第 3 章 图形与文本

Windows 是一个图形操作系统，Windows 使用图形设备接口(GDI)进行图形和文本的输出。MFC 封装了 GDI 对象，提供 CGdiObject 类和 CDC 类来支持图形和文本的输出。

本章介绍有关图形处理的基本原理，并结合实例介绍使用 CGdiObject 类和 CDC 类在视图中输出简单图形、文本的方法和技巧。

3.1 图形设备接口和设备环境

3.1.1 图形设备接口

Windows 提供了一个称为图形设备接口(Graphics Device Interface，GDI)的抽象接口。GDI 作为 Windows 的重要组成部分，负责管理用户进行绘图操作时功能的转换。用户通过调用 GDI 函数与设备打交道，GDI 通过不同设备提供的驱动程序将绘图语句转换为对应的绘图指令，避免了用户直接对硬件进行操作，从而实现设备无关性。

Windows 引入 GDI 的主要目的是实现设备无关性。所谓设备无关性，是指操作系统屏蔽了硬件设备的差异，使用户编程时一般无须考虑设备的类型，例如不同种类的显示器或打印机。当然，实现设备无关性的另一个重要环节是设备驱动程序。不同设备根据其自身不同的特点(例如分辨率和色彩数目)提供相应的驱动程序。图 3.1 描述了 Windows 应用程序的绘图过程。

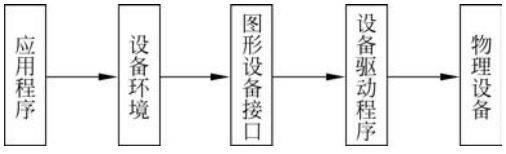

图 3.1 Windows 应用程序的绘图过程

应用程序可以使用 GDI 创建 3 种类型的图形输出，即矢量图形输出、光栅图形输出和文本。

1. 矢量图形输出

矢量图形输出是指画线和填充图形，包括点、直线、曲线、多边形、扇形和矩形等。

2. 光栅图形输出

光栅图形输出是指用光栅图形函数对以位图形式存储的数据进行操作，它包括各种位图和图标的输出。在屏幕上表现为对若干行和列的像素操作，在打印机上则是若干行和列的点阵输出。光栅图形是直接从内存到显存的复制操作，所以速度快，但是对内存的大小要求高。

3. 文本

与在 DOS 下输出文本的方式不同，Windows 中的文本是按图形方式输出的。这样，在输出文本时对输出位置的计算不是以行为单位，而是以逻辑坐标为单位进行计算，这比 DOS 下文本的输出要难一些。但用户可以设置文本的各种效果，例如加粗、斜体、设置颜色等。

3.1.2 设备环境

为了体现 Windows 的设备无关性，应用程序的输出不直接面向显示器或打印机等物理设备，而是面向一个称为设备环境(Device Context，DC)的虚拟逻辑设备。设备环境也称为设备描述表或设备上下文。设备环境是由 GDI 创建用来代表设备连接的数据结构。DC 的功能主要有以下几种：

(1) 允许应用程序使用一个输出设备。

(2) 提供 Windows 应用程序、设备驱动和输出设备之间的连接。

(3) 保存当前信息，例如当前的画笔、画刷、字体和位图等图形对象及其属性，以及颜色和背景等影响图形输出的绘图模式。

(4) 保存窗口剪切区域(Clipping Region)，限制程序输出到输出设备中窗口覆盖的区域。

3.1.3 设备环境类

1. 设备环境类 CDC 及其功能

MFC 封装了 DC，提供 CDC 类及其子类以访问 GDI。MFC 提供的设备环境类包括 CDC、CClientDC、CMetaFileDC、CPaintDC 和 CWindowDC 等，其中 CDC 类是 MFC 设备环境类的基类，其他 MFC 设备环境类都是 CDC 类的派生类，如图 3.2 所示。表 3.1 给出了这几个设备环境类的功能。

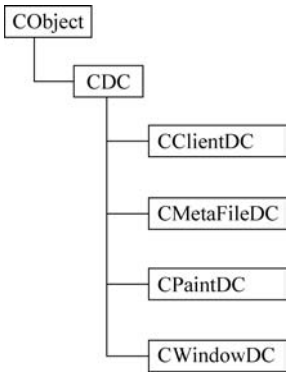

图 3.2 CDC 类及其子类

表 3.1 设备环境类的功能

设备环境类	功 能 描 述
CDC	所有设备环境类的基类，对 GDI 的所有绘图函数进行了封装；可用来直接访问整个显示器或非显示设备(如打印机等)的上下文
CClientDC	代表窗口客户区的设备环境，一般在响应非窗口消息并对客户区绘图时要用到该类
CMetaFileDC	代表 Windows 图元文件的设备环境；一个 Windows 图元文件包括一系列的图形设备接口命令，可以通过重放这些命令来创建图形；对 CMetaFileDC 对象进行的各种绘制操作可以被记录到一个图元文件中
CPaintDC	CPaintDC 用于响应窗口重绘消息(WM_PAINT)的绘图输出，不仅可以对客户区进行操作，还可以对非客户区进行操作
CWindowDC	代表整个窗口的设备环境，包括客户区和非客户区；除非要自己绘制窗口边框和按钮，否则一般不用它

2. 设备环境类 CDC 的一些常用函数

CDC 类提供了基本的绘图操作函数，例如画点、画线、画圆、画矩形、画多边形等。表 3.2

列出了一些常用函数及其功能，其他绘图函数的使用可以查阅 MSDN。

表 3.2　设备环境类 CDC 的一些常用函数

函　数	功　能	举　例
Arc()	根据指定的矩形绘制内切椭圆上的一段弧	pDC-> Arc(20,200,200,300,200,250,20,200);
Chord()	绘制弦形，弦形是一条椭圆弧和其对应的弦所组成的封闭图形	pDC-> Chord (420, 120, 540, 240, 520, 160, 420, 180);
Ellipse()	根据指定的矩形绘制一个内切圆或椭圆	CRect rect(0,0,100,100); pDC-> Ellipse(&rect);　　//在矩形内画圆 CRect rect(0,0,50,100); pDC-> Ellipse(&rect);　　//在矩形内画椭圆
LineTo()	从当前位置到指定位置画一条直线	pDC-> LineTo(100,100);
MoveTo()	移动当前位置到指定的坐标	pDC-> MoveTo(0,0);
Polyline()	绘制连接指定点的折线段	POINT pt[3]={{10,100},{50,60},{120,80}}; pDC-> Polyline(pt,3);
PolyBezier()	根据两个端点和两个控制点绘制贝塞尔曲线	POINT pt[4] = {{10, 100}, {50, 60}, {120, 80}, {150, 160}}; pDC-> PolyBezier(pt,4);
Pie()	绘制一个饼块	pDC-> Pie(420,120,540,240,520,160,420,180);
Polygon()	根据两个或两个以上的顶点绘制一个多边形	POINTpt[3]={{10,100},{50,60},{120,80}}; pDC-> Polygon(pt,3);
Rectangle()	根据指定的左上角和右下角坐标绘制一个矩形	CRect rect(0,0,100,100); pDC-> Rectangle(&rect);
RoundRect()	绘制一个圆角矩形	pDC-> RoundRect(400,30,550,100,20,20);
SetPixel()	用指定颜色在指定坐标画一个点	pDC-> SetPixel(CPoint(200,200),RGB(255,0, 0));

3.1.4　颜色的设置

在绘制图形和图像时，颜色是一个重要的因素。Windows 用 COLORREF 类型的数据存放颜色，它实际上是一个 32 位整数。任何一种颜色都是由红、绿、蓝 3 种基本颜色组成的，COLORREF 类型数据的低位字节存放红色强度值，第 2 个字节存放绿色强度值，第 3 个字节存放蓝色强度值，高位字节为 0，每一种颜色分量的取值范围为 0～255。如果显卡能支持，用户利用 COLORREF 数据类型定义颜色的种类可以超过 1600 多万种。

直接设置 COLORREF 类型的数据不太方便。MFC 提供了 RGB 宏，用于设置颜色，它将其中的红、绿、蓝分量值转换为 COLORREF 类型的颜色数据，其使用形式为

```
RGB(byRed,byGreen,byBlue)
```

其中，参数 byRed、byGreen 和 byBlue 分别表示红、绿、蓝分量值（范围为 0～255）。例如，RGB(0,0,0)表示黑色，RGB(255,0,0)表示红色，RGB(0,255,0)表示绿色，RGB(0,0,255)表示蓝色。表 3.3 列出了一些常用颜色的 RGB 值。

表 3.3　常用颜色的 RGB 值

颜　色	RGB 值	颜　色	RGB 值
黑色	0,0,0	深青色	0,128,128
白色	255,255,255	红色	255,0,0
蓝色	0,0,255	深红色	128,0,0
深蓝色	0,0,128	灰色	192,192,192
绿色	0,255,0	深灰色	128,128,128
深绿色	0,128,0	黄色	255,255,0
青色	0,255,255	深黄色	128,128,0

很多涉及颜色的 GDI 函数都需要使用 COLORREF 类型的参数，例如设置背景色的成员函数 CDC::SetBkColor()、设置文本颜色的成员函数 CDC::SetTextColor()。

下面的代码说明如何使用 RGB 宏。

```
COLORREF rgbBkClr = RGB(192,192,192);                    //定义灰色
pDC->SetBkColor(rgbBkClr);                               //背景色为灰色
pDC->SetTextColor(RGB(0,0,255));                         //文本颜色为蓝色
```

3.1.5　获取设备环境

在绘图前必须准备好设备环境。设备环境不像其他 Windows 结构，在程序中不能直接存取，只能通过系统提供的一系列函数或使用设备环境句柄 HDC 来间接地获取或设置设备环境结构中的各项属性，这些属性包括显示器的高度和宽度、支持的颜色数及分辨率等。

1. 传统的 SDK 获取设备环境的方法

如果采用传统的 SDK 方法编程，获取设备环境的方法有两种：在 WM_PAINT 消息处理函数中通过调用 API 函数 BeginPaint()获取设备环境，在消息处理函数返回前调用 API 函数 EndPaint()释放设备环境；如果绘图操作不是在 WM_PAINT 消息处理函数中，需要通过调用 API 函数 GetDC()获取设备环境，调用 API 函数 ReleaseDC()释放设备环境。

2. MFC 应用程序获取设备环境的方法

如果采用 MFC 方法编程，由于 MFC 提供了不同类型的设备环境类 CDC，每一个类都封装了设备环境句柄，并且它们的构造函数可自动调用上述获取设备环境的 Win32 API 函数，析构函数可自动调用释放设备环境的 Win32 API 函数。因此，在程序中通过声明一个 MFC 设备环境类的对象自动获取了一个设备环境，而当该对象被取消时也就自动释放了获取的设备环境。并且，通过 MFC 项目模板创建的 OnDraw()函数自动支持所获取的设备环境，接受一个参数为指向 CDC 对象的指针。可见在一个 MFC 应用程序中获得 DC 的方法主要有两种：一种是接受一个参数为指向 CDC 对象的指针；另一种是声明一个 MFC 设备环境类的对象，并使用 this 指针为该对象赋值。

视频讲解

3.1.6　编程实例

掌握了上述基本知识后，就可以进行简单图形的绘制与输出。下面结合实例介绍上述知识点，进一步比较 CDC 类及其子类的用途。

【例 3.1】 编写一个单文档的 MFC 应用程序 Li3_1，利用表 3.2 中的函数绘制几种常

见的几何图形。当程序运行时，显示如图 3.3 所示的结果。

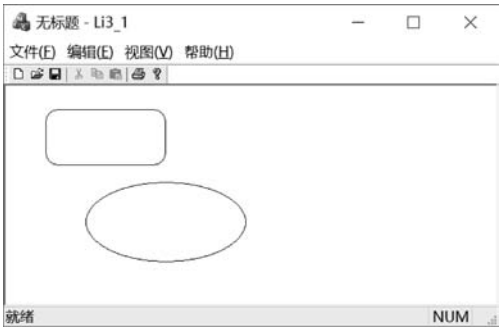

图 3.3　程序 Li3_1 的运行结果

其操作步骤如下：

(1) 创建一个名为 chap03 的解决方案，使用“MFC 应用”项目模板在该解决方案中创建一个单文档的 MFC 应用程序项目 Li3_1，项目样式选择 MFC Standard。

(2) 打开 IDE 中的“类视图”窗口，展开 CLi31View，双击打开 OnDraw()函数，并添加如下代码。

```
void CLi31View::OnDraw(CDC * pDC)
{
    CLi31Doc * pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if(!pDoc)
        return;
    //TODO: 在此处为本机数据添加绘制代码
    pDC->RoundRect(50,30,200,100,30,30);          //绘制圆角矩形
    pDC->Ellipse(100,120,300,220);                //绘制椭圆
}
```

(3) 编译、链接并运行项目，结果如图 3.3 所示。

结果分析与讨论：采用第 3.1.5 节介绍的在一个 MFC 应用程序中获得 DC 的第一种方法；OnDraw()函数自动接受一个参数为指向 CDC 对象的指针，通过该指针调用成员函数即可绘出相应的图形。

视频讲解

【例 3.2】 编写一个单文档的 MFC 应用程序 Li3_2，使用 CPaintDC 类完成与例 3.1 同样的功能。

其操作步骤如下：

(1) 打开解决方案资源管理器，使用“MFC 应用”项目模板在 chap03 解决方案中创建一个单文档的 MFC 应用程序项目 Li3_2，项目样式选择 MFC Standard。

(2) 在“解决方案资源管理器”窗口中右击项目 Li3_2，选择“类向导”菜单命令，打开“类向导”工具。使用该工具在 CLi32View 类中添加消息 WM_PAINT 的消息响应函数 OnPaint()。

(3) 在 OnPaint()函数中添加以下代码。

```
void CLi32View::OnPaint()
{
```

```
    CPaintDC dc(this);
    //TODO: 在此处添加消息处理程序代码
    dc.RoundRect(100, 30, 250, 100, 30, 30);          //绘制圆角矩形
    dc.Ellipse(200, 100, 400, 150);                   //绘制椭圆
    //不为绘图消息调用 CView::OnPaint()
}
```

(4) 编译、链接并运行程序,结果如图 3.3 所示。

结果分析与讨论:采用第 3.1.5 节介绍的在一个 MFC 应用程序中获得 DC 的第二种方法;在 OnPaint()函数中声明一个 CPaintDC 类的对象,并使用 this 指针为该对象赋值。

【例 3.3】 编写一个基于对话框的 MFC 应用程序 Li3_3,程序运行后,当用户在视图窗口中单击鼠标左键时,在窗口中绘制一个矩形。


视频讲解

其操作步骤如下:

(1) 打开解决方案资源管理器,使用“MFC 应用”项目模板在 chap03 解决方案中创建一个基于对话框的 MFC 应用程序项目 Li3_3,设置对话框标题为“Li3_3”。

(2) 打开对话框编辑器,删除主对话框中的静态文本及按钮。

(3) 打开“类向导”工具,在 CLi33Dlg 类中增加消息 WM_LBUTTONDOWN 的消息响应函数 OnLButtonDown()。

(4) 在函数 OnLButtonDown()中添加以下代码。

```
void CLi33Dlg::OnLButtonDown(UINT nFlags, CPoint point)
{
    //TODO: 在此添加消息处理程序代码或调用默认值
    CClientDC dc(this);
    dc.Rectangle(100, 0, 200, 150);
    CDialogEx::OnLButtonDown(nFlags, point);
}
```

(5) 编译、链接并运行程序,结果如图 3.4 所示。

结果分析与讨论:采用第 3.1.5 节中介绍的在一个 MFC 应用程序中获得 DC 的第二种方法;在 OnLButtonDown()函数中声明一个 CClientDC 类的对象,并使用 this 指针为该对象赋值;通过该对象引用成员函数即可绘制出相应的图形;可以将 CClientDC 类对象添加到其他消息映射处理函数中,这样当处理相应消息映射时就可以绘制图形。

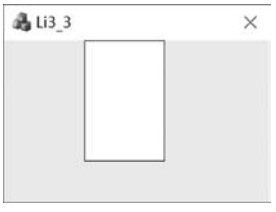

图 3.4 程序 Li3_3 的运行结果

【例 3.4】 编写一个基于对话框的 MFC 应用程序 Li3_4,使用 CWindowDC 类完成与例 3.3 同样的功能。


视频讲解

其操作步骤如下:

(1) 打开解决方案资源管理器,使用“MFC 应用”项目模板在 chap03 解决方案中创建一个基于对话框的 MFC 应用程序项目 Li3_4,设置对话框标题为“Li3_4”。

(2) 打开对话框编辑器,删除主对话框中的静态文本及按钮。

(3) 打开“类向导”工具,在 CLi34Dlg 类中增加消息 WM_LBUTTONDOWN 的消息响应函数 OnLButtonDown()。

(4) 在函数 OnLButtonDown()中添加以下代码。

```
void CLi34Dlg::OnLButtonDown(UINT nFlags, CPoint point)
{
    //TODO: 在此添加消息处理程序代码或调用默认值
    CWindowDC dc(this);
    dc.Ellipse(100, 0, 200, 150);
    CDialogEx::OnLButtonDown(nFlags, point);
}
```

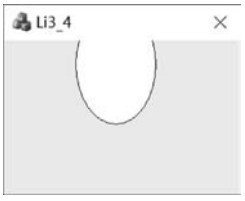

图 3.5 程序 Li3_4 的运行结果

(5) 编译、链接并运行程序,结果如图 3.5 所示。

结果分析与讨论:运行后得到一个超出窗口客户区域的椭圆;CWindowDC 类与 CClientDC 类相似,最大的不同就是 CWindowDC 在整个应用程序窗口上画图;除非要自己绘制窗口边框和按钮,否则一般不使用它。

可见 CPaintDC、CClientDC 与 CWindowDC 有一个共同的特点,就是需要使用某个窗口对象定义设备环境。

3.2 GDI 对象类

在默认状态下,当用户创建一个设备环境并在其中绘图时,系统使用设备环境默认的绘图工具及其属性。如果要使用不同风格和颜色的绘图工具进行绘图,用户必须重新为设备环境设置自定义的画笔和画刷等绘图工具。这些绘图工具统称为 GDI 对象。

GDI 对象是 Windows 图形设备接口的抽象绘图工具。除了画笔和画刷以外,其他 GDI 对象还包括字体、位图和调色板。MFC 对 GDI 对象进行了很好的封装,提供了封装 GDI 对象的类,例如 CPen、CBrush、CFont、CBitmap 和 CPalette 等,这些类都是 GDI 对象类 CGdiObject 的派生类,它们的继承关系如图 3.6 所示。

1. CPen 类

该类封装 GDI 画笔,用于绘制对象的边框以及直线和曲线。默认的画笔用于绘制与一个像素等宽的黑色实线。

2. CBrush 类

该类封装 GDI 画刷。画刷用来填充一个封闭图形对象(例如矩形、圆形)的内部区域,默认的画刷颜色是白色。

3. CFont 类

该类封装 GDI 字体对象,用来绘制文本。用户可以建立一种 GDI 字体,并使用 CFont 的成员函数来访问它,主要用于设置文本的输出效果,包括文字的大小、是否加粗、是否斜体、是否加下画线等。

4. CBitmap 类

该类封装 GDI 位图,提供成员函数装载和位图操作。位图可以用于填充区域。

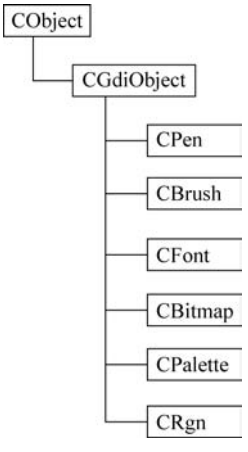

图 3.6 CGdiObject 类及其子类

5. CPalette 类

该类封装 GDI 调色板，包含系统可用的色彩信息，是应用程序和彩色输出设备环境（例如显示器）的接口。

6. CRgn 类

该类封装 GDI 区域。区域是窗口内的一块多边形或椭圆形区域。CRgn 用于设备环境（通常是窗口）内的区域操作，通常和 CDC 类中与裁剪（clipping）有关的成员函数配合使用。

3.3 画笔和画刷的使用

3.3.1 使用库存对象

无论是以 CDC 类对象指针形式还是以 CDC 子类对象形式获得设备环境，系统都默认指定了一组绘图属性，见表 3.4。

表 3.4 系统默认的绘图属性

绘图属性	默认值	改变默认值的函数
画笔	一个像素宽的黑色实线	SelectObject()，SelectStockObject()
画刷	白色的实心刷	SelectObject()，SelectStockObject()
背景颜色	白色	SetBKColor()
背景模式	OPAQUE	SetBKMode()
刷子原点	设备坐标(0,0)	SetBrushOrg()
当前绘图位置	逻辑坐标(0,0)	MoveTo()
混合模式	R2_COPYPEN	SetRop2()
映射模式	MM_TEXT	SetMapMode()

库存（stock）对象是由操作系统维护的用于绘制屏幕的常用对象，包括库存画笔、画刷、字体等。使用 SelectStockObject()函数可以直接选择库存对象，修改系统默认值。如果选择成功，SelectStockObject()函数将返回以前的 CGdiObject 对象的指针，需要将返回值转换为相匹配的 GDI 对象的指针。函数参数用于指定选择的是哪一种 GDI 库存对象，可直接选用的库存对象见表 3.5。

表 3.5 可供选用的库存对象

GDI 分类	库存对象值	说明
Pens	BLACK_PEN	黑色画笔
	WHITE_PEN	白色画笔
	NULL_PEN	空画笔
Brushs	BLACK_BRUSH	黑色画刷
	WHITE_BRUSH	白色画刷
	GRAY_BRUSH	灰色画刷
	LTGRAY_BRUSH	浅灰色画刷
	DKGRAY_BRUSH	深灰色画刷
	HOLLOW_BRUSH	虚画刷
	NULL_BRUSH	空画刷

用户也可以利用 CGdiObject 类的成员函数 CreateStockObject()将 GDI 对象设置成指定的库存对象，这时需要首先声明一个 GDI 对象，最后还需要调用成员函数 SelectObject()，将与库存对象关联的 GDI 对象选入当前的设备环境，代码如下：

```
CBrush * BrushOld,BrushNew;
BrushNew.CreateStockObject(BLACK_BRUSH);                    //关联库存画刷对象
BrushOld = pDC->SelectObject(&BrushNew);
```

视频讲解

【例 3.5】 编写一个单文档应用程序 Li3_5，使用库存画笔和画刷在视图中绘制图形。其操作步骤如下：

(1) 打开解决方案资源管理器，使用“MFC 应用”项目模板在 chap03 解决方案中创建一个单文档的 MFC 应用程序项目 Li3_5，项目样式选择 MFC Standard。

(2) 在“类视图”中选择 Li3_5 项目的 CLi35View 类，打开其成员函数 OnDraw()。

(3) 在 OnDraw()函数中添加以下代码。

```
void CLi35View::OnDraw(CDC* pDC)
{
    CLi35Doc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if(!pDoc)
        return;
    //TODO: 在此处为本机数据添加绘制代码
    CPen* PenOld, PenNew;
    CBrush* BrushOld, BrushNew;
    //选用库存黑色画笔
    PenOld = (CPen*)pDC->SelectStockObject(BLACK_PEN);
    //选用库存浅灰色画刷
    BrushOld = (CBrush*)pDC->SelectStockObject(LTGRAY_BRUSH);
    pDC->Rectangle(100, 100, 300, 300);
    //关联 GDI 库存对象
    PenNew.CreateStockObject(WHITE_PEN);
    pDC->SelectObject(&PenNew);
    pDC->MoveTo(100, 100);
    pDC->LineTo(300, 300);
    pDC->MoveTo(100, 300);
    pDC->LineTo(300, 100);
    //恢复系统默认的 GDI 对象
    pDC->SelectObject(PenOld);
    pDC->SelectObject(BrushOld);
}
```

(4) 编译、链接并运行程序，结果如图 3.7 所示。

结果分析与讨论：采用两种不同的方法使用库存对象，首先用库存黑色画笔画一个矩形外框，然后用库存浅灰色画刷填充矩形内部，最后用另外一种方法选择库存白色画笔在矩形内画两条白色的对角线。

3.3.2 创建和使用自定义画笔

如果要在设备环境中使用自己的画笔绘图，首先需要创建一个指定风格的画笔，然后选

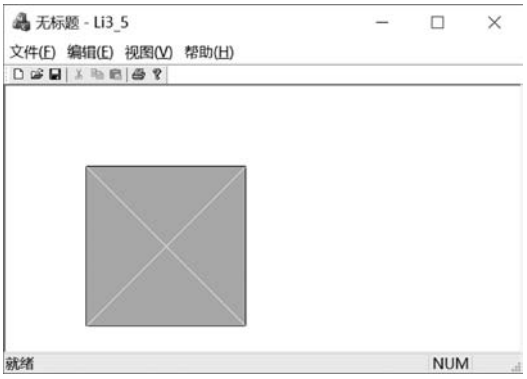

图 3.7　使用库存对象绘制图形

择所创建的画笔,最后还原画笔。

1. 创建画笔

创建画笔的方法有以下两种。

(1) 在定义画笔对象时直接创建,例如:

```
CPen PenNew(PS_DASH,1,RGB(255,0,0));                    //创建一个红色虚线画笔
```

(2) 先定义一个没有初始化的画笔对象,再调用 CreatePen()函数创建指定画笔。例如:

```
CPen Pen;
pen.CreatePen(PS_DASH,1,RGB(255,0,0));
```

其中,第 1 个参数是笔的样式,笔的样式及其说明见表 3.6;第 2 个参数是线的宽度;第 3 个参数是线的颜色。

表 3.6　画笔的样式及其说明

样　式	说　明
PS_SOLID	实线
PS_DOT	点线,当笔的宽度为 1 或更小时有效
PS_DASH	虚线,当笔的宽度为 1 或更小时有效
PS_DASHDOT	点画线,当笔的宽度为 1 或更小时有效
PS_DASHDOTDOT	双点画线,当笔的宽度为 1 或更小时有效
PS_NULL	无,创建不可见笔
PS_INSIDEFRAME	边框实线

2. 选择创建的画笔

不管采用哪种方法,在创建画笔后必须调用 CDC 类的成员函数 SelectObject()将创建的画笔选入当前设备环境。如果选择成功,SelectObject()函数将返回以前画笔对象的指针。在选择新的画笔时应该保存以前的画笔对象。例如:

```
CPen * PenOld;
PenOld = pDC -> SelectObject(&PenNew);                   //保存原来的画笔
```

3. 还原画笔

在创建和选择画笔工具后，应用程序就可以使用该画笔绘图了。当绘图完成后，应该通过调用 CDC 类的成员函数 SelectObject()来恢复设备环境以前的画笔工具，并通过调用 CGdiObject 类的成员函数 DeleteObject()来删除画笔，释放 GDI 对象所占的内存资源。例如：

```
pDC->SelectObject(PenOld);                    //恢复设备环境中原来的画笔
PenNew.DeleteObject();                        //删除创建的画笔
```

视频讲解

【例 3.6】 编写一个单文档应用程序 Li3_6，绘制不同样式、线宽及颜色的矩形。

其操作步骤如下：

(1) 打开解决方案资源管理器，使用“MFC 应用”项目模板在 chap03 解决方案中创建一个单文档的 MFC 应用程序项目 Li3_6，项目样式选择 MFC Standard。

(2) 在“类视图”中选择 Li3_6 项目的 CLi36View 类，打开其成员函数 OnDraw()。

(3) 在 OnDraw()函数中添加以下代码。

```
void CLi36View::OnDraw(CDC* pDC)
{
    CLi36Doc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if(!pDoc)
        return;
    //TODO: 在此处为本机数据添加绘制代码
    CPen* PenOld, PenNew;
    int PenStyle[] = {PS_SOLID,PS_DOT,PS_DASH};              //画笔样式
    COLORREF rgbPenClr[] = {RGB(255,0,0),RGB(0,255,0),RGB(0,0,255)};
    for(int i = 0; i < 3; i++)
    {
        PenNew.CreatePen(PenStyle[i], 2 - i, rgbPenClr[i]); //创建画笔
        PenOld = pDC->SelectObject(&PenNew);                //选用画笔
        pDC->Rectangle(20 + 50 * i, 20 + 50 * i, 120 + 50 * i, 50 + 50 * i);
        pDC->SelectObject(PenOld);                          //还原画笔
        PenNew.DeleteObject();                              //释放画笔
    }
}
```

(4) 编译、链接并运行程序，结果如图 3.8 所示。

图 3.8 使用自定义画笔绘图

注意：不能删除正被选入设备环境的 GDI 对象，并且删除 GDI 对象不等于删除相关联的 C++对象。以上面的例子为例，调用 CPen 对象的 DeleteObject()(该成员函数在基类

CGdiObject 中定义)只是删除与之相关联的画笔对象所占用的系统资源,而 C++语义上的 CPen 对象并没有被删除。另外,还可以调用其成员函数 CreatePen()创建新的画笔对象,并将它们与 CPen 对象相关联。

3.3.3 创建和使用自定义画刷

和画笔一样,使用自定义画刷也包括创建画刷、选择创建的画刷和还原画刷等步骤。

在创建画刷时首先构造一个没有初始化的 CBrush 画刷对象,然后调用 CBrush 类的初始化成员函数创建定制的画刷工具。不同于 CPen 类的是,类型不同的画刷使用不同的函数实现。CBrush 类提供的创建函数常用的有以下几个。

1. 创建指定颜色的实心画刷函数 CreateSolidBrush()

其原型为

```
BOOL CreateSolidBrush(COLORREF crColor);
```

例如,创建一个红色的实心画刷:

```
CBrush brush;
brush.CreateSolidBrush(RGB(255,0,0));
```

2. 创建阴影画刷函数 CreateHatchBrush()

其原型为

```
BOOL CreateHatchBrush(int nIndex,COLORREF crColor);
```

其中,参数 nIndex 用于指定阴影样式,它的值见表 3.7。例如,创建一个具有水平和垂直交叉阴影线的红色画刷:

```
CBrush brush;
brush.CreateHatchBrush(HS_CROSS,RGB(255,0,0));
```

表 3.7 画刷的阴影样式

阴影样式	说明
HS_BDIAGONAL	从左下角到右上角的 45°阴影线
HS_FDIAGONAL	从左上角到右下角的 45°阴影线
HS_DIAGCROSS	十字交叉的 45°阴影线
HS_ CROSS	水平和垂直交叉的阴影线
HS_HORIZONTAL	水平阴影线
HS_VERTICAL	垂直阴影线

3. 创建位图画刷函数 CreatePatternBrush()

该函数只有一个指向对象的指针参数,它使用该 CBitmap 所代表的位图做画刷。一般采用 8×8 像素的位图,因为画刷可以看作 8×8 像素的小位图。即使提供给成员函数 CreatePatternBrush()的位图比这个大,也仅有左上角的 8 像素被使用。当 Windows 桌面背景采用图案(例如 weave)填充时使用的就是这种位图画刷。例如:

```
CBitMap mybmp;
```

```
mybmp.LoadBitMap(IDB_MYBMP);
CBrush brush;
brush.CreatePatternBrush(&mybmp);
```


选择创建的画刷。在使用结束后,恢复原来画刷的方法与画笔工具完全一样。


视频讲解

【例 3.7】 编写一个单文档应用程序 Li3_7,绘制不同颜色、不同阴影形式的填充矩形。其操作步骤如下:

(1) 打开解决方案资源管理器,使用“MFC 应用”项目模板在 chap03 解决方案中创建一个单文档的 MFC 应用程序项目 Li3_7,项目样式选择 MFC Standard。

(2) 在“类视图”中选择 Li3_7 项目的 CLi37View 类,打开其成员函数 OnDraw()。

(3) 在 OnDraw()函数中添加以下代码。

```
void CLi37View::OnDraw(CDC * pDC)
{
    CLi37Doc * pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if(!pDoc)
        return;
    //TODO: 在此处为本机数据添加绘制代码
    CBrush * BrushOld, BrushNew;
    int HatchStyle[] = {HS_BDIAGONAL,HS_FDIAGONAL,HS_CROSS};          //阴影样式
    COLORREF BrushClr[] = {RGB(255,0,0),RGB(0,255,0),RGB(0,0,255)};
    for(int i = 0; i < 6; i++)
    {
        if(i < 3)                                                      //创建实心画刷
            BrushNew.CreateSolidBrush(BrushClr[i]);
        else                                                           //创建阴影画刷
            BrushNew.CreateHatchBrush(HatchStyle[i - 3], RGB(0, 0, 0));
        BrushOld = pDC->SelectObject(&BrushNew);                       //选用画刷
        pDC->Rectangle(20 + 40 * i, 20 + 40 * i, 120 + 40 * i, 50 + 40 * i);
        pDC->SelectObject(BrushOld);                                   //还原画刷
        BrushNew.DeleteObject();                                       //释放画刷
    }
}
```

(4) 编译、链接并运行程序,结果如图 3.9 所示。

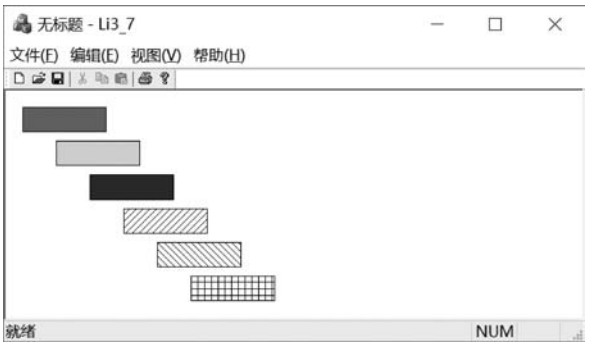

图 3.9 使用自定义画刷绘图

3.4 处 理 文 本

在 Windows 应用程序中经常使用 GDI 处理文本，处理文本的过程包括设置文本的属性、设置字体、格式化文本、调用文本输出函数输出文本等具体步骤。

3.4.1 设置文本的显示属性

在默认情况下输出文本时，字体颜色是黑色，背景颜色是白色，背景模式为不透明模式。用户可以通过调用 CDC 类成员函数重新设置字体颜色、背景颜色和文本对齐方式等文本的显示属性，表 3.8 列出了设置文本显示属性的常用函数。

表 3.8 文本显示属性函数

函 数	功 能	函 数	功 能
SetBkMode()	设置文本的背景模式	SetTextAlign()	设置显示文本的对齐方式
GetBkMode()	获得当前文本的背景模式	GetTextAlign()	获得当前文本的对齐方式

1. 设置背景模式

使用 SetBkColor()函数设置的背景颜色，只有在使用 CDC 成员函数 SetBkMode()设置背景模式后才能在输出时有效。SetBkMode()函数的原型为

```
int SetBkMode(int nBkMode);
```

其中，nBkMode 指定背景模式，其值可以是 OPAQUE(不透明)或 TRANSPARENT(透明)。如果选择 OPAQUE，则在输出文本、使用画笔或画刷前使用当前设置的背景颜色填充背景；如果选择 TRANSPARENT，则在绘制之前背景不改变。其默认方式为 OPAQUE。

2. 设置文本的对齐方式

文本的对齐方式描述了文本坐标(x,y)和文本框之间的关系。其默认的对齐方式是 TA_LEFT|TA_TOP，用户可使用 SetTextAlign()函数改变文本的对齐方式，该函数的原型为

```
UINT  SetTextAlign(UINT nFlags);
```

其中，nFlag 为表 3.9 中列出的标志的组合。例如：

```
CDC * pDC;
pDC->SetTextAlign(TA_RIGHT|TA_BOTTOM);
```

表 3.9 文本对齐标志

对齐标志	说 明	对齐标志	说 明
TA_TOP	上对齐	TA_CENTER	水平居中对齐
TA_BOTTOM	下对齐	TA_BASELINE	垂直居中对齐
TA_LEFT	左对齐	TA_UPDATECP	更新当前位置
TA_RIGHT	右对齐	TA_NOUPDATECP	不更新当前位置

3.4.2 设置字体

Windows 支持光栅字体、矢量字体和 TrueType 共 3 种类型的字体。光栅字体即点阵字体，这种字体需要为每一种大小的字体创建独立的字体文件。矢量字体以一系列线段存储字符。TrueType 字体是与设备无关的字体，字符以轮廓的形式存储，包括线段和曲线。现在 TrueType 字体正成为主流字体，这种字体能够以一种非常出色的字体技术绘制文本，并且 TrueType 字体能够缩放为任何大小的字体，而不会降低图形的质量。

处理字体最简单的办法是使用系统提供的默认字体。如果需要，用户可以自己设置文本的字体。字体也是一种 GDI 对象，字体对象的创建、选择、使用和删除步骤与其他 GDI 对象类似。在 CFont 类提供的创建函数中常用的有以下几个。

1. CreatePointFont()

该函数仅含有 3 个参数，其原型为

```
BOOL CreatePointFont(int nPointSize,LPCTSTR lpszFaceName,CDC * pDC = NULL);
```

其中，第 1 个参数为字体大小，它以 1/10 磅为单位；第 2 个参数为创建字体对象所使用的字体名称；第 3 个参数 pDC 指向一个设备环境对象，当指针为空时，CreatePointFont()函数将字体大小以设备单位表示。

2. CreateFontIndirect()

该函数仅需一个参数，其原型为

```
BOOL CreateFontIndirect(const LOGFONT * lpLogFont);
```

其中，参数 lpLogFont 为指向 LOGFONT 结构的指针。LOGFONT 结构用来说明一种字体的所有属性，其定义如下：

```
typedef  struct  tagLOGFONT {
LONG lfHeight;
LONG lfWidth;
LONG lfEscapement;
LONG lfOrientation;
LONG lfWeight;
BYTE lfItalic;
BYTE lfUnderline;
BYTE lfStrikeOut;
BYTE lfCharSet;
BYTE lfOutPrecision;
BYTE lfClipPrecision;
BYTE lfQuality;
BYTE lfPitchAndFamily;
TCHAR lfFaceName[LF_FACESIZE];
}LOGFONT;
```

结构中各成员的含义见表 3.10。

表 3.10　LOGFONT 结构中各成员的含义

结构成员	含　义
lfHeight	以逻辑单位表示的字体高度，为 0 时采用系统默认值
lfWidth	以逻辑单位表示的字体平均宽度，为 0 时由系统根据高度取最佳值
lfEscapement	文本行相对页面底端的倾斜度，以 1/10°为单位
lfOrientation	字符基线相对页面底端的倾斜度，以 1/10°为单位
lfWeight	字体粗细，取 0～1000 之值，为 0 时使用默认值
lfItalic	为真时表示创建斜体字体
lfUnderline	为真时表示创建带下画线的字体
lfStrikeOut	为真时表示创建带删除线的字体
lfCharSet	指定字体所属字符集（ANSI _ CHARSET、OEM _ CHARSET、SYMBOL _ CHARSET、DEFAULT_CHARSET）
lfOutPrecision	指定字符输出精度（OUT_CHARACTER_PRECIS、OUT_DEFAULT_PRECIS、OUT_STRING_PRECIS、OUT_STROKE_PRECIS）
lfClipPrecision	指定裁剪精度（CLIP_CHARACTER_PRECIS、CLIP_ DEFAULT _PRECIS、CLIP_ STROKE _PRECIS）
lfQuality	指定输出质量，可选默认质量（DEFAULT_QUALITY）、草稿质量（DRAFT_ QUALITY）或正稿质量（PROOF_ QUALITY）
lfPitchAndFamily	指定字体间距和所属的字库族
lfFaceName	指定所用字体名，如果为 NULL，则使用默认字体名

3. CreateFont()

该函数包括大量参数，其原型为

```
BOOL CreateFont(
int nHeight;
int nWidth;
int nEscapement;
int nOrientation;
int nWeight;
BYTE bItalic;
BYTE bUnderline;
BYTE cStrikeOut;
BYTE nCharSet;
BYTE nOutPrecision;
BYTE nClipPrecision;
BYTE nQuality;
BYTE nPitchAndFamily;
LPCTSTR lpszFacename;
);
```

其参数的含义与 LOGFONT 结构完全一致。在调用该函数时，若参数为 0，表示使用系统默认值。

用户也可以选择表 3.11 中的库存字体来重新设置字体。当选择库存字体作为文本输出的字体时无须创建字体对象，只需简单地调用成员函数 CDC::SelectStockObject()将库存字体对象选入设备环境。

表 3.11　库存字体对象及说明

库存字体对象	说　明
ANSI_FIXED_FONT	ANSI 标准的等宽字体
ANSI_VAR_FONT	ANSI 标准的非等宽字体
SYSTEM_FONT	Windows 默认的非等宽系统字体
SYSTEM_FIXED_FONT	Windows 等宽系统字体
DEVICE_DEFAULT_FONT	当前设备字体
OEM_FIXED_FONT	与 OEM 相关的等宽字体

视频讲解

【例 3.8】 编写一个单文档的应用程序 Li3_8，采用不同方法创建字体，并根据创建的字体输出不同的文本。

其操作步骤如下：

(1) 打开解决方案资源管理器，使用“MFC 应用”项目模板在 chap03 解决方案中创建一个单文档的 MFC 应用程序项目 Li3_8，项目样式选择 MFC Standard。

(2) 在“类视图”中选择 Li3_8 项目的 CLi38View 类，打开其成员函数 OnDraw()。

(3) 在 OnDraw()函数中添加以下代码。

```
void CLi38View::OnDraw(CDC * pDC)
{
    CLi38Doc * pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if(!pDoc)
        return;
    //TODO: 在此处为本机数据添加绘制代码
    CString outstr[5];
    outstr[1] = "1.使用函数 CreatePointFont()创建宋体字";
    outstr[2] = "2.使用函数 CreateFontIndirect()创建倾斜、带下画线的黑体字";
    outstr[3] = "3.使用函数 CreateFont()创建带删除线的大号字";
    outstr[4] = "4.使用库存字体对象创建 ANSI 标准的等宽字";
    CFont * OldFont, NewFont;
    LOGFONT MyFont = {
            30,
            10,
            0,
            0,
            0,
            1,
            1,
            0,
            ANSI_CHARSET,
            OUT_DEFAULT_PRECIS,
            CLIP_DEFAULT_PRECIS,
            DEFAULT_QUALITY,
            DEFAULT_PITCH,
            L"黑体"
    };
    pDC->TextOut(0, 10, L"创建字体的几种方法: ");
```

```
    for(int i = 1; i < 5; i++)
    {
        switch(i)
        {
        case 1:
            //使用 CreatePointFont()函数创建字体
            NewFont.CreatePointFont(200, L"宋体", NULL);
            break;
        case 2:
            //使用 CreateFontIndirect()函数创建字体
            NewFont.CreateFontIndirect(&MyFont);
            break;
        case 3:
            //使用 CreateFont()函数创建字体
            NewFont.CreateFont(30, 10, 0, 0, FW_HEAVY, false, false,
                true, ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                DEFAULT_PITCH | FF_DONTCARE, L"大号字");
            break;
        case 4:
            //使用库存字体对象创建字体
            pDC->SelectStockObject(ANSI_FIXED_FONT);
        }
        OldFont = pDC->SelectObject(&NewFont);
        pDC->TextOut(0, 60 * i, outstr[i]);
        pDC->SelectObject(OldFont);
        NewFont.DeleteObject();
    }
}
```

(4) 编译、链接并运行程序,结果如图 3.10 所示。

图 3.10　创建字体的几种方法

3.4.3　格式化文本

Windows 系统不参与窗口客户区的管理,这就意味着在客户区内输出文本时必须由应用程序管理换行、后继字符的位置等输出格式。文本的显示以像素为单位,因此在绘制任何文本之前需要精确地知道文本的详细属性,例如高度、宽度等,用来计算文本的坐标。CDC

类提供了几个文本测量成员函数，下面分别介绍。

1. GetTextExtent()

使用该函数可以获得所选字体中指定字符串的宽度和高度，该函数的原型为

```
CSize GetTextExtent(LPCTSTR lpszString,int nCount);
```

其中，lpszString 是字符串的指针，nCount 是所包括的字符数。返回值 CSize 是包含两个成员的结构，cx 是字符串的宽度，cy 是字符串的高度。

2. GetTextMetrics()

调用 GetTextMetric()函数可以获得当前字体的 TEXTMETRIC 结构数据，该函数的原型为

```
BOOL GetTextMetric(const TEXTMETRIC * lpTextMetric);
```

其中，参数 lpTextMetric 为指向结构 TEXTMETRIC 的指针。TEXTMETRIC 结构用来描述字体信息，其定义如下：

```
typedef  struct  tagTEXTMETRIC{
   int   tmHeight;                          //字符的高度
   int   tmAscent;                          //字符基线以上的高度
   int   tmDescent;                         //字符基线以下的高度
   int   tmInternalLeading;                 //字符高度的内部间距
   int   tmExternalLeading;                 //两行之间的间距(外部间距)
   int   tmAveCharWidth;                    //字符的平均宽度
   int   tmMaxCharWidth;                    //字符的最大宽度
   int   tmWeight;                          //字体的粗细
   BYTE   tmItalic;                         //指明斜体,零值表示非斜体
   BYTE tmUnderlined;                       //指明下画线,零值表示不带下画线
   BYTE tmStruckOut;                        //指明删除线,零值表示不带删除线
   BYTE tmFirstChar;                        //字体中第一个字符的值
   BYTE tmLastChar;                         //字体中最后一个字符的值
   BYTE tmDefaultChar;                      //字库中所没有字符的替代字符
   BYTE tmBreakChar;                        //文本对齐时作为分隔符的字符
   BYTE tmPitchAndFamily;                   //给出所选字体的间距和所属的字库族
   BYTE tmCharSet;                          //字体的字符集
   int  tmOverhang;                         //合成字体(如斜体和黑体)的附加宽度
   int  tmDigitizedAspectX;                 //设计字体时的横向比例
   int  tmDigitizedAspectY;                 //设计字体时的纵向比例
} TEXTMETRIC;
```

视频讲解

【例 3.9】 编写一个单文档的应用程序 Li3_9，采用不同的格式输出文本串。

其操作步骤如下：

(1) 打开解决方案资源管理器，使用“MFC 应用”项目模板在 chap03 解决方案中创建一个单文档的 MFC 应用程序项目 Li3_9，项目样式选择 MFC Standard。

(2) 在“类视图”中选择 Li3_9 项目的 CLi39View 类，打开其成员函数 OnDraw()。

(3) 在 OnDraw()函数中添加以下代码。

```
int ExternalLeading, y = 0;
TEXTMETRIC tm;
```

```
CString c, outString = L"Visual C++";
CFont * OldFont, NewFont;
CSize size;
LOGFONT MyFont = {
        30,
        10,
        0,
        0,
        0,
        0,
        0,
        0,
        ANSI_CHARSET,
        OUT_DEFAULT_PRECIS,
        CLIP_DEFAULT_PRECIS,
        DEFAULT_QUALITY,
        DEFAULT_PITCH,
        L"黑体"
};
for(int i = 0; i < outString.GetLength(); i++)
{
    MyFont.lfHeight = 30 + 10 * i;              //设置字符高度
    NewFont.CreateFontIndirect(&MyFont);        //创建字体
    OldFont = pDC->SelectObject(&NewFont);
    pDC->GetTextMetrics(&tm);                   //获得当前字体的 TEXTMETRIC 结构数据
    ExternalLeading = tm.tmExternalLeading;     //获得行间距值
    size = pDC->GetTextExtent(outString, i);    //获得字符串的宽度和高度值
    y = 10 + y + (int)(0.1 * size.cy) + ExternalLeading;    //计算纵坐标值
    c.Format(L"%c", outString.GetAt(i));
    pDC->TextOut(200 + size.cx, y, c);          //输出字符
    pDC->SelectObject(OldFont);
    NewFont.DeleteObject();
}
```

(4) 编译、链接并运行程序,结果如图 3.11 所示。

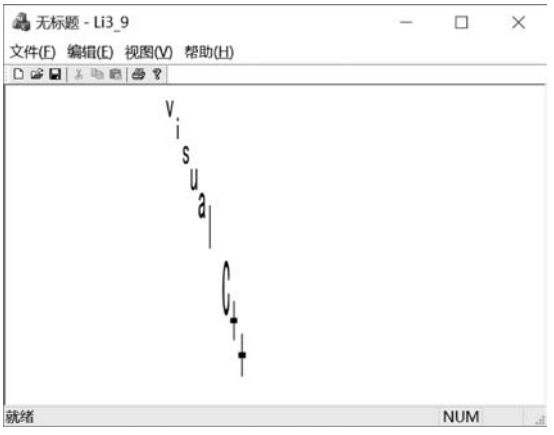

图 3.11 文本的格式化

3.4.4 常用的文本输出函数

MFC CDC 类中常用的文本输出函数有以下几种。

1. TextOut()

该函数使用当前设定的字体、颜色、对齐方式在指定位置上输出文本。该函数的原型为

```
virtual BOOL TextOut(int x,int y,LPCTSTR lpszString,int nCount);
```

或

```
BOOL TextOut(int x,int y,const Cstring &str);
```

其中,参数(x,y)指定输出文本字符串的开始位置;参数 lpszString 和 str 为输出的文本字符串;参数 nCount 指定文本字符串的长度。

2. DrawText()

该函数在给定的矩形区域内输出文本,并可调整文本在矩形区域内的对齐方式以及对文本行进行换行处理等。该函数的原型为

```
Virtual int DrawText(LPCTSTR lpszString,int nCount,LPRECT lpRect,UINT nFormat);
```

或

```
int DrawText(const CString&str,LPRECT lpRect,UINT nFormat);
```

其中,参数 lpszString 和 str 为要输出的文本字符串,可以使用换行符"\n";参数 nCount 指定文本字符串的长度;参数 lpRect 指定用于显示文本字符串的矩形区域;参数 nFormat 指定如何格式化文本字符串。

3. ExtTextOut()

该函数的功能与 TextOut()相似,但可以根据指定的矩形区域裁剪文本字符串,并调整字符间距。该函数的原型为

```
Virtual BOOL ExtTextOut(int x,int y,UINT nOptions,LPCRECT lpRect,
                    LPCTSTR lpszString,UINT nCount,LPINT lpDxWidths);
```

或

```
BOOL ExtTextOut(int x,int y,UINT nOptions,LPCRECT lpRect,
                    const CString &str,LPINT lpDxWidths);
```

其中,参数(x,y)给出指定输出文本字符串的坐标;参数 nOptions 用于指定裁剪类型,可以为 ETO_CLIPPED(裁剪文本以适应矩形)或 ETO_OPAQUE(用当前背景颜色填充矩形);参数 lpRect 用于指定裁剪的矩形;参数 lpszString 和 str 为要输出的文本串;参数 nCount 为要输出的字符数;参数 lpDxWidths 为字符间距数组,若该参数为 NULL,则使用默认间距。

4. TabbedTextOut()

该函数的功能与 TextOut()相似,但可按指定的制表间距扩展制表符。该函数的原型为

```
Virtual CSize TabbedTextOut(int x,int y,LPCTSTR lpszString, int nCount,
                int nTabPositions,LPINT lpnTabStopPositions,int nTabOrigin);
```

或

```
CSize TabbedTextOut(int x,int y,const CString&str,int nTabPositions,
                                    LPINT lpnTabStopPositions,int nTabOrigin);
```

其中,参数(x,y)指定输出文本字符串的开始位置;参数 lpszString 和 str 为要输出的文本字符串;参数 nCount 指定文本字符串的长度;参数 nTabPositions、lpnTabStopPositions 和 nTabOrigin 用于指定制表间距。


视频讲解

【例 3.10】 编写一个单文档的应用程序 Li3_10,采用不同的函数输出文本串。

其操作步骤如下:

(1) 打开解决方案资源管理器,使用"MFC 应用"项目模板在 chap03 解决方案中创建一个单文档的 MFC 应用程序项目 Li3_10,项目样式选择 MFC Standard。

(2) 在"类视图"中选择 Li3_10 项目的 CLi310View 类,打开其成员函数 OnDraw()。

(3) 在 OnDraw()函数中添加以下代码。

```
void CLi310View::OnDraw(CDC * pDC)
{
    CLi310Doc * pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if(!pDoc)
        return;
    //TODO: 在此处为本机数据添加绘制代码
    CString outstr[5];
    CRect rect;
    outstr[1] = "使用 TextOut()函数输出文本";
    outstr[2] = "使用 DrawText()函数输出文本";
    outstr[3] = "使用 ExtTextOut()函数输出文本";
    outstr[4] = "使用 TabbedTextOut()函数输出文本";
    //使用 TextOut()函数输出文本
    pDC->TextOut(10, 10, L"常用文本输出函数: ");
    pDC->TextOut(50, 30, outstr[1]);
    pDC->SetBkColor(RGB(255, 255, 0));
    //使用 DrawText()函数输出文本
    rect.SetRect(CPoint(50, 60), CPoint(250, 110));
    pDC->DrawText(outstr[2], &rect, DT_WORDBREAK | DT_CENTER);
    //使用 ExtTextOut( )函数输出文本
    rect.SetRect(CPoint(50, 100), CPoint(200, 200));
    pDC->ExtTextOut(50, 120, ETO_CLIPPED, &rect, outstr[3], NULL);
}
```

(4) 编译、链接并运行程序,结果如图 3.12 所示。

图 3.12 文本的输出

3.5 位　　图

位图是由位构成的图像，它是由一系列 0 和 1 排列而成的点阵结构。位图中的每一个像素点由位图文件中的一位或者多位数据表示，整个位图的信息被细化为每个像素的属性值。Windows 支持两种不同形式的位图，即设备相关位图(Device Dependent Bitmap，DDB)和设备无关位图(Device Independent Bitmap，DIB)。

3.5.1 设备相关位图和设备无关位图

DDB 又称 GDI 位图，它依赖于具体设备，只能存在于内存中。这主要体现在以下两个方面：一方面，DDB 的颜色模式必须与输出设备相一致；另一方面，在 256 色以下位图中存储的像素值是系统调色板的索引，其颜色依赖于系统调色板。因此，当在一台计算机上创建的位图在另一台计算机上显示时可能会出现问题。

DIB 是不依赖硬件的位图，因为它包含了创建 DIB 时所在设备的颜色格式、分辨率和调色板等信息。DIB 不能直接显示，需要先转换为与设备相关的格式，再由具体的设备显示。DIB 通常以.bmp 扩展名的文件形式存储在磁盘中，或者以资源的形式存在于 EXE 或 DLL 文件中。

3.5.2 位图操作函数

MFC 提供了大量的类和函数来处理位图。

1. 创建 DDB 函数

创建 DDB 函数的原型为

```
BOOL LoadBitmap(LPCTSTR lpszResourceName);
BOOL LoadBitmap(UINT nlDResource);
```

其中，参数 lpszResourceName 或 nlDResource 分别为位图资源名称或位图资源标识。该函数从资源中载入一幅位图，若载入成功，返回值为真，否则返回值为假。位图资源实际上是一个 DIB，该函数在载入时把它转换成了 DDB。

```
BOOL CreateCompatibleBitmap(CDC * pDC, int nWidth, int nHeight);
```

其中，参数 pDC 指向一个设备环境，参数 nWidth 和 nHeight 以像素为单位，用来指定位图的宽度和高度。该函数创建一个与指定设备环境兼容的 DDB。若创建成功，函数的返回值为真，否则为假。

2. 获取位图信息函数

获取位图信息函数的原型为

```
int GetBitmap(BITMAP * pBitMap);
```

该函数用来获取与 DDB 有关的信息，参数 pBitMap 是 BITMAP 结构的指针。BITMAP 结构的定义为

```
typedef struct tabBITMAP{
```

```
LONG bmType;                    //必须为 0
LONG bmWidth;                   //位图的宽度(以像素为单位)
LONG bmHeight;                  //位图的高度(以像素为单位)
LONG bmWidthBytes;              //每一扫描行所需的字节数,应为偶数
WORD bmPlanes;                  //色平面数
WORD bmBitsPixel;               //像素位数
LPVOID bmBits;                  //位图位置的地址
}BITMAP;
```

3. 输出位图函数

输出位图函数的原型为

```
BOOL BitBlt(int x,int y,int nWidth,int nHeight,CDC * pSrcDC,int xSrc,
                                        int ySrc,DWORD dwRop);
```

该函数共有 8 个参数,其中 x、y、nWidth、nHeight 定义当前设备环境的复制区域;pSrcDC 为指向原设备环境对象的指针;xSrc、ySrc 为原位图的左上角坐标;dwRop 定义了进行复制时的光栅操作方式,其取值如表 3.12 所示。该函数把源设备环境中的位图复制到目标设备环境中。

表 3.12 dwRop 参数的取值

参 数	含 义
BLACKNESS	将所有输出变黑色
DSTINVERT	反转目标位图
MERGECOPY	合并模式和原位图
MERGEPAINT	用或(or)运算合并反转的原位图和目标位图
NOTSRCCOPY	将反转的原位图复制到目标位图
NOTSRCERASE	用或(or)运算合并原位图和目标位图,然后反转
PATCOPY	将模式复制到目标位图
PATINVERT	用异或(xor)运算合并目标位图与模式
PATPAINT	用或(or)运算合并反转的原位图与模式,然后用或(or)运算合并上述结果与目标位图
SRCAND	用与(and)运算合并目标像素与原位图
SRCCOPY	将原位图复制到目标位图
SRCERASE	反转目标位图并用与(and)运算合并所得结果与原位图
SRCINVERT	用异或(xor)运算合并目标像素与原位图
SRCPAINT	用或(or)运算合并目标像素与原位图
WHITENESS	将所有输出变白色

有时需要相对位图进行放大或缩小的操作,这时就可以使用 StretchBlt()函数来显示位图,该函数的原型为

```
BOOL StretchBlt(int x,int y,int nWidth,int nHeight,CDC * pSrcDC,int xSrc,int ySrc,
                                int nSrcWidth,int nSrcHeight,DWORD dwRop);
```

其中,除了参数 nSrcWidth 和 nSrcHeight 表示目标图像的新的宽度与高度之外,其他参数 x、y、nWidth 和 nHeight 等的含义与 BitBlt()函数中的同名参数相同。该函数提供了将图

形拉伸、压缩的复制方式。

3.5.3 位图的显示

在采用 MFC 方法编程时，显示一个 DDB 需要执行以下几个步骤。

(1) 声明一个 CBitmap 类的对象，使用 LoadBitmap()函数将位图装入内存。

(2) 声明一个 CDC 类的对象，使用 CreateCompatibleDC()函数创建一个与显示设备环境兼容的内存设备环境。CreateCompatibleDC()函数的原型为

```
Virtual BOOL CreateCompatibleDC(CDC * pDC);
```

其中，参数 pDC 是指向设备环境的指针。如果 pDC 为 NULL，则创建与系统显示器兼容的内存设备环境。

(3) 使用 CDC::SelectObject()函数将位图对象选入设备环境中，并保存原来设备环境的指针。

(4) 使用 CDC 的相关输出函数输出位图。

(5) 使用 CDC::SelectObject()函数恢复原来设备环境。

视频讲解

【例 3.11】 编写一个单文档的应用程序 Li3_11，在视图中按原有大小显示一幅位图及它的两个缩小图像。

其操作步骤如下：

(1) 打开解决方案资源管理器，使用“MFC 应用”项目模板在 chap03 解决方案中创建一个单文档的 MFC 应用程序项目 Li3_11，项目样式选择 MFC Standard。

(2) 添加位图资源。在解决方案资源管理器中选中 Li3_11 项目，选择“项目”|“添加资源”菜单命令，打开“添加资源”对话框。在该对话框中选择 Bitmap，单击“导入”按钮，打开文件对话框选择准备插入的位图文件，然后单击“打开”按钮完成位图资源的添加。

(3) 在“类视图”中选择 Li3_11 项目的 CLi311View 类，打开其成员函数 OnDraw()。

(4) 在 OnDraw()函数中添加以下代码。

```
void CLi311View::OnDraw(CDC * pDC)
{
    CLi311Doc * pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if(!pDoc)
        return;
    //TODO: 在此处为本机数据添加绘制代码
    CBitmap Bitmap;
    Bitmap.LoadBitmap(IDB_BITMAP1);                          //将位图装入内存
    CDC MemDC;
    MemDC.CreateCompatibleDC(pDC);                           //创建内存设备环境
    CBitmap * OldBitmap = MemDC.SelectObject(&Bitmap);
    BITMAP bm;                                               //创建 BITMAP 结构变量
    Bitmap.GetBitmap(&bm);                                   //获取位图信息
    //显示位图
    //将内存设备环境复制到真正的设备环境中
    pDC->BitBlt(0, 0, bm.bmWidth, bm.bmHeight, &MemDC, 0, 0, SRCCOPY);
    //缩小一半显示
```

```
    pDC->StretchBlt(bm.bmWidth, 0, bm.bmWidth / 2, bm.bmHeight / 2, &MemDC, 10, 0, bm.
bmWidth, bm.bmHeight, SRCCOPY);
    //缩小一半显示
    pDC->StretchBlt(bm.bmWidth, bm.bmHeight / 2, bm.bmWidth / 2, bm.bmHeight / 2, &MemDC,
10, 0, bm.bmWidth, bm.bmHeight, SRCCOPY);
    pDC->SelectObject(OldBitmap);                          //恢复设备环境
}
```

(5) 编译、链接并运行程序,结果如图 3.13 所示。

图 3.13　位图的显示

3.6　应 用 实 例

视频讲解

3.6.1　实例简介

编写一个应用程序,输出“空心文本”“阴影文本”和“位图文本”3 组具有特殊效果的文本。“空心文本”使用红色、实线画笔绘制,“阴影文本”使用 45°十字交叉阴影线画刷绘制,“位图文本”使用位图画刷绘制。效果如图 3.14 所示。

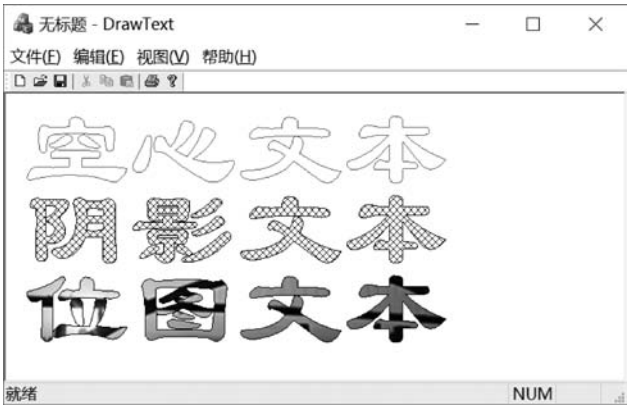

图 3.14　画笔及画刷的使用

3.6.2 创建过程

1. 创建 MFC 应用程序框架

启动 Visual Studio IDE 并打开解决方案资源管理器，在 chap03 解决方案中使用“MFC 应用”项目模板创建一个名为 DrawText 的单文档 MFC 应用程序，项目样式选择 MFC Standard。

2. 插入位图资源

由于在程序中使用了位图画刷，所以首先必须为应用程序添加位图资源。在解决方案资源管理器中选中 DrawText 项目，选择“项目”|“添加资源”菜单命令，打开“添加资源”对话框。在该对话框中选择 Bitmap，单击“导入”按钮，打开文件对话框选择准备插入的位图文件，然后单击“打开”按钮完成位图资源的添加。接受系统默认新位图资源的标识 IDB_BITMAP1。

3. 添加代码

在“类视图”中选择 DrawText 项目的 CDrawTextView 类，双击其成员函数 OnDraw()，在函数体中添加以下代码。

```
void CDrawTextView::OnDraw(CDC * pDC)
{
    CDrawTextDoc * pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if(!pDoc)
        return;
    //TODO: 在此处为本机数据添加绘制代码
    LOGFONT lf;
    CFont NewFont, *OldFont;
    CPen NewPen, *OldPen;
    CBrush NewBrush, *OldBrush;
    //获取原有字体参数,并进行修改
    pDC->GetCurrentFont()->GetLogFont(&lf);
    /*GetCurrentFont()返回指向当前设备上下文所使用的字体指针,GetLogFont()将当前字体的
信息填入 lf 中*/
    lf.lfCharSet = DEFAULT_CHARSET;
    lf.lfHeight = -120;
    lf.lfWidth = 0;
    ::lstrcpy(lf.lfFaceName, L"隶书");        //不能对 lfFaceName 直接赋值
    //设置字体
    NewFont.CreateFontIndirect(&lf);
    OldFont = pDC->SelectObject(&NewFont);
    pDC->SetBkMode(TRANSPARENT);              //透明背景模式
    for(int i = 1; i <= 3; i++)
    {
        switch(i)
        {
        case 1:
            //创建画笔
            NewPen.CreatePen(PS_SOLID, 1, RGB(255, 0, 0));
```

```
            OldPen = pDC->SelectObject(&NewPen);
            //开始一个路径
            pDC->BeginPath();
            pDC->TextOut(20, 0, L"空心文本");
            pDC->EndPath();
            //绘制路径
            pDC->StrokePath();
            //释放资源
            NewPen.DeleteObject();
            //恢复设备环境的原有设置
            pDC->SelectObject(OldPen);
            break;
        case 2:
            //创建阴影画刷
            NewBrush.CreateHatchBrush(HS_DIAGCROSS, RGB(0, 0, 255));
            OldBrush = pDC->SelectObject(&NewBrush);
            //开始一个路径
            pDC->BeginPath();
            pDC->TextOut(20, 90, L"阴影文本");
            pDC->EndPath();
            //绘制路径
            pDC->StrokeAndFillPath();
            //释放资源
            NewBrush.DeleteObject();
            //恢复设备环境的原有设置
            pDC->SelectObject(OldBrush);
            break;
        case 3:
            CBitmap bitmap;
            //装入位图
            bitmap.LoadBitmap(IDB_BITMAP1);
            //创建位图画刷
            NewBrush.CreatePatternBrush(&bitmap);
            OldBrush = pDC->SelectObject(&NewBrush);
            pDC->BeginPath();
            pDC->TextOut(20, 180, L"位图文本");
            pDC->EndPath();
            pDC->StrokeAndFillPath();
            //释放资源
            NewBrush.DeleteObject();
            //恢复设备环境的原有设置
            pDC->SelectObject(OldBrush);
        }
    }
    //恢复设备环境的原有设置
    pDC->SelectObject(OldFont);
}
```

在上述代码中使用了 CDC 类的 4 个成员函数，即 BeginPath()、EndPath()、StrokePath() 和 StrokeAndFillPath()，它们的作用分别是开始一个路径、结束一个路径、绘制路径以及绘

制并填充路径所包含的区域。

编译、链接并运行程序，结果如图 3.14 所示。

习　　题

1. 填空题

(1) Windows 引入 GDI 的主要目的是实现_____。

(2) 与 DOS 下输出文本的方式不同，Windows 中的文本是按_____方式输出的。

(3) 为了体现 Windows 的设备无关性，应用程序的输出面向一个称为_____的虚拟逻辑设备。

(4) 在一个 MFC 应用程序中获得 DC 的方法主要有两种：一种是_____；另一种是_____，并使用_____指针为该对象赋值。

(5) Windows 用_____类型的数据存放颜色，它实际上是一个_____位整数。它采用 3 个参数表示红、绿、蓝分量值，这 3 个值的取值范围为_____。

(6) 库存对象是由操作系统维护的用于绘制屏幕的常用对象，包括库存_____等。

(7) 可以利用 CGdiObject 类的成员函数_____将 GDI 对象设置成指定的库存对象。

(8) 在创建画笔后必须调用 CDC 类的成员函数_____将创建的画笔选入当前设备环境。

(9) 在默认情况下输出文本时，字体颜色是_____，背景颜色是_____，背景模式为_____。

(10) 创建画笔的方法有两种：一种是_____；另一种是_____。

2. 选择题

(1) 下面(　　)不是 MFC 设备环境类 CDC 的派生类。

A. GDI 类　　B. CPaintDC 类　　C. CClientDC 类　　D. CWindowDC 类

(2) 下面(　　)不是 GDI 对象的类。

A. CFont 类　　B. CPalette 类　　C. CClientDC 类　　D. CBitmap 类

(3) 下列描述中，(　　)是错误的。

A. CreatePointFont()是 CFont 类提供的创建函数

B. 可使用 SetTextAlign()函数改变文本的对齐方式

C. 使用 GetTextMetrics()函数可以获得所选字体中指定字符串的宽度和高度

D. 可使用 DrawText()函数在给定的矩形区域内输出文本

(4) 下面(　　)不是 CDC 类中常用的文本输出函数。

A. TextOut()　　B. DrawText()

C. ExtTextOut()　　D. ExtDrawText()

3. 判断题

(1) CDC 类是 MFC 设备环境类的基类。(　　)

(2) CClientDC 代表整个窗口的设备环境。(　　)

(3) CPen 和 CFont 均是 GDI 对象。(　　)

(4) 深绿色的 RGB 值为(0,128,0)。(　　)

(5) 删除 CPen 对象可调用 CPen 对象的 DeleteObject()函数。 ( )

(6) 创建阴影画刷的函数是 CreateHatchBrush()。 ( )

(7) 默认的对齐方式是 TA_LEFT| TA_BOTTOM。 ( )

(8) DDB 又称 GDI 位图,它依赖于具体设备,只能存在于内存中。 ( )

4. 简答题

(1) GDI 创建哪几种类型的图形输出?

(2) 什么是设备环境? 它的主要功能有哪些?

(3) 什么是 GDI? 它有什么功能? MFC 将 GDI 函数封装在哪个类中?

(4) 请简述设备无关性的含义,说明实现设备无关性需要的环节。

(5) MFC 提供了哪几种设备环境类? 它们各有什么用途?

(6) 简述传统的 SDK 获取设备环境的方法。

(7) 简述创建和使用自定义画笔的步骤。

(8) 简述采用 MFC 方法编程时显示一个 DDB 位图的步骤。

5. 操作题

(1) 编写程序在客户区中显示一行文本,要求文本颜色为红色、背景颜色为黄色。

(2) 编写一个单文档应用程序,在客户区使用不同的画笔和画刷绘制点、折线、曲线、圆角矩形、弧、扇形和多边形等几何图形。

(3) 编程利用函数 CreateFontIndirect()创建黑体字体,字体高度为 30 像素、宽度为 25 像素,并利用函数 DrawText()在客户区以该字体输出文本。

(4) 编写一个单文档应用程序,在视图窗口中显示 3 个圆,通过使用不同颜色的画笔及画刷来模拟交通红绿灯。

(5) 编写一个程序,实现一行文本的水平滚动显示。要求每个周期中文本为红、黄两种颜色,字体为宋、楷两种字体。

提示: 操作题(4)、(5)需要利用 WM_TIMER 消息,请参考本书的例 4.10。