Lab 7.3: Building a Text Viewer

Objectives

After completing this lab, you will be able to:

- Implement supporting code for a viewer.
- Display streams of text.
- Control fonts for on-screen display.

Prerequisites

Because you will be working with many member functions of CDC, you may find it helpful to review the Visual C++ Help topics on graphics before attempting this lab.

Lab Setup

To run the solution to this lab, click this icon.

To see a demonstration of the solution for this lab, click this icon.

Estimated time to complete this lab: **15 minutes**.

Exercises

The following exercises provide practice working with the concepts and techniques covered in this chapter.

Exercise 1: Implementing a Basic Text Viewer

In this exercise, you will add text display capability to a viewer.

Exercise 2: Adding Font Support

In this exercise, you will implement a user interface for text viewer you created in Exercise 1.

There is no setup for this lab. The completed code for these exercises is in \Labs\C07\Lab03\Xxx, where Xxx is the exercise number.

Exercise 1: Implementing a Basic Text Viewer

In this exercise, you will add text display capability to a viewer based on **CScrollView**. There are four parts to this exercise:

1. Creating an AppWizard MDI application
2. Adding file-handling to the document
3. Calculating the basic metrics of the view
4. Displaying the text

At the end of this exercise, you will have created an MDI text viewer with file selection, display, and scrolling.

Creating an AppWizard MDI Application

If you have not created an AppWizard MDI-based application before, please review the Chapter 5 Labs and follow the procedure described there, with the following exceptions:

- Name the new project workspace **Text**.
- In Step 6, derive **CTextView** from **CScrollView,** rather than **CView**.

Finish and create the new project. Build it at this point.

Adding File Handling to the Document

In Text.Exe, **CTextDoc** is little more than a holder for a **CStringList**, into which you will read the lines of a selected file. **CDocument** provides the menuing and the File Open dialog. You will provide the file-handling code and the parsing of the file into **CStringList**.

➤ Prepare CTextDoc for file reading

1. Open TextDoc.H.
2. Declare a protected **CStringList** member.

   ```
   CStringList m_LineList;
   ```

3. Declare a public member function to return a pointer to m_LineList.

   ```
   CStringList*GetLineList() { return &m_LineList; }
   ```

4. Save TextDoc.H.
5. CTextDoc::OnNewDocument is called when the application starts and when a user action occurs. Because Text.Exe is a read-only application, you will want to explicitly disable this function in TextDoc.Cpp.

   ```
   BOOL CTextDoc::OnNewDocument()
   {
       return FALSE;
   }
   ```

➤ Implement OnOpenDocument

1. CTextDoc::OnOpenDocument will read the selected text file, line by line, into the CStringList member. OnOpenDocument is called from the application after CTextApp has queried the user with a File Open dialog box. Create CTextDoc::OnOpenDocument from ClassWizard or WizardBar. Edit the code to remove the default handler.
2. Reading a file into memory one line at a time could take a while. Show the wait cursor with **CCmdTarget::BeginWaitCursor**.

   ```
   BeginWaitCursor();
   ```

3. Clear all the items from m_LineList.

   ```
   m_LineList.RemoveAll();
   ```

4. CStdioFile provides stream-oriented file access with line-oriented file access. Open the file passed in lpszPathName by constructing a CStdioFile object.

   ```
   CStdioFile  file(lpszPathName,
                   CFile::modeRead | CFile::typeText);
   ```

5. You will need a **CString** into which to read each line.

   ```
   CString strLine;
   ```

6. CStdioFile::ReadString returns TRUE if anything was read and FALSE if the end of the file was encountered before reading any data. Read data as long as there is anything to read.

   ```
   while (file.ReadString(strLine) != NULL)
   {
   ```

7. You will want to clean up the ends of the lines for white space and control characters.

```
int nLastCharIndex = strLine.GetLength()-1;
        while (nLastCharIndex >= 0 && strLine[nLastCharIndex] < ' ')
        {
```

```
            strLine.SetAt(nLastCharIndex--, '\0');
        }
```

8. Once the string is clean, add it to the end of m_LineList.

```
m_LineList.AddTail(strLine);
```

9. At the end of the read loop, restore the cursor.

```
EndWaitCursor();
```

10. Return TRUE to indicate that you have handled the message.

```
return TRUE;
```

11. Save TextDoc.Cpp.

The completed function is shown in the following sample code.

```
BOOL CTextDoc::OnOpenDocument(LPCTSTR lpszPathName)
{
    //  Could be a big file
    BeginWaitCursor();

    //  Clear List, this will cleanup the CString objects
    m_LineList.RemoveAll();

    //  Read the file and store as a list
    //  of CStrings
    CStdioFile  file(lpszPathName,
                    CFile::modeRead | CFile::typeText);


    CString strLine;
    while (file.ReadString(strLine) != NULL)
    {
        //remove the noise characters at the end of the line
        int nLastCharIndex = strLine.GetLength()-1;
        while (nLastCharIndex >= 0 && strLine[nLastCharIndex] < ' ')
        {
        strLine.SetAt(nLastCharIndex--, '\0');
        }

        //  Add to CStringList
        m_LineList.AddTail(strLine);
    }

    EndWaitCursor();
    return TRUE;
}
```

➢ **Save TextDoc.H and TextDoc.Cpp**

Calculating the Basic Metrics of the View

➢ **Declare the basic metrics members**

You will need to have a number of basic metrics for the text view. These do not need to be calculated each time you draw text on the screen if you hold them in member variables.

1. Right-click **CTextView** in ClassView and add the following protected variables.

```
CSize   m_ViewCharSize
CSize   m_DocSize
CFont*  m_pFont
```

2. Right-click **CTextView** in ClassView and declare a public function as follows.

```
CFont*    GetFont()
```

3. Right-click **CTextView** in ClassView and declare a protected function as follows.

```
void ComputeViewMetrics()
```

4. Add public member functions manually to CtextView.h as follows.

```
CSize       GetDocSize() const { return m_DocSize; }
CSize       GetCharSize() conts { return m_ViewCharSize; }
```

5. Open Textview.Cpp.
6. Initialize m_ViewCharSize, m_DocSize, and m_pFont in the constructor. The constructor looks like the following:

```
CTextView:: CTextView()
    :   m_ViewCharSize(0,0),
        m_DocSize(0,0)
{
    m_pFont = NULL;
}
```

7. Save TextView.Cpp.

➤ **Get the current font**

1. In TextView.Cpp, define GetFont.

```
CFont*  CTextView::GetFont()
```

2. If no font has been created, construct a new font.

```
if(m_pFont == NULL)
{
    m_pFont = new Cfont;
```

3. Create a nine-point Arial font in m_pFont.

```
if(m_pFont)
{
    //  Default to 9 pt Arial
    m_pFont->CreatePointFont(90, "Arial");
}
```

4. Return m_pFont. Save TextView.Cpp.

The complete function is shown in the following sample code.

```
CFont * CTextView::GetFont()
{

    if(m_pFont == NULL)
    {
        m_pFont = new CFont;
        if(m_pFont)
        {
            //  Default to 9 pt Arial
            m_pFont->CreatePointFont(90, "Arial");
        }
    }
    return m_pFont;
}
```

➤ **Compute the basic metrics of the view by defining ComputeViewMetrics**

1. Get the pointer to the screen DC and save the state of the DC.

   ```
   CDC* pDC = CDC::FromHandle(::GetDC(NULL));
       int nSaveDC = pDC->SaveDC();
   ```

2. Set the mapping mode to MM_LOENGLISH.

   ```
   pDC->SetMapMode(MM_LOENGLISH);
   ```

3. Select the display font into the DC and get the font's text metrics.

   ```
   CFont* pPreviousFont = pDC->SelectObject(GetFont());
   TEXTMETRIC tm;
   pDC->GetTextMetrics(&tm);
   ```

4. The actual height of a font element (character) is the sum of its internal height (tmHeight) and the space between lines (tmExternalLeading).

   ```
   m_ViewCharSize.cy = tm.tmHeight + tm.tmExternalLeading;
   m_ViewCharSize.cx = tm.tmAveCharWidth;
   ```

5. Convert the character size to device units (pixels, on the screen).

   ```
   pDC->LPtoDP(&m_ViewCharSize);
   ```

6. Get a pointer to a document so you can access the **CStringList** member that holds the data.

   ```
   CTextDoc* pDoc = GetDocument();
   ```

7. Initialize the document width to 0. To calculate the document height, you simply multiply the number of lines by the height of a line.

   ```
   m_DocSize.cx = 0;
   m_DocSize.cy = m_ViewCharSize.cy *
                   pDoc->GetLineList()->GetCount();
   ```

8. The longest line has to be calculated by looking at each line of the document using the current font. Declare variables for a loop to interrogate each line.

   ```
   CString Line;
   CSize size;
   ```

9. Because **CStringList** is a collection with an iterator, you will use a POSITION to iterate through the list.

   ```
   POSITION pos = pDoc->GetLineList()->GetHeadPosition();
   while( pos != NULL )
   ```

10. Get the current line, and from it get its text extent.

    ```
    Line = pDoc->GetLineList()->GetNext( pos );
    size = pDC->GetTextExtent(Line, Line.GetLength());
    ```

11. Set the width of the document to the largest size found.

    ```
    m_DocSize.cx = max(size.cx, m_DocSize.cx);
    ```

12. After the loop is closed, add a four-pixel margin.

```
m_DocSize.cx += 4 * m_ViewCharSize.cx;
```

13. Select the application font out of the DC.

```
if(pPreviousFont)
    {
    pDC->SelectObject(pPreviousFont);
    }
```

14. Restore the DC to its original state.

```
pDC->RestoreDC(nSaveDC);
```

15. Release the DC.

```
::ReleaseDC(NULL, pDC->GetSafeHdc());
```

16. Save TextView.Cpp.

The complete function is shown in the following sample code.

```
void CTextView::ComputeViewMetrics()
{

    // get a CDC* for the screen
    CDC* pDC = CDC::FromHandle(::GetDC(NULL));
    int nSaveDC = pDC->SaveDC();


    // select mapping mode
    pDC->SetMapMode(MM_LOENGLISH);

    // select the font and get its metrics
    CFont* pPreviousFont = pDC->SelectObject(GetFont());
    TEXTMETRIC tm;
    pDC->GetTextMetrics(&tm);

    //  Calculate view character size
    m_ViewCharSize.cy = tm.tmHeight + tm.tmExternalLeading;
    m_ViewCharSize.cx = tm.tmAveCharWidth;

    // convert to device units to minimize round off error
    pDC->LPtoDP(&m_ViewCharSize);

    //  Calculate document size
    CTextDoc* pDoc = GetDocument();
    m_DocSize.cy = m_ViewCharSize.cy *
                    pDoc->GetLineList()->GetCount();

    // loop through the document and find the longest line
    CString Line;
    CSize size;
    POSITION pos = pDoc->GetLineList()->GetHeadPosition();
    while( pos != NULL )
    {
        Line = pDoc->GetLineList()->GetNext( pos );
        size = pDC->GetTextExtent(Line, Line.GetLength());
        m_DocSize.cx = max(size.cx, m_DocSize.cx);
    }
    //  Account for our simple margin
    m_DocSize.cx += 4 * m_ViewCharSize.cx;

    // clean up
    if(pPreviousFont)
    {
        pDC->SelectObject(pPreviousFont);
    }
    pDC->RestoreDC(nSaveDC);
    ::ReleaseDC(NULL,pDC->GetSafeHdc());
}
```

➢ **Save TextView.H and TextView.Cpp**

Displaying the Text

When you have very small files, they can fit into the view window. In this case, your task would be simply to draw the lines in the file to the screen, one line at a time. With larger files, however, only part of the file can be displayed at one time. With files of less than a few hundred lines in length, you could draw all the text into the view as if it were all visible. The general solution is to calculate the lines that can fit into the window and paint those lines. In addition, you will need to process the OnUpdate message that is sent when the view window is resized.

➢ **Implement OnDraw**

1. Move to the top of **CTextView::OnDraw**. Delete the contents of the function and declare variables.

```
int nFirstLn, nLastLn;
```

2. Calculate the lines to draw by calling **ComputeVisibleLines**.

```
ComputeVisibleLines(pDC, nFirstLn, nLastLn);
```

3. Calculate the position of the first line, relative to the origin of the window.

```
int nYPos = - nFirstLn * GetCharSize().cy;
int nXPos = 4 * GetCharSize().cx;
```

4. Call the core **OnDraw** handler.

```
OnDraw(pDC, nFirstLn, nLastLn,nXPos,nYPos);
```

5. Save TextView.Cpp.

The complete function is shown in the following sample code.

```
void CTextView::OnDraw(CDC* pDC)
{
    int nFirstLn, nLastLn;

    ComputeVisibleLines(pDC, nFirstLn, nLastLn);

    int nYPos = - nFirstLn * GetCharSize().cy;
    int nXPos = 4 * GetCharSize().cx;
    OnDraw(pDC, nFirstLn, nLastLn,nXPos,nYPos);
}
```

➢ **Implement ComputeVisibleLines**

1. Right-click **CTextView** in ClassView and add **ComputeVisibleLines** as a protected function:

```
void ComputeVisibleLines(CDC* pDC, int& nFirst, int& nLast)
```

2. Begin coding the **ComputeVisibleLines** function by getting the number of lines in the **CStringList**.

   ```
   int nLineCount = GetDocument()->GetLineList()->GetCount();
   ```

3. Get the viewport origin, in logical coordinates.

   ```
   CPoint pt = pDC->GetViewportOrg();
   pDC->DPtoLP(&pt,1);
   ```

4. Get the clipping region, in logical coordinates.

   ```
   CRect rc;
   pDC->GetClipBox(&rc);
   ```

5. Get the line height.

   ```
   CSize CharSize = GetCharSize();
   ```

6. The algorithm for the first visible line accomplishes these points, and is as follows.
   a. Calculate the distance from the top of the viewport to the top of clipping region.
   b. Divide this distance by the height of a line, giving the number of lines.
   c. Ensure that at least one line will be shown.

   ```
   nFirst = min(abs((rc.top - pt.y)/CharSize.cy),
               nLineCount-1);
   ```

7. The algorithm for the last visible line accomplishes these points, and is as follows:
   a. Calculate the number of lines that will fit into the clipping region.
   b. Add that to the starting line.
   c. Add one more line to make sure that partial lines are displayed.
   d. Ensure that this is less than the total number of lines.

   ```
   nLast = min(abs(rc.Height())/CharSize.cy + nFirst + 1,
               nLineCount-1);
   ```

8. Save TextView.Cpp.

The complete function is shown in the following sample code.

```
void CTextView::ComputeVisibleLines(CDC* pDC, int& nFirst, int& nLast)
{
    int nLineCount = GetDocument()->GetLineList()->GetCount();

    // Get the viewport origin, convert to logical coordinates
    CPoint pt = pDC->GetViewportOrg();
    pDC->DPtoLP(&pt,1);

    // Get the clipping region, in logical coordinates
    CRect rc;
    pDC->GetClipBox(&rc);

    // Get the logical line height
    CSize CharSize = GetCharSize();

    // Compute the first visible line
    nFirst = min(abs((rc.top - pt.y)/CharSize.cy),
                nLineCount-1);

    // compute the last visible line
    nLast = min(abs(rc.Height())/CharSize.cy + nFirst + 1,
```

```
                nLineCount-1);
}
```

➤ **Implement the core OnDraw handler**

1. Declare a second OnDraw handler in TextView.H.

   ```
   virtual void OnDraw(CDC* pDC, int nFirstLn, int nLastLn,
                       int nXPos = 0, int nYPos = 0);
   ```

2. Define the second OnDraw handler in TextView.Cpp.

   ```
   void CTextView::OnDraw(CDC* pDC, int nFirstLn, int nLastLn,
                          int nXPos /*= 0*/, int nYPos /*= 0*/)
   ```

3. Select your chosen font into the DC.

   ```
   CFont* pPreviousFont = pDC->SelectObject(GetFont());
   ```

4. Get the size of the font.

   ```
   CSize CharSize = GetCharSize();
   ```

5. Get the string list from the document.

   ```
   CStringList *pLineList = GetDocument()->GetLineList();
   ```

6. You will loop through the lines in pLineList from the first line passed (which will be an index) to the last line passed, drawing the text on the screen and moving down the screen (that is, to lower Y-coordinate values). Declare the necessary variables.

   ```
   CString strLine;
   POSITION    pos;
   ```

7. Control the loop.

   ```
   while (nFirstLn <= nLastLn)
   ```

8. Find the item in the list.

   ```
   if( ( pos = pLineList->FindIndex( nFirstLn )) != NULL )
   ```

9. If you have a valid item, copy it to the string and display it.

   ```
   strLine = pLineList->GetAt(pos);
   pDC->TabbedTextOut(nXPos, nYPos, strLine, 0, NULL, 0);
   ```

10. Decrement the Y-coordinate and increment the line count.

    ```
    nYPos -= CharSize.cy;
    nFirstLn++;
    ```

11. Back outside the loop, select your font out of the DC.

    ```
    if(pPreviousFont)
    {
        pDC->SelectObject(pPreviousFont);
    }
    ```

12. Save TextView.Cpp.

The complete function is shown in the following sample code.

```
void CTextView::OnDraw(CDC* pDC, int nFirstLn, int nLastLn,
                        int nXPos /*= 0*/, int nYPos /*= 0*/)
{
```

```
    //  Select specified font
    CFont* pPreviousFont = pDC->SelectObject(GetFont());

    //  Needed for height of each line
    CSize CharSize = GetCharSize();

    //  Get list of strings from the document
    //  and output them to the display context
    CStringList *pLineList = GetDocument()->GetLineList();

    CString strLine;
    POSITION    pos;
    while (nFirstLn <= nLastLn)
    {
        if( ( pos = pLineList->FindIndex( nFirstLn )) != NULL )
        {
            strLine = pLineList->GetAt(pos);
            pDC->TabbedTextOut(nXPos, nYPos, strLine, 0, NULL, 0);
            nYPos -= CharSize.cy;
            nFirstLn++;
        }
    }

    //  Cleanup and restore original GDI Objects
    if(pPreviousFont)
    {
        pDC->SelectObject(pPreviousFont);
    }
}
```

➤ **Implement OnUpdate**

1. Using ClassWizard, delete **CTextView::OnInitialUpdate** and manually remove the associated code.
2. Using ClassWizard or the WizardBar, add **CTextView::OnUpdate**. Edit the code to compute the view metrics.

   ```
   ComputeViewMetrics();
   ```

3. CScrollView::SetScrollSizes sets the mapping mode for the scroll view. Because CTextView::ComputeViewMetrics uses MM_LOENGLISH for its calculations, you will need to use that mode here. It also sets the scrolling ranges. Get these ranges from the document size.

   ```
   SetScrollSizes( MM_LOENGLISH, GetDocSize());
   ```

4. CView::OnUpdate is sent whenever the view window is changed. Invalidate the window so that CTextView::OnDraw will be called to appropriately redisplay the text.

   ```
   Invalidate();
   ```

5. Save TextView.Cpp.

The complete function is shown in the following sample code.

```
void CTextView::OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint)
{
    ComputeViewMetrics();

    SetScrollSizes(MM_LOENGLISH, GetDocSize());
    Invalidate();
}
```

➢ **Save TextView.H and TextView.Cpp**

➢ **Build and run Text.Exe**

The completed code for this exercise is in \Labs\C07\Lab03\Ex01.

Exercise 2: Adding Font Support

Continue with the files you created in Exercise 1 or, if you do not have a starting point for this exercise, the code that forms the basis for this exercise is in \Labs\C07\Lab03\Ex01.

Most font support for the text viewer is already in place. You have only to implement a user interface to **CTextView.m_pFont**.

➢ **Add a menu item for Font**

1. Open the IDR_TEXTTYPE menu resource.
2. Add a menu between the View and Window menus. Give it the caption **Font**.
3. Add a menu item below the font menu, giving it the ID **ID_FORMAT_FONT** and the prompt, **Change Font**.
4. Save Text.Rc.

➢ **Add a handler for the menu message**

Use ClassWizard or WizardBar. Add a handler for ID_FORMAT_FONT to **CTextView**, and accept the default **OnFormatFont** function name.

➢ **Implement CTextView::OnFormatFont**

1. Go to the head of **OnFormatFont**. Get (or create, if it has not yet been created) the current font with **GetFont**.

```
CFont * pFont = GetFont();
```

2. Retrieve a LOGFONT structure with the font information.

```
LOGFONT lf;
pFont->GetObject(sizeof(LOGFONT), &lf);
```

3. Use this structure to initialize a common font dialog box.

```
CFontDialog dlg(&lf, CF_SCREENFONTS | CF_INITTOLOGFONTSTRUCT);
```

4. Show it modally.

```
if(dlg.DoModal() == IDOK)
```

5. If the user closed the font dialog box with OK, delete the current font (remember that this font is selected out of a DC each time it is selected in, so this deletion is safe).

```
if(m_pFont)
{
    delete m_pFont;
}
```

6. Construct the new font and initialize it using the LOGFONT returned from **CFontDialog**.

```
m_pFont = new CFont;
if(m_pFont)
{
    m_pFont->CreateFontIndirect(&lf);
}
```

7. Finally, you must recalculate all the metrics and invalidate the view's window to redisplay the file using the new font. You already have a function that does this, **OnUpdate**, but it will take some significant code to set up the call. The simplest way to accomplish this is to call **CDocument::UpdateAllViews**. Because there is only one view of this document, this will be the equivalent of calling **OnUpdate** directly.

   ```
   GetDocument()->UpdateAllViews(NULL);
   ```

8. Save TextView.Cpp.

The complete function is shown in the following sample code.

```
void CTextView::OnFormatFont()
{

    CFont * pFont = GetFont();

    LOGFONT lf;
    pFont->GetObject(sizeof(LOGFONT), &lf);

    CFontDialog dlg(&lf, CF_SCREENFONTS | CF_INITTOLOGFONTSTRUCT);

    if(dlg.DoModal() == IDOK)
    {
        if(m_pFont)
        {
            delete m_pFont;
        }

        m_pFont = new CFont;
        if(m_pFont)
        {
            m_pFont->CreateFontIndirect(&lf);
        }

        //  This will cause OnUpdate() to be called ensuring
        //  that our cached metrics and scrolling get updated
        GetDocument()->UpdateAllViews(NULL);
    }
}
```

The completed code for this exercise is in \Labs\C07\Lab03\Ex02.