Module 6: The Modeless Dialog and Windows Common Dialogs

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 Modeless Dialog and Windows Common Dialogs
Modeless Dialogs
Creating Modeless Dialogs
User-Defined Messages
Dialog Ownership
A Modeless Dialog Example: **MYMFC9**
The `CFormView` Class: A Modeless Dialog Alternative
The Windows Common Dialogs
Using the `CFileDialog` Class Directly
Deriving from the Common Dialog Classes
Nested Dialogs
A `CFileDialog` Example: **MYMFC10**
Other Customization for `CfileDialog`

The Modeless Dialog and Windows Common Dialogs

Now you'll move on to the modeless dialog and to the common dialogs. Modeless dialogs allow the user to work elsewhere in the application while the dialog is active. The common dialog classes are the C++ programming interface to the group of Windows utility dialogs that include **File Open**, **Page Setup**, **Color**, and so forth and that are supported by the dynamic link library **COMDLG32.DLL**. In this Module's first example, you'll build a simple modeless dialog that is controlled from a view. In the second example, you'll derive from the **COMDLG32** `CFileDialog` class a class that allows file deletion.

Modeless Dialogs

In the MFC Library version 6.0, modal and modeless dialogs share the same base class, `CDialog`, and they both use a dialog resource that you can build with the dialog editor. If you're using a modeless dialog with a view, you'll need to know some specialized programming techniques.

Creating Modeless Dialogs

For modal dialogs, you've already learned that you construct a dialog object using a `CDialog` constructor that takes a resource template ID as a parameter, and then you display the modal dialog window by calling the `DoModal()` member function. The window ceases to exist as soon as `DoModal()` returns. Thus, you can construct a modal dialog object on the stack, knowing that the dialog window has been destroyed by the time the C++ dialog object goes out of scope.
Modeless dialogs are more complicated. You start by invoking the `CDialog` default constructor to construct the dialog object, but then to create the dialog window you need to call the `CDialog::Create` member function instead of `DoModal()`. Create takes the resource ID as a parameter and returns immediately with the dialog window still on the screen. You must worry about exactly when to construct the dialog object, when to create the dialog window, when to destroy the dialog, and when to process user-entered data. Here's a summary of the differences between creating a modal dialog and a modeless dialog.

	Modal Dialog	Modeless Dialog
Constructor used	Constructor with resource ID param	Default constructor (no params)
Function used to create window	`DoModal()`	`Create()` with resource ID param

Table 1

User-Defined Messages

Suppose you want the modeless dialog window to be destroyed when the user clicks the dialog's OK button. This presents a problem. How does the view know that the user has clicked the OK button? The dialog could call a view class member function directly, but that would "marry" the dialog to a particular view class. A better solution is for the dialog to send the view a user-defined message as the result of a call to the OK button message-handling function. When the view gets the message, it can destroy the dialog window (but not the object). This sets the stage for the creation of a new dialog. You have two options for sending Windows messages: the `CWnd::SendMessage` function or the `PostMessage()` function. The former causes an immediate call to the message-handling function, and the latter posts a message in the Windows message queue. Because there's a slight delay with the `PostMessage()` option, it's reasonable to expect that the handler function has returned by the time the view gets the message.

Dialog Ownership

Now suppose you've accepted the dialog default pop-up style, which means that the dialog isn't confined to the view's client area. As far as Windows is concerned, the dialog's "owner" is the application's main frame window, not the view. You need to know the dialog's view to send the view a message. Therefore, your dialog class must track its own view through a data member that the constructor sets. The `CDialog` constructor's `pParent` parameter doesn't have any effect here, so don't bother using it.

A Modeless Dialog Example: MYMFC9

We could convert the previous Module monster dialog to a modeless dialog, but starting from scratch with a simpler dialog is easier. Example MYMFC9 uses a dialog with one edit control, an **OK** button, and a **Cancel** button. As in the previous Module example, pressing the left mouse button while the mouse cursor is inside the view window brings up the dialog, but now we have the option of destroying it in response to another event, pressing the right mouse button when the mouse cursor is inside the view window. We'll allow only one open dialog at a time, so we must be sure that a second left button press doesn't bring up a duplicate dialog.
To summarize the upcoming steps, the MYMFC9 view class has a single associated dialog object that is constructed on the heap when the view is constructed. The dialog window is created and destroyed in response to user actions, but the dialog object is not destroyed until the application terminates. Here are the steps to create the MYMFC9 example:

Run AppWizard to produce \mfcproject\mymfc9 (or whatever directory you have designated for the project). Accept all the defaults but two: select **Single Document** and deselect **Printing And Print Preview** and **ActiveX Controls**. The options and the default class names are shown here.

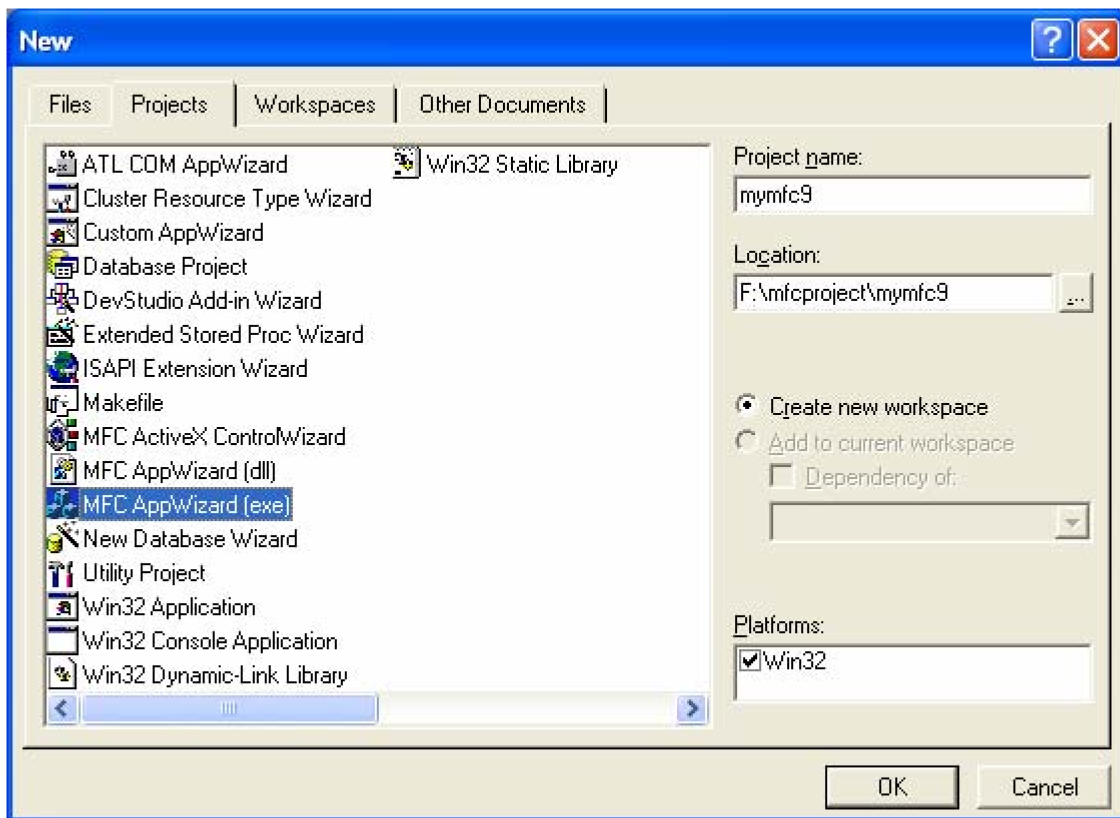

Figure 1: MFC AppWizard new project creation dialog.

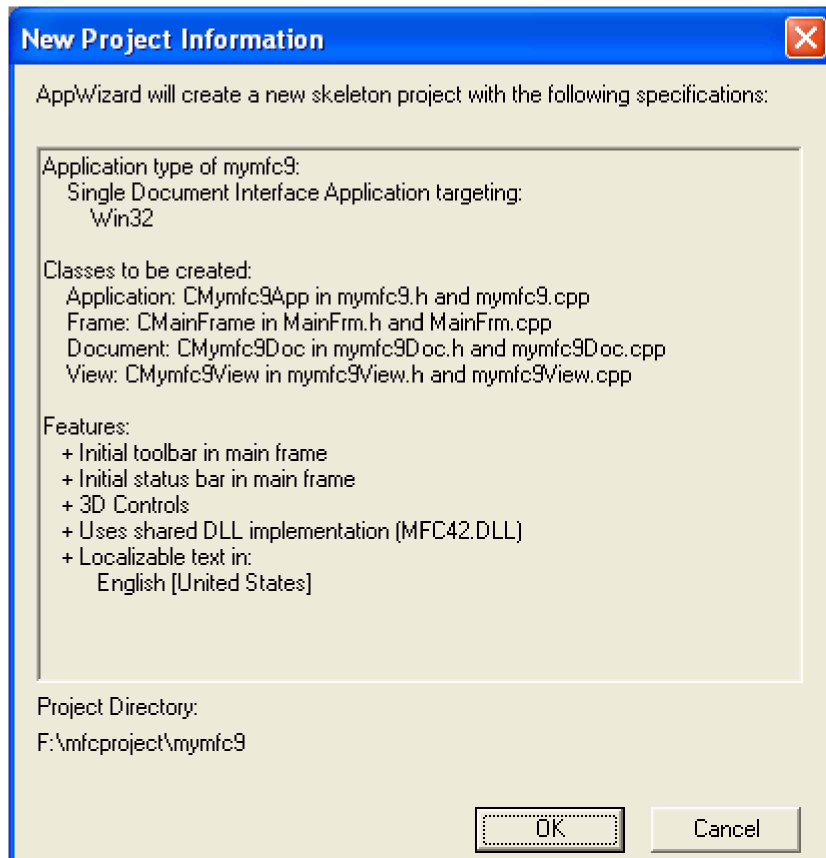

Figure 2: MYMFC9 project summary.

Use the dialog editor to create a dialog resource. Choose **Resource** from Visual C++'s **Insert** menu, and then select **Dialog**. The dialog editor assigns the ID `IDD_DIALOG1` (the default ID) to the new dialog. Change the dialog caption to **Modeless Dialog**. Accept the default **OK** and **Cancel** buttons with IDs `IDOK` and `IDCANCEL`, and then add a static text control and an edit control with the default ID `IDC_EDIT1`. Change the static text control's caption to **Edit 1**. Here is the completed dialog. Be sure to select the dialog's **Visible** property.

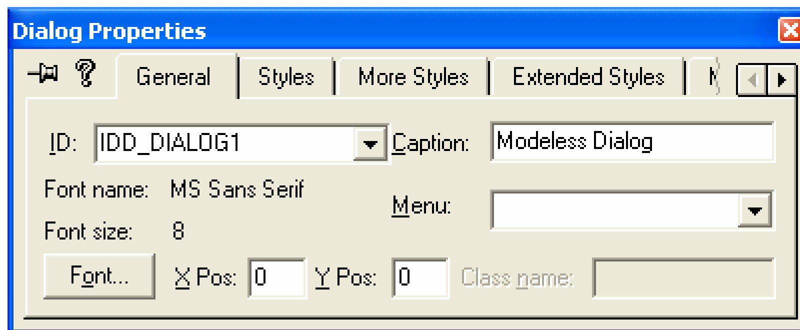

Figure 3: Modifying the dialog properties.

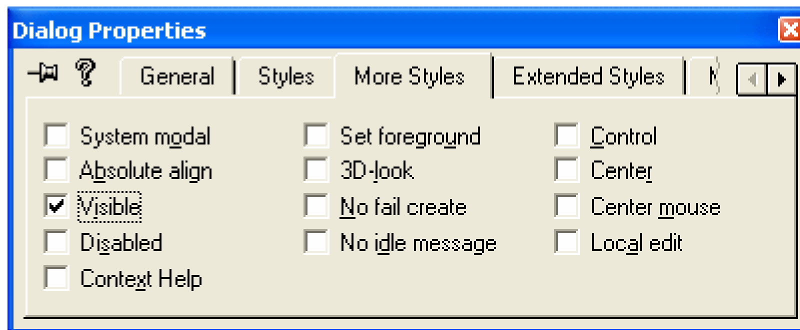

Figure 4: More dialog properties modification.

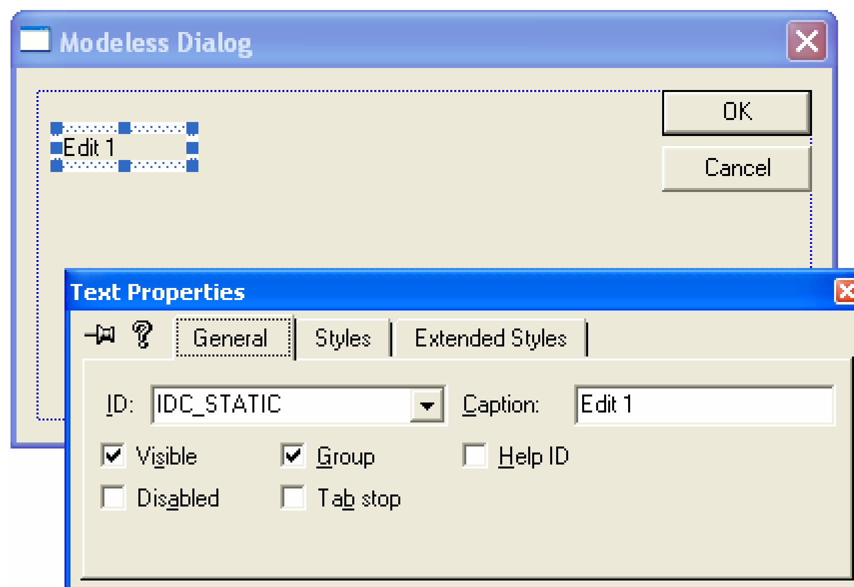

Figure 5: Modifying the static text control properties.

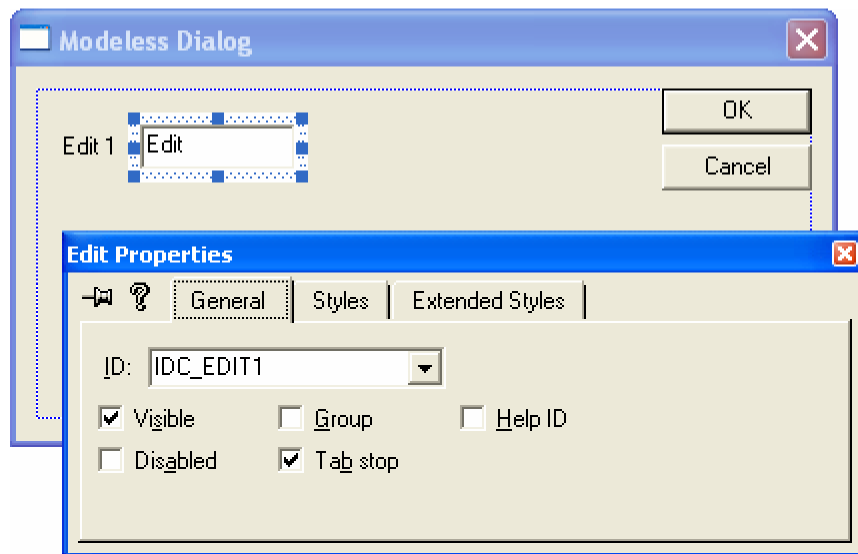

Figure 6: Modifying the Edit control properties.

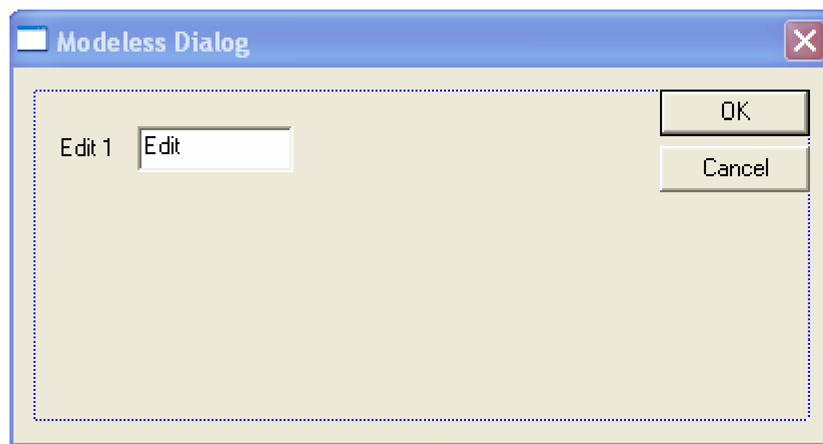

Figure 7: Modeless dialog with its controls.

Use ClassWizard to create the `CMymfc9Dialog` class. Choose ClassWizard from Microsoft Visual C++'s **View** menu. Fill in the **New Class** dialog as shown here, and then click the **OK** button.

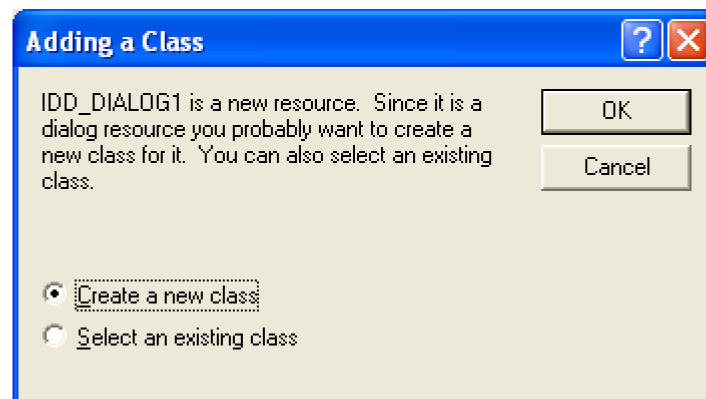

Figure 8: Creating a new `CMymfc9Dialog` class dialog prompt.

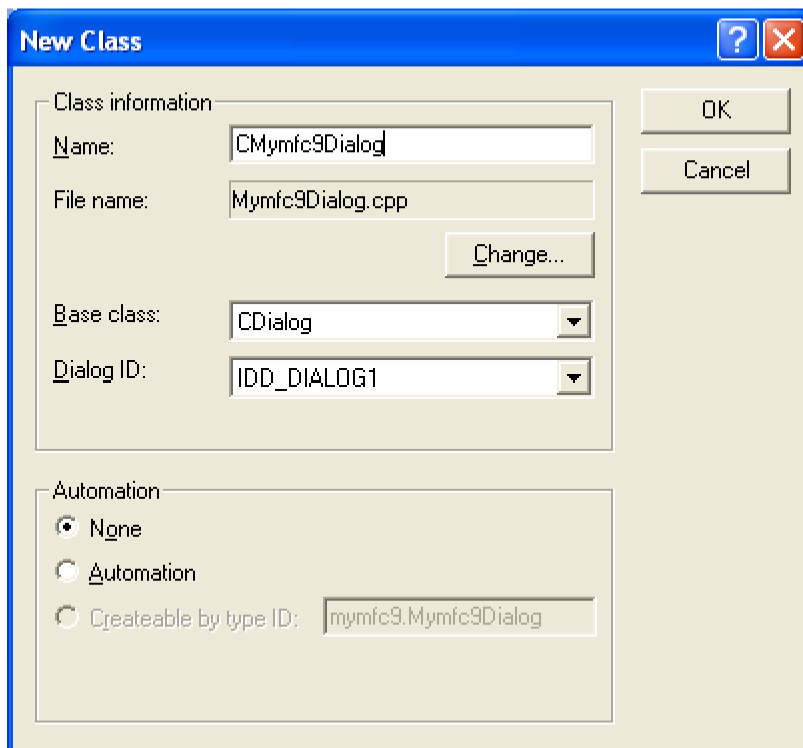

Figure 9: The `CMymfc9Dialog` class information.

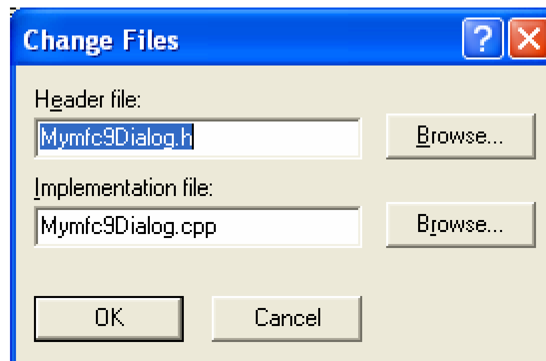

Figure 10: Changing the default class header and implementation file names if required, not for this example.

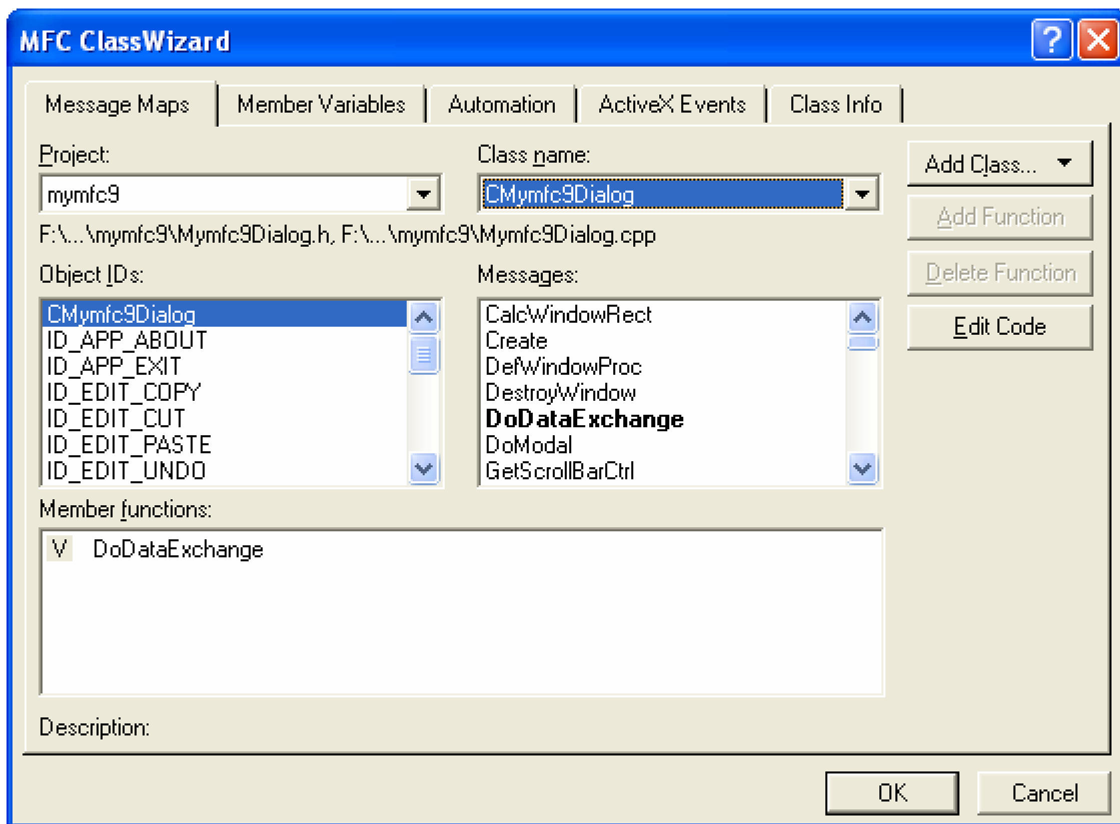

Figure 11: New class included in MYMFC9 project, ready to be used.

Add the message-handling functions shown below.

Object ID	Message	Member Function
IDCANCEL	BN_CLICKED	OnCancel
IDOK	BN_CLICKED	OnOK

Table 2

To add a message-handling function, click on an object ID, click on a message, and then click the **Add Function** button. The **Add Member Function** dialog box appears. Edit the function name if necessary, and click the **OK** button.

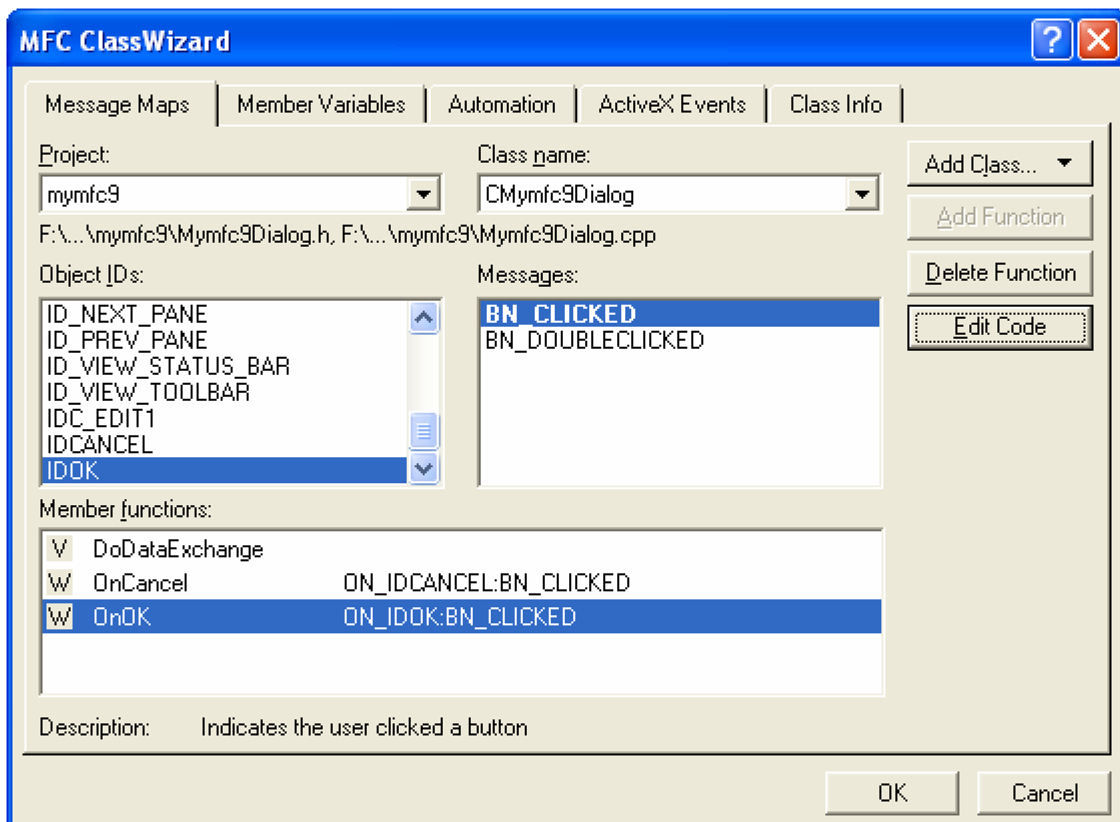

Figure 12: Add a message-handling function for `IDOK` object.

Add a variable to the `CMymfc9Dialog` class. While in ClassWizard, click on the **Member Variables** tab, choose the `IDC_EDIT1` control, and then click the **Add Variable** button to add the `CString` variable `m_strEdit1`.

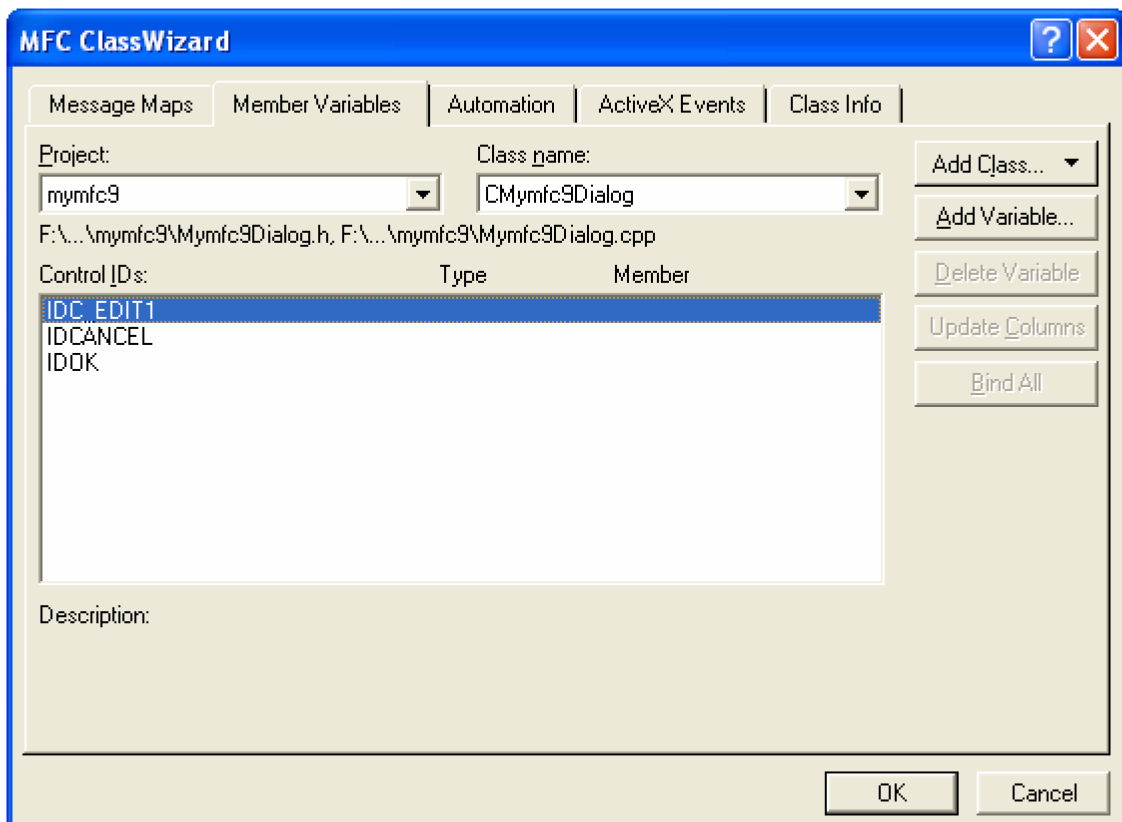

Figure 13: Adding a member variable to the `CMymfc9Dialog` class.

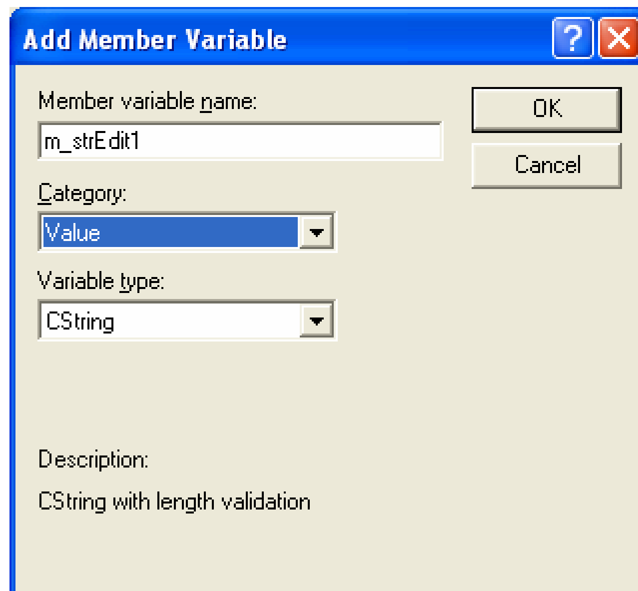

Figure 14: Entering the member variable name and type.

Edit **mymfc9Dialog.h** to add a view pointer and function prototypes. Type in the following code in the `CMymfc9Dialog` class declaration:

```
private:
    CView* m_pView;
```

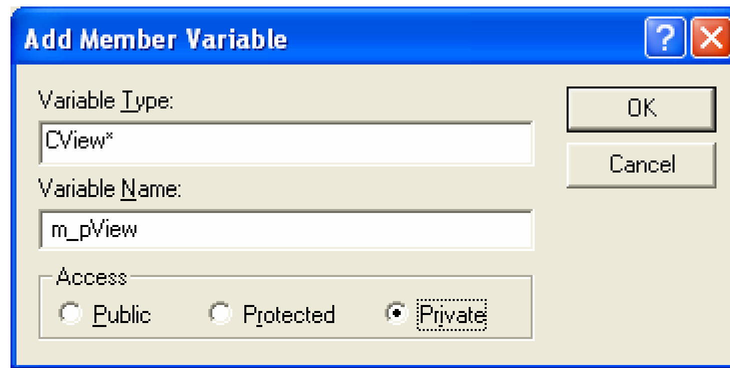

Figure 15: Adding a view pointer's type and name.

Also, add the function prototypes as follows:

```
public:
    CMymfc9Dialog(CView* pView);
    BOOL Create();
```

```
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
private:
    CView* m_pView;

public:
    CMymfc9Dialog(CView* pView);
    BOOL Create();
};
```

Listing 1.

Using the `CView` class rather than the `CMymfc9View` class allows the dialog class to be used with any view class. Edit **mymfc9Dialog.h** to define the `WM_GOODBYE` message ID. Add the following line of code:

```
#define WM_GOODBYE  WM_USER + 5
```

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

#define WM_GOODBYE  WM_USER + 5

// Mymfc9Dialog.h : header file
```

Listing 2.

The Windows constant `WM_USER` is the first message ID available for user-defined messages. The application framework uses a few of these messages, so we'll skip over the first five messages.

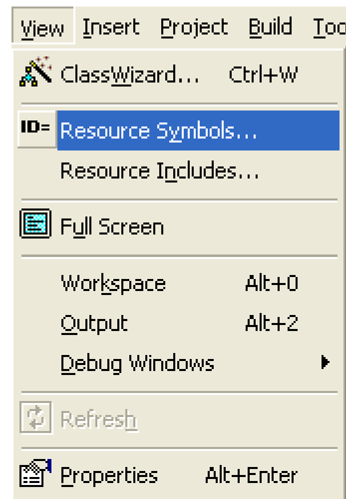

Figure 16: Viewing the resource symbols in the project.

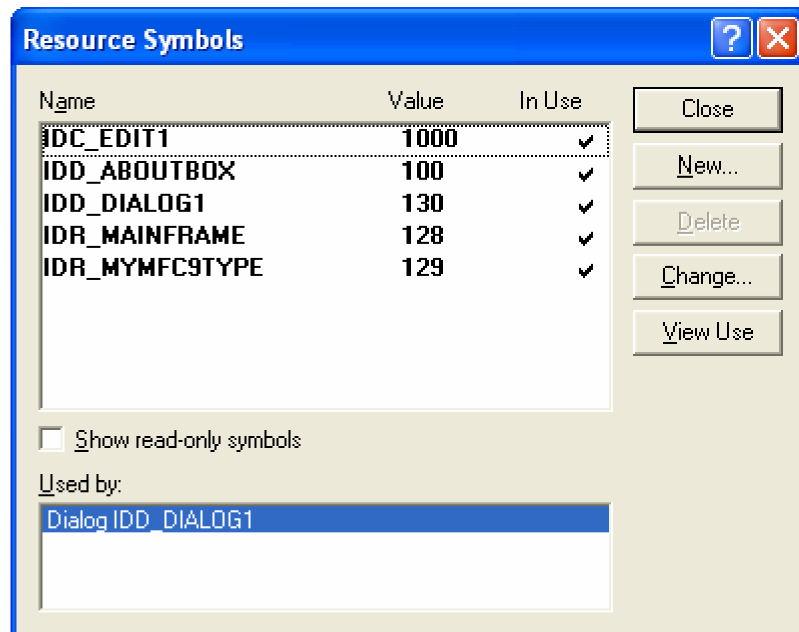

Figure 17: Inserting and deleting resource symbol through the Resource Symbol dialog.

Visual C++ maintains a list of symbol definitions in your project's resource.h file, but the resource editor does not understand constants based on other constants. Don't manually add `WM_GOODBYE` to resource.h because Visual C++ might delete it.

Add the modeless constructor in the file **mymfc9Dialog.cpp**. You could modify the existing `CMymfc9Dialog` constructor, but if you add a separate one, the dialog class can serve for both modal and modeless dialogs. Add the lines shown below.

```
// modeless constructor
CMymfc9Dialog::CMymfc9Dialog(CView* pView)
{
    m_pView = pView;
}
```

```
CMymfc9Dialog::CMymfc9Dialog(CWnd* pParent /*=NULL*/)
    : CDialog(CMymfc9Dialog::IDD, pParent)
{
    //{{AFX_DATA_INIT(CMymfc9Dialog)
    m_strEdit1 = _T("");
    //}}AFX_DATA_INIT
}

// modeless constructor
CMymfc9Dialog::CMymfc9Dialog(CView* pView)
{
    m_pView = pView;
}
```

Listing 3.

You should also add the following line to the AppWizard-generated modal constructor:

```
m_pView = NULL;
```

```
CMymfc9Dialog::CMymfc9Dialog(CWnd* pParent /*=NULL*/)
    : CDialog(CMymfc9Dialog::IDD, pParent)
{
    //{{AFX_DATA_INIT(CMymfc9Dialog)
    m_strEdit1 = _T("");
    //}}AFX_DATA_INIT
    m_pView = NULL;
}
```

Listing 4.

The C++ compiler is clever enough to distinguish between the modeless constructor `CMymfc9Dialog(CView*)` and the modal constructor `CMymfc9Dialog(CWnd*)`. If the compiler sees an argument of class `CView` or a derived `CView` class, it generates a call to the modeless constructor. If it sees an argument of class `CWnd` or another derived `CWnd` class, it generates a call to the modal constructor.
Add the `Create()` function in **mymfc9Dialog.cpp**. This derived dialog class `Create()` function calls the base class function with the dialog resource ID as a parameter. Add the following lines:

```
BOOL CMymfc9Dialog::Create()
{
    return CDialog::Create(CMymfc9Dialog::IDD);
}
```

```
// CMymfc9Dialog message handlers

BOOL CMymfc9Dialog::Create()
{
    return CDialog::Create(CMymfc9Dialog::IDD);
}

void CMymfc9Dialog::OnCancel()
{
    // TODO: Add extra cleanup here
```

Listing 5.

`Create()` is not a virtual function. You could have chosen a different name if you had wanted to. Edit the `OnOK()` and `OnCancel()` functions in **mymfc9Dialog.cpp**. These virtual functions generated by ClassWizard are called in response to dialog button clicks. Add the following code:

```
// not really a message handler
```

```
void CMymfc9Dialog::OnCancel()
{
    if (m_pView != NULL)
    {
        // modeless case - do not call base class OnCancel
        m_pView->PostMessage(WM_GOODBYE, IDCANCEL);
    }
    else
    {
        CDialog::OnCancel(); // modal case
    }
}
```


```
void CMymfc9Dialog::OnCancel()
{
    // TODO: Add extra cleanup here
    if (m_pView != NULL)
    {
        // modeless case - do not call base class OnCancel
        m_pView->PostMessage(WM_GOODBYE, IDCANCEL);
    }
    else
    {
        CDialog::OnCancel(); // modal case
    }
}
```


Listing 6.

```
// not really a message handler
void CMymfc9Dialog::OnOK()
{
    if (m_pView != NULL)
    {
        // modeless case -- do not call base class OnOK
        UpdateData(TRUE);
        m_pView->PostMessage(WM_GOODBYE, IDOK);
    }
    else
    {
        CDialog::OnOK(); // modal case
    }
}
```


```
void CMymfc9Dialog::OnOK()
{
    // TODO: Add extra validation here
    if (m_pView != NULL)
    {
        // modeless case -- do not call base class OnOK
        UpdateData(TRUE);
        m_pView->PostMessage(WM_GOODBYE, IDOK);
    }
    else
    {
        CDialog::OnOK(); // modal case
    }
}
```


Listing 7.

If the dialog is being used as a modeless dialog, it sends the user-defined message `WM_GOODBYE` to the view. We'll worry about handling the message later.

For a modeless dialog, be sure you do not call the `CDialog::OnOK` or `CDialog::OnCancel` function. This means you must override these virtual functions in your derived class; otherwise, using the **Esc** key, the **Enter** key, or a button click would result in a call to the base class functions, which call the Windows `EndDialog()` function. `EndDialog()` is appropriate only for modal dialogs. In a modeless dialog, you must call `DestroyWindow()` instead, and if necessary, you must call `UpdateData()` to transfer data from the dialog controls to the class data members. Edit the **mymfc9View.h** header file. You need a data member to hold the dialog pointer:

```
private:
    CMymfc9Dialog* m_pDlg;
```

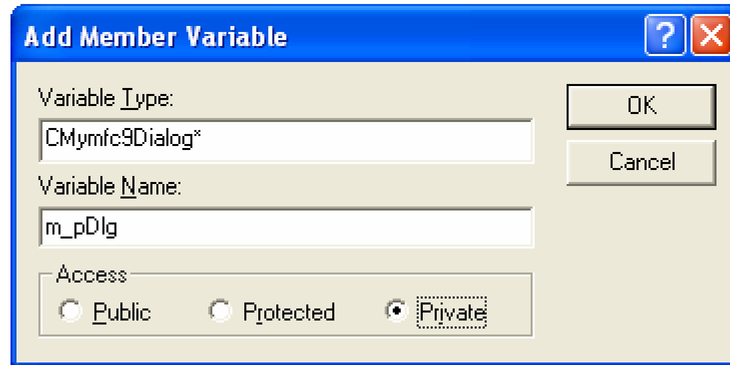

Figure 18: Adding a data member/member variable to hold the dialog pointer.

```
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
private:
    CMymfc9Dialog* m_pDlg;
};
```

Listing 8.

If you add the forward declaration:

```
class CMymfc9Dialog;
```

```
class CMymfc9Dialog;
// mymfc9View.h : interface
//
//////////////////////////////
```

Listing 9.

At the beginning of **mymfc9View.h**, you won't have to include **mymfc9Dialog.h** in every module that includes **mymfc9View.h**.

Modify the `CMymfc9View` constructor and destructor in the file **mymfc9View.cpp**. The `CMymfc9View` class has a data member `m_pDlg` that points to the view's `CMymfc9Dialog` object. The view constructor constructs the dialog object on the heap, and the view destructor deletes it. Add the following code:

```
CMymfc9View::CMymfc9View()
{
    m_pDlg = new CMymfc9Dialog(this);
}

CMymfc9View::~CMymfc9View()
{
  // destroys window if not already destroyed
  delete m_pDlg;
```

```
}
```

```
// CMymfc9View construction/destruction

CMymfc9View::CMymfc9View()
{
    // TODO: add construction code here
    m_pDlg = new CMymfc9Dialog(this);
}

CMymfc9View::~CMymfc9View()
{
    // destroys window if not already destroyed
    delete m_pDlg;
}
```

Listing 10.

Add code to the virtual `OnDraw()` function in the **mymfc9View.cpp** file. The `CMymfc9View` `OnDraw()` function which skeleton was generated by AppWizard should be coded as follows in order to prompt the user to press the mouse button:

```
void CMymfc9View::OnDraw(CDC* pDC)
{
    pDC->TextOut(30, 30, "Press the left mouse button here.");
}
```

```
// CMymfc9View drawing

void CMymfc9View::OnDraw(CDC* pDC)
{
    // TODO: add draw code for native data here
    pDC->TextOut(30, 30, "Press the left mouse button here.");
}
```

Listing 11.

Use ClassWizard to add `CMymfc9View` mouse message handlers. Add handlers for the `WM_LBUTTONDOWN` and `WM_RBUTTONDOWN` messages.

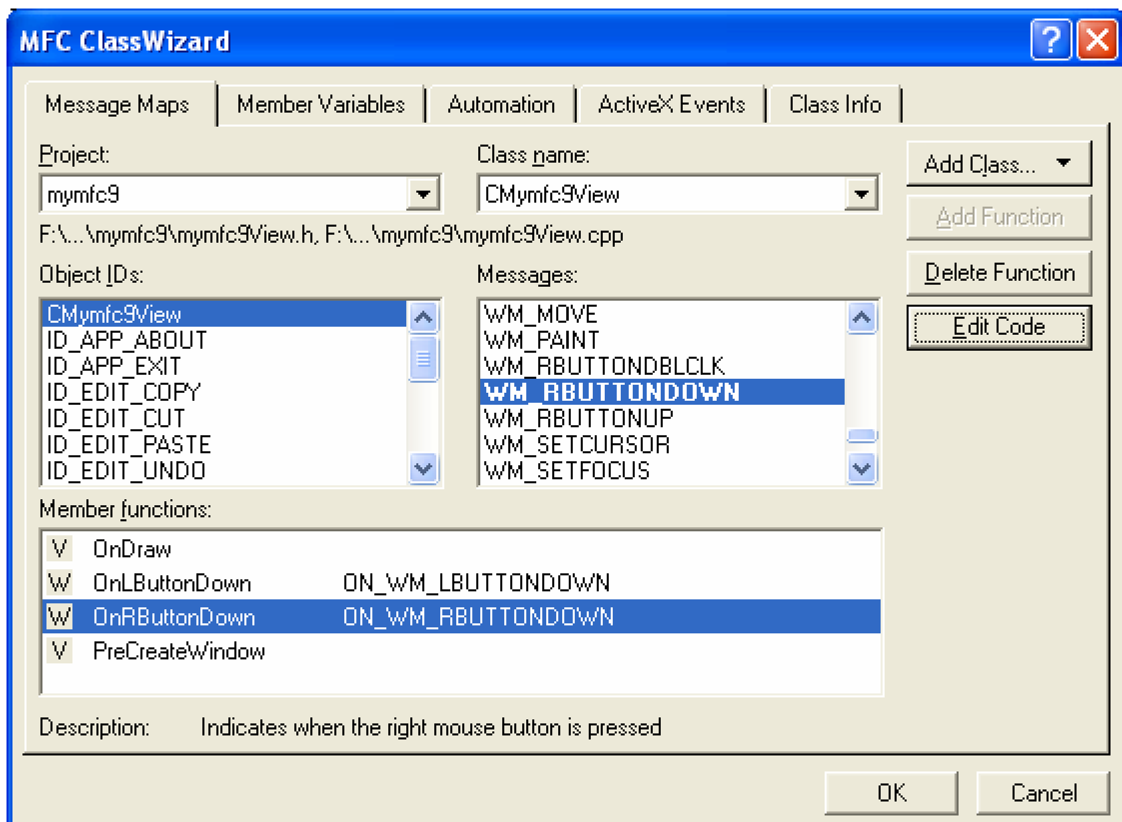

Figure 19: Adding handlers for the `WM_LBUTTONDOWN` and `WM_RBUTTONDOWN` messages.

Now edit the code in file **mymfc9View.cpp** as follows:

```
void CMymfc9View::OnLButtonDown(UINT nFlags, CPoint point)
{
    // creates the dialog if not created already
    if (m_pDlg->GetSafeHwnd() == 0)
    {
        m_pDlg->Create(); // displays the dialog window
    }
}

void CMymfc9View::OnRButtonDown(UINT nFlags, CPoint point)
{
    m_pDlg->DestroyWindow();
    // no problem if window was already destroyed
}
```

```
// CMymfc9View message handlers

void CMymfc9View::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    // creates the dialog if not created already
    if (m_pDlg->GetSafeHwnd() == 0)
    {
        m_pDlg->Create(); // displays the dialog window
    }
}

void CMymfc9View::OnRButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    m_pDlg->DestroyWindow();
    // no problem if window was already destroyed
}
```

Listing 12.

For most window types except main frame windows, the `DestroyWindow()` function does not destroy the C++ object. We want this behavior because we'll take care of the dialog object's destruction in the view destructor.
Add the dialog header include statement to file **mymfc9View.cpp**. While you're in **mymfc9View.cpp**, add the following dialog header include statement after the view header include statement:

```
#include "mymfc9Dialog.h"
```

```
#include "stdafx.h"
#include "mymfc9.h"

#include "mymfc9Doc.h"
#include "mymfc9View.h"
#include "mymfc9Dialog.h"

#ifdef _DEBUG
```

Listing 13.

Add your own message code for the `WM_GOODBYE` message. Because ClassWizard does not support user-defined messages, you must write the code yourself. This task makes you appreciate the work ClassWizard does for the other messages.

In **mymfc9View.cpp**, add the following line after the `BEGIN_MESSAGE_MAP` statement but outside the `AFX_MSG_MAP` brackets:

```
ON_MESSAGE(WM_GOODBYE, OnGoodbye)
```

```
// CMymfc9View

IMPLEMENT_DYNCREATE(CMymfc9View, CView)

BEGIN_MESSAGE_MAP(CMymfc9View, CView)
    //{{AFX_MSG_MAP(CMymfc9View)
    ON_WM_LBUTTONDOWN()
    ON_WM_RBUTTONDOWN()
    //}}AFX_MSG_MAP
    ON_MESSAGE(WM_GOODBYE, OnGoodbye)
END_MESSAGE_MAP()
```

Listing 14.

Also in **mymfc9View.cpp**, add the message handler function itself:

```
LRESULT CMymfc9View::OnGoodbye(WPARAM wParam, LPARAM lParam)
{
    // message received in response to modeless dialog OK
    // and Cancel buttons
    TRACE("CMymfc9View::OnGoodbye %x, %lx\n", wParam, lParam);
    TRACE("Dialog edit1 contents = %s\n", (const char*) m_pDlg->m_strEdit1);
    m_pDlg->DestroyWindow();
    return 0L;
}
```

```
LRESULT CMymfc9View::OnGoodbye(WPARAM wParam, LPARAM lParam)
{
    // message received in response to modeless dialog OK
    //  and Cancel buttons
    TRACE("CMymfc9View::OnGoodbye %x, %lx\n", wParam, lParam);
    TRACE("Dialog edit1 contents = %s\n", (const char*) m_pDlg->m_strEdit1);
    m_pDlg->DestroyWindow();
    return 0L;
}
```

Listing 15.

In **mymfc9View.h**, add the following function prototype before the `DECLARE_MESSAGE_MAP()` statement but outside the `AFX_ MSG` brackets:

```
afx_msg LRESULT OnGoodbye(WPARAM wParam, LPARAM lParam);
```

```
// Generated message map functions
protected:
    //{{AFX_MSG(CMymfc9View)
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
    afx_msg void OnRButtonDown(UINT nFlags, CPoint point);
    //}}AFX_MSG
    afx_msg LRESULT OnGoodbye(WPARAM wParam, LPARAM lParam);
    DECLARE_MESSAGE_MAP()
private:
    CMymfc9Dialog* m_pDlg;
};
```

Listing 16.

With Win32, the `wParam` and `lParam` parameters are the usual means of passing message data. In a mouse button down message, for example, the mouse x and y coordinates are packed into the `lParam` value. With the MFC library, message data is passed in more meaningful parameters. The mouse position is passed as a `CPoint` object. User-defined messages must use `wParam` and `lParam`, so you can use these two variables however you want. In this example, we've put the button ID in `wParam`.
Build and test the application. Build and run MYMFC9. **Press the left mouse button** and then **press the right button**. Be sure the mouse cursor is outside the dialog window when you press the right mouse button. Press the left mouse button again and enter some data, and then click the dialog's OK button. Does the view's `TRACE` statement correctly list the edit control's contents?

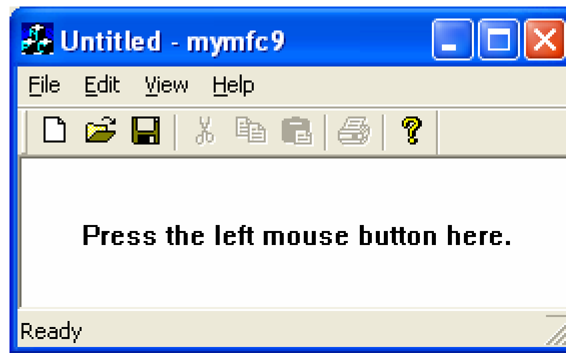

Figure 20: MYMFC9 program output.

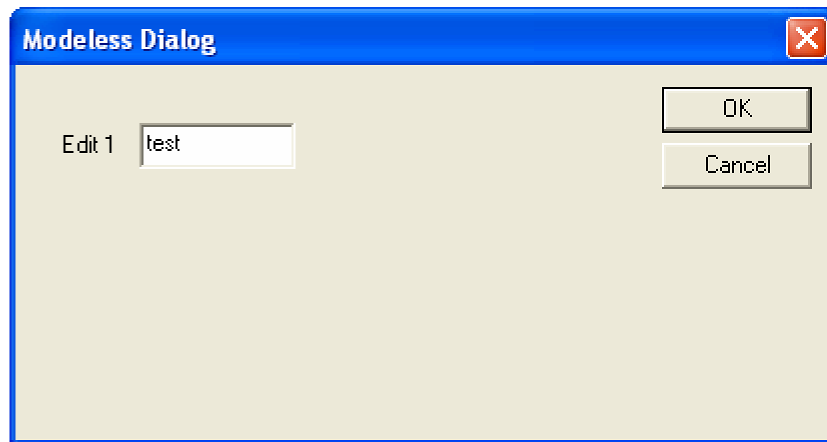

Figure 21: MYMFC9 program output, modeless dialog launched when the left mouse button clicked.

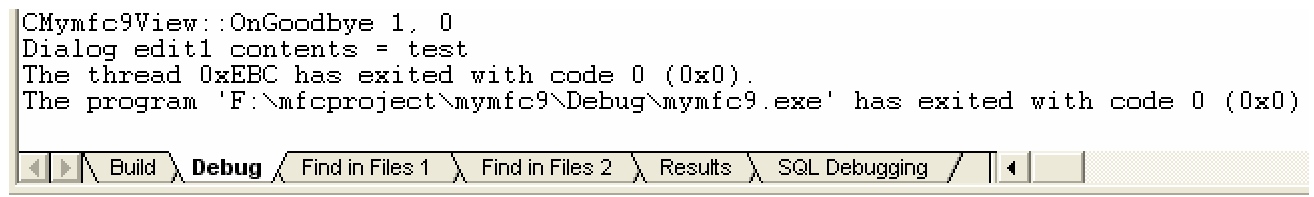

Figure 22.

If you use the MYMFC9 view and dialog classes in an MDI application, each MDI child window can have one modeless dialog. When the user closes an MDI child window, the child's modeless dialog is destroyed because the view's destructor calls the dialog destructor, which, in turn, destroys the dialog window.

The `CFormView` Class: A Modeless Dialog Alternative

If you need an application based on a single modeless dialog, the `CFormView` class will save you a lot of work and will be discussed in another Module together with the `CDocument` class, because the `CFormView` class is most useful when coupled with it.

The Windows Common Dialogs

Windows provides a group of standard user interface dialogs, and these are supported by the MFC library classes. You are probably familiar with all or most of these dialogs because so many Windows-based applications, including Visual C++, already use them. All the common dialog classes are derived from a common base class, `CCommonDialog`. A list some of the `COMDLG32` classes is shown in the following table.

Class	Purpose
CColorDialog	Allows the user to select or create a color.
CFileDialog	Allows the user to open or save a file.
CFindReplaceDialog	Allows the user to substitute one string for another.
CPageSetupDialog	Allows the user to input page measurement parameters.
CFontDialog	Allows the user to select a font from a list of available fonts.
CPrintDialog	Allows the user to set up the printer and print a document.

Table 3: Some of the COMDLG32 classes.

Here's one characteristic that all common dialogs share: they gather information from the user, but they don't do anything with it. The file dialog can help the user select a file to open, but it really just provides your program with the pathname, your program must make the call that opens the file. Similarly, a font dialog fills in a structure that describes a font, but it doesn't create the font.

Using the CFileDialog Class Directly

Using the CFileDialog class to open a file is easy. The following code opens a file that the user has selected through the dialog:

```
CFileDialog dlg(TRUE, "bmp", "*.bmp");
if (dlg.DoModal() == IDOK)
{
    CFile file;
    VERIFY(file.Open(dlg.GetPathName(), CFile::modeRead));
}
```

The first constructor parameter (TRUE) specifies that this object is a "File Open" dialog instead of a "File Save" dialog. The default file extension is **bmp**, and ***.bmp** appears first in the filename edit box. The CFileDialog::GetPathName function returns a CString object that contains the full pathname of the selected file.

Deriving from the Common Dialog Classes

Most of the time, you can use the common dialog classes directly. If you derive your own classes, you can add functionality without duplicating code. Each COMDLG32 dialog works a little differently, however. The next example is specific to the file dialog, but it should give you some ideas for customizing the other common dialogs. In the early editions of this book, the MYMFC10 example dynamically created controls inside the standard file dialog. That technique doesn't work in Win32, but the nested dialog method described here has the same effect.

Nested Dialogs

Win32 provides a way to "nest" one dialog inside another so that multiple dialogs appear as one seamless whole. You must first create a dialog resource template with a "hole" in it, typically a group box control, with the specific child window ID **stc32** (= 0x045f). Your program sets some parameters that tell COMDLG32 to use your template. In addition, your program must hook into the COMDLG32 message loop so that it gets first crack at selected notifications. When you're done with all of this, you'll notice that you have created a dialog window that is a child of the COMDLG32 dialog window, even though your template wraps COMDLG32's template.
This sounds difficult, and it is unless you use MFC. With MFC, you build the dialog resource template as described above, derive a class from one of the common dialog base classes, add the class-specific connection code in OnInitDialog(), and then happily use ClassWizard to map the messages that originate from your template's new controls.

Windows NT 3.51 uses an earlier version of the common dialogs DLL that does not support the new Windows namespace feature. The nested dialog technique illustrated in the MYMFC10 example won't work with the Windows NT 3.51 version of the file dialog.

A `CFileDialog` Example: MYMFC10

In this example, you will derive a class `CMymfc10Dialog` that adds a working **Delete All Matching Files** button to the standard file dialog. It also changes the dialog's title and changes the **Open** button's caption to **Delete** (to delete a single file). The example illustrates how you can use nested dialogs to add new controls to standard common dialogs. The new file dialog is activated as in the previous examples, by pressing the left mouse button when the mouse cursor is in the view window. Because you should be gaining skill with Visual C++, the following steps won't be as detailed as those for the earlier examples. Figure 23 shows what the dialog will look like.

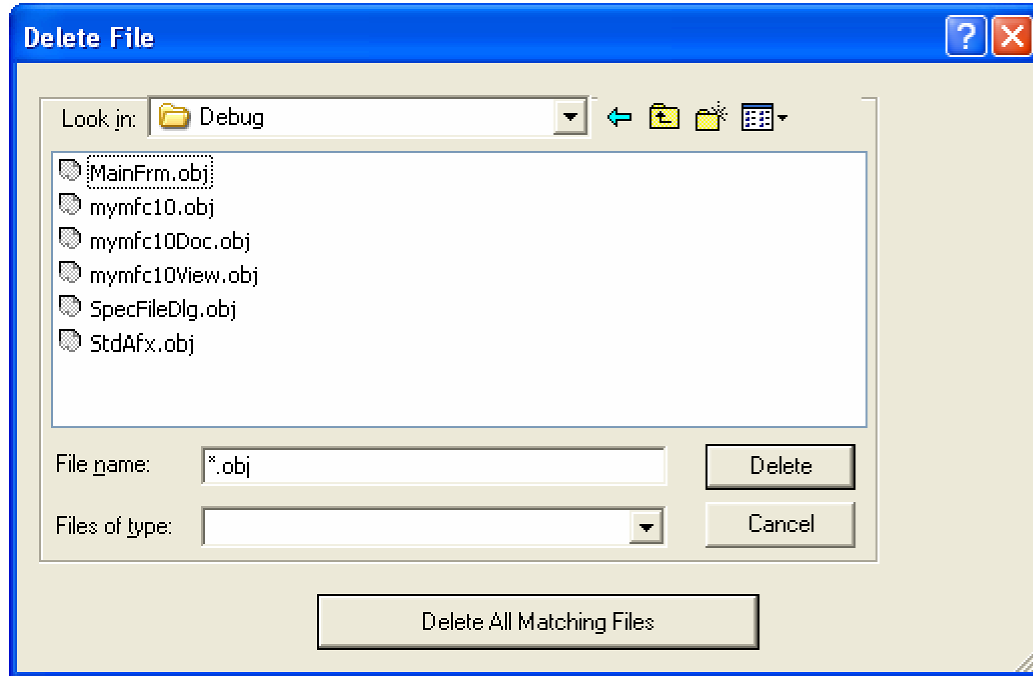

Figure 23: The MYMFC10's Delete File dialog in action.

Follow these steps to build the MYMFC10 application:

Run AppWizard to produce \mfcproject\mymfc10 project (change accordingly to directory that you have designated for your project). Accept all the defaults but two: select **Single Document** and deselect **Printing And Print Preview** and **ActiveX Controls**. The options and the default class names are shown in the next graphic.

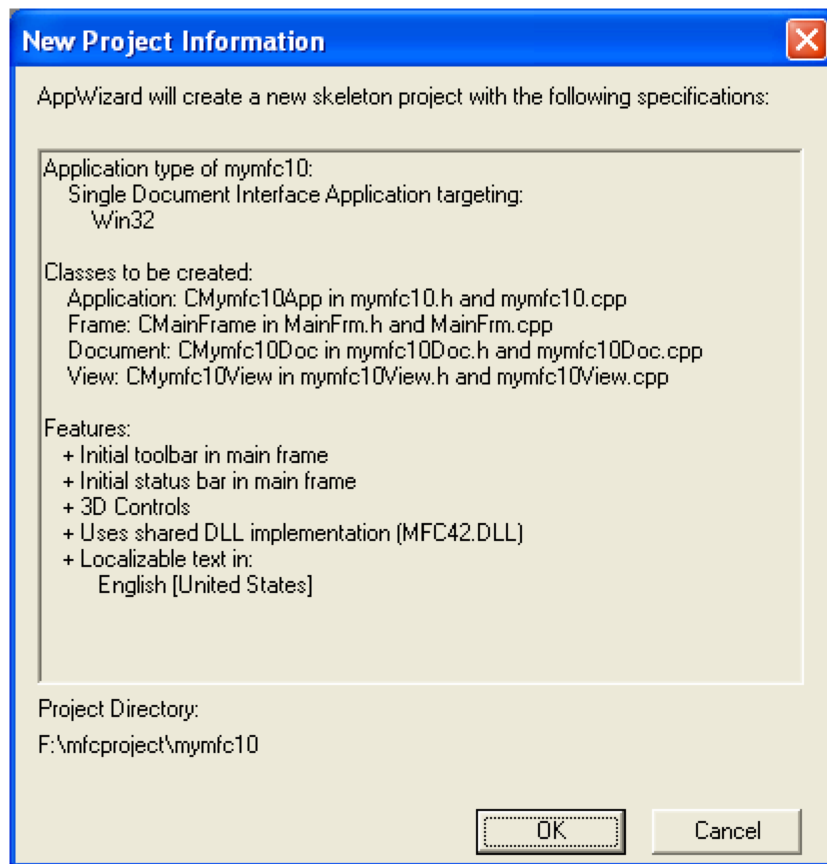

Figure 24: MYMFC10 project summary.

Use the dialog editor to create a dialog resource. Make the dialog box about 3-by-5 inches, and use the ID `IDD_FILESPECIAL`. Set the dialog's **Style** property to **Child**, its **Border** property to **None** and select its **Clip Siblings** and **Visible** properties. Delete the **OK** and **Cancel** button.

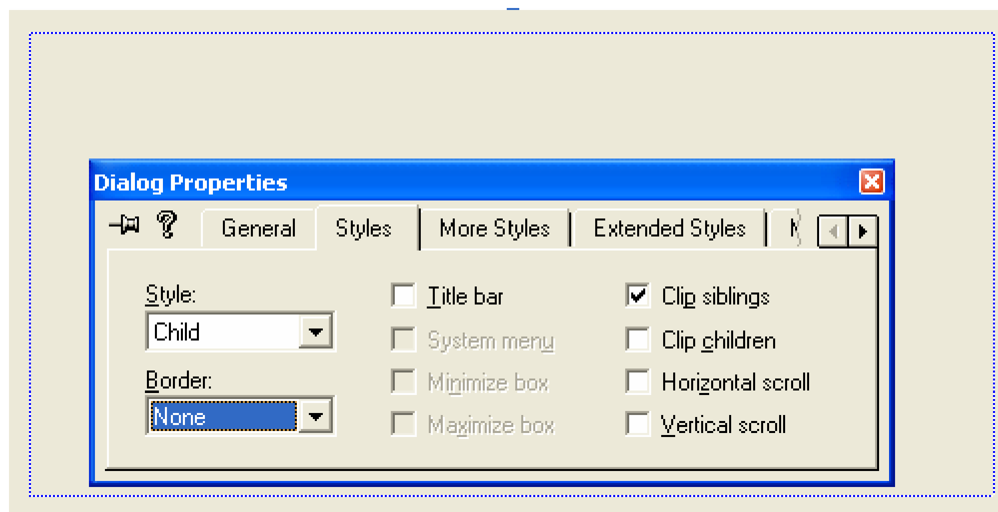

Figure 25: Modifying the dialog properties.

Create a button with ID `IDC_DELETE` and a group box with ID **`stc32`** (=0x045f in hexadecimal), as shown here.

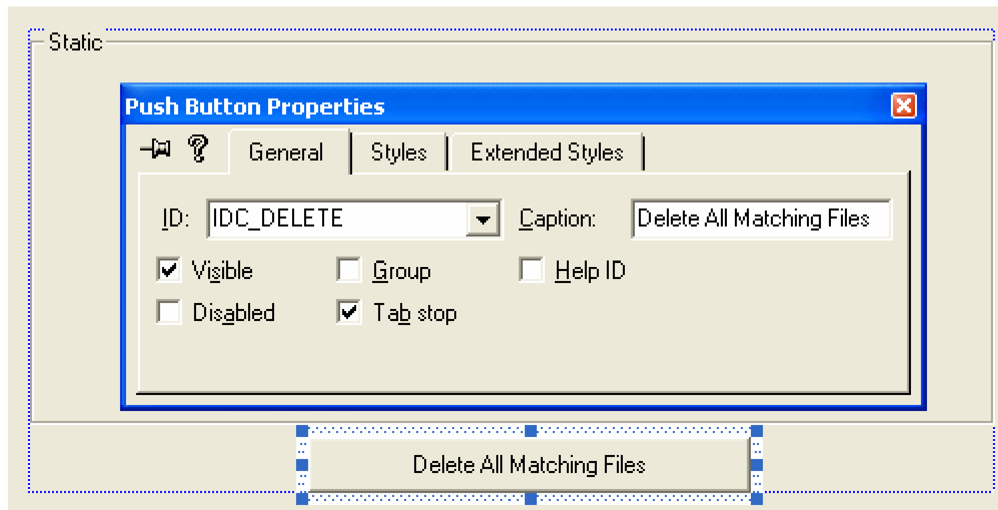

Figure 26: Modifying push button's properties.

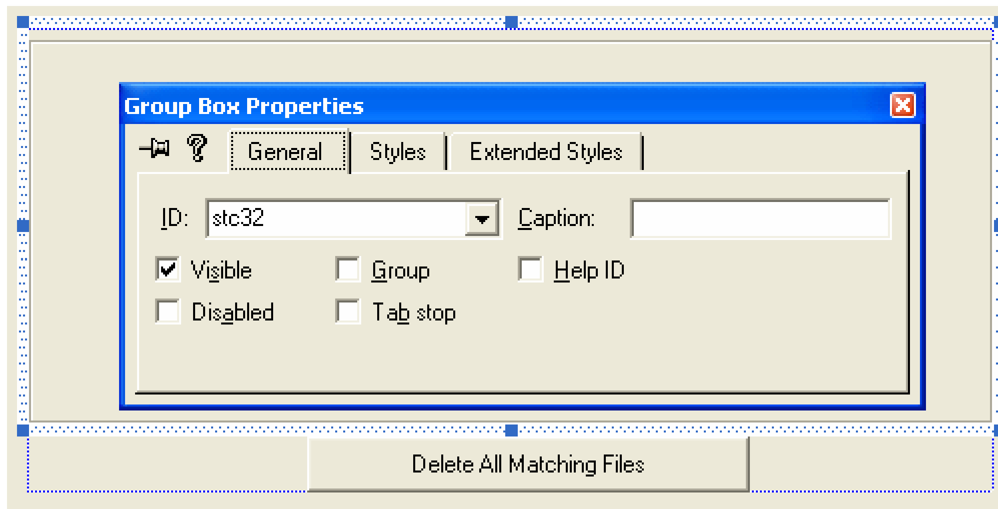

Figure 27: Modifying group box's properties.

Check your work by choosing **Resource Symbols** from the Visual C++ **View** menu. You should see a symbol list like the one shown in the graphic below.

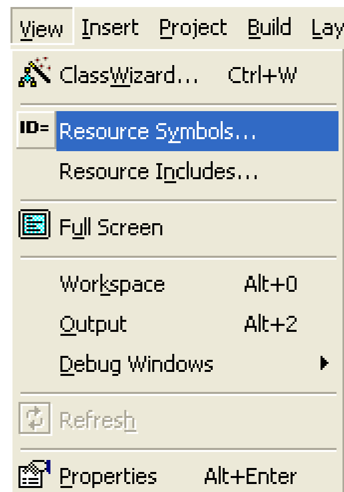

Figure 28: Viewing and adding project's resource symbols.

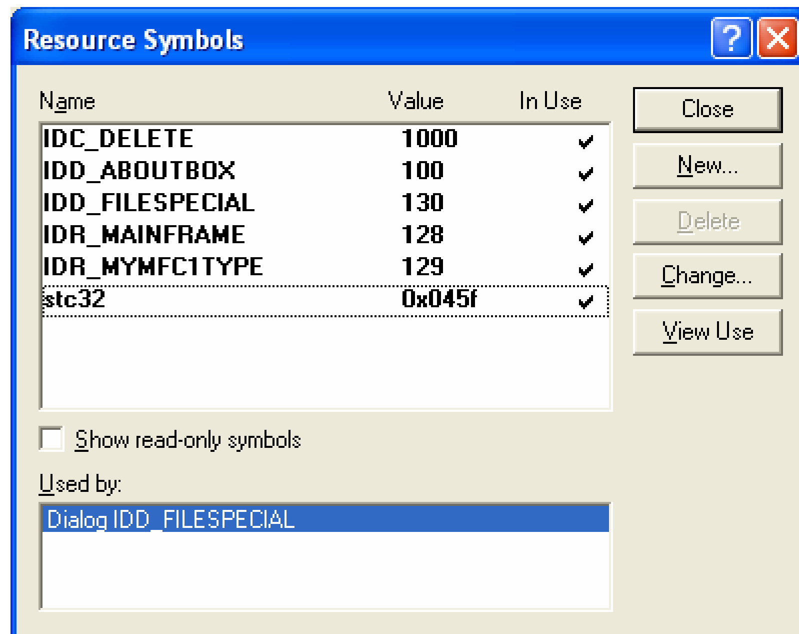

Figure 29: MYMFC10 resource symbols.

Use ClassWizard to create the `CSpecialFileDialog` class.

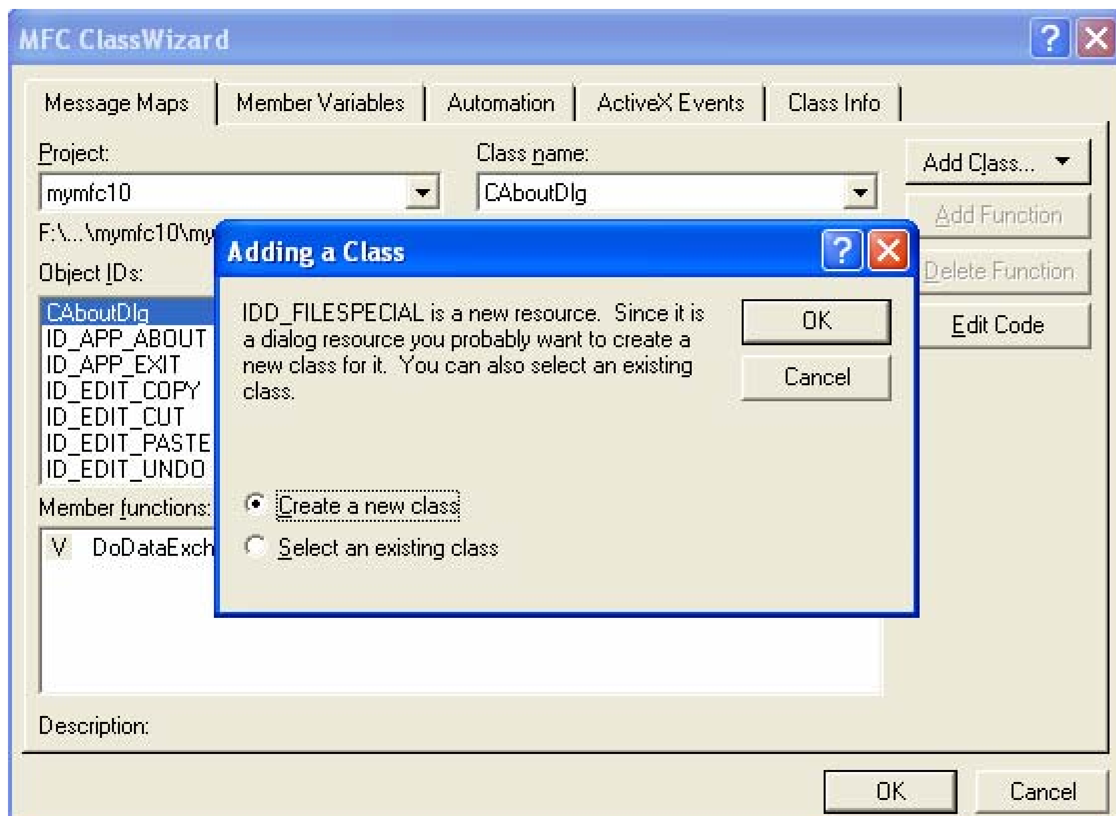

Figure 30: Creating the `CSpecialFileDialog` class.

Fill in the **New Class** dialog, as shown here, and then click the **Change** button.

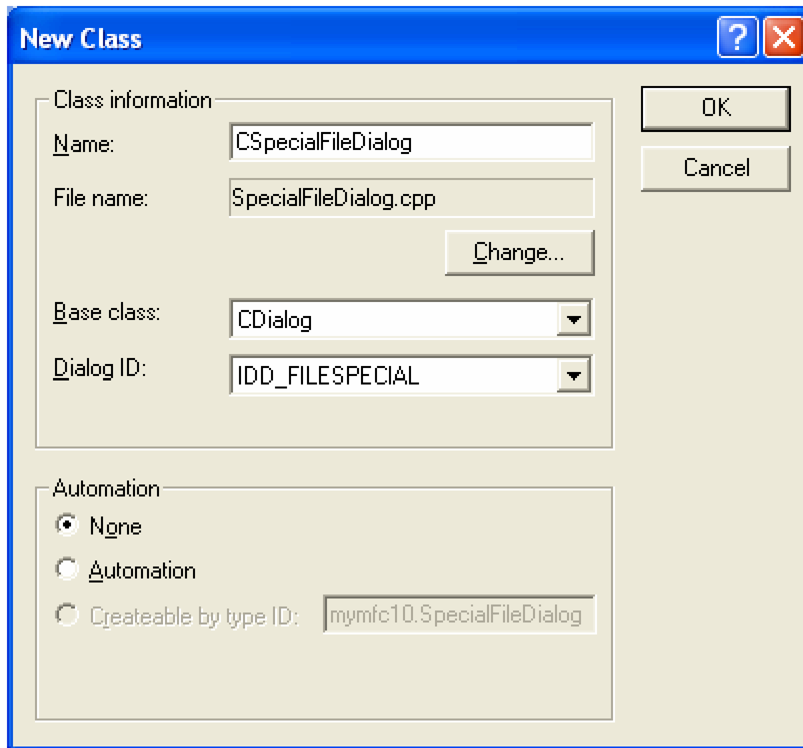

Figure 31: `CSpecialFileDialog` class information.

Change the names to **SpecFileDlg.h** and **SpecFileDlg.cpp**. Unfortunately, we cannot use the **Base Class** drop-down list to change the base class to `CFileDialog`, as that would decouple our class from the `IDD_FILESPECIAL` template. We have to change the base class by hand.

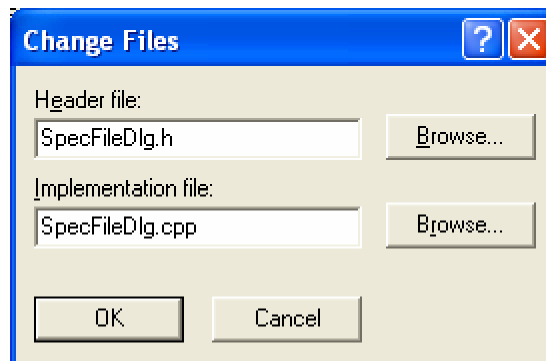

Figure 32: Changing `CSpecialFileDialog` class's header and implementation file names.

Edit the file **SpecFileDlg.h**. Change the line:

```
class CSpecialFileDialog : public CDialog
```

To

```
class CSpecialFileDialog : public CFileDialog
```

```
// CSpecialFileDialog dialog

class CSpecialFileDialog : public CFileDialog
{
// Construction
public:
```

Listing 17.

Add the following two public data members:

```
CString m_strFilename;
BOOL m_bDeleteAll;
```

```
class CSpecialFileDialog : public CFileDialog
{
// Construction
public:
    CString m_strFilename;
    BOOL m_bDeleteAll;

    CSpecialFileDialog(CWnd* pParent = NULL);
```

Listing 18.

Finally, edit the constructor declaration:

```
CSpecialFileDialog(BOOL bOpenFileDialog,
    LPCTSTR lpszDefExt = NULL,
    LPCTSTR lpszFileName = NULL,
    DWORD dwFlags = OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
    LPCTSTR lpszFilter = NULL,
    CWnd* pParentWnd = NULL);
```

```
class CSpecialFileDialog : public CFileDialog
{
// Construction
public:
    CString m_strFilename;
    BOOL m_bDeleteAll;

CSpecialFileDialog(BOOL bOpenFileDialog,
    LPCTSTR lpszDefExt = NULL,
    LPCTSTR lpszFileName = NULL,
    DWORD dwFlags = OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
    LPCTSTR lpszFilter = NULL,
    CWnd* pParentWnd = NULL);
```

Listing 19.

Replace `CDialog` with `CFileDialog` in **SpecFileDlg.h**. Choose **Replace** from Visual C++'s **Edit** menu, and replace this name globally.

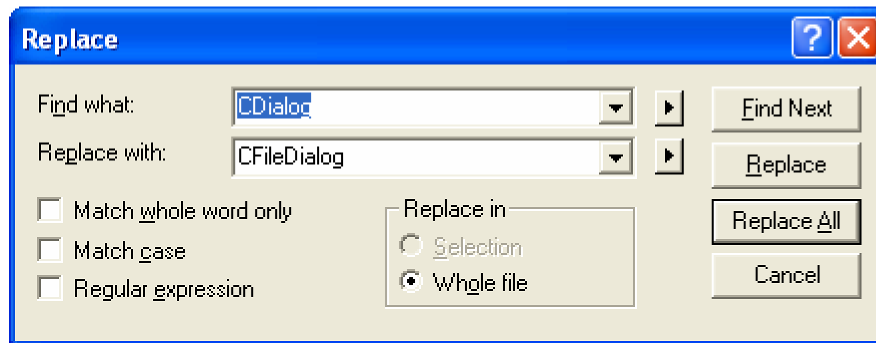

Figure 33: Replacing `CDialog` with `CFileDialog` in **SpecFileDlg.h**.

Edit the `CSpecialFileDialog` constructor in SpecFileDlg.cpp. The derived class destructor must invoke the base class constructor and initialize the `m_bDeleteAll` data member. In addition, it must set some members of the `CFileDialog` base class data member `m_ofn`, which is an instance of the Win32 `OPENFILENAME` structure. The `Flags` and `lpTemplateName` members control the coupling to your `IDD_FILESPECIAL` template, and the `lpstrTitle` member changes the main dialog box title. Edit the constructor as follows:

```
CSpecialFileDialog::CSpecialFileDialog(BOOL bOpenFileDialog, LPCTSTR lpszDefExt,
LPCTSTR lpszFileName, DWORD dwFlags,
        LPCTSTR lpszFilter, CWnd* pParentWnd) : CFileDialog(bOpenFileDialog,
lpszDefExt, lpszFileName, dwFlags, lpszFilter, pParentWnd)
{
    //{{AFX_DATA_INIT(CSpecialFileDialog)
        // NOTE: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT
    m_ofn.Flags |= OFN_ENABLETEMPLATE;
    m_ofn.lpTemplateName = MAKEINTRESOURCE(IDD_FILESPECIAL);
    m_ofn.lpstrTitle = "Delete File";
    m_bDeleteAll = FALSE;
}
```

```
// CSpecialFileDialog dialog

CSpecialFileDialog::CSpecialFileDialog(BOOL bOpenFileDialog,
        LPCTSTR lpszDefExt, LPCTSTR lpszFileName, DWORD dwFlags,
        LPCTSTR lpszFilter, CWnd* pParentWnd)
    : CFileDialog(bOpenFileDialog, lpszDefExt,
    lpszFileName, dwFlags, lpszFilter, pParentWnd)
{
    //{{AFX_DATA_INIT(CSpecialFileDialog)
        // NOTE: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT
    m_ofn.Flags |= OFN_ENABLETEMPLATE;
    m_ofn.lpTemplateName = MAKEINTRESOURCE(IDD_FILESPECIAL);
    m_ofn.lpstrTitle = "Delete File";
    m_bDeleteAll = FALSE;
}
```

Listing 20.

Map the `WM_INITDIALOG` message in the `CSpecialFileDialog` class. The `OnInitDialog()` member function needs to change the common dialog's **Open** button caption to **Delete**. The child window ID is `IDOK`.

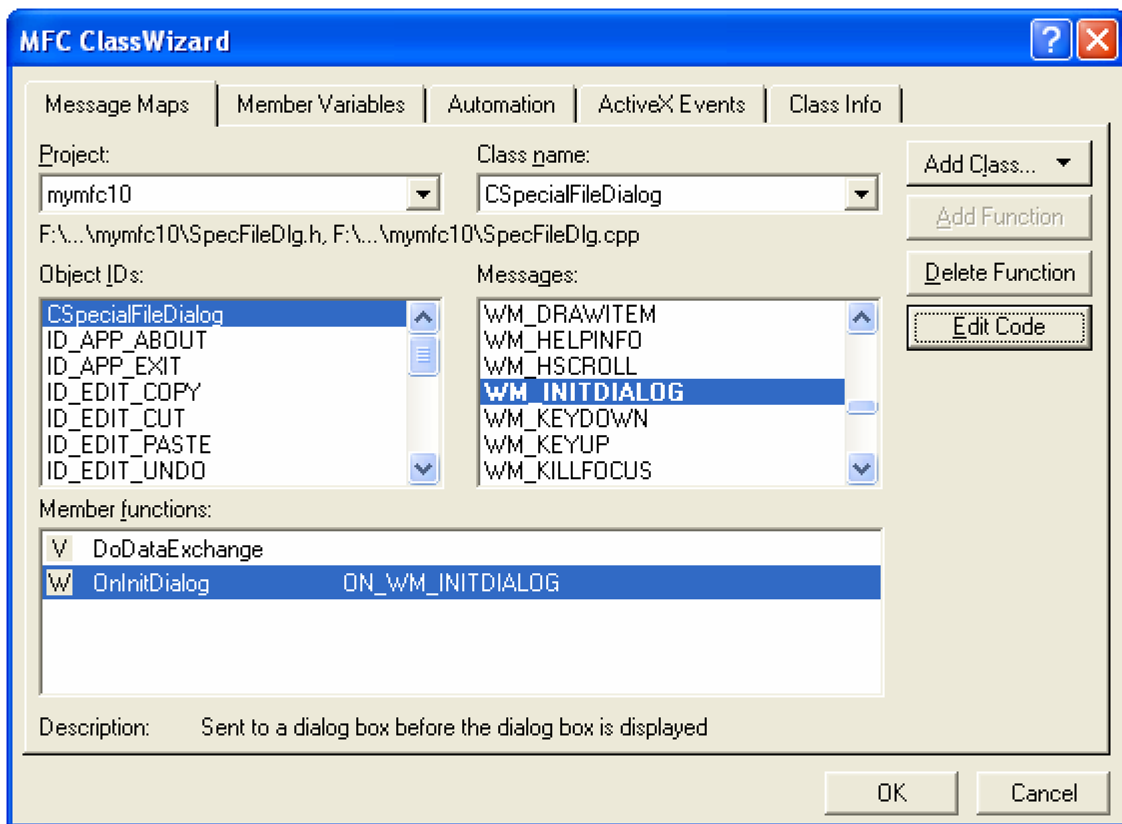

Figure 34: Mapping the `WM_INITDIALOG` message in the `CSpecialFileDialog` class.

```
BOOL CSpecialFileDialog::OnInitDialog()
{

        BOOL bRet = CFileDialog::OnInitDialog();
        if (bRet == TRUE)
        {
                GetParent()->GetDlgItem(IDOK)->SetWindowText("Delete");
        }
        return bRet;
}
```

```
// CSpecialFileDialog message handlers

BOOL CSpecialFileDialog::OnInitDialog()
{

     BOOL bRet = CFileDialog::OnInitDialog();
     if (bRet == TRUE)
     {
          GetParent()->GetDlgItem(IDOK)->SetWindowText("Delete");
     }
     return bRet;
}
```

Listing 21.

Map the new `IDC_DELETE` button (**Delete All Matching Files**) in the `CSpecialFileDialog` class.

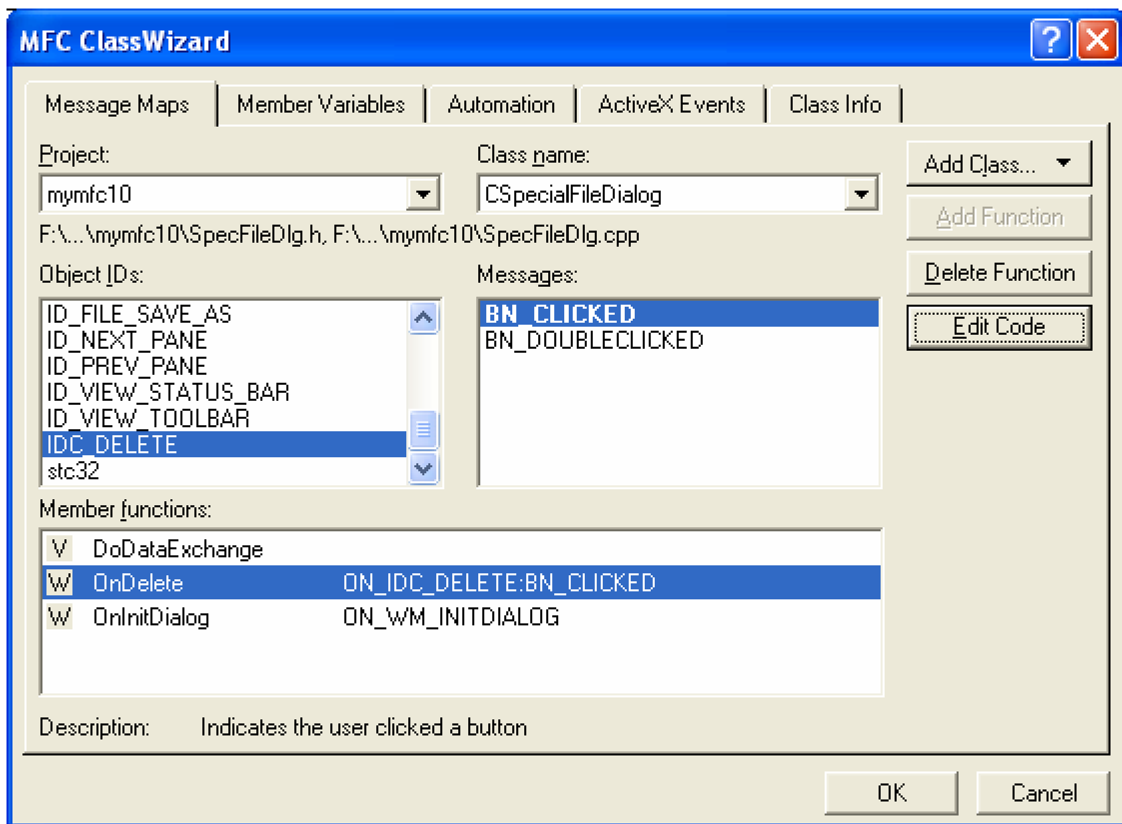

Figure 35: Mapping the new `IDC_DELETE` button in the `CSpecialFileDialog` class.

The `OnDelete()` member function sets the `m_bDeleteAll` flag and then forces the main dialog to exit as if the **Cancel** button had been clicked. The client program (in this case, the view) gets the `IDCANCEL` return from `DoModal()` and reads the flag to see whether it should delete all files. Here is the function:

```
void CSpecialFileDialog::OnDelete()
{
    m_bDeleteAll = TRUE;
    // 0x480 is the child window ID of the File Name edit control
    // (as determined by SPY++)
    GetParent()->GetDlgItem(0x480)->GetWindowText(m_strFilename);
    GetParent()->SendMessage(WM_COMMAND, IDCANCEL);
}
```

```
void CSpecialFileDialog::OnDelete()
{
    // TODO: Add your control notification handler code here
    m_bDeleteAll = TRUE;
    // 0x480 is the child window ID of the File Name edit control
    // (as determined by SPYXX)
    GetParent()->GetDlgItem(0x480)->GetWindowText(m_strFilename);
    GetParent()->SendMessage(WM_COMMAND, IDCANCEL);
}
```

Listing 22.

Add code to the virtual `OnDraw()` function in file **mymfc10View.cpp**. The `CMymfc10View` `OnDraw()` function which skeleton was generated by AppWizard should be coded as follows to prompt the user to press the mouse button:

```
void Cmymfc10View::OnDraw(CDC* pDC)
{
    pDC->TextOut(30, 30, "Press the left mouse button lol! ");
```

```
        }

// CMymfc10View drawing

void CMymfc10View::OnDraw(CDC* pDC)
{
    // TODO: add draw code for native data here
    pDC->TextOut(30, 30, "Press the left mouse button lol!");
}
```

Listing 23.

Add the `OnLButtonDown()` message handler to the `Cmymfc10View` class. Use ClassWizard to create the message handler for `WM_LBUTTONDOWN`.

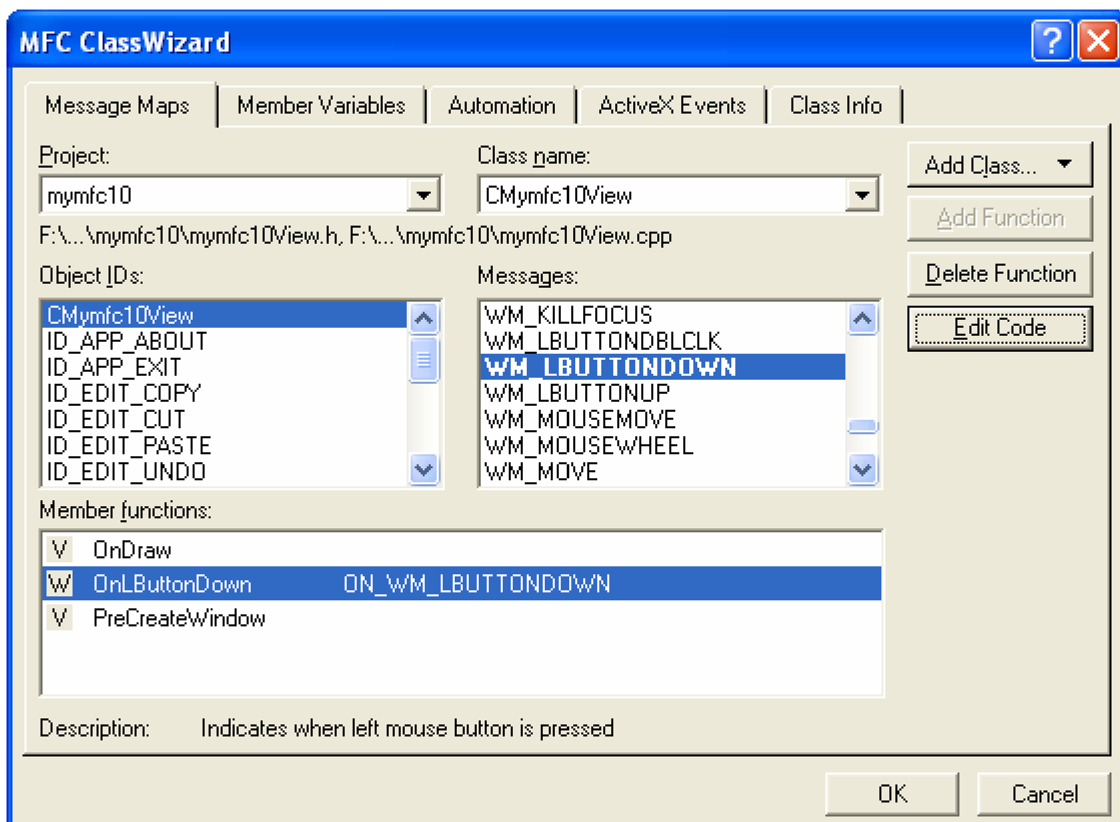

Figure 36: Adding the `OnLButtonDown()` message handler to the `Cmymfc10View` class.

And then edit the code as follows:

```
void Cmymfc10View::OnLButtonDown(UINT nFlags, Cpoint point)
{
    CspecialFileDialog dlgFile(TRUE, NULL, "*.obj");
    Cstring strMessage;
    int nModal = dlgFile.DoModal();
    if ((nModal == IDCANCEL) && (dlgFile.m_bDeleteAll))
    {
        strMessage.Format("Are you very sure you want to delete all %s files? ",
dlgFile.m_strFilename);
        if (AfxMessageBox(strMessage, MB_YESNO) == IDYES)
        {
         HANDLE h;
         WIN32_FIND_DATA fData;
```

```
        while((h = ::FindFirstFile(dlgFile.m_strFilename, &fData)) !=
(HANDLE)0xFFFFFFFF)
            { // no MFC equivalent
                if (::DeleteFile(fData.cFileName) == FALSE)
                {
                    strMessage.Format("Unable to delete file %s\n", fData.cFileName);
                    AfxMessageBox(strMessage);
                    break;
                }
            }
        }
    }
    else if (nModal == IDOK)
    {
        Cstring strSingleFilename = dlgFile.GetPathName();
        strMessage.Format("Are you very sure you want to delete %s?", strSingleFilename);
        if (AfxMessageBox(strMessage, MB_YESNO) == IDYES)
        {
            Cfile::Remove(strSingleFilename);
        }
    }
}
```

```
void CMymfc10View::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    CSpecialFileDialog dlgFile(TRUE, NULL, "*.obj");
    CString strMessage;
    int nModal = dlgFile.DoModal();
    if ((nModal == IDCANCEL) && (dlgFile.m_bDeleteAll))
    {
        strMessage.Format("Are you very sure you want to delete all %s files?",
            dlgFile.m_strFilename);
        if (AfxMessageBox(strMessage, MB_YESNO) == IDYES)
        {
         HANDLE h;
         WIN32_FIND_DATA fData;
         while((h = ::FindFirstFile(dlgFile.m_strFilename, &fData))
             != (HANDLE)0xFFFFFFFF)
            { // no MFC equivalent
                if (::DeleteFile(fData.cFileName) == FALSE)
                {
                    strMessage.Format("Unable to delete file %s\n",
                        fData.cFileName);
                    AfxMessageBox(strMessage);
                    break;
                }
            }
        }
    }
    else if (nModal == IDOK)
    {
        CString strSingleFilename = dlgFile.GetPathName();
        strMessage.Format("Are you very sure you want to delete %s?",
            strSingleFilename);
        if (AfxMessageBox(strMessage, MB_YESNO) == IDYES)
        {
            CFile::Remove(strSingleFilename);
        }
    }
}
```

Listing 24.

Remember that common dialogs just gather data. Since the view is the client of the dialog, the view must call `DoModal()` or the file dialog object and then figure out what to do with the information returned. In this case, the view

has the return value from DoModal() (either IDOK or IDCANCEL) and the value of the public m_bDeleteAll data member, and it can call various CfileDialog member functions such as GetPathName(). If DoModal() returns IDCANCEL and the flag is TRUE, the function makes the Win32 file system calls necessary to delete all the matching files. If DoModal() returns IDOK, the function can use the MFC Cfile() functions to delete an individual file.

Using the global AfxMessageBox() function is a convenient way to pop up a simple dialog that displays some text and then queries the user for a Yes/No answer. Finally add the include the statement:

```
#include "SpecFileDlg.h"
```

After the line:

```
#include "mymfc10View.h"
```

```
// mymfc10View.cpp : implementation

#include "stdafx.h"
#include "mymfc10.h"

#include "mymfc10Doc.h"
#include "mymfc10View.h"
#include "SpecFileDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```

Listing 25.

Build and test the application. Build and run MYMFC10. Pressing the left mouse button should bring up the **Delete File** dialog, and you should be able to use it to navigate through the disk directory and to delete files. Be careful not to delete your important source files!

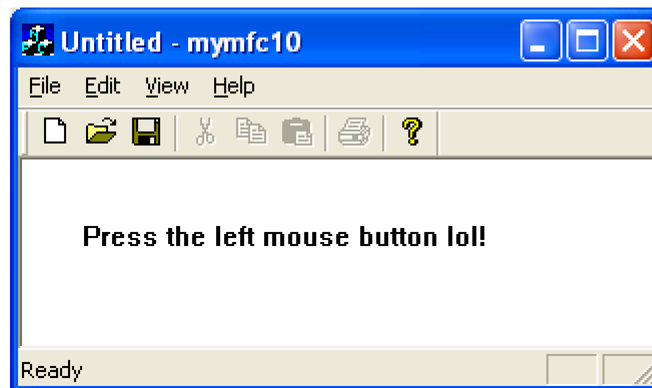

Figure 37: MYMFC10 program output.

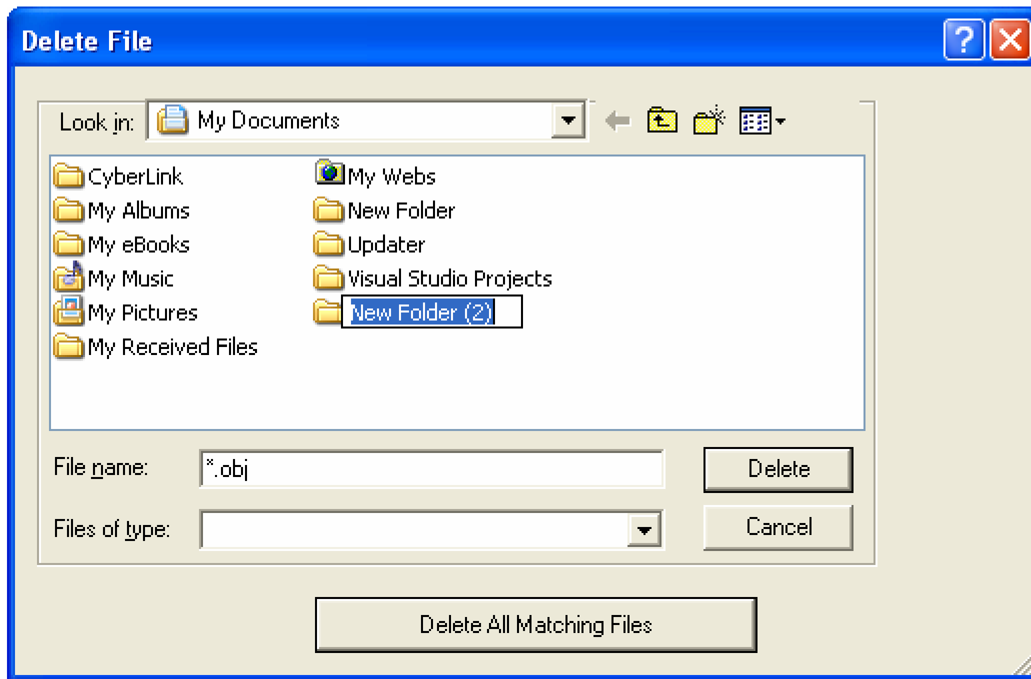

Figure 38: MYMFC10 program output, when left mouse button is clicked, launching a delete file dialog.

Other Customization for CfileDialog

In the MYMFC10 example, you added a pushbutton to the dialog. It's easy to add other controls too. Just put them in the resource template, and if they are standard Windows controls such as edit controls or list boxes, you can use ClassWizard to add data members and DDX/DDV code to your derived class. The client program can set the data members before calling `DoModal()`, and it can retrieve the updated values after `DoModal()` returns. Even if you don't use nested dialogs, two windows are still associated with a `CfileDialog` object. Suppose you have overridden `OnInitDialog()` in a derived class and you want to assign an icon to the file dialog. You must call `CWnd::GetParent` to get the top-level window, just as you did in the MYMFC10 example. Here's the code:

```
HICON hIcon = AfxGetApp()->LoadIcon(ID_MYICON);
GetParent()->SetIcon(hIcon, TRUE);    // Set big icon
GetParent()->SetIcon(hIcon, FALSE);   // Set small icon
```

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.