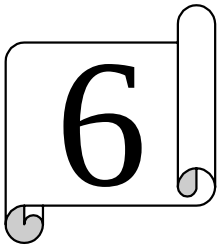


经典的 GDI 函数、字体和位图

您已经看到过一些 GDI(图形设备接口，Graphics Device Interface)元素了。任何时候当程序需要直接在屏幕或打印机上绘图时，都需要调用 GDI 或 GDI+函数。我们将在本章介绍经典的 GDI 函数，而在第 33 章讨论.NET 的时候再介绍 GDI+函数。

GDI 提供了一些用于绘制点、线、矩形、多边形、椭圆、位图以及文本的函数。利用这些函数画圆和正方形很直观，但对文本的编程就不那么容易了。本章将致力于帮助您在 Visual C++ .NET 环境下更加有效地使用 GDI，同时您还将学习如何针对显示器和打印机使用字体和位图。

6.1 设备环境类

在第 3 章和第 5 章，我们传递给视图类的 OnDraw 成员函数的参数是一个指向设备环境的指针。OnDraw 函数首先选择一个刷子，然后再画一个椭圆。Microsoft Windows 的设备环境是关键的 GDI 元素，它代表了一个物理设备。每一个 C++设备环境对象都有与之相对应的 Windows 设备环境，并且通过一个类型为 HDC 的 32 位句柄来标识。

Microsoft 基本类(MFC)库提供了一些设备环境类，其中基类 CDC 包含了绘图所需要的所有成员函数(其中一些是虚函数)，并且除了古怪的 CMetaFileDC 类之外，所有其他的派生类均只有构造函数和析构函数有所不同。如果您(或应用程序框架)构造了一个派生的设备环境类对象，就可以将 CDC 指针传给诸如 OnDraw 之类的函数。对于显示器来说，常用的派生类有 CClientDC 和 CWindowDC，而对于其他设备(如打印机或内存缓冲区)来说，则可以构造一个基类 CDC 对象。

CDC 类的“ 虚拟性 ”是应用程序框架的一个非常重要的特性。在第 17 章将会看到，我们很容易就能够编写既适用于打印机又适用于显示器的代码。OnDraw 函数中的语句

```
pDC->TextOut(0,0,"Hello");
```

既可以将文本送往显示器，也可以送往打印机，还可以送往打印预览窗口，而所有这些都依赖于

CView::OnDraw 函数中 pDC 参数所指向的对象的类。

对于显示器和打印机设备环境对象来说，应用程序框架会直接将句柄附在对象上；而对于其他设备环境(比如在后面章节中将会见到的内存设备环境)来说，为了将对象与句柄相联系，在构造完对象之后，还必须调用一个成员函数。

6.1.1　显示设备环境类 CClientDC 和 CWindowDC

请回想一下，通常窗口的客户区域并不包括边框、标题栏和菜单栏。因此，如果您创建了一个 CClientDC 对象，则该设备环境的映射区域也仅限于客户区域，即您不可能在客户区域之外绘图。点(0，0)通常指的是客户区域的左上角。在以后的章节中我们将会讲到，MFC 库中的 CView 对象总是和某个位于独立的框架窗口内部的“子窗口”相对应，通常该框架窗口还伴随有工具栏、状态栏和滚动条。因此，视图的客户区域并不包括这些伴随的窗口。如果窗口含有工具栏的话，那么点(0，0)指的就是工具栏下最左边的点。

如果创建的是 CWindowDC 对象，那么点(0，0)指的就是整个窗口的非客户区域的左上角。因此，利用这个全窗口的设备环境，您就可以在窗口的任何地方绘图，包括窗口边框、标题栏等等。但是我们要记住，视图窗口没有非客户区域，因此 CWindowDC 更适合于框架窗口，而不是视图窗口。

6.1.2　构造和析构 CDC 对象

当创建了一个 CDC 对象后，很重要的一点就是要在合适的时候将它销毁掉。Windows 限制了可用的设备环境的数目，而且，如果 Windows 设备环境对象没有被成功地释放的话，则程序在退出之前有小部分内存就丢失了。在大多数情况下，我们会在消息控制函数内部创建设备环境对象，如在 OnLButtonDown 中。要保证设备环境对象能够被适时地销毁掉(并且释放掉底层相应的 Windows 设备环境)，最简单的办法就是像下面的程序中所做的那样，在栈中构造对象：

```
void CMyView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CRect rect;

    CClientDC dc(this);  // constructs dc on the stack
    dc.GetClipBox(rect); // retrieves the clipping rectangle
} // dc automatically released
```

我们不难注意到，在以上的例子中，CClientDC 的构造函数以一个窗口指针作为参数，而它的析构函数在函数返回时被自动调用。我们也可以像下面的代码所示的那样，通过调用 CWnd 的 GetDC 成员函数来获得设备环境的指针，但此时必须注意要通过调用 ReleaseDC 函数来释放设备环境。

```
void CMyView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CRect rect;

    CDC* pDC = GetDC();     // a pointer to an internal dc
    pDC->GetClipBox(rect); // retrieves the clipping rectangle
    ReleaseDC(pDC);        // Don't forget this
}
```

警告　千万不能销毁被作为参数以指针形式传递给 OnDraw 函数的 CDC 对象，应用程序框架会控制它的销毁动作。

6.1.3　设备环境的状态

我们已经知道，绘图时离不开设备环境。当利用 CDC 对象在屏幕上(或硬拷贝到打印机上)绘图时，比如说画一个椭圆时，我们所见到的绘出的图形都要依赖于设备环境的当前“状态”，这种状态包括：

- 被选中的 GDI 绘图对象，如笔、刷子和字体。
- 决定绘制时的缩放尺寸的映射模式。(关于映射模式我们已经在第 5 章中介绍过了。)
- 其他各种细节，如文本的对齐参数、多边形的填充模式等。

例如，在前面我们已经见到过，如果事先我们已经选中了一个灰色刷子，那么我们在绘制椭圆时，其内部就会被灰色填充。当创建设备环境对象时，它通常具有一些默认特性，如绘制边界时使用一支黑色的笔，而所有其他特性都是通过 CDC 类的成员函数来设定的。我们可以通过重载 SelectObject 函数将 GDI 对象选进设备环境中，例如，我们可以随时将一种笔、一种刷子或一种字体等选进设备环境中。

6.1.4　CPaintDC 类

只有当您改写视图类的 OnPaint 函数时，才会需要用到 CPaintDC 类。尽管默认的 OnPaint 函数会使用已经设置好的设备环境来调用 OnDraw，但有时您还是需要编写一些针对显示器的特殊绘图代码。CPaintDC 类是非常特殊的，它的构造函数和析构函数所完成的工作都是针对显示用的。然而，一旦您获得了一个 CDC 指针，就可以将它当作任何设备环境指针来使用。

在下面的例子中，OnPaint 函数创建了一个 CPaintDC 对象：

```
void CMyView::OnPaint()
{
    CPaintDC dc(this);
    OnPrepareDC(&dc); // explained later
    dc.TextOut(0, 0, "for the display, not the printer");
```

```
    OnDraw(&dc);      // stuff that's common to display and printer
}
```

> 致 Win32 程序员
>
> CPaintDC 类的构造函数会自动调用 BeginPaint，而它的析构函数则会自动调用 EndPaint，因此，如果设备环境是在栈中被创建的话，那么 EndPaint 的调用完全是自动的。

6.2　GDI 对象

Windows GDI 对象的类型是通过 MFC 库中的类来表示的，而 CGdiObject 正是所有 GDI 对象类的抽象基类，即 Windows GDI 对象是通过 CGdiObject 派生类的 C++对象来表示的。

下面列出了 GDI 派生类的列表：

- CBitmap　位图是一种位矩阵，每一个显示像素都对应于其中的一个或多个位。您可以利用位图来表示图像，也可以利用它来创建刷子。
- CBrush　刷子定义了一种位图形式的像素组合，利用它可对区域内部填充颜色。
- CFont　字体是一种具有某种风格和尺寸的所有字符的完整集合，它常常被当作资源保存在磁盘中，其中有一些还是设备相关的。
- CPalette　调色板是一种颜色映射接口，它允许一个应用程序在不干扰其他应用程序的前提下，可以充分利用输出设备的颜色描绘能力。
- CPen　笔是一种用来画线及绘制有形边框的工具，您可以指定它的颜色及厚度，并且可以指定它画实线、点线或虚线。
- CRgn　区域是由多边形、椭圆或二者组合形成的一种范围，可以利用它来进行填充、裁剪以及鼠标点中测试。

6.2.1　GDI 对象的构造与析构

您永远也不会创建 CGdiObject 类的对象，相反，您需要创建它的派生类的对象。有些 GDI 派生类(如 CPen 和 CBrush)的构造函数允许您提供足够的信息，从而一步即可完成对象的创建任务；而另外一些类(如 CFont 和 CRgn)的对象的创建则需要二步才能完成，对于这些类，您首先要调用它的默认构造函数来构造 C++对象，然后还要进一步调用相应的创建函数，如 CreateFont 或 CreatePolygonRgn 等。

CGdiObject 类有一个虚析构函数，它的派生类的析构函数需要将与 C++对象相关联的 Windows GDI 对象删除掉。如果构造了一个 CGdiObject 派生类的对象，则在退出程序之前，必须先将它删除

掉。为了将 GDI 对象删除掉，首先必须将它从设备环境中分离出来。我们将在下一小节中给出具体的例子。

> 致 Win32 程序员
>
> 在 Win32 中，GDI 占用的是进程的内存，当程序终止时会自动释放。不过，一个没有释放的 GDI 位图对象可能会占用很多内存。

6.2.2　跟踪 GDI 对象

好，现在您已经明白了必须将 GDI 对象删除掉，并且在此之前必须先将它从设备环境中分离出来。那么如何将它分离出来呢?事实上，CDC 类的 SelectObject 成员函数族在将 GDI 对象选进设备环境的同时，还返回了指向前一次被选对象的指针(在这一过程中，它实际上已被从设备环境中分离了出来)。麻烦的是，在未将新的对象选中之前，还不能将旧的对象分离。一种简单的跟踪对象的办法是，在选进自己的 GDI 对象的同时，将原来的 GDI 对象也保存起来，当任务完成后，再将原来的对象恢复，这样就可以将自己的 GDI 对象删除掉了。下面给出一个例子：

```
void CMyView::OnDraw(CDC* pDC)
{
    CPen newPen(PS_DASHDOTDOT, 2, (COLORREF) 0);  // black pen,
                                                  //  2 pixels wide
    CPen* pOldPen = pDC->SelectObject(&newPen);

    pDC->MoveTo(10, 10);
    pDC->Lineto(110, 10);
    pDC->SelectObject(pOldPen);                   // newPen is deselected
} // newPen automatically destroyed on exit
```

当设备环境对象被销毁时，它的所有 GDI 对象都将处于落选状态。因此，如果您知道某个设备环境在它的所有被选中的对象被销毁之前先被销毁，则此时就没有必要先使它的所有被选中的对象落选。例如，如果定义了一支笔作为视图类的一个数据成员(当对视图进行初始化时就对该笔进行了初始化)，则此时就没有必要在 OnDraw 内部使该笔落选，因为由该视图基类的 OnPaint 函数控制的设备环境将首先被销毁。

6.2.3　库存的 GDI 对象

Windows 包含了一些库存的 GDI 对象可供您使用。由于它们是 Windows 系统的一部分，因此您用不着删除它们。(Windows 对任何企图删除库存 GDI 对象的动作都将不予理会。)MFC 库函数 CDC::SelectStockObject 可以把一个库存对象选进设备环境中，并返回原先被选中的对象的指针，同

时使该对象被分离出来。所以，如果您希望将自己的非库存 GDI 对象分离出来，进而再将其销毁，则可以随时利用这些库存对象。在上一例子中，您也可以用库存对象来代替“旧”的对象。

```
void CMyView::OnDraw(CDC* pDC)
{
    CPen newPen(PS_DASHDOTDOT, 2, (COLORREF) 0);  // black pen,
                                                  //  2 pixels wide

    pDC->SelectObject(&newPen);
    pDC->MoveTo(10, 10);
    pDC->Lineto(110, 10);
    pDC->SelectStockObject(BLACK_PEN);           // newPen is deselected
} // newPen destroyed on exit
```

在《MFC Library Reference》的 CDC∷SelectStockObject 条目中列出了所有有关笔、刷子、字体和调色板的库存对象。

6.2.4　GDI 选择的有效期

对于显示设备环境来说，在每一个消息控制函数的入口处，设备环境都是未被初始化的。当函数退出之后，在该函数内部所进行的任何 GDI 选择(或者是映射模式，或者是其他设备环境设置)都不再有效。因此，每次都必须从头开始设置设备环境。尽管 CView 类的虚成员函数 OnPrepareDC 对于设置映射模式来说是非常有用的，但您仍然需要小心谨慎地管理好自己的 GDI 对象。

对于其他设备环境来说，比如对于打印机和内存缓冲区的设备环境，您所做的设置可以持续得长久一些。对于这些有效期较长的设备环境来说，事情就变得有些复杂了，这种复杂性是由 SelectObject 函数所返回的 GDI C++对象指针的临时性所导致的。(在程序的空闲循环处理阶段，应用程序框架将销毁临时的 C++对象。有时在控制函数返回时也会进行这一销毁工作。详见联机文档中的 MFC Technical Note #3。)因此，您不能简单地将这一指针保存在类的数据成员中，而应该借助于 GetSafeHdc 成员函数将它转换为 Windows 的句柄(唯一能够持久存在的 GDI 标识)。下面是一个具体的例子：

```
// m_pPrintFont points to a CFont object created in CMyView's constructor
// m_hOldFont is a CMyView data member of type HFONT, initialized to 0

void CMyView::SwitchToCourier(CDC* pDC)
{
    m_pPrintFont->CreateFont(30, 10, 0, 0, 400, FALSE, FALSE,
                             0, ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                             CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                             DEFAULT_PITCH | FF_MODERN,
                             "Courier New"); // TrueType
    CFont* pOldFont = pDC->SelectObject(m_pPrintFont);
```

```
    // m_hOldFont is the CGdiObject public data member that stores
    //  the handle
    m_hOldFont = (HFONT) pOldFont->GetSafeHandle();
}

void CMyView:SwitchToOriginalFont(CDC* pDC)
{
    // FromHandle is a static member function that returns an
    //  object pointer
     if (m_hOldFont) {
        pDC->SelectObject(CFont::FromHandle(m_hOldFont));
    }
}

// m_pPrintFont is deleted in the CMyView destructor
```

警告　当删除由 SelectObject 返回的指针所指向的对象时，一定要当心。如果该对象是由您自己申请的，那么可以将它删除；如果该指针是临时的，则由于它所指向的对象也许是最初被选进设备环境的，所以您不能删除此 C++对象。

6.3　字　　体

老式的字符方式(character-mode)的应用程序只能在屏幕上显示单调的系统字体，而 Windows 则提供了多种多样的、与设备无关的、各种尺寸的字体。只要有效地利用 Windows 的字体，则您用不着在编程上下多大的功夫，就可以极大地改善各种应用程序的显示效果。而在 Windows 3.1 中首次引入的 TrueType 字体要比以前的设备相关字体更易于编程。在本章的后面您将会看到几个使用各种字体的例子。

6.3.1　字体是 GDI 对象

字体是 Windows GDI 必要的组成部分，也就是说，在行为上它和其他的 GDI 对象是完全一样的。它们可以被按比例缩放，也可以被裁剪，还可以像笔或刷子那样被选进设备环境。所有关于落选及删除的 GDI 规则都适用于字体。

6.3.2　选择字体

您可以在两种字体类型之间进行选择，一种是与设备无关的 TrueType 字体，另一种则是和设备

相关的字体，如 Windows 用于显示的 System 字体和 LaserJet LinePrinter 字体。或者，您也可以指定字体的类别和尺寸，而让 Windows 为您选择合适的字体。如果您让 Windows 来选择字体的话，它将尽可能地选择 TrueType 字体。由于 MFC 库提供了和当前所选择的打印机相关联的字体选择对话框，因此您没有必要去关心打印机字体的选择。您可以让用户为打印机选择具体的字体和尺寸，而您自己所要做的就是尽可能精确地在屏幕上进行显示。

6.3.3　打印字体

对于大量使用文本的应用程序来说，也许应该以“磅”(point)为单位来指定打印字体的尺寸(1 磅=1/72 英寸)。这是为什么呢？因为在大多数情况下，系统内含的打印机字体的尺寸都是以磅为单位来进行定义的，如 LaserJet LinePrinter 字体的尺寸定义为 8.5 磅。当然，对于 TrueType 字体，您可以将其尺寸指定为任何磅值。如果您以磅作为尺寸单位的话，那么就应该选择适合于以磅作为单位的映射模式，那就是 MM_TWIPS 映射模式。尺寸为 8.5 磅的字体相当于 8.5×20=170 个 twip，也就是说此时字符的高度为 170 个 twip。

6.3.4　显示字体

如果您不关心文本显示和打印机输出之间的匹配问题，那么您就可以有更多的灵活性。您既可以选择任意尺寸的 Windows TrueType 字体，也可以选择固定尺寸的系统字体(库存对象)。对于 TrueType 字体来说，它和当前所使用的映射模式关系不大，您可以简单地指定它的字体高度，而不必去考虑它相当于多少磅。

要使打印输出能和屏幕显示相匹配，这里面还存在一些问题，但 TrueType 字体使得这些问题比以前更容易解决。然而，即使使用 TrueType 字体来进行打印，您也无法精确地使屏幕显示和打印输出相匹配。为什么呢？这是因为字符最终是以像素(或点)为单位在屏幕上进行显示的，字符串的宽度则相当于它所包含的所有字符像素宽度的总和，有时为了使字母紧排或许还做了一些调整，而字符的像素宽度取决于字体、映射模式以及输出设备的精度。只有在打印和显示的映射模式都被置为 MM_TEXT 的情况下(此时 1 个像素或一个点(dot) = 1 个逻辑单位)，显示和打印才能够得到精确的匹配。如果您使用 CDC 的 GetTextExtent 函数来计算折行点的话，所得出的屏幕上的折行点和打印输出的折行点偶尔也会有所不同。


说明　在本书后面的第 15 章，我们将会讲到 MFC 库的打印预览模式，在该模式下，屏幕折行点和打印输出的折行点是完全一致的，但在处理的过程中，打印质量却受到了一定的影响。

如果要在屏幕上匹配一种打印机字体，则使用 TrueType 字体会使这一工作变得相对较为容易一

些。Windows 会自动取与它最接近的 TrueType 字体来与之相匹配。对于 8.5 磅大小的 LinePrinter 字体来说，Windows 采用它的 Courier New 字体就可以很好地与之近似匹配。

6.3.5　显示器的逻辑英寸和物理英寸

CDC 成员函数 GetDeviceCaps 可以返回各种显示参数，这些参数对于图形编程是很重要的。表 6.1 列出的 6 个参数提供了显示尺寸的信息。这些列出的数值是 Windows 2000 和 Windows XP 环境中 640×480 分辨率下的典型显示配置参数。

表 6.1　逻辑英寸和物理英寸

索引值	描　述	值
HORZSIZE	物理宽度(mm)	320
VERTSIZE	物理高度(mm)	240
HORZRES	像素宽度	640
VERTRES	高度(光栅行数)	480
LOGPIXELSX	每逻辑英寸的水平像素数	96
LOGPIXELSY	每逻辑英寸的垂直像素数	96

索引 HORZSIZE 和 VERTSIZE 表示显示器的物理尺寸。(这些值并不一定正确，因为 Windows 并不知道连接到显示适配器的显示器的真正尺寸。) 您也可以用 HORZRES 除以 LOGPIXELSX、VERTRES 除以 LOGPIXELSY 来计算显示器的尺寸。用这种方法得到的是显示器的逻辑尺寸。用上面这些值，根据每英寸为 25.4 mm，我们就可以计算出在 Windows 2000 和 Windows XP 的 640×480 分辨率的情况下，物理显示器尺寸为 12.60×9.45 英寸，逻辑尺寸为 6.67×5.00 英寸。因此物理尺寸和逻辑尺寸并不一定相同。

在 Windows 2000 和 Windows XP 下，HORZSIZE 和 VERTSIZE 通常独立于显示器的分辨率，而且 LOGPIXELSX 和 LOGPIXELSY 总是 96。因此逻辑尺寸在各种不同的分辨率下是不一样的，但物理尺寸不变。

如果您使用固定尺寸的映射模式如 MM_HIMETRIC 或者 MM_TWIPS，则显示驱动程序将用物理尺寸做映射，所以在 Windows 2000 和 Windows XP 下，在小的监视器上的文字就非常小。但其实您并不期望这样，您当然希望字体尺寸对应于逻辑显示器尺寸而不是物理尺寸。

您可以定义一种特殊的映射模式，称为逻辑 twips，此时一个逻辑单位等于 1/1 440 逻辑英寸。这种映射模式独立于操作系统，也独立于显示器的分辨率，常用于像 Microsoft Word 这样的程序。下面是把映射模式设置成逻辑 twips 的代码：

```
pDC->SetMapMode(MM_ANISOTROPIC);
```

```
pDC->SetWindowExt(1440, 1440);
pDC->SetViewportExt(pDC->GetDeviceCaps(LOGPIXELSX),
                    -pDC->GetDeviceCaps(LOGPIXELSY));
```

说明　通过 Windows 的控制面板，您可以调整显示字体的尺寸和显示器的分辨率。如果把显示字体的尺寸从默认的 100%调整到 200%，则 HORZSIZE 将变成 160，VERTSIZE 将变成 120，每英寸点数则变成 192。在这种情况下，逻辑尺寸将除以 2，所有在逻辑 twips 映射模式下显示的文字的尺寸将增加一倍。

6.3.6　计算字符高度

通过 CDC 的 GetTextMetrics 函数，我们可以得到 5 个有关字体高度的测量参数，但其中只有 3 个有实际意义。图 6.1 显示了一些非常重要的字体测量参数。tmHeight 参数代表了字体的整个高度，其中包括字符基线以下的伸展部分(如 g、j、p、q 和 y)和上至大写字母上面的音标；tmExternalLeading 参数给出的是本行音标以上至上一行基线以下的伸展部分之间的距离。tmHeight 和 tmExternalLeading 之和就是字符的总高度，tmExternalLeading 的值可以为 0。

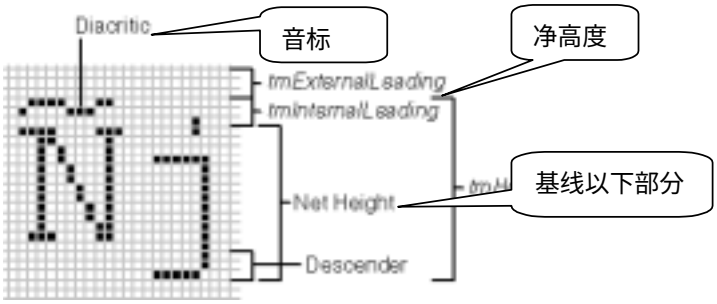


图 6.1　字体高度测量

您可能会认为 tmHeight 所表示的实际上就是以磅为单位所给出的字体的尺寸，但实际上这是完全错误的！我们还必须考虑通过 GetTextMetrics 函数所取到的另外一个参数 tmInternalLeading。以磅为单位所给出的字体的尺寸实际上是 tmHeight 与 tmInternalLeading 的差。在 MM_TWIPS 映射模式下，12 磅大小的字体的 tmHeight 值可能为 295 个逻辑单位，tmInternalLeading 的值可能为 55 个逻辑单位，那么字体的纯高度则为 240 个逻辑单位，这才相当于 12 磅的尺寸。

6.4　Ex06a 示例程序

本示例创建了一个以逻辑 twips 为映射模式的视图窗口，并且对文本字符串以 10 磅的尺寸进行了显示，所用字体为 Arial TrueType 字体。下面给出创建这个应用程序的具体步骤：

1. **运行 MFC Application Wizard 来产生一个名为 Ex06a 的项目。**从 File 菜单中选择 New 和 Project，然后选择 MFC Application。然后在 Application Type 页面上选择 Single Document，并取消 Advanced Features 页面上对 Printing And Print Preview 的选择，其他则都用默认设置。
2. 在 Class View 中选择 CEx06aView，然后通过 Properties 窗口，对 CEx06aView 类的 OnPrepareDC 函数进行改写。编辑 Ex06aView.cpp 中的该段代码如下：

```
void CEx06aView::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
    pDC->SetMapMode(MM_ANISOTROPIC);
    pDC->SetWindowExt(1440, 1440);
    pDC->SetViewportExt(pDC->GetDeviceCaps(LOGPIXELSX),
                        pDC->GetDeviceCaps(LOGPIXELSY));
}
```

3. 在视图类中加入一个私有的 ShowFont 辅助函数。在 Ex06aView.h 中加入如下的函数原型：

```
private:
    void ShowFont(CDC* pDC, int& nPos, int nPoints);
```

然后在 Ex06aView.cpp 中加入函数体：

```
void CEx06aView::ShowFont(CDC* pDC, int& nPos, int nPoints)
{
    TEXTMETRIC tm;
    CFont      fontText;
    CString    strText;
    CSize      sizeText;

    fontText.CreateFont(-nPoints * 20, 0, 0, 0, 400,
                        FALSE, FALSE, 0,
                        ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                        CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                        DEFAULT_PITCH | FF_SWISS, "Arial");
    CFont* pOldFont = (CFont*) pDC->SelectObject(&fontText);
    pDC->GetTextMetrics(&tm);
    TRACE("points = %d, tmHeight = %d, tmInternalLeading = %d,"
          " tmExternalLeading = %d\n", nPoints, tm.tmHeight,
          tm.tmInternalLeading, tm.tmExternalLeading);
    strText.Format("This is %d-point Arial", nPoints);
    sizeText = pDC->GetTextExtent(strText);
    TRACE("string width = %d, string height = %d\n", sizeText.cx,
          sizeText.cy);
    pDC->TextOut(0, nPos, strText);
    pDC->SelectObject(pOldFont);
    nPos += tm.tmHeight + tm.tmExternalLeading;
}
```

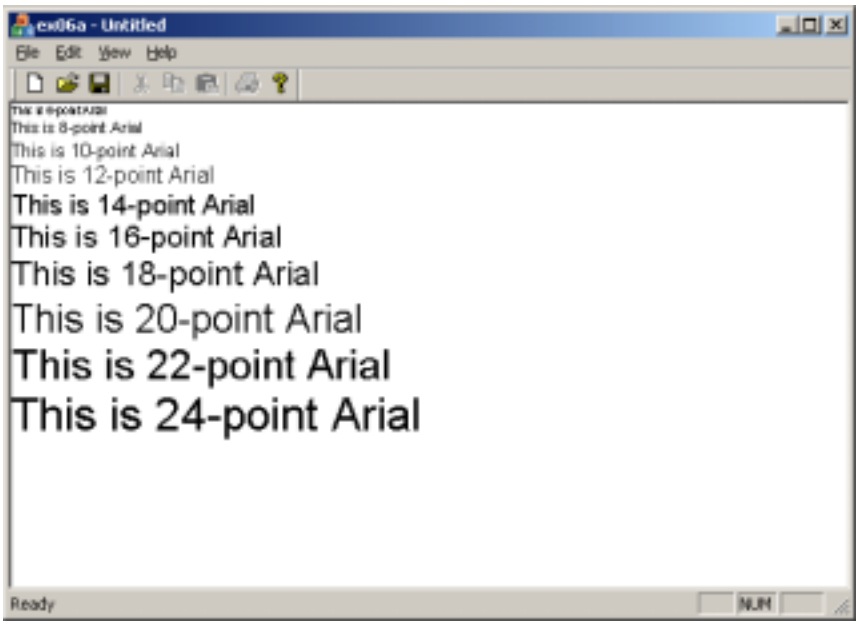
4. **编辑 Ex06aView.cpp 文件中的 OnDraw 函数。**MFC Application Wizard 已经为视图类自动生成了 OnDraw 函数的骨架，请找到该函数，并按如下的代码对它进行编辑：

```
void CEx06aView::OnDraw(CDC* pDC)
{
    int nPosition = 0;

    for (int i = 6; i <= 24; i += 2) {
        ShowFont(pDC, nPosition, i);
    }
    TRACE("LOGPIXELSX = %d, LOGPIXELSY = %d\n",
          pDC->GetDeviceCaps(LOGPIXELSX),
          pDC->GetDeviceCaps(LOGPIXELSY));
    TRACE("HORZSIZE = %d, VERTSIZE = %d\n",
          pDC->GetDeviceCaps(HORZSIZE),
          pDC->GetDeviceCaps(VERTSIZE));
    TRACE("HORZRES = %d, VERTRES = %d\n",
          pDC->GetDeviceCaps(HORZRES),
          pDC->GetDeviceCaps(VERTRES));
}
```

5. **编译并运行 Ex06a 程序。**如果您想看到 TRACE 语句的输出的话，就必须从调试器中运行本程序。为此，在 Visual C++ .NET 的 Debug 菜单中选择 Start，或者按 F5 键，或者单击 Debug 工具栏上的 Continue 按钮(这些都将强迫当前项目被编译和链接)：

如果使用的是标准 VGA 显示卡，则最终的输出结果应如下所示：
注意，显示输出的字符串的尺寸和所指定的以磅(point)为单位的字体尺寸并没有完全吻合，这种差异来源于字体引擎将尺寸从逻辑单位到像素单位所进行的转换。该程序的跟踪输出结果给出了字体的尺寸值，这里只显示一部分结果：(下面的数据依赖于显示驱动程序和视频驱动程序。)

```
points = 6, tmHeight = 150, tmInternalLeading = 30, tmExternalLeading = 4
string width = 990, string height = 150
points = 8, tmHeight = 210, tmInternalLeading = 45, tmExternalLeading = 5
string width = 1380, string height = 210
points = 10, tmHeight = 240, tmInternalLeading = 45, tmExternalLeading = 6
string width = 1770, string height = 240
points = 12, tmHeight = 270, tmInternalLeading = 30, tmExternalLeading = 8
string width = 2130, string height = 270
```



6.4.1　Ex06a 程序的组成元素

下面讨论在 Ex06a 示例程序中的重要的组成元素。

在 OnPrepareDC 函数中设置映射模式

应用程序框架在调用 OnDraw 函数之前，首先要调用 OnPrepareDC 函数，因此从逻辑上来讲，在 OnPrepareDC 函数中准备设备环境是再合适不过的了。如果其他的消息控制函数也需要设置正确的映射模式的话，那么在这些函数中也可以包含对 OnPrepareDC 的调用。

ShowFont 私有成员函数

ShowFont 函数包含了一个运行 10 次的循环。如果用 C 来编写该程序的话，您可以将 ShowFont 函数作为一个全局函数，但对于 C++，最好还是将它作为私有(private)的类成员函数，有时也将这种函数称为辅助函数(helper function)。

该函数首先创建字体，然后将该字体选进设备环境中，再在窗口中显示一个字符串，接着将该字体选出。如果在程序中选择包含调试信息，那么 ShowFont 就会显示出非常有用的字体尺寸信息，其中包括字符串的实际宽度。

调用 CFont::CreateFont

这个调用包含了许多参数，但其中最重要的是头两个参数：字体的高度和宽度。宽度值为 0 意味着所选字体的高宽比为字体的设计者所指定的值；如果该值被指定为一个非零值，即正如您在下一个例子中将看到的那样，则字体的高宽比将会发生变化。

提示　如果希望所选择的字体为指定的尺寸，那么 CreateFont 中的字体高度参数(第一个参数)就必须为负值。例如，如果对打印机使用映射模式 MM_TWIPS，那么高度参数－240 就保证了 12 磅大小的字体，即 tmHeight － tmInternalLeading = 240。而高度参数若为+240，则所创建的字体要小一些，其中 tmHeight = 240。

CreateFont 函数的最后一个参数用来指定字体名，在本例中为 Arial TrueType 字体。如果该参数为 NULL，那么指定 FF_SWISS(即不带衬线的匀称字体)将导致 Windows 自动选择最佳匹配的字体，此时所选择的字体依赖于指定的尺寸，可能为 System 字体，也可能是 Arial TrueType 字体。在这里，字体名是被优先考虑的。如果在指定 Arial 字体的同时，还指定了 FF_ROMAN(即带衬线的匀称字体)，那么最终得到的肯定是 Arial 字体。

6.5　Ex06b 示例程序

本程序和 Ex06a 非常相似，不过它显示了多种字体，并且映射模式被设置为 MM_ANISOTROPIC，同时，比例也依赖于窗口的尺寸。随着窗口尺寸的改变，字符的尺寸也会发生相应的变化。本例有效地展示了一些 TrueType 字体，并将它们和老式字体进行了对照。下面给出创建本应用程序的步骤：

1. **运行 MFC Application Wizard 来产生 Ex06b 项目。**请选择 SDI Document，并且在 Advanced Features 页面上去掉对 Printing And Print Preview 的选择。
2. 在 Class View 中选择 CEx06bView，然后用 Properties 窗口来改写 CEx06bView 类中的 OnPrepareDC 函数。编辑 Ex06bView.cpp 中的相应代码如下：

```
void CEx06bView::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
    CRect clientRect;

    GetClientRect(clientRect);
    pDC->SetMapMode(MM_ANISOTROPIC); // +y = down
    pDC->SetWindowExt(400, 450);
    pDC->SetViewportExt(clientRect.right, clientRect.bottom);
    pDC->SetViewportOrg(0, 0);
}
```

3. 向视图类中加入私有的 TraceMetrics 辅助函数。在 Ex06bView.h 中加入如下的函数原型：

```
private:
    void TraceMetrics(CDC* pDC);
```

然后在 Ex06bView.cpp 中加入函数体：

```
void CEx06bView::TraceMetrics(CDC* pDC)
{
```

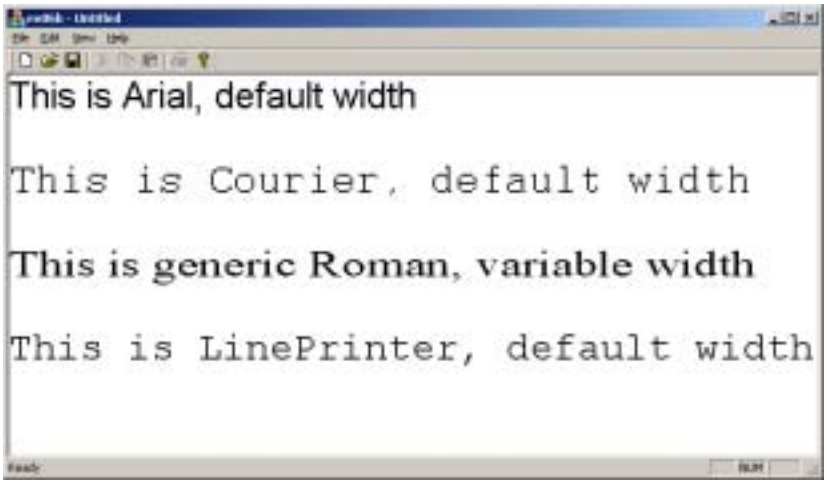
```
    TEXTMETRIC tm;
    char       szFaceName[100];

    pDC->GetTextMetrics(&tm);
    pDC->GetTextFace(99, szFaceName);
    TRACE("font = %s, tmHeight = %d, tmInternalLeading = %d,"
          " tmExternalLeading = %d\n", szFaceName, tm.tmHeight,
          tm.tmInternalLeading, tm.tmExternalLeading);
}
```

4. **编辑 Ex06bView.cpp 中的 OnDraw 函数。**MFC Application Wizard 已经为视图类自动生成了 OnDraw 函数的骨架，只需找到该函数，并按如下的代码对它进行编辑：

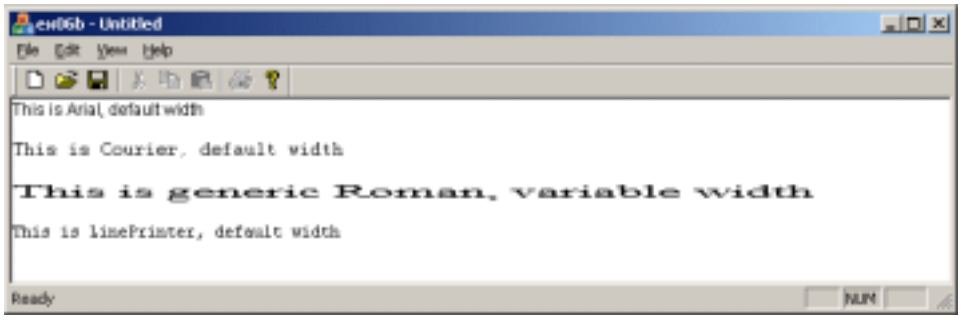
```
void CEx06bView::OnDraw(CDC* pDC)
{
    CFont fontTest1, fontTest2, fontTest3, fontTest4;

    fontTest1.CreateFont(50, 0, 0, 0, 400, FALSE, FALSE, 0,
                         ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                         CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                         DEFAULT_PITCH | FF_SWISS, "Arial");
    CFont* pOldFont = pDC->SelectObject(&fontTest1);
    TraceMetrics(pDC);
    pDC->TextOut(0, 0, "This is Arial, default width");

     fontTest2.CreateFont(50, 0, 0, 0, 400, FALSE, FALSE, 0,
                         ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                         CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                         DEFAULT_PITCH | FF_MODERN, "Courier");
                         // not TrueType
    pDC->SelectObject(&fontTest2);
    TraceMetrics(pDC);
    pDC->TextOut(0, 100, "This is Courier, default width");

    fontTest3.CreateFont(50, 10, 0, 0, 400, FALSE, FALSE, 0,
                         ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                         CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                         DEFAULT_PITCH | FF_ROMAN, NULL);
    pDC->SelectObject(&fontTest3);
    TraceMetrics(pDC);
    pDC->TextOut(0, 200, "This is generic Roman, variable width");

    fontTest4.CreateFont(50, 0, 0, 0, 400, FALSE, FALSE, 0,
                         ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                         CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                         DEFAULT_PITCH | FF_MODERN, "LinePrinter");
```

```
    pDC->SelectObject(&fontTest4);
    TraceMetrics(pDC);
    pDC->TextOut(0, 300, "This is LinePrinter, default width");
    pDC->SelectObject(pOldFont);
}
```

5. **编译并运行 Ex06b 程序。**从调试器中运行本程序，即可以看到 TRACE 输出。本程序的输出窗口如下：



请缩小窗口的尺寸，并观察字体尺寸的变化情况。将下面的窗口与前面的窗口进行比较。



如果继续不断地缩小窗口尺寸的话，就会发现，当窗口缩小到一定程度时，Courier 字体就会停止变小，同时请注意观察 Roman 字体的变化。

6.5.1　Ex06b 程序的组成元素

下面讨论在 Ex06b 示例程序中的重要的组成元素。

OnDraw 成员函数

OnDraw 函数分别用如下的 4 种字体来显示字符串：

- fontTest1　TrueType Arial 字体，采用了默认宽度。
- fontTest2　老式的 Courier 字体，采用了默认宽度。不难注意到，当尺寸较大时，字形就会出现锯齿。
- fontTest3　普通的 Roman 字体，在这里，Windows 提供的是 TrueType Times New Roman 字体，其宽度是由程序指定的。由于该宽度和水平窗口比例相关联，因此该字体总能相应地进行伸缩以适合窗口。
- fontTest4　被指定为 LinePrinter 字体，由于该字体并不是 Windows 用来供显示用的，因此字体引擎会根据所指定的 FF_MODERN 选项自动选择 TrueType Courier New 字体。

TraceMetrics 辅助函数

TraceMetrics 辅助函数调用了 CDC::GetTextMetrics 和 CDC::GetTextFace，以得到当前字体的参数，然后在 Debug 窗口中输出。

6.6　Ex06c 示例程序——再次使用 CScrollView

在第 5 章(在 Ex05c 示例程序中)您已经看到过 CScrollView 类。这里的示例程序 Ex06c 可以让用户用鼠标移动椭圆，通过 MM_IOENGLISH 映射模式来使用滚动窗口。在本例中去掉了键盘滚动功能，不过您可以将 Ex05c 示例程序中的 OnKeyDown 成员函数搬过来，这样就加入了键盘滚动功能。

这里，我们将使用一个图案刷子(一个真正的 GDI 对象)来绘制椭圆，而不再使用库存刷子。使用图案刷子有一点比较复杂：在窗口滚动过程中必须重新设置原点，否则图案的裁掉部分就会对不齐，看起来就不美观。

和 Ex05c 程序一样，本示例程序包含了一个由 CScrollView 派生出来的视图类。下面给出本应用程序的创建步骤：

1. **运行 MFC Application Wizard 来产生 Ex06c 项目。**请选择 SDI Document，并且在 Advanced Features 页面上去掉对 Printing And Print Preview 的选择。一定要将视图的基类设为 CScrollView。
2. 在文件 Ex06cView.h 中对 CEx06cView 类声明进行编辑。在类 CEx06cView 的类声明中加入如下的几行：

```
private:
    const CSize m_sizeEllipse; // logical
    CPoint m_pointTopLeft; // logical, top left of ellipse rectangle
    CSize  m_sizeOffset;  // device, from rect top left
                          //  to capture point
    BOOL   m_bCaptured;
```

3. 在 Class View 中选择 CEx06cView 类，然后通过 Properties 窗口为 CEx06cView 类加入三个消息控制函数。要加入的消息控制函数如下所示：

消　息	成员函数
WM_LBUTTONDOWN	OnLButtonDown
WM_LBUTTONUP	OnLButtonUp
WM_MOUSEMOVE	OnMouseMove

4. **编辑 CEx06cView 的消息控制函数。**Properties 窗口的代码向导已经产生了上面所列出的各函数的骨架，您只需在 Ex06cView.cpp 中找到它们，并按如下的代码对它们进行编辑：

```
void CEx06cView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // still logical
    CRect rectEllipse(m_pointTopLeft, m_sizeEllipse);
    CRgn  circle;

    CClientDC dc(this);
    OnPrepareDC(&dc);
    dc.LPtoDP(rectEllipse); // Now it's in device coordinates
    circle.CreateEllipticRgnIndirect(rectEllipse);
    if (circle.PtInRegion(point)) {
        // Capturing the mouse ensures subsequent LButtonUp message
        SetCapture();
        m_bCaptured = TRUE;
        CPoint pointTopLeft(m_pointTopLeft);
        dc.LPtoDP(&pointTopLeft);
        m_sizeOffset = point - pointTopLeft; // device coordinates
        // New mouse cursor is active while mouse is captured
        ::SetCursor(::LoadCursor(NULL, IDC_CROSS));
    }
}

void CEx06cView::OnLButtonUp(UINT nFlags, CPoint point)
{
    if (m_bCaptured) {
        ::ReleaseCapture();
        m_bCaptured = FALSE;
    }
}

void CEx06cView::OnMouseMove(UINT nFlags, CPoint point)
{
    if (m_bCaptured) {
```

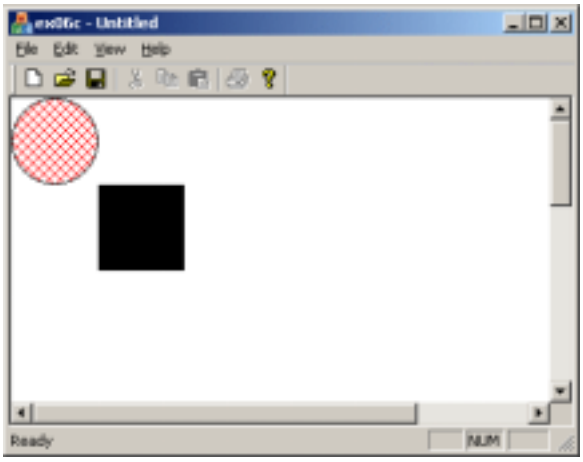
```
        CClientDC dc(this);
        OnPrepareDC(&dc);
        CRect rectOld(m_pointTopLeft, m_sizeEllipse);
        dc.LPtoDP(rectOld);
        InvalidateRect(rectOld, TRUE);
        m_pointTopLeft = point - m_sizeOffset;
        dc.DPtoLP(&m_pointTopLeft);
        CRect rectNew(m_pointTopLeft, m_sizeEllipse);
        dc.LPtoDP(rectNew);
        InvalidateRect(rectNew, TRUE);
    }
}
```

5. **对 CEx06cView 的构造函数、OnDraw 函数及 OnInitialUpdate 函数进行编辑。**MFC Application Wizard 已经产生了这些函数的骨架，您只需在 Ex06cView.cpp 中找到它们，并按如下的代码对它们进行编辑：

```
CEx06cView::CEx06cView() : m_sizeEllipse(100, -100),
                           m_pointTopLeft(0, 0),
                           m_sizeOffset(0, 0){
    m_bCaptured = FALSE;
}

void CEx06cView::OnDraw(CDC* pDC)
{
    CBrush brushHatch(HS_DIAGCROSS, RGB(255, 0, 0));
    CPoint point(0, 0);                 // logical (0, 0)

    pDC->LPtoDP(&point);                // In device coordinates,
    pDC->SetBrushOrg(point);            //  align the brush with
                                      //  the window origin
    pDC->SelectObject(&brushHatch);
    pDC->Ellipse(CRect(m_pointTopLeft, m_sizeEllipse));
    pDC->SelectStockObject(BLACK_BRUSH); // Deselect brushHatch
                                      // Test invalid rect
    pDC->Rectangle(CRect(100, -100, 200, -200));
}

void CEx06cView::OnInitialUpdate()
{
    CScrollView::OnInitialUpdate();

    CSize sizeTotal(800, 1050); // 8-by-10.5 inches
    CSize sizePage(sizeTotal.cx / 2, sizeTotal.cy / 2);
    CSize sizeLine(sizeTotal.cx / 50, sizeTotal.cy / 50);
```

```
    SetScrollSizes(MM_LOENGLISH, sizeTotal, sizePage, sizeLine);
}
```

6. **编译并运行 Ex06c 程序。**本程序允许用鼠标对椭圆进行拖动，也允许窗口被滚动。本程序的窗口看起来应该如右图所示。在移动椭圆时，注意观察黑色矩形，您应该能够看到矩形无效时的效果。



6.6.1　Ex06c 程序的组成元素

下面我们对 Ex06c 示例程序中的一些主要元素进行讨论。

数据成员 m_sizeEllipse 和 m_pointTopLeft

本程序没有将椭圆的外接矩形作为单个的 CRect 对象来保存，而是分成了两个对象，一个保存椭圆的大小(m_sizeEllipse)，另一个保存椭圆左上角的位置(m_pointTopLeft)。为了移动椭圆，程序只需重新计算 m_pointTopLeft，这样，任何四舍五入的误差都不会影响椭圆的大小。

数据成员 m_sizeOffset

当 OnMouseMove 移动椭圆时，鼠标在椭圆内的相对位置必须与用户第一次按下鼠标时的相对位置相同。对象 m_sizeOffset 用于保存鼠标按下时鼠标与椭圆矩形的左上角之间的偏移值。

数据成员 m_bCaptured

在鼠标的拖动过程中，Boolean 变量 m_bCaptured 被设置为 TRUE。

函数 SetCapture 和 ReleaseCapture

SetCapture 是 CWnd 的成员函数，它用于“捕获”鼠标，这样即使鼠标的光标移出了窗口，鼠标移动消息仍然可以发送给该窗口。这个函数的不利之处在于，椭圆可以被移出窗口，结果就好像它被“丢”了一样。但是，正如我们所期望的，所有后续的鼠标消息都发送给了窗口，包括

WM_LBUTTONUP 消息；如果不“捕获”的话，这些消息就会丢失。Win32 ReleaseCapture 函数用于关闭对鼠标的捕获。

Win32 函数 SetCursor 和 LoadCursor

MFC 库没有把一些 Win32 函数包装起来。按惯例，我们在直接调用 Win32 函数时应使用 C++作用域分辨符(::)，这样就不会把它们与 CView 的成员函数相混淆。但是，您也可以在调用类成员函数的地方，用相同的名字调用 Win32 函数，这时，::分辨符可以保证您调用的是全局的 Win32 函数。

如果 LoadCursor 的第一个参数为 NULL，那么该函数将从 Windows 使用的预定义的鼠标指针中建立一个光标资源。SetCursor 函数用于激活特定的光标资源。在鼠标被捕获期间，这个光标总是处于激活状态。

CScrollView　OnPrepareDC 成员函数

CView 类包含了一个不干任何事情的虚 OnPrepareDC 函数。在 CScrollView 类中，该函数的目的在于，根据在 OnInitialUpdate 函数中传给 SetScrollSizes 的一些参数，设置视图的映射模式和坐标原点。由于应用程序框架在调用 OnDraw 函数之前，要首先调用 OnPrepareDC 函数，因此这一点您不必担心。不过，对于其他任何消息控制函数，如果它们使用了视图的设备环境，那么就必须在其内部调用 OnPrepareDC 函数，如 OnLButtonDown 和 OnMouseMove 函数就是这样。

OnMouseMove 中的坐标变换代码

正如我们所见到的，该函数包含了一些坐标变换语句。从逻辑上来看，它主要包含了以下几个步骤：

1. 构造老的椭圆矩形，并把它从逻辑坐标转换到设备坐标。
2. 使老的椭圆矩形无效。
3. 更新椭圆矩形的左上角坐标。
4. 构造新的矩形，并将它转换为设备坐标。
5. 使新的椭圆矩形无效。

该函数调用了两次 InvalidRect。Windows 保存了两个无效矩形，并根据它们的联合及客户矩形窗口的限制来计算得到一个新的无效矩形。

OnDraw 函数

SetBrushOrg 函数调用在这里是非常必要的，这样就可以保证当视图滚动时，所有椭圆内的图案都是对齐的。此时刷子将根据参考点进行调整，该参考点为逻辑窗口的左上角，其坐标已被转换为设备坐标。对于 CDC 成员函数要求使用逻辑坐标这一规则来说，这是一个明显的例外。

6.6.2　CScrollView 的 SetScaleToFitSize 模式

CScrollView 类有一种自适应伸缩模式(stretch-to-fit mode)，在该模式下，整个可滚动区域都会完整地在屏幕上显示出来。在该模式下，映射模式被自动设置为 MM_ANISOTROPIC，并且规定 y 坐标向下递增，就和 MM_TEXT 映射模式中一样。

为了使用自适应伸缩模式，可用如下的调用来代替视图函数中的 SetScrollSizes 调用：

```
SetScaleToFitSize(sizeTotal);
```

您也可以用这个调用来响应 Shrink To Fit 菜单命令，这样，相应的显示就可以在滚动模式和自适应伸缩模式之间进行切换。

6.6.3　在滚动视图中使用逻辑 twips 映射模式

MFC CScrollView 类只允许您使用标准映射模式。在第 17 章的 Ex17a 示例程序中，我们建立了一个新的类 CLogScrollView，它可以适用逻辑 twips 模式。

6.7　位　　图

如果离开了图形图像，基于 Windows 的应用程序就会变得非常单调。有一些应用程序需要依赖于图像才能有用，但所有的应用程序都可以通过各种各样的裁剪修饰手段使得自己的界面更加美观。Windows 的位图实际上就是一些和显示像素相对应的位阵列。这听起来似乎非常简单，但要想能够利用它们来创建一些具有专业水准的基于 Windows 的应用程序，那么还必须学习许多有关位图的知识。

在本节中，您将学习如何创建设备无关的位图(DIB，device-independent bitmap)。通过使用 DIB，处理颜色和打印将更加容易，而且有些情况下也可以获得更好的性能。Win32 函数 CreateDIBSection 可以帮助我们把 DIB 的优势和 GDI 位图的各种特性结合起来。

您还将学习如何使用 MFC 的 CBitmapButton 类来把位图放到按钮上。(在使用 CBitmapButton 将位图放到按钮上时并不需要对 DIB 进行任何处理，但这种很有用的技术如果没有例子的话，是难以领会的。)

6.7.1　GDI 位图和与设备无关的位图

在这一小节中，我们来看一看 DIB。关于 DIB 最好的参考资料是 MSDN 帮助系统中的 Platform SDK

部分。Windows 有两种位图：GDI 位图和 DIB。GDI 位图已经有相当长的时间了，您在其他地方可以找到大量关于 GDI 位图的信息。

GDI 位图对象是由 MFC 库中的 CBitmap 类表示的。在 GDI 位图对象中有一个相关联的 Windows 数据结构，该数据结构由 Windows 的 GDI 模块来维护，它是设备相关的。应用程序可以得到 GDI 位图数据的一份拷贝，但其中位的安排则完全依赖于显示设备。在同一台机器中，我们可以将 GDI 位图在不同的应用程序间任意地传送，但由于 GDI 位图对设备的依赖性，因此，通过磁盘或调制解调器对它们进行传送就没有太大的意义了。

像笔和字体一样，GDI 位图只不过是另外一种 GDI 对象。首先，您必须创建一个位图，然后再把它选进设备环境中；当对它使用完毕以后，还必须将它从设备环境中选出来，然后再把它删除掉。所有这些步骤您都已经很清楚了。

不过，这里还有点问题。由于显示器的“位图”实际上就是显示器映象，而打印机设备的“位图”实际上也就是打印页本身，因此，您还不能直接把一个位图选进显示设备环境或打印机设备环境中。您必须利用 CDC::CreateCompatibleDC 函数，为位图创建一个特殊的内存设备环境，然后再利用 CDC 的 StretchBlt 或 BitBlt 成员函数，将内存设备环境中的各个位复制到真正的设备环境中去。这些位操作函数一般来说都是在视图类的 OnDraw 函数中被调用的。当然，在完成了这些工作以后，别忘了将内存设备环境清除掉。

> 致 Win32 程序员
>
> 在 Win32 中，可以把一个 GDI 位图句柄放到剪贴板上，传输到另外一个进程中，但在这种现象的背后，Windows 实际上是把设备相关位图转换成一个 DIB，并把 DIB 拷贝到共享内存里。这是从一开始就应该考虑使用 DIB 的一个很好的理由。

DIB 比 GDI 位图有很多编程优势，因为 DIB 自带颜色信息，所以调色板管理更容易。DIB 也使得打印时灰度阴影的控制更加容易。任何运行 Windows 的机器都可以处理 DIB，它通常以后缀为 BMP 的文件形式被保留在磁盘中，或者作为资源存在于程序的 EXE 或 DLL 文件中。当 Windows 启动时，其中的墙纸背景图案就是从 BMP 文件中读取的，Microsoft Paint 的基本存储格式就是 BMP 文件，Visual C++ .NET 中的各种工具栏按钮及其他图像也使用了 BMP 文件。其他的的图形交换格式也可以利用，如 TIFF、GIF 和 JPEG，但只有 DIB 格式直接被 Win32 API 支持。

6.7.2 彩色位图和单色位图

Windows 对彩色位图的处理与对刷子颜色的处理稍有不同。大多数的彩色位图只有 16 种颜色。标准的 VGA 卡有 4 个相邻的颜色位面，每个像素的颜色都是由这 4 个位面相应位的颜色组合而成的，当位图被创建时，4 位颜色值也同时被设置了。对于标准 VGA 卡，位图颜色被限定在标准 16 种颜色

里。Windows 在位图中不使用抖动颜色(dithered color)。

单色位图只有一个位面，每个像素由 1 位表示，或者为前景色(0)或者为背景色(1)。显示时的前景色由 CDC∷SetTextColor 函数来设置，而背景色则由 SetBkColor 函数来设置。无论是前景色还是背景色，您都可以通过 Windows 的 RGB 宏来指定。

6.8　DIB 和 CDib 类

在 MFC 库中有针对 GDI 位图的类(CBitmap)，但没有用于管理 DIB 的类。所以，本章将提供一个用于管理 DIB 的类。类 CDib 完全被重写了，故不同于本书以前版本(第四版以前)中提供的类。该类充分利用了 Win32 的一些特性，如内存映射文件、改进的内存管理和 DIB 项(section)，还支持了调色板。然而，在查看 CDib 类之前，您可能需要一些关于 DIB 的背景知识。

6.8.1　关于调色板编程的术语

Windows 调色板编程非常复杂，但只有当用户在 8-bpp(每像素 8 位)模式下运行程序时，才需要处理调色板——而确有许多用户由于显示卡只有 1MB 显存或更少而需要这样。

假定您在一个窗口里只显示一个 DIB。首先，您必须创建一个“逻辑调色板”，逻辑调色板是包含了 DIB 中颜色的 GDI 对象。然后您必须“施行”(realize)这个调色板，使它进入硬件的“系统调色板”，系统调色板是显示卡同时能显示的 256 种颜色的颜色表。如果一个程序是一个前台程序，则“施行”处理过程会尽量把所有的颜色拷贝到系统调色板中，但它不会改变 20 种标准 Windows 颜色。总之，它会使您的 DIB 在最大程度上接近于您所希望的那样。

如果前台程序是另一个程序，而且该程序有一个包含 236 种不同层次的绿色的森林景色 DIB，情况又会怎么样呢？您的程序仍将“施行”调色板，但这时情况发生了变化。现在系统调色板并不改变，但是，Windows 在您的逻辑调色板和系统调色板之间建立一个新的映射。例如，如果 DIB 包含一种粉红色，则 Windows 会将它映射到标准红色。如果程序没有施行调色板，则当其他程序激活时，粉红色物体可能会变成绿色的物体。

森林景色的例子有点极端，因为我们假定另一个程序占用了 236 种颜色。如果另一个程序施行的逻辑调色板只有 200 种颜色，那么 Windows 会让您的程序装入自己的 36 种颜色，包括您最希望的粉红色。

那么，怎么知道一个程序施行了自己的调色板呢？只要一个程序(包括您的程序)施行了调色板，Windows 消息 WM_PALETTECHANGED 就会被发送到您的程序的主窗口。而只要您程序中的某个窗口获得了输入焦点，那么另一消息 WM_QUERYNEWPALETTE 也会被发送到您的程序。对应这两个

消息(假设这不是您的程序产生的消息)，您的程序都应该施行调色板。然而，这些调色板消息不会被发送到视图窗口。您必须在应用程序的主框架窗口中映射这些消息，然后再通知视图。第 14 章讨论了框架窗口和视图之间的关系。

通过调用 Win32 函数 RealizePalette 可以完成“ 施行 ”过程，但首先必须调用 SelectPalette 函数把 DIB 的逻辑调色板选进设备环境。SelectPalette 有一个标记参数，通常应在 WM_PALETTECHANGED 和 WM_QUERYNEWPALETTE 消息控制函数里把该标记参数设置为 FALSE。这个标记确保当应用程序运行在前台时，相应的调色板被“ 施行 ”为前台调色板。如果使用 TRUE 参数，则可以强制 Windows 施行调色板，即使应用程序运行在后台也如此。

您必须在 OnDraw 函数中为每一个将要显示的 DIB 调用 SelectPalette 函数。这次您可以用 TRUE 参数调用该函数。如果要显示多个 DIB，而每个都有它自己的调色板，那么事情就会很复杂。基本上来说，在调色板消息控制函数里，您应该为其中一个 DIB 选择调色板并施行该调色板(选择调色板时使用 FALSE 参数)。被选中的 DIB 看起来会比其他的 DIB 更好看。虽然有一些合并调色板的方法，但去买更多的显示内存可能会更容易一些。

6.8.2 DIB、像素和颜色表

DIB 包含一个二维数组，该数组的元素称为像素。大多数情况下，每个 DIB 像素都被映射到一个显示像素，但 DIB 像素也可能会被映射到显示器上的一个逻辑区域，这取决于映射模式和显示函数的伸缩参数。

一个像素可以由 1、4、8、16、24 或者 32 个连续位组成，这取决于 DIB 的颜色分辨率。对于 16-bpp、24-bpp 和 32-bpp 的 DIB，每个像素代表一种 RGB 颜色。在 16-bpp 的 DIB 里的每个像素包含 5 位红、5 位绿、5 位蓝值；在 24-bpp 的 DIB 里的每个像素包含 8 位红、8 位绿、8 位蓝值。16-bpp 和 24-bpp 的 DIB 被作了优化，以适应那些可同时显示 65 536 或 16.7 兆种颜色的显示卡。

1-bpp 的 DIB 为单色 DIB，但这些 DIB 并不只有黑和白——它们可以包含 DIB 内部颜色表中的任何两种颜色。一个单色位图有两个 32 位颜色表项入口，每个包含 8 位红、绿、蓝值加上另外 8 位标记。0 像素表示使用第一个表项，1 像素表示使用第二个表项。当您用 65 536 色显示卡或 16 兆色显示卡时，Windows 可以直接显示这两种颜色。(对于 65 536 色显示卡，Windows 会把 8 位颜色值截短到 5 位。)如果显示卡运行在 256 色调色板模式下，则您的程序可以调整系统调色板以装入这两种特殊的颜色。

8-bpp 的 DIB 很常用。类似于单色 DIB，8-bpp 的 DIB 也有一个颜色表，但该颜色表有 256(或少一些)个 32 位表项入口。每个像素是颜色表的索引。如果已经有了调色板模式的显示卡，则您的程序可以根据 256 个表项创建逻辑调色板。如果另外一个程序(运行在前台)控制了系统调色板，那么 Windows 会为逻辑调色板和系统调色板做最佳匹配。

如果在 256 色调色板模式的显示卡上显示 24-bpp 的 DIB，情况会怎么样呢？如果 DIB 的创建者明智的话，他会在 DIB 中包含一个由最重要的颜色组成的颜色表。您的程序可以根据这张表创建一个逻辑调色板，这样 DIB 会看起来很好。如果 DIB 没有颜色表，就使用由 Win32 函数 CreateHalftonePalette 返回的调色板，它比完全不用调色板而只用 20 种标准色要好一些。另一种办法是分析 DIB，得到最重要的颜色，但您完全可以买一个工具来完成这一分析工作。

6.8.3　BMP 文件中 DIB 的结构

我们知道 DIB 是标准的 Windows 位图格式，且 BMP 文件包含一个 DIB。所以我们来分析一下 BMP 文件，看一看 BMP 文件中到底有什么。图 6.2 显示了 BMP 文件的布局。

BITMAPFILEHEADER 结构包含指向图像位的偏移值，您可以用它来计算 BITMAPFILEHEADER 结构和后面的颜色表合起来的大小。BITMAPFILEHEADER 结构包含一个文件大小成员，但您不能依赖于它，因为您不知道该值是按字节、字还是双字计算的。

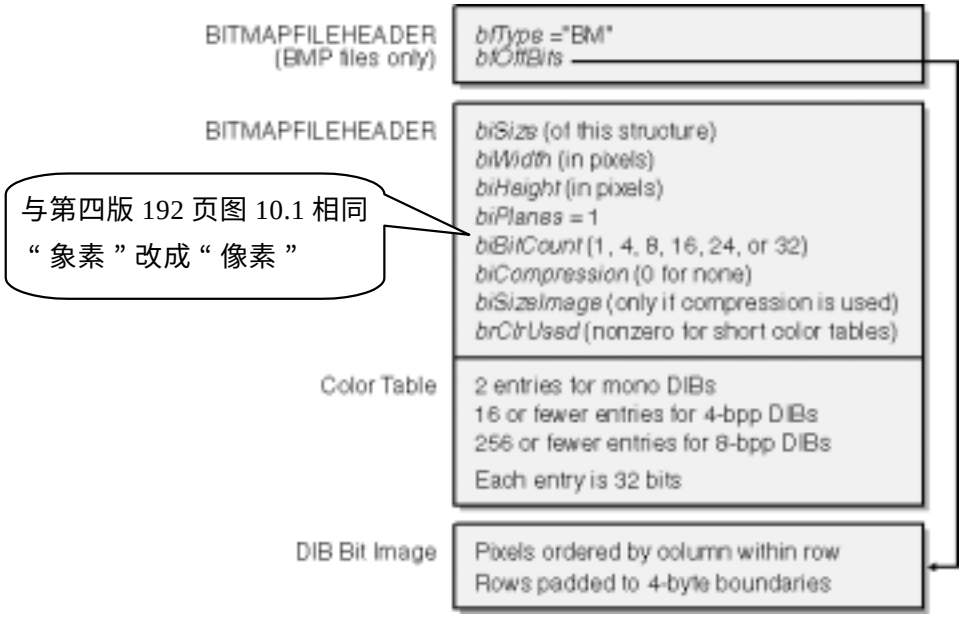


图 6.2　BMP 文件的布局

BITMAPFILEHEADER 结构包含位图的维数、每像素的位数、4-bpp 和 8-bpp 位图的压缩信息和颜色表入口数目。如果 DIB 被压缩了，那么头结构还包含了像素阵列的大小；否则，您可以从位图维数和每像素的位数计算出像素阵列的大小。头结构下面紧接着的是颜色表(如果该 DIB 有颜色表的话)，DIB 图像跟在颜色表后面。DIB 图像包含的像素按行排列，每一行又按列排列，从最下面的行开始排列。每行被扩展到 4 字节边界。

不过，您唯一可以找到 BITMAPFILEHEADER 结构的地方就是在 BMP 文件里。如果您从剪贴板获取 DIB，就没有文件头了。您总是可以通过计算，找到颜色表，它在 BITMAPINFOHEADER 结构的后面，但您不能根据颜色表的尾部来计算图像的位置。例如，如果您使用 CreateDIBSection 函数，则必须先定位位图信息头和颜色表，然后让 Windows 来定位图像的位置。

6.8.4　DIB 访问函数

Windows 支持一些重要的 DIB 访问函数。这些函数都没有被封装到 MFC 中，所以您需要参考联机 Win32 文档中的细节。下面是一些概要介绍：

- SetDIBitsToDevice　该函数直接在显示器或打印机上显示 DIB。显示时不进行缩放，位图的每一位对应一个显示像素或一个打印点。不能进行缩放限制了它的使用。该函数不能像 BitBlt 那样使用，因为 BitBlt 使用的是逻辑坐标。
- StretchDIBits　该函数按照与 StretchBlt 类似的方式将 DIB 直接显示在显示器或打印机上。
- GetDIBits　该函数利用申请到的内存，由 GDI 位图来构造 DIB。您可以对 DIB 的格式进行控制，因为您可以指定每个像素的颜色位数，并且可以指定是否对它进行压缩。如果使用了压缩格式，就必须对 GetDIBits 进行两次调用，一次用于计算所需要的内存，另一次用来产生 DIB 数据。
- CreateDIBitmap　该函数从 DIB 出发来创建 GDI 位图。与所有这些 DIB 函数一样，您必须提供一个设备环境指针作为参数。这里用一个显示设备环境就可以了，不需要内存设备环境。
- CreateDIBSection　该 Win32 函数可以创建一种特殊的 DIB，称为 DIB 项(DIB Section)，然后返回一个 GDI 位图句柄。该函数提供了 DIB 和 GDI 位图最好的特性。您可以直接访问 DIB 的内存，而且利用位图句柄和内存设备环境，还可以调用 GDI 函数在 DIB 中画图。

6.8.5　CDib 类

如果觉得 DIB 有点太限制了，不用担心。CDib 类会使 DIB 编程更容易。了解 CDib 的最好办法是查看它的公有成员函数和数据成员。下面给出了 CDib 的头文件。附带的 CD 中(在 Ex06d 目录下)有函数的实现代码。

CDib.h

```
#ifndef _INSIDE_VISUAL_CPP_CDIB
#define _INSIDE_VISUAL_CPP_CDIB

class CDib : public CObject
{
    enum Alloc {noAlloc, crtAlloc,
```

```
            heapAlloc}; // applies to BITMAPINFOHEADER
    DECLARE_SERIAL(CDib)
public:
    LPVOID  m_lpvColorTable;
    HBITMAP m_hBitmap;
    LPBYTE  m_lpImage;  // starting address of DIB bits
    LPBITMAPINFOHEADER m_lpBMIH; // buffer containing the
                              //  BITMAPINFOHEADER
private:
    HGLOBAL m_hGlobal; // for external windows we need to free;
                       //  could be allocated by this class or
                       //  allocated externally
    Alloc m_nBmihAlloc;
    Alloc m_nImageAlloc;
    DWORD m_dwSizeImage; // of bits—not BITMAPINFOHEADER
                         // or BITMAPFILEHEADER
    int m_nColorTableEntries;

    HANDLE m_hFile;
    HANDLE m_hMap;
    LPVOID m_lpvFile;
    HPALETTE m_hPalette;
public:
    CDib();
    CDib(CSize size, int nBitCount);  // builds BITMAPINFOHEADER
    ~CDib();
    int GetSizeImage() {return m_dwSizeImage;}
    int GetSizeHeader()
        {return sizeof(BITMAPINFOHEADER) +
                sizeof(RGBQUAD) * m_nColorTableEntries;}
    CSize GetDimensions();
    BOOL AttachMapFile(const char* strPathname,
        BOOL bShare = FALSE);
    BOOL CopyToMapFile(const char* strPathname);
    BOOL AttachMemory(LPVOID lpvMem, BOOL bMustDelete = FALSE,
        HGLOBAL hGlobal = NULL);
    BOOL Draw(CDC* pDC, CPoint origin,
        CSize size); // until we implement CreateDibSection
    HBITMAP CreateSection(CDC* pDC = NULL);
    UINT UsePalette(CDC* pDC, BOOL bBackground = FALSE);
    BOOL MakePalette();
    BOOL SetSystemPalette(CDC* pDC);
    BOOL Compress(CDC* pDC,
        BOOL bCompress = TRUE); // FALSE means decompress
    HBITMAP CreateBitmap(CDC* pDC);
```

```
    BOOL Read(CFile* pFile);
    BOOL ReadSection(CFile* pFile, CDC* pDC = NULL);
    BOOL Write(CFile* pFile);
    void Serialize(CArchive& ar);
    void Empty();
private:
    void DetachMapFile();
    void ComputePaletteSize(int nBitCount);
    void ComputeMetrics();
};
#endif // _INSIDE_VISUAL_CPP_CDIB
```

下面让我们来看一下 CDib 的成员函数，首先从构造函数和析构函数开始：

- 默认的构造函数　为了从文件中装入 DIB，或者与内存中的一个 DIB 相联系，可先使用默认构造函数。该函数将创建一个空的 DIB 对象。
- **DIB** 项构造函数　如果需要通过 CreateDIBSection 函数来创建一个 DIB 项的话，就可以使用该构造函数。该函数的参数包括 DIB 的尺寸和颜色数。该构造函数分配了信息头的内存，但不分配图像内存。如果需要自己分配图像内存的话，也可以使用该构造函数。

参 数	说　明
Size	CSize 对象，包含 DIB 的宽度和高度。
nBitCount	每个像素的位数，应该为 1、4、8、16、24 或 32。

- 析构函数　CDib 的析构函数被用来释放所有申请到的 DIB 内存。
- **AttachMapFile**　该函数用读模式打开一个内存映射文件，并与 CDib 对象联系起来。该函数会立即返回，因为文件直到被使用时才会被读进内存。不过，当您访问 DIB 时，在文件装入到内存页时，可能会有一点延迟。AttachMapFile 函数会释放原先已经分配的内存，并关闭原先已经与它相联系的内存映射文件。

参 数	说　明
strPathname	被映射的文件路径名。
bShare	如果文件以共享方式被打开则为 TRUE，默认值为 FALSE。
返回值	如果成功则为 TRUE。

- **AttachMemory**　该函数使已经存在的 CDib 对象和内存中的一个 DIB 联系起来。该内存可能是程序的资源，或者可以是剪贴板甚至是 OLE 数据对象内存。该内存可以是用 new 在 CRT 堆中分配的，也可以是用 GlobalAlloc 在 Windows 堆中分配的。

参 数	说　明
lpvMem	要被联系的内存的地址。
bMustDelete	标记，如果由 CDib 类负责释放内存则为 TRUE，默认值为 FALSE。
hGlobal	假定 bMustDelete 被设置为 TRUE，那么当内存是通过调用 Win32 GlobalAlloc 函数而得到的时，CDib 对象就必须保存相关的句柄，以便以后释放该句柄。
返回值	如果成功则为 TRUE。

- **Compress**　该函数重新产生压缩的或非压缩的 DIB。在函数内部，实际上它把原来的 DIB 转换成一个 GDI 位图，然后再构造一个新的压缩或非压缩的 DIB。只有对于 4-bpp 和 8-bpp 方式的 DIB 才支持压缩。您不能压缩一个 DIB 项(DIB section)。

参 数	说　明
pDC	指向显示设备环境的指针。
bCompress	TRUE 表示压缩 DIB，FALSE 表示解压缩 DIB。
返回值	如果成功则为 TRUE。

- **CopyToMapFile**　该函数创建一个新的内存映射文件，并把 CDib 的数据拷贝到该文件的内存中，同时释放任何原来已经分配的内存并关闭任何已有的内存映射文件。数据要直到新的文件被关闭时才会真正被写到磁盘上，不过当 CDib 对象被重新使用或被销毁时数据也会被写到磁盘上。

参 数	说　明
strPathname	被映射的文件路径名。
返回值	如果成功则为 TRUE。

- **CreateBitmap**　该函数从一个已经存在的 DIB 中创建一个 GDI 位图，Compress 函数调用了该函数。不要把它和 CreateSection 混淆起来，CreateSection 产生一个 DIB，并保存句柄。

参 数	说　明
pDC	指向显示器或打印机设备环境的指针。
返回值	GDI 位图句柄——如果不成功返回 NULL。该句柄并不作为公有数据成员被保存。

- **CreateSection**　该函数调用 Win32 函数 CreateDIBSection 来创建一个 DIB 项。图像内存没有被初始化。

参 数	说 明
pDC	指向显示器或打印机设备环境的指针。
返回值	GDI 位图句柄——如果不成功返回 NULL。该句柄被作为公有数据成员保存起来。

- **Draw** 该函数调用 Win32 函数 StretchDIBits 来将 CDib 对象输出到显示器(或打印机)上。位图会被拉伸到适合指定的矩形。

参 数	说 明
pDC	指向显示器或打印机设备环境的指针，它将接收 DIB 图像。
origin	CPoint 对象，它保存了显示 DIB 时的逻辑坐标。
size	CSize 对象，它保存了显示 DIB 时的矩形的逻辑宽度和高度。
返回值	如果成功，返回 TRUE。

- **Empty** 该函数使 DIB 为空，必要的话，释放分配的内存，关闭映射文件。
- **GetDimensions** 该函数返回的是位图以像素为单位表示的宽和高。

参 数	说 明
返回值	CSize 对象。

- **GetSizeHeader** 该函数返回的是信息头和颜色表合起来的字节数。

参 数	说 明
返回值	32 位整数。

- **GetSizeImage** 该函数返回的是 DIB 图像的字节数(不包括信息头和颜色表)。

参 数	说 明
返回值	32 位整数。

- **MakePalette** 如果颜色表存在，则该函数读入颜色表并创建一个 Windows 调色板。HPALETTE 句柄被保存在一个数据成员中。

参 数	说 明
返回值	如果成功则返回 TRUE。

- **Read** 该函数从文件中将 DIB 读进 CDib 对象中，其中文件必须已经被成功地打开。如果文件是一个 BMP 文件，则从文件的开始处读入；如果文件是一个文档，则从当前的文件指针处开始读入。

参 数	说　明
pfile	指向 CFile 对象的指针，相应的磁盘文件中包含了 DIB。
返回值	如果成功的话，返回值为 TRUE。

- **ReadSection**　该函数从 BMP 文件读入信息头，调用 CreateDIBSection 函数分配图像的内存，然后把图像位从文件中读入到内存里。如果想从磁盘读入 DIB 并调用 GDI 函数编辑该图像的话，就使用该函数。您可以用 Write 或 CopyToMapFile 将 DIB 写回到磁盘上。

参 数	说　明
pfile	指向 CFile 对象的指针，相应的磁盘文件中包含了 DIB。
pDC	指向显示器或打印机设备环境的指针。
返回值	如果成功的话，返回值为 TRUE。

- **Serialize**　序列化过程将在第 16 章中讲述。CDib::Serialize 函数改写了 MFC CObject::Serialize 函数，它调用 Read 和 Write 成员函数。请参见《MSDN Library》中对其参数的说明。
- **SetSystemPalette**　如果您的 16-bpp、24-bpp 或 32-bpp DIB 没有颜色表，则可以调用该函数为 CDib 对象创建一个逻辑调色板，它可与由 CreateHalftonePalette 函数返回的调色板相匹配。如果您的程序运行在 256 色调色板模式显示器下，并且不调用 SetSystemPalette 的话，则根本就不会有调色板，因此在 DIB 中将只出现 20 种标准 Windows 颜色。

参 数	说　明
pDC	指向显示设备环境的指针。
返回值	如果成功则为 TRUE。

- **UsePalette**　该函数把 CDib 对象的逻辑调色板选进设备环境里，然后施行(realize)该调色板。Draw 成员函数在绘制 DIB 之前会调用 UsePalette。

参 数	说　明
pDC	指向要施行调色板的显示设备环境的指针。
bBackground	如果此标记为 FALSE(默认值)，并且应用程序在前台运行，则 Windows 将作为前台调色板施行该调色板(尽可能把更多的颜色拷贝到系统调色板中)。如果此标记为 TRUE，则 Windows 将作为后台调色板施行该调色板(尽可能将逻辑调色板映射到系统调色板)。
返回值	逻辑调色板中被映射到系统调色板中的颜色入口数。如果函数失败，返回值为 GDI_ERROR。

- **Write**　该函数把 DIB 从 CDib 对象写入到文件中，该文件必须已经被成功地打开或创建。

参 数	说　明
pFile	指向 CFile 对象的指针，DIB 将被写进相应的磁盘文件中。
返回值	如果成功，返回值为 TRUE。

为了方便使用，类中提供了四个公有数据成员，它们使我们可以访问 DIB 内存和 DIB 项的句柄。这些成员为我们了解 CDib 对象的结构提供了线索。实际上 CDib 只是一些指向堆内存的指针，这些内存可能为 DIB 占有，也可能为别的模块占有。附加的私有数据成员决定了 CDib 类是否要释放这些内存。

6.8.6　DIB 显示性能

优化的 DIB 处理是 Windows 的一大特点。现代的视频卡有适合标准 DIB 图像格式的帧缓存。如果您有这些卡的话，您的程序就可以充分发挥新的 Windows DIB 引擎的优势，它可以加速直接从 DIB 显示图像的处理过程。然而，如果仍然在 VGA 模式下运行的话，那就很不幸了，程序仍然可以运行，只不过没那么快。

如果 Windows 在 256 色模式下运行，则不管用 StretchBlt 还是 StretchDIBits，8-bpp 位图都会显示得很快。然而，如果显示 16-bpp 或 24-bpp 位图，这些绘制函数就会显得很慢。在这种情况下，如果您创建一个单独的 8-bpp GDI 位图，然后调用 StretchBlt，那么位图的显示会快得多。当然，在创建位图和绘制位图之前必须施行正确的调色板。

在从 BMP 文件装入 CDib 对象后，可以紧接着插入下面的代码：

```
// m_hBitmap is a data member of type HBITMAP
// m_dcMem is a memory device context object of class CDC
m_pDib->UsePalette(&dc);
m_hBitmap = m_pDib->CreateBitmap(&dc); // could be slow
::SelectObject(m_dcMem.GetSafeHdc(), m_hBitmap);
```

您可以用下面的代码代替视图类的 OnDraw 成员函数中的 CDib::Draw：

```
  m_pDib->UsePalette(pDC); // could be in palette msg handler
CSize sizeDib = m_pDib->GetDimensions();
pDC->StretchBlt(0, 0, sizeDib.cx, sizeDib.cy, &m_dcMem,
              0, 0, sizeToDraw.cx, sizeToDraw.cy, SRCCOPY);
```

不要忘了在用完 m_hBitmap 后，要对它调用 DeleteObject。

6.8.7　Ex06d 示例程序

现在您可以把 CDib 类放到应用程序中去。Ex06d 程序显示两个 DIB，一个从资源中装入，另一个从 BMP 文件装入，所以用户可以在运行时选择 BMP 文件。此程序管理系统调色板，可以在打印机上正确显示 DIB。

下面是创建 Ex06d 的步骤。您可以在视图类中输入代码，但您必须用到附带 CD 上的 cdib.h 和 cdib.cpp 文件。

1. **运行 MFC Application Wizard 来产生 Ex06d 项目。**除了下面两项，接受其他所有的默认选项：选择 Single Document，选择 CScrollView 作为视图 CEx06dView 的基类。
2. **引入 Red Blocks 位图。**从 Visual C++ .NET 的 Project 菜单中选择 Add Resource；在 Add Resource 对话框中，点击 Import 按钮。然后，从附带 CD 上的\vcppnet\bitmaps 目录中引入位图 Red Blocks.bmp。Visual C++ .NET 会将该位图文件拷贝到项目的\res 子目录下。请指定其 ID 为 IDB_REDBLOCKS，并保存所做的修改。
3. **把 CDib 类集成到项目中。**如果您是从头开始创建项目的话，请把附带 CD 上的\vcppnet\Ex06d 目录下的 cdib.h 和 cdib.cpp 文件拷贝到项目目录下。仅仅拷贝文件还不够，您还必须把 CDib 文件加到项目中。从 Visual C++ .NET 的 Project 菜单中选择 Add Existing Item，然后选择 cdib.h 和 cdib.cpp，并单击 Open 按钮。现在如果您切换到 Class View 或者 Solution Explorer，就可以看到类 CDib 和它的所有成员变量和函数。
4. **在 CEx06dView 类中加入两个私有 CDib 数据成员。**在 ClassView 窗口里，右键单击 CEx06dView 类，从弹出的快捷菜单中选择 Add Variable，然后加入 m_dibResource 成员。用同样的方法加入 m_dibFile。最终在头文件的最后应该出现下面两个数据成员：

```
CDib m_dibFile;
CDib m_dibResource;
```

ClassView 还会在 Ex06dView.h 文件的最前面加入下面的语句：

```
#include "cdib.h"   // Added by Class View
```

5. **编辑 Ex06dView.cpp 中的 OnInitialUpdate 成员函数。**该函数设置 MM_HIMETRIC 映射模式，并从 IDB_REDBLOCKS 资源中直接装入 m_dibResource 对象。CDib::AttachMemory 函数把 m_dibResource 对象和 EXE 文件里的资源连接起来。请加入下面的黑体代码：

```
void CEx06dView::OnInitialUpdate()
{
    CScrollView::OnInitialUpdate();
    CSize sizeTotal(30000, 40000); // 30-by-40 cm
    CSize sizeLine = CSize(sizeTotal.cx / 100, sizeTotal.cy / 100);
```

```
    SetScrollSizes(MM_HIMETRIC, sizeTotal, sizeTotal, sizeLine);

    LPVOID lpvResource = (LPVOID) ::LoadResource(NULL,
        ::FindResource(NULL, MAKEINTRESOURCE(IDB_REDBLOCKS),
        RT_BITMAP));
    m_dibResource.AttachMemory(lpvResource); // no need for
                                                //  ::LockResource
    CClientDC dc(this);
    TRACE("bits per pixel = %d\n", dc.GetDeviceCaps(BITSPIXEL));
}
```

6. **编辑 Ex06dView.cpp 中的 OnDraw 成员函数。**这部分代码对每一个 DIB 调用 CDib::Draw。UsePalette 函数本应在 WM_QUERYNEWPALETTE 和 WM_PALETTECHANGED 消息控制函数中被调用，但因为这些消息并不被直接发送到视图里，所以难以处理这些消息，因此我们作了简化处理。请在函数中加入下面的黑体代码：

```
void CEx06cView::OnDraw(CDC* pDC)
{
    BeginWaitCursor();
    m_dibResource.UsePalette(pDC); // should be in palette
    m_dibFile.UsePalette(pDC);     //  message handlers, not here
    pDC->TextOut(0, 0,
        "Press the left mouse button here to load a file.");
    CSize sizeResourceDib = m_dibResource.GetDimensions();
    sizeResourceDib.cx *= 30;
    sizeResourceDib.cy *= -30;
    m_dibResource.Draw(pDC, CPoint(0, -800), sizeResourceDib);
    CSize sizeFileDib = m_dibFile.GetDimensions();
    sizeFileDib.cx *= 30;
    sizeFileDib.cy *= -30;
    m_dibFile.Draw(pDC, CPoint(1800, -800), sizeFileDib);
    EndWaitCursor();
}
```

7. **在 CEx06dView 类里映射 WM_LBUTTONDOWN 消息。**编辑 Ex06dView.cpp 文件。OnLButtonDown 包含读入一个 DIB 的两种不同的方法。如果您保留了 MEMORY_MAPPED_FILES 宏定义，则可以调用 AttachMapFile 读入一个内存映射文件；如果您注释掉第一行，则会使用 Read 调用。最后为不带颜色表的 DIB 调用 SetSystemPalette。请在函数中加入下面的黑体代码：

```
#define MEMORY_MAPPED_FILESvoid CEx06cView::OnLButtonDown(UINT nFlags,
CPoint point)
{
    CFileDialog dlg(TRUE, "bmp", "*.bmp");
    if (dlg.DoModal() != IDOK) {
```

```
            return;
        }
    #ifdef MEMORY_MAPPED_FILES
        if (m_dibFile.AttachMapFile(dlg.GetPathName(),
                TRUE) == TRUE) { // share
            Invalidate();
        }
     #else
        CFile file;
        file.Open(dlg.GetPathName(), CFile::modeRead);
        if (m_dibFile.Read(&file) == TRUE) {
            Invalidate();
        }
    #endif // MEMORY_MAPPED_FILES
        CClientDC dc(this);
        m_dibFile.SetSystemPalette(&dc);
    }
```

8. **编译并运行该应用程序。**附带 CD 上的 bitmaps 目录下包含一些有趣的位图。Chicago.bmp 文件是一个 8-bpp DIB，有 256 个颜色表项；forest.bmp 和 clouds.bmp 文件也是 8-bpp 的，但它们的颜色表项少一些。balloons.bmp 是一个 24-bpp DIB，没有颜色表。您可以试一些其他的 BMP 文件。注意，Red Blocks 是一个 16 色 DIB，它使用标准颜色，而这些颜色总是被包括在系统调色板中。

6.9　进一步使用 DIB

Windows 的每一个新版本都会提供更多的 DIB 编程选择。Windows 2000 提供了 LoadImage 和 DrawDibDraw 函数，它们可以用来代替前面讲述的一些 DIB 函数。您可以用这些函数试验一下，看自己的应用程序能否正常工作。

6.9.1　LoadImage 函数

LoadImage 函数可以直接从一个磁盘文件读入位图，并返回一个 DIB 项句柄。假定您要为 CDib 加入一个 ImageLoad 成员函数，其功能类似于 ReadSection。为此，您可以在 cdib.cpp 中加入下面的代码：

```
BOOL CDib::ImageLoad(const char* lpszPathName, CDC* pDC)
{
    Empty();
    m_hBitmap = (HBITMAP) ::LoadImage(NULL, lpszPathName,
```

```
            IMAGE_BITMAP, 0, 0,
            LR_LOADFROMFILE | LR_CREATEDIBSECTION | LR_DEFAULTSIZE);
        DIBSECTION ds;
        VERIFY(::GetObject(m_hBitmap, sizeof(ds), &ds) == sizeof(ds));
        // Allocate memory for BITMAPINFOHEADER
        //  and biggest possible color table
        m_lpBMIH = (LPBITMAPINFOHEADER) new
            char[sizeof(BITMAPINFOHEADER) + 256 * sizeof(RGBQUAD)];
        memcpy(m_lpBMIH, &ds.dsBmih, sizeof(BITMAPINFOHEADER));
        TRACE("CDib::LoadImage, biClrUsed = %d, biClrImportant = %d\n",
            m_lpBMIH->biClrUsed, m_lpBMIH->biClrImportant);
        ComputeMetrics(); // sets m_lpvColorTable
        m_nBmihAlloc = crtAlloc;
        m_lpImage = (LPBYTE) ds.dsBm.bmBits;
        m_nImageAlloc = noAlloc;
        // Retrieve the DIB section's color table
        //  and make a palette from it
        CDC memdc;
        memdc.CreateCompatibleDC(pDC);
        ::SelectObject(memdc.GetSafeHdc(), m_hBitmap);
        UINT nColors = ::GetDIBColorTable(memdc.GetSafeHdc(), 0, 256,
            (RGBQUAD*) m_lpvColorTable);
        if (nColors != 0) {
            ComputePaletteSize(m_lpBMIH->biBitCount);
            MakePalette();
        }
        // memdc deleted and bitmap deselected
        return TRUE;
    }
```

注意，该函数提取并拷贝 BITMAPINFOHEADER 结构，并设置了 CDib 指针数据成员的值。为了提取 DIB 项的调色板，您必须做一点工作，但 Win32 函数 GetDIBColorTable 可以作为起点。有趣的是，GetDIBColorTable 并不能告诉您一个 DIB 使用了多少个调色板项。例如，如果 DIB 只用了 60 项，那么 GetDIBColorTable 会产生一个包含 256 项的颜色表，其中后面的 196 项为 0。

6.9.2 DrawDibDraw 函数

Windows 包含了窗口视频(VFW, Video for Windows)组件，Visual C++ .NET 支持该组件。VFW 的 DrawDibDraw 是一个可以替换 StretchDIBits 的函数。DrawDibDraw 的一个优点是，它可以使用抖动颜色(dithered color)；另一个优点是，当 DIB 的 bpp 值与当前视频模式不匹配时，它能提高显示 DIB 的速度。它主要的缺点是运行时需要把 VFW 代码链接到客户进程中。

下面是一个 CDib 类的 DrawDib 成员函数，它调用了 DrawDibDraw：

```
BOOL CDib::DrawDib(CDC* pDC, CPoint origin, CSize size)
{
    if (m_lpBMIH == NULL) return FALSE;
    if (m_hPalette != NULL) {
        ::SelectPalette(pDC->GetSafeHdc(), m_hPalette, TRUE);
    }
    HDRAWDIB hdd = ::DrawDibOpen();
    CRect rect(origin, size);
    pDC->LPtoDP(rect); // Convert DIB's rectangle
                       //  to MM_TEXT coordinates
    rect -= pDC->GetViewportOrg();
    int nMapModeOld = pDC->SetMapMode(MM_TEXT);
    ::DrawDibDraw(hdd, pDC->GetSafeHdc(), rect.left, rect.top,
        rect.Width(), rect.Height(), m_lpBMIH, m_lpImage, 0, 0,
        m_lpBMIH->biWidth, m_lpBMIH->biHeight, 0);
    pDC->SetMapMode(nMapModeOld);
    VERIFY(::DrawDibClose(hdd));
    return TRUE;
}
```

注意，DrawDibDraw 需要 MM_TEXT 坐标和 MM_TEXT 映射模式。这样，逻辑坐标必须被转换到相对于滚动窗口左上角的像素位置，而不是设备坐标。

为了使用 DrawDibDraw，程序需要#include <vfw.h>语句，而且还要在链接器输入文件中加入 vfw32.lib。DrawDibDraw 可能会假定它所显示的位图在可读写的内存里——如果您把位图的内存映射到 BMP 文件的话，请注意这个假设。

6.10　在按钮上放置位图

使用 MFC 库可以很容易地在按钮上显示一个位图(而不是文字)。如果您要完全自己从头做起，可以先设置按钮的 Owner Draw 属性，然后在对话框类中写一个消息控制函数，以便在该按钮的窗口中画上位图。现在如果使用 MFC 的 CBitmapButton 类，您就可以不用做这么多的事情了，但您必须按照下面的指示去做。不用担心它是如何实现的，反正不用写很多的代码就是了。

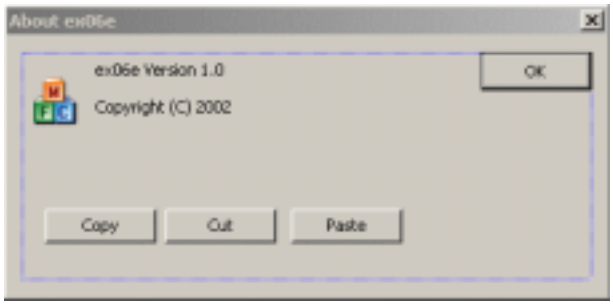
长话短说，像往常一样，先设计一个对话框资源，并为将要添加位图的按钮指定唯一的文本标题，然后在项目中加入一些位图资源，并用名字标识这些资源而不要用数字 ID。最后，在对话框类中加入一些 CBitmapButton 数据成员，并为每一个成员调用 AutoLoad 成员函数，这样就可以使位图名字和按钮标题相匹配。如果按钮标题是“ Copy ”，则加入两个位图：”COPYU”用于按钮的凸起状态，”COPYD”用于按钮的凹下状态。同时，还必须设置按钮的 Owner Draw 属性。(当您编写程序时，可能会更有意思。)

说明　如果您看了 CBitmapButton 类的源代码，就会发现此时的位图是一个普通的 GDI 位图，通过调用 BitBlt 画到按钮上。因此，您就不能指望任何有关调色板的支持。一般情况下，这不会是个问题，因为按钮位图常常是 16 色位图，它依赖标准 VGA 颜色。

6.10.1　Ex06e 示例程序

Ex06e 演示了如何在普通按钮上显示不同的位图。下面是创建 Ex06e 的步骤。

1. **运行 MFC Application Wizard 来创建 Ex06e 项目。**除了下面两项，接受其他所有的默认选项：选择 Single Document，取消对 Printing And Print Preview 的选择。
2. **在 Resource View 中修改项目的 IDD_ABOUTBOX 对话框资源。**我们将使用由 MFC Application Wizard 产生的 About 对话框作为容纳位图按钮的宿主对话框。请加入三个按钮，标题如下页图所示。接受默认的 ID 为 IDC_BUTTON1，IDC_BUTTON2 和 IDC_BUTTON3。按钮的大小并不重要，因为框架会在运行时调整按钮的大小，以适合位图的大小。



将三个按钮的 Owner Draw 属性都设置为 True。

3. **从附带 CD 的\vcppnet\Ex06e 目录中引入三个位图：EditCopy.bmp、EditPast.bmp 和 EditCut.bmp。**从 Project 菜单中选择 Add Resource，然后单击 Import 按钮，从而将这些位图引入到当前项目中。从 EditCopy.bmp 开始，为它分配名字“ COPYU ”。

一定要在名字两边加引号，这样就表示用名字标识资源而不是用 ID。现在就有了按钮凸起时的位图。关闭位图窗口，从 Resource View 窗口中用剪贴板对位图做一份拷贝。重命名该拷贝为“ COPYD ”(凹下状态)，然后编辑该位图。从 Image 菜单中选择 Invert Colors。有一些其他的办法可以进行凸起位图的各种变化，但翻转是最简捷的。

重复以上步骤，建立 EditCut 和 EditPast 位图。完成后，在项目中应该有以下一些位图资源：

资源名	原始文件	翻转颜色
" COPYU "	EditCopy.bmp	否
" COPYD "	EditCopy.bmp	是

续表

资源名	原始文件	翻转颜色
" CUTU "	EditCut.bmp	否
" CUTD "	EditCut.bmp	是
" PASTEU "	EditPast.bmp	否
" PASTED "	EditPast.bmp	是

4. **编辑 CAboutDlg 类的代码。**CAboutDlg 类的声明和实现都在 Ex06e.cpp 文件中。首先，在类的声明里加入三个私有数据成员如下：

```
CBitmapButton m_editCopy;
CBitmapButton m_editCut;
CBitmapButton m_editPaste;
```

然后通过 Properties 窗口，利用代码向导来改写 OnInitDialog 虚函数。对该函数按如下编码：

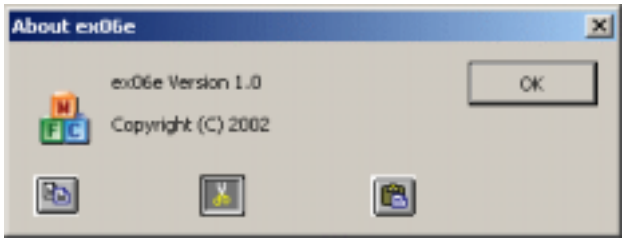
```
BOOL CAboutDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
    VERIFY(m_editCopy.AutoLoad(IDC_BUTTON1, this));
    VERIFY(m_editCut.AutoLoad(IDC_BUTTON2, this));
    VERIFY(m_editPaste.AutoLoad(IDC_BUTTON3, this));
    return TRUE;
 //return TRUE unless you set the focus to a control
 //EXCEPTION: OCX Property Pages should return FALSE
}
```

AutoLoad 函数把每个按钮和两个匹配的资源连接起来。宏 VERIFY 是 MFC 诊断用的，如果位图名字不正确的话就显示一个消息框。

5. **编辑 Ex06eView.cpp 中的 OnDraw 函数。**用下面的代码行代替由 MFC Application Wizard 产生的代码：

```
pDC->TextOut(0, 0, "Choose About from the Help menu.");
```

6. **编译并测试该应用程序。**当程序启动后，从 Help 菜单中选择 About，观察按钮的行为。下图显示了 CUT 按钮被按下时的状态：



注意　位图按钮跟普通按钮一样，也会发送 BN_CLICKED 通知消息。当然，通过 Properties 窗口使用代码向导也可以为对话框类映射这些消息。

6.10.2　进一步使用位图按钮

前面您看到了按钮的凸起和凹下状态的位图。CBitmapButton 类也支持有焦点时和无效时的位图。对于 COPY 按钮，有焦点时的位图名字应该为“ COPYF ”，无效时的位图名字应该为“ COPYX ”。如果您想测试无效时的情况，先做一个“ COPYX ”位图，例如可以在位图中间加一条红线，然后在程序里加入下面的代码行：

```
m_editCopy.EnableWindow(FALSE);
```

文件名：	ch06.doc
目录：	C:\WINDOWS\Desktop\desk
模板：	C:\WINDOWS\Application Data\Microsoft\Templates\技术内幕.dot
题目：	第 6 章 经典的 GDI 函数、字体和位图
主题：	
作者：	liyj
关键词：	
备注：	
创建日期：	2004-4-19 16:06
更改编号：	8
上次保存日期：	2004-6-2 11:29
上次保存者：	zyp
总共编辑时间：	36 分钟
上次打印时间：	2004-6-2 11:34
打印最终结果	
页数：	41
字数：	6,609 （约）
字符数：	37,676 （约）