Chapter 6

# File I/O and Serialization

File input and output (I/O) services are a staple of any operating system. Not surprisingly, Microsoft Windows provides an assortment of API functions for reading, writing, and manipulating disk files. MFC casts these functions in an object-oriented mold with its *CFile* class, which lets files be viewed as objects that are operated on with *CFile* member functions such as *Read* and *Write*. *CFile* has all the tools the MFC programmer needs to perform low-level file I/O.

The most common reason for writing file I/O code is to support document saving and loading. Although there's nothing wrong with using *CFile* objects to write documents to disk and read them back, most MFC applications don't do it that way; they use *CArchive* objects instead. Thanks to some strategic operator overloading performed by MFC, most data can be serialized—that is, output as a byte stream—to a *CArchive* or deserialized from a *CArchive* with syntactical ease. Moreover, if a *CArchive* object is attached to a *CFile* object, data that is serialized to the *CArchive* is transparently written to disk. You can later reconstitute data archived in this manner by deserializing it from a *CArchive* associated with the same file.

The ability to save and load documents by serializing them to or from a *CArchive* is one of the fundamental building blocks of MFC's document/view architecture. Although knowledge of *CArchive* is of limited use for now, rest assured that it will come in exceedingly handy when we begin writing document/view applications in Chapter 9.

# The *CFile* Class

*CFile* is a relatively simple class that encapsulates the portion of the Win32 API that deals with file I/O. Among its 25-plus member functions are functions for opening and closing files, reading and writing file data, deleting and renaming files, and retrieving file information. Its one public data member, *m\_hFile*, holds the handle of the file associated with a *CFile* object. A protected *CString* data member named *m\_strFileName* holds the file name. The member functions *GetFilePath*, *GetFileName*, and *GetFileTitle* can be used to extract the file name, in whole or in part. For example, if the full file name, path name included, is C:\Personal\File.txt, *GetFilePath* returns the entire string, *GetFileName* returns "File.txt," and *GetFileTitle* returns "File."

But to dwell on these functions is to disregard the features of *CFile* that are the most important to programmers—that is, the functions used to write data to disk and read it back. The next several sections offer a brief tutorial in the use of *CFile* and its rather peculiar way of letting you know when an error occurs. (*Hint*: If you've never used C++ exception handling, now is a good time to dust off the manual and brush up on it.)

## Opening, Closing, and Creating Files

Files can be opened with *CFile* in either of two ways. The first option is to construct an uninitialized *CFile* object and call *CFile::Open*. The following code fragment uses this technique to open a file named File.txt with read/write access. Because no path name is provided in the function's first parameter, *Open* will fail unless the file is located in the current directory:

|  |
| --- |
| CFile file;  file.Open (\_T ("File.txt"), CFile::modeReadWrite); |

*CFile::Open* returns a BOOL indicating whether the operation was successful. The following example uses that return value to verify that the file was successfully opened:

|  |
| --- |
| CFile file;  if (file.Open (\_T ("File.txt"), CFile::modeReadWrite)) {  // It worked!    } |

A nonzero return value means the file was opened; 0 means it wasn't. If *CFile::Open* returns 0 and you want to know *why* the call failed, create a *CFileException* object and pass its address to *Open* in the third parameter:

|  |
| --- |
| CFile file;  CFileException e;  if (file.Open (\_T ("File.txt"), CFile::modeReadWrite, &e)) {  // It worked!    }  else {  // Open failed. Tell the user why.  e.ReportError ();  } |

If *Open* fails, it initializes the *CFileException* object with information describing the nature of the failure. *ReportError* displays an error message based on that information. You can find out what caused the failure by examining the *CFileException*'s public *m\_cause* data member. The documentation for *CFileException* contains a complete list of error codes.

The second option is to open the file using *CFile*'s constructor. Rather than construct an empty *CFile* object and call *Open*, you can create a *CFile* object and open a file in one step like this:

|  |
| --- |
| CFile file (\_T ("File.txt"), CFile::modeReadWrite); |

If the file can't be opened, *CFile*'s constructor throws a *CFileException*. Therefore, code that opens files using *CFile::CFile* normally uses *try* and *catch* blocks to trap errors:

|  |
| --- |
| try {  CFile file (\_T ("File.txt"), CFile::modeReadWrite);    }  catch (CFileException\* e) {  // Something went wrong.  e->ReportError ();  e->Delete ();  } |

It's up to you to delete the *CFileException* objects MFC throws to you. That's why this example calls *Delete* on the exception object after processing the exception. The only time you don't want to call *Delete* is the rare occasion when you use *throw* to rethrow the exception.

To create a new file rather than open an existing one, include a *CFile::modeCreate* flag in the second parameter to *CFile::Open* or the *CFile* constructor:

|  |
| --- |
| CFile file (\_T ("File.txt"), CFile::modeReadWrite ¦ CFile::modeCreate); |

If a file created this way already exists, its length is truncated to 0. To create the file if it doesn't exist or to open it without truncating it if it does exist, include a *CFile::modeNoTruncate* flag as well:

|  |
| --- |
| CFile file (\_T ("File.txt"), CFile::modeReadWrite ¦ CFile::modeCreate ¦  CFile::modeNoTruncate); |

An open performed this way almost always succeeds because the file is automatically created for you if it doesn't already exist.

By default, a file opened with *CFile::Open* or *CFile::CFile* is opened for exclusive access, which means that no one else can open the file. If desired, you can specify a sharing mode when opening the file to explicitly grant others permission to access the file, too. Here are the four sharing modes that you can choose from:

|  |  |
| --- | --- |
| ***Sharing Mode*** | ***Description*** |
| *CFile::shareDenyNone* | Opens the file nonexclusively |
| *CFile::shareDenyRead* | Denies read access to other parties |
| *CFile::shareDenyWrite* | Denies write access to other parties |
| *CFile::shareExclusive* | Denies both read and write access to other parties (default) |

In addition, you can specify any one of the following three types of read/write access:

|  |  |
| --- | --- |
| ***Access Mode*** | ***Description*** |
| *CFile::modeReadWrite* | Requests read and write access |
| *CFile::modeRead* | Requests read access only |
| *CFile::modeWrite* | Requests write access only |

A common use for these options is to allow any number of clients to open a file for reading but to deny any client the ability to write to it:

|  |
| --- |
| CFile file (\_T ("File.txt"), CFile::modeRead ¦ CFile::shareDenyWrite); |

If the file is already open for writing when this statement is executed, the call will fail and *CFile* will throw a *CFileException* with *m\_cause* equal to *CFileException::sharingViolation*.

An open file can be closed in two ways. To close a file explicitly, call *CFile::Close* on the corresponding *CFile* object:

|  |
| --- |
| file.Close (); |

If you'd prefer, you can let *CFile*'s destructor close the file for you. The class destructor calls *Close* if the file hasn't been closed already. This means that a *CFile* object created on the stack will be closed automatically when it goes out of scope. In the following example, the file is closed the moment the brace marking the end of the *try* block is reached:

|  |
| --- |
| try {  CFile file (\_T ("File.txt"), CFile::modeReadWrite);  // CFile::~CFile closes the file.  } |

One reason programmers sometimes call *Close* explicitly is to close the file that is currently open so that they can open another file using the same *CFile* object.

## Reading and Writing

A file opened with read access can be read using *CFile::Read*. A file opened with write access can be written with *CFile::Write*. The following example allocates a 4-KB file I/O buffer and reads the file 4 KB at a time. Error checking is omitted for clarity.

|  |
| --- |
| BYTE buffer[0x1000];  CFile file (\_T ("File.txt"), CFile::modeRead);  DWORD dwBytesRemaining = file.GetLength ();  while (dwBytesRemaining) {  UINT nBytesRead = file.Read (buffer, sizeof (buffer));  dwBytesRemaining -= nBytesRead;  } |

A count of bytes remaining to be read is maintained in *dwBytesRemaining*, which is initialized with the file size returned by *CFile::GetLength*. After each call to *Read*, the number of bytes read from the file (*nBytesRead*) is subtracted from *dwBytesRemaining*. The *while* loop executes until *dwBytesRemaining* reaches 0.

The following example builds on the code in the previous paragraph by using *::CharLowerBuff* to convert all the uppercase characters read from the file to lowercase and using *CFile::Write* to write the converted text back to the file. Once again, error checking is omitted for clarity.

|  |
| --- |
| BYTE buffer[0x1000];  CFile file (\_T ("File.txt"), CFile::modeReadWrite);  DWORD dwBytesRemaining = file.GetLength ();  while (dwBytesRemaining) {  DWORD dwPosition = file.GetPosition ();  UINT nBytesRead = file.Read (buffer, sizeof (buffer));  ::CharLowerBuff ((LPTSTR)buffer, nBytesRead);  file.Seek (dwPosition, CFile::begin);  file.Write (buffer, nBytesRead);  dwBytesRemaining -= nBytesRead;  } |

This example uses the *CFile* functions *GetPosition* and *Seek* to manipulate the file pointer—the offset into the file at which the next read or write is performed—so that the modified data is written over the top of the original. *Seek*'s second parameter specifies whether the byte offset passed in the first parameter is relative to the beginning of the file (*CFile::begin*), the end of the file (*CFile::end*), or the current position (*CFile::current*). To quickly seek to the beginning or end of a file, use *CFile::SeekToBegin* or *CFile::SeekToEnd*.

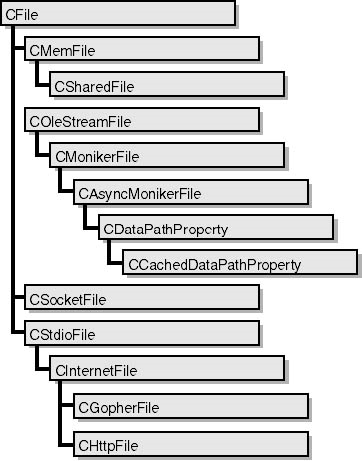
*Read*, *Write*, and other *CFile* functions throw a *CFileException* if an error occurs during a file I/O operation. *CFileException::m\_cause* tells you why the error occurred. For example, attempting to write to a disk that is full throws a *CFileException* with *m\_cause* equal to *CFileException::diskFull*. Attempting to read beyond the end of a file throws a *CFileException* with *m\_cause* equal to *CFileException::endOfFile*. Here's how the routine that converts all the lowercase text in a file to uppercase might look with error checking code included:

|  |
| --- |
| BYTE buffer[0x1000];  try {  CFile file (\_T ("File.txt"), CFile::modeReadWrite);  DWORD dwBytesRemaining = file.GetLength ();  while (dwBytesRemaining) {  DWORD dwPosition = file.GetPosition ();  UINT nBytesRead = file.Read (buffer, sizeof (buffer));  ::CharLowerBuff ((LPTSTR)buffer, nBytesRead);  file.Seek (dwPosition, CFile::begin);  file.Write (buffer, nBytesRead);  dwBytesRemaining -= nBytesRead;  }  }  catch (CFileException\* e) {  e->ReportError ();  e->Delete ();  } |

If you don't catch exceptions thrown by *CFile* member functions, MFC will catch them for you. MFC's default handler for unprocessed exceptions uses *ReportError* to display a descriptive error message. Normally, however, it's in your best interest to catch file I/O exceptions to prevent critical sections of code from being skipped.

## *CFile* Derivatives

*CFile* is the root class for an entire family of MFC classes. The members of this family and the relationships that they share with one another are shown in Figure 6-1.



**Figure 6-1.** *The* CFile *family.*

Some members of the *CFile* family exist solely to provide filelike interfaces to nonfile media. For example, *CMemFile* and *CSharedFile* let blocks of memory be read and written as if they were files. MFC's *COleDataObject::GetFileData* function, which is discussed in [Chapter 19](mk:@MSITStore:C:\Program%20Files%20(x86)\MSPress\BooksOnline\Programming%20Windows%20with%20MFC%20Second%20Edition\progmfc2.chm::/ch19a.htm), uses this handy abstraction to allow OLE drop targets and users of the OLE clipboard to retrieve data from memory with *CFile*::*Read*. *CSocketFile* provides a similar abstraction for TCP/IP sockets. MFC programmers sometimes place a *CSocketFile* object between a *CSocket* object and a *CArchive* object so that C++'s insertion and extraction operators can be used to write to and read from an open socket. *COleStreamFile* makes a stream object—a COM object that represents a byte stream—look like an ordinary file. It plays an important role in MFC applications that support object linking and embedding (OLE).

*CStdioFile* simplifies the programmatic interface to text files. It adds just two member functions to those it inherits from *CFile*: a *ReadString* function for reading lines of text and a *WriteString* function for outputting lines of text. In *CStdioFile*-speak, a line of text is a string of characters delimited by a carriage return and line feed pair (0x0D and 0x0A). *ReadString* reads everything from the current file position up to, and optionally including, the next carriage return. *WriteString* outputs a text string and writes a carriage return and line feed to the file, too. The following code fragment opens a text file named File.txt and dumps its contents to the debug output window:

|  |
| --- |
| try {  CString string;  CStdioFile file (\_T ("File.txt"), CFile::modeRead);  while (file.ReadString (string))  TRACE (\_T ("%s\n"), string);  }  catch (CFileException\* e) {  e->ReportError ();  e->Delete ();  } |

Like *Read* and *Write*, *ReadString* and *WriteString* throw exceptions if an error prevents them from carrying out their missions.

## Enumerating Files and Folders

*CFile* includes a pair of static member functions named *Rename* and *Remove* that can be used to rename and delete files. It doesn't, however, include functions for enumerating files and folders. For that, you must resort to the Windows API.

The key to enumerating files and folders is a pair of API functions named *::FindFirstFile* and *::FindNextFile*. Given an absolute or relative file name specification (for example, "C:\\\*.\*" or "\*.\*"), *::FindFirstFile* opens a *find handle* and returns it to the caller. *::FindNextFile* uses that handle to enumerate file system objects. The general strategy is to call *::FindFirstFile* once to begin an enumeration and then to call *::FindNextFile* repeatedly until the enumeration is exhausted. Each successful call to *::FindFirstFile* or *::FindNextFile*—that is, a call to *::FindFirstFile* that returns any value other than INVALID\_HANDLE\_VALUE or a call to *::FindNextFile* that returns a non-NULL value—fills a WIN32\_FIND\_DATA structure with information about one file or directory. WIN32\_FIND\_DATA is defined this way in ANSI code builds:

|  |
| --- |
| typedef struct \_WIN32\_FIND\_DATAA {  DWORD dwFileAttributes;  FILETIME ftCreationTime;  FILETIME ftLastAccessTime;  FILETIME ftLastWriteTime;  DWORD nFileSizeHigh;  DWORD nFileSizeLow;  DWORD dwReserved0;  DWORD dwReserved1;  CHAR cFileName[ MAX\_PATH ];  CHAR cAlternateFileName[ 14 ];  } WIN32\_FIND\_DATAA;  typedef WIN32\_FIND\_DATAA WIN32\_FIND\_DATA; |

To determine whether the item represented by the WIN32\_FIND\_DATA structure is a file or a directory, test the *dwFileAttributes* field for a FILE\_ATTRIBUTE\_DIRECTORY flag:

|  |
| --- |
| if (fd.dwFileAttributes & FILE\_ATTRIBUTE\_DIRECTORY) {  // It's a directory.  }  else {  // It's a file.  } |

The *cFileName* and *cAlternateFileName* fields hold the file or directory name. *cFileName* contains the long name; *cAlternateFileName* contains the short (8.3 format) name. When the enumeration is complete, you should close any handles returned by *::FindFirstFile* with *::FindClose*.

To demonstrate, the following routine enumerates all the files in the current directory and writes their names to the debug output window:

|  |
| --- |
| WIN32\_FIND\_DATA fd;  HANDLE hFind = ::FindFirstFile (\_T ("\*.\*"), &fd);  if (hFind != INVALID\_HANDLE\_VALUE) {  do {  if (!(fd.dwFileAttributes & FILE\_ATTRIBUTE\_DIRECTORY))  TRACE (\_T ("%s\n"), fd.cFileName);  } while (::FindNextFile (hFind, &fd));  ::FindClose (hFind);  } |

Enumerating all the subdirectories in the current directory requires just one simple change:

|  |
| --- |
| WIN32\_FIND\_DATA fd;  HANDLE hFind = ::FindFirstFile (\_T ("\*.\*"), &fd);  if (hFind != INVALID\_HANDLE\_VALUE) {  do {  if (fd.dwFileAttributes & FILE\_ATTRIBUTE\_DIRECTORY)  TRACE (\_T ("%s\n"), fd.cFileName);  } while (::FindNextFile (hFind, &fd));  ::FindClose (hFind);  } |

The more interesting case is how you can enumerate all the directories in a given directory *and its subdirectories*. The following function enumerates all the directories in the current directory and its descendants, writing the name of each directory to the debug output window. The secret? Whenever it encounters a directory, *EnumerateFolders* descends into that directory and calls itself recursively.

|  |
| --- |
| void EnumerateFolders ()  {  WIN32\_FIND\_DATA fd;  HANDLE hFind = ::FindFirstFile (\_T ("\*.\*"), &fd);  if (hFind != INVALID\_HANDLE\_VALUE) {  do {  if (fd.dwFileAttributes & FILE\_ATTRIBUTE\_DIRECTORY) {  CString name = fd.cFileName;  if (name != \_T (".") && name != \_T ("..")) {  TRACE (\_T ("%s\n"), fd.cFileName);  ::SetCurrentDirectory (fd.cFileName);  EnumerateFolders ();  ::SetCurrentDirectory (\_T (".."));  }  }  } while (::FindNextFile (hFind, &fd));  ::FindClose (hFind);  }  } |

To use this function, navigate to the directory in which you want the enumeration to begin and call *EnumerateFolders*. The following statements enumerate all the directories on drive C:

|  |
| --- |
| ::SetCurrentDirectory (\_T ("C:\\"));  EnumerateFolders (); |

We'll use a similar technique in [Chapter 10](mk:@MSITStore:C:\Program%20Files%20(x86)\MSPress\BooksOnline\Programming%20Windows%20with%20MFC%20Second%20Edition\progmfc2.chm::/ch10a.htm) to populate a tree view with items representing all the folders on a drive.

# Serialization and the *CArchive* Class

Although MFC's *CFile* class makes reading and writing file data rather easy, most MFC applications don't interact with *CFile* objects directly. Instead, they do their reading and writing through *CArchive* objects that in turn use *CFile* functions to perform file I/O. MFC overloads the << and >> operators used with *CArchive* to make serializing data to or from a *CArchive* simple. The most common reason for serializing to or from an archive is to save an application's persistent data to disk or to read it back again.

Serialization is an important concept in MFC programming because it is the basis for MFC's ability to open and save documents in document/view applications. As you'll learn in Chapter 9, when someone using a document/view application selects Open or Save from the application's File menu, MFC opens the file for reading or writing and passes the application a reference to a *CArchive* object. The application, in turn, serializes its persistent data to or from the archive and, by so doing, saves a complete document to disk or reads it back again. A document whose persistent data consists entirely of primitive data types or serializable objects can often be serialized with just a few lines of code. This is in contrast to the hundreds of lines that might be required if the application were to query the user for a file name, open the file, and do all the file I/O itself.

## Serialization Basics

Assume that a *CFile* object named *file* represents an open file, that the file was opened with write access, and that you want to write a pair of integers named *a* and *b* to that file. One way to accomplish this is to call *CFile::Write* once for each integer:

|  |
| --- |
| file.Write (&a, sizeof (a));  file.Write (&b, sizeof (b)); |

An alternative method is to create a *CArchive* object, associate it with the *CFile* object, and use the << operator to serialize the integers into the archive:

|  |
| --- |
| CArchive ar (&file, CArchive::store);  ar << a << b; |

*CArchive* objects can be used for reading, too. Assuming *file* once again represents an open file and that the file is open with read access, the following code snippet attaches a *CArchive* object to the file and reads, or *deserializes*, the integers from the file:

|  |
| --- |
| CArchive ar (&file, CArchive::load);  ar >> a >> b; |

MFC allows a wide variety of primitive data types to be serialized this way, including BYTEs, WORDs, LONGs, DWORDs, floats, doubles, ints, unsigned ints, shorts, and chars.

MFC also overrides the << and >> operators so that *CString*s and certain other nonprimitive data types represented by MFC classes can be serialized to or from an archive. If *string* is a *CString* object and *ar* is a *CArchive* object, writing the string to the archive is as simple as this:

|  |
| --- |
| ar << string; |

Turning the operator around reads the string from the archive:

|  |
| --- |
| ar >> string; |

Classes that can be serialized this way include *CString*, *CTime*, *CTimeSpan*, *COleVariant*, *COleCurrency*, *COleDateTime*, *COleDateTimeSpan*, *CSize*, *CPoint*, and *CRect*. Structures of type SIZE, POINT, and RECT can be serialized, too.

Perhaps the most powerful aspect of MFC's serialization mechanism is the fact that you can create serializable classes of your own that work with *CArchive*'s insertion and extraction operators. And you don't have to do any operator overloading of your own to make it work. Why? Because MFC overloads the << and >> operators for pointers to instances of classes derived from *CObject*.

To demonstrate, suppose you've written a drawing program that represents lines drawn by the user with instances of a class named *CLine*. Also suppose that *CLine* is a serializable class that derives, either directly or indirectly, from *CObject*. If *pLines* is an array of *CLine* pointers, *nCount* is an integer that holds the number of pointers in the array, and *ar* is a *CArchive* object, you could archive each and every *CLine* along with a count of the number of *CLine*s like this:

|  |
| --- |
| ar << nCount;  for (int i=0; i<nCount; i++)  ar << pLines[i]; |

Conversely, you could re-create the *CLine*s from the information in the archive and initialize *pLines* with *CLine* pointers with the statements

|  |
| --- |
| ar >> nCount;  for (int i=0; i<nCount; i++)  ar >> pLines[i]; |

How do you write serializable classes like *CLine*? It's easy; the next section describes how.

If an error occurs as data is serialized to or from an archive, MFC throws an exception. The type of exception that's thrown depends on the nature of the error. If a serialization request fails because of a lack of memory (for example, if there's too little memory to create an instance of an object that's being deserialized from an archive), MFC throws a *CMemoryException*. If a request fails because of a file I/O error, MFC throws a *CFileException*. If any other error occurs, MFC throws a *CArchiveException*. If you'd like, you can supply *catch* handlers for exceptions of these types to enact your own special processing regimen if and when errors occur.

## Writing Serializable Classes

For an object to support serialization, it must be an instance of a serializable class. You can write a serializable class by following these five steps:

1. Derive the class, either directly or indirectly, from *CObject*.
2. Include MFC's DECLARE\_SERIAL macro in the class declaration. DECLARE\_SERIAL accepts just one parameter: your class's name.
3. Override the base class's *Serialize* function, and serialize the derived class's data members.
4. If the derived class doesn't have a default constructor (one that takes no arguments), add one. This step is necessary because when an object is deserialized, MFC creates it on the fly using the default constructor and initializes the object's data members with values retrieved from the archive.
5. In the class implementation, include MFC's IMPLEMENT\_SERIAL macro. The IMPLEMENT\_SERIAL macro takes three parameters: the class name, the name of the base class, and a schema number. The *schema number* is an integer value that amounts to a version number. You should change the schema number any time you modify the class's serialized data format. Versioning of serializable classes is discussed in the next section.

Suppose you've written a simple class named *CLine* to represent lines. The class has two *CPoint* data members that store the line's endpoints, and you'd like to add serialization support. Originally, the class declaration looks like this:

|  |
| --- |
| class CLine  {  protected:  CPoint m\_ptFrom;  CPoint m\_ptTo;  public:  CLine (CPoint from, CPoint to) { m\_ptFrom = from; m\_ptTo = to; }  }; |

It's easy to make this class serializable. Here's how it looks after serialization support is added:

|  |
| --- |
| class CLine : public CObject  {  DECLARE\_SERIAL (CLine)  protected:  CPoint m\_ptFrom;  CPoint m\_ptTo;  public:  CLine () {} // Required!  CLine (CPoint from, CPoint to) { m\_ptFrom = from; m\_ptTo = to; }  void Serialize (CArchive& ar);  }; |

The *Serialize* function looks like this:

|  |
| --- |
| void CLine::Serialize (CArchive& ar)  {  CObject::Serialize (ar);  if (ar.IsStoring ())  ar << m\_ptFrom << m\_ptTo;  else // Loading, not storing  ar >> m\_ptFrom >> m\_ptTo;  } |

And somewhere in the class implementation the statement

|  |
| --- |
| IMPLEMENT\_SERIAL (CLine, CObject, 1) |

appears. With these modifications, the class is fully serializable. The schema number is 1, so if you later add a persistent data member to *CLine*, you should bump the schema number up to 2 so that the framework can distinguish between *CLine* objects serialized to disk by different versions of your program. Otherwise, a version 1 *CLine* on disk could be read into a version 2 *CLine* in memory, with possibly disastrous consequences.

When an instance of this class is asked to serialize or deserialize itself, MFC calls the instance's *CLine::Serialize* function. Before serializing its own data members, *CLine::Serialize* calls *CObject::Serialize* to serialize the base class's data members. In this example, the base class's *Serialize* function doesn't do anything, but that might not be the case if the class you're writing derives indirectly from *CObject*. After the call to the base class returns, *CLine::Serialize* calls *CArchive::IsStoring* to determine the direction of data flow. A nonzero return means data is being serialized into the archive; 0 means data is being serialized out. *CLine::Serialize* uses the return value to decide whether to write to the archive with the << operator or to read from it using the >> operator.

## Versioning Serializable Classes: Versionable Schemas

When you write a serializable class, MFC uses the schema number that you assign to enact a crude form of version control. MFC tags instances of the class with the schema number when it writes them to the archive, and when it reads them back, it compares the schema number recorded in the archive to the schema number of the objects of that type in use within the application. If the two numbers don't match, MFC throws a *CArchiveException* with *m\_cause* equal to *CArchiveException::badSchema*. An unhandled exception of this type prompts MFC to display a message box with the warning "Unexpected file format." By incrementing the schema number each time you revise an object's serialized storage format, you create an effective safeguard against inadvertent attempts to read an old version of an object stored on disk into a new version that resides in memory.

One problem that frequently crops up in applications that use serializable classes is one of backward compatibility—that is, deserializing objects that were created with older versions of the application. If an object's persistent storage format changes from one version of the application to the next, you'll probably want the new version to be able to read both formats. But as soon as MFC sees the mismatched schema numbers, it throws an exception. Because of the way MFC is architected, there's no good way to handle the exception other than to do as MFC does and abort the serialization process.

That's where versionable schemas come in. A versionable schema is simply a schema number that includes a VERSIONABLE\_SCHEMA flag. This flag tells MFC that the application can handle multiple serialized data formats for a given class. It suppresses the *CArchiveException* and allows an application to respond intelligently to different schema numbers. An application that uses versionable schemas can provide the backward compatibility that users expect.

Writing a serializable class that takes advantage of MFC's versionable schema support involves two steps:

1. OR the value VERSIONABLE\_SCHEMA into the schema number in the IMPLEMENT\_SERIAL macro.
2. Modify the class's *Serialize* function to call *CArchive::GetObjectSchema* when loading an object from an archive and adapt its deserialization routine accordingly. *GetObjectSchema* returns the schema number of the object that's about to be deserialized.

You need to be aware of a few rules when you use *GetObjectSchema*. First, it should be called only when an object is being deserialized. Second, it should be called before any of the object's data members are read from the archive. And third, it should be called only once. If called a second time in the context of the same call to *Serialize*, *GetObjectSchema* returns -1.

Let's say that in version 2 of your application, you decide to modify the *CLine* class by adding a member variable to hold a line color. Here's the revised class declaration:

|  |
| --- |
| class CLine : public CObject  {  DECLARE\_SERIAL (CLine)  protected:  CPoint m\_ptFrom;  CPoint m\_ptTo;  COLORREF m\_clrLine; // Line color (new in version 2)  public:  CLine () {}  CLine (CPoint from, CPoint to, COLORREF color)  { m\_ptFrom = from; m\_ptTo = to; m\_clrLine = color }  void Serialize (CArchive& ar);  }; |

Because the line color is a persistent property (that is, a red line saved to an archive should still be red when it is read back), you want to modify *CLine::Serialize* to serialize *m\_clrLine* in addition to *m\_ptFrom* and *m\_ptTo*. That means you should bump up *CLine*'s schema number to 2. The original class implementation invoked MFC's IMPLEMENT\_SERIAL macro like this:

|  |
| --- |
| IMPLEMENT\_SERIAL (CLine, CObject, 1) |

In the revised class, however, IMPLEMENT\_SERIAL should be called like this:

|  |
| --- |
| IMPLEMENT\_SERIAL (CLine, CObject, 2 ¦ VERSIONABLE\_SCHEMA) |

When the updated program reads a *CLine* object whose schema number is 1, MFC won't throw a *CArchive* exception because of the VERSIONABLE\_SCHEMA flag in the schema number. But it will know that the two schemas are different because the base schema number was increased from 1 to 2.

You're halfway there. The final step is to modify *CLine::Serialize* so that it deserializes a *CLine* differently depending on the value returned by *GetObjectSchema*. The original *Serialize* function looked like this:

|  |
| --- |
| void CLine::Serialize (CArchive& ar)  {  CObject::Serialize (ar);  if (ar.IsStoring ())  ar << m\_ptFrom << m\_ptTo;  else // Loading, not storing  ar >> m\_ptFrom >> m\_ptTo;  } |

You should implement the new one like this:

|  |
| --- |
| void CLine::Serialize (CArchive& ar)  {  CObject::Serialize (ar);  if (ar.IsStoring ())  ar << m\_ptFrom << m\_ptTo << m\_clrLine;  else {  UINT nSchema = ar.GetObjectSchema ();  switch (nSchema) {  case 1: // Version 1 CLine  ar >> m\_ptFrom >> m\_ptTo;  m\_clrLine = RGB (0, 0, 0); // Default color  break;  case 2: // Version 2 CLine  ar >> m\_ptFrom >> m\_ptTo >> m\_clrLine;  break;  default: // Unknown version  AfxThrowArchiveException (CArchiveException::badSchema);  break;  }  }  } |

See how it works? When a *CLine* object is written *to* the archive, it's always formatted as a version 2 *CLine*. But when a *CLine* is read *from* the archive, it's treated as a version 1 *CLine* or a version 2 *CLine*, depending on the value returned by *GetObjectSchema*. If the schema number is 1, the object is read the old way and *m\_clrLine* is set to a sensible default. If the schema number is 2, all of the object's data members, including *m\_clrLine*, are read from the archive. Any other schema number results in a *CArchiveException* indicating that the version number is unrecognized. (If this occurs, you're probably dealing with buggy code or a corrupted archive.) If, in the future, you revise *CLine* again, you can bump the schema number up to 3 and add a *case* block for the new schema.

## How Serialization Works

Looking under the hood to see what happens when data is serialized to or from an archive provides a revealing glimpse into both the operation and the architecture of MFC. MFC serializes primitive data types such as ints and DWORDs by copying them directly to the archive. To illustrate, here's an excerpt from the MFC source code file Arccore.cpp showing how the *CArchive* insertion operator for DWORDs is implemented:

|  |
| --- |
| CArchive& CArchive::operator<<(DWORD dw)  {  if (m\_lpBufCur + sizeof(DWORD) > m\_lpBufMax)  Flush();  if (!(m\_nMode & bNoByteSwap))  \_AfxByteSwap(dw, m\_lpBufCur);  else  \*(DWORD\*)m\_lpBufCur = dw;  m\_lpBufCur += sizeof(DWORD);  return \*this;  } |

For performance reasons, *CArchive* objects store the data that is written to them in an internal buffer. *m\_lpBufCur* points to the current location in that buffer. If the buffer is too full to hold another DWORD, it is flushed before the DWORD is copied to it. For a *CArchive* object that's attached to a *CFile*, *CArchive::Flush* writes the current contents of the buffer to the file.

*CString*s, *CRect*s, and other nonprimitive data types formed from MFC classes are serialized differently. MFC serializes a *CString*, for example, by outputting a character count followed by the characters themselves. The writing is done with *CArchive::Write*. Here's an excerpt from Arccore.cpp that shows how a *CString* containing less than 255 characters is serialized:

|  |
| --- |
| CArchive& AFXAPI operator<<(CArchive& ar, const CString& string)  {    if (string.GetData()->nDataLength < 255)  {  ar << (BYTE)string.GetData()->nDataLength;  }    ar.Write(string.m\_pchData,  string.GetData()->nDataLength\*sizeof(TCHAR));  return ar;  } |

*CArchive::Write* copies a specified chunk of data to the archive's internal buffer and flushes the buffer if necessary to prevent overflows. Incidentally, if a *CString* serialized into an archive with the << operator contains Unicode characters, MFC writes a special 3-byte signature into the archive before the character count. This enables MFC to identify a serialized string's character type so that, if necessary, those characters can be converted to the format that a client expects when the string is deserialized from the archive. In other words, it's perfectly acceptable for a Unicode application to serialize a string and for an ANSI application to deserialize it, and vice versa.

The more interesting case is what happens when a *CObject* pointer is serialized into an archive. Here's the relevant code from Afx.inl:

|  |
| --- |
| \_AFX\_INLINE CArchive& AFXAPI operator<<(CArchive& ar,  const CObject\* pOb)  { ar.WriteObject(pOb); return ar; } |

As you can see, the << operator calls *CArchive::WriteObject* and passes it the pointer that appears on the right side of the insertion operator—for example, the *pLine* in

|  |
| --- |
| ar << pLine; |

*WriteObject* ultimately calls the object's *Serialize* function to serialize the object's data members, but before it does, it writes additional information to the archive that identifies the class from which the object was created.

For example, suppose the object being serialized is an instance of *CLine*. The very first time it serializes a *CLine* to the archive, *WriteObject* inserts a *new class tag*—a 16-bit integer whose value is -1, or 0xFFFF—into the archive, followed by the object's 16-bit schema number, a 16-bit value denoting the number of characters in the class name, and finally the class name itself. *WriteObject* then calls the *CLine*'s *Serialize* function to serialize the *CLine*'s data members.

If a second *CLine* is written to the archive, *WriteObject* behaves differently. When it writes a new class tag to the archive, *WriteObject* adds the class name to an in-memory database (actually, an instance of *CMapPtrToPtr*) and assigns the class a unique identifier that is in reality an index into the database. If no other classes have been written to the archive, the first *CLine* written to disk is assigned an index of 1. When asked to write a second *CLine* to the archive, *WriteObject* checks the database, sees that *CLine* is already recorded, and instead of writing redundant information to the archive, writes a 16-bit value that consists of the class index ORed with an *old class tag* (0x8000). It then calls the *CLine*'s *Serialize* function as before. Thus, the first instance of a class written to an archive is marked with a new class tag, a schema number, and a class name; subsequent instances are tagged with 16-bit values whose lower 15 bits identify a previously recorded schema number and class name.

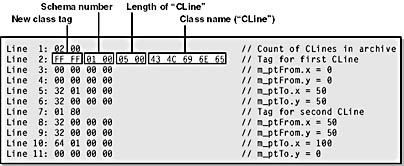
Figure 6-2 shows a hex dump of an archive that contains two serialized version 1 *CLine*s. The *CLine*s were written to the archive with the following code fragment:

|  |
| --- |
| // Create two CLines and initialize an array of pointers.  CLine line1 (CPoint (0, 0), CPoint (50, 50));  CLine line2 (CPoint (50, 50), CPoint (100, 0));  CLine\* pLines[2] = { &line1, &line2 };  int nCount = 2;  // Serialize the CLines and the CLine count.  ar << nCount;  for (int i=0; i<nCount; i++)  ar << pLines[i]; |

The hex dump is broken down so that each line in the listing represents one component of the archive. I've numbered the lines for reference. Line 1 contains the object count (2) written to the archive when the statement

|  |
| --- |
| ar << nCount; |

was executed. Line 2 contains information written by *WriteObject* defining the *CLine* class. The first 16-bit value is the new class tag; the second is the class's schema number (1); and the third holds the length of the class name (5). The final 5 bytes on line 2 hold the class name ("CLine"). Immediately following the class information, in lines 3 through 6, is the first serialized *CLine*: four 32-bit values that specify, in order, the *x* component of the *CLine*'s *m\_ptFrom* data member, the *y* component of *m\_ptFrom*, the *x* component of *m\_ptTo*, and the *y* component of *m\_ptTo*. Similar information for the second *CLine* appears on lines 8 through 11, but in between—on line 7—is a 16-bit tag that identifies the data that follows as a serialized *CLine*. *CLine*'s class index is 1 because it was the first class added to the archive. The 16-bit value 0x8001 is the class index ORed with an old class tag.



**Figure 6-2.** *Hex dump of an archive containing two CLines.*

So far, so good. It's not difficult to understand what goes into the archive. Now let's see what happens when the *CLine*s are read out of the archive. Assume that the *CLine*s are deserialized with the following code:

|  |
| --- |
| int nCount;  ar >> nCount;  CLine\* pLines = new CLine[nCount];  for (int i=0; i<nCount; i++)  ar >> pLines[i]; |

When the

|  |
| --- |
| ar >> nCount; |

statement is executed, *CArchive* reaches into the archive, retrieves 4 bytes, and copies them to *nCount*. That sets the stage for the *for* loop that retrieves *CLine*s from the archive. Each time the

|  |
| --- |
| ar >> pLines[i]; |

statement is executed, the >> operator calls *CArchive::ReadObject* and passes in a NULL pointer. Here's the relevant code in Afx.inl:

|  |
| --- |
| \_AFX\_INLINE CArchive& AFXAPI operator>>(CArchive& ar, CObject\*& pOb)  { pOb = ar.ReadObject(NULL); return ar; }  \_AFX\_INLINE CArchive& AFXAPI operator>>(CArchive& ar,  const CObject\*& pOb)  { pOb = ar.ReadObject(NULL); return ar; } |

*ReadObject* calls another *CArchive* function named *ReadClass* to determine what kind of object it's about to deserialize. The first time through the loop, *ReadClass* reads one word from the archive, sees that it's a new class tag, and proceeds to read the schema number and class name from the archive. *ReadClass* then compares the schema number obtained from the archive to the schema number stored in the *CRuntimeClass* structure associated with the class whose name was just retrieved. (The DECLARE\_SERIAL and IMPLEMENT\_SERIAL macros create a static *CRuntimeClass* structure containing important information about a class, including its name and schema number. MFC maintains a linked list of *CRuntimeClass* structures that can be searched to locate run-time information for a particular class.) If the schemas are the same, *ReadClass* returns the *CRuntimeClass* pointer to *ReadObject*. *ReadObject*, in turn, calls *CreateObject* through the *CRuntimeClass* pointer to create a new instance of the class and then calls the object's *Serialize* function to load the data from the archive into the object's data members. The pointer to the new class instance returned by *ReadClass* is copied to the location specified by the caller—in this case, the address of *pLines*[*i*].

As class information is read from the archive, *ReadObject* builds a class database in memory just as *WriteObject* does. When the second *CLine* is read from the archive, the 0x8001 tag preceding it tells *ReadClass* that it can get the *CRuntimeClass* pointer requested by *ReadObject* from the database.

That's basically what happens during the serialization process if all goes well. I've skipped many of the details, including the numerous error checks MFC performs and the special treatment given to NULL object pointers and multiple references to the same object.

What happens if the schema number read from the archive doesn't match the schema number stored in the corresponding *CRuntimeClass*? Enter versionable schemas. MFC first checks for a VERSIONABLE\_SCHEMA flag in the schema number stored in the *CRuntimeClass*. If the flag is absent, MFC throws a *CArchiveException*. At that point, the serialization process is over; done; finis. There's very little you can do about it other than display an error message, which MFC will do for you if you don't catch the exception. If the VERSIONABLE\_SCHEMA flag is present, however, MFC skips the call to *AfxThrowArchiveException* and stores the schema number where the application can retrieve it by calling *GetObjectSchema*. That's why VERSIONABLE\_SCHEMA and *GetObjectSchema* are the keys that open the door to successful versioning of serializable classes.

## Serializing *CObjects*

I'll close this chapter with a word of advice regarding the serialization of *CObject*s. MFC overloads *CArchive*'s insertion and extraction operators for *CObject* pointers, but not for *CObject*s. That means this will work:

|  |
| --- |
| CLine\* pLine = new CLine (CPoint (0, 0), CPoint (100, 50));  ar << pLine; |

But this won't:

|  |
| --- |
| CLine line (CPoint (0, 0), CPoint (100, 50));  ar << line; |

In other words, *CObject*s can be serialized by pointer but not by value. This normally isn't a problem, but it can be troublesome if you write serializable classes that use other serializable classes as embedded data members and you want to serialize those data members.

One way to serialize *CObject*s by value instead of by pointer is to do your serialization and deserialization like this:

|  |
| --- |
| // Serialize.  CLine line (CPoint (0, 0), CPoint (100, 50));  ar << &line;  // Deserialize.  CLine\* pLine;  ar >> pLine;  CLine line = \*pLine; // Assumes CLine has a copy constructor.  delete pLine; |

The more common approach, however, is to call the other class's *Serialize* function directly, as demonstrated here:

|  |
| --- |
| // Serialize.  CLine line (CPoint (0, 0), CPoint (100, 50));  line.Serialize (ar);  // Deserialize.  CLine line;  line.Serialize (ar); |

Although calling *Serialize* directly is perfectly legal, you should be aware that it means doing without versionable schemas for the object that is being serialized. When you use the << operator to serialize an object pointer, MFC writes the object's schema number to the archive; when you call *Serialize* directly, it doesn't. If called to retrieve the schema number for an object whose schema is not recorded, *GetObjectSchema* will return -1 and the outcome of the deserialization process will depend on how gracefully *Serialize* handles unexpected schema numbers.