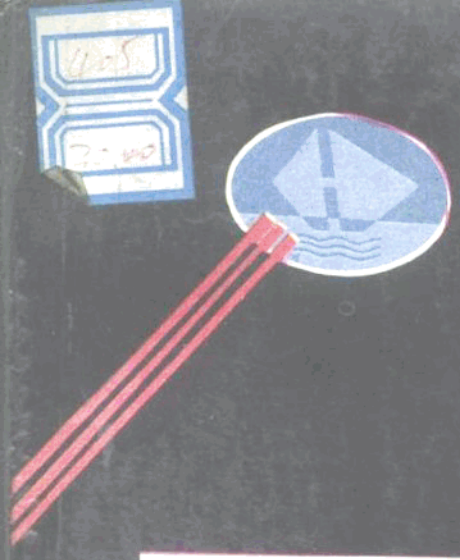

TURBO PASCAL5.5

程序设计技术及库函数集锦

中国科学院希望高级电脑技术公司

刘 京
宋明华
史 峰 编译
薛 梅

本书全面系统介绍Turbopascal5.5程序设计方法及最新增强特性，并配精选过的习题与参考答案。

本书分上、下册，上册介绍其基本程序设计方法；下册介绍其高级程序设计技术和库函数。

下册

目 录

第十三章　高级技术

§13.0　关键字和标识符

Absolute、ChDir、DosExitCode、Exec、FillChar、Halt、Mem、MemL、MemW、MkDir、Move、OvrClearBuf、OvrFileMode、OvrGetRetry、OvrloadCount、OvrReadBuf、OvrReadFunc、OvrSetRetry、OvrTrapCount、ParamCount、ParamStr、Port、PortW、PROCEDURE、FUNCTION、RmDir、SetTextBuf、SwapVectors

本章集中了各种技巧和窍门，使 Turbo Pascal 与众不同。在这里，读者可学到条件编译，类型检查，特殊变量，正文文件设备驱动器和用户出口过程的设计以及其它高级技术。

§13.1　条件编译

使用一特殊的编译指令集，可写出基于不同条件做不同编译的程序，这种技术称为条件编译它，有益于调试和设计易于用户化的软件。

过程很简单，首先定义各种符号，类似于其它的 Pascal 标识符。(参见图 1－2。)然后，根据某一符号是否定义告诉 Turbo Pascal 来编译相应程序段。定义不同的符号就改变了被编译的代码段。

以编译器指令形式出现的命令构成了 Pascal 内部的一种微型语言。要切记，条件编译命令并不是 Pascal 语句。象其它编译器指令一样，这些命令是少数改变编译器操作的指令。表 13-1 列出了在一种语言内部的微型语言的编译器指令。

表 13-1　　条件编译指令

指令	描述
{$DEFINE id}	定义符号id
{$UNDEF id}	未定义符号id
{$IFDEF id}	如果id被定义，正常继续编译。
{$IFNDEF id}	如果id未定义，正常继续。
{$IFOPT id}	正常继续否取决于开关id
{$ELSE}	如果前面IF…失败，编译下一段。
{$ENDIF}	标志IF和ELSE段结束。

在表 13-1 中，id 表示一条件符号，用户可以发明符号，也可使用几个已定义的符号之一。例如，下面定义了一个符号，名为 Debugging:

```
{$DEFINE Debugging}
```

定义一个符号之后，用户可使用指令 IFDEF(如果已定义)和 IFNDEF(如果未定义)来测试某一符号是否存在。为了编译二条测试二个程序变量的 Writeln 语句，可这样写:

```
{$IFDEF Debugging}
  writeln ('Debugging');
```

```
  writeln ('i=', i, 'j=',j);
{$ENDIF}
```

这和 Pascal 中的 IF 语句不同。只有事先定义了符号 Debugging，这二条 writeln 语句才被编译。如果用户未使用{$DEFINE Debugging}命令，这两条 writeln 完全不被理睬。一个典型程序会含上两条这样的调试语句。调试完程序后，将 DEFINE 换成 UNDEF：

```
{$UNDEF Debugging}
```

使 Debugging 符号跟从来未被定义一样。现在，尽管这二条 writeln 语句仍在源程序中，但不被编译，用户当然可以删除这些调试语句，但是，如果将来程序重新开发，又需调试，你将不得不重新键入这些语句。使用条件编译方法，你只需定义 Debugging 符号来重编译。一定要弄清楚条件编译符号，象 Debugging 既不是变量也不是常量。符号没有值。它们也没有逻辑值；它们仅仅是存在或不存在。条件命令{$IFDEF Debugging}并不测试 Debugging 的逻辑值，仅仅判断符号 Debugging 是否事先定义过。

IFDEF 测试是否符号定义过，而 IFNDEF 测试符号未定义。用户可以用这种思想来给出信息标明程序是实用产品还是调试版本：

```
{$IFNDEF Debugging}
  writeln('Production Version 1.00');
{$ELSE}
  writeln ('Debugging Version 1.00');
{$ENDIF}
```

注意{$ELSE}指令标志另一 writeln 语句。根据 Debugging 在程序中是否事先定义来编译一条 writeln 语句。{$ENDIF}跟在{$ELSE}段后。

§13.1.1 测试选择项

正如所知，Turbo Pascal 有许多选择项可选。例如：可以用{$N+}和{$N-}决定是否打开数字协处理器数据类型。同样，{$R+}和{$R-}决定范围检查的开关。

使用条件编译指令{$IFOPT id}来测试任意选择项的状态，并且，如果该选择项具有指定的值，就编译其后的正文直到遇见下一个{$ELSE}或{$ENDIF}。例如，要编译一程序，其中有一变量 distance，其类型在一般系统中为 Real，在带有数字协处理器的系统中为 Extended 可以写变量段如下：

```
VAR
  {$IFOPT N+}
    distance:Extended;
  {$ELSE}
    distance:Real;
  {$ENDIF}
```

对本声明，改{$N+}为{$N-}，或改变 Options：Compiler：Numeric－processing 为 Software，都将把程序编译在无协处理器的系统上。条件编译命令根据自动选定的 Extended 或 Real 类型来改变源代码。

§13.1.2 预定义符号

除了自己定义符号外，用户还可以测试某内部符号是否存在。表 13－1 列出了 Turbo Pascal 的预定义条件符号。

用符号 VERn 来设计在不同版本的 Turbo Pascal 上编译的程序，n 等于 40 对应版本 4.0，41 对应版本 4.1 等等。假设只有版本 3.9(实际上不存在此版本)支持一数据类型 Quark。其它版本都用 Real 代替。为了设计一能在任意版本下正确编译的程序，可写：

```
VAR
  {$IFDEF VER39}
    BlackHole:Quark;
  {$ELSE}
    BlackHole:Real;
  {$ENDIF}
```

如果将来在各种系统上兼容的 Turbo Pascal 被实现，表 13－2 中的 MSDOS 和 CPU86 预定义符号将会更有意义。这可能是这些符号现今很少用到的原因。

表 13-2 中最后一个符号 CPU87，仅在计算机安有'87 族算术协处理器芯片时被定义。注意在编译时刻进行条件检查，而不是在运行时刻。程序 13-1 示范了怎样用 CPU87 和 IFOPT 来写一程序在算术芯片可用时自动使用协处理器数据类型。

表 13-2　　预定义条件符号

符号	定义含义
VER40	Turbo pascal 版本 4.0
VER50	Turbo pascal 版本 5.0
VER55	Turbo pascal 版本 5.5
VERn	Turbo pascal 版本 n
MSDOS	计算机支持MS-DOS或PC-DOS
CPU86	计算机有'86族处理器
CPU87	计算机有算术协处理器

第一行到第五行的条件指令测试编译器是否为版本 4.0。如果是，并且第二行测试为正，指令{$N+}无数学协处理器仿真使扩展的实数数据类型可用；否则，{$N-}使之不可用。3~5 行处理 Turbo Pascal 5.0 或更高版本；此时，{$N+,E+}自动测试和使用协处理器(如果提供的话)或仿真程序。注意，可以将指令串放在一起，象第二行，也可以分别插入它们，象第一行和第 3-5 行。

11~15 行用 IFOPT，ELSE 和 ENDIF 在{$N+}起作用时定义变量常数 SpeedOfLight 为 Double 类型，在{$N+}不起作用时定义为 Real 类型(只有在版本 4.0 中)。使用类似的技术在 23~27 行中定义两个变量 Miles 和 Seconds。根据版本号和 N+/-的设置，30~38 行显示下列信息之一：

```
Coprocessor installed
Coprocessor not installed
Coprocessor or emulation installed
```

记住所有这些决定都在编译时作出，而不是在运行时。条件编译指令不生成代码——它们仅仅告诉编译器哪一段编译、哪一段跳过。

程序 13-1

```
 1: {$IFDEF VER40}
 2:    {$IFDEF CPU87} {$N+} {$ELSE} {$N-} {$ENDIF}
 3: {$ELSE}
 4:    {$N+,E+}
 5: {$ENDIF}
 6:
 7: PROGRAM Light;
 8:
 9: CONST
10:
11: {$IFOPT N+}
12:    SpeedOflight: Double = 186282.3976; { Miles per second }
13: {$ELSE}
14:    SpeedOflight: Real = 186282.3976;
15: {$ENDIF}
16:
17:    PromptChar = ']';
18:    ProgramName = 'S p e e d o f l i g h t';
19:    PromptString = 'Enter number of miles';
20:
21: VAR
22:
23: {$IFOPT N+}
24:    Miles, Seconds: Double;
25: {$ELSE}
26:    Miles, Seconds: Real;
27: {$ENDIF}
28:
29: BEGIN
30:    Writeln( ProgramName );
31:    Write( 'Coprocessor' );
32: {$IFNDEF VER40}
33:    Write( 'or emulator' );
34: {$ENDIF}
35: {$IFOPT N-}
36:    Write( 'not' );
37: {$ENDIF}
```

```
38:     Writeln( 'installed' );
39:     Writeln;
40:     Writeln( PromptString );
41:     Writeln;
42:     Write( PromptChar );
43:     Readln( Miles );
44:     Seconds:= Miles/SpeedOfLight;
45:     Writeln( 'Light travels', Miles, ' miles in');
46:     write( Seconds, ' seconds,or ');
47:     Writeln( Seconds/60.0, ' minutes.' )
48: END.
```

§13.2 类型强制转换(Typecasting)

Turbo pascal 的赋值语句一般要求:=两侧的变量和值与表达式类型相同。类型强制转换是一种技术使你在需要时打破这一传统规则。破坏 Turbo pascal 强有力的类型检查能力蕴涵着危险。

有两种不同的类型强制转换：值和变量。值的类型强制转换将转换变量或表达式为另一种类型。变量的类型强制转换告诉编译器将变量看作与先前定义不同的另一类型。下面的值类型强制转换将值 66 在 Writeln 语句中转变为字符 B：

```
writeln ('The letter is',char(66));
```

数据类型标识符 Char 括起你要编译器重新指定为另一不同类型，这里是字符 B。变量类型强制转换看起来类似，但操作不同。下列转换四个字节的数组为三个字符的串：

```
TYPE
   Str3 = String[3];
VAR
   a:ARRAY[0..3] OF Byte;

a[0]:=3;
a[1]:= Byte('A'); a[2]:= Byte('B'); a[3]:= Byte('C')
Writeln (Str3(a));
```

数组和 Str3 数据类型各占四个字节。a[0]的值告诉后面有几个字符。三个值类型强制转换按下标以 ASCII 值表示的字符赋给 a[1]，a[2]，a[3]。最后，Writeln 语句写这个被强制转换为类型 Str3 的奇怪结构的数组，克服了编译器中不许用 Writeln 写字节数组的规定。(试编译 Writeln(a)，看出错信息。)变量类型强制转换通过了编译器对正确数据类型的检查，但要保证本例数组中有正确的值。

值类型强制转换和变量类型强制转换的不同有时分不清。例如,转换一个简单 Byte 变量为类型 LongInt 是值类型强制转换，因为 Byte 变量表示值。标量变量和指针也是如此。转换这些目标的类型总是一个值而不是一个变量。

尽管是隐蔽的，这个区别仍很重要。Turbo Pascal 允许不同长度的对象之间的值类型强制转换。例如：

```
VAR
  Short:ShortInt;
  Long: LongInt;
Short:=-123;
Long:=LongInt(short);
```

尽管第二个赋值语句用了变量 Short，这仍是一个值类型强制转换。(本例只为演示需要，实际可以直接把 Short 变量赋给 Long。)Short 的值转换为 LongInt 数据类型。同样 Short 正负号扩展为 Long。换句话说：类型强制转换后，Long 和 Short 有同样值。

前面示例的变量数组类型强制转换与此相反。在当时，数组被解释为不同的数据类型——无转换执行。这是要理解的重要而微妙的差别。一般说来：

- 值类型强制转换转换表达式或标量变量的值为不同的数据类型。赋值时，两种类型长度不必相同。值是正负号扩展的，即正值为正；负值为负。
- 变量类型强制转换解释变量中字节为不同的数据类型。赋值时，两种类型必须等长。

§13.2.1 类型强制转换等于自由合并

第五章解释了怎样用自由合并记录骗过编译器，使之把一个变量看成二个或更多的不同类型。使用变量类型强制转换可完成同样任务。例如，假设需要分别访问一个 32 位 LongInt 变量的高 16 位字和低 16 位字的低位和高位字节。通过变量强制转换能提出 LongInt 中的单个字或字节。

程序 13-2 示范了怎样做到这一点。程序把一个 LongInt 变量转换为二个记录类型变量，Words(3~5 行)有二个字域和 Bytes(6~8 行)有二个字节域。Writeln 显示原值(Long)，高位和低位字在此值中，且 4 个字节在这二个字中。(第十四行不必有分号，它只为方便敲入上一行。)

注意限定词——句号和域名——在 18~23 行中括号后。只有变量类型强制转换才有这样的限定词。正如例中所示，在使用值时，用记录进行类型强制转换是完成变量类型强制转换的一种方法。

程序 13-2

```
1:  PROGRAM LongBytes;
2:  TYPE
3:      Words = RECORD
4:                  LoWord, HiWord: Word
5:              END;
6:      Bytes = RECORD
7:                  LoByte, HiByte: Byte
8:              END;
```

```
 9:  VAR
10:     Long: LongInt;
11:  BEGIN
12:   Writeln('                                    LoWord          HiWord'     );
13:   Writeln( '    Value    LoWord   HiWord   LoByte   HiByte   LoByte   HiByte' );
14;            (* 12345678123456781234567812345678123456781234567812345678 *)
15:
16:   FOR Long:= -10 TO 10 DO
17:         Writeln( Long: 8,
18:                  Words( Long ).LoWord : 8,
19:                  Words( Long ).HiWord :8,
20:                  Bytes( Words( Long ).LoWord ).LoByte : 8,
21:                  Bytes( Words( Long ).LoWord ).HiByte : 8,
22:                  Bytes( Words( Long ).HiWord ).LoByte : 8,
23:                  Bytes( Words( Long ).HiWord ).HiByte : 8 )
24:  END.
```

§13.2.2 类型强制转换指针

类型强制转换的另一种用途是把一种数据类型的指针转变为另一种。假设有一 String 类型的指针，但你想让编译器把变量当作一个字节，来记算串中字节数。声明如下：

```
TYPE
   sptr = ^String;
   bptr = ^Byte;
   VAR
     sp:sptr;
     bp:bptr;
```

然后用 New 在堆中创建一串变量并赋值：

```
New (sp);
sp^:= 'String em up';
```

因为 Turbo Pascal 有很强的类型检查的能力，所以不能这样写：

```
bp:= sp; {错误}
Writeln('Length of String =',bp^);
```

Turbo Pascal 不允许把指针赋给不同类型的另一指针。为了解决这一问题，并定位串中第一个字节，用类型强制转换技术：

```
Writeln ('Length of String =',bptr(sp)^);
```

这就把 sp 转换为类型 bptr，显示串长度为 14。(不再需要 bp 变量。)这种技术的重要性在于它不生成任何代码。bptr(sp)不是函数调用，尽管很象。它仅仅在编译时指出 sp 为一 bptr 类型的变量。

§13.3 Mem、MemW 和 MemL

Mem、MemW 和 MemL 这三个数组为用户程序打开所有内存。这些特殊变量是预声

明的——用户可直接使用之。

用 Mem 访问内存的单个字节；MemW 访问一个字即二个字节。MemL 用来读写 4 个字节的长整型数。在两种情况下，用户要用段和偏移量作为数组下标来指定内存地址。可对内存读或者写。例如，下面语句读出当前时间值到 Word 变量 TimerCount 中：

```
TimerCount:= MemW[0000:$046C];
```

如果你懂 Basic，会看出这和 PEEK 命令相同。把数组放在赋值号左边，就变成了给内存赋值(类似于 Basic 中 POKE)。下面例子控制 IBM－PC 打印屏幕键的开关。运行该语句一次后，SHIFT－PrtSc 键不再打印屏幕内容。再运行一次，恢复功能：

```
Mem[$0050:0]:= Mem[$0050:0] XOR 1;
```

程序 13－3 把地址为 0000：$0410 中的 16 位值赋给变量 EquipFlag(16 行)。因为是内容为连机设备信息的二个字节的字，所以用 MemW。程序用逻辑 AND 和移位译出信息并显示配置表。

当然，这种程序要保证 0000：$0410 中有需要的信息。因为不能保证所有机器在同一地址存放同一信息，所以程序 13-3 是一个“系统依赖”的例子程序，在其它系统可能不可运行。(尽管如此，本程序可在大多数的 PC 及其兼容机上运行。)

程序 13-3

```
 1:  PROGRAM Devices;
 2:  VAR
 3:     EquipFlag: Integer;
 4:
 5:  PROCEDURE ShowVid( n : Integer );
 6:  BEGIN
 7:     CASE n OF
 8:        1 : Writeln('40x25 (Color)' );
 9:        2 : Writeln('80x25 (Color)' );
10:        3 : Writeln('80x25 (Monochrome)' );
11:        ELSE Writeln('<not used>' )
12:     END  {case}
13:  END; {ShowVid}
14:
15:  BEGIN
16:     EquipFlag:= MemW[ 0000:$0410 ];
17:     Writeln;
18:     Writeln( 'IBM PC Devices' );
19:     Writeln;
20:     Writeln( 'Number of printers ..... ',
21:        EquipFlag SHR 14 );
22:     Writeln( 'Game I/O attached ..... ',
```

```
23:         ( EquipFlag AND $1000 )=1 );
24:     Writeln( 'Number of serial ports ..... ',
25:         ( EquipFlg SHR 9 ) AND $07 );
26:     Writeln( 'Number of diskette drives .. ',
27:         ( (EquipFlag AND 1) * (1+ ( EquipFlag SHR 6 ) AND $03 ) ) );
28:     Write( 'Initial video mode ..... ');
29:         ShowVid( ( EquipFlag SHR 4 ) AND $03 );
30:     Writeln
31: END.
```

§13.4 Port 和 Portw

内部数组 Port 和 PortW 访问机器的输入和输出端口。一个端口是连在处理器上的一个指定通道。通过数据口访问设备是和硬件外围通讯的最简单的方式。比方说，可以直接读写存在象视频显示或串行通讯行等各种设备的控制电路和芯片上的数据。

Port 访问 8 位数据。PortW 访问 16 位口。把数组放在赋值号右端即为读口。放在左端即为写口。下面语句和汇编语言的 IN 指令作用相同，假设你有单色视频显示和打印机适配器在 IBM PC 机上(变量 PrintStats 为 Byte 类型)：

```
printStatus:= port [$03BD];
```

试用上面语句写一短程序，然后分别在打印机开和关时运行程序。如果系统硬件正确，会看到二次从地址$3BD 中取的值不同。程序应译出信息来判断打印机状态。

为向口写值，过程相反。下面命令设置和清除无屏蔽中断：

```
Port [$AO]:= $80; {设置}
Port [$AO]:= 0;   {清除}
```

在实验对数据口的操作时，要特别小心。尽管这里选的例子都相对安全，有些口不仅在写时，在读时也会激活功作。你可以很容易地打开磁盘驱功器马达，影响视频显示、破坏盘中数据和其它事故——如果你乱动数据口的话。

为了实验口操作，一相对安全的设备是 PC 的时钟，它有特殊的数据寄存器在口$40，$42 和$43。一个相关的芯片为 8255A 可编程外围接口(PPI)口地址为$61，连时钟到喇叭上使之发声。

程序 13－4 用 Port 数组赋给时钟寄存器一个原始值(16～18 行)。然后打开 PPI 输出第 0 位和第一位，送到喇叭使之发声。因为 PPI 还有其它用途，所以要把所有位保存起来而不是仅仅保存控制时钟输出到喇叭的位。程序通过存起原始口值做到这一点(19 行)。然后在 22 行重装值。这也就关闭了喇叭。

IBM PC 技术参考中册中详细讲了各种口的信息和口值的含义。

程序 13-4

```
1: PROGRAM Beeper;
2: USES Crt;
3: CONST
```

```
4:      PortB          = $61;  { 8255 地址 }
5:      Timer          = $40;
6:      TimerOut       = $42;
7:      TimerControl = $43;
8:  VAR
9:      ch : Char;
10:     Pitch, Duration : Integer;
11:
12: PROCEDURE Beep( Frequency, Milliseconds : Integer );
13: VAR
14:     SavePortB : Byte;
15: BEGIN
16:     Port[ TimerControl ] := $B6;                { Select timer mode }
17:     Port[ TimerOut     ] := lo( Frequency ); { Set frequency divisor }
18      Port[ TimerOut     ] := hi( Frequency );
19:     SavePortB := Port[ PortB ];                 { Save old PortB setting }
20      Port[ PortB ]:= SavePortB OR $03;           { Turn on speaker }
21:     Delay( Milliseconds );                      { Wait }
22:     Port[ PortB ]:= SavePortB                   { Turn off speaker }
23: END; {Beep}
24:
25: BEGIN
26:     WRITELN( 'Beeper' );
27:     Pitch:= $255; Duration:= 75;  {Default values }
28:     REPEAT
29:         Writeln;
30:         Write( 'Pitch? '     ); Readln( Pitch );
31:         Write( 'Duration? ' ); Readln( Duration );
32:         Writeln;
33:         Writeln( ' Pitch     = ', Pitch );
34:         Writeln( ' Duration = ', Duration );
35:         Writeln;
36:         Write(  '  Press any key to stop...' );
37:         WHILE NOT Keypressed DO
38:         BEGIN
39:             Beep( Pitch, Duration );
40:             Delay( 100 )
41:         END; {while}
42:         Write( Readkey );
43:         Writeln; Writeln;
```

```
44:          Write( 'Again? ' );
45:          ch:= Readkey;
46:          Writeln( ch )
47:      UNTIL Upcase ( ch ) <> 'Y'
48:  END.
```

§13.5 写大型程序

许多程序可能有几千、几万条语句，可 Turbo Pascal 的编辑器只有 6 万字符的上限。如何解决这个矛盾？这里有几种方法。

§13.5.1 编译大型程序

Turbo Pascal 能编译几千个符号的大程序。但是，如果系统只有有限的自由内存，编译器将无法工作。下面方法能得到更多空间：

- 重新引导不装入常驻内存工具、键盘增强器、函数一键宏等。
- 使用命令行 TPC.EXE 编译而不用集成环境式 TURBO.EXE，所有 Turbo Pascal 编辑器和调试器所占内存均为编译器所用。
- 改集成环境式编译器的 Options－Linker－Link buffer 从“Memery”到“Disk”或在命令行编译器中用 IL 选择。这将延长编译时间但释放了连接时通常由编译器所占空间。
- 用单元去“藏”起实现段单元的信息。把程序组织成单元，并在每一个单元接口段放最少的符号数，以减少编译器放在内存中的符号。
- 运行 TPUMOVER 工具删除 TURBO.TPL 中一些单元。仅留下最常用的单元。

《Turbo Pascal 用户手册》列出了几种其它方法，但这里的几种方法效果最好。

当然，可以买更大内存的机器。Turbo Pascal 自动检查和使用编辑用 64K 扩充内存(EMS)，如果其它程序使用了 EMS RAM，将检查是否有办法为 Turbo pascal 保留 64K。

§13.5.2 包含文件

处理大型程序的一个办法是把它分成一些可编辑的块，把程序正文存在 include 文件中。一个包含文件是一个独立的盘正文文件，编译器在编译其它正文时包含该文件。例如，下面一行告诉编译器包含 LIBRARY.PAS 文件，该文件内有程序要用到的许多函数和过程。

```
{$I LIBRARY.PAS}
```

不要和指令{$I+}，{$I－}混淆，这两个指令用来打开、关闭 I/O 错误检查。在 I 后面跟空格时，编译器希望发现一个文件名。也可以写成下面形式：

```
(*$I LIBRARY.PAS*)
```

编译器碰到包含指令时，它读包含文件就象是源代码的一部分。包含文件本身也可以包含其它文件，这种关系最好不要超过 8 层，尽管可以超过。唯一限制是包含指令不能出现在 BEGIN 和 END 之间。例如，就不能包含过程的一部分。

编译器完成包含文件后，将继续源文件。当然，编译父文件时包含文件应在盘中。否则，将出错：

```
Error15: File not found
```

如果文件无扩展名，自动加.PAS。下面指令仍包含 LIBRARY.PAS：

{$I LIBRARY}

§13.5.3 使用 Dos 单元执行过程

另外一种分段方法是把一个大程序分成几个模块——独立运行的子程序，然后写出控制程序去调用各个模块。控制程序可能是主菜单，从这里选择主要程序操作。每一操作是一独立模块．从设计到编译到调试都是独立的。

Dos 单元有一个过程 EXEC，它能让用户运行其它程序或使用操作系统进程。一个进程结束后，控制返回到调用程序中。EXEC 有两个串类型参数：

```
Exec ( Path, cmdline : String );
```

Path 串是你要运行的 EXE 或 COM 程序。它可以是编译过的 Turbo Pascal(程序也可以是其它能在 DOS 命令行下运行的程序。Path 可以含驱动器和子目录名，象：C:C\UTIL\FORMAT。

串 CmdLine 为要传给程序的信息。例如，如果有一程序名为 SORT.EXE(此处未列出)，它能输入和输出文件名，然后把 NAMES.TXT 排列到 NAMES.SRT，可以这样写：

```
Exec ('SORT','NAMES.TXT NAMES.SRT');
```

程序 13－5，13－6，13－7 示范了怎样用 EXEC 来设计多模块程序。为了编译完整程序，把程序 13－5 存入 MENU.PAS 并编译或 MENJ.EXE。程序 13－6 存入 SUB1.PAS，编译成 SUB1.EXE。最后把 13－7 存入 SUB2.PAS，编译成 SUB2.EXE。退出 Turbo Pascal，留下二个.EXE 文件在盘中，键入 MENU 运行程序。按 1 选子程序 1，2 选子程序 2，3 退出。

每个程序的第一行用一个特殊的编译指令去限制内存的作用。指令格式如下：

```
{$M Stacksize, heapmin, heapmax }
```

Stacksize 为字节数，从 1024 到 65520。heapmin 和 heapmax 控制用 NEW 和 GetMem 生成的变量可用堆的上下界。范围从 0 到 655，360。

调用 Exec 运行程序时一定要限制栈和堆的大小，这是因为 Turbo Pascal 和 DOS 通常为程序保留所有内存。因此，除非改变主程序的内存需要，否则子程序将无处运行。

如果程序中不用到 New 和 GetMem，heapmin 和 heapmax 应置为零。正确地设置 Staksize 限制比较棘手。每个程序都不同的栈需要，主要根据有多少过程调用其它的过程。(换句话说，有多少层过程和函数嵌套)，加上参数的数目和类似。一般，8K 左右的栈较适合于多数中小程序。缺省值为 16K，32K 的栈就很大了。

程序 13-5 中 36~37 行调用 Exec，传送要运行的子程序名。第一个调用不传送参数。第二个传送二个参数。注意这两个参数写成 Single 串。(第 37 行)。

程序 13-5

```
1:  {$M 4000, 0, 0 }      { 4K stack, no heap minimum or maximum }
2:
3:  PROGRAM Menu;           { Compile to MENU.EXE }
4:
5:  USES Crt, Dos;
6:
```