Module 3: Basic Event Handling, Mapping Modes, and a Scrolling View

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:

Basic Event Handling, Mapping Modes, and a Scrolling View

In the previous Module, you saw how the MFC Library application framework called the view class's virtual `OnDraw()` function. If you look at the MSDN documentation for the `CView` class and its base class, `CWnd`, you'll see several hundred member functions. Functions whose names begin with **On** - such as `OnKeyDown()` and `OnLButtonUp()` - are member functions that the application framework calls in response to various Windows "**events**" such as keystrokes and mouse clicks.

Most of these application framework-called functions, such as `OnKeyDown()`, aren't virtual functions and thus require more programming steps. This Module explains how to use the Visual C++ ClassWizard to set up the message map structure necessary for connecting the application framework to your functions' code.

The first two examples use an ordinary `CView` class. In MYMFC1, you'll learn about the interaction between user-driven events and the `OnDraw()` function. In MYMFC2, you'll see the effects of different Windows mapping modes. More often than not, you'll want a scrolling view. The last example, MYMFC3, uses `CScrollView` in place of the `CView` base class. This allows the MFC library application framework to insert scroll bars and connect them to the view.

Getting User Input: Message Map Functions

Your mymfc application from previous Module did not accept user input other than the standard Microsoft Windows resizing and window close commands. The window contained menus and a toolbar, but these were not "connected" to the view code. The menus and the toolbar won't be discussed until the time comes, because they depend on the frame class, but plenty of other Windows input sources will keep you busy until then. Before you can process any Windows event, even a mouse click, however, you must learn how to use the **MFC message map system**.

The Message Map

When the user presses the left mouse button in a view window, Windows **sends a message** - specifically `WM_LBUTTONDOWN` - to that window. If your program needs to take action in response to `WM_LBUTTONDOWN`, your view class must have a **member function** that looks like this:

```
void CMyView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // event processing code here
}
```

Your class header file must also have the corresponding **prototype**:

```
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
```

The `afx_msg` notation is a "no-op" that alerts you that this is a prototype for a message map function. Next, your code file needs a **message map macro** that connects your `OnLButtonDown()` function to the application framework:

```
BEGIN_MESSAGE_MAP(CMyView, CView)
    ON_WM_LBUTTONDOWN() // entry specifically for OnLButtonDown
    // other message map entries
END_MESSAGE_MAP()
```

Finally, your class header file needs the statement:

```
DECLARE_MESSAGE_MAP()
```

How do you know which function goes with which Windows message? The MFC library online documentation includes a table that lists all standard Windows messages and corresponding member function prototypes. You can manually code the message-handling functions - indeed, that is still necessary for certain messages. Fortunately, Visual C++ provides a tool, **ClassWizard**, that automates the coding of most message map functions.

Saving the View's State - Class Data Members

If your program accepts user input, you'll want the user to have some visual feedback. The view's `OnDraw()` function draws an image based on the view's current "state," and user actions can alter that state. In a full-blown MFC application, the **document** object holds the state of the application, but you're not to that point yet. For now, you'll use two view class data members, `m_rectEllipse` and `m_nColor`. The first is an object of class `CRect`, which holds the current bounding rectangle of an ellipse, and the second is an integer that holds the current ellipse color value. By convention, MFC library non-static class data member names begin with **`m_`**.
You'll make a message-mapped member function toggle the ellipse color (the view's state) between gray and white and the toggle is activated by the event of pressing the left mouse button. The initial values of `m_rectEllipse` and `m_nColor` are set in the view's constructor, and the color is changed in the `OnLButtonDown()` member function. Why not use a global variable for the view's state? Because if you did, you'd be in trouble if your application had multiple views. Besides, encapsulating data in objects is a big part of what object-oriented programming is all about.

Initializing a View Class Data Member

The most efficient place to initialize a **class data member** is in the constructor, like this:

```
CMyView::CMyView() : m_rectEllipse(0, 0, 200, 200) {...}
```

You could initialize `m_nColor` with the same syntax. Because we're using a built-in type (integer), the generated code is the same if you use an assignment statement in the constructor body.

Invalid Rectangle Theory

The `OnLButtonDown()` function could toggle the value of `m_nColor` all day, but if that's all it did, the `OnDraw()` function wouldn't get called (unless, for example, the user resized the view window). The `OnLButtonDown()` function must call the `InvalidateRect()` function (a member function that the view class inherits from `CWnd`). `InvalidateRect()` triggers a Windows `WM_PAINT` message, which is mapped in the `CView` class to call to the virtual `OnDraw()` function. If necessary, `OnDraw()` can access the "invalid rectangle" parameter that was passed to `InvalidateRect()`.
There are two ways to optimize painting in Windows. First of all, you must be aware that Windows updates only those pixels that are inside the invalid rectangle. Thus, the smaller you make the invalid rectangle (in the `OnLButtonDown()` handler, for instance), the quicker it can be repainted. Second, it's a waste of time to execute drawing instructions outside the invalid rectangle. Your `OnDraw()` function could call the `CDC` member function `GetClipBox()` to determine the invalid rectangle, and then it could avoid drawing objects outside it. Remember that `OnDraw()` is being called not only in response to your `InvalidateRect()` call but also when the user resizes or exposes the window. Thus, `OnDraw()` is responsible for all drawing in a window, and it has to adapt to whatever invalid rectangle it gets.

The Window's Client Area

A window has a **rectangular client area** that excludes the border, caption bar, menu bar, and any docking toolbars. The `CWnd` member function `GetClientRect()` supplies you with the client-area dimensions. Normally, you're not allowed to draw outside the client area, and most **mouse messages** are received only when the mouse cursor is in the client area. Various child windows, including the toolbar window, the view window, and the status bar window, occupy the main frame window's **client area**, as shown in Figure 1.

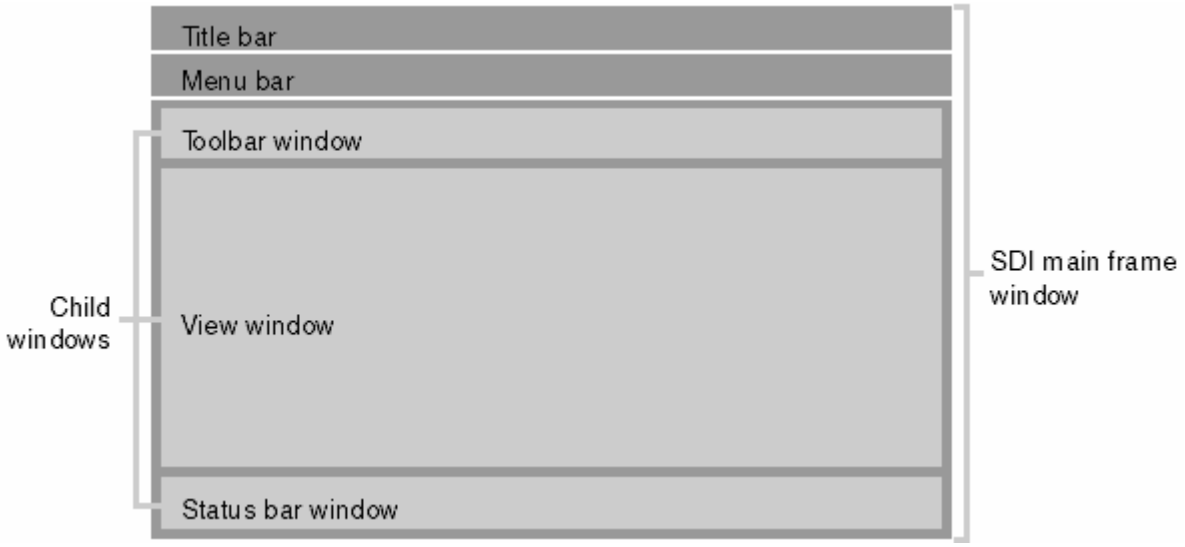

Figure 1: Window client area.

`CRect`, `CPoint`, and `CSize` Arithmetic

The `CRect`, `CPoint`, and `CSize` classes are derived from the Windows `RECT`, `POINT`, and `SIZE` structures, and thus they inherit public integer data members as follows:

Class	Data member
`CRect`	left, top, right, bottom

CPoint	x, y
CSize	cx, cy

Table 1.

If you look in the Microsoft Foundation Class Reference, you will see that these three classes have a number of overloaded operators. You can, among other things, do the following:

- Add a `CSize` object to a `CPoint` object.
- Subtract a `CSize` object from a `CPoint` object.
- Subtract one `CPoint` object from another, yielding a `CSize` object.
- Add a `CPoint` or `CSize` object to a `CRect` object.
- Subtract a `CPoint` or `CSize` object from a `CRect` object.

The `CRect` class has member functions that relate to the `CSize` and `CPoint` classes. For example, the `TopLeft()` member function returns a `CPoint` object, and the `Size()` member function returns a `CSize` object. From this, you can begin to see that a `CSize` object is the "difference between two `CPoint` objects" and that you can "bias" a `CRect` object by a `CPoint` object.

Is a Point Inside a Rectangle?

The `CRect` class has a member function `PtInRect` that tests a point to see whether it falls inside a rectangle. The second `OnLButtonDown()` parameter (point) is an object of class `CPoint` that represents the cursor location in the client area of the window. If you want to know whether that point is inside the `m_rectEllipse` rectangle, you can use `PtInRect()` in this way:

```
if (m_rectEllipse.PtInRect(point))
{
    // point is inside rectangle
}
```

As you'll soon see, however, this simple logic applies only if you're working in device coordinates (which you are at this stage).

The `CRect LPCRECT` Operator

If you read the Microsoft Foundation Class Reference carefully, you will notice that `CWnd::InvalidateRect` takes an `LPCRECT` parameter (a pointer to a `RECT` structure), not a `CRect` parameter. A `CRect` parameter is allowed because the `CRect` class defines an overloaded operator, `LPCRECT` that returns the address of a `CRect` object, which is equivalent to the address of a `RECT` object. Thus, the compiler converts `CRect` arguments to `LPCRECT` arguments when necessary. You call functions as though they had `CRect` reference parameters. The view member function code:

```
CRect rectClient;
GetClientRect(rectClient);
```

Retrieves the client rectangle coordinates and stores them in `rectClient`.

Is a Point Inside an Ellipse?

The MYMFC1 code determines whether the mouse hit is inside the rectangle. If you want to make a better test, you can find out whether the hit is inside the ellipse. To do this, you must construct an object of class `CRgn` that corresponds to the ellipse and then use the `PtInRegion()` function instead of `PtInRect()`. Here's the code:

```
CRgn rgn;
rgn.CreateEllipticRgnIndirect(m_rectEllipse);
if (rgn.PtInRegion(point)) {
// point is inside ellipse
}
```

Note that the `CreateEllipticRgnIndirect()` function is another function that takes an `LPCRECT` parameter. It builds a special region structure within Windows that represents an elliptical region inside a window. That structure is then attached to the C++ `CRgn` object in your program. The same type of structure can also represent a polygon.

The MYMFC1 Example

In the MYMFC1 example, an ellipse (which happens to be a circle) changes color when the user presses the left mouse button while the mouse cursor is inside the rectangle that bounds the ellipse. You'll use the view class data members to hold the view's state, and you'll use the `InvalidateRect()` function to cause the view to be redrawn. In the previous Module example, drawing in the window depended on only one function, `OnDraw()`. The MYMFC1 example requires three customized functions (including the constructor) and two data members. The complete `CMymfc1View` header and source code files are listed in Listing 1. The steps for creating the program are shown after the program listings. All changes to the original AppWizard and `OnLButtonDown()` ClassWizard output are in orange color.

MYMFC1VIEW.H

```
// mymfc1View.h : interface of the CMymfc1View class
//
/////////////////////////////////////////////////////////////////////////////

#if !defined(AFX_MYMFC1VIEW_H__B188BE41_6377_11D0_8FD4_00C04FC2A0C2
__INCLUDED_)
#define AFX_MYMFC1VIEW_H__B188BE41_6377_11D0_8FD4_00C04FC2A0C2
__INCLUDED_

#if _MFC_VER > 1000
#pragma once
#endif // _MFC_VER > 1000
class CMymfc1View : public CView
{
protected: // create from serialization only
    CMymfc1View();
    DECLARE_DYNCREATE(CMymfc1View)

// Attributes
public:
    CMymfc1Doc* GetDocument();

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CMymfc1View)
    public:
    virtual void OnDraw(CDC* pDC);  // overridden to draw this view
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    protected:
    virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
    virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
    virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CMymfc1View();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:
```

```
// Generated message map functions
protected:
    //{{AFX_MSG(CMymfc1View)
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
private:
    int m_nColor;
    CRect m_rectEllipse;
};
#ifndef _DEBUG  // debug version in mymfc1View.cpp
inline CMymfc1Doc* CMymfc1View::GetDocument()
   { return (CMymfc1Doc*)m_pDocument; }
#endif

/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations
//  immediately before the previous line.

#endif // !defined(AFX_MYMFC1VIEW_H__B188BE41_6377_11D0_8FD4_00C04FC2A0C2__INCLUDED_)
```

MYMFC1VIEW.CPP

```
// mymfc1View.cpp : implementation of the CMymfc1View class
//

#include "stdafx.h"
#include "mymfc1.h"

#include "mymfc1Doc.h"
#include "mymfc1View.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE
__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CMymfc1View

IMPLEMENT_DYNCREATE(CMymfc1View, CView)

BEGIN_MESSAGE_MAP(CMymfc1View, CView)
    //{{AFX_MSG_MAP(CMymfc1View)
    ON_WM_LBUTTONDOWN()
    //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////////////////////
// CMymfc1View construction/destruction

CMymfc1View::CMymfc1View() : m_rectEllipse(0, 0, 200, 200)
{
    m_nColor = GRAY_BRUSH;
}

CMymfc1View::~CMymfc1View()
{
}
```

```
BOOL CMymfc1View::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    // the CREATESTRUCT cs

    return CView::PreCreateWindow(cs);
}

/////////////////////////////////////////////////////////////////////////
// CMymfc1View drawing

void CMymfc1View::OnDraw(CDC* pDC)
{
    pDC->SelectStockObject(m_nColor);
    pDC->Ellipse(m_rectEllipse);
}

/////////////////////////////////////////////////////////////////////////
// CMymfc1View printing

BOOL CMymfc1View::OnPreparePrinting(CPrintInfo* pInfo)
{
    // default preparation
    return DoPreparePrinting(pInfo);
}

void CMymfc1View::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add extra initialization before printing
}

void CMymfc1View::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add cleanup after printing
}

/////////////////////////////////////////////////////////////////////////
// CMymfc1View diagnostics

#ifdef _DEBUG
void CMymfc1View::AssertValid() const
{
    CView::AssertValid();
}

void CMymfc1View::Dump(CDumpContext& dc) const
{
    CView::Dump(dc);
}

CMymfc1Doc* CMymfc1View::GetDocument() // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CMymfc1Doc)));
    return (CMymfc1Doc*)m_pDocument;
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////
// CMymfc1View message handlers

void CMymfc1View::OnLButtonDown(UINT nFlags, CPoint point)
{
    if (m_rectEllipse.PtInRect(point)) {
        if (m_nColor == GRAY_BRUSH) {
            m_nColor = WHITE_BRUSH;
```

```
        }
        else {
            m_nColor = GRAY_BRUSH;
        }
        InvalidateRect(m_rectEllipse);
    }
}
```

Listing 1.

Using ClassWizard with MYMFC1

Look at the following **mymfc1View.h** source code, the declaration part:

```
//{{AFX_MSG(CMymfc1View)
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
//}}AFX_MSG
```

Now look at the following **mymfc1View.cpp** source code, the implementation part:

```
//{{AFX_MSG_MAP(CMymfc1View)
ON_WM_LBUTTONDOWN()
//}}AFX_MSG_MAP
```

AppWizard generated the funny-looking comment lines for the benefit of ClassWizard. ClassWizard adds message handler prototypes between the `AFX_MSG` brackets and message map entries between the `AFX_MSG_MAP` brackets. In addition, ClassWizard generates a skeleton `OnLButtonDown()` member function in **mymfc1View.cpp**, complete with the correct parameter declarations and return type. You run AppWizard to generate the application only once, but you can run ClassWizard as many times as necessary, and you can edit the code at any time. You're safe as long as you don't alter what's inside the `AFX_MSG` and `AFX_MSG_MAP` brackets.

Using AppWizard and ClassWizard Together

The following steps show how you use AppWizard and ClassWizard together to create this application:

Run AppWizard to create MYMFC1 project. Use AppWizard to generate an **SDI project** named MYMFC1 in the \mfcproject\mymfc1 subdirectory or any subdirectory that you have selected during the new project creation. The options and the default class names are shown here.

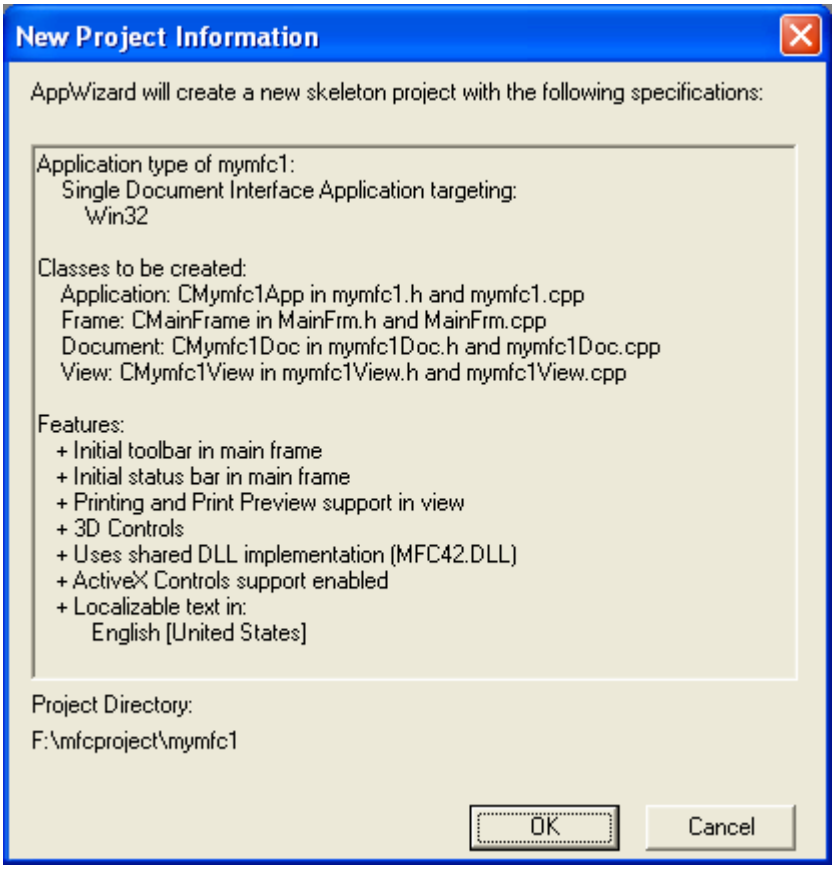

Figure 2: AppWizard MYMFC1 project summary.

Add the `m_rectEllipse` and `m_nColor` data members to `CMymfc1View`. With the Workspace window set to `ClassView`, right-click the `CMymfc1View` class, select **Add Member Variable…**, and then insert the following two data members:

```
private:
    CRect m_rectEllipse;
    int m_nColor;
```

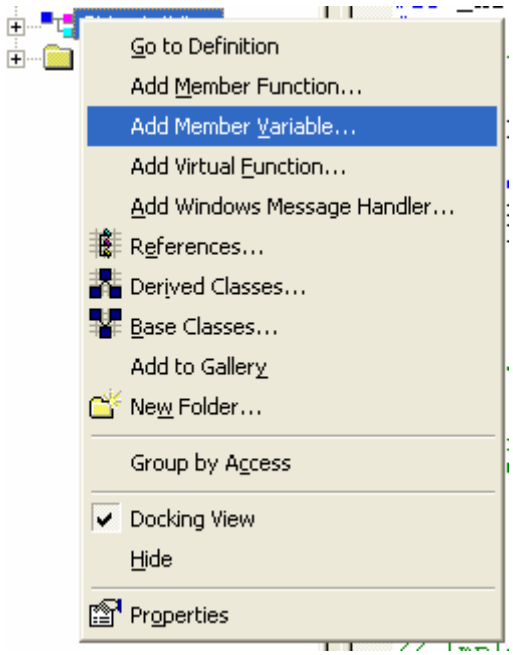

Figure 3: Adding Member Variable through the ClassView.

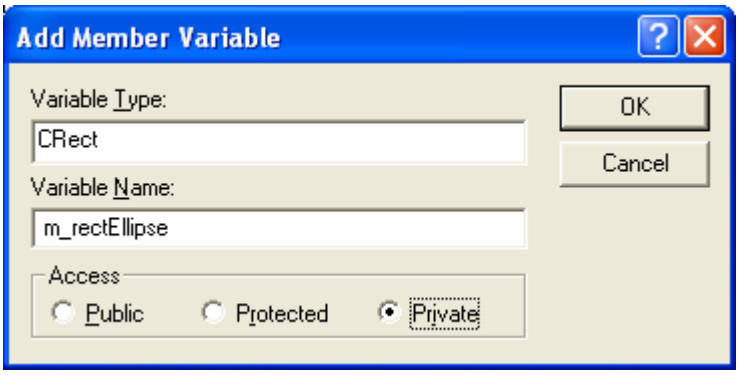

Figure 4: The variable type and name.

If you prefer, you could type the above code inside the class declaration in the file **mymfc1View.h**.

Use ClassWizard to add a `CMymfc1View` class message handler. Choose **ClassWizard** from the **View** menu of Visual C++, or right-click inside a source code window and choose ClassWizard from the context menu. Both shown below.

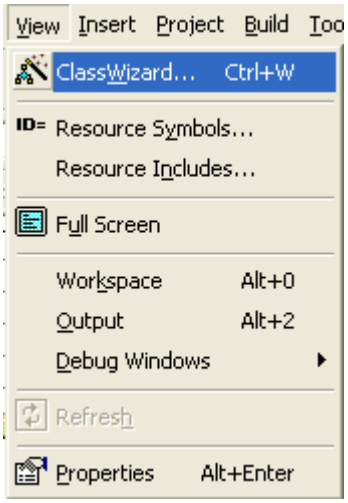

Figure 5: Invoking the ClassWizard through the **View** menu.

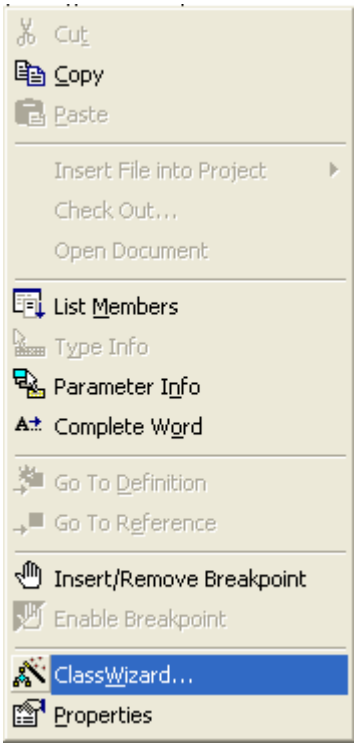

Figure 6: Invoking the ClassWizard through the context menu.

When the MFC ClassWizard dialog appears, be sure that the `CMymfc1View` class is selected, as shown in the illustration below. Now click on `CMymfc1View` at the top of the **Object IDs** list box, and then scroll down past the virtual functions in the **Messages** list box and double-click on `WM_LBUTTONDOWN`. The `OnLButtonDown()` function name should appear in the **Member Functions** list box, and the message name should be displayed in bold in the **Messages** list box. Here's the ClassWizard dialog box.

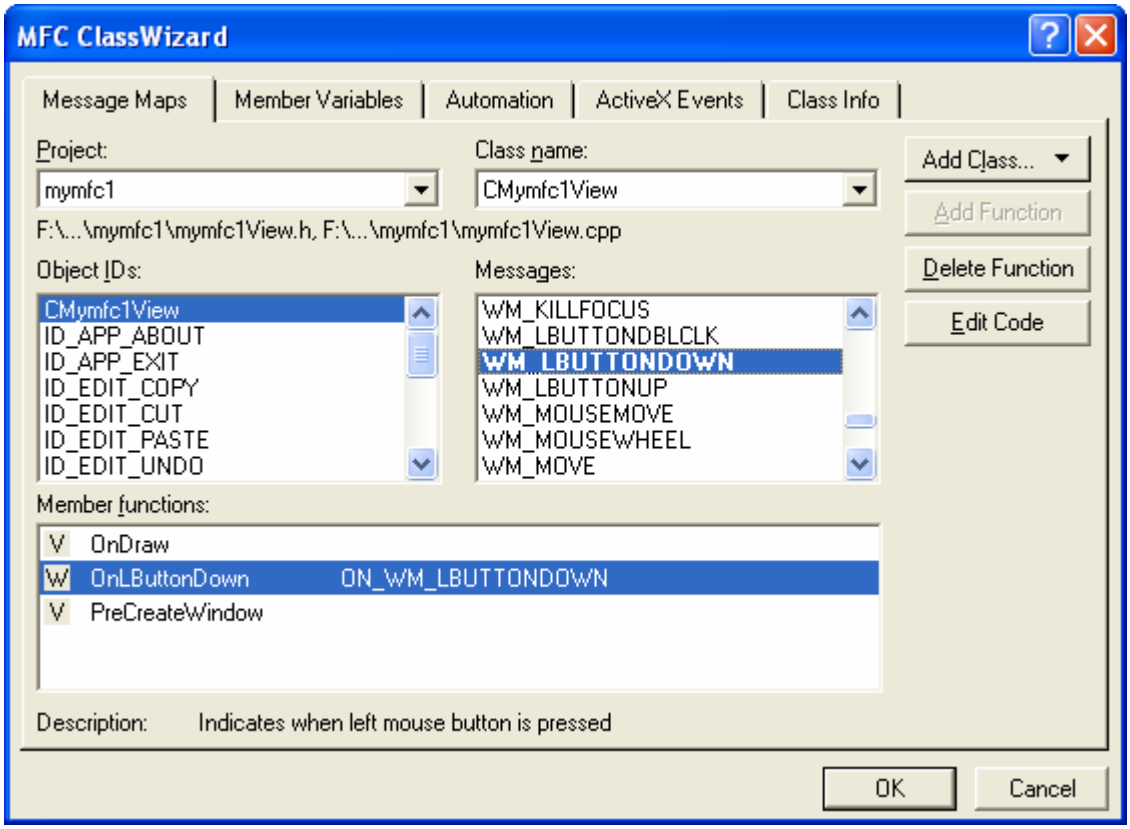

Figure 7: ClassWizard dialog.

Instead of using ClassWizard, you can map the function from the Visual C++ WizardBar.

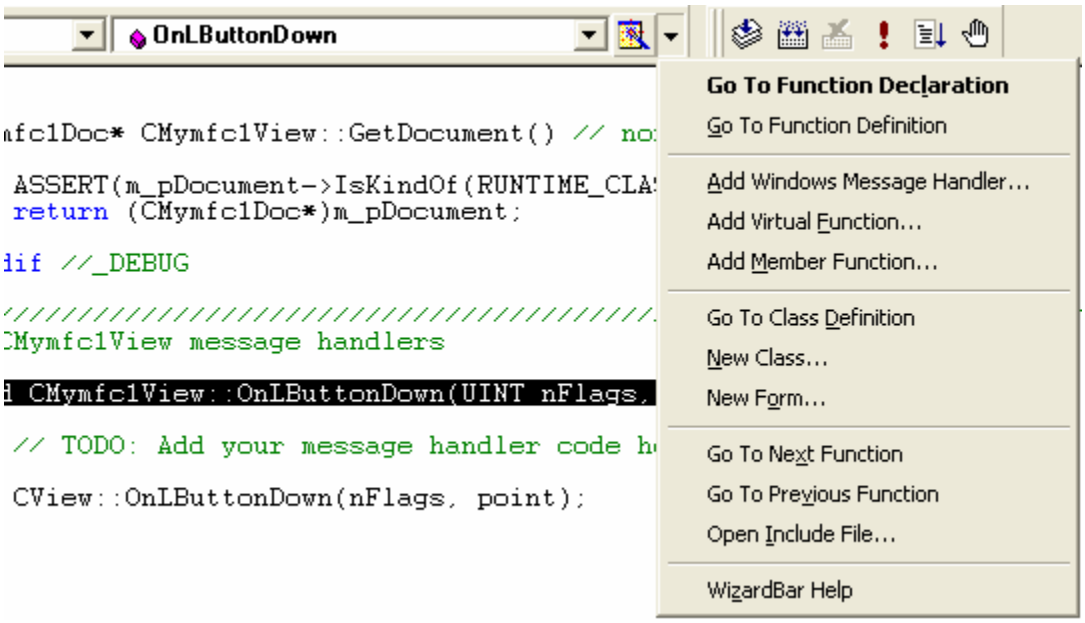

Figure 8: The WizardBar.

Edit the `OnLButtonDown()` code in **mymfc1View.cpp**. Click the **Edit Code** button in the ClassWizard dialog box. ClassWizard opens an edit window for **mymfc1View.cpp** in Visual C++ and positions the cursor on the newly generated `OnLButtonDown()` member function. The following replaces the previous code:

```
void CMymfc1View::OnLButtonDown(UINT nFlags, CPoint point)
{
    if (m_rectEllipse.PtInRect(point))
    {
        if (m_nColor == GRAY_BRUSH)
        {
            m_nColor = WHITE_BRUSH;
        }
        else
        {
            m_nColor = GRAY_BRUSH;
        }
        InvalidateRect(m_rectEllipse);
    }
}
```

```
void CMymfc1View::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default

    if (m_rectEllipse.PtInRect(point))
    {
        if (m_nColor == GRAY_BRUSH)
        {
            m_nColor = WHITE_BRUSH;
        }
        else
        {
            m_nColor = GRAY_BRUSH;
        }
        InvalidateRect(m_rectEllipse);
    }

}
```

Listing 2: C++ code segment.

Edit the constructor and the `OnDraw()` function in **mymfc1View.cpp**. The following replaces the previous code:

```
CMymfc1View::CMymfc1View() : m_rectEllipse(0, 0, 200, 200)
{
    m_nColor = GRAY_BRUSH;
}
.
.
.
void CMymfc1View::OnDraw(CDC* pDC)
{
    pDC->SelectStockObject(m_nColor);
    pDC->Ellipse(m_rectEllipse);
}
```

```
CMymfc1View::CMymfc1View() : m_rectEllipse(0, 0, 200, 200)
{
    // TODO: add construction code here
    m_nColor = GRAY_BRUSH;

}
```

```
void CMymfc1View::OnDraw(CDC* pDC)
{
    // TODO: add draw code for native data here
    pDC->SelectStockObject(m_nColor);
    pDC->Ellipse(m_rectEllipse);
}
```

Listing 3: C++ code segment.

Build and run the MYMFC1 program. Choose **Build Mymfc1.exe** from the **Build** menu, or, on the **Build** toolbar, click the  button. Then choose **Execute Mymfc1.exe** from the **Build** menu or, on the **Build** toolbar, click the button. The resulting program responds to presses of the left mouse button by changing the color of the circle in the view window. Don't press the mouse's left button too quickly in succession; Windows interprets this as a double click rather than two single clicks.

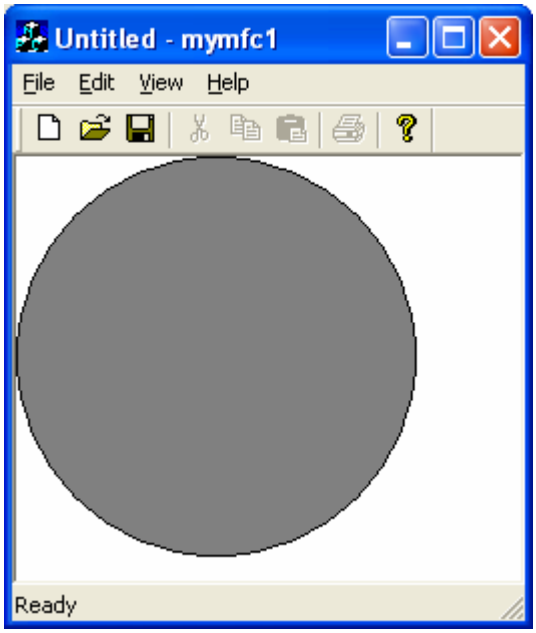

Figure 9: MYMFC1 program output.

Mapping Modes

Up to now, your drawing units have been display **pixels**, also known as **device coordinates**. The MYMFC1 drawing units are pixels because the device context has the default mapping mode, **`MM_TEXT`**, assigned to it. The statement:

```
pDC->Rectangle(CRect(0, 0, 200, 200));
```

draws a square of 200-by-200 pixels, with its top-left corner at the top left of the window's client area. The positive y values increase as you move down the window. This square would look smaller on a high-resolution display of 1024-by-768 pixels than it would look on a standard VGA display that is 640-by-480 pixels, and it would look tiny if printed on a laser printer with 600-dpi resolution. Try MYMFC1's **Print Preview** feature to see for yourself.

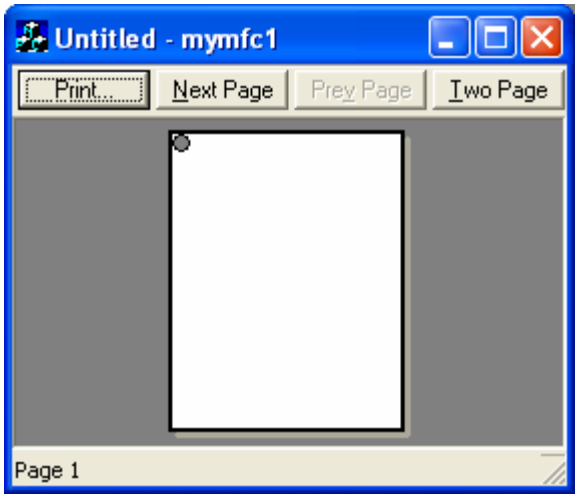

Figure 10: Print preview of the MYMFC1 project.

What if you want the square to be 4-by-4 centimeters (cm), regardless of the display device? Windows provides a number of other mapping modes, or coordinate systems that can be associated with the device context. Coordinates in the current mapping mode are called logical coordinates. If you assign the `MM_HIMETRIC` mapping mode, for example, a logical unit is **1/100** millimeter (mm) instead of 1 pixel. In the `MM_HIMETRIC` mapping mode, the y axis runs in the opposite direction to that in the `MM_TEXT` mode: y values decrease as you move down. Thus, a 4-by-4-cm square is drawn in logical coordinates this way:

```
pDC->Rectangle(CRect(0, 0, 4000, -4000));
```

Looks easy, doesn't it? Well, it isn't, because you can't work only in logical coordinates. Your program is always switching between device coordinates and logical coordinates, and you need to know when to convert between them. This section gives you a few rules that could make your programming life easier. First you need to know what mapping modes Windows gives you.

The `MM_TEXT` Mapping Mode

At first glance, `MM_TEXT` appears to be no mapping mode at all, but rather another name for device coordinates. Almost. In `MM_TEXT`, coordinates map to pixels, values of x increase as you move right, and values of y increase as you move down, but you're allowed to change the origin through calls to the CDC functions `SetViewportOrg()` and `SetWindowOrg()`. Here's some code that sets the window origin to (100, 100) in logical coordinate space and then draws a 200-by-200-pixel square offset by (100, 100). An illustration of the output is shown in the following Figure. The logical point (100, 100) maps to the device point (0, 0). A scrolling window uses this kind of transformation.

```
void CMyView::OnDraw(CDC* pDC)
{
    pDC->SetMapMode(MM_TEXT);
    pDC->SetWindowOrg(CPoint(100, 100));
    pDC->Rectangle(CRect(100, 100, 300, 300));
}
```

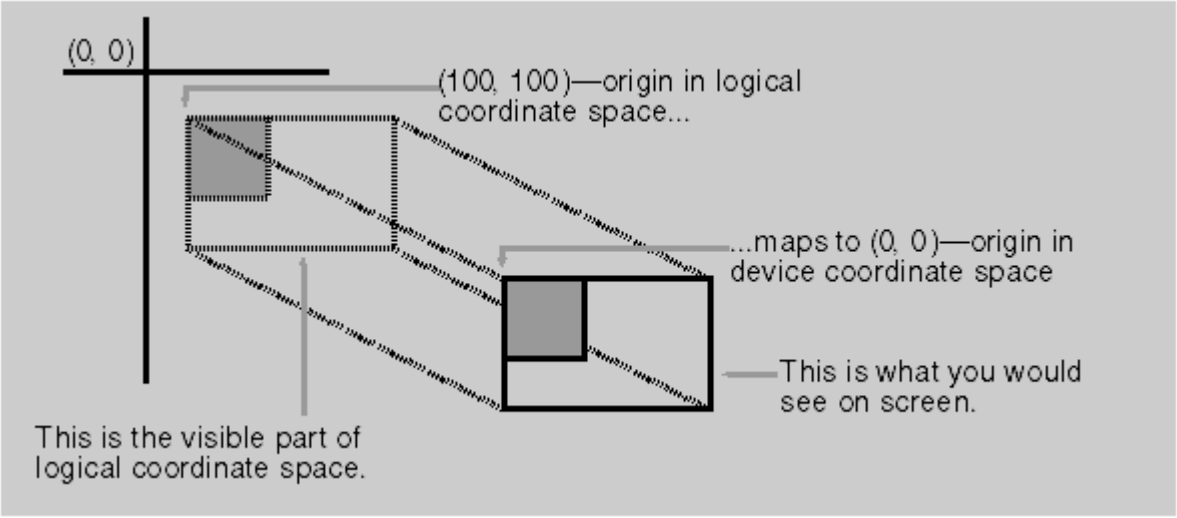

Figure 11: A square drawn after the origin has been moved to (100, 100).

The Fixed-Scale Mapping Modes

One important group of Windows mapping modes provides fixed scaling. You have already seen that, in the `MM_HIMETRIC` mapping mode, x values increase as you move right and y values decrease as you move down. All **fixed mapping modes** follow this convention, and you can't change it. The only difference among the fixed mapping modes is the actual scale factor, listed in the table shown here.

Mapping Mode	Logical Unit
`MM_LOENGLISH`	0.01 inch
`MM_HIENGLISH`	0.001 inch
`MM_LOMETRIC`	0.1 mm
`MM_HIMETRIC`	0.01 mm
`MM_TWIPS`	1/1440 inch

Table 2.

The last mapping mode, `MM_TWIPS`, is most often used with printers. One twip unit is 1/20 point. A point is a type measurement unit. In Windows it equals exactly 1/72 inch. If the mapping mode is `MM_TWIPS` and you want, for example, 12-point type, set the character height to 12 × 20, or 240, twips.

The Variable-Scale Mapping Modes

Windows provides two mapping modes, **MM_ISOTROPIC** and **MM_ANISOTROPIC** that allow you to change the scale factor as well as the origin. With these mapping modes, your drawing can change size as the user changes the size of the window. Also, if you invert the scale of one axis, you can "flip" an image about the other axis and you can define your own arbitrary fixed-scale factors. With the `MM_ISOTROPIC` mode, a 1:1 aspect ratio is always preserved. In other words, a circle is always a circle as the scale factor changes. With the `MM_ANISOTROPIC` mode, the x and y scale factors can change independently. Circles can be squished into ellipses. Here's an `OnDraw()` function that draws an ellipse that fits exactly in its window:

```
void CMyView::OnDraw(CDC* pDC)
{
    CRect rectClient;

    GetClientRect(rectClient);
    pDC->SetMapMode(MM_ANISOTROPIC);
    pDC->SetWindowExt(1000, 1000);
    pDC->SetViewportExt(rectClient.right, -rectClient.bottom);
    pDC->SetViewportOrg(rectClient.right / 2, rectClient.bottom / 2);
```

```
        pDC->Ellipse(CRect(-500, -500, 500, 500));
    }
```

The functions `SetWindowExt()` and `SetViewportExt()` work together to set the scale, based on the window's current client rectangle returned by the `GetClientRect()` function. The resulting window size is exactly 1000-by-1000 logical units. The `SetViewportOrg()` function sets the origin to the center of the window. Thus, a centered ellipse with a radius of 500 logical units fills the window exactly, as illustrated in the following Figure.

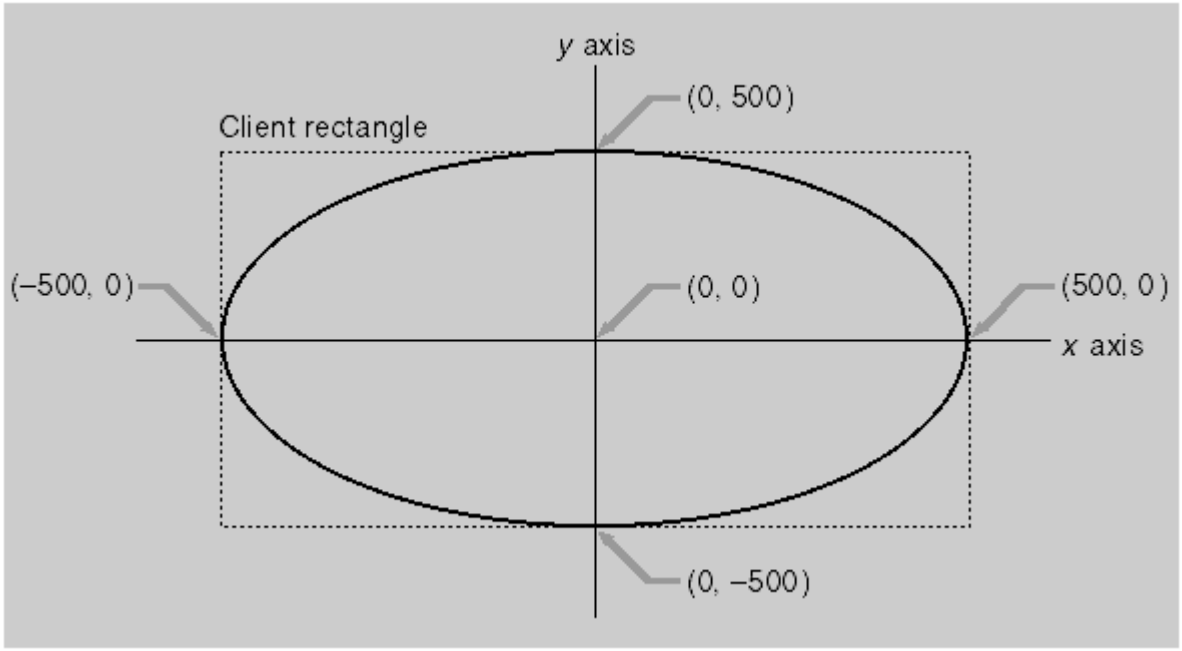

Figure 12: A centered ellipse drawn in the `MM_ANISOTROPIC` mapping mode.

Here are the formulas for converting logical units to device units:

x scale factor = x viewport extent / x window extent
y scale factor = y viewport extent / y window extent
device x = logical x × x scale factor + x origin offset
device y = logical y × y scale factor + y origin offset

Suppose the window is 448 pixels wide (`rectClient.right`). The right edge of the ellipse's client rectangle is 500 logical units from the origin. Then:

The x scale factor is 448/1000, and
The x origin offset is 448/2 device units.

If you use the formulas shown previously, the right edge of the ellipse's client rectangle comes out to 448 device units, the right edge of the window. The `x` scale factor is expressed as a ratio (viewport extent/window extent) because Windows device coordinates are integers, not floating-point values. The extent values are meaningless by themselves. If you substitute `MM_ISOTROPIC` for `MM_ANISOTROPIC` in the preceding example, the "ellipse" is always a circle, as shown in the following Figure. It expands to fit the smallest dimension of the window rectangle.

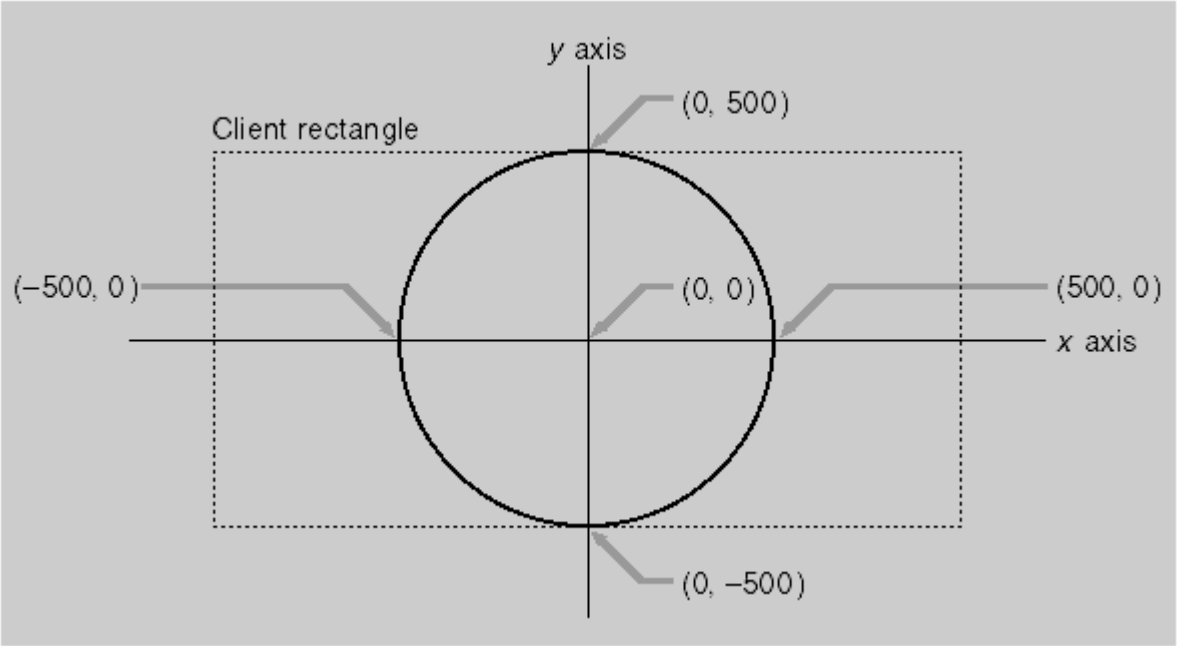

Figure 13: A centered ellipse drawn in the `MM_ISOTROPIC` mapping mode.

Coordinate Conversion

Once you set the mapping mode (plus the origin) of a device context, you can use logical coordinate parameters for most `CDC` member functions. If you get the mouse cursor coordinates from a Windows mouse message (the point parameter in `OnLButtonDown()`), for example, you're dealing with device coordinates. Many other MFC functions, particularly the member functions of class `CRect`, work correctly only with device coordinates. The `CRect` arithmetic functions use the underlying Win32 `RECT` arithmetic functions, which assume that right is greater than left and bottom is greater than top. A rectangle (0, 0, 1000, -1000) in `MM_HIMETRIC` coordinates, for example, has bottom less than top and cannot be processed by functions such as `CRect::PtInRect` unless your program first calls `CRect::NormalizeRect`, which changes the rectangle's data members to (0, -1000, 1000, 0).
Furthermore, you're likely to need a third set of coordinates that we will call physical coordinates. Why do you need another set? Suppose you're using the `MM_LOENGLISH` mapping mode in which a logical unit is 0.01 inch, but an inch on the screen represents a foot (12 inches) in the real world. Now suppose the user works in inches and decimal fractions. A measurement of 26.75 inches translates to 223 logical units, which must be ultimately translated to device coordinates. You will want to store the physical coordinates as either floating-point numbers or scaled long integers to avoid rounding-off errors. For the physical-to-logical translation you're on your own, but the Windows GDI takes care of the logical-to-device translation for you. The CDC functions `LPtoDP()` and `DPtoLP()` translate between the two systems, assuming the device context mapping mode and associated parameters have already been set. Your job is to decide when to use each system. Here are a few rules of thumb:

- Assume that the `CDC` member functions take logical coordinate parameters.
- Assume that the `CWnd` member functions take device coordinate parameters.
- Do all hit-test operations in device coordinates. Define regions in device coordinates. Functions such as `CRect::PtInRect` work best with device coordinates.
- Store long-term values in logical or physical coordinates. If you store a point in device coordinates and the user scrolls through a window, that point is no longer valid.

Suppose you need to know whether the mouse cursor is inside a rectangle when the user presses the left mouse button. The code is shown here.

```
// m_rect is CRect data member of the derived view class with
//  MM_LOENGLISH logical coordinates

void CMyView::OnLButtonDown(UINT nFlags, CPoint point)
```

```
{
    CRect rect = m_rect; // rect is a temporary copy of m_rect.
    CClientDC dc(this);  // This is how we get a device context
                         //  for SetMapMode and LPtoDP
                         //  -- more in next module
    dc.SetMapMode(MM_LOENGLISH);
    dc.LPtoDP(rect);     // rect is now in device coordinates
    if (rect.PtInRect(point))
    {
        TRACE("Mouse cursor is inside the rectangle.\n");
    }
}
```

The `TRACE` is a macro. As you'll soon see, it's better to set the mapping mode in the virtual `CView` function `OnPrepareDC()` instead of in the `OnDraw()` function.

The MYMFC2 Example: Converting to the `MM_HIMETRIC` Mapping Mode

MYMFC2 is MYMFC1 converted to `MM_HIMETRIC` coordinates. Like MYMFC1, MYMFC2 performs a hit-test so that the ellipse changes color only when you click inside the bounding rectangle. Use ClassWizard to override the virtual `OnPrepareDC()` function. ClassWizard can override virtual functions for selected MFC base classes, including `CView`. It generates the correct function prototype in the class's header file and a skeleton function in the CPP file. Select the class name `CMymfc1View` in the **Object IDs** list, and then double-click on the `OnPrepareDC()` function in the **Messages** list.

Edit the function as shown here:

Using the ClassWizard to override the virtual `OnPrepareDC()` function.

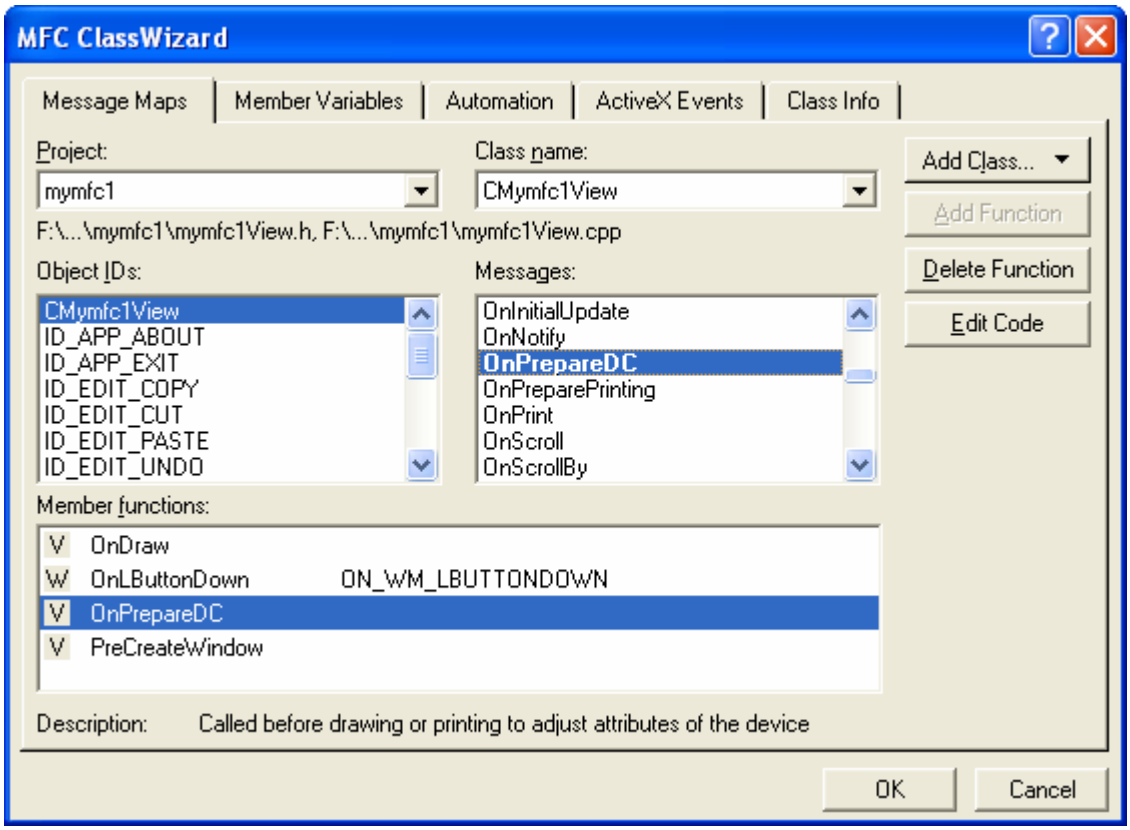

Figure 14: Using ClassWizard to override the virtual `OnPrepareDC()` function.

Enter the `OnPrepareDC()` code.

```
void CMymfc1View::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
    pDC->SetMapMode(MM_HIMETRIC);
    CView::OnPrepareDC(pDC, pInfo);
}
```

```
void CMymfc1View::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
    // TODO: Add your specialized code here and/or call the base class
    pDC->SetMapMode(MM_HIMETRIC);
    CView::OnPrepareDC(pDC, pInfo);
}
```

Listing 4.

The application framework calls the virtual `OnPrepareDC()` function just before it calls `OnDraw()`.
Edit the view class constructor. You must change the coordinate values for the ellipse rectangle. That rectangle is now 4-by-4 centimeters instead of 200-by-200 pixels. Note that the y value must be negative; otherwise, the ellipse will be drawn on the "virtual screen" right above your monitor! Change the values as shown here:

```
CMymfc1View::CMymfc1View() : m_rectEllipse(0, 0, 4000, -4000)
{
    m_nColor = GRAY_BRUSH;
}
```

```
// CMymfc1View construction/destruction

CMymfc1View::CMymfc1View() : m_rectEllipse(0, 0, 4000, -4000)
{
    // TODO: add construction code here
    m_nColor = GRAY_BRUSH;
}
```

Listing 5.

Edit the `OnLButtonDown()` function. This function must now convert the ellipse rectangle to device coordinates in order to do the hit-test. Change the function as shown in the following code:

```
void CMymfc1View::OnLButtonDown(UINT nFlags, CPoint point)
{
    CClientDC dc(this);
    OnPrepareDC(&dc);
    CRect rectDevice = m_rectEllipse;
    dc.LPtoDP(rectDevice);
    if (rectDevice.PtInRect(point)) {
        if (m_nColor == GRAY_BRUSH) {
            m_nColor = WHITE_BRUSH;
        }
        else {
            m_nColor = GRAY_BRUSH;
        }
        InvalidateRect(rectDevice);
    }
}
```

```
void CMymfc1View::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    CClientDC dc(this);
    OnPrepareDC(&dc);
    CRect rectDevice = m_rectEllipse;
    dc.LPtoDP(rectDevice);
    if (rectDevice.PtInRect(point)) {
        if (m_nColor == GRAY_BRUSH) {
            m_nColor = WHITE_BRUSH;
        }
        else {
            m_nColor = GRAY_BRUSH;
        }
        InvalidateRect(rectDevice);
    }
}
```

Listing 6.

Build and run the MYMFC2 program. The output should look similar to the output from MYMFC1, except that the ellipse size will be different.

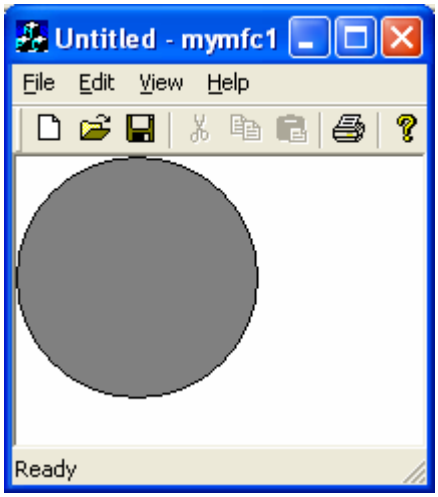

Figure 15: MYMFC2 program output.

If you try using **Print Preview** again, the ellipse should appear much larger than it did in MYMFC1.

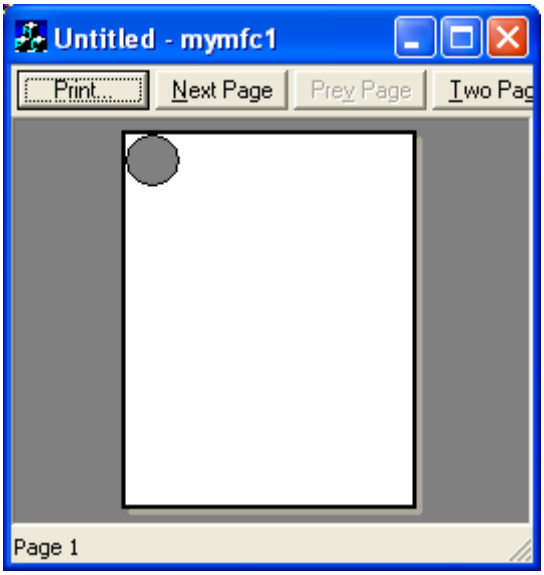

Figure 16: Better print preview of the MYMFC1.

A Scrolling View Window

As the lack of scroll bars in MYMFC1 and MYMFC2 indicates, the MFC CView class, the base class of `CMymfc2View`, doesn't directly support scrolling. Another MFC library class, `CScrollView`, does support scrolling. `CScrollView` is derived from `CView`. We'll create a new program, MYMFC3 that uses `CScrollView` in place of `CView`. All the coordinate conversion code you added in MYMFC2 sets you up for scrolling. The `CScrollView` class supports scrolling from the scroll bars but not from the keyboard. It's easy enough to add keyboard scrolling, so we'll do it.

A Window Is Larger than What You See

If you use the mouse to shrink the size of an ordinary window, the contents of the window remain anchored at the top left of the window, and items at the bottom and/or on the right of the window disappear. When you expand the window, the items reappear. You can correctly conclude that a window is larger than the viewport that you see on the screen. The viewport doesn't have to be anchored at the top left of the window area, however. Through the use of the `CWnd` functions `ScrollWindow()` and `SetWindowOrg()`, the `CScrollView` class allows you to move the viewport anywhere within the window, including areas above and to the left of the origin.

Scroll Bars

Microsoft Windows makes it easy to display scroll bars at the edges of a window, but Windows by itself doesn't make any attempt to connect those scroll bars to their window. That's where the `CScrollView` class fits in. `CScrollView` member functions process the `WM_HSCROLL` and `WM_VSCROLL` messages sent by the scroll bars to the view. Those functions move the viewport within the window and do all the necessary housekeeping.

Scrolling Alternatives

The `CScrollView` class supports a particular kind of scrolling that involves one big window and a small viewport. Each item is assigned a unique position in this big window. If, for example, you have 10,000 address lines to display, instead of having a window 10,000 lines long, you probably want a smaller window with scrolling logic that selects only as many lines as the screen can display. In that case, you should write your own scrolling view class derived from `CView`.
Microsoft Windows NT uses 32-bit numbers for logical coordinates, so your logical coordinate space is almost unlimited. Microsoft Windows 95, however, still has some 16-bit components, so it uses 16-bit numbers for logical coordinates, limiting values to the range -32,768 to 32,767. Scroll bars send messages with 16-bit values in both

operating systems. With these facts in mind, you probably want to write code to the lowest common denominator, which is Windows 95.

The `OnInitialUpdate()` Function

You'll be seeing more of the `OnInitialUpdate()` function when you study the document-view architecture. The virtual `OnInitialUpdate()` function is important here because it is the first function called by the framework after your view window is fully created. The framework calls `OnInitialUpdate()` before it calls `OnDraw()` for the first time, so `OnInitialUpdate()` is the natural place for setting the logical size and mapping mode for a scrolling view. You set these parameters with a call to the `CScrollView::SetScrollSizes` function.

Accepting Keyboard Input

Keyboard input is really a two-step process. Windows sends `WM_KEYDOWN` and `WM_KEYUP` messages, with virtual key codes, to a window, but before they get to the window they are translated. If an ANSI character is typed (resulting in a `WM_KEYDOWN` message), the translation function checks the keyboard shift status and then sends a `WM_CHAR` message with the proper code, either uppercase or lowercase. Cursor keys and function keys don't have codes, so there's no translation to do. The window gets only the `WM_KEYDOWN` and `WM_KEYUP` messages. You can use ClassWizard to map all these messages to your view. If you're expecting characters, map `WM_CHAR`; if you're expecting other keystrokes, map `WM_KEYDOWN`. The MFC library neatly supplies the character code or virtual key code as a handler function parameter.

The MYMFC3 Example - Scrolling

The goal of MYMFC3 is to make a logical window 20 centimeters wide by 30 centimeters high. The program draws the same ellipse that it drew in the MYMFC2 project. You could edit the MYMFC2 source files to convert the `CView` base class to a `CScrollView` base class, but it's easier to start over with AppWizard. AppWizard generates the `OnInitialUpdate()` override function for you. Here are the steps:

Run AppWizard to create MYMFC3.Use AppWizard to generate a program named MYMFC3 in the mfcproject\mymfc3 subdirectory. In AppWizard Step 6, set the `CMymfc3View` base class to `CScrollView`, as shown here.

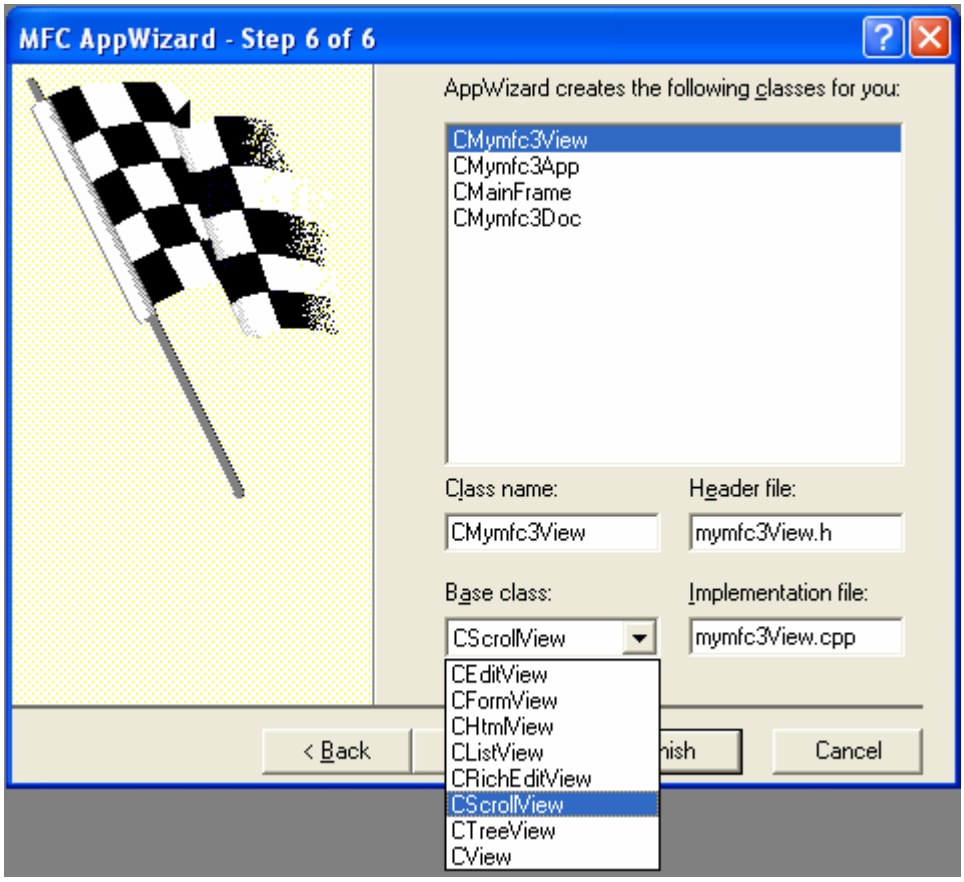

Figure 17: AppWizard Step 6, setting the `CMymfc3View` base class to `CScrollView`.

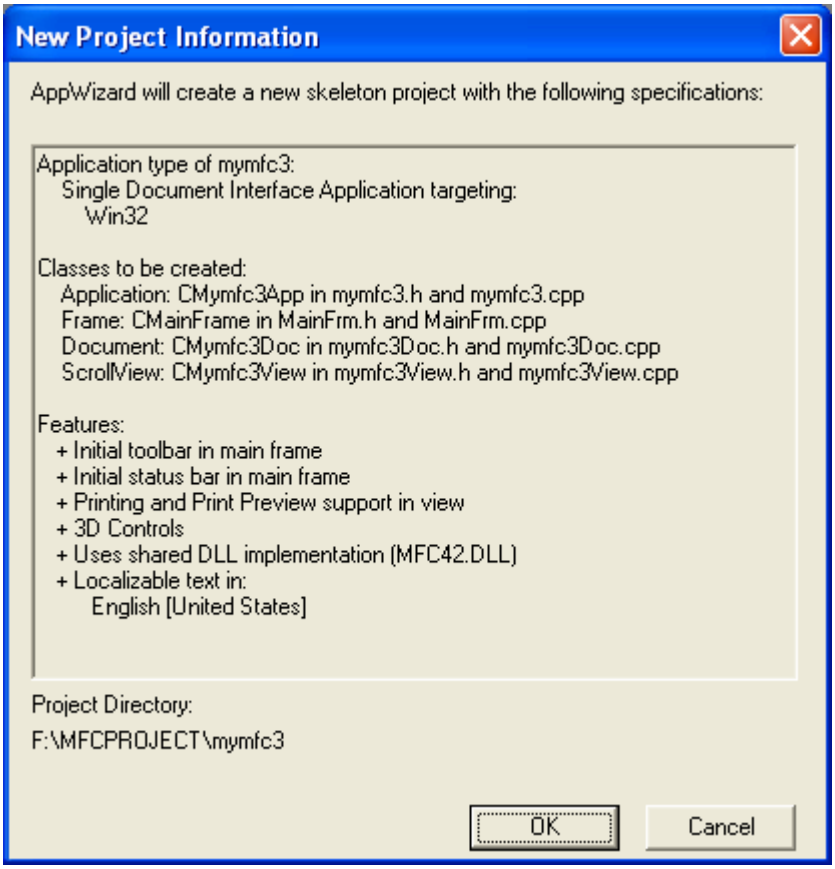

Figure 18: MYMFC3 project summary.

Add the `m_rectEllipse` and `m_nColor` data members in mymfc3View.h. Insert the following code by right-clicking the `CMymfc3View` class in the Workspace window or by typing inside the **CMymfc3View.h** class declaration:

```
private:
    CRect m_rectEllipse;
    int m_nColor;
```

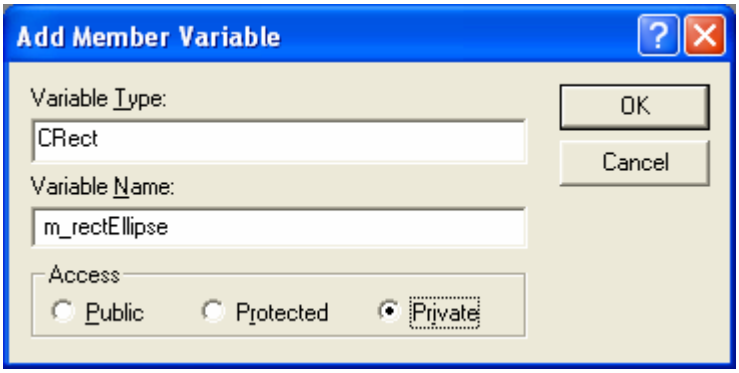

Figure 19: Adding private member variable.

These are the same data members that were added in the MYMFC1 and MYMFC2 projects.

Modify the AppWizard-generated `OnInitialUpdate()` function. Edit `OnInitialUpdate()` in **mymfc3View.cpp** as shown here:

```
void CMymfc3View::OnInitialUpdate()
```

```
{
    CScrollView::OnInitialUpdate();
    // 20 by 30 cm
    CSize sizeTotal(20000, 30000);
    CSize sizePage(sizeTotal.cx / 2, sizeTotal.cy / 2);
    CSize sizeLine(sizeTotal.cx / 50, sizeTotal.cy / 50);
    SetScrollSizes(MM_HIMETRIC, sizeTotal, sizePage, sizeLine);
}
```

```
void CMymfc3View::OnInitialUpdate()
{
    CScrollView::OnInitialUpdate();
    CSize sizeTotal(20000, 30000); // 20 by 30 cm
    CSize sizePage(sizeTotal.cx / 2, sizeTotal.cy / 2);
    CSize sizeLine(sizeTotal.cx / 50, sizeTotal.cy / 50);
    SetScrollSizes(MM_HIMETRIC, sizeTotal, sizePage, sizeLine);
}
```

Listing 7.

Use ClassWizard to add a message handler for the `WM_KEYDOWN` message.

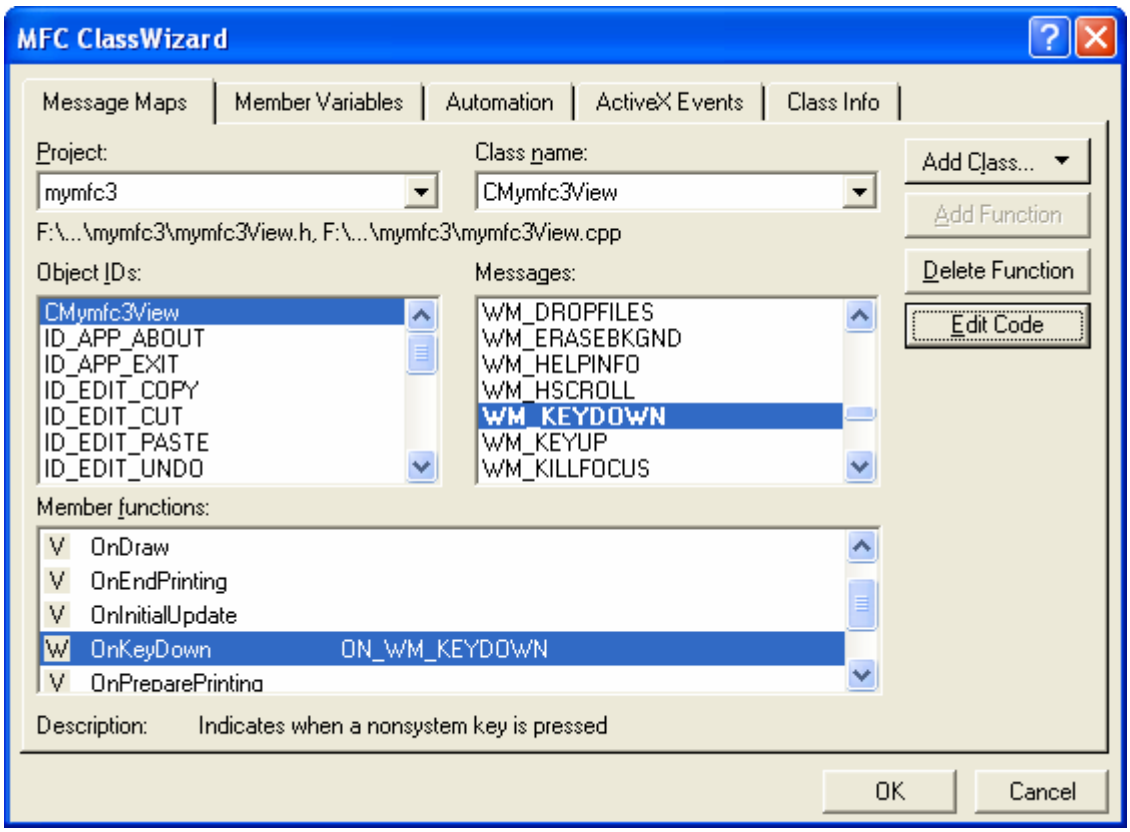

Figure 20: Using ClassWizard to add a message handler for the `WM_KEYDOWN` message.

ClassWizard generates the member function `OnKeyDown()` along with the necessary message map entries and prototypes. Edit the code as follows:

```
void CMymfc3View::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    switch (nChar) {
    case VK_HOME:
        OnVScroll(SB_TOP, 0, NULL);
        OnHScroll(SB_LEFT, 0, NULL);
```

```
            break;
        case VK_END:
            OnVScroll(SB_BOTTOM, 0, NULL);
            OnHScroll(SB_RIGHT, 0, NULL);
            break;
        case VK_UP:
            OnVScroll(SB_LINEUP, 0, NULL);
            break;
        case VK_DOWN:
            OnVScroll(SB_LINEDOWN, 0, NULL);
            break;
        case VK_PRIOR:
            OnVScroll(SB_PAGEUP, 0, NULL);
            break;
        case VK_NEXT:
            OnVScroll(SB_PAGEDOWN, 0, NULL);
            break;
        case VK_LEFT:
            OnHScroll(SB_LINELEFT, 0, NULL);
            break;
        case VK_RIGHT:
            OnHScroll(SB_LINERIGHT, 0, NULL);
            break;
        default:
            break;
        }
    }
```

```
void CMymfc3View::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    // TODO: Add your message handler code here and/or call default
switch (nChar) {
    case VK_HOME:
        OnVScroll(SB_TOP, 0, NULL);
        OnHScroll(SB_LEFT, 0, NULL);
        break;
    case VK_END:
        OnVScroll(SB_BOTTOM, 0, NULL);
        OnHScroll(SB_RIGHT, 0, NULL);
        break;
    case VK_UP:
        OnVScroll(SB_LINEUP, 0, NULL);
        break;
    case VK_DOWN:
        OnVScroll(SB_LINEDOWN, 0, NULL);
        break;
    case VK_PRIOR:
        OnVScroll(SB_PAGEUP, 0, NULL);
        break;
    case VK_NEXT:
        OnVScroll(SB_PAGEDOWN, 0, NULL);
        break;
    case VK_LEFT:
        OnHScroll(SB_LINELEFT, 0, NULL);
        break;
    case VK_RIGHT:
        OnHScroll(SB_LINERIGHT, 0, NULL);
        break;
    default:
        break;
    }
}
```

Listing 8.

Edit the constructor and the `OnDraw()` function. Change the AppWizard-generated constructor and the `OnDraw()` function in **mymfc3View.cpp** as follows:

```
CMymfc3View::CMymfc3View() : m_rectEllipse(0, 0, 4000, -4000)
{
    m_nColor = GRAY_BRUSH;
}
.
.
.
void CMymfc3View::OnDraw(CDC* pDC)
{
    pDC->SelectStockObject(m_nColor);
    pDC->Ellipse(m_rectEllipse);
}
```

```
// CMymfc3View construction/destruction

CMymfc3View::CMymfc3View() : m_rectEllipse(0, 0, 4000, -4000)
{
    // TODO: add construction code here
    m_nColor = GRAY_BRUSH;
}
```

Listing 9.

```
void CMymfc3View::OnDraw(CDC* pDC)
{
    pDC->SelectStockObject(m_nColor);
    pDC->Ellipse(m_rectEllipse);
}
```

Listing 10.

These functions are identical to those used in the MYMFC1 and MYMFC2 projects. Map the `WM_LBUTTONDOWN` message and edit the message handler by clicking the **Edit Code** button.

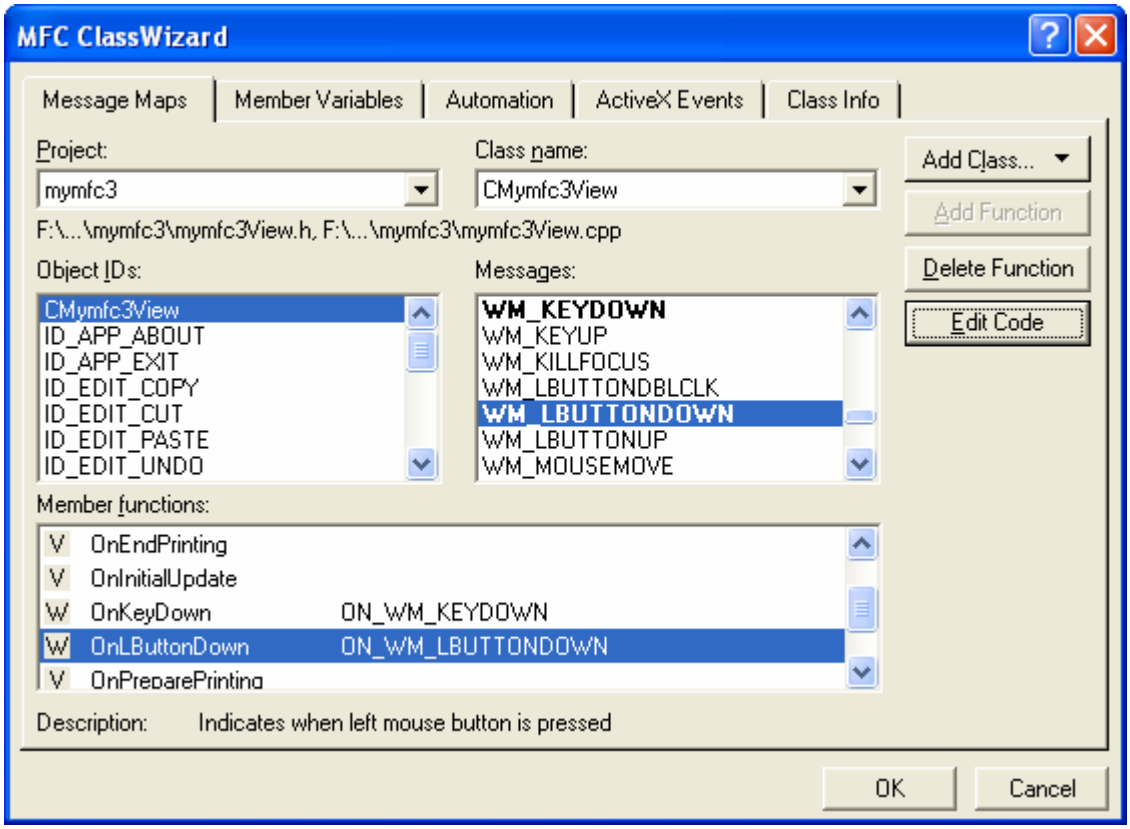

Figure 21: Using ClassWizard to map the `WM_LBUTTONDOWN` message when left mouse button is clicked.

Make the following changes to the ClassWizard-generated code:

```
void CMymfc3View::OnLButtonDown(UINT nFlags, CPoint point)
{
    CClientDC dc(this);
    OnPrepareDC(&dc);
    CRect rectDevice = m_rectEllipse;
    dc.LPtoDP(rectDevice);
    if (rectDevice.PtInRect(point)) {
        if (m_nColor == GRAY_BRUSH) {
            m_nColor = WHITE_BRUSH;
        }
        else {
            m_nColor = GRAY_BRUSH;
        }
        InvalidateRect(rectDevice);
    }
}
```

```
void CMymfc3View::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
        CClientDC dc(this);
    OnPrepareDC(&dc);
    CRect rectDevice = m_rectEllipse;
    dc.LPtoDP(rectDevice);
    if (rectDevice.PtInRect(point)) {
        if (m_nColor == GRAY_BRUSH) {
            m_nColor = WHITE_BRUSH;
        }
        else {
            m_nColor = GRAY_BRUSH;
        }
        InvalidateRect(rectDevice);
    }
}
```

Listing 11.

This function is identical to the `OnLButtonDown()` handler in the MYMFC2 project. It calls `OnPrepareDC()` as before, but there is something different. The `CMymfc2View` class doesn't have an overridden `OnPrepareDC()` function, so the call goes to `CScrollView::OnPrepareDC`. That function sets the mapping mode based on the first parameter to `SetScrollSizes()`, and it sets the window origin based on the current scroll position. Even if your scroll view used the `MM_TEXT` mapping mode, you'd still need the coordinate conversion logic to adjust for the origin offset.

Build and run the MYMFC3 program. Check to be sure the mouse hit logic is working even if the circle is scrolled partially out of the window (use the horizontal and vertical scroll bar). Also check the keyboard logic. The output should look like this.

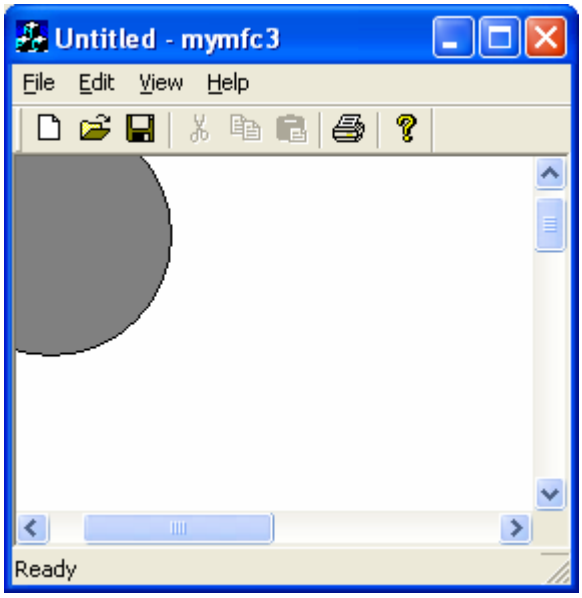

Figure 22: MYMFC3 program output with horizontal and vertical scroll bar.

Other Windows Messages

The MFC library directly supports hundreds of Windows message-handling functions. In addition, you can define your own messages. You will see plenty of message-handling examples in later chapters, including handlers for menu items, child window controls, and so forth. In the meantime, five special Windows messages deserve special attention: `WM_CREATE`, `WM_CLOSE`, `WM_QUERYENDSESSION`, `WM_DESTROY`, and `WM_NCDESTROY`.

The `WM_CREATE` Message

This is the first message that Windows sends to a view. It is sent when the window's Create function is called by the framework, so the window creation is not finished and the window is not visible. Therefore, your `OnCreate()` handler cannot call Windows functions that depend on the window being completely alive. You can call such functions in an overridden `OnInitialUpdate()` function, but you must be aware that in an SDI application `OnInitialUpdate()` can be called more than once in a view's lifetime.

The `WM_CLOSE` Message

Windows sends the `WM_CLOSE` message when the user closes a window from the system menu and when a parent window is closed. If you implement the `OnClose()` message map function in your derived view class, you can control the closing process. If, for example, you need to prompt the user to save changes to a file, you do it in `OnClose()`. Only when you have determined that it is safe to close the window do you call the base class `OnClose()` function, which continues the close process. The view object and the corresponding window are both still active. When you're using the full application framework, you probably won't use the `WM_CLOSE` message handler. You can override the `CDocument::SaveModified` virtual function instead, as part of the application framework's highly structured program exit procedure.

The `WM_QUERYENDSESSION` Message

Windows sends the `WM_QUERYENDSESSION` message to all running applications when the user exits Windows. The `OnQueryEndSession()` message map function handles it. If you write a handler for `WM_CLOSE`, write one for `WM_QUERYENDSESSION` too.

The `WM_DESTROY` Message

Windows sends this message after the `WM_CLOSE` message and the `OnDestroy()` message map function handles it. When your program receives this message, it should assume that the view window is no longer visible on the screen but that it is still active and its child windows are still active. Use this message handler to do cleanup that depends on the existence of the underlying window. Be sure to call the base class `OnDestroy()` function. You cannot "abort" the window destruction process in your view's `OnDestroy()` function. `OnClose()` is the place to do that.

The `WM_NCDESTROY` Message

This is the last message that Windows sends when the window is being destroyed. All child windows have already been destroyed. You can do final processing in `OnNcDestroy()` that doesn't depend on a window being active. Be sure to call the base class `OnNcDestroy()` function. Do not try to destroy a dynamically allocated window object in `OnNcDestroy()`. That job is reserved for a special `CWnd` virtual function, `PostNcDestroy()`, that the base class `OnNcDestroy()` calls. MFC Technical Note #17 in the online documentation gives hints on when it's appropriate to destroy a window object.

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.