Chapter 6: Working with Device Contexts and GDI Objects

In This Chapter

- Device Contexts in MFC
- Brushes and Pens
- MFC Classes for GDI Operations
- Working with Fonts
- Creating and Loading Bitmaps
- Drawing with Bitmaps
- Creating a Device-Independent Bitmap Class

The device context, or DC, is the interface between an application that draws graphics and text on a two-dimensional surface and the device drivers and hardware that render those graphics.

Applications can draw with the same graphics device interface (GDI) functions, regardless of the actual hardware used, to achieve a consistent image to the best capabilities of the device.

The device context also can be interrogated to inform an application of the capabilities and dimensions supported by the device. The device context can use this dimensional information to allow an application to render images in physical coordinate systems, such as inches and millimeters.

Device Contexts in MFC

Device contexts are Win32 objects; they are represented by hdc device-context handles. The MFC provides wrapper classes for the device-context object as the CDC base class, as well as a number of more specialized derived classes.

The basic CDC class is huge and supports all the GDI drawing functions, coordinate mapping functions, clipping functions, font-manipulation and rendering functions, printer-specific functions, path-tracking functions, and metafile-playing functions.

This section briefly covers the various device-context classes and when and how to use and obtain objects of these classes.

The CDC Class

The CDC base class encapsulates all the device-context functionality and drawing functions that use a Win32 hdc object. The actual Win32 device-context handle is accessible via the public m_hdc member. You can retrieve this handle with the device context’s GetSafeHdc() function.

You often will be handed a pointer to an initialized CDC object from MFC framework functions, such as CView::OnDraw() and CView::OnBeginPrinting(). These objects are nicely clipped to the dimensions of the window client area so that the results of drawing functions do not appear outside the area of the window.
You also can obtain a pointer to a CDC object for the client area of a window using the CWnd::GetDC() function. If you want a CDC pointer for the entire window area (including the title bar and borders), you can use the CWnd::GetWindowDC() function instead. You can even get a pointer to the entire Windows desktop by calling GetDesktopWindow()->GetWindowDC().

A single Win32 device-context object may be shared among windows of the same registered window class (windows registered with RegisterClass(), not to be confused with Object-Oriented classes). The object may be part of a parent window’s device context, or it may be private to the specific window.

After you obtain a pointer to the device context, you can perform a number of drawing operations or other device context–specific functions.

If you have obtained the pointer to a window’s device context via GetDC(), however, you must call CWnd::ReleaseDC() on the same CWnd object to release that window’s device context for other applications to use.

Warning - Some platforms, such as Windows 95, share a limited number of common device contexts among applications. You may be returned a NULL pointer if the GetDC() operation cannot succeed. You therefore should always call ReleaseDC() to alleviate this situation by freeing the device context for other applications.

You also can use the GetDCEx() function, which retrieves the device context and lets you pass a flag value to control the clipping of the returned device context.

The CDC constructor function constructs an uninitialized device context, where the m_hDC member is invalid. You then can initialize it in a number of ways. The CreateDC() function lets you initialize the device context for a specific device with a driver and device name. For example, you can create a device context for the screen like this:

```cpp
CDC dc;
dc.CreateDC("Display", NULL, NULL, NULL);
```

Or you could create a device context for a printer like this:

```cpp
CDC dc;
dc.CreateDC(NULL,"Cannon Bubble-Jet BJ330",NULL,NULL);
```

The last parameter lets you pass a DEVMODE structure that can be used to set specific initialization defaults. Otherwise, the driver’s own defaults are used.

If you just want to retrieve information regarding the capabilities of a device, you can use the CreateIC() function and then use the GetDeviceCaps() function to retrieve the capabilities of the specified device. The GetDeviceCaps() function can be used with any device context, but a device context created with CreateIC() is a special cut-down device context that only supplies information about a device and cannot be used for drawing operations.

You can use the CreateCompatibleDC() function to create a memory device context compatible with a reference device context supplied as a pointer to a real CDC object. The memory device context is a memory-based representation of the actual device. If you pass NULL as the reference device context, a memory device context is created that is compatible with the screen.

The most common use of a memory device context is to create an offscreen buffer to perform screen capture copy operations via bit-blitting (fast memory copying), or to copy images from the memory device context onto the screen device context.
Note - Memory device contexts created with CreateCompatibleDC() are created with a minimum monochrome bitmap. If you want to copy color images between a memory device context and the screen, you must create a compatible memory bitmap.

If you have created a device context (instead of obtaining one via GetDC() or using a pointer to a framework device context), you should delete the Win32 device-context object after use by calling the DeleteDC() member function.

You also can initialize an uninitialized device context from an HDC device-context handle by using the Attach() function and passing the HDC handle. You can detach the handle by calling the corresponding Detach() function, which returns the old HDC handle.

If you just want to quickly create and attach a temporary CDC object to an HDC handle, the FromHandle() function allocates and attaches a CDC object for you, returning the pointer. You should not keep any references to this object, however, because the object is destroyed automatically the next time your program returns to the application’s message loop by a call to DeleteTempMap().

You can find the window associated with a device context by calling the device context’s GetWindow() function, which returns a CWnd pointer to the associated window.

The device context maintains information relating to coordinate mapping modes, clipping rules, and the current GDI objects used for drawing (such as brushes, pens, bitmaps, and palettes). The details of these are discussed in subsequent sections. You can save and restore all these settings by using SaveDC() and RestoreDC(). When you call SaveDC(), an integer is returned identifying the saved instance (or zero, if it fails). You later can pass this integer to RestoreDC() to restore those saved defaults.

The CClientDC Class

You can use the CClientDC class to quickly connect a CDC object to a window by passing a pointer to the desired window. This is equivalent to constructing a CDC object and attaching an HDC handle obtained from a window with the Win32 ::GetDC() function.

If the constructor succeeds, the m_hWnd member variable is initialized with the handle of the window donating the HDC handle; otherwise, a CResourceException is raised.

When the CClientDC() object falls out of scope or is deleted, the Win32 device context object (HDC) is released automatically.

For example, you might want to access the client device context of a view window from a view-based menu handler to draw a circle, like this:

```cpp
void CCDCBaseView::OnDrawEllipse()
{
    CClientDC dcClient(this);
    dcClient.Ellipse(0,0,100,100);
}
```

You will notice that the C++ this pointer simply is passed to the CClientDC constructor to initialize the device context with the view’s client device context. After initialization, the circle can be drawn in the view using the CDC::Ellipse() function. Finally, the destructor of the CClientDC object ensures that the view’s HDC is released.

The CPaintDC Class

A CPaintDC object is constructed in response to a WM_PAINT message. If you add a handler for this message, you will see that ClassWizard automatically generates an OnPaint() handler starting with the following line:
CPaintDC dc(this); // device context for painting

The CPaintDC constructor automatically calls the BeginPaint() function for you. The BeginPaint() function is responsible for initializing a PAINTSTRUCT structure. PAINTSTRUCT holds information about the smallest rectangle that needs to be redrawn in response to an invalid portion of a window (usually when one window is moved to reveal a portion of the one behind it).

This process is automated in the constructor so that you can immediately use the device context to redraw the invalid area.

The PAINTSTRUCT information is available through the m_ps member, and the attached window handle can be found from the m_hWnd member.

When the object falls out of scope and its destructor is invoked, the EndPaint() function is called to complete the usual WM_PAINT procedure and mark the invalid region as valid.

The CMetaFileDC Class

A Windows metafile is a file that contains a list of GDI drawing instructions required to render an image in a device context. The metafile can be stored on disk, reloaded on another computer with different display capabilities, and redrawn by repeating the drawing instructions to produce a similar image.

You can always play a metafile using the CDC base class's PlayMetaFile() and passing a handle to a metafile object. The CMetaFileDC class lets you also create and record metafiles, however.

To record a metafile, you should construct a CMetaFileDC object. Then call Create() or CreateEnhanced() to create a simple or enhanced metafile on disk passing a filename and a reference device context for the enhanced metafile.

Note - Enhanced metafiles store palette and size information, so they can be reproduced more accurately on different computers with various display capabilities.

After creating the metafile, you can perform a number of drawing operations in the device context, just as you can in any other device context. The details of these drawing operations are stored in sequence in the metafile as you call each drawing function.

Finally, you should call CMetaFileDC::Close() or CMetaFileDC::CloseEnhanced() to close the metafile. The close functions return an HMETAFILE or HENHMETAFILE handle, which can be used by the Win32 metafile manipulation functions, such as CopyMetaFile() or DeleteMetaFile().

Brushes and Pens

So far, you have examined the CDC base class and its derivatives and looked at how to obtain and create device contexts in various situations. The main purpose of a device context, however, is to provide a uniform drawing surface for the GDI rendering functions. These rendering functions use two main GDI objects to draw lines and filled areas. Lines are drawn with objects called pens, and filled areas are drawn with objects called brushes.

These two objects are represented at the Win32 level by the HPEN and HBRUSH handles. All of the GDI objects, including pens and brushes, are wrapped by the CGdiObject MFC base class. Although the CGdiObject is not strictly an abstract base class, you normally would construct one of its derived classes, such as CPen or CBrush.

CGdiObject is responsible for manipulating these handles and provides member functions such as Attach() and Detach() to attach the object to a GDI handle and detach it after use.
The handle itself is stored in the m_hObject member, which you can retrieve by calling the CGdiObject::GetSafeHandle() function. A static function (similar to the device contexts) called FromHandle() lets you dynamically create a CGdiObject for the duration of a Windows message. This object then is deleted by DeleteTempMap() when the thread returns to the Windows message loop.

You can find details about the underlying object by using the GetObject() function, which fills a structure with the object’s attributes. The type of structure filled depends on the type of underlying object. The function itself just takes an LPVOID pointer to a buffer to receive the details and the size of the buffer.

The DeleteObject() function deletes the underlying GDI object from memory, so you are free to reuse the same CGdiObject by calling one of the derived classes’ Create() functions to create another GDI object.

The CGdiObject class is extended by the MFC CPen and CBrush classes, which greatly simplify creating and manipulating these two drawing objects. This section shows you how these classes are constructed and used.

**Pens and the CPen Class**

Pens are used to draw lines, curves, and the boundaries of filled shapes, such as ellipses, polygons, and chords. You can create pens of various sizes, styles, and colors (even invisible pens for drawing shapes without outlines), and then select a pen into the device context and draw with it. You can select only one pen into a device context at any time. The old pen is swapped out automatically when you select a new pen.

There are two main pen types: cosmetic and geometric. The cosmetic pens are simple to create and quick to draw with. The geometric pens support more precise world units and let you use complex patterns and hatching effects in ways similar to brushes (described later in this chapter, in the section, “Brushes and the CBrush Class”).

You can create a variety of pens using the CPen constructor. There are two versions of the constructor. You can pass one version of the constructor a set of style flags (as shown in Table 6.1), the pen width, and a COLORREF to indicate the color of the new pen.

<table>
<thead>
<tr>
<th>Style</th>
<th>Pen Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS_SOLID</td>
<td>Draws in a solid color.</td>
</tr>
<tr>
<td>PS_NULL</td>
<td>Uses an invisible pen.</td>
</tr>
<tr>
<td>PS_DASH</td>
<td>Draws in a dashed style.</td>
</tr>
<tr>
<td>PS_DASHDOT</td>
<td>Draws alternating dashes and dots.</td>
</tr>
<tr>
<td>PS_DOT</td>
<td>Draws in a dotted style.</td>
</tr>
<tr>
<td>PS_INSIDEFRAME</td>
<td>Draws lines inside filled areas.</td>
</tr>
</tbody>
</table>

**Table 6.1 Simple Pen Styles**

Note - The PS_DASH, PS_DASHDOT, and PS_DOT styles only work with pens that have a width of 1 pixel; otherwise, they are drawn as solid lines.

The other constructor form lets you supply a pointer to a LOGBRUSH structure to initialize the attributes of a geometric pen. You also can pass an array of DWORD values to specify complex dot and dash patterns. You can use some additional flags to change the mitering (how lines are joined) and end-cap styles (how lines end). Table 6.2 explains these flags.
Table 6.2 Advanced Pen Styles

<table>
<thead>
<tr>
<th>Style</th>
<th>Pen Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS_COSMETIC</td>
<td>Specifies a cosmetic pen.</td>
</tr>
<tr>
<td>PS_GEOMETRIC</td>
<td>Specifies a geometric pen.</td>
</tr>
<tr>
<td>PS_ALTERNATE</td>
<td>Sets every other pixel (only cosmetic pens).</td>
</tr>
<tr>
<td>PS_USERSTYLE</td>
<td>Uses the user-style DWORD array to create the dash-dot pattern.</td>
</tr>
<tr>
<td>PS_JION_ROUND</td>
<td>Draws round, joined lines.</td>
</tr>
<tr>
<td>PS_JION_BEVEL</td>
<td>Draws beveled, joined lines.</td>
</tr>
<tr>
<td>PS_JION_MITER</td>
<td>Draws mitered, joined lines.</td>
</tr>
<tr>
<td>PS_ENDCAP_FLAT</td>
<td>Makes the ends of lines flat.</td>
</tr>
<tr>
<td>PS_ENDCAP_ROUND</td>
<td>Makes the ends of lines round.</td>
</tr>
<tr>
<td>PS_ENDCAP_SQUARE</td>
<td>Makes the ends of lines square.</td>
</tr>
</tbody>
</table>

You also can construct an uninitialized pen by using the default constructor. You then initialize it by using CreatePen() and passing the same parameters as the two constructors for the CPen class, or by using CreatePenIndirect() and passing a pointer to a LOGPEN structure that holds the pen-initialization details.

Selecting Pens into the Device Context

After you create a pen, you must select it into a device context in order for the drawing functions to start using that pen. You can select all of the GDI objects by using the device context’s overloaded SelectObject() function.

The SelectObject() function that accepts a CPen object returns a pointer to the currently selected CPen object. When you select your new object, the current object is deselected automatically. You should save the pointer to this deselected object to reselect it when you finish drawing with your pen.

Warning - You should never let a pen that is currently selected in a device context fall out of scope. You should reselect the original pen into the device context so that you safely can delete your pen after it is out of the device context.

You also can use the device context’s SaveDC() and RestoreDC() functions to reselect the original pens and release your created pen. (Remember that this action also restores all the other GDI objects you have selected.)

Using Stock Pens

The operating system can lend out a number of common GDI objects to applications; these objects are called stock objects. These include a white pen, a black pen, and a null pen. You can select these straight into a device context by calling the CDC::SelectStockObject() function and passing the WHITE_PEN, BLACK_PEN, or NULL_PEN index values. As usual, the currently selected pen is returned from SelectStockObject().

You also can use the pen’s base class CGdiObject::CreateStockObject() function, passing the same index values to create a stock CPen object.
Note - From Windows NT 5 and Windows 98 onward, you can use a new stock pen object called a DC_PEN. You can change the color of the DC_PEN by using the global ::SetDCPenColor() function, passing the handle of the device context and a COLORREF value. When selected, the DC_PEN draws in the specified color. You also can retrieve this color by calling GetDCPenColor(). At the time of this writing, no equivalent CDC member function exists for this Win32 function.

Drawing with Pens

A number of rendering functions use just pens. Other functions use brushes to fill an area and pens to draw the outline of that area. You also can use the area-filling functions to draw only the outline of a shape by selecting a NULL brush.

The device context stores a current graphics cursor position, and many of the line-drawing functions use their starting point as the current graphics cursor position. You can change this position without drawing a line by using the device context’s MoveTo() function. You can find the current graphics position from GetCurrentPosition().

The LineTo() function draws a line from the current position to the specified position and updates the graphics cursor to the specified endpoint.

You can use the Polyline() function to draw a number of lines from coordinates stored in an array of POINT structures, and you can use PolyPolyLine() to draw a number of independent line sections. The PolylineTo() function performs a task similar to Polyline() but also updates the current graphics cursor. You can plot a number of Bezier splines (special type of curves) by using the PolyBezier() and PolyBezierTo() functions. The PolyDraw() function lets you draw a number of connected lines and Bezier splines.

You can draw elliptical arcs by using Arc(), ArcTo(), and AngleArc(). You can set the device context’s default direction for these arcs to clockwise or counterclockwise by passing AD_CLOCKWISE or AD_COUNTERCLOCKWISE to the device context’s SetArcDirection() function. This value also affects the rendering of chords drawn with the Chord() function. You can find the current direction by the flag returned from GetArcDirection().

You can call the area-filling functions to outline the areas with the current pen, as discussed in the following sections about brushes.

The sample OnDraw() function shown in Listing 6.1 draws a number of shapes using different drawing functions in a variety of pens to produce the output shown in Figure 6.1.

Listing 6.1 An OnDraw() Function Using Different Pens to Create Graphics

```cpp
void CPensDemoView::OnDraw(CDC* pDC)
{
    CPen penRed(PS_SOLID, 5, RGB(255, 0, 0));
    CPen penThick(PS_SOLID, 10, RGB(0, 0, 255));
    CPen penDash(PS_DASH, 1, RGB(0, 128, 0));
    CPen penDot(PS_DOT, 1, RGB(255, 0, 255));
    CPen penDashDot(PS_DASHDOT, 1, RGB(0, 0, 0));

    CRect rcClient;
    GetClientRect(&rcClient);
    int nSaved = pDC->SaveDC(); // No Area Filling
    pDC->SelectStockObject(NULL_BRUSH);
    for(int i=0;i<rcClient.Height()/6;i++)
    {
        int x=rcClient.Width()/(6); i++
        {
            int x=rcClient.Height()/(6); i++
            CRect rcDrw(x*6, y*6, x*6+w, y*6+h);
        }
    }
}
```
rcDrw.DeflateRect(CSize(i,i));
switch(i%6)
{
  case 0:
    pDC->SelectObject(&penThick);
    pDC->MoveTo(rcDrw.TopLeft());
    pDC->LineTo(rcDrw.BottomRight());
    pDC->MoveTo(rcDrw.right,rcDrw.top);
    pDC->LineTo(rcDrw.left,rcDrw.bottom);
    break;
  case 1:
    pDC->SelectObject(&penRed);
    pDC->Ellipse(rcDrw);
    break;
  case 2:
    POINT pts[] = {
      {(short)rcDrw.left,(short)rcDrw.bottom},
      {(short)rcDrw.right,(short)rcDrw.bottom},
      {(short)rcDrw.left+rcDrw.Width()/2,(short)rcDrw.top},
      {(short)rcDrw.left,(short)rcDrw.bottom}};
    pDC->SelectObject(&penDash);
    pDC->Polyline(pts,sizeof(pts)/sizeof(POINT));
    break;
  case 3:
    pDC->SelectObject(&penDot);
    pDC->Rectangle(rcDrw);
    break;
  case 4:
    pDC->SelectObject(&penDashDot);
    pDC->Pie(rcDrw,CPoint(rcDrw.right,rcDrw.top),
             rcDrw.BottomRight());
    break;
  case 5:
    pDC->SelectStockObject(BLACK_PEN);
    pDC->Chord(rcDrw,rcDrw.TopLeft(),
              rcDrw.BottomRight());
    break;
}
pDC->RestoreDC(nSaved);
}

Figure 6.1
Drawing with various pen styles.

Brushes and the CBrush Class

Whereas pens are used to draw lines, brushes are used to fill areas and shapes. The CBrush class wraps the GDI brush object and, like other MFC GDI wrapper classes, is derived from the CGdiObject base class.

You can create brushes that fill an area with a solid color, a pattern from a bitmap, or a hatching scheme. You can use one of three constructor functions to construct brushes initialized with one of these filling techniques. One form lets you pass a COLORREF value for a solid-color brush. Another lets you pass a pointer to a CBitmap object to create a filling pattern. The third form lets you pass an index describing a hatching technique (as shown in Table 6.3) and a COLORREF value to specify the hatching-pattern color.

Table 6.3 Brush Hatching-Pattern Flags

<table>
<thead>
<tr>
<th>Style</th>
<th>Brush Hatching</th>
</tr>
</thead>
<tbody>
<tr>
<td>HS_DIAGCROSS</td>
<td>Diagonal crisscross</td>
</tr>
<tr>
<td>HS_CROSS</td>
<td>Horizontal crisscross</td>
</tr>
<tr>
<td>HS_HORIZONTAL</td>
<td>Horizontal lines</td>
</tr>
<tr>
<td>HS_VERTICAL</td>
<td></td>
</tr>
</tbody>
</table>
Vertical lines

| HS_FDIAGONAL | Bottom-left to top-right lines |
| HS_BDIAGONAL | Top-left to bottom-right lines |

You can use the default constructor and then one of the creation functions—such as `CreateSolidBrush()`, `CreateHatchBrush()`, or `CreatePatternBrush()`—to create the GDI brush object. The `CreateBrushIndirect()` function lets you initialize and pass a pointer to a `LOGBRUSH` structure to create the brush. You can set the style from a number of flag values to indicate what sort of brush you want with the `lbStyle` member of `LOGBRUSH`. The other members are `lbColor` for the desired color and `lbHatch` to specify any hatching-style flags (remember that the same structure is used with geometric pens).

You can use the `CreateDIBPatternBrush()` creation function to create a patterned brush from a *device-independent bitmap* (DIB). You can use a `CreateSysColorbrush()` function to create a brush initialized with one of the current system colors specified by an index value.

---

Warning - Although Windows NT lets you create patterned brushes using any size of bitmap (bigger or equal to 8x8 pixels), Windows 95/98 can create only 8x8 pixel brushes; larger bitmaps are truncated to the top 8x8 pixels.

---

Selecting Brushes into the Device Context

You can select a brush into a device context in the same way you would a pen. The device context’s `SelectObject()` function has an overload that accepts a pointer to a `CBrush` object and returns the previously selected `CBrush` object.

After you select your new brush, it can use any of the drawing functions that draw filled shapes. As with the pen, you must ensure that the brush is selected out of the device context before it falls out of scope or is deleted.

Using Stock Brushes

You can select a number of stock brushes into a device context with the `SelectStockObject()` function, as Table 6.4 shows.

### Table 6.4 Stock Brush Objects

<table>
<thead>
<tr>
<th>Stock Object Flag</th>
<th>Brush Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHITE_BRUSH</td>
<td>White</td>
</tr>
<tr>
<td>LT_GRAY_BRUSH</td>
<td>Light gray</td>
</tr>
<tr>
<td>GRAY_BRUSH</td>
<td>Gray</td>
</tr>
<tr>
<td>DKGRAY_BRUSH</td>
<td>Dark gray</td>
</tr>
<tr>
<td>BLACK_BRUSH</td>
<td>Black</td>
</tr>
<tr>
<td>HOLLOW_BRUSH, NULL_BRUSH</td>
<td>Transparent, like the NULL pen</td>
</tr>
</tbody>
</table>

You also can use the `CGdiObject` base class’s `CreateStockObject()` function to create a `CBrush` object initialized from a stock object flag.

Drawing with Brushes
You can use a number of device-context member functions with a brush to draw filled shapes. These shapes are outlined with the currently selected pen.

Some functions, such as `FillRect()`, let you pass a CBrush object by pointer to draw with that brush. Other functions draw a rectangle from a COLORREF value, such as `FillSolidRect()`. Most functions, however, use the currently selected brush, such as the `Rectangle()` function. `FillRect()`, `Rectangle()`, and `FillSolidRect()` all render a filled rectangle from a set of rectangle coordinates. The `RoundRect()` function lets you draw rectangles with rounded corners.

You can draw filled polygons by using the `Polygon()` or `PolyPolygon()` function, which draws a single polygon or number of filled polygons, respectively. These two functions are similar to the `Polyline()` functions, because they take an array of POINT variables to define the polygon’s vertices. When drawing polygons, you can specify one of two filling techniques: winding or alternate. These techniques affect which areas of a crisscrossing polygon are designated as filled. You can set these modes by using the `SetPolyFillMode()` function and passing the WINDING or ALTERNATE flag value.

You can draw circles and ellipses using the `Ellipse()` function, pie segments using `Pie()`, and chord sections with the `Chord()` function.

The `FloodFill()` function lets you fill an area bounded by a specified color with the current brush. You can use `ExtFloodFill()` to fill an area bounded by a specific color or fill an area of a specific color with the current brush.

**MFC Classes for GDI Operations**

Many of the GDI functions use sophisticated MFC helper classes that let you specify coordinates in a two-dimensional coordinate system. These classes also let you manipulate and perform arithmetic operations on sets of coordinates.

The `CPoint`, `CSize`, and `CRect` classes can hold point coordinates, size coordinates, or rectangles, respectively. They wrap `POINT`, `SIZE`, and `RECT` structures and can be cast into each of these structures. Overloaded operator functions accept these structures as parameters, so arithmetic between the various classes can be performed interchangeably.

This section examines these MFC coordinate-manipulation and storage classes in more detail.

**The CPoint Class**

The `CPoint` class wraps a `POINT` structure to hold a single two-dimensional coordinate specified by its x and y member variables. Many of the drawing functions can use `CPoint` objects as parameters.

You can construct a `CPoint` object from two integers specifying the x and y coordinates, another `CPoint` object, a `SIZE` structure, or a `DWORD` value (using the low and high words). You can call the `Offset()` function to add an offset value to move the specified coordinates passing x and y coordinates, another `CPoint` object, or a `SIZE` structure.

Mostly, however, you probably would use the `CPoint`’s operator overloads to add, subtract, or compare two `CPoint` objects. Many of these operators let you specify `POINT`, `SIZE`, or `RECT` structures as parameters and return `SIZE`, `POINT`, or `BOOL` values as appropriate.

Table 6.5 lists the available `CPoint` operator functions.

**Table 6.5 CPoint Operator Overload Functions**
### The CSize Class

The `CSize` class wraps a `SIZE` structure. This structure stores a two-dimensional size as `cx` and `cy` integer members, which are declared as public accessible members. You can construct a `CSize` object from two integers, a `SIZE` structure, a `POINT` structure, or a `DWORD` value.

You will find that many of the `CPoint` and `CRect` functions and operators can take `SIZE` structure objects as parameters for arithmetic manipulation. Some of the GDI and Windows functions require `SIZE` structures—usually to specify the size of an object (such as a window).

The `CSize` class implements the operators listed in Table 6.6.

### Table 6.6 CSize Operator Overload Functions

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>=</code></td>
<td>Copies the <code>SIZE</code>.</td>
</tr>
<tr>
<td><code>+=</code></td>
<td>Adds a <code>SIZE</code> value.</td>
</tr>
<tr>
<td><code>-=</code></td>
<td>Subtracts a <code>SIZE</code> value.</td>
</tr>
<tr>
<td><code>+</code></td>
<td>Adds <code>POINT</code>, <code>SIZE</code>, or <code>RECT</code> values.</td>
</tr>
<tr>
<td><code>-</code></td>
<td>Subtracts <code>POINT</code>, <code>SIZE</code>, or <code>RECT</code> values.</td>
</tr>
<tr>
<td><code>==</code></td>
<td>Compares for equality with another <code>SIZE</code> or <code>CSize</code> object.</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>Compares for inequality with another <code>SIZE</code> or <code>CSize</code> object.</td>
</tr>
</tbody>
</table>

### The CRect Class

`CRect` is probably the most sophisticated of the coordinate storing classes. `CRect` wraps a `RECT` structure that exposes its coordinates as two coordinate pairs. These pairs correspond to the top-left and bottom-right points in a rectangle and are accessible from the `RECT` structure as the `top`, `left`, `bottom`, and `right` member integers.

You also can obtain these coordinates as `CPoint` objects returned by the `TopLeft()` and `BottomRight()` functions. The `CenterPoint()` function is quite useful, because it returns a `CPoint` object representing the center of the rectangle. You can find the width and height of the rectangle by using the `Width()` and `Height()` functions, and you can find the size represented as the `CSize` object returned from the `Size()` function.
You can increase and decrease the size of the rectangle by using the \texttt{InflateRect()} and \texttt{DeflateRect()} functions, which are overloaded to take a variety of parameter types. The \texttt{OffsetRect()} function lets you move the position of the rectangle by an amount specified by a \texttt{SIZE}, \texttt{POINT}, or pair of integers.

Intersection, union, and subtraction functions also are provided by \texttt{IntersectRect()}, \texttt{UnionRect()}, and \texttt{SubtractRect()}.

- \texttt{IntersectRect()} makes the current \texttt{CRect} object the intersection rectangle of two source rectangles (where they overlap), as shown in Figure 6.2; the resulting \texttt{CRect} object is shaded.

Figure 6.2
Determining the intersection of two rectangles.

- \texttt{UnionRect()} makes the \texttt{CRect} object the union of two source rectangles (the smallest rectangle that encloses both), as shown in Figure 6.3; the resulting \texttt{CRect} object is shaded.

Figure 6.3
Determining the union of two rectangles.

- \texttt{SubtractRect()} sets its first \texttt{RECT} parameter’s coordinates to the smallest rectangle that is not intersected by two overlapping rectangles (where the second rectangle encloses the first), as shown in Figure 6.4; the resulting \texttt{CRect} object is shaded.

Figure 6.4
Subtracting one rectangle from another.

Each of these functions requires that you previously called \texttt{NormalizeRect()}. \texttt{NormalizeRect()} ensures that the top-left point coordinates are lower than the bottom-right coordinates; if they are not, it sets them to be so. You may find that some operations leave the coordinates in a condition where the width or height calculations may give negative results. If this is the case, you can call \texttt{NormalizeRect()} to fix them.

If you are implementing drag and drop or want to perform bounds checking, you will find the \texttt{PtInRect()} member function useful. This returns \texttt{TRUE} if the specified point lies within the rectangle.

Several operator overload functions are available, as Table 6.7 shows.

**Table 6.7 \texttt{CRect} Operator Overload Functions**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Copies the \texttt{RECT}.</td>
</tr>
<tr>
<td>+=</td>
<td>Offsets the rectangle by a \texttt{POINT} or \texttt{SIZE} value, or inflates each side by the coordinates of a \texttt{RECT}.</td>
</tr>
<tr>
<td>-=</td>
<td>Offsets or deflates the rectangle.</td>
</tr>
<tr>
<td>+</td>
<td>Offsets or inflates the rectangle, returning a \texttt{RECT} result.</td>
</tr>
<tr>
<td>-</td>
<td>Offsets or deflates the rectangle, returning a \texttt{RECT} result.</td>
</tr>
<tr>
<td>&amp;=</td>
<td>Sets the rectangle to the intersection of the rectangle’s current value, and another \texttt{RECT} or \texttt{CRect}.</td>
</tr>
<tr>
<td></td>
<td>=</td>
</tr>
</tbody>
</table>
The CRgn Class and Clipping

The CRgn class is another CgdiObject-derived GDI object wrapper class that wraps the HRGN GDI handle. You can retrieve the HRGN handle by casting a CRgn object.

Regions are used primarily for clipping; you can select a region into a device context by using the device context’s SelectObject() or SelectClipRegion() function. Thereafter, all GDI-rendering functions performed in the device context are clipped to the specified region.

You can specify complex regions that have overlapped borders, simple regions that do not have overlapping borders, or null regions where there is no specified region data. Many of the functions use and return the type of region specified by the COMPLEXREGION, SIMPLEREGION, and NULLREGION flag values. If an error occurs when combining or selecting regions, the ERROR flag is returned.

To create an initialized CRgn object, you must construct it by using the default CRgn constructor and then use one of the creation functions, such as CreateRectRgn() for rectangles, CreateEllipticRgn() for ellipses, CreatePolygonRgn() for polygons, or CreateRoundRgn() for rounded rectangles. Other functions are available that initialize from structures or create multiple polygons.

You can combine two regions by using the CombineRgn() function. The two source regions specified by pointers to CRgn objects are combined using a logic operation specified by a flag as the last parameter. This flag value can be any one of the following: RGN_AND, RGN_OR, RGN_XOR, RGN_COPY, or RGN_DIFF (the non-overlapping areas).

By using the PtInRegion() or RectInRegion() function, you can perform complex bound checking.

You can use the OffsetRgn() function to move a region and CopyRgn() to copy one region from another. You can test for equivalence between two regions with the EqualRegion() function.

---

Warning - You should bear in mind that regions (especially complex regions) can be fairly memory hungry and are limited to a maximum coordinate displacement of 32,767 pixels. Like other GDI objects, regions can be deleted after you use them by using the DeleteObject() function, which is called automatically when a CRgn object is deleted.

---

Working with Fonts

Much of Windows graphical output consists of text in a variety of fonts. Like pens, brushes, and regions, a GDI handle (HFONT) also represents font object instances.

The fonts and their related rendering data for each character make a considerable memory footprint for each instance of a font object. To reduce the overhead of memory and processing time required to load, initialize, and store the details for each instance of a font, Windows uses a font mapper.

Whenever a new font object is created, the font mapper looks for the nearest match of the requested characteristics from the list of installed fonts. It then constructs a font that is as close as possible to the one you have requested.
Fonts and the CFont Class

The MFC CFont class is a wrapper for the HFONT GDI object. To create a font using CFont, you first must construct a CFont object with the default constructor and then call one of the font-creation routines.

The quickest and easiest way to create a font is with CFont’s CreatePointFont() function. You can pass a desired point size (in tenths of a point), a typeface name, and optionally a reference device context pointer to create the font. You should pass a pointer to the device context to create an accurate point-size match. Otherwise, the default screen device context is used (which is inaccurate when printing).

You also can use CreatePointFontIndirect() to create a font from the lfHeight member set in a LOGFONT structure.

A much more sophisticated font-creation function is CreateFont(), which lets you specify a huge number of required attributes for the font. These attributes specify the width, height, escapement, orientation, weight, effects, character set, clipping, rendering precision, font family, pitch, and typeface with its 14 parameters. The orientation and escapement attributes are closely related. The orientation attribute refers to the rotation angle of each individual character. The escapement is similar, but refers to the entire line of text. On Windows 95/98, these values must be set to the same value. On Windows NT/2000, these values may be set independently of each other.

Each of these parameters has several associated flag values to hone the type of font required. The font mapper then uses all these attributes to try to find the best matching font for the requested specifications.

You can initialize the LOGFONT structure with these specifications and pass a pointer to the LOGFONT structure to the CreateFontIndirect() function, which then creates a font and returns a font handle. This creation form is especially useful when you have enumerated the currently installed fonts with one of the font-enumeration functions, such as EnumFonts(). The callback functions for these enumerators are passed a pointer to a LOGFONT structure so that you can create a font object instance directly from it.

After a LOGFONT structure is initialized, you can fill it with the details of a font by using the GetLogFont() function.

The CFont class also has an HFONT operator to retrieve the underlying GDI handle when cast as an HFONT.

Selecting Fonts into the Device Context

As with the other GDI objects, a font must be selected into the device context before you can use it, and the previously selected font is restored after use. You must ensure that the font is not currently selected in a device context when it is deleted.

After the font is selected, it is used whenever any of the text-output functions are called.

Stock Fonts

A number of stock fonts can be selected with the SelectStockObject() function, as Table 6.8 shows.

Table 6.8 Stock Font Objects

<table>
<thead>
<tr>
<th>Stock Object Flag</th>
<th>Font Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI_FIXED_FONT</td>
<td>ANSI fixed pitch</td>
</tr>
<tr>
<td>ANSI_VAR_FONT</td>
<td>ANSI variable pitch</td>
</tr>
<tr>
<td>SYSTEM_FONT</td>
<td>Current Windows system font</td>
</tr>
<tr>
<td>DEVICE_DEFAULT_FONT</td>
<td>Device’s default font</td>
</tr>
</tbody>
</table>
Device Context Font Interrogation Functions

The device context has a number of member functions that let you retrieve information regarding the currently selected font.

With variable-pitch fonts, you may need to know about the average and specific widths of characters when rendered in a specific device context. You can use the `GetCharWidth()` function to fill an array with the widths of individual characters for non-TrueType fonts or `GetABCCharWidths()` for TrueType fonts.

You can find the average widths, height, and many other specific elements of a font from its `TEXTMETRICS` structure. The device context’s `GetOutputTextMetrics()` function fills such a structure for you with details of the currently selected font.

You can retrieve the typeface name of the currently selected font by calling the `GetTextFace()` function and passing a `CString` object by reference to receive the typeface name.

*Kerning pairs* specify the width between two characters placed together in a variable-pitch font. These values often may be negative, because characters such as ‘l’ and ‘i’ are thin and can be placed close together. The `GetKerningPairs()` function can fill an array of `KERNINGPAIR` structures to retrieve this information.

The `GetOutlineTextMetrics()` function returns an array of `OUTLINETEXTMETRICS` structures. These structures are full of information about TrueType fonts.

Text-Rendering Functions

Many text-rendering functions are available that perform slightly different jobs in different circumstances. The simplest is `TextOut()`, which lets you specify an x and y coordinate and a `CString` holding the text to display.

The device context’s text-alignment flags adjust the position of the text relative to the given coordinate. You can adjust these flags by using the `SetTextAlign()` function and passing a combination of the alignment flags, such as `TA_TOP`, `TA_CENTER`, `TA_RIGHT`, `TA_LEFT`, `TA_BASELINE`, and `TA_BOTTOM`. You can combine this flag value with the `TA_NOUPDATECP` and `TA_UPDATECP` to not update or update the current graphic cursor position after the text rendering. The corresponding `GetTextAlign()` returns the current flag settings.

You can change the color of the rendered text with the device context’s `SetTextColor()` and `SetBkColor()` functions to set the foreground and background colors to a specified `COLORREF` value. You can make the background behind the text transparent or opaque by passing the `TRANSPARENT` or `OPAQUE` flag to the `SetBkMode()` function.

The `ExtTextOut()` function lets you clip text to a specified rectangle by using the `ETO_CLIPPED` flag. You can supply an array of spacing values to separate the individual character cells to `ExtTextOut()`.

You can use `TabbedTextout()` to display a text string with embedded tab characters. These tabs then are expanded to positions specified by an array of consecutive tab positions relative to a specified origin.

The `DrawText()` function performs some quite advanced text formatting, such as word wrapping and justification. You can pass a combination of formatting flag values, such as `DT_WORDBREAK`, `DT_LEFT`, `DT_RIGHT`, `DT_TOP`, and `DT_CENTER` (and many others).

You can call the `GrayString()` function to draw grayed text using a specific graying brush and optionally pass a pointer to your own text-rendering function.

You often will want to know the dimensions required by a text string without actually rendering it. You can find these dimensions by using the `GetOutputTextExtent()`, `GetTabbedTextExtent()`, or `GetOutputTabbedTextExtent()` function.
function. These functions use the device context to calculate and return a CSize object holding the size required to render the text using the currently selected font and device-context settings.

Creating and Loading Bitmaps

Images are used extensively in Windows; a bitmap instance is another low-level GDI object represented by the HBITMAP handle.

Before you can use bitmaps in your application, you generally create a number of bitmap resources that are bound with your .EXE or .DLL when linked. These bitmaps then are loaded from the current module ready for drawing.

Note - You should not confuse the "loading" terminology used here and expressed by the LoadBitmap() function with loading and saving to specific disk files. However, DIBs may be stored in individual disk files. (DIBs are discussed later in this chapter in the "Creating a Device-Independent Bitmap Class" section.) The LoadBitmap() function loads bitmap resources only from the application’s executable module and dynamic link libraries (DLLs).

Creating a Bitmap Resource with the Resource Editor

You can create a new bitmap resource from the Insert Resource dialog box in the Visual Studio Resource Editor. You can change the size, the resource ID, and the source bitmap filename from the Bitmap Properties dialog box.

You then can draw the bitmap using the Resource Editor drawing tools, as Figure 6.5 shows.

Figure 6.5
Editing a bitmap resource.

You also can open BMP, JPEG, GIF, and many other common formats and copy and paste the image into a new bitmap.

When you compile the program, the bitmap is bundled into the resulting executable file.

Loading a Bitmap

Before you can use a bitmap, you must load the bitmap resource from the executable file. The CBitmap class is another CgdiObject-derived class, and it wraps the HBITMAP GDI object handle. You can construct a CBitmap object using its default constructor and then load a specific bitmap by calling its LoadBitmap() function, passing a resource ID (or resource name) that identifies the specific bitmap.

If the bitmap is found and loaded from the executable module, LoadBitmap() returns a TRUE value, and the HBITMAP handle is valid. You also can use the LoadMappedBitmap() function to load color-mapped bitmaps and specify a set of COLORMAP structures to define the remapped COLORREF values.

You can load a system stock bitmap with the LoadOEMBitmap() function passing a flag value that loads a few standard bitmaps.

Creating Bitmaps

You can create a bitmap image without an associated resource by using one of the bitmap-creation functions. The CreateBitmap() function lets you create a bitmap in memory and specify its width, height, number of bit planes (color planes), number of color bits per pixel, and (optionally) an array of short values to initialize the bitmap.
Warning - When specifying the color planes or color bits, you should set one or the other to 1; otherwise, you will get a multiple of these values!

The CreateBitmapIndirect() function lets you specify these values in a BITMAP structure instead of through parameter settings.

You should use the CreateCompatibleBitmap() function to create bitmaps that are compatible with specific devices (such as the current display configuration). To do this, you must pass a pointer to a valid device context along with a width and height. This creates a bitmap that has a compatible color depth with a device attached to the device context.

Drawing with Bitmaps

After you create a GDI bitmap object you can draw with it using a memory device context and then copy it to a destination device context. A bitmap is copied to a device using the Bit Block Transfer functions. These functions are commonly known as BitBlt (pronounced bit-blit) functions. These functions, such as BitBlt() and StretchBlt() are discussed in the next section.

As you read earlier in this chapter, you can create a compatible memory device context by using the CreateCompatibleDC() function.

When a memory device context is created, it is initialized with a monochrome bitmap. You then should select your custom bitmap into the device context (saving the pointer returned to the old bitmap) and perform the copy into the screen device context. After you finish, you must select the original monochrome bitmap back into the device context (thus deselecting your custom bitmap) before deleting the CBitmap object or calling DeleteObject() to destroy the GDI bitmap object.

Bitmap Copying

After you select a valid bitmap into a memory-based device context, you can use the BitBlt() device-context function to transfer all or part of the image onto a display-based (or printer-based) device context. BitBlt() lets you specify the source device context (your memory DC), the source coordinates to copy from (relative to the memory DC), the destination coordinates (relative to the destination DC), and the width and height describing the area to copy.

You also can specify a raster operation flag that can be used to invert the image during copying, merge the source with the destination image, just copy the source to the destination, or perform a number of other logical operations between the source and destination during the copy.

The StretchBlt() function lets you specify a source width and height, as well as a destination width and height. The image then expands or shrinks to fit the destination width and height. This method provides a fast and easy way to perform zoom operations. You can change the technique used to copy the image area by passing a flag value to the SetStretchBltMode() function. This lets you perform color-averaging stretch copies that give a better image representation (at the expense of speed) or allow a number of different stretching techniques.

You can use the MaskBlt() function to perform an image copy from a source device context through the holes in a mask provided by a monochrome bitmap into the destination device context.

The following code fragment shows how to create a memory device context, load a resource bitmap, and select it into the memory device context. Then you use StretchBlt() to stretch it so that it fills the view.
void CBitmapdemoView::OnDraw(CDC* pDC)
{
    // Create a compatible memory device context
    CDC dcMemory;
    dcMemory.CreateCompatibleDC(pDC);

    // Load and select the bitmap resource
    CBitmap bmImage;
    bmImage.LoadBitmap(IDB_TSTBITMAP);
    CBitmap *pbmOriginal = dcMemory.SelectObject(&bmImage);

    CRect rcClient;
    GetClientRect(rcClient);
    pDC->StretchBlt(0,0,rcClient.Width(),rcClient.Height(),
        &dcMemory,0,0,48,48,SRCCOPY);
    pDC->SelectObject(pbmOriginal);
}

Creating a Device-Independent Bitmap Class

The bitmaps discussed so far are called device-dependent bitmaps because they rely on the current display device context for their palette colors and are dependent on the current display resolution to determine the displayed size.

A device-independent bitmap (DIB) not only stores the bitmap image but also uses a BITMAPINFO structure to store the color depth, sizing information, and colors. The first part of the structure consists of one of the various header structures followed by an array of RGBQUAD structures that store the colors used in the bitmap. You can create a BITMAPINFO and associate it with a bitmap image to form a DIB.

After you create a DIB, you can transfer it onto different machines and display the same image, represented to the best capabilities of various devices while preserving the original size and colors.

This section shows some of the steps required to create, display, and store a DIB. The sample code builds a DIB class derived from CBitmap to extend the bitmap functionality into a DIB.

Creating a DIB

To create a DIB, you first must initialize a BITMAPINFOHEADER structure or one of its more modern counterparts, such as the BITMAPV5HEADER (which requires Windows NT5 or Windows 98).

Warning - The BITMAPV5HEADER lets you specify advanced compression techniques such as JPEG. To support this capability, you need Windows NT 5 or Windows 98. If your DIB will be used on older platforms, you should use BITMAPV4HEADER for Windows NT 4 or Windows 95, or BITMAPINFOHEADER for older platforms.

You can initialize a number of member variables in the DIB header (BITMAPINFOHEADER structure) to specify the DIB width, height, number of bit planes, bits per pixel, compression technique, pixels per meter in the x and y dimensions, and the actual number of colors used. The more modern versions of the BITMAPINFOHEADER structure (such as BITMAPV5HEADER) offer even more members to specify advanced compression techniques and gamma corrections.

Specifying the pixels per meter provides an important piece of information that lets you render the DIB to the correct size of various devices. By using the information about the resolution of the device, or by using one of the device-context mapping modes (especially the MM_LOMETRIC or MM_HIRESOMETRIC mode), you can render the image to the correct size.
After specifying the header details, you should initialize the `RGBQUAD` structures that specify each color used in the bitmap. You may want to use the colors from the palette selected in a specific device context by using the `SetDIBColorTable()` function. This function fills the `RGBQUAD` array for you from the specified device-context handle.

The following code fragment shows a class definition for a DIB manipulation class:

```cpp
class CDIB : public CObject
{
public:
    CDIB();
    virtual ~CDIB();
    BOOL CreateDIB(DWORD dwWidth, DWORD dwHeight, int nBits);
    BOOL CreateDIBFromBitmap(CDC* pDC);
    void InitializeColors();
    int GetDIBCols() const;
    VOID* GetDIBBitArray() const;
    BOOL CopyDIB(CDC* pDestDC, int x, int y);

public:
    CBitmap       m_bmBitmap;

private:
    LPBITMAPINFO    m_pDIB;
};
```

You will notice that the class uses a `BITMAPINFO` pointer (`m_pDIB`) to the buffer containing the DIB and has an embedded device-dependent bitmap (`m_bmBitmap`) for transferring the image from an existing bitmap resource (discussed in the next section).

You then could implement the `CreateDIB()` and associated functions to allocate the memory for a DIB and initialize the `BITMAPINFO` structure, like this:

```cpp
CDIB::CDIB() : m_pDIB(NULL)
{
}

CDIB::~CDIB()
{
    if (m_pDIB) delete m_pDIB;
}

BOOL CDIB::CreateDIB(DWORD dwWidth, DWORD dwHeight, int nBits)
{
    if (m_pDIB) return FALSE;

    const DWORD dwcBihSize = sizeof(BITMAPINFOHEADER);
    // Calculate the memory required for the DIB
    DWORD dwSize = dwcBihSize +
        (2>>nBits) * sizeof(RGBQUAD) +
        ((nBits * dwWidth) * dwHeight);

    m_pDIB = (LPBITMAPINFO)new BYTE[dwSize];
    if (!m_pDIB) return FALSE;
    m_pDIB->bmiHeader.biSize = dwcBihSize;
    m_pDIB->bmiHeader.biWidth = dwWidth;
    m_pDIB->bmiHeader.biHeight = dwHeight;
    m_pDIB->bmiHeader.biBitCount = nBits;
    m_pDIB->bmiHeader.biPlanes = 1;
    m_pDIB->bmiHeader.biCompression = BI_RGB;
    m_pDIB->bmiHeader.biXPelsPerMeter = 1000;
    m_pDIB->bmiHeader.biYPelsPerMeter = 1000;
    m_pDIB->bmiHeader.biClrUsed = 0;
    m_pDIB->bmiHeader.biClrImportant = 0;

    InitializeColors();
    return TRUE;
}
```
You will notice that \texttt{dwSize} is calculated from the size of the \texttt{BITMAPINFOHEADER} structure, the size of the \texttt{RGBQUAD} array required for the specified color depth, and the size of the resulting bitmap buffer.

The \texttt{BITMAPINFOHEADER} structure is initialized with dimensions of 1,000x1,000 pixels per meter and uses the specified width, height, and color depth. The compression flag \texttt{biCompression} lets you specify the type of compression used in the DIB image buffer. The \texttt{BI_RGB} flag shown in the code sample indicates that no compression is used.

The colors then could be initialized (all to black) like this:

```cpp
void CDIB::InitializeColors()
{
    if (!m_pDIB) return;
    // This just initializes all colors to black
    LPRGBQUAD lpColors =
        (LPRGBQUAD)(m_pDIB+m_pDIB->bmiHeader.biSize);
    for(int i=0;i<GetDIBCols();i++)
    {
        lpColors[i].rgbRed=0;
        lpColors[i].rgbBlue=0;
        lpColors[i].rgbGreen=0;
        lpColors[i].rgbReserved=0;
    }
}
```

You might want to initialize a specific set of \texttt{RGBQUAD} colors, depending on the requirements of your DIB.

Creating a DIB from a Device-Dependent Bitmap

After you set the color values for the DIB, you can set the pixels for the image directly into a buffer (usually stored immediately after the \texttt{BITMAPINFO} structure) from within your code, or from an existing device-dependent bitmap.

Although it is not necessary, it usually is desirable to keep the bitmap image information directly after the \texttt{BITMAPINFO} structure, as shown earlier in the \texttt{CreateDIB()} implementation.

The following helper functions can help you retrieve the number of colors calculated from the color depth in \texttt{BITMAPINFOHEADER}, a pointer to the bitmap image buffer calculated from the size of the header structure, and the following color value array:

```cpp
int CDIB::GetDIBCols() const
{
    if (!m_pDIB) return 0;
    return (2>>m_pDIB->bmiHeader.biBitCount);
}

VOID* CDIB::GetDIBBitArray() const
{
    if (!m_pDIB) return FALSE;
    return (m_pDIB + m_pDIB->bmiHeader.biSize +
            GetDIBCols() * sizeof(RGBQUAD));
}
```

You can find the size and color-depth information from a device-dependent bitmap by using the \texttt{CBitmap} class’s \texttt{GetBitmap()} function. This function fills a \texttt{BITMAP} structure (passed by pointer) with the details about a specific GDI bitmap object.

You then can copy these details along with the extra DIB information into the DIB’s \texttt{BITMAPINFOHEADER} to create a DIB with the same width, height, and color depth as the device-dependent bitmap.

The DIB color-value information then can be initialized with the \texttt{SetDIBColorTable()} from a reference device context.
After the DIB header and color values are initialized from the bitmap, it only remains to copy the image bits themselves. The `GetDIBits()` function can perform this task for you. You just need to pass the DIB information, a reference device context, and the GDI bitmap handle; `GetDIBits()` then copies the entire image bitmap into your supplied buffer.

If you want to perform the reverse operation of copying a DIB bitmap to a device-dependent bitmap, you can use the corresponding `SetDIBits()` function. You can even create an `HBITMAP` GDI object initialized from a DIB directly by using the global `CreateDIBitmap()` function.

The following implementation for the `CDIB` class’s `CreateDIBFromBitmap()` function demonstrates these functions using the device-dependent `m_bmBitmap` member. The `CreateDIBFromBitmap()` function assumes that this `CBitmap` member was initialized previously through a `CreateBitmap()` or `LoadBitmap()`:

```cpp
BOOL CDIB::CreateDIBFromBitmap(CDC* pDC)
{
    if (!pDC) return FALSE;
    HDC hDC = pDC->GetSafeHdc();
    BITMAP bimapInfo;
    m_bmBitmap.GetBitmap(&bimapInfo);
    if (!CreateDIB(bimapInfo.bmWidth,bimapInfo.bmHeight,
                   bimapInfo.bmBitsPixel)) return FALSE;
    LPRGBQUAD lpColors =
        (LPRGBQUAD)(m_pDIB+m_pDIB->bmiHeader.biSize);
    SetDIBColorTable(hDC,0,GetDIBCols(),lpColors);
    // This implicitly assumes that the source bitmap
    // is at the 1 pixel per mm resolution
    BOOL bSuccess = (GetDIBits(hDC,(HBITMAP)m_bmBitmap,
                                0,bimapInfo.bmHeight,GetDIBBitArray(),
                                m_pDIB,DIB_RGB_COLORS) > 0);
    return bSuccess;
}
```

You will notice the `DIB_RGB_COLORS` flag also is passed to the `GetDIBits()` function. You can set this flag to either `DIB_RGB_COLORS` to indicate that the color values are literal RGB values, or `DIB_RGB_PAL` to specify color index positions in the device context’s current palette.

You might call the `CreateDIBFromBitmap()` function after loading a resource-based bitmap like this:

```cpp
#include "dib.h"
CDIB g_DIB;

void CBitmapdemoView::OnInitialUpdate()
{
    CView::OnInitialUpdate();
    CClientDC dc(this);
    g_DIB.m_bmBitmap.LoadBitmap(IDB_TSTBITMAP);
    g_DIB.CreateDIBFromBitmap(&dc);
}
```

After calling the `CreateDIBFromBitmap()` function, the embedded DIB will be initialized in the memory pointed at by the `m_pDIB` pointer.

### Drawing with a DIB

You can use several GDI functions to draw from a DIB directly into a normal device context. These functions are similar to the device-dependent blit functions you saw earlier.

The global `SetDIBitsToDevice()` function copies the image from a DIB buffer directly to a specified device context at a specified position. You can specify the number of lines to copy, as well as the width, height, and position of the source DIB image. If the operation succeeds, `SetDIBitsToDevice()` returns the number of scan lines copied.
Warning - With the introduction of Windows 98 and Windows NT 5, SetDIBitsToDevice() can render JPEG image types. The device-context driver may not support this type, however. If it doesn’t, SetDIBitsToDevice() returns zero, and GetLastError() returns a GDI_ERROR value.

You can use StretchDIBits() in a way similar to StretchBlt() to stretch the DIB bitmap to the required size. This function often is useful when you are trying to preserve the original bitmap size. You can use the DIB’s pixels-per-meter values to calculate the correct destination size, as well as a specific mapping mode to draw the image at a specific size.

The following example shows an implementation of the CopyDIB() function for the CDIB class defined previously. This function uses StretchDIBits() to render the DIB into a device context that is provided.

This method maintains the standard size of the DIB by setting the mapping mode to MM_LOMETRIC. In this mapping mode, the logical unit size is 0.1 mm, so coordinates specified by the GDI functions will be converted by the device context to represent the correct specified size on the destination device. Therefore the coordinates specified in the destination width and height should be multiplied by 10 so that each DIB pixel represents 1 mm to maintain the 1,000 pixel-per-meter specification:

```c
BOOL CDIB::CopyDIB(CDC* pDestDC, int x, int y)
{
    if (!m_pDIB || !pDestDC) return FALSE;

    int nOldMapMode = pDestDC->SetMapMode(MM_LOMETRIC);
    BOOL bOK = StretchDIBits(pDestDC->GetSafeHdc(),
                                x, y,
                                m_pDIB->bmiHeader.biWidth * 10, // Dest Width
                                m_pDIB->bmiHeader.biHeight * -10, // Dest Height
                                0, 0,
                                m_pDIB->bmiHeader.biWidth, // Source Width
                                m_pDIB->bmiHeader.biHeight, // Source Height
                                GetDIBBitArray(), m_pDIB, DIB_RGB_COLORS, SRCCOPY) > 0;

    pDestDC->SetMapMode(nOldMapMode);
    return bOK;
}
```

You could invoke this code from your application’s view class in the OnDraw() and OnPrint() functions, after previously creating the DIB from the loaded bitmap resource (as shown earlier), like this:

```c
void CBitmapdemoView::OnPrint(CDC* pDC, CPrintInfo* pInfo)
{
    g_DIB.CopyDIB(pDC, 50, -50);
}

void CBitmapdemoView::OnDraw(CDC* pDC)
{
    g_DIB.CopyDIB(pDC, 50, -50);
}
```

Regardless of the device context passed to CopyDIB() and the ultimate destination device, the DIB should be rendered to the same size and with the same colors to the best capabilities of the device.

Warning - Mapping modes used with display monitors are usually very inaccurate, so the size of a displayed DIB may be physically wrong on a video display.

Printer device contexts provide very accurate physical unit mapping, however, so they should print the image to the correct size.
Summary

In this chapter, you saw how Windows maintains a common interface to applications to produce consistent output across a range of devices, such as screens, printers, and plotters.

You saw how the CDC class wraps an HDC device-context handle and provides member functions for the vast majority of API functions that use the device context.

You learned how device-context objects can be created directly for various devices, such as screens and printers, or obtained directly from windows to give you access to the client or entire rendering area of the window.

You looked at how to use the special CDC-derived classes that are used to get access to client areas, to handle window painting, or to play metafiles.

You saw how to create the basic GDI-rendering objects, such as pens and brushes, to perform a wide range of graphical drawing operations using a variety of line-drawing and area-filling styles and techniques. You also saw how to use the various coordinate storing classes and regions with the drawing functions to perform coordinate manipulation and clipping of the rendered output.

You saw how to draw text in a number of fonts using the various text-drawing functions of the device context. You also looked at how to create fonts, as well as the role of the font mapper in the creation of fonts.

Finally, you learned how to load device-dependent bitmaps from the application’s resources or create and use them with memory-device contexts. You examined the limitations of device-dependent bitmaps and how to overcome these limitations with device-independent bitmaps with their additional information structures.