Module 7: Menus, Keyboard Accelerators, the Rich Edit Control, and Property Sheets – Part 1

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

The Document View Architecture
Menus, Keyboard Accelerators, the Rich Edit Control, and Property Sheets
The Main Frame Window and Document Classes
Windows Menus
Keyboard Accelerators
Command Processing
Command Message Handling in Derived Classes
Update Command User Interface Handlers
Commands That Originate in Dialogs
The Application Framework's Built-In Menu Items
Enabling/Disabling Menu Items
MFC Text Editing Options
The `CEditView` Class
The `CRichEditView` Class
The `CRichEditCtrl` Class
The **MYMFCPRO Program Example**

The Document View Architecture

Menus, Keyboard Accelerators, the Rich Edit Control, and Property Sheets

In all the book's examples to this point, mouse clicks have triggered most program activity. Even though menu selections might have been more appropriate, you've used mouse clicks because mouse-click messages are handled simply and directly within the MFC Library version 6.0 view window. If you want program activity to be triggered when the user chooses a command from a menu, you must first become familiar with the other application framework elements.

This module concentrates on menus and the command routing architecture. Along the way, we introduce frames and documents, explaining the relationships between these new application framework elements and the already familiar view element. You'll use the menu editor to lay out a menu visually, and you'll use ClassWizard to link **document** and **view** member functions to menu items. You'll learn how to use special update command user interface (UI) member functions to check and disable menu items, and you'll see how to use keyboard accelerators as menu shortcut keys. Because you're probably tired of circles and dialogs, next you'll examine two new MFC building blocks. The rich edit common control can add powerful text editing features to your application. Property sheets are ideal for setting edit options.

The Main Frame Window and Document Classes

Up to now, you've been using a view window as if it were the application's only window. In an SDI application, the view window sits inside another window: the application's main frame window. The main frame window has the title bar and the menu bar. Various child windows, including the toolbar window, the view window, and the status bar window, occupy the main frame window's **client area**, as shown in Figure 2 and Figure 1 is a WordPad, provided as a comparison. The application framework controls the interaction between the **frame** and the **view** by routing messages from the frame to the view.

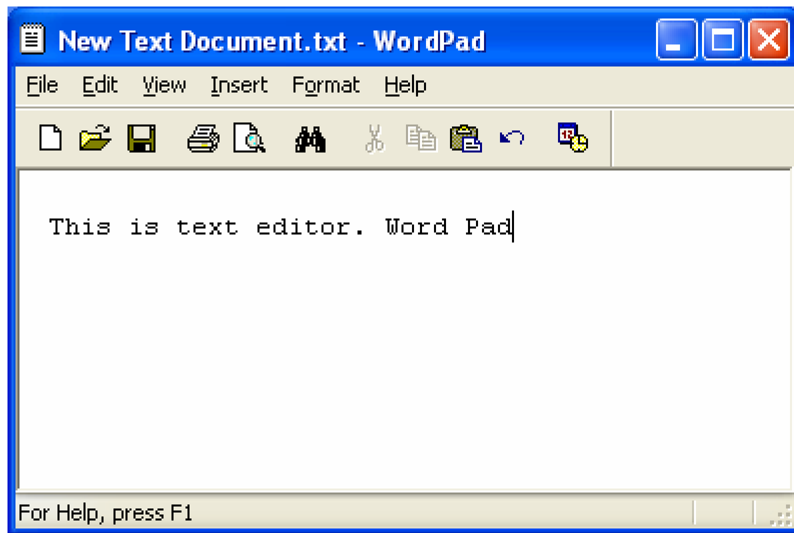

Figure 1: WordPad as a real application.

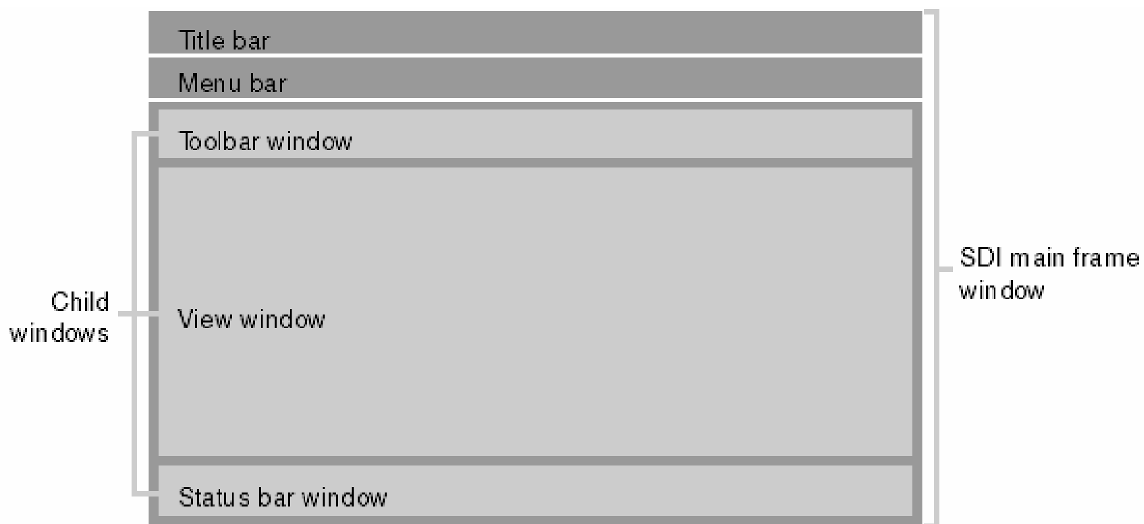

Figure 2: The child windows within an SDI main frame window.

Look again at any project files generated by AppWizard. The **MainFrm.h** and **MainFrm.cpp** files contain the code for the application's main frame window class, derived from the class `CFrameWnd`. Other files, with names such as **MymfcproDoc.h** and **MymfcproDoc.cpp**, contain code for the application's document class, which is derived from `CDocument`. In this module you'll begin working with the MFC document class. You'll start by learning that each view object has exactly one document object attached and that the view's inherited `GetDocument()` member function returns a pointer to that object.

Windows Menus

A Microsoft Windows menu is a familiar application element that consists of a top-level horizontal list of items with associated pop-up menus that appear when the user selects a top-level item. Most of the time, you define for a frame window a default menu resource that loads when the window is created. You can also define a menu resource independent of a frame window. In that case, your program must call the functions necessary to load and activate the menu.

A menu resource completely defines the initial appearance of a menu. Menu items can be grayed or have check marks, and bars can separate groups of menu items. Multiple levels of pop-up menus are possible. If a first-level menu item is associated with a subsidiary pop-up menu, the menu item carries a right-pointing arrow symbol, as shown next to the **Start Debug** menu item in Figure 3.

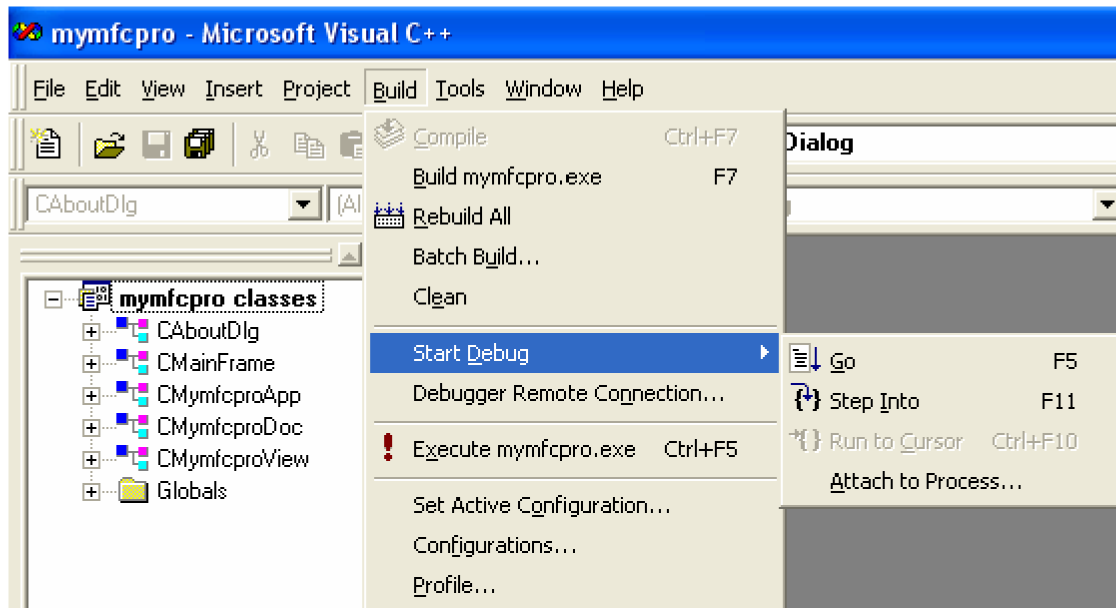

Figure 3: Multilevel pop-up menus example (from Microsoft Visual C++).

Visual C++ includes an easy-to-use menu-resource editing tool. This tool lets you edit menus in a wysiwyg environment. Each menu item has a properties dialog that defines all the characteristics of that item. The resulting resource definition is stored in the application's resource script (RC) file. Each menu item is associated with an ID, such as `ID_FILE_OPEN`, that is defined in the resource.h file.
The MFC library extends the functionality of the standard menus for Windows. Each menu item can have a prompt string that appears in the frame's status bar when the item is highlighted. These prompts are really Windows string resource elements linked to the menu item by a common ID. From the point of view of the menu editor and your program, the prompts appear to be part of the menu item definition.

Keyboard Accelerators

You've probably noticed that most menu items contain an underlined letter. In Visual C++ (and most other applications), pressing **Alt-F** followed by **S** activates the File Save menu item. This **shortcut system** is the standard Windows method of using the keyboard to choose commands from menus. If you look at an application's menu resource script (or the menu editor's properties dialog), you will see an ampersand (**&**) preceding the character that is underlined in each of the application's menu items.
Windows offers an alternative way of linking keystrokes to menu items. The keyboard accelerator resource consists of a table of key combinations with associated command IDs. The Edit Copy menu item (with command ID `ID_EDIT_COPY`), for example, might be linked to the Ctrl-C key combination through a keyboard accelerator entry. A keyboard accelerator entry does not have to be associated with a menu item. If no Edit Copy menu item were present, the Ctrl-C key combination would nevertheless activate the `ID_EDIT_COPY` command.

Command Processing

The MFC application framework provides a sophisticated routing system for command messages. These messages originate from menu selections, keyboard accelerators, and toolbar and dialog button clicks. Command messages can also be sent by calls to the `CWnd::SendMessage` or `PostMessage()` function. Each message is identified by a `#define` constant that is often assigned by a resource editor. The application framework has its own set of internal command message IDs, such as `ID_FILE_PRINT` and `ID_FILE_OPEN`. Your project's resource.h file contains IDs that are unique to your application.
Most command messages originate in the application's frame window, and without the application framework in the picture, that's where you would put the message handlers. With command routing, however, you can handle a message almost anywhere. When the application framework sees a frame window command message, it starts looking for message handlers in one of the sequences listed here.

SDI Application	MDI Application
View	View
Document	Document
SDI main frame window	MDI child frame window
Application	MDI main frame window Application

Table 1: SDI vs MDI.

Most applications have a particular command handler in only one class, but suppose your one-view application has an identical handler in both the view class and the document class. Because the view is higher in the command route, only the view's command handler function will be called.

What is needed to install a command handler function? The installation requirements are similar to those of the window message handlers you've already seen. You need the function itself, a corresponding message map entry, and the function prototype. Suppose you have a menu item named **Zoom** (with `IDM_ZOOM` as the associated ID) that you want your view class to handle. First you add the following code to your view implementation file:

```
BEGIN_MESSAGE_MAP(CMyView, CView)
    ON_COMMAND(IDM_ZOOM, OnZoom)
END_MESSAGE_MAP()

void CMyView::OnZoom()
{
    // command message processing code
}
```

Now add the following function prototype to the `CMyView` class header file (before the `DECLARE_MESSAGE_MAP` macro):

```
afx_msg void OnZoom();
```

Of course, ClassWizard automates the process of inserting command message handlers the same way it facilitates the insertion of window message handlers. You'll learn how this works in the next example, MYMFCPRO.

Command Message Handling in Derived Classes

The command routing system is one dimension of command message handling. The class hierarchy is a second dimension. If you look at the source code for the MFC library classes, you'll see lots of `ON_COMMAND` message map entries. When you derive a class from one of these base classes, for example, `CView`, the derived class inherits all the `CView` message map functions, including the command message functions. To override one of the base class message map functions, you must add both a function and a message map entry to your derived class.

Update Command User Interface Handlers

You often need to change the appearance of a menu item to match the internal state of your application. If your application's **Edit** menu includes a **Clear All** item, for example, you might want to disable that item if there's nothing to clear. You've undoubtedly seen such **grayed menu** items in Windows-based applications, and you've probably also seen check marks next to menu items.

The MFC library takes a different approach by calling a special update command **user interface** (UI) handler function whenever a pop-up menu is first displayed. The handler function's argument is a `CCmdUI` object, which contains a pointer to the corresponding menu item. The handler function can then use this pointer to modify the menu item's appearance. Update command UI handlers apply only to items on pop-up menus, not to top-level menu items that are permanently displayed. You can't use an update command UI handler to disable a File menu item, for example.

The update command UI coding requirements are similar to those for commands. You need the function itself, a special message map entry, and of course the prototype. The associated ID, in this case, `IDM_ZOOM`, is the same constant used for the command. Here is an example of the necessary additions to the view class code file:

```
BEGIN_MESSAGE_MAP(CMyView, CView)
    ON_UPDATE_COMMAND_UI(IDM_ZOOM, OnUpdateZoom)
END_MESSAGE_MAP()

void CMyView::OnUpdateZoom(CCmdUI* pCmdUI)
{
    pCmdUI->SetCheck(m_bZoomed); // m_bZoomed is a class data member
}
```

Here is the function prototype that you must add to the class header (before the `DECLARE_MESSAGE_MAP` macro):

```
afx_msg void OnUpdateZoom(CCmdUI* pCmdUI);
```

Needless to say, ClassWizard automates the process of inserting update command UI handlers.

Commands That Originate in Dialogs

Suppose you have a pop-up dialog with buttons, and you want a particular button to send a command message. Command IDs must be in the range **`0x8000`** to **`0xDFFF`**, the same ID range that the resource editor uses for your menu items. If you assign an ID in this range to a dialog button, the button will generate a routable command. The application framework first routes this command to the main frame window because the frame window owns all pop-up dialogs. The command routing then proceeds normally; if your view has a handler for the button's command, that's where it will be handled. To ensure that the ID is in the range `0x8000` to `0xDFFF`, you must use Visual C++'s symbol editor to enter the ID prior to assigning the ID to a button.

The Application Framework's Built-In Menu Items

You don't have to start each frame menu from scratch; the MFC library defines some useful menu items for you, along with all the command handler functions, as shown in Figure 4.

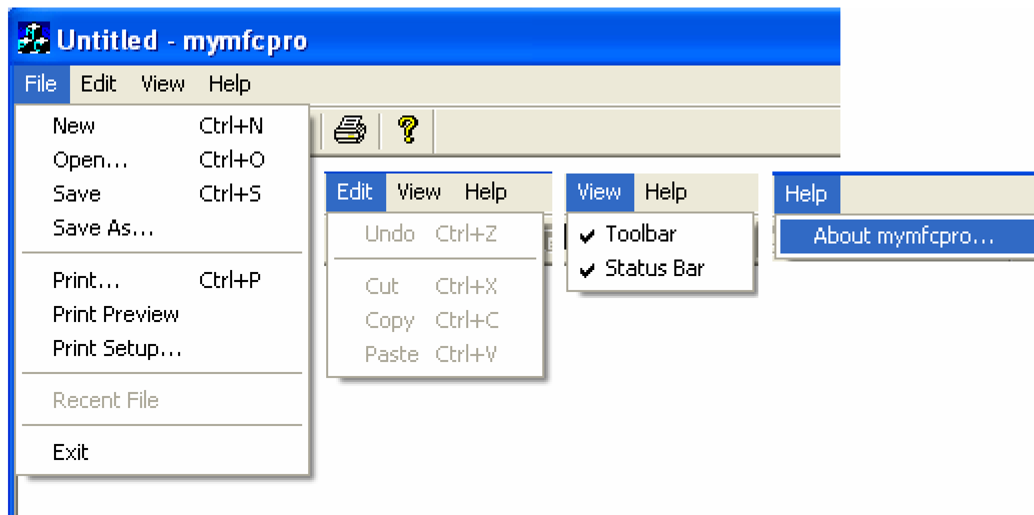

Figure 4: The standard SDI frame menus.

The menu items and command message handlers that you get depend on the options you choose in AppWizard. If you deselect **Printing and Print Preview**, for example, the **Print and Print Preview** menu items don't appear. Because printing is optional, the message map entries are not defined in the `CView` class but are generated in your derived view class. That's why entries such as the following are defined in the `CMyView` class instead of in the `CView` class:

```
ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
```

Enabling/Disabling Menu Items

The application framework can disable a menu item if it does not find a command message handler in the current command route. This feature saves you the trouble of having to write ON_UPDATE_COMMAND_UI handlers. You can disable the feature if you set the CFrameWnd data member m_bAutoMenuEnable to FALSE.
Suppose you have two views for one document, but only the first view class has a message handler for the IDM_ZOOM command. The **Zoom** item on the frame menu will be enabled only when the first view is active. Or consider the application framework-supplied **Edit**, **Cut**, **Copy**, and **Paste** menu items. These will be disabled if you have not provided message handlers in your derived view or document class.

MFC Text Editing Options

Windows itself supplies two text editing tools: edit control and Windows rich edit common control. Both can be used as controls within dialogs, but both can also be made to look like view windows. The MFC library supports this versatility with the CEditView and CRichEditView classes.

The CEditView Class

This class is based on the Windows edit control, so it inherits all the edit control's limitations. Text size is limited to 64 KB, and you can't mix fonts. AppWizard gives you the option of making CEditView the base class of your view class. When the framework gives you an edit view object, it has all the functionality of both CView and CEdit. There's no multiple inheritance here, just some magic that involves window subclassing. The CEditView class implements and maps the clipboard cut, copy, and paste functions, so they appear active on the **Edit** menu.

The CRichEditView Class

This class uses the rich edit control, so it supports mixed formats and large quantities of text. The CRichEditView class is designed to be used with the CRichEditDoc and CRichEditCntrItem classes to implement a complete ActiveX container application.

The CRichEditCtrl Class

This class wraps the rich edit control, and you can use it to make a fairly decent text editor. That's exactly what we'll do in the MYMFCPRO example. We'll use an ordinary view class derived from CView, and we'll cover the view's client area with a big rich edit control that resizes itself when the view size changes. The CRichEditCtrl class has dozens of useful member functions, and it picks up other functions from its CWnd base class. The functions we'll use in this module are as follows.

Function	Description
Create()	Creates the rich edit control window (called from the parent's WM_CREATE handler)
SetWindowPos()	Sets the size and position of the edit window (sizes the control to cover the view's client area)
GetWindowText()	Retrieves plain text from the control (other functions available to retrieve the text with rich text formatting codes)
SetWindowText()	Stores plain text in the control
GetModify()	Gets a flag that is TRUE if the text has been modified (text modified if the user types in the control or if the program calls SetModify(TRUE))
SetModify()	Sets the modify flag to TRUE or FALSE
GetSel()	Gets a flag that indicates whether the user has selected text
SetDefaultCharFormat()	Sets the control's default format characteristics
SetSelectionCharFormat()	Sets the format characteristics of the selected text

Table 1: Functions used in MYMFCPRO project.

If you use the dialog editor to add a rich edit control to a dialog resource, your application class InitInstance() member function must call the function AfxInitRichEdit().

Program Example

This example illustrates the routing of menu and keyboard accelerator commands to both documents and views. The application's view class is derived from CView and contains a rich edit control. View-directed menu commands, originating from a new pop-up menu named Transfer, move data between the view object and the document object, and a **Clear Document** menu item erases the document's contents. On the **Transfer** menu, the **Store Data In Document** item is grayed when the view hasn't been modified since the last time the data was transferred. The **Clear Document** item, located on the **Edit** menu, is grayed when the document is empty. Figure 5 shows the first version of the MYMFCPRO program in use.

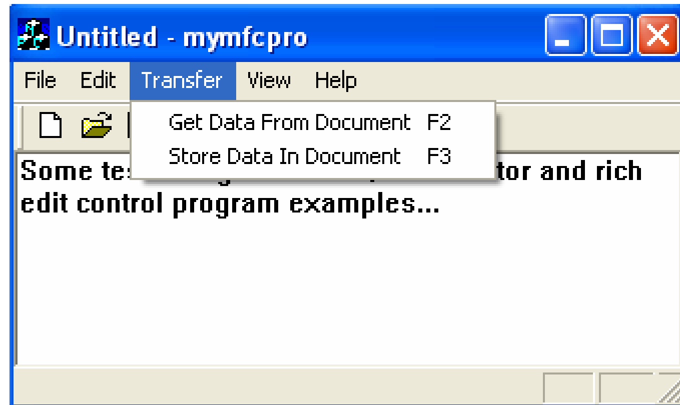

Figure 5: The MYMFCPRO program in use.

If we exploited the document-view architecture fully, we would tell the rich edit control to keep its text inside the document, but that's rather difficult to do. Instead, we'll define a document `CString` data member named `m_strText`, the contents of which the user can transfer to and from the control. The initial value of `m_strText` is a `Hello` message; choosing Clear Document from the Edit menu sets it to empty. By running this example, you'll start to understand the **separation of the document and the view**.
The first part of the MYMFCPRO example exercises Visual C++'s wysiwyg menu editor and keyboard accelerator editor together with ClassWizard. You'll need to do very little C++ coding. Simply follow these steps:

Run AppWizard as in the following steps.

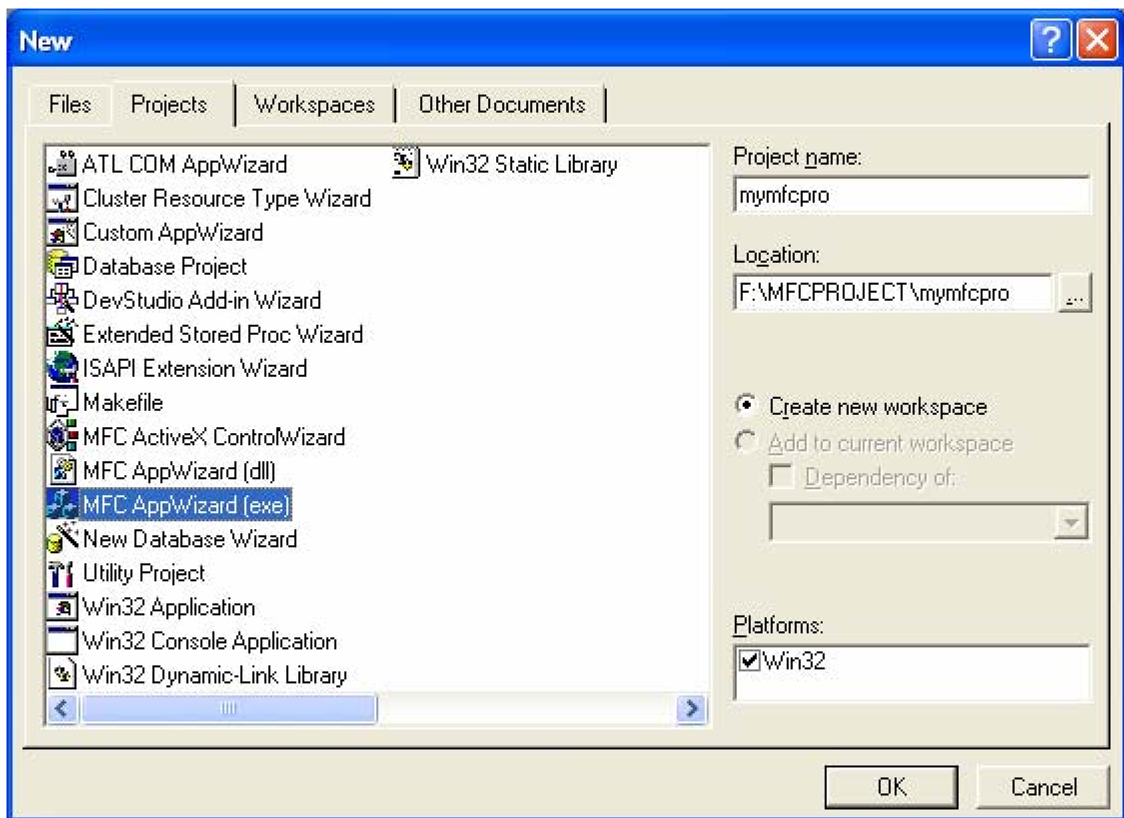

Figure 6: AppWizard new MYMFCPRO project creation dialog.

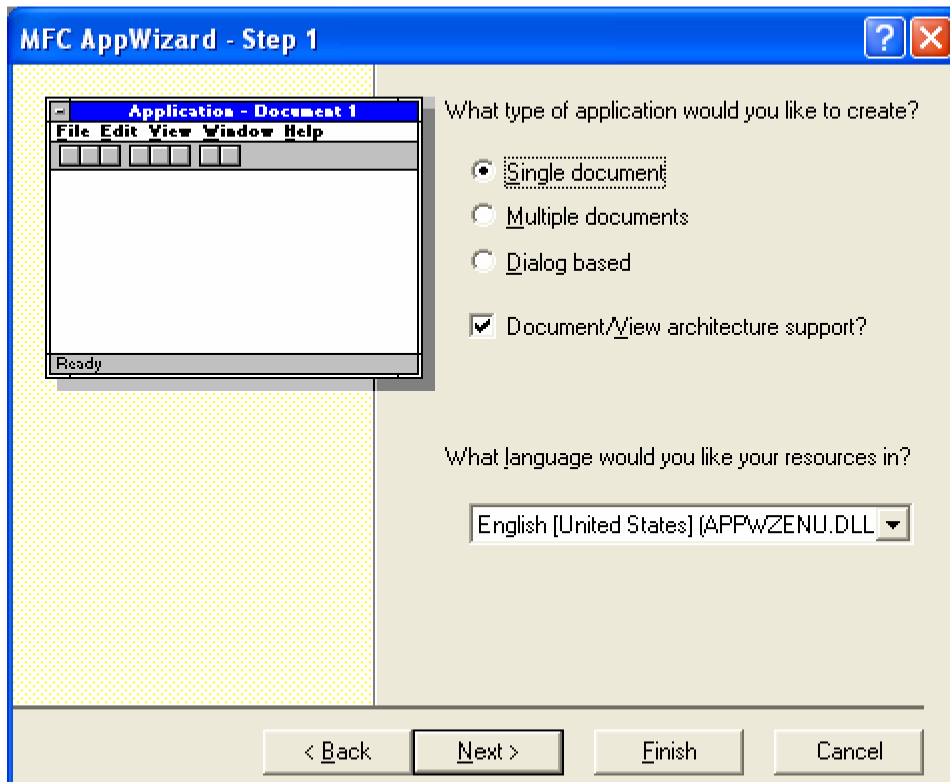

Figure 7: AppWizard step 1 of 6 dialog, selecting SDI application with Document/View architecture support.

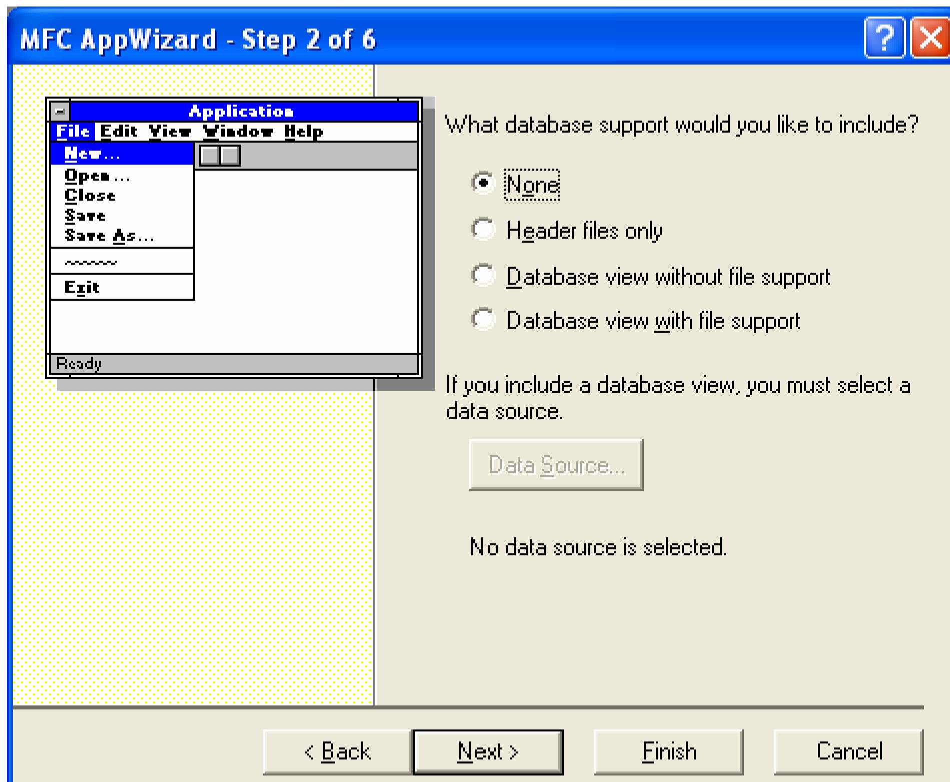

Figure 8: AppWizard step 2 of 6 dialog.

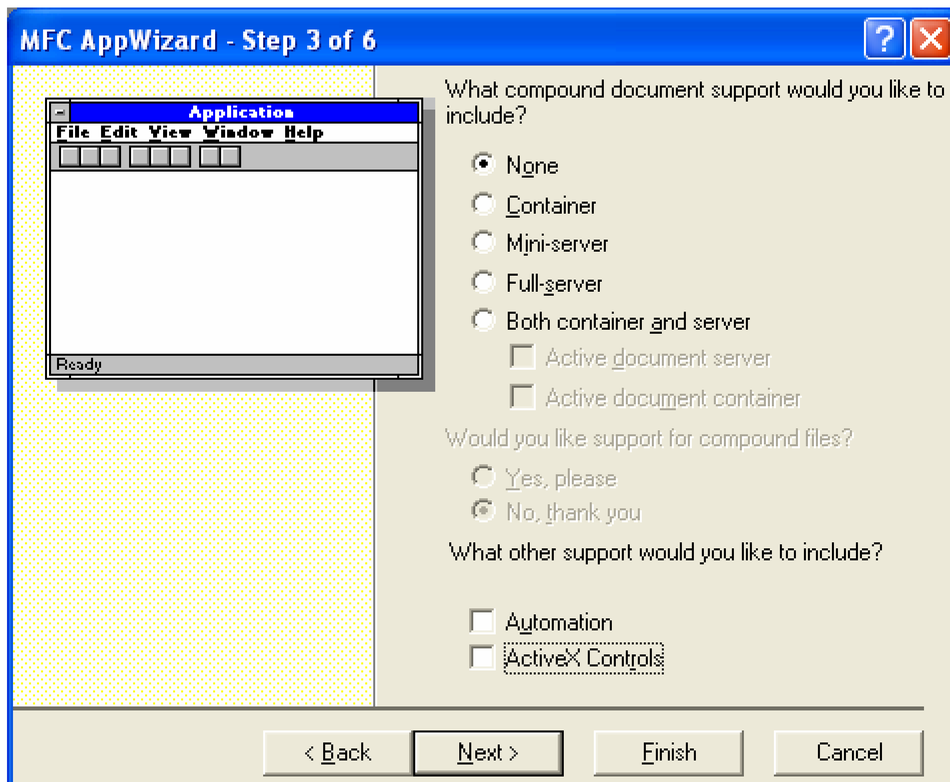

Figure 9: AppWizard step 3 of 6 dialog.

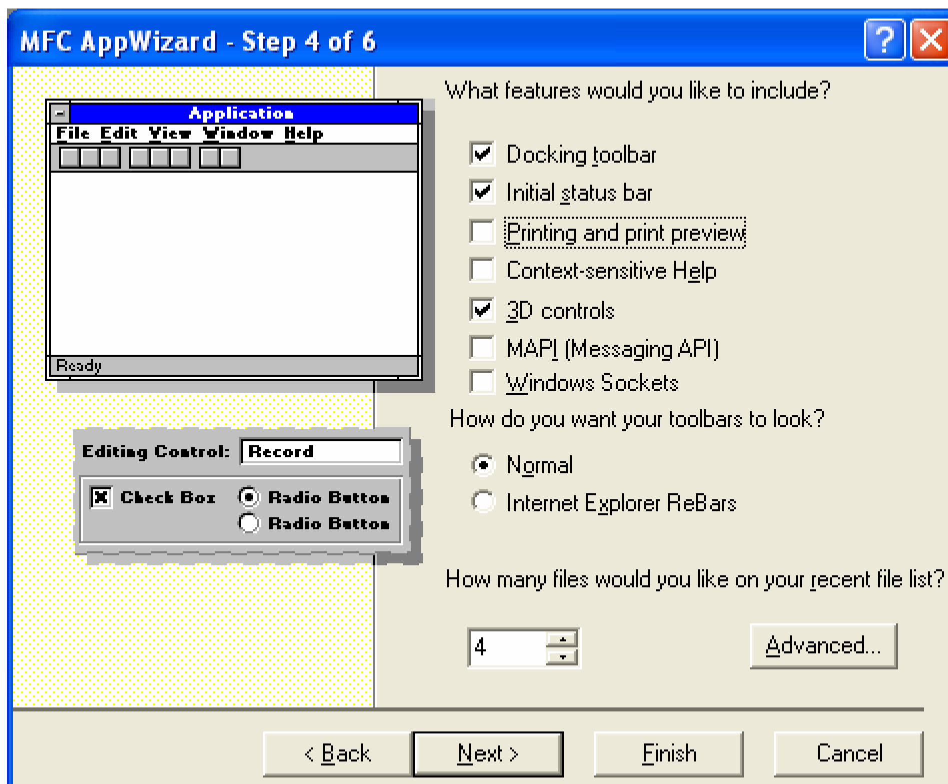

Figure 10: AppWizard step 4 of 6 dialog, deselecting **Printing and print preview** option.

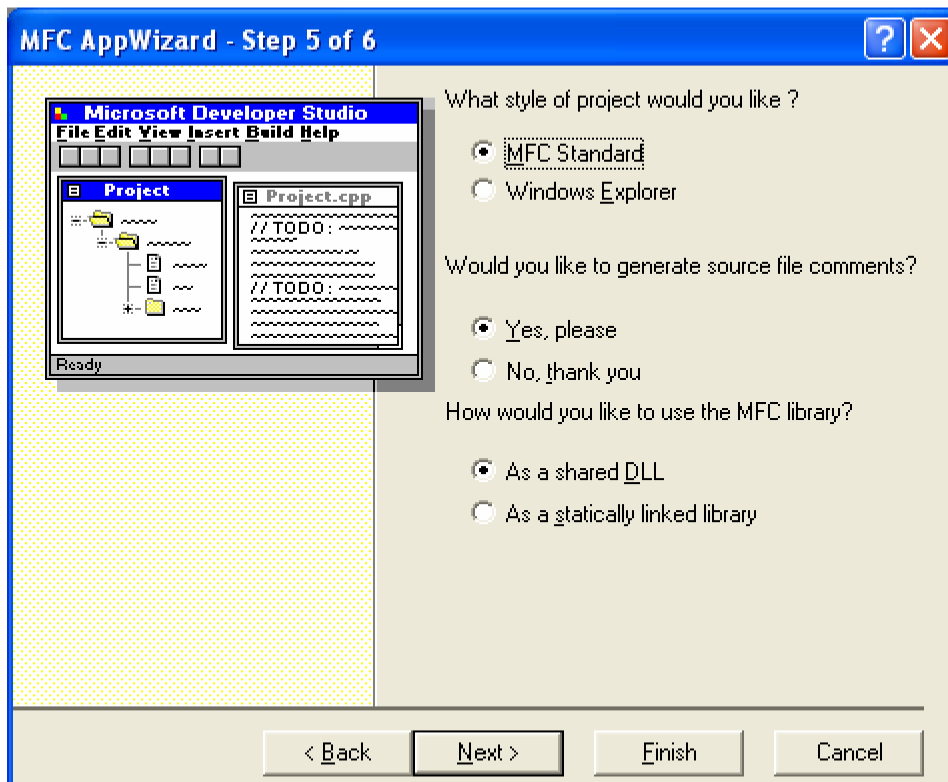

Figure 11: AppWizard step 5 of 6 dialog.

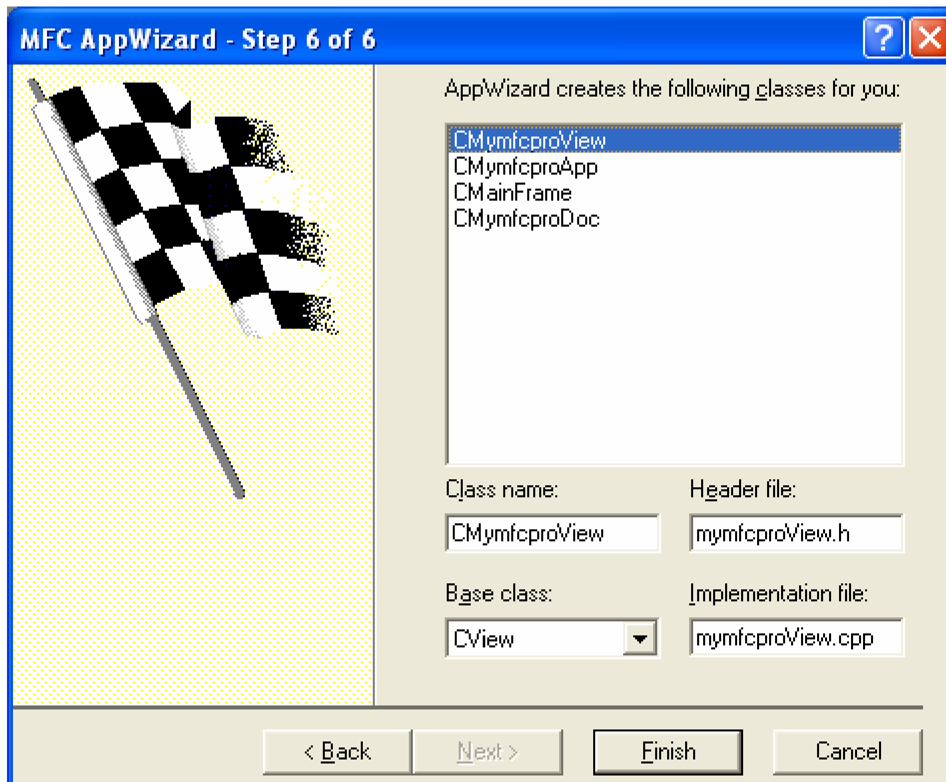

Figure 12: AppWizard step 6 of 6 dialog.

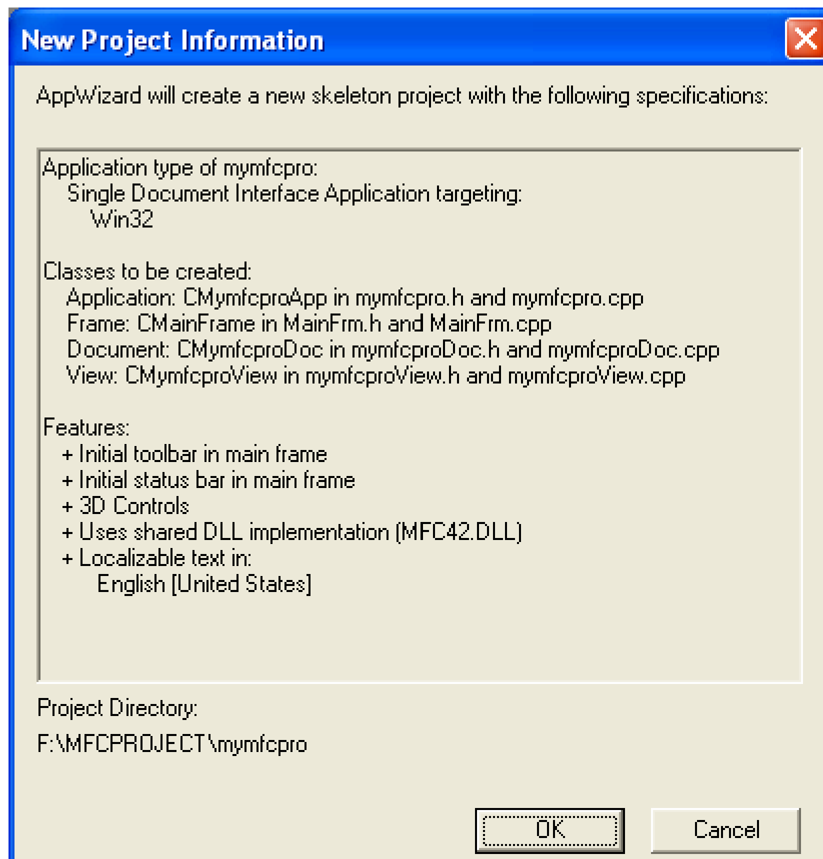

Figure 13: MYMFCPRO project summary.

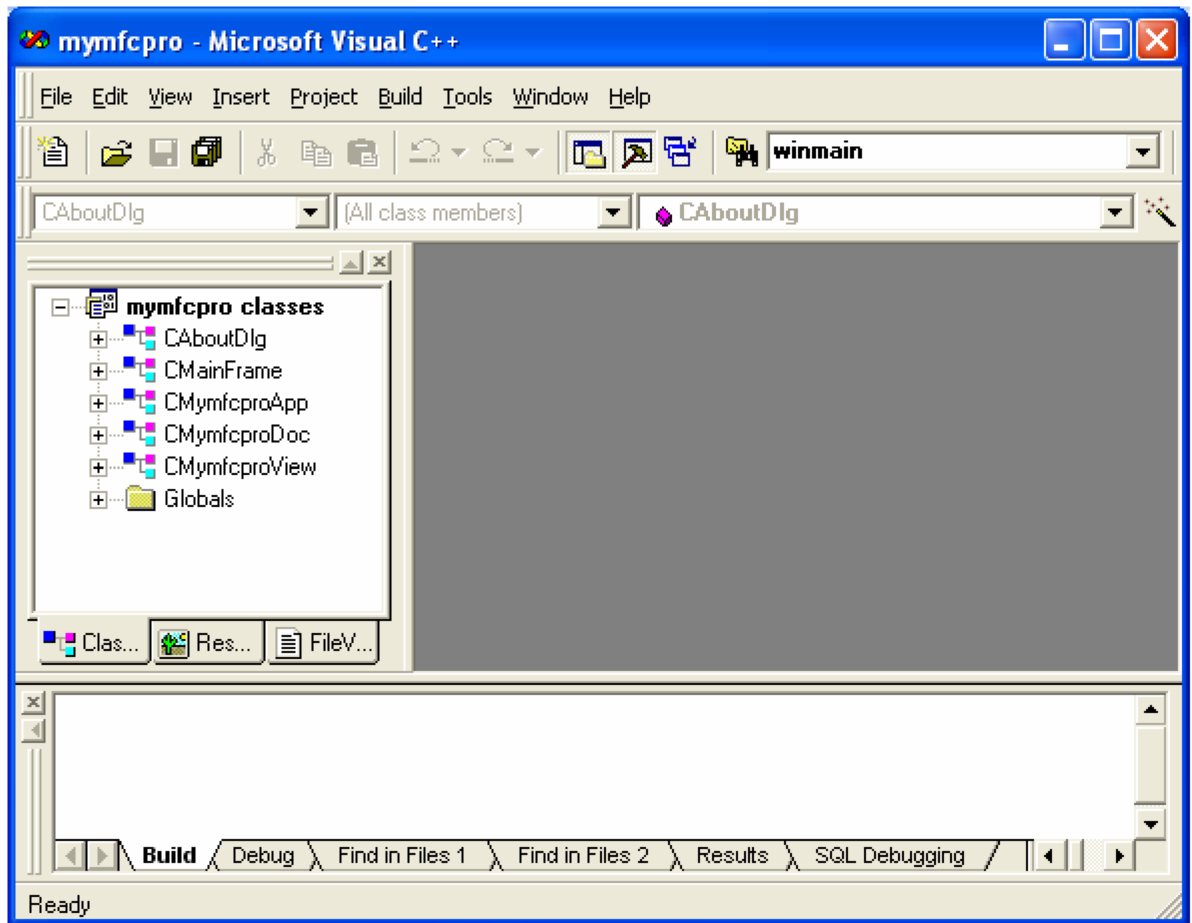

Figure 14: MYMFCPRO project IDE launched.

Use the resource editor to edit the application's main menu. Click on the **ResourceView** tab in the **Workspace** window. Edit the `IDR_MAINFRAME` menu resource to add a separator and a **Clear Document** item to the **Edit** menu, as shown here.

Menu	**Caption**	**Command ID**
Edit	Clear &Document	`ID_EDIT_CLEAR_ALL`

Table 1

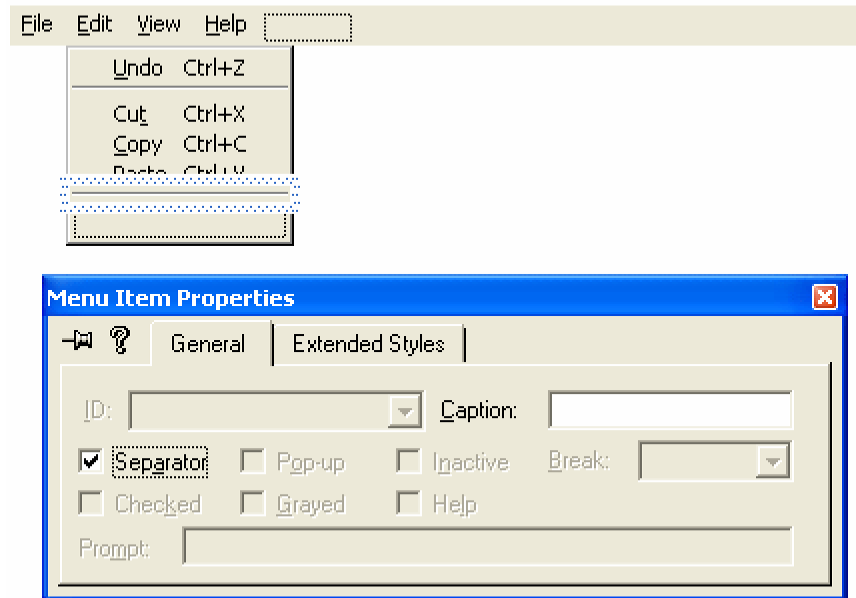

Figure 15: Using the resource editor to add a separator.

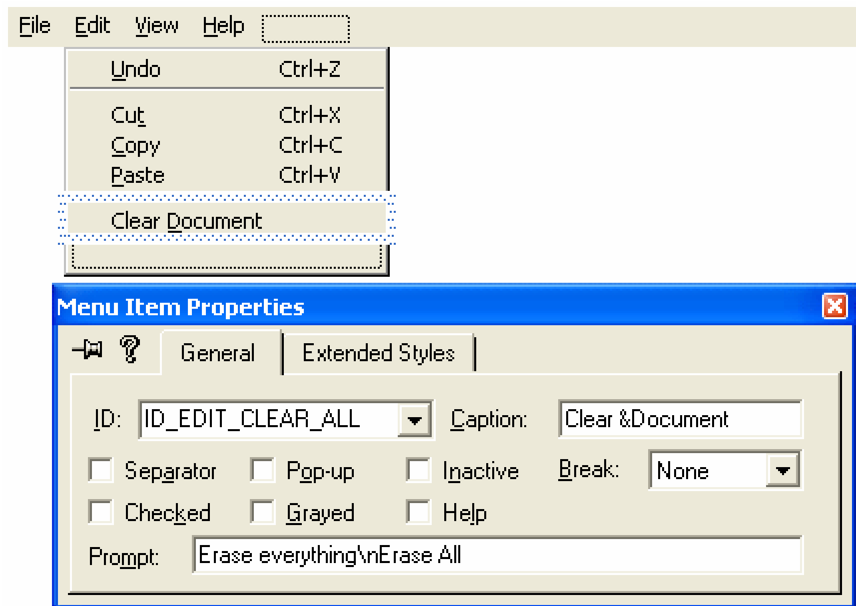

Figure 16: Using the resource editor to add **Clear Document** menu.

Now add a **Transfer** menu, and then define the underlying items. Using the mouse, drag the blank item to the insertion position to define a new item. A new blank item will appear at the bottom when you're finished.

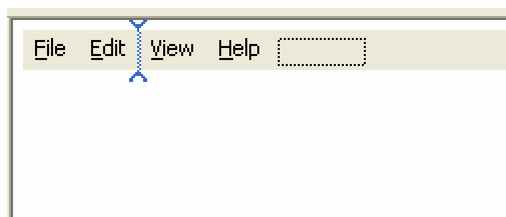

Figure 17: Using the resource editor to add a **Transfer** main menu.

Use the following command IDs for your new menu items.

Menu	Caption	Command ID
Transfer	&Get Data From Document\tF2	ID_TRANSFER_GETDATA
Transfer	&Store Data In Document\tF3	ID_TRANSFER_STOREDATA

Table 1

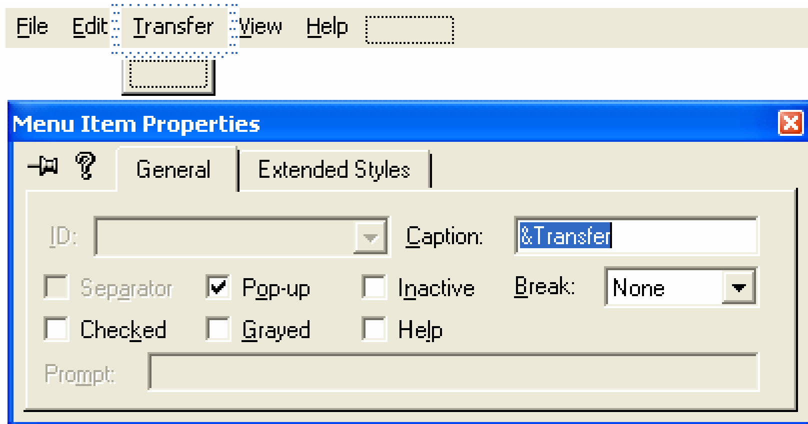

Figure 18: Adding and modifying the **Transfer** menu properties.

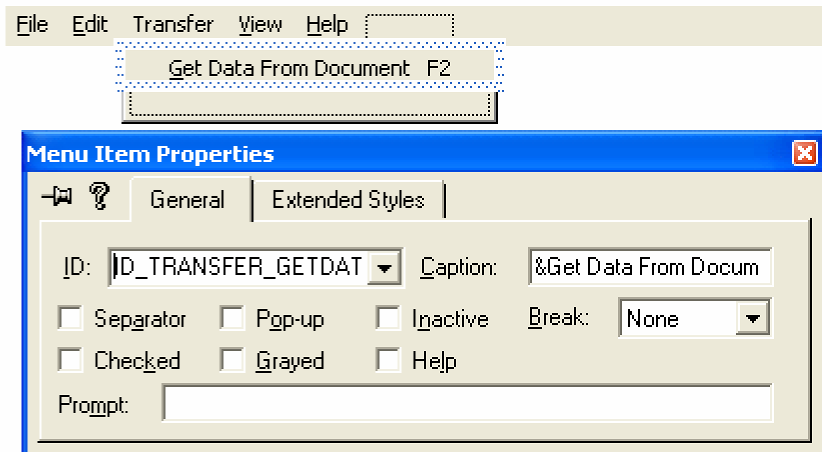

Figure 19: Adding and modifying the **Get Data From Document** menu properties.

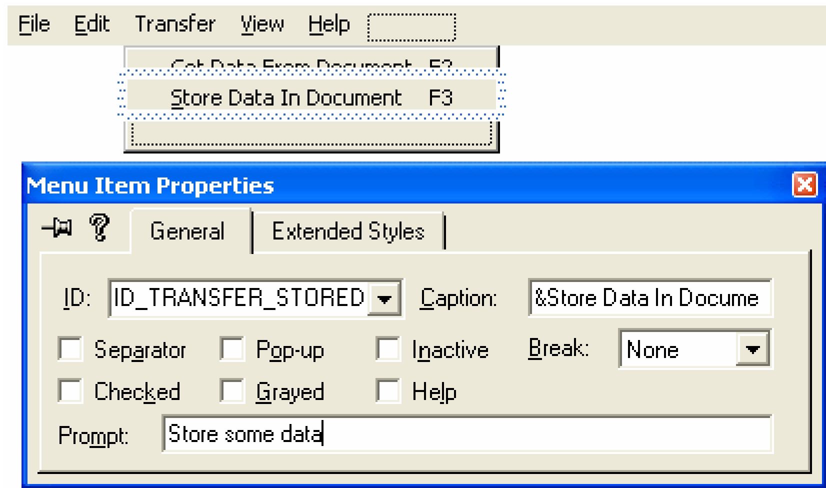

Figure 20: Adding and modifying the **Store Data In Document** menu properties.

The MFC library has defined the first item, `ID_EDIT_CLEAR_ALL`. Note that the **\t** is a tab character, but type **\t** manually; don't just press the Tab key.
When you add the menu items, type appropriate prompt strings in the **Menu Item Properties** dialog for the **Prompt**. These prompts will appear in the application's status bar window when the menu item is highlighted.

Use the resource editor to add keyboard accelerators. Open the `IDR_MAINFRAME` accelerator table, and then use the insert key to add the following items.

Accelerator ID	Key
ID_TRANSFER_GETDATA	VK_F2
ID_TRANSFER_STOREDATA	VK_F3

Table 1

Be sure to turn off the **Ctrl**, **Alt**, and **Shift** modifiers. The **Accelerator** edit screen and **Accel Properties** dialog are shown in the illustration below.

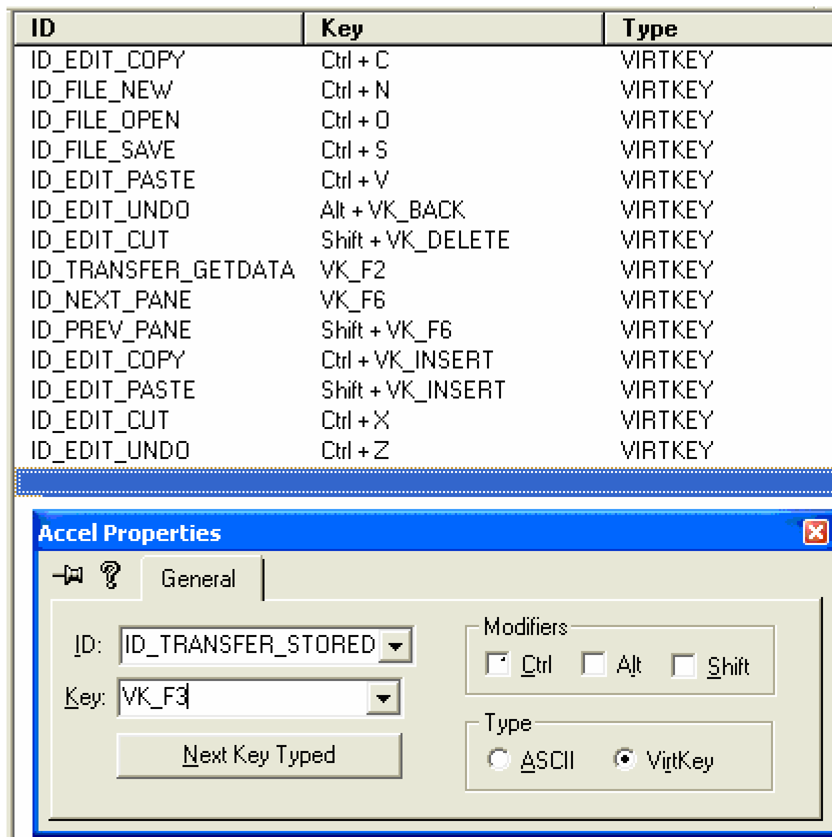

Figure 21: The Accelerator resource edit editor and properties dialog.

Use ClassWizard to add the view class command and update command UI message handlers. Select the `CMymfcproView` class, and then add the following member functions.

Object ID	Message	Member Function
ID_TRANSFER_GETDATA	COMMAND	OnTransferGetData()
ID_TRANSFER_STOREDATA	COMMAND	OnTransferStoreData()
ID_TRANSFER_STOREDATA	UPDATE_COMMAND_UI	OnUpdateTransferStoreData()

Table 1

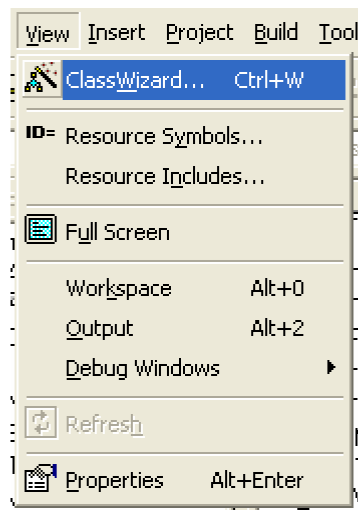

Figure 22: Invoking ClassWizard to add the view class command and update command UI message handlers.

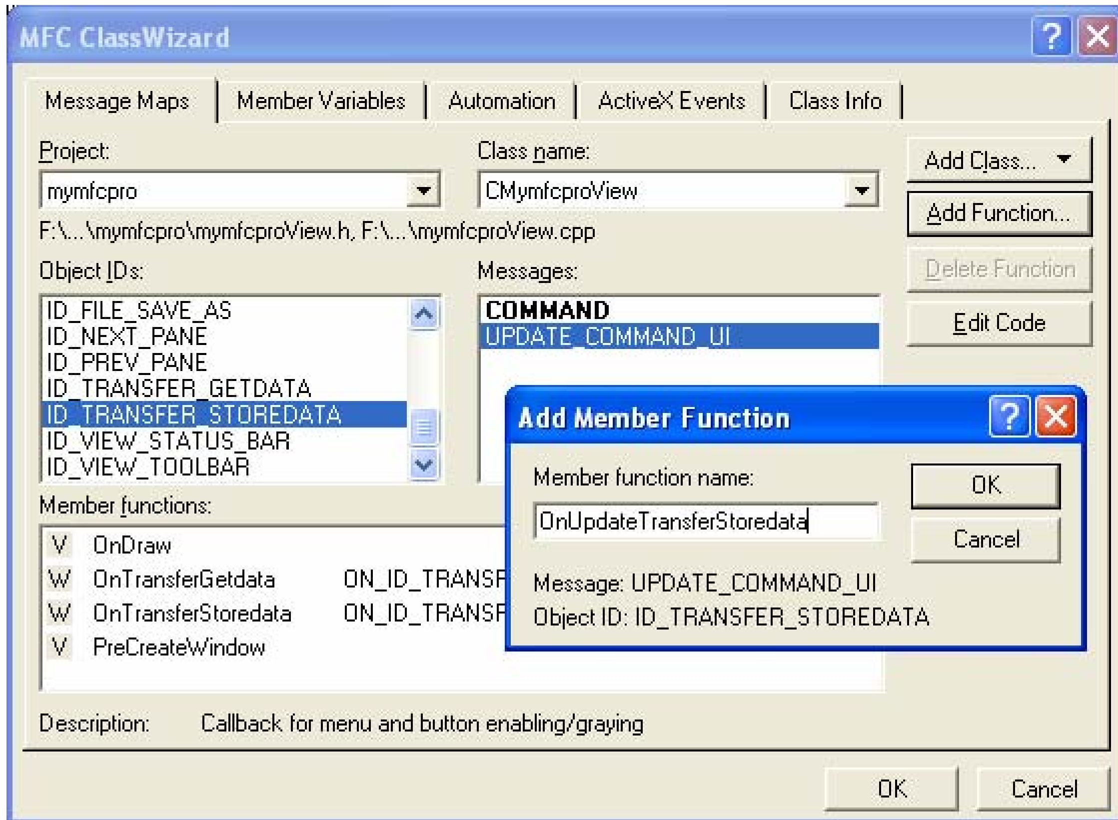

Figure 23: Adding the view class command and update command UI message handlers.

Use ClassWizard to add the document class command and update command UI message handlers. Select the `CMymfcproDoc` class, and then add the following member functions.

Object ID	Message	Member Function
ID_EDIT_CLEAR_ALL	COMMAND	OnEditClearDocument()
ID_EDIT_CLEAR_ALL	UPDATE_COMMAND_UI	OnUpdateEditClearDocument()

Table 1

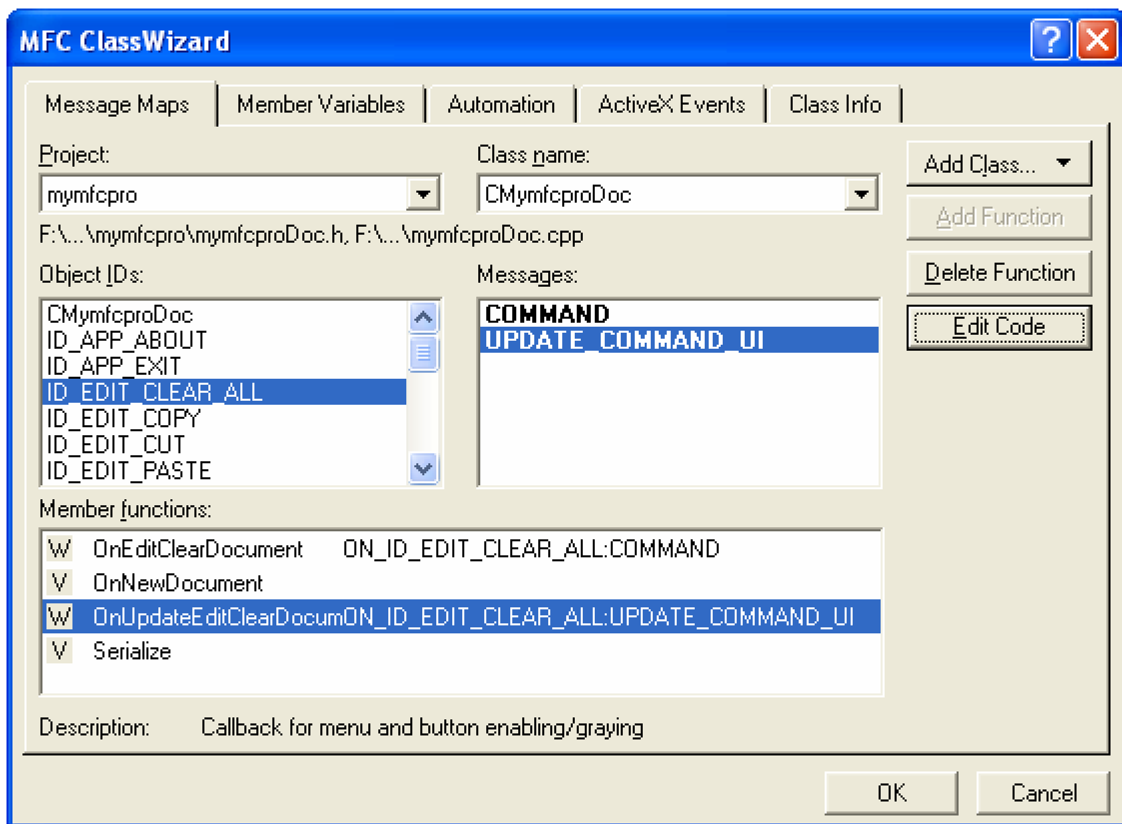

Figure 24: Using ClassWizard to add the document class command and update command UI message handlers.

Add a `CString` data member to the `CMymfcproDoc` class. Edit the file **MymfcproDoc.h** or use ClassView.

```
public:
    CString m_strText;
```

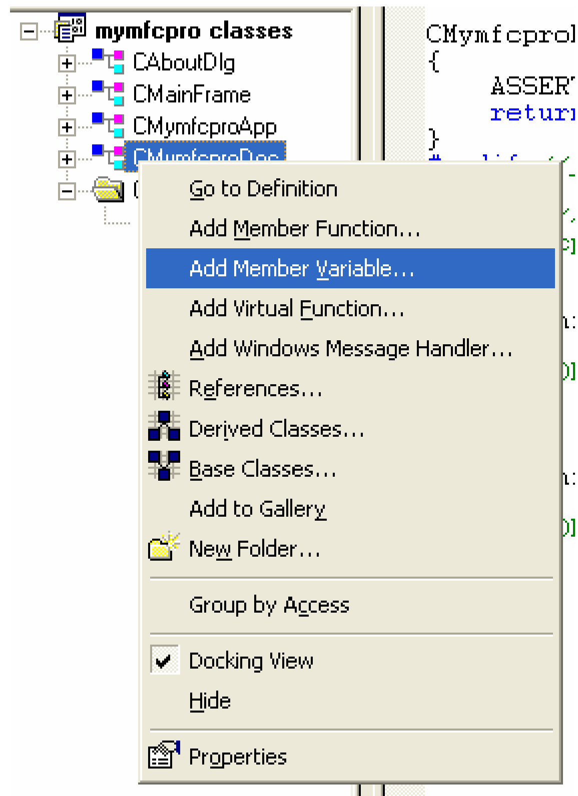

Figure 25: Using ClassView to add new member variable to `CMymfcproDoc` class.

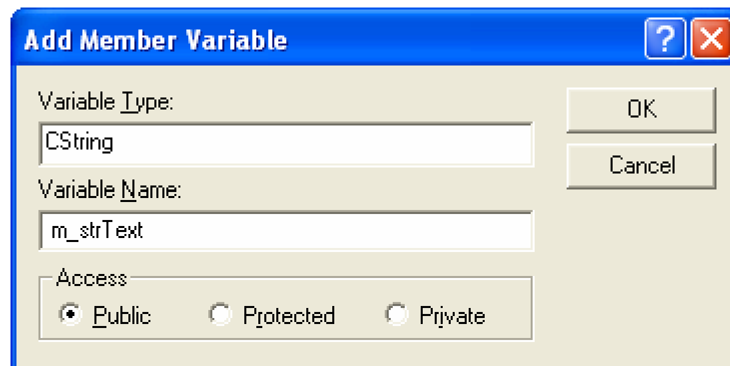

Figure 26: Entering the variable type and name.

Edit the document class member functions in **MymfcproDoc.cpp**. The `OnNewDocument()` function was generated by ClassWizard. The framework calls this function after it first constructs the document and when the user chooses New from the File menu. Your version sets some text in the string data member. Add the following code:

```
BOOL CMymfcproDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;
    m_strText = "Hello...from CMymfcproDoc::OnNewDocument";
    return TRUE;
}
```

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

    // TODO: add reinitialization code here
    // (SDI documents will reuse this document)
    m_strText = "Hello...from CMymfcproDoc::OnNewDocument";

    return TRUE;
}
```

Listing 1

The **Edit Clear Document** message handler sets `m_strText` to empty, and the update command UI handler grays the menu item if the string is already empty. Remember that the framework calls `OnUpdateEditClearDocument()` when the **Edit** menu pops up. Add the following code:

```
void CMymfcproDoc::OnEditClearDocument()
{
    m_strText.Empty();
}

void CMymfcproDoc::OnUpdateEditClearDocument(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(!m_strText.IsEmpty());
}
```

```
void CMymfcproDoc::OnEditClearDocument()
{
    // TODO: Add your command handler code here
    m_strText.Empty();

}

void CMymfcproDoc::OnUpdateEditClearDocument(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    pCmdUI->Enable(!m_strText.IsEmpty());

}
```

Listing 2.

Add a `CRichEditCtrl` data member to the **CMymfcproView** class. Edit the file **MymfcproView.h** or use ClassView.

```
public:
    CRichEditCtrl m_rich;
```

```
public:
    CRichEditCtrl m_rich;

// Attributes
```

Listing 3.

Or using ClassView

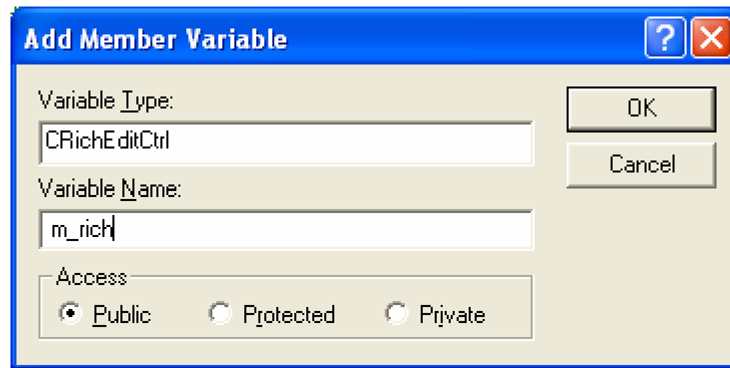

Figure 27: Adding a `CRichEditCtrl` data member to the **CMymfcproView** class.

Use ClassWizard to map the `WM_CREATE` and `WM_SIZE` messages in the **CMymfcproView** class. The `OnCreate()` function creates the rich edit control.

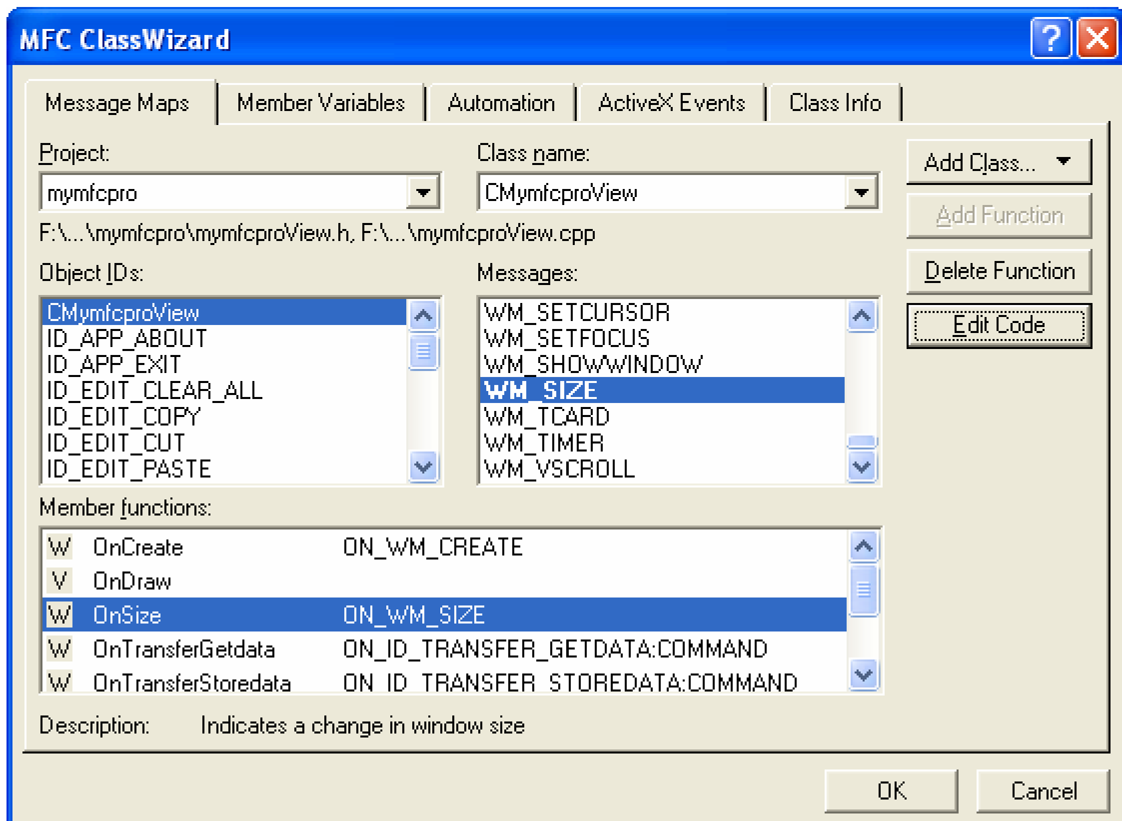

Figure 28: Using ClassWizard to map the `WM_CREATE` and `WM_SIZE` messages in the **CMymfcproView** class.

The control's size is `0` here because the view window doesn't have a size yet. The code for the two handlers is shown below. Click the **Edit Code** button and add the following codes.

```
int CMymfcproView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    CRect rect(0, 0, 0, 0);
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;
    m_rich.Create(ES_AUTOVSCROLL | ES_MULTILINE | ES_WANTRETURN |
                  WS_CHILD | WS_VISIBLE | WS_VSCROLL, rect, this, 1);
    return 0;
}
```

```
Source Files
  MainFrm.cpp
  mymfcpro.cpp
  mymfcpro.rc
  mymfcproDoc.cpp
  mymfcproView.cpp
  StdAfx.cpp
Header Files
  MainFrm.h
  mymfcpro.h
  mymfcproDoc.h
```

```
int CMymfcproView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    CRect rect(0, 0, 0, 0);
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;

    // TODO: Add your specialized creation code here
    m_rich.Create(ES_AUTOVSCROLL | ES_MULTILINE | ES_WANTRETURN |
                  WS_CHILD | WS_VISIBLE | WS_VSCROLL, rect, this, 1);

    return 0;
}
```

Listing 4.

Windows sends the `WM_SIZE` message to the view as soon as the view's initial size is determined and again each time the user changes the frame size. This handler simply adjusts the rich edit control's size to fill the view client area. Add the following code:

```
void CMymfcproView::OnSize(UINT nType, int cx, int cy)
{
    CRect rect;
    CView::OnSize(nType, cx, cy);
    GetClientRect(rect);
    m_rich.SetWindowPos(&wndTop, 0, 0, rect.right - rect.left, rect.bottom -
rect.top, SWP_SHOWWINDOW);
}
```

```
  MainFrm.cpp
  mymfcpro.cpp
  mymfcpro.rc
  mymfcproDoc.cpp
  mymfcproView.cpp
  StdAfx.cpp
Header Files
  MainFrm.h
  mymfcpro.h
  mymfcproDoc.h
```

```
void CMymfcproView::OnSize(UINT nType, int cx, int cy)
{
    CRect rect;
    CView::OnSize(nType, cx, cy);

    // TODO: Add your message handler code here
    GetClientRect(rect);
    m_rich.SetWindowPos(&wndTop, 0, 0, rect.right - rect.left,
                        rect.bottom - rect.top, SWP_SHOWWINDOW);

}
```

Listing 5.

Edit the menu command handler functions in **MymfcproView.cpp**. ClassWizard generated these skeleton functions when you mapped the menu commands in the previous step. The `OnTransferGetData()` function gets the text from the document data member and puts it in the rich edit control. The function then clears the control's modified flag. There is no update command UI handler. Add the following code:

```
void CMymfcproView::OnTransferGetData()
{
    CmymfcproDoc* pDoc = GetDocument();
    m_rich.SetWindowText(pDoc->m_strText);
    m_rich.SetModify(FALSE);
}
```

mymfcpro.rc
mymfcproDoc.cpp
mymfcproView.cpp
StdAfx.cpp
Header Files
MainFrm.h
mymfcpro.h

```
void CMymfcproView::OnTransferGetdata()
{
    // TODO: Add your command handler code here
    CMymfcproDoc* pDoc = GetDocument();
    m_rich.SetWindowText(pDoc->m_strText);
    m_rich.SetModify(FALSE);

}
```

Listing 6.

The `OnTransferStoreData()` function copies the text from the view's rich edit control to the document string and resets the control's modified flag. The corresponding update command UI handler grays the menu item if the control has not been changed since it was last copied to or from the document. Add the following code:

```
void CmymfcproView::OnTransferStoreData()
{
    CmymfcproDoc* pDoc = GetDocument();
    m_rich.GetWindowText(pDoc->m_strText);
    m_rich.SetModify(FALSE);
}

void CmymfcproView::OnUpdateTransferStoreData(CcmdUI* pCmdUI)
{
    pCmdUI->Enable(m_rich.GetModify());
}
```

ıace 'mymfcpro': 1 projeı
mfcpro files
Source Files
MainFrm.cpp
mymfcpro.cpp
mymfcpro.rc
mymfcproDoc.cpp
mymfcproView.cpp
StdAfx.cpp
Header Files
MainFrm.h
mymfcpro.h
mymfcproDoc.h

```
void CMymfcproView::OnTransferStoredata()
{
    // TODO: Add your command handler code here
    CMymfcproDoc* pDoc = GetDocument();
    m_rich.GetWindowText(pDoc->m_strText);
    m_rich.SetModify(FALSE);

}

void CMymfcproView::OnUpdateTransferStoredata(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    pCmdUI->Enable(m_rich.GetModify());

}
```

Listing 7.

Build and test the MYMFCPRO application.

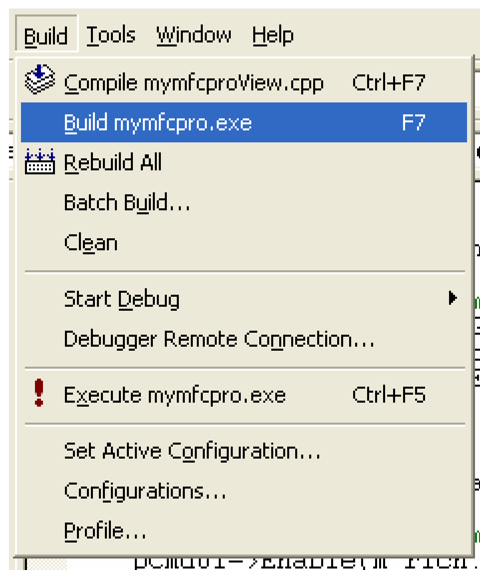

Figure 29: Building the MYMFCPRO program.

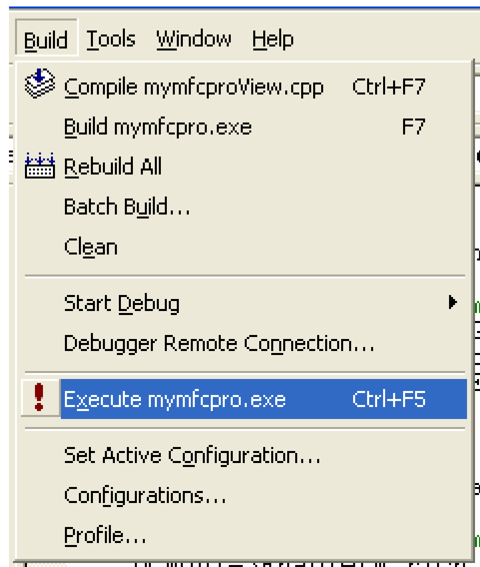

Figure 30: Executing the MYMFCPRO program.

The following is the program output. When the application starts, the **Clear Document** item on the **Edit** menu should be enabled. Choose **Get Data From Document** from the **Transfer** menu. Some text should appear. Edit the text, and then choose **Store Data In Document**. That menu item should now appear gray. Try choosing the **Clear Document** command, and then choose **Get Data From Document** again.

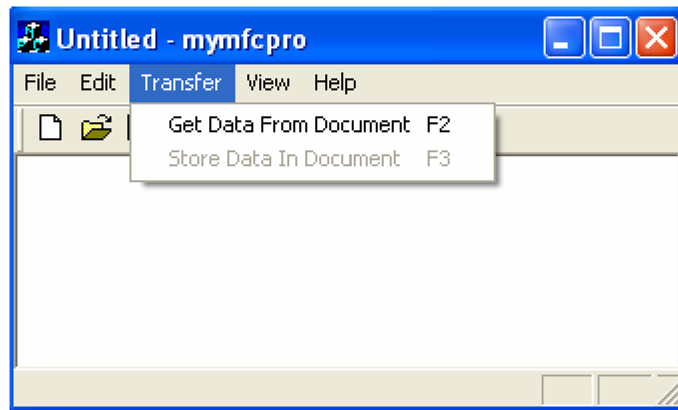

Figure 31: Testing MYMFCPRO program, getting the data from the document.

Again, let see the relationship of the Document -View. Just click the **Get Data From Document** sub menu of the **Transfer** menu.

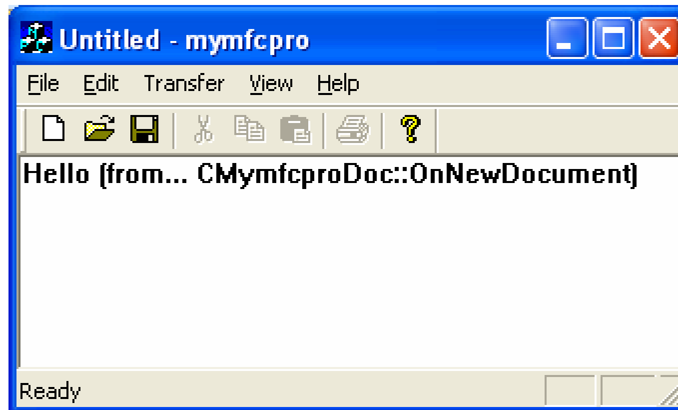

Figure32: The default data from the document.

The program display string from Document (the default, because we do not store any string/data yet). Next, type some text and click the **Store Data In Document** menu. The data sent to and store in the Document.

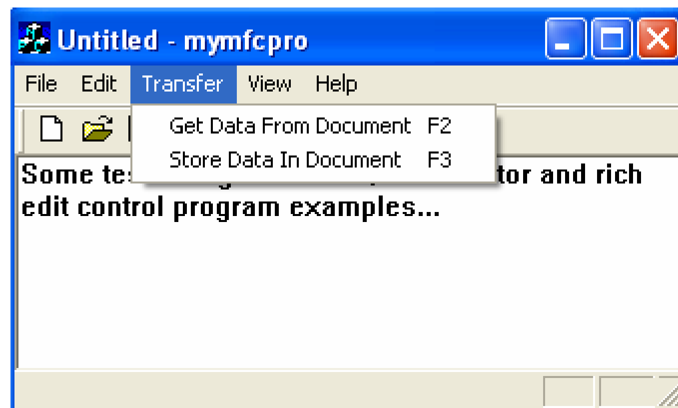

Figure 33: Storing some data (text) into the document, doing it through view.

Next, delete the string manually. So the View doesn't have any displayed data. Remember that the data has been stored in Document.

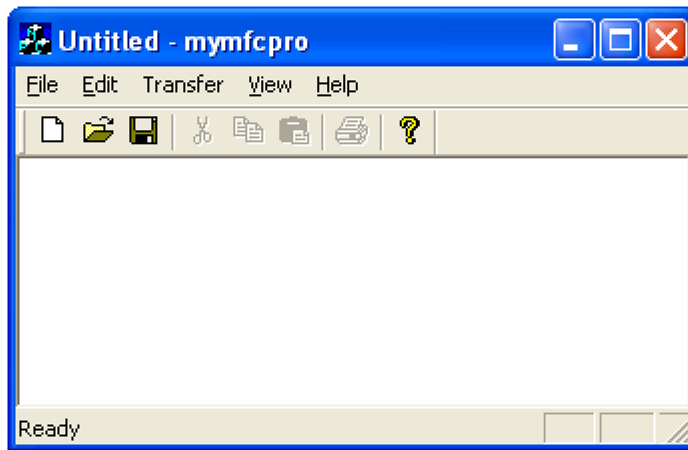

Figure 34: Deleting the data in view (a copy of the data is in document).

Then Click the **Get Data From Document**. The previous sample string will be displayed.

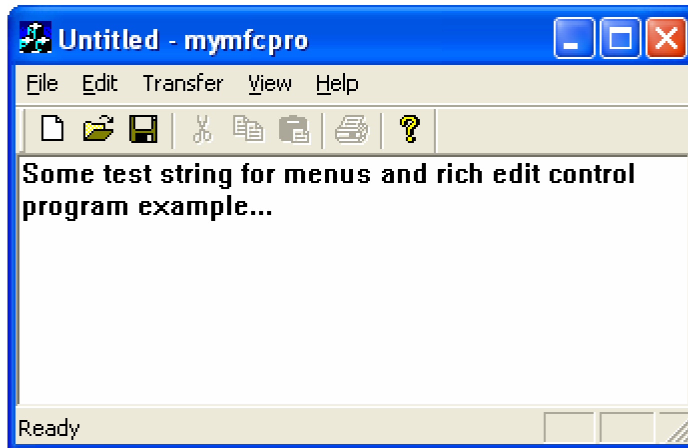

Figure 35: Getting the data from the document through view.

Through the **Clear Document** sub menu we can delete the stored data in the document. Click the **Clear Document** sub menu.

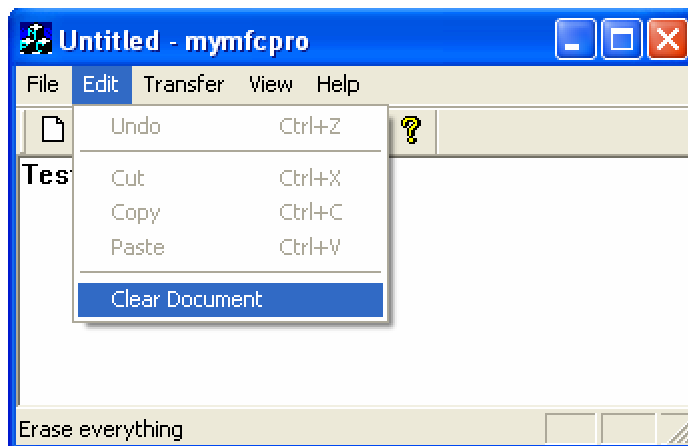

Figure 36: Deleting data stored in the document through view.

Then when we request the previous data from **Document**, through the **Get Data From Document**, of course the **View** displays nothing.

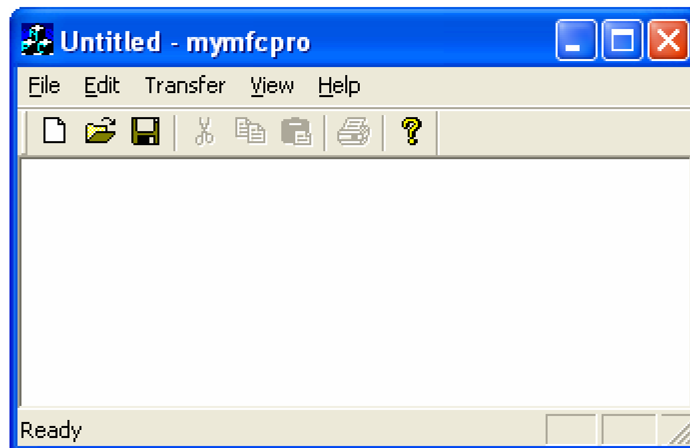

Figure 37: Viewing the data in the document after deleting it, no more data.

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.