

表 3-4 消息框类型

类型	按钮	可能返回的代码
MB_ABORTRETRYIGNORE	Abort, Retry, Ignore	IDABORT, IDRETRY, IDIGNORE
MB_OK	OK	IDOK
MB_OKCANCEL	OK, Cancel	IDOK, IDCANCEL
MB_RETRYCANCEL	Retry, Cancel	IDRETRY, IDCANCEL
MB_YESNO	Yes, No	IDYES, IDNO
MB_YESNOCANCEL	Yes, No, Cancel	IDYES, IDNO, IDCANCEL

在有多个按钮的消息框中,第一个(最左边的)按钮通常是默认按钮。您可以将 MB\_DEFBUTTON2 或 MB\_DEFBUTTON3 加入表示特定消息框样式的值中,使第二或第三个按钮成为默认按钮。语句

```
MessageBox(_T("Your document contains unsaved data. Save it?"),
    _T("My Application"), MB_YESNOCANCEL | MB_DEFBUTTON3);
```

显示与前一个相同的消息框,但 Cancel 按钮成为了默认按钮。

在默认情况下,消息框处于应用程序模式,就是说调用 MessageBox 函数的应用程序只有在消息框释放后才能结束。您可以把 MB\_SYSTEMMODAL 加到 nType 参数中,使消息框处于系统模式。在 16 位 Windows 系统中,系统模式意味着直到消息框被释放,所有应用程序的输入都是被挂起的。在 Win32 环境中,Windows 让消息框作为最顶层窗口位于其他窗口上,但是用户仍然可以自由地切换到别的应用程序。系统模式消息框只应该被用在出现了要求立即引起注意的严重错误的情况下。

您可以通过使用 MB\_ICON 标识符给消息框添加一些有趣的东西。MB\_ICONINFORMATION 在消息框的左上角显示一个带有“i”字的小汽球,其中“i”代表“信息”。通常在给用户提供信息且无问题提出时使用“i”,如:

```
MessageBox(_T("No errors found. Click OK to continue"),
    _T("My Application"), MB_ICONINFORMATION | MB_OK);
```

MB\_ICONQUESTION 显示一个问号来替代“i”,通常在查询如“在退出前保存吗?”这样的问题时使用。MB\_ICONSTOP 显示一个带有 X 的红色圆圈,通常说明有不可恢复的错误发生,如:内存溢出错误使程序提前结束。最后,MB\_ICONEXCLAMATION 显示一个包含感叹号的黄色三角形(参见图 3-3)。

MFC 以全局 AfxMessageBox 函数的形式为 CWnd::MessageBox 提供了一个可选对象。虽然两者很相似,但 AfxMessageBox 可以从应用程序类、文档类,以及别的非窗口类中调用。

AfxMessageBox 一个不可替代的用处体现在当您想在应用程序对象的 InitInstance 函数中报告一个错误时。MessageBox 需要一个有效的 CWnd 指针,因此在一个窗口创建之前无法调用它。而 AfxMessageBox 可以在任何时候被调用。

什么? 没有框架窗口?

TicTac 不同于第 1 和第 2 章中示例程序的很重要的一点就是:它的主窗口没有使用框架窗口,而是从 CWnd 中派生了自己的窗口类。并不是 CFrameWnd 不能工作,而是 CWnd 含有 TicTac 需要的所有甚至更多的东西。在 MFC 中,CWnd 是所有窗口类的根。根据编写应用程序的不同,你可能会经常从 CWnd 派生些需要的东西,或许根本什么也不需要。可是每个 MFC 程序员仍然应该知道一些这方面的知识,学习从 CWnd 派生一个窗口类也可以帮助我们明确 MFC 程序没必要非使用框架窗口不可。

创建自己的 CWnd 派生窗口类很简单。对初学者,您可以从 CWnd 而不是从 CFrameWnd 中派生窗口类。在 BEGIN\_MESSAGE\_MAP 宏中,确保指定 CWnd 而不是 CFrameWnd 作为基类。然后,在窗口的构造函数中,使用 AfxRegisterWndClass 来注册 WNDCLASS 并调用 CWnd::CreateEx 创建窗口。还记得在第 1 章的开头,学习了一个 SDK 风格的 Windows 应用程序的 C 语言源代码吗? 在创建一个窗口之前,WinMain 用描述窗口类属性的值初始化了一个 WNDCLASS 结构,然后调用::RegisterClass 来注册 WNDCLASS。通常由于 MFC 已经为您做了,因此在 MFC 程序中不需要注册 WNDCLASS。将 CFrameWnd::Create 的第一个参数指定为 NULL 表示接受默认的 WNDCLASS。但是,当您从 CWnd 派生时,必须注册自己的 WNDCLASS,这是由于 CWnd::CreateEx 不接受 NULL WNDCLASS 名称。

### AfxRegisterWndClass 函数

在 MFC 中,用它的全局函数 AfxRegisterWndClass 注册 WNDCLASS 非常容易。如果您使用::RegisterClass 或 MFC 的 AfxRegisterClass 来注册一个 WNDCLASS,则您必须初始化 WNDCLASS 结构中的每个字段。而 AfxRegisterWndClass 却为您填写了大多数字段,只让您指定 MFC 应用程序通常关心的 4 个值。AfxRegisterWndClass 的原型如下:

```
LPCSTR AfxRegisterWndClass (UINT nClassStyle, HCURSOR hCursor = 0,
    HBRUSH hbrBackground = 0, HICON hIcon = 0)
```

返回值是指向包含 WNDCLASS 名称的非空结尾字符串的指针。在学习 TicTac 如何使用 AfxRegisterWndClass 之前,让我们仔细看一下这个函数本身和它接收的参数。

nClassStyle 指定了类样式,定义了窗口的某种操作特性。nClassStyle 是表 3-5 列出的位标志的零个或多个组合。

表 3-5 WNDCLASS 样式标志

类样式	说 明
CS_BYTEALIGNCLIENT	确保窗口客户区在视频缓冲区总是以一字节边界对齐以加速绘图操作
CS_BYTEALIGNWINDOW	确保窗口自身在视频缓冲区总是以一字节边界对齐以加速绘图和缩放操作
CS_CLASSDC	指定窗口应该与其他从同一个 WNDCLASS 创建的窗口共享设备描述表
CS_DBLCLKS	指定应该用 WM_xBUTTONDBLCLK 消息通知窗口有双击事件发生
CS_GLOBALCLASS	将 WNDCLASS 注册为全局型,以便所有应用程序都可以用它。(默认情况下,只有注册了 WNDCLASS 的应用程序才能使用从它创建的窗口。)主要用于子窗口控件
CS_HREDRAW	指定在窗口被水平缩放时整个客户区无效
CS_NOCLOSE	使系统菜单中的“关闭”命令以及标题栏上的关闭按钮失效
CS_OWNDC	指定由 WNDCLASS 创建的窗口都应该具有自己的设备描述表。对于优化重绘操作这样做是有益的,因为应用程序就可以不必每次在申请设备描述表时都初始化私有设备描述表了
CS_PARENTDC	指定子窗口继承父亲的设备描述表
CS_SAVEBITS	指定当屏幕上的部分内容被由 WNDCLASS 创建的窗口遮住时,这些部分应该以位图形式保存,以便快速重绘。主要用于菜单和其他生命周期短的窗口
CS_VREDRAW	指定在窗口被垂直缩放时整个客户区无效

CS\_BYTEALIGNCLIENT 和 CS\_BYTEALIGNWINDOW 样式在过去无声框架缓冲器 dumb frame buffer 和单色视频系统中很有用,但今天它们已经过时了。CS\_CLASSDC、CS\_OWNDC 以及 CS\_PARENTDC 被用来实现特殊的设备描述表处理。您只有在编写自定义控件来补充列表框、按钮以及其他嵌入控件时,才可能会使用 CS\_GLOBALCLASS。CS\_HREDRAW 和 CS\_VREDRAW 样式对于创建其中内容可随窗口大小变化的可缩放窗口很有用。

hCursor 为由 WNDCLASS 创建的窗口标识“类光标”。当光标在窗口客户区上移动时,Windows 从窗口的 WNDCLASS 中检索类光标的句柄并使用它来绘制光标的图像。您可以使用图标编辑器来创建自定义光标,或使用 Windows 提供的预定义系统光标。CWinApp::LoadStandardCursor 加载一个系统光标。语句

```
AfxGetApp() -> LoadStandardCursor(IDC_ARROW);
```

返回大多数 Windows 应用程序使用的箭头光标句柄。想得到系统光标的一个完整列表,可以参阅 CWinApp::LoadStandardCursor 或::LoadCursor API 函数的帮助文档。一般说来,只有 IDC\_ARROW、IDC\_IBEAM 以及 IDC\_CROSS 光标是有用的类光标。

传递给 AfxRegisterWndClass 的 hbrBackground 参数定义了窗口的默认背景色。具体地说,

hbrBackground 标识 GDI 画刷,它用来在每次 WM\_ERASEBKGD 消息到达时清除窗口内部。在响应 WM\_PAINT 消息而调用 ::BeginPaint 时,窗口接收一个 WM\_ERASEBKGD 消息。如果您不处理 WM\_ERASEBKGD 消息,则 Windows 会检索类背景画刷并使用它填充窗口客户区。(您可以自己处理 WM\_ERASEBKGD 消息并返回非零值来创建自定义窗口,例如背景可以用位图形成。返回的非零值用来防止 Windows 绘制背景覆盖您写的内容。)既可给 hbrBackground 提供一个画刷句柄,也可以指定一个预定义的 Windows 系统颜色并把 1 加到该值上,如 COLOR\_WINDOW + 1 或 COLOR\_APPWORKSPACE + 1。参阅 ::GetSysColor API 函数帮助文档可得到一个完整的系统颜色列表。

最后一个 AfxRegisterWndClass 的参数, hIcon, 指定 Windows 用来在桌面上、任务栏和其他地方代表应用程序的图标句柄。您可以为自己的应用程序创建自定义图标并用 CWinApp::LoadIcon 加载,或用 CWinApp::LoadStandardIcon 来加载预定义的系统图标。甚至可以使用 ::ExtractIcon API 函数从其他可执行文件中加载图标。

以下给出在 TicTac.cpp 中注册自定义 WNDCLASS 的代码:

```
CString strWndClass = AfxRegisterWndClass (
    CS_DBLCLKS,
    AfxGetApp() -> LoadStandardCursor (IDC_ARROW),
    (HBRUSH) (COLOR_3DFACE + 1),
    AfxGetApp() -> LoadStandardIcon (IDI_WINLOGO)
```

类样式 CS\_DBLCLKS 为 TicTac 窗口注册了接收双击消息。IDC\_ARROW 告诉 Windows 当光标在 TicTac 窗口上时显示标准箭头, IDI\_WINLOGO 是对于所有应用程序都有效的 Windows 标准图标之一。COLOR\_3DFACE + 1 指定 TicTac 窗口具有与按钮、对话框一样的背景色和其他的一些 3D 显示特性。COLOR\_3DFACE 默认为亮灰色,但是您可以使用系统的“显示属性”属性表来更改颜色。使用 COLOR\_3DFACE 作为背景色使您的窗口看上去和对话框或消息框的 3D 效果相同,并且能使窗口可以适应 Windows 配色方案中的变化。

### AfxRegisterWndClass 和框架窗口

AfxRegisterWndClass 函数不仅仅用于从 CWnd 派生窗口类的应用程序,您还可以使用它为框架窗口注册自定义的 WNDCLASS。MFC 为框架窗口注册的默认 WNDCLASS 具有下列属性:

- nClassStyle = CS\_DBLCLKS | CS\_HREDRAW | CS\_VREDRAW
- hCursor = 预定义光标 IDC\_ARROW 的句柄
- hbrBackground = COLOR\_WINDOW + 1
- hIcon = 资源 ID 是 AFX\_IDL\_STD\_FRAME 或 AFX\_IDL\_STD\_MDIFRAME 的图标句柄,或者如果没有定义这些资源,则是系统图标 IDI\_APPLICATION 的句柄

假定您想要创建一个没有 CS\_DBLCLKS 样式的 CFrameWnd 框架窗口,它使用 IDI\_WIN-LOGO 图标,并使用 COLOR\_APPWORKSPACE 作为它的默认背景色。以下给出符合这些限制的框架窗口:

```
CString strWndClass = AfxRegisterWndClass (
    CS_HREDRAW | CS_VREDRAW,
    AfxGetApp () -> LoadStandardCursor (IDC_ARROW),
    (HBRUSH) (COLOR_APPWORKSPACE + 1),
    AfxGetApp () -> LoadStandardIcon (IDI_WINLOGO)
);

Create (strWndClass, T("My Frame Window"));
```

这些语句代替了通常在框架窗口的构造函数中出现的语句

```
Create (NULL, _T("My Frame Window"));
```

### 有关 TicTac 窗口的更多内容

在注册了 WNDCLASS 以后,TicTac 调用 CWnd::CreateEx 来创建它的主窗口:

```
CreateEx (0, strWndClass, _T("Tic-Tac-Toe"),
    WS_OVERLAPPED | WS_SYSMENU | WS_CAPTION | WS_MINIMIZEBOX,
    CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
    NULL, NULL);
```

第一个参数指定扩展窗口样式,是零个或多个 WS\_EX 标志的组合。而 TicTac 不需要扩展窗口样式,因此这个参数为 0。第二参数是 AfxRegisterWndClass 返回的 WNDCLASS 名称,第三个是窗口标题。第四个参数是窗口样式。WS\_OVERLAPPED、WS\_SYSMENU、WS\_CAPTION 以及 WS\_MINIMIZEBOX 的组合创建了一个与 WS\_OVERLAPPEDWINDOW 样式相似的窗口,但缺少一个最大化按钮并且无法缩放窗口。是什么使窗口成为不可缩放的呢?在 Winuser.h(Visual C++ 的若干大型头文件之一)中查阅一下 WS\_OVERLAPPEDWINDOW 的定义,您会看到如下的内容:

```
#define WS_OVERLAPPEDWINDOW (WS_OVERLAPPED | WS_CAPTION | \
    WS_SYSMENU | WS_THICKFRAME | WS_MINIMIZE | WS_MAXIMIZE)
```

WS\_THICKFRAME 样式增加了一个其边和角可以用鼠标抓取并拖动的可调整大小的边框。而 TicTac 窗口中缺少此样式,因此用户无法缩放它。接下来的 4 个传递给 CWnd::CreateEx 的参数指定了窗口的初始位置和大小。TicTac 让 4 个参数都使用 CW\_USEDEFAULT 值,而 Windows 将拾取初始位置和大小。然而很明显 TicTac 窗口不应该是任意大小;它的尺寸应该与游戏网格的大小匹配。但怎样做呢?下而这个跟随 CreateEx 调用的语句给出了答案:

```

CRect rect(0, 0, 352, 352);
CalcWindowRect(&rect);

SetWindowPos(NULL, 0, 0, rect.Width(), rect.Height(),
    SWP_NOZORDER | SWP_NOMOVE | SWP_NOREDRAW);

```

第一个语句创建了一个 CRect 对象,它保存着窗口客户区理想的尺寸 352×352 像素大小。由于 CreateEx 的尺寸参数指定的是全部窗口的大小,而不是客户区,因此不会直接把这些值传递给 CreateEx。因为窗口非客户区中许多元素的尺寸(例如,标题栏的高度)随视频驱动程序和显示器图形分辨率的不同而不同,所以必须根据客户区矩形的大小计算窗口矩形的尺寸,并使两者匹配。

MFC 的 CWnd::CalcWindowRect 是完成此工作的最佳工具。给定一个指向包含窗口客户区坐标的 CRect 对象的指针,CalcWindowRect 将计算出相应的窗口矩形。矩形的宽度和高度然后被传递给 CWnd::SetWindowPos,进而得到合适的窗口大小。唯一值得注意的是,CalcWindowRect 必须在窗口创建之后被调用,以便它在窗口非客户区尺寸中生效。

### PostNcDestroy 函数

从 CWnd 派生自己的窗口类时您必须考虑到,一旦它被创建以后,窗口对象最终必须被删除。正如在第2章描述的那样,窗口在被销毁之前接收的最后一个消息是 WM\_NCDESTROY。MFC 的 CWnd 类包括一个默认 OnNcDestroy 处理程序,它执行一些例程来清除杂务,然后作为最后一次操作,将调用名为 PostNcDestroy 的虚函数。当它们所附属的窗口销毁时, CFrameWnd 对象会删除自己,覆盖 PostNcDestroy 函数并执行一个 delete this 语句。CWnd::PostNcDestroy 并不执行 delete this 语句,因此从 CWnd 派生的类应该提供自己的 PostNcDestroy 来执行。TicTac 包括一个小型 PostNcDestroy 函数,它在程序终止以前销毁 CMainWindow 对象:

```

void CMainWindow::PostNcDestroy()
{
    delete this;
}

```

无论何时从 CWnd 派生一个窗口类都应该考虑“到底是谁来删除它”这个问题。除了覆盖 CWnd::PostNcDestroy 以外,还可以覆盖 CWinApp::ExitInstance 并用在 m\_pMainWnd 中存储的指针来调用 delete。

### 3.1.3 非客户区鼠标消息

当鼠标在窗口的非客户区上单击或移动时,Windows 就会给窗口发送一个非客户区鼠标消息。表 3-6 中列出了非客户区鼠标消息。

表 3-6 非客户区鼠标消息

消息	发送条件
WM_NCLBUTTONDOWN	鼠标左键被按下
WM_NCLBUTTONUP	鼠标左键被释放
WM_NCLBUTTONDBLCLK	鼠标左键被双击
WM_NCMBUTTONDOWN	鼠标中间键被按下
WM_NCMBUTTONUP	鼠标中间键被释放
WM_NCMBUTTONDBLCLK	鼠标中间键被双击
WM_NCRBUTTONDOWN	鼠标右键被按下
WM_NCRBUTTONUP	鼠标右键被释放
WM_NCRBUTTONDBLCLK	鼠标右键被双击
WM_NCMOUSEMOVE	在窗口非客户区移动了光标

注意非客户区鼠标消息与客户区鼠标消息很相似,唯一不同的是消息 ID 中的字母 NC。与窗口客户区双击消息不同,WM\_NCXBUTTONDBLCLK 消息无论窗口注册了 CS\_DBLCLKS 样式与否都要被发送。而和客户区鼠标消息一样,消息映射表输入项会把消息传递给相应的类成员函数。表 3-7 列出了非客户区鼠标消息的消息映射宏和消息处理程序。

表 3-7 非客户区鼠标消息的消息映射宏和消息处理程序

消 息	消息映射宏	处理函数
WM_NCLBUTTONDOWN	ON_WM_NCLBUTTONDOWN	OnNcLButtonDown
WM_NCLBUTTONUP	ON_WM_NCLBUTTONUP	OnNcLButtonUp
WM_NCLBUTTONDBLCLK	ON_WM_NCLBUTTONDBLCLK	OnNcLButtonDblClk
WM_NCMBUTTONDOWN	ON_WM_NCMBUTTONDOWN	OnNcMButtonDown
WM_NCMBUTTONUP	ON_WM_NCMBUTTONUP	OnNcMButtonUp
WM_NCMBUTTONDBLCLK	ON_WM_NCMBUTTONDBLCLK	OnNcMButtonDblClk
WM_NCRBUTTONDOWN	ON_WM_NCRBUTTONDOWN	OnNcRButtonDown
WM_NCRBUTTONUP	ON_WM_NCRBUTTONUP	OnNcRButtonUp
WM_NCRBUTTONDBLCLK	ON_WM_NCRBUTTONDBLCLK	OnNcRButtonDblClk
WM_NCMOUSEMOVE	ON_WM_NCMOUSEMOVE	OnNcMouseMove

对于非客户区鼠标消息,OnNcMouseMove 消息处理程序的原型为:

```
afx_msg void OnMsgName(UINT nHitTest, CPoint point)
```

同样,point 参数指定了事件在窗口中发生的位置。但是对于非客户区鼠标消息,point.x 和 point.y 指的是屏幕坐标而非客户区坐标。在屏幕坐标中,(0,0)点指的是屏幕左上角,而 x 和 y 轴正向分别为向右和向下,在任何方向上的一个单位都相当于一个像素。如果愿意,您

可以用 `CWnd::ScreenToClient` 函数将屏幕坐标变换为客户区坐标。`nHitTest` 参数包含标识窗口非客户区上事件发生地方的命中测试码。表 3-8 列出了一些的最有用的命中测试码。在 `WM_NCHITTEST` 或 `CWnd::OnNcHitTest` 的帮助文档中可以找到完整的命中测试码列表。

表 3-8 常用命中测试码

值	相应位置
<code>HTCAPTION</code>	标题栏
<code>HTCLOSE</code>	关闭按钮
<code>HTGROWBOX</code>	还原按钮(与 <code>HTSIZE</code> 相同)
<code>HTHSCROLL</code>	窗口的水平滚动栏
<code>HTMENU</code>	菜单栏
<code>HTREDUCE</code>	最小化按钮
<code>HTSIZE</code>	还原按钮(与 <code>HTGROWBOX</code> 相同)
<code>HTSYSMENU</code>	系统菜单框
<code>HTVSCROLL</code>	窗口的垂直滚动栏
<code>HTZOOM</code>	最大化按钮

程序通常并不处理非客户区鼠标消息,而是让 Windows 为它们处理。Windows 提供了适当的默认响应,而这些响应时常导致更多的消息发送给窗口。例如,当 Windows 用命中测试码的值等于 `HTCAPTION` 来处理一个 `WM_NCLBUTTONDBLCLK` 消息时,它会给窗口发送一个 `WM_SYSCOMMAND` 消息,其中 `wParam` 等于 `SC_MAXIMIZE` 或 `SC_RESTORE` 使窗口最大化或恢复原状态。通过在窗口类中包含下列消息处理程序,您可以阻止在一个标题栏上的双击影响窗口:

```
// In CMainWindow's message map
ON_WM_NCLBUTTONDBLCLK()

.
.
.

void CMainWindow::OnNcLButtonDblClk(UINT nHitTest, CPoint point)
{
    if (nHitTest != HTCAPTION)
        CWnd::OnNcLButtonDblClk(nHitTest, point);
}
```

调用基类的 `OnNcLButtonDblClk` 处理程序把消息传递给 Windows 并且允许默认处理。不调用基类就返回可以避免 Windows 知道双击事件的发生。您还可以使用其他命中测试码对自定义窗口对非客户区鼠标事件的响应。



### 3.1.4 WM\_NCHITTEST 消息

窗口在接收一个客户区或非客户区鼠标消息之前,它先接收到光标的屏幕坐标和 WM\_NCHITTEST 消息。大多数应用程序并不处理 WM\_NCHITTEST 消息,而是让 Windows 处理它们。当 Windows 处理 WM\_NCHITTEST 消息时,首先使用光标坐标来确定光标所在窗口上的位置,然后再产生一个客户区或非客户区鼠标消息。

一个巧妙使用 OnNcHitTest 处理程序的方法是用 HTCAPTION 命中测试码代替 HTCLIENT,创建一个可以在客户区拖动的窗口:

```
// In CMainWindow's message map
ON_WM_NCHITTEST()

...

UINT CMainWindow::OnNcHitTest (CPoint point)
{
    UINT nHitTest = CFrameWnd::OnNcHitTest (point);
    if (nHitTest == HTCLIENT)
        nHitTest = HTCAPTION;
    return nHitTest;
}
```

正如上例说明的那样,您自己不处理的 WM\_NCHITTEST 消息应该传递给基类以便程序的其他方面不受影响。

### 3.1.5 WM\_MOUSELEAVE 和 WM\_MOUSEHOVER 消息

因为窗口接收 WM\_MOUSEMOVE 消息,所以很容易知道何时光标进入了窗口或在窗口中移动了。而 ::TrackMouseEvent 函数首先在 Windows NT 4.0 中被使用,现在也得到了 Windows 98 的支持,它使得确定光标何时离开窗口或在窗口上停止不动很容易。使用 ::TrackMouseEvent,一个应用程序可以注册,当光标离开窗口时接收 WM\_MOUSELEAVE 消息,而光标在窗口上停滞时接收 WM\_MOUSEHOVER 消息。

::TrackMouseEvent 只接收一个参数:一个指向 TRACKMOUSEEVENT 结构的指针。在 Winuser.h 中结构的定义如下:

```
typedef struct tagTRACKMOUSEEVENT {
    DWORD cbSize;
    DWORD dwFlags;
    HWND hwndTrack;
    DWORD dwHoverTime;
} TRACKMOUSEEVENT;
```

cbSize 保存结构的大小。dwFlags 保存位标志用来指定调用者想要执行的操作：注册接收 WM\_MOUSELEAVE 消息(TME\_LEAVE),注册接收 WM\_MOUSEHOVER 消息(TME\_HOVER),取消 WM\_MOUSELEAVE 和 WM\_MOUSEHOVER 消息 (TME\_CANCEL),或允许系统用当前::TrackMouseEvent 设置填写 TRACKMOUSEEVENT 结构(TME\_QUERY)。hwndTrack 是窗口的句柄,对此窗口将生成 WM\_MOUSELEAVE 和 WM\_MOUSEHOVER 消息。dwHoverTime 是以毫秒计时的时间长度,光标必须暂停这么久,WM\_MOUSEHOVER 消息才向它下面的窗口发送。您可以设置 dwHoverTime 等于 HOVER\_DEFAULT,以接受系统提供的默认值 400 毫秒。

光标并非必须纹丝不动系统才产生 WM\_MOUSEHOVER 消息。如果光标所在矩形的宽度和高度与用 SPI\_GETMOUSEHOVERWIDTH 和 SPI\_GETMOUSEHOVERHEIGHT 参数调用::SystemParametersInfo 返回的值相同,并且它停留的时间与用 SPI\_SETMOUSEHOVERTIME 调用 SystemParametersInfo 返回的值也相同时,就会产生 WM\_MOUSEHOVER 消息。如果愿意,您可以用 SPI\_SETMOUSEHOVERWIDTH, SPI\_SETMOUSEHOVERHEIGHT 以及 SPI\_SETMOUSEHOVERTIME 值调用::SystemParametersInfo 来修改这些参数。

有关::TrackMouseEvent 的使用,更有趣的是在产生 WM\_MOUSELEAVE 或 WM\_MOUSEHOVER 消息时,它的影响就消失了。这意味着如果您在光标离开或停留在窗口上时任何时候想接收这些消息,就必须在收到 WM\_MOUSELEAVE 或 WM\_MOUSEHOVER 消息时重新调用::TrackMouseEvent。为便于说明,下列代码断在鼠标进入、离开或停留在窗口上时将“Mouse enter”、“Mouse leave”、或“Mouse hover”写到了调试输出窗口。m\_bMouseOver 是一个 BOOL CMainWindow 成员变量。它应该在类构造函数中被设置为 FALSE:

```
// In the message map
ON_WM_MOUSEMOVE()
ON_MESSAGE(WM_MOUSELEAVE, OnMouseLeave)
ON_MESSAGE(WM_MOUSEHOVER, OnMouseHover)
.
.
.
void CMainWindow::OnMouseMove(UINT nFlags, CPoint point)
{
    if (!m_bMouseOver) {
        TRACE(_T("Mouse enter\n"));
        m_bMouseOver = TRUE;

        TRACKMOUSEEVENT tme;
        tme.cbSize = sizeof(tme);
        tme.dwFlags = TME_HOVER!TME_LEAVE;
        tme.hwndTrack = m_hWnd;
        tme.dwHoverTime = HOVER_DEFAULT;
        ::TrackMouseEvent(&tme);
    }
}
```

```

    }
}

LRESULT CMainWindow::OnMouseLeave (WPARAM wParam, LPARAM lParam)
{
    TRACE (_T("Mouse leave\n"));
    m_bMouseOver = FALSE;
    return 0;
}

LRESULT CMainWindow::OnMouseHover (WPARAM wParam, LPARAM lParam)
{
    TRACE (_T("Mouse hover (x = %d, y = %d)\n"),
        LOWORD (lParam), HIWORD (lParam));

    TRACKMOUSEEVENT tme;
    tme.cbSize = sizeof(tme);
    tme.dwFlags = TME_HOVER | TME_LEAVE;
    tme.hwndTrack = m_hWnd;
    tme.dwHoverTime = HOVER_DEFAULT;
    ::TrackMouseEvent (&tme);
    return 0;
}

```

MFC 并没有为 WM\_MOUSELEAVE 和 WM\_MOUSEHOVER 消息提供特定类型的消息映射宏,因此正如此例说明的那样,您必须使用 ON\_MESSAGE 宏把这些消息与类成员函数连接起来。伴随 WM\_MOUSEHOVER 消息的 lParam 参数值在它的高位字中保存了光标的 y 坐标在低位字中保存光标的 x 坐标。wParam 是未被使用。在 WM\_MOUSELEAVE 消息中 wParam 和 lParam 都未被使用。关于 ::TrackMouseEvent 最后要注意的是:要使用它,必须在源代码中包括下列 #define 语句:

```
#define _WIN32_WINNT 0x0400
```

请确保在 #include Afxwin.h 之前加入此行,否则它将不起作用。

### 3.1.6 鼠标滚轮

Windows 上使用的许多鼠标都有一个滚轮,利用它,不需要单击滚动条就能滚动一个窗口。在滚轮滚动时,有输入焦点的窗口将接收 WM\_MOUSEWHEEL 消息。MFC 的 CScrollView 类为这些消息提供了默认的处理程序,可以自动地滚动窗口,但是如果想用鼠标滚轮消息滚动一个非 CScrollView 窗口,则必须自己处理 WM\_MOUSEWHEEL 消息。

MFC 的 ON\_WM\_MOUSEWHEEL 宏将 WM\_MOUSEWHEEL 消息映射到消息处理程序 OnMouseWheel。OnMouseWheel 的原型如下:

```
BOOL OnMouseWheel (UINT nFlags, short zDelta, CPoint point)
```

nFlags 和 point 参数与传递给 OnLButtonDown 的相同。zDelta 是滚轮旋转的距离。zDelta 等于 WHEEL\_DELTA (120) 意味着滚轮向前旋转了一个增量, 或称为槽口, 而 zDelta 等于 -WHEEL\_DELTA 意味着滚轮向后旋转一单位槽口。如果滚轮向前旋转了 5 个槽口, 则窗口将接收到 5 个 WM\_MOUSEWHEEL 消息, 每个消息都有一个 zDelta 值为 WHEEL\_DELTA。如果 OnMouseWheel 滚动了窗口, 则它应该返回一个非零值, 否则返回零值。

响应 WM\_MOUSEWHEEL 消息的一个简单方法是对每一 WHEEL\_DELTA 单位都将窗口向上滚动一行(如果 zDelta 是正的)或向下滚动一行(如果 zDelta 是负的)。然而, 推荐的方法却稍微有些复杂。首先要询问系统与 WHEEL\_DELTA 单位对应的行数。在 Windows NT 4.0 及其更高版本和 Windows 98 中, 您可以调用 ::SystemParametersInfo, 它的第一个参数等于 SPI\_GETWHEELSCROLLLINES。然后用 zDelta 乘以结果并除以 WHEEL\_DELTA 来确定滚动的行数。通过给 CMainWindow 添加下列消息映射表输入项和消息处理程序, 您可以修改第 2 章给出的 Accel 程序中对 WM\_MOUSEWHEEL 消息的响应:

```
// In the message map
ON_WM_MOUSEWHEEL ()

.
.
.

BOOL CMainWindow::OnMouseWheel (UINT nFlags, short zDelta, CPoint point)
{
    BOOL bUp = TRUE;
    int nDelta = zDelta;

    if (zDelta < 0) {
        bUp = FALSE;
        nDelta = -nDelta;
    }

    UINT nWheelScrollLines;
    ::SystemParametersInfo (SPI_GETWHEELSCROLLLINES, 0,
        &nWheelScrollLines, 0);

    if (nWheelScrollLines == WHEEL_PAGESCROLL) {
        SendMessage (WM_VSCROLL,
            MAKEWPARAM (bUp ? SB_PAGEUP : SB_PAGEDOWN, 0), 0);
    }
    else {
        int nLines = (nDelta * nWheelScrollLines) / WHEEL_DELTA;
        while (nLines--)
```

```

        SendMessage(WM_VSCROLL,
                    MAKEWPARAM(bUp ? SB_LINEUP : SB_LINEDOWN, 0), 0);
    }
    return TRUE;
}

```

如果将来使用的鼠标滚轮的间隔尺寸不到 120 单位,那么用 `zDelta` 除以 `WHEEL_DELTA` 就确保了应用程序不会滚动太快。`WHEEL_PAGESCROLL` 是一个特殊值,它告诉应用程序应该模拟单击滚动条轴,也就是说执行向上或向下滚动一页的操作。`WHEEL_DELTA` 和 `WHEEL_PAGESCROLL` 的定义在 `Winuser.h` 中。

对于此示例程序请注意它与 Windows 95 不兼容。为什么? 因为调用带有 `SPI_GETWHEELSCROLLLINES` 值的 `SystemParametersInfo` 函数在 Windows 95 中不起作用。如果程序想要支持 Windows 95,那么可以假定 `SystemParametersInfo` 将返回 3(默认值)或求助于更复杂的方法来获得用户的参数。MFC 使用称为 `_AfxGetMouseScrollLines` 的内部函数来获取这个值。`_AfxGetMouseScrollLines` 具有平台中立性,它先尝试多种方法来获得滚动行数。如果那些方法都不起作用,则返回默认值 3。如果您在程序中想模拟这种方法,可以参阅 MFC 源代码文件 `Viewscr1.cpp`。

如果鼠标滚轮被单击而非被旋转,则在滚轮按下时,在光标下面的窗口通常会接收到中间键鼠标消息 `WM_MBUTTONDOWN`,当滚轮被释放时,则会接收到 `WM_MBUTTONUP` 消息。(我所谓的“通常”是指默认状态;可以通过控制面板进行修改。)一些应用程序以特殊的方法响应滚轮单击事件。例如在 Microsoft Word 97 中,在滚轮被按下并接收到 `WM_MOUSEMOVE` 消息时将滚动当前显示的文档。知道了鼠标滚轮产生中间键消息,您就可以随心所欲地自定义应用程序来响应鼠标滚轮事件了。

### 3.1.7 捕获鼠标

在处理鼠标消息的程序中时常出现的问题是,在接收到鼠标按下消息后并不一定会跟着接收到鼠标抬起消息。假定您编写了一个绘图程序,它保存传递给 `OnLButtonDown` 的点参数,同时使用这个点作为固定端点并通过光标确定的另一个端点画条直线,也就是所谓的“橡皮筋”画线操作。在 `WM_LBUTTONUP` 消息到达时,应用程序清除橡皮筋线并在原地绘条实线。但是如果用户在释放鼠标键之前,将鼠标移动到了窗口客户区以外,这时会发生什么情况呢? 应用程序不会得到 `WM_LBUTTONUP` 消息,橡皮筋线被遗忘而悬置,实线也不会被画出。

Windows 为此问题提供了一个非常好的解决方法,它允许应用程序“捕获”鼠标,即在接收了鼠标键按下消息之后可以继续接收鼠标消息而不管光标在屏幕的什么地方,直到鼠标键被释放或“捕获”撤消。(在 Win32 环境下,为防止应用程序独占鼠标,系统将停止向鼠标键已释放而仍然拥有捕获能力的窗口发送鼠标消息。)用 `CWnd::SetCapture` 来捕获鼠标,用

::ReleaseCapture 来释放鼠标。对这些函数的调用通常发生在鼠标键按下或抬起处理程序中,如下:

```
// In CMainWindow's message map
ON_WM_LBUTTONDOWN()
ON_WM_LBUTTONUP()
.
.
.

void CMainWindow::OnLButtonDown(UINT nFlags, CPoint point)
{
    SetCapture();
}

void CMainWindow::OnLButtonUp(UINT nFlags, CPoint point)
{
    ::ReleaseCapture();
}
```

在此期间,就算光标离开了,CMainWindow 仍然接收报告光标位置的 WM\_MOUSEMOVE 消息。客户区鼠标消息继续以客户区坐标报告光标位置,但是坐标现在可以为负,也可以超出窗口的客户区尺寸。

相关函数 CWnd::GetCapture 返回一个 CWnd 指针,指向拥有捕获能力的窗口。在 Win32 环境下,如果鼠标没被捕获,或它被属于别的线程的窗口捕获,则 GetCapture 将返回 NULL。GetCapture 的最普通的用处就是确定自己的窗口是否捕获了鼠标。语句

```
if (GetCapture() == this)
```

当且仅当 this 标识的窗口在当前捕获了鼠标才为真值。

捕获鼠标又怎么解决画橡皮筋线的问题呢?通过响应 WM\_LBUTTONDOWN 消息捕获鼠标,并在 WM\_LBUTTONUP 消息到达时释放它,就可以确保当鼠标键释放时得到 WM\_LBUTTONUP 消息了。在下节中的示例程序就说明了这种技术的实际作用。

### 3.1.8 鼠标捕获的应用

图 3-4 中的 MouseCap 应用程序是一个最基本的画图程序,用户可以使用鼠标画直线。要画一条直线,可以在窗口客户区任何地方按下鼠标左键,保持按下状态并拖动光标。在鼠标移动时,会在固定点和光标之间出现一条细的橡皮筋线。当鼠标键释放后,橡皮筋线就会被清除,一条红色 16 像素宽的线将占据它的位置。因为鼠标键被按着时鼠标被捕获了,所以即使鼠标移动到了窗口的外面,橡皮筋线还起作用。而且在鼠标键被释放时,不管光标在

哪儿,一条红色的线都会在固定点和端点之间画出。MouseCap 的源代码如图 3-5 所示。

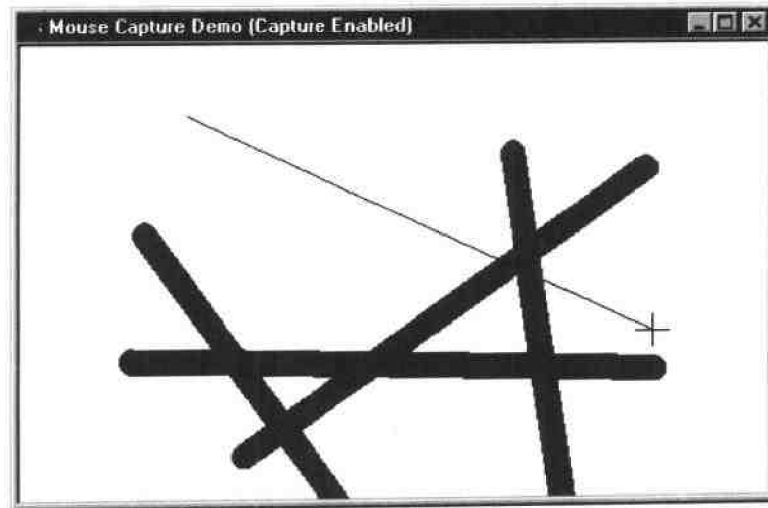


图 3-4 MouseCap 程序窗口

#### MouseCap.h

```
class CMyApp : public CWinApp
{
public:
    virtual BOOL InitInstance();
};

class CMainWindow : public CFrameWnd
{
protected:
    BOOL m_bTracking;           // TRUE if rubber banding
    BOOL m_bCaptureEnabled;     // TRUE if capture enabled
    CPoint m_ptFrom;            // "From" point for rubber banding
    CPoint m_ptTo;              // "To" point for rubber banding

    void InvertLine(CDC * pDC, CPoint ptFrom, CPoint ptTo);

public:
    CMainWindow();

protected:
    afx_msg void OnLButtonDown (UINT nFlags, CPoint point);
    afx_msg void OnLButtonUp (UINT nFlags, CPoint point);
    afx_msg void OnMouseMove (UINT nFlags, CPoint point);
    afx_msg void OnNcLButtonDown (UINT nHitTest, CPoint point);
};
```

---

```

    DECLARE_MESSAGE_MAP ()
};

```

---

### MouseCap.cpp

```

#include <afxwin.h>
#include "MouseCap.h"

CMyApp myApp;

////////////////////////////////////
// CMyApp member functions

BOOL CMyApp::InitInstance ()
{
    m_pMainWnd = new CMainWindow;
    m_pMainWnd->ShowWindow (m_nCmdShow);
    m_pMainWnd->UpdateWindow ();
    return TRUE;
}

////////////////////////////////////
// CMainWindow message map and member functions

BEGIN_MESSAGE_MAP (CMainWindow, CFrameWnd)
    ON_WM_LBUTTONDOWN ()
    ON_WM_LBUTTONUP ()
    ON_WM_MOUSEMOVE ()
    ON_WM_NCLBUTTONDOWN ()
END_MESSAGE_MAP ()

CMainWindow::CMainWindow ()
{
    m_bTracking = FALSE;
    m_bCaptureEnabled = TRUE;
    //
    // Register a WNDCLASS.
    //

    CString strWndClass = AfxRegisterWndClass (
        0,
        AfxGetApp ()->LoadStandardCursor (IDC_CROSS),
        (HBRUSH) (COLOR_WINDOW + 1),
        AfxGetApp ()->LoadStandardIcon (IDI_WINLOGO)
    );
    //
    // Create a window.
    //

```



```

        Create (strWndClass, _T ("Mouse Capture Demo (Capture Enabled)"));
    }

void CMainWindow::OnLButtonDown (UINT nFlags, CPoint point)
{
    //
    // Record the anchor point and set the tracking flag.
    //
    m_ptFrom = point;
    m_ptTo = point;
    m_bTracking = TRUE;
    //
    // If capture is enabled, capture the mouse.
    //
    if (m_bCaptureEnabled)
        SetCapture ();
}

void CMainWindow::OnMouseMove (UINT nFlags, CPoint point)
{
    //
    // If the mouse is moved while we're "tracking" (that is, while a
    // line is being rubber-banded), erase the old rubber-band line and
    // draw a new one.
    //
    if (m_bTracking) {
        CClientDC dc (this);
        InvertLine (&dc, m_ptFrom, m_ptTo);
        InvertLine (&dc, m_ptFrom, point);
        m_ptTo = point;
    }
}

void CMainWindow::OnLButtonUp (UINT nFlags, CPoint point)
{
    //
    // If the left mouse button is released while we're tracking, release
    // the mouse if it's currently captured, erase the last rubber-band
    // line, and draw a thick red line in its place.
    //
    if (m_bTracking) {
        m_bTracking = FALSE;
        if (GetCapture () == this)
            ::ReleaseCapture ();

        CClientDC dc (this);
        InvertLine (&dc, m_ptFrom, m_ptTo);
    }
}

```

---

```

        CPen pen (PS_SOLID, 16, RGB (255, 0, 0));
        dc.SelectObject (&pen);

        dc.MoveTo (m_ptFrom);
        dc.LineTo (point);
    }

|
|
void CMainWindow::OnNcLButtonDown (UINT nHitTest, CPoint point)
|
|
    //
    // When the window's title bar is clicked with the left mouse button,
    // toggle the capture flag on or off and update the window title.
    //
    if (nHitTest == HTCAPTION) {
        m_bCaptureEnabled = m_bCaptureEnabled ? FALSE : TRUE;
        SetWindowText (m_bCaptureEnabled ?
            _T ("Mouse Capture Demo (Capture Enabled)") :
            _T ("Mouse Capture Demo (Capture Disabled)"));
    }
    CFrameWnd::OnNcLButtonDown (nHitTest, point);
|

void CMainWindow::InvertLine (CDC * pDC, CPoint ptFrom, CPoint ptTo)
|
|
    //
    // Invert a line of pixels by drawing a line in the R2_NOT drawing mode.
    //
    int nOldMode = pDC->SetROP2 (R2_NOT);
    pDC->MoveTo (ptFrom);
    pDC->LineTo (ptTo);

    pDC->SetROP2 (nOldMode);
|

```

---

图 3-5 MouseCap 应用程序

大多数操作发生在程序的 OnLButtonDown、OnMouseMove 以及 OnLButtonUp 处理程序中。OnLButtonDown 通过初始化 CMainWindow 类的三个成员变量,开始画图处理:

```

m_ptFrom = point;
m_ptTo = point;
m_bTracking = TRUE;

```

m\_ptFrom 和 m\_ptTo 是橡皮筋线的始点和终点。随着鼠标的移动,m\_ptTo 被 OnMouseMove 处理程序不断更新。m\_bTracking 是一个标志,在鼠标左键按下时为 TRUE,否则为 FALSE,它

告诉 OnMouseMove 和 OnLButtonUp 是否已画了橡皮筋线。如果 m\_bCaptureEnabled 是真,则 OnLButtonDown 的唯一的操作就是捕获鼠标:

```
if (m_bCaptureEnabled)
    SetCapture();
```

m\_bCaptureEnabled 由 CMainWindow 的构造函数初始化为 TRUE。它由窗口的 OnNcLButtonDown 处理程序切换,以便可以打开和关闭鼠标捕获来查看鼠标捕获对程序操作的影响。(待会儿详细讨论。)

OnMouseMove 的工作就是无论何时移动了鼠标,都要移动橡皮筋线并用新的光标位置更新 m\_ptTo。语句

```
InvertLine (&dc, m_ptFrom, m_ptTo);
```

清除以前画的橡皮筋线,而语句

```
InvertLine (&dc, m_ptFrom, point);
```

画一条新的线。InvertLine 是 CMainWindow 的一个成员。它画线不是通过将每个像素设置为某种颜色,而是通过转换现有的像素颜色。这就能确保线条无论在什么背景上都可以被看见,并且只要恢复原有的屏幕颜色就可以清除已有的线。转换是用下列语句将设备描述表的绘图方式设置为 R2\_NOT 完成的:

```
int nOldMode = pDC->SetROP2 (R2_NOT);
```

可以参阅第2章中有关 R2\_NOT 以及其他绘图模式的介绍。

在鼠标左键释放时,调用 CMainWindow::OnLButtonUp。在 m\_bTracking 被设置为 FALSE 并释放鼠标以后,它将清除橡皮筋线,最后调用 OnMouseMove:

```
CClientDC dc (this);
InvertLine (&dc, m_ptFrom, m_ptTo);
```

OnLButtonUp 然后创建一个 16 像素宽原色为红色的笔,将它选入设备描述表,并绘制粗的红色线条:

```
CPen pen (PS_SOLID, 16, RGB (255, 0, 0));
dc.SelectObject (&pen);

dc.MoveTo (m_ptFrom);
dc.LineTo (point);
```

处理结束后,OnLButtonUp 返回,绘图操作完成。图 3-4 显示了 MouseCap 窗口含有一条绘制好的线和新的橡皮筋线的样子。

在熟悉了程序以后,可以单击标题栏激活 OnNcLButtonDown 处理程序,将 m\_bCaptureEn-

abled 标志从 TRUE 切换到 FALSE。窗口标题应该由“Mouse Capture Demo(Capture Enabled)”变为“Mouse Capture Demo(Capture Disabled)”。OnNcLButtonDown 处理在非客户区中的鼠标左键单击,并在 nHitTest 的命中测试码等于 HTCAPTION 时(说明单击发生在标题栏上)使用 CWnd::SetWindowText 来修改窗口标题。

现在在鼠标捕获无效的情况下画几条线。观察:如果在画橡皮筋线时将鼠标移到了窗口外面,则线就会被冻结直到鼠标重新进入客户区,而且如果在窗口外面释放了鼠标键,则程序会失去同步控制。在鼠标回到窗口内部之后,橡皮筋线就会跟随鼠标(即使鼠标键不再被按下),而且永远不会被清除。再次单击标题栏,可以使鼠标捕获恢复功能,程序将恢复到正常状态。

### 3.1.9 光标

MouseCap 并没有使用在大多数 Windows 应用程序中看到的箭头形光标,而是使用了一个十字形光标。箭头和十字形光标是若干 Windows 为您提供的预定义光标类型中的两个,如果这些预定义的光标都不合乎要求,则您可以创建自己的光标。通常,Windows 在这个领域中给程序员提供了很大的创造空间。

首先,学一些光标工作的背景知识。我们知道,每个窗口都有相应的 WNDCLASS,它的特性在 WNDCLASS 结构中定义。WNDCLASS 结构的字段之一是 hCursor,它保存类光标的句柄,该光标就是在窗口客户区上出现的图像。在鼠标移动时,Windows 通过重画光标的背景把光标从旧位置上清除。然后,它给光标下的窗口发送包含命中测试码的 WM\_SETCURSOR 消息。对此消息系统的默认响应是调用::SetCursor,如果命中测试码代码为 HTCLIENT,则显示类光标;如果命中测试码表明光标在客户区以外,则显示箭头。这样光标在屏幕上移动时会得到自动更新。例如:将光标移入一个编辑控件时,它会变化为一条竖直线或 I 形光标。这是由于 Windows 为编辑控件注册了一个专门的 WNDCLASS 类,并指定 I 形光标作为类光标。

因此,改变光标外观的一个方法就是注册 WNDCLASS,并指定希望的光标类型作为类光标。在 MouseCap 程序中,CMainWindow 的构造函数注册了类光标是 IDC\_CROSS 的 WNDCLASS,并把 WNDCLASS 名称传递给了 CFrameWnd::Create:

```
CString strWndClass = AfxRegisterWndClass (
    0,
    AfxGetApp ()->LoadStandardCursor (IDC_CROSS),
    (HBRUSH) (COLOR_WINDOW + 1),
    AfxGetApp ()->LoadStandardIcon (IDI_WINLOGO)
);
```

```
Create (strWndClass, _T("Mouse Capture Demo (Capture Enabled)"));
```

这样,每当鼠标指针在 CMainWindow 的客户区里出现时,Windows 都将显示一个十字光标。

设定光标的第二种方法是调用 API 函数 `::SetCursor` 来响应 `WM_SETCURSOR` 消息。当光标在 CMainWindow 的客户区上时,下列 `OnSetCursor` 函数将显示其句柄在 CMainWindow::`m_hCursor` 中保存的光标:

```
// In CMainWindow's message map
ON_WM_SETCURSOR ()

.
.
.

BOOL CMainWindow::OnSetCursor (CWnd* pWnd, UINT nHitTest,
    UINT message)
{
    if (nHitTest == HTCLIENT) {
        ::SetCursor (m_hCursor);
        return TRUE;
    }
    return CFrameWnd::OnSetCursor (pWnd, nHitTest, message);
}
```

在调用 `::SetCursor` 之后返回 `TRUE`,以通知 Windows 光标已被设置。在窗口的客户区以外产生的 `WM_SETCURSOR` 消息被传递给基类以便显示默认光标。因为 `OnSetCursor` 从不给 Windows 显示类光标的机会,所以类光标被忽略了。

为什么想要使用 `OnSetCursor` 而不采用把 `m_hCursor` 注册为类光标的方法呢?假定当光标在窗口的上半部分时想要显示一个箭头光标,而在下半部分时显示一个 I 形光标。用类光标处理不了这种情况,但是 `OnSetCursor` 可以漂亮地完成任务。当光标在 CMainWindow 的客户区时,下列 `OnSetCursor` 处理程序即可以将光标设置为 `m_hCursorArrow`,也可以为 `m_hCursorIBeam`:

```
BOOL CMainWindow::OnSetCursor (CWnd* pWnd, UINT nHitTest,
    UINT message)
{
    if (nHitTest == HTCLIENT) {
        DWORD dwPos = ::GetMessagePos ();
        CPoint point (LOWORD (dwPos), HIWORD (dwPos));
        ScreenToClient (&point);
        CRect rect;
        GetClientRect (&rect);
        ::SetCursor ((point.y < rect.Height () / 2) ?
            m_hCursorArrow : m_hCursorIBeam);
        return TRUE;
    }
```

```
|  
return CFrameWnd::OnSetCursor (pwnd, nHitTest, message);
```

在 WM\_SETCURSOR 消息从消息队列中被检索到后, ::SendMessagePos 返回一个 DWORD 值, 其中包含光标的 x, y 屏幕坐标。CWnd::ScreenToClient 将屏幕坐标变换为客户区坐标。如果变换后点的 y 坐标小于窗口客户区高度的一半, 则光标被设置为 m\_hCursorArrow。而如果 y 值大于或等于客户区高度的一半, 光标则设置为 m\_hCursorIBeam。在本章稍后提供的 VisualKB 应用程序中也使用了相似的技术, 将进入包围文本输入区域的矩形的光标更改为 I 形光标。

如果需要,可以用下列语句隐藏光标

```
:: ShowCursor (FALSE);
```

要再次显示它,用语句

```
:: ShowCursor (TRUE);
```

在 Windows 内部有一个显示计数器,每当::ShowCursor(TRUE)被调用都加1,而在::ShowCursor(FALSE)调用后减1。如果鼠标已安装计数器,则其初值为0,否则为-1。无论何时只要计数器的值大于或等于0,就显示光标。因此,如果调用了::ShowCursor(FALSE)两次来隐藏光标,就必须调用::ShowCursor(TRUE)两次才能再显示它。

### 3.1.10 沙漏形光标

当应用程序通过执行一个很长的处理任务来响应消息时,通常它会将光标设置为沙漏形,用以提醒用户应用程序很“忙”。(当一个消息处理程序执行时,不从消息队列检索下一个消息,程序冻结输入。在第 17 章,您将学习如何在继续检索并调度消息的同时执行后台处理任务。)

Windows 为您提供了沙漏形光标;它的标识符是 IDC\_WAIT。显示沙漏形光标的简单的方法是在堆栈上声明 CWaitCursor 变量,如下:

```
CWaitCursor wait;
```

CWaitCursor 的构造函数显示沙漏形光标,它的析构函数还原原来的光标。如果在变量超出范围之前,您希望还原光标,可以调用 CWaitCursor::Restore:

```
wait, Restore ();
```

应该在执行那些允许 WM\_SETCURSOR 消息通过并破坏沙漏光标的操作以前调用 Restore，例如在显示一个消息框或对话框之前。

可以通过覆盖 CWinApp 的虚拟函数 DoWaitCursor 来更改由 CWaitCursor::CWaitCursor 和

BeginWaitCursor 显示的光标。以 CWinApp::DoWaitCursor 的默认操作为模板,可以实现自己想要的功能,它在 MFC 源程序代码文件 Appui.cpp 中可以找到。

### 3.1.11 鼠标杂录

前面已经谈到,调用带有 SM\_CMOUSEBUTTONS 参数的 ::GetSystemMetrics API 函数可以查询鼠标键的个数。(在 MFC 中没有等价的 ::GetSystemMetrics,因此必须直接调用它。)通常返回值是 1,2 或 3,但如果是 0,就意味着没有连接鼠标。以如下方式调用 ::GetSystemMetrics 就可以知道是否连接了鼠标:

```
::GetSystemMetrics (SM_MOUSEPRESENT)
```

如果连接了鼠标,则返回非零值,否则为零。在 Windows 应用的早期,程序员不得不考虑某人没有鼠标而使用 Windows 的可能性。但在今天,已经很少关心这个问题了,那些查询系统以确定有没有连接鼠标的程序也确实很少见了。

其他与鼠标有关的 ::GetSystemMetrics 参数还包括 SM\_CXDOUBLECLK 和 SM\_CYDOUBLECLK,它们指定了可以区分出双击事件的两次单击间的最大水平和垂直距离(以像素为单位),还有 SM\_SWAPBUTTON,它在用户使用“控制面板”将鼠标左右键互换时返回非零值。在鼠标键被交换以后,鼠标左键产生 WM\_RBUTTONDOWN 消息,而鼠标右键产生 WM\_LBUTTONDOWN 消息。通常不需要关心这种问题,但是如果出于某种原因您的应用程序想要确定鼠标左键确实就是鼠标左键,那么可以使用 ::GetSystemMetrics 来确定鼠标键是否被交换了。

API 函数 ::SetDoubleClickTime 和 ::GetDoubleClickTime 可以使应用程序设置和检索鼠标双击时间,即在鼠标双击时两次单击之间所允许的的最大时间量。表达式

```
::GetDoubleClickTime ()
```

返回以毫秒为单位的双击时间,而语句

```
::SetDoubleClickTime (250);
```

将双击时间设置为 250 毫秒,即四分之一秒。当同一个鼠标键连续被单击两次后,Windows 将使用双击时间和由 ::GetSystemMetrics 返回的 SM\_CXDOUBLECLK 和 SM\_CYDOUBLECLK 值来确定是否把第二次单击报告为双击。

处理鼠标消息的函数可以通过检查传递给消息处理程序的 nFlags 参数来确定(如果有的话)是哪个鼠标键被按下了。也可以在鼠标消息处理程序外面调用带有 VK\_LBUTTON、VK\_MBUTTON 或者 VK\_RBUTTON 参数的 ::GetKeyState 或 ::GetAsyncKeyState 来查询鼠标键的状态。::GetKeyState 只能从键盘消息处理程序中调用,这是因为它在键盘消息产生时才返回指定鼠标键的状态。::GetAsyncKeyState 却可以在任何地方,任何时候被调用。它在实时状态下工作,在函数调用时返回键的状态。从

```
::GetKeyState (VK_LBUTTON)
```

或

```
::GetAsyncKeyState (VK_LBUTTON)
```

返回负值表明鼠标左键被按下。交换鼠标键不影响 `::GetAsyncKeyState`, 因此如果要使用这个函数, 也应该使用 `::GetSystemMetrics` 来确定鼠标键是否被交换了。表达式

```
::GetAsyncKeyState (::GetSystemMetrics (SM_SWAPBUTTON) ?  
VK_RBUTTON : VK_LBUTTON)
```

以异步方式自动地检查鼠标左键的状态, 如果鼠标键被交换, 则查询鼠标右键的状态。

Windows 提供了一对 API 函数 `::GetCursorPos` 和 `::SetCursorPos`, 用来手工获取和设置光标位置。`::GetCursorPos` 将光标坐标复制到 `POINT` 结构。名为 `::GetMessagePos` 的相关函数返回 `DWORD` 值, 它包含一对 16 位坐标值, 用来指定上一个消息从消息队列中检索到时的光标位置。可以使用 Windows 的 `LOWORD` 和 `HIWORD` 宏来提取那些坐标:

```
DWORD dwPos = ::GetMessagePos ();  
int x = LOWORD (dwPos);  
int y = HIWORD (dwPos);
```

`::GetCursorPos` 和 `::GetMessagePos` 都以屏幕坐标报告光标的位置。通过调用窗口的 `ClientToScreen` 函数可以把屏幕坐标转换为客户区坐标。

Windows 还提供了一个名为 `::ClipCursor` 的函数, 它把光标限制在屏幕的一个特定区域中。`::ClipCursor` 接受一个指向 `RECT` 结构的指针, 该结构在屏幕坐标下描述了剪裁矩形。由于光标是由所有应用程序共享的全局资源, 所以应用程序必须在结束之前调用:

```
::ClipCursor (NULL);
```

来释放它, 否则光标将被永远锁在剪裁矩形中。

## 3.2 从键盘获取输入

Windows 应用程序了解键盘事件的方式与了解鼠标事件的相同: 都是通过消息。任何时候只要一个键被按下或释放, 程序都会接收到一个消息。如果想要知道 `Page Up` 或 `Page Down` 是否被按下, 以便应用程序执行相应的操作, 您可以处理 `WM_KEYDOWN` 消息并检查标识 `Page Up` 或 `Page Down` 键的键代码。如果想知道某键是否被释放, 可以处理 `WM_KEYUP` 消息。对于那些生成可打印字符的键, 可以不管键按下和键抬起消息而直接处理 `WM_CHAR` 消息, 它可以指出从键盘输入的字符。依靠 `WM_CHAR` 消息而不是 `WM_KEYUP/DOWN` 消息可以简化字符处理过程, 因为这样可以把其他事件以及击键时的环境条



件,例如 Shift 键是否被按下,Caps Lock 是打开还是关闭,以及键盘布局不同等等这些事情交给 Windows 处理。

### 3.2.1 输入焦点

和鼠标一样,键盘也是被所有应用程序共享的全局硬件资源。通过标识光标下的窗口,Windows 可以确定给哪个窗口发送鼠标消息。而键盘消息的去向却不是这样。Windows 将键盘消息送到带有“输入焦点”的窗口。在任何时候,只有一个窗口具有输入焦点。通常有输入焦点的窗口是活动应用程序的主窗口。然而,输入焦点也可能属于主窗口的子窗口或在对话框中的控件。不管是谁,Windows 总是将键盘消息送到拥有焦点的窗口。如果您的应用程序的窗口没有子窗口,键盘处理就比较直接:当应用程序处于活动状态,它的主窗口就接收键盘消息。如果焦点转移到子窗口,键盘消息也会转移到子窗口而且以后将停止给主窗口发送消息。

Windows 用 WM\_SETFOCUS 和 WM\_KILLFOCUS 消息通知即将接收或失去输入焦点的窗口,MFC 程序处理如下:

```
// In CMainWindow's message map
ON_WM_SETFOCUS()
ON_WM_KILLFOCUS()

.
.
.

void CMainWindow::OnSetFocus(CWnd* pOldWnd)
{
    // CMainWindow now has the input focus. pOldWnd
    // identifies the window that lost the input focus.
    // pOldWnd will be NULL if the window that lost the
    // focus was created by another thread.
}

void CMainWindow::OnKillFocus(CWnd* pNewWnd)
{
    // CMainWindow is about to lose the input focus.
    // pNewWnd identifies the window that will receive
    // the input focus. pNewWnd will be NULL if the
    // window that's receiving the focus is owned by
    // another thread.
}
```

应用程序可以用 CWnd::SetFocus 把输入焦点转移到另一个窗口:

```
pWnd->SetFocus();
```

它还可以使用静态 `CWnd::GetFocus` 函数找到当前拥有输入焦点的窗口：

```
CWnd* pFocusWnd = CWnd::GetFocus();
```

在 Win32 环境下,如果拥有焦点的窗口还没有被调用线程创建, `GetFocus` 将返回 `NULL`。虽然无法使用 `GetFocus` 得到指向由其他应用程序所创建窗口的指针,但可以使用它标识属于自己应用程序的窗口。

### 3.2.2 击键消息

Windows 通过给拥有输入焦点的窗口发送 `WM_KEYDOWN` 和 `WM_KEYUP` 消息来报告键被按下还是被释放事件。这些消息通常称作击键消息。当一个键被按下时,有输入焦点的窗口会接收到 `WM_KEYDOWN` 消息以及一个标识键的虚拟键代码。当键被释放时,窗口接收到 `WM_KEYUP` 消息。当一个键被按着时,如果另外的键被按下并释放,那么刚产生的 `WM_KEYDOWN` 和 `WM_KEYUP` 消息将会把按着的键产生的 `WM_KEYDOWN` 和 `WM_KEYUP` 消息分开。Windows 按照发生的顺序报告键盘事件,因此只要检查进入应用程序的击键消息流,就可以知道在何时有什么输入。

除了两个键以外所有的键都产生 `WM_KEYDOWN` 和 `WM_KEYUP` 消息。两个例外的键是 `Alt` 和 `F10`,它们是“系统”键,对 Windows 有特殊的意义。它们中任何一个键被按下和释放,窗口都会接收到由 `WM_SYSKEYUP` 消息跟着的 `WM_SYSKEYDOWN` 消息。如果在 `Alt` 键被按着时,别的键被按下了,它们也会产生 `WM_SYSKEYDOWN` 和 `WM_SYSKEYUP` 消息,而不是 `WM_KEYDOWN` 和 `WM_KEYUP` 消息。按下 `F10` 键使 Windows 处于特殊模态下,它将把下一次击键当作菜单选择的快捷方式。例如:在按了 `F10` 之后再按 `F` 键,在大多数应用程序中会拉下“文件”菜单。

应用程序对感兴趣的击键消息提供了消息映射表输入项以及消息处理函数来处理它们。`WM_KEYDOWN`、`WM_KEYUP`、`WM_SYSKEYDOWN` 以及 `WM_SYSKEYUP` 消息分别由类的 `OnKeyDown`、`OnKeyUp`、`OnSysKeyDown` 以及 `OnSysKeyUp` 成员函数处理。相应的消息映射宏是 `ON_WM_KEYDOWN`、`ON_WM_KEYUP`、`ON_WM_SYSKEYDOWN` 以及 `ON_WM_SYSKEYUP`。如果被激活,一个击键处理程序会接收到许多有关击键的信息,其中包括一个代码用来标识被按下或释放的键。

击键消息处理程序的原型如下:

```
afx_msg void OnMsgName (UINT nChar, UINT nRepCnt, UINT nFlags)
```

`nChar` 是被按下或释放的键的虚拟键代码。`nRepCnt` 是重复数,就是消息中击键次数编码。对于 `WM_KEYDOWN` 或 `WM_SYSKEYDOWN` 消息,`nRepCnt` 通常等于 1,对于 `WM_KEYUP` 或 `WM_SYSKEYUP` 消息也是 1。但如果击键消息来的很快,以致应用程序无法跟上,Windows 就将两个或更多的 `WM_KEYDOWN` 或 `WM_SYSKEYDOWN` 消息合并为一个,并相应增加重

复数。大多数程序都忽略重复次数,它们将合并的键按下消息(消息中 `nRepCnt` 大于 1)作为一个消息处理,这样可以防止溢出情况发生,溢出情况是指即使在用户的手指释放了键以后,程序还在继续滚动或继续响应击键消息。与 PC 的键盘 BIOS 相比,它将击键信息存入缓冲区并单独报告每个事件,Windows 报告应用程序中同一个键被连续按下的方法提供了一种根本上防止键盘溢出的措施。

`nFlags` 参数包含了键的扫描码以及下面列出的零个或多个位标志:

位	含义	说 明
0-7	OEM 扫描码	8 位 OEM 扫描码
8	扩展键标志	如果是扩展键则为 1,否则为 0
9-12	保留	N/A
13	上下文代码	如果 Alt 键被按下则为 1,否则为 0
14	先前键状态	如果先前键被按下则为 1,抬起则为 0
15	过渡状态	如果键被按下则为 0,被释放则为 1

扩展键标志允许应用程序区分在大多数键盘上出现的复制键。对于与 IBM 兼容的 PC 机上所使用的 101 和 102 键盘,扩展键标志是为下列键设置的:键盘右边的 Ctrl 和 Alt 键;集中在键盘主体和数字小键盘之间的 Home、End、Insert、Delete、Page Up、Page Down 以及箭头键;还有键区中的回车键(Enter)和斜杠(/)键。对所有其他键,扩展键标志为 0。OEM 扫描码是标识键盘 BIOS 键的 8 位值。因为它本质上依赖硬件,所以大多数 Windows 应用程序都忽略此字段。(如果需要,扫描码可以用 `MapVirtualKey` API 函数转换为虚拟键代码。)过渡状态、先前键状态以及上下文代码通常也会被忽略,但是它们偶尔也有用。先前键状态值等于 1 说明产生了自动重复输入,也就是一个键被按下并保持了一段时间。例如:按下 Shift 键并保持一秒左右,将产生下表顺序的消息:

消息	虚拟键代码	先前键状态
WM_KEYDOWN	VK_SHIFT	0
WM_KEYDOWN	VK_SHIFT	1
WM_KEYDOWN	VK_SHIFT	1
WM_KEYDOWN	VK_SHIFT	1
WM_KEYDOWN	VK_SHIFT	1
WM_KEYDOWN	VK_SHIFT	1
WM_KEYDOWN	VK_SHIFT	1
WM_KEYDOWN	VK_SHIFT	1
WM_KEYDOWN	VK_SHIFT	1
WM_KEYUP	VK_SHIFT	1

如果希望自己的应用程序不管自动重复输入产生的击键事件,您只要让它忽略带有先前键状态值等于1的 WM\_KEYDOWN 消息就可以了。对于 WM\_KEYUP 和 WM\_SYSKEYUP 消息,过渡状态值为0,而对于 WM\_KEYDOWN 和 WM\_SYSKEYDOWN 消息值为1。最后,上下文代码说明在消息产生时 Alt 键是否被按下了。在例外情况下(通常不重要),对于 WM\_KEYDOWN 和 WM\_KEYUP 消息代码为1,而对于 WM\_SYSKEYDOWN 和 WM\_SYSKEYUP 消息值为0。

总的来说,应用程序并不处理 WM\_SYSKEYDOWN 和 WM\_SYSKEYUP 消息,而是让 Windows 处理它们。如果这些消息最终不能到达 ::DefWindowProc,像 Alt-Tab 和 Alt-Esc 这样的系统键盘命令将停止工作。尽管许多消息在操作系统中处于很重要的地位,但 Windows 还是首先把通过应用程序处理这些消息的强大能力交给了您。对于非客户区鼠标消息,如果系统击键消息处理不当,特别是没有把这些消息传递给操作系统,就会导致各种各样奇怪的现象。

### 3.2.3 虚拟键代码

目前传递给击键消息处理程序的最重要的值应该是 nChar,它标识了被按下或释放的键。Windows 用下页表中给出的虚拟键代码来标识键,这样应用程序就不必依赖硬编码值或随键盘不同而不同的 OEM 扫描码了。




显然表中缺少了字母 A 到 Z 和 a 到 z 以及数字 0 到 9 的虚拟键代码。这些键的虚拟键代码与它们相应 ANSI 码一样: 0x41 到 0x5A 对应 A 到 Z, 0x61 到 0x7A 对应 a 到 z, 以及 0x30 到 0x39 对应 0 到 9。

如果看一下 Winuser.h,其中定义了虚拟键代码,您会发现一些键代码没有在表 3-9 中列出,包括 VK\_SELECT, VK\_EXECUTE 以及从 VK\_F13 到 VK\_F24。这些代码用于其他平台操作系统,用常规 IBM 键盘无法生成。Windows 没有给非字符和非数字键提供虚拟键代码,例如:分号(;)和方括号([ ])键,在处理键按下和键抬起消息时最好避免接触它们,因为在国际上它们的 ID 值可能随键盘的不同而不同。这并不是意味着您的程序不能处理标点符号或其他没有 VK\_ 标识符的字符,只是说有比依赖键按下和键抬起消息更好的方法来处理它们。这种“更好的方法”就是通过 WM\_CHAR 消息,我们过一会儿将讨论它。

表 3-9 虚拟键代码

虚拟键代码	相应的键
VK_F1 - VK_F12	功能键 F1 - F12
VK_NUMPAD0-VK_NUMPAD9	数字键 0 - 9 Num Lock 开状态
VK_CANCEL	Ctrl-Break
VK_RETURN	Enter
VK_BACK	Backspace
VK_TAB	Tab

续表

虚拟键代码	相应的键
VK_CLEAR	数字小键盘 5 (Num Lock 关状态)
VK_SHIFT	Shift
VK_CONTROL	Ctrl
VK_MENU	Alt
VK_PAUSE	Pause
VK_ESCAPE	Esc
VK_SPACE	Spacebar
VK_PRIOR	Page Up 和 PgUp
VK_NEXT	Page Down 和 PgDn
VK_END	End
VK_HOME	Home
VK_LEFT	向左箭头
VK_UP	向上箭头
VK_RIGHT	向右箭头
VK_DOWN	向下箭头
VK_SNAPSHOT	Print Screen
VK_INSERT	Insert 和 Ins
VK_DELETE	Delete 和 Del
VK_MULTIPLY	数字小键盘 *
VK_ADD	数字小键盘 +
VK_SUBTRACT	数字小键盘 -
VK_DECIMAL	数字小键盘 .
VK_DIVIDE	数字小键盘 /
VK_CAPITAL	Caps Lock
VK_NUMLOCK	Num Lock
VK_SCROLL	Scroll Lock
VK_LWIN	左 Windows key (  )
VK_RWIN	右 Windows key (  )
VK_APPS	菜单键 (  )

### 3.2.4 Shift 状态及切换

在您为 WM\_KEYDOWN、WM\_KEYUP、WM\_SYSKEYDOWN 或 WM\_SYSKEYUP 消息编写处理程序时,可能需要在决定执行什么操作之前确定 Shift、Ctrl 或 Alt 键是否被按下了。如

同鼠标消息一样,有关 Shift 和 Ctrl 键状态的信息也没有编入键盘消息,因此 Windows 提供了 `::GetKeyState` 函数。给定一个虚拟键代码, `::GetKeyState` 将报告所询问的键是否被按下。表达式

```
::GetKeyState (VK_SHIFT)
```

返回负值说明 Shift 键被按下,否则返回非负值。同样,表达式

```
::GetKeyState (VK_CONTROL)
```

在 Ctrl 键是按下时,返回负值。因此,下列从 `OnKeyDown` 处理程序取出的代码段落中加括号语句只有当 Ctrl-Left 键( Ctrl 键与左箭头键组合)被按下时才执行:

```
if ((nChar == VK_LEFT) && (::GetKeyState (VK_CONTROL) < 0)) {
    .
    .
    .
}
```

要查询 Alt 键,可以用 `VK_MENU` 参数调用 `::GetKeyState`,或简单地在 `nFlags` 参数中检查上下文代码位。通常这些工作也不是必要的,因为如果 Alt 键被按下,窗口将接收到 `WM_SYSKEYDOWN` 或 `WM_SYSKEYUP` 消息而不是 `WM_KEYDOWN` 或 `WM_KEYUP` 消息。也就是说,消息 ID 通常会告诉您所需要知道的有关 Alt 键的一切信息。另外,还可以与 `::GetKeyState` 一起使用标识符 `VK_LBUTTON`、`VK_MBUTTON` 以及 `VK_RBUTTON` 来确定是否有鼠标键被按下了。

应用程序也可以使用 `::GetKeyState` 来确定 Num Lock、Caps Lock 以及 Scroll Lock 键是处于锁定还是关闭状态。返回码的高位显示当前是否有键被按下(高位为 1 时生成负数),而低位(0 位)则显示切换的状态。表达式

```
::GetKeyState (VK_NUMLOCK) & 0x01
```

在 Num Lock 锁定时为非零值,否则为 0。同样的技巧可以用到 `VK_CAPITAL` (Caps Lock)和 `VK_SCROLL` (Scroll Lock) 键上。有一点很重要就是在测试之前要将返回码中除了最低位以外的全部都屏蔽掉,这是因为高位仍然显示键抬起或按下的状态。

在所有情况下, `::GetKeyState` 都是键盘消息生成时,而不是在函数被调用的那一刻,报告键盘键或鼠标键状态的。这是一种特性,不是缺陷,因为这意味着您可以不必担心在消息处理程序开始查询键状态之前键是否被释放了。由于它返回的信息只有在键盘消息从消息队列中被检索到之后才有效,所以 `::GetKeyState` 函数绝对不应该在键盘消息处理程序以外使用。如果您确实需要知道键盘键或鼠标键的当前状态,或者想要在键盘消息处理程序以外检查键盘键或鼠标键,可以使用 `::GetAsyncKeyState` 来代替。

### 3.2.5 字符消息

如果对于键盘输入只依靠键抬起和键按下消息,那么在下面的情形下就会遇到一个问题。假定您正在编写一个文本编辑器,它把报告字符键被按下的消息转换为在屏幕上显示的字符。A 键被按下,带有虚拟键代码等于 0x41 的 WM\_KEYDOWN 消息就到达。在把 A 放置在屏幕上之前,要调用::GetKeyState 来确定 Shift 键是否被按下了。如果是,将输出一个大写字母“A”;否则,输出一个小写“a”。如果是这样,那很好。但是如果 Caps Lock 也被激活了怎么办? Caps Lock 键撤消了 Shift 键的影响,将“A”转换为“a”以及“a”转换为“A”。现在您需要考虑 A 字母的 4 种不同转换了:

虚拟键代码	VK_SHIFT	Caps Lock	结果
0x41	否	关	a
0x41	是	关	A
0x41	否	开	A
0x41	是	开	a

您可能非常希望编写能够处理这些变换和切换所有可能状态的程序来解决这个问题,这样您的工作就被用户可能按下 Ctrl 键这个事实搞复杂了。而且如果您的应用程序在美国以外使用时,问题只能更大,在别处键盘布局与美国的可能完全不同。一个美国用户按下 Shift-0 会输入一个右括号。但是 Shift-0 在大多数国际性键盘产生一个等号而在荷兰键盘上产生撇号。如果您的程序显示的字符与他们输入的字符不匹配,我想用户是不太会欣赏这个程序的吧。

这就是 Windows 提供了::TranslateMessage API 函数的原因。::TranslateMessage 将与字符键有关的击键消息转换为 WM\_CHAR 消息。MFC 提供的消息循环会为您调用::TranslateMessage,因此在 MFC 应用程序中您不必执行特殊任务来将击键消息转换为 WM\_CHAR 消息。当对键盘输入使用 WM\_CHAR 消息时,因为每个 WM\_CHAR 消息都包含一个与 ANSI 字符集(Windows98)或 Unicode 字符集(Windows 2000)中的符号直接映射的字符代码,所以您不必为虚拟键代码和转换状态担心。假设 Caps Lock 没被打开,按下 Shift-A 会产生如下消息序列:

消息	虚拟键代码	字符代码
WM_KEYDOWN	VK_SHIFT	
WM_KEYDOWN	0x41	
WM_CHAR		0x41 ("A")
WM_KEYUP	0x41	
WM_KEYUP	VK_SHIFT	

现在您可以安心忽略键抬起和键按下消息了,因为对于击键事件所需要知道的一切都已编写在 WM\_CHAR 消息中了。如果在 Shift-A 键按下时 Alt 键处于按下状态,应用程序可能会收到 WM\_SYSCHAR 消息:

消息	虚拟键代码	字符代码
WM_SYSKEYDOWN	VK_SHIFT	
WM_SYSKEYDOWN	0x41	
WM_SYSCHAR		0x41 ("A")
WM_SYSKEYUP	0x41	
WM_SYSKEYUP	VK_SHIFT	

由于 Alt 组合键通常针对特殊目的使用,所以大多数应用程序将忽略 WM\_SYSCHAR 消息而处理 WM\_CHAR 消息。

图 3-6 给出了 ANSI 字符集中的字符。ANSI 只有 8 位,所以仅有 256 个可能的字符。Unicode 使用 16 位字符代码,将可能的字符数扩展到了 65 536 个。幸好,在 Unicode 字符集中前 256 个字符与 ANSI 字符集中的 256 个字符相同。因此如下代码:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00:	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
10:	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
20:		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
30:	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40:	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50:	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60:	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70:	p	q	r	s	t	u	v	w	x	y	z	{		}	~	■
80:	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
90:	■	'	'	■	■	■	■	■	■	■	■	■	■	■	■	■
A0:		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	­	®	¯
B0:	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C0:	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D0:	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E0:	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F0:	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

图 3-6 ANSI 字符集



```
case _T('a'):
case _T('A'):
```

对任一个字符集都有效。

在类的消息映射表中 ON\_WM\_CHAR 输入项把 WM\_CHAR 消息传递给成员函数 OnChar, 它的原型如下:

```
afx_msg void OnChar (UINT nChar, UINT nRepCnt, UINT nFlags)
```

nRepCnt 和 nFlags 与在击键消息中的含义相同。nChar 保存了 ANSI 或 Unicode 字符代码。下列代码段落将俘获字母键、回车键以及空格键的按下操作, 都产生 WM\_CHAR 消息:

```
// In CMainWindow's message map
ON_WM_CHAR ()

.
.
.

void CMainWindow::OnChar (UINT nChar, UINT nRepCnt, UINT nFlags)
{
    if (((nChar >= _T('A')) && (nChar <= _T('Z')) ||
        ((nChar >= _T('a')) && (nChar <= _T('z')))) |
        // Display the character
    {
    }
    else if (nChar == VK_RETURN) {
        // Process the Enter key
    }
    else if (nChar == VK_BACK) {
        // Process the Backspace key
    }
}
```

如果您对某个具体的键是否产生 WM\_CHAR 消息还不清楚, 可以通过一个简单的方法弄明白。只要运行一下本书附带的 VisualKB 应用程序并按下相应的键即可。如果该键产生 WM\_CHAR 消息, 消息将在 VisualKB 的窗口里出现。

### 3.2.6 死键消息

有两个键盘消息没有提及, 因为很少有应用程序使用它们。许多国际上使用的键盘驱动程序允许用户通过区分符输入重音符, 先键入代表区分符的“死键”然后再输入字符自身。::TranslateMessage 将与死键相应的 WM\_KEYUP 消息转换为 WM\_DEADCHAR 消息, 并将死键生成的 WM\_SYSKEYUP 消息转换成 WM\_SYSDEADCHAR 消息。Windows 提供了一种逻辑处理将字符消息与这些消息组合产生被强调的字符, 因此死键消息通常被传递去进行默认处理。一些应用程序甚至中断死键消息和显示对应的区分符。跟随死键后的击键将用被强

调的字符代替区分符。这样就给用户提供了可视的反馈防止了死键被“盲”输入。

可以在 MFC 应用程序中处理死键消息,在消息映射中包括 ON\_WM\_DEADCHAR 或 ON\_WM\_SYSDEADCHAR 输入项并提供处理函数 OnDeadChar 和 OnSysDeadChar 即可。在 MFC 帮助文档中可以找到这些函数的说明。

3.2.7 插入符

在字处理程序和其他 Windows 应用程序中,闪烁的竖直条称为插入符,它被用来标记下一个字符插入的地方。在 Windows 应用程序中,插入符所起的作用与在字符模式应用程序中闪烁的下划线光标所起的作用一样。下面给出 MFC 的 CWnd 类提供的 7 个处理插入符函数。在表 3-10 中少了一个基本函数,::DestroyCaret,由于在 MFC 中没有等价的,所以用户必须从 Windows API 直接调用它。

表 3-10 CWND 插入符处理函数

函数	说 明
CreateCaret	由位图创建一个插入符
CreateSolidCaret	创建实线或块插入符
CreateGrayCaret	创建灰线或块插入符
GetCaretPos	检索当前插入符位置
SetCaretPos	设置插入符位置
ShowCaret	显示插入符
HideCaret	隐藏插入符

和鼠标光标一样,插入符也是共享资源。但是又与光标不同,光标是全局的共享资源,而插入符是单线程共享资源,它被运行在同一个线程上的所有窗口共享。为确保合适的处理,使用插入符的应用程序应该遵循以下简单的规则:

- 使用插入符的窗口应该在接收到输入焦点时“创建”插入符,在失去输入焦点时“销毁”插入符。插入符可以用 CreateCaret、CreateSolidCaret 或者 CreateGrayCaret 创建,用 ::DestroyCaret 销毁。
- 在创建了插入符之后,直到调用 ShowCaret 使它可见之前,它是不可见的。插入符可以调用 HideCaret 再次隐藏起来。如果对 HideCaret 进行了两次以上的连续调用,对 ShowCaret 也必须进行相同次数的调用才能使插入符可见。
- 当在 OnPaint 处理程序以外包含插入符的窗口区域里绘图时,应该隐藏插入符以避免显示冲突。在画图完成后重新显示插入符。不需要在 OnPaint 处理程序中隐藏和重新显示插入符,这是因为 ::BeginPaint 和 ::EndPaint 已经为您做了。
- 程序调用 SetCaretPos 来移动插入符,Windows 并不为您移动插入符,处理输入的键

盘消息(或是鼠标消息)并相应地控制插入符是您应用程序的工作。调用 `GetCaretPos` 可以检索插入符当前的位置。

我们知道,当窗口得到输入焦点时接收 `WM_SETFOCUS` 消息,而失去输入焦点时接收 `WM_KILLFOCUS` 消息。当窗口获得输入焦点时,下列 `WM_SETFOCUS` 处理程序将创建插入符,把它定位,并显示它:

```
void CMainWindow::OnSetFocus (CWnd * pWnd)
{
    CreateSolidCaret (2, m_cyChar);
    SetCaretPos (m_ptCaretPos);
    ShowCaret ();
}
```

而当输入焦点失去时, `WM_KILLFOCUS` 处理程序将保存插入符位置,隐蔽它并销毁插入符:

```
void CMainWindow::OnKillFocus (CWnd * pWnd)
{
    HideCaret ();
    m_ptCaretPos = GetCaretPos ();
    ::DestroyCaret ();
}
```

在这些例子中, `m_cyChar` 保存插入符的高度, `m_ptCaretPos` 保存插入符的位置。在焦点失去时,插入符的位置被保存起来,当焦点重新获得时,恢复原来位置。由于任何时刻仅有一个窗口拥有输入焦点,键盘消息被送到有输入焦点的窗口,所以这种对插入符处理的方法确保了“拥有”键盘的窗口也拥有了插入符。

插入符创建函数有两个作用:定义插入符的样子和声明插入符的所有权。插入符实际上是个位图,因此可以给 `CWnd::CreateCaret` 提供一个位图来设定它的外观。但是通常您会发现更好用的是 `CreateSolidCaret` 函数(由于不需要位图所以它更易使用)。`CreateSolidCaret` 生成一个实体块状插入符,根据不同的处理,它可以成为矩形,水平或竖直线条,或介乎它们之间的形状。在上面 `OnSetFocus` 例子中,语句

```
CreateSolidCaret (2, m_cyChar);
```

生成一个竖直线插入符,具有两个像素宽,高度等于当前字体(`m_cyChar`)字符的高度。与字体比例对照是生成插入符的传统方法,但是有的程序也将插入符的宽度与窗口边框的宽度对应。通过调用带有 `SM_CXBORDER` 的 `::GetSystemMetrics` 可以获得边框的宽度。对于固定调距字体,您可能比较喜欢使用块形插入符,它的宽度和高度等于字符的宽度和高度,如:

```
CreateSolidCaret (m_cxChar, m_cyChar);
```

因为存在不同的字符宽度,块形插入符对按比例调距的字体没有多大意义。`CWnd` 的

CreateGrayCaret 函数与 CreateSolidCaret 功能基本相同,不同处在于它创建灰色插入符而不是实体黑色插入符。插入符尺寸以逻辑单位表示,因此如果您在创建插入符之前更改了映象模式,您指定的尺寸也将相应地变换。

上面已经提到,移动插入符是您的工作。CWnd::SetCaretPos 重新定位插入符,它接受包含新建光标位置的 x 和 y 客户区坐标的 CPoint 对象。如果您使用固定调距字体,在一串文本中确定插入符的位置相当直接,可以用字符宽度乘以字符位置来计算字符串中新的 x 偏移量。如果字体是按比例调距的,就必须多做些工作。MFC 的 CDC::GetTextExtent 和 CDC::GetTabbedTextExtent 函数使应用程序可以以逻辑单位确定所用的按比例调距字体的字符串的宽度。(如果字符串包含制表符,就使用 GetTabbedTextExtent。)给定一个字符的位置 n,就可以调用 GetTextExtent 或 GetTabbedTextExtent 来找到前 n 个字符的累积宽度,进而计算出相应插入符的位置。如果字符串“Hello,world”显示的位置由名为 point 的 CPoint 对象指定,dc 是设备描述表对象,下面的语句将插入符放在了“world”中的“w”和“o”之间:

```
CSize size = dc.GetTextExtent (_T("Hello, w"), 8);
SetCaretPos (CPoint (point.x + size.cx, point.y));
```

GetTextExtent 返回一个 CSize 对象,它的 cx 和 cy 成员反映出字符串的宽度和高度。

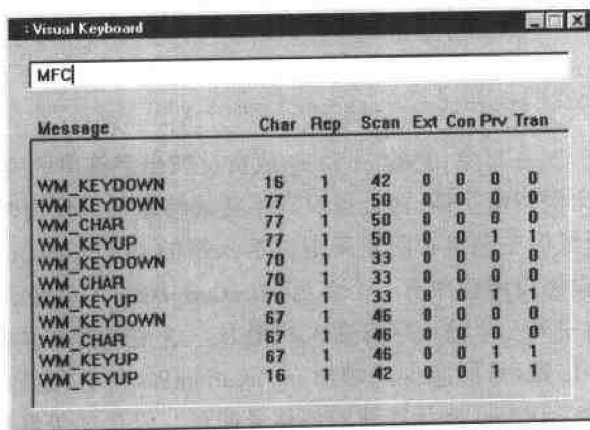
如果您正在使用比例均衡字体,而且也没有字符偏移量可供使用时,插入符定位就会更麻烦一些,在编写处理鼠标左键单击而重新定位插入符的 OnLButtonDown 处理程序时,您就会遇到这种情况。假设您应用程序有一个名为 m\_nCurrentPos 的变量,它指示当前字符的位置,该位置就是字符串中下一个输入字符插入的地方。这很容易在左箭头或右箭头键按下时,计算出新的插入符位置:只要减少或增加 m\_nCurrentPos 值并用新的字符位置调用 GetTextExtent 或 GetTabbedTextExtent 来计算新的偏移量即可。但是如果鼠标左键在字符串中任意位置单击,将怎么办呢?这时鼠标单击和 m\_nCurrentPos 就没关系了,因此您必须使用光标位置和字符串起点的水平差值倒推出字符位置,然后计算出最终的插入符位置。这样就会不可避免地涉及一些迭代计算,因为既没有 Windows API 函数也没有 MFC 类成员函数可以接受字符串和像素的偏移量而返回此偏置处的字符。幸好,自己编写这些函数并不很难。在下一节中将给您介绍。

### 3.3 VISUALKB 应用程序

现在我们把所有学过的知识用来开发一个示例应用程序,该程序键盘输入的文本,在窗口中显示它,并允许用户执行一些简单的编辑工作包括用箭头键和鼠标来移动插入符。为便于教学,我们将滚动显示程序接收的键盘消息以及伴随这些消息的参数,正像 Charles Petzold 的《Programming Windows》中的 KEYLOOK 程序那样。除提供了一个可以实践的鼠标键盘处理教程以外,VisualKB 程序还说明了一些处理按比例对齐文本的技巧。VisualKB 还提

提供了一个方便的工具用来检查从键盘输入的消息流,并且实际查看了特殊击键和键组合产生的消息。

图 3-7 显示了 VisualKB 启动之后输入了“MFC”时的样子。键入字符出现在窗口顶端的文本输入矩形(文本框)中,键盘消息在下面的矩形(消息列表)中显示。第一个和最后一个消息是 Shift 键被按下和释放产生的。在两个消息之间,可以看到由 M、F 以及 C 击键产生的 WM\_KEYDOWN、WM\_CHAR 以及 WM\_KEYUP 消息。在每个消息名称的右边,VisualKB 显示了消息参数。“Char”是 nChar 中传递给消息处理程序的虚拟键代码或字符代码。“Rep”是 nRepCnt 中的重复次数。“Scan”是 OEM 扫描代码,被保存在 nFlags 参数的 0 到 7 位,而“Ext”,“Con”,“Prv”,和“Tran”分别代表扩展键标志、上下文代码、先前键状态、变换状态值。VisualKB 也可以显示 WM\_SYSKEYDOWN、WM\_SYSCHAR 以及 WM\_SYSKEYUP 消息,可以按下 Alt 组合键如 Alt-S 来显示它。



Message	Char	Rep	Scan	Ext	Con	Prv	Tran
WM_KEYDOWN	16	1	42	0	0	0	0
WM_KEYDOWN	77	1	50	0	0	0	0
WM_CHAR	77	1	50	0	0	0	0
WM_KEYUP	77	1	50	0	0	1	1
WM_KEYDOWN	70	1	33	0	0	0	0
WM_CHAR	70	1	33	0	0	0	0
WM_KEYUP	70	1	33	0	0	1	1
WM_KEYDOWN	67	1	46	0	0	0	0
WM_CHAR	67	1	46	0	0	0	0
WM_KEYUP	67	1	46	0	0	1	1
WM_KEYUP	16	1	42	0	0	1	1

图 3-7 输入字符 MFC 后的 VisualKB 窗口

花一点儿时间使用一下 VisualKB,看看按下不同的键和组合键时会发生什么事情。除了可以输入文本以外,还可以使用下列编辑键:

- 左右箭头键可以将插入符移到一个字符的左边和右边。Home 和 End 键可以将插入符移到一行的开头和结尾。插入符能用鼠标单击来移动。
- Backspace 键可以删除插入符左边的字符并将插入符向左移动一个位置。
- Esc 和 Enter 键可以清除文本并将插入符重置到一行的开头。

可打印字符的输入处于替换模式,因此如果插入符不在每行的结尾,输入的下一个字符将替换掉右边的字符。如果输入超出边界线(大约文本框最右端靠左一个字符的位置),文本会被自动清除。我没有急于给程序添加如水平滚动条和插入方式这样的特性,如果这样做会使程序变得不必要地复杂。而且,在实际工作中您完全可以使用文本编辑控件而不必要像本程序那样写这么多代码,文本编辑控件提供了相似的文本输入功能而且还支持剪切、

粘贴和滚动,以及其他特性。除非您正在编写未来世界上最伟大的字处理器,否则文本编辑控件可能已经够您用了。但是,学习用复杂的方法实现文本输入仍然是有益的,因为它不仅有启发性,而且您会了解在开始使用文本编辑控件时 Windows 内部在做什么。

在 VisualKB 的源程序代码中有许多要学习的东西,图 3-8 给出了程序代码。其后的部分指出一些重要的内容。

### VisualKB.h

```
#define MAX_STRINGS 12

class CMyApp : public CWinApp
{
public:
    virtual BOOL InitInstance();
};

class CMainWindow : public CWnd
{
protected:
    int m_cxChar;           // Average character width
    int m_cyChar;           // Character height
    int m_cyLine;           // Vertical line spacing in message box
    int m_nTextPos;         // Index of current character in text box
    int m_nTabStops[7];     // Tab stop locations for tabbed output
    int m_nTextLimit;       // Maximum width of text in text box
    int m_nMsgPos;          // Current position in m_strMessages array

    HCURSOR m_hCursorArrow; // Handle of arrow cursor
    HCURSOR m_hCursorIBeam; // Handle of I-beam cursor

    CPoint m_ptTextOrigin;   // Origin for drawing input text
    CPoint m_ptHeaderOrigin; // Origin for drawing header text
    CPoint m_ptUpperMsgOrigin; // Origin of first line in message box
    CPoint m_ptLowerMsgOrigin; // Origin of last line in message box
    CPoint m_ptCaretPos;     // Current caret position

    CRect m_rcTextBox;       // Coordinates of text box
    CRect m_rcTextBoxBorder; // Coordinates of text box border
    CRect m_rcMsgBoxBorder;  // Coordinates of message box border
    CRect m_rcScroll;       // Coordinates of scroll rectangle

    CString m_strInputText; // Input text
    CString m_strMessages[MAX_STRINGS]; // Array of message strings

public:
    CMainWindow();

protected:
```

```

    int GetNearestPos (CPoint point);
    void PositionCaret (CDC * pDC = NULL);
    void DrawInputText (CDC * pDC);
    void ShowMessage (LPCTSTR pszMessage, UINT nChar, UINT nRepCnt,
        UINT nFlags);
    void DrawMessageHeader (CDC * pDC);
    void DrawMessages (CDC * pDC);

protected:
    virtual void PostNcDestroy ();

    afx_msg int OnCreate (LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnPaint ();
    afx_msg void OnSetFocus (CWnd * pWnd);
    afx_msg void OnKillFocus (CWnd * pWnd);
    afx_msg BOOL OnSetCursor (CWnd * pWnd, UINT nHitTest, UINT message);
    afx_msg void OnLButtonDown (UINT nFlags, CPoint point);
    afx_msg void OnKeyDown (UINT nChar, UINT nRepCnt, UINT nFlags);
    afx_msg void OnKeyUp (UINT nChar, UINT nRepCnt, UINT nFlags);
    afx_msg void OnSysKeyDown (UINT nChar, UINT nRepCnt, UINT nFlags);
    afx_msg void OnSysKeyUp (UINT nChar, UINT nRepCnt, UINT nFlags);
    afx_msg void OnChar (UINT nChar, UINT nRepCnt, UINT nFlags);
    afx_msg void OnSysChar (UINT nChar, UINT nRepCnt, UINT nFlags);

    DECLARE_MESSAGE_MAP ()
};

```

### VisualKB.cpp

```

#include <afxwin.h>
#include "VisualKB.h"

CMyApp myApp;

////////////////////////////////////
// CMyApp member functions

BOOL CMyApp::InitInstance ()
{
    m_pMainWnd = new CMainWindow;
    m_pMainWnd->ShowWindow (m_nCmdShow);
    m_pMainWnd->UpdateWindow ();
    return TRUE;
}

////////////////////////////////////
// CMainWindow message map and member functions

BEGIN_MESSAGE_MAP (CMainWindow, CWnd)

```

```

ON_WM_CREATE ()
ON_WM_PAINT ()
ON_WM_SETFOCUS ()
ON_WM_KILLFOCUS ()
ON_WM_SETCURSOR ()
ON_WM_LBUTTONDOWN ()
ON_WM_KEYDOWN ()
ON_WM_KEYUP ()
ON_WM_SYSKEYDOWN ()
ON_WM_SYSKEYUP ()
ON_WM_CHAR ()
ON_WM_SYSCCHAR ()
END_MESSAGE_MAP ()

CMainWindow::CMainWindow ()
{
    m_nTextPos = 0;
    m_nMsgPos = 0;

    //
    // Load the arrow cursor and the I-beam cursor and save their handles.
    //
    m_hCursorArrow = AfxGetApp () -> LoadStandardCursor (IDC_ARROW);
    m_hCursorIBeam = AfxGetApp () -> LoadStandardCursor (IDC_IBEAM);

    //
    // Register a WNDCLASS.
    //
    CString strWndClass = AfxRegisterWndClass (
        0,
        NULL,
        (HBRUSH) (COLOR_3DFACE + 1),
        AfxGetApp () -> LoadStandardIcon (IDI_WINLOGO)
    );

    //
    // Create a window.
    //
    CreateEx (0, strWndClass, _T ("Visual Keyboard"),
        WS_OVERLAPPED | WS_SYSMENU | WS_CAPTION | WS_MINIMIZEBOX,
        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
        NULL, NULL);
}

int CMainWindow::OnCreate (LPCREATESTRUCT lpCreateStruct)
{
    if (CWnd::OnCreate (lpCreateStruct) == -1)
        return -1;
}

```



```

//
// Initialize member variables whose values are dependent upon screen
// metrics.
//
CClientDC dc (this);

TEXTMETRIC tm;
dc.GetTextMetrics (&tm);
m_cxChar = tm.tmAveCharWidth;
m_cyChar = tm.tmHeight;
m_cyLine = tm.tmHeight + tm.tmExternalLeading;

m_rcTextBoxBorder.SetRect (16, 16, (m_cxChar * 64) + 16,
    ((m_cyChar * 3) / 2) + 16);

m_rcTextBox = m_rcTextBoxBorder;
m_rcTextBox.InflateRect (-2, -2);

m_rcMsgBoxBorder.SetRect (16, (m_cyChar * 4) + 16,
    (m_cxChar * 64) + 16, (m_cyLine * MAX_STRINGS) +
    (m_cyChar * 6) + 16);

m_rcScroll.SetRect (m_cxChar + 16, (m_cyChar * 6) + 16,
    (m_cxChar * 63) + 16, (m_cyLine * MAX_STRINGS) +
    (m_cyChar * 5) + 16);

m_ptTextOrigin.x = m_cxChar + 16;
m_ptTextOrigin.y = (m_cyChar / 4) + 16;
m_ptCaretPos = m_ptTextOrigin;
m_nTextLimit = (m_cxChar * 63) + 16;

m_ptHeaderOrigin.x = m_cxChar + 16;
m_ptHeaderOrigin.y = (m_cyChar * 3) + 16;

m_ptUpperMsgOrigin.x = m_cxChar + 16;
m_ptUpperMsgOrigin.y = (m_cyChar * 5) + 16;

m_ptLowerMsgOrigin.x = m_cxChar + 16;
m_ptLowerMsgOrigin.y = (m_cyChar * 5) +
    (m_cyLine * (MAX_STRINGS - 1)) + 16;

m_nTabStops[0] = (m_cxChar * 24) + 16;
m_nTabStops[1] = (m_cxChar * 30) + 16;
m_nTabStops[2] = (m_cxChar * 36) + 16;
m_nTabStops[3] = (m_cxChar * 42) + 16;
m_nTabStops[4] = (m_cxChar * 46) + 16;
m_nTabStops[5] = (m_cxChar * 50) + 16;
m_nTabStops[6] = (m_cxChar * 54) - 16;

```

```

        //
        // Size the window.
        //
        CRect rect (0, 0, m_rcMsgBoxBorder.right + 16,
                    m_rcMsgBoxBorder.bottom + 16);
        CalcWindowRect (&rect);

        SetWindowPos (NULL, 0, 0, rect.Width(), rect.Height (),
                      SWP_NOZORDER | SWP_NOMOVE | SWP_NOREDRAW);
        return 0;
    }

void CMainWindow::PostNcDestroy ()
{
    delete this;
}

void CMainWindow::OnPaint ()
{
    CPaintDC dc (this);

    //
    // Draw the rectangles surrounding the text box and the message list.
    //
    dc.DrawEdge (m_rcTextBoxBorder, EDGE_SUNKEN, BF_RECT);
    dc.DrawEdge (m_rcMsgBoxBorder, EDGE_SUNKEN, BF_RECT);

    //
    // Draw all the text that appears in the window.
    //
    DrawInputText (&dc);
    DrawMessageHeader (&dc);
    DrawMessages (&dc);
}

void CMainWindow::OnSetFocus (CWnd * pWnd)
{
    //
    // Show the caret when the VisualKB window receives the input focus.
    //
    CreateSolidCaret (max (2, ::GetSystemMetrics (SM_CXBORDER)),
                     m_cyChar);
    SetCaretPos (m_ptCaretPos);
    ShowCaret ();
}

void CMainWindow::OnKillFocus (CWnd * pWnd)
{
    //

```



```

        |
        break;

    case VK_RIGHT:
        if (m_nTextPos != m_strInputText.GetLength()) {
            m_nTextPos++;
            PositionCaret();
        }
        break;

    case VK_HOME:
        m_nTextPos = 0;
        PositionCaret();
        break;

    case VK_END:
        m_nTextPos = m_strInputText.GetLength();
        PositionCaret();
        break;
    }
}

void CMainWindow::OnChar (UINT nChar, UINT nRepCnt, UINT nFlags)
{
    ShowMessage (_T ("WM_CHAR"), nChar, nRepCnt, nFlags);

    CClientDC dc (this);

    //
    // Determine which character was just input from the keyboard.
    //
    switch (nChar) {
    case VK_ESCAPE:
    case VK_RETURN:
        m_strInputText.Empty();
        m_nTextPos = 0;
        break;

    case VK_BACK:
        if (m_nTextPos != 0) {
            m_strInputText = m_strInputText.Left (m_nTextPos - 1) +

                m_strInputText.Right (m_strInputText.GetLength () -
                    m_nTextPos);
            m_nTextPos--;
        }
        break;

    default:

```

```

        if ((nChar >= 0) && (nChar <= 31))
            return;

        if (m_nTextPos == m_strInputText.GetLength()) {
            m_strInputText += nChar;
            m_nTextPos++;
        }
        else
            m_strInputText.SetAt (m_nTextPos++, nChar);

        CSize size = dc.GetTextExtent (m_strInputText,
            m_strInputText.GetLength());

        if ((m_ptTextOrigin.x + size.cx) > m_nTextLimit) {
            m_strInputText = nChar;
            m_nTextPos = 1;
        }
        break;
    }

    //
    // Update the contents of the text box.
    //
    HideCaret ();
    DrawInputText (&dc);
    PositionCaret (&dc);
    ShowCaret ();
}

void CMainWindow::OnKeyUp (UINT nChar, UINT nRepCnt, UINT nFlags)
{
    ShowMessage (_T ("WM_KEYUP"), nChar, nRepCnt, nFlags);
    CWnd::OnKeyUp (nChar, nRepCnt, nFlags);
}

void CMainWindow::OnSysKeyDown (UINT nChar, UINT nRepCnt, UINT nFlags)
{
    ShowMessage (_T ("WM_SYSKEYDOWN"), nChar, nRepCnt, nFlags);
    CWnd::OnSysKeyDown (nChar, nRepCnt, nFlags);
}

void CMainWindow::OnSysChar (UINT nChar, UINT nRepCnt, UINT nFlags)
{
    ShowMessage (_T ("WM_SYSCHAR"), nChar, nRepCnt, nFlags);
    CWnd::OnSysChar (nChar, nRepCnt, nFlags);
}

void CMainWindow::OnSysKeyUp (UINT nChar, UINT nRepCnt, UINT nFlags)
{

```

```

        ShowMessage (_T ("WM_SYSKEYUP"), nChar, nRepCnt, rFlags);
        CWnd::OnSysKeyUp (nChar, nRepCnt, nFlags);
    }

void CMainWindow::PositionCaret (CDC * pDC)
{
    BOOL bRelease = FALSE;

    //
    // Create a device context if pDC is NULL.
    //
    if (pDC == NULL) {
        pDC = GetDC ();
        bRelease = TRUE;
    }

    //
    // Position the caret just right of the character whose 0-based
    // index is stored in m_nTextPos.
    //
    CPoint point = m_ptTextOrigin;
    CString string = m_strInputText.Left (m_nTextPos);
    point.x += (pDC->GetTextExtent (string, string.GetLength ())).cx;
    SetCaretPos (point);

    //
    // Release the device context if it was created inside this function.
    //
    if (bRelease)
        ReleaseDC (pDC);
}

int CMainWindow::GetNearestPos (CPoint point)
{
    //
    // Return 0 if (point.x, point.y) lies to the left of the text in
    // the text box.
    //
    if (point.x <= m_ptTextOrigin.x)
        return 0;

    //
    // Return the string length if (point.x, point.y) lies to the right
    // of the text in the text box.
    //
    CClientDC dc (this);
    int nLen = m_strInputText.GetLength ();
    if (point.x >= (m_ptTextOrigin.x +

```

```

        (dc.GetTextExtent (m_strInputText, nLen)).cx))
        return nLen;

    //
    // Knowing that (point.x, point.y) lies somewhere within the text
    // in the text box, convert the coordinates into a character index.
    //
    int i = 0;
    int nPrevChar = m_ptTextOrigin.x;
    int nNextChar = m_ptTextOrigin.x;

    while (nNextChar < point.x) {
        i++;
        nPrevChar = nNextChar;
        nNextChar = m_ptTextOrigin.x +
            (dc.GetTextExtent (m_strInputText.Left (i), i)).cx;
    }
    return ((point.x - nPrevChar) < (nNextChar - point.x)) ? i - 1 : i;
}

void CMainWindow::DrawInputText (CDC * pDC)
{
    pDC->ExtTextOut (m_ptTextOrigin.x, m_ptTextOrigin.y,
        ETO_OPAQUE, m_rcTextBox, m_strInputText, NULL);
}

void CMainWindow::ShowMessage (LPCTSTR pszMessage, UINT nChar,
    UINT nRepCnt, UINT nFlags)
{
    //
    // Formulate a message string.
    //
    CString string;
    string.Format (_T ("%s\t %u\t %u\t %u\t %u\t %u\t %u\t %u"),
        pszMessage, nChar, nRepCnt, nFlags & 0xFF,
        (nFlags >> 8) & 0x01,
        (nFlags >> 13) & 0x01,
        (nFlags >> 14) & 0x01,
        (nFlags >> 15) & 0x01);

    //
    // Scroll the other message strings up and validate the scroll
    // rectangle to prevent OnPaint from being called.
    //
    ScrollWindow (0, -m_cyLine, &m_rcScroll);
    ValidateRect (m_rcScroll);

    //

```

```

// Record the new message string and display it in the window.
//
CClientDC dc (this);
dc.SetBkColor ((COLORREF) ::GetSysColor (COLOR_3DFACE));

m_strMessages[m_nMsgPos] = string;
dc.TabbedTextOut (m_ptLowerMsgOrigin.x, m_ptLowerMsgOrigin.y,
    m_strMessages[m_nMsgPos], m_strMessages[m_nMsgPos].GetLength (),
    sizeof (m_nTabStops), m_nTabStops, m_ptLowerMsgOrigin.x);

//
// Update the array index that specifies where the next message
// string will be stored.
//
if (++m_nMsgPos == MAX_STRINGS)
    m_nMsgPos = 0;
|

void CMainWindow::DrawMessageHeader (CDC * pDC)
{
    static CString string =
        _T ("Message\tChar\tRep\tScan\tExt\tCon\tPrv\tTran");

    pDC->SetBkColor ((COLORREF) ::GetSysColor (COLOR_3DFACE));
    pDC->TabbedTextOut (m_ptHeaderOrigin.x, m_ptHeaderOrigin.y,
        string, string.GetLength (), sizeof (m_nTabStops), m_nTabStops,
        m_ptHeaderOrigin.x);
|

void CMainWindow::DrawMessages (CDC * pDC)
{
    int nPos = m_nMsgPos;
    pDC->SetBkColor ((COLORREF) ::GetSysColor (COLOR_3DFACE));

    for (int i = 0; i < MAX_STRINGS; i++) {
        pDC->TabbedTextOut (m_ptUpperMsgOrigin.x,
            m_ptUpperMsgOrigin.y + (m_cyLine * i),
            m_strMessages[nPos], m_strMessages[nPos].GetLength (),
            sizeof (m_nTabStops), m_nTabStops, m_ptUpperMsgOrigin.x);

        if (++nPos == MAX_STRINGS)
            nPos = 0;
    }
}

```

图 3-8 VisualKB 应用程序



### 3.3.1 处理插入符

CMainWindow 的 OnSetFocus 和 OnKillFocus 处理程序在 VisualKB 窗口接收到输入焦点后生成插入符,失去焦点时销毁插入符。OnSetFocus 将插入符宽度设置为 2 或调用 ::GetSystemMetrics 返回的 SM\_CXBORDER 值,哪个更大选择哪个,以便插入符即使在很高的图形分辨率显示器上都可见:

```
void CMainWindow::OnSetFocus (CWnd* pWnd)
{
    CreateSolidCaret (max (2, ::GetSystemMetrics (SM_CXBORDER)),
        m_cyChar);
    SetCaretPos (m_ptCaretPos);
    ShowCaret ();
}
```

OnKillFocus 隐藏插入符,保存当前插入符位置以便下次调用 OnSetFocus 时还原它,然后销毁插入符:

```
void CMainWindow::OnKillFocus (CWnd* pWnd)
{
    HideCaret ();
    m_ptCaretPos = GetCaretPos ();
    ::DestroyCaret ();
}
```

在 CMainWindow::OnCreate 中用最左边字符单元的坐标将 m\_ptCaretPos 初始化。在窗口失去输入焦点时,它被当前插入符位置重新初始化。因此,在 OnSetFocus 中对 SetCaretPos 的调用会在程序第一次启动时把插入符放到文本框的开头,而在随后的程序调用中将插入符还原到上次占据的位置。

当左箭头键、右箭头键、Home 键或 End 键被按下时,OnKeyDown 将移动插入符。这些键都不产生 WM\_CHAR 消息,因此 VisualKB 处理 WM\_KEYDOWN。基于 nChar 中的虚拟键代码,switch-case 语句块将执行相应的处理例程。左箭头键(它的虚拟键代码是 VK\_LEFT)的处理程序由下列语句组成:

```
case VK_LEFT:
    if (m_nTextPos != 0) {
        m_nTextPos--;
        PositionCaret ();
    }
    break;
```

m\_nTextPos 是插入文本字符串中下一个字符的位置。文本字符串保存在 CString 对象

`m_strInputText`中。`PositionCaret`是 `CMainWindow` 中保护类型成员函数,它使用 `GetTextExtent` 来找到与保存在 `m_nTextPos` 中的字符位置相对应的文本字符串中像素的位置,然后调用 `SetCaretPos` 将插入符移动到该位置。在核实 `m_nTextPos` 没有超出范围以至使插入符移出左边之后,`VK_LEFT` 处理程序将减少 `m_nTextPos` 的值并调用 `PositionCaret` 来移动插入符。如果 `m_nTextPos` 是 0,表明插入符已经在输入项字段的最左端了,此时键击被忽略。其他 `VK_` 处理程序的操作也同样直接。例如:`VK_END` 处理程序,通过语句

```
m_nTextPos = m_strInputText.GetLength();
PositionCaret();
```

将插入符移动到文本字符串的结尾处。`GetLength`是 `CString` 的成员函数,返回字符串中字符的数量。使用 `CString` 对象保存输入给 `VisualKB` 的文本使得文本处理比以往简单地使用字符数组处理字符串更简单。例如,把一个新字符添加到字符串结尾 `OnChar` 处理程序所要做的只是

```
m_strInputText += nChar;
```

在涉及到字符串处理时,没有比这更容易的了。浏览一下 `VisualKB.cpp`,会看到若干 `CString` 成员函数和运算符,包括 `CString::Left`,它返回包含字符串左边 `n` 个字符的 `CString` 对象;`CString::Right`,返回最右边 `n` 个字符;`CString::Format`,它执行类似 `printf` 的字符串格式化。

整整半章内容讲述了鼠标输入以后,不让 `VisualKB` 执行一些与鼠标有关的操作好像有些说不过去,因此我增加了一个 `OnLButtonDown` 处理程序,它允许在文本框中使用鼠标左键单击来移动插入符。除了给程序添加一个好的功能以外,`OnLButtonDown` 处理程序还允许我们检查一个函数,该函数获取鼠标单击发生的位置,并返回文本字符串中相应字符位置。按键处理程序本身极其简单:

```
void CMainWindow::OnLButtonDown (UINT nFlags, CPoint point)
{
    if (m_rcTextBox.PtInRect (point)) {
        m_nTextPos = GetNearestPos (point);
        PositionCaret ();
    }
}
```

`m_rcTextBox` 是包围文本框的矩形。调用 `CRect::PtInRect` 确定是否有单击在矩形内发生(如果没有则不执行任何操作而返回)之后,`OnLButtonDown` 用 `CMainWindow::GetNearestPos` 为 `m_nTextPos` 计算一个新的值并调用 `PositionCaret` 将插入符重新定位。`GetNearestPos` 首先检查是否在字符串左边鼠标单击发生了,如果发生则为新的字符位置返回 0:

```
if (point.x <= m_ptTextOrigin.x)
    return 0;
```

`m_ptTextOrigin` 保存字符串左上角的坐标。如果鼠标在字符串的最右端以外被单击了, `GetNearestPos` 将返回一个等于字符串长度的整数:

```
CCClientDC dc (this);
int nLen = m_strInputText.GetLength();
if (point.x >= (m_ptTextOrigin.x +
    (dc.GetTextExtent (m_strInputText, nLen)).cx))
    return nLen;
```

结果如何? 如果鼠标在文本矩形内被单击, 但是是在最右边字符的右边, 插入符就会移到字符串的结尾处。

如果 `GetNearestPos` 跳过 `return nLen` 语句, 我们就知道在文本框中被单击的光标位置处于字符串左端和右端之间。 `GetNearestPos` 接下来将初始化 3 个变量并执行 `while` 循环, 反复调用 `GetTextExtent` 直到 `nPrevChar` 和 `nNextChar` 保存的值刚好包括单击发生处的 `x` 坐标值:

```
while (nNextChar < point.x) {
    i++;
    nPrevChar = nNextChar;
    nNextChar = m_ptTextOrigin.x +
        (dc.GetTextExtent (m_strInputText.Left (i), i)).cx;
}
```

在退出循环后, `i` 中保存单击发生处右边字符位置, `i - 1` 则保存左边字符的位置。查找字符位置比较简单, 只要确定 `point.x` 靠 `nNextChar` 近还是靠 `nPrevChar` 近, 并返回 `i` 或 `i - 1` 即可。用如下一行语句就可实现:

```
return ((point.x - nPrevChar) < (nNextChar - point.x)) ? i - 1 : i;
```

这样, 给定窗口客户区中的任意点, `GetNearestPos` 就可以返回字符串 `m_strInputText` 中相匹配的字符位置。这种处理过程效率不太高, 这是因为距离点的右边越远, 调用 `GetTextExtent` 的次数也越多。(while 循环从字符串最左端字符开始每次向右移动一个字符, 直到找到单击发生处右边的字符为止。)通过使用二分法可以使 `GetNearestPos` 的执行效率提高: 首先从字符串的中点开始将字符串分成具有相同字符数的两段, 再对左边或右边未检索过的字符串重复相同的工作, 直到找到这样的一个点, 在它左边和右边的字符数为 0, 这个位置就是单击发生的地方。在 128 个字符长的字符串中, 用此方法不超过 8 次调用 `GetTextExtent` 即可确定字符位置。而 `GetNearestPos` 中使用的笨办法却需要 127 次调用。

### 3.3.2 输入及编辑文本

处理输入和编辑文本的方法在 `CMainWindow::OnChar` 实现。 `OnChar` 的处理策略归结如下:

1. 将消息返回给屏幕。

2. 使用 `nChar` 中的字符代码修改文本字符串。
3. 在屏幕上绘制修改后的文本字符串。
4. 重新给插入符定位。

步骤 1 由调用 `CMainWindow::ShowMessage` 来完成,下一节再讨论它。步骤 2 中文本字符串的修改依赖于 `nChar` 中的字符代码。如果字符是换码符或换行符(`VK_ESCAPE` 或 `VK_RETURN`),则 `m_strInputText` 被 `CString::Empty`(另一个便于使用的 `CString` 类成员)清空,且 `m_nTextPos` 设置为 0。如果字符是退格键(`VK_BACK`)且 `m_nTextPos` 不是 0,则在 `m_nTextPos - 1` 处的字符被删除,`m_nTextPos` 减 1。如果字符是 0 和 31 之间且包括 0 和 31 的任何值,它就被忽略。如果 `nChar` 代表任何其他字符,就在当前字符位置添加到 `m_strInputText` 中并且 `m_nTextPos` 相应加 1。

将现在刚输入的字符添加到 `m_strInputText` 之后,`OnChar` 隐藏插入符并进行到步骤 3。修改后的字符串用 `CMainWindow::DrawInputText` 输出到屏幕,而 `CMainWindow::DrawInputText` 接着依靠 `CDC::ExtTextOut` 来执行文本输出。`ExtTextOut` 与 `TextOut` 相似,但它具有一些 `TextOut` 没有的选项。其中的一个选项是 `ETO_OPAQUE` 标志,用设备描述表的当前背景颜色填充包围文本的矩形框。如果字符串的新宽度小于先前的宽度,整个矩形重画就会删除从前文本输出操作留下的结果。围绕文本框的边框(消息列表周围的边框)是用 `CDC::DrawEdge` 函数绘制的,该函数进而再调用 `DrawEdge API` 函数。`DrawEdge` 是绘制 3D 边框的简便方法,用它绘制的 3D 边框遵循 Windows 界面指南中的规定,而且可以自动适应系统中加亮和阴影颜色的变化。可以使用相关的 `CDC` 函数 `Draw3dRect`,由自己选择颜色来绘制简单的 3D 矩形。

`OnChar` 结束时调用 `PositionCaret` 用 `m_nTextPos` 中保存的值重新设置插入符的位置,然后调用 `ShowCaret` 重新显示插入符。作为试验,评价 `OnChar` 调用 `HideCaret` 和 `ShowCaret` 的作用,可以重新编译一下系统,在文本输入域中输入一些字符。这个简单的练习会使您明白为什么在插入符后面绘出文本之前隐藏插入符是多么的重要。

### 3.3.3 其他有趣内容

当在 `VisualKB` 窗口内移动光标时,注意到光标会在文本窗口外变为箭头形,在窗口内变为 I 形。`CMainWindow` 的构造函数用 `NULL` 类光标注册了 `WNDCLASS`,并且在成员变量 `m_hCursorArrow` 和 `m_hCursorIBeam` 中为系统的箭头和 I 形光标保存了句柄。每当 `CMainWindow` 接收到 `WM_SETCURSOR` 消息后,它的 `OnSetCursor` 处理程序就检查当前的光标位置并调用 `SetCursor` 来显示适当的光标。

在每次收到消息时,`VisualKB` 都会 `CMainWindow::ShowMessage` 来回应键盘消息。`ShowMessage` 在 `CString::Format` 的帮助下格式化新的输出字符串,将结果复制到 `m_strMessages` 数组中最近很少用到的输入项中,将消息列表向上滚动一行,并调用 `CDC::Tabbed-`

TextOut 来显示最后一行新的消息字符串。TabbedTextOut 用来代替 TextOut 以便输出结果中列可以被恰当地对齐。(没有制表符,几乎不能将使用按比例调距字体的字符在列格式中排列对齐。)制表位设置在 OnCreate 中进行初始化,初始化用的是保存在 m\_nTabStops 数组中基于默认字体平均字符宽度的值。消息字符串保存在 m\_strMessages 数组中,因此 OnPaint 处理程序可以在必要时重绘消息显示。CMainWindow 数据成员 m\_nMsgPos 标记数组中的当前位置,在其中复制下一个字符串的数组元素的索引号。m\_nMsgPos 在每次调用 ShowMessage 时都加 1,当达到数组极限值时返回到 0,这样在 m\_strMessages 中可以保存最近接收到的 12 个键盘消息记录。

VisualKB 的 CMainWindow 类包括 OnKeyUp、OnSysKeyDown、OnSysKeyUp 以及 OnSysChar 处理程序,其唯一目的就是将键盘消息回应给屏幕。在返回以前,每个消息处理程序都小心地调用基类中对应的消息处理程序,如下:

```
void CMainWindow::OnSysKeyDown (UINT nChar, UINT nRepCnt, UINT nFlags)
{
    .
    .
    .
    CWnd::OnSysKeyDown (nChar, nRepCnt, nFlags);
}
```

非客户区鼠标消息和系统键盘消息通常是其他消息的催化剂,因此把它们提交给基类执行默认处理是很重要的。

## 第 4 章 菜 单

迄今为止,我们编制的程序还缺少一个所有 Microsoft Windows 应用程序都具有的特性:菜单。现在该是学习如何将菜单放置在我们的代码中,进而弥补程序缺陷的时候了。

下拉式菜单可能是世界上最常见的用户界面元素。几乎每个坐在计算机前浏览菜单的人都知道:单击某个菜单项,就会出现一个命令下拉表。即使一个初学使用计算机的人在看过一、两次示范菜单动作后也能很快掌握并使用。许多计算机用户还记得如何使用新的 MS-DOS 应用程序吧。为完成基本任务,用户要学会多个不直观的键的组合并记住各种晦涩难懂的命令。菜单是 1970 年在 Xerox 著名的 Palo Alto Research Center (PARC) 进行的研究中诞生的,在 1980 年由 Apple Macintosh 推广应用。通过制作简明且方便易用的命令列表,计算机的使用大大简化了。用户只需简单地指定并单击即可选中命令。在 Windows 程序中菜单不是必需的,但是它确实简化了操作。程序和它的命令结构越复杂,基于菜单的用户界面就越显示其优越性。

由于菜单对用户界面来说如此重要,Windows 为使用菜单的应用程序提供了大量的支持。操作系统在管理菜单方面做了大部分工作,包括显示菜单栏,菜单栏中某项被单击时显示一个下拉菜单,以及菜单项被选中时通知应用程序。MFC 进一步增强了菜单处理模型,它将菜单项命令与指定的类成员函数联系起来,并提供了一种新型机制使菜单项与应用程序的状态保持同步,等等。

本章从回顾菜单操作并编制一个菜单的初级程序讲起,进而深入到一些较高级的话题,并创建一个具有某些辅助功能的应用程序。

### 4.1 菜单基础

首先定义几个术语。出现在窗口顶端的菜单栏叫作应用程序“顶层菜单”,其中的命令称为“顶层菜单项”。最高级菜单项被单击时出现的菜单叫作“下拉菜单”,其中的菜单项则称为“菜单项”。菜单项由整型数 menu item IDs(菜单 ID)或 command IDs(命令 ID)唯一确定。Windows 还支持“弹出式菜单”。这种菜单在外观上很像下拉式菜单,但是可以在屏幕的任一位置弹出。用右键单击 Windows shell 中的对象时,弹出的上下文菜单就是一例。下拉式菜单实际上也是弹出式菜单,只是它是应用程序最高级菜单的子菜单。

大部分高级窗口都具有“系统菜单”,包括命令如恢复、移动、尺寸、放大、缩小以及关闭窗口。只要用鼠标左键单击窗口标题栏中的小图标,或用右键单击标题栏体,或同时按下

Alt 和空格键,Windows 中的菜单就能显示。

MFC封装了可以在 CMenu 类中执行的菜单和动作。CMenu 含有一个公用数据成员——即名为 m\_hMenu 的 HMENU,它持有相应菜单的句柄,以及几个提供了面向对象的 Windows API函数封装的成员函数。例如: CMenu::TrackPopupMenu 显示上下文菜单,而 CMenu::EnableMenuItem使菜单项有效或灰化。CMenu 还包含一对虚函数: DrawItem 和 MeasureItem,在创建包含位图和其他用户界面图形元素风格的菜单项时,可以重载这对函数。

在 MFC 应用程序中可以用以下 3 种方式创建菜单:

- 用编程方法创建菜单,调用 CreateMenu、InsertMenu 和其他 CMenu 函数将各部分连接起来。
- 将一系列定义菜单内容的数据结构初始化,并用 CMenu::LoadMenuIndirect 创建菜单。
- 创建菜单资源,并在应用程序运行时加载生成的菜单。

第 3 种方法是最常见的。因为它允许使用资源编辑器或者(如果您喜欢)简单的文本编辑器离线定义一个菜单。本章前半部分将集中讲述这种方法。

#### 4.1.1 创建菜单

创建菜单最简单的方法是:向您的应用程序资源文件(resourcefile)中添加一个菜单模板。resource file 是一个脚本式的文本文件,它定义应用程序的资源。按惯例它的文件扩展名为.rc,因此常称它为 RC 文件。resource 是一个二进制对象,如菜单或图标。Windows 支持几种资源,包括(但是不限于)菜单、图标、位图和字符串。Windows Software Development Kit (SDK)支持资源编译程序 Rc.exe。并且资源编译程序 Rc.exe 也是 Microsoft Visual C++ 的一部分。它编译 RC 文件中的语句,并将生成的资源链接到应用程序的 EXE 文件。每个资源都由一个字符串或一个整型 ID 唯一确定,比如 MyMenu(字符串)或 IDR\_MYMENU(整型数)。整型资源 ID 具有头文件中用 # define 语句定义的有涵义的名字,如 IDR\_MYMENU。一旦资源被编译并链接到 EXE 文件,它就能通过一个简单的函数调用被加载到程序中。

菜单模板包含了资源编译程序创建菜单资源时需要的所有信息,包括菜单的资源 ID、菜单项名称和菜单项的 ID。图 4-1 所示的菜单模板是由 Visual C++ MFC AppWizard 创建的项目生成的。它定义的菜单包含 1 个最高级菜单和 4 个子菜单——文件、编辑、视图和帮助。IDR\_MAINFRAME 是这个菜单的资源 ID。PRELOAD 和 DISCARDABLE 是资源的属性。PRELOAD 告诉 Windows 在应用程序开始运行时将菜单资源装载到内存。如果因为其他原因需要用到该资源占用的内存,DISCARDABLE 允许 Windows 将资源卸载。(如果再次用到已卸载资源,可以将它从应用程序的 EXE 文件重新装载进来。)PRELOAD 和 DISCARDABLE 都是 16 位 Windows 的产品,对 32 位应用程序的执行和性能毫无作用。

---

```

IDR_MAINFRAME MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&New\tCtrl + N",    ID_FILE_NEW
        MENUITEM "&Open..\tCtrl + O",    ID_FILE_OPEN
        MENUITEM "&Save\tCtrl + S",    ID_FILE_SAVE
        MENUITEM "Save &As...",    ID_FILE_SAVE_AS
        MENUITEM SEPARATOR
        MENUITEM "Recent File".    ID_FILE_MRU_FILE1,GRAYED
        MENUITEM SEPARATOR
        MENUITEM "E&xit",    ID_APP_EXIT
    END
    POPUP "&Edit"
    BEGIN
        MENUITEM "&Undo\tCtrl + Z", ID_EDIT_UNDO
        MENUITEM SEPARATOR
        MENUITEM "Cu&t\tCtrl + X", ID_EDIT_CUT
        MENUITEM "&Copy\tCtrl + C", ID_EDIT_COPY
        MENUITEM "&Paste\tCtrl + V", ID_EDIT_PASTE
    END
    POPUP "&View"
    BEGIN
        MENUITEM "&Toolbar", ID_VIEW_TOOLBAR
        MENUITEM "&Status Bar", ID_VIEW_STATUS_BAR
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&About MyApp...", ID_APP_ABOUT
    END
END

```

---

图 4-1 MFC AppWizard 生成的菜单模板

在 BEGIN 和 END 间的语句定义了菜单的内容;而 POPUP 语句定义了最高级菜单项和相关的子菜单。在 POPUP 语句之后,位于 BEGIN 和 END 语句之间的 MENUITEM 语句定义了子菜单的各项。MENUITEM SEPARATOR 语句在菜单中添加了一条细的水平线,从而在视觉上将菜单项分组。菜单项正文中的“&”定义了和 Alt 键一起使用的快捷键,用来显示子菜单和选中子菜单项。在这个例子中,先按 Alt-F,而后再按 X 键就能选中 File 中的 Exit(“文件”中的“退出”)命令。Windows 在 File 中的 F 和 Exit 中的 x 下画横线,使用户很容易明白它们是快捷键。如果同一菜单下给两个或两个以上的菜单项分配了同一个快捷键,快捷方式



就会在菜单项间循环,直到按下 Enter 键才会选中某菜单项。

菜单项正文中的省略号(...)表示该项选中后还需要进一步输入。如果用户选中 Save,文档则立刻被保存。但是如果用户选择了 Save As,则会出现一个对话框。为了和其他 Windows 应用程序保持一致,对于需要等待用户进一步输入的菜单项都一律使用省略号。如果最高级菜单没有显示子菜单而是执行了一个命令,则该项正文后要接一个惊叹号。如下所示:

```
IDR_MAINFRAME MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "&File"
        [...]
    POPUP "&Edit"
        [...]
    POPUP "&View"
        [...]
    POPUP "&Help"
        [...]
    MENUITEM "E&xit!", ID_APP_EXIT
END
```

按这种方式将 MENUITEM 语句写在最高级菜单中是合法的,但是如今大家公认这是种糟糕的形式。由于大部分用户习惯于看到最高级菜单显示子菜单而不是执行动作,所以这样做可能会使他们感到很怪异。

MENUITEM 语句中跟在菜单项名称后的 ID\_值是命令 ID。每个菜单项都应该赋给一个唯一的命令 ID。因为在用户选择时,它确定了您自己应用程序的菜单项。按规定,ID 由 #define 语句定义,每个赋得的名称都由 ID\_或 IDM\_与后面大写的菜单项名称组成。MFC 的 Afxres.h 头文件定义了一些常用命令的 ID\_值,比如 File-New 和 Edit-Paste。编写文档/视图应用程序时,使用这些预先定义好的 ID 能自动将某个菜单项连接到主结构提供的处理程序上。在非文档/视图应用程序中,应考虑是否选用预先定义的 ID。

菜单项 ID 的有效值范围是 1~0xEFFF,但是 MFC Technical Note #20 建议将范围限制在 0x8000~0xDFFF。0xF000 或更高的 ID 值则预留给 Windows——尤其是系统菜单中的各项。范围 0xE000~0xEFFF 则预留给 MFC。实际上,用低于 0x8000 的值是完全可以放心的,而且将菜单项 ID 限制在 1~0x7FFF 范围内能避免 Windows 95 中的一个可怕错误,该错误会影响用户绘制的菜单项。本章将在后面解释这个问题并提出解决的办法。

某些菜单项里跟在制表符后的文本(例如:"Open... AtCtrl + O"中的"Ctrl + O")表示加速键。加速键是一个键或几个键的组合。在被按下时,加速键同选中菜单项的作用相同。常用的加速键包括相对于 Edit-Cut 的 Ctrl-X、相对于 Edit-Copy 的 Ctrl-C 和相对于 Edit-Paste 的 Ctrl-V。为了使文本对齐,制表符放在表示加速键的字符串的前面。菜单中的默认字体是按比例间隔开的,所以用空格对齐菜单文本是没有意义的。

用 `MENUITEM` 定义菜单项时,还需要选择菜单项的初始状态。图 4-1 中伴随 `File-Recent File` 命令使用的关键字 `GRAYED` 使菜单项无效,使它不能被选中。将一个无效的菜单项灰化可以在视觉上提醒用户:这个菜单项是无效的。灰化菜单的正文是以系统颜色 `COLOR_GRAYTEXT` 显示的,它的默认值为灰色,并带有细边框造成三维效果。另一个可选的关键字是 `CHECKED`,它在菜单项旁边添加了一个复选标记。尽管确定菜单项状态在 SDK 支持的应用 C 语言编写的 Windows 应用程序中很常见,但在 MFC 应用程序中却很少用到。因为主结构提供了一个强大的机制,允许运用编程方法更新菜单项。不久您就能更多地了解这个机制。

### 4.1.2 加载并显示菜单

在运行时,需要加载菜单资源并将它挂接到窗口。窗口显示时,菜单也一同显示了。

将菜单挂接到窗口上的一种方法是将菜单的资源 ID 传递给 `CFrameWnd::Create`。下面的语句创建了一个主窗口并挂接了资源 ID 为 `IDR_MAINFRAME` 的菜单:

```
Create(NULL, _T("My Application"), WS_OVERLAPPEDWINDOW,
       rectDefault, NULL, MAKEINTRESOURCE(IDR_MAINFRAME));
```

传递给 `Create` 的第 6 个参数确定了菜单资源。`MAKEINTRESOURCE` 宏将一个整型资源 ID 转换为 `LPTSTR` 数据类型的 ID,该类型 ID 与接受字符串型资源 ID 的函数兼容。窗口出现在屏幕上时,菜单就会显示在标题栏的下面。

第 2 种方法用到 `CFrameWnd::LoadFrame` 函数。如果给定一个资源 ID,同 `Create` 一样 `LoadFrame` 就创建主窗口并挂接一个菜单。语句

```
LoadFrame(IDR_MAINFRAME, WS_OVERLAPPEDWINDOW, NULL, NULL);
```

创建了一个窗口并挂接了菜单 `IDR_MAINFRAME`。一些 MFC 程序——特别是向导生成的应用程序——使用 `LoadFrame` 而不用 `Create`。因为 `LoadFrame` 还可以加载图标和其他资源。在这个例子中不需要 `MAKEINTRESOURCE`,因为 `LoadFrame` 已包含了 `MAKEINTRESOURCE`。

而另一种加载最高级菜单和挂接菜单到窗口的方法是构造一个 `CMenu` 对象,调用 `CMenu::LoadMenu` 加载菜单资源,然后再调用 `CWnd::SetMenu`,如下所示:

```
CMenu menu;
menu.LoadMenu(IDR_MAINFRAME);
SetMenu(&menu);
menu.Detach();
```

在这个例子中,调用 `CMenu::Detach` 将菜单从 `CMenu` 对象上卸下,防止菜单在 `menu` 超出范围时过早地被清除。`CMenu` 类通过在它的析构函数中调用 `CMenu::DestroyMenu` 可以帮助防止资源泄露。一般来说,用 `LoadMenu` 加载的菜单在应用程序结束之前会被

DestroyMenu清除。然而挂接到窗口的菜单会在清除窗口时被自动清除。所以在菜单挂接到窗口之后将菜单从 CMenu 对象中卸下不会引起资源泄露,除非后来菜单未经调用 DestroyMenu 就与窗口分离。

程序只包含一个菜单时,SetMenu 方法与简单地将菜单 ID 传递给 Create 或 LoadFrame 相比没有任何优越之处,但在程序包含两个或更多个菜单时它就变得非常有用。假如您想编写一个应用程序,允许用户选择短的或长的菜单,这有一种方法。首先创建两个菜单资源——一个用于短的,另一个用于长的。在启动时,将菜单资源加载到 CMenu 数据成员 m\_menuLong 和 m\_menuShort 中。然后选择菜单类型。菜单类型由 BOOL 数据成员 m\_bShortMenu 的值表示。如果选中短菜单,则 m\_bShortMenu 的值为 TRUE,如果没有选中,则为 FALSE。下面是窗口构造函数的示例:

```
Create(NULL,_T("My Application"));
m_menuLong.LoadMenu(IDR_LONGMENU);
m_menuShort.LoadMenu(IDR_SHORTMENU);
SetMenu(m_bShortMenu ? &m_menuShort : &m_menuLong);
```

为响应用户发出的命令,下面的程序代码实现从长菜单到短菜单的转换:

```
m_bShortMenu = TRUE;
SetMenu(&m_menuShort);
DrawMenuBar();
```

下面的语句又将短菜单转换成长菜单:

```
m_bShortMenu = FALSE;
SetMenu(&m_menuLong);
DrawMenuBar();
```

CWnd::DrawMenuBar 重画菜单栏反映这种变化。除非窗口不在屏幕上,否则一定要在调用 SetMenu 之后调用 DrawMenuBar。

由于在应用程序结束时只允许在窗口上挂接一个菜单,所以问题是:用什么程序代码删除菜单呢?如果 m\_menuLong 和 m\_menuShort 是主窗口类的数据成员,在清除主窗口时它们的析构函数就会被调用,与之相关的菜单也同时被删除。因此不需要显式调用 DestroyMenu。

### 4.1.3 响应菜单命令

用户打开一个下拉菜单时,菜单挂接的窗口就会接收到一系列消息。首先传来的是 WM\_INITMENU 消息,通知窗口选中一个最高级菜单项。在显示子菜单之前,窗口收到 WM\_INITMENUPOPUP 消息。有时 Windows 程序趁这个时候更新子菜单的菜单项,例如:如果应用程序的工具条已显示,则在 View 菜单 Toolbar 项的旁边加一个复选标记;如果工具条

当前处于隐藏状态,则取消这个菜单项的复选标记。在加亮条上下移动时,窗口接收到 WM\_MENUSELECT 消息,报告加亮条在菜单中的最新位置。在 SDK 风格的程序中, WM\_MENUSELECT 消息有时用来显示状态栏中上下文敏感的菜单帮助。

所有消息中最重要的是用户选中某菜单项时发送的 WM\_COMMAND 消息。消息的参数 wParam 的低位字保留着该菜单项的命令 ID。SDK 编程人员经常使用 switch-case 逻辑引导程序执行正确的处理例程,但是 MFC 提供了更好的方法。消息映射表中的 ON\_COMMAND 语句将根据您的选择把引用特定菜单项的 WM\_COMMAND 消息链接到类成员函数或 command handler。当 ID\_FILE\_SAVE 菜单项被选中时,下面消息映射表中的输入项通知 MFC 调用 OnFileSave。

```
ON_COMMAND (ID_FILE_SAVE, OnFileSave)
```

File 菜单中的其他各项映射为:

```
ON_COMMAND (ID_FILE_NEW, OnFileNew)
ON_COMMAND (ID_FILE_OPEN, OnFileOpen)
ON_COMMAND (ID_FILE_SAVE, OnFileSave)
ON_COMMAND (ID_FILE_SAVE_AS, OnFileSaveAs)
ON_COMMAND (ID_FILE_EXIT, OnFileExit)
```

现在如果选中 File-New,则 OnFileNew 被激活;如果选中 File-Open,则 OnFileOpen 被调用。

命令处理程序没有参数也不返回值。例如,通常这样调用 OnFileExit 函数:

```
void CMainWindow::OnFileExit ()
{
    PostMessage (WM_CLOSE, 0, 0);
}
```

通过向主窗口发送 WM\_CLOSE 消息,该命令处理程序结束应用程序的运行。该消息致使 WM\_QUIT 消息出现在应用程序的消息队列中,从而最终结束运行应用程序。

可以随意命名命令处理程序。和 WM\_消息处理程序不一样,命令处理程序没有一定的命名标准。除非您想重写 MFC 的宏 ON\_WM\_PAINT 和 ON\_WM\_CREATE,否则 WM\_PAINT 和 WM\_CREATE 的消息处理程序必须命名为 OnPaint 和 OnCreate。但是您可以简单地按如下方式写出 File 菜单对应的各消息映射项:

```
ON_COMMAND (ID_FILE_NEW, CreateMeAFile)
ON_COMMAND (ID_FILE_OPEN, OpenMeAFile)
ON_COMMAND (ID_FILE_SAVE, SaveThisFile)
ON_COMMAND (ID_FILE_SAVE_AS, SaveThisFileUnderAnotherName)
ON_COMMAND (ID_FILE_EXIT, KillThisAppAndDoItNow)
```

#### 4.1.4 命令范围

有时用单个命令处理程序处理一组菜单项 ID 要比为每个 ID 提供一个独立成员函数更快些。绘图应用程序包含一个 Color 菜单,从中用户可以选择红色、绿色或蓝色。在菜单中选中一个颜色等于将成员变量 `m_nCurrentColor` 设置成 0、1 或 2,并随后改变用户在屏幕上画的图的颜色。这些菜单项的消息映射项和命令处理程序的运用方法示范如下:

```
// In CMainWindow's message map
ON_COMMAND(ID_COLOR_RED, OnColorRed)
ON_COMMAND(ID_COLOR_GREEN, OnColorGreen)
ON_COMMAND(ID_COLOR_BLUE, OnColorBlue)

.
.
.

void CMainWindow::OnColorRed()
{
    m_nCurrentColor = 0;
}

void CMainWindow::OnColorGreen()
{
    m_nCurrentColor = 1;
}

void CMainWindow::OnColorBlue()
{
    m_nCurrentColor = 2;
}
```

这样处理来自 Color 菜单的消息并不是最有效的。因为每个消息处理程序本质上做的是同一件事。如果菜单包含 10 或 20 种颜色而不是仅仅 3 种,那么这种方法的效率就更差了。

降低 Color 菜单命令处理程序冗余的一种方法是将 3 个菜单项映射为同一个 `CMainWindow` 成员函数并调用 `CWnd::GetCurrentMessage` 获取菜单项 ID,如下所示:

```
// In CMainWindow's message map
ON_COMMAND(ID_COLOR_RED, OnColor)
ON_COMMAND(ID_COLOR_GREEN, OnColor)
ON_COMMAND(ID_COLOR_BLUE, OnColor)

.
.
.

void CMainWindow::OnColor()
{
    . . .
```

```
UINT nID = (UINT) LOWORD (GetCurrentMessage ()->wParam);
m_nCurrentColor = nID - ID_COLOR_RED;
```

只要命令 ID 能构成从 ID\_COLOR\_RED 起始的连续序列,这种方法还是不错的,但是缺点在于它依赖于 wParam 的值。如果新的 Windows 版本中消息 WM\_COMMAND 的 wParam 参数的含义发生了变化(如 Windows 3.1 到 Windows 95),您可能只好修改程序代码而使它能正常运行了。而且即便将命令处理程序的个数从 3 个减少到 1 个,您始终要以每项 24 字节的代价将 3 个独立的消息映射项添加到类的消息映射表中。

更好的解决方法是使用 MFC ON\_COMMAND\_RANGE 宏。它将一组连续的命令 ID 映射为一个公用处理程序。假设 ID\_COLOR\_RED 的值是这组命令 ID 中最低的,而 ID\_COLOR\_BLUE 的值是最高的,则 ON\_COMMAND\_RANGE 允许按下面的方式重新编写 Color 菜单的程序代码:

```
// In CMainWindow's message map
ON_COMMAND_RANGE (ID_COLOR_RED, ID_COLOR_BLUE, OnColor)

.
.
.

void CMainWindow::OnColor (UINT nID)
{
    m_nCurrentColor = nID - ID_COLOR_RED;
}
```

当用户在 Color 菜单中选择了某个菜单项而使 OnColor 被调用时,nID 将包含 ID\_COLOR\_RED、ID\_COLOR\_GREEN 或 ID\_COLOR\_BLUE。无论选择的是哪个菜单项,只需一条简单的语句就为 m\_nCurrentColor 设定了适当的值。

#### 4.1.5 更新菜单中的菜单项

在许多应用程序中,必须不断更新菜单项,使它反映出应用程序的内部状态和数据。例如:从 Color 菜单中选择了—个颜色,相应的菜单项就应该加上复选标记或单选标记,指明当前选中的颜色。如果应用程序具有菜单项为 Cut、Copy 和 Paste 的 Edit 菜单,则在没有对象选中时,应用程序使 Cut 和 Copy 菜单项无效,或在剪贴板为空时使 Paste 菜单项无效。菜单不只是命令的罗列。如果安排得合理,菜单能直观地将应用程序的当前状态反馈给用户,并在某一时刻清晰地表明哪些命令是有效的(哪些是无效的)。

Windows 编程人员通常采用两种方法之一更新菜单项。第一种方法由下面的示例说明。它是在前面给出的 OnColor 函数基础上修改得到的。

```
void CMainWindow::OnColor (UINT nID)
```

```

    {
        CMenu * pMenu = GetMenu();
        pMenu->CheckMenuItem(m_nCurrentColor + ID_COLOR_RED, MF_UNCHECKED);
        pMenu->CheckMenuItem(nID, MF_CHECKED);
        m_nCurrentColor = nID - ID_COLOR_RED;
    }

```

在这个示例中,一选中某菜单项,Color 菜单就被更新。首先调用带 MF\_UNCHECKED 标志的 CMenu::CheckMenuItem,将当前被选中的菜单项恢复成未选中状态。然后调用带 MF\_CHECKED 标志的 CheckMenuItem,在刚刚选中的菜单项旁边添放一个复选标记。下次 Color 下拉菜单显示时,复选标记指明了当前颜色。

第2种方法是更新菜单的程序代码移到响应 WM\_INITMENUPOPUP 消息的 OnInitMenuPopup 处理程序中去。每次 Color 菜单拉下时,就在下拉菜单真正显示之前,这种方法都要给复选标记定位。OnInitMenuPopup 接收 3 个参数: CMenu 指针,它指向即将显示的子菜单;UINT 值,保留子菜单在最高级菜单中基于 0 的索引值;BOOL 值,当消息属于系统菜单而不是子菜单时为非零值。下面是 Color 菜单的 OnInitMenuPopup 处理程序的示例。COLOR\_MENU\_INDEX 是一个索引值,确定 Color 菜单在最高级菜单中的位置:

```

// In CMainWindow's message map
ON_WM_INITMENUPOPUP()

{
    .
    .
    .

void CMainWindow::OnInitMenuPopup(CMenu * pPopupMenu, UINT nIndex,
    BOOL bSysMenu)
{
    if (!bSysMenu && (nIndex == COLOR_MENU_INDEX)) {
        pPopupMenu->CheckMenuItem(ID_COLOR_RED, MF_UNCHECKED);
        pPopupMenu->CheckMenuItem(ID_COLOR_GREEN, MF_UNCHECKED);
        pPopupMenu->CheckMenuItem(ID_COLOR_BLUE, MF_UNCHECKED);
        pPopupMenu->CheckMenuItem(m_nCurrentColor + ID_COLOR_RED,
            MF_CHECKED);
    }
}

```

因为这种方法把处理命令的代码与更新菜单的代码分开了,所以比第一种方法适应性更好。如果应用程序中任一函数改变了绘图的颜色,则在下次显示时菜单就会自动更新。

MFC 提供了一种类似但更加方便的机制更新菜单项。通过消息映射表中的 ON\_UPDATE\_COMMAND\_UI 宏,可以给单个菜单项分配选中的成员函数作为更新处理程序。当用户点中一个下拉菜单时,MFC 捕获随之发生的 WM\_INITMENUPOPUP 消息,并调用菜单中所

有菜单项的更新处理程序。每个更新处理程序获得一个指向 CCmdUI 对象的指针,并利用该对象的成员函数修改菜单项。并且由于 CCmdUI 类不是专用于某一类型用户界面 (UI) 元素的,所以适用于菜单项的更新处理程序还可用于工具条和其他 UI 对象。用这种方法将 UI 的更新抽象出来,可以简化程序逻辑并使应用程序不受操作系统的影响。

利用更新处理程序重新编写 Color 菜单的程序代码如下:

```
// In CMainWindow's message map
ON_COMMAND_RANGE (ID_COLOR_RED, ID_COLOR_BLUE, OnColor)
ON_UPDATE_COMMAND_UI (ID_COLOR_RED, OnUpdateCclorRed)
ON_UPDATE_COMMAND_UI (ID_COLOR_GREEN, OnUpdateColorGreen)
ON_UPDATE_COMMAND_UI (ID_COLOR_BLUE, OnUpdateColorBlue)

.
.
.

void CMainWindow::OnColor (UINT nID)
{
    m_nCurrentColor = nID - ID_COLOR_RED;
}

void CMainWindow::OnUpdateColorRed (CCmdUI * pCmdUI)
{
    pCmdUI->SetCheck (m_nCurrentColor == 0);
}

void CMainWindow::OnUpdateColorGreen (CCmdUI * pCmdUI)
{
    pCmdUI->SetCheck (m_nCurrentColor == 1);
}

void CMainWindow::OnUpdateColorBlue (CCmdUI * pCmdUI)
{
    pCmdUI->SetCheck (m_nCurrentColor == 2);
}
```

同 ON\_COMMAND 连接菜单项和命令处理程序一样,ON\_UPDATE\_COMMAND\_UI 将菜单项连接到更新处理程序上。此时在 Color 菜单中选择一种颜色会激活 CMainWindow::OnColor,并且在 Color 菜单显示之前,各菜单项的更新处理程序都会被调用。通过调用 CCmdUI::SetCheck 选中或取消选中相应的菜单项,下列处理程序完成更新任务。如果参数为非零值,调用 SetCheck 就会在相应的菜单项旁边添加一个复选标记;如果参数为零,则无复选标记显示。

SetCheck 是 CCmdUI 的方法之一,可以用来更新菜单项。表 4-1 列出了全部方法,并解释了各个函数对菜单项的作用。



表 4-1 CCmdUI 的方法

函数	说 明
CCmdUI::Enable	使菜单项有效或无效
CCmdUI::SetCheck	选中或取消选中菜单项
CCmdUI::SetRadio	给菜单项添加或删除单选标记
CCmdUI::SetText	改变菜单项的正文

SetRadio 和 SetCheck 使用方法一样,只是 SetRadio 添加或删除单选而不是复选标记。SetRadio是在 Windows API 中直接找不到等价函数的 MFC 函数之一;为了允许菜单项能被单选而不是复选,主结构在后台做了一些工作。理想情况下,bullet 用来指示一组排它菜单项中被选中的那一个,而复选标记用来指示功能是打开的还是关闭的。(实际上,两种情况都经常用到复选标记)Enable 使菜单项有效或无效,并且 SetText 允许轻松改变菜单项的正文。

#### 4.1.6 更新范围

对于用一个更新处理程序更新成组的菜单项的情况,MFC 提供了 ON\_UPDATE\_COMMAND\_UI\_RANGE 宏。该宏与 ON\_COMMAND\_RANGE 的关系正如 ON\_UPDATE\_COMMAND\_UI 与 ON\_COMMAND 的关系。如果要了解 ON\_UPDATE\_COMMAND\_UI\_RANGE 是如何运用的,让我们先回顾 Color 菜单并假定它包含八个颜色选项,按顺序为:黑色、蓝色、绿色、青色、红色、品红、黄色和白色。相应的菜单项 ID 为 ID\_COLOR\_BLACK 到 ID\_COLOR\_WHITE。再假定我们想在当前颜色选项旁加一个单选标记。下面是最简捷的方法:

```
// In CMainWindow's message map
ON_COMMAND_RANGE (ID_COLOR_BLACK, ID_COLOR_WHITE, OnColor)
ON_UPDATE_COMMAND_UI_RANGE (ID_COLOR_BLACK, ID_COLOR_WHITE,
    OnUpdateColorUI)
.
.
.
void CMainWindow::OnColor (UINT nID)
{
    m_nCurrentColor = nID - ID_COLOR_BLACK;
}

void CMainWindow::OnUpdateColorUI (CCmdUI * pCmdUI)
{
    pCmdUI->SetRadio (pCmdUI->m_nID - ID_COLOR_BLACK ==
        m_nCurrentColor);
}
```

m\_nID 是 CCmdUI 的一个公用数据成员,它保留需要更新的菜单项的 ID。将 m\_nID 减去

ID\_COLOR\_BLACK 后的值与 m\_nCurrentColor 作比较,并把比较结果传递给 SetRadio,就可以确定只有当前颜色是加单选标记的。

MFC 的命令更新机制有多大的用处呢?本章后部分将给出一个示例程序。它使用了两个相同的 Color 菜单——一个从最高级菜单调出,另一个从右击鼠标弹出的上下文菜单调出。同一个命令和更新程序可用于两个菜单,而且无论选中哪个颜色两个菜单都会随之更新——程序一行不少。很难想像还有更容易的方法更新菜单项。

### 4.1.7 键盘加速键

设计应用程序菜单时,您可以选择使用键盘加速键给任意或全部菜单项分配快捷键。同选择某个菜单项一样,加速键也引发 WM\_COMMAND 消息。给应用程序增加键盘加速键本身是很简单的。先创建一个加速键表资源——一个特殊的资源,它将菜单项 ID 和键或某些键的组合对应起来——而后通过调用一个函数将资源加载到程序中。如果应用程序的主窗口是框架窗口,Windows 和主结构会完成剩下的工作:自动捕获加速键按下时发生的信号,并发送 WM\_COMMAND 消息通知应用程序。

在 RC 文件中用 ACCELERATORS 数据块定义加速键表资源。常规格式如下:

```
ResourceID ACCELERATORS
BEGIN
.
.
.
END
```

ResourceID 是加速键表的资源 ID。BEGIN 和 END 之间的语句确定了加速键和相应的菜单项 ID。MFC AppWizard 用下面的格式生成加速键表:

```
IDR_MAINFRAME ACCELERATORS PRELOAD MOVEABLE
BEGIN
    "N",          ID_FILE_NEW,          VIRTKEY,CONTROL
    "O",          ID_FILE_OPEN,        VIRTKEY,CONTROL
    "S",          ID_FILE_SAVE,        VIRTKEY,CONTROL
    "Z",          ID_EDIT_UNDO,        VIRTKEY,CONTROL
    "X",          ID_EDIT_CUT,         VIRTKEY,CONTROL
    "C",          ID_EDIT_COPY,        VIRTKEY,CONTROL
    "V",          ID_EDIT_PASTE,       VIRTKEY,CONTROL
    VK_BACK,      ID_EDIT_UNDO,        VIRTKEY,ALT
    VK_DELETE,    ID_EDIT_CUT,         VIRTKEY,SHIFT
    VK_INSERT,    ID_EDIT_COPY,        VIRTKEY,CONTROL
    VK_INSERT,    ID_EDIT_PASTE,       VIRTKEY,SHIFT
END
```

在这个示例中, IDR\_MAINFRAME 是加速键表的资源 ID。PRELOAD and MOVEABLE 是加载选项,同 MENU 语句中等价的关键字一样,在 Win32 环境下这两个选项不起作用。表中每一行定义一个加速键。每行的第一项定义加速键,第二项表示相应的菜单项。VIRTKEY 关键字告诉资源编译程序第一项是虚拟键代码,随后的关键字——CONTROL、ALT 或 SHIFT——定义了一个可选的控制键。在这个例子中, Ctrl-N 是 File-New 的加速键, Ctrl-O 是 File-Open 的加速键等等。Edit 菜单的 Undo、Cut、Copy 和 Paste 功能每个都有两个定义好的加速键: Undo 的是 Ctrl-Z 和 Alt-Backspace, Cut 的是 Ctrl-X 和 Shift-Del, Copy 的是 Ctrl-C 和 Ctrl-Ins, Paste 的是 Ctrl-V 和 Shift-Ins。

和菜单一样,键盘加速键必须在使用之前加载和挂接到窗口。对于框架窗口, LoadAccelerTable 可以一次完成加载和挂接:

```
LoadAccelerTable (MAKEINTRESOURCE (IDR_MAINFRAME));
```

LoadFrame 做这个工作做得也很好。实际上,如果菜单资源和加速键表资源共享同一个 ID,则调用同一个函数可以同时加载菜单和加速键表:

```
LoadFrame (IDR_MAINFRAME, WS_OVERLAPPEDWINDOW, NULL, NULL);
```

如果要加速键起作用,消息循环必须包含一个 API 函数调用::TranslateAccelerator,如下所示:

```
while (GetMessage (&msg, NULL, 0, 0)) {
    if (!TranslateAccelerator (hwnd, hAccel, &msg)) {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
}
```

MFC 的 CFrameWnd 类替您处理这部分。特别地,如果发现加速键表已被加载,即如果框架窗口的 m\_hAccelTable 数据成员包含一个非 NULL 的加速键表句柄,它将重载从 CWnd 继承来的虚函数 PreTranslateMessage 并调用::TranslateAccelerator。不必感到奇怪, LoadAccelerTable 装载加速键资源并将句柄复制到 m\_hAccelTable。通过调用 LoadAccelerTable LoadFrame 也完成同样的工作。

由于非框架窗口缺少 CFrameWnd 中对加速键的支持,所以加速键在装载时一定要用不同的方式处理。假定您从 CWnd 派生出一个用户窗口并也想使用加速键。那么应该这样做:

1. 给派生类添加一个 m\_hAccelTable 数据成员(类型 HACCEL)。
2. 在应用程序生存期初期,调用 API 函数::LoadAccelerators 加载加速键表。将::LoadAccelerators 返回的句柄复制到 m\_hAccelTable。
3. 在窗口类中重载 PreTranslateMessage 并调用将句柄存储在 m\_hAccelTable 中的::TranslateAccelerator。将由::TranslateAccelerator 返回的值作为 PreTranslateMessage

的返回值,如果`::TranslateAccelerator`已将消息发送,则不需要再转换和发送消息了。

以下是程序代码形式:

```
// In CMainWindow's constructor
m_hAccelTable = ::LoadAccelerators (AfxGetInstanceHandle (),
    MAKEINTRESOURCE (IDR_ACCELERATORS));

// PreTranslateMessage override
BOOL CMainWindow::PreTranslateMessage (MSG * pMsg)
{
    if (CWnd::PreTranslateMessage (pMsg))
        return TRUE;
    return ((m_hAccelTable != NULL) &&
        ::TranslateAccelerator (m_hWnd, m_hAccelTable, pMsg));
}
```

通过适当地使用此结构, `CWnd` 类型的窗口和框架窗口一样使用加速键。注意在应用程序终止之前不需要删除通过`::LoadAccelerators` (或 `LoadAccelTable`) 加载的加速键。因为 Windows 会自动删除它们。

使用加速键给常用的菜单命令提供快捷键。这是用来人工处理按键消息的较好的方法。原因有两个。第 1 个原因是加速键简化了编程逻辑。如果不是不得不编写 `WM_KEYDOWN` 和 `WM_CHAR` 处理程序,为什么要编写呢? 第 2 个原因是如果应用程序的窗口包含几个子窗口并且其中一个子窗口具有输入焦点,那么键盘消息发送到子窗口而不是主窗口。(子窗口在第 7 章中介绍)如在第 3 章中学到的那样,键盘消息总是发送到具有输入焦点的窗口。但是在加速键按下时,Windows 保证引发的 `WM_COMMAND` 消息发送到主窗口,即使其中一个子窗口具有输入焦点。

加速键对于捕获击键消息是如此有用,以致于有时脱离菜单使用。例如:如果您希望 `Ctrl-Shift-F12` 组合键一按下您就被通知到,只要用下面的语句给这个组合键创建一个加速键即可:

```
VK_F12, ID_CTRL_SHIFT_F12, VIRTKEY, CONTROL, SHIFT
```

然后通过消息映射表中添加一项将加速键映射为一个类成员函数。

```
ON_COMMAND (ID_CTRL_SHIFT_F12, OnCtrlShiftF12)
```

此后即使没有菜单项赋得 ID `ID_CTRL_SHIFT_F12`, 按下 `Ctrl-Shift-F12` 就会激活 `OnCtrlShiftF12`。

## 4.2 SHAPES 应用程序

现在让我们用所学的知识 and 技巧建立一个包含菜单、加速键的应用程序吧。同时,应用

程序还要运用 MFC 的 UI 更新机制,保证菜单项和反映应用程序内部状态的数据成员同步。这里我们首次使用 AppWizard 生成应用程序初始状态资源代码以及用 ClassWizard 来编写消息处理程序。另外我们还用 ClassWizard 给应用程序的菜单项编写命令处理程序和更新处理程序。AppWizard 和 ClassWizard 是 MFC 代码生成器,它能减少您需要编写的代码量,从而缩短开发应用程序的时间。

这个名为 Shapes 的应用程序显示在图 4-2 中。Shapes 在框架窗口的中心位置显示了一个多边形。通过在 Shape 菜单中选择一个命令(Circle、Triangle 或 Square),或按下相应的键盘加速键(F7、F8 或 F9),您可以改变多边形的形状。

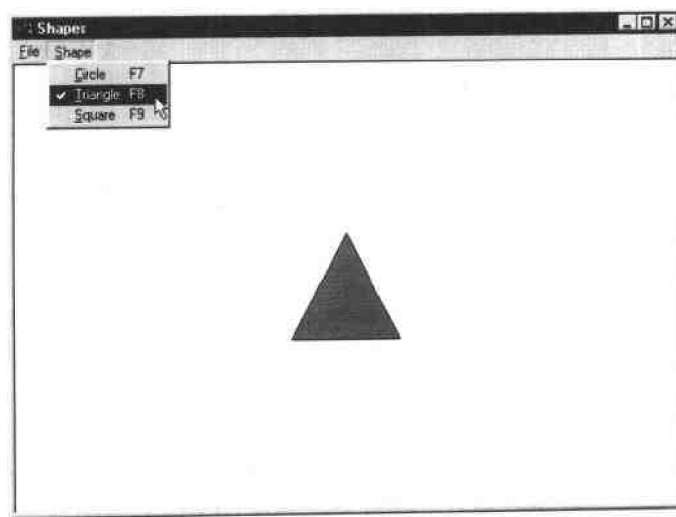


图 4-2 Shapes 窗口

程序的源代码复制在图 4-3。然而在用向导编写应用程序的时候,源代码并不能说明一切。理解源代码是如何生成的,由谁生成的同样非常重要。因此,我将一步一步讲述如何用 MFC AppWizard 创建 Shapes 的源代码。然后我们停下来看看 AppWizard 生成了些什么。

#### Shapes.h

```
// Shapes.h : main header file for the SHAPES application
//

#ifdef AFX_SHAPES_H__437C8B37_5C45_11D2_8E53_006008A82731__INCLUDED_
#define AFX_SHAPES_H__437C8B37_5C45_11D2_8E53_006008A82731__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
```

---

```

#ifndef __AFXWIN_H__
    #error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h" // main symbols
////////////////////////////////////
// CShapesApp:
// See Shapes.cpp for the implementation of this class
//

class CShapesApp : public CWinApp
{
public:
    CShapesApp();

// Overrides
    // ClassWizard generated virtual function overrides
    //||AFX_VIRTUAL(CShapesApp)
    public:
        virtual BOOL InitInstance();
    //||AFX_VIRTUAL

// Implementation
public:
    //||AFX_MSG(CShapesApp)
    afx_msg void OnAppAbout();
    //||AFX_MSG
    DECLARE_MESSAGE_MAP()
};

////////////////////////////////////
//||AFX_INSERT_LOCATION|
// Microsoft Visual C++ will insert additional declarations immediately
// before the previous line.

#endif
// !defined(AFX_SHAPES_H__437C8B37_5C45_11D2_8E53_006008A82731__INCLUDED_)

```

---

### Shapes.cpp

```

// Shapes.cpp : Defines the class behaviors for the application.
//

#include "stdafx.h"
#include "Shapes.h"

#include "MainFrm.h"

#ifdef _DEBUG

```

```

// =====
// CShapesApp
// =====
BEGIN_MESSAGE_MAP(CShapesApp, CWinApp)
    //| AFX_MSG_MAP(CShapesApp)
    ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
    //| AFX_MSG_MAP
END_MESSAGE_MAP()

// =====
// CShapesApp construction
// =====
CShapesApp::CShapesApp()
{
}

// =====
// The one and only CShapesApp object
// =====
CShapesApp theApp;

// =====
// CShapesApp initialization
// =====
BOOL CShapesApp::InitInstance()
{
    // Standard initialization

    // Change the registry key under which our settings are stored.
    SetRegistryKey(_T("Local AppWizard-Generated Applications"));

    CMainFrame* pFrame = new CMainFrame;
    m_pMainWnd = pFrame;

    // create and load the frame with its resources
    pFrame->LoadFrame(IDR_MAINFRAME,
        WS_OVERLAPPEDWINDOW|FWS_ADDTOTITLE, NULL,
        NULL);
    pFrame->ShowWindow(SW_SHOW);
    pFrame->UpdateWindow();

    return TRUE;
}
// =====

```

```

// CShapesApp message handlers
////////////////////////////////////
// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
    CAboutDlg();

// Dialog Data
    ///|AFX_DATA(CAboutDlg)
    enum { IDD = IDD_ABOUTBOX };
    ///|AFX_DATA

    // ClassWizard generated virtual function overrides
    ///|AFX_VIRTUAL(CAboutDlg)
protected:
    virtual void DoDataExchange(CDataExchange * pDX); // DDX/DDV support
    ///|AFX_VIRTUAL

// Implementation
protected:
    ///|AFX_MSG(CAboutDlg)
        // No message handlers
    ///|AFX_MSG
    DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
    ///|AFX_DATA_INIT(CAboutDlg)
    ///|AFX_DATA_INIT

}

void CAboutDlg::DoDataExchange(CDataExchange * pDX)
{
    CDialog::DoDataExchange(pDX);
    ///|AFX_DATA_MAP(CAboutDlg)
    ///|AFX_DATA_MAP

}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
    ///|AFX_MSG_MAP(CAboutDlg)
        // No message handlers
    ///|AFX_MSG_MAP
END_MESSAGE_MAP()

// App command to run the dialog
void CShapesApp::OnAppAbout()

```



\_\_\_\_\_ 1997年12月15日

```

        virtual void Dump(CDumpContext& dc) const;
    #endif
        CChildview m_wndView;

// Generated message map functions
protected:
    //||AFX_MSG(CMainFrame)
    afx_msg void OnSetFocus(CWnd * pOldWnd);
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    //||AFX_MSG
    DECLARE_MESSAGE_MAP()
};
//////////////////////////////////////
//||AFX_INSERT_LOCATION||
// Microsoft Visual C++ will insert additional declarations immediately
// before the previous line.

    #endif
// !defined(AFX_MAINFRM_H_437C833B_5C45_11D2_8E53_006008A82731__INCLUDED_)

```

### MainFrm.cpp

```

// MainFrm.cpp : implementation of the CMainFrame class
//

#include "stdafx.h"
#include "Shapes.h"
#include "MainFrm.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//////////////////////////////////////
// CMainFrame

IMPLEMENT_DYNAMIC(CMainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
    //||AFX_MSG_MAP(CMainFrame)
        ON_WM_SETFOCUS()
        ON_WM_CREATE()
    //||AFX_MSG_MAP
END_MESSAGE_MAP()

//////////////////////////////////////

```

```

// CMainFrame construction/destruction

CMainFrame::CMainFrame()
{
}

CMainFrame::~CMainFrame()
{
}

BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    if( !CFrameWnd::PreCreateWindow(cs) )
        return FALSE;
    cs.dwExStyle &= ~WS_EX_CLIENTEDGE;
    cs.lpszClass = AfxRegisterWndClass(0);
    return TRUE;
}

/////////////////////////////////////////////////////////////////
// CMainFrame diagnostics

#ifdef _DEBUG
void CMainFrame::AssertValid() const
{
    CFrameWnd::AssertValid();
}

void CMainFrame::Dump(CDumpContext& dc) const
{
    CFrameWnd::Dump(dc);
}

#endif // _DEBUG

/////////////////////////////////////////////////////////////////
// CMainFrame message handlers
void CMainFrame::OnSetFocus(CWnd* pOldWnd)
{
    // forward focus to the view window
    m_wndView.SetFocus();
}

BOOL CMainFrame::OnCmdMsg(UINT nID, int nCode, void* pExtra,
    AFX_CMDHANDLERINFO* pHandlerInfo)
{
    // let the view have first crack at the command
    if (m_wndView.OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
        return TRUE;

```

---

```

        // otherwise, do default handling
        return CFrameWnd::OnCmdMsg(nID, nCode, pExtra, pHandlerInfo);
    }

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;
    if (!m_wndView.Create(NULL, NULL, AFX_WS_DEFAULT_VIEW,
        CRect(0, 0, 0, 0), this, AFX_IDW_PANE_FIRST, NULL))
    {
        TRACE0("Failed to create view window\n");
        return -1;
    }
    return 0;
}

```

---

### ChildView.h

```

// ChildView.h : interface of the CChildView class
//
////////////////////////////////////////////////////////////////////
#ifdef AFX_CHILDVIEW_H__437C8B3D_5C45_11D2_8E53_006008A82731__INCLUDED_
#define AFX_CHILDVIEW_H__437C8B3D_5C45_11D2_8E53_006008A82731__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

// CChildView window

class CChildView : public CWnd
{
// Construction
public:
    CChildView();

// Attributes
public:

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{AFX_VIRTUAL(CChildView)
protected:

```

---

```

        virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
        //||AFX_VIRTUAL
// Implementation
public:
    virtual ~CChildView();

    // Generated message map functions
protected:
    int m_nShape;
    //||AFX_MSG(CChildView)
    afx_msg void OnPaint();
    afx_msg void OnShapeCircle();
    afx_msg void OnShapeTriangle();
    afx_msg void OnShapeSquare();
    afx_msg void OnUpdateShapeCircle(CCmdUI * pCmdUI);
    afx_msg void OnUpdateShapeTriangle(CCmdUI * pCmdUI);
    afx_msg void OnUpdateShapeSquare(CCmdUI * pCmdUI);
    //||AFX_MSG
    DECLARE_MESSAGE_MAP()
};

////////////////////////////////////

//||AFX_INSERT_LOCATION||
// Microsoft Visual C++ will insert additional declarations immediately
// before the previous line.

#ifdef _AFXDLL
// !defined(AFX_CHILDVIEW_H__437C8B3D_5C45_11D2_8E53_006008A82731__INCLUDED_)

```

---

### ChildView.cpp

```

// ChildView.cpp : implementation of the CChildView class
//

```

```

#include "stdafx.h"
#include "Shapes.h"
#include "ChildView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////////////////////
// CChildView

CChildView::CChildView()

```

```

        m_nShape = 1; // Triangle
    |

CChildView::~CChildView()
{
}

BEGIN_MESSAGE_MAP(CChildView, CWnd)
    //||AFX_MSG_MAP(CChildView)
    ON_WM_PAINT()
    ON_COMMAND(ID_SHAPE_CIRCLE, OnShapeCircle)
    ON_COMMAND(ID_SHAPE_TRIANGLE, OnShapeTriangle)
    ON_COMMAND(ID_SHAPE_SQUARE, OnShapeSquare)
    ON_UPDATE_COMMAND_UI(ID_SHAPE_CIRCLE, OnUpdateShapeCircle)
    ON_UPDATE_COMMAND_UI(ID_SHAPE_TRIANGLE, OnUpdateShapeTriangle)
    ON_UPDATE_COMMAND_UI(ID_SHAPE_SQUARE, OnUpdateShapeSquare)
    //||AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////
// CChildView message handlers

BOOL CChildView::PreCreateWindow(CREATESTRUCT& cs)
{
    if (!CWnd::PreCreateWindow(cs))
        return FALSE;

    cs.dwExStyle |= WS_EX_CLIENTEDGE;
    cs.style &= ~WS_BORDER;
    cs.lpszClass = AfxRegisterWndClass(CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS,
        ::LoadCursor(NULL, IDC_ARROW), HBRUSH(COLOR_WINDOW + 1), NULL);

    return TRUE;
}

void CChildView::OnPaint()
{
    CPoint points[3];
    CPaintDC dc(this);

    CRect rcClient;
    GetClientRect(&rcClient);
    int cx = rcClient.Width() / 2;
    int cy = rcClient.Height() / 2;
    CRect rcShape(cx - 45, cy - 45, cx + 45, cy + 45);

    CBrush brush(RGB(255, 0, 0));
    CBrush* pOldBrush = dc.SelectObject(&brush);

```

```
        switch (m_nShape) {
        case 0: // Circle
            dc.Ellipse(rcShape);
            break;

        case 1: // Triangle
            points[0].x = cx - 45;
            points[0].y = cy + 45;
            points[1].x = cx;
            points[1].y = cy - 45;
            points[2].x = cx + 45;
            points[2].y = cy + 45;
            dc.Polygon(points, 3);
            break;

        case 2: // Square
            dc.Rectangle(rcShape);
            break;
        }
        dc.SelectObject(pOldBrush);
    }

void CChildView::OnShapeCircle()
{
    m_nShape = 0;
    Invalidate();
}

void CChildView::OnShapeTriangle()
{
    m_nShape = 1;
    Invalidate();
}

void CChildView::OnShapeSquare()
{
    m_nShape = 2;
    Invalidate();
}

void CChildView::OnUpdateShapeCircle(CCmdUI * pCmdUI)
{
    pCmdUI->SetCheck(m_nShape == 0);
}

void CChildView::OnUpdateShapeTriangle(CCmdUI * pCmdUI)
{
    pCmdUI->SetCheck(m_nShape == 1);
}
```





```

# if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
# ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
# pragma code_page(1252)
# endif //_WIN32

# ifdef APSTUDIO_INVOKED
////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "# include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "# define _AFX_NO_SPLITTER_RESOURCES\r\n"
    "# define _AFX_NO_OLE_RESOURCES\r\n"
    "# define _AFX_NO_TRACKER_RESOURCES\r\n"
    "# define _AFX_NO_PROPERTY_RESOURCES\r\n"
    "\r\n"
    "# if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)\r\n"
    "# ifdef _WIN32\r\n"
    "LANGUAGE 9, 1\r\n"
    "# pragma code_page(1252)\r\n"
    "# endif //_WIN32\r\n"
    "# include ""res\Shapes.rc2""
        "// non-Microsoft Visual C++ edited resources\r\n"
    "# include ""afxres.rc"" // Standard components\r\n"
    "# endif\r\n"
    "\0"
END

# endif // APSTUDIO_INVOKED

////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon

```

```

// remains consistent on all systems.
IDR_MAINFRAME ICON DISCARDABLE "res\Shapes.ico"

////////////////////////////////////
//
// Menu
//

IDR_MAINFRAME MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "E&xit", ID_APP_EXIT
    END
    POPUP "&Shape"
    BEGIN
        MENUITEM "&Circle\tF7", ID_SHAPE_CIRCLE
        MENUITEM "&Triangle\tF8", ID_SHAPE_TRIANGLE
        MENUITEM "&Square\tF9", ID_SHAPE_SQUARE
    END
END

////////////////////////////////////
//
// Accelerator
//

IDR_MAINFRAME ACCELERATORS PRELOAD MOVEABLE PURE
BEGIN
    VK_F7, ID_SHAPE_CIRCLE, VIRTKEY, NOINVERT
    VK_F8, ID_SHAPE_TRIANGLE, VIRTKEY, NOINVERT
    VK_F9, ID_SHAPE_SQUARE, VIRTKEY, NOINVERT
END

////////////////////////////////////
//
// Dialog
//

IDD_ABOUTBOX DIALOG DISCARDABLE 0, 0, 235, 55
STYLE DS_MODALFRAME|WS_POPUP|WS_CAPTION|WS_SYSMENU
CAPTION "About Shapes"
FONT 8, "MS Sans Serif"
BEGIN
    ICON IDR_MAINFRAME, IDC_STATIC, 11, 17, 20, 20
    LTEXT "Shapes Version 1.0", IDC_STATIC, 40, 10, 119, 8, SS_NOPREFIX
    LTEXT "Copyright (c) 1998", IDC_STATIC, 40, 25, 119, 8
    DEFUSHBUTTON "OK", IDOK, 178, 7, 50, 14, WS_GROUP
END

```

```

#ifndef _MAC
////////////////////////////////////
//
// Version
//

VS_VERSION_INFO VERSIONINFO
    FILEVERSION 1,0,0,1
    PRODUCTVERSION 1,0,0,1
    FILEFLAGSMASK 0x3fL
#ifdef _DEBUG
    FILEFLAGS 0x1L
#else
    FILEFLAGS 0x0L
#endif
    FILEOS 0x4L
    FILETYPE 0x1L
    FILESUBTYPE 0x0L
BEGIN
    BLOCK "StringFileInfo"
    BEGIN
        BLOCK "040904B0"
        BEGIN
            VALUE "CompanyName", "\0"
            VALUE "FileDescription", "Shapes MFC Application\0"
            VALUE "FileVersion", "1, 0, 0, 1\0"
            VALUE "InternalName", "Shapes\0"
            VALUE "LegalCopyright", "Copyright (c) 1998\0"
            VALUE "LegalTrademarks", "\0"
            VALUE "OriginalFilename", "Shapes.EXE\0"
            VALUE "ProductName", "Shapes Application\0"
            VALUE "ProductVersion", "1, 0, 0, 1\0"
        END
    END
    BLOCK "VarFileInfo"
    BEGIN
        VALUE "Translation", 0x409, 1200
    END
END

#endif // !MAC

////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED

```

```

GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_ABOUTBOX, DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 228
        TOPMARGIN, 7
        BOTTOMMARGIN, 48
    END
END
# endif // APSTUDIO_INVOKED

//////////////////////////////////////
//
// String Table
//

STRINGTABLE PRELOAD DISCARDABLE
BEGIN
    IDR_MAINFRAME    "Shapes"
END

STRINGTABLE PRELOAD DISCARDABLE
BEGIN
    AFX_IDS_APP_TITLE    "Shapes"
    AFX_IDS_IDLEMESSAGE  "Ready"
END

STRINGTABLE DISCARDABLE
BEGIN
    ID_INDICATOR_EXT      "EXT"
    ID_INDICATOR_CAPS     "CAP"
    ID_INDICATOR_NUM      "NUM"
    ID_INDICATOR_SCRL     "SCRL"
    ID_INDICATOR_OVR      "OVR"
    ID_INDICATOR_REC      "REC"
END

STRINGTABLE DISCARDABLE
BEGIN
    ID_APP_ABOUT    "Display program information, version number and copyright\nAbout"
    ID_APP_EXIT     "Quit the application; prompts to save documents\nExit"
END

STRINGTABLE DISCARDABLE
BEGIN
    ID_NEXT_PANE    "Switch to the next window pane\nNext Pane"
    ID_PREV_PANE    "Switch back to the previous window pane\nPrevious Pane"

```

```

END

STRINGTABLE DISCARDABLE
BEGIN
    ID_WINDOW_SPLIT    "Split the active window into panes\nSplit"
END

STRINGTABLE DISCARDABLE
BEGIN
    ID_EDIT_CLEAR      "Erase the selection\nErase"
    ID_EDIT_CLEAR_ALL  "Erase everything\nErase All"
    ID_EDIT_COPY       "Copy the selection and put it on the Clipboard\nCopy"
    ID_EDIT_CUT        "Cut the selection and put it on the Clipboard\nCut"
    ID_EDIT_FIND       "Find the specified text\nFind"
    ID_EDIT_PASTE      "Insert Clipboard contents\nPaste"
    ID_EDIT_REPEAT     "Repeat the last action\nRepeat"
    ID_EDIT_REPLACE    "Replace specific text with different text\nReplace"
    ID_EDIT_SELECT_ALL "Select the entire document\nSelect All"
    ID_EDIT_UNDO       "Undo the last action\nUndo"
    ID_EDIT_REDO       "Redo the previously undone action\nRedo"
END

STRINGTABLE DISCARDABLE
BEGIN
    AFX_IDS_SCSIZE     "Change the window size"
    AFX_IDS_SCMOVE     "Change the window position"
    AFX_IDS_SCMINIMIZE "Reduce the window to an icon"
    AFX_IDS_SCMAXIMIZE "Enlarge the window to full size"
    AFX_IDS_SCNEXTWINDOW "Switch to the next document window"
    AFX_IDS_SCPREVIEWWINDOW "Switch to the previous document window"
    AFX_IDS_SCCLOSE    "Close the active window and prompts to save the documents"
END

STRINGTABLE DISCARDABLE
BEGIN
    AFX_IDS_SCRESTORE  "Restore the window to normal size"
    AFX_IDS_SCTASKLIST "Activate Task List"
END

// Generated from the TEXTINCLUDE 3 resource.
// #define AFX_NO_SPLITTER_RESOURCES
// #define AFX_NO_OLE_RESOURCES

```

```

# define _AFX_NO_TRACKER_RESOURCES
# define _AFX_NO_PROPERTY_RESOURCES

# if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
# ifdef _WIN32
LANGUAGE 9, 1
# pragma code_page(1252)
# endif //_WIN32
# include "res\Shapes.rc2" // non-Microsoft Visual C++ edited resources
# include "afxres.rc"      // Standard components
# endif

////////////////////////////////////
# endif // not APSTUDIO_INVOKED

```

图 4-3 Shapes 程序

### 4.2.1 运行 MFC AppWizard

Shapes 的源代码是向导生成代码和手工编写代码的组合。创建源代码的第 1 步是运行 MFC AppWizard。它是这样开始的：

1. 创建名为 Shapes 的 Visual C++ 新项目。选择 MFC AppWizard (Exe) 作为应用程序类型,如图 4-4 所示。这样能启动 AppWizard。在生成项目之前,AppWizard 会问一系列问题。

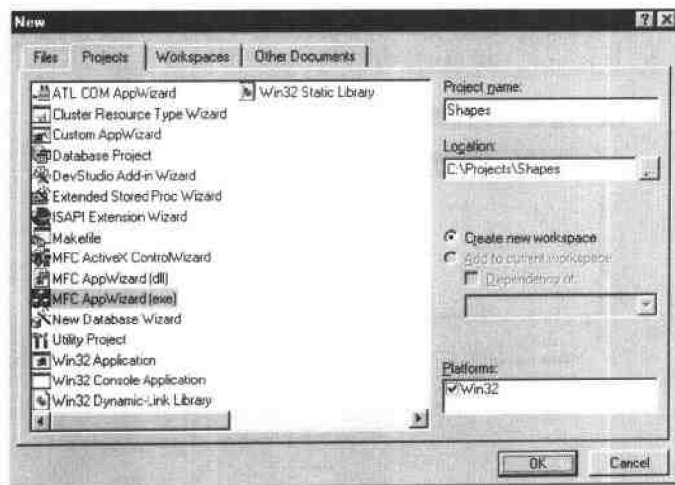


图 4-4 创建 Shapes 项目

2. 在 AppWizard 第 1 步对话框中选择 Single Document 作为应用程序类型,并取消选中标注着 Document/View Architecture Support 的方框,见图 4-5。后者是 Visual C++ 6 中的一个新选项。它禁止 AppWizard 生成 MFC 文档/视图应用程序。Single Document 的含义将在第 8 章中介绍。

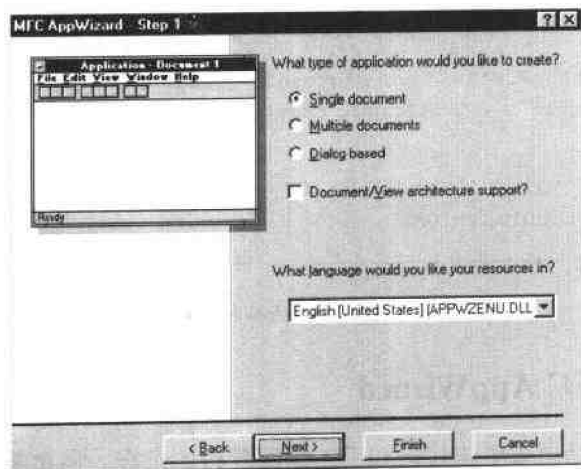


图 4-5 AppWizard 第 1 步对话框

3. 在 AppWizard 第 2 步对话框中,接受默认值。
4. 在 AppWizard 第 3 步对话框中,取消选中 ActiveX 控件框。当选中时,这个选项添加基本功能块,允许 MFC 窗口支配 ActiveX 控件——将在第 21 章讲解。
5. 在 AppWizard 第 4 步对话框中,取消选中 Docking Toolbar、Initial Status Bar 和 3D Controls 复选框,见图 4-6。接受对话框中其他地方的默认值。

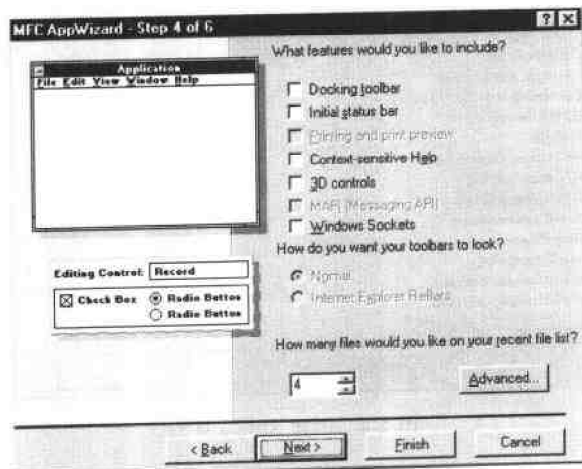


图 4-6 AppWizard 第 4 步对话框

6. 接受以后 AppWizard 对话框中的默认值,并允许 AppWizard 创建项目。不必在第 5 步和第 6 步对话框中接受默认值,在第 4 步对话框中单击 Finish 按钮即可。

单击 Finish 之后,AppWizard 会显示它要创建的程序代码的概要。单击 OK 表示确认,或单击 Cancel,然后使用 Back 和 Next 按钮在这些对话框之中向前和向后移动,并在需要的时候,做些改动。

注意,因为 Visual C++ 6.0 中有个缺点,如果您按照上面的步骤进行,CMainFrame 的最重要部分可能不会出现在您的源代码中。框架窗口的一个最重要的任务是创建视图窗口。一般应使用下面的 WM\_CREATE 处理程序创建视图窗口:

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;

    if (!m_wndView.Create(NULL, NULL, AFX_WS_DEFAULT_VIEW,
        CRect(0, 0, 0, 0), this, AFX_IDW_PANE_FIRST, NULL))
    {
        TRACE0("Failed to create view window\n");
        return -1;
    }
    return 0;
}
```

很不幸,当第 4 步对话框中的工具栏和状态栏选项被关闭时,Visual C++ 6.0 AppWizard 会错误地删除这个处理程序。因此,必须自己把它添加上。而且不要忘记在消息映射表中还要添加 ON\_WM\_CREATE 语句。

## 4.2.2 分析 AppWizard 的输出

那么 AppWizard 到底做了些什么?首先,它创建了一个包含 MFC 应用程序要求的所有创建设置的新项目。其次,它从 MFC 基本类派生出几个类并将它们放到项目中。再次,它创建了应用程序用的一组资源并将它们也放到项目中。一种熟悉 AppWizard 输出的好方法是查看它生成的文件。注意,这个输出会随您在 AppWizard 对话框中的不同选择发生很大的变化。下面几部分精简地介绍了 Shapes 应用程序中由 AppWizard 生成的各源代码文件,以及其中包含的重要编程原理。

### StdAfx.h 和 StdAfx.cpp

通过利用 Visual C++ 中称为 precompiled headers(预编译头文件)的功能,AppWizard 生成的项目加速了应用程序的构建。如果由 AppWizard 实现建造设置,则 StdAfx.h 包含的所有头文件被预先编译成文件 projectname.pch 和 StdAfx.obj。因为编译一次后,它们就不必重新



编译了。AppWizard将 StdAfx.h 包含在它生成的 CPP 文件中,并且在 StdAfx.h 中,它将内核 MFC 头文件比如 Afxwin.h 等都包含进来。您可以生成自己的 #include 语句,包含其他 MFC 头文件、C 运行期间头文件和其他类型的静态头文件。不能包含那些在应用程序开发过程中会发生变化的头文件,否则就失去预先编译头文件的意义了。

介绍预先编译的头文件时,有趣的现象是:Visual C++ 在声明包含 StdAfx.h 的语句之前,完全忽略那些出现在资源代码中的语句。这意味着这样的代码一样能编译:

```
kjasdfj;oai4efj
#include "Stdafx.h"
```

这一点为什么很重要?因为许多 MFC 编程人员都为这样的代码感到烦恼:

```
#include <math.h>
#include "Stdafx.h"
```

为避免这样的错误,需把 Math.h 的 #include 语句放在 StdAfx.h 的 #include 语句之后(或者更好的方法是把它放在 StdAfx.h 文件中)。

### Resource.h 和 Shapes.rc

在 AppWizard 生成的资源代码文件中有一个 RC 文件,它包含所有应用程序资源的定义;还有一个头文件(Resource.h),它包含命令 ID 和其他 RC 文件用到的符号的定义。查看 RC 文件,您会发现其中有一个菜单模板和加速键表。不需要人工编辑这些资源,Visual C++ 的资源编辑器允许您通过界面直观地编辑菜单、加速键、图标和其他资源,然后按照您作的变动调整 RC 文件。如果想先进入菜单编辑器,则单击 Visual C++ 工作空间窗口中的 ResourceView 标签,然后再双击菜单资源 IDR\_MAINFRAME。这样就可以打开菜单编辑器中的菜单。在这里修改工作就同在对话框中进行鼠标定位、单击和键入信息那样简单。您也可以直接编辑 RC 文件,但是如果这样做,一定要使用 Open 对话框中的 Open As Text 选项按照普通文本文件方式打开。

### Shapes.h 和 Shapes.cpp

您知道每个 MFC 应用程序都包含一个代表应用程序本身的 CWinApp 派生类的公用实例。AppWizard 派生了一个名为 CshapesApp 应用程序类并将其资源代码放在 Shapes.h 和 Shapes.cpp 中。它还通过在 Shapes.cpp 中写入下面的语句声明了该类的一个公用实例:

```
CShapesApp theApp;
```

CShapesApp::InitInstance 看起来与第 1、2 和 3 章中的 InitInstance 函数稍有不同。它通过将名为 CMainFrame 的类实例化并针对生成的对象调用 LoadFrame 创建一个框架窗口。

```
CMainFrame* pFrame = new CMainFrame;
```

```

m_pMainWnd = pFrame;
.
.
.
pFrame->LoadFrame(IDR_MAINFRAME,
    WS_OVERLAPPEDWINDOW|FWS_ADDTOTITLE, NULL,
    NULL);

```

CMainFrame 是 AppWizard 生成的另一个类,代表应用程序的顶层窗口。像前几章介绍的 CMainWnd 类,CMainFrame 的基本类是 CFrameWnd。和 CMainWnd 不一样,CMainFrame 的构造函数并不调用 Create。因此,这里是由 InitInstance 创建框架窗口对象以及相应的框架窗口。

AppWizard 的 CShapesApp 类还包括一个名为 OnAppAbout 的命令处理程序:

```

// In the message map
ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
.
.
.
void CShapesApp::OnAppAbout()
{
    CAboutDlg aboutDlg;
    aboutDlg.DoModal();
}

```

在学习第 8 章的对话框之后,这段程序代码会更容易理解些。它的目的是显示一个 About 对话框——一个包含程序有关信息的对话框,比如作者和版权。CAboutDlg 是代表这个 About 框的类,它的源代码也在 Shapes.h 和 Shapes.cpp 文件中。为支持这个功能,AppWizard 将一个 About Shapes 命令 (ID = ID\_APP\_ABOUT) 插放在应用程序的 Help 菜单中。选中这个 Help-About Shapes 命令,则执行 CShapesApp::OnAppAbout 并显示一个简单的 About 框。

### ChildView.h 和 ChildView.cpp

由 AppWizard 生成的 Shapes 应用程序和前面手工编写的应用程序的最大区别在于增加了一个名为 CChildView 的新类。CChildView 是 CWnd 派生类,代表应用程序的“view”——一个尺寸调整到与应用程序框架窗口中用户区相符合并正好放在用户区顶部的特殊窗口。框架窗口的用户区实际上就是这个视图窗口。这意味着我们是在 CChildView 而不是在 CMainFrame 中编写 WM\_PAINT 处理程序。实际上,AppWizard 在 CChildView 中已包含了一个没有具体内容的 OnPaint 函数。同时它还重载了 CWnd::PreCreateWindow,并在重载过程中,包含了登录视图专用的 WNDCLASS 和向视图的窗口样式添加 WS\_EX\_CLIENTEDGE 的程序代码。WS\_EX\_CLIENTEDGE 通过使视图显得凹进框架窗口使窗口获得了三维立体效果。

MFC 的 `CFrameWnd` 类包含这样的程序代码,它随着框架窗口尺寸的调整自动调整视图窗口的尺寸,从而保证视图窗口粘贴在框架窗口上。

实际上,AppWizard 已经创建了一个同文档/视图应用程序一样使用视图的应用程序。问题是,为什么这样做?这是设计应用程序现成的好办法吗?AppWizard 加入视图的主要原因是基于视图的体系结构简化了使用工具条和其他 UI 对象框架窗口的管理任务。如果想在包含工具条框架窗口的用户区画图,则每次调用 `GetClientRect` 时必须从框架窗口的用户区矩形中减去工具条矩形得到一块“有效”用户区。在基于视图的应用程序中就不用这么麻烦。因为只要框架窗口的尺寸发生了变化或工具条、状态栏的尺寸、位置和可见性发生了变化,MFC 都会调整视图使它与框架窗口的有效用户区一致。在视图类中调用 `GetClientRect`,您会得到可用空间的准确尺寸。

基于视图的应用程序体系结构会对您编写的代码有如下影响:

- `WM_PAINT` 消息要在视图中处理,而不能在框架窗口中处理。
- 用户区鼠标消息要在视图中处理,而不能在框架窗口中处理。因为视图完全隐藏了框架窗口的用户区,框架窗口不会接收到任何用户区鼠标消息。
- 键盘消息处理程序也要在视图中处理,而不能在框架窗口中处理。

现在编写基于视图的应用程序是在为编写第9章开始介绍的完整的文档/视图 MFC 应用程序做准备。

### MainFrm.h 和 MainFrm.cpp

这两个文件包含 AppWizard 生成的框架窗口类 `CMainFrame` 的源代码。这个框架窗口类在以下几个方面与我们目前使用的 `CMainWindow` 类相区别:

- 它重载 `CFrameWnd::PreCreateWindow`。因为 `CMainFrame` 不能在它的类构造函数中创建窗口,所以重载 `PreCreateWindow` 是它实现控制窗口样式和窗口其他特性的唯一途径。
- 它重载两个 `CObject` 函数: `AssertValid` 和 `Dump`,用于诊断测试。
- 它包含一个代表视图窗口的 `CChildView` 成员变量 `m_wndView`。
- 它包含 `WM_SETFOCUS` 处理程序。只要框架窗口接收到输入焦点,该处理程序就将输入焦点转移到视图。这个转移很重要。因为鼠标和键盘输入的主要资源是视图,而不是框架窗口。如果输入焦点给了框架窗口,而没有转移给视图,则视图类中的键盘消息处理程序就不起作用。
- 它重载 `CFrameWnd::OnCmdMsg` 并将命令传递给视图和应用程序对象(间接地)。该应用程序对象使用文档/视图应用程序中用到的命令分配体系结构的简化形式。其实用性在于应用程序菜单项的命令处理程序和更新处理程序可以放在框架窗口

类、视图类或应用程序类中。如果没有 OnCmdMsg,这两种处理程序就被限制在框架窗口中了。命令传递将在第9章和第11章介绍。

### 4.2.3 AppWizard 之外的工作

AppWizard 生成一个标准的应用程序骨架。一旦 AppWizard 按步骤运行完,为了使应用程序与众不同,您还得编写其他代码。没必要全用手工编写这些程序代码,可以用 ClassWizard 完成一些基本任务,比如添加消息处理程序、命令处理程序和更新处理程序。实际上,ClassWizard 编写通用代码,而您就可以集中力量编写应用程序特有的程序代码了。根据这种思路,下而是复制图 4-3 所示源代码所需的步骤:

1. 在 Visual C++ Shapes 项目打开的前提下,在 CChildView 类中添加一个受保护的整型成员变量-m\_nShape。可通过手工编程或界面而操作添加这个成员变量。如果要通过界面操作添加,则单击工作空间窗口中的 ClassView 标签,右击 ClassView 中的 CChildView,在上下文菜单中选择 Add Member Variable,并填写图 4-7 所示的 Add Member Variable 对话框。

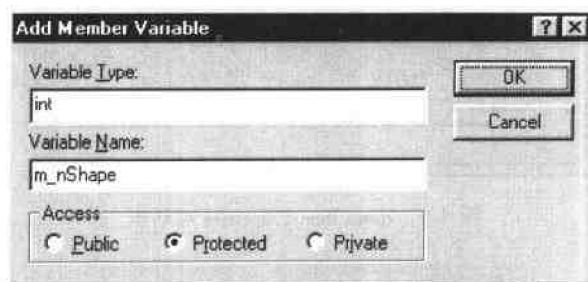


图 4-7 Add Member Variable 对话框

2. 通过在 CChildView 的构造函数中添加以下语句将 m\_nShape 的初始值设为 1:

```
m_nShape = 1; // Triangle
```

m\_nShape 的值可以是 0、1 或 2,表示画在视图中的形状分别为圆、三角或正方形。把 m\_nShape 的初始值设为 1 等于将三角设置成默认值。

3. 修改视图的 OnPaint 处理程序,使它成为图 4-3 所示的那样。AppWizard 已经在视图类中添加了一个空的 OnPaint 处理程序,您要做的就是删除它。
4. 单击工作空间窗口底部的 ResourceView 标签,查看 AppWizard 创建的资源表列。双击 IDR\_MAINFRAME 菜单资源,打开并进行编辑。删除 Edit 和 Help 菜单,然后在 File 菜单右边添加一个 Shape 菜单,并将表 4-2 中的 3 项添加到 Shape 菜单中:

表 4-2 添 加 项

菜单项正文	命令 ID
&Circle\tF7	ID_SHAPE_CIRCLE
&Triangle\tF8	ID_SHAPE_TRIANGLE
&Square\tF9	ID_SHAPE_SQUARE

如果要从菜单中删除某项,则单击选中该项,而后按下 Delete 键。如果要增加一项,则双击出现在菜单中的空矩形,然后在 Menu Item Properties 对话框中敲入菜单项正文和命令 ID。(参见图 4-8)顶层菜单项不需要命令 ID,因此对于顶层菜单 ID 框是无效的。对于其他菜单项,您可敲入命令 ID 或由 Visual C++ 选择一个。如果您清除了 Menu Item Properties 对话框并且 ID 框是空的,Visual C++ 会生成一个形式为 ID\_top\_item 的命令 ID,其中 top 是顶层菜单项的名称,item 是给菜单项指定的正文。不管命令 ID 是怎样生成的,Visual C++ 在 Resource.h 文件中添加一个 #define 语句,赋给 ID 一个数值。完成的 Shape 菜单如图 4-9 所示。

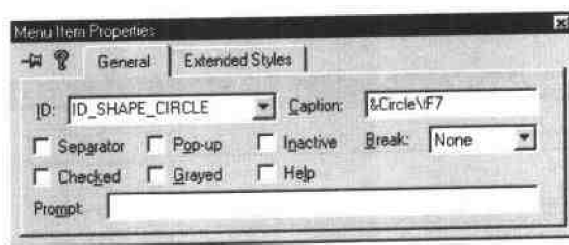


图 4-8 Menu Item Properties 对话框

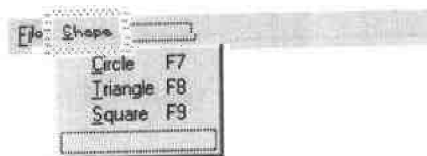


图 4-9 Shape 菜单

5. 给 Circle、Triangle 和 Sqare 命令在视图类中添加命令处理程序。以下是完整的代码:

```
// In CChildView's message map
ON_COMMAND(ID_SHAPE_CIRCLE, OnShapeCircle)
ON_COMMAND(ID_SHAPE_TRIANGLE, OnShapeTriangle)
ON_COMMAND(ID_SHAPE_SQUARE, OnShapeSquare)
.
.
.
void CChildView::OnShapeCircle()
{
```

```

        m_nShape = 0;
        Invalidate();
    }

void CChildView::OnShapeTriangle()
{
    m_nShape = 1;
    Invalidate();
}

void CChildView::OnShapeSquare()
{
    m_nShape = 2;
    Invalidate();
}

```

可以通过手工编程或让 ClassWizard 替您添加这些命令处理程序。如果用 ClassWizard 给 Circle 命令添加命令处理程序,则单击工作空间窗口底部的 ClassView 标签,右击 ClassView 中的 CChildView,并在上下文菜单中选择 Add Windows Message Handler 显示 New Windows Message And Event Handlers 对话框。(参见图 4-10)在 Class Or Object To Handle 列表框中找到 ID\_SHAPE\_CIRCLE 并单击它。然后双击 New Windows Messages/Events 列表框中的 COMMAND。当 ClassWizard 向您要函数名时,接受默认值-OnShapeCircle。COMMAND 就会移到 Existing Message/Event Handlers 列表框,指明 ID\_SHAPE\_CIRCLE 菜单项的命令处理程序现在有了。最后单击 Edit Existing 按钮进入空的命令处理程序并在函数体中添加语句:

```

m_nShape = 0;
Invalidate();

```

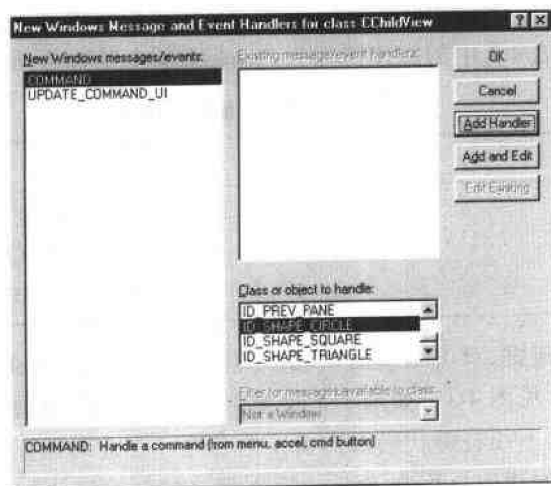


图 4-10 New Windows Message and Event Handlers 对话框

如果要编写 Triangle 和 Square 命令的命令处理程序,则重复这个过程,但要在它们的函数体中将 m\_nShape 分别设为 1 和 2。

6. 给 Circle、Triangle 和 Sqare 命令在视图类中添加更新处理程序。以下是完整的代码:

```
ON_UPDATE_COMMAND_UI(ID_SHAPE_CIRCLE, OnUpdateShapeCircle)
ON_UPDATE_COMMAND_UI(ID_SHAPE_TRIANGLE, OnUpdateShapeTriangle)
ON_UPDATE_COMMAND_UI(ID_SHAPE_SQUARE, OnUpdateShapeSquare)
.
.
.
void CChildView::OnUpdateShapeCircle(CCmdUI * pCmdUI)
{
    pCmdUI->SetCheck(m_nShape == 0);
}

void CChildView::OnUpdateShapeTriangle(CCmdUI * pCmdUI)
{
    pCmdUI->SetCheck(m_nShape == 1);
}

void CChildView::OnUpdateShapeSquare(CCmdUI * pCmdUI)
{
    pCmdUI->SetCheck(m_nShape == 2);
}
```

又一次,您可以通过手工编程或用 ClassWizard 添加这些处理程序。如果要用 ClassWizard 编写更新处理程序,则按照编写命令处理程序的步骤进行。但是要双击 New Windows Messages/Events 列表框中的 UPDATE\_COMMAND\_UI 而不是 COMMAND。

7. 单击工作空间窗口中的 ResourceView 标签,打开要编辑的加速键资源 IDR\_MAIN\_FRAME。添加表 4-3 中的加速键作为 Shape 菜单中各菜单项的快捷键。

表 4-3 添加的加速键

快捷键	命令 ID
F7	ID_SHAPE_CIRCLE
F8	ID_SHAPE_TRIANGLE
F9	ID_SHAPE_SQUARE

如果要添加加速键,双击编辑窗口底部的空矩形,并在 Accel Properties 对话框中定义加速键。(参见图 4-11)如果不记得虚拟键代码,可以单击 Next Key Typed 按钮并按下快捷键,而不用将键代码敲入 Key 组合框。在此对话框中,可以删除其他加速键(AppWizard 创建的),因为 Shapes 用不着这些加速键。如果要删除加速键,只需单击选中它并按下 Delete 键。

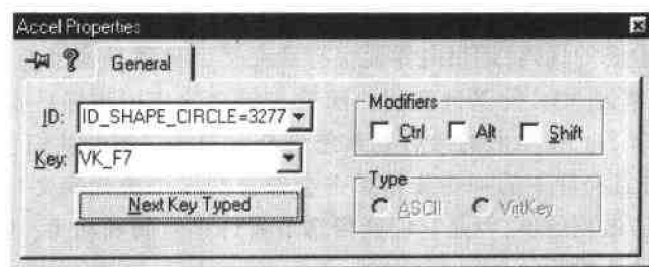


图 4-11 Accel Properties 对话框

8. 如果 CMainFrame 没有包含 186 页“注意”中介绍的 OnCreate 处理程序,现在把它添加进来。不必手工编程添加这个消息处理程序,您可以用 ClassWizard 做这件事。怎么做呢? 右击 ClassView 窗口中的 CMainFrame,选中 Add Windows Message Handler,双击 WM\_CREATE,并单击 Edit Existing。这时会出现一个空的消息处理程序体,等待敲入完成的代码。ClassWizard 已经完成了其他所有工作,包括将 ON\_WM\_CREATE 项添加到消息映射表。

到这,您已顺利建立了图 4-2 中描述的 Shapes 应用程序。这是一个简单的应用程序,它的 OnPaint 处理程序检测成员变量(m\_nShape)的值,并画一个圆、三角或正方形。Shape 菜单中各菜单项的命令处理程序将 m\_nShape 设置成 0、1 或 2,并通过调用 CWnd::Invalidate 实现重新绘制。更新处理程序在当前选中的形状旁边放一个复选标记。所有绘图和菜单命令处理在视图类中完成。该视图类在一定程度上代理框架窗口用户区的工作。借助于已添加的加速键,功能键 F7、F8 和 F9 为 Circle、Triangle 和 Square 命令提供了快捷方式。有了这个基础,您应该可以给任意应用程序添加菜单项并给它们编写命令和更新处理程序。

关于 Shapes 有一点很有意思,并值得深思。File-Exit 命令关闭应用程序,但是在程序源代码中您找不到 File-Exit 的命令处理程序。秘密是 CWinApp 消息映射表中的下面这个语句。消息映射表在 MFC 源代码文件 Appcore.cpp 中:

```
ON_COMMAND(ID_APP_EXIT, OnAppExit)
```

记住消息映射表同函数和数据成员一样都要传递给通过继承得到的派生类。即使 CShapesApp 消息映射表中没有这一项,由于 CShapesApp 是 CWinApp 派生的,所以它还是隐含地存在着。因为 AppWizard 给 Exit 命令分配了 ID ID\_APP\_EXIT,选中该命令就会激活 OnAppExit。OnAppExit 是通过继承得到的,它向应用程序主窗口发送 WM\_CLOSE 消息。它的源代码在 Appui.cpp 中。

#### 4.2.4 过程小结

使用 AppWizard 和 ClassWizard 建立应用程序和手工编制应用程序完全不同。首先要认



识到向导不能做您自己做不到的事,作为代码生成工具,它们只能在一定程度上提高编程效率。如果明白向导生成的代码,运用向导就很有意义了。这就是本书前三章没有使用向导的原因-为了帮助您掌握 MFC 的基础知识。随着您建立的应用程序日益复杂时,向导生成的代码也越来越复杂了。在本书的最后几章您就会明白了。那时我们使用 MFC 建立启用 COM 的应用程序,并且向导中的一些按钮单击会牵扯到多个源代码文件,而不是一、两个。向导从不会做您无法做到的事,但是它们能帮您节省许多时间和精力,否则对每个 Windows 应用程序您都必须从头做起。

## 4.3 菜单魔术

本章的前半部分覆盖了您需要掌握的关于菜单的 80% 的内容。然而,有时还要做一些较复杂的、特别的工作。本章后半部分将介绍一些“特别的工作”。

- 创建和修改浮动标签上菜单的技巧
- 系统菜单和定制系统菜单的方法
- 不显示正文而显示图形的菜单(自制菜单)
- 层叠菜单
- 上下文菜单

在本章结束时,我们将修改 Shapes 应用程序,添加一个自制菜单和右击上下文菜单。

### 4.3.1 通过手工编程创建菜单

从应用程序的 EXE 文件加载菜单资源并不是创建菜单的唯一途径。用 MFC 的 CMenu 类和它的成员函数编程同样也能达到这个目的。因为基本的菜单支持不需要 CMenu,所以我们还未对它做深入的研究。如果要创建浮动标签上的菜单,CMenu 很方便获取,或在运行时生成的信息中,或在要修改现有菜单时(下一部分将介绍到)。在这种情况下,CMenu 会非常有用。

您可以用 CMenu::CreateMenu、CMenu::CreatePopupMenu 和 CMenu::AppendMenu 编程创建菜单。用 CreateMenu 创建菜单,用 CreatePopupMenu 创建子菜单,并用 AppendMenu 将子菜单挂接到顶层菜单,最终形成一个顶层菜单和它的子菜单。下列程序创建的菜单与 Shapes 应用程序中的相同,并把它挂接到框架窗口。唯一的区别在于:这个菜单是编程实现的,而不是资源编辑器创建的。

```
CMenu menuMain;
menuMain.CreateMenu();

CMenu menuPopup;
menuPopup.CreatePopupMenu();
```

... ..

```

menuPopup.AppendMenu(MF_STRING, ID_FILE_EXIT, "E&xit");
menuMain.AppendMenu(MF_POPUP, (UINT) menuPopup.Detach(), "&File");

menuPopup.CreatePopupMenu();
menuPopup.AppendMenu(MF_STRING, ID_SHAPE_CIRCLE, "&Circle\tF7");
menuPopup.AppendMenu(MF_STRING, ID_SHAPE_TRIANGLE, "&Triangle\tF8");
menuPopup.AppendMenu(MF_STRING, ID_SHAPE_SQUARE, "&Square\tF9");
menuMain.AppendMenu(MF_POPUP, (UINT) menuPopup.Detach(), "&Shape");

SetMenu(&menuMain);
menuMain.Detach();

```

前面两个语句创建了一个称为 menuMain 的 CMenu 对象,它代表一个空的顶层菜单。下一段语句创建了 File 菜单,并把它挂接到顶层菜单。传递给 AppendMenu 的 MF\_POPUP 参数通知 Windows 它的第 2 个参数是一个菜单句柄,而不是菜单项 ID,Detach 将菜单与 menuPopup 对象分离并检索菜单句柄。第 3 个语句段创建了 Shape 菜单并把它挂接到顶层菜单。最后,调用 SetMenu 将刚建成的菜单挂接到框架窗口,而 Detach 将顶层菜单和 menuMain 分离以便在该函数结束时顶层菜单不至于消除。如果调用 SetMenu 时窗口可见,则应调用 DrawMenuBar 将菜单画在屏幕上。

### 4.3.2 通过手工编程修改菜单

除了动态创建菜单,可以修改现有菜单。用表 4-4 中的 CMenu 成员函数可添加、修改和删除菜单项。

表 4-4 CMenu 成员函数

函数	说 明
AppendMenu	在菜单尾部添加一个菜单项
InsertMenu	在菜单给定位置插入一个菜单项
ModifyMenu	改变菜单项的命令 ID、正文或其他特性
DeleteMenu	删除菜单项和相关的子菜单
RemoveMenu	删除菜单项

RemoveMenu 和 DeleteMenu 的区别在于:如果被删除的菜单项有子菜单,则 DeleteMenu 将菜单项和子菜单同时删除,而 RemoveMenu 则只删除菜单项但把子菜单留在内存中。通常使用 DeleteMenu,但如果想保留子菜单以备后用,则应使用 RemoveMenu。

在添加、改变或删除菜单项之前,您需要 CMenu 指针指向菜单。MFC 的 CWnd::GetMenu 函数返回一个 CMenu 指针指向窗口顶层菜单,或在窗口没有顶层菜单时返回 NULL。假设您想在运行中删除 Shapes 应用程序中的 Shape 菜单,应该这样做:

```
CMenu * pMenu = GetMenu();
```

```
pMenu->DeleteMenu(1, MF_BYPOSITION);
```

传递给 DeleteMenu 的 1 是 Shape 菜单的 0 级索引值。File 菜单占据位置 0, Shape 菜单占据 1。MF\_BYPOSITION 告诉 DeleteMenu 第 1 个参数是位置索引而不是菜单项 ID。在这种情况下, 由于 Shape 是没有菜单项 ID 的子菜单, 所以只能通过位置确定菜单项。

DeleteMenu 和其他 CMenu 函数作用于子菜单项时, 还需要指向主菜单或子菜单的指针。CMenu::GetSubMenu 返回子菜单的指针。下面的程序段用 GetMenu 获得指向主菜单的指针, 用 GetSubMenu 获得指向 Shape 菜单的指针。然后删除 Square 和 Circle 命令。

```
CMenu * pMenu = GetMenu()->GetSubMenu(1);
pMenu->DeleteMenu(2, MF_BYPOSITION);    // Delete Square
pMenu->DeleteMenu(ID_SHAPE_CIRCLE, MF_BYCOMMAND); // Delete Circle
```

首次调用 DeleteMenu 时, 根据在菜单中的位置确定该菜单项; 第 2 次调用 DeleteMenu 时, 根据命令 ID 确定菜单项。MF\_BYPOSITION 和 MF\_BYCOMMAND 标记告诉 Windows 当前使用何种方法。如果没有指定, 则默认值为 MF\_BYCOMMAND。传递给 GetSubMenu 的唯一参数是子菜单 0 级索引。因为 Circle 是用 ID 而不是用位置确定的, 所以也可以通过指向主菜单的指针调用 DeleteMenu 来删除它:

```
CMenu * pMenu = GetMenu();
pMenu->DeleteMenu(ID_SHAPE_CIRCLE, MF_BYCOMMAND);
```

如果菜单项是由 ID 确定的, 您可以通过指向菜单项所在的菜单的指针或指向任一顶层菜单的指针访问该菜单项。不要试图用 MF\_BYPOSITION 借助 GetMenu 返回的指针删除子菜单项, 否则可能会错误地删除子菜单。

如果要改变现有菜单项的特性, 调用 CMenu::ModifyMenu。如果 pMenu 指向 Shape 菜单, 语句

```
pMenu->ModifyMenu(ID_SHAPE_TRIANGLE, MF_STRING|MF_BYCOMMAND,
    ID_SHAPE_TRIANGLE, "&Three-Sided Polygon");
pMenu->ModifyMenu(2, MF_STRING|MF_BYPOSITION,
    ID_SHAPE_SQUARE, "&Four-Sided Polygon");
```

修改 Triangle 和 Square 命令, 使它们分别输出“三角形”和“四边形”。传递给 ModifyMenu 函数的第 3 个参数是新的菜单项命令 ID, 如果您不想改变它, 就保持最初的内容不变。如果被改变的菜单项是一个子菜单而不是普通的菜单项, 则这个参数保留的是菜单句柄而不是菜单项 ID。如果给定指向子菜单的 Cmenu 指针, 则总能在对象的 m\_hMenu 数据成员中获取菜单句柄。

### 4.3.3 系统菜单

正如调用 CWnd::GetMenu 获取指向顶层菜单的 CMenu 指针, 窗口还可以调用 CWnd::

GetSystemMenu 获取指向系统菜单的指针。大部分应用程序都满足于由 Windows 操纵系统菜单,但是偶尔也需要做一些特别的事情,比如在系统菜单中添加一个自定义菜单项,或改变现有菜单项的行为。

假设您要在应用程序的系统菜单中添加一个 About MyApp 菜单项。About 命令正常情况下都放在 Help 菜单中,但是或许您的应用程序没有 Help 菜单。还有可能,您的应用程序只是一个小实用程序,根本没有菜单,这时在系统菜单中添加 About MyApp 要比为了一个命令加载整个菜单轻松得多。

首先要获取指向系统菜单的指针,如下所示:

```
CMenu * pSystemMenu = GetSystemMenu (FALSE);
```

FALSE 参数通知 GetSystemMenu 编程者需要一个指针,指向可以修改的系统菜单副本。(TRUE 把系统菜单重置为默认状态。)

第二步是把“About MyApp”添加到系统菜单。

```
pSystemMenu->AppendMenu (MF_SEPARATOR);
pSystemMenu->AppendMenu (MF_STRING, ID_SYSMENU_ABOUT,
    _T("&About MyApp"));
```

第一次调用 AppendMenu 在系统菜单中添加一个菜单项分隔符,以便区分您的菜单项和其他菜单项;第二次调用 AppendMenu 添加“About MyApp”,其 ID 为 ID\_SYSMENU\_ABOUT。最好将这段代码放在主窗口的 OnCreate 处理程序中。注意往系统菜单中添加的菜单项必须赋有 ID,它们是 16 的倍数(16、32、48 等等)。Windows 保留系统菜单命令 ID 的低四位自己用,所以使用其中的任何一位,您都可能得到意外的结果。

到目前为止,新菜单项会出现在系统菜单上,但不能做任何事情。用户选中系统菜单中的某项时,窗口接收到 WM\_SYSCOMMAND 消息,并且 wParam 等于菜单项 ID。下面的 OnSysCommand 处理程序检查菜单项 ID 并在 ID 等于 ID\_SYSMENU\_ABOUT 时显示一个 About 框:

```
// In CMainWindow's message map
ON_WM_SYSCOMMAND ()

.
.
.

void CMainWindow::OnSysCommand (UINT nID, LPARAM lParam)
{
    if ((nID & 0xFFFF) == ID_SYSMENU_ABOUT) {
        // Display the About box.
        return;
    }
    CFrameWnd::OnSysCommand (nID, lParam);
}
```