Chapter 1

**Hello, MFC**

A few short years ago, the person learning to program Microsoft Windows for the first time had a limited number of programming tools to choose from. C was the language spoken by the Windows Software Development Kit (SDK), and alternative Windows programming environments such as Microsoft Visual Basic hadn't arrived on the scene. Most Windows applications were written in C, and the fledgling Windows programmer faced the daunting task not only of learning the ins and outs of a new operating system but also of getting acquainted with the hundreds of different application programming interface (API) functions that Windows supports.

Today many Windows programs are still written in C. But the variety of Windows programming environments available means that commercial-quality Windows programs can be written in C, C++, Pascal, BASIC, and a number of other languages. Moreover, C++ has all but replaced C as the professional Windows programmer's language of choice because the sheer complexity of Windows, coupled with the wide-ranging scope of the Windows API, cries out for an object-oriented programming language. Many Windows programmers have concluded that C++ offers a compelling alternative to C that, combined with a class library that abstracts the API and encapsulates the basic behavior of windows and other objects in reusable classes, makes Windows programming simpler. And an overwhelming majority of C++ programmers have settled on the Microsoft Foundation Class library, better known by the acronym MFC, as their class library of choice. Other Windows class libraries are available, but only MFC was written by the company that writes the operating system. MFC is continually updated to incorporate the latest changes to Windows itself, and it provides a comprehensive set of classes representing everything from windows to ActiveX controls in order to make the job of writing Windows applications easier.

If you're coming to MFC from a traditional Windows programming environment such as C and the Windows SDK, you're already familiar with many of the concepts you need to know to understand Windows programming with MFC. But if you're coming from a character-oriented environment such as MS-DOS or UNIX, you'll find that Windows programming is fundamentally different from anything you've done before. This chapter begins with an overview of the Windows programming model and a peek under the hood at how Windows applications work. It continues with an introduction to MFC. After the preliminaries are out of the way, you'll develop your very first Windows application—one that uses MFC to create a resizeable window containing the message "Hello, MFC."

Chapter 2

**Drawing in a Window**

If you've been around PCs for a while, you probably remember what graphics programming was like before Microsoft Windows came along. If you were lucky, you had a decent graphics library with routines like *DrawLine* and *DrawCircle* to draw graphics primitives for you. If you weren't so lucky, you probably spent a lot of time writing your own output routines and tweaking them to shave off a few microseconds here and there. And whether it was your code or someone else's doing the drawing, you knew that when a new graphics standard emerged—in those days, that meant whenever IBM introduced a new graphics adapter like the EGA or the VGA—you'd be scrambling to support the latest hardware. That invariably meant buying an updated version of the graphics library, adding new code to your own routines, or writing a driver for the new video card. For the graphics programmer, the platform was a moving target that never seemed to stand still for very long. And even if you did manage to draw a bead on the video hardware, you still had plenty of work to do to adapt your code to work with printers and other output devices.

Windows changed all that by bringing to the PC platform something it sorely needed: a device-independent graphics output model. In Windows, the graphics code you write will work on any video adapter for which a Windows driver is available. These days, that's just about every adapter on the planet. And to a large extent, the same code that sends output to the screen will also work with printers and other hardcopy devices. This one-size-fits-all approach to graphics programming has a number of advantages, chief among them the fact that programmers can now spend their time developing code for their applications rather than code for the hardware their applications will run on. Moreover, you no longer need third-party graphics libraries in order to do your work because Windows provides a wide assortment of graphics API functions that do everything from draw lines to create complex clipping regions that serve as stencils for other output routines.

The part of Windows responsible for graphics output is the Graphics Device Interface, or GDI. The GDI provides a number of services that an application can call. Together, these services constitute a powerful and robust graphics programming language whose richness rivals that of some third-party graphics libraries. MFC works on top of the graphics API and codifies the interface with C++ classes that represent the various components of the Windows GDI.

Now that you know how to create a window, it's time to do something with that window. The Hello application in [Chapter 1](mk:@MSITStore:C:\Program%20Files%20(x86)\MSPress\BooksOnline\Programming%20Windows%20with%20MFC%20Second%20Edition\progmfc2.chm::/ch01a.htm) used *CDC::DrawText* to output text to a window. *DrawText* is just one of many member functions that the *CDC* class provides for text and graphics output. This chapter looks at the *CDC* class and its derivative classes in more detail and introduces three of the most commonly used GDI primitives: pens, brushes, and fonts. It also demonstrates how to add scroll bars to a window.

# The Windows GDI

In a single-tasking environment such as MS-DOS, the name of the game when it comes to screen output is "anything goes." A running application is free to do just about whatever it wants whenever it wants, whether that involves drawing a line on the screen, reprogramming the adapter's color palette, or switching to another video mode. In a windowed, multitasking environment such as Windows, programs can't be afforded such freedom because the output from program A must be protected from the output of program B. First and foremost, this means that each program's output must be restricted to its own window. The GDI uses a simple mechanism to make sure every program that draws in a window plays by the rules. That mechanism is the device context.

When a Windows program draws to a screen, a printer, or any other output device, it doesn't output pixels directly to the device. Instead, it draws to a logical "display surface" represented by a device context (DC). A device context is a data structure deep inside Windows that contains fields describing everything the GDI needs to know about the display surface, including the physical device with which it is associated and assorted state information. Before it draws anything on the screen, a Windows program acquires a device context handle from the GDI. It then passes that handle back to the GDI each time it calls a GDI output function. Without a valid device context handle, the GDI won't draw the first pixel. And through the device context, the GDI can make sure that everything the program draws is clipped to a particular area of the screen. Device contexts play a huge role in making the GDI device-independent because, given a handle to a device context, the same GDI functions can be used to draw to a diverse assortment of output devices.

When you program Windows with MFC, the device context has even greater significance. In addition to serving as the key that unlocks the door to output devices, a device context object encapsulates the GDI functions that programs use to generate output. In MFC, you don't grab a handle to a device context and call GDI output functions, at least not directly; instead, you create a device context object and call its member functions to do your drawing. MFC's *CDC* class wraps a Windows device context and the GDI functions that require a device context handle into one convenient package, and *CDC*-derived classes such as *CPaintDC* and *CClientDC* represent the different types of device contexts that Windows applications use.

## The MFC Device Context Classes

One way to get a device context in an MFC application is to call *CWnd::GetDC*, which returns a pointer to a *CDC* object representing a Windows device context. A device context pointer acquired with *CWnd::GetDC* should be released with *CWnd::ReleaseDC* when drawing is completed. The following code gets a *CDC* pointer from *GetDC*, does some drawing, and calls *ReleaseDC* to release the device context:

|  |
| --- |
| CDC\* pDC = GetDC ();  // Do some drawing  ReleaseDC (pDC); |

If the same code were to appear in an *OnPaint* handler, you would use *CWnd::BeginPaint* and *CWnd::EndPaint* in place of *GetDC* and *ReleaseDC* to ensure proper handling of the WM\_PAINT message:

|  |
| --- |
| PAINTSTRUCT ps;  CDC\* pDC = BeginPaint (&ps);  // Do some drawing  EndPaint (&ps); |

The GDI also supports *metafiles,* which store sequences of GDI commands that can be "played back" to produce physical output. To acquire a device context for a metafile's output, you would use yet another set of functions to obtain and release the *CDC* pointer. And to acquire a *CDC* pointer for a device context that permits drawing anywhere in the window (as opposed to one that permits drawing only in the window's client area), you would call *CWnd::GetWindowDC* rather than *GetDC* and release the device context with *ReleaseDC*.

To save you the trouble of having to remember which functions to call to acquire and release a device context (and to help ensure that a device context is properly released when the message handler that uses the device context ends), MFC provides the *CDC*-derived classes listed in the following table.

**Special-Purpose Device Context Classes**

|  |  |
| --- | --- |
| **Class Name** | **Description** |
| CPaintDC | For drawing in a window's client area (*OnPaint* handlers only) |
| CClientDC | For drawing in a window's client area (anywhere but *OnPaint*) |
| CWindowDC | For drawing anywhere in a window, including the nonclient area |
| CMetaFileDC | For drawing to a GDI metafile |

These classes are designed to be instantiated directly. Each class's constructor and destructor call the appropriate functions to get and release the device context so that using a device context is no more complicated than this:

|  |
| --- |
| CPaintDC dc (this);  // Do some drawing |

The pointer passed to the class constructor identifies the window that the device context pertains to.

When a device context object is constructed on the stack, its destructor is called automatically when the object goes out of scope. And when the destructor is called, the device context is released back to Windows. The only time you need to be concerned about releasing one of these device contexts yourself is when (and if) you create a device context object on the heap with *new*, as shown here:

|  |
| --- |
| CPaintDC\* pDC = new CPaintDC (this); |

In this case, it's important to execute a

|  |
| --- |
| delete pDC; |

statement before the function that created the device context ends so that the object's destructor will be called and the device context will be released. On some occasions, it's useful to create a device context on the heap rather than on the stack, but generally you're a lot better off creating device context objects on the stack and letting the compiler do the deleting for you.

### The *CPaintDC* Class

MFC's *CPaintDC* class lets you paint in a window's client area in response to WM\_PAINT messages. You should use it only in *OnPaint* handlers and never anywhere else. WM\_PAINT messages are different from all other Windows messages in one very important respect: If the handler fails to call the Windows *::BeginPaint* and *::EndPaint* functions (or the MFC equivalents, *CWnd::BeginPaint* and *CWnd::EndPaint*), the message will not be removed from the message queue no matter how much drawing you do. Consequently, the application will get stuck processing the same WM\_PAINT message over and over. *CPaintDC* virtually ensures that this won't happen by calling *::BeginPaint* and *::EndPaint* from its constructor and destructor, respectively.

### The *CClientDC* and *CWindowDC* Classes

Windows programs don't always limit their painting to *OnPaint*. If you write an application that draws a circle on the screen whenever a mouse button is clicked, you'll probably want to paint the circle immediately—when you receive the button-click message—rather than wait for the next WM\_PAINT message.

That's what MFC's *CClientDC* class is for. *CClientDC* creates a client-area device context that can be used outside *OnPaint*. The following message handler uses *CClientDC* and two *CDC* member functions to draw an X connecting the corners of the window's client area when the left mouse button is clicked:

|  |
| --- |
| void CMainWindow::OnLButtonDown (UINT nFlags, CPoint point)  {  CRect rect;  GetClientRect (&rect);  CClientDC dc (this);  dc.MoveTo (rect.left, rect.top);  dc.LineTo (rect.right, rect.bottom);  dc.MoveTo (rect.right, rect.top);  dc.LineTo (rect.left, rect.bottom);  } |

*left*, *right*, *top*, and *bottom* are public member variables defined in MFC's *CRect* class. They store the coordinates of the rectangle's four sides. *MoveTo* and *LineTo* are line-drawing functions that *CClientDC* inherits from *CDC*. You'll learn more about these two functions in a moment.

For the rare occasions on which you'd like to paint not only the window's client area but also the nonclient area (the title bar, the window border, and so on), MFC provides the *CWindowDC* class. *CWindowDC* is similar to *CClientDC*, but the device context it represents encompasses everything within the window's borders. Programmers sometimes use *CWindowDC* for unusual effects such as custom-drawn title bars and windows with rounded corners. In general, you won't need *CWindowDC* very often. If you do want to do your own painting in a window's nonclient area, you can trap WM\_NCPAINT messages with an *OnNcPaint* handler to determine when the nonclient area needs to be painted. Unlike *OnPaint*, an *OnNcPaint* handler need not (and should not) call *BeginPaint* and *EndPaint*.

For the even rarer occasions on which a program requires access to the entire screen, you can create a *CClientDC* or *CWindowDC* object and pass its constructor a NULL pointer. The statements

|  |
| --- |
| CClientDC dc (NULL);  dc.Ellipse (0, 0, 100, 100); |

draw a circle in the upper left corner of the screen. Screen capture programs frequently use full-screen DCs to access the whole screen. Needless to say, drawing outside your own window is a very unfriendly thing to do unless you have a specific reason for doing so.

## Device Context Attributes

When you draw to the screen with *CDC* output functions, certain characteristics of the output aren't specified in the function call but are obtained from the device context itself. When you call *CDC::DrawText*, for example, you specify the text string and the rectangle in which the string will appear, but you don't specify the text color or the font because both are attributes of the device context. The following table lists some of the most useful device context attributes and the *CDC* functions used to access them.

**Key Device Context Attributes**

|  |  |  |  |
| --- | --- | --- | --- |
| **Attribute** | **Default** | **Set with** | **Get with** |
| Text color | Black | CDC::SetTextColor | CDC::GetTextColor |
| Background color | White | CDC::SetBkColor | CDC::GetBkColor |
| Background mode | OPAQUE | CDC::SetBkMode | CDC::GetBkMode |
| Mapping mode | MM\_TEXT | CDC::SetMapMode | CDC::GetMapMode |
| Drawing mode | R2\_COPYPEN | CDC::SetROP2 | CDC::GetROP2 |
| Current position | (0,0) | CDC::MoveTo | CDC::GetCurrentPosition |
| Current pen | BLACK\_PEN | CDC::SelectObject | CDC::SelectObject |
| Current brush | WHITE\_BRUSH | CDC::SelectObject | CDC::SelectObject |
| Current font | SYSTEM\_FONT | CDC::SelectObject | CDC::SelectObject |

Different *CDC* output functions use device context attributes in different ways. For example, when you draw a line with *LineTo*, the current pen determines the line's color, width, and style (solid, dotted, dashed, and so on). Similarly, when you draw a rectangle with the *Rectangle* function, the GDI borders the rectangle with the current pen and fills the rectangle with the current brush. All text output functions use the current font. The text color and the background color control the colors used when text is output. The text color determines the color of the characters, and the background color determines what color is used to fill behind them. The background color is also used to fill the gaps between line segments when dotted or dashed lines are drawn with the *LineTo* function and to fill the open areas between hatch marks painted by a hatch brush. If you'd like the background color to be ignored entirely, you can set the background mode to "transparent," like this:

|  |
| --- |
| dc.SetBkMode (TRANSPARENT); |

Inserting this statement before the call to *DrawText* in Chapter 1's Hello program eliminates the white rectangle surrounding "Hello, MFC" that's visible when the window background color is nonwhite.

The *CDC* function you'll use more than any other to modify the attributes of a device context is *SelectObject*. The following six items are GDI objects that can be selected into a device context with *SelectObject*:

* Pens
* Brushes
* Fonts
* Bitmaps
* Palettes
* Regions

In MFC, pens, brushes, and fonts are represented by the classes *CPen*, *CBrush*, and *CFont*. (Bitmaps, palettes, and regions are discussed in [Chapter 15](mk:@MSITStore:C:\Program%20Files%20(x86)\MSPress\BooksOnline\Programming%20Windows%20with%20MFC%20Second%20Edition\progmfc2.chm::/ch15a.htm).) Unless you call *SelectObject* to change the current pen, brush, or font, the GDI uses the device context's defaults. The default pen draws solid black lines 1 pixel wide. The default brush paints solid white. The default font is a rather plain proportional font with a height of roughly 12 points. You can create pens, brushes, and fonts of your own and select them into a device context to change the attributes of the output. To draw a solid red circle with a 10-pixel-wide black border, for example, you can create a black pen 10 pixels wide and a red brush and select them into the device context with *SelectObject* before calling *Ellipse*. If *pPen* is a pointer to a *CPen* object, *pBrush* is a pointer to a *CBrush* object, and *dc* represents a device context, the code might look like this:

|  |
| --- |
| dc.SelectObject (pPen);  dc.SelectObject (pBrush);  dc.Ellipse (0, 0, 100, 100); |

*SelectObject* is overloaded to accept pointers to objects of various types. Its return value is a pointer to the object of the same type that was previously selected into the device context.

Each time you acquire a device context from Windows, its attributes are reset to the defaults. Consequently, if you want to use a red pen and a blue brush to paint your window in response to WM\_PAINT messages, you must select them into the device context each time *OnPaint* is called and a new *CPaintDC* object is created. Otherwise, the default pen and brush will be used. If you'd like to avoid reinitializing a device context every time you use it, you can save its state with the *CDC::SaveDC* function and restore it the next time around with *CDC::RestoreDC*. Another option is to register a custom WNDCLASS that includes the CS\_OWNDC style, which causes Windows to allocate to each instance of your application its own private device context that retains its settings. (A related but seldom used WNDCLASS style, CS\_CLASSDC, allocates a "semiprivate" device context that is shared by all windows created from the same WNDCLASS.) If you select a red pen and a blue brush into a private device context, they remain selected until they're explicitly replaced.

## The Drawing Mode

When the GDI outputs pixels to a logical display surface, it doesn't simply output pixel colors. Rather, it combines the colors of the pixels that it's outputting with the colors of the pixels at the destination using a combination of Boolean operations. The logic that's employed depends on the device context's current drawing mode, which you can change with *CDC::SetROP2* (short for "Set Raster Operation To"). The default drawing mode is R2\_COPYPEN, which does, in fact, copy pixels to the display surface. But there are 15 other drawing modes to choose from, as shown in the table below. Together, these drawing modes represent all the possible operations that can be performed by combining the Boolean primitives AND, OR, XOR, and NOT.

Why would you ever need to change the drawing mode? Suppose you want to draw a line not by copying pixels to the display surface but by inverting the colors of the pixels already there. It's easy to do; you just set the drawing mode to R2\_NOT before drawing the line:

|  |
| --- |
| dc.SetROP2 (R2\_NOT);  dc.MoveTo (0, 0);  dc.LineTo (100, 100); |

This little trick might be more useful than you think, because it's a great way to rubber-band lines and rectangles. You'll see an example of what I mean in [Chapter 3](mk:@MSITStore:C:\Program%20Files%20(x86)\MSPress\BooksOnline\Programming%20Windows%20with%20MFC%20Second%20Edition\progmfc2.chm::/ch03a.htm).

**GDI Drawing Modes**

|  |  |
| --- | --- |
| **Drawing Mode** | **Operation(s) Performed** |
| R2\_NOP | dest = dest |
| R2\_NOT | dest = NOT dest |
| R2\_BLACK | dest = BLACK |
| R2\_WHITE | dest = WHITE |
| R2\_COPYPEN | dest = src |
| R2\_NOTCOPYPEN | dest = NOT src |
| R2\_MERGEPENNOT | dest = (NOT dest) OR src |
| R2\_MASKPENNOT | dest = (NOT dest) AND src |
| R2\_MERGENOTPEN | dest = (NOT src) OR dest |
| R2\_MASKNOTPEN | dest = (NOT src) AND dest |
| R2\_MERGEPEN | dest = dest OR src |
| R2\_NOTMERGEPEN | dest = NOT (dest OR src) |
| R2\_MASKPEN | dest = dest AND src |
| R2\_NOTMASKPEN | dest = NOT (dest AND src) |
| R2\_XORPEN | dest = src XOR dest |
| R2\_NOTXORPEN | dest = NOT (src XOR dest) |
|  |  |

## The Mapping Mode

Without a doubt, the aspect of GDI programming that new Windows programmers find the most confusing is the mapping mode. Simply put, the *mapping mode* is the attribute of the device context that governs how logical coordinates are translated into device coordinates. *Logical coordinates* are the coordinates you pass to *CDC* output functions. *Device coordinates* are the corresponding pixel positions within a window. When you call the *Rectangle* function like this:

|  |
| --- |
| dc.Rectangle (0, 0, 200, 100); |

you're not necessarily telling the GDI to draw a rectangle that's 200 pixels wide and 100 pixels tall; you're telling it to draw a rectangle that's 200 units wide and 100 units tall. In the default mapping mode, MM\_TEXT, it just so happens that 1 unit equals 1 pixel. But in other mapping modes, logical units are translated into device units differently. In the MM\_LOENGLISH mapping mode, for example, 1 unit equals 1/100 of an inch. Therefore, drawing a rectangle that measures 200 units by 100 units in the MM\_LOENGLISH mapping mode produces a 2-inch by 1-inch rectangle. Using a non-MM\_TEXT mapping mode is a convenient way to scale your output so that sizes and distances are independent of the output device's physical resolution.

Windows supports eight different mapping modes. Their properties are summarized in the following table.

**GDI Mapping Modes**

|  |  |  |
| --- | --- | --- |
| **Mapping Mode** | **Distance Corresponding to One Logical Unit** | **Orientation of the x and y Axes** |
| MM\_TEXT | 1 pixel |  |
| MM\_LOMETRIC | 0.1 mm |  |
| MM\_HIMETRIC | 0.01 mm |  |
| MM\_LOENGLISH | 0.01 in. |  |
| MM\_HIENGLISH | 0.001 in. |  |
| MM\_TWIPS | 1/1440 in. (0.0007 in.) |  |
| MM\_ISOTROPIC | User-defined (*x* and *y* scale identically) | User-defined |
| MM\_ANISOTROPIC | User-defined (*x* and *y* scale independently) | User-defined |

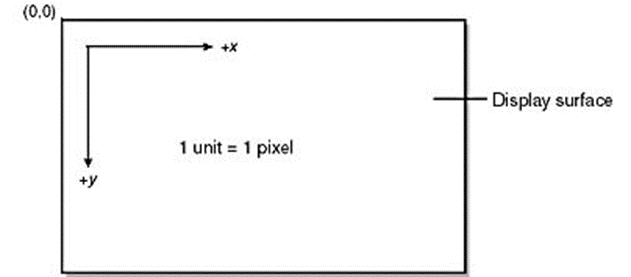
When you draw in the MM\_TEXT mapping mode, you're using the coordinate system shown in Figure 2-1. The origin is in the upper left corner of the window, the positive *x* axis points to the right, the positive *y* axis points downward, and 1 unit equals 1 pixel. If you switch to one of the "metric" mapping modes—MM\_LOENGLISH, MM\_HIENGLISH, MM\_LOMETRIC, MM\_HIMETRIC, or MM\_TWIPS—the *y* axis flips so that positive *y* points upward and logical units are scaled to represent real distances rather than raw pixel counts. The origin, however, remains in the upper left corner. One thing to remember when using a metric mapping mode is that you must use negative *y* values if you want to see your output. The statement

|  |
| --- |
| dc.Rectangle (0, 0, 200, 100); |

draws a 200-pixel by 100-pixel rectangle in the MM\_TEXT mapping mode. The same statement produces no output in the MM\_LOENGLISH mapping mode because positive *y* coordinates lie outside the visible part of the window. To make the rectangle visible, you must negate the *y* coordinates, as shown here:

|  |
| --- |
| dc.Rectangle (0, 0, 200, -100); |

If you switch to a non-MM\_TEXT mapping mode and suddenly your application's output is no longer visible, check the sign of your *y* coordinates. Positive *y* coordinates will be the problem almost every time.



**Figure 2-1.** *The MM\_TEXT coordinate system.*

The default mapping mode is MM\_TEXT. If you want to use one of the other mapping modes, you must call *CDC::SetMapMode* to change the mapping mode. The following statements switch to the MM\_LOMETRIC mapping mode and draw an ellipse whose major axis is 5 centimeters long and whose minor axis measures 3 centimeters:

|  |
| --- |
| dc.SetMapMode (MM\_LOMETRIC);  dc.Ellipse (0, 0, 500, -300); |

You can see that there's really nothing tricky about mapping modes. Things get slightly more complicated when you use the MM\_ISOTROPIC and MM\_ANISOTROPIC modes and when you do hit-testing on objects drawn in non-MM\_TEXT mapping modes, but even that doesn't have to be difficult. The MM\_ISOTROPIC and MM\_ANISOTROPIC mapping modes are discussed in the next section.

One thing to keep in mind when you use the metric mapping modes is that on display screens, 1 logical inch usually doesn't equal 1 physical inch. In other words, if you draw a line that's 100 units long in the MM\_LOENGLISH mapping mode, the line probably won't be exactly 1 inch long. The reason? Windows doesn't know the physical resolution of your monitor—the number of dots per inch (dpi) it's capable of displaying horizontally and vertically. (This might change in a future version of Windows.) The same is not true of printers and other hardcopy devices, however. The printer driver knows that a 600 dpi laser printer can print exactly 600 dots per inch, so a 100-unit line drawn in the MM\_LOENGLISH mapping mode will measure exactly 1 inch on the printed page.

## Programmable Mapping Modes

The MM\_ISOTROPIC and MM\_ANISOTROPIC mapping modes differ from the other mapping modes in one important respect: It's you, not Windows, who determines how logical coordinates are converted into device coordinates. For this reason, these mapping modes are sometimes called the "roll-your-own" or "programmable" mapping modes. Want a mapping mode in which 1 unit equals 1 centimeter? No problem: Just use the MM\_ANISOTROPIC mapping mode and set its scaling parameters accordingly.

The most common use for the MM\_ISOTROPIC and MM\_ANISOTROPIC mapping modes is for drawing output that automatically scales to match the window size. The following code fragment uses the MM\_ANISOTROPIC mapping mode to draw an ellipse that touches all four borders of the window in which it is drawn:

|  |
| --- |
| CRect rect;  GetClientRect (&rect);  dc.SetMapMode (MM\_ANISOTROPIC);  dc.SetWindowExt (500, 500);  dc.SetViewportExt (rect.Width (), rect.Height ());  dc.Ellipse (0, 0, 500, 500); |

See how it works? No matter what physical size the window is, you've told Windows that the window's *logical* size is 500 units by 500 units. Therefore, a bounding box that stretches from (0,0) to (500,500) encompasses the entire window. Initializing a device context in this way places the origin at the upper left corner of the window and orients the axes so that positive *x* points to the right and positive *y* points downward. If you'd rather have the *y* axis point upward (as it does in the metric mapping modes), you can reverse its direction by negating the *y* value passed to either *SetWindowExt* or *SetViewportExt*:

|  |
| --- |
| CRect rect;  GetClientRect (&rect);  dc.SetMapMode (MM\_ANISOTROPIC);  dc.SetWindowExt (500, -500);  dc.SetViewportExt (rect.Width (), rect.Height ());  dc.Ellipse (0, 0, 500, -500); |

Now you must use negative *y* coordinates to draw in the window. Only the MM\_ISOTROPIC and MM\_ANISOTROPIC mapping modes allow the directions of the *x* and *y* axes to be reversed. That's why the table in the previous section listed these two mapping modes' axis orientations as user defined.

The only difference between the MM\_ISOTROPIC and MM\_ANISOTROPIC mapping modes is that in the former, the scaling factors for the *x* and *y* directions are always the same. In other words, 100 horizontal units equals the same physical distance as 100 vertical units. Isotropic means "equal in all directions." The MM\_ISOTROPIC mapping mode is ideal for drawing circles and squares. The following code draws a circle that spans the width or height of a window, whichever is smaller:

|  |
| --- |
| CRect rect;  GetClientRect (&rect);  dc.SetMapMode (MM\_ISOTROPIC);  dc.SetWindowExt (500, 500);  dc.SetViewportExt (rect.Width (), rect.Height ());  dc.Ellipse (0, 0, 500, 500); |

As far as Windows is concerned, the window's logical size is once again 500 units by 500 units. But now the GDI takes the output device's aspect ratio into consideration when converting logical units to device units. [Chapter 14](mk:@MSITStore:C:\Program%20Files%20(x86)\MSPress\BooksOnline\Programming%20Windows%20with%20MFC%20Second%20Edition\progmfc2.chm::/ch14a.htm)'s Clock program uses the MM\_ISOTROPIC mapping mode to draw a round clock face and to automatically scale the clock size to the window size. Without the MM\_ISOTROPIC mapping mode, Clock would have to do all of the scaling manually.

Let's talk a bit about the *SetWindowExt* and *SetViewportExt* functions. Officially, *SetWindowExt* sets the "window extents" and *SetViewportExt* sets the "viewport extents." Think of a window as something whose size is measured in logical units and a viewport as something whose size is measured in device units, or pixels. When Windows converts between logical coordinates and device coordinates, it uses a pair of formulas that factor in the window's logical dimensions (the window extents) and its physical dimensions (the viewport extents) as well as the location of the origin. When you set the window extents and viewport extents, you're effectively programming in your own scaling parameters. Generally, the viewport extents are simply the size (in pixels) of the window you're drawing in and the window extents are the window's desired size in logical units.

One caveat regarding the use of *SetWindowExt* and *SetViewportExt* is that in the MM\_ISOTROPIC mapping mode, you should call *SetWindowExt* first. Otherwise, a portion of the window's client area might fall outside the window's logical extents and become unusable. In the MM\_ANISOTROPIC mapping mode, it doesn't matter which are set first—the window extents or the viewport extents.

## Coordinate Conversions

You can translate logical coordinates to device coordinates using the *CDC::LPtoDP* function. Conversely, you can translate device coordinates to logical coordinates with *CDC::DPtoLP*.

Let's say you want to know where the center of a window is in device coordinates. All you have to do is halve the window's pixel width and height. *CWnd::GetClientRect* returns a window's pixel dimensions.

|  |
| --- |
| CRect rect;  GetClientRect (&rect);  CPoint point (rect.Width () / 2, rect.Height () / 2); |

If you want to know where the center point is in MM\_LOENGLISH units, however, you need *DPtoLP*:

|  |
| --- |
| CRect rect;  GetClientRect (&rect);  CPoint point (rect.Width () / 2, rect.Height () / 2);  CClientDC dc (this);  dc.SetMapMode (MM\_LOENGLISH);  dc.DPtoLP (&point); |

When *DPtoLP* returns, *point* holds the coordinates of the center point in logical (that is, MM\_LOENGLISH) coordinates. If, on the other hand, you want to know the pixel coordinates of the point whose MM\_LOENGLISH coordinates are (100,100), you use *LPtoDP*:

|  |
| --- |
| CPoint point (100, 100);  CClientDC dc (this);  dc.SetMapMode (MM\_LOENGLISH);  dc.LPtoDP (&point); |

One situation in which *LPtoDP* and *DPtoLP* are indispensable is when you're performing hit-testing in response to mouse clicks. Mouse clicks are always reported in device coordinates, so if you've drawn a rectangle in MM\_LOENGLISH coordinates and you want to know whether a mouse click occurred inside that rectangle, you must either convert the rectangle's coordinates to device coordinates or convert the click coordinates to logical coordinates. Otherwise, you'll be comparing apples and oranges.

## Moving the Origin

By default, a device context's origin is in the upper left corner of the display surface. Even if you change the mapping mode, the origin remains in the upper left corner. But just as you can change the mapping mode, you can also move the origin. MFC's *CDC* class provides two functions for moving the origin. *CDC::SetWindowOrg* moves the window origin, and *CDC::SetViewportOrg* moves the viewport origin. You'll normally use one but not both. Using both can be very confusing.

Suppose you'd like to move the origin to the center of the window so that you can center what you draw by centering your output around the point (0,0). Assuming that *dc* is a device context object, here's one way to do it:

|  |
| --- |
| CRect rect;  GetClientRect (&rect);  dc.SetViewportOrg (rect.Width () / 2, rect.Height () / 2); |

Here's another way to accomplish the same thing, assuming that you're working in the MM\_LOENGLISH mapping mode:

|  |
| --- |
| CRect rect;  GetClientRect (&rect);  CPoint point (rect.Width () / 2, rect.Height () / 2);  dc.SetMapMode (MM\_LOENGLISH);  dc.DPtoLP (&point);  dc.SetWindowOrg (-point.x, -point.y); |

It's easy to get *SetViewportOrg* and *SetWindowOrg* confused, but the distinction between them is actually quite clear. Changing the viewport origin to (*x*,*y*) with *SetViewportOrg* tells Windows to map the logical point (0,0) to the device point (*x*,*y*). Changing the window origin to (*x*,*y*) with *SetWindowOrg* does essentially the reverse, telling Windows to map the logical point (*x*,*y*) to the device point (0,0)—the upper left corner of the display surface. In the MM\_TEXT mapping mode, the only real difference between the two functions is the signs of *x* and *y*. In other mapping modes, there's more to it than that because *SetViewportOrg* deals in device coordinates and *SetWindowOrg* deals in logical coordinates. You'll see examples of how both functions are used later in this chapter.

As a final example, suppose you're drawing in the MM\_HIMETRIC mapping mode, where 1 unit equals 1/100 of a millimeter, positive *x* points to the right, and positive *y* points upward, and you'd like to move the origin to the lower left corner of the window. Here's an easy way to do it:

|  |
| --- |
| CRect rect;  GetClientRect (&rect);  dc.SetViewportOrg (0, rect.Height ()); |

Now you can draw with positive *x* and *y* values using coordinates relative to the window's lower left corner.

## A Final Word on Coordinate Systems

When you talk about mapping modes, window origins, viewport origins, and other idioms related to the GDI's handling of coordinates, it's easy to get tangled up in the terminology. Understanding the difference between the device coordinate system and the logical coordinate system might help clear some of the cobwebs.

In the device coordinate system, distances are measured in pixels. The device point (0,0) is always in the upper left corner of the display surface, and the positive *x* and *y* axes always point right and downward. The logical coordinate system is altogether different. The origin can be placed anywhere, and both the orientation of the *x* and *y* axes and the scaling factor (the number of pixels that correspond to 1 logical unit) vary with the mapping mode. To be precise, they vary with the window extents and the viewport extents. You can change these extents in the MM\_ISOTROPIC and MM\_ANISOTROPIC mapping modes but not in the other mapping modes.

You'll sometimes hear Windows programmers talk about "client coordinates" and "screen coordinates." Client coordinates are simply device coordinates relative to the upper left corner of a window's client area. Screen coordinates are device coordinates relative to the upper left corner of the screen. You can convert from client coordinates to screen coordinates and vice versa using the *CWnd::ClientToScreen* and *CWnd::ScreenToClient* functions. Why these functions are useful will become apparent to you the first time you call a Windows function that returns screen coordinates and you pass them to a function that requires client coordinates, or vice versa.

## Getting Information About a Device

Sometimes it's helpful to get information about a device before you send output to it. The *CDC::GetDeviceCaps* function lets you retrieve all kinds of information about a device, from the number of colors it supports to the number of pixels it can display horizontally and vertically. The following code initializes *cx* and *cy* to the width and height of the screen, in pixels:

|  |
| --- |
| CClientDC dc (this);  int cx = dc.GetDeviceCaps (HORZRES);  int cy = dc.GetDeviceCaps (VERTRES); |

If the screen resolution is 1,024 by 768, *cx* and *cy* will be set to 1,024 and 768, respectively.

The table below lists some of the parameters you can pass to *GetDeviceCaps* to acquire information about the physical output device associated with a device context. How you interpret the results depends somewhat on the device type. For example, calling *GetDeviceCaps* with a HORZRES parameter for a screen DC returns the screen width in pixels. Make the same call to a printer DC and you get back the width of the printable page, once more in pixels. As a rule, values that imply any kind of scaling (for example, LOGPIXELSX and LOGPIXELSY) return physically correct values for printers and other hardcopy devices but not for screens. For a 600 dpi laser printer, both LOGPIXELSX and LOGPIXELSY return 600. For a screen, both will probably return 96, regardless of the physical screen size or resolution.

Interpreting the color information returned by the NUMCOLORS, BITSPIXEL, and PLANES parameters of *GetDeviceCaps* is a bit tricky. For a printer or a plotter, you can usually find out how many colors the device is capable of displaying from the NUMCOLORS parameter. For a monochrome printer, NUMCOLORS returns 2.

**Useful *GetDeviceCaps* Parameters**

|  |  |
| --- | --- |
| **Parameter** | **Returns** |
| HORZRES | Width of the display surface in pixels |
| VERTRES | Height of the display surface in pixels |
| HORZSIZE | Width of the display surface in millimeters |
| VERTSIZE | Height of the display surface in millimeters |
| LOGPIXELSX | Number of pixels per logical inch horizontally |
| LOGPIXELSY | Number of pixels per logical inch vertically |
| NUMCOLORS | For a display device, the number of static colors; for a printer or plotter, the number of colors supported |
| BITSPIXEL | Number of bits per pixel |
| PLANES | Number of bit planes |
| RASTERCAPS | Bit flags detailing certain characteristics of the device, such as whether it is palettized and whether it can display bitmapped images |
| TECHNOLOGY | Bit flags identifying the device type—screen, printer, plotter, and so on |

However, the color resolution of the screen (the number of colors that can be displayed onscreen simultaneously) is computed by multiplying BITSPIXEL and PLANES and raising 2 to the power of the result, as demonstrated here:

|  |
| --- |
| CClientDC dc (this);  int nPlanes = dc.GetDeviceCaps (PLANES);  int nBPP = dc.GetDeviceCaps (BITSPIXEL);  int nColors = 1 << (nPlanes \* nBPP); |

If this code is executed on a PC equipped with a 256-color video adapter, *nColors* equals 256. Calling *GetDeviceCaps* with a NUMCOLORS parameter, meanwhile, returns not 256 but 20—the number of "static colors" that Windows programs into the video adapter's color palette. I'll have more to say about the color characteristics of screens and video adapters and also about static colors in [Chapter 15](mk:@MSITStore:C:\Program%20Files%20(x86)\MSPress\BooksOnline\Programming%20Windows%20with%20MFC%20Second%20Edition\progmfc2.chm::/ch15a.htm).

I'll use *GetDeviceCaps* several times in this book to adapt the sample programs' output to the physical attributes of the output device. The first use will come later in this chapter, when the screen's LOGPIXELSX and LOGPIXELSY parameters are used to draw rectangles 1 logical inch long and 1/4 logical inch tall in the MM\_TEXT mapping mode.

# Drawing with the GDI

Enough of the preliminaries. By now, you probably feel as if you asked for the time and got an explanation of watchmaking. Everything you've learned so far in this chapter will come in handy sooner or later—trust me. But now let's talk about functions for outputting pixels to the screen.

The functions discussed in the next several sections are by no means all of the available GDI output functions. A full treatment of every one would require a chapter much larger than this one. When you finish reading this chapter, look at the complete list of *CDC* member functions in your MFC documentation. Doing so will give you a better feel for the wide-ranging scope of the Windows GDI and let you know where to go when you need help.

## Drawing Lines and Curves

MFC's *CDC* class includes a number of member functions that you can use to draw lines and curves. The following table lists the key functions. There are others, but these paint a pretty good picture of the range of available line-drawing and curve-drawing functions.

***CDC* Functions for Drawing Lines and Curves**

|  |  |
| --- | --- |
| **Function** | **Description** |
| *MoveTo* | Sets the current position in preparation for drawing |
| *LineTo* | Draws a line from the current position to a specified position and moves the current position to the end of the line |
| *Polyline* | Connects a set of points with line segments |
| *PolylineTo* | Connects a set of points with line segments beginning with the current position and moves the current position to the end of the polyline |
| *Arc* | Draws an arc |
| *ArcTo* | Draws an arc and moves the current position to the end of the arc |
| *PolyBezier* | Draws one or more Bézier splines |
| *PolyBezierTo* | Draws one or more Bézier splines and moves the current position to the end of the final spline |
| *PolyDraw* | Draws a series of line segments and Bézier splines through a set of points and moves the current position to the end of the final line segment or spline |

Drawing a straight line is simple. You just set the current position to one end of the line and call *LineTo* with the coordinates of the other:

|  |
| --- |
| dc.MoveTo (0, 0);  dc.LineTo (0, 100); |

To draw another line that's connected to the previous one, you call *LineTo* again. There's no need to call *MoveTo* a second time because the first call to *LineTo* sets the current position to the end of the line:

|  |
| --- |
| dc.MoveTo (0, 0);  dc.LineTo (0, 100);  dc.LineTo (100, 100); |

You can draw several lines in one fell swoop using *Polyline* or *PolylineTo*. The only difference between the two is that *PolylineTo* uses the device context's current position and *Polyline* does not. The following statements draw a box that measures 100 units to a side from a set of points describing the box's vertices:

|  |
| --- |
| POINT aPoint[5] = { 0, 0, 0, 100, 100, 100, 100, 0, 0, 0 };  dc.Polyline (aPoint, 5); |

These statements draw the same box using *PolylineTo*:

|  |
| --- |
| dc.MoveTo (0, 0);  POINT aPoint[4] = { 0, 100, 100, 100, 100, 0, 0, 0 };  dc.PolylineTo (aPoint, 4); |

When *PolylineTo* returns, the current position is set to the endpoint of the final line segment—in this case, (0,0). If *Polyline* is used instead, the current position is not altered.

Charles Petzold's *Programming Windows* contains an excellent example showing how and why polylines can be useful. The following *OnPaint* function, which is basically just an MFC adaptation of Charles's code, uses *CDC::Polyline* to draw a sine wave that fills the interior of a window:

|  |
| --- |
| #include <math.h>  #define SEGMENTS 500  #define PI 3.1415926    void CMainWindow::OnPaint ()  {  CRect rect;  GetClientRect (&rect);  int nWidth = rect.Width ();  int nHeight = rect.Height ();  CPaintDC dc (this);  CPoint aPoint[SEGMENTS];  for (int i=0; i<SEGMENTS; i++) {  aPoint[i].x = (i \* nWidth) / SEGMENTS;  aPoint[i].y = (int) ((nHeight / 2) \*  (1 - (sin ((2 \* PI \* i) / SEGMENTS))));  }  dc.Polyline (aPoint, SEGMENTS);  } |

You can see the results for yourself by substituting this code for the *OnPaint* function in Chapter 1's Hello program. Note the use of the *CRect* functions *Width* and *Height* to compute the width and height of the window's client area.

An arc is a curve taken from the circumference of a circle or an ellipse. You can draw arcs quite easily with *CDC::Arc*. You just pass it a rectangle whose borders circumscribe the ellipse and a pair of points that specify the endpoints of two imaginary lines drawn outward from the center of the ellipse. The points at which the lines intersect the ellipse are the starting and ending points of the arc. (The lines must be long enough to at least touch the circumference of the ellipse; otherwise, the results won't be what you expect.) The following code draws an arc representing the upper left quadrant of an ellipse that is 200 units wide and 100 units high:

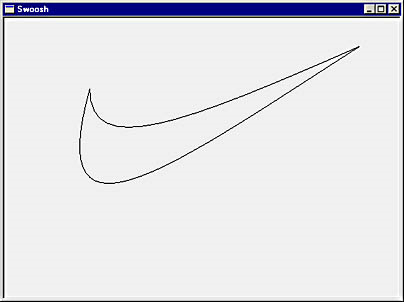
|  |
| --- |
| CRect rect (0, 0, 200, 100);  CPoint point1 (0, -500);  CPoint point2 (-500, 0);  dc.Arc (rect, point1, point2); |

To reverse the arc and draw the upper right, lower right, and lower left quadrants of the ellipse, simply reverse the order in which *point1* and *point2* are passed to the *Arc* function. If you'd like to know where the arc ended (an item of information that's useful when using lines and arcs to draw three-dimensional pie charts), use *ArcTo* instead of *Arc* and then use *CDC::GetCurrentPosition* to locate the endpoint. Be careful, though. In addition to drawing the arc itself, *ArcTo* draws a line from the old current position to the arc's starting point. What's more, *ArcTo* is one of a handful of GDI functions that's not implemented in Windows 98. If you call it on a platform other than Windows NT or Windows 2000, nothing will be output.

If splines are more your style, the GDI can help out there, too. *CDC::PolyBezier* draws Bézier splines—smooth curves defined by two endpoints and two intermediate points that exert "pull." Originally devised to help engineers build mathematical models of car bodies, Bézier splines, or simply "Béziers," as they are more often known, are used today in everything from fonts to warhead designs. The following code fragment uses two Bézier splines to draw a figure that resembles the famous Nike "swoosh" symbol. (See Figure 2-2.)

|  |
| --- |
| POINT aPoint1[4] = { 120, 100, 120, 200, 250, 150, 500, 40 };  POINT aPoint2[4] = { 120, 100, 50, 350, 250, 200, 500, 40 };  dc.PolyBezier (aPoint1, 4);  dc.PolyBezier (aPoint2, 4); |

The curves drawn here are independent splines that happen to join at the endpoints. To draw a continuous curve by joining two or more splines, add three points to the POINT array for each additional spline and increase the number of points specified in *PolyBezier*'s second parameter accordingly.



**Figure 2-2.** *A famous shoe logo drawn with Bézier splines.*

One peculiarity of all GDI line-drawing and curve-drawing functions is that the final pixel is never drawn. If you draw a line from (0,0) to (100,100) with the statements

|  |
| --- |
| dc.MoveTo (0, 0);  dc.LineTo (100, 100); |

the pixel at (0,0) is set to the line color, as are the pixels at (1,1), (2,2), and so on. But the pixel at (100,100) is still the color it was before. If you want the line's final pixel to be drawn, too, you must draw it yourself. One way to do that is to use the *CDC::SetPixel* function, which sets a single pixel to the color you specify.

## Drawing Ellipses, Polygons, and Other Shapes

The GDI doesn't limit you to simple lines and curves. It also lets you draw ellipses, rectangles, pie-shaped wedges, and other closed figures. MFC's *CDC* class wraps the associated GDI functions in handy class member functions that you can call on a device context object or through a pointer to a device context object. The following table lists a few of those functions.

***CDC* Functions for Drawing Closed Figures**

|  |  |
| --- | --- |
| **Function** | **Description** |
| *Chord* | Draws a closed figure bounded by the intersection of an ellipse and a line |
| *Ellipse* | Draws a circle or an ellipse |
| *Pie* | Draws a pie-shaped wedge |
| *Polygon* | Connects a set of points to form a polygon |
| *Rectangle* | Draws a rectangle with square corners |
| *RoundRect* | Draws a rectangle with rounded corners |

GDI functions that draw closed figures take as a parameter the coordinates of a "bounding box." When you draw a circle with the *Ellipse* function, for example, you don't specify a center point and a radius; instead, you specify the circle's bounding box. You can pass the coordinates explicitly, like this:

|  |
| --- |
| dc.Ellipse (0, 0, 100, 100); |

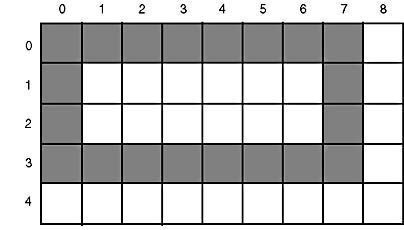
or pass them in a RECT structure or a *CRect* object, like this:

|  |
| --- |
| CRect rect (0, 0, 100, 100);  dc.Ellipse (rect); |

When this circle is drawn, it touches the *x*=0 line at the left of the bounding box and the *y*=0 line at the top, but it falls 1 pixel short of the *x*=100 line at the right and 1 pixel short of the *y*=100 line at the bottom. In other words, figures are drawn from the left and upper limits of the bounding box up to (but not including) the right and lower limits. If you call the *CDC::Rectangle* function, like this:

|  |
| --- |
| dc.Rectangle (0, 0, 8, 4); |

you get the output shown in Figure 2-3. Observe that the right and lower limits of the rectangle fall at *x*=7 and *y*=3, not *x*=8 and *y*=4.



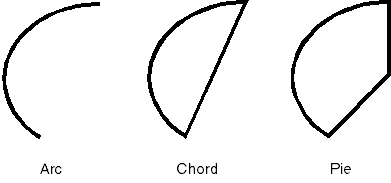
**Figure 2-3.** *A rectangle drawn with the statement dc.Rectangle (0, 0, 8, 4).*

*Rectangle* and *Ellipse* are about as straightforward as they come. You provide the bounding box, and Windows does the drawing. If you want to draw a rectangle that has rounded corners, use *RoundRect* instead of *Rectangle*.

The *Pie* and *Chord* functions merit closer scrutiny, however. Both are syntactically identical to the *Arc* function discussed in the previous section. The difference is in the output. (See Figure 2-4.) *Pie* draws a closed figure by drawing straight lines connecting the ends of the arc to the center of the ellipse. *Chord* closes the figure by connecting the arc's endpoints. The following *OnPaint* handler uses *Pie* to draw a pie chart that depicts four quarterly revenue values:

|  |
| --- |
| #include <math.h>  #define PI 3.1415926    void CMainWindow::OnPaint ()  {  CPaintDC dc (this);  int nRevenues[4] = { 125, 376, 252, 184 };  CRect rect;  GetClientRect (&rect);  dc.SetViewportOrg (rect.Width () / 2, rect.Height () / 2);  int nTotal = 0;  for (int i=0; i<4; i++)  nTotal += nRevenues[i];  int x1 = 0;  int y1 = -1000;  int nSum = 0;  for (i=0; i<4; i++) {  nSum += nRevenues[i];  double rad = ((double) (nSum \* 2 \* PI) / (double) nTotal) + PI;  int x2 = (int) (sin (rad) \* 1000);  int y2 = (int) (cos (rad) \* 1000 \* 3) / 4;  dc.Pie (-200, -150, 200, 150, x1, y1, x2, y2);  x1 = x2;  y1 = y2;  }  } |

Note that the origin is moved to the center of the window with *SetViewportOrg* before any drawing takes place so that the chart will also be centered.



**Figure 2-4.** *Output from the Arc, Chord, and Pie functions.*

## GDI Pens and the *CPen* Class

Windows uses the pen that is currently selected into the device context to draw lines and curves and also to border figures drawn with *Rectangle*, *Ellipse*, and other shape-drawing functions. The default pen draws solid black lines that are 1 pixel wide. To change the way lines are drawn, you must create a GDI pen and select it into the device context with *CDC::SelectObject*.

MFC represents GDI pens with the class *CPen*. The simplest way to create a pen is to construct a *CPen* object and pass it the parameters defining the pen:

|  |
| --- |
| CPen pen (PS\_SOLID, 1, RGB (255, 0, 0)); |

A second way to create a GDI pen is to construct an uninitialized *CPen* object and call *CPen::CreatePen*:

|  |
| --- |
| CPen pen;  pen.CreatePen (PS\_SOLID, 1, RGB (255, 0, 0)); |

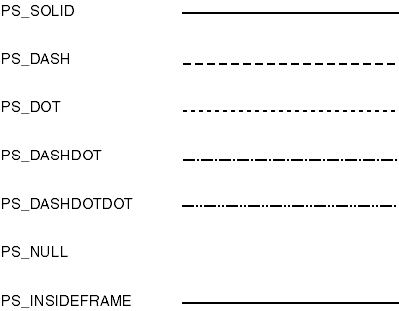
Yet a third method is to construct an uninitialized *CPen* object, fill in a LOGPEN structure describing the pen, and then call *CPen::CreatePenIndirect* to create the pen:

|  |
| --- |
| CPen pen;  LOGPEN lp;  lp.lopnStyle = PS\_SOLID;  lp.lopnWidth.x = 1;  lp.lopnColor = RGB (255, 0, 0);  pen.CreatePenIndirect (&lp); |

LOGPEN's *lopnWidth* field is a POINT data structure. The structure's *x* data member specifies the pen width. The *y* data member is not used.

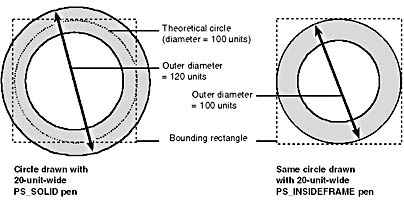
*CreatePen* and *CreatePenIndirect* return TRUE if a pen is successfully created, FALSE if it is not. If you allow *CPen*'s constructor to create the pen, an exception of type *CResourceException* is thrown if the pen can't be created. This should happen only if Windows is critically low on memory.

A pen has three defining characteristics: style, width, and color. The examples above create a pen whose style is PS\_SOLID, whose width is 1, and whose color is bright red. The first of the three parameters passed to *CPen::CPen* and *CPen::CreatePen* specifies the pen style, which defines the type of line the pen draws. PS\_SOLID creates a pen that draws solid, unbroken lines. Other pen styles are shown in Figure 2-5.



**Figure 2-5.** *Pen styles.*

The special PS\_INSIDEFRAME style draws solid lines that stay within the bounding rectangle, or "inside the frame," of the figure being drawn. If you use any of the other pen styles to draw a circle whose diameter is 100 units using a PS\_SOLID pen that is 20 units wide, for example, the actual diameter of the circle, measured across the circle's outside edge, is 120 units, as shown in Figure 2-6. Why? Because the border drawn by the pen extends 10 units outward on either side of the theoretical circle. Draw the same circle with a PS\_INSIDEFRAME pen, and the diameter is exactly 100 units. The PS\_INSIDEFRAME style does not affect lines drawn with *LineTo* and other functions that don't use a bounding rectangle.



**Figure 2-6.** *The PS\_INSIDEFRAME pen style.*

The pen style PS\_NULL creates what Windows programmers refer to as a "NULL pen." Why would you ever want to create a NULL pen? Believe it or not, there are times when a NULL pen can come in handy. Suppose, for example, that you want to draw a solid red circle with no border. If you draw the circle with MFC's *CDC::Ellipse* function, Windows automatically borders the circle with the pen currently selected into the device context. You can't tell the *Ellipse* function that you don't want a border, but you *can* select a NULL pen into the device context so that the circle will have no visible border. NULL brushes are used in a similar way. If you want the circle to have a border but want the interior of the circle to be transparent, you can select a NULL brush into the device context before you draw.

The second parameter passed to *CPen*'s pen-create functions specifies the width of the lines drawn with the pen. Pen widths are specified in logical units whose physical meanings depend on the current mapping mode. You can create PS\_SOLID, PS\_NULL, and PS\_INSIDEFRAME pens of any logical width, but PS\_DASH, PS\_DOT, PS\_DASHDOT, and PS\_DASHDOTDOT pens must be 1 logical unit wide. Specifying a pen width of 0 in any style creates a pen that is 1 pixel wide, no matter what the mapping mode.

The third and final parameter specified when a pen is created is the pen's color. Windows uses a 24-bit RGB color model in which each possible color is defined by red, green, and blue color values from 0 through 255. The higher the value, the brighter the corresponding color component. The RGB macro combines values that specify the three independent color components into one COLORREF value that can be passed to the GDI. The statement

|  |
| --- |
| CPen pen (PS\_SOLID, 1, RGB (255, 0, 0)); |

creates a bright red pen, and the statement

|  |
| --- |
| CPen pen (PS\_SOLID, 1, RGB (255, 255, 0)); |

creates a bright yellow pen by combining red and green. If the display adapter doesn't support 24-bit color, Windows compensates by dithering colors that it can't display directly. Be aware, however, that only PS\_INSIDEFRAME pens greater than 1 logical unit in width can use dithered colors. For the other pen styles, Windows maps the color of the pen to the nearest solid color that can be displayed. You can be reasonably certain of getting the exact color you want on all adapters by sticking to the "primary" colors shown in the table below. These colors are part of the basic palette that Windows programs into the color registers of every video adapter to ensure that a common subset of colors is available to all programs.

**Primary GDI Colors**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Color** | **R** | **G** | **B** | **Color** | **R** | **G** | **B** |
| Black | 0 | 0 | 0 | Light gray | 192 | 192 | 192 |
| Blue | 0 | 0 | 192 | Bright blue | 0 | 0 | 255 |
| Green | 0 | 192 | 0 | Bright green | 0 | 255 | 0 |
| Cyan | 0 | 192 | 192 | Bright cyan | 0 | 255 | 255 |
| Red | 192 | 0 | 0 | Bright red | 255 | 0 | 0 |
| Magenta | 192 | 0 | 192 | Bright magenta | 255 | 0 | 255 |
| Yellow | 192 | 192 | 0 | Bright yellow | 255 | 255 | 0 |
| Dark gray | 128 | 128 | 128 | White | 255 | 255 | 255 |

How do you use a pen once it's created? Simple: You select it into a device context. The following code snippet creates a red pen that's 10 units wide and draws an ellipse with it:

|  |
| --- |
| CPen pen (PS\_SOLID, 10, RGB (255, 0, 0));  CPen\* pOldPen = dc.SelectObject (&pen);  dc.Ellipse (0, 0, 100, 100); |

The ellipse is filled with the color or pattern of the current brush, which defaults to white. To change the default, you need to create a GDI brush and select it into the device context before calling *Ellipse*. I'll demonstrate how to do that in just a moment.

### Extended Pens

If none of the basic pen styles suits your needs, you can use a separate class of pens known as "extended" pens, which the Windows GDI and MFC's *CPen* class support. These pens offer a greater variety of output options. For example, you can create an extended pen that draws a pattern described by a bitmap image or uses a dithered color. You can also exercise precise control over endpoints and joins by specifying the end cap style (flat, round, or square) and join style (beveled, mitered, or rounded). The following code creates an extended pen 16 units wide that draws solid green lines with flat ends. Where two lines meet, the adjoining ends are rounded to form a smooth intersection:

|  |
| --- |
| LOGBRUSH lb;  lb.lbStyle = BS\_SOLID;  lb.lbColor = RGB (0, 255, 0);  CPen pen (PS\_GEOMETRIC ¦ PS\_SOLID ¦ PS\_ENDCAP\_FLAT ¦  PS\_JOIN\_ROUND, 16, &lb); |

Windows places several restrictions on the use of extended pens, not the least of which is that endpoint joins will work only if the figure is first drawn as a "path" and is then rendered with *CDC::StrokePath* or a related function. You define a path by enclosing drawing commands between calls to *CDC::BeginPath* and *CDC::EndPath*, as shown here:

|  |
| --- |
| dc.BeginPath (); // Begin the path definition  dc.MoveTo (0, 0); // Create a triangular path  dc.LineTo (100, 200);  dc.LineTo (200, 100);  dc.CloseFigure ();  dc.EndPath (); // End the path definition  dc.StrokePath (); // Draw the triangle |

Paths are a powerful feature of the GDI that you can use to create all sorts of interesting effects. We'll look more closely at paths—and at the *CDC* functions that use them—in [Chapter 15](mk:@MSITStore:C:\Program%20Files%20(x86)\MSPress\BooksOnline\Programming%20Windows%20with%20MFC%20Second%20Edition\progmfc2.chm::/ch15a.htm).

## GDI Brushes and the *CBrush* Class

By default, closed figures drawn with *Rectangle*, *Ellipse*, and other *CDC* output functions are filled with white pixels. You can change the fill color by creating a GDI brush and selecting it into the device context prior to drawing.

MFC's *CBrush* class encapsulates GDI brushes. Brushes come in three basic varieties: solid, hatch, and pattern. Solid brushes paint with solid colors. If the display hardware won't allow a solid brush color to be displayed directly, Windows simulates the color by dithering colors that *can* be displayed. A hatch brush paints with one of six predefined crosshatch patterns that are similar to ones commonly found in engineering and architectural drawings. A pattern brush paints with a bitmap. The *CBrush* class provides a constructor for each different brush style.

You can create a solid brush in one step by passing a COLORREF value to the *CBrush* constructor:

|  |
| --- |
| CBrush brush (RGB (255, 0, 0)); |

Or you can create a solid brush in two steps by creating an uninitialized *CBrush* object and calling *CBrush::CreateSolidBrush*:

|  |
| --- |
| CBrush brush;  brush.CreateSolidBrush (RGB (255, 0, 0)); |

Both examples create a solid brush that paints in bright red. You can also create a brush by initializing a LOGBRUSH structure and calling *CBrush::CreateBrushIndirect*. As with *CPen* constructors, all *CBrush* constructors that create a brush for you throw a resource exception if the GDI is low on memory and a brush can't be created.

Hatch brushes are created by passing *CBrush*'sconstructor both a hatch index and a COLORREF value or by calling *CBrush::CreateHatchBrush*. The statement

|  |
| --- |
| CBrush brush (HS\_DIAGCROSS, RGB (255, 0, 0)); |

creates a hatch brush that paints perpendicular crosshatch lines oriented at 45-degree angles, as do these statements:

|  |
| --- |
| CBrush brush;  brush.CreateHatchBrush (HS\_DIAGCROSS, RGB (255, 0, 0)); |

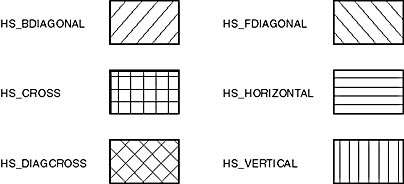
HS\_DIAGCROSS is one of six hatch styles you can choose from. (See Figure 2-7.) When you paint with a hatch brush, Windows fills the space between hatch lines with the default background color (white) unless you change the device context's current background color with *CDC::SetBkColor* or turn off background fills by changing the background mode from OPAQUE to TRANSPARENT with *CDC::SetBkMode*. The statements

|  |
| --- |
| CBrush brush (HS\_DIAGCROSS, RGB (255, 255, 255));  dc.SelectObject (&brush);  dc.SetBkColor (RGB (192, 192, 192));  dc.Rectangle (0, 0, 100, 100); |

draw a rectangle 100 units square and fill it with white crosshatch lines drawn against a light gray background. The statements

|  |
| --- |
| CBrush brush (HS\_DIAGCROSS, RGB (0, 0, 0));  dc.SelectObject (&brush);  dc.SetBkMode (TRANSPARENT);  dc.Rectangle (0, 0, 100, 100); |

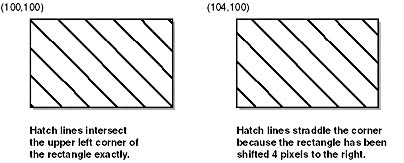
draw a black crosshatched rectangle against the existing background.



**Figure 2-7.** *Hatch brush styles.*

### The Brush Origin

One attribute of a device context that you should be aware of when using dithered brush colors or hatch brushes is the brush origin. When Windows fills an area with a hatched or dithered brush pattern, it tiles an 8-pixel by 8-pixel pattern horizontally and vertically within the affected area. By default, the origin for this pattern, better known as the *brush origin,* is the device point (0,0)—the screen pixel in the upper left corner of the window. This means that a pattern drawn in a rectangle that begins 100 pixels to the right of and below the origin will be aligned somewhat differently with respect to the rectangle's border than a pattern drawn in a rectangle positioned a few pixels to the left or right, as shown in Figure 2-8. In many applications, it doesn't matter; the user isn't likely to notice minute differences in brush alignment. However, in some situations it matters a great deal.



**Figure 2-8.** *Brush alignment.*

Suppose you're using a hatch brush to fill a rectangle and you're animating the motion of that rectangle by repeatedly erasing it and redrawing it 1 pixel to the right or the left. If you don't reset the brush origin to a point that stays in the same position relative to the rectangle before each redraw, the hatch pattern will "walk" as the rectangle moves.

The solution? Before selecting the brush into the device context and drawing the rectangle, call *CGdiObject::UnrealizeObject* on the brush object to permit the brush origin to be moved. Then call *CDC::SetBrushOrg* to align the brush origin with the rectangle's upper left corner, as shown here:

|  |
| --- |
| CPoint point (x1, y1);  dc.LPtoDP (&point);  point.x %= 8;  point.y %= 8;  brush.UnrealizeObject ();  dc.SetBrushOrg (point);  dc.SelectObject (&brush);  dc.Rectangle (x1, y1, x2, y2); |

In this example, *point* is a *CPoint* object that holds the logical coordinates of the rectangle's upper left corner. *LPtoDP* is called to convert logical coordinates into device coordinates (brush origins are always specified in device coordinates), and a modulo-8 operation is performed on the resulting values because coordinates passed to *SetBrushOrg* should fall within the range 0 through 7. Now the hatch pattern will be aligned consistently no matter where in the window the rectangle is drawn.

## Drawing Text

You've already seen one way to output text to a window. The *CDC::DrawText* function writes a string of text to a display surface. You tell *DrawText* where to draw its output by specifying both a formatting rectangle and a series of option flags indicating how the text is to be positioned within the rectangle. In Chapter 1's Hello program, the statement

|  |
| --- |
| dc.DrawText (\_T ("Hello, MFC"), -1, &rect,  DT\_SINGLELINE ¦ DT\_CENTER ¦ DT\_VCENTER); |

drew "Hello, MFC" so that it was centered in the window. *rect* was a rectangle object initialized with the coordinates of the window's client area, and the DT\_CENTER and DT\_VCENTER flags told *DrawText* to center its output in the rectangle both horizontally and vertically.

*DrawText* is one of several text-related functions that are members of MFC's *CDC* class. Some of the others are listed in the table below. One of the most useful is *TextOut*, which outputs text like *DrawText* but accepts an *x*-*y* coordinate pair that specifies where the text will be output and also uses the current position if it is asked to. The statement

|  |
| --- |
| dc.TextOut (0, 0, CString (\_T ("Hello, MFC"))); |

writes "Hello, MFC" to the upper left of the display surface represented by *dc*. A related function named *TabbedTextOut* works just like *TextOut* except that it expands tab characters into white space. (If a string passed to *TextOut* contains tabs, the characters show up as rectangles in most fonts.) Tab positions are specified in the call to *TabbedTextOut*. A related function named *ExtTextOut* gives you the added option of filling a rectangle surrounding the output text with an opaque background color. It also gives the programmer precise control over intercharacter spacing.

By default, the coordinates passed to *TextOut*, *TabbedTextOut*, and *ExtTextOut* specify the location of the upper left corner of the text's leftmost character cell. However, the relationship between the coordinates passed to *TextOut* and the characters in the output string, a property known as the *text alignment*, is an attribute of the device context. You can change it with *CDC::SetTextAlign*. For example, after a

|  |
| --- |
| dc.SetTextAlign (TA\_RIGHT); |

statement is executed, the *x* coordinate passed to *TextOut* specifies the rightmost position in the character cell—perfect for drawing right-aligned text.

You can also call *SetTextAlign* with a TA\_UPDATECP flag to instruct *TextOut* to ignore the *x* and *y* arguments passed to it and use the device context's current position instead. When the text alignment includes TA\_UPDATECP, *TextOut* updates the *x* component of the current position each time a string is output. One use for this feature is to achieve proper spacing between two or more character strings that are output on the same line.

***CDC* Text Functions**

|  |  |
| --- | --- |
| **Function** | **Description** |
| *DrawText* | Draws text in a formatting rectangle |
| *TextOut* | Outputs a line of text at the current or specified position |
| *TabbedTextOut* | Outputs a line of text that includes tabs |
| *ExtTextOut* | Outputs a line of text and optionally fills a rectangle with a background color or varies the intercharacter spacing |
| *GetTextExtent* | Computes the width of a string in the current font |
| *GetTabbedTextExtent* | Computes the width of a string with tabs in the current font |
| *GetTextMetrics* | Returns font metrics (character height, average character width, and so on) for the current font |
| *SetTextAlign* | Sets alignment parameters for *TextOut* and other output functions |
| *SetTextJustification* | Specifies the added width that is needed to justify a string of text |
| *SetTextColor* | Sets the device context's text output color |
| *SetBkColor* | Sets the device context's background color, which determines the fill color used behind characters that are output to a display surface |

Two functions—*GetTextMetrics* and *GetTextExtent*—let you retrieve information about the font that is currently selected into the device context. *GetTextMetrics* fills a TEXTMETRIC structure with information on the characters that make up the font. *GetTextExtent* returns the width of a given string, in logical units, rendered in that font. (Use *GetTabbedTextExtent* if the string contains tab characters.) One use for *GetTextExtent* is to gauge the width of a string prior to outputting it in order to compute how much space is needed between words to fully justify the text. If *nWidth* is the distance between left and right margins, the following code outputs the text "Now is the time" and justifies the output to both margins:

|  |
| --- |
| CString string = \_T ("Now is the time");  CSize size = dc.GetTextExtent (string);  dc.SetTextJustification (nWidth - size.cx, 3);  dc.TextOut (0, y, string); |

The second parameter passed to *SetTextJustification* specifies the number of break characters in the string. The default break character is the space character. After *SetTextJustification* is called, subsequent calls to *TextOut* and related text output functions distribute the space specified in the *SetTextJustification*'s first parameter evenly between all the break characters.

## GDI Fonts and the *CFont* Class

All *CDC* text functions use the font that is currently selected into the device context. A *font* is a group of characters of a particular size (height) and typeface that share common attributes such as character weight—for example, normal or boldface. In classical typography, font sizes are measured in units called *points*. One point equals about 1/72 inch. Each character in a 12-point font is nominally 1/6 inch tall, but in Windows, the actual height can vary somewhat depending on the physical characteristics of the output device. The term *typeface* describes a font's basic style. Times New Roman is one example of a typeface; Courier New is another.

A font is a GDI object, just as a pen or a brush is. In MFC, fonts are represented by objects of the *CFont* class. Once a *CFont* object is constructed, you create the underlying GDI font by calling the *CFont* object's *CreateFont*, *CreateFontIndirect*, *CreatePointFont*, or *CreatePointFontIndirect* function. Use *CreateFont* or *CreateFontIndirect* if you want to specify the font size in pixels, and use *CreatePointFont* and *CreatePointFontIndirect* to specify the font size in points. Creating a 12-point Times New Roman screen font with *CreatePointFont* requires just two lines of code:

|  |
| --- |
| CFont font;  font.CreatePointFont (120, \_T ("Times New Roman")); |

Doing the same with *CreateFont* requires you to query the device context for the logical number of pixels per inch in the vertical direction and to convert points to pixels:

|  |
| --- |
| CClientDC dc (this);  int nHeight = -((dc.GetDeviceCaps (LOGPIXELSY) \* 12) / 72);  CFont font;  font.CreateFont (nHeight, 0, 0, 0, FW\_NORMAL, 0, 0, 0,  DEFAULT\_CHARSET, OUT\_CHARACTER\_PRECIS, CLIP\_CHARACTER\_PRECIS,  DEFAULT\_QUALITY, DEFAULT\_PITCH ¦ FF\_DONTCARE,  \_T ("Times New Roman")); |

Incidentally, the numeric value passed to *CreatePointFont* is the desired point size *times 10*. This allows you to control the font size down to 1/10 point—plenty accurate enough for most applications, considering the relatively low resolution of most screens and other commonly used output devices.

The many parameters passed to *CreateFont* specify, among other things, the font weight and whether characters in the font are italicized. You can't create a bold, italic font with *CreatePointFont*, but you can with *CreatePointFontIndirect*. The following code creates a 12-point bold, italic Times New Roman font using *CreatePointFontIndirect*.

|  |
| --- |
| LOGFONT lf;  ::ZeroMemory (&lf, sizeof (lf));  lf.lfHeight = 120;  lf.lfWeight = FW\_BOLD;  lf.lfItalic = TRUE;  ::lstrcpy (lf.lfFaceName, \_T ("Times New Roman"));  CFont font;  font.CreatePointFontIndirect (&lf); |

LOGFONT is a structure whose fields define all the characteristics of a font. *::ZeroMemory* is an API function that zeroes a block of memory, and *::lstrcpy* is an API function that copies a text string from one memory location to another. You can use the C run time's *memset* and *strcpy* functions instead (actually, you should use *\_tcscpy* in lieu of *strcpy* so the call will work with ANSI or Unicode characters), but using Windows API functions frequently makes an executable smaller by reducing the amount of statically linked code.

After creating a font, you can select it into a device context and draw with it using *DrawText*, *TextOut*, and other *CDC* text functions. The following *OnPaint* handler draws "Hello, MFC" in the center of a window. But this time the text is drawn using a 72-point Arial typeface, complete with drop shadows. (See Figure 2-9.)

|  |
| --- |
| void CMainWindow::OnPaint ()  {  CRect rect;  GetClientRect (&rect);  CFont font;  font.CreatePointFont (720, \_T ("Arial"));  CPaintDC dc (this);  dc.SelectObject (&font);  dc.SetBkMode (TRANSPARENT);  CString string = \_T ("Hello, MFC");  rect.OffsetRect (16, 16);  dc.SetTextColor (RGB (192, 192, 192));  dc.DrawText (string, &rect, DT\_SINGLELINE ¦  DT\_CENTER ¦ DT\_VCENTER);  rect.OffsetRect (-16, -16);  dc.SetTextColor (RGB (0, 0, 0));  dc.DrawText (string, &rect, DT\_SINGLELINE ¦  DT\_CENTER ¦ DT\_VCENTER);  } |



**Figure 2-9.** *"Hello, MFC" rendered in 72-point Arial with drop shadows.*

The shadow effect is achieved by drawing the text string twice—once a few pixels to the right of and below the center of the window, and once in the center. MFC's *CRect::OffsetRect* function makes it a snap to "move" rectangles by offsetting them a specified distance in the *x* and *y* directions.

What happens if you try to create, say, a Times New Roman font on a system that doesn't have Times New Roman installed? Rather than fail the call, the GDI will pick a similar typeface that *is* installed. An internal font-mapping algorithm is called to pick the best match, and the results aren't always what one might expect. But at least your application won't output text just fine on one system and mysteriously output nothing on another.

## Raster Fonts vs. TrueType Fonts

Most GDI fonts fall into one of two categories: raster fonts and TrueType fonts. Raster fonts are stored as bitmaps and look best when they're displayed in their native sizes. One of the most useful raster fonts provided with Windows is MS Sans Serif, which is commonly used (in its 8-point size) on push buttons, radio buttons, and other dialog box controls. Windows can scale raster fonts by duplicating rows and columns of pixels, but the results are rarely pleasing to the eye due to stair-stepping effects.

The best fonts are TrueType fonts because they scale well to virtually any size. Like PostScript fonts, TrueType fonts store character outlines as mathematical formulas. They also include "hint" information that's used by the GDI's TrueType font rasterizer to enhance scalability. You can pretty much bank on the fact that any system your application runs on will have the following TrueType fonts installed, because all four are provided with Windows:

* Times New Roman
* Arial
* Courier New
* Symbol

In [Chapter 7](mk:@MSITStore:C:\Program%20Files%20(x86)\MSPress\BooksOnline\Programming%20Windows%20with%20MFC%20Second%20Edition\progmfc2.chm::/ch07a.htm), you'll learn how to query the system for font information and how to enumerate the fonts that are installed. Such information can be useful if your application requires precise character output or if you want to present a list of installed fonts to the user.

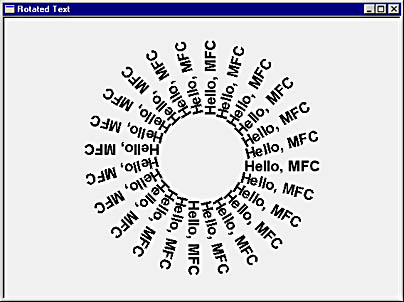
## Rotated Text

One question that's frequently asked about GDI text output is "How do I display rotated text?" There are two ways to do it, one of which works only in Microsoft Windows NT and Windows 2000. The other method is compatible with all 32-bit versions of Windows, so it's the one I'll describe here.

The secret is to create a font with *CFont::CreateFontIndirect* or *CFont::CreatePointFontIndirect* and to specify the desired rotation angle (in degrees) times 10 in the LOGFONT structure's *lfEscapement* and *lfOrientation* fields. Then you output the text in the normal manner—for example, using *CDC::TextOut*. Conventional text has an escapement and orientation of 0; that is, it has no slant and is drawn on a horizontal. Setting these values to 450 rotates the text counterclockwise 45 degrees. The following *OnPaint* handler increments *lfEscapement* and *lfOrientation* in units of 15 degrees and uses the resulting fonts to draw the radial text array shown in Figure 2-10:

|  |
| --- |
| void CMainWindow::OnPaint ()  {  CRect rect;  GetClientRect (&rect);  CPaintDC dc (this);  dc.SetViewportOrg (rect.Width () / 2, rect.Height () / 2);  dc.SetBkMode (TRANSPARENT);  for (int i=0; i<3600; i+=150) {  LOGFONT lf;  ::ZeroMemory (&lf, sizeof (lf));  lf.lfHeight = 160;  lf.lfWeight = FW\_BOLD;  lf.lfEscapement = i;  lf.lfOrientation = i;  ::lstrcpy (lf.lfFaceName, \_T ("Arial"));  CFont font;  font.CreatePointFontIndirect (&lf);  CFont\* pOldFont = dc.SelectObject (&font);  dc.TextOut (0, 0, CString (\_T (" Hello, MFC")));  dc.SelectObject (pOldFont);  }  } |

This technique works great with TrueType fonts, but it doesn't work at all with raster fonts.



**Figure 2-10.** *Rotated text.*

## Stock Objects

Windows predefines a handful of pens, brushes, fonts, and other GDI objects that can be used without being explicitly created. Called *stock objects*, these GDI objects can be selected into a device context with the *CDC::SelectStockObject* function or assigned to an existing *CPen*, *CBrush*, or other object with *CGdiObject::CreateStockObject*. *CGdiObject* is the base class for *CPen*, *CBrush*, *CFont*, and other MFC classes that represent GDI objects.

The following table shows a partial list of the available stock objects. Stock pens go by the names WHITE\_PEN, BLACK\_PEN, and NULL\_PEN. WHITE\_PEN and BLACK\_PEN draw solid lines that are 1 pixel wide. NULL\_PEN draws nothing. The stock brushes include one white brush, one black brush, and three shades of gray. HOLLOW\_BRUSH and NULL\_BRUSH are two different ways of referring to the same thing—a brush that paints nothing. SYSTEM\_FONT is the font that's selected into every device context by default.

**Commonly Used Stock Objects**

|  |  |
| --- | --- |
| **Object** | **Description** |
| NULL\_PEN | Pen that draws nothing |
| BLACK\_PEN | Black pen that draws solid lines 1 pixel wide |
| WHITE\_PEN | White pen that draws solid lines 1 pixel wide |
| NULL\_BRUSH | Brush that draws nothing |
| HOLLOW\_BRUSH | Brush that draws nothing (same as NULL\_BRUSH) |
| BLACK\_BRUSH | Black brush |
| DKGRAY\_BRUSH | Dark gray brush |
| GRAY\_BRUSH | Medium gray brush |
| LTGRAY\_BRUSH | Light gray brush |
| WHITE\_BRUSH | White brush |
| ANSI\_FIXED\_FONT | Fixed-pitch ANSI font |
| ANSI\_VAR\_FONT | Variable-pitch ANSI font |
| SYSTEM\_FONT | Variable-pitch system font |
| SYSTEM\_FIXED\_FONT | Fixed-pitch system font |

Suppose you want to draw a light gray circle with no border. How do you do it? Here's one way:

|  |
| --- |
| CPen pen (PS\_NULL, 0, (RGB (0, 0, 0)));  dc.SelectObject (&pen);  CBrush brush (RGB (192, 192, 192));  dc.SelectObject (&brush);  dc.Ellipse (0, 0, 100, 100); |

But since NULL pens and light gray brushes are stock objects, here's a better way:

|  |
| --- |
| dc.SelectStockObject (NULL\_PEN);  dc.SelectStockObject (LTGRAY\_BRUSH);  dc.Ellipse (0, 0, 100, 100); |

The following code demonstrates a third way to draw the circle. This time the stock objects are assigned to a *CPen* and a *CBrush* rather than selected into the device context directly:

|  |
| --- |
| CPen pen;  pen.CreateStockObject (NULL\_PEN);  dc.SelectObject (&pen);  CBrush brush;  brush.CreateStockObject (LTGRAY\_BRUSH);  dc.SelectObject (&brush);  dc.Ellipse (0, 0, 100, 100); |

Which of the three methods you use is up to you. The second method is the shortest, and it's the only one that's guaranteed not to throw an exception since it doesn't create any GDI objects.

## Deleting GDI Objects

Pens, brushes, and other objects created from *CGdiObject*-derived classes are resources that consume space in memory, so it's important to delete them when you no longer need them. If you create a *CPen*, *CBrush*, *CFont*, or other *CGdiObject* on the stack, the associated GDI object is automatically deleted when *CGdiObject* goes out of scope. If you create a *CGdiObject* on the heap with *new*, be sure to delete it at some point so that its destructor will be called. The GDI object associated with a *CGdiObject* can be explicitly deleted by calling *CGdiObject::DeleteObject*. You never need to delete stock objects, even if they are "created" with *CreateStockObject*.

In 16-bit Windows, GDI objects frequently contributed to the problem of resource leakage, in which the Free System Resources figure reported by Program Manager gradually decreased as applications were started and terminated because some programs failed to delete the GDI objects they created. All 32-bit versions of Windows track the resources a program allocates and deletes them when the program ends. However, it's *still* important to delete GDI objects when they're no longer needed so that the GDI doesn't run out of memory while a program is running. Imagine an *OnPaint* handler that creates 10 pens and brushes every time it's called but neglects to delete them. Over time, *OnPaint* might create thousands of GDI objects that occupy space in system memory owned by the Windows GDI. Pretty soon, calls to create pens and brushes will fail, and the application's *OnPaint* handler will mysteriously stop working.

In Visual C++, there's an easy way to tell whether you're failing to delete pens, brushes, and other resources: Simply run a debug build of your application in debugging mode. When the application terminates, resources that weren't freed will be listed in the debugging window. MFC tracks memory allocations for *CPen*, *CBrush*, and other *CObject*-derived classes so that it can notify you when an object hasn't been deleted. If you have difficulty ascertaining from the debug messages which objects weren't deleted, add the statement

|  |
| --- |
| #define new DEBUG\_NEW |

to your application's source code files after the statement that includes Afxwin.h. (In AppWizard-generated applications, this statement is included automatically.) Debug messages for unfreed objects will then include line numbers and file names to help you pinpoint leaks.

## Deselecting GDI Objects

It's important to delete the GDI objects you create, but it's equally important to never delete a GDI object while it's selected into a device context. Code that attempts to paint with a deleted object is buggy code. The only reason it doesn't crash is that the Windows GDI is sprinkled with error-checking code to prevent such crashes from occurring.

Abiding by this rule isn't as easy as it sounds. The following *OnPaint* handler allows a brush to be deleted while it's selected into a device context. Can you figure out why?

|  |
| --- |
| void CMainWindow::OnPaint ()  {  CPaintDC dc (this);  CBrush brush (RGB (255, 0, 0));  dc.SelectObject (&brush);  dc.Ellipse (0, 0, 200, 100);  } |

Here's the problem. A *CPaintDC* object and a *CBrush* object are created on the stack. Since the *CBrush* is created second, its destructor gets called first. Consequently, the associated GDI brush is deleted before *dc* goes out of scope. You could fix this by creating the brush first and the DC second, but code whose robustness relies on stack variables being created in a particular order is bad code indeed. As far as maintainability goes, it's a nightmare.

The solution is to select the *CBrush* out of the device context before the *CPaintDC* object goes out of scope. There is no *UnselectObject* function, but you can select an object out of a device context by selecting in another object. Most Windows programmers make it a practice to save the pointer returned by the first call to *SelectObject* for each object type and then use that pointer to reselect the default object. An equally viable approach is to select stock objects into the device context to replace the objects that are currently selected in. The first of these two methods is illustrated by the following code:

|  |
| --- |
| CPen pen (PS\_SOLID, 1, RGB (255, 0, 0));  CPen\* pOldPen = dc.SelectObject (&pen);  CBrush brush (RGB (0, 0, 255));  CBrush\* pOldBrush = dc.SelectObject (&brush);    dc.SelectObject (pOldPen);  dc.SelectObject (pOldBrush); |

The second method works like this:

|  |
| --- |
| CPen pen (PS\_SOLID, 1, RGB (255, 0, 0));  dc.SelectObject (&pen);  CBrush brush (RGB (0, 0, 255));  dc.SelectObject (&brush);    dc.SelectStockObject (BLACK\_PEN);  dc.SelectStockObject (WHITE\_BRUSH); |

The big question is why this is necessary. The simple truth is that it's not. In modern versions of Windows, there's no harm in allowing a GDI object to be deleted a split second before a device context is released, especially if you're absolutely sure that no drawing will be done in the interim. Still, cleaning up a device context by deselecting the GDI objects you selected in is a common practice in Windows programming. It's also considered good form, so it's something I'll do throughout this book.

Incidentally, GDI objects are occasionally created on the heap, like this:

|  |
| --- |
| CPen\* pPen = new CPen (PS\_SOLID, 1, RGB (255, 0, 0));  CPen\* pOldPen = dc.SelectObject (pPen); |

At some point, the pen must be selected out of the device context and deleted. The code to do it might look like this:

|  |
| --- |
| dc.SelectObject (pOldPen);  delete pPen; |

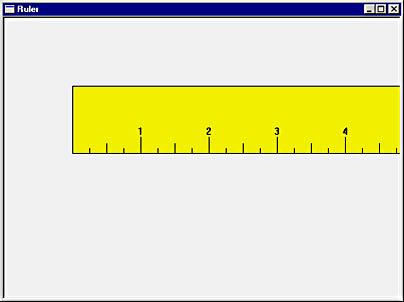
Since the *SelectObject* function returns a pointer to the object selected out of the device context, it might be tempting to try to deselect the pen and delete it in one step:

|  |
| --- |
| delete dc.SelectObject (pOldPen); |

But don't do this. It works fine with pens, but it might not work with brushes. Why? Because if you create two identical *CBrush*es, 32-bit Windows conserves memory by creating just one GDI brush and you'll wind up with two *CBrush* pointers that reference the same HBRUSH. (An HBRUSH is a handle that uniquely identifies a GDI brush, just as an HWND identifies a window and an HDC identifies a device context. A *CBrush* wraps an HBRUSH and stores the HBRUSH handle in its *m\_hObject* data member.) Because *CDC::SelectObject* uses an internal table maintained by MFC to convert the HBRUSH handle returned by *SelectObject* to a *CBrush* pointer and because that table assumes a one-to-one mapping between HBRUSHes and *CBrush*es, the *CBrush* pointer you get back might not match the *CBrush* pointer returned by *new*. Be sure you pass *delete* the pointer returned by *new*. Then both the GDI object and the C++ object will be properly destroyed.

## The Ruler Application

The best way to get acquainted with the GDI and the MFC classes that encapsulate it is to write code. Let's start with a very simple application. Figure 2-12 contains the source code for Ruler, a program that draws a 12-inch ruler on the screen. Ruler's output is shown in Figure 2-11.



**Figure 2-11.** *The Ruler window.*

**Figure 2-12.** *The Ruler application.*

|  |
| --- |
| Ruler.h class CMyApp : public CWinApp  {  public:  virtual BOOL InitInstance ();  };  class CMainWindow : public CFrameWnd  {  public:  CMainWindow ();  protected:  afx\_msg void OnPaint ();  DECLARE\_MESSAGE\_MAP ()  }; |

|  |
| --- |
| Ruler.cpp #include <afxwin.h>  #include "Ruler.h"  CMyApp myApp;  /////////////////////////////////////////////////////////////////////////  // CMyApp member functions  BOOL CMyApp::InitInstance ()  {  m\_pMainWnd = new CMainWindow;  m\_pMainWnd->ShowWindow (m\_nCmdShow);  m\_pMainWnd->UpdateWindow ();  return TRUE;  }  /////////////////////////////////////////////////////////////////////////  // CMainWindow message map and member functions  BEGIN\_MESSAGE\_MAP (CMainWindow, CFrameWnd)  ON\_WM\_PAINT ()  END\_MESSAGE\_MAP ()  CMainWindow::CMainWindow ()  {  Create (NULL, \_T ("Ruler"));  }  void CMainWindow::OnPaint ()  {  CPaintDC dc (this);    //  // Initialize the device context.  //  dc.SetMapMode (MM\_LOENGLISH);  dc.SetTextAlign (TA\_CENTER ¦ TA\_BOTTOM);  dc.SetBkMode (TRANSPARENT);  //  // Draw the body of the ruler.  //  CBrush brush (RGB (255, 255, 0));  CBrush\* pOldBrush = dc.SelectObject (&brush);  dc.Rectangle (100, -100, 1300, -200);  dc.SelectObject (pOldBrush);  //  // Draw the tick marks and labels.  //  for (int i=125; i<1300; i+=25) {  dc.MoveTo (i, -192);  dc.LineTo (i, -200);  }  for (i=150; i<1300; i+=50) {  dc.MoveTo (i, -184);  dc.LineTo (i, -200);  }  for (i=200; i<1300; i+=100) {  dc.MoveTo (i, -175);  dc.LineTo (i, -200);  CString string;  string.Format (\_T ("%d"), (i / 100) - 1);  dc.TextOut (i, -175, string);  }  } |

The structure of Ruler is similar to that of the Hello application presented in Chapter 1. The *CMyApp* class represents the application itself. *CMyApp::InitInstance* creates a frame window by constructing a *CMainWindow* object, and *CMainWindow*'s constructor calls *Create* to create the window you see on the screen. *CMainWindow::OnPaint* handles all the drawing. The body of the ruler is drawn with *CDC::Rectangle,* and the hash marks are drawn with *CDC::LineTo* and *CDC::MoveTo*. Before the rectangle is drawn, a yellow brush is selected into the device context so that the body of the ruler will be painted yellow. Numeric labels are drawn with *CDC::TextOut* and positioned over the tick marks by calling *SetTextAlign* with TA\_CENTER and TA\_BOTTOM flags and passing *TextOut* the coordinates of the top of each tick mark. Before *TextOut* is called for the first time, the device context's background mode is set to TRANSPARENT. Otherwise, the numbers on the face of the ruler would be drawn with white backgrounds.

Rather than hardcode the strings passed to *TextOut*, Ruler uses *CString::Format* to generate text on the fly. *CString* is the MFC class that represents text strings. *CString::Format* works like C's *printf* function, converting numeric values to text and substituting them for placeholders in a formatting string. Windows programmers who work in C frequently use the *::wsprintf* API function for text formatting. *Format* does the same thing for *CString* objects without requiring an external function call. And unlike *::wsprintf*, *Format* supports the full range of *printf* formatting codes, including codes for floating-point and string variable types.

Ruler uses the MM\_LOENGLISH mapping mode to scale its output so that 1 inch on the ruler corresponds to 1 logical inch on the screen. Hold a real ruler up to the screen and on most PCs you'll find that 1 logical inch equals a little more than 1 physical inch. If the ruler is output to a printer instead, logical inches and physical inches will match exactly.

# Seeing What You've Drawn

Unfortunately, there is one small problem with Ruler's output: Unless you're running the program on a very high resolution video adapter, you can't see everything it draws. Even on a 1,280-pixel by 1,204-pixel screen, the window can't be stretched wide enough to make all the output visible. What doesn't fit inside the window's client area is clipped by the GDI. You could modify the sample program to make the ruler shorter, but that still wouldn't do much for someone running Windows on a 640-by-480 screen. No, there's a better solution, one that's entirely independent of the screen resolution. That solution is a scroll bar.

## Adding a Scroll Bar to a Window

A scroll bar is a window with an arrow at each end and a traveling "thumb" in between that can be dragged with the mouse. Scroll bars can be oriented horizontally or vertically, but never at an angle. When the user clicks one of the scroll bar arrows, moves the thumb, or clicks the scroll bar shaft, the scroll bar informs the window it's attached to by sending it a message. It's up to the window to decide what, if anything, to do with that message because a scroll bar does very little on its own. It doesn't, for example, magically scroll the window's contents. What it does do is provide a very intuitive and universally recognized mechanism for scrolling backward and forward over a virtual landscape that's too large to fit within the physical confines of a window.

Adding a scroll bar to a window is one of the easiest things you'll ever do in a Windows program. To add a vertical scroll bar, create the window with the WS\_VSCROLL style. To add a horizontal scroll bar, use the WS\_HSCROLL style. To add horizontal and vertical scroll bars, use both WS\_VSCROLL and WS\_HSCROLL. Recall from [Chapter 1](mk:@MSITStore:C:\Program%20Files%20(x86)\MSPress\BooksOnline\Programming%20Windows%20with%20MFC%20Second%20Edition\progmfc2.chm::/ch01a.htm) that the third parameter passed to *CFrameWnd::Create* is the window style, and that the default is WS\_OVERLAPPEDWINDOW. An application that creates a conventional frame window with the statement

|  |
| --- |
| Create (NULL, \_T ("My Application")); |

can create a frame window containing a vertical scroll bar with the statement

|  |
| --- |
| Create (NULL, \_T ("My Application"), WS\_OVERLAPPEDWINDOW ¦ WS\_VSCROLL); |

Accordingly, Windows provides a scroll bar that extends the height of the window's client area from top to bottom on the right side. If you'd rather have the scroll bar appear on the left, include a WS\_EX\_LEFTSCROLLBAR flag in *Create*'s optional *dwExStyle* (seventh) parameter.

## Setting a Scroll Bar's Range, Position, and Page Size

After you create a scroll bar, you should initialize it with a range, position, and page size. The *range* is a pair of integers that define the upper and lower limits of the scroll bar's travel. The *position* is an integer value that specifies the current location within that range; its value is reflected in the position of the scroll bar thumb. The *page size* sets the size of the thumb to provide a visual representation of the relationship between the size of the window and the size of the scrollable view. For example, if the scroll bar range is 0 to 100 and the page size is 50, the thumb size is half the scroll bar length. If you don't set the page size, Windows picks a default, nonproportional thumb size for you.

One way to set a scroll bar's range and position is with the *CWnd::SetScrollRange* and *CWnd::SetScrollPos* functions. The statement

|  |
| --- |
| SetScrollRange (SB\_VERT, 0, 100, TRUE); |

sets a vertical scroll bar's range to 0 through 100, while the statement

|  |
| --- |
| SetScrollPos (SB\_VERT, 50, TRUE); |

sets the current position to 50 and consequently moves the thumb to the middle of the scroll bar. (For horizontal scroll bars, use SB\_HORZ instead of SB\_VERT.) A scroll bar maintains a record of its current range and position internally. You can query for those values at any time with *CWnd*::*GetScrollRange* and *CWnd::GetScrollPos*.

The TRUE parameter passed to *SetScrollRange* and *SetScrollPos* specifies that the scroll bar should be redrawn to reflect the change. You can prevent redraws by specifying FALSE. If you specify neither TRUE nor FALSE, both *SetScrollRange* and *SetScrollPos* default to TRUE. You generally want a scroll bar to redraw itself after one of these functions is called, but not if both are called in quick succession. Redrawing a scroll bar twice in a very short period of time produces an undesirable flashing effect. If you're setting the range and the position together, do it like this:

|  |
| --- |
| SetScrollRange (SB\_VERT, 0, 100, FALSE);  SetScrollPos (SB\_VERT, 50, TRUE); |

*SetScrollPos* and *SetScrollRange* date back to the very first version of Windows. In today's versions, the preferred way to set a scroll bar's range and position is with the *CWnd::SetScrollInfo* function. In addition to allowing the range and the position to be set with a single function call, *SetScrollInfo* also provides a means—the *only* means, as it turns out—for setting the page size. *SetScrollInfo* accepts three parameters:

* An SB\_VERT or SB\_HORZ parameter that specifies whether the scroll bar is vertical or horizontal (or SB\_BOTH if you want to initialize two scroll bars at once)
* A pointer to a SCROLLINFO structure
* A BOOL value (TRUE or FALSE) that specifies whether the scroll bar should be redrawn

SCROLLINFO is defined as follows in Winuser.h:

|  |
| --- |
| typedef struct tagSCROLLINFO  {  UINT cbSize;  UINT fMask;  int nMin;  int nMax;  UINT nPage;  int nPos;  int nTrackPos;  } SCROLLINFO, FAR \*LPSCROLLINFO; |

*cbSize* specifies the size of the structure, *nMin* and *nMax* specify the scroll bar range, *nPage* specifies the page size, and *nPos* specifies the position. *nTrackPos* is not used in calls to *SetScrollInfo*, but it returns the scroll bar's thumb position when the complementary *GetScrollInfo* function is called to retrieve information about the scroll bar while the thumb is being dragged. The *fMask* field holds a combination of one or more of the following bit flags:

* SIF\_DISABLENOSCROLL, which disables the scroll bar
* SIF\_PAGE, which indicates that *nPage* holds the page size
* SIF\_POS, which indicates that *nPos* holds the scroll bar position
* SIF\_RANGE, which indicates that *nMin* and *nMax* hold the scroll bar range
* SIF\_ALL, which is equivalent to SIF\_PAGE ¦ SIF\_POS ¦ SIF\_RANGE.

*SetScrollInfo* ignores fields for which bit flags are not specified. The statements

|  |
| --- |
| SCROLLINFO si;  si.fMask = SIF\_POS;  si.nPos = 50;  SetScrollInfo (SB\_VERT, &si, TRUE); |

set the position while leaving the range and page size unaffected, and

|  |
| --- |
| SCROLLINFO si;  si.fMask = SIF\_RANGE ¦ SIF\_POS ¦ SIF\_PAGE; // Or SIF\_ALL  si.nMin = 0;  si.nMax = 99;  si.nPage = 25;  si.nPos = 50;  SetScrollInfo (SB\_VERT, &si, TRUE); |

sets the range, page size, and position in one operation. You don't need to initialize *cbSize* before calling *SetScrollInfo* or *GetScrollInfo* because MFC initializes it for you.

You can make a scroll bar disappear by setting the upper limit of its range equal to the lower limit. The scroll bar doesn't go away entirely; it's still there, even though you can't see it, and—more important—you can bring it back by making the range upper and lower limits different again. This turns out to be quite a useful trick if you want to hide the scroll bar because the window has been enlarged to the point that a scroll bar is no longer required. *SetScrollInfo*'s SIF\_DISABLENOSCROLL flag prevents a scroll bar from accepting further input, but it doesn't make the scroll bar disappear. Having a disabled scroll bar visible inside a window can be confusing to users, who are apt to wonder why the scroll bar is there if it can't be used.

When you set a scroll bar's range, page size, and position, here's a convenient model to keep in mind. Suppose your window's client area is 100 units high and the workspace you want to cover with a vertical scroll bar is 400 units high. Set the scroll bar range to 0-399 and the page size to 100. Accordingly, Windows will draw the scroll bar thumb so that it is one-fourth the height of the scroll bar. When the scroll bar position is 0, the thumb is positioned at the top of the scroll bar. As the thumb is scrolled down, scroll the contents of your window up an amount proportional to the distance the thumb was moved. If you limit the scroll bar's maximum position to 300 (the difference between the magnitude of the scroll bar range and the page size), the bottom of the thumb will reach the bottom of the scroll bar at the same time that the bottom of the workspace scrolls into view at the bottom of the window.

## Synchronizing the Thumb Size and the Window Size

Since a scroll bar's thumb size reflects the relative size of the window compared to the width or the height of the virtual workspace, you should update the thumb size when the window size changes. It's easy to do: Just call *SetScrollInfo* with an SIF\_PAGE flag each time your window receives a WM\_SIZE message. The first WM\_SIZE message comes when a window is created. Subsequent WM\_SIZE messages arrive whenever the window's size changes. In MFC, an ON\_WM\_SIZE entry in a class's message map directs WM\_SIZE messages to a handler named *OnSize*. The handler is prototyped as follows:

|  |
| --- |
| afx\_msg void OnSize (UINT nType, int cx, int cy) |

The *nType* parameter informs the window whether it has been minimized, maximized, or simply resized by using the code SIZE\_MINIMIZED, SIZE\_MAXIMIZED, or SIZE\_RESTORED, respectively. *cx* and *cy* are the client area's new width and height in pixels. If you know the dimensions of your application's virtual workspace, you can set the thumb size accordingly.

## Processing Scroll Bar Messages

A scroll bar notifies its owner (the window to which it is attached) of scroll bar events by sending messages. A horizontal scroll bar sends WM\_HSCROLL messages, and a vertical scroll bar sends WM\_VSCROLL messages. In MFC, these messages are directed to a window's *OnHScroll* and *OnVScroll* functions by ON\_WM\_HSCROLL and ON\_WM\_VSCROLL entries in the window's message map. Scroll bar message handlers are prototyped like this:

|  |
| --- |
| afx\_msg void OnHScroll (UINT nCode, UINT nPos, CScrollBar\* pScrollBar)  afx\_msg void OnVScroll (UINT nCode, UINT nPos, CScrollBar\* pScrollBar) |

*nCode* identifies the type of event that precipitated the message; *nPos* contains the latest thumb position if the thumb is being dragged or was just dragged and released; and, for a scroll bar that was created by adding a WS\_HSCROLL or WS\_VSCROLL style bit to a window, *pScrollBar* is NULL.

There are seven different event notifications that an application might receive in *OnVScroll*'s *nCode* parameter, as shown in the table below.

|  |  |
| --- | --- |
| **Event Code** | **Sent When** |
| SB\_LINEUP | The arrow at the top of the scroll bar is clicked. |
| SB\_LINEDOWN | The arrow at the bottom of the scroll bar is clicked. |
| SB\_PAGEUP | The scroll bar shaft is clicked between the up arrow and the thumb. |
| SB\_PAGEDOWN | The scroll bar shaft is clicked between the thumb and down arrow. |
| SB\_ENDSCROLL | The mouse button is released, and no more SB\_LINEUP, SB\_LINEDOWN, SB\_PAGEUP, or SB\_PAGEDOWN notifications are forthcoming. |
| SB\_THUMBTRACK | The scroll bar thumb is dragged. |
| SB\_THUMBPOSITION | The thumb is released after being dragged. |

Horizontal scroll bars send the same notifications as vertical scroll bars, but the notifications have slightly different meanings. For a horizontal scroll bar, SB\_LINEUP signifies that the left arrow was clicked, SB\_LINEDOWN means the right arrow was clicked, SB\_PAGEUP means the scroll bar was clicked between the left arrow and the thumb, and SB\_PAGEDOWN means the scroll bar was clicked between the thumb and the right arrow. If you prefer, you can use SB\_LINELEFT, SB\_LINERIGHT, SB\_PAGELEFT, and SB\_PAGERIGHT rather than SB\_LINEUP, SB\_LINEDOWN, SB\_PAGEUP, and SB\_PAGEDOWN. The discussions in the remainder of this chapter deal exclusively with vertical scroll bars, but keep in mind that anything said about vertical scroll bars also applies to horizontal scroll bars.

If the user clicks a scroll bar or scroll bar arrow and leaves the mouse button pressed, a series of SB\_LINEUP, SB\_LINEDOWN, SB\_PAGEUP, or SB\_PAGEDOWN notifications will arrive in rapid succession—similar to the stream of typematic key codes generated when a key is held down. SB\_ENDSCROLL terminates a stream of UP or DOWN notifications and indicates that the mouse button has been released. Even a single click of a scroll bar or scroll bar arrow generates an UP or a DOWN notification followed by an SB\_ENDSCROLL notification. Similarly, a window is bombarded with SB\_THUMBTRACK notifications that report new thumb positions as a scroll bar thumb is dragged, and it receives an SB\_THUMBPOSITION notification when the thumb is released. When an SB\_THUMBTRACK or SB\_THUMBPOSITION notification arrives, the message's *nPos* parameter holds the latest thumb position. For other event codes, the value of *nPos* is undefined.

How your program responds to scroll bar event messages is up to you. Most programs that use scroll bars disregard SB\_ENDSCROLL messages and respond to SB\_LINEUP, SB\_LINEDOWN, SB\_PAGEUP, and SB\_PAGEDOWN messages instead. A typical response to SB\_LINEUP and SB\_LINEDOWN messages is to scroll the contents of the window up or down one line and call *SetScrollPos* or *SetScrollInfo* to set the new scroll bar position and update the thumb location. "Line" can have whatever physical meaning you want it to have; it might mean 1 pixel, or it might mean the height of one line of text. Similarly, the usual response to SB\_PAGEUP and SB\_PAGEDOWN messages is to scroll up or down a distance equal to or slightly less than one "page," which is typically defined as the height of the window's client area or slightly less, and to call *SetScrollInfo* to set the new scroll position. In any event, it's your responsibility to update the scroll bar position. The scroll bar doesn't do that by itself.

Another, though less common, approach to processing UP and DOWN notifications is to continually move the scroll bar thumb by calling *SetScrollPos* or *SetScrollInfo* but to defer scrolling the window until an SB\_ENDSCROLL notification arrives. I once used this technique in a multimedia application that was relatively slow to respond to positional changes so that the latency of commands sent to a CD-ROM drive wouldn't impede the smooth movement of the scroll bar thumb.

SB\_THUMBTRACK and SB\_THUMBPOSITION notifications are handled a little differently. Since SB\_THUMBTRACK notifications are liable to come fast and furious when the thumb is dragged, some Windows applications ignore SB\_THUMBTRACK notifications and respond only to SB\_THUMBPOSITION notifications. In this case, the window doesn't scroll until the thumb is released. If you can scroll the contents of your window quickly enough to keep up with SB\_THUMBTRACK notifications, you can make your program more responsive to user input by scrolling as the thumb is dragged. It's still up to you to update the scroll bar position each time you scroll the window. Windows animates the movement of the scroll bar thumb as it's dragged up and down, but if you fail to call *SetScrollPos* or *SetScrollInfo* in response to SB\_THUMBTRACK or SB\_THUMBPOSITION notifications, the thumb will snap back to its original position the moment it's released.

## Scrolling a Window

Now that you understand how a scroll bar works, it's time to think about how to scroll the contents of a window in response to scroll bar messages.

The simplest approach is to change the scroll bar position each time a scroll bar message arrives and to call *CWnd::Invalidate* to force a repaint. The window's *OnPaint* handler can query the scroll bar for its current position and factor that information into its output. Unfortunately, scrolling a window this way is slow—*very* slow, for that matter. If the user clicks the up arrow to scroll the window contents up one line, it's wasteful to redraw the entire window because most of the information you want to display is already there, albeit in the wrong location. A more efficient approach to processing SB\_LINEUP messages is to copy everything currently displayed in the window down one line using a fast block copy and then to draw just the new top line. That's what *CWnd::ScrollWindow* is for.

*ScrollWindow* scrolls the contents of a window's client area—in whole or in part—up or down, left or right, by 1 or more pixels using a fast block pixel transfer. Moreover, it invalidates only the part of the window contents that is "uncovered" by the scrolling operation so that the next WM\_PAINT message doesn't have to repaint the entire window. If *ScrollWindow* is called to scroll a window downward by 10 pixels, it performs the scroll by doing a block copy. Then it invalidates the window's top 10 rows. This activates *OnPaint* and causes only the top 10 rows to be redrawn. Even if *OnPaint* tries to redraw the contents of the entire client area, performance is improved because most of the output is clipped. A smart *OnPaint* handler can further boost performance by restricting its GDI calls to those that affect pixels in the window's invalid rectangle. You'll see sample programs in Chapters 10 and 13 that use this technique to optimize scrolling performance.

*ScrollWindow* accepts four parameters. Two are required and two are optional. The function is prototyped as follows:

|  |
| --- |
| void ScrollWindow (int xAmount, int yAmount,  LPCRECT lpRect = NULL, LPCRECT lpClipRect = NULL) |

*xAmount* and *yAmount* are signed integers that specify the number of pixels to scroll horizontally and vertically. Negative values scroll left and up, while positive values scroll right and down. *lpRect* points to a *CRect* object or a RECT structure that specifies the part of the client area to scroll, and *lpClipRect* points to a *CRect* object or a RECT structure that specifies a clipping rectangle. Specifying NULL for *lpRect* and *lpClipRect* scrolls the contents of the entire client area. The statement

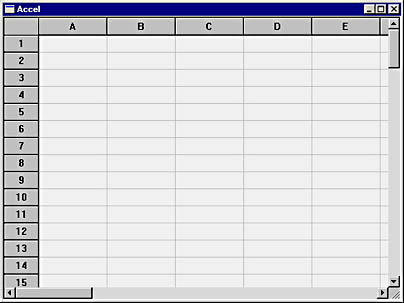
|  |
| --- |
| ScrollWindow (0, 10); |

scrolls everything in a window's client area downward by 10 pixels and prompts a redraw of the first 10 rows.

You can use *ScrollWindow* whether your application displays text, graphics, or both. In Windows all things are graphical—including text.

## The Accel Application

Let's put this newfound knowledge to work by writing an application that scrolls. Accel draws a window that resembles Microsoft Excel. (See Figure 2-13.) The spreadsheet depicted in the window is 26 columns wide and 99 rows high—much too large to be displayed all at once. However, scroll bars allow the user to view all parts of the spreadsheet. In addition to providing a hands-on look at the principles discussed in the preceding sections, Accel demonstrates another way that a program can scale its output. Rather than use a non-MM\_TEXT mapping mode, it uses *CDC::GetDeviceCaps* to query the display device for the number of pixels per inch displayed horizontally and vertically. Then it draws each spreadsheet cell so that it's 1 inch wide and ¼ inch tall using raw pixel counts.



**Figure 2-13.** *The Accel window.*

**Figure 2-14.** *The Accel application.*

|  |
| --- |
| Accel.h #define LINESIZE 8  class CMyApp : public CWinApp  {  public:  virtual BOOL InitInstance ();  };  class CMainWindow : public CFrameWnd  {  protected:  int m\_nCellWidth; // Cell width in pixels  int m\_nCellHeight; // Cell height in pixels  int m\_nRibbonWidth; // Ribbon width in pixels  int m\_nViewWidth; // Workspace width in pixels  int m\_nViewHeight; // Workspace height in pixels  int m\_nHScrollPos; // Horizontal scroll position  int m\_nVScrollPos; // Vertical scroll position  int m\_nHPageSize; // Horizontal page size  int m\_nVPageSize; // Vertical page size  public:  CMainWindow ();  protected:  afx\_msg void OnPaint ();  afx\_msg int OnCreate (LPCREATESTRUCT lpCreateStruct);  afx\_msg void OnSize (UINT nType, int cx, int cy);  afx\_msg void OnHScroll (UINT nCode, UINT nPos,  CScrollBar\* pScrollBar);  afx\_msg void OnVScroll (UINT nCode, UINT nPos,  CScrollBar\* pScrollBar);  DECLARE\_MESSAGE\_MAP ()  }; |

|  |
| --- |
| Accel.cpp #include <afxwin.h>  #include "Accel.h"  CMyApp myApp;  /////////////////////////////////////////////////////////////////////////  // CMyApp member functions  BOOL CMyApp::InitInstance ()  {  m\_pMainWnd = new CMainWindow;  m\_pMainWnd->ShowWindow (m\_nCmdShow);  m\_pMainWnd->UpdateWindow ();  return TRUE;  }  /////////////////////////////////////////////////////////////////////////  // CMainWindow message map and member functions  BEGIN\_MESSAGE\_MAP (CMainWindow, CFrameWnd)  ON\_WM\_CREATE ()  ON\_WM\_SIZE ()  ON\_WM\_PAINT ()  ON\_WM\_HSCROLL ()  ON\_WM\_VSCROLL ()  END\_MESSAGE\_MAP ()  CMainWindow::CMainWindow ()  {  Create (NULL, \_T ("Accel"),  WS\_OVERLAPPEDWINDOW ¦ WS\_HSCROLL ¦ WS\_VSCROLL);  }  int CMainWindow::OnCreate (LPCREATESTRUCT lpCreateStruct)  {  if (CFrameWnd::OnCreate (lpCreateStruct) == -1)  return -1;  CClientDC dc (this);  m\_nCellWidth = dc.GetDeviceCaps (LOGPIXELSX);  m\_nCellHeight = dc.GetDeviceCaps (LOGPIXELSY) / 4;  m\_nRibbonWidth = m\_nCellWidth / 2;  m\_nViewWidth = (26 \* m\_nCellWidth) + m\_nRibbonWidth;  m\_nViewHeight = m\_nCellHeight \* 100;  return 0;  }  void CMainWindow::OnSize (UINT nType, int cx, int cy)  {  CFrameWnd::OnSize (nType, cx, cy);  //  // Set the horizontal scrolling parameters.  //  int nHScrollMax = 0;  m\_nHScrollPos = m\_nHPageSize = 0;  if (cx < m\_nViewWidth) {  nHScrollMax = m\_nViewWidth - 1;  m\_nHPageSize = cx;  m\_nHScrollPos = min (m\_nHScrollPos, m\_nViewWidth -  m\_nHPageSize - 1);  }  SCROLLINFO si;  si.fMask = SIF\_PAGE ¦ SIF\_RANGE ¦ SIF\_POS;  si.nMin = 0;  si.nMax = nHScrollMax;  si.nPos = m\_nHScrollPos;  si.nPage = m\_nHPageSize;  SetScrollInfo (SB\_HORZ, &si, TRUE);  //  // Set the vertical scrolling parameters.  //  int nVScrollMax = 0;  m\_nVScrollPos = m\_nVPageSize = 0;  if (cy < m\_nViewHeight) {  nVScrollMax = m\_nViewHeight - 1;  m\_nVPageSize = cy;  m\_nVScrollPos = min (m\_nVScrollPos, m\_nViewHeight -  m\_nVPageSize - 1);  }  si.fMask = SIF\_PAGE ¦ SIF\_RANGE ¦ SIF\_POS;  si.nMin = 0;  si.nMax = nVScrollMax;  si.nPos = m\_nVScrollPos;  si.nPage = m\_nVPageSize;  SetScrollInfo (SB\_VERT, &si, TRUE);  }  void CMainWindow::OnPaint ()  {  CPaintDC dc (this);  //  // Set the window origin to reflect the current scroll positions.  //  dc.SetWindowOrg (m\_nHScrollPos, m\_nVScrollPos);  //  // Draw the grid lines.  //  CPen pen (PS\_SOLID, 0, RGB (192, 192, 192));  CPen\* pOldPen = dc.SelectObject (&pen);  for (int i=0; i<99; i++) {  int y = (i \* m\_nCellHeight) + m\_nCellHeight;  dc.MoveTo (0, y);  dc.LineTo (m\_nViewWidth, y);  }  for (int j=0; j<26; j++) {  int x = (j \* m\_nCellWidth) + m\_nRibbonWidth;  dc.MoveTo (x, 0);  dc.LineTo (x, m\_nViewHeight);  }  dc.SelectObject (pOldPen);    //  // Draw the bodies of the rows and the column headers.  //  CBrush brush;  brush.CreateStockObject (LTGRAY\_BRUSH);  CRect rcTop (0, 0, m\_nViewWidth, m\_nCellHeight);  dc.FillRect (rcTop, &brush);  CRect rcLeft (0, 0, m\_nRibbonWidth, m\_nViewHeight);  dc.FillRect (rcLeft, &brush);  dc.MoveTo (0, m\_nCellHeight);  dc.LineTo (m\_nViewWidth, m\_nCellHeight);  dc.MoveTo (m\_nRibbonWidth, 0);  dc.LineTo (m\_nRibbonWidth, m\_nViewHeight);  dc.SetBkMode (TRANSPARENT);  //  // Add numbers and button outlines to the row headers.  //  for (i=0; i<99; i++) {  int y = (i \* m\_nCellHeight) + m\_nCellHeight;  dc.MoveTo (0, y);  dc.LineTo (m\_nRibbonWidth, y);  CString string;  string.Format (\_T ("%d"), i + 1);  CRect rect (0, y, m\_nRibbonWidth, y + m\_nCellHeight);  dc.DrawText (string, &rect, DT\_SINGLELINE ¦  DT\_CENTER ¦ DT\_VCENTER);  rect.top++;  dc.Draw3dRect (rect, RGB (255, 255, 255),  RGB (128, 128, 128));  }  //  // Add letters and button outlines to the column headers.  //  for (j=0; j<26; j++) {  int x = (j \* m\_nCellWidth) + m\_nRibbonWidth;  dc.MoveTo (x, 0);  dc.LineTo (x, m\_nCellHeight);  CString string;  string.Format (\_T ("%c"), j + `A');  CRect rect (x, 0, x + m\_nCellWidth, m\_nCellHeight);  dc.DrawText (string, &rect, DT\_SINGLELINE ¦  DT\_CENTER ¦ DT\_VCENTER);  rect.left++;  dc.Draw3dRect (rect, RGB (255, 255, 255),  RGB (128, 128, 128));  }  }  void CMainWindow::OnHScroll (UINT nCode, UINT nPos, CScrollBar\* pScrollBar)  {  int nDelta;  switch (nCode) {  case SB\_LINELEFT:  nDelta = -LINESIZE;  break;  case SB\_PAGELEFT:  nDelta = -m\_nHPageSize;  break;  case SB\_THUMBTRACK:  nDelta = (int) nPos - m\_nHScrollPos;  break;  case SB\_PAGERIGHT:  nDelta = m\_nHPageSize;  break;  case SB\_LINERIGHT:  nDelta = LINESIZE;  break;  default: // Ignore other scroll bar messages  return;  }  int nScrollPos = m\_nHScrollPos + nDelta;  int nMaxPos = m\_nViewWidth - m\_nHPageSize;  if (nScrollPos < 0)  nDelta = -m\_nHScrollPos;  else if (nScrollPos > nMaxPos)  nDelta = nMaxPos - m\_nHScrollPos;  if (nDelta != 0) {  m\_nHScrollPos += nDelta;  SetScrollPos (SB\_HORZ, m\_nHScrollPos, TRUE);  ScrollWindow (-nDelta, 0);  }  }  void CMainWindow::OnVScroll (UINT nCode, UINT nPos, CScrollBar\* pScrollBar)  {  int nDelta;  switch (nCode) {  case SB\_LINEUP:  nDelta = -LINESIZE;  break;  case SB\_PAGEUP:  nDelta = -m\_nVPageSize;  break;  case SB\_THUMBTRACK:  nDelta = (int) nPos - m\_nVScrollPos;  break;  case SB\_PAGEDOWN:  nDelta = m\_nVPageSize;  break;  case SB\_LINEDOWN:  nDelta = LINESIZE;  break;  default: // Ignore other scroll bar messages  return;  }  int nScrollPos = m\_nVScrollPos + nDelta;  int nMaxPos = m\_nViewHeight - m\_nVPageSize;  if (nScrollPos < 0)  nDelta = -m\_nVScrollPos;  else if (nScrollPos > nMaxPos)  nDelta = nMaxPos - m\_nVScrollPos;  if (nDelta != 0) {  m\_nVScrollPos += nDelta;  SetScrollPos (SB\_VERT, m\_nVScrollPos, TRUE);  ScrollWindow (0, -nDelta);  }  } |

*GetDeviceCaps* is called from *CMainWindow*'s *OnCreate* handler, which is called upon receipt of a WM\_CREATE message. WM\_CREATE is the first message a window receives. It is sent just once, and it arrives very early in the window's lifetime—before the window is even visible on the screen. An ON\_WM\_CREATE entry in the window's message map connects WM\_CREATE messages to the member function named *OnCreate*. *OnCreate* is the ideal place to initialize member variables whose values can only be determined at run time. It is prototyped as follows:

|  |
| --- |
| afx\_msg int OnCreate (LPCREATESTRUCT lpCreateStruct) |

*lpCreateStruct* is a pointer to a structure of type CREATESTRUCT, which contains useful information about a window such as its initial size and location on the screen. The value returned by *OnCreate* determines what Windows does next. If all goes as planned, *OnCreate* returns 0, signaling to Windows that the window was properly initialized. If *OnCreate* returns -1, Windows fails the attempt to create the window. A prototype *OnCreate* handler looks like this:

|  |
| --- |
| int CMainWindow::OnCreate (LPCREATESTRUCT lpCreateStruct)  {  if (CFrameWnd::OnCreate (lpCreateStruct) == -1)  return -1;      return 0;  } |

*OnCreate* should always call the base class's *OnCreate* handler to give the framework the opportunity to execute its own window-creation code. This is especially important when you write document/view applications, because it is a function called by *CFrameWnd::OnCreate* that creates the view that goes inside a frame window.

You'll find the code that does the scrolling in the window's *OnHScroll* and *OnVScroll* handlers. *switch-case* logic converts the notification code passed in *nCode* into a signed *nDelta* value that represents the number of pixels the window should be scrolled. Once *nDelta* is computed, the scroll position is adjusted by *nDelta* pixels and the window is scrolled with the statements

|  |
| --- |
| m\_nVScrollPos += nDelta;  SetScrollPos (SB\_VERT, m\_nVScrollPos, TRUE);  ScrollWindow (0, -nDelta); |

for the vertical scroll bar and

|  |
| --- |
| m\_nHScrollPos += nDelta;  SetScrollPos (SB\_HORZ, m\_nHScrollPos, TRUE);  ScrollWindow (-nDelta, 0); |

for the horizontal scroll bar.

How are the scroll positions stored in *m\_nHScrollPos* and *m\_nVScrollPos* factored into the program's output? When *OnPaint* is called to paint the part of the workspace that was exposed by the scrolling operation, it repositions the window origin with the statement

|  |
| --- |
| dc.SetWindowOrg (m\_nHScrollPos, m\_nVScrollPos); |

Recall that *CDC::SetWindowOrg* tells Windows to map the logical point (*x*,*y*) to the device point (0,0), which, for a client-area device context, corresponds to the upper left corner of the window's client area. The statement above moves the origin of the coordinate system left *m\_nHScrollPos* pixels and upward *m\_nVScrollPos* pixels. If *OnPaint* tries to paint the pixel at (0,0), the coordinate pair is transparently transformed by the GDI into (\_*m\_nHScrollPos*,\_*m\_nVScrollPos*). If the scroll position is (0,100), the first 100 rows of pixels are clipped from the program's output and the *real* output—the output the user can see—begins with the 101st row. Repositioning the origin in this manner is a simple and effective way to move a scrollable window over a virtual display surface.

If you could enlarge the window enough to see the entire spreadsheet, you would see the scroll bars disappear. That's because *CMainWindow::OnSize* sets the scroll bar range to 0 if the window size equals or exceeds the size of the virtual workspace. The *OnSize* handler also updates the scrolling parameters whenever the window size changes so that the thumb size accurately reflects the relative proportions of the window and the virtual workspace.

And with that, all the pieces are in place. The user clicks a scroll bar or drags a scroll bar thumb; *OnHScroll* or *OnVScroll* receives the message and responds by updating the scroll position and scrolling the window; and *OnPaint* redraws the window, using *SetWindowOrg* to move the drawing origin an amount that equals the current scroll position. The program's entire workspace is now accessible, despite the physical limitations that the window size imposes on the output. And all for less than 100 additional lines of code. How could it be any easier?

Funny you should ask. Because that's exactly what MFC's *CScrollView* class is for: to make scrolling easier. *CScrollView* is an MFC class that encapsulates the behavior of a scrolling window. You tell *CScrollView* how large a landscape you wish to view, and it handles everything else. Among other things, *CScrollView* processes WM\_VSCROLL and WM\_HSCROLL messages for you, scrolls the window in response to scroll bar events, and updates the thumb size when the window size changes.

While it's perfectly possible to wire a *CScrollView* into an application like Accel, *CScrollView* was designed primarily for document/view applications. [Chapter 10](mk:@MSITStore:C:\Program%20Files%20(x86)\MSPress\BooksOnline\Programming%20Windows%20with%20MFC%20Second%20Edition\progmfc2.chm::/ch10a.htm) examines *CScrollView* more closely and also introduces some of the other view classes that MFC provides.

# Loose Ends

Before we close out the chapter, we need to tie up one loose end. All the programs presented thus far have created a window with the statement

|  |
| --- |
| m\_pMainWnd = new CMainWindow; |

in *InitInstance*. Since the object is instantiated with *new*, it remains in memory after *InitInstance* ends and, in fact, will not go away until it is deleted with a *delete* statement. Yet nowhere in the programs' source code will you find such a statement. On the surface, this would seem to be a problem. After all, every C++ programmer knows that every *new* must be countered with a *delete* or objects will be left behind in memory. So what gives?

As you probably suspected, the class library deletes the object for you. To be more precise, the object deletes itself. The key to this little trick is that the very last message a window receives before it goes away for good is WM\_NCDESTROY. If you look at the source code for *CWnd::OnNcDestroy*, you'll see that it calls a virtual function named *PostNcDestroy*. *CFrameWnd* overrides *PostNcDestroy* and executes a

|  |
| --- |
| delete this; |

statement. Therefore, when a frame window is destroyed, the object associated with that window is automatically deleted, too. A frame window is destroyed when the user closes the application.

It's worth noting that *CWnd*'s own implementation of *PostNcDestroy* does not delete the associated window object. Therefore, if you derive your own window class directly from *CWnd*, you also need to override *PostNcDestroy* in the derived class and execute a *delete this* statement. Otherwise, the *CWnd* object will not be properly deleted. You'll see what I mean in the next chapter.

Chapter 3

**The Mouse and the Keyboard**

If life were like the movies, traditional input devices would have given way long ago to speech-recognition units, 3D headsets, and other human-machine interface gadgets. At present, however, the two most common input devices remain the mouse and the keyboard. Microsoft Windows handles some mouse and keyboard input itself, automatically dropping down a menu, for example, when the user clicks an item on the menu bar, and sending the application a WM\_COMMAND message when an item is selected from the menu. It's entirely possible to write a full-featured Windows program that processes no mouse or keyboard input directly, but as an application developer, you'll eventually discover the need to read input from the mouse and keyboard directly. And when you do, you'll need to know about the mouse and keyboard interfaces that Windows provides.

Not surprisingly, mouse and keyboard input comes in the form of messages. Device drivers process mouse and keyboard interrupts and place the resultant event notifications in a systemwide queue known as the *raw input queue*. Entries in the raw input queue have WM\_ message identifiers just as conventional messages do, but the data in them requires further processing before it is meaningful to an application. A dedicated thread owned by the operating system monitors the raw input queue and transfers each message that shows up there to the appropriate thread message queue. The "cooking" of the message data is performed later, in the context of the receiving application, and the message is ultimately retrieved and dispatched just as any other message is.

This input model differs from that of 16-bit Windows, which stored mouse and keyboard messages in a single systemwide input queue until they were retrieved by an application. This arrangement proved to be an Achilles' heel of sorts because it meant that an application that failed to dispose of input messages in a timely manner could prevent other applications from doing the same. Win32's asynchronous input model solves this problem by using the raw input queue as a temporary holding buffer and moving input messages to thread message queues at the earliest opportunity. (In 32-bit Windows, each thread that calls certain Windows API functions is given its own message queue, so a multithreaded application can have not one, but many, message queues.) A 32-bit application that goes too long without checking the message queue responds sluggishly to user input, but it doesn't affect the responsiveness of other applications running on the system.

Learning to respond to mouse and keyboard input in a Windows application is largely a matter of learning about which messages to process. This chapter introduces mouse and keyboard messages and the various functions, both in MFC and the API, that are useful for processing them. We'll apply the concepts presented here to the real world by developing three sample applications:

* TicTac, a tic-tac-toe game that demonstrates how to respond to mouse clicks
* MouseCap, a simple drawing program that demonstrates how mouse capturing works and how nonclient-area mouse messages are processed
* VisualKB, a typing program that brings mouse and keyboard handlers together under one roof and lists the keyboard messages it receives

We have a lot of ground to cover, so let's get started.

# Getting Input from the Mouse

Windows uses a number of different messages—more than 20 in all—to report input events involving the mouse. These messages fall into two rather broad categories: client-area mouse messages, which report events that occur in a window's client area, and nonclient-area mouse messages, which pertain to events in a window's nonclient area. An "event" can be any of the following:

* The press or release of a mouse button
* The double click of a mouse button
* The movement of the mouse

You'll typically ignore events in the nonclient area of your window and allow Windows to handle them. If your program processes mouse input, it's client-area mouse messages you'll probably be concerned with.

## Client-Area Mouse Messages

Windows reports mouse events in a window's client area using the messages shown in the following table.

**Client-Area Mouse Messages**

|  |  |
| --- | --- |
| ***Message*** | ***Sent When*** |
| WM\_LBUTTONDOWN | The left mouse button is pressed. |
| WM\_LBUTTONUP | The left mouse button is released. |
| WM\_LBUTTONDBLCLK | The left mouse button is double-clicked. |
| WM\_MBUTTONDOWN | The middle mouse button is pressed. |
| WM\_MBUTTONUP | The middle mouse button is released. |
| WM\_MBUTTONDBLCLK | The middle mouse button is double-clicked. |
| WM\_RBUTTONDOWN | The right mouse button is pressed. |
| WM\_RBUTTONUP | The right mouse button is released. |
| WM\_RBUTTONDBLCLK | The right mouse button is double-clicked. |
| WM\_MOUSEMOVE | The cursor is moved over the window's client area. |

Messages that begin with WM\_LBUTTON pertain to the left mouse button, WM\_MBUTTON messages to the middle mouse button, and WM\_RBUTTON messages to the right mouse button. An application won't receive WM\_MBUTTON messages if the mouse has only two buttons. (This rule has one important exception: mice that have mouse wheels. Mouse wheels are discussed later in this chapter.) An application won't receive WM\_RBUTTON messages if the mouse has just one button. The vast majority of PCs running Windows have two-button mice, so it's reasonably safe to assume that the right mouse button exists. However, if you'd like to be certain (or if you'd like to determine whether there is a third button, too), you can use the Windows *::GetSystemMetrics* API function:

|  |
| --- |
| int nButtonCount = ::GetSystemMetrics (SM\_CMOUSEBUTTONS); |

The return value is the number of mouse buttons, or it is 0 in the unlikely event that a mouse is not installed.

WM\_*x*BUTTONDOWN and WM\_*x*BUTTONUP messages report button presses and releases. A WM\_LBUTTONDOWN message is normally followed by a WM\_LBUTTONUP message, but don't count on that being the case. Mouse messages go to the window under the cursor (the Windows term for the mouse pointer), so if the user clicks the left mouse button over a window's client area and then moves the cursor outside the window before releasing the button, the window receives a WM\_LBUTTONDOWN message but not a WM\_LBUTTONUP message. Many programs react only to button-down messages and ignore button-up messages, in which case the pairing of the two isn't important. If pairing is essential, a program can "capture" the mouse on receipt of a button-down message and release it when a button-up message arrives. In between, all mouse messages, even those pertaining to events outside the window, are directed to the window that performed the capture. This ensures that a button-up message is received no matter where the cursor is when the button is released. Mouse capturing is discussed later in this chapter.

When two clicks of the same button occur within a very short period of time, the second button-down message is replaced by a WM\_*x*BUTTONDBLCLK message. Significantly, this happens only if the window's WNDCLASS includes the class style CS\_DBLCLKS. The default WNDCLASS that MFC registers for frame windows has this style, so frame windows receive double-click messages by default. For a CS\_DBLCLKS-style window, two rapid clicks of the left mouse button over the window's client area produce the following sequence of messages:

|  |
| --- |
| WM\_LBUTTONDOWN  WM\_LBUTTONUP  WM\_LBUTTONDBLCLK  WM\_LBUTTONUP |

If the window is not registered to be notified of double clicks, however, the same two button clicks produce the following sequence of messages:

|  |
| --- |
| WM\_LBUTTONDOWN  WM\_LBUTTONUP  WM\_LBUTTONDOWN  WM\_LBUTTONUP |

How your application responds to these messages—or whether it responds to them at all—is up to you. You should, however, steer away from having clicks and double clicks of the same mouse button carry out two unrelated tasks. A double-click message is always preceded by a single-click message, so the actions that generate the two messages are not easily divorced. Applications that process single and double clicks of the same button typically select an object on the first click and take some action upon that object on the second click. When you double-click a folder in the right pane of the Windows Explorer, for example, the first click selects the folder and the second click opens it.

WM\_MOUSEMOVE messages report that the cursor has moved within the window's client area. As the mouse is moved, the window under the cursor receives a flurry of WM\_MOUSEMOVE messages reporting the latest cursor position. Windows has an interesting way of delivering WM\_MOUSEMOVE messages that prevents slow applications from being overwhelmed by messages reporting every position in the cursor's path. Rather than stuff a WM\_MOUSEMOVE message into the message queue each time the mouse is moved, Windows simply sets a flag in an internal data structure. The next time the application retrieves a message, Windows, seeing that the flag is set, manufactures a WM\_MOUSEMOVE message with the current cursor coordinates. Therefore, an application receives WM\_MOUSEMOVE messages only as often as it can handle them. If the cursor is moved very slowly, every point in its journey is reported unless the application is busy doing other things. But if the cursor is whisked very rapidly across the screen, most applications receive only a handful of WM\_MOUSEMOVE messages.

In an MFC program, message-map entries route mouse messages to class member functions that are provided to handle those messages. The following table lists the message-map macros and message handler names for client-area mouse messages.

**Message-Map Macros and Message Handlers for Client-Area Mouse Messages**

|  |  |  |
| --- | --- | --- |
| ***Message*** | ***Message-Map Macro*** | ***Handling Function*** |
| WM\_LBUTTONDOWN | ON\_WM\_LBUTTONDOWN | *OnLButtonDown* |
| WM\_LBUTTONUP | ON\_WM\_LBUTTONUP | *OnLButtonUp* |
| WM\_LBUTTONDBLCLK | ON\_WM\_LBUTTONDBLCLK | *OnLButtonDblClk* |
| WM\_MBUTTONDOWN | ON\_WM\_MBUTTONDOWN | *OnMButtonDown* |
| WM\_MBUTTONUP | ON\_WM\_MBUTTONUP | *OnMButtonUp* |
| WM\_MBUTTONDBLCLK | ON\_WM\_MBUTTONDBLCLK | *OnMButtonDblClk* |
| WM\_RBUTTONDOWN | ON\_WM\_RBUTTONDOWN | *OnRButtonDown* |
| WM\_RBUTTONUP | ON\_WM\_RBUTTONUP | *OnRButtonUp* |
| WM\_RBUTTONDBLCLK | ON\_WM\_RBUTTONDBLCLK | *OnRButtonDblClk* |
| WM\_MOUSEMOVE | ON\_WM\_MOUSEMOVE | *OnMouseMove* |

*OnLButtonDown* and other client-area mouse message handlers are prototyped as follows:

|  |
| --- |
| afx\_msg void On*MsgName* (UINT nFlags, CPoint point) |

*point* identifies the location of the cursor. In WM\_*x*BUTTONDOWN and WM\_*x*BUTTONDBLCLK messages, *point* specifies the location of the cursor when the button was pressed. In WM\_*x*BUTTONUP messages, *point* specifies the location of the cursor when the button was released. And in WM\_MOUSEMOVE messages, *point* specifies the latest cursor position. In all cases, positions are reported in device coordinates relative to the upper left corner of the window's client area. A WM\_LBUTTONDOWN message with *point.x* equal to 32 and *point.y* equal to 64 means the left mouse button was clicked 32 pixels to the right of and 64 pixels below the client area's upper left corner. If necessary, these coordinates can be converted to logical coordinates using MFC's *CDC::DPtoLP* function.

The *nFlags* parameter specifies the state of the mouse buttons and of the Shift and Ctrl keys at the time the message was generated. You can find out from this parameter whether a particular button or key is up or down by testing for the bit flags listed in the following table.

**The *nFlags* Parameter**

|  |  |
| --- | --- |
| ***Mask*** | ***Meaning If Set*** |
| MK\_LBUTTON | The left mouse button is pressed. |
| MK\_MBUTTON | The middle mouse button is pressed. |
| MK\_RBUTTON | The right mouse button is pressed. |
| MK\_CONTROL | The Ctrl key is pressed. |
| MK\_SHIFT | The Shift key is pressed. |

The expression

|  |
| --- |
| nFlags & MK\_LBUTTON |

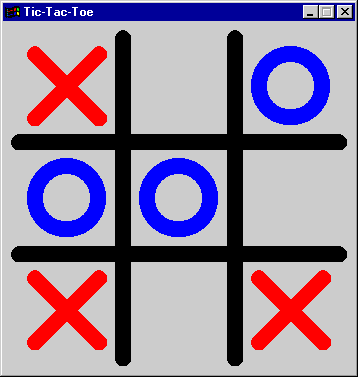
is nonzero if and only if the left mouse button is pressed, while

|  |
| --- |
| nFlags & MK\_CONTROL |

is nonzero if the Ctrl key was held down when the event occurred. Some programs respond differently to mouse events if the Shift or Ctrl key is held down. For example, a drawing program might constrain the user to drawing only horizontal or vertical lines if the Ctrl key is pressed as the mouse is moved by checking the MK\_CONTROL bit in the *nFlags* parameter accompanying WM\_MOUSEMOVE messages. At the conclusion of a drag-and-drop operation, the Windows shell interprets the MK\_CONTROL bit to mean that the objects involved in the drop should be copied rather than moved.

## The TicTac Application

To show how easy it is to process mouse messages, let's look at a sample application that takes input from the mouse. TicTac, whose output is shown in Figure 3-1, is a tic-tac-toe program that responds to three types of client-area mouse events: left button clicks, right button clicks, and left button double clicks. Clicking the left mouse button over an empty square places an X in that square. Clicking the right mouse button places an O in an empty square. (The program prevents cheating by making sure that Xs and Os are alternated.) Double-clicking the left mouse button over the thick black lines that separate the squares clears the playing grid and starts a new game. After each X or O is placed, the program checks to see if there's a winner or the game has been played to a draw. A draw is declared when all nine squares are filled and neither player has managed to claim three squares in a row horizontally, vertically, or diagonally.



**Figure 3-1.** *The TicTac window.*

In addition to providing a hands-on demonstration of mouse-message processing, TicTac also introduces some handy new MFC functions such as *CWnd::MessageBox*, which displays a message box window, and *CRect::PtInRect*, which quickly tells you whether a point lies inside a rectangle represented by a *CRect* object. TicTac's source code appears in Figure 3-2.

**Figure 3-2.** *The TicTac application.*

|  |
| --- |
| TicTac.h #define EX 1  #define OH 2  class CMyApp : public CWinApp  {  public:  virtual BOOL InitInstance ();  };  class CMainWindow : public CWnd  {  protected:  static const CRect m\_rcSquares[9]; // Grid coordinates  int m\_nGameGrid[9]; // Grid contents  int m\_nNextChar; // Next character (EX or OH)  int GetRectID (CPoint point);  void DrawBoard (CDC\* pDC);  void DrawX (CDC\* pDC, int nPos);  void DrawO (CDC\* pDC, int nPos);  void ResetGame ();  void CheckForGameOver ();  int IsWinner ();  BOOL IsDraw ();  public:  CMainWindow ();  protected:  virtual void PostNcDestroy ();  afx\_msg void OnPaint ();  afx\_msg void OnLButtonDown (UINT nFlags, CPoint point);  afx\_msg void OnLButtonDblClk (UINT nFlags, CPoint point);  afx\_msg void OnRButtonDown (UINT nFlags, CPoint point);  DECLARE\_MESSAGE\_MAP ()  }; |

|  |
| --- |
| **TicTac.cpp**  #include <afxwin.h>  #include "TicTac.h"  CMyApp myApp;  /////////////////////////////////////////////////////////////////////////  // CMyApp member functions  BOOL CMyApp::InitInstance ()  {  m\_pMainWnd = new CMainWindow;  m\_pMainWnd->ShowWindow (m\_nCmdShow);  m\_pMainWnd->UpdateWindow ();  return TRUE;  }  /////////////////////////////////////////////////////////////////////////  // CMainWindow message map and member functions  BEGIN\_MESSAGE\_MAP (CMainWindow, CWnd)  ON\_WM\_PAINT ()  ON\_WM\_LBUTTONDOWN ()  ON\_WM\_LBUTTONDBLCLK ()  ON\_WM\_RBUTTONDOWN ()  END\_MESSAGE\_MAP ()  const CRect CMainWindow::m\_rcSquares[9] = {  CRect ( 16, 16, 112, 112),  CRect (128, 16, 224, 112),  CRect (240, 16, 336, 112),  CRect ( 16, 128, 112, 224),  CRect (128, 128, 224, 224),  CRect (240, 128, 336, 224),  CRect ( 16, 240, 112, 336),  CRect (128, 240, 224, 336),  CRect (240, 240, 336, 336)  };  CMainWindow::CMainWindow ()  {  m\_nNextChar = EX;  ::ZeroMemory (m\_nGameGrid, 9 \* sizeof (int));  //  // Register a WNDCLASS.  //  CString strWndClass = AfxRegisterWndClass (  CS\_DBLCLKS, // Class style  AfxGetApp ()->LoadStandardCursor (IDC\_ARROW), // Class cursor  (HBRUSH) (COLOR\_3DFACE + 1), // Background brush  AfxGetApp ()->LoadStandardIcon (IDI\_WINLOGO) // Class icon  );  //  // Create a window.  //  CreateEx (0, strWndClass, \_T ("Tic-Tac-Toe"),  WS\_OVERLAPPED | WS\_SYSMENU | WS\_CAPTION | WS\_MINIMIZEBOX,  CW\_USEDEFAULT, CW\_USEDEFAULT, CW\_USEDEFAULT, CW\_USEDEFAULT,  NULL, NULL);  //  // Size the window.  //  CRect rect (0, 0, 352, 352);  CalcWindowRect (&rect);  SetWindowPos (NULL, 0, 0, rect.Width (), rect.Height (),  SWP\_NOZORDER | SWP\_NOMOVE | SWP\_NOREDRAW);  }  void CMainWindow::PostNcDestroy ()  {  delete this;  }  void CMainWindow::OnPaint ()  {  CPaintDC dc (this);  DrawBoard (&dc);  }  void CMainWindow::OnLButtonDown (UINT nFlags, CPoint point)  {  //  // Do nothing if it's O's turn, if the click occurred outside the  // tic-tac-toe grid, or if a nonempty square was clicked.  //  if (m\_nNextChar != EX)  return;  int nPos = GetRectID (point);  if ((nPos == -1) || (m\_nGameGrid[nPos] != 0))  return;  //  // Add an X to the game grid and toggle m\_nNextChar.  //  m\_nGameGrid[nPos] = EX;  m\_nNextChar = OH;  //  // Draw an X on the screen and see if either player has won.  //  CClientDC dc (this);  DrawX (&dc, nPos);  CheckForGameOver ();  }  void CMainWindow::OnRButtonDown (UINT nFlags, CPoint point)  {  //  // Do nothing if it's X's turn, if the click occurred outside the  // tic-tac-toe grid, or if a nonempty square was clicked.  //  if (m\_nNextChar != OH)  return;  int nPos = GetRectID (point);  if ((nPos == -1) || (m\_nGameGrid[nPos] != 0))  return;  //  // Add an O to the game grid and toggle m\_nNextChar.  //  m\_nGameGrid[nPos] = OH;  m\_nNextChar = EX;  //  // Draw an O on the screen and see if either player has won.  //  CClientDC dc (this);  DrawO (&dc, nPos);  CheckForGameOver ();  }  void CMainWindow::OnLButtonDblClk (UINT nFlags, CPoint point)  {  //  // Reset the game if one of the thick black lines defining the game  // grid is double-clicked with the left mouse button.  //  CClientDC dc (this);  if (dc.GetPixel (point) == RGB (0, 0, 0))  ResetGame ();  }  int CMainWindow::GetRectID (CPoint point)  {  //  // Hit-test each of the grid's nine squares and return a rectangle ID  // (0-8) if (point.x, point.y) lies inside a square.  //  for (int i=0; i<9; i++) {  if (m\_rcSquares[i].PtInRect (point))  return i;  }  return -1;  }  void CMainWindow::DrawBoard (CDC\* pDC)  {  //  // Draw the lines that define the tic-tac-toe grid.  //  CPen pen (PS\_SOLID, 16, RGB (0, 0, 0));  CPen\* pOldPen = pDC->SelectObject (&pen);  pDC->MoveTo (120, 16);  pDC->LineTo (120, 336);  pDC->MoveTo (232, 16);  pDC->LineTo (232, 336);  pDC->MoveTo (16, 120);  pDC->LineTo (336, 120);  pDC->MoveTo (16, 232);  pDC->LineTo (336, 232);  //  // Draw the Xs and Os.  //  for (int i=0; i<9; i++) {  if (m\_nGameGrid[i] == EX)  DrawX (pDC, i);  else if (m\_nGameGrid[i] == OH)  DrawO (pDC, i);  }  pDC->SelectObject (pOldPen);  }  void CMainWindow::DrawX (CDC\* pDC, int nPos)  {  CPen pen (PS\_SOLID, 16, RGB (255, 0, 0));  CPen\* pOldPen = pDC->SelectObject (&pen);  CRect rect = m\_rcSquares[nPos];  rect.DeflateRect (16, 16);  pDC->MoveTo (rect.left, rect.top);  pDC->LineTo (rect.right, rect.bottom);  pDC->MoveTo (rect.left, rect.bottom);  pDC->LineTo (rect.right, rect.top);  pDC->SelectObject (pOldPen);  }  void CMainWindow::DrawO (CDC\* pDC, int nPos)  {  CPen pen (PS\_SOLID, 16, RGB (0, 0, 255));  CPen\* pOldPen = pDC->SelectObject (&pen);  pDC->SelectStockObject (NULL\_BRUSH);  CRect rect = m\_rcSquares[nPos];  rect.DeflateRect (16, 16);  pDC->Ellipse (rect);  pDC->SelectObject (pOldPen);  }  void CMainWindow::CheckForGameOver ()  {  int nWinner;  //  // If the grid contains three consecutive Xs or Os, declare a winner  // and start a new game.  //  if (nWinner = IsWinner ()) {  CString string = (nWinner == EX) ?  \_T ("X wins!") : \_T ("O wins!");  MessageBox (string, \_T ("Game Over"), MB\_ICONEXCLAMATION | MB\_OK);  ResetGame ();  }  //  // If the grid is full, declare a draw and start a new game.  //  else if (IsDraw ()) {  MessageBox (\_T ("It's a draw!"), \_T ("Game Over"),  MB\_ICONEXCLAMATION | MB\_OK);  ResetGame ();  }  }  int CMainWindow::IsWinner ()  {  static int nPattern[8][3] = {  0, 1, 2,  3, 4, 5,  6, 7, 8,  0, 3, 6,  1, 4, 7,  2, 5, 8,  0, 4, 8,  2, 4, 6  };  for (int i=0; i<8; i++) {  if ((m\_nGameGrid[nPattern[i][0]] == EX) &&  (m\_nGameGrid[nPattern[i][1]] == EX) &&  (m\_nGameGrid[nPattern[i][2]] == EX))  return EX;  if ((m\_nGameGrid[nPattern[i][0]] == OH) &&  (m\_nGameGrid[nPattern[i][1]] == OH) &&  (m\_nGameGrid[nPattern[i][2]] == OH))  return OH;  }  return 0;  }  BOOL CMainWindow::IsDraw ()  {  for (int i=0; i<9; i++) {  if (m\_nGameGrid[i] == 0)  return FALSE;  }  return TRUE;  }  void CMainWindow::ResetGame ()  {  m\_nNextChar = EX;  ::ZeroMemory (m\_nGameGrid, 9 \* sizeof (int));  Invalidate ();  } |

The first step in processing mouse input is to add entries for the messages you want to handle to the message map. *CMainWindow*'s message map in TicTac.cpp contains the following message-map entries:

|  |
| --- |
| ON\_WM\_LBUTTONDOWN ()  ON\_WM\_LBUTTONDBLCLK ()  ON\_WM\_RBUTTONDOWN () |

These three statements correlate WM\_LBUTTONDOWN, WM\_LBUTTONDBLCLK, and WM\_RBUTTONDOWN messages to the *CMainWindow* member functions *OnLButtonDown*, *OnLButtonDblClk*, and *OnRButtonDown*. When the messages start arriving, the fun begins.

The *OnLButtonDown* handler processes clicks of the left mouse button in *CMainWindow*'s client area. After checking *m\_nNextChar* to verify that it's X's turn and not O's (and returning without doing anything if it's not), *OnLButtonDown* calls the protected member function *GetRectID* to determine whether the click occurred in one of the nine rectangles corresponding to squares in the tic-tac-toe grid. The rectangles' coordinates are stored in the static array of *CRect* objects named *CMainWindow::m\_rcSquares*. *GetRectID* uses a *for* loop to determine whether the cursor location passed to it by the message handler lies inside any of the squares:

|  |
| --- |
| for (int i=0; i<9; i++) {  if (m\_rcSquares[i].PtInRect (point))  return i;  }  return -1; |

*CRect::PtInRect* returns a nonzero value if the point passed to it lies within the rectangle represented by the *CRect* object, or 0 if it does not. If *PtInRect* returns nonzero for any of the rectangles in the *m\_rcSquares* array, *GetRectID* returns the rectangle ID. The ID is an integer from 0 through 8, with 0 representing the square in the upper left corner of the grid, 1 the square to its right, 2 the square in the upper right corner, 3 the leftmost square in the second row, and so on. Each square has a corresponding element in the *m\_nGameGrid* array, which initially holds all zeros representing empty squares. If none of the calls to *PtInRect* returns TRUE, *GetRectID* returns -1 to indicate that the click occurred outside the squares and *OnLButtonDown* ignores the mouse click. If, however, *GetRectID* returns a valid ID and the corresponding square is empty, *OnLButtonDown* records the X in the *m\_nGameGrid* array and calls *CMainWindow::DrawX* to draw an X in the square. *DrawX* creates a red pen 16 pixels wide and draws two perpendicular lines oriented at 45-degree angles.

*OnRButtonDown* works in much the same way as *OnLButtonDown*, except that it draws an O instead of an X. The routine that does the drawing is *CMainWindow::DrawO*. Before it draws an O with the *CDC::Ellipse* function, *DrawO* selects a NULL brush into the device context:

|  |
| --- |
| pDC->SelectStockObject (NULL\_BRUSH); |

This prevents the interior of the O from being filled with the device context's default white brush. (As an alternative, we could have created a brush whose color matched the window's background color and selected it into the device context. But drawing with a NULL brush is slightly faster because it produces no physical screen output.) The O is then drawn with the statements

|  |
| --- |
| CRect rect = m\_rcSquares[nPos];  rect.DeflateRect (16, 16);  pDC->Ellipse (rect); |

The first statement copies the rectangle representing the grid square to a local *CRect* object named *rect*; the second uses *CRect::DeflateRect* to "deflate" the rectangle by 16 pixels in each direction and form the circle's bounding box; and the third draws the circle. The result is a nicely formed O that's centered in the square in which it is drawn.

Double-clicking the grid lines separating the squares clears the Xs and Os and begins a new game. While this is admittedly a poor way to design a user interface, it does provide an excuse to write a double-click handler. (A better solution would be a push button control with the words *New Game* stamped on it or a New Game menu item, but since we haven't covered menus and controls yet, the perfect user interface will just have to wait.) Left mouse button double clicks are processed by *CMainWindow::OnLButtonDblClk*, which contains these simple statements:

|  |
| --- |
| CClientDC dc (this);  if (dc.GetPixel (point) == RGB (0, 0, 0))  ResetGame (); |

To determine whether the double click occurred over the thick black strokes separating the squares in the playing grid, *OnLButtonDblClk* calls *CDC::GetPixel* to get the color of the pixel under the cursor and compares it to black (RGB (0, 0, 0)). If there's a match, *ResetGame* is called to reset the game. Otherwise, *OnLButtonDblClk* returns and the double click is ignored. Testing the color of the pixel under the cursor is an effective technique for hit-testing irregularly shaped areas, but be wary of using nonprimary colors that a display driver is likely to dither. Pure black (RGB (0, 0, 0)) and pure white (RGB (255, 255, 255)) are supported on every PC that runs Windows, so you can safely assume that neither of these colors will be dithered.

To be consistent with published user interface guidelines, applications should not use the right mouse button to carry out application-specific tasks as TicTac does. Instead, they should respond to right mouse clicks by popping up context menus. When a WM\_RBUTTONUP message is passed to the system for default processing, Windows places a WM\_CONTEXTMENU message in the message queue. You'll learn more about this feature of the operating system in the next chapter.

### Message Boxes

Before returning, TicTac's *OnLButtonDown* and *OnRButtonDown* handlers call *CMainWindow::CheckForGameOver* to find out if the game has been won or played to a draw. If either player has managed to align three Xs or Os in a row or if no empty squares remain, *CheckForGameOver* calls *CMainWindow*'s *MessageBox* function to display a message box announcing the outcome, as shown in Figure 3-3. *MessageBox* is a function that all window classes inherit from *CWnd*. It is an extraordinarily useful tool to have at your disposal because it provides a one-step means for displaying a message on the screen and optionally obtaining a response.



**Figure 3-3.** *A Windows message box.*

*CWnd::MessageBox* is prototyped as follows:

|  |
| --- |
| int MessageBox (LPCTSTR lpszText, LPCTSTR lpszCaption = NULL,  UINT nType = MB\_OK) |

*lpszText* specifies the text in the body of the message box, *lpszCaption* specifies the caption for the message box's title bar, and *nType* contains one or more bit flags defining the message box's style. The return value identifies the button that was clicked to dismiss the message box. *lpszText* and *lpszCaption* can be *CString* objects or pointers to conventional text strings. (Because the *CString* class overloads the LPCTSTR operator, you can always pass a *CString* to a function that accepts an LPCTSTR data type.) A NULL *lpszCaption* value displays the caption "Error" in the title bar.

The simplest use for *MessageBox* is to display a message and pause until the user clicks the message box's OK button:

|  |
| --- |
| MessageBox (\_T ("Click OK to continue"), \_T ("My Application")); |

Accepting the default value for *nType* (MB\_OK) means the message box will have an OK button but no other buttons. Consequently, the only possible return value is IDOK. But if you want to use a message box to ask the user whether to save a file before exiting the application, you can use the MB\_YESNOCANCEL style:

|  |
| --- |
| MessageBox (\_T ("Your document contains unsaved data. Save it?"),  \_T ("My Application"), MB\_YESNOCANCEL); |

Now the message box contains three buttons—Yes, No, and Cancel—and the value returned from the *MessageBox* function is IDYES, IDNO, or IDCANCEL. The program can then test the return value and save the data before closing (IDYES), close without saving (IDNO), or return to the application without shutting down (IDCANCEL). The table below lists the six message box types and the corresponding return values, with the default push button—the one that's "clicked" if the user presses the Enter key—highlighted in boldface type.

**Message Box Types**

|  |  |  |
| --- | --- | --- |
| ***Type*** | ***Buttons*** | ***Possible Return Codes*** |
| MB\_ABORTRETRYIGNORE | **Abort**, Retry, Ignore | IDABORT, IDRETRY, IDIGNORE |
| MB\_OK | **OK** | IDOK |
| MB\_OKCANCEL | **OK**, Cancel | IDOK, IDCANCEL |
| MB\_RETRYCANCEL | **Retry**, Cancel | IDRETRY, IDCANCEL |
| MB\_YESNO | **Yes**, No | IDYES, IDNO |
| MB\_YESNOCANCEL | **Yes**, No, Cancel | IDYES, IDNO, IDCANCEL |

In message boxes with multiple buttons, the first (leftmost) button is normally the default push button. You can make the second or third button the default by ORing MB\_DEFBUTTON2 or MB\_DEFBUTTON3 into the value that specifies the message box type. The statement

|  |
| --- |
| MessageBox (\_T ("Your document contains unsaved data. Save it?"),  \_T ("My Application"), MB\_YESNOCANCEL ¦ MB\_DEFBUTTON3); |

displays the same message box as before but makes Cancel the default action.

By default, message boxes are application modal, which means the application that called the *MessageBox* function is disabled until the message box is dismissed. You can add MB\_SYSTEMMODAL to the *nType* parameter and make the message box system modal. In 16-bit Windows, system-modal means that input to *all* applications is suspended until the message box is dismissed. In the Win32 environment, Windows makes the message box a topmost window that stays on top of other windows, but the user is still free to switch to other applications. System-modal message boxes should be used only for serious errors that demand immediate attention.

You can add an artistic touch to your message boxes by using MB\_ICON identifiers. MB\_ICONINFORMATION displays a small text balloon with an "i" for "information" in it in the upper left corner of the message box. The "i" is generally used when information is provided to the user but no questions are being asked, as in

|  |
| --- |
| MessageBox (\_T ("No errors found. Click OK to continue"),  \_T ("My Application"), MB\_ICONINFORMATION ¦ MB\_OK); |

MB\_ICONQUESTION displays a question mark instead of an "i" and is normally used for queries such as "Save before closing?" MB\_ICONSTOP displays a red circle with an X and usually indicates that an unrecoverable error has occurred—for example, an out-of-memory error is forcing the program to terminate prematurely. Finally, MB\_ICONEXCLAMATION displays a yellow triangle containing an exclamation mark. (See Figure 3-3.)

MFC provides an alternative to *CWnd::MessageBox* in the form of the global *AfxMessageBox* function. The two are similar, but *AfxMessageBox* can be called from application classes, document classes, and other non-window classes. One situation in which *AfxMessageBox* is irreplaceable is when you want to report an error in the application object's *InitInstance* function. *MessageBox* requires a valid *CWnd* pointer and therefore can't be called until after a window is created. *AfxMessageBox*, on the other hand, can be called at any time.

### What? No Frame Window?

TicTac differs from the sample programs in Chapters 1 and 2 in one important respect: Rather than using a frame window for its main window, it derives its own window class from *CWnd*. It's not that a *CFrameWnd* wouldn't work; it's that *CWnd* has everything TicTac needs and more. *CWnd* is the root of all window classes in MFC. Depending on what kinds of applications you write, deriving from *CWnd* is something you might need to do often or not at all. Still, it's something every MFC programmer should know *how* to do, and seeing a window class derived from *CWnd* also helps to underscore the point that MFC programs don't have to use frame windows.

Creating your own *CWnd*-derived window class is simple. For starters, you derive the window class from *CWnd* instead of from *CFrameWnd*. In the BEGIN\_MESSAGE\_MAP macro, be sure to specify *CWnd*, not *CFrameWnd*, as the base class. Then, in the window's constructor, use *AfxRegisterWndClass* to register a WNDCLASS and call *CWnd::CreateEx* to create the window. Remember the beginning of Chapter 1, where we looked at the C source code for an SDK-style Windows application? Before creating a window, *WinMain* initialized a WNDCLASS structure with values describing the window's class attributes and then called *::RegisterClass* to register the WNDCLASS. Normally you don't register a WNDCLASS in an MFC program because MFC registers one for you. Specifying NULL in the first parameter to *CFrameWnd::Create* accepts the default WNDCLASS. When you derive from *CWnd*, however, you must register your own WNDCLASS because *CWnd::CreateEx* does not accept a NULL WNDCLASS name.

### The *AfxRegisterWndClass* Function

MFC makes WNDCLASS registration easy with its global *AfxRegisterWndClass* function. If you use *::RegisterClass* or MFC's *AfxRegisterClass* to register a WNDCLASS, you must initialize every field in the WNDCLASS structure. But *AfxRegisterWndClass* fills in most of the fields for you, leaving you to specify values for just the four that MFC applications are typically concerned with. *AfxRegisterWndClass* is prototyped as follows:

|  |
| --- |
| LPCTSTR AfxRegisterWndClass (UINT nClassStyle, HCURSOR hCursor = 0,  HBRUSH hbrBackground = 0, HICON hIcon = 0) |

The value returned by *AfxRegisterWndClass* is a pointer to a null-terminated string containing the WNDCLASS name. Before seeing how TicTac uses *AfxRegisterWndClass*, let's take a closer look at the function itself and the parameters it accepts.

*nClassStyle* specifies the class style, which defines certain behavioral characteristics of a window. *nClassStyle* is a combination of zero or more of the bit flags shown in the following table.

**WNDCLASS Style Flags**

|  |  |
| --- | --- |
| ***Class Style*** | ***Description*** |
| CS\_BYTEALIGNCLIENT | Ensures that a window's client area is always aligned on a byte boundary in the video buffer to speed drawing operations. |
| CS\_BYTEALIGNWINDOW | Ensures that the window itself is always aligned on a byte boundary in the video buffer to speed moving and resizing operations. |
| CS\_CLASSDC | Specifies that the window should share a device context with other windows created from the same WNDCLASS. |
| CS\_DBLCLKS | Specifies that the window should be notified of double clicks with WM\_*x*BUTTONDBLCLK messages. |
| CS\_GLOBALCLASS | Registers the WNDCLASS globally so that all applications can use it. (By default, only the application that registers a WNDCLASS can create windows from it.) Used primarily for child window controls. |
| CS\_HREDRAW | Specifies that the entire client area should be invalidated when the window is resized horizontally. |
| CS\_NOCLOSE | Disables the Close command on the system menu and the close button on the title bar. |
| CS\_OWNDC | Specifies that each window created from this WNDCLASS should have its own device context. Helpful when optimizing repaint performance because an application doesn't have to reinitialize a private device context each time the device context is acquired. |
| CS\_PARENTDC | Specifies that a child window should inherit the device context of its parent. |
| CS\_SAVEBITS | Specifies that areas of the screen covered by windows created from this WNDCLASS should be saved in bitmap form for quick repainting. Used primarily for menus and other windows with short life spans. |
| CS\_VREDRAW | Specifies that the entire client area should be invalidated when the window is resized vertically. |

The CS\_BYTEALIGNCLIENT and CS\_BYTEALIGNWINDOW styles were useful back in the days of dumb frame buffers and monochrome video systems, but they are largely obsolete today. CS\_CLASSDC, CS\_OWNDC, and CS\_PARENTDC are used to implement special handling of device contexts. You'll probably use CS\_GLOBALCLASS only if you write custom controls to complement list boxes, push buttons, and other built-in control types. The CS\_HREDRAW and CS\_VREDRAW styles are useful for creating resizeable windows whose content scales with the window size.

*hCursor* identifies the "class cursor" for windows created from this WNDCLASS. When the cursor moves over a window's client area, Windows retrieves the class cursor's handle from the window's WNDCLASS and uses it to draw the cursor image. You can create custom cursors using an icon editor, or you can use the predefined system cursors that Windows provides. *CWinApp::LoadStandardCursor* loads a system cursor. The statement

|  |
| --- |
| AfxGetApp ()->LoadStandardCursor (IDC\_ARROW); |

returns the handle of the arrow cursor that most Windows applications use. For a complete list of system cursors, see the documentation for *CWinApp::LoadStandardCursor* or the *::LoadCursor* API function. Generally speaking, only the IDC\_ARROW, IDC\_IBEAM, and IDC\_CROSS cursors are useful as class cursors.

The *hbrBackground* parameter passed to *AfxRegisterWndClass* defines the window's default background color. Specifically, *hbrBackground* identifies the GDI brush that is used to erase the window's interior each time a WM\_ERASEBKGND message arrives. A window receives a WM\_ERASEBKGND message when it calls *::BeginPaint* in response to a WM\_PAINT message. If you don't process WM\_ERASEBKGND messages yourself, Windows processes them for you by retrieving the class background brush and using it to fill the window's client area. (You can create custom window backgrounds—for example, backgrounds formed from bitmap images—by processing WM\_ERASEBKGND messages yourself and returning a nonzero value. The nonzero return prevents Windows from painting the background and overwriting what you wrote.) You can either provide a brush handle for *hbrBackground* or specify one of the predefined Windows system colors with the value 1 added to it, as in COLOR\_WINDOW+1 or COLOR\_APPWORKSPACE+1. See the documentation for the *::GetSysColor* API function for a complete list of system colors.

The final *AfxRegisterWndClass* parameter, *hIcon*, specifies the handle of the icon that Windows uses to represent the application on the desktop, in the taskbar, and elsewhere. You can create a custom icon for your application and load it with *CWinApp::LoadIcon*, or you can load a predefined system icon with *CWinApp::LoadStandardIcon*. You can even load icons from other executable files using the *::ExtractIcon* API function.

Here's what the code to register a custom WNDCLASS looks like in TicTac.cpp:

|  |
| --- |
| CString strWndClass = AfxRegisterWndClass (  CS\_DBLCLKS,  AfxGetApp ()->LoadStandardCursor (IDC\_ARROW),  (HBRUSH) (COLOR\_3DFACE + 1),  AfxGetApp ()->LoadStandardIcon (IDI\_WINLOGO)  ); |

The class style CS\_DBLCLKS registers the TicTac window to receive double-click messages. IDC\_ARROW tells Windows to display the standard arrow when the cursor is over the TicTac window, and IDI\_WINLOGO is one of the standard icons that Windows makes available to all applications. COLOR\_3DFACE+1 assigns the TicTac window the same background color as push buttons, dialog boxes, and other 3D display elements. COLOR\_3DFACE defaults to light gray, but you can change the color by using the system's Display Properties property sheet. Using COLOR\_3DFACE for the background color gives your window the same 3D look as a dialog box or message box *and* enables it to adapt to changes in the Windows color scheme.

### *AfxRegisterWndClass* and Frame Windows

The *AfxRegisterWndClass* function isn't only for applications that derive window classes from *CWnd*; you can also use it to register custom WNDCLASSes for frame windows. The default WNDCLASS that MFC registers for frame windows has the following attributes:

* *nClassStyle* = CS\_DBLCLKS ¦ CS\_HREDRAW ¦ CS\_VREDRAW
* *hCursor* = The handle of the predefined cursor IDC\_ARROW
* *hbrBackground* = COLOR\_WINDOW+1
* *hIcon* = The handle of the icon whose resource ID is AFX\_IDI\_STD\_FRAME or AFX\_IDI\_STD\_MDIFRAME, or the system icon ID IDI\_APPLICATION if no such resource is defined

Suppose you want to create a *CFrameWnd* frame window that lacks the CS\_DBLCLKS style, that uses the IDI\_WINLOGO icon, and that uses COLOR\_APPWORKSPACE as its default background color. Here's how to create a frame window that meets these qualifications:

|  |
| --- |
| CString strWndClass = AfxRegisterWndClass (  CS\_HREDRAW ¦ CS\_VREDRAW,  AfxGetApp ()->LoadStandardCursor (IDC\_ARROW),  (HBRUSH) (COLOR\_APPWORKSPACE + 1),  AfxGetApp ()->LoadStandardIcon (IDI\_WINLOGO)  );  Create (strWndClass, \_T ("My Frame Window")); |

These statements replace the

|  |
| --- |
| Create (NULL, \_T ("My Frame Window")); |

statement that normally appears in a frame window's constructor.

### More About the TicTac Window

After registering a WNDCLASS, TicTac creates its main window with a call to *CWnd::CreateEx*:

|  |
| --- |
| CreateEx (0, strWndClass, \_T ("Tic-Tac-Toe"),  WS\_OVERLAPPED ¦ WS\_SYSMENU ¦ WS\_CAPTION ¦ WS\_MINIMIZEBOX,  CW\_USEDEFAULT, CW\_USEDEFAULT, CW\_USEDEFAULT, CW\_USEDEFAULT,  NULL, NULL); |

The first parameter specifies the extended window style and is a combination of zero or more WS\_EX flags. TicTac requires no extended window styles, so this parameter is 0. The second parameter is the WNDCLASS name returned by *AfxRegisterWndClass*, and the third is the window title. The fourth is the window style. The combination of WS\_OVERLAPPED, WS\_SYSMENU, WS\_CAPTION, and WS\_MINIMIZEBOX creates a window that resembles a WS\_OVERLAPPEDWINDOW-style window but lacks a maximize button and can't be resized. What is it about the window that makes it nonresizeable? Look up the definition of WS\_OVERLAPPEDWINDOW in Winuser.h (one of several large header files that comes with Visual C++), and you'll see something like this:

|  |
| --- |
| #define WS\_OVERLAPPEDWINDOW (WS\_OVERLAPPED ¦ WS\_CAPTION ¦ \  WS\_SYSMENU ¦ WS\_THICKFRAME ¦ WS\_MINIMIZE ¦ WS\_MAXIMIZE) |

The WS\_THICKFRAME style adds a resizing border whose edges and corners can be grabbed and dragged with the mouse. TicTac's window lacks this style, so the user can't resize it.

The next four parameters passed to *CWnd::CreateEx* specify the window's initial position and size. TicTac uses CW\_USEDEFAULT for all four so that Windows will pick the initial position and size. Yet clearly the TicTac window is not arbitrarily sized; it is sized to match the playing grid. But how? The statements following the call to *CreateEx* hold the answer:

|  |
| --- |
| CRect rect (0, 0, 352, 352);  CalcWindowRect (&rect);  SetWindowPos (NULL, 0, 0, rect.Width (), rect.Height (),  SWP\_NOZORDER ¦ SWP\_NOMOVE ¦ SWP\_NOREDRAW); |

The first of these statements creates a *CRect* object that holds the desired size of the window's client area—352 by 352 pixels. It wouldn't do to pass these values directly to *CreateEx* because *CreateEx*'s sizing parameters specify the size of the entire window, not just its client area. Since the sizes of the various elements in the window's nonclient area (for example, the height of the title bar) vary with different video drivers and display resolutions, we must calculate the size of the window rectangle from the client rectangle and then size the window to fit.

MFC's *CWnd::CalcWindowRect* is the perfect tool for the job. Given a pointer to a *CRect* object containing the coordinates of a window's client area, *CalcWindowRect* calculates the corresponding window rectangle. The width and height of that rectangle can then be passed to *CWnd::SetWindowPos* to effect the proper window size. The only catch is that *CalcWindowRect* must be called *after* the window is created so that it can factor in the dimensions of the window's nonclient area.

### The *PostNcDestroy* Function

Something you must consider when you derive your own window class from *CWnd* is that once created, the window object must somehow be deleted. As described in Chapter 2, the last message a window receives before it is destroyed is WM\_NCDESTROY. MFC's *CWnd* class includes a default *OnNcDestroy* handler that performs some routine cleanup chores and then, as its very last act, calls a virtual function named *PostNcDestroy*. *CFrameWnd* objects delete themselves when the windows they are attached to are destroyed; they do this by overriding *PostNcDestroy* and executing a *delete this* statement. *CWnd::PostNcDestroy* does not perform a *delete this*, so a class derived from *CWnd* should provide its own version of *PostNcDestroy* that does. TicTac includes a trivial *PostNcDestroy* function that destroys the *CMainWindow* object just before the program terminates:

|  |
| --- |
| void CMainWindow::PostNcDestroy ()  {  delete this;  } |

The question of "who deletes it" is something you should think about whenever you derive a window class from *CWnd*. One alternative to overriding *CWnd::PostNcDestroy* is to override *CWinApp::ExitInstance* and call *delete* on the pointer stored in *m\_pMainWnd*.

## Nonclient-Area Mouse Messages

When the mouse is clicked inside or moved over a window's nonclient area, Windows sends the window a nonclient-area mouse message. The following table lists the nonclient-area mouse messages.

**Nonclient-Area Mouse Messages**

|  |  |
| --- | --- |
| ***Message*** | ***Sent When*** |
| WM\_NCLBUTTONDOWN | The left mouse button is pressed. |
| WM\_NCLBUTTONUP | The left mouse button is released. |
| WM\_NCLBUTTONDBLCLK | The left mouse button is double-clicked. |
| WM\_NCMBUTTONDOWN | The middle mouse button is pressed. |
| WM\_NCMBUTTONUP | The middle mouse button is released. |
| WM\_NCMBUTTONDBLCLK | The middle mouse button is double-clicked. |
| WM\_NCRBUTTONDOWN | The right mouse button is pressed. |
| WM\_NCRBUTTONUP | The right mouse button is released. |
| WM\_NCRBUTTONDBLCLK | The right mouse button is double-clicked. |
| WM\_NCMOUSEMOVE | The cursor is moved over the window's nonclient area. |

Notice the parallelism between the client-area mouse messages shown in the table below and the nonclient-area mouse messages; the only difference is the letters *NC* in the message ID. Unlike double-click messages in a window's client area, WM\_NC*x*BUTTONDBLCLK messages are transmitted regardless of whether the window was registered with the CS\_DBLCLKS style.

As with client-area mouse messages, message-map entries route messages to the appropriate class member functions. The following table lists the message-map macros and message handlers for nonclient-area mouse messages.

**Message-Map Macros and Message Handlers for Nonclient-Area Mouse Messages**

|  |  |  |
| --- | --- | --- |
| ***Message*** | ***Message-Map Macro*** | ***Handling Function*** |
| WM\_NCLBUTTONDOWN | ON\_WM\_NCLBUTTONDOWN | *OnNcLButtonDown* |
| WM\_NCLBUTTONUP | ON\_WM\_NCLBUTTONUP | *OnNcLButtonUp* |
| WM\_NCLBUTTONDBLCLK | ON\_WM\_NCLBUTTONDBLCLK | *OnNcLButtonDblClk* |
| WM\_NCMBUTTONDOWN | ON\_WM\_NCMBUTTONDOWN | *OnNcMButtonDown* |
| WM\_NCMBUTTONUP | ON\_WM\_NCMBUTTONUP | *OnNcMButtonUp* |
| WM\_NCMBUTTONDBLCLK | ON\_WM\_NCMBUTTONDBLCLK | *OnNcMButtonDblClk* |
| WM\_NCRBUTTONDOWN | ON\_WM\_NCRBUTTONDOWN | *OnNcRButtonDown* |
| WM\_NCRBUTTONUP | ON\_WM\_NCRBUTTONUP | *OnNcRButtonUp* |
| WM\_NCRBUTTONDBLCLK | ON\_WM\_NCRBUTTONDBLCLK | *OnNcRButtonDblClk* |
| WM\_NCMOUSEMOVE | ON\_WM\_NCMOUSEMOVE | *OnNcMouseMove* |

Message handlers for nonclient-area mouse messages are prototyped this way:

|  |
| --- |
| afx\_msg void On*MsgName* (UINT nHitTest, CPoint point) |

Once again, the *point* parameter specifies the location in the window at which the event occurred. But for nonclient-area mouse messages, *point.x* and *point.y* contain screen coordinates rather than client coordinates. In screen coordinates, (0,0) corresponds to the upper left corner of the screen, the positive *x* and *y* axes point to the right and down, and one unit in any direction equals one pixel. If you want, you can convert screen coordinates to client coordinates with *CWnd::ScreenToClient*. The *nHitTest* parameter contains a hit-test code that identifies where in the window's nonclient area the event occurred. Some of the most interesting hit-test codes are shown in the following table. You'll find a complete list of hit-test codes in the documentation for WM\_NCHITTEST or *CWnd::OnNcHitTest*.

**Commonly Used Hit-Test Codes**

|  |  |
| --- | --- |
| ***Value*** | ***Corresponding Location*** |
| HTCAPTION | The title bar |
| HTCLOSE | The close button |
| HTGROWBOX | The restore button (same as HTSIZE) |
| HTHSCROLL | The window's horizontal scroll bar |
| HTMENU | The menu bar |
| HTREDUCE | The minimize button |
| HTSIZE | The restore button (same as HTGROWBOX) |
| HTSYSMENU | The system menu box |
| HTVSCROLL | The window's vertical scroll bar |
| HTZOOM | The maximize button |

Programs don't usually process nonclient-area mouse messages; they allow Windows to process them instead. Windows provides appropriate default responses that frequently result in still more messages being sent to the window. For example, when Windows processes a WM\_NCLBUTTONDBLCLK message with a hit-test value equal to HTCAPTION, it sends the window a WM\_SYSCOMMAND message with *wParam* equal to SC\_MAXIMIZE or SC\_RESTORE to maximize or unmaximize the window. You can prevent double clicks on a title bar from affecting a window by including the following message handler in the window class:

|  |
| --- |
| // In CMainWindow's message map  ON\_WM\_NCLBUTTONDBLCLK ()      void CMainWindow::OnNcLButtonDblClk (UINT nHitTest, CPoint point)  {  if (nHitTest != HTCAPTION)  CWnd::OnNcLButtonDblClk (nHitTest, point);  } |

Calling the base class's *OnNcLButtonDblClk* handler passes the message to Windows and allows default processing to take place. Returning without calling the base class prevents Windows from knowing that the double click occurred. You can use other hit-test values to customize the window's response to other nonclient-area mouse events.

## The WM\_NCHITTEST Message

Before a window receives a client-area or nonclient-area mouse message, it receives a WM\_NCHITTEST message accompanied by the cursor's screen coordinates. Most applications don't process WM\_NCHITTEST messages, instead electing to let Windows process them. When Windows processes a WM\_NCHITTEST message, it uses the cursor coordinates to determine what part of the window the cursor is over and then generates either a client-area or nonclient-area mouse message.

One clever use of an *OnNcHitTest* handler is for substituting the HTCAPTION hit-test code for HTCLIENT, which creates a window that can be dragged by its client area:

|  |
| --- |
| // In CMainWindow's message map  ON\_WM\_NCHITTEST ()    UINT CMainWindow::OnNcHitTest (CPoint point)  {  UINT nHitTest = CFrameWnd::OnNcHitTest (point);  if (nHitTest == HTCLIENT)  nHitTest = HTCAPTION;  return nHitTest;  } |

As this example demonstrates, WM\_NCHITTEST messages that you don't process yourself should be forwarded to the base class so that other aspects of the program's operation aren't affected.

## The WM\_MOUSELEAVE and WM\_MOUSEHOVER Messages

It's easy to tell when the cursor enters a window or moves over it because the window receives WM\_MOUSEMOVE messages. The *::TrackMouseEvent* function, which debuted in Windows NT 4.0 and is also supported in Windows 98, makes it equally easy to determine when the cursor leaves a window or hovers motionlessly over the top of it. With *::TrackMouseEvent*, an application can register to receive WM\_MOUSELEAVE messages when the cursor leaves a window and WM\_MOUSEHOVER messages when the cursor hovers over a window.

*::TrackMouseEvent* accepts just one parameter: a pointer to a TRACKMOUSEEVENT structure. The structure is defined this way in Winuser.h:

|  |
| --- |
| typedef struct tagTRACKMOUSEEVENT {  DWORD cbSize;  DWORD dwFlags;  HWND hwndTrack;  DWORD dwHoverTime;  } TRACKMOUSEEVENT; |

*cbSize* holds the size of the structure. *dwFlags* holds bit flags specifying what the caller wants to do: register to receive WM\_MOUSELEAVE messages (TME\_LEAVE), register to receive WM\_MOUSEHOVER messages (TME\_HOVER), cancel WM\_MOUSELEAVE and WM\_MOUSEHOVER messages (TME\_CANCEL), or have the system fill the TRACKMOUSEEVENT structure with the current *::TrackMouseEvent* settings (TME\_QUERY). *hwndTrack* is the handle of the window for which WM\_MOUSELEAVE and WM\_MOUSEHOVER messages are generated. *dwHoverTime* is the length of time in milliseconds that the cursor must pause before a WM\_MOUSEHOVER message is sent to the underlying window. You can accept the system default of 400 milliseconds by setting *dwHoverTime* equal to HOVER\_DEFAULT.

The cursor doesn't have to be perfectly still for the system to generate a WM\_MOUSEHOVER message. If the cursor stays within a rectangle whose width and height equal the values returned by *::SystemParametersInfo* when it's called with SPI\_GETMOUSEHOVERWIDTH and SPI\_GETMOUSEHOVERHEIGHT values, and if it stays there for the number of milliseconds returned by *::SystemParametersInfo* when it's called with an SPI\_GETMOUSEHOVERTIME value, a WM\_MOUSEHOVER message ensues. If you want, you can change these parameters by calling *::SystemParametersInfo* with SPI\_SETMOUSEHOVERWIDTH, SPI\_SETMOUSEHOVERHEIGHT, and SPI\_SETMOUSEHOVERTIME values.

One of the more interesting aspects of *::TrackMouseEvent* is that its effects are cancelled when a WM\_MOUSELEAVE or WM\_MOUSEHOVER message is generated. This means that if you want to receive these message anytime the cursor exits or pauses over a window, you must call *::TrackMouseEvent* again whenever a WM\_MOUSELEAVE or WM\_MOUSEHOVER message is received. To illustrate, the following code snippet writes "Mouse enter," "Mouse leave," or "Mouse hover" to the debug output window anytime the mouse enters, leaves, or pauses over a window. *m\_bMouseOver* is a BOOL *CMainWindow* member variable. It should be set to FALSE in the class constructor:

|  |
| --- |
| // In the message map  ON\_WM\_MOUSEMOVE ()  ON\_MESSAGE (WM\_MOUSELEAVE, OnMouseLeave)  ON\_MESSAGE (WM\_MOUSEHOVER, OnMouseHover)    void CMainWindow::OnMouseMove (UINT nFlags, CPoint point)  {  if (!m\_bMouseOver) {  TRACE (\_T ("Mouse enter\n"));  m\_bMouseOver = TRUE;  TRACKMOUSEEVENT tme;  tme.cbSize = sizeof (tme);  tme.dwFlags = TME\_HOVER | TME\_LEAVE;  tme.hwndTrack = m\_hWnd;  tme.dwHoverTime = HOVER\_DEFAULT;  ::TrackMouseEvent (&tme);  }  }  LRESULT CMainWindow::OnMouseLeave (WPARAM wParam, LPARAM lParam)  {  TRACE (\_T ("Mouse leave\n"));  m\_bMouseOver = FALSE;  return 0;  }  LRESULT CMainWindow::OnMouseHover (WPARAM wParam, LPARAM lParam)  {  TRACE (\_T ("Mouse hover (x=%d, y=%d)\n"),  LOWORD (lParam), HIWORD (lParam));  TRACKMOUSEEVENT tme;  tme.cbSize = sizeof (tme);  tme.dwFlags = TME\_HOVER | TME\_LEAVE;  tme.hwndTrack = m\_hWnd;  tme.dwHoverTime = HOVER\_DEFAULT;  ::TrackMouseEvent (&tme);  return 0;  } |

MFC doesn't provide type-specific message-mapping macros for WM\_MOUSELEAVE and WM\_MOUSEHOVER messages, so as this example demonstrates, you must use the ON\_MESSAGE macro to link these messages to class member functions. The *lParam* value accompanying a WM\_MOUSEHOVER message holds the cursor's *x* coordinate in its low word and the cursor's *y* coordinate in its high word. *wParam* is unused. Both *wParam* and *lParam* are unused in WM\_MOUSELEAVE messages.

One final note regarding *::TrackMouseEvent*: In order to use it, you must include the following #define in your source code:

|  |
| --- |
| #define \_WIN32*\_*WINNT 0x0400 |

Be sure to include this line before the line that #includes Afxwin.h. Otherwise, it will have no effect.

## The Mouse Wheel

Many of the mice used with Windows today include a wheel that can be used to scroll a window without clicking the scroll bar. When the wheel is rolled, the window with the input focus receives WM\_MOUSEWHEEL messages. MFC's *CScrollView* class provides a default handler for these messages that automatically scrolls the window, but if you want mouse wheel messages to scroll a non-*CScrollView* window, you must process WM\_MOUSEWHEEL messages yourself.

MFC's ON\_WM\_MOUSEWHEEL macro maps WM\_MOUSEWHEEL messages to the message handler *OnMouseWheel*. *OnMouseWheel* is prototyped like this:

|  |
| --- |
| BOOL OnMouseWheel (UINT nFlags, short zDelta, CPoint point) |

The *nFlags* and *point* parameters are identical to those passed to *OnLButtonDown*. *zDelta* is the distance the wheel was rotated. A *zDelta* equal to WHEEL\_DELTA (120) means the wheel was rotated forward one increment, or *notch*, and \_WHEEL\_DELTA means the wheel was rotated backward one notch. If the wheel is rotated forward five notches, the window will receive five WM\_MOUSEWHEEL messages, each with a *zDelta* of WHEEL\_DELTA. *OnMouseWheel* should return a nonzero value if it scrolled the window, or zero if it did not.

A simple way to respond to a WM\_MOUSEWHEEL message is to scroll the window one line up (if *zDelta* is positive) or one line down (if *zDelta* is negative) for every WHEEL\_DELTA unit. The recommended approach, however, is slightly more involved. First you ask the system for the number of lines that corresponds to WHEEL\_DELTA units. In Windows NT 4.0 and higher and in Windows 98, you can get this value by calling *::SystemParametersInfo* with a first parameter equal to SPI\_GETWHEELSCROLLLINES. Then you multiply the result by *zDelta* and divide by WHEEL\_DELTA to determine how many lines to scroll. You can modify the Accel program presented in [Chapter 2](mk:@MSITStore:C:\Program%20Files%20(x86)\MSPress\BooksOnline\Programming%20Windows%20with%20MFC%20Second%20Edition\progmfc2.chm::/ch02a.htm) to respond to WM\_MOUSEWHEEL messages in this manner by adding the following message-map entry and message handler to *CMainWindow*:

|  |
| --- |
| // In the message map  ON\_WM\_MOUSEWHEEL ()    BOOL CMainWindow::OnMouseWheel (UINT nFlags, short zDelta, CPoint point)  {  BOOL bUp = TRUE;  int nDelta = zDelta;  if (zDelta < 0) {  bUp = FALSE;  nDelta = -nDelta;  }  UINT nWheelScrollLines;  ::SystemParametersInfo (SPI\_GETWHEELSCROLLLINES, 0,  &nWheelScrollLines, 0);  if (nWheelScrollLines == WHEEL\_PAGESCROLL) {  SendMessage (WM\_VSCROLL,  MAKEWPARAM (bUp ? SB\_PAGEUP : SB\_PAGEDOWN, 0), 0);  }  else {  int nLines = (nDelta \* nWheelScrollLines) / WHEEL\_DELTA;  while (nLines--)  SendMessage (WM\_VSCROLL,  MAKEWPARAM (bUp ? SB\_LINEUP : SB\_LINEDOWN, 0), 0);  }  return TRUE;  } |

Dividing *zDelta* by WHEEL\_DELTA ensures that the application won't scroll too quickly if, in the future, it's used with a mouse that has a wheel granularity less than 120 units. WHEEL\_PAGESCROLL is a special value that indicates the application should simulate a click of the scroll bar shaft—in other words, perform a page-up or page-down. Both WHEEL\_DELTA and WHEEL\_PAGESCROLL are defined in Winuser.h.

One issue to be aware of regarding this code sample is that it's not compatible with Windows 95. Why? Because calling *::SystemParametersInfo* with an SPI\_GETWHEELSCROLLLINES value does nothing in Windows 95. If you want to support Windows 95, you can either assume that *::SystemParametersInfo* would return 3 (the default) or resort to more elaborate means to obtain the user's preference. MFC uses an internal function named *\_AfxGetMouseScrollLines* to get this value. *\_AfxGetMouseScrollLines* is platform-neutral; it uses various methods to attempt to obtain a scroll line count and defaults to 3 if none of those methods work. See the MFC source code file Viewscrl.cpp if you'd like to mimic that behavior in your code.

If the mouse wheel is clicked rather than rotated, the window under the cursor generally receives middle-button mouse messages—WM\_MBUTTONDOWN messages when the wheel is pressed, WM\_MBUTTONUP messages when the wheel is released. (I say "generally" because this is the default behavior; it can be changed through the Control Panel.) Some applications respond to wheel clicks in a special way. Microsoft Word 97, for example, scrolls the currently displayed document when it receives WM\_MOUSEMOVE messages with the wheel held down. Knowing that the mouse wheel produces middle-button messages, you can customize your applications to respond to mouse wheel events any way you see fit.

## Capturing the Mouse

One problem that frequently crops up in programs that process mouse messages is that the receipt of a button-down message doesn't necessarily mean that a button-up message will follow. Suppose you've written a drawing program that saves the *point* parameter passed to *OnLButtonDown* and uses it as an anchor point to draw a line whose other endpoint follows the cursor—an action known as "rubber-banding" a line. When a WM\_LBUTTONUP message arrives, the application erases the rubber-band line and draws a real line in its place. But what happens if the user moves the mouse outside the window's client area before releasing the mouse button? The application never gets that WM\_LBUTTONUP message, so the rubber-band line is left hanging in limbo and the real line isn't drawn.

Windows provides an elegant solution to this problem by allowing an application to "capture" the mouse upon receiving a button-down message and to continue receiving mouse messages no matter where the cursor goes on the screen until the button is released or the capture is canceled. (In the Win32 environment, to prevent applications from monopolizing the mouse, the system stops sending mouse messages to a window that owns the capture if the button is released.) The mouse is captured with *CWnd::SetCapture* and released with *::ReleaseCapture*. Calls to these functions are normally paired in button-down and button-up handlers, as shown here:

|  |
| --- |
| // In CMainWindow's message map  ON\_WM\_LBUTTONDOWN ()  ON\_WM\_LBUTTONUP ()    void CMainWindow::OnLButtonDown (UINT nFlags, CPoint point)  {  SetCapture ();  }  void CMainWindow::OnLButtonUp (UINT nFlags, CPoint point)  {  ::ReleaseCapture ();  } |

In between, *CMainWindow* receives WM\_MOUSEMOVE messages that report the cursor position even if the cursor leaves it. Client-area mouse messages continue to report cursor positions in client coordinates, but coordinates can now go negative and can also exceed the dimensions of the window's client area.

A related function, *CWnd::GetCapture*, returns a *CWnd* pointer to the window that owns the capture. In the Win32 environment, *GetCapture* returns NULL if the mouse is not captured or if it's captured by a window belonging to another thread. The most common use of *GetCapture* is for determining whether your own window has captured the mouse. The statement

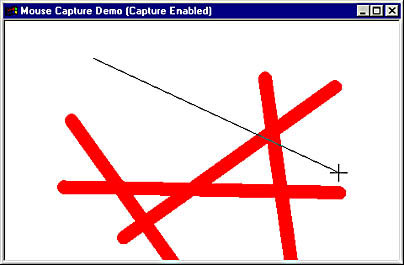
|  |
| --- |
| if (GetCapture () == this) |

is true if and only if the window identified by *this* currently has the mouse captured.

How does capturing the mouse solve the problem with the rubber-banded line? By capturing the mouse in response to a WM\_LBUTTONDOWN message and releasing it when a WM\_LBUTTONUP message arrives, you're guaranteed to get the WM\_LBUTTONUP message when the mouse button is released. The sample program in the next section illustrates the practical effect of this technique.

## Mouse Capturing in Action

The MouseCap application shown in Figure 3-4 is a rudimentary paint program that lets the user draw lines with the mouse. To draw a line, press the left mouse button anywhere in the window's client area and drag the cursor with the button held down. As the mouse is moved, a thin line is rubber-banded between the anchor point and the cursor. When the mouse button is released, the rubber-band line is erased and a red line 16 pixels wide is drawn in its place. Because the mouse is captured while the button is depressed, rubber-banding works even if the mouse is moved outside the window. And no matter where the cursor is when the mouse button is released, a red line is drawn between the anchor point and the endpoint. MouseCap's source code appears in Figure 3-5.



**Figure 3-4.** *The MouseCap window.*

**Figure 3-5.** *The MouseCap application.*

|  |
| --- |
| MouseCap.h class CMyApp : public CWinApp  {  public:  virtual BOOL InitInstance ();  };  class CMainWindow : public CFrameWnd  {  protected:  BOOL m\_bTracking; // TRUE if rubber banding  BOOL m\_bCaptureEnabled; // TRUE if capture enabled  CPoint m\_ptFrom; // "From" point for rubber banding  CPoint m\_ptTo; // "To" point for rubber banding  void InvertLine (CDC\* pDC, CPoint ptFrom, CPoint ptTo);  public:  CMainWindow ();  protected:  afx\_msg void OnLButtonDown (UINT nFlags, CPoint point);  afx\_msg void OnLButtonUp (UINT nFlags, CPoint point);  afx\_msg void OnMouseMove (UINT nFlags, CPoint point);  afx\_msg void OnNcLButtonDown (UINT nHitTest, CPoint point);  DECLARE\_MESSAGE\_MAP ()  }; |

|  |
| --- |
| MouseCap.cpp #include <afxwin.h>  #include "MouseCap.h"  CMyApp myApp;  /////////////////////////////////////////////////////////////////////////  // CMyApp member functions  BOOL CMyApp::InitInstance ()  {  m\_pMainWnd = new CMainWindow;  m\_pMainWnd->ShowWindow (m\_nCmdShow);  m\_pMainWnd->UpdateWindow ();  return TRUE;  }  /////////////////////////////////////////////////////////////////////////  // CMainWindow message map and member functions  BEGIN\_MESSAGE\_MAP (CMainWindow, CFrameWnd)  ON\_WM\_LBUTTONDOWN ()  ON\_WM\_LBUTTONUP ()  ON\_WM\_MOUSEMOVE ()  ON\_WM\_NCLBUTTONDOWN ()  END\_MESSAGE\_MAP ()  CMainWindow::CMainWindow ()  {  m\_bTracking = FALSE;  m\_bCaptureEnabled = TRUE;  //  // Register a WNDCLASS.  //  CString strWndClass = AfxRegisterWndClass (  0,  AfxGetApp ()->LoadStandardCursor (IDC\_CROSS),  (HBRUSH) (COLOR\_WINDOW + 1),  AfxGetApp ()->LoadStandardIcon (IDI\_WINLOGO)  );  //  // Create a window.  //  Create (strWndClass, \_T ("Mouse Capture Demo (Capture Enabled)"));  }  void CMainWindow::OnLButtonDown (UINT nFlags, CPoint point)  {  //  // Record the anchor point and set the tracking flag.  //  m\_ptFrom = point;  m\_ptTo = point;  m\_bTracking = TRUE;  //  // If capture is enabled, capture the mouse.  //  if (m\_bCaptureEnabled)  SetCapture ();  }  void CMainWindow::OnMouseMove (UINT nFlags, CPoint point)  {  //  // If the mouse is moved while we're "tracking" (that is, while a  // line is being rubber-banded), erase the old rubber-band line and  // draw a new one.  //  if (m\_bTracking) {  CClientDC dc (this);  InvertLine (&dc, m\_ptFrom, m\_ptTo);  InvertLine (&dc, m\_ptFrom, point);  m\_ptTo = point;  }  }  void CMainWindow::OnLButtonUp (UINT nFlags, CPoint point)  {  //  // If the left mouse button is released while we're tracking, release  // the mouse if it's currently captured, erase the last rubber-band  // line and draw a thick red line in its place.  //  if (m\_bTracking) {  m\_bTracking = FALSE;  if (GetCapture () == this)  ::ReleaseCapture ();  CClientDC dc (this);  InvertLine (&dc, m\_ptFrom, m\_ptTo);  CPen pen (PS\_SOLID, 16, RGB (255, 0, 0));  dc.SelectObject (&pen);  dc.MoveTo (m\_ptFrom);  dc.LineTo (point);  }  }  void CMainWindow::OnNcLButtonDown (UINT nHitTest, CPoint point)  {  //  // When the window's title bar is clicked with the left mouse button,  // toggle the capture flag on or off and update the window title.  //  if (nHitTest == HTCAPTION) {  m\_bCaptureEnabled = m\_bCaptureEnabled ? FALSE : TRUE;  SetWindowText (m\_bCaptureEnabled ?  \_T ("Mouse Capture Demo (Capture Enabled)") :  \_T ("Mouse Capture Demo (Capture Disabled)"));  }  CFrameWnd::OnNcLButtonDown (nHitTest, point);  }  void CMainWindow::InvertLine (CDC\* pDC, CPoint ptFrom, CPoint ptTo)  {  //  //Invert a line of pixels by drawing a line in the R2\_NOT drawing mode.  //  int nOldMode = pDC->SetROP2 (R2\_NOT);  pDC->MoveTo (ptFrom);  pDC->LineTo (ptTo);  pDC->SetROP2 (nOldMode);  } |

Most of the action takes place in the program's *OnLButtonDown*, *OnMouseMove*, and *OnLButtonUp* handlers. *OnLButtonDown* starts the drawing process by initializing a trio of variables that are members of the *CMainWindow* class:

|  |
| --- |
| m\_ptFrom = point;  m\_ptTo = point;  m\_bTracking = TRUE; |

*m\_ptFrom* and *m\_ptTo* are the starting and ending points for the rubber-band line. *m\_ptTo* is continually updated by the *OnMouseMove* handler as the mouse is moved. *m\_bTracking*, which is TRUE when the left button is down and FALSE when it is not, is a flag that tells *OnMouseMove* and *OnLButtonUp* whether a line is being rubber-banded. *OnLButtonDown*'s only other action is to capture the mouse if *m\_bCaptureEnabled* is TRUE:

|  |
| --- |
| if (m\_bCaptureEnabled)  SetCapture (); |

*m\_bCaptureEnabled* is initialized to TRUE by *CMainWindow*'s constructor. It is toggled by the window's *OnNcLButtonDown* handler so that you can turn mouse capturing on and off and see the effect that mouse capturing has on the program's operation. (More on this in a moment.)

*OnMouseMove*'s job is to move the rubber-band line and update *m\_ptTo* with the new cursor position whenever the mouse is moved. The statement

|  |
| --- |
| InvertLine (&dc, m\_ptFrom, m\_ptTo); |

erases the previously drawn rubber-band line, and

|  |
| --- |
| InvertLine (&dc, m\_ptFrom, point); |

draws a new one. *InvertLine* is a member of *CMainWindow*. It draws a line not by setting each pixel to a certain color, but by inverting the existing pixel colors. This ensures that the line can be seen no matter what background it is drawn against and that drawing the line again in the same location will erase it by restoring the original screen colors. The inversion is accomplished by setting the device context's drawing mode to R2\_NOT with the statement

|  |
| --- |
| int nOldMode = pDC->SetROP2 (R2\_NOT); |

See [Chapter 2](mk:@MSITStore:C:\Program%20Files%20(x86)\MSPress\BooksOnline\Programming%20Windows%20with%20MFC%20Second%20Edition\progmfc2.chm::/ch02a.htm) for a discussion of R2\_NOT and other drawing modes.

When the left mouse button is released, *CMainWindow::OnLButtonUp* is called. After setting *m\_bTracking* to FALSE and releasing the mouse, it erases the rubber-band line drawn by the last call to *OnMouseMove*:

|  |
| --- |
| CClientDC dc (this);  InvertLine (&dc, m\_ptFrom, m\_ptTo); |

*OnLButtonUp* then creates a solid red pen 16 pixels wide, selects it into the device context, and draws a thick red line:

|  |
| --- |
| CPen pen (PS\_SOLID, 16, RGB (255, 0, 0));  dc.SelectObject (&pen);  dc.MoveTo (m\_ptFrom);  dc.LineTo (point); |

Its work done, *OnLButtonUp* returns, and the drawing operation is complete. Figure 3-4 above shows what the MouseCap window looks like after a few lines have been drawn and as a new line is rubber-banded.

After you've played around with the program a bit, click the title bar to activate the *OnNcLButtonDown* handler and toggle the *m\_bCaptureEnabled* flag from TRUE to FALSE. The window title should change from "Mouse Capture Demo (Capture Enabled)" to "Mouse Capture Demo (Capture Disabled)." *OnNcLButtonDown* processes left button clicks in the nonclient area and uses *CWnd::SetWindowText* to change the window title if the hit-test code in *nHitTest* is equal to HTCAPTION, indicating that the click occurred in the title bar.

Now draw a few lines with mouse capturing disabled. Observe that if you move the mouse outside the window while rubber-banding, the line freezes until the mouse reenters the client area, and that if you release the mouse button outside the window, the program gets out of sync. The rubber-band line follows the mouse when you move it back to the interior of the window (even though the mouse button is no longer pressed), and it never gets erased. Click the title bar once again to reenable mouse capturing, and the program will revert to its normal self.

## The Cursor

Rather than use the arrow-shaped cursor you see in most Windows applications, MouseCap uses a crosshair cursor. Arrows and crosshairs are just two of several predefined cursor types that Windows places at your disposal, and if none of the predefined cursors fits the bill, you can always create your own. As usual, Windows gives programmers a great deal of latitude in this area.

First, a bit of background on how cursors work. As you know, every window has a corresponding WNDCLASS whose characteristics are defined in a WNDCLASS structure. One of the fields of the WNDCLASS structure is *hCursor*, which holds the handle of the class cursor—the image displayed when the cursor is over a window's client area. When the mouse is moved, Windows erases the cursor from its old location by redrawing the background behind it. Then it sends the window under the cursor a WM\_SETCURSOR message containing a hit-test code. The system's default response to this message is to call *::SetCursor* to display the class cursor if the hit-test code is HTCLIENT or to display an arrow if the hit-test code indicates that the cursor is outside the client area. As a result, the cursor is automatically updated as it is moved about the screen. When you move the cursor into an edit control, for example, it changes into a vertical bar or "I-beam" cursor. This happens because Windows registers a special WNDCLASS for edit controls and specifies the I-beam cursor as the class cursor.

It follows that one way to change the cursor's appearance is to register a WNDCLASS and specify the desired cursor type as the class cursor. In MouseCap, *CMainWindow*'s constructor registers a WNDCLASS whose class cursor is IDC\_CROSS and passes the WNDCLASS name to *CFrameWnd::Create*:

|  |
| --- |
| CString strWndClass = AfxRegisterWndClass (  0,  AfxGetApp ()->LoadStandardCursor (IDC\_CROSS),  (HBRUSH) (COLOR\_WINDOW + 1),  AfxGetApp ()->LoadStandardIcon (IDI\_WINLOGO)  );  Create (strWndClass, \_T ("Mouse Capture Demo (Capture Enabled)")); |

Windows then displays a crosshair cursor anytime the mouse pointer is positioned in *CMainWindow*'s client area.

A second way to customize the cursor is to call the API function *::SetCursor* in response to WM\_SETCURSOR messages. The following *OnSetCursor* function displays the cursor whose handle is stored in *CMainWindow::m\_hCursor* when the cursor is over *CMainWindow*'s client area:

|  |
| --- |
| // In CMainWindow's message map  ON\_WM\_SETCURSOR ()    BOOL CMainWindow::OnSetCursor (CWnd\* pWnd, UINT nHitTest,  UINT message)  {  if (nHitTest == HTCLIENT) {  ::SetCursor (m\_hCursor);  return TRUE;  }  return CFrameWnd::OnSetCursor (pWnd, nHitTest, message);  } |

Returning TRUE after calling *::SetCursor* tells Windows that the cursor has been set. WM\_SETCURSOR messages generated outside the window's client area are passed to the base class so that the default cursor is displayed. The class cursor is ignored because *OnSetCursor* never gives Windows the opportunity to display it.

Why would you want to use *OnSetCursor* rather than just registering *m\_hCursor* as the class cursor? Suppose you want to display an arrow cursor when the cursor is in the top half of the window and an I-beam cursor when the cursor is in the bottom half. A class cursor won't suffice in this case, but *OnSetCursor* will do the job quite nicely. The following *OnSetCursor* handler sets the cursor to either *m\_hCursorArrow* or *m\_hCursorIBeam* when the cursor is in *CMainWindow*'s client area:

|  |
| --- |
| BOOL CMainWindow::OnSetCursor (CWnd\* pWnd, UINT nHitTest,  UINT message)  {  if (nHitTest == HTCLIENT) {  DWORD dwPos = ::GetMessagePos ();  CPoint point (LOWORD (dwPos), HIWORD (dwPos));  ScreenToClient (&point);  CRect rect;  GetClientRect (&rect);  ::SetCursor ((point.y < rect.Height () / 2) ?  m\_hCursorArrow : m\_hCursorIBeam);  return TRUE;  }  return CFrameWnd::OnSetCursor (pWnd, nHitTest, message);  } |

*::GetMessagePos* returns a DWORD value containing the cursor's *x* and *y* screen coordinates at the moment the WM\_SETCURSOR message was retrieved from the message queue. *CWnd::ScreenToClient* converts screen coordinates to client coordinates. If the converted point's *y* coordinate is less than half the height of the window's client area, the cursor is set to *m\_hCursorArrow*. But if *y* is greater than or equal to half the client area height, the cursor is set to *m\_hCursorIBeam* instead. The VisualKB application presented later in this chapter uses a similar technique to change the cursor to an I-beam when it enters a rectangle surrounding a text-entry field.

Should the need ever arise, you can hide the cursor with the statement

|  |
| --- |
| ::ShowCursor (FALSE); |

and display it again with

|  |
| --- |
| ::ShowCursor (TRUE); |

Internally, Windows maintains a display count that's incremented each time *::ShowCursor (TRUE)* is called and decremented by each call to *::ShowCursor (FALSE)*. The count is initially set to 0 if a mouse is installed and to -1 if no mouse is present, and the cursor is displayed whenever the count is greater than or equal to 0. Thus, if you call *::ShowCursor (FALSE)* twice to hide the cursor, you must call *::ShowCursor (TRUE)* twice to display it again.

## The Hourglass Cursor

When an application responds to a message by undertaking a lengthy processing task, it's customary to change the cursor to an hourglass to remind the user that the application is "busy." (While a message handler executes, no further messages are retrieved from the message queue and the program is frozen to input. In [Chapter 17](mk:@MSITStore:C:\Program%20Files%20(x86)\MSPress\BooksOnline\Programming%20Windows%20with%20MFC%20Second%20Edition\progmfc2.chm::/ch17a.htm), you'll learn about ways to perform background processing tasks while continuing to retrieve and dispatch messages.)

Windows provides the hourglass cursor for you; its identifier is IDC\_WAIT. An easy way to display an hourglass cursor is to declare a *CWaitCursor* variable on the stack, like this:

|  |
| --- |
| CWaitCursor wait; |

*CWaitCursor*'s constructor displays an hourglass cursor, and its destructor restores the original cursor. If you'd like to restore the cursor before the variable goes out of scope, simply call *CWaitCursor::Restore*:

|  |
| --- |
| wait.Restore (); |

You should call *Restore* before taking any action that would allow a WM\_SETCURSOR message to seep through and destroy the hourglass—for example, before displaying a message box or a dialog box.

You can change the cursor displayed by *CWaitCursor::CWaitCursor* and *BeginWaitCursor* by overriding *CWinApp*'s virtual *DoWaitCursor* function. Use the default implementation of *CWinApp::DoWaitCursor* found in the MFC source code file Appui.cpp as a model for your own implementations.

## Mouse Miscellanea

As mentioned earlier, calling the *::GetSystemMetrics* API function with an SM\_CMOUSEBUTTONS argument queries the system for the number of mouse buttons. (There is no MFC equivalent to *::GetSystemMetrics*, so you must call it directly.) The usual return value is 1, 2, or 3, but a 0 return means no mouse is attached. You can also find out whether a mouse is present by calling *::GetSystemMetrics* this way:

|  |
| --- |
| ::GetSystemMetrics (SM\_MOUSEPRESENT) |

The return value is nonzero if there is a mouse attached, 0 if there is not. In the early days of Windows, programmers had to consider the possibility that someone might be using Windows without a mouse. Today that's rarely a concern, and a program that queries the system to determine whether a mouse is present is a rare program indeed.

Other mouse-related *::GetSystemMetrics* parameters include SM\_CXDOUBLECLK and SM\_CYDOUBLECLK, which specify the maximum horizontal and vertical distances (in pixels) that can separate the two halves of a double click, and SM\_SWAPBUTTON, which returns a nonzero value if the user has swapped the left and right mouse buttons using the Control Panel. When the mouse buttons are swapped, the left mouse button generates WM\_RBUTTON messages and the right mouse button generates WM\_LBUTTON messages. Generally you don't need to be concerned about this, but if for some reason your application wants to be sure that the left mouse button *really* means the left mouse button, it can use *::GetSystemMetrics* to determine whether the buttons have been swapped.

The API functions *::SetDoubleClickTime* and *::GetDoubleClickTime* enable an application to set and retrieve the mouse double-click time—the maximum amount of time permitted between clicks when a mouse button is double-clicked. The expression

|  |
| --- |
| ::GetDoubleClickTime () |

returns the double-click time in milliseconds, while the statement

|  |
| --- |
| ::SetDoubleClickTime (250); |

sets the double-click time to 250 milliseconds, or one quarter of a second. When the same mouse button is clicked twice in succession, Windows uses both the double-click time and the SM\_CXDOUBLECLK and SM\_CYDOUBLECLK values returned by *::GetSystemMetrics* to determine whether to report the second of the two clicks as a double click.

A function that processes mouse messages can determine which, if any, mouse buttons are pressed by checking the *nFlags* parameter passed to the message handler. It's also possible to query the state of a mouse button outside a mouse message handler by calling *::GetKeyState* or *::GetAsyncKeyState* with a VK\_LBUTTON, VK\_MBUTTON, or VK\_RBUTTON parameter. *::GetKeyState* should be called only from a keyboard message handler because it returns the state of the specified mouse button at the time the keyboard message was generated. *::GetAsyncKeyState* can be called anywhere, anytime. It works in real time, returning the state of the button at the moment the function is called. A negative return value from

|  |
| --- |
| ::GetKeyState (VK\_LBUTTON) |

or

|  |
| --- |
| ::GetAsyncKeyState (VK\_LBUTTON) |

indicates that the left mouse button is pressed. Swapping the mouse buttons does not affect *::GetAsyncKeyState*, so if you use this function, you should also use *::GetSystemMetrics* to determine whether the buttons have been swapped. The expression

|  |
| --- |
| ::GetAsyncKeyState (::GetSystemMetrics (SM\_SWAPBUTTON) ?  VK\_RBUTTON : VK\_LBUTTON) |

checks the state of the left mouse button asynchronously and automatically queries the right mouse button instead if the buttons have been swapped.

Windows provides a pair of API functions named *::GetCursorPos* and *::SetCursorPos* for getting and setting the cursor position manually. *::GetCursorPos* copies the cursor coordinates to a POINT structure. A related function named *::GetMessagePos* returns a DWORD value containing a pair of 16-bit coordinates specifying where the cursor was when the last message was retrieved from the message queue. You can extract those coordinates using the Windows LOWORD and HIWORD macros:

|  |
| --- |
| DWORD dwPos = ::GetMessagePos ();  int x = LOWORD (dwPos);  int y = HIWORD (dwPos); |

*::GetCursorPos* and *::GetMessagePos* both report the cursor position in screen coordinates. Screen coordinates can be converted to client coordinates by calling a window's *ClientToScreen* function.

Windows also provides a function named *::ClipCursor* that restricts the cursor to a particular area of the screen. *::ClipCursor* accepts a pointer to a RECT structure that describes, in screen coordinates, the clipping rectangle. Since the cursor is a global resource shared by all applications, an application that uses *::ClipCursor* must free the cursor by calling

|  |
| --- |
| ::ClipCursor (NULL); |

before terminating, or else the cursor will remain locked into the clipping rectangle indefinitely.