Sketcher (Part 3)

Building Graphical User Interfaces (GUIs)

Using the Microsoft Foundation Classes (MFC)

All of these examples assume that Microsoft Visual C++ 2017 is the compiler being used. Based on a tutorial written by Dr. Rick Coleman.

You also must have the "Visual C++ MFC for x86 and x64" feature enabled in the VS Installer.

Exercise 5: Sketcher (Part 3) - a drawing program

Adding Shape Classes

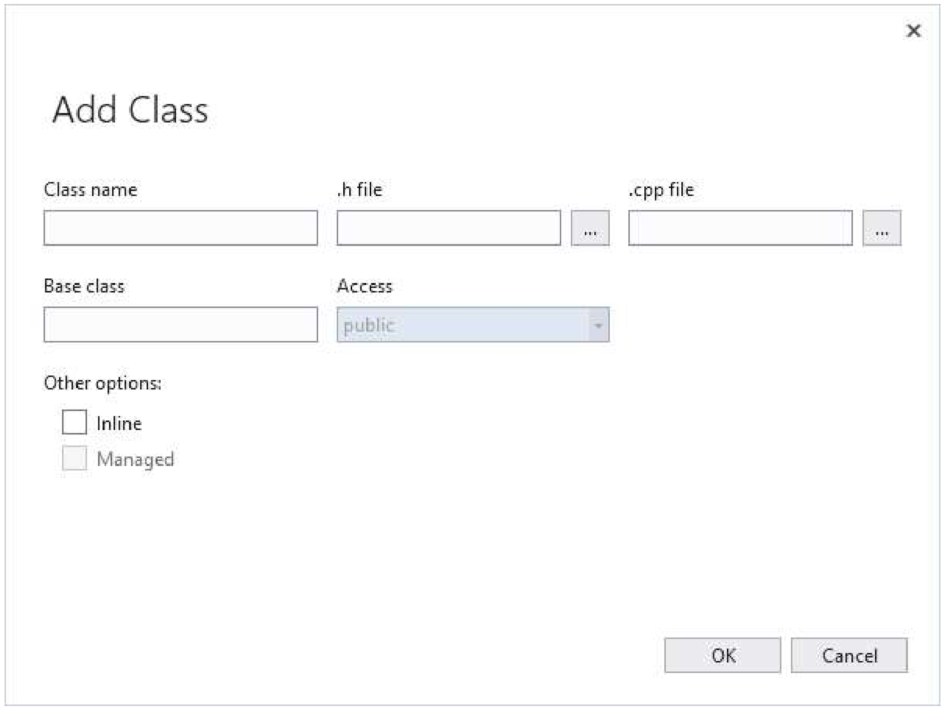
Since it will be necessary to redraw everything in the application every time it gets a WM_PAINT command (which calls the OnDraw() function), it will be necessary to find a way of storing all of the drawing activities of a user. To do this with a proper **Object Oriented** approach we will define a parent class which we will call **CShape** and then define sub-classes called **CLine**, **CRectangle**, **COval**, and **CCurve.** (Remember the classic Shape example from your early days of learning OO programming? We're actually going to use it now.) These classes will define each of the shape types for Sketcher. All of the CShape objects created by the user will be kept in a **vector** template object.

Define a CShape class file

To define this parent class we'll take advantage of the **Class Wizard** which will create both the .h and .cpp file for us.

1. Click the **Class View** tab (it should be next to the **Solution Explorer** tab).
2. Right click the project name and Select **Add->Class...**

3. A dialog similar to the following will pop up:



4. In the class wizard dialog box enter the name for the class as **CShape**. Change the name of both the .h and .cpp files so that they also are **CShape** (if they're not already).
5. Click the **OK** button.
6. Open the CShape.h file and copy and paste the code below into it (replacing any code that Visual Studio already put there).

```
//===========================================================
// CShape.h
// Class definition file for parent class for all shapes to
//   be drawn in Sketcher
// Author: Dr. Rick Coleman
//===========================================================
#pragma once

#include "Constants.h"

class CShape
{
	protected:
		COLORREF m_PenColor;    // Reference the one in Constants.h
		int m_iPenPattern;
		int m_iPenWidth;
		COLORREF m_BrushColor; // Reference the one in Constants.h
		int m_iBrushPattern;
		CRect m_EnclosingRect;

	public:
		CShape(void);
		~CShape(void);
		virtual void Draw(CDC *pDC); // Force all sub-classes to implement

		// Get and set functions for all properties
		COLORREF getPenColor();
		void setPenColor(COLORREF pCol);
```

```
                int getPenPattern();
                void setPenPattern(int pPat);
                int getPenWidth();
                void setPenWidth(int pWd);
                COLORREF getBrushColor();
                void setBrushColor(COLORREF bCol);
                int getBrushPattern();
                void setBrushPattern(int bPat);
                CRect getEnclosingRect();
                void setEnclosingRect(CRect pRect);
                void setEnclosingRect(int left, int top, int right, int bottom);
                void setEnclosingRect(CPoint ul, CPoint lr);
};
```

7. Open the CShape.cpp file and copy and paste the code below into it (again replacing any code that Visual Studio already put there).

```
//==========================================================
// CShape.cpp
// Implementation file for parent class for all shapes to be
//   drawn in Sketcher
// Author: Dr. Rick Coleman
//==========================================================
// Depending on how your project is configured, you may need to replace
//  the "pch.h" include with something else, but this should work if your
//  project is configured as shown in the screenshot from Part 1

#include "pch.h"
#include "CShape.h"

//---------------------------------------
// Default constructor
//---------------------------------------
CShape::CShape(void)
{
        m_PenColor = COLOR_BLACK;
        m_iPenPattern = PS_SOLID;
        m_iPenWidth = 1;
        m_BrushColor = COLOR_WHITE;
        m_iBrushPattern = BRUSH_PATTERN_SOLID;
        m_EnclosingRect.left = 0;
        m_EnclosingRect.top = 0;
        m_EnclosingRect.right = 0;
        m_EnclosingRect.bottom = 0;
}

//---------------------------------------
// Default destructor
//---------------------------------------
CShape::~CShape(void)
{
}

//---------------------------------------
// Draw(): all subclasses must override.
//---------------------------------------
void CShape::Draw(CDC *pDC)
{
}

//---------------------------------------
// Get the pen color
//---------------------------------------
COLORREF CShape::getPenColor()
{
        return m_PenColor;
}
```

```
//----------------------------------------
// Set the pen color
//----------------------------------------
void CShape::setPenColor(COLORREF pCol)
{
        m_PenColor = pCol;
}

//----------------------------------------
// Get the pen pattern
//----------------------------------------
int CShape::getPenPattern()
{
        return m_iPenPattern;
}

//----------------------------------------
// Set the pen pattern
//----------------------------------------
void CShape::setPenPattern(int pPat)
{
        m_iPenPattern = pPat;
}

//----------------------------------------
// Get the pen width
//----------------------------------------
int CShape::getPenWidth()
{
        return m_iPenWidth;
}

//----------------------------------------
// Set the pen width
//----------------------------------------
void CShape::setPenWidth(int pWd)
{
        m_iPenWidth = pWd;
}

//----------------------------------------
// Get the brush color
//----------------------------------------
COLORREF CShape::getBrushColor()
{
        return m_BrushColor;
}

//----------------------------------------
// Set the brush color
//----------------------------------------
void CShape::setBrushColor(COLORREF bCol)
{
        m_BrushColor = bCol;
}

//----------------------------------------
// Get the brush pattern
//----------------------------------------
int CShape::getBrushPattern()
{
        return m_iBrushPattern;
}

//----------------------------------------
// Set the brush pattern
```

```
//----------------------------------------
void CShape::setBrushPattern(int bPat)
{
        m_iBrushPattern = bPat;
}

//----------------------------------------
// Get the enclosing rectangle
//----------------------------------------
CRect CShape::getEnclosingRect()
{
        return m_EnclosingRect;
}

//-----------------------------------------------------
// Set the enclosing rectangle from another rectangle
//-----------------------------------------------------
void CShape::setEnclosingRect(CRect rect)
{
        m_EnclosingRect.left = rect.left;
        m_EnclosingRect.top = rect.top;
        m_EnclosingRect.right = rect.right;
        m_EnclosingRect.bottom = rect.bottom;
}

//----------------------------------------------------
// Set the enclosing rectangle from X,Y coordinates
//----------------------------------------------------
void CShape::setEnclosingRect(int left, int top, int right, int bottom)
{
        m_EnclosingRect.left = left;
        m_EnclosingRect.top = top;
        m_EnclosingRect.right = right;
        m_EnclosingRect.bottom = bottom;
}

//-------------------------------------------------------
// Set the enclosing rectangle from two CPoint objects
//-------------------------------------------------------
void CShape::setEnclosingRect(CPoint ul, CPoint lr)
{
        m_EnclosingRect.left = ul.x;
        m_EnclosingRect.top = ul.y;
        m_EnclosingRect.right = lr.x;
        m_EnclosingRect.bottom = lr.y;
}
```

Adding the CLine class

We will add just the **CLine** sub-class to **CShape** to start with and then use it to test all of the pen color and pattern options. Just to show you the two approaches to adding a class to the project, we'll add this one ourselves instead of letting the *Add Class Wizard* do it for us.

1. Right click the **Header Files** folder in your project (in Solution Explorer).
2. Select **Add->New Item...**
3. In the dialog box that appears select **Code** from the list on the left, and **Header File (.h)** from the list on the right.
4. Enter the name for the header file as **CLine.h**

5. Click the **Add** button.
6. Open the CLine.h file and copy and paste the code below into it.

```
//============================================================
// CLine.h
// Class definition file for a line shape
// Author: Dr. Rick Coleman
//============================================================
#pragma once

#include "Constants.h"
#include "CShape.h"

class CLine : public CShape
{
        public:
                CLine();
                ~CLine();
                void Draw(CDC *pDC);
};
```

7. Right click the **Source Files** folder in your project (in Solution Explorer).
8. Select **Add->New Item...**
9. In the dialog box that appears select **Code** from the list on the left, and **C++ File (.cpp)** from the list on the right.
10. Enter the name for the .cpp file as **CLine.cpp**
11. Click the **Add** button.
12. Open the CLine.cpp file and copy and paste the code below into it.

```
//============================================================
// CLine.cpp
// Class implementation file for a line shape
// Author: Dr. Rick Coleman
//============================================================
#include "pch.h"
#include "CLine.h"

//-----------------------------------------------
// Default constructor
//-----------------------------------------------
CLine::CLine()
{
}

//-----------------------------------------------
// Default destructor
//-----------------------------------------------
CLine::~CLine()
{
}

//-----------------------------------------------
// Implement the draw function for lines
//-----------------------------------------------
void CLine::Draw(CDC *pDC)
{
        // Create a pen for drawing
        CPen pen, *oldPen;
        // If creating a transparent pen use PS_NULL style
        if(this->m_PenColor == COLOR_CLEAR)
                pen.CreatePen(PS_NULL, this->m_iPenWidth, COLOR_BLACK);
        else
                pen.CreatePen(this->m_iPenPattern, this->m_iPenWidth, this->m_PenColor);
        // Set the drawing pen and hold the current pen
        oldPen = pDC->SelectObject(&pen);
        // Draw the line with the upper left and lower right corner
        //  of the enclosing rectangle defining the start and end
        //  points of the line.
        pDC->MoveTo(this->m_EnclosingRect.left, this->m_EnclosingRect.top);
        pDC->LineTo(this->m_EnclosingRect.right, this->m_EnclosingRect.bottom);
        // Reset the current pen
        pDC->SelectObject(oldPen);
        // Delete the pen we created to avoid memory leaks
        pen.DeleteObject();
}
```

Take a moment to look at the **Draw()** function in CLine.cpp. First of all we are going to define a pen to draw with. Normally this would be a simple process, but we have provided the option of creating a transparent pen. A transparent pen is not created by setting some "transparent color", but by setting the style to PS_NULL and it doesn't matter what the color is. So we check to see if m_PenColor is COLOR_CLEAR and if it is then we create a pen whose style is PS_NULL, otherwise we just create a pen in the normal way.

Next we set the pen to use in the Device Context by calling SelectObject() and hold the returned pointer in oldPen so we can reset this before we leave the draw function. With the pen set, we use the Device Context functions MoveTo() and LineTo() to draw the line.

Finally, we reset the pen back to the original pen and delete the pen we created.

But,...now for the big question:

How do we get the original points into our enclosing rectangle to do the drawing?

OK! Time Out!

We need to take a look at the procedure for drawing. Here is the basic algorithm for creating and drawing a line:

- Listen for a mouse down event and when it happens record the mouse location as the starting point.
- Listen for a mouse up event and when it happens record the mouse location as the ending point.
- Create a CLine object and set the enclosing rectangle as (startPt.x, startPt.y, endPt.x, endPt.y).
- Set the current settings for line color, line style, and line width in the CLine object. (We don't have to worry about the brush color and style since those are only used to fill shapes, and 2-D lines don't have any interior to fill.)
- Add a pointer to this CLine object to a vector of pointers to CShape objects in the ***projectName*****Doc.cpp**.
- Every time there is a call to OnDraw() in ***projectName*****View.cpp** call a draw function in ***projectName*****Doc.cpp** which will iterate through its vector of CShape objects and call the Draw() function in each.

Now, what would be really cool is, if after getting the starting point, we could continuously draw a line from the start point to the current mouse location, until we get the mouse up event. You have, no doubt, seen this in other drawing programs. The line you are drawing seems to act like a rubber band stretching from the starting point to where the cursor currently is located. This procedure is actually called **rubber banding**. Here is how the algorithm works.

- Listen for a mouse down event and when it happens get the starting point.
- Listen for mouse moved events.
- When a mouse moved event is received check to see if the mouse button is still down. If it is erase the last line drawn and redraw the line from the starting point to the current mouse position.
- Continue doing this until a mouse up event is received.

The Secret to Rubber Banding
The secret to implementing rubber banding, and what makes it really easy to do, is the fact that there are several different **Drawing Modes** in a window (the same holds true for all graphics systems, not just Windows).

There are only two drawing modes we are concerned about for this application:

- **R2_COPYPEN** - the default drawing mode
- **R2_NOTXORPEN** - the actual pen color produced is the current pen color XOR (exclusive OR) the background color.

When you set the mode to **R2_COPYPEN** you see normal drawing. But when you set the mode to **R2_NOTXORPEN** and draw a shape it appears normal until you draw it a second time in the exact same location. Then the shape disappears.
Rubber banding then becomes a very simple process. After getting the starting point from a mouse down event, every time there is a mouse moved event you would:

- Set the draw mode to **R2_NOTXORPEN**
- Draw the shape using starting point and the previous mouse moved location point to define the bounding rectangle, which will actually erase it.
- Draw the shape again using starting point and the current mouse moved location point to define the bounding rectangle, which will draw it.

The drawing mode is set in the Device Context by calling the function **SetROP2(*mode*)**. The function name is short for **Set R**aster **OP**eration **TO**. For example:

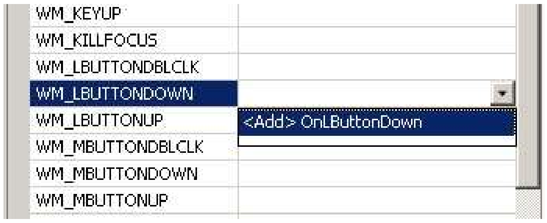
```
        pDC->SetROP2(R2_NOTXORPEN);
```

We'll implement that after we add the mouse handlers...

Capturing Mouse Events

For the Sketcher program we are interested in three mouse events: left mouse down (WM_LBUTTONDOWN), mouse moved (WM_MOUSEMOVE), and left mouse up (WM_LBUTTONUP). We want to add event handlers for these three events. This time we want to add them to the ***projectName*View** class. To do this do the following:

1. Click the Class View tab in Visual Studio.
2. Right click the ***projectName*View** class and select Properties.
3. At the top of the Properties window click the Messages button. (If you don't know which one that is just place the cursor over each button and wait a few seconds till the tool-tip message appears.)
4. Scroll down the list of messages and select WM_LBUTTONDOWN.
5. To the right, click the combobox drop down button and select <Add>OnLButtonDown.



You will see that the following function has been added to ***projectName*View**.

```
void CDemo05SketcherView::OnLButtonDown(UINT nFlags, CPoint point)
{
        // TODO: Add your message handler code here and/or call default

        CView::OnLButtonDown(nFlags, point);
}
```

Repeat this procedure to add mouse handlers for the WM_MOUSEMOVE and WM_LBUTTONUP messages.

Mouse Handler Functions

Note the arguments to the mouse handler function. The first, **nFlags** is an unsigned int which contains several bit flags indicating which keys and buttons might be down for this mouse event. These events are defined as:

- **MK_CONTROL** - The Ctrl key is pressed.
- **MK_LBUTTON** - The left mouse button is pressed.
- **MK_MBUTTON** - The middle mouse button is pressed.
- **MK_RBUTTON** - The right mouse button is pressed.
- **MK_SHIFT** - The Shift key is pressed.

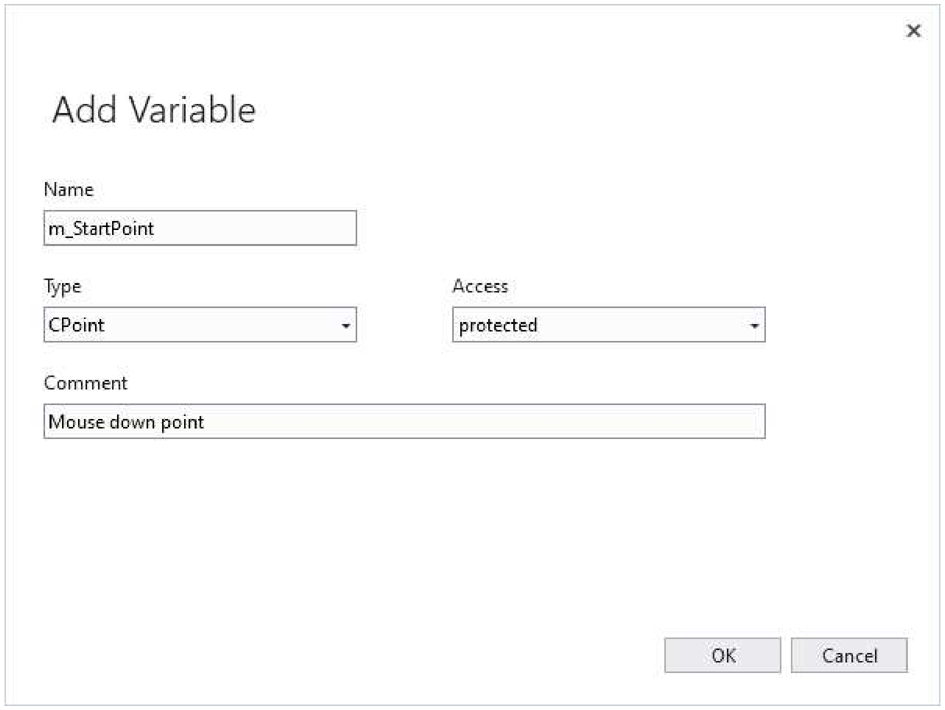
Well, of course, if we are in the left mouse down handler we know that the MK_LBUTTON bit will be set, but this is handy to know if we are in the mouse moved handler.

The second argument to the mouse handler function is a CPoint object giving the current location of the mouse cursor in the window's client area. BTW: It is possible to get a mouse up event without a corresponding mouse down event. If the mouse button was pressed while the cursor was outside this application's window, but released while inside the window then we will get a WM_LBUTTONUP event without ever having received the WM_LBUTTONDOWN event.

Add variables to record start and end points

Now we need to add a couple of variables to the ***projectName*View** class to store the starting and ending points. To do this do the following:

1. Right click the ***projectName*View** class in Class View and select **Add->Add Variable**.
2. In the dialog box select **protected** from the **Access:** combobox.
3. The **Variable type:** combobox only lists primitive data types so you will have to click in this text area and type in **CPoint**
4. In the **Variable name:** text box enter a name such as **m_StartPoint.**
5. If you want, you can add an appropriate comment as shown in the image below.
6. Click the **OK** button.


Add Variable

Name: m_StartPoint

Type: CPoint

Access: protected

Comment: Mouse down point

Add another variable, **m_EndPoint** to store the ending point in. **Important**: Scroll back up to the constructor for ***projectName*View**. We want to initialize the two points to (0, 0), so add some code to initialize both points to that:

```
CDemo05SketcherView::CDemo05SketcherView()
        m_StartPoint.x = 0;
        m_StartPoint.y = 0;
        m_EndPoint.x = 0;
        m_EndPoint.y = 0;
```

Now we are ready to add some code to the mouse event handlers.

Adding Code to the Mouse Handlers

Add the following code to the OnLButtonDown() function:

```
        void CDemo05SketcherView::OnLButtonDown(UINT nFlags, CPoint point)
        {
                m_StartPoint = point;

                CView::OnLButtonDown(nFlags, point);
        }
```

Add the following code to the OnLButtonUp() function:

```
        void CDemo05SketcherView::OnLButtonUp(UINT nFlags, CPoint point)
        {
                // If the user double clickes in the client area m_pTempShape
                //   will be Null so check for this first.
                if(m_pTempShape == NULL) return;

                // Comment out the next line till we finish the document class
                //GetDocument()->AddShape(m_pTempShape);
                m_pTempShape = NULL; // Remove the pointer from this shape

                // Note: Do not delete m_pTempShape as the shape object is
                //   now in the Document's vector

                // Force a redraw after we finish creating a shape
                InvalidateRect(NULL, TRUE);

                CView::OnLButtonUp(nFlags, point); // Pass event to super for
                                           //  any further processing
        }
```

Add the following code to the OnMouseMove() function:

```
void CDemo05SketcherView::OnMouseMove(UINT nFlags, CPoint point)
{
        if(nFlags & MK_LBUTTON)  // Check to see if the left button is down
        {
                m_EndPoint = point;  // If so save the current point

                // Test for a previous temporary CShape object
                {
                        // We get here if there was a previous mouse move event
                        // So erase the old element
                }

                // Create a new temporary CShape object
                // Draw the temporary CShape object
        }

        CView::OnMouseMove(nFlags, point);
}
```

Notice that we are checking for the left mouse button being down to be sure the user is still trying to draw. We'll fill in the missing code for this function after we add a few other needed things.

First we need to add a temporary CShape object to be drawn by the ***projectName*View** class. Add another variable just as you did before. Make its **Access = protected**, its **Variable type = CShape*** and its **Variable name = m_pTempShape**. Make sure you also modify the constructor to initialize m_pTempShape to null.

You will also need to open ***projectName*View.h** and look at the beginning of the file. Since you have now created a pointer to a CShape object in this class you will have to add the line **#include "CShape.h"** there so the program will compile.

Now add the following code to the OnMouseMove() function. Note that we are going to draw directly from the function so we have to get a device context to draw into.

```
//-------------------------------------------------------------
// Handle mouse move events
//-------------------------------------------------------------
void CDemo05SketcherView::OnMouseMove(UINT nFlags, CPoint point)
{
        if(nFlags & MK_LBUTTON)  // Check to see if the left button is down
        {
                // Define a Device Contest object for this view
                CClientDC aDC(this);
                aDC.SetROP2(R2_NOTXORPEN);              // Set the XOR drawing mode

                m_EndPoint = point;  // If so save the current point

                // Test for a previous temporary CShape object
                {
                        // We get here if there was a previous mouse move event
                        // So erase the old element
                }

                // Create a new temporary CShape object
                this->m_pTempShape = CreateShape();
                // Draw the temporary CShape object
                this->m_pTempShape->Draw(&aDC);
        }

        CView::OnMouseMove(nFlags, point);
}
```

Notice that we are calling a new function, **CreateShape().** This will simplify things by having a function that will check the current settings and create the correct shape type for us. Add the following to the ***projectName*View.h** file:

```
protected:
        CShape *CreateShape(void);   // Create a shape
```

Now add the function to the ***projectName*View.cpp** file:

```
//----------------------------------------------------------------
// Create a shape based on the current settings.
//----------------------------------------------------------------
CShape *CDemo05SketcherView::CreateShape(void)
{
        // Make sure we have a valid document just to be safe
        CDemo05SketcherDoc *pDoc = GetDocument();
        ASSERT_VALID(pDoc);                     // Crash and burn if it is not good

        // Select the shape type to create
        switch(pDoc->getCurrentShape())
        {
                case LINE : // Create a LINE object and return it
                {
                        CLine *newLine = new CLine();
                        newLine->setEnclosingRect(this->m_StartPoint, this->m_EndPoint);
                        newLine->setPenColor(pDoc->getCurrentPenColor());
                        newLine->setPenWidth(pDoc->getCurrentPenWidth());
                        newLine->setPenPattern(pDoc->getCurrentPenPattern());
                        return newLine;
                        break;
                }
                case RECTANGLE : // Create a RECTANGLE object and return it
                {
                        // Code to be added later
                        break;
                }
                case OVAL : // Create a OVAL object and return it
                {
                        // Code to be added later
                        break;
                }
                case CURVE : // Create a CURVE object and return it
                {
                        // Code to be added later
                        break;
                }
                default :   // Oops! something is wrong.
                        AfxMessageBox(_T("Bad Shape code"), MB_OK);
                        AfxAbort(); // Crash and burn
                        return NULL;
        }
        return NULL;
}
```

Notice that we must enclose the statements in each case in brackets because we are defining a new variable inside each.
BTW: Don't forget to add the **#include "CLine.h"** to the ***projectName*View.h** file.

We are almost ready to start testing. We just have to add one more thing to the OnMouseMove() function to complete it.

```
//---------------------------------------------------------------
// Handle mouse move events
//---------------------------------------------------------------
void CDemo05SketcherView::OnMouseMove(UINT nFlags, CPoint point)
{
        if(nFlags & MK_LBUTTON)  // Check to see if the left button is down
        {
                // Define a Device Contest object for this vies
                CClientDC aDC(this);
                aDC.SetROP2(R2_NOTXORPEN);              // Set the XOR drawing mode

                m_EndPoint = point;  // If so save the current point

                // Test for a previous temporary CShape object
                if(m_pTempShape != NULL)
                {
                        // Redraw the old element so it disappears
                        this->m_pTempShape->Draw(&aDC);
                        delete m_pTempShape;    // Delete the old one
                        m_pTempShape = NULL;    // Reset the pointer to NULL
                }

                // Create a new temporary CShape object
                this->m_pTempShape = CreateShape();
                // Draw the temporary CShape object
                this->m_pTempShape->Draw(&aDC);
        }

        CView::OnMouseMove(nFlags, point);
}
```

OK, compile and run. You should be able to press the left mouse button anywhere in the client area then drag the mouse around and see a line "rubber banding" between the start point and the current point. But, the line won't stay there. We'll fix that in the next part.

Congratulations, most of the hard work is done. Next we'll define all the other shapes, fill in some blanks so we can draw each of the shapes. Then, we'll add the ability to store the shapes in the document object and see how to repaint everything when we need to.