Structured Storage

Program examples compiled using Visual C++ 6.0 compiler on Windows XP Pro machine with Service Pack 2. Topics and sub topics for this tutorial are listed below. Don't forget to read Tenouk's small disclaimer. The supplementary notes for this tutorial are statstg, Thread.h, ReadThread.cpp and WriteThread.cpp.

Index:

Intro

Like **Automation** and **Uniform Data Transfer**, structured storage is one of those **COM features** that you can use effectively by itself. Of course, it's also behind much of the **ActiveX technology**, particularly **compound documents**.
In this module, you'll learn to write and read compound files with the `IStorage` and `IStream` interfaces. The `IStorage` interface is used to create and manage structured storage objects. `IStream` is used to manipulate the data contained by the storage object. The `IStorage` and `IStream` interfaces, like all COM interfaces, are simply virtual function declarations. Compound files, on the other hand, are implemented by code in the Microsoft Windows OLE32 DLL. **Compound files** represent a Microsoft file I/O standard that you can think of as "a file system inside a file."
When you're familiar with `IStorage` and `IStream`, you'll move on to the `IPersistStorage` and `IPersistStream` interfaces. With the `IPersistStorage` and `IPersistStream` interfaces, you can program a class to save and load objects to and from a compound file. You say to an object, "Save yourself," and it knows how.

Compound Files

This MFC Tutorial discusses four options for file I/O. You can read and write whole sequential files (like the MFC archive files you saw first in Module 11). You can use a database management system. You can write your own code for random file access. Finally, you can use compound files.

Think of a compound file as **a whole file system within a file**. Figure 1 shows a traditional disk directory as supported by early MS-DOS systems and by Microsoft Windows. This directory is composed of files and subdirectories, with a root directory at the top. Now imagine the same structure inside **a single disk file**. The files are called **streams**, and the directories are called **storages**. Each is identified by a name of up to 32 wide characters in length. A stream is a logically **sequential array of bytes**, and a storage is a collection of streams and sub-storages.

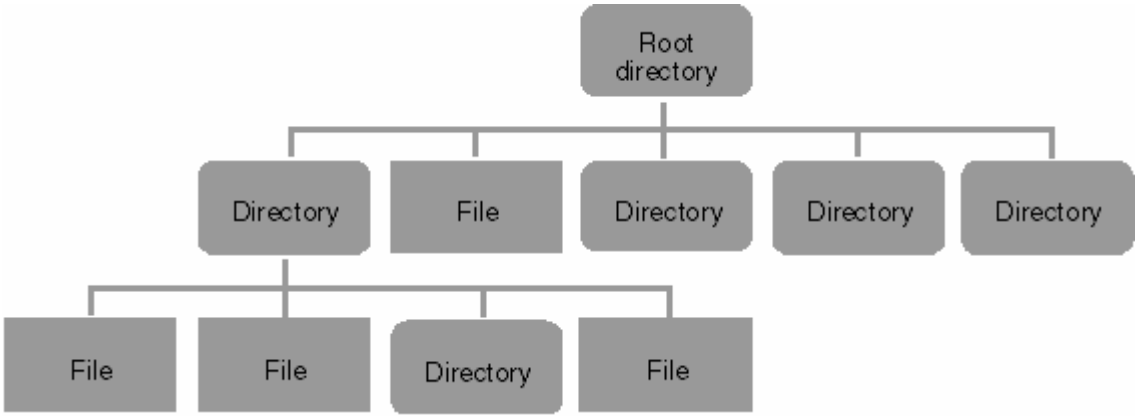

Figure 1: A disk directory with files and subdirectories.

A storage can contain other storages, just as a directory can contain subdirectories. In a disk file, the bytes aren't necessarily stored in contiguous clusters. Similarly, the bytes in a stream aren't necessarily contiguous in their compound file. They just appear that way.
Storage and stream names cannot contain the characters `/`, `\`, `:`, or `!`. If the first character has an ASCII value of less than 32, the element is marked as managed by some agent other than the owner.
You can probably think of many applications for a compound file. The classic example is a large document composed of modules and paragraphs within modules. The document is so large that you don't want to read the whole thing into memory when your program starts, and you want to be able to insert and delete portions of the document. You could design a compound file with a root storage that contains sub-storages for modules. The module sub-storages would contain streams for the paragraphs. Other streams could be for index information.
One useful feature of compound files is transactioning. When you start a transaction for a compound file, all changes are written to a temporary file. The changes are made to your file only when you commit the transaction.

Storages and the `IStorage` Interface

If you have a storage object, you can manipulate it through the `IStorage` interface. Pay attention to these functions because Microsoft Foundation Class offers no support for storage access. The `IStorage` interface inherits the methods of the standard COM interface `IUnknown` (`QueryInterface()`, `AddRef()` and `Release()`). In addition, `IStorage` defines the following methods.

Method/Function	Description
`CreateStream()`	Creates and opens a stream object with the specified name contained in this storage object. The name must not exceed **31** characters in length (not including the string terminator). The **000** through **01f** characters, serving as the first character of the stream/storage name, are reserved for use by OLE. This is a compound file restriction, not a structured storage restriction.
`OpenStream()`	Opens an existing stream object within this storage object using the specified access permissions in `grfMode`. The name must not exceed 31 characters in length (not including the string terminator). The 000 through 01f characters, serving as the first character of the stream/storage name, are reserved for use by OLE. This is a compound file restriction, not a structured storage restriction.
`CreateStorage()`	Creates and opens a new storage object within this storage object. The name must not exceed 31 characters in length (not including the string terminator). The 000 through 01f characters, serving as the first character of the stream/storage name, are reserved for use by OLE. This is a compound

	file restriction, not a structured storage restriction.
`OpenStorage()`	Opens an existing storage object with the specified name according to the specified access mode. The name must not exceed 31 characters in length (not including the string terminator). The 000 through 01f characters, serving as the first character of the stream/storage name, are reserved for use by OLE. This is a compound file restriction, not a structured storage restriction.
`CopyTo()`	Copies the entire contents of this open storage object into another storage object. The layout of the destination storage object may differ.
`MoveElementTo()`	Copies or moves a substorage or stream from this storage object to another storage object.
`Commit()`	Reflects changes for a transacted storage object to the parent level.
`Revert()`	Discards all changes that have been made to the storage object since the last commit operation.
`EnumElements()`	Returns an enumerator object that can be used to enumerate the storage and stream objects contained within this storage object.
`DestroyElement()`	Removes the specified storage or stream from this storage object.
`RenameElement()`	Renames the specified storage or stream in this storage object.
`SetElementTimes()`	Sets the modification, access, and creation times of the indicated storage element, if supported by the underlying file system.
`SetClass()`	Assigns the specified `CLSID` to this storage object.
`SetStateBits()`	Stores up to 32 bits of state information in this storage object.
`Stat()`	Returns the `STATSTG` structure for this open storage object.

Table 1.

Following are some of the important member functions and their significant parameters.

```
HRESULT Commit(…);
```

Commits all the changes to this storage and to all elements below it.

```
HRESULT CopyTo(…, IStorage**pStgDest);
```

Copies a storage, with its name and all its sub-storages and streams (recursively), to another existing storage. Elements are merged into the target storage, replacing elements with matching names.

```
HRESULT CreateStorage(const WCHAR*
pName, …, DWORD mode, …, IStorage** ppStg);
```

Creates a new sub-storage under this storage object.

```
HRESULT CreateStream(const WCHAR*
pName, …, DWORD mode, …, IStream** ppStream);
```

Creates a new stream under this storage object.

```
HRESULT DestroyElement(const WCHAR* pName);
```

Destroys the named storage or stream that is under this storage object. A storage cannot destroy itself.

```
HRESULT EnumElements(…, IEnumSTATSTG** ppEnumStatstg);
```

Iterates through all the storages and streams under this storage object. The `IEnumSTATSTG` interface has `Next()`, `Skip()`, and `Clone()` member functions, as do other COM enumerator interfaces.

```
HRESULT MoveElementTo(const WCHAR* pName,
IStorage* pStgDest, const LPWSTR* pNewName, DWORD flags);
```

Moves an element from this storage object to another storage object.

```
HRESULT OpenStream(const WCHAR*
pName, …, DWORD mode, …, IStorage** ppStg);
```

Opens an existing stream object, designated by name, under this storage object.

```
HRESULT OpenStorage(const WCHAR*
pName, …, DWORD mode, …, IStorage** ppStg);
```

Opens an existing substorage object, designated by name, under this storage object.

```
DWORD Release(void);
```

Decrements the reference count. If the storage is a root storage representing a disk file, `Release()` closes the file when the reference count goes to 0.

```
HRESULT RenameElement(const
WCHAR* pOldName, const WCHAR* pNewName);
```

Assigns a new name to an existing storage or stream under this storage object.

```
HRESULT Revert(void);
```

Abandons a transaction, leaving the compound file unchanged.

```
HRESULT SetClass(CLSID& clsid);
```

Inserts a 128-bit class identifier into this storage object. This ID can then be retrieved with the `Stat()` function.

```
HRESULT Stat(STATSTG* pStatstg, DWORD flag);
```

Fills in a `STATSTG` structure with useful information about the storage object, including its name and class ID.

Getting an `IStorage` Pointer

Where do you get the first `IStorage` pointer? COM gives you the global function `StgCreateDocfile()` to create a new structured storage file on disk and the function `StgOpenStorage()` to open an existing file. Both of these set a pointer to the file's root storage. Here's some code that opens an existing storage file named **MyStore.stg** and then creates a new sub-storage:

```
IStorage* pStgRoot;
IStorage* pSubStg;

if (::StgCreateDocfile(L"MyStore.stg",
    STGM_READWRITE | STGM_SHARE_EXCLUSIVE | STGM_CREATE, 0, &pStgRoot) ==
S_OK)
    {
    if (pStgRoot->CreateStorage(L"MySubstorageName",
        STGM_READWRITE | STGM_SHARE_EXCLUSIVE | STGM_CREATE,
        0, 0, &pSubStg) == S_OK) {
        // Do something with pSubStg
        pSubStg->Release();
    }
    pStgRoot->Release();
}
```

Freeing `STATSTG` Memory

When you call `IStorage::Stat` with a `STATFLAG_DEFAULT` value for the flag parameter, COM allocates memory for the element name. You must free this memory in a manner compatible with its allocation. COM has its own allocation system that uses an allocator object with an `IMalloc` interface. You must get an `IMalloc` pointer from COM, call `IMalloc::Free` for the string, and then release the allocator. The code below illustrates this. If you want just the element size and type and not the name, you can call `Stat()` with the `STATFLAG_NONAME` flag. In that case, no memory is allocated and you don't have to free it. This seems like an irritating detail, but if you don't follow the recipe, you'll have a memory leak.

Enumerating the Elements in a Storage Object

Following is some code that iterates through all the elements under a storage object, differentiating between sub-storages and streams. The elements are retrieved in a seemingly random sequence, independent of the sequence in which they were created; however, I've found that streams are always retrieved first. The `IEnumSTATSTG::Next` element fills in a `STATSTG` structure that tells you whether the element is a stream or a storage object.

```
IEnumSTATSTG* pEnum;
IMalloc* pMalloc;
STATSTG statstg;
extern IStorage* pStg;  // maybe from OpenStorage
::CoGetMalloc(MEMCTX_TASK, &pMalloc); // assumes AfxOleInit called
VERIFY(pStg->EnumElements(0, NULL, 0, &pEnum) == S_OK)
while (pEnum->Next(1, &statstg, NULL) == NOERROR) {
    if (statstg.type == STGTY_STORAGE) {
        if (pStg->OpenStorage(statstg.pwcsName, NULL,
            STGM_READ | STGM_SHARE_EXCLUSIVE,
            NULL, 0, &pSubStg) == S_OK) {
            // Do something with the substorage
        }
        else if (statstg.type == STGTY_STREAM)
        {
            // Process the stream
        }
        pMalloc->Free(statstg.pwcsName); // avoids memory leaks
    }
    pMalloc->Release();
}
```

Sharing Storages Among Processes

If you pass an `IStorage` pointer to another process, the marshaling code ensures that the other process can access the corresponding storage element and everything below it. This is a convenient way of sharing part of a file. One of the standard data object media types of the `TYMED` enumeration is `TYMED_ISTORAGE`, and this means you can pass an `IStorage` pointer on the clipboard or through a drag-and-drop operation.

Streams and the `IStream` Interface

If you have a stream object, you can manipulate it through the `IStream` interface. Streams are always located under a root storage or a substorage object. Streams grow automatically (in 512-byte increments) as you write to them. An MFC class for streams, `COleStreamFile`, makes a stream look like a `CFile` object. That class won't be of much use to us in this module, however. The `IStream` interface inherits the methods of the standard COM interface `IUnknown` (`QueryInterface()`, `AddRef()` and `Release()`). In addition, `IStream` defines the following methods. Once you have a pointer to `IStream`, a number of functions are available to you for manipulating the stream. Here is a list and the detail of all the `IStream` functions.

Method	Description
`Read()`	Reads a specified number of bytes from the stream object into memory starting at the current seek pointer.
`Write()`	Writes a specified number of bytes into the stream object starting at the current seek pointer.

`Seek()`	Changes the seek pointer to a new location relative to the beginning of the stream, the end of the stream, or the current seek pointer.
`SetSize()`	Changes the size of the stream object.
`CopyTo()`	Copies a specified number of bytes from the current seek pointer in the stream to the current seek pointer in another stream.
`Commit()`	Ensures that any changes made to a stream object open in transacted mode are reflected in the parent storage object.
`Revert()`	Discards all changes that have been made to a transacted stream since the last call to `IStream::Commit`.
`LockRegion()`	Restricts access to a specified range of bytes in the stream. Supporting this functionality is optional since some file systems do not provide it.
`UnlockRegion()`	Removes the access restriction on a range of bytes previously restricted with `IStream::LockRegion`.
`Stat()`	Retrieves the `STATSTG` structure for this stream.
`Clone()`	Creates a new stream object that references the same bytes as the original stream but provides a separate seek pointer to those bytes.

Table 2.

The following are some of the important functions used in this tutorial.

```
HRESULT CopyTo(IStream** pStm, ULARGE_INTEGER cb, …);
```

Copies `cb` bytes from this stream to the named stream. `ULARGE_INTEGER` is a structure with two 32-bit members—`HighPart` and `LowPart`.

```
HRESULT Clone(IStream** ppStm);
```

Creates a new stream object with its own seek pointer that references the bytes in this stream. The bytes are not copied, so changes in one stream are visible in the other.

```
HRESULT Commit(…);
```

Transactions are not currently implemented for streams.

```
HRESULT Read(void const* pv, ULONG cb, ULONG* pcbRead);
```

Tries to read `cb` bytes from this stream into the buffer pointed to by `pv`. The variable `pcbRead` indicates how many bytes were actually read.

```
DWORD Release(void);
```

Closes this stream.

```
HRESULT Revert(void);
```

Has no effect for streams.

```
HRESULT Seek(LARGE_INTEGER dlibMove,
DWORD dwOrigin, ULARGE_INTEGER* NewPosition);
```

Seeks to the specified position in this stream. The `dwOrigin` parameter specifies the origin of the offset defined in the `NewPosition` parameter.

```
HRESULT SetSize(ULARGE_INTEGER libNewSize);
```

Extends or truncates a stream. Streams grow automatically as they are written, but calling `SetSize()` can optimize performance.

```
HRESULT Stat(STATSTG* pStatstg, DWORD flag);
```

Fills in the STATSTG structure with useful information about the stream, including the stream name and size. The size is useful if you need to allocate memory for a read.

```
HRESULT Write(void const* pv, ULONG cb, ULONG* pcbWritten);
```

Tries to write cb bytes to this stream from the buffer pointed to by pv. The variable pcbWritten indicates how many bytes were actually written.

IStream Programming

Here is some sample code that creates a stream under a given storage object and writes some bytes from m_buffer to the stream:

```
extern IStorage* pStg;
IStream* pStream;
ULONG nBytesWritten;

if (pStg->CreateStream(L"MyStreamName",
    STGM_CREATE | STGM_READWRITE | STGM_SHARE_EXCLUSIVE,
    0, 0, &pStream) == S_OK)
{
    ASSERT(pStream != NULL);
    pStream->Write(m_buffer, m_nLength, &nBytesWritten);
    pStream->Release();
}
```

The ILockBytes Interface

As already mentioned, the compound file system you've been looking at is implemented in the OLE32 DLL. The structured storage interfaces are flexible enough, however, to permit you to change the underlying implementation. The key to this flexibility is the ILockBytes interface. The ILockBytes interface inherits the methods of the standard COM interface IUnknown (QueryInterface(), AddRef() and Release()). In addition, ILockBytes defines the following methods.

Method	Description
ReadAt()	Reads a specified number of bytes starting at a specified offset from the beginning of the byte array.
WriteAt()	Writes a specified number of bytes to a specified location in the byte array.
Flush()	Ensures that any internal buffers maintained by the byte array object are written out to the backing storage.
SetSize()	Changes the size of the byte array.
LockRegion()	Restricts access to a specified range of bytes in the byte array.
UnlockRegion()	Removes the access restriction on a range of bytes previously restricted with ILockBytes::LockRegion.
Stat()	Retrieves a STATSTG structure for this byte array object.

Table 3.

The StgCreateDocfile() and StgOpenStorage() global functions use the default Windows file system. You can write your own file access code that implements the ILockBytes interface and then call StgCreateDocfileOnILockBytes() or StgOpenStorageOnILockBytes() to create or open the file, instead of calling the other global functions.
Rather than implement your own ILockBytes interface, you can call CreateILockBytesOnHGlobal() to create a compound file in RAM. If you wanted to put compound files inside a database, you would implement an ILockBytes interface that used the database's blobs (binary large objects).

The MYMFC31A Example: Structured Storage

When you choose the **Storage Write** option in the MYMFC31A example, the program walks through your entire disk directory looking for TXT files. As it looks, it writes a compound file (**\direct.stg**) on the top level of your directory structure. This file contains storages that match your subdirectories. For each TXT file that the program finds in a subdirectory, it copies the first line of text to a stream in the corresponding storage. When you choose the **Storage Read** option, the program reads the **direct.stg** compound file and prints the contents of this file in the **Debug** window. If you create such an example from scratch, use AppWizard without any **ActiveX** or **Automation** options and then add the following lines in your **StdAfx.h** file:

```
#include <afxole.h>
#include <afxpriv.h> // for wide-character conversion
```

Then delete the following line:

```
#define VC_EXTRALEAN
```

To prepare MYMFC31A, open the **mymfc31a.dsw** workspace and build the project. Run the program from the debugger. First choose **Write** from the **Storage** menu and wait for a "Write complete" message box. Then choose **Read**. Observe the output in the **Debug** window.

The MYMFC31A From Scratch

This is SDI application without **ActiveX** or **Automation** support.

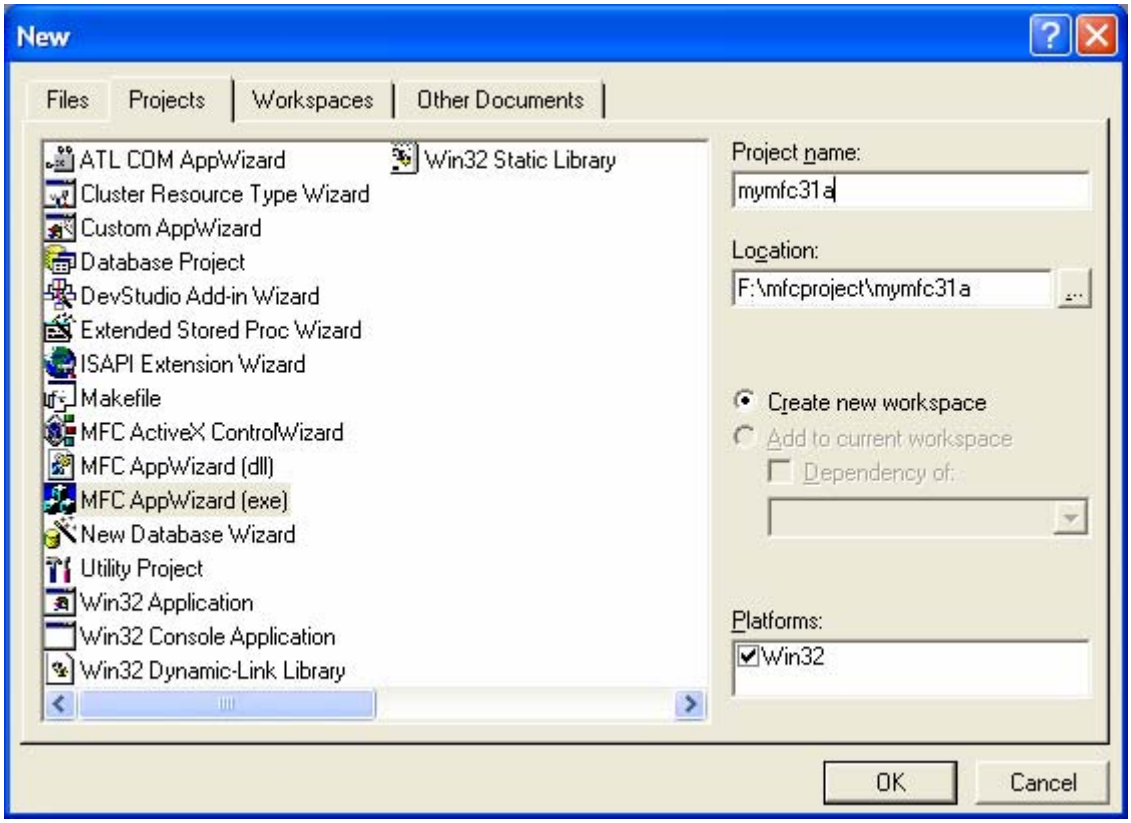

Figure 2: MYMFC31A – Visual C++ new project dialog.

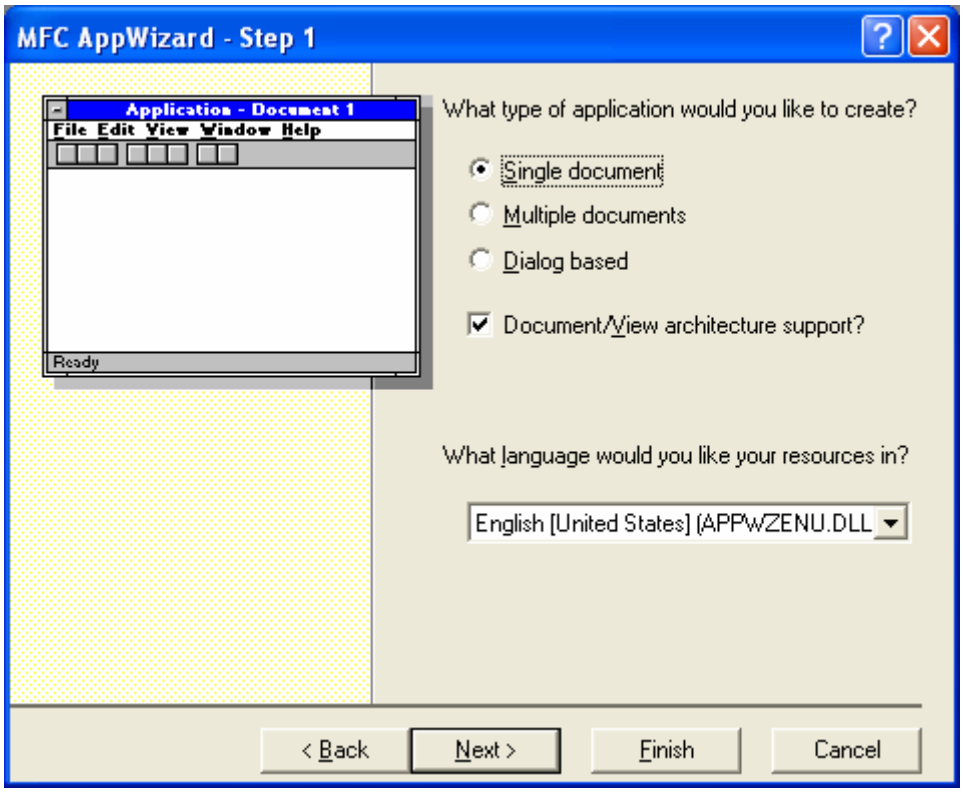

Figure 3: MYMFC31A – AppWizard step 1 of 6, an SDI application.

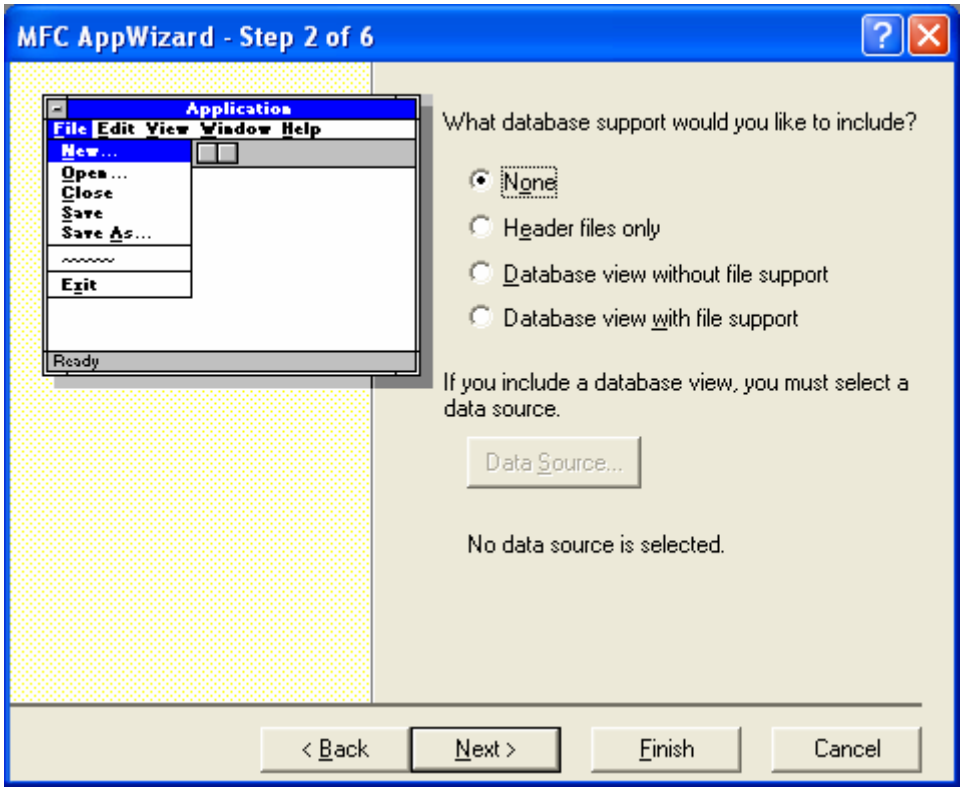

Figure 4: MYMFC31A – AppWizard step 2 of 6.

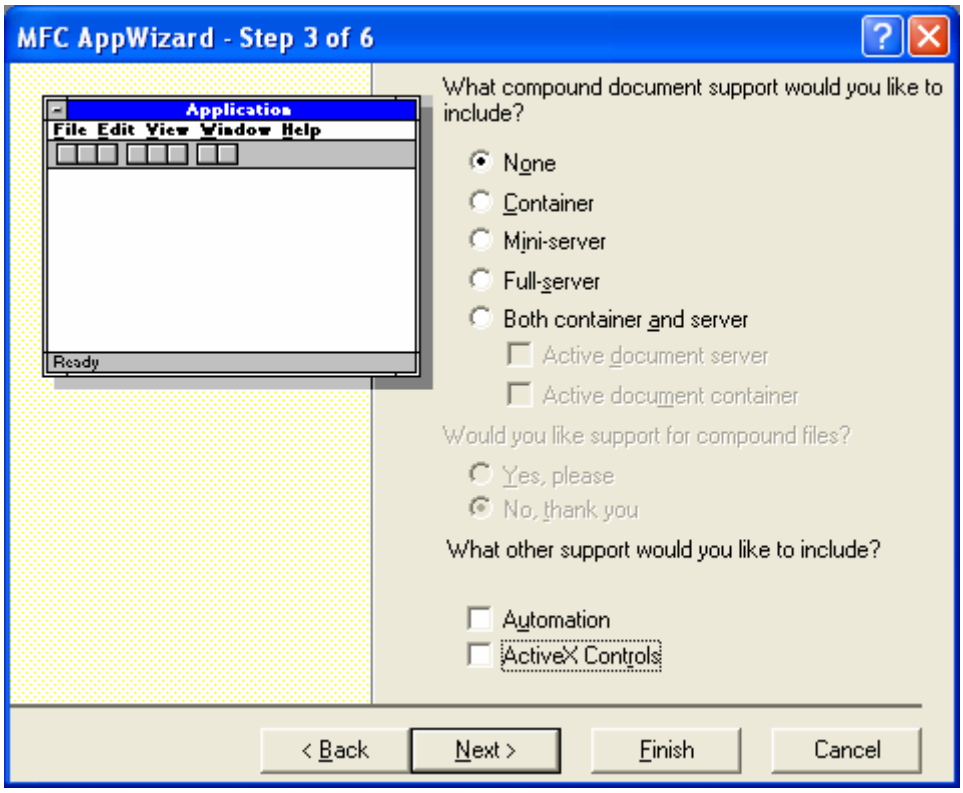

Figure 5: MYMFC31A – AppWizard step 3 of 6, deselect Automation and ActiveX Controls support options.

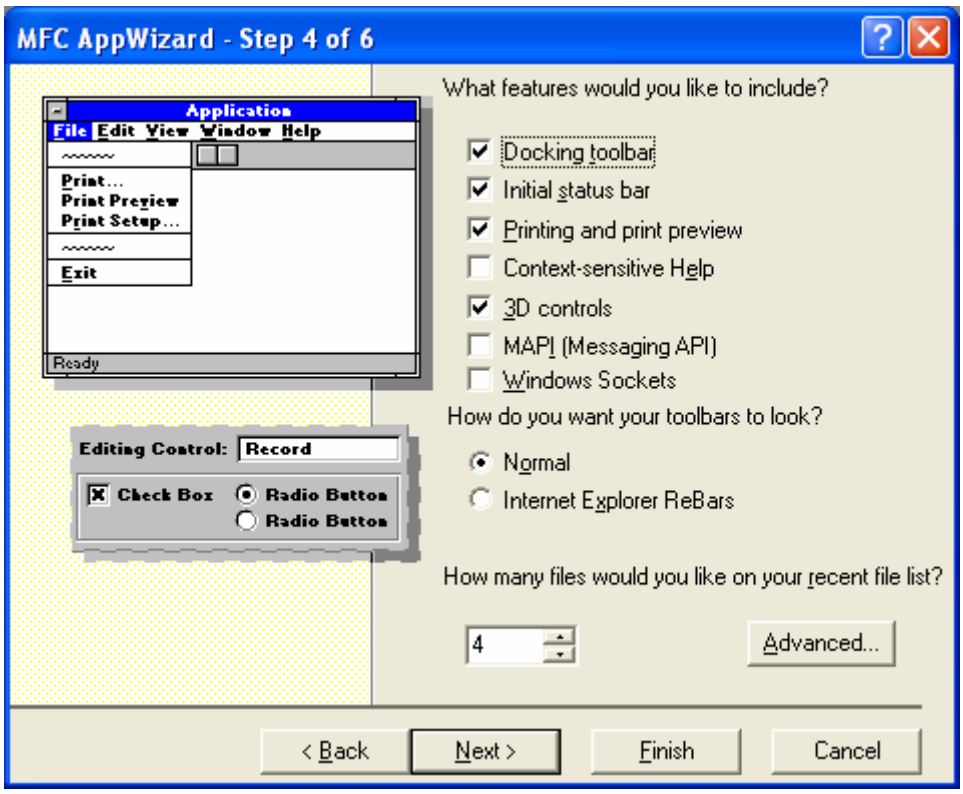

Figure 6: MYMFC31A – AppWizard step 4 of 6

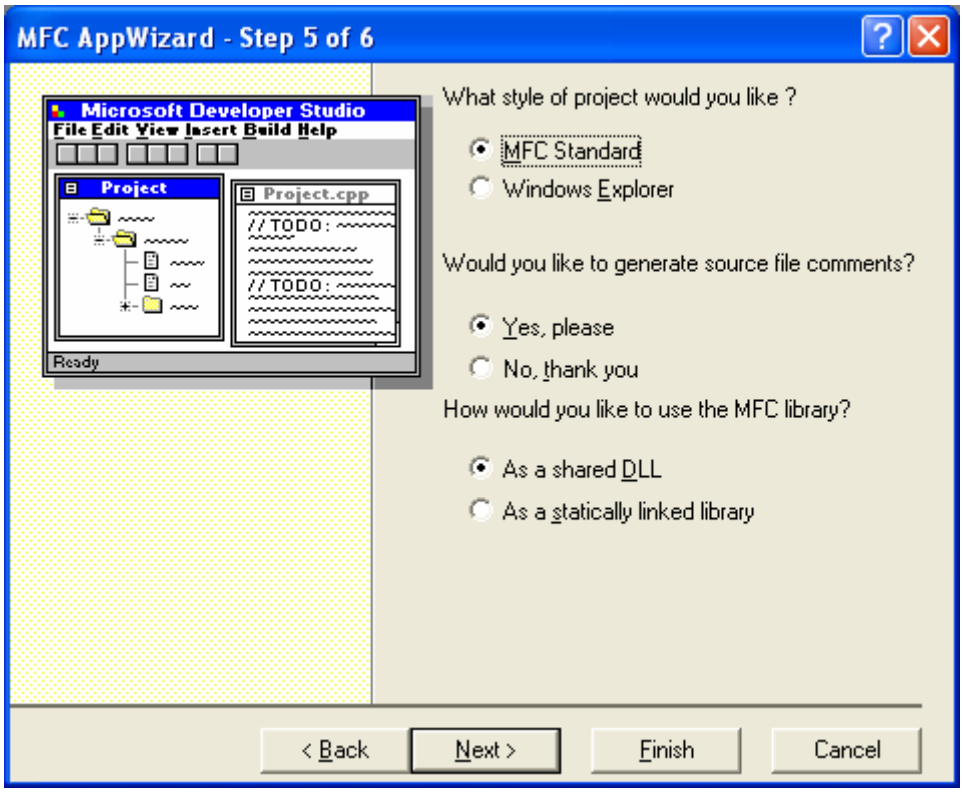

Figure 7: MYMFC31A – AppWizard step 5 of 6

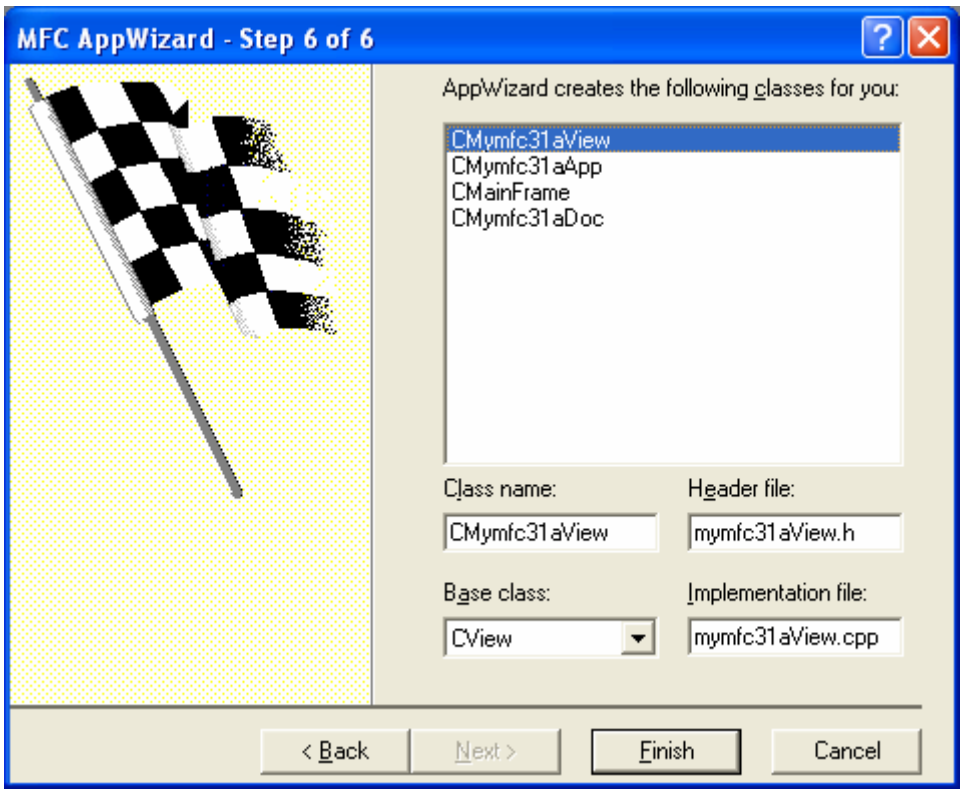

Figure 8: MYMFC31A – AppWizard step 6 of 6

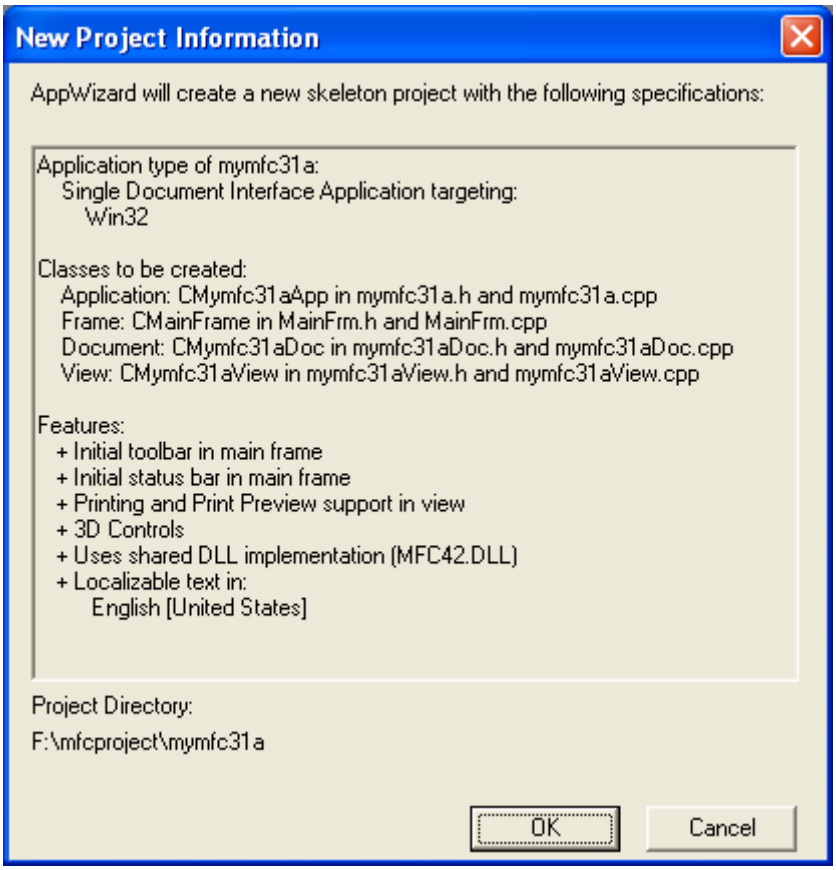

Figure 9: MYMFC31A project summary.

Add the following menu items.

ID	Caption
-	Storage
ID_STORAGE_READ	Read
ID_STORAGE_WRITE	Write

Table 4.

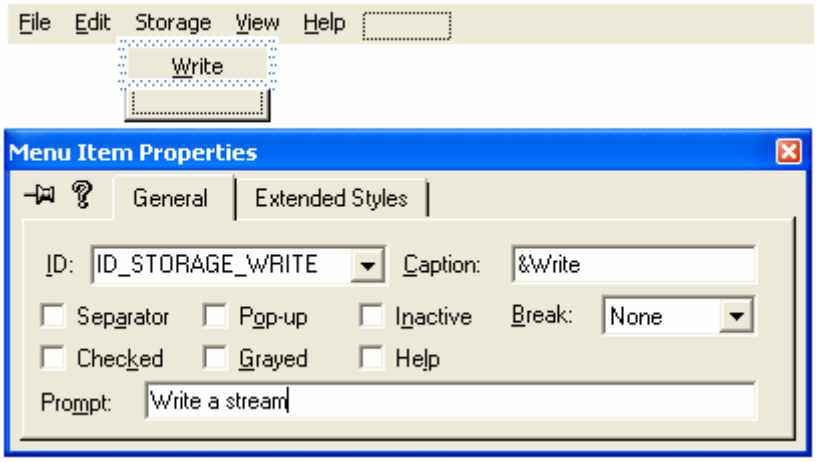

Figure 10: Adding **Storage** menu and **Write** sub menu.

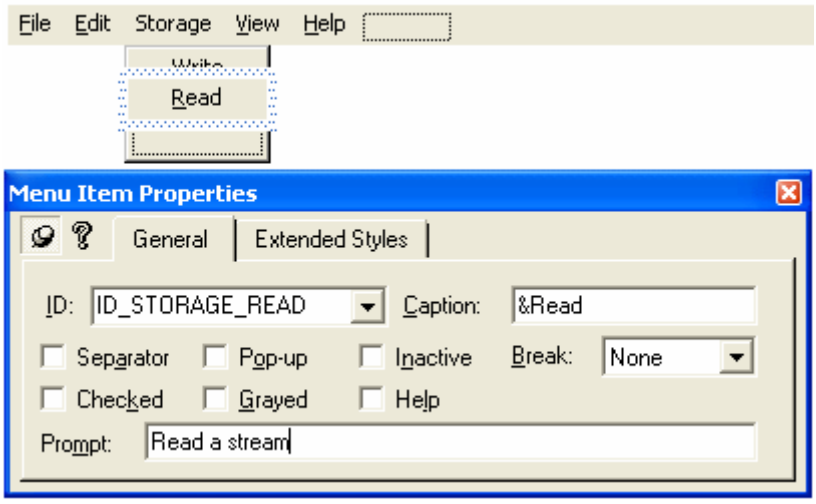

Figure 11: Adding **Read** sub menu.

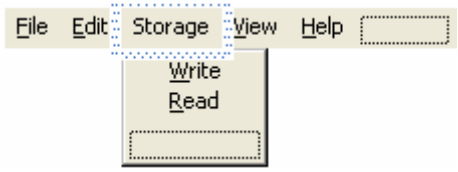

Figure 12: A completed MYMFC31A menus.

Use ClassWizard to add command handlers as listed in the following Table for the menu items in `CMymfc31aView` class.

ID	Function
`ID_STORAGE_READ`	`OnStorageRead()`
`ID_STORAGE_WRITE`	`OnStorageWrite()`

Table 5.

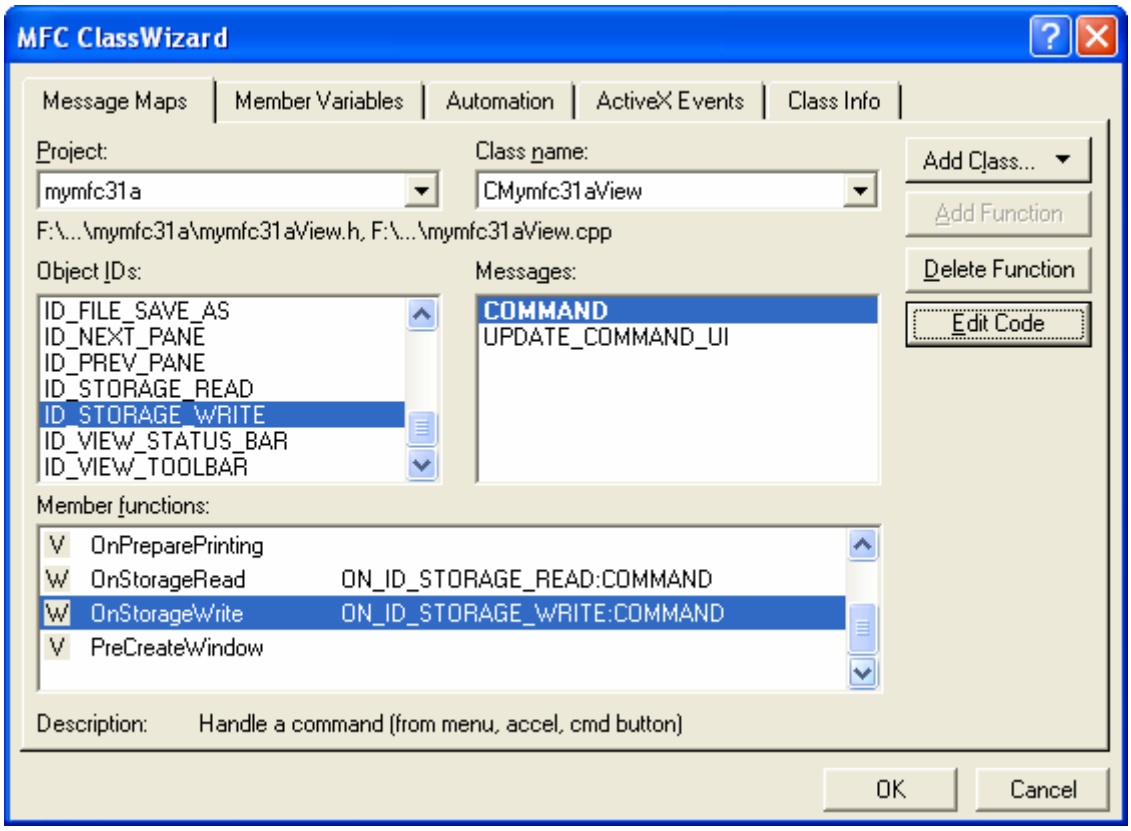

Figure 13: Adding command handlers for menu items.

Then, add codes to the command handlers.

```
void CMymfc31aView::OnStorageRead()
{
	// TODO: Add your command handler code here
	CWinThread* pThread = AfxBeginThread(ReadThreadProc, GetSafeHwnd());
}

void CMymfc31aView::OnStorageWrite()
{
	// TODO: Add your command handler code here
	CWinThread* pThread = AfxBeginThread(WriteThreadProc, GetSafeHwnd());
}
```

```
// CMymfc31aView message handlers

void CMymfc31aView::OnStorageRead()
{
    // TODO: Add your command handler code here
    CWinThread* pThread = AfxBeginThread(ReadThreadProc, GetSafeHwnd());
}

void CMymfc31aView::OnStorageWrite()
{
    // TODO: Add your command handler code here
    CWinThread* pThread = AfxBeginThread(WriteThreadProc, GetSafeHwnd());
}
```

Listing 1.

Add the `#include` directive to **mymfc31aView.cpp**.

```
#include "Thread.h"
```

```
#include "stdafx.h"
#include "mymfc31a.h"

#include "mymfc31aDoc.h"
#include "mymfc31aView.h"
#include "Thread.h"
```

Listing 2.

Add files to the project, copy and paste the codes (the codes are provided in the following Listings). Add Thread.h file.

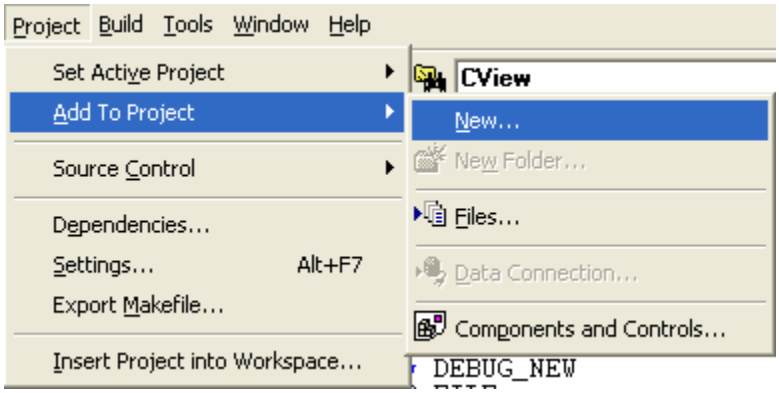

Figure 14: Adding new files for new class to the project.

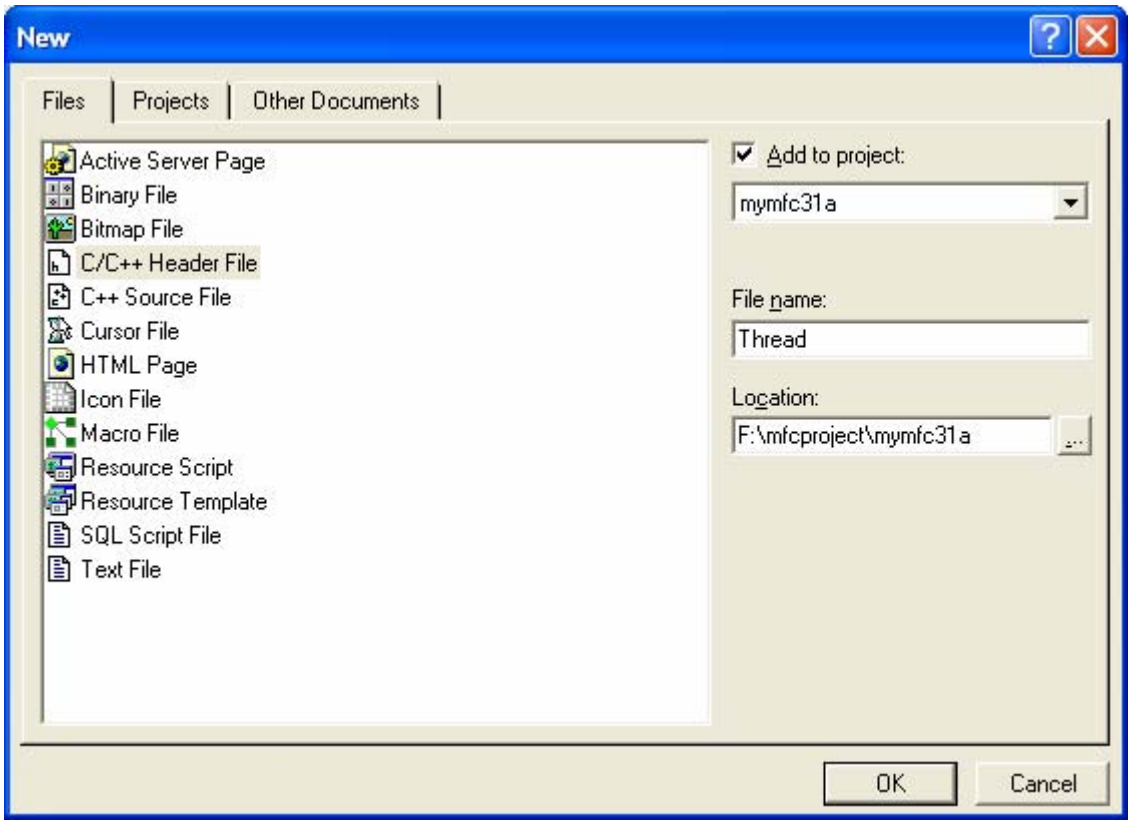

Figure 15: Adding **Thread.h** header file to project.

Repeat the same step for WriteThread.cpp and ReadThread.cpp source files (of course you have to select the **C++ Source File** in the previous Figure), copy and paste the codes to those files.

Add the following lines in your **StdAfx.h** file:

```
#include <afxole.h>
#include <afxpriv.h> // for wide-character conversion
```

Then delete the following line:

```
#define VC_EXTRALEAN
```

```
// #define VC_EXTRALEAN     // Exclude rarely-used stu

#include <afxwin.h>         // MFC core and standard c
#include <afxext.h>         // MFC extensions
#include <afxdtctl.h>       // MFC support for Interne
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>         // MFC support for Windows
#endif // _AFX_NO_AFXCMN_SUPPORT

#include <afxole.h>
#include <afxpriv.h> // for wide-character conversion

//{{AFX_INSERT_LOCATION}}
```

Listing 3.

Build and run in debug mode. Click the **Write** menu.

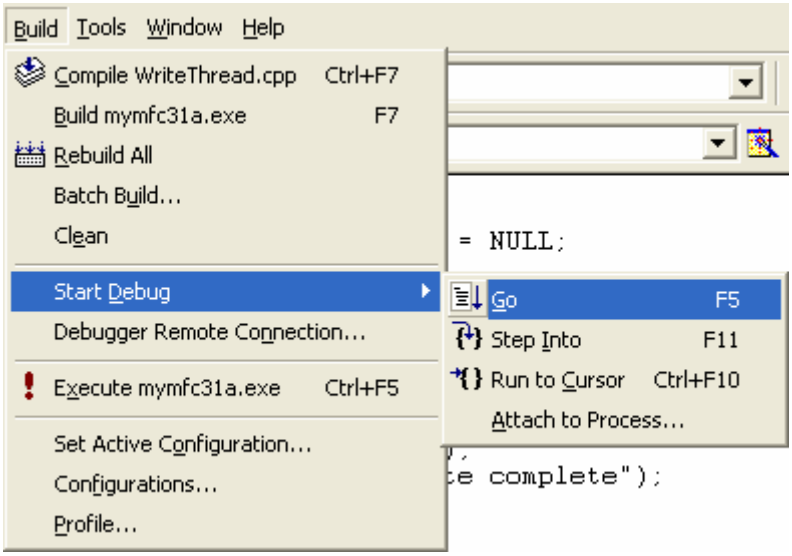

Figure 16: Running MYMFC31A in debug mode.

Select the Write menu.

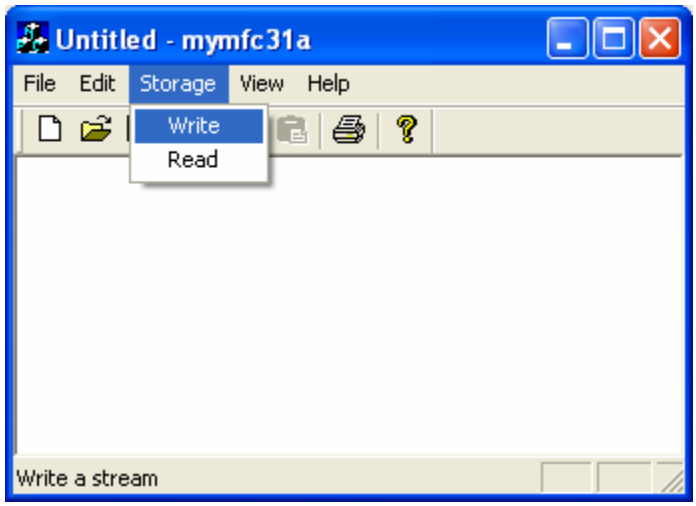

Figure 17: MYMFC31A output.

The message box displayed when **Write** menu is selected as shown below.

Figure 18: Message box displayed when **Write** menu was selected.

Then, click the **Read** menu.

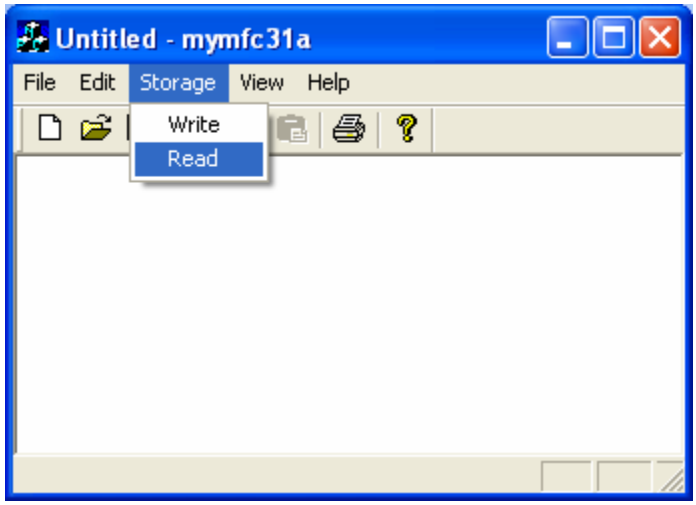

Figure 19: Testing the **Read** menu.

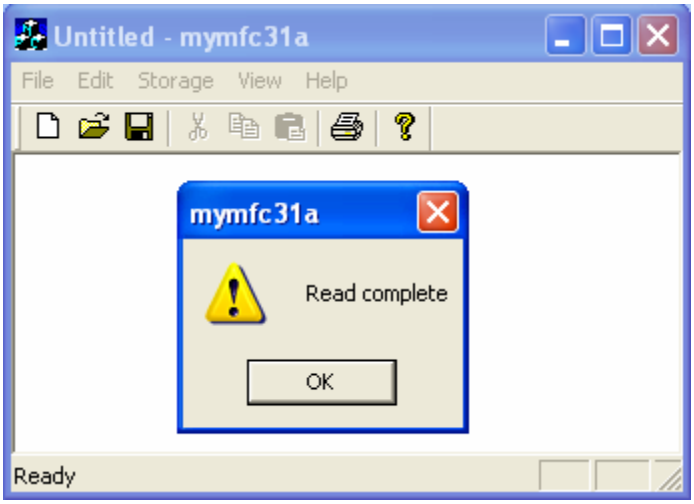

Figure 20: Message box displayed when **Read** menu was selected.

Check the **Debug** window. It will look something like this (the storage and stream should be different with yours).

```
Storage = mfcjadik
    Storage = final
        Stream = \mfcjadik\final\log.txt


    Storage = myscribble
        Storage = Debug
        Stream = \mfcjadik\myscribble\ReadMe.txt
===================================================================
...
[trimmed]
...
===================================================================

    Storage = mymfc27B
        Storage = res
        Storage = Debug
        Stream = ReadMe.txt
===================================================================

The thread 0x398 has exited with code 0 (0x0).
The thread 0xD04 has exited with code 0 (0x0).
The program 'F:\mfcproject\mymfc31a\Debug\mymfc31a.exe' has exited with code 0 (0x0).
```

Verify the **direct.stg** file creation.

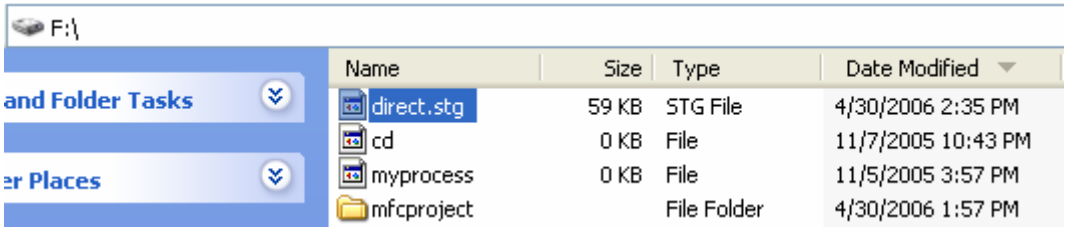

Figure 21: Verifying the **direct.stg** file creation by the write operation in the previous step.

The Menu

The MYMFC31A example has an added top-level **Storage** menu with **Write** and **Read** options.

The `CMymfc31aView` Class

This class maps the new **Storage Read** and **Write** menu commands listed above to start worker threads. The handlers are shown here:

```
void CMymfc31aView::OnStorageRead()
{
    CWinThread* pThread = AfxBeginThread(ReadThreadProc, GetSafeHwnd());
}

void CMymfc31aView::OnStorageWrite()
{
    CWinThread* pThread = AfxBeginThread(WriteThreadProc, GetSafeHwnd());
}
```

The Worker Threads

Listing 4 lists the code for the **Storage Write** and **Storage Read** worker threads.

THREAD.H

```
// Thread.h (MYMFC31A)

extern int g_nIndent;
extern const char* g_szBlanks;
extern const char* g_szRootStorageName;

UINT WriteThreadProc(LPVOID pParam);
UINT ReadThreadProc(LPVOID pParam);
void ReadDirectory(const char* szPath, LPSTORAGE pStg);
void ReadStorage(LPSTORAGE pStg);
```

WRITETHREAD.CPP

```
// WriteThread.cpp (MYMFC31A)

#include "StdAfx.h"
#include "Thread.h"

int g_nIndent = 0;
const char* g_szBlanks = "                  ";
// the file for storage...
const char* g_szRootStorageName = "\\direct.stg";


UINT WriteThreadProc(LPVOID pParam)
{
    USES_CONVERSION;
    LPSTORAGE pStgRoot = NULL;
    g_nIndent = 0;
    VERIFY(::StgCreateDocfile(T2COLE(g_szRootStorageName),
           STGM_READWRITE | STGM_SHARE_EXCLUSIVE | STGM_CREATE,
           0, &pStgRoot) == S_OK);
    // you can start at any root...or directory...
    // if just "\\", current root will be used...
    ReadDirectory("\\", pStgRoot);
    pStgRoot->Release();
    AfxMessageBox("Write complete");
    return 0;
}

void ReadDirectory(const char* szPath, LPSTORAGE pStg)
{
    // recursive function
    USES_CONVERSION;
    WIN32_FIND_DATA fData;
```

```
    HANDLE h;
    char szNewPath[MAX_PATH];
    char szStorageName[10000];
    char szStreamName[10000];
       char szData[10001];
    char* pch = NULL;

    LPSTORAGE pSubStg = NULL;
    LPSTREAM pStream = NULL;

    g_nIndent++;
    strcpy(szNewPath, szPath);
    strcat(szNewPath, "*.*");
    h = ::FindFirstFile(szNewPath, &fData);
    if (h == (HANDLE) 0xFFFFFFFF) return;  // can't find directory
    do {
      if (!strcmp(fData.cFileName, "..") ||
          !strcmp(fData.cFileName, ".")) continue;
         while((pch = strchr(fData.cFileName, '!')) != NULL) {
             *pch = '|';
         }
      if (fData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) {
         // It's a directory, so make a storage
        strcpy(szNewPath, szPath);
        strcat(szNewPath,fData.cFileName);
        strcat(szNewPath, "\\");

        strcpy(szStorageName, fData.cFileName);
        szStorageName[31] = '\0';    // limit imposed by OLE
        TRACE("%0.*sStorage = %s\n", (g_nIndent - 1) * 4, g_szBlanks, szStorageName);
        VERIFY(pStg->CreateStorage(T2COLE(szStorageName),
            STGM_CREATE | STGM_READWRITE | STGM_SHARE_EXCLUSIVE,
            0, 0, &pSubStg) == S_OK);
        ASSERT(pSubStg != NULL);
        ReadDirectory(szNewPath, pSubStg);
        pSubStg->Release();
      }
      else {
        if ((pch = strrchr(fData.cFileName, '.')) != NULL)
             {
          if (!stricmp(pch, ".TXT")) {
             // It's a text file, so make a stream
             strcpy(szStreamName, fData.cFileName);
            strcpy(szNewPath, szPath);
            strcat(szNewPath, szStreamName);
            szStreamName[32] = '\0'; // OLE max length
            TRACE("%0.*sStream = %s\n", (g_nIndent - 1) * 4, g_szBlanks, szNewPath);
              CStdioFile file(szNewPath, CFile::modeRead);
            // Ignore zero-length files
                     if(file.ReadString(szData, 10000))
                     {
              TRACE("%s\n", szData);
                       // For a very 'big' file number the Write() will trigger the
                       // Exception - Access violation, just click the ignore
                       // button...
                       // To avoid the Exception, memory allocation must be properly
                       // done...T2COLE do the string conversion...hohoho...
                           VERIFY(pStg->CreateStream(T2COLE(szStreamName),
                                                     STGM_CREATE | STGM_READWRITE |
                                                     STGM_SHARE_EXCLUSIVE,
                                                     0, 0, &pStream) == S_OK);
                           ASSERT(pStream != NULL);
                           // Include the null terminator in the stream
                           pStream->Write(szData, strlen(szData) + 1, NULL);
                           pStream->Release();

                  // ---This is another snippet version----
                   // if(pStg->CreateStream(T2COLE(szStreamName),
```

```
                    //   STGM_CREATE | STGM_READWRITE |
                     //   STGM_SHARE_EXCLUSIVE,
                    //   0, 0, &pStream) == S_OK)
                                // {
                                //   ASSERT(pStream != NULL);
                                // // Include the null terminator in the stream
                                //   pStream->Write(szData, strlen(szData) + 1, NULL);
                                //   pStream->Release();
                                // }
            }
          }
        }
      }
    } while (::FindNextFile(h, &fData));
    g_nIndent--;
}
```

READTHREAD.CPP

```
// ReadThread.cpp (MYMFC31A)

#include "StdAfx.h"
#include "Thread.h"

UINT ReadThreadProc(LPVOID pParam)
{
    USES_CONVERSION;
    LPSTORAGE pStgRoot = NULL;
    // doesn't work without STGM_SHARE_EXCLUSIVE
    g_nIndent = 0;
    if (::StgOpenStorage(T2COLE(g_szRootStorageName), NULL,
        STGM_READ | STGM_SHARE_EXCLUSIVE,
        NULL, 0, &pStgRoot) == S_OK) {
      ASSERT(pStgRoot!= NULL);
      ReadStorage(pStgRoot);
      pStgRoot->Release();
    }
    else {
      AfxMessageBox("Storage file not available or not readable");
    }
    AfxMessageBox("Read complete");
    return 0;
}

void ReadStorage(LPSTORAGE pStg)
// reads one storage -- recursive calls for substorages
{
    USES_CONVERSION;
    LPSTORAGE pSubStg = NULL;
    LPSTREAM pStream = NULL;
    LPENUMSTATSTG pEnum = NULL;
    LPMALLOC pMalloc = NULL; // for freeing statstg
    STATSTG statstg;
    ULONG nLength;
    BYTE buffer[101];

    g_nIndent++;
    ::CoGetMalloc(MEMCTX_TASK, &pMalloc); // assumes AfxOleInit
                                          //  was called
    VERIFY(pStg->EnumElements(0, NULL, 0, &pEnum) == S_OK);
    while (pEnum->Next(1, &statstg, NULL) == S_OK) {
      if (statstg.type == STGTY_STORAGE) {
        VERIFY(pStg->OpenStorage(statstg.pwcsName, NULL,
               STGM_READ | STGM_SHARE_EXCLUSIVE,
               NULL, 0, &pSubStg) == S_OK);
        ASSERT(pSubStg != NULL);
        TRACE("%0.*sStorage = %s\n", (g_nIndent - 1) * 4,
              g_szBlanks, OLE2CT(statstg.pwcsName));
```

```
        ReadStorage(pSubStg);
        pSubStg->Release();
      }
      else if (statstg.type == STGTY_STREAM) {
        VERIFY(pStg->OpenStream(statstg.pwcsName, NULL,
               STGM_READ | STGM_SHARE_EXCLUSIVE,
               0, &pStream) == S_OK);
        ASSERT(pStream != NULL);
        TRACE("%0.*sStream = %s\n", (g_nIndent - 1) * 4,
               g_szBlanks, OLE2CT(statstg.pwcsName));
        pStream->Read(buffer, 100, &nLength);
        buffer[nLength] = '\0';
        TRACE("%s\n", buffer);
        pStream->Release();
      }
      else {
        ASSERT(FALSE);  // LockBytes?
      }
      pMalloc->Free(statstg.pwcsName); // avoids memory leaks
    }
    pMalloc->Release();
    pEnum->Release();
    g_nIndent--;
}
```

Listing 4: The **Storage** menu worker threads.

To keep the program simple, there's no synchronization between the main thread and the two worker threads. You could run both threads at the same time if you used two separate compound files.
From your study of the Win32 threading model, you might expect that closing the main window would cause the read thread or write thread to terminate "midstream," possibly causing **memory leaks**. But this does not happen because MFC senses that the worker threads are using COM objects. Even though the window closes immediately, the program does not exit until all threads exit.
Both threads use recursive functions. The `ReadStorage()` function reads a storage and calls itself to read the substorages. The `ReadDirectory()` function reads a directory and calls itself to read the subdirectories. This function calls the Win32 functions `FindFirstFile()` and `FindNextFile()` to iterate through the elements in a directory. The `dwFileAttributes` member of the `WIN32_FIND_DATA` structure indicates whether the element is a file or a subdirectory. `ReadDirectory()` uses the MFC `CStdioFile` class because the class is ideal for reading text.
The `USES_CONVERSION` macro is necessary to support the **wide-character conversion** macros `OLE2CT` and `T2COLE`. These macros are used here because the example doesn't use the `CString` class, which has built-in conversion logic.

Structured Storage and Persistent COM Objects

The MYMFC31A program explicitly called member functions of `IStorage` and `IStream` to write and read a compound file. In the object-oriented world, objects should know how to save and load themselves to and from a compound file. That's what the `IPersistStorage` and `IPersistStream` interfaces are for. If a COM component implements these interfaces, a container program can "connect" the object to a compound file by passing the file's `IStorage` pointer as a parameter to the `Save()` and `Load()` member functions of the `IPersistStorage` interface. Such objects are said to be persistent. Figure 22 shows the process of calling the `IPersistStorage::Save` function.
A COM component is more likely to work with an `IStorage` interface than an `IStream` interface. If the COM object is associated with a particular storage, the COM component can manage substorages and streams under that storage once it gets the `IStorage` pointer. A COM component uses the `IStream` interface only if it stores all its data in an array of bytes. **ActiveX controls** implement the `IStream` interface for storing and loading property values.

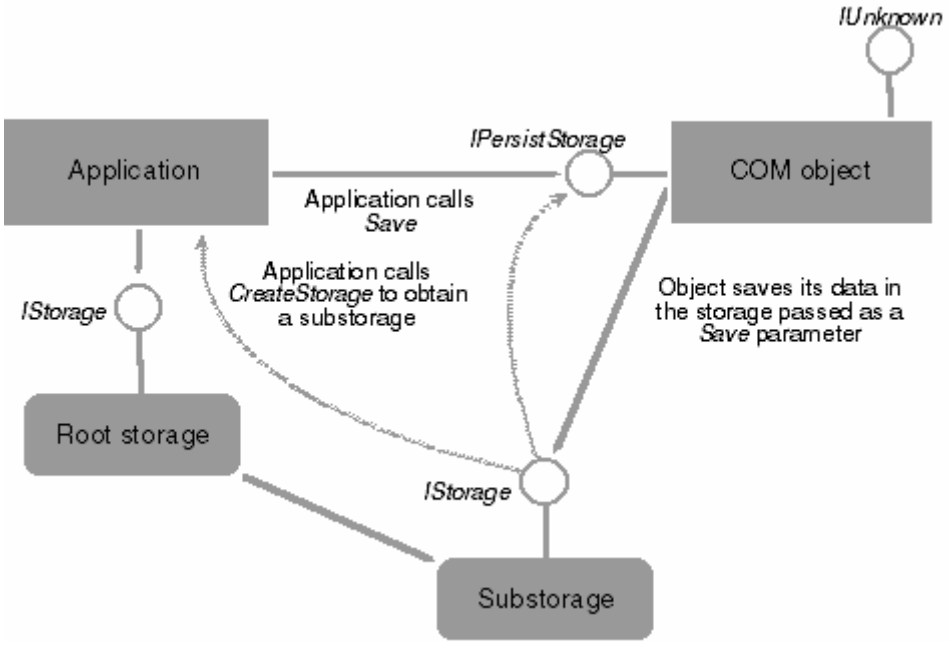

Figure 22: Calling `IPersistStorage::Save`.

The `IPersistStorage` Interface

When an OLE container creates a new object, loads an existing object from storage, or inserts a new object in a clipboard or a drag-and-drop operation, the container uses the `IPersistStorage` interface to initialize the object and put it in the loaded or running state. When an object is loaded or running, an OLE container calls other `IPersistStorage` methods to instruct the object to perform various save operations or to release its storage. Typically, applications use helper functions such as `OleLoad()` or `OleCreate()`, rather than calling the `IPersistStorage::Load` or `IPersistStorage::InitNew` methods directly. Similarly, applications typically call the `OleSave()` helper function rather than calling `IPersistStorage::Save` directly. Methods/functions in `IPersistStorage` is listed below.

IUnknown Methods	Description
`QueryInterface()`	Returns pointers to supported interfaces.
`AddRef()`	Increments reference count.
`Release()`	Decrements reference count.

Table 6.

IPersist Method	Description
`GetClassID()`	Returns the class identifier (`CLSID`) for the object on which it is implemented.

Table 7.

IPersistStorage Methods	Description
`IsDirty()`	Indicates whether the object has changed since it was last saved to its current storage.
`InitNew()`	Initializes a new storage object.
`Load()`	Initializes an object from its existing storage.
`Save()`	Saves an object, and any nested objects that it contains, into the specified storage object. The object enters

	NoScribble mode.
`SaveCompleted()`	Notifies the object that it can revert from **NoScribble** or **HandsOff** mode, in which in must not write to its storage object, to **Normal** mode in which it can.
`HandsOffStorage()`	Instructs the object to release all storage objects that have been passed to it by its container and to enter **HandsOffAfterSave** or **HandsOffFromNormal** mode.

Table 8.

Both the `IPersistStorage` and `IPersistStream` interfaces are derived from `IPersist`, which contributes the `GetClassID()` member function. Of course the `IUnknown` member function also included. Here's a summary of the important `IPersistStorage` member functions in detail.

```
HRESULT GetClassID(CLSID* pClsid);
```

Returns the COM component's 128-bit class identifier.

```
HRESULT InitNew(IStorage* pStg);
```

Initializes a newly created object. The component might need to use the storage for temporary data, so the container must provide an `IStorage` pointer that's valid for the life of the object. The component should call `AddRef()` if it intends to use the storage. The component should not use this `IStorage` pointer for saving and loading; it should wait for `Save()` and `Load()` calls and then use the passed-in `IStorage` pointer to call `IStorage::Write` and `Read()`.

```
HRESULT IsDirty(void);
```

Returns `S_OK` if the object has changed since it was last saved; otherwise, returns `S_FALSE`.

```
HRESULT Load(IStorage* pStg);
```

Loads the COM object's data from the designated storage.

```
HRESULT Save(IStorage* pStg, BOOL fSameAsLoad);
```

Saves the COM object's data in the designated storage.

The IPersistStream Interface

Here's a summary of the `IPersistStream` member functions:

```
HRESULT GetClassID(CLSID* pClsid);
```

Returns the COM component's 128-bit class identifier.

```
HRESULT GetMaxSize(ULARGE_INTEGER* pcbSize);
```

Returns the number of bytes needed to save the object.

```
HRESULT IsDirty(void);
```

Returns `S_OK` if the object has changed since it was last saved; otherwise, returns `S_FALSE`.

```
HRESULT Load(IStream* pStm);
```

Loads the COM object's data from the designated stream.

```
HRESULT Save(IStream* pStm, BOOL fClearDirty);
```

Saves the COM object's data to the designated stream. If the `fClearDirty` parameter is `TRUE`, Save clears the object's dirty flag.

`IPersistStream`

The `IPersistStream` interface provides methods for saving and loading objects that use a simple serial stream for their storage needs. The `IPersistStream` interface inherits its definition from the `IPersist` interface, and so includes the `GetClassID` method of IPersist. Call methods of `IPersistStream` from a container application to save or load objects that are contained in a simple stream. When used to save or load monikers, typical applications do not call the methods directly, but allow the default link handler to make the calls to save and load the monikers that identify the link source. These monikers are stored in a stream in the storage for the linked object. If you are writing a custom link handler for your class of objects, you would call the methods of `IPersistStream` to implement the link handler. Methods/functions available for `IPersistStream` are listed below.

IUnknown Methods	Description
`QueryInterface()`	Returns pointers to supported interfaces.
`AddRef()`	Increments the reference count.
`Release()`	Decrements the reference count.

Table 9.

IPersist Method	Description
`GetClassID()`	Returns the class identifier (`CLSID`) for the component object.

Table 10.

IPersistStream Methods	Description
`IsDirty()`	Checks the object for changes since it was last saved.
`Load()`	Initializes an object from the stream where it was previously saved.
`Save()`	Saves an object into the specified stream and indicates whether the object should reset its dirty flag.
`GetSizeMax()`	Return the size in bytes of the stream needed to save the object.

Table 11.

`IPersistStream` Programming

The following container program code fragment creates a stream and saves a COM object's data in it. Both the `IPersistStream` pointer for the COM object and the `IStorage` pointer are set elsewhere.

```
extern IStorage* pStg;
extern IPersistStream* pPersistStream;
IStream* pStream;
if (pStg->CreateStream(L"MyStreamName",
    STGM_CREATE | STGM_READWRITE | STGM_SHARE_EXCLUSIVE,
    0, 0, &pStream) == S_OK)
{
    ASSERT(pStream != NULL);
    pPersistStream->Save(pStream, TRUE);
    pStream->Release();
}
```

If you program your own COM class for use in a container, you'll need to use the MFC interface macros to add the `IPersistStream` interface. Too bad there's not an "interface wizard" to do the job.

The MYMFC31B Example: A Persistent DLL Component

The MYMFC31B program, which is used by MYMFC31C, is a COM DLL that contains the `CText` component. This is a simple COM class that implements the `IDispatch` and `IPersistStream` interfaces. The `IDispatch` interface allows access to the component's one and only property, `Text`, and the `IPersistStream` interface allows an object to save and load that Text property to and from a structured storage file.
To prepare MYMFC31B, open the **mymfc31b.dsw** workspace and build the project. Use **regsvr32** to register the DLL. Listing 5 lists the code for the `CText` class in **Text.h** and **Text.cpp**.

TEXT.H

```
#if !defined(AFX_TEXT_H__14635AFC_DF23_4875_B984_BFC39089C60F__INCLUDED_)
#define AFX_TEXT_H__14635AFC_DF23_4875_B984_BFC39089C60F__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// Text.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// CText command target

class CText : public CCmdTarget
{
	DECLARE_DYNCREATE(CText)

	CText();	// protected constructor used by dynamic creation

// Attributes
public:

// Operations
public:

// Overrides
	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(CText)
	public:
	virtual void OnFinalRelease();
	//}}AFX_VIRTUAL

// Implementation
protected:
	virtual ~CText();

	// Generated message map functions
	//{{AFX_MSG(CText)
		// NOTE - the ClassWizard will add and remove member functions here.
	//}}AFX_MSG

	DECLARE_MESSAGE_MAP()
	DECLARE_OLECREATE(CText)

	// Generated OLE dispatch map functions
	//{{AFX_DISPATCH(CText)
	afx_msg VARIANT GetText();
	afx_msg void SetText(const VARIANT FAR& newValue);
	//}}AFX_DISPATCH
	DECLARE_DISPATCH_MAP()
	DECLARE_INTERFACE_MAP()

	BEGIN_INTERFACE_PART(PersistStream, IPersistStream)
		STDMETHOD(GetClassID)(LPCLSID);
		STDMETHOD(IsDirty)();
```

```
            STDMETHOD(Load)(LPSTREAM);
            STDMETHOD(Save)(LPSTREAM, BOOL);
            STDMETHOD(GetSizeMax)(ULARGE_INTEGER FAR*);
        END_INTERFACE_PART(PersistStream)

private:
        char* m_pchText;

};

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

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

#endif // !defined(AFX_TEXT_H__14635AFC_DF23_4875_B984_BFC39089C60F__INCLUDED_)
```

TEXT.CPP

```
// Text.cpp : implementation file
//

#include "stdafx.h"
#include "mymfc31b.h"
#include "Text.h"

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

/////////////////////////////////////////////////////////////////////////////
// CText

IMPLEMENT_DYNCREATE(CText, CCmdTarget)

CText::CText()
{
        EnableAutomation();

        // To keep the application running as long as an OLE automation
        //     object is active, the constructor calls AfxOleLockApp.

        AfxOleLockApp();
}

CText::~CText()
{
        // To terminate the application when all objects created with
        //     with OLE automation, the destructor calls AfxOleUnlockApp.

        AfxOleUnlockApp();
}

void CText::OnFinalRelease()
{
        // When the last reference for an automation object is released
        // OnFinalRelease is called.  The base class will automatically
        // deletes the object.  Add additional cleanup required for your
        // object before calling the base class.

        CCmdTarget::OnFinalRelease();
}

BEGIN_MESSAGE_MAP(CText, CCmdTarget)
        //{{AFX_MSG_MAP(CText)
                // NOTE - the ClassWizard will add and remove mapping macros here.
```

```
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()

BEGIN_DISPATCH_MAP(CText, CCmdTarget)
	//{{AFX_DISPATCH_MAP(CText)
	DISP_PROPERTY_EX(CText, "Text", GetText, SetText, VT_VARIANT)
	//}}AFX_DISPATCH_MAP
END_DISPATCH_MAP()

// Note: we add support for IID_IText to support typesafe binding
//  from VBA.  This IID must match the GUID that is attached to the
//  dispinterface in the .ODL file.

// {84D30689-5963-400D-8D4C-19CA756679B1}
static const IID IID_IText =
{ 0x84d30689, 0x5963, 0x400d, { 0x8d, 0x4c, 0x19, 0xca, 0x75, 0x66, 0x79, 0xb1 }
};

BEGIN_INTERFACE_MAP(CText, CCmdTarget)
	INTERFACE_PART(CText, IID_IPersistStream, PersistStream)
	INTERFACE_PART(CText, IID_IText, Dispatch)
END_INTERFACE_MAP()

// {B34965D8-9295-45E9-A4C6-EC0B51EBD1BD}
IMPLEMENT_OLECREATE(CText, "mymfc31b.Text", 0xb34965d8, 0x9295, 0x45e9, 0xa4,
0xc6, 0xec, 0xb, 0x51, 0xeb, 0xd1, 0xbd)

/////////////////////////////////////////////////////////////////////////////
// CText message handlers

VARIANT CText::GetText()
{
	return COleVariant(m_pchText).Detach();
}

void CText::SetText(const VARIANT FAR& newValue)
{
	// TODO: Add your property handler here
	CString strTemp;
	ASSERT(newValue.vt == VT_BSTR);
	if(m_pchText != NULL) {
		delete [] m_pchText;
	}
	strTemp = newValue.bstrVal; // converts to narrow chars
	m_pchText = new char[strTemp.GetLength() + 1];
	strcpy(m_pchText, strTemp);
}

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

STDMETHODIMP_(ULONG) CText::XPersistStream::AddRef()
{
	METHOD_PROLOGUE(CText, PersistStream)
	return (ULONG) pThis->ExternalAddRef();
}

STDMETHODIMP_(ULONG) CText::XPersistStream::Release()
{
	METHOD_PROLOGUE(CText, PersistStream)
	return (ULONG) pThis->ExternalRelease();
}

STDMETHODIMP CText::XPersistStream::QueryInterface(REFIID iid,
		void FAR* FAR* ppvObj)
{
	METHOD_PROLOGUE(CText, PersistStream)
	// ExternalQueryInterface looks up iid in the macro-generated tables
	return (HRESULT) pThis->ExternalQueryInterface(&iid, ppvObj);
```

```
}

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

STDMETHODIMP CText::XPersistStream::GetClassID(LPCLSID lpClassID)
{
	TRACE("Entering CText::XPersistStream::GetClassID\n");
	METHOD_PROLOGUE(CText, PersistStream)
	ASSERT_VALID(pThis);

	*lpClassID = CText::guid;
	return NOERROR;
}

STDMETHODIMP CText::XPersistStream::IsDirty()
{
	TRACE("Entering CText::XPersistStream::IsDirty\n");
	METHOD_PROLOGUE(CText, PersistStream)
	ASSERT_VALID(pThis);

	return NOERROR;
}

STDMETHODIMP CText::XPersistStream::Load(LPSTREAM pStm)
{
	ULONG nLength;
	STATSTG statstg;

	METHOD_PROLOGUE(CText, PersistStream)
	ASSERT_VALID(pThis);
	if(pThis->m_pchText != NULL) {
		delete [] pThis->m_pchText;
	}
	// don't need to free statstg.pwcsName because of NONAME flag
	VERIFY(pStm->Stat(&statstg, STATFLAG_NONAME) == NOERROR);
	int nSize = statstg.cbSize.LowPart; // assume < 4 GB
	if(nSize > 0) {
		pThis->m_pchText = new char[nSize];
		pStm->Read(pThis->m_pchText, nSize, &nLength);
	}
	return NOERROR;
}

STDMETHODIMP CText::XPersistStream::Save(LPSTREAM pStm, BOOL fClearDirty)
{
	METHOD_PROLOGUE(CText, PersistStream)
	ASSERT_VALID(pThis);
	int nSize = strlen(pThis->m_pchText) + 1;
	pStm->Write(pThis->m_pchText, nSize, NULL);
	return NOERROR;
}

STDMETHODIMP CText::XPersistStream::GetSizeMax(ULARGE_INTEGER FAR* pcbSize)
{
	TRACE("Entering CText::XPersistStream::GetSizeMax\n");
	METHOD_PROLOGUE(CText, PersistStream)
	ASSERT_VALID(pThis);
	pcbSize->LowPart = strlen(pThis->m_pchText) + 1;
	pcbSize->HighPart = 0; // assume < 4 GB
	return NOERROR;
}
```

Listing 5: The code listing for the `CText` class in **Text.h** and **Text.cpp**.

The MYMFC31B From Scratch

This is DLL application with Automation support.

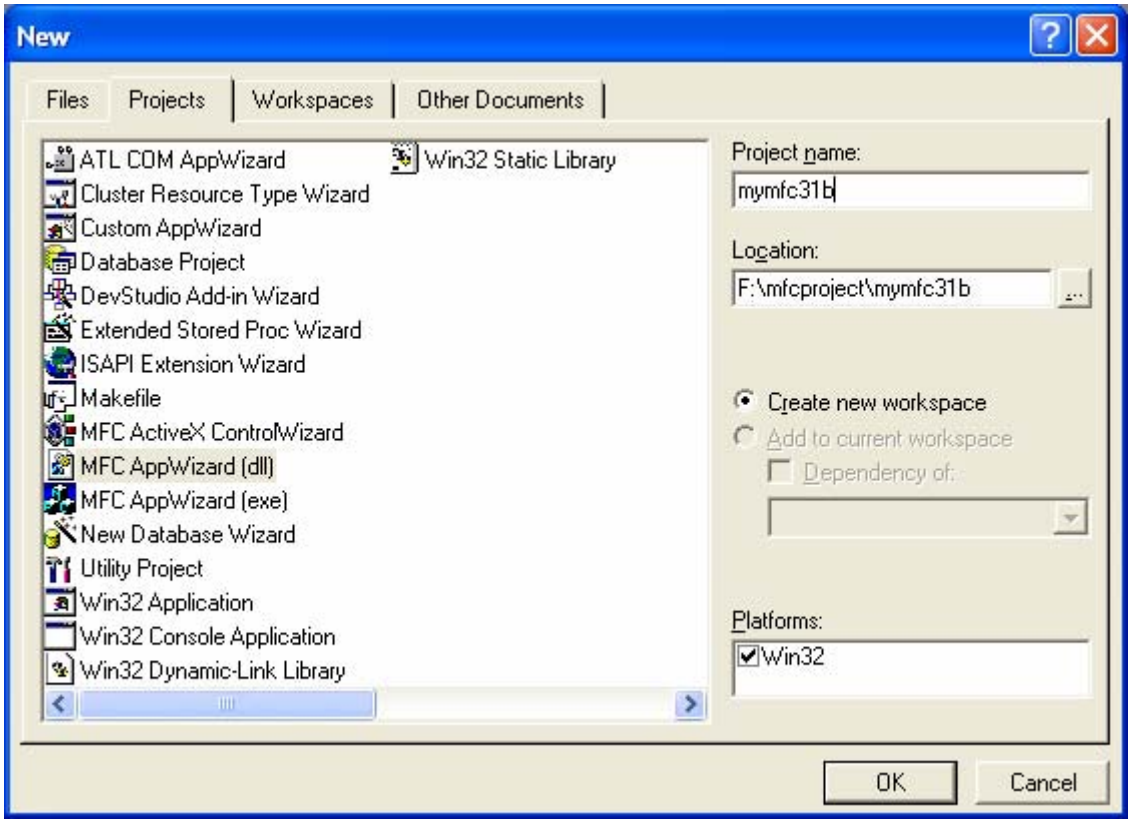

Figure 23: MYMFC31B – Visual C++ new project dialog, select **MFC AppWizard (dll)** project.

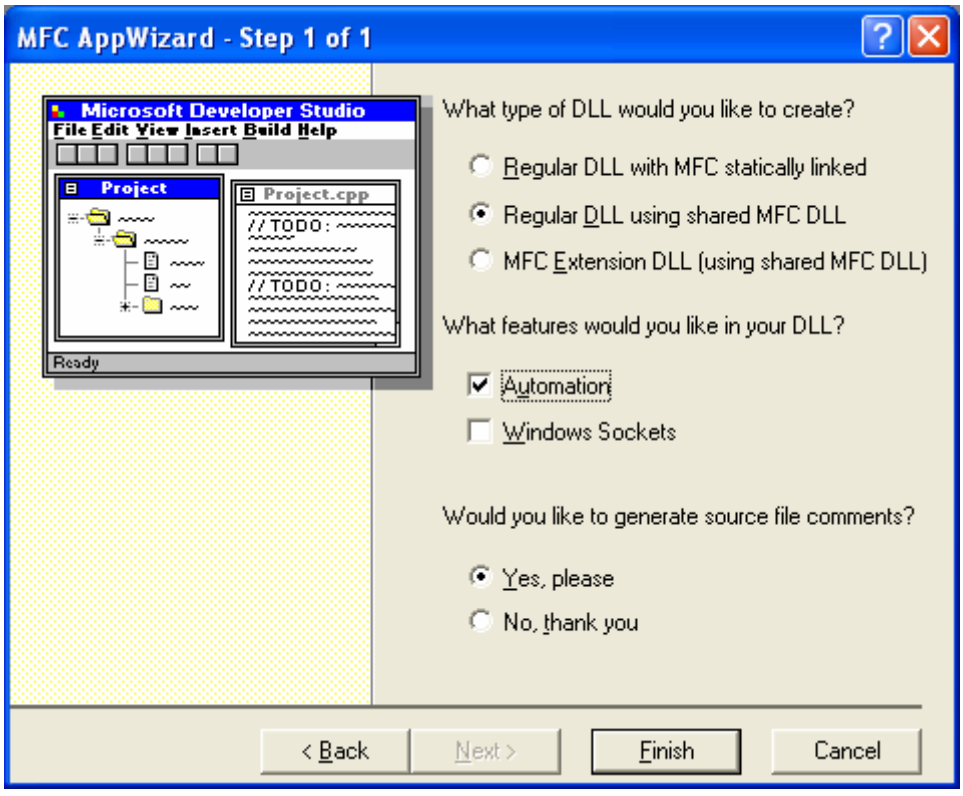

Figure 24: MYMFC31B – AppWizard step 1 of 1, select the **Automation** option.

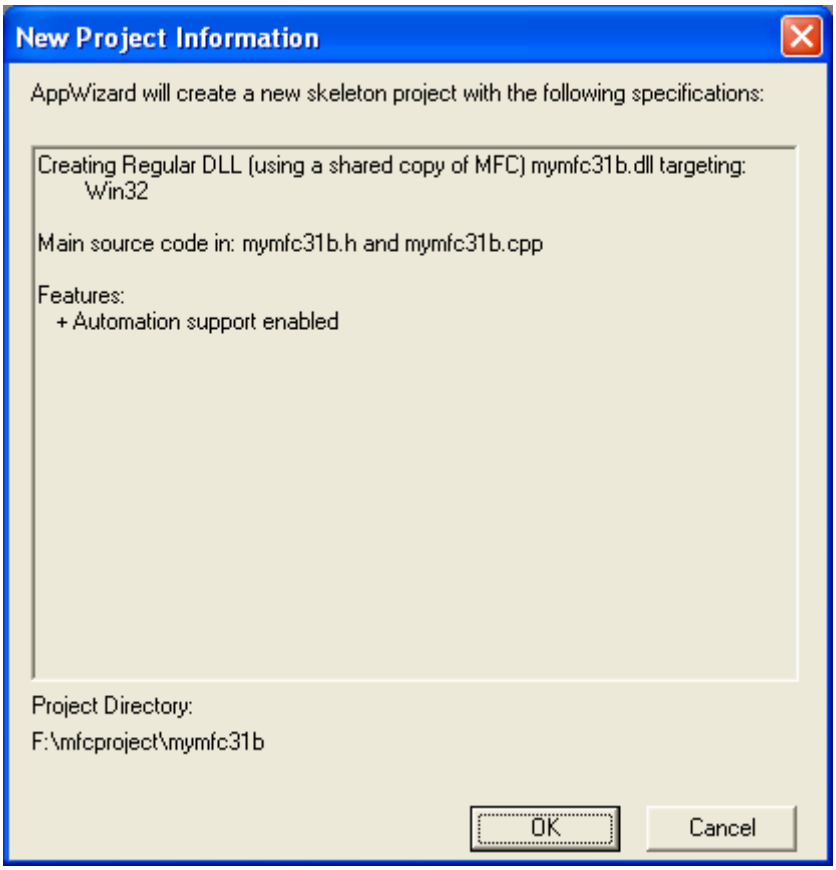

Figure 1: MYMFC31B project summary.

Add new class using ClassWizard. Click the **Add Class** button.

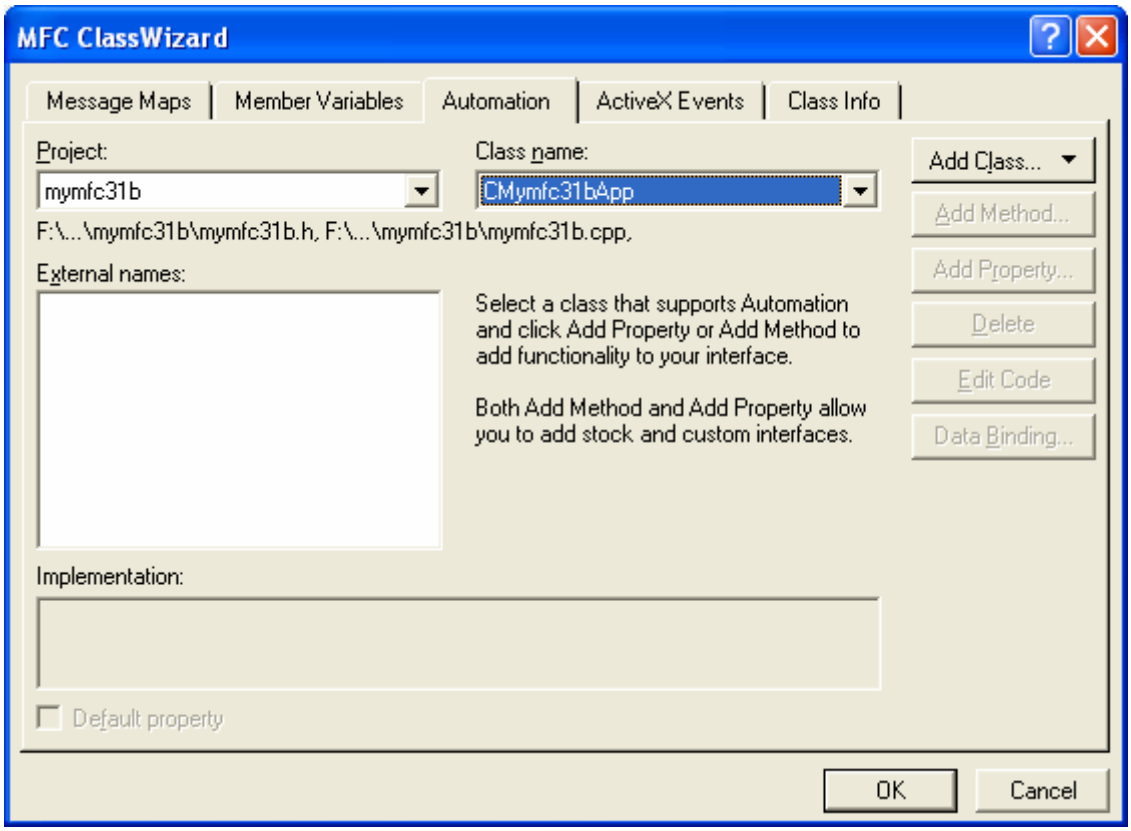

Figure 25: Adding new class using ClassWizard.

Fill up the `CText` class information and don't forget to select **Creatable by type ID** option.

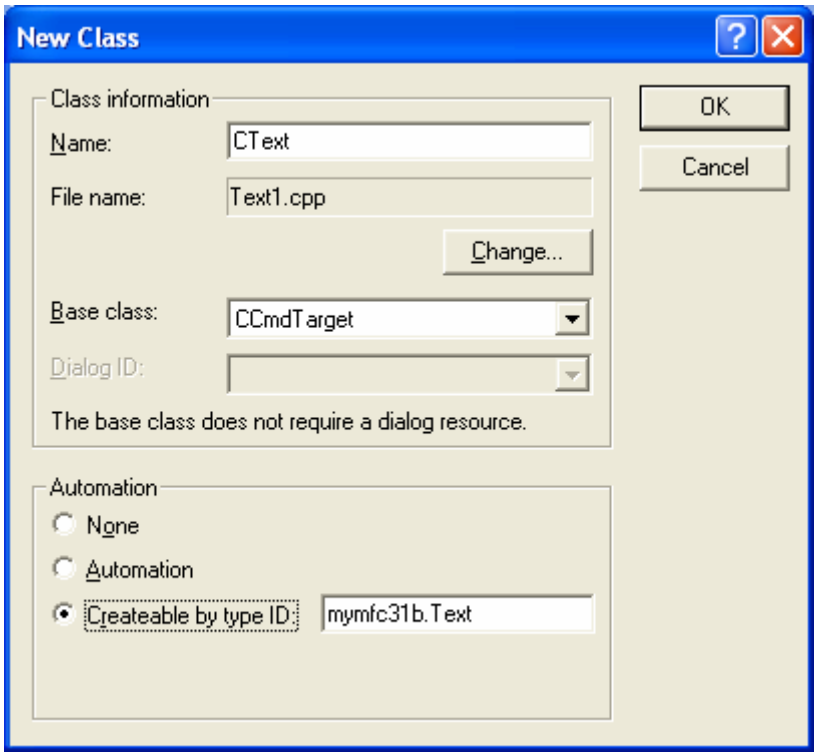

Figure 26: Entering `CText` class information.

Add/override `OnFinalRelease()` function to `CMymfc31bApp` class.

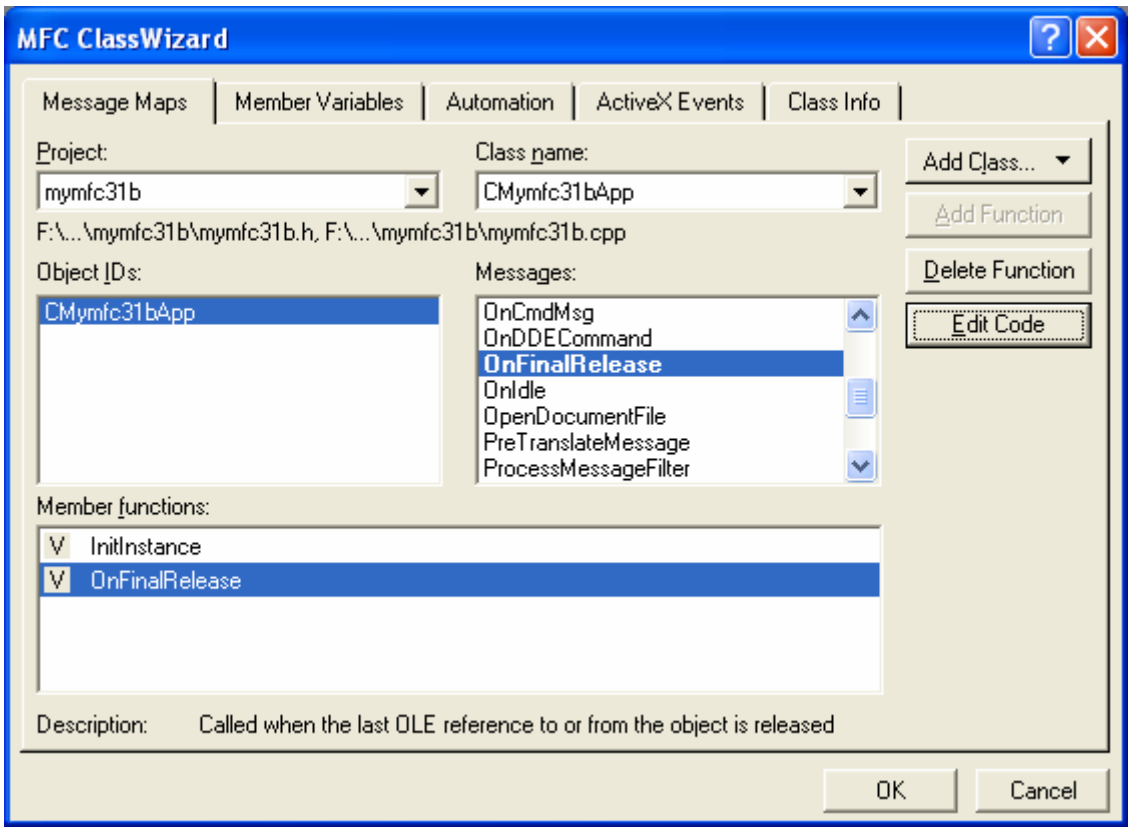

Figure 27: Adding `OnFinalRelease()` function to `CMymfc31bApp` class.

Add property to `CText` class. In the **Automation** page of ClassWizard, click the **Add Property** button. Make sure `CText` class is selected in the **Class name** combo box. Fill up the following information. This is a Get/Set method.

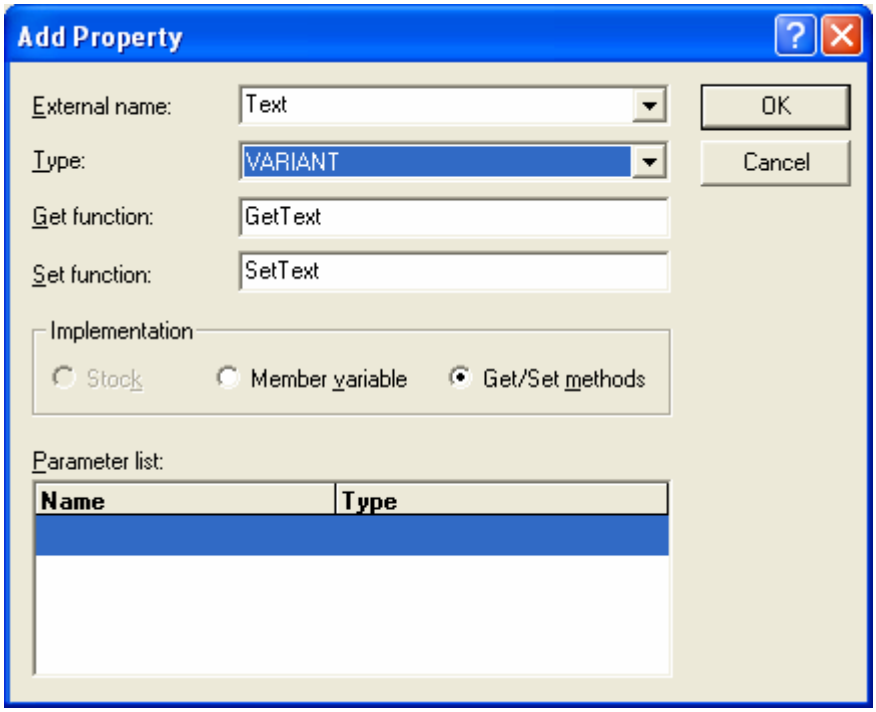

Figure 28: Adding `Text` property to `CText` class.

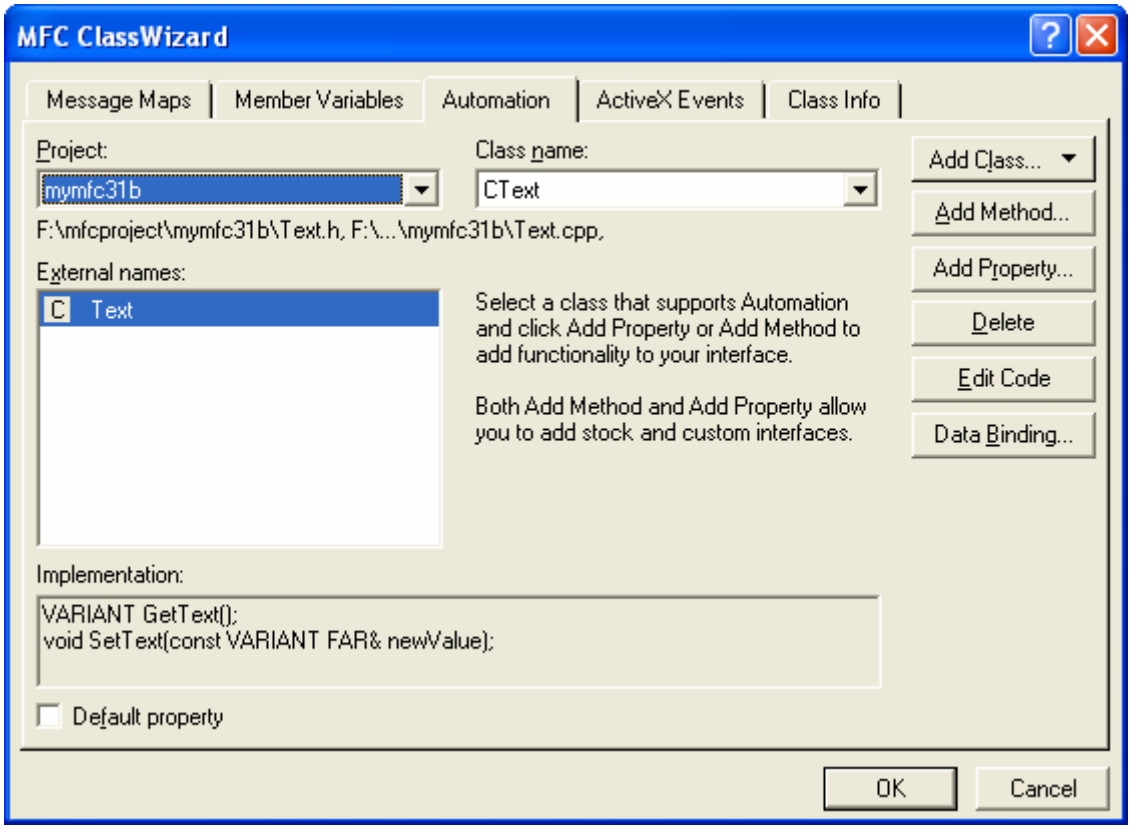

Figure 29: The added `Text` property.

Add a private variable to `CText` class as shown below.

```
private:
      char* m_pchText;
```

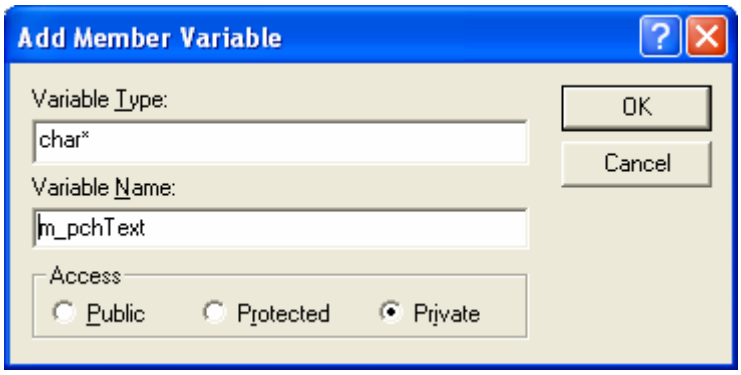

Figure 30: Adding a private member variable to `CText` class.

Add manually the following in the **Text.h**.

```
BEGIN_INTERFACE_PART(PersistStream, IPersistStream)
            STDMETHOD(GetClassID)(LPCLSID);
            STDMETHOD(IsDirty)();
            STDMETHOD(Load)(LPSTREAM);
            STDMETHOD(Save)(LPSTREAM, BOOL);
            STDMETHOD(GetSizeMax)(ULARGE_INTEGER FAR*);
      END_INTERFACE_PART(PersistStream)
```

```
    DECLARE_INTERFACE_MAP()

    BEGIN_INTERFACE_PART(PersistStream, IPersistStream)
        STDMETHOD(GetClassID)(LPCLSID);
        STDMETHOD(IsDirty)();
        STDMETHOD(Load)(LPSTREAM);
        STDMETHOD(Save)(LPSTREAM, BOOL);
        STDMETHOD(GetSizeMax)(ULARGE_INTEGER FAR*);
    END_INTERFACE_PART(PersistStream)

private:
    char* m_pchText;
};
```

Listing 6.

Then, add the following `PersistStream` interface in **Text.cpp**

```
      INTERFACE_PART(CText, IID_IPersistStream, PersistStream)
```

```
static const IID IID_IText =
{ 0x84d30689, 0x5963, 0x400d, { 0x8d, 0x4c, 0x19, 0xca, 0x75,

BEGIN_INTERFACE_MAP(CText, CCmdTarget)
    INTERFACE_PART(CText, IID_IPersistStream, PersistStream)
    INTERFACE_PART(CText, IID_IText, Dispatch)
END_INTERFACE_MAP()

// {B34965D8-9295-45E9-A4C6-EC0B51EBD1BD}
IMPLEMENT_OLECREATE(CText, "mymfc31b.Text", 0xb34965d8, 0x929
```

Listing 7.

Next, add the following `PersistStream` member function codes at the end of the **Text.cpp**.

```
STDMETHODIMP_(ULONG) CText::XPersistStream::AddRef()
{
	METHOD_PROLOGUE(CText, PersistStream)
	return (ULONG) pThis->ExternalAddRef();
}

STDMETHODIMP_(ULONG) CText::XPersistStream::Release()
{
	METHOD_PROLOGUE(CText, PersistStream)
	return (ULONG) pThis->ExternalRelease();
}

STDMETHODIMP CText::XPersistStream::QueryInterface(REFIID iid,
		void FAR* FAR* ppvObj)
{
	METHOD_PROLOGUE(CText, PersistStream)
	// ExternalQueryInterface looks up iid in the macro-generated tables
	return (HRESULT) pThis->ExternalQueryInterface(&iid, ppvObj);
}

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

STDMETHODIMP CText::XPersistStream::GetClassID(LPCLSID lpClassID)
{
	TRACE("Entering CText::XPersistStream::GetClassID\n");
	METHOD_PROLOGUE(CText, PersistStream)
	ASSERT_VALID(pThis);

	*lpClassID = CText::guid;
	return NOERROR;
}

STDMETHODIMP CText::XPersistStream::IsDirty()
{
	TRACE("Entering CText::XPersistStream::IsDirty\n");
	METHOD_PROLOGUE(CText, PersistStream)
	ASSERT_VALID(pThis);

	return NOERROR;
}

STDMETHODIMP CText::XPersistStream::Load(LPSTREAM pStm)
{
	ULONG nLength;
	STATSTG statstg;

	METHOD_PROLOGUE(CText, PersistStream)
	ASSERT_VALID(pThis);
	if(pThis->m_pchText != NULL)
	{ delete [] pThis->m_pchText; }
	// don't need to free statstg.pwcsName because of NONAME flag
	VERIFY(pStm->Stat(&statstg, STATFLAG_NONAME) == NOERROR);
	int nSize = statstg.cbSize.LowPart; // assume < 4 GB
	if(nSize > 0) {
		pThis->m_pchText = new char[nSize];
		pStm->Read(pThis->m_pchText, nSize, &nLength);
	}
```

```
	return NOERROR;
}

STDMETHODIMP CText::XPersistStream::Save(LPSTREAM pStm, BOOL fClearDirty)
{
	METHOD_PROLOGUE(CText, PersistStream)
	ASSERT_VALID(pThis);
	int nSize = strlen(pThis->m_pchText) + 1;
	pStm->Write(pThis->m_pchText, nSize, NULL);
	return NOERROR;
}

STDMETHODIMP CText::XPersistStream::GetSizeMax(ULARGE_INTEGER FAR* pcbSize)
{
	TRACE("Entering CText::XPersistStream::GetSizeMax\n");
	METHOD_PROLOGUE(CText, PersistStream)
	ASSERT_VALID(pThis);
	pcbSize->LowPart = strlen(pThis->m_pchText) + 1;
	pcbSize->HighPart = 0; // assume < 4 GB
	return NOERROR;
}
```

Add code for `GetText()` and `SetText()`.

```
VARIANT CText::GetText()
{
	return COleVariant(m_pchText).Detach();
}

void CText::SetText(const VARIANT FAR& newValue)
{
	CString strTemp;
	ASSERT(newValue.vt == VT_BSTR);
	if(m_pchText != NULL) {
		delete [] m_pchText;
	}
	strTemp = newValue.bstrVal; // converts to narrow chars
	m_pchText = new char[strTemp.GetLength() + 1];
	strcpy(m_pchText, strTemp);
}
```

Build MYMFC31B to generate the DLL. Make sure there is no error or warning. Then, use **regsvr32** to register the component as shown below so that MYMFC31C is ready to be used. To test MYMFC31B we need a client program, MYMFC31C.

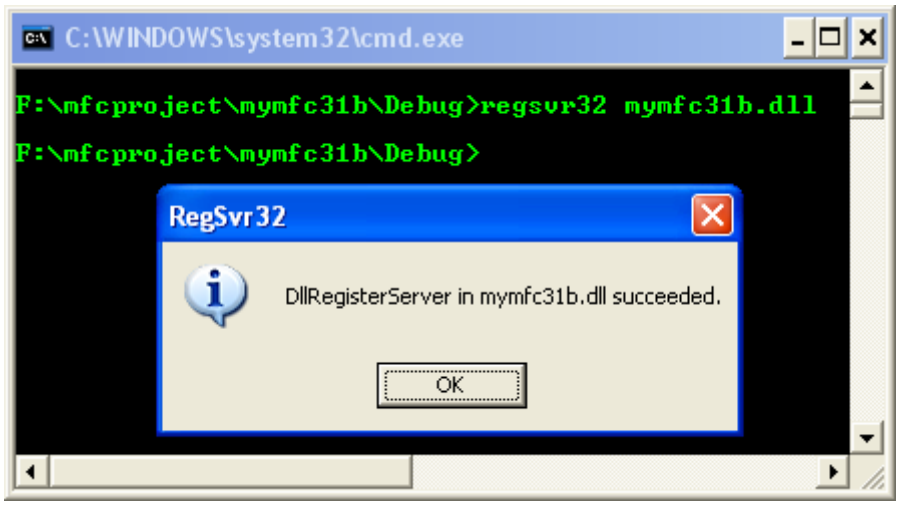

Figure 31: Registering **mymfc31b.dll** using **regsvr32** at command prompt.

ClassWizard generated the `CText` class as an ordinary **Automation** component. The `IPersistStream` interface was **added manually**. Look carefully at the `XPersistStream::Load` and `XPersistStream::Save` functions. The `Load()` function allocates heap memory and then calls `IStream::Read` to load the contents of the stream. The `Save()` function copies the object's data to the stream by calling `IStream::Write`.

The MYMFC31C Example: A Persistent Storage Client Program

This program is similar to MYMFC31A in function, indeed, the storage files are compatible. Internally, however, both worker threads use the persistent COM class `CText` (MYMFC31B) for loading and storing text.
To prepare MYMFC31C, open the **mymfc31c.dsw** workspace and build the project. Run the program from the debugger, first choosing **Write** from the **Storage** menu and then choosing **Read**. Observe the output in the **Debug** window or you can start it from scratch.

The MYMFC31C From Scratch

This is an SDI application without Automation and ActiveX Controls support.

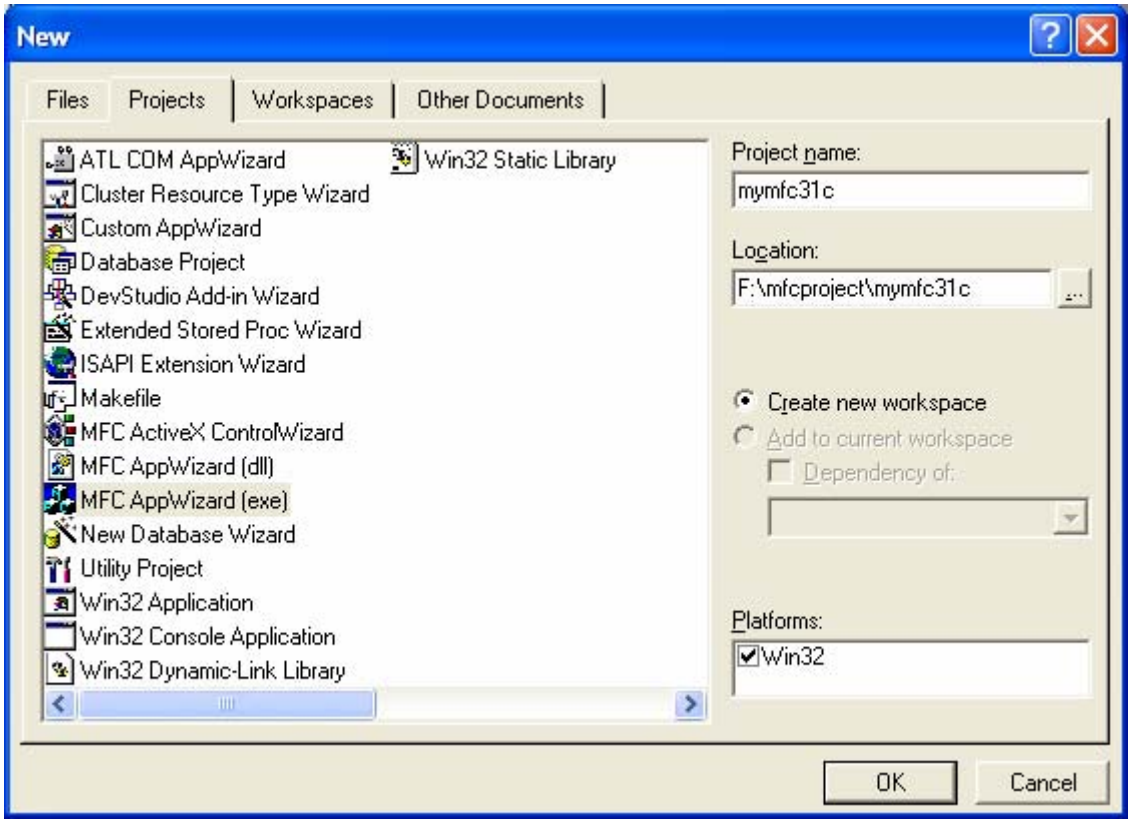

Figure 32: MYMFC31C – Visual C++ new project dialog.

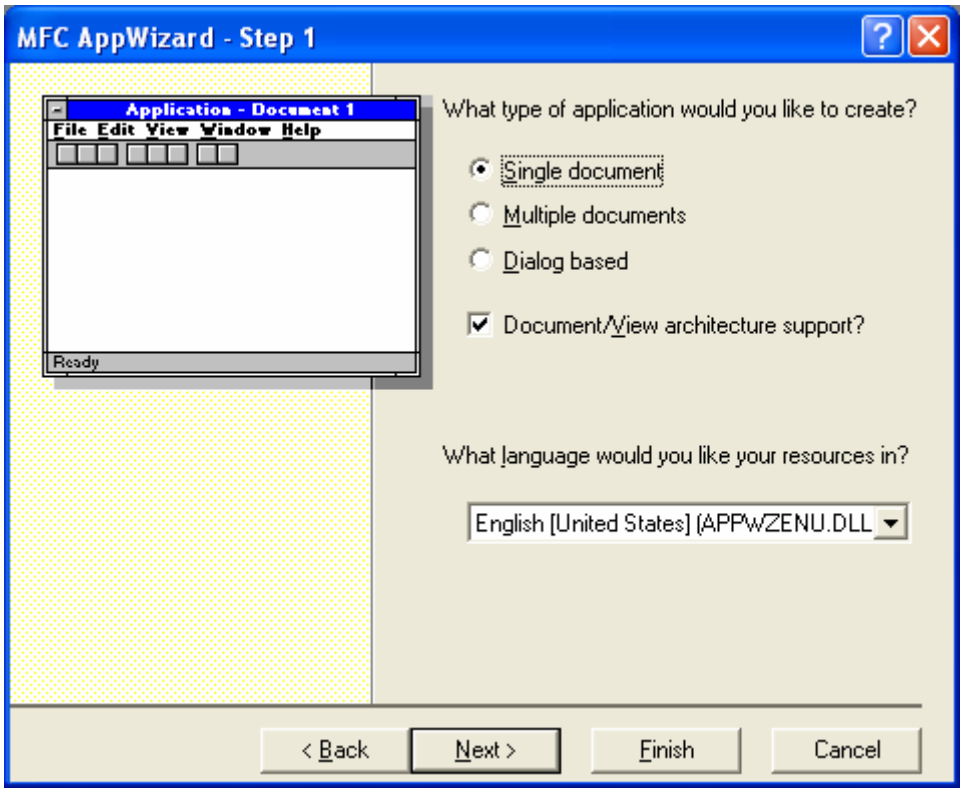

Figure 33: MYMFC31C – AppWizard step 1 of 6, select **Single document**.

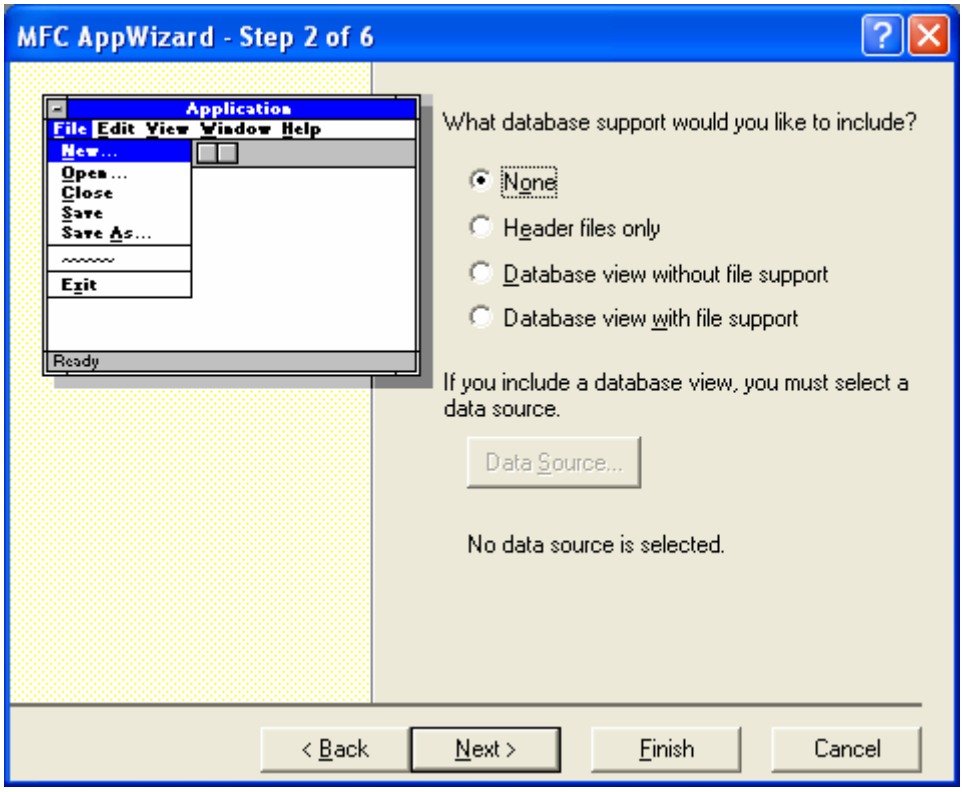

Figure 34: MYMFC31C – AppWizard step 2 of 6.

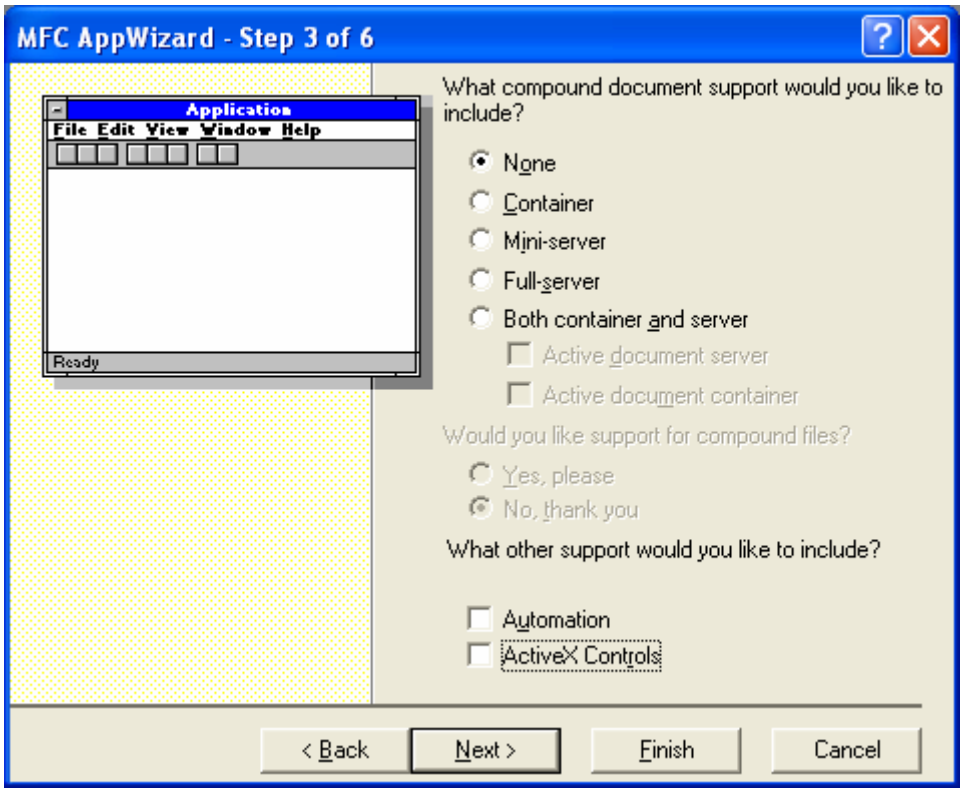

Figure 35: MYMFC31C – AppWizard step 3 of 6, deselect **Automation** and **ActiveX Controls**.

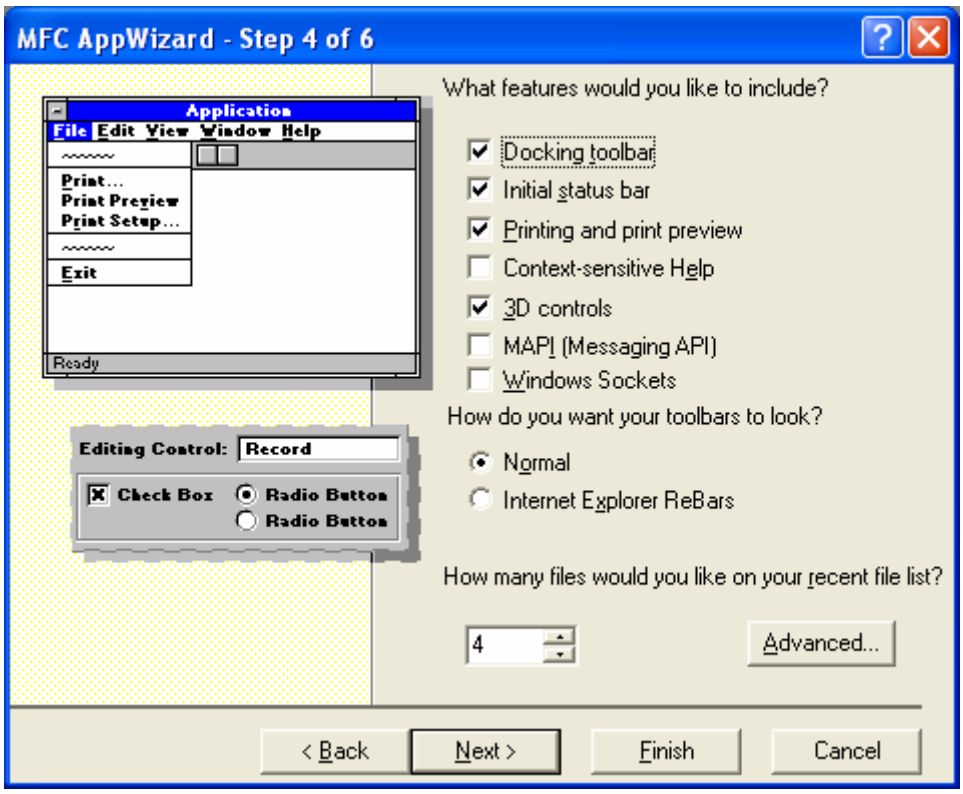

Figure 36: MYMFC31C – AppWizard step 4 of 6.

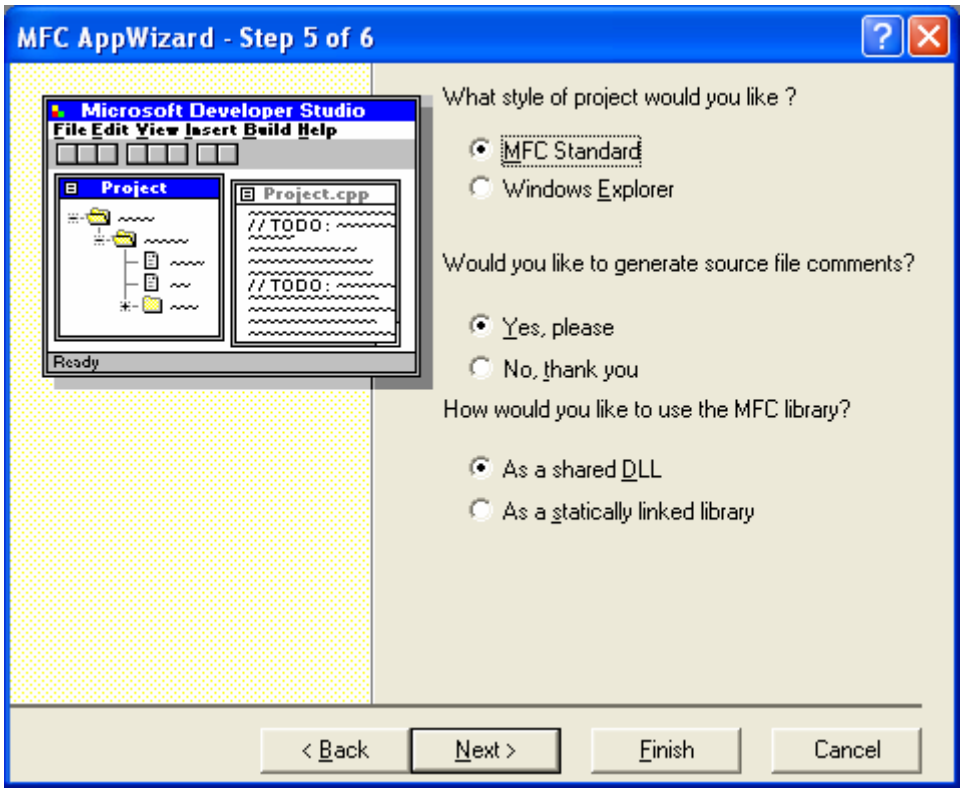

Figure 37: MYMFC31C – AppWizard step 5 of 6.

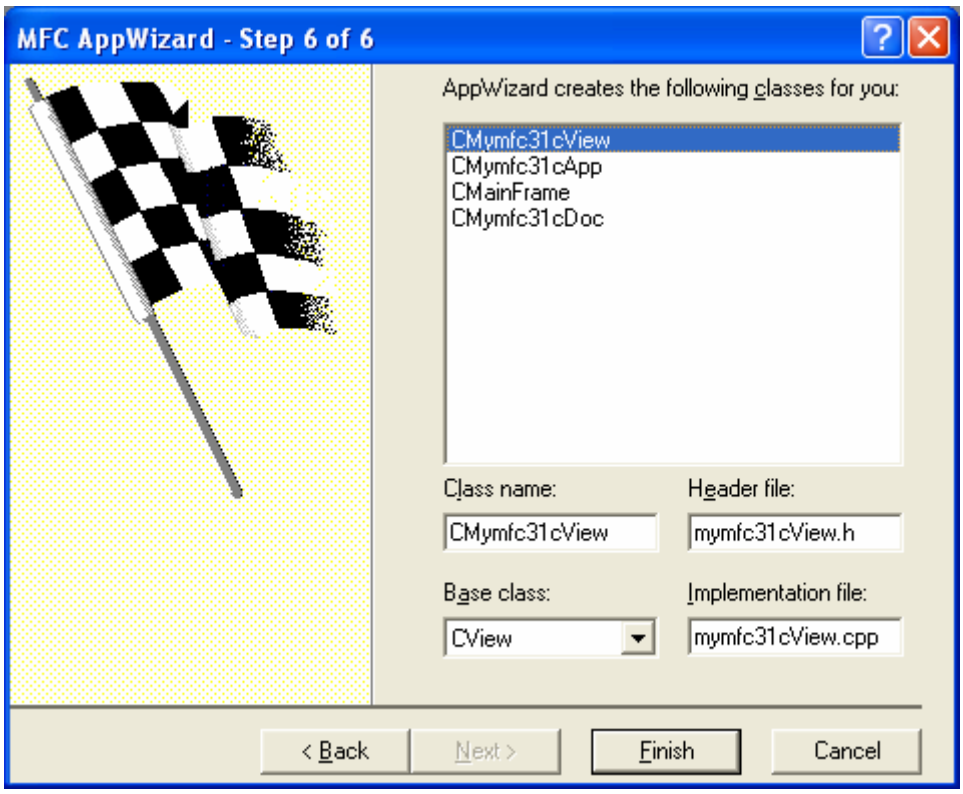

Figure 38: MYMFC31C – AppWizard step 6 of 6.

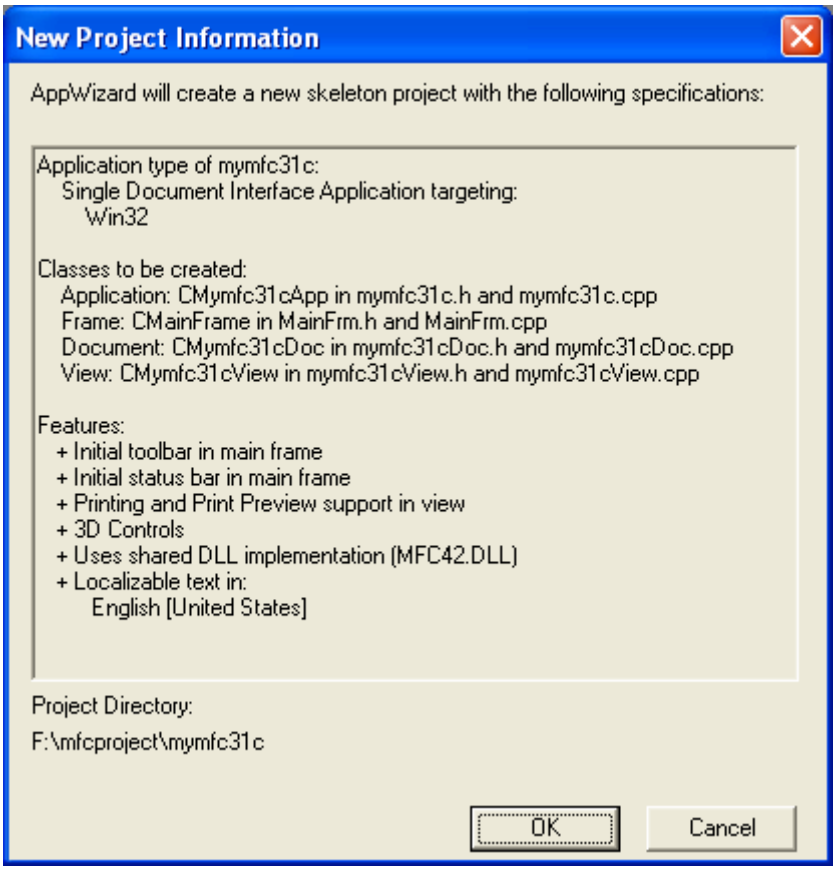

Figure 39: MYMFC31C project summary.

Add the following menu items.

ID	Caption
-	Storage
ID_STORAGE_READ	Read
ID_STORAGE_WRITE	Write

Table 12.

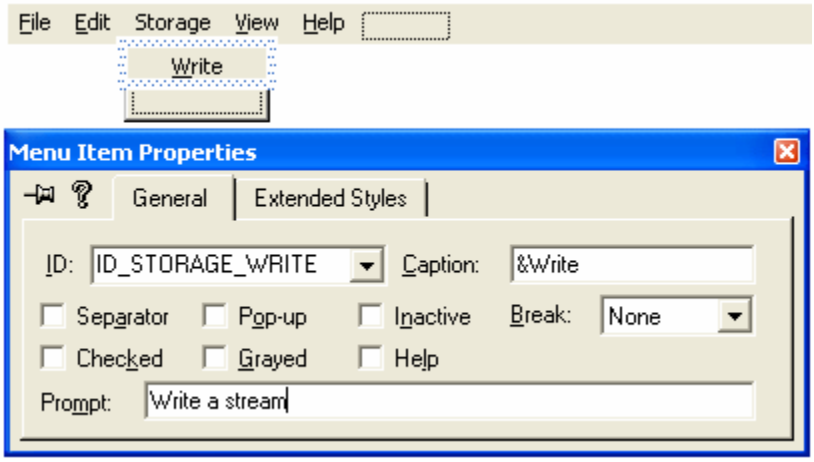

Figure 40: Adding **Write** menu item.

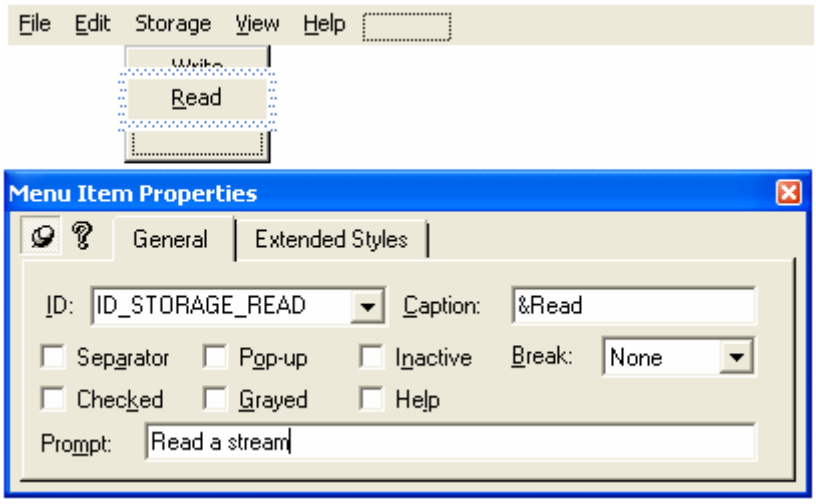

Figure 41: Adding **Read** menu item.

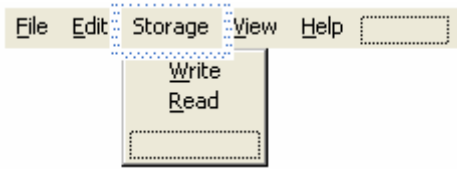

Figure 42: A completed MYMFC31C menus.

Use ClassWizard to add command handlers for the previous menu items as shown in the following Table to `CMymfc31cView` class.

ID	Type	Function
`ID_STORAGE_READ`	Command	`OnStorageRead()`
`ID_STORAGE_WRITE`	Command	`OnStorageWrite()`

Table 13.

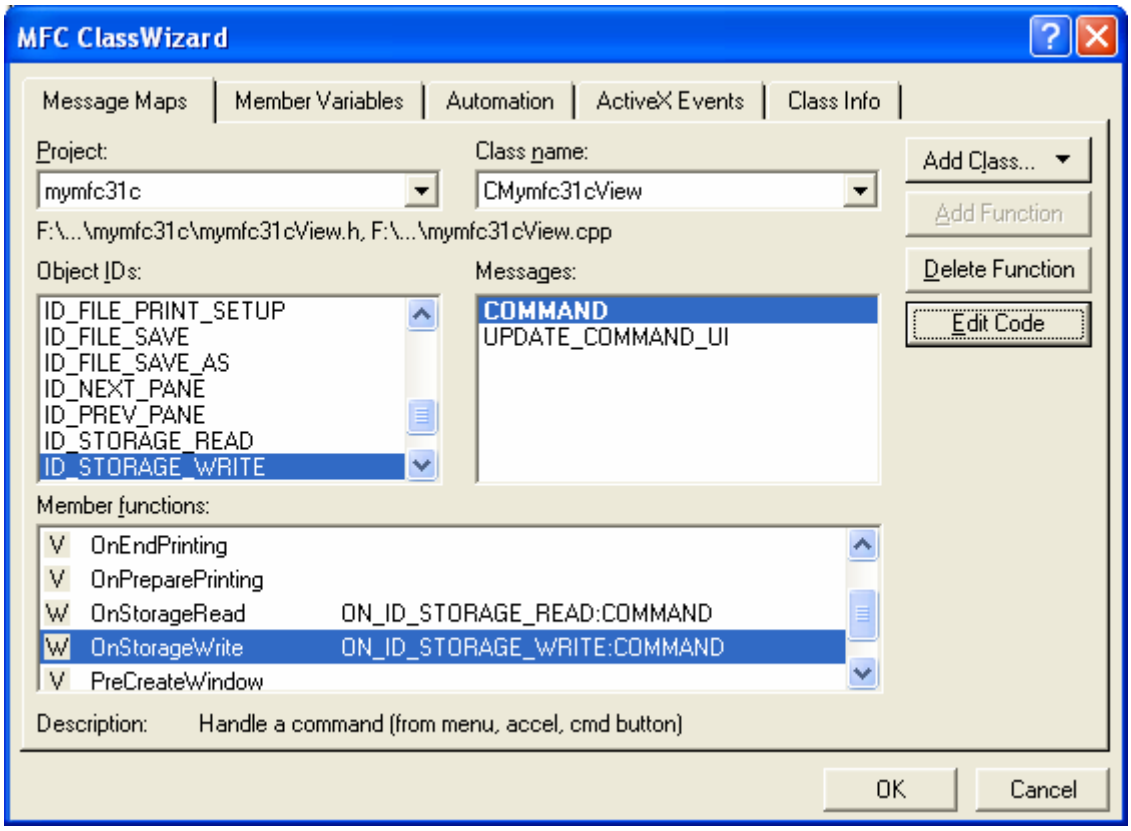

Figure 43: Adding command handlers to `CMymfc31cView` class.

Add codes to the command handlers.

```
void CMymfc31cView::OnStorageRead()
{
	// TODO: Add your command handler code here
	CWinThread* pThread = AfxBeginThread(ReadThreadProc, GetSafeHwnd());
}

void CMymfc31cView::OnStorageWrite()
{
	// TODO: Add your command handler code here
	CWinThread* pThread = AfxBeginThread(WriteThreadProc, GetSafeHwnd());
}
```

```
// CMymfc31cView message handlers

void CMymfc31cView::OnStorageRead()
{
    // TODO: Add your command handler code here
    CWinThread* pThread = AfxBeginThread(ReadThreadProc, GetSafeHwnd());
}

void CMymfc31cView::OnStorageWrite()
{
    // TODO: Add your command handler code here
    CWinThread* pThread = AfxBeginThread(WriteThreadProc, GetSafeHwnd());
}
```

Listing 8.

Add the `#include` directive to **mymfc31aView.cpp**.

```
#include "Thread.h"
```

```
#include "stdafx.h"
#include "mymfc31c.h"

#include "Thread.h"

#include "mymfc31cDoc.h"
#include "mymfc31cView.h"
```

Listing 9.

Add Thread.h, WriteThread.cpp and ReadThread.cpp files, the modified version of MYMFC31A to the project. Copy and paste the codes (provided in the following Listings) to the respective files.

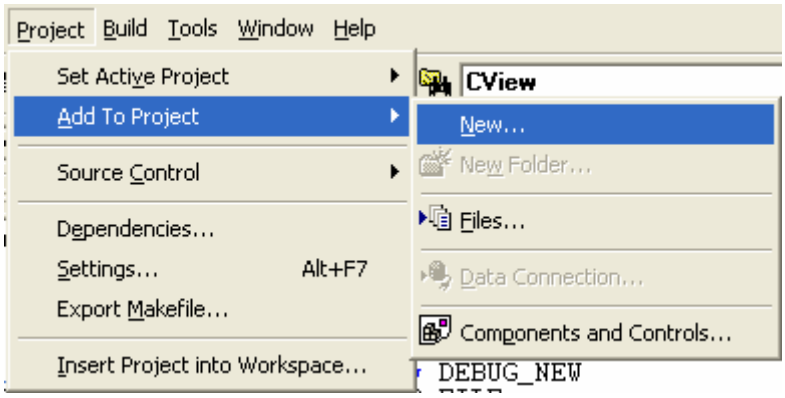

Figure 44: Adding new files to project.

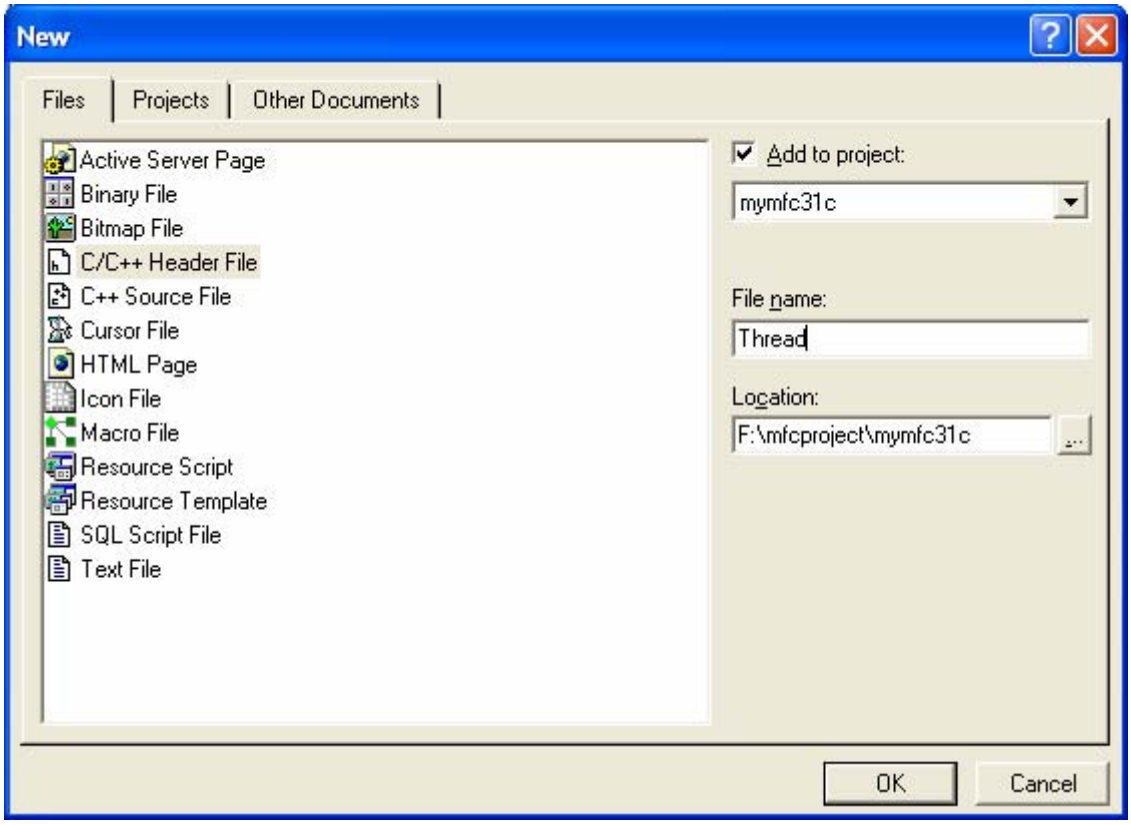

Figure 45: Adding **Thread.h** file to MYMFC31C. Repeat for other two source files.

Add the following lines in your **StdAfx.h** file:

```
#include <afxole.h>
#include <afxpriv.h> // for wide-character conversion
```

Then delete the following line (or commented out):

```
#define VC_EXTRALEAN
```

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

// #define VC_EXTRALEAN     // Exclude rarely-used stu

#include <afxwin.h>         // MFC core and standard c
#include <afxext.h>         // MFC extensions
#include <afxdtctl.h>       // MFC support for Interne
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>         // MFC support for Windows
#endif // _AFX_NO_AFXCMN_SUPPORT

#include <afxole.h>
#include <afxpriv.h> // for wide-character conversion

//{{AFX_INSERT_LOCATION}}
```

Listing 10.

Add MYMFC31B class from the type library to the project. Click **Add Class** button in ClassWizard and select **From a type library** button.

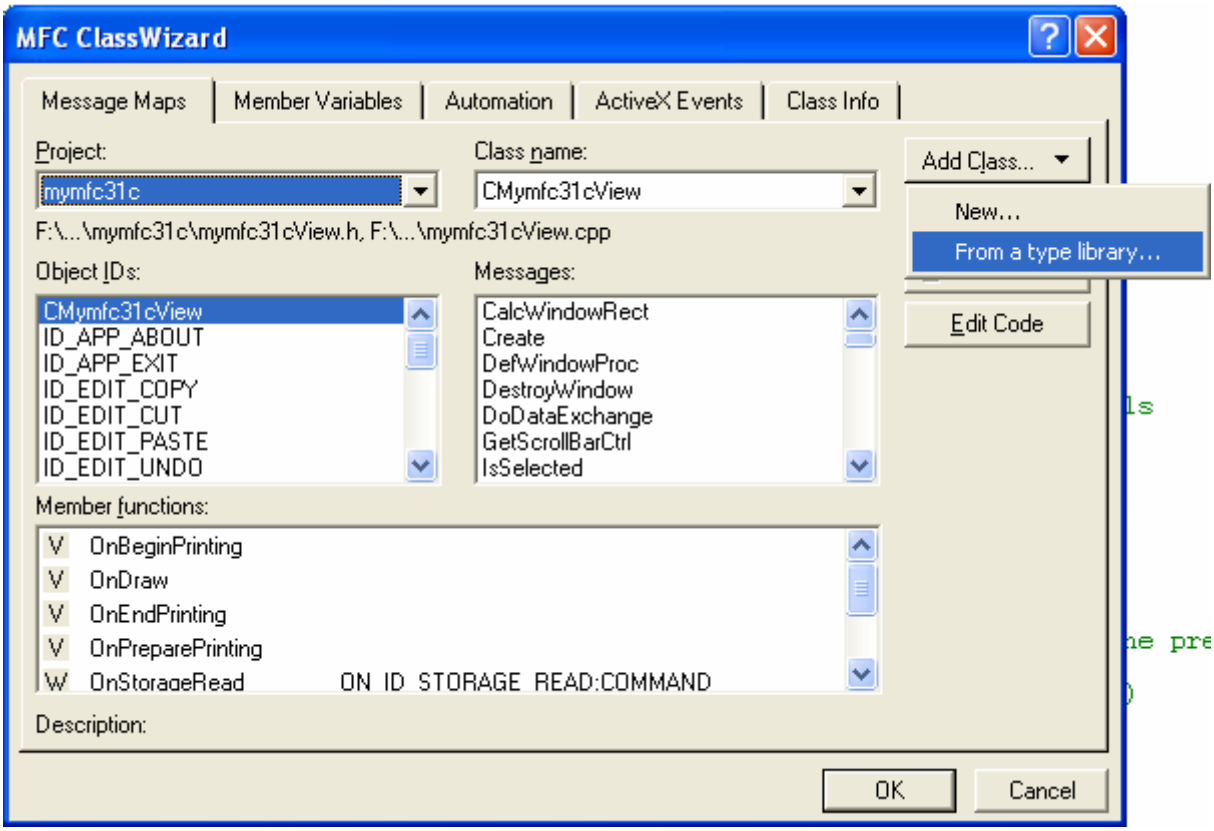

Figure 46: Adding new class from type library.

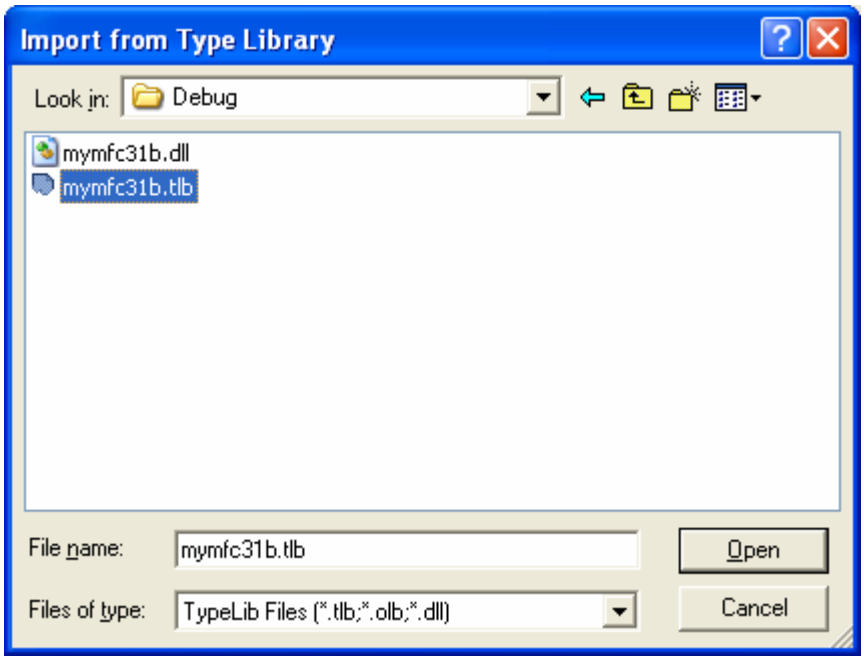

Figure 47: Browsing and selecting the mymfc31b.tlb type library file.

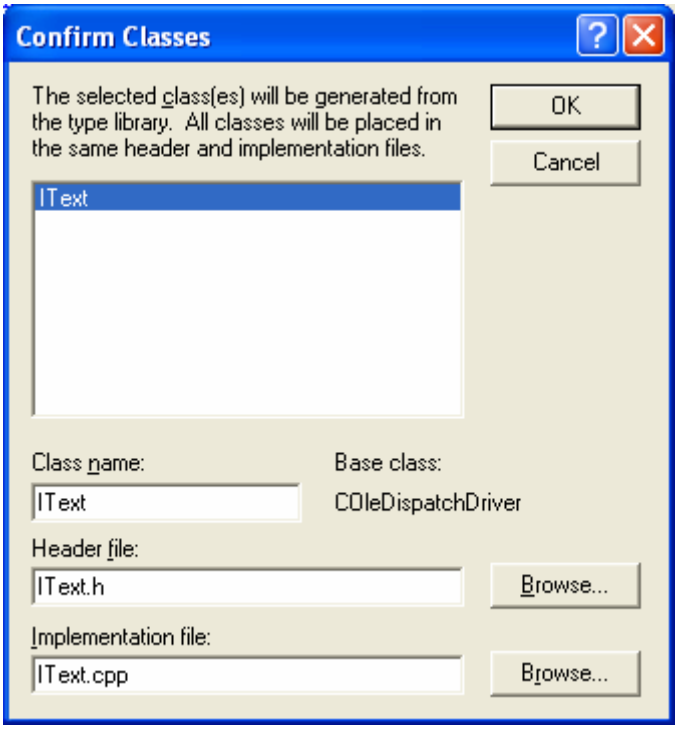

Figure 48: `IText` class from MYMFC31B addition confirmation dialog.

You can verify the added class through ClassView.

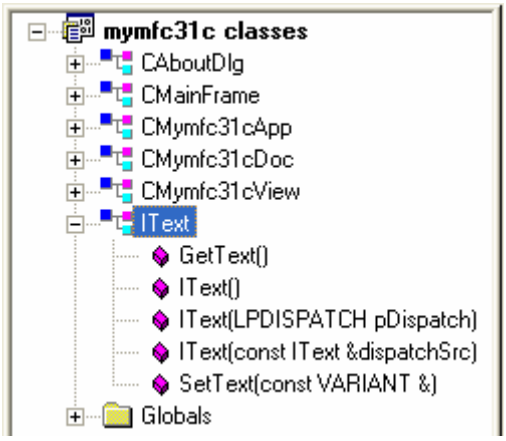

Figure 49: Verifying the added class through ClassView.

Build and run in debug mode. Click the **Write** menu.

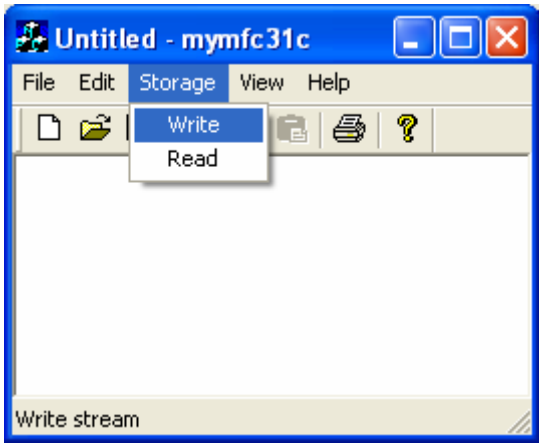

Figure 50: MYMFC31C in action.

Figure 51: The write operation message box displayed.

Then, click the **Read** menu.

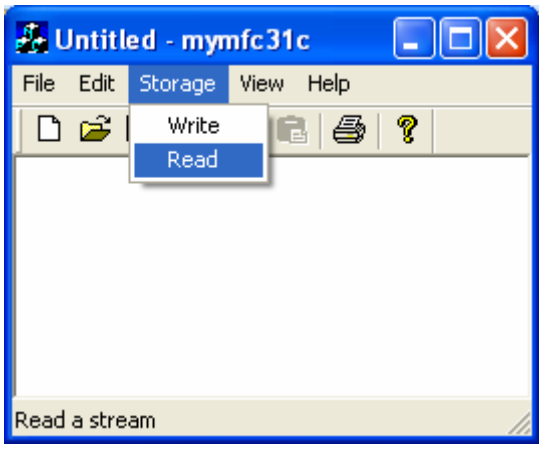

Figure 52: testing the **Read** menu, provided the previous write operation was successful.

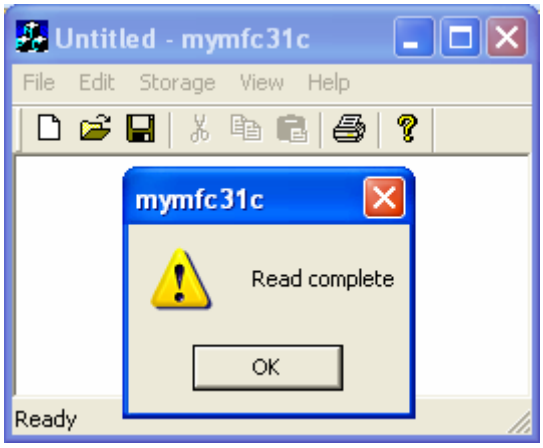

Figure 53: The completed read operation message box.

Check the **Debug** window. It will look something like this (the storage and stream should be different with yours).

```
Storage = mfcjadik
    Storage = final
        Stream = \mfcjadik\final\log.txt

Loaded 'C:\WINDOWS\system32\xpsp2res.dll', no matching symbolic information found.
Warning: CreateDispatch returning scode = REGDB_E_CLASSNOTREG ($80040154).
    Storage = myscribble
        Storage = Debug
        Stream = \mfcjadik\myscribble\ReadMe.txt
=============================================================================

Warning: CreateDispatch returning scode = REGDB_E_CLASSNOTREG ($80040154).
        Storage = res
Storage = mfcproject
    Storage = bkup
        Storage = mymfc29A
            Storage = Debug
            Stream = \mfcproject\bkup\mymfc29A\ReadMe.txt
=============================================================================
…
[Trimmed]
…
=============================================================================

Warning: CreateDispatch returning scode = REGDB_E_CLASSNOTREG ($80040154).
                                          Storage = res
                                          Storage = myproject
```

```
                                        Storage = childprocess
Storage = Debug
                                        Storage = console
Storage = Debug
                                        Storage = crtexecprog
…
[Trimmed]
…
Storage = Debug
Storage = Release
                                        Storage = testmydllruntime
Storage = Debug
                                        Storage = testsharedll
Storage = Debug
                                        Storage = win32prog
Storage = Debug
                                        Storage = myuserprog
                                        Storage = myuserprog
Storage = Debug
                                        Storage = RECYCLER
                                        Storage = NPROTECT
                                        Storage = S-1-5-21-1343024091-1647877149-
                                        Storage = S-1-5-21-1547161642-1580818891-
Storage = Df1
    Storage = Debug
    Stream = \RECYCLER\S-1-5-21-1547161642-1580818891-725345543-1003\Df1\ReadMe.txt
=====================================================================

Warning: CreateDispatch returning scode = REGDB_E_CLASSNOTREG ($80040154).
    Storage = res
                                        Storage = S-1-5-21-1547161642-1580818891-
                                        Storage = S-1-5-21-1757981266-1202660629-
                                        Storage = S-1-5-21-448539723-1364589140-8
                                        Storage = S-1-5-21-448539723-1364589140-8
                                        Storage = S-1-5-21-602162358-1682526488-8
                                        Storage = System Volume Information
The thread 0x968 has exited with code 0 (0x0).
The thread 0x970 has exited with code 0 (0x0).
The thread 0x8C has exited with code 0 (0x0).
The thread 0xBCC has exited with code 0 (0x0).
The thread 0x8E4 has exited with code 0 (0x0).
The thread 0x8DC has exited with code 0 (0x0).
The program 'F:\mfcproject\mymfc31c\Debug\mymfc31c.exe' has exited with code 0 (0x0).
```

Verify the **directdll.stg** file creation.

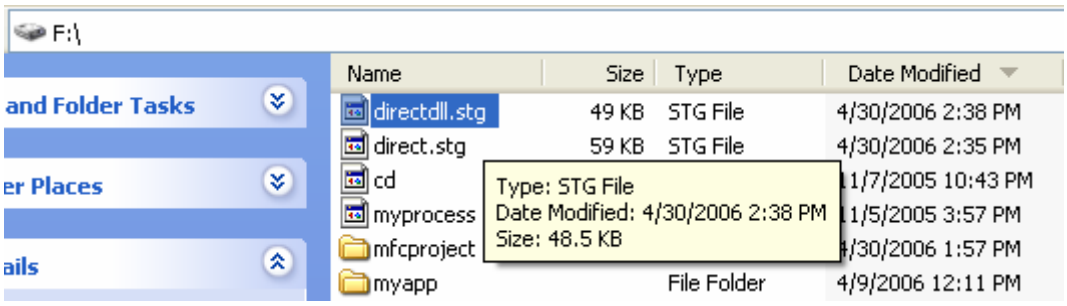

Figure 54: Verifying the **directdll.stg** file creation during the previous write operation.

The menu, the view class, and the application class are the same as the MYMFC31A versions. Only the thread code is different. Listing 11 lists the code for both the **WriteThread.cpp** and the **ReadThread.cpp** files.

WRITETHREAD.CPP

```
// WriteThread.cpp (MYMFC31C)
```

```
#include "StdAfx.h"
#include "Thread.h"
#include "itext.h"

CLSID g_clsid; // for the Text server
int g_nIndent = 0;
const char* g_szBlanks = "                    ";
const char* g_szRootStorageName = "\\directdll.stg";

UINT WriteThreadProc(LPVOID pParam)
{
    USES_CONVERSION;
    LPSTORAGE pStgRoot = NULL;
    g_nIndent = 0;
       ::CoInitialize(NULL);
       ::CLSIDFromProgID(L"MYMFC31B.TEXT", &g_clsid);
    VERIFY(::StgCreateDocfile(T2COLE(g_szRootStorageName),
           STGM_READWRITE | STGM_SHARE_EXCLUSIVE | STGM_CREATE,
            0, &pStgRoot) == S_OK);
    ReadDirectory("\\", pStgRoot);
    pStgRoot->Release();
    AfxMessageBox("Write complete");
    return 0;
}

void ReadDirectory(const char* szPath, LPSTORAGE pStg)
{
    // recursive function
    USES_CONVERSION;
    WIN32_FIND_DATA fData;
    HANDLE h;
    char szNewPath[MAX_PATH];
    char szStorageName[100];
    char szStreamName[100];
       char szData[81];
    char* pch = NULL;

    LPSTORAGE pSubStg = NULL;
    LPSTREAM pStream = NULL;
    LPPERSISTSTREAM pPersistStream = NULL;

    g_nIndent++;
    strcpy(szNewPath, szPath);
    strcat(szNewPath, "*.*");
    h = ::FindFirstFile(szNewPath, &fData);
    if (h == (HANDLE) 0xFFFFFFFF) return;  // can't find directory
    do {
      if (!strcmp(fData.cFileName, "..") ||
          !strcmp(fData.cFileName, ".")) continue;
         while((pch = strchr(fData.cFileName, '!')) != NULL) {
             *pch = '|';
         }
      if (fData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) {
              // It's a directory, so make a storage
        strcpy(szNewPath, szPath);
        strcat(szNewPath,fData.cFileName);
        strcat(szNewPath, "\\");

        strcpy(szStorageName, fData.cFileName);
        szStorageName[31] = '\0';    // limit imposed by OLE
        TRACE("%0.*sStorage = %s\n", (g_nIndent - 1) * 4, g_szBlanks,
szStorageName);
        VERIFY(pStg->CreateStorage(T2COLE(szStorageName),
               STGM_CREATE | STGM_READWRITE | STGM_SHARE_EXCLUSIVE,
               0, 0, &pSubStg) == S_OK);
        ASSERT(pSubStg != NULL);
        ReadDirectory(szNewPath, pSubStg);
        pSubStg->Release();
```

```
      }
      else {
        if ((pch = strrchr(fData.cFileName, '.')) != NULL) {
          if (!stricmp(pch, ".TXT")) {
                    // It's a text file, so make a stream
                    strcpy(szStreamName, fData.cFileName);
            strcpy(szNewPath, szPath);
            strcat(szNewPath, szStreamName);
                    szStreamName[32] = '\0'; // OLE max length
            TRACE("%0.*sStream = %s\n", (g_nIndent - 1) * 4,
                  g_szBlanks, szNewPath);
                    CStdioFile file(szNewPath, CFile::modeRead);
            // Ignore zero-length files
              if(file.ReadString(szData, 80)) {
              TRACE("%s\n", szData);
              VERIFY(pStg->CreateStream(T2COLE(szStreamName),
                     STGM_CREATE | STGM_READWRITE |
                             STGM_SHARE_EXCLUSIVE,
                     0, 0, &pStream) == S_OK);
              ASSERT(pStream != NULL);
                       // Include the null terminator in the stream
                       IText text;
                       //VERIFY
                      if(text.CreateDispatch(g_clsid)) //;
                      {
                       text.m_lpDispatch->QueryInterface(IID_IPersistStream,
                                    (void**) &pPersistStream);
                       ASSERT(pPersistStream != NULL);
                       text.SetText(COleVariant(szData));
                       pPersistStream->Save(pStream, TRUE);
                       pPersistStream->Release();
                      pStream->Release();
                             }
            }
          }
        }
      }
    } while (::FindNextFile(h, &fData));
    g_nIndent--;
}
```

READTHREAD.CPP

```
// ReadThread.cpp (MYMFC31C)

#include "StdAfx.h"
#include "Thread.h"
#include "itext.h"

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

UINT ReadThreadProc(LPVOID pParam)
{
       g_nIndent = 0;
       ::CoInitialize(NULL);
       ::CLSIDFromProgID(L"MYMFC31B.TEXT", &g_clsid);
       LPSTORAGE pStgRoot = NULL;
       if(::StgOpenStorage(L"\\DIRECTDLL.STG", NULL,
                           STGM_READ|STGM_SHARE_EXCLUSIVE,
                                 NULL, 0, &pStgRoot) == S_OK)
       {
              ASSERT(pStgRoot!= NULL);
              ReadStorage(pStgRoot);
              pStgRoot->Release();
```

```
        }
        else {
                AfxMessageBox("Storage file not available or not readable");
        }
        AfxMessageBox("Read complete");
        return 0;
}

void ReadStorage(LPSTORAGE pStg)
// reads one storage -- recursive calls for substorages
{
    LPSTORAGE pSubStg = NULL;
    LPSTREAM pStream = NULL;
    LPENUMSTATSTG pEnum = NULL;
    STATSTG statstg;
    LPPERSISTSTREAM pPersistStream = NULL;

        g_nIndent++;
        if(pStg->EnumElements(0, NULL, 0, &pEnum) != NOERROR)
        {
                ASSERT(FALSE);
                return;
        }
        while(pEnum->Next(1, &statstg, NULL) == NOERROR)
        {
                if(statstg.type == STGTY_STORAGE) {
                        VERIFY(pStg->OpenStorage(statstg.pwcsName, NULL,
                                STGM_READ|STGM_SHARE_EXCLUSIVE,
                                 NULL, 0, &pSubStg) == S_OK);
                        ASSERT(pSubStg != NULL);
                        ReadStorage(pSubStg);
                        pSubStg->Release();
                }
                else if(statstg.type == STGTY_STREAM)
                {
                        VERIFY(pStg->OpenStream(statstg.pwcsName, NULL,
                                STGM_READ|STGM_SHARE_EXCLUSIVE,
                                0, &pStream) == S_OK);
                        ASSERT(pStream != NULL);
                        IText text;
                        //VERIFY
                        if(text.CreateDispatch(g_clsid)) //;
                        {
                        text.m_lpDispatch->QueryInterface(IID_IPersistStream,
                                        (void**) &pPersistStream);
                        ASSERT(pPersistStream != NULL);
                        pPersistStream->Load(pStream);
                        pPersistStream->Release();
                        COleVariant va = text.GetText();
                        ASSERT(va.vt == VT_BSTR);
                        CString str = va.bstrVal;
                        TRACE("%s\n", str);
                        pStream->Release();
                        }
                }
                else {
                        ASSERT(FALSE);  // LockBytes?
                }
                ::CoTaskMemFree(statstg.pwcsName);
        }
        pEnum->Release();
        g_nIndent--;
}
```

Listing 11: The code listing for the two worker threads in MYMFC31C.

Look at the second half of the `ReadDirectory()` function in the **WriteThread.cpp** file in Figure 27-5. For each TXT file, the program constructs a `CText` object by constructing an IText driver object and then calling `CreateDispatch()`. Then it calls the `SetText()` member function to write the first line of the file to the object. After that, the program calls `IPersistStream::Save` to write the object to the compound file. The `CText` object is deleted after the `IPersistStream` pointer is released and after the `IText` object is deleted, releasing the object's IDispatch pointer.

Now look at the second half of the `ReadStorage()` function in the **ReadThread.cpp** file. Like `ReadDirectory()`, it constructs an `IText` driver object and calls `CreateDispatch()`. Then it calls `QueryInterface()` to get the object's `IPersistStream` pointer, which it uses to call `Load()`. Finally, the program calls `GetText()` to retrieve the line of text for tracing.

As you've learned already, a COM component usually implements `IPersistStorage`, not `IPersistStream`. The `CText` class could have worked this way, but then the compound file would have been more complex because each TXT file would have needed both a storage element (to support the interface) and a subsidiary stream element (to hold the text).

Now get ready to take a giant leap. Suppose you have a true creatable-by-CLSID COM component that supports the `IPersistStorage` interface. Recall the `IStorage` functions for class IDs. If a storage element contains a class ID, together with all the data an object needs, COM can load the server, use the class factory to construct the object, get an `IPersistStorage` pointer, and call `Load()` to load the data from a compound file. This is a preview of compound documents, which you'll see in Module 27.

Compound File Fragmentation

Structured storage has a dark side. Like the disk drive itself, compound files can become **fragmented** with frequent use. If a disk drive becomes fragmented, however, you still have the same amount of free space. With a compound file, space from deleted elements isn't always recovered. This means that compound files can keep growing even if you delete data.

Fortunately, there is a way to recover unused space in a compound file. You simply create a new file and copy the contents. The `IStorage::CopyTo` function can do the whole job in one call if you use it to copy the root storage. You can either write a stand-alone utility or build a file regeneration capability into your application.

Other Compound File Advantages

You've seen how compound files add a kind of random access capability to your programs, and you can appreciate the value of transactioning. Now consider the brave new world in which every program can read any other program's documents. We're not there yet, but we have a start. Compound files from Microsoft applications have a stream under the root storage named **\005SummaryInformation**. This stream is formatted as a property set, as defined for ActiveX controls. If you can decode the format for this stream, you can open any conforming file and read the summary.

Visual C++ comes with a **compound file viewing** utility named **DocFile Viewer** (**Dfview.exe**), which uses a tree view to display the file's storages and streams.

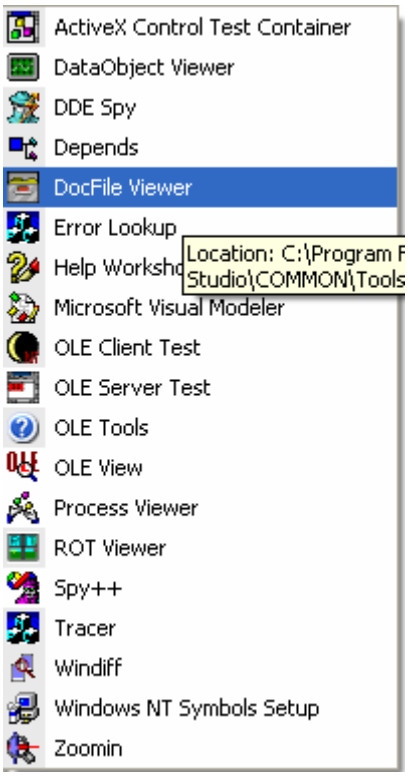

Figure 55: Using DocFile Viewer to view structured storage file.

Here is the **DocFile Viewer** output for the structured storage file generated by MYMFC31C (**directdll.stg**).

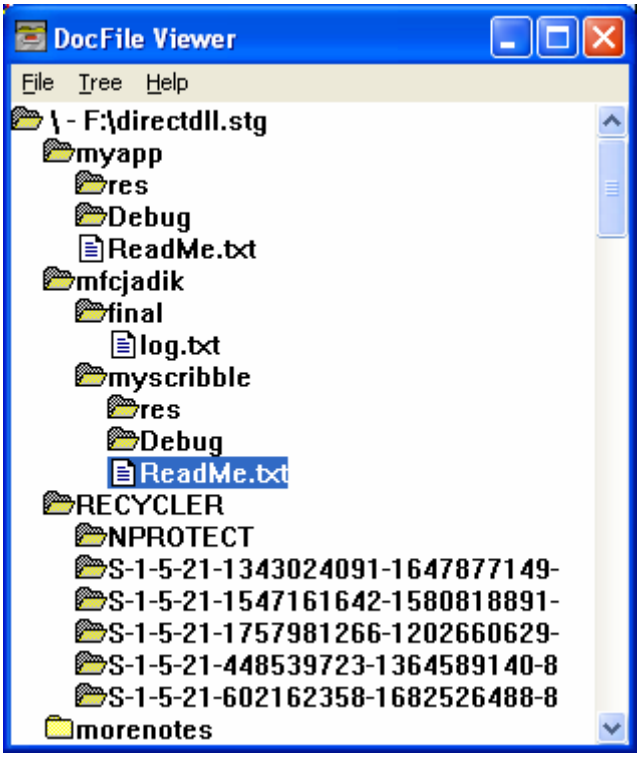

Figure 56: MYMFC31C's **directdll.stg** structured storage file content.

As a matter of fact, you can use **DocFile Viewer** to view the structure of any compound file. Are you starting to see the potential of this "universal file format?"

----------End----------

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. DCOM at MSDN.
5. COM+ at MSDN.
6. COM at MSDN.
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.