

GDI+

Purpose

Windows GDI+ is a class-based API for C/C++ programmers. It enables applications to use graphics and formatted text on both the video display and the printer. Applications based on the Microsoft Win32 API do not access graphics hardware directly. Instead, GDI+ interacts with device drivers on behalf of applications. GDI+ is also supported by Microsoft Win64.

Where applicable

GDI+ functions and classes are not supported for use within a Windows service. Attempting to use these functions and classes from a Windows service may produce unexpected problems, such as diminished service performance and run-time exceptions or errors.

Note When you use the GDI+ API, you must never allow your application to download arbitrary fonts from untrusted sources. The operating system requires elevated privileges to assure that all installed fonts are trusted.

Developer audience

The GDI+ C++ class-based interface is designed for use by C/C++ programmers. Familiarity with the Windows graphical user interface and message-driven architecture is required.

Run-time requirements

GDI+ can be used in all Windows-based applications. GDI+ was introduced in Windows XP and Windows Server 2003. For information about which operating systems are required to use a particular class or method, see the More Information section of the documentation for the class or method. GDI+ is available as a redistributable for earlier operating systems. To download the latest redistributable, see <http://go.microsoft.com/fwlink/?LinkID=20993>.

Note If you are redistributing GDI+ to a down-level platform or a platform that does not ship with that version of GDI+ natively, install Gdiplus.dll in your application directory. This puts it in your address space, but you should use the linker's `/BASE` option to rebase the Gdiplus.dll to prevent address space conflict. For more information, see [/BASE \(Base Address\)](#).

Topic	Description
1) Overview	General information about GDI+.
2) Using	Tasks and examples using GDI+.
3) Reference	Documentation of GDI+ C++ class-based API.

1) Overview - About GDI+

Windows GDI+ is the portion of the Windows XP operating system or Windows Server 2003 operating system that provides two-dimensional vector graphics, imaging, and typography. GDI+ improves on Windows Graphics Device Interface (GDI) (the graphics device interface included with earlier versions of Windows) by adding new features and by optimizing existing features.

The following topics provide information about the GDI+ API with the C++ programming language.

- [Introduction to GDI+](#)
 - [What's New In GDI+?](#)
 - [Lines, Curves, and Shapes](#)
 - [Images, Bitmaps, and Metafiles](#)
 - [Coordinate Systems and Transformations](#)
 - [Graphics Containers](#)
-

1.1) Introduction to GDI+

Windows GDI+ is a graphics device interface that allows programmers to write device-independent applications. The services of GDI+ are exposed through a set of C++ classes.

- [Overview of GDI+](#)
 - [The Three Parts of GDI+](#)
 - [The Structure of the Class-Based Interface](#)
-

1.1.1) Overview of GDI+

Windows GDI+ is the subsystem of the Windows XP operating system or Windows Server 2003 that is responsible for displaying information on screens and printers. GDI+ is an API that is exposed through a set of C++ classes.

As its name suggests, GDI+ is the successor to Windows Graphics Device Interface (GDI), the graphics device interface included with earlier versions of Windows. Windows XP or Windows Server 2003 supports GDI for compatibility with existing applications, but programmers of new applications should use GDI+ for all their graphics needs because GDI+ optimizes many of the capabilities of GDI and also provides additional features.

A graphics device interface, such as GDI+, allows application programmers to display information on a screen or printer without having to be concerned about the details of a particular display device. The application programmer makes calls to methods provided by GDI+ classes and those methods in turn make the appropriate calls to specific device drivers. GDI+ insulates the application from the graphics hardware, and it is this insulation that allows developers to create device-independent applications.

1.1.2) The Three Parts of GDI+

The services of Windows GDI+ fall into the following three broad categories:

- [2-D vector graphics](#)
- [Imaging](#)
- [Typography](#)

=> 2-D vector graphics

Vector graphics involves drawing primitives (such as lines, curves, and figures) that are specified by sets of points on a coordinate system. For example, a straight line can be specified by its two endpoints, and a rectangle can be specified by a point giving the location of its upper-left corner and a pair of numbers giving its width and height. A simple path can be specified by an array of points to be connected by straight lines. A Bézier spline is a sophisticated curve specified by four control points.

GDI+ provides classes that store information about the primitives themselves, classes that store information about how the primitives are to be drawn, and classes that actually do the drawing. For example, the **Rect** class stores the location and size of a rectangle; the **Pen** class stores information about line color, line width, and line style; and the **Graphics** class has methods for drawing lines, rectangles, paths, and other figures. There are also several **Brush** classes that store information about how closed figures and paths are to be filled with colors or patterns.

=> Imaging

Certain kinds of pictures are difficult or impossible to display with the techniques of vector graphics. For example, the pictures on toolbar buttons and the pictures that appear as icons would be difficult to specify as collections of lines and curves. A high-resolution digital photograph of a crowded baseball stadium would be even more difficult to create with vector techniques. Images of this type are stored as bitmaps, arrays of numbers that represent the colors of individual dots on the screen. Data structures that store information about bitmaps tend to be more complex than those required for vector graphics, so there are several classes in GDI+ devoted to this purpose. An example of such a class is **CachedBitmap**, which is used to store a bitmap in memory for fast access and display.

=> Typography

Typography is concerned with the display of text in a variety of fonts, sizes, and styles. GDI+ provides an impressive amount of support for this complex task. One of the new features in GDI+ is subpixel antialiasing, which gives text rendered on an LCD screen a smoother appearance.

1.1.3) The Structure of the Class-Based Interface

The C++ interface to Windows GDI+ contains about 40 classes, 50 enumerations, and 6 structures. There are also a few functions that are not members of any class.

You must indicate that the namespace Gdiplus is being used before any GDI+ functions are called. The following statement indicates that the Gdiplus namespace is being used in the application.

```
using namespace Gdiplus;
```

The [Graphics](#) class is the core of the GDI+ interface; it is the class that actually draws lines, curves, figures, images, and text.

Many classes work together with the [Graphics](#) class. For example, the [Graphics::DrawLine](#) method receives a pointer to a [Pen](#) object, which holds attributes (color, width, dash style, and the like) of the line to be drawn. The [Graphics::FillRectangle](#) method can receive a pointer to a [LinearGradientBrush](#) object, which works with the **Graphics** object to fill a rectangle with a gradually changing color. [Font](#) and [StringFormat](#) objects influence the way a **Graphics** object draws text. A [Matrix](#) object stores and manipulates the world transformation of a **Graphics** object, which is used to rotate, scale, and flip images.

Certain classes serve primarily as structured data types. Some of those classes (for example, [Rect](#), [Point](#), and [Size](#)) are for general purposes. Others are for specialized purposes and are considered helper classes. For example, the [BitmapData](#) class is a helper for the [Bitmap](#) class, and the [PathData](#) class is a helper for the [GraphicsPath](#) class. GDI+ also defines a few structures that are used for organizing data. For example, the [ColorMap](#) structure holds a pair of [Color](#) objects that form one entry in a color conversion table.

GDI+ defines several enumerations, which are collections of related constants. For example, the [LineJoin](#) enumeration contains the elements [LineJoinBevel](#), [LineJoinMiter](#), and [LineJoinRound](#), which specify styles that can be used to join two lines.

GDI+ provides a few functions that are not part of any class. Two of those functions are [GdiplusStartup](#) and [GdiplusShutdown](#). You must call **GdiplusStartup** before you make any other GDI+ calls, and you must call **GdiplusShutdown** when you have finished using GDI+.

1.2) What's New In GDI+?

Windows GDI+ is different from Windows Graphics Device Interface (GDI) in a couple of ways. First, GDI+ expands on the features of GDI by providing new capabilities, such as gradient brushes and alpha blending. Second, the programming model has been revised to make graphics programming easier and more flexible.

- [New Features](#)
 - [Changes in the Programming Model](#)
-

1.2.1) New Features

The following sections describe several of the new features in Windows GDI+.

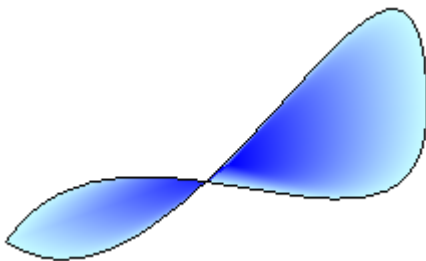
- [Gradient Brushes](#)
- [Cardinal Splines](#)
- [Independent Path Objects](#)
- [Transformations and the Matrix Object](#)
- [Scalable Regions](#)
- [Alpha Blending](#)
- [Support for Multiple Image Formats](#)

=> Gradient Brushes

GDI+ expands on Windows Graphics Device Interface (GDI) by providing linear gradient and path gradient brushes for filling shapes, paths, and regions. Gradient brushes can also be used to draw lines, curves, and paths. When you fill a shape with a linear gradient brush, the color gradually changes as you move across the shape. For example, suppose you create a horizontal gradient brush by specifying blue at the left edge of a shape and green at the right edge. When you fill that shape with the horizontal gradient brush, it will gradually change from blue to green as you move from its left edge to its right edge. Similarly, a shape filled with a vertical gradient brush will change color as you move from top to bottom. The following illustration shows an ellipse filled with a horizontal gradient brush and a region filled with a diagonal gradient brush.

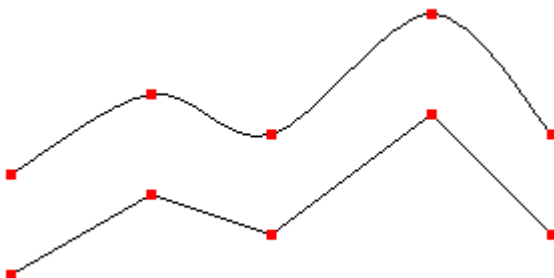


When you fill a shape with a path gradient brush, you have a variety of options for specifying how the colors change as you move from one portion of the shape to another. One option is to have a center color and a boundary color so that the pixels change gradually from one color to the other as you move from the middle of the shape towards the outer edges. The following illustration shows a path (created from a pair of Bézier splines) filled with a path gradient brush.



=> Cardinal Splines

GDI+ supports cardinal splines, which are not supported in GDI. A cardinal spline is a sequence of individual curves joined to form a larger curve. The spline is specified by an array of points and passes through each point in that array. A cardinal spline passes smoothly (no sharp corners) through each point in the array and thus is more refined than a path created by connecting straight lines. The following illustration shows two paths, one created by connecting straight lines and one created as a cardinal spline.

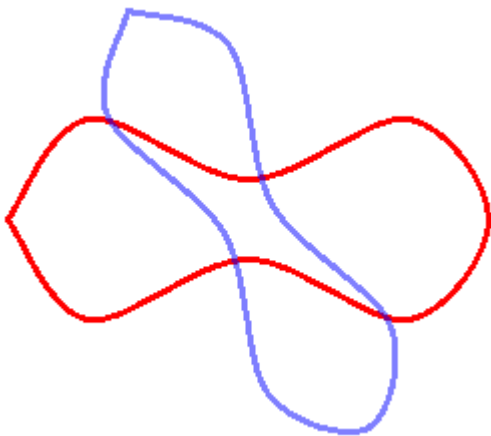


=> Independent Path Objects

In GDI, a path belongs to a device context, and the path is destroyed as it is drawn. With GDI+, drawing is performed by a [Graphics](#) object, and you can create and maintain several [GraphicsPath](#) objects that are separate from the **Graphics** object. A **GraphicsPath** object is not destroyed by the drawing action, so you can use the same **GraphicsPath** object to draw a path several times.

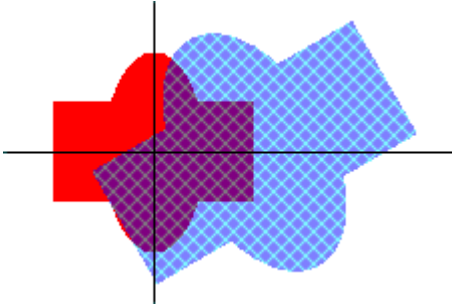
=> Transformations and the Matrix Object

GDI+ provides the [Matrix](#) object, a powerful tool that makes transformations (rotations, translations, and so on) easy and flexible. A matrix object works in conjunction with the objects that are transformed. For example, a [GraphicsPath](#) object has a [GraphicsPath::Transform](#) method that receives the address of a **Matrix** object as an argument. A single 3×3 matrix can store one transformation or a sequence of transformations. The following illustration shows a path before and after a sequence of two transformations (first scale, then rotate).



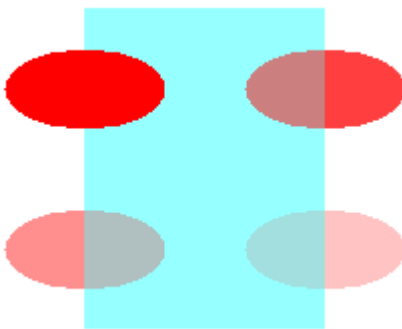
=> Scalable Regions

GDI+ expands greatly on GDI with its support for regions. In GDI, regions are stored in device coordinates, and the only transformation that can be applied to a region is a translation. GDI+ stores regions in world coordinates and allows a region to undergo any transformation (scaling, for example) that can be stored in a transformation matrix. The following illustration shows a region before and after a sequence of three transformations: scale, rotate, and translate.



=> Alpha Blending

Note that in the previous figure, you can see the untransformed region (filled with red) through the transformed region (filled with a hatch brush). This is made possible by alpha blending, which is supported by GDI+. With alpha blending, you can specify the transparency of a fill color. A transparent color is blended with the background color — the more transparent you make a fill color, the more the background shows through. The following illustration shows four ellipses that are filled with the same color (red) at different transparency levels.



=> Support for Multiple Image Formats

GDI+ provides the [Image](#), [Bitmap](#), and [Metafile](#) classes, which allow you to load, save and manipulate images in a variety of formats. The following formats are supported:

- BMP
- Graphics Interchange Format (GIF)
- JPEG
- Exif
- PNG
- TIFF
- ICON
- WMF
- EMF

1.2.2) Changes in the Programming Model

The following sections describe several ways that programming with Windows GDI+ is different from programming with Windows Graphics Device Interface (GDI).

- [Device Contexts, Handles, and Graphics Objects](#)
- [Two Ways to Draw a Line](#)
 - [Drawing a line with GDI](#)
 - [Drawing a line with GDI+ and the C++ class interface](#)
- [Pens, Brushes, Paths, Images, and Fonts as Parameters](#)
- [Method Overloading](#)
- [No More Current Position](#)
- [Separate Methods for Draw and Fill](#)
- [Constructing Regions](#)

=> Device Contexts, Handles, and Graphics Objects

If you have written programs using GDI (the graphics device interface included in previous versions of Windows), you are familiar with the idea of a device context (DC). A device context is a structure used by Windows to store information about the capabilities of a particular display device and attributes that specify how items will be drawn on that device. A device context for a video display is also associated with a particular window on the display. First you obtain a handle to a device context (HDC), and then you pass that handle as an argument to GDI functions that actually do the drawing. You also pass the handle as an argument to GDI functions that obtain or set the attributes of the device context.

When you use GDI+, you don't have to be as concerned with handles and device contexts as you do when you use GDI. You simply create a [Graphics](#) object and then invoke its methods in the familiar object-oriented style — `myGraphicsObject.DrawLine(parameters)`. The **Graphics** object is at the core of GDI+ just as the device context is at the core of GDI. The device context and the **Graphics** object play similar roles, but there are some fundamental differences between the handle-based programming model used with device contexts (GDI) and the object-oriented model used with **Graphics** objects (GDI+).

The [Graphics](#) object, like the device context, is associated with a particular window on the screen and contains attributes (for example, smoothing mode and text rendering hint) that specify how items are to be drawn. However, the **Graphics** object is not tied to a pen, brush, path, image, or font as a device context is. For example, in GDI, before you can use a device context to draw a line, you must call [SelectObject](#) to associate a pen object with the device context. This is referred to as selecting the pen into the device context. All lines drawn in the device context will use that pen until you select a different pen. With GDI+, you pass a [Pen](#) object as an argument to the [DrawLine](#) method of the **Graphics** class. You can use a different **Pen** object in each of a series of `DrawLine` calls without having to associate a given **Pen** object with a **Graphics** object.

=> Two Ways to Draw a Line

The following two examples each draw a red line of width 3 from location (20, 10) to location (200,100). The first example calls GDI, and the second calls GDI+ through the C++ class interface.

- [Drawing a line with GDI](#)
- [Drawing a line with GDI+ and the C++ class interface](#)

+> Drawing a line with GDI

To draw a line with GDI, you need two objects: a device context and a pen. You get a handle to a device context by calling [BeginPaint](#), and a handle to a pen by calling [CreatePen](#). Next, you call [SelectObject](#) to select the pen into the device context. You set the pen position to (20, 10) by calling [MoveToEx](#) and then draw a line from that pen position to (200, 100) by calling [LineTo](#). Note that [MoveToEx](#) and [LineTo](#) both receive **hdc** as an argument.

```
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);
```

+> Drawing a line with GDI+ and the C++ class interface

To draw a line with GDI+ and the C++ class interface, you need a [Graphics](#) object and a [Pen](#) object. Note that you don't ask Windows for handles to these objects. Instead, you use constructors to create an instance of the **Graphics** class (a **Graphics** object) and an instance of the **Pen** class (a **Pen** object). Drawing a line involves calling the [Graphics::DrawLine](#) method of the **Graphics** class. The first parameter of the **Graphics::DrawLine** method is a pointer to your **Pen** object. This is a simpler and more flexible scheme than selecting a pen into a device context as shown in the preceding GDI example.

```
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);

```

=> Pens, Brushes, Paths, Images, and Fonts as Parameters

The preceding examples show that [Pen](#) objects can be created and maintained separately from the [Graphics](#) object, which supplies the drawing methods. [Brush](#), [GraphicsPath](#), [Image](#), and [Font](#) objects can also be created and maintained separately from the **Graphics** object. Many of the drawing methods provided by the **Graphics** class receive a **Brush**, **GraphicsPath**, **Image**, or **Font** object as an argument. For example, the address of a **Brush** object is passed as an argument to the [FillRectangle](#) method, and the address of a **GraphicsPath** object is passed as an argument to the [Graphics::DrawPath](#) method. Similarly, addresses of **Image** and **Font** objects are passed to the [DrawImage](#) and [DrawString](#) methods. This is in contrast to GDI where you select a brush, path, image, or font into the device context and then pass a handle to the device context as an argument to a drawing function.

=> Method Overloading

Many of the GDI+ methods are overloaded; that is, several methods share the same name but have different parameter lists. For example, the [DrawLine](#) method of the [Graphics](#) class comes in the following forms:

```

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);
```

All four of the [DrawLine](#) variations above receive a pointer to a [Pen](#) object, the coordinates of the starting point, and the coordinates of the ending point. The first two variations receive the coordinates as floating point numbers, and the last two variations receive the coordinates as integers. The first and third variations receive the coordinates as a list of four separate numbers, while the second and fourth variations receive the coordinates as a pair of [Point](#) (or [PointF](#)) objects.

=> No More Current Position

Note that in the [DrawLine](#) methods shown previously both the starting point and the ending point of the line are received as arguments. This is a departure from the GDI scheme where you call to set the current pen position followed by to draw a line starting at (x1, y1) and ending at (x2, y2). GDI+ as a whole has abandoned the notion of current position.

=> Separate Methods for Draw and Fill

GDI+ is more flexible than GDI when it comes to drawing the outlines and filling the interiors of shapes like rectangles. GDI has a [Rectangle](#) function that draws the outline and fills the interior of a rectangle all in one step. The outline is drawn with the currently selected pen, and the interior is filled with the currently selected brush.

```
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);
```

GDI+ has separate methods for drawing the outline and filling the interior of a rectangle. The [DrawRectangle](#) method of the [Graphics](#) class has the address of a [Pen](#) object as one of its parameters, and the [FillRectangle](#) method has the address of a [Brush](#) object as one of its parameters.

```
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);
```

Note that the [FillRectangle](#) and [DrawRectangle](#) methods in GDI+ receive arguments that specify the rectangle's left edge, top, width, and height. This is in contrast to the [GDIRectangle](#) function, which takes arguments that specify the rectangle's left edge, right edge, top, and bottom. Also note that the constructor for the [Color](#) class in GDI+ has four parameters. The last three parameters are the usual red, green, and blue values; the first parameter is the alpha value, which specifies the extent to which the color being drawn is blended with the background color.

=> Constructing Regions

GDI provides several functions for creating regions: `CreateRectRgn`, `CreateEllpticRgn`, `CreateRoundRectRgn`, `CreatePolygonRgn`, and `CreatePolyPolygonRgn`. You might expect the [Region](#) class in GDI+ to have analogous constructors that take rectangles, ellipses, rounded rectangles, and polygons as arguments, but that is not the case. The **Region** class in GDI+ provides a constructor that receives a [Rect](#) object reference and another constructor that receives the address of a [GraphicsPath](#) object. If you want to construct a region based on an ellipse, rounded rectangle, or polygon, you can easily do so by creating a **GraphicsPath** object (that contains an ellipse, for example) and then passing the address of that **GraphicsPath** object to a **Region** constructor.

GDI+ makes it easy to form complex regions by combining shapes and paths. The [Region](#) class has [Union](#) and [Intersect](#) methods that you can use to augment an existing region with a path or another region. One nice feature of the GDI+ scheme is that a [GraphicsPath](#) object is not destroyed when it is passed as an argument to a **Region** constructor. In GDI, you can convert a path to a region with the [PathToRegion](#) function, but the path is destroyed in the process. Also, a **GraphicsPath** object is not destroyed when its address is passed as an argument to a `Union` or `Intersect` method, so you can use a given path as a building block for several separate regions. This is shown in the following example. Assume that **onePath** is a pointer to a **GraphicsPath** object (simple or complex) that has already been initialized.

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

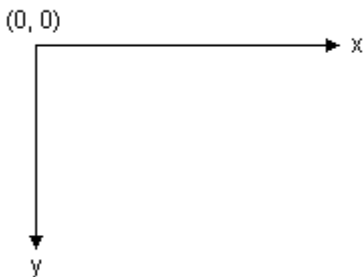
1.3) Lines, Curves, and Shapes

The vector graphics portion of GDI+ is used to draw lines, to draw curves, and to draw and fill shapes.

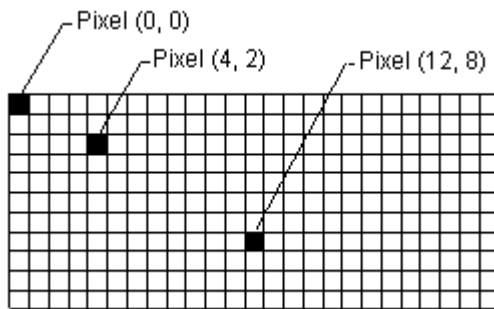
- [Overview of Vector Graphics](#)
- [Pens, Lines, and Rectangles](#)
- [Ellipses and Arcs](#)
- [Polygons](#)
- [Cardinal Splines](#)
- [Bézier Splines](#)
- [Paths](#)
- [Brushes and Filled Shapes](#)
- [Open and Closed Curves](#)
- [Regions](#)
- [Clipping](#)
- [Flattening Paths](#)
- [Antialiasing with Lines and Curves](#)

1.3.1) Overview of Vector Graphics

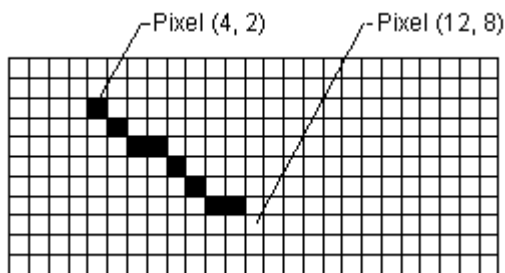
Windows GDI+ draws lines, rectangles, and other figures on a coordinate system. You can choose from a variety of coordinate systems, but the default coordinate system has the origin in the upper left corner with the x-axis pointing to the right and the y-axis pointing down. The unit of measure in the default coordinate system is the pixel.



A computer monitor creates its display on a rectangular array of dots called picture elements or pixels. The number of pixels appearing on the screen varies from one monitor to the next, and the number of pixels appearing on an individual monitor can usually be configured to some extent by the user.



When you use GDI+ to draw a line, rectangle, or curve, you provide certain key information about the item to be drawn. For example, you can specify a line by providing two points, and you can specify a rectangle by providing a point, a height, and a width. GDI+ works in conjunction with the display driver software to determine which pixels must be turned on to show the line, rectangle, or curve. The following illustration shows the pixels that are turned on to display a line from the point (4, 2) to the point (12, 8).



Over time, certain basic building blocks have proven to be the most useful for creating two-dimensional pictures. These building blocks, which are all supported by GDI+, are given in the following list:

- Lines
- Rectangles
- Ellipses
- Arcs
- Polygons
- Cardinal splines
- Bézier splines

The [Graphics](#) class in GDI+ provides the following methods for drawing the items in the previous list: [DrawLine](#), [DrawRectangle](#), [DrawEllipse](#), [DrawPolygon](#), [DrawArc](#), [DrawCurve](#) (for cardinal splines), and [DrawBezier](#). Each of these methods is overloaded; that is, each method comes in several variations with different parameter lists. For example, one variation of the [DrawLine](#) method receives the address of a [Pen](#) object and four integers, while another variation of the [DrawLine](#) method receives the address of a [Pen](#) object and two [Point](#) object references.

The methods for drawing lines, rectangles, and Bézier splines have plural companion methods that draw several items in a single call: [DrawLines](#), [DrawRectangles](#), and [DrawBeziers](#). Also, the [DrawCurve](#) method has a companion method, [DrawClosedCurve](#), that closes a curve by connecting the ending point of the curve to the starting point.

All the drawing methods of the [Graphics](#) class work in conjunction with a [Pen](#) object. Thus, in order to draw anything, you must create at least two objects: a **Graphics** object and a **Pen** object. The **Pen** object stores attributes of the item to be drawn, such as line width and color. The address of the **Pen** object is passed as one of the arguments to the drawing method. For example, one variation of the [DrawRectangle](#) method receives the address of a **Pen** object and four integers as shown in the following code, which draws a rectangle with a width of 100, a height of 50 and an upper-left corner of (20, 10).

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

1.3.2) Pens, Lines, and Rectangles

To draw lines with Windows GDI+ you need to create a [Graphics](#) object and a [Pen](#) object. The **Graphics** object provides the methods that actually do the drawing, and the **Pen** object stores attributes of the line, such as color, width, and style. Drawing a line is simply a matter of calling the [DrawLine](#) method of the **Graphics** object. The address of the **Pen** object is passed as one of the arguments to the [DrawLine](#) method. The following example draws a line from the point (4, 2) to the point (12, 6).

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

[DrawLine](#) is an overloaded method of the [Graphics](#) class, so there are several ways you can supply it with arguments. For example, you can construct two [Point](#) objects and pass references to the **Point** objects as arguments to the [DrawLine](#) method.

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

You can specify certain attributes when you construct a [Pen](#) object. For example, one [Pen](#) constructor allows you to specify color and width. The following example draws a blue line of width 2 from (0, 0) to (60, 30).


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

The [Pen](#) object also has attributes, such as dash style, that you can use to specify features of the line. For example, the following example draws a dashed line from (100, 50) to (300, 80).

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

You can use various methods of the [Pen](#) object to set many more attributes of the line. The [Pen::SetStartCap](#) and [Pen::SetEndCap](#) methods specify the appearance of the ends of the line; the ends can be flat, square, rounded, triangular, or a custom shape. The [Pen::SetLineJoin](#) method lets you specify whether connected lines are mitered (joined with sharp corners), beveled, rounded, or clipped. The following illustration shows lines with various cap and join styles.



Drawing rectangles with GDI+ is similar to drawing lines. To draw a rectangle, you need a [Graphics](#) object and a [Pen](#) object. The **Graphics** object provides a [DrawRectangle](#) method, and the **Pen** object stores attributes, such as line width and color. The address of the **Pen** object is passed as one of the arguments to the DrawRectangle method. The following example draws a rectangle with its upper-left corner at (100, 50), a width of 80, and a height of 40.

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

[DrawRectangle](#) is an overloaded method of the [Graphics](#) class, so there are several ways you can supply it with arguments. For example, you can construct a [Rect](#) object and pass a reference to the **Rect** object as an argument to the DrawRectangle method.

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

A [Rect](#) object has methods for manipulating and gathering information about the rectangle. For example, the [Inflate](#) and [Offset](#) methods change the size and position of the rectangle. The [Rect::Intersects](#)

method tells you whether the rectangle intersects another given rectangle, and the [Contains](#) method tells you whether a given point is inside the rectangle.

1.3.3) Ellipses and Arcs

An ellipse is specified by its bounding rectangle. The following illustration shows an ellipse along with its bounding rectangle.



To draw an ellipse, you need a [Graphics](#) object and a [Pen](#) object. The **Graphics** object provides the [DrawEllipse](#) method, and the **Pen** object stores attributes of the ellipse, such as line width and color. The address of the **Pen** object is passed as one of the arguments to the DrawEllipse method. The remaining arguments passed to the DrawEllipse method specify the bounding rectangle for the ellipse. The following example draws an ellipse; the bounding rectangle has a width of 160, a height of 80, and an upper-left corner of (100, 50).

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

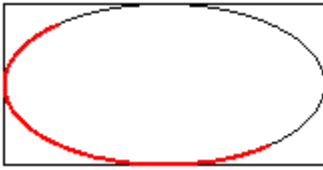
[DrawEllipse](#) is an overloaded method of the [Graphics](#) class, so there are several ways you can supply it with arguments. For example, you can construct a [Rect](#) object and pass a reference to the **Rect** object as an argument to the DrawEllipse method.

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

An arc is a portion of an ellipse. To draw an arc, you call the [DrawArc](#) method of the [Graphics](#) class. The parameters of the DrawArc method are the same as the parameters of the [DrawEllipse](#) method, except that DrawArc requires a starting angle and sweep angle. The following example draws an arc with a starting angle of 30 degrees and a sweep angle of 180 degrees.

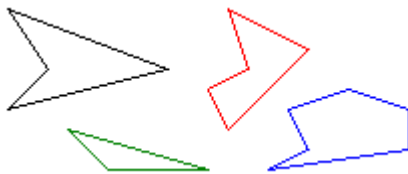
```
myGraphics.DrawArc(&myPen, 100, 50, 160, 80, 30, 180);
```

The following illustration shows the arc, the ellipse, and the bounding rectangle.



1.3.4) Polygons

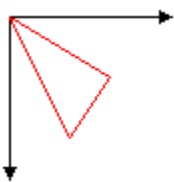
A polygon is a closed figure with three or more straight sides. For example, a triangle is a polygon with three sides, a rectangle is a polygon with four sides, and a pentagon is a polygon with five sides. The following illustration shows several polygons.



To draw a polygon, you need a [Graphics](#) object, a [Pen](#) object, and an array of [Point](#) (or [PointF](#)) objects. The **Graphics** object provides the [DrawPolygon](#) method. The **Pen** object stores attributes of the polygon, such as line width and color, and the array of **Point** objects stores the points to be connected by straight lines. The addresses of the **Pen** object and the array of **Point** objects are passed as arguments to the DrawPolygon method. The following example draws a three-sided polygon. Note that there are only three points in **myPointArray**: (0, 0), (50, 30), and (30, 60). The DrawPolygon method automatically closes the polygon by drawing a line from (30, 60) back to the starting point (0, 0);

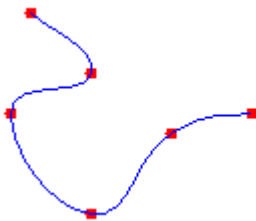
```
Point myPointArray[] =  
    {Point(0, 0), Point(50, 30), Point(30, 60)};  
myGraphics.DrawPolygon(&myPen, myPointArray, 3);
```

The following illustration shows the polygon.



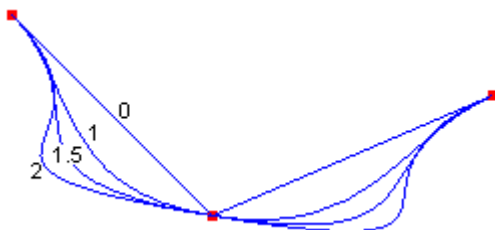
1.3.5) Cardinal Splines

A cardinal spline is a sequence of individual curves joined to form a larger curve. The spline is specified by an array of points and a tension parameter. A cardinal spline passes smoothly through each point in the array; there are no sharp corners and no abrupt changes in the tightness of the curve. The following illustration shows a set of points and a cardinal spline that passes through each point in the set.



A physical spline is a thin piece of wood or other flexible material. Before the advent of mathematical splines, designers used physical splines to draw curves. A designer would place the spline on a piece of paper and anchor it to a given set of points. The designer could then create a curve by drawing along the spline with a pencil. A given set of points could yield a variety of curves, depending on the properties of the physical spline. For example, a spline with a high resistance to bending would produce a different curve than an extremely flexible spline.

The formulas for mathematical splines are based on the properties of flexible rods, so the curves produced by mathematical splines are similar to the curves that were once produced by physical splines. Just as physical splines of different tension will produce different curves through a given set of points, mathematical splines with different values for the tension parameter will produce different curves through a given set of points. The following illustration shows four cardinal splines passing through the same set of points. The tension is shown for each spline. Note that a tension of 0 corresponds to infinite physical tension, forcing the curve to take the shortest way (straight lines) between points. A tension of 1 corresponds to no physical tension, allowing the spline to take the path of least total bend. With tension values greater than 1, the curve behaves like a compressed spring, pushed to take a longer path.



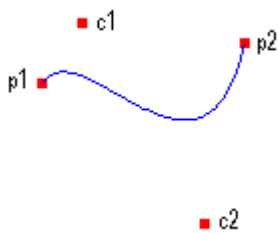
Note that the four splines in the preceding figure share the same tangent line at the starting point. The tangent is the line drawn from the starting point to the next point along the curve. Likewise, the shared tangent at the ending point is the line drawn from the ending point to the previous point on the curve.

To draw a cardinal spline, you need a [Graphics](#) object, a [Pen](#) object, and an array of [Point](#) objects. The **Graphics** object provides the [DrawCurve](#) method, which draws the spline, and the **Pen** object stores attributes of the spline, such as line width and color. The array of **Point** objects stores the points that the curve will pass through. The following example draws a cardinal spline that passes through the points in *myPointArray*. The third parameter is the tension.

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

1.3.6) Bezier Splines

A Bézier spline is a curve specified by four points: two end points (p1 and p2) and two control points (c1 and c2). The curve begins at p1 and ends at p2. The curve doesn't pass through the control points, but the control points act as magnets, pulling the curve in certain directions and influencing the way the curve bends. The following illustration shows a Bézier curve along with its endpoints and control points.

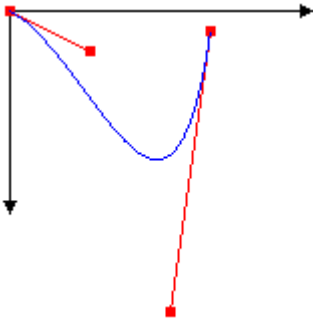


Note that the curve starts at p1 and moves toward the control point c1. The tangent line to the curve at p1 is the line drawn from p1 to c1. Also note that the tangent line at the endpoint p2 is the line drawn from c2 to p2.

To draw a Bézier spline, you need a [Graphics](#) object and a [Pen](#) object. The **Graphics** object provides the [DrawBezier](#) method, and the **Pen** object stores attributes of the curve, such as line width and color. The address of the **Pen** object is passed as one of the arguments to the DrawBezier method. The remaining arguments passed to the DrawBezier method are the endpoints and the control points. The following example draws a Bézier spline with starting point (0, 0), control points (40, 20) and (80, 150), and ending point (100, 10).

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

The following illustration shows the curve, the control points, and two tangent lines.



Bézier splines were originally developed by Pierre Bézier for design in the automotive industry. They have since proven to be very useful in many types of computer-aided design and are also used to define the outlines of fonts. Bézier splines can yield a wide variety of shapes, some of which are shown in the following illustration.

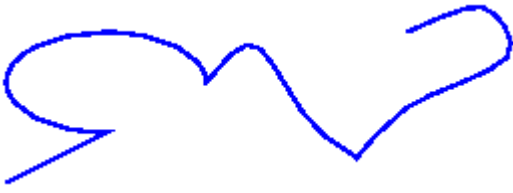


1.3.7) Paths

Paths are formed by combining lines, rectangles, and simple curves. Recall from the [Overview of Vector Graphics](#) that the following basic building blocks have proven to be the most useful for drawing pictures.

- Lines
- Rectangles
- Ellipses
- Arcs
- Polygons
- Cardinal splines
- Bézier splines

In Windows GDI+, the [GraphicsPath](#) object allows you to collect a sequence of these building blocks into a single unit. The entire sequence of lines, rectangles, polygons, and curves can then be drawn with one call to the [Graphics::DrawPath](#) method of the [Graphics](#) class. The following illustration shows a path created by combining a line, an arc, a Bézier spline, and a cardinal spline.



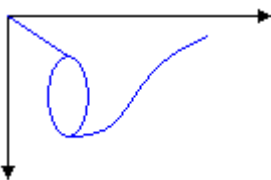
The [GraphicsPath](#) class provides the following methods for creating a sequence of items to be drawn: [AddLine](#), [AddRectangle](#), [AddEllipse](#), [AddArc](#), [AddPolygon](#), [AddCurve](#) (for cardinal splines), and [AddBezier](#). Each of these methods is overloaded; that is, each method comes in several variations with different parameter lists. For example, one variation of the [AddLine](#) method receives four integers, and another variation of the [AddLine](#) method receives two [Point](#) objects.

The methods for adding lines, rectangles, and Bézier splines to a path have plural companion methods that add several items to the path in a single call: [AddLines](#), [AddRectangles](#), and [AddBeziers](#). Also, the [AddCurve](#) method has a companion method, [AddClosedCurve](#), that adds a closed curve to the path.

To draw a path, you need a [Graphics](#) object, a [Pen](#) object, and a [GraphicsPath](#) object. The **Graphics** object provides the [Graphics::DrawPath](#) method, and the **Pen** object stores attributes of the path, such as line width and color. The **GraphicsPath** object stores the sequence of lines, rectangles, and curves that make up the path. The addresses of the **Pen** object and the **GraphicsPath** object are passed as arguments to the **Graphics::DrawPath** method. The following example draws a path that consists of a line, an ellipse, and a Bézier spline.

```
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);
```

The following illustration shows the path.



In addition to adding lines, rectangles, and curves to a path, you can add paths to a path. This allows you to combine existing paths to form large, complex paths. The following code adds **graphicsPath1** and **graphicsPath2** to **myGraphicsPath**. The second parameter of the [GraphicsPath::AddPath](#) method specifies whether the added path is connected to the existing path.

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

There are two other items you can add to a path: strings and pies. A pie is a portion of the interior of an ellipse. The following example creates a path from an arc, a cardinal spline, a string, and a pie.

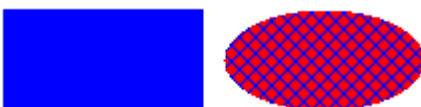
```
myGraphicsPath.AddArc(0, 0, 30, 20, -90, 180);  
myGraphicsPath.AddCurve(myPointArray, 3);  
myGraphicsPath.AddString(L"a string in a path", 18, &myFontFamily,  
    0, 24, myPointF, &myStringFormat);  
myGraphicsPath.AddPie(230, 10, 40, 40, 40, 110);  
myGraphics.DrawPath(&myPen, &myGraphicsPath);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

The following illustration shows the path. Note that a path does not have to be connected; the arc, cardinal spline, string, and pie are separated.



1.3.8) Brushes and Filled Shapes

A closed figure such as a rectangle or an ellipse consists of an outline and an interior. The outline is drawn with a [Pen](#) object and the interior is filled with a [Brush](#) object. Windows GDI+ provides several brush classes for filling the interiors of closed figures: [SolidBrush](#), [HatchBrush](#), [TextureBrush](#), [LinearGradientBrush](#), and [PathGradientBrush](#). All these classes inherit from the **Brush** class. The following illustration shows a rectangle filled with a solid brush and an ellipse filled with a hatch brush.



- [Solid Brushes](#)
- [Hatch Brushes](#)
- [Texture Brushes](#)
- [Gradient Brushes](#)

=> Solid Brushes

To fill a closed shape, you need a [Graphics](#) object and a [Brush](#) object. The **Graphics** object provides methods, such as [FillRectangle](#) and [FillEllipse](#), and the **Brush** object stores attributes of the fill, such as color and pattern. The address of the **Brush** object is passed as one of the arguments to the fill method. The following example fills an ellipse with a solid red color.

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

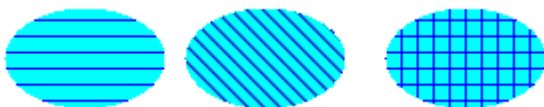
Note that in the preceding example, the brush is of type [SolidBrush](#), which inherits from [Brush](#).

=> Hatch Brushes

When you fill a shape with a hatch brush, you specify a foreground color, a background color, and a hatch style. The foreground color is the color of the hatching.

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

GDI+ provides more than 50 hatch styles, specified in [HatchStyle](#). The three styles shown in the following illustration are Horizontal, ForwardDiagonal, and Cross.



=> Texture Brushes

With a texture brush, you can fill a shape with a pattern stored in a bitmap. For example, suppose the following picture is stored in a disk file named MyTexture.bmp.



The following example fills an ellipse by repeating the picture stored in MyTexture.bmp.

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

The following illustration shows the filled ellipse.



=> Gradient Brushes

You can use a gradient brush to fill a shape with a color that changes gradually from one part of the shape to another. For example, a horizontal gradient brush will change color as you move from the left side of a figure to the right side. The following example fills an ellipse with a horizontal gradient brush that changes from blue to green as you move from the left side of the ellipse to the right side.

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

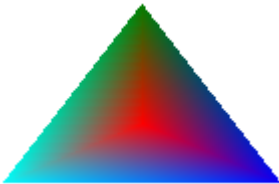
The following illustration shows the filled ellipse.



A path gradient brush can be configured to change color as you move from the center of a figure toward the boundary.



Path gradient brushes are quite flexible. The gradient brush used to fill the triangle in the following illustration changes gradually from red at the center to each of three different colors at the vertices.



1.3.9) Open and Closed Curves

The following illustration shows two curves: one open and one closed.



Closed curves have an interior and therefore can be filled with a brush. The [Graphics](#) class in Windows GDI+ provides the following methods for filling closed figures and curves: [FillRectangle](#), [FillEllipse](#), [FillPie](#), [FillPolygon](#), [FillClosedCurve](#), [Graphics::FillPath](#), and [Graphics::FillRegion](#). Whenever you call one of these methods, you must pass the address of one of the specific brush types ([SolidBrush](#), [HatchBrush](#), [TextureBrush](#), [LinearGradientBrush](#), or [PathGradientBrush](#)) as an argument.

The [FillPie](#) method is a companion to the [DrawArc](#) method. Just as the [DrawArc](#) method draws a portion of the outline of an ellipse, the [FillPie](#) method fills a portion of the interior of an ellipse. The following example draws an arc and fills the corresponding portion of the interior of the ellipse.

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

The following illustration shows the arc and the filled pie.



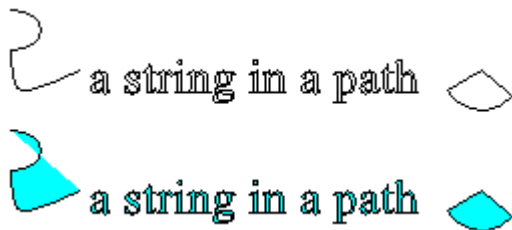
The [FillClosedCurve](#) method is a companion to the [DrawClosedCurve](#) method. Both methods automatically close the curve by connecting the ending point to the starting point. The following example draws a curve that passes through (0, 0), (60, 20), and (40, 50). Then, the curve is automatically closed by connecting (40, 50) to the starting point (0, 0), and the interior is filled with a solid color.

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

A path can consist of several figures (subpaths). The [Graphics::FillPath](#) method fills the interior of each figure. If a figure is not closed, the **Graphics::FillPath** method fills the area that would be enclosed if the figure were closed. The following example draws and fills a path that consists of an arc, a cardinal spline, a string, and a pie.

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

The following illustration shows the path before and after it is filled with a solid brush. Note that the text in the string is outlined, but not filled, by the [Graphics::DrawPath](#) method. It is the [Graphics::FillPath](#) method that paints the interiors of the characters in the string.



1.3.10) Regions

A region is a portion of the display surface. Regions can be simple (a single rectangle) or complex (a combination of polygons and closed curves). The following illustration shows two regions: one constructed from a rectangle, and the other constructed from a path.



Regions are often used for clipping and hit testing. Clipping involves restricting drawing to a certain region of the screen, usually the portion of the screen that needs to be updated. Hit testing involves checking to see whether the cursor is in a certain region of the screen when a mouse button is pressed.

You can construct a region from a rectangle or from a path. You can also create complex regions by combining existing regions. The [Region](#) class provides the following methods for combining regions: [Intersect](#), [Union](#), [Xor](#), [Exclude](#), and [Region::Complement](#).

The intersection of two regions is the set of all points belonging to both regions. The union is the set of all points belonging to one or the other or both regions. The complement of a region is the set of all points that are not in the region. The following illustration shows the intersection and union of the two regions in the previous figure.



Intersection

Union

The [Xor](#) method, applied to a pair of regions, produces a region that contains all points that belong to one region or the other, but not both. The [Exclude](#) method, applied to a pair of regions, produces a region that contains all points in the first region that are not in the second region. The following illustration shows the regions that result from applying the Xor and Exclude methods to the two regions shown at the beginning of this topic.



Xor



The curved region
excluded from the
rectangular region

To fill a region, you need a [Graphics](#) object, a [Brush](#) object, and a [Region](#) object. The **Graphics** object provides the [Graphics::FillRegion](#) method, and the **Brush** object stores attributes of the fill, such as color or pattern. The following example fills a region with a solid color.

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

1.3.11) Clipping

Clipping involves restricting drawing to a certain region. The following illustration shows the string "Hello" clipped to a heart-shaped region.



Regions can be constructed from paths, and paths can contain the outlines of strings, so you can use outlined text for clipping. The following illustration shows a set of concentric ellipses clipped to the interior of a string of text.



To draw with clipping, create a [Graphics](#) object, call its [SetClip](#) method, and then call the drawing methods of that same **Graphics** object. The following example draws a line that is clipped to a rectangular region.

```
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);
```

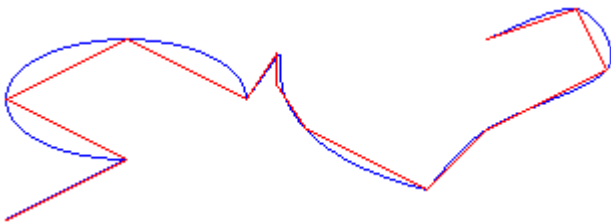
The following illustration shows the rectangular region along with the clipped line.



1.3.12) Flattening Paths

A [GraphicsPath](#) object stores a sequence of lines and Bézier splines. You can add several types of curves (ellipses, arcs, cardinal splines) to a path, but each curve is converted to a Bézier spline before it is stored in the path. Flattening a path consists of converting each Bézier spline in the path to a sequence of straight lines.

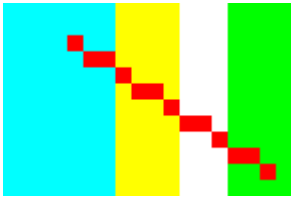
To flatten a path, call the [GraphicsPath::Flatten](#) method of a [GraphicsPath](#) object. The **GraphicsPath::Flatten** method receives a flatness argument that specifies the maximum distance between the flattened path and the original path. The following illustration shows a path before and after flattening.



1.2.13) Antialiasing with Lines and Curves

When you use Windows GDI+ to draw a line, you provide the starting point and ending point of the line, but you don't have to provide any information about the individual pixels on the line. GDI+ works in conjunction with the display driver software to determine which pixels will be turned on to show the line on a particular display device.

Consider a straight red line that goes from the point (4, 2) to the point (16, 10). Assume the coordinate system has its origin in the upper-left corner and that the unit of measure is the pixel. Also assume that the x-axis points to the right and the y-axis points down. The following illustration shows an enlarged view of the red line drawn on a multicolored background.



Note that the red pixels used to render the line are opaque. There are no partially transparent pixels involved in displaying the line. This type of line rendering gives the line a jagged appearance, and the line looks a bit like a staircase. This technique of representing a line with a staircase is called aliasing; the staircase is an alias for the theoretical line.

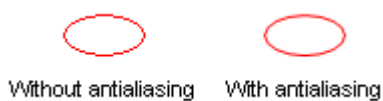
A more sophisticated technique for rendering a line involves using partially transparent pixels along with pure red pixels. Pixels are set to pure red or to some blend of red and the background color depending on how close they are to the line. This type of rendering is called antialiasing and results in a line that the human eye perceives as more smooth. The following illustration shows how certain pixels are blended with the background to produce an antialiased line.



Antialiasing (smoothing) can also be applied to curves. The following illustration shows an enlarged view of a smoothed ellipse.



The following illustration shows the same ellipse in its actual size, once without antialiasing and once with antialiasing.



To draw lines and curves that use antialiasing, create a [Graphics](#) object and pass *SmoothingModeAntiAlias* to its [Graphics::SetSmoothingMode](#) method. Then call one of the drawing methods of that same **Graphics** object.

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

SmoothingModeAntiAlias is an element of the [SmoothingMode](#) enumeration.

1.4) Images, Bitmaps, and Metafiles

Windows GDI+ provides the [Image](#) class for working with raster images (bitmaps) and vector images (metafiles). The [Bitmap](#) class and the [Metafile](#) class both inherit from the **Image** class. The **Bitmap** class expands on the capabilities of the **Image** class by providing additional methods for loading, saving, and manipulating raster images. The **Metafile** class expands on the capabilities of the **Image** class by providing additional methods for recording and examining vector images.

- [Types of Bitmaps](#)
 - [Metafiles](#)
 - [Drawing, Positioning, and Cloning Images](#)
 - [Cropping and Scaling Images](#)
-

1.4.1) Types of Bitmaps

A bitmap is an array of bits that specifies the color of each pixel in a rectangular array of pixels. The number of bits devoted to an individual pixel determines the number of colors that can be assigned to that pixel. For example, if each pixel is represented by 4 bits, then a given pixel can be assigned one of 16 different colors ($2^4 = 16$). The following table shows a few examples of the number of colors that can be assigned to a pixel represented by a given number of bits.

Bits per pixel	Number of colors that can be assigned to a pixel
----------------	--

1	$2^1 = 2$
2	$2^2 = 4$
4	$2^4 = 16$
8	$2^8 = 256$
16	$2^{16} = 65,536$
24	$2^{24} = 16,777,216$

Disk files that store bitmaps usually contain one or more information blocks that store information such as number of bits per pixel, number of pixels in each row, and number of rows in the array. Such a file might also contain a color table (sometimes called a color palette). A color table maps numbers in the bitmap to specific colors. The following illustration shows an enlarged image along with its bitmap and color table. Each pixel is represented by a 4-bit number, so there are $2^4 = 16$ colors in the color table. Each color in the table is represented by a 24-bit number: 8 bits for red, 8 bits for green, and 8 bits for blue. The numbers are shown in hexadecimal (base 16) form: A = 10, B = 11, C = 12, D = 13, E = 14, F = 15.

```

3 3 3 3 3 3 3 3
0 1 4 1 4 1 4 0
0 4 1 4 1 4 1 0
0 5 5 5 5 5 5 0
0 5 5 5 5 5 5 0
0 1 4 1 4 1 4 0
0 4 1 4 1 4 1 0
2 2 2 2 2 2 2 2

```



0	000000	
1	FF0000	
2	00FF00	
3	0000FF	
4	FFFFFF	
5	FFFF00	
6	FF00FF	
7	00FFFF	
8	FF0080	
9	FF8040	
A	804000	
B	008080	
C	800000	
D	800080	
E	8080FF	

Look at the pixel in row 3, column 5 of the image. The corresponding number in the bitmap is 1. The color table tells us that 1 represents the color red, so the pixel is red. All the entries in the top row of the bitmap are 3. The color table tells us that 3 represents blue, so all the pixels in the top row of the image are blue.

Note Some bitmaps are stored in bottom-up format; the numbers in the first row of the bitmap correspond to the pixels in the bottom row of the image.

A bitmap that stores indexes into a color table is called a *palette-indexed* bitmap. Some bitmaps have no need for a color table. For example, if a bitmap uses 24 bits per pixel, that bitmap can store the colors themselves rather than indexes into a color table. The following illustration shows a bitmap that stores colors directly (24 bits per pixel) rather than using a color table. The illustration also shows an enlarged view of the corresponding image. In the bitmap, FFFFFFFF represents white, FF0000 represents red, 00FF00 represents green, and 0000FF represents blue.

```

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

```



=> Graphics File Formats

There are many standard formats for saving bitmaps in files. Windows GDI+ supports the graphics file formats described in the following paragraphs.

+> Bitmap (BMP)

BMP is a standard format used by Windows to store device-independent and application-independent images. The number of bits per pixel (1, 4, 8, 15, 24, 32, or 64) for a given BMP file is specified in a file header. BMP files with 24 bits per pixel are common.

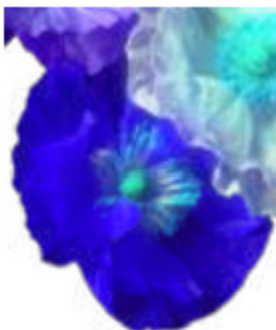
+> Graphics Interchange Format (GIF)

GIF is a common format for images that appear on Web pages. GIFs work well for line drawings, pictures with blocks of solid color, and pictures with sharp boundaries between colors. GIFs are compressed, but no information is lost in the compression process; a decompressed image is exactly the same as the original. One color in a GIF can be designated as transparent, so that the image will have the background color of any Web page that displays it. A sequence of GIF images can be stored in a single file to form an animated GIF. GIFs store at most 8 bits per pixel, so they are limited to 256 colors.

+> Joint Photographic Experts Group (JPEG)

JPEG is a compression scheme that works well for natural scenes, such as scanned photographs. Some information is lost in the compression process, but often the loss is imperceptible to the human eye. Color JPEG images store 24 bits per pixel, so they are capable of displaying more than 16 million colors. There is also a grayscale JPEG format that stores 8 bits per pixel. JPEGs do not support transparency or animation.

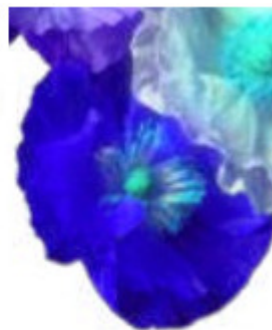
The level of compression in JPEG images is configurable, but higher compression levels (smaller files) result in more loss of information. A 20:1 compression ratio often produces an image that the human eye finds difficult to distinguish from the original. The following illustration shows a BMP image and two JPEG images that were compressed from that BMP image. The first JPEG has a compression ratio of 4:1 and the second JPEG has a compression ratio of about 8:1.



BMP 67 KB



JPEG 17 KB
Compression 4:1



JPEG 8 KB
Compression 8:1

JPEG compression does not work well for line drawings, blocks of solid color, and sharp boundaries. The following illustration shows a BMP along with two JPEGs and a GIF. The JPEGs and the GIF were compressed from the BMP. The compression ratio is 4:1 for the GIF, 4:1 for the smaller JPEG, and 8:3 for the larger JPEG. Note that the GIF maintains the sharp boundaries along the lines, but the JPEGs tends to blur the boundaries.



JPEG is a compression scheme, not a file format. JPEG File Interchange Format (JFIF) is a file format commonly used for storing and transferring images that have been compressed according to the JPEG scheme. JFIF files displayed by Web browsers use the .jpg extension.

+=> Exchangeable Image File (Exif)

Exif is a file format used for photographs captured by digital cameras. An Exif file contains an image that is compressed according to the JPEG specification. An Exif file also contains information about the photograph (date taken, shutter speed, exposure time, and so on) and information about the camera (manufacturer, model, and so on).

+=> Portable Network Graphics (PNG)

The PNG format retains many of the advantages of the GIF format but also provides capabilities beyond those of GIF. Like GIF files, PNG files are compressed with no loss of information. PNG files can store colors with 8, 24, or 48 bits per pixel and gray scales with 1, 2, 4, 8, or 16 bits per pixel. In contrast, GIF files can use only 1, 2, 4, or 8 bits per pixel. A PNG file can also store an alpha value for each pixel, which specifies the degree to which the color of that pixel is blended with the background color.

PNG improves on GIF in its ability to progressively display an image; that is, to display better and better approximations of the image as it arrives over a network connection. PNG files can contain gamma correction and color correction information so that the images can be accurately rendered on a variety of display devices.

+=> Tag Image File Format (TIFF)

TIFF is a flexible and extendable format that is supported by a wide variety of platforms and image-processing applications. TIFF files can store images with an arbitrary number of bits per pixel and can employ a variety of compression algorithms. Several images can be stored in a single, multiple-page TIFF file. Information related to the image (scanner make, host computer, type of

compression, orientation, samples per pixel, and so on) can be stored in the file and arranged through the use of tags. The TIFF format can be extended as needed by the approval and addition of new tags.

1.4.2) Metafiles

Windows GDI+ provides the [Metafile](#) class so that you can record and display metafiles. A metafile, also called a vector image, is an image that is stored as a sequence of drawing commands and settings. The commands and settings recorded in a **Metafile** object can be stored in memory or saved to a file or stream.

GDI+ can display metafiles that have been stored in the following formats:

- Windows Metafile Format (WMF)
- Enhanced Metafile (EMF)
- EMF+

GDI+ can record metafiles in the EMF and EMF+ formats, but not in the WMF format.

EMF+ is an extension to EMF that allows GDI+ records to be stored. There are two variations on the EMF+ format: EMF+ Only and EMF+ Dual. EMF+ Only metafiles contain only GDI+ records. Such metafiles can be displayed by GDI+ but not by Windows Graphics Device Interface (GDI). EMF+ Dual metafiles contain GDI+ and GDI records. Each GDI+ record in an EMF+ Dual metafile is paired with an alternate GDI record. Such metafiles can be displayed by GDI+ or by GDI.

The following example records one setting and one drawing command in a disk file. Note that the example creates a [Graphics](#) object and that the constructor for the **Graphics** object receives the address of a [Metafile](#) object as an argument.

```
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;
```

As the preceding example shows, the [Graphics](#) class is the key to recording instructions and settings in a [Metafile](#) object. Any call made to a method of a **Graphics** object can be recorded in a **Metafile** object.

Likewise, you can set any property of a **Graphics** object and record that setting in a **Metafile** object. The recording ends when the **Graphics** object is deleted or goes out of scope.

The following example displays the metafile created in the preceding example. The metafile is displayed with its upper-left corner at (100, 100).

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

The following example records several property settings (clipping region, world transformation, and smoothing mode) in a [Metafile](#) object. Then the code records several drawing instructions. The instructions and settings are saved in a disk file.

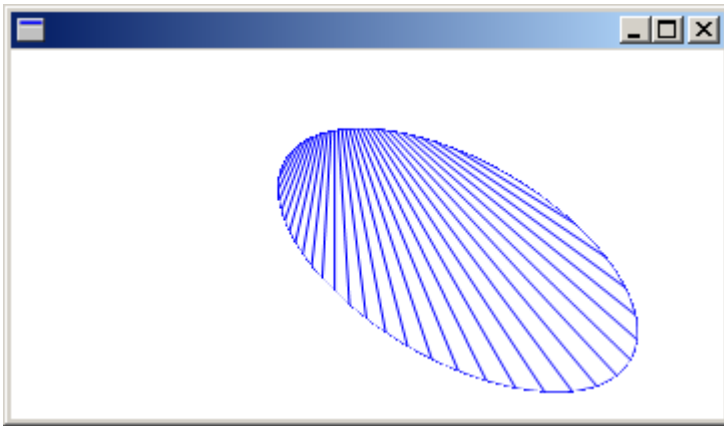
```
myMetafile = new Metafile(L"MyDiskFile2.emf", hdc);  
myGraphics = new Graphics(myMetafile);  
    myGraphics->SetSmoothingMode(SmoothingModeAntiAlias);  
    myGraphics->RotateTransform(30);  
  
    // Create an elliptical clipping region.  
    GraphicsPath myPath;  
    myPath.AddEllipse(0, 0, 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;
```

The following example displays the metafile image created in the preceding example.

```
myGraphics = new Graphics(hdc);  
myMetafile = new Metafile(L"MyDiskFile.emf");
```

```
myGraphics->DrawImage(myMetafile, 10, 10);
```

The following illustration shows the output of the preceding code. Note the antialiasing, the elliptical clipping region, and the 30-degree rotation.



1.4.3) Drawing, Positioning, and Cloning Images

You can use the [Image](#) class to load and display raster images (bitmaps) and vector images (metafiles). To display an image, you need a [Graphics](#) object and an **Image** object. The **Graphics** object provides the [Graphics::DrawImage](#) method, which receives the address of the **Image** object as an argument.

The following example constructs an [Image](#) object from the file Climber.jpg and then displays the image. The destination point for the upper-left corner of the image, (10, 10), is specified in the second and third parameters of the [Graphics::DrawImage](#) method.

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

The preceding code, along with a particular file, Climber.jpg, produced the following output.



You can construct [Image](#) objects from a variety of graphics file formats: BMP, GIF, JPEG, Exif, PNG, TIFF, WMF, EMF, and ICON.

The following example constructs [Image](#) objects from a variety of file types and then displays the images.

```
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);
```

The [Image](#) class provides a [Image::Clone](#) method that you can use to make a copy of an existing **Image**, [Metafile](#), or [Bitmap](#) object. The [Clone](#) method is overloaded in the **Bitmap** class, and one of the variations has a source-rectangle parameter that you can use to specify the portion of the original image that you want to copy. The following example creates a **Bitmap** object by cloning the top half of an existing **Bitmap** object. Then both images are displayed.

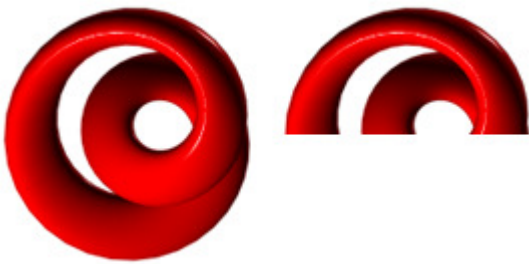
```
Bitmap* originalBitmap = new Bitmap(L"Spiral.png");
RectF sourceRect(
    0.0f,
    0.0f,
    (REAL)(originalBitmap->GetWidth()),
```

```
(REAL)(originalBitmap->GetHeight())/2.0f);

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

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

The preceding code, along with a particular file, Spiral.png, produced the following output.



1.4.4) Cropping and Scaling Images

You can use the [DrawImage](#) method of the [Graphics](#) class to draw and position images. DrawImage is an overloaded method, so there are several ways you can supply it with arguments. One variation of the [Graphics::DrawImage](#) method receives the address of an [Image](#) object and a reference to a [Rect](#) object. The rectangle specifies the destination for the drawing operation; that is, it specifies the rectangle in which the image will be drawn. If the size of the destination rectangle is different from the size of the original image, the image is scaled to fit the destination rectangle. The following example draws the same image three times: once with no scaling, once with an expansion, and once with a compression.

```
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);
```

The preceding code, along with a particular file, Spiral.png, produced the following output.



Some variations of the [Graphics::DrawImage](#) method have a source-rectangle parameter as well as a destination-rectangle parameter. The source rectangle specifies the portion of the original image that will be drawn. The destination rectangle specifies where that portion of the image will be drawn. If the size of the destination rectangle is different from the size of the source rectangle, the image is scaled to fit the destination rectangle.

The following example constructs a [Bitmap](#) object from the file Runner.jpg. The entire image is drawn with no scaling at (0, 0). Then a small portion of the image is drawn twice: once with a compression and once with an expansion.

```
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);

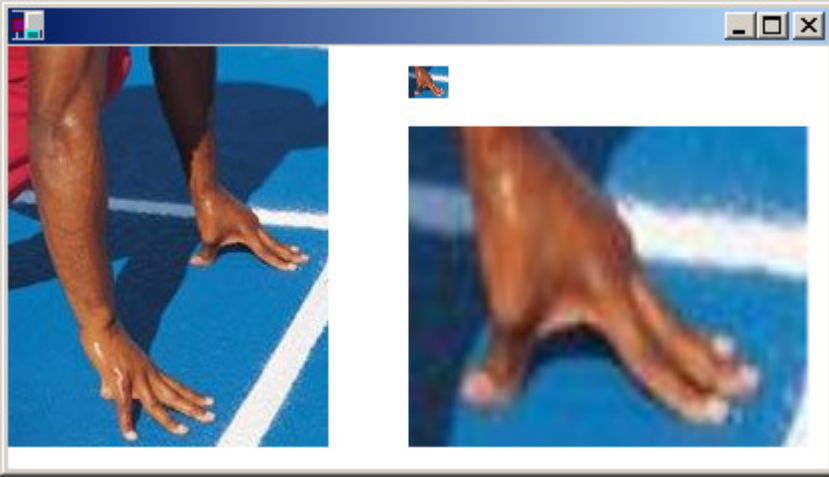
// Large destination rectangle for expanded hand.
Rect destRect2(200, 40, 200, 160);

// 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);
```

The following illustration shows the unscaled image, and the compressed and expanded image portions.



1.5) Coordinate Systems and Transformations

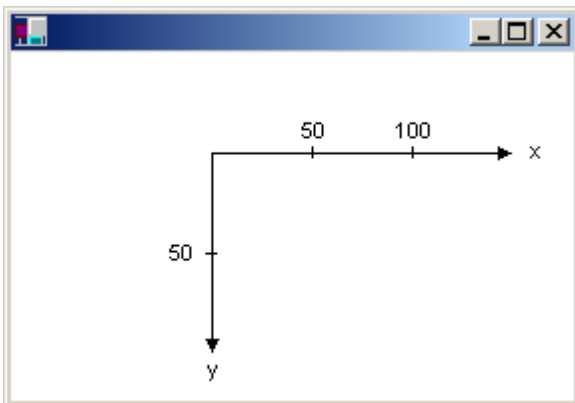
Windows GDI+ provides a world transformation and a page transformation so that you can transform (rotate, scale, translate, and so on) the items you draw. The two transformations also allow you to work in a variety of coordinate systems.

- [Types of Coordinate Systems](#)
 - [Matrix Representation of Transformations](#)
 - [Global and Local Transformations](#)
-

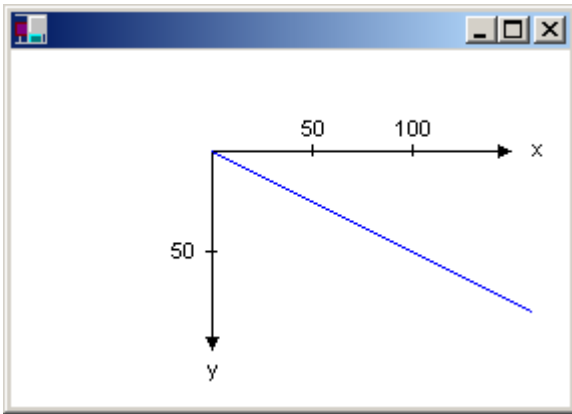
1.5.1) Types of Coordinate Systems

Windows GDI+ uses three coordinate spaces: world, page, and device. When you make the call `myGraphics.DrawLine(&myPen, 0, 0, 160, 80)`, the points that you pass to the [Graphics::DrawLine](#) method — (0, 0) and (160, 80) — are in the world coordinate space. Before GDI+ can draw the line on the screen, the coordinates pass through a sequence of transformations. One transformation converts world coordinates to page coordinates, and another transformation converts page coordinates to device coordinates.

Suppose you want to work with a coordinate system that has its origin in the body of the client area rather than the upper-left corner. Say, for example, that you want the origin to be 100 pixels from the left edge of the client area and 50 pixels from the top of the client area. The following illustration shows such a coordinate system.



When you make the call `myGraphics.DrawLine(&myPen, 0, 0, 160, 80)`, you get the line shown in the following illustration.



The coordinates of the endpoints of your line in the three coordinate spaces are as follows:

World (0, 0) to (160, 80)

Page (100, 50) to (260, 130)

Device (100, 50) to (260, 130)

Note that the page coordinate space has its origin at the upper-left corner of the client area; this will always be the case. Also note that because the unit of measure is the pixel, the device coordinates are the same as the page coordinates. If you set the unit of measure to something other than pixels (for example, inches), then the device coordinates will be different from the page coordinates.

The transformation that maps world coordinates to page coordinates is called the *world transformation* and is maintained by a [Graphics](#) object. In the previous example, the world transformation is a translation 100 units in the x direction and 50 units in the y direction. The following example sets the world transformation of a **Graphics** object and then uses that **Graphics** object to draw the line shown in the previous figure.

```
myGraphics.TranslateTransform(100.0f, 50.0f);
```

```
myGraphics.DrawLine(&myPen, 0, 0, 160, 80);
```

The transformation that maps page coordinates to device coordinates is called the *page transformation*. The [Graphics](#) class provides four methods for manipulating and inspecting the page transformation: [Graphics::SetPageUnit](#), [Graphics::GetPageUnit](#), [Graphics::SetPageScale](#), and [Graphics::GetPageScale](#). The **Graphics** class also provides two methods, [Graphics::GetDpiX](#) and [Graphics::GetDpiY](#), for examining the horizontal and vertical dots per inch of the display device.

You can use the [Graphics::SetPageUnit](#) method of the [Graphics](#) class to specify a unit of measure. The following example draws a line from (0, 0) to (2, 1) where the point (2, 1) is 2 inches to the right and 1 inch down from the point (0, 0).

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

Note If you don't specify a pen width when you construct your pen, the previous example will draw a line that is one inch wide. You can specify the pen width in the second argument to the [Pen](#) constructor:

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

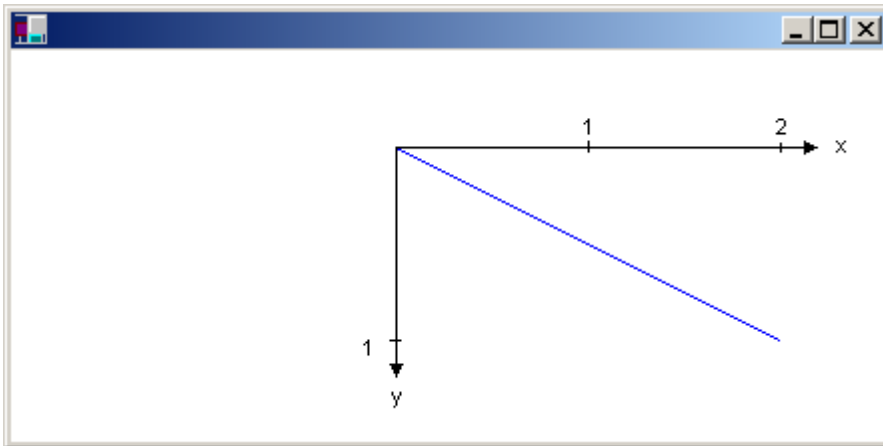
If we assume that the display device has 96 dots per inch in the horizontal direction and 96 dots per inch in the vertical direction, the endpoints of the line in the previous example have the following coordinates in the three coordinate spaces:

```
World (0, 0) to (2, 1)  
Page (0, 0) to (2, 1)  
Device (0, 0, to (192, 96)
```

You can combine the world and page transformations to achieve a variety of effects. For example, suppose you want to use inches as the unit of measure and you want the origin of your coordinate system to be 2 inches from the left edge of the client area and 1/2 inch from the top of the client area. The following example sets the world and page transformations of a [Graphics](#) object and then draws a line from (0, 0) to (2, 1).

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

The following illustration shows the line and coordinate system.



If we assume that the display device has 96 dots per inch in the horizontal direction and 96 dots per inch in the vertical direction, the endpoints of the line in the previous example have the following coordinates in the three coordinate spaces:

World (0, 0) to (2, 1)

Page (2, 0.5) to (4, 1.5)

Device (192, 48) to (384, 144)

1.5.2) Matrix Representation of Transformations

An $m \times n$ matrix is a set of numbers arranged in m rows and n columns. The following illustration shows several matrices.

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

$$\begin{array}{ccc}
 \begin{bmatrix} 2 & 0 \\ 0 & 3.5 \end{bmatrix} & \begin{bmatrix} 5 \\ 3 \end{bmatrix} & \begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 40 & 20 & 1 \end{bmatrix} \\
 2 \times 2 & 2 \times 1 & 3 \times 3
 \end{array}$$

You can add two matrices of the same size by adding individual elements. The following illustration shows two examples of matrix addition.

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

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

An $m \times n$ matrix can be multiplied by an $n \times p$ matrix, and the result is an $m \times p$ matrix. The number of columns in the first matrix must be the same as the number of rows in the second matrix. For example, a 4×2 matrix can be multiplied by a 2×3 matrix to produce a 4×3 matrix.

Points in the plane and rows and columns of a matrix can be thought of as vectors. For example, $(2, 5)$ is a vector with two components, and $(3, 7, 1)$ is a vector with three components. The dot product of two vectors is defined as follows:

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

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

For example, the dot product of $(2, 3)$ and $(5, 4)$ is $(2)(5) + (3)(4) = 22$. The dot product of $(2, 5, 1)$ and $(4, 3, 1)$ is $(2)(4) + (5)(3) + (1)(1) = 24$. Note that the dot product of two vectors is a number, not another vector. Also note that you can calculate the dot product only if the two vectors have the same number of components.

Let $A(i, j)$ be the entry in matrix A in the i th row and the j th column. For example $A(3, 2)$ is the entry in matrix A in the 3rd row and the 2nd column. Suppose A , B , and C are matrices, and $AB = C$. The entries of C are calculated as follows:

$$C(i, j) = (\text{row } i \text{ of } A) \cdot (\text{column } j \text{ of } B)$$

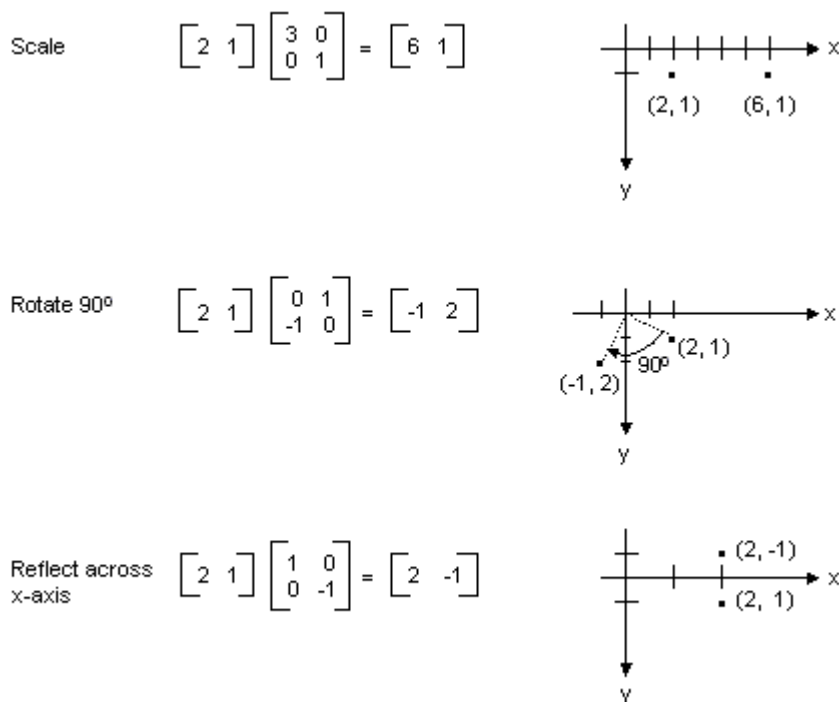
The following illustration shows several examples of matrix multiplication.

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

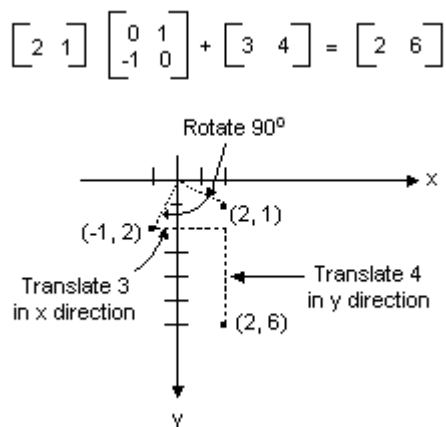
$$\begin{bmatrix} 1 & 3 & 2 \end{bmatrix}_{1 \times 3} \begin{bmatrix} 1 & 0 \\ 2 & 4 \\ 5 & 1 \end{bmatrix}_{3 \times 2} = \begin{bmatrix} 17 & 14 \end{bmatrix}_{1 \times 2}$$

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

If you think of a point in the plane as a 1×2 matrix, you can transform that point by multiplying it by a 2×2 matrix. The following illustration shows several transformations applied to the point $(2, 1)$.



All the transformations shown in the previous figure are linear transformations. Certain other transformations, such as translation, are not linear, and cannot be expressed as multiplication by a 2×2 matrix. Suppose you want to start with the point $(2, 1)$, rotate it 90 degrees, translate it 3 units in the x direction, and translate it 4 units in the y direction. You can accomplish this by performing a matrix multiplication followed by a matrix addition.

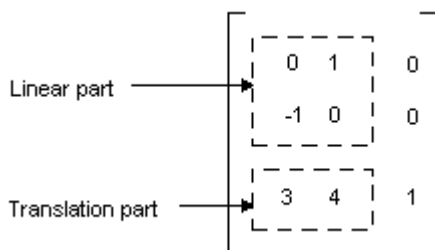


A linear transformation (multiplication by a 2×2 matrix) followed by a translation (addition of a 1×2 matrix) is called an affine transformation. An alternative to storing an affine transformation in a pair of matrices (one for the linear part and one for the translation) is to store the entire transformation in a 3×3 matrix. To make this work, a point in the plane must be stored in a 1×3 matrix with a dummy 3rd coordinate. The usual technique is to make all 3rd coordinates equal to 1 . For example, the point $(2, 1)$ is

represented by the matrix $\begin{bmatrix} 2 & 1 & 1 \end{bmatrix}$. The following illustration shows an affine transformation (rotate 90 degrees; translate 3 units in the x direction, 4 units in the y direction) expressed as multiplication by a single 3×3 matrix.

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

In the previous example, the point (2, 1) is mapped to the point (2, 6). Note that the third column of the 3×3 matrix contains the numbers 0, 0, 1. This will always be the case for the 3×3 matrix of an affine transformation. The important numbers are the six numbers in columns 1 and 2. The upper-left 2×2 portion of the matrix represents the linear part of the transformation, and the first two entries in the 3rd row represent the translation.



In Windows GDI+ you can store an affine transformation in a [Matrix](#) object. Because the third column of a matrix that represents an affine transformation is always (0, 0, 1), you specify only the six numbers in the first two columns when you construct a **Matrix** object. The statement `Matrix myMatrix(0.0f, 1.0f, -1.0f, 0.0f, 3.0f, 4.0f);` constructs the matrix shown in the previous figure.

=> Composite Transformations

A composite transformation is a sequence of transformations, one followed by the other. Consider the matrices and transformations in the following list:

- Matrix A Rotate 90 degrees
- Matrix B Scale by a factor of 2 in the x direction
- Matrix C Translate 3 units in the y direction

If you start with the point (2, 1) — represented by the matrix $\begin{bmatrix} 2 & 1 & 1 \end{bmatrix}$ — and multiply by A, then B, then C, the point (2,1) will undergo the three transformations in the order listed.

$$\begin{bmatrix} 2 & 1 & 1 \end{bmatrix} ABC = \begin{bmatrix} -2 & 5 & 1 \end{bmatrix}$$

Rather than store the three parts of the composite transformation in three separate matrices, you can multiply A, B, and C together to get a single 3×3 matrix that stores the entire composite transformation. Suppose $ABC = D$. Then a point multiplied by D gives the same result as a point multiplied by A, then B, then C.

$$\begin{bmatrix} 2 & 1 & 1 \end{bmatrix} D = \begin{bmatrix} -2 & 5 & 1 \end{bmatrix}$$

The following illustration shows the matrices A, B, C, and D.

$$\begin{matrix} \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} & \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} & \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 3 & 1 \end{bmatrix} & = & \begin{bmatrix} 0 & 1 & 0 \\ -2 & 0 & 0 \\ 0 & 3 & 1 \end{bmatrix} \\ A & B & C & & D \end{matrix}$$

The fact that the matrix of a composite transformation can be formed by multiplying the individual transformation matrices means that any sequence of affine transformations can be stored in a single [Matrix](#) object.

Note The order of a composite transformation is important. In general, rotate, then scale, then translate is not the same as scale, then rotate, then translate. Similarly, the order of matrix multiplication is important. In general, ABC is not the same as BAC.

The [Matrix](#) class provides several methods for building a composite transformation: [Matrix::Multiply](#), [Matrix::Rotate](#), [Matrix::RotateAt](#), [Matrix::Scale](#), [Matrix::Shear](#), and [Matrix::Translate](#). The following example creates the matrix of a composite transformation that first rotates 30 degrees, then scales by a factor of 2 in the y direction, and then translates 5 units in the x direction.

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

The following illustration shows the matrix.

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

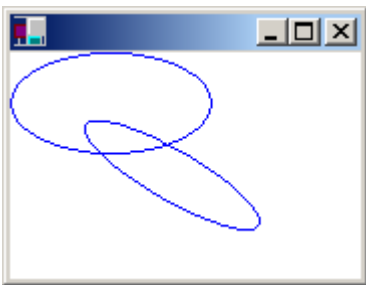
1.5.3) Global and Local Transformations

1000: A global transformation is a transformation that applies to every item drawn by a given [Graphics](#) object. To create a global transformation, construct a **Graphics** object, and then call its [Graphics::SetTransform](#) method. The **Graphics::SetTransform** method manipulates a [Matrix](#) object that is associated with the **Graphics** object. The transformation stored in that **Matrix** object is called the *world transformation*. The world transformation can be a simple affine transformation or a complex sequence of affine transformations, but regardless of its complexity, the world transformation is stored in a single **Matrix** object.

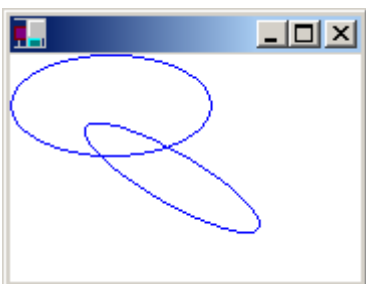
The [Graphics](#) class provides several methods for building up a composite world transformation: [Graphics::MultiplyTransform](#), [Graphics::RotateTransform](#), [Graphics::ScaleTransform](#), and [Graphics::TranslateTransform](#). The following example draws an ellipse twice: once before creating a world transformation and once after. The transformation first scales by a factor of 0.5 in the y direction, then translates 50 units in the x direction, and then rotates 30 degrees.

```
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);
```

The following illustration shows the original ellipse and the transformed ellipse.



The following illustration shows the matrices involved in the transformation.



Note In the previous example, the ellipse is rotated about the origin of the coordinate system, 1000: which is at the upper-left corner of the client area. This produces a different result than rotating the ellipse about its own center.

A local transformation is a transformation that applies to a specific item to be drawn. For example, a [GraphicsPath](#) object has a [GraphicsPath::Transform](#) method that allows you to transform the data points of that path. The following example draws a rectangle with no transformation and a path with a rotation transformation. (Assume that there is no world transformation.)

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

You can combine the world transformation with local transformations to achieve a variety of results. For example, you can use the world transformation to revise the coordinate system and use local transformations to rotate and scale objects drawn on the new coordinate system.

Suppose you want a coordinate system that has its origin 200 pixels from the left edge of the client area and 150 pixels from the top of the client area. Furthermore, assume that you want the unit of measure to be the pixel, with the x-axis pointing to the right and the y-axis pointing up. The default coordinate system has the y-axis pointing down, so you need to perform a reflection across the horizontal axis. The following illustration shows the matrix of such a reflection.

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

Next, assume you need to perform a translation 200 units to the right and 150 units down.

The following example establishes the coordinate system just described by setting the world transformation of a [Graphics](#) object.

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

The following code (placed after the code in the preceding example) creates a path that consists of a single rectangle with its lower-left corner at the origin of the new coordinate system. The rectangle is filled once with no local transformation and once with a local transformation. The local transformation consists of a horizontal scaling by a factor of 2 followed by a 30-degree rotation.

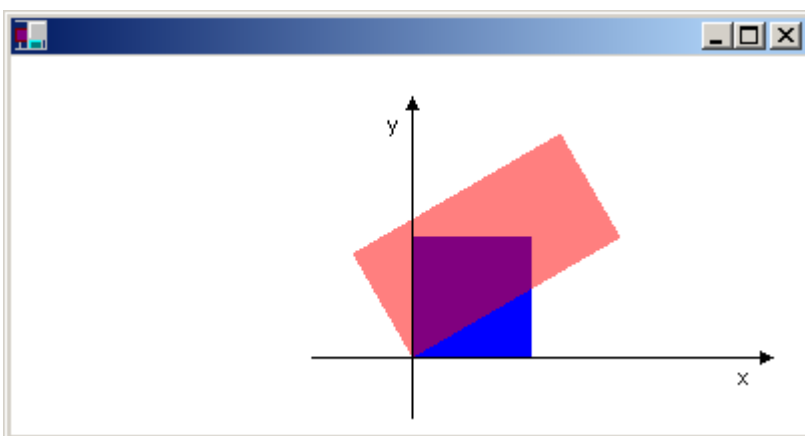
```
// Create the path.
GraphicsPath myGraphicsPath;
Rect myRect(0, 0, 60, 60);
myGraphicsPath.AddRectangle(myRect);

// 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);
```

The following illustration shows the new coordinate system and the two rectangles.



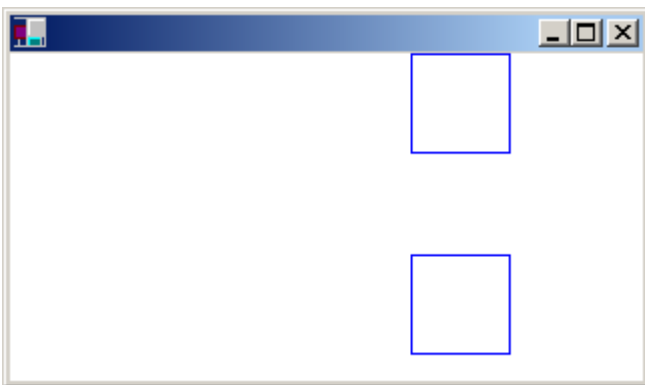
1.6) Graphics Containers

Graphics state — clipping region, transformations, and quality settings — is stored in a [Graphics](#) object. Windows GDI+ allows you to temporarily replace or augment part of the state in a **Graphics** object by using a container. You start a container by calling the [Graphics::BeginContainer](#) method of a **Graphics** object, and you end a container by calling the [Graphics::EndContainer](#) method. In between **Graphics::BeginContainer** and **Graphics::EndContainer**, any state changes you make to the **Graphics** object belong to the container and do not overwrite the existing state of the **Graphics** object.

The following example creates a container within a [Graphics](#) object. The world transformation of the **Graphics** object is a translation 200 units to the right, and the world transformation of the container is a translation 100 units down.

```
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);
```

Note that in the previous example, the statement `myGraphics.DrawRectangle(&myPen, 0, 0, 50, 50)` made in between the calls to [Graphics::BeginContainer](#) and [Graphics::EndContainer](#) produces a different rectangle than the same statement made after the call to **Graphics::EndContainer**. Only the horizontal translation applies to the **DrawRectangle** call made outside of the container. Both transformations — the horizontal translation of 200 units and the vertical translation of 100 units — apply to the [Graphics::DrawRectangle](#) call made inside the container. The following illustration shows the two rectangles.



Containers can be nested within containers. The following example creates a container within a [Graphics](#) object and another container within the first container. The world transformation of the **Graphics** object is a translation 100 units in the x direction and 80 units in the y direction. The world transformation of the first container is a 30-degree rotation. The world transformation of the second container is a scaling by a factor of 2 in the x direction. A call to the [Graphics::DrawEllipse](#) method is made inside the second container.

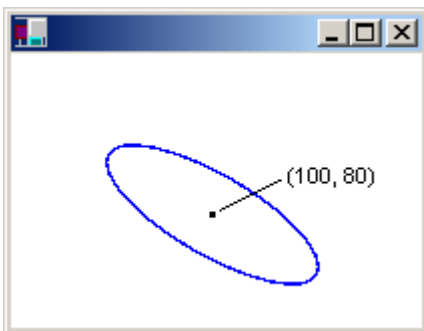
```
myGraphics.TranslateTransform(100.0f, 80.0f, MatrixOrderAppend);

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);
```

The following illustration shows the ellipse.



Note that all three transformations apply to the [Graphics::DrawEllipse](#) call made in the second (innermost) container. Also note the order of the transformations: first scale, then rotate, then translate. The innermost transformation is applied first, and the outermost transformation is applied last.

Any property of a [Graphics](#) object can be set inside a container (in between calls to [Graphics::BeginContainer](#) and [Graphics::EndContainer](#)). For example, a clipping region can be set inside a container. Any drawing done inside the container will be restricted to the clipping region of that container and will also be restricted to the clipping regions of any outer containers and the clipping region of the **Graphics** object itself.

The properties discussed so far — the world transformation and the clipping region — are combined by nested containers. Other properties are temporarily replaced by a nested container. For example, if you set the smoothing mode to `SmoothingModeAntiAlias` within a container, any drawing methods called

inside that container will use the antialias smoothing mode, but drawing methods called after [Graphics::EndContainer](#) will use the smoothing mode that was in place before the call to [Graphics::BeginContainer](#).

For another example of combining the world transformations of a [Graphics](#) object and a container, suppose you want to draw an eye and place it at various locations on a sequence of faces. The following example draws an eye centered at the origin of the coordinate system.

```
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);
}

```

The following illustration shows the eye and the coordinate axes.



The DrawEye function, defined in the previous example receives the address of a [Graphics](#) object and immediately creates a container within that **Graphics** object. This container insulates any code that calls the DrawEye function from property settings made during the execution of the DrawEye function. For example, code in the DrawEye function sets the clipping region of the **Graphics** object, but when DrawEye returns control to the calling routine, the clipping region will be as it was before the call to DrawEye.

The following example draws three ellipses (faces), each with an eye inside.

```

// 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);

```

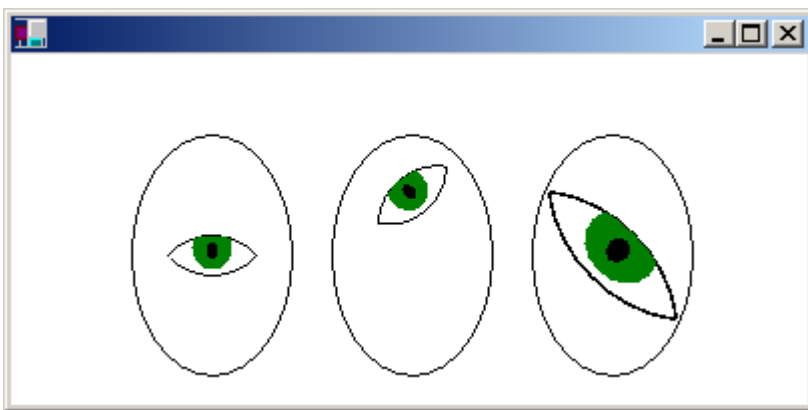
```

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);

```

The following illustration shows the three ellipses.



In the previous example, all ellipses are drawn with the call `DrawEllipse(&myBlackPen, -40, -60, 80, 120)`, which draws an ellipse centered at the origin of the coordinate system. The ellipses are moved away from the upper-left corner of the client area by setting the world transformation of the [Graphics](#) object. The statement causes the first ellipse to be centered at (100, 100). The statement causes the center of the second ellipse to be 100 units to the right of the center of the first ellipse. Likewise, the center of the third ellipse is 100 units to the right of the center of the second ellipse.

The containers in the previous example are used to transform the eye relative to the center of a given ellipse. The first eye is drawn at the center of the ellipse with no transformation, so the `DrawEye` call is not inside a container. The second eye is rotated 40 degrees and drawn 30 units above the center of the ellipse, so the `DrawEye` function and the methods that set the transformation are called inside a container. The third eye is stretched and rotated and drawn at the center of the ellipse. As with the second eye, the `DrawEye` function and the methods that set the transformation are called inside a container.

2) Using GDI+

The following topics describe how to use the GDI+ API with the C++ programming language:

- [Getting Started](#)
 - [Using a Pen to Draw Lines and Shapes](#)
 - [Using a Brush to Fill Shapes](#)
 - [Using Images, Bitmaps, and Metafiles](#)
 - [Using Image Encoders and Decoders](#)
 - [Alpha Blending Lines and Fills](#)
 - [Using Text and Fonts](#)
 - [Constructing and Drawing Curves](#)
 - [Filling Shapes with a Gradient Brush](#)
 - [Constructing and Drawing Paths](#)
 - [Using Graphics Containers](#)
 - [Transformations](#)
 - [Using Regions](#)
 - [Recoloring](#)
 - [Printing](#)
-

2.1) Getting Started

This section shows how to get started using Windows GDI+ in a standard C++ Windows application. Drawing lines and strings are two of the simplest tasks you can perform in GDI+. The following topics discuss these two tasks:

[Drawing a Line](#)

[Drawing a String](#)

2.1.1) Drawing a Line

This topic demonstrates how to draw a line using GDI Plus.

To draw a line in Windows GDI+ you need a [Graphics](#) object, a [Pen](#) object, and a [Color](#) object. The **Graphics** object provides the [DrawLine Methods](#) method, and the **Pen** object holds attributes of the line,

such as color and width. The address of the **Pen** object is passed as an argument to the `DrawLine` `Methods` method.

The following program, which draws a line from (0, 0) to (200, 100), consists of three functions: **WinMain**, **WndProc**, and **OnPaint**. The **WinMain** and **WndProc** functions provide the fundamental code common to most Windows applications. There is no GDI+ code in the **WndProc** function. The **WinMain** function has a small amount of GDI+ code, namely the required calls to [GdiplusStartup](#) and [GdiplusShutdown](#). The GDI+ code that actually creates a [Graphics](#) object and draws a line is in the **OnPaint** function.

The **OnPaint** function receives a handle to a device context and passes that handle to a [Graphics](#) constructor. The argument passed to the [Pen](#) constructor is a reference to a [Color](#) object. The four numbers passed to the color constructor represent the alpha, red, green, and blue components of the color. The alpha component determines the transparency of the color; 0 is fully transparent and 255 is fully opaque. The four numbers passed to the [DrawLine](#) `Methods` method represent the starting point (0, 0) and the ending point (200, 100) of the line.

```
#include <stdafx.h>
#include <windows.h>
#include <objidl.h>
#include <gdiplus.h>
using namespace Gdiplus;
#pragma comment (lib, "Gdiplus.lib")

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

```

Note the call to [GdiplusStartup](#) in the **WinMain** function. The first parameter of the **GdiplusStartup** function is the address of a `ULONG_PTR`. **GdiplusStartup** fills that variable with a token that is later passed to the [GdiplusShutdown](#) function. The second parameter of the **GdiplusStartup** function is the address of a [GdiplusStartupInput](#) structure. The preceding code relies on the default **GdiplusStartupInput** constructor to set the structure members appropriately.

2.1.2) Drawing a String

The topic [Drawing a Line](#) shows how to write a Windows application that uses Windows GDI+ to draw a line. To draw a string, replace the **OnPaint** function shown in that topic with the following **OnPaint** function:

```

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);
}

```


}

The previous code creates several GDI+ objects. The [Graphics](#) object provides the [DrawString Methods](#) method, which does the actual drawing. The [SolidBrush](#) object specifies the color of the string.

The [FontFamily](#) constructor receives a single, string argument that identifies the font family. The address of the **FontFamily** object is the first argument passed to the [Font](#) constructor. The second argument passed to the [Font](#) constructor specifies the font size, and the third argument specifies the style. The value **FontStyleRegular** is a member of the [FontStyle](#) enumeration, which is declared in `Gdiplusenums.h`. The last argument to the **Font** constructor indicates that the size of the font (24 in this case) is measured in pixels. The value **UnitPixel** is a member of the [Unit](#) enumeration.

The first argument passed to the [DrawString Methods](#) method is the address of a wide-character string. The second argument, `-1`, specifies that the string is null terminated. (If the string is not null terminated, the second argument should specify the number of wide characters in the string.) The third argument is the address of the [Font](#) object. The fourth argument is a reference to a [PointF](#) object that specifies the location where the string will be drawn. The last argument is the address of the [Brush](#) object, which specifies the color of the string.

2.2) Using a Pen to Draw Lines and Shapes

One of the arguments that you pass to such a drawing method is the address of a [Pen](#) object.

The following topics cover the use of pens in more detail:

- [Using a Pen to Draw Lines and Rectangles](#)
- [Setting Pen Width and Alignment](#)
- [Drawing a Line with Line Caps](#)
- [Joining Lines](#)
- [Drawing a Custom Dashed Line](#)
- [Drawing a Line Filled with a Texture](#)

The [Graphics](#) class provides a variety of drawing methods including those shown in the following list:

- [DrawLine Methods](#)
- [DrawRectangle Methods](#)
- [DrawEllipse Methods](#)
- [DrawArc Methods](#)
- [Graphics::DrawPath](#)
- [DrawCurve Methods](#)
- [DrawBezier Methods](#)

2.2.1) Using a Pen to Draw Lines and Rectangles

To draw lines and rectangles, you need a [Graphics](#) object and a [Pen](#) object. The **Graphics** object provides the [DrawLine](#) method, and the **Pen** object stores features of the line, such as color and width.

The following example draws a line from (20, 10) to (300, 100). Assume **graphics** is an existing [Graphics](#) object.

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

The first statement of code uses the [Pen](#) class constructor to create a black pen. The one argument passed to the **Pen** constructor is a [Color](#) object. The values used to construct the **Color** object — (255, 0,

0, 0) — correspond to the alpha, red, green, and blue components of the color. These values define an opaque black pen.

The following example draws a rectangle with its upper-left corner at (10, 10). The rectangle has a width of 100 and a height of 50. The second argument passed to the [Pen](#) constructor indicates that the pen width is 5 pixels.

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

When the rectangle is drawn, the pen is centered on the rectangle's boundary. Because the pen width is 5, the sides of the rectangle are drawn 5 pixels wide, such that 1 pixel is drawn on the boundary itself, 2 pixels are drawn on the inside, and 2 pixels are drawn on the outside. For more details on pen alignment, see [Setting Pen Width and Alignment](#).

The following illustration shows the resulting rectangle. The dotted lines show where the rectangle would have been drawn if the pen width had been one pixel. The enlarged view of the upper-left corner of the rectangle shows that the thick black lines are centered on those dotted lines.



2.2.2) Setting Pen Width and Alignment

When you create a [Pen](#) object, you can supply the pen width as one of the arguments to the constructor. You can also change the pen width by using the [Pen::SetWidth](#) method.

A theoretical line has a width of zero. When you draw a line, the pixels are centered on the theoretical line. The following example draws a specified line twice: once with a black pen of width 1 and once with a green pen of width 10.

```
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);
```

The following illustration shows the output of the preceding code. The green pixels and the black pixels are centered on the theoretical line.



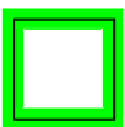
The following example draws a specified rectangle twice: once with a black pen of width 1 and once with a green pen of width 10. The code passes the value **PenAlignmentCenter** (an element of the [PenAlignment](#) enumeration) to the [Pen::SetAlignment](#) method to specify that the pixels drawn with the green pen are centered on the boundary of the rectangle.

```
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);
```

The following illustration shows the output of the preceding code. The green pixels are centered on the theoretical rectangle, which is represented by the black pixels.



You can change the green pen's alignment by modifying the third statement in the preceding example as follows:

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

Now the pixels in the wide green line appear on the inside of the rectangle as shown in the following illustration.



2.2.3) Drawing a Line with Line Caps

You can draw the start or end of a line in one of several shapes called line caps. Windows GDI+ supports several line caps, such as round, square, diamond, and arrowhead.

You can specify line caps for the start of a line (start cap), the end of a line (end cap), or the dashes of a dashed line (dash cap).

The following example draws a line with an arrowhead at one end and a round cap at the other end:

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

The following illustration shows the resulting line.



LineCapArrowAnchor and **LineCapRoundAnchor** are elements of the [LineCap](#) enumeration.

2.2.4) Joining Lines

A line join is the common area that is formed by two lines whose ends meet or overlap. Windows GDI+ provides four line join styles: miter, bevel, round, and miter clipped. Line join style is a property of the [Pen](#) class. When you specify a line join style for a pen and then use that pen to draw a path, the specified join style is applied to all the connected lines in the path.

You can specify the line join style by using the [Pen::SetLineJoin](#) method of the [Pen](#) class. The following example demonstrates a beveled line join between a horizontal line and a vertical line:

```
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);
```

The following illustration shows the resulting beveled line join.



In the preceding example, the value (**LineJoinBevel**) passed to the [Pen::SetLineJoin](#) method is an element of the [LineJoin](#) enumeration.

2.2.5) Drawing a Custom Dashed Line

Windows GDI+ provides several dash styles that are listed in the [DashStyle](#) enumeration. If those standard dash styles don't suit your needs, you can create a custom dash pattern.

To draw a custom dashed line, put the lengths of the dashes and spaces in an array and pass the address of the array as an argument to the [Pen::SetDashPattern](#) method of a [Pen](#) object. The following example draws a custom dashed line based on the array {5, 2, 15, 4}. If you multiply the elements of the array by

the pen width of 5, you get {25, 10, 75, 20}. The displayed dashes alternate in length between 25 and 75, and the spaces alternate in length between 10 and 20.

```
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));
```

The following illustration shows the resulting dashed line. Note that the final dash has to be shorter than 25 units so that the line can end at (405, 5).



2.2.6) Drawing a Line Filled with a Texture

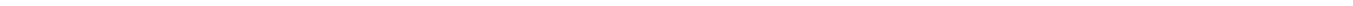
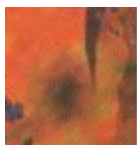
Instead of drawing a line or curve with a solid color, you can draw with a texture. To draw lines and curves with a texture, create a [TextureBrush](#) object, and pass the address of that **TextureBrush** object to a [Pen](#) constructor. The image associated with the texture brush is used to tile the plane (invisibly), and when the pen draws a line or curve, the stroke of the pen uncovers certain pixels of the tiled texture.

The following example creates an [Image](#) object from the file Texture1.jpg. That image is used to construct a [TextureBrush](#) object, and the **TextureBrush** object is used to construct a [Pen](#) object. The call to [Graphics::DrawImage](#) draws the image with its upper-left corner at (0, 0). The call to [Graphics::DrawEllipse](#) uses the **Pen** object to draw a textured ellipse.

```
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);
```

The following illustration shows the image and the textured ellipse.



2.3) Using a Brush to Fill Shapes

A Windows GDI+ [Brush](#) object is used to fill the interior of a closed shape. GDI+ defines several fill styles: solid color, hatch pattern, image texture, and color gradient.

The following topics cover the use of brushes in more detail:

- [Filling a Shape with a Solid Color](#)
- [Filling a Shape with a Hatch Pattern](#)
- [Filling a Shape with an Image Texture](#)
- [Tiling a Shape with an Image](#)
- [Filling a Shape with a Color Gradient](#)

2.3.1) Filling a Shape with a Solid Color

To fill a shape with a solid color, create a [SolidBrush](#) object, and then pass the address of that **SolidBrush** object as an argument to one of the fill methods of the [Graphics](#) class. The following example shows how to fill an ellipse with the color red:

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

In the preceding example, the [SolidBrush](#) constructor takes a [Color](#) object reference as its only argument. The values used by the **Color** constructor represent the alpha, red, green, and blue components of the color. Each of these values must be in the range 0 through 255. The first 255 indicates that the color is fully opaque, and the second 255 indicates that the red component is at full intensity. The two zeros indicate that the green and blue components both have an intensity of 0.

The four numbers (0, 0, 100, 60) passed to the [Graphics::FillEllipse](#) method specify the location and size of the bounding rectangle for the ellipse. The rectangle has an upper-left corner of (0, 0), a width of 100, and a height of 60.

2.3.2) Filling a Shape with a Hatch Pattern

A hatch pattern is made from two colors: one for the background and one for the lines that form the pattern over the background. To fill a closed shape with a hatch pattern, use a [HatchBrush](#) object. The following example demonstrates how to fill an ellipse with a hatch pattern:

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

The following illustration shows the filled ellipse.



The [HatchBrush](#) constructor takes three arguments: the hatch style, the color of the hatch line, and the color of the background. The hatch style argument can be any element of the [HatchStyle](#) enumeration. There are more than fifty elements in the **HatchStyle** enumeration; a few of those elements are shown in the following list:

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

2.3.3) Filling a Shape with an Image Texture

You can fill a closed shape with a texture by using the [Image](#) class and the [TextureBrush](#) class.

The following example fills an ellipse with an image. The code constructs an [Image](#) object, and then passes the address of that **Image** object as an argument to a [TextureBrush](#) constructor. The third code statement scales the image, and the fourth statement fills the ellipse with repeated copies of the scaled image:

```
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));
```

In the preceding code example, the [TextureBrush::SetTransform](#) method sets the transformation that is applied to the image before it is drawn. Assume that the original image has a width of 640 pixels and a height of 480 pixels. The transform shrinks the image to 75 × 75, by setting the horizontal and vertical scaling values.

Note In the preceding example, the image size is 75 × 75, and the ellipse size is 150 × 250. Because the image is smaller than the ellipse it is filling, the ellipse is tiled with the image. Tiling means that the image is repeated horizontally and vertically until the boundary of the shape is reached. For more information on tiling, see [Tiling a Shape with an Image](#).

2.3.4) Tiling a Shape with an Image

Just as tiles can be placed next to each other to cover a floor, rectangular images can be placed next to each other to fill (tile) a shape. To tile the interior of a shape, use a texture brush. When you construct a [TextureBrush](#) object, one of the arguments you pass to the constructor is the address of an [Image](#) object. When you use the texture brush to paint the interior of a shape, the shape is filled with repeated copies of this image.

The wrap mode property of the [TextureBrush](#) object determines how the image is oriented as it is repeated in a rectangular grid. You can make all the tiles in the grid have the same orientation, or you can make the image flip from one grid position to the next. The flipping can be horizontal, vertical, or both. The following examples demonstrate tiling with different types of flipping.

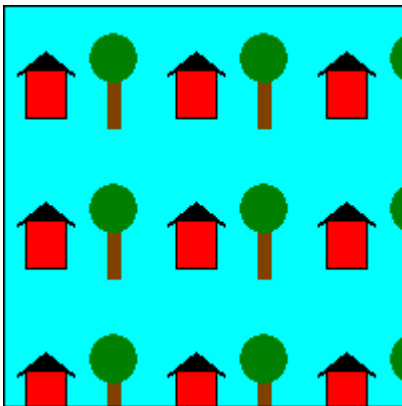
=> Tiling an Image

This example uses the following 75 × 75 image to tile a 200 × 200 rectangle:



```
Image image(L"HouseAndTree.png");
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));
```

The following illustration shows how the rectangle is tiled with the image. Note that all tiles have the same orientation; there is no flipping.

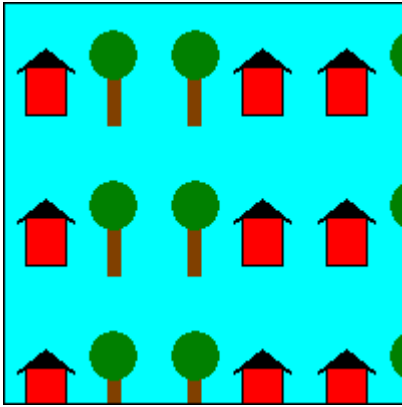


=> Flipping an Image Horizontally While Tiling

This example uses a 75 × 75 image to fill a 200 × 200 rectangle. The wrap mode is set to flip the image horizontally.

```
Image image(L"HouseAndTree.png");
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));
```

The following illustration shows how the rectangle is tiled with the image. Note that as you move from one tile to the next in a given row, the image is flipped horizontally.

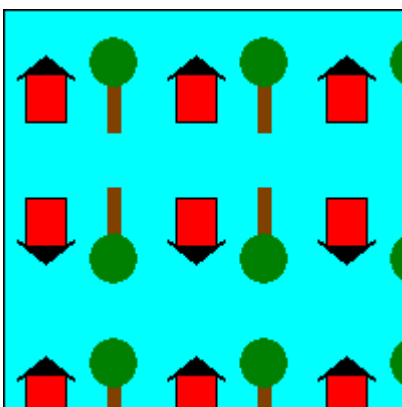


=> Flipping an Image Vertically While Tiling

This example uses a 75 ×75 image to fill a 200 ×200 rectangle. The wrap mode is set to flip the image vertically.

```
Image image(L"HouseAndTree.png");
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));
```

The following illustration shows how the rectangle is tiled with the image. Note that as you move from one tile to the next in a given column, the image is flipped vertically.

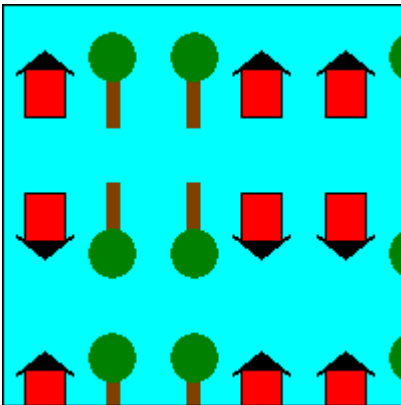


Flipping an Image Horizontally and Vertically While Tiling

This example uses a 75 ×75 image to tile a 200 ×200 rectangle. The wrap mode is set to flip the image both horizontally and vertically.

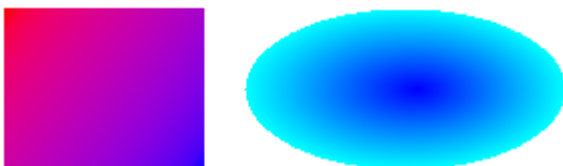
```
Image image(L"HouseAndTree.png");
TextureBrush tBrush(&image);
Pen blackPen(Color(255, 0, 0, 0));
stat = tBrush.SetWrapMode(WrapModeTileFlipXY);
stat = graphics.FillRectangle(&tBrush, Rect(0, 0, 200, 200));
stat = graphics.DrawRectangle(&blackPen, Rect(0, 0, 200, 200));
```

The following illustration shows how the rectangle is tiled by the image. Note that as you move from one tile to the next in a given row, the image is flipped horizontally, and as you move from one tile to the next in a given column, the image is flipped vertically.



2.3.5) Filling a Shape with a Color Gradient

You can fill a shape with a gradually changing color by using a gradient brush. Windows GDI+ provides linear gradient brushes and path gradient brushes. The following illustration shows a rectangle filled with a linear gradient brush and an ellipse filled with a path gradient brush.



For more information about gradient brushes, see [Filling Shapes With a Gradient Brush](#).

2.4) Using Images, Bitmaps, and Metafiles

Windows GDI+ provides the [Image](#) class for working with raster images (bitmaps) and vector images (metafiles). The [Bitmap](#) class and the [Metafile](#) class both inherit from the **Image** class. The **Bitmap** class expands on the capabilities of the **Image** class by providing additional methods for loading, saving, and manipulating raster images. The **Metafile** class expands on the capabilities of the **Image** class by providing additional methods for recording and examining vector images.

The following topics cover the [Image](#), [Bitmap](#), and [Metafile](#) classes in more detail:

- [Loading and Displaying Bitmaps](#)
- [Loading and Displaying Metafiles](#)
- [Recording Metafiles](#)
- [Cropping and Scaling Images](#)
- [Rotating, Reflecting, and Skewing Images](#)
- [Using Interpolation Mode to Control Image Quality During Scaling](#)
- [Creating Thumbnail Images](#)
- [Using a Cached Bitmap to Improve Performance](#)
- [Improving Performance by Avoiding Automatic Scaling](#)
- [Reading and Writing Metadata](#)

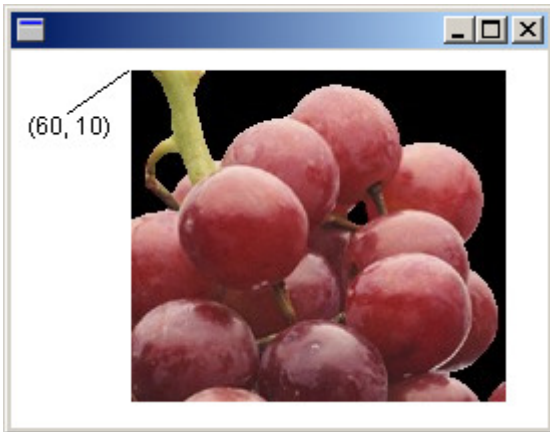
2.4.1) Loading and Displaying Bitmaps

To display a raster image (bitmap) on the screen, you need an [Image](#) object and a [Graphics](#) object. Pass the name of a file (or a pointer to a stream) to an **Image** constructor. After you have created an **Image** object, pass the address of that **Image** object to the **DrawImage** method of a **Graphics** object.

The following example creates an [Image](#) object from a JPEG file and then draws the image with its upper-left corner at (60, 10):

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

The following illustration shows the image drawn at the specified location.



The [Image](#) class provides basic methods for loading and displaying raster images and vector images. The [Bitmap](#) class, which inherits from the **Image** class, provides more specialized methods for loading, displaying, and manipulating raster images. For example, you can construct a **Bitmap** object from an icon handle (HICON).

The following example obtains a handle to an icon and then uses that handle to construct a [Bitmap](#) object. The code displays the icon by passing the address of the **Bitmap** object to the **DrawImage** method of a [Graphics](#) object.

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

2.4.2) Loading and Displaying Metafiles

The [Image](#) class provides basic methods for loading and displaying raster images and vector images. The [Metafile](#) class, which inherits from the **Image** class, provides more specialized methods for recording, displaying, and examining vector images.

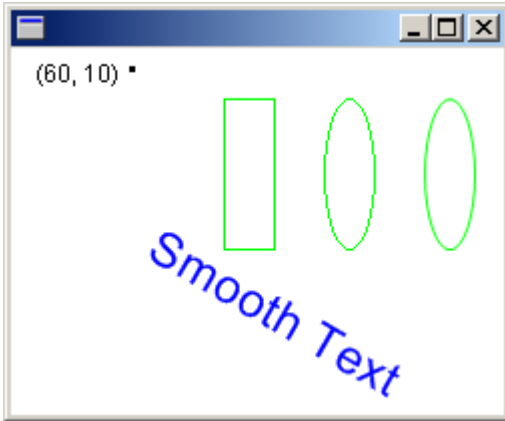
To display a vector image (metafile) on the screen, you need an [Image](#) object and a [Graphics](#) object. Pass the name of a file (or a pointer to a stream) to an **Image** constructor. After you have created an **Image** object, pass the address of that **Image** object to the **DrawImage** method of a **Graphics** object.

The following example creates an [Image](#) object from an EMF (enhanced metafile) file and then draws the image with its upper-left corner at (60, 10):

```
Image image(L"SampleMetafile.emf");
```

```
graphics.DrawImage(&image, 60, 10);
```

The following illustration shows the image drawn at the specified location.



2.4.3) Recording Metafiles

The [Metafile](#) class, which inherits from the [Image](#) class, allows you to record a sequence of drawing commands. The recorded commands can be stored in memory, saved to a file, or saved to a stream. Metafiles can contain vector graphics, raster images, and text.

The following example creates a [Metafile](#) object. The code uses the **Metafile** object to record a sequence of graphics commands and then saves the recorded commands in a file named SampleMetafile.emf. Note that the **Metafile** constructor receives a device context handle, and the [Graphics](#) constructor receives the address of the **Metafile** object. The recording stops (and the recorded commands are saved to the file) when the **Graphics** object goes out of scope. The last two lines of code display the metafile by creating a new **Graphics** object and passing the address of the **Metafile** object to the **DrawImage** method of that **Graphics** object. Note that the code uses the same **Metafile** object to record and to display (play back) the metafile.

```
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);

```

Note To record a metafile, you must construct a [Graphics](#) object based on a [Metafile](#) object. The recording of the metafile ends when that **Graphics** object is deleted or goes out of scope.

A metafile contains its own graphics state, which is defined by the [Graphics](#) object used to record the metafile. Any properties of the **Graphics** object (clip region, world transformation, smoothing mode, and the like) that you set while recording the metafile will be stored in the metafile. When you display the metafile, the drawing will be done according to those stored properties.

In the following example, assume that the smoothing mode was set to `SmoothingModeNormal` during the recording of the metafile. Even though the smoothing mode of the [Graphics](#) object used for playback is set to `SmoothingModeHighQuality`, the metafile will be played according to the `SmoothingModeNormal` setting. It is the smoothing mode set during the recording that is important, not the smoothing mode set prior to playback.

```

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

```

2.4.4) Cropping and Scaling Images

The [Graphics](#) class provides several **DrawImage** methods, some of which have source and destination rectangle parameters that you can use to crop and scale images.

The following example constructs an [Image](#) object from the file Apple.gif. The code draws the entire apple image in its original size. The code then calls the **DrawImage** method of a [Graphics](#) object to draw a portion of the apple image in a destination rectangle that is larger than the original apple image.

The **DrawImage** method determines which portion of the apple to draw by looking at the source rectangle, which is specified by the third, fourth, fifth, and sixth arguments. In this case, the apple is cropped to 75 percent of its width and 75 percent of its height.

The **DrawImage** method determines where to draw the cropped apple and how big to make the cropped apple by looking at the destination rectangle, which is specified by the second argument. In this case, the destination rectangle is 30 percent wider and 30 percent taller than the original image.

```
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);
```

The following illustration shows the original apple and the scaled, cropped apple.



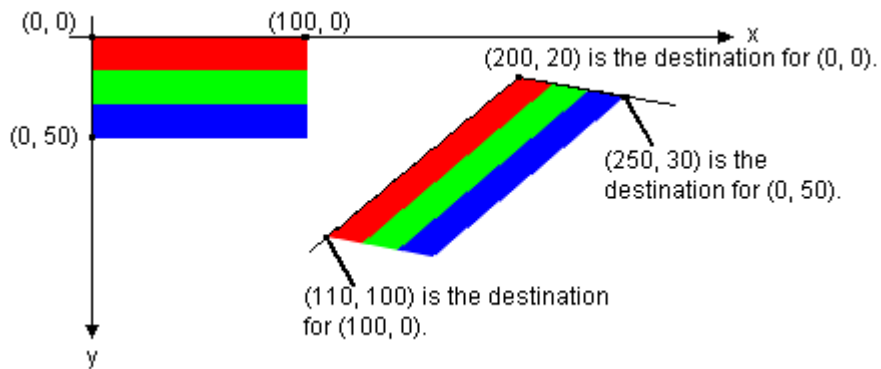
2.4.5) Rotating, Reflecting, and Skewing Images

You can rotate, reflect, and skew an image by specifying destination points for the upper-left, upper-right, and lower-left corners of the original image. The three destination points determine an affine transformation that maps the original rectangular image to a parallelogram. (The lower-right corner of the original image is mapped to the fourth corner of the parallelogram, which is calculated from the three specified destination points.)

For example, suppose the original image is a rectangle with upper-left corner at (0, 0), upper-right corner at (100, 0), and lower-left corner at (0, 50). Now suppose we map those three points to destination points as follows.

Original point	Destination point
Upper-left (0, 0)	(200, 20)
Upper-right (100, 0)	(110, 100)
Lower-left (0, 50)	(250, 30)

The following illustration shows the original image and the image mapped to the parallelogram. The original image has been skewed, reflected, rotated, and translated. The x-axis along the top edge of the original image is mapped to the line that runs through (200, 20) and (110, 100). The y-axis along the left edge of the original image is mapped to the line that runs through (200, 20) and (250, 30).



The following example produces the images shown in the preceding illustration.

```
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);
```

The following illustration shows a similar transformation applied to a photographic image.



The following illustration shows a similar transformation applied to a metafile.



2.4.6) Using Interpolation Mode to Control Image Quality During Scaling

The interpolation mode of a [Graphics](#) object influences the way Windows GDI+ scales (stretches and shrinks) images. The [InterpolationMode](#) enumeration in `Gdiplusenums.h` defines several interpolation modes, some of which are shown in the following list:

- `InterpolationModeNearestNeighbor`
- `InterpolationModeBilinear`
- `InterpolationModeHighQualityBilinear`
- `InterpolationModeBicubic`
- `InterpolationModeHighQualityBicubic`

To stretch an image, each pixel in the original image must be mapped to a group of pixels in the larger image. To shrink an image, groups of pixels in the original image must be mapped to single pixels in the smaller image. The effectiveness of the algorithms that perform these mappings determines the quality of a scaled image. Algorithms that produce higher-quality scaled images tend to require more processing time. In the preceding list, `InterpolationModeNearestNeighbor` is the lowest-quality mode and `InterpolationModeHighQualityBicubic` is the highest-quality mode.

To set the interpolation mode, pass one of the members of the [InterpolationMode](#) enumeration to the **SetInterpolationMode** method of a [Graphics](#) object.

The following example draws an image and then shrinks the image with three different interpolation modes:

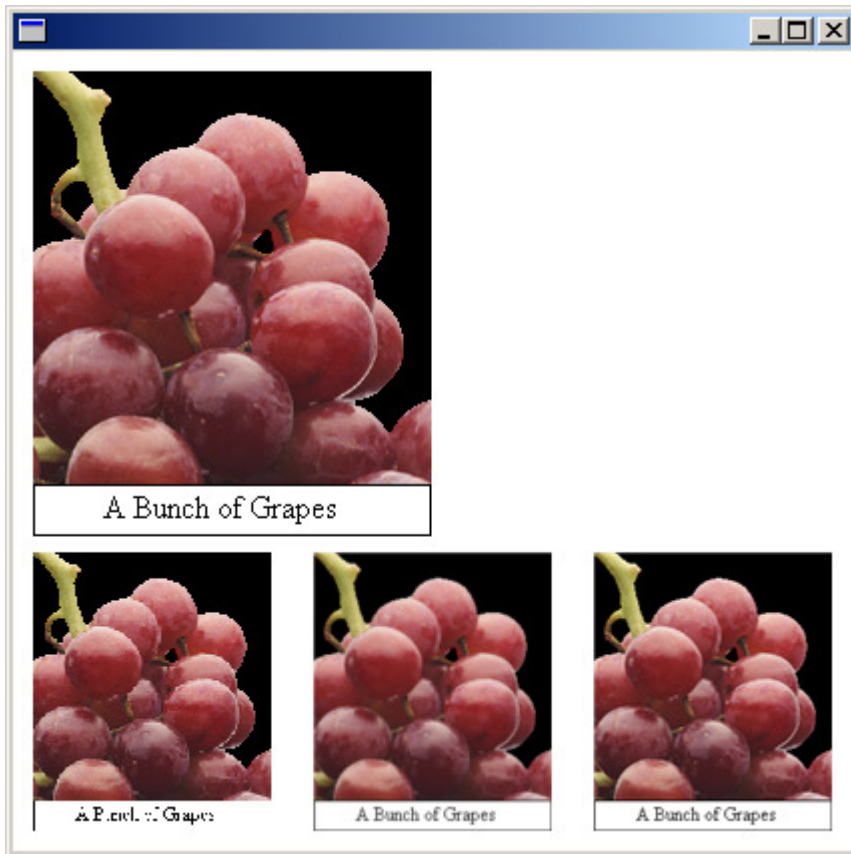
```
Image image(L"GrapeBunch.bmp");
UINT width = image.GetWidth();
UINT height = image.GetHeight();
// Draw the image with no shrinking or stretching.
graphics.DrawImage(
    &image,
    Rect(10, 10, width, height), // destination rectangle
```

```

    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);

```

The following illustration shows the original image and the three smaller images.



2.4.7) Creating Thumbnail Images

A thumbnail image is a small version of an image. You can create a thumbnail image by calling the **GetThumbnailImage** method of an [Image](#) object.

The following example constructs an [Image](#) object from the file Compass.bmp. The original image has a width of 640 pixels and a height of 479 pixels. The code creates a thumbnail image that has a width of 100 pixels and a height of 100 pixels.

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

The following illustration shows the thumbnail image.



2.4.8) Using a *Cached Bitmap* to Improve Performance

[Image](#) and [Bitmap](#) objects store images in a device-independent format. A [CachedBitmap](#) object stores an image in the format of the current display device. Rendering an image stored in a **CachedBitmap** object is fast because no processing time is spent converting the image to the format required by the display device.

The following example creates a [Bitmap](#) object and a [CachedBitmap](#) object from the file Texture.jpg. The **Bitmap** and the **CachedBitmap** are each drawn 30,000 times. If you run the code, you will see that the **CachedBitmap** images are drawn substantially faster than the **Bitmap** images.

```
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 );
```

Note A [CachedBitmap](#) object matches the format of the display device at the time the **CachedBitmap** object was constructed. If the user of your program changes the display settings, your code should construct a new **CachedBitmap** object. The **DrawImage** method will fail if you pass it a **CachedBitmap** object that was created prior to a change in the display format.

2.4.9) Improving Performance by Avoiding Automatic Scaling

If you pass only the upper-left corner of an image to the **DrawImage** method, Windows GDI+ might scale the image, which would decrease performance.

The following call to the **DrawImage** method specifies an upper-left corner of (50, 30) but does not specify a destination rectangle:

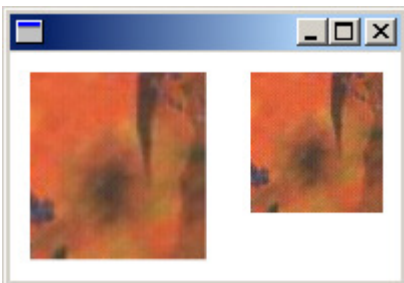
```
graphics.DrawImage(&image, 50, 30); // upper-left corner at (50, 30)
```

Although this is the easiest version of the **DrawImage** method in terms of the number of required arguments, it is not necessarily the most efficient. If the number of dots per inch on the current display device is different than the number of dots per inch on the device where the image was created, GDI+ scales the image so that its physical size on the current display device is as close as possible to its physical size on the device where it was created.

If you want to prevent such scaling, pass the width and height of a destination rectangle to the **DrawImage** method. The following example draws the same image twice. In the first case, the width and height of the destination rectangle are not specified, and the image is automatically scaled. In the second case, the width and height (measured in pixels) of the destination rectangle are specified to be the same as the width and height of the original image.

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

The following illustration shows the image rendered twice.



2.4.10) Reading and Writing Metadata

Some image files contain metadata that you can read to determine features of the image. For example, a digital photograph might contain metadata that you can read to determine the make and model of the camera used to capture the image. With Windows GDI+, you can read existing metadata, and you can also write new metadata to image files.

GDI+ provides a uniform way of storing and retrieving metadata from image files in various formats. In GDI+, a piece of metadata is called a *property item*. You can store and retrieve metadata by calling the **SetPropertyItem** and **GetPropertyItem** methods of the [Image](#) class, and you don't have to be concerned about the details of how a particular file format stores that metadata.

GDI+ currently supports metadata for the TIFF, JPEG, Exif, and PNG file formats. The Exif format, which specifies how to store images captured by digital still cameras, is built on top of the TIFF and JPEG formats. Exif uses the TIFF format for uncompressed pixel data and the JPEG format for compressed pixel data.

GDI+ defines a set of property tags that identify property items. Certain tags are general-purpose; that is, they are supported by all of the file formats mentioned in the preceding paragraph. Other tags are special-purpose and apply only to certain formats. If you try to save a property item to a file that does not support that property item, GDI+ ignores the request. More specifically, the [Image::SetPropertyItem](#) method returns `PropertyNotSupported`.

You can determine the property items that are stored in an image file by calling [Image::GetPropertyIdList](#). If you try to retrieve a property item that is not in the file, GDI+ ignores the request. More specifically, the [Image::GetPropertyItem](#) method returns `PropertyNotFound`.

=> Reading Metadata from a File

The following console application calls the **GetPropertySize** method of an [Image](#) object to determine how many pieces of metadata are in the file `FakePhoto.jpg`.

```
#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;
}

```

The preceding code, along with a particular file, FakePhoto.jpg, produced the following output:

```

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

```

GDI+ stores an individual piece of metadata in a [PropertyItem](#) object. You can call the **GetAllPropertyItems** method of the [Image](#) class to retrieve all the metadata from a file. The **GetAllPropertyItems** method returns an array of **PropertyItem** objects. Before you call **GetAllPropertyItems**, you must allocate a buffer large enough to receive that array. You can call the **GetPropertySize** method of the **Image** class to get the size (in bytes) of the required buffer.

A [PropertyItem](#) object has the following four public members:

id	A tag that identifies the metadata item. The values that can be assigned to id (PropertyTagImageTitle, PropertyTagEquipMake, PropertyTagExifExposureTime, and the like) are defined in Gdiplusimaging.h.
length	The length, in bytes, of the array of values pointed to by the value data member. Note that if the type data member is set to PropertyTagTypeASCII, then the length data member is the length of a null-terminated character string, including the NULL terminator.
type	The data type of the values in the array pointed to by the value data member. Constants (PropertyTagTypeByte, PropertyTagTypeASCII, and the like) that represent various data types are described in Image Property Tag Type Constants .
value	A pointer to an array of values.

The following console application reads and displays the seven pieces of metadata in the file FakePhoto.jpg. The main function relies on the helper function PropertyTypeFromWORD, which is shown following the main function.

```
#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;
}

```

The preceding console application produces the following output:

```

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
```

The preceding output shows a hexadecimal ID number for each property item. You can look up those ID numbers in [Image Property Tag Constants](#) and find out that they represent the following property tags.

Hexadecimal value	Property tag
0x0320	PropertyTagImageTitle
0x010f	PropertyTagEquipMake
0x0110	PropertyTagEquipModel
0x9003	PropertyTagExifDTOriginal
0x829a	PropertyTagExifExposureTime

0x5090 PropertyTagLuminanceTable

0x5091 PropertyTagChrominanceTable

The second (index 1) property item in the list has **id** PropertyTagEquipMake and **type** PropertyTagTypeASCII. The following example, which is a continuation of the previous console application, displays the value of that property item:

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

The preceding line of code produces the following output:

The equipment make is Northwind Traders.

The fifth (index 4) property item in the list has **id** PropertyTagExifExposureTime and **type** PropertyTagTypeRational. That data type (PropertyTagTypeRational) is a pair of **LONGs**. The following example, which is a continuation of the previous console application, displays those two **LONG** values as a fraction. That fraction, 1/125, is the exposure time measured in seconds.

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

The preceding code produces the following output:

The exposure time is 1/125.

=> Writing Metadata to a File

To write an item of metadata to an [Image](#) object, initialize a [PropertyItem](#) object and then pass the address of that **PropertyItem** object to the **SetPropertyItem** method of the **Image** object.

The following console application writes one item (the image title) of metadata to an [Image](#) object and then saves the image in the disk file FakePhoto2.jpg. The main function relies on the helper function GetEncoderClsid, which is shown in the topic [Retrieving the Class Identifier for an Encoder](#).

```
#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;
}
```

2.5) Using Image Encoders and Decoders

Windows GDI+ provides the [Image](#) class and the [Bitmap](#) class for storing images in memory and manipulating images in memory. GDI+ writes images to disk files with the help of image encoders and loads images from disk files with the help of image decoders. An encoder translates the data in an **Image** or **Bitmap** object into a designated disk file format. A decoder translates the data in a disk file to the format required by the **Image** and **Bitmap** objects. GDI+ has built-in encoders and decoders that support the following file types:

- BMP
- GIF
- JPEG
- PNG
- TIFF

GDI+ also has built-in decoders that support the following file types:

- WMF
- EMF
- ICON

The following topics discuss encoders and decoders in more detail:

- [Listing Installed Encoders](#)
- [Listing Installed Decoders](#)
- [Retrieving the Class Identifier for an Encoder](#)
- [Determining the Parameters Supported by an Encoder](#)
- [Converting a BMP Image to a PNG Image](#)
- [Setting JPEG Compression Level](#)
- [Transforming a JPEG Image Without Loss of Information](#)
- [Creating and Saving a Multiple-Frame Image](#)
- [Copying Individual Frames from a Multiple-Frame Image](#)

2.5.1) Listing Installed Encoders

GDI+ provides the [GetImageEncoders](#) function so that you can determine which image encoders are available on your computer. **GetImageEncoders** returns an array of [ImageCodecInfo](#) objects. Before you call **GetImageEncoders**, you must allocate a buffer large enough to receive that array. You can call [GetImageEncodersSize](#) to determine the size of the required buffer.

The following console application lists the available image encoders:

```

#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;
}

```

When you run the preceding console application, the output will be similar to the following:

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

2.5.2) Listing Installed Decoders

Windows GDI+ provides the [GetImageDecoders](#) function so that you can determine which image decoders are available on your computer. **GetImageDecoders** returns an array of [ImageCodecInfo](#) objects. Before you call **GetImageDecoders**, you must allocate a buffer large enough to receive that array. You can call [GetImageDecodersSize](#) to determine the size of the required buffer.

The following console application lists the available image decoders:

```
#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;
}

```

When you run the preceding console application, the output will be similar to the following:

```

image/bmp
image/jpeg
image/gif
image/x-emf
image/x-wmf
image/tiff
image/png
image/x-icon

```

2.5.3) Retrieving the Class Identifier for an Encoder

The function `GetEncoderClsid` in the following example receives the MIME type of an encoder and returns the class identifier (**CLSID**) of that encoder. The MIME types of the encoders built into Windows GDI+ are as follows:

- `image/bmp`
- `image/jpeg`
- `image/gif`
- `image/tiff`
- `image/png`

The function calls [GetImageEncoders](#) to get an array of [ImageCodecInfo](#) objects. If one of the **ImageCodecInfo** objects in that array represents the requested encoder, the function returns the index of the **ImageCodecInfo** object and copies the **CLSID** into the variable pointed to by **pClsid**. If the function fails, it returns `-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
}

```

The following console application calls the `GetEncoderClsid` function to determine the **CLSID** of the PNG encoder:

```

#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;
}

```



```
}
```

When you run the preceding console application, you get an output similar to the following:

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}.

2.5.4) Determining the Parameters Supported by an Encoder

The [Image](#) class provides the [Image::GetEncoderParameterList](#) method so that you can determine the parameters that are supported by a given image encoder. The **Image::GetEncoderParameterList** method returns an array of [EncoderParameter](#) objects. You must allocate a buffer to receive that array before you call **Image::GetEncoderParameterList**. You can call [Image::GetEncoderParameterListSize](#) to determine the size of the required buffer.

The following console application obtains the parameter list for the JPEG encoder. The main function relies on the helper function GetEncoderClsid, which is shown in [Retrieving the Class Identifier for an Encoder](#).

```
#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;
}

```

When you run the preceding console application, you get an output similar to the following:

```

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

```

Each of the [EncoderParameter](#) objects in the array has the following four public data members:

The following code is a continuation of the console application shown in the preceding example. The code looks at the second [EncoderParameter](#) object in the array returned by [Image::GetEncoderParameterList](#). The code calls [StringFromGUID2](#), which is a system function declared in `Objbase.h`, to convert the **Guid** member of the **EncoderParameter** object to a string.

```
// 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);
```

The preceding code produces the following output:

```
Parameter[1]
    The GUID is {1D5BE4B5-FA4A-452D-9CDD-5DB35105E7EB}.
    The value type is 6.
    The number of values is 1.
```

You can look up the GUID in Gdiplusimaging.h and find out that the category of this [EncoderParameter](#) object is EncoderQuality. You can use this category (EncoderQuality) of parameter to set the compression level of a JPEG image.

In Gdiplusenums.h, the [EncoderParameterValueType](#) enumeration indicates that data type 6 is **ValueLongRange**. A long range is a pair of **ULONG** values.

The number of values is one, so we know that the **Value** member of the [EncoderParameter](#) object is a pointer to an array that has one element. That one element is a pair of **ULONG** values.

The following code is a continuation of the console application that is shown in the preceding two examples. The code defines a data type called **PLONGRANGE** (pointer to a long range). A variable of type **PLONGRANGE** is used to extract the minimum and maximum values that can be passed as a quality setting to the JPEG encoder.

```
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);

```

The preceding code produces the following output:

```

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

```

In the preceding example, the value returned in the [EncoderParameter](#) object is a pair of **ULONG** values that indicate the minimum and maximum possible values for the quality parameter. In some cases, the values returned in an **EncoderParameter** object are members of the [EncoderValue](#) enumeration. The following topics discuss the **EncoderValue** enumeration and methods for listing possible parameter values in more detail:

- [Using the EncoderValue Enumeration](#)
- [Listing Parameters and Values for All Encoders](#)

2.5.4.1) Using the EncoderValue Enumeration

A given encoder supports certain parameter categories, and for each of those categories, that encoder allows certain values. For example, the JPEG encoder supports the EncoderValueQuality parameter category, and the allowable parameter values are the integers 0 through 100. Some of the allowable parameter values are the same across several encoders. These standard values are defined in the [EncoderValue](#) enumeration in Gdiplusenums.h:

```

enum EncoderValue
{
    EncoderValueColorTypeCMYK,           // 0
    EncoderValueColorTypeYCKK,          // 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
};

```

One of the parameter categories supported by the JPEG encoder is the EncoderTransformation category. By examining the [EncoderValue](#) enumeration, you might speculate (and you would be correct) that the EncoderTransformation category allows the following five values:

```

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

```

The following console application verifies that the JPEG encoder supports the EncoderTransformation parameter category and that there are five allowable values for such a parameter. The main function relies on the helper function GetEncoderClsid, which is shown in [Retrieving the Class Identifier for an Encoder](#).

```

#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;
}

```

```
}
```

When you run the preceding console application, you get an output similar to the following:

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.

You can look up the GUID in Gdiplusimaging.h and find out that the category of this [EncoderParameter](#) object is EncoderTransformation. In Gdiplusenums.h, the [EncoderParameterValueType](#) enumeration indicates that data type 4 is ValueLong (32-bit unsigned integer). The number of values is five, so we know that the **Value** member of the **EncoderParameter** object is a pointer to an array of five **ULONG** values.

The following code is a continuation of the console application that is shown in the preceding example. The code lists the allowable values for an EncoderTransformation parameter:

```
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]);
}
```

The preceding code produces the following output:

The allowable values are 13 14 15 16 17

The allowable values (13, 14, 15, 16, and 17) correspond to the following members of the [EncoderValue](#) enumeration:

```
EncoderValueTransformRotate90,          // 13
EncoderValueTransformRotate180,         // 14
```

```
EncoderValueTransformRotate270,      // 15
EncoderValueTransformFlipHorizontal,  // 16
EncoderValueTransformFlipVertical,    // 17
```

2.5.4.2) Listing Parameters and Values for All Encoders

The following console application lists all the parameters supported by the various encoders installed on the computer. The main function calls [GetImageEncoders](#) to discover which encoders are available. For each available encoder, the main function calls the helper function ShowAllEncoderParameters.

The ShowAllEncoderParameters function calls the [Image::GetEncoderParameterList](#) method to discover which parameters are supported by a given encoder. For each supported parameter, the function lists the category, data type, and number of values. The ShowAllEncoderParameters function relies on two helper functions: EncoderParameterCategoryFromGUID and 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
```

When you run the preceding console application, you get an output similar to the following:

```
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.
```

You can draw the following conclusions by examining the preceding program output:

- The JPEG encoder supports the Transformation, Quality, LuminanceTable, and ChrominanceTable parameter categories.
- The TIFF encoder supports the Compression, ColorDepth, and SaveFlag parameter categories.

You can also see the number of acceptable values for each parameter category. For example, you can see that the ColorDepth parameter category (TIFF codec) has five values of type **ULONG**. The following code lists those five values. Assume that **pEncoderParameters** is a pointer to an [EncoderParameters](#) object that represents the TIFF encoder.

```
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]);
}
```

The preceding code produces the following output:

The allowable values for ColorDepth are

```
1
4
8
24
32
```

Note In some cases, the values in an [EncoderParameter](#) object are the numeric values of elements of the [EncoderValue](#) enumeration. However, the numbers in the preceding list do not relate to the **EncoderValue** enumeration. The numbers mean 1 bit per pixel, 2 bits per pixel, and so on.

If you write code similar to the preceding example to investigate the allowable values for the other parameter categories, you will obtain a result similar to the following.

JPEG encoder parameter	Allowable values
Transformation	EncoderValueTransformRotate90
	EncoderValueTransformRotate180
	EncoderValueTransformRotate270
	EncoderValueTransformFlipHorizontal
Quality	EncoderValueTransformFlipVertical 0 through 100

TIFF encoder parameter	Allowable values
Compression	EncoderValueCompressionLZW
	EncoderValueCompressionCCITT3
	EncoderValueCompressionCCITT4
	EncoderValueCompressionRle

	EncoderValueCompressionNone
ColorDepth	1, 4, 8, 24, 32
SaveFlag	EncoderValueMultiFrame

Note If the width and height of a JPEG image are multiples of 16, you can apply any of the transformations allowed by the EncoderTransformation parameter category (for example, 90-degree rotation) without loss of information.

2.5.5) Converting a BMP Image to a PNG Image

To save an image to a disk file, call the [Save](#) method of the [Image](#) class. The following console application loads a BMP image from a disk file, converts the image to the PNG format, and saves the converted image to a new disk file. The main function relies on the helper function GetEncoderClsid, which is shown in [Retrieving the Class Identifier for an Encoder](#).

```
#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 == 0k)
    printf("Bird.png was saved successfully\n");
else
    printf("Failure: stat = %d\n", stat);

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

```

2.5.6) Setting JPEG Compression Level

To specify the compression level when you save a JPEG image, initialize an [EncoderParameters](#) object and pass the address of that object to the [Save](#) method of the [Image](#) class. Initialize the **EncoderParameters** object so that it has an array consisting of one [EncoderParameter](#) object. Initialize that one **EncoderParameter** object so that its **Value** member points to a **ULONG** value from 0 through 100. Set the **Guid** member of the **EncoderParameter** object to **EncoderQuality**.

The following console application saves three JPEG images, each with a different quality level. A quality level of 0 corresponds to the greatest compression, and a quality level of 100 corresponds to the least compression.

The main function relies on the helper function `GetEncoderClsid`, which is shown in [Retrieving the Class Identifier for an Encoder](#):

```

#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 == 0k)
    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;
}

```

2.5.7) Transforming a JPEG Image Without Loss of Information

When you compress a JPEG image, some of the information in the image is lost. If you open a JPEG file, alter the image, and save it to another JPEG file, the quality will decrease. If you repeat that process many times, you will see a substantial degradation in the image quality.

Because JPEG is one of the most popular image formats on the Web, and because people often like to modify JPEG images, GDI+ provides the following transformations that can be performed on JPEG images without loss of information:

- Rotate 90 degrees
- Rotate 180 degrees
- Rotate 270 degrees
- Flip horizontally
- Flip vertically

You can apply one of the transformations shown in the preceding list when you call the [Save](#) method of an [Image](#) object. If the following conditions are met, then the transformation will proceed without loss of information:

- The file used to construct the [Image](#) object is a JPEG file.
- The width and height of the image are both multiples of 16.

If the width and height of the image are not both multiples of 16, GDI+ will do its best to preserve the image quality when you apply one of the rotation or flipping transformations shown in the preceding list.

To transform a JPEG image, initialize an [EncoderParameters](#) object and pass the address of that object to the [Save](#) method of the [Image](#) class. Initialize the **EncoderParameters** object so that it has an array that consists of one [EncoderParameter](#) object. Initialize that one **EncoderParameter** object so that its **Value** member points to a **ULONG** variable that holds one of the following elements of the [EncoderValue](#) enumeration:

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

Set the **Guid** member of the [EncoderParameter](#) object to EncoderTransformation.

The following console application creates an [Image](#) object from a JPEG file and then saves the image to a new file. During the save process, the image is rotated 90 degrees. If the width and height of the image are both multiples of 16, the process of rotating and saving the image causes no loss of information.

The main function relies on the helper function GetEncoderClsid, which is shown in [Retrieving the Class Identifier for an Encoder](#).

```
#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 = EncoderParameterValueTypeLong;
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;
}
```

2.5.8) Creating and Saving a Multiple-Frame Image

With certain file formats, you can save multiple images (frames) to a single file. For example, you can save several pages to a single TIFF file. To save the first page, call the [Save](#) method of the [Image](#) class. To save subsequent pages, call the [SaveAdd](#) method of the **Image** class.

Note You cannot use [SaveAdd](#) to add frames to an animated gif file.

The following console application creates a TIFF file with four pages. The images that become the pages of the TIFF file come from four disk files: Shapes.bmp, Cereal.gif, Iron.jpg, and House.png. The code first constructs four [Image](#) objects: **multi**, **page2**, **page3**, and **page4**. At first, **multi** contains only the image from Shapes.bmp, but eventually it contains all four images. As the individual pages are added to the **multi Image** object, they are also added to the disk file Multiframe.tif.

Note that the code calls [Save](#) (not [SaveAdd](#)) to save the first page. The first argument passed to the [Save](#) method is the name of the disk file that will eventually contain several frames. The second argument passed to the [Save](#) method specifies the encoder that will be used to convert the data in the **multi Image** object to the format (in this case TIFF) required by the disk file. That same encoder is used automatically by all subsequent calls to the [SaveAdd](#) method of the **multi Image** object.

The third argument passed to the [Save](#) method is the address of an [EncoderParameters](#) object. The **EncoderParameters** object has an array that contains a single [EncoderParameter](#) object. The **Guid** member of that **EncoderParameter** object is set to EncoderSaveFlag. The **Value** member of the **EncoderParameter** object points to a **ULONG** that contains the value EncoderValueMultiFrame.

The code saves the second, third, and fourth pages by calling the [SaveAdd](#) method of the **multi Image** object. The first argument passed to the [SaveAdd](#) method is the address of an **Image** object. The image in that **Image** object is added to the **multi Image** object and is also added to the Multiframe.tif disk file. The second argument passed to the [SaveAdd](#) method is the address of the same [EncoderParameters](#) object that was used by the [Save](#) method. The difference is that the **ULONG** pointed to by the **Value** member now contains the value EncoderValueFrameDimensionPage.

The main function relies on the helper function GetEncoderClsid, which is shown in [Retrieving the Class Identifier for an Encoder](#).

```

#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 == 0k)
    printf("Page 2 saved successfully.\n");

// Save the third page (frame).
parameterValue = EncoderValueFrameDimensionPage;
stat = multi->SaveAdd(page3, &encoderParameters);
if(stat == 0k)
    printf("Page 3 saved successfully.\n");

// Save the fourth page (frame).
parameterValue = EncoderValueFrameDimensionPage;
stat = multi->SaveAdd(page4, &encoderParameters);
if(stat == 0k)
    printf("Page 4 saved successfully.\n");

// Close the multiframe file.
parameterValue = EncoderValueFlush;
stat = multi->SaveAdd(&encoderParameters);
if(stat == 0k)
    printf("File closed successfully.\n");

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

2.5.9) Copying Individual Frames from a Multiple-Frame Image

The following example retrieves the individual frames from a multiple-frame TIFF file. When the TIFF file was created, the individual frames were added to the Page dimension (see [Creating and Saving a Multiple-Frame Image](#)). The code displays each of the four pages and saves each page to a separate PNG disk file.

The code constructs an [Image](#) object from the multiple-frame TIFF file. To retrieve the individual frames (pages), the code calls the [Image::SelectActiveFrame](#) method of that **Image** object. The first argument passed to the **Image::SelectActiveFrame** method is the address of a GUID that specifies the dimension in which the frames were previously added to the multiple-frame TIFF file. The GUID `FrameDimensionPage` is defined in `Gdiplusimaging.h`. Other GUIDs defined in that header file are `FrameDimensionTime` and `FrameDimensionResolution`. The second argument passed to the **Image::SelectActiveFrame** method is the zero-based index of the desired page.

The code relies on the helper function `GetEncoderClsid`, which is shown in [Retrieving the Class Identifier for an Encoder](#).

```
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);

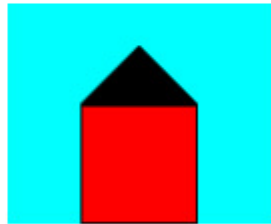
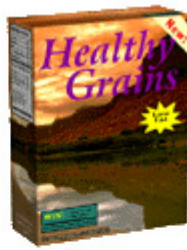
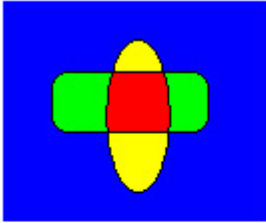
// Display and save the third page.
multi.SelectActiveFrame(&pageGuid, 2);
graphics.DrawImage(&multi, 10, 150);
multi.Save(L"Page2.png", &encoderClsid, NULL);

// Display and save the fourth page.
multi.SelectActiveFrame(&pageGuid, 3);
graphics.DrawImage(&multi, 200, 150);
```



```
multi.Save(L"Page3.png", &encoderClsid, NULL);
```

The following illustration shows the individual pages as displayed by the preceding code.



2.6) Alpha Blending Lines and Fills

In Windows GDI+, a color is a 32-bit value with 8 bits each for alpha, red, green, and blue. The alpha value indicates the transparency of the color — the extent to which the color is blended with the background color. Alpha values range from 0 through 255, where 0 represents a fully transparent color, and 255 represents a fully opaque color.

Alpha blending is a pixel-by-pixel blending of source and background color data. Each of the three components (red, green, blue) of a given source color is blended with the corresponding component of the background color according to the following formula:

$$\text{displayColor} = \text{sourceColor} \times \text{alpha} / 255 + \text{backgroundColor} \times (255 - \text{alpha}) / 255$$

For example, suppose the red component of the source color is 150 and the red component of the background color is 100. If the alpha value is 200, the red component of the resultant color is calculated as follows:

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

The following topics cover alpha blending in more detail:

- [Drawing Opaque and Semitransparent Lines](#)
 - [Drawing with Opaque and Semitransparent Brushes](#)
 - [Using Compositing Mode to Control Alpha Blending](#)
 - [Using a Color Matrix to Set Alpha Values in Images](#)
 - [Setting the Alpha Values of Individual Pixels](#)
-

2.6.1) Drawing Opaque and Semitransparent Lines

When you draw a line, you must pass the address of a [Pen](#) object to the [DrawLine](#) method of the [Graphics](#) class. One of the parameters of the **Pen** constructor is a [Color](#) object. To draw an opaque line, set the alpha component of the color to 255. To draw a semitransparent line, set the alpha component to any value from 1 through 254.

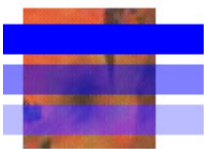
When you draw a semitransparent line over a background, the color of the line is blended with the colors of the background. The alpha component specifies how the line and background colors are mixed; alpha values near 0 place more weight on the background colors, and alpha values near 255 place more weight on the line color.

The following example draws an image and then draws three lines that use the image as a background. The first line uses an alpha component of 255, so it is opaque. The second and third lines use an alpha

component of 128, so they are semitransparent; you can see the background image through the lines. The call to [Graphics::SetCompositingQuality](#) causes the blending for the third line to be done in conjunction with gamma correction.

```
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);
```

The following illustration shows the output of the preceding code.



2.6.2) Drawing with Opaque and Semitransparent Brushes

When you fill a shape, you must pass the address of a [Brush](#) object to one of the fill methods of the [Graphics](#) class. The one parameter of the [SolidBrush](#) constructor is a [Color](#) object. To fill an opaque shape, set the alpha component of the color to 255. To fill a semitransparent shape, set the alpha component to any value from 1 through 254.

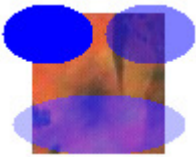
When you fill a semitransparent shape, the color of the shape is blended with the colors of the background. The alpha component specifies how the shape and background colors are mixed; alpha values near 0 place more weight on the background colors, and alpha values near 255 place more weight on the shape color.

The following example draws an image and then fills three ellipses that overlap the image. The first ellipse uses an alpha component of 255, so it is opaque. The second and third ellipses use an alpha component of 128, so they are semitransparent; you can see the background image through the ellipses. The call to [Graphics::SetCompositingQuality](#) causes the blending for the third ellipse to be done in conjunction with gamma correction.

```
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);
```

The following illustration shows the output of the preceding code.



2.6.3) Using Compositing Mode to Control Alpha Blending

There might be times when you want to create an off-screen bitmap that has the following characteristics:

- Colors have alpha values that are less than 255.
- Colors are not alpha blended with each other as you create the bitmap.
- When you display the finished bitmap, colors in the bitmap are alpha blended with the background colors on the display device.

To create such a bitmap, construct a blank [Bitmap](#) object, and then construct a [Graphics](#) object based on that bitmap. Set the compositing mode of the **Graphics** object to `CompositingModeSourceCopy`.

The following example creates a [Graphics](#) object based on a [Bitmap](#) object. The code uses the **Graphics** object along with two semitransparent brushes (alpha = 160) to paint on the bitmap. The code fills a red ellipse and a green ellipse using the semitransparent brushes. The green ellipse overlaps the red ellipse, but the green is not blended with the red because the compositing mode of the **Graphics** object is set to `CompositingModeSourceCopy`.

Next the code prepares to draw on the screen by calling [BeginPaint](#) and creating a [Graphics](#) object based on a device context. The code draws the bitmap on the screen twice: once on a white background and once on a multicolored background. The pixels in the bitmap that are part of the two ellipses have an alpha component of 160, so the ellipses are blended with the background colors on the screen.

```

// 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;
EndPaint(hWnd, &ps);

```

The following illustration shows the output of the preceding code. Note that the ellipses are blended with the background, but they are not blended with each other.



The preceding code example has the following statement:

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

If you want the ellipses to be blended with each other as well as with the background, change that statement to the following:

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

The following illustration shows the output of the revised code.



2.6.4) Using a Color Matrix to Set Alpha Values in Images

The [Bitmap](#) class (which inherits from the [Image](#) class) and the [ImageAttributes](#) class provide functionality for getting and setting pixel values. You can use the **ImageAttributes** class to modify the alpha values for an entire image, or you can call the [Bitmap::SetPixel](#) method of the **Bitmap** class to modify individual pixel values. For more information on setting individual pixel values, see [Setting the Alpha Values of Individual Pixels](#).

The following example draws a wide black line and then displays an opaque image that covers part of that line.

```
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()));
```

The following illustration shows the resulting image, which is drawn at (30, 0). Note that the wide black line doesn't show through the image.



The [ImageAttributes](#) class has many properties that you can use to modify images during rendering. In the following example, an **ImageAttributes** object is used to set all the alpha values to 80 percent of what they were. This is done by initializing a color matrix and setting the alpha scaling value in the matrix to 0.8. The address of the color matrix is passed to the [ImageAttributes::SetColorMatrix](#) method of the **ImageAttributes** object, and the address of the **ImageAttributes** object is passed to the [DrawImage](#) method of a [Graphics](#) object.

```
// 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);
```

During rendering, the alpha values in the bitmap are converted to 80 percent of what they were. This results in an image that is blended with the background. As the following illustration shows, the bitmap image looks transparent; you can see the solid black line through it.



Where the image is over the white portion of the background, the image has been blended with the color white. Where the image crosses the black line, the image is blended with the color black.

2.6.5) Setting the Alpha Values of Individual Pixels

The topic [Using a Color Matrix to Set Alpha Values in Images](#) shows a nondestructive method for changing the alpha values of an image. The example in that topic renders an image semitransparently, but the pixel data in the bitmap is not changed. The alpha values are altered only during rendering.

The following example shows how to change the alpha values of individual pixels. The code in the example actually changes the alpha information in a [Bitmap](#) object. The approach is much slower than using a color matrix and an [ImageAttributes](#) object but gives you control over the individual pixels in the bitmap.

```
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);
```

The following illustration shows the resulting image.



The preceding code example uses nested loops to change the alpha value of each pixel in the bitmap. For each pixel, [Bitmap::GetPixel](#) gets the existing color, [Color::SetValue](#) creates a temporary color that contains the new alpha value, and then [Bitmap::SetPixel](#) sets the new color. The alpha value is set based on the column of the bitmap. In the first column, alpha is set to 0. In the last column, alpha is set to 255. So the resulting image goes from fully transparent (on the left edge) to fully opaque (on the right edge).

[Bitmap::GetPixel](#) and [Bitmap::SetPixel](#) give you control of the individual pixel values. However, using **Bitmap::GetPixel** and **Bitmap::SetPixel** is not nearly as fast as using the [ImageAttributes](#) class and the [ColorMatrix](#) structure.

2.7) Using Text and Fonts

Windows GDI+ provides several classes that form the foundation for drawing text. The [Graphics](#) class has several [DrawString](#) methods that allow you to specify various features of text, such as location, bounding rectangle, font, and format. Other classes that contribute to text rendering include [FontFamily](#), [Font](#), [StringFormat](#), [InstalledFontCollection](#), and [PrivateFontCollection](#).

The following topics cover text and fonts in more detail:

- [Constructing Font Families and Fonts](#)
 - [Drawing Text](#)
 - [Formatting Text](#)
 - [Enumerating Installed Fonts](#)
 - [Creating a Private Font Collection](#)
 - [Obtaining Font Metrics](#)
 - [Antialiasing with Text](#)
-

2.7.1) Constructing Font Families and Fonts

Windows GDI+ groups fonts with the same typeface but different styles into font families. For example, the Arial font family contains the following fonts:

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

GDI+ uses four styles to form families: regular, bold, italic, and bold italic. Adjectives such as *narrow* and *rounded* are not considered styles; rather they are part of the family name. For example, Arial Narrow is a font family whose members are the following:

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

Before you can draw text with GDI+, you need to construct a [FontFamily](#) object and a [Font](#) object. The **FontFamily** object specifies the typeface (for example, Arial), and the **Font** object specifies the size, style, and units.

The following example constructs a regular style Arial font with a size of 16 pixels:

```
FontFamily fontFamily(L"Arial");  
Font font(&fontFamily, 16, FontStyleRegular, UnitPixel);
```

In the preceding code, the first argument passed to the [Font](#) constructor is the address of the [FontFamily](#) object. The second argument specifies the size of the font measured in units identified by the fourth argument. The third argument identifies the style.

[UnitPixel](#) is a member of the **Unit** enumeration, and [FontStyleRegular](#) is a member of the **FontStyle** enumeration. Both enumerations are declared in Gdiplusenums.h.

2.7.2) Drawing Text

You can use the [DrawString](#) method of the [Graphics](#) class to draw text at a specified location or within a specified rectangle.

- [Drawing Text at a Specified Location](#)
- [Drawing Text in a Rectangle](#)

=> Drawing Text at a Specified Location

To draw text at a specified location, you need [Graphics](#), [FontFamily](#), [Font](#), [PointF](#), and [Brush](#) objects.

The following example draws the string "Hello" at location (30, 10). The font family is Times New Roman. The font, which is an individual member of the font family, is Times New Roman, size 24 pixels, regular style. Assume that **graphics** is an existing [Graphics](#) object.

```
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);
```

The following illustration shows the output of the preceding code.



In the preceding example, the [FontFamily](#) constructor receives a string that identifies the font family. The address of the **FontFamily** object is passed as the first argument to the [Font](#) constructor. The second argument passed to the **Font** constructor specifies the size of the font measured in units given by the fourth argument. The third argument specifies the style (regular, bold, italic, and so on) of the font.

The [DrawString](#) method receives five arguments. The first argument is the string to be drawn, and the second argument is the length (in characters, not bytes) of that string. If the string is null-terminated, you can pass `-1` for the length. The third argument is the address of the [Font](#) object that was constructed previously. The fourth argument is a [PointF](#) object that contains the coordinates of the upper-left corner of the string. The fifth argument is the address of a [SolidBrush](#) object that will be used to fill the characters of the string.

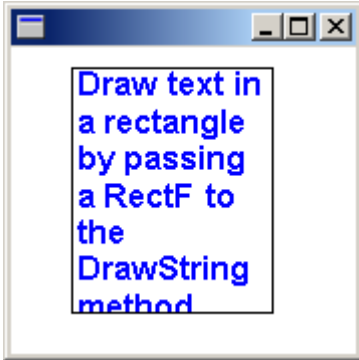
=> Drawing Text in a Rectangle

One of the [DrawString](#) methods of the [Graphics](#) class has a *RectF* parameter. By calling that **DrawString** method, you can draw text that wraps in a specified rectangle. To draw text in a rectangle, you need **Graphics**, [FontFamily](#), [Font](#), [RectF](#), and [Brush](#) objects.

The following example creates a rectangle with upper-left corner (30, 10), width 100, and height 122. Then the code draws a string inside that rectangle. The string is restricted to the rectangle and wraps in such a way that individual words are not broken.

```
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);
```

The following illustration shows the text drawn in the rectangle.



In the preceding example, the fourth argument passed to the [DrawString](#) method is a [RectF](#) object that specifies the bounding rectangle for the text. The fifth parameter is of type [StringFormat](#)— the argument is **NULL** because no special string formatting is required.

2.7.3) *Formatting Text*

To apply special formatting to text, initialize a [StringFormat](#) object and pass the address of that object to the [DrawString](#) method of the [Graphics](#) class.

To draw formatted text in a rectangle, you need [Graphics](#), [FontFamily](#), [Font](#), [RectF](#), [StringFormat](#), and [Brush](#) objects.

- [Aligning Text](#)
- [Setting Tab Stops](#)
- [Drawing Vertical Text](#)

=> **Aligning Text**

The following example draws text in a rectangle. Each line of text is centered (side to side), and the entire block of text is centered (top to bottom) in the rectangle.

```
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);

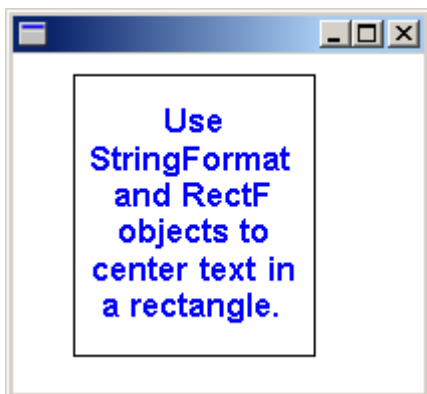
// Center the block of text (top to bottom) in the rectangle.
stringFormat.SetLineAlignment(StringAlignmentCenter);

graphics.DrawString(string, -1, &font, rectF, &stringFormat, &solidBrush);

Pen pen(Color(255, 0, 0, 0));
graphics.DrawRectangle(&pen, rectF);

```

The following illustration shows the rectangle and the centered text.



The preceding code calls two methods of the [StringFormat](#) object: [StringFormat::SetAlignment](#) and [StringFormat::SetLineAlignment](#). The call to **StringFormat::SetAlignment** specifies that each line of text is centered in the rectangle given by the third argument passed to the [DrawString](#) method. The call to **StringFormat::SetLineAlignment** specifies that the block of text is centered (top to bottom) in the rectangle.

The value [StringAlignmentCenter](#) is an element of the **StringAlignment** enumeration, which is declared in `Gdiplusenums.h`.

=> Setting Tab Stops

You can set tab stops for text by calling the [StringFormat::SetTabStops](#) method of a [StringFormat](#) object and then passing the address of that **StringFormat** object to the [DrawString](#) method of the [Graphics](#) class.

The following example sets tab stops at 150, 250, and 350. Then the code displays a tabbed list of names and test scores.

```
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);
```

The following illustration shows the tabbed text.

Name	Test 1	Test 2	Test 3
Joe	95	88	91
Mary	98	84	90
Sam	42	76	98
Jane	65	73	92

The preceding code passes three arguments to the [StringFormat::SetTabStops](#) method. The third argument is the address of an array containing the tab offsets. The second argument indicates that there are three offsets in that array. The first argument passed to **StringFormat::SetTabStops** is 0, which indicates that the first offset in the array is measured from position 0, the left edge of the bounding rectangle.

=> Drawing Vertical Text

You can use a [StringFormat](#) object to specify that text be drawn vertically rather than horizontally.

The following example passes the value [StringFormatFlagsDirectionVertical](#) to the [StringFormat::SetFormatFlags](#) method of a [StringFormat](#) object. The address of that **StringFormat** object is passed to the [DrawString](#) method of the [Graphics](#) class. The value [StringFormatFlagsDirectionVertical](#) is an element of the **StringFormatFlags** enumeration, which is declared in `Gdiplusenums.h`.

```
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);
```

The following illustration shows the vertical text.



2.7.4) Enumerating Installed Fonts

The [InstalledFontCollection](#) class inherits from the [FontCollection](#) abstract base class. You can use an **InstalledFontCollection** object to enumerate the fonts installed on the computer. The [FontCollection::GetFamilies](#) method of an **InstalledFontCollection** object returns an array of [FontFamily](#) objects. Before you call **FontCollection::GetFamilies**, you must allocate a buffer large enough to hold that array. To determine the size of the required buffer, call the [FontCollection::GetFamilyCount](#) method and multiply the return value by **sizeof(FontFamily)**.

The following example lists the names of all the font families installed on the computer. The code retrieves the font family names by calling the [FontFamily::GetFamilyName](#) method of each [FontFamily](#) object in the array returned by [FontCollection::GetFamilies](#). As the family names are retrieved, they are concatenated to form a comma-separated list. Then the [DrawString](#) method of the [Graphics](#) class draws the comma-separated list in a rectangle.

```
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"");

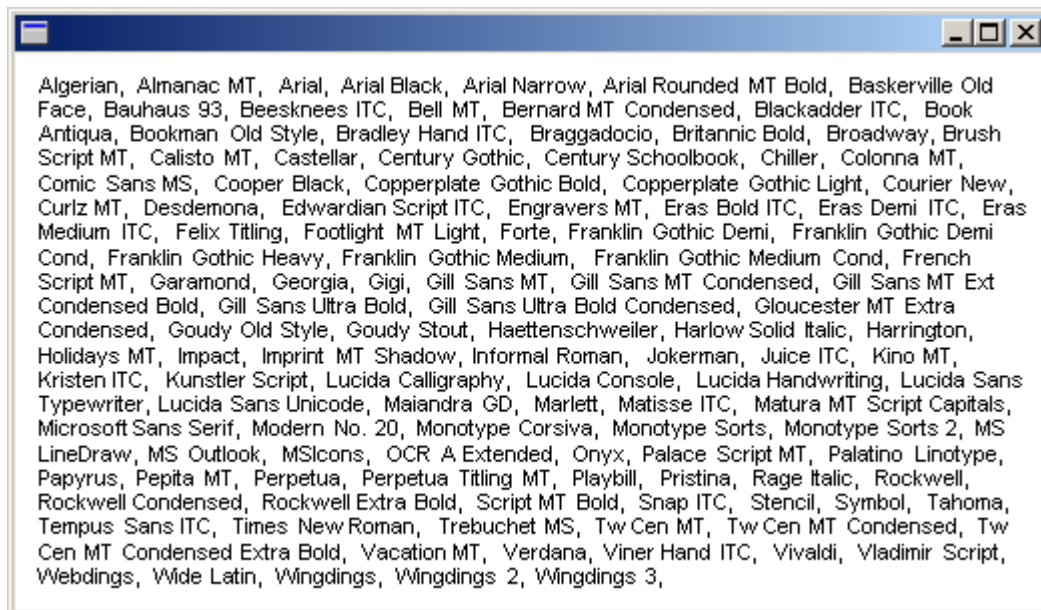
for(INT j = 0; j < count; ++j)
{
    pFontFamily[j].GetFamilyName(familyName);
    StringCchCatW(familyList, count*(sizeof(familyName)+ 3), familyName);
    StringCchCatW(familyList, count*(sizeof(familyName)+ 3), L",  ");
}

// Draw the large string (list of all families) in a rectangle.
graphics.DrawString(
    familyList, -1, &font, rectF, NULL, &solidBrush);

delete [] pFontFamily;
delete [] familyList;

```

The following illustration shows a possible output of the preceding code. If you run the code, the output might be different, depending on the fonts installed on your computer.



2.7.5) Creating a Private Font Collection

The [PrivateFontCollection](#) class inherits from the [FontCollection](#) abstract base class. You can use a **PrivateFontCollection** object to maintain a set of fonts specifically for your application.

A private font collection can include installed system fonts as well as fonts that have not been installed on the computer. To add a font file to a private font collection, call the [PrivateFontCollection::AddFontFile](#) method of a [PrivateFontCollection](#) object.

Note When you use the GDI+ API, you must never allow your application to download arbitrary fonts from untrusted sources. The operating system requires elevated privileges to assure that all installed fonts are trusted.

The [FontCollection::GetFamilies](#) method of a [PrivateFontCollection](#) object returns an array of [FontFamily](#) objects. Before you call **FontCollection::GetFamilies**, you must allocate a buffer large enough to hold that array. To determine the size of the required buffer, call the [FontCollection::GetFamilyCount](#) method and multiply the return value by **sizeof(FontFamily)**.

The number of font families in a private font collection is not necessarily the same as the number of font files that have been added to the collection. For example, suppose you add the files ArialBd.tff, Times.tff, and TimesBd.tff to a collection. There will be three files but only two families in the collection because Times.tff and TimesBd.tff belong to the same family.

The following example adds the following three font files to a [PrivateFontCollection](#) object:

- C:\WINNT\Fonts\Arial.tff (Arial, regular)
- C:\WINNT\Fonts\CourBI.tff (Courier New, bold italic)
- C:\WINNT\Fonts\TimesBd.tff (Times New Roman, bold)

The code calls the [FontCollection::GetFamilyCount](#) method of the [PrivateFontCollection](#) object to determine the number of families in the private collection, and then calls [FontCollection::GetFamilies](#) to retrieve an array of [FontFamily](#) objects.

For each [FontFamily](#) object in the collection, the code calls the [FontFamily::IsStyleAvailable](#) method to determine whether various styles (regular, bold, italic, bold italic, underline, and strikeout) are available. The arguments passed to the **FontFamily::IsStyleAvailable** method are members of the [FontStyle](#) enumeration, which is declared in Gdiplusenums.h.

If a particular 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.

```

#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;

```

The following illustration shows the output of the preceding code.



Arial.ttf (which was added to the private font collection in the preceding code example) is the font file for the Arial regular style. Note, however, that the program output shows several available styles other than regular for the Arial font family. That is because Windows GDI+ can simulate the bold, italic, and bold italic styles from the regular style. GDI+ can also produce underlines and strikeouts from the regular style.

Similarly, GDI+ can simulate the bold italic style from either the bold style or the italic style. The program output shows that the bold italic style is available for the Times family even though TimesBd.ttf (Times New Roman, bold) is the only Times file in the collection.

This table specifies the non-system fonts that GDI+ supports.

	GDI	GDI+ on Windows 7	GDI+ on Windows 8 Consumer Preview	DirectWrite
.FON	yes no		no	no
.FNT	yes no		no	no
.TTF	yes yes		yes	yes
.OTF with TrueType	yes yes		yes	yes
.OTF with Adobe CFF	yes no		yes	yes
Adobe Type 1	yes no		no	no

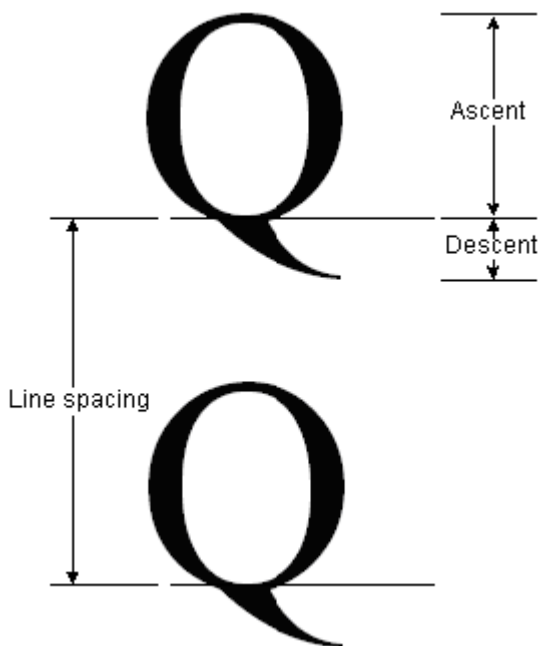
2.7.6) Obtaining Font Metrics

The [FontFamily](#) class provides the following methods that retrieve various metrics for a particular family/style combination:

- [FontFamily::GetEmHeight](#)(FontStyle)
- [FontFamily::GetCellAscent](#)(FontStyle)
- [FontFamily::GetCellDescent](#)(FontStyle)
- [FontFamily::GetLineSpacing](#)(FontStyle)

The numbers returned by these methods are in font design units, so they are independent of the size and units of a particular [Font](#) object.

The following illustration shows ascent, descent, and line spacing.



The following example displays the metrics for the regular style of the Arial font family. The code also creates a [Font](#) object (based on the Arial family) with size 16 pixels and displays the metrics (in pixels) for that particular **Font** object.

```
#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);

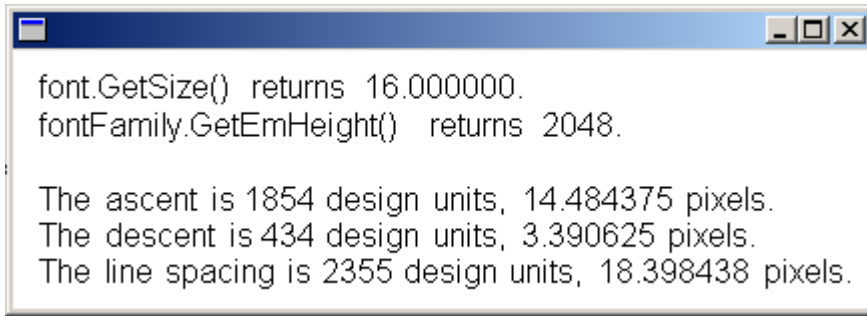
//  $18.398438 = 16.0 * 2355 / 2048$ 
lineSpacingPixel =
    font.GetSize() * lineSpacing / fontFamily.GetEmHeight(FontStyleRegular);

StringCchPrintf(
    infoString,
    INFO_STRING_SIZE,
    L"The line spacing is %d design units, %f pixels.",
    lineSpacing,
    lineSpacingPixel);

graphics.DrawString(infoString, -1, &font, pointF, &solidBrush);

```

The following illustration shows the output of the preceding code.



```
font.GetSize() returns 16.000000.  
fontFamily.GetEmHeight() returns 2048.  
  
The ascent is 1854 design units, 14.484375 pixels.  
The descent is 434 design units, 3.390625 pixels.  
The line spacing is 2355 design units, 18.398438 pixels.
```

Note the first two lines of output in the preceding illustration. The [Font](#) object returns a size of 16, and the [FontFamily](#) object returns an em height of 2,048. These two numbers (16 and 2,048) are the key to converting between font design units and the units (in this case pixels) of the **Font** object.

For example, you can convert the ascent from design units to pixels as follows:

$$\frac{1854 \text{ design units}}{1} \times \frac{16 \text{ pixels}}{2048 \text{ design units}} = 14.484375 \text{ pixels}$$

The preceding code positions text vertically by setting the y data member of a [PointF](#) object. The y-coordinate is increased by `font.GetHeight(0.0f)` for each new line of text. The [Font::GetHeight](#) method of a [Font](#) object returns the line spacing (in pixels) for that particular **Font** object. In this example, the number returned by **Font::GetHeight** is 18.398438. Note that this is the same as the number obtained by converting the line spacing metric to pixels.

2.7.7) Antialiasing with Text

Windows GDI+ provides various quality levels for drawing text. Typically, higher quality rendering takes more processing time than lower quality rendering.

The quality level is a property of the [Graphics](#) class. To set the quality level, call the [Graphics::SetTextRenderingHint](#) method of a **Graphics** object. The **Graphics::SetTextRenderingHint** method receives one of the elements of the [TextRenderingHint](#) enumeration, which is declared in `Gdiplusenums.h`.

GDI+ provides traditional antialiasing and a new kind of antialiasing based on Microsoft ClearType display technology only available on Windows XP and Windows Server 2003. ClearType smoothing improves readability on color LCD monitors that have a digital interface, such as the monitors in laptops and high-quality flat desktop displays. Readability on CRT screens is also somewhat improved.

ClearType is dependent on the orientation and ordering of the LCD stripes. Currently, ClearType is implemented only for vertical stripes that are ordered RGB. This might be a concern if you are using a tablet PC, where the display can be oriented in any direction, or if you are using a screen that can be turned from landscape to portrait.

The following example draws text with two different quality settings:

```
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);
```

The following illustration shows the output of the preceding code.

SingleBitPerPixel

AntiAlias

2.8) Constructing and Drawing Curves

GDI+ supports several types of curves: ellipses, arcs, cardinal splines, and Bézier splines. An ellipse is defined by its bounding rectangle; an arc is a portion of an ellipse defined by a starting angle and a sweep angle. A cardinal spline is defined by an array of points and a tension parameter — the curve passes smoothly through each point in the array, and the tension parameter influences the way the curve bends. A Bézier spline is defined by two end points and two control points — the curve does not pass through the control points, but the control points influence the direction and bend as the curve goes from one end point to the other.

The following topics cover cardinal splines and Bézier splines in more detail:

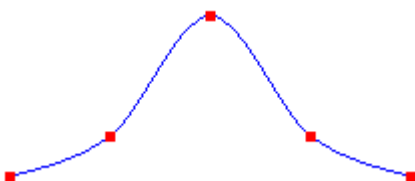
- [Drawing Cardinal Splines](#)
 - [Drawing Bézier Splines](#)
-

2.8.1) Drawing Cardinal Splines

A cardinal spline is a curve that passes smoothly through a given set of points. To draw a cardinal spline, create a [Graphics](#) object and pass the address of an array of points to the [Graphics::DrawCurve](#) method. The following example draws a bell-shaped cardinal spline that passes through five designated points:

```
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);
```

The following illustration shows the curve and five points.



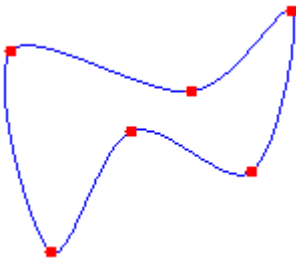
You can use the [Graphics::DrawClosedCurve](#) method of the [Graphics](#) class to draw a closed cardinal spline. In a closed cardinal spline, the curve continues through the last point in the array and connects with the first point in the array.

The following example draws a closed cardinal spline that passes through six designated points.

```
Point points[] = {Point(60, 60),
    Point(150, 80),
    Point(200, 40),
    Point(180, 120),
    Point(120, 100),
    Point(80, 160)};

Pen pen(Color(255, 0, 0, 255));
graphics.DrawClosedCurve(&pen, points, 6);
```

The following illustration shows the closed spline along with the six points:

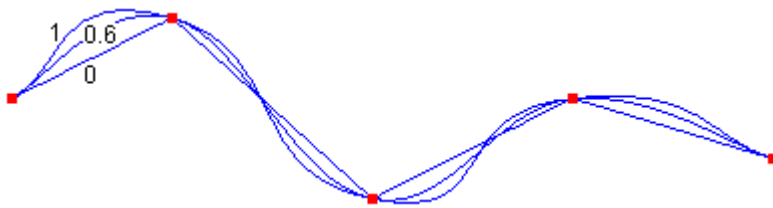


You can change the way a cardinal spline bends by passing a tension argument to the [Graphics::DrawCurve](#) method. The following example draws three cardinal splines that pass through the same set of points:

```
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
```

The following illustration shows the three splines along with their tension values. Note that when the tension is 0, the points are connected by straight lines.

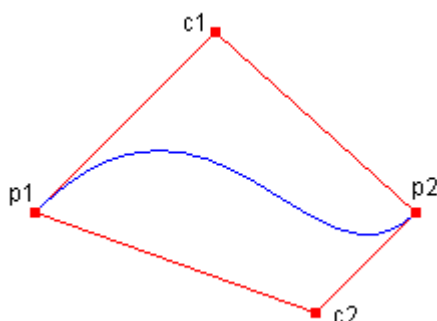


2.8.2) Drawing Bézier Splines

A Bézier spline is defined by four points: a start point, two control points, and an end point. The following example draws a Bézier spline with start point (10, 100) and end point (200, 100). The control points are (100, 10) and (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);
```

The following illustration shows the resulting Bézier spline along with its start point, control points, and end point. The illustration also shows the spline's convex hull, which is a polygon formed by connecting the four points with straight lines.



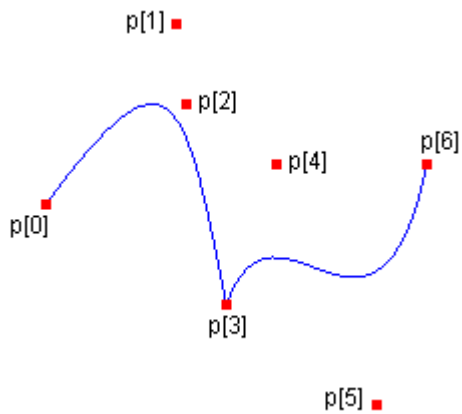
You can use the [DrawBeziers](#) method of the [Graphics](#) class to draw a sequence of connected Bézier splines. The following example draws a curve that consists of two connected Bézier splines. The end point of the first Bézier spline is the start point of the second Bézier spline.

```

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);

```

The following illustration shows the connected splines along with the seven points.



2.9) Filling Shapes with a Gradient Brush

You can use a gradient brush to fill a shape with a gradually changing color. For example, you can use a horizontal gradient to fill a shape with color that changes gradually as you move from the left edge of the shape to the right edge. Imagine a rectangle with a left edge that is black (represented by red, green, and blue components 0, 0, 0) and a right edge that is red (represented by 255, 0, 0). If the rectangle is 256 pixels wide, the red component of a given pixel will be one greater than the red component of the pixel to its left. The leftmost pixel in a row has color components (0, 0, 0), the second pixel has (1, 0, 0), the third pixel has (2, 0, 0), and so on, until you get to the rightmost pixel, which has color components (255, 0, 0). These interpolated color values make up the color gradient.

A linear gradient changes color as you move horizontally, vertically, or parallel to a specified slanted line. A path gradient changes color as you move about the interior and boundary of a path. You can customize path gradients to achieve a wide variety of effects.

GDI+ provides the [LinearGradientBrush](#) and [PathGradientBrush](#) classes, both of which inherit from the [Brush](#) class.

The following topics cover linear and path gradients in more detail:

- [Creating a Linear Gradient](#)
- [Creating a Path Gradient](#)
- [Applying Gamma Correction to a Gradient](#)

2.9.1) Creating a Linear Gradient

GDI+ provides horizontal, vertical, and diagonal linear gradients. By default, the color in a linear gradient changes uniformly. However, you can customize a linear gradient so that the color changes in a non-uniform fashion.

- [Horizontal Linear Gradients](#)
- [Customizing Linear Gradients](#)
- [Diagonal Linear Gradients](#)

=> Horizontal Linear Gradients

The following example uses a horizontal linear gradient brush to fill a line, an ellipse, and a rectangle:

```
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);

```

```

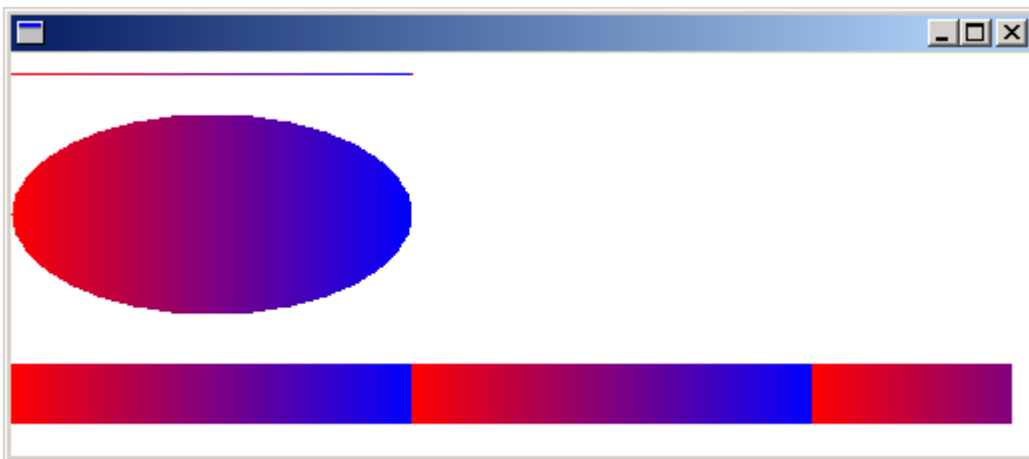
graphics.DrawLine(&pen, 0, 10, 200, 10);
graphics.FillEllipse(&linGrBrush, 0, 30, 200, 100);
graphics.FillRectangle(&linGrBrush, 0, 155, 500, 30);

```

The [LinearGradientBrush](#) constructor receives four arguments: two points and two colors. The first point (0, 10) is associated with the first color (red), and the second point (200, 10) is associated with the second color (blue). As you would expect, the line drawn from (0, 10) to (200, 10) changes gradually from red to blue.

The 10s in the points (50, 10) and (200, 10) are not important. What's important is that the two points have the same second coordinate — the line connecting them is horizontal. The ellipse and the rectangle also change gradually from red to blue as the horizontal coordinate goes from 0 to 200.

The following illustration shows the line, the ellipse, and the rectangle. Note that the color gradient repeats itself as the horizontal coordinate increases beyond 200.



=> Customizing Linear Gradients

In the preceding example, the color components change linearly as you move from a horizontal coordinate of 0 to a horizontal coordinate of 200. For example, a point whose first coordinate is halfway between 0 and 200 will have a blue component that is halfway between 0 and 255.

GDI+ allows you to adjust the way a color varies from one edge of a gradient to the other. Suppose you want to create a gradient brush that changes from black to red according to the following table.

Horizontal coordinate RGB components

0	(0, 0, 0)
40	(128, 0, 0)
200	(255, 0, 0)

Note that the red component is at half intensity when the horizontal coordinate is only 20 percent of the way from 0 to 200.

The following example calls the [LinearGradientBrush::SetBlend](#) method of a [LinearGradientBrush](#) object to associate three relative intensities with three relative positions. As in the preceding table, a relative intensity of 0.5 is associated with a relative position of 0.2. The code fills an ellipse and a rectangle with the gradient brush.

```
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);
```

The following illustration shows the resulting ellipse and rectangle.

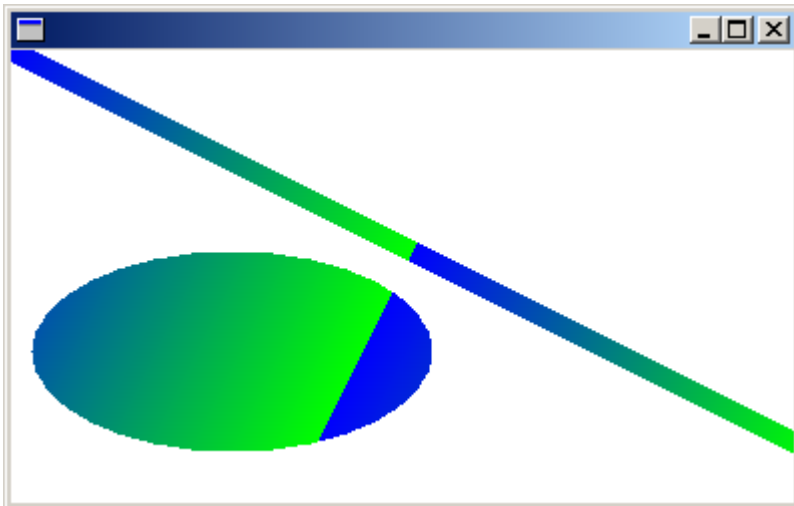


=> Diagonal Linear Gradients

The gradients in the preceding examples have been horizontal; that is, the color changes gradually as you move along any horizontal line. You can also define vertical gradients and diagonal gradients. The following code passes the points (0, 0) and (200, 100) to a [LinearGradientBrush](#) constructor. The color blue is associated with (0, 0), and the color green is associated with (200, 100). A line (with pen width 10) and an ellipse are filled with the linear gradient brush.

```
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);
```

The following illustration shows the line and the ellipse. Note that the color in the ellipse changes gradually as you move along any line that is parallel to the line passing through (0, 0) and (200, 100).



2.9.2) Creating a Path Gradient

The [PathGradientBrush](#) class allows you to customize the way you fill a shape with gradually changing colors. A **PathGradientBrush** object has a boundary path and a center point. You can specify one color for the center point and another color for the boundary. You can also specify separate colors for each of several points along the boundary.

Note In GDI+, a path is a sequence of lines and curves maintained by a [GraphicsPath](#) object. For more information about GDI+ paths, see [Paths](#) and [Constructing and Drawing Paths](#).

The following example fills an ellipse with a path gradient brush. The center color is set to blue and the boundary color is set to aqua.

```
// 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);
```

The following illustration shows the filled ellipse.



By default, a path gradient brush does not extend outside the boundary of the path. If you use the path gradient brush to fill a shape that extends beyond the boundary of the path, the area of the screen outside the path will not be filled. The following illustration shows what happens if you change the

[Graphics::FillEllipse](#) call in the preceding code to `graphics.FillRectangle(&pthGrBrush, 0, 10, 200, 40)`.



=> Specifying Points on the Boundary

The following example constructs a path gradient brush from a star-shaped path. The code calls the [PathGradientBrush::SetCenterColor](#) method to set the color at the centroid of the star to red. Then the code calls the [PathGradientBrush::SetSurroundColors](#) method to specify various colors (stored in the [colors](#) array) at the individual points in the [points](#) array. The final code statement fills the star-shaped path with the path gradient brush.

```
// 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);
```

The following illustration shows the filled star.



The following example constructs a path gradient brush based on an array of points. A color is assigned to each of the five points in the array. If you were to connect the five points by straight lines, you would get a five-sided polygon. A color is also assigned to the center (centroid) of that polygon — in this example, the center (80, 75) is set to white. The final code statement in the example fills a rectangle with the path gradient brush.

The color used to fill the rectangle is white at (80, 75) and changes gradually as you move away from (80, 75) toward the points in the array. For example, as you move from (80, 75) to (0, 0), the color changes gradually from white to red, and as you move from (80, 75) to (160, 0), the color changes gradually from white to green.

```
// 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);

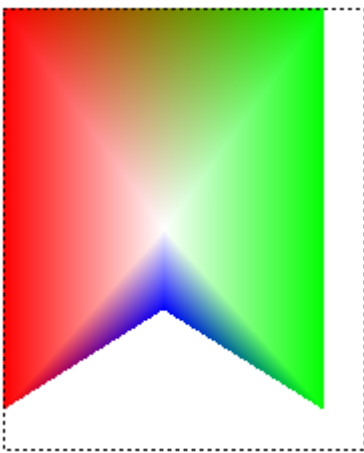
// An array of five points was used to construct the path gradient
// brush. Set the color of each point in that array.
Color colors[] = {Color(255, 255, 0, 0), // (0, 0) red
                  Color(255, 0, 255, 0), // (160, 0) green
                  Color(255, 0, 255, 0), // (160, 200) green
                  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));
```

Note that there is no [GraphicsPath](#) object in the preceding code. The particular [PathGradientBrush](#) constructor in the example receives a pointer to an array of points but does not require a **GraphicsPath** object. Also, note that the path gradient brush is used to fill a rectangle, not a path. The rectangle is larger than the path used to define the brush, so some of the rectangle is not painted by the brush. The following illustration shows the rectangle (dotted line) and the portion of the rectangle painted by the path gradient brush.



=> Customizing a Path Gradient

One way to customize a path gradient brush is to set its focus scales. The focus scales specify an inner path that lies inside the main path. The center color is displayed everywhere inside that inner path rather than only at the center point. To set the focus scales of a path gradient brush, call the [PathGradientBrush::SetFocusScales](#) method.

The following example creates a path gradient brush based on an elliptical path. The code sets the boundary color to blue, sets the center color to aqua, and then uses the path gradient brush to fill the elliptical path.

Next the code sets the focus scales of the path gradient brush. The x focus scale is set to 0.3, and the y focus scale is set to 0.8. The code calls the [Graphics::TranslateTransform](#) method of a [Graphics](#) object so that the subsequent call to [Graphics::FillPath](#) fills an ellipse that sits to the right of the first ellipse.

To see the effect of the focus scales, imagine a small ellipse that shares its center with the main ellipse. The small (inner) ellipse is the main ellipse scaled (about its center) horizontally by a factor of 0.3 and vertically by a factor of 0.8. As you move from the boundary of the outer ellipse to the boundary of the

inner ellipse, the color changes gradually from blue to aqua. As you move from the boundary of the inner ellipse to the shared center, the color remains aqua.

```
// 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);

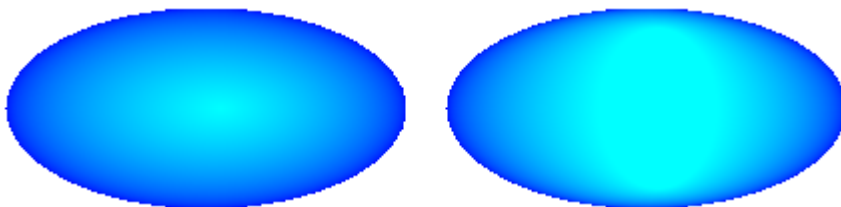
// Set the center color to aqua.
pthGrBrush.SetCenterColor(Color(255, 0, 255, 255));

// 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);
```

The following illustration shows the output of the preceding code. The ellipse on the left is aqua only at the center point. The ellipse on the right is aqua everywhere inside the inner path.



Another way to customize a path gradient brush is to specify an array of preset colors and an array of interpolation positions.

The following example creates a path gradient brush based on a triangle. The code calls the [PathGradientBrush::SetInterpolationColors](#) method of the path gradient brush to specify an array of interpolation colors (dark green, aqua, blue) and an array of interpolation positions (0, 0.25, 1). As you move from the boundary of the triangle to the center point, the color changes gradually from dark green to aqua and then from aqua to blue. The change from dark green to aqua happens in 25 percent of the distance from dark green to blue.

```
// 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);

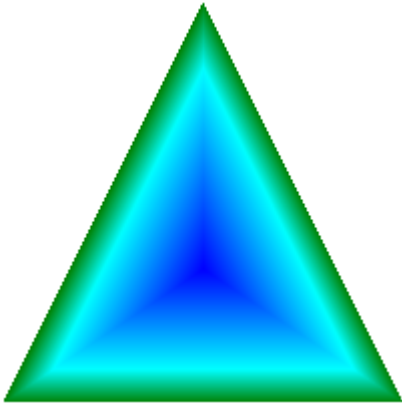
Color presetColors[] = {
    Color(255, 0, 128, 0),    // Dark green
    Color(255, 0, 255, 255), // Aqua
    Color(255, 0, 0, 255)};  // Blue

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);
```

The following illustration shows the output of the preceding code.



=> Setting the Center Point

By default, the center point of a path gradient brush is at the centroid of the path used to construct the brush. You can change the location of the center point by calling the [PathGradientBrush::SetCenterPoint](#) method of the [PathGradientBrush](#) class.

The following example creates a path gradient brush based on an ellipse. The center of the ellipse is at (70, 35), but the center point of the path gradient brush is set to (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);

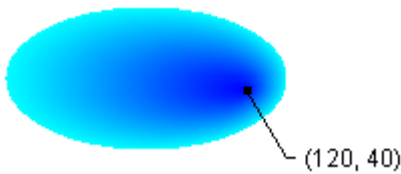
// Set the center point to a location that is not the centroid of the path.
pthGrBrush.SetCenterPoint(Point(120, 40));

// Set the color at the center point to blue.
pthGrBrush.SetCenterColor(Color(255, 0, 0, 255));

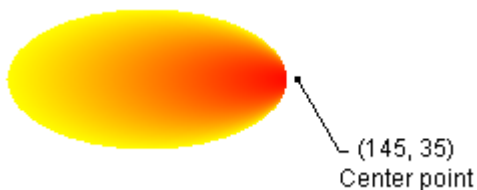
// Set the color along the entire boundary of the path to aqua.
Color colors[] = {Color(255, 0, 255, 255)};
int count = 1;
pthGrBrush.SetSurroundColors(colors, &count);

graphics.FillEllipse(&pthGrBrush, 0, 0, 140, 70);
```

The following illustration shows the filled ellipse and the center point of the path gradient brush.



You can set the center point of a path gradient brush to a location outside the path that was used to construct the brush. In the preceding code, if you replace the call to [PathGradientBrush::SetCenterPoint](#) with `pthGrBrush.SetCenterPoint(Point(145, 35))`, you will get the following result.



In the preceding illustration, the points at the far right of the ellipse are not pure blue (although they are very close). The colors in the gradient are positioned as if the fill had been allowed to reach the point (145, 35), the color would have reached pure blue (0, 0, 255). But the fill never reaches (145, 35) because a path gradient brush paints only inside its path.

2.9.3) Applying Gamma Correction to a Gradient

You can enable gamma correction for a gradient brush by passing **TRUE** to the [PathGradientBrush::SetGammaCorrection](#) method of that brush. You can disable gamma correction by passing **FALSE** to the **PathGradientBrush::SetGammaCorrection** method. Gamma correction is disabled by default.

The following example creates a linear gradient brush and uses that brush to fill two rectangles. The first rectangle is filled without gamma correction and the second rectangle is filled with gamma correction.

```
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);
```

The following illustration shows the two filled rectangles. The top rectangle, which does not have gamma correction, appears dark in the middle. The bottom rectangle, which has gamma correction, appears to have more uniform intensity.



The following example creates a path gradient brush based on a star-shaped path. The code uses the path gradient brush with gamma correction disabled (the default) to fill the path. Then the code passes **TRUE** to the [PathGradientBrush::SetGammaCorrection](#) method to enable gamma correction for the path gradient brush. The call to [Graphics::TranslateTransform](#) sets the world transformation of a [Graphics](#) object so that the subsequent call to [Graphics::FillPath](#) fills a star that sits to the right of the first star.

```
// 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);

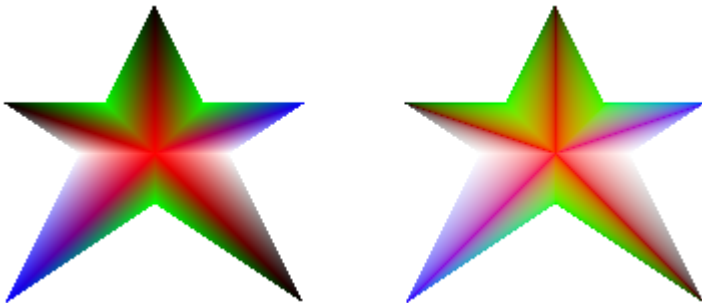
// Set the color at the center of the path to red.
pthGrBrush.SetCenterColor(Color(255, 255, 0, 0));

// 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);
```

The following illustration shows the output of the preceding code. The star on the right has gamma correction. Note that the star on the left, which does not have gamma correction, has areas that appear dark.



2.10) Constructing and Drawing Paths

A path is a sequence of graphics primitives (lines, rectangles, curves, text, and the like) that can be manipulated and drawn as a single unit. A path can be divided into *figures* that are either open or closed. A figure can contain several primitives.

You can draw a path by calling the [Graphics::DrawPath](#) method of the [Graphics](#) class, and you can fill a path by calling the [Graphics::FillPath](#) method of the **Graphics** class.

The following topics cover paths in more detail:

- [Creating Figures from Lines, Curves, and Shapes](#)
- [Filling Open Figures](#)

2.10.1) Creating Figures from Lines, Curves, and Shapes

To create a path, construct a [GraphicsPath](#) object, and then call methods, such as [AddLine](#) and [AddCurve](#), to add primitives to the path.

The following example creates a path that has a single arc. The arc has a sweep angle of -180 degrees, which is counterclockwise in the default coordinate system.

```
Pen pen(Color(255, 255, 0, 0));
GraphicsPath path;
path.AddArc(175, 50, 50, 50, 0, -180);
graphics.DrawPath(&pen, &path);
```

The following example creates a path that has two figures. The first figure is an arc followed by a line. The second figure is a line followed by a curve, followed by a line. The first figure is left open, and the second figure is closed.

```
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);

```

In addition to adding lines and curves to paths, you can add closed shapes: rectangles, ellipses, pies, and polygons. The following example creates a path that has two lines, a rectangle, and an ellipse. The code uses a pen to draw the path and a brush to fill the path.

```

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);

```

The path in the preceding example has three figures. The first figure consists of the two lines, the second figure consists of the rectangle, and the third figure consists of the ellipse. Even when there are no calls to [GraphicsPath::CloseFigure](#) or [GraphicsPath::StartFigure](#), intrinsically closed shapes, such as rectangles and ellipses, are considered separate figures.

2.10.2) Filling Open Figures

You can fill a path by passing the address of a [GraphicsPath](#) object to the [Graphics::FillPath](#) method. The **Graphics::FillPath** method fills the path according to the fill mode (alternate or winding) currently set for the path. If the path has any open figures, the path is filled as if those figures were closed. GDI+ closes a figure by drawing a straight line from its ending point to its starting point.

The following example creates a path that has one open figure (an arc) and one closed figure (an ellipse). The [Graphics::FillPath](#) method fills the path according to the default fill mode, which is `FillModeAlternate`.

```
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);
```

The following illustration shows the output of the preceding code. Note that path is filled (according to `FillModeAlternate`) as if the open figure were closed by a straight line from its ending point to its starting point.



2.11) Using Graphics Containers

A [Graphics](#) object provides methods such as [DrawLine](#), [DrawImage](#), and [DrawString](#) for displaying vector images, raster images, and text. A **Graphics** object also has several properties that influence the quality and orientation of the items that are drawn. For example, the smoothing mode property determines whether antialiasing is applied to lines and curves, and the world transformation property influences the position and rotation of the items that are drawn.

A [Graphics](#) object is often associated with a particular display device. When you use a **Graphics** object to draw in a window, the **Graphics** object is also associated with that particular window.

A [Graphics](#) object can be thought of as a container because it holds a set of properties that influence drawing, and it is linked to device-specific information. You can create a secondary container within an existing **Graphics** object by calling the [BeginContainer](#) method of that **Graphics** object.

The following topics cover [Graphics](#) objects and nested containers in more detail:

- [The State of a Graphics Object](#)
 - [Nested Graphics Containers](#)
-

2.11.1) The State of a Graphics Object

The [Graphics](#) class is at the heart of Windows GDI+. To draw anything, you create a **Graphics** object, set its properties, and call its methods ([DrawLine](#), [DrawImage](#), [DrawString](#), and the like).

The following example constructs a [Graphics](#) object and a [Pen](#) object and then calls the [Graphics::DrawRectangle](#) method of the **Graphics** object:

```
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);
```

In the preceding code, the [BeginPaint](#) method returns a handle to a device context, and that handle is passed to the [Graphics](#) constructor. A device context is a structure (maintained by Windows) that holds information about the particular display device being used.

=> Graphics State

A [Graphics](#) object does more than provide drawing methods, such as [DrawLine](#) and [DrawRectangle](#). A **Graphics** object also maintains graphics state, which can be divided into the following categories:

- A link to a device context
- Quality settings
- Transformations
- A clipping region

=> Device Context

As an application programmer, you don't have to think about the interaction between a [Graphics](#) object and its device context. This interaction is handled by GDI+ behind the scenes.

=> Quality Settings

A [Graphics](#) object has several properties that influence the quality of the items that are drawn on the screen. You can view and manipulate these properties by calling get and set methods. For example, you can call the [Graphics::SetTextRenderingHint](#) method to specify the type of antialiasing (if any) applied to text. Other set methods that influence quality are [Graphics::SetSmoothingMode](#), [Graphics::SetCompositingMode](#), [Graphics::SetCompositingQuality](#), and [Graphics::SetInterpolationMode](#).

The following example draws two ellipses, one with the smoothing mode set to [SmoothingModeAntiAlias](#) and one with the smoothing mode set to [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);
```

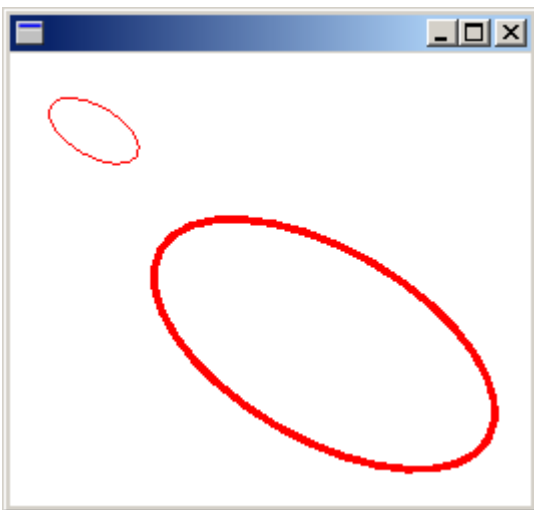
=> Transformations

A [Graphics](#) object maintains two transformations (world and page) that are applied to all items drawn by that **Graphics** object. Any affine transformation can be stored in the world transformation. Affine transformations include scaling, rotating, reflecting, skewing, and translating. The page transformation can be used for scaling and for changing units (for example, pixels to inches). For more information on transformations, see [Coordinate Systems and Transformations](#).

The following example sets the world and page transformations of a [Graphics](#) object. The world transformation is set to a 30-degree rotation. The page transformation is set so that the coordinates passed to the second [Graphics::DrawEllipse](#) will be treated as millimeters instead of pixels. The code makes two identical calls to the **Graphics::DrawEllipse** method. The world transformation is applied to the first **Graphics::DrawEllipse** call, and both transformations (world and page) are applied to the second **Graphics::DrawEllipse** call.

```
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);
```

The following illustration shows the two ellipses. Note that the 30-degree rotation is about the origin of the coordinate system (upper-left corner of the client area), not about the centers of the ellipses. Also note that the pen width of 1 means 1 pixel for the first ellipse and 1 millimeter for the second ellipse.



=> Clipping Region

A [Graphics](#) object maintains a clipping region that applies to all items drawn by that **Graphics** object. You can set the clipping region by calling the [SetClip](#) method.

The following example creates a plus-shaped region by forming the union of two rectangles. That region is designated as the clipping region of a [Graphics](#) object. Then the code draws two lines that are restricted to the interior of the clipping region.

```
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);
```

The following illustration shows the clipped lines.



2.11.2) Nested Graphics Containers

Windows GDI+ provides containers that you can use to temporarily replace or augment part of the state in a [Graphics](#) object. You create a container by calling the [Graphics::BeginContainer](#) method of a **Graphics** object. You can call **Graphics::BeginContainer** repeatedly to form nested containers.

=> Transformations in Nested Containers

The following example creates a [Graphics](#) object and a container within that **Graphics** object. The world transformation of the **Graphics** object is a translation 100 units in the x direction and 80 units in the y direction. The world transformation of the container is a 30-degree rotation. The code makes the call

```
DrawRectangle(&pen, -60, -30, 120, 60)
```

twice. The first call to [Graphics::DrawRectangle](#) is *inside the container*; that is, the call is in between the calls to [Graphics::BeginContainer](#) and [Graphics::EndContainer](#). The second call to **Graphics::DrawRectangle** is after the call to **Graphics::EndContainer**.

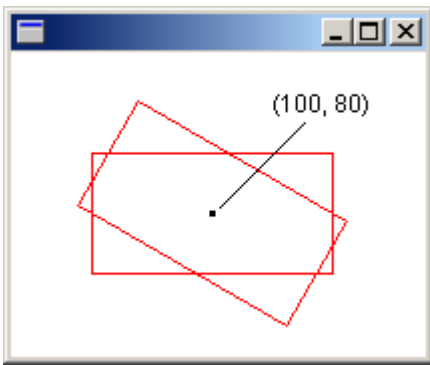
```
Graphics          graphics(hdc);
Pen               pen(Color(255, 255, 0, 0));
GraphicsContainer graphicsContainer;

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);
```

In the preceding code, the rectangle drawn from inside the container is transformed first by the world transformation of the container (rotation) and then by the world transformation of the [Graphics](#) object (translation). The rectangle drawn from outside the container is transformed only by the world transformation of the **Graphics** object (translation). The following illustration shows the two rectangles.



=> Clipping in Nested Containers

The following example illustrates how nested containers handle clipping regions. The code creates a [Graphics](#) object and a container within that **Graphics** object. The clipping region of the **Graphics** object is a rectangle, and the clipping region of the container is an ellipse. The code makes two calls to the [Graphics::DrawLine](#) method. The first call to **Graphics::DrawLine** is inside the container, and the second call to **Graphics::DrawLine** is outside the container (after the call to [Graphics::EndContainer](#)). The first line is clipped by the intersection of the two clipping regions. The second line is clipped only by the rectangular clipping region of the **Graphics** object.

```
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);

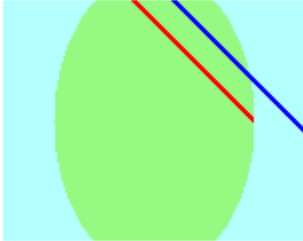
graphicsContainer = graphics.BeginContainer();
    // Create a path that consists of a single ellipse.
    GraphicsPath path;
    path.AddEllipse(75, 50, 100, 150);

    // 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);
```

The following illustration shows the two clipped lines.



As the two preceding examples show, transformations and clipping regions are cumulative in nested containers. If you set the world transformations of the container and the [Graphics](#) object, both transformations will apply to items drawn from inside the container. The transformation of the container will be applied first, and the transformation of the **Graphics** object will be applied second. If you set the clipping regions of the container and the **Graphics** object, items drawn from inside the container will be clipped by the intersection of the two clipping regions.

=> Quality Settings in Nested Containers

Quality settings ([SmoothingMode](#), [TextRenderingHint](#), and the like) in nested containers are not cumulative; rather, the quality settings of the container temporarily replace the quality settings of a [Graphics](#) object. When you create a new container, the quality settings for that container are set to default values. For example, suppose you have a **Graphics** object with a smoothing mode of [SmoothingModeAntiAlias](#). When you create a container, the smoothing mode inside the container is the default smoothing mode. You are free to set the smoothing mode of the container, and any items drawn from inside the container will be drawn according to the mode you set. Items drawn after the call to [Graphics::EndContainer](#) will be drawn according to the smoothing mode ([SmoothingModeAntiAlias](#)) that was in place before the call to [Graphics::BeginContainer](#).

=> Several Layers of Nested Containers

You are not limited to one container in a [Graphics](#) object. You can create a sequence of containers, each nested in the preceding, and you can specify the world transformation, clipping region, and quality settings of each of those nested containers. If you call a drawing method from inside the innermost container, the transformations will be applied in order, starting with the innermost container and ending with the outermost container. Items drawn from inside the innermost container will be clipped by the intersection of all the clipping regions.

The following example creates a [Graphics](#) object and sets its text rendering hint to [TextRenderingHintAntiAlias](#). The code creates two containers, one nested within the other. The text rendering hint of the outer container is set to [TextRenderingHintSingleBitPerPixel](#), and the text rendering hint of the inner container is set to [TextRenderingHintAntiAlias](#). The code draws three strings: one from the inner container, one from the outer container, and one from the **Graphics** object itself.

```
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);
```

The following illustration shows the three strings. The strings drawn from the inner container and the [Graphics](#) object are smoothed by antialiasing. The string drawn from the outer container is not smoothed by antialiasing because of the [TextRenderingHintSingleBitPerPixel](#) setting.

Inner Container
Outer Container
Graphics Object

2.12) Transformations

Affine transformations include rotating, scaling, reflecting, shearing, and translating. In Windows GDI+, the [Matrix](#) class provides the foundation for performing affine transformations on vector drawings, images, and text.

The following topics cover transformations in more detail:

- [Using the World Transformation](#)
 - [Why Transformation Order Is Significant](#)
-

2.12.1) Using the World Transformation

The world transformation is a property of the [Graphics](#) class. The numbers that specify the world transformation are stored in a [Matrix](#) object, which represents a 3 × 3 matrix. The **Matrix** and **Graphics** classes have several methods for setting the numbers in the world transformation matrix. The examples in this section manipulate rectangles because rectangles are easy to draw and it is easy to see the effects of transformations on rectangles.

We start by creating a 50 by 50 rectangle and locating it at the origin (0, 0). The origin is at the upper-left corner of the client area.

```
Rect rect(0, 0, 50, 50);  
Pen pen(Color(255, 255, 0, 0), 0);  
graphics.DrawRectangle(&pen, rect);
```

The following code applies a scaling transformation that expands the rectangle by a factor of 1.75 in the x direction and shrinks the rectangle by a factor of 0.5 in the y direction:

```
Rect rect(0, 0, 50, 50);  
Pen pen(Color(255, 255, 0, 0), 0);  
graphics.ScaleTransform(1.75f, 0.5f);  
graphics.DrawRectangle(&pen, rect);
```

The result is a rectangle that is longer in the x direction and shorter in the y direction than the original.

To rotate the rectangle instead of scaling it, use the following code instead of the preceding code:

```
Rect rect(0, 0, 50, 50);
Pen pen(Color(255, 255, 0, 0), 0);
graphics.RotateTransform(28.0f);
graphics.DrawRectangle(&pen, rect);
```

To translate the rectangle, use the following code:

```
Rect rect(0, 0, 50, 50);
Pen pen(Color(255, 255, 0, 0), 0);
graphics.TranslateTransform(150.0f, 150.0f);
graphics.DrawRectangle(&pen, rect);
```

2.12.2) Why Transformation Order Is Significant

A single [Matrix](#) object can store a single transformation or a sequence of transformations. The latter is called a *composite* 聽 *transformation*. The matrix of a composite transformation is obtained by multiplying the matrices of the individual transformations.

In a composite transformation, the order of the individual transformations is important. For example, if you first rotate, then scale, then translate, you get a different result than if you first translate, then rotate, then scale. In Windows GDI+, composite transformations are built from left to right. If S, R, and T are scale, rotation, and translation matrices respectively, then the product SRT (in that order) is the matrix of the composite transformation that first scales, then rotates, then translates. The matrix produced by the product SRT is different from the matrix produced by the product TRS.

One reason order is significant is that transformations like rotation and scaling are done with respect to the origin of the coordinate system. Scaling an object that is centered at the origin produces a different result than scaling an object that has been moved away from the origin. Similarly, rotating an object that is centered at the origin produces a different result than rotating an object that has been moved away from the origin.

The following example combines scaling, rotation and translation (in that order) to form a composite transformation. The argument [MatrixOrderAppend](#) passed to the [Graphics::RotateTransform](#) method specifies that the rotation will follow the scaling. Likewise, the argument [MatrixOrderAppend](#) passed to the [Graphics::TranslateTransform](#) method specifies that the translation will follow the rotation.

```
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);
```

The following example makes the same method calls as the previous example, but the order of the calls is reversed. The resulting order of operations is first translate, then rotate, then scale, which produces a very different result than first scale, then rotate, then translate:

```
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);
```

One way to reverse the order of the individual transformations in a composite transformation is to reverse the order of a sequence of method calls. A second way to control the order of operations is to change the matrix order argument. The following example is the same as the previous example, except that [MatrixOrderAppend](#) has been changed to [MatrixOrderPrepend](#). The matrix multiplication is done in the order SRT, where S, R, and T are the matrices for scale, rotate, and translate, respectively. The order of the composite transformation is first scale, then rotate, then translate.

```
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);
```

The result of the preceding example is the same result that we achieved in the first example of this section. This is because we reversed both the order of the method calls and the order of the matrix multiplication.

2.13) Using Regions

The Windows GDI+ [Region](#) class allows you to define a custom shape. The shape can be made up of lines, polygons, and curves.

Two common uses for regions are hit testing and clipping. Hit testing is determining whether the mouse was clicked in a certain region of the screen. Clipping is restricting drawing to a certain region.

The following topics cover regions in more detail:

- [Hit Testing with a Region](#)
 - [Clipping with a Region](#)
-

2.13.1) Hit Testing with a Region

The purpose of hit testing is to determine whether the cursor is over a given object, such as an icon or a button. The following example creates a plus-shaped region by forming the union of two rectangular regions. Assume that the variable **point** holds the location of the most recent click. The code checks to see whether **point** is in the plus-shaped region. If **point** is in the region (a hit), the region is filled with an opaque red brush. Otherwise, the region is filled with a semitransparent red brush.

```
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);
```

2.13.2) Clipping with a Region

One of the properties of the [Graphics](#) class is the clipping region. All drawing done by a given **Graphics** object is restricted to the clipping region of that **Graphics** object. You can set the clipping region by calling the **SetClip** method.

The following example constructs a path that consists of a single polygon. Then the code constructs a region based on that path. The address of the region is passed to the **SetClip** method of a [Graphics](#) object, and then two strings are drawn.

```
// 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);
```

The following illustration shows the clipped strings.



2.14) Recoloring

Recoloring is the process of adjusting image colors. Some examples of recoloring are changing one color to another, adjusting a color's intensity relative to another color, adjusting the brightness or contrast of all colors, and converting colors to shades of gray.

The following topics cover recoloring in more detail:

- [Using a Color Matrix to Transform a Single Color](#)
- [Translating Colors](#)
- [Scaling Colors](#)
- [Rotating Colors](#)
- [Shearing Colors](#)
- [Using a Color Remap Table](#)

2.14.1) Using a Color Matrix to Transform a Single Color

Windows GDI+ provides the [Image](#) and [Bitmap](#) classes for storing and manipulating images. **Image** and **Bitmap** objects store the color of each pixel as a 32-bit number: 8 bits each for red, green, blue, and alpha. Each of the four components is a number from 0 through 255, with 0 representing no intensity and 255 representing full intensity. The alpha component specifies the transparency of the color: 0 is fully transparent, and 255 is fully opaque.

A color vector is a 4-tuple of the form (red, green, blue, alpha). For example, the color vector (0, 255, 0, 255) represents an opaque color that has no red or blue, but has green at full intensity.

Another convention for representing colors uses the number 1 for maximum intensity and the number 0 for minimum intensity. Using that convention, the color described in the preceding paragraph would be

represented by the vector (0, 1, 0, 1). GDI+ uses the convention of 1 as full intensity when it performs color transformations.

You can apply linear transformations (rotation, scaling, and the like) to color vectors by multiplying by a 4 × 4 matrix. However, you cannot use a 4 × 4 matrix to perform a translation (nonlinear). If you add a dummy fifth coordinate (for example, the number 1) to each of the color vectors, you can use a 5 × 5 matrix to apply any combination of linear transformations and translations. A transformation consisting of a linear transformation followed by a translation is called an affine transformation. A 5 × 5 matrix that represents an affine transformation is called a homogeneous matrix for a 4-space transformation. The element in the fifth row and fifth column of a 5 × 5 homogeneous matrix must be 1, and all of the other entries in the fifth column must be 0.

For example, suppose you want to start with the color (0.2, 0.0, 0.4, 1.0) and apply the following transformations:

1. Double the red component
2. Add 0.2 to the red, green, and blue components

The following matrix multiplication will perform the pair of transformations in the order listed.

$$\begin{bmatrix} 0.2 & 0.0 & 0.4 & 1.0 & 1.0 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0.2 & 0.2 & 0.2 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0.6 & 0.2 & 0.6 & 1.0 & 1.0 \end{bmatrix}$$

The elements of a color matrix are indexed (zero-based) by row and then column. For example, the entry in the fifth row and third column of matrix M is denoted by M[4][2].

The 5 × 5 identity matrix (shown in the following illustration) has 1s on the diagonal and 0s everywhere else. If you multiply a color vector by the identity matrix, the color vector does not change. A convenient way to form the matrix of a color transformation is to start with the identity matrix and make a small change that produces the desired transformation.

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

Identity Matrix

For a more detailed discussion of matrices and transformations, see [Coordinate Systems and Transformations](#).

The following example takes an image that is all one color (0.2, 0.0, 0.4, 1.0) and applies the transformation described in the preceding paragraphs.

```
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);
```

The following illustration shows the original image on the left and the transformed image on the right.



The code in the preceding example uses the following steps to perform the recoloring:

1. Initialize a [ColorMatrix](#) structure.

2. Create an [ImageAttributes](#) object and pass the address of the [ColorMatrix](#) structure to the [ImageAttributes::SetColorMatrix](#) method of the **ImageAttributes** object.
 3. Pass the address of the [ImageAttributes](#) object to the [DrawImage Methods](#) method of a [Graphics](#) object.
-

2.14.2) Translating Colors

A translation adds a value to one or more of the four color components. The color matrix entries that represent translations are given in the following table.

Component to be translated	Matrix entry
----------------------------	--------------

Red	[4][0]
Green	[4][1]
Blue	[4][2]
Alpha	[4][3]

The following example constructs an [Image](#) object from the file ColorBars.bmp. Then the code adds 0.75 to the red component of each pixel in the image. The original image is drawn alongside the transformed image.

```
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);

```

The following illustration shows the original image on the left and the transformed image on the right.



The following table lists the color vectors for the four bars before and after the red translation. Note that because the maximum value for a color component is 1, the red component in the second row does not change. (Similarly, the minimum value for a color component is 0.)

Original	Translated
Black (0, 0, 0, 1)	(0.75, 0, 0, 1)
Red (1, 0, 0, 1)	(1, 0, 0, 1)
Green (0, 1, 0, 1)	(0.75, 1, 0, 1)
Blue (0, 0, 1, 1)	(0.75, 0, 1, 1)

2.14.3) *Scaling Colors*

A scaling transformation multiplies one or more of the four color components by a number. The color matrix entries that represent scaling are given in the following table.

Component to be scaled	Matrix entry
Red	[0] [0]
Green	[1] [1]
Blue	[2] [2]
Alpha	[3] [3]

The following example constructs an [Image](#) object from the file ColorBars2.bmp. Then the code scales the blue component of each pixel in the image by a factor of 2. The original image is drawn alongside the transformed image.

```
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);
```

The following illustration shows the original image on the left and the scaled image on the right.



The following table shows the color vectors for the four bars before and after the blue scaling. Note that the blue component in the fourth color bar went from 0.8 to 0.6. That is because GDI+ retains only the

fractional part of the result. For example, $(2)(0.8) = 1.6$, and the fractional part of 1.6 is 0.6. Retaining only the fractional part ensures that the result is always in the interval $[0, 1]$.

Original	Scaled
(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)

聽

The following example constructs an [Image](#) object from the file ColorBars2.bmp. Then the code scales the red, green, and blue components of each pixel in the image. The red components are scaled down 25 percent, the green components are scaled down 35 percent, and the blue components are scaled down 50 percent.

```
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);`

The following illustration shows the original image on the left and the scaled image on the right.

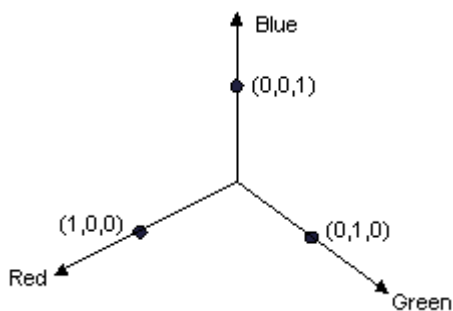


The following table shows the color vectors for the four bars before and after the red, green and blue scaling.

Original	Scaled
(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)

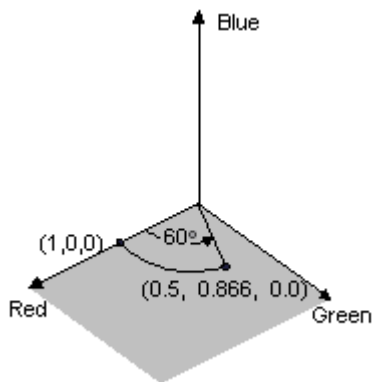
2.14.4) Rotating Colors

Rotation in a four-dimensional color space is difficult to visualize. We can make it easier to visualize rotation by agreeing to keep one of the color components fixed. Suppose we agree to keep the alpha component fixed at 1 (fully opaque). Then we can visualize a three-dimensional color space with red, green, and blue axes as shown in the following illustration.

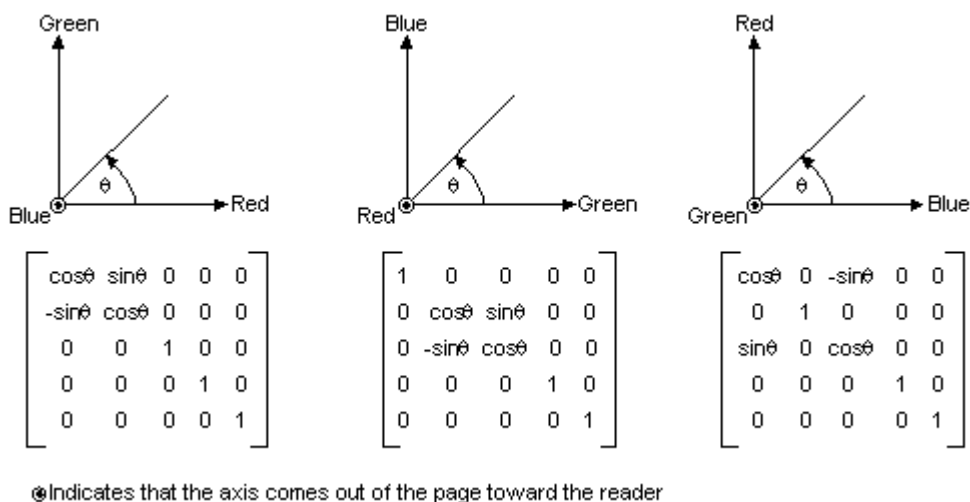


A color can be thought of as a point in 3-D space. For example, the point (1, 0, 0) in space represents the color red, and the point (0, 1, 0) in space represents the color green.

The following illustration shows what it means to rotate the color (1, 0, 0) through an angle of 60 degrees in the Red-Green plane. Rotation in a plane parallel to the Red-Green plane can be thought of as rotation about the blue axis.



The following illustration shows how to initialize a color matrix to perform rotations about each of the three coordinate axes (red, green, blue).



The following example takes an image that is all one color (1, 0, 0.6) and applies a 60-degree rotation about the blue axis. The angle of the rotation is swept out in a plane that is parallel to the Red-Green plane.

```
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);

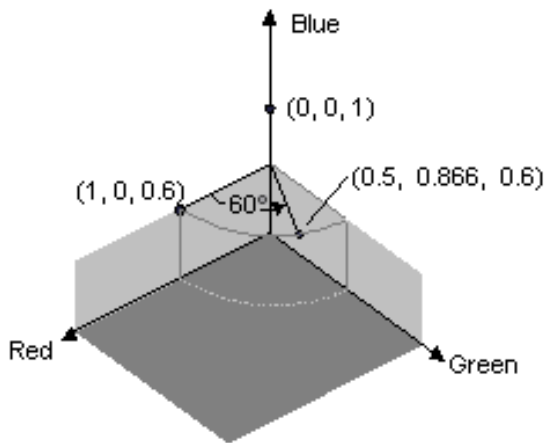
graphics.DrawImage(
    &image,
    Rect(130, 10, width, height), // destination rectangle
    0, 0, // upper-left corner of source rectangle
    width, // width of source rectangle
    height, // height of source rectangle
    UnitPixel,
    &imageAttributes);

```

The following illustration shows the original image on the left and the color-rotated image on the right.



The color rotation performed in the preceding code example can be visualized as follows.



The rotation takes place in the plane $\text{Blue}=0.6$, which is parallel to the Red-Green plane.

2.14.5) Shearing Colors

Shearing increases or decreases a color component by an amount proportional to another color component. For example, consider the transformation where the red component is increased by one half the value of the blue component. Under such a transformation, the color $(0.2, 0.5, 1)$ would become $(0.7, 0.5, 1)$. The new red component is $0.2 + (1/2)(1) = 0.7$.

The following example constructs an [Image](#) object from the file `ColorBars4.bmp`. Then the code applies the shearing transformation described in the preceding paragraph to each pixel in the image.

```
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);
```

The following illustration shows the original image on the left and the sheared image on the right.



The following table shows the color vectors for the four bars before and after the shearing transformation.

Original	Sheared
(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)

2.14.6) Using a Color Remap Table

Remapping is the process of converting the colors in an image according to a color remap table. The color remap table is an array of [ColorMap](#) structures. Each **ColorMap** structure in the array has an **oldColor** member and a **newColor** member.

When GDI+ draws an image, each pixel of the image is compared to the array of old colors. If a pixel's color matches an old color, its color is changed to the corresponding new color. The colors are changed only for rendering — the color values of the image itself (stored in an [Image](#) or [Bitmap](#) object) are not changed.

To draw a remapped image, initialize an array of [ColorMap](#) structures. Pass the address of that array to the [ImageAttributes::SetRemapTable](#) method of an [ImageAttributes](#) object, and then pass the address of the **ImageAttributes** object to the [DrawImage Methods](#) method of a [Graphics](#) object.

The following example creates an [Image](#) object from the file RemapInput.bmp. The code creates a color remap table that consists of a single [ColorMap](#) structure. The **oldColor** member of the **ColorMap** structure is red, and the **newColor** member is blue. The image is drawn once without remapping and once with remapping. The remapping process changes all the red pixels to blue.

```
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);
```

The following illustration shows the original image on the left and the remapped image on the right.



2.15) Printing

With a few minor adjustments to your code, you can send Windows GDI+ output to a printer rather than to a screen. To draw on a printer, obtain a device context handle for the printer and pass that handle to a [Graphics](#) constructor. Place your GDI+ drawing commands in between calls to [StartDoc](#) and [EndDoc](#).

The following topics cover sending GDI+ output to printers in more detail:

- [Sending GDI+ Output to a Printer](#)
- [Displaying a Print Dialog Box](#)
- [Optimizing Printing by Providing a Printer Handle](#)

2.15.1) Sending GDI+ Output to a Printer

Using Windows GDI+ to draw on a printer is similar to using GDI+ to draw on a computer screen. To draw on a printer, obtain a device context handle for the printer and then pass that handle to a [Graphics](#) constructor.

The following console application draws a line, a rectangle, and an ellipse on a printer named MyPrinter:

```
#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;
}

```

In the preceding code, the three GDI+ drawing commands are in between calls to the [StartDoc](#) and [EndDoc](#) functions, each of which receives the printer device context handle. All graphics commands between StartDoc and EndDoc are routed to a temporary metafile. After the call to EndDoc, the printer driver converts the data in the metafile into the format required by the specific printer being used.

Note If spooling is not enabled for the printer being used, the graphics output is not routed to a metafile. Instead, individual graphics commands are processed by the printer driver and then sent to the printer.

Generally you won't want to hard-code the name of a printer as was done in the preceding console application. One alternative to hard-coding the name is to call [GetDefaultPrinter](#) to obtain the name of the default printer. Before you call GetDefaultPrinter, you must allocate a buffer large enough to hold the printer name. You can determine the size of the required buffer by calling GetDefaultPrinter, passing **NULL** as the first argument.

Note The [GetDefaultPrinter](#) function is supported only on Windows 2000 and later.

The following console application gets the name of the default printer and then draws a rectangle and an ellipse on that printer. The [Graphics::DrawRectangle](#) call is in between calls to [StartPage](#) and [EndPage](#), so the rectangle is on a page by itself. Similarly, the ellipse is on a page by itself.

```

#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;
}

```

2.15.2) Displaying a Print Dialog Box

One way to get a device context handle for a printer is to display a print dialog box and allow the user to choose a printer. The [PrintDlg](#) function (which displays the dialog box) has one parameter that is the address of a [PRINTDLG](#) structure. The PRINTDLG structure has several members, but you can leave most of them set to their default values. The two members you need to set are **IStructSize** and **Flags**. Set **IStructSize** to the size of a PRINTDLG variable, and set **Flags** to PD_RETURNDC. Setting **Flags** to PC_RETURNDC specifies that you want the PrintDlg function to fill the **hDC** field with a device context handle for the printer chosen by the user.

```

#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;
}

```


2.15.3) Optimizing Printing by Providing a Printer Handle

One of the constructors for the [Graphics](#) class receives a device context handle and a printer handle. When you send Windows GDI+ commands to certain PostScript printers, the performance will be better if you create your **Graphics** object with that particular constructor.

The following console application calls [GetDefaultPrinter](#) to get the name of the default printer. The code passes the printer name to [CreateDC](#) to obtain a device context handle for the printer. The code also passes the printer name to [OpenPrinter](#) to obtain a printer handle. Both the device context handle and the printer handle are passed to the [Graphics](#) constructor. Then two figures are drawn on the printer.

Note The [GetDefaultPrinter](#) function is supported only on Windows 2000 and later.

```
#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;
}

```