

序言.....	- 4 -
目标.....	- 4 -
适用范围.....	- 4 -
适用读者.....	- 4 -
运行环境.....	- 4 -
文档组织.....	- 4 -
相关主题.....	- 4 -
GDI+的安全考虑.....	- 6 -
检验构造函数调用成功与否.....	- 6 -
分配缓冲区.....	- 6 -
错误校验.....	- 7 -
线程同步.....	- 9 -
相关主题.....	- 9 -
关于 GDI+.....	- 10 -
GDI+ 介绍.....	- 11 -
GDI+ 概览.....	- 11 -
GDI+ 的三个组成部分.....	- 11 -
基于类的接口架构.....	- 12 -
GDI+提供了哪些新东西? .....	- 12 -
新特征.....	- 12 -
编程模式的改变.....	- 15 -
线条、曲线和图形.....	- 19 -
矢量图概览.....	- 19 -
钢笔、线条和矩形.....	- 20 -
椭圆和弧.....	- 22 -
多边形.....	- 22 -
基数样条.....	- 23 -
贝塞尔样条.....	- 24 -
路径.....	- 25 -
画刷和填充图形.....	- 27 -
开放与闭合曲线.....	- 29 -
区域.....	- 30 -
裁剪.....	- 31 -
路径平直化.....	- 32 -
线条和曲线的抗锯齿功能.....	- 32 -
图象、位图和图元文件.....	- 33 -
位图类型.....	- 33 -
图元文件.....	- 37 -
绘制、定位和复制图片.....	- 38 -
裁剪和缩放图象.....	- 40 -
坐标系统和转换.....	- 42 -
坐标系统类型.....	- 42 -

以矩阵来表示转换.....	- 44 -
全局和局部转换.....	- 48 -
图形容器.....	- 51 -
使用 GDI+.....	- 55 -
使用入门.....	- 56 -
绘制线条.....	- 56 -
绘制字符串.....	- 58 -
使用钢笔绘制线条和形状.....	- 59 -
使用钢笔绘制线条和矩形.....	- 59 -
设置钢笔的宽度和对齐方式.....	- 60 -
绘制具有线帽的线条.....	- 61 -
联接线条.....	- 62 -
绘制自定义虚线.....	- 62 -
绘制用纹理填充的线条.....	- 63 -
使用画笔填充形状.....	- 63 -
用纯色填充形状.....	- 63 -
用阴影图案填充形状.....	- 64 -
用图像纹理填充形状.....	- 64 -
在形状中平铺图像.....	- 65 -
用渐变色填充形状.....	- 67 -
使用图像、位图和图元文件.....	- 68 -
加载和显示位图.....	- 68 -
加载和显示图元文件.....	- 69 -
记录图元文件.....	- 69 -
剪裁和缩放图像.....	- 71 -
旋转、反射和扭曲图像.....	- 72 -
缩放时使用插值模式控制图像质量.....	- 73 -
创建缩略图像.....	- 75 -
采用高速缓存位图来提高性能.....	- 76 -
通过避免自动缩放改善性能.....	- 76 -
读取图像元数据.....	- 77 -
使用图像编码器和解码器.....	- 83 -
列出已安装的编码器.....	- 83 -
列出已安装的解码器.....	- 84 -
获取解码器的类标识符.....	- 86 -
获取编码器的参数列表.....	- 88 -
将 BMP 图像转换为 PNG 图像.....	- 100 -
设定 JPEG 的压缩等级.....	- 100 -
对 JPEG 图像进行无损变换.....	- 102 -
创建和保存多帧图像.....	- 105 -
从多帧图像中复制单帧.....	- 107 -
Alpha 混合线条和填充.....	- 109 -
绘制不透明和半透明的线条.....	- 109 -
用不透明和半透明的画笔绘制.....	- 110 -

使用复合模式控制 Alpha 混合.....	- 110 -
使用颜色矩阵设置图像中的 Alpha 值.....	- 112 -
设置单个像素的 alpha 值.....	- 114 -
使用字体和文本.....	- 115 -
构造字体系列和字体.....	- 115 -
绘制文本.....	- 116 -
格式化文本.....	- 117 -
枚举已安装的字体.....	- 120 -
创建专用的字体集合.....	- 122 -
获取字体规格.....	- 126 -
对文本使用消除锯齿效果.....	- 130 -
构造并绘制曲线.....	- 131 -
绘制基数样条曲线.....	- 131 -
绘制贝塞尔样条.....	- 132 -
用渐变画刷填充形状.....	- 134 -
创建线性渐变.....	- 134 -
创建路径渐变.....	- 137 -
将 Gamma 校正应用于渐变.....	- 143 -
构造并绘制路径.....	- 145 -
使用线条、曲线和形状创建图形.....	- 145 -
填充开放式图形.....	- 147 -
使用图形容器.....	- 147 -
管理 Graphics 对象的状态.....	- 148 -
使用嵌套的 Graphics 容器.....	- 151 -
变换.....	- 154 -
使用世界变换.....	- 154 -
为什么变换顺序非常重要.....	- 155 -
使用区域.....	- 156 -
对区域使用点击检测.....	- 156 -
对区域使用剪辑.....	- 157 -
对图像重新着色.....	- 158 -
使用颜色矩阵对单色进行变换.....	- 158 -
转换图像颜色.....	- 160 -
缩放颜色.....	- 161 -
旋转颜色.....	- 164 -
剪取颜色.....	- 166 -
使用颜色重映射表.....	- 168 -
打印.....	- 169 -
将 GDI+ 输出至打印机.....	- 169 -
显示一个打印对话框.....	- 172 -
通过提供打印机句柄优化打印.....	- 173 -
附录：GDI+ 参考.....	- 175 -

# 序言

## 目标

Microsoft Windows GDI+是为 C/C++ 开发者提供的一个基于类的应用程序编程接口（API）。它使得程序可以同时视频显示器和打印机上使用图形和格式化文本。基于 Microsoft Win32 的应用程序不能直接访问图形硬件，而是通过 GDI+ 来协调设备驱动和程序动作的交互。GDI+ 同样支持 Microsoft Win64。

## 适用范围

GDI 适用于所有的 Window 应用程序。GDI+ 是包含于 Windows XP 和 Windows Server 2003 中的一项新技术。运行于 Microsoft Windows NT 4.0 SP6、Windows 2000、Windows 98 和 Windows Me 操作系统的应用程序在分发的时候需要包含 GDI+。

## 适用读者

GDI+ C++ 基类接口是为 C/C++ 开发者所设计。需要精通 Windows 图形用户接口和消息驱动机制。

## 运行环境

Gdiplus.dll 包含在 Windows XP 中。某些特定的类和方法可能需要特定操作系统的支持，请参阅文档中相应的类和方法。GDI+ 可以在 Windows NT 4.0 SP6、Windows 2000、Windows 98 以及 Windows Me 操作系统中重新分发部署。下载最新的可重新分发安装包，参见：

<http://www.microsoft.com/msdownload/platformsdk/sdkupdate/psdkredist.htm> 

**注意：**如果您将 GDI+ 分发到低版本系统平台或者该平台本身不包含该版本的 GDI+，则将 Gdiplus.dll 安装到您的应用程序目录下。这样将它放到了您自己的地址空间下，但是您应该使用 /BASE 连接器选项重构基地址，避免地址空间冲突。

## 文档组织

GDI+ 参考文档采用下面表格所示进行组织：

概览	关于 GDI+ 的概要信息。
用法	使用 GDI+ 的任务和例子。
参考	GDI+ C++ 基类 API 参考文档。


## 相关主题



[Windows GDI](#)



[DirectX](#)

-  [Windows Image Acquisition](#)
-  [OpenGL](#)
-  [Windows Multimedia](#)

## GDI+的安全考虑

本主题提供关于采用 Microsoft Windows GDI+进行开发的安全考虑。本主题没有提供所有您想了解的安全性相关主题——取而代之，我们将它作为本技术领域的出发点和基准。

### 检验构造函数调用成功与否

很多 GDI+类提供一个 `Image::GetLastStatus` 方法，您可以调用它检测某个对象的方法调用是否成功。您也可以调用 **GetLastStatus** 来判断构造函数是否成功调用。

下面的例子展示了如何构造一个 `Image` 对象并且调用 **GetLastStatus** 方法来判断构造成功与否。返回值 **OK** 和 **InvalidParameter** 是 `Status` 枚举的成员。

```
Image myImage(L"Climber.jpg");
Status st = myImage.GetLastStatus();

if(Ok == st)
    // The constructor was successful. Use myImage.
else if(InvalidParameter == st)
    // The constructor failed because of an invalid parameter.
else
    // Compare st to other elements of the Status
    // enumeration or do general error processing.
```

### 分配缓冲区

有几个 GDI+方法采用调用者分配的缓冲区来返回数字或者字符数据。其中每个方法都有一个关联方法来提供所需缓冲区的大小。例如，`GraphicsPath::GetPathPoints` 方法返回一个 `Point` 对象数组。在您调用 **GetPathPoints** 之前，您必须分配足够大的缓冲区来容纳这个数组。而您可以调用 `GraphicsPath` 对象的 **GetPointCount** 方法来计算所需缓冲区的大小。

下面的例子展示了如何判断一个 `GraphicsPath` 对象的点数，并分配足够大的缓冲区来容纳这些点，然后调用 **GetPathPoints** 方法来填充该缓冲区。在调用 **GetPathPoints** 的代码之前，需要检测缓冲区分配是否成功，方法是看缓冲区指针是否不等于 `NULL`。

```
GraphicsPath path;
path.AddEllipse(10, 10, 200, 100);

INT count = path.GetPointCount();           // get the size
Point* pointArray = new Point[count];       // allocate the buffer

if(pointArray) // Check for successful allocation.
{
```

```

path.GetPathPoints(pointArray, count); // get the data
...                                  // use pointArray
delete[] pointArray;                 // release the buffer
pointArray = NULL;
}

```

前面的例子采用 **New** 操作符分配缓冲区。采用 **New** 操作符非常方便，因为缓冲区是由明确类型的 **Point** 对象所构成。有些时候，GDI+ 写入比 GDI+ 对象数组更多的信息到缓冲区。有时缓冲区在填入 GDI+ 对象数组的同时还通过这些对象的成员来指向额外的数据信息。例如，**Image::GetAllPropertyItems** 方法返回一个 **PropertyItem** 对象数组，每个属性项目（一段元数据）都存储在图像中。但是 **GetAllPropertyItems** 返回的信息不止这些对象数组；它还为此数组追加了额外的数据。

在您调用 **GetAllPropertyItems** 之前，您必须为对象数组以及这些附加数据分配足够大的空间。您可以调用 **Image** 对象的 **GetPropertySize** 方法来判断所需缓冲区的总大小。

下面的例子展示了如何创建一个 **Image** 对象，然后调用该对象的 **GetAllPropertyItems** 方法来获取所有存储于图像中的属性项目（元数据）。该代码采用 **GetPropertySize** 方法返回的大小值来分配缓冲区。**GetPropertySize** 同时返回一个数值，表示该 **Image** 中的属性项目的数量。需要注意的是，这里没有采用 `count*sizeof(PropertyItem)` 来计算缓冲器大小。显然，采用这种方法计算出来的值太小了。

```

UINT count = 0;
UINT size = 0;
Image myImage(L"FakePhoto.jpg");
myImage.GetPropertySize(&size, &count);

// GetAllPropertyItems returns an array of PropertyItem objects
// along with additional data. Allocate a buffer large enough to
// receive the array and the additional data.
PropertyItem* propBuffer = (PropertyItem*)malloc(size);

if(propBuffer)
{
    myImage.GetAllPropertyItems(size, count, propBuffer);
    ...
    free(propBuffer);
    propBuffer = NULL;
}

```

## 错误校验

GDI+ 参考中的大部分例子代码都没有进行错误校验。因为完整的错误校验将会使得示例代码变得更大，使得例子本身想要阐述的要点变得模糊。您不应该直接把文档中的例子代码粘贴到您的产品代码中使用；您应该适当的加入您自己的错误校验代码来增强该代码。

下面的代码展示了实现 GDI+ 错误校验的方法。每次一旦某个 GDI+ 对象被构造，代码都将检查该构造函数是否成功。该检查对于 **Image** 的构造函数而言尤其重要，因为它依赖于对一个文件的读取。如果所有四个 GDI+ 对象（**Graphics**, **GraphicsPath**, **Image**, and **TextureBrush**）均成功构造，代码才可以调用这些对象的方法。每个方法调用也要检测是否成功，一旦发生错误，剩下的方法调用将被略过。

```
Status GdipExample(HDC hdc)
{
    Status status = GenericError;
    INT count = 0;
    Point* points = NULL;

    Graphics graphics(hdc);
    status = graphics.GetLastStatus();
    if(Ok != status)
        return status;

    GraphicsPath path;
    status = path.GetLastStatus();
    if(Ok != status)
        return status;

    Image image(L"MyTexture.bmp");
    status = image.GetLastStatus();
    if(Ok != status)
        return status;

    TextureBrush brush(&image);
    status = brush.GetLastStatus();
    if(Ok != status)
        return status;

    status = path.AddEllipse(10, 10, 200, 100);

    if(Ok == status)
    {
        status = path.AddBezier(40, 130, 200, 130, 200, 200, 60, 200);
    }

    if(Ok == status)
    {
        count = path.GetPointCount();
        status = path.GetLastStatus();
    }

    if(Ok == status)
```



```

{
    points = new Point[count];

    if(NULL == points)
        status = OutOfMemory;
}

if(Ok == status)
{
    status = path.GetPathPoints(points, count);
}

if(Ok == status)
{
    status = graphics.FillPath(&brush, &path);
}

if(Ok == status)
{
    for(int j = 0; j < count; ++j)
    {
        status = graphics.FillEllipse(
            &brush, points[j].X - 5, points[j].Y - 5, 10, 10);
    }
}

if(points)
{
    delete[] points;
    points = NULL;
}

return status;
}

```

## 线程同步

有可能采用超过一个线程来访问同一 GDI+ 对象。然而，GDI+ 对象并不具备自动同步的机制。因此如果您的应用程序中有 2 个线程具有指向同一个 GDI+ 对象的指针的话，同步存取该对象就是您自己的责任了。

如果一个线程试图调用某对象的一个方法，而此时另一个线程正在同一个对象上执行一个方法，此时有些 GDI+ 方法会返回 **ObjectBusy**。不要在返回 **ObjectBusy** 值的时候试图同步访问。可行的是，每次您访问该对象的某个成员或者调用它的某个方法的时候，将这些调用放到某个临界区段中，或者采用其他的标准同步技术。

#### 相关主题

-  [MSDN Library Security Home Page](#) 
-  [Security How-To Resources](#) 
-  [TechNet Security Resources](#) 

## 关于 GDI+

Microsoft Windows GDI+ 是 Windows XP 操作系统或者 Windows 2003 操作系统的一部分，它提供二维矢量图形、图象和排版。GDI+ 由 Windows 图象设备接口（GDI，包含于较早版本的 Windows 中的图形设备接口）演化而来，加入了新的特性的同时优化了现有特性。

以下主题提供了有关采用 C++ 开发语言进行 GDI+ 应用程序开发接口（API）的信息。

### GDI+ 介绍

Microsoft Windows GDI+ 是一个图形设备接口，它允许开发者创建设备无关的应用程序。GDI+ 服务通过一组的 C++ 类实现。

### GDI+ 概览

Microsoft Windows GDI+ 是 Windows XP 和 Windows Server 2003 操作系统的子系统，它负责在屏幕和打印机上显示信息。GDI+ 是一个应用程序编程接口（API），它由一组 C++ 类实现。

顾名思义，GDI+ 是对于 GDI 的继承，后者包含于早期的 Windows 版本中。Windows XP 或者 Windows Server 2003 支持 GDI 以保持对现有程序的兼容性，但是编写新程序的开发者应该使用 GDI+ 来实现他们所有的图形需求，因为 GDI+ 不仅优化了大部分 GDI 性能而且提供了更多特性。

一个图形设备接口，比如 GDI+，允许应用程序开发者将信息显示在显示器或者打印机上，而无需考虑该显示设备的详细情况。应用程序开发者调用 GDI+ 类所提供的方法，而这些方法又依次适当的调用特定设备驱动程序。GDI+ 使得应用程序和图形硬件隔离开来，得益于此，开发者因而可以创建设备无关的应用程序。

### GDI+ 的三个组成部分

Microsoft Windows GDI+ 服务分为以下 3 个主要部分：

- **二维矢量图形**

矢量图形由图元（比如线条、曲线和图形）组成，它们由一系列坐标系统的点集组成。例如，一条直线可以由它的两个端点所确定，一个矩形可以通过给出它的左上角点的位置加上它的宽度、高度来确定。一个简单的路径可以由一个由直线连接而成的点数组来描述。一条贝塞尔样条是一个由 4 个控制点所描述的高级曲线。

GDI+ 提供了用于存储这些图元本身信息的类、如何绘制这些图元信息的类以及实际绘制这些图元的类。例如，Rect 类存储一个矩形的尺寸位置；Pen 类存储线条颜色、线条宽度以及线条样式等信息；而 Graphics 类则提供绘制线条、矩形、路径和其他图形的方法。同时，有几个 Brush 类还用于存储有关闭合图形和路径内部填充颜色和图案的信息。

- 图象

某些图片很难或者不可能采用矢量图形技术来显示。比如，工具栏按钮图片和图标就很难通过一系列线条和曲线来描述。一张拥挤的棒球馆的高分辨率的数码照片更难采用矢量技术来创建。这种类型的图象采用位图进行存储，即由表示屏幕上独立点的颜色的数字型数组所组成。用于存储位图信息的数据结构往往比矢量图形要复杂的多，因此出于此种原因 GDI+ 中提供了好几种类。类似的类比如 **CachedBitmap**，用于存储一张内存图片，供快速存取和显示用。

- 排版

排版关系到多种字体、尺寸和样式文字的显示。GDI+ 提供给人深刻印象数量的对这项复杂的任务的支持。新的特性中包括了子像素抗锯齿功能，它使得在液晶显示屏上可以显示更加平滑的显示文本。

### 基于类的接口架构

Microsoft Windows GDI+ 包含大约 40 个类、50 个枚举和 6 个结构体。同时也有少数几个函数不属于任何类。**Graphics** 类是整个 GDI+ 接口的核心；它是实际进行线条、曲线、图形、图象和文本绘制的类。

多数类和 **Graphics** 类配合使用。例如，**Graphics::DrawLine** 方法接受一个点传给一个 **Pen** 对象，该对象保存了即将绘制的线条的属性（颜色、宽度、虚线类型及其他）。**Graphics::FillRectangle** 方法接受一个点传给 **LinearGradientBrush** 对象，该对象协同 **Graphics** 对象实现矩形的渐变色填充。**Font** 和 **StringFormat** 对象影响到 **Graphics** 对象绘制文本的方式。**Matrix** 对象用于存储和生成一个 **Graphics** 对象的世界变换矩阵，用于旋转、缩放和翻转图象。

有些类是主要用作数据类型结构体。这些类中大多数（例如 **Rect**、**Point** 和 **Size** 类）用于普通目的。其他的则用于特殊目的，被看作是辅助类。例如，**BitmapData** 类是 **Bitmap** 类的辅助类，**PathData** 类是 **GraphicsPath** 类的辅助类。GDI+ 同时定义了少数几个结构体用于组织数据。例如，**ColorMap** 结构体存储一对颜色对象，构成了一个颜色转换表的入口项目。

GDI+ 定义了数个枚举常量，它们是相关常数的集合。例如，**LineJoin** 枚举包含了 **LineJoinBevel**，**LineJoinMiter** 和 **LineJoinRound** 几个元素，表示两根线条的连接方式。

GDI+ 提供了少数几个不属于任何类的函数。其中的 2 个是 **GdiplusStartup** 和 **GdiplusShutdown**。您必须在进行任何 GDI+ 调用之前调用 **GdiplusStartup**，在您结束 GDI+ 调用的时候调用 **GdiplusShutdown**。

### GDI+ 提供了哪些新东西？

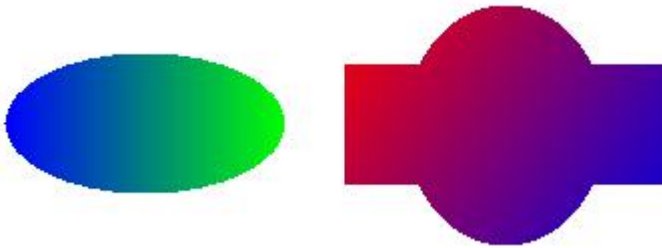
Microsoft Windows GDI+ 不同于 GDI 体现在 2 个方面。第一，GDI+ 通过提供新的功能扩展了 GDI 的特性，比如渐变画刷和半透明混合。第二，编程模型的改进使得图形开发更加简单和灵活。

### 新特征

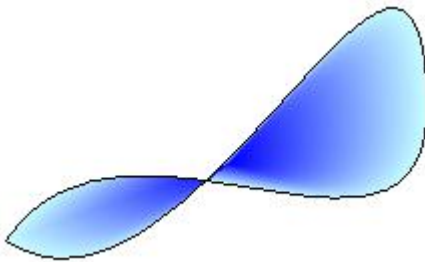
下面将阐述 Microsoft Windows GDI+ 的几个新特征：

- **渐变画刷**

GDI+扩展了 GDI 提供线性渐变和路径渐变画刷，用于填充图形、路径和区域。渐变画刷也可用于绘制线条、曲线和路径。当您采用线性渐变画刷填充图形的时候，颜色将在穿越该图形时逐渐改变。例如，假设您创建了一个水平渐变画刷，同时指定左边缘为蓝色、右边缘为绿色。当您采用水平渐变画刷填充图形时，颜色将从它的左边到右边逐渐由蓝色过渡到绿色。同样地，一个垂直渐变画刷填充的图形的颜色将自上而下渐变。下面两幅插图显示了一个水平渐变画刷填充的椭圆和一个对角渐变画刷填充的区域。

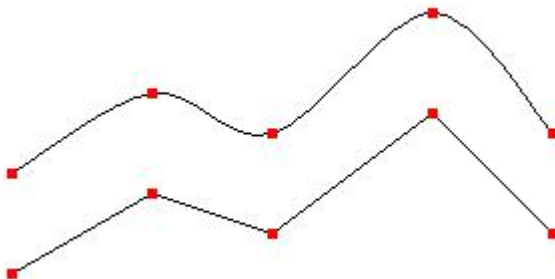


如果您采用路径渐变画刷填充图形，您可以有许多选择来指定颜色从图形的一个区域到另一个区域的如何过渡。一种选择就是设置一个中心颜色和边界颜色，这样图形由内向外像素颜色将逐渐改变。下图显示的是一个由路径渐变画刷填充的路径（由一对贝塞尔样条所创建）。



- **基数样条**

GDI+支持基数样条，而 GDI 不支持。基数样条是一组单个曲线按照一定的顺序连接而成的一条较大曲线。样条由一系列点指定，并通过每一个指定的点。由于基数样条平滑地穿过组中的每一个点（不出现尖角），因而它比用直线连接创建的路径更精确。下面是分别使用两种方法创建的图形，一个使用基数样条，一个使用直线。

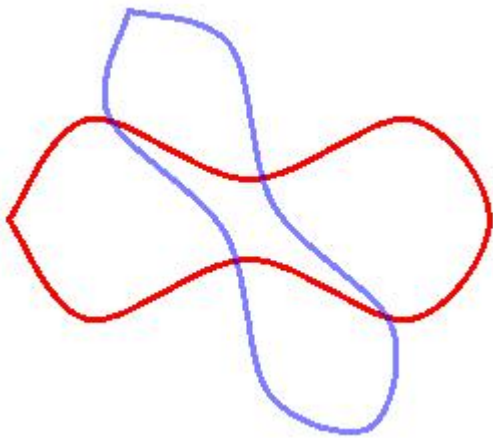


- 独立的路径对象

在 GDI 中，一条路径是属于一个设备场景的，在绘制完成后就被销毁了。在 GDI+ 中，绘图工作由 Graphics 对象来完成，您可以创建和保留几个与 **Graphics** 独立的 GraphicsPath 对象。绘图操作时 **GraphicsPath** 对象不被破坏，这样您就可以多次使用同一个 **GraphicsPath** 对象画路径了。

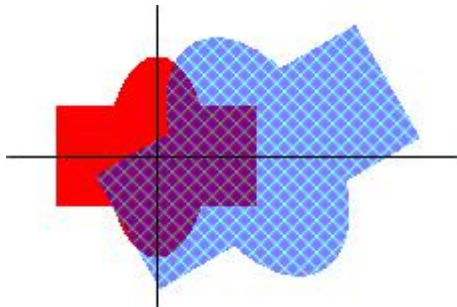
- 变形和矩阵对象

GDI+ 提供了矩阵对象，一个非常强大的工具，使得图形变换（旋转、平移等）变得轻松灵活。一个矩阵对象总是和一个图形变换相联系起来。比方说，**GraphicsPath** 对象有一个 Transform 方法，它的一个参数能够接受 **Matrix** 对象的地址。单个  $3 \times 3$  矩阵可以存储一个变形或者一系列变形。下图是一个图形变换前后的例子，变换按照先缩放后旋转完成。



- 可伸缩区域

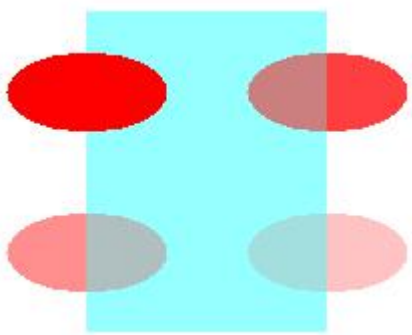
GDI+ 在对区域的支持上对 GDI 进行了很大的改进。在 GDI 中，区域存储在设备坐标中，唯一可进行的图形变换的操作是平移。而 GDI+ 用世界坐标存储区域，允许对区域进行任何图形变换（例如缩放），图形变换以变换矩阵存储。下图显示的是经过连续 3 个变换前后的一个区域（缩放、旋转、平移）：



- alpha 混合










请注意上图中，您可以通过变换后的区域（阴影填充）看到变换前的区域（红色填充）。这种效果可以通过 GDI+ 支持的  $\alpha$  混合来实现。利用  $\alpha$  混合，您可以指定填充颜色的透明度。透明色将与背景色混合

一 填充色越透明，背景色显示越清晰。下图所示的四个椭圆被填充了同样的颜色（红色），但由于拥有不同的透明度而呈现不同的显示效果。



- 多种图像格式支持

GDI+提供了 Image、Bitmap 和 Metafile 类，用于载入、保存和处理多种格式的图形。以下格式均支持：

-  [BMP](#)
-  [Graphics Interchange Format \(GIF\)](#)
-  [JPEG](#)
-  [Exif](#)
-  [PNG](#)
-  [TIFF](#)
-  [ICON](#)
-  [WMF](#)
-  [EMF](#)

### 编程模式的改变

下面将阐述采用 GDI+ 编程与 GDI 的几个方面的不同之处：

- 设备场景、句柄和 **Graphics** 对象

如果您曾经使用过 GDI 编写过应用程序，您肯定对设备场景（DC）的概念非常熟悉。设备场景是 Windows 使用的一个数据结构，用于存储具体设备性能和与如何在设备上绘制项目的相关信息。而且视频显示器的设备场景还与显示器上的特定窗口有关。首先您必须获得一个设备场景句柄（HDC），然后在图形绘制时您把这个句柄作为一个参数传递给 GDI 函数。当然您也可以把它传递给获得或设置设备场景有关属性的函数。

您在使用 GDI+ 的时候，您不必像在 GDI 中那样关心设备场景句柄。您只需简单地创建一个 Graphics 对象，然后以您熟悉的面向对象的方式（比如 myGraphicsObject.DrawLine(parameters)）调用它的方法即可。**Graphics** 对象是 GDI+ 的核心，正如设备场景是 GDI 的核心一样。设备场景（DC）和图形对象（Graphics）在不同的环境下扮演着同样的角色，发挥着类似的作用，但是两者也存在着本质的不同。前者使用基于句柄的编程模型而后者使用面向对象的编程模型。

**Graphics** 对象和设备场景一样，与屏幕上的特定窗体有关，它包含着图形绘制的有关属性信息（比如，平滑模式和隐式文本渲染）。然而，**Graphics** 对象并没有像 GDI 那样与 Pen、Path、Image 或者 Font 等搅在一起。比如，在 GDI 中，您必须先调用 **SelectObject** 将一个 Pen 对象与设备场景关联，然后您才能使用设备场景绘制一条线。这被称为将画笔选入设备场景。您在设备场景中绘制的所有线条都将采用这个画笔直到您选择另一个画笔为止。在 GDI+ 中，您只需将一个 Pen 对象作为参数传递给 **Graphics** 类的 **DrawLine** 方法即可。您可以在一连串的 **DrawLine** 调用中传入不同的 **Pen** 对象，而不必将给定的 **Pen** 对象与 **Graphics** 对象关联。

- 画线的 2 种方法

下面两个例子都是从位置(20, 10)到位置(200,100)绘制一条宽度为 3 的红色线条。前一个例子采用 GDI 调用，第二个例子通过 C++ 类接口调用 GDI+。

- **Drawing a line with GDI**

采用 GDI 画线，您需要 2 个对象：设备场景和画笔。您通过调用 **CreatePen** 获取一个句柄。紧接着，您调用 **SelectObject** 将画笔选入设备场景。然后调用 **MoveToEx** 将画笔位置设为(20,10)，然后调用绘制一条直线到位置(200,100)。注意 **MoveToEx** 和 **LineTo** 均需要一个 *hdc* 作为参数。

```
HDC          hdc;
PAINTSTRUCT  ps;
HPEN         hPen;
HPEN         hPenOld;
hdc = BeginPaint(hWnd, &ps);
    hPen = CreatePen(PS_SOLID, 3, RGB(255, 0, 0));
    hPenOld = (HPEN)SelectObject(hdc, hPen);
    MoveToEx(hdc, 20, 10, NULL);
    LineTo(hdc, 200, 100);
    SelectObject(hdc, hPenOld);
    DeleteObject(hPen);
EndPaint(hWnd, &ps);
```

- 采用 **GDI+** 和 **C++** 类接口画线

通过 GDI+ 和 C++ 类接口画线，您需要一个 **Graphics** 对象和一个 **Pen** 对象。注意您不需要向这些对象提供窗体句柄。相反，您只需要构造一个 **Graphics** 类（一个 **Graphics** 对象）和一个 **Pen** 类（一个 **Pen** 对象）即可。画线涉及 **Graphics** 类的 **Graphics::DrawLine** 方法。**Graphics::DrawLine** 方法的第一个参数是一个 **Pen** 对象的指针。较之前面将 Pen 选入设备场景的 GDI 例子，这种方案更加更加简单灵活。

```
HDC          hdc;
PAINTSTRUCT  ps;
Pen*         myPen;
Graphics*    myGraphics;
hdc = BeginPaint(hWnd, &ps);
    myPen = new Pen(Color(255, 255, 0, 0), 3);
```



```

myGraphics = new Graphics(hdc);
myGraphics->DrawLine(myPen, 20, 10, 200, 100);
delete myGraphics;
delete myPen;
EndPaint(hWnd, &ps);

```

- **钢笔、画刷、路径、图形和字体作为参数**

前面的例子说明了 **Pen** 对象可以独立于 **Graphics** 对象被创建和保留，该对象提供具体绘图方式。同样地，画刷、图形路径、图象和字体对象可以独立于 **Graphics** 对象被创建和保留。**Graphics** 类提供的许多绘图方法都接受 **Brush**、**GraphicsPath**、**Image** 或者 **Font** 对象作为参数之一。例如，一个 **Brush** 对象的地址作为参数传递给 **FillRectangle** 方法，而一个 **GraphicsPath** 对象的地址作为参数传递给 **DrawPath** 方法。类似地，**Image** 和 **Font** 对象的地址传递给 **DrawImage** 和 **DrawString** 方法。这与 GDI 中需要先选入一个画刷、路径、图象或者字体到指定设备场景，然后再将设备句柄作为参数传递给绘图函数的方法相比形成了强烈对比。

- **方法重载**

许多 GDI+ 方法被重载了；也就是说，存在不同的方法，它们拥有相同的函数名和不同的参数列表。例如，**Graphics** 类的 **DrawLine** 方法有以下几种形式：

```

Status DrawLine(IN const Pen* pen,
                IN REAL x1,
                IN REAL y1,
                IN REAL x2,
                IN REAL y2);
Status DrawLine(IN const Pen* pen,
                IN const PointF& pt1,
                IN const PointF& pt2);
Status DrawLine(IN const Pen* pen,
                IN INT x1,
                IN INT y1,
                IN INT x2,
                IN INT y2);
Status DrawLine(IN const Pen* pen,
                IN const Point& pt1,
                IN const Point& pt2);

```

所有上面 **DrawLine** 方法的四种变体都接受一个 **Pen** 对象指针、起点坐标和终点坐标。前两个方法接受浮点数字型的坐标参数，而后面两个方法接受整数类型的坐标参数。第一个和第三个方法接受由数字分隔开的坐标；第二个和第四个方法接受由一对 **Point**（或者 **PointF**）对象表示的坐标。

- **不再有当前位置**

注意，前面显示的 **DrawLine** 方法同时提供了线段的起点和终点作为参数。不同的是，在 GDI 中您必须先设置当前位置然后才能绘制一个起于(x1, y1)终于(x2, y2)的线段。GDI+中已经完全摒弃了当前位置的概念。

- **绘图与填充方法分离**

在绘制图形比如矩形的边框和填充其内部区域方面，GDI+比 GDI 更加灵活。GDI 有个 **Rectangle** 函数可以一步实现绘制矩形和填充其内部区域。边框采用当前选入的画笔绘制，内部区域采用当前选入的笔刷填充。

```
hBrush = CreateHatchBrush(HS_CROSS, RGB(0, 0, 255));
hPen = CreatePen(PS_SOLID, 3, RGB(255, 0, 0));
SelectObject(hdc, hBrush);
SelectObject(hdc, hPen);
Rectangle(hdc, 100, 50, 200, 80);
```

在 GDI+中，绘制矩形边框和填充其内部区域的方法是独立开来的。**Graphics** 类的 **DrawRectangle** 方法有一个参数，用于传入 **Pen** 对象的地址。而 **FillRectangle** 方法有一个参数，用于传入 **Brush** 对象的地址。

```
HatchBrush* myHatchBrush = new HatchBrush(
    HatchStyleCross,
    Color(255, 0, 255, 0),
    Color(255, 0, 0, 255));
Pen* myPen = new Pen(Color(255, 255, 0, 0), 3);
myGraphics.FillRectangle(myHatchBrush, 100, 50, 100, 30);
myGraphics.DrawRectangle(myPen, 100, 50, 100, 30);
```

注意，GDI+中 **FillRectangle** 和 **DrawRectangle** 方法的参数指定矩形的左、上、宽度和高度。相对地，在 GDI 的 **Rectangle** 函数中，参数指定的是矩形的左、右、上、下。同时还要注意，GDI+中的 **Color** 类的构造函数有 4 个参数。后面的 3 个参数就是通常的红、绿、蓝的值；第一个参数指定的是  $\alpha$  半透明值，它表示该颜色与背景色混合的程度。

- **构造区域**

GDI 提供了几个函数可以创建区域: **CreateRectRgn**、**CreateEllpticRgn**、**CreateRoundRectRgn**、**CreatePolygonRgn** 和 **CreatePolyPolygonRgn**。您可能也期望 GDI+中的 **Region** 类提供类似的构造函数，传入矩形、椭圆、圆角矩形和多边形作为参数，但是情况并非如此。在 GDI+中，**Region** 类提供一个传入 **Rect** 对象引用的构造函数，和另一个传入 **GraphicsPath** 对象地址的构造函数。如果您想创建一个基于椭圆、圆角矩形或者多边形的区域，您可以很容易地通过创建一个 **GraphicsPath** 对象（包含椭圆等），然后将 **GraphicsPath** 对象的地址传递给 **Region** 构造函数即可。

在 GDI+中可以很简单地通过图形和路径的组合创建一个复杂的区域。**Region** 类有 **Union** 和 **Intersect** 方法，用于加入一个路径或者另一个区域到一个已有区域中。GDI+中一个很好的特性是，当一个 **GraphicsPath** 对象作为 **Region** 构造函数的参数传入的时候，它本身并不会被销毁。而在 GDI 中，

您可以通过 **PathToRegion** 函数将一个路径转换为一个区域，但是同时路径也被销毁了。同样地，**GraphicsPath** 对象在其地址作为 **Union**（合并）和 **Intersect**（相交）方法的参数传入时本身也不会被销毁，因此您可以将一个指定的路径作为一个组成部分来创建几个独立的区域。下面例子中，假设 *onePath* 是一个 **GraphicsPath** 对象（简单抑或复杂）的指针并且已经被初始化：

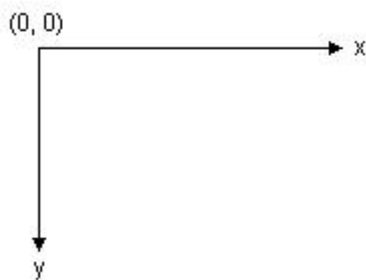
```
Region region1(rect1);  
Region region2(rect2);  
region1.Union(onePath);  
region2.Intersect(onePath);
```

## 线条、曲线和图形

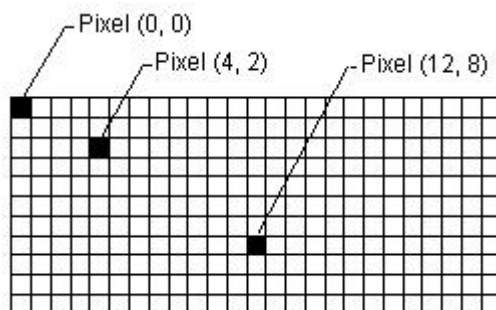
GDI+的矢量图部分用于绘制线条、绘制曲线，绘制和填充图形。

### 矢量图概览

Microsoft Windows GDI+在一个坐标系统中绘制线条、矩形和其他图形。您可以选择各种不同的坐标系统，但是默认的坐标系统规定其左上角位置为起点，X 轴指向右边、Y 轴之指向下边。默认坐标系统的度量单位是像素（Pixel）。

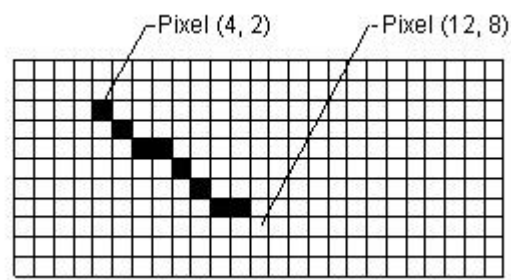


电脑显示器在一个矩形点阵上创建其显示画面，这些点称为图片要素或者像素。不同的显示器其在屏幕上显示的点数不尽相同，并且通常一个独立显示器其显示的像素总数可以由用户进行调节。



在您采用 GDI+绘制线条、矩形和曲线的时候，您需要提供一些关于绘制对象的关键性信息。例如，您可以通过 2 点确定一条线段，您可以通过一个点、高度和宽度确定一个矩形。GDI+与显示器驱动程序

协同工作，来判断哪个像素应该开启用于显示线条、矩形和曲线。下面所示，显示一条从点(4, 2)到点(12, 8)的线条的像素情况。



总的来说，某些基本的构成块已经证明是最对建立二维图形非常有用。这些基本构成块，GDI+均支持，包括：

- [Lines](#)（线条）
- [Rectangles](#)（矩形）
- [Ellipses](#)（椭圆）
- [Arcs](#)（弧形）
- [Polygons](#)（多边形）
- [Cardinal splines](#)（基数样条）
- [Bézier splines](#)（贝塞尔样条）

GDI+中的 **Graphics** 类提供这些方法来绘制前面列表中的对象：**DrawLine**、**DrawRectangle**、**DrawEllipse**、**DrawPolygon**、**DrawArc**、**DrawCurve** (用于基数样条)和 **DrawBezier**。每个方法都被重载；也就是说，每种方法都以不同参数列表的变体出现。例如，**DrawLine** 方法的一个变体接受一个 **Pen** 对象的地址和四个整型值，而它的另一个变体则接受一个 **Pen** 对象地址和两个 **Point** 对象引用。

绘制线条、矩形和贝塞尔样条的方法都有其复数形式的伴随方法，这些方法在一次调用中绘制多个项目：**DrawLines**、**DrawRectangles** 和 **DrawBeziers**。同样地，**DrawCurve** 方法也有一个伴随方法 **DrawClosedCurve**，该函数通过连接曲线的起点和终点创建一个闭合图形。

所有的 **Graphics** 类的方法都得配合 **Pen** 对象使用。因此，为了绘制任何东西，您必须至少创建 2 个对象：一个 **Graphics** 对象和一个 **Pen** 对象。**Pen** 对象存储诸如线条宽度、颜色等绘图属性。**Pen** 对象的地址将作为一个参数传递给绘图方法。例如，一个 **DrawRectangle** 方法的变体需要传入一个 **Pen** 对象地址和 4 各整数值作为参数，该方法绘制一个左上角为(20,10)、宽度为 100、高度为 50 的矩形。

```
myGraphics.DrawRectangle(&myPen, 20, 10, 100, 50);
```

### 钢笔、线条和矩形

采用 GDI+ 绘制线条需要一个 **Graphics** 对象和一个 **Pen** 对象。**Graphics** 对象提供实际的绘图方法，而 **Pen** 对象存储线条属性，例如颜色、宽度和样式等。绘制线条只需要简单调用 **Graphics** 对象的 **DrawLine** 方法即可。**Pen** 对象的地址作为参数之一传递给 **DrawLine** 方法。下面的例子是绘制一条从点(4, 2)到点(12, 6)的线段。

```
myGraphics.DrawLine(&myPen, 4, 2, 12, 6);
```

**DrawLine** 是一个在 **Graphics** 类中被重载的方法，因此您可以提供几种不同的参数。例如，您可以构造 2 个 **Point** 对象，然后将这 2 个 **Point** 对象的引用作为参数传递给 **DrawLine** 方法。

```
Point myStartPoint(4, 2);
Point myEndPoint(12, 6);
myGraphics.DrawLine(&myPen, myStartPoint, myEndPoint);
```

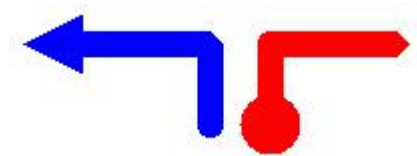
您也可以在构造 **Pen** 对象时给它的属性赋值。例如，有个 **Pen** 的构造函数允许您指定颜色和宽度。下面的例子是从点(0, 0)到点(60, 30)绘制一条宽度为 2 的蓝色线段。

```
Pen myPen(Color(255, 0, 0, 255), 2);
myGraphics.DrawLine(&myPen, 0, 0, 60, 30);
```

**Pen** 对象同样有自己的属性，例如虚线样式，您可以用来指定线条特征。例如，下面的例子用于绘制一条从点(100, 50)到点(300, 80)的虚线。

```
myPen.SetDashStyle(DashStyleDash);
myGraphics.DrawLine(&myPen, 100, 50, 300, 80);
```

您可以使用 **Pen** 对象的多种方法来设置更多的线条属性。**SetStartCap** 和 **SetEndCap** 方法指定线条末端的表现形式；线帽可以为扁平、方块、圆角、三角或者用户定义形状。**SetLineJoin** 方法允许您设置线条间的联接方式是斜接（有明显边角）、斜切、圆或者裁剪。下图显示的是具有不同的端点类型和连接方式的线条。



绘制矩形的方法和画线的方法相似。绘制一个矩形，您需要一个 **Graphics** 对象和一个 **Pen** 对象。**Graphics** 对象提供一个 **DrawRectangle** 方法，而 **Pen** 对象存储输入线条宽度和颜色等属性。**Pen** 对象的地址作为参数之一传递给 **DrawRectangle** 方法。下面的例子是绘制一个左上角为(100,50)、宽度为 80、高度为 40 的矩形。

```
myGraphics.DrawRectangle(&myPen, 100, 50, 80, 40);
```

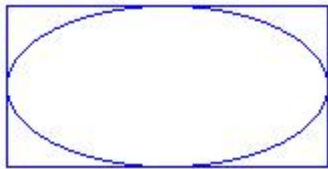
**DrawRectangle** 是一个在 **Graphics** 类中被重载的方法，因此也有几种不同传递参数的方式。例如，您可以先构造一个 **Rect** 对象然后将 **Rect** 对象的引用作为参数之一传递给 **DrawRectangle** 方法。

```
Rect myRect(100, 50, 80, 40);
myGraphics.DrawRectangle(&myPen, myRect);
```

**Rect** 对象的某些方法可以控制和收集矩形的有关信息。例如，**Inflate** 和 **Offset** 方法可以改变矩形的尺寸和位置。**IntersectsWith** 方法告诉您一个矩形是否和另一个矩形交叉了，**Contains** 方法则告诉您指定的点是否在一个矩形内部。

## 椭圆和弧

一个椭圆由它的外接矩形来描述。下图显示了一个椭圆和它的外接矩形。



绘制一个椭圆，您需要一个 **Graphics** 对象和一个 **Pen** 对象。**Graphics** 对象提供 **DrawEllipse** 方法，**Pen** 对象存储诸如线条宽度和颜色等信息。**Pen** 对象的地址作为参数之一传递给 **DrawEllipse** 方法。传递给 **DrawEllipse** 方法其余的参数指定其外接矩形。下面的例子将绘制一个椭圆；其外接矩形宽度为 160、高度为 80，左上角位置为(100,50)。

```
myGraphics.DrawEllipse(&myPen, 100, 50, 160, 80);
```

**DrawEllipse** 是一个在 **Graphics** 类中被重载的方法，因此也有几种不同传递参数的方式。例如，您可以先构造一个 **Rect** 对象然后将 **Rect** 对象的引用作为参数之一传递给 **DrawEllipse** 方法。

```
Rect myRect(100, 50, 160, 80);  
myGraphics.DrawEllipse(&myPen, myRect);
```

弧是椭圆的一部分。绘制弧形，您需要调用 **Graphics** 类的 **DrawArc** 方法。**DrawArc** 方法的参数和 **DrawEllipse** 方法的参数一样，除此之外还需要提供起始角和扫描角。下面的例子绘制一条弧形，其起始角为 30 度，扫描角为 180 度。

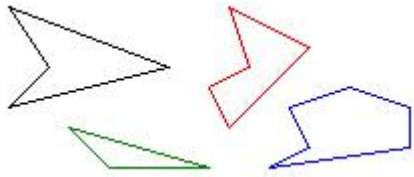
```
myGraphics.DrawArc(&myPen, 100, 50, 160, 80, 30, 180);
```

下图同时显示了弧形、椭圆与外接矩形。



## 多边形

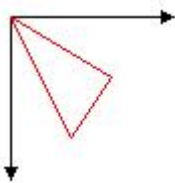
多边形是由 3 个或 3 个以上的直边所组成的闭合图形。例如，三角形就是 3 条边所组成，矩形由 4 条边组成，五角星由 5 条边所组成。下图显示了几种不同的多边形。



绘制多边形，您需要一个 **Graphics** 对象，一个 **Pen** 对象，以及一个 **Point**（或者 **PointF**）对象数组。**Graphics** 对象提供 **DrawPolygon** 方法，Pen 对象存储多边形的线条宽度和颜色等信息，Point 对象数组则存储相连直线的点集。**Pen** 对象的地址和 **Point** 对象数组作为参数传递给 **DrawPolygon** 方法。下面的例子将绘制一个 3 条边的多边形。注意在 *myPointArray* 中只有 3 个点：(0, 0)、(50, 30) 和 (30, 60)。**DrawPolygon** 方法自动绘制从(30, 60)返回点(0, 0)的线条使该多边形闭合。

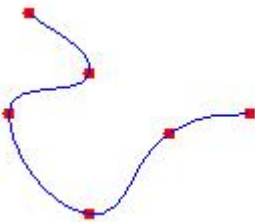
```
Point myPointArray[] =
    {Point(0, 0), Point(50, 30), Point(30, 60)};
myGraphics.DrawPolygon(&myPen, myPointArray, 3);
```

下图显示了该多边形。



## 基数样条

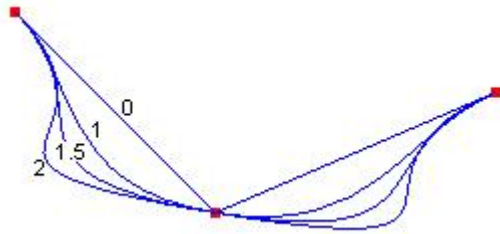
基数样条是一组单个曲线按照一定的顺序连接而成的一条较大曲线。样条由一个点数组和一个张力参数描述。由于基数样条平滑地穿过数组中的每一个点；在曲线的密度上不会出现锐角和突变。下图显示一组点和穿过它们每个点的基数样条。



物理样条是一小片木头或者其他柔性材质做成的。在数学样条诞生之前，设计人员采用物理样条来绘制曲线。它们将样条置于纸上然后定位一系列锚点，然后用铅笔沿着样条绘制曲线。给出的一系列点可能产生不同的曲线，这取决于物理样条的性质。例如，与一个极其易弯曲的样条相比，有较高的抗弯能力的一个样条将生产一条不同的曲线。

数学样条的公式基于柔性杆的特性，因此数学样条生产的曲线类似于曾经由物理样条生产的曲线。正如物理样条通过给定的一组点时在不同的张力下的将生成一条不同的曲线一样，数学样条在张力参数不同的时候也将生成不同的曲线。下图显示了通过相同一组点集的 4 条基数样条。每条样条都标注了它的张

力参数。注意张力系数为 0 的情况下相当于无限的物理张力，迫使曲线走点之间的最短路径(直线)。张力系数为 1 表示没有物理张力，此时样条采用最小弯程。如果张力系数大于 1，此时的样条看起来就像被压扁的弹簧，被迫经过更长的路径。



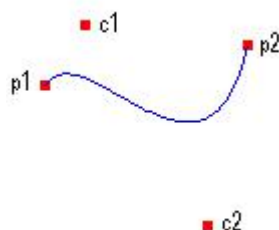
需要注意的是，以上 4 条样条在顶点处都拥有相同的切线。切线表示从一个起始点沿曲线指向下一个点间的直线。同样的，终点共享的切线表示从终点开始的沿曲线曲线指向前一个点。

绘制一条基数样条，您需要一个 **Graphics** 对象，一个 **Pen** 对象和一个 **Point** 对象数组。**Graphics** 对象提供 **DrawCurve** 方法用于绘制基数样条，**Pen** 对象存储诸如线条宽度和颜色等信息，**Point** 对象数组存储曲线经过的点集。下面的例子将绘制一条基数样条，它穿过 *myPointArray* 点集。第三个参数是张力参数。

```
myGraphics.DrawCurve(&myPen, myPointArray, 3, 1.5f);
```

### 贝塞尔样条

贝塞尔样条是由 4 个点所确定的曲线：2 个端点 (p1 和 p2) 和 2 个控制点(c1 和 c2)。曲线始于 p1 终于 p2。曲线并不经过控制点，但是控制点扮演了磁铁的角色，将曲线往某个方向拉从而影响了曲线的走向。下图显示了贝塞尔样条和它的顶点以及控制点。



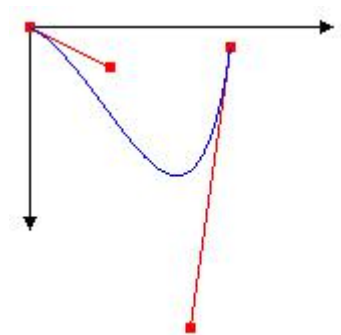
注意，曲线从 p1 开始向控制点 c1 移动。P1 位置的切线是从 p1 到 c1。同时请注意，终点 p2 处的切线是从 c2 到 p2 的。

绘制贝塞尔样条，您需要一个 **Graphics** 对象和一个 **Pen** 对象。**Graphics** 对象提供 **DrawBezier** 方法，而 **Pen** 对象存储诸如线条宽度和颜色等信息。**Pen** 对象的地址作为参数之一传递给 **DrawBezier** 方法。**DrawBezier** 方法余下的参数传入顶点和控制点。下面的例子将绘制一条贝塞尔样条，它的起点为 (0,0)，控制点为(40,20)和(80,150)，终点为(100,10)。

```
myGraphics.DrawBezier(&myPen, 0, 0, 40, 20, 80, 150, 100, 10);
```



下图显示该曲线、控制点和两条切线。



贝塞尔样条最初由 **Pierre Bézier** 在汽车工业设计中发明。它被证明对于多种类型的计算机辅助设计非常有用，同时还用于定义字体轮廓。贝塞尔样条可以产生多种形状，下图罗列了其中一些：

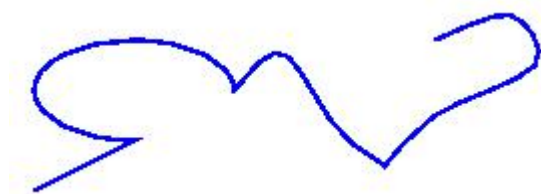


## 路径

路径由线条、矩形以及简单曲线等组合而成。回顾矢量图形概念部分，以下基本构成块被证明对于绘制图象非常有用。

- **Lines** （线条）
- **Rectangles** （矩形）
- **Ellipses** （椭圆）
- **Arcs** (弧线)
- **Polygons** (多边形)
- **Cardinal splines** (基数样条)
- **Bézier splines** (贝塞尔样条)

在 **GDI+** 中，**GraphicsPath** 对象允许您将这些基本组成部分组合成一个单独的单位。整个一组线条、矩形、多边形和曲线可以通过 **Graphics** 类的 **DrawPath** 方法一次性绘制。下图显示的是一条有线条、弧线、贝塞尔样条以及基数样条所组成的路径。



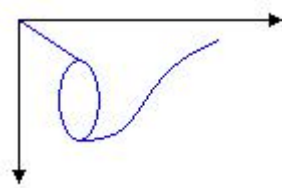
**GraphicsPath** 类提供如下方法用于创建一系列对象：**AddLine**、**AddRectangle**、**AddEllipse**、**AddArc**、**AddPolygon**、**AddCurve** (用于基数样条)以及 **AddBezier**。它们中的每个方法都已被重载；也就是说，您可以传入不同的参数列表。例如，**AddLine** 方法的变体之一需要传入 4 个整形值，而另外一个变体则需要传入 2 个 **Point** 对象。

添加线条、矩形和贝塞尔样条的方法都有其复数形式的伴随方法，这些方法在一次调用中添加多个项目：**AddLines**、**AddRectangles** 和 **AddBeziers**。同样地，**AddCurve** 方法也有一个伴随方法 **AddClosedCurve**，该函数通过连接曲线的起点和终点添加一个闭合曲线。

绘制一条路径，您需要一个 **Graphics** 对象，一个 **Pen** 对象和一个 **GraphicsPath** 对象。**Graphics** 对象提供方法。**Pen** 对象存储诸如线条宽度和颜色等信息。**GraphicsPath** 对象存储线条、矩形和曲线序列用以构成一条路径。**Pen** 对象和 **GraphicsPath** 对象的地址作为参数传给 **DrawPath** 方法。下面的例子将绘制一条路径，它由一根线条、一个椭圆和一条贝塞尔样条组成。

```
myGraphicsPath.AddLine(0, 0, 30, 20);  
myGraphicsPath.AddEllipse(20, 20, 20, 40);  
myGraphicsPath.AddBezier(30, 60, 70, 60, 50, 30, 100, 10);  
myGraphics.DrawPath(&myPen, &myGraphicsPath);
```

下图所示为该路径：



除了添加线条、矩形和曲线到路径外，您还可以添加路径到路径。这允许您将已有路径组合为更大更复杂的路径。下面的代码将 *graphicsPath1* 和 *graphicsPath2* 加入到 *myGraphicsPath* 中。**AddPath** 方法的第二个参数用于指定新增路径是否与已有路径相连。

```
myGraphicsPath.AddPath(&graphicsPath1, FALSE);  
myGraphicsPath.AddPath(&graphicsPath2, TRUE);
```

另外还有 2 个项目您可以加入路径中：字符串和饼图。饼图是椭圆的一部分。下面的例子将创建一个由弧线、基数样条、字符串和饼图组成的路径。

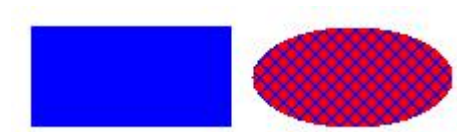
```
myGraphicsPath.AddArc(0, 0, 30, 20, -90, 180);  
myGraphicsPath.AddCurve(myPointArray, 3);  
myGraphicsPath.AddString(L"a string in a path", 18, &myFontFamily,  
    0, 24, myPointF, &myStringFormat);  
myGraphicsPath.AddPie(230, 10, 40, 40, 40, 110);  
myGraphics.DrawPath(&myPen, &myGraphicsPath);
```

下图所示为这个路径。注意这个路径并没有连接起来：弧线、基数样条、字符串和饼图都是相离的。



## 画刷和填充图形

一个闭合图形比如矩形和椭圆包含一个边框和内部区域。边框是由 **Pen** 对象绘制，而内部区域由 **Brush** 对象进行填充。Microsoft Windows GDI+ 提供几种画刷类用于填充闭合图形的内部区域：**SolidBrush**、**HatchBrush**、**TextureBrush**、**LinearGradientBrush** 和 **PathGradientBrush**。所有这些类都继承于 **Brush** 类。下图显示了一个由纯色画刷填充的矩形和一个由阴影画刷填充的椭圆。



- 纯色画刷

填充一个闭合图形，您需要一个 **Graphics** 对象和一个 **Brush** 对象。**Graphics** 对象提供方法，比如 **FillRectangle** 和 **FillEllipse**，而 **Brush** 对象存储诸如颜色和图案等填充属性。**Brush** 对象的地址作为参数之一传递给填充方法。下面的例子将用实行红色填充一个椭圆。

```
SolidBrush mySolidBrush(Color(255, 255, 0, 0));  
myGraphics.FillEllipse(&mySolidBrush, 0, 0, 60, 40);
```

注意上例中，画刷采用的是 **SolidBrush** 类型，它继承于 **Brush**。

- 阴影画刷

当您需要填充一个阴影画刷时，您需要指定其前景色、背景色和阴影样式。前景色就是阴影的颜色。

```
HatchBrush myHatchBrush(  
    HatchStyleVertical,  
    Color(255, 0, 0, 255),  
    Color(255, 0, 255, 0));
```

GDI+ 提供了超过 50 种阴影样式，这些样式在 **HatchStyle** 中定义。下面显示的 3 种阴影分别是水平、正向对角线和十字交叉阴影。



- 纹理画刷

通过纹理画刷，您可以使用存储于位图中的纹理来填充图形。例如，假设下面的图形存储在磁盘文件 `MyTexture.bmp` 中。



下面的例子将创建一个由 `MyTexture.bmp` 中存储的图片反复填充得到的椭圆。

```
Image myImage(L"MyTexture.bmp");
TextureBrush myTextureBrush(&myImage);
myGraphics.FillEllipse(&myTextureBrush, 0, 0, 100, 50);
```

下图所示为填充结果：



- **渐变画刷**

您可以采用渐变画刷填充一个图形，使得该图形可以从一个部分到另一部分由一种颜色渐变为其他颜色。例如，一个水平渐变画刷将使得从左至右颜色渐变。下面的例子将采用水平渐变画刷填充一个椭圆，从左至右颜色由蓝色逐渐变为绿色。

```
LinearGradientBrush myLinearGradientBrush(
    myRect,
    Color(255, 0, 0, 255),
    Color(255, 0, 255, 0),
    LinearGradientModeHorizontal);
myGraphics.FillEllipse(&myLinearGradientBrush, myRect);
```

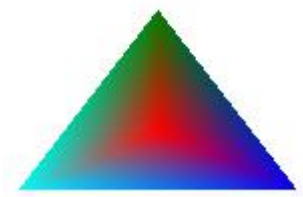
下图所示为填充后的椭圆：



路径渐变画刷允许您设置从中心向边界渐变得画刷。



路径渐变画刷非常灵活。下图中用于填充三角形的渐变画刷中心为红色，向每个顶点渐变为 3 个不同的颜色。



### 开放与闭合曲线

下面显示了两个曲线：一个开放，一个闭合。



闭合曲线因为有内部区域因而可以被画刷填充。GDI+ 中的 **Graphics** 类提供如下方法用于填充闭合图形和曲线: **FillRectangle**、**FillEllipse**、**FillPie**、**FillPolygon**、**FillClosedCurve**、**FillPath** 和 **FillRegion**。任何时候在您调用这些方法的其中之一时，您必须将一个指定类型的画刷 (**SolidBrush**、**HatchBrush**、**TextureBrush**、**LinearGradientBrush** 或者 **PathGradientBrush**) 地址作为参数之一传入。

**FillPie** 方法伴随着 **DrawArc** 方法。正如 **DrawArc** 方法绘制椭圆边界的一部分，**FillPie** 方法填充椭圆内部区域的一部分。下面的例子将绘制一段弧线，然后填充该椭圆内部的相应区域。

```
myGraphics.FillPie(&mySolidBrush, 0, 0, 140, 70, 0, 120);  
myGraphics.DrawArc(&myPen, 0, 0, 140, 70, 0, 120);
```

下图所示为这条弧线和填充的饼图。



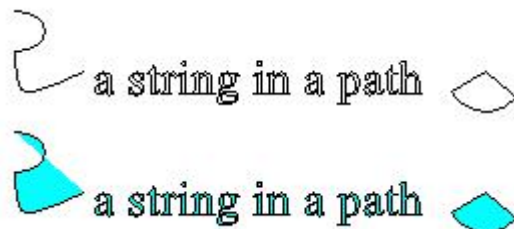
**FillClosedCurve** 方法伴随 **DrawClosedCurve** 方法。这两个方法都将终点自动连接起点从而使曲线闭合。下面的例子将绘制一条穿过(0, 0)，(60, 20)和(40, 50)的曲线。然后该曲线通过连接(40, 50)和起点(0, 0)自动闭合，然后采用实心画刷对该区域进行填充。

```
Point myPointArray[] =
    {Point(10, 10), Point(60, 20), Point(40, 50)};
myGraphics.DrawClosedCurve(&myPen, myPointArray, 3);
myGraphics.FillClosedCurve(&mySolidBrush, myPointArray, 3, FillModeAlternate)
```

一条路径可以包含多个图形（子路径）。**FillPath** 方法填充每个图形的内部区域。如果一个图形不是闭合的，那么 **FillPath** 方法将假设它是闭合的然后进行填充。下面的例子将填充一个由弧线、基数样条、字符串和饼图组成的路径。

```
myGraphics.FillPath(&mySolidBrush, &myGraphicsPath);
myGraphics.DrawPath(&myPen, &myGraphicsPath);
```

下图为该路径采用纯色画刷填充前后的样子。注意，采用 **DrawPath** 方法时，字符串描出轮廓，但是没有填充。而 **FillPath** 方法将字符串各字符内部进行了着色。



## 区域

区域指的是显示表面的一部分。区域可以简单（单个矩形）也可以复杂（由一个由多边形和闭合曲线组成）。下图显示了 2 个区域：一个由矩形构成，另一个由路径构成。



通常区域用来裁剪（Clipping）和进行点击测试（Hit Testing）。裁剪包括限制对显示区域的某个特定区域进行绘制，通常该区域为必须进行更新的部分。点击测试包括检查并判断当按下鼠标按钮时，光标是否位于屏幕的某个特定区域中。

您可以从矩形或路径建立区域。您也可以通过组合现有的区域来建立复杂的区域。**Region** 类提供下列用来组合区域的方法：**Intersect**、**Union**、**Xor**、**Exclude** 和 **Complement**。

两个区域的交集是指隶属于这两个区域的所有点的组合。联集是指属于其中一个或两个区域的所有点的组合。区域的补码（Complement）是指所有区域以外的点。下图将显示前图所说明的两个区域的交集和联集。



套用到区域组的 **Xor** 方法会产生一个区域，其中含有隶属于其中一个区域（而非两个）的所有点。套用到区域组合的 **Exclude** 方法会产生一个区域，其中含有第一个区域中但位于第二个区域以外的所有点。下图将显示因套用 **Xor** 和 **Exclude** 方法到本主题一开始所说明的两个区域时，所产生的区域。



若要绘制区域，您需要 **Graphics** 对象、**Brush** 对象和 **Region** 对象。**Graphics** 对象提供 **FillRegion** 方法，而 **Brush** 对象则是储存填充的特性，例如色彩或图样。下列范例将纯色填入区域中：

```
myGraphics.FillRegion(&mySolidBrush, &myRegion);
```

#### 裁剪

裁剪是指限制对特定区域进行绘制。下图将显示 "Hello" 字符串被裁剪为心形的区域。



区域可从路径建立，而路径中可包含字符串的字型外框，因此您可以使用外框文字来进行裁剪。下图将显示裁减为文字字符串内景的一组同心椭圆。

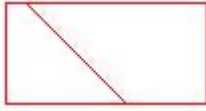


若要使用裁剪来进行绘制，请建立 **Graphics** 对象，调用其 **SetClip** 方法，然后调用相同 **Graphics** 对象的绘图方法。下例将绘制一条被矩形区域裁剪的直线：

```
Region myRegion(Rect(20, 30, 100, 50));
myGraphics.DrawRectangle(&myPen, 20, 30, 100, 50);
```

```
myGraphics.SetClip(&myRegion, CombineModeReplace);  
myGraphics.DrawLine(&myPen, 0, 0, 200, 200);
```

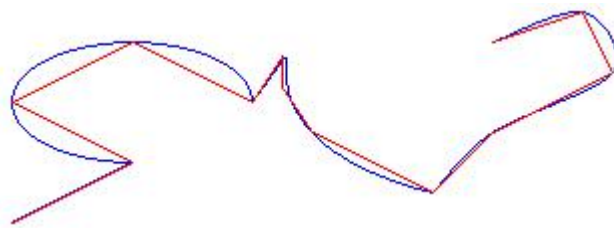
下图显示了一个矩形区域裁剪的直线。



## 路径平直化

对象存储一系列的线条和贝塞尔样条。您可以加入多种类型的曲线（椭圆、弧线、基数样条）到路径中，但是其中每条曲线在存储为路径之前都将转化为贝塞尔样条。路径平直化处理表示将路径中的每条贝塞尔样条转化为一一系列的直线。

若要平直化一条路径，需要调用 **GraphicsPath** 对象的 **Flatten** 方法。**Flatten** 方法有一个平直度的参数，表示平直化后的路径与原始路径的最大距离。下图显示了一条路径平直化处理前后情形：



## 线条和曲线的抗锯齿功能

使用 **GDI+** 绘制线条时，您必须提供线条的起始点和结束点，但不必提供该线条的个别像素相关信息。**GDI+** 是与显示驱动程序软件搭配使用，来决定必须开启哪些像素，才可以在特定显示装置上显示该线条。

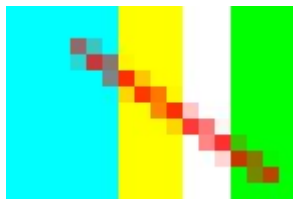
举这条从点 (4, 2) 到点 (16, 10) 的直色红线作为范例说明。假设坐标系统的原点在左上角，而度量单位为像素。同时我们也假设 **X** 轴指向右方，而 **Y** 轴则向下延伸。下图将显示在多重色彩背景上绘制的红色线条的放大画面。



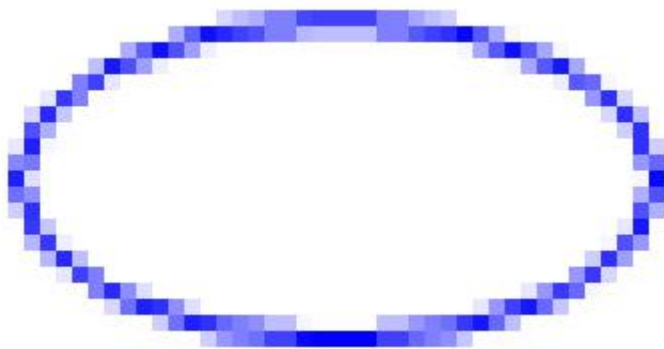
用来呈现线条的红色像素是不透明的。这条直线的像素全部都是不透明的。此类线条绘制方式所产生的线条会出现锯齿形的外观，看起来有点像阶梯。这种将线条绘制成阶梯状的技术称为锯齿 (**Aliasing**)；这种阶梯为理论线条的别名。



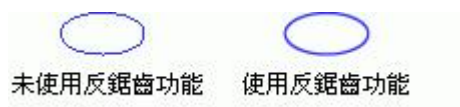
另一种更为复杂的绘制线条技术是同时使用透明的像素和不透明的像素。像素将设为纯红色或红色与背景色彩的混色，但是这需要视像素与线条接近的程度。此类绘制方式称为反锯齿 (Antialiasing)，可产生肉眼感觉更为平滑的线条。下图将显示某些像素将与背景混色以产生反锯齿线条。



反锯齿 (亦称为平滑化) 也可套用至曲线。下图将显示平滑椭圆形的放大画面。



下图将显示同一个椭圆形的实际大小，一个未使用反锯齿功能，另一个则使用反锯齿功能。



若要绘制使用平滑的线条和曲线，请先建立一个 **Graphics** 类对象，并传递 **SmoothingModeAntiAlias** 给它的 **SetSmoothingMode** 方法。然后，调用 **Graphics** 类的相同的绘图方法。

```
myGraphics.SetSmoothingMode(SmoothingModeAntiAlias);  
myGraphics.DrawLine(&myPen, 0, 0, 12, 8);
```

**SmoothingModeAntiAlias** 是 **SmoothingMode** 枚举中的元素之一。

## 图象、位图和图元文件

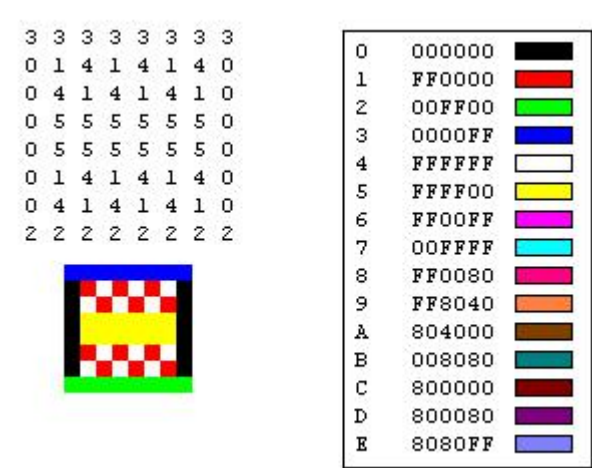
GDI+ 提供了 **Image** 类用于进行光栅图像 (位图) 和矢量图像 (图元文件)。**Bitmap** 类和 **Metafile** 类都是继承自 **Image** 类。**Bitmap** 类通过提供加载、储存和管理光栅图像的其它方法，增强了 **Image** 类的功能。**Metafile** 类别通过提供记录和检验矢量图像的其它方法，增强 **Image** 类别的功能。

位图类型

位图是由指定矩形数组中的每个像素色彩所组成的位数组。单个像素的组成位数将决定指派给该像素的色彩数目。例如，如果每个像素都是由 4 个字节成，则可将指定像素指派给 16 个不同的色彩之一 ( $2^4 = 16$ )。下表将显示一些范例，说明可指派给由指定位数所表示的像素的色彩数目。

每个象素的位数	可指派给像素的色彩数目
1	$2^1 = 2$
2	$2^2 = 4$
4	$2^4 = 16$
8	$2^8 = 256$
16	$2^{16} = 65,536$
24	$2^{24} = 16,777,216$

储存位图的磁盘文件通常都会包含一个或多个信息区块，其中存放每个像素位数、每行的像素数目和数组的行数等信息。这类文件可能也会包含颜色表（又称为调色板）。颜色表会将位图中的数字映射为特定色彩。下图显示一个放大的影像及其位图和色表。每个像素都是由 4 位数字来表示，因此色表中共有  $2^4 = 16$  个色彩。表格中的每一个色彩都是用 24 位数字来表示：8 位用来表示红色、8 位用来表示绿色，还有 8 位用来表示蓝色。这些数字是以十六进制格式显示：A = 10、B = 11、C = 12、D = 13、E = 14、F = 15。



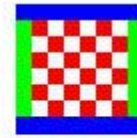
请看图片的第 3 行第 5 列的像素。位图中对应的数字为 1。我们可由色表得知 1 代表红色，因此该像素为红色。位图顶端列的所有项目都是 3。我们可由色表得知 3 表示蓝色，因此影像顶端列的所有像素都是蓝色。

**注意事项：**有些位图是以由下到上的格式储存；位图首行的数目表示的是图片末行的像素

将索引储存至色彩表的位图称为索引色位图。有些位图并不需要色表。例如，如果位图使用每像素 24 位，则该位图可储存色彩本身，而非存入色表的索引。下图显示直接储存色彩的位图（每像素 24 位），

而不是使用色表。该图同时显示对应影像的放大画面。在位图中，FFFFFF 表示白色、FF0000 代表红色、00FF00 为绿色，而 0000FF 则为蓝色。

```
0000FF 0000FF 0000FF 0000FF 0000FF 0000FF 0000FF 0000FF
00FF00 FF0000 FFFFFFFF FF0000 FFFFFFFF FF0000 FFFFFFFF 00FF00
00FF00 FFFFFFFF FF0000 FFFFFFFF FF0000 FFFFFFFF FF0000 00FF00
00FF00 FF0000 FFFFFFFF FF0000 FFFFFFFF FF0000 FFFFFFFF 00FF00
00FF00 FFFFFFFF FF0000 FFFFFFFF FF0000 FFFFFFFF FF0000 00FF00
00FF00 FF0000 FFFFFFFF FF0000 FFFFFFFF FF0000 FFFFFFFF 00FF00
00FF00 FFFFFFFF FF0000 FFFFFFFF FF0000 FFFFFFFF FF0000 00FF00
0000FF 0000FF 0000FF 0000FF 0000FF 0000FF 0000FF 0000FF
```



- 图片文件格式

您可以使用许多标准格式将位图储存在磁盘文件中。GDI+ 支持以下各种图片文件格式。

- 位图 (BMP)

位图是 Windows 用来储存设备无关和与应用程序无关的图片的标准格式。文件头决定了指定的位图文件的每个像素位数 (1、4、8、15、24、32 或 64)。常见的位图文件为每像素 24 位。通常 BMP 档不会被压缩，因此并不适合透过因特网传输。

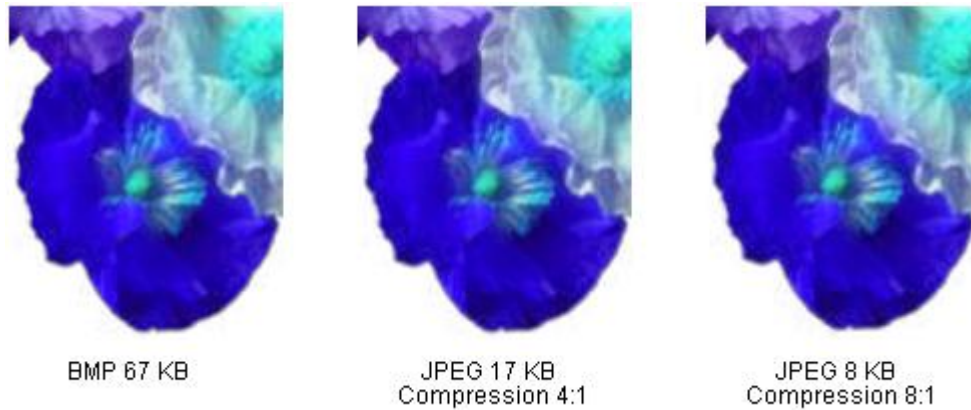
- 图象交换格式 (GIF)

GIF 是 Web 网页上常见的图片格式。GIF 适用于线条图形、具有实色区块的图片和色彩之间具有明显界线的图片。GIF 可被压缩而且不会在压缩过程中遗失任何信息；解压缩后的影像将和原始影像完全相同。GIF 的色彩可指定为透明，这样一来影像则可以显示该影像的 Web 网页作为背景色彩。GIF 影像的序列可储存在单一档案中，作为动画 GIF。GI 最多储存为每像素 8 位，这样便将它们限制在 256 色彩。

- 联合图像专家组(JPEG)

JPEG 是一种压缩结构，适用于自然景观图片，例如扫描的照片。有些信息可能会在压缩过程中丧失，但肉眼并无法看出变化。JPEG 可储存每像素 24 位，因此它们可以显示超过 1 千 6 百万个色彩。JPEG 不支持透明效果或动画。

您可以设定 JPEG 图片文件的压缩比，但压缩比越高 (档案越小)，便会丢失更多信息。通常肉眼无法辨识以 20:1 的压缩比所产生的图片和源文件之间的差异。下图将显示 BMP 图片和两个从该 BMP 图片压缩而来的 JPEG 图片。第一个 JPEG 的压缩比率为 4:1，第二个 JPEG 的压缩比率约为 8:1。



JPEG 压缩不适用于线条图形、实色区块和明显的界线。下图将显示一个 BMP 和两个 JPEG 及一个 GIF。这两个 JPEG 和 GIF 都是从 BMP 压缩而来的。GIF 的压缩比率为 4:1、较小的 JPEG 为 4:1，较大的 JPEG 为 8:3。请注意，GIF 中的线条间仍然出现明显边界，但 JPEG 中的边界似乎比较模糊。



JPEG 是一种压缩公式，而非文件格式。JPEG 档案交换格式（JFIF）才是通常来储存和传送图片的文件格式，它们是根据 JPEG 公式进行压缩。Web 浏览器显示的 JFIF 文件将使用.jpg 扩展名。

#### ○ 可交换图像文件 (Exif)

EXIF 是用于数字相机所拍摄的相片文件格式。EXIF 文件包含一个根据 JPEG 规范来压缩的图片。EXIF 文件同时还包含了照片信息（拍摄日期、快门速度、曝光时间等信息）和相机信息（制造商、型号等信息）。

#### ○ 可移植网络图片 (PNG)

PNG 格式保留了许多 GIF 格式的优点，同时提供比 GIF 更强大的功能。PNG 文件和 GIF 文件同样都不会在压缩过程中丧失任何信息。PNG 文件可储存每像素 8、24 或 48 位的色彩，以及每像素 1、2、4、8 或 16 位的灰阶。相较之下，GIF 档只能使用每像素 1、2、4 或 8 位。PNG 文件还可储存每个像素的 Alpha 值，指定该像素与背景色彩混合的程度。

PNG 改进了 GIF 渐进式显示影像的功能；当 PNG 收到透过网络联机传送的影像时，可以显示较佳的影像效果。PNG 文件可包含 Gamma 修正和色彩修正信息，这样一来可将影像正确地对应到各种不同的显示装置。

#### ○ 标记图像文件格式 (TIFF)

TIFF 是一种灵活且可扩充的格式，各种平台和图片处理应用程序都支持这种格式。TIFF 文件可储存每像素任意位数的图片，并可使用各种压缩算法。单一、多页的 TIFF 文件可储存数个影像。影像相关

信息（扫描仪制作、主机计算机、压缩类型、方向、每像素范例等等）也可储存在档案中，并可使用标记进行排列。**TIFF** 格式可按照需要（如情况允许而且必须增加新的标记）进行扩充。

## 图元文件

**GDI+** 提供 **Metafile** 类来记录和显示图元文件。图元文件又称为矢量图片，它是一种被储存为绘图命令和设定序列的图片。**Metafile** 对象中所记录的命令和设定可储存于内存或储存至文件或数据流。

**GDI+** 可显示储存为下列格式的图元文件：



**Windows 图元文件格式(WMF)**



**加强型图元文件格式(EMF)**



**EMF+**

**GDI+** 可记录 **EMF** 和 **EMF+** 格式的图元文件，但无法记录 **WMF** 格式的图元文件。

**EMF+** 是 **EMF** 的功能扩充，可用来储存 **GDI+** 记录。**EMF+** 格式有两种变化：**EMF+ Only** 和 **EMF+ Dual**。**EMF+ Only** 图元文件仅包含 **GDI+** 记录。此类图元文件可由 **GDI+** 显示，但无法使用 **GDI** 显示。**EMF+ Dual** 图元文件包含 **GDI+** 和 **GD** 记录。**EMF+ Dual** 图元文件的每个 **GDI+** 数据记录都会与替代 **GDI** 记录配对。此类图元文件可同时由 **GDI+** 或 **GDI** 显示。

下列范例将存储一个设置语句和一条绘图命令到一个磁盘文件中。注意这里构造一个 **Graphics** 对象，而且是通过在构造函数中传递一个 **MetaFile** 对象作为参数来构造的。

```
myMetafile = new Metafile(L"MyDiskFile.emf", hdc);
myGraphics = new Graphics(myMetafile);
    myPen = new Pen(Color(255, 0, 0, 200));
    myGraphics->SetSmoothingMode(SmoothingModeAntiAlias);
    myGraphics->DrawLine(myPen, 0, 0, 60, 40);
delete myGraphics;
delete myPen;
delete myMetafile;
```

如上例所示，**Graphics** 类是将指令和设置保存到 **Metafile** 对象的关键。任何对 **Graphics** 对象进行的方法调用均被记录到 **Metafile** 对象中。同样地，您可以设置任何 **Graphics** 对象的属性并将这个设置保存到 **Metafile** 对象中。在 **Graphics** 对象被删除或者超出作用域的时候，将停止记录。

下列范例显示先前储存为档案的图元文件。图元文件的左上角显示在 (100, 100)。

```
Graphics myGraphics(hdc);
Image myImage(L"MyDiskFile.emf");
myGraphics.DrawImage(&myImage, 100, 100);
```

下列范例记录多个属性设置（剪切区域、世界变换和平滑模式）到 **Metafile** 对象中。然后代码将记录多个绘图指令，然后将这些指令和设置都保存到一个磁盘文件中。

```

myMetafile = new Metafile(L"MyDiskFile2.emf", hdc);
myGraphics = new Graphics(myMetafile);
    myGraphics->SetSmoothingMode(SmoothingModeAntiAlias);
    myGraphics->RotateTransform(30);

    // Create an elliptical clipping region.
    GraphicsPath myPath;
    myPath.AddEllipse(0, 0, 200, 100);
    Region myRegion(&myPath);
    myGraphics->SetClip(&myRegion);

    Pen myPen(Color(255, 0, 0, 255));
    myGraphics->DrawPath(&myPen, &myPath);

    for(INT j = 0; j <= 300; j += 10)
    {
        myGraphics->DrawLine(&myPen, 0, 0, 300 - j, j);
    }
delete myGraphics;
delete myMetafile;

```

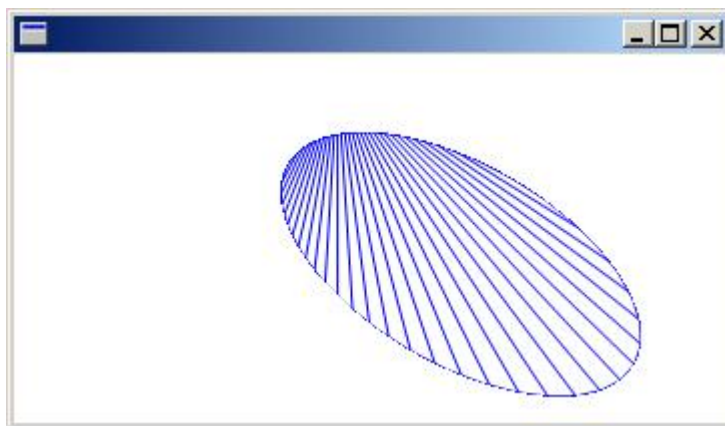
下列范例显示先前储存为档案的图元文件。

```

myGraphics = new Graphics(hdc);
myMetafile = new Metafile(L"MyDiskFile.emf");
myGraphics->DrawImage(myMetafile, 10, 10);

```

下图显示了前面代码的执行结果。注意观察抗锯齿效果、椭圆形裁剪和 30 度角旋转。



## 绘制、定位和复制图片

您可以使用 **Image** 类来加载和显示点阵图片（位图）和矢量图片（图元文件）。若要显示图片，您需要一个 **Graphics** 对象和一个 **Image** 对象。**Graphics** 对象提供 **DrawImage** 方法来接收 **Image** 对象作为它的一个参数。

下列代码示例会示范如何从 **Climber.jpg** 文件建立一个 **Image** 对象并显示此图片。图片左上角目标点 (10, 10) 是由第二个和第三个参数指定。

```
Image myImage(L"Climber.jpg");  
myGraphics.DrawImage(&myImage, 10, 10);
```

上面的代码执行效果如图：



您可以使用下列各种图形文件格式来建构 **Image** 对象：BMP、GIF、JPEG、EXIF、PNG、TIFF 和 ICON。

下列代码示例会示范如何从各种文件类型建立 **Image** 对象，并显示该图片。

```
Image myBMP(L"SpaceCadet.bmp");  
Image myEMF(L"Metafile1.emf");  
Image myGIF(L"Soda.gif");  
Image myJPEG(L"Mango.jpg");  
Image myPNG(L"Flowers.png");  
Image myTIFF(L"MS.tif");  
  
myGraphics.DrawImage(&myBMP, 10, 10);  
myGraphics.DrawImage(&myEMF, 220, 10);  
myGraphics.DrawImage(&myGIF, 320, 10);  
myGraphics.DrawImage(&myJPEG, 380, 10);  
myGraphics.DrawImage(&myPNG, 150, 200);  
myGraphics.DrawImage(&myTIFF, 300, 200);
```



**Image** 类提供 **Clone** 方法，可用来制作现有 **Image**、**Metafile** 或者 **Bitmap** 对象的副本。**Clone** 方法在 **Bitmap** 类中进行了重载，变体之一具有来源矩形参数，可指定您要复制的原始位图区域。下列程序代码范例会示范如何藉由复制现有 **Bitmap** 上半部来建立 **Bitmap**。然后同时显示这两个图象。

```
Bitmap* originalBitmap = new Bitmap(L"Spiral.png");
RectF sourceRect(
    0.0f,
    0.0f,
    (REAL)(originalBitmap->GetWidth()),
    (REAL)(originalBitmap->GetHeight())/2.0f);

Bitmap* secondBitmap = originalBitmap->Clone(sourceRect, PixelFormatDontCare);

myGraphics.DrawImage(originalBitmap, 10, 10);
myGraphics.DrawImage(secondBitmap, 100, 10);
```

结果如下：



### 裁剪和缩放图像

您可以使用 **Graphics** 类的 **DrawImage** 方法来绘制并定位图像。**DrawImage** 是一个重载方法，因此可使用许多种方法提供自变量给它。**DrawImage** 方法的其中一个变异可接收 **Image** 对象的地址和 **Rectangle** 对象的引用。矩形可指定绘图操作的目标位置；也就是说，它可以指定绘制图片的位置。如果目的矩形的大小和原始图片大小并不相同，该图片将缩放至适合目的矩形的大小。下列程序代码范例会示范如何绘制三次相同的图片：一次不使用缩放、一次使用放大，还有一次使用缩小：

```
Bitmap myBitmap(L"Spiral.png");
Rect expansionRect(80, 10, 2 * myBitmap.GetWidth(), myBitmap.GetHeight());
Rect compressionRect(210, 10, myBitmap.GetWidth() / 2,
    myBitmap.GetHeight() / 2);

myGraphics.DrawImage(&myBitmap, 10, 10);
myGraphics.DrawImage(&myBitmap, expansionRect);
myGraphics.DrawImage(&myBitmap, compressionRect);
```

上面代码执行效果如图：





有些 **DrawImage** 方法的变异具有来源矩形参数和目的矩形参数。来源矩形参数指定要绘制的原始图片区域。目的矩形指定用来绘制该图片区域的位置。如果目的矩形大小和来源矩形大小并不相同，图片将缩放至适合目的矩形的大小。

下列程序代码范例会示范如何从 **Runner.jpg** 文档构造一个 **Bitmap** 对象。整个图片从 (0, 0) 开始绘制，且不进行缩放。接着图片中的一小部分会绘制两次：一次使用缩小，另一次使用放大。

```
Bitmap myBitmap(L"Runner.jpg");

// The rectangle (in myBitmap) with upper-left corner (80, 70),
// width 80, and height 45, encloses one of the runner's hands.

// Small destination rectangle for compressed hand.
Rect destRect1(200, 10, 20, 16);

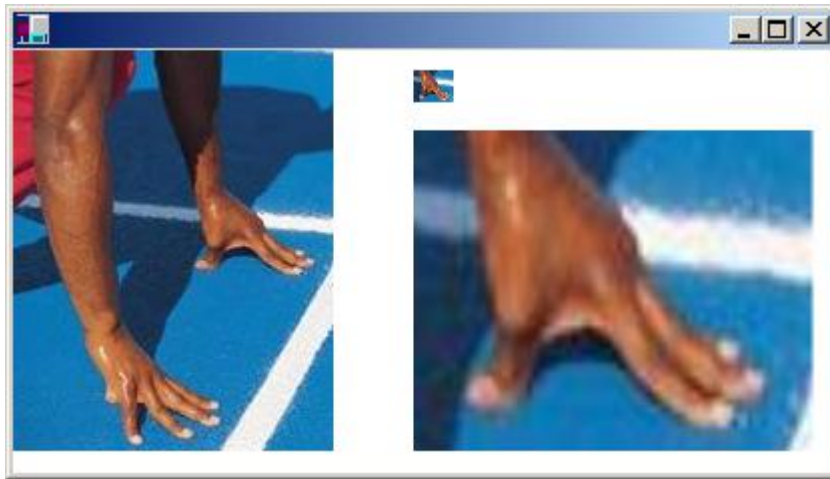
// Large destination rectangle for expanded hand.
Rect destRect2(200, 40, 200, 160);

// Draw the original image at (0, 0).
myGraphics.DrawImage(&myBitmap, 0, 0);

// Draw the compressed hand.
myGraphics.DrawImage(
    &myBitmap, destRect1, 80, 70, 80, 45, UnitPixel);

// Draw the expanded hand.
myGraphics.DrawImage(
    &myBitmap, destRect2, 80, 70, 80, 45, UnitPixel);
```

下列图标将显示未缩放的影像，以及经过缩小和放大的影像部分。



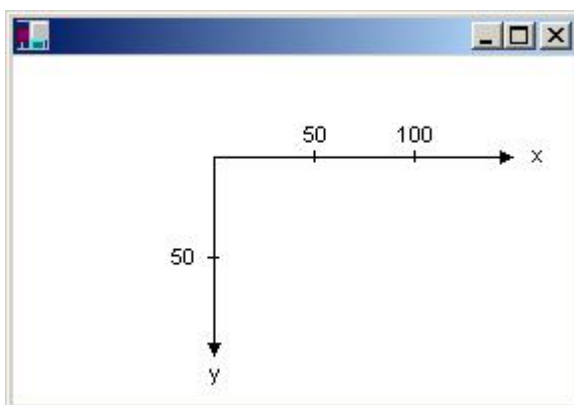
## 坐标系统和转换

GDI+ 提供世界变换和页面变换功能，可让您转换（旋转、缩放、平移等）所绘制的项目。这两种转换功能适用于各种坐标系统。

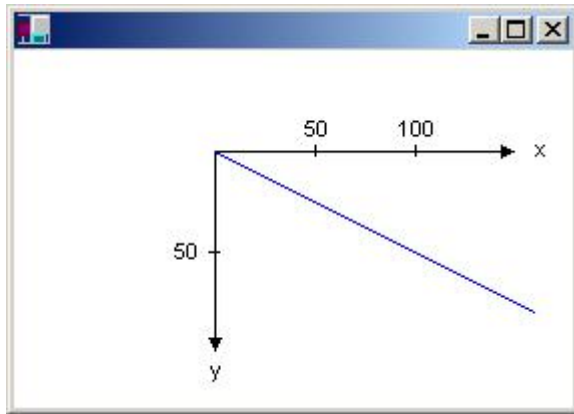
## 坐标系统类型

GDI+ 使用三个坐标空间：世界、页面和设备。世界坐标 (World Coordinate) 是用来制作特定绘图自然模型的坐标。页面坐标 (Page Coordinate) 则是指绘图接口（例如窗体或控件）使用的坐标系统。设备坐标 (Device Coordinate) 是在其上绘图的实际装置（例如屏幕或纸张）所使用的坐标。调用 `myGraphics.DrawLine(&myPen, 0, 0, 160, 80)` 时，传递至 `DrawLine` 方法的点  $(0, 0)$  和  $(160, 80)$  位于世界坐标空间。在 GDI+ 可以在屏幕上绘制线条之前，坐标会先经过一系列转换。“世界变换”会将世界坐标转换为页面坐标，而“页面变换”则是将页面坐标转换为设备坐标。

假设您想要在一个原点置于客户区域中心处而非左上角的坐标系统中工作。例如，您希望原点位于距离工作区左边缘 100 像素和距离工作区顶端 50 像素的位置。下图将显示此坐标系统。



调用 `myGraphics.DrawLine(&myPen, 0, 0, 160, 80)` 时，您会得到下图中所显示的线条。



在这三个坐标空间中，您线条的终点坐标位置如下：

世界坐标	(0, 0) - (160, 80)
页面坐标	(100, 50) - (260, 130)
设备坐标	(100, 50) - (260, 130)

请注意，页面坐标空间的原点一律位于工作区的左上角。此外，由于度量单位为像素，因此设备坐标和页面坐标是相同的。如果您将度量单位设为像素以外的单位（例如英吋），则设备坐标便与页面坐标不同。

将世界坐标映射到页面坐标称为“世界变换”，由 **Graphics** 对象来进行。上例中，全局转换是指在 X 方向平移 100 个单位和在 Y 方向平移 50 个单位。下列范例将设定 **Graphics** 对象的世界变换，然后使用该 **Graphics** 对象来绘制上图所显示的线条：

```
myGraphics.TranslateTransform(100.0f, 50.0f);
myGraphics.DrawLine(&myPen, 0, 0, 160, 80);
```

将页面坐标映射到设备坐标称为“页面变换”。**Graphics** 类提供了 4 种方法用于操作和检测页面变换：SetPageUnit、GetPageUnit、SetPageScale 和 GetPageScale。**Graphics** 类同时提供 2 个方法 GetDpiX 和 GetDpiY，用于获取显示设备每英吋的水平点数和垂直点数。

您可以使用 **Graphics** 类的 **SetPageUnit** 方法来指定一个度量单位。下列范例从 (0, 0) 到 (2, 1) 绘制一条线，其中点(2, 1)是指距离(0, 0)点右边 2 英吋和下方 1 英吋的位置：

```
myGraphics.SetPageUnit(UnitInch);
myGraphics.DrawLine(&myPen, 0, 0, 2, 1);
```

**🚧 注意事项：** 如果建构画笔时未指定画笔宽度，则上述范例将绘制出一条宽为一英吋的线条。您可以在 Pen 构造函数的第二个自变量中指定画笔宽度：

```
Pen myPen(Color(255, 0, 0, 0), 1/myGraphics.GetDpiX()).
```

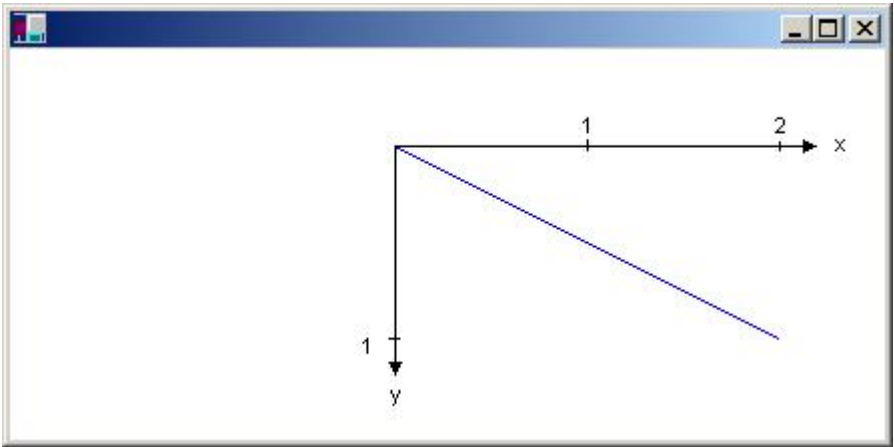
如果假设显示设备的水平方向每英吋有 96 个点，且其垂直方向每英吋有 96 个点，则上述范例的线条结束点会分别在三种坐标空间中使用下列坐标：

世界坐标	(0, 0) - (2, 1)
页面坐标	(0, 0) - (2, 1)
设备坐标	(0, 0) - (192, 96)

您可以结合世界和页面变换来达到各种效果。例如，假设您想要使用英吋做为度量单位，而且想要您的坐标系统的原点位于工作区左边缘的 2 英吋和工作区顶端的 1/2 英吋处。下列范例将设定 **Graphics** 对象的世界变换和页面变换，然后从(0, 0)到(2, 1)绘制出一条线：

```
myGraphics.TranslateTransform(2.0f, 0.5f);
myGraphics.SetPageUnit(UnitInch);
myGraphics.DrawLine(&myPen, 0, 0, 2, 1);
```

下列图标将显示该线条和坐标系统。



如果假设显示设备的水平方向每英吋有 96 个点，且其垂直方向每英吋有 96 个点，则上述范例的线条结束点会分别在三种坐标空间中使用下列坐标：

世界坐标	(0, 0) - (2, 1)
页面坐标	(2, 0.5) - (4, 1.5)
设备坐标	(192, 48) - (384, 144)

#### 以矩阵来表示转换

$m \times n$  阶矩阵是按照  $m$  行和  $n$  列排列的一组数字。下图将显示几种矩阵。

$$\begin{array}{ccc}
 \begin{bmatrix} 3 & 1 & 4 \\ 2 & 5 & 0 \end{bmatrix} & \begin{bmatrix} 1 & 3 \\ 2 & 8 \\ 0 & 4 \\ 5 & 6 \end{bmatrix} & \begin{bmatrix} 1.6 & 0.2 & 1.0 \end{bmatrix} \\
 2 \times 3 & 4 \times 2 & 1 \times 3 \\
 \\
 \begin{bmatrix} 2 & 0 \\ 0 & 3.5 \end{bmatrix} & \begin{bmatrix} 5 \\ 3 \end{bmatrix} & \begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 40 & 20 & 1 \end{bmatrix} \\
 2 \times 2 & 2 \times 1 & 3 \times 3
 \end{array}$$

您可以利用将各独立的元素相加的方法来实现两个大小相同的矩阵相加。下图将显示两个矩阵加法范例。

$$\begin{bmatrix} 5 & 4 \end{bmatrix} + \begin{bmatrix} 20 & 30 \end{bmatrix} = \begin{bmatrix} 25 & 34 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 \\ 0 & 2 \\ 1 & 3 \end{bmatrix} + \begin{bmatrix} 2 & 4 \\ 1 & 5 \\ 0 & 6 \end{bmatrix} = \begin{bmatrix} 3 & 4 \\ 1 & 7 \\ 1 & 9 \end{bmatrix}$$

$m \times n$  阶矩阵可乘以  $n \times p$  阶矩阵，其产生的结果便为  $m \times p$  阶矩阵。第一个矩阵的列数目必须和第二个矩阵的数据列数目相同。例如， $4 \times 2$  阶矩阵可乘以  $2 \times 3$  阶矩阵以产生  $4 \times 3$  阶矩阵。

平面上的点和矩阵的行列都可视为向量。例如， $(2, 5)$  是具有两个分量的向量，而  $(3, 7, 1)$  是具有三个分量的向量。两个向量的点乘定义如下：

$$(a, b) \bullet (c, d) = ac + bd$$

$$(a, b, c) \bullet (d, e, f) = ad + be + cf$$

例如， $(2, 3)$  和  $(5, 4)$  的点乘等于  $(2)(5) + (3)(4) = 22$ 。 $(2, 5, 1)$  和  $(4, 3, 1)$  的点乘等于  $(2)(4) + (5)(3) + (1)(1) = 24$ 。请注意，这两种向量的点乘是数字，而非另一个向量。此外，只有当这两个向量拥有相同的分量数目时，您才可以进行点乘运算。

假设  $A(i, j)$  是矩阵  $A$  第  $i$  行和第  $j$  列的项目。例如  $A(3, 2)$  是矩阵  $A$  第 3 行和第 2 列的项目。假设  $A$ 、 $B$  和  $C$  都是矩阵，而  $AB = C$ 。则  $C$  项目的计算结果如下：

$$C(i, j) = (A \text{ 的第 } i \text{ 行}) \bullet (B \text{ 的第 } j \text{ 列})$$

下图将显示几个矩阵乘法的范例。

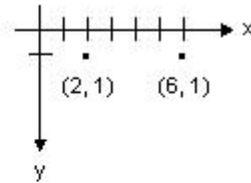
$$\begin{bmatrix} 2 & 3 \end{bmatrix}_{1 \times 2} \begin{bmatrix} 2 \\ 4 \end{bmatrix}_{2 \times 1} = \begin{bmatrix} 16 \end{bmatrix}_{1 \times 1}$$

$$\begin{bmatrix} 1 & 3 & 2 \end{bmatrix}_{1 \times 3} \begin{bmatrix} 1 & 0 \\ 2 & 4 \\ 5 & 1 \end{bmatrix}_{3 \times 2} = \begin{bmatrix} 17 & 14 \end{bmatrix}_{1 \times 2}$$

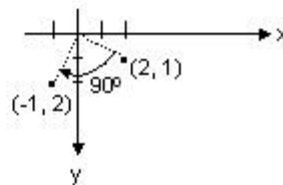
$$\begin{bmatrix} 2 & 5 & 1 \\ 4 & 3 & 1 \end{bmatrix}_{2 \times 3} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 2 & 3 & 1 \end{bmatrix}_{3 \times 3} = \begin{bmatrix} 4 & 13 & 1 \\ 6 & 9 & 1 \end{bmatrix}_{2 \times 3}$$

如果将平面中的点视为一个  $1 \times 2$  阶矩阵，您可以将该矩阵乘以  $2 \times 2$  阶矩阵以进行转换。下图将说明数个应用于点  $(2, 1)$  的转换。

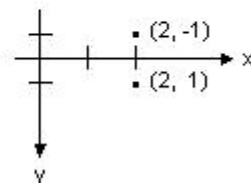
Scale  $\begin{bmatrix} 2 & 1 \end{bmatrix} \begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 6 & 1 \end{bmatrix}$



Rotate  $90^\circ$   $\begin{bmatrix} 2 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} = \begin{bmatrix} -1 & 2 \end{bmatrix}$

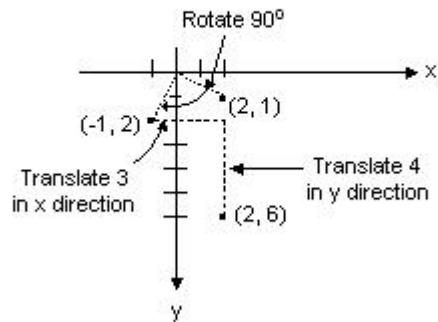


Reflect across x-axis  $\begin{bmatrix} 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} 2 & -1 \end{bmatrix}$



上图所显示的所有转换皆为线性转换。当然其它转换（例如平移）是非线性的，而且无法乘以  $2 \times 2$  矩阵来表示。假设您想要以点  $(2, 1)$  做为开始、将它旋转  $90^\circ$  度、在 X 方向平移 3 个单位和在 Y 方向平移 4 个单位。您可以使用矩阵乘法之后再使用矩阵加法来达到这个目的。

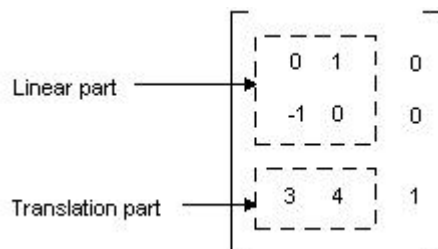
$$\begin{bmatrix} 2 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} + \begin{bmatrix} 3 & 4 \end{bmatrix} = \begin{bmatrix} 2 & 6 \end{bmatrix}$$



在线性变换（乘以  $2 \times 2$  阶矩阵）后再进行平移变换（加上一个  $1 \times 2$  阶矩阵）的变换称为仿射变换。另一种将仿射变换储存为矩阵组（其中一个做为线性部分，另一个做为平移部分）的方法是将整个转换储存在  $3 \times 3$  矩阵中。若要执行这项工作，平面的其中一点必须以虚设的第三坐标储存于  $1 \times 3$  阶矩阵中。常用的技巧是让所有第三坐标都为 1，例如，点  $(2, 1)$  便以矩阵  $\begin{bmatrix} 2 & 1 & 1 \end{bmatrix}$  表示。下图将显示仿射变换（旋转  $90^\circ$ ；在 X 方向平移 3 个单位、在 Y 方向平移 4 个单位）将乘以  $3 \times 3$  矩阵来表示。

$$\begin{bmatrix} 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 3 & 4 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 6 & 1 \end{bmatrix}$$

例中，点  $(2, 1)$  将映射到点  $(2, 6)$ 。请注意， $3 \times 3$  阶矩阵的第三个列包含数字  $0, 0, 1$ 。仿射变换的  $3 \times 3$  阶矩阵一律都是这样。第 1 列和第 2 列中的六个数字非常重要。矩阵的左上  $2 \times 2$  部分代表转换的线性部分，而第三个数据列的前两个项目则代表转换。



在 GDI+ 中，您可以将仿射变换储存在 **Matrix** 对象中。由于用来表示仿射转换的矩阵第三列永远为  $(0, 0, 1)$ ，因此在建立 **Matrix** 对象时，您只能指定前两个数据行中的六个数字。语句 `Matrix myMatrix = new Matrix(0, 1, -1, 0, 3, 4)` 表示构造上图中显示的矩阵。

## • 复合转换

复合转换由一系列转换构成，一个接着另一个。举下表的矩阵和转换为例：

矩阵 A	旋转 $90^\circ$ 度
矩阵 B	在 X 方向缩放 2 个系数
矩阵 C	在 Y 方向平移 3 个单位

如果从点 (2, 1) 开始 (以矩阵 [2 1 1] 代表), 并且乘以 A, 然后乘以 B, 再乘以 C, 则点 (2, 1) 将会依照列出的顺序经历三种转换。

$$[2 \ 1 \ 1]ABC = [-2 \ 5 \ 1]$$

如果不想将复合转换的三个部分分别储存在三个矩阵中, 您可以同时乘以 A、B 和 C 以取得储存整个复合转换的单个 3×3 阶矩阵。假设  $ABC = D$ 。则任何一个点乘以 D 所得的结果都等于任何一个点乘以 A、然后 B、然后 C。

$$[2 \ 1 \ 1]D = [-2 \ 5 \ 1]$$

下图将显示矩阵 A、B、C 和 D。

$$\begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 3 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ -2 & 0 & 0 \\ 0 & 3 & 1 \end{bmatrix}$$

A                  B                  C                  =                  D

因为乘以单个转换矩阵可形成复合转换的矩阵, 这表示仿射转换的任何序列都可以储存在单个 **Matrix** 对象中。

#### 警告

复合转换的顺序非常重要。通常, 旋转、然后缩放、然后转换与缩放、然后旋转、然后在转换并不相同。同样的, 矩阵乘法的顺序也很重要。一般而言,  $ABC$  与  $BAC$  并不相同。

**Matrix** 类提供几个方法, 用来构建复合转换: **Multiply**、**Rotate**、**RotateAt**、**Scale**、**Shear** 和 **Translate**。下列范例建立复合转换矩阵, 它会先旋转 30 度、然后在 Y 方向缩放 2 个系数, 然后在 X 方向平移 5 个单位:

```
Matrix myMatrix;
myMatrix.Rotate(30.0f);
myMatrix.Scale(1.0f, 2.0f, MatrixOrderAppend);
myMatrix.Translate(5.0f, 0.0f, MatrixOrderAppend);
```

下列图标将显示该矩阵。

$$\begin{bmatrix} \cos 30^\circ & 2\sin 30^\circ & 0 \\ -\sin 30^\circ & 2\cos 30^\circ & 0 \\ 5 & 0 & 1 \end{bmatrix} \approx \begin{bmatrix} 0.866 & 1.0 & 0 \\ -0.5 & 1.73 & 0 \\ 5 & 0 & 1 \end{bmatrix}$$

## 全局和局部转换

全局转换应用于特定 **Graphics** 对象绘制的所有项目。相对地, 区域转换只应用于要绘制的特定项目。如果要建立全局转换, 请建立 **Graphics** 对象, 然后调用它的 **Graphics::SetTransform** 方法。



**Graphics::SetTransform** 方法处理一个与 **Graphics** 对象相关的 **Matrix** 对象。储存在 **Matrix** 对象的变换称为**世界变换**。世界变换可以是一个简单的仿射变换或者一系列复杂的仿射变换，但是不管有多复杂，世界变换都只存储在单独一个 **Matrix** 对象中。

**Graphics** 类提供几个方法，用来建置复合的全局转换：**Graphics::MultiplyTransform**、**Graphics::RotateTransform**、**Graphics::ScaleTransform** 和 **Graphics::TranslateTransform**。下列范例会绘制两次椭圆形：一次是在建立世界变换之前，一次是在建立之后。转换会先在 Y 方向缩放 0.5 个系数，然后在 X 方向转换 50 个单位，然后再旋转 30 度。

```
myGraphics.DrawEllipse(&myPen, 0, 0, 100, 50);
myGraphics.ScaleTransform(1.0f, 0.5f);
myGraphics.TranslateTransform(50.0f, 0.0f, MatrixOrderAppend);
myGraphics.RotateTransform(30.0f, MatrixOrderAppend);
myGraphics.DrawEllipse(&myPen, 0, 0, 100, 50);
```

下图显示了原始椭圆和变形后的椭圆：



下图将显示转换所使用的矩阵。

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 50 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos 30^\circ & \sin 30^\circ & 0 \\ -\sin 30^\circ & \cos 30^\circ & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos 30^\circ & \sin 30^\circ & 0 \\ -0.5 \sin 30^\circ & 0.5 \cos 30^\circ & 0 \\ 50 \cos 30^\circ & 50 \sin 30^\circ & 1 \end{bmatrix}$$

Scale                  Translate                  Rotate

#### 注意事项

在上述范例中，椭圆形会绕着坐标系统的原点旋转，该原点位于工作区的左上角。该操作产生的结果和在椭圆形中心点旋转并不相同。

局部转换应用于要绘制的特定项目。例如，**GraphicsPath** 对象的 **GraphicsPath::Transform** 方法可用来转换该路径的数据点。下列范例绘制一个未转换的矩形和一个经过旋转变换的路径（假设没有世界变换）。

```
Matrix myMatrix;
myMatrix.Rotate(45.0f);
myGraphicsPath.Transform(&myMatrix);
myGraphics.DrawRectangle(&myPen, 10, 10, 100, 50);
myGraphics.DrawPath(&myPen, &myGraphicsPath);
```

您可以结合全局转换和局部转换来达到各种效果。例如，您可以使用全局转换来修订坐标系统，并可使用局部转换来旋转和缩放新坐标系统所绘制的对象。

假设您想要使用一个坐标系统，其原点距离工作区左边缘 200 像素和工作区顶端 150 像素处。此外，假设您想使用像素做为度量单位，其 **x** 轴指向右方，且其 **y** 轴指向上方。预设坐标系统的 **y** 轴指向下方，因此您必须沿着水平轴执行反射。下图将显示此类反射的矩阵。

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

接下来，假设您必须执行向右 200 单位和向下 150 单位的平移。

下列范例会建立上述说明的坐标系统，使用的方法是设定 **Graphics** 对象的全局转换。

```
Matrix myMatrix(1.0f, 0.0f, 0.0f, -1.0f, 0.0f, 0.0f);
myGraphics.SetTransform(&myMatrix);
myGraphics.TranslateTransform(200.0f, 150.0f, MatrixOrderAppend);
```

下列程序代码（放到上述范例的结束处）建立单一矩形组成的路径，其中该矩形的左上角位于新坐标系统的原点。该矩形将填入色彩两次，其中一次并未使用区域转换，另外一次则使用区域转换。区域转换包括水平缩放 2 个系数，随后进行 30 度的旋转。

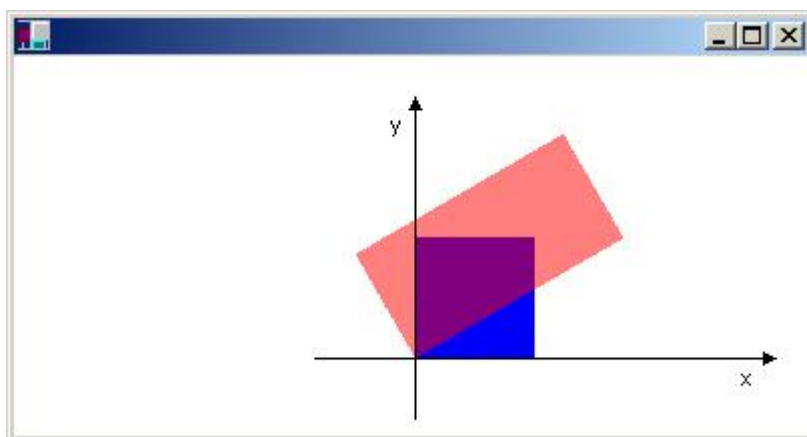
```
// Create the path.
GraphicsPath myGraphicsPath;
Rect myRect(0, 0, 60, 60);
myGraphicsPath.AddRectangle(myRect);

// Fill the path on the new coordinate system.
// No local transformation
myGraphics.FillPath(&mySolidBrush1, &myGraphicsPath);

// Transform the path.
Matrix myPathMatrix;
myPathMatrix.Scale(2, 1);
myPathMatrix.Rotate(30, MatrixOrderAppend);
myGraphicsPath.Transform(&myPathMatrix);

// Fill the transformed path on the new coordinate system.
myGraphics.FillPath(&mySolidBrush2, &myGraphicsPath);
```

下图将显示新的坐标系统和两个矩形。



## 图形容器

图片状态（裁剪区域、变形、质量设置等）存储于 **Graphics** 对象中。GDI+ 允许您采用一个容器临时替换或者扩充一个 **Graphics** 对象的状态。调用 **Graphics** 对象的 **BeginContainer** 方法开始一个容器，直至调用 **EndContainer** 方法终止一个容器。在这期间，您对于属于该容器的 **Graphics** 对象进行的任何状态改变都不会覆盖掉 **Graphics** 对象现存的状态。

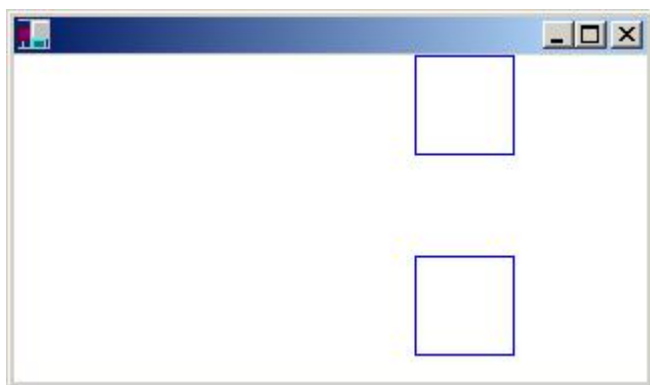
下面的例子在一个 **Graphics** 对象中创建一个容器。**Graphics** 对象的世界变换是向右平移 200 个单位，而容器的世界变换是向下平移 100 个单位。

```
myGraphics.TranslateTransform(200.0f, 0.0f);

myGraphicsContainer = myGraphics.BeginContainer();
    myGraphics.TranslateTransform(0.0f, 100.0f);
    myGraphics.DrawRectangle(&myPen, 0, 0, 50, 50);
myGraphics.EndContainer(myGraphicsContainer);

myGraphics.DrawRectangle(&myPen, 0, 0, 50, 50);
```

注意，在前面的例子中，置于 **BeginContainer** 和 **EndContainer** 调用之间的语句 `myGraphics.DrawRectangle(&myPen, 0, 0, 50, 50)` 产生的矩形和 **EndContainer** 调用之后的同样的语句产生了不同的 2 个矩形。应用于 `DrawRectangle` 调用的两种变换（水平平移 200 个单位和垂直平移 100 个单位）都在容器内部进行。下图显示了这两个矩形。



容器可以与容器嵌套。下面的例子在一个 **Graphics** 对象内创建一个容器，紧接着在前一个容器中再创建另一个容器。**Graphics** 容器的世界变换是沿 X 轴平移 100 个单位然后沿 Y 轴平移 80 个单位。第一个容器的世界变换是旋转 30 度角。第二个容器的世界变换是沿 X 轴缩放 2 倍单位。在第二个容器中调用 **DrawEllipse** 方法。

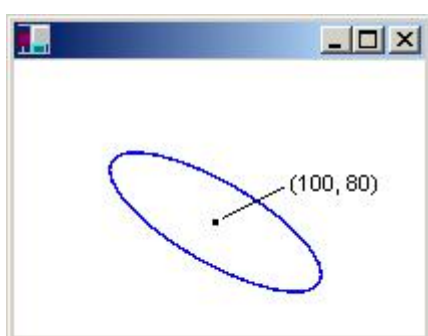
```
myGraphics.TranslateTransform(100.0f, 80.0f, MatrixOrderAppend);

container1 = myGraphics.BeginContainer();
    myGraphics.RotateTransform(30.0f, MatrixOrderAppend);

    container2 = myGraphics.BeginContainer();
        myGraphics.ScaleTransform(2.0f, 1.0f);
        myGraphics.DrawEllipse(&myPen, -30, -20, 60, 40);
    myGraphics.EndContainer(container2);

myGraphics.EndContainer(container1);
```

下图显示该椭圆。



注意，所有的变换都应用于第二个（最里面的）容器里面的 **DrawEllipse** 调用。同时注意，变换的顺序：首先缩放，然后是旋转，然后才是平移。最内层的变换最先被执行，最外层的变换最迟执行。

在容器内部（在 **BeginContainer** 和 **EndContainer** 调用之间）可以对 **Graphics** 对象的任意属性进行设置。例如，可以在容器内设置一个剪切区域。容器内的任何绘图操作都将被该容器的剪切区域所裁剪，同时也会被外层容器以及 **Graphics** 对象本身的剪切区域所裁剪。

关于属性的讨论已经很深入了，嵌套容器中的世界变换和裁剪区域。其他属性则由嵌套容器临时替换。例如，当您在一个容器中通过 **SmoothingModeAntiAlias** 设置了平滑模式后，在容器内部的任何绘图操作都将采用这个平滑模式，但是在 **EndContainer** 调用之后的绘图操作则采用 **BeginContainer** 调用之前的平滑模式。

再举个容器内部一个 **Graphics** 对象的世界变换的例子，假设立希望绘制一只眼睛然后将它放到若干脸部的不同位置。下面的例子将在坐标系统原点位置绘制一只眼睛。

```
void DrawEye(Graphics* pGraphics)
{
    GraphicsContainer eyeContainer;

    eyeContainer = pGraphics->BeginContainer();

    Pen myBlackPen(Color(255, 0, 0, 0));
    SolidBrush myGreenBrush(Color(255, 0, 128, 0));
    SolidBrush myBlackBrush(Color(255, 0, 0, 0));

    GraphicsPath myTopPath;
    myTopPath.AddEllipse(-30, -50, 60, 60);

    GraphicsPath myBottomPath;
    myBottomPath.AddEllipse(-30, -10, 60, 60);

    Region myTopRegion(&myTopPath);
    Region myBottomRegion(&myBottomPath);

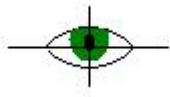
    // Draw the outline of the eye.
    // The outline of the eye consists of two ellipses.
    // The top ellipse is clipped by the bottom ellipse, and
    // the bottom ellipse is clipped by the top ellipse.
    pGraphics->SetClip(&myTopRegion);
    pGraphics->DrawPath(&myBlackPen, &myBottomPath);
    pGraphics->SetClip(&myBottomRegion);
    pGraphics->DrawPath(&myBlackPen, &myTopPath);

    // Fill the iris.
    // The iris is clipped by the bottom ellipse.
    pGraphics->FillEllipse(&myGreenBrush, -10, -15, 20, 22);

    // Fill the pupil.
    pGraphics->FillEllipse(&myBlackBrush, -3, -7, 6, 9);

    pGraphics->EndContainer(eyeContainer);
}
```

下图所示为该坐标系统和眼睛：



上面例子中的 `DrawEye` 函数接受一个 **Graphics** 对象地址，然后立即在该 **Graphics** 对象中创建一个容器。容器使得 `DrawEye` 函数内部的代码调用与在执行过程中的属性设置相脱离。例如，`DrawEys` 函数内部代码设置了 **Graphics** 对象的剪切区域，但是当 `DrawEys` 将控制权返回给调用例程后，裁剪区域将恢复为 `DrawEye` 调用之前的状态。

下面的代码演示了绘制 3 个椭圆（脸），每个内部都有一只眼睛。

```
// Draw an ellipse with center at (100, 100).
myGraphics.TranslateTransform(100.0f, 100.0f);
myGraphics.DrawEllipse(&myBlackPen, -40, -60, 80, 120);

// Draw the eye at the center of the ellipse.
DrawEye(&myGraphics);

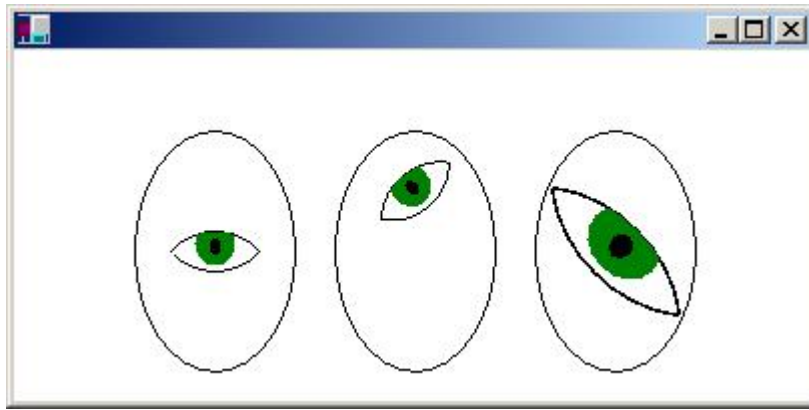
// Draw an ellipse with center at 200, 100.
myGraphics.TranslateTransform(100.0f, 0.0f, MatrixOrderAppend);
myGraphics.DrawEllipse(&myBlackPen, -40, -60, 80, 120);

// Rotate the eye 40 degrees, and draw it 30 units above
// the center of the ellipse.
myGraphicsContainer = myGraphics.BeginContainer();
    myGraphics.RotateTransform(-40.0f);
    myGraphics.TranslateTransform(0.0f, -30.0f, MatrixOrderAppend);
    DrawEye(&myGraphics);
myGraphics.EndContainer(myGraphicsContainer);

// Draw a ellipse with center at (300.0f, 100.0f).
myGraphics.TranslateTransform(100.0f, 0.0f, MatrixOrderAppend);
myGraphics.DrawEllipse(&myBlackPen, -40, -60, 80, 120);

// Stretch and rotate the eye, and draw it at the
// center of the ellipse.
myGraphicsContainer = myGraphics.BeginContainer();
    myGraphics.ScaleTransform(2.0f, 1.5f);
    myGraphics.RotateTransform(45.0f, MatrixOrderAppend);
    DrawEye(&myGraphics);
myGraphics.EndContainer(myGraphicsContainer);
```

结果如下：



上例中，所有的椭圆都是采用 `DrawEllipse(&myBlackPen, -40, -60, 80, 120)` 调用绘制的，该语句将在坐标系统的原点位置绘制椭圆。通过设置 **Graphics** 对象的世界变换来将椭圆的工作区从左上角移开。该语句使得第一个椭圆在(100,100)的位置居中，使得第二个椭圆在第一个椭圆中心位置向右平移了 100 个单位。同样地，第三个椭圆的中心又在第二个椭圆的基础上向右平移了 100 个单位。

上例中的容器用于相对于给定椭圆中心对眼睛进行平移。第一只绘制于椭圆中心位置的眼睛没有任何变换，因此没有将 `DrawEye` 调用放到容器中。第二只眼睛旋转 40 度后又相对于椭圆中心向上平移了 30 个单位，因此 `DrawEye` 函数调用和变换等操作是在容器内部进行的。第三个眼睛进行了缩放、旋转操作然后绘制于椭圆中心位置。与第二只眼睛类似的，`DrawEye` 函数调用和变换等操作也是置于容器内部的。

# 使用 GDI+

下列主题描述如何在 C++ 编程语言中使用 GDI+ API。

## 使用入门

本章节将说明如何在一个标准的 C++ Windows 应用程序中开始使用 GDI+。画线和文字的操作是您能在 GDI+ 中完成的最简单的操作。下面将讨论如何做。

## 绘制线条

若要在 GDI+ 中绘制线条, 你需要一个 **Graphics** 对象、一个 **Pen** 对象和一个 **Color** 对象。**Graphics** 对象提供 **Graphics::DrawLine** 方法, **Pen** 对象存储诸如颜色和宽度等线条属性。**Pen** 对象地址作为参数之一传递给 **Graphics::DrawLine** 方法。

下面的程序绘制一条从(0, 0)到(200, 100)的线条, 包含 3 个函数: *WinMain*、*WndProc* 和 *OnPaint*。*WinMain* 和 *WndProc* 函数为大多数 Windows 应用程序提供最基本的代码。在 *WndProc* 函数中没有任何 GDI+ 代码。而 *WinMain* 函数中有少量的 GDI+ 代码, 也就是对最基本的 *GdiplusStartup* 和 *GdiplusShutdown* 的调用。实际创建 **Graphics** 对象并进行线条绘制的 GDI+ 代码是在 *OnPaint* 函数中。

*OnPaint* 函数接受一个设备上下文句柄, 并将它传递给 **Graphics** 构造函数。传给 **Pen** 构造函数的参数之一是一个 **Color** 对象, 它的构造函数同时又传入 4 个参数, 分别表示颜色的 Alpha 值、红、绿、蓝四个分量。Alpha 分量决定了颜色的透明度; 0 表示完全透明, 255 表示完全不透明。传给 **DrawLine** 方法的四个数字表示线段的起点(0, 0)和终点(200, 100)。

```
#define UNICODE
#include <windows.h>
#include <gdiplus.h>
using namespace Gdiplus;

VOID OnPaint(HDC hdc)
{
    Graphics graphics(hdc);
    Pen pen(Color(255, 0, 0, 255));
    graphics.DrawLine(&pen, 0, 0, 200, 100);
}

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

INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE, PSTR, INT iCmdShow)
{
    HWND hWnd;
    MSG msg;
```



```

WNDCLASS      wndClass;
GdiplusStartupInput gdiplusStartupInput;
ULONG_PTR     gdiplusToken;

// Initialize GDI+.
GdiplusStartup(&gdiplusToken, &gdiplusStartupInput, NULL);

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

RegisterClass(&wndClass);

hWnd = CreateWindow(
    TEXT("GettingStarted"), // window class name
    TEXT("Getting Started"), // window caption
    WS_OVERLAPPEDWINDOW,    // window style
    CW_USEDEFAULT,          // initial x position
    CW_USEDEFAULT,          // initial y position
    CW_USEDEFAULT,          // initial x size
    CW_USEDEFAULT,          // initial y size
    NULL,                   // parent window handle
    NULL,                   // window menu handle
    hInstance,              // program instance handle
    NULL);                  // creation parameters

ShowWindow(hWnd, iCmdShow);
UpdateWindow(hWnd);

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

GdiplusShutdown(gdiplusToken);
return msg.wParam;

```

```

} // WinMain

LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
    WPARAM wParam, LPARAM lParam)
{
    HDC          hdc;
    PAINTSTRUCT  ps;

    switch(message)
    {
    case WM_PAINT:
        hdc = BeginPaint(hWnd, &ps);
        OnPaint(hdc);
        EndPaint(hWnd, &ps);
        return 0;
    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
} // WndProc

```

请注意 *WinMain* 函数中 **GdiplusStartup** 的调用。传给该函数的第一个参数是一个 `ULONG_PTR` 类型的地址。**GdiplusStartup** 在该变量中填入一个标记，而该标记将用于后来的 **GdiplusShutdown** 函数。**GdiplusStartup** 函数的第二个参数是一个 `GdiplusStartupInput` 结构体的地址。前面的例子基于默认的 **GdiplusStartupInput** 构造函数来设置该结构体成员变量。

## 绘制字符串

上面的主题阐述了如何在一个 **Windows** 应用程序中采用 **GDI+** 绘制线条。如果要绘制字符串，只需要将上面的 *OnPaint* 函数用以下函数代替即可：

```

VOID OnPaint(HDC hdc)
{
    Graphics    graphics(hdc);
    SolidBrush  brush(Color(255, 0, 0, 255));
    FontFamily  fontFamily(L"Times New Roman");
    Font        font(&fontFamily, 24, FontStyleRegular, UnitPixel);
    PointF      pointF(10.0f, 20.0f);

    graphics.DrawString(L"Hello World!", -1, &font, pointF, &brush);
}

```

上面的代码创建了几个 GDI+ 对象。Graphics 对象提供 DrawString 方法来执行实际的绘制操作。SolidBrush 对象指定字符串的颜色。

FontFamily 构造函数接受单个字符串参数来指定其字体家族。**FontFamily** 对象的地址作为第一个参数传递给 Font 构造函数。Font 构造函数的第二个参数指定字体大小，第三个参数则指定字体样式。**FontStyleRegular** 是 FontStyle 枚举成员之一，在 Gdiplusenums.h 头文件中有定义。Font 构造函数的最后一个参数表示字体尺寸采用像素来度量。**UnitPixel** 是 Unit 枚举成员之一。

传递给 **Graphics::DrawString** 方法的第一个参数是一个宽位字符串的地址。第二个参数等于 -1，表示该字符串以 Null 作为终止符。（如果该字符串不是以 Null 终止，那么第二个参数就需要给出该字符串的宽位字符数）。第三个参数是 **Font** 对象的地址。第四个参数是一个 **PointF** 对象的引用，它指定了字符串将被绘制的位置。最后一个参数是一个 **Brush** 对象的地址，它用于指定字符串的颜色。

### 使用钢笔绘制线条和形状

Graphics 类提供了如下列表所示的多种绘图方法：

-  Graphics::DrawLine 方法
-  Graphics::DrawRectangle 方法
-  Graphics::DrawEllipse 方法
-  Graphics::DrawArc 方法
-  Graphics::DrawPath 方法
-  Graphics::DrawCurve 方法
-  Graphics::DrawBezier 方法

上面的绘图方法中你都需要传入一个 Pen 对象的地址。

下面主题中间详细讲述 Pen 的用法：

### 使用钢笔绘制线条和矩形

若要绘制线条，需要 Graphics 对象和 Pen 对象。Graphics 对象提供 DrawLine 方法，而 Pen 对象则存储线条的特征，如颜色和宽度。

下面的示例绘制一条从 (20, 10) 到 (300, 100) 的直线。假定 *graphics* 是一个已经存在的 Graphics 对象。

```
Pen pen(Color(255, 0, 0, 0));  
graphics.DrawLine(&pen, 20, 10, 300, 100);
```

第一条语句使用 Pen 类构造函数创建黑色钢笔。传递给 Pen 构造函数的参数之一是 Color 对象。用于创建 Color 对象的值 (255、0、0、0) 对应于颜色的 alpha、红色、绿色和蓝色分量。这些值定义不透明的黑色钢笔。

下面的示例绘制一个左上角位于 (10, 10) 的矩形。该矩形的宽度为 100，高度为 50。传递给 Pen 构造函数的第二个参数表明钢笔的宽度为 5 个像素。

```
Pen blackPen(Color(255, 0, 0, 0), 5);
stat = graphics.DrawRectangle(&blackPen, 10, 10, 100, 50);
```

绘制该矩形时，钢笔以矩形边界为中心线居中。因为钢笔的宽度是 5，矩形的边被绘制为 5 个像素宽，因此 1 个像素绘制在边界本身，2 个像素绘制在内侧，2 个像素绘制在外侧。有关钢笔对齐方式的详细信息，请参见[如何：设置钢笔的宽度和对齐方式](#)。

下面的插图显示结果矩形。虚线表明当钢笔的宽度为 1 个像素时矩形被绘制的位置。矩形左上角的放大视图显示黑色粗线条以这些虚线为中心线居中。



### 设置钢笔的宽度和对齐方式

在创建 Pen 时，可将笔的宽度作为参数之一提供给构造函数。还可用 Pen 类的 Width 属性更改笔的宽度。

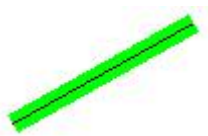
理论线条的宽度为 0。当绘制一条 1 个像素宽的线条时，像素以理论线条为中心线分布。下面的代码示例绘制一段线条两次：一次用宽度为 1 的黑色钢笔，一次用宽度为 10 的绿色钢笔。

```
Pen blackPen(Color(255, 0, 0, 0), 1);
Pen greenPen(Color(255, 0, 255, 0), 10);
stat = greenPen.SetAlignment(PenAlignmentCenter);

// Draw the line with the wide green pen.
stat = graphics.DrawLine(&greenPen, 10, 100, 100, 50);

// Draw the same line with the thin black pen.
stat = graphics.DrawLine(&blackPen, 10, 100, 100, 50);
```

下面的插图显示结果线条。绿色钢笔绘制的像素以理论线条为中心线分布。



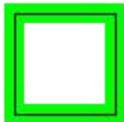
下面的代码示例绘制一个矩形两次：一次用宽度为 1 的黑色钢笔，一次用宽度为 10 的绿色钢笔。将 **PenAlignmentCenter**（PenAlignment 枚举成员之一）传给 **SetAlignment** 方法，表示以指定用绿色钢笔绘制的像素以矩形边界为中心分布。下面的插图显示结果矩形。

```
Pen blackPen(Color(255, 0, 0, 0), 1);
Pen greenPen(Color(255, 0, 255, 0), 10);
stat = greenPen.SetAlignment(PenAlignmentCenter);

// Draw the rectangle with the wide green pen.
stat = graphics.DrawRectangle(&greenPen, 10, 100, 50, 50);

// Draw the same rectangle with the thin black pen.
stat = graphics.DrawRectangle(&blackPen, 10, 100, 50, 50);
```

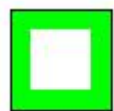
下面的插图显示结果线条。绿色钢笔绘制的像素以理论矩形为中心线分布，该矩形由黑色像素表示。



可通过将上述代码示例中的第三条语句修改为以下语句来更改绿色钢笔的对齐方式：

```
stat = greenPen.SetAlignment(PenAlignmentInset);
```

现在，绿色宽线条中的像素出现在矩形的内侧，如下面的插图所示。



## 绘制具有线帽的线条

可用形状多样的线帽来绘制线条的起点或终点。GDI+ 支持多种线帽，如圆形、方形、菱形和箭头。

您可为线条的起点、线条的终点或虚线的短划线指定线帽，分别称为起始线帽、终止线帽和短划线线帽。

下面的示例绘制一端为箭头线帽、另一端为圆形线帽的线条。

```
Pen pen(Color(255, 0, 0, 255), 8);
stat = pen.SetStartCap(LineCapArrowAnchor);
stat = pen.SetEndCap(LineCapRoundAnchor);
stat = graphics.DrawLine(&pen, 20, 175, 300, 175);
```

下面的插图显示产生的线条：



**LineCapArrowAnchor** 和 **LineCapRoundAnchor** 是 **LineCap** 枚举变量之一。

### 联接线条

线条联接点是由两条端点相交或重叠的线条构成的共同区域。**GDI+** 提供了三种线条联接样式：斜接、斜切和圆。线条联接样式是 **Pen** 类的一个属性。当为 **Pen** 对象指定线条联接样式时，联接样式将应用到任何使用该笔绘制的 **GraphicsPath** 对象中的所有连接线条。

你可以通过 **Pen** 类的 **SetLineJoin** 方法指定一种联接样式。下面的示例演示水平线条和垂直线条之间的斜切线条联接。

```
GraphicsPath path;
Pen penJoin(Color(255, 0, 0, 255), 8);

path.StartFigure();
path.AddLine(Point(50, 200), Point(100, 200));
path.AddLine(Point(100, 200), Point(100, 250));

penJoin.SetLineJoin(LineJoinBevel);
graphics.DrawPath(&penJoin, &path);
```

下面的插图演示产生的斜切线条联接的结果。



在上例中，传递给 **SetLineJoin** 方法的值（**LineJoinBevel**）是枚举成员之一。

### 绘制自定义虚线

**GDI+** 提供 **DashStyle** 枚举中列出的几种虚线样式。如果这些标准的虚线样式不能满足需求，则可创建自定义的虚线模式。

若要绘制自定义虚线，请将短划线和间距的长度放在一个数组中，并作为 **SetDashPattern** 方法的参数之一传递给 **Pen** 对象。下面的示例绘制了一条基于 {5, 2, 15, 4} 数组的自定义的虚线。如果将数组元素乘以钢笔的宽度 5，可得到 {25, 10, 75, 20}。显示的短划线的长度在 25 和 75 之间交替，间距的长度在 10 和 20 之间交替。

```
REAL dashValues[4] = {5, 2, 15, 4};
Pen blackPen(Color(255, 0, 0, 0), 5);
blackPen.SetDashPattern(dashValues, 4);
stat = graphics.DrawLine(&blackPen, Point(5, 5), Point(405, 5));
```

下面的插图显示结果虚线。请注意，最后一段短划线不得不短于 25 个单位，以便线条的终点落在 (405, 5) 上。



### 绘制用纹理填充的线条

您可以不用纯色绘制线条，而用纹理绘制线条。若要绘制带有纹理的直线和曲线，请先创建 **TextureBrush** 对象，并将该 **TextureBrush** 对象传递给 **Pen** 构造函数。与该纹理刷相关联的位图用于平铺平面（不可见），然后当钢笔绘制直线或曲线时，钢笔的笔划揭开平铺纹理的某些像素。

下面的示例从文件 **Texture1.jpg** 创建 **Bitmap** 对象。位图用于构造 **TextureBrush** 对象，而 **TextureBrush** 对象用于构造 **Pen** 对象。对 **Graphics::DrawImage** 的调用将绘制该位图，位图的左上角位于 (0, 0)。对 **Graphics::DrawImage** 的调用将使用 **Pen** 对象绘制带纹理的椭圆。

```
Image      image(L"Texture1.jpg");
TextureBrush tBrush(&image);
Pen        texturedPen(&tBrush, 30);

graphics.DrawImage(&image, 0, 0, image.GetWidth(), image.GetHeight());
graphics.DrawEllipse(&texturedPen, 100, 20, 200, 100);
```

下面的插图显示该位图和带纹理的椭圆。



### 使用画笔填充形状

**GDI+ Brush** 对象用于填充闭合形状的内部。**GDI+** 定义多种填充样式：纯色、阴影图案、图像纹理和颜色渐变。

## 用纯色填充形状

若要用纯色填充形状，请创建 **SolidBrush** 对象，然后将该 **SolidBrush** 对象作为一个参数传递给 **Graphics** 类的某个填充方法。下面的示例演示如何用红色填充椭圆。

```
SolidBrush solidBrush(Color(255, 255, 0, 0));  
stat = graphics.FillEllipse(&solidBrush, 0, 0, 100, 60);
```

在上面的例子中，**SolidBrush** 构造函数采用一个 **Color** 对象作为其仅有的参数。**Color** 构造函数使用的值分别表示颜色的 **alpha**、红色、绿色和蓝色分量。这些值中的每一个都必须在 0 到 255 之间。第一个 255 表示颜色是完全不透明的，第二个 255 表示红色分量的强度达到最大。两个零表示绿色和蓝色分量的强度为 0。

传递给 **FillEllipse** 方法的四个数 (0, 0, 100, 60) 指定该椭圆的外接矩形的位置和尺寸。该矩形的左上角位于 (0, 0)，宽度为 100，高度为 60。

## 用阴影图案填充形状

阴影图案由两种颜色组成：一种是背景色，一种是在背景上形成图案的线条的颜色。若要用阴影图案填充闭合的形状，请使用 **HatchBrush** 对象。下面的示例演示如何用阴影图案填充椭圆：

```
HatchBrush hBrush(HatchStyleHorizontal, Color(255, 255, 0, 0),  
    Color(255, 128, 255, 255));  
stat = graphics.FillEllipse(&hBrush, 0, 0, 100, 60);
```

下面的插图显示已填充的椭圆。



**HatchBrush** 构造函数带有三个参数：阴影样式、阴影线颜色和背景颜色。阴影样式参数可以为 **HatchStyle** 枚举中的任何值。在 **HatchStyle** 枚举中有 50 多个元素；在下面的列表中显示了其中的几个元素：

-  **HatchStyleHorizontal**
-  **HatchStyleVertical**
-  **HatchStyleForwardDiagonal**
-  **HatchStyleBackwardDiagonal**
-  **HatchStyleCross**
-  **HatchStyleDiagonalCross**

## 用图像纹理填充形状

通过使用 **Image** 类和 **TextureBrush** 类，可用纹理填充闭合的形状。



下面的示例用图像填充椭圆。该代码构造 **Image** 对象，然后将该 **Image** 对象的地址作为参数传递给 **TextureBrush** 构造函数。第三条语句缩放图像，第四条语句用缩放后图像的重复副本填充椭圆。

```
Image image(L"ImageFile.jpg");
TextureBrush tBrush(&image);
stat = tBrush.SetTransform(&Matrix(75.0/640.0, 0.0f, 0.0f,
    75.0/480.0, 0.0f, 0.0f));
stat = graphics.FillEllipse(&tBrush, Rect(0, 150, 150, 250));
```

在前面的代码中，**SetTransform** 方法设置了在绘制图像之前应用到该图像的转换。假定原始图像的宽度为 640 像素，高度为 480 像素。该转换通过设置水平和垂直缩放值将图像缩小到 75 x 75。

#### 注意

在前面的示例中，图像大小为 75 x 75、椭圆大小为 150 x 250。因为图像比它所填充的椭圆小，所以图像平铺在椭圆上。平铺意味着图像水平和垂直重复排列，直到到达形状的边界。有关平铺的更多信息，请参见[在形状中平铺图像](#)。

### 在形状中平铺图像

正如瓷砖可一块挨一块地放置以覆盖地面一样，矩形图像也可一个挨一个地放置以填充（平铺）形状。若要平铺形状的内部，请使用纹理画笔。在构造 **TextureBrush** 对象时，传递给构造函数的一个参数是 **Image** 对象的地址。在使用纹理画笔为形状的内部涂色时，将用该图像的重复副本填充该形状。

**TextureBrush** 对象的覆盖模式属性确定图像在矩形网格中重复时的定向方式。既可让网格中所有平铺图像的方向都相同，也可让平铺图像在相邻网格间翻转。可水平或垂直翻转，也可同时进行水平和垂直翻转。下面的示例演示用不同类型的翻转进行平铺。

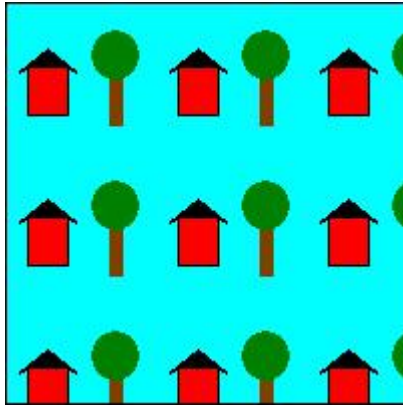
#### • 平铺图像

该示例使用下面的 75×75 图像来平铺 200×200 的矩形。



```
Image image(L"HouseAndTree.gif");
TextureBrush tBrush(&image);
Pen blackPen(Color(255, 0, 0, 0));
stat = graphics.FillRectangle(&tBrush, Rect(0, 0, 200, 200));
stat = graphics.DrawRectangle(&blackPen, Rect(0, 0, 200, 200));
```

下面的插图显示如何使用图像平铺此矩形。请注意，所有的平铺图像都采用相同的方向，没有翻转。

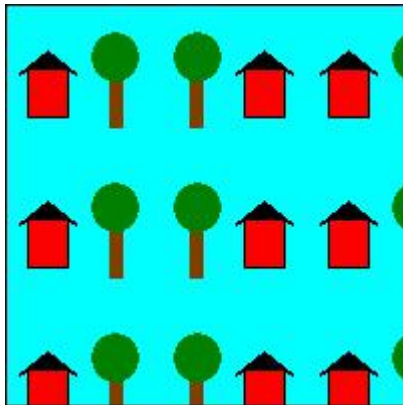


- **平铺时水平翻转图像**

该示例使用相同的  $75 \times 75$  图像来填充  $200 \times 200$  的矩形。覆盖模式被设置为水平翻转图像。

```
Image image(L"HouseAndTree.gif");
TextureBrush tBrush(&image);
Pen blackPen(Color(255, 0, 0, 0));
stat = tBrush.SetWrapMode(WrapModeTileFlipX);
stat = graphics.FillRectangle(&tBrush, Rect(0, 0, 200, 200));
stat = graphics.DrawRectangle(&blackPen, Rect(0, 0, 200, 200));
```

下面的插图显示如何使用图像平铺此矩形。请注意，在给定行中从一个平铺图像移到下一个时，图像将水平翻转。



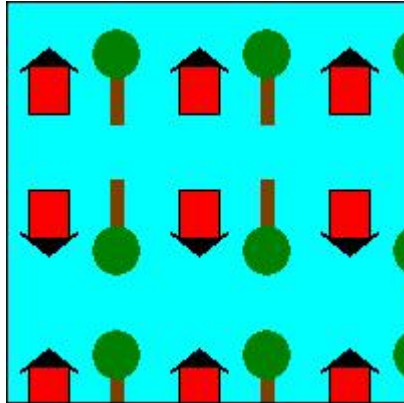
- **平铺时垂直翻转图像**

该示例使用相同的  $75 \times 75$  图像来填充  $200 \times 200$  的矩形。覆盖模式被设置为垂直翻转图像。

```
Image image(L"HouseAndTree.gif");
TextureBrush tBrush(&image);
Pen blackPen(Color(255, 0, 0, 0));
stat = tBrush.SetWrapMode(WrapModeTileFlipY);
stat = graphics.FillRectangle(&tBrush, Rect(0, 0, 200, 200));
```

```
stat = graphics.DrawRectangle(&blackPen, Rect(0, 0, 200, 200));
```

下面的插图显示如何使用图像平铺此矩形。请注意，在给定行中从一个平铺图像移到下一个时，图像将垂直翻转。

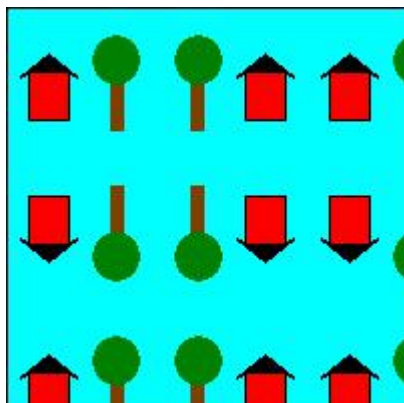


- 平铺时水平和垂直翻转图像

该示例使用相同的  $75 \times 75$  图像来平铺  $200 \times 200$  的矩形。覆盖模式被设置为同时在水平和垂直方向翻转图像。

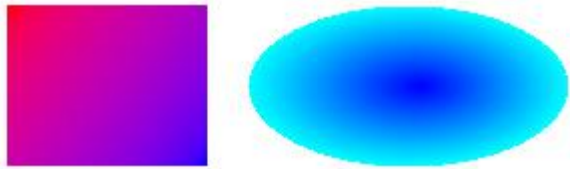
```
Image image(L"HouseAndTree.gif");
TextureBrush tBrush(&image);
Pen blackPen(Color(255, 0, 0, 0));
stat = tBrush.SetWrapMode(WrapModeTileFlipXY);
stat = graphics.FillRectangle(&tBrush, Rect(0, 0, 200, 200));
stat = graphics.DrawRectangle(&blackPen, Rect(0, 0, 200, 200));
```

下面的插图显示如何使用图像平铺此矩形。请注意，在给定行中从一个平铺图像移到下一个时，图像将水平翻转；在给定列中从一个平铺图像移到下一个时，图像将被垂直翻转。



## 用渐变色填充形状

可借助渐变画笔用渐变的颜色填充形状。GDI+ 提供线性渐变画刷和路径渐变画刷。下面的插图显示用线性渐变画笔填充矩形，用路径渐变画笔填充椭圆。



更多信息，请参考[采用渐变画刷填充图形](#)。

## 使用图像、位图和图元文件

GDI+ 提供用于处理光栅图像的 **Bitmap** 类和用于处理矢量图像的 **Metafile** 类。**Bitmap** 和 **Metafile** 类都是从 **Image** 类继承的。**Bitmap** 类通过提供加载、储存和管理光栅图像的其它方法，增强了 **Image** 类的功能。**Metafile** 类别通过提供记录和检验矢量图像的其它方法，增强 **Image** 类别的功能。

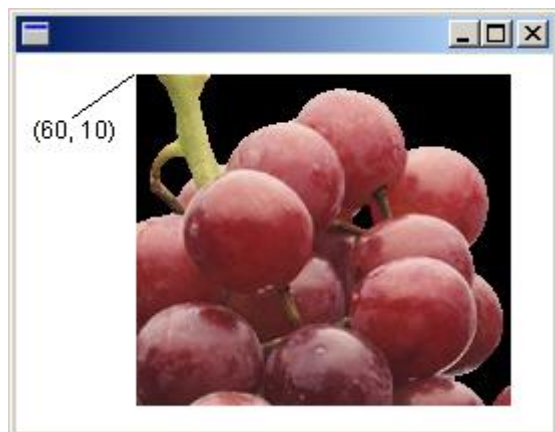
### 加载和显示位图

若要从文件中加载位图并将其显示在屏幕上，则需要 **Image** 对象和 **Graphics** 对象。将文件的名称传递给 **Image** 构造函数。创建 **Image** 对象之后，将此 **Image** 对象的地址传递给 **Graphics** 对象的 **DrawImage** 方法。

下面的示例从 JPEG 文件创建 **Image** 对象，然后绘制该位图，其左上角位于 (60, 10)。

```
Image image(L"Grapes.jpg");  
graphics.DrawImage(&image, 60, 10);
```

下面的插图显示在指定位置绘制的位图。



**Image** 类提供用于加载和显示光栅图像和矢量图像的基本方法。继承自 **Image** 类的 **Bitmap** 类提供了更多的专门方法用于加载、显示和处理光栅图像。比如，你可以从一个图标句柄(HICON)构造一个 **Bitmap** 对象。

下面的例子中，获取一个图标的句柄，然后利用该句柄创建一个 **Bitmap** 对象。通过将 **Bitmap** 对象的地址传递给 **Graphics** 对象的 **DrawImage** 方法来显示该图标。

```
HICON hIcon = LoadIcon(NULL, IDI_APPLICATION);  
Bitmap bitmap(hIcon);  
graphics.DrawImage(&bitmap, 10, 10);
```

### 加载和显示图元文件

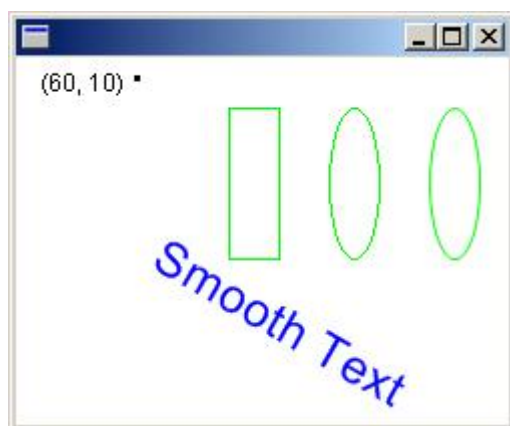
**Image** 类提供用于加载和显示光栅图像和矢量图像的基本方法。继承自 **Image** 类的 **Metafile** 类提供了更多的专门方法用于加载、显示和处理矢量图像。

要在屏幕上显示矢量图像（图元文件），则需要 **Metafile** 对象和 **Graphics** 对象。将文件名称（或流的指针）传递给 **Metafile** 构造函数。在创建 **Metafile** 对象之后，将此 **Metafile** 对象传递给 **Graphics** 对象的 **DrawImage** 方法。

下面的示例从 **EMF** 文件（增强型图元文件）创建 **Metafile** 对象，然后绘制该图像，其左上角位于 (60, 10)。

```
Image image(L"SampleMetafile.emf");  
graphics.DrawImage(&image, 60, 10);
```

下面的插图显示在指定位置绘制的图元文件。



### 记录图元文件

**Metafile** 类继承自 **Image** 类，它允许你记录一系列绘图命令。这些命令记录可以存储在内存中、文件中或者流中。**Metafile** 可以包含矢量图像、光栅图像和文本。

下面的例子创建一个 **Metafile** 对象。代码用 **Metafile** 对象来记录一连串的图像命令，然后将它保存为以 **SampleMetafile.emf** 命名的文件中。需要注意的是将设备场景句柄传递给 **Metafile** 构造函数，然后又将 **Metafile** 对象的地址传递给 **Graphics** 构造函数。当 **Graphics** 对象作用域消失的时候记录结束（命令记录并被存储在文件中）。最末两行通过创建一个 **Graphics** 对象然后将 **Metafile** 对象的地址传递给 **Graphics** 对象的 **DrawImage** 方法来显示这个图元文件。注意，代码使用同一个 **Metafile** 来记录和显示这个图元文件。

```
Metafile metafile(L"SampleMetafile.emf", hdc);
{
    Graphics graphics(&metafile);
    Pen greenPen(Color(255, 0, 255, 0));
    SolidBrush solidBrush(Color(255, 0, 0, 255));

    // Add a rectangle and an ellipse to the metafile.
    graphics.DrawRectangle(&greenPen, Rect(50, 10, 25, 75));
    graphics.DrawEllipse(&greenPen, Rect(100, 10, 25, 75));

    // Add an ellipse (drawn with antialiasing) to the metafile.
    graphics.SetSmoothingMode(SmoothingModeHighQuality);
    graphics.DrawEllipse(&greenPen, Rect(150, 10, 25, 75));

    // Add some text (drawn with antialiasing) to the metafile.
    FontFamily fontFamily(L"Arial");
    Font font(&fontFamily, 24, FontStyleRegular, UnitPixel);

    graphics.SetTextRenderingHint(TextRenderingHintAntiAlias);
    graphics.RotateTransform(30.0f);
    graphics.DrawString(L"Smooth Text", 11, &font,
        PointF(50.0f, 50.0f), &solidBrush);
} // End of recording metafile.

// Play back the metafile.
Graphics playbackGraphics(hdc);
playbackGraphics.DrawImage(&metafile, 200, 100);
```

#### 注意

为了记录一个图元文件，你必须基于一个 **Metafile** 对象来构造 **Graphics** 对象。在 **Graphics** 对象被删除或者作用域实效后对于图元文件的记录才终止。

一个图元文件包含了它自身的图像属性，这些属性在记录图元文件的时候由 **Graphics** 对象定义。在记录过程中您对 **Graphics** 对象的任何属性进行的设置都将被保存到图元文件中。在您显示该图元文件的时候，将按照您存储的这些属性来进行绘图。

下面的例子中，假定在记录图元文件的过程中平滑模式设置为 **SmoothingModeNormal**。即便 **Graphics** 对象在回放时将平滑模式设置为了 **SmoothingModeHighQuality**，但是图元文件在显示的时候仍然是按照记录中的 **SmoothingModeNormal** 设置。此时采用的是记录时的平滑模式，而非回放前设置的平滑模式，这点很重要。

```
graphics.SetSmoothingMode(SmoothingModeHighQuality);  
graphics.DrawImage(&meta, 0, 0);
```

### 剪裁和缩放图像

**Graphics** 类提供几个 **DrawImage** 方法，其中的某些方法包含可用于裁切和缩放图像的源矩形参数和目标矩形参数。

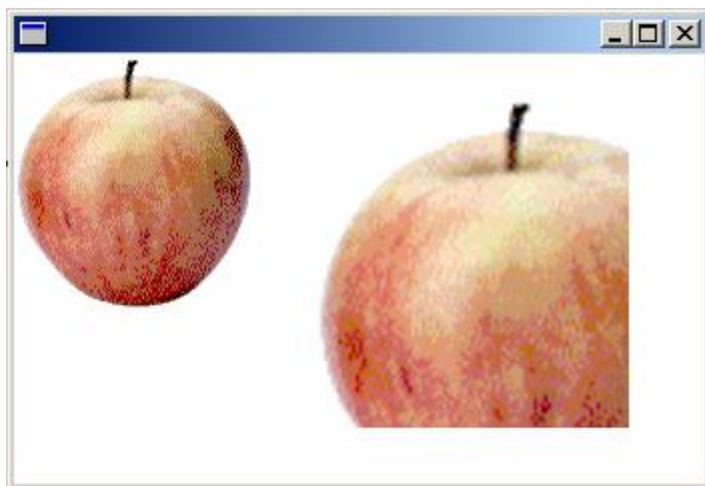
下面的示例从磁盘文件 **Apple.gif** 构造 **Image** 对象。该代码按照其原始尺寸绘制整个苹果图像。然后，该代码调用 **Graphics** 对象的 **DrawImage** 方法，以便在大于原始苹果图像的目标矩形中绘制该苹果图像的一部分。

**DrawImage** 方法通过查看源矩形来确定要绘制苹果的哪部分，这由第三、第四、第五和第六个参数来指定。在本例中，苹果在宽度和高度上均被裁切为原始尺寸的 **75%**。

**DrawImage** 方法通过查看目标矩形来确定在何处绘制裁切后的苹果以及裁切后的苹果大小，这由第二个参数指定。在本例中，目标矩形在宽度和高度上都比原始图像大 **30%**。

```
Image image(L"Apple.gif");  
UINT width = image.GetWidth();  
UINT height = image.GetHeight();  
// Make the destination rectangle 30 percent wider and  
// 30 percent taller than the original image.  
// Put the upper-left corner of the destination  
// rectangle at (150, 20).  
Rect destinationRect(150, 20, 1.3 * width, 1.3 * height);  
// Draw the image unaltered with its upper-left corner at (0, 0).  
graphics.DrawImage(&image, 0, 0);  
// Draw a portion of the image. Scale that portion of the image  
// so that it fills the destination rectangle.  
graphics.DrawImage(  
    &image,  
    destinationRect,  
    0, 0,          // upper-left corner of source rectangle  
    0.75 * width,  // width of source rectangle  
    0.75 * height, // height of source rectangle  
    UnitPixel);
```

下面的插图显示原始苹果和缩放、裁切后的苹果。



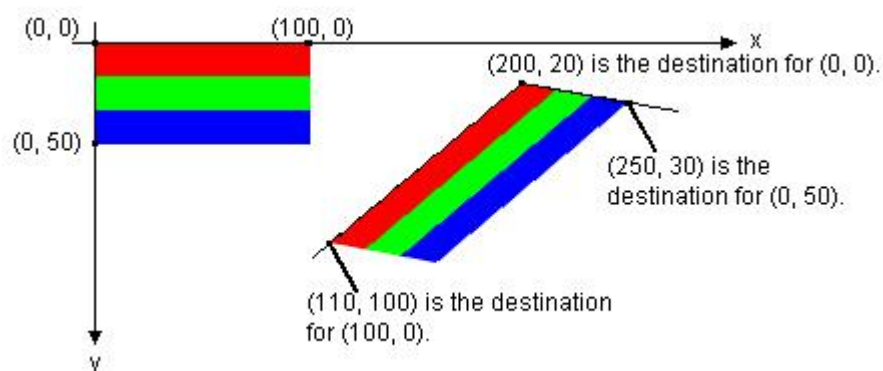
### 旋转、反射和扭曲图像

通过指定原始图像的左上角、右上角和左下角的目标点可旋转、反射和扭曲图像。这三个目标点确定将原始矩形图像映射为平行四边形的仿射变换。（原始图片的左下角将映射到平行四边形的第四个角，这可以通过给出的 3 个点求出。）

例如，假设原始图像是一个矩形，其左上角、右上角和左下角分别位于  $(0, 0)$ 、 $(100, 0)$  和  $(0, 50)$ 。现在假设我们将这三个点按以下方式映射到目标点。

原始点	目标点
左上角 $(0, 0)$	$(200, 20)$
右上角 $(100, 0)$	$(110, 100)$
左下角 $(0, 50)$	$(250, 30)$

下面的插图显示原始图像以及映射为平行四边形的图像。原始图像已被扭曲、反射、旋转和平移。沿着原始图像上边缘的  $x$  轴被映射到通过  $(200, 20)$  和  $(110, 100)$  的直线。沿着原始图像左边缘的  $y$  轴被映射到通过  $(200, 20)$  和  $(250, 30)$  的直线。



下面的示例生成第一个插图所示的图像。



```

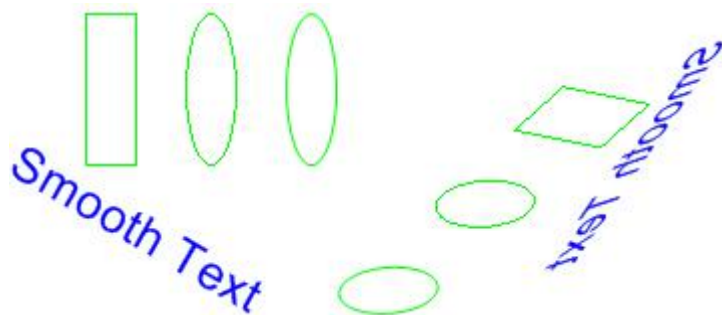
Point destinationPoints[] = {
    Point(200, 20), // destination for upper-left point of original
    Point(110, 100), // destination for upper-right point of original
    Point(250, 30)}; // destination for lower-left point of original
Image image(L"Stripes.bmp");
// Draw the image unaltered with its upper-left corner at (0, 0).
graphics.DrawImage(&image, 0, 0);
// Draw the image mapped to the parallelogram.
graphics.DrawImage(&image, destinationPoints, 3);

```

下面的插图显示应用到照片图像的类似变换。








下面的插图显示应用到图元文件的类似变换。



### 缩放时使用插值模式控制图像质量

Graphics 对象的插值模式会影响 GDI+ 缩放（拉伸和收缩）图像的方式。InterpolationMode 枚举定义了几种插值模式，其中一些模式显示在下面的列表中：

-  InterpolationModeNearestNeighbor
-  InterpolationModeBilinear
-  InterpolationModeHighQualityBilinear
-  InterpolationModeBicubic
-  InterpolationModeHighQualityBicubic

若要拉伸图像，原始图像中的每个像素都必须映射为较大图像中的一组像素。若要收缩图像，必须将原始图像中成组的像素映射为较小图像中单个的像素。执行这些映射的算法的效果决定缩放后图像的质量。生成优质缩放图像的算法往往需要更长的处理时间。在上面的列表中，**NearestNeighbor** 是质量最差的模式，**HighQualityBicubic** 是质量最好的模式。

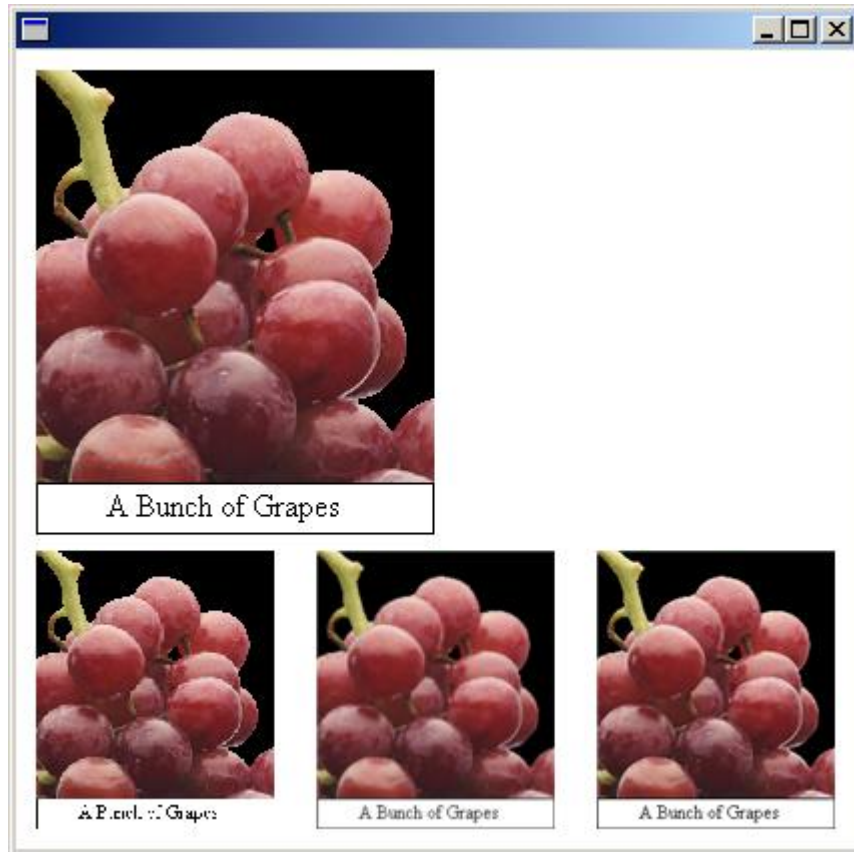
若要设置插值模式，请将 **InterpolationMode** 枚举的一个成员传递给 **Graphics** 对象的 **SetInterpolationMode** 方法。

下面的示例绘制一个图像，然后用三种不同的插值模式收缩图像。

```
Image image(L"GrapeBunch.bmp");
UINT width = image.GetWidth();
UINT height = image.GetHeight();
// Draw the image with no shrinking or stretching.
graphics.DrawImage(
    &image,
    Rect(10, 10, width, height), // destination rectangle
    0, 0, // upper-left corner of source rectangle
    width, // width of source rectangle
    height, // height of source rectangle
    UnitPixel);
// Shrink the image using low-quality interpolation.
graphics.SetInterpolationMode(InterpolationModeNearestNeighbor);
graphics.DrawImage(
    &image,
    Rect(10, 250, 0.6*width, 0.6*height), // destination rectangle
    0, 0, // upper-left corner of source rectangle
    width, // width of source rectangle
    height, // height of source rectangle
    UnitPixel);
// Shrink the image using medium-quality interpolation.
graphics.SetInterpolationMode(InterpolationModeHighQualityBilinear);
graphics.DrawImage(
    &image,
    Rect(150, 250, 0.6 * width, 0.6 * height), // destination rectangle
    0, 0, // upper-left corner of source rectangle
    width, // width of source rectangle
    height, // height of source rectangle
    UnitPixel);
// Shrink the image using high-quality interpolation.
graphics.SetInterpolationMode(InterpolationModeHighQualityBicubic);
graphics.DrawImage(
    &image,
    Rect(290, 250, 0.6 * width, 0.6 * height), // destination rectangle
    0, 0, // upper-left corner of source rectangle
```

```
width,      // width of source rectangle
height,     // height of source rectangle
UnitPixel);
```

下面的插图显示原始图像和三个较小的图像。



### 创建缩略图像

缩略图像是图像的小版本。可通过调用 `Image` 对象的 `GetThumbnailImage` 方法创建缩略图像。

下面的示例从文件 `Compass.bmp` 构造 `Image` 对象。原始图像的宽度为 640 像素, 高度为 479 像素。该代码创建宽度和高度均为 100 像素的缩略图像。

```
Image image(L"Compass.bmp");
Image* pThumbnail = image.GetThumbnailImage(100, 100, NULL, NULL);
graphics.DrawImage(pThumbnail, 10, 10,
    pThumbnail->GetWidth(), pThumbnail->GetHeight());
```

下面的插图显示此缩略图像。



### 采用高速缓存位图来提高性能

存储图像的 **Image** 和 **Bitmap** 对象采用设备无关格式。**CachedBitmap** 对象采用当前显示设备的格式存储一幅图像。处理一幅存储在 **CachedBitmap** 对象中的图像会很快，因为它不需要将图片格式转化为所需的显示设备的格式。

下面的例子将通过 **Texture.jpg** 文件创建一个 **Bitmap** 对象和一个 **CachedBitmap** 对象。**Bitmap** 和 **CachedBitmap** 均被绘制 30,000 次。如果您运行这段代码，您会发现 **CachedBitmap** 图像的绘制比 **Bitmap** 图像要快得多。

```
Bitmap      bitmap(L"Texture.jpg");
UINT        width  = bitmap.GetWidth();
UINT        height = bitmap.GetHeight();
CachedBitmap cBitmap(&bitmap, &graphics);
int         j, k;
for(j = 0; j < 300; j += 10)
    for(k = 0; k < 1000; ++k)
        graphics.DrawImage(&bitmap, j, j / 2, width, height);
for(j = 0; j < 300; j += 10)
    for(k = 0; k < 1000; ++k)
        graphics.DrawCachedBitmap(&cBitmap, j, 150 + j / 2 );
```

#### 注意

一个 **CachedBitmap** 对象与它被创建时的显示设备的格式是相匹配的。如果您的程序用户改变了显示设置，您的代码就需要构建一个新的 **CachedBitmap** 对象。如果将改变显示格式前的 **CachedBitmap** 对象传递给 **DrawImage** 方法将会导致错误。

### 通过避免自动缩放改善性能

如果您只传递图像的左上角给 **DrawImage** 方法，GDI+ 会自动缩放图像，这将会导致性能降低。

例如，以下对 **DrawImage** 方法的调用指定左上角的位置为 (50, 30)，但是未指定目标矩形。

```
graphics.DrawImage(&image, 50, 30); // upper-left corner at (50, 30)
```

尽管从所需参数的数量上来说，这是 **DrawImage** 方法最方便的版本，但它不一定是最有效的。如果 GDI+ 使用的分辨率(通常是 96 点/英寸)与 **Image** 对象中存储的分辨率不同，则 **DrawImage** 方法将缩放图像。例如，假定一个 **Image** 对象的宽度为 216 像素而存储的水平分辨率值为 72 点/英寸。因为 216 除以 72 等于 3，所以 **DrawImage** 将缩放该图像，使其在 96 点/英寸的分辨率下的宽度

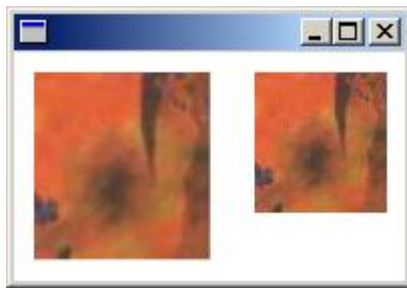
为 3 英寸。也就是说，`DrawImage` 将显示一个宽度为  $96 \times 3 = 288$  像素的图像。即使您的屏幕分辨率不是 96 点/英寸，GDI+ 也可能会像屏幕分辨率是 96 点/英寸那样缩放图像。这是因为 GDI+Graphics 对象是与设备上下文关联的，当 GDI+ 查询设备上下文以获取屏幕分辨率时，不管实际屏幕分辨率是多少，结果通常都是 96。

通过在 `DrawImage` 方法中指定目标矩形，可以避免自动缩放图像。

下面的示例将相同的图像绘制两次。在第一个例子中，未指定目标矩形的宽度和高度，图像被自动缩放。在第二个例子中，目标矩形的宽度和高度（单位是像素）被指定为与原始图像的宽度和高度相同。

```
Image image(L"Texture.jpg");
graphics.DrawImage(&image, 10, 10);
graphics.DrawImage(&image, 120, 10, image.GetWidth(), image.GetHeight());
```

下面的插图显示两次呈现的图像。



## 读取图像元数据

一些图像文件中包含可供您读取以确定图像特征的元数据。例如，数字照片中可能包含可供您读取以确定用于捕获该图像的照相机的品牌和型号的元数据。利用 GDI+，可以读取现有的元数据，也可以将新的元数据写入图像文件中。

GDI+ 提供统一的方法来向不同格式的图像中存储和获取图元数据。在 GDI+ 中，一些图元数据叫做属性项目。您可以通过调用 `Image` 类的 `SetPropertyItem` 和 `GetPropertyItem` 方法来存储和获取这些图元数据，而不必关心特定的文件格式中具体是如何存储这些图元数据的。

目前 GDI+ 支持 TIFF、JPEG、EXIF 和 PNG 文件格式中的图元数据。EXIF 格式规定了数码相机如何存储捕捉的图像，它是构建于 TIFF 和 JPEG 格式之上的。EXIF 采用 TIFF 来存储未压缩的像素数据，而采用 JPEG 来存储压缩的像素数据。

GDI+ 定义了一系列的属性标记来标识属性项目。某些标记具有通用意义；也就是说，前面提及的所有文件格式都支持。其余标记只被特定文件格式所支持。如果试图存储一个属性项目到一个文件中而该文件不支持这个属性项目的话，GDI+ 将忽略该请求。特别地，`Image::SetPropertyItem` 方法将返回 `PropertyNotSupported` 值。

你可以通过调用 `Image::GetPropertyIdList` 方法来了解存储在一幅图像中的属性项目。如果试图检索一个在文件中并不存在的属性项目，GDI+将忽略该请求。特别地，`Image::GetPropertyItem` 方法将返回 `PropertyNotFound` 值。

- 从文件中读取图元数据

下面的控制台应用程序调用了 **Image** 对象的 **GetPropertySize** 方法来判断有多少图元数据存储存储在图片 `FakePhoto.jpg` 中。

```
#include <windows.h>
#include <gdiplus.h>
#include <stdio.h>
using namespace Gdiplus;
INT main()
{
    // Initialize <tla rid="tla_gdiplus"/>.
    GdiplusStartupInput gdiplusStartupInput;
    ULONG_PTR gdiplusToken;
    GdiplusStartup(&gdiplusToken, &gdiplusStartupInput, NULL);
    UINT size = 0;
    UINT count = 0;
    Bitmap* bitmap = new Bitmap(L"FakePhoto.jpg");
    bitmap->GetPropertySize(&size, &count);
    printf("There are %d pieces of metadata in the file.\n", count);
    printf("The total size of the metadata is %d bytes.\n", size);

    delete bitmap;
    GdiplusShutdown(gdiplusToken);
    return 0;
}
```

上面的代码产生如下结果：

```
There are 7 pieces of metadata in the file.
The total size of the metadata is 436 bytes.
```

GDI+ 在一个 `PropertyItem` 对象中存储单个图元数据。你可以调用 **Image** 类的 **GetAllPropertyItems** 方法从一个文件中获取所有的图元数据。**GetAllPropertyItems** 方法将返回一个 **PropertyItem** 对象数组。在你调用该方法之前，你必须预先分配足够的缓冲区以容纳这个数组。你可以调用 **Image** 类的 **GetPropertySize** 方法来获取所需缓冲区的大小（单位：字节）

一个 **PropertyItem** 对象有以下 4 个全局成员变量：

<b>id</b>	用于标识一个图元数据项目的标记。可以赋值给 ID 的值（如 <code>PropertyTagImageTitle</code> 、 <code>PropertyTagEquipMake</code> 、 <code>PropertyTagExifExposureTime</code> 等等）定义在头文件 <code>Gdiplusimaging.h</code> 中。
-----------	--

<b>length</b>	指向 <b>Value</b> 数据成员的值的数组的长度（单位：字节）。请注意，如果 <b>Type</b> 成员设置为 PropertyTagTypeASCII，那么 <b>length</b> 数据成员指的是以 Null 结尾的字符串的长度，包括 NULL 中止符。
<b>type</b>	指向 <b>Value</b> 数据成员的值的数组的数据类型。表示不同数据类型的常数（PropertyTagTypeByte、PropertyTagTypeASCII 等等）在图像属性标记类型常数中定义。
<b>value</b>	指向值数组的一个指针。

下面的控制台应用程序读取和显示文件 **FakePhoto.jpg** 中的七个图元数据。**main** 函数基于辅助函数 **PropertyTypeFromWORD**，该函数位于 **main** 函数后面。

```
#include <windows.h>
#include <gdiplus.h>
#include <strsafe.h>
using namespace Gdiplus;

INT main()
{
    // Initialize GDI+
    GdiplusStartupInput gdiplusStartupInput;
    ULONG_PTR gdiplusToken;
    GdiplusStartup(&gdiplusToken, &gdiplusStartupInput, NULL);

    UINT size = 0;
    UINT count = 0;

    #define MAX_PROPTYPE_SIZE 30
    WCHAR strPropertyType[MAX_PROPTYPE_SIZE] = L"";

    Bitmap* bitmap = new Bitmap(L"FakePhoto.jpg");

    bitmap->GetPropertySize(&size, &count);
    printf("There are %d pieces of metadata in the file.\n\n", count);

    // GetAllPropertyItems returns an array of PropertyItem objects.
    // Allocate a buffer large enough to receive that array.
    PropertyItem* pPropBuffer =(PropertyItem*)malloc(size);

    // Get the array of PropertyItem objects.
    bitmap->GetAllPropertyItems(size, count, pPropBuffer);

    // For each PropertyItem in the array, display the id, type, and length.
    for(UINT j = 0; j < count; ++j)
    {
        // Convert the property type from a WORD to a string.
        PropertyTypeFromWORD(
```

```

        pPropBuffer[j].type, strPropertyType, MAX_PROPTYPE_SIZE);

    printf("Property Item %d\n", j);
    printf("  id: 0x%x\n", pPropBuffer[j].id);
    wprintf(L"  type: %s\n", strPropertyType);
    printf("  length: %d bytes\n\n", pPropBuffer[j].length);
}

free(pPropBuffer);
delete bitmap;
GdiplusShutdown(gdiplusToken);
return 0;
} // main

// Helper function
HRESULT PropertyTypeFromWORD(WORD index, WCHAR* string, UINT maxChars)
{
    HRESULT hr = E_FAIL;

    WCHAR* propertyTypes[] = {
        L"Nothing",           // 0
        L"PropertyTagTypeByte", // 1
        L"PropertyTagTypeASCII", // 2
        L"PropertyTagTypeShort", // 3
        L"PropertyTagTypeLong", // 4
        L"PropertyTagTypeRational", // 5
        L"Nothing",           // 6
        L"PropertyTagTypeUndefined", // 7
        L"Nothing",           // 8
        L"PropertyTagTypeSLONG", // 9
        L"PropertyTagTypeSRational"}; // 10

    hr = StringCchCopyW(string, maxChars, propertyTypes[index]);
    return hr;
}

```

上面的控制台应用程序产生如下输出：

```

Property Item 0
  id: 0x320
  type: PropertyTagTypeASCII
  length: 16 bytes
Property Item 1
  id: 0x10f
  type: PropertyTagTypeASCII

```



```

    length: 17 bytes
Property Item 2
    id: 0x110
    type: PropertyTagTypeASCII
    length: 7 bytes
Property Item 3
    id: 0x9003
    type: PropertyTagTypeASCII
    length: 20 bytes
Property Item 4
    id: 0x829a
    type: PropertyTagTypeRational
    length: 8 bytes
Property Item 5
    id: 0x5090
    type: PropertyTagTypeShort
    length: 128 bytes
Property Item 6
    id: 0x5091
    type: PropertyTagTypeShort
    length: 128 bytes

```

上面的输出显示了每个属性项目的 16 进制的 ID 值。您可以通过查阅图像属性标识常数然后得出它们代表的标记如下表所示：

16 进制值	属性标记
0x0320	PropertyTagImageTitle
0x010f	PropertyTagEquipMake
0x0110	PropertyTagEquipModel
0x9003	PropertyTagExifDTOriginal
0x829a	PropertyTagExifExposureTime
0x5090	PropertyTagLuminanceTable
0x5091	PropertyTagChrominanceTable

列表中的第二个属性项目（索引为 1）ID 为 `PropertyTagEquipMake`，类型为 `PropertyTagTypeASCII`。下面的例子，扩展了上面的例子，将显示出这个属性项目的值：

```
printf("The equipment make is %s.\n", pPropBuffer[1].value);
```

上面的代码输出结果如下：

```
The equipment make is Northwind Traders.
```

列表中的第五个属性项目（索引为 4）ID 为 PropertyTagExifExposureTime，类型为 PropertyTagTypeRational。这个数据类型(PropertyTagTypeRational)为一对长整型值。下面的例子，扩展了上面的控制台程序，将这两个长整型值显示为一个分数。这个分数，表示以秒为单位的曝光时间值。

```
long* ptrLong = (long*)(pPropBuffer[4].value);
printf("The exposure time is %d/%d.\n", ptrLong[0], ptrLong[1]);
```

上面的代码输出结果如下：

```
The exposure time is 1/125.
```

- 向文件中写入图元数据

若要想一个 **Image** 对象中写入一个项目的元数据，需要先初始化一个 **PropertyItem** 对象，然后将 **PropertyItem** 对象地址传递给 **Image** 对象的方法。

下面的控制台应用程序将向一个 **Image** 对象中写入一个项目（图像标题）的元数据，然后将这个图像存储为磁盘文件 FakePhoto2.jpg。main 函数基于辅助函数 GetEncoderClsid，请参看[为编码器获取类标识符](#)。

```
#include <windows.h>
#include <gdiplus.h>
#include <stdio.h>
using namespace Gdiplus;
INT main()
{
    // Initialize <tla rid="tla_gdiplus"/>.
    GdiplusStartupInput gdiplusStartupInput;
    ULONG_PTR gdiplusToken;
    GdiplusStartup(&gdiplusToken, &gdiplusStartupInput, NULL);
    Status stat;
    CLSID clsid;
    char propertyValue[] = "Fake Photograph";
    Bitmap* bitmap = new Bitmap(L"FakePhoto.jpg");
    PropertyItem* propertyItem = new PropertyItem;
    // Get the CLSID of the JPEG encoder.
    GetEncoderClsid(L"image/jpeg", &clsid);
    propertyItem->id = PropertyTagImageTitle;
    propertyItem->length = 16; // string length including NULL terminator
    propertyItem->type = PropertyTagTypeASCII;
    propertyItem->value = propertyValue;
    bitmap->SetPropertyItem(propertyItem);
    stat = bitmap->Save(L"FakePhoto2.jpg", &clsid, NULL);
    if(stat == Ok)
        printf("FakePhoto2.jpg saved successfully.\n");
}
```

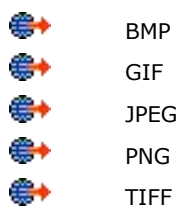
```

delete propertyItem;
delete bitmap;
GdiplusShutdown(gdiplusToken);
return 0;
}

```

## 使用图像编码器和解码器

GDI+提供了 **Image** 类和 **Bitmap** 类用于存储和处理内存图像。GDI+借助于图像编码器将图像写入磁盘文件，借助图像解码器从磁盘文件中载入图像。编码器将 **Image** 或者 **Bitmap** 对象的数据转换为指定的磁盘文件格式。解码器则将磁盘文件数据转换为 **Image** 和 **Bitmap** 对象所需的格式。GDI+内建编码器和解码器支持如下文件类型：



GDI+同时具有支持如下文件类型的内建解码器：



## 列出已安装的编码器

GDI+提供 **GetImageEncoders** 函数用于判断您的计算机支持哪些图像编码器。

**GetImageEncoders** 返回一个 **ImageCodecInfo** 对象数组。在你调用该函数之前，你必须分配足够的缓冲区来容纳这个数组。你可以调用 **GetImageEncodersSize** 方法来获取你所需要的缓冲区的大小。

下面的控制台程序将列出目前可用的图像编码器。

```

#include <windows.h>
#include <gdiplus.h>
#include <stdio.h>
using namespace Gdiplus;

INT main()
{
    // Initialize GDI+.
    GdiplusStartupInput gdiplusStartupInput;
    ULONG_PTR gdiplusToken;
    GdiplusStartup(&gdiplusToken, &gdiplusStartupInput, NULL);

    UINT num;          // number of image encoders

```

```

UINT size;          // size, in bytes, of the image encoder array

ImageCodecInfo* pImageCodecInfo;

// How many encoders are there?
// How big (in bytes) is the array of all ImageCodecInfo objects?
GetImageEncodersSize(&num, &size);

// Create a buffer large enough to hold the array of ImageCodecInfo
// objects that will be returned by GetImageEncoders.
pImageCodecInfo = (ImageCodecInfo*)(malloc(size));

// GetImageEncoders creates an array of ImageCodecInfo objects
// and copies that array into a previously allocated buffer.
// The third argument, imageCodecInfo, is a pointer to that buffer.
GetImageEncoders(num, size, pImageCodecInfo);

// Display the graphics file format (MimeType)
// for each ImageCodecInfo object.
for(UINT j = 0; j < num; ++j)
{
    wprintf(L"%s\n", pImageCodecInfo[j].MimeType);
}

free(pImageCodecInfo);
GdiplusShutdown(gdiplusToken);
return 0;
}

```

运行前面的控制台程序，结果类似于下面：

```

image/bmp
image/jpeg
image/gif
image/tiff
image/png

```

### 列出已安装的解码器

GDI+ 提供 `GetImageDecoders` 函数用于判断您的计算机支持哪些图像解码器。

**GetImageDecoders** 返回一个 `ImageCodecInfo` 对象数组。在你调用该函数之前，你必须分配足够的缓冲区来容纳这个数组。你可以调用 `GetImageDecodersSize` 方法来获取你需要的缓冲区的大小。

下面的控制台程序将列出目前可用的图像解码器。

```
#include <windows.h>
```

```

#include <gdiplus.h>
#include <stdio.h>
using namespace Gdiplus;

INT main()
{
    // Initialize GDI+.
    GdiplusStartupInput gdiplusStartupInput;
    ULONG_PTR gdiplusToken;
    GdiplusStartup(&gdiplusToken, &gdiplusStartupInput, NULL);

    UINT num;        // number of image decoders
    UINT size;        // size, in bytes, of the image decoder array

    ImageCodecInfo* pImageCodecInfo;

    // How many decoders are there?
    // How big (in bytes) is the array of all ImageCodecInfo objects?
    GetImageDecodersSize(&num, &size);

    // Create a buffer large enough to hold the array of ImageCodecInfo
    // objects that will be returned by GetImageDecoders.
    pImageCodecInfo = (ImageCodecInfo*)(malloc(size));

    // GetImageDecoders creates an array of ImageCodecInfo objects
    // and copies that array into a previously allocated buffer.
    // The third argument, imageCodecInfo, is a pointer to that buffer.
    GetImageDecoders(num, size, pImageCodecInfo);

    // Display the graphics file format (MimeType)
    // for each ImageCodecInfo object.
    for(UINT j = 0; j < num; ++j)
    {
        wprintf(L"%s\n", pImageCodecInfo[j].MimeType);
    }

    free(pImageCodecInfo);
    GdiplusShutdown(gdiplusToken);
    return 0;
}

```

运行前面的控制台程序，结果类似于下面：

```

image/bmp
image/jpeg

```

```
image/gif
image/x-emf
image/x-wmf
image/tiff
image/png
image/x-icon
```

### 获取解码器的类标识符

下面的例子将采用 **GetEncoderClsid** 函数来获取一个编码器的 **MIME** (多用途网际邮件扩充协议)类型, 然后返回该编码器的类标识符 (CLSID)。GDI+中编码器的 MIME 类型如下:

	image/bmp
	image/jpeg
	image/gif
	image/tiff
	image/png

**GetImageEncoders** 函数获取一个 **ImageCodecInfo** 对象数组。如果数组中的某个 **ImageCodecInfo** 对象是你所需要的编码器, 函数将返回该 **ImageCodecInfo** 对象的索引值, 然后将它的 CLSID 复制到 *pClsid* 指向的变量中。如果函数调用失败, 返回-1。

```
int GetEncoderClsid(const WCHAR* format, CLSID* pClsid)
{
    UINT num = 0;          // number of image encoders
    UINT size = 0;         // size of the image encoder array in bytes

    ImageCodecInfo* pImageCodecInfo = NULL;

    GetImageEncodersSize(&num, &size);
    if(size == 0)
        return -1; // Failure

    pImageCodecInfo = (ImageCodecInfo*)(malloc(size));
    if(pImageCodecInfo == NULL)
        return -1; // Failure

    GetImageEncoders(num, size, pImageCodecInfo);

    for(UINT j = 0; j < num; ++j)
    {
        if( wcsncmp(pImageCodecInfo[j].MimeType, format) == 0 )
        {
            *pClsid = pImageCodecInfo[j].Clsid;
            free(pImageCodecInfo);
            return j; // Success
        }
    }
}
```

```

    }
}

free(pImageCodecInfo);
return -1; // Failure
}

```

下面的控制台程序调用 **GetEncoderClsid** 函数获取 PNG 编码器的 CLSID:

```

#include <windows.h>
#include <gdiplus.h>
#include <stdio.h>
using namespace Gdiplus;

#include "GdiplusHelperFunctions.h"

INT main()
{
    // Initialize GDI+.
    GdiplusStartupInput gdiplusStartupInput;
    ULONG_PTR gdiplusToken;
    GdiplusStartup(&gdiplusToken, &gdiplusStartupInput, NULL);

    CLSID encoderClsid;
    INT result;
    WCHAR strGuid[39];

    result = GetEncoderClsid(L"image/png", &encoderClsid);

    if(result < 0)
    {
        printf("The PNG encoder is not installed.\n");
    }
    else
    {
        StringFromGUID2(encoderClsid, strGuid, 39);
        printf("An ImageCodecInfo object representing the PNG encoder\n");
        printf("was found at position %d in the array.\n", result);
        wprintf(L"The CLSID of the PNG encoder is %s.\n", strGuid);
    }

    GdiplusShutdown(gdiplusToken);
    return 0;
}

```

运行前面的控制台程序，结果类似于下面：

```
An ImageCodecInfo object representing the PNG encoder
was found at position 4 in the array.
The CLSID of the PNG encoder is {557CF406-1A04-11D3-9A73-0000F81EF32E}.
```

### 获取编码器的参数列表

Image 类提供 **GetEncoderParameterList** 方法用于判断指定的图像编码器的参数列表。

**GetEncoderParameterList** 方法返回一个 **EncoderParameter** 对象数组。你必须在调用该函数之前为这个数组分配缓冲区。您可以通过调用 **Image::GetEncoderParameterListSize** 来获取你所需要的缓冲区大小。

下面的控制台程序获取 JPEG 编码器的参数列表。Main 函数依赖于辅助函数 **GetEncoderClsid**，后者用于检索一个编码器的类标识符。

```
#include <windows.h>
#include <gdiplus.h>
#include <stdio.h>
using namespace Gdiplus;

INT GetEncoderClsid(const WCHAR* format, CLSID* pClsid); // helper function

INT main()
{
    // Initialize GDI+.
    GdiplusStartupInput gdiplusStartupInput;
    ULONG_PTR gdiplusToken;
    GdiplusStartup(&gdiplusToken, &gdiplusStartupInput, NULL);

    // Create Bitmap (inherited from Image) object so that we can call
    // GetParameterListSize and GetParameterList.
    Bitmap* bitmap = new Bitmap(1, 1);

    // Get the JPEG encoder CLSID.
    CLSID encoderClsid;
    GetEncoderClsid(L"image/jpeg", &encoderClsid);

    // How big (in bytes) is the JPEG encoder's parameter list?
    UINT listSize = 0;
    listSize = bitmap->GetEncoderParameterListSize(&encoderClsid);
    printf("The parameter list requires %d bytes.\n", listSize);

    // Allocate a buffer large enough to hold the parameter list.
    EncoderParameters* pEncoderParameters = NULL;
    pEncoderParameters = (EncoderParameters*)malloc(listSize);
```



```

// Get the parameter list for the JPEG encoder.
bitmap->GetEncoderParameterList(
    &encoderClsid, listSize, pEncoderParameters);

// pEncoderParameters points to an EncoderParameters object, which
// has a Count member and an array of EncoderParameter objects.
// How many EncoderParameter objects are in the array?
printf("There are %d EncoderParameter objects in the array.\n",
    pEncoderParameters->Count);

free(pEncoderParameters);
delete(bitmap);
GdiplusShutdown(gdiplusToken);
return 0;
}

```

运行前面的控制台程序，结果类似于下面：

```

The parameter list requires 172 bytes.
There are 4 EncoderParameter objects in the array.

```

数组中的每个 **EncoderParameter** 对象都具有如下的公共数据成员：

下面的代码是前面例子中控制台程序的扩展。代码查询 **GetEncoderParameterList** 函数返回的数组中第二个 **EncoderParameter** 对象。调用 **StringFromGUID2**（这个系统函数定义在头文件 **Objbase.h** 中）将 **EncoderParameter** 对象的 **Guid** 成员转换为字符串。

```

// Look at the second (index 1) EncoderParameter object in the array.
printf("Parameter[1]\n");

WCHAR strGuid[39];
StringFromGUID2(pEncoderParameters->Parameter[1].Guid, strGuid, 39);
wprintf(L"    The GUID is %s.\n", strGuid);

printf("    The value type is %d.\n",
    pEncoderParameters->Parameter[1].Type);

printf("    The number of values is %d.\n",
    pEncoderParameters->Parameter[1].NumberOfValues);

```

前面的代码参数如下输出：

```

Parameter[1]
    The GUID is {1D5BE4B5-FA4A-452D-9CDD-5DB35105E7EB}.

```

```
The value type is 6.
The number of values is 1.
```

你可以在 `Gdiplusimaging.h` 中查阅这个 GUID, 得知这个 **EncoderParameter** 对象的类别 6 表示编码质量。你可以使用这个类别（编码质量）参数来设置 JPEG 图像的压缩级别。

在 `Gdiplusenums.h` 中, `EncoderParameterValueType` 枚举显示数据类型 6 是一个 **ValueLongRange**。一个长整型范围是一对 **ULONG** 值。

数值等于 1, 表示对象的 **Value** 成员是一个只有一个元素的数组的指针。这个元素是由一对 **ULONG** 值组成的。

下面的代码是上面例子中代码的扩展。代码定义了一个称为 **PLONGRANGE**（长整型范围的指针）的数据类型。一个 **PLONGRANGE** 类型的变量用于可以传给 JPEG 编码器用于质量设置的最小值和最大值。

```
typedef struct
{
    long min;
    long max;
}* PLONGRANGE;

PLONGRANGE pLongRange =
    (PLONGRANGE) (pEncoderParameters->Parameter[1].Value);

printf("  The minimum possible quality value is %d.\n",
    pLongRange->min);

printf("  The maximum possible quality value is %d.\n",
    pLongRange->max);
```

前面的代码产生如下输出:

```
The minimum possible quality value is 0.
The maximum possible quality value is 100.
```

上例中, **EncoderParameter** 对象返回的值是一对 **ULONG** 值, 表示质量参数可接受的最小值与最大值。有些情况下, 对象返回的值是一个 **EncoderValue** 枚举。下面的主题将更加详细地讨论 **EncoderValue** 枚举以及列出可能的参数值的方法。

- 使用 **EncoderValue** 枚举

给定的编码器支持特定的参数类别, 对这些参数类别而言, 编码器接受特定的值。例如, JPEG 编码器支持 **EncoderValueQuality** 参数类型, 允许的参数值从 0 到 100 的整数。好多编码器都具有同样的交叉的可接受的参数值。这些标准值定义在头文件 `Gdiplusenums.h` 的 **EncoderValue** 枚举中。

```
enum EncoderValue
{
    EncoderValueColorTypeCMYK,          // 0
    EncoderValueColorTypeYCK,           // 1
    EncoderValueCompressionLZW,         // 2
    EncoderValueCompressionCCITT3,      // 3
    EncoderValueCompressionCCITT4,      // 4
    EncoderValueCompressionRle,         // 5
    EncoderValueCompressionNone,        // 6
    EncoderValueScanMethodInterlaced,   // 7
    EncoderValueScanMethodNonInterlaced, // 8
    EncoderValueVersionGif87,           // 9
    EncoderValueVersionGif89,           // 10
    EncoderValueRenderProgressive,       // 11
    EncoderValueRenderNonProgressive,    // 12
    EncoderValueTransformRotate90,       // 13
    EncoderValueTransformRotate180,      // 14
    EncoderValueTransformRotate270,     // 15
    EncoderValueTransformFlipHorizontal, // 16
    EncoderValueTransformFlipVertical,   // 17
    EncoderValueMultiFrame,              // 18
    EncoderValueLastFrame,               // 19
    EncoderValueFlush,                   // 20
    EncoderValueFrameDimensionTime,      // 21
    EncoderValueFrameDimensionResolution, // 22
    EncoderValueFrameDimensionPage       // 23
};
```

JPEG 编码器支持的参数类型之一是 **EncoderTransformation** 类型。通过检查 **EncoderValue** 枚举，你可以推测（你将是正确的）**EncoderTransformation** 类型可以接受如下 5 个值：

```
EncoderValueTransformRotate90,        // 13
EncoderValueTransformRotate180,       // 14
EncoderValueTransformRotate270,       // 15
EncoderValueTransformFlipHorizontal,   // 16
EncoderValueTransformFlipVertical,     // 17
```

下面的控制台程序将验证 JPEG 编码器支持的 **EncoderTransformation** 参数类型就是上面列出的 5 个可接受值。**Main** 函数依赖于辅助函数 **GetEncoderClsid**，后者用于获取一个编码器的类标识符。

```
#include <windows.h>
#include <gdiplus.h>
#include <stdio.h>
using namespace Gdiplus;
INT GetEncoderClsid(const WCHAR* format, CLSID* pClsid);
```

```

INT main()
{
    // Initialize GDI+.
    GdiplusStartupInput gdiplusStartupInput;
    ULONG_PTR gdiplusToken;
    GdiplusStartup(&gdiplusToken, &gdiplusStartupInput, NULL);
    // Create a Bitmap (inherited from Image) object so that we can call
    // GetParameterListSize and GetParameterList.
    Bitmap* bitmap = new Bitmap(1, 1);
    // Get the JPEG encoder CLSID.
    CLSID encoderClsid;
    GetEncoderClsid(L"image/jpeg", &encoderClsid);
    // How big (in bytes) is the JPEG encoder's parameter list?
    UINT listSize = 0;
    listSize = bitmap->GetEncoderParameterListSize(&encoderClsid);
    printf("The parameter list requires %d bytes.\n", listSize);
    // Allocate a buffer large enough to hold the parameter list.
    EncoderParameters* pEncoderParameters = NULL;
    pEncoderParameters = (EncoderParameters*)malloc(listSize);
    // Get the parameter list for the JPEG encoder.
    bitmap->GetEncoderParameterList(
        &encoderClsid, listSize, pEncoderParameters);
    // pEncoderParameters points to an EncoderParameters object, which
    // has a Count member and an array of EncoderParameter objects.
    // How many EncoderParameter objects are in the array?
    printf("There are %d EncoderParameter objects in the array.\n",
        pEncoderParameters->Count);
    // Look at the first (index 0) EncoderParameter object in the array.
    printf("Parameter[0]\n");
    WCHAR strGuid[39];
    StringFromGUID2(pEncoderParameters->Parameter[0].Guid, strGuid, 39);
    wprintf(L"    The guid is %s.\n", strGuid);
    printf("    The data type is %d.\n",
        pEncoderParameters->Parameter[0].Type);
    printf("    The number of values is %d.\n",
        pEncoderParameters->Parameter[0].NumberOfValues);
    free(pEncoderParameters);
    delete bitmap;
    GdiplusShutdown(gdiplusToken);
    return 0;
}

```

运行上面的控制台程序，你将得到类似于下面的输出结果：

```
The parameter list requires 172 bytes.
```

```
There are 4 EncoderParameter objects in the array.
```

```
Parameter[0]
```

```
    The GUID is {8D0EB2D1-A58E-4EA8-AA14-108074B7B6F9}.
```

```
    The value type is 4.
```

```
    The number of values is 5.
```

通过查阅 `Gdiplusing.h` 头文件中的 GUID 可以得知这个 `EncoderParameter` 对象的类别为 `EncoderTransformation`。在 `Gdiplusing.h` 头文件中, `EncoderParameterValueType` 枚举显示数据类型 4 是一个 `ValueLong` (32 位无符号整型)。`Value` 的值等于 5, 因此我们可以知道 **EncoderParameter** 对象的 `Value` 成员是一个指向由 5 个 `ULONG` 值组成的数组的指针。

下面的代码是上面例子的扩展。代码将列出 `EncoderTransformation` 参数的可接受值。

```
ULONG* pUlong = (ULONG*) (pEncoderParameters->Parameter[0].Value);
ULONG numVals = pEncoderParameters->Parameter[0].NumberOfValues;
printf("%s", "    The allowable values are");
for(ULONG j = 0; j < numVals; ++j)
{
    printf("    %d", pUlong[j]);
}
```

输出结果如下:

```
The allowable values are 13 14 15 16 17
```

可接受值 (13, 14, 15, 16, and 17) 表示下面的 `EncoderValue` 枚举常量:

```
EncoderValueTransformRotate90,        // 13
EncoderValueTransformRotate180,       // 14
EncoderValueTransformRotate270,       // 15
EncoderValueTransformFlipHorizontal,  // 16
EncoderValueTransformFlipVertical,    // 17
```

- 列出所有编码器的参数和取值

下面的控制台程序列出计算机上安装的不同编码器所支持的所有参数。`Main` 函数调用 `GetImageEncoders` 来判断哪些编码器可用。对每个可用的编码器, `main` 函数调用辅助函数 `ShowAllEncoderParameters`。

`ShowAllEncoderParameters` 函数调用 `Image::GetEncoderParameterList` 方法来判断指定的编码器支持哪些参数。对每个支持的参数, 函数列出其类别、数据类型和取值。`ShowAllEncoderParameters` 函数又依赖于 2 个辅助函数: `EncoderParameterCategoryFromGUID` 和 `ValueTypeFromULONG`。

```
#include <windows.h>
#include <gdiplus.h>
```

```

#include <strsafe.h>
using namespace Gdiplus;

// Helper functions
void ShowAllEncoderParameters(ImageCodecInfo*);
HRESULT EncoderParameterCategoryFromGUID(GUID guid, WCHAR* category, UINT
maxChars);
HRESULT ValueTypeFromULONG(ULONG index, WCHAR* strValueType, UINT maxChars);

INT main()
{
    // Initialize GDI+
    GdiplusStartupInput gdiplusStartupInput;
    ULONG_PTR gdiplusToken;
    GdiplusStartup(&gdiplusToken, &gdiplusStartupInput, NULL);

    UINT num;        // Number of image encoders
    UINT size;        // Size of the image encoder array in bytes

    ImageCodecInfo* pImageCodecInfo;

    // How many encoders are there?
    // How big (in bytes) is the array of all ImageCodecInfo objects?
    GetImageEncodersSize(&num, &size);

    // Create a buffer large enough to hold the array of ImageCodecInfo
    // objects that will be returned by GetImageEncoders.
    pImageCodecInfo = (ImageCodecInfo*)(malloc(size));

    // GetImageEncoders creates an array of ImageCodecInfo objects
    // and copies that array into a previously allocated buffer.
    // The third argument, imageCodecInfos, is a pointer to that buffer.
    GetImageEncoders(num, size, pImageCodecInfo);

    // For each ImageCodecInfo object in the array, show all parameters.
    for(UINT j = 0; j < num; ++j)
    {
        ShowAllEncoderParameters(&(pImageCodecInfo[j]));
    }

    GdiplusShutdown(gdiplusToken);
    return 0;
}

```

```

////////////////////////////////////
// Helper functions

VOID ShowAllEncoderParameters(ImageCodecInfo* pImageCodecInfo)
{
    CONST MAX_CATEGORY_LENGTH = 50;
    CONST MAX_VALUE_TYPE_LENGTH = 50;
    WCHAR strParameterCategory[MAX_CATEGORY_LENGTH] = L"";
    WCHAR strValueType[MAX_VALUE_TYPE_LENGTH] = L"";

    wprintf(L"\n\n%s\n", pImageCodecInfo->MimeType);

    // Create a Bitmap (inherited from Image) object so that we can call
    // GetParameterListSize and GetParameterList.
    Bitmap bitmap(1, 1);

    // How big (in bytes) is the encoder's parameter list?
    UINT listSize = 0;
    listSize = bitmap.GetEncoderParameterListSize(&pImageCodecInfo->Clsid);
    printf(" The parameter list requires %d bytes.\n", listSize);

    if(listSize == 0)
        return;

    // Allocate a buffer large enough to hold the parameter list.
    EncoderParameters* pEncoderParameters = NULL;
    pEncoderParameters = (EncoderParameters*)malloc(listSize);

    if(pEncoderParameters == NULL)
        return;

    // Get the parameter list for the encoder.
    bitmap.GetEncoderParameterList(
        &pImageCodecInfo->Clsid, listSize, pEncoderParameters);

    // pEncoderParameters points to an EncoderParameters object, which
    // has a Count member and an array of EncoderParameter objects.
    // How many EncoderParameter objects are in the array?
    printf(" There are %d EncoderParameter objects in the array.\n",
        pEncoderParameters->Count);

    // For each EncoderParameter object in the array, list the
    // parameter category, data type, and number of values.

```

```

    for(UINT k = 0; k < pEncoderParameters->Count; ++k)
    {
        EncoderParameterCategoryFromGUID(
            pEncoderParameters->Parameter[k].Guid, strParameterCategory,
MAX_CATEGORY_LENGTH);

        ValueTypeFromULONG(
            pEncoderParameters->Parameter[k].Type, strValueType,
MAX_VALUE_TYPE_LENGTH);

        printf("    Parameter[%d]\n", k);
        wprintf(L"        The category is %s.\n", strParameterCategory);
        wprintf(L"        The data type is %s.\n", strValueType);

        printf("        The number of values is %d.\n",
            pEncoderParameters->Parameter[k].NumberOfValues);
    } // for

    free(pEncoderParameters);
} // ShowAllEncoderParameters

HRESULT EncoderParameterCategoryFromGUID(GUID guid, WCHAR* category, UINT
maxChars)
{
    HRESULT hr = E_FAIL;

    if(guid == EncoderCompression)
        hr = StringCchCopyW(category, maxChars, L"Compression");
    else if(guid == EncoderColorDepth)
        hr = StringCchCopyW(category, maxChars, L"ColorDepth");
    else if(guid == EncoderScanMethod)
        hr = StringCchCopyW(category, maxChars, L"ScanMethod");
    else if(guid == EncoderVersion)
        hr = StringCchCopyW(category, maxChars, L"Version");
    else if(guid == EncoderRenderMethod)
        hr = StringCchCopyW(category, maxChars, L"RenderMethod");
    else if(guid == EncoderQuality)
        hr = StringCchCopyW(category, maxChars, L"Quality");
    else if(guid == EncoderTransformation)
        hr = StringCchCopyW(category, maxChars, L"Transformation");
    else if(guid == EncoderLuminanceTable)
        hr = StringCchCopyW(category, maxChars, L"LuminanceTable");
    else if(guid == EncoderChrominanceTable)

```



```

        hr = StringCchCopyW(category, maxChars, L"ChrominanceTable");
    else if(guid == EncoderSaveFlag)
        hr = StringCchCopyW(category, maxChars, L"SaveFlag");
    else
        hr = StringCchCopyW(category, maxChars, L"Unknown category");

    return hr;
} // EncoderParameterCategoryFromGUID

HRESULT ValueTypeFromULONG(ULONG index, WCHAR* strValueType, UINT maxChars)
{
    HRESULT hr = E_FAIL;

    WCHAR* valueTypes[] = {
        L"Nothing",           // 0
        L"ValueTypeByte",     // 1
        L"ValueTypeASCII",    // 2
        L"ValueTypeShort",    // 3
        L"ValueTypeLong",     // 4
        L"ValueTypeRational",  // 5
        L"ValueTypeLongRange", // 6
        L"ValueTypeUndefined", // 7
        L"ValueTypeRationalRange" // 8
    };

    hr = StringCchCopyW(strValueType, maxChars, valueTypes[index]);
    return hr;
} // ValueTypeFromULONG

```

运行上面的控制台程序，将得到类似下面的输出结果：

```

image/bmp
The parameter list requires 0 bytes.

image/jpeg
The parameter list requires 172 bytes.
There are 4 EncoderParameter objects in the array.
Parameter[0]
    The category is Transformation.
    The data type is Long.
    The number of values is 5.
Parameter[1]
    The category is Quality.
    The data type is LongRange.

```

```

    The number of values is 1.
Parameter[2]
    The category is LuminanceTable.
    The data type is Short.
    The number of values is 0.
Parameter[3]
    The category is ChrominanceTable.
    The data type is Short.
    The number of values is 0.

image/gif
    The parameter list requires 0 bytes.

image/tiff
    The parameter list requires 160 bytes.
    There are 3 EncoderParameter objects in the array.
    Parameter[0]
        The category is Compression.
        The data type is Long.
        The number of values is 5.
    Parameter[1]
        The category is ColorDepth.
        The data type is Long.
        The number of values is 5.
    Parameter[2]
        The category is SaveFlag.
        The data type is Long.
        The number of values is 1.

image/png
    The parameter list requires 0 bytes.

```

通过检查前面的程序输出，你可以得出如下结论：

JPEG 编码器支持 Transformation、Quality、LuminanceTable 和 ChrominanceTable 参数类别。

TIFF 编码器支持 Compression、ColorDepth 和 SaveFlag 参数类别。

你也可以查看每个参数类别的取值。例如，你可以看到 ColorDepth 参数类别（TIFF 编码）有 5 个 **ULONG** 类型的取值。下面的代码列出这 5 个值。假定 pEncoderParameters 是一个表示 TIFF 编码器的 EncoderParameters 对象的指针。

```


ULONG* pUlong = (ULONG*)(pEncoderParameters->Parameter[1].Value);
ULONG numVals = pEncoderParameters->Parameter[1].NumberOfValues;
printf("\nThe allowable values for ColorDepth are\n");

```

```
for(ULONG k = 0; k < numVals; ++k)
{
    printf("  %u\n", pUlong[k]);
}
```

上面的代码输出结果如下：

```
The allowable values for ColorDepth are
1
4
8
24
32
```

 **注意**  
有些情况下，一个 **EncoderParameter** 对象的值是 **EncoderValue** 枚举的元素值。然而，前面列表中的值与 **EncoderValue** 枚举无关。该数值表示每个像素 1 位、每个像素 2 位、依此类推。

如果你采用上面的代码来统计其他参数类别的取值范围，你可以得到类似如下的表格：

JPEG 编码器参数	可接受值
Transformation	EncoderValueTransformRotate90
	EncoderValueTransformRotate180
	EncoderValueTransformRotate270
	EncoderValueTransformFlipHorizontal
	EncoderValueTransformFlipVertical
Quality	0 through 100
TIFF 编码器参数	可接受值
Compression	EncoderValueCompressionLZW
	EncoderValueCompressionCCITT3
	EncoderValueCompressionCCITT4
	EncoderValueCompressionRle
	EncoderValueCompressionNone
ColorDepth	1, 4, 8, 24, 32

SaveFlag

EncoderValueMultiFrame

### 注意

如果 JPEG 图像的宽度和高度是 16 的倍数，你可以应用 `EncoderTransformation` 参数类别中的任何变换（例如，90 度旋转）而不丢失信息。

### 将 BMP 图像转换为 PNG 图像

若要保存一幅图像为磁盘文件，需要调用 `Image` 类的 `Save` 方法。下面的控制台程序从一个磁盘文件载入一幅 BMP 图像，将它转换为 PNG 格式，然后将转换后的图像存为新的磁盘文件。`Main` 函数依赖于辅助函数 `GetEncoderClsid`，后者用于获取一个编码器的类标识符。

```
#include <windows.h>
#include <gdiplus.h>
#include <stdio.h>
using namespace Gdiplus;

INT GetEncoderClsid(const WCHAR* format, CLSID* pClsid); // helper function

INT main()
{
    // Initialize GDI+.
    GdiplusStartupInput gdiplusStartupInput;
    ULONG_PTR gdiplusToken;
    GdiplusStartup(&gdiplusToken, &gdiplusStartupInput, NULL);

    CLSID encoderClsid;
    Status stat;
    Image* image = new Image(L"Bird.bmp");

    // Get the CLSID of the PNG encoder.
    GetEncoderClsid(L"image/png", &encoderClsid);

    stat = image->Save(L"Bird.png", &encoderClsid, NULL);

    if(stat == Ok)
        printf("Bird.png was saved successfully\n");
    else
        printf("Failure: stat = %d\n", stat);

    delete image;
    GdiplusShutdown(gdiplusToken);
    return 0;
}
```

## 设定 JPEG 的压缩等级

若要在保存一幅 JPEG 图像时指定压缩级别，需要初始化一个 **EncoderParameters** 对象，然后将该对象的地址传给 **Image** 类的 **Save** 方法。初始化 **EncoderParameters** 对象让它包含一个含有单个 **EncoderParameter** 对象的数组。初始化这个对象让它的 **Value** 成员指向一个在 0 到 100 以内的 ULONG 值。设置对象的 **Guid** 成员为 **EncoderQuality**。

下面的控制台程序将采用 3 中不同的质量等级来存储 3 幅 JPEG 图像，质量等级为 0 表示最大压缩，100 表示最小压缩。

**Main** 函数依赖于辅助函数 **GetEncoderClsid**，后者用于获取一个指定编码器的类标识符。

```
#include <windows.h>
#include <gdiplus.h>
#include <stdio.h>
using namespace Gdiplus;

INT GetEncoderClsid(const WCHAR* format, CLSID* pClsid); // helper function

INT main()
{
    // Initialize GDI+.
    GdiplusStartupInput gdiplusStartupInput;
    ULONG_PTR gdiplusToken;
    GdiplusStartup(&gdiplusToken, &gdiplusStartupInput, NULL);

    CLSID          encoderClsid;
    EncoderParameters encoderParameters;
    ULONG          quality;
    Status          stat;

    // Get an image from the disk.
    Image* image = new Image(L"Shapes.bmp");

    // Get the CLSID of the JPEG encoder.
    GetEncoderClsid(L"image/jpeg", &encoderClsid);

    // Before we call Image::Save, we must initialize an
    // EncoderParameters object. The EncoderParameters object
    // has an array of EncoderParameter objects. In this
    // case, there is only one EncoderParameter object in the array.
    // The one EncoderParameter object has an array of values.
    // In this case, there is only one value (of type ULONG)
    // in the array. We will let this value vary from 0 to 100.
```

```

encoderParameters.Count = 1;
encoderParameters.Parameter[0].Guid = EncoderQuality;
encoderParameters.Parameter[0].Type = EncoderParameterValueTypeLong;
encoderParameters.Parameter[0].NumberOfValues = 1;

// Save the image as a JPEG with quality level 0.
quality = 0;
encoderParameters.Parameter[0].Value = &quality;
stat = image->Save(L"Shapes001.jpg", &encoderClsid, &encoderParameters);

if(stat == Ok)
    wprintf(L"%s saved successfully.\n", L"Shapes001.jpg");
else
    wprintf(L"%d Attempt to save %s failed.\n", stat, L"Shapes001.jpg");

// Save the image as a JPEG with quality level 50.
quality = 50;
encoderParameters.Parameter[0].Value = &quality;
stat = image->Save(L"Shapes050.jpg", &encoderClsid, &encoderParameters);

if(stat == Ok)
    wprintf(L"%s saved successfully.\n", L"Shapes050.jpg");
else
    wprintf(L"%d Attempt to save %s failed.\n", stat, L"Shapes050.jpg");

// Save the image as a JPEG with quality level 100.
quality = 100;
encoderParameters.Parameter[0].Value = &quality;
stat = image->Save(L"Shapes100.jpg", &encoderClsid, &encoderParameters);

if(stat == Ok)
    wprintf(L"%s saved successfully.\n", L"Shapes100.jpg");
else
    wprintf(L"%d Attempt to save %s failed.\n", stat, L"Shapes100.jpg");

delete image;
GdiplusShutdown(gdiplusToken);
return 0;
}

```



## 对 JPEG 图像进行无损变换

压缩 JPEG 图像将会使图像某些信息丢失。一旦你打开一个 JPEG 文件，改动图片然后将它存储为另一个 JPEG 文件，质量将会降低。如果您重复执行多次，您会发现图像质量降低了很多。

因为 JPEG 是 Web 上最流行的图像格式，而且人们通常更愿意编辑 JPEG 图像，因此 GDI+ 提供如下 JPEG 图像的无损变换。






-  旋转 90 度
-  旋转 180 度
-  旋转 270 度
-  水平翻转
-  垂直翻转

你可以在调用 **Image** 对象的 **Save** 方法的时候应用上面列表中的其中一种变换。在下面的情况被满足的时候，该变换将不会丢失信息：

-  采用 JPEG 文件来构造 **Image** 对象
-  图像的高度和宽度是 16 的倍数

如果图像的宽度和高度并非都是 16 的倍数，GDI+ 将在应用上面列表中的一个旋转或者翻转的时候尽最大努力保持图像质量。

若要转换一幅 JPEG 图像，需要先初始化 **EncoderParameters** 对象，然后将该对象的地址传给 **Image** 类的 **Save** 方法。初始化 **EncoderParameters** 对象让它包含一个含有单个 **EncoderParameter** 对象的数组。初始化这个对象让它的 **Value** 成员指向一个包含了下列枚举元素之一的 **ULONG** 变量。

-  **EncoderValueTransformRotate90,**
-  **EncoderValueTransformRotate180,**
-  **EncoderValueTransformRotate270,**
-  **EncoderValueTransformFlipHorizontal,**
-  **EncoderValueTransformFlipVertical**

将 **EncoderParameter** 对象的 **Guid** 成员设置为 **EncoderTransformation**。

下面的控制台程序从一个 JPEG 文件创建一个 **Image** 对象，然后将图像存储为一个新文件。在保存过程中，图像旋转了 90 度。如果图像的宽度和高度都是 16 的倍数的话，旋转和保存图像的过程将不会丢失信息。

**Main** 函数依赖于辅助函数 **GetEncoderClsid**，后者用于获取指定编码器的类标识符。

```
#include <windows.h>
#include <gdiplus.h>
#include <stdio.h>
using namespace Gdiplus;

INT GetEncoderClsid(const WCHAR* format, CLSID* pClsid); // helper function

INT main()
```

```

{
    // Initialize GDI+.
    GdiplusStartupInput gdiplusStartupInput;
    ULONG_PTR gdiplusToken;
    GdiplusStartup(&gdiplusToken, &gdiplusStartupInput, NULL);

    CLSID          encoderClsid;
    EncoderParameters encoderParameters;
    ULONG          transformation;
    UINT           width;
    UINT           height;
    Status          stat;

    // Get a JPEG image from the disk.
    Image* image = new Image(L"Shapes.jpg");

    // Determine whether the width and height of the image
    // are multiples of 16.
    width = image->GetWidth();
    height = image->GetHeight();

    printf("The width of the image is %u", width);
    if(width / 16.0 - width / 16 == 0)
        printf(", which is a multiple of 16.\n");
    else
        printf(", which is not a multiple of 16.\n");

    printf("The height of the image is %u", height);
    if(height / 16.0 - height / 16 == 0)
        printf(", which is a multiple of 16.\n");
    else
        printf(", which is not a multiple of 16.\n");

    // Get the CLSID of the JPEG encoder.
    GetEncoderClsid(L"image/jpeg", &encoderClsid);

    // Before we call Image::Save, we must initialize an
    // EncoderParameters object. The EncoderParameters object
    // has an array of EncoderParameter objects. In this
    // case, there is only one EncoderParameter object in the array.
    // The one EncoderParameter object has an array of values.
    // In this case, there is only one value (of type ULONG)
    // in the array. We will set that value to EncoderValueTransformRotate90.

```



```

encoderParameters.Count = 1;
encoderParameters.Parameter[0].Guid = EncoderTransformation;
encoderParameters.Parameter[0].Type = EncoderParameterValueLong;
encoderParameters.Parameter[0].NumberOfValues = 1;

// Rotate and save the image.
transformation = EncoderValueTransformRotate90;
encoderParameters.Parameter[0].Value = &transformation;
stat = image->Save(L"ShapesR90.jpg", &encoderClsid, &encoderParameters);

if(stat == Ok)
    wprintf(L"%s saved successfully.\n", L"ShapesR90.jpg");
else
    wprintf(L"%d Attempt to save %s failed.\n", stat, L"ShapesR90.jpg");

delete image;
GdiplusShutdown(gdiplusToken);
return 0;
}

```

## 创建和保存多帧图像

在某些文件格式中，你可以保存多幅图像（帧）到单个文件中。例如，你可以将多个页面存储到一个 TIFF 文件中。通过调用 **Image** 类的 **Save** 方法保存第一页，**SaveAdd** 方法保存后面的页。

下面的控制台程序创建一个具有 4 个页面的 TIFF 文件。组成 TIFF 文件页面的图像来自于四个磁盘文件：Shapes.bmp、Cereal.gif、ron.jpg 和 House.png。代码首先创建 4 个 **Image** 对象：*multi*、*page2*、*page3* 和 *page4*。*multi* 开始时只包含图像，但是最后它包含了所有的 4 幅图像。在单独页面被加入 *multi Image* 对象的时候，它们也同样被加入到磁盘文件 **Multiframe.tif** 中。

请注意代码调用 **Save**（而非 **SaveAdd**）来保存的第一页。传给 **Save** 方法的第一个参数是磁盘文件的名称，这个文件最后将包含多个帧。传给 **Save** 方法的第二个参数指定编码器，该编码器将用于将 *multi Image* 对象中的数据转化为磁盘文件所需格式。同一个编码器将自动用于后面多个 **Image** 对象对 **SaveAdd** 方法的调用。

传给 **Save** 方法的第三个参数是一个 **EncoderParameters** 对象的地址。**EncoderParameters** 对象有一个数组，它包含一个单独 **EncoderParameter** 的对象。**EncoderParameter** 对象的 **Guid** 成员设置为 **EncoderSaveFlag**。**EncoderParameter** 对象的 **Value** 成员指向一个 **ULONG** 类型的值 **EncoderValueMultiFrame**。

代码调用 **SaveAdd** 方法加入 *multi Image* 对象的第二、第三和第四个页面。**SaveAdd** 方法的第一个参数是一个 **Image** 对象的地址。在这个 **Image** 对象中的图像将被加入 *multi Image* 对象中，同时也被加入到 **Multiframe.tif** 磁盘文件中。**SaveAdd** 方法的第二个参数是与 **Save** 方法采用的同一个 **EncoderParameters** 对象的地址。不同之处在于，现在的 **Value** 成员指向的 **ULONG** 类型值等于 **EncoderValueFrameDimensionPage**。

Main 函数依赖于辅助函数 **GetEncoderClsid**，后者用于获取指定编码器的类标识符。

```
#include <windows.h>
#include <gdiplus.h>
#include <stdio.h>
using namespace Gdiplus;

INT GetEncoderClsid(const WCHAR* format, CLSID* pClsid); // helper function

INT main()
{
    // Initialize GDI+.
    GdiplusStartupInput gdiplusStartupInput;
    ULONG_PTR gdiplusToken;
    GdiplusStartup(&gdiplusToken, &gdiplusStartupInput, NULL);

    EncoderParameters encoderParameters;
    ULONG           parameterValue;
    Status          stat;

    // An EncoderParameters object has an array of
    // EncoderParameter objects. In this case, there is only
    // one EncoderParameter object in the array.
    encoderParameters.Count = 1;

    // Initialize the one EncoderParameter object.
    encoderParameters.Parameter[0].Guid = EncoderSaveFlag;
    encoderParameters.Parameter[0].Type = EncoderParameterValueTypeLong;
    encoderParameters.Parameter[0].NumberOfValues = 1;
    encoderParameters.Parameter[0].Value = &parameterValue;

    // Get the CLSID of the TIFF encoder.
    CLSID encoderClsid;
    GetEncoderClsid(L"image/tiff", &encoderClsid);

    // Create four image objects.
    Image* multi = new Image(L"Shapes.bmp");
    Image* page2 = new Image(L"Cereal.gif");
    Image* page3 = new Image(L"Iron.jpg");
    Image* page4 = new Image(L"House.png");

    // Save the first page (frame).
    parameterValue = EncoderValueMultiFrame;
    stat = multi->Save(L"MultiFrame.tif", &encoderClsid, &encoderParameters);
    if(stat == Ok)
```

```

    printf("Page 1 saved successfully.\n");

    // Save the second page (frame).
    parameterValue = EncoderValueFrameDimensionPage;
    stat = multi->SaveAdd(page2, &encoderParameters);
    if(stat == Ok)
        printf("Page 2 saved successfully.\n");

    // Save the third page (frame).
    parameterValue = EncoderValueFrameDimensionPage;
    stat = multi->SaveAdd(page3, &encoderParameters);
    if(stat == Ok)
        printf("Page 3 saved successfully.\n");

    // Save the fourth page (frame).
    parameterValue = EncoderValueFrameDimensionPage;
    stat = multi->SaveAdd(page4, &encoderParameters);
    if(stat == Ok)
        printf("Page 4 saved successfully.\n");

    // Close the multiframe file.
    parameterValue = EncoderValueFlush;
    stat = multi->SaveAdd(&encoderParameters);
    if(stat == Ok)
        printf("File closed successfully.\n");

    delete multi;
    delete page2;
    delete page3;
    delete page4;
    GdiplusShutdown(gdiplusToken);
    return 0;
}

```

### 从多帧图像中复制单帧

下面的例子从多帧 TIFF 文件中获取单独的帧。在 TIFF 文件被创建的时候，单独的帧被加到页面维数中（参见[创建和保存多帧图像](#)）。代码将显示 4 页中的每一页并且将它们分别保存为一个单独的 PNG 磁盘文件。

代码从一个多帧 TIFF 文件构建了一个 **Image** 对象。为了获取单独帧（页面），代码调用了 **Image** 对象的 **SelectActiveFrame** 方法。**SelectActiveFrame** 方法第一个参数是一个全球唯一标识符（GUID）的地址，它指定了先前加入到多帧 TIFF 文件中的帧的维度。**FrameDimensionPage** 的 GUID 定义在头文件 **Gdiplusimaging.h** 中。该头文件中定义的其他 GUID 还有 **FrameDimensionTime** 和 **FrameDimensionResolution**。**SelectActiveFrame** 的第二个参数是一个索引值从 0 开始的期望页数。

代码依赖于辅助函数 **GetEncoderClsid**，后者用于获取一个编码器的类标识符。

```
GUID    pageGuid = FrameDimensionPage;
CLSID   encoderClsid;
Image   multi(L"Multiframe.tif");

// Get the CLSID of the PNG encoder.
GetEncoderClsid(L"image/png", &encoderClsid);

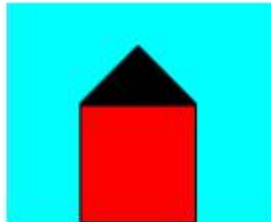
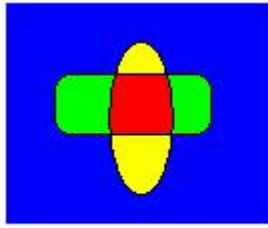
// Display and save the first page (index 0).
multi.SelectActiveFrame(&pageGuid, 0);
graphics.DrawImage(&multi, 10, 10);
multi.Save(L"Page0.png", &encoderClsid, NULL);

// Display and save the second page.
multi.SelectActiveFrame(&pageGuid, 1);
graphics.DrawImage(&multi, 200, 10);
multi.Save(L"Page1.png", &encoderClsid, NULL);

// Display and save the third page.
multi.SelectActiveFrame(&pageGuid, 2);
graphics.DrawImage(&multi, 10, 150);
multi.Save(L"Page2.png", &encoderClsid, NULL);

// Display and save the fourth page.
multi.SelectActiveFrame(&pageGuid, 3);
graphics.DrawImage(&multi, 200, 150);
multi.Save(L"Page3.png", &encoderClsid, NULL);
```

下面的插图显示了上面代码显示的单独页面。



### Alpha 混合线条和填充

在 GDI+ 中，颜色为 32 位值：alpha、红色、绿色和蓝色各 8 位。alpha 值指示颜色的透明度，即颜色与背景色的混合程度。Alpha 值的范围是 0 到 255，其中 0 表示完全透明的颜色，255 表示完全不透明的颜色。

Alpha 混合是源颜色数据和背景颜色数据之间逐个像素的混合。给定源颜色的三个分量（红色、绿色和蓝色）都按照以下公式与背景颜色的相应分量混合：

$$\text{显示颜色} = \text{源颜色} \times \alpha / 255 + \text{背景颜色} \times (255 - \alpha) / 255$$

例如，假设源颜色的红色分量是 150，背景颜色的红色分量是 100。如果 alpha 值是 200，则结果颜色的红色分量按以下公式计算：

$$150 \times 200 / 255 + 100 \times (255 - 200) / 255 = 139$$

### 绘制不透明和半透明的线条

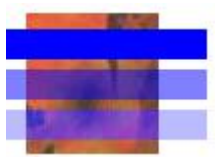
在绘制线条时，必须将 Pen 对象地址传递给 Graphics 类的 DrawLine 方法。Pen 构造函数的参数之一是 Color 对象。若要绘制不透明的线条，请将颜色的 alpha 分量设置为 255。要绘制半透明的线条，请将 alpha 分量设置为 1 到 254 之间的任意值。

在背景上绘制半透明的线条时，线条的颜色与背景的颜色相混合。alpha 分量指定线条颜色和背景颜色的混合方式：alpha 值越接近于 0，背景颜色的权重越大；alpha 值越接近于 255，线条颜色的权重越大。

下面的示例先绘制位图，然后以该位图作为背景再绘制三条线条。第一个线条的 alpha 分量的值是 255，因此它是不透明的。第二个和第三个线条的 alpha 分量的值为 128，因此它们是半透明的，您可透过线条看到背景图像。调用 SetCompositingQuality 导致在混合第三个线条时伴随灰度校正。

```
Image image(L"Texture1.jpg");
graphics.DrawImage(&image, 10, 5, image.GetWidth(), image.GetHeight());
Pen opaquePen(Color(255, 0, 0, 255), 15);
Pen semiTransPen(Color(128, 0, 0, 255), 15);
graphics.DrawLine(&opaquePen, 0, 20, 100, 20);
graphics.DrawLine(&semiTransPen, 0, 40, 100, 40);
graphics.SetCompositingQuality(CompositingQualityGammaCorrected);
graphics.DrawLine(&semiTransPen, 0, 60, 100, 60);
```

下面的插图显示下面代码的输出。



### 用不透明和半透明的画笔绘制

在填充形状时，必须将 **Brush** 对象传递给 **Graphics** 类的某个填充方法。**SolidBrush** 构造函数的一个参数是 **Color** 对象。若要填充不透明的形状，请将颜色的 **alpha** 分量设置为 255。若要填充半透明形状，请将 **alpha** 分量设置为 1 到 254 之间的任意值。

在填充半透明形状时，形状的颜色与背景的颜色相混合。**Alpha** 分量指定形状颜色和背景颜色的混合方式。**Alpha** 值越接近于 0，背景颜色的权重越大；**Alpha** 值越接近于 255，形状颜色的权重越大。

下面的示例绘制一个位图，然后填充三个遮盖该位图的椭圆。第一个椭圆的 **alpha** 分量的值是 255，因此它是不透明的。第二个和第三个椭圆的 **alpha** 分量是 128，因此它们是半透明的，所以您可透过椭圆看到背景图像。调用 **Graphics::SetCompositingQuality** 导致在混合第三个椭圆的同时进行灰度校正。

```
Image image(L"Texture1.jpg");
graphics.DrawImage(&image, 50, 50, image.GetWidth(), image.GetHeight());
SolidBrush opaqueBrush(Color(255, 0, 0, 255));
SolidBrush semiTransBrush(Color(128, 0, 0, 255));
graphics.FillEllipse(&opaqueBrush, 35, 45, 45, 30);
graphics.FillEllipse(&semiTransBrush, 86, 45, 45, 30);
graphics.SetCompositingQuality(CompositingQualityGammaCorrected);
graphics.FillEllipse(&semiTransBrush, 40, 90, 86, 30);
```

下面的插图显示以下代码的输出。



## 使用复合模式控制 Alpha 混合

有时可能希望创建具有以下特征的离屏位图：

- 颜色的 alpha 值小于 255。
- 在创建位图时颜色之间不进行 alpha 混合。
- 在显示完成的位图时，在显示设备上，位图中的颜色与背景色进行 alpha 混合。

若要创建这样的位图，请构造一个空白的 **Bitmap** 对象，然后基于该位图构造 **Graphics** 对象。将 **Graphics** 对象的复合模式设置为 **CompositingModeSourceCopy**。

下面的示例创建一个基于 **Bitmap** 对象的 **Graphics** 对象。该代码使用 **Graphics** 对象和两个半透明画笔 (**alpha = 160**) 在该位图上绘画。该代码使用半透明画笔填充红色椭圆和绿色椭圆。绿色椭圆与红色椭圆重叠，但是绿色并不与红色混合，这是因为 **Graphics** 对象的复合模式已设置为 **CompositingModeSourceCopy**。

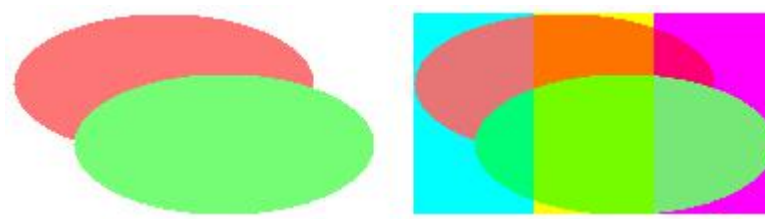
紧接着，该代码准备调用 **BeginPaint** 以及创建一个基于设备场景的 **Graphics** 对象在屏幕上进行绘图。该代码在屏幕上绘制该位图两次：一次是在白色背景上，一次是在多色背景上。位图中属于两个椭圆的像素的 alpha 分量的值是 160，因此这些椭圆与屏幕上的背景色相混合。

```
// Create a blank bitmap.
Bitmap bitmap(180, 100);
// Create a Graphics object that can be used to draw on the bitmap.
Graphics bitmapGraphics(&bitmap);
// Create a red brush and a green brush, each with an alpha value of 160.
SolidBrush redBrush(Color(210, 255, 0, 0));
SolidBrush greenBrush(Color(210, 0, 255, 0));
// Set the compositing mode so that when overlapping ellipses are drawn,
// the colors of the ellipses are not blended.
bitmapGraphics.SetCompositingMode(CompositingModeSourceCopy);
// Fill an ellipse using a red brush that has an alpha value of 160.
bitmapGraphics.FillEllipse(&redBrush, 0, 0, 150, 70);
// Fill a second ellipse using green brush that has an alpha value of 160.
// The green ellipse overlaps the red ellipse, but the green is not
// blended with the red.
bitmapGraphics.FillEllipse(&greenBrush, 30, 30, 150, 70);
// Prepare to draw on the screen.
hdc = BeginPaint(hWnd, &ps);
Graphics* pGraphics = new Graphics(hdc);
pGraphics->SetCompositingQuality(CompositingQualityGammaCorrected);
// Draw a multicolored background.
SolidBrush brush(Color((ARGB)Color::Aqua));
pGraphics->FillRectangle(&brush, 200, 0, 60, 100);
brush.SetColor(Color((ARGB)Color::Yellow));
pGraphics->FillRectangle(&brush, 260, 0, 60, 100);
```

```
brush.SetColor(Color((ARGB)Color::Fuchsia));
pGraphics->FillRectangle(&brush, 320, 0, 60, 100);

// Display the bitmap on a white background.
pGraphics->DrawImage(&bitmap, 0, 0);
// Display the bitmap on a multicolored background.
pGraphics->DrawImage(&bitmap, 200, 0);
delete pGraphics;
EndPoint(hWnd, &ps);
```

下面的插图显示代码示例的输出。请注意，这些椭圆与背景相混合，但椭圆之间不进行混合。



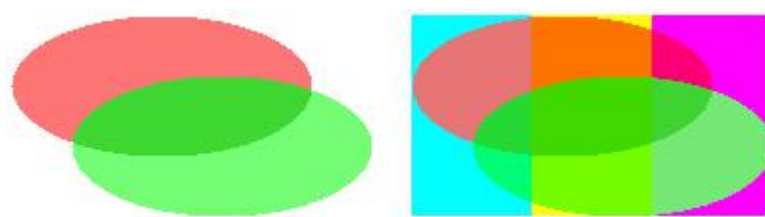
前面的代码示例包含这条语句：

```
bitmapGraphics.SetCompositingMode(CompositingModeSourceCopy);
```

如果您希望这些椭圆除了与背景相混合外，椭圆之间也相互混合，请将上述语句更改为如下语句：

```
bitmapGraphics.SetCompositingMode(CompositingModeSourceOver);
```

下面的插图显示修改后的代码的输出。



### 使用颜色矩阵设置图像中的 Alpha 值

Bitmap 类（从 Image 类继承）和 ImageAttributes 类提供用于获取和设置像素值的函数。可使用 ImageAttributes 类修改整个图像的 alpha 值，或可调用 Bitmap 类的 SetPixel 方法修改单个像素的值。关于设置单个像素值的更多信息，请参考[设置单个像素的 alpha 值](#)。

下面的例子绘制一条宽黑线，然后显示一幅不透明的位图覆盖部分线条。

```
Bitmap bitmap(L"Texture1.jpg");
```



```

Pen pen(Color(255, 0, 0, 0), 25);

// First draw a wide black line.

graphics.DrawLine(&pen, Point(10, 35), Point(200, 35));

// Now draw an image that covers part of the black line.

graphics.DrawImage(&bitmap,

    Rect(30, 0, bitmap.GetWidth(), bitmap.GetHeight()));

```

下图所示为结果图像，它绘制在点(30,0)的位置。注意纯黑的线条被图片完全遮盖。



**ImageAttributes** 类具有许多可用于在呈现过程中修改图像的属性。在下面的示例中，**ImageAttributes** 对象用于将所有的 **alpha** 值设置为原来的 80%。这是通过初始化一个颜色矩阵并将矩阵中的 **alpha** 缩放值设置为 0.8 来实现的。颜色矩阵的地址传递给 **ImageAttributes** 对象的 **SetColorMatrix** 方法，而 **ImageAttributes** 对象传递给 **Graphics** 对象的 **DrawString** 方法。

```

// Create a Bitmap object and load it with the texture image.
Bitmap bitmap(L"Texture1.jpg");
Pen pen(Color(255, 0, 0, 0), 25);
// Initialize the color matrix.
// Notice the value 0.8 in row 4, column 4.
ColorMatrix colorMatrix = {1.0f, 0.0f, 0.0f, 0.0f, 0.0f,
                           0.0f, 1.0f, 0.0f, 0.0f, 0.0f,
                           0.0f, 0.0f, 1.0f, 0.0f, 0.0f,
                           0.0f, 0.0f, 0.0f, 0.8f, 0.0f,
                           0.0f, 0.0f, 0.0f, 0.0f, 1.0f};
// Create an ImageAttributes object and set its color matrix.
ImageAttributes imageAtt;
imageAtt.SetColorMatrix(&colorMatrix, ColorMatrixFlagsDefault,
    ColorAdjustTypeBitmap);
// First draw a wide black line.
graphics.DrawLine(&pen, Point(10, 35), Point(200, 35));
// Now draw the semitransparent bitmap image.
INT iWidth = bitmap.GetWidth();
INT iHeight = bitmap.GetHeight();
graphics.DrawImage(
    &bitmap,

```

```
Rect(30, 0, iWidth, iHeight), // Destination rectangle
0, // Source rectangle X
0, // Source rectangle Y
iWidth, // Source rectangle width
iHeight, // Source rectangle height
UnitPixel,
&imageAtt);
```

在呈现过程中，位图中的各个 **alpha** 值被转换成它们的原始值的 80%，这将导致与背景相混合的图像。正如下面的插图所显示的那样，位图图像看上去是透明的：您可透过它看到纯黑的线条。



当图像位于背景的白色部分之上时，图像就与白色相混合。在图像与黑色线条的相交处，图像与黑色相混合。

### 设置单个像素的 **alpha** 值

上面的例子中采用颜色矩阵来设置一个图像的 **alpha** 值是一种非破坏性的改变图像 **alpha** 值的方法。上面的例子将图片半透明呈现出来但是位图中的像素数据并没有改变。**Alpha** 值只在渲染过程中才被改变。

下面的例子显示了如何改变单个像素的 **alpha** 值。该代码实际改变了位图对象中单个点的 **alpha** 信息。这种方法比采用颜色矩阵和 **ImageAttributes** 对象的方法要慢很多，但是允许你控制位图的单个像素。

```
INT iWidth = bitmap.GetWidth();
INT iHeight = bitmap.GetHeight();
Color color, colorTemp;
for(INT iRow = 0; iRow < iHeight; iRow++)
{
    for(INT iColumn = 0; iColumn < iWidth; iColumn++)
    {
        bitmap.GetPixel(iColumn, iRow, &color);
        colorTemp.SetValue(color.MakeARGB(
            (BYTE)(255 * iColumn / iWidth),
            color.GetRed(),
            color.GetGreen(),
            color.GetBlue()));
        bitmap.SetPixel(iColumn, iRow, colorTemp);
    }
}
```

```
// First draw a wide black line.
Pen pen(Color(255, 0, 0, 0), 25);
graphics.DrawLine(&pen, 10, 35, 200, 35);
// Now draw the modified bitmap.
graphics.DrawImage(&bitmap, 30, 0, iWidth, iHeight);
```

下图所示为结果图像：



上例中采用了嵌套循环来改变位图中每个像素的 **alpha** 值。对于每个像素，**Bitmap::GetPixel** 获取其当前颜色，**Color::SetValue** 创建一个包含新的 **alpha** 值得临时颜色值，然后 **Bitmap::SetPixel** 设置这个新的颜色。**Alpha** 值的设定取决于位图的列。第一列的 **alpha** 值为 0，最末一列的 **alpha** 值为 255。因此结果图片看起来左边完全透明而右边完全不透明。

**Bitmap::GetPixel** 和 **Bitmap::SetPixel** 使你可以控制单个像素的值。然而，使用这两个方法没有使用 **ImageAttributes** 类和颜色矩阵处理速度快。

## 使用字体和文本

GDI+ 提供多个类构成了绘制文本的基础。**Graphics** 类有多个允许指定文本的各种特征（如位置、边框、字体和格式）的 **DrawString** 方法。涉及文本呈现的其他类包括 **FontFamily**、**Font**、**StringFormat**、**InstalledFontCollection** 和 **PrivateFontCollection**。

### 构造字体系列和字体

GDI+ 将字样相同但字形不同的字体分组为字体系列。例如，**Arial** 字体系列中包含以下字体：

- Arial Regular
- Arial Bold
- Arial Italic
- Arial Bold Italic

GDI+ 使用四种字形形成字体系列：常规、粗体、倾斜和粗斜体。像 **narrow** 和 **rounded** 之类的形容词不被视为字形；而是作为字体系列名的一部分。例如，**Arial Narrow** 是包含以下成员的字体系列：

- Arial Narrow Regular
- Arial Narrow Bold
- Arial Narrow Italic
- Arial Narrow Bold Italic

在可以使用 GDI+ 绘制文本之前，您需要构造一个 **FontFamily** 对象和一个 **Font** 对象。**FontFamily** 对象指定字样（例如 **Arial**），而 **Font** 对象指定字号、字形和单位。

下面的示例构造一个字号为 16 像素、常规字形的 **Arial** 字体。

```
FontFamily fontFamily(L"Arial");
Font font(&fontFamily, 16, FontStyleRegular, UnitPixel);
```

在上面的代码中，传递给 **Font** 构造函数的第一个参数是 **FontFamily** 对象。第二个参数指定字体大小，其单位由第四个参数确定。第三个参数确定字形。

**Pixel** 为 **GraphicsUnit** 枚举的一个成员，**Regular** 是 **FontStyle** 枚举的一个成员。这两个枚举都定义在 **Gdiplusenums.h** 头文件中。

## 绘制文本

您可以使用 **Graphics** 类的 **DrawString** 方法在指定位置绘制文本或者在一个指定矩形内绘制文本。

- 在指定位置绘制文本

若要在指定位置绘制文本，您需要 **Graphics**、**FontFamily**、**Font**、**PointF** 和 **Brush** 对象。

下面的代码将在(30, 10)的位置绘制一个字符串“Hello”。字体系列为“Times New Roman”。字体是字体系列的一个独立成员，这里是 **Times New Roman**、24 像素、规则样式。假定 *graphics* 是一个已经存在的 **Graphics** 对象。

```
FontFamily fontFamily(L"Times New Roman");
Font font(&fontFamily, 24, FontStyleRegular, UnitPixel);
PointF pointF(30.0f, 10.0f);
SolidBrush solidBrush(Color(255, 0, 0, 255));

graphics.DrawString(L"Hello", -1, &font, pointF, &solidBrush);
```

下面的插入显示了上面代码的输出结果：



前面的例子中，**FontFamily** 构造函数通过一个字符串参数来指定字体系列。**FontFamily** 的地址又作为 **Font** 构造函数的第一个参数传入。**Font** 构造函数的第二个参数表示字体大小，其单位由第四个参数给出。第三个参数给出字体样式（规则、粗体、斜体等）。

**DrawString** 方法接受 5 个参数。第一个参数是要绘制的文本。第二个参数是该文本的长度（单位是字符数，而不是字节）。如果字符串是以 **Null** 结束的，你可以传入 -1。第三个参数是前面构造的 **Font**

对象的地址。第四个参数是一个 **PointF** 对象，它包含了字符串左上角的坐标。第五个参数是一个 **SolidBrush** 对象的地址，他将用于填充该字符串。

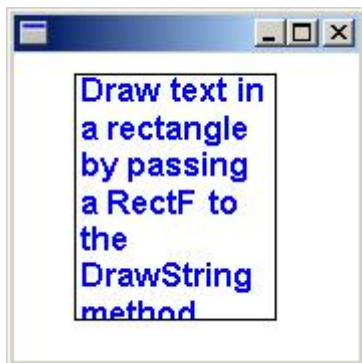
- 在矩形中绘制文本

**Graphics** 类中的其中一个 **DrawString** 方法有一个 **RectF** 参数。通过调用这个方法，你可以在指定的矩形中绘制换行文本。您还将需要 **Graphics**、**FontFamily**、**Font**、**RectF** 和 **Brush** 对象。

下面的例子将创建一个左上角在(30, 10)、宽度为 100、高度为 122 的矩形。然后代码在矩形中绘制一个字符串。字符串因为矩形的限制而采用不破坏单个单词的方式折行显示。

```
WCHAR string[] =  
    L"Draw text in a rectangle by passing a RectF to the DrawString method.>";  
  
FontFamily    fontFamily(L"Arial");  
Font          font(&fontFamily, 12, FontStyleBold, UnitPoint);  
RectF         rectF(30.0f, 10.0f, 100.0f, 122.0f);  
SolidBrush    solidBrush(Color(255, 0, 0, 255));  
  
graphics.DrawString(string, -1, &font, rectF, NULL, &solidBrush);  
  
Pen pen(Color(255, 0, 0, 0));  
graphics.DrawRectangle(&pen, rectF);
```

下图显示使用 **DrawString** 方法在矩形中绘制的文本的输出。



在前面的例子中，**DrawString** 方法的第四个参数是一个 **RectF** 对象，它制定了用于限制文本输出的矩形。第五个类型为 **StringFormat** 的参数为 **Null**，因为并不需要特殊的字符串格式。

## 格式化文本

若要为文本设置特殊格式，先初始化一个 **StringFormat** 对象，然后传递该对象的地址给 **Graphics** 类的 **DrawString** 方法。

若要在矩形中绘制格式化文本，你需要 **Graphics**、**FontFamily**、**Font**、**RectF**、**StringFormat** 和 **Brush** 对象。

- 对齐文本

下面的例子将绘制在矩形中绘制文本。文本的每行都居中（从左到右），整个文本块在矩形中居中（从上到下）。

```
WCHAR string[] =
    L"Use StringFormat and RectF objects to center text in a rectangle.";
FontFamily    fontFamily(L"Arial");
Font          font(&fontFamily, 12, FontStyleBold, UnitPoint);
RectF         rectF(30.0f, 10.0f, 120.0f, 140.0f);
StringFormat   stringFormat;
SolidBrush    solidBrush(Color(255, 0, 0, 255));

// Center-justify each line of text.
stringFormat.SetAlignment(StringAlignmentCenter);

// Center the block of text (top to bottom) in the rectangle.
stringFormat.SetLineAlignment(StringAlignmentCenter);

graphics.DrawString(string, -1, &font, rectF, &stringFormat, &solidBrush);

Pen pen(Color(255, 0, 0, 0));
graphics.DrawRectangle(&pen, rectF);
```

下图显示了该矩形和居中的文本。



上面的例子中调用了 **StringFormat** 对象的 2 个方法：**SetAlignment** 和 **SetLineAlignment**。**SetAlignment** 调用指定矩形中的每行文本是否居中显示。**SetLineAlignment** 调用指定矩形中的文本块是否居中显示（垂直方向）。

**StringAlignmentCenter** 是 **StringAlignment** 枚举成员之一，定义于头文件 **Gdiplusenums.h** 中。

- 设置制表位

通过调用 **StringFormat** 对象的 **SetTabStops** 方法,然后将该 **StringFormat** 对象的地址传递给 **Graphics** 类的 **DrawString** 方法,可以设置文本的制表位。

下面的示例在 150、250 和 350 处设置制表位。然后,该代码显示用制表符分隔的名称和测验得分的列表。

```
WCHAR string[150] =
    L"Name\tTest 1\tTest 2\tTest 3\n";

StringCchCatW(string, 150, L"Joe\t95\t88\t91\n");
StringCchCatW(string, 150, L"Mary\t98\t84\t90\n");
StringCchCatW(string, 150, L"Sam\t42\t76\t98\n");
StringCchCatW(string, 150, L"Jane\t65\t73\t92\n");

FontFamily    fontFamily(L"Courier New");
Font          font(&fontFamily, 12, FontStyleRegular, UnitPoint);
RectF         rectF(10.0f, 10.0f, 450.0f, 100.0f);
StringFormat  stringFormat;
SolidBrush    solidBrush(Color(255, 0, 0, 255));
REAL          tabs[] = {150.0f, 100.0f, 100.0f};

stringFormat.SetTabStops(0.0f, 3, tabs);

graphics.DrawString(string, -1, &font, rectF, &stringFormat, &solidBrush);

Pen pen(Color(255, 0, 0, 0));
graphics.DrawRectangle(&pen, rectF);
```

下面的插图显示用制表符分隔的文本。

Name	Test 1	Test 2	Test 3
Joe	95	88	91
Mary	98	84	90
Sam	42	76	98
Jane	65	73	92

上例中 **SetTabStops** 方法传入了 3 个参数。第三个参数是包含制表位偏移量的数组的地址。第二个参数表示数组中有 4 个偏移量。第一个参数为 0, 表示第一个偏移量从位置 0 开始度量, 即外框矩形的左边界。

- 绘制垂直文本

可使用 `StringFormat` 对象指定在垂直方向而不是在水平方向绘制文本。

下面的示例将值 `StringFormatFlagsDirectionVertical` 传给 `StringFormat` 对象的 `SetFormatFlags` 方法。该 `StringFormat` 对象地址被传递给 `Graphics` 类的 `DrawString` 方法。`StringFormatFlagsDirectionVertical` 值是 `StringFormatFlags` 枚举的成员，定义在头文件 `Gdiplusenums.h` 中。

```
WCHAR string[] = L"Vertical text";

FontFamily   fontFamily(L"Lucida Console");
Font         font(&fontFamily, 14, FontStyleRegular, UnitPoint);
PointF       pointF(40.0f, 10.0f);
StringFormat stringFormat;
SolidBrush   solidBrush(Color(255, 0, 0, 255));

stringFormat.SetFormatFlags(StringFormatFlagsDirectionVertical);

graphics.DrawString(string, -1, &font, pointF, &stringFormat, &solidBrush);
```

下面的插图显示竖排文本。



### 枚举已安装的字体

`InstalledFontCollection` 类继承自 `FontCollection` 抽象基类。可使用 `InstalledFontCollection` 对象枚举计算机上安装的字体。`InstalledFontCollection` 对象的 `GetFamilies` 方法返回一个 `FontFamily` 对象数组。在你调用 **GetFamilies** 之前，你必须分配足够大的缓冲区用以存储这个数组。可以通过调用 `GetFamilyCount` 方法同时乘以 **sizeof(FontFamily)** 的值来获得你所需要的缓冲区的大小。

下面的示例列出计算机上安装的所有字体系列的名称。该代码通过调用 `GetFamilyName` 方法检索每个 `FontFamily` 对象的 `Name` 属性，这些对象存放在由 **GetFamilies** 方法返回的数组中。检索到字体系列名称时，将它们连接起来以形成用逗号分隔的列表。然后，`Graphics` 类的 `DrawString` 方法在矩形中绘制此用逗号分隔的列表。



```

FontFamily  fontFamily(L"Arial");
Font        font(&fontFamily, 8, FontStyleRegular, UnitPoint);
RectF       rectF(10.0f, 10.0f, 500.0f, 500.0f);
SolidBrush  solidBrush(Color(255, 0, 0, 0));

INT         count = 0;
INT         found = 0;
WCHAR       familyName[LF_FACESIZE]; // enough space for one family name
WCHAR*      familyList = NULL;
FontFamily* pFontFamily = NULL;

InstalledFontCollection installedFontCollection;

// How many font families are installed?
count = installedFontCollection.GetFamilyCount();

// Allocate a buffer to hold the array of FontFamily
// objects returned by GetFamilies.
pFontFamily = new FontFamily[count];

// Get the array of FontFamily objects.
installedFontCollection.GetFamilies(count, pFontFamily, &found);

// The loop below creates a large string that is a comma-separated
// list of all font family names.
// Allocate a buffer large enough to hold that string.
familyList = new WCHAR[count*(sizeof(familyName)+ 3)];
StringCchCopy(familyList, 1, L "");

for(INT j = 0; j < count; ++j)
{
    pFontFamily[j].GetFamilyName(familyName);
    StringCchCatW(familyList, count*(sizeof(familyName)+ 3), familyName);
    StringCchCatW(familyList, count*(sizeof(familyName)+ 3), L", ");
}

// Draw the large string (list of all families) in a rectangle.
graphics.DrawString(
    familyList, -1, &font, rectF, NULL, &solidBrush);

delete pFontFamily;
delete familyList;

```

如果运行该示例代码，输出将与下面的插图所显示的情况相似。



## 创建专用的字体集合

**PrivateFontCollection** 类从 **FontCollection** 抽象基类继承。可使用 **PrivateFontCollection** 对象维护应用程序专用的字体集。专用的字体集合既可包括计算机上安装的系统字体，又可包括计算机上尚未安装的字体。若要向专用的字体集内添加字体文件，请调用 **PrivateFontCollection** 对象的 **AddFontFile** 方法。

**PrivateFontCollection** 对象的 **GetFamilies** 方法返回一个 **FontFamily** 对象数组。在你调用该方法之前，必须先分配足够大的缓冲区以便容纳该数组。要获取所需缓冲区大小，你可以通过调用 **GetFamilyCount** 方法然后乘以 **sizeof(FontFamily)** 的值算出。

专用字体集合中字体系列的数量不必与已添加到字体集内的字体文件的数量相同。例如，假定将 **ArialBd.ttf**、**Times.ttf** 和 **TimesBd.ttf** 文件添加到一个集合内。该字体集合内将有三个文件，但是只有两个系列，这是因为 **Times.ttf** 和 **TimesBd.ttf** 属于相同的系列。

下面的示例将以下三个字体文件添加到 **PrivateFontCollection** 对象中：

-  C:\WINNT\Fonts\Arial.ttf (Arial, 常规)
-  C:\WINNT\Fonts\CourBI.ttf (Courier New, 加粗倾斜)
-  C:\WINNT\Fonts\TimesBd.ttf (Times New Roman, 加粗)

代码调用 **PrivateFontCollection** 对象的 **GetFamilyCount** 方法来判断专用字体集合中的字体系列的个数，然后调用 **GetFamilies** 获取 **FontFamily** 对象数组。

对于集合内的每个 **FontFamily** 对象，代码调用 **IsStyleAvailable** 方法以确定各种字形（常规、加粗、倾斜、加粗倾斜、下划线和删除线）是否可用。传递给 **IsStyleAvailable** 方法的参数为 **FontStyle** 枚举的成员，定义在头文件 **Gdiplusenums.h** 中。

If a given family/style combination is available, a **Font** object is constructed using that family and style. The first argument passed to the **Font** constructor is the font family name (not a **FontFamily** object as is the case for other variations of the **Font** constructor), and the final argument is the address of the **PrivateFontCollection** object. After the **Font** object is constructed, its address is passed to the **DrawString** method of the **Graphics** class to display the family name along with the name of the style.

如果给定的字体/字形组合可用，则使用该字体/字形构造 **Font** 对象。传递给 **Font** 构造函数的第一个参数是字体名称（不是用于 **Font** 构造函数的其他变体的 **FontFamily** 对象），最后一个参数是 **PrivateFontCollection** 对象的地址。在构造 **Font** 对象之后，它将被传递给 **Graphics** 类的 **DrawString** 方法，以显示字体名称和字形名称。

```
#define MAX_STYLE_SIZE 20
#define MAX_FACEANDSTYLE_SIZE (LF_FACESIZE + MAX_STYLE_SIZE + 2)

PointF      pointF(10.0f, 0.0f);
SolidBrush  solidBrush(Color(255, 0, 0, 0));
INT          count = 0;
INT          found = 0;
WCHAR        familyName[LF_FACESIZE];
WCHAR        familyNameAndStyle[MAX_FACEANDSTYLE_SIZE];
FontFamily*  pFontFamily;
PrivateFontCollection privateFontCollection;

// Add three font files to the private collection.
privateFontCollection.AddFontFile(L"c:\\Winnt\\Fonts\\Arial.ttf");
privateFontCollection.AddFontFile(L"c:\\Winnt\\Fonts\\CourBI.ttf");
privateFontCollection.AddFontFile(L"c:\\Winnt\\Fonts\\TimesBd.ttf");

// How many font families are in the private collection?
count = privateFontCollection.GetFamilyCount();

// Allocate a buffer to hold the array of FontFamily
// objects returned by GetFamilies.
pFontFamily = new FontFamily[count];

// Get the array of FontFamily objects.
privateFontCollection.GetFamilies(count, pFontFamily, &found);

// Display the name of each font family in the private collection
// along with the available styles for that font family.
for(INT j = 0; j < count; ++j)
{
    // Get the font family name.
    pFontFamily[j].GetFamilyName(familyName);
```

```

// Is the regular style available?
if(pFontFamily[j].IsStyleAvailable(FontStyleRegular))
{
    StringCchCopyW(familyNameAndStyle, LF_FACESIZE, familyName);
    StringCchCatW(familyNameAndStyle, MAX_FACEANDSTYLE_SIZE, L" Regular");

    Font* pFont = new Font(
        familyName, 16, FontStyleRegular, UnitPixel,
&privateFontCollection);

    graphics.DrawString(familyNameAndStyle, -1, pFont, pointF, &solidBrush);

    pointF.Y += pFont->GetHeight(0.0f);
    delete(pFont);
}

// Is the bold style available?
if(pFontFamily[j].IsStyleAvailable(FontStyleBold))
{
    StringCchCopyW(familyNameAndStyle, LF_FACESIZE, familyName);
    StringCchCatW(familyNameAndStyle, MAX_FACEANDSTYLE_SIZE, L" Bold");

    Font* pFont = new Font(
        familyName, 16, FontStyleBold, UnitPixel, &privateFontCollection);

    graphics.DrawString(familyNameAndStyle, -1, pFont, pointF, &solidBrush);

    pointF.Y += pFont->GetHeight(0.0f);
    delete(pFont);
}

// Is the italic style available?
if(pFontFamily[j].IsStyleAvailable(FontStyleItalic))
{
    StringCchCopyW(familyNameAndStyle, LF_FACESIZE, familyName);
    StringCchCatW(familyNameAndStyle, MAX_FACEANDSTYLE_SIZE, L" Italic");

    Font* pFont = new Font(
        familyName, 16, FontStyleItalic, UnitPixel, &privateFontCollection);

    graphics.DrawString(familyNameAndStyle, -1, pFont, pointF, &solidBrush);

    pointF.Y += pFont->GetHeight(0.0f);

```

```

        delete(pFont);
    }

    // Is the bold italic style available?
    if(pFontFamily[j].IsStyleAvailable(FontStyleBoldItalic))
    {
        StringCchCopyW(familyNameAndStyle, LF_FACESIZE, familyName);
        StringCchCatW(familyNameAndStyle, MAX_FACEANDSTYLE_SIZE, L" BoldItalic");

        Font* pFont = new Font(familyName, 16,
            FontStyleBoldItalic, UnitPixel, &privateFontCollection);

        graphics.DrawString(familyNameAndStyle, -1, pFont, pointF, &solidBrush);

        pointF.Y += pFont->GetHeight(0.0f);
        delete(pFont);
    }

    // Is the underline style available?
    if(pFontFamily[j].IsStyleAvailable(FontStyleUnderline))
    {
        StringCchCopyW(familyNameAndStyle, LF_FACESIZE, familyName);
        StringCchCatW(familyNameAndStyle, MAX_FACEANDSTYLE_SIZE, L" Underline");

        Font* pFont = new Font(familyName, 16,
            FontStyleUnderline, UnitPixel, &privateFontCollection);

        graphics.DrawString(familyNameAndStyle, -1, pFont, pointF, &solidBrush);

        pointF.Y += pFont->GetHeight(0.0);
        delete(pFont);
    }

    // Is the strikethrough style available?
    if(pFontFamily[j].IsStyleAvailable(FontStyleStrikethrough))
    {
        StringCchCopyW(familyNameAndStyle, LF_FACESIZE, familyName);
        StringCchCatW(familyNameAndStyle, MAX_FACEANDSTYLE_SIZE, L" Strikethrough");

        Font* pFont = new Font(familyName, 16,
            FontStyleStrikethrough, UnitPixel, &privateFontCollection);

        graphics.DrawString(familyNameAndStyle, -1, pFont, pointF, &solidBrush);
    }

```

```

    pointF.Y += pFont->GetHeight(0.0f);
    delete(pFont);
}

// Separate the families with white space.
pointF.Y += 10.0f;

} // for

delete pFontFamily;

```

下面的代码的输出类似于下面插图中所显示的输出。







**Arial.ttf**（在上面的代码示例中已添加到专用字体集合中）是 **Arial** 常规字形的字体文件。然而，请注意，该程序的输出显示了 **Arial** 字体系列的除了常规字形以外的几种可用字形。这是因为 **GDI+** 能从常规字形模拟加粗、倾斜和加粗倾斜字形。**GDI+** 还可从常规字形产生下划线和删除线字形。

同样，**GDI+** 可从加粗字形或倾斜字形模拟加粗倾斜字形。该程序的输出表明，即使 **TimesBd.ttf**（**Times New Roman**，加粗）是字体集合内唯一的 **Times** 文件，**Times** 系列仍可使用加粗倾斜字形。

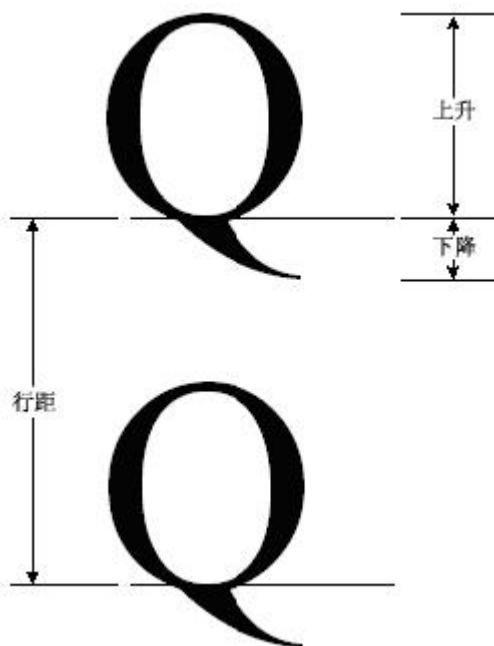
### 获取字体规格

**FontFamily** 类提供了下列方法来检索特定字体/字形组合的各种规格：

-  [GetEmHeight\(FontStyle\)](#)
-  [GetCellAscent\(FontStyle\)](#)
-  [GetCellDescent\(FontStyle\)](#)
-  [GetLineSpacing\(FontStyle\)](#)

这些方法返回的数字使用的是字体设计单位，因此，它们与特定的 **Font** 对象的大小和单位无关。

下面的插图显示了升高值、降低值和行距等各种规格。



下面的示例显示 **Arial** 字体系列常规字形的规格。该代码还创建一个大小为 **16** 像素的 **Font** 对象（基于 **Arial** 系列），并显示该特定 **Font** 对象的规格（以像素为单位）。

```
#define INFO_STRING_SIZE 100 // one line of output including null terminator
WCHAR infoString[INFO_STRING_SIZE] = L"";

UINT ascent; // font family ascent in design units
REAL ascentPixel; // ascent converted to pixels
UINT descent; // font family descent in design units
REAL descentPixel; // descent converted to pixels
UINT lineSpacing; // font family line spacing in design units
REAL lineSpacingPixel; // line spacing converted to pixels

FontFamily fontFamily(L"Arial");
Font font(&fontFamily, 16, FontStyleRegular, UnitPixel);
PointF pointF(10.0f, 10.0f);
SolidBrush solidBrush(Color(255, 0, 0, 0));

// Display the font size in pixels.
StringCchPrintf(
    infoString,
    INFO_STRING_SIZE,
    L"font.GetSize() returns %f.", font.GetSize());

graphics.DrawString(
    infoString, -1, &font, pointF, &solidBrush);
```

```

// Move down one line.
pointF.Y += font.GetHeight(0.0);

// Display the font family em height in design units.
StringCchPrintf(
    infoString,
    INFO_STRING_SIZE,
    L"fontFamily.GetEmHeight() returns %d.",
    fontFamily.GetEmHeight(FontStyleRegular));

graphics.DrawString(infoString, -1, &font, pointF, &solidBrush);

// Move down two lines.
pointF.Y += 2.0f * font.GetHeight(0.0f);

// Display the ascent in design units and pixels.
ascent = fontFamily.GetCellAscent(FontStyleRegular);

// 14.484375 = 16.0 * 1854 / 2048
ascentPixel =
    font.GetSize() * ascent / fontFamily.GetEmHeight(FontStyleRegular);

StringCchPrintf(
    infoString,
    INFO_STRING_SIZE,
    L"The ascent is %d design units, %f pixels.",
    ascent,
    ascentPixel);

graphics.DrawString(infoString, -1, &font, pointF, &solidBrush);

// Move down one line.
pointF.Y += font.GetHeight(0.0f);

// Display the descent in design units and pixels.
descent = fontFamily.GetCellDescent(FontStyleRegular);

// 3.390625 = 16.0 * 434 / 2048
descentPixel =
    font.GetSize() * descent / fontFamily.GetEmHeight(FontStyleRegular);

StringCchPrintf(
    infoString,

```



```

    INFO_STRING_SIZE,
    L"The descent is %d design units, %f pixels.",
    descent,
    descentPixel);

graphics.DrawString(infoString, -1, &font, pointF, &solidBrush);

// Move down one line.
pointF.Y += font.GetHeight(0.0f);

// Display the line spacing in design units and pixels.
lineSpacing = fontFamily.GetLineSpacing(FontStyleRegular);

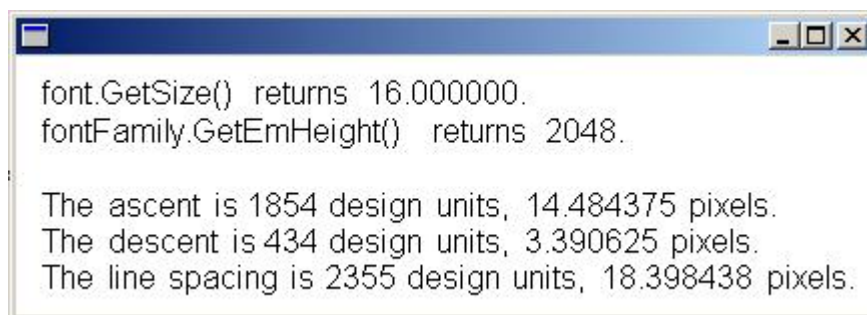
// 18.398438 = 16.0 * 2355 / 2048
lineSpacingPixel =
    font.GetSize() * lineSpacing / fontFamily.GetEmHeight(FontStyleRegular);

StringCchPrintf(
    infoString,
    INFO_STRING_SIZE,
    L"The line spacing is %d design units, %f pixels.",
    lineSpacing,
    lineSpacingPixel);

graphics.DrawString(infoString, -1, &font, pointF, &solidBrush);

```

下面的插图显示示例代码的输出。



请注意上面的插图中的前两行输出。**Font** 对象返回的大小为 **16**，**FontFamily** 对象返回的 **em** 高度为 **2,048**。这两个数字（**16** 和 **2,048**）是在字体设计单位和 **Font** 对象的单位（在本例中为像素）之间转换的关键。

例如，您可按照以下方法将升高值从设计单位转换为像素：

$$\frac{1854 \text{ 个设计单位}}{1} \times \frac{16 \text{ 个像素}}{2048 \text{ 个设计单位}} = 14.484375 \text{ 个像素}$$

前面的代码通过设置 **PointF** 对象的 **Y** 数据成员来在垂直方向上定位文本。每新增一行文本，**y** 坐标就增加 **font.Height**。**Font** 对象的 **GetHeight** 方法返回该特定 **Font** 对象的行距（以像素为单位）。在此示例中，**Height** 返回的数为 **18.398438**。请注意，这与通过将行距单位转换为像素而得到的数是相同的。

注意 **em** 高度（也叫大小或 **em** 大小）不是升高值与降低值之和。升高值与降低值之和称为单元格高度。单元格高度减去内部间隙等于 **em** 高度。单元格高度加外部间隙等于行距。

### 对文本使用消除锯齿效果

“消除锯齿”是指对绘制的图形和文本的粗糙边缘进行平滑处理以改进它们的外观或可读性。**GDI+** 既可以呈现高质量的消除锯齿的文本，也可以呈现低质量文本。**GDI+** 提供多种文本呈现的质量等级。通常，呈现的质量越高，所需的处理时间越长。

质量等级是 **Graphics** 类的一个属性。若要设置质量等级，需要调用 **Graphics** 对象的 **SetTextRenderingHint** 方法。该方法接受一个 **TextRenderingHint** 枚举成员之一，该枚举定义在头文件 **Gdiplusenums.h** 中。

**GDI+** 既提供传统的 **AntiAlias**，也提供一种基于 **Microsoft® ClearType®** 显示技术的新型 **AntiAlias**，后者只在 **Windows XP** 和 **Windows Server 2003** 中才可用。它可改善具有数字化界面的彩色 **LCD** 监视器（例如便携机的监视器和高质量纯平台式显示器）的可读性。还可稍微改善 **CRT** 屏幕的可读性。

**ClearType** 与 **LCD** 条纹的方向和顺序有关。目前，**ClearType** 只在顺序为 **RGB** 的垂直条纹中实现。如果您使用的是显示器可面向任何方向的 **Tablet PC**，或者如果您使用的屏幕可从横向旋转到纵向，则这可能是需要考虑的事项。

下面的代码示例以两种不同的质量设置绘制文本：

```
FontFamily fontFamily(L"Times New Roman");
Font        font(&fontFamily, 32, FontStyleRegular, UnitPixel);
SolidBrush  solidBrush(Color(255, 0, 0, 255));
WCHAR       string1[] = L"SingleBitPerPixel";
WCHAR       string2[] = L"AntiAlias";

graphics.SetTextRenderingHint(TextRenderingHintSingleBitPerPixel);
graphics.DrawString(string1, -1, &font, PointF(10.0f, 10.0f), &solidBrush);

graphics.SetTextRenderingHint(TextRenderingHintAntiAlias);
graphics.DrawString(string2, -1, &font, PointF(10.0f, 60.0f), &solidBrush);
```

下面的插图显示修改后的代码示例的输出。

# SingleBitPerPixel

## AntiAlias

### 构造并绘制曲线

GDI+ 支持多种类型的曲线：椭圆、弧形、基数样条和贝塞尔样条。椭圆是由其边界矩形定义的；弧是椭圆的一部分，由一个起始角和一个扫描角定义。基数样条由一系列点和张力参数定义，即曲线平滑地通过系列中的每个点，张力参数影响曲线的弯曲方式。贝塞尔样条由两个端点和两个控制点定义，即该曲线不通过控制点，但是控制点影响曲线从一个端点到另一个端点时的方向和弯曲程度。

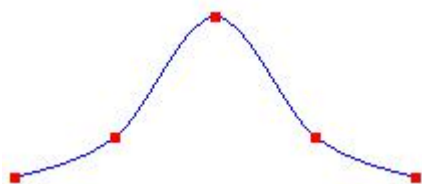
### 绘制基数样条曲线

基数样条是平滑通过一组给定点的曲线。若要绘制基数样条，请创建 **Graphics** 对象并将一个点的数组的地址传递给 **DrawCurve** 方法。下面的示例绘制了一条通过五个指定点的钟形基数样条。

```
Point points[] = {Point(0, 100),
                  Point(50, 80),
                  Point(100, 20),
                  Point(150, 80),
                  Point(200, 100)};

Pen pen(Color(255, 0, 0, 255));
graphics.DrawCurve(&pen, points, 5);
```

下面的插图显示该曲线和五个点。



使用 **Graphics** 类的 **DrawClosedCurve** 方法可绘制闭合的基数样条。在闭合的基数样条中，曲线连续通过序列中最后一个点，并与序列中的第一个点连接。

下面的示例绘制了一条通过六个指定点的闭合的基数样条。

```
Point points[] = {Point(60, 60),
                  Point(150, 80),
                  Point(200, 40),
                  Point(100, 20),
                  Point(50, 80),
                  Point(0, 100)};
```

```

    Point(180, 120),

    Point(120, 100),

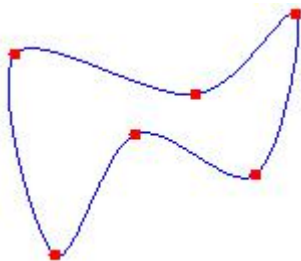
    Point(80, 160));

Pen pen(Color(255, 0, 0, 255));

graphics.DrawClosedCurve(&pen, points, 6);

```

下面的插图显示闭合的样条和六个点。



通过将张力参数传递给 **DrawCurve** 方法，可更改基数样条的弯曲方式。下面的示例绘制了三条通过同一组点的基数样条。

```

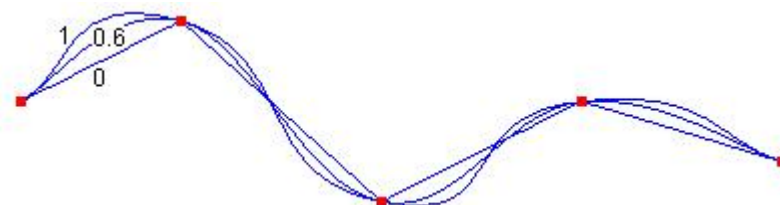
Point points[] = {Point(20, 50),
                  Point(100, 10),
                  Point(200, 100),
                  Point(300, 50),
                  Point(400, 80)};

Pen pen(Color(255, 0, 0, 255));

graphics.DrawCurve(&pen, points, 5, 0.0f); // tension 0.0
graphics.DrawCurve(&pen, points, 5, 0.6f); // tension 0.6
graphics.DrawCurve(&pen, points, 5, 1.0f); // tension 1.0

```

下面的插图显示三条样条及其张力值。请注意，当张力为 **0** 时，这些点由一条直线连接。

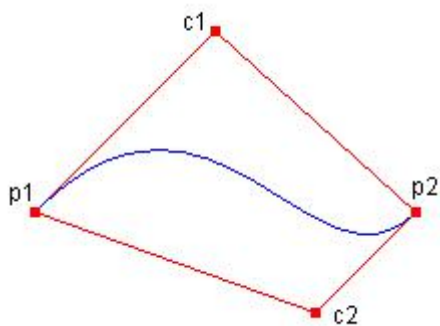


## 绘制贝塞尔样条

用四个点定义一条贝塞尔样条：一个起点、两个控制点和一个终点。下面的示例绘制了一条起点为 (10, 100)、终点为 (200, 100) 的贝塞尔样条。两个控制点分别为 (100, 10) 和 (150, 150)。

```
Point p1(10, 100);    // start point
Point c1(100, 10);    // first control point
Point c2(150, 150);   // second control point
Point p2(200, 100);   // end point
Pen pen(Color(255, 0, 0, 255));
graphics.DrawBezier(&pen, p1, c1, c2, p2);
```

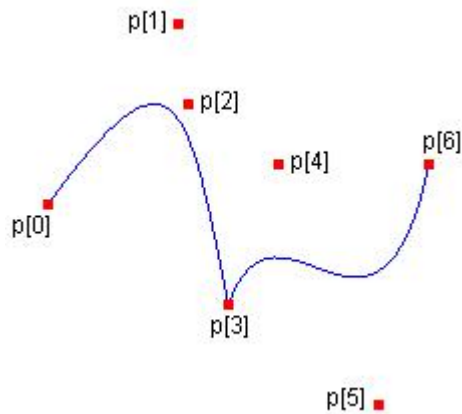
下面的插图显示产生的贝塞尔样条及其起点、控制点和终点。该插图还显示样条的凸包络，凸包络是通过将这四个点用直线相连而形成的多边形。



可使用 **Graphics** 类的 **DrawBeziers** 方法绘制相连的贝塞尔样条序列。下面的示例绘制一条由两条相连的贝塞尔样条组成的曲线。第一条贝塞尔样条的终点是第二条贝塞尔样条的起点。

```
Point p[] = {
    Point(10, 100),    // start point of first spline
    Point(75, 10),     // first control point of first spline
    Point(80, 50),     // second control point of first spline
    Point(100, 150),   // end point of first spline and
                        // start point of second spline
    Point(125, 80),    // first control point of second spline
    Point(175, 200),   // second control point of second spline
    Point(200, 80)};  // end point of second spline
Pen pen(Color(255, 0, 0, 255));
graphics.DrawBeziers(&pen, p, 7);
```

下面的插图显示相连的样条和这七个点。

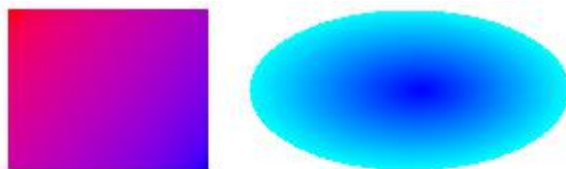


### 用渐变画刷填充形状

可借助渐变画刷用渐变的颜色填充形状。例如，可借助水平渐变画刷，从形状的左边缘到右边缘用逐渐变化的颜色来填充形状。设想这样一个矩形：它的左边缘为黑色（红色、绿色和蓝色分量均为 0）；右边为红色（这三个分量分别为 255, 0, 0）。如果矩形的宽度为 256 个像素，则给定像素的红色分量将多于其左侧的像素的红色分量。在一行中，最左边像素的颜色分量为 (0, 0, 0)；第二个像素的分量为 (1, 0, 0)；第三个为 (2, 0, 0)，依此类推，直到到达最右边的像素，它的分量为 (255, 0, 0)。这些插值颜色的值构成了颜色渐变。

当水平地、垂直地或平行指定的斜线移动到时，线性渐变改变颜色。当在路径的内部和边界来回移动时，路径渐变改变颜色。可自定义路径渐变以获得多种效果。

下面的插图显示用线性渐变画笔填充矩形，用路径渐变画笔填充椭圆。



GDI+ 提供 **LinearGradientBrush** 和 **PathGradientBrush** 类，二者都从 **Brush** 类继承。

### 创建线性渐变

GDI+ 提供水平、垂直和对角线方向线性渐变。在默认情况下，线性渐变中的颜色均匀地变化。当然，也可自定义线性渐变，使颜色非均匀变化。

- **水平线性渐变**

下面的示例使用水平线性渐变画笔填充线条、椭圆和矩形。

```
LinearGradientBrush linGrBrush(
    Point(0, 10),
```

```

Point(200, 10),
Color(255, 255, 0, 0),    // opaque red
Color(255, 0, 0, 255));  // opaque blue

Pen pen(&linGrBrush);

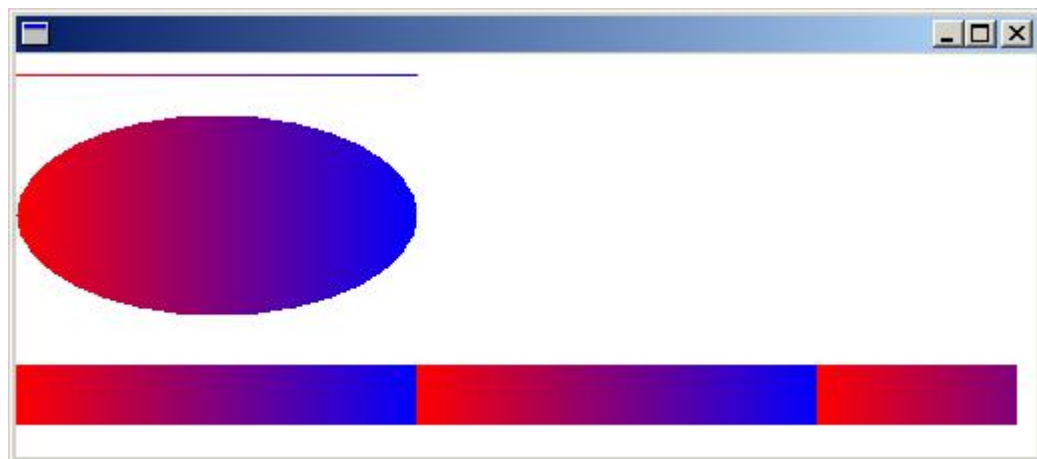
graphics.DrawLine(&pen, 0, 10, 200, 10);
graphics.FillEllipse(&linGrBrush, 0, 30, 200, 100);
graphics.FillRectangle(&linGrBrush, 0, 155, 500, 30);

```

**LinearGradientBrush** 构造函数接收四个参数：两个点和两种颜色。第一个点 (0, 10) 与第一种颜色（红色）相关联，第二个点 (200, 10) 与第二种颜色（蓝色）相关联。正如您所期望的那样，从 (0, 10) 绘制到 (200, 10) 的线条的颜色从红色逐渐变成蓝色。

点 (50, 10) 和点 (200, 10) 中的 10 无关紧要。重要的是这两个点的第二个坐标相同——它们之间的连线是水平的。当水平坐标从 0 移到 200 时，椭圆和矩形也逐渐从红色变成蓝色。

下面的插图显示线条、椭圆和矩形。请注意，当水平坐标增加到 200 以上时，颜色渐变重复其自身。



- **自定义线性渐变**

在上面的示例中，当您从水平坐标 0 移到水平坐标 200 时，颜色分量成线性变化。例如，如果某个点的第一个坐标位于 0 和 200 的正中间，则其蓝色分量将是 0 和 255 正中间的值。

GDI+ 允许调整颜色从渐变的一个边缘到另一个边缘变化的方式。假设您希望按照下表创建从黑色变到红色的渐变画笔。

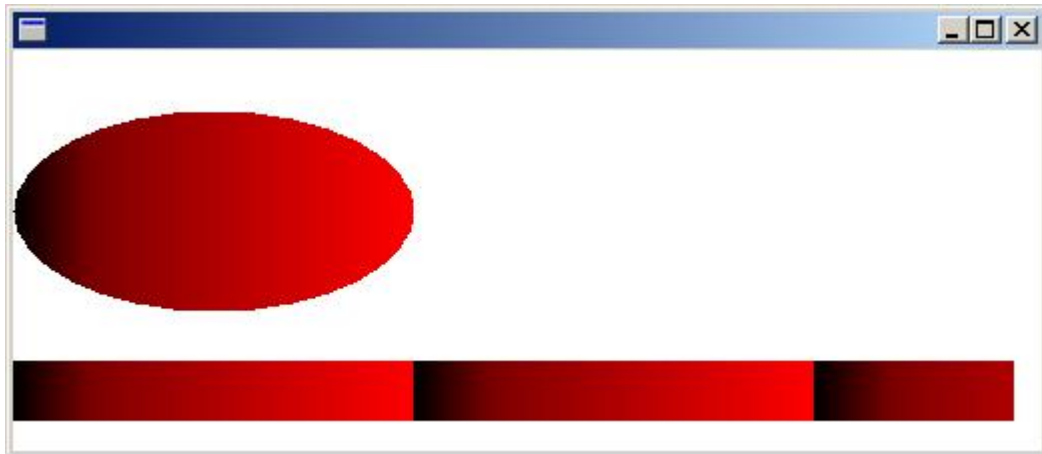
水平坐标	RGB 分量
0	(0, 0, 0)
40	(128, 0, 0)
200	(255, 0, 0)

请注意，当水平坐标才达到 0 到 200 之间的 20% 时，红色分量已达到一半亮度。

下面的示例调用 `LinearGradientBrush` 对象的 `SetBlend` 方法，以便使三个相对亮度与三个相对位置相关联。正如上表所示，相对亮度 0.5 与相对位置 0.2 相关联。该代码用渐变画笔填充椭圆和矩形。

```
LinearGradientBrush linGrBrush(  
    Point(0, 10),  
    Point(200, 10),  
    Color(255, 0, 0, 0),    // opaque black  
    Color(255, 255, 0, 0)); // opaque red  
  
REAL relativeIntensities[] = {0.0f, 0.5f, 1.0f};  
REAL relativePositions[]   = {0.0f, 0.2f, 1.0f};  
  
linGrBrush.SetBlend(relativeIntensities, relativePositions, 3);  
  
graphics.FillEllipse(&linGrBrush, 0, 30, 200, 100);  
graphics.FillRectangle(&linGrBrush, 0, 155, 500, 30);
```

下面的插图显示所得到的椭圆和矩形。



- **对角线性渐变**

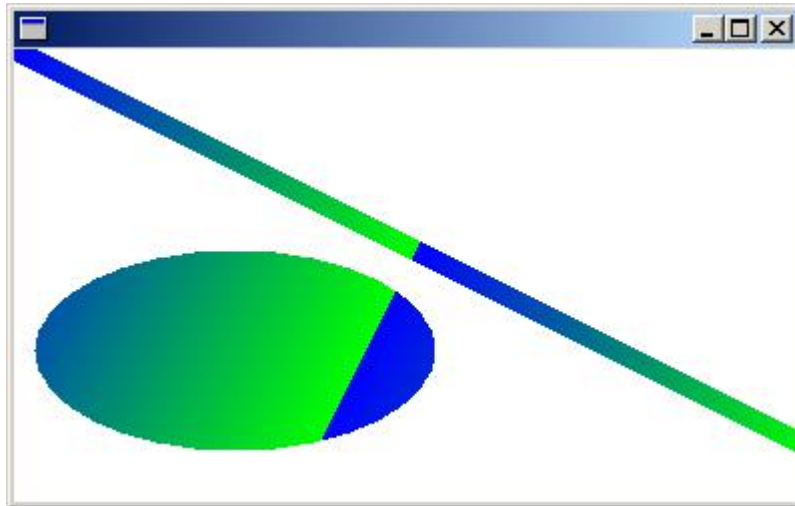
在上面的示例中，渐变的方向是水平的；即，当沿着水平线移动时，颜色逐渐变化。还可以定义垂直渐变和对角线渐变。下面的示例将点 (0, 0) 和点 (200, 100) 传递给 `LinearGradientBrush` 构造函数。蓝色与 (0, 0) 相关联，绿色与 (200, 100) 相关联。用线性渐变画笔填充线条（画笔的宽度为 10）和椭圆。

```
LinearGradientBrush linGrBrush(  
    Point(0, 0),  
    Point(200, 100),  
    Color(255, 0, 0, 255), // opaque blue  
    Color(255, 0, 255, 0)); // opaque green  
  
Pen pen(&linGrBrush, 10);
```



```
graphics.DrawLine(&pen, 0, 0, 600, 300);
graphics.FillEllipse(&linGrBrush, 10, 100, 200, 100);
```

下面的插图显示该线条和椭圆。请注意，当沿着任何与穿过 (0, 0) 和 (200, 100) 的直线平行的直线移动时，椭圆中的颜色逐渐变化。



### 创建路径渐变

**PathGradientBrush** 类使您可以自定义用渐变色来填充形状的方式。一个 **PathGradientBrush** 对象有一个轨迹边界和一个中心点。可为轨迹的中心指定一种颜色，为轨迹的边界指定另一种颜色。还可为轨迹边界上的多个点分别指定颜色。

#### 注意

在 GDI+ 中，路径是由 **GraphicsPath** 对象维护的一系列线条和曲线。有关 GDI+ 路径的更多信息，请参见 GDI+ 中的图形路径和构造并绘制轨迹。

下面的示例用路径渐变画笔填充椭圆。中心的颜色设置为蓝色；边界的颜色设置为浅绿色。

```
// Create a path that consists of a single ellipse.
GraphicsPath path;
path.AddEllipse(0, 0, 140, 70);

// Use the path to construct a brush.
PathGradientBrush pthGrBrush(&path);

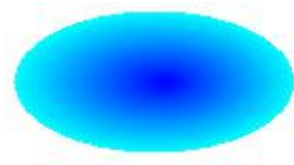
// Set the color at the center of the path to blue.
pthGrBrush.SetCenterColor(Color(255, 0, 0, 255));

// Set the color along the entire boundary of the path to aqua.
Color colors[] = {Color(255, 0, 255, 255)};
```

```
int count = 1;
pthGrBrush.SetSurroundColors(colors, &count);

graphics.FillEllipse(&pthGrBrush, 0, 0, 140, 70);
```

下面的插图显示已填充的椭圆。



默认情况下，路径渐变画刷不延伸到路径边界的外部。如果使用路径渐变画刷填充超出路径边界的形状，则路径外部的屏幕区域将不会被填充。下面的插图显示将上述代码中的 `FillEllipse` 调用更改为 `e.Graphics.FillRectangle(pthGrBrush, 0, 10, 200, 40)` 时发生的情况。



- 在边界上指定点

下面的示例由星形轨迹构造路径渐变画笔。该代码调用 `SetCenterColor` 方法，它将星形中心的颜色设置为红色。然后，该代码调用 `SetSurroundColors` 方法，以便在 `points` 数组中的各个点处指定不同的颜色（存储在 `colors` 数组中）。最后一个代码语句用路径渐变画笔填充星形轨迹。

```
// Put the points of a polygon in an array.
Point points[] = {Point(75, 0),    Point(100, 50),
                  Point(150, 50),  Point(112, 75),
                  Point(150, 150), Point(75, 100),
                  Point(0, 150),   Point(37, 75),
                  Point(0, 50),    Point(50, 50)};

// Use the array of points to construct a path.
GraphicsPath path;
path.AddLines(points, 10);

// Use the path to construct a path gradient brush.
PathGradientBrush pthGrBrush(&path);

// Set the color at the center of the path to red.
pthGrBrush.SetCenterColor(Color(255, 255, 0, 0));

// Set the colors of the points in the array.
Color colors[] = {Color(255, 0, 0, 0),    Color(255, 0, 255, 0),
```

```

        Color(255, 0, 0, 255), Color(255, 255, 255, 255),
        Color(255, 0, 0, 0),   Color(255, 0, 255, 0),
        Color(255, 0, 0, 255), Color(255, 255, 255, 255),
        Color(255, 0, 0, 0),   Color(255, 0, 255, 0));

int count = 10;
pthGrBrush.SetSurroundColors(colors, &count);

// Fill the path with the path gradient brush.
graphics.FillPath(&pthGrBrush, &path);

```

下面的插图显示填充后的星形。



下面的示例基于一个点数组构造路径渐变画刷。为数组中的五个点各分配一种颜色。如果用直线连接这五个点，就会得到一个五边形。还会为该多边形的中心（形心）分配一种颜色。在该示例中，中心 (80, 75) 设置为白色。该示例中最后一条语句用轨迹梯度刷填充矩形。

用于填充矩形的颜色在 (80, 75) 处是白色而且从 (80, 75) 移向数组中的各点时逐渐变化。例如，当您从 (80, 75) 移到 (0, 0) 时，颜色将逐渐从白色变成红色；当您从 (80, 75) 移到 (160, 0) 时，颜色逐渐从白色变成绿色。

```

// Construct a path gradient brush based on an array of points.
PointF ptsF[] = {PointF(0.0f, 0.0f),
                  PointF(160.0f, 0.0f),
                  PointF(160.0f, 200.0f),
                  PointF(80.0f, 150.0f),
                  PointF(0.0f, 200.0f)};

PathGradientBrush pBrush(ptsF, 5);

// An array of five points was used to construct the path gradient
// brush. Set the color of each point in that array.
Color colors[] = {Color(255, 255, 0, 0), // (0, 0) red
                  Color(255, 0, 255, 0), // (160, 0) green
                  Color(255, 0, 255, 0), // (160, 200) green
                  Color(255, 0, 0, 255), // (80, 150) blue

```

```

        Color(255, 255, 0, 0)); // (0, 200) red

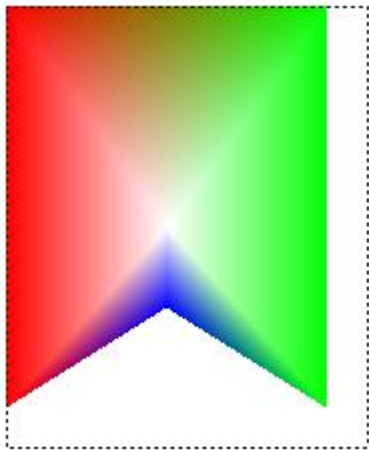
int count = 5;
pBrush.SetSurroundColors(colors, &count);

// Set the center color to white.
pBrush.SetCenterColor(Color(255, 255, 255, 255));

// Use the path gradient brush to fill a rectangle.
graphics.FillRectangle(&pBrush, Rect(0, 0, 180, 220));

```

请注意，在上面的代码中没有 **GraphicsPath** 对象。在该示例中，特定的 **PathGradientBrush** 构造函数接收一个点数组的指针，但是不需要 **GraphicsPath** 对象。同时，请注意，路径渐变画刷用于填充矩形而非填充路径。矩形比用于定义画笔的闭合路径大，因此矩形的某些部分未由画笔涂色。下面的插图显示该矩形（虚线）以及该矩形被路径渐变画刷涂色的那部分。



- 自定义路径渐变

自定义路径渐变画笔的一种方法就是设置它的聚焦缩放。聚焦缩放指定位于主轨迹内部的内部轨迹。中心颜色显示在内部轨迹中的任何地方，而不是只显示在中心点。可以调用 **SetFocusScales** 方法来设置一个路径渐变画刷的聚焦缩放。

下面的示例根据椭圆路径创建路径渐变画笔。该代码将边界颜色设置为蓝色，将中心颜色设置为浅绿色，然后使用路径渐变画笔填充椭圆路径。

接着，该代码设置路径渐变画笔的聚焦缩放。**x** 聚焦缩放被设置为 **0.3**，**y** 聚焦缩放被设置为 **0.8**。该代码调用 **Graphics** 对象的 **TranslateTransform** 方法，以便后通过对 **FillPath** 的调用填充位于第一个椭圆右侧的椭圆。

若要观看聚焦缩放的效果，请设想一个与主椭圆共用一个中心的小椭圆。小（内部）椭圆是由主椭圆在水平方向上缩小 **0.3** 倍，在垂直方向上缩小 **0.8** 倍（围绕其中心）得到的。当从外部椭圆的边界移到内部椭圆的边界时，颜色逐渐从蓝色变成浅绿色。当从内部椭圆的边界移到共用中心时，颜色保持浅绿色。

```

// Create a path that consists of a single ellipse.
GraphicsPath path;
path.AddEllipse(0, 0, 200, 100);

// Create a path gradient brush based on the elliptical path.
PathGradientBrush pthGrBrush(&path);
pthGrBrush.SetGammaCorrection(TRUE);

// Set the color along the entire boundary to blue.
Color color(Color(255, 0, 0, 255));
INT num = 1;
pthGrBrush.SetSurroundColors(&color, &num);

// Set the center color to aqua.
pthGrBrush.SetCenterColor(Color(255, 0, 255, 255));

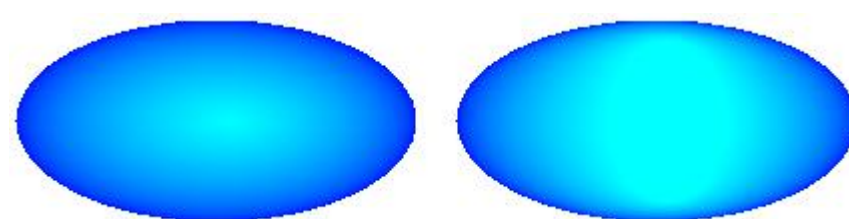
// Use the path gradient brush to fill the ellipse.
graphics.FillPath(&pthGrBrush, &path);

// Set the focus scales for the path gradient brush.
pthGrBrush.SetFocusScales(0.3f, 0.8f);

// Use the path gradient brush to fill the ellipse again.
// Show this filled ellipse to the right of the first filled ellipse.
graphics.TranslateTransform(220.0f, 0.0f);
graphics.FillPath(&pthGrBrush, &path);

```

下面的插图显示下面代码的输出。左边的椭圆只在中心点上为浅绿色。右边的椭圆在内部轨迹内部的任何地方都为浅绿色。



自定义路径渐变画笔的另一种方法是指定插值颜色数组和插值位置数组。

下面的示例基于三角形创建路径渐变画笔。该代码调用路径渐变画笔的 `SetInterpolationColors` 方法，以便指定插值颜色数组（深绿色，浅绿色，蓝色）和插值位置数组 `(0, 0.25, 1)`。当从三角形的边界移到中心点时，颜色将从深绿色逐渐变成浅绿色，然后从浅绿色变成蓝色。深绿色到浅绿色的转变发生在深绿色到蓝色转变的距离的 25% 处。

```

// Vertices of the triangle

```

```

Point points[] = {Point(100, 0),
                  Point(200, 200),
                  Point(0, 200)};

// No GraphicsPath object is created. The PathGradient
// brush is constructed directly from the array of points.
PathGradientBrush pthGrBrush(points, 3);

Color presetColors[] = {
    Color(255, 0, 128, 0),    // Dark green
    Color(255, 0, 255, 255), // Aqua
    Color(255, 0, 0, 255)};  // Blue

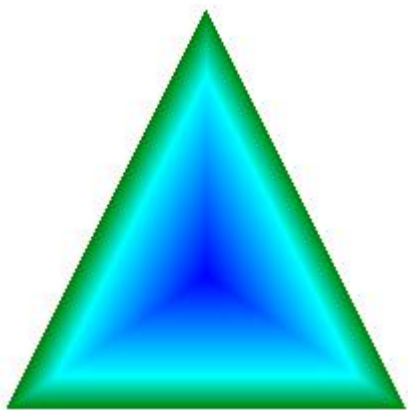
REAL interpPositions[] = {
    0.0f, // Dark green is at the boundary of the triangle.
    0.25f, // Aqua is 25 percent of the way from the boundary
           // to the center point.
    1.0f}; // Blue is at the center point.

pthGrBrush.SetInterpolationColors(presetColors, interpPositions, 3);

// Fill a rectangle that is larger than the triangle
// specified in the Point array. The portion of the
// rectangle outside the triangle will not be painted.
graphics.FillRectangle(&pthGrBrush, 0, 0, 200, 200);

```

下面的插图显示用自定义路径渐变画笔填充的三角形。



- 设置中心点

在默认情况下，路径渐变画笔的中心点位于用来构造梯度刷的轨迹的形心。可通过设置 `PathGradientBrush` 类的 `SetCenterPoint` 方法更改中心点的位置。

下面的示例基于椭圆来创建路径渐变画笔。椭圆的中心位于 (70, 35)，但是路径渐变画笔的中心点设置在 (120, 40)。

```
// Create a path that consists of a single ellipse.
GraphicsPath path;
path.AddEllipse(0, 0, 140, 70);

// Use the path to construct a brush.
PathGradientBrush pthGrBrush(&path);

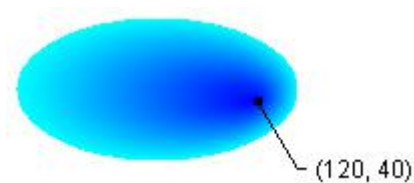
// Set the center point to a location that is not the centroid of the path.
pthGrBrush.SetCenterPoint(Point(120, 40));

// Set the color at the center point to blue.
pthGrBrush.SetCenterColor(Color(255, 0, 0, 255));

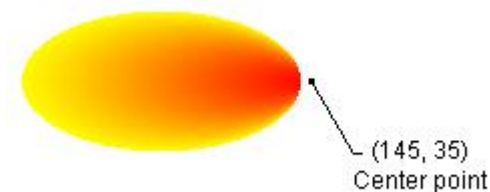
// Set the color along the entire boundary of the path to aqua.
Color colors[] = {Color(255, 0, 255, 255)};
int count = 1;
pthGrBrush.SetSurroundColors(colors, &count);

graphics.FillEllipse(&pthGrBrush, 0, 0, 140, 70);
```

下面的插图显示实心椭圆和路径渐变画笔的中心点。



可将路径渐变画笔的中心点设置在用于构造梯度刷的轨迹外部的某个位置。在上面的代码中，如果你将 `SetCenterPoint` 调用改为 `pthGrBrush.SetCenterPoint(Point(145, 35))`，那么将得到如下的图像：



在上面的插图中，椭圆最右边的那些点不是纯红色（尽管它们非常接近）。渐变中颜色的定位就好像是填充到颜色为纯蓝色 (0, 0, 255) 的点 (145, 35)。但是，由于路径渐变画笔只在其轨迹内部涂色，所以并未填充到点 (145, 35)。

## 将 Gamma 校正应用于渐变

可通过调用渐变画笔的 `SetGammaCorrection` 方法传入 `true` 来启用线性渐变画笔的灰度校正。可通过将 `GammaCorrection` 属性设置为 `false` 来禁用灰度校正。在默认情况下，禁用灰度校正。

下面的示例创建一个线性渐变画笔并使用它填充两个矩形。第一个矩形在填充时未启用灰度校正；第二个矩形在填充时启用了灰度校正。

```
LinearGradientBrush linGrBrush(  
    Point(0, 10),  
    Point(200, 10),  
    Color(255, 255, 0, 0),    // Opaque red  
    Color(255, 0, 0, 255));  // Opaque blue  
  
graphics.FillRectangle(&linGrBrush, 0, 0, 200, 50);  
linGrBrush.SetGammaCorrection(TRUE);  
graphics.FillRectangle(&linGrBrush, 0, 60, 200, 50);
```

下面的插图显示这两个实心矩形。上面的矩形未采用灰度校正，它的中间部分看上去较暗。下面的矩形采用了灰度校正，看上去亮度更均匀。



下面的例子构建了一个基于五角星路径的路径渐变画刷。代码采用关闭灰度校正（默认）的路径渐变画刷来填充路径。然后代码通过调用 `SetGammaCorrection` 方法并传入 `TRUE` 来开启路径渐变画刷的灰度校正。调用 `Graphics::TranslateTransform` 设置 `Graphics` 对象的世界变换，以便后面的 `FillPath` 调用将填充的五角星放置到开头那个五角星的右边。

```
// Put the points of a polygon in an array.  
Point points[] = {Point(75, 0), Point(100, 50),  
                  Point(150, 50), Point(112, 75),  
                  Point(150, 150), Point(75, 100),  
                  Point(0, 150), Point(37, 75),  
                  Point(0, 50), Point(50, 50)};  
  
// Use the array of points to construct a path.  
GraphicsPath path;  
path.AddLines(points, 10);  
  
// Use the path to construct a path gradient brush.
```



```

PathGradientBrush pthGrBrush(&path);

// Set the color at the center of the path to red.
pthGrBrush.SetCenterColor(Color(255, 255, 0, 0));

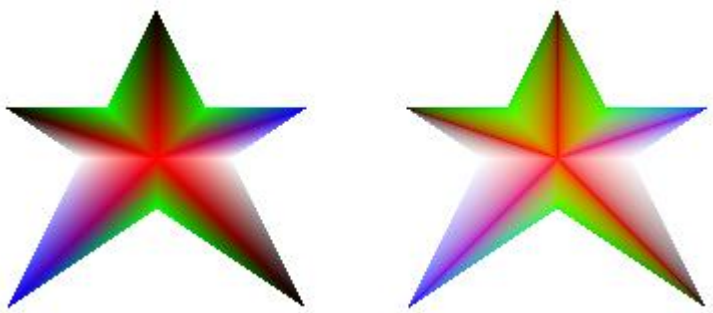
// Set the colors of the points in the array.
Color colors[] = {Color(255, 0, 0, 0),   Color(255, 0, 255, 0),
                  Color(255, 0, 0, 255), Color(255, 255, 255, 255),
                  Color(255, 0, 0, 0),   Color(255, 0, 255, 0),
                  Color(255, 0, 0, 255), Color(255, 255, 255, 255),
                  Color(255, 0, 0, 0),   Color(255, 0, 255, 0)};

int count = 10;
pthGrBrush.SetSurroundColors(colors, &count);

// Fill the path with the path gradient brush.
graphics.FillPath(&pthGrBrush, &path);
pthGrBrush.SetGammaCorrection(TRUE);
graphics.TranslateTransform(200.0f, 0.0f);
graphics.FillPath(&pthGrBrush, &path);

```

下面的插图显示了上面代码的输出结果。右边的五角星使用了灰度校正。请注意左边的五角星，因为没有启用灰度校正看起来较暗。



### 构造并绘制路径

路径由一系列可作为单个单元来操作和绘制的图形基元（直线、矩形、曲线、文本等）组成。路径可划分为开放式或闭合式“图形”。一个图形中可包含几个基元。

可通过调用 **Graphics** 类的 **DrawPath** 方法绘制轨迹；调用 **Graphics** 类的 **FillPath** 方法填充轨迹。

### 使用线条、曲线和形状创建图形

若要创建轨迹，请构造 **GraphicsPath** 对象，然后调用诸如 **AddLine** 和 **AddCurve** 之类的方法以便将基元添加到轨迹中。

下面的示例创建包含单一图形的轨迹。该图形由一段弧组成。该弧的扫描角为-180 度，该扫描角在默认的坐标系中是沿逆时针方向旋转的。

```
Pen pen(Color(255, 255, 0, 0));
GraphicsPath path;
path.AddArc(175, 50, 50, 50, 0, -180);
graphics.DrawPath(&pen, &path);
```

下面的示例创建包含两个图形的轨迹。第一个图形是一段弧，后跟一条直线。第二个图形是一条直线，后跟一条曲线，再后面又是一条直线。第一个图形的左边是敞开的；第二个图形是闭合的。

```
Point points[] = {Point(40, 60), Point(50, 70), Point(30, 90)};

Pen pen(Color(255, 255, 0, 0), 2);
GraphicsPath path;

// The first figure is started automatically, so there is
// no need to call StartFigure here.
path.AddArc(175, 50, 50, 50, 0.0f, -180.0f);
path.AddLine(100, 0, 250, 20);

path.StartFigure();
path.AddLine(50, 20, 5, 90);
path.AddCurve(points, 3);
path.AddLine(50, 150, 150, 180);
path.CloseFigure();

graphics.DrawPath(&pen, &path);
```

除了加入线条和曲线到路径中以外，你还可以加入闭合形状：矩形、椭圆、饼图和多边形。下面的例子将创建一个由 2 条线段、1 个矩形、1 个椭圆所组成的路径。代码使用了 **Pen** 来绘制路径，**Brush** 来填充路径。

```
GraphicsPath path;
Pen pen(Color(255, 255, 0, 0), 2);
SolidBrush brush(Color(255, 0, 0, 200));

path.AddLine(10, 10, 100, 40);
path.AddLine(100, 60, 30, 60);
path.AddRectangle(Rect(50, 35, 20, 40));
path.AddEllipse(10, 75, 40, 30);

graphics.DrawPath(&pen, &path);
graphics.FillPath(&brush, &path);
```

上面例子中的路径有 3 个图形。第一个图形包含 2 条线，第二个图形包含一个矩形，而第三个图形包含一个椭圆。虽然并没有调用 `CloseFigure` 和 `StartFigure`，但是如矩形和椭圆等本质上为闭合的图形将被视为独立图形。

### 填充开放式图形

可通过将 `GraphicsPath` 对象地址传递给 `Graphics::FillPath` 方法来填充路径。`FillPath` 方法根据当前为路径设置的填充模式（交替或缠绕）填充路径。如果轨迹包含任何开放图形，也会像这些图形是闭合图形一样来填充轨迹。`GDI+` 通过绘制一条从图形终点到起点的直线来闭合图形。

下面的示例创建包含一个开放式图形（弧）和一个闭合式图形（椭圆）的轨迹。**`Graphics::FillPath`** 方法按照默认的填充模式（即 `FillModeAlternate`）填充轨迹。

```
GraphicsPath path;

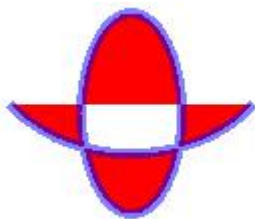
// Add an open figure.
path.AddArc(0, 0, 150, 120, 30, 120);

// Add an intrinsically closed figure.
path.AddEllipse(50, 50, 50, 100);

Pen pen(Color(128, 0, 0, 255), 5);
SolidBrush brush(Color(255, 255, 0, 0));

// The fill mode is FillModeAlternate by default.
graphics.FillPath(&brush, &path);
graphics.DrawPath(&pen, &path);
```

下面的插图显示示例代码的输出。请注意，如同用一条从开放式图形终点到起点的直线闭合了开放式图形一样填充了轨迹（`FillModeAlternate`）。



### 使用图形容器

`Graphics` 对象提供了诸如 `DrawLine`、`DrawImage` 和 `DrawString` 之类的显示矢量图像、光栅图像和文本的方法。`Graphics` 对象还具有几个影响所绘制项的质量和方向的属性。例如，平滑模式属性确定是否对直线和曲线应用 `Antialias`，世界变换属性影响所绘制项目的位置和旋转。

Graphics 对象常常与特定的显示设备相关联。使用 Graphics 对象在窗口中绘图时，Graphics 对象还与该特定的窗口相关联。

Graphics 对象可被视为容器，因为它包含一组影响绘图的属性并且与设备特定的信息相链接。通过调用该 Graphics 对象的 BeginContainer 方法可以在现有的 Graphics 对象中创建一个辅助容器。

### 管理 Graphics 对象的状态

Graphics 类为 GDI+ 的核心。若要进行绘制，请创建 Graphics 对象，设置其属性，并调用其方法（DrawLine、DrawImage 和 DrawString 等）。

下面的示例调用 Graphics 对象的 DrawRectangle 方法。传递给 DrawRectangle 方法的第一个参数是 Pen 对象。

```
HDC          hdc;
PAINTSTRUCT  ps;

hdc = BeginPaint(hWnd, &ps);
{
    Graphics graphics(hdc);
    Pen pen(Color(255, 0, 0, 255)); // opaque blue
    graphics.DrawRectangle(&pen, 10, 10, 200, 100);
}
EndPaint(hWnd, &ps);
```

上面的例子中，BeginPaint 方法返回一个设备场景句柄，然后这个句柄被传给 Graphics 构造函数。一个设备场景是一个包含当前的特定显示设备信息的结构体（由窗体维护）。

### Graphics 状态

Graphics 对象不仅仅提供绘制方法，如 DrawLine 和 DrawRectangle。Graphics 对象还维护图形状态，图形状态可划分为以下几类：

-  设备场景
-  质量设置
-  变换
-  剪辑区域

- 设备场景

作为一个开发人员，您不必考虑一个 Graphics 对象和其设备场景之间的交互。这个交互由 GDI+ 在幕后进行处理。

- 质量设置

**Graphics** 对象具有几个影响屏幕绘制质量的属性。您可以通过 **Get** 和 **Set** 方法来查看和操作这些属性。例如，可调用 **SetTextRenderingHint** 方法以指定应用于文本的消除锯齿的类型（如果有的话）。影响质量的其他方法是 **SetSmoothingMode**、**SetCompositingMode**、**SetCompositingQuality** 和 **SetInterpolationMode**。

下面的示例绘制了两个椭圆，一个的平滑模式设置为 **SmoothingModeAntiAlias**，另一个的平滑模式设置为 **SmoothingModeHighSpeed**：

```
Graphics graphics(hdc);
Pen pen(Color(255, 0, 255, 0)); // opaque green

graphics.SetSmoothingMode(SmoothingModeAntiAlias);
graphics.DrawEllipse(&pen, 0, 0, 200, 100);
graphics.SetSmoothingMode(SmoothingModeHighSpeed);
graphics.DrawEllipse(&pen, 0, 150, 200, 100);
```

- **变换**

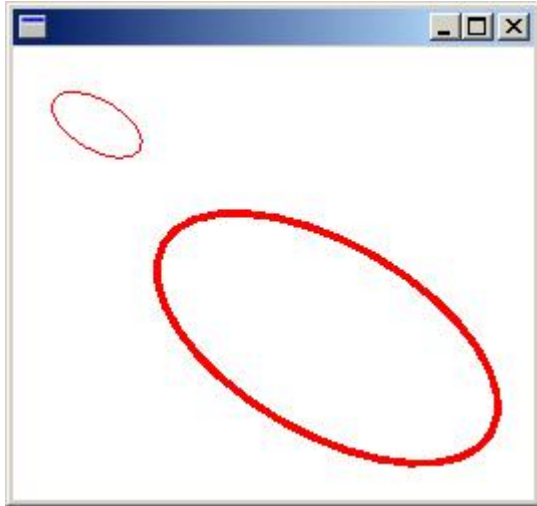
**Graphics** 对象包含两种应用于 **Graphics** 对象绘制的所有项目的变换（世界变换和页面变换）。任何仿射变换都可存储在世界变换中。仿射变换包括缩放、旋转、反射、扭曲和平移。页面变换可用于缩放和更改单位（例如，像素到英寸）。有关更多信息，请参见[坐标系统和变换](#)。

下面的示例设置 **Graphics** 对象的世界变换和页面变换。世界变换被设置为旋转 30 度。设置页面变换，使传递给第二个 **DrawEllipse** 的坐标按毫米计，而不是像素。该代码对 **DrawEllipse** 方法进行两次相同的调用。世界变换应用于第一个 **DrawEllipse** 调用，两种变换（世界变换和页变换）都应用于第二个 **DrawEllipse** 调用。

```
Graphics graphics(hdc);
Pen pen(Color(255, 255, 0, 0));

graphics.ResetTransform();
graphics.RotateTransform(30.0f); // World transformation
graphics.DrawEllipse(&pen, 30, 0, 50, 25);
graphics.SetPageUnit(UnitMillimeter); // Page transformation
graphics.DrawEllipse(&pen, 30, 0, 50, 25);
```

下面的插图显示这两个椭圆。请注意，30 度的旋转是针对坐标系的原点（工作区的左上角）的，而不是椭圆的中心。还要注意，画笔的宽度 1 对于第一个椭圆来说表示 1 个像素，对于第二个椭圆来说表示 1 毫米。



- **剪辑区域**

**Graphics** 对象包含应用于 **Graphics** 对象绘制的所有项目的剪辑区域。可通过调用 **SetClip** 方法设置剪辑区域。

下面的示例通过合并两个矩形来创建形状为加号的区域。该区域被指定为 **Graphics** 对象的剪辑区域。然后，该代码绘制两条限制在剪辑区域内部的直线。

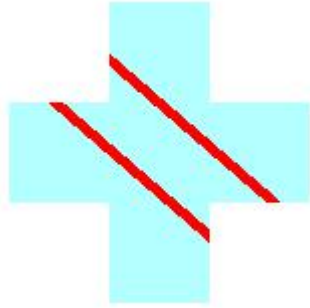
```
Graphics graphics(hdc);
Pen pen(Color(255, 255, 0, 0), 5); // opaque red, width 5
SolidBrush brush(Color(255, 180, 255, 255)); // opaque aqua

// Create a plus-shaped region by forming the union of two rectangles.
Region region(Rect(50, 0, 50, 150));
region.Union(Rect(0, 50, 150, 50));
graphics.FillRegion(&brush, &region);

// Set the clipping region.
graphics.SetClip(&region);

// Draw two clipped lines.
graphics.DrawLine(&pen, 0, 30, 150, 160);
graphics.DrawLine(&pen, 40, 20, 190, 150);
```

下面的插图显示这两条剪辑过的直线。



## 使用嵌套的 **Graphics** 容器

GDI+ 提供可用于在 **Graphics** 对象中临时替换或增加状态部件的容器。通过调用 **Graphics** 对象的 **BeginContainer** 方法创建容器。可重复调用 **BeginContainer** 以形成嵌套容器。每调用一次 **BeginContainer**，都要调用一次 **EndContainer**。

- 嵌套容器中的变换

下面的示例创建一个 **Graphics** 对象并在该 **Graphics** 对象中创建一个容器。**Graphics** 对象的世界变换是在 **x** 方向平移 100 个单位，在 **y** 方向平移 80 个单位。该容器的世界变换是旋转 30 度。该代码调用两次 **DrawRectangle(pen, -60, -30, 120, 60)**。对 **DrawRectangle** 的第一次调用是在容器的内部，也就是说，此调用是在调用 **BeginContainer** 和 **EndContainer** 之间发生的。对 **DrawRectangle** 的第二次调用是在调用 **EndContainer** 之后。

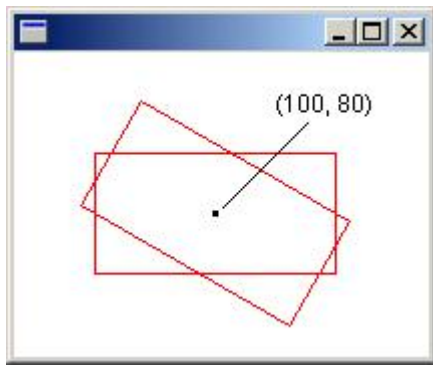
```
Graphics          graphics(hdc);
Pen               pen(Color(255, 255, 0, 0));
GraphicsContainer graphicsContainer;

graphics.TranslateTransform(100.0f, 80.0f);

graphicsContainer = graphics.BeginContainer();
    graphics.RotateTransform(30.0f);
    graphics.DrawRectangle(&pen, -60, -30, 120, 60);
graphics.EndContainer(graphicsContainer);

graphics.DrawRectangle(&pen, -60, -30, 120, 60);
```

在上面的代码中，从容器内部绘制的矩形依次经过容器的世界变换（旋转）和 **Graphics** 对象的世界变换（平移）。从容器外部绘制的矩形只经过 **Graphics** 对象的世界变换（平移）。下面的插图显示这两个矩形。



- 在嵌套容器中剪辑

下面的示例演示嵌套容器如何处理剪辑区域。该代码创建一个 **Graphics** 对象并在该 **Graphics** 对象中创建一个容器。**Graphics** 对象的剪辑区域是矩形，容器的剪辑区域是椭圆。该代码调用两次 **DrawLine** 方法。第一次调用 **DrawLine** 是在容器内部，第二次调用 **DrawLine** 是在容器的外部（在调用 **EndContainer** 之后）。第一条直线被两个剪辑区域的相交部分剪辑。第二条直线只被 **Graphics** 对象的矩形剪辑区域剪辑。

```
Graphics          graphics(hdc);
GraphicsContainer  graphicsContainer;
Pen               redPen(Color(255, 255, 0, 0), 2);
Pen               bluePen(Color(255, 0, 0, 255), 2);
SolidBrush        aquaBrush(Color(255, 180, 255, 255));
SolidBrush        greenBrush(Color(255, 150, 250, 130));

graphics.SetClip(Rect(50, 65, 150, 120));
graphics.FillRectangle(&aquaBrush, 50, 65, 150, 120);

graphicsContainer = graphics.BeginContainer();
    // Create a path that consists of a single ellipse.
    GraphicsPath path;
    path.AddEllipse(75, 50, 100, 150);

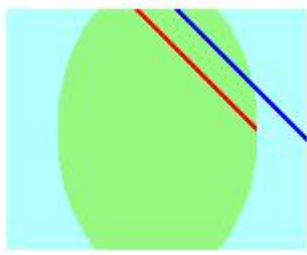
    // Construct a region based on the path.
    Region region(&path);
    graphics.FillRegion(&greenBrush, &region);

    graphics.SetClip(&region);
    graphics.DrawLine(&redPen, 50, 0, 350, 300);
graphics.EndContainer(graphicsContainer);

graphics.DrawLine(&bluePen, 70, 0, 370, 300);
```

下面的插图显示这两条剪辑过的直线。





正如上面的示例所示，变换和剪辑区域在嵌套容器中是累积的。如果设置容器和 **Graphics** 对象的世界变换，则这两种变换都将应用于从容器内部绘制的项目。首先应用容器的变换，再应用 **Graphics** 对象的变换。如果设置容器和 **Graphics** 对象的剪辑区域，则从容器内部绘制的项目将被两个剪辑区域的相交部分剪辑。

- **嵌套容器中的品质设置**

嵌套容器中的品质设置（**SmoothingMode** 和 **TextRenderingHint** 等）是不累积的；相反，容器的品质设置会临时替换 **Graphics** 对象的品质设置。在创建新容器时，该容器的品质设置将被设置为默认值。例如，假设有一个 **Graphics** 对象，其平滑模式为 **AntiAlias**。在创建容器时，容器内部的平滑模式是默认的平滑模式。您可随意设置该容器的平滑模式，而在容器内部绘制的任何项目将按照您所设置的模式绘制。在调用 **EndContainer** 之后绘制的项目，将按照在调用 **BeginContainer** 之前就已设置好的 (**AntiAlias**) 平滑模式绘制。

- **多层嵌套容器**

在 **Graphics** 对象中，并不限于只有一个容器。您可创建一系列容器，每个容器都嵌套在前一个容器中；您可为每个嵌套容器指定世界变换、剪辑区域和品质设置。如果从最里边的容器内部调用绘图方法，则变换将按顺序应用，即，始于最里边的容器，终于最外边的容器。从最里边的容器内部绘制的项目将被所有剪辑区域的相交部分剪辑。

下面的示例创建 **Graphics** 对象并将其文本呈现提示设置为 **AntiAlias**。该代码创建两个容器，一个嵌套在另一个中。外部容器和内部容器的文本呈现提示分别设置为 **TextRenderingHintSingleBitPerPixel** 和 **TextRenderingHintAntiAlias**。该代码绘制三个字符串：一个从内部容器绘制，另一个从外部容器绘制，第三个从 **Graphics** 对象本身绘制。

```
Graphics graphics(hdc);
GraphicsContainer innerContainer;
GraphicsContainer outerContainer;
SolidBrush brush(Color(255, 0, 0, 255));
FontFamily fontFamily(L"Times New Roman");
Font font(&fontFamily, 36, FontStyleRegular, UnitPixel);

graphics.SetTextRenderingHint(TextRenderingHintAntiAlias);

outerContainer = graphics.BeginContainer();
```

```

graphics.SetTextRenderingHint(TextRenderingHintSingleBitPerPixel);

innerContainer = graphics.BeginContainer();
    graphics.SetTextRenderingHint(TextRenderingHintAntiAlias);
    graphics.DrawString(L"Inner Container", 15, &font,
        PointF(20, 10), &brush);
graphics.EndContainer(innerContainer);

    graphics.DrawString(L"Outer Container", 15, &font, PointF(20, 50), &brush);

graphics.EndContainer(outerContainer);

graphics.DrawString(L"Graphics Object", 15, &font, PointF(20, 90), &brush);

```

下面的插图显示这三个字符串。从内部容器和 **Graphics** 对象绘制的字符串都由“消除锯齿”平滑。从外部容器绘制的字符串不由“消除锯齿”平滑，这是由于 **TextRenderingHint** 属性被设置为 **TextRenderingHintSingleBitPerPixel**。

Inner Container  
Outer Container  
Graphics Object

## 变换

仿射变换包括旋转、缩放、反射、剪切和平移。在 **GDI+** 中，**Matrix** 类提供了用于在矢量图形、图像和文本上执行仿射变换的基础。

### 使用世界变换

世界变换是 **Graphics** 类的一个属性。指定世界变换的数字存储在表示一个  $3 \times 3$  矩阵的 **Matrix** 对象中。**Matrix** 和 **Graphics** 类有几种方法用于设置世界变换矩阵中的数字。本节中将以矩形为例进行说明，因为矩形容易绘制也容易看出变换效果。

在下面的示例中，该代码先创建  $50 \times 50$  的矩形，然后将其定位到原点 (0, 0)。原点位于工作区的左上角。

```

Rect rect(0, 0, 50, 50);
Pen pen(Color(255, 255, 0, 0), 0);
graphics.DrawRectangle(&pen, rect);

```

下面的代码应用缩放变换，将矩形的 **x** 方向放大到 1.75 倍；将 **y** 方向缩小到 0.5 倍：

```
Rect rect(0, 0, 50, 50);
Pen pen(Color(255, 255, 0, 0), 0);
graphics.ScaleTransform(1.75f, 0.5f);
graphics.DrawRectangle(&pen, rect);
```

其结果是一个比原矩形在 **x** 方向上变长、在 **y** 方向上变短的矩形。

若要旋转而非缩放矩形，请使用下面的代码：

```
Rect rect(0, 0, 50, 50);
Pen pen(Color(255, 255, 0, 0), 0);
graphics.RotateTransform(28.0f);
graphics.DrawRectangle(&pen, rect);
```

若要平移矩形，请使用下面的代码：

```
Rect rect(0, 0, 50, 50);
Pen pen(Color(255, 255, 0, 0), 0);
graphics.TranslateTransform(150.0f, 150.0f);
graphics.DrawRectangle(&pen, rect);
```

### 为什么变换顺序非常重要

单个 **Matrix** 对象可存储一个变换或一系列变换。后者称为“复合变换”。复合变换的矩阵是通过单个变换的矩阵相乘得到的。

在复合变换中，单个变换的顺序非常重要。例如，依次旋转、缩放和平移与依次平移、旋转和缩放得到的结果不同。在 **GDI+** 中，复合变换是从左到右构造的。如果用 **S**、**R** 和 **T** 分别代表缩放、旋转和平移矩阵，则乘积 **SRT**（按照这个顺序）就是依次缩放、旋转和平移获得的复合变换的矩阵。由乘积 **SRT** 生成的矩阵与由乘积 **TRS** 生成的矩阵不同。

造成顺序很重要的一个原因就是，像旋转和缩放这样的变换是针对坐标系的原点进行的。缩放以原点为中心的对象与缩放已离开原点的对象所得到的结果不同。同样，旋转以原点为中心的对象与旋转已离开原点的对象所得到的结果也不同。

下面的示例结合缩放、旋转和平移（按照这个顺序）以形成复合变换。传递给 **RotateTransform** 方法的 **MatrixOrderAppend** 参数指示将在缩放之后进行旋转。同样地，传递给 **TranslateTransform** 方法的参数 **MatrixOrderAppend** 指示将在旋转之后进行平移。

```
Rect rect(0, 0, 50, 50);
Pen pen(Color(255, 255, 0, 0), 0);
graphics.ScaleTransform(1.75f, 0.5f);
graphics.RotateTransform(28.0f, MatrixOrderAppend);
graphics.TranslateTransform(150.0f, 150.0f, MatrixOrderAppend);
graphics.DrawRectangle(&pen, rect);
```

下面的示例与上面的示例调用相同的方法，但是调用的顺序完全相反。作为其结果的操作顺序依次是平移、旋转和缩放，与依次缩放、旋转和平移所产生的结果大不相同。

```
Rect rect(0, 0, 50, 50);
Pen pen(Color(255, 255, 0, 0), 0);
graphics.TranslateTransform(150.0f, 150.0f);
graphics.RotateTransform(28.0f, MatrixOrderAppend);
graphics.ScaleTransform(1.75f, 0.5f, MatrixOrderAppend);
graphics.DrawRectangle(&pen, rect);
```

在复合变换中，颠倒单个变换顺序的一种方法是颠倒方法调用序列的顺序。控制操作顺序的第二种方法是更改矩阵顺序的参数。下面的示例与上面的示例相同，不同的是 **MatrixOrderAppend** 已更改为 **MatrixOrderPrepend**。矩阵是按照 **SRT** 顺序进行相乘的，其中 **S**、**R** 和 **T** 分别表示缩放、旋转和平移的矩阵。复合变换的顺序依次是缩放、旋转和平移。

```
Rect rect(0, 0, 50, 50);
Pen pen(Color(255, 255, 0, 0), 0);
graphics.TranslateTransform(150.0f, 150.0f, MatrixOrderPrepend);
graphics.RotateTransform(28.0f, MatrixOrderPrepend);
graphics.ScaleTransform(1.75f, 0.5f, MatrixOrderPrepend);
graphics.DrawRectangle(&pen, rect);
```

刚才的示例与本主题中的第一个示例所产生的结果完全相同。这是因为我们同时颠倒了方法调用的顺序和矩阵相乘的顺序。

## 使用区域

**GDI+ Region** 类用于定义自定义形状。该形状可由直线、多边形和曲线组成。

区域的两个常见用途是点击检测和剪辑。点击检测确定鼠标是否在屏幕的某个区域单击。剪辑是将绘图限制在某个区域。

## 对区域使用点击检测

点击检测的目的是确定光标是否位于给定对象（如图标或按钮）上。下面的示例通过合并两个矩形区域创建形状为加号的区域。假设变量 **point** 中保存最近单击过的位置。该代码查看 **point** 是否位于形状为加号的区域。如果该点位于该区域（命中），则该区域将用不透明的红色画笔填充。否则，该区域将用半透明的红色画笔填充。

```
Point point(60, 10);
// Assume that the variable "point" contains the location
// of the most recent click.
// To simulate a hit, assign (60, 10) to point.
// To simulate a miss, assign (0, 0) to point.
SolidBrush solidBrush(Color());
```

```

Region region1(Rect(50, 0, 50, 150));
Region region2(Rect(0, 50, 150, 50));
// Create a plus-shaped region by forming the union of region1 and region2.
// The union replaces region1.
region1.Union(&region2);
if(region1.IsVisible(point, &graphics))
{
    // The point is in the region. Use an opaque brush.
    solidBrush.SetColor(Color(255, 255, 0, 0));
}
else
{
    // The point is not in the region. Use a semitransparent brush.
    solidBrush.SetColor(Color(64, 255, 0, 0));
}
graphics.FillRegion(&solidBrush, &region1);

```

### 对区域使用剪辑

**Graphics** 类的一个属性是剪辑区域。所有由给定的 **Graphics** 对象进行的绘制都限制在 **Graphics** 对象的剪辑区域中。可通过调用 **SetClip** 方法设置剪辑区域。

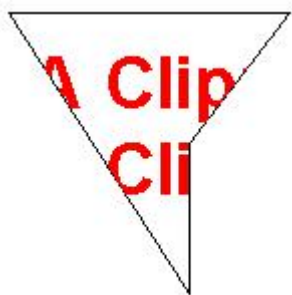
下面的示例构造由一个多边形组成的路径。然后，该代码根据该路径构造一个区域。将该区域传递给 **Graphics** 对象的 **SetClip** 方法，然后绘制两个字符串。

```

// Create a path that consists of a single polygon.
Point polyPoints[] = {Point(10, 10), Point(150, 10),
    Point(100, 75), Point(100, 150)};
GraphicsPath path;
path.AddPolygon(polyPoints, 4);
// Construct a region based on the path.
Region region(&path);
// Draw the outline of the region.
Pen pen(Color(255, 0, 0, 0));
graphics.DrawPath(&pen, &path);
// Set the clipping region of the Graphics object.
graphics.SetClip(&region);
// Draw some clipped strings.
FontFamily fontFamily(L"Arial");
Font font(&fontFamily, 36, FontStyleBold, UnitPixel);
SolidBrush solidBrush(Color(255, 255, 0, 0));
graphics.DrawString(L"A Clipping Region", 20, &font,
    PointF(15, 25), &solidBrush);
graphics.DrawString(L"A Clipping Region", 20, &font,
    PointF(15, 68), &solidBrush);

```

下面的插图显示这个经剪辑的字符串。



## 对图像重新着色

重新着色是调整图像颜色的过程。重新着色包括：将一种颜色更改为另一种颜色，调整某种颜色相对于另一种颜色的亮度，调整所有颜色的亮度或对比度，以及增加颜色的灰度。

## 使用颜色矩阵对单色进行变换

GDI+ 提供了用于存储和操作图像的 **Image** 和 **Bitmap** 类。**Image** 和 **Bitmap** 对象按照 32 位数存储每个像素的颜色：红、绿、蓝和 Alpha 各 8 位。这四个分量的值都是 0 到 255，其中 0 表示没有亮度，255 表示最大亮度。alpha 分量指定颜色的透明度：0 表示完全透明，255 表示完全不透明。

颜色矢量采用 4 元组形式（红色、绿色、蓝色、alpha）。例如，颜色矢量 (0, 255, 0, 255) 表示一种没有红色和蓝色但绿色达到最大亮度的不透明颜色。

表示颜色的另一种惯例是用数字 1 表示亮度达到最大。使用这种约定，上一段中描述的颜色将可以由矢量 (0, 1, 0, 1) 表示。在执行颜色转换时，GDI+ 约定 1 作为最大亮度。

可通过用  $4 \times 4$  矩阵乘以这些颜色矢量将线性变换（旋转和缩放等）应用到颜色矢量中。但是，您不能使用  $4 \times 4$  矩阵进行平移（非线性）。如果在每个颜色矢量中再添加一个虚拟的第 5 坐标（例如，数字 1），则可使用  $5 \times 5$  矩阵应用任何组合形式的线性变换和平移。由线性变换组成的后跟平移的变换称为仿射变换。

例如，假设您希望从颜色 (0.2, 0.0, 0.4, 1.0) 开始并应用下面的变换：

1. 将红色分量乘以 2。
2. 将红色、绿色和蓝色分量各加 0.2。

下面的矩阵乘法将按照列出的顺序进行这对变换。

$$\begin{bmatrix} 0.2 & 0.0 & 0.4 & 1.0 & 1.0 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0.2 & 0.2 & 0.2 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0.6 & 0.2 & 0.6 & 1.0 & 1.0 \end{bmatrix}$$

颜色矩阵的元素按照先行后列（从 0 开始）的顺序进行索引。例如，矩阵 **M** 的第五行第三列由 **M[4][2]** 表示。

5×5 单位矩阵（在下面的插图中显示）在对角线上为 1，在其他任何地方为 0。如果用单位矩阵乘以颜色矢量，则颜色矢量不会发生改变。形成颜色变换矩阵的一种简便方法是从单位矩阵开始，然后进行较小的改动以产生所需的变换。

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Identity Matrix

有关矩阵和变换的更详细的讨论，请参见[坐标系统和变形](#)。

下面的示例采用一个使用一种颜色 (0.2, 0.0, 0.4, 1.0) 的图像，并应用上一段中描述的变换。

```
Image          image(L"InputColor.bmp");
ImageAttributes imageAttributes;
UINT           width  = image.GetWidth();
UINT           height = image.GetHeight();

ColorMatrix colorMatrix = {
    2.0f, 0.0f, 0.0f, 0.0f, 0.0f,
    0.0f, 1.0f, 0.0f, 0.0f, 0.0f,
    0.0f, 0.0f, 1.0f, 0.0f, 0.0f,
    0.0f, 0.0f, 0.0f, 1.0f, 0.0f,
    0.2f, 0.2f, 0.2f, 0.0f, 1.0f};

imageAttributes.SetColorMatrix(
    &colorMatrix,
    ColorMatrixFlagsDefault,
    ColorAdjustTypeBitmap);

graphics.DrawImage(&image, 10, 10);

graphics.DrawImage(
    &image,
```

```
Rect(120, 10, width, height), // destination rectangle
0, 0, // upper-left corner of source rectangle
width, // width of source rectangle
height, // height of source rectangle
UnitPixel,
&imageAttributes);
```

下面的插图在左侧显示原来的图像，在右侧显示变换后的图像。



上面示例中的代码使用以下步骤进行重新着色：

- 1. 初始化 ColorMatrix 对象。
- 2. 创建一个 ImageAttributes 对象，并将 ColorMatrix 对象传递给 ImageAttributes 对象的 SetColorMatrix 方法。
- 3. 将 ImageAttributes 对象传递给 Graphics 对象的 DrawImage 方法。

转换图像颜色

平移是指在这四个颜色分量中的一个或多个中添加值。下表给出表示平移的颜色矩阵项。

要平移的分量	矩阵项
红色	[4][0]
绿色	[4][1]
蓝色	[4][2]
Alpha	[4][3]

下面的示例使用文件 ColorBars.bmp 构造 Image 对象。然后，该代码为该图像中每个像素的红色分量增加 0.75。原来的图像绘制在变换后的图像旁边。

```
Image image(L"ColorBars.bmp");
ImageAttributes imageAttributes;
UINT width = image.GetWidth();
UINT height = image.GetHeight();

ColorMatrix colorMatrix = {
    1.0f, 0.0f, 0.0f, 0.0f, 0.0f,
    0.0f, 1.0f, 0.0f, 0.0f, 0.0f,
```



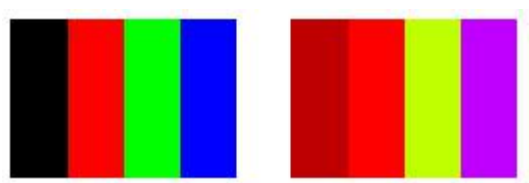
```
0.0f, 0.0f, 1.0f, 0.0f, 0.0f,
0.0f, 0.0f, 0.0f, 1.0f, 0.0f,
0.75f, 0.0f, 0.0f, 0.0f, 1.0f});

imageAttributes.SetColorMatrix(
    &colorMatrix,
    ColorMatrixFlagsDefault,
    ColorAdjustTypeBitmap);

graphics.DrawImage(&image, 10, 10, width, height);

graphics.DrawImage(
    &image,
    Rect(150, 10, width, height), // destination rectangle
    0, 0, // upper-left corner of source rectangle
    width, // width of source rectangle
    height, // height of source rectangle
    UnitPixel,
    &imageAttributes);
```

下面的插图在左侧显示原来的图像，在右侧显示变换后的图像。



下表列出在进行红色平移前后，四栏的颜色矢量。请注意，因为颜色分量的最大值是 1，所以第二行中的红色分量不发生改变。（同样，颜色分量的最小值为 0。）

原有	平移后
黑色 (0, 0, 0, 1)	(0.75, 0, 0, 1)
红色 (1, 0, 0, 1)	(1, 0, 0, 1)
绿色 (0, 1, 0, 1)	(0.75, 1, 0, 1)
蓝色 (0, 0, 1, 1)	(0.75, 0, 1, 1)

缩放颜色

缩放变换是指用一个数字与这四个颜色分量中的一个或多个相乘。下表给出表示缩放的色彩矩阵项。

要缩放的分量	矩阵项
--------	-----

Red	[0][0]
Green	[1][1]
Blue	[2][2]
Alpha	[3][3]

下面的示例从文件 **ColorBars2.bmp** 构造一个 **Image** 对象。然后，该代码将图像中每个像素的蓝色分量乘以 2。原来的图像绘制在变换后的图像旁边。

```
Image          image(L"ColorBars2.bmp");
ImageAttributes imageAttributes;
UINT           width  = image.GetWidth();
UINT           height = image.GetHeight();

ColorMatrix colorMatrix = {
    1.0f, 0.0f, 0.0f, 0.0f, 0.0f,
    0.0f, 1.0f, 0.0f, 0.0f, 0.0f,
    0.0f, 0.0f, 2.0f, 0.0f, 0.0f,
    0.0f, 0.0f, 0.0f, 1.0f, 0.0f,
    0.0f, 0.0f, 0.0f, 0.0f, 1.0f};

imageAttributes.SetColorMatrix(
    &colorMatrix,
    ColorMatrixFlagsDefault,
    ColorAdjustTypeBitmap);

graphics.DrawImage(&image, 10, 10, width, height);

graphics.DrawImage(
    &image,
    Rect(150, 10, width, height), // destination rectangle
    0, 0, // upper-left corner of source rectangle
    width, // width of source rectangle
    height, // height of source rectangle
    UnitPixel,
    &imageAttributes);
```

下面的插图在左侧显示原来的图像，在右侧显示缩放后的图像。



下表列出在进行蓝色缩放前后，四栏的颜色矢量。请注意，第四个颜色栏中的蓝色分量从 **0.8** 变到 **0.6**。这是因为 GDI+ 只保留结果的小数部分。例如， $(2)(0.8) = 1.6$ ，1.6 的小数部分是 0.6。只保留小数部分可确保结果总是在 **[0, 1]** 之间。

原始图像	缩放后
(0.4, 0.4, 0.4, 1)	(0.4, 0.4, 0.8, 1)
(0.4, 0.2, 0.2, 1)	(0.4, 0.2, 0.4, 1)
(0.2, 0.4, 0.2, 1)	(0.2, 0.4, 0.4, 1)
(0.4, 0.4, 0.8, 1)	(0.4, 0.4, 0.6, 1)

下面的示例从文件 **ColorBars2.bmp** 构造一个 **Image** 对象。然后，该代码缩放图像中每个像素的红色、绿色和蓝色分量。红色分量缩小了 **25%**，绿色分量缩小了 **35%**，蓝色分量缩小了 **50%**。

```
Image          image(L"ColorBars.bmp");
ImageAttributes imageAttributes;
UINT           width  = image.GetWidth();
UINT           height = image.GetHeight();

ColorMatrix colorMatrix = {
    0.75f, 0.0f, 0.0f, 0.0f, 0.0f,
    0.0f, 0.65f, 0.0f, 0.0f, 0.0f,
    0.0f, 0.0f, 0.5f, 0.0f, 0.0f,
    0.0f, 0.0f, 0.0f, 1.0f, 0.0f,
    0.0f, 0.0f, 0.0f, 0.0f, 1.0f};

imageAttributes.SetColorMatrix(
    &colorMatrix,
    ColorMatrixFlagsDefault,
    ColorAdjustTypeBitmap);

graphics.DrawImage(&image, 10, 10, width, height);

graphics.DrawImage(
    &image,
    Rect(150, 10, width, height), // destination rectangle
    0, 0, // upper-left corner of source rectangle
    width, // width of source rectangle
    height, // height of source rectangle
    UnitPixel,
    &imageAttributes);
```

下面的插图在左侧显示原来的图像，在右侧显示缩放后的图像。

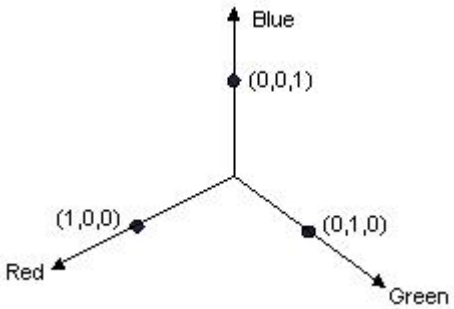


下表列出在缩放红色、绿色和蓝色前后，四栏的颜色矢量。

原始图像	缩放后
(0.6, 0.6, 0.6, 1)	(0.45, 0.39, 0.3, 1)
(0, 1, 1, 1)	(0, 0.65, 0.5, 1)
(1, 1, 0, 1)	(0.75, 0.65, 0, 1)
(1, 0, 1, 1)	(0.75, 0, 0.5, 1)

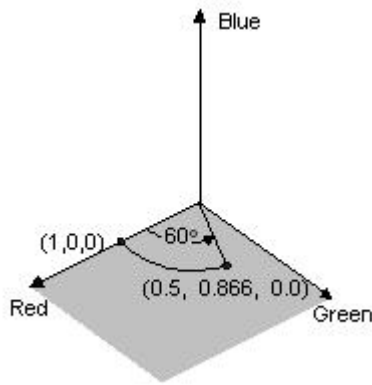
### 旋转颜色

在四维颜色空间中进行的旋转难于可视化。可通过固定一种颜色分量以便使旋转可视化。假设我们同意将 **alpha** 分量固定在 **1**（完全不透明），则可用红色、绿色和蓝色的轴形象地表示三维颜色空间，如下面的插图所示。

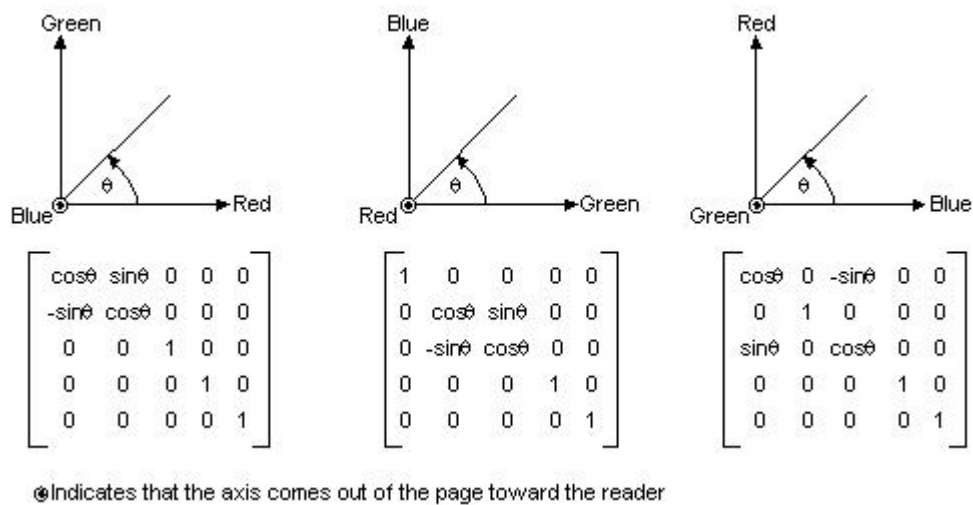


颜色可视为 3-D 空间中的一个点。例如，空间中的点 **(1, 0, 0)** 表示红色，空间中的点 **(0, 1, 0)** 表示绿色。

下面的插图显示将 **(1, 0, 0)** 颜色在红色-绿色平面中旋转 **60 度** 后的结果。在与红色-绿色平面平行的平面中进行旋转可视围绕蓝色轴旋转。



下面的插图显示如何初始化颜色矩阵以围绕三个坐标轴（红色、绿色和蓝色）中的每一个进行旋转。



下面的示例将一个使用一种颜色  $(1, 0, 0.6)$  的图像围绕蓝色轴旋转 60 度。旋转角度在与红色-绿色平面平行的平面上扫出。

```
Image          image(L"RotationInput.bmp");
ImageAttributes imageAttributes;
UINT           width  = image.GetWidth();
UINT           height = image.GetHeight();
REAL           degrees = 60;
REAL           pi = acos(-1); // the angle whose cosine is -1.
REAL           r = degrees * pi / 180; // degrees to radians

ColorMatrix colorMatrix = {
    cos(r),  sin(r), 0.0f, 0.0f, 0.0f,
    -sin(r), cos(r), 0.0f, 0.0f, 0.0f,
    0.0f,    0.0f,  1.0f, 0.0f, 0.0f,
    0.0f,    0.0f,  0.0f, 1.0f, 0.0f,
    0.0f,    0.0f,  0.0f, 0.0f, 1.0f};
```

```

imageAttributes.SetColorMatrix(
    &colorMatrix,
    ColorMatrixFlagsDefault,
    ColorAdjustTypeBitmap);

graphics.DrawImage(&image, 10, 10, width, height);

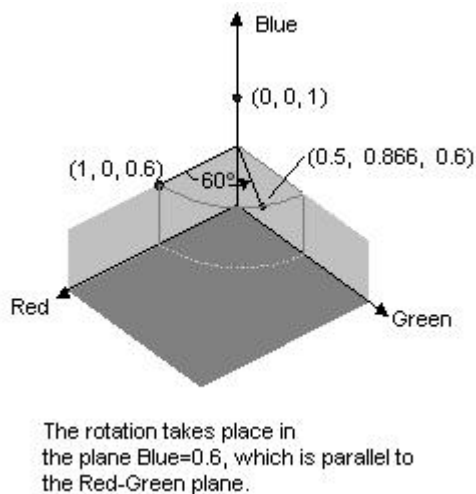
graphics.DrawImage(
    &image,
    Rect(130, 10, width, height), // destination rectangle
    0, 0, // upper-left corner of source rectangle
    width, // width of source rectangle
    height, // height of source rectangle
    UnitPixel,
    &imageAttributes);

```

下面的插图在左侧显示原来的图像，在右侧显示颜色旋转后的图像。



下面的插图形象地显示利用上面的代码进行的颜色旋转。



## 剪取颜色

剪切是指按照与另一种颜色分量成比例的量来增加或减少颜色分量。例如，考虑这样一种变换，将红色分量增加蓝色分量值的一半。在这样的变换下， $(0.2, 0.5, 1)$  颜色将变成  $(0.7, 0.5, 1)$ 。新的颜色分量是  $0.2 + (1/2)(1) = 0.7$ 。

下面的示例从文件 **ColorBars4.bmp** 构造一个 **Image** 对象。然后，该代码将上一段中描述的剪切变换应用到图像中的每个像素。

```
Image          image(L"ColorBars4.bmp");
ImageAttributes imageAttributes;
UINT           width  = image.GetWidth();
UINT           height = image.GetHeight();

ColorMatrix colorMatrix = {
    1.0f, 0.0f, 0.0f, 0.0f, 0.0f,
    0.0f, 1.0f, 0.0f, 0.0f, 0.0f,
    0.5f, 0.0f, 1.0f, 0.0f, 0.0f,
    0.0f, 0.0f, 0.0f, 1.0f, 0.0f,
    0.0f, 0.0f, 0.0f, 0.0f, 1.0f};

imageAttributes.SetColorMatrix(
    &colorMatrix,
    ColorMatrixFlagsDefault,
    ColorAdjustTypeBitmap);

graphics.DrawImage(&image, 10, 10, width, height);

graphics.DrawImage(
    &image,
    Rect(150, 10, width, height), // destination rectangle
    0, 0,                          // upper-left corner of source rectangle
    width,                         // width of source rectangle
    height,                        // height of source rectangle
    UnitPixel,
    &imageAttributes);
```

下面的插图在左侧显示原来的图像，在右侧显示剪切后的图像。



下表列出在剪切变换前后，四栏的颜色矢量。

原始值	剪切后
(0, 0, 1, 1)	(0.5, 0, 1, 1)

(0.5, 1, 0.5, 1)	(0.75, 1, 0.5, 1)
(1, 1, 0, 1)	(1, 1, 0, 1)
(0.4, 0.4, 0.4, 1)	(0.6, 0.4, 0.4, 1)

## 使用颜色重映射表

重映射是按照颜色重映射表转换图像中的颜色的过程。颜色重新映射表是 **ColorMap** 结构体数组。该数组中的每个 **ColorMap** 结构体有一个 **OldColor** 成员和一个 **NewColor** 成员。

**GDI+** 绘制图像时，图像的每个像素都与旧颜色的数组进行比较。如果像素的颜色与旧颜色相匹配，则它的颜色将更改为相应的新颜色。只改变颜色的呈现方式，图像本身的颜色值（存储在 **Image** 或 **Bitmap** 对象中）不发生改变。

若要绘制再映射的图像，请初始化 **ColorMap** 结构体数组。将该数组传递给 **ImageAttributes** 对象的 **SetRemapTable** 方法，然后将 **ImageAttributes** 对象地址传递给 **Graphics** 对象的 **DrawImage** 方法。

下面的示例从 **RemapInput.bmp** 文件创建 **Image** 对象。该代码创建由一个 **ColorMap** 结构体组成的颜色重新映射表。**ColorRemap** 结构体的 **OldColor** 成员为 **red**，**NewColor** 成员为 **blue**。图像被绘制两次，一次不进行再变换，另一次进行再变换。重映射过程将所有的红色像素都更改为蓝色。

```
Image          image(L"RemapInput.bmp");
ImageAttributes imageAttributes;
UINT           width  = image.GetWidth();
UINT           height = image.GetHeight();
ColorMap       colorMap[1];

colorMap[0].oldColor = Color(255, 255, 0, 0); // opaque red
colorMap[0].newColor = Color(255, 0, 0, 255); // opaque blue

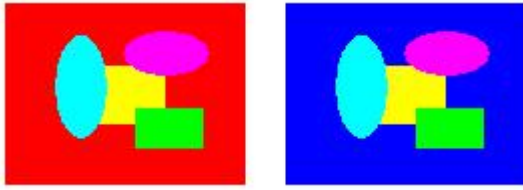
imageAttributes.SetRemapTable(1, colorMap, ColorAdjustTypeBitmap);

graphics.DrawImage(&image, 10, 10, width, height);

graphics.DrawImage(
    &image,
    Rect(150, 10, width, height), // destination rectangle
    0, 0,                          // upper-left corner of source rectangle
    width,                         // width of source rectangle
    height,                        // height of source rectangle
    UnitPixel,
    &imageAttributes);
```

下面的插图在左侧显示原来的图像，在右侧显示再变换后的图像。





## 打印

将您的代码稍微调整一下，您就可以将 **GDI+** 的结果输出到打印机而非显示器。若要在打印机上绘图，需要获取打印机的设备场景句柄，然后将它传给 **Graphics** 构造函数。将你的绘图命令放到 **StartDoc** 和 **EndDoc** 之间即可。

### 将 **GDI+** 输出至打印机

采用 **GDI+** 在打印机上绘图与在电脑显示屏上绘图相似。若要在打印机上绘图，需要获取打印机的设备场景句柄，然后将它传给 **Graphics** 构造函数。

下面的控制台应用程序将在名为 **MyPrinter** 的打印机上绘制一条直线、一个矩形和一个椭圆。

```
#include <windows.h>
#include <gdiplus.h>
#include <stdio.h>
using namespace Gdiplus;

INT main()
{
    // Initialize GDI+.
    GdiplusStartupInput gdiplusStartupInput;
    ULONG_PTR gdiplusToken;
    GdiplusStartup(&gdiplusToken, &gdiplusStartupInput, NULL);

    // Get a device context for the printer.
    HDC hdcPrint = CreateDC(NULL, TEXT("\\\\.\\printserver\\print1"), NULL, NULL);

    DOCINFO docInfo;
    ZeroMemory(&docInfo, sizeof(docInfo));
    docInfo.cbSize = sizeof(docInfo);
    docInfo.lpszDocName = "GdiplusPrint";

    StartDoc(hdcPrint, &docInfo);
    StartPage(hdcPrint);

    Graphics* graphics = new Graphics(hdcPrint);
    Pen* pen = new Pen(Color(255, 0, 0, 0));
    graphics->DrawLine(pen, 50, 50, 350, 550);
    graphics->DrawRectangle(pen, 50, 50, 300, 500);
```

```

    graphics->DrawEllipse(pen, 50, 50, 300, 500);
    delete pen;
    delete graphics;
    EndPage(hdcPrint);
    EndDoc(hdcPrint);

    DeleteDC(hdcPrint);
    GdiplusShutdown(gdiplusToken);
    return 0;
}

```

上面的代码中，三条绘图命令位于 **StartDoc** 和 **EndDoc** 函数之间，这两个函数都需要传入打印机的设备场景句柄。在 **StartDoc** 和 **EndDoc** 之间的所有的图形命令都会被发送到一个临时的图元文件。在调用了 **EndDoc** 之后，打印机驱动程序将图元文件的数据转换为指定的打印机所采用的格式。

#### 注意

如果所使用的打印机的假脱机不可用，那么图形输出不会发送到图元文件。取而代之的是，图形命令将被打印机驱动程序单条执行，然后发送给打印机。

通常您不愿意像上面例子中那样通过硬编码指定打印机的名字。您也还选择通过调用 **GetDefaultPrinter** 来获取默认打印机的名字。在调用 **GetDefaultPrinter** 之前，你必须预先分配足够的缓冲区已容纳打印机的名称。你可以在调用 **GetDefaultPrinter** 时将第一个参数置为 NULL 得到所需缓冲区的大小。

#### 注意

**GetDefaultPrinter** 函数只在 Window 2000 及以后版本中才支持。

下面的控制台应用程序获取默认打印机名称，然后在这台打印机上绘制一个矩形和椭圆。**DrawRectangle** 调用放在 **StartPage** 和 **EndPage** 之间，因此这个矩形独占一页。同样地，椭圆也独占一页。

```

#include <windows.h>
#include <gdiplus.h>
#include <stdio.h>
using namespace Gdiplus;

INT main()
{
    // Initialize GDI+.
    GdiplusStartupInput gdiplusStartupInput;
    ULONG_PTR gdiplusToken;
    GdiplusStartup(&gdiplusToken, &gdiplusStartupInput, NULL);

    DWORD size;
    HDC hdcPrint;

```

```

DOCINFO docInfo;
ZeroMemory(&docInfo, sizeof(docInfo));
docInfo.cbSize = sizeof(docInfo);
docInfo.lpszDocName = "GdiplusPrint";

// Get the size of the default printer name.
GetDefaultPrinter(NULL, &size);

// Allocate a buffer large enough to hold the printer name.
TCHAR* buffer = new TCHAR[size];

// Get the printer name.
if(!GetDefaultPrinter(buffer, &size))
{
    printf("Failure");
}
else
{
    // Get a device context for the printer.
    hdcPrint = CreateDC(NULL, buffer, NULL, NULL);

    StartDoc(hdcPrint, &docInfo);
    Graphics* graphics;
    Pen* pen = new Pen(Color(255, 0, 0, 0));

    StartPage(hdcPrint);
    graphics = new Graphics(hdcPrint);
    graphics->DrawRectangle(pen, 50, 50, 200, 300);
    delete graphics;
    EndPage(hdcPrint);

    StartPage(hdcPrint);
    graphics = new Graphics(hdcPrint);
    graphics->DrawEllipse(pen, 50, 50, 200, 300);
    delete graphics;
    EndPage(hdcPrint);

    delete pen;
    EndDoc(hdcPrint);

    DeleteDC(hdcPrint);
}

```

```

delete buffer;

GdiplusShutdown(gdiplusToken);
return 0;
}

```

### 显示一个打印对话框

获取打印机设备场景句柄的另一种方法是显示一个打印对话框，供用户选择一个打印机。**PrintDlg** 函数（用于显示打印对话框）具有一个参数，它传入一个 **PRINTDLG** 结构体的地址。**PRINTDLG** 结构体有好几个成员，不过你可以让它们保持默认值即可。有两个成员您需要设置：**IStructSize** 和 **Flags**。将 **IStructSize** 设置为 **PRINTDLG** 变量的大小，然后把 **Flags** 设置为 **PD\_RETURNDC**。把 **Flags** 设置为 **PD\_RETURNDC** 表示你希望 **PrintDlg** 函数把用户选择的打印机的设备场景句柄填入 **hDC** 域中。

```

#include <windows.h>
#include <gdiplus.h>
#include <stdio.h>
using namespace Gdiplus;

INT main()
{
    // Initialize GDI+.
    GdiplusStartupInput gdiplusStartupInput;
    ULONG_PTR gdiplusToken;
    GdiplusStartup(&gdiplusToken, &gdiplusStartupInput, NULL);

    DOCINFO docInfo;
    ZeroMemory(&docInfo, sizeof(docInfo));
    docInfo.cbSize = sizeof(docInfo);
    docInfo.lpszDocName = "GdiplusPrint";

    // Create a PRINTDLG structure, and initialize the appropriate fields.
    PRINTDLG printDlg;
    ZeroMemory(&printDlg, sizeof(printDlg));
    printDlg.lStructSize = sizeof(printDlg);
    printDlg.Flags = PD_RETURNDC;

    // Display a print dialog box.
    if(!PrintDlg(&printDlg))
    {
        printf("Failure\n");
    }
    else
    {
        // Now that PrintDlg has returned, a device context handle

```

```

// for the chosen printer is in printDlg->hDC.

StartDoc(printDlg.hDC, &docInfo);
StartPage(printDlg.hDC);
    Graphics* graphics = new Graphics(printDlg.hDC);
    Pen* pen = new Pen(Color(255, 0, 0, 0));
    graphics->DrawRectangle(pen, 200, 500, 200, 150);
    graphics->DrawEllipse(pen, 200, 500, 200, 150);
    graphics->DrawLine(pen, 200, 500, 400, 650);
    delete pen;
    delete graphics;
EndPage(printDlg.hDC);
EndDoc(printDlg.hDC);
}
if(printDlg.hDevMode)
    GlobalFree(printDlg.hDevMode);
if(printDlg.hDevNames)
    GlobalFree(printDlg.hDevNames);
if(printDlg.hDC)
    DeleteDC(printDlg.hDC);

GdiplusShutdown(gdiplusToken);
return 0;
}

```

### 通过提供打印机句柄优化打印

**Graphics** 类的构造函数之一接受一个设备场景句柄和一个打印机句柄。当你将 **GDI+** 指令发送到指定的附属打印机的时候，采用这种特殊的构造函数构造你的 **Graphics** 对象将获得更佳的性能。

下面的控制台应用程序调用 **GetDefaultPrinter** 来获取默认打印机名称。代码将打印机名称传给 **CreateDC** 用于获取该打印机的设备场景句柄。同时代码将打印机名称传给 **OpenPrinter** 用于获取打印机句柄。然后将打印机设备场景句柄和打印机句柄同时传递给 **Graphics** 构造函数。然后在打印机上绘制两个图形。

#### 注意

**GetDefaultPrinter** 函数只在 Window 2000 及以后版本中才支持。

```

#include <windows.h>
#include <gdiplus.h>
#include <stdio.h>
using namespace Gdiplus;

INT main()
{
    // Initialize GDI+.

```

```

GdiplusStartupInput gdiplusStartupInput;
ULONG_PTR gdiplusToken;
GdiplusStartup(&gdiplusToken, &gdiplusStartupInput, NULL);

DWORD   size;
HDC     hdcPrint;
HANDLE  printerHandle;

DOCINFO docInfo;
ZeroMemory(&docInfo, sizeof(docInfo));
docInfo.cbSize = sizeof(docInfo);
docInfo.lpszDocName = "GdiplusPrint";

// Get the length of the printer name.
GetDefaultPrinter(NULL, &size);
TCHAR* buffer = new TCHAR[size];

// Get the printer name.
if(!GetDefaultPrinter(buffer, &size))
{
    printf("Failure");
}
else
{
    // Get a device context for the printer.
    hdcPrint = CreateDC(NULL, buffer, NULL, NULL);

    // Get a printer handle.
    OpenPrinter(buffer, &printerHandle, NULL);

    StartDoc(hdcPrint, &docInfo);
    StartPage(hdcPrint);
    Graphics* graphics = new Graphics(hdcPrint, printerHandle);
    Pen* pen = new Pen(Color(255, 0, 0, 0));
    graphics->DrawRectangle(pen, 200, 500, 200, 150);
    graphics->DrawEllipse(pen, 200, 500, 200, 150);
    delete(pen);
    delete(graphics);
    EndPage(hdcPrint);
    EndDoc(hdcPrint);

    ClosePrinter(printerHandle);
    DeleteDC(hdcPrint);
}

```

```
delete buffer;

GdiplusShutdown(gdiplusToken);
return 0;
}
```

## 附录：GDI+ 参考

有关使用 C++ 开发语言进行 GDI+ 应用程序开发的详细参考信息，请参阅：

<ms-help://MS.MSDNQTR.v80.chs/MS.MSDN.v80/MS.WIN32COM.v10.en/gdicpp/GDIPlusreference.htm> 