学生应知信息知识

Visual C++ 简明教程（中）

狄登峰 主编

目　　录

第五章　图形绘制和输出

自从 Windows 操作系统诞生以来，凭借其漂亮的图形界面和友好的用户接口（当然不仅仅是凭借这两点），很快就击败了采用冷冰冰的字符界面的 DOS 操作系统，虽然后者也曾经是 Microsoft 公司的成功产品。Windows 的图形界面也给程序员丰富的想象力提供了发挥的空间，程序员可以使用各种样式的图形和文本来完成程序，即便同样是使用文本，Windows 也是“画”上去的。

当然，程序员的发挥是受到一定限制的。在 Windows 应用程序中，一般都是在窗口的客户区中使用图形和文本，以显示应用程序的数据。当然，使用对话框和控件同样也能够将程序的数据显示给用户。但是对于一个具有“窗口”的应用程序来说，不在窗口中显示点什么似乎总是一种缺憾。因此，在窗口客户区中按特定的方式绘制图形或显示文本就成为了开发 Windows 应用程序时的一项重要工作。

在 MFC 应用程序中，广泛采用了文档/视图结构。文档中保存了应用程序的数据，视图则负责将文档的数据显示出来，因此，在 MFC 应用程序中，大多数的显示工作都是在应用程序的视图类中完成的。本章中将向用户介绍如何在应用程序中使用图形或文本显示数据。

5.1　在文档窗口中绘图

对 Windows 程序而言，将程序运行的结果在文档窗

口上显示出来是最自然的一种选择，本节中就将通过一个例子程序向用户介绍如何在 MFC 应用程序中实现绘图。用户将会看到，在文档窗口上绘图并不困难，基本上就像使用画笔在画布上作画那样简单。但在开始建立例子程序之前，必须对一些基本的概念进行一些解释。

5.1.1 理解设备环境

如果稍微考虑一下实际作画的过程就可以发现，有两样东西是画家所必须的：画布与画笔。在 Windows 程序中“绘图”也是类似的，用户必须有一块电子画布，一支电子画笔，然后才能开始绘图。

在 Windows 程序中，所谓的“设备环境（DC：Device Context）”就是这样的一块电子画布。值得注意的是，设备环境并不仅仅只是用户的显示器屏幕，也有可能是用户的打印机，或者是其他的输出设备。用户在绘图的时候，不需要考虑面对的是显示器设备环境或者打印机设备环境，只要保证了在设备环境上的正确绘图，Windows 系统会通过具体物理设备的驱动程序，将用户需要的图形显现出来，这就是 Windows 系统的设备无关性。

开始绘图的另外一个条件就是用户需要有一支电子画笔。在 Windows 系统中，电子画笔被称为“GDI 对象（GDI Object）”。除了画笔之外，GDI 对象还包括画刷、字体、位图和调色板等。因此，前面简单地说电子画笔可能是不准确的，应该说，GDI 对象就是那些可以用来在设备环境上绘图的工具。

在 Windows 程序中，一个设备环境只能拥有一种画

笔、一种画刷和一种字体，也就是说，一个设备环境不能同时拥有同一类型的多个 GDI 对象。因此，如果用户需要使用一支粗一点的画笔，就需要先创建这样的一支画笔，然后将它选进设备环境代替原来的画笔，然后使用这支新的画笔。这就是在 Windows 应用程序中使用 GDI 对象的一般过程。

1．CDC 类

在 MFC 中，CDC 类实现了对设备环境的封装。所有的绘图操作都必须通过一个 CDC 类（或其派生类）的对象来完成。CDC 类提供了许多函数，可以完成各种与设备环境有关的操作，如选择 GDI 对象、使用颜色与调色板、绘制图形、显示字体、设置设备环境的属性和坐标转换等等。

在 MFC 基础类库中，CDC 类还有几个派生类，包括 CPaintDC 类、CClientDC 类、CWindowDC 类和 CMetaFileDC 类。

在 CDC 类的派生类中，CPaintDC 类只在响应 WM_PAINT 消息的函数中使用，大多数情况下都是 OnPaint()函数。当应用程序的窗口出于某种原因需要更新时，系统就会向应用程序发送 WM_PAINT 消息，从而调用 OnPaint()函数。但是用户不要试图去截获视图类的 WM_PAINT 消息或重载 OnPaint()函数，而应该在 OnDraw()函数中完成绘图任务。这是因为 CView 类已经完成了对 WM_PAINT 消息的捕获，并在 OnPaint()函数中完成了特定了工作。

OnPaint()函数在最后调用了 OnDraw()函数，并且向 OnDraw()传递了一个 CDC 类对象的指针，用户可以通过该指针完成各种绘图操作。这里需要提醒用户的是，

并不是一定要在 OnDraw()函数完成绘图操作，在其他函数中同样是可以的，本节的例子程序就绕过了 OnDraw()函数，在其他的消息处理函数中完成了绘图操作。

CClientDC 类可能是使用最多的 CDC 的派生类，正如其类名所指出，CClientDC 类代表了应用程序窗口的客户区。因此，所有使用 CClientDC 类对象完成的绘图操作都位于窗口的客户区内。

CWindowDC 类封装了与整个窗口有关的设备环境，包括窗口的客户区和非客户区；CMetaFileDC 类封装了与 Windows 元文件有关的绘图操作。用户可以通过联机手册获得这两个派生类的详细信息。

前面已经指出，在使用设备环境之前，必须构造一个 CDC 类（或其派生类）的对象。在 MFC 应用程序中，大部分的绘图操作是针对应用程序窗口客户区的，因此，大多数情况下都使用了 CClientDC 类的对象。

有两种方法可以建立一个 CClientDC 对象，最简单的是这样：

```
CClientDC dc(this)
```

这样的一条语句在程序的堆栈上建立了一个 CClientDC 对象，该对象与当前使用的窗口的客户区相关联。在本节的例子程序中，都是使用这种方式建立 CClientDC 对象的。

另外一种建立 CClientDC 对象是调用 GetDC()函数，该函数的返回值虽然是一个 CDC 类对象的指针，但是实际该对象所代表的是窗口的客户区。需要注意的是，在 CDC 类对象使用结束后，需要调用 ReleaseDC()函数释放设备环境。

2．CGdiObject 类

在 MFC 中，CGdiObject 类封装了 GDI 对象。但是，用户几乎从来不需要直接使用 CGdiObject 类，而是使用该类的派生类。CGdiOjbect 类的派生类包括：CPen 类、CBrush 类、CFont 类、CBitmap 类、CPalette 类和 CRgn 类。

CPen 类封装了 GDI 画笔对象，在 Windows 中，画笔用来绘制各种线条，包括直线、曲线以及各种封闭图形的边框；CBrush 类封装了 GDI 画刷对象，画刷用来填充矩形、椭圆等各种封闭图形的内部区域；CFont 类封装了 GDI 字体对象，字体用来显示文本。

CBitmap 类封装了 GDI 位图对象，并提供了相应的成员函数对位图进行操作，在本书的第四章的“NewCtrl”程序中已经使用过了 CBitmap 类；CPalette 类封装了 Windows 调色板，调色板用来在用户需要的颜色和系统能够提供的颜色之间进行协调；CRgn 类封装了 GDI 区域，区域是窗口中特定的一块，通常用来指定操作的范围，以免干扰其他不需要修改的区域。

在本节的例子程序中，将介绍画笔对象和画刷对象的使用方法。在下一章的例子程序中，介绍了一些使用字体对象进行文本输出的知识。

在上面的叙述中，简要介绍了在 MFC 应用程序中实现屏幕输出的 CDC 类和 CGdiObject 类（及其派生类）的一些基础知识。在本节的例子程序中，将向用户演示使用 CDC 类和 CGdiObject 类对象的具体方法。

5.1.2 建立新的项目

在本节中使用的“Draw”例子程序，将向用户介绍的内容包括：如何建立和使用设备环境对象、如何创建和使用不同的画笔和画刷、如何响应鼠标消息等等。在“Draw”程序中，OnDraw()函数没有承担任何绘图工作，所有的绘图工作都是在相应的消息响应函数中完成的。用户只要理解了设备环境的工作过程，自然也就能够在OnDraw()函数中进行绘图工作。

在“Draw”程序中，用户可以作出相当多的绘图方式选择：

- 选择绘图功能：绘直线、矩形或者是随手画；
- 选择画笔的线宽：1 个像素宽、2 个像素宽或者 5 个像素宽；
- 选择画笔的线型：实线、虚线或者点线；
- 选择画刷的类型：空画刷、白色画刷、垂直条纹画刷或者是斜条纹的画刷；
- 选择画笔和画刷使用的颜色。

此外，“Draw”程序还提供了图形拉伸的功能。本节将向用户展示如何逐步实现这些功能。

建立“Draw”程序的工作与前面的例子程序没有什么不同。在“File”菜单下选择“New”命令，建立新的 MFC 应用程序项目“Draw”。在 AppWizard 中，在步骤一中选择“单文档界面”，选择“英语（美国）”为应用程序资源的语言类型；在步骤二中接受缺省设置；在步骤三中清除“ActiveX Controls”选项；在步骤四中清除“Print and print preview”选项；在步骤五、六中保持缺

省设置。最后，单击“ Finish ”按钮完成应用程序的设置，并生成“ Draw ”程序的框架代码。

5.1.3 实现绘图功能

对于“ Draw ”程序，首要的任务是实现计划中的三种绘图功能：画直线、画矩形和随手画。用户或许期待着能够绘制更多类型的图形，如椭圆、多边形等等，但是作为一个例子程序而言，实现这三种绘图功能已经足够了。用户如果真正理解了实现这些功能的过程，其他的绘图功能也不是什么困难的事情，只要调用合适的函数就可以了。

1．选择绘图功能

在“ Draw ”程序中，每次只能选择和使用一种绘图功能，因此必须提供给用户在绘图功能中进行切换的方法。“ Draw ”程序通过菜单提供了选择绘图功能的方法。为了让程序能够识别当前使用的功能，“ Draw ”程序采用了一种比较“ 笨 ”的方法，就是对每一种绘图功能相应地设置了一个标志变量，在程序运行的过程中检测相应的变量是否为“ TRUE ”就可以了。这有些类似于本书第三章“ AlignMode ”程序中选择水平对齐方式的处理方法，虽有重复累赘之嫌，对于这种小的例子程序还是适用的。

“ Draw ”程序需要在“ Edit ”菜单和“ View ”菜单之间增加一个新的下拉菜单“ Draw ”，在该下拉菜单中提供了菜单项供用户选择当前的绘图功能，最后程序运行时的“ Draw ”菜单如图 5-1 所示。

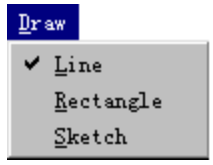

图 5-1 “Draw”菜单

表 5-1 中列出了各菜单项的需要设置的属性。

表 5-1 “Draw”菜单中的菜单项

命令 ID	标题文本	提示信息
ID_DRAW_LINE	Line	Draw a line.
ID_DRAW_RECT	Rectangle	Draw a rectangle.
ID_DRAW_SKETCH	Sketch	Sketch as you want.

相应地，在 CDrawView 类中需要添加三个 BOOL 类型的成员变量，如下所示：

```
protected:
    BOOL m_bLine;          // 绘直线功能标志
    BOOL m_bRect;          // 绘矩形功能标志
    BOOL m_bSketch;        // 随手画功能标志
```

用户需要使用 ClassWizard 对“Draw”菜单下各菜单项的 COMMAND 消息和 UPDATE_COMMAND_UI 消息生成消息处理函数，在这些消息处理函数中，需要对上述标志变量进行相应的设置。程序清单 5-1 中列出了这些函数的代码，这其中还包括 CDrawView 类的构造函数，在构造函数中对上述标志变量进行了初始化。

程序清单 5-1 设置标志变量

```
CDrawView::CDrawView()
{
    // TODO: add construction code here
```

```
    m_bLine=TRUE;
    m_bRect=FALSE;
    m_bSketch=FALSE;
}
void CDrawView::OnDrawLine()
{
    // TODO: Add your command handler code here
    m_bLine=TRUE;
    m_bRect=FALSE;
    m_bSketch=FALSE;
}
void CDrawView::OnDrawRect()
{
    // TODO: Add your command handler code here
    m_bLine=FALSE;
    m_bRect=TRUE;
    m_bSketch=FALSE;
}
void CDrawView::OnDrawSketch()
{
    // TODO: Add your command handler code her
    m_bLine=FALSE;
    m_bRect=FALSE;
    m_bSketch=TRUE;
}
void CDrawView::OnUpdateDrawLine(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
```

```
    pCmdUI->SetCheck(m_bLine);
}
void CDrawView::OnUpdateDrawRect(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    pCmdUI->SetCheck(m_bRect);
}
void CDrawView::OnUpdateDrawSketch(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    pCmdUI->SetCheck(m_bSketch);
}
```

在程序清单 5-1 中的代码都很简单，因此没有添加任何注释，用户应该可以看懂。这些函数只是完成了设置各标志变量的任务，真正的绘图功能的实现还需要更多的努力。

2．实现绘直线与绘矩形功能

在很多绘图程序中，最简单的如 Windows 系统附带的“ PaintBrush ”程序，都是使用鼠标进行操作的。在这些程序中，只要选择了相应的功能，通过鼠标的“ 拖曳 ”就可以完成任务。“ Draw ”程序也将使用鼠标完成绘图操作。

用户已经知道，一条直线需要两点才能确定，在 MFC 应用程序中也不例外。设想一下具体的操作就可以发现，按下鼠标左键的位置应该是直线的起点，松开鼠标左键的位置则是直线的终点。因此，需要响应鼠标的 WM_LBUTTONDOWN 和 WM_LBUTTONUP 消息，并在相应的消息处理函数中记录直线的起点和终点，完成

画线任务。

为了记录直线的起点和终点，还需要向 CDrawView 类添加两个成员变量，如下所示：

```
protected:
    CPoint m_LastPoint;          //上一点的位置
    CPoint m_CurPoint;           //当前点的位置
```

用户可能已经注意到这两个成员变量的名称和注释，并没有使用“起点”、“终点”之类的名称，这是因为这两个变量还将被其他的绘图功能所使用，而其他的功能中不一定有“起点”和“终点”这样的说法。CPoint 类的两个成员 cx 和 cy 保存了点在 x 方向和 y 方向上的坐标。

用户可以使用 ClassWizard 或者 WizardBar 生成对鼠标消息的处理函数，由 CDrawView 类接收鼠标消息，并接受缺省的函数名：OnLButtonDown() 和 OnLButtonUp()。程序清单 5-2 中列出了这两个函数的代码。

程序清单 5-2　OnLButtonDown()和 OnLButtonUp()

```
void CDrawView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    // 记录上一点的位置
    m_LastPoint=point;
    CView::OnLButtonDown(nFlags, point);
}
void CDrawView::OnLButtonUp(UINT nFlags, CPoint point)
```

```
{
    // TODO: Add your message handler code here and/or call default
    // 建立客户区设备环境对象
    CClientDC dc(this);
    // 记录当前点的位置
    m_CurPoint=point;
    // 绘直线
    if(m_bLine)
    {
        dc.MoveTo(m_LastPoint);
        dc.LineTo(m_CurPoint);
    }
    // 绘矩形
    if(m_bRect)
    {

dc.Rectangle(m_LastPoint.x,m_LastPoint.y,m_CurPoint.x,m_CurPoint.y);
    }
    CView::OnLButtonUp(nFlags, point);
}
```

OnLButtonDown()的参数有两个，nFlags 参数中包含了一些当前输入设备的信息，这里不需要；point 参数中保存了 WM_LBUTTONDOWN 事件发生时鼠标的位置，这也正是需要保存的直线的“起点”。

OnLButtonUp()函数中，首先建立了一个客户区设备环境对象，注意传递给 CClientDC 对象构造函数的参数“this”。在 C++语言中，“this”关键字只能在类、结构或联合中使用，代表的就是调用该成员函数的对象。在

这里，OnLButtonUp()函数是 CDrawView 类的一个成员函数，因此，“ this ”代表的是一个 CDrawView 类对象，即应用程序的视图。因此，最终建立的设备环境就是视图的客户区的设备环境，所有的绘图操作都显示在视图的客户区中。

在 OnLButtonUp()函数中，接着保存了直线的“ 终点 ”。在对当前选中的绘图功能进行判断后，调用了 CClientDC 类的 MoveTo()函数将设备环境的当前位置移动到直线的“ 起点 ”，然后调用 LineTo()函数从“ 起点 ”向“ 终点 ”画了一条直线。这样就实现了绘直线的功能。

绘矩形的代码是类似的，只不过 Rectangle()函数可以直接接受两个点的坐标作为参数，因此不需要进行当前点的移动而已。

3．实现随手画功能

与实现绘直线和绘矩形相比，实现随手画功能要复杂一点。实际上，随手画过程就是用一小段一小段的直线跟踪鼠标的移动过程。试设想一下具体的操作过程：按下鼠标左键，随手画过程开始；在鼠标的移动过程中始终按下鼠标左键保持随手画过程；松开鼠标左键，结束随手画过程。

因此，实现随手画同样需要响应鼠标的 WM_LBUTTONDOWN 和 WM_LBUTTONUP 消息，同时还要响应 WM_MOUSEMOVE 消息。注意并不是鼠标每移动一个像素的距离系统就会发送一次 WM_MOUSEMOVE 消息，而是鼠标每移动一小段距离才发送一次 WM_MOUSEMOVE 消息。

对随手画功能来说，OnLButtonDown()函数中只要记录下随手画开始的位置就可以了，因此现有的代码已