Module 13: Printing and Print Preview

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:

Printing and Print Preview

If you're depending on the Win32 API alone, printing is one of the tougher programming jobs you'll have. If you don't believe me, just skim through the 60-page module "Using the Printer" in Charles Petzold's Programming Windows 95 (Microsoft Press, 1996). Other books about Microsoft Windows ignore the subject completely. The Microsoft Foundation Class (MFC) Library version 6.0 application framework goes a long way toward making printing easy. As a bonus, it adds a print preview capability that behaves like the print preview functions in commercial Windows-based programs such as Microsoft Word and Microsoft Excel.
In this module, you'll learn how to use the MFC library **Print** and **Print Preview** features. In the process, you'll get a feeling for what's involved in Windows printing and how it's different from printing in MS-DOS. First you'll do some What You See Is What You Get - WYSIWYG printing, in which the printer output matches the screen display. This option requires careful use of mapping modes. Later you'll print a paginated data processing-style report that doesn't reflect the screen display at all. In that example, you will use a template array to structure your document so that the program can print any specified range of pages on demand.

Windows Printing

In the old days, programmers had to worry about configuring their applications for dozens of printers. Now Windows makes life easy because it provides all of the printer drivers you'll ever need. It also supplies a consistent user interface for printing.

Standard Printer Dialogs

When the user chooses **Print** from the **File** menu of a Windows-based application, the standard **Print** dialog appears, as shown in Figure 2.

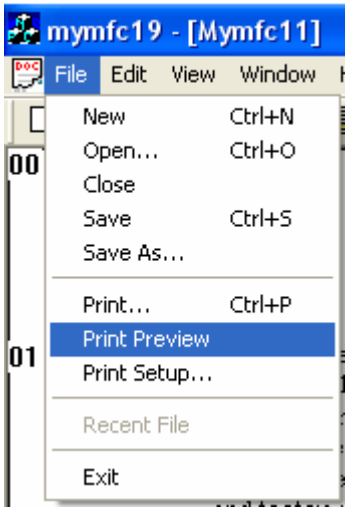

Figure 1: Standard **Print**, **Print Preview** and **Print Setup** sub menus.

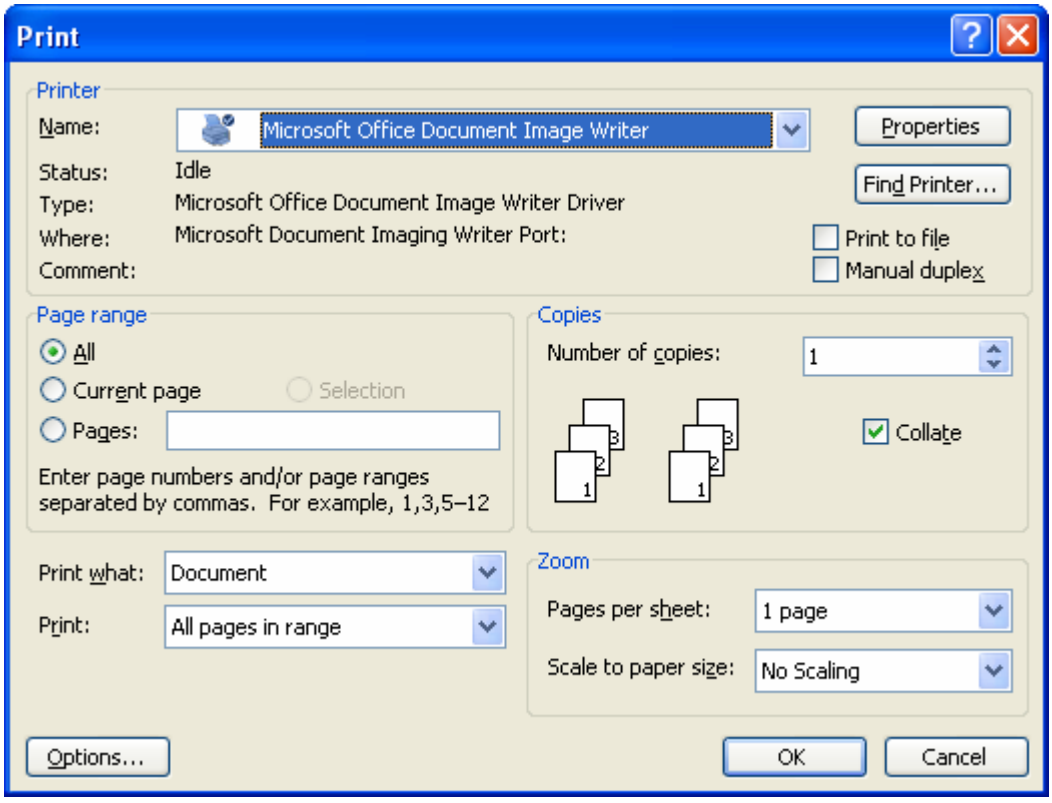

Figure 2: The standard **Print** dialog.

If the user chooses **Print Setup** from the **File** menu, the standard **Print Setup** dialog appears, as shown in Figure 3.

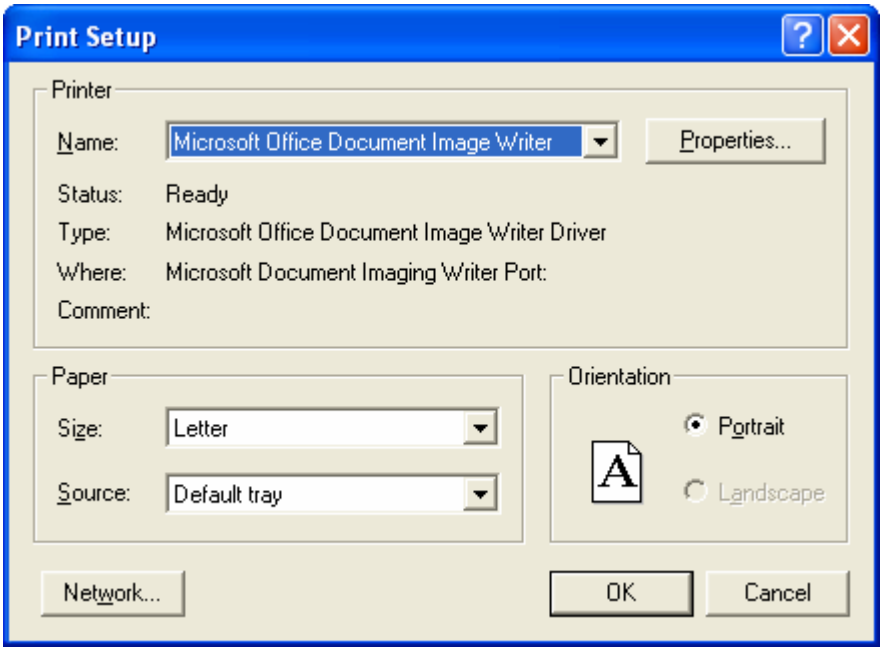

Figure 3: The standard **Print Setup** dialog.

During the printing process, the application displays a standard printer status dialog, as shown in Figure 4.

Figure 4: The standard printer status dialog.

Interactive Print Page Selection

If you've worked in the data processing field, you might be used to batch-mode printing. A program reads a record and then formats and prints selected information as a line in a report. Let's say, for example, that every time 50 lines have been printed the program ejects the paper and prints a new page heading. The programmer assumes that the whole report will be printed at one time and makes no allowance for interactively printing selected pages.

As Figure 19-1 shows, page numbers are important in Windows-based printing. A program must respond to a user's page selection by calculating which information to print and then printing the selected pages. If you're aware of this page selection requirement, you can design your application's data structures accordingly.

Remember the student list from Module 11? What if the list included 1000 students' names and the user wanted to print page 5 of a student report? If you assumed that each student record required one print line and that a page held 50 lines, page 5 would include records 201 through 250. With an MFC list collection class, you're stuck iterating through the first 200 list elements before you can start printing. Maybe the list isn't the ideal data structure. How about an array collection instead? With the `CObArray` class (or with one of the template array classes), you can directly access the 201st student record.

Not every application has elements that map to a fixed number of print lines. Suppose the student record contained a multi-line text biography field. Because you wouldn't know how many biography lines each record included, you'd have

to search through the whole file to determine the page breaks. If your program could remember those page breaks as it calculated them, its efficiency would increase.

Display Pages vs. Printed Pages

In many cases, you'll want a printed page to correspond to a display page. As you learned in Module 4, you cannot guarantee that objects will be printed exactly as they are displayed on screen. With **TrueType** fonts, however, your printed page will be **pretty close**. If you're working with full-size paper and you want the corresponding display to be readable, you'll certainly want a display window that is larger than the screen. Thus, a scrolling view such as the one that the `CScrollView` class provides is ideal for your printable views.
Sometimes, however, you might not care about display pages. Perhaps your view holds its data in a list box, or maybe you don't need to display the data at all. In these cases, your program can contain stand-alone print logic that simply extracts data from the document and sends it to the printer. Of course, the program must properly respond to a user's page-range request. If you query the printer to determine the paper size and orientation (portrait or landscape), you can adjust the pagination accordingly.

Print Preview

The MFC library **Print Preview** feature shows you on screen the exact page and line breaks you'll get when you print your document on a selected printer. The fonts might look a little funny, especially in the smaller sizes, but that's not a problem. Look now at the print preview window that appears in "The MYMFC19 Example - A WYSIWYG Print Program".
Print Preview is an MFC library feature, not a Windows feature. Don't underestimate how much effort went into programming Print Preview. The Print Preview program examines each character individually, determining its position based on the printer's device context. After selecting an approximating font, the program displays the character in the print preview window at the proper location.

Programming for the Printer

The application framework does most of the work for printing and print preview. To use the printer effectively, you must understand the sequence of function calls and know which functions to override.

The Printer Device Context and the `CView::OnDraw` Function

When your program prints on the printer, it uses a device context object of class `CDC`. Don't worry about where the object comes from; the application framework constructs it and passes it as a parameter to your view's `OnDraw()` function. If your application uses the printer to duplicate the display, the `OnDraw()` function can do double duty. If you're displaying, the `OnPaint()` function calls `OnDraw()` and the device context is the display context. If you're printing, `OnDraw()` is called by another `CView` virtual function, `OnPrint()`, with a printer device context as a parameter. The `OnPrint()` function is called once to print an entire page. In print preview mode, the `OnDraw()` parameter is actually a pointer to a `CPreviewDC` object. Your `OnPrint()` and `OnDraw()` functions work the same regardless of whether you're printing or previewing.

The `CView::OnPrint` Function

You've seen that the base class `OnPrint()` function calls `OnDraw()` and that `OnDraw()` can use both a display device context and a printer device context. The mapping mode should be set before `OnPrint()` is called. You can override `OnPrint()` to print items that you don't need on the display, such as a title page, headers, and footers. The `OnPrint()` parameters are as follows:

- A pointer to the device context.
- A pointer to a print information object (`CPrintInfo`) that includes page dimensions, the current page number, and the maximum page number.

In your overridden `OnPrint()` function, you can elect not to call `OnDraw()` at all to support print logic that is totally independent of the display logic. The application framework calls the `OnPrint()` function once for each page to be

printed, with the current page number in the CPrintInfo structure. You'll soon find out how the application framework determines the page number.

Preparing the Device Context: The CView::OnPrepareDC Function

If you need a display mapping mode other than MM_TEXT (and you often do), that mode is usually set in the view's OnPrepareDC() function. You override this function yourself if your view class is derived directly from CView, but it's already overridden if your view is derived from CScrollView. The OnPrepareDC() function is called in OnPaint() immediately before the call to OnDraw(). If you're printing, the same OnPrepareDC() function is called, this time immediately before the application framework calls OnPrint(). Thus, the mapping mode is set before both the painting of the view and the printing of a page.
The second parameter of the OnPrepareDC() function is a pointer to a CPrintInfo structure. This pointer is valid only if OnPrepareDC() is being called prior to printing. You can test for this condition by calling the CDC member function IsPrinting(). The IsPrinting() function is particularly handy if you're using OnPrepareDC() to set different mapping modes for the display and the printer. If you do not know in advance how many pages your print job requires, your overridden OnPrepareDC() function can detect the end of the document and reset the m_bContinuePrinting flag in the CPrintInfo structure. When this flag is FALSE, the OnPrint() function won't be called again and control will pass to the end of the print loop.

The Start and End of a Print Job

When a print job starts, the application framework calls two CView functions, OnPreparePrinting() and OnBeginPrinting(). (AppWizard generates the OnPreparePrinting(), OnBeginPrinting(), and OnEndPrinting() functions for you if you select the **Printing And Print Preview** option.) The first function, OnPreparePrinting(), is called before the display of the **Print** dialog. If you know the first and last page numbers, call CPrintInfo::SetMinPage and CPrintInfo::SetMaxPage in OnPreparePrinting(). The page numbers you pass to these functions will appear in the **Print** dialog for the user to override.
The second function, OnBeginPrinting(), is called after the Print dialog exits. Override this function to create Graphics Device Interface (GDI) objects, such as fonts, that you need for the entire print job. A program runs faster if you create a font once instead of re-creating it for each page. The CView function OnEndPrinting() is called at the end of the print job, after the last page has been printed. Override this function to get rid of GDI objects created in OnBeginPrinting(). The following table summarizes the important overridable CView print loop functions.

Function	Common Override Behavior
OnPreparePrinting()	Sets first and last page numbers.
OnBeginPrinting()	Creates GDI objects.
OnPrepareDC() (for each page)	Sets mapping mode and optionally detects end of print job.
OnPrint()	Does print-specific output and then calls OnDraw() (for each page)
OnEndPrinting()	Deletes GDI objects.

Table 1

The MYMFC19 Example: A WYSIWYG Print Program

This example displays and prints a single page of text stored in a document. The printed image should match the displayed image. The MM_TWIPS mapping mode is used for both printer and display. First we'll use a fixed drawing rectangle; later we'll base the drawing rectangle on the printable area rectangle supplied by the printer driver.
Here are the steps for building the example:

Run AppWizard to generate \mfcproject\mymfc19 MDI project. Accept the default options, and for **step 6**, **rename the document and view classes as shown in Table 2**, select the CScrollView as the view base class and files as shown here.

Header File	Source Code File	Class
PoemDoc.h	PoemDoc.cpp	`CPoemDoc`
StringView.h	StringView.cpp	`CStringView`

Table 2: New name for the view and document class files.

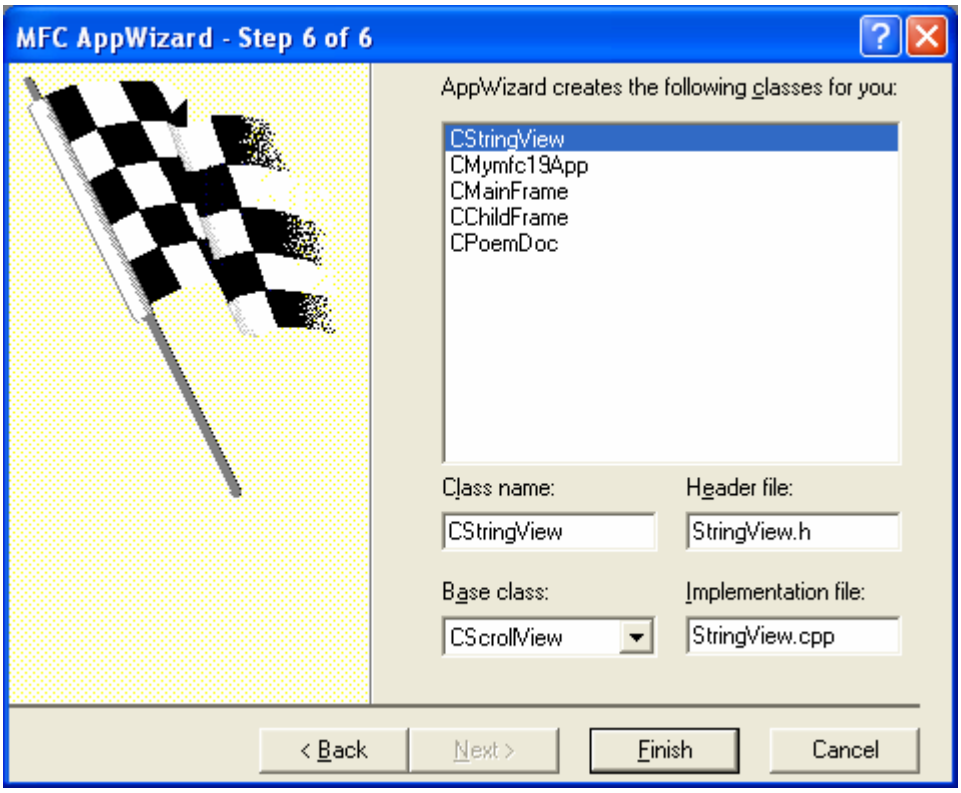

Figure 5: AppWizard step 6 of 6, **renaming the files** and selecting `CScrollView` as a view base class.

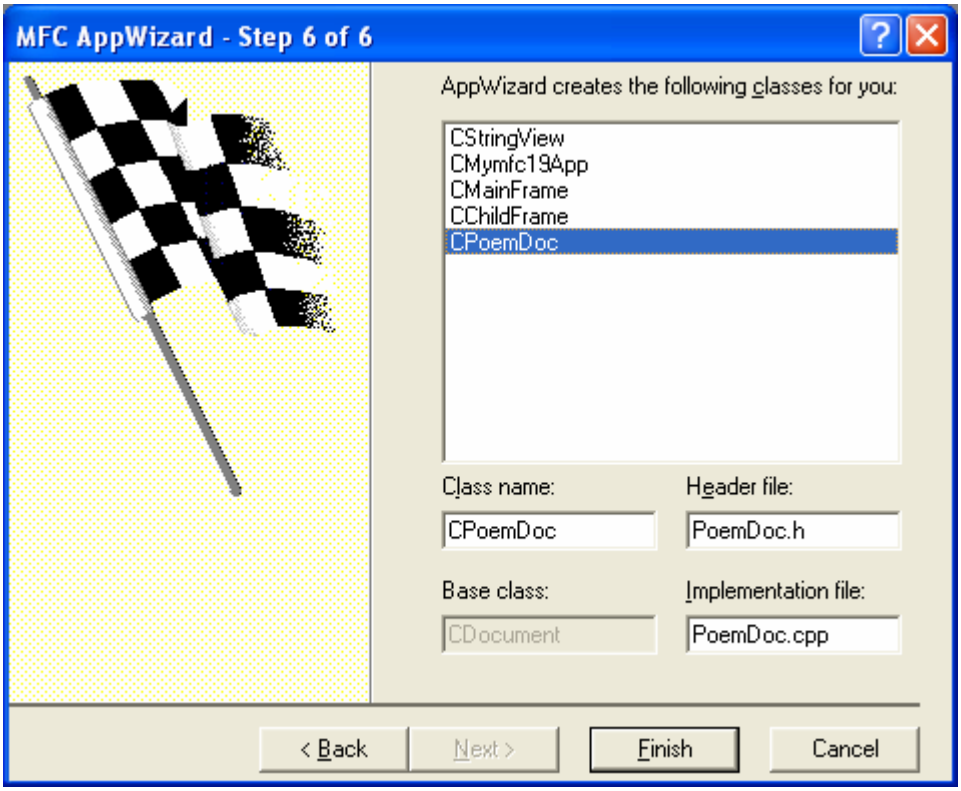

Figure 6: Renaming the document files and class.

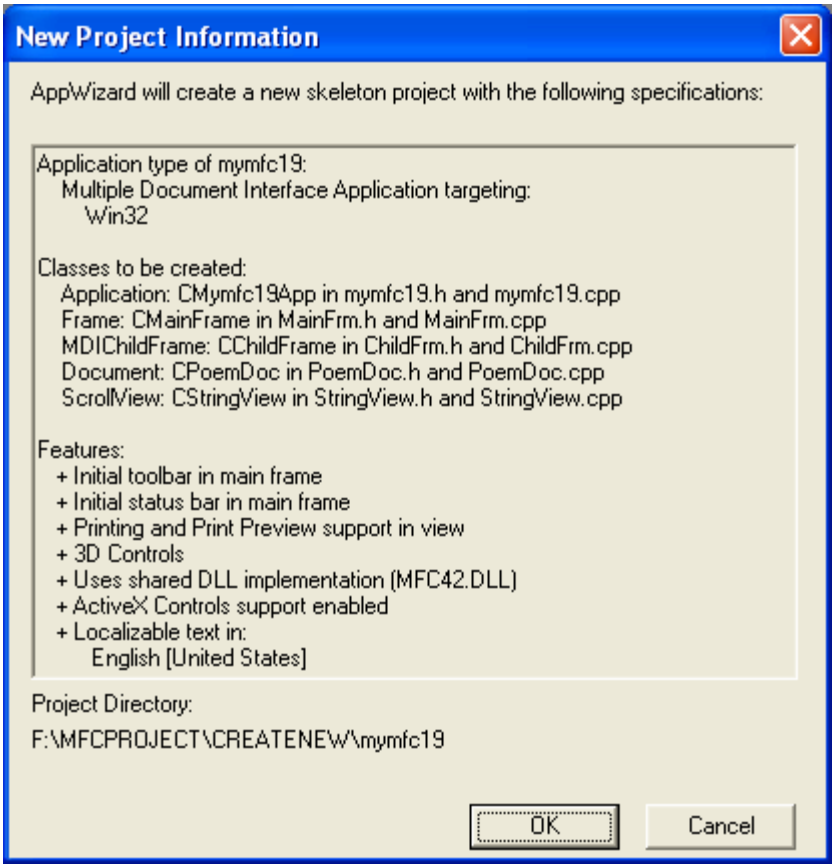

Figure 7: MYMFC19 project summary.

Note that this is an MDI application. Add a `CStringArray` data member to the `CPoemDoc` class. Edit the **PoemDoc.h** header file or use ClassView.

```
public:
    CStringArray m_stringArray;
```

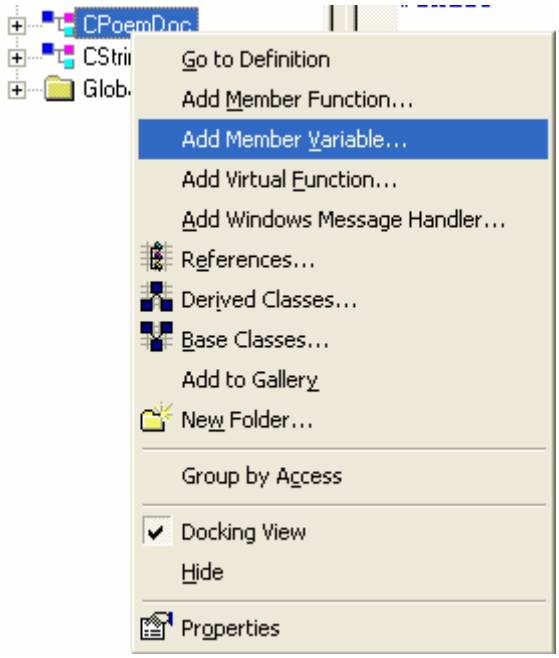

Figure 8: Adding member variable context menu.

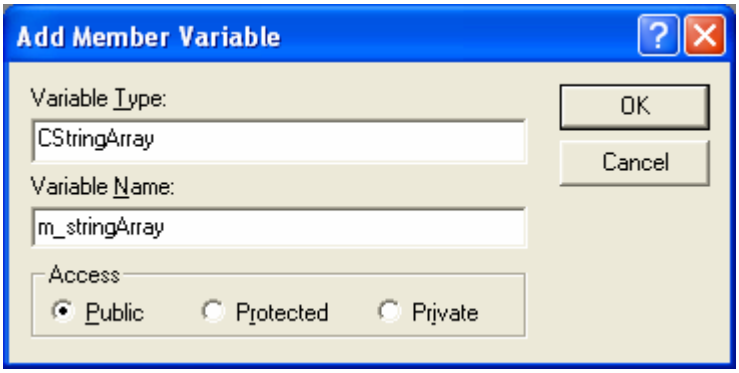

Figure 9: Adding a `CStringArray` data member to the `CPoemDoc` class.

```
// Implementation
public:
    CStringArray m_stringArray;
    virtual ~CPoemDoc();
#ifdef _DEBUG
```

Listing 1.

The document data is stored in a string array. The MFC library `CStringArray` class holds an array of `CString` objects, accessible by a zero-based subscript. You need not set a maximum dimension in the declaration because the array is dynamic.
Add a `CRect` data member to the `CStringView` class. Edit the **StringView.h** header file or use ClassView:

```
private:
    CRect m_rectPrint;
```

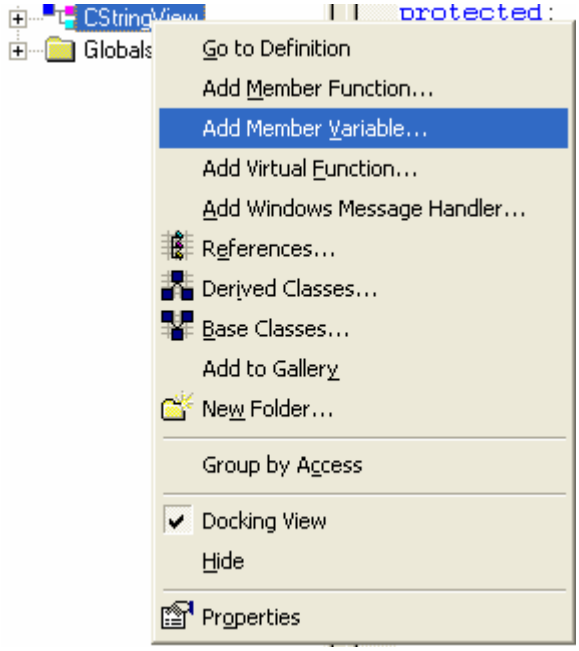

Figure 10: Adding another member variable.

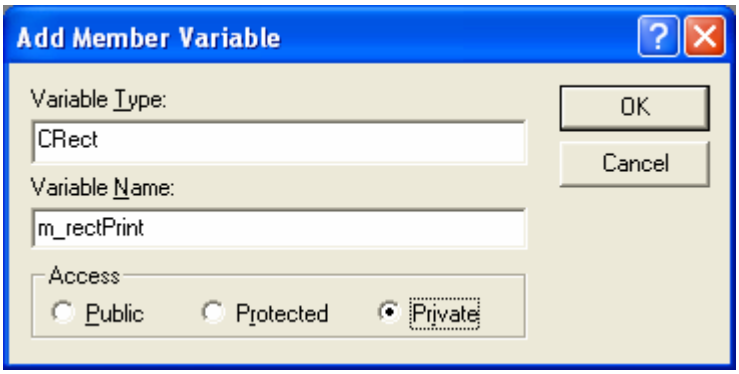

Figure 11: Adding a `CRect` data member to the `CStringView` class.

```
    DECLARE_MESSAGE_MAP()
private:
    CRect m_rectPrint;
};
```

Listing 2.

Edit three `CPoemDoc` member functions in the file **PoemDoc.cpp**. AppWizard generated skeleton `OnNewDocument()` and `Serialize()` functions, but we'll have to use ClassWizard to override the `DeleteContents()` function. We'll initialize the poem document in the overridden `OnNewDocument()` function. `DeleteContents()` is called in `CDocument::OnNewDocument`, so by calling the base class function first we're sure the poem won't be deleted. The text, by the way, is an excerpt from the twentieth poem in Lawrence Ferlinghetti's book A Coney Island of the Mind. Type 10 lines of your choice. You can substitute another poem or maybe your favorite Win32 function description. Add the following code:

```
BOOL CPoemDoc::OnNewDocument()
```

```
{
    if (!CDocument::OnNewDocument())
        return FALSE;

    m_stringArray.SetSize(10);
    m_stringArray[0] = "The pennycandystore beyond the El";
    m_stringArray[1] = "is where I first";
    m_stringArray[2] = "                  fell in love";
    m_stringArray[3] = "                     with unreality";
    m_stringArray[4] = "Jellybeans glowed in the semi-gloom";
    m_stringArray[5] = "of that september afternoon";
    m_stringArray[6] = "A cat upon the counter moved among";
    m_stringArray[7] = "                    the licorice sticks";
    m_stringArray[8] = "                 and tootsie rolls";
    m_stringArray[9] = "           and Oh Boy Gum";

    return TRUE;
}
```

```
BOOL CPoemDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;

    m_stringArray.SetSize(10);
    m_stringArray[0] = "The pennycandystore beyond the El";
    m_stringArray[1] = "is where I first";
    m_stringArray[2] = "                  fell in love";
    m_stringArray[3] = "                     with unreality";
    m_stringArray[4] = "Jellybeans glowed in the semi-gloom";
    m_stringArray[5] = "of that september afternoon";
    m_stringArray[6] = "A cat upon the counter moved among";
    m_stringArray[7] = "                    the licorice sticks";
    m_stringArray[8] = "                 and tootsie rolls";
    m_stringArray[9] = "           and Oh Boy Gum";

    return TRUE;
}
```

Listing 3.

The `CStringArray` class supports dynamic arrays, but here we're using the `m_stringArray` object as though it were a static array of 10 elements. The application framework calls the document's virtual `DeleteContents()` function when it closes the document; this action deletes the strings in the array. A `CStringArray` contains actual objects, and a `CObArray` contains pointers to objects. This distinction is important when it's time to delete the array elements. Here the `RemoveAll()` function actually deletes the string objects:

```
void CPoemDoc::DeleteContents()
{
    // called before OnNewDocument() and when document is closed
    m_stringArray.RemoveAll();
}
```

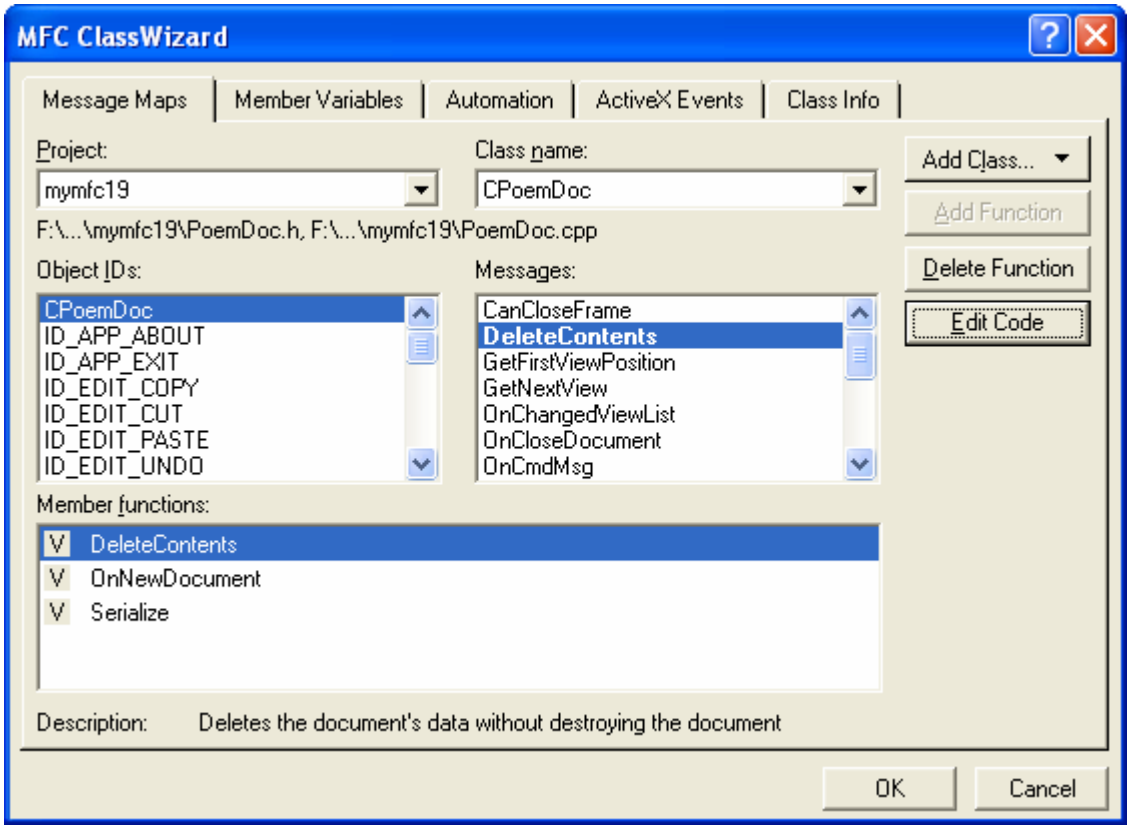

Figure 12: Adding the `RemoveAll()` function to the document class.

```
void CPoemDoc::DeleteContents()
{
    // TODO: Add your specialized code here and/or call the base class
    // called before OnNewDocument() and when document is closed
    m_stringArray.RemoveAll();
}
```

Listing 4.

Serialization isn't important in this example, but the following function illustrates how easy it is to serialize strings. The application framework calls the `DeleteContents()` function before loading from the archive, so you don't have to worry about emptying the array. Add the following boldface code:

```
void CPoemDoc::Serialize(CArchive& ar)
{
    m_stringArray.Serialize(ar);
}
```

```
void CPoemDoc::Serialize(CArchive& ar)
{
    m_stringArray.Serialize(ar);
}
```

Listing 5.

Edit the `OnInitialUpdate()` function in **StringView.cpp**. You must override the function for all classes derived from `CScrollView`. This function's job is to set the logical window size and the mapping mode. Add the following code:

```
void CStringView::OnInitialUpdate()
```

```
    {
        CScrollView::OnInitialUpdate();
        CSize sizeTotal(m_rectPrint.Width(), -m_rectPrint.Height());
        CSize sizePage(sizeTotal.cx / 2, sizeTotal.cy / 2);   // page scroll
        CSize sizeLine(sizeTotal.cx / 100, sizeTotal.cy / 100); // line scroll
        SetScrollSizes(MM_TWIPS, sizeTotal, sizePage, sizeLine);
    }
```

```
void CStringView::OnInitialUpdate()
{
    // TODO: Add your specialized code here and/or call the base class
    CScrollView::OnInitialUpdate();
    CSize sizeTotal(m_rectPrint.Width(), -m_rectPrint.Height());
    CSize sizePage(sizeTotal.cx / 2, sizeTotal.cy / 2);   // page scroll
    CSize sizeLine(sizeTotal.cx / 100, sizeTotal.cy / 100); // line scroll
    SetScrollSizes(MM_TWIPS, sizeTotal, sizePage, sizeLine);
}
```

Listing 6.

Edit the `OnDraw()` function in **StringView.cpp**. The `OnDraw()` function of class `CStringView` draws on both the display and the printer. In addition to displaying the poem text lines in 10-point roman font, it draws a border around the printable area and a crude ruler along the top and left margins. `OnDraw()` assumes the `MM_TWIPS` mapping mode, in which 1 inch = 1440 units. Add the boldface code shown below.

```
    void CStringView::OnDraw(CDC* pDC)
    {
        int        i, j, nHeight;
        CString    str;
        CFont      font;
        TEXTMETRIC tm;

        CPoemDoc* pDoc = GetDocument();
        // Draw a border – slightly smaller to avoid truncation
        pDC->Rectangle(m_rectPrint + CRect(0, 0, -20, 20));
        // Draw horizontal and vertical rulers
        j = m_rectPrint.Width() / 1440;
        for (i = 0; i <= j; i++)
        {
            str.Format("%02d", i);
            pDC->TextOut(i * 1440, 0, str);
        }
        j = -(m_rectPrint.Height() / 1440);
        for (i = 0; i <= j; i++)
        {
            str.Format("%02d", i);
            pDC->TextOut(0, -i * 1440, str);
        }
        // Print the poem 0.5 inch down and over;
        // use 10-point roman font
        font.CreateFont(-200, 0, 0, 0, 400, FALSE, FALSE, 0, ANSI_CHARSET,
                        OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS,
                        DEFAULT_QUALITY, DEFAULT_PITCH | FF_ROMAN,
                        "Times New Roman");
        CFont* pOldFont = (CFont*) pDC->SelectObject(&font);
        pDC->GetTextMetrics(&tm);
        nHeight = tm.tmHeight + tm.tmExternalLeading;
        TRACE("font height = %d, internal leading = %d\n", nHeight,
    tm.tmInternalLeading);
        j = pDoc->m_stringArray.GetSize();
        for (i = 0; i < j; i++)
        {
            pDC->TextOut(720, -i * nHeight - 720, pDoc->m_stringArray[i]);
        }
```

```
        pDC->SelectObject(pOldFont);
        TRACE("LOGPIXELSX = %d, LOGPIXELSY = %d\n", pDC->GetDeviceCaps(LOGPIXELSX),
              pDC->GetDeviceCaps(LOGPIXELSY));
        TRACE("HORZSIZE = %d, VERTSIZE = %d\n", pDC->GetDeviceCaps(HORZSIZE),
              pDC->GetDeviceCaps(VERTSIZE));
    }
```

```
void CStringView::OnDraw(CDC* pDC)
{
    // TODO: add draw code for native data here
    int         i, j, nHeight;
    CString     str;
    CFont       font;
    TEXTMETRIC tm;

    CPoemDoc* pDoc = GetDocument();
    // Draw a border - slightly smaller to avoid truncation
    pDC->Rectangle(m_rectPrint + CRect(0, 0, -20, 20));

    // Draw horizontal and vertical rulers
    j = m_rectPrint.Width() / 1440;

    for (i = 0; i <= j; i++)
    {
        str.Format("%02d", i);
        pDC->TextOut(i * 1440, 0, str);
    }

    j = -(m_rectPrint.Height() / 1440);

    for (i = 0; i <= j; i++)
```

Listing 7.

Edit the `OnPreparePrinting()` function in **StringView.cpp**. This function sets the maximum number of pages in the print job. This example has only one page. It's absolutely necessary to call the base class `DoPreparePrinting()` function in your overridden `OnPreparePrinting()` function. Add the following code:

```
    BOOL CStringView::OnPreparePrinting(CPrintInfo* pInfo)
    {
        pInfo->SetMaxPage(1);
        return DoPreparePrinting(pInfo);
    }
```

```
BOOL CStringView::OnPreparePrinting(CPrintInfo* pInfo)
{
    pInfo->SetMaxPage(1);
    return DoPreparePrinting(pInfo);
}
```

Listing 8.

Edit the constructor in **StringView.cpp**. The initial value of the print rectangle should be 8-by-15 inches, expressed in twips (1 inch = 1440 twips). Add the following boldface code:

```
    CStringView::CStringView() : m_rectPrint(0, 0, 11520, -15120)
    {
    }
```

```
CStringView::CStringView() : m_rectPrint(0, 0, 11520, -15120)
{
    // TODO: add construction code here
}
```

Listing 9.

Build and test the application. If you run the MYMFC19 application under Microsoft Windows NT with the lowest screen resolution, your MDI child window will look like the one shown below. The text will be larger under higher resolutions and under Windows 95 and Windows 98.

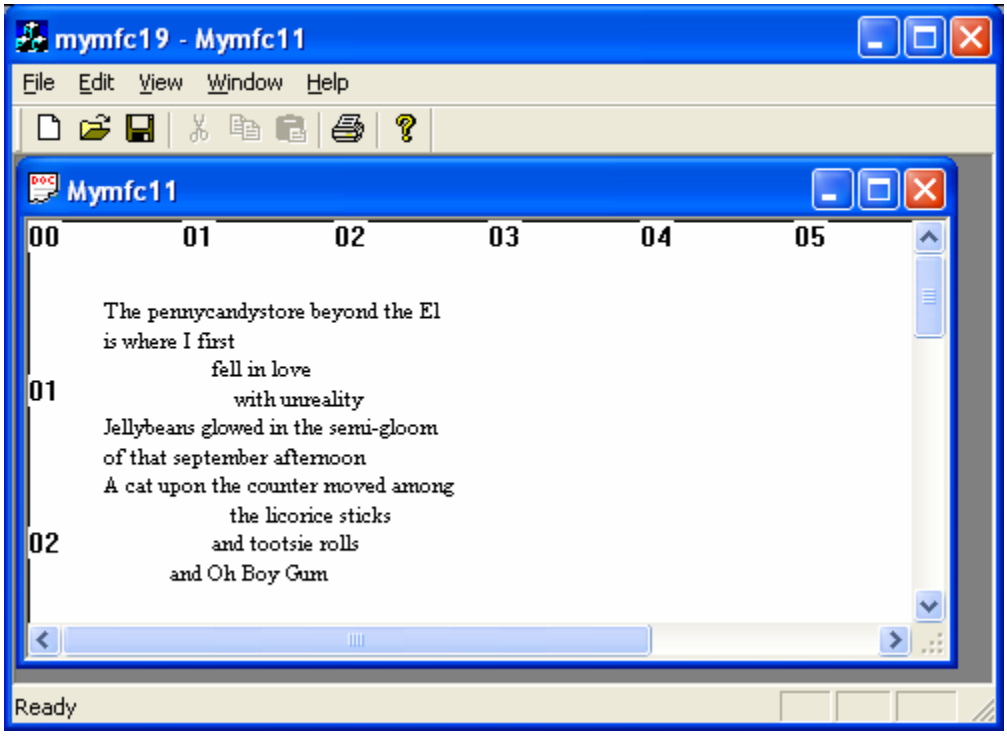

Figure 13: MYMFC19 program output.

The window text is too small, isn't it? Go ahead and choose **Print Preview** from the **File** menu, and then click twice with the magnifying glass to enlarge the image. The print preview output is illustrated here.

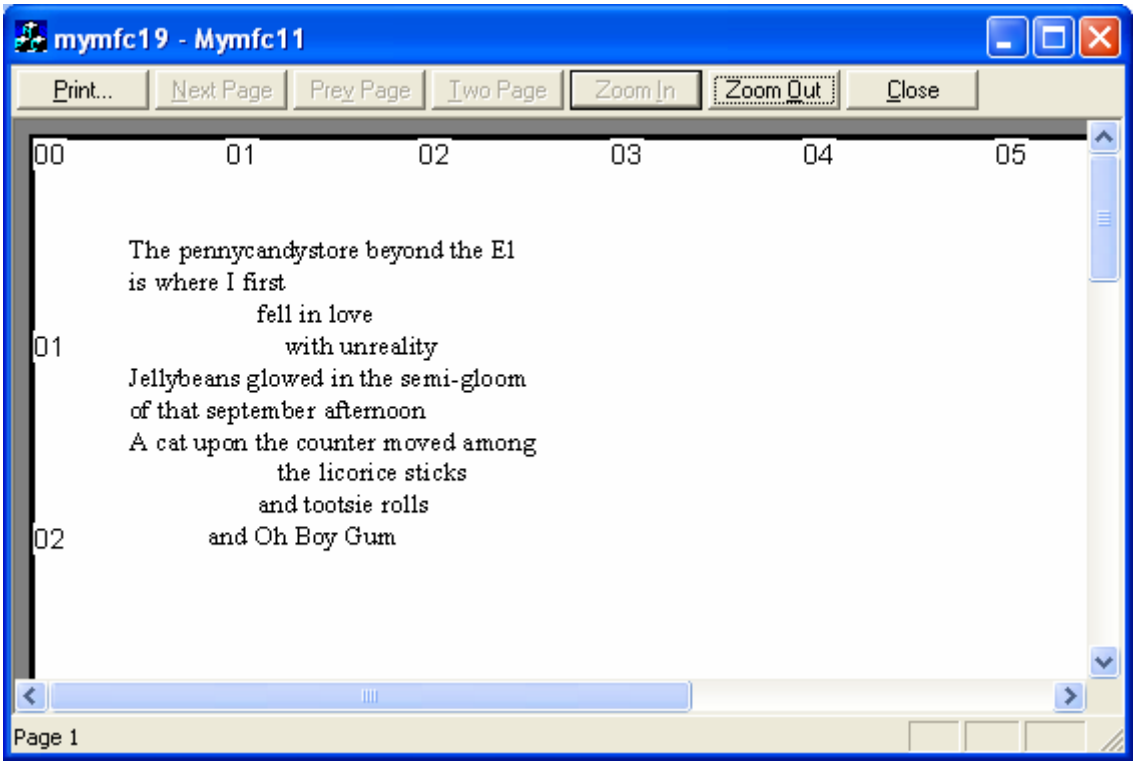

Figure 14: MYMFC19 program output when activating the **Print Preview** menu.

Remember "logical twips" from Module 4? We're going to use logical twips now to enlarge type on the display while keeping the printed text the same size. This requires some extra work because the `CScrollView` class wasn't designed for non-standard mapping modes. You will be changing the view's base class from `CScrollView` to `CLogScrollView`, which is a class that I created by copying and modifying the MFC code in **ViewScrl.cpp**. The files **LogScrollView.h** and **LogScrollView.cpp** links are given at the end of this Module.

Insert the `CScrollView` class into the project. Copy the files **LogScrollView.h** and **LogScrollView.cpp** from the given links to the project directory mfcproject\mymfc19 if you have not done so already. Choose **Add To Project** from the **Project** menu, and then choose **Files** from the submenu. Select the two new files and click **OK** to insert them into the project.

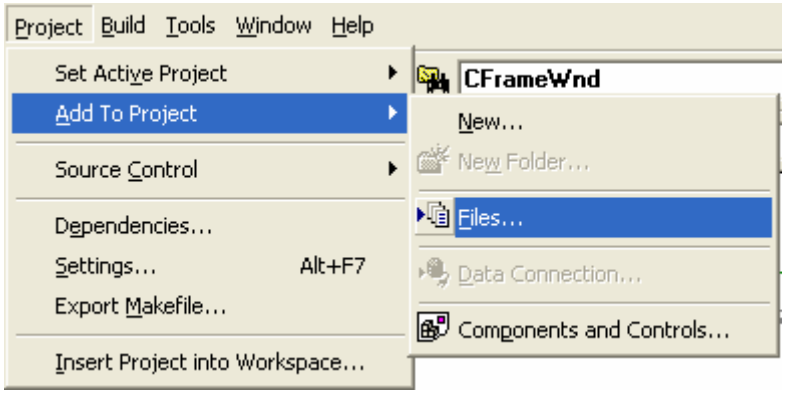

Figure 15: Adding the already available files into the project.

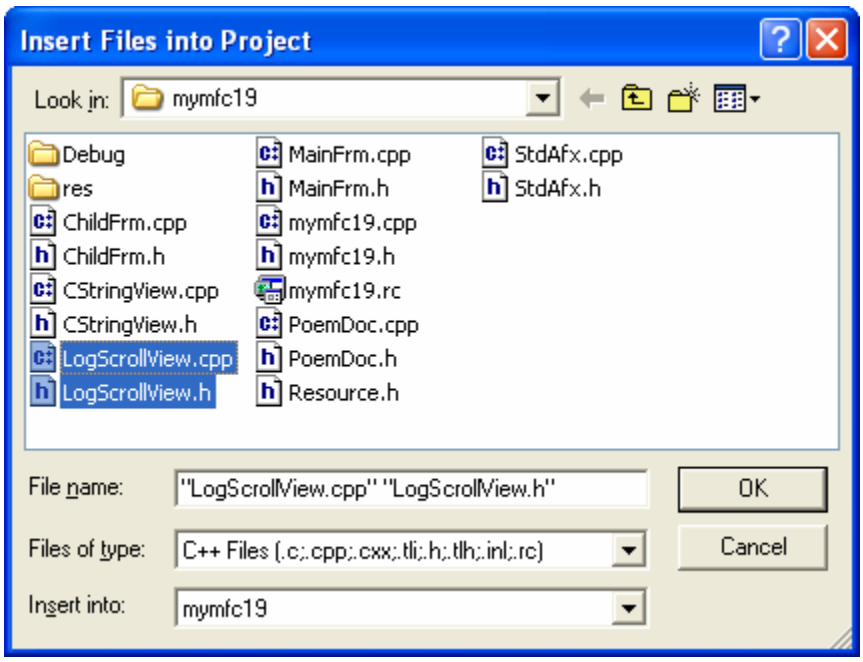

Figure 16: Selecting files to be included in the project.

Edit the **StringView.h** header file. Add the following line at the top of the file:

```
#include "LogScrollView.h"
```

```
////////////////////////////////

#include "LogScrollView.h"

#if !defined(AFX_CSTRINGVIEW_
#define AFX_CSTRINGVIEW_H__C3
```

Listing 10.

Then change the line:

```
class CStringView : public CScrollView
```

to

```
class CStringView : public CLogScrollView
```

```
#endif // _MSC_VER > 1000

class CStringView : public CLogScrollView
{
protected: // create from serialization only
    CStringView();
```

Listing 11.

Edit the **StringView.cpp** file. Globally replace all occurrences of `CScrollView` with `CLogScrollView`. You can use the **Edit Replace** menu for this task.

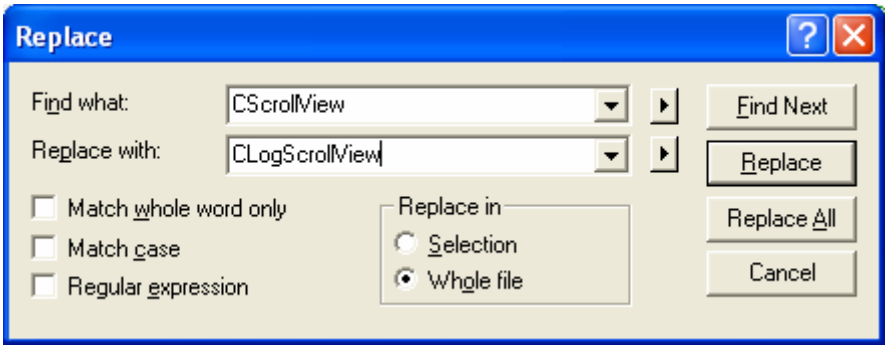

Figure 17: Using find and replace to replace all occurrences of `CScrollView` with `CLogScrollView`.

Then edit the `OnInitialUpdate()` function. Here is the edited code, which is much shorter:

```
void CStringView::OnInitialUpdate()
{
    CLogScrollView::OnInitialUpdate();
    CSize sizeTotal(m_rectPrint.Width(), -m_rectPrint.Height());
    SetLogScrollSizes(sizeTotal);
}
```

```
void CStringView::OnInitialUpdate()
{
    CLogScrollView::OnInitialUpdate();
    CSize sizeTotal(m_rectPrint.Width(), -m_rectPrint.Height());
    SetLogScrollSizes(sizeTotal);
}

////////////////////////////////////////////////////////////////////////
// CStringView printing
```

Listing 12.

Build and test the application again. Now the window text is larger.

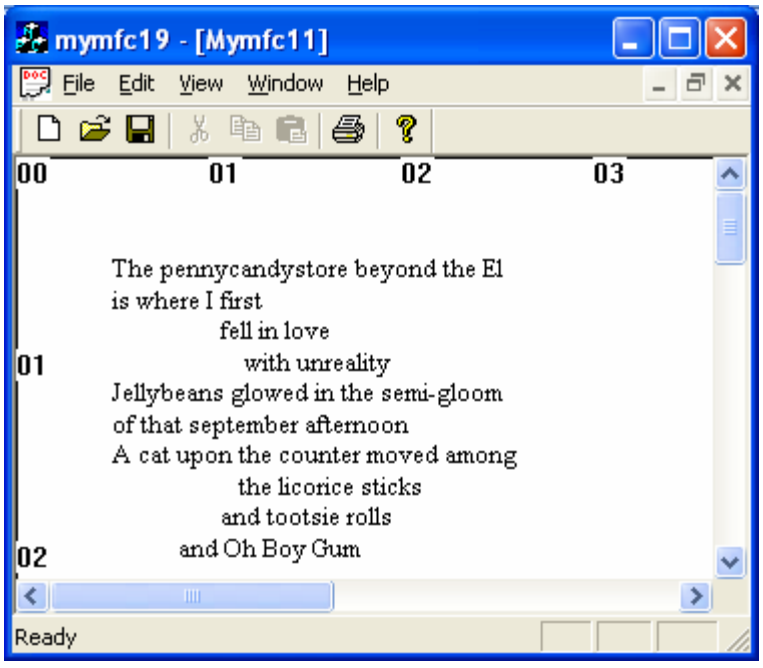

Figure 18: Modified text size of the MYMFC19 printing program output.

Reading the Printer Rectangle

The MYMFC19 program prints in a fixed-size rectangle that's appropriate for a laser printer set to portrait mode with 8.5-by-11-inch (letter-size) paper. But what if you load European-size paper or you switch to landscape mode? The program should be able to adjust accordingly.
It's relatively easy to read the printer rectangle. Remember the `CPrintInfo` pointer that's passed to `OnPrint()`? That structure has a data member `m_rectDraw` that contains the rectangle in logical coordinates. Your overridden `OnPrint()` function simply stuffs the rectangle in a view data member, and `OnDraw()` uses it. There's only one problem: you can't get the rectangle until you start printing, so the constructor still needs to set a default value for `OnDraw()` to use before printing begins.
If you want the MYMFC19 program to read the printer rectangle and adjust the size of the scroll view, use ClassWizard to override `OnPrint()` and then code the function as follows:

```
void CStringView::OnPrint(CDC* pDC, CPrintInfo* pInfo)
{
    m_rectPrint = pInfo->m_rectDraw;
    SetLogScrollSizes(CSize(m_rectPrint.Width(), -m_rectPrint.Height()));
    CLogScrollView::OnPrint(pDC, pInfo);
}
```

Template Collection Classes Revisited: The `CArray` Class

In MYMFC16 in Module 10, you saw the MFC library `CTypedPtrList` template collection class, which was used to store a list of pointers to `CStudent` objects. Another collection class, `CArray`, is appropriate for the next example, MYMFC20. This class is different from `CTypedPtrList` in two ways. First, it's an array, with elements accessible by index, just like `CStringArray` in MYMFC19. Second, the array holds actual objects, not pointers to objects. In MYMFC20, the elements are `CRect` objects. The elements' class does not have to be derived from `CObject`, and indeed, `CRect` is not. As in MYMFC16, a `typedef` makes the template collection easier to use. We use the statement:

```
typedef CArray<CRect, CRect&> CRectArray;
```

to define an array class that holds `CRect` objects and whose functions take `CRect` reference parameters. It's cheaper to pass a 32-bit pointer than to copy a 128bit object. To use the template array, you declare an instance of `CRectArray` and then you call `CArray` member functions such as `SetSize()`. You can also use the `CArray` subscript operator to get and set elements.
The template classes `CArray`, `CList`, and `CMap` are easy to use if the element class is sufficiently simple. The `CRect` class fits that description because it contains no pointer data members. Each template class uses a global function, `SerializeElements()`, to serialize all the elements in the collection. The default `SerializeElements()` function does a bitwise copy of each element to and from the archive. If your element class contains pointers or is otherwise complex, you'll need to write your own `SerializeElements()` function. If you wrote this function for the rectangle array (not required), your code would look like this:

```
void AFXAPI SerializeElements(CArchive& ar, CRect* pNewRects, int nCount)
{
    for (int i = 0; i < nCount; i++, pNewRects++)
    {
        if (ar.IsStoring()) {
            ar << *pNewRects;
        }
        else {
            ar >> *pNewRects;
        }
    }
}
```

When the compiler sees this function, it uses the function to replace the `SerializeElements()` function inside the template. This only works, however, if the compiler sees the `SerializeElements()` prototype before it sees the template class declaration. The template classes depend on two other global functions, `ConstructElements()` and

`DestructElements()`. Starting with Visual C++ version 4.0, these functions call the element class constructor and destructor for each object. Therefore, there's no real need to replace them.

The MYMFC20 Example: A Multipage Print Program

In this example, the document contains an array of 50 `CRect` objects that define circles. The circles are randomly positioned in a 6-by-6-inch area and have random diameters of as much as 0.5 inch. The circles, when drawn on the display, look like two-dimensional simulations of soap bubbles. Instead of drawing the circles on the printer, the application prints the corresponding `CRect` coordinates in numeric form, 12 to a page, with headers and footers.

Run AppWizard to generate\mfcproject \mymfc20. Select **Single Document**, and accept the defaults for all the other settings. The options and the default class names are shown here.

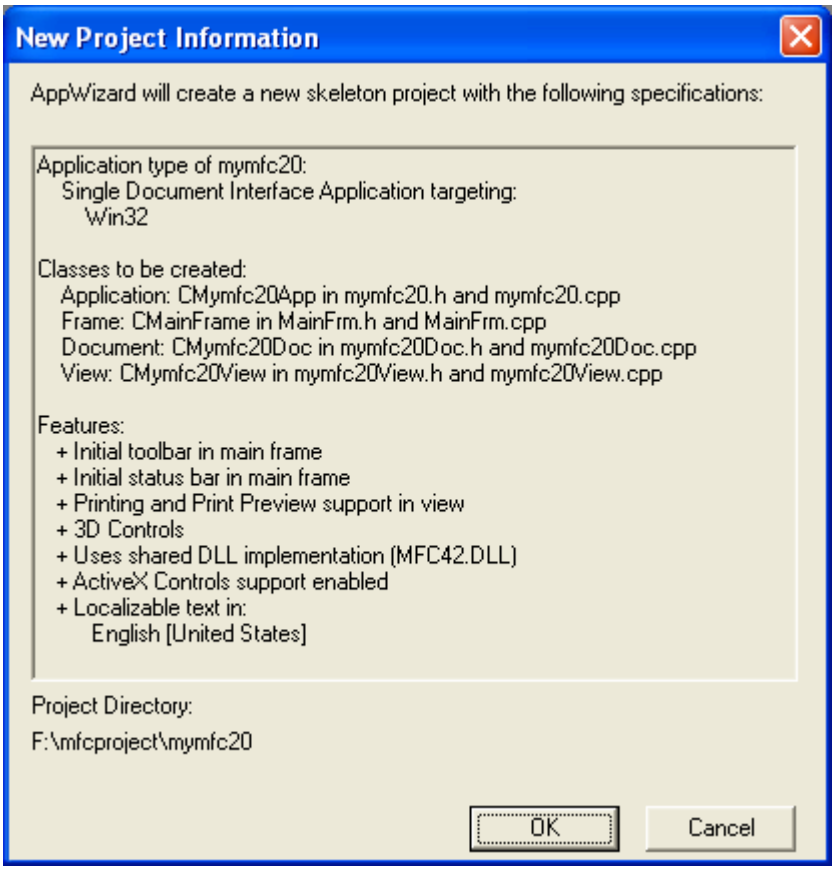

Figure 19: MYMFC20 project summary.

Edit the **StdAfx.h** header file. You'll need to bring in the declarations for the MFC template collection classes. Add the following statement:

```
   #include <afxtempl.h>

#endif // _AFX_NO_AFXCMN_SUPPORT

#include <afxtempl.h>

//{{AFX_INSERT_LOCATION}}
```

Listing 13.

Edit the **mymfc20Doc.h** header file. In the MYMFC19 example, the document data consists of strings stored in a `CStringArray` collection. Because we're using a template collection for ellipse rectangles, we'll need a `typedef` statement outside the class declaration, as shown here:

```
typedef CArray<CRect, CRect&> CRectArray;
```

```
#if !defined(AFX_MYMFC20DOC_H__2E385788_CI
#define AFX_MYMFC20DOC_H__2E385788_CBB6_4

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

typedef CArray<CRect, CRect&> CRectArray;

class CMymfc20Doc : public CDocument
{
```

Listing 14.

Next add the following public data members to the **mymfc20Doc.h** header file:

```
public:
    enum { nLinesPerPage = 12 };
    enum { nMaxEllipses = 50 };
    CRectArray m_ellipseArray;
```

```
// Attributes
public:

public:
    enum { nLinesPerPage = 12 };
    enum { nMaxEllipses = 50 };
    CRectArray m_ellipseArray;

// Operations
public:
```

Listing 15.

The two enumerations are object-oriented replacements for #defines.

Edit the **mymfc20Doc.cpp** implementation file. The overridden `OnNew()` Document function initializes the ellipse array with some random values, and the `Serialize()` function reads and writes the whole array. AppWizard generated the skeletons for both functions. You don't need a `DeleteContents()` function because the `CArray` subscript operator writes a new `CRect` object on top of any existing one. Add the following code:

```
BOOL CMymfc20Doc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;

    int n1, n2, n3;
    // Make 50 random circles
    srand((unsigned) time(NULL));
    m_ellipseArray.SetSize(nMaxEllipses);

    for (int i = 0; i < nMaxEllipses; i++)
    {
        n1 = rand() * 600 / RAND_MAX;
        n2 = rand() * 600 / RAND_MAX;
        n3 = rand() * 50  / RAND_MAX;
        m_ellipseArray[i] = CRect(n1, -n2, n1 + n3, -(n2 + n3));
```

```
        }

        return TRUE;
    }
```

```
BOOL CMymfc20Doc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;

    int n1, n2, n3;
    // Make 50 random circles
    srand((unsigned) time(NULL));
    m_ellipseArray.SetSize(nMaxEllipses);

    for (int i = 0; i < nMaxEllipses; i++)
    {
        n1 = rand() * 600 / RAND_MAX;
        n2 = rand() * 600 / RAND_MAX;
        n3 = rand() * 50  / RAND_MAX;
        m_ellipseArray[i] = CRect(n1, -n2, n1 + n3, -(n2 + n3));
    }

    return TRUE;
}
```

Listing 16.

```
    void CMymfc20Doc::Serialize(CArchive& ar)
    {
        m_ellipseArray.Serialize(ar);
    }
```

```
// CMymfc20Doc serialization

void CMymfc20Doc::Serialize(CArchive& ar)
{
    m_ellipseArray.Serialize(ar);
}
```

Listing 17.

Edit the **mymfc20View.h** header file. Use ClassView to add the member variable and two function prototypes listed below. ClassView will also generate skeletons for the functions in **mymfc20View.cpp**.

```
    public:
        int m_nPage;

    private:
        void PrintPageHeader(CDC *pDC);
        void PrintPageFooter(CDC *pDC);
```

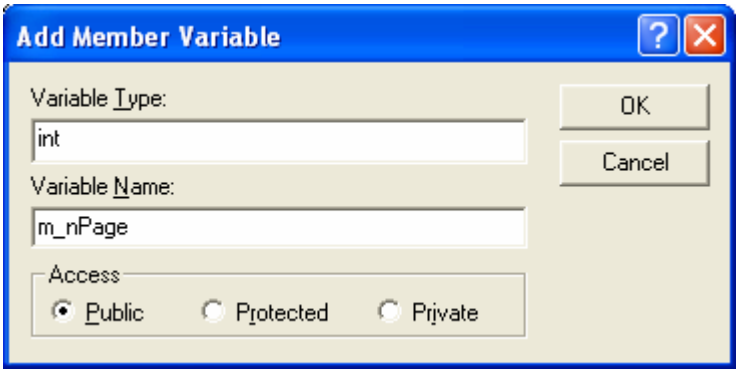

Figure 20: Adding a member variable using ClassView.

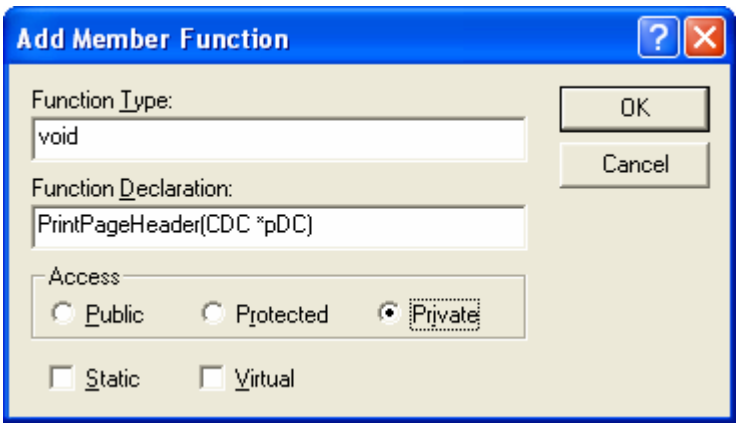

Figure 21: Adding a function prototype using ClassView.

The `m_nPage` data member holds the document's current page number for printing. The private functions are for the header and footer subroutines. Edit the `OnDraw()` function in **mymfc20View.cpp**. The overridden `OnDraw()` function simply draws the bubbles in the view window. Add the code shown here:

```
void CMymfc20View::OnDraw(CDC* pDC)
{
    int i, j;

    CMymfc20Doc* pDoc = GetDocument();

    j = pDoc->m_ellipseArray.GetUpperBound();

    for (i = 0; i < j; i++)
    {
        pDC->Ellipse(pDoc->m_ellipseArray[i]);
    }
}
```

```
// CMymfc20View drawing

void CMymfc20View::OnDraw(CDC* pDC)
{
    // TODO: add draw code for native data here
    int i, j;

    CMymfc20Doc* pDoc = GetDocument();

    j = pDoc->m_ellipseArray.GetUpperBound();

    for (i = 0; i < j; i++)
    {
        pDC->Ellipse(pDoc->m_ellipseArray[i]);
    }
}
```

Listing 18.

Insert the `OnPrepareDC()` function in **mymfc20View.cpp**. The view class is not a scrolling view, so the mapping mode must be set in this function. Use ClassWizard to override the `OnPrepareDC()` function:

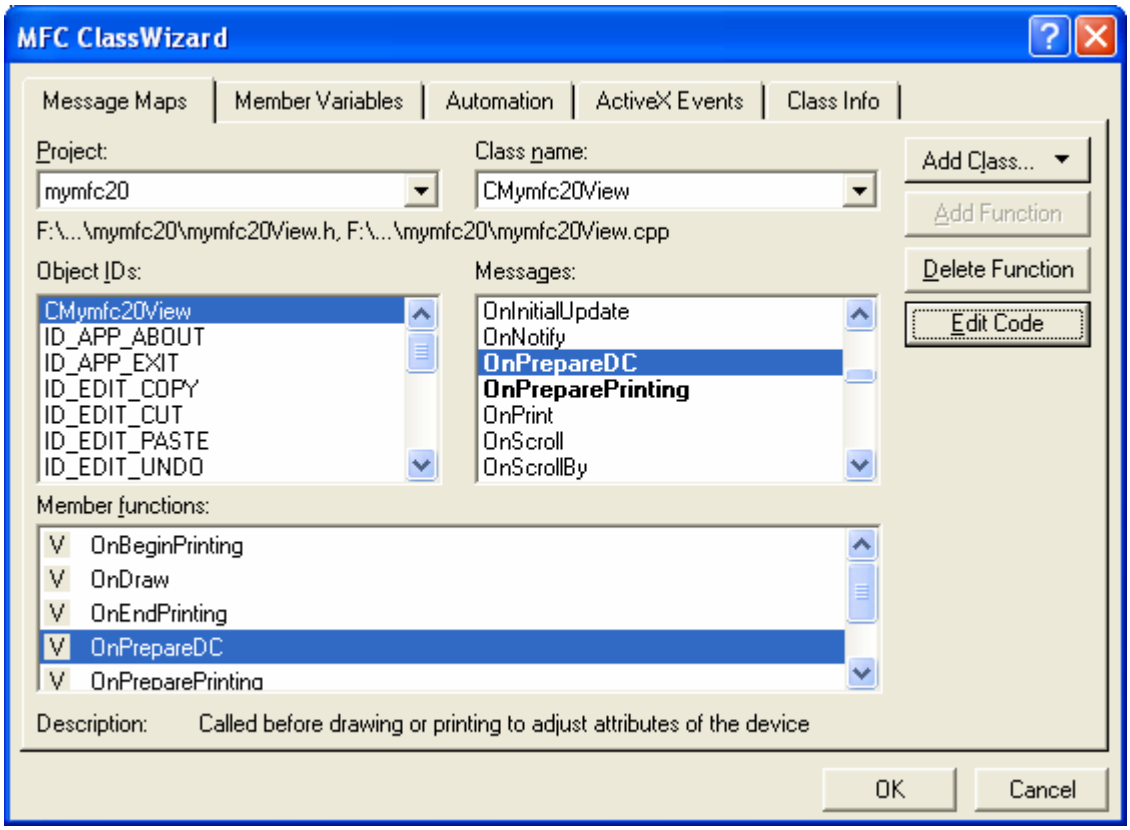

Figure 22: Overriding the `OnPrepareDC()` function.

And then add the following code:

```
void CMymfc20View::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
    pDC->SetMapMode(MM_LOENGLISH);
}
```

```
void CMymfc20View::OnPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
    // TODO: Add your specialized code here and/or call the base class
    pDC->SetMapMode(MM_LOENGLISH);
}
```

Listing 19.

Insert the `OnPrint()` function in **mymfc20View.cpp**. The `CView` default `OnPrint()` function calls `OnDraw()`. In this example, we want the printed output to be entirely different from the displayed output, so the `OnPrint()` function must take care of the print output without calling `OnDraw()`. `OnPrint()` first sets the mapping mode to `MM_TWIPS`, and then it creates a fixed-pitch font. After printing the numeric contents of 12 `m_ellipseArray` elements, `OnPrint()` deselects the font. You could have created the font once in `OnBeginPrinting()`, but you wouldn't have noticed the increased efficiency. Use ClassWizard to override the `OnPrint()` function, and then add the following code:

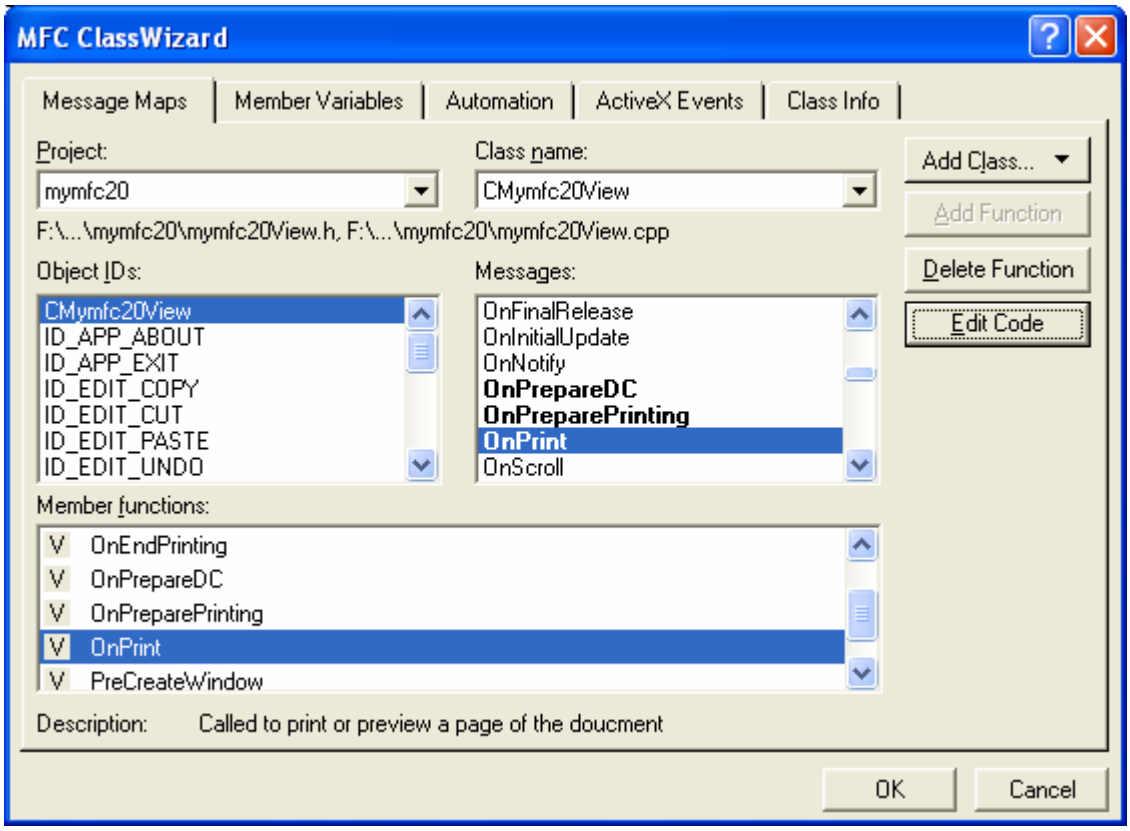

Figure 23: Inserting the `OnPrint()` function in **mymfc20View.cpp**.

```
void CMymfc20View::OnPrint(CDC* pDC, CPrintInfo* pInfo)
{
    int        i, nStart, nEnd, nHeight;
    CString    str;
    CPoint     point(720, -1440);
    CFont      font;
    TEXTMETRIC tm;

    pDC->SetMapMode(MM_TWIPS);
    CMymfc20Doc* pDoc = GetDocument();
    // for PrintPageFooter's benefit
    m_nPage = pInfo->m_nCurPage;
    nStart = (m_nPage - 1) * CMymfc20Doc::nLinesPerPage;
    nEnd = nStart + CMymfc20Doc::nLinesPerPage;
```

```
        // 14-point fixed-pitch font
        font.CreateFont(-280, 0, 0, 0, 400, FALSE, FALSE,
                        0, ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                        CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                        DEFAULT_PITCH | FF_MODERN, "Courier New");
                        // Courier New is a TrueType font
        CFont* pOldFont = (CFont*) (pDC->SelectObject(&font));
        PrintPageHeader(pDC);
        pDC->GetTextMetrics(&tm);
        nHeight = tm.tmHeight + tm.tmExternalLeading;
        for (i = nStart; i < nEnd; i++) {
            if (i > pDoc->m_ellipseArray.GetUpperBound()) {
                break;
            }
            str.Format("%6d %6d %6d %6d %6d", i + 1,
                       pDoc->m_ellipseArray[i].left,
                       pDoc->m_ellipseArray[i].top,
                       pDoc->m_ellipseArray[i].right,
                       pDoc->m_ellipseArray[i].bottom);
            point.y -= nHeight;
            pDC->TextOut(point.x, point.y, str);
        }
        PrintPageFooter(pDC);
        pDC->SelectObject(pOldFont);
    }
```

```
void CMymfc20View::OnPrint(CDC* pDC, CPrintInfo* pInfo)
{
    // TODO: Add your specialized code here and/or call the bas
        int         i, nStart, nEnd, nHeight;
    CString     str;
    CPoint      point(720, -1440);
    CFont       font;
    TEXTMETRIC tm;

    pDC->SetMapMode(MM_TWIPS);
    CMymfc20Doc* pDoc = GetDocument();
    // for PrintPageFooter's benefit
    m_nPage = pInfo->m_nCurPage;
    nStart = (m_nPage - 1) * CMymfc20Doc::nLinesPerPage;
    nEnd = nStart + CMymfc20Doc::nLinesPerPage;

    // 14-point fixed-pitch font
    font.CreateFont(-280, 0, 0, 0, 400, FALSE, FALSE,
                    0, ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                    CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                    DEFAULT_PITCH | FF_MODERN, "Courier New");
                    // Courier New is a TrueType font
    CFont* pOldFont = (CFont*) (pDC->SelectObject(&font));
    PrintPageHeader(pDC);
```

Listing 20.

Edit the `OnPreparePrinting()` function in **mymfc20View.cpp**. The `OnPreparePrinting()` function (whose skeleton is generated by AppWizard) computes the number of pages in the document and then communicates that value to the application framework through the `SetMaxPage()` function. Add the following code:

```
BOOL CMymfc20View::OnPreparePrinting(CPrintInfo* pInfo)
{
    CMymfc20Doc* pDoc = GetDocument();
    pInfo->SetMaxPage(pDoc->m_ellipseArray.GetUpperBound() /
CMymfc20Doc::nLinesPerPage + 1);
    return DoPreparePrinting(pInfo);
}
```

```
// CMymfc20View printing

BOOL CMymfc20View::OnPreparePrinting(CPrintInfo* pInfo)
{
    CMymfc20Doc* pDoc = GetDocument();
    pInfo->SetMaxPage(pDoc->m_ellipseArray.GetUpperBound() /
                      CMymfc20Doc::nLinesPerPage + 1);
    return DoPreparePrinting(pInfo);
}
```

Listing 21.

Insert the page header and footer functions in **mymfc20View.cpp**. These private functions, called from `OnPrint()`, print the page headers and the page footers. The page footer includes the page number, stored by `OnPrint()` in the view class data member `m_nPage`. The `CDC::GetTextExtent` function provides the width of the page number so that it can be right-justified. Add the code shown here:

```
void CMymfc20View::PrintPageHeader(CDC* pDC)
{
    CString str;

    CPoint point(0, 0);
    pDC->TextOut(point.x, point.y, "Bubble Report");
    point += CSize(720, -720);
    str.Format("%6.6s %6.6s %6.6s %6.6s %6.6s", "Index", "Left", "Top", "Right",
"Bottom");
    pDC->TextOut(point.x, point.y, str);
}
```

```
// CMymfc20View message handlers

void CMymfc20View::PrintPageHeader(CDC *pDC)
{
    CString str;

    CPoint point(0, 0);
    pDC->TextOut(point.x, point.y, "Bubble Report");
    point += CSize(720, -720);
    str.Format("%6.6s %6.6s %6.6s %6.6s %6.6s",
               "Index", "Left", "Top", "Right", "Bottom");
    pDC->TextOut(point.x, point.y, str);
}
```

Listing 22.

```
void CMymfc20View::PrintPageFooter(CDC* pDC)
{
    CString str;

    CPoint point(0, -14400); // Move 10 inches down
    CMymfc20Doc* pDoc = GetDocument();
    str.Format("Document %s", (LPCSTR) pDoc->GetTitle());
    pDC->TextOut(point.x, point.y, str);
    str.Format("Page %d", m_nPage);
    CSize size = pDC->GetTextExtent(str);
    point.x += 11520 - size.cx;
    pDC->TextOut(point.x, point.y, str); // right-justified
}
```

```
void CMymfc20View::PrintPageFooter(CDC *pDC)
{
    CString str;

    CPoint point(0, -14400); // Move 10 inches down
    CMymfc20Doc* pDoc = GetDocument();
    str.Format("Document %s", (LPCSTR) pDoc->GetTitle());
    pDC->TextOut(point.x, point.y, str);
    str.Format("Page %d", m_nPage);
    CSize size = pDC->GetTextExtent(str);
    point.x += 11520 - size.cx;
    pDC->TextOut(point.x, point.y, str); // right-justified
}
```

Listing 23.

Build and test the application. For one set of random numbers, the bubble view window looks something like this.

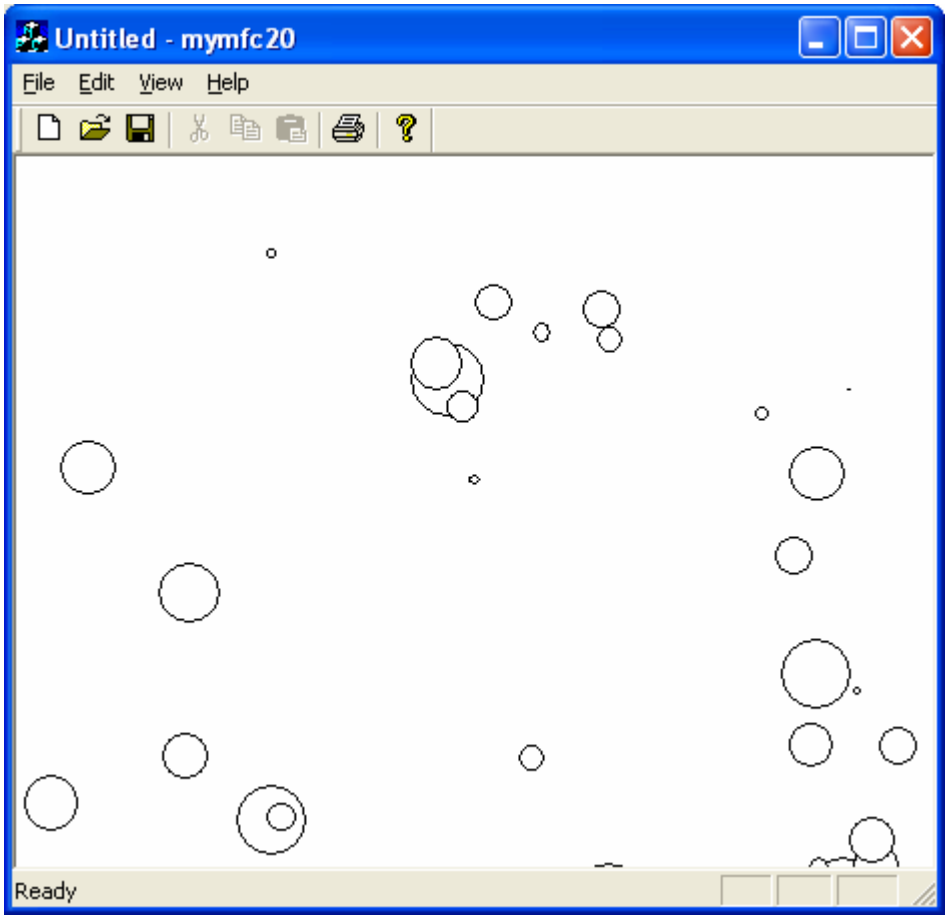

Figure 24: MYMFC20 program output in action.

Each time you choose **New** from the **File** menu, you should see a different picture. In **Print Preview**, the first page of the output should look like this.

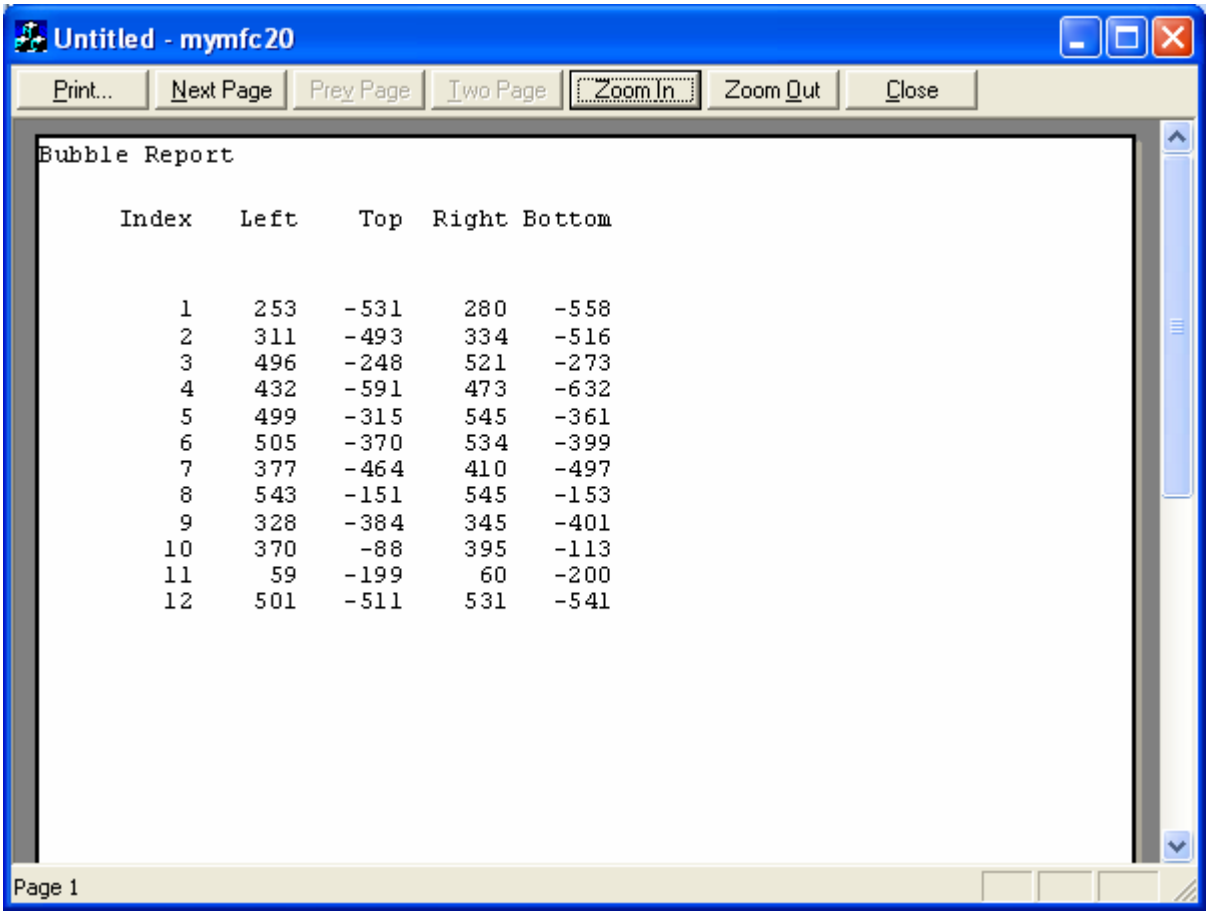

Figure 25: MYMFC20 **Print Preview** page.

With the **Print** dialog, you can specify any range of pages to print.

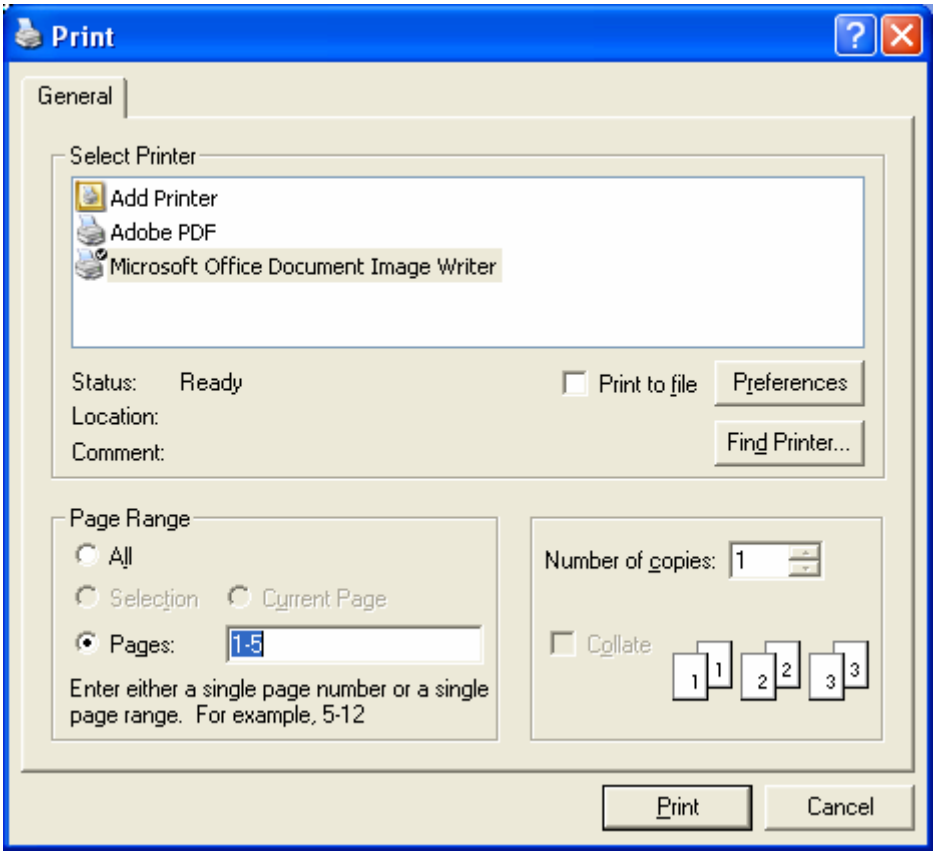

Figure 26: MYMFC20 with **Page Range** setting for printing.

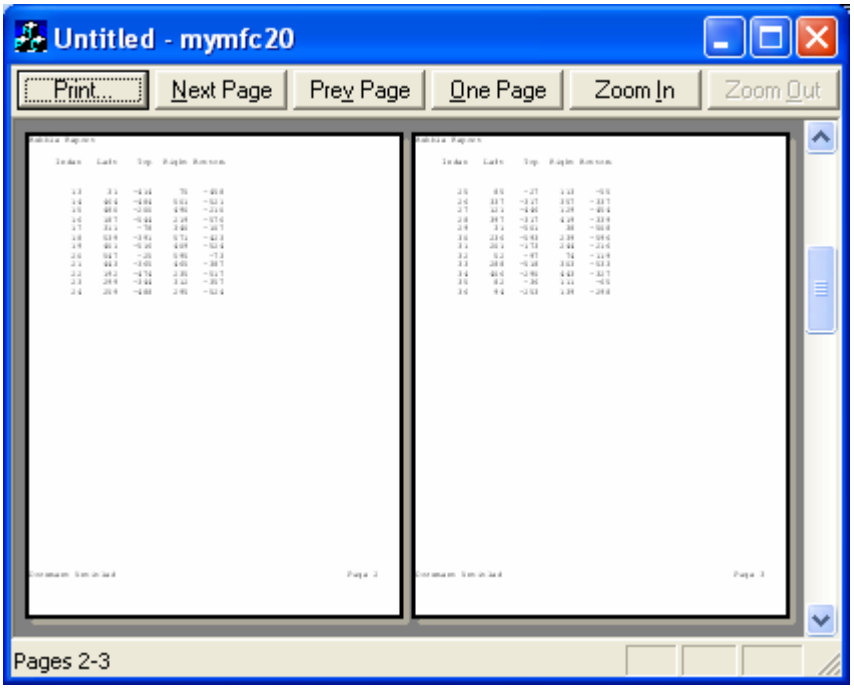

Figure 27: MYMFC20 with multiple pages of the **Print Preview**.

Link to LogScrollView.h and LogScrollView.cpp.

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.