <结构说明表> <结构说明>
 <结构说明> ::= <类型指示符表> <结构说明符表>;
 <类型指示符表> ::= <类型指示符> | <类型指示符表> <类型指示符>
 <结构说明符表> ::= <结构说明符> | <结构说明符表>, <结构说明符>
 <结构说明符> ::= <说明符> | <说明符>_{opt}; <常量表达式>
 <枚举指示符> ::= enum <标识符>_{opt} { <枚举项表> } | enum <标识符>
 <枚举项表> ::= <枚举项> | <枚举项表>, <枚举项>
 <枚举项> ::= <枚举常量> | <枚举常量> = <常量表达式>
 <说明符> ::= <指针>_{opt} <直接说明符> | <修饰符表>_{opt}
 <直接说明符> ::= <标识符> | (<说明符>) |
 <直接说明符> [<常量表达式>_{opt}] |
 <直接说明符> (<参数类型表>) |
 <直接说明符> (<标识符表>_{opt})
 <指针> ::= * <类型指示符表>_{opt} |
 * <类型指示符表>_{opt} <指针>
 <修饰符表> ::= <修饰符> | <修饰符表> <修饰符>
 <修饰符> ::= cdecl | pascal | interrupt | near | far | huge
 <参数类型表> ::= <参数表> | <参数表>, ...
 <参数表> ::= <参数说明> | <参数表>, <参数说明>
 <参数说明> ::= <说明指示符> <说明符> |
 <说明指示符> <抽象说明符>_{opt}
 <标识符表> ::= <标识符> | <标识符表>, <标识符>

<类型名> ::= <类型指示符表> <抽象说明符>_{opt}
 <抽象说明符> ::= <指针> | <指针>_{opt} <直接抽象说明符>_{opt} |
 <修饰符表>_{opt}
 <直接抽象说明符> ::= (<抽象说明符>) |
 <直接抽象说明符>_{opt} [<常量表达式>_{opt}] |
 <直接抽象说明符>_{opt} (<参数类型表>_{opt})
 <typedef名> ::= <标识符>
 <初始化元> ::= <赋值表达式> | { <初始化元表> } | { <初始化元表>, }
 <初始化元表> ::= <初始化元> | <初始化元表>, <初始化元>

A.2.3 语 句

<语句> ::= <带标号语句> | <复合语句> |
 <表达式语句> | <选择语句> |
 <循环语句> | <转向语句> |
 <asm语句>
 <asm语句> ::= asm <符号列> ; <换行符> | asm <符号列>;
 <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>_{opt} <语句表>_{opt} }
 <说明表> ::= <说明> | <说明表> <说明>
 <语句表> ::= <语句> | <语句表> <语句>
 <表达式语句> ::= <表达式>_{opt} ;
 <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(<表达式>_{opt}; <表达式>_{opt}; <表达式>_{opt}) <语句>
 <转向语句> ::= goto <标识符>; | continue; |
 break; | return <表达式>_{opt};

A.2.4 外部定义

<文件> ::= <外部定义> | <文件> <外部定义>
 <外部定义> ::= <函数定义> | <说明>
 <asm语句> ::= asm <符号列> <换行符> | asm <符号列>;
 <函数定义> ::= <说明指示符>_{opt} <说明符>
 <说明表>_{opt} <复合语句>

A.3 预 处 理 指 令

<预处理文件> ::= <组>
 <组> ::= <组部分> | <组> <组部分>
 <组部分> ::= <pp符号序列>_{opt} <换行符> | <if节> | <控制行>
 <if节> ::= <if组> <elif组>_{opt} <else组>_{opt}
 <end if行>
 <if组> ::= # if <常量表达式> <换行符> <组>_{opt} |
 # ifdef <标识符> <换行符> <组>_{opt} |
 # ifndef <标识符> <换行符> <组>_{opt}

<elif组列> ::= <elif组> | <elif组列><elif组>
 <elif组> ::= #elif<常量表达式><换行符><组>_{opt}
 <else组> ::= #else<换行符><组>_{opt}
 <endif行> ::= #endif<换行符>
 <控制行> ::= #include<pp符号序列><换行符> |
 #define<标识符><置换表><换行符> |
 #define<标识符><左括号><标识符表>_{opt}>><置换表><换行符> |
 #undef<标识符><换行符> |
 #line<pp符号序列><换行符> |
 #error<pp符号序列>_{opt}<换行符> |
 #pragma<pp符号序列>_{opt}<换行符> |
 #pragma warn<动作符><缩写符><换行符>
 #pragma inline<换行符> |
 #<换行符>
 <动作符> ::= + | - | ,
 <缩写符> ::= 下列符号之一,
 amb dyn pia str
 amp dup pro stu
 apt eff rch stv
 ans fun ret sus
 cfn ign rpt use
 cpt mot rvl voi
 def par sig zst
 <左括号> ::= 无前置空格的左括号字符 “(”
 <置换表> ::= <pp符号序列>_{opt}
 <pp符号序列> ::= <预处理符号> | <pp符号序列><预处理符号>
 <预处理符号> ::= 仅用在#include指令内的<嵌入名> |
 无关键字之分的<标识符> |
 <常量> | <字符串常量> | <运算符> |
 <标点符号> | 除上述符号外的非空格字符
 <嵌入名> ::= <h字符序列>
 <h字符序列> ::= <h字符> | <h字符序列><h字符>
 <h字符> ::= 源字符集中除换行符和大于符号(>)外的任何字符
 <换行符> ::= 换行字符

附录B Turbo C字符屏幕管理和图形处理库函数

Turbo C 配备有四百多个库函数和宏。C 程序员可以从自己的程序内调用这些例行程序以便执行广泛的任务, 这些任务包括低级的和高级的输入/输出、串和文件操作、存储分配、进程控制、数据转换、数学计算、正文处理和作图等等。

Turbo C 的例行程序包含在库文件(Cx.LIB和MATHx.LIB)中。因为 Turbo C 支持六种不同的存储模式, 所以每一种模式都有相应的库文件和数学文件, 这些文件包含为特殊模式书写的例行程序描述。

Turbo C 支持 ANSI C 标准, 其中包括允许给出用户C程序中库函数的原型。所有的 Turbo C 库例行程序都在一个或多个嵌入文件(即 .H 文件或从配给磁盘拷贝到 INCLUDE 目录中的“包括文件”中说明了原型。

由于 Turbo C 提供的库函数之多, 要一一详细介绍其细节, 是本书篇幅所不允许的。因此本附录只对字符屏幕管理和图形处理两类函数作详细介绍(这两类函数也是 Turbo C(1.0版)所不具有的)。字符屏幕管理类库函数的原型说明文件为 conio.h, 图形处理库函数的原型说明文件为 graphics.h。

本附录的主要内容包括:

(1) 给出两类函数查询的索引(包括: 函数名, 功能以及在 B.2 节中的序号)。

(2) 按字母顺序列出并简介字符屏幕管理和图形处理的全部函数。

欲了解其他各类函数的详细信息以及在这些函数中设置的公共全程变量的读者, 可查阅参考文献 [2]、[3] 和 [13]。

B.1 函数索引

类别	函 数 名	功 能	No
一、 字 符 屏 幕 管 理 函 数	clrcol	清除正文窗口中的内容直到行末	8
	clrscr	清除正文模式窗口	9
	delline	删除正文窗口中的行	10
	gettext	复制正文模式屏幕的正文到内存	31
	gettextinfo	取正文模式显示信息	32
	gotoxy	在正文窗口中定位光标	37
	highvideo	选择高亮度的正文字符	43
	insline	在正文窗口插入空行	46
	lowvideo	选择低亮度字符	50
	movctext	把屏幕正文从一个矩形区域拷贝到另一矩形区域	52
	normvideo	选择标准亮度字符	54
	puttext	从内存复制正文到屏幕	50

类别	函 数 名	功 能	No
一	textattr	设置正文属性	80
	textbackground	选择新的正文背景颜色	81
	textcolor	在正文模式中选择新的字符颜色	82
	textmode	设置屏幕为正文模式	84
	wherex	给出窗口内水平光标位置	86
	wherey	给出窗口内垂直光标位置	87
	window	定义活动正文模式窗口	88
二、 图 形 处 理 函 数	arc	画圆弧	1
	bar	画条形图	2
	bar3d	画三维条形图	3
	circle	画圆	4
	cleardevice	清除图形屏幕	5
	clearviewport	清除当前视区	6
	closegraph	关闭图形系统	7
	detectgraph	决定要使用的图形驱动器和模式	11
	drawpoly	画多边形	12
	ellipse	画椭圆	13
	fillpoly	画多边形并着色	14
	floodfill	对一个有界区域着色	15
	getarcoords	取arc的上一次调用的坐标	16
	getaspectratio	返回当前图形模式的纵横比	17
	getbkcolor	返回当前背景颜色	18
	getcolor	返回当前绘图颜色	19
	getfillpattern	复制一个用户定义的着色模式到内存	20
	getfillsettings	取有关当前填充模式和颜色信息	21
	getgraphmode	返回当前图形模式	22
	getimage	保存指定区域的位图象到内存	23
	getlinesettings	取当前画线类型、模式和宽度	24
	getmaxcolor	返回最大的颜色值	25
	getmaxx	返回屏幕最大的X坐标	26

类别	函 数 名	功 能	No
图 形 处 理 函 数	getmaxy	返回屏幕最大的Y坐标	27
	getmoderange	取一给定的图形驱动器的模式范围	28
	getpalette	返回当前调色板的信息	29
	getpixel	取指定象素的颜色	30
	gettextsettings	返回当前正文设置的信息	33
	getviewsettings	返回当前视区的信息	34
	getx	返回当前位置的X坐标	35
	gety	返回当前位置的Y坐标	36
	graphdefaults	将所有图形设置复位成其缺省值	38
	grapherrormsg	返回错误信息串	39
	-graphfreemem	释放用户可修改的图形存储器	40
	-graphgetmem	分配用户可修改的图形存储器	41
	graphresult	返回上次不成功的图形操作错误代码	42
	imagesize	返回存储一个位图象所需的字节数	44
	initgraph	初始化图形系统	45
	line	画线	47
	linereel	从当前位置(cp)到与CP有一个相对距离的点画一条线	48
	lineto	从cp到(x, y)之间画一条线	49
	moverel	把当前位置(cp)移动一段相对距离	51
	moveto	移动cp到(x, y)处	53
	outtext	在视区显示一字符串	55
	outtextxy	发送一个字符串到指定位置	56
	pieslice	画扇形图并着色	57
	putimage	在屏幕上显示一个位图象	58
	putpixel	在指定处画一个象素	59
	rectangle	画矩形	61
	registebgidriver	注册已连接的图形驱动器代码	62
	regis erbg ifount	注册已连接的笔划字体代码	63
	restorecrtmode	恢复屏幕模式为其初始化设置值	64

类别	函 数 名	功 能	No
图 形 处 理 函 数	setactivepage	为图形输出设置活动页	65
	setallpalette	改变所有调色板颜色为指定色	66
	setbkcolor	利用调色板设置当前背景颜色	67
	setcolor	利用调色板设置当前绘图颜色	68
	setfillpattern	选择用户定义的着色模式	69
	setfillstyle	设置着色模式和颜色	70
	setgraphbufsize	设置内部图形缓存区大小	71
	setgraphmode	设置图形模式并清屏	72
	setlinestyle	设置当前画线的宽度和类型	73
	setpalette	改变一种调色板颜色	74
	settextjustify	设置正文对齐方式	75
	settextstyle	设置当前正文属性	76
	setusercharsize	为笔划字形用户自定义字符放大因子	77
	setviewport	为图形输出设置当前视区	78
	setvisualpage	设置可见图形页号	79
	textheight	返回一字符串高度的图素数	83
	textwidth	返回一字符串宽度的图素数	85

B.2 按字母顺序组织的库函数

B.2.1 库函数描述说明

在给出按字母顺序组织的库函数之前,先介绍所采用的描述格式和内容。

在B.2.2中描述的库函数均按下述格式给出有关信息:

编 号 该编号就是B.1中索引栏中使用的序号;

函 数 名 紧接编号后的是函数名,然后跟以一个破折号,后面是函数功能的摘要。

用 法 列出必须包括的嵌入文件并说明语法。

相关函数用法 指出使用当前介绍的函数时涉及到的有关函数的用法。

说 明 描述函数的功能,它所使用的参数和使用函数需要的所有细节、有关的函数和列表等信息。

返回值 如果函数有返回值,在此段给出。如果全局变量errno有值,也在此段列出。若无返回值,则此段不出现。

可移植性 说明函数适用的系统。

参 阅 列出有助于理解函数功能和用法的其他函数。

例 子 提供函数用法的例子程序或程序片段。

B.2.2 字符屏幕管理和图形处理库函数

(1) arc——画圆弧

用 法 `#include <graphics.h>`

`void far arc (int x, int y, int stangle, int endangle, int radius);`

相关函数

用 法 `void far circle (int x, int y, int radius);`

`void far ellipse(int x, int y, int stangle, int endangle,`

`int xradius, int yradius);`

`void far getarccoords (struct arccoordstype, far* arccoords);`

`void far getaspectratio(int far*xasp, int far*yasp);`

`void far pieslice (int x, int y, int stangle, int endangle, int radius);`

说 明 这里描述的四种绘图函数(arc, circle, ellipse和pieslice均用当前的绘图颜色绘出各自形状。arc以(x,y)为圆心和radius给出的半径画一圆弧,并从起始角度stangle画到终止角度endangle,若stangle=0且endangle=360,则对arc的调用就是画一个完整的圆。

circle函数是画圆,圆心(x,y),半径由radius给出。

ellipse函数是画椭圆,其中心位于(x,y),其水平轴与垂直轴的大小分别由参数xradius和yradius给出。ellipse函数从起始角stangle画到终止角endangle,若stangle=0且endangle=360,则对ellipse函数的调用将是画一个完整的椭圆。

pieslice函数是画并充填(即着色)一个扇形图,圆心位于(x,y)处,半径由参数radius提供。其中圆是从stangle画到endangle。该圆轮廓的颜色为当前的绘图颜色,而充填使用当前充填模式和充填颜色。

arc, ellipse, pieslice的角度均在时钟记数器范围内。0度在3点钟,90度在12点钟等等。

每一个图形驱动器及图形模式均有一个纵横比与它们相联系,该纵横比是由arc, circle及pieslice函数在屏幕上保证画圆而使用的比例因子。该纵横比可由调用getaspectratio过程来计算。它可控制*xasp和*yasp。

y纵横因子即*yasp被规格化为10000;除了VGA,在所有图形适配器上,*xasp(x纵横因子)比*yasp小,因为图素的高度总是大于它们的宽度。在VGA上,有“方形”图素,其中*xasp=*yasp。*yasp与*xasp之间的关系大致可表示为:

`*yasp = 10000`

`*xasp <= 10000`

getarccoords过程是由arccoords指向的arccoordstype结构中填入有关上一次对arc调用的信息。

arccoordstype结构在GRAPHICS.H中定义成如下形式:

```
struct arccoordstype {
    int x, y;
    int xstart, ystart, xend, yend;
};
```

该结构的成员用来说明arc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arc函数的末尾画一条线, 则这些值是很有用的.

返回值 如果在填充扇形图时出现了错误, 则graphresult函数将返回-8值.

可移植性 Turbo Pascal 4.0 中存在类似的函数.

参 阅 getfillsettings

例 子

```
#include <graphics.h>
main()
{ int graphdriver = DETECT, graphmode; /* 需要自动检测 */
  struct arccoordstype arcinfo;
  int xasp, yasp;
  long xlong;
  initgraph(&graphdriver, &graphmode, ""); /* 初始化图形 */

  /* 画一半径为 50 的 90 度的圆弧 */
  arc(150, 150, 0, 89, 50);

  /* 取弧的坐标, 并连接两端点 */
  getarccoords(&arcinfo);
  line(arcinfo.xstart, arcinfo.ystart, arcinfo.xend, arcinfo.yend);

  /* 画一圆 */
  circle(150, 150, 100);

  /* 在圆中画一椭圆 */
  ellipse(150, 150, 0, 359, 100, 50);

  /* 画圆饼图 */
  setcolor(WHITE); /* 用白线画边框 */
  setfillstyle(SOLID_FILL, LIGHTRED);
  pieslice(100, 100, 0, 135, 49);
  setfillstyle(SOLID_FILL, LIGHTBLUE);
  pieslice(100, 100, 135, 225, 49);
  setfillstyle(SOLID_FILL, WHITE);
  pieslice(100, 100, 225, 360, 49);

  /* 画一"正"方形图 */
  getaspectratio(&xasp, &yasp);
  xlong = (100L * (long) yasp) / (long) xasp;
  rectangle(0, 0, (int) xlong, 100);

  closegraph();
}
```

(2) bar——画条形图

用 法 #include<graphics.h>

相关函数 void far bar(int left, int top, int right, int bottom);

用 法 void far bar3d(int left, int top, int right,
int bottom, int depth, int topflag);

说 明 bar画长方条形图并着色。该条形图用当前的填充模式及当前填充颜色进行着色。bar并不画出条形图的外轮廓，对于画一个两维的条形图，可使用bar3d并令depth=0。

bar3d画一个三维的矩形条形图并用当前的填充模式和填充颜色着色。三维条形图的轮廓由当前的线类型和颜色画出。用图素表示的条形图深度由参数depth提供。

topflag参数用来决定是否在条形图上放一个三维顶。若topflag不为0，则放一个顶，否则不放顶（使得有可能将几个条形从顶端累积起来。在两个函数中，矩形的左上角和右上角分别由(left, top)和(right, bottom)给出。要计算一个典型的bar3d的深度，并取原条形图的1/4的宽度，可按如下方法计算：bar3d(left, top, bottom, (right-left)/4, 1);

可移植性：在Turbo Pascal 4.0中存在类似的函数。

参 阅 getbkcolor, getfillsettings, getlinegstyle, graphresult, rectangle

例 子

```
#include <graphics.h>
```

```
main()
```

```
{ int graphdriver = DETECT, graphmode; /* 需要自动检测 */
```

```
initgraph(&graphdriver, &graphmode, ""); /* 初始化图形 */
```

```
setfillstyle(SOLID_FILL, MAGENTA);
```

```
bar3d(100, 10, 200, 100, 5, 1);
```

```
setfillstyle(BATCH_FILL, RED);
```

```
bar(30, 30, 80, 60);
```

```
closegraph();
```

```
};
```

(3) bar3d——画三维条形图

用 法 #include<graphics.h>

void far bar3d(int left, int top, int right,
int bottom, int depth, int topflag);

说 明 参见bar。

(4) circle——画圆

用 法 #include<graphics.h>

void far circle(int x, int y, int radius);

说 明 参见arc。

(5) cleardevice——清除图形屏幕

用 法 #include<graphics.h>

void far cleardevice(void);

说 明 cleardevice擦除整个图形屏幕，并将cp(当前位置)移至原点(0,0)。

可移植性 在Turbo Pascal 4.0中存在相似的例行子程序。

参 阅 clearviewport

(6) clearviewport——清除当前的视区

用 法 #include<graphics.h>

`void far clearviewport(void);`

说明 `cleardevice` 擦除视区并将 `cp` (当前位置) 移至原点 (0, 0)。

可移植性 Turbo Pascal 4.0 中存在相似的例行子程序。

参阅 `getviewsettings`, `cleardevice`

例子 `setviewport(30, 30, 130, 130, 0); outtexty(10, 10, "Hit any key to clear viewport..."); getch(); clearviewport();`

(7) `closegraph`——关闭图形系统

用法 `#include <graphisc.h>`

`void far closegraph(void);`

说明 参见 `initgraph`

(8) `clreol`——清除正文窗口中的内容直到行末

用法 `void clreol(void);`

说明 `clreol` 在当前正文窗口中将从光标处至行末之间的所有字符删除, 删除过程中光标不移动。

可移植性 该函数仅对 IBM PC 及其兼容机有效, 在 Turbo Pascal 中存在对应的函数。

参阅 `clrscr`, `delline`, `window`

(9) `clrscr`——清除正文模式窗口

用法 `void clrscr(void);`

说明 `clrscr` 清除当前正文窗口, 并将光标移至左上角, 即位置 (1, 1)。

可移植性 该函数仅适用于 IBM PC 及其兼容机在 Turbo Pascal 中存在对应的函数。

参阅 `clreol`, `delline`, `window`

(10) `delline`——删除正文窗口的行

用法 `void delline(void);`

说明 `delline` 删除光标所在的行, 其后各行均向上移一行。 `delline` 在当前激活的正文窗口中操作。

可移植性 该函数适用于 IBM PC 及其兼容机。 Turbo Pascal 中存在对应的函数。

参阅 `clreol`, `insline`, `window`

(11) `detectgraph`——通过检查硬件决定要使用的图形驱动程序和模式

用法 `#include <graphics.h>`

`void far detectgraph(int far*graphdriver, int far*graphmode);`

说明 参见 `initgraph`

(12) `drawpoly`——画多边形

用法 `#include <graphies.h>`

`void far drawpoly(int numpoints, int far*polypoints);`

相关函数

用法 `void far fillpoly(int numpoints, int far*polypoints);`

说明 `drawpoly` 画一个顶点数为 `numpoints` 的多边形, 使用当前的线类型和颜色。 `fillpoly` 用当前的画线类型和颜色画一个多边形 (正如 `drawpoly` 所作)。 然后用当前着色类型和填充颜色涂这个多边形。 `polypoints` 指向一个整型序列 (共有 `numpoints * 2` 个整型数)。 每一对整数给出了该多边形顶点的坐标 `x` 和 `y`。

注意, 为了画一个 `n` 条边形的封闭图形, 必须将 `n+1` 个坐标点送往 `drawpoly`, 其中第 `n` 个坐标点与第 0 个坐标点相同。

返回值 如果在填充多边形时出现了错误, 则函数 `graphresult` 返回 -6 值。

可移植性 在Turbo Pascal 4.0 中存在相似的函数

参 阅 getfillsettings, getlinesettings, getbkcolor, graphresult

例 子

```
#include <graphics.h>

main()
{ int graphdriver = DETECT, graphmode;      /* 需要自动检测 */
  int triangle[] = {50, 100, 100, 100, 150, 150, 50, 100};
  int rhombus[] = {50, 10, 90, 50, 50, 90, 10, 50};

  initgraph(&graphdriver, &graphmode, "");    /* 初始化图形 */

  /* 画一个三角形 */
  drawpoly(sizeof(triangle)/(2*sizeof(int)), triangle);

  /* 画并填一个菱形 */
  fillpoly(sizeof(rhombus)/(2*sizeof(int)), rhombus);

  closegraph();
}
```

(13) ellipse——画椭圆弧

用 法 #include<graphics.h>

```
void far ellipse(int x, int y, int stangle, int endangle,
                 int xradius, int yradius);
```

说 明 参见arc

(14) fillpoly——画一个多边形并着色

用 法 #include<graphics.h>

```
void far fillpoly(int numpoints, int far *polypoints);
```

说 明 见drawpoly

(15) floodfill——对一个有界区域着色

用 法 #include<graphics.h>

```
void far floodfill(int x, int y, int border);
```

说 明 floodfill是在位图形设备上充填一个封闭区域。(x, y)是封闭区域内将填充的“起点”(seed point), 由颜色border所围起来的这个区域将用当前的填充模式和填充颜色充填。如果起点在一个封闭区域内, 则其内将被充填。若起点在封闭区域之外, 则其外部将被充填。如果可能用fillpoly代替floodfill, 这样可与将来的版本保持代码的兼容性。

返回值 若在充填区域时出现了错误, graphresult将返回-7值。

可移植性 在Turbo Pascal 4.0 中存在类似的函数。

参 阅 drawpoly, getbkcolor, getfillsettings, getlinesettings, graphresult

例 子

```

#include <graphics.h>

main()
{
    int graphdriver = DETECT, graphmode;
    initgraph(&graphdriver, &graphmode, "");
    setcolor(WHITE);
    setfillstyle(HATCH_FILL, LIGHTMAGENTA);
    bar3d(10, 10, 100, 100, 10, 1);
    setfillstyle(SOLID_FILL, LIGHTGREEN);
    floodfill(102, 50, WHITE);
    floodfill(50, 0, WHITE);
    closegraph();
}

```

/* 需要自动检测 */
/* 初始化图形 */
/* 填充边 */
/* 填充顶端 */

(16) getarccoords——取arc的上一次调用的坐标

用法 #include<graphics.h>
void far getarccoords(struct arccoordstype far*arccoods);

说明 参见arc

(17) getaspectratio——返回当前图形模式的纵横比

用法 #include<graphics.h>
void far getaspectratio (int far *xasp, int far *yasp);

说明 参见arc

(18) getbkcolor——返回当前背景颜色

用法 #include<graphics.h>
int far getbkcolor(void);

相关函数

用法 void far setbkcolor (void);

说明 getbkcolor返回当前背景颜色(细节参见以下表格)。

setbkcolor将背景置成由参数color说明的颜色。参数color可为名字或数字,如表B.1所列。

表B.1

数字	名	数字	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

注,这些符号名均在GRAPHICS.H中定义。

例如,如果想将背景置为蓝色,则可调用setbkcolor(BLUE)或setbkcolor(1)。

对于CGA或EGA系统, setbkcolor的参数指出当前调色板的项数,而不是指出一种特殊的颜色(除了参数值为0,将把黑色作为背景颜色的情况)。setbkcolor可通过改变调色板上的第一项从而改变背景颜色。

注:如果在使用EGA或VGA系统时,调用setpalette或setallpalette函数改变了调色板颜色,则所定义的符号常量可能就不会给出正确的颜色。

返回值 getbkcolor返回当前背景颜色, setbcolor无返回值。

可移植性 在Turbo Pascal 4.0中存在类似的函数。

● 阅 getpalette, initgraph

例子

```
#include <graphics.h>

main()
{ int graphdriver = DETECT, graphmode; /* 需要自动检测 */
  int svcolor;

  initgraph(&graphdriver, &graphmode, ""); /* 初始化图形 */

  svcolor = getcolor(); /* 存当前背景颜色 */
  setbkcolor(svcolor ^ 1); /* 改变背景颜色 */
  delay(5000); /* 等5秒钟 */
  setbkcolor(svcolor); /* 重存旧背景颜色 */

  closegraph();
};
```

(19) getcolor——返回当前绘图颜色

用法 #include <graphics.h>

int far getcolor(void);

相关函数

用法 void far setcolor(int color);

说明 getcolor返回当前绘图颜色。setcolor将当前绘图颜色置为color, 参数color可从0变化到getmaxcolor()。绘图颜色是指在画线及其他图形的时候, 图素所置的值。例如, 在CGA模式下, 调色板包括四种颜色, 背景颜色, 淡绿色, 浅红色和黄色。在这个模式下, 如果getcolor()的返回值为1, 则当前绘图颜色为淡绿色; 类似地, setcolor(3)选择黄色作为绘图颜色。

返回值 getcolor返回当前绘图颜色, setcolor无返回值。

可移植性 在Turbo Pascal 4.0中存相似的函数。

参 阅 getpalette, getmaxcolor

例子

```
#include <graphics.h>

main()
{ int graphdriver = DETECT, graphmode; /* 需要自动检测 */
  int svcolor;

  initgraph(&graphdriver, &graphmode, ""); /* 初始化图形 */

  svcolor = getcolor(); /* 存当前绘图颜色 */
  setcolor(3); /* 置绘图颜色为存在调色板# 3入口处的颜色 */
  circle(100, 100, 5); /* 小的有色圆图 */
  setcolor(svcolor); /* 重存旧的绘图颜色 */

  closegraph();
};
```

(20) getfillpattern——复制一个用户定义的着色模式到内存

用 法 #include<graphics.h>

void far getfillpattern(char far*upattern);

相关函数

用 法 void far setfillpattern(char far*upattern, int color);

说 明 getfillpattern将用户定义的着色模式复制到由upattern所指向的八个字节的存储区, 该用户定义的模式同 setfillpattern 所置的一样。setfillpattern 同 setfillstyle 相似, 所不同的是, setfillpattern设置用户定义的8×8模式而不是预定义的模式。upattern是一个指针, 指向一个8字节的序列, 每个字节与该模式下的8个图素相对应。只要其中某个模式字节的某一位为1, 则其相应的图素将被绘出。例如, 如下用户定义的着色模式代表了一个检测板:

```
char checkboard [8] = {
    0XAA, 0X55, 0XAA, 0X55, 0XAA, 0X55, 0XAA, 0X55 };

```

可移植性 在Turbo Pascal 4.0中存在相似的函数

参 阅 getfillsettings

(21) getfillsettings——取有关当前着色模式和颜色信息

用 法 #include<graphics.h>

void far getfillsettings(struct fillsettingstype far*fillinfo);

相关函数

用 法 void far setfillstyle(int pattern, int color);

说 明 函数bar, bar3d, fillpoly, floodfill及pieslice都是用当前着色模式和当前着色颜色来着色一块区域。有11种预定义的着色模式类型(例如, 实心、阴影、虚点等)。预定义模式的符号名由GRAPHICS.H中的列表fill_patterns提供(见如下表格)。另外, 可以定义用户自己的着色模式。

getfillsettings是用有关当前着色模式和着色颜色的信息着色fillinfo指向的fillsettingstype结构, 在GRAPHICS.H中如下定义:

```
struct fillsettingstype {
    int pattern    /*当前着色模式*/
    int color      /*当前着色颜色*/
};

```

如果pattern=12(USER_FILL), 则使用用户定义的着色模式; 否则由参数pattern给出预定义模式数。

setfillstyle置当前着色模式和颜色。若要置一个用户定义的着色模式, 则不是将setfillstyle的值取模式12, 而是调用setfillpattern。在GRAPHICS.H中定义的列表fill_pattern为预定义着色模式提供名字, 并为用户定义的模式提供一个指针(见表B.2)。

表B.2

名 称	值	描 述
EMPTY_FILL	0	用背景颜色着色
SOLID_FILL	1	单色着色
LINE_FILL	2	用 “—” 着色
LTSLASH_FILL	3	用 “///” 着色
SLASH_FILL	4	用粗线 “///” 着色
BKSLASH_FILL	5	用粗线 “\\” 着色
LTBKSLASH_FILL	6	用 “\\” 着色
HATCH_FILL	7	用淡影线着色
XHATCH_FILL	8	用深色的交叉形影线着色
INTERLEAVE_FILL	9	用交错线着色
WIDEDOT_FILL	10	用松散的空白点着色
CLOSEDOT_FILL	11	用紧凑的空白点着色
USER_FILL	12	用户定义的着色模式

除了EMPTY_FILL, 所有模式均用当前着色颜色填充; EMPTY_FILL则使用当前背景颜色。
返回值 无。但如果把无效的输入传送给setfillstyle, 则graphresult将返回-11值, 而当前着色模式和着色颜色将保持不变。

可移植性 在Turbo Pascal 4.0 中存在相似的函数

参 阅 arc, bar, fillpoly, floodfill, getfillpattern

例 子

```
#include <graphics.h>

main()
{ int graphdriver = DETECT, graphmode; /* 需要自动检测 */

  struct fillsettingstype save;
  char savepattern[8];
  char gray50[] = { 0xaa, 0x55, 0xaa, 0x55, 0xaa, 0x55, 0xaa, 0x55 };

  initgraph(&graphdriver, &graphmode, ""); /* 初始化图形 */

  getfillsettings(&save); /* 检索当前环境 */
  if (save.pattern == USER_FILL) /* 如果是用户定义的程序 */
    getfillpattern(savepattern); /* 存储用户着色模式 */
  setfillstyle(SLASH_FILL, BLUE); /* 改变着色类型 */
  bar(0, 0, 100, 100); /* 画slash-fill模式下的兰色条形 */
  setfillpattern(gray50, YELLOW); /* 用户着色模式 */
  bar(100, 100, 200, 200); /* 画用户的黄色条形 */

  if (save.pattern == USER_FILL) /* 如果是用户定义的模式 */
    setfillpattern(savepattern, savecolor); /* 重存用户着色模式 */
  else
    setfillstyle(save.pattern, save.color); /* 重存旧模式 */

  closegraph();
};
```

(22) getgraphmode——返回当前图形模式

用 法 #include <graphics.h>
 int far getgraphmode(void);

相关函数

用 法 void far setgraphmode(int mode);

说 明 getgraphmode返回由initgraph或setgraphmode所置的当前图形模式。setgraphmode选择一个与initgraph所置标准值不同的图形模式。参数mode对于当前设备驱动器来说应该是一个有效的模式。

setgraphmode清除屏幕并且将所有图形设置复位成它们的标准值(cp、调色板、颜色、视区以及其他等)。可以使用setgraphmode和restorecrtmode来实现正文与图形模式之间的转换。

程序在调用这些函数之前必须首先成功地调用一次initgraph。

GRAPHICS.H中定义的枚举项graphics_mode为预定义的图形模式提供了名字。可用一张表格列出这些枚举值。这可参见initgraph的描述。

返回值 无。若对当前设备驱动器给予setgraphmode一个非法的模式, graphresult将返回-10。

可移植性 在Turbo Pascal 4.0 中存在类似的函数。

参 阅 getmoderange, initgraph, restorecrtmode

```

int cmode;

cmode = getgraphmode();    /* 存当前模式 */
restorecrtmode();          /* 转换成正文 */
printf("Now in text mode -press any key to go back to graph ...");
getch();
setgraphmode(cmode);       /* 重置成图形 */

```

(23) **getimage**——将指定区域的一个位图象保存到内存

用 法 #include<graphics.h>
 void far getimage(int left,int top,int right,int bottom,
 void far*bitmap);

相关函数

用 法 unsigned far imagesize(int left,int top,int right,int bottom)
 void far putimage(int left, int top,void far*bitmap,int op)

说 明 这三个函数用来把屏幕上的一个图象复制到主存储器，然后将它放回到屏幕上。

getimage将屏幕上的一个矩形区域的位图象存储到主存储器中。left,top, right和bottom这四个参数定义了屏幕上的矩形。bitmap指向主存储器中将存储这个位图象的区域。该区域的头两个字节用来存放矩形的宽和高；其余部分存放位图象本身。

imagesize决定了getimage用于存放这个指定的矩形所需的字节数。返回的图象大小应包括存放矩形的宽和高所占的空间。

putimage将先前由getimage所存的位图象重新放回屏幕上，图象的左上角位于(left, top)处。bitmap指向存放源图象的内存区域。

putimage中的参数op说明了一个组合操作符，用来控制如何计算目标图素的颜色，它是基于已在屏幕上的图素和在内存的对应源图素来计算的。在GRAPHICS.H中定义的枚举putimage_ops给出了这些操作符的名字，见表B.3。也就是说，COPY_PUT将源位图象复制到屏幕上，XOR_PUT把源图象与已在屏幕上的图象进行异或操作，OR_PUT把源图象与已在屏幕上图象进行或操作，等等。

表B.3

名 称	值	描 述
COPY_PUT	0	复制
XOR_PUT	1	“异或”
OR_PUT	2	“或”
AND_PUT	3	与
NOT_PUT	4	复制源图象的逆元素

返回值 imagesize返回所需的存储区大小，若所选择的图象所需的大小大于或等于64K字节，imagesize将返回0xFFFF(-1)值，getimage和putimage返回空。

可移植性 Turbo Pascal 4.0中存在相似的函数。

例 子

```
#include <graphics.h>
main()
{ int graphdriver = DETECT, graphmode; /*需自动检测 */
  void * buffer;
  unsigned size;
  initgraph(&graphdriver, &graphmode, ""); /* 初始化图形 */

  size = imagesize(0, 0, 20, 10);
  buffer = malloc (size); /* 为图象分配内存 */
  getimage(0, 0, 20, 10, buffer); /* 存位 */

  /* ... */

  putimage(0, 0, buffer, COPY_PUT); /* 重存位 */
  free(buffer); /* 释放缓冲区 */

  colsograph();
}
```

(24) **getlinesettings**——取当前画线类型、模式以及宽度

用 法 #include<graphic.h>

```
void far getlinesettings(struct linesettingstype far*lineinfo);
```

相关函数

用 法 void far setlinestyle(int linestyle,unsigned upattern,int thickness);

说 明 getlinesettings用有关当前画线类型、模式和宽度的信息填充由lineinfo所指的linesettingstype结构。可通过调用setlinestyle来改变这些值。这个函数为所有由line、lineto、rectangle、drawpoly、arc、circle、ellipse、pieslice等所画的线条设置类型。

linesettingstype结构在GRAPHICS.H中如下定义:

```
struct linesettingstype {
    int linestyle;
    unsigned upattern;
    int thickness;
}
```

linestyles说明了以何种类型或称样式来画后续线条(例如实线、点虚线、中心线、破折线),在GRAPHICS.H中定义的枚举lines_tylet,给出了这些操作符的名称(见表B.4)。

表B.4

名 称	值	描 述
SOLID_LINE	0	实线
DOTTED_LINE	1	点线(点虚线)
CENTER_LINE	2	中心线(长短虚线)
DASHED_LINE	3	破折线(长虚线)
USERBIT_LINE	4	用户定义的画线类型

thickness说明后续所画的线的宽度为正常的还是加粗的。

名 称	值	描 述
NORM.WIDTN	1	1个图素宽
THICK.WIDTH	8	8个图素宽

upattern是一个16位模式,它只有当linestyle为USERBIT_LINE(4)时才起作用。在那种情形下,每当模式字中的一位为1时,则线上的相应因素用当前绘图颜色绘出。例如,实线与值为0xFFFF的upattern相对,而破折线与值为0x3333或0xF0F0的upattern相对应。如果setlinestyle的参数linestyle不为USERBIT_LINE(4),则upattern参数仍需要提供,只是不起作用。

返回值 无。若非法的输入传送给setlinestyle,则graphresult将返回-11。并且当前的画线类型保持不变。

可移植性 在Turbo Pascal 4.0中存在类似的函数

例子

```
#include <graphics.h>
main()
{ int graphdriver = DETECT, graphmode; /*将自动检测 */
  struct linesettingstype saveline;
  initgraph(&graphdriver, &graphmode, ""); /* 初始化图形 */

  getlinesettings(&saveline); /* 存当前线类型 */
  setlinestyle(SOLID_LINE, 0, THICK_WIDTH);
  rectangle(10, 10, 17, 15); /* 画有点粗的线框 */
  setlinestyle(saveline.linestyle, /* 重存旧线设置 */
               saveline.upattern,
               saveline.thickness);

  colsegraph();
```

(25) getmaxcolor——返回最大的颜色值

用法 #include<graphics.h>

int far getmaxcolor(void);

说明 getmaxcolor返回当前图形驱动器和模式下的最大的有效因素值(调色板size-1)。

返回值 getmaxcolor返回最大的有效颜色值。

可移植性 在Turbo Pascal 4.0中存在相似的函数。

参阅 getbkcolor, getpalette

(26) getmaxx——返回屏幕最大的x坐标

用法 #include<graphics.h>

int far getmaxx(void);

相关函数

用法 int far getmaxy(void);

说明 getmaxx返回当前图形驱动器和模式下最大的(相对于屏幕)x值。getmaxy返回当前图形驱动器和模式下最大的(相对于屏幕)y值。例如:在CGA的320×200模式下, getmaxx返回值319,而getmaxy返回值199, getmaxx和getmaxy为一个屏幕上的区域确定中心和确定区域边界是非常有用的。

返回值 getmaxx返回屏幕的最大x坐标, getmaxy返回屏幕的最大y坐标。

可移植性 在Turbo Pascal 4.0中存在相似的函数。

参见 getx

例子 printf("The screen resolution is %d pixels by %d pixels.\n",
getmaxx() + 1, getmaxy() + 1);

(27) **getmaxy**——返回屏幕的最大y坐标

用 法 `#include <graphics.h>`
`int far getmaxy(void);`
说 明 见 **getmaxx**

(28) **getmoderange**——为一个给定的图形驱动器取模式的范围

用 法 `#include <graphics.h>`
`void far getmoderange(int gaphdriver, int far *lomode,`
`int far *himode);`

说 明 **getmoderange**取一个给定的图形驱动器`gaphdriver`的有效图形模式的范围,最低允许的模式值由`*lomode`返回,最高值则由`*himode`返回。如果`gaphdriver`说明了一个非法的图形驱动器,则`*lomode`与`*himode`均被置为-1。

参 阅 `initgraph`, `getgraphmode`

例 子 `int lo, hi;`
`getmoderange(CGA &lo, &hi);`
`printf("CGA supports modes %d through %d\n", lo, hi);`

(29) **getpalette**——返回关于当前调色板的信息

用 法 `#include <graphics.h>`
`void far getpalette(struct palettetype far *palette);`

相关函数

用 法 `void far setallpalette(struct palettetype far *palette);`
`void far setpalette(int index, int actual_color);`

说 明

getpalette用有关当前调色板的大小和颜色的信息填入`palette`指向的`palettetype`结构中。

用户可以使用**setpalette**(或**setallpalette**)在EGA/VGA调色板中部分地(或全部地)改变颜色。在CGA中,只能调用**setpalette**改变调色板的第一项(`index = 0`, 背景颜色)。

setallpalette设置当前调色板为`palette`指向的`palettetype`结构中给出的值。

setpalette可将调色板中的`index`项改为`actual_color`。例如,**setpalette(0,5)**将当前调色板中的第一个颜色变为实际颜色数5(背景颜色)。如果`size`是当前调色板中的项数,则`index`可在0到(`size-1`)间变动。

`palettetype`结构(由**getpalette**和**setpalette**使用的)和`MAXCOLORS`常量在`GRAPHICS.H`中如下定义:

```
#define MAXCOLORS 15
struct palettetype {
    unsigned char size;
    signed char colors [MAXCOLORS + 1];
};
```

`size`给出了当前模式中当前图形驱动器中调色板的颜色数目。

`colors`是一`size`个字节的数组,它含有对应于调色板中每项的实际原始颜色编号。在**setallpalette**例行程序中,如果`colors`的某个元素为-1,则那个项的调色板颜色将不改变。

传送给**setpalette**的`actual_color`参数以及由**setallpalette**所使用的`colors`数组中的元素,都定义在`GRAPHICS.H`中,由符号常量表示(见表B.5)。

注意:有效的颜色取决于当前图形驱动器和当前图形模式。

调色板的变化可立即在屏幕上看到。每当改变一种调色板颜色时,屏幕上出现的所有那种颜色的象素都将改为新的颜色。

表 B.5

ACTUALCOLORTABLE			
CGA		EGA/VGA	
Name.....	Value	Name.....	Value
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_BOWN	20
LIGHTGRAY	7	EGA_LIGHTGRAY	7
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

返回值 无, 但如果传送给setpalette是非法输入, graphresult将返回-11值, 当前调色板不改变。

可移植性 在Turbo Pascal4.0中有对应函数。

参 阅 getbkcolor, getcolor

例 子

```
#include <graphics.h>
main()
{ int graphdriver = DETECT, graphmode; /* will request autodetection */
  struct palettetype palette;
  int color;
  initgraph(&graphdriver, &graphmode, ""); /* initialize graphics */

  getpalette(&palette); /* get current palette */
  for ( color = 0; color < palette.size; color++)
  { setfillstyle (SOLID_FILL, color); /* draw some colorful bars */
    bar(20 * (color-1), 0, 20 * color, 20);
  };
  if (palette.size > 1) /* only if more than 1 color */
  { do /* switch colors randomly */
    setpalette(random(palette.size), random(palette.size));
    while (!kbhit()); /* until a key is hit */
    getch(); /* discard keystroke */
  };
  setpalette(&palette); /* restore original palette */

  colsegraph();
};
```

(30) **getpixel**——取指定象素的颜色

用 法 `#include <graphis.h>`
`int far getpixel(int x, int y);`

相关函数

用 法 `void far putpixel(int x, int y, int pixelcolor);`

说 明 `getpixel`取得位于(x,y)处的象素的颜色, `putpixel`用`pixelcolor`定义的颜色在(x,y)处画一个点。

返 回 值 `getpixel`返回所给出象素的颜色, `putpixel`返回空。

可移植性 在Turbo Pascal 4.0中有相应的函数。

参 见 `getimage`

例 子

```
#include <graphics.h>
main()
{
    int graphdriver = DETECT, graphmode;           /* 需自动检测 */

    int i, color, max;

    initgraph(&graphdriver, &graphmode, "");        /* 初始化图形 */

    max=getmaxcolor()+1; /* 改变一对角线上象素的颜色 */

    for (i = 1; i < 288; i++)
    { color = getpixel(i, i);
      putpixel(i, i, (color ^ i) * max);
    }

    colsograph();
};
```

31) **gettext**——复制正文模式屏幕上的正文到内存

用 法 `int gettext(int left, int top, int right,`
`int bottom, void *destin);`

相关函数

用 法 `int puttext(int left, int top, int right,`
`int bottom, void *source);`

说 明 `gettext`将一个屏幕上由`left`、`top`、`right`和`bottom`定义的矩形区的内容存入`destin`指向的内存区域。

`puttext`将`source`指向的内存区域的内容写入由`left`、`top`、`right`和`bottom`定义的屏幕上矩形区。所有的坐标都为绝对的屏幕坐标,而不是相对于窗口的坐标。

`gettext`将矩形区的内容从左向右,自上向下顺序地读入内存。`puttext`用同样方式将内存内容放入定义的矩形区中。

屏幕上的每个位置占两个内存字节,第一个字节为单元中的字符,第二个字节为单元的显示属性。一个长为`h`行,宽为`w`列的矩形所需空间定义如下:

$$\text{字节数} = (h \times w) \times 2$$

返 回 值 如果操作成功,这些函数返回1;如果失败(例如:所给的坐标超出当前屏幕模式的范围)则返回0。

可移植性 这些正文模式函数只适用于IBM PC以及与BIOS兼容的系统。

参 阅 movetext
例 子

```
char buf[20 * 10 * 2];
gettext(0, 0, 20, 10, buf);    /* save rectangle */
/* ... */
puttext(0, 0, buf);            /* restore screen */
```

(32) gettextinfo——取正文模式显示信息

用 法 #include <conio.h>

void gettextinfo(struct text_info* infoec);

说 明 gettextinfo用当前正文显示模式信息填入infoec指向text_info结构。
CONIO.H中定义的text_info结构如下:

```
struct text_info
{ unsigned char winleft;      /* 左窗口坐标 */
  unsigned char wintop;      /* 顶窗口坐标 */
  unsigned char winright;    /* 右窗口坐标 */
  unsigned char winbottom;   /* 底窗口坐标 */
  unsigned char attribute;   /* 正文属性 */
  unsigned char normattr;    /* 正常属性 */
  unsigned char curmode;     /* BW40, BW80, C40, or C80 */
  unsigned char screenheight; /* 由底向上 */
  unsigned char screenwidth; /* 从左到右 */
  unsigned char curx;        /* 当前窗口X坐标 */
  unsigned char cury;        /* 当前窗口Y坐标 */
}
```

返回值 无。结果返回在由infoec指向的结构中。

可移植性 该函数只适用于IBM PC及其兼容机。

参 阅 textattr, textbackground, textcolor, textmode, wherex,
wherey, window

例 子

```
#include <conio.h>
struct text_info initial_info;
main()
{
  gettextinfo(&initial_info);
  /* ... */
  /* 将正文模式恢复成初始值 */
  textmode(initial_info, curmode);
}
```

(33) gettextsettings——返回关于当前正文设置的信息

用 法 #include <graphics.h>

void far gettextsettings(struct textsettingstype
far* textinfo);

相关函数

用 法 void far setttextjustify(int horiz, int vert);
void far setttextstyle (int font, int direction, int charsize);

说 明

gettextsettings用关于当前正文字体、方向、大小和对齐方式(由settextstyle和settextjustify设置)的信息填入textinfo指向的textsettingstype结构。

gettextsettings使用的textsettingstype结构在GRAPHICS.H中如下定义:

```
struct textsettingstype {  
    int font;  
    int direction;  
    int charsize,  
    int horiz;  
    int vert;  
}
```

调用settextjustify后的正文输出将按说明在CP处进行垂直和水平对齐、缺省的对齐设置为LEFTTEXT(水平的)和TOPTEXT(垂直的)。GRAPHICS.H中定义枚举text_just提供传送给settextjustify的horiz(水平)和vert(垂直)方式名(见表B.6)。

表B.6

名	值	说 明
LEFTTEXT	0	水平
CENTERTEXT	1	水平和垂直
RIGHTTEXT	2	水平
BOTTOMTEXT	0	垂直
TOPTEXT	2	垂直

如果horiz等于LEFTTEXT且direction=HORIZ-DIR,则CP的X分量(CPX)将在outtext(string)调用后向前推进textwidth(string)。

settextstyle设置正文字形,正文显示的方向和字符的大小。对settextstyle的调用将影响outtext和outtextxy的所有正文输出。

传送给settextstyle的参数font, direction和charsize描述如下:

font: 可以有一种8×8的位图字形和几种“笔划”的字形。缺省为8×8的位图字形,在GRAPHICS.H中定义的枚举font-names提供了这些不同的字形设置的名(见表B.7)。缺省的位图字形建立在图形系统内。笔划字形存储在*CHR磁盘文件中,且每次只有一个文件被保留在内存中,因此当用户选择一种笔划字形时(与上次选择的笔划字形不同),相应的*CHR文件需从磁盘装入。当几种笔划字形被使用时,为避免上述装入,用户可以将字形文件连接到程序中,可先用BGIOBI实用程序将它们转换成目标文件,再用registerbgifont进行注册,然后进行连接,请参考附录C。

表B.7

名	值	说 明
DEFAULT_FONT	0	8×8 位图字形
TRIPLEX_FONT	1	三倍笔划字形
SMALL_FONT	2	小号笔划字形
SANSERIF_FONT	3	无衬线笔划字形
GOTHIC_FONT	4	黑体笔划字形

direction: 所支持的字形方向是水平正文从左向右和垂直正文(顺时针旋转90°),缺省方向为HORIZ-DIR。

名	值	描 述
HORIZ-DIR	0	从左向右
VERT-DIR	1	自底向上

charsize: 每个字符的大小可用charsize因子放大。如果charsize非零, 它可以影响位图或笔划字符, 如果charsize为0, 仅影响笔划字符。

• 如果charsize=1, outtext和outtextxy将在屏幕上的一个8×8像素矩形区中显示8×8位图字形字符。

• 如果charsize=2, 这些输出函数将在一个16×16像素矩形区中显示8×8位图字形字符, 以次类推(最大到普通大小的16倍)。

• 当charsize=0时, 输出函数outtext和outtextxy使用setusercharsize给出的用户定义字符大小或默认字符放大因子(4)放大笔划字形正文。

textheight和textwidth常用于决定正文的实际尺寸。

返回值 无。由于笔划字符可被存入磁盘, 当试图装入它们时可能出错, 如发生错误, graphresult将返回下列值之一

- 8 未找到字形文件
- 9 没有足够的存储区域装入选择的字形
- 11 (一般错误)
- 12 图形I/O错误
- 13 非法字形文件
- 14 非法字形号

如果传送给settextjustify的是非法的输入, graphresult将回送-11值, 且当前正文对齐方式不改变。

可移植性 Turbo Pascal 4.0中有类似的例行程序。

参 阅 graphresult, outtext, registerbgifont, textheight

例 子

```
#include <graphics.h>
main()
{
    int graphdriver = DETECT, graphmode; /* 需自动检测 */
    struct textsettingstype oldtext;
    initgraph(&graphdriver, &graphmode, ""); /* 初始化图形 */

    gettextsettings(&oldtext); /* 取当前设置 */
    /* switch to horizontal, upper-left-justified */
    /* gothic font, scaled by a factor of 5 */
    settextstyle(GOTHIC_FONT, HORIZ_DIR, 5);
    outtext("Gothic Text");
    /* 恢复原先的设置 */
    settextjustify(oldtext.horiz, oldtext.vert);
    settextstyle(oldtext.font, oldtext.direction, oldtext.charsize);

    colsegraph();
}
```

(34) getviewsettings——返回关于当前视区的信息

用 法 #include<graphics.h>

void far getviewsettings(struct viewporttype far*viewport);

相关函数

用 法 void far setviewport(int left, int top, int right,
int bottom, int clipflag);

说 明 getviewsettings用有关当前视区的信息填入viewprt指向的viewporttype结构。

setviewport为图形输出建立一个新的视区, 视区的边角用绝对屏幕坐标(left, top)和(right, bottom)给出, 当前位置(CP)被移向视区(0, 0), 参数clipflag决定绘图在当前视区边界是否裁剪(截断), 当用户程序调用setviewport时, 如果clipflag为非零值, 则所有的图形在当前视区边缘被剪切

掉。

getviewport使用的viewporttype结构在GRAPHICS.H中如下定义:

```
struct viewporttype {
    int left, top, right, bottom;
    int clipflag;
};
```

注: initgraph和setgraphmode将视区置为全图形屏幕。

返回值 无。如果传送给setviewport的是非法的输入, graphresult将返回-11值, 且当前视区设置不改变。

可移植性 Turbo Pascal 4.0中有相应的函数。

参阅 clearviewphmode

例子

```
struct viewporttype view;
getviewsettings(&view);          /* 取当前设置 */
if (!view.clip)                  /* 如果裁剪开关未打开, 则打开它 */
    setviewport(view.left, view.top,
                view.right, view.bottom, 1);
```

(35) getx——返回当前位置的x坐标

用法 #include<graphics.h>
int far getx(void);

相关函数

用法 int far gety(void);

说明 getx返回当前位置的x坐标, gety返回当前位置的y坐标。

返回值 getx返回CP的x坐标; gety返回CP的y坐标(坐标值相对于视区)。

可移植性 在Turbo Pascal 4.0中有相应的函数。

参阅 getviewsettings, initgraph, moveto

例子

```
int oldx, oldy;
/* save current position */
oldx = getx();
oldy = gety();
circle(100, 100, 2); /* draw a blob at [100, 100] */
moveto(99, 100);
linere1(2, 0);
moveto(oldx, oldy); /* back to the old position */
```

(36) gety——返回当前位置的y坐标

用法 #include<graphics.h>
int far gety(void);

说明 见getx

(37) gotoxy——在正文窗口中定位光标

用法 void gotoxy(int x, int y);

说明 gotoxy移动光标到当前正文窗口中的给定位置。如果坐标无效, 对gotoxy的调用不起作用。

例如, 当(35, 25)为窗口的右下角位置时, 对gotoxy(40, 30)的调用就是这种情况。wherex和wherey两个函数将分别返回光标的当前x和y的位置。

可移植性 这个函数只对IBM PC及兼容机有效, 在Turbo Pascal 4.0中有相应的函数。

参 阅 wherex, wherey, window

例 子 gotoxy(10, 20) /*将光标定位于第10列, 20行*/

(38) graphdefaults——将所有图形设置复位为它们的缺省值

用 法 #include <graphics.h>
void far graphdefaults(void);

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。
- 将当前位置移至(0, 0)。
- 置缺省调色板颜色和绘图颜色。
- 置缺省填充类型和模式。
- 置缺省正文字体和对齐方式。

可移植性 在Turbo Pascal 4.0中有相应的函数。

参 阅 initgraph, getgraphmode

(39) grapherrormsg——返回一个错误信息串

用 法 #include <graphics.h>
char far*far grapherrormsg(int errorcode);

说 明 见graphresult

(40) graphfreemem——释放用户可修改的图形存储器

用 法 #include <graphics.h>
void far _graphfreemem(void far*ptr, unsigned size);

说 明 见 _graphgetmem

(41) _graphgetmem——分配用户可修改的图形存储器

用 法 #include <graphics.h>
void far*far _graphgetmem(unsigned size);

相关函数

用 法 void far _graphfreemem(void far*ptr, unsigned size);

说 明 图形库调用_graphgetmem为内部缓存区、图形驱动器和字符集分配存储区。可以选择通过定义用户自己的_graphgetmem版本来控制图形库的存储区管理(须像用法中一样显式说明)。这个例行程序的默认版本只调用malloc。

图形库调用_graphfreemem释放先前通过_graphgetmem分配的存储区, 用户可以选择通过定义用户自己的_graphfreemem版本控制该图形库存储区管理(须像用法中一样显式说明)。这个例行程序的默认版本只调用free。

可移植性 在Turbo Pascal 4.0中有类似的函数。

例 子 /*用户自定义的图形管理例行程序的例子*/

```

#include <graphics.h>
#include <stdio.h>
#include <conio.h>
#include <process.h>
#include <alloc.h>
main()
{
    int errorcode;
    int graphdriver;
    int graphmode;
    graphdriver = DETECT;
    initgraph(&graphdriver, &graphmode, "c:\\");

    errorcode = graphresult();
    if (errorcode != grOk)
    {
        printf("graphics error,%s\n", grapherrormsg(errorcode));
        exit(1);
    };
    settextstyle(GOTHIC_FONT, HORIZ_DIR, 4);
    outtextxy(100, 100, "BGI TEST");

    closegraph();
}

void far * far_graphgetmem(unsigned size)
{
    printf("_graphgetmem called [size=%d]--hit any key", size);
    getch(); printf("\n");
    return(farmalloc(size));    /* use "far" heap */
}

void far_graphfreemem(void far *ptr, unsigned size)
{
    printf("_graphfreemem called [size=%d]--hit any key", size);
    getch(); printf("\n");    /* "size" not used */
    farfree(ptr);
}

```

(42) **graphresult**——返回上一次不成功的图形操作错误代码

用 法 `#include <graphics.h>`
`int far graphresult(void);`

相关函数

用 法 `char far*far grapherrormsg(int errorcode);`

说 明 **graphresult**返回上一次出错图形操作的错误代码。

grapherrormsg 返回一个指向与**errorcode**联系的字符串的指针，而**errorcode**为**graphresult**返回的错误代码。这样很容易为程序显示一个描述错误的信息，如用“Device driver not found (CGA, BGI)”代替“error code 3”，这样使程序易读。表B.7列出由**graphresult**返回的错误代码，与错误代码相联系的**graphics-errors**类型常量和相应的错误信息。

表B.7

错误 代码	graphics_errors 常 量	相 应 的 错误信息串
0	grOk	NO error
-1	grNOInitGraph	(BGI) graphics not installed (use initgraph)
-2	grNotDetected	Graphics hardware not detected
-3	grFileNotFound	Device driver file not found
-4	grInvalidDriver	Invalid device driver file
-5	grNoLoadMem	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

注意 **graphresult**被调用后内部错误返回代码复位为0。因此，应将**graphresult**的值存入一个临时变量，再测试它。

返 回 值 **graphresult**将返回当前图形错误号，即-15到0间的一个整数，**grapherrormsg**返回一个指向与**graphresult**返回值相联系的串的指针。

可移植性 在 Turbo Pascal 4.0中存在类似的例行程序。

参 阅 **initgraph**

(43) **highvideo**——选择高亮度的正文字符

用 法 `void highvideo(void);`

相关函数

用 法 `void lowvideo(void);`
`void normvideo(void);`

说 明 **highvideo**通过设置当前选择的前景颜色的高亮度位来选择高亮度的字符。

normvideo 在程序启动后，通过将正文属性(前景和背景)返回它所含的值来选择标准字符。

lowvideo 通过清除当前已选择的前景颜色的高亮度位来选择低亮度字符。

这些函数不影响当前屏幕上的任何字符，当这些函数被调用后，它们只影响那些用直接控制台输出函数(例如**cprintf**)显示的字符。

可移植性 这些函数只作用于IBM PC 及其兼容机，在Turbo Pascal中也有相应的函数。

参 阅 cprintf, cputs, gettextinfo, putch, textattr

(44) **imagesize**——返回存储一个位图象所需的字节数

用 法 `#include <graphics.h>`
`unsigned far imagesize(int left, int top, int right, int bottom);`
说 明 参见getimage

(45) **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);`

相关函数

用 法 `void far detectgraph(int far*graphdriver, int far*graphmode);`
`far void closegraph(void);`

说 明 **initgraph** 初始化图形系统，它从磁盘装入一个图形驱动器(或验证一个已注册的驱动器)且把系统置成图形模式。

detectgraph 检查系统的图形适配器和为适配器选择提供最高分辨率的模式，如果没有检测到图形硬件，则***graphdriver**参数置成-2，且**graphresult**也将返回-2。

注意：直接调用**detectgraph**的主要原因是覆盖由**detectgraph**提供给**initgraph**的图模形式。

closegraph释放由图形系统分配的所有存储器，然后恢复屏幕为调用**initgraph**之前的模式。(图形系统通过调用**graphfreemen**释放存储器，例如驱动器、字体和内部缓冲区所占的存储器。)

为了驱动图形系统，首先要调用**initgraph**函数，**initgraph**装入图形驱动器并把系统置成图形模式。用户可以告诉**initgraph**使用一个特殊的图形驱动器和模式，或者在运行时刻自动检查与之相联的显示适配器以及选择与之相应的驱动器。如果用户告诉**initgraph**自动检查，则它便调用**detectgraph**选择一个图形驱动器和模式。**initgraph**也可以把所有图形设置复位为它们的缺省值(当前位置，调色板，颜色，视区等)并复位**graphresult**为0。一般地，**initgraph**通过为驱动器分配存储器(通过**graphgetmem**)来装一个图形驱动器，然后从磁盘上装入适当的.BGI文件。作为动态装入机制的一个选择，可将一个图形驱动器的文件(或几个)直接连接到可执行程序文件中(参见附录C.6)。

pathtodriver说明目录路径，**initgraph**通过该目录查找图形驱动器。**initgraph**首先在**pathtodriver**指定的路径中查找，然后(若不在那里)在当前目录中查找。相应地，如果**pathtodriver**为空，则驱动器文件(*.BGI)一定存在于当前目录中。**settextstyle**搜索笔划字符字形(*.CHR)文件也是这个路径。

***graphdriver**是一个说明所使用图形驱动器的整型数。用户可以使用**graphics_drivers**枚举类型中的常量对它赋值，这些常量定义在GRAPHICS.H中，如下表所列(表B.8)。

表B.8

graphics_driver		graphics_driver	
常 量	数 值	常 量	数 值
DETECT	0 (需自动检测)		
CGA	1	RESERVED	6
MCGA	2	HERCMONO	7
EGA	3	ATT400	8
EGA64	4	VGA	9
EGAMONO	5	PC3270	10

graphmode 是一个说明初始图形模式的(除非***graphdriver**=DETECT，在这种情况下，**graphmode**被置为所检查驱动器有效的最高分辨率)。用户可以使用**graphics_mode**枚举类型的常量给***graphmode**赋值，这些常量定义在GRAPHICS.H中，现列表如下(表B.9)。

表B.9

图形驱动器	图型模式	值	行×列	调色板	页
CGA	CGAC0	0	320×200	C0	1
	CGAC1	1	320×200	C1	1
	CGAC2	2	320×200	C2	1
	CGAC3	3	320×200	C3	1
	CGAHI	4	640×200	2色	1
MCGA	MCGAC0	0	320×200	C0	1
	MCGAC1	1	320×200	C1	1
	MCGAC2	2	320×200	C2	1
	MCGAC3	3	320×200	C3	1
	MCGAMED	4	640×200	2色	1
	MCGAHI	5	640×480	2色	1
	EGALO	0	640×200	16色	4
EGA	EGAHI	1	640×350	16色	2
EGA64	EGA64LO	0	640×200	16色	1
	EGA64HI	1	640×350	4色	1
EGAMONO	EGAMONOH1	3	640×350	2色	1**
	EGAMONOH1	3	640×350	2色	2*
HERC	HERCMONOH1	0	720×348	2色	2
ATT400	ATT400C0	0	320×200	C0	1
	ATT400C1	1	320×200	C1	1
	ATT400C2	2	320×200	C2	1
	ATT400C3	3	320×200	C3	1
	ATT400MED	4	640×200	2色	1
	ATT400HI	5	640×400	2色	1
VGA	VGALO	0	640×200	16色	2
	VGAMED	1	640×350	16色	2
	VGAHI	2	640×480	16色	1
PC3270	PC3270HI	0	720×350	2色	1

* * 64K on EGAMONO card * 256K on EGAMONO card

在上表中, 调色板栏列出的C0, C1, C2和C3是指在CGA(及兼容卡)系统上有效的四种预定义的四色调色板。在每个调色板中, 可选择背景项颜色(是#0), 但其他颜色不变。有关这些调色板的详细内容在第八章中作了描述, 现总结在表B.10中。

表B.10

调色板 数	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

调用initgraph之后, *graphdriver被置成当前图形驱动器, 而*graphmode被置成当前图形模式
返回值 无。initgraph总要设置内部错误代码。如果成功, 代码置为0; 如果出错, *graphdriver置成-2, -3, -4或-5且graphresult返回相同值, 如下所示:

- 2 不能检测图形卡
- 3 不能查找驱动器文件

- 4 非法驱动器
- 5 无足够内存加载驱动器

可移植性 在Turbo Pascal 4.0中有相应的函数。

参 阅 getgraphmode, _graphgetmem,
registerbgidriver, restorecrtmode, setgraphbufsize

例 子

```
#include <graphics.h>
#include <stdio.h>
#include <conio.h>
#include <process.h>
main()
{
    int g_driver, g_mode, g_error;
    detectgraph(&g_driver, &g_mode, "");

    if (g_driver < 0)
    { printf("No graphics hardware detected.\n");
      exit(1);
    }

    printf("Detected graphics driver %d, mode %d\n",
           g_driver, g_mode);
    getch();

    if (g_mode==EGAH1) g_mode=EGALO /* override mode if EGA detected */
    initgraph(&g_driver, &g_mode);
    g_error = graphresult();

    if (g_error < 0)
    {
        printf("initgraph error,%s.\n", grapherrormsg(g_error));
        exit(1);
    }

    bar(0, 0, getmaxx()/2, getmaxy());
    getch();

    closegraph();
}
```

(46) insline——在正文窗口插入空行

用 法 void insline(void);

说 明 利用当前正文背景颜色，在正文窗口的光标位置插入一空行，空行下面的所有行下移一行且底行下卷到窗口的底部。

可移植性 这个函数只适用于IBM PC及其兼容机，在Turbo Pascal中存在相应的函数。

参 阅 clrscr, delline, window

(47) line——在两个指定点之间画一条线

用 法 #include <graphcis.h>

void far line (int x0, int y0, int x1, int y1);

相关函数

用 法 void far lineto(int x, int y);

void far linerel(int dx, int dy);

说明 这些画线函数用当前颜色, 当前画线类型和宽度, 画一条线。

line 在两指定点(x_0, y_0)和(x_1, y_1)之间画一条线, 不修改当前位置(CP)。

linto 从CP到(x, y)画一条线, 然后移动CP到(x, y)。

linere1 画从CP到某一点的一条线, 但该点是相对CP的点(dx, dy), CP增大(dx, dy)。

可移植性 在Turbo Pascal 4.0中有类似的函数。

参阅 getcolor, getlinesettings

(48) **linere1**——从当前位置(CP)到与CP有一个相对距离的点画一条线

用法 `#include <graphics.h>`
`void far linere1(int dx, int dy);`

说明 参见line

(49) **lineto**——从CP到(x, y)之间画一条线

用法 `#include <graphics.h>`
`void far lineto(int x, int y);`

说明 参见line

(50) **lowvideo**——选择低亮度字符

用法 `void lowvideo(void);`

说明 参见highvideo

(51) **moverel**——把当前位置(CP)移动一相对距离

用法 `#include <graphics.h>`
`void far moverel(int dx, int dy);`

说明 参见moveto

(52) **movetext**——把屏幕正文从一个矩形区复制到另一个矩形区

用法 `int movetext(int left, int top, int right,`
`int bottom, int newleft, int newtop);`

说明 movetext把由left, top, right和bottom所定义屏幕矩形区的内容复制到一个新的具有同样尺寸的矩形区。新矩形区的左上角位置由(newleft, newtop)确定, 所有坐标均是绝对屏幕坐标。

返回值 如果操作成功, movetext返回1, 如果操作失败(例如, 所给坐标不在当前屏幕模式范围之内), movetext返回0。

可移植性 这些正文模式函数可用于IBM PC机及和BIOS兼容的系统。

参阅 gettext

例子

```
/*拷贝旧矩形区内容到新矩形区, 旧矩形区左上角和右下角分别是(5, 15)和(20, 25),  
新矩形区左上角为(10, 05)*/  
movetext(5, 15, 20, 25, 10, 20);
```

(55) **moveto**——移动CP到(x, y)处

用法 `#include <graphics.h>`
`void far moveto(int x, int y);`

相关函数

用法 `void far moverel(int dx, int dy);`

说明 这些“移动当前位置”函数将CP在屏幕上移动到另一个位置。

moveto移动当前位置(CP)到视区位置(x, y)。

moverel将当前位置(CP)在x方向移动dx像素, 在y方向上移动dy像素。

可移植性 在Turbo Pascal 4.0中有类似函数。

(54) normvideo——选择标准亮度字符

用 法 void normvideo(void);

说 明 参见highvideo

(55) outtext——在视区显示一字符串

用 法 #include<graphics.h>
void far outtext(char far*textstring);

相关函数

用 法 void far outtextxy(int x, int y, char far*textstring);

说 明 这些函数使用当前对齐方式和当前字形、方向和大小，在视区显示一正文字符串。

outtext在CP处输出textstring。如果水平正文对齐方式是LEFT.TEXT且正文方向是HORIZ.DIR，那么CP的X坐标增量textwidth(textstring)，否则CP保持不变。

outtextxy 在给定位置(x, y)处输出textstring，在使用数种字形时，为保持代码的兼容性，要求使用textwidth和textheight函数决定字符串的尺寸。

可移植性 Turbo Pascal 4.0中有类似的函数。

参 阅 gettextsettings, textheight

(56) outtextxy——发送一个字符串到指定位置

用 法 #include<graphics.h>
void far outtextxy(int x, int y, char far*textstring);

说 明 参见outtext

(57) pieslice——画扇形图并充填

用 法 #include<graphics.h>
void far pieslice(int x, int y, int stangle,
int endangle, int radius);

说 明 参见arc

(58) putimage——在屏幕上显示一个位图象

用 法 #include<graphics.h>
void far putimage(int x, int y, void far*bitmap, int op);

说 明 参见getimage

(59) putpixel——在指定点处画一个象素

用 法 #include<graphics.h>
void far putpixel(int x, int y, int pixel color);

说 明 参见getpixel

(60) puttext——从内存复制正文到屏幕

用 法 int puttext(int left, int top, int right,
int bottom, void*source);

说 明 参见gettext

(61) rectangle——画矩形

用 法 #include<graphics.h>
void far rectangle(int left, int top, int right, int bottom);

说 明 rectangle用当前线类型、宽度和绘图颜色画一个矩形。(left, top)是矩形的左上角，而

(right, bottom)是它的右下角。

可移植性 在Turbo Pascal4.0中存在类似的函数。

参 阅 bar, getlinesettings, getcolor

例 子

```
int i;  
for(i=0; i<10; i++)  
    rectangle(20-2*i, 20-2*i, 10*(i+2), 10*(i+2));
```

(62) registerbgidriver——注册已链接的图形驱动器代码

用 法 #include <graphics.h>
int registerbgidriver(void(*driver)(void));

相关函数

用 法 int registerbgifont(void(*font)(void));

说 明 调用registerbgidriver通知图形系统存在一个已链接驱动器,同样地,调用registerbgifont标志一个已链接的笔划字符字形文件。这些例行程序为指定驱动器或字形检查已链接代码;如果代码有效,就把它登记到内部表中。通过在对registerbgidriver或registerbgifont的调用中使用已链接文件的名,用户也可以使用该公共名通知编译程序和连接程序连接目标文件。

返回值 如果指定的驱动器或字形非法,则该函数均返回负的图形错误代码。否则,registerbgidriver返回一个内部驱动器号,而registerbgifont返回已注册字形的字形号。

可移植性 Turbo Pascal4.0中有相似的函数。

参 阅 initgraph, gettextsettings

例 子

```
/*注册EGAVGA驱动器*/  
if(registerbgidriver(EGAVGA_driver)<0) exit( );  
/*注册黑体字形*/  
if(registerbgifont(gothic_font)!=GOTHIC_FONT) exit(1);
```

(63) registerbgifont——注册已连接笔划字形代码

用 法 #include<graphics.h>
int registerbgifont(void(*font)(void));

说 明 参见registerbgidriver

(64) restorecrtmode——恢复屏幕模式为其initgraph设置值

用 法 #include<graphics.h>
void far restorecrtmode(void);

说 明 restorecrtmode恢复由initgraph检测的初始视屏模式。该函数可和setgraphmode一起来转换正文和图形模式。

可移植性 Turbo Pascal4.0中有相应的函数。

参 阅 initgraph, setgraphmode

(65) setactivepage——为图形输出设置活动页

用 法 #include<graphics.h>
void far setactivepage(int pagenum);

相关函数

用 法 void far setvisualpage(int pagenum);

说 明 setactivepage使得pagenum为活动图形页。所有后续的图形输出将针对pagenum图形页。setvisualpage使得pagenum为可见的图形页。活动图形页可出现也可不出现在屏幕上,这取决于系统有多少有效图形页,只有EGA, VGA和Hercules图形卡支持多页。有

了多图形页，程序可直接将图形输出到屏幕后的页，然后通过调用setvisualpage改变为可见页，可快速显示屏幕后的图形，该技术对动画特别有用。

可移植性 Turbo Pascal4.0中有相似的例行程序。

例子

```
cleardevice();
setvisualpage(0); /*make page 0(=blank) visible*/
setactivepage(1); /*will use page 1 for output*/
bar(50,50,150,150); /*draw a bar in page 1 */
setvisualpage(1); /*show page 1 (with bar)*/
```

(66) setallpalette——改变所有调色板颜色为指定色

用法 #include<graphics.h>
void far setallpalette(struct paletteType far *palette);
说明 参见getpalette

(67) setbkcolor——利用调色板设置当前背景颜色

用法 #include<graphics.h>
void far setbkcolor(int color);
说明 参见getbkcolor

(68) setcolor——利用调色板设置当前绘图颜色

用法 #include<graphics.h>
void far setcolor(int color);
说明 参见getbkcolor

(69) setfillpattern——选择用户定义的着色模式

用法 #include<graphics.h>
void far setfillpattern(char far *upattern, int color);
说明 参见getfillpattern

(70) setfillstyle——设置着色模式和颜色

用法 #include<graphics.h>
void far setfillstyle (int pattern,int color);
说明 参见getfillsettings

(71) setgraphbufsize——改变内部图形缓存区大小

用法 #include<graphics.h>
unsigned far setgraphbufsize(unsigned bufsize);

说明 一些图形例行程序(例如floodfill)使用一个存储器缓存区,该区在调用initgraph时分配,在调用closegraph时释放,由-graphgetmem所分配的该缓存区的标准大小是4096字节。可以使该缓冲区更小(为节省存储空间)或更大(例如,如果调用floodfill产生错误-7:越出存储器范围)。setgraphbufsize通知initgraph在调用-graphgetmem时为该内部图形缓存区分配了多少内存。

在调用initgraph之前必须先调用setgraphbufsize。

返回值 setgraphbufsize返回内部缓存的原大小。

可移植性 Turbo Pascal4.0中有相似的函数。

参阅 closegraph, initgraph

例子

int cbsize;

```
cbsize = setgraphbufsize(1000);    /* get current size */  
setgraphbufsize(cbsize);          /* restore size */  
printf(" The graphics buffer is currently %d bytes.", cbsize);
```

(72) **setgraphmode**——把系统设置成图形模式并清屏

用 法 **#include<graphics.h>**
void far setgraphmode(int mode);
说 明 参见**getgraphmode**

(73) **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);
说 明 参见**getlinesettings**

(74) **setpalette**——改变一个调色板颜色

用 法 **#include<graphics.h>**
void far setpalette(int index, int actual_color);
说 明 参见**getpalette**

(75) **settextjustify**——设置正文对齐方式

用 法 **#include<graphics.h>**
void far settextjustify(int horiz, int vert);
说 明 参见**gettextsettings**

(76) **settextstyle**——设置当前正文特征

用 法 **#include<graphics.h>**
void far settextstyle(int font, int direction, int charsize);
说 明 参见**gettextsettings**

(77) **setusercharsize**——为笔划字体用户自定义字符放大因子

用 法 **#include<graphics.h>**
void far setusercharsize(int multx, int divx,
int multy, int divy);

说 明 **setusercharsize**可以很好地控制由笔划字体产生的正文的大小，只有当**charsize=0**时，由**setusercharsize**设置的值才是有效的，就像通过先调用**settextstyle**所设置一样。

使用**setusercharsize**，用户可指定宽度和高度的比例因子，缺省宽度比例是**multx:divx**，而缺省高度比例是**multy:divy**。例如，生成一正文，使其宽度二倍于缺省值，高度比缺省值高50%，可置，

```
multx = 2; divx = 1;  
multy = 3; divy = 2;
```

可移植性 Turbo Pascal 4.0中有类似的例行程序。

参 阅 **gettextsettings**

例 子

```

#include <graphics.h>

main()
{
    int graphdriver = DETECT, graphmode; /* will request autodetection */
    char *title = "TEXT in a BOX";

    initgraph(&graphdriver, &graphmode, ""); /* initialize graphics */

    /* draw a rectangle and fit a text string inside */
    settextjustify(CENTER_TEXT, CENTER_TEXT);
    setusercharsize(1, 1, 1, 1);
    settextstyle(TRIPLEX_FONT, HORIZ_DIR, USER_CHAR_SIZE);
    setusercharsize(200, textwidth(title), 100, textheight(title));
    settextstyle(TRIPLEX_FONT, HORIZ_DIR, USER_CHAR_SIZE);
    rectangle(0, 0, 200, 100);
    outtextxy(100, 50, title);

    closegraph();
}

```

(78) setviewport——为图形输出设置当前视区

用法 #include<graphics.h>
 void far setviewport(int left, int top, int right, int bottom,
 int clipflag);

说明 参见getviewsettings

(79) setvisualpage——设置可见图形页号

用法 #include<graphics.h>
 void far setvisualpage(int pagenum);

说明 参见setactivepage

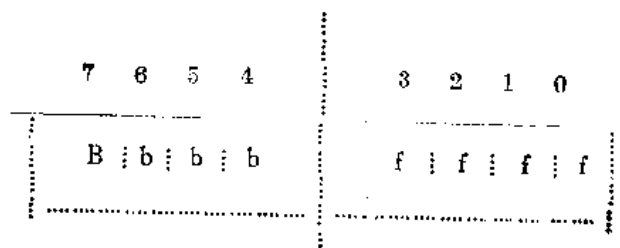
(80) textattr——设置正文属性

用法 void textattr(int attribute);

说明 对textattr的一次调用就可设置前景和背景颜色(一般,利用textcolor和textbackground设置其属性)。

该函数不影响当前在屏幕上的任何字符,它只影响那些在该函数被调用后由直接控制台输出函数(像cprintf)显示的字符。

参数attribute中颜色信息编码如下:



在这个8位的参数attribute中:

ffff 是4位前景颜色(0到15)

bbb 是3位背景颜色(0到7)

B 是闪烁允许位。

如果闪烁允许位为打开状态, 则字符将会闪烁, 这可以通过对属性参数增加常量BLINK实现。如果使用符号颜色常量, 这些常量定义在CONIO.H中用于textattr建立正文属性, 则注意对所选择背景颜色有下列限制:

只能为背景选择前面八种颜色之一。

必须对所选择的背景颜色左移4位, 以得到正确的颜色。

(这些符号常量列表参见textbackground。)

可移植性 该正文模式函数只用于IBM PC机及BIOS兼容系统。

参 阅 textbackground, textcolor

例 子

```
/*select blinking yellow characters on a blue background*/
textattr(YELLOW+(BLUE<<4)+BLINK),
cputs("Hello, world");
```

(81) textbackground——选择新的正文背景颜色

用 法 void textbackground(int color);

相关函数

用 法 void textcolor(int color)

说 明 这些函数为正文字符和正文背景选择新的颜色。

textcolor选择前景字符颜色。

textbackground选择背景正文颜色。

由控制台输出函数后续输出的所有字符的前景(或背景)颜色均由color确定, 这些函数不影响当前屏幕上的任何字符, 只影响在该函数被调用后由直接控制台输出(如cprintf)所显示的字符。

对textbackground, color是从0到7的一个整型数, 而对textcolor是则一个从0到15的整型数, 可以使用定义在CONIO.H中的符号常量来确定颜色, 如果使用这些常量, 则必须使用#include<conio.h>。

表B.11列出所允许使用的颜色(作为符号常量)、它们的数值以及是在前景和背景下有效还是只在前景下有效。

表B.11

符号常量	数 值	前景或背景
BLACK	0	前、背景
BLUE	1	前、背景
GREEN	2	前、背景
CYAN	3	前、背景
RED	4	前、背景
MAGENTA	5	前、背景
BROWN	6	前、背景
LIGHTGRAY	7	前、背景
DARKGRAY	8	前景
LIGHTBLUE	9	前景
LIGHTGREEN	10	前景
LIGHTCYAN	11	前景
LIGHTRED	12	前景
LIGHTMAGENTA	13	前景
YELLOW	14	前景
WHITE	15	前景
BLINK	128	前景

对前景颜色加128, 就可使字符闪烁, 预定义常量BLINK便是为此目的而设置的。

例如: `textcolor(CYAN+BLINK);`

注意: 一些显示器不能识别用来产生八种“淡”颜色(8-15)的强度信号, 在这样的显示器上, 淡颜色的显示相同于“深颜色”颜色(0-7)的显示。同样, 不具有彩色显示的系统可将这些数字看成是一种颜色的阴影、特殊模式或特殊属性(例如下线、黑体、斜体等)具体地要根据所用系统中的硬件而确定。

可移植性 这些函数只适用于IBM PC及其兼容机, Turbo Pascal中有相应的函数。

参 阅 `textattr`

例 子

```
textcolor( GREEN );           /*selects green characters*/
textbackground( MAGENTA );    /*on a magenta background*/
```

(82) `textcolor`——在正文模式中选择新的字符颜色

用 法 `#include<conio. h>`
`void textcolor(int color);`

说 明 参见`textbackground`

(83) `textheight`——以图案为单位返回一个字符串的高度

用 法 `#include<graphics. h>`
`int far textheight(char far*textstring);`

相关函数

用 法 `int far textwidth(char far*textstring);`

说 明 `textheight`取当前字形大小和放大因子, 用图案单位确定`textstring`的高度。`textwidth`取字符串长度、当前字形大小和放大因子, 并且以图案作为单位 确定`textstring`的宽度。

这些函数对下列情况有很大用处, 调整两行之间的间距大小, 计算视区的高度和宽度, 确定一个标题使之在图形或方块的适中位置等等。

例如, 使用8×8位图字形和一个为1的放大因子(由`settextstyle`设置), 字符串Turbo C高8个图案宽48图案。

利用`textheight`和`textwidth`计算字符串的高度和宽度是非常重要的, 它代替了人工计算, 通过使用这些函数, 在选择不同字形后, 就不需要修改源代码。

返回值 `textheight`以图案为单位返回正文高度, `textwidth`以图案为单位返回正文宽度。

可移植性 Turbo Pascal 4.0中有相似的例行子程序。

参 阅 `gettextsettings`, `outtext`

(84) `textmode`——设置屏幕为正文模式

用 法 `void textmode(int mode);`

说 明 `textmode`选择一个指定正文模式, 通过使用一个枚举类型`text-modes`(定义在`CONIO.H`中)的字符常量给出正文模式(参数`mode`)。使用这些常量时, 必须使用`#include<conio. h>`。`text-modes`类型常量, 其数字值和指定的模式列于表B.12之中。

表B.12

符号常量	数字值	正文模式
LAST	-1	先前正文模式
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

当`textmode`被调用时, 当前窗口复位为全屏幕, 且当前正文属性复位成正常, 这相同于对`normvideo`的调用。对`textmode`说明LAST将导致最近所选的正文模式被重新选择。该特征只在使用图形模式后起

返回到正文模式时使用。

可移植性 该函数只适用于IBM PC及其兼容机, Turbo Pascal中有类似函数。

参 阅 gettextinfo

(85) **textwidth**——以图案为单位返回字符串宽度

用 法 #include<graphics.h>
int far textwidth(char far*textstring);
说 明 参见textheight

(86) **wherex**——给出窗口水平光标位置

用 法 int wherex(void);

相关函数

用 法 int wherey(void)

说 明 wherex返回当前光标位置(在当前正文窗口中)的x坐标。wherey返回当前光标位置(在当前正文窗口中)的y坐标。

返 回 值 wherex返回1到80之间的一个整数。

wherey返回1到25之间的一个整数。

可移植性 这些函数只适用于IBM PC及其兼容机, Turbo Pascal中有相应函数。

参 阅 gotoxy

例 子 printf("The cursor is at(%d, %d)\n", wherex(), wherey());

(87) **wherey**——给出窗口内垂直光标位置

用 法 int wherey(void);
说 明 参见wherex

(88) **window**——定义活动正文模式窗口

用 法 void window(int left, int top, int right, int bottom);

说 明 window在屏幕上定义一正文窗口, 如果坐标非法, 则对window的调用不起作用, left和top是窗口的左上角的屏幕坐标。right和bottom是右下角的屏幕坐标。正文窗口的最小尺寸是1行、1列。标准窗口是全屏幕, 其坐标为:

80列模式: 1, 1, 80, 25

40列模式: 1, 1, 40, 25

可移植性 该函数只适用于IBM PC及其兼容机, Turbo Pascal中有相应函数。

参 阅 gettextinfo, textmode

附录C Turbo C 的实用程序及其使用

Turbo C程序包不仅提供了两种当今最快的C语言编译程序，还提供了六个可独立运行的十分有效的实用程序，这些实用程序既能用于Turbo C文件，也可用于其他模块。

这六个程序是：CPP、MAKE、TLINK、TLIB、GREP和BGI0BJ。

本附录介绍了这些实用程序的功能，并用代码和命令行例子描述如何使用它们。

C.1 Turbo C预处理程序CPP

CPP程序用于扩充Turbo C编译程序的功能。它对于C语言程序的一般编译是根本不需要的。CPP的目的在于产生一个对嵌入文件和宏定义进行了扩展的C源程序的列表文件。

通常，当编译程序在宏或嵌入文件内发现了错误时，如果能看到宏或嵌入文件扩展的结果，就可以更多地了解到有关此错误的信息。在许多多趟扫描编译程序中，有专门一趟扫描负责这项工作，并且这次扫描的结果可被查看。

由于Turbo C使用集中式单趟扫描编译程序，CPP充当了其他编译程序中第一趟扫描的功能。此外，CPP还可用作宏预处理程序。

CPP程序的使用方式和运行编译程序TCC一样。CPP从同一个TJURBOC.CFG文件中，读入默认选择项，且命令行选择项也和TCC相同。

CPP忽略与CPP无关的TCC选择项。要想查看由CPP处理的自变量表，在DOS提示符下键入CPP即可。

CPP命令行中的文件名和TCC中一样处理，且允许使用通配符。只是所有文件都作为C源文件处理，对于.OBJ、.LIB和.ASM文件没有特殊处理。

每一个由CPP处理的文件，其输出结果均写到当前目录(或由-n选择项命名的输出目录中)的文件中，其名为源文件名加扩展名.i。

该输出文件是一个正文文件，它包含了源文件的各行和所有嵌入文件。所有预处理指令以及所有不参加编译条件正文行都被删去，正文行均前缀以文件名和源文件的行号或嵌入文件名及行号，正文行中所有宏指令都用它们的扩展正文代替。

由于每行前缀都附有文件名和行号，CPP的输出结果不能被编译。

CPP作为宏预处理程序

CPP的-P选择项要求在每行加以源文件名和行号。若给定-P-，则CPP省略行号信息。这样，CPP能被用作宏预处理程序，产生的.i文件则可以被TC或TCC编译。现举例说明。

下面的程序说明了CPP如何先使用-P和-P-选择项来预处理文件。

源文件：HELLOJOE.C

```
/*这是一个CPP输出的例子*/
#define NAME "Joe Smith"
#define BEGIN {
#define END }
main( )
BEGIN
    printf( "%s\n" , NAME );
END
```

将CPP用作预处理程序的命令行：

CPP hellojoe.c

输出:

```
hellojoe.c2:
hellojoe.c3:
hellojoe.c4:
hellojoe.c6: main( )
hellojoe.c7: {
hellojoe.c8: printf( "%s\n", "Joe Smith" );
hellojoe.c9: }
```

将CPP用作宏预处理程序的命令行:

```
CPP -P- hellojoe.c
```

输出:

```
main( )
{
    printf( "%s\n", "Joe Smith" );
}
```

C.2 独立运行的MAKE程序

Turbo C具有较强的功能和灵活性,可以用来管理由多个嵌入文件、源文件和目标文件构成的大而复杂程序。但要求记住哪些文件需用来产生其他文件。这是因为如果对某一文件作了改动,就必须作必需的重编译和连接。当然,一种解决办法是每当作了改动就对所有模块重编译一次。但当程序越来越长,重编译就要花费很多的时间,那么,该怎么做呢?

答案很简单,使用MAKE。MAKE是一个聪明的管理程序,只要给定适当指令,它能做所有必要的工作使得程序保持新貌。实际上,MAKE能做的远远不止这个,它可以复制副本,从不同子目录中取出文件,甚至当程序使用的数据文件被修改了,它能自动运行程序。多用几次MAKE,就可发现各种新的、不同的有助于程序开发的手段。

MAKE是一个可独立运行的程序,它与Project—Make不同,后者只是集成开发环境的一部分。

本节将介绍如何与TCC和TLINK一起使用MAKE。

C.2.1 一个简单的例子

下面以一个例子来说明MAKE的用途。假设有一显示银河系信息的程序GETSTARS,它读入一个有关银河系的正文文件,进行一些处理,然后产生一个存有结果信息的二进制数据文件。

GETSTARS使用了STARDEF.S中的某些定义和STARLIB.C(在STARLIB.H中说明)的某些子程序。GETSTARS程序本身由三个文件组成:GSPARSE.C, GSCOMP.C, GETSTARS.C。前两个文件GSPARSE和GSCOMP,具有相应的嵌入文件(GSPARSE.H和GSCOMP.H),第三个文件GETSTARS.C包含了程序主体。在三个文件中,只有GSCOMP.C和GETSTARS.C使用了STARLIB程序。

下面是每个.C文件必需的用户嵌入文件(除说明标准运行时刻库子程序的Turbo C嵌入文件)。

.C文件	嵌入文件
STARLIB.C	无
GSPARSE.C	STARDEF.S.H,
GSCOMP.C	STARDEF.S.H, STARLIB.H,
GETSTARS.C	STARDEF.S.H, STARLIB.H, GSPARSE.H, GSCOMP.H

为产生GETSTARS.EXE(假设用中数据模式),需键入下面的命令行:

```
tcc -c -mm -f starlib
tcc -c -mm -f gsparse
```

```
tcc -c -mm -f gscomp
tcc -c -mm -f getstars
tlink lib\com starlib gsparse gscomp getstars, getstars, getstars, lib\emu
lib\mathmlib\cm
```

注意: DOS要求TLINK命令行在一行上, 这里把它写作两行只是因为页宽不够, 一行无法容纳。

查看上面的信息, 可看出一些相互依赖关系:

- (1) GSPARSE, GSCOMP和GETSTARS均依赖于STARDEFS.H。换句话说, 如果对STARDEFS.H作了改动, 那么这三个文件均要重新编译。
- (2) 同样地, 对STARLIB.H的任何改动也将要求GSCOMP和GETSTARS重新编译。
- (3) GSPARSE.H的改动意味着GETSTARS必须重新编译, 对GSCOMP.H也是一样。
- (4) 当然, 对任何源代码文件(STARLIB.C, GSPARSE.C等)的改动意味着该文件要重新编译。
- (5) 最后, 如果进行了重编译, 那么就必须再进行连接。

这要保留相当多的信息。如果改变了STARLIB.H, 重新编译了GETSTARS.C, 但忘记重编译GSCOMP.C, 将发生什么情况呢? 当然可以使用一个.BAT文件进行四次重编译一次连接, 但每改变一次, 就必须再做一遍。让我们看看MAKE是如何简化这些工作的。

C.2.1.1 建立MAKE文件

由上述两组信息(依赖关系及相应的编译连接命令)即可组成一MAKE文件。

例如, 上述两组信息加在一起, 稍微改变就成为:

```
getstars.exe, getstars.obj gscomp.obj gsparse.obj starlib.obj
tlink lib\com starlib gsparse gscomp getstars, getstars, \
getstars, lib\emu lib\mathmlib\cm

getstars.obj, getstars.c stardefs.h starlib.h gscomp.h gsparse.h
tcc -c -mm -f getstars.c

gscomp.obj, gscomp.c stardefs.h starlib.h
tcc -c -mm -f gscomp.c

gsparse.obj, gsparse.c stardefs.h
tcc -c -mm -f gsparse.c

starlib.obj, starlib.c
tcc -c -mm -f starlib.c
```

这仅仅是以前所说的重述, 不过次序有点颠倒, MAKE按如下方式解释这个文件:

- (1) 文件GETSTARS.EXE依赖于4个文件, GETSTARS.OBJ, GSCOMP.OBJ, GSPARSE.OBJ和STARLIB.OBJ。四个.OBJ文件中任何一个改变, GETSTARS.EXE都必须重编译。这可以通过使用给出的TLINK命令实现。
- (2) 文件GETSTARS.OBJ依赖于五个文件, GETSTARS.C, STARDEFS.H, STARLIB.H, GSCOMP.H和GSPARSE.H。如果其中之一改变, GETSTARS.OBJ必须使用给出的TCC命令进行重编译。
- (3) GSCOMP.OBJ文件依赖于三个文件: GSCOMP.C, STARDEFS.H和STARLIB.H, 若其中之一改变, GSCOMP.OBJ必须用给出的TCC命令进行重编译。
- (4) 文件GSPARSE.OBJ依赖于两个文件, GSPARSE.OBJ和STARDEFS.H, 同样, 若其中之一改变, 该文件需用给出的TCC命令重编译。
- (5) 文件STARLIB.OBJ只依赖于一个文件: STARLIB.C, 如果它改变了, 则必须经TCC重新编译。

将上述信息输入一文件(例如MAKEFILE), 即可使用MAKE.EXE。

C.2.1.2 使用MAKE文件

假设按上面描述已生成了一个MAKEFILE, 同时假设各源代码和嵌入文件存在, 即可键入命令:

```
make
```

MAKE查寻MAKEFILE(可以称之为其他名, 后面将要讨论)并且读描述GETSTARS.EXE依赖关系的第一行, 检查GETSTARS.EXE是否有最新的存在。

这要求对GETSTARS.EXE依赖的每个文件(GETSTARS.OBJ, GSCOMP.OBJ, GSPARSE.OBJ和STARLIB.OBJ)作相同的检查。其中各个文件所依赖的其他文件也要被检查。为更新各.OBJ文件, 需多次调用TCC命令, 最后执行TLINK(若必要的话)产生GETSTARS.EXE的最新版本。

如果GETSTARS.EXE和所有.OBJ文件均已存在, MAKE则将每个.OBJ文件的最后一次修改日期、时间和它依赖的文件的日期、时间作比较。如果有一依赖的文件比.OBJ文件新, MAKE知道从上次.OBJ文件生成以来已经有了修改, 则执行TCC命令。

如果MAKE更新了任一.OBJ文件, 则当它将GETSTARS.EXE的日期、时间与它们比较时就知道必须执行TLINK命令以产生GETSTARS.EXE的最新版本。

C.2.1.3 Make使用例

下面举一例子以帮助理解以上描述。假设GETSTARS.EXE和所有的.OBJ文件均存在, 并且GETSTARS.EXE比所有.OBJ文件更新, 同样地每个.OBJ文件比它所依赖的各文件更新。这样, 若打入命令:

```
make
```

则不会有任何事发生, 因为没有任何文件需要更新。

现在, 假设修改STARLIB.C和STARLIB.H, 如改变某些常量的值, 当打入命令

```
make
```

MAKE看到STARLIB.C比STARLIB.OBJ更新。因此它发出命令:

```
tcc -c -mm -f starlib.c
```

然后它看到STARLIB.H比GSCOMP.OBJ更新, 于是它发出命令:

```
tcc -c -mm -f gscomp.c
```

STARLIB.H也比GETSTARS.OBJ更新, 于是下一条命令是:

```
tcc -c -mm -f getstars.c
```

最后, 由于这三条命令, 文件STARLIB.OBJ, GSCOMP.OBJ和GETSTARS.OBJ都比GETSTARS.EXE要新, 因此MAKE发出的最后一条命令是:

```
tlink lib\com starlib gsparse gscomp getstars,  
getstars, getstars, lib\emu lib\mathm lib\cm
```

它将所有的.OBJ文件连接在一起生成GETSTARS.EXE的一个新版本(注意, TLINK命令行实际上必须为一行)。

现已对MAKE有了一些基本的认识: 它是干什么的, 如何生成一个make文件, 以及MAKE如何解释该文件。下面将更详细地介绍之。

C.2.2 MAKE文件的组成及生成

make文件包含了以助MAKE使程序保持“最新”的定义和关系。可以任意生成多个make文件, 并可随意给它们命名。MAKEFILE只是当运行MAKE而又未指定make文件时, MAKE查找的一个默认文件名。

可以使用任何ASCII码正文编辑程序(如Turbo C的内部交互式编辑程序)来生成make文件。所有的规则、定义和指令均需以换行符结束。若一行太长(像在前面例子中的TLINK命令), 可用反斜杠作为行的最后一个字符来继续到下一行。

空白(空格和制表键)用于分隔相邻的标识符(例如依赖文件名), 并且在规则中用于对齐命令。

生成make文件类似于编写程序, 它带有定义、命令和指令。make文件中允许出现下述成分:

- (1) 注解
- (2) 显式规则
- (3) 隐式规则
- (4) 宏定义
- (5) 指令: 文件包含、条件执行、错误检测、取消宏定义。

C.2.2.1 注解

注解以字符#开始，#后该行的其余部分MAKE是忽略的，注解可放在任何位置并且不必在某一特定列开始。

反斜杠\不能把注解继续到下一行，因而每行必须使用一个#。实际上，在含有注解的行中不能使用反斜杠作为继续字符，如果它在#前面，它不再是行的最后一个字符；如果它跟着#，则它是注解本身的一部分。

下面是make文件中注解的几个例子：

```
#makefile for GETSTARS.EXE
#does complete project maintenance
getstars.exe, getstars.obj gscomp.obj gsparse.obj starlib.obj
#can't put a comment at the end of the next line
tlink lib\c0m starlib gsparse gscomp getstars, \
    getstars, getstars, lib\emu lib\mathm lib\cm
#legal comment
#can't put a comment between the next two lines
getstars.obj, getstars.c stardefs.h starlib.h \
    gscomp.h gsparse.h
tcc -c -mm -f getstars.c

#you can't put a comment here
```

C.2.2.2 显式规则

前面的例中已使用了显式规则。显式规则的形式为：

```
target [target...]: [source source...]
[command]
[command]
...
```

这里 target 是要更新的文件，source 是 target 所依赖的文件。command 是任意有效的MS-DOS命令(包括.BAT文件的调用和.COM及.EXE文件的执行)。

显式规则定义一至多个目标名，零个或多个源文件以及一个任选命令表。在显式规则中列出的目标文件名、源文件名，可以含有驱动器号和指定目录，但不能含有通配字符。

这里的语法格式相当重要。target 必须始于行首(第1列)，而每个command 前面至少有一个空白或制表符。如前所述，当源文件表或一给定命令太长，一行无法容纳时，可用反斜杠(\)作为继续符。注意，源文件和命令都是任选的。只含target [target...] 的显式规则也是合法的。

显式规则的含意是：列出的命令能通过使用源文件生成或更新目标文件。当MAKE 遇到一个显式规则，它首先在makefile 中检查各源文件本身是否为目标文件。如果是，首先处理那些规则。

一旦根据其他显式(或隐式)规则生成或更新了所有源文件，MAKE 就检查目标文件是否存在。如果不存在，则按给定次序执行每个命令。如果存在，则将各源文件的最后修改的日期、时间和目标文件的日期、时间比较。若某源文件比目标文件更晚修改，则执行命令表。

在一给定的MAKE 执行中，一个给定的文件名只能在一显式规则的左侧出现一次。

在显式规则中，每个命令以空白开始。MAKE 认为在显式规则后的所有行都是此规则命令表的一部分，直至在第一列开始的行(前面无空白)或文件的结束。空行将被忽略。

特殊情况

没有命令行的显式规则的处理与有命令行的显式规则的处理稍有不同：

- (1) 如果某目标的显式规则中有命令行，则此目标依赖的文件仅为显式规则中列出的那些文件。
- (2) 如果无命令行，目标不仅依赖于显式规则中给出的文件，还依赖于任何与本目标的隐式规则匹配的文件。

参见下节介绍的隐式规则。

例子

下面是显式规则的几个例子，

```
myprog.obj, myprog.c  
tcc -c myprog.c
```

```
prog2.obj, prog2.c include\stdio.h  
tcc -c -k prog2.c
```

```
prog.exe, myprog.c prog2.c include\stdio.h  
tcc -c myprog.c  
tcc -c -k prog2.c  
tlink lib\cos myprog prog2, prog, , lib\cs
```

- (1) 第一个显式规则说明MYPROG.OBJ依赖于MYPROG.C, 并且 MYPROG.OBJ是通过执行给出的TCC命令生成的。
- (2) 第二个规则说明PROG2.OBJ依赖于PROG2.C和 STDIO.H(在 INCLUDE 子目录中), 并且它是由给出的TCC命令生成的。
- (3) 最后一条规则说明 PROG.EXE 依赖于 MYPROG.C, PROG2.C和 STDIO.H, 并且若有一个改变, PROG.EXE 就可由给定的命令串重建。但是这可能产生不必要的工作, 因为即使只有 MYPROG.C 改动了, PROG2.C 仍将重编译。这是由于只要规则的目标过时了, 此规则的所有命令就将被执行。
- (4) 如将规则:

```
prog.exe, myprog.obj prog2.obj  
tlink lib\cos myprog prog2, prog, , lib\cs
```

作为make文件的第一条规则, 后面跟(对MYPROG.OBJ和PROG2.OBJ)给出的规则, 那么只有那些需要重编译的文件才被重编译。

C.2.2.3 隐式规则

MAKE 同样允许定义隐式规则。隐式规则是显式规则的拓广形式, 下面的例子将说明了两种规则之间的关系。考虑前面例子中的显式规则,

```
starlib.obj, starlib.c  
tcc -c -mm -f starlib.c
```

此规则是很普通的, 它遵循一般规则: .OBJ 文件依赖于具有相同名字的.C文件, 并且由执行TCC命令生成。实际上 make 文件可能有若干个(甚至几十个)具有这种格式的显式规则。

将显式规则重定义成隐式规则, 就能避免所有具有相同形式的显式规则。隐式规则形式如下:

```
.c.obj:  
tcc -c -mm -f $<
```

这规则意为“所有以.OBJ 结尾的文件依赖于具有相同名的以.C结尾的文件, 并且该.OBJ 文件 通过使用命令tcc -c -mm -f \$< 生成。其中 \$< 代表带有.C扩展名的文件名”(符号 \$<是一个特殊的宏命令, 将在下一节讨论)。

隐式规则的语法为:

```
source_extension.target_extension:  
{ command }  
{ command }  
...
```

与前面一样, 其中诸命令是任选的且必须缩进书写。source_extension(必须在第一列开始)是源文件的扩展名。即它适用于任何如下格式的文件:

```
fname.source extension
```


类似地, `target-extension` 是目标文件的扩展名, 它适用于文件:

```
fname,target-extension
```

其中 `fname` 对两个文件来说是相同的。换句话说, 此隐式规则代替了所有如下格式的显式规则:

```
fname,target-extension; fname,source-extension
    { command }
    { command }
...
```

其中 `fname` 为任何名。

如果找不到给定目标文件的显式规则, 或者其显式规则无命令, 则使用隐式规则。

此时该目标文件名的扩展名用于确定使用哪个隐式规则。若发现一个文件具有与目标相同的名, 且有提到的源扩展名, 则使用该隐式规则。例如, 假设有一个 `make` 文件 (名为 `MAKEFILE`) 其内容为:

```
.c.obj:
```

```
tcc -c -ms -f $<
```

如果有一个名为 `RATIO.C` 的 C 程序, 需编译成 `RATIO.OBJ`。可以使用命令:

```
make ratio.obj
```

`MAKE` 将认为 `RATIO.OBJ` 为目标。由于没有显式规则能生成 `RATIO.OBJ`, `MAKE` 使用隐式规则并产生命令:

```
tcc -c -ms -f ratio.c
```

当然, 该命令确能通过编译来生成 `RATIO.OBJ`。

当一个显式规则无命令时也使用隐式规则。假设在 `make` 文件开始有如下隐式规则:

```
.c.obj:
```

```
tcc -c -mm -f $<
```

原显式规则可重写成为下面形式:

```
getstars.obj,stardefs.h    starlib.h    gscomp.h    gsparse.h
gscomp.obj,stardefs.h    starlib.h
gsparse.obj,stardefs.h
```

由于没有关于如何生成 `.OBJ` 文件的显式信息, `MAKE` 使用以前定义的隐式规则。由于 `STARLIB.OBJ` 只依赖于 `STARLIB.C`, 该规则不用列出, `MAKE` 将自动地使用之。

多条隐式规则可用相同的目标扩展名, 但在某一时刻, 其中只有一条规则可使用。如果对一给定的目标扩展名存在多个隐式规则, 对每个规则按其出现次序进行检查, 直至所有可用规则全部检查过。

`MAKE` 仅使用第一条能找到带有相应源扩展名的文件的隐式规则。即使那条规则命令失败, 也不再继续检查其他隐式规则。

隐式规则后的所有行均认为是此规则命令表的一部分, 直至以无空白开始的行或文件的结尾。空行将被略去。命令行的语法将在后面说明。

特殊情况

和显式规则不同, `MAKE` 使用隐式规则时, 不知道文件的全名。为此 Turbo C 为 `MAKE` 提供了特殊的宏命令, 它允许在规则中加入文件名 (详见本章宏定义的讨论)。

例子

下面是几个隐式规则的例子:

```
.c.obj:
```

```
tcc c $<
```

```
.asm.obj:
```

```
masm $*/mx,
```

在第一个隐式规则例子中, 目标文件是 `.OBJ` 文件, 它们的源文件是 `.C` 文件。其命令表中仅有一个命令行。

第二个例子指示 `MAKE` 使用带有 `/mx` 选择项的 `MASM` 对一给定文件的 `.ASM` 源文件进行汇编。

显式和隐式规则中的命令表

我们已经讨论了显式和隐式规则，它们均可带有命令表。下面详细讨论这些命令，以及其中可用的选项。

命令表中的命令必须缩进书写，即至少前面有一个空格或制表符，其格式为：

{ 前缀... } 命令体

命令表中每一命令行包含一个任选的前缀表，后跟一个命令体。

前缀

命令中的前缀改变了 MAKE 对这些命令的处理。前缀可以是@或连字符(-)紧跟一数字。

@ 使MAKE 在执行命令前不显示此命令。即使在 MAKE 命令行上未给出-S选项。此显示也要隐蔽。该前缀只适用于它出现的命令。

-num, 改变 MAKE 的出口代码处理。如果提供一个数字(num), 那么当出口代码超过给定数字时, MAKE 中止处理。在下面例子中, 只有出口状态超过4时MAKE才中止处理:

 -4 myprog sample.x

如果无-num前缀, MAKE 检查命令的出口状态, 如果状态非0, MAKE 将停止执行并删除当前目标文件。

若只有连字符无数字, 则 MAKE 不检查出口状态, 不管出口状态是什么, MAKE 都继续进行工作。

命令体

命令体的处理就像它是输入到 COMMAND.COM 的命令行一样, 但不支持重定向和管道操作。MAKE通过调用一个 COMMAND.COM 的拷贝来执行下面的内部命令:

break	ed	chdir	cls	copy
ctty	date	del	dir	erase
md	mkdir	path	prompt	ren
rename	set	time	type	ver
verify	vol			

MAKE 使用MS-DOS搜索算法查找其他命令名:

- (1) 先查寻当前目录, 然后查找路径中的其他子目录。
 - (2) 每个目录中, 先检查带扩展名.COM的文件, 然后是带.EXE 的, 最后是.BAT。
 - (3) 如果找到一个.BAT 文件, 则调用一个 COMMAND.COM 的拷贝来执行这一文件。
- 显然, 若在命令行中提供了扩展名, MAKE 只需找那个扩展名。

例子

下面命令使 COMMAND.COM 执行命令:

 cd c:\include

下面命令将用全搜索算法查找:

 tlink lib\c0s xy, z, z, lib\cs

下面命令将只用.com扩展名查找:

 myprog.com geo.xyz

下面命令将使用提供的显式文件名执行:

 c:\myprogs\fil.exe -r

C.2.2.4 宏

通常某些命令、文件名或选择项在 make 文件中经常重复使用。在本附录开始时的例子中, 所有 TCC命令均使用-mm 开关, 它的意思是在中存储模式下进行编译。同样, TLINK 命令使用COM.OBJ, MATHM.LIB和 CM.LIB。假设要换成大存储器模式, 则需把所有-mm选择项换为-mt, 并对 TLINK 命令中的相应文件重新命名。另外一种方法是定义一个宏。

宏是一个代表某字符串的名字, 宏定义给宏命名并且给出扩展正文。此后, 当 MAKE 遇到这个宏

名字，它就用扩展正文来替代此名。

假设在 `make` 文件开始定义了如下的宏，

```
MDL = m
```

则表示定义了宏 `MDL`，它与字符串 `m` 相同。 `make` 文件可重写成（注：由于打印机的原因，下面出现的 `¥` 应为 `$`），

```
MDL=m
```

```
getstars.exe, getstars.obj gscomp.obj
    gsparse.obj starlib.obj
tlink lib\c¥(MDL) starlib gsparse gscomp
    getstars, ¥getstars, getstars, lib\emu
    lib\math¥(MDL) lib\c¥(MDL)

getstars.obj, getstars.c stardefs.h starlib.h
    gscomp.h gsparse.h
tcc -c -m¥(MDL) getstars.c

gscomp.obj, gscomp.c stardefs.h starlib.h
tcc -c -m¥(MDL) gscomp.c

gsparse.obj, gsparse.c stardefs.h
tcc -c -m¥(MDL) gsparse.c

starlib.obj, starlib.c
tcc -c -m¥(MDL) starlib.c
```

在使用存储模式的每个地方，使用宏调用 `$(MDL)`。当运行 `MAKE` 时，`$(MDL)` 由它的扩展正文 `m` 替代。结果与前面的命令集相同。

这样能带来灵活性。把第一行改为，

```
MDL = l
```

就把所有命令改为使用大存储模式。实际上，如果去掉第一行，在每次运行 `MAKE` 时，可使用 `-D` (`Define`) 选择项来确定使用什么样的存储模式：

```
make -DMDL = l
```

这告诉 `MAKE` 把 `MDL` 作为扩展正文为 `l` 的宏来处理。

宏定义

宏定义的格式为：

```
macro_name = 扩展正文
```

其中 `macro_name` 是宏名，它由字母、数字组成，不能含空格，但在 `macro_name` 与等号间可加空白。扩展正文是任意的包含字母、数字、空白和标点符号的字符串，它以新行结束。

如果 `macro_name` 已定义过，不管是由 `make` 文件中的宏定义，还是由 `MAKE` 命令行中的 `-D` 选择项定义，新定义总代替旧定义。

在宏中，字母的大小写是有意义的，即宏名 `mdl`，`Mdl` 和 `MDL` 被认为是不同的。

宏调用

在 `make` 文件中，宏以如下形式调用：

```
$(宏名)
```

除了三种将要讨论的特殊预定义宏，即使宏名只是一个字符，在调用时括号也不可省。此结构称为宏调用。

当 `MAKE` 遇到一个宏调用，它就用此宏的扩展正文来替代宏名。如果宏未定义，`MAKE` 就用空串来替代它。

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。换句话说，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中，宏调用立即被扩展。如果在!if或!elif指令中被调用的宏当前未定义，则它扩展成0值(FALSE)。

命令中的宏：当命令执行时，命令中的宏调用被扩展。

预定义的宏

MAKE具有若干个特殊的内部定义宏：\$d, \$*, \$<, \$!, \$, 和 &\$。第一个是已定义的测试宏，用于条件指令!if和!elif中，其余的是文件名宏，用于显式和隐式规则中。另外，当前SET环境变量字符串也自动地被当作宏，并且宏--MAKE--定义为1。

决义的测试宏(\$d)：如果给出的宏名已经定义，则已定义的测试宏\$d扩展成1，不然扩展成0。宏的扩展正文的内容无关紧要。该特殊宏只允许在!if和!elif指令中。例如，假设想修改make文件，以便在未确定存储模式时，使用中存储模式，在make文件开始写：

```
! if ! $d(MDL)                #if MDL is not defined
MDL = m                        #define it to m(MEDIUM)
! endif
```

如果用如下命令行调用MAKE：

```
make-DMDL = 1
```

MDL定义为1，但如果仅用MAKE自身调用：

```
make
```

则MDE定义为m，作为“默认”存储模式。

各种文件名宏

各种文件名宏的工作方式大致相同，扩展成不同的文件完全路径名。

基本文件名宏(\$*)

基本文件名宏允许出现在显式规则或隐式规则的命令中。此宏扩展成正被建立的文件名，不带扩展名，形式如下：

文件名是A, \P\TESTFILE.C

\$*扩展成A, \P\TESTFILE

例如：可将已给定的GETSTARS.EXE显式规则改为如下形式：

```
getstars.exe: getstars.obj gscomp.obj gsparse obj starlib.obj
    tlink lib \c0 $(MDL) starlib gsparse gscomp $*, $*, $*, \
    lib \emu lib \math $(MDL) lib\c $(MDL)
```

当此规则中命令执行时，宏指令\$*由目标文件名getstars(不带扩展名)替换，此宏指令对隐式规则十分有用。

例如，TCC的一个隐式规则可能为如下形式(假设MDL宏指令已经或将被定义，并且现在不使用浮点子程序)：

```
.c.obj,
    tcc -c $*
```

全文件名宏(\$<)

全文件名宏(\$<)既用在显式规则的命令中也用在隐式规则的命令中。在显式规则里，\$<扩展成完全目标文件名(包括扩展名)形式如下：

文件名是：A, \P\TESTFILE.C

\$<扩展为: A: \P\TESTFILE.C

例如, 规则

```
starlib.obj: starlib.c
    copy $<\oldobjs
    tcc -c $*
```

在编译 STARLIB.C 前将 STARLIB.OBJ 复制到目录\OLDOBJs中。

在隐式规则中, \$<用文件名加源扩展名替代。例如对前面的隐式规则:

```
.obj.c:
    tcc -c $*.c
```

可重写为:

```
.obj.c:
    tcc -c $<
```

文件名路径宏(\$:)

此宏扩展为路径名(不带文件名), 形式如下:

文件名是: A: \P\TESTFILE.C

\$: 扩展成: A: \P\

文件名与扩展名宏(\$.)

此宏扩展成文件名带扩展名, 形式如下:

文件名是: A: \P\TESTFILE.C

\$.扩展为: TESTFILE.C

纯文件名宏(\$&)

此宏只扩展为文件名, 不带路径和扩展名, 形式如下:

文件名是: A: \P\TESTFILE.C

\$&扩展为: TESTFILE

C. 2.2.5 指令

Turbo C 的MAKE 允许其他 MAKE 版本所不允许的一些东西, 如C语言本身允许的指令。这些指令可用于包括其他make文件, 产生规则和命令条件, 打印出错信息和‘取消’宏。

在 make 文件中的指令用惊叹号(!)作为行的第一个字符。与C不同, C使用字符‘#’。下面是所有 MAKE 指令。

```
! include
! if
! else
! elif
! endif
! error
! undef
```

文件嵌入指令

文件嵌入指令(! include)指定在该指令出现处嵌入一个文件。其形式为:

```
! include "filename"
```

嵌入指令可嵌套任意多重。如果一个嵌入指令企图包含一个在某外层上已嵌入的文件(因此产生嵌套循环), 则内层嵌入指令被拒绝, 产生出错。

假设有一文件MODEL.MAC文件, 包含如下指令:

```
! if! $d(MDL)
MDL=m
! endif
```

通过嵌入指令: include "MODEL.MAC", 可在任何make文件中使用此条件宏定义。

MAKE 遇到 `!include` 指令, 它打开相应文件, 并且读其内容, 就好像它们在 `make` 文件中一样。

条件指令

条件指令(`!if`, `!elif`, `!else`和`!endif`)为程序员提供了构造 `make` 文件的灵活手段。规则和宏可以条件化, 以便命令行中的宏定义(使用 `-D` 选项)能够允许或禁止 `make` 文件的某些部分。

这些指令的格式与 C 预处理程序指令的格式相同:

```
!if expression
    [lines]
!endif
!if expression
    [lines]
!else
    [lines]
!endif
!if expression
    [lines]
!elif expression
    [lines]
!endif
```

注意: `[lines]` 可为下面任何一种:

```
macro_definition
explicit_rule
implicit_rule
include_directive
if_group
error_directive
undef_directive
```

条件指令是成组出现的, 它至少以一个 `!if` 指令开始, 以 `!endif` 指令结束。

(1) 组中可出现一个 `!else` 指令。

(2) `!elif` 指令可出现在 `!if` 和任一 `!else` 指令之间。

(3) 规则、宏和其他指令可在各条件指令间出现任意多次。注意: 带命令的完全规则不能分散在条件指令的两边。

(4) 条件指令组可嵌套任意多层。

在一个源文件中, 所有规则、命令和指令都必须是完整的。

在同一源文件中, 所有 `!if` 指令必须与 `!endif` 匹配。因此不管各包括文件内容是什么, 下面的包括文件是非法的。因为它没有匹配的 `!endif` 指令:

```
!if $(FILE-COUNT) > 5
    some rules
!else
    other rules
<end_of_file>
```

条件指令中允许的表达式

在 `!if` 和 `!elif` 指令中允许的表达式使用一种类似 C 语言的语法, 表达式按一个简单的 32 位带符号整数表达式计算。

数字可作为十进制、八进制或十六进制常数输入。例如, 下面都是在表达式中合法的常数:

```
4536      # 十进制常数
0677      # 八进制常数
0x23aF    # 十六进制常数
```

表达式可使用下列单目运算符

- 取负
- ~ 按位求反
- ! 逻辑非

表达式可使用下列双目运算符:

- | | | | |
|----|-----|----|------|
| + | 加 | & | 按位异或 |
| - | 减 | && | 逻辑与 |
| * | 乘 | | 逻辑或 |
| / | 除 | > | 大于 |
| % | 取余数 | < | 小于 |
| >> | 右移 | >= | 大于等于 |
| << | 左移 | <= | 小于等于 |
| & | 按位与 | = | 相等 |
| ! | 按位或 | != | 不等 |

表达式可使用下列三目运算符:

?:

?: 前的运算分量作为条件。如果此运算分量的值非 0, 则第二运算分量(位于?: 和: 之间)是结果, 若为 0, 结果是第三运算分量的值(位于: 之后)。

在表达式中可用括号把运算分量括起来。若无括号, 双目运算按在 C 语言中给出的优先级处理。

同在 C 语言中一样, 相同优先级运算是从左到右计算。但三目运算(?:)例外, 它是从右到左。

宏可在表达式中使用, 并且识别特殊宏 \$d()。在所有宏被扩展后, 表达式应具有正确的语法。已扩展表达式中的任何一个标识符均作错误处理。

报错指令

报错指令(!error)使得 MAKE 中止执行并打印!error 后的致命错误诊断。其格式为:

!error 任何正文

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(MDL)
# if MDL is not defined
!error MDL not defined
!endif
```

如果到达这点时未定义 MDL, 则 MAKE 将停止并报告如下出错信息:

Fatal makefile 5: Error directive, MDL not defined

取消定义指令

取消定义指令(!undef)使得 MAKE 忘却指定的宏。若此宏指令当前无定义, 该指令不起作用。语法是:

!undef 宏名

C.2.3 使用 MAKE

以上叙述了 make 文件的编写, 现在介绍如何和 make 程序一起使用这些文件。

C.2.3.1 命令行语法

使用 MAKE 最简单的方法是在 MS-DOS 提示符下键入命令:

make

然后 MAKE 寻找 MAKEFILE。若找不到, 它就找 MAKEFILE.MAK。若还找不到, 它就停止, 显示出错信息。

如果想使用一个名字不是 MAKEFILE 或 MAKEFILE.MAK 的文件, 只需给 MAKE 指定一文件选项(-f), 例如:

```
make -fstars.mak
```

调用MAKE的一般格式为:

```
make 选择项 选择项... 目标 目标...
```

其中选择项是一个MAKE选择项, 目标是由显式规则处理的目标文件的名。

下面是语法规则:

(1) make后为一空白, 然后是一系列make选择项。

(2) 每个make选择项与其相邻选择项由一空格隔开, 选择项可按任意次序排列且选择项个数不限(只要一行容纳得下)。

(3) 在make选择项后有一空格, 然后是一系列任选的目标。

(4) 每个目标与其相邻目标也以空格分开。MAKE依次处理目标, 必要的话则进行重编译。

如果在命令行中未包括目标名, MAKE则使用显式规则中的第一个目标文件。若在命令行中有一个或若干个目标, 它们将按需要生成。

下面是MAKE命令行的另外一些例子:

```
make -n -fstars.mak
```

```
make -s
```

```
make -linclude _DMDL=c
```

C. 2.3.2 异常中止MAKE时的注意事项

如果MAKE执行的任何命令由Ctrl-C中止, MAKE亦将停止, 即Ctrl-C同时停止执行当前命令和MAKE。

C. 2.3.3 BUILTINS. MAK文件

使用MAKE时会发现, 有些宏和规则(通常是隐式规则)被一再使用。有三种处理它们的方法:

(1) 可以把它们放到生成的每个文件中。

(2) 把它们全放在一个文件中, 在生成的每个文件中使用!include指令。

(3) 把它们全部放在一个名叫BUILTINS. MAK的文件中。

每次运行MAKE时, 它寻找一个名为BUILTINS. MAK的文件, 若找到, 在处理MAKEFILE(或任一要处理的make文件)前, MAKE先读它。

BUILTINS. MAK文件是为经常使用的宏和规则(常为隐式规则)而建立的。

并不要求BUILTINS. MAK必须存在。若MAKE找到BUILTINS. MAK文件, 它首先解释此文件。若找不到, MAKE就直接去解释MAKEFILE(或任一指定的makefile)。

C. 2.3.4 MAKE如何搜索MAKE文件

MAKE在当前目录或路径中任一目录下搜索BUILTINS. MAK。该文件应与MAKE. EXE文件放在同一目录中。

MAKE总是只在当前目录中寻找make文件。该文件包含了构造当前特定可执行程序规则的规则。这两个文件具有相同的语法规则。

MAKE也在当前目录中搜索!include文件。如使用-I(包括)选择项, 它将在指定目录中搜索。

C. 2.4 TOUCH程序

即使目标文件的源文件未改变, 有时也要重编译或重建一目标文件。这可使用Turbo C的TOUCH程序来做。TOUCH把一个或若干个文件的日期和时间变为当前的日期和时间, 使其比任何依赖于它的文件都‘新’。

为重建一个目标文件, 只需用TOUCH更新该目标所依赖的某一文件。即在MS-DOS提示符下键入,

```
touch 文件名[文件名...]
```

TOUCH将更新此文件的生成日期和时间。

一旦这样做后, 就可以调用MAKE来重建修改后的目标文件(TOUCH可使用通配符*和?)。

C. 2.5 MAKE命令行选择项

我们已经提到过若干个MAKE命令行选择项, 现将所有的选择项列于下表中。

项 择 选	含 义
- Didentifier	将指定标识符定义为只有单个字符的字符串。
- Diden = string	将指定标识符iden定义为等号后面的字符串。
- Idirectory	MAKE在指定的目录中(也在当前目录中)搜索包括文件。
- Uidentifier	取消指定标识符以前的定义。
- s	通常, 当一条命令要执行时, MAKE将其打印, 若有-s选择项, 在执行前不打印命令。
- n	使MAKE打印命令, 但不真正执行它们, 这对调试make文件很有用。
- ffilename	把filename用作MAKE文件。若filename不存在并且未给出扩展名, 试用filename.mak
- ? 或 -h	打印帮助信息。

注意: 字母的大小写是有意义的。-d选择项与-D是不同的。

C.2.6 MAKE错误信息

MAKE诊断信息分为两类:灾难性错误和一般错误。当发生一个灾难性错误时,编译立即停止。必须采用适当措施,然后重新进行编译。一般错误表示在make文件中有某些语义、语法错误,MAKE在完成对make文件的解释后停止工作。

C.2.6.1 灾难性错误

不知道如何生成 × × × × × × × ×

当MAKE在生成序列中遇到一个不存在的文件,并且不存在允许建立该文件的规则时发出此信息。

错误指令: × × × ×

当MAKE处理源文件中的# error指令时给出此信息,指令的正文也将显示出。

非法命令行参数: × × ×

执行MAKE时若使用非法命令行参数,则发生此错误。

无足够存储区

当全部工作存储区用尽时发生此错误。可在具有更大存储容量的机器上试一下,如果内存容量已有640KB,则必须简化源文件。

命令无法执行

当需要执行一条命令但无法执行时发出此信息,这可能是由于找不到命令文件或命令拼错引起的,另一较小可能性是命令存在但被无端破坏了。

make文件无法打开

当前目录不含有MAKEFILE时,发出此错误。

C.2.6.2 一般错误

在包括语句中文件名格式错

包括文件名必须由引号或尖括号括起来。发出此信息,说明文件名前可能漏掉开引号或开尖括号。

undef语句语法错

`!undef`语句必须仅有一个标识符, 语句体中不能再有其他东西。

字符常量太长

字符常量只能是一个或两个字符长。

命令参量太长

MS-DOS限定, 由MAKE执行的命令的自变量不能多于127字符。

命令语法错

在下列几种情况下, 会出现此信息:

- (1) `make`文件的第一规则行以空白开头
- (2) 隐式规则未包含`.ext. ext;`
- (3) 显式规则在`:`前没有名字
- (4) 宏定义在`=`前没有名字。

除数为零

在`!if`语句中, 除法或取余运算以0为除数。

!if语句中表达式语法错

`!if`语句中的表达式结构有错, 例如圆括号不匹配, 多余运算符或缺少运算符, 缺少或多余常量。

文件名太长

在`!include`指令中给出的文件名太长, 编译程序无法处理。MS-DOS中的文件名不得超过64字符。

常量表达式中有非法字符

MAKE在常量表达式中遇到了不允许的字符。若字符是一个字母, 则可能是标识符拼写错误。

非法的八进制数字

八进制数制中含有0--7以外的数字。

宏扩展太长

宏扩展后不得超过4096个字符。当宏递归地扩展自身时经常发生这种错误。宏不能有效地扩展自身。

elif语句位置错

`!elif`指令没有与之匹配的`!if`指令。

else语句位置错

`!else`指令无相匹配的`!if`指令。

endif语句位置错

`!endif`指令无与之匹配的`!if`指令。

文件名无结束字符

在包括语句中的文件名后无相应的圆引号或闭尖括号。

目标重定义: xxxxxxxx

在多条显式规则的左边出现同一目标。

无法打开包括文件, xxxxxxxx. xxx

找不到指定的文件。如果一个包括文件包含其自身, 也会引起此错误。出错时, 检查一下该文件是否存在。

在#行开始的条件下出现非预期文件结束

在MAKE遇到`!endif`前源文件就结束。可能是`!endif`丢失了, 也可能是把`!endif`拼写错了。

非法预处理语句

在一行开始遇到字符`!`, 但跟在后面的语句名不是`error`, `undef`, `if`, `elif`, `include`, `else`或`endif`。

C. 3 Turbo连接程序TLINK

Turbo C集成开发环境版本(TC)含有一内部连接程序。而对于Turbo C的命令行版本(TCC), 连接程序TLINK是作为一个独立的程序被调用的。

TLINK简洁典雅, 它不像其他连接程序复杂繁琐, 相当紧凑快速。

在缺省情况下, Turbo C在成功地编译后调用TLINK。由TLINK把目标模块和库文件连接起来,

产生可执行文件。

本节介绍如何将TLINK作为独立运行的连接程序使用。

C. 3.1 调用TLINK

在DOS提示符下键入带或不带参数的tlink就可调用TLINK。

当无参数调用TLINK时，它显示如下参数和选择项表：

```
Turbo Link Version 1.0 Copyright(c) 1987 Borland International
The syntax is, TLINK objfiles, exefile, mapfile, libfiles
@xxxx indicates use response file xxxx
Options, /m = map file with public
        /x = no map file at all
        /i = initialize all segments
        /l = include source line numbers
        /s = detailed map of segments
        /n = no default libraries
        /d = warn if duplicate symbols in libraries
        /c = lower case significant in symbols
        /x = no map file at all
```

其中，The syntax is, TLINK objfiles, exefile, mapfile, libfile指出文件名需按一定次序给出，文件类型由逗号分开。例如，如有命令行：

```
tlink /c mainline wd ln tx, fin, mfin, lib\comm lib\support
```

TLINK将解释为：

- (1) 在连接时，字母大小写是有意义的(/c)。
- (2) 需连接的.OBJ文件是MAINLINE.OBJ, WD.OBJ, LN.OBJ和TX.OBJ。
- (3) 可执行程序名是FIN.EXE。
- (4) map文件是MFIN.MAP。
- (5) 连接的库文件是COMM.LIB和SUPPORT.LIB，它们均在子目录LIB中。

TLINK对无扩展名的文件名添上扩展名：

- (1) 对目标文件加.OBJ
- (2) 对可执行文件加文.EXE
- (3) 对map文件加.MAP
- (4) 对库文件加.LIB。

若未指定可执行文件名，TLINK将第一个目标文件名加.EXE产生可执行文件名。例如，若在上例中没有指定FIN为.EXE文件名，TLINK将生成MAINLINE.EXE作为可执行文件。

除非显式地在命令行中用/x选择项，TLINK一般都生成map文件。

- (1) 如果使用/m选择项，map文件包含公共量。
- (2) 如果使用/s选择项，map文件为详细分段映象。

下面是当决定map文件名时，TLINK遵循的规则：

- (1) 如果未指定.MAP文件，TLINK对.EXE文件的名加扩展名.MAP产生map文件名(。EXE文件名可在命令行或响应文件中给出。若未给出.EXE文件名，TLINK由第一个.OBJ文件的名生成.EXE文件)。

- (2) 若在命令行中(或在响应文件中)指定了map文件名，TLINK将扩展名.MAP加到给出的名上。

注意：即使指定了map文件名，如果使用了/x选择项，则仍不产生map文件。

C. 3.2 使用响应文件

TLINK的各种参数可在命令行上给出，也可在响应文件(response file)中给出，甚至可一部分在命令行给出，一部分在响应文件中给出。

响应文件是正文文件，它包含了命令行中TLINK之后键入的各选择项和/或文件名。

与命令行不同, 响应文件可以有连续几行。在行尾加上一个 + 字符, 即可将一长串目标文件或库文件分成多行。

此外, 连接程序的四部份参数: 目标文件、可执行文件、map文件、库文件可以在不同行开始。如果这样做就必须去掉用于分隔各部分的逗号。

为说明这些特征, 假设将前例中的命令行重写成如下响应文件FINRESP,

```
/c mainline wd +  
ln tx, fin +  
mfin +  
lib\comm lib\support
```

这时需键入的TLINK命令为: tlink @finresp。

注意: 必须在文件名前写上@字符以示下个名字是响应文件名。

可以将连接命令分为几个响应文件。例如可以把前面的命令行分为如下两个响应文件,

文件名	内容
LISTOBSJ	mainline + wd + ln tx
LISTLIBS	lib\comm + lib\support

然后键入命令: tlink/c @listobjs, fin, mfin, @listlibs

C.3.3 使用TLINK连接Turbo C模块

Turbo C支持六种不同的存储模式: 极小, 小, 紧凑, 中, 大, 特大。使用TLINK生成可执行的Turbo C文件时, 必须包含当前存储模式使用的初始化模块和库文件。

使用TLINK连接Turbo C程序的一般格式是:

```
tlink C0x<myobjs>, <exe>, [map], <mylibs> [emu|fp87 msthx] Cx
```

其中的文件名含义如下:

<myobjs> = 要连接的.OBJ文件

<exe> = 指定的可执行文件名

[map] 指定的map文件名〔任选〕

<mylibs> = 在连接时刻要包括的库文件

在此TLINK命令行中的其他文件名代表Turbo C文件, 含义如下:

C0x = 对应存储模式x的初始化模块

emu|fp87 = 浮点库文件(选择一种)

mathx = 对应存储模式x的数学库

Cx = 对应存储模式x的运行时刻库

C.3.3.1 初始化模块

初始化模块名为C0x.OBJ, 其中x是对应存储模式的单个字符: t, s, c, m, l, h。若未连入正确的初始化模块, 则将产生一串错误信息, 告诉用户某一标识符未找到或栈未创建。

在目标文件序列中, 初始化模块必须作为第一个目标文件出现。初始化模块安排程序各段的次序。如它不是第一个, 程序段便不能正确存放在存储器中, 引起破坏性程序错误。

必须保证在TLINK命令行显式给出一个.EXE文件名, 不然程序名将是C0x.EXE, 可能不是所期望的。

C.3.3.2 库

在连接命令中, 在用户库之后, 也必须包含相应存储模式的库。这些库必须按一确定次序出现: 浮点库及一适当的数学库(这些是任选的), 以及相应运行时刻库。下面按此次序讨论这些库。

如果程序使用了浮点运算, 就必须在连接命令中包含一个浮点库(EMU.LIB或FP87.LIB)和一个数学库(MATHx.LIB)。

Turbo C的两个浮点库与程序的存储模式是无关的。

(1) 若想包含浮点仿真程序以便程序可在有/无8087或80287协处理器的机器上均能运行, 必须使用EMU.LIB。

(2) 如果程序总在带有协处理器的机器上运行, FP87.LIB库将产生一个较小且较快的可执行程序。

数学库的名字为MATHx.LIB, 其中x是对应的存储模式的单个字母, t, s, c, m, l, h。

连接命令行中, 总可以包含仿真程序库和数学库, 如果程序不作浮点运算, 库中不会有任何东西加到可执行程序文件中。当然, 如果知道在程序中无浮点运算, 直接从命令行中删去那些库, 可节省连接时间。

相应存储模式的C运行时刻库必须包含在命令行中。C运行时刻库名为Cx.LIB, 其中x是相应存储模式的单个字母。

注意: 如果使用浮点运算, 就必须在C运行时刻库之前包含数学库和仿真程序库, 如不这样做将可能导致连接失败。

C. 3.4 利用TCC使用TLINK

独立运行的Turbo C编译程序TCC, 可作为TLINK的‘前端程序’它将用正确的启动文件、库和可执行程序名调用TLINK。

为做到这点, 须在TCC命令行中显式给出文件扩展名.OBJ和.LIB。例如, 给定如下命令行:

```
tcc -mx mainfile.obj sub1.obj mylib.lib
```

TCC将使用文件C0x.OBJ, EMU.LIB, MATHx.LIB和Cx.LIB(初始化模块, 默认8087仿真库、数学库和运行时刻库, 其中x代表存储模式)来调用TLINK。TLINK把这些文件与用户模块MAINLINE.OBJ和SUB1.OBJ, 用户库MYLIB.LIB连接在一起。

注意: 当TCC调用TLINK时, 它总是使用/c(字母大小写敏感连接)选择项。

C. 3.5 TLINK选择项

TLINK选择项可出现在命令行中的任一位置。选择项由斜杠(/)后跟一指定选择项字母(m, s, l, i, n, d, x, c)组成。

如果有多个选择项, 其间空格可省略(即: /m/c与/m /c相同), 并且它们可以出现在命令行中的不同位置。下面讨论各种选择项。

/x, /m, /s 默认TLINK总是生成可执行文件的映象文件, 此默认映象文件仅包含此程序中的段, 程序始地址, 以及所有连接时产生的警告或错误信息的表。

如果想生成更完整的映象文件, /m选择将在映象文件中加入公共符号表, 这些符号将按递增地址序排列。这种映象文件在调试时很有用, 许多调试程序, 例如SYMDEB利用公共符号表可在调试时引用符号地址。

/s 选择项生成一个带段、公共符号表和程序首地址(与/m选择项相同)的映象文件, 但还加入了详细的段映象。下表是详细段映象的例子。

段的详细映象的例子

地 址	字节长	类	段 名	组	模 块	对齐/组合
0000,0000	0E5B	C = CODE	S = SYMB_TEXT	G = (none)	M = SYMB.C	ACBP = 28
00E5,000B	2735	C = CODE	S = QUAL_TEXT	G = (none)	M = QUAL.C	ACBP = 28
0359,0000	002B	C = CODE	S = SCOPY_TEXT	G = (none)	M = SCOPY	ACBP = 28
035B,000B	003A	C = CODE	S = LRSH_TEXT	G = (none)	M = LRSH	ACBP = 20
035F,0005	0083	C = CODE	S = PADA_TEXT	G = (none)	M = PADA	ACBP = 20
0367,0008	005B	C = CODE	S = PADD_TEXT	G = (none)	M = PADD	ACBP = 20
036D,0003	0025	C = CODE	S = PSBP_TEXT	G = (none)	M = PSBP	ACRP = 20

地 址	字节长	类	段 名	组	模 块	对齐/组合
036F,0008	05CE	C = CODE	S = BRK_TEXT	G = (none)	M = BRK	ACBP = 28
03CC,0006	066F	C = CODE	S = FLOAT_TEXT	G = (none)	M = FLOAT	ACBP = 20
0433,0006	000B	C = DATA	S = _DATA	G = DGROUP	M = SYMB.C	ACBP = 48
0433,0012	00D3	C = DATA	P = _DATA	G = DGROUP	M = QUAL.C	ACBP = 48
0433,00E6	000E	C = DATA	S = _DATA	G = DGROUP	M = BRK	ACBP = 48
0442,0004	0004	C = BSS	S = _BSS	G = DGROUP	M = SYMB.C	ACBP = 48
0442,0008	0002	C = BSS	S = _BSS	G = DGROUP	M = QUAL.C	ACBP = 48
0442,000A	000E	C = BSS	S = _BSS	G = DGROUP	M = BRK	ACBP = 48

对于每个模块中的每个段，此映象包含地址、字节长、类、段名、组、模块和ACBP信息。

如果同一段出现在多个模块中，每个模块将作为独立的行出现（例如，SYMB.C），详细段映象中的大部分信息含意是明显的。

按Intel定义ACBP域将A（对齐）和C（组合）属性编码为4个二进制域的集合。TLINK只使用两个域A域和C域。映象中ACBP值以十六进制数打印。

下列域的值必须‘或’在一起，以产生要打印的ACBP值。

域	值	描 述
A 域	00	绝对段
(对齐)	20	字节编址段
	40	字编址段
	90	段编址段
	80	页编址段
	A0	寄存器的未命名绝对部份
C 域	00	可不被组合
(组合)	08	公共组合段

／1 该选择项在MAP文件中生成一段源代码行号。为使用该选择项，编译时须用-y（Line numbers-on）选择项来生成OBJ文件。如果不要TLINK生成映象文件（使用/x选择项），此选择项将不起作用。

／i 即使尾部段中不包含数据记录，／i选择项也使得尾部段输出到可执行文件。当然，一般不需要这样做。

／n 该选择项使连接程序忽略由某些编译程序指定的默认库。如果默认库在另一个目录中，此选择项是必要的。因为TLINK不支持库搜索。在连接其他语言写的模块时可用此选择项。

／c 该选择项使得在外部量和公共量中区分大小写。例如默认情况下，TLINK把fred，Fred和FRED看作相同的，而／c选择项使得TLINK认为它们是不同的。

／d 一般地，如果一个符号在多个库文件中出现，TLINK将不提出警告。如果此符号必须包含在程序中，TLINK将使用命令行中第一个说明此符号的文件的副本，因为相同符号经常出现，TLINK一般不对此发出警告。下面假设一些情况介绍如何使用此特性。

假设有两个库，一个叫SUPPORT.LIB，另一个辅助库叫DEBUGSUP.LIB，还假设DEBUGSUP.LIB包含了SUPPORT.LIB中某些子程序（但DEBUGSUP.LIB中的子程序功能略微不同，例如子程序的调试版本）。如在连接命令中先写DEBUGSUP.LIB，则得到调试子程序而不是SUPPORT.LIB中的子程序。

如果不使用此特性或者不知道哪个子程序重名，用／d选择项，这使得TLINK列出库中所有重名符号，即使这些符号在程序中不使用。

／d选择项同样使TLINK对既在OBJ文件又在LIB文件中出现的符号发出警告。因为出现于命令行的第一个文件中的符号被连入，所以使用的是OBJ文件的符号。

Turbo C各种连接命令所要使用的分配的库均不含重名符号，这样虽然EMU.LIB和TP87.LIB（或CS.LIB和CL.LIB）明显有重名符号，但它们不能在一次连接中一起使用。其他库，例如

EMU, LIB, MATHS, LIB和CS. LIB之间无重名符号。

C. 3.6 限制

如前所说TLINK并未提供过多的选择项, 下面是TLINK的限制,

- (1) 不支持覆盖。
- (2) 不支持Microsoft CodeView调试程序(但SST, SYMDEB和PFIIX能很好地运行)。
- (3) 只部分支持公共变量, 须提供一个公共段来统一它们。
- (4) 最多能有8182个符号和4000个逻辑段。
- (5) 具有相同名和类的段或者能完全组合, 或者不能组合(只有汇编程序员会遇到这种问题)。
- (6) 用Microsoft C 或 Microsoft Fortran编译的代码不能用TLINK连接。这是因为Microsoft语言在其.OBJ文件中有未公开的目标记录格式, 当前TLINK是不支持的。TLINK可用于Turbo C(集成开发环境和命令行版本), 也可用于MASM和其他编译程序。但一般地它不能代替MS连接程序。

C. 3.7 出错信息

TLINK区别三类出错: 警告、非灾难性错误和灾难性错误。

- (1) 警告只是对用户可能要略作修改的情况加以提示。当发出警告时, 仍能生成.EXE和.MAP文件。
- (2) 非灾难性错误不删除.EXE文件或.MAP文件, 但不能执行.EXE文件。
- (3) 灾难性错误使TLINK立即停止, 并删去.EXE和.MAP文件。

本节列出的出错信息中将用下列符号。这些符号在具体出错信息中, 将替换成适当的名或值。

< sname >	符号名
< mname >	模块名
< fname >	文件名
x x x x h	4位十六进制数, 后跟'h'

C. 3.7.1 警告

TLINK仅有三种警告。前两种有关符号的重复定义。第三种适用于极小模式程序, 它指出栈未定义。下面是警告信息:

警告: XXX在模块YYY中重名

符号XXX在模块YYY中定义了两次。

这可能出现在Turbo C的目标文件中, 例如, 在一源文件中, 两个不同的Pascal名字差别只在大小写上。

警告: 模块YYY中定义的XXX在ZZZ模块重名

符号XXX在两个模块中被定义。如果在命令行中有一目标文件写了两次, 或某符号的两个副本之一拼写错, 就会发生此情况。

警告: 无栈

如果连接包括的所有目标文件和库中均没有定义栈区, 则发生此警告。这对于Turbo C中的极小存储模式或要转换为.COM文件的应用程序来说是一个正常的信息。对于其他程序, 这意味着错误。

如果除极小存储模式外的Turbo C程序产生此信息, 则要检查Cox启动目标文件是否正确。

C. 3.7.2 非灾难性错误

TLINK只有两种非灾难性错误。如前所说, 当非灾难性错误发生时, .EXE和.MAP文件不被删除。但这些错误在'集成开发环境'中当作灾难性错误处理, 下面是出错信息:

模块YYY中XXX未定义

模块YYY引用了符号XXX, 但连接的各目标文件和库均未定义该符号。检查此符号拼写的正确性。通常, 如果在不同源文件中pascal和cdecl类型的符号说明匹配不当, 可能产生此错误。

地址计算溢出, 模块XXXXXXXX中框架 = xxxxh, 目标 = xxxxh, 位移 = xxxxh;

这意味着某目标文件中出现了错误的数据或代码引用, 连接时TLINK地址计算出错。在地址计算中目标文件确定了被引用的存储单元名和存储单元所在的区段名。框架值是根据目标文件, 该存储单元应该在的段; ‘目标’值是存储单元实际所在的段, ‘位移’值是存储单元所在段中的位移。

此信息通常是由于存储模式不匹配引起的。最可能的原因是, 在另一代码段中使用近调用。如果对数据变量或函数引用数据产生近调用, 也会发生此错误。

为了诊断此问题, 应生成一个带公共符号(/m)的映象文件。出错信息中的目标值和位移域值应为被引用符号的地址。如果目标和位移域与映象文件中的符号不匹配, 则寻找与给出信息中地址最接近的符号。引用出现在模块XXXXXXXX中, 因此在该模块源文件中寻找错误引用。

如果这些技术未能找到失败的原因, 或者编写程序用的是汇编语言或Turbo C以外的高级语言, 则还有其他产生此信息的原因。即使在Turbo C中, 如果对一个给定的存储模式使用不同于默认值的段或组名后, 也会产生此信息。

C. 3.7.3 灾难性错误

当灾难性错误发生时, TLINK停止工作并删除.EXE和.MAP文件。

XXXXXXXX, XXX: 目标文件格式错

遇到了格式不正确的目标文件。这通常是由于在目标文件位置写了源文件或目标文件未完全生成而引起的。在编译期间重启动机器, 或碰到Ctrl-brk时编译程序未将其输出目标文件删除时会发生此错误。

XXXXXXXX, XXX: 文件打不开

当该文件不存在或拼错时, 发生此错误。

参数中有错误字符

在命令行或响应文件中遇到了下列符号之一: " * < = > ? [] | 以及除了制表符、换行、回车、Ctrl-Z外的任何控制字符。

msdos出错, ax = XXXXh

如果MS-DOS调用返回一个非期望错误则产生此信息。ax值是错误代码。这可能表示TLINK的内部错误或MS-DOS错。TLINK使用的可能产生此错误的MS-DOS调用只有文件读、写、关闭。

无足够内存

没有足够的存储区来完成连接处理。请移出当前已装入内存的任何终端和常驻内存的应用程序, 或者减小当前RAM磁盘的大小, 然后再运行TLINK。

段超过64K

当不同源文件中相同名的段结合在一起时, 如果对一个数据或代码段定义了太多的数据就会发生此错误。

当某组的段联结在一起时, 如果该组超过64k字节也会发生此错误。

符号超出限制

在一次连接中, 最多可以定义8182个公共符号、名和组名。如果超出此限, 则发生错误。

非期望的组定义

目标文件中组定义必须以一个特定序列出现。仅当编译程序产生有缺陷的目标文件时才发出此信息。如果这是由Turbo C建立的文件中发生, 则请重编译此文件。若仍发生此问题, 则须与研制单位联系。

非期望的段定义

目标文件中的段定义必须以一个特殊序列出现, 只有当编译程序产生一个有缺陷的目标文件时, 才会产生此信息。如果这发生在由Turbo C处理的文件中, 则请重编译此文件, 若仍有该问题, 请与研制单位联系。

不认识的选择项

在命令行或某个响应文件中, 斜杠(/)后不是所允许的选择项。

写失败，磁盘满

如果TLINK无法把想写的数据写完，发生此错误，这一般是由磁盘满引起的。

C. 4 Turbo 库管理程序 TLIB

本节描述Turbo C1.5版新增加的库管理程序TLIB。

C. 4.1 TLIB是什么？

TLIB是Borland公司的 Turbo 库管理实用程序，用于管理包含.OBJ（目标模块）文件的库。库作为一个单独的整体，对处理目标模块的集合是很方便的。

包括在Turbo C中的库是用TLIB建立的。用户可以使用TLIB建立自己的库，也可以修改Turbo C的库、用户自己的库、由其他程序员完成的库、或者用户通过商业途径获得的库。利用TLIB可完成以下工作：

- 使用一组目标模块建立一个新库，
- 将目标模块或其他库加入到一个已有的库中，
- 从原有的库中删除（remove）目标模块，
- 替换原有的库中的目标模块，
- 从原有的库中抽取（extract）目标模块，
- 列出一个新的或原有的库的内容等。

当修改一个库时，TLIB总是建立原库的副本，其扩展名为.BAK。

虽然TLIB对于Turbo C建立的可执行程序不是很重要的，但它是一个非常有用的程序开发工具。通过实际应用将会发现，TLIB对一个大型开发工程来说是必不可少的。如果用户使用别人开发的目标模块库，那么在需要时，就可用TLIB来维护那些库。

C. 4.2 使用目标库的优点

当用户用C语言编制程序时，可建立一个有用的C语言函数集合，就像C运行库中的函数一样。由于C语言的模块性，用户可以把那些函数分成许多独立的可编译源文件。在任何特定的程序中，用户可能只使用函数集中的一个子集。然而，选出所要使用的文件这件工作可能是乏味的。另一方面，如果总是包括所有的源文件，那么程序将变得非常庞大而不实用。

目标模块库能解决C函数集的管理问题。当用户将程序和一个库进行连接时，连接程序扫描库并且自动选出仅当前程序需要的那些模块。另外，库占用的磁盘空间比目标模块文件集少得多，尤其是当每一目标文件很小时。由于使用库只需打开一个单独的文件来替代打开每一个目标模块文件，所以能提高连接程序的速度。

C. 4.3 TLIB命令行的组成

在DOS提示符下，键入TLIB命令行就可以运行TLIB。若要了解TLIB的使用说明，仅需打入TLIB并按ENTER键即可。

TLIB命令行采用下面的语法格式，列在方括号中的项是任选的。

tlib libname [/c] [operations] [, listile]

其中：tlib 启动TLIB的命令名。

libname 欲建立或管理的库的DOS路径名。TLIB命令必须给出一个libname，通配符是不允许的。如果未给出扩展名，TLIB就假设扩展名为.LIB。在此建议不要使用除.LIB外的其他扩展名，因为为了识别库文件，TCC和TC的工程制作（project-make）工具都要求扩展名.LIB。

注意：如果命名库不存在，且具有add操作，那么TLIB就创建这个库。

/c “大小写敏感”标志。该选择项不常使用，详见“高级操作”的/c选择项。

operations TLIB完成的操作表。操作可按任意次序出现。如果只想检查库的内容，则不要给出任何操作命令。

listfile 库内容的列表文件名。listfile名(如果给出)必须在前面加逗号(,)。如果未给出文件名则不会产生任何列表。列表时,首先按字母顺序列出各个模块,然后按字母顺序列出该模块中定义的公共符号。listfile的缺省扩展名为.LST。

列表文件名可取为CON,表示在屏幕上列表;为PRN,则在打印机上列表。

以上简要描述了TLIB命令行各成份,以下对有关成份作进一步的描述。

C.4.3.1 操作表

操作表描述了需要TLIB程序完成的动作,它由一系列操作组成,每个操作都由一个或两个动作符号和文件名或模块名组成。在动作符号、文件名或模块名之间可用空格,但不能在两个动作符号之间或一个名中出现空格。

可以在命令行中放置多个操作,但不能超过DOS规定的每行127个字符的限制。操作排列的顺序无关紧要,TLIB总是以特定的顺序来执行操作:

(1) 首先执行所有抽取操作。

(2) 然后执行所有删除操作。

(3) 最后执行所有添加操作。

替换一个模块时,首先是删除它,然后添加一个替代它的模块。

(1) 文件和模块名

当TLIB在库中增加一个目标模块文件时,该文件被简单地称为一个模块。TLIB根据给出的文件名、驱动器名、路径名和扩展名信息来寻找模块名(一般不给出驱动器名、路径名和扩展名信息)。

注意:TLIB总是假设合理的缺省值。例如,将当前目录中一个扩展名为*.OBJ的模块添加到库中,只需给出模块名即可,而不必给出路径名和OBJ扩展名。

在文件或模块名中不允许含有通配符。

(2) TLIB操作

TLIB识别三个动作符号(-, +, *),可单独地或组合成对地使用它们完成五个不同的操作。若操作由一符号对来实现,则符号的次序无关紧要。操作符和它们的功能如下:

动作符号	名	功 能
+	添加	TLIB在库中增加指定的文件。如果文件无扩展名,则TLIB假定扩展名为*.OBJ。如果文件本身是一个库(有*.LIB扩展名),则该操作将指定库中的所有模块添加到目标库中。 如果目标库中已存在一个要增加的模块,则TLIB显示出信息并且不执行该操作。
-	删除	TLIB从库中删除指定的模块。如果库中无此模块,则TLIB显示出信息。
*	抽取	TLIB通过从库中将相应的模块拷贝到文件中的方式来建立命名的文件。如果这个模块不存在,则TLIB给出信息而不建立此文件。若指定的命名文件已存在,则原文件被覆盖。
-+	替换	TLIB用相应的文件替换指定的模块。其操作是先删除后添加。
+-		
-*	抽取和删除	TLIB将指定模块拷贝到相应的文件中,然后从库中删除该模块。其操作是先抽取后删除。
*-		

删除操作仅需要模块名,但TLIB允许输入具有驱动器名和扩展名的完全路径名。当然,除了模块名外,其他都可以忽略。

对库中的模块重新命名是不可能的。为了给库中模块换名,必须首先抽取并且将它删除,重新命名新建的文件,然后将其添加到库中。

C.4.3.2 建立一个库

要建立库,只需简单地将模块添加到不存在的库中即可。

C.4.4 使用响应文件

当处理大量操作或发现在重复某些操作时,可使用响应文件。响应文件仅是一个ASCII正文文件(用Turbo C编辑程序很容易建立),该文件包含了所有或部分TLIB命令。使用响应文件可以建立一个大于DOS命令行的TLIB命令。

在TLIB命令行的任何位置设置@(<pathname>)就可以使用响应文件的pathname。

- 一行以上的正文可组成一个响应文件。在行末使用“and”字符(&)指出下行是上面一行的继续。

- 在响应文件中不必放入所有TLIB命令,该文件可提供TLIB命令行的一部分,用户可输入其余的部分。

- 在只有一行的TLIB命令行中,可使用一个以上的响应文件。

见后面的“例C.4.1”,它给出了一个样板响应文件和一个与它相应的TLIB命令行。

C.4.5 高级操作:/c选择项

在库中加入一个模块时,TLIB保持由库中模块定义的所用符号为字典次序。在库中的所有符号必须有所区分。如果欲在库中增加一个引起符号重复的模块,TLIB将给出信息并且不增加该模块。

一般情况下,当TLIB查看库中重复符号时,把大写与小写字母看作是相同的。例如,符号lookup和LOOKUP是作为重复的符号来对待的。因为C语言把大写字母和小写字母看作是不相同的,所以用户必须使用/c选择项来向库中加入一个模块,该模块中含有一个符号,它仅在大小写上与库中已有的符号不同,/c选择项使得TLIB接受带有与库中符号仅在大小写上不同的符号的模块。

没有/c选择项,TLIB将拒绝仅是大小写不同的符号,用户也许不可理解,特别是C语言是大小写敏感的语言。其原因是一些连接程序不能正确识别库中仅大小写不同的符号。

TLINK能够识别大写和小写符号,并且很可能接受含有仅大小写不同的符号的库。只要是TLINK用到的库,就可使用TLIB的/c选择项,而不会有任何问题。

然而,如果使用别的连接程序(或者允许其他人用别的连接程序连接该库),为保险起见,这时不应使用/c选择项。

C.4.6 例子

[例C.4.1] 现给出一些简单的例子以说明TLIB的使用。

- (1) 用模块X.OBJ, Y.OBJ和Z.OBJ建立一个名为MYLIB.LIB的库,可输入:
tlib mylib +x +y +z
- (2) 建立相同于(1)的库并建立列表,可输入:
tlib mylib +x +y +z, mylib.lst
- (3) 建立已存在库CS.LIB的列表,输入:
tlib cs, cs.lst
- (4) 用一个新的拷贝替换模块X.OBJ,在MYLIB.LIB中增加A.OBJ并且删除Z.OBJ,可输入:
tlib - +x +a -z
- (5) 从MYLIB.LIB中抽取模块Y.OBJ并建立一个列表,可输入:
tlib mylib *y, mylib.lst
- (a) 用A.OBJ, B.OBJ, ..., G.OBJ建立一个新库,要求使用响应文件,
首先建立一个正文文件ALPHA.RSP,其内容为:

```
+a.obj +b.obj +c.obj &
+d.obj +e.obj +f.obj &
+g.obj
```

然后用TLIB命令

```
tlib alpha @alpha.rsp, alpha.lst
```

[例C.4.2] 把GRAPHICS.LIB添加到Turbo C库中。

当在Turbo C程序中使用图形,首先必须在命令行中包含graphics.lib(如果使用TCC,

“`tcc foo.c graphics.lib`”)或是创建一个有`graphics.lib`在内的工程。为避免这样做,用TLIB把图形库模块加到标准`c*.lib`库中。采用下列一条或若干条TLIB命令即可(对`graphics.lib`不要忘了扩展名“`lib`”)。

```
tlb cs+graphics.lib (小模式库)
tlb cc+graphics.lib (紧凑模式库)
tlb cm+graphics.lib (中模式库)
tlb cl+graphics.lib (大模式库)
tlb ch+graphics.lib (特大模式库)
```

把图形模块加到所有标准库文件中可能是一个很好的办法,但如果知道在用图形程序时只用到某些存储模式,或磁盘空间不够时,应有选择地改变库文件。

当前目录必须是保存C库的那个目录,否则必须在TLIB命令行中加入完整的路径。例如:

```
cd\turboc\lib
tlb cm+graphics.lib
或 tlb\turboc\lib\cm+\turboc\lib\graphics.lib
```

注意,tlb将把原来的`C*.LIB`库改名为`C*.BAK`,对这个文件用户可以删除,或保留下来作为参考。

C.5 文件搜索程序GREP

本节描述GREP.COM,这是著名的UNIX文件搜索实用程序的Turbo C快速版本。

C.5.1 GREP是什么?

GREP是能同时在几个文件中搜索正文的强功能的搜索实用程序。

GREP命令行语法的一般格式是:

```
grep [options] searchstring filespec [filespec ...filespec]
```

例如,如果想查看哪个源文件调用了`setupmodem`函数,则可用GREP来查看所在目录中的所有C文件的内容以寻找串`setupmodem`,其命令为:

```
grep setupmodem *.C
```

C.5.2 GREP的选择项及优先次序

在命令行中, `options`由一个或多个破折号(`-`)加字符组成。每个选择项都是可打开或关闭的开关;在字符后输入一个加号(`+`),可使选择项打开,而输入破折号(`-`)则关闭选择项。

缺省表示打开(即隐含`+`);例如, `-r`等同于`-r+`。用户可以列出多个独立的选择项(如`-i -d -l`等)或者把它们组合起来(如`-ied`或`-ie-d`等),对GREP来说,它们的作用都是相同的。

下面列出GREP的选择项字符及其含义:

- c 只计数;只打印匹配行的数目。即对至少包括一个匹配行的文件, GREP打印文件名和匹配行的数目,而不打印匹配行。
- d 目录;对每一个命令行中说明的`filespec`, GREP在指定目录以及在指定目录的所有子目录中查找与文件说明(`filespec`)匹配的文件。如果给出`filespec`而没有给出路径,则GREP假定文件在当前目录中。
- i 忽略大小写;GREP忽略大小写字母之间的差别,即GREP把字母`a-z`与对应字母`A-Z`看作是相同的。
- l 列出匹配文件;打印相匹配的文件名。GREP找到匹配文件后,就打印出该文件名,并立即处理下一个文件。
- n 行号;GREP打印的匹配行前都有它的行号。
- r 搜索正则表达式;由`searchstring`定义的正文作为正则表达式(`regular expression`),而不作为文字串。
- v 不匹配;只打印不匹配行。只有那些不包括查找串的行才被看作是不匹配行。
- w 写选择项;GREP把在命令行中给出的选择项和它的缺省选择项组合起来,作为一种新的缺省值

写入GREG.COM文件(换句话说,GREG是自构型的)。该选择项允许用户定制适合自己需要的缺省设置。

-z 所有;GREG打印所搜索的每一文件的文件名。在每一匹配行前置有行号,即使计数值为零,也给出每一文件中匹配行的计数。

GREG中的有些选择项覆盖某些其他的选择项,其优先次序为,

-z 覆盖-l-c-n

-l 覆盖-c-n

-c 覆盖-n

例如,输入下面的GREG命令行,

```
grep -c-z main(my*.c
```

GREG在所有与MY*.C匹配的文件中搜索字符串 main(的匹配行,并且打印所搜索文件的文件名、匹配行的行号和每个所搜索文件中匹配行计数。

记住,每个选择项是一个开关;它的状态反映了最后一次转换的方式。任一给定时刻,选择项只能是开或关。命令中当前给出的选择项覆盖它先前的定义。例如,在命令行输入

```
grep -r -i -d -i -r- main(my*.c
```

则GREG置-d选择项开,-i选择项开,-r选择项关并运行。

在GREG.COM中可以用-w选择项来安装用户自己喜欢的每一选择项的缺省值。例如,如果想要GREG总是执行所有搜索(-z开),则用户可以用下面的命令行来安装:

```
grep -w-z
```

C.5.3 搜索字符串及正则表达式中的操作符

searchstring的值定义了GREG要搜索的模式。一个搜索串可以是一个正则表达式或者是一个文字串。在正则表达式中,某些字符有特殊含义;它们是控制搜索的操作符。在文字串中,没有操作符,每个字符都作为文字形式处理。

可以用引号把搜索串括起来,以避免把空格和制表符作为分界符对待。匹配不能超出行的边界(一个匹配必须包含在一个行中)。

表达式可以是单个的字符,也可以是用括号括起来的一组字符,正则表达式的并置仍然是正则表达式。

使用-r选择时,搜索串就作为一个正则表达式(不是字面表达式,literal expression)来对待,并且下面的字符有特殊含义:

^ 在表达式首部的^符号匹配行首

\$ 在表达式尾部的美元符 \$匹配行尾

• 句点匹配任意字符

* 后接星号通配符的表达式匹配零个或多个该表达式的出现 例如,在fo*中,*操作是对表达式o而言的,它匹配f, fo, foo等(f后接零个或多个o),但不匹配fa。

+ 后接加号的表达式匹配一个或多个该表达式的出现;fo+匹配fo, foo等,但不匹配f。

[] 在方括号中的串与该串中的任何字符匹配,但不匹配其他字符。如果串的第一个字符是^符号,则表达式与除该串中字符外的任何字符匹配。例如,[xyz]匹配x,y或z,而[^xyz]匹配a和b,但不匹配x,y或z。用户可以用(-)联结两个字符来说明字符范围。它们可组合起来形成表达式(如[a-bd-z?]将匹配?和除c以外的任何小写字母)。

\ 斜线转义符(\)使GREG搜索它后面的字面字符。例如,\.匹配一个句号而非其他任何字符。

注意:前面描述的四个字符(\$,*和+)在方括号内使用时无任何特殊含义。另外,字符^只有紧跟在集合定义的开头时(即紧跟在[之后),才特殊对待。

在上述表中未提到的字符只与它本身匹配(例如,>匹配>,#匹配#,等)。

C.5.4 文件说明

GREG命令行的第三项是filespec,即文件说明,它指示GREG要搜索的文件(或一组文件)。filespec可以是一个显式文件名,也可以是嵌入?和*通配符的“类属”文件名。另外,用户可以键入一个路径(驱动器和目录信息)作为filespec的一部分。如果给出的filespec不带有路径,则GREG只在当前目录下搜索。

0.5.5 带注释的例子

下面的例子中,假定GREP所有选择项的缺省值都为关闭(off)状态。

例1

命令行: `grep main(*.c`

匹 配: `main()`

`mymain(`

不匹配: `mymainfunc()`

`MAiN(i, integer);`

所搜索的文件: 当前目录下的*.c

注: 根据缺省知, 搜索是大小写敏感的。

例2

命令行: `grep -r [^a-z] main\ (*.c`

匹 配: `main(i, integer)`

`main(i, j, integer)`

`if (main()) halt,`

不匹配: `mymain()`

`MAIN(i, integer);`

所搜索的文件: 当前目录下的*.c

注: 这里的搜索串使GREP搜索前面没有小写字母([`^a-z`])的字main, 但它后面可接零个或多个空格(`*`), 然后是一个左括号。

由于空格和制表符通常被看作是命令行的分界符, 因此如果想把它们作为正则表达式的一部分, 则必须括引它们。这种情况下, main后的空格可用斜线转义符(`\`)来括引, 也可将空格置于双引号中([`^a-z main" *`])来做到。

例3

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`

所搜索的文件: 当前目录的*.C和*.INC

注: 由于斜杠(`\`)和句点(.)通常具有特殊含义, 所以, 若想搜索它们, 就必须在它们的前面设置斜线转义符(`\`)。

例4

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 tile has at least 2000 words.`

`He misspells toward as toward.`

所搜索的文件: 当前目录中的*.DOC

注: 这一格式从基本上定义了如何对给定的字进行搜索。

例5

命令行: `grep "search string with spaces" *.doc *.asm`

`a, \work \myfile.*`

匹 配: `This is a search string with spaces in it.`

不匹配: `THIS IS A SEARCH STRING WITH SPACES IN IT.`

`This is a search string with many spaces in it.`

所搜索的文件;当前目录中的*.DOC和*.ASM及驱动器A上\WORK目录中的MYFILE.*。

注:这是一个如何搜索嵌入空格的串的例子。

例6

命令行: `grep -rd "[, .: ? , \"]" $*.doc`

匹配: He said hi to me.

Where are you going?

Happening in anticipation of a unique situation,

Examples include the following:

"Mang men smoke, but fu man chu."

不匹配: He said "Hi" to me

Where are you going? I'm headed to the beach this

所搜索的文件: 当前驱动器的根目录和所有它的子目录中的*.DOC。

注:该例子搜索行末字符为, .: ? '和"的行。注意,在方括号中双引号前有转义符(\),所以它作为正常字符而不是串的结束引号。同时注意字符\$是如何出现在引用串外的。该例子说明了正则表达式是怎样并置从而形成一个较长的表达式。

例7

命令行: `grep -ild "the" *.doc`

或 `grep -i-l-d "the" *.doc`

或 `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're headed to

所搜索的文件: 当前驱动器的根目录和它所有的子目录中的*.DOC。

注:该例忽略大小写并且打印至少含有一个匹配的文件名。命令行的三种形式说明了指定多重选择项的不同方法。

C.6 图形驱动程序和字体转换程序 BGIOBJ

本节说明BGIOBJ的使用方法。BGIOBJ是一个实用程序,它允许用户使用非动态机制加载字符字体到用户图形程序中。

C.6.1 BGIOBJ 是什么?

BGIOBJ是把图形驱动程序文件和字符集合(笔划字体文件)转换成目标文件的实用程序。一旦文件被转换,就可以将其连接到用户程序中,使之成为可执行文件的一部分。该实用程序扩充了图形软件包的动态加载机制。用这种方法,可在执行时从磁盘上把图形驱动程序和字符集合(笔划字体)加载到用户程序。

直接把驱动程序和字符字体连接到用户程序中是一个很大的优点,因为可执行文件包含所有(或大多数)它所需要的驱动程序或和字体文件,并且在运行时不必到磁盘上存取驱动程序和字体文件。然而,把驱动程序和字体文件连入可执行文件,将增加可执行文件的长度。

使用BGIOBJ.EXE可把驱动程序和字体文件转换成可连接的目标文件,其语法为:

`BGIOBJ <source file>`

这里<source file>是要转换成目标文件的驱动程序或字体文件。所建立的目标文件的文件名与源文件相同,但扩展名为.OBJ。例如,EGAVGA.BGI产生EGAVGA.OBJ,SANS.CHR产生SANS.OBJ等。

C.6.1.1 添加新的.OBJ文件到GRAPHICS.LIB

应该把驱动程序和字体文件的目标模块加入到GRAPHICS.LIB中,这样,当连接图形子程序时,连接程序就能够找到它们。如果不把这些新目标模块加入到GRAPHICS.LIB中,则需在TC.L工程文件(.PRJ)中,TCC命令行或TLINK命令行下加入这些目标模块。为把这些目标模块加入到GRAPHICS.LIB中,可用下列命令启动Turbo C库管理程序TLIB:

`tlib graphics + <object file name> [+ <object file name>...]`

这里<object file name>是由BGIOBJ.EXE建立的目标文件名(如CGA,EGAVGA,GOTH

等)。扩展名.OBJ是隐含的,故可省略,在同一命令行中可加入数个文件以节省时间,参见下面的例子(有关TLIB的信息参见C.4节)。

C.6.1.2 注册驱动程序和字体文件

在把驱动程序和字体目标模块加入 GRAPHICS.LIB 后,必须注册所有需连接的驱动程序和字体文件。实现方法是在程序中(调用 initgraph之前)调用registerbgidriver和registerbgifont。这就通知图形系统,那些文件已存在,并且保证连接程序建立可执行文件时将其连上。

每一注册子程序可有一参数,即在 GRAPHICS.H 中定义的符号名。如果驱动程序或字体文件注册成功,则注册子程序返回一非负值。

下表列出了 Turbo C 所有的驱动程序和字体文件。它给出了 registerbgidriver 和 registerbgifont 可用的名。

驱动程序文件 (* .BGI)	registerbgidriver 可用符号名	字体文件 (* .CHR)	registerbgifont 可用符号名
CGA	CGA_driver	TRIP	triplex_font
EGAVGA	EGAVGA_drive	LITT	small_font
HERC	Herc_driver	SANS	sansserif_font
ATT	ATT_driver	GOTH	gothic_font
IBM8514	IBM8514_driver		
PC3270	PC3270_driver		

现给出一个完整的例子,它把 CGA 图形驱动程序、粗黑体字形文件和三倍字形文件转换成目标模块,然后将它们连接到程序中。

(1) 下面三个命令表示用 BGI OBJ.EXE 来实现将二进制文件转换成目标文件。

```
bgiobj cga
bgiobj trip
bgiobj goth
```

上述命令建立了三个文件: CGA.OBJ, TRIP.OBJ 和 GOTH.OBJ。

(2) 用下面的 TLIB 命令行将目标模块加入 GRAPHICS.LIB 中。

```
tlb graphics +cga +trip +goth
```

如果不将目标文件加入到 GRAPHICS.LIB 中,则需把目标文件名 CGA.OBJ 和 TRIP.OBJ 和 GOTH.OBJ 加到工程文件中(如果用户正使用 Turbo C 的集成环境),或加到 TCC 命令行中。例如, TCC 命令行应为:

```
tcc nftgraf graphics.lib cga.obj trip.obj goth.obj
```

(3) 在用户图形程序中注册这些文件,如下所示:

```
/*header file declares CGA_driver, triplex_font, and gothic_font*/
#include <graphics.h>
/*register and check for errors (one never knows...)*
if (registerbgidriver(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*/
/*...*/
```

如果连接了某个驱动程序和/或字体文件以后遇到连接程序错误:“段长超过64K”,则参看下节的处理。

C.6.2 /F 选择项

本节解决上节未提出的问题(特别是对极小、小和紧凑模式的程序)。

按照缺省,由 BGI OBJ.EXE 建立的文件都使用相同的段(称为-TEXT)。如果用户程序连接了许多驱动程序和/或字体文件,或者使用了极小、小或紧凑存储模式,就会产生错误。

如果在极小模式程序中产生了这个错误,则无法解决这个问题。用户必须除去一些或所有的驱动程序和字体文件,并且使用动态的驱动程序/字体文件加载机制。

对其他模式的程序,在转换一个或多个驱动程序或字形文件时,可使用BGIOBJ的/F选择项。该选择项使BGIOBJ使用形为<filename>_TEXT的段名,这样,连接所有的驱动程序和字体文件就不会使缺省段处于超负荷状态(在小和紧凑模式程序中,连接所有程序代码)。例如,下面两个BGIOBJ命令行使BGIOBJ使用形为EGAVGA_TEXT和SANS_TEXT的段名:

```
bgiobj /F egavga
bgiobj /F sans
```

当使用了/F选择项,BGIOBJ即将F加入目标文件名中(EGAVGAF.OBJ, SANSF.OBJ等),并把-far加入到registerfarbgidriver和registerfarbgifont所使用的名中(如,EGAVGA_driver变成EGAVGA_driver_far)。对于用/F建立的文件,必须使用远程(far)注册子程序代替标准的registerbgidriver和registerbgifont。例如:

```
if(registerfarbgidriver(EGAVGA_driver_far) < 0) exit(1);
if(registerfarbgifont(sansserif_font_far) < 0) exit(1);
```

C.6.3 BGIOBJ的高级特征

本节介绍BGIOBJ的一些高级特征以及子程序registerfarbgidriver和registerfarbgifont。Turbo C的初学者不要轻易使用它们。

下面是BGIOBJ.EXE实用程序完整的语法:

BGIOBJ [/F] <source> <destination> <public name> <seg_name> <seg_class>

成份	功 能
/F或-F	该选择项使BGIOBJ.EXE使用一个非-TEXT(缺省段)的段名,并且改变公用名和目标文件名(有关/F细节,见前一节)。
<source>	需转换的驱动程序或字体文件。如果不是用Turbo C所带的驱动程序或字体文件,则应指定完全文件名(包括扩展名)。
<destination>	要建立的目标文件名。缺省目标文件名是<source>.OBJ。如果使用/F选择项,则是<source>F.OBJ。
<public name>	这是在调用registerfarbgidriver或registerfarbgifont(或其far版本)来连接目标模块的程序时所用的名。public name是由连接程序使用的外部名,所以它应该是在程序中使用的名,并且在其名前有下列线。如果程序使用pascal调用约定,则仅用大写字母,不再加下列线。
<seg_name>	这是任选的段名,缺省值是-TEXT(如果使用/F,则是<filename>_TEXT)。
<seg_class>	这是一个任选的段类,缺省值是CODE。

除<source>外的所有参数都是任选的。如果需要说明一个任选参数,则其前面的所有参数都须说明。

如果用户要使用自己的公用名,则需使用下面的格式对程序加以说明:

```
void public_name(void); /* if /F not used, default segment name used */
extern int far public_name[]; /* if /F used, or segment name not TEXT */
```

在这些说明中,public_name与BGIOBJ转换时所用的<public_name>匹配。GRAPHICS.H嵌入文件中包含了缺省驱动程序和字体文件公用名的说明。如果使用缺省公用名,则不必像上面描述的那样去说明它们。

在说明之后,须在程序中注册所有驱动程序和字体文件。如果没有使用/F选择项并且不改变缺省段名,则应通过registerbgidriver和registerbgifont进行注册,否则使用registerfarbgidriver和registerfarbgifont。

C.6.4 运行时刻装入驱动程序和字体文件的例子

可以在运行时刻装入图形驱动程序和/或字体文件,并通过 registerfarbgidriver 和/或 registerfarbgifont 函数对它们进行注册。下面的例子说明这种技术(这部分的内容取自 README 文件)。

/* 装入字形文件到内存的例子 */

```
#include <graphics.h>
#include <io.h>
#include <fcntl.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <process.h>
#include <alloc.h>

void main()
{
    void * gothic_fontp;           /* 指向存储器中字形缓冲 */
    int handle;                   /* 用于I/O的文件处理程序 */
    unsigned fsize;               /* 文件(和缓冲区)的大小 */

    int errorcode;
    int graphdriver;
    int graphmode;

    handle = open("GOTH.CHR", O_RDONLY|O_BINARY);    /* 打开字形文件 */
    if (handle == -1)
    { printf("unable to open font file 'GOTH.CHR'\n");
      exit(1);
    }

    fsize = filelength(handle);    /* 找出文件的大小 */
    gothic_fontp = malloc(fsize); /* 缓冲区分配 */
    if (gothic_fontp == NULL)
    { printf("unable to allocate memory for font file 'GOTH.CHR'\n");
      exit(1);
    }

    if (read(handle, gothic_fontp, fsize) != fsize) /* 把字形读入存储器 */
    {
        printf("unable to read font file 'GOTH.CHR'\n");
        exit(1);
    }
    close(handle);    /* 关闭字形文件 */

    if (registerfarbgifont(gothic_fontp) != GOTHIC_FONT) /* 注册字形 */
    {
        printf("unable to register font file 'GOTH.CHR'\n");
        exit(1);
    }
}
```

```

graphdriver = DETECT;          /* 检测和初始图形系统 */
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");

getch();                      /* 打入一健, 结束 */

closegraph();                  /* 关闭图形系统 */

```

附录D 编译出错信息

Turbo C编译程序查出的源程序出错信息分成三类：灾难性错误、一般性错误和警告。

灾难性错误很少出现，它通常指内部的编译错误。当一个灾难性错误出现时，编译立即停止，必须采取适当的措施并重新启动编译。

一般性错误指程序的语法错误，如磁盘、存储器的访问错误，命令行出错等。编译程序将完成现阶段的编译然后停止。编译程序在每个阶段(预处理，语法分析，优化，代码生成)，尽可能多地找出源程序中的错误。

警告并不影响编译进行。警告指出那些值得怀疑的情况，而这些情况本身作为程序的一部分又是合理的。一旦源文件中使用了与机器有关的结构，编译程序也将产生警告信息。

编译程序首先输出这三种信息。然后输出源文件名及编译程序发现出错处的行号。最后输出信息的内容。

下面列出这三类出错信息。对每条出错信息，指出了可能发生的原因及纠正办法。

请注意出错信息处有关行号的一个细节：编译程序仅产生被检查到的信息。因为Turbo C并不限定在正文的某行设置语句。这样，真正产生错误的原因就可能出现在提示行号的前面一行或几行。在下面的信息栏里，我们指出了这样的信息。

D.1 灾难性错误

内部函数的不合法调用

出现这种错误，可能由于在使用一个宏定义的内部函数时，没能正确地调用。一个内部函数是以两个下划线符(--)开始和结束。

不可约表达式树

这种编译错误表示：所指出文件行中的表达式使得代码生成器无法为它生成代码。像这些低劣表达式必须避免使用。如果编译时总遇到这种错误，就应该通知Borland公司。

寄存器分配失败

这类错误表示，所指出源程序行中的表达式太复杂，编译程序的代码生成器不能为它生成代码。这时，应简化这种繁琐的表达式，如果这样还不行，就要避免使用它。如果编译时总遇到这种错误，应通知Borland公司。

D.2 一般性错误

运算符#后没有跟以宏变元名

在宏定义中，#可以用来标识一个宏变元是串。“#”后必须跟一个宏变元名。

‘×××××’不是函数参量

源程序中将该标识符定义为一个函数参量，但此标识符没有出现在函数参数表里。

歧义的符号‘×××××××’

两个或多个结构的某域(结构分量)名相同，但所具有的位移、类型不同，若某些变量或表达式中引用这些结构分量而未带结构名时，将会产生歧义。这时，需改某个域名，或附上结构名。

参量#名丢失

参量名已脱离用于定义函数的函数原型。如果函数以一个函数原型定义，此原型必须包含所有参数名。

参量表出现语法错误

一组函数调用的参量必须以空格隔开，并以一个右括号结束。若源文件含有一个其后不是逗号也不是右括号的参量，则出现此错误。

数组的分隔符]丢失

源文件定义了一个数组，此数组没有以一个右方括号为终止分隔符。

数组长度太大

定义的数组太大,致使存储容量不够分配。

汇编语句太长

直接插入的汇编语句最长不能超过480字节。

配置文件不正确

TURBOC.CFG文件含有不适合命令行选择项的非注解文字。配置文件命令选择项必须以 一个短横线(-)开始。

包括指令中文件名格式不正确

包括文件名必须用引号("filename.h")或尖括号(<filename.h>)括起来。若文件名无引号或尖括号,则出现这种错误。如使用了一个宏,则产生的扩展的程序文本也是不正确的(因为没有加上引号)。

ifdef指令语法错

#ifdef指令必须以单个标识符(仅此一个)作为该指令的体。

under指令语法错

#under指令必须以单个标识符(仅此一个)作为该指令的体。

位字段长语法错

一个位字段必须是1~16位的常量表达式。

调用未定义函数

正被调用的函数无定义。一般是由于不正确的函数说明或函数名拼写错引起的。

不能修改一个常量对象

这是指对定义为常量的对象进行不合法操作。比如常量的赋值运算。

Case出现在switch外面

编译程序发现case语句位于switch语句外面。这通常是由于括号不配对引起的。

Case语句漏掉,

case语句必须含有一个以冒号终结的常量表达式。可能是丢了冒号,或在冒号前面多了别的符号。

Cast语法错误

cast包含了一些不正确符号。

字符常量太长

字符常量只能是一个或两个字符长。

复合语句漏掉}

编译程序扫描到源文件结束时,发现没有结束标记。这大多是由于花括号不配对引起的。

冲突的类型修饰符

对于同一个指针,只能指定一种变址修饰符,对一个函数,也只能给出一种语言修饰符(cdecl,pascal或interrupt)。例如,当对同一个指针包含两个关键字(near和far),就会出现这种错误。

常量表达式请求

数组的大小必须是常量。这种错误通常是由于一个#define常量的拼写出错引起的。

找不到文件'××××××××'

编译程序找不到命令行给出的文件。

遗漏说明

源文件包含了一个struct或者union域说明。但说明时,后面漏掉了分号(;)。

说明需要给出类型或存储类

说明必须包含一个类型或一个存储类。这表示下面的一个语句是不正确的:

i, j;

说明出现语法错误

源文件中,某个说明丢失了某些符号或有多余的符号。

Default出现于switch之外

编译程序发现default语句出现在switch语句之外。这通常由于括号不匹配引起。

Define指令须有一个标识符

#define后面的第一个非空格符必须是一个标识符,若编译程序发现了一些其他字符,则出现这种错误。

除数为零

源程序的常量表达式中, 出现除数为零的错误。

DO语句必须包含while

源文件do语句中漏掉while关键字。

DO-while语句漏掉了(

在do语句中, 编译程序查出while关键字后无左括号。

DO-while语句漏掉了)

在do语句中, 编译程序查出条件表达式的后面无右括号。

DO-while语句漏掉了;

在do语句的条件表达中, 编译程序发现右括号后面无分号。

Case的情况值不唯一

switch语句的每一个case必须有一个唯一的常量表达式值。

Enum语法错

enum说明的标识符表的格式不对。

枚举常量语法错

enum值应是常量, 若给予enum类型变量的表达式的值不为常量, 则出现这种错误。

Error指令: x x x x

处理文件中有#error指令时, 就显示该指令定义的信息。

写输出文件出错

工作磁盘无空间时此错误就会出现。出现该错误时, 尽量删除工作盘上不必要的文件并重新启动编译。

表达式语法错

当编译程序分析一个表达式并遇到一些严重错误时, 就指出所有出错信息。这些错误常常由于两个连续运算符, 括号不匹配或缺少括号, 以及前面的句中漏写了分号而引起的。

调用时出现多余参数

调用函数时, 其参数个数超过该函数定义中参数的个数。

对 x x x x x x x x 调用时出现多余参数

调用一个指定的函数时(此函数由原型定义), 在调用中出现了过多的参数。

文件名太长

#include指令给出的文件名太长, 编译程序不能进行处理。在DOS中的文件名应不超过64个字符。

For语句漏掉了(

在for语句中, 编译程序发现for关键词后缺少左括号。

For语句缺少)

在for语句中, 编译程序发现在控制表达式后缺少右括号。

For语句缺少;

在for语句中, 编译程序发现在某个表达式后缺少分号。

函数调用时缺少)

函数调用的参量表有几种语法错误, 如左括号丢失或括号不配对。

函数定义位置出错

函数定义不可以出现在另一个函数内。函数内的任何说明, 只要以类似于带有一个参量表的函数开始, 则被认为是一个函数定义。

函数的参数个数必须确定

源文件中的某个函数内使用了va_start宏, 此函数不能接受可变数量的参量。

Goto语句缺少标号

goto关键字后必须有一个标识符。

If语句缺少(

在if语句中, 编译程序发现if关键字后缺少左括号。

If语句缺少)

在if语句中, 编译程序发现测试表达式后缺少右括号。

非法字符/C/(0 x x x)

编译程序发现输入文件中有一些非法字符。以十六进制形式打印该非法字符。

非法初始化

初始化必须是常量表达式，或是一个全局变量extern或static的地址加减一个常量。

非法八进制数

编译程序发现一个八进制常数包含了非八进制数字(例如8或9)。

非法指针相减

这是由于试图以一个非指针变量减去一个指针变量而引起的。

非法结构操作

结构只可以使用点(·)，取地址(&)及赋值(=)运算符，或者作为函数的参数传递。当编译程序发现结构使用了其他运算符时，出现此错误。

非法浮点运算

浮点运算分量不允许出现在移位运算符、按位逻辑运算符、条件(?:)、间接(*)以及其他一些运算符中。编译程序发现上述运算符使用了浮点运算分量时，则产生出错信息。

指针使用不合法

施于指针的运算符仅可以是加、减、赋值、比较、间接(*)或箭头(→)。若对指针使用其他运算符则出现这种错误。

Typedef符号的使用不恰当

源文件使用了一个typedef符号，符号处的变量应该出现在一个表达式中。检查一下此符号说明及可能的拼写错误。

不允许的内部汇编成份

源文件含有直接插入汇编语言语句，若在集成开发环境下进行编译，则产生该错误。必须使用TCC命令行编译此源文件。

存储类别不相容

源文件的一个函数定义中使用了extern关键字。仅static(或者根本没有存储类型)是允许的。

不相容类型转换

源文件试图把一种类型转换为另一类型，但此两种类型是不可相互转换的。例如，函数与非函数的转换，一种结构或数组与一种标准类型的转换，以及一个浮点数与指针的转换。

不正确的命令行参数: xxxxxxxx

编译程序把此命令行参数看成是非法的。

不正确的配置文件参数: xxxxxxxx

编译程序把此配置文件参数看作非法的。检查前面的短横线("-")。

数据格式错

编译程序发现在十六进制中出现十进制小数点。

default的使用错

编译程序发现default关键字后缺少冒号。

初始化语法错误

初始化过程缺少或多出了运算符，括号不匹配或其他不正常情况。

间接运算符错

间接运算符(*)要求非空指针作为运算分量。

宏参量分隔符错

在宏定义中，参量必须以逗号分隔。编译程序发现在参量名后面有其他非法字符，则指出该错误。

非法指针相加

源程序中试图把两个指针相加。

运算符使用错

标识符必须紧跟在箭头运算符(→)后面。

点用法错

标识符必须紧跟在点运算符(·)之后。

赋值请求

赋值号的左边必须是一个地址表达式，包括数值变量，指针变量，结构引用域，间接指针以及数组分量。

宏参量语法错误

宏定义中的参数必须是一个标识符。编译程序发现所期望的参量不为标识符的字符，则指出该类错误。

宏扩展太长

一个宏不能扩展超过4096个字符。当宏递归扩展自身时常出现此错误。宏不能对自身进行扩展。

给出一个输出文件名可能仅编译一个文件

使用命令行选择项时，只允许一个输出文件名。这样，仅编译第一个文件，而其他文件被忽略。

(函数)定义中参数个数不匹配

(函数)定义中的参数和函数原型中提供的信息不匹配。

中断语句位置错

编译程序发现break语句在switch语句或循环结构之外。

Continue语句位置错

编译程序发现continue语句在switch语句或循环结构之外。

十进制小数点错

编译时发现浮点常数的指数部分有一个十进制小数点。

else位置错

编译程序发现else语句缺少与之匹配的if语句，该错误的产生，除了由于else系多余外，还可能由于多余的分号，漏写了花括号，或前面的if语句出现语法错误引起的。

elif指令位置错

编译程序找不到与# elif指令匹配的# if，# ifdef或# ifndef指令。

endif指令位置错

编译程序发现一个# endif指令，但找不到与之匹配的# if，# ifdef或# ifndef指令。

else指令放错位置

编译程序发现一个# else指令，但找不到与之匹配的# if，# ifdef或# ifndef指令。

地址运算符&用于一个不可编址的对象

运算符&作用于一个不可编址的对象，如一个寄存器变量。

地址运算符&作用于不可编址的表达式

源文件对不可编址的表达式使用了地址运算符&，如对一个寄存器变量表达式。

无文件名终止符

在# include语句中，文件名缺少正确的闭引号(")或尖括号(>)。

未给出文件名

Turbo C编译命令(TCC)中没有包含文件名。

对不可移植指针赋值

源文件中把一个指针赋值给一非指针，或反之。作为一个特例，把常量零赋值给一个指针是允许的。如果比较恰当，可以强行抑制此错误信息。

不可移植指针比较

源程序将一个指针和一个非指针进行比较(常量零除外)。只要比较恰当，应该强行抑制错误信息。

不可移植返回类型转换

在返回语句中的表达式类型与函数说明中的类型不同。但如果函数或返回表达式是一个指针，则可以进行转换。在此情形下，返回指针的函数可能送回一个常量零，而零可转变为一个适当的指针值。

非许可类型使用

源文件中说明了几种禁止的类型，例如一个函数返回一个函数或数组。

内存不够

所有工作内存耗尽。应当把文件放到一台有较大存储器的机器去执行。若现在的机器已有640K，可以对源文件进行简化。

指针须位于→左边

→左边必须是一指针。

'××××××××'重定义

此标识符已定义过。

结构或数组大小不定

有些表达式(如sizeof或存储说明)中出现一个未定义的结构或一个空长度数组。如果结构长度不必要,在定义之前就可以引用。如果数组不申请存储空间或者初始化时给定了长度,那么数组就可以定义为空长。

语句缺少;

编译程序发现一个表达式语句后面没有分号。

结构或联合语法错误

编译程序发现在struct或union关键词后面没有标识符或花括号({})。

结构太长

源文件中定义了一个结构,其中申请的存储区域太大以致存储空间不够。

下标缺少]

编译程序发现一个下标表达式缺少闭括号]。这可能是由于遗漏或多写运算符,或括号不匹配引起的。

Switch语句缺少(

在switch语句中,关键词switch后面缺少左括号。

Switch语句缺少)

在switch语句中,测试表达式后面缺少右括号。

函数调用时的参量太少

对带有原型的函数调用时(通过一个函数指针)参量太少。函数原型要求给出所有参量。

对‘××××××××’的调用参量太少

调用指定的函数时(此函数用一原型说明)给出的参量太少。

Cases太多

Switch语句最多只能有257个Cases。

十进制小数点太多

编译时发现一个浮点常量带有不止一个十进制小数点。

default 太多

编译时发现switch语句中不止一个default语句。

阶码太多

编译时发现在一个浮点常量中,不止一个阶码。

初始化太多

编译时发现初始化比说明所允许的要多。

说明中存储类太多

一个说明只允许有一个存储类。

说明中类型太多

一个说明只允许有一种下列基本类型:

char, int, float, double, struct, union, enum或typedef-名。

函数中自动存储太多

当前函数说明的自动存储超过了可用的存储器空间。

文件定义的代码太多

当前源文件中函数的总长度超过了64K字节。可以移出不必要的代码,或把源文件分块。

文件中定义的全局数据太多

全局数据说明的总数超过了64K字节,检查一下一些数组的说明是否太大。如果所有的说明都是必要的,考虑重新组织程序。

两个连续点

一个省略号包括三个点(...),而十进制小数点和选择运算符使用一个点(.),因此在C程序中出现两个连续点是不允许的。

参数#类型不匹配

通过一个指针访问已由原型说明的函数时,给定参数#N(从左到右N逐个加1而计)不能转换为已说明的参数类型。

调用‘××××××××’时参数#类型不匹配

源文件中通过一个原型说明了指定的函数,而给定的参数#N(从左到右N逐个加1而计)不能转换

为已说明的参数类型。

参数‘××××××××’类型不匹配

源文件中通过一个原型说明了可由函数调用的函数，而此指定的参数不能转换为已说明的参数类型。

调用‘yyyyyyyy’时参数‘××××××××’类型不匹配

源文件中通过一个原型说明了指定的函数，而指定的参数不能转换为另一个已说明的参数类型。

重定义类型不匹配

源文件中把一个已说明过的变量重新说明成另一种类型。如果一个函数被调用，而后又被说明成非整型也会产生此错。发生这种情况时，必须在第一次调用函数前，给函数加上extern说明。

不能生成输出文件‘××××××××.×××’

当工作软盘已满或有写保护时，此错误就发生。如果软盘已满，则删除不必要的文件并重新启动编译。如果软盘有写保护，则把源文件转移到一个可写的软盘上并重新启动编译。

不能生成turbo.ink

编译程序不能生成临时文件TURBOC.\$LN，这是因为它不能存取磁盘或者磁盘已满。

不能执行命令‘××××××××’

找不到TLINK或MASM，或者磁盘出错。

不能打开包括文件‘××××××××.×××’

编译程序找不到此文件，这可能是由于一个#include文件包含它本身而引起的，检查此文件是否存在。

不能打开输入文件‘××××××××.×××’

当源文件找不到时就给出此错误。检查文件名有无拼写错误，或检查相应的软盘或目录中是否有此文件。

标号‘××××××××’未定义

函数中出现在goto后的标号无定义。

结构‘××××××××’未定义

源文件中某些行使用了某个结构，但此结构未经说明，这可能是由于结构名拼写错或缺少结构说明而引起的。

符号‘××××××××’无定义

该标识符无定义。这可能是由于说明时或使用处拼写出错引起的，也可能是标识符说明错误引起的。

源文件在#行开始的注解中突然结束

源文件在某个注解中结束。这通常是由于注解结尾标记(*/)丢失引起的。

源文件在#行开始的语句中突然结束

在编译程序遇到#endif之前源程序突然中断，这是由于#endif丢失或拼写错引起的。

预处理指令：‘×××’不认识

在某行开始，编译程序碰到一个#字符，但#后的指令名不是下列之一：define，undef，line，if，ifdef，ifndef，include，或endif。

无终止符常量

编译程序遇到一个无配对的省略符。

无终止的串

编译时发现一个无配对的引号。

无终止的串或字符常量

在串或字符常量开始后编译程序发现无终止符。

用户中断

在集成开发环境里进行编译或连接时，用户按了Ctrl-Break键。

while语句缺少(

在while语句中，关键字while后缺少左括号。

while语句缺少)

在while语句中，测试表达式之后缺少右括号。

调用‘××××××××’时参量数目错

源文件中调用某个宏时,参量数目出错。

D.3 警告

说明了‘××××××’但未使用

源文件中说明了该变量,但并未使用。当编译程序遇到复合语句或函数的结束处括号时,就发出警告。

‘××××××××’被赋以一个不使用的值

该变量出现在一个赋值语句里,但直到函数结束都未曾使用过。当编译遇到结束的闭括号时,就发出警告。

‘××××××××’不是结构的部分

出现在点(·)或箭头(→)左边的域名不是结构的部分。或者(·)的左边不是结构,箭头(→)的左边不是指向结构。

歧义运算符需要括号

当两个位移、关系或、按位逻辑运算符在一起使用而不加括号时,就发出警告。而且当一个加法或减法运算符不加括号与一位移运算符出现在一处时,也将产生警告。程序员总是混淆这些运算符的优先级。因为这些运算符的优先级不太直观。

既用返回又用返回值

当编译程序遇到一个与原先定义的return语句不一致的returu语句时,就发出警告。当一个函数仅仅在一些return语句中返回值时,一般会产生错误。

调用无原型函数

如果“原型请求”警告可用,就显示此信息。表示调用了—个没有先给出函数原型的函数。

调用无原型函数‘××××’

如果“原型请求”警告可用,且调用了—个没有先给出原型的函数‘××××’,此信息就显示。

代码无效

当编译程序遇到一个含有无效运算符的语句时,就发出警告,例如语句a+b;中无有效变量,无需运算,且可能引出一个错误。

常量是long

编译程序遇到一个十进制常量大于32767,或者一个八进制常量大于65535(常量后面无字母l或L),则此常量当作一个long处理。

比较时常量越界

源文件中含有一个比较,其中一个常量子表达式越出了另一个子表达式类型所允许的范围。例如,一个无符号量跟-1比较就没意义。为了得到一个大于32767(十进制)的无符号常量,应该在常量前加上unsigned(如(unsigned)65535),或者在常量后加上字母u或U(如65535u)。

转换可能丢失有效数字

在赋值运算符或其他情况下,源程序要求把long或unsigned long类型转变成int或unsigned int类型。在某些机器上,因为int型或long类变量具有相同长度,这种转换可能改变程序输出的特性。

不论此信息何时给出,编译程序仍将产生代码去做比较。如果代码比较后总是给出同样结果,比如一个字符表达式与4000比较,则代码总执行测试。这还表示一个无符号表达式与-1比较会有些用处,因为在8086机器上一个无符号表达式与-1有相同的位模式。

函数应该返回一个值

源文件说明的当前函数的返回类型既非int也非void,但编译程序未遇返回值。这是常见的一些错误,int函数应该避免使用,因为在C语言旧版里,没有void类型来指出函数不返回值。

混淆signed和unsigned char字符指针

没有通过一个显式模型,就把一个字符指针转变为无符号指针,或反之(严格说,这是不正确的,但在8086机器上,此亦无妨)。

函数‘××××××××’无定义

如果调用了—个没有预先说明的函数,且“说明请求”警告可用,则给出此信息。函数说明可以是传统的,也可是现代的(原型)风格。

不可移植指针赋值

源文件把一个指针赋给另一非指针, 或把一非指针赋给一指针。作为特例, 把一个常量零赋给一指针是可以的。如果是恰当的, 那么应该强行抑制这种警告。

不可移植指针比较

源文件将一个指针和另一非指针(非常量零)作比较。如果可行的话, 应该强行抑制这种警告。

不可移植返回类型转换

return语句中的表达式类型和函数说明的类型一致。作为例外, 如果函数或返回表达式是一个指针, 这是允许的。在此情况下返回指针的函数可能返回一个常量零, 此常量零将转变为一个适当的指针值。

参数‘×××××××’从未使用

函数说明中的该参数在函数体里从未使用, 这不一定是错误。在函数体内, 如果此标识符被重新定义为一个自动变量, 也将发出警告。此参数被标识为一个自动变量但未曾使用。

在定义‘×××××××’前就可能已使用

对该变量赋值之前, 在源文件的某个表达式中使用了该变量。编译程序将对源文件进行简单扫描以确定此条件。如果该变量出现的物理位置在它赋值之前, 就会产生警告。当然程序的实际流程可能在程序使用它之前就已赋值。

可能不正确赋值

当编译程序遇到赋值号作为一个条件表达式(如if, while, do-while语句的部分)的主运算符时, 就产生警告。通常的情况是, 由于输入把赋值号当作等号了。如果希望禁止此警告, 可把赋值语用括号括起来, 并且把它与零显式比较。于是:

```
if (a = b) ...
```

可以改写成

```
if ((a = b) != 0
```

‘×××××××’的重定义不相同

源文件对该宏重定义时, 使用的正文与第一次定义它时不尽相同, 那么新的正文将代替旧正文。

用汇编重新启动编译

编译程序遇到一个未使用命令行选择项-B或#pragma inline语句的asm。通过汇编重新启动编译。

结构按值传送

如果“结构按值传送”警告信息可用, 则该警告信息在结构作为参数按值传送时即产生。这通常是在编制程序时, 把结构作为参数传递, 而又丢失了地址运算符(&)。由于结构可以按值传送, 故此遗漏是可接受的。该警告信息在于提示这种错误。

&出现在函数或数组中

地址运算符(&)对一个数组或函数是不必要的。这样的运算符应该除去。

可疑指针转换

编译程序遇到一些指针转换, 这些转换引起指针指向不同的类型。如果转换恰当, 应该强行抑制该警告。

结构‘×××××××’无定义

源文件使用了该结构, 但没有定义之, 这可能是由于结构名拼写错误或忘记定义引起的。

不认识的汇编指令

编译程序发现在插入的汇编语句中使用了一个不允许的操作码。检查此操作码的拼写, 并查看允许的操作码表, 看此指令是否可接受。

不可达代码

break, continue, goto或return语句后没有跟标号或循环函数的结束符。编译程序使用一个常量测试条件来检查while, do和for循环, 并试图知道循环没有失败。

void函数不可以有返回值

源文件中的当前函数说明为void, 但编译时遇到一个带值的返回语句, 该返回语句的值将被忽略。

零长度结构

源文件中定义了一个总长度为零的结构。对此结构的任何使用都是错误的。

附录E Turbo C 2.0集成调试环境的使用

Turbo C 2.0除了对原有的一些功能作了改进外,还新增加了若干新的功能。其中最显著的特征是,提供了一个集成调试环境,可以单步执行和跟踪程序代码,设置断点,观察和求表达式的值。本附录主要介绍如何利用集成调试环境来调试程序。

一般来说,即使改正了程序中的编译和连接错误,程序也未必能正确地工作。一个新的程序几乎总是包含着许多需要识别和排除的设计和编码产生的错误。发现并定位程序中错误的过程称为调试。

仅仅靠观察非正确程序的行为是很难找到这些错误的,因此大多数程序员借助调试程序对程序中的错误进行定位。调试程序是这样的一种软件,它能够控制程序的运行,可以在任一点停止程序的执行,每次执行一条语句,观察程序处理的数据。

现在的Turbo C 2.0的集成环境中包含了一个调试程序称为集成调试程序。本附录将用一些例子来说明如何使用集成调试程序,第一个例子说明如何使用调试程序的最简单的性能来识别一个“容易的”错误,进一步的例子将说明调试程序的高级特性。

然后介绍调试程序的菜单命令和对应的热键或热键组合以及每个命令的说明。

最后,将给出一些能使调试更容易的一些准则,这些准则关心的是如何设计和书写一个程序,以及如何调试它。可以把这些思想应用于任一种计算机语言,而不仅仅是Turbo C。

E.1 集成调试程序如何工作?

Turbo C的集成调试程序是一个源程序级的调试程序,这意味着可以使用书写程序的同样“语言”来控制调试程序。例如,可以通过让调试程序显示下列表达式的值来显示一个数组中的某个元素的值,

```
nptr→image [ nptr+0x80 ]
```

只需通过Run/Run菜单选择项(热键:Ctrl-F9)调试程序,如果程序被编译时Source Debugging开关已打开,则调试程序就把程序接管过去。(为了设置该开关,可选Debug/Source Debugging.)

在运行程序前,可在源文件的一至多行设置断点,当运行的程序遇到断点时,它就在执行断点所在行的第一个语句之前挂起,并把调试程序的控制权交给程序员。

在程序暂停时,可用许多方法来研究和处理它。例如,可以:

- 显示一个变量或表达式的值
- 在一个特定窗口设置一些表达式并观察其值的变化
- 改变变量的值
- 清除现有的断点或者设置新断点
- 一次执行一行程序
- 编辑文件,重新编译和连接程序,或者使用Turbo C菜单窗口的其它特性
- 让程序继续执行,直到遇到下一个断点

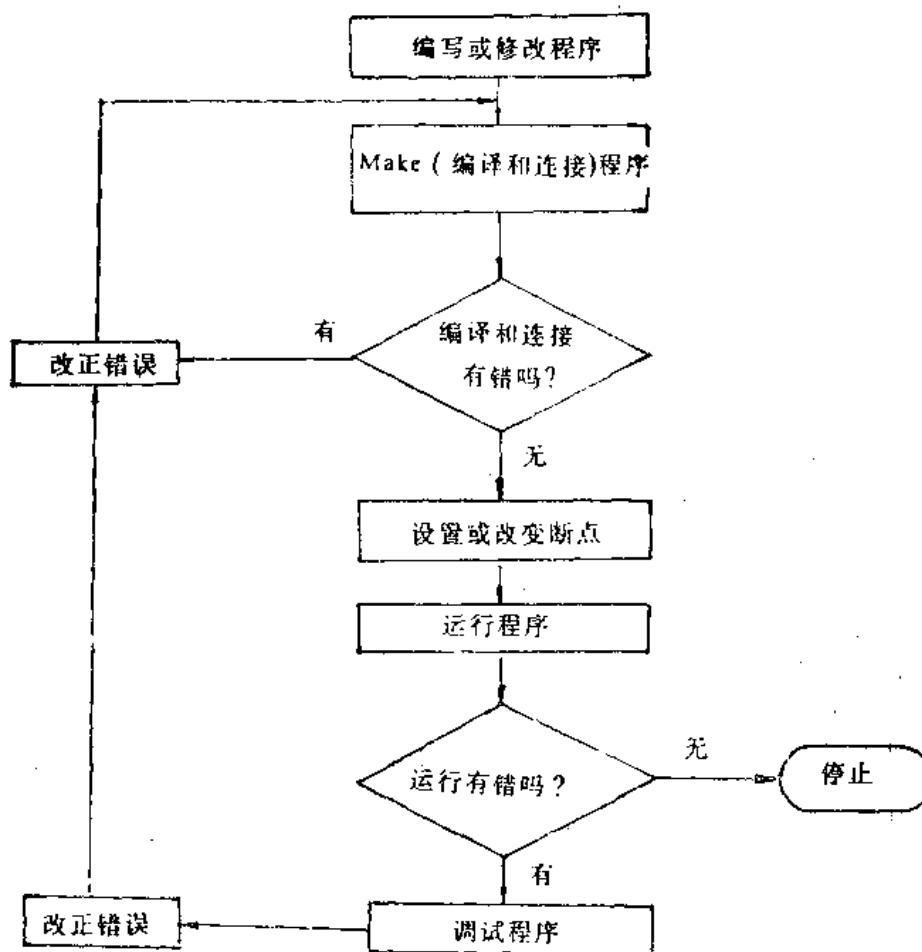
图E.1说明了调试阶段的典型过程。(注意它并没有显示在集成调试程序中每一步所做的事。)

E.1.1 练习1: 调试简单程序

在第一次使用Turbo C进行调试时,可以使用下例给出的程序,该程序称为WORDCNT,意思是显示一个正文文件的内容并将词的长度造表,即报告只有一个字符的词有多少,两个字符长的词有多少等等。不幸的是,WORDCNT包含了几个错误,需要使用调试程序来找到这些错误。

可在配书软盘中的文件WORDCNT.C里找到WORDCNT将它拷贝进用户的Turbo C目录,这样就有了一个最新的含有错误的版本。

如果不在Turbo C目录下工作,在做WORDCNT.C的工作副本的同时,还要拷贝工程文件WORDCNT.PRJ和数据文件WORDCNT.DAT。这三个文件在配置盘和Turbo C目录下都有。



图E.1 调试过程的典型步骤

```

/*****
 * WORDCNT. C - 用于指导调试的样本程序
 *
 * 注意 本程序是作调试教学用的, 它故意包含了一些错误
 *****/
#include <stdio. h>
#include <ctype. h>
#define MAXWORDLEN 16
#define NUL ((char)0)
#define SPACE ((char)0x20)
/*****
 * 寻找行缓冲区中的下一个单词。
 * 输入: wordptr 指向单词的第一个字符或前面的空格。
 * 输出: 指向单词第一个字符的指针, 如果没有更多的单词了, 则为指向结束符 NUL 的指针。
 *****/
char *nextword (char *wordptr)
{

```

```

/* Advance to the first non-space. */
while ( *wordptr == SPACE )
    wordptr++;
return ( wordptr );
}

/* ****
 * 找出一个单词的长度。单词定义为一个以空格或 NUL 结束的字符序列。
 * 输入： wordptr 指向一个单词。
 * 输出： 单词的长度。
 * ****/
int wordlen ( char *wordptr )
{
    char *wordlimit;
    wordlimit = wordptr;
    while ( *wordlimit & *wordlimit != SPACE )
        wordlimit++;
    return ( wordlimit - wordptr );
}

/* ****
 * 主函数。
 * ****/
void main ( void )
{
    FILE *infile;          /* 输入文件。 */
    char linebfr [ 1024 ], /* 输入文件缓冲区,足够长。 */
        *wordptr;          /* linebfr 中指向下一个单词的指针 */
    int i;                  /* 临时变量 */
    static int wordlenent [ MAXWORDLEN ],
        /* 从单元1到MAXWORDLEN统计单词的长度,不使用单元0,这是一个静态数组,
           因此这个单元在运行时不必置0。 */
        /*
           overlenent;      /* 此处统计超长单词 */

    printf ( "WARNING: This is an example program for the practice\n" );
    printf ( "debugging session. If you are not running this under the. \n" );
    printf ( "Integrated Development Environment press control-break now. \n" );
    printf ( "See the debugging chapter of the User's Guide for details. \n\n" );
    printf ( "Enter the input file's name: " );
    gets ( linebfr );
    if ( ! strlen ( linebfr ) ) {
        printf ( "You must specify an input file name! \n" );
        exit ( );
    }
    infile = fopen ( linebfr, "r" );
    if ( ! infile ) {
        printf ( "Can't open input file! \n" );
        exit ( );
    }
    /* 每次循环处理一行。

```

注意：如果一行比输入缓冲区长，程序将产生无效的结果。很长的缓冲区使这种现象不太可能出现。

```
*/
while ( fgets( linebfr, sizeof( linebfr ), infile ) ) {
    printf( "%s\n", linebfr );
    /* 检查缓冲区溢出并删去换行符 */
    i = strlen( linebfr );
    if ( linebfr[i-1] != '\n' )
        printf( "Overlength line beginning\n\t%70s\n", linebfr );
    else
        linebfr[i-1] = NUL;
    /* lineptr 指向 linebfr 中的第一个单词（忽略前面的空格）。 */
    wordptr = nextword( linebfr );
    /* 每次循环处理一个单词。当 [nextword] 返回表示没有更多单词的NULL时，循环结束。 */
    while ( *wordptr ) {
        /* 找出每个单词的长度，增加长度计数数组相应单元的值，并指向单词后的空格 */
        i = wordlen( wordptr );
        if ( i > MAXWORDLEN )
            overlenct++;
        else
            wordlenct[i]++;
        wordptr += i;
        /* Find the next word ( if any ). */
        wordptr = nextword( wordptr );
    }
}
/* 打印单词长度统计。每次循环打印一个值 */
printf( "Length Count\n" );
for ( i = 1; i < MAXWORDLEN; i++ )
    printf( " %5d %5d\n", i, wordlenct[i] );
printf( "Greater %5d\n", overlenct );
/* 关闭文件并退出 */
fclose( infile );
}
```

注意开关Debug/Source和O/C/C/Obj Debug Information都要设为On。在DOS提示符下键入TC WORDCNT以启动TC。构造该程序（选择Compile/Build All）。Turbo C将编译和连接WORDCNT。准备执行，然后暂停。现在选择Run/Trace Into（或者按F7）。

此时，调试程序将main的开始行滚到Edit窗口中。包含main函数说明（void main（void））的那一行被加亮，指出这是运行WORDCNT时要执行的第一行。这个亮条称为执行条，它标志着执行位置；包含下一条执行语句的行。

选择RUN/RUN让WORDCNT运行，WORDCNT提示用户键入输入文件的名字。键入WORDCNT.DAT并按Enter键。WORDCNT将读出和显示数据文件的第一行，然后就锁起了机器。这是由于程序中的错误所致。键入Ctrl-Break解冻并按Esc确认。然后选择RUN/Program Reset（或按热键：Ctrl-F2）来结束调试阶段并按F7（Run/Trace Into）重新开始。

E.1.2 设置和使用断点

如源文件的注释所述，main的第一部分提示输入文件名并打开文件。WORDCNT读出并显示输入文件第一行的事实有力地证明该部分的代码工作良好，起码不会导致我们所观察到的错误。因此，可以越过main的第一部分并在可疑的部

位停下来。为此，需在要停止的那一行设置一个断点。

用PgDn和向下箭头将光标移到以while (fgets (..... 开始的那一行 (它就在注释：每次循环处理.....的下面一行)。注意执行条并不移动。这是因为此时并没有执行程序中的语句，而只是移动编辑程序光标而已。

可以选择Break/Watch/Toggle Breakpoint (或者热键组合Ctrl-F8)在光标所在行设置断点。调试程序将该行加亮为指示在那里设置了一个断点。注意断点的亮条和执行条的样子是不同的。

现在选择RUN/RUN (或热键组合Ctrl-F9)，WORDCNT将继续运行 (此时为开始运行)。

WORDCNT提示输入数据文件名后，它就暂停并等待键盘输入。调试程序通过显示执行屏幕来告诉程序员WORDCNT需要什么，这就和没有调试程序时程序运行的输出一样。键入WORDCNT、DAT和Enter，WORDCNT将继续运行直到遇到断点，于是它停下来并再次出现编辑窗口。光标和执行条现在都停在while语句上。注意：刚才在while语句设置的断点仍在那里，由于执行条遮住了它故无法看到，但执行条移走后它又会出现的，WORDCNT每次到达这个断点时都要停下来，直到把这个断点关闭。

E.1.3 使用Ctrl-Break

除了可以设置的断点外，在运行时还有一个“即刻”的断点：按Ctrl-Break可以在任意时刻中断程序。当按下Ctrl-Break时，就可以停止程序的运行并回到TC编辑程序，此时执行条停在下一次要执行的行上并准备执行程序的剩余部分。实际上调试程序和DOS、BIOS以及其它服务程序保持联系，因此它能了解当前执行的代码是DOS例程、BIOS例程还是程序本身。按下Ctrl-Break后，调试程序就等待直到运行程序本身，然后就开始单步执行每条机器级指令，直到遇到C源代码行，此时就中断，因此执行条就在代码的那一行上。

如果在调试程序定位并显示源代码行之前又探测到第二个Ctrl-Break，调试程序就立刻结束程序，不再寻找源代码行。此时，程序没有填充任何输出或调用任何出口函数就结束了。(这和通过_exit函数结束很相似)，因此，当程序陷入死循环时应该按两次Ctrl-Break，只按一次Ctrl-Break则不能跳出死循环。

E.1.4 单步执行函数调用

既然已经到了WORDCNT中可能发生错误的一部分，就应更仔细地进行调试。对main的剩余部分每次运行一行，并暂停一下观察是否得到了预期的结果。

选择Run/Step Over可以运行main的一行。调试程序运行while语句并读取输入的第一行。接着把亮条移到包含可执行语句的下一行。再次选择Run/Step Over运行随后对printf的调用。

while语句的运行结果是否是我们所期望的？为了解这一点可从Run菜单选择User Screen，或者按Alt-F5，这条命令显示执行屏幕。可以看到输入的第一行显示在屏幕上，因此可以认为while和printf都工作正常。按任意键回到编辑窗口。

调试程序有一个热键代表Run/Step Over，这就是F8键。现在按F8来执行下一条语句，它计算输入行的长度并将它赋给变量l。

警告：对库函数longjmp的跟踪或单步执行将不会在下一行停止 (因为该函数永不返回)。它将使得程序运行到下个断点或者结束。

E.1.5 计算表达式

可以通过显示l的值来验证赋值语句的结果正确性。

选择Debug/Evaluate，调试程序打开一个包含三个域的上托窗口，我们可以利用这些域的功能：

(1) 计算域：在此可以输入一个要计算和可能要修改的表达式。(注意：如果输入的表达式对表达式域框来说太长，可以使用Right, Left, Home, 和End键来卷动该表达式。)

(2) 结果域：调试程序在此显示表达式的值。

(3) 新值域：可在此输入让表达式以取新值 (任选)。

注意，计算域里包一个单词，这是该域的默认值，它是从编辑窗口将光标处所在的单词拷贝而来。稍后我们再来考虑它的用途。现在它的位置输入表达式l，并按Enter键，调试程序在结果域显示l的值，它是4?，这是正确的。按F10退出菜单系统。

下一组语句检查由fgets读入的行是否以换行符结束。如果不是，输入行太长而无法装入缓冲区，程序将显示一条警告信息。如果该行确实是以换行符结尾则正确，从串中删去换行符，这样它就不会被统计为最后一个单词中的一个字符。

在执行if语句前，显示输入行了解预期结果是很有用的。将编辑窗的光标移到单词linebfr处并且选择Debug/Evaluate

(或热键组合Ctrl-F4), 调试程序在Debug/Evaluate窗口的Evaluate域显示linebfr。按Enter键, 调试程序将显示

To be or not to be, that is the question. \n

\n表示换行符, 这和C源程序里一样。

现在选择F8来运行if语句, 执行条移到else子句, 因此if语句工作正确。运行else子句里的赋值语句, 然后再计算linebfr (可以使用热键Ctrl-F4表示Debug/Evaluate)。换行符(\n)被删掉了。到目前为止, 一切正常。

E.1.6 函数nextword和wordlen

下一条语句调用函数nextword来对串中的下一个单词定位(此时为第一个单词)。用F8键执行这条语句并求wordptr以了解nextword的返回值, 可以看出wordptr指向To be or not to be; that is the question里的T。如果是这样, nextword工作正常, 至少在这个简单的例子里是这样。

接着程序进入一个while循环, 循环的每一次重复处理linebfr中的一个单词, 并把wordptr指向下一个单词, 循环处理完最后一个单词后, wordptr应该指向结束该行的空字符, 然后循环结束。

运行while语句, 执行条移到循环体的第一个语句, 它调用函数wordlen, 该函数确定wordptr指向的单词的长度。运行这条语句并求l的值, l的值为0, 这是不对的: 第一个单词的长度应该为2, 终于找到了一个错误!

E.1.7 停一停并思考

有些程序员可能急于想改正错误, 但先考虑一下它对程序的影响还是值得的。错误的单词长度0有两个作用。首先, 它增加了wordlen的0号单元的值, 这是不对的。其次, 它使语句wordptr += 1没有改变wordptr的值。这使while循环以第一次时的wordptr值开始第二次重复。由于第一次调用wordlen时返回了0, 可以推测第二次它仍然返回0, 因此第三次执行while循环时wordptr仍为原来的值, 第四次仍是这样, 等等, 一直下去仍为如此。这完全符合观察到的WORDCNT的行为。正是这个错误使WORDCNT挂起了机器。

上述研究有什么价值呢? 读者也许会发现这个错误只能解释WORDCNT的一部分错误行为, 或者它与这些错误动作根本无关。此时你也许想更进一步地运行程序看看下面会发生什么事。但是现在要集中力量进行错误定位, 并且要相信程序的行为将得到改正, 或者起码是改善。

E.1.8 亦已完成的工作

已经发现WORDCNT的错误行为是由于wordlen中的一个错误引起的。并且还确切地发现了wordlen出了什么错。等一会我们再回过头来对它进行研究。现在首先花一些时间来复习已经用过的调试程序命令。

在调试WORDCNT的首次努力中, 已经做了下列事情:

- 确保Debug/Source Debugging和O/C/C/OBJ Debug Information 开关都设置为on。
- 选择Compile/Build All为调试WORDCNT做准备。
- 使用编辑程序命令将光标移到WORDCNT的可疑部位; 选择Break/Watch/Toggle Breakpoint来设置一个断点; 选择Run/Run (或热键组合Ctrl-F9) 运行WORDCNT直到遇到断点。
- 使用Run/User Screen或其热键Alt-F5来检查程序在用户屏幕的输出。
- 选Run/Step Over (或按热键F8) 来执行main里的语句, 每次一行。
- 选Debug/Evaluate (或按热键Ctrl-F4) 来显示几个变量的值。
- 思考发现的错误, 推断出它解释了所观察到的WORDCNT的错误行为, 因此需要立刻改正它。

E.1.9 Evaluate窗口的默认表达式

回忆一下Debug/Evaluate将编辑窗口光标处的单词拷贝到Evaluate域。如果在选择Debug/Evaluate之前将光标移到要求值的变量处就能节省工作。即使不是所要求的表达式, 通过编辑默认表达式输入也比从头敲入快得多。更进一步, 可以通过按右箭头将编辑窗里更多的正文拷进默认表达式。每按一次右箭头, 就拷进一个字符。

例如, 假设要计算表达式linebfr[i-1]的值, 它出现在源文件的如下行中:

```
if (linebfr[i-1] != '\n')
```

将光标移到linebfr并选择Debug/Evaluate, 则Evaluate域显示默认表达式linebfr, 按五次右箭头键将[i-1]拼入表达式中, 然后按Enter键。

E.1.10 改变所求表达式的值

Debug/Evaluate可以改变一些表达式的值。它可以改变任何表示单个数据单元的表达式值,如`i, linebfr[i]`或者`*(linebfr+i)`。

试求变量`i`的值并改变它。当按下Enter来求`i`的值时,调试程序将`i`的值显示在结果(Result)域,使用下箭头键移进新值(New Value)域并输入想赋给`i`的值。例如,可在新值域输入`i+1`(即`i`增加1),或者输入17,按下Enter后调试程序计算该输入的值,改变`i`的值并显示新值于结果窗口。(注意:请记住一旦在新值域中按了Enter键,所求变量的值就改变了,即使再按Esc也不能恢复这个改变)按Esc离开Debug/Evaluate,再用这条命令重新显示`i`的值以证实它确实被改变了。然后把它改回来并且再次退出Debug/Evaluate。

可以修改一个表达式值以消除错误的影响,这样便仍能运行程序以找出别的错误。也可使用它来深入了解程序的行为。例如,假设要了解当传递了无效的参数时某个函数的行为。让程序来向该函数传递这些值也许是很困难的,但可以通过在程序调用该函数前改变一些变量的值来达到目的。

若按Esc而不是Enter退出了新值域,调试程序将不改变表达式的值。如果改错了新值或者在修改时改变了想法,就按Esc键。

可以对不包含下列元素的任何合法C表达式求值:

- 函数调用
- 已有定义或已有类型定义的符号或宏
(`*wordptr = 0x20` 正确; `*wordptr = SPACE` 不正确,因为SPACE已有定义)
- 不在可执行的函数范围内的局部或静态变量,除非它们是充分受限的。

E.1.11 受限变量名

有两种典型的情况需要完全限定表达式中所用的变量名:

- 当要检查别的模块里的静态变量时
- 当要查看一个函数里的自动(局部)或静态变量时

为了完全地限定变量名,可使用下列语法:

· <模块名> · <函数名> · <变量名>

注意在一定条件下可以省略模块名或函数名。例如,如果正在单步执行主函数而又想看看在另一个名为`mysubs`的模块里的名为`myvar`的静态变量,可以键入`·mysubs·myvar`。但是如果`myvar`在模块`mysubs`的函数`myfunc`中出现,就要用`·mysubs·myfunc·myvar`另一方面,如果`myfunc`和函数`main`同在一个模块中,则只须键入`·myfunc·myvar`。

E.1.12 格式区分符

为了精确地控制Debug/Evaluate窗口的信息显示,Turbo C允许求值(Evaluate)域的表达式使用可选的格式区分符(Watch窗口也一样)。格式区分符跟在表达式后边,用一个逗号隔开,它既可大写也可小写。

格式区分符包含一个可选的重复计数(一个整数),后随零或多个格式字符。重复计数和格式字符之间不需空格,表1列出了各种格式字符及其作用。

重复计数用于显示连续的变量,典型的例子就是数组中的元素。例如,如果`list`是有10个整数的数组,表达式`list`可以显示为:

```
list: { 10,20,30,40,50,60,70,80,90,100 }
```

如果要查看该数组某一范围内的数,可以指定要观察的第一个元素的索引,再加上重复计数。

```
list[5].3: 60,70,80
```

在外理一行无法全部显示的数组时这项技术很有用。

重复计数并不仅限于数组,任何变量都可以跟随重复计数,通用语法`var,<n>`只是从`var`的地址开始显示`n`个连续的和`var`同类型的变量。但是,如果表达式不是一个变量则忽略重复计数。有个好的经验就是如果某个结构能够合法地出现在赋值语句左部,或者可以作为函数的参数,那么它是一个变量。

对格式区分符有如下规则:

(1) 格式区分符只有和适当类型的变量一起出现才起作用,否则被忽略。

注意,如果要求值的表达式引起显示多个对象(如在结构中),且又给该表达式以多个格式区分符,则将对每个对象分别采用适当的格式区分符。例如,如果输入`Struct F5H`来显示包含整数和实数的结构,则整数显示为16进制,而实数显示为有5位有效数字的浮点格式。

(2) 如果对同种类型的表达式输入了多个格式区分符, 类型的优先级决定使用冲突的区分符中的哪一个, 即选用具有最高优先级的那一种。当然这些问题一般只与结构和联合有关。对简单变量和简单变量的数组, 一般只会输入一种格式区分符。

表1 调试用格式区分符

字 符	作 用																								
C	字符。对控制字符 (ASCII 0 到 31) 显示特殊的显示字符: 例如, 显示一张快乐的笑脸。它对字符和字符串都有效。																								
S	字符串。使用 C 的适当的 Esc 序列显示控制字符 (ASCII 0 到 31) 为其 ASCII 值。这是默认的字符和字符串显示格式, 因此只有和 M 区分符连用时 S 区分符才有效。																								
D	十进制数。所有的整数值都按十进制显示。对简单的整数表达式和包含整数的数组和结构有效。																								
H 或 X	十六进制。所有整数都显示为以 0x 为前缀的十六进制数。对简单整数表达式和包含整数的数组和结构有效。																								
F <n>	浮点数。n 是 2 到 18 之间的整数, 它指定了显示的有效数字的位数。只对浮点值有效。																								
M	内存转储。从表达式给定的地址开始显示内存转储。表达式应为可作为赋值语句左部的有效结构, 即一个指示内存地址的结构; 否则将忽略 M 区分符。变量的每一字节隐含地显示为两个十六进制数。加上 D 区分符将使每个字节显示为十进制数, 而加上 H 或 X 区分符将显示为十六进制。C 或 S 区分符将使变量显示为一个字符串 (有或没有特殊字符), 默认的显示字节数与变量的长度对应, 但可用重复计数来指定确切的字节数。																								
P	指针。显示附有被指向地址等信息的 seg: ofs 格式的指针, 而不是默认的面向硬件的 seg: ofs 格式。具体地说, 它告知段所在的存储区, 它还在正确的位移地址处给出变量的名字, 内存区域如下划分。																								
<table border="1"> <thead> <tr> <th>内存区域</th><th>求值信息</th></tr> </thead> <tbody> <tr> <td>0000: 0000—0000; 03FF</td><td>中断向量表</td></tr> <tr> <td>0000: 0400—0000; 04FF</td><td>BIOS 数据区</td></tr> <tr> <td>0000: 0500—Turbo C</td><td>MSDOS/TSR</td></tr> <tr> <td>Turbo C—User Program PSP</td><td>Turbo C</td></tr> <tr> <td>User Program PSP</td><td>用户程序 PSP</td></tr> <tr> <td>User Program—RAM 顶</td><td>如果某静态用户变量的地址落在变量的内存地址之内则给出其名字, 否则不显示任何东西。</td></tr> <tr> <td>A000: 0000—AFFF; 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—EFFF; FFFF</td><td>EMS 页/适配器的 BIOS ROM</td></tr> <tr> <td>F000: 0000—FFFF; FFFF</td><td>BIOS ROM</td></tr> </tbody> </table>		内存区域	求值信息	0000: 0000—0000; 03FF	中断向量表	0000: 0400—0000; 04FF	BIOS 数据区	0000: 0500—Turbo C	MSDOS/TSR	Turbo C—User Program PSP	Turbo C	User Program PSP	用户程序 PSP	User Program—RAM 顶	如果某静态用户变量的地址落在变量的内存地址之内则给出其名字, 否则不显示任何东西。	A000: 0000—AFFF; 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—EFFF; FFFF	EMS 页/适配器的 BIOS ROM	F000: 0000—FFFF; FFFF	BIOS ROM
内存区域	求值信息																								
0000: 0000—0000; 03FF	中断向量表																								
0000: 0400—0000; 04FF	BIOS 数据区																								
0000: 0500—Turbo C	MSDOS/TSR																								
Turbo C—User Program PSP	Turbo C																								
User Program PSP	用户程序 PSP																								
User Program—RAM 顶	如果某静态用户变量的地址落在变量的内存地址之内则给出其名字, 否则不显示任何东西。																								
A000: 0000—AFFF; 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—EFFF; FFFF	EMS 页/适配器的 BIOS ROM																								
F000: 0000—FFFF; FFFF	BIOS ROM																								
R	结构/联合。显示域名和值, 如 { X: 1, Y: 10, Z: 5 }, 只对结构和联合有效。																								

表2 格式区分符的优先级及默认值

类 型	区分符的优先级顺序	默认值
char	C S H D	S
unsigned char	C S H D	S
int	H D C* S*	D
unsigned int	H D C* S*	D
long	H D C* S*	D
unsigned long	H D C* S*	D
char ptr	C S P H	S
other ptr	P H	P
enum	H D C* S*	D (后随成员名)
float	Fn	F7
double	Fn	F1
long double	Fn	F18
array of char	C S H D	S
other array	由大括弧{ }括起和逗号分隔的元素。	
structure	R	

* 只有值在适当范围之内才可用字符类型区分符
(对signed类型为 - 128到127,对unsigned类型为0到255)

注：对指针变量单独使用H格式区分符将把指针显示为一个十六进制整数值。

为了说明格式区分符的使用方法,假设说明了下列结构和变量：

```
struct {
    int account;
    char name[ 10 ];
} client = { 5000, "Jones" };
int list [ 5 ] = { 0, 10, 20, 30, 40 };
char *ptr = list;
void main ( )
{
}
```

然后在求值域输入下列表达式,将在结果域产生对应的值：

所求表达式	结果
list	{ 0, 10, 20, 30, 40 }
list [2], 3	20, 30, 40
list [2], 3x	0x14, 0x1E, 0x28
list , m	00 00 0A 00 00 14 00 1E 00 28 00
ptr	DS: 0198
ptr, p	DS: 0198 [—list]
*ptr3	0, 10, 20
client	{ 5000, "Jones\0\0\0\0\0" }
client, r	{ account: 5000, name: "Jones\0\0\0\0\0" }

E.2 练习2: 找出wordlen中的错误

现在再回到WORDCNT找出函数wordlen中的错误。

一般来说仅靠研究代码来看看能否找出错误是一个好的想法。这通常是找出错误的 fastest 途径。现在先做下面的练习。

如果正处于练习一的调试状态,就选择Run/Program Reset (或者按热键, Ctrl-F2) 来取消。这使Turbo C释放WORDCNT占用的内存,关闭所打开的文件,并结束WORDCNT当前的运行。

但是,如果完成练习一后离开了集成环境而又想用该环境来运行别的程序,就重新启动并选择工程文件WORDCNT.PRI,然后再在语句while (fgets (... 设置一个断点。

现在选择Run/Run开始新的调试过程。Turbo C又为WORDCNT的执行做好准备, Turbo C让程序一直运行到断点。

在文件名提示下,输入WORDCNT.DAT并按Enter。

选择Run/Step Over运行WORDCNT到调用wordlen的语句,然后选择Run/Trace Into (加用热键F7) 来让调试程序进入wordlen,而不是只运行调用它的语句。

Run/Trace Into每次运行一行程序,这和Run/Step Over相似,但它能进入函数调用而不是忽略掉。此刻,执行条在wordlen的说明部分。

先研究一下wordlen的逻辑,该函数期望一个参数,这是一个指向行缓冲区里一个单词的指针wordptr,将wordptr的值赋给一个局部变量wordlimit,然后while循环增加wordlimit直到遇到结束单词的空格或结束一行的空字符。返回wordlimit和wordptr的差值作为单词的长度。

运行wordlen定义部分的两行到达赋值语句,既可使用Run/Trace Into也可使用Run/Step Over,这是因为wordlen没有调用更低层的函数,因此这两条命令具有同样的效果。

求一下wordlimit指向的串(即输入的第一行)的值,该值是正确的。运行while语句,执行条应该移到下一行增加limit的值,但它却移到了再下一行的返回语句,这也许是要寻找的错误了。

注:求值后不可不必按两次Esc来返回TC编辑程序,可直接使用该操作的热键F10。

考虑一下接着会发生什么事,由于wordlimit没有增加,wordlimit与wordptr的差为0,因此wordlen返回0。这正是要寻找的错误,现在把错误的范围从“wordlen里的某处”缩小为“wordlen里while语句的表达式某处”,现在花时间检查代码,看看能否找到出了什么错仍然是值得的。

如果不能,就试着求错误表达式每一部分的值以了解它们是怎样共同作用的。可以发现*wordlimit的值为ASCII T。由于SPACE为一个未defined符号,因此无法求*wordlimit != SPACE的值;但是可以求*wordlimit != ' ' 的值(因SPACE的值为' ') ,并且其值为1(真),整个表达式的值应该为真,但它却为假。看来操作符&有问题。

&确实是错误的操作符,它是C的“按位与”操作符,它把一个操作数的每一位和另一个操作数的对应位进行“与”操作,由于*wordlimit != SPACE 总为0或1,则当*wordlimit为偶数时*wordlimit & *wordlimit != SPACE就等于0了。正确的操作符应为==,当两个操作数都非零时给出结果1(试用一个&和两个==来求整个表达式的值,看一看这种区别)。

(如果不是自己找出问题的解也不必灰心,搞混&与==和|与||是C程序设计新手最常犯的错误。当从自己的程序中找到几次这种错误之后,就会很容易学会区别它们了。

E.2.1 改正错误

将&改为==后即改正了错误,可按F2将改正后的程序保存好,以免在下面的调试中因程序出错时把它冲掉。然后再选Run/Run。由于改动了源程序,调试程序将问是否要重新编译和连接,输入Y,对程序进行重新编译和连接。现在,可以调试改正后的程序了。

再找别的错误之前,建议先总结一下刚才所学并更深入地理解所遇到的特征。

E.2.2 已完成的工作

用Run/Program Reset 或热键Ctrl-F2取消了第一次的调试,然后通过设置断点和选择Run/Run (或热键组合Ctrl-F9) 将WORDCNT运行到对wordlen的调用。

用Run/Trace Into (热键F7) 跟踪进入wordlen的调用,单步执行wordlen并找出错误,改正错误,保存源文件,用Run/Run将修改后的程序准备好用于调试。

E.2.3 断点的进一步讨论

如果做完练习一后未离开Turbo C,则在开始第二个练习时在while (fgetc (...) 设的断点仍然存在,这时WORDCNT运行到断点而不是运行到程序结束。可以看到,断点从一次调试期间保留到下一次,即使时而编辑时而重编译和连接程序,依然如此。Turbo C将断点上下移动以将它保留在正确的语句。

即使在程序中设置了断点之后保存该文件,并且编辑别的文件,断点仍然存在。只有在下列情况才会失去断点:

- 离开了集成环境
- 删除源文件中设置断点的行。
- 选择Break/Watch/Clear All Breakpoints清除断点。

但是, Turbo C在下列两种情况下也会失去断点的踪迹:

(1) 如果编辑了一个包含断点的文件,然后又放弃了(没有保存)该文件的编辑版本, Turbo C不能记住原来文件中断点所在的行,因此就会把它们显示在错误的行上。

如果要放弃一个正在编辑的源文件,就选用Break/Watch/Clear All Breakpoints清除所有的断点并重新设立所属的断点。注意Break/Watch/Clear All Breakpoints将清除程序中的所有断点,而不只是正在编辑的源文件中的那些。

(2) 如果编辑了一个文件并继续进行调试,或者没有重新编译和连接就进行新的调试,此时断点实际上还处在正确的位置,但由于源文件和执行文件的长度不匹配,断点条就会在错误的行出现(执行条也出现在错误的行。)

一般来说不太可能遇到这种情况,因为继续或重新开始调试时Turbo C会显示警告提示“源文件已修改过,重建吗?” (“Source modified, rebuild?”)。

在编译源文件前可以将断点设在包含不可执行语句的行,如注解和空行。此时, Turbo C在执行程序前将警告断点设在了包含不可执行代码的源行上了。文件编译后, Turbo C知道哪些行包含可执行语句,要在这些行设置断点Turbo C就会给出警告。

可以选择Break/Watch/View Next Breakpoint将光标移到文件中的下一个断点。注意这条命令是将光标移到当前工程的下一个断点,而不是程序运行时要执行的下一个断点。要清除某些断点而不是全部时可以使用Break/Watch/View Next Breakpoint检查这些断点。

E.3 回到程序

看看改正错误后有什么变化没有。如果完成练习2后离开了集成环境或运行了别的程序,就重新启动集成环境并使WORDCNT.PRJ成为当前的工程。选择Run/Run开始新的调试过程。运行WORDCNT直到调用wordlen: 跟踪下去并单步执行。这回它正确地工作并返回值2,成功了!

但是工作还没完。程序的大部分还没有测试,而它也许还有别的错误。下一件事是测试main里的while循环,最彻底的方法是单步执行main直到输入的第一行全部处理完并且控制离开了循环。验证每一步是否都正确地工作并且控制是否在正确的位置离开循环的。

这样做是否工作量太大了? 调试程序的Watch窗口使这容易得多, Watch窗口显示一个或多个表达式及它们的当前值。每次调试程序暂停时,它就重新计算Watch窗口中每个表达式的值。这样就能在运行程序时观察表达式值的变化。

对内部while循环的所有重要数据元素都要建立观察表达式: 变量i, 以wordptr开始的串以及wordlenent的开头几个元素。

在调试期间Watch窗口通常只占用屏幕的下部,就像编译和连接时的信息(Message)窗一样,最初它只有一行而且是空的。如果看不到Watch窗口,这是由于编辑窗被放大为全屏了;按F5就可以回到分屏环境并重新显示Watch窗口。

选择Break/Watch/Add Watch (或按组合键Ctrl-F7), 调试程序打开一个窗口并提示输入一个观察表达式。和Debug/Evaluate一样, Break/Watch/Add Watch使用编辑窗内光标处的单词作为默认表达式。如果默认表达式不是i, 则输入i, 按Enter键。调试程序将表达式i及其当前值加进Watch窗口。

对五个表达式wordptr和wordlenent[1]到wordlenent[4]重复此过程, 每当加进一个观察表达式, Watch窗口就扩大一行以容纳它。现在单步执行while循环直到WORDCNT处理完输入的第一行。在执行过程中, 可以看到WORDCNT让wordptr每次前进一个词并增加wordlenent中的相应元素。当wordptr到达行尾并且执行条离开内部while循环后停下来看看, 循环工作正常。

E.3.1 编辑和删除观察表达式

可以编辑和删除Watch窗口中的观察表达式,也可以增加表达式。

为编辑和删除观察表达式, 先使 Watch 窗口激活。如果在菜单系统中, 按 F10 离开, 然后按 F6 在编辑窗和观察窗之间进行转换, Watch Editor 用亮条覆盖住将要被编辑或删除的表达式, 可以使用 Home, End, 上下箭头移动亮条。

让我们先来编辑一个观察表达式, 将亮条移到 wordcnt [4] 并选择 Break/Watch/Edit Watch (或按 Enter), 调试程序打开一个包含该观察表达式的窗口, 并提示对它编辑。

将数组下标 4 改为 6 并按 Enter, 调试程序改变 Watch 窗口中的观察表达式并显示新表达式的值。

将亮条移到表达式 wordcnt [3] 并选择 Break/Watch/Delete Watch (或按 Del 或 Ctrl-y), 调试程序就删去这个观察表达式。

按 F6 激活编辑窗口, 注意, 当观察窗口被激活时在有亮条的观察表达式前有一个菱形符号标志, 按 F6 再次激活观察窗。

可以通过选择 Break/Watch/Remove All Watches 一次删除所有的观察表达式 (这样做时不必激活 Watch 窗), 现在删除所有的观察表达式, 于是观察窗又回到了原来的空状态。

再按 F6 激活编辑窗, 每次输入这条命令时它就把活动窗口从编辑窗改为观察窗, 或者相反。

E.3.2 窗口的扩大和转换

调试程序窗口的扩大与转换规则只是所学到的编辑程序, 编译程序和连接程序的规则的扩展。

在调试阶段屏幕通常被分为编辑窗和观察窗, 就象在编译和连接时通常被分为编辑窗和信息窗一样, 要在这两个可见窗口之间进行转换, 可按 F6。

按 F5 可以将活动窗口扩充为全屏, 若活动窗口已经被扩充了, 按 F5 就使屏幕回到分屏显示状态, 不妨对编辑窗和观察窗试试这个用法。

为了显示执行窗口, 可以选择 Run/User Screen, 或者按 Alt-F5, 按任意键就可以回到原来的显示状态。

用 Alt-F6 可以改变窗口中的内容:

- 当编辑窗活动时, 再次装入当前文件的前一文件。
- 如果观察窗或信息窗活动, 按 Alt-F6 就在观察编辑器和信息跟踪窗之间进行转换。

E.3.3 卷滚观察表达式

向观察窗口增加表达式后, 它就逐渐增长直到占据大约半个屏幕, 如果再增加更多的表达式, 有些就要卷出窗口外边了, 可以通过使用 PgUp, PgDn, 上, 下光标键来卷滚来观察这些表达式。

若某个观察表达式太长以至窗口装不下, 可以使用 Home, End, 左, 右光标键来卷滚以观察其头部和尾部。

E.4 练习4: 调试打印循环

对整个读取和处理输入文件的 while 循环作了调试后, 这个循环里有些地方还可能潜伏着错误, 但这要向 WORDCNT 传递一些特殊类型的输入才能找出错误, 如空行等。因此, 先暂不寻找这类错误而继续调试到打印结果的 for 循环。

先运行 WORDCNT 到 for 循环开始的那一行这可以通过将光标移到那一行, 设置一个断点, 然后选择 Run/Run 来办到。这回若希望在某一点只停一次, 可以使用另一种更方便的技术。

将光标移到 for 循环开始的那一行, 然后选择 Run/Goto Cursor (也可使用热键 F4), 调试程序运行 WORDCNT 并在光标所在行停下, (如果它先遇到断点就在断点处停下, 但这部分代码没有设置断点)。

运行一条语句, 执行条移到 for 循环里的 printf 语句, 注意, 此时 for 所在行并没有断点亮条, Run/Go to Cursor 是一次性操作, 它并未设置一个永久断点。

用 Run/Trace Into 单步执行 for 循环, 注意, Run/Trace 并不跟踪进入 printf 函数, 这是因为 printf 并不是在带调试信息编译的源文件中定义的, 调试程序能够运行这些函数, 但不能对它们进行跟踪 (另一个不能跟踪该函数的原因是: printf 的源文件不在磁盘上)。

每次运行包含 printf 调用的行时, 调试程序就换到执行屏幕, 这使 printf 在适当的条件下能够实现其输出, 每次运行包含一个函数的一条语句时调试程序就显示执行窗口, 因为它不能区别哪个函数将写或不写屏幕。

在执行屏幕可见的短时间里无法读取程序的输出, 故可用 Run/User Screen 或 Alt-F5 重新显示它以了解 WORDCNT 在屏幕上写了些什么。

注意: 如果程序的输出复盖了编辑窗里的源程序, 可以选择 Debug/Refresh Display 来刷新屏幕, 然后检查并保证 Debug/Display Swapping 选择项设置为 Smart 或 Always。

这样就完成了第4个练习。现在可来确定for循环中是否有错误，如果有的话又如何改正。

注意：WORDCNT中包含不止一个错误。试找出它们。我们将在稍后的“软件测试指南”一节中提及它。

E.5 练习5：调试大型程序

调试程序还有一些特征可以帮助调试大型源程序和多文件程序。将在下面几节中进行说明。

为了使用将要说明的这些特征，在编译程序时须将Options/Compiler/Code Generation/Standard Stack Frame选择项设为on。检查这个选择项，如果是关闭的，就打开它，然后确认WORDCNT已装入编辑窗并按Alt-F9重新编译。选择Run/Step Over或按相应热键F8开始新的调试过程。

E.5.1 寻找函数的定义

Debug/Find Function将一个函数的定义部分调入编辑窗。它可以找到以Debug/Source Debugging和O/C/C/OBJ Debug Information选择项为on进行编译的程序中的任何函数。

如果要处理程序的某部分时发现了一个错误，但必须通过修改另一部分来改正它时，Debug/Find Function就非常有用。它显示某个函数的代码或有助于理解该函数是如何工作的注解。

为了试试这个命令，先将光标移到main中调用wordlen的那一行。选择Run/Goto Cursor；然后将光标移到名字wordlen并选择Debug/Find Function。调试程序打开一个窗口并提示输入一个函数名；由于wordlen是要找的函数，键入它，然后按Enter。调试程序就在编辑窗口显示wordlen的定义。

E.5.2 调用栈

当调试一个调用多层函数的程序时，有时需要了解调用栈，它能指出程序已调用了哪些函数，以及调用的次序等。这项特征需要将Options/Compiler/Code Generate/Standard Stack Frame选择项设为on。

跟踪到wordlen，然后选择Debug/Call Stack或按相应热键Ctrl-F3。调试程序在一个上托窗口中显示调用栈。现在正在执行的函数在栈顶上，而main在栈底。注意，该调用栈只显示程序已调用的函数名，而不显示它们的参数值。

可以通过将调用栈窗口的亮条移至调用栈中的任何函数并按Enter以显示该函数的当前执行行。例如，要显示main中的当前执行行，将亮条移至调用栈中main的项并按Enter。调试程序将main中包含对wordlen调用的那部分卷入编辑窗口。如果调用栈中还有别的项，可以重选Debug/Call Stack并选择别的调用栈项以显示任何函数的当前执行行。

E.5.3 返回执行位置

在察看程序的任意部分后，也可以使用Debug/Call Stack选择项返回到执行条。为了把执行条重新调入编辑窗，只须选择调用栈最顶部的函数调用。读者不妨一试。

E.6 关于多源文件

所有的调试程序命令都能处理包含多个源文件的程序。例如，如果选择Debug/Find Function寻找一个不在编辑窗的文件中定义的函数，调试程序就把适当的文件调入该窗口。如果对当前窗口内的文件做了改动，调试程序在调入新文件之前要问是否要保存当前文件。

类似地，用Debug/Call Stack选择了调用栈中的某个函数，调试程序就把包含该执行位置的源文件重新装入编辑窗。若修改了别的文件，调试程序在重装入源文件之前提示是否要保存。如果程序中包含许多源文件，最好每次只调试其中几个。程序中同时只有几个部分改动则控制工作就容易得多。

E.7 调试程序命令和热键综述

本附录提供集成调试程序中最常使用的命令。熟练地掌握了调试程序之后，还应学习另外一些命令，这可以参照表3和表4。

许多调试程序命令和其它菜单命令都可以用热键或组合热键引用。为了避免混淆细节，上面我们只提到了最重要的一些。表3给出了学过的调试命令所对应的所有热键。表4列出了另外一些集成调试程序常用的菜单命令。

表3 调试命令与热键 (1)

热键	菜单命令	说 明
F4	Run/Go to Cursor	执行程序至光标所在行。可以用它来开始调试过程。
Ctrl-F2	Run/Program Reset	取消当前的调试过程, 释放分配的空间并关闭文件。只在调试阶段有效。
F7	Run/Trace Into	运行当前函数的下一个语句, 它可以调用以 Debug/Source Debugging 和 O/C/C/Obj Debug Information 为 on 编译的低层函数, 并跟踪进入该函数。它引起调试的开始。
F8	Run/Step Over	运行当前函数的下一语句, 但不跟踪进入调用的语句。它可引发调试阶段。
	O/C/C/Standard Stack Frame	是 Option/Compiler/Code Generation/Standard Stack Frame 选择项的触发开关。如果要让 Debug/Call Stack 选择项正确工作, 则在编译程序应将时该选择项设为 on。
	O/C/C/Obj Debug Information	是 O/C/C/Obj Debug Information 选择项的触发开关。只有在该项设为 on 时编译和连接的源文件才可以进行调试。
Ctrl-F4	Debug/Evaluate	求一个 C 表达式的值; 允许修改变量的值。
	Debug/Find Function	找到一个函数定义并将它显示在编辑窗口。只对调试阶段有效。
Ctrl-F3	Debug/Call Stack	显示调用栈。可以从调用栈中选择函数名来显示一个函数的当前执行行。只对调试阶段有效。
	Debug/Source Debugging	对是否允许调试的控制, 当它设为 on 时, 集成的或独立的调试都可进行。当它设为 Standalone 时, 虽然仍能在 TC 中运行程序, 但只能在独立的调试程序中作调试。当它设为 None 时, .EXE 文件中没有放调试信息, 因此两种调试程序都不能对程序调试。
Ctrl-F7	Break/Watch/Add Watch	增加一个观察表达式。
	Break/Watch/Delete Watch	删除一个观察表达式。
	Break/Watch/Edit Watch	允许编辑观察表达式。
	Break/Watch/Rename All Watch	删除所有的观察表达式。
Ctrl-F8	Break/Watch/Toggle Breakpoint	在光标所在行设置或消除断点。
	Break/Watch/Clear Breakpoint	清除程序中的所有断点。
	Break/Watch/Next Breakpoint	显示下一个断点。

表4 调试命令与热键 (2)

热键	菜单命令	说 明
F5		在全屏和分屏模式之间扩大或缩小活动窗口。
Alt-F5		显示用户屏幕, 按任意键回到集成环境。
F6		在编辑窗和观察窗或信息窗之间转换活动窗口。
Alt-F6		如果编辑窗是活动的, 就把上一个文件调入编辑窗。如果下部窗口是活动的, 就在观察窗和信息窗之间进行转换。
Ctrl-F9	Run/Run	带调试程序或不带调试程序运行程序。如有必要就编译源文件并连接。当程序是以 Debug/Source Debugging 和 O/C/C/Obj Debug Information 为 on 时编译和连接的, 就运行程序到一个断点或到程序结束。
	Project/Remove Message	删除信息窗的内容。

E. 8 软件测试指南

测试软件比单纯学会使用调试程序复杂得多,判断程序是否不正常工作和为什么不正常工作是程序设计中更困难的阶段。

这里提出一些技术将使测试工作变得容易些。

E.8.1 开发标准方法

开发软件测试的标准方法:经验步骤检查表可使程序员得到一个可靠的程序。

这还不是测试程序的“精确的”方法:检查表将依赖于所写程序的类型,程序员的优缺点及其个人的风格。下面的检查表可以作为起点:

(1)给程序一些简单但不琐碎的输入,对代码进行跟踪,使用Debug/Evaluate充分观察表达式以检查数据项的值。每次改正正找到的一个或几个错误。

(2)输入另一组数据以检查上一步未测试的部位。

(3)测试程序中的每条语句,也许能够找到被认为不会存在的错误。例如,在WORDCNT中,测试每一条语句将会发现else后的一个分号在遇到一个比MAXWORDLEN长的单词时具有灾难性的后果(这个分号通常处于编辑窗的右边缘之外,因此很可能不注意到它)。

(4)注意有些语句和表达式要用多种方法进行测试。例如:

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

strcmp可能返回三种值:0($a=b$), -1($a<b$), 1($a>b$)。这说明要用三组数据来测试这条语句,以验证strcmp在每种情况下都能正确地工作。

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

这条语句包含一个可以产生两种不同值的“隐含的if”。

(5)特别要注意边界条件:即,使程序从循环中退出及填满数组等的条件。在不能正确地处理边界条件时尤其要找出错误。

WORDCNT包含两个与边界条件有关的例子。首先,最后一个for循环不能打印wordlen中具有长度为MAXWORDLEN的元素。其次,wordlen为每个元素分配的空间太少,因此该元素甚至无法存在。

(6)不考虑调试程序和测试整个程序的正确行为。假如该程序被别人使用,别人总是希望它能很好地工作。用它可能遇到的各种类型的错误来测试它。一个能正确地处理多种类型错误的程序被称为是坚定性好的。

(7)如果编写的程序要被别人使用,就要让至少一个旁人来测试它。通常选择那些技能与要求都与未来的用户相似的人,但他必须要有足够的耐心和热情来挖掘隐藏的错误并且准确地报告。

E.8.2 彻底地测试修改

修改程序之后,要彻底地对受影响的部分重新进行测试。或许还要测试那些虽未改动但受别的改动影响的部分。

如果程序很复杂,就要保存所完成的测试的记录。在修改程序时,这些记录有助于重复那些结果可能受改动影响的测试。如果测试涉及某个输入文件,就把该文件保存起来。

E.8.3 预防性设计

采用预防性设计程序可以避免错误,就象防御性地开车能避免交通事故一样。

将代码写清楚,使用一致的缩进格式、注释以及有意义的变量名。

保持代码的简单性。将复杂的操作作用多个简单的语句来表达而不用少数几个复杂的语句。Turbo C的代码优化将使代码效率显著提高,并且更易于调试、阅读和修改。

用目的简单且完善定义的函数来构造程序。这使设计测试实例和分析结果变得容易。也使程序易读和易修改。

将每个函数所需的数据和改变的元素个数减至最小。这也使设计测试实例和分析结果以及阅读和修改程序变得容易。它还能限制由错误的函数所引起的灾难,允许在调试期间多次运行该函数。这样设计的程序就说是松散耦合的。

在写程序时不要试图多节约程序的最后一位。当试图把代码变得尽可能有效时,它也将变得难以阅读和调试。只是在程序的运行实在太慢时,才有必要确定程序的哪一部分可以显著提高速度,怎样做最好。

注意书写那些可在程序中以多种方式使用的函数，或者可被别的程序使用的函数。书写和调试一个通用函数通常比书写两个或更多的专用函数容易得多。

● E.8.4 自底向上调试

首先尽可能集中精力调试那些最低层的函数（即没有调用别的函数的函数），然后再向上调试：一直到main。这样就打下了可靠的函数基础，在程序的其它部分调用它们时只须单步执行即可。

E.8.5 寻找同类错误

当找到一个错误后，就可以仔细寻找同一类型或程序同一部位的错误。例如，如果发现函数调用：

```
fp=fopen("rb", filename);
```

应该是：

```
fp=fopen(filename, "rb");
```

时，就可检查程序中其它对fopen的调用或对相似函数的调用以找出同样的错误。

E.9 调试在线汇编代码

如果使用了与Borland的TASM汇编程序相连的集成开发环境，就可以不使用外部调试程序而单步执行汇编级的代码了。（如果想了解汇编级调试的所有特征，需要有Borland的独立调试程序）。

当以TASM的-z开关汇编了某个汇编语言模块后，TC就能从汇编源程序中识别行和符号。如果单步进入汇编级函数，TC就显示该函数的汇编语言源代码。可以对汇编语言源代码使用一般的TC调试命令，如Debug/Go to Cursor (F4)，Debug/Trace Into (F7)以及Debug/Step Over (F8)等。

在汇编语言源代码中定义的许多符号都可用以求表达式的值和观察表达式。另外，还可以存取伪寄存器（—AX，—BX等）和反映CPU的标志寄存器状态的—FLAGS变量。但是，由于TC是一个C的开发环境，它不能识别任何汇编语言结构或表达式。