Module 11: Serialization: Reading and Writing Documents—SDI Applications

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. You can compare the standard C file I/O, standard C++ file I/O and Win32 directory, file and access controls with the MFC serialization. So many things lor! Similar but not same. Those links also given at the end of this tutorial.

Reading and Writing Documents: SDI Applications

As you've probably noticed, every AppWizard-generated program has a File menu that contains the familiar New, Open, Save, and Save As commands. In this module, you'll learn how to make your application respond to **read** and **write** documents.

Here we'll stick with the **Single Document Interface** (SDI) application because it's familiar territory. Module 12 introduces the **Multiple Document Interface** (MDI) application, which is more flexible in its handling of documents and files. In both modules, you'll get a heavy but necessary dose of application-framework theory; you'll learn a lot

about the various helper classes that have been concealed up to this point. The going will be rough, but believe me, you must know the details to get the most out of the application framework.
This module's example, MYMFC17, is an SDI application based on the MYMFC16 example from the previous module. It uses the student list document with a CFormView-derived view class. Now the student list can be written to and read from disk through a process called serialization. Module 12 shows you how to use the same view and document classes to make an MDI application.

Serialization: What Is It?

The term "serialization" might be new to you, but it's already seen some use in the world of object-oriented programming. The idea is that **objects can be persistent**, which means they **can be saved** on disk when a program exits and then **can be restored** when the program is restarted. This **process of saving and restoring objects** is called serialization. In the MFC library, designated classes have a member function named `Serialize()`. When the application framework calls Serialize() for a particular object, for example, an object of class CStudent, the data for the student is either saved on disk or read from disk. In the MFC library, serialization is not a substitute for a database management system. All the objects associated with a document are **sequentially read** from or **written** to a single disk file. It's not possible to access individual objects at random disk file addresses. If you need database capability in your application, consider using the **Microsoft Open Database Connectivity** (ODBC) software or **Data Access Objects** (DAO). The MFC framework already uses **structured storage** (for database) for container programs that support embedded objects.

Disk Files and Archives

How do you know whether `Serialize()` should read or write data? How is `Serialize()` connected to a disk file? With the MFC library, objects of class CFile represent disk files. A `CFile` object encapsulates the binary file handle that you get through the Win32 function `CreateFile()`. This is not the buffered `FILE` pointer that you'd get with a call to the C runtime `fopen()` function; rather, it's a handle to a **binary file**. The application framework uses this file handle for Win32 `ReadFile()`, `WriteFile()`, and `SetFilePointer()` calls.
If your application does no direct disk I/O but instead relies on the serialization process, you can avoid direct use of `CFile` objects. Between the `Serialize()` function and the `CFile` object is an archive object of class `CArchive`, as shown in Figure 1.
The `CArchive` object buffers data for the `CFile` object, and it maintains an internal flag that indicates whether the archive is storing (writing to disk) or loading (reading from disk). Only one active archive is associated with a file at any one time. The application framework takes care of constructing the `CFile` and `CArchive` objects, opening the disk file for the `CFile` object and associating the archive object with the file. All you have to do (in your `Serialize()` function) is load data from or store data in the archive object. The application framework calls the document's `Serialize()` function during the **File Open** and **File Save** processes.

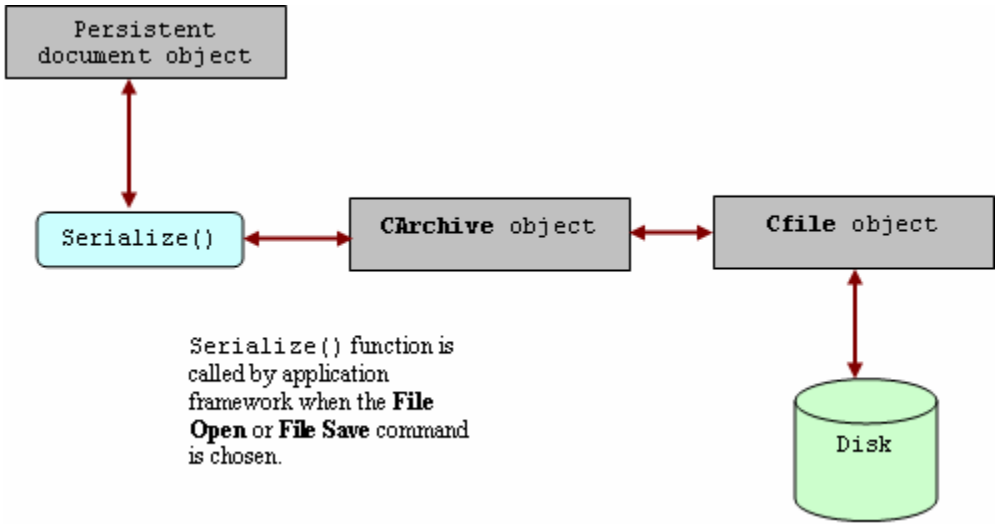

Figure 1: The serialization process.

Making a Class Serializable

A serializable class must be derived directly or indirectly from `CObject`. In addition (with some exceptions), the class declaration must contain the `DECLARE_SERIAL` macro call, and the class implementation file must contain the `IMPLEMENT_SERIAL` macro call. See the Microsoft Foundation Class Reference for a description of these macros. This module's `CStudent` class example is modified from the class in Module 10 to include these macros.

Writing a Serialize Function

In Module 10, you saw a `CStudent` class, derived from `CObject`, with these data members:

```
public:
       CString m_strName;
       int     m_nGrade;
```

Now, your job is to write a `Serialize()` member function for `CStudent`. Because `Serialize()` is a virtual member function of class `CObject`, you must be sure that the return value and parameter types match the `CObject` declaration. The `Serialize()` function for the `CStudent` class is below.

```
void CStudent::Serialize(CArchive& ar)
{
    TRACE("Entering CStudent::Serialize\n");
    if (ar.IsStoring())
    {
        ar << m_strName << m_nGrade;
    }
    else
    {
        ar >> m_strName >> m_nGrade;
    }
}
```

Most serialization functions call the `Serialize()` functions of their base classes. If `CStudent` were derived from `CPerson`, for example, the first line of the `Serialize()` function would be:

```
CPerson::Serialize(ar);
```

The `Serialize()` function for `CObject` (and for `CDocument`, which doesn't override it) doesn't do anything useful, so there's no need to call it. Notice that ar is a `CArchive` reference parameter that identifies the application's archive object. The `CArchive::IsStoring` member function tells us whether the archive is currently being used for storing or loading. The `CArchive` class has overloaded **insertion operators** (<<) and **extraction operators** (>>) for many of the C++ built-in types, as shown in the following table.

Type	Description
`BYTE`	8 bits, unsigned
`WORD`	16 bits, unsigned
`LONG`	32 bits, signed
`DWORD`	32 bits, unsigned
`float`	32 bits
`double`	64 bits, IEEE standard
`int`	32 bits, signed
`short`	16 bits, signed
`char`	8 bits, unsigned
`unsigned`	32 bits, unsigned

Table 1

The insertion operators are **overloaded for values**; the extraction operators are **overloaded for references**. Sometimes you must use a cast to satisfy the compiler. Suppose you have a data member `m_nType` that is an enumerated type. Here's the code you would use:

```
ar << (int) m_nType;
ar >> (int&) m_nType;
```

MFC classes that are not derived from `CObject`, such as `CString` and `CRect`, have their own overloaded insertion and extraction operators for `CArchive`.

Loading from an Archive: Embedded Objects vs. Pointers

Now suppose your `CStudent` object has other objects embedded in it, and these objects are not instances of standard classes such as `CString`, `CSize`, and `CRect`. Let's add a new data member to the `CStudent` class:

```
public:
    CTranscript m_transcript;
```

Assume that `CTranscript` is a custom class, derived from `CObject`, with its own `Serialize()` member function. There's no overloaded << or >> operator for `CObject`, so the `CStudent::Serialize` function now becomes:

```
void CStudent::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        ar << m_strName << m_nGrade;
    }
    else
    {
        ar >> m_strName >> m_nGrade;
    }
    m_transcript.Serialize(ar);
}
```

Before the `CStudent::Serialize` function can be called to load a student record from the archive, a `CStudent` object must exist somewhere. The embedded `CTranscript` object `m_transcript` is constructed along with the `CStudent` object before the call to the `CTranscript::Serialize` function. When the virtual `CTranscript::Serialize` function does get called, it can load the archived transcript data into the embedded `m_transcript` object. If you're looking for a rule, here it is: always make a direct call to `Serialize()` for embedded objects of classes derived from `CObject`. Suppose that, instead of an embedded object, your `CStudent` object contained a `CTranscript` pointer data member such as this:

```
public:
    CTranscript* m_pTranscript;
```

You could use the `Serialize()` function, as shown below, but as you can see, you must construct a new `CTranscript` object yourself.

```
void CStudent::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
        ar << m_strName << m_nGrade;
    else
    {
        m_pTranscript = new CTranscript;
        ar >> m_strName >> m_nGrade;
    }
    m_pTranscript->Serialize(ar);
}
```

Because the CArchive insertion and extraction operators are indeed overloaded for CObject pointers, you could write Serialize() this way instead:

```
void CStudent::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
        ar << m_strName << m_nGrade << m_pTranscript;
    else
        ar >> m_strName >> m_nGrade >> m_pTranscript;
}
```

But how is the CTranscript object constructed when the data is loaded from the archive? That's where the DECLARE_SERIAL and IMPLEMENT_SERIAL macros in the CTranscript class come in. When the CTranscript object is written to the archive, the macros ensure that the class name is written along with the data. When the archive is read, the class name is read in and an object of the correct class is dynamically constructed, under the control of code generated by the macros. Once the CTranscript object has been constructed, the overridden Serialize() function for CTranscript can be called to do the work of reading the student data from the disk file. Finally the CTranscript pointer is stored in the m_pTranscript data member. To avoid a memory leak, you must be sure that m_pTranscript does not already contain a pointer to a CTranscript object. If the CStudent object was just constructed and thus was not previously loaded from the archive, the transcript pointer will be null. The insertion and extraction operators do not work with embedded objects of classes derived from CObject, as shown here:

```
ar >> m_strName >> m_nGrade >> &m_transcript; // Don't try this
```

Serializing Collections

Because all collection classes are derived from the CObject class and the collection class declarations contain the DECLARE_SERIAL macro call, you can conveniently serialize collections with a call to the collection class's Serialize() member function. If you call Serialize() for a CObList collection of CStudent objects, for example, the Serialize() function for each CStudent object will be called in turn. You should, however, remember the following specifics about loading collections from an archive:

- If a collection contains pointers to objects of mixed classes (all derived from CObject), the individual class names are stored in the archive so that the objects can be properly constructed with the appropriate class constructor.
- If a container object, such as a document, contains an embedded collection, loaded data is appended to the existing collection. You might need to empty the collection before loading from the archive. This is usually done in the document's virtual DeleteContents() function, which is called by the application framework.
- When a collection of CObject pointers is loaded from an archive, the following processing steps take place for each object in the collection:

    1. The object's class is identified.
    2. Heap storage is allocated for the object.
    3. The object's data is loaded into the newly allocated storage.
    4. A pointer to the new object is stored in the collection.

The MYMFC17 example shows serialization of an embedded collection of CStudent records.

The Serialize() Function and the Application Framework

OK, so you know how to write Serialize() functions, and you know that these function calls can be nested. But do you know when the first Serialize() function gets called to start the serialization process? With the application framework, everything is keyed to the document (the object of a class derived from CDocument). When you choose Save or Open from the File menu, the application framework creates a CArchive object (and an underlying CFile object) and then calls your document class's Serialize() function, passing a reference to the CArchive object. Your derived document class Serialize() function then serializes each of its non-temporary data members. If you

take a close look at any AppWizard-generated document class, you'll notice that the class includes the DECLARE_DYNCREATE and IMPLEMENT_DYNCREATE macros rather than the DECLARE_SERIAL and IMPLEMENT_SERIAL macros. The SERIAL macros are unneeded because document objects are never used in conjunction with the CArchive extraction operator or included in collections; the application framework calls the document's Serialize() member function directly. You should include the DECLARE_SERIAL and IMPLEMENT_SERIAL macros in all other serializable classes.

The SDI Application

You've seen many SDI applications that have one document class and one view class. We'll stick to a single view class in this module, but we'll explore the interrelationships among the application object, the main frame window, the document, the view, the document template object, and the associated string and menu resources.

The Windows Application Object

For each of your applications, AppWizard has been quietly generating a class derived from CWinApp. It has also been generating a statement such as this:

```
CMyApp theApp;
```

```
/////////////////////////////////////////////
// The one and only CMymfc17App object

CMymfc17App theApp;
/////////////////////////////////////////////
// CMymfc17App initialization
```

Listing 1.

What you're seeing here is the mechanism that starts an MFC application. The class CMyApp is derived from the class CWinApp, and theApp is a globally declared instance of the class. This global object is called the Windows application object. Here's a summary of the startup steps in a Microsoft Windows MFC library application:

1. Windows loads your program into memory.
2. The global object theApp is constructed. All globally declared objects are constructed immediately when the program is loaded.
3. Windows calls the global function WinMain(), which is part of the MFC library. WinMain() is equivalent to the non-Windows main function, each is a main program entry point.
4. WinMain() searches for the one and only instance of a class derived from CWinApp.
5. WinMain() calls the InitInstance() member function for theApp, which is overridden in your derived application class.
6. Your overridden InitInstance() function starts the process of loading a document and displaying the main frame and view windows.
7. WinMain() calls the Run() member function for theApp, which starts the processes of dispatching window messages and command messages.

You can override another important CWinApp member function. The ExitInstance() function is called when the application terminates, after all its windows are closed. Windows allows multiple instances of programs to run. The InitInstance() function is called each time a program instance starts up. In Win32, each instance runs as an independent process. It's only incidental that the same code is mapped to the virtual memory address space of each process. If you want to locate other running instances of your program, you must either call the Win32 FindWindow() function or set up a shared data section or memory-mapped file for communication.

The Document Template Class

If you look at the InitInstance() function that AppWizard generates for your derived application class, you'll see that the following statements are featured:

```
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
    IDR_MAINFRAME,
    RUNTIME_CLASS(CStudentDoc),
    RUNTIME_CLASS(CMainFrame),       // main SDI frame window
    RUNTIME_CLASS(CStudentView));
AddDocTemplate(pDocTemplate);
```

```
// Register the application's document templates.
//  Document templates serve as the connection between
//  documents, frame windows and views.

CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
    IDR_MAINFRAME,
    RUNTIME_CLASS(CMymfc17Doc),
    RUNTIME_CLASS(CMainFrame),       // main SDI frame window
    RUNTIME_CLASS(CMymfc17View));
AddDocTemplate(pDocTemplate);
```

Listing 1.

Unless you start doing fancy things with splitter windows and multiple views, this is the only time you'll actually see a document template object. In this case, it's an object of class `CSingleDocTemplate`, which is derived from `CDocTemplate`. The `CSingleDocTemplate` class applies only to SDI applications because SDI applications are limited to one document object. `AddDocTemplate()` is a member function of class `CWinApp`.
The `AddDocTemplate()` call, together with the document template constructor call, establishes the relationships among classes, the application class, the document class, the view window class, and the main frame window class. The application object exists, of course, before template construction, but the document, view, and frame objects are not constructed at this time. The application framework later dynamically constructs these objects when they are needed. This dynamic construction is a sophisticated use of the C++ language. The `DECLARE_DYNCREATE` and `IMPLEMENT_DYNCREATE` macros in a class declaration and implementation enable the MFC library to construct objects of the specified class dynamically. If this dynamic construction capability weren't present, more relationships among your application's classes would have to be hard-coded. Your derived application class, for example, would need code for constructing document, view, and frame objects of your specific derived classes. This would compromise the object-oriented nature of your program.
With the template system, all that's required in your application class is use of the `RUNTIME_CLASS` macro. Notice that the target class's declaration must be included for this macro to work.
Figure 2 illustrates the relationships among the various classes, and Figure 3 illustrates the object relationships. An SDI application can have only one template (and associated class groups), and when the SDI program is running, there can be only one document object and only one main frame window object.

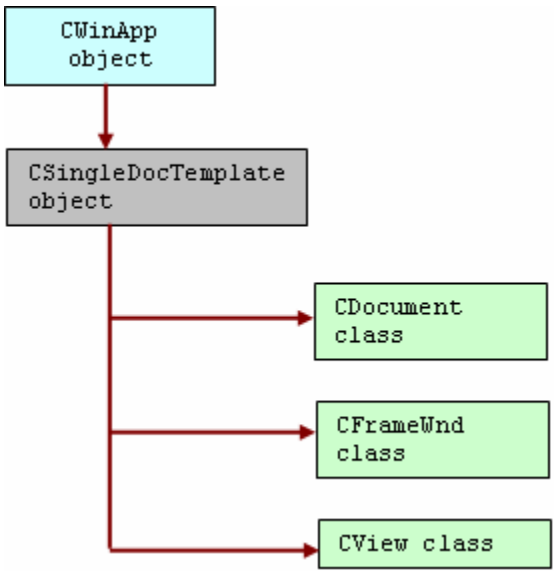

Figure 2: Class relationships.

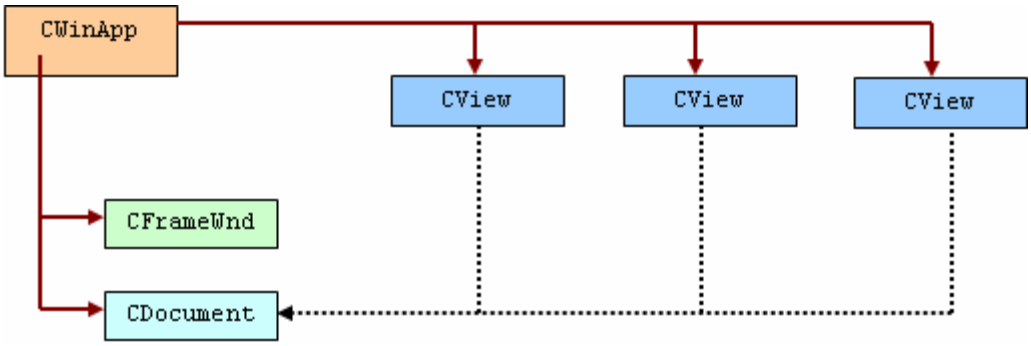

Figure 3: Object relationships.

The MFC library dynamic construction capability was designed before the runtime type identification (RTTI) feature was added to the C++ language. The original MFC implementation goes beyond RTTI, and the MFC library continues to use it for dynamic object construction.

The Document Template Resource

The first `AddDocTemplate()` parameter is `IDR_MAINFRAME`, the identifier for a string table resource. Here is the corresponding string that AppWizard generates for MYMFC17 in the application's RC file:

```
IDR_MAINFRAME
    "mymfc17\n"                 // application window caption
    "\n"                        // root for default document name
                                // ("Untitled" used if none provided)
    "Mymfc1\n"                  // document type name
    "Mymfc1 Files (*.myext)\n" // document type description and filter
    ".myext\n"                  // extension for documents of this type
    "Mymfc17.Document\n"        // Registry file type ID
    "Mymfc1 Document"           // Registry file type description
```

You can see this by double clicking the **String Table** in **ResourceView** and `IDR_MAINFRAME` as shown below.

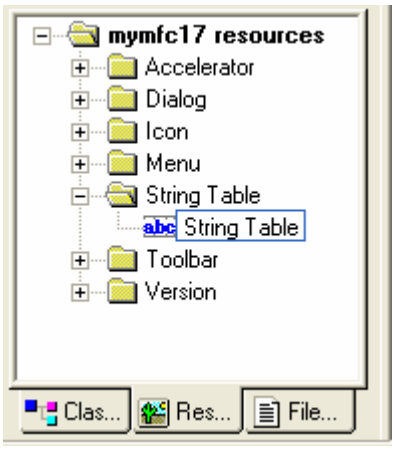

Figure 4: String table in ResourceView.

ID	Value	Caption
IDR_MAINFRAME	128	mymfc17\n\nMymfc1\nMymfc1 Files (*.myext)\n.myext\nMymfc17.Document\nMymfc1 Document

Figure 5: String for `IDR_MAINFRAME`.

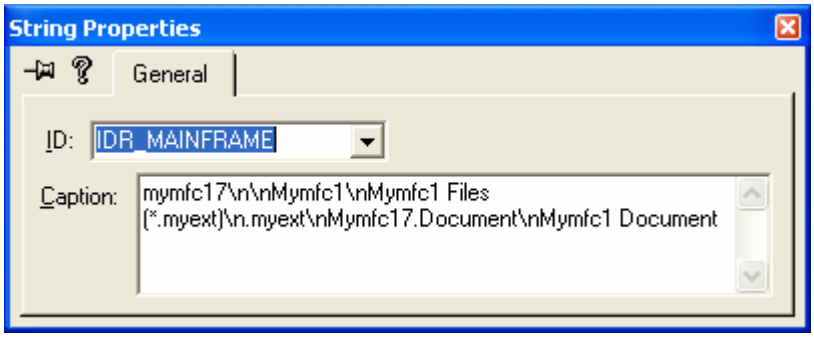

Figure 6: String properties for `IDR_MAINFRAME` (double-clicking the previous figure).

The resource compiler won't accept the string concatenations as shown above. If you examine the **mymfc17.rc** file, you'll see the substrings combined in one long string. `IDR_MAINFRAME` specifies one string that is separated into substrings by newline characters (**\n**). The substrings show up in various places when the application executes. The string `myext` is the default document file extension specified to AppWizard.
The `IDR_MAINFRAME` ID, in addition to specifying the application's strings, identifies the application's icon, toolbar resources, and menu. AppWizard generates these resources, and you can maintain them with the resource editors. So now you've seen how the `AddDocTemplate()` call ties all the application elements together. Be aware, though, that no windows have been created yet and therefore nothing appears on the screen.

Multiple Views of an SDI Document

Providing multiple views of an SDI document is a little more complicated. You could provide a menu item that allows the user to choose a view, or you could allow multiple views in a splitter window. Module 14 shows you how to implement both techniques.

Creating an Empty Document: The `CWinApp::OnFileNew` Function

After your application class's `InitInstance()` function calls the `AddDocTemplate()` member function, it calls `OnFileNew()` (indirectly through `CWinApp::ProcessShellCommand`), another important `CWinApp` member function. `OnFileNew()` sorts through the web of interconnected class names and does the following:

1. Constructs the document object but does not attempt to read data from disk.
2. Constructs the main frame object (of class `CMainFrame`); also creates the main frame window but does not show it. The main frame window includes the `IDR_MAINFRAME` menu, the toolbar, and the status bar.
3. Constructs the view object; also creates the view window but doesn't show it.
4. Establishes connections among the document, main frame, and view objects. Do not confuse these object connections with the class connections established by the call to `AddDocTemplate()`.
5. Calls the virtual `CDocument::OnNewDocument` member function for the document object, which calls the virtual `DeleteContents()` function.
6. Calls the virtual `CView::OnInitialUpdate` member function for the view object.
7. Calls the virtual `CFrameWnd::ActivateFrame` for the frame object to show the main frame window together with the menus, view window, and control bars.

Some of the functions listed above are not called directly by `OnFileNew()` but are called indirectly through the application framework.
In an SDI application, the document, main frame, and view objects are created only once, and they last for the life of the program. The `CWinApp::OnFileNew` function is called by InitInstance. It's also called in response to the user choosing the File New menu item. In this case, `OnFileNew()` must behave a little differently. It can't construct the document, frame, and view objects because they're already constructed. Instead, it reuses the existing document object and performs steps 5, 6, and 7 above. Notice that `OnFileNew()` always calls `DeleteContents()` (indirectly) to empty the document.

The Document Class's `OnNewDocument()` Function

You've seen the view class `OnInitialUpdate()` member function and the document class `OnNewDocument()` member function in Module 10. If an SDI application didn't reuse the same document object, you wouldn't need `OnNewDocument()` because you could perform all document initialization in your document class constructor. Now you must override `OnNewDocument()` to initialize your document object each time the user chooses File New or File Open. AppWizard helps you by providing a skeleton function in the derived document class it generates.
It's a good idea to minimize the work you do in constructor functions. The fewer things you do, the less chance there is for the constructor to fail, and constructor failures are messy. Functions such as `CDocument::OnNewDocument` and `CView::OnInitialUpdate` are excellent places to do initial housekeeping. If anything fails at creation time, you can pop up a message box and in the case of `OnNewDocument()`, you can return `FALSE`. Be advised that both functions can be called more than once for the same object. If you need certain instructions executed only once, declare a "first time" flag data member and then test/set it appropriately.

Connecting File Open to Your Serialization Code: The `OnFileOpen()` Function

When AppWizard generates an application, it maps the File Open menu item to the `CWinApp::OnFileOpen` member function. When called, this function invokes a sequence of functions to accomplish these steps:

1. Prompts the user to select a file.
2. Calls the virtual function `CDocument::OnOpenDocument` for the already existing document object. This function opens the file, calls `CDocument::DeleteContents`, and constructs a `CArchive` object set for loading. It then calls the document's `Serialize()` function, which loads data from the archive.
3. Calls the view's `OnInitialUpdate()` function.

The **Most Recently Used** (MRU) file list is a handy alternative to the File Open menu item. The application framework tracks the four (default) most recently used files and display their names on the File menu. These filenames are stored in the **Windows Registry** between program executions. You can change the number of recent files tracked by supplying a parameter to the `LoadStdProfileSetting()` function in the application class `InitInstance()` function.

The Document Class's `DeleteContents()` Function

When you load an existing SDI document object from a disk file, you must somehow erase the existing contents of the document object. The best way to do this is to override the CDocument::DeleteContents virtual function in your derived document class. The overridden function, as you've seen in Module 10, does whatever is necessary to clean up your document class's data members. In response to both the File New and File Open menu items, the `CDocument` functions `OnNewDocument()` and `OnOpenDocument()` both call the `DeleteContents()` function, which means `DeleteContents()` is called immediately after the document object is first constructed. It's called again when you close a document. If you want your document classes to work in SDI applications, plan on emptying the document's contents in the `DeleteContents()` member function rather than in the destructor. Use the destructor only to clean up items that last for the life of the object.

Connecting File Save and File Save As to Your Serialization Code

When AppWizard generates an application, it maps the File Save menu item to the `OnFileSave()` member function of the `CDocument` class. `OnFileSave()` calls the `CDocument` function `OnSaveDocument()`, which in turn calls your document's `Serialize()` function with an archive object set for storing. The **File Save As** menu item is handled in a similar manner: it is mapped to the `CDocument` function `OnFileSaveAs()`, which calls `OnSaveDocument()`. Here the application framework does all the file management necessary to save a document on disk. Yes, it is true that the File New and File Open menu options are mapped to application class member functions, but File Save and File Save As are mapped to document class member functions. File New is mapped to `OnFileNew()`. The SDI version of `InitInstance()` also calls `OnFileNew()` (indirectly). No document object exists when the application framework calls `InitInstance()`, so `OnFileNew()` can't possibly be a member function of `CDocument`. When a document is saved, however, a document object certainly exists.

The Document's "Dirty" Flag

Many document-oriented applications for Windows track the user's modifications of a document. If the user tries to close a document or exit the program, a message box asks whether the user wants to save the document. The MFC application framework directly supports this behavior with the `CDocument` data member `m_bModified`. This Boolean variable is `TRUE` if the document has been modified (has become "dirty"); otherwise, it is `FALSE`.
The protected `m_bModified` flag is accessed through the `CDocument` member functions `SetModifiedFlag()` and `IsModified()`. The framework sets the document object's flag to `FALSE` when the document is created or read from disk and when it is saved on disk. You, the programmer, must use the `SetModifiedFlag()` function to set the flag to `TRUE` when the document data changes. The virtual function `CDocument::SaveModified`, which the framework calls when the user closes the document, displays a message box if the `m_bModified` flag is set to `TRUE`. You can override this function if you need to do something else.
In the MYMFC17 example, you'll see how a one-line update command UI function can use `IsModified()` to control the state of the disk button and the corresponding menu item. When the user modifies the file, the disk button is enabled; when the user saves the file, the button changes to gray. In one respect, MFC SDI applications behave a little differently from other Windows SDI applications such as Notepad. Here's a typical sequence of events:

1. The user creates a document and saves it on disk under the name (for example) **test.dat**.
2. The user modifies the document.
3. The user chooses **File Open** and then specifies **test.dat**.

When the user chooses **File Open**, **Notepad** asks whether the user wants to save the changes made to the document (in Step 2 above). If the user answers no, the program rereads the document from disk. An MFC application, on the other hand, assumes that the changes are permanent and does not reread the file.

The MYMFC17 Example: SDI with Serialization

The MYMFC17 example is similar to example MYMFC16. The student dialog and the toolbar are the same except the **step 4** where we set the **Advanced Options** (shown below) and the view class is the same. The steps have been simplified in the following Figures.

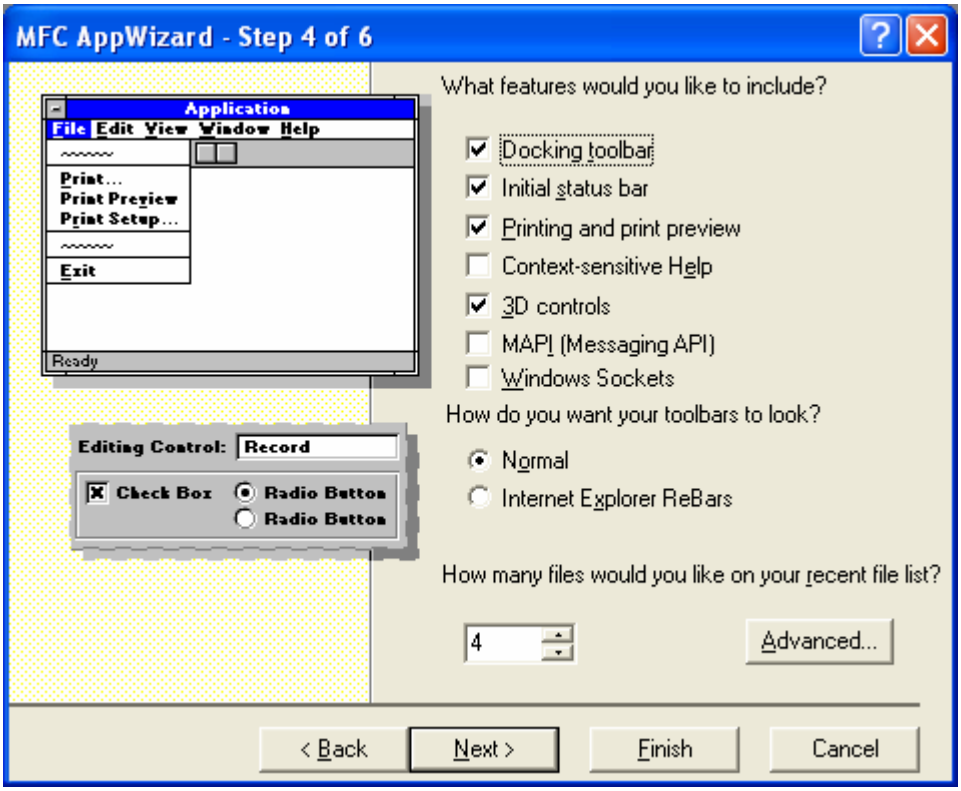

Figure 7: MYMFC17 AppWizard step 4 of 6, setting the **Advanced** options.

Click the **Advanced** button, fill the **File extension** as shown and for other fields will be provided automatically. Take note that name length will be truncated. You can change to other name but for this example, just accept the default.

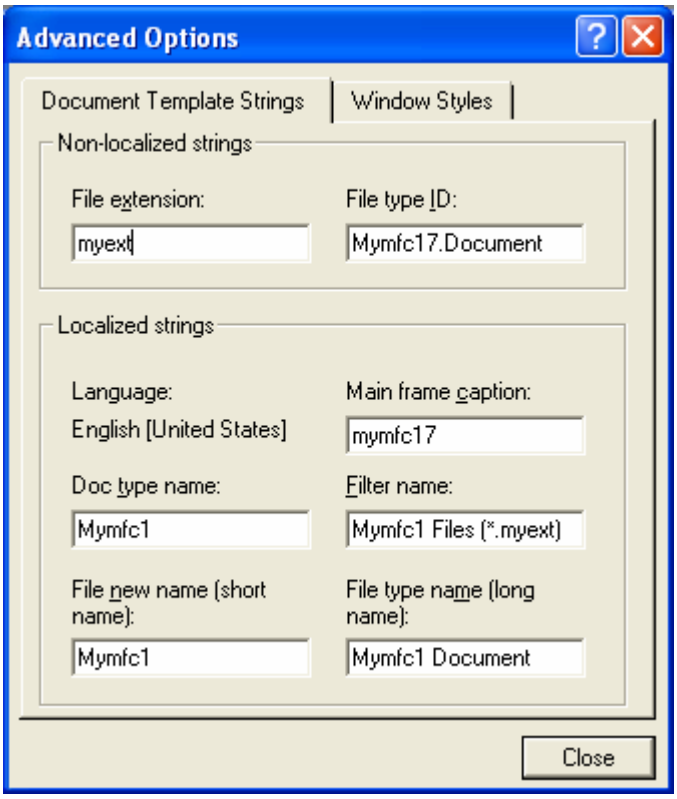

Figure 8: The file extension used is **myext**.

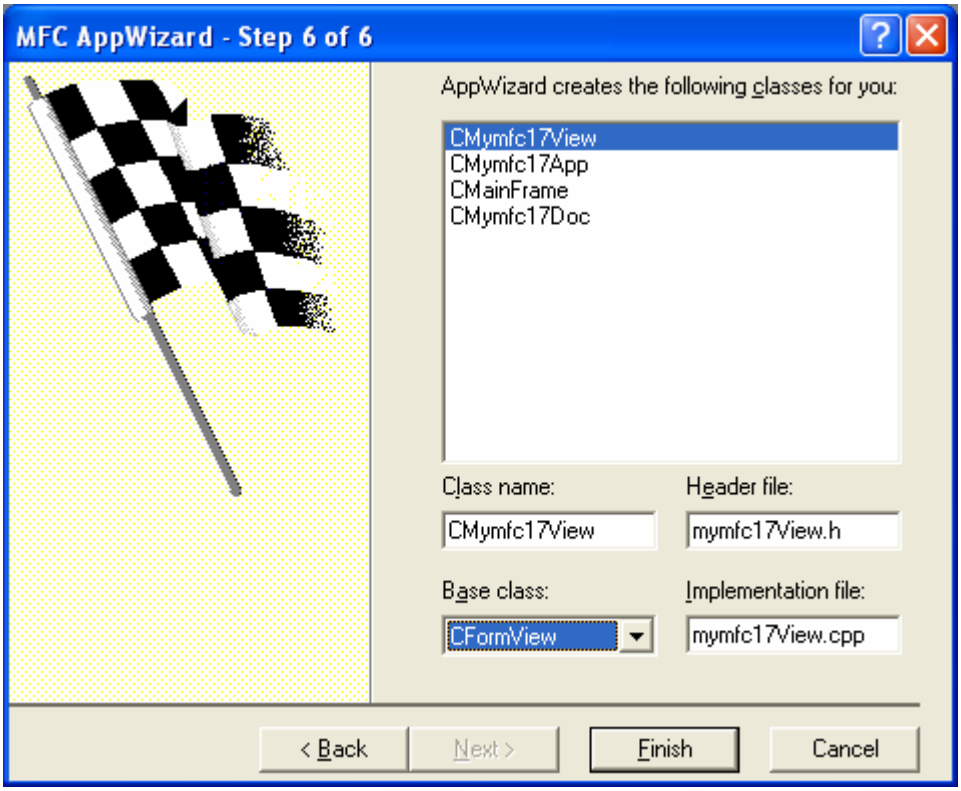

Figure 9: AppWizard step 6 of 6 for MYMFC17 project, using a `CFormView` class.

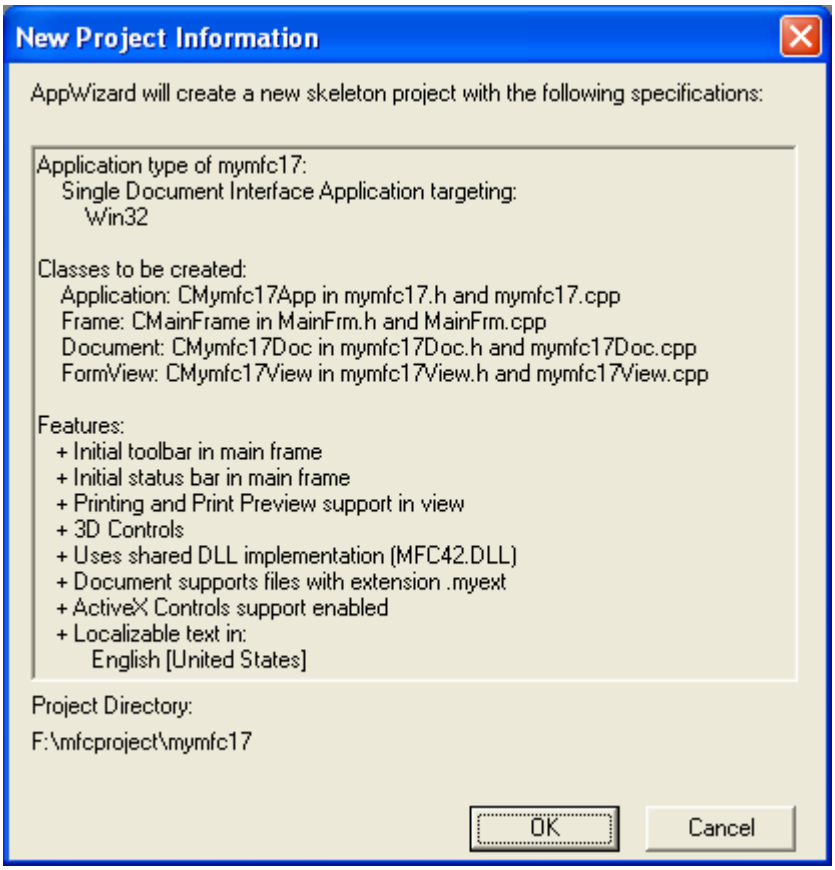

Figure 10: MYMFC17 project summary.

Control	ID
The dialog template	IDD_MYMFC17_FORM
Name edit control	IDC_NAME
Grade edit control	IDC_GRADE
Clear pushbutton	IDC_CLEAR

Table 2.

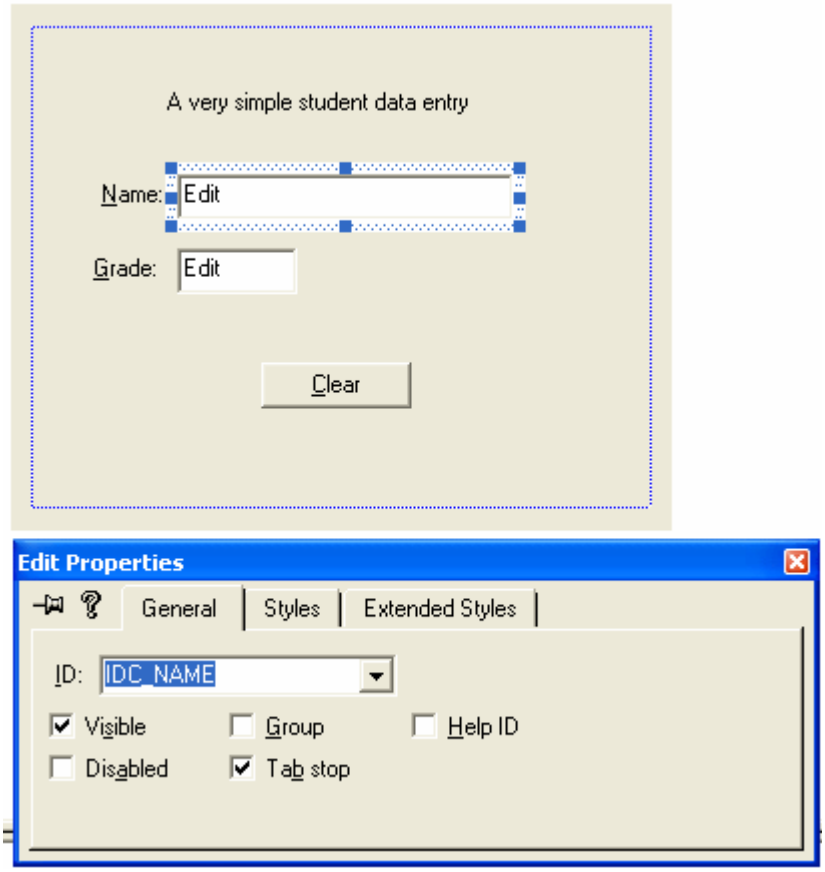

Figure 11: MYMFC17 dialog and its controls, similar to MYMFC16.

Object ID	Message	Member Function
ID_EDIT_CLEAR_ALL	COMMAND	OnEditClearAll()
ID_EDIT_CLEAR_ALL	ON_UPDATE_COMMAND_UI	OnUpdateEditClearAll()

Table 4.

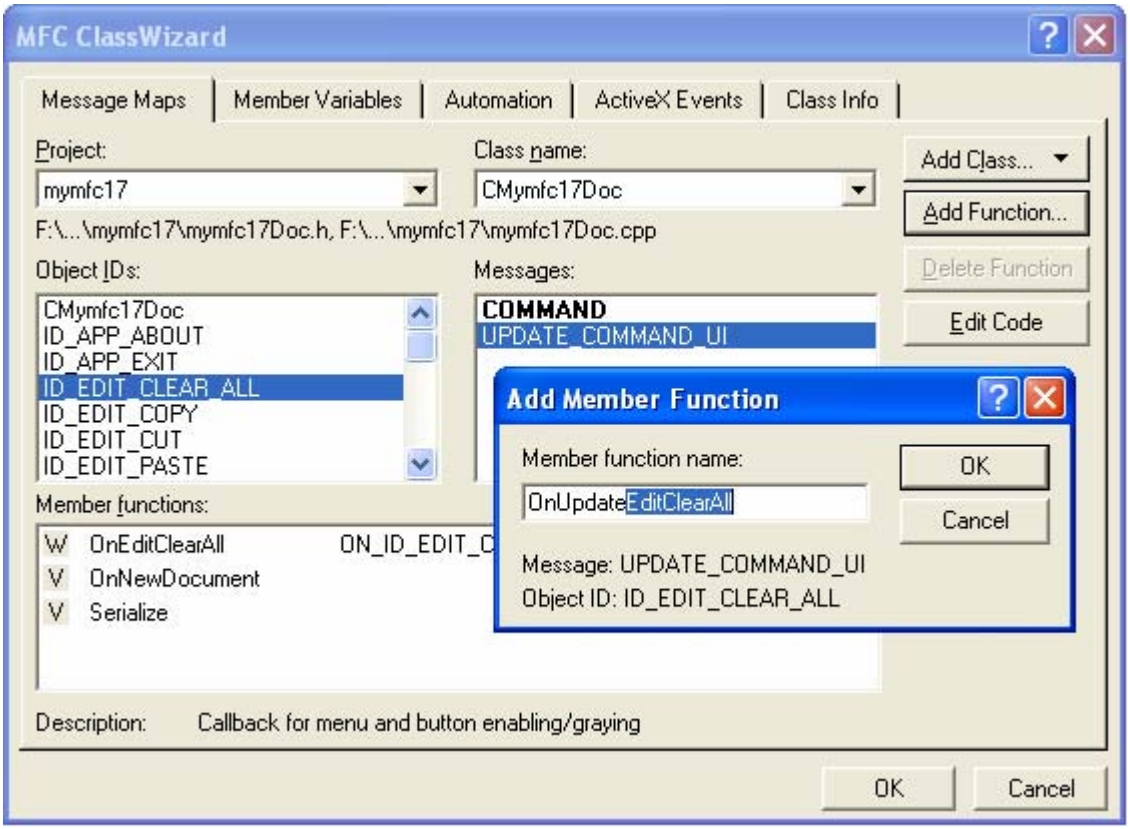

Figure 12: Message mapping for `IDC_EDIT_CLEAR_ALL`.

Object ID	Message	Member Function
`IDC_CLEAR`	`BN_CLICKED`	`OnClear()`

Table 5.

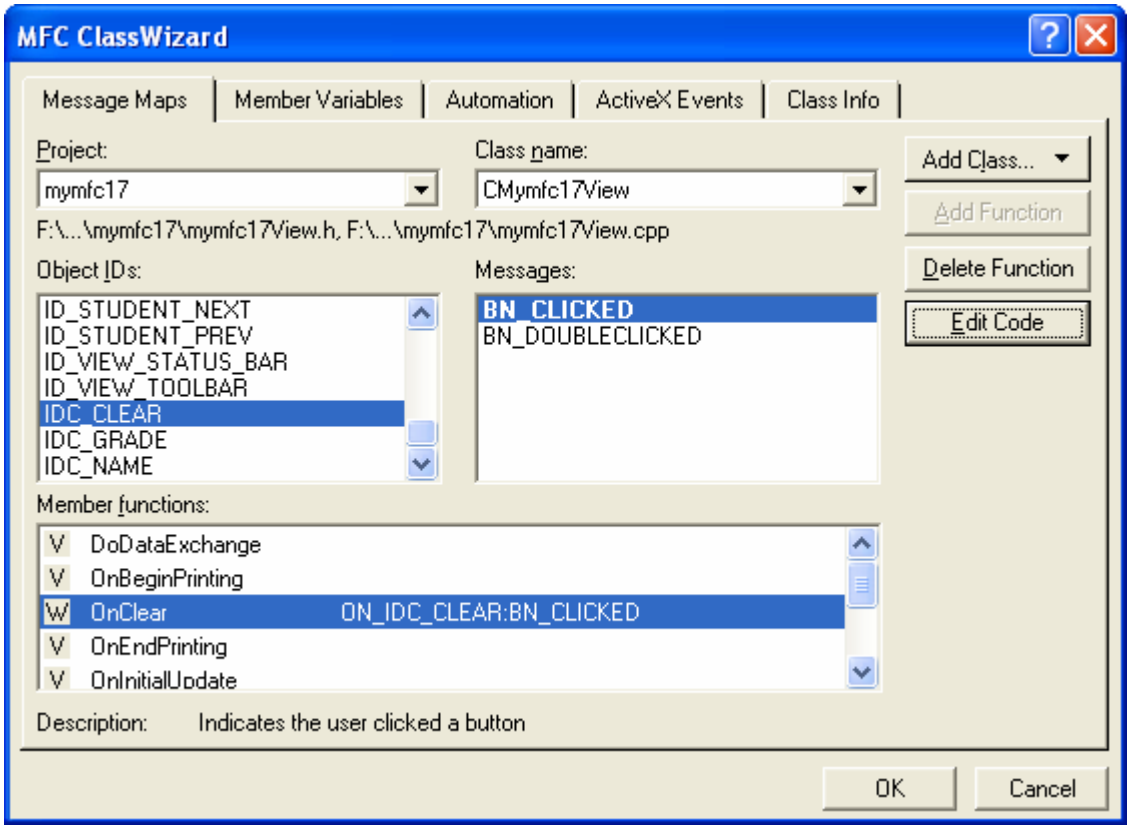

Figure 13: Message mapping for `IDC_ CLEAR`.

Control ID	Member Variable	Category	Variable Type
`IDC_GRADE`	`m_nGrade`	`Value`	`int`
`IDC_NAME`	`m_strName`	`Value`	`CString`

Table 6.

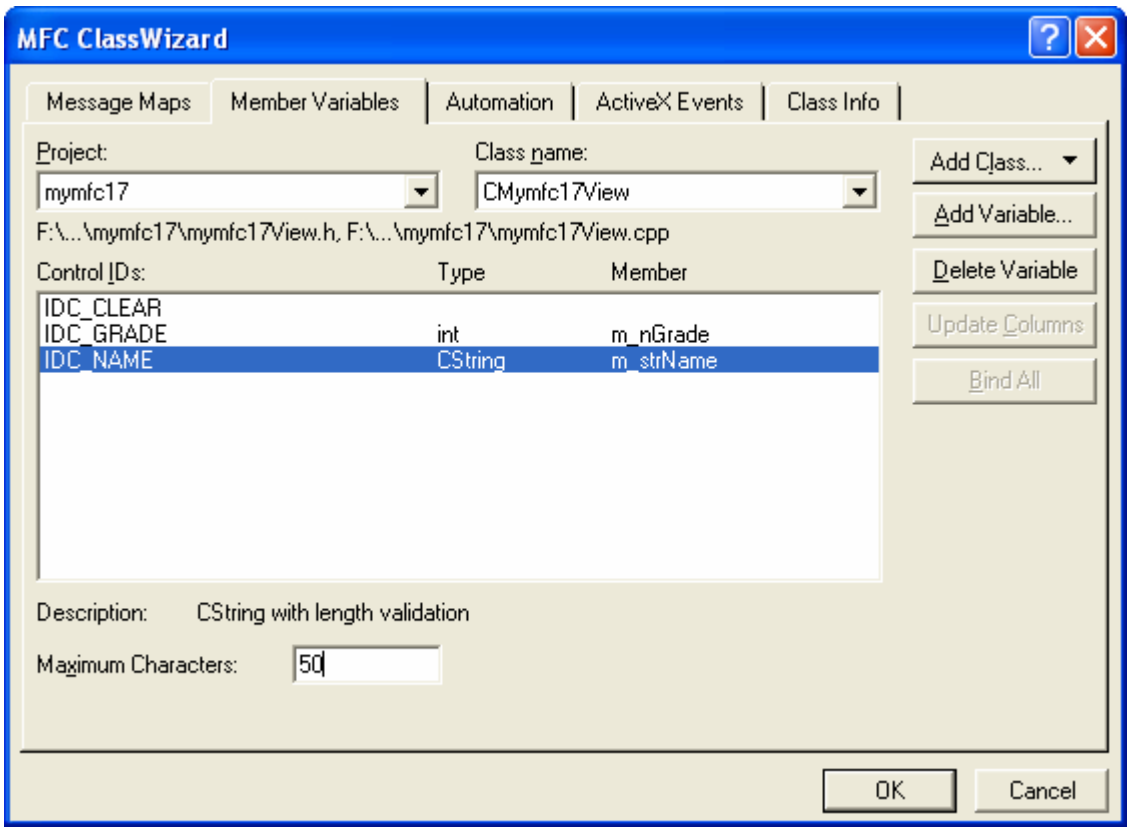

Figure 14: Adding member variables.

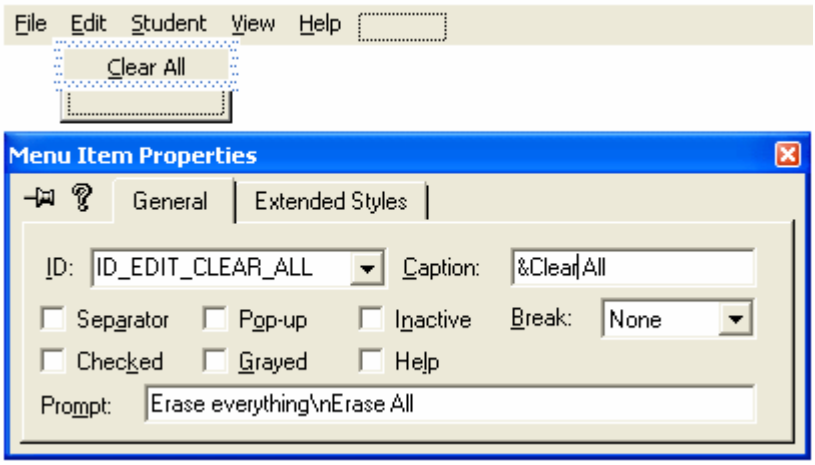

Figure 15: Adding and modifying **Clear All** menu properties.

	Object ID	Message	Member Function
	ID_STUDENT_HOME	COMMAND	OnStudentHome()
	ID_STUDENT_END	COMMAND	OnStudentEnd()
	ID_STUDENT_PREV	COMMAND	OnStudentPrev()
	ID_STUDENT_NEXT	COMMAND	OnStudentNext()
	ID_STUDENT_INS	COMMAND	OnStudentIns()
	ID_STUDENT_DEL	COMMAND	OnStudentDel()

Table 7.

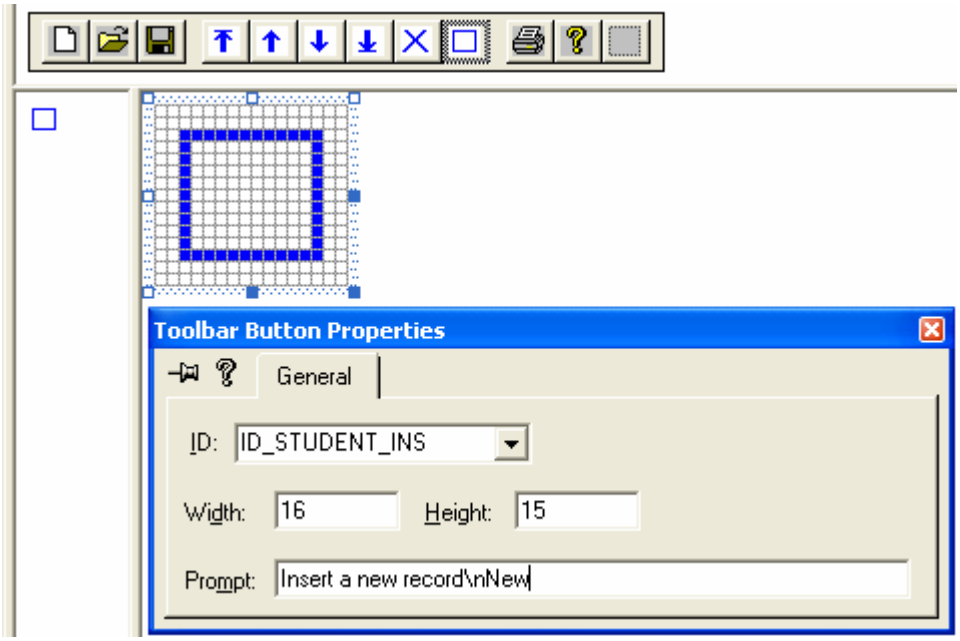

Figure 16: Adding and modifying toolbar buttons properties.

Object ID	Message	Member Function
ID_STUDENT_HOME	UPDATE_COMMAND_UI	OnUpdateStudentHome()
ID_STUDENT_END	UPDATE_COMMAND_UI	OnUpdateStudentEnd()
ID_STUDENT_PREV	UPDATE_COMMAND_UI	OnUpdateStudentHome()
ID_STUDENT_NEXT	UPDATE_COMMAND_UI	OnUpdateStudentEnd()
ID_STUDENT_DEL	UPDATE_COMMAND_UI	OnUpdateCommandDel()

Table 8.

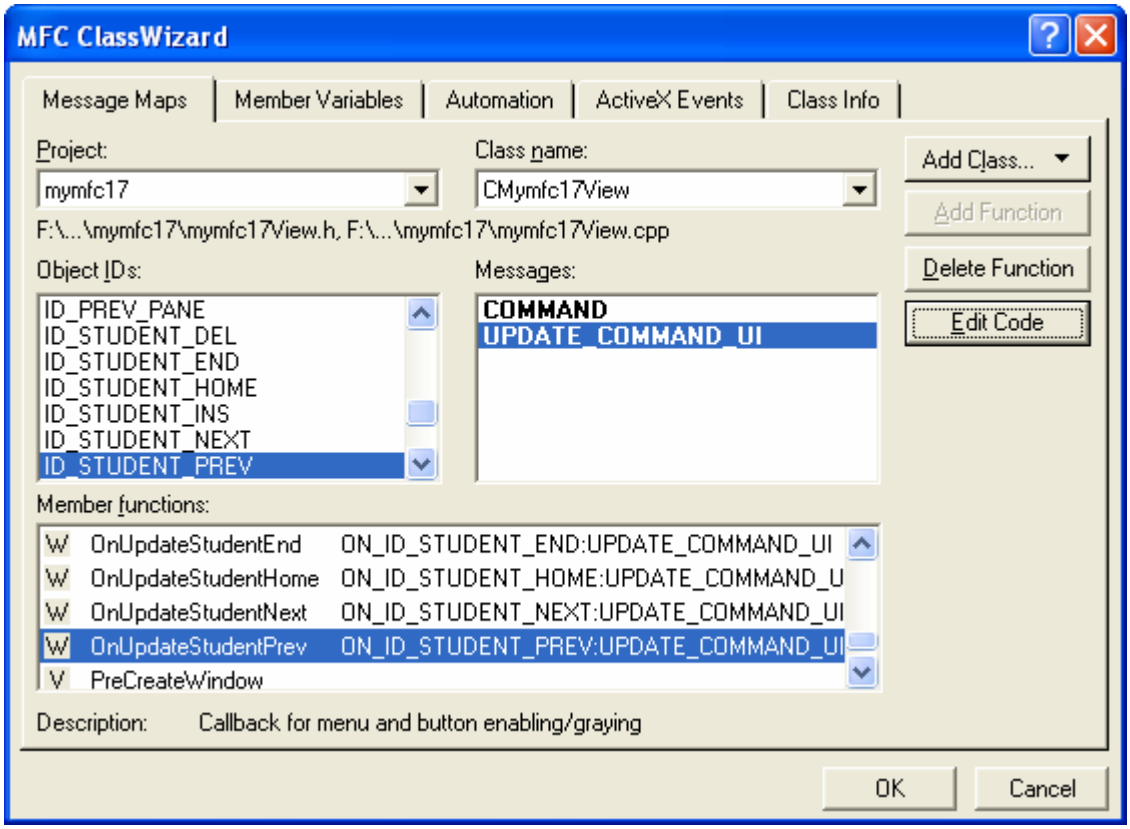

Figure 17: Messages mapping for toolbar buttons.

Serialization has been added, together with an update command UI function for File Save. The header and implementation files for the view and document classes will be reused in example MYMFC18 in the next module. All the new code (code that is different from MYMFC16) is listed, with additions and changes to the AppWizard-generated code and the ClassWizard code in orange if any. A list of the files and classes in the MYMFC17 example is shown in the following table.

Header File	Source Code File	Class	Description
mymfc17.h	mymfc17.cpp	`CMymfc17App`	Application class (from AppWizard)
		`CAboutDlg`	About dialog
MainFrm.h	MainFrm.cpp	`CMainFrame`	SDI main frame
mymfc17Doc.h	mymfc17Doc.cpp	`CMymfc17Doc`	Student document
mymfc17View.h	mymfc17View.cpp	`CMymfc17View`	Student form view (from MYMFC17)
Student.h	Student.cpp	`CStudent`	Student record
StdAfx.h	StdAfx.cpp		Precompiled headers (with **afxtempl.h** included)

Table 9.

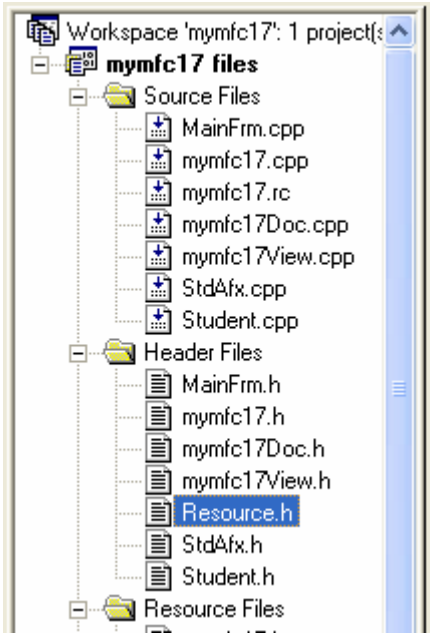

Figure 18: MYMFC17 files seen through FileView.

`CStudent` Class

The following steps show how to add the `CStudent` class (**Student.h** and **Student.cpp**).

Figure 19: Creating and adding new files for `CStudent` class to the project.

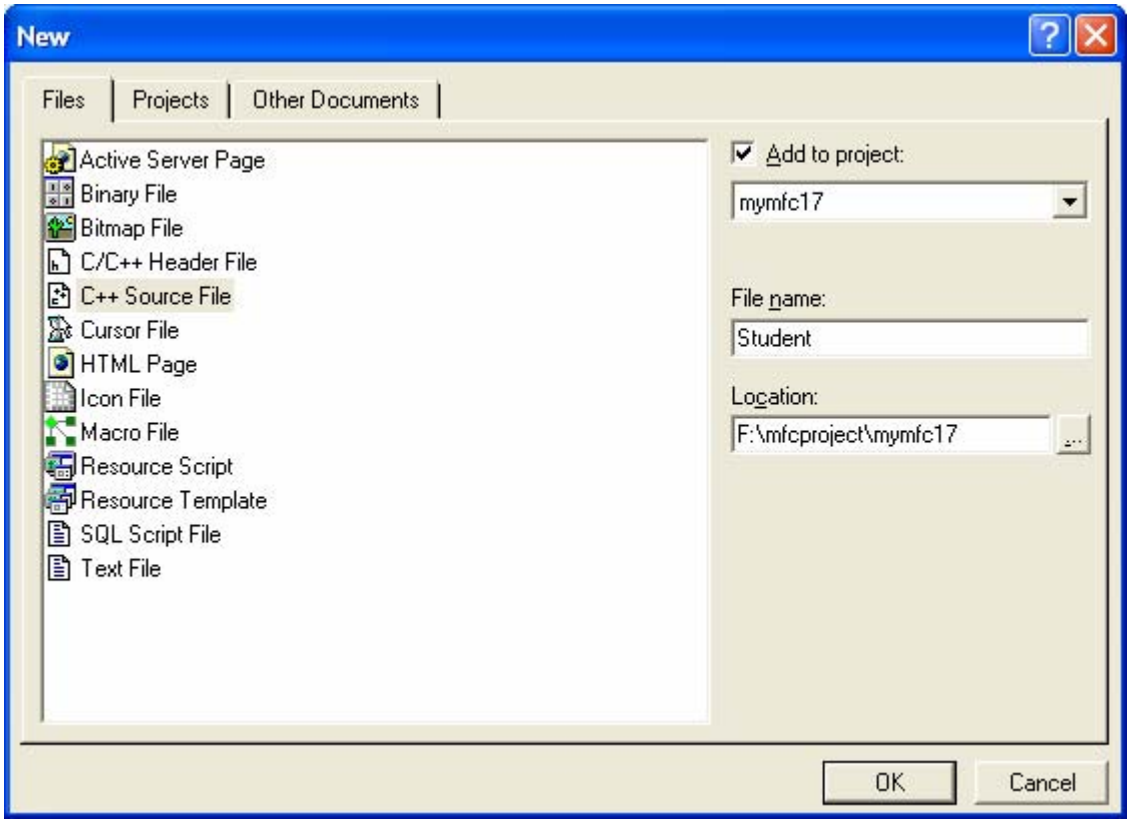

Figure 20: Creating and adding **Student.cpp** file to the project.

STUDENT.H

```
// student.h

#ifndef _INSIDE_VISUAL_CPP_STUDENT
#define _INSIDE_VISUAL_CPP_STUDENT

class CStudent : public CObject
{
    DECLARE_SERIAL(CStudent)
public:
    CString m_strName;
    int m_nGrade;

    CStudent()
    {
        m_nGrade = 0;
    }

    CStudent(const char* szName, int nGrade) : m_strName(szName)
    {
        m_nGrade = nGrade;
    }

    CStudent(const CStudent& s) : m_strName(s.m_strName)
    {
        // copy constructor
        m_nGrade = s.m_nGrade;
    }

    const CStudent& operator =(const CStudent& s)
    {
        m_strName = s.m_strName;
        m_nGrade = s.m_nGrade;
```

```
        return *this;
    }

    BOOL operator ==(const CStudent& s) const
    {
        if ((m_strName == s.m_strName) && (m_nGrade == s.m_nGrade))
              {
            return TRUE;
        }
        else
              {
            return FALSE;
        }
    }

    BOOL operator !=(const CStudent& s) const
    {
        // Let's make use of the operator we just defined!
        return !(*this == s);
    }
#ifdef _DEBUG
    void Dump(CDumpContext& dc) const;
#endif // _DEBUG
};

#endif // _INSIDE_VISUAL_CPP_STUDENT

typedef CTypedPtrList<CObList, CStudent*> CStudentList;
```

STUDENT.CPP

```
#include "stdafx.h"
#include "student.h"

IMPLEMENT_SERIAL(CStudent, CObject, 0)

#ifdef _DEBUG
void CStudent::Dump(CDumpContext& dc) const
{
    CObject::Dump(dc);
    dc << "m_strName = " << m_strName << "\nm_nGrade = " <<m_nGrade;
}
#endif // _DEBUG
```

Listing 1: `CStudent` class.

The use of the MFC template collection classes requires the following statement in **StdAfx.h**:

```
#include <afxtempl.h>
```

The MYMFC17 **Student.h** file is almost the same as the file in the MYMFC17 project except the header contains the macro:

```
DECLARE_SERIAL(CStudent)
```

instead of:

```
DECLARE_DYNAMIC(CStudent)
```

```
// student.h

#ifndef _INSIDE_VISUAL_CPP_STUDENT
#define _INSIDE_VISUAL_CPP_STUDENT

class CStudent : public CObject
{
    DECLARE_SERIAL(CStudent)
public:
    CString m_strName;
    int m_nGrade;
```

Listing 2.

and the implementation file contains the macro:

```
IMPLEMENT_SERIAL(CStudent, CObject, 0)
```

instead of:

```
IMPLEMENT_DYNAMIC(CStudent, Cobject)
```

```
#include "stdafx.h"
#include "student.h"

IMPLEMENT_SERIAL(CStudent, CObject, 0)
```

Listing 3.

The virtual `Serialize()` function has also been added.

`CMymfc17App` Class

The application class files, shown in Listing 4, contain only code generated by AppWizard. The application was generated with a default file extension and with the Microsoft Windows Explorer launch and drag-and-drop capabilities. These features are described later in this module. To generate additional code, you must do the following when you first run AppWizard (already shown in the previous steps): in the AppWizard Step 4 page, click the **Advanced** button. When the **Advanced Options** dialog appears, you must enter the **Filename extension** in the upper-left control, as shown here.

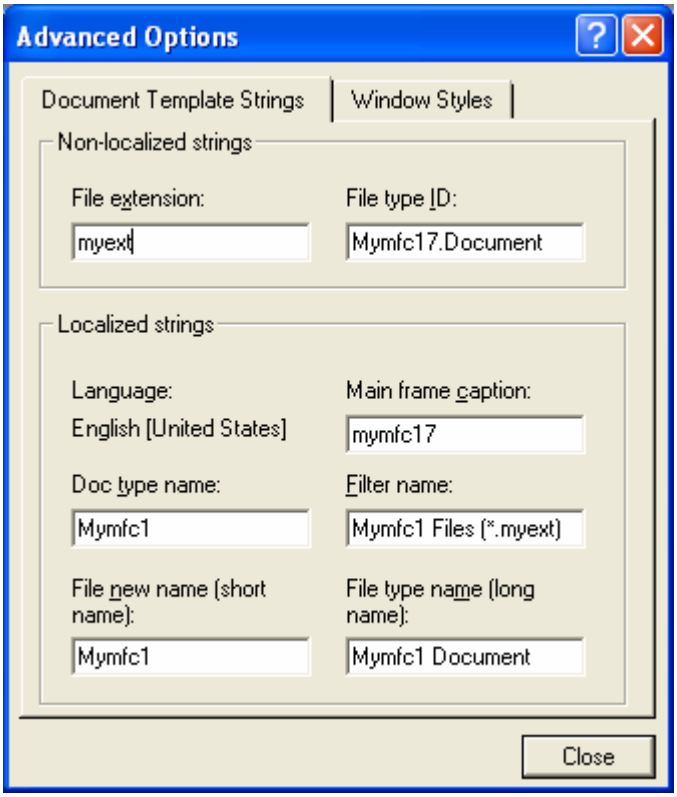

Figure 21: Setting the **File extension** for MYMFC17 project, as done in step 4 of 6 AppWizard.

This ensures that the document template resource string contains the **correct default extension** and that the correct Explorer-related code is inserted into your application class `InitInstance()` member function. You can change some of the other resource substrings if you want. The generated calls to `Enable3dControls()` and `Enable3dControlsStatic()` in `CMymfc17App::InitInstance` are not necessary with Microsoft Windows 95, Microsoft Windows 98, or Microsoft Windows NT 4.0. These two functions support an older DLL that is shipped with Microsoft Windows 3.51.

```
MYMFC17.H
// mymfc17.h : main header file for the MYMFC17 application
//

#if
!defined(AFX_MYMFC17_H__1A036EA3_821A_11D0_8FE2_00C04FC2A0C2__INCLUDED_)
#define AFX_MYMFC17_H__1A036EA3_821A_11D0_8FE2_00C04FC2A0C2__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
#ifndef __AFXWIN_H__
    #error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"       // main symbols

/////////////////////////////////////////////////////////////////////////////
// CMymfc17App:
// See mymfc17.cpp for the implementation of this class
//

class CMymfc17App : public CWinApp
{
public:
```

```
    CMymfc17App();

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CMymfc17App)
    public:
    virtual BOOL InitInstance();
    //}}AFX_VIRTUAL

// Implementation

    //{{AFX_MSG(CMymfc17App)
    afx_msg void OnAppAbout();
        // NOTE - the ClassWizard will add and remove member functions
here.
        //    DO NOT EDIT what you see in these blocks of generated
code!
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////
//

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations
// immediately before the previous line.

#endif //
!defined(AFX_MYMFC17_H__1A036EA3_821A_11D0_8FE2_00C04FC2A0C2__INCLUDED_)
```

MYMFC17.CPP

```
// mymfc17.cpp : Defines the class behaviors for the application.
//

#include "stdafx.h"
#include "mymfc17.h"

#include "MainFrm.h"
#include "mymfc17Doc.h"
#include "mymfc17View.h"

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

/////////////////////////////////////////////////////////////////////////
// CMymfc17App

BEGIN_MESSAGE_MAP(CMymfc17App, CWinApp)
    //{{AFX_MSG_MAP(CMymfc17App)

    ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
        // NOTE - the ClassWizard will add and remove mapping macros
here.
        //    DO NOT EDIT what you see in these blocks of generated
code!
    //}}AFX_MSG_MAP
    // Standard file based document commands
    ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
    ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
END_MESSAGE_MAP()
```

```
/////////////////////////////////////////////////////////////////////////////
// CMymfc17App construction

CMymfc17App::CMymfc17App()
{
    // TODO: add construction code here,
    // Place all significant initialization in InitInstance
}

/////////////////////////////////////////////////////////////////////////////
// The one and only CMymfc17App object

CMymfc17App theApp;
/////////////////////////////////////////////////////////////////////////////
// CMymfc17App initialization

BOOL CMymfc17App::InitInstance()
{
    AfxEnableControlContainer();

    // Standard initialization
    // If you are not using these features and wish to reduce the size
    //  of your final executable, you should remove from the following
    // the specific initialization routines you do not need.

#ifdef _AFXDLL
    Enable3dControls();       // Call this when using MFC in a shared
DLL
#else
    Enable3dControlsStatic(); // Call this when linking to MFC
statically
#endif

    // Change the registry key under which our settings are stored.
    // You should modify this string to be something appropriate
    // such as the name of your company or organization.
    SetRegistryKey(_T("Local AppWizard-Generated Applications"));

    LoadStdProfileSettings();  // Load standard INI file options
                               //  (including MRU)

    // Register the application's document templates.
    // Document templates serve as the connection between
    // documents, frame windows and views.

    CSingleDocTemplate* pDocTemplate;
    pDocTemplate = new CSingleDocTemplate(
        IDR_MAINFRAME,
        RUNTIME_CLASS(CMymfc17Doc),
        RUNTIME_CLASS(CMainFrame),       // main SDI frame window
        RUNTIME_CLASS(CMymfc17View));
    AddDocTemplate(pDocTemplate);

    // Enable DDE Execute open
    EnableShellOpen();
    RegisterShellFileTypes(TRUE);
    // Parse command line for standard shell commands, DDE, file open
    CCommandLineInfo cmdInfo;
    ParseCommandLine(cmdInfo);

    // Dispatch commands specified on the command line
    if (!ProcessShellCommand(cmdInfo))
        return FALSE;

    // The one and only window has been initialized,
    // so show and update it.
```

```
    m_pMainWnd->ShowWindow(SW_SHOW);
    m_pMainWnd->UpdateWindow();

    // Enable drag/drop open
    m_pMainWnd->DragAcceptFiles();

    return TRUE;
}

/////////////////////////////////////////////////////////////////////////////
// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
    CAboutDlg();

// Dialog Data
    //{{AFX_DATA(CAboutDlg)

    enum { IDD = IDD_ABOUTBOX };
    //}}AFX_DATA

    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CAboutDlg)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:
    //{{AFX_MSG(CAboutDlg)
        // No message handlers
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
    //{{AFX_DATA_INIT(CAboutDlg)
    //}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CAboutDlg)
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
    //{{AFX_MSG_MAP(CAboutDlg)
        // No message handlers
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

// App command to run the dialog
void CMymfc17App::OnAppAbout()
{
    CAboutDlg aboutDlg;
    aboutDlg.DoModal();
}

/////////////////////////////////////////////////////////////////////////////
// CMymfc17App commands
```

Listing 4: The `CMymfc17App` class listing.

`CMainFrame` Class

The main frame window class code, shown in Listing 5, is almost unchanged from the code that AppWizard generated. The overridden `ActivateFrame()` function and the `WM_DROPFILES` handler exist solely for trace purposes.

```
MAINFRM.H
// MainFrm.h : interface of the CMainFrame class
//
/////////////////////////////////////////////////////////////////////////////

#if
!defined(AFX_MAINFRM_H__1A036EA7_821A_11D0_8FE2_00C04FC2A0C2__INCLUDED_)
#define AFX_MAINFRM_H__1A036EA7_821A_11D0_8FE2_00C04FC2A0C2__INCLUDED_
#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

class CMainFrame : public CFrameWnd
{
protected: // create from serialization only
    CMainFrame();
    DECLARE_DYNCREATE(CMainFrame)

// Attributes
public:

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CMainFrame)
    public:
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    virtual void ActivateFrame(int nCmdShow = -1);
    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CMainFrame();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:  // control bar embedded members
    CStatusBar  m_wndStatusBar;
    CToolBar    m_wndToolBar;

// Generated message map functions
protected:
    //{{AFX_MSG(CMainFrame)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnDropFiles(HDROP hDropInfo);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations
// immediately before the previous line.
```

```
#endif //
!defined(AFX_MAINFRM_H__1A036EA7_821A_11D0_8FE2_00C04FC2A0C2__INCLUDED_)


MAINFRM.CPP
// MainFrm.cpp : implementation of the CMainFrame class
//

#include "stdafx.h"
#include "mymfc17.h"

#include "MainFrm.h"

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

/////////////////////////////////////////////////////////////////////////////
// CMainFrame

IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
    //{{AFX_MSG_MAP(CMainFrame)
    ON_WM_CREATE()
    ON_WM_DROPFILES()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

static UINT indicators[] =
{
    ID_SEPARATOR,           // status line indicator
    ID_INDICATOR_CAPS,
    ID_INDICATOR_NUM,
    ID_INDICATOR_SCRL,
};
/////////////////////////////////////////////////////////////////////////////
// CMainFrame construction/destruction

CMainFrame::CMainFrame()
{
    // TODO: add member initialization code here

}

CMainFrame::~CMainFrame()
{
}

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;

    if (!m_wndToolBar.Create(this) ||
!m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
    {
        TRACE0("Failed to create toolbar\n");
        return -1;      // fail to create
    }

    if (!m_wndStatusBar.Create(this) ||
        !m_wndStatusBar.SetIndicators(indicators,
          sizeof(indicators)/sizeof(UINT)))
```

```
    {
        TRACE0("Failed to create status bar\n");
        return -1;      // fail to create
    }

    // TODO: Remove this if you don't want tool tips
    //  or a resizable toolbar
    m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |
        CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);

    // TODO: Delete these three lines if you don't want the toolbar to
    //  be dockable
    m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
    EnableDocking(CBRS_ALIGN_ANY);

DockControlBar(&m_wndToolBar);
    return 0;
}

BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    //  the CREATESTRUCT cs

    return CFrameWnd::PreCreateWindow(cs);
}

/////////////////////////////////////////////////////////////////////////////
// CMainFrame diagnostics

#ifdef _DEBUG
void CMainFrame::AssertValid() const
{
    CFrameWnd::AssertValid();
}

void CMainFrame::Dump(CDumpContext& dc) const
{
    CFrameWnd::Dump(dc);
}

#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CMainFrame message handlers
void CMainFrame::ActivateFrame(int nCmdShow)
{
    TRACE("Entering CMainFrame::ActivateFrame\n");
    CFrameWnd::ActivateFrame(nCmdShow);
}

void CMainFrame::OnDropFiles(HDROP hDropInfo)
{
    TRACE("Entering CMainFrame::OnDropFiles\n");
    CFrameWnd::OnDropFiles(hDropInfo);
}
```

Listing 5: The `CMainFrame` class listing.

`CMymfc17Doc` Class

AppWizard originally generated the `CMymfc17Doc` class. Figure 6 shows the code used in the MYMFC17 example. The `CMymfc17Doc` class is the same as the `CMymfc16Doc` class from the previous module except for four functions: `Serialize()`, `DeleteContents()`, `OnOpenDocument()`, and `OnUpdateFileSave()`.

MYMFC17DOC.H

```
//MYMFC17DOC.H
// Mymfc17Doc.h : interface of the CMymfc17Doc class
//
/////////////////////////////////////////////////////////////////////////////

#if
!defined(AFX_MYMFC17DOC_H__4D011047_7E1C_11D0_8FE0_00C04FC2A0C2__INCLUDED_)
#define AFX_MYMFC17DOC_H__4D011047_7E1C_11D0_8FE0_00C04FC2A0C2__INCLUDED_

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

#include "student.h"

class CMymfc17Doc : public CDocument
{
protected: // create from serialization only
    CMymfc17Doc();
    DECLARE_DYNCREATE(CMymfc17Doc)

// Attributes
public:
    CStudentList* GetList() {
        return &m_studentList;
    }

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CMymfc17Doc)
       public:
    virtual BOOL OnNewDocument();
    virtual void Serialize(CArchive& ar);
    virtual void DeleteContents();
       virtual BOOL OnOpenDocument(LPCTSTR lpszPathName);
       //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CMymfc17Doc();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
    //{{AFX_MSG(CMymfc17Doc)
    afx_msg void OnEditClearAll();
    afx_msg void OnUpdateEditClearAll(CCmdUI* pCmdUI);
       afx_msg void OnUpdateFileSave(CCmdUI* pCmdUI);
       //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
private:
    CStudentList m_studentList;
```

```
};

/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations
// immediately before the previous line.

#endif //
!defined(AFX_MYMFC16DOC_H__4D011047_7E1C_11D0_8FE0_00C04FC2A0C2__INCLUDED_)
```

MYMFC17DOC.CPP

```
// Mymfc17Doc.cpp : implementation of the CMymfc17Doc class
//

#include "stdafx.h"
#include "mymfc17.h"

#include "Mymfc17Doc.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CMymfc17Doc

IMPLEMENT_DYNCREATE(CMymfc17Doc, CDocument)

BEGIN_MESSAGE_MAP(CMymfc17Doc, CDocument)
    //{{AFX_MSG_MAP(CMymfc17Doc)
    ON_COMMAND(ID_EDIT_CLEAR_ALL, OnEditClearAll)
    ON_UPDATE_COMMAND_UI(ID_EDIT_CLEAR_ALL, OnUpdateEditClearAll)
       ON_UPDATE_COMMAND_UI(ID_FILE_SAVE, OnUpdateFileSave)
       //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CMymfc17Doc construction/destruction

CMymfc17Doc::CMymfc17Doc()
{
    TRACE("Entering CMymfc17Doc constructor\n");
#ifdef _DEBUG
    afxDump.SetDepth(1); // Ensure dump of list elements
#endif // _DEBUG
}

CMymfc17Doc::~CMymfc17Doc()
{
}

BOOL CMymfc17Doc::OnNewDocument()
{
    TRACE("Entering CMymfc17Doc::OnNewDocument\n");
    if (!CDocument::OnNewDocument())
        return FALSE;

    // TODO: add re-initialization code here
    // (SDI documents will reuse this document)

    return TRUE;
}
/////////////////////////////////////////////////////////////////////////////
// CMymfc17Doc serialization
```

```
void CMymfc17Doc::Serialize(CArchive& ar)
{
    TRACE("Entering CMymfc17Doc::Serialize\n");
    if (ar.IsStoring())
    {
        // TODO: add storing code here
    }
    else
    {
        // TODO: add loading code here
    }
    m_studentList.Serialize(ar);
}

/////////////////////////////////////////////////////////////////////////////
// CMymfc17Doc diagnostics

#ifdef _DEBUG
void CMymfc17Doc::AssertValid() const
{
    CDocument::AssertValid();
}

void CMymfc17Doc::Dump(CDumpContext& dc) const
{
    CDocument::Dump(dc);
    dc << "\n" << m_studentList << "\n";
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CMymfc17Doc commands

void CMymfc17Doc::DeleteContents()
{
    TRACE("Entering CMymfc17Doc::DeleteContents\n");
    while (m_studentList.GetHeadPosition())
       {
        delete m_studentList.RemoveHead();
    }
}

void CMymfc17Doc::OnEditClearAll()
{
    DeleteContents();
    UpdateAllViews(NULL);
}

void CMymfc17Doc::OnUpdateEditClearAll(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(!m_studentList.IsEmpty());
}

BOOL CMymfc17Doc::OnOpenDocument(LPCTSTR lpszPathName)
{
     TRACE("Entering CMymfc17Doc::OnOpenDocument\n");
     if (!CDocument::OnOpenDocument(lpszPathName))
        return FALSE;

    // TODO: Add your specialized creation code here

    return TRUE;
}

void CMymfc17Doc::OnUpdateFileSave(CCmdUI* pCmdUI)
```

```
{
        // TODO: Add your command update UI handler code here
        pCmdUI->Enable(IsModified());

}
```

Listing 6: The `CMymfc17Doc` class listing.

`Serialize()`

One line has been added to the AppWizard-generated function to serialize the document's student list, as shown here:

```
////////////////////////////////////////////////////////////////////
// CStudentDoc serialization

void CStudentDoc::Serialize(CArchive& ar)
{
    TRACE("Entering CStudentDoc::Serialize\n");
    if (ar.IsStoring())
    {
        // TODO: add storing code here
    }
    else
    {
        // TODO: add loading code here
    }
    m_studentList.Serialize(ar);
}
```

`DeleteContents()`

The `Dump` statement is replaced by a simple `TRACE` statement. Here is the modified code:

```
void CStudentDoc::DeleteContents()
{
    TRACE("Entering CStudentDoc::DeleteContents\n");
    while (m_studentList.GetHeadPosition())
    {
        delete m_studentList.RemoveHead();
    }
}
```

`OnOpenDocument()`

This virtual function is overridden only for the purpose of displaying a `TRACE` message, as shown below.

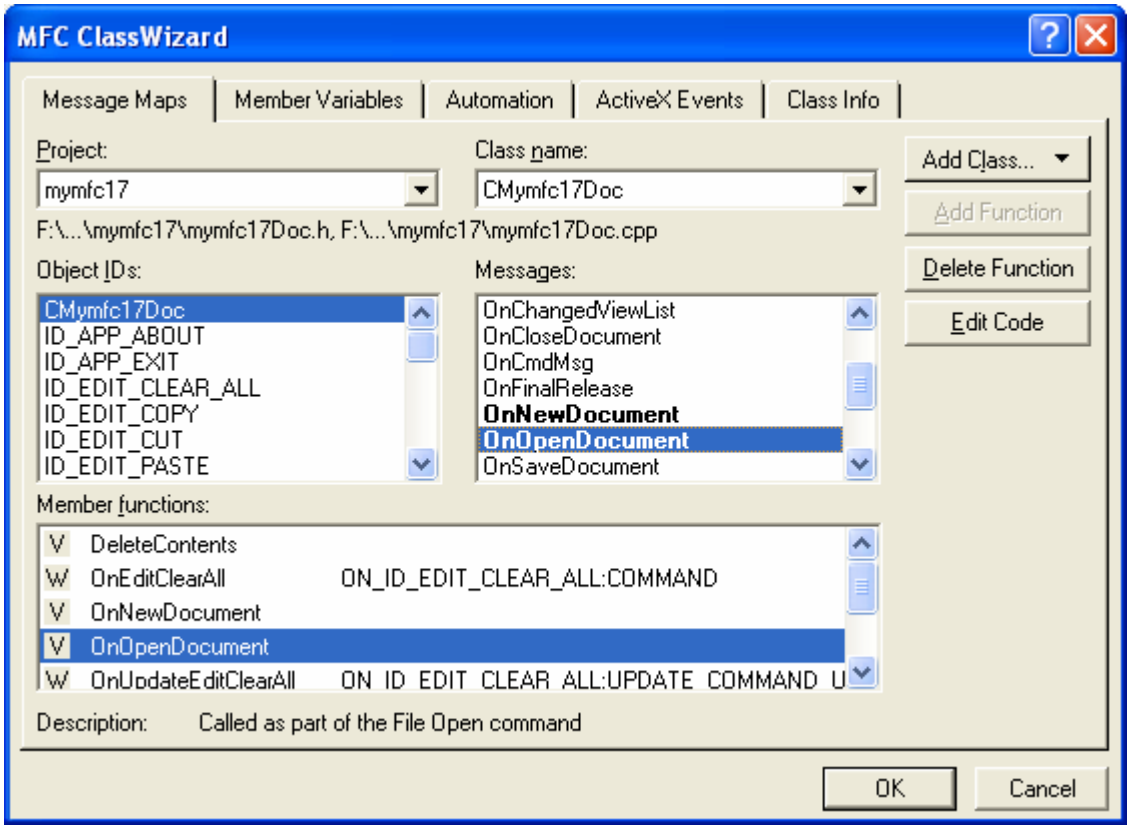

Figure 22: `OnOpenDocument()` message mapping for `CMymfc17Doc` class.

```
BOOL CStudentDoc::OnOpenDocument(LPCTSTR lpszPathName)
{
     TRACE("Entering CStudentDoc::OnOpenDocument\n");
     if (!CDocument::OnOpenDocument(lpszPathName))
        return FALSE;
    // TODO: Add your specialized creation code here
    return TRUE;
}
```

`OnUpdateFileSave()`

This message map function grays the **File Save** toolbar button when the document is in the unmodified state. The view controls this state by calling the document's `SetModifiedFlag()` function, as shown here:

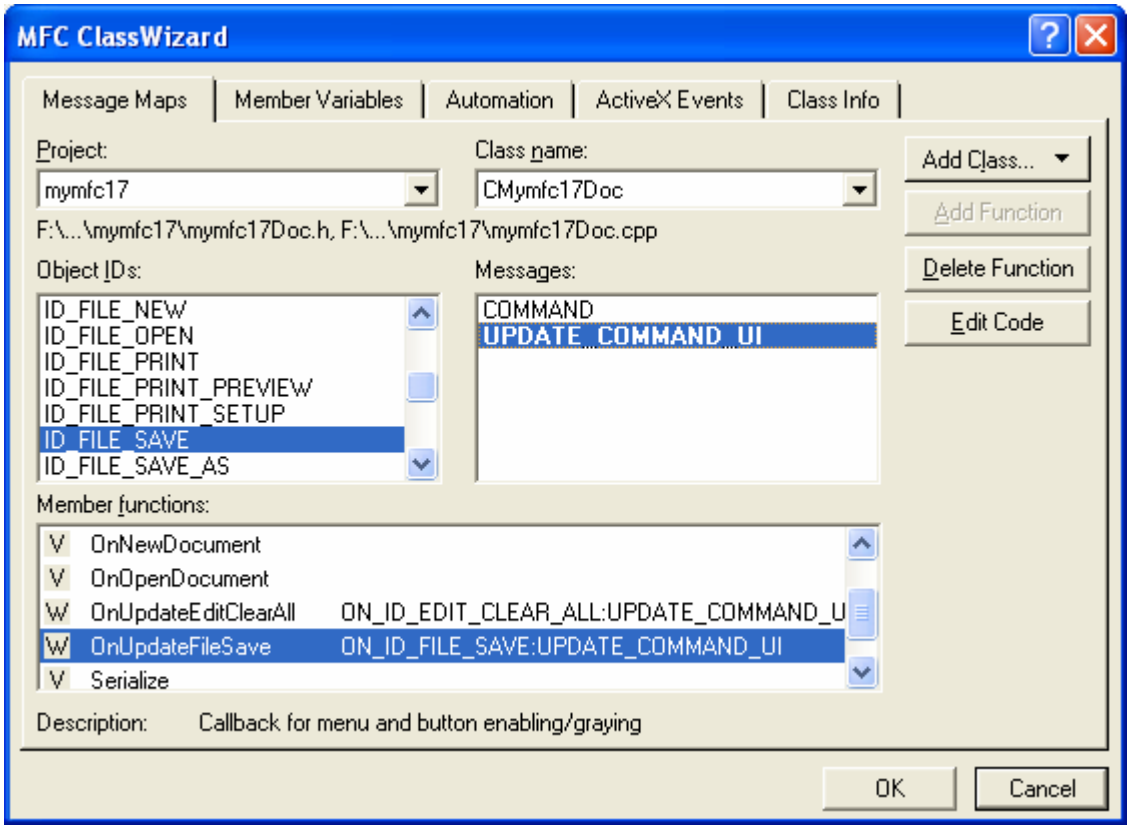

Figure 23: Adding message handler for `ID_FILE_SAVE`.

```
void CStudentDoc::OnUpdateFileSave(CCmdUI* pCmdUI)
{
    // Disable disk toolbar button if file is not modified
    pCmdUI->Enable(IsModified());
}
```

`CMymfc17View`

The code for the `CMymfc17View` class comes from the previous module. Listing 7 shows the code.

MYMFC17VIEW.H

```
// Mymfc17View.h : interface of the CMymfc17View class
//
/////////////////////////////////////////////////////////////////////////////

#if
!defined(AFX_MYMFC17VIEW_H__4D011049_7E1C_11D0_8FE0_00C04FC2A0C2__INCLUDED_)
#define AFX_MYMFC17VIEW_H__4D011049_7E1C_11D0_8FE0_00C04FC2A0C2__INCLUDED_

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

class CMymfc17View : public CFormView
{
protected:
    POSITION      m_position; // current position in document list
    CStudentList* m_pList;    // copied from document

protected: // create from serialization only
    CMymfc17View();
```

```
    DECLARE_DYNCREATE(CMymfc17View)

public:
    //{{AFX_DATA(CMymfc17View)
    enum { IDD = IDD_MYMFC17_FORM };
    int     m_nGrade;
    CString m_strName;
    //}}AFX_DATA

// Attributes
public:
    CMymfc17Doc* GetDocument();

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CMymfc17View)
    public:
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    protected:
    virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support
    virtual void OnInitialUpdate(); // called first time after construct
    virtual void OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint);
    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CMymfc17View();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:
    virtual void ClearEntry();
    virtual void InsertEntry(POSITION position);
    virtual void GetEntry(POSITION position);

// Generated message map functions
protected:
    //{{AFX_MSG(CMymfc17View)
    afx_msg void OnClear();
    afx_msg void OnStudentHome();
    afx_msg void OnStudentEnd();
    afx_msg void OnStudentPrev();
    afx_msg void OnStudentNext();
    afx_msg void OnStudentIns();
    afx_msg void OnStudentDel();
    afx_msg void OnUpdateStudentHome(CCmdUI* pCmdUI);
    afx_msg void OnUpdateStudentEnd(CCmdUI* pCmdUI);
    afx_msg void OnUpdateStudentDel(CCmdUI* pCmdUI);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG  // debug version in Mymfc17View.cpp
inline CMymfc17Doc* CMymfc17View::GetDocument()
   { return (CMymfc17Doc*)m_pDocument; }
#endif

/////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations
```

```
// immediately before the previous line.

#endif //
!defined(AFX_MYMFC17VIEW_H__4D011049_7E1C_11D0_8FE0_00C04FC2A0C2__INCLUDED_)
```

MYMFC17VIEW.CPP

```
// Mymfc17View.cpp : implementation of the CMymfc17View class
//

#include "stdafx.h"
#include "mymfc17.h"

#include "Mymfc17Doc.h"
#include "Mymfc17View.h"

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

/////////////////////////////////////////////////////////////////////////////
// CMymfc17View

IMPLEMENT_DYNCREATE(CMymfc17View, CFormView)
BEGIN_MESSAGE_MAP(CMymfc17View, CFormView)
    //{{AFX_MSG_MAP(CMymfc17View)
    ON_BN_CLICKED(IDC_CLEAR, OnClear)
    ON_COMMAND(ID_STUDENT_HOME, OnStudentHome)
    ON_COMMAND(ID_STUDENT_END, OnStudentEnd)
    ON_COMMAND(ID_STUDENT_PREV, OnStudentPrev)
    ON_COMMAND(ID_STUDENT_NEXT, OnStudentNext)
    ON_COMMAND(ID_STUDENT_INS, OnStudentIns)
    ON_COMMAND(ID_STUDENT_DEL, OnStudentDel)
    ON_UPDATE_COMMAND_UI(ID_STUDENT_HOME, OnUpdateStudentHome)
    ON_UPDATE_COMMAND_UI(ID_STUDENT_END, OnUpdateStudentEnd)
    ON_UPDATE_COMMAND_UI(ID_STUDENT_PREV, OnUpdateStudentHome)
    ON_UPDATE_COMMAND_UI(ID_STUDENT_NEXT, OnUpdateStudentEnd)
    ON_UPDATE_COMMAND_UI(ID_STUDENT_DEL, OnUpdateStudentDel)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CMymfc17View construction/destruction

CMymfc17View::CMymfc17View() : CFormView(CMymfc17View::IDD)
{
    TRACE("Entering CMymfc17View constructor\n");
    //{{AFX_DATA_INIT(CMymfc17View)
    m_nGrade = 0;
    m_strName = _T("");
    //}}AFX_DATA_INIT
    m_position = NULL;
}

CMymfc17View::~CMymfc17View()
{
}

void CMymfc17View::DoDataExchange(CDataExchange* pDX)
{
    CFormView::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CMymfc17View)
    DDX_Text(pDX, IDC_GRADE, m_nGrade);
    DDV_MinMaxInt(pDX, m_nGrade, 0, 100);
    DDX_Text(pDX, IDC_NAME, m_strName);
```

```
    DDV_MaxChars(pDX, m_strName, 20);
    //}}AFX_DATA_MAP
}
BOOL CMymfc17View::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    //  the CREATESTRUCT cs
    return CFormView::PreCreateWindow(cs);
}

void CMymfc17View::OnInitialUpdate()
{
    TRACE("Entering CMymfc17View::OnInitialUpdate\n");
    m_pList = GetDocument()->GetList();
    CFormView::OnInitialUpdate();
}

/////////////////////////////////////////////////////////////////////////
// CMymfc17View diagnostics

#ifdef _DEBUG
void CMymfc17View::AssertValid() const
{
    CFormView::AssertValid();
}

void CMymfc17View::Dump(CDumpContext& dc) const
{
    CFormView::Dump(dc);
}

CMymfc17Doc* CMymfc17View::GetDocument() // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CMymfc17Doc)));
    return (CMymfc17Doc*)m_pDocument;
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////
// CMymfc17View message handlers

void CMymfc17View::OnClear()
{
    TRACE("Entering CMymfc17View::OnClear\n");
    ClearEntry();
}

void CMymfc17View::OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint)
{
    // called by OnInitialUpdate and by UpdateAllViews
    TRACE("Entering CMymfc17View::OnUpdate\n");
    m_position = m_pList->GetHeadPosition();
    GetEntry(m_position); // initial data for view
}

void CMymfc17View::OnStudentHome()
{
    TRACE("Entering CMymfc17View::OnStudentHome\n");
    // need to deal with list empty condition
    if (!m_pList->IsEmpty()) {
        m_position = m_pList->GetHeadPosition();
        GetEntry(m_position);
    }
}

void CMymfc17View::OnStudentEnd()
```

```
{
    TRACE("Entering CMymfc17View::OnStudentEnd\n");
    if (!m_pList->IsEmpty()) {
        m_position = m_pList->GetTailPosition();
        GetEntry(m_position);
    }
}

void CMymfc17View::OnStudentPrev()
{
    POSITION pos;
    TRACE("Entering CMymfc17View::OnStudentPrev\n");
    if ((pos = m_position) != NULL) {
        m_pList->GetPrev(pos);
        if (pos) {
            GetEntry(pos);
      m_position = pos;
        }
    }
}

void CMymfc17View::OnStudentNext()
{
    POSITION pos;
    TRACE("Entering CMymfc17View::OnStudentNext\n");
    if ((pos = m_position) != NULL) {
        m_pList->GetNext(pos);
        if (pos) {
            GetEntry(pos);
            m_position = pos;
        }
    }
}
void CMymfc17View::OnStudentIns()
{
    TRACE("Entering CMymfc17View::OnStudentIns\n");
    InsertEntry(m_position);
    GetDocument()->SetModifiedFlag();
    GetDocument()->UpdateAllViews(this);
}

void CMymfc17View::OnStudentDel()
{
    // deletes current entry and positions to next one or head
    POSITION pos;
    TRACE("Entering CMymfc17View::OnStudentDel\n");
    if ((pos = m_position) != NULL) {
        m_pList->GetNext(pos);
        if (pos == NULL) {
            pos = m_pList->GetHeadPosition();
            TRACE("GetHeadPos = %ld\n", pos);
            if (pos == m_position) {
                pos = NULL;
            }
        }
        GetEntry(pos);
        CStudent* ps = m_pList->GetAt(m_position);
        m_pList->RemoveAt(m_position);
        delete ps;
        m_position = pos;
        GetDocument()->SetModifiedFlag();
        GetDocument()->UpdateAllViews(this);
    }
}

void CMymfc17View::OnUpdateStudentHome(CCmdUI* pCmdUI)
```

```
{
    // called during idle processing and when Student menu drops down
    POSITION pos;

    // enables button if list not empty and not at home already
    pos = m_pList->GetHeadPosition();
    pCmdUI->Enable((m_position != NULL) && (pos != m_position));
}

void CMymfc17View::OnUpdateStudentEnd(CCmdUI* pCmdUI)
{
    // called during idle processing and when Student menu drops down
    POSITION pos;

    // enables button if list not empty and not at end already
    pos = m_pList->GetTailPosition();
    pCmdUI->Enable((m_position != NULL) && (pos != m_position));
}

void CMymfc17View::OnUpdateStudentDel(CCmdUI* pCmdUI)
{
    // called during idle processing and when Student menu drops down
    pCmdUI->Enable(m_position != NULL);
}

void CMymfc17View::GetEntry(POSITION position)
{
    if (position) {
        CStudent* pStudent = m_pList->GetAt(position);
        m_strName = pStudent->m_strName;
        m_nGrade = pStudent->m_nGrade;
    }
    else {
        ClearEntry();
    }
    UpdateData(FALSE);
}

void CMymfc17View::InsertEntry(POSITION position)
{
    if (UpdateData(TRUE)) {
        // UpdateData returns FALSE if it detects a user error
        CStudent* pStudent = new CStudent;
        pStudent->m_strName = m_strName;
        pStudent->m_nGrade = m_nGrade;
        m_position = m_pList->InsertAfter(m_position, pStudent);
    }
}

void CMymfc17View::ClearEntry()
{
    m_strName = "";
    m_nGrade = 0;
    UpdateData(FALSE);
    ((CDialog*) this)->GotoDlgCtrl(GetDlgItem(IDC_NAME));
}
```

Listing 7: The `CMymfc17View` class listing.

Testing the MYMFC17 Application

Build the program and start it from the debugger, and then test it by typing some data and saving it on disk with the filename **Test.myext**. You don't need to type the **.myext**.

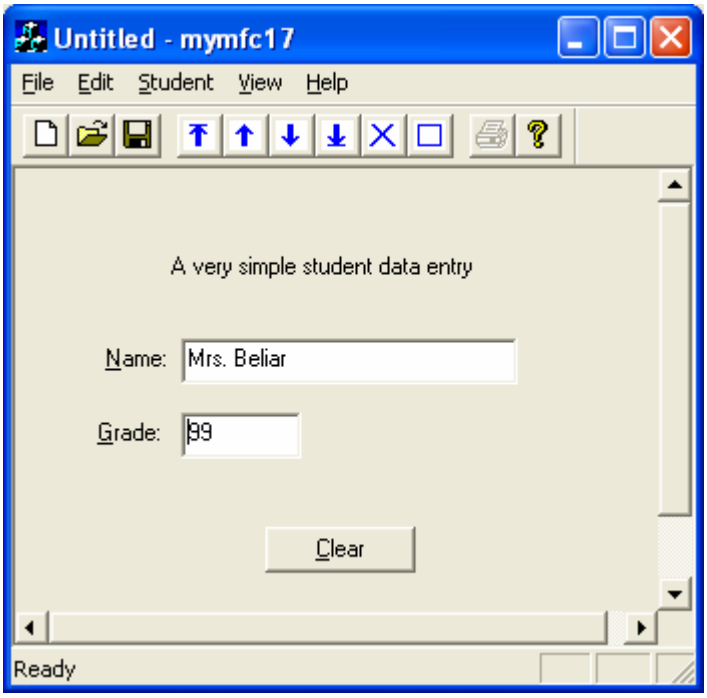

Figure 24: MYMFC17 program output in action.

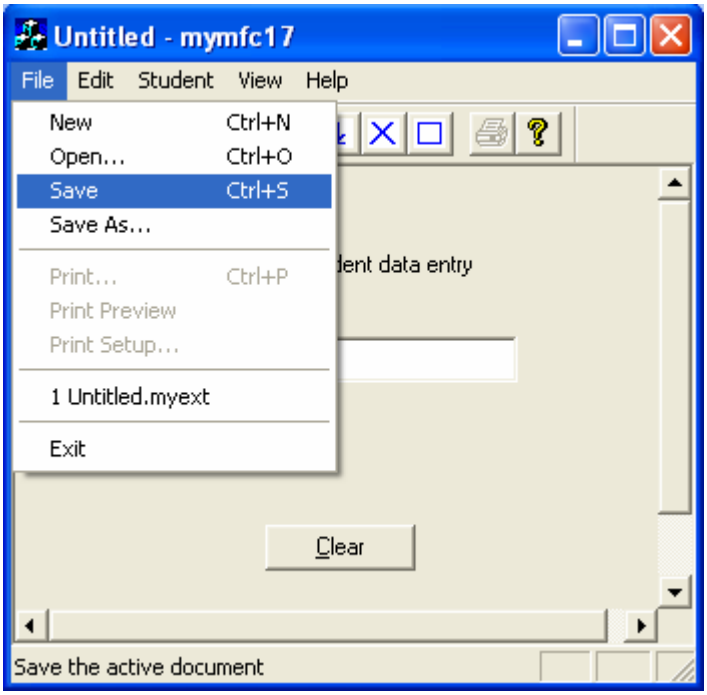

Figure 25: MYMFC17 **Save** menu in action, saving some data.

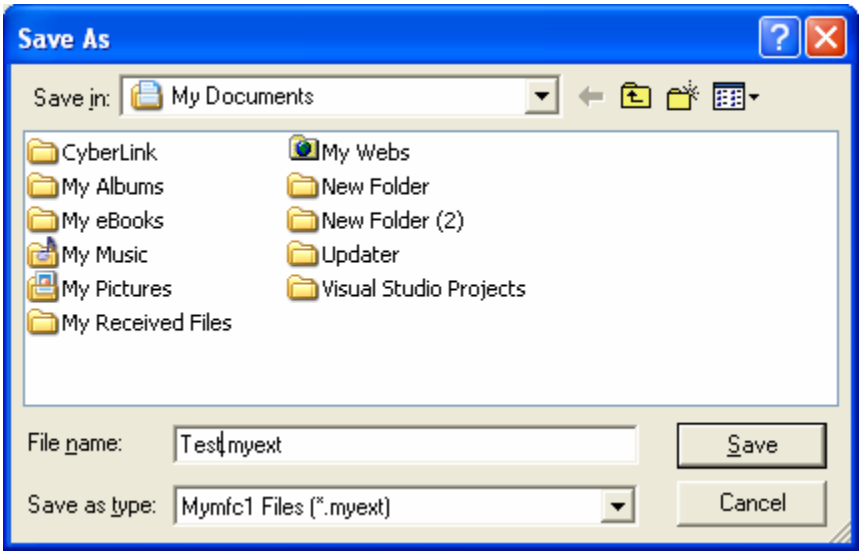

Figure 26: Save dialog, prompting the file name.

Exit the program, and then restart it and open the file you saved. Did the data you typed come back? Take a look at the **Debug** window and observe the sequence of function calls. Is the following sequence produced when you start the application and open the file?

```
...
Entering CMymfc17Doc constructor
...
Entering CMymfc17View constructor
Entering CMymfc17Doc::OnNewDocument
Entering CMymfc17Doc::DeleteContents
Entering CMymfc17View::OnInitialUpdate
Entering CMymfc17View::OnUpdate
Entering CMainFrame::ActivateFrame
...
Entering CMymfc17Doc::OnOpenDocument
Entering CMymfc17Doc::DeleteContents
Entering CMymfc17Doc::Serialize
Entering CMymfc17Doc::Serialize
Entering CMymfc17Doc::Serialize

Entering CMymfc17View::OnInitialUpdate
Entering CMymfc17View::OnUpdate
Entering CMainFrame::ActivateFrame
...
The thread 0xB30 has exited with code 0 (0x0).
The program 'F:\mfcproject\mymfc17\Debug\mymfc17.exe' has exited with code 0 (0x0).
```

Explorer Launch and Drag and Drop

In the past, PC users were accustomed to starting up a program and then selecting a disk file (sometimes called a document) that contained data the program understood. Many MS-DOS-based programs worked this way. The old Windows Program Manager improved things by allowing the user to double-click on a program icon instead of typing a program name. Meanwhile, Apple Macintosh users were double-clicking on a document icon; the Macintosh operating system figured out which program to run. While Windows Explorer still lets users double-click on a program, it also lets users double-click on a document icon to run the document's program. But how does Explorer know which program to run? Explorer uses the Windows Registry to make the connection between document and program. The link starts with the filename extension that you typed into AppWizard, but as you'll see, there's more to it than that. Once the association is made, users can launch your program by double-clicking on its document icon or by dragging the icon from Explorer to a running instance of your program. In addition, users can drag the icon to a printer, and your program will print it.

Program Registration

In Module 9, you saw how MFC applications store data in the Windows Registry by calling `SetRegistryKey()` from the `InitInstance()` function. Independent of this `SetRegistryKey()` call, your program can write file association information in a different part of the Registry on startup. To activate this feature, you must type in the filename extension when you create the application with AppWizard. (Use the Advanced button in AppWizard Step 4.) After you do that, AppWizard adds the extension as a substring in your template string and adds the following line in your `InitInstance()` function:

```
RegisterShellFileTypes(TRUE);
```

```
// Enable DDE Execute open
EnableShellOpen();
RegisterShellFileTypes(TRUE);
// Parse command line for standard
CCommandLineInfo cmdInfo;
```

Listing 8.

Now your program adds two items to the **Registry**. Under the `HKEY_CLASSES_ROOT` top-level key, it adds a subkey and a data string as shown here for the MYMFC17 example:

```
.myext = Mymfc17.Document
```

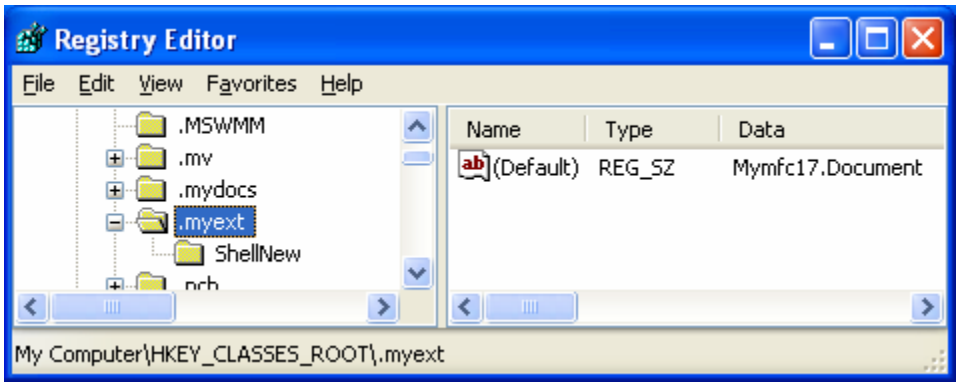

Figure 27: Registry information for MYMFC17 project.

The data item is the file type ID that AppWizard has chosen for you. `Mymfc17.Document`, in turn, is the key for finding the program itself. The Registry entries for `Mymfc17.Document`, also beneath `HKEY_CLASSES_ROOT`, are shown here.

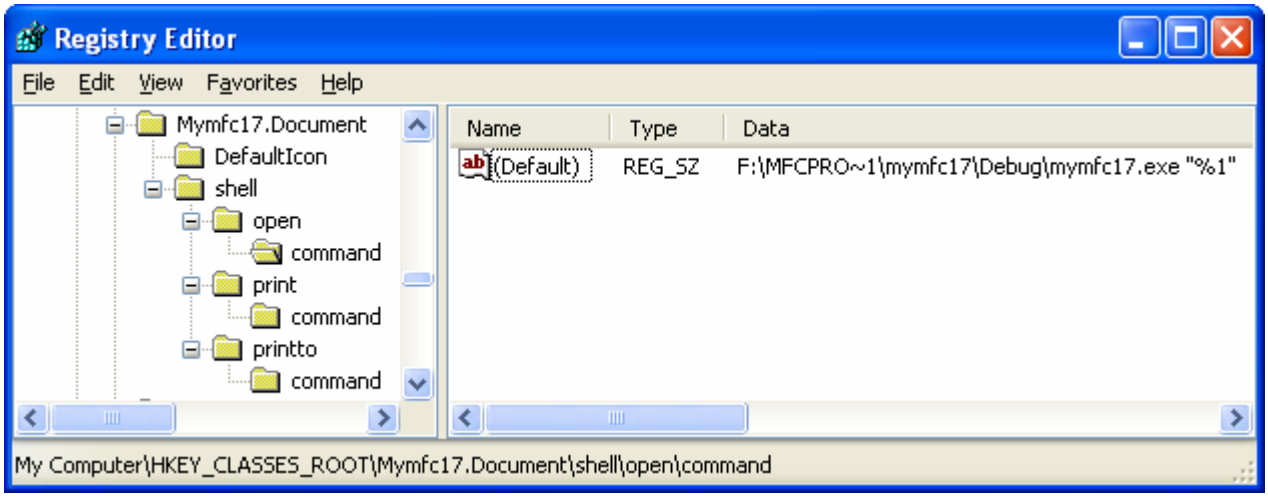

Figure 28: Another Registry information for MYMFC17.

Notice that the Registry contains the full pathname of the MYMFC17 program. Now Explorer can use the Registry to navigate from the extension to the file type ID to the actual program itself. After the extension is registered, Explorer finds the document's icon and displays it next to the filename, as shown here.

Figure 29: Icon and file extension.

Double-Clicking on a Document

When the user double-clicks on a document icon, Explorer executes the associated SDI program, passing in the selected filename on the command line. You might notice that AppWizard generates a call to `EnableShellOpen()` in the application class `InitInstance()` function. This supports execution via DDE message, the technique used by the File Manager in Windows NT 3.51. Explorer can launch your SDI application without this call.

Enabling Drag and Drop

If you want your already-running program to open files dragged from Explorer, you must call the `CWnd` function `DragAcceptFiles()` for the application's main frame window. The application object's public data member `m_pMainWnd` points to the `CFrameWnd` (or `CMDIFrameWnd`) object. When the user drops a file anywhere inside the frame window, the window receives a `WM_DROPFILES` message, which triggers a call to `FrameWnd::OnDropFiles`. The following line in `InitInstance()`, generated by AppWizard, enables drag and drop:

```
      m_pMainWnd->DragAcceptFiles();
```

```
    // Enable drag/drop open
    m_pMainWnd->DragAcceptFiles();

    return TRUE;
}
```

Listing 9.

Program Startup Parameters

When you choose **Run** from the **Start** menu, or when you double-click the program directly in Explorer, there is no command-line parameter. The `InitInstance()` function processes the command line with calls to `ParseCommandLine()` and `ProcessShellCommand()`. If the command line contains something that looks like a filename, the program immediately loads that file. Thus, you create a Windows shortcut that can run your program with a specific document file.

Experimenting with Explorer Launch and Drag and Drop

Once you have built MYMFC17, you can try running it from Explorer. You must execute the program directly, however, in order to write the initial entries in the Registry. Be sure that you've saved at least one **myext** file to disk, and then exit MYMFC17. Start Explorer, and then open the \vcpp32\mymfc17 directory. Double-click on one of the myext files in the panel on the right. Your program should start with the selected file loaded. Now, with both MYMFC17 and Explorer open on the desktop, try dragging another file from Explorer to the MYMFC17 window. The program should open the new file just as if you had chosen **File Open** from the MYMFC17 menu.
You might also want to look at the MYMFC17 entries in the Registry. Run the **Regedit** program (possibly named **Regedt32** in Windows NT), and expand the `HKEY_CLASSES_ROOT` key. Look under "`.myext`" and "`Mymfc17.Document`."
Also expand the `HKEY_CURRENT_USER` (or `HKEY_USERS\.DEFAULT`) key, and look under "`Software`." You should see a **Recent File List** under the subkey **mymfc17**.

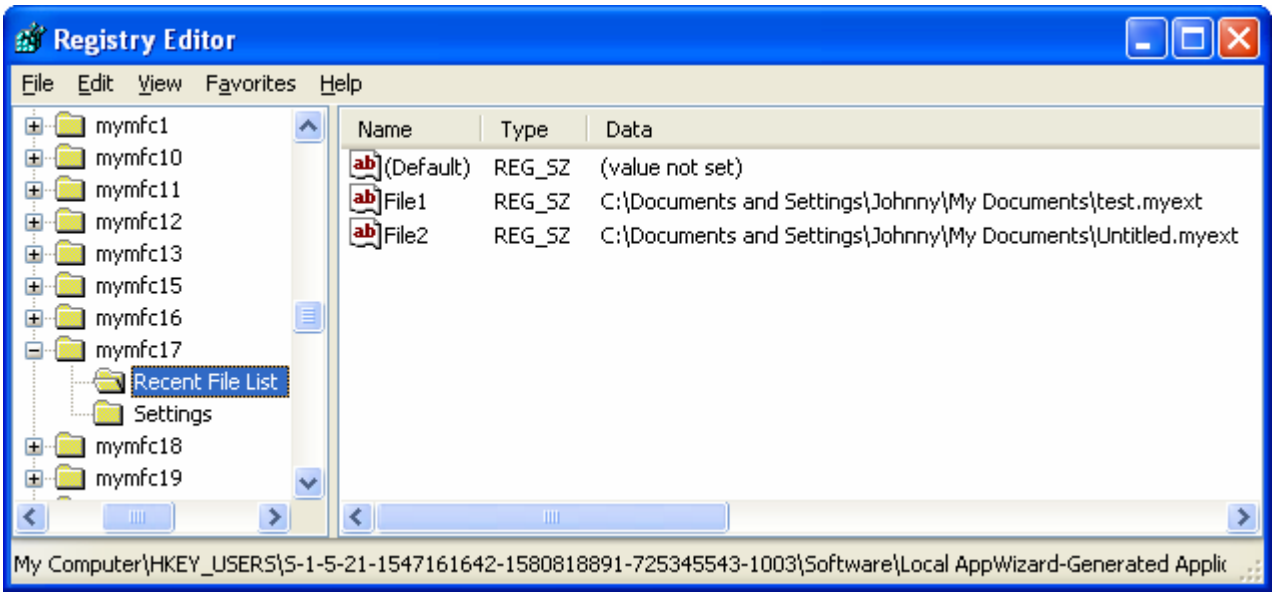

Figure 30: **Recent File List** in Registry for MYMFC17 project.

The MYMFC17 program calls `SetRegistryKey()` with the string "`Local AppWizard-Generated Applications`", so the program name goes beneath the **mymfc17** subkey.

Further reading and digging:

1. Standard C File Input/Output.
2. Standard C++ File Input/Output.
3. Win32 File Input/Output: Module C, Module D and Module E.
4. MSDN MFC 6.0 class library online documentation - used throughout this Tutorial.
5. MSDN MFC 7.0 class library online documentation - used in .Net framework and also backward compatible with 6.0 class library
6. MSDN Library
7. Windows data type.
8. Win32 programming Tutorial.
9. The best of C/C++, MFC, Windows and other related books.
10. Unicode and Multibyte character set: Story and program examples.