

Module 4: The Graphics Device Interface (GDI), Colors, and Fonts

Program examples compiled using Visual C++ 6.0 (MFC 6.0) compiler on Windows XP Pro machine with Service Pack 2. Topics and sub topics for this Tutorial are listed below:

The Graphics Device Interface (GDI), Colors, and Fonts

The Device Context Classes

The Display Context Classes `CClientDC` and `CWindowDC`

Constructing and Destroying `CDC` Objects

The State of the Device Context

The `CPaintDC` Class

For Win32 Programmers

GDI Objects

Constructing and Destroying GDI Objects

Tracking GDI Objects

Stock GDI Objects

The Lifetime of a GDI Selection

Windows Color Mapping

Standard Video Graphics Array Video Cards

256-Color Video Cards

16-Bit-Color Video Cards

24-Bit-Color Video Cards

Fonts

Fonts Are GDI Objects

Choosing a Font

Printing with Fonts

Displaying Fonts

Logical Inches and Physical Inches on the Display

Computing Character Height

The MYMFC4 Example

Setting the Mapping Mode in the `OnPrepareDC()` Function

The `ShowFont()` Private Member Function

The MYMFC5 Example

The MYMFC5 Program Elements

The `OnDraw()` Member Function

The `TraceMetrics()` Helper Function

The MYMFC6 Example: `CScrollView` Revisited

The MYMFC6 Program Elements

The `m_sizeEllipse` and `m_pointTopLeft` Data Members

The `m_sizeOffset` Data Member

The `m_bCaptured` Data Member

The `SetCapture()` and `ReleaseCapture()` Functions

The `SetCursor()` and `LoadCursor()` [Win32 Functions](#)

The `CScrollView::OnPrepareDC` Member Function

The `OnMouseMove()` Coordinate Transformation Code

The `OnDraw()` Function

The `CScrollView` `SetScaleToFitSize()` Mode

Using the Logical Twips Mapping Mode in a Scrolling View

The Graphics Device Interface (GDI), Colors, and Fonts

You've already seen some elements of the **Graphics Device Interface** (GDI). Anytime your program draws to the display or the printer, it must use the GDI functions. The GDI provides functions for drawing points, lines, rectangles, polygons, ellipses, bitmaps, and text. You can draw circles and squares intuitively once you study the available functions, but text programming is more difficult. This module gives you the information you need to start using the GDI effectively in the Microsoft Visual C++ environment. You'll learn how to use fonts on both the display and the printer.

The Device Context Classes

In the previous Modules, the view class's `OnDraw()` member function was passed a pointer to a device context object. `OnDraw()` selected a brush and then drew an ellipse. The Microsoft Windows device context is the key GDI element that represents a physical device. Each C++ device context object has an associated Windows device context, identified by a 32-bit handle of type `HDC`.

MFC Library version 6.0 provides a number of device context classes. The base class `CDC` has all the member functions (including some virtual functions) that you'll need for drawing. Except for the oddball `CMetaFileDC` class, derived classes are distinct only in their constructors and destructors. If you (or the application framework) construct an object of a derived device context class, you can pass a `CDC` pointer to a function such as `OnDraw()`. For the display, the usual derived classes are `CClientDC` and `CWindowDC`. For other devices, such as printers or memory buffers, you construct objects of the base class `CDC`. The "virtualness" of the `CDC` class is an important feature of the application framework. In [Module 13](#), you'll see how easy it is to write code that works with both the printer and the display. A statement in `OnDraw()` such as:

```
// Print "Hello" at (0, 0) location
pDC->TextOut(0, 0, "Hello");
```

Sends text to the display, the printer, or the Print Preview window, depending on the class of the object referenced by the `CView::OnDraw()` function's `pDC` parameter. For display and printer device context objects, the application framework attaches the handle to the object. For other device contexts, such as the memory device context, you must call a member function after construction in order to attach the handle.

The Display Context Classes `CClientDC` and `CWindowDC`

Recall that a window's client area excludes the border, the caption bar, and the menu bar. If you create a `CClientDC` object, you have a device context that is mapped only to this client area - you can't draw outside it. The point (0, 0) usually refers to the upper-left corner of the client area. As you'll see later, an MFC `CView` object corresponds to a child window that is contained inside a separate frame window, often along with a toolbar, a status bar, and scroll bars. The client area of the view, then, does not include these other windows. If the window contains a docked toolbar along the top, for example, (0, 0) refers to the point immediately under the left edge of the toolbar.

If you construct an object of class `CWindowDC`, the point (0, 0) is at the upper-left corner of the non-client area of the window. With this whole-window device context, you can draw in the window's border, in the caption area, and so forth. Don't forget that the view window doesn't have a non-client area, so `CWindowDC` is more applicable to frame windows than it is to view windows.

Constructing and Destroying `CDC` Objects

After you construct a `CDC` object, it is important to destroy it promptly when you're done with it. Microsoft Windows limits the number of available device contexts, and if you fail to release a Windows device context object, a small amount of memory is lost until your program exits. Most frequently, you'll construct a device context object inside a message handler function such as `OnLButtonDown()`. The easiest way to ensure that the device context object is destroyed (and that the underlying Windows device context is released) is to construct the object on the stack in the following way:

```
void CMyView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CRect rect;
    // constructs dc on the stack
    CClientDC dc(this);
    // retrieves the clipping rectangle
```

```

        dc.GetClipBox(rect);
    } // dc automatically released

```

Notice that the `CClientDC` constructor takes a window pointer as a parameter. The destructor for the `CClientDC` object is called when the function returns. You can also get a device context pointer by using the `CWnd::GetDC` member function, as shown in the following code. You must be careful here to call the `ReleaseDC()` function to release the device context.

```

void CMyView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CRect rect;

    CDC* pDC = GetDC(); // a pointer to an internal dc
    pDC->GetClipBox(rect); // retrieves the clipping rectangle
    ReleaseDC(pDC); // Don't forget this
}

```

You must not destroy the `CDC` object passed by the pointer to `OnDraw()`. The application framework handles the destruction for you.

The State of the Device Context

You already know that a device context is required for drawing. When you use a `CDC` object to draw an ellipse, for example, what you see on the screen (or on the printer's hard copy) depends on the current "state" of the device context. This state includes the following:

- Attached GDI drawing objects such as pens, brushes, and fonts.
- The mapping mode that determines the scale of items when they are drawn. You've already experimented with the mapping mode in the previous Module.
- Various details such as text alignment parameters and polygon filling mode.

You have already seen, for example, that choosing a gray brush prior to drawing an ellipse results in the ellipse having a gray interior. When you create a device context object, it has certain default characteristics, such as a black pen for shape boundaries. All other state characteristics are assigned through `CDC` class member functions. GDI objects are selected into the device context by means of the overloaded `SelectObject()` functions. A device context can, for example, have one pen, one brush, or one font selected at any given time.

The `CPaintDC` Class

You'll need the `CPaintDC` class only if you override your view's `OnPaint()` function. The default `OnPaint()` calls `OnDraw()` with a properly set up device context, but sometimes you'll need display-specific drawing code. The `CPaintDC` class is special because its constructor and destructor do housekeeping unique to drawing to the display. Once you have a `CDC` pointer, however, you can use it as you would any other device context pointer. Here's a sample `OnPaint()` function that creates a `CPaintDC` object:

```

void CMyView::OnPaint()
{
    CPaintDC dc(this);
    OnPrepareDC(&dc); // explained later
    dc.TextOut(0, 0, "for the display, not the printer");
    OnDraw(&dc); // stuff that's common to display and printer
}

```

For Win32 Programmers

The `CPaintDC` constructor calls `BeginPaint()` for you, and the destructor calls `EndPaint()`. If you construct your device context on the stack, the `EndPaint()` call is completely automatic.

GDI Objects

A Windows GDI object type is represented by an MFC library class. `CGdiObject` is the abstract base class for the GDI object classes. A Windows GDI object is represented by a C++ object of a class derived from `CGdiObject`. Here's a list of the GDI derived classes:

Derived Class	Description
<code>CBitmap</code>	A bitmap is an array of bits in which one or more bits correspond to each display pixel. You can use bitmaps to represent images, and you can use them to create brushes.
<code>CBrush</code>	A brush defines a bitmapped pattern of pixels that is used to fill areas with color.
<code>CFont</code>	A font is a complete collection of characters of a particular typeface and a particular size. Fonts are generally stored on disk as resources, and some are device-specific.
<code>CPalette</code>	A palette is a color mapping interface that allows an application to take full advantage of the color capability of an output device without interfering with other applications.
<code>CPen</code>	A pen is a tool for drawing lines and shape borders. You can specify a pen's color and thickness and whether it draws solid, dotted, or dashed lines.
<code>CRgn</code>	A region is an area whose shape is a polygon, an ellipse, or a combination of polygons and ellipses. You can use regions for filling, clipping, and mouse hit-testing.

Table 1.

Constructing and Destroying GDI Objects

You never construct an object of class `CGdiObject`; instead, you construct objects of the derived classes. Constructors for some GDI derived classes, such as `CPen` and `CBrush`, allow you to specify enough information to create the object in one step. Others, such as `CFont` and `CRgn`, require a second creation step. For these classes, you construct the C++ object with the default constructor and then you call a create function such as the `CreateFont()` or `CreatePolygonRgn()` function.

The `CGdiObject` class has a virtual destructor. The derived class destructors delete the Windows GDI objects that are attached to the C++ objects. If you construct an object of a class derived from `CGdiObject`, you must delete it prior to exiting the program. To delete a GDI object, you must first separate it from the device context. You'll see an example of this in the next section. Failure to delete a GDI object was a serious offense with Win16. GDI memory was not released until the user restarted Windows. With Win32, however, the GDI memory is owned by the process and is released when your program terminates. Still, an unreleased GDI bitmap object can waste a significant amount of memory.

Tracking GDI Objects

OK, so you know that you have to delete your GDI objects and that they must first be disconnected from their device contexts. How do you disconnect them? A member of the `CDC::SelectObject` family of functions does the work of selecting a GDI object into the device context, and in the process it returns a pointer to the previously selected object (which gets deselected in the process). Trouble is, you can't deselect the old object without selecting a new object. One easy way to track the objects is to "save" the original GDI object when you select your own GDI object and "restore" the original object when you're finished. Then you'll be ready to delete your own GDI object. Here's an example:

```
void CMyView::OnDraw(CDC* pDC)
{
    // black pen, 2 pixels wide
    CPen newPen(PS_DASHDOTDOT, 2, (COLORREF) 0);
    CPen* pOldPen = pDC->SelectObject(&newPen);

    pDC->MoveTo(10, 10);
    pDC->Lineto(110, 10);
    // newPen is deselected
    pDC->SelectObject(pOldPen);
} // newPen automatically destroyed on exit
```

When a device context object is destroyed, all its GDI objects are deselected. Thus, if you know that a device context will be destroyed before its selected GDI objects are destroyed, you don't have to deselect the objects. If, for example, you declare a pen as a view class data member (and you initialize it when you initialize the view), you don't have to

deselect the pen inside `OnDraw()` because the device context, controlled by the view base class's `OnPaint()` handler, will be destroyed first.

Stock GDI Objects

Windows contains a number of stock GDI objects that you can use. Because these objects are part of Windows, you don't have to worry about deleting them. Windows ignores requests to delete stock objects. The MFC library function `CDC::SelectStockObject` selects a stock object into the device context and returns a pointer to the previously selected object, which it deselects. Stock objects are handy when you want to deselect your own non-stock GDI object prior to its destruction. You can use a stock object as an alternative to the "old" object you used in the previous example, as shown here:

```
void CMyView::OnDraw(CDC* pDC)
{
    // black pen, 2 pixels wide
    CPen newPen(PS_DASHDOTDOT, 2, (COLORREF) 0);

    pDC->SelectObject(&newPen);
    pDC->MoveTo(10, 10);
    pDC->Lineto(110, 10);
    // newPen is deselected
    pDC->SelectStockObject(BLACK_PEN);
} // newPen destroyed on exit
```

The [Microsoft Foundation Class Reference](#) lists, under `CDC::SelectStockObject`, the stock objects available for pens, brushes, fonts, and palettes.

The Lifetime of a GDI Selection

For the display device context, you get a "fresh" device context at the beginning of each message handler function. No GDI selections (or mapping modes or other device context settings) persist after your function exits. You must, therefore, set up your device context from scratch each time. The `CView` class virtual member function `OnPrepareDC()` is useful for setting the mapping mode, but you must manage your own GDI objects. For other device contexts, such as those for printers and memory buffers, your assignments can last longer. For these long-life device contexts, things get a little more complicated. The complexity results from the temporary nature of GDI C++ object pointers returned by the `SelectObject()` function. The temporary "object" will be destroyed by the application framework during the idle loop processing of the application, sometime after the handler function returns the call. You can't simply store the pointer in a class data member; instead, you must convert it to a Windows handle (the only permanent GDI identifier) with the `GetSafeHdc()` member function. Here's an example:

```
// m_pPrintFont points to a CFont object created in CMyView's constructor
// m_hOldFont is a CMyView data member of type HFONT, initialized to 0
void CMyView::SwitchToCourier(CDC* pDC)
{
    m_pPrintFont->CreateFont(30, 10, 0, 0, 400, FALSE, FALSE,
                           0, ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                           CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                           DEFAULT_PITCH | FF_MODERN,
                           "Courier New"); // TrueType
    CFont* pOldFont = pDC->SelectObject(m_pPrintFont);

    // m_hOldFont is the CGdiObject public data member that stores
    // the handle
    m_hOldFont = (HFONT) pOldFont->GetSafeHandle();
}

void CMyView::SwitchToOriginalFont(CDC* pDC)
{
    // FromHandle is a static member function that returns an
    // object pointer
    if (m_hOldFont)
    {
```

```

        pDC->SelectObject(CFont::FromHandle(m_hOldFont));
    }
}
// m_pPrintFont is deleted in the CMyView destructor

```

Be careful when you delete an object whose pointer is returned by `SelectObject()`. If you've allocated the object yourself, you can delete it. If the pointer is temporary, as it will be for the object initially selected into the device context, you won't be able to delete the C++ object.

Windows Color Mapping

The Windows GDI provides a hardware-independent color interface. Your program supplies an "absolute" color code, and the GDI maps that code to a suitable color or color combination on your computer's video display. Most programmers of applications for Windows try to optimize their applications' color display for a few common video card categories.

Standard Video Graphics Array Video Cards

A standard Video Graphics Array (VGA) video card uses 18-bit color registers and thus has a palette of 262,144 colors. Because of video memory constraints, however, the standard VGA board accommodates 4-bit color codes, which means it can display only 16 colors at a time. Because Windows needs fixed colors for captions, borders, scroll bars, and so forth; your programs can use only 16 "standard" pure colors. You cannot conveniently access the other colors that the board can display. Each Windows color is represented by a combination of 8-bit "red," "green," and "blue" values. The 16 standard VGA "pure" (non-dithered) colors are shown in the table below. Color-oriented GDI functions accept 32-bit `COLORREF` parameters that contain 8-bit color codes each for red, green, and blue. The **Windows RGB macro** converts 8-bit red, green, and blue values to a `COLORREF` parameter. The following statement, when executed on a system with a standard VGA board, constructs a brush with a dithered color (one that consists of a pattern of pure-color pixels):

```
CBrush brush( RGB(128, 128, 192) );
```

16 standard VGA colors			
Red	Green	Blue	Color
0	0	0	Black
0	0	255	Blue
0	255	0	Green
0	255	255	Cyan
255	0	0	Red
255	0	255	Magenta
255	255	0	Yellow
255	255	255	White
0	0	128	Dark blue
0	128	0	Dark green
0	128	128	Dark cyan
128	0	0	Dark red
128	0	128	Dark magenta
128	128	0	Dark yellow
128	128	128	Dark gray
192	192	192	Light gray

Table 2: The R (Red), G (Green) and B (Blue) color combination..

The following statement (in your view's `OnDraw()` function) sets the text background to red:

```
pDC->SetBkColor( RGB(255, 0, 0) );
```

The CDC functions `SetBkColor()` and `SetTextColor()` don't display dithered colors as the brush-oriented drawing functions do. If the dithered color pattern is too complex, the closest matching pure color is displayed.

256-Color Video Cards

Most video cards can accommodate 8-bit color codes at all resolutions, which means they can display 256 colors simultaneously. This 256-color mode is now considered to be the "lowest common denominator" for color programming.

If Windows is configured for a 256-color display card, your programs are limited to 20 standard pure colors unless you activate the Windows color palette system as supported by the MFC library `Cpalette` class and the Windows API, in which case you can choose your 256 colors from a total of more than 16.7 million. In this Module, we'll assume that the Windows default color mapping is in effect. With an SVGA 256-color display driver installed, you get the 16 VGA colors listed in the previous table plus 4 more, for a total of 20. The following table lists the 4 additional colors.

4 extra SVGA 256-color			
Red	Green	Blue	Color
192	220	192	Money green
166	202	240	Sky blue
255	251	240	Cream
160	160	164	Medium gray

Table 3: Extra RGB combination for SVGA 256 colors.

The RGB macro works much the same as it does with the standard VGA. If you specify one of the 20 standard colors for a brush, you get a pure color; otherwise, you get a dithered color. If you use the `PALETTERGB` macro instead, you don't get dithered colors; you get the closest matching standard pure color as defined by the current palette.

16-Bit-Color Video Cards

Most modern video cards support a resolution of 1024-by-768 pixels, and 1 MB of video memory can support 8-bit color at this resolution. If a video card has 2 MB of memory, it can support 16-bit color, with 5 bits each for red, green, and blue. This means that it can display 32,768 colors simultaneously. That sounds like a lot, but there are only 32 shades each of pure red, green, and blue. Often, a picture will look better in 8-bit-color mode with an appropriate palette selected. A forest scene, for example, can use up to 236 shades of green. Palettes are not supported in 16-bit-color mode.

24-Bit-Color Video Cards

High-end cards (which are becoming more widely used) support 24-bit color. This 24-bit capability enables the display of more than 16.7 million pure colors. If you're using a 24-bit card, you have direct access to all the colors. The RGB macro allows you to specify the exact colors you want. You'll need 2.5 MB of video memory, though, if you want 24-bit color at 1024-by-768-pixel resolution.

Fonts

Old-fashioned character-mode applications could display only the boring system font on the screen. Windows provides multiple device-independent fonts in variable sizes. The effective use of these Windows fonts can significantly energize an application with minimum programming effort. TrueType fonts, first introduced with Windows version 3.1, are even more effective and are easier to program than the previous device-dependent fonts. You'll see several example programs that use various fonts later in this Module.

Fonts Are GDI Objects

Fonts are an integral part of the Windows GDI. This means that fonts behave the same way other GDI objects do. They can be scaled and clipped, and they can be selected into a device context as a pen or a brush can be selected. All GDI rules about de-selection and deletion apply to fonts.

Choosing a Font

Choosing a Windows font used to be like going to a fruit stand and asking for "a piece of reddish-yellow fruit, with a stone inside, that weighs about 4 ounces." You might have gotten a peach or a plum or even a nectarine, and you could be sure that it wouldn't have weighed exactly 4 ounces. Once you took possession of the fruit, you could weigh it and check the fruit type. Now, with TrueType, you can specify the fruit type, but you still can't specify the exact weight. Today you can choose between two font types - device-independent TrueType fonts and device-dependent fonts such as the Windows display System font and the LaserJet LinePrinter font or you can specify a font category and size and let Windows select the font for you. If you let Windows select the font, it will choose a **TrueType** font if possible. The MFC library provides a font selection dialog box tied to the currently selected printer, so there's little need for printer font guesswork. You let the user select the exact font and size for the printer, and then you approximate the display the best you can.

Printing with Fonts

For text-intensive applications, you'll probably want to specify printer font sizes in points (1 point = 1/72 inch). Why? Most, if not all, built-in printer fonts are defined in terms of points. The LaserJet LinePrinter font, for example, comes in one size, 8.5 point. You can specify TrueType fonts in any point size. If you work in points, you need a mapping mode that easily accommodates points. That's what `MM_TWIPS` is for. An 8.5-point font is 8.5×20 , or 170, twips, and that's the character height you'll want to specify.

Displaying Fonts

If you're not worried about the display matching the printed output, you have a lot of flexibility. You can choose any of the scalable Windows TrueType fonts, or you can choose the fixed-size system fonts (stock objects). With the TrueType fonts, it doesn't much matter what mapping mode you use; simply choose a font height and go for it. No need to worry about points.

Matching printer fonts to make printed output match the screen presents some problems, but TrueType makes it easier than it used to be. Even if you're printing with **TrueType** fonts, however, you'll never quite get the display to match the printer output. Characters are ultimately displayed in pixels (or dots), and the width of a string of characters is equal to the sum of the pixel widths of its characters, possibly adjusted for kerning. The pixel width of the characters depends on the font, the mapping mode, and the resolution of the output device. Only if both the printer and the display were set to `MM_TEXT` mode (1 pixel or dot = 1 logical unit) would you get an exact correspondence. If you're using the `CDC::GetTextExtent` function to calculate line breaks, the screen breakpoint will occasionally be different from the printer breakpoint.

In the MFC Print Preview mode, line breaks occur exactly as they do on the printer, but the print quality in the preview window suffers in the process. If you're matching a printer-specific font on the screen, TrueType again makes the job easier. Windows substitutes the closest matching TrueType font. For the 8.5-point **LinePrinter** font, Windows comes pretty close with its `Courier New` font.

Logical Inches and Physical Inches on the Display

The `CDC` member function `GetDeviceCaps()` returns various display measurements that are important to your graphics programming. The six described below provide information about the display size. The values listed are for a typical display card configured for a resolution of 640-by-480 pixels with Microsoft Windows NT 4.0.

Index	Description	Value
<code>HORZSIZE</code>	Physical width in millimeters	320
<code>VERTSIZE</code>	Physical height in millimeters	240
<code>HORZRES</code>	Width in pixels	640
<code>VERTRES</code>	Height in raster lines	480
<code>LOGPIXELSX</code>	Horizontal dots per logical inch	96
<code>LOGPIXELSY</code>	Vertical dots per logical inch	96

Table 4.

The indexes `HORZSIZE` and `VERTSIZE` represent the physical dimensions of your display. These indexes might not be true since Windows doesn't know what size display you have connected to your video adapter. You can also calculate a display size by multiplying `HORZRES` and `VERTRES` by `LOGPIXELSX` and `LOGPIXELSY`, respectively. The size calculated this way is known as the logical size of the display. Using the values above and the fact that there are 25.4 millimeters per inch, we can quickly calculate the two display sizes for a 640-by-480 pixel display under Windows NT 4.0. The physical display size is 12.60-by-9.45 inches, and the logical size is 6.67-by-5.00 inches. So the physical size and the logical size need not be the same.

For Windows NT 4.0, it turns out that `HORZSIZE` and `VERTSIZE` are independent of the display resolution, and `LOGPIXELSX` and `LOGPIXELSY` are always 96. So the logical size changes for different display resolutions, but the physical size does not. For Windows 95, the logical size and the physical size are equal, so both change with the display resolution. At a resolution of 640-by-480 pixels with Windows 95, `HORZSIZE` is 169 and `VERTSIZE` is 127. Whenever you use a fixed mapping mode such as `MM_HIMETRIC` or `MM_TWIPS`, the display driver uses the physical display size to do the mapping. So, for Windows NT, text is smaller on a small monitor; but that's not what you want. Instead, you want your font sizes to correspond to the logical display size, not the physical size. You can invent a special mapping mode, called logical twips, for which one logical unit is equal to 1/1440 logical inch. This mapping mode is independent of the operating system and display resolution and is used by programs such as Microsoft Word. Here is the code that sets the mapping mode to logical twips:

```
pDC->SetMapMode(MM_ANISOTROPIC);  
pDC->SetWindowExt(1440, 1440);  
pDC->SetViewportExt(pDC->GetDeviceCaps(LOGPIXELSX), -pDC-  
>GetDeviceCaps(LOGPIXELSY));
```

From the Windows **Control Panel**, you can adjust both the display font size and the display resolution through the **Appearance** and **Settings** of the **Display Properties** shown below. If you change the display font size from the default 100 percent to 200 percent, `HORZSIZE` becomes 160, `VERTSIZE` becomes 120, and the dots-per-inch value becomes 192. In that case, the logical size is divided by 2, and all text drawn with the logical twips mapping mode is doubled in size.

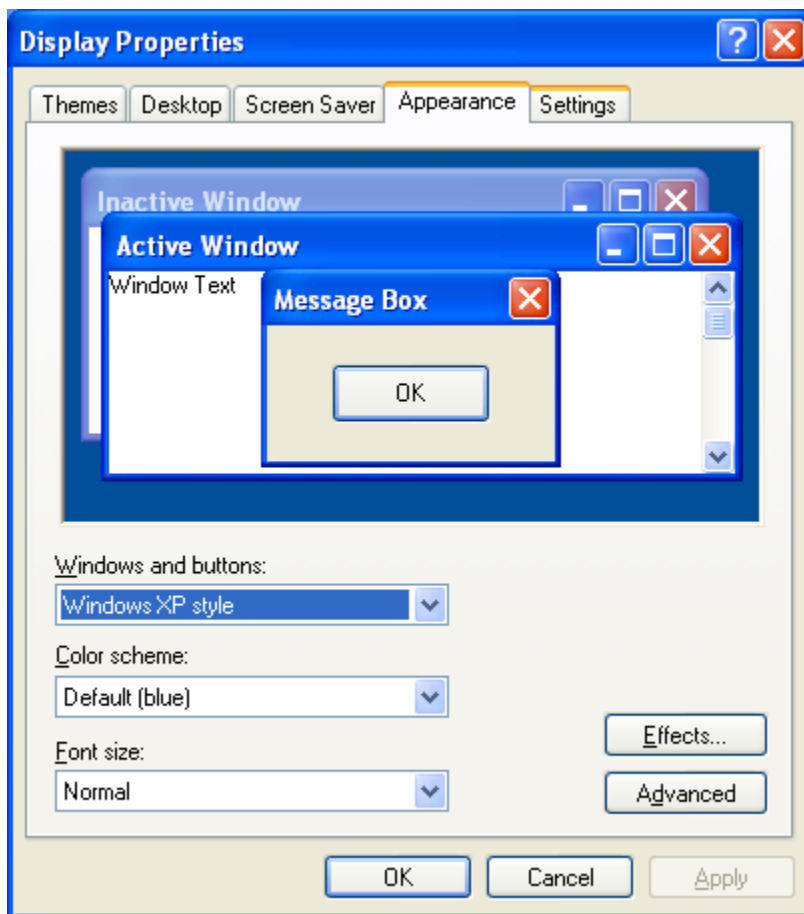


Figure 1: Adjusting the display's font size and resolution.

Computing Character Height

Five font height measurement parameters are available through the CDC function `GetTextMetrics()`, but only three are significant. Figure 2 shows the important font measurements. The `tmHeight` parameter represents the full height of the font, including descenders (for the characters g, j, p, q, and y) and any diacritics that appear over capital letters. The `tmExternalLeading` parameter is the distance between the top of the diacritic and the bottom of the descender from the line above. The sum of `tmHeight` and `tmExternalLeading` is the total character height. The value of `tmExternalLeading` can be 0.

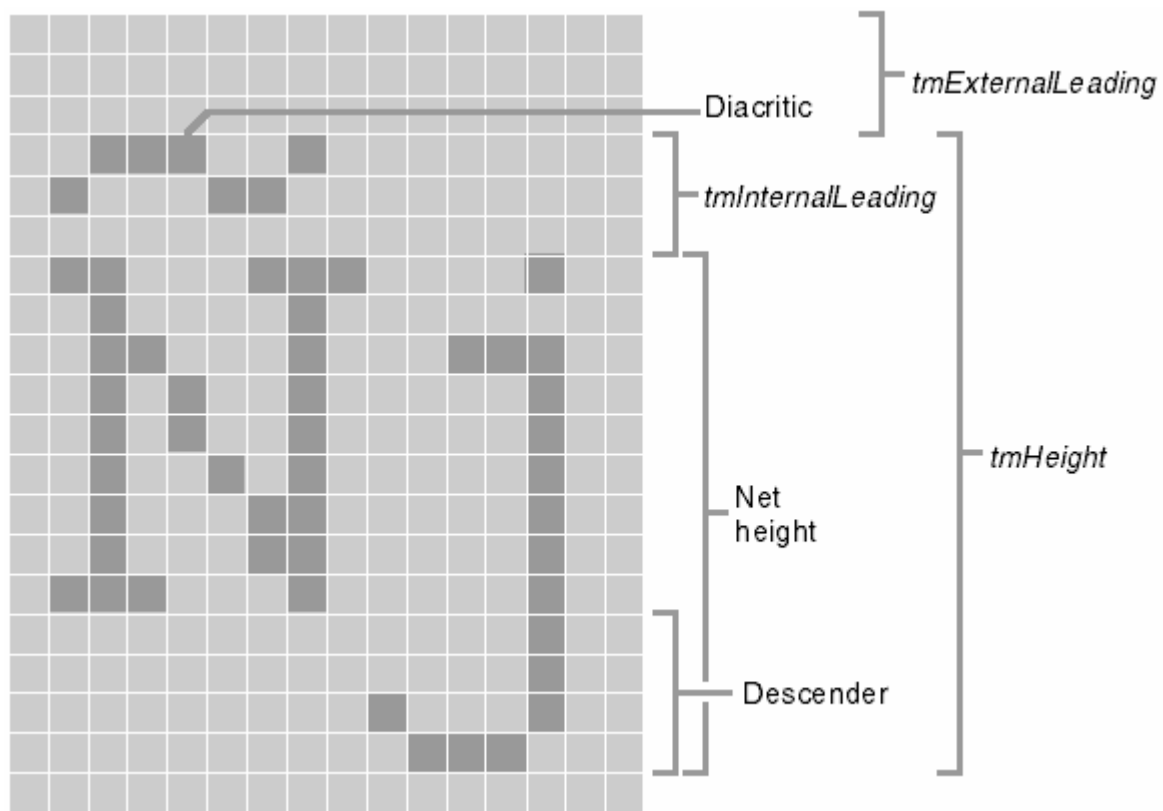


Figure 2: Font height measurements.

You would think that `tmHeight` would represent the font size in points. Wrong! Another `GetTextMetrics()` parameter, `tmInternalLeading`, comes into play. The point size corresponds to the difference between `tmHeight` and `tmInternalLeading`. With the `MM_TWIPS` mapping mode in effect, a selected 12-point font might have a `tmHeight` value of 295 logical units and a `tmInternalLeading` value of 55. The font's net height of 240 corresponds to the point size of 12.

The MYMFC4 Example

This example sets up a view window with the logical twips mapping mode. A text string is displayed in 10 point sizes with the Arial TrueType font. Here are the steps for building the application:

Run AppWizard to generate the MYMFC4 project. Start by choosing **New** from the **File** menu and then select **MFC AppWizard (exe)** on the **Project** tab. Select **Single Document** and deselect **Printing And Print Preview** and **ActiveX Controls**; accept all the other default settings. The options and the default class names are shown in the following Figure.

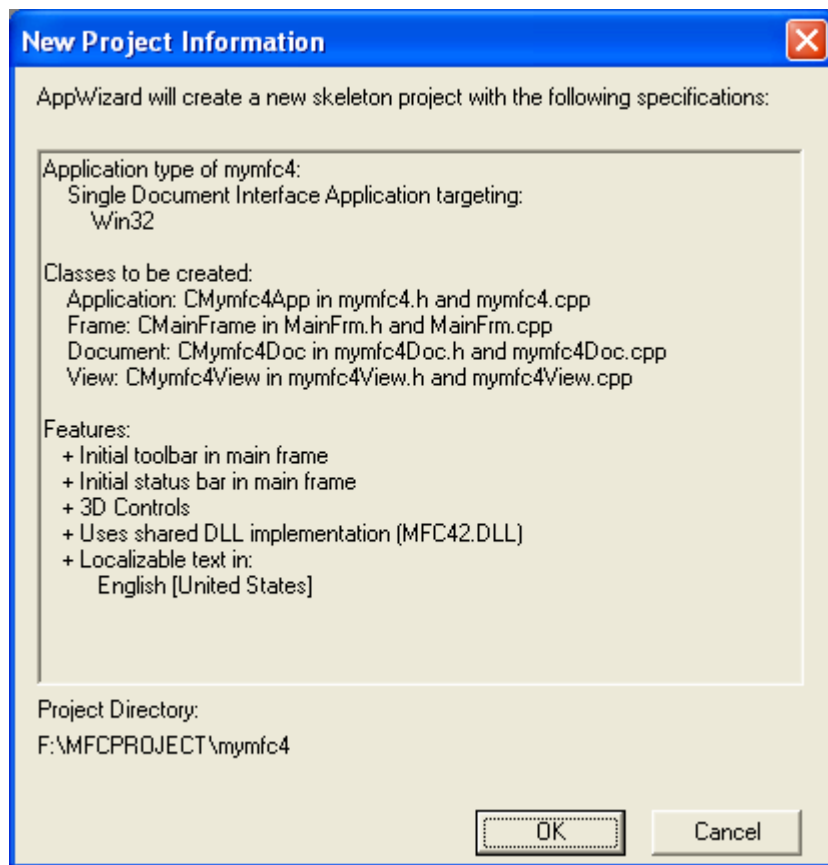


Figure 3: MYMFC4 project summary.

Use ClassWizard to override the `OnPrepareDC()` function in the `CMymfc4View` class. By clicking the **Edit Code** button, edit the code in **mymfc4View.cpp** as follows:

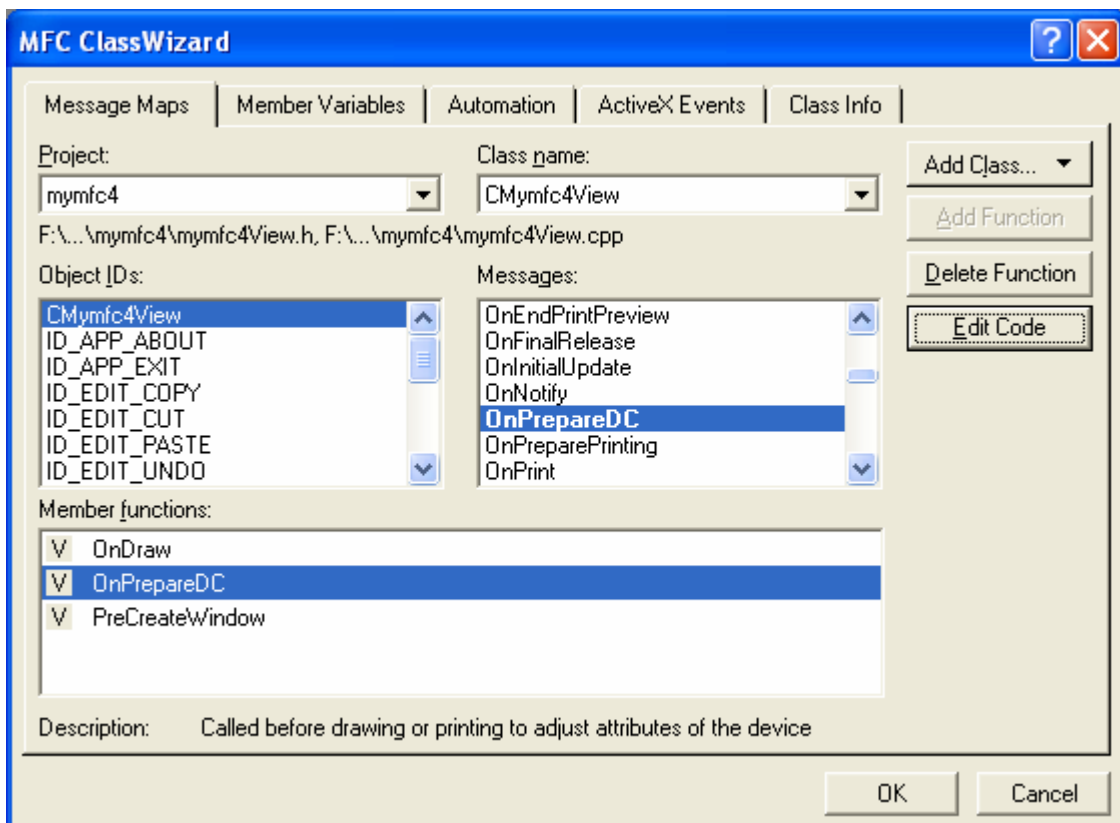


Figure 4: Using ClassWizard to override the OnPrepareDC () function.

```

void CMymfc4View::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
    pDC->SetMapMode(MM_ANISOTROPIC);
    pDC->SetWindowExt(1440, 1440);
    pDC->SetViewportExt(pDC->GetDeviceCaps(LOGPIXELSX),
        -pDC->GetDeviceCaps(LOGPIXELSY));
}

void CMymfc4View::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
    // TODO: Add your specialized code here and/or call the base class
    pDC->SetMapMode(MM_ANISOTROPIC);
    pDC->SetWindowExt(1440, 1440);
    pDC->SetViewportExt(pDC->GetDeviceCaps(LOGPIXELSX), |
        -pDC->GetDeviceCaps(LOGPIXELSY));
}

```

Listing 1.

Add a private ShowFont () helper function to the view class as shown below. In the ClassView, select the CMymfc4View class and right click your mouse. Select **Add Member Function...** context menu and fill up the **Function Type** and **Function Declaration**. As usual, you can add manually by opening the **mymfc4View.h**:



Figure 5: Adding a member function using ClassView.

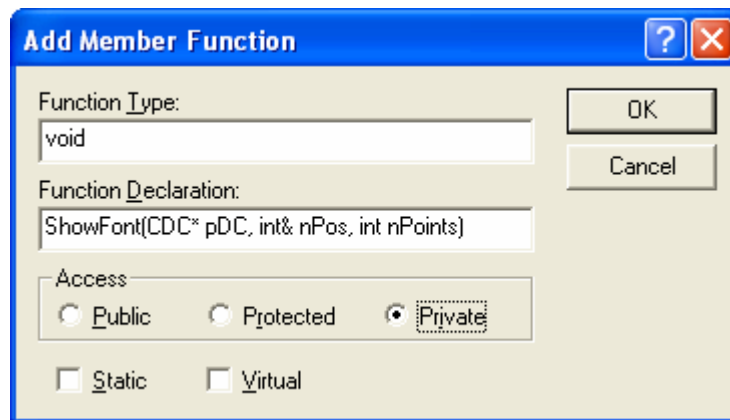


Figure 6: Member function type and declaration.

```
private:
    void ShowFont(CDC* pDC, int& nPos, int nPoints);
```

Then add the function itself in **mymfc4View.cpp**:

```
void CMymfc4View::ShowFont(CDC* pDC, int& nPos, int nPoints)
{
    TEXTMETRIC tm;
    CFont      fontText;
    CString    strText;
    CSize      sizeText;

    fontText.CreateFont(-nPoints * 20, 0, 0, 0, 400, FALSE, FALSE, 0,
                       ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                       CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                       DEFAULT_PITCH | FF_SWISS, "Arial");
    CFont* pOldFont = (CFont*) pDC->SelectObject(&fontText);
    pDC->GetTextMetrics(&tm);
    TRACE("points = %d, tmHeight = %d, tmInternalLeading = %d,
          " tmExternalLeading = %d\n", nPoints, tm.tmHeight,
```

```

        tm.tmInternalLeading, tm.tmExternalLeading);
    strText.Format("This is %d-point Arial", nPoints);
    sizeText = pDC->GetTextExtent(strText);
    TRACE("string width = %d, string height = %d\n", sizeText.cx, sizeText.cy);
    pDC->TextOut(0, nPos, strText);
    pDC->SelectObject(pOldFont);
    nPos -= tm.tmHeight + tm.tmExternalLeading;
}

void CMymfc4View::ShowFont(CDC *pDC, int &nPos, int nPoints)
{
    TEXTMETRIC tm;
    CFont      fontText;
    CString    strText;
    CSize      sizeText;

    fontText.CreateFont(-nPoints * 20, 0, 0, 0, 400, FALSE, FALSE, 0,
                        ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                        CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                        DEFAULT_PITCH | FF_SWISS, "Arial");
    CFont* pOldFont = (CFont*) pDC->SelectObject(&fontText);
    pDC->GetTextMetrics(&tm);
    TRACE("points = %d, tmHeight = %d, tmInternalLeading = %d,"
          " tmExternalLeading = %d\n", nPoints, tm.tmHeight,
          tm.tmInternalLeading, tm.tmExternalLeading);
    strText.Format("This is %d-point Arial", nPoints);
    sizeText = pDC->GetTextExtent(strText);
    TRACE("string width = %d, string height = %d\n", sizeText.cx, sizeText.cy);
    pDC->TextOut(0, nPos, strText);
    pDC->SelectObject(pOldFont);
    nPos -= tm.tmHeight + tm.tmExternalLeading;
}

```

Listing 2.

Edit the `OnDraw()` function in `mymfc4View.cpp`. AppWizard always generates a skeleton `OnDraw()` function for your view class. Find the function, and replace the code with the following:

```

void CMymfc4View::OnDraw(CDC* pDC)
{
    int nPosition = 0;

    for (int i = 6; i <= 24; i += 2)
    {
        ShowFont(pDC, nPosition, i);
    }
    TRACE("LOGPIXELSX = %d, LOGPIXELSY = %d\n",
          pDC->GetDeviceCaps(LOGPIXELSX),
          pDC->GetDeviceCaps(LOGPIXELSY));
    TRACE("HORZSIZE = %d, VERTSIZE = %d\n",
          pDC->GetDeviceCaps(HORZSIZE),
          pDC->GetDeviceCaps(VERTSIZE));
    TRACE("HORZRES = %d, VERTRES = %d\n",
          pDC->GetDeviceCaps(HORZRES),
          pDC->GetDeviceCaps(VERTRES));
}

```


```

void CMymfc4View::OnDraw(CDC* pDC)
{
    // TODO: add draw code for native data here
    int nPosition = 0;

    for (int i = 6; i <= 24; i += 2)
    {
        ShowFont(pDC, nPosition, i);
    }
    TRACE("LOGPIXELSX = %d, LOGPIXELSY = %d\n",
        pDC->GetDeviceCaps(LOGPIXELSX),
        pDC->GetDeviceCaps(LOGPIXELSY));
    TRACE("HORZSIZE = %d, VERTSIZE = %d\n",
        pDC->GetDeviceCaps(HORZSIZE),
        pDC->GetDeviceCaps(VERTSIZE));
    TRACE("HORZRES = %d, VERTRES = %d\n",
        pDC->GetDeviceCaps(HORZRES),
        pDC->GetDeviceCaps(VERTRES));
}

```

Listing 3.

Build and run the MYMFC4 program. You must run the program from the debugger if you want to see the output from the TRACE statements. You can choose **Go** from the **Start Debug** submenu of the **Build** menu in Visual C++, or click the following button on the **Build** toolbar, .

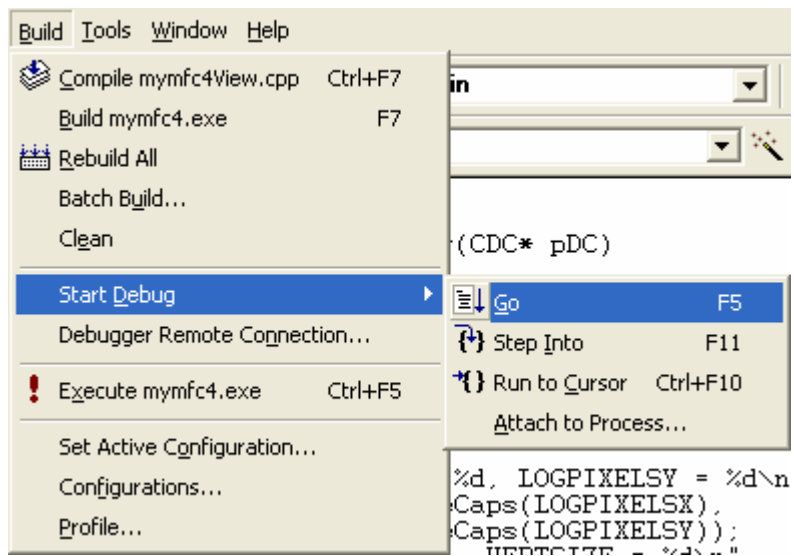


Figure 7: The Visual C++ program debug menu, tracing the TRACE.

The output (assuming the use of a standard VGA card) looks like the following.

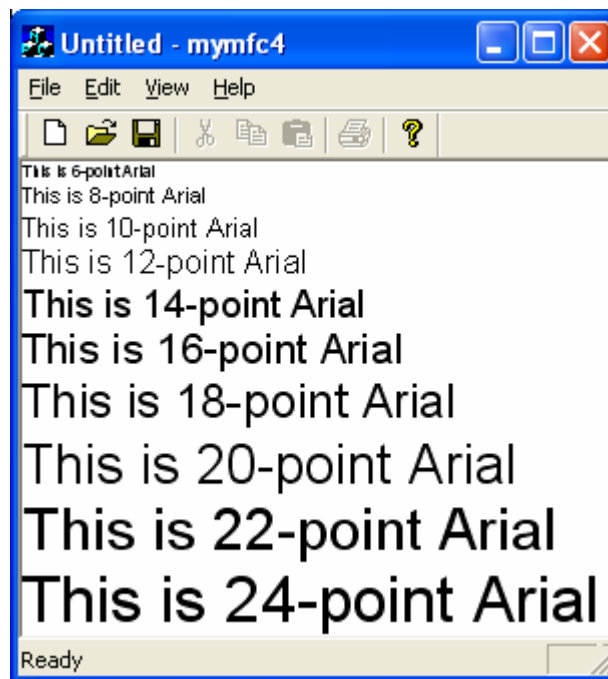


Figure 8: MYMFC4 program output.

Notice that the output string sizes don't quite correspond to the point sizes. This discrepancy results from the font engine's conversion of logical units to pixels. The program's trace output, partially shown below, shows the printout of font metrics. The numbers depend on your display driver and your video driver.

```

points = 6, tmHeight = 150, tmInternalLeading = 30, tmExternalLeading = 4
string width = 990, string height = 150
points = 8, tmHeight = 210, tmInternalLeading = 45, tmExternalLeading = 5
string width = 1380, string height = 210
points = 10, tmHeight = 240, tmInternalLeading = 45, tmExternalLeading = 6
string width = 1770, string height = 240
points = 12, tmHeight = 270, tmInternalLeading = 30, tmExternalLeading = 8
string width = 2130, string height = 270
points = 14, tmHeight = 330, tmInternalLeading = 45, tmExternalLeading = 9
string width = 2565, string height = 330
points = 16, tmHeight = 360, tmInternalLeading = 45, tmExternalLeading = 10
string width = 2850, string height = 360
points = 18, tmHeight = 405, tmInternalLeading = 45, tmExternalLeading = 12
string width = 3255, string height = 405
points = 20, tmHeight = 480, tmInternalLeading = 75, tmExternalLeading = 13
string width = 3705, string height = 480
points = 22, tmHeight = 495, tmInternalLeading = 60, tmExternalLeading = 14
string width = 4005, string height = 495
points = 24, tmHeight = 540, tmInternalLeading = 60, tmExternalLeading = 16
string width = 4320, string height = 540
LOGPIXELSX = 96, LOGPIXELSY = 96
HORZSIZE = 340, VERTSIZE = 255
HORZRES = 1024, VERTRES = 768

```

Figure 9: The Visual C++ Debug window messages generated using the TRACE.

The following is a discussion of the important elements in the MYMFC4 example.

Setting the Mapping Mode in the OnPrepareDC() Function

The application framework calls `OnPrepareDC()` prior to calling `OnDraw()`, so the `OnPrepareDC()` function is the logical place to prepare the device context. If you had other message handlers that needed the correct mapping mode, those functions would have contained calls to `OnPrepareDC()`.

The `ShowFont()` Private Member Function

`ShowFont()` contains code that is executed 10 times in a loop. With C, you would have made this a global function, but with C++ it's better to make it a private class member function, sometimes known as a helper function. This function creates the font, selects it into the device context, prints a string to the window, and then deselects the font. If you choose to include debug information in the program, `ShowFont()` also displays useful font metrics information, including the actual width of the string.

Calling `CFont::CreateFont`

This call includes lots of parameters, but the important ones are the first two: the font height and width. A width value of 0 means that the aspect ratio of the selected font will be set to a value specified by the font designer. If you put a nonzero value here, as you'll see in the next example, you can change the font's aspect ratio. If you want your font to be a specific point size, the `CreateFont()` font height parameter (the first parameter) must be negative. If you're using the `MM_TWIPS` mapping mode for a printer, for example, a height parameter of -240 ensures a true 12-point font, with `tmHeight - tmInternalLeading = 240`. A +240 height parameter gives you a smaller font, with `tmHeight = 240`.

The `MYMFC5` Example

This program is similar to `MYMFC4` except that it shows multiple fonts. The mapping mode is `MM_ANISOTROPIC`, with the scale dependent on the window size. The characters change size along with the window. This program effectively shows off some TrueType fonts and contrasts them with the old-style fonts. Here are the steps for building the application:

Run AppWizard to generate the `MYMFC5` project. The options and the default class names are shown here and it is similar to the `MYMFC4`.

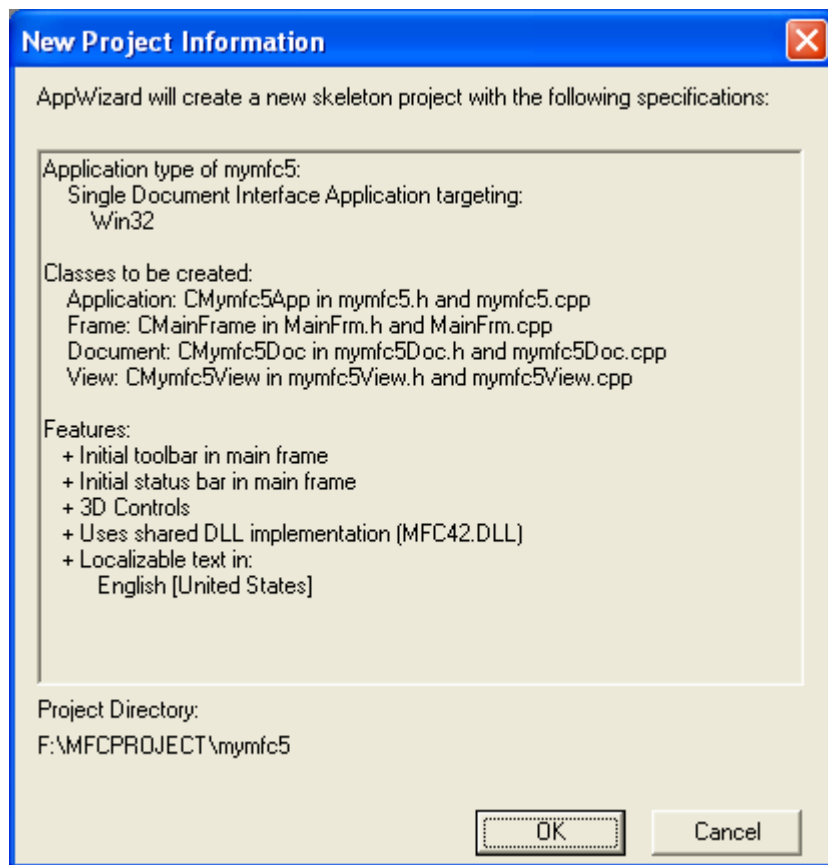


Figure 10: MYMFC5 project summary.

Use ClassWizard to override the `OnPrepareDC()` function in the `CMymfc5View` class. Edit the code in **mymfc5View.cpp** as shown below.

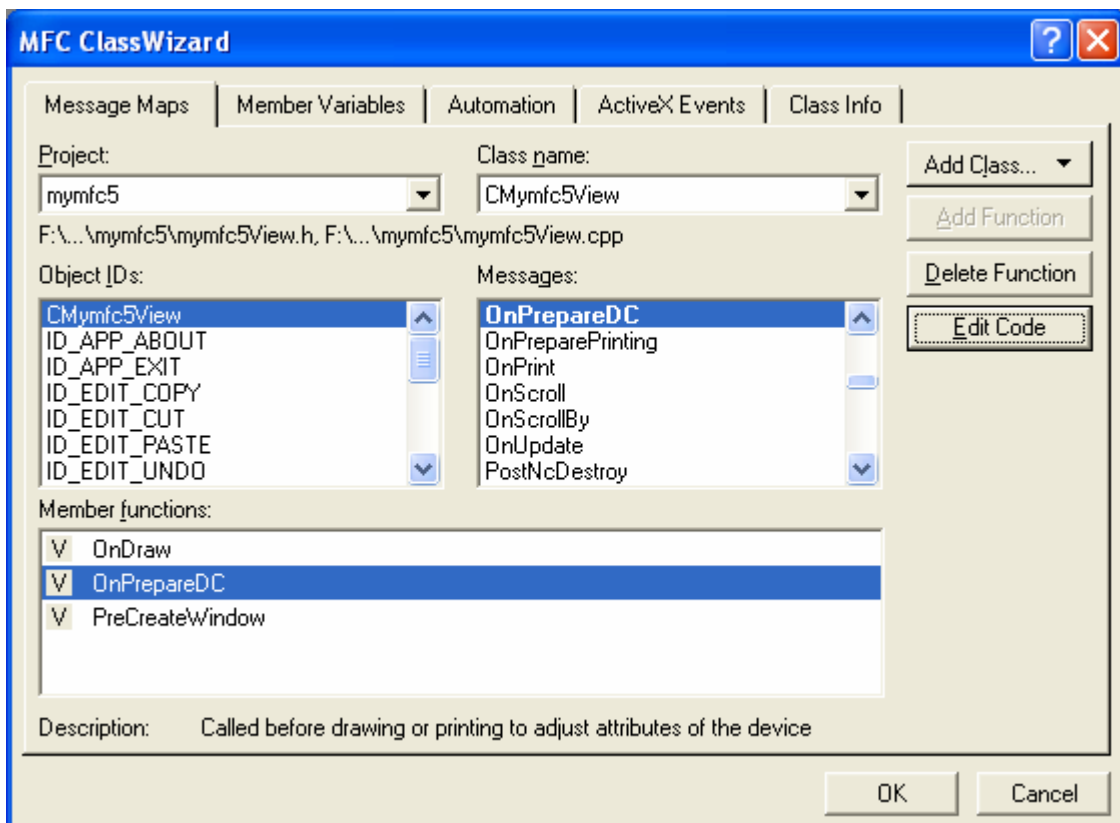


Figure 11: Using ClassWizard to override the OnPrepareDC() function.

```

void CMymfc5View::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
    CRect clientRect;

    GetClientRect(clientRect);
    pDC->SetMapMode(MM_ANISOTROPIC); // +y = down
    pDC->SetWindowExt(400, 450);
    pDC->SetViewportExt(clientRect.right, clientRect.bottom);
    pDC->SetViewportOrg(0, 0);
}

void CMymfc5View::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
    // TODO: Add your specialized code here and/or call the base class
    CRect clientRect;

    GetClientRect(clientRect);
    pDC->SetMapMode(MM_ANISOTROPIC); // +y = down
    pDC->SetWindowExt(400, 450);
    pDC->SetViewportExt(clientRect.right, clientRect.bottom);
    pDC->SetViewportOrg(0, 0);
}

```

Listing 4.

Add a private TraceMetrics() helper function to the view class. Add the following prototype in **mymfc5View.h**:

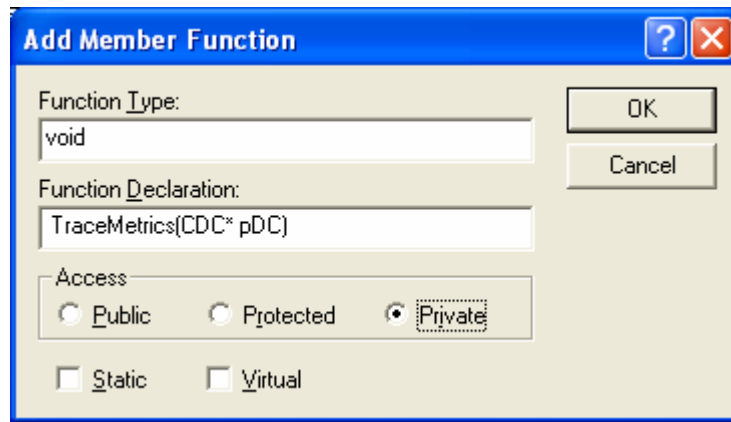


Figure 12: Adding a private TraceMetrics() helper function.

```
private:
    void TraceMetrics(CDC* pDC);
```

Then add the function itself in **mymfc5View.cpp**:

```
void CMymfc5View::TraceMetrics(CDC* pDC)
{
    TEXTMETRIC tm;
    char        szFaceName[100];

    pDC->GetTextMetrics(&tm);
    pDC->GetTextFace(99, szFaceName);
    TRACE("font = %s, tmHeight = %d, tmInternalLeading = %d,"
        " tmExternalLeading = %d\n", szFaceName, tm.tmHeight,
        tm.tmInternalLeading, tm.tmExternalLeading);
}

void CMymfc5View::TraceMetrics(CDC *pDC)
{
    TEXTMETRIC tm;
    char        szFaceName[100];

    pDC->GetTextMetrics(&tm);
    pDC->GetTextFace(99, szFaceName);
    TRACE("font = %s, tmHeight = %d, tmInternalLeading = %d,"
        " tmExternalLeading = %d\n", szFaceName, tm.tmHeight,
        tm.tmInternalLeading, tm.tmExternalLeading);
}
```

Listing 5.

Edit the OnDraw() function in **mymfc5View.cpp**. AppWizard always generates a skeleton OnDraw() function for your view class. Find the function, and edit the code as follows:

```
void CMymfc5View::OnDraw(CDC* pDC)
{
    CFont fontTest1, fontTest2, fontTest3, fontTest4;

    fontTest1.CreateFont(50, 0, 0, 0, 400, FALSE, FALSE, 0,
        ANSI_CHARSET, OUT_DEFAULT_PRECIS,
        CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
        DEFAULT_PITCH | FF_SWISS, "Arial");
    CFont* pOldFont = pDC->SelectObject(&fontTest1);
    TraceMetrics(pDC);
    pDC->TextOut(0, 0, "This is Arial, default width");
}
```

```

fontTest2.CreateFont(50, 0, 0, 0, 400, FALSE, FALSE, 0,
                    ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                    CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                    DEFAULT_PITCH | FF_MODERN, "Courier");
                    // not TrueType
pDC->SelectObject(&fontTest2);
TraceMetrics(pDC);
pDC->TextOut(0, 100, "This is Courier, default width");

fontTest3.CreateFont(50, 10, 0, 0, 400, FALSE, FALSE, 0,
                    ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                    CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                    DEFAULT_PITCH | FF_ROMAN, NULL);
pDC->SelectObject(&fontTest3);
TraceMetrics(pDC);
pDC->TextOut(0, 200, "This is generic Roman, variable width");

fontTest4.CreateFont(50, 0, 0, 0, 400, FALSE, FALSE, 0,
                    ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                    CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                    DEFAULT_PITCH | FF_MODERN, "LinePrinter");
pDC->SelectObject(&fontTest4);
TraceMetrics(pDC);
pDC->TextOut(0, 300, "This is LinePrinter, default width");
pDC->SelectObject(pOldFont);
}

void CMymfc5View::OnDraw(CDC* pDC)
{
    // TODO: add draw code for native data here
    CFont fontTest1, fontTest2, fontTest3, fontTest4;

    fontTest1.CreateFont(50, 0, 0, 0, 400, FALSE, FALSE, 0,
                        ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                        CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                        DEFAULT_PITCH | FF_SWISS, "Arial");
    CFont* pOldFont = pDC->SelectObject(&fontTest1);
    TraceMetrics(pDC);
    pDC->TextOut(0, 0, "This is Arial, default width");

    fontTest2.CreateFont(50, 0, 0, 0, 400, FALSE, FALSE, 0,
                        ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                        CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                        DEFAULT_PITCH | FF_MODERN, "Courier");
                        // not TrueType
    pDC->SelectObject(&fontTest2);
    TraceMetrics(pDC);
    pDC->TextOut(0, 100, "This is Courier, default width");

    fontTest3.CreateFont(50, 10, 0, 0, 400, FALSE, FALSE, 0,
                        ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                        CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                        DEFAULT_PITCH | FF_ROMAN, NULL);
    pDC->SelectObject(&fontTest3);
    TraceMetrics(pDC);
    pDC->TextOut(0, 200, "This is generic Roman, variable width");

    fontTest4.CreateFont(50, 0, 0, 0, 400, FALSE, FALSE, 0,
                        ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                        CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                        DEFAULT_PITCH | FF_MODERN, "LinePrinter");
    pDC->SelectObject(&fontTest4);
    TraceMetrics(pDC);
    pDC->TextOut(0, 300, "This is LinePrinter, default width");
    pDC->SelectObject(pOldFont);
}

```

Listing 6.

Build and run the MYMFC5 program. Run the program from the debugger to see the TRACE output. The program's output window is shown here.

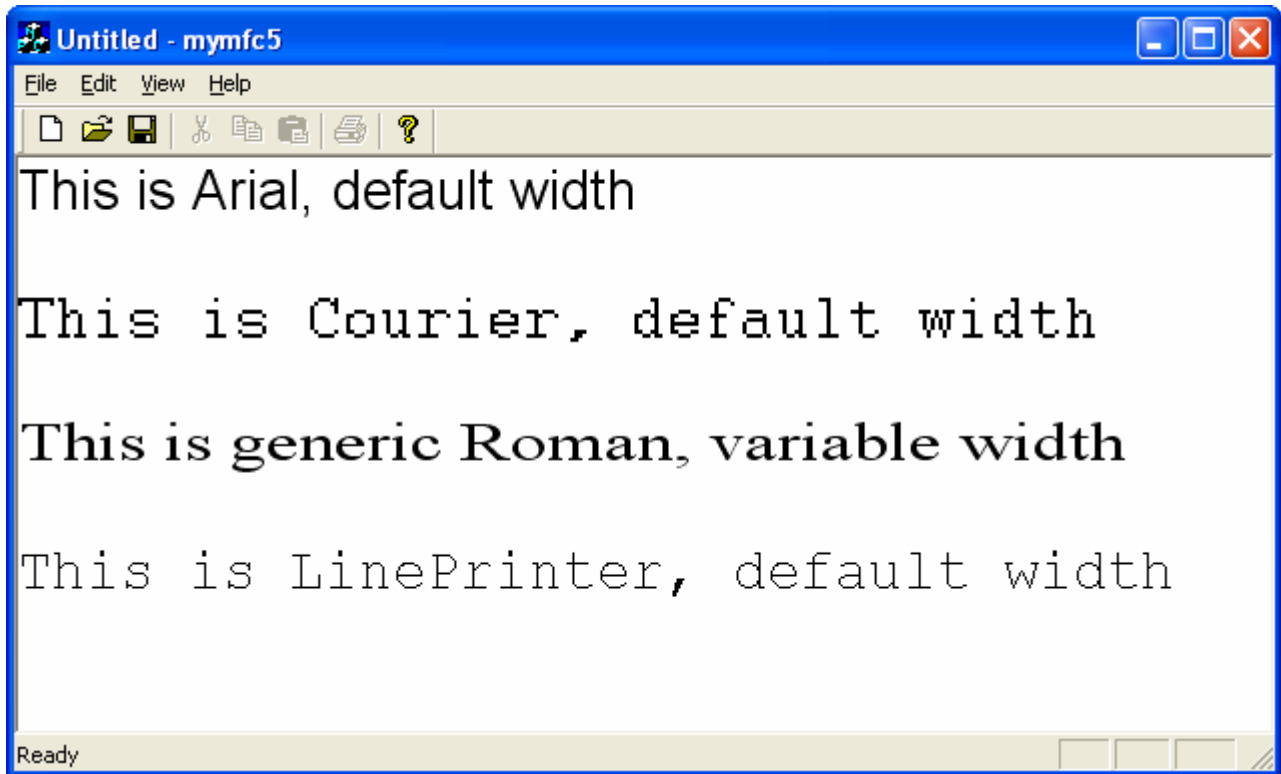


Figure 13: MYMFC5 program output.

Resize the window to make it smaller, and watch the font sizes change. Compare this window with the previous one.

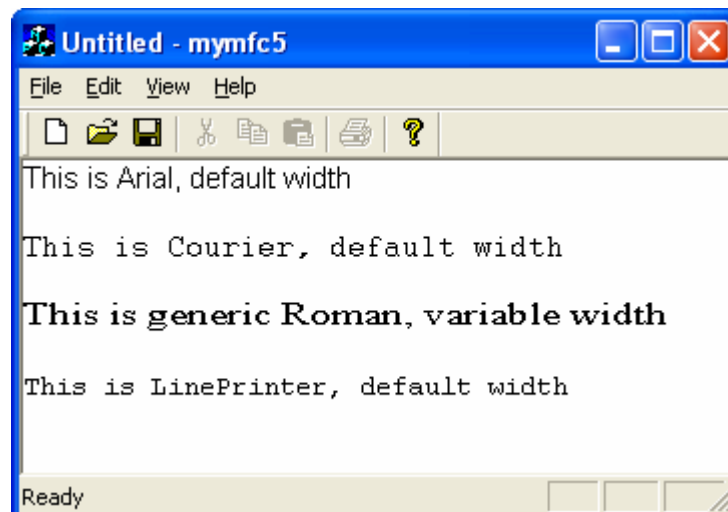


Figure 14: MYMFC5 program output when the window is resized.

If you continue to downsize the window, notice how the Courier font stops shrinking after a certain size and how the Roman font width changes.

The MYMFC5 Program Elements

Following is a discussion of the important elements in the MYMFC5 example.

The `OnDraw()` Member Function

The `OnDraw()` function displays character strings in four fonts, as follows:

Fonts	Description
fontTest1	The TrueType font Arial with default width selection.
fontTest2	The old-style font Courier with default width selection. Notice how jagged the font appears in larger sizes.
fontTest3	The generic Roman font for which Windows supplies the TrueType font Times New Roman with programmed width selection. The width is tied to the horizontal window scale, so the font stretches to fit the window.
fontTest4	The LinePrinter font is specified, but because this is not a Windows font for the display, the font engine falls back on the <code>FF_MODERN</code> specification and chooses the TrueType Courier New font.

Table 5.

The `TraceMetrics()` Helper Function

The `TraceMetrics()` helper function calls `CDC::GetTextMetrics` and `CDC::GetTextFace` to get the current font's parameters, which it prints in the Debug window.

The MYMFC6 Example: `CScrollView` Revisited

You saw the `CScrollView` class in previous Module. The MYMFC6 program allows the user to move an ellipse with a mouse by "capturing" the mouse, using a scrolling window with the `MM_LOENGLISH` mapping mode. Keyboard scrolling is left out, but you can add it by borrowing the `OnKeyDown()` member function from MYMFC3 example in previous Module. Instead of a stock brush, we'll use a pattern brush for the ellipse: a real GDI object. There's one complication with pattern brushes: you must reset the origin as the window scrolls; otherwise, strips of the pattern don't line up and the effect is ugly. As with the MYMFC3 program, this example involves a view class derived from `CScrollView`. Here are the steps to create the application:

Run AppWizard to generate the MYMFC6 project. Be sure to set the view base class to `CScrollView` in step 6. The options and the default class names are shown here.

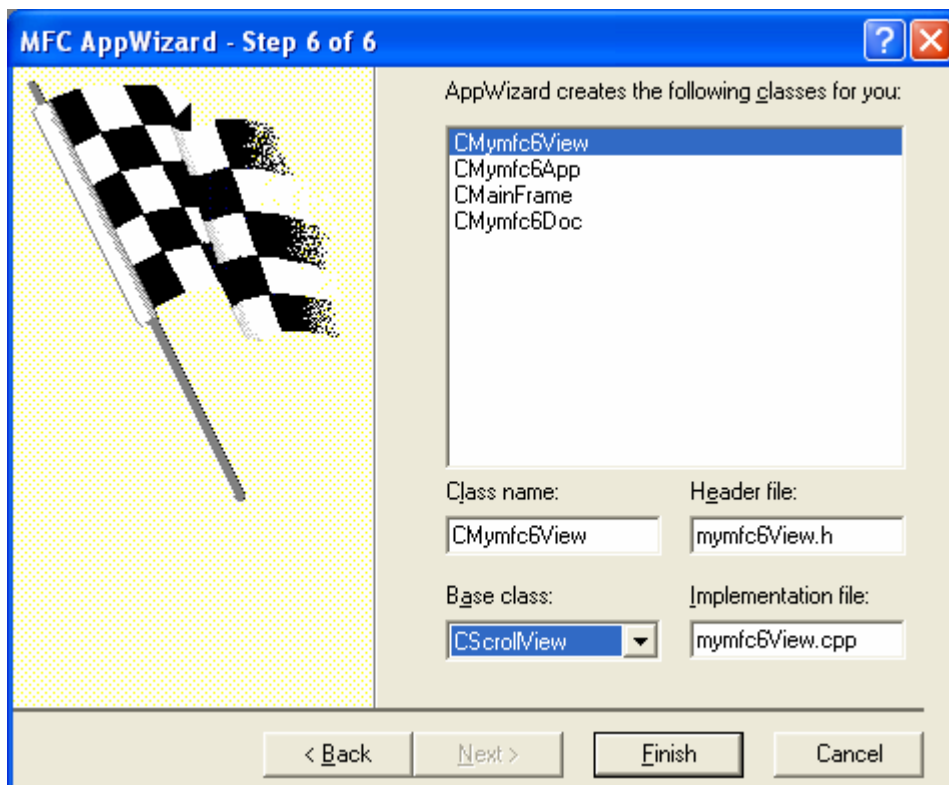


Figure 15: Step 6 of the MYMFC6 project, using CScrollView as the View base class.

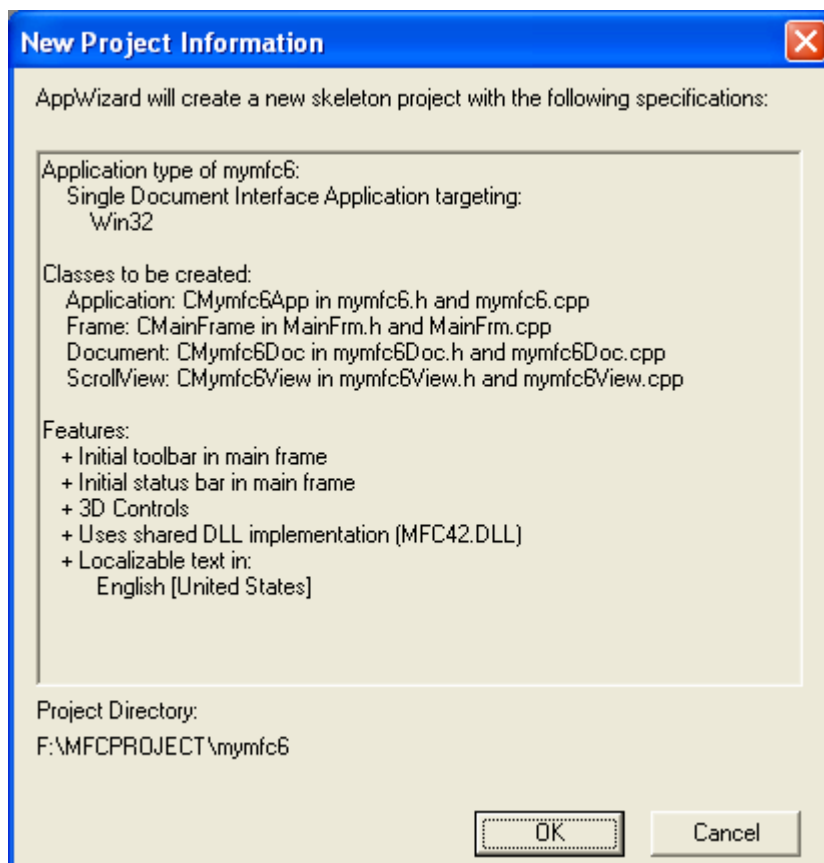


Figure 16: MYMFC6 project summary.

Edit the CMymfc6View class header in the file **mymfc6View.h**. Add the following lines in the class CMymfc6View declaration:

```
private:
    const CSize m_sizeEllipse;
    CPoint m_pointTopLeft; // logical, top left of ellipse rectangle
    CSize m_sizeOffset; // device, from rect top left

    // to capture point
    BOOL m_bCaptured;

// Attributes
public:
    CMymfc6Doc* GetDocument();

private:
    const CSize m_sizeEllipse;
    CPoint m_pointTopLeft; // logical, top left of ellipse rectangle
    CSize m_sizeOffset; // device, from rect top left

    // to capture point
    BOOL m_bCaptured;
```

Listing 7.

Use ClassWizard to add three message handlers to the CMymfc6View class. Add the message handlers as follows:

Message	Member Function
WM_LBUTTONDOWN	OnLButtonDown()
WM_LBUTTONUP	OnLButtonUp()
WM_MOUSEMOVE	OnMouseMove()

Table 6.

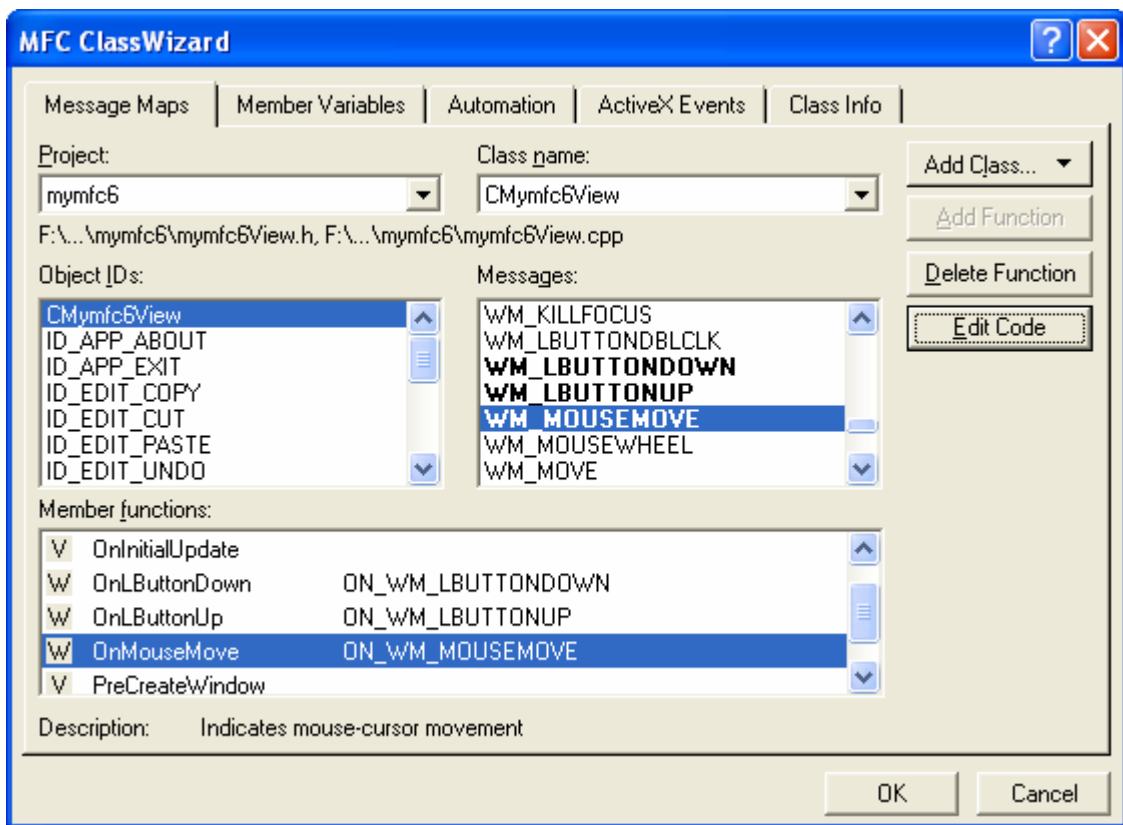


Figure 17: Adding three message handlers using ClassWizard.

Edit the `CMymfc6View` message handler functions by clicking the **Edit Code** button. ClassWizard generated the skeletons for the functions listed in the preceding step. Find the functions in `mymfc6View.cpp`, and code them as follows.

```
void CMymfc6View::OnLButtonDown(UINT nFlags, CPoint point)
{
    CRect rectEllipse(m_pointTopLeft, m_sizeEllipse); // still logical
    CRgn circle;

    CClientDC dc(this);
    OnPrepareDC(&dc);
    dc.LPtoDP(rectEllipse); // Now it's in device coordinates
    circle.CreateEllipticRgnIndirect(rectEllipse);
    if (circle.PtInRegion(point)) {
        // Capturing the mouse ensures subsequent LButtonUp message
        SetCapture();
        m_bCaptured = TRUE;
        CPoint pointTopLeft(m_pointTopLeft);
        dc.LPtoDP(&pointTopLeft);
        m_sizeOffset = point - pointTopLeft; // device coordinates
        // New mouse cursor is active while mouse is captured
        ::SetCursor(::LoadCursor(NULL, IDC_CROSS));
    }
}
```

```

void CMymfc6View::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    CRect rectEllipse(m_pointTopLeft, m_sizeEllipse); // still logical
    CRgn circle;

    CClientDC dc(this);
    OnPrepareDC(&dc);
    dc.LPtoDP(rectEllipse); // Now it's in device coordinates
    circle.CreateEllipticRgnIndirect(rectEllipse);
    if (circle.PtInRegion(point)) {
        // Capturing the mouse ensures subsequent LButtonDown message
        SetCapture();
        m_bCaptured = TRUE;
        CPoint pointTopLeft(m_pointTopLeft);
        dc.LPtoDP(&pointTopLeft);
        m_sizeOffset = point - pointTopLeft; // device coordinates
        // New mouse cursor is active while mouse is captured
        ::SetCursor(::LoadCursor(NULL, IDC_CROSS));
    }
}

```

Listing 8.

```

void CMymfc6View::OnLButtonUp(UINT nFlags, CPoint point)
{
    if (m_bCaptured)
    {
        ::ReleaseCapture();
        m_bCaptured = FALSE;
    }
}

```

```

void CMymfc6View::OnMouseMove(UINT nFlags, CPoint point)
{
    if (m_bCaptured)
    {
        CClientDC dc(this);
        OnPrepareDC(&dc);
        CRect rectOld(m_pointTopLeft, m_sizeEllipse);
        dc.LPtoDP(rectOld);
        InvalidateRect(rectOld, TRUE);
        m_pointTopLeft = point - m_sizeOffset;
        dc.DPtoLP(&m_pointTopLeft);
        CRect rectNew(m_pointTopLeft, m_sizeEllipse);
        dc.LPtoDP(rectNew);
        InvalidateRect(rectNew, TRUE);
    }
}

```

```

void CMymfc6View::OnLButtonUp(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    if (m_bCaptured)
    {
        ::ReleaseCapture();
        m_bCaptured = FALSE;
    }
}

void CMymfc6View::OnMouseMove(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    if (m_bCaptured)
    {
        CClientDC dc(this);
        OnPrepareDC(&dc);
        CRect rectOld(m_pointTopLeft, m_sizeEllipse);
        dc.LPtoDP(rectOld);
        InvalidateRect(rectOld, TRUE);    const CSize CMymfc6View::m_sizeEllipse
        m_pointTopLeft = point - m_sizeOffset;
        dc.DPtoLP(&m_pointTopLeft);
        CRect rectNew(m_pointTopLeft, m_sizeEllipse);
        dc.LPtoDP(rectNew);
        InvalidateRect(rectNew, TRUE);
    }
}

```

Listing 9.

Edit the CMymfc6View constructor, the OnDraw() function, and the OnInitialUpdate() function. AppWizard generated these skeleton functions. Find them in **mymfc6View.cpp**, and code them as follows:

```

CMymfc6View::CMymfc6View() : m_sizeEllipse(100, -100),
                             m_pointTopLeft(0, 0),
                             m_sizeOffset(0, 0)
{
    m_bCaptured = FALSE;
}

CMymfc6View::CMymfc6View() : m_sizeEllipse(100, -100),
                             m_pointTopLeft(0, 0),
                             m_sizeOffset(0, 0)
{
    m_bCaptured = FALSE;
}

```

Listing 10.

```

void CMymfc6View::OnDraw(CDC* pDC)
{
    CBrush brushHatch(HS_DIAGCROSS, RGB(255, 0, 0));
    // logical (0, 0)
    CPoint point(0, 0);

    // In device coordinates, align the brush with the window origin
    pDC->LPtoDP(&point);
    pDC->SetBrushOrg(point);

    pDC->SelectObject(&brushHatch);
    pDC->Ellipse(CRect(m_pointTopLeft, m_sizeEllipse));
    pDC->SelectStockObject(BLACK_BRUSH); // Deselect brushHatch
    pDC->Rectangle(CRect(100, -100, 200, -200)); // Test invalid rect
}

```

```

void CMymfc6View::OnDraw(CDC* pDC)
{
    // TODO: add draw code for native data here
    CBrush brushHatch(HS_DIAGCROSS, RGB(255, 0, 0));
    CPoint point(0, 0); // logical (0, 0)

    pDC->LPtoDP(&point); // In device coordinates,
    pDC->SetBrushOrg(point); // align the brush with
    // the window origin

    pDC->SelectObject(&brushHatch);
    pDC->Ellipse(CRect(m_pointTopLeft, m_sizeEllipse));
    pDC->SelectStockObject(BLACK_BRUSH); // Deselect brushHatch
    pDC->Rectangle(CRect(100, -100, 200, -200)); // Test invalid rect
}

```

Listing 11.

```

void CMymfc6View::OnInitialUpdate()
{
    CScrollView::OnInitialUpdate();

    CSize sizeTotal(800, 1050); // 8-by-10.5 inches
    CSize sizePage(sizeTotal.cx / 2, sizeTotal.cy / 2);
    CSize sizeLine(sizeTotal.cx / 50, sizeTotal.cy / 50);
    SetScrollSizes(MM_LOENGLISH, sizeTotal, sizePage, sizeLine);
}

void CMymfc6View::OnInitialUpdate()
{
    // TODO: calculate the total size of this view
    CScrollView::OnInitialUpdate();

    CSize sizeTotal(800, 1050); // 8-by-10.5 inches
    CSize sizePage(sizeTotal.cx / 2, sizeTotal.cy / 2);
    CSize sizeLine(sizeTotal.cx / 50, sizeTotal.cy / 50);
    SetScrollSizes(MM_LOENGLISH, sizeTotal, sizePage, sizeLine);
}

```

Listing 12.

Build and run the MYMFC6 program. The program allows an ellipse to be dragged with the mouse, and it allows the window to be scrolled through. The program's window should look like the one shown here. As you move the ellipse, observe the black rectangle. You should be able to see the effects of invalidating the rectangle.

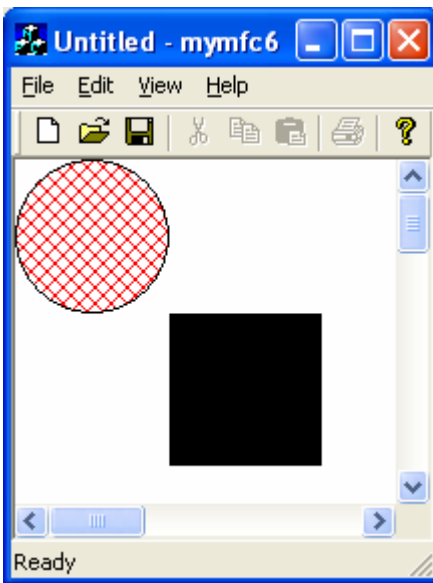


Figure 18: MYMFC6 program output with horizontal and vertical scroll bar.

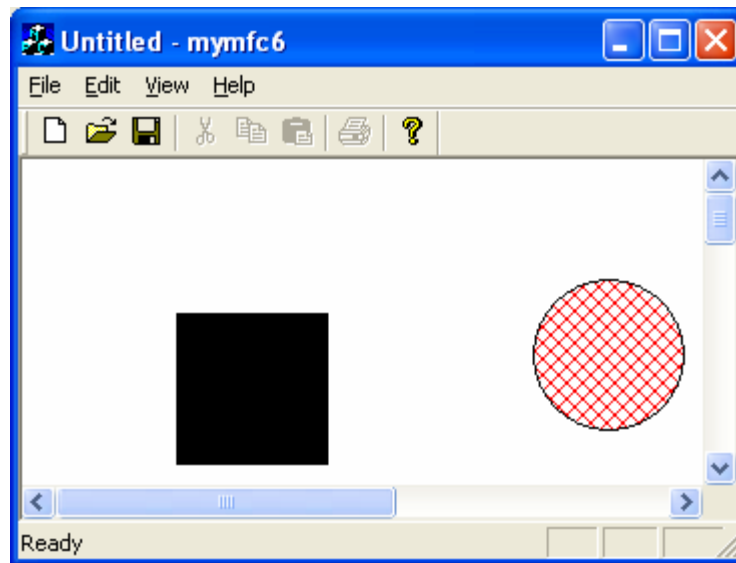


Figure 19: MYMFC6 program output, dragging an ellipse.

The MYMFC6 Program Elements

Following is a discussion of the important elements in the MYMFC6 example.

The `m_sizeEllipse` and `m_pointTopLeft` Data Members

Rather than store the ellipse's bounding rectangle as a single `CRect` object, the program separately stores its size (`m_sizeEllipse`) and the position of its top left corner (`m_pointTopLeft`). To move the ellipse, the program merely recalculates `m_pointTopLeft`, and any round-off errors in the calculation won't affect the size of the ellipse.

The `m_sizeOffset` Data Member

When `OnMouseMove()` moves the ellipse, the relative position of the mouse within the ellipse must be the same as it was when the user first pressed the left mouse button. The `m_sizeOffset` object stores this original offset of the mouse from the top left corner of the ellipse rectangle.

The `m_bCaptured` Data Member

The `m_bCaptured` Boolean variable is set to `TRUE` when mouse tracking is in progress.

The `SetCapture()` and `ReleaseCapture()` Functions

`SetCapture()` is the `CWnd` member function that "captures" the mouse, such that mouse movement messages are sent to this window even if the mouse cursor is outside the window. An unfortunate side effect of this function is that the ellipse can be moved outside the window and "lost." A desirable and necessary effect is that all subsequent mouse messages are sent to the window, including the `WM_LBUTTONDOWN` message, which would otherwise be lost. The Win32 `ReleaseCapture()` function turns off mouse capture.

The `SetCursor()` and `LoadCursor()` Win32 Functions

The MFC library does not "wrap" some Win32 functions. By convention, we use the C++ scope resolution operator (`::`) when calling Win32 functions directly. In this case, there is no potential for conflict with a `CView` member function, but you can deliberately choose to call a Win32 function in place of a class member function with the same name. In that case, the `::` operator ensures that you call the globally scoped Win32 function. When the first parameter is `NULL`, the `LoadCursor()` function creates a cursor resource from the specified predefined mouse cursor that Windows uses. The `SetCursor()` function activates the specified cursor resource. This cursor remains active as long as the mouse is captured.

The `CScrollView::OnPrepareDC` Member Function

The `CView` class has a virtual `OnPrepareDC()` function that does nothing. The `CScrollView` class implements the function for the purpose of setting the view's mapping mode and origin, based on the parameters that you passed to `SetScrollSizes()` in `OnCreate()`. The application framework calls `OnPrepareDC()` for you prior to calling `OnDraw()`, so you don't need to worry about it. You must call `OnPrepareDC()` yourself in any other message handler function that uses the view's device context, such as `OnLButtonDown()` and `OnMouseMove()`.

The `OnMouseMove()` Coordinate Transformation Code

As you can see, this function contains several translation statements. The logic can be summarized by the following steps:

1. Construct the previous ellipse rectangle and convert it from logical to device coordinates.
2. Invalidate the previous rectangle.
3. Update the top left coordinate of the ellipse rectangle.
4. Construct the new rectangle and convert it to device coordinates.
5. Invalidate the new rectangle.

The function calls `InvalidateRect()` twice. Windows "saves up" the two invalid rectangles and computes a new invalid rectangle that is the union of the two, intersected with the client rectangle.

The `OnDraw()` Function

The `SetBrushOrg()` call is necessary to ensure that all of the ellipse's interior pattern lines up when the user scrolls through the view. The brush is aligned with a reference point, which is at the top left of the logical window, converted to device coordinates. This is a notable exception to the rule that `CDC` member functions require logical coordinates.

The `CScrollView SetScaleToFitSize()` Mode

The `CScrollView` class has a stretch-to-fit mode that displays the entire scrollable area in the view window. The Windows `MM_ANISOTROPIC` mapping mode comes into play, with one restriction: positive `y` values always increase in

the down direction, as in `MM_TEXT` mode. To use the stretch-to-fit mode, make the following call in your view's function in place of the call to `SetScrollSizes()`:

```
SetScaleToFitSize(sizeTotal);
```

You can make this call in response to a **Shrink To Fit** menu command. Thus, the display can toggle between scrolling mode and shrink-to-fit mode.

Using the Logical Twips Mapping Mode in a Scrolling View

The MFC `CScrollView` class allows you to specify only standard mapping modes. You can use a new class, `CLogScrollView` to accommodate the logical twips mode.

Further reading and digging:

1. MSDN [MFC 6.0 class library online documentation](#) - used throughout this Tutorial.
2. MSDN [MFC 7.0 class library online documentation](#) - used in .Net framework and also backward compatible with 6.0 class library
3. [MSDN Library](#)
4. [Windows data type](#).
5. [Win32 programming Tutorial](#).
6. [The best of C/C++, MFC, Windows and other related books](#).
7. Unicode and Multibyte character set: [Story](#) and [program examples](#).