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*.

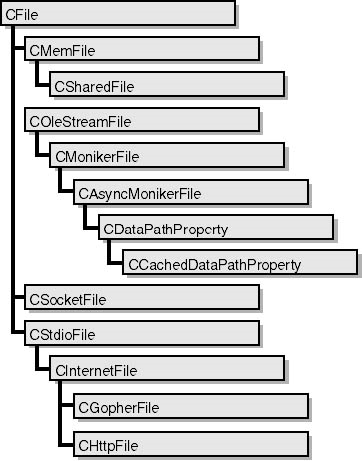
*Read*, *Write*, and other *CFile* functions throw a *CFileException* if an error occurs during a file I/O operation. *CFileException::m\_cause* tells you why the error occurred. For example, attempting to write to a disk that is full throws a *CFileException* with *m\_cause* equal to *CFileException::diskFull*. 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.

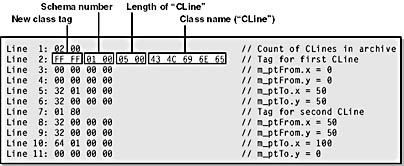
Figure 6-2 shows a hex dump of an archive that contains two serialized version 1 *CLine*s. The *CLine*s were written to the archive with the following code fragment:

|  |
| --- |
| // 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.

## iSerializing *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.

Chapter 7

# Controls

One of the ingredients found in the recipe for nearly every successful Microsoft Windows application is the control. A *control* is a special kind of window designed to convey information to the user or to acquire input. Most controls appear in dialog boxes, but they work just fine in top-level windows and other nondialog windows, too. The push button is one example of a control; the edit control—a simple rectangle for entering and editing text—is another.

Controls reduce the tedium of Windows programming and promote a consistent user interface by providing canned implementations of common user interface elements. Controls are prepackaged objects that come complete with their own window procedures. An application that uses a push button control doesn't have to draw the push button on the screen and process mouse messages to know when the push button is clicked. Instead, it creates the push button with a simple function call and receives notifications whenever the push button is clicked. The control's WM\_PAINT handler paints the push button on the screen, and other message handlers inside the control translate user input into notification messages.

Controls are sometimes referred to as *child window controls* because of the parent-child relationships that they share with other windows. Unlike top-level windows, which have no parents, controls are child windows that are parented to other windows. A child window moves when its parent moves, is automatically destroyed when its parent is destroyed, and is clipped to its parent's window rectangle. And when a control transmits a notification message signifying an input event, its parent is the recipient of that message.

Current versions of Windows come with more than 20 types of controls. Six, which we'll refer to as the *classic controls*, have been around since the first version of Windows and are implemented in User.exe. The others, which are collectively known as the *common controls,* are relatively new to Windows (most debuted in Windows 95) and are implemented in Comctl32.dll. This chapter introduces the classic controls and the MFC classes that encapsulate them. The common controls are covered in [Chapter 16](mk:@MSITStore:C:\Program%20Files%20(x86)\MSPress\BooksOnline\Programming%20Windows%20with%20MFC%20Second%20Edition\progmfc2.chm::/ch16a.htm).

# The Classic Controls

Windows makes the classic controls available to the application programs that it hosts by registering six predefined WNDCLASSes. The control types, their WNDCLASSes, and the corresponding MFC classes are shown in the following table.

**The Classic Controls**

|  |  |  |
| --- | --- | --- |
| ***Control Type*** | ***WNDCLASS*** | ***MFC Class*** |
| Buttons | "BUTTON" | *CButton* |
| List boxes | "LISTBOX" | *CListBox* |
| Edit controls | "EDIT" | *CEdit* |
| Combo boxes | "COMBOBOX" | *CComboBox* |
| Scroll bars | "SCROLLBAR" | *CScrollBar* |
| Static controls | "STATIC" | *CStatic* |

A control is created by instantiating one of the MFC control classes and calling the resulting object's *Create* function. If *m\_wndPushButton* is a *CButton* object, the statement

|  |
| --- |
| m\_wndPushButton.Create (\_T ("Start"), WS\_CHILD ¦ WS\_VISIBLE ¦  BS\_PUSHBUTTON, rect, this, IDC\_BUTTON); |

creates a push button control labeled "Start." The first parameter is the text that appears on the button face. The second is the button style, which is a combination of conventional (WS\_) window styles and button-specific (BS\_) window styles. Together, WS\_CHILD, WS\_VISIBLE, and BS\_PUSHBUTTON create a push button control that is a child of thewindow identified in the fourth parameter and that is visible on the screen. (If you omit WS\_VISIBLE from the window style, the control won't become visible until you call *ShowWindow* on it.). *rect* is a RECT structure or a *CRect* object specifying the control's size and location, in pixels, relative to the upper left corner of its parent's client area. *this* identifies the parent window, and IDC\_BUTTON is an integer value that identifies the control. This value is also known as the *child window ID* or *control ID*. It's important to assign a unique ID to each control you create within a given window so that you can map the control's notification messages to member functions in the parent window class.

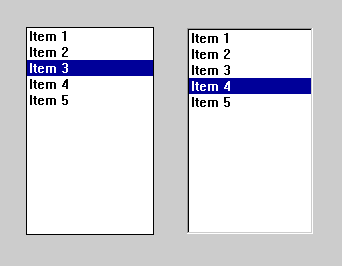
List boxes and edit controls assume a "flat" look when they're created with *Create*. To endow them with the contemporary chiseled look that most users have grown accustomed to (Figure 7-1), you need to create list boxes and edit controls with *CreateEx* instead of *Create* and include a WS\_EX\_CLIENTEDGE flag in the extended style specified in the function's first parameter. If *m\_wndListBox* is a *CListBox* object, the following statement creates a list box with chiseled edges and parents it to the window identified by the *this* pointer:

|  |
| --- |
| m\_wndListBox.CreateEx (WS\_EX\_CLIENTEDGE, \_T ("LISTBOX"), NULL,  WS\_CHILD | WS\_VISIBLE | LBS\_STANDARD, rect, this, IDC\_LISTBOX); |

As an alternative, you can derive your own class from *CListBox*, override *PreCreateWindow* in the derived class, and apply WS\_EX\_CLIENTEDGE to the window style in the *PreCreateWindow*, as demonstrated here:

|  |
| --- |
| BOOL CMyListBox::PreCreateWindow (CREATESTRUCT& cs)  {  if (!CListBox::PreCreateWindow (cs))  return FALSE;  cs.dwExStyle |= WS\_EX\_CLIENTEDGE;    return TRUE;  } |

With *PreCreateWindow* implemented like this, a *CMyListBox* object will have chiseled borders regardless of how it's created.



**Figure 7-1.** *A list box with flat edges (left) and chiseled edges (right).*

A control sends notifications to its parent in the form of WM\_COMMAND messages. The kinds of notifications that are sent vary with the control type, but in each case, information encoded in the message's *wParam* and *lParam* parameters identifies the control that sent the message and the action that prompted the message. For example, the WM\_COMMAND message sent when a push button is clicked contains the notification code BN\_CLICKED in the upper 16 bits of *wParam*, the control ID in the lower 16 bits of *wParam*, and the control's window handle in *lParam*.

Rather than process raw WM\_COMMAND messages, most MFC applications use message maps to link control notifications to class member functions. For example, the following message-map entry maps clicks of the push button whose control ID is IDC\_BUTTON to the member function *OnButtonClicked*:

|  |
| --- |
| ON\_BN\_CLICKED (IDC\_BUTTON, OnButtonClicked) |

ON\_BN\_CLICKED is one of several control-related message-map macros that MFC provides. For example, there are ON\_EN macros for edit controls and ON\_LBN macros for list box controls. There's also the generic ON\_CONTROL macro, which handles all notifications and all control types, and ON\_CONTROL\_RANGE, which maps identical notifications from two or more controls to a common notification handler.

Controls send messages to their parents, but it's no less common for parents to send messages to controls. For example, a check mark is placed in a check box control by sending the control a BM\_SETCHECK message with *wParam* equal to BST\_CHECKED. MFC simplifies message-based control interfaces by building member functions into its control classes that wrap BM\_SETCHECK and other control messages. For example, the statement

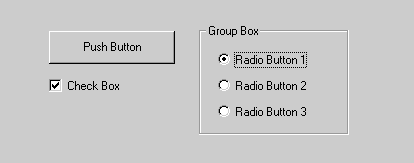
|  |
| --- |
| m\_wndCheckBox.SetCheck (BST\_CHECKED); |

places a check mark inside a check box represented by a *CButton* object named *m\_wndCheckBox*.

Because a control is a window, some of the member functions that the control classes inherit from *CWnd* are useful for control programming. For example, the same *SetWindowText* function that changes the text in a window's title bar inserts text into an edit control, too. Other useful *CWnd* functions include *GetWindowText*, which retrieves text from a control; *EnableWindow*, which enables and disables a control; and *SetFont*, which changes a control's font. If you want to do something to a control and can't find an appropriate member function in the control class, check *CWnd*'s list of member functions. You'll probably find the one you're looking for.

## The *CButton* Class

*CButton* represents button controls based on the "BUTTON" WNDCLASS. Button controls come in four flavors: push buttons, check boxes, radio buttons, and group boxes. All four button types are shown in Figure 7-2.



**Figure 7-2.** *The four types of button controls.*

When you create a button control, you specify which of the four button types you want to create by including one of the following flags in the button's window style:

|  |  |
| --- | --- |
| ***Style*** | ***Description*** |
| BS\_PUSHBUTTON | Creates a standard push button control |
| BS\_DEFPUSHBUTTON | Creates a default push button; used in dialog boxes to identify the push button that's clicked if Enter is pressed |
| BS\_CHECKBOX | Creates a check box control |
| BS\_AUTOCHECKBOX | Creates a check box control that checks and unchecks itself when clicked |
| BS\_3STATE | Creates a three-state check box control |
| BS\_AUTO3STATE | Creates a three-state check box control that cycles through three states—checked, unchecked, and indeterminate—when clicked |
| BS\_RADIOBUTTON | Creates a radio button control |
| BS\_AUTORADIOBUTTON | Creates a radio button control that, when clicked, checks itself and unchecks other radio buttons in the group |
| BS\_GROUPBOX | Creates a group box control |

In addition, you can OR one or more of the following values into the window style to control the alignment of the text on the button face:

|  |  |
| --- | --- |
| ***Style*** | ***Description*** |
| BS\_LEFTTEXT | Moves the text accompanying a radio button or check box control from the button's right (the default) to its left |
| BS\_RIGHTBUTTON | Same as BS\_LEFTTEXT |
| BS\_LEFT | Left justifies the button text in the control rectangle |
| BS\_CENTER | Centers the button text in the control rectangle |
| BS\_RIGHT | Right justifies the button text in the control rectangle |
| BS\_TOP | Positions the button text at the top of the control rectangle |
| BS\_VCENTER | Positions the button text in the center of the control rectangle vertically |
| BS\_BOTTOM | Positions the button text at the bottom of the control rectangle |
| BS\_MULTILINE | Allows text too long to fit on one line to be broken into two or more lines |

There are other button styles, but most of them are rarely used. For example, BS\_NOTIFY programs a button to send BN\_DOUBLECLICKED, BN\_KILLFOCUS, and BN\_SETFOCUS notifications. BS\_OWNERDRAW creates an *owner-draw* button—one whose appearance is maintained by the button's parent rather than the button itself. Owner-draw buttons have been largely superseded by bitmap buttons and icon buttons. You'll learn more about bitmap buttons and icon buttons later in this chapter.

### Push Buttons

A push button is a button control created with the style BS\_PUSHBUTTON. When clicked, a push button control sends its parent a BN\_CLICKED notification encapsulated in a WM\_COMMAND message. Absent the button style BS\_NOTIFY, a push button sends no other types of notifications.

MFC's ON\_BN\_CLICKED macro links BN\_CLICKED notifications to member functions in the parent window class. The message-map entry

|  |
| --- |
| ON\_BN\_CLICKED (IDC\_BUTTON, OnButtonClicked) |

connects *OnButtonClicked* to clicks of the push button whose control ID is IDC\_BUTTON. A trivial implementation of *OnButtonClicked* looks like this:

|  |
| --- |
| void CMainWindow::OnButtonClicked ()  {  MessageBox (\_T ("I've been clicked!"));  } |

Like command handlers for menu items, BN\_CLICKED handlers accept no parameters and return no values.

### Check Boxes

Check boxes are buttons created with the style BS\_CHECKBOX, BS\_AUTOCHECKBOX, BS\_3STATE, or BS\_AUTO3STATE. BS\_CHECKBOX and BS\_AUTOCHECKBOX check boxes can assume two states: checked and unchecked. A check box is checked and unchecked with *CButton::SetCheck*:

|  |
| --- |
| m\_wndCheckBox.SetCheck (BST\_CHECKED); // Check  m\_wndCheckBox.SetCheck (BST\_UNCHECKED); // Uncheck |

To find out whether a check box is checked, use *CButton::GetCheck*. A return value equal to BST\_CHECKED means the box is checked. BST\_UNCHECKED means it's not.

Like push button controls, check boxes send BN\_CLICKED notifications to their parents when clicked. The check mark in a BS\_AUTOCHECKBOX check box toggles on and off automatically in response to button clicks. The check mark in a BS\_CHECKBOX check box doesn't. Therefore, BS\_CHECKBOX-style check boxes are of little use unless you write BN\_CLICKED handlers to go with them. The following BN\_CLICKED handler toggles *m\_wndCheckBox*'s check mark on and off:

|  |
| --- |
| void CMainWindow::OnCheckBoxClicked ()  {  m\_wndCheckBox.SetCheck (m\_wndCheckBox.GetCheck () ==  BST\_CHECKED ? BST\_UNCHECKED : BST\_CHECKED);  } |

The BS\_3STATE and BS\_AUTO3STATE button styles create a check box that can assume a third state in addition to the checked and unchecked states. The third state is called the *indeterminate* state and is entered when the user clicks a BS\_AUTO3STATE check box that is currently checked or when *SetCheck* is called with a BST\_INDETERMINATE parameter:

|  |
| --- |
| m\_wndCheckBox.SetCheck (BST\_INDETERMINATE); |

An indeterminate check box contains a grayed check mark. The indeterminate state is used to indicate that something is neither wholly on nor wholly off. For example, a word processing program might set a check box labeled "Bold" to the indeterminate state when the user selects a mix of normal and boldface text.

### Radio Buttons

A radio button is a button control with the style BS\_RADIOBUTTON or BS\_AUTORADIOBUTTON. Radio buttons normally come in groups, with each button representing one in a list of mutually exclusive options. When clicked, a BS\_AUTORADIOBUTTON radio button checks itself *and* unchecks the other radio buttons in the group. If you use the BS\_RADIOBUTTON style instead, it's up to you to do the checking and unchecking using *CButton::SetCheck*.

Radio buttons send BN\_CLICKED notifications to their parents, just as push buttons and check boxes do. The following BN\_CLICKED handler checks the *m\_wndRadioButton1* radio button and unchecks three other radio buttons in the same group:

|  |
| --- |
| void CMainWindow::OnRadioButton1Clicked ()  {  m\_wndRadioButton1.SetCheck (BST\_CHECKED);  m\_wndRadioButton2.SetCheck (BST\_UNCHECKED);  m\_wndRadioButton3.SetCheck (BST\_UNCHECKED);  m\_wndRadioButton4.SetCheck (BST\_UNCHECKED);  } |

Unchecking the other radio buttons maintains the exclusivity of the selection. A BN\_CLICKED handler isn't necessary for BS\_AUTORADIOBUTTON radio buttons, though you can still provide one if you want to respond to changes in a radio button's state at the instant the button is clicked.

For BS\_AUTORADIOBUTTON radio buttons to properly deselect the other buttons in the group, you must group the buttons so that Windows knows which buttons belong to the group. To create a group of BS\_AUTORADIOBUTTON radio buttons, follow this procedure:

1. In your application's code, create the buttons in sequence, one after another; don't create any other controls in between.
2. To mark the beginning of the group, assign the style WS\_GROUP to the first radio button you create.
3. If you create additional controls after the last radio button is created, assign the WS\_GROUP style to the first additional control that you create. This implicitly marks the previous control (the last radio button) as the final one in the group. If there are no other controls after the radio buttons but there are other controls in the window, mark the first control with WS\_GROUP to prevent the radio button group from wrapping around.

The following example demonstrates how to create four BS\_AUTORADIOBUTTON radio buttons belonging to one group and three belonging to another group, with a check box control in between:

|  |
| --- |
| m\_wndRadioButton1.Create (\_T ("COM1"), WS\_CHILD ¦ WS\_VISIBLE ¦  WS\_GROUP ¦ BS\_AUTORADIOBUTTON, rect1, this, IDC\_COM1);  m\_wndRadioButton2.Create (\_T ("COM2"), WS\_CHILD ¦ WS\_VISIBLE ¦  BS\_AUTORADIOBUTTON, rect2, this, IDC\_COM2);  m\_wndRadioButton3.Create (\_T ("COM3"), WS\_CHILD ¦ WS\_VISIBLE ¦  BS\_AUTORADIOBUTTON, rect3, this, IDC\_COM3);  m\_wndRadioButton4.Create (\_T ("COM4"), WS\_CHILD ¦ WS\_VISIBLE ¦  BS\_AUTORADIOBUTTON, rect4, this, IDC\_COM4);  m\_wndRadioButton1.SetCheck (BST\_CHECKED);  m\_wndCheckBox.Create (\_T ("Save settings on exit"),  WS\_CHILD ¦ WS\_VISIBLE ¦ WS\_GROUP ¦ BS\_AUTOCHECKBOX,  rectCheckBox, this, IDC\_SAVESETTINGS);  m\_wndRadioButton5.Create (\_T ("9600"), WS\_CHILD ¦ WS\_VISIBLE ¦  WS\_GROUP ¦ BS\_AUTORADIOBUTTON, rect5, this, IDC\_9600);  m\_wndRadioButton6.Create (\_T ("14400"), WS\_CHILD ¦ WS\_VISIBLE ¦  BS\_AUTORADIOBUTTON, rect6, this, IDC\_14400);  m\_wndRadioButton7.Create (\_T ("28800"), WS\_CHILD ¦ WS\_VISIBLE ¦  BS\_AUTORADIOBUTTON, rect7, this, IDC\_28800);  m\_wndRadioButton5.SetCheck (BST\_CHECKED); |

Because of the BS\_AUTORADIOBUTTON styles and the logical grouping provided by the WS\_GROUP bits, checking any of the first four radio buttons automatically unchecks the other three in the group, and checking any radio button in the second group automatically unchecks the other two.

For good form, the code above calls *SetCheck* to check a button in each group. One of the buttons in a group of radio buttons should always be checked, even if the user has yet to provide any input. Radio buttons are never checked by default, so it's your responsibility to do the initializing.

### Group Boxes

A group box is a button control created with the style BS\_GROUPBOX. It is unlike other button controls in that it never receives the input focus and never sends notifications to its parent.

The sole function of the group box is to visually delineate control groups. Enclosing groups of controls in group boxes makes it apparent to the user which controls go together. Group boxes have nothing to do with the *logical* grouping of controls, so don't expect a series of radio buttons to function as a group simply because there's a group box around them.

## The *CListBox* Class

MFC's *CListBox* class encapsulates list box controls, which display lists of text strings called *items*. A list box optionally sorts the items that are added to it, and scrolling is built in so that the number of items a list box can display isn't limited by the physical dimensions of the list box window.

List boxes are extremely useful for presenting lists of information and allowing users to select items from those lists. When an item is clicked or double-clicked, most list boxes (technically, those with LBS\_NOTIFY in their window styles) notify their parents with WM\_COMMAND messages. MFC simplifies the processing of these messages by providing ON\_LBN message-map macros that you can use to route list box notifications to handling functions in the parent window class.

A standard list box displays text strings in a vertical column and allows only one item to be selected at a time. The currently selected item is highlighted with the system color COLOR\_HIGHLIGHT. Windows supports a number of variations on the standard list box, including multiple-selection list boxes, multicolumn list boxes, and owner-draw list boxes that display images instead of text.

### Creating a List Box

The following statement creates a standard list box from a *CListBox* object named *m\_wndListBox*:

|  |
| --- |
| m\_wndListBox.Create (WS\_CHILD ¦ WS\_VISIBLE ¦ LBS\_STANDARD,  rect, this, IDC\_LISTBOX); |

LBS\_STANDARD combines the styles WS\_BORDER, WS\_VSCROLL, LBS\_NOTIFY, and LBS\_SORT to create a list box that has a border and a vertical scroll bar, that notifies its parent when the selection changes or an item is double-clicked, and that alphabetically sorts the strings that are added to it. By default, the scroll bar is visible only when the number of items in the list box exceeds the number that can be displayed. To make the scroll bar visible at all times, include the style LBS\_DISABLENOSCROLL. A list box doesn't have a vertical scroll bar unless the style WS\_VSCROLL or LBS\_STANDARD is included. Similarly, it doesn't have a border unless it is created with the style WS\_BORDER or LBS\_STANDARD. You might want to omit the border if you create a list box that encompasses the entire client area of its parent. These and other styles used to customize a list box's appearance and behavior are summarized in the table below.

List boxes have keyboard interfaces built in. When a single-selection list box has the input focus, the up arrow, down arrow, Page Up, Page Down, Home, and End keys move the highlighted bar identifying the current selection. In addition, pressing a character key moves the selection to the next item beginning with that character. Keyboard input works in multiple-selection list boxes, too, but it's the position of a dotted focus rectangle, not the selection, that changes. Pressing the spacebar toggles the selection state of the item with the focus in a multiple-selection list box.

You can customize a list box's keyboard interface by including the LBS\_WANTKEYBOARDINPUT style and processing WM\_VKEYTOITEM and WM\_CHARTOITEM messages. An MFC application can map these messages to *OnVKeyToItem* and *OnCharToItem* handlers using the ON\_WM\_VKEYTOITEM and ON\_WM\_CHARTOITEM macros. A derived list box class can handle these messages itself by overriding the virtual *CListBox::VKeyToItem* and *CListBox::CharToItem* functions. One use for this capability is to create a self-contained list box class that responds to presses of Ctrl-D by deleting the item that is currently selected.

**List Box Styles**

|  |  |
| --- | --- |
| ***Style*** | ***Description*** |
| LBS\_STANDARD | Creates a "standard" list box that has a border and a vertical scroll bar, notifies its parent window when the selection changes or an item is double-clicked, and sorts items alphabetically. |
| LBS\_SORT | Sorts items that are added to the list box. |
| LBS\_NOSEL | Creates a list box whose items can be viewed but not selected. |
| LBS\_NOTIFY | Creates a list box that notifies its parent when the selection changes or an item is double-clicked. |
| LBS\_DISABLENOSC ROLL | Disables the list box's scroll bar when it isn't needed. Without this style, an unneeded scroll bar is hidden rather than disabled. |
| LBS\_MULTIPLESEL | Creates a multiple-selection list box. |
| LBS\_EXTENDEDSEL | Adds extended selection support to a multiple-selection list box. |
| LBS\_MULTICOLUMN | Creates a multicolumn list box. |
| LBS\_OWNERDRAWVARIABLE | Creates an owner-draw list box whose items can vary in height. |
| LBS\_OWNERDRAWFIXED | Creates an owner-draw list box whose items are the same height. |
| LBS\_USETABSTOPS | Configures the list box to expand tab characters in item text. |
| LBS\_NOREDRAW | Creates a list box that doesn't automatically redraw itself when an item is added or removed. |
| LBS\_HASSTRINGS | Creates a list box that "remembers" the strings added to it. Conventional list boxes have this style by default; owner-draw list boxes don't. |
| LBS\_WANTKEYBOARDINPUT | Creates a list box that sends its parent a WM\_VKEYTOITEM or WM\_CHARTOITEM message when a key is pressed. This style is used to customize the list box's response to keyboard input. |
| LBS\_NOINTEGRALHEIGHT | Allows a list box to assume any height. By default, Windows sets a list box's height to a multiple of the item height to prevent items from being partially clipped. |

Because the default font that Windows uses for list boxes is proportionally spaced, it is virtually impossible to line up columns of information in a list box by separating them with space characters. One way to create a columnar list box display is to use *SetFont* to apply a fixed-pitch font to the list box. A better solution is to assign the list box the style LBS\_USETABSTOPS and separate columns of information with tab characters. An LBS\_USETABSTOPS list box treats tab characters the way a word processor does, automatically advancing to the next tab stop when a tab character is encountered. By default, tab stops are evenly spaced about eight character widths apart. You can change the default tab stop settings with the *CListBox::SetTabStops* function. *SetTabStops* measures distances in *dialog units*. One dialog unit is approximately equal to one-fourth the width of a character in the system font. The statement

|  |
| --- |
| m\_wndListBox.SetTabStops (64); |

sets the space between tab stops to 64 dialog units, and

|  |
| --- |
| int nTabStops[] = { 32, 48, 64, 128 };  m\_wndListBox.SetTabStops (4, nTabStops); |

places tab stops at 32, 48, 64, and 128 dialog units from the left margin.

By default, a list box repaints itself whenever an item is added or removed. Usually that's just what you want, but if you're adding hundreds or perhaps thousands of items in rapid-fire fashion, the repeated repaints produce an unsightly flashing effect and slow down the insertion process. You can use LBS\_NOREDRAW to create a list box that doesn't automatically repaint itself. Such a list box will be repainted only when its client area is invalidated.

An alternative to using LBS\_NOREDRAW is to disable redraws before beginning a lengthy insertion process and to reenable them after the last item is inserted. You can enable and disable redraws programmatically by sending a list box WM\_SETREDRAW messages, as shown here:

|  |
| --- |
| m\_wndListBox.SendMessage (WM\_SETREDRAW, FALSE, 0); // Disable redraws.    m\_wndListBox.SendMessage (WM\_SETREDRAW, TRUE, 0); // Enable redraws. |

A list box is automatically repainted when redraws are enabled with WM\_SETREDRAW, so it's not necessary to follow up with a call to *Invalidate*.

Unless a list box is created with the style LBS\_MULTIPLESEL, only one item can be selected at a time. In a single-selection list box, clicking an unselected item both selects that item and deselects the one that was formerly selected. In a multiple-selection list box, any number of items can be selected. Most multiple-selection list boxes are also created with the style LBS\_EXTENDEDSEL, which enables extended selections. In an extended-selection list box, the user selects the first item by clicking it and selects subsequent items by clicking with the Ctrl key pressed. In addition, the user can select entire ranges of contiguous items by clicking the first item in the range and then clicking the last item in the range with the Shift key held down. The Ctrl and Shift keys can be combined to select multiple items and ranges, the net result being a handy interface for selecting arbitrary combinations of items.

The LBS\_MULTICOLUMN style creates a multicolumn list box. Multicolumn list boxes are usually created with the WS\_HSCROLL style so that their contents can be scrolled horizontally if not all the items can be displayed at once. (Multicolumn list boxes can't be scrolled vertically.) You can adjust the column width with the *CListBox::SetColumnWidth* function. Normally, the column width should be based on the average width of a character in the list box font. The default column width is enough to display about 16 characters in the default list box font, so if you'll be inserting strings longer than that, you should expand the column width to prevent columns from overlapping.

### Adding and Removing Items

A list box is empty until items are added to it. Items are added with *CListBox::AddString* and *CListBox::InsertString*. The statement

|  |
| --- |
| m\_wndListBox.AddString (string); |

adds the *CString* object named *string* to the list box. If the list box style includes LBS\_SORT, the string is positioned according to its lexical value; otherwise, it is added to the end of the list. *InsertString* adds an item to the list box at a location specified by a 0-based index. The statement

|  |
| --- |
| m\_wndListBox.InsertString (3, string); |

inserts *string* into the list box and makes it the fourth item. LBS\_SORT has no effect on strings added with *InsertString*.

Both *AddString* and *InsertString* return a 0-based index specifying the string's position in the list box. If either function fails, it returns LB\_ERRSPACE to indicate that the list box is full or LB\_ERR to indicate that the insertion failed for other reasons. You shouldn't see the LB\_ERRSPACE return value very often in 32-bit Windows because the capacity of a list box is limited only by available memory. *CListBox::GetCount* returns the number of items in a list box.

*CListBox::DeleteString* removes an item from a list box. It takes a single parameter: the index of the item to be removed. It returns the number of items remaining in the list box. To remove all items from a list box at once, use *CListBox::ResetContent*.

If desired, you can use *CListBox::SetItemDataPtr* or *CListBox::SetItemData* to associate a 32-bit pointer or a DWORD value with an item in a list box. A pointer or DWORD associated with an item can be retrieved with *CListBox::GetItemDataPtr* or *CListBox::GetItemData*. One use for this feature is to associate extra data with the items in a list box. For example, you could associate a data structure containing an address and a phone number with a list box item holding a person's name. Because *GetItemDataPtr* returns a pointer to a void data type, you'll need to cast the pointer that it returns.

Another technique programmers use to associate extra data—particularly text-based data—with list box items is to create an LBS\_USETABSTOPS-style list box, set the first tab stop to a position beyond the list box's right border, and append a string consisting of a tab character followed by the extra data to the list box item. The text to the right of the tab character will be invisible, but *CListBox::GetText* will return the full text of the list box item—additional text included.

### Finding and Retrieving Items

The *CListBox* class also includes member functions for getting and setting the current selection and for finding and retrieving items. *CListBox::GetCurSel* returns the 0-based index of the item that is currently selected. A return value equal to LB\_ERR means that nothing is selected. *GetCurSel* is often called following a notification signifying that the selection changed or an item was double-clicked. A program can set the current selection with the *SetCurSel* function. Passing *SetCurSel* the value -1 deselects all items, causing the bar highlighting the current selection to disappear from the list box. To find out whether a particular item is selected, use *CListBox::GetSel*.

*SetCurSel* identifies an item by its index, but items can also be selected by content. *CListBox::SelectString* searches a single-selection list box for an item that begins with a specified text string and selects the item if a match is found. The statement

|  |
| --- |
| m\_wndListBox.SelectString (-1, \_T ("Times")); |

starts the search with the first item in the list box and highlights the first item that begins with "Times"—for example, "Times New Roman" or "Times Roman." The search is not case-sensitive. The first parameter to *SelectString* specifiesthe index of the item before the one at which the search begins; -1 instructs the list box to start with item 0. If the search is begun anywhere else, the search will wrap around to the first item if necessary so that all list box items are searched.

To search a list box for a particular item without changing the selection, use *CListBox::FindString* or *CListBox::FindStringExact*. *FindString* performs a string search on a list box's contents and returns the index of the first item whose text matches or begins with a specified string. A return value equal to LB\_ERR means that no match was found. *FindStringExact* does the same but reports a match only if the item text matches the search text exactly. Once you have an item's index in hand, you can retrieve the text of the item with *CListBox::GetText*. The following statements query the list box for the currently selected item and copy the text of that item to a *CString* named *string*:

|  |
| --- |
| CString string;  int nIndex = m\_wndListBox.GetCurSel ();  if (nIndex != LB\_ERR)  m\_wndListBox.GetText (nIndex, string); |

An alternative form of *GetText* accepts a pointer to a character array rather than a *CString* reference. You can use *CListBox::GetTextLen* to determine how large the array should be before calling the array version of *GetText*.

Selections in multiple-selection list boxes are handled differently than selections in single-selection list boxes. In particular, the *GetCurSel*, *SetCurSel*, and *SelectString* functions don't work with multiple-selection list boxes. Instead, items are selected (and deselected) with the *SetSel* and *SelItemRange* functions. The following statements select items 0, 5, 6, 7, 8, and 9 and deselect item 3:

|  |
| --- |
| m\_wndListBox.SetSel (0);  m\_wndListBox.SelItemRange (TRUE, 5, 9);  m\_wndListBox.SetSel (3, FALSE); |

*CListBox* also provides the *GetSelCount* function for getting a count of selected items and the *GetSelItems* function for retrieving the indexes of all selected items. In a multiple-selection list box, the dotted rectangle representing the item with the focus can be moved without changing the current selection. The focus rectangle can be moved and queried with *SetCaretIndex* and *GetCaretIndex*. Most other list box functions, including *GetText*, *GetTextLength*, *FindString*, and *FindStringExact*, work the same for multiple-selection list boxes as they do for the single-selection variety.

### List Box Notifications

A list box sends notifications to its parent window via WM\_COMMAND messages. In an MFC application, list box notifications are mapped to class member functions with ON\_LBN message-map entries. The table below lists the six notification types and the corresponding ON\_LBN macros. LBN\_DBLCLK, LBN\_SELCHANGE, and LBN\_SELCANCEL notifications are sent only if the list box was created with the style LBS\_NOTIFY or LBS\_STANDARD. The others are sent regardless of list box style.

**List Box Notifications**

|  |  |  |  |
| --- | --- | --- | --- |
| ***Notification*** | ***Sent When*** | ***Message-Map Macro*** | ***LBS\_NOTIFY Required?*** |
| LBN\_SETFOCUS | The list box gains the input focus. | ON\_LBN\_SETFOCUS | No |
| LBN\_KILLFOCUS | The list box loses the input focus. | ON\_LBN\_KILLFOCUS | No |
| LBN\_ERRSPACE | An operation failed because of insufficient memory. | ON\_LBN\_ERRSPACE | No |
| LBN\_DBLCLK | An item is double-clicked. | ON\_LBN\_DBLCLK | Yes |
| LBN\_SELCHANGE | The selection changes. | ON\_LBN\_SELCHANGE | Yes |
| LBN\_SELCANCEL | The selection is canceled. | ON\_LBN\_SELCANCEL | Yes |

The two list box notifications that programmers rely on most are LBN\_DBLCLK and LBN\_SELCHANGE. LBN\_DBLCLK is sent when a list box item is double-clicked. To determine the index of the item that was double-clicked in a single-selection list box, use *CListBox::GetCurSel*. The following code fragment displays the item in a message box:

|  |
| --- |
| // In CMainWindow's message map  ON\_LBN\_DBLCLK (IDC\_LISTBOX, OnItemDoubleClicked)    void CMainWindow::OnItemDoubleClicked ()  {  CString string;  int nIndex = m\_wndListBox.GetCurSel ();  m\_wndListBox.GetText (nIndex, string);  MessageBox (string);  } |

For a multiple-selection list box, use *GetCaretIndex* instead of *GetCurSel* to determine which item was double-clicked.

A list box sends an LBN\_SELCHANGE notification when the user changes the selection, but not when the selection is changed programmatically. A single-selection list box sends an LBN\_SELCHANGE notification when the selection moves because of a mouse click or keystroke. A multiple-selection list box sends an LBN\_SELCHANGE notification when an item is clicked, when an item's selection state is toggled with the spacebar, and when the focus rectangle is moved.

## The *CStatic* Class

*CStatic*, which represents static controls created from the "STATIC" WNDCLASS, is the simplest of the MFC control classes. At least it *used* to be: Windows 95 added so many new features to static controls that *CStatic* now rivals *CButton* and some of the other control classes for complexity.

Static controls come in three flavors: text, rectangles, and images. Static text controls are often used to label other controls. The following statement creates a static text control that displays the string "Name":

|  |
| --- |
| m\_wndStatic.Create (\_T ("Name"), WS\_CHILD ¦ WS\_VISIBLE ¦ SS\_LEFT,  rect, this, IDC\_STATIC); |

SS\_LEFT creates a static text control whose text is left-aligned. If the control text is too long to fit on one line, it wraps around to the next one. To prevent wrapping, use SS\_LEFTNOWORDWRAP instead of SS\_LEFT. Text can be centered horizontally or right-aligned in a static control by substituting SS\_CENTER or SS\_RIGHT for SS\_LEFT or SS\_LEFTNOWORDWRAP. Another alternative is the little-used SS\_SIMPLE style, which is similar to SS\_LEFT but creates a control whose text can't be altered with *CWnd::SetWindowText*.

By default, the text assigned to a static text control is aligned along the upper edge of the control rectangle. To center text vertically in the control rectangle, OR an SS\_CENTERIMAGE flag into the control style. You can also draw a sunken border around a static control by including the style SS\_SUNKEN.

A second use for static controls is to draw rectangles. The control style specifies the type of rectangle that is drawn. Here are the styles you can choose from:

|  |  |
| --- | --- |
| ***Style*** | ***Description*** |
| SS\_BLACKFRAME | Hollow rectangle painted in the system color COLOR\_WINDOWFRAME (default = black) |
| SS\_BLACKRECT | Solid rectangle painted in the system color COLOR\_WINDOWFRAME (default = black) |
| SS\_ETCHEDFRAME | Hollow rectangle with etched borders |
| SS\_ETCHEDHORZ | Hollow rectangle with etched top and bottom borders |
| SS\_ETCHEDVERT | Hollow rectangle with etched left and right borders |
| SS\_GRAYFRAME | Hollow rectangle painted in the system color COLOR\_BACKGROUND (default = gray) |
| SS\_GRAYRECT | Solid rectangle painted in the system color COLOR\_BACKGROUND (default = gray) |
| SS\_WHITEFRAME | Hollow rectangle painted in the system color COLOR\_WINDOW (default = white) |
| SS\_WHITERECT | Solid rectangle painted in the system color COLOR\_WINDOW (default = white) |

The statement

|  |
| --- |
| m\_wndStatic.Create (\_T (""), WS\_CHILD ¦ WS\_VISIBLE ¦ SS\_ETCHEDFRAME,  rect, this, IDC\_STATIC); |

creates a static control that resembles a group box. For best results, you should draw etched rectangles on surfaces whose color is the same as the default dialog box color (the system color COLOR\_3DFACE). A static rectangle control doesn't display text, even if you specify a nonnull text string in the call to *Create*.

A third use for static controls is to display images formed from bitmaps, icons, cursors, or GDI metafiles. A static image control uses one of the following styles:

|  |  |
| --- | --- |
| ***Style*** | ***Description*** |
| SS\_BITMAP | A static control that displays a bitmap |
| SS\_ENHMETAFILE | A static control that displays a metafile |
| SS\_ICON | A static control that displays an icon or a cursor |

After creating an image control, you associate a bitmap, metafile, icon, or cursor with it by calling its *SetBitmap*, *SetEnhMetaFile*, *SetIcon*, or *SetCursor* function. The statements

|  |
| --- |
| m\_wndStatic.Create (\_T (""), WS\_CHILD ¦ WS\_VISIBLE ¦ SS\_ICON,  rect, this, IDC\_STATIC);  m\_wndStatic.SetIcon (hIcon); |

create a static control that displays an icon and assign it the icon whose handle is *hIcon*. By default, the icon image is positioned in the upper left corner of the control, and if the image is larger than the control rectangle, the rectangle is automatically expanded so the image won't be clipped. To center the image in the control rectangle, OR SS\_CENTERIMAGE into the control style. SS\_CENTERIMAGE prevents the system from automatically sizing the control rectangle if it's too small to show the entire image, so if you use SS\_CENTERIMAGE, be sure that the control rectangle is large enough to display the image. Sizing isn't an issue with SS\_ENHMETAFILE-style controls because metafile images scale to match the control size. For a neat special effect, place a sunken border around an image control by ORing SS\_SUNKEN into the control style.

By default, a static control sends no notifications to its parent. But a static control created with the SS\_NOTIFY style sends the four types of notifications listed in the following table.

**Static Control Notifications**

|  |  |  |
| --- | --- | --- |
| ***Notification*** | ***Sent When*** | ***Message-Map Macro*** |
| STN\_CLICKED | The control is clicked. | ON\_STN\_CLICKED |
| STN\_DBLCLK | The control is double-clicked. | ON\_STN\_DBLCLK |
| STN\_DISABLE | The control is disabled. | ON\_STN\_DISABLE |
| STN\_ENABLE | The control is enabled. | ON\_STN\_ENABLE |

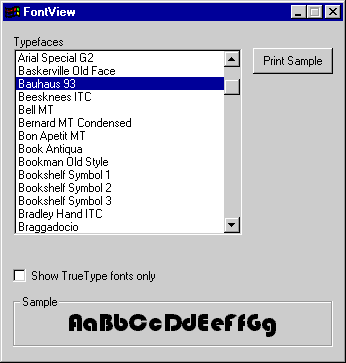
The STN\_CLICKED and STN\_DBLCLK notifications allow you to create static controls that respond to mouse clicks. The statements

|  |
| --- |
| // In CMainWindow's message map  ON\_STN\_CLICKED (IDC\_STATIC, OnClicked)    // In CMainWindow::OnCreate  m\_wndStatic.Create (\_T ("Click me"), WS\_CHILD ¦ WS\_VISIBLE ¦  SS\_CENTER ¦ SS\_CENTERIMAGE ¦ SS\_NOTIFY ¦ SS\_SUNKEN, rect,  this, IDC\_STATIC);    void CMainWindow::OnClicked ()  {  m\_wndStatic.PostMessage (WM\_CLOSE, 0, 0);  } |

create a static control that displays "Click me" in the center of a sunken rectangle and disappears from the screen when clicked. If a static control lacks the SS\_NOTIFY style, mouse messages go through to the underlying window because the control's window procedure returns HTTRANSPARENT in response to WM\_NCHITTEST messages.

## The FontView Application

Let's put what we've learned so far about buttons, list boxes, and static controls to use in an application. The FontView program shown in Figure 7-3 lists the names of all the fonts installed on the host PC in a list box. When a font name is selected, a sample is drawn in the group box at the bottom of the window. The sample text is really a static control, so all FontView has to do to display a font sample is call the control's *SetFont* function. If the check box labeled Show TrueType Fonts Only is checked, non-TrueType fonts are excluded from the list. In addition to showing how push button, check box, list box, group box, and static controls are used, FontView also demonstrates a very important MFC programming technique—the use of C++ member functions as callback functions. The term *callback function* might not mean much to you at the moment, but you'll learn all about it shortly.



**Figure 7-3.** *The FontView window.*

FontView's source code appears in Figure 7-4. The controls are created one by one in *CMainWindow::OnCreate*. All but one—the static control that displays the font sample—is assigned an 8-point MS Sans Serif font. Rather than use raw pixel counts to size and position the controls, FontView uses distances based on the width and height of 8-point MS Sans Serif characters to achieve independence from the physical resolution of the display device. The character height and width are measured by selecting the font into a device context and calling *CDC::GetTextMetrics* with a pointer to a TEXTMETRIC structure:

|  |
| --- |
| CFont\* pOldFont = dc.SelectObject (&m\_fontMain);  TEXTMETRIC tm;  dc.GetTextMetrics (&tm);  m\_cxChar = tm.tmAveCharWidth;  m\_cyChar = tm.tmHeight + tm.tmExternalLeading; |

On return, the structure's *tmAveCharWidth* field holds the average character width. (Actual character width can vary from character to character in a proportionally spaced font.) Summing the *tmHeight* and *tmExternalLeading* fields yields the height of one line of text, including interline spacing.

**Figure 7-4.** *The FontView application.*

|  |
| --- |
| FontView.h class CMyApp : public CWinApp  {  public:  virtual BOOL InitInstance ();  };  class CMainWindow : public CWnd  {  protected:  int m\_cxChar;  int m\_cyChar;  CFont m\_fontMain;  CFont m\_fontSample;  CStatic m\_wndLBTitle;  CListBox m\_wndListBox;  CButton m\_wndCheckBox;  CButton m\_wndGroupBox;  CStatic m\_wndSampleText;  CButton m\_wndPushButton;  void FillListBox ();  public:  CMainWindow ();  static int CALLBACK EnumFontFamProc (ENUMLOGFONT\* lpelf,  NEWTEXTMETRIC\* lpntm, int nFontType, LPARAM lParam);  protected:  virtual void PostNcDestroy ();  afx\_msg int OnCreate (LPCREATESTRUCT lpcs);  afx\_msg void OnPushButtonClicked ();  afx\_msg void OnCheckBoxClicked ();  afx\_msg void OnSelChange ();  DECLARE\_MESSAGE\_MAP ()  }; |

|  |
| --- |
| FontView.cpp #include <afxwin.h>  #include "FontView.h"  #define IDC\_PRINT 100  #define IDC\_CHECKBOX 101  #define IDC\_LISTBOX 102  #define IDC\_SAMPLE 103  CMyApp myApp;  /////////////////////////////////////////////////////////////////////////  // CMyApp member functions  BOOL CMyApp::InitInstance ()  {  m\_pMainWnd = new CMainWindow;  m\_pMainWnd->ShowWindow (m\_nCmdShow);  m\_pMainWnd->UpdateWindow ();  return TRUE;  }  /////////////////////////////////////////////////////////////////////////  // CMainWindow message map and member functions  BEGIN\_MESSAGE\_MAP (CMainWindow, CWnd)  ON\_WM\_CREATE ()  ON\_BN\_CLICKED (IDC\_PRINT, OnPushButtonClicked)  ON\_BN\_CLICKED (IDC\_CHECKBOX, OnCheckBoxClicked)  ON\_LBN\_SELCHANGE (IDC\_LISTBOX, OnSelChange)  END\_MESSAGE\_MAP ()  CMainWindow::CMainWindow ()  {  CString strWndClass = AfxRegisterWndClass (  0,  myApp.LoadStandardCursor (IDC\_ARROW),  (HBRUSH) (COLOR\_3DFACE + 1),  myApp.LoadStandardIcon (IDI\_WINLOGO)  );  CreateEx (0, strWndClass, \_T ("FontView"),  WS\_OVERLAPPED ¦ WS\_SYSMENU ¦ WS\_CAPTION ¦ WS\_MINIMIZEBOX,  CW\_USEDEFAULT, CW\_USEDEFAULT, CW\_USEDEFAULT, CW\_USEDEFAULT,  NULL, NULL, NULL);  CRect rect (0, 0, m\_cxChar \* 68, m\_cyChar \* 26);  CalcWindowRect (&rect);  SetWindowPos (NULL, 0, 0, rect.Width (), rect.Height (),  SWP\_NOZORDER ¦ SWP\_NOMOVE ¦ SWP\_NOREDRAW);  }  int CMainWindow::OnCreate (LPCREATESTRUCT lpcs)  {  if (CWnd::OnCreate (lpcs) == -1)  return -1;  //  // Create an 8-point MS Sans Serif font to use in the controls.  //  m\_fontMain.CreatePointFont (80, \_T ("MS Sans Serif"));  //  // Compute the average width and height of a character in the font.  //  CClientDC dc (this);  CFont\* pOldFont = dc.SelectObject (&m\_fontMain);  TEXTMETRIC tm;  dc.GetTextMetrics (&tm);  m\_cxChar = tm.tmAveCharWidth;  m\_cyChar = tm.tmHeight + tm.tmExternalLeading;  dc.SelectObject (pOldFont);  //  // Create the controls that will appear in the FontView window.  //  CRect rect (m\_cxChar \* 2, m\_cyChar, m\_cxChar \* 48, m\_cyChar \* 2);  m\_wndLBTitle.Create (\_T ("Typefaces"), WS\_CHILD ¦ WS\_VISIBLE ¦ SS\_LEFT,  rect, this);  rect.SetRect (m\_cxChar \* 2, m\_cyChar \* 2, m\_cxChar \* 48,  m\_cyChar \* 18);  m\_wndListBox.CreateEx (WS\_EX\_CLIENTEDGE, \_T ("listbox"), NULL,  WS\_CHILD ¦ WS\_VISIBLE ¦ LBS\_STANDARD, rect, this, IDC\_LISTBOX);  rect.SetRect (m\_cxChar \* 2, m\_cyChar \* 19, m\_cxChar \* 48,  m\_cyChar \* 20);  m\_wndCheckBox.Create (\_T ("Show TrueType fonts only"), WS\_CHILD ¦  WS\_VISIBLE ¦ BS\_AUTOCHECKBOX, rect, this, IDC\_CHECKBOX);  rect.SetRect (m\_cxChar \* 2, m\_cyChar \* 21, m\_cxChar \* 66,  m\_cyChar \* 25);  m\_wndGroupBox.Create (\_T ("Sample"), WS\_CHILD ¦ WS\_VISIBLE ¦ BS\_GROUPBOX,  rect, this, (UINT) -1);  rect.SetRect (m\_cxChar \* 4, m\_cyChar \* 22, m\_cxChar \* 64,  (m\_cyChar \* 99) / 4);  m\_wndSampleText.Create (\_T (""), WS\_CHILD ¦ WS\_VISIBLE ¦ SS\_CENTER, rect,  this, IDC\_SAMPLE);  rect.SetRect (m\_cxChar \* 50, m\_cyChar \* 2, m\_cxChar \* 66,  m\_cyChar \* 4);  m\_wndPushButton.Create (\_T ("Print Sample"), WS\_CHILD ¦ WS\_VISIBLE ¦  WS\_DISABLED ¦ BS\_PUSHBUTTON, rect, this, IDC\_PRINT);  //  // Set each control's font to 8-point MS Sans Serif.  //  m\_wndLBTitle.SetFont (&m\_fontMain, FALSE);  m\_wndListBox.SetFont (&m\_fontMain, FALSE);  m\_wndCheckBox.SetFont (&m\_fontMain, FALSE);  m\_wndGroupBox.SetFont (&m\_fontMain, FALSE);  m\_wndPushButton.SetFont (&m\_fontMain, FALSE);  //  // Fill the list box with typeface names and return.  //  FillListBox ();  return 0;  }  void CMainWindow::PostNcDestroy ()  {  delete this;  }  void CMainWindow::OnPushButtonClicked ()  {  MessageBox (\_T ("This feature is currently unimplemented. Sorry!"),  \_T ("Error"), MB\_ICONINFORMATION ¦ MB\_OK);  }  void CMainWindow::OnCheckBoxClicked ()  {  FillListBox ();  OnSelChange ();  }  void CMainWindow::OnSelChange ()  {  int nIndex = m\_wndListBox.GetCurSel ();  if (nIndex == LB\_ERR) {  m\_wndPushButton.EnableWindow (FALSE);  m\_wndSampleText.SetWindowText (\_T (""));  }  else {  m\_wndPushButton.EnableWindow (TRUE);  if ((HFONT) m\_fontSample != NULL)  m\_fontSample.DeleteObject ();  CString strFaceName;  m\_wndListBox.GetText (nIndex, strFaceName);  m\_fontSample.CreateFont (-m\_cyChar \* 2, 0, 0, 0, FW\_NORMAL,  0, 0, 0, DEFAULT\_CHARSET, OUT\_CHARACTER\_PRECIS,  CLIP\_CHARACTER\_PRECIS, DEFAULT\_QUALITY, DEFAULT\_PITCH ¦  FF\_DONTCARE, strFaceName);  m\_wndSampleText.SetFont (&m\_fontSample);  m\_wndSampleText.SetWindowText (\_T ("AaBbCcDdEeFfGg"));  }  }  void CMainWindow::FillListBox ()  {  m\_wndListBox.ResetContent ();  CClientDC dc (this);  ::EnumFontFamilies ((HDC) dc, NULL, (FONTENUMPROC) EnumFontFamProc,  (LPARAM) this);  }  int CALLBACK CMainWindow::EnumFontFamProc (ENUMLOGFONT\* lpelf,  NEWTEXTMETRIC\* lpntm, int nFontType, LPARAM lParam)  {  CMainWindow\* pWnd = (CMainWindow\*) lParam;  if ((pWnd->m\_wndCheckBox.GetCheck () == BST\_UNCHECKED) ||  (nFontType & TRUETYPE\_FONTTYPE))  pWnd->m\_wndListBox.AddString (lpelf->elfLogFont.lfFaceName);  return 1;  } |

*CMainWindow* processes three types of control notifications: BN\_CLICKED notifications from the push button, BN\_CLICKED notifications from the check box, and LBN\_SELCHANGE notifications from the list box. The corresponding message-map entries look like this:

|  |
| --- |
| ON\_BN\_CLICKED (IDC\_PRINT, OnPushButtonClicked)  ON\_BN\_CLICKED (IDC\_CHECKBOX, OnCheckBoxClicked)  ON\_LBN\_SELCHANGE (IDC\_LISTBOX, OnSelChange) |

*OnPushButtonClicked* is activated when the Print Sample button is clicked. Because printing is a complex undertaking in a Windows application, *OnPushButtonClicked* does nothing more than display a message box. *OnCheckBoxClicked* handles BN\_CLICKED notifications from the check box. Since the check box style includes a BS\_AUTOCHECKBOX flag, the check mark toggles on and off automatically in response to button clicks. *OnCheckBoxClicked*'s job is to refresh the contents of the list box each time the check mark is toggled. To do that, it calls *CMainWindow::FillListBox* to reinitialize the list box and then calls *CMainWindow::OnSelChange* to update the sample text.

*OnSelChange* is also called whenever the list box selection changes. It calls *GetCurSel* to get the index of the currently selected item. If *GetCurSel* returns LB\_ERR, indicating that nothing is selected, *OnSelChange* disables the push button and erases the sample text. Otherwise, it enables the button, retrieves the text of the selected item with *CListBox::GetText*, and creates a font whose typeface name equals the string returned by *GetText*. It then assigns the font to the static control and sets the control text to "AaBbCcDdEeFfGg."

### Font Enumerations and Callback Functions

The job of filling the list box with font names falls to *CMainWindow::FillListBox*. *FillListBox* is called by *OnCreate* to initialize the list box when the program is started. It is also called by *OnCheckBoxClicked* to reinitialize the list box when the Show TrueType Fonts Only check box is clicked. *FillListBox* first clears the list box by calling *CListBox::ResetContent*. It then enumerates all the fonts installed in the system and adds the corresponding typeface names to the list box.

*FillListBox* begins the enumeration process by constructing a device context object named *dc*, using the *CDC* class's HDC operator to extract a device context handle, and passing that handle to the *::EnumFontFamilies* function:

|  |
| --- |
| CClientDC dc (this);  ::EnumFontFamilies ((HDC) dc, NULL, (FONTENUMPROC) EnumFontFamProc,  (LPARAM) this); |

The NULL second parameter tells *::EnumFontFamilies* to enumerate all installed fonts. The next parameter is the address of a callback function*.* A *callback function* is a function in your application that Windows *calls back* with information you requested. For each font that *::EnumFontFamilies* enumerates, Windows calls your callback function one time. An *::EnumFontFamilies* callback function must be prototyped like this:

|  |
| --- |
| int CALLBACK EnumFontFamProc (ENUMLOGFONT\* lpelf,  NEWTEXTMETRIC\* lpntm, int nFontType, LPARAM lParam) |

*lpelf* is a pointer to an ENUMLOGFONT structure, which contains a wealth of information about the font, including its typeface name. *lpntm* is a pointer to a structure of type NEWTEXTMETRIC, which contains font metrics—height, average character width, and so on. *nFontType* specifies the font type. TrueType fonts are identified by logically ANDing *nFontType* with the value TRUETYPE\_FONTTYPE. If the result is nonzero, the font is a TrueType font. The fourth and final parameter, *lParam*, is an optional 32-bit LPARAM value passed to *::EnumFontFamilies*. *FillListBox* passes the *this* pointer referring to *CMainWindow*, for reasons I'll explain in a moment.

FontView's callback function is a member of *CMainWindow*. It's actually the callback function, not *FillListBox*, that adds the typeface names to the list box. Each time *CMainWindow::EnumFontFamProc* is called, it casts the *lParam* value passed to it from *FillListBox* into a *CMainWindow* pointer:

|  |
| --- |
| CMainWindow\* pWnd = (CMainWindow\*) lParam; |

It then uses the pointer to add the typeface name to the list box, but only if the Show TrueType Fonts Only check box is unchecked or the font is a TrueType font:

|  |
| --- |
| if ((pWnd->m\_wndCheckBox.GetCheck () == BST\_UNCHECKED) ¦¦  (nFontType & TRUETYPE\_FONTTYPE))  pWnd->m\_wndListBox.AddString (lpelf->elfLogFont.lfFaceName);  return 1; |

The nonzero return value tells Windows to continue the enumeration process. (The callback function can halt the process at any time by returning 0, a handy option to have if you've allocated a fixed amount of memory to store font information and the memory fills up.) After Windows has called *EnumFontFamProc* for the last time, the call that *FillListBox* placed to *::EnumFontFamilies* returns and the enumeration process is complete.

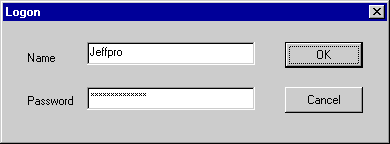
Why does *FillListBox* pass a *this* pointer to the callback function, and why does *EnumFontFamProc* cast the pointer to a *CMainWindow* pointer when it, too, is a member of *CMainWindow*? Look closely at the declaration for *CMainWindow* in FontView.h, and you'll see that *EnumFontFamProc* is a static member function.A static class member function doesn't receive a *this* pointer, so it can't access nonstatic members of its own class. To call *m\_wndCheckBox*'s *GetCheck* function and *m\_wndListBox*'s *AddString*, *EnumFontFamProc* needs pointers to *m\_wndCheckBox* and *m\_wndListBox* or a pointer to the *CMainWindow* object to which those objects belong. By casting the *lParam* value passed to *FillListBox* to a *CMainWindow* pointer, *EnumFontFamProc* is able to access nonstatic members of the *CMainWindow* class just as if it were a nonstatic member function.

*EnumFontFamProc* is static because callbacks require special handling in C++ applications. Windows rigidly defines a callback function's interface—the parameters passed to it through its argument list. When a member function of a C++ class is declared, the compiler silently tacks on an extra argument to hold the *this* pointer. Unfortunately, the added parameter means that the callback function's argument list doesn't match the argument list Windows expects, and all sorts of bad things can happen as a result, including invalid memory access errors, the nemeses of all Windows programmers. There are several solutions to this problem, but declaring the callback to be a static member function is among the simplest and most direct. In C++, a static member function isn't passed a *this* pointer, so its argument list is unaltered.

Callback functions are common in Windows, so the technique demonstrated here is useful for more than just enumerating fonts. Many Windows API functions that rely on callbacks support an application-defined *lParam* value, which is perfect for passing *this* pointers to statically declared callback functions. Should you use an enumeration function that doesn't support an application-defined *lParam*, you'll have to resort to other means to make a pointer available. One alternative is to make the *this* pointer visible to the callback function by copying it to a global variable.

## The *CEdit* Class

MFC's *CEdit* class encapsulates the functionality of edit controls. Edit controls are used for text entry and editing and come in two varieties: single-line and multiline. Single-line edit controls are perfect for soliciting one-line text strings such as names, passwords, and product IDs. (See Figure 7-5.) To see a multiline edit control in action, start the Notepad applet that comes with Windows. The client area of the Notepad window is a multiline edit control.



**Figure 7-5.** *A dialog box with two single-line edit controls.*

An edit control is limited to about 60 KB of text. That's not much of a restriction for single-line edit controls, but for a multiline edit control it can be constraining. If you need to handle large amounts of text, use the *rich edit control* instead—an enhanced version of the standard edit control that is part of the common controls library. Though designed to handle richly formatted text of the type seen in word processors, rich edit controls are quite capable of handling ordinary text, too. The Windows WordPad applet uses a rich edit control for text entry and editing. You'll use a rich edit control to build a WordPad-like application of your own in [Chapter 12](mk:@MSITStore:C:\Program%20Files%20(x86)\MSPress\BooksOnline\Programming%20Windows%20with%20MFC%20Second%20Edition\progmfc2.chm::/ch12a.htm).

### Creating an Edit Control

If *m\_wndEdit* is a *CEdit* object, the statement

|  |
| --- |
| m\_wndEdit.Create (WS\_CHILD ¦ WS\_VISIBLE ¦ WS\_BORDER ¦  ES\_AUTOHSCROLL, rect, this, IDC\_EDIT); |

creates a single-line edit control that automatically scrolls horizontally when the caret moves beyond the control's border. Including ES\_MULTILINE in the window style creates a multiline edit control instead:

|  |
| --- |
| m\_wndEdit.Create (WS\_CHILD ¦ WS\_VISIBLE ¦ WS\_BORDER ¦  WS\_HSCROLL ¦ WS\_VSCROLL ¦ ES\_MULTILINE, rect, this, IDC\_EDIT); |

WS\_HSCROLL and WS\_VSCROLL add horizontal and vertical scroll bars to the control. You can use *CEdit::SetRect* or *CEdit::SetRectNP* to define the control's editable area independent of the control's borders. One use for these functions is to define a page size that remains constant even if the control is resized. You can also use *CEdit::SetMargins* to specify left and right margin widths in pixels. The default margin widths are 0. The window styles listed in the table below are specific to edit controls.

When it is first created, an edit control will accept only about 30,000 characters. You can raise or lower the limit with *CEdit::LimitText* or the Win32-specific *CEdit::SetLimitText*. The following statement sets the maximum number of characters that an edit control will accept to 32:

|  |
| --- |
| m\_wndEdit.SetLimitText (32); |

When used with a multiline edit control, *SetLimitText* limits the total amount of text entered into the control, not the length of each line. There is no built-in way to limit the number of characters per line in a multiline edit control, but there are ways you can do it manually. One approach is to use *SetFont* to switch the edit control font to a fixed-pitch font and *CEdit::SetRect* to specify a formatting rectangle whose width is slightly greater than the width of a character times the desired number of characters per line.

**Edit Control Styles**

|  |  |
| --- | --- |
| ***Style*** | ***Description*** |
| ES\_LEFT | Left-aligns text in the control. |
| ES\_CENTER | Centers text in the control. |
| ES\_RIGHT | Right-aligns text in the control. |
| ES\_AUTOHSCROLL | Permits the edit control to scroll horizontally without a horizontal scroll bar. To add a horizontal scroll bar, include the style WS\_HSCROLL. |
| ES\_AUTOVSCROLL | Permits the edit control to scroll vertically without a vertical scroll bar. To add a vertical scroll bar, include the style WS\_VSCROLL. |
| ES\_MULTILINE | Creates a multiline edit control. |
| ES\_LOWERCASE | Displays all characters in lowercase. |
| ES\_UPPERCASE | Displays all characters in uppercase. |
| ES\_PASSWORD | Displays asterisks instead of typed characters. |
| ES\_READONLY | Creates an edit control whose text can't be edited. |
| ES\_NOHIDESEL | Prevents the edit control from hiding the selection when the control loses the input focus. |
| ES\_OEMCONVERT | Performs an ANSI-to-OEM-to-ANSI conversion on all characters typed into the control so that the application won't get unexpected results if it performs an ANSI-to-OEM conversion of its own. Obsolete. |
| ES\_WANTRETURN | Programs the Enter key to insert line breaks instead of invoking the default push button for multiline edit controls used in dialog boxes. |

Another function sometimes used to initialize an edit control is *CEdit::SetTabStops*, which sets the spacing between tab stops. Default tab stops are set about 8 character widths apart. You can space the tab stops however you like and can even vary the spacing between stops. Like *CListBox::SetTabStops*, *CEdit::SetTabStops* measures distances in dialog units.

### Inserting and Retrieving Text

Text is inserted into an edit control with *SetWindowText* and retrieved with *GetWindowText*. *CEdit* inherits both functions from its base class, *CWnd*. The statement

|  |
| --- |
| m\_wndEdit.SetWindowText (\_T ("Hello, MFC")); |

inserts the text string "Hello, MFC" into the edit control *m\_wndEdit*, and

|  |
| --- |
| m\_wndEdit.GetWindowText (string); |

retrieves the text into a *CString* object named *string*. *GetWindowText* and *SetWindowText* work with both single-line and multiline edit controls. Text inserted with *SetWindowText* replaces existing text, and *GetWindowText* returns all the text in the edit control, even if the text spans multiple lines. To erase all the text in an edit control, call *SetWindowText* with a null string:

|  |
| --- |
| m\_wndEdit.SetWindowText (\_T ("")); |

You can insert text into an edit control without erasing what's already there with *CEdit::ReplaceSel*. If one or more characters are selected when *ReplaceSel* is called, the inserted text replaces the selected text; otherwise, the new text is inserted at the current caret position.

A multiline edit control inserts line breaks automatically. If you'd like to know where the line breaks fall in text retrieved from a multiline edit control, use *CEdit::FmtLines* to enable soft line breaks before calling *GetWindowText*:

|  |
| --- |
| m\_wndEdit.FmtLines (TRUE); |

With soft line breaks enabled, each line is delimited with two carriage returns (13) followed by a line feed character (10). To disable soft line breaks, call *FmtLines* with a FALSE parameter:

|  |
| --- |
| m\_wndEdit.FmtLines (FALSE); |

Now line breaks won't be denoted in any special way. Hard returns—line breaks inserted manually when the user presses the Enter key—are signified by single carriage return/line feed pairs regardless of the *FmtLines* setting. *FmtLines* doesn't affect the appearance of the text in a multiline edit control. It affects only the way in which the control stores text internally and the format of text retrieved with *GetWindowText*.

To read just one line of text from a multiline edit control, use *CEdit::GetLine*. *GetLine* copies the contents of a line to a buffer whose address you provide. The line is identified with a 0-based index. The statement

|  |
| --- |
| m\_wndEdit.GetLine (0, pBuffer, nBufferSize); |

copies the first line of text in a multiline edit control to the buffer pointed to by *pBuffer*. The third parameter is the buffer size, in bytes (not characters). *GetLine* returns the number of bytes copied to the buffer. You can determine how much buffer space you need before retrieving a line with *CEdit::LineLength*. And you can find out how many lines of text a multiline edit control contains by calling *CEdit::GetLineCount*. Note that *GetLineCount* never returns 0; the return value is 1 even if no text has been entered.

### Clear, Cut, Copy, Paste, and Undo

*CEdit* provides easy-to-use member functions that perform the programmatic equivalents of the Clear, Cut, Copy, Paste, and Undo items in the Edit menu. The statement

|  |
| --- |
| m\_wndEdit.Clear (); |

removes the selected text without affecting what's on the clipboard. The statement

|  |
| --- |
| m\_wndEdit.Cut (); |

removes the selected text and copies it to the clipboard. And the statement

|  |
| --- |
| m\_wndEdit.Copy (); |

copies the selected text to the clipboard without altering the contents of the edit control.

You can query an edit control for the current selection by calling *CEdit::GetSel*, which returns a DWORD value with two packed 16-bit integers specifying the indexes of the beginning and ending characters in the selection. An alternate form of *GetSel* copies the indexes to a pair of integers whose addresses are passed by reference. If the indexes are equal, no text is currently selected. The following *IsTextSelected* function, which you might add to an edit control class derived from *CEdit*, returns a nonzero value if a selection exists and 0 if one doesn't exist:

|  |
| --- |
| BOOL CMyEdit::IsTextSelected ()  {  int nStart, nEnd;  GetSel (nStart, nEnd);  return (nStart != nEnd);  } |

*CEdit::Cut* and *CEdit::Copy* do nothing if no text is selected.

Text can be selected programmatically with *CEdit::SetSel*. The statement

|  |
| --- |
| m\_wndEdit.SetSel (100, 150); |

selects 50 characters beginning with the 101st (the character whose 0-based index is 100) and scrolls the selection into view if it isn't visible already. To prevent scrolling, include a third parameter and set it equal to TRUE.

When programmatically selecting text in a multiline edit control, you often need to convert a line number and possibly an offset within that line into an index that can be passed to *SetSel*. *CEdit::LineIndex* accepts a 0-based line number and returns the index of the first character in that line. The next example uses *LineIndex* to determine the index of the first character in the eighth line of a multiline edit control, *LineLength* to retrieve the line's length, and *SetSel* to select everything on that line:

|  |
| --- |
| int nStart = m\_wndEdit.LineIndex (7);  int nLength = m\_wndEdit.LineLength (nStart);  m\_wndEdit.SetSel (nStart, nStart + nLength); |

*CEdit* also provides a function named *LineFromChar* for computing a line number from a character index.

*CEdit::Paste* pastes text into an edit control. The following statement pastes the text that currently resides in the Windows clipboard into an edit control named *m\_wndEdit*:

|  |
| --- |
| m\_wndEdit.Paste (); |

If the clipboard contains no text, *CEdit::Paste* does nothing. If no text is selected when *Paste* is called, the clipboard text is inserted at the current caret position. If a selection exists, the text retrieved from the clipboard replaces the text selected in the control. You can determine ahead of time whether the clipboard contains text (and therefore whether the *Paste* function will actually do anything) by calling *::IsClipboardFormatAvailable*. The statement

|  |
| --- |
| BOOL bCanPaste = ::IsClipboardFormatAvailable (CF\_TEXT); |

sets *bCanPaste* to nonzero if text is available from the clipboard, and 0 if it isn't.

Edit controls also feature a built-in undo capability that "rolls back" the previous editing operation. The statement

|  |
| --- |
| m\_wndEdit.Undo (); |

undoes the last operation, provided that the operation can be undone. You can determine ahead of time whether calling *Undo* will accomplish anything with *CEdit::CanUndo*. A related function, *CEdit::EmptyUndoBuffer*, manually resets the undo flag so that subsequent calls to *Undo* will do nothing (and calls to *CanUndo* will return FALSE) until another editing operation is performed.

### Edit Control Notifications

Edit controls send notifications to their parents to report various input events. In MFC applications, these notifications are mapped to handling functions with ON\_EN message map macros. Edit control notifications and the corresponding message map macros are summarized in the table below.

A common use for EN\_CHANGE notifications is to dynamically update other controls as text is entered into an edit control. The following code updates a push button (*m\_wndPushButton*) as text is entered into an edit control (*m\_wndEdit*, ID=IDC\_EDIT) so that the push button is enabled if the edit control contains at least one character and disabled if it doesn't:

|  |
| --- |
| // In CMainWindow's message map  ON\_EN\_CHANGE (IDC\_EDIT, OnUpdatePushButton)    void CMainWindow::OnUpdatePushButton ()  {  m\_wndPushButton.EnableWindow (m\_wndEdit.LineLength ());  } |

**Edit Control Notifications**

|  |  |  |
| --- | --- | --- |
| ***Notification*** | ***Sent When*** | ***Message-Map Macro*** |
| EN\_UPDATE | The control's text is about to change. | ON\_EN\_UPDATE |
| EN\_CHANGE | The control's text has changed. | ON\_EN\_CHANGE |
| EN\_KILLFOCUS | The edit control loses the input focus. | ON\_EN\_KILLFOCUS |
| EN\_SETFOCUS | The edit control receives the input focus. | ON\_EN\_SETFOCUS |
| EN\_HSCROLL | The edit control is scrolled horizontally using a scroll bar. | ON\_EN\_HSCROLL |
| EN\_VSCROLL | The edit control is scrolled vertically using a scroll bar. | ON\_EN\_VSCROLL |
| EN\_MAXTEXT | A character can't be entered because the edit control already contains the number of characters specified with *CEdit::LimitText* or *CEdit::SetLimitText*. This notification is also sent if a character can't be entered because the caret is at the right or the bottom edge of the control's formatting rectangle and the control doesn't support scrolling. | ON\_EN\_MAXTEXT |
| EN\_ERRSPACE | An operation fails because of insufficient memory. | ON\_EN\_ERRSPACE |

Providing interactive feedback of this nature is generally considered good user interface design. Most users would rather see a button remain disabled until all of the required information is entered than click a button and receive an error message.

## Presto! Instant Notepad

The MyPad application, portions of whose source code are reproduced in Figure 7-6, uses a multiline edit control to create a near clone of the Windows Notepad applet. As you can see from the source code, the edit control does the bulk of the work. *CEdit* functions such as *Undo* and *Cut* allow you to implement commands in the Edit menu with just one line of code.

MyPad is a view-based application that I began by running the MFC AppWizard but unchecking the Document/View Architecture Support box in Step 1. To avoid unnecessary code, I unchecked the ActiveX Controls box in AppWizard's Step 3 dialog, too. After running AppWizard, I added a New command to the File menu and a Delete command to the Edit menu using the Visual C++ resource editor. I also used the resource editor to add an accelerator (Ctrl-N) for the New command. I then used ClassWizard to add command handlers, update handlers, and message handlers.

The view's WM\_CREATE message handler creates the edit control by calling *Create* on the *CEdit* data member named *m\_wndEdit*. *OnCreate* sets the control's width and height to 0, but *OnSize* resizes the control to fill the view's client area whenever the view receives a WM\_SIZE message. The first WM\_SIZE message arrives before the view becomes visible on the screen; subsequent WM\_SIZE messages arrive anytime the MyPad window (and consequently, the view) is resized. A one-line WM\_SETFOCUS handler in the view class shifts the input focus to the edit control whenever the view receives the input focus.

**Figure 7-6.** *The MyPad application.*

|  |
| --- |
| MainFrm.h // MainFrm.h : interface of the CMainFrame class  //  ///////////////////////////////////////////////////////////////////////////  #if !defined(AFX\_MAINFRM\_H\_\_0FA1D288\_8471\_11D2\_8E53\_006008A82731\_\_INCLUDED\_)  #define AFX\_MAINFRM\_H\_\_0FA1D288\_8471\_11D2\_8E53\_006008A82731\_\_INCLUDED\_  #if \_MSC\_VER > 1000  #pragma once  #endif // \_MSC\_VER > 1000  #include "ChildView.h"  class CMainFrame : public CFrameWnd  {    public:  CMainFrame();  protected:  DECLARE\_DYNAMIC(CMainFrame)  // Attributes  public:  // Operations  public:  // Overrides  // ClassWizard generated virtual function overrides  //{{AFX\_VIRTUAL(CMainFrame)  virtual BOOL PreCreateWindow(CREATESTRUCT& cs);  virtual BOOL OnCmdMsg(UINT nID, int nCode, void\* pExtra,  AFX\_CMDHANDLERINFO\* pHandlerInfo);  //}}AFX\_VIRTUAL  // Implementation  public:  virtual ~CMainFrame();  #ifdef \_DEBUG  virtual void AssertValid() const;  virtual void Dump(CDumpContext& dc) const;  #endif  CChildView m\_wndView;  // Generated message map functions  protected:  //{{AFX\_MSG(CMainFrame)  afx\_msg void OnSetFocus(CWnd \*pOldWnd);  afx\_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);  //}}AFX\_MSG  DECLARE\_MESSAGE\_MAP()  };  ///////////////////////////////////////////////////////////////////////////  //{{AFX\_INSERT\_LOCATION}}  // Microsoft Visual C++ will insert additional declarations  // immediately before the previous line.  #endif  // !defined(AFX\_MAINFRM\_H\_\_0FA1D288\_8471\_11D2\_8E53\_006008A82731\_\_INCLUDED\_) |

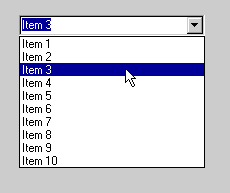
|  |
| --- |
| MainFrm.cpp // MainFrm.cpp : implementation of the CMainFrame class  //  #include "stdafx.h"  #include "MyPad.h"  #include "MainFrm.h"  #ifdef \_DEBUG  #define new DEBUG\_NEW  #undef THIS\_FILE  static char THIS\_FILE[] = \_\_FILE\_\_;  #endif  ///////////////////////////////////////////////////////////////////////////  // CMainFrame  IMPLEMENT\_DYNAMIC(CMainFrame, CFrameWnd)  BEGIN\_MESSAGE\_MAP(CMainFrame, CFrameWnd)  //{{AFX\_MSG\_MAP(CMainFrame)  ON\_WM\_SETFOCUS()  ON\_WM\_CREATE()  //}}AFX\_MSG\_MAP  END\_MESSAGE\_MAP()  ///////////////////////////////////////////////////////////////////////////  // CMainFrame construction/destruction  CMainFrame::CMainFrame()  {  }  CMainFrame::~CMainFrame()  {  }  BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)  {  if( !CFrameWnd::PreCreateWindow(cs) )  return FALSE;  cs.dwExStyle &= ~WS\_EX\_CLIENTEDGE;  cs.lpszClass = AfxRegisterWndClass(0);  return TRUE;  }  ///////////////////////////////////////////////////////////////////////////  // CMainFrame diagnostics  #ifdef \_DEBUG  void CMainFrame::AssertValid() const  {  CFrameWnd::AssertValid();  }  void CMainFrame::Dump(CDumpContext& dc) const  {  CFrameWnd::Dump(dc);  }  #endif //\_DEBUG  ///////////////////////////////////////////////////////////////////////////  // CMainFrame message handlers  void CMainFrame::OnSetFocus(CWnd\* pOldWnd)  {  // forward focus to the view window  m\_wndView.SetFocus();  }  BOOL CMainFrame::OnCmdMsg(UINT nID, int nCode, void\* pExtra,  AFX\_CMDHANDLERINFO\* pHandlerInfo)  {  // let the view have first crack at the command  if (m\_wndView.OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))  return TRUE;  // otherwise, do default handling  return CFrameWnd::OnCmdMsg(nID, nCode, pExtra, pHandlerInfo);  }  int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)  {  if (CFrameWnd::OnCreate(lpCreateStruct) == -1)  return -1;    if (!m\_wndView.Create(NULL, NULL, AFX\_WS\_DEFAULT\_VIEW,  CRect(0, 0, 0, 0), this, AFX\_IDW\_PANE\_FIRST, NULL))  {  TRACE0("Failed to create view window\n");  return -1;  }  return 0;  } |

|  |
| --- |
| ChildView.h // ChildView.h : interface of the CChildView class  //  ///////////////////////////////////////////////////////////////////////////  #if !defined(AFX\_CHILDVIEW\_H\_\_0FA1D28A\_8471\_11D2\_8E53\_006008A82731\_\_INCLUDED\_)  #define AFX\_CHILDVIEW\_H\_\_0FA1D28A\_8471\_11D2\_8E53\_006008A82731\_\_INCLUDED\_  #if \_MSC\_VER > 1000  #pragma once  #endif // \_MSC\_VER > 1000  ///////////////////////////////////////////////////////////////////////////  // CChildView window  class CChildView : public CWnd  {  // Construction  public:  CChildView();  // Attributes  public:  // Operations  public:  // Overrides  // ClassWizard generated virtual function overrides  //{{AFX\_VIRTUAL(CChildView)  protected:  virtual BOOL PreCreateWindow(CREATESTRUCT& cs);  //}}AFX\_VIRTUAL  // Implementation  public:  virtual ~CChildView();  // Generated message map functions  protected:  BOOL IsTextSelected ();  CEdit m\_wndEdit;  //{{AFX\_MSG(CChildView)  afx\_msg void OnPaint();  afx\_msg void OnEditCut();  afx\_msg void OnEditCopy();  afx\_msg void OnEditPaste();  afx\_msg void OnEditDelete();  afx\_msg void OnEditUndo();  afx\_msg void OnUpdateEditCut(CCmdUI\* pCmdUI);  afx\_msg void OnUpdateEditCopy(CCmdUI\* pCmdUI);  afx\_msg void OnUpdateEditPaste(CCmdUI\* pCmdUI);  afx\_msg void OnUpdateEditDelete(CCmdUI\* pCmdUI);  afx\_msg void OnUpdateEditUndo(CCmdUI\* pCmdUI);  afx\_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);  afx\_msg void OnSize(UINT nType, int cx, int cy);  afx\_msg void OnFileNew();  afx\_msg void OnSetFocus(CWnd\* pOldWnd);  //}}AFX\_MSG  DECLARE\_MESSAGE\_MAP()  };  ///////////////////////////////////////////////////////////////////////////  //{{AFX\_INSERT\_LOCATION}}  // Microsoft Visual C++ will insert additional declarations  // immediately before the previous line.  #endif  // !defined(AFX\_CHILDVIEW\_H\_\_0FA1D28A\_8471\_11D2\_8E53\_006008A82731\_\_INCLUDED\_) |

|  |
| --- |
| ChildView.cpp // ChildView.cpp : implementation of the CChildView class  //  #include "stdafx.h"  #include "MyPad.h"  #include "ChildView.h"  #ifdef \_DEBUG  #define new DEBUG\_NEW  #undef THIS\_FILE  static char THIS\_FILE[] = \_\_FILE\_\_;  #endif  ///////////////////////////////////////////////////////////////////////////  // CChildView  CChildView::CChildView()  {  }  CChildView::~CChildView()  {  }  BEGIN\_MESSAGE\_MAP(CChildView,CWnd )  //{{AFX\_MSG\_MAP(CChildView)  ON\_WM\_PAINT()  ON\_WM\_CREATE()  ON\_WM\_SIZE()  ON\_WM\_SETFOCUS()  ON\_COMMAND(ID\_EDIT\_CUT, OnEditCut)  ON\_COMMAND(ID\_EDIT\_COPY, OnEditCopy)  ON\_COMMAND(ID\_EDIT\_PASTE, OnEditPaste)  ON\_COMMAND(ID\_EDIT\_DELETE, OnEditDelete)  ON\_COMMAND(ID\_EDIT\_UNDO, OnEditUndo)  ON\_UPDATE\_COMMAND\_UI(ID\_EDIT\_CUT, OnUpdateEditCut)  ON\_UPDATE\_COMMAND\_UI(ID\_EDIT\_COPY, OnUpdateEditCopy)  ON\_UPDATE\_COMMAND\_UI(ID\_EDIT\_PASTE, OnUpdateEditPaste)  ON\_UPDATE\_COMMAND\_UI(ID\_EDIT\_DELETE, OnUpdateEditDelete)  ON\_UPDATE\_COMMAND\_UI(ID\_EDIT\_UNDO, OnUpdateEditUndo)  ON\_COMMAND(ID\_FILE\_NEW, OnFileNew)  //}}AFX\_MSG\_MAP  END\_MESSAGE\_MAP()  ///////////////////////////////////////////////////////////////////////////  // CChildView message handlers  BOOL CChildView::PreCreateWindow(CREATESTRUCT& cs)  {  if (!CWnd::PreCreateWindow(cs))  return FALSE;  cs.dwExStyle ¦= WS\_EX\_CLIENTEDGE;  cs.style &= ~WS\_BORDER;  cs.lpszClass = AfxRegisterWndClass(CS\_HREDRAW¦CS\_VREDRAW¦CS\_DBLCLKS,  ::LoadCursor(NULL, IDC\_ARROW), HBRUSH(COLOR\_WINDOW+1), NULL);  return TRUE;  }  void CChildView::OnPaint()  {  CPaintDC dc(this);  }  int CChildView::OnCreate(LPCREATESTRUCT lpCreateStruct)  {  if (CWnd ::OnCreate(lpCreateStruct) == -1)  return -1;    m\_wndEdit.Create (WS\_CHILD ¦ WS\_VISIBLE ¦ WS\_VSCROLL ¦ ES\_MULTILINE ¦  ES\_AUTOVSCROLL, CRect (0, 0, 0, 0), this, IDC\_EDIT);  return 0;  }  void CChildView::OnSize(UINT nType, int cx, int cy)  {  CWnd ::OnSize(nType, cx, cy);  m\_wndEdit.MoveWindow (0, 0, cx, cy);  }  void CChildView::OnSetFocus(CWnd\* pOldWnd)  {  m\_wndEdit.SetFocus ();  }  void CChildView::OnEditCut()  {  m\_wndEdit.Cut ();  }  void CChildView::OnEditCopy()  {  m\_wndEdit.Copy ();  }  void CChildView::OnEditPaste()  {  m\_wndEdit.Paste ();  }  void CChildView::OnEditDelete()  {  m\_wndEdit.Clear ();  }  void CChildView::OnEditUndo()  {  m\_wndEdit.Undo ();  }  void CChildView::OnUpdateEditCut(CCmdUI\* pCmdUI)  {  pCmdUI->Enable (IsTextSelected ());  }  void CChildView::OnUpdateEditCopy(CCmdUI\* pCmdUI)  {  pCmdUI->Enable (IsTextSelected ());  }  void CChildView::OnUpdateEditPaste(CCmdUI\* pCmdUI)  {  pCmdUI->Enable (::IsClipboardFormatAvailable (CF\_TEXT));  }  void CChildView::OnUpdateEditDelete(CCmdUI\* pCmdUI)  {  pCmdUI->Enable (IsTextSelected ());  }  void CChildView::OnUpdateEditUndo(CCmdUI\* pCmdUI)  {  pCmdUI->Enable (m\_wndEdit.CanUndo ());  }  void CChildView::OnFileNew()  {  m\_wndEdit.SetWindowText (\_T (""));  }  BOOL CChildView::IsTextSelected()  {  int nStart, nEnd;  m\_wndEdit.GetSel (nStart, nEnd);  return (nStart != nEnd);  } |

## The *CComboBox* Class

The combo box combines a single-line edit control and a list box into one convenient package. Combo boxes come in three varieties: simple, drop-down, and drop-down list. Figure 7-7 shows a drop-down list combo box with its list displayed.



**Figure 7-7.** *A combo box with a drop-down list displayed.*

Simple combo boxes are the least used of the three combo box types. A simple combo box's list box is permanently displayed. When the user selects an item from the list, that item is automatically copied to the edit control. The user can also type text directly into the edit control. If the text the user enters matches an item in the list box, the item is automatically highlighted and scrolled into view.

A drop-down combo box differs from a simple combo box in that its list box is displayed only on demand. A drop-down list combo box works the same way but doesn't allow text to be typed into the edit control. This restriction effectively limits the user's selection to items appearing in the list box.

The style flags you pass to *Create* or *CreateEx* determine what type of combo box you create. CBS\_SIMPLE creates a simple combo box, CBS\_DROPDOWN creates a drop-down combo box, and CBS\_DROPDOWNLIST creates a drop-down list combo box. Other styles control additional aspects of the combo box's appearance and behavior, as shown in the table below. Many of these styles will look familiar because they're patterned after list box and edit control styles. CBS\_AUTOHSCROLL, for example, does the same thing for the edit control portion of a combo box control that ES\_AUTOHSCROLL does for a stand-alone edit control. When you create a combo box control, don't forget to include the style WS\_VSCROLL if you want the list box to have a vertical scroll bar and WS\_BORDER if you want the control's border to be visible. If *m\_wndComboBox* is a *CComboBox* object, the statement

|  |
| --- |
| m\_wndComboBox.Create (WS\_CHILD ¦ WS\_VISIBLE ¦ WS\_BORDER ¦  WS\_VSCROLL ¦ CBS\_DROPDOWNLIST ¦ CBS\_SORT, rect, this,  IDC\_COMBOBOX); |

creates a drop-down list combo box whose list box contains a vertical scroll bar when the number of items in the list box exceeds the number of items that can be displayed and that automatically sorts the items added to it. The control rectangle you specify in the call to *CComboBox::Create* should be large enough to encompass the list box part of the control as well as the edit box.

**Combo Box Styles**

|  |  |
| --- | --- |
| ***Style*** | ***Description*** |
| CBS\_AUTOHSCROLL | Enables horizontal scrolling in the edit control portion of a combo box. |
| CBS\_DISABLENOSCROLL | Disables the combo box list box's scroll bar when it isn't needed. Without this style, an unneeded scroll bar is hidden rather than disabled. |
| CBS\_DROPDOWN | Creates a drop-down combo box. |
| CBS\_DROPDOWNLIST | Creates a drop-down list combo box. |
| CBS\_HASSTRINGS | Creates a combo box that "remembers" the strings added to it. Conventional combo boxes have this style by default; owner-draw combo boxes don't. |
| CBS\_LOWERCASE | Forces all text in the combo box to lowercase. |
| CBS\_NOINTEGRALHEIGHT | Prevents the combo box's list box height from having to be an exact multiple of the item height. |
| CBS\_OEMCONVERT | A combo box whose edit control performs an ANSI-to-OEM-to-ANSI conversion on all characters so that the application won't get unexpected results if it performs an ANSI-to-OEM conversion of its own. Obsolete. |
| CBS\_OWNERDRAWFIXED | Creates an owner-draw combo box whose items are all the same height. |
| CBS\_OWNERDRAWVARIABLE | Creates an owner-draw combo box whose items can vary in height. |
| CBS\_SIMPLE | Creates a simple combo box. |
| CBS\_SORT | Automatically sorts items as they are added. |
| CBS\_UPPERCASE | Forces all text in the combo box to uppercase. |

Not surprisingly, the list of *CComboBox* member functions reads a lot like the list of member functions for *CEdit* and *CListBox*. Items are added to a combo box, for example, with *CComboBox::AddString* and *CComboBox::InsertString*, and the maximum character count for a combo box's edit control is set with *CComboBox::LimitText*. The *GetWindowText* and *SetWindowText* functions that *CComboBox* inherits from *CWnd* get and set the text in the edit control. Functions unique to combo boxes include *GetLBText*, which retrieves the text of an item identified by a 0-based index; *GetLBTextLen*, which returns the length of an item, in characters; *ShowDropDown*, which hides or displays the drop-down list box; and *GetDroppedState*, which returns a value indicating whether the drop-down list is currently displayed.

### Combo Box Notifications

Combo boxes send notifications to their parents much as edit controls and list boxes do. The following table lists the notifications the parent can expect, the corresponding MFC message-map macros, and the types of combo boxes the notifications apply to.

**Combo Box Notifications**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***Notification*** | ***Message-Macro Map*** | ***Simple*** | ***Drop-Down*** | ***Drop-Down List*** |
| CBN\_DROPDOWN Sent when the drop-down list is displayed. | ON\_CBN\_DROPDOWN |  | √ | √ |
| CBN\_CLOSEUP Sent when the drop-down list is closed. | ON\_CBN\_CLOSEUP |  | √ | √ |
| CBN\_DBLCLK Sent when an item is double-clicked. | ON\_CBN\_DBLCLK | √ |  |  |
| CBN\_SELCHANGE Sent when the selection changes. | ON\_CBN\_SELCHANGE | √ | √ | √ |
| CBN\_SELENDOK Sent when a selection is made. | ON\_CBN\_SELENDOK | √ | √ | √ |
| CBN\_SELENDCANCEL Sent when a selection is canceled. | ON\_CBN\_SELENDCANCEL |  | √ | √ |
| CBN\_EDITUPDATE Sent when the text in the edit control is about to change. | N\_CBN\_EDITUPDATE | √ | √ |  |
| CBN\_EDITCHANGE Sent when the text in the edit control has changed. | ON\_CBN\_EDITCHANGE | √ | √ |  |
| CBN\_KILLFOCUS Sent when the combo box loses the input focus. | ON\_CBN\_KILLFOCUS | √ | √ | √ |
| CBN\_SETFOCUS Sent when the combo box receives the input focus. | ON\_CBN\_SETFOCUS | √ | √ | √ |
| CBN\_ERRSPACE Sent when an operation fails because of insufficient memory. | ON\_CBN\_ERRSPACE | √ | √ | √ |

Not all notifications apply to all combo box types. CBN\_DROPDOWN and CBN\_CLOSEUP notifications, for example, aren't sent to CBS\_SIMPLE combo boxes because a simple combo box's list box doesn't open and close. By the same token, CBS\_DROPDOWN and CBS\_DROPDOWNLIST-style combo boxes don't receive CBN\_DBLCLK notifications because the items in their lists can't be double-clicked. (Why? Because the list box closes after the first click.) CBN\_EDITUPDATE and CBN\_EDITCHANGE notifications are equivalent to EN\_UPDATE and EN\_CHANGE notifications sent by edit controls, and CBN\_SELCHANGE is to combo boxes as LBN\_SELCHANGE is to list boxes.

One nuance you should be aware of when processing CBN\_SELCHANGE notifications is that when a notification arrives, the edit control might not have been updated to match the list box selection. Therefore, you should use *GetLBText* to retrieve the newly selected text instead of *GetWindowText*. You can get the index of the selected item with *CComboBox::GetCurSel*.

## The *CScrollBar* Class

MFC's *CScrollBar* class encapsulates scroll bar controls created from the "SCROLLBAR" WNDCLASS. Scroll bar controls are identical in most respects to the "window" scroll bars used in Chapter 2's Accel application. But whereas window scroll bars are created by adding WS\_VSCROLL and WS\_HSCROLL flags to the window style, scroll bar controls are created explicitly with *CScrollBar::Create*. And though a window scroll bar runs the full length of the window's client area and is inherently glued to the window border, scroll bar controls can be placed anywhere in the window and can be set to any height and width.

You create vertical scroll bars by specifying the style SBS\_VERT and horizontal scroll bars by specifying SBS\_HORZ. If *m\_wndVScrollBar* and *m\_wndHScrollBar* are *CScrollBar* objects, the statements

|  |
| --- |
| m\_wndVScrollBar.Create (WS\_CHILD ¦ WS\_VISIBLE ¦ WS\_BORDER ¦  SBS\_VERT, rectVert, this, IDC\_VSCROLLBAR);  m\_wndHScrollBar.Create (WS\_CHILD ¦ WS\_VISIBLE ¦ WS\_BORDER ¦  SBS\_HORZ, rectHorz, this, IDC\_HSCROLLBAR); |

create two scroll bar controls, one vertical and the other horizontal.

You can query Windows for the standard width of a vertical scroll bar or the standard height of a horizontal scroll bar with the *::GetSystemMetrics* API function. The following code fragment sets *nWidth* and *nHeight* to the system's standard scroll bar width and height:

|  |
| --- |
| int nWidth = ::GetSystemMetrics (SM\_CXVSCROLL);  int nHeight = ::GetSystemMetrics (SM\_CYHSCROLL); |

An alternative method for creating a scroll bar with a standard height or width is to specify the style SBS\_TOPALIGN, SBS\_BOTTOMALIGN, SBS\_LEFTALIGN, or SBS\_RIGHTALIGN when creating it. SBS\_LEFTALIGN and SBS\_RIGHTALIGN align a vertical scroll bar control along the left or right border of the rectangle specified in the call to *Create* and assign it a standard width. SBS\_TOPALIGN and SBS\_BOTTOMALIGN align a horizontal scroll bar control along the top or bottom border of the rectangle and assign it a standard height.

Unlike the other classic controls, scroll bar controls don't send WM\_COMMAND messages; they send WM\_VSCROLL and WM\_HSCROLL messages instead. MFC applications process these messages with *OnVScroll* and *OnHScroll* handlers, as described in Chapter 2. I didn't mention two scroll bar notification codes in Chapter 2 because they apply only to scroll bar controls. SB\_TOP means that the user pressed the Home key while the scroll bar had the input focus, and SB\_BOTTOM means the user pressed End.

MFC's *CScrollBar* class includes a handful of functions for manipulating scroll bars, most of which should seem familiar to you because they work just like the similarly named *CWnd* functions. *CScrollBar::GetScrollPos* and *CScrollBar::SetScrollPos* get and set the scroll bar's thumb position. *CScrollBar::GetScrollRange* and *CScrollBar::SetScrollRange* get and set the scroll bar range. You use *CScrollBar::SetScrollInfo* to set the range, position, and thumb size in one step. For details, refer to the discussion of *CWnd::SetScrollInfo* in Chapter 2.

# Advanced Control Programming

One of the benefits of programming controls the MFC way is the ease with which you can modify a control's behavior by deriving classes of your own from the MFC control classes. It's easy, for example, to create an edit control that accepts only numbers or a list box that displays pictures instead of text. You can also build reusable, self-contained control classes that respond to their own notification messages.

The remainder of this chapter is about techniques you can use to shape the controls to make them work the way *you* want them to work by combining the best features of C++ and MFC.

## Numeric Edit Controls

The MFC control classes are useful in their own right because they provide an object-oriented interface to the built-in control types. But their utility is enhanced by the fact that you can use them as base classes for control classes of your own. By adding new message handlers to a derived class or overriding message handlers acquired through inheritance, you can modify certain aspects of the control's behavior while leaving other aspects unchanged.

A perfect example of a derived control class is a numeric edit control. A normal edit control accepts a wide range of characters, including numbers, letters of the alphabet, and punctuation symbols. A numeric edit control accepts only numbers. It's perfect for entering phone numbers, serial numbers, IP addresses, and other numeric data.

Creating a numeric edit control is no big deal in an MFC application because the basic features of an edit control are defined in *CEdit*. Thanks to C++ inheritance and MFC message mapping, you can derive a control class from *CEdit* and supply custom message handlers to change the way the control responds to user input. The following *CNumEdit* class models an edit control that accepts numbers but rejects all other characters:

|  |
| --- |
| class CNumEdit : public CEdit  {  protected:  afx\_msg void OnChar (UINT nChar, UINT nRepCnt, UINT nFlags);  DECLARE\_MESSAGE\_MAP ()  };  BEGIN\_MESSAGE\_MAP (CNumEdit, CEdit)  ON\_WM\_CHAR ()  END\_MESSAGE\_MAP ()  void CNumEdit::OnChar (UINT nChar, UINT nRepCnt, UINT nFlags)  {  if (((nChar >= \_T (`0')) && (nChar <= \_T (`9'))) ¦¦  (nChar == VK\_BACK))  CEdit::OnChar (nChar, nRepCnt, nFlags);  } |

How does *CNumEdit* work? When an edit control has the input focus and a character key is pressed, the control receives a WM\_CHAR message. By deriving a new class from *CEdit*, mapping WM\_CHAR messages to the derived class's *OnChar* handler, and designing *OnChar* so that it passes WM\_CHAR messages to the base class if and only if the character encoded in the message is a number, you create an edit control that rejects nonnumeric characters. VK\_BACK is included in the list of acceptable character codes so that the Backspace key won't cease to function. It's not necessary to test for other editing keys such as Home and Del because they, unlike the Backspace key, don't generate WM\_CHAR messages.

## Owner-Draw List Boxes

By default, items in a list box consist of strings of text. Should you need a list box that displays graphical images instead of text, you can create an owner-draw list box—one whose contents are drawn by your application, not by Windows—by following two simple steps.

1. Derive a new list box class from *CListBox,* and override *CListBox::MeasureItem* and *CListBox::DrawItem*. Also override *PreCreateWindow,* and make sure that either LBS\_OWNERDRAWFIXED or LBS\_OWNERDRAWVARIABLE is included in the list box style.
2. Instantiate the derived class, and use *Create* or *CreateEx* to create the list box.

Functionally, owner-draw list boxes are similar to owner-draw menus. When an item in an owner-draw list box needs to be drawn (or redrawn), Windows sends the list box's parent a WM\_DRAWITEM message with a pointer to a DRAWITEMSTRUCT structure containing a device context handle, a 0-based index identifying the item to be drawn, and other information. Before the first WM\_DRAWITEM message arrives, the list box's parent receives one or more WM\_MEASUREITEM messages requesting the height of the list box's items. If the list box style is LBS\_OWNERDRAWFIXED, WM\_MEASUREITEM is sent just once. For LBS\_OWNERDRAWVARIABLE list boxes, a WM\_MEASUREITEM message is sent for each item. MFC calls the list box object's virtual *DrawItem* function when the parent receives a WM\_DRAWITEM message and *MeasureItem* when it receives a WM\_MEASUREITEM message. Therefore, you don't have to modify the parent window class or worry about message maps and message handlers; just override *DrawItem* and *MeasureItem* in the list box class, and your list box can do its own drawing without any help from its parent.

*CListBox* supports two other owner-draw overridables in addition to *DrawItem* and *MeasureItem*. The first is *CompareItem*. If an owner-draw list box is created with the style LBS\_SORT and items are added to it with *AddString*, *CListBox::CompareItem* must be overridden with a version that compares two arbitrary items packaged in COMPAREITEMSTRUCT structures. The overridden function must return -1 if item 1 comes before item 2, 0 if the items are lexically equal, or 1 if item 1 comes after item 2. Owner-draw list boxes are seldom created with the style LBS\_SORT because nontextual data typically has no inherent order. (How would you sort a list of colors, for example?) And if you don't use LBS\_SORT, you don't have to write a *CompareItem* function. If you don't implement *CompareItem* in a derived owner-draw list box class, it's prudent to override *PreCreateWindow* and make sure the list box style doesn't include LBS\_SORT.

The final owner-draw list box overridable is *DeleteItem*. It's called when an item is deleted with *DeleteString*, when the list box's contents are erased with *ResetContent*, and when a list box containing one or more items is destroyed. *DeleteItem* is called once per item, and it receives a pointer to a DELETEITEMSTRUCT structure containing information about the item. If a list box uses per-item resources (for example, bitmaps) that need to be freed when an item is removed or the list box is destroyed, override *DeleteItem* and use it to free those resources.

The following *COwnerDrawListBox* class is a nearly complete C++ implementation of an LBS\_OWNERDRAWFIXED-style owner-draw list box:

|  |
| --- |
| class COwnerDrawListBox : public CListBox  {  public:  virtual BOOL PreCreateWindow (CREATESTRUCT&);  virtual void MeasureItem (LPMEASUREITEMSTRUCT);  virtual void DrawItem (LPDRAWITEMSTRUCT);  };  BOOL COwnerDrawListBox::PreCreateWindow (CREATESTRUCT& cs)  {  if (!CListBox::PreCreateWindow (cs))  return FALSE;  cs.style &= ~(LBS\_OWNERDRAWVARIABLE ¦ LBS\_SORT);  cs.style ¦= LBS\_OWNERDRAWFIXED;  return TRUE;  }  void COwnerDrawListBox::MeasureItem (LPMEASUREITEMSTRUCT lpmis)  {  lpmis->itemHeight = 32; // Item height in pixels  }  void COwnerDrawListBox::DrawItem (LPDRAWITEMSTRUCT lpdis)  {  CDC dc;  dc.Attach (lpdis->hDC);  CRect rect = lpdis->rcItem;  UINT nIndex = lpdis->itemID;  CBrush\* pBrush = new CBrush (::GetSysColor ((lpdis->itemState &  ODS\_SELECTED) ? COLOR\_HIGHLIGHT : COLOR\_WINDOW));  dc.FillRect (rect, pBrush);  delete pBrush;  if (lpdis->itemState & ODS\_FOCUS)  dc.DrawFocusRect (rect);  if (nIndex != (UINT) -1) {  // Draw the item.  }  dc.Detach ();  } |

Before you use *COwnerDrawListBox* in an application of your own, change the 32 in *COwnerDrawListBox::MeasureItem* to the desired item height in pixels and replace the comment "Draw the item" in *COwnerDrawListBox::DrawItem* with code that draws the item whose index is *nIndex*. Use the *dc* device context object to do the drawing and restrict your output to the rectangle specified by *rect*, and the list box should function superbly. (Be sure to preserve the state of the device context so that it's the same going out as it was coming in.) *COwnerDrawListBox*'s implementation of *DrawItem* paints the item's background with the system color COLOR\_HIGHLIGHT if the item is selected (if the *lpdis*->*itemState*'s ODS\_SELECTED bit is set) or COLOR\_WINDOW if it isn't, and it draws a focus rectangle if the item has the input focus (if the *lpdis*->*itemState*'s ODS\_FOCUS bit is set). All you have to do is draw the item itself. The *PreCreateWindow* override ensures that LBS\_OWNERDRAWFIXED is set and that LBS\_OWNERDRAWVARIABLE isn't. It also clears the LBS\_SORT bit to prevent calls to *CompareItem*.

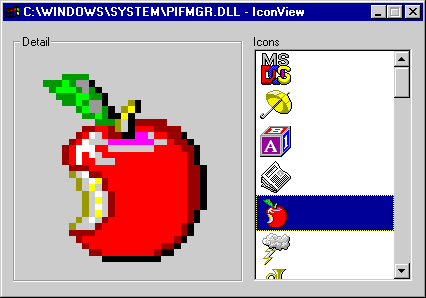
A final feature needed to transform *COwnerDrawListBox* into a complete class is an *AddItem* function that can be called to add a nontextual item to the list box. For a list box that displays bitmaps, for example, *AddItem* might look like this:

|  |
| --- |
| int COwnerDrawListBox::AddItem (HBITMAP hBitmap)  {  int nIndex = AddString (\_T (""));  if ((nIndex != LB\_ERR) && (nIndex != LB\_ERRSPACE))  SetItemData (nIndex, (DWORD) hBitmap);  return nIndex;  } |

In this example, *AddItem* uses *SetItemData* to associate a bitmap handle with a list box index. For a given item, the list box's *DrawItem* function can retrieve the bitmap handle with *GetItemData* and draw the bitmap. Bitmaps are resources that must be deleted when they're no longer needed. You can either leave it to the list box's parent to delete the bitmaps or override *CListBox::DeleteItem* and let the list box delete them itself. The choice is up to you.

The IconView application shown in Figure 7-8 uses an owner-draw list box class named *CIconListBox* to displays icons. *CIconListBox* overrides the *PreCreateWindow*, *MeasureItem*, and *DrawItem* functions it inherits from *CListBox* and adds two functions of its own. *AddIcon* adds an icon to the list box, and *ProjectImage "*projects" an icon onto a display surface, shrinking or expanding the image as needed to fit a specified rectangle. IconView's source code is shown in Figure 7-9.

The only form of input that IconView accepts is drag-and-drop. To try it out, grab an EXE, DLL, or ICO file with the left mouse button, drag it to the IconView window, and release the mouse button. Any icons contained in the file will be displayed in the list box, and an enlarged image of the first icon will be displayed in the Detail window. To get a close-up view of any of the other icons in the file, just click the icon or cursor through the list with the up and down arrow keys.



**Figure 7-8.** *IconView showing the icons contained in Pifmgr.dll.*

IconView uses MFC's handy *CDC::DrawIcon* function to draw icons into the list box. The core code is found in *CIconListBox::DrawItem*:

|  |
| --- |
| if (nIndex != (UINT) -1)  dc.DrawIcon (rect.left + 4, rect.top + 2,  (HICON) GetItemData (nIndex)); |

Icon handles are stored with *SetItemData* and retrieved with *GetItemData*. The call to *DrawIcon* is skipped if *nIndex*—the index of the currently selected list box item—is -1. That's important, because *DrawItem* is called with a list box index of -1 when an empty list box receives the input focus. *DrawItem*'s job in that case is to draw a focus rectangle around the nonexistent item 0. You shouldn't assume that *DrawItem* will always be called with a valid item index.

*CMainWindow*'s *OnPaint* handler does nothing more than construct a paint device context and call the list box's *ProjectImage* function to draw a blown-up version of the currently selected icon in the window's client area. *ProjectImage* uses the *CDC* functions *BitBlt* and *StretchBlt* to project the image. This code probably won't make a lot of sense to you right now, but its meaning will be crystal clear once you've read about bitmaps in [Chapter 15](mk:@MSITStore:C:\Program%20Files%20(x86)\MSPress\BooksOnline\Programming%20Windows%20with%20MFC%20Second%20Edition\progmfc2.chm::/ch15a.htm).

The drag-and-drop mechanism that IconView uses is a primitive form of drag-and-drop that was introduced in Windows 3.1. Briefly, the call to *DragAcceptFiles* in *CMainWindow::OnCreate* registers *CMainWindow* as a drop target. Once registered, the window receives a WM\_DROPFILES message whenever a file is dragged from the shell and dropped on top of it. *CMainWindow::OnDropFiles* responds to WM\_DROPFILES messages by using the *::DragQueryFile* API function to retrieve the name of the file that was dropped. It then uses *::ExtractIcon* to extract icons from the file and *CIconListBox::AddIcon* to add the icons to the list box.

In [Chapter 19](mk:@MSITStore:C:\Program%20Files%20(x86)\MSPress\BooksOnline\Programming%20Windows%20with%20MFC%20Second%20Edition\progmfc2.chm::/ch19a.htm), you'll learn about a richer form of drag-and-drop called *OLE drag-and-drop*. "Old" drag-and-drop is still supported in 32-bit Windows, but it's not nearly as flexible as OLE drag-and-drop. That's why I haven't gone into more detail about it. Once you see OLE drag-and-drop in action, I think you'll agree that time spent understanding Windows 3.1-style drag-and-drop is time better spent elsewhere.

**Figure 7-9.** *The IconView application.*

|  |
| --- |
| IconView.h class CMyApp : public CWinApp  {  public:  virtual BOOL InitInstance ();  };  class CIconListBox : public CListBox  {  public:  virtual BOOL PreCreateWindow (CREATESTRUCT& cs);  virtual void MeasureItem (LPMEASUREITEMSTRUCT lpmis);  virtual void DrawItem (LPDRAWITEMSTRUCT lpdis);  int AddIcon (HICON hIcon);  void ProjectImage (CDC\* pDC, LPRECT pRect, COLORREF clrBackColor);  };  class CMainWindow : public CWnd  {  protected:  int m\_cxChar;  int m\_cyChar;  CFont m\_font;  CRect m\_rcImage;  CButton m\_wndGroupBox;  CIconListBox m\_wndIconListBox;  CStatic m\_wndLabel;  public:  CMainWindow ();  protected:  virtual void PostNcDestroy ();  afx\_msg int OnCreate (LPCREATESTRUCT lpcs);  afx\_msg void OnPaint ();  afx\_msg void OnSetFocus (CWnd\* pWnd);  afx\_msg void OnDropFiles (HDROP hDropInfo);  afx\_msg void OnSelChange ();  DECLARE\_MESSAGE\_MAP ()  }; |

|  |
| --- |
| IconView.cpp #include <afxwin.h>  #include "IconView.h"  #define IDC\_LISTBOX 100  CMyApp myApp;  /////////////////////////////////////////////////////////////////////////  // CMyApp member functions  BOOL CMyApp::InitInstance ()  {  m\_pMainWnd = new CMainWindow;  m\_pMainWnd->ShowWindow (m\_nCmdShow);  m\_pMainWnd->UpdateWindow ();  return TRUE;  }  /////////////////////////////////////////////////////////////////////////  // CMainWindow message map and member functions  BEGIN\_MESSAGE\_MAP (CMainWindow, CWnd)  ON\_WM\_CREATE ()  ON\_WM\_PAINT ()  ON\_WM\_SETFOCUS ()  ON\_WM\_DROPFILES ()  ON\_LBN\_SELCHANGE (IDC\_LISTBOX, OnSelChange)  END\_MESSAGE\_MAP ()  CMainWindow::CMainWindow ()  {  CString strWndClass = AfxRegisterWndClass (  0,  myApp.LoadStandardCursor (IDC\_ARROW),  (HBRUSH) (COLOR\_3DFACE + 1),  myApp.LoadStandardIcon (IDI\_WINLOGO)  );  CreateEx (0, strWndClass, \_T ("IconView"),  WS\_OVERLAPPED ¦ WS\_SYSMENU ¦ WS\_CAPTION ¦ WS\_MINIMIZEBOX,  CW\_USEDEFAULT, CW\_USEDEFAULT, CW\_USEDEFAULT, CW\_USEDEFAULT,  NULL, NULL, NULL);  CRect rect (0, 0, m\_cxChar \* 84, m\_cyChar \* 21);  CalcWindowRect (&rect);  SetWindowPos (NULL, 0, 0, rect.Width (), rect.Height (),  SWP\_NOZORDER ¦ SWP\_NOMOVE ¦ SWP\_NOREDRAW);  }  int CMainWindow::OnCreate (LPCREATESTRUCT lpcs)  {  if (CWnd::OnCreate (lpcs) == -1)  return -1;  m\_font.CreatePointFont (80, \_T ("MS Sans Serif"));  CClientDC dc (this);  CFont\* pOldFont = dc.SelectObject (&m\_font);  TEXTMETRIC tm;  dc.GetTextMetrics (&tm);  m\_cxChar = tm.tmAveCharWidth;  m\_cyChar = tm.tmHeight + tm.tmExternalLeading;  dc.SelectObject (pOldFont);  m\_rcImage.SetRect (m\_cxChar \* 4, m\_cyChar \* 3, m\_cxChar \* 46,  m\_cyChar \* 19);  m\_wndGroupBox.Create (\_T ("Detail"), WS\_CHILD ¦ WS\_VISIBLE ¦ BS\_GROUPBOX,  CRect (m\_cxChar \* 2, m\_cyChar, m\_cxChar \* 48, m\_cyChar \* 20),  this, (UINT) -1);  m\_wndLabel.Create (\_T ("Icons"), WS\_CHILD ¦ WS\_VISIBLE ¦ SS\_LEFT,  CRect (m\_cxChar \* 50, m\_cyChar, m\_cxChar \* 82, m\_cyChar \* 2),  this);  m\_wndIconListBox.Create (WS\_CHILD ¦ WS\_VISIBLE ¦ WS\_VSCROLL ¦  WS\_BORDER ¦ LBS\_NOTIFY ¦ LBS\_NOINTEGRALHEIGHT,  CRect (m\_cxChar \* 50, m\_cyChar \* 2, m\_cxChar \* 82, m\_cyChar \* 20),  this, IDC\_LISTBOX);  m\_wndGroupBox.SetFont (&m\_font);  m\_wndLabel.SetFont (&m\_font);  DragAcceptFiles ();  return 0;  }  void CMainWindow::PostNcDestroy ()  {  delete this;  }  void CMainWindow::OnPaint ()  {  CPaintDC dc (this);  m\_wndIconListBox.ProjectImage (&dc, m\_rcImage,  ::GetSysColor (COLOR\_3DFACE));  }  void CMainWindow::OnSetFocus (CWnd\* pWnd)  {  m\_wndIconListBox.SetFocus ();  }  void CMainWindow::OnDropFiles (HDROP hDropInfo)  {  //  // Find out how many files were dropped.  //  int nCount = ::DragQueryFile (hDropInfo, (UINT) -1, NULL, 0);  if (nCount == 1) { // One file at a time, please  m\_wndIconListBox.ResetContent ();  //  // Extract the file's icons and add them to the list box.  //  char szFile[MAX\_PATH];  ::DragQueryFile (hDropInfo, 0, szFile, sizeof (szFile));  int nIcons = (int) ::ExtractIcon (NULL, szFile, (UINT) -1);  if (nIcons) {  HICON hIcon;  for (int i=0; i<nIcons; i++) {  hIcon = ::ExtractIcon (AfxGetInstanceHandle (),  szFile, i);  m\_wndIconListBox.AddIcon (hIcon);  }  }  //  // Put the file name in the main window's title bar.  //  CString strWndTitle = szFile;  strWndTitle += \_T (" - IconView");  SetWindowText (strWndTitle);  //  // Select item number 0.  //  CClientDC dc (this);  m\_wndIconListBox.SetCurSel (0);  m\_wndIconListBox.ProjectImage (&dc, m\_rcImage,  ::GetSysColor (COLOR\_3DFACE));  }  ::DragFinish (hDropInfo);  }  void CMainWindow::OnSelChange ()  {  CClientDC dc (this);  m\_wndIconListBox.ProjectImage (&dc, m\_rcImage,  ::GetSysColor (COLOR\_3DFACE));  }  /////////////////////////////////////////////////////////////////////////  // CIconListBox member functions  BOOL CIconListBox::PreCreateWindow (CREATESTRUCT& cs)  {  if (!CListBox::PreCreateWindow (cs))  return FALSE;  cs.dwExStyle ¦= WS\_EX\_CLIENTEDGE;  cs.style &= ~(LBS\_OWNERDRAWVARIABLE ¦ LBS\_SORT);  cs.style ¦= LBS\_OWNERDRAWFIXED;  return TRUE;  }  void CIconListBox::MeasureItem (LPMEASUREITEMSTRUCT lpmis)  {  lpmis->itemHeight = 36;  }  void CIconListBox::DrawItem (LPDRAWITEMSTRUCT lpdis)  {  CDC dc;  dc.Attach (lpdis->hDC);  CRect rect = lpdis->rcItem;  int nIndex = lpdis->itemID;  CBrush\* pBrush = new CBrush;  pBrush->CreateSolidBrush (::GetSysColor ((lpdis->itemState &  ODS\_SELECTED) ? COLOR\_HIGHLIGHT : COLOR\_WINDOW));  dc.FillRect (rect, pBrush);  delete pBrush;  if (lpdis->itemState & ODS\_FOCUS)  dc.DrawFocusRect (rect);  if (nIndex != (UINT) -1)  dc.DrawIcon (rect.left + 4, rect.top + 2,  (HICON) GetItemData (nIndex));  dc.Detach ();  }  int CIconListBox::AddIcon (HICON hIcon)  {  int nIndex = AddString (\_T (""));  if ((nIndex != LB\_ERR) && (nIndex != LB\_ERRSPACE))  SetItemData (nIndex, (DWORD) hIcon);  return nIndex;  }  void CIconListBox::ProjectImage (CDC\* pDC, LPRECT pRect,  COLORREF clrBackColor)  {  CDC dcMem;  dcMem.CreateCompatibleDC (pDC);  CBitmap bitmap;  bitmap.CreateCompatibleBitmap (pDC, 32, 32);  CBitmap\* pOldBitmap = dcMem.SelectObject (&bitmap);  CBrush\* pBrush = new CBrush (clrBackColor);  dcMem.FillRect (CRect (0, 0, 32, 32), pBrush);  delete pBrush;  int nIndex = GetCurSel ();  if (nIndex != LB\_ERR)  dcMem.DrawIcon (0, 0, (HICON) GetItemData (nIndex));  pDC->StretchBlt (pRect->left, pRect->top, pRect->right - pRect->left,  pRect->bottom - pRect->top, &dcMem, 0, 0, 32, 32, SRCCOPY);  dcMem.SelectObject (pOldBitmap);  } |

## Graphical Push Buttons

MFC includes three derived control classes of its own: *CCheckListBox*, *CDragListBox*, and *CBitmapButton*. *CCheckListBox* turns a normal list box into a "check" list box—a list box with a check box by each item and added functions such as *GetCheck* and *SetCheck* for getting and setting check box states. *CDragListBox* creates a list box that supports its own primitive form of drag-and-drop. *CBitmapButton* encapsulates owner-draw push buttons that display pictures instead of text. It supplies its own *DrawItem* handler that draws a push button in response to WM\_DRAWITEM messages. All you have to do is create the button and supply four bitmaps representing the button in various states.

*CBitmapButton* was a boon back in the days of 16-bit Windows because it simplified the task of creating graphical push buttons. Today, however, owner-draw push buttons are rarely used. Two button styles that were first introduced in Windows 95—BS\_BITMAP and BS\_ICON—make graphical push buttons a breeze by taking a single image and creating a push button from it. A BS\_BITMAP-style push button (henceforth, a *bitmap push button*) displays a bitmap on the face of a push button. A BS\_ICON-style push button (an *icon push button*) displays an icon. Most developers prefer icon push buttons because icons, unlike bitmaps, can have transparent pixels. Transparent pixels are great for displaying nonrectangular images on button faces because they decouple the image's background color from the button color.

Creating an icon push button is a two-step process:

1. Create a push button whose style includes a BS\_ICON flag.
2. Call the button's *SetIcon* function, and pass in an icon handle.

The following example creates an icon push button from an icon whose resource ID is IDI\_OK:

|  |
| --- |
| m\_wndIconButton.Create (\_T (""), WS\_CHILD ¦ WS\_VISIBLE ¦ BS\_ICON,  rect, this, IDC\_BUTTON);  m\_wndIconButton.SetIcon (AfxGetApp ()->LoadIcon (IDI\_OK)); |

The icon is drawn in the center of the button unless you alter its alignment by applying one or more of the following button styles:

|  |  |
| --- | --- |
| ***Button Style*** | ***Description*** |
| BS\_LEFT | Aligns the icon image with the left edge of the button face |
| BS\_RIGHT | Aligns the icon image with the right edge of the button face |
| BS\_TOP | Aligns the icon image with the top edge of the button face |
| BS\_BOTTOM | Aligns the icon image with the bottom edge of the button face |
| BS\_CENTER | Centers the icon image horizontally |
| BS\_VCENTER | Centers the icon image vertically |

Chapter 8's Phone application uses icon push buttons to represent the OK and Cancel buttons in a dialog box.

The procedure for creating a bitmap button is almost the same as the one for creating an icon button. Just change BS\_ICON to BS\_BITMAP and *SetIcon* to *SetBitmap* and you're set. Of course, you'll have to replace the call to *LoadIcon* with code that loads a bitmap, too. You'll learn how that's done in [Chapter 15](mk:@MSITStore:C:\Program%20Files%20(x86)\MSPress\BooksOnline\Programming%20Windows%20with%20MFC%20Second%20Edition\progmfc2.chm::/ch15a.htm).

One problem to watch out for when you're using icon push buttons is what happens when the button becomes disabled. Windows generates a disabled button image from the button's icon, but the results aren't always what you'd expect. In general, the simpler the image, the better. Unfilled figures render better when disabled than filled figures.

## Customizing a Control's Colors

The most glaring deficiency in the Windows control architecture is that there's no obvious way to change a control's colors. You can change a control's font with *SetFont*, but there is no equivalent function for changing a control's colors.

MFC supports two mechanisms for changing a control's colors. Both rely on the fact that before a control paints itself, it sends its parent a message containing the handle of the device context used to do the painting. The parent can call *CDC*::*SetTextColor* and *CDC::SetBkColor* on that device context to alter the attributes of any text drawn by the control. It can also alter the control's background color by returning a brush handle (HBRUSH).

The message that a control sends to its parent prior to painting varies with the control type. For example, a list box sends a WM\_CTLCOLORLISTBOX message; a static control sends a WM\_CTLCOLORSTATIC message. In any event, the message's *wParam* holds the device context handle, and *lParam* holds the control's window handle. If a window processes a static control's WM\_CTLCOLORSTATIC messages by setting the device context's text color to red and background color to white and returning a brush handle for a blue brush, the control text will be red, the gaps in and between characters will be white, and the control background—everything inside the control's borders not covered by text—will be blue.

MFC's ON\_WM\_CTLCOLOR message-map macro directs WM\_CTLCOLOR messages of all types to a handler named *OnCtlColor*. *OnCtlColor* is prototyped as follows:

|  |
| --- |
| afx\_msg HBRUSH OnCtlColor (CDC\* pDC, CWnd\* pWnd, UINT nCtlColor) |

*pDC* is a pointer to the control's device context, *pWnd* is a *CWnd* pointer that identifies the control itself, and *nCtlColor* identifies the type of WM\_CTLCOLOR message that prompted the call. Here are the possible values for *nCtlColor*.

|  |  |
| --- | --- |
| ***nCtlColor*** | ***Control Type or Window Type*** |
| CTLCOLOR\_BTN | Push button. Processing this message has no effect on a button's appearance. |
| CTLCOLOR\_DLG | Dialog box. |
| CTLCOLOR\_EDIT | Edit control and the edit control part of a combo box. |
| CTLCOLOR\_LISTBOX | List box and the list box part of a combo box. |
| CTLCOLOR\_MSGBOX | Message box. |
| CTLCOLOR\_SCROLLBAR | Scroll bar. |
| CTLCOLOR\_STATIC | Static control, check box, radio button, group box, read-only or disabled edit control, and the edit control in a disabled combo box. |

Five *nCtlColor* values pertain to controls, and two—CTLCOLOR\_DLG and CTLCOLOR\_MSGBOX—apply to dialog boxes and message boxes. (That's right: You can control the color of dialog boxes and message boxes by processing WM\_CTLCOLOR messages.) Static controls aren't the only controls that send WM\_CTLCOLORSTATIC messages. You'd think that a radio button would send a WM\_CTLCOLORBTN message, but in fact it sends a WM\_CTLCOLORSTATIC message in 32-bit Windows.

One way, then, to change a control's colors is to implement *OnCtlColor* in the parent window class. The following *OnCtlColor* implementation changes the color of a static text control named *m\_wndText* to white-on-red in a frame window:

|  |
| --- |
| HBRUSH CMainWindow::OnCtlColor (CDC\* pDC, CWnd\* pWnd,  UINT nCtlColor)  {  if (m\_wndText.m\_hWnd == pWnd->m\_hWnd) {  pDC->SetTextColor (RGB (255, 255, 255));  pDC->SetBkColor (RGB (255, 0, 0));  return (HBRUSH) m\_brRedBrush;  }  CFrameWnd::OnCtlColor (pDC, pWnd, nCtlColor);  } |

*m\_brRedBrush* is a *CMainWindow* data member whose type is *CBrush*. It is initialized as follows:

|  |
| --- |
| m\_brRedBrush.CreateSolidBrush (RGB (255, 0, 0)); |

Note that this implementation of *OnCtlColor* compares the window handle of the control whose color it wishes to change with the window handle of the control that generated the message. If the two are not equal, the message is forwarded to the base class. If this check were not performed, *OnCtlColor* would affect all the controls in *CMainWindow*, not just *m\_wndText*.

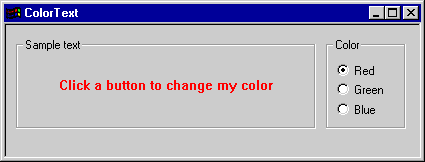
That's one way to change a control's color. The problem with this technique is that it's up to the parent to do the changing. What happens if you want to derive a control class of your own and include in it a *SetColor* function for modifying the control's color?

A derived control class can set its own colors by using MFC's ON\_WM\_CTLCOLOR\_REFLECT macro to pass WM\_CTLCOLOR messages that aren't handled by the control's parent back to the control. Here's the code for a *CStatic*-like control that paints itself white-on-red:

|  |
| --- |
| class CColorStatic : public CStatic  {  public:  CColorStatic ();  protected:  CBrush m\_brRedBrush;  afx\_msg HBRUSH CtlColor (CDC\* pDC, UINT nCtlColor);  DECLARE\_MESSAGE\_MAP ()  };  BEGIN\_MESSAGE\_MAP (CColorStatic, CStatic)  ON\_WM\_CTLCOLOR\_REFLECT ()  END\_MESSAGE\_MAP ()  CColorStatic::CColorStatic ()  {  m\_brRedBrush.CreateSolidBrush (RGB (255, 0, 0));  }  HBRUSH CColorStatic::CtlColor (CDC\* pDC, UINT nCtlColor)  {  pDC->SetTextColor (RGB (255, 255, 255));  pDC->SetBkColor (RGB (255, 0, 0));  return (HBRUSH) m\_brRedBrush;  } |

*CtlColor* is similar to *OnCtlColor*, but it doesn't receive the *pWnd* parameter that *OnCtlColor* does. It doesn't need to because the control to which the message applies is implicit in the call.

The ColorText application shown in Figure 7-10 uses a static text control whose colors are configurable. *CColorStatic* implements the control. This version of *CColorStatic* is more versatile than the one in the previous paragraph because rather than use hardcoded colors, it includes member functions named *SetTextColor* and *SetBkColor* that can be used to change its colors. When ColorText's Red, Green, or Blue radio button is clicked, the control's text color changes. The button click activates a handler that calls the control's *SetTextColor* function. (See Figure 7-11.) ColorText doesn't use the control's *SetBkColor* function, but I included the function anyway for completeness. *SetBkColor* controls the fill color drawn behind the text. *CColorStatic*'sdefault colors are black (foreground) and the system color COLOR\_3DFACE (background), but a simple function call is sufficient to change either one.



**Figure 7-10.** *The ColorText window.*

**Figure 7-11.** *The ColorText application.*

|  |
| --- |
| ColorText.h #define IDC\_RED 100  #define IDC\_GREEN 101  #define IDC\_BLUE 102  class CColorStatic : public CStatic  {  protected:  COLORREF m\_clrText;  COLORREF m\_clrBack;  CBrush m\_brBkgnd;  public:  CColorStatic ();  void SetTextColor (COLORREF clrText);  void SetBkColor (COLORREF clrBack);  protected:  afx\_msg HBRUSH CtlColor (CDC\* pDC, UINT nCtlColor);  DECLARE\_MESSAGE\_MAP ()  };  class CMyApp : public CWinApp  {  public:  virtual BOOL InitInstance ();  };  class CMainWindow : public CFrameWnd  {  protected:  int m\_cxChar;  int m\_cyChar;  CFont m\_font;  CColorStatic m\_wndText;  CButton m\_wndRadioButtonRed;  CButton m\_wndRadioButtonGreen;  CButton m\_wndRadioButtonBlue;  CButton m\_wndGroupBox1;  CButton m\_wndGroupBox2;  public:  CMainWindow ();  protected:  afx\_msg int OnCreate (LPCREATESTRUCT lpcs);  afx\_msg void OnRedButtonClicked ();  afx\_msg void OnGreenButtonClicked ();  afx\_msg void OnBlueButtonClicked ();  DECLARE\_MESSAGE\_MAP ()  }; |

|  |
| --- |
| ColorText.cpp #include <afxwin.h>  #include "ColorText.h"  CMyApp myApp;  /////////////////////////////////////////////////////////////////////////  // CMyApp member functions  BOOL CMyApp::InitInstance ()  {  m\_pMainWnd = new CMainWindow;  m\_pMainWnd->ShowWindow (m\_nCmdShow);  m\_pMainWnd->UpdateWindow ();  return TRUE;  }  /////////////////////////////////////////////////////////////////////////  // CMainWindow message map and member functions  BEGIN\_MESSAGE\_MAP (CMainWindow, CFrameWnd)  ON\_WM\_CREATE ()  ON\_BN\_CLICKED (IDC\_RED, OnRedButtonClicked)  ON\_BN\_CLICKED (IDC\_GREEN, OnGreenButtonClicked)  ON\_BN\_CLICKED (IDC\_BLUE, OnBlueButtonClicked)  END\_MESSAGE\_MAP ()  CMainWindow::CMainWindow ()  {  CString strWndClass = AfxRegisterWndClass (  0,  myApp.LoadStandardCursor (IDC\_ARROW),  (HBRUSH) (COLOR\_3DFACE + 1),  myApp.LoadStandardIcon (IDI\_WINLOGO)  );  Create (strWndClass, \_T ("ColorText"));  }  int CMainWindow::OnCreate (LPCREATESTRUCT lpcs)  {  if (CFrameWnd::OnCreate (lpcs) == -1)  return -1;  m\_font.CreatePointFont (80, \_T ("MS Sans Serif"));  CClientDC dc (this);  CFont\* pOldFont = dc.SelectObject (&m\_font);  TEXTMETRIC tm;  dc.GetTextMetrics (&tm);  m\_cxChar = tm.tmAveCharWidth;  m\_cyChar = tm.tmHeight + tm.tmExternalLeading;  dc.SelectObject (pOldFont);  m\_wndGroupBox1.Create (\_T ("Sample text"), WS\_CHILD ¦ WS\_VISIBLE ¦  BS\_GROUPBOX, CRect (m\_cxChar \* 2, m\_cyChar, m\_cxChar \* 62,  m\_cyChar \* 8), this, UINT (-1));  m\_wndText.Create (\_T ("Click a button to change my color"),  WS\_CHILD ¦ WS\_VISIBLE ¦ SS\_CENTER, CRect (m\_cxChar \* 4,  m\_cyChar \* 4, m\_cxChar \* 60, m\_cyChar \* 6), this);  m\_wndGroupBox2.Create (\_T ("Color"), WS\_CHILD ¦ WS\_VISIBLE ¦  BS\_GROUPBOX, CRect (m\_cxChar \* 64, m\_cyChar, m\_cxChar \* 80,  m\_cyChar \* 8), this, UINT (-1));  m\_wndRadioButtonRed.Create (\_T ("Red"), WS\_CHILD ¦ WS\_VISIBLE ¦  WS\_GROUP ¦ BS\_AUTORADIOBUTTON, CRect (m\_cxChar \* 66, m\_cyChar \* 3,  m\_cxChar \* 78, m\_cyChar \* 4), this, IDC\_RED);  m\_wndRadioButtonGreen.Create (\_T ("Green"), WS\_CHILD ¦ WS\_VISIBLE ¦  BS\_AUTORADIOBUTTON, CRect (m\_cxChar \* 66, (m\_cyChar \* 9) / 2,  m\_cxChar \* 78, (m\_cyChar \* 11) / 2), this, IDC\_GREEN);  m\_wndRadioButtonBlue.Create (\_T ("Blue"), WS\_CHILD ¦ WS\_VISIBLE ¦  BS\_AUTORADIOBUTTON, CRect (m\_cxChar \* 66, m\_cyChar \* 6,  m\_cxChar \* 78, m\_cyChar \* 7), this, IDC\_BLUE);  m\_wndRadioButtonRed.SetCheck (1);  m\_wndText.SetTextColor (RGB (255, 0, 0));  m\_wndGroupBox1.SetFont (&m\_font, FALSE);  m\_wndGroupBox2.SetFont (&m\_font, FALSE);  m\_wndRadioButtonRed.SetFont (&m\_font, FALSE);  m\_wndRadioButtonGreen.SetFont (&m\_font, FALSE);  m\_wndRadioButtonBlue.SetFont (&m\_font, FALSE);  return 0;  }  void CMainWindow::OnRedButtonClicked ()  {  m\_wndText.SetTextColor (RGB (255, 0, 0));  }  void CMainWindow::OnGreenButtonClicked ()  {  m\_wndText.SetTextColor (RGB (0, 255, 0));  }  void CMainWindow::OnBlueButtonClicked ()  {  m\_wndText.SetTextColor (RGB (0, 0, 255));  }  /////////////////////////////////////////////////////////////////////////  // CColorStatic message map and member functions  BEGIN\_MESSAGE\_MAP (CColorStatic, CStatic)  ON\_WM\_CTLCOLOR\_REFLECT ()  END\_MESSAGE\_MAP ()  CColorStatic::CColorStatic ()  {  m\_clrText = RGB (0, 0, 0);  m\_clrBack = ::GetSysColor (COLOR\_3DFACE);  m\_brBkgnd.CreateSolidBrush (m\_clrBack);  }  void CColorStatic::SetTextColor (COLORREF clrText)  {  m\_clrText = clrText;  Invalidate ();  }  void CColorStatic::SetBkColor (COLORREF clrBack)  {  m\_clrBack = clrBack;  m\_brBkgnd.DeleteObject ();  m\_brBkgnd.CreateSolidBrush (clrBack);  Invalidate ();  }  HBRUSH CColorStatic::CtlColor (CDC\* pDC, UINT nCtlColor)  {  pDC->SetTextColor (m\_clrText);  pDC->SetBkColor (m\_clrBack);  return (HBRUSH) m\_brBkgnd;  } |

Different controls respond to actions performed by *OnCtlColor* and *CtlColor* handlers in different ways. You've seen how static controls respond to *CDC::SetTextColor* and *CDC::SetBkColor* . For a scroll bar control, *SetTextColor* and *SetBkColor* do nothing, but the brush handle returned by a WM\_CTLCOLORSCROLLBAR message handler sets the color of the scroll bar's shaft. For a list box, *SetTextColor* and *SetBkColor* affect unhighlighted list box items but have no effect on highlighted items, and the brush handle controls the color of the list box's background—anything on an empty or unhighlighted line that isn't painted over with text. For a push button, *OnCtlColor* and *CtlColor* have no effect whatsoever because Windows uses system colors to draw push button controls. If *nCtlType* contains the code CTLCOLOR\_BTN, you might as well pass it on to the base class because nothing you do to the device context will affect how the control is drawn.

## Message Reflection

ON\_WM\_CTLCOLOR\_REFLECT is one of several message-map macros introduced in MFC 4.0 that permit notification messages to be reflected back to the controls that sent them. Message reflection is a powerful tool for building reusable control classes because it empowers derived control classes to implement their own behavior independent of their parents. Previous versions of MFC reflected certain messages back to the controls that sent them using a virtual *CWnd* function named *OnChildNotify*. Modern versions of MFC make the concept of message reflection generic so that a derived control class can map *any* message sent to its parent to a class member function. You saw an example of message reflection at work in the previous section when we derived a new class from *CStatic* and allowed it to handle its own WM\_CTLCOLOR messages.

The following table contains a list of the message reflection macros MFC provides and short descriptions of what they do.

**MFC Message Reflection Macros**

|  |  |
| --- | --- |
| ***Macro*** | ***Description*** |
| ON\_CONTROL\_REFLECT | Reflects notifications relayed through WM\_COMMAND messages |
| ON\_NOTIFY\_REFLECT | Reflects notifications relayed through WM\_NOTIFY messages |
| ON\_UPDATE\_COMMAND\_UI\_REFLECT | Reflects update notifications to toolbars, status bars, and other user interface objects |
| ON\_WM\_CTLCOLOR\_REFLECT | Reflects WM\_CTLCOLOR messages |
| ON\_WM\_DRAWITEM\_REFLECT | Reflects WM\_DRAWITEM messages sent by owner-draw controls |
| ON\_WM\_MEASUREITEM\_REFLECT | Reflects WM\_MEASUREITEM messages sent by owner-draw controls |
| ON\_WM\_COMPAREITEM\_REFLECT | Reflects WM\_COMPAREITEM messages sent by owner-draw controls |
| ON\_WM\_DELETEITEM\_REFLECT | Reflects WM\_DELETEITEM messages sent by owner-draw controls |
| ON\_WM\_CHARTOITEM\_REFLECT | Reflects WM\_CHARTOITEM messages sent by list boxes |
| ON\_WM\_VKEYTOITEM\_REFLECT | Reflects WM\_VKEYTOITEM messages sent by list boxes |
| ON\_WM\_HSCROLL\_REFLECT | Reflects WM\_HSCROLL messages sent by scroll bars |
| ON\_WM\_VSCROLL\_REFLECT | Reflects WM\_VSCROLL messages sent by scroll bars |
| ON\_WM\_PARENTNOTIFY\_REFLECT | Reflects WM\_PARENTNOTIFY messages |

Suppose you want to write a list box class that responds to its own LBN\_DBLCLK notifications by displaying a message box containing the text of the item that was double-clicked. In an SDK-style application, the list box's parent would have to process the notification message and pop up the message box. In an MFC application, the list box can handle the notification and display the message box itself. Here's a derived list box class that does just that:

|  |
| --- |
| class CMyListBox : public CListBox  {  protected:  afx\_msg void OnDoubleClick ();  DECLARE\_MESSAGE\_MAP ()  };  BEGIN\_MESSAGE\_MAP (CMyListBox, CListBox)  ON\_CONTROL\_REFLECT (LBN\_DBLCLK, OnDoubleClick)  END\_MESSAGE\_MAP ()  void CMyListBox::OnDoubleClick ()  {  CString string;  int nIndex = GetCurSel ();  GetText (nIndex, string);  MessageBox (string);  } |

The ON\_CONTROL\_REFLECT entry in the derived class's message map tells MFC to call *CMyListBox::OnDoubleClick* anytime the list box sends an LBN\_DBLCLK notification to its parent. It's worth noting that the notification is reflected only if the parent doesn't process the notification itself—that is, if the parent's message map doesn't include an ON\_LBN\_DBLCLK entry for this list box. The parent receives precedence, which is consistent with the fact that Windows expects the parent to process any notifications in which it is interested.