第九章　子视窗控制项

回忆第七章的 CHECKER 程式。这些程式显示了矩形网格。当您在一个矩形中按下滑鼠按键时，该程式就画一个 x；如果您再按一次滑鼠按键，那么 x 就消失。虽然这个程式的 CHECKER1 和 CHECKER2 版本只使用一个主视窗，但 CHECKER3 版本却为每个矩形使用一个子视窗。这些矩形由一个叫做 ChildProc 的独立视窗讯息处理程式维护。

如果有必要，无论矩形是否被选中，都可以给 ChildProc 增加一种向其父视窗讯息处理程式(WndProc)发送讯息的手段。通过呼叫 GetParent，子视窗讯息处理程式能确定其父视窗的视窗代号：

```
hwndParent = GetParent (hwnd) ;
```

其中，hwnd 是子视窗的视窗代号。它可以向其父视窗讯息处理程式发送讯息：

```
SendMessage (hwndParent, message, wParam, lParam) ;
```

那么 message 应该设定为什么呢？您可以随意地设定，数值大小可以与 WM_USER 相同或更大，这些数字代表和预先定义的 WM_ 讯息不冲突的讯息。也许对这个讯息，子视窗可以将 wParam 设定为它的子视窗 ID。如果在该子视窗单击，那么 lParam 可以被设为 1；如果未在该子视窗上单击，那么 lParam 将被设为 0。这是处理方式的一种选择。

事实上，这是在建立一个「子视窗控制项」。当子视窗的状态改变时，子视窗处理滑鼠和键盘讯息并通知父视窗。使用这种方法，子视窗就变成了其父视窗的高阶输入装置。它将与自己在萤幕上的图形外观相应的处理，对使用者输入的回应以及在发生重要的输入事件时通知另一个视窗的方法给封装起来。

虽然您可以建立自己的子视窗控制项，但是也可以利用一些预先定义的视窗类别（和视窗讯息处理程式）来建立标准的子视窗控制项，您一定在别的 Windows 程式中看到过这些控制项。这些控制项采用的形式有：按钮、核取方块、编辑方块、清单方块、下拉式清单方块、字串标签和卷动列。例如，如果想在您的试算表程式的某个角落放置一个标有「Recalculate」的按钮，那么您可以通过呼叫 CreateWindow 来建立这个按钮。您不必担心滑鼠操作、按钮显示操作或按下该按钮时的自动闪烁操作，这些是由 Windows 内部完成的。您所要做的只是拦截 WM_COMMAND 讯息——当按钮被按下时，它通过这一讯息通知您的视窗讯息处理程式。真的这样简单吗？是的，一点也没错。

子视窗控制项在对话方块中最常用。在第十一章中您将会看到，子视窗控制项的位置和尺寸，是在范例程式的资源描述叙述中的对话方块模板里定义的。

但是，您也可以使用预先定义的，在普通视窗显示区域里的子视窗控制项。您可以呼叫一次 CreateWindow 来建立一个子视窗，并通过呼叫 MoveWindow 来调整子视窗的位置和尺寸。父视窗讯息处理程式向子视窗控制项发送讯息，子视窗控制项向父视窗讯息处理程式传回讯息。

在建立普通视窗时，首先定义视窗类别，并使用 RegisterClass 将其注册到 Windows 中，然後用 CreateWindow 命令依据该视窗类别建立一个普通视窗，从第三章开始，我们就是这么做的。但是，当您使用预先定义的某个控制项时，不必为子视窗注册视窗类别，视窗类别已经存在於 Windows 之中，并且有一个预先定义的名字。您只需在 CreateWindow 中把它们用作视窗类别参数。CreateWindow 中的视窗样式参数准确地定义了子视窗控制项的外形和功能。Windows 内建了处理发送给依据这些视窗类别建立的子视窗讯息的视窗讯息处理程式。

直接在您的视窗上使用子视窗控制项完成某些任务，这些任务的层次低於在对话方块中使用子视窗控制项所要求的层次。这里，对话方块管理器在您的程式和控制项之间增加一个隔离层。值得一提的，您可能会发现在您的视窗上建立的子视窗控制项，没有利用 Tab 键或方向键将输入焦点从一个控制项移动到另一个控制项的内部功能。子视窗控制项能够获得输入焦点，但是获得後，它将不能把输入焦点传回给父视窗。这就是本章要解决的问题。

Windows 程式设计的文件在两个地方讨论了子视窗控制项：首先是，简单的常用控制项，我们可以在/Platform SDK/User Interface Services/Controls 的文件所描述的无数对话方块中看到。这些子视窗包括按钮（其中包括核取方块的单选按钮）、静态控制项（例如文字标签）、编辑方块（您可以在此编辑一行或多行文字）、卷动列、清单方块和下拉式清单方块。除下拉式清单方块以外，在 Windows 1.0 中就包括了这些控制项。这部分的 Windows 文件还包括 Rich Text 文字编辑控制项，它与编辑方块相似，但还允许编辑不同字体与样式的格式化文字，以及桌面应用工具列。

相对於「常用控制项」，还有一些神秘的特殊控制项。这些控制项在/Platform SDK/User Interface Services/Shell and Common Controls/Common Controls 描述。本章不讨论常用控制项，但它们将出现在本书的其他部分。在这部分的 Windows 文件中，很容易找到您想从别的 Windows 应用程式中应用到您自己的应用程式里头那些部分资讯。

按钮类别

下面我们将通过叫做 BTNLOOK（「button look」）的程式来开始介绍按钮

视窗类别，如程式 9-1 所示。BTNLOOK 建立 10 个子视窗按钮控制项，每个控制项对应一个标准的按钮样式，因此共有 10 种标准按钮样式。

程式 9-1 BTNLOOK

```
BTNLOOK.C
/*------------------------------------------------------------------------
      BTNLOOK.C --      Button Look Program
                                (c) Charles Petzold, 1998
------------------------------------------------------------------------*
/

#include <windows.h>
struct
{
      int               iStyle ;
      TCHAR *           szText ;
}
button[] =
{
      BS_PUSHBUTTON,          TEXT ("PUSHBUTTON"),
      BS_DEFPUSHBUTTON,       TEXT ("DEFPUSHBUTTON"),
      BS_CHECKBOX,            TEXT ("CHECKBOX"),
      BS_AUTOCHECKBOX,        TEXT ("AUTOCHECKBOX"),
      BS_RADIOBUTTON,         TEXT ("RADIOBUTTON"),
      BS_3STATE,              TEXT ("3STATE"),
      BS_AUTO3STATE,          TEXT ("AUTO3STATE"),
      BS_GROUPBOX,            TEXT ("GROUPBOX"),
      BS_AUTORADIOBUTTON,     TEXT ("AUTORADIO"),
      BS_OWNERDRAW,           TEXT ("OWNERDRAW")
} ;

#define NUM (sizeof button / sizeof button[0])
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                                          PSTR szCmdLine, int iCmdShow)
{
      static TCHAR      szAppName[] = TEXT ("BtnLook") ;
      HWND                          hwnd ;
      MSG                     msg ;
      WNDCLASS                wndclass ;

      wndclass.style                    = CS_HREDRAW | CS_VREDRAW ;
      wndclass.lpfnWndProc              = WndProc ;
      wndclass.cbClsExtra               = 0 ;
      wndclass.cbWndExtra               = 0 ;
      wndclass.hInstance                = hInstance ;
      wndclass.hIcon                    =       LoadIcon        (NULL,
IDI_APPLICATION) ;
```

```
     wndclass.hCursor          = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground       = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName        = NULL ;
     wndclass.lpszClassName       = szAppName ;

     if (!RegisterClass (&wndclass))
    {
                MessageBox (     NULL, TEXT ("This program requires Windows
NT!"),
                                                          szAppName,
MB_ICONERROR) ;
                return 0 ;
     }

     hwnd = CreateWindow (  szAppName, TEXT ("Button Look"),
                            WS_OVERLAPPEDWINDOW,
                            CW_USEDEFAULT, CW_USEDEFAULT,
                            CW_USEDEFAULT, CW_USEDEFAULT,
                            NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
     TranslateMessage (&msg) ;
                     DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
     static HWND      hwndButton[NUM] ;
     static RECT      rect ;
     static TCHAR     szTop[]          = TEXT ("message   wParam  lParam"),
                      szUnd[]          = TEXT ("_______   ______  ______"),
                      szFormat[]  = TEXT ("%-16s%04X-%04X   %04X-%04X"),
                      szBuffer[50] ;
     static int       cxChar, cyChar ;
     HDC              hdc ;
     PAINTSTRUCT      ps ;
     int              i ;

     switch (message)
     {
     case  WM_CREATE :
                cxChar = LOWORD (GetDialogBaseUnits ()) ;
```

```
                cyChar = HIWORD (GetDialogBaseUnits ()) ;

                for (i = 0 ; i < NUM ; i++)
                                hwndButton[i] =  CreateWindow
( TEXT("button"),button[i].szText,
  WS_CHILD | WS_VISIBLE | button[i].iStyle,
   cxChar, cyChar * (1 + 2 * i),
   20 * cxChar, 7 * cyChar / 4,
                        hwnd, (HMENU) i,
            ((LPCREATESTRUCT) lParam)->hInstance, NULL) ;
                return 0 ;

     case  WM_SIZE :
                rect.left             = 24 * cxChar ;
                rect.top         =  2 * cyChar ;
                rect.right            = LOWORD (lParam) ;
                rect.bottom           = HIWORD (lParam) ;
                return 0 ;
     case  WM_PAINT :
                InvalidateRect (hwnd, &rect, TRUE) ;

                hdc = BeginPaint (hwnd, &ps) ;
                SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;
                SetBkMode (hdc, TRANSPARENT) ;

                TextOut (hdc, 24 * cxChar, cyChar, szTop, lstrlen (szTop)) ;
                TextOut (hdc, 24 * cxChar, cyChar, szUnd, lstrlen (szUnd)) ;

                EndPaint (hwnd, &ps) ;
                return 0 ;

     case  WM_DRAWITEM :
     case  WM_COMMAND :
                ScrollWindow (hwnd, 0, -cyChar, &rect, &rect) ;

                hdc = GetDC (hwnd) ;
                SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;

                TextOut(   hdc, 24 * cxChar, cyChar * (rect.bottom / cyChar
- 1),
                                    szBuffer,
                                    wsprintf (szBuffer, szFormat,
                                          message   ==   WM_DRAWITEM   ?
     TEXT ("WM_DRAWITEM") :
                                TEXT ("WM_COMMAND"),
                                          HIWORD (wParam), LOWORD (wParam),
                                          HIWORD     (lParam),     LOWORD
(lParam))) ;
```

```
                ReleaseDC (hwnd, hdc) ;
                ValidateRect (hwnd, &rect) ;
                break ;

      case  WM_DESTROY :
                PostQuitMessage (0) ;
                return 0 ;
    }
      return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

单击按钮时，按钮就给父视窗讯息处理程式发送一个 WM_COMMAND 讯息，也就是我们所熟悉的 WndProc。BTNLOOK 的 WndProc 将该讯息的 wParam 参数和 lParam 参数显示在显示区域的右边，如图 9-1 所示。

具有 BS_OWNERDRAW 样式的按钮在视窗上显示为一个背景阴影，因为这种样式的按钮是由程式来负责绘制的。该按钮表示它需要由包含 lParam 讯息参数的 WM_DRAWITEM 讯息来绘制，而 lParam 讯息参数是一个指向 DRAWITEMSTRUCT 型态结构的指标。在 BTNLOOK 中，这些讯息也同样被显示。我将在本章的後面更详细地讨论这种拥有者绘制（owner draw）按钮。

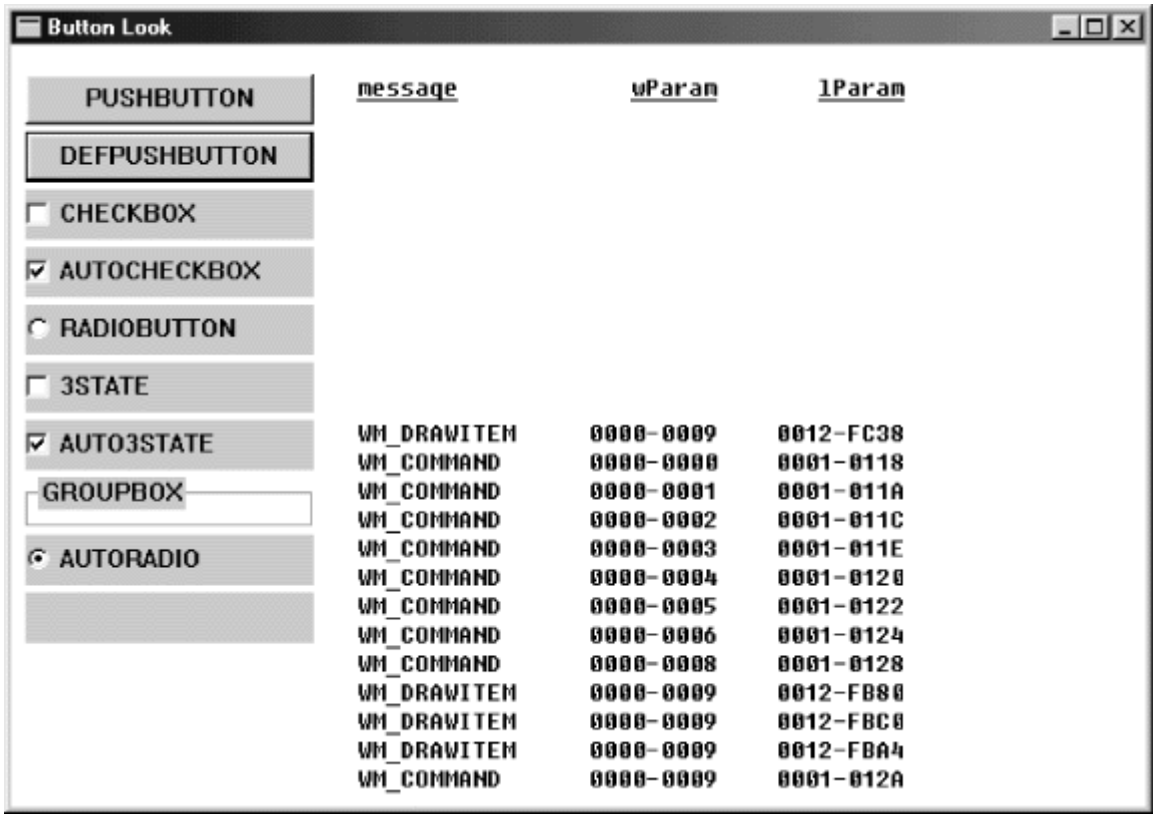

图 9-1 BTNLOOK 的萤幕显示

建立子视窗

BTNLOOK 定义了一个叫做 button 的结构，它包括了按钮视窗样式和描述性

字串，它们对应於 10 个按钮型态，所有按钮视窗样式都以字母「BS」开头，它表示「按钮样式」。10 个按钮子视窗是在 WndProc 中处理 WM_CREATE 讯息的过程中使用一个 for 回圈建立的。CreateWindow 呼叫使用下面这些参数：

Class name（类别名称）	TEXT ("button")
Window text（视窗文字）	button[i].szText
Window style（视窗样式）	WS_CHILD | WS_VISIBLE | button[i].iStyle
x position（x 位置）	cxChar
y position（y 位置）	cyChar * (1 + 2 * i)
Width（宽度）	20 * xChar
Height（高度）	7 * yChar / 4
Parent window（父视窗）	hwnd
Child window ID（子视窗 ID）	(HMENU) i
Instance handle（执行实体代号）	((LPCREATESTRUCT) lParam) -> hInstance
Extra parameters（附加参数）	NULL

类别名称参数是预先定义的名字。视窗样式使用 WS_CHILD、WS_VISIBLE 以及在 button 结构中定义的 10 个按钮样式之一（BS_PUSHBUTTON、BS_DEFPUSHBUTTON 等等）。视窗文字参数（对於普通视窗来说，它是显示在标题列中的文字）将在每个按钮上显示出来。我简单地使用标识按钮样式文字的 x 位置和 y 位置参数，说明子视窗左上角相对於父视窗显示区域左上角的位置。宽度和高度参数规定了每个子视窗的宽度和高度。请注意，我用的是 GetDialogBaseUnits 函式来获得内定字体字元的宽度和高度。这是对话方块用来获得文字尺寸的函式。此函式传回一个 32 位元的值，其中低字组表示宽度，高字组表示高度。由於 GetDialogBaseUnits 传回的值与从 GetTextMetrics 获得的值大致上相同，但 GetDialogBaseUnits 有时使用起来会更方便些，而且能够与对话方块控制项更好地保持一致。

对每个子视窗，它的子视窗 ID 参数应该各不相同。在处理来自子视窗的 WM_COMMAND 讯息时，ID 帮助您的视窗讯息处理程式识别出相应的子视窗。注意子视窗 ID 是作为 CreateWindow 的一个参数传递的，该参数通常用於指定程式的功能表，因此子视窗 ID 必须被强制转换为 HMENU。

CreateWindow 呼叫的执行实体代号看起来有点奇怪，但是它利用了如下的事实,亦即在处理 WM_CREATE 讯息的过程中,lParam 实际上是指向 CREATESTRUCT（「建立结构」）结构的指标，该结构有一个 hInstance 成员。所以将 lParam 转换成指向 CREATESTRUCT 结构的一个指标，并取出 hInstance。

有些 Windows 程式使用名为 hInst 的整体变数，使视窗讯息处理程式能存

取 WinMain 中的执行实体代号。在 WinMain 中，您只需在建立主视窗之前设定：

```
hInst = hInstance ;
```

在第七章中的 CHECKER3 程式中，我们曾用 GetWindowLong 取得执行实体代号：

```
GetWindowLong (hwnd, GWL_HINSTANCE)
```

这几种方法都是正确的。

在呼叫 CreateWindow 之後，我们不必再为这些子视窗做任何事情，由 Windows 中的按钮视窗讯息处理程式负责维护它们，并处理所有的重画工作（BS_OWNERDRAW 样式的按钮例外，它要求程式绘制它，这些将在後面加以讨论）。在程式终止时，如果父视窗已经被清除，那么 Windows 将清除这些子视窗。

子视窗向父视窗发讯息

当您执行 BTNLOOK 时，将看到在显示区域的左边会显示出不同的按钮型态。我在前面已经提到过，用滑鼠单击按钮时，子视窗控制项就向其父视窗发送一个 WM_COMMAND 讯息。BTNLOOK 拦截 WM_COMMAND 讯息并显示 wParam 和 lParam 的值，它们的含义如下：

LOWORD (wParam)	子视窗 ID
HIWORD (wParam)	通知码
lParam	子视窗代号

如果您正在移植 16 位元 Windows 程式，那么要注意改变这些讯息参数以容纳 32 位元的代号。

子视窗 ID 是在建立子视窗时传递给 CreateWindow 的值。在 BTNLOOK 中，这些 ID 被显示在显示区域中，并使用 0 到 9 分别标识 10 个按钮。子视窗代号是 Windows 从 CreateWindow 传回的值。

通知码更详细表示了讯息的含义。按钮通知码的可能值在 Windows 表头档案中定义如下：

表 9-1

按钮通知码识别字	**值**
BN_CLICKED	0
BN_PAINT	1
BN_HILITE or BN_PUSHED	2
BN_UNHILITE or BN_UNPUSHED	3
BN_DISABLE	4
BN_DOUBLECLICKED or BN_DBLCLK	5

BN_SETFOCUS	6
BN_KILLFOCUS	7

实际上，您不会看到这些按钮值中的大多数。从 1 到 4 的通知码是用於一种叫做 BS_USERBUTTON 的已不再使用的按钮的（它已经由 BS_OWNERDRAW 和另一种不同的通知方式所替换）。通知码 6 到 7 只有当按钮样式包括标识 BS_NOTIFY 才发送。通知码 5 只对 BS_RADIOBUTTON、BS_AUTORADIOBUTTON 和 BS_OWNERDRAW 按钮发送，或者当按钮样式中包括 BS_NOTIFY 时，也为其他按钮发送。

您会注意到，在用滑鼠单击按钮时，该按钮文字的周围会有虚线。这表示该按钮拥有了输入焦点，所有键盘输入都将传送给子视窗按钮控制项，而不是传送给主视窗。但是，当该按钮控制项拥有输入焦点时，它将忽略所有的键盘输入，除了 Spacebar 键例外，此时 Spacebar 键与滑鼠具有相同的效果。

父视窗向子视窗发送讯息

虽然 BTNLOOK 中没有显示这一事实，但是父视窗讯息处理程式也能向子视窗控制项发送讯息。这些讯息包括以字首 WM 开头的许多讯息。另外，在 WINUSER.H 中还定义了 8 个按钮说明讯息；字首 BM 表示「按钮讯息」。这些按钮讯息如下表所示：

表 9-2

按钮讯息	**值**
BM_GETCHECK	0x00F0
BM_SETCHECK	0x00F1
BM_GETSTATE	0x00F2
BM_SETSTATE	0x00F3
BM_SETSTYLE	0x00F4
BM_CLICK	0x00F5
BM_GETIMAGE	0x00F6
BM_SETIMAGE	0x00F7

BM_GETCHECK 和 BM_SETCHECK 讯息由父视窗发送给子视窗控制项，以取得或者设定核取方块和单选按钮的选中标记。BM_GETSTATE 和 BM_SETSTATE 讯息表示按钮处於正常状态还是（滑鼠或 Spacebar 键按下时的）「按下」状态。我们将在讨论按钮的每种型态时，看到这些讯息是如何起作用的。BM_SETSTYLE 讯息允许您在按钮建立之後改变按钮样式。

每个子视窗控制项都具有一个在其兄弟中唯一的视窗代号和 ID 值。对於代号和 ID 这两者，知道其中的一个您就可以获得另一个。如果您知道子视窗控制

项的视窗代号，那么您可以用下面的叙述来获得 ID：

```
id = GetWindowLong (hwndChild, GWL_ID) ;
```

第七章的 CHECKER3 程式曾用此函式（与 SetWindowLong 一起）来维护注册视窗类别时保留的特殊区域的资料。在建立子视窗时，Windows 保留了 GWL_ID 识别字存取的资料。您也可以使用：

```
id = GetDlgCtrlID (hwndChild) ;
```

虽然函式中的「Dlg」部分指的是对话方块，但实际上这是一个通用的函式。

知道 ID 和父视窗代号，您就能获得子视窗代号：

```
hwndChild = GetDlgItem (hwndParent, id) ;
```

按键

在 BTNLOOK 中显示的前两个按钮是「压入」按钮。按钮是一个矩形，包括了 CreateWindow 呼叫中视窗文字参数所指定的文字。该矩形占用了在 CreateWindow 或者 MoveWindow 呼叫中给出的全部高度和宽度，而文字在矩形的中心。

按键控制项主要用来触发一个立即回应的动作，而不保留任何形式的开/关指示。两种型态的按钮控制项有两种视窗样式，分别叫做 BS_PUSHBUTTON 和 BS_DEFPUSHBUTTON，BS_DEFPUSHBUTTON 中的「DEF」代表「内定」。当用来设计对话方块时，BS_PUSHBUTTON 控制项和 BS_DEFPUSHBUTTON 控制项的作用不同。但是当用作子视窗控制项时，两种型态的按钮作用相同，尽管 BS_DEFPUSHBUTTON 的边框要粗一些。

当按钮的高度为文字字元高度的 7/4 倍时，按钮的外观看起来最好，其中文字字元由 BTNLOOK 使用；而按钮的宽度至少调节到文字的宽度再加上两个字元的宽度。

当滑鼠游标在按钮中时，按下滑鼠按键将使按钮用三维阴影重画自己，就好像真的被按下一样。放开滑鼠按键时，就恢复按钮的原貌，并向父视窗发送一个 WM_COMMAND 讯息和 BN_CLICKED 通知码。与其他按钮型态相似，当按钮拥有输入焦点时，在文字的周围就有虚线，按下及释放 Spacebar 键与按下及释放滑鼠按键具有相同的效果。

您可以通过给视窗发送 BM_SETSTATE 讯息来模拟按钮闪动。以下的操作将导致按钮被按下：

```
SendMessage (hwndButton, BM_SETSTATE, 1, 0) ;
```

下面的呼叫使按钮恢复正常：

```
SendMessage (hwndButton, BM_SETSTATE, 0, 0) ;
```

hwndButton 视窗代号是从 CreateWindow 呼叫传回的值。

您也可以向按键发送 BM_GETSTATE 讯息，子视窗控制项传回按钮目前的状态：如果按钮被按下，则传回 TRUE；如果按钮处於正常状态，则传回 FALSE。但是，绝大多数应用并不需要这一讯息。因为按钮不保留任何开/关资讯，所以 BM_SETCHECK 讯息和 BM_GETCHECK 讯息不会被用到。

核取方块

核取方块是一个文字方块，文字通常出现在核取方块的右边（如果您在建立按钮时指定了 BS_LEFTTEXT 样式，那么文字会出现在左边；您也许将用 BS_RIGHT 直接调整文字来组合此样式）。核取方块通常用於允许使用者对选项进行选择的应用程式中。核取方块的常用功能如同一个开关：单击框一次将显示勾选标记，再次单击清除勾选标记。

核取方块最常用的两种样式是 BS_CHECKBOX 和 BS_AUTOCHECKBOX。在使用 BS_CHECKBOX 时，您需要自己向该控制项发送 BM_SETCHECK 讯息来设定勾选标记。wParam 参数设 1 时设定勾选标记，设 0 时清除勾选标记。通过向该控制项发送 BM_GETCHECK 讯息，您可以得到该核取方块的目前状态。在处理来自控制项的 WM_COMMAND 讯息时，您可以用如下的指令来翻转 X 标记：

```
SendMessage             ((HWND) lParam, BM_SETCHECK, (WPARAM)
                !SendMessage ((HWND) lParam, BM_GETCHECK, 0, 0), 0) ;
```

注意第二个 SendMessage 呼叫前面的运算子「!」，其中 lParam 是在 WM_COMMAND 讯息中传给使用者视窗讯息处理程式的子视窗代号。如果您以後又想知道按钮的状态，那么可以向它发送另一条 BM_GETCHECK 讯息；您也可以将目前状态储存在您的视窗讯息处理程式中的一个静态变数里，或者向它发送 BM_SETCHECK 讯息来初始化带勾选标记的 BS_CHECKBOX 核取方块：

```
SendMessage (hwndButton, BM_SETCHECK, 1, 0) ;
```

对 BS_AUTOCHECKBOX 样式，按钮自己触发勾选标记的开和关，所以您的视窗讯息处理程式可以忽略 WM_COMMAND 讯息。当您需要按钮目前的状态时，可以向控制项发送 BM_GETCHECK 讯息：

```
iCheck = (int) SendMessage (hwndButton, BM_GETCHECK, 0, 0) ;
```

如果该按钮被选中，则 iCheck 的值为 TRUE 或者非零数；如果按钮末被选中，则 iCheck 的值为 FALSE 或 0。

其余两种核取方块样式是 BS_3STATE 和 BS_AUTO3STATE，正如它们名字所暗示的，这两种样式能显示第三种状态——核取方块内是灰色——它出现在向控制项发送 wParam 等於 2 的 WM_SETCHECK 讯息时。灰色是向使用者表示此框不能被选本章的或者禁止使用。

核取方块沿矩形的左边框对齐，并集中在呼叫 CreateWindow 时规定的矩形

的顶边和底边之间，在该矩形内的任何地方按下滑鼠都会向其父视窗发送一个 WM_COMMAND 讯息。核取方块的最小高度是一个字元的高度，最小宽度是文字中的字元数加 2。

单选按钮

单选按钮的名称在一列按钮的後面，这些按钮就像汽车上的收音机一样。汽车收音机上的每一个按钮都对应一种收音状态，而且一次只能有一个按钮被按下。在对话方块中，单选按钮组常常用来表示相互排斥的选项。与核取方块不同，单选按钮的工作与开关不一样，也就是说，当第二次按单选按钮时，它的状态会保持不变。

单选按钮的形状是一个圆圈，而不是方框，除此之外，它非常像核取方块。圆圈内的加重圆点表示该单选按钮已经被选中。单选按钮有视窗样式 BS_RADIOBUTTON 或 BS_AUTORADIOBUTTON 两种，但是後者只用於对话方块。

当您收到来自单选按钮的 WM_COMMAND 讯息时，应该向它发送 wParam 等於 1 的 BM_SETCHECK 讯息来显示其选中状态：

```
SendMessage (hwndButton, BM_SETCHECK, 1, 0) ;
```

对同组中的其他所有单选按钮，您可以通过向它们发送 wParam 等於 0 的 BM_SETCHECK 讯息来显示其未选中状态：

```
SendMessage (hwndButton, BM_SETCHECK, 0, 0) ;
```

分组方块

分组方块即样式为 BS_GROUPBOX 的选择框，它是按钮类中的特例，既不处理滑鼠输入和键盘输入，也不向其父视窗发送 WM_COMMAND 讯息。分组方块是一个矩形框，分组方块标题在其顶部显示。分组方块常用来包含其他的按钮控制项。

改变按钮文字

您可以通过 SetWindowText 来改变按钮（或者其他任何视窗）内的文字：

```
SetWindowText (hwnd, pszString) ;
```

其中 hwnd 是欲改变视窗的代号，pszString 是一个指向以 null 为终结的字串指标。对於一般的视窗来说，这个文字是标题列的文字；对於按钮控制项来说，它是随著该按钮显示的文字。

您也可以取得视窗目前的文字：

```
iLength = GetWindowText (hwnd, pszBuffer, iMaxLength) ;
```

iMaxLength 指定复制到 pszBuffer 指向的缓冲区中的最大字元数。该函式

传回复制的字元数。您可以首先通过下面的呼叫来获得特定文字的长度：

```
iLength = GetWindowTextLength (hwnd) ;
```

可见的和启用的按钮

为了接收滑鼠和键盘输入，子视窗必须是可见的（被显示）和被启用的。当视窗是可见的而未被启用时，那么视窗将以灰色而非黑色显示文字。

如果在建立子视窗时，您没有将 WS_VISIBLE 包含在视窗类别中，那么直到呼叫 ShowWindow 时子视窗才会被显示出来：

```
ShowWindow (hwndChild, SW_SHOWNORMAL) ;
```

如果您将 WS_VISIBLE 包含在视窗类别中，就没有必要呼叫 ShowWindow。但是，您可以通过呼叫 ShowWindow 将子视窗隐藏起来：

```
ShowWindow (hwndChild, SW_HIDE) ;
```

您可以通过下面的呼叫来确定子视窗是否可见：

```
IsWindowVisible (hwndChild) ;
```

您也可以使子视窗被启用或者不被启用。在内定情况下，视窗是被启用的。您可以通过下面的呼叫使视窗不被启用：

```
EnableWindow (hwndChild, FALSE) ;
```

对於按钮控制项，这具有使按钮字串变成灰色的作用。按钮将不再对滑鼠输入和键盘输入做出回应，这是表示按钮选项目前不可用的最好方法。

您可以通过下面的呼叫使子视窗再次被启用：

```
EnableWindow (hwndChild, TRUE) ;
```

您还可以使用下面的呼叫来确定子视窗是否被启用：

```
IsWindowEnabled (hwndChild) ;
```

按钮和输入焦点

我在本章前面已经提到过，当用滑鼠单击按钮、核取方块、单选框和拥有者绘制按钮时，它们接收到输入焦点。这些控制项使用文字周围的虚线来表示它拥有了输入焦点。当子视窗控制项得到输入焦点时，其父视窗就失去了输入焦点；所有的键盘输入都进入子视窗控制项，而不会进入父视窗中。但是，子视窗控制项只对 Spacebar 键作出回应，此时 Spacebar 键的作用就如同滑鼠按键一样。这种情形导致了一个明显的问题：您的程式失去了对键盘处理的控制项。让我们看看我们对此能做一些什么。

我在第六章中已经提到过，当 Windows 将输入焦点从一个视窗（例如一个父视窗）转换到另一个视窗（例如一个子视窗控制项）时，它首先给正在失去输入焦点的视窗发送一个 WM_KILLFOCUS 讯息，wParam 参数是接收输入焦点的视窗的代号。然後，Windows 向正在接收输入焦点的视窗发送一个 WM_SETFOCUS 讯

息，同时 wParam 是还在失去输入焦点的视窗的代号（在这两种情况中，wParam 值可能为 NULL，它表示没有视窗拥有或者正在接收输入焦点）。

通过处理 WM_KILLFOCUS 讯息，父视窗可以阻止子视窗控制项获得输入焦点。假定阵列 hwndChild 包含了所有子视窗的视窗代号（它们是在呼叫 CreateWindow 来建立视窗的时候储存到阵列中的）。 NUM 是子视窗的数目：

```
case  WM_KILLFOCUS :
            for ( i = 0 ; i < NUM ; i++)
                                    if (hwndChild [i] == (HWND) wParam)
                  {
                                          SetFocus (hwnd) ;
                                          break ;
                  }
      return 0 ;
```

在这段程式码中，当父视窗获知它正在失去输入焦点，而让它的某个子视窗得到输入焦点时，它将呼叫 SetFocus 来重新取得输入焦点。

下面是可达到相同目的、但更为简单（但不太直观）的方法：

```
case WM_KILLFOCUS :
    if (hwnd == GetParent ((HWND) wParam))
                        SetFocus (hwnd) ;
      return 0 ;
```

但是，这两种方法都有缺点：它们阻止按钮对 Spacebar 键作出回应，因为该按钮总是得不到输入焦点。一个更好的方法是使按钮得到输入焦点，也能让使用者用 Tab 键从一个按钮转移到另一个按钮。这听起来似乎不太可能，在本章的後面，我们将要说明在 COLORS1 程式中如何用「视窗子类别化」技术来实作这种方法。

控制项与颜色

您可以在图 9-1 中看到，许多按钮的显示看起来并不正确。按键还好，但是其他按钮却带有一个本不应该在那里的一个矩形灰色背景。这是因为这些按钮本来是为对话方块中的显示而设计的，而在 Windows 98 中，对话方块有一个灰色的表面。我们的视窗有一个白色的表面，这是因为我们在 WNDCLASS 结构中就是这样定义的。

```
wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
```

我们已经这么做了，因为我们经常在显示区域中显示文字，而 GDI 使用在内定装置内容中定义的文字颜色和背景颜色，它们总是黑色和白色。为了使这些按钮更加美观一些，我们必须要改变显示区域的颜色使之和按钮的背景颜色一致，所以要以某种方法将按钮的背景颜色改为白色。

解决此问题的第一步，是理解 Windows 对「系统颜色」的使用。

系统颜色

Windows 保留了 29 种系统颜色以供各种显示使用。您可以使用 GetSysColor 和 SetSysColors 来获得和设定这些颜色。在 Windows 表头档案中定义的识别字规定了系统颜色。使用 SetSysColors 设定的系统颜色只在目前 Windows 对话过程中有效。

借助 Windows「控制台」程式的「显示器」部分，您可以改变一些（但不是全部）系统颜色。若是 Microsoft Windows NT，选中的颜色会储存在系统登录中；若是 Microsoft Windows 98，则储存在 WIN.INI 档案中。系统登录和 WIN.INI 档案都为这 29 种系统颜色使用了关键字（与 GetSysColor 和 SetSysColors 的识别字不同），在系统颜色的後面跟著红、绿、蓝三种颜色的值，该值的变化范围是 0 到 255。下表说明了这 29 种系统颜色是如何在 GetSysColor、SetSysColors 以及 WIN.INI 关键字中用常数来标识的。这张表是按照 COLOR_ 常数值（从 0 开始到 28 结束）顺序排列的：

表 9-3

GetSysColor 和 SetSysColors	系统登录键或 WIN.INI 识别字	内定的 RGB 值
COLOR_SCROLLBAR	Scrollbar	C0-C0-C0
COLOR_BACKGROUND	Background	00-80-80
COLOR_ACTIVECAPTION	ActiveTitle	00-00-80
COLOR_INACTIVECAPTION	InactiveTitle	80-80-80
COLOR_MENU	Menu	C0-C0-C0
COLOR_WINDOW	Window	FF-FF-FF
COLOR_WINDOWFRAME	WindowFrame	00-00-00
COLOR_MENUTEXT	MenuText	C0-C0-C0
COLOR_WINDOWTEXT	WindowText	00-00-00
COLOR_CAPTIONTEXT	TitleText	FF-FF-FF
COLOR_ACTIVEBORDER	ActiveBorder	C0-C0-C0
COLOR_INACTIVEBORDER	InactiveBorder	C0-C0-C0
COLOR_APPWORKSPACE	AppWorkspace	80-80-80
COLOR_HIGHLIGHT	Highlight	00-00-80
COLOR_HIGHLIGHTTEXT	HighlightText	FF-FF-FF
COLOR_BTNFACE	ButtonFace	C0-C0-C0
COLOR_BTNSHADOW	ButtonShadow	80-80-80
COLOR_GRAYTEXT	GrayText	80-80-80
COLOR_BTNTEXT	ButtonText	00-00-00

COLOR_INACTIVECAPTIONTEXT	InactiveTitleText	C0-C0-C0
COLOR_BTNHIGHLIGHT	ButtonHighlight	FF-FF-FF
COLOR_3DDKSHADOW	ButtonDkShadow	00-00-00
COLOR_3DLIGHT	ButtonLight	C0-C0-C0
COLOR_INFOTEXT	InfoText	00-00-00
COLOR_INFOBK	InfoWindow	FF-FF-FF
[no identifier; use value 25]	ButtonAlternateFace	B8-B4-B8
COLOR_HOTLIGHT	HotTrackingColor	00-00-FF
COLOR_GRADIENTACTIVECAPTION	GradientActiveTitle	00-00-80
COLOR_GRADIENTINACTIVECAPTION	GradientInactiveTitle	80-80-80

这 29 种颜色的预设值是由显示驱动程式提供的，在不同的机器上可能略有不同。

坏消息：虽然这些颜色中有许多似乎都可以从颜色常数名称上了解其代表意义（例如，COLOR_BACKGROUND 是所有视窗後面的桌面区域颜色），在最近版本的 Windows 中系统颜色的使用变得非常混乱。以前，Windows 在视觉上要比今天简单得多。实际上，在 Windows 3.0 以前，只定义了前 13 种系统颜色。但随著使用看起来越来越难以控制的立体外观，相对应地也需要更多的系统颜色。

按钮颜色

对需要多种颜色的每一个按钮来说，这个问题更加地明显。COLOR_BTNFACE 被用於按键主要的表面颜色，以及其他按钮主要的背景颜色（这也是用於对话方块和讯息方块的系统颜色）。COLOR_BTNSHADOW 被建议用作按键右下边、以及核取方块内部和单选按钮圆点的阴影。对於按键，COLOR_BTNTEXT 被用作文字颜色；而对於其他的按钮，则使用 COLOR_WINDOWTEXT 作为文字颜色。还有其他几种系统颜色用於按钮设计的各个部分。

因此，如果您想在我们的显示区域表面显示按钮，那么一种避免颜色冲突的方法便是屈服於这些系统颜色。首先，在定义视窗类别时使用 COLOR_BTNFACE 作为您显示区域的背景颜色：

```
wndclass.hbrBackground = (HBRUSH) (COLOR_BTNFACE + 1) ;
```

您可以在 BTNLOOK 程式中尝试这种方法。当 WNDCLASS 结构中的 hbrBackground 值是这个值时，Windows 会明白这实际上指的是一种系统颜色而非一个实际的代号。Windows 要求当您在 WNDCLASS 结构的 hbrBackground 栏中指定这些识别字时加上 1，这样做的目的是防止其值为 NULL，而没有任何其他目的。如果您的在程式执行过程中，系统颜色恰好发生了变化，那么显示区域

将变得无效，而 Windows 将使用新的 COLOR_BTNFACE 值。但是现在我们又引发了另一个问题。当您使用 TextOut 显示文字时，Windows 使用的是在装置内容中为背景颜色（它擦除文字後的背景）和文字颜色定义的值，其预设值为白色（背景）和黑色（文字），而不管系统颜色和视窗类别结构中的 hbrBackground 栏位为何值。所以，您需要使用 SetTextColor 和 SetBkColor 将文字和文字背景的颜色改变为系统颜色。您可以在获得装置内容代号之後这么做：

```
SetBkColor (hdc, GetSysColor (COLOR_BTNFACE)) ;
SetTextColor (hdc, GetSysColor (COLOR_WINDOWTEXT)) ;
```

这样，显示区域背景、文字背景和文字的颜色都与按钮的颜色一致了。但是，如果当您的程式执行时，使用者改变了系统颜色，您可能要改变文字背景颜色和文字颜色。这时您可以使用下面的程式码：

```
case  WM_SYSCOLORCHANGE:
      InvalidateRect (hwnd, NULL, TRUE) ;
      break ;
```

WM_CTLCOLORBTN 讯息

在这边已经看到了如何将显示区域的颜色和文字颜色调节成按钮的背景颜色。我们是否可以将程式中按钮的颜色调节为我们喜欢的颜色呢？理论上没有问题，但在实际中请别这样做。用 SetSysColors 来改变按钮的外观可能不是您想做的，这会影响目前在 Windows 下执行的所有程式，这也是使用者不太喜欢的。

更好的方法（同样也只是理论上）是处理 WM_CTLCOLORBTN 讯息，这是当子视窗即将为其显示区域著色时，由按钮控制项发送给其父视窗讯息处理程式的一个讯息。父视窗可以利用这个机会来改变子视窗讯息处理程式将用来著色的颜色（在 Windows 的 16 位元版本中，一个称为 WM_CTLCOLOR 的讯息被用於所有的控制项，现在针对每种型态的标准控制项，分别代之以不同的讯息）。

当父视窗讯息处理程式收到 WM_CTLCOLORBTN 讯息时，wParam 讯息参数是按钮的装置内容代号，lParam 是按钮的视窗代号。当父视窗讯息处理程式得到这个讯息时，按钮控制项已经获得了它的装置内容。当您的视窗讯息处理程式处理一个 WM_CTLCOLORBTN 讯息时，您必须完成以下三个动作：

- 使用 SetTextColor 选择设定一种文字颜色。
- 使用 SetBkColor 选择设定一种文字背景颜色。
- 将一个画刷代号传回给子视窗。

理论上，子视窗使用该画刷来著色背景。当不再需要这个画刷时，您应该负责清除它。

下面是使用 WM_CTLCOLORBTN 的问题所在：只有按键和拥有者绘制按钮才给其父视窗发送 WM_CTLCOLORBTN，而只有拥有者绘制按钮才会回应父视窗讯息处理程式对讯息的处理，而使用画刷来著色背景。这基本上是没有意义的，因为无论怎样都是由父视窗来负责绘制拥有者绘制按钮。

在本章後面，我们将说明，在某些情况下，一些类似於 WM_CTLCOLORBTN 但适用於其他型态控制项的讯息将更为有用。

拥有者绘制按钮

如果您想对按钮的所有可见部分实行全面控制，而不想被键盘和滑鼠讯息处理所干扰，那么您可以建立 BS_OWNERDRAW 样式的按钮，如程式 9-2 所展示的那样。

程式 9-2　OWNDRAW

```
OWNDRAW.C
/*------------------------------------------------------------------------
   OWNDRAW.C --   Owner-Draw Button Demo Program
                                (c) Charles Petzold, 1996
------------------------------------------------------------------------*/

#include <windows.h>

#define ID_SMALLER                                      1
#define ID_LARGER                                  2
#define BTN_WIDTH                                  (     8 * cxChar)
#define BTN_HEIGHT                                 (     4 * cyChar)

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
HINSTANCE hInst ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                                PSTR szCmdLine, int iCmdShow)
{
      static TCHAR     szAppName[] = TEXT ("OwnDraw") ;
      MSG                           msg ;
      HWND                                hwnd ;
      WNDCLASS                      wndclass ;

      hInst = hInstance ;
      wndclass.style                      = CS_HREDRAW | CS_VREDRAW ;
      wndclass.lpfnWndProc                = WndProc ;
      wndclass.cbClsExtra                 = 0 ;
      wndclass.cbWndExtra                 = 0 ;
      wndclass.hInstance                  = hInstance ;
      wndclass.hIcon                      = LoadIcon (NULL, IDI_APPLICATION) ;
      wndclass.hCursor                    = LoadCursor (NULL, IDC_ARROW) ;
```

```
      wndclass.hbrBackground      = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
      wndclass.lpszMenuName       = szAppName ;
      wndclass.lpszClassName      = szAppName ;

      if (!RegisterClass (&wndclass))
      {
                  MessageBox (     NULL, TEXT ("This program requires Windows
NT!"),
                                                              szAppName,
MB_ICONERROR) ;
                  return 0 ;
      }

      hwnd = CreateWindow (  szAppName, TEXT ("Owner-Draw Button Demo"),
                     WS_OVERLAPPEDWINDOW,
                     CW_USEDEFAULT, CW_USEDEFAULT,
                     CW_USEDEFAULT, CW_USEDEFAULT,
                     NULL, NULL, hInstance, NULL) ;

      ShowWindow (hwnd, iCmdShow) ;
      UpdateWindow (hwnd) ;

      while (GetMessage (&msg, NULL, 0, 0))
      {
                  TranslateMessage (&msg) ;
                  DispatchMessage (&msg) ;
     }
      return msg.wParam ;
}

void Triangle (HDC hdc, POINT pt[])
{
      SelectObject (hdc, GetStockObject (BLACK_BRUSH)) ;
      Polygon (hdc, pt, 3) ;
      SelectObject (hdc, GetStockObject (WHITE_BRUSH)) ;
}

LRESULT CALLBACK WndProc (   HWND  hwnd,  UINT  message,  WPARAM  wParam,LPARAM
lParam)
{
      static HWND                 hwndSmaller, hwndLarger ;
      static int                        cxClient, cyClient, cxChar, cyChar ;
      int                               cx, cy ;
      LPDRAWITEMSTRUCT pdis ;
      POINT                             pt[3] ;
      RECT                                    rc ;

      switch (message)
```

```
     {
     case  WM_CREATE :
                cxChar = LOWORD (GetDialogBaseUnits ()) ;
                cyChar = HIWORD (GetDialogBaseUnits ()) ;

                          // Create the owner-draw pushbuttons

                hwndSmaller = CreateWindow (TEXT ("button"), TEXT (""),
                        WS_CHILD | WS_VISIBLE | BS_OWNERDRAW,
                         0, 0, BTN_WIDTH, BTN_HEIGHT,
                         hwnd, (HMENU) ID_SMALLER, hInst, NULL) ;

                hwndLarger  = CreateWindow (TEXT ("button"), TEXT (""),
                        WS_CHILD | WS_VISIBLE | BS_OWNERDRAW,
                         0, 0, BTN_WIDTH, BTN_HEIGHT,
                         hwnd, (HMENU) ID_LARGER, hInst, NULL) ;
                return 0 ;

     case  WM_SIZE :
                cxClient = LOWORD (lParam) ;
                cyClient = HIWORD (lParam) ;
                          // Move the buttons to the new center

          MoveWindow (     hwndSmaller,      cxClient    /    2    -    3    *
     BTN_WIDTH  / 2,
                           cyClient / 2 -      BTN_HEIGHT / 2,
                                  BTN_WIDTH, BTN_HEIGHT, TRUE) ;
                MoveWindow (      hwndLarger, cxClient / 2 +    BTN_WIDTH    /
2,cyClient / 2 -  BTN_HEIGHT / 2,
                                        BTN_WIDTH, BTN_HEIGHT, TRUE) ;
                return 0 ;

     case  WM_COMMAND :
                GetWindowRect (hwnd, &rc) ;

                          // Make the window 10% smaller or larger

                switch (wParam)
                {
                case  ID_SMALLER :
                          rc.left     += cxClient / 20 ;
                          rc.right    -= cxClient / 20 ;
                          rc.top           += cyClient / 20 ;
                          rc.bottom   -= cyClient / 20 ;
                          break ;

                case  ID_LARGER :
                          rc.left     -= cxClient / 20 ;
```

```
                                rc.right    += cxClient / 20 ;
                                rc.top           -= cyClient / 20 ;
                                rc.bottom   += cyClient / 20 ;
                                break ;
          }

                MoveWindow (     hwnd, rc.left, rc.top, rc.right  - rc.left,
                                                                rc.bottom    -
rc.top, TRUE) ;
                return 0 ;

     case  WM_DRAWITEM :
                pdis = (LPDRAWITEMSTRUCT) lParam ;

                                // Fill area with white and frame it black

                FillRect          (pdis->hDC, &pdis->rcItem,
                            (HBRUSH) GetStockObject (WHITE_BRUSH)) ;

                FrameRect ( pdis->hDC, &pdis->rcItem,
                                (     HBRUSH)               GetStockObject
(BLACK_BRUSH)) ;
                                //               Draw  inward  and  outward
black triangles
                cx =        pdis->rcItem.right  - pdis->rcItem.left ;
                cy =        pdis->rcItem.bottom - pdis->rcItem.top  ;

                switch (pdis->CtlID)
                {
                case  ID_SMALLER :
                                pt[0].x = 3 * cx / 8 ;  pt[0].y = 1 * cy / 8 ;
                                pt[1].x = 5 * cx / 8 ;  pt[1].y = 1 * cy / 8 ;
                                pt[2].x = 4 * cx / 8 ;  pt[2].y = 3 * cy / 8 ;

                                Triangle (pdis->hDC, pt) ;

                                pt[0].x = 7 * cx / 8 ;  pt[0].y = 3 * cy / 8 ;
                                pt[1].x = 7 * cx / 8 ;  pt[1].y = 5 * cy / 8 ;
                                pt[2].x = 5 * cx / 8 ;  pt[2].y = 4 * cy / 8 ;

                                Triangle (pdis->hDC, pt) ;

                                pt[0].x = 5 * cx / 8 ;  pt[0].y = 7 * cy / 8 ;
                                pt[1].x = 3 * cx / 8 ;  pt[1].y = 7 * cy / 8 ;
                                pt[2].x = 4 * cx / 8 ;  pt[2].y = 5 * cy / 8 ;

                                Triangle (pdis->hDC, pt) ;
```

```
                              pt[0].x = 1 * cx / 8 ;  pt[0].y = 5 * cy / 8 ;
                              pt[1].x = 1 * cx / 8 ;  pt[1].y = 3 * cy / 8 ;
                              pt[2].x = 3 * cx / 8 ;  pt[2].y = 4 * cy / 8 ;

                         Triangle (pdis->hDC, pt) ;
                         break ;

                    case ID_LARGER :
                         pt[0].x = 5 * cx / 8 ;  pt[0].y = 3 * cy / 8 ;
                         pt[1].x = 3 * cx / 8 ;  pt[1].y = 3 * cy / 8 ;
                         pt[2].x = 4 * cx / 8 ;  pt[2].y = 1 * cy / 8 ;

                         Triangle (pdis->hDC, pt) ;

                         pt[0].x = 5 * cx / 8 ;  pt[0].y = 5 * cy / 8 ;
                         pt[1].x = 5 * cx / 8 ;  pt[1].y = 3 * cy / 8 ;
                         pt[2].x = 7 * cx / 8 ;  pt[2].y = 4 * cy / 8 ;

                         Triangle (pdis->hDC, pt) ;
                         pt[0].x = 3 * cx / 8 ;  pt[0].y = 5 * cy / 8 ;
                         pt[1].x = 5 * cx / 8 ;  pt[1].y = 5 * cy / 8 ;
                         pt[2].x = 4 * cx / 8 ;  pt[2].y = 7 * cy / 8 ;

                         Triangle (pdis->hDC, pt) ;
                         pt[0].x = 3 * cx / 8 ;  pt[0].y = 3 * cy / 8 ;
                         pt[1].x = 3 * cx / 8 ;  pt[1].y = 5 * cy / 8 ;
                         pt[2].x = 1 * cx / 8 ;  pt[2].y = 4 * cy / 8 ;

                         Triangle (pdis->hDC, pt) ;
                         break ;
               }

                         // Invert the rectangle if the button is selected

               if (pdis->itemState & ODS_SELECTED)
                              InvertRect (pdis->hDC, &pdis->rcItem) ;

                         // Draw a focus rectangle if the button has the focus

               if (pdis->itemState & ODS_FOCUS)
               {
                              pdis->rcItem.left  += cx / 16 ;
                              pdis->rcItem.top   += cy / 16 ;
                              pdis->rcItem.right -= cx / 16 ;
                              pdis->rcItem.bottom-= cy / 16 ;

                              DrawFocusRect (pdis->hDC, &pdis->rcItem) ;
               }
```

```
                return 0 ;

     case  WM_DESTROY :
                PostQuitMessage (0) ;
                return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

该程式在其显示区域的中央包含了两个按钮，如图 9-2 所示。左边的按钮有四个三角形指向按钮的中央，按下该按钮时，视窗的尺寸将缩小 10%。右边的按钮有四个向外指的三角形，按下此按钮时，视窗的尺寸将增大 10%。

如果您只需要在按钮中显示图示或点阵图，您可以用 BS_ICON 或 BS_BITMAP 样式，并用 BM_SETIMAGE 讯息设定点阵图。但是，对於 BS_OWNERDRAW 样式的按钮，它允许完全自由地绘制按钮。

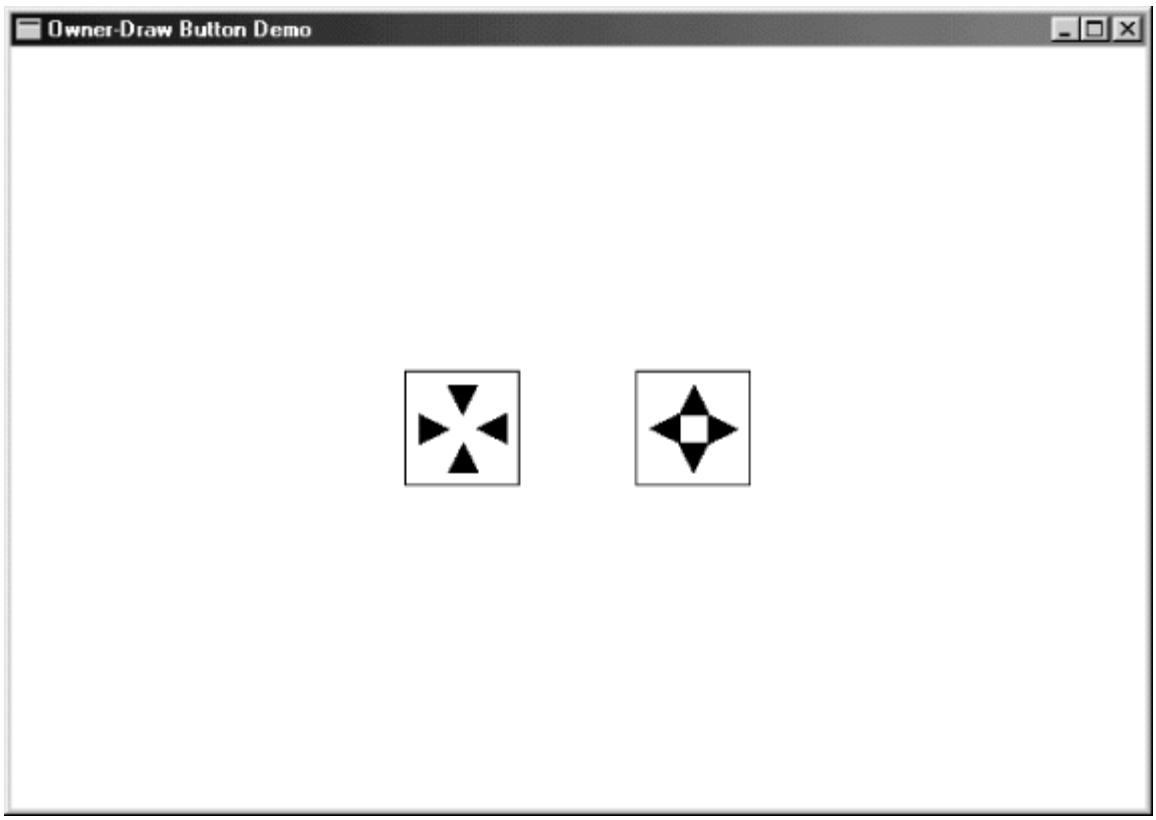

图 9-2 OWNDRAW 的萤幕显示

在处理 WM_CREATE 讯息处理期间，OWNDRAW 建立了两个 BS_OWNERDRAW 样式的按钮；按钮的宽度是系统字体的 8 倍，高度是系统字体的 4 倍（在使用预先定义好的点阵图绘制按钮时，这些尺寸在 VGA 上建立的按钮为 64 图素宽 64 图素高，知道这些资料将非常有用）。这些按钮尚未就定位，在处理 WM_SIZE 讯息处理期间，通过呼叫 MoveWindow 函式，OWNDRAW 将按钮位置放在显示区域的中心。

按下这些按钮时，它们就会产生 WM_COMMAND 讯息。为了处理这些 WM_COMMAND

讯息，OWNDRAW 呼叫 GetWindowRect，将整个视窗（不只是显示区域）的位置和尺寸存放在 RECT（矩形）结构中，这个位置是相对於萤幕的。然後，根据按下的是左边还是右边的按钮，OWNDRAW 调节这个矩形结构的各个栏位值。程式再通过呼叫 MoveWindow 来重新确定位置和尺寸。这将产生另一个 WM_SIZE 讯息，按钮被重新定位在显示区域的中央。

如果这是程式所做的全部处理，那么这完全可以，只不过按钮是不可见的。使用 BS_OWNERDRAW 样式建立的按钮会在需要重新著色的任何时候都向它的父视窗发送一个 WM_DRAWITEM 讯息。这出现在以下几种情况中：当按钮被建立时，当按钮被按下或被放开时，当按钮得到或者失去输入焦点时，以及当按钮需要重新著色的任何时候。

在处理 WM_DRAWITEM 讯息处理期间，lParam 讯息参数是指向型态 DRAWITEMSTRUCT 结构的指标，OWNDRAW 程式将这个指标储存在 pdis 变数中，这个结构包含了画该按钮时程式所必需的讯息（这个结构也可以让自绘清单方块和功能表使用）。对按钮而言非常重要的结构栏位有 hDC（按钮的装置内容）、rcItem（提供按钮尺寸的 RECT 结构）、CtlID（控制项视窗 ID）和 itemState（它说明按钮是否被按下，或者按钮是否拥有输入焦点）。

呼叫 FillRect 用白色画刷抹掉按钮的内面，呼叫 FrameRect 在按钮的周围画上黑框，由此 OWNDRAW 便启动了 WM_DRAWITEM 处理过程。然後，通过呼叫 Polygon，OWNDRAW 在按钮上画出 4 个黑色实心的三角形。这是一般的情形。

如果按钮目前被按下，那么 DRAWITEMSTRUCT 的 itemState 栏位中的某位元将被设为 1。您可以使用 ODS_SELECTED 常数来测试这些位元。如果这些位元被设立，那么 OWNDRAW 将通过呼叫 InvertRect 将按钮翻转为相反的颜色。如果按钮拥有输入焦点，那么 itemState 的 ODS_FOCUS 位元将被设立。在这种情况下，OWNDRAW 通过呼叫 DrawFocusRect，在按钮的边界内画一个虚线的矩形。

在使用拥有者绘制按钮时，应该注意以下几个方面：Windows 获得装置内容并将其作为 DRAWITEMSTRUCT 结构的一个栏位。保持装置内容处於您找到它时所处的状态，任何被选进装置内容的 GDI 物件都必需被释放。另外，当心不要在定义按钮边界的矩形外面进行绘制。

静态类别

在 CreateWindow 函式中指定视窗类别为「static」，您就可以建立静态文字的子视窗控制项。这些子视窗非常「文静」。它既不接收滑鼠或键盘输入，也不向父视窗发送 WM_COMMAND 讯息。

当您在静态子视窗上移动或者按下滑鼠时，这个子视窗将拦截

WM_NCHITTEST 讯息并将 HTTRANSPARENT 的值传回给 Windows，这将使 Windows 向其下层视窗，通常是它的父视窗，发送相同的 WM_NCHITTEST 讯息。父视窗常常将该讯息传递给 DefWindowProc，在这里，它被转换为显示区域的滑鼠讯息。

前六个静态视窗样式只简单地在子视窗的显示区域内画一个矩形或者边框。在下表的上部，「RECT」静态样式（左列）是填入图样的矩形样式；三个「FRAME」样式（右列）是没有填入图样的矩形轮廓：

「RECT」静态样式	「FRAME」样式
SS_BLACKRECT	SS_BLACKFRAME
SS_GRAYRECT	SS_GRAYFRAME
SS_WHITERECT	SS_WHITEFRAME

「BLACK」、「GRAY」、「WHITE」并不意味著黑、灰和白色，这些颜色是由系统颜色决定的，如表 9-4 所示。

表 9-4

静态控制项	系统颜色
BLACK	COLOR_3DDKSHADOW
GRAY	COLOR_BTNSHADOW
WHITE	COLOR_BTNHIGHLIGHT

对这些样式，CreateWindow 呼叫中的视窗文字栏位被忽略。矩形的左上角开始於 x 位置座标和 y 位置座标，这些座标都相对於父视窗。您也可以使用 SS_ETCHEDHORZ、SS_ETCHEDVERT 或者 SS_ETCHEDFRAME ，采用灰色和白色建立一个形似阴影的边框。

静态类别也包括了三种文字样式：SS_LEFT、SS_RIGHT 和 SS_CENTER。它们建立左对齐、置右对齐和居中文字。文字在 CreateWindow 呼叫的视窗文字参数中给出，并且在以後可以用 SetWindowText 来改变它。当静态控制项的视窗讯息处理程式显示文字时，它使用 DrawText 函式以及 DT_WORDBREAK、DT_NOCLIP 和 DT_EXPANDTABS 参数。文字在子视窗的矩形内可以按文字进行换行。

这三种文字样式子视窗的背景通常为 COLOR_BTNFACE，而文字本身是 COLOR_WINDOWTEXT。在拦截 WM_CTLCOLORSTATIC 讯息时，您可以通过呼叫 SetTextColor 来改变文字颜色，通过 SetBkColor 来改变背景颜色，并传回背景画刷代号。後面的 COLORS1 程式展示了这一点。

最後，静态类别还包括了视窗样式 SS_ICON 和 SS_USERITEM，但是当它们被用作子视窗控制项时却没有任何意义。我们在讨论对话方块时还要提及它们。

卷动列类别

我在第四章首次讨论了卷动列，也讨论了「视窗卷动列」和「卷动列控制项」之间的一些区别。SYSMETS 程式使用视窗卷动列，它出现在视窗的右边和底部。您可以在建立视窗时通过将识别字 WS_VSCROLL、WS_HSCROLL 或者两者都包含在视窗样式中，让视窗加上卷动列。现在我们准备建立一些卷动列控制项，它们是能在父视窗的显示区域的任何地方出现的子视窗。您可以使用预先定义的视窗类别「scrollbar」以及两个卷动列样式 SBS_VERT 和 SBS_HORZ 中的一个来建立子视窗卷动列控制项。

与按钮控制项（以及将在後面讨论的编辑和清单方块控制项）不同，卷动列控制项不向父视窗发送 WM_COMMAND 讯息，而是像视窗卷动列那样发送 WM_VSCROLL 和 WM_HSCROLL 讯息。在处理卷动讯息时，您可以通过 lParam 参数来区分视窗卷动列与卷动列控制项。对子视窗卷动列其值为 0，对於卷动列控制项其值为卷动列视窗代号。对视窗卷动列和卷动列控制项来说，wParam 参数的高字组和低字组的含义相同。

虽然视窗卷动列有固定的宽度，Windows 使用 CreateWindow 呼叫中（或者在後面的 MoveWindow 呼叫中）给定的矩形尺寸来确定卷动列控制项的尺寸。您可以建立细而长的卷动列控制项，也可以建立短而粗的卷动列控制项。

如果您想建立与视窗卷动列尺寸相同的卷动列控制项，那么可以使用 GetSystemMetrics 取得水平卷动列的高度：

```
GetSystemMetrics (SM_CYHSCROLL) ;
```

或者垂直卷动列的宽度：

```
GetSystemMetrics (SM_CXVSCROLL) ;
```

根据 Windows 文件，卷动列窗样式识别字 SBS_LEFTALIGN、SBS_RIGHTALIGN、SBS_TOP ALIGN 和 SBS_BOTTOMALIGN 给出卷动列的标准尺寸，但是这些样式只在对话方块中对卷动列有效。

对视窗卷动列，您可以使用同样的呼叫来建立卷动列控制项的范围和位置：

```
SetScrollRange (hwndScroll, SB_CTL, iMin, iMax, bRedraw) ;
SetScrollPos (hwndScroll, SB_CTL, iPos, bRedraw) ;
SetScrollInfo (hwndScroll, SB_CTL, &si, bRedraw) ;
```

其区别在於：视窗卷动列将父视窗的代号作为第一个参数，并且以 SB_VERT 或者 SB_HORZ 作为第二个参数。

令人吃惊的是，名为 COLOR_SCROLLBAR 的系统颜色不再用於卷动列。两端的按钮和小方块的颜色由 COLOR_BTNFACE、COLOR_BTNHILIGHT、COLOR_BTNSHADOW、COLOR_BTNTEXT（用於小箭头）、COLOR_DKSHADOW 和 COLOR_BTNLIGHT 决定。两端按钮之间区域的颜色由 COLOR_BTNFACE 和

COLOR_BTNHIGHLIGHT 决定。

如果您拦截了 WM_CTLCOLORSCROLLBAR 讯息，那么可以在讯息处理中传回画刷以取代该颜色。让我们来试一下。

COLORS1 程式

为了解卷动列和静态子视窗的一些用法——也为了深入了解颜色——我们将使用 COLORS1 程式，如程式 9-3 所示。COLORS1 在显示区域的左半部显示三种卷动列，并分别标以「Red」、「 Green」和「Blue」。当您挪动卷动列时，显示区域的右半部将变为三种原色混合而成的合成色，三种原色的数值显示在三个卷动列的下面。

程式 9-3 COLORS1

```
COLORS1.C
/*------------------------------------------------------------------------
   COLORS1.C -- Colors Using Scroll Bars
                            (c) Charles Petzold, 1998
------------------------------------------------------------------------*/

#include <windows.h>
LRESULT CALLBACK WndProc          (HWND, UINT, WPARAM, LPARAM) ;
LRESULT CALLBACK ScrollProc  (HWND, UINT, WPARAM, LPARAM) ;

int    idFocus ;
WNDPROC OldScroll[3] ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                                              PSTR szCmdLine, int iCmdShow)
{
      static TCHAR     szAppName[] = TEXT ("Colors1") ;
      HWND                                hwnd ;
      MSG                           msg ;
      WNDCLASS                      wndclass ;

      wndclass.style                      = CS_HREDRAW | CS_VREDRAW ;
      wndclass.lpfnWndProc                = WndProc ;
      wndclass.cbClsExtra                 = 0 ;
      wndclass.cbWndExtra                 = 0 ;
      wndclass.hInstance                  = hInstance ;
      wndclass.hIcon                      = LoadIcon (NULL, IDI_APPLICATION) ;
      wndclass.hCursor                    = LoadCursor (NULL, IDC_ARROW) ;
      wndclass.hbrBackground        = CreateSolidBrush (0) ;
      wndclass.lpszMenuName         = NULL ;
      wndclass.lpszClassName        = szAppName ;
```

```
     if (!RegisterClass (&wndclass))
     {
                MessageBox (     NULL, TEXT ("This program requires Windows
NT!"),
                                        szAppName, MB_ICONERROR) ;
                return 0 ;
     }
     hwnd = CreateWindow (  szAppName, TEXT ("Color Scroll"),
                  WS_OVERLAPPEDWINDOW,
                  CW_USEDEFAULT, CW_USEDEFAULT,
                  CW_USEDEFAULT, CW_USEDEFAULT,
                  NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
                TranslateMessage (&msg) ;
                DispatchMessage  (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (   HWND  hwnd,  UINT  message,  WPARAM  wParam,LPARAM
lParam)
{
     static COLORREF crPrim[3] = {     RGB (255, 0, 0), RGB (0, 255, 0),
            RGB (0, 0, 255) } ;
     static HBRUSH          hBrush[3], hBrushStatic ;
     static HWND            hwndScroll[3],    hwndLabel[3],    hwndValue[3],
hwndRect ;
     static int             color[3], cyChar ;
     static RECT            rcColor ;
     static TCHAR *   szColorLabel[] = {     TEXT ("Red"), TEXT ("Green"),
      TEXT ("Blue") } ;
     HINSTANCE                    hInstance ;
     int                          i, cxClient, cyClient ;
     TCHAR                        szBuffer[10] ;

     switch (message)
     {
     case  WM_CREATE :
                hInstance = (HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE) ;

                // Create the white-rectangle window against which the
                // scroll bars will be positioned. The child window ID is 9.
```

```
                hwndRect = CreateWindow (TEXT ("static"), NULL,
                   WS_CHILD | WS_VISIBLE | SS_WHITERECT,
                   0, 0, 0, 0,
                   hwnd, (HMENU) 9, hInstance, NULL) ;

                for (i = 0 ; i < 3 ; i++)
                {
                            // The three scroll bars have IDs 0, 1, and 2, with
                            // scroll bar ranges from 0 through 255.

                hwndScroll[i] = CreateWindow (TEXT ("scrollbar"), NULL,
                          WS_CHILD | WS_VISIBLE |
                          WS_TABSTOP | SBS_VERT,
                            0, 0, 0, 0,
            hwnd, (HMENU) i, hInstance, NULL) ;

                       SetScrollRange (hwndScroll[i], SB_CTL, 0, 255, FALSE) ;
                       SetScrollPos   (hwndScroll[i], SB_CTL, 0, FALSE) ;

     // The three color-name labels have IDs 3, 4, and 5,
    // and text strings "Red", "Green", and "Blue".

     hwndLabel [i] = CreateWindow (TEXT ("static"), zColorLabel[i],
                              WS_CHILD | WS_VISIBLE | SS_CENTER,
                              0, 0, 0, 0,
                              hwnd, (HMENU) (i + 3),
                              hInstance, NULL) ;

                       // The three color-value text fields have IDs 6, 7,
                       // and 8, and initial text strings of "0".

hwndValue [i] = CreateWindow (TEXT ("static"), TEXT ("0"),
                           WS_CHILD | WS_VISIBLE | SS_CENTER,
                           0, 0, 0, 0,
                           hwnd, (HMENU) (i + 6),
                           hInstance, NULL) ;

                       OldScroll[i] = (WNDPROC) SetWindowLong (hwndScroll[i],
                            GWL_WNDPROC, (LONG) ScrollProc) ;

                       hBrush[i] = CreateSolidBrush (crPrim[i]) ;
                }

                   hBrushStatic = CreateSolidBrush (
                          GetSysColor (COLOR_BTNHIGHLIGHT)) ;

           cyChar = HIWORD (GetDialogBaseUnits ()) ;
           return 0 ;
```

```
case  WM_SIZE :
            cxClient = LOWORD (lParam) ;
            cyClient = HIWORD (lParam) ;
            SetRect (&rcColor, cxClient / 2, 0, cxClient, cyClient) ;

            MoveWindow (hwndRect, 0, 0, cxClient / 2, cyClient, TRUE) ;

            for (i = 0 ; i < 3 ; i++)
{
                  MoveWindow (hwndScroll[i],
                        (2 * i + 1) * cxClient / 14, 2 * cyChar,
                        cxClient / 14, cyClient - 4 * cyChar, TRUE) ;

                  MoveWindow (hwndLabel[i],
                        (4 * i + 1) * cxClient / 28, cyChar / 2,
                        cxClient / 7, cyChar, TRUE)

                  MoveWindow (hwndValue[i],
                        (4 * i + 1) * cxClient / 28,
                        cyClient - 3 * cyChar / 2,
                        cxClient / 7, cyChar, TRUE) ;
      }
            SetFocus (hwnd) ;
            return 0 ;

case  WM_SETFOCUS :
            SetFocus (hwndScroll[idFocus]) ;
            return 0 ;

case  WM_VSCROLL :
            i = GetWindowLong ((HWND) lParam, GWL_ID) ;

            switch (LOWORD (wParam))
            {
            case  SB_PAGEDOWN :
                              color[i] += 15 ;
                  // fall through
            case  SB_LINEDOWN :
                              color[i] = min (255, color[i] + 1) ;
                              break ;

            case  SB_PAGEUP :
                              color[i] -= 15 ;
                  // fall through
            case  SB_LINEUP :
                              color[i] = max (0, color[i] - 1) ;
                              break ;
```

```
                case  SB_TOP :
                                color[i] = 0 ;
                                break ;

                case  SB_BOTTOM :
                                color[i] = 255 ;
                                break ;

                case  SB_THUMBPOSITION :
                case  SB_THUMBTRACK :
                                color[i] = HIWORD (wParam) ;
                                break ;

                default :
                                break ;
                }
                SetScrollPos  (hwndScroll[i], SB_CTL, color[i], TRUE) ;
                wsprintf (szBuffer, TEXT ("%i"), color[i]) ;
                SetWindowText (hwndValue[i], szBuffer) ;

                DeleteObject ((HBRUSH)
                            SetClassLong (hwnd, GCL_HBRBACKGROUND, (LONG)
              CreateSolidBrush (RGB (color[0], color[1], color[2])))) ;

                InvalidateRect (hwnd, &rcColor, TRUE) ;
                return 0 ;

     case  WM_CTLCOLORSCROLLBAR :
                i = GetWindowLong ((HWND) lParam, GWL_ID) ;
                return (LRESULT) hBrush[i] ;

     case  WM_CTLCOLORSTATIC :
                i = GetWindowLong ((HWND) lParam, GWL_ID) ;

     if (i >= 3 && i <= 8)                    // static text controls
          {
                        SetTextColor ((HDC) wParam, crPrim[i % 3]) ;
                        SetBkColor       ((HDC)       wParam,       GetSysColor
(COLOR_BTNHIGHLIGHT));
                        return (LRESULT) hBrushStatic ;
          }
          break ;
     case  WM_SYSCOLORCHANGE :
                DeleteObject (hBrushStatic) ;
                hBrushStatic               =               CreateSolidBrush
(GetSysColor(COLOR_BTNHIGHLIGHT)) ;
                return 0 ;
```

```
      case  WM_DESTROY :
                  DeleteObject ((HBRUSH)
                              SetClassLong (hwnd, GCL_HBRBACKGROUND, (LONG)
                                    GetStockObject (WHITE_BRUSH))) ;

                  for (i = 0 ; i < 3 ; i++)
                                    DeleteObject (hBrush[i]) ;

                  DeleteObject (hBrushStatic) ;
                  PostQuitMessage (0) ;
                  return 0 ;
      }
      return DefWindowProc (hwnd, message, wParam, lParam) ;
}

LRESULT CALLBACK ScrollProc (HWND hwnd, UINT message,
                        WPARAM wParam, LPARAM lParam)
{
      int id = GetWindowLong (hwnd, GWL_ID) ;
      switch (message)
      {
      case WM_KEYDOWN :
                  if (wParam == VK_TAB)
                                    SetFocus (GetDlgItem (GetParent (hwnd),
                  (id + (GetKeyState (VK_SHIFT) < 0 ? 2 : 1)) % 3)) ;
                  break ;
      case  WM_SETFOCUS :
                  idFocus = id ;
                  break ;
      }
      return CallWindowProc (OldScroll[id], hwnd, message, wParam,lParam) ;
}
```

COLORS1 利用子视窗进行工作，该程式使用 10 个子视窗控制项：3 个卷动列、6 个静态文字视窗和 1 个静态矩形框。COLORS1 拦截 WM_CTLCOLORSCROLLBAR 讯息来给红、绿、蓝 3 个卷动列的内部著色，并拦截 WM_CTLCOLORSTATIC 讯息来著色静态文字。

您可以使用滑鼠或者键盘来挪动卷动列，从而利用 COLORS1 作为一种实验颜色显示的开发工具，为您自己的 Windows 程式选择漂亮的颜色（或者，您可能更喜欢难看的颜色）。COLORS1 的显示如图 9-3 所示。不幸的是，这些颜色在印表纸上被显示为不同深浅的灰色。

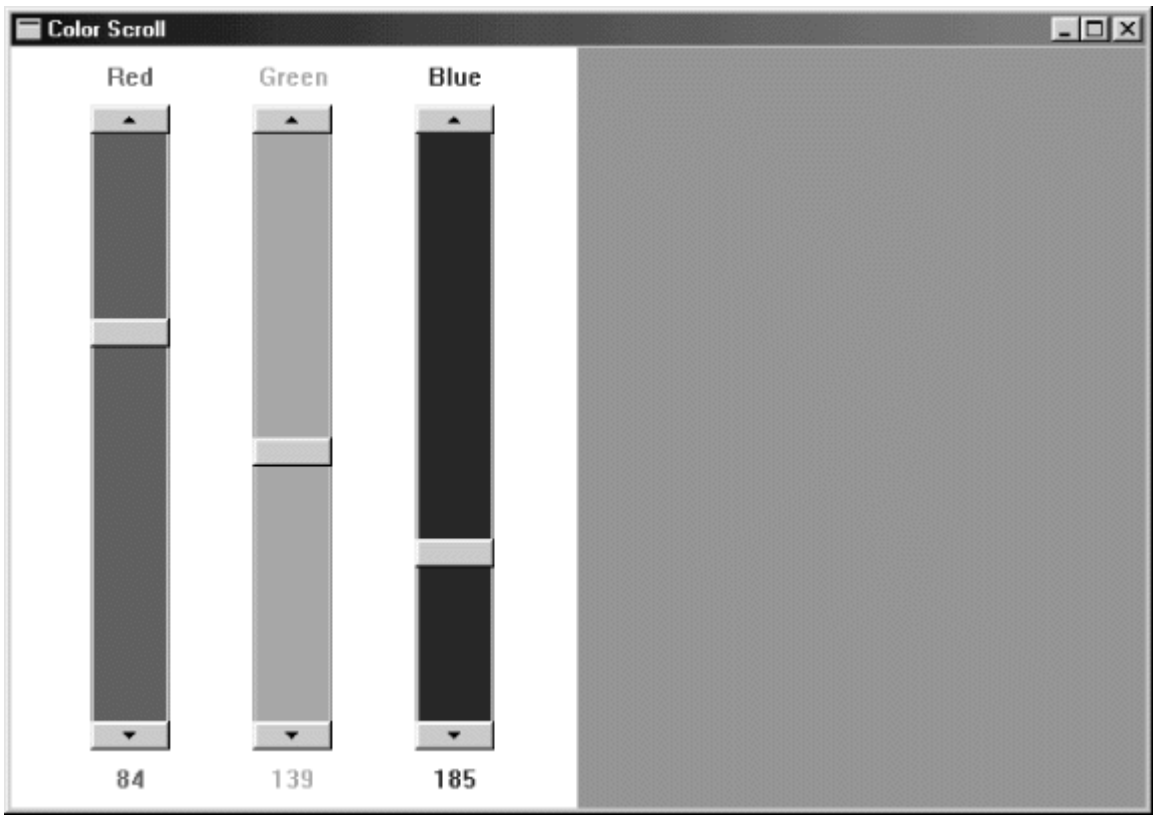

图 9-3 COLORS1 的萤幕显示

COLORS1 不处理 WM_PAINT 讯息，所有的工作几乎都是由子视窗完成的。

显示区域右半部显示的颜色实际上是视窗的背景颜色。SS_WHITERECT 样式的静态子视窗显示在显示区域的左半部。三个卷动列是 SBS_VERT 样式的子视窗控制项，它们被定位在 SS_WHITERECT 子视窗的顶部。另外六个 SS_CENTER 样式（居中文字）的静态子视窗提供标签和颜色值。COLORS1 在 WinMain 函式中用 CreateWindow 建立它的普通重叠式视窗和 10 个子视窗。SS_WHITERECT 和 SS_CENTER 静态视窗使用视窗类别「static」；三个卷动列使用视窗类别「scrollbar」。

CreateWindow 呼叫中的 x 位置、y 位置、宽度和高度参数最初设为 0，因为位置和大小都取决於显示区域的尺寸，而它目前尚未确定。COLORS1 的视窗讯息处理程式在接收到 WM_SIZE 讯息时，就使用 MoveWindow 给 10 个子视窗重新确定大小。所以，每当您对 COLORS1 视窗进行缩放时，卷动列的尺寸就会按比例变化。

当 WndProc 视窗讯息处理程式收到 WM_VSCROLL 讯息时，lParam 参数的高字组就是子视窗的代号。我们可以使用 GetWindowWord 来得到子视窗的 ID：

```
i = GetWindowLong ((HWND) lParam, GWL_ID) ;
```

对於这三个卷动列，我们已经按习惯将其 ID 设为 0、1、2，所以 WndProc 能区别出是哪个卷动列在产生讯息。

由於子视窗的代号在建立时就被储存在阵列中，所以 WndProc 就能对相对

应的卷动列讯息进行处理，并通过呼叫 SetScrollPos 来设定相对应的新值：

```
SetScrollPos (hwndScroll[i], SB_CTL, color[i], TRUE) ;
```

WndProc 也改变卷动列底部子视窗的文字：

```
wsprintf (szBuffer, TEXT ("%i"), color[I]) ;
SetWindowText (hwndValue[i], szBuffer) ;
```

自动键盘介面

卷动列控制项也能处理键盘输入，但是只有在拥有输入焦点时才行。下表说明怎样将键盘游标键转变为卷动讯息：

表 9-5

游标键	卷动讯息的 wParam 值
Home	SB_TOP
End	SB_BOTTOM
Page Up	SB_PAGEUP
Page Down	SB_PAGEDOWN
左或上	SB_LINEUP
右或下	SB_LINEDOWN

事实上，SB_TOP 和 SB_BOTTOM 卷动讯息只能用键盘产生。在使用滑鼠按动卷动列时，如果想使该卷动列获得输入焦点，那么您必须将 WS_TABSTOP 识别字包含到 CreateWindow 呼叫的视窗类别参数中。当卷动列拥有输入焦点时，在该卷动列的小方框上将显示一个闪烁的灰色块。

为了给卷动列提供全面的键盘介面，还需要另外一些工作。首先，WndProc 视窗讯息处理程式必须使卷动列拥有输入焦点，它是通过处理 WM_SETFOCUS 讯息来完成这一点的，该 WM_SETFOCUS 讯息是当卷动列获得输入焦点时其父视窗接收到的。WndProc 给其中一个卷动列设定输入焦点。

```
SetFocus (hwndScroll[idFocus]) ;
```

其中 idFocus 是一个整体变数。

但是，还需要一些借助键盘尤其是 Tab 键，来从一个卷动列转换到另一个卷动列的方法。这比较困难，因为一旦某个卷动列拥有了输入焦点，它就处理所有的键盘输入，但卷动列只关心游标键，而忽略 Tab 键。解决这一两难处境的方法是「视窗子类别化」。我们将用它来给 COLORS1 增加使用 Tab 键从一个卷动列跳到另一个卷动列的功能。

视窗子类别化（Window Subclassing）

卷动列控制项的视窗讯息处理程式是Windows内部的。但是，将GWL_WNDPROC

识别字作为参数来呼叫 GetWindowLong，您就可以得到这个视窗讯息处理程式的位址。另外，您可以呼叫 SetWindowLong 给该卷动列设定一个新的视窗讯息处理程式，这个技术叫做「视窗子类别化」，非常有用。它能让您给现存的视窗讯息处理程式设定「挂勾」，以便在自己的程式中处理一些讯息，同时将其他所有讯息传递给旧的视窗讯息处理程式。

在 COLORS1 中对卷动讯息进行初步处理的视窗讯息处理程式叫做 ScrollProc，它在 COLORS1.C 档案的尾部。由於 ScrollProc 是 COLORS1 中的函式，而 Windows 将呼叫 COLORS1，所以 ScrollProc 必须被定义为 callback 函式。

对三个卷动列中的每一个，COLORS1 使用 SetWindowLong 来设定新的卷动列视窗讯息处理程式的位址，并取得现存卷动列视窗讯息处理程式的位址：

```
OldScroll[i] = (WNDPROC) SetWindowLong (hwndScroll[i], GWL_WNDPROC,
        (LONG) ScrollProc)) ;
```

现在，函式 ScrollProc 得到了 Windows 发送到 COLORS1 中三个卷动列（当然不是其他程式中的卷动列）的卷动列视窗讯息处理程式的全部讯息。ScrollProc 视窗讯息处理程式在接收到 Tab 或者 Shift-Tab 键时，就将输入焦点改变到下一个（或者上一个）卷动列。它使用 CallWindowProc 呼叫旧的卷动列视窗讯息处理程式。

给背景著色

当 COLORS1 定义它的视窗类别时，也为其显示区域背景定义了一个实心的黑色画刷：

```
wndclass.hbrBackground = CreateSolidBrush (0) ;
```

当您改变 COLORS1 的卷动列设定时，程式必须建立一个新的画刷，并将该新画刷代号放入视窗类别结构中。如同使用 GetWindowLong 和 SetWindowLong 能得到并设定卷动列视窗讯息处理程式一样，用 GetClassWord 和 SetClassWord 能得到这个画刷的代号。

您可以建立新的画刷并将其代号插入视窗类别结构中，然後删除旧的画刷：

```
DeleteObject ((HBRUSH)
      SetClassLong (hwnd, GCL_HBRBACKGROUND, (LONG)
                  CreateSolidBrush (RGB (color[0], color[1], color[2])))) ;
```

Windows 下一次重新为视窗的背景著色时，将使用这个新画刷。为了强迫 Windows 抹掉背景，我们将使整个显示区域无效：

```
InvalidateRect (hwnd, &rcColor, TRUE) ;
```

TRUE（非零）值作为第三个参数，表示希望在重新著色之前删去背景。

InvalidateRect 使 Windows 在视窗讯息处理程式的讯息伫列中放进一个 WM_PAINT 讯息。由於 WM_PAINT 讯息的优先等级比较低，所以，如果您还在使用

滑鼠或者游标键移动卷动列的话，这个讯息将不会立即被处理。如果您想在颜色改变之後使该视窗立即变成最新的(目前的)，那么您可以在 InvalidateRect 之後增加下面的叙述：

```
UpdateWindow (hwnd) ;
```

但这会使得键盘和滑鼠处理变慢。

COLORS1 中的 WndProc 函式不处理 WM_PAINT 讯息，而是将其传给 DefWindowProc。Windows 对 WM_PAINT 讯息的内定处理只是呼叫 BeginPaint 和 EndPaint 使视窗生效。因为在 InvalidateRect 呼叫中已经指定背景要被抹掉，所以 BeginPaint 呼叫使 Windows 发出一个 WM_ERASEBKGND（删除背景）讯息，WndProc 也将忽略这个讯息。Windows 用视窗类别中指定的画刷将显示区域的背景抹去，这样就处理了这个讯息。

在终止以前进行清除总是一个好主意，因此在处理 WM_DESTROY 讯息处理期间，再一次呼叫 DeleteObject：

```
DeleteObject ((HBRUSH)
      SetClassLong (hwnd, GCL_HBRBACKGROUND,
                  (LONG) GetStockObject (WHITE_BRUSH))) ;
```

给卷动列和静态文字著色

在 COLORS1 中，三个卷动列的内部和六个文字栏位中的文字著色为红、绿和蓝色。卷动列的著色是通过处理 WM_CTLCOLORSCROLLBAR 讯息来完成的。

在 WndProc 中，我们为画刷定义了一个由三个代号组成的静态阵列：

```
static HBRUSH hBrush [3] ;
```

在处理 WM_CREATE 期间，我们建立三个画刷：

```
for (I = 0 ; I < 3 ; I++)
    hBrush[0] = CreateSolidBrush (crPrim [I]) ;
```

其中 crPrim 阵列中包含三种原色的 RGB 值。在 WM_CTLCOLORSCROLLBAR 处理期间视窗讯息处理程式传回这三画刷中的一个：

```
case  WM_CTLCOLORSCROLLBAR:
      i = GetWindowLong ((HWND) lParam, GWL_ID) ;
      return (LRESULT) hBrush [i] ;
```

在处理 WM_DESTROY 讯息的过程中，这些画刷必须被删除：

```
for   (i = 0 ; i < 3 ; i++)
      DeleteObject (hBrush [i])) ;
```

同样地，静态文字栏位中的文字是在处理 WM_CTLCOLORSTATIC 讯息中呼叫 SetTextColor 来著色的。文字背景用 SetBkColor 函式设定为系统颜色 COLOR_BTNHIGHLIGHT，这导致文字背景颜色和卷动列与文字後面的静态矩形控制项的颜色一样。对於静态文字控制项，这种文字背景颜色只用於字串中每个

字元後面的矩形，而不会用於整个控制项视窗。为了实作这一点，视窗讯息处理程式还必须传回 COLOR_BTNHIGHLIGHT 颜色画刷的代号。这个画刷被称为 hBrushStatic，它在 WM_CREATE 讯息处理期间建立，在 WM_DESTROY 讯息处理期间清除。

在 WM_CREATE 讯息处理期间依据 COLOR_BTNHIGHLIGHT 颜色建立画刷，并且在执行期间使用这一画刷时，我们遇到了一个小问题。如果程式在执行期间改变了 COLOR_BTNHIGHLIGHT 颜色，那么静态矩形的颜色将发生变化，并且文字背景的颜色也会变化，但是文字视窗控制项的整个背景将保持原有的 COLOR_BTNHIGHLIGHT 颜色。

为了解决这个问题，COLORS1 也简单地通过使用新颜色重新建立 hBrushStatic 来处理 WM_SYSCOLORCHANGE 讯息。

编辑类别

在某些方面，编辑类别是最简单的预先定义视窗类别；在另一方面，它又是最复杂的视窗类别。当您使用类别名称「edit」建立子视窗时，您根据 CreateWindow 呼叫中的 x 位置、y 位置、宽度和高度这些参数定义了一个矩形。此矩形含有可编辑文字。当子视窗控制项拥有输入焦点时，您可以输入文字，移动游标，使用滑鼠或者 Shift 键与一个游标键来选取部分文字，按 Ctrl-X 来删除所选文字或按 Ctrl-C 来复制所选文字、并送到剪贴簿上，按 Ctrl-V 键插入剪贴簿上的文字。

编辑控制项的最简单的应用之一是作为单行输入区域。但是编辑控制项并不仅限於单行，这一点我将在程式 9-4 POPPAD1 中说明。和我们在这本书中所遇到的各种其他问题一样，POPPAD 程式将逐步增强以使用功能表、对话方块(载入与储存档案)和列印。最後的版本将是一个简单而完整的文字编辑器，且其程式码将非常简洁。

程式 9-4　POPPAD1

```
POPPAD1.C
/*------------------------------------------------------------------------
   POPPAD1.C -- Popup Editor using child window edit box
                             (c) Charles Petzold, 1998
------------------------------------------------------------------------*/

#include <windows.h>
#define ID_EDIT    1

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
```

```
TCHAR szAppName[] = TEXT ("PopPad1") ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     HWND             hwnd ;
     MSG                 msg ;
     WNDCLASS         wndclass ;

     wndclass.style               = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc         = WndProc ;
     wndclass.cbClsExtra          = 0 ;
     wndclass.cbWndExtra          = 0 ;
     wndclass.hInstance           = hInstance ;
     wndclass.hIcon               = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor             = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground       = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName        = NULL ;
     wndclass.lpszClassName       = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (     NULL, TEXT ("This program requires Windows NT!"),
                                        szAppName, MB_ICONERROR) ;
                    return 0 ;
     }

     hwnd = CreateWindow (  szAppName, szAppName,
                    WS_OVERLAPPEDWINDOW,
                    CW_USEDEFAULT, CW_USEDEFAULT,
                    CW_USEDEFAULT, CW_USEDEFAULT,
                    NULL, NULL, hInstance, NULL) ;
     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
                    TranslateMessage (&msg) ;
                    DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (   HWND  hwnd,  UINT  message,  WPARAM  wParam,LPARAM
lParam)
{
     static HWND hwndEdit ;
     switch (message)
```

```
     {
     case  WM_CREATE :
                hwndEdit = CreateWindow (TEXT ("edit"), NULL,
             WS_CHILD | WS_VISIBLE | WS_HSCROLL | WS_VSCROLL |
             WS_BORDER | ES_LEFT | ES_MULTILINE |
             ES_AUTOHSCROLL | ES_AUTOVSCROLL,
             0, 0, 0, 0, hwnd, (HMENU) ID_EDIT,
             ((LPCREATESTRUCT) lParam) -> hInstance, NULL) ;
                return 0 ;

     case  WM_SETFOCUS :
                SetFocus (hwndEdit) ;
                return 0 ;

     case  WM_SIZE :
           MoveWindow (hwndEdit, 0, 0, LOWORD (lParam), HIWORD (lParam), TRUE) ;
                return 0 ;

     case  WM_COMMAND :
                if (LOWORD (wParam) == ID_EDIT)
             if (HIWORD (wParam) == EN_ERRSPACE ||
                HIWORD (wParam) == EN_MAXTEXT)
           MessageBox (hwnd, TEXT ("Edit control out of space."),
                szAppName, MB_OK | MB_ICONSTOP) ;
                return 0 ;

     case  WM_DESTROY :
                PostQuitMessage (0) ;
                return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

POPPAD1 是一个多行编辑器（只是没有档案 I/O），其 C 语言原始码不到 100 行（不过，有一个缺陷，即预先定义的多行编辑控制项只限於 30,000 字元的文字）。您可以看到，POPPAD1 本身并没有做多少工作，预先定义的编辑控制项完成了许多工作，这样，您可以知道，无需额外的程式时编辑控制项能做些什么。

编辑类别样式

如前面所提到的，在 CreateWindow 呼叫中将「edit」作为视窗类别建立了一个编辑控制项，视窗样式是 WS_CHILD 加上几个选项。如同在静态子视窗控制项中一样，编辑控制项中的文字可以置左对齐、置右对齐或者居中，您使用视窗样式 ES_LEFT、ES_RIGHT 和 ES_CENTER 来指定这些格式。

内定状态下，编辑控制项是单行的。您使用 ES_MULTILINE 视窗样式可以建

立多行编辑控制项。对於单行编辑控制项，您一般只可以在编辑控制项矩形的尾部输入文字。要建立一个自动水平卷动的编辑控制项，您可以采用样式 ES_AUTOHSCROLL。对一个多行编辑控制项，文字会自动跳行，除非使用 ES_AUTOHSCROLL 样式。在这种情况下，您必须按 Enter 键来开始新的一行。您还可以便用样式 ES_AUTOVSCROLL 来将垂直卷动列包括在多行编辑控制项中。

当您在多行编辑控制项中包括这些卷动样式时，也许还想给编辑控制项增加卷动列。要做到这些，可以对非子视窗使用同一视窗样式识别字 WS_HSCROLL 和 WS_VSCROLL。内定状态下，编辑控制项没有边界，利用样式 WS_BORDER 则可以增加边界。

当您在编辑控制项中选择文字时，Windows 将选择的文字反白显示。但是当编辑控制项失去输入焦点时，被选择的文字将不再被加亮。如果希望在编辑控制项没有输入焦点时被选择的文字仍然被加亮，您可以使用样式 ES_NOHIDESEL。

在 POPPAD1 建立其编辑控制项时，CreateWindow 呼叫依如下形式给出样式：

```
WS_CHILD | WS_VISIBLE | WS_HSCROLL | WS_VSCROLL |
      WS_BORDER | ES_LEFT | ES_MULTILINE |
      ES_AUTOHSCROLL | ES_AUTOVSCROLL
```

在 POPPAD1 中，编辑控制项的大小是後来当 WndProc 接收到 WM_SIZE 讯息时通过呼叫 MoveWindow 来定义的。编辑控制项的尺寸被简单地设定为主视窗的尺寸：

```
MoveWindow (hwndEdit, 0, 0, LOWORD (lParam),
                            HIWORD (lParam), TRUE) ;
```

对於单行编辑控制项，控制项的高度必须可以容纳一个字元。如果编辑控制项有边界（大多数都有），那么使用一个字元高度的 1.5 倍（包括外部间距）。

编辑控制项通知

编辑控制项给父视窗讯息处理程式发送 WM_COMMAND 讯息，对按钮控制项来说，wParam 和 lParam 变数的含义是相同的：

LOWORD (wParam)	子视窗 ID
HIWORD (wParam)	通知码
lParam	子视窗代号

通知码如下所示：

EN_SETFOCUS	编辑控制项已经获得输入焦点
EN_KILLFOCUS	编辑控制项已经失去输入焦点
EN_CHANGE	编辑控制项的内容将改变
EN_UPDATE	编辑控制项的内容已经改变
EN_ERRSPACE	编辑控制项执行已经超出中间

EN_MAXTEXT	编辑控制项在插入时执行超出空间
EN_HSCROLL	编辑控制项的水平卷动列已经被按下
EN_VSCROLL	编辑控制项的垂直卷动列已经被按下

POPPAD1 只拦截 EN_ERRSPACE 和 EN_MAXTEXT 通知码，并显示一个讯息方块。

使用编辑控制项

如果在您的主视窗上使用了几个单行编辑控制项，那么您需要将视窗子类别化以便把输入焦点从一个控制项转移到另一个控制项。您可以通过拦截 Tab 键和 Shift-Tab 键来完成这种移动，非常像 COLORS1 中所做的（视窗子类别化的另一个例子在後面的 HEAD 程式中说明）。如何处理 Enter 键取决於您，可以像 Tab 键那样使用，也可以当成给程式的信号，表示所有的编辑栏位都准备好了。

如果您想在编辑区中插入文字，那么可以使用 SetWindowText 来做到。将文字从编辑控制项中取出涉及了 GetWindowTextLength 和 GetWindowText，我们将在 POPPAD 程式的修订版本中看到这些操作的实例。

发送给编辑控制项的讯息

因为用 SendMessage 发送给编辑控制项的讯息很多，并且其中的几个还将在後面 POPPAD 修订版本中用到，所以这里不解说所有用 SendMessage 发送给编辑控制项的讯息，只概要地说明一下。

这些讯息允许您剪下、复制或者清除目前被选择的文字。使用者使用滑鼠或者 Shift 键加上游标控制项键来选择文字并进行上面的操作，这样，在编辑控制项中选中的文字将被加亮：

```
SendMessage (hwndEdit, WM_CUT, 0, 0) ;
SendMessage (hwndEdit, WM_COPY, 0, 0) ;
SendMessage (hwndEdit, WM_CLEAR, 0, 0) ;
```

WM_CUT 将目前选择的文字从编辑控制项中移走，并将其发送到剪贴簿中；WM_COPY 将选择的文字复制到剪贴簿上并保持编辑控制项中的内容完好无损；WM_CLEAR 将选择的内容从编辑控制项中删除，但是不向剪贴簿中发送。

您也可以将剪贴簿上的文字插入到编辑控制项中的游标位置：

```
SendMessage (hwndEdit, WM_PASTE, 0, 0) ;
```

您可以取得目前选择的起始位置和末尾位置：

```
SendMessage (hwndEdit, EM_GETSEL, (WPARAM) &iStart,
                                  (LPARAM) &iEnd) ;
```

结束位置实际上是最後一个选择字元的位置加 1。

您可以选择文字：

```
SendMessage (hwndEdit, EM_SETSEL, iStart, iEnd) ;
```

您还可以使用别的文字来置换目前的选择内容：

```
SendMessage (hwndEdit, EM_REPLACESEL, 0, (LPARAM) szString) ;
```

对多行编辑控制项，您可以取得行数：

```
iCount = SendMessage (hwndEdit, EM_GETLINECOUNT, 0, 0) ;
```

对任何特定的行，您可以取得距离编辑缓冲区文字开头的偏移量：

```
iOffset = SendMessage (hwndEdit, EM_LINEINDEX, iLine, 0) ;
```

行数从 0 开始计算，iLine 值为-1 时传回包含游标所在行的偏移量。您可以取得行的长度：

```
iLength = SendMessage (hwndEdit, EM_LINELENGTH, iLine, 0) ;
```

并将行本身复制到一个缓冲区中：

```
iLength = SendMessage (hwndEdit, EM_GETLINE, iLine, (LPARAM) szBuffer) ;
```

清单方块类别

我在本章讨论的最後一个预先定义子视窗控制项是清单方块。一个清单方块是字串的集合，这些字串是一个矩形中可以卷动显示的清单。——程式通过向清单方块视窗讯息处理程式发送讯息，可以在清单中增加或者删除字串。当清单方块中的某项被选择时，清单方块控制项就向其父视窗发送 WM_COMMAND 讯息，父视窗也就可以确定选择的是哪一项。

一个清单方块可以是单选的，也可以是多选的，後者允许使用者从清单方块中选择多个项目。当清单方块拥有输入焦点时，其中项目的周围显示有虚线。在清单方块中，游标位置并不指明被选择的项目。被选择的项目被加亮显示，并且是反白显示的。

在单项选择的清单方块中，使用者按 Spacebar 键就可以选择游标所在位置的项目。方向键移动游标和目前选择指示，并且能够滚动清单方块的内容。Page Up 和 Page Down 键也能滚动清单方块，但它移动的是游标而不是选择指示。按字母键能将游标和选择指示移到以此字母开头的第一个（或下一个）选项。也可以使用滑鼠在要选择的项目上单击或者双击来选择它。

在多项选择清单方块中，Spacebar 键可以切换游标所在位置的项目的选择状态（如果该项已经被选择，则取消选择）。如同在单项选择清单方块中一样，方向键取消前面选择过的项目，并且移动游标和选择指示。但是，Ctrl 键和方向键能够在移动游标的同时不移动选择，Shift 键加方向键能扩展一个选择。

在多项选择清单方块中，单击或者双击滑鼠按键能取消之前所有的选择，而选择被点中的项目。但是，如果在滑鼠点中某一项的同时也按下 Shift 键，则只能切换该项的选择状态，而不会改变任何其他项的选择状态。

清单方块样式

当您使用 CreateWindow 建立清单方块子视窗时，您应该将「listbox」作为视窗类别，将 WS_CHILD 作为视窗样式。但是，这个内定清单方块样式不向其父视窗发送 WM_COMMAND 讯息，这样一来，程式必须向清单方块询问其中的项目的选择状态（借助於发送给清单方块控制项的讯息）。所以，清单方块控制项通常都包括清单方块样式识别字 LBS_NOTIFY，它允许父视窗接收来自清单方块的 WM_COMMAND 讯息。如果您希望清单方块控制项对清单方块中的项目进行排序，那么您可以使用另一种常用的样式 LBS_SORT。

内定情况下，清单方块是单项选择的。多项选择的清单方块相当少。如果您想建立一个多项选择清单方块，那么您可以使用样式 LBS_MULTIPLESEL。通常，当给有卷动列的清单方块增加新项目时，清单方块本身会自己重画。您可以通过将样式 LBS_NOREDRAW 包含进去来防止这种现象。但是您也许不想使用这种样式，这时可以使用 WM_SETREDRAW 讯息来暂时防止清单方块控制项重新画过，我将在稍後讨论 WM_SETREDRAW 讯息。

内定状态下，清单方块视窗讯息处理程式只显示列表项目，它的周围没有任何边界。您可以使用视窗样式识别字 WS_BORDER 来加上边界。另外，您可以使用视窗样式识别字 WS_VSCROLL 来增加垂直卷动列，以便用滑鼠来卷动列表项目。

Windows 表头档案定义了一个清单方块样式，叫做 LBS_STANDARD，它包含了最常用的样式，其定义如下：

```
(LBS_NOTIFY | LBS_SORT | WS_VSCROLL | WS_BORDER)
```

您也可以采用 WS_SIZEBOX 和 WS_CAPTION 识别字，但是这两个识别字允许您重新定义清单方块的大小，也允许您在清单方块父视窗的显示区域中移动清单方块。

清单方块的宽度应该能够容纳最长字串的宽度加上卷动列的宽度。您可以使用：

```
GetSystemMetrics (SM_CXVSCROLL) ;
```

来获得垂直卷动列的宽度。您用一个字元的高度乘以想要在视埠中显示的项目数来计算出清单方块的高度。

将字串放入清单方块

建立清单方块之後，下一步是将字串放入其中，您可以通过呼叫 SendMessage 为清单方块视窗讯息处理程式发送讯息来做到这一点。字串通常通过以 0 开始计数的索引数来引用，其中 0 对应於最顶上的项目。在下面的例子

中，hwndList 是子视窗清单方块控制项的代号，而 iIndex 是索引值。在使用 SendMessage 传递字串的情况下，lParam 参数是指向以 null 字元结尾字串的指标。

在大多数例子中，当视窗讯息处理程式储存的清单方块内容超过了可用记忆体空间时，SendMessage 将传回 LB_ERRSPACE（定义为-2）。如果是因为其他原因而出错，那么 SendMessage 将传回 LB_ERR（-1）。如果操作成功，那么 SendMessage 将传回 LB_OKAY（0）。您可以通过测试 SendMessage 的非零值来判断这两种错误。

如果您采用 LBS_SORT 样式（或者如果您在清单方块中按照想要呈现的顺序排列字串），那么填入清单方块最简单的方法是借助 LB_ADDSTRING 讯息：

```
SendMessage (hwndList, LB_ADDSTRING, 0, (LPARAM) szString) ;
```

如果您没有采用 LBS_SORT，那么可以使用 LB_INSERTSTRING 指定一个索引值，将字串插入到清单方块中：

```
SendMessage (hwndList, LB_INSERTSTRING, iIndex, (LPARAM) szString) ;
```

例如，如果 iIndex 等於 4，那么 szString 将变为索引值为 4 的字串——从顶头开始算起的第 5 个字串（因为是从 0 开始计数的），位於这个点後面的所有字串都将向後推移。索引值为-1 时，将字串增加在最後。您可以对样式为 LBS_SORT 的清单方块使用 LB_INSERTSTRING，但是这个清单方块的内容不能被重新排序（您也可以使用 LB_DIR 讯息将字串插入到清单方块中，这将在本章的最後进行讨论）。

您可以在指定索引值的同时使用 LB_DELETESTRING 参数，这就可以从清单方块中删除字串：

```
SendMessage (hwndList, LB_DELETESTRING, iIndex, 0) ;
```

您可以使用 LB_RESETCONTENT 清除清单方块中的内容：

```
SendMessage (hwndList, LB_RESETCONTENT, 0, 0) ;
```

当在清单方块中增加或者删除字串时，清单方块视窗讯息处理程式将更新显示。如果您有许多字串需要增加或者删除，那么您也许希望暂时阻止这一动作，其方法是关掉控制项的重画旗标：

```
SendMessage (hwndList, WM_SETREDRAW, FALSE, 0) ;
```

当您完成後，可以再打开重画旗标：

```
SendMessage (hwndList, WM_SETREDRAW, TRUE, 0) ;
```

使用 LBS_NOREDRAW 样式建立的清单方块开始时其重画旗标是关闭的。

选择和取得项

SendMessage 完成了下面所描述的任务之後，通常传回一个值。如果出错，那么这个值将被设定为 LB_ERR（定义为-1）。

当清单方块中放入一些项目之後，您可以弄清楚清单方块中有多少项目：

```
iCount = SendMessage (hwndList, LB_GETCOUNT, 0, 0) ;
```

其他一些呼叫对单项选择清单方块和多项选择清单方块是不同的。让我们先来看看单项选择清单方块。

通常，您让使用者在清单方块中选择条目。但是如果您想加亮显示一个内定选择，则可以使用：

```
SendMessage (hwndList, LB_SETCURSEL, iIndex, 0) ;
```

将 iParam 设定为-1 则取消所有选择。

您也可以根据项目的第一个字母来选择：

```
iIndex = SendMessage (hwndList, LB_SELECTSTRING, iIndex,
                      (LPARAM) szSearchString) ;
```

在 SendMessage 呼叫中将 iIndex 作为 iParam 参数时，iIndex 是索引，可以根据它搜索其开头字元与 szSearchString 相匹配的项目。iIndex 的值等於-1 时从头开始搜索，SendMessage 传回被选中项目的索引。如果没有开头字元与 szSearchString 相匹配的项目时，SendMessage 传回 LB_ERR。

当您得到来自清单方块的 WM_COMMAND 讯息时（或者在任何其他时候），您可以使用 LB_GETCURSEL 来确定目前选项的索引：

```
iIndex = SendMessage (hwndList, LB_GETCURSEL, 0, 0) ;
```

如果没有项目被选中，那么从呼叫中传回的 iIndex 值为 LB_ERR。

您可以确定清单方块中字串的长度：

```
iLength = SendMessage (hwndList, LB_GETTEXTLEN, iIndex, 0) ;
```

并可以将某项目复制到文字缓冲区中：

```
iLength = SendMessage (      hwndList, LB_GETTEXT, iIndex,
                                                   (LPARAM) szBuffer) ;
```

在这两种情况下，从呼叫传回的 iLength 值是字串的长度。对以 NULL 字元终结的字串长度来说，szBuffer 阵列必须够大。您也许想用 LB_GETTEXTLEN 先分配一些局部记忆体来存放字串。

对於一个多项选择清单方块，您不能使用 LB_SETCURSEL、LB_GETCURSEL 或者 LB_SELECTSTRING,但是您可以使用 LB_SETSEL 来设定某特定项目的选择状态，而不影响有可能被选择的其他项：

```
SendMessage (hwndList, LB_SETSEL, wParam, iIndex) ;
```

wParam 参数不为 0 时，选择并加亮某一项目；wParam 为 0 时，取消选择。如果 wParam 等於-1，那么将选择所有项目或者取消所有被选中的项目。您可以如下确定某特定项目的选择状态：

```
iSelect = SendMessage (hwndList, LB_GETSEL, iIndex, 0) ;
```

其中，如果由 iIndex 指定的项目被选中，iSelect 被设为非 0，否则被设为 0。

接收来自清单方块的讯息

当使用者用滑鼠单击清单方块时，清单方块将接收输入焦点。下面的操作可以使父视窗将输入焦点转交给清单方块控制项：

```
SetFocus (hwndList) ;
```

当清单方块拥有输入焦点时，游标移动键、字母键和 Spacebar 键都可以用来在该清单方块中选择某项。

清单方块控制项向其父视窗发送 WM_COMMAND 讯息，对按钮和编辑控制项来说，wParam 和 lParam 变数的含义是相同的：

LOWORD (wParam)	子视窗 ID
HIWORD (wParam)	通知码
lParam	子视窗代号

通知码及其值如下所示：

LBN_ERRSPACE	-2
LBN_SELCHANGE	1
LBN_DBLCLK	2
LBN_SELCANCEL	3
LBN_SETFOCUS	4
LBN_KILLFOCUS	5

只有清单方块视窗样式包括 LBS_NOTIFY 时，清单方块控制项才会向父视窗发送 LBN_SELCHANGE 和 LBN_DBLCLK。

LBN_ERRSPACE 表示清单方块已经超出执行空间。LBN_SELCHANGE 表示目前选择已经被改变。这些讯息出现在下列情况下：使用者在清单方块中移动加亮的项目时，使用者使用 Spacebar 键切换选择状态或者使用滑鼠单击某项时。LBN_DBLCLK 说明某项目已经被滑鼠双击（LBN_SELCHANGE 和 LBN_DBLCLK 通知码的值表示滑鼠按下的次数）。

根据应用的需要，您也许要使用 LBN_SELCHANGE 或 LBN_DBLCLK，也许二者都要使用。您的程式会收到许多 LBN_SELCHANGE 讯息，但是 LBN_DBLCLK 讯息只有当使用者双击滑鼠时才会出现。如果您的程式使用双击，那么您需要提供一个复制 LBN_DBLCLK 的键盘介面。

一个简单的清单方块应用程式

既然您知道了如何建立清单方块，如何使用文字项目填入清单方块，如何接收来自清单方块的控制项以及如何取得字串，现在是到了写一个应用程式的时候了。如程式 9-5 中所示，ENVIRON 程式在显示区域中使用清单方块来显示目

前作业系统环境变数（例如 PATH 和 WINDIR）。当您选择一个环境变数时，其内容将显示在显示区域的顶部。

程式 9-5 ENVIRON

```
ENVIRON.C
/*------------------------------------------------------------------------
      ENVIRON.C -- Environment List Box
                           (c) Charles Petzold, 1998
------------------------------------------------------------------------*
/

#include <windows.h>
#define ID_LIST    1
#define ID_TEXT    2

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                                          PSTR szCmdLine, int iCmdShow)
{
      static TCHAR     szAppName[] = TEXT ("Environ") ;
      HWND                              hwnd ;
      MSG                          msg ;
      WNDCLASS                     wndclass ;

      wndclass.style                          = CS_HREDRAW | CS_VREDRAW ;
      wndclass.lpfnWndProc                    = WndProc ;
      wndclass.cbClsExtra                     = 0 ;
      wndclass.cbWndExtra                     = 0 ;
      wndclass.hInstance                      = hInstance ;
      wndclass.hIcon                          =        LoadIcon        (NULL,
IDI_APPLICATION) ;
      wndclass.hCursor                        = LoadCursor (NULL, IDC_ARROW) ;
      wndclass.hbrBackground             = (HBRUSH) (COLOR_WINDOW + 1) ;
      wndclass.lpszMenuName              = NULL ;
      wndclass.lpszClassName             = szAppName ;

      if (!RegisterClass (&wndclass))
      {
                  MessageBox (     NULL, TEXT ("This program requires Windows
NT!"),
                                                          szAppName,
MB_ICONERROR) ;
                  return 0 ;
      }

      hwnd = CreateWindow (  szAppName, TEXT ("Environment List Box"),
                    WS_OVERLAPPEDWINDOW,
```

```
                    CW_USEDEFAULT, CW_USEDEFAULT,
                    CW_USEDEFAULT, CW_USEDEFAULT,
                    NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
                    TranslateMessage (&msg) ;
                    DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

void FillListBox (HWND hwndList)
{
     int    iLength ;
     TCHAR * pVarBlock, * pVarBeg, * pVarEnd, * pVarName ;

     pVarBlock = GetEnvironmentStrings () ;  // Get pointer to environment
block

     while (*pVarBlock)
     {
          if (*pVarBlock != '=')       // Skip variable names beginning with
'='
          {
            pVarBeg = pVarBlock ;      // Beginning of variable name
          while (*pVarBlock++ != '=') ;      // Scan until '='
          pVarEnd = pVarBlock - 1 ;          // Points to '=' sign
          iLength = pVarEnd - pVarBeg ;      // Length of variable name

          // Allocate memory for the variable name and terminating
          // zero. Copy the variable name and append a zero.

          pVarName = calloc (iLength + 1, sizeof (TCHAR)) ;
          CopyMemory (pVarName, pVarBeg, iLength * sizeof (TCHAR)) ;
          pVarName[iLength] = '\0' ;

          // Put the variable name in the list box and free memory.
          SendMessage (hwndList, LB_ADDSTRING, 0, (LPARAM) pVarName) ;
                         free (pVarName) ;
          }
          while (*pVarBlock++ != '\0') ;     // Scan until terminating zero
     }
     FreeEnvironmentStrings (pVarBlock) ;
}
```

```
LRESULT CALLBACK WndProc (  HWND  hwnd,  UINT  message,  WPARAM  wParam,LPARAM
lParam)
{
      static HWND hwndList, hwndText ;
      int                         iIndex, iLength, cxChar, cyChar ;
      TCHAR                 *     pVarName, * pVarValue ;

      switch (message)
      {
      case  WM_CREATE :
                  cxChar = LOWORD (GetDialogBaseUnits ()) ;
                  cyChar = HIWORD (GetDialogBaseUnits ()) ;
                              // Create listbox and static text windows.

                  hwndList = CreateWindow (TEXT ("listbox"), NULL,
                        WS_CHILD | WS_VISIBLE | LBS_STANDARD,
                        cxChar, cyChar * 3,
                        cxChar * 16 + GetSystemMetrics (SM_CXVSCROLL),
                        cyChar * 5,
                        hwnd, (HMENU) ID_LIST,
                        (HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE),
                                                      NULL) ;

                  hwndText = CreateWindow (TEXT ("static"), NULL,
                        WS_CHILD | WS_VISIBLE | SS_LEFT,
                        cxChar, cyChar,
                        GetSystemMetrics (SM_CXSCREEN), cyChar,
                        hwnd, (HMENU) ID_TEXT,
                        (HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE),
                                                            NULL) ;

                  FillListBox (hwndList) ;
                  return 0 ;

      case  WM_SETFOCUS :
                  SetFocus (hwndList) ;
                  return 0 ;
      case  WM_COMMAND :
                  if  (LOWORD  (wParam)  ==  ID_LIST  &&  HIWORD  (wParam)  ==
LBN_SELCHANGE)
                  {
                                          // Get current selection.

               iIndex  = SendMessage (hwndList, LB_GETCURSEL, 0, 0) ;
               iLength = SendMessage (hwndList, LB_GETTEXTLEN, iIndex, 0) + 1 ;
              pVarName = calloc (iLength, sizeof (TCHAR)) ;
              SendMessage (hwndList, LB_GETTEXT, iIndex, (LPARAM) pVarName) ;
```

```
                                        // Get environment string.

            iLength = GetEnvironmentVariable (pVarName, NULL, 0) ;
            pVarValue = calloc (iLength, sizeof (TCHAR)) ;
            GetEnvironmentVariable (pVarName, pVarValue, iLength) ;

                                        // Show it in window.

            SetWindowText (hwndText, pVarValue) ;
            free (pVarName) ;
            free (pVarValue) ;
                 }
                 return 0 ;

      case  WM_DESTROY :
                 PostQuitMessage (0) ;
                 return 0 ;
      }
      return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

ENVIRON 建立两个子视窗：一个是 LBS_STANDARD 样式的清单方块，另一个是 SS_LEFT 样式（置左对齐文字）的静态视窗。ENVIRON 使用函式 GetEnvironmentStrings 来获得一个指标，该指标指向存有全部环境变数名及其值的记忆体区块。ENVIRON 用 FillListBox 函式来分析此记忆体区块，并使用 LB_ADDSTRING 讯息来指定清单方块视窗讯息处理程式将每个字串放入清单方块中。

当您执行 ENVIRON 时，可以使用滑鼠或者键盘来选择环境变数。每次您改变选择时，清单方块都会给其父视窗 WndProc 发送一个 WM_COMMAND 讯息。当 WndProc 收到 WM_COMMAND 讯息时，它就检查 wParam 的低字组是否为 ID_LIST（清单方块的子视窗 ID）和 wParam 的高字组（通知码）是否等於 LBN_SELCHANGE。如果是的，那么它就使用 LB_GETCURSEL 讯息来获得选中项目的索引，并使用 LB_GETTEXT 来获得外部环境变数名的字串本身。ENVIRON 程式使用 C 语言函式 GetEnvironmentVariable 来获得与变数相对应的环境字串，使用 SetWindowText 将该字串传递到静态子视窗控制项中，这个静态子视窗控制项被用来显示文字。

档案列表

我将最好的留在最後：LB_DIR，这是功能最强的清单方块讯息。它用档案目录列表填入清单方块，并且可以选择将子目录和有效的磁碟机也包括进来：

```
SendMessage (hwndList, LB_DIR, iAttr, (LPARAM) szFileSpec) ;
```

使用档案属性码

iAttr 参数是档案属性代码，其最低位元组是档案属性代码，该代码可以是表 9-6 资料的组合：

表 9-6

iAttr	值	属性
DDL_READWRITE	0x0000	普通档案
DDL_READONLY	0x0001	唯读档案
DDL_HIDDEN	0x0002	隐藏档案
DDL_SYSTEM	0x0004	系统档案
DDL_DIRECTORY	0x0010	子目录
DDL_ARCHIVE	0x0020	归档位元设立的档案

高位元组提供了一些对所要求项目的附加控制：

表 9-7

iAttr	值	属性
DDL_DRIVES	0x4000	包括磁碟机代号
DDL_EXCLUSIVE	0x8000	互斥搜索

字首 DDL 表示「对话目录列表」。

当 LB_DIR 讯息的 iAttr 值为 DDL_READWRITE 时，清单方块列出普通档案、唯读档案和归档位元设立的档案。当值为 DDL_DIRECTORY 时，清单方块除了列出上述档案之外，还列出子目录，目录位於中括号之内。当值为 DDL_DRIVES | DDL_DIRECTORY 时，那么列表将扩展到包括所有有效的磁碟机，而磁碟机代号显示在虚线之间。

将 iAttr 的最高位元设立就可以只列出符合条件的档案，而不包括其他档案。例如，对 Windows 的档案备份程式，也许您只想列出最後一次备份後修改过的档案，这种档案的归档位元设立，因此您可以使用 DDL_EXCLUSIVE | DDL_ARCHIVE。

档案列表的排序

lParam 参数是指向档案指定字串如「*.*」的指标，这个档案指定字串不影响清单方块中的子目录。

您也许希望给列有档案清单的清单方块使用 LBS_SORT 讯息。清单方块首先列出符合档案指定要求的档案，再（可选择）列出子目录名。列出的第一个子目录名将采用下面的格式：

[..]

这一个「两个点」的子目录项允许使用者向根目录回溯一层（在根目录下列出档案名时此项目不会出现）。最後，具体的子目录名称采用下面的形式：

[SUBDIR]

再来是以下列形式列出的有效磁碟机（也是可选择的）：

[-A-]

Windows 的 head 程式

UNIX 中有一个著名的实用程式叫做 head，它显示档案开始的几行。让我们使用清单方块为 Windows 编写一个类似的程式。如程式 9-6 所示，HEAD 将所有档案和子目录列在清单方块中。您可以挑选某个被选择的档案来显示，方法是在该档案上使用滑鼠双击或者使用 Enter 键按下要选的档案。您也可以使用这两种方法之一来改变子目录。这个程式在 HEAD 视窗显示区域的右边，从档案的开头开始显示，它最多能够显示 8 KB 的内容。

程式 9-6 HEAD

```
HEAD.C
/*------------------------------------------------------------------------
      HEAD.C -- Displays beginning (head) of file
                        (c) Charles Petzold, 1998
------------------------------------------------------------------------*/

#include <windows.h>
#define ID_LIST    1
#define ID_TEXT    2

#define MAXREAD    8192
#define DIRATTR    (DDL_READWRITE | DDL_READONLY | DDL_HIDDEN | DDL_SYSTEM | \
                 DDL_DIRECTORY | DDL_ARCHIVE  | DDL_DRIVES)
#define DTFLAGS    (DT_WORDBREAK | DT_EXPANDTABS | DT_NOCLIP |DT_NOPREFIX)
LRESULT CALLBACK WndProc  (HWND, UINT, WPARAM, LPARAM) ;
LRESULT CALLBACK ListProc (HWND, UINT, WPARAM, LPARAM) ;

WNDPROC OldList ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                                          PSTR szCmdLine, int iCmdShow)
{
      static TCHAR           szAppName[] = TEXT ("head") ;
      HWND                                hwnd ;
      MSG                           msg ;
      WNDCLASS                      wndclass ;
      wndclass.style                      = CS_HREDRAW | CS_VREDRAW ;
```

```
     wndclass.lpfnWndProc         = WndProc ;
     wndclass.cbClsExtra          = 0 ;
     wndclass.cbWndExtra          = 0 ;
     wndclass.hInstance           = hInstance ;
     wndclass.hIcon               = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor             = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground       = (HBRUSH) (COLOR_BTNFACE + 1) ;
     wndclass.lpszMenuName        = NULL ;
     wndclass.lpszClassName       = szAppName ;

     if (!RegisterClass (&wndclass))
     {
          MessageBox (     NULL, TEXT ("This program requires Windows NT!"),
                                              szAppName, MB_ICONERROR) ;
               return 0 ;
     }

     hwnd = CreateWindow (  szAppName, TEXT ("head"),
                  WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN,
                  CW_USEDEFAULT, CW_USEDEFAULT,
                  CW_USEDEFAULT, CW_USEDEFAULT,
                 NULL, NULL, hInstance, NULL) ;

          ShowWindow (hwnd, iCmdShow) ;
          UpdateWindow (hwnd) ;

          while (GetMessage (&msg, NULL, 0, 0))
    {
               TranslateMessage (&msg) ;
               DispatchMessage (&msg) ;
    }
     return msg.wParam ;
}

LRESULT CALLBACK WndProc (   HWND  hwnd,  UINT  message,  WPARAM  wParam,LPARAM
lParam)
{
     static BOOL              bValidFile ;
     static BYTE                 buffer[MAXREAD] ;
     static HWND                 hwndList, hwndText ;
     static RECT                 rect ;
     static TCHAR                szFile[MAX_PATH + 1] ;
     HANDLE                      hFile ;
     HDC                         hdc ;
     int                         i, cxChar, cyChar ;
     PAINTSTRUCT                 ps ;
     TCHAR                       szBuffer[MAX_PATH + 1] ;
     switch (message)
```

```
    {
      case  WM_CREATE :
                  cxChar = LOWORD (GetDialogBaseUnits ()) ;
                  cyChar = HIWORD (GetDialogBaseUnits ()) ;

                  rect.left = 20 * cxChar ;
                  rect.top  =  3 * cyChar ;

                  hwndList = CreateWindow (TEXT ("listbox"), NULL,
                        WS_CHILDWINDOW | WS_VISIBLE | LBS_STANDARD,
                        cxChar, cyChar * 3,
                        cxChar * 13 + GetSystemMetrics (SM_CXVSCROLL),
                        cyChar * 10,
                        hwnd, (HMENU) ID_LIST,
                        (HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE),
                                                          NULL) ;

                  GetCurrentDirectory (MAX_PATH + 1, szBuffer) ;

                  hwndText =  CreateWindow (TEXT ("static"), szBuffer,
                        WS_CHILDWINDOW | WS_VISIBLE | SS_LEFT,
                        cxChar, cyChar, cxChar * MAX_PATH, cyChar,
                        hwnd, (HMENU) ID_TEXT,
                        (HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE),
                                                       NULL) ;

            OldList = (WNDPROC) SetWindowLong  (hwndList, GWL_WNDPROC,
                (LPARAM) ListProc) ;

            SendMessage (hwndList, LB_DIR, DIRATTR, (LPARAM) TEXT ("*.*")) ;
      return 0 ;

case  WM_SIZE :
            rect.right       = LOWORD (lParam) ;
      rect.bottom                  = HIWORD (lParam) ;
      return 0 ;
case  WM_SETFOCUS :
      SetFocus (hwndList) ;
      return 0 ;

case  WM_COMMAND :
      if (LOWORD (wParam) == ID_LIST && HIWORD (wParam) == LBN_DBLCLK)
      {
            if (LB_ERR == (i = SendMessage (hwndList, LB_GETCURSEL, 0, 0)))
                        break ;

            SendMessage (hwndList, LB_GETTEXT, i, (LPARAM) szBuffer) ;
```

```
          if (INVALID_HANDLE_VALUE != (hFile = CreateFile (szBuffer,
                    GENERIC_READ, FILE_SHARE_READ, NULL,
                    OPEN_EXISTING, 0, NULL)))
          {
                    CloseHandle (hFile) ;
                    bValidFile = TRUE ;
                    lstrcpy (szFile, szBuffer) ;
                    GetCurrentDirectory (MAX_PATH + 1, szBuffer) ;

          if (szBuffer [lstrlen (szBuffer) - 1] != '\\')
                    lstrcat (szBuffer, TEXT ("\\")) ;
          SetWindowText (hwndText, lstrcat (szBuffer, szFile)) ;
          }
          else
          {
                bValidFile = FALSE ;
                szBuffer [lstrlen (szBuffer) - 1] = '\0' ;

                    // If setting the directory doesn't work, maybe it's
                    // a drive change, so try that.

          if (!SetCurrentDirectory (szBuffer + 1))
                {
                    szBuffer [3] = ':' ;
                                        szBuffer [4] = '\0' ;
                                        SetCurrentDirectory (szBuffer + 2) ;
                              }

                              // Get the new directory name and fill the list box.

                              GetCurrentDirectory (MAX_PATH + 1, szBuffer) ;
                              SetWindowText (hwndText, szBuffer) ;
                              SendMessage (hwndList, LB_RESETCONTENT, 0, 0) ;
                              SendMessage (hwndList, LB_DIR, DIRATTR,
                             (LPARAM) TEXT ("*.*")) ;
                              }
                    InvalidateRect (hwnd, NULL, TRUE) ;
                }
                return 0 ;
     case  WM_PAINT :
                if (!bValidFile)
                                  break ;

                if (INVALID_HANDLE_VALUE == (hFile = CreateFile (szFile,
               GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL)))
                {
                              bValidFile = FALSE ;
                              break ;
```

```
                }

                ReadFile (hFile, buffer, MAXREAD, &i, NULL) ;
                CloseHandle (hFile) ;

                      // i now equals the number of bytes in buffer.
                      // Commence getting a device context for displaying text.

                hdc = BeginPaint (hwnd, &ps) ;
                SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;
                SetTextColor (hdc, GetSysColor (COLOR_BTNTEXT)) ;
                SetBkColor   (hdc, GetSysColor (COLOR_BTNFACE)) ;

                            // Assume the file is ASCII

                DrawTextA (hdc, buffer, i, &rect, DTFLAGS) ;

                EndPaint (hwnd, &ps) ;
                return 0 ;

      case  WM_DESTROY :
                PostQuitMessage (0) ;
                return 0 ;
      }
      return DefWindowProc (hwnd, message, wParam, lParam) ;
}

LRESULT CALLBACK ListProc (HWND hwnd, UINT message,
               WPARAM wParam, LPARAM lParam)
{
      if (message == WM_KEYDOWN && wParam == VK_RETURN)
                SendMessage (GetParent (hwnd), WM_COMMAND,
               MAKELONG (1, LBN_DBLCLK), (LPARAM) hwnd) ;
      return CallWindowProc (OldList, hwnd, message, wParam, lParam) ;
}
```

在 ENVIRON 中，当我们选择一个环境变数时——无论是使用滑鼠还是键盘——程式都将显示一个环境字串。但是，如果我们在 HEAD 中使用这种选择显示方法，那么程式回应会很慢，这是因为在清单方块中移动选择时，程式仍然要不断地打开和关闭档案。然而，HEAD 要求档案或者子目录被双击，从而引起一些问题，这是因为清单方块控制项没有滑鼠双击的自动键盘介面。前面讲过，如果可能，应该尽量提供键盘介面。

解决的方法是什么呢？当然是视窗子类别化。HEAD 中的清单方块子类则函式叫做 ListProc，它寻找 wParam 参数等於 VK_RETURN 的 WM_KEYDOWN 讯息，并给其父视窗发送一条带有 LBN_DBLCLK 通知码的 WM_COMMAND 讯息。在 WndProc

中，对 WM_COMMAND 的处理使用了 Windows 函式的 CreateFile 来检查清单方块中的选择。如果 CreateFile 传回一个错误资讯，则表示该选择不是档案，而可能是一个子目录。然後 HEAD 使用 SetCurrentDirectory 来改变这个子目录。如果 SetCurrentDirectory 不能执行，程式将假定使用者已经选择了一个磁碟机代号。改变磁碟机也需要呼叫 SetCurrentDirectory，作为该函式参数的字串则为是选择字串中拿掉开头的斜线，并加上一个冒号。它向清单方块发送一条 LB_RESETCONTENT 讯息来清除其中的内容，再发送一条 LB_DIR 讯息，使用新子目录中的档案来填入清单方块。

WndProc 中的 WM_PAINT 讯息是用 Windows 的 CreateFile 函式来打开档案的，这将传回一个档案代号，该代号可以传递给 Windows 的 ReadFile 和 CloseHandle 函式。

现在，在本章中，我们第一次碰到这个问题：Unicode。我们所希望最完美的方式大概就是让作业系统辨认文字档案的种类，使 ReadFile 能将 ASCII 档案转换成 Unicode 文字，或者将 Unicode 档案转换成 ASCII 文字。但现实并非如此完美。ReadFile 的功能只是读取档案中未经转换的位元组，也就是说，DrawTextA（在编译好的可执行档中没有定义 UNICODE 识别字）会把文字解释为 ASCII，而 DrawTextW（Unicode 版）会假设文字是 Unicode 的。

因此程式真正应该做的是去判别档案所包含的是 ASCII 文字还是 Unicode 文字，然後再恰当地呼叫 DrawTextA 或者 DrawTextW。实际上，HEAD 采用一个比较简单的方式，它只呼叫了 DrawTextA。