Chapter Two
Conventions, C++, and Sample Code

Throughout this book we'll watch two applications evolve as we demonstrate how applications can take advantage of the various OLE 2.0 technologies and what pieces of code are necessary to achieve the apex of in-place activation, which will doubtless be the goal of many readers.

One application will be written from scratch, that is, it will not implement certain features such as file I/O until we can use compound files. This application is suitable to become a container application but before that time it will server to illustrate how we incorporate non-compound document features.

The other sample is a full-featured application (albeit without doing anything terribly useful) that uses traditional Windows API from the start, implementing number of common features of all applications (clipboard exchanges, file I/O). As we take this application through each chapter we'll be replacing use of Window APIs with the use of OLE 2.0 technologies, such as converting existing file I/O into compound files. Beginning in Chapter 4 we'll break a piece of this application into an object DLL and develop it into an in-place object.

Along the way we'll also create a number of useful components (either code fragments or DLLs) that you might find useful in your own implementations.

To C or Not to C, with Apologies to Shakespeare

The sample code provided in this book is mostly in C++ primarily because the concepts and features of OLE 2.0 are best expressed in that language. Authoring a book of this sort presents a few philosophical difficulties such as what language to use, how it will all fit on the companion disks, and how not to alienate a large portion of your audience.

C++ code is smaller and simplifies code reuse reducing the amount of code I have to write and the amount of code you have to read. C programmers will no doubt be a little put off with this, so I've provided critical explanations of basic C++ concepts and notations that should help the C programmer understand the sample code. While writing the code I've tried to remember that it has to be understandable by a typical C programmer, so I've purposely kept myself from going hog-wild on everything C++ can do, such as deep multiple inheritance or long strings of virtual functions. This will no doubt put off a number of you C++ programmers, but believe me, is not as bad as forcing everyone to labor through verbose C.

Another possibly offensive fact is that I've used no class library such as Microsoft's MFC that many of you might be using or for which some might foster a religious zeal. Such libraries, while very convenient, have a strong tendency to render an application utterly foreign to C programmers, who start asking glazed questions like "where is WinMain?" and "where's the window procedure?" The samples in this book have a WinMain from which you can begin to follow the thread of execution.

With the exception of how I express the applications data structures, the sample code is really written in straight C. In fact, these applications were ported from original C versions mostly by changing structures into classes. A C programmer briefed on fundamental C++ knowledge should be capable of taking the classes back to structures mostly via global search and replace instead of a line-by-line rewrite.

The remainder of this section will attempt to explain some fundamental elements of C++ from a C programming perspective, enough for C programmers to understand the code in this book. There are quite a number of C++ books available for further study if you find it necessary to make perfect sense of a few more specific C++ techniques. If you are already comfortable with your C++ knowledge feel free to skip to the "Sample Applications" section.

User-Defined Types: C++ Classes

Many a C application is built on top of a number of data structures, one of which might be a typical user-defined structure of application variables:

```
typedef struct tagAPPVARS
  {
  HINSTANCE   hInst;          //WinMain parameters
  HINSTANCE   hInstPrev;
  LPSTR       pszCmdLine;
  int         nCmdShow;

  HWND        hWnd;           //Main window handle
  } APPVARS;

typedef APPVARS FAR *LPAPPVARS;
```

To manage this structure an application will implement a function to allocate one of these structures, a function to initialize it, and a function to free it.

```
LPAPPVARS AppPAllocate(HINSTANCE, HINSTANCE, LPSTR, int);
BOOL      AppFInit(LPAPPVARS)
LPAPPVARS AppPFree(LPAPPVARS);
```

When another piece of code wishes to obtain one of these structures it calls AppPAllocate to retrieve a pointer. Through that pointer is can initialize the structure with AppFInit (which in this case might create a window and store it in hWnd) or access each field in the structure.

In creating this structure and providing functions that know how to manipulate that structure you have defined a type. C++ formalizes this commonly used technique into a *class* defined with the class keyword:

```
class __far CAppVars
  {
  public:
    HINSTANCE   m_hInst;          //WinMain parameters
    HINSTANCE   m_hInstPrev;
    LPSTR       m_pszCmdLine;
    int         m_nCmdShow;

    HWND        m_hWnd;           //Main window handle

  public:
    AppVars(HINSTANCE, HINSTANCE, LPSTR, int);
    ~AppVars(void);
    BOOL FInit(void);
  };

typedef CAppVars FAR *LPCAppVars;
```

The name after class is whatever name you desire. Though we could have used APPVARS paralleling the C structure, the name CAppVars conforms to a C++ convention of using mixed-case names for classes prefixed with a "C" for class. Another convention in C++ classes, at least around Microsoft, is to name data fields with an m_ prefix to clearly identify the variable as belonging to an object.

When another piece of code wishes to use this class it must instantiate a C++ object of this class. In C terms, CAppVars is a structure–to use the structure you still have to allocate one. In C++ we do not need separate functions to allocate the structure, nor do we use typical memory allocation functions. Instead we use C++'s new operator, which allocates an object of this class and returns a pointer to it:

```
LPCAppVars    pAV;

pAV=new CAppVars(hInst, hPrevInst, pszCmdLine, nCmdShow);
```

Since CAppVars was declared as __far, new allocates far memory and returns a far pointer. If the allocation fails then new returns NULL. But this is not the whole story: after the allocation is complete, and before returning, new calls the class *constructor* function which is that funny-looking entry in the class declaration:

```
public:
  AppVars(HINSTANCE, HINSTANCE, LPSTR, int);
```

To implement a constructor you supply a piece of code where the function name is <class>::<class> (<parameter list>) where :: means "member function of":

```
CAppVars::CAppVars(HINSTANCE hInst, HINSTANCE hPrevInst
   , LPSTR pszCmdLine, int nCmdShow)
   {
   //Initialize members of the object
   m_hInst=hInst;
   m_hPrevInst=hPrevInst;
   m_pszCmdLine=pszCmdLine;
   m_nCmdShow=nCmdShow;
   }
```

The :: notation allows different classes to have member functions with identical names, because the actual name of the function known internally to the compiler is a combination of the class name and the member function name. This allows programmers to remove the extra characters from function names that are used in C to identify the structure on which those functions operate.

The constructor, which always has the same name as the class, can take any list of parameters but, unlike a C function, has no return value since the new operator will return whether or not the allocation succeeded. Since the constructor cannot return a value C++ programmers typically avoid placing code in the constructor that might fail, opting rather for a second function to initialize the object once it has positively been instantiated.

Inside the constructor, as well as any other member function of the class, you can directly access the data members in this object instantiation. The m_ prefix on data members is the common convention to distinguish their names from other variables, especially since they often conflict with parameter names.

Implicitly the all members (both data and functions) are dereferenced off a pointer named this, which provides the member function with a pointer to the object that's being affected. Accessing the a member like m_hInst directly is equivalent to writing this->m_hWnd; the latter is more verbose and so not used often.

The code that called new will have a pointer through which it can access members in the object just as it would access any field in a data structure:

```
UpdateWindow(pAV->m_hWnd);
```

What is special about C++ object pointers is that you can also call the *member functions* defined in the class through that same pointer. In the class declaration above, you'll notice that the functions we had defined separately from a structure are pulled into the class itself. Instead of having to call a functions and passing a structure pointer:

```
//C call to a function that operates on a structure pointer
if (!AppFInit(pAV))
   {
   [Other code here]
   }
```

the caller can dereference a member function through the pointer:

```
//C++ call to an object's member function
if (!pAV->FInit())
   {
   [Other code here]
   }
```

The FInit function is implemented using the same :: notation as the constructor

```
CAppVars::FInit(void)
   {
   //Code to register the window class might go here.

   m_hWnd=CreateWindow(...);   //Create the main app window

   if (NULL!=m_hWnd)
      {
      ShowWindow(m_hWnd, m_nCmdShow);
      UpdateWindow(m_hWnd);
      }

   return (NULL!=m_hWnd);
   }
```

Again, since the constructor cannot indicate failure through a return value, C++ programmers typically supply a second initialization function such as FInit that performs anything prone to failure.

You could, of course, still provide a separate function outside the class that took a pointer to an object and

manipulated it in some way. However, a great advantage to using member functions is that you can only call member functions in a class through a pointer to an object of that class. This prevents all sorts of problems when you accidentally pass the wrong pointer to the wrong function usually resulting in some very wrong events.

Finally, where you are done with this object you want to perform cleanup on the object and free the memory it occupies. Instead of calling a specific function for this purpose, you use C++'s delete operator:

```
delete pAV;
```

delete generates a call to the object's *destructor* which is that even funnier looking function in the class declaration but one with an implementation like any other member function:

```
//In the class
public:
   ~AppVars(void);

...

//Destructor implementation
CAppVars::~CAppVars(void)
   {
   //Perform any cleanup on the object.
   if (IsWindow(m_hWnd))
      DestroyWindow(m_hWnd);

   return;
   }
```

The destructor has no parameters and has no return type. This is a great location to perform final cleanup on any allocations made in the course of this object's lifetime. Once this function returns the delete operator then frees the memory allocated for the object and returns to the original caller.

Of course, there are many other ways to define classes and to use constructors, destructors, and member functions that I've shown here. However, this reflects how I've implemented all the sample code in this book.

Access Rights

You probably noticed those public: labels in the class definition are should, by now, be wondering what they're for. Two other variations on public: may appear anywhere in the class definition, as public: can: protected: and private:.

When a data member or member function is declared under a public: label, any other piece of code that has a pointer to an object of this class may directly access those members through dereferencing:

```
LPCAppVars   pAV;
HINSTANCE    hInst2;

pAV=new CAppVars(hInst, hPrevInst, pszCmdLine, nCmdShow);

hInst2=pAV->m_hInst;  //Public data member access

if (!pAV->FInit)     //Public member function access
   {
   [etc.]
   }
```

When data members are marked as public: another piece of code is allowed to change that data without the object knowing:

```
pAV->m_hInst=NULL;    //Generally NOT a good idea
```

This is a nasty thing to do to some poor object that assumes that m_hInst never changes. In order to prevent such arbitrary access to its data members, you would mark such data members as private: in the class:

```
class __far CAppVars
   {
   private:
      HINSTANCE   m_hInst;          //WinMain parameters
      HINSTANCE   m_hInstPrev;
      LPSTR       m_pszCmdLine;
      int         m_nCmdShow;

      HWND        m_hWnd;           //Main window handle

   public:
```

```
    AppVars(HINSTANCE, HINSTANCE, LPSTR, int);
    ~AppVars(void);
    BOOL FInit(void);
};
```

Now code like pAV->hInst=NULL will fail with a *compiler* error since the user of the object does not have access to private members of the object. If you want to allow read-only access to a data member, provide a public member function to return that data. If you want to allow write access but would like to validate the data before storing it in the object, provide a public member function to change a data member.

Both data members and member functions can be private. Private member functions can only be called from within the implementation of any other member function. Note also that the constructor and destructor always have to be public.

If a class wants to provide full access to its private members it can declare another class or a specific function as a *friend*. Any *friend* code has as much right to access the object as the object's implementation itself. If, for example, a window procedure for a window created inside an object's member function is a good case for a friend:

```
class __far CAppVars
  {
  friend LRESULT __export FAR PASCAL AppWndProc([WndProc Parameters]);

  private:
    [Private members accessible in AppWndProc]

  ...
  };
```

Any member declared after a protected: label are the same as private: as far as the object implementation or the object's user is concerned. The difference between private: and protected: manifests itself in derived classes which us to the subject of inheritance.

Single Inheritance

A key feature of the C++ language is code reusability through a mechanism called *inheritance*, that is, one class can inherit the members and implementation of those members from another class. The inheriting class is called a *derived class*; the class from which the derived class inherits is called a *base class*.

Inheritance is a technique to concentrate code common to a number of other classes into one base class, that is, placing the code in a place where other classes can reuse it. Applications for Windows written in C++ typically have some sort of base class to manage a window, as the CWindow class in the sample code:

```
class __far CWindow
  {
  protected:
    HINSTANCE   m_hInst;
    HWND        m_hWnd;

  public:
    CWindow(HINSTANCE);
    ~CWindow(void);

    HWND Window(void);
  };
```

The member function ::Window (note the notation) simply returns m_hWnd allowing read-only access to that member.

If you wanted to now make a more specific type of window, like a frame window, we can inherit the members and the implementation from CWindow by specifying CWindow in the class definition using a colon to separate the derived class from the base class:

```
class __far CFrame : public CWindow
  {
  //CWindow's protected members are now private in CFrame
```

```
protected:
   //We can now add more members specific to our class.
   HMENU    hMenu;

public:
   CFrame(HINSTANCE);
   ~CFrame(void);

//We also get CWindow's ::Window function.
};
```

Sometimes, for explanatory purposes only, the __far and public keywords might be eliminated from the above class declaration leaving only class CFrame : CWindow. This is a matter of convenience.

The implementation of CFrame can access any member marked protected: in its base class CWindow; however, CFrame has no access to private: members of CWindow. Note also that inheriting protected: members converts them to private: unless you explicitly redeclare them under a protected: label.

You will also see a strange notation in constructor functions:

```
CFrame::CFrame(HINSTANCE hInst) : CWindow(hInst)
```

This notation means that the hInst parameter to the CFrame constructor is passed to the constructor of the CWindow base class first before we start executing the CFrame constructor.

Code that has a pointer to a CFrame object can call ::Window through that pointer. The code that executes will be the implementation of CWindow. The implementation of CFrame can, if it desires, redeclare ::Window in its class and provide a separate implementation that might perform other operations:

```
class __far CFrame : public CWindow
   {
   ...
   HWND Window(void);
   };

CFrame::Window(void)
   {
   //Other code here...

   return m_hWnd;    //Member inherited from CWindow
   }
```

If a function in a derived class wants to call the implementation in the base class it explicitly uses the base class's name in the function call. For example, we could write an equivalent CFrame::Window as:

```
CFrame::Window(void)
   {
   return CWindow::Window();
   }
```

In programming one often finds it convenient to typecast pointers of various types to a single type that contains the common elements. In C++, a CFrame pointer can be legally typecast into a CWindow pointer, since CFrame looks like a CWindow. However, calling a member function through that pointer may not do what you expect:

```
CWindow * pWindow;
HWND     hWnd;

pWindow=(CWindow *)new CFrame();    //Legal conversion
hWnd=pWindow->Window();
```

Whose ::Window gets called? Because you are calling through a pointer of type CWindow *, you call CWindow::Window and not CFrame::Window.

Programmers would like to be able to write a piece of code that only knows about the CWindow class but that is capable of calling the ::Window member functions of any derived class. That is, a call to pWindow->Window would call CFrame::Window if, in fact, pWindow is physically a pointer to a CFrame. To accomplish this requires what is known as a *virtual function*.

Virtual Functions and Abstract Base Classes

To solve the typecast problem shown in the previous section we have to redefine the CWindow class to make ::Window a virtual function using the keyword virtual:

```
class __far CWindow
  {
  ....
    virtual HWND Window(void);
  };
```

If CFrame wants to override the ::Window it then declares the same function in its own class and provides an implementation of ::Window:

```
class __far CFrame : public CWindow
  {
  ...
  virtual HWND Window(void);
  };

CFrame::Window(void)
  {
  [Code that overrides the default behavior of a CWindow]
  }
```

Such an override might be useful in a class that hides the fact that it actually contains two windows; the implementation of ::Window would then perhaps return one or the other window handle depending on some condition. With ::Window declared as virtual, the same piece of code we saw before has a different behavior:

```
pWindow=(CWindow *)new CFrame();   //Legal conversion
hWnd=pWindow->Window();
```

The compiler, knowing that ::Window is virtual, is now responsible for figuring out what type pWindow really points to, although the program itself thinks it a pointer to a CWindow. In this code, pWindow->Window calls CFrame::Window. If pWindow really points to a CWindow, then the same code would call CWindow::Window instead.

C++ compilers implement this mechanism through a function table (sometimes referred to as a Vtbl) that lives with each object. The function table of a CWindow will contain one pointer to CWindow::Window. If CFrame overrides the virtual functions in CWindow then its table will contain a pointer to CFrame::Window. If CFrame does *not* override the ::Window function, its table contains a pointer to CWindow::Window.

A pointer to any object in certain implementations of C++[1] is really a pointer to the objects function table first, then to the other data in the class. Whenever the compiler needs to call a member function through an object pointer it looks in the table to find the appropriate address as shown in Figure 2-1. So if the virtual ::Window of the CWindow class and all derived classes always occupies the first position in the table, the calls like pWindow->Window actually calls whatever address is in that position.

Figure 2-1: C++ compilers call virtual functions of an object through a function table.

Virtual functions can also be declared as *pure virtual* by appending "=0" at the end of the function in the class declaration:

```
class __far CWindow
  {
  ....
    virtual HWND Window(void)=0;
  };
```

Pure virtual means "no implementation defined" which renders CWindow into an *abstract base class* which means that you cannot instantiate a CWindow by itself. In other words, pure virtual functions do not create entries in an object's function table and so C++ cannot create an object through which someone might try to make that call. As long as a class has at least one pure virtual member function it is an abstract base class and cannot be instantiated, a fact compilers will kindly mention.

An abstract base class tells derived classes "You *must* override my pure virtual functions!" A normal base class with normal virtual functions tells derived classes "You *can* override these, *if you really care*."

1At least Microsoft C/C++ 7.0, Visual C++ 1.0, and Borland C++ 3.1.

You may have noticed by now that an OLE 2.0 interface is exactly a C++ function table, and this is intentional. OLE 2.0's interfaces are defined as abstract base classes and so an object that inherits from an interface must override every interface member function, that is, when implementing an object in C++, you must create a function table for each interface, and since interfaces themselves cannot create a table, you must provide the implementations that will. OLE 2.0, however, does not require that you use C++ to generate the function table; while C++ compilers naturally create function tables you can just as easily write explicit C code to do the same.

Multiple Inheritance

The inheritance section above described single inheritance, that is, inheritance from a single base class. C++ allows a derived class to inherit from multiple base classes and thus inherit implementations and members from multiple sources. The samples in this book only use multiple inheritance to inherit from one base class and one abstract base class, simply to add a few more virtual functions to the function table of the derived class. Multiple inheritance shows in the class declaration:

```
class __far CBase
   {
   public:
      virtual FunctionA(void);
      virtual FunctionB(void);
      virtual FunctionC(void);
   };

class __far CAbstractBase
   {
   public:
      virtual FunctionD(void)=0;
      virtual FunctionE(void)=0;
      virtual FunctionF(void)=0;
   };

//Note the comma delineating multiple base classes.
class __far CDerived : public CBase, public CAbstractBase
   {
   public:
      virtual FunctionA(void);
      virtual FunctionB(void);
      virtual FunctionC(void);
      virtual FunctionD(void);
      virtual FunctionE(void);
      virtual FunctionF(void);
   };
```

An object of a class using multiple inheritance actually lives with multiple function tables as shown in Figure 2-2. A pointer to an object of the derived class points to a table that contains all the members functions of all the base classes. If this pointer is typecast to a pointer to one of the derived classes, the pointer actually used will refer to a table for that specific base class. In all cases the compiler ignorantly calls the function in whatever table the pointer referenced.

Figure 2-2: Objects of classes using multiple inheritance contain multiple tables.

Of course, there are limitations to using multiple inheritance, primarily when the base classes have member functions with the same names. In such cases the object can have only one implementation of a given member which is shared between all function tables, just like each function in Figure 2-2 is shared between the base class table and the derived class table.

Sample Applications

This book follows the development of two applications that you'll find on the companion disk in the CHAP02 directory: Schmoo and Patron.[1] Both these applications will compile into single-document or multiple-document versions. They share a common code base uncreatively called CLASSLIB that is described in section 2.3 with a code listing in Appendix A. In addition, they make use of several custom control DLLs

1Readers familiar with my work on OLE 1.0 will recognize the applications. Schmoo is a complete rewrite of the 1.0 version and Patron is simply a new application but will eventually be compatible with Patron 1.0 files.

described in section 2.4 with source code listings in Appendices B, C, and D.

Generally I will not be showing the code for the complete sample because the code will become fairly large and the modifications made in each chapter may only affect a few lines in a file. I will not include makefiles (except for one shown below to show how they all generally look), .DEF files, .ICOs or .BMPs (except where absolutely necessary), .RC files, and even some .H files. Since we'll usually be discussing changes made to .CPP (C++) files, I will show the portions of those files relevant to the discussion.

Keep in mind as you examine the code that I designed it such that code changes or additions made to accommodate OLE 2.0 occur in one place. This is the same idea as centralizing drawing code in a window in its WM_PAINT message handling such that any other code that wishes to draw something changes the state of the data and causes a repaint. Such a design is belief, not Truth, so feel free to do things how you see fit. Otherwise Dr. Tyree's Philosophy class is right down the hall.

Schmoo: A Graphical Editor, with Apologies to Al Capp

Schmoo is an application with a silly name that does nothing important, more or less like a silly bulbous armless biped who graced the funny pages for so many years. In the present tense, however, Schmoo is a typical application that creates some kind of graphic data, in this case an image called a Polyline. The polyline is simply any number of points between 0 and 20 connected by lines as shown in Figure 2-3. Schmoo stores data for a single polyline figure in .MOO files.

Figure 2-3: Multiple-document version of Schmoo with several open Polylines.

The user is able to add up to 20 lines by clicking in the polyline region–Schmoo adds the points to an array of 20 POINT structures and increments a point count. The user can reverse added points using Undo, which simply decrements the number of points drawn and repaints. The user can also change both the line and background colors as well as change the line style, but these operations are not reversible. All commands are available from either menus or the toolbar. Schmoo also sports a simple status line on its window.

Schmoo, whose code lives in CHAP02\SCHMOO[1], is a complete application: it performs traditional Windows file I/O, uses the Windows API to support the clipboard, and handles conversion to and from the file format it used in a previous version (Schmoo 1.0 that was OLE 1.0 enabled). It certainly lacks a few features that keep it from possibly being something I could sell, such as printing, Help, and maybe some real Undo functionality. It does, however, maintain those elements you would typically find in most applications of a higher caliber.

Schmoo will follow a course of evolution that takes it from a standard application for Windows and turns it into an OLE 2.0 application. Starting with Chapter 4, we'll fork off a version of Schmoo that breaks its Polyline editing window into an object DLL. This branch will explore creating objects in DLLs while the other, more conservative branch, will maintain Schmoo as a single .EXE. Both cases are important to illustrate and follow paths detailed in Table 2-1.

DLL Object Path

Chapter	Features
4	Polyline control of Schmoo split into an object DLL with the Schmoo EXE modified to instantiate the object using Component Object Model APIs.
5	Schmoo and Polyline are converted to use compound files.
6	Polyline uses a data object to describe data transfers.
7	Schmoo converts clipboard transfers to using data objects.
8	Schmoo adds drag-drop functionality.
10	Polyline is made into a compound document object with editing capabilities and the ability to run as a stand-alone embedded object server. Schmoo continues to use it through some custom interfaces.
16	Polyline becomes in-place capable.

Application Object Path

Chapter	Features

1The code is a tad lengthy to include here.

5	Schmoo is converted to use Compound Files.
7	Schmoo implements a data object and converts clipboard transfers to using data objects.
8	Schmoo adds drag-drop functionality.
10	Schmoo is made into a compound document object application.
11	An object handler is created for Schmoo objects.
13	Schmoo is capable of providing links to its documents.
16	Schmoo becomes in-place capable.

Table 2-1: Evolutionary steps for the Schmoo application into Windows Objects.

Patron: A Page Container, with Apologies to Merriam-Webster

I could have called it Commode, but that would have been tasteless. When I created the first version of this application for OLE 1.0 we called containers "clients" and so a brief intimate encounter with a thesaurus generated Patron. In this case "patron" is defined as either "one who uses the services of another establishment" or "the proprietor of an establishment (such as an inn)."[1] After all, as a container Patron will use the implementations of compound document objects and provide a place (a document) in which they stay. Patron seems a better choice than another butchered version of "container".

Patron's documents are pages that match the size and orientation of whatever printer setup you choos. You can add or delete pages and navigate through them as well as scrolling the view of the current page around in the document window. These commands are available from the menu or a toolbar as shown in Figure 2-4. Like Schmoo, Patron also sports (an identical) status line since we'll eventually make use of it in demonstrating in-place activation.

Figure 2-4: Multiple-document version of Patron with several open documents.

Besides changing the number of pages or navigating through them, the only meaningful commands that Patron supports (in CHAP02\PATRON on the companion disk) are Printer Setup and Print. Printer Setup lets you change size and orientation as you would with any other real application. Print will actually pump out a printed page for every page you've created, complete with page number. In debug builds Patron draws a rectangle diagonal lines on the page so you can see the printable boundaries. How exciting can it get?

Well, features stop there as you might surmise from the code in CHAP02\PATRON. Patron's only reason for existence is to become an OLE 2.0 container application. Patron will be used to demonstrate writing a relatively new application to take advantage of OLE 2.0 technologies. I did not bother implementing any file I/O for Patron because we'll have it use compound files from the start. Nor did I bother making Patron capable of pasting metafiles of bitmaps from the clipboard as that would bring a horrendous amount of code to draw those formats and to somehow serialize them to a file; call me lazy, I call it planning.

Because we programmers instinctively try to avoid as much work as possible, we'll use functionality that OLE 2.0 already provides to add metafile and bitmap capabilities. As we'll see in the chapters ahead, OLE 2.0 already knows how to display and serialize these formats, so we need not consider writing such code ourselves. How convenient! We then add a little more code to enable Patron to contain OLE 2.0 objects and work from there to in-place activation. As we progress through chapters we'll add various features to Patron as shown in Table 2-2.

Chapter	Features
5	Patron adds file I/O using compound files.
7	Patron implements clipboard functions using a data object. Pastes metafiled and bitmaps using OLE 2.0 to for drawing and serializization to a compound file. Patron also implements a data object for clipboard operations.
8	Patron adds drag-drop functionality.
9	Patron is made into a simple compound document container.
13	Patron handles linking.

1Webster's Ninth New Collegiate Dictionary, Merriam-Webster, Inc, 1987

14	Patron adds advanced container features such as linking to its contained objects.
17	Patron becomes in-place capable.

Table 2-2: Evolution of the Patron application by chapter.

Class Libraries

Both Schmoo and Patron are implemented using a very specific C++ class library that you'll find in the CLASSLIB directory. The simplest application that uses this class library needs only a WinMain to instantiate a frame window object, initialize it, and call a member function in that object to spin in a message loop.

The class library handles single-document or multiple-document cases appropriately, provides for the full toolbar and status line, and implements most basic functionality. However, it doesn't provide any code to print, read or write files, or do anything other than create and manage windows. The most skeletal application using this class library is provided in CHAP02\SKEL and shown in Listing 2-1 if you want to examine the code without any extra baggage. This, with Appendix A, illustrates the basis for most other samples.

SKEL.CPP

```
/*
 * SKEL.CPP
 *
 * Skeleton application using CLASSLIB which only needs a WinMain
 * and a number of standard resources.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */
```

Listing 2-1: The SKEL application showing the most basic implementation with CLASSLIB.

```
#include <windows.h>
#include <classlib.h>

/*
 * WinMain
 *
 * Purpose:
 *  Main entry point of application.   Should register the app class is a previous
 *  instance has not done so and do any other one-time initializations.
 */

int PASCAL WinMain (HINSTANCE hInst, HINSTANCE hPrev, LPSTR pszCmdLine, int
nCmdShow)
   {
   LPCFrame       pFR;
   FRAMEINIT      fi;
   WPARAM         wRet;

   //Attempt to allocate and initialize the application
```

```
   pFR=new CFrame(hInst, hPrev, pszCmdLine, nCmdShow);


   fi.idsMin=IDS_STANDARDFRAMEMIN;
   fi.idsMax=IDS_STANDARDFRAMEMAX;
   fi.idsStatMin=IDS_STANDARDSTATMESSAGEMIN;
   fi.idsStatMax=IDS_STANDARDSTATMESSAGEMAX;
   fi.idStatMenuMin=ID_MENUFILE;
   fi.idStatMenuMax=ID_MENUHELP;
   fi.iPosWindowMenu=WINDOW_MENU;
   fi.cMenus=CMENUS;

   //If we can initialize pFR, start chugging messages
   if (pFR->FInit(&fi))
      wRet=pFR->MessageLoop();

   delete pFR;
   return wRet;
   }
```

SKEL.DEF

```
NAME           SKEL
DESCRIPTION    'CLASSLIB Skeleton Chapter 2 (c)1993 Microsoft Corp.'
EXETYPE        WINDOWS
STUB           'WINSTUB.EXE'

CODE           PRELOAD MOVEABLE DISCARDABLE
DATA           PRELOAD MOVEABLE MULTIPLE

HEAPSIZE       4096
STACKSIZE      4096
```

SKEL.RC

```
/*
 * SKEL.RC
 *
 * Basic resources for an application based on CLASSLIB.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 *
```

```
 */

#include <windows.h>
#include <classres.h>


//Use BTTNCUR's IDs for our own.
IDB_STANDARDIMAGES72     BITMAP stdgz72.bmp
IDB_STANDARDIMAGES96     BITMAP stdgz96.bmp
IDB_STANDARDIMAGES120    BITMAP stdgz120.bmp

Icon              ICON   app.ico

#ifdef MDI
IDR_DOCUMENTICON    ICON   document.ico
#endif

IDR_MENU    MENU MOVEABLE DISCARDABLE
   BEGIN
    POPUP "&File"
     BEGIN
      MENUITEM "&New",              IDM_FILENEW
      MENUITEM "&Open...",          IDM_FILEOPEN
      MENUITEM "&Close",            IDM_FILECLOSE
      MENUITEM "&Save",             IDM_FILESAVE
      MENUITEM "Save &As...",       IDM_FILESAVEAS
      MENUITEM SEPARATOR
      MENUITEM "E&xit",             IDM_FILEEXIT
     END

    POPUP "&Edit"
     BEGIN
      MENUITEM "&Undo\tCtrl+Z",     IDM_EDITUNDO
      MENUITEM SEPARATOR
      MENUITEM "&Cut\tCtrl+X",      IDM_EDITCUT
      MENUITEM "C&opy\tCtrl+C",     IDM_EDITCOPY
      MENUITEM "&Paste\tCtrl+V",    IDM_EDITPASTE
     END

   #ifdef MDI
    POPUP   "&Window"
     BEGIN
      MENUITEM "&Cascade",          IDM_WINDOWCASCADE
      MENUITEM "Tile &Horizontally",  IDM_WINDOWTILEHORZ
      MENUITEM "&Tile Vertically",    IDM_WINDOWTILEVERT
      MENUITEM "Arrange &Icons",      IDM_WINDOWICONS
```

```
    END
  #endif

   POPUP "&Help"
    BEGIN
     MENUITEM "&About...",          IDM_HELPABOUT
    END
  END



//Stringtable for Frame windows
STRINGTABLE
  BEGIN
   IDS_CAPTION,          "CLASSLIB Skeleton"
   IDS_UNTITLED,         "(Untitled)"
   IDS_FILEDIRTY,        "%s has been modified.\nDo you wish to save it?"
   IDS_DEFEXT,           "app"
   IDS_DOTEXT,           ".app"
   IDS_FILEOPENFILTER,   "Files (*.app)|*.app|"
   IDS_FILESAVEFILTER,   "Files (*.app)|*.app|"
   IDS_FILEOPEN,         "File Open"
   IDS_FILESAVEAS,       "File Save As"
  END

//Stringtable for document windows.
STRINGTABLE
  BEGIN
   IDS_CLIPBOARDFORMAT,  "Skeleton Vapors"
   IDS_DOCUMENTCAPTION,  "Skeleton Document"
   IDS_VERSIONMISMATCH,  "Loaded data is not a readable version."
   IDS_FILELOADERROR,    "Could not read the requested document."
   IDS_FILESAVEERROR,    "Could not write to the specified document."
   IDS_FILEDOESNOTEXIST, "Document does not exist on the file system."
   IDS_FILEOPENERROR,    "Could not open the requested document."
   IDS_UNKNOWNERROR,     "Action failed due to an unknown error."
  END


IDR_ACCELERATORS ACCELERATORS
 BEGIN
  VK_BACK,     IDM_EDITUNDO, ALT, VIRTKEY
  VK_DELETE,   IDM_EDITCUT, SHIFT, VIRTKEY
  VK_INSERT,   IDM_EDITCOPY, CONTROL, VIRTKEY
  VK_INSERT,   IDM_EDITPASTE, SHIFT, VIRTKEY
  "^Z",        IDM_EDITUNDO
```

```
    "^X",       IDM_EDITCUT
    "^C",       IDM_EDITCOPY
    "^V",       IDM_EDITPASTE
   END


//Tables and such for StatStrip

IDR_STATMESSAGEMAP RCDATA
   BEGIN
    ID_MESSAGEEMPTY,        IDS_EMPTYMESSAGE,
    ID_MESSAGEREADY,        IDS_READYMESSAGE

    ID_MENUSYS,             IDS_MENUMESSAGESYSTEM,
    SC_SIZE,               IDS_SYSMESSAGESIZE,
    SC_MOVE,               IDS_SYSMESSAGEMOVE,
    SC_MINIMIZE,           IDS_SYSMESSAGEMINIMIZE,
    SC_MAXIMIZE,           IDS_SYSMESSAGEMAXIMIZE,
    SC_NEXTWINDOW,         IDS_SYSMESSAGENEXTWINDOW,
    SC_PREVWINDOW,         IDS_SYSMESSAGEPREVWINDOW,
    SC_CLOSE,              IDS_SYSMESSAGECLOSE,
    SC_RESTORE,            IDS_SYSMESSAGERESTORE,
    SC_TASKLIST,           IDS_SYSMESSAGETASKLIST,

    ID_MENUFILE,           IDS_MENUMESSAGEFILE,
    IDM_FILENEW,           IDS_ITEMMESSAGEFILENEW,
    IDM_FILEOPEN,          IDS_ITEMMESSAGEFILEOPEN,
    IDM_FILECLOSE,         IDS_ITEMMESSAGEFILECLOSE,
    IDM_FILESAVE,          IDS_ITEMMESSAGEFILESAVE,
    IDM_FILESAVEAS,        IDS_ITEMMESSAGEFILESAVEAS,
    IDM_FILEEXIT,          IDS_ITEMMESSAGEFILEEXIT,

    ID_MENUEDIT,           IDS_MENUMESSAGEEDIT,
    IDM_EDITUNDO,          IDS_ITEMMESSAGEEDITUNDO,
    IDM_EDITCUT,           IDS_ITEMMESSAGEEDITCUT,
    IDM_EDITCOPY,          IDS_ITEMMESSAGEEDITCOPY,
    IDM_EDITPASTE,         IDS_ITEMMESSAGEEDITPASTE,

    ID_MENUWINDOW,         IDS_MENUMESSAGEWINDOW,
    IDM_WINDOWCASCADE,     IDS_ITEMMESSAGEWINDOWCASCADE,
    IDM_WINDOWTILEHORZ,    IDS_ITEMMESSAGEWINDOWTILEHORZ,
    IDM_WINDOWTILEVERT,    IDS_ITEMMESSAGEWINDOWTILEVERT,
    IDM_WINDOWICONS,       IDS_ITEMMESSAGEWINDOWICONS,

    ID_MENUHELP,           IDS_MENUMESSAGEHELP,
    IDM_HELPABOUT,         IDS_ITEMMESSAGEHELPABOUT,
```

```
   END



STRINGTABLE
   BEGIN
    IDS_EMPTYMESSAGE,              ""
    IDS_READYMESSAGE,              "Ready"

    IDS_MENUMESSAGESYSTEM,          "Commands for working with the application
window"
    IDS_SYSMESSAGESIZE,             "Changes the size of the window"
    IDS_SYSMESSAGEMOVE,               "Moves the window to another position"
    IDS_SYSMESSAGEMINIMIZE,           "Reduces the window to an icon"
    IDS_SYSMESSAGEMAXIMIZE,           "Enlarges the window to its maximum size"
    IDS_SYSMESSAGENEXTWINDOW,         "Switches to the next window"
    IDS_SYSMESSAGEPREVWINDOW,         "Switches to the previous window"
    IDS_SYSMESSAGECLOSE,            "Closes the window"
    IDS_SYSMESSAGERESTORE,            "Restores the window to its previous size"
    IDS_SYSMESSAGETASKLIST,           "Opens the Task List"

    IDS_MENUMESSAGEFILE,            "Commands for working with files"
    IDS_ITEMMESSAGEFILENEW,           "Creates a new, blank document"
    IDS_ITEMMESSAGEFILEOPEN,          "Opens an existing document"
    IDS_ITEMMESSAGEFILECLOSE,         "Closes the currently active document"
    IDS_ITEMMESSAGEFILESAVE,          "Saves the currently active document"
    IDS_ITEMMESSAGEFILESAVEAS,        "Saves the currently active document under a
new name"
    IDS_ITEMMESSAGEFILEEXIT,          "Closes the application"

    IDS_MENUMESSAGEEDIT,            "Commands for manipulating data"
    IDS_ITEMMESSAGEEDITUNDO,          "Reverses previous actions"
    IDS_ITEMMESSAGEEDITCUT,           "Moves data to the clipboard"
    IDS_ITEMMESSAGEEDITCOPY,          "Copies data to the clipboard"
    IDS_ITEMMESSAGEEDITPASTE,         "Pastes data from the clipboard into the active
document"

    IDS_MENUMESSAGEWINDOW,          "Commands for working with document
windows"
    IDS_ITEMMESSAGEWINDOWCASCADE,     "Arranges documents in cascading
style"
    IDS_ITEMMESSAGEWINDOWTILEHORZ,    "Tiles documents favoring width"
    IDS_ITEMMESSAGEWINDOWTILEVERT,    "Tiles documents favoring height"
    IDS_ITEMMESSAGEWINDOWICONS,       "Arranges document icons"

    IDS_MENUMESSAGEHELP,            "Commands for accessing Help"
```

```
    IDS_ITEMMESSAGEHELPABOUT,          "Provides author information"
   END

rcinclude about.dlg
```

ABOUT.DLG

```
DLGINCLUDE RCDATA DISCARDABLE
BEGIN
   "CLASSRES.H\0"
END

IDD_ABOUT DIALOG 11, 22, 149, 98
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "About..."
FONT 8, "Helv"
BEGIN
   ICON          "Icon", ID_NULL, 9, 6, 18, 16
   CTEXT         "CLASSLIB Skeleton", ID_NULL, 40, 3, 100, 9, NOT
                 WS_GROUP
   CTEXT         "Copyright (c)1993 Microsoft Corp.", ID_NULL, 35, 12,
                 112, 12, NOT WS_GROUP
   CTEXT         "Kraig Brockschmidt", ID_NULL, 0, 30, 149, 9, NOT
                 WS_GROUP
   CTEXT         "Software Design Engineer", ID_NULL, 0, 40, 149, 8
   CTEXT         "Microsoft Developer Relations", ID_NULL, 0, 50, 149, 10,
                 NOT WS_GROUP
   CTEXT         "Internet:  kraigb@microsoft.com", ID_NULL, 0, 60, 149, 10,
                 NOT WS_GROUP
   DEFPUSHBUTTON   "OK", IDOK, 54, 80, 42, 15
END
```

MAKEFILE

```
#
# MAKEFILE
# CLASSLIB Skeleton
#
# Copyright (c)1993 Microsoft Corporation, All Rights Reserved
#

#Add '#' to the next line for 'noisy' operation
```

```
!CMDSWITCHES +s

#
#Compiler flags
#Use "SET RETAIL=1" from MS-DOS to compile non-debug version.
#
!ifndef RETAIL
CFLAGS  = -c -nologo -Od -AS -Zipe -G2s -W3 -GA -GEes
LINK    = /al:16/ONERROR:NOEXE/li/CO
DEFS    = -DSTRICT -DDEBUG
!else
CFLAGS  = -c -nologo -Oas -AS -Zpe -G2s -W3 -GA -GEes
LINK    = /al:16/ONERROR:NOEXE/li
DEFS    = -DSTRICT
!endif


!ifdef SDI
DOC     = -DSDI
CLASSLIB= classSDI
DIR     = SDI
SRC_DIR = ..
!else
DOC     = -DMDI
CLASSLIB= classMDI
DIR     = MDI
SRC_DIR = ..
!endif


.SUFFIXES: .h .obj .exe .cpp .res .rc

TARGET  = skel

goal:   cd_build $(TARGET).exe cd_src

cd_build:
   cd $(DIR)

cd_src:
   cd ..


clean:
   cd $(DIR)
   del *.obj
```

```
   del *.res
   del *.exe
   cd ..


INCLS    =
LIBS     = libw slibcew commdlg bttncur gizmobar stastrip $(CLASSLIB)

OBJS1    = $(TARGET).obj
OBJS     = $(OBJS1)

RCFILES1 = $(SRC_DIR)\app.ico $(SRC_DIR)\document.ico $(SRC_DIR)\about.dlg
RCFILES2 = $(SRC_DIR)\stdgz72.bmp $(SRC_DIR)\stdgz96.bmp $(SRC_DIR)
\stdgz120.bmp
RCFILES  = $(RCFILES1) $(RCFILES2)

#####

{$(SRC_DIR)}.cpp{}.obj:
   echo ++++++++++
   echo Compiling $*.cpp
   cl $(CFLAGS) $(DEFS) $(DOC) $(SRC_DIR)\$*.cpp


{$(SRC_DIR)}.rc{}.res:
   echo +++++++++
   echo Compiling Resources
   rc -r -i$(SRC_DIR) $(DEFS) $(DOC) -fo$@ $(SRC_DIR)\$*.rc

#This rule builds a linker response file on the fly depending on debug flags
$(TARGET).exe : $(OBJS) $(TARGET).res $(SRC_DIR)\$(TARGET).def
   echo ++++++++++
   echo Linking $@
   echo +                              > $(TARGET).lrf
   echo $(OBJS1)                           >> $(TARGET).lrf

   echo $(TARGET) $(LINK)                    >> $(TARGET).lrf
   echo nul/li                         >> $(TARGET).lrf
   echo $(LIBS) /NOD/NOE                     >> $(TARGET).lrf
   echo $(SRC_DIR)\$(TARGET).def             >> $(TARGET).lrf

   link @$(TARGET).lrf
   del $(TARGET).lrf
   rc -v $(TARGET).res $(TARGET).exe

##### Dependencies #####
```

```
$(TARGET).res : $(SRC_DIR)\$(TARGET).rc     $(INCLS) $(RCFILES)

#Application level things
$(TARGET).obj : $(SRC_DIR)\$(TARGET).cpp    $(INCLS)
```

STDGZ72.BMP, STDGZ96.BMP, STDGZ120.BMP

APP.ICO, DOCUMENT.ICO

With this class library, the code for both Schmoo and Patron in their respective directories deal almost exclusively with their special features and pieces of code over which they need full control. This way we keep the typical windowing code out of our way to show only the application features and how OLE 2.0 affects them. Throughout this book CLASSLIB will remain unaltered–all modifications to accommodate OLE 2.0 will be made only in the respective application's source code.

This class library is not intended to be a basis for your own application, but of course, there's nothing stopping you. If you are interested, the C++ classes implemented in this library are listed in Table 2-3. Be forewarned, however, that this class library will not evolve and has no support as a real product does. I encourage you to use a professional development environment in your own endeavors to produce applications.

Class	Purpose
CStringTable	Loads a range of strings from the application's resources into memory and provides an overloaded [ ] (array lookup) operator to access those strings.
CWindow	Base class for other window-related classes.
CFrame	Creates and manages a frame window that owns a menu, toolbar, and a client window. Compiles differently for MDI and SDI cases.
CClient	Creates and manages a client window identical to an MDI client window for MDI cases. Under SDI, provides a client window that responds the MDI messages to appear the same to the rest of the application.
CDocument	Creates and manages a document window that lives in the client window. The window is either an MDI child or a simple child window depending on the build.

Table 2-3: C++ classes in CLASSLIB and their uses.

CLASSLIB also contains resource files necessary for building a skeletal application which are exactly those used by SKEL. These are not compiled into the library itself by reside here as templates for applications using this library.

In addition to the library in CLASSLIB is another set of source code in INTERFAC. The code contained here provides templates for various parts of OLE 2.0 implementation which generally involves a lot of member functions of objects. It's generally a tedious job to sit down and write a mass of function headers, so these files provide you with templates to copy and perform a few global replaces to generate a new file for your project. Where applicable, some default implementation has been included.

Additional DLLs: GizmoBar, StatStrip, and BttnCur

In order to fully demonstrate in-place activation we need a few of the sexier controls on our applications. The GizmoBar is an implementation of a typical toolbar control that builds on code provided in BttnCur, a DLL that draws up to six states (for example, up, down, disabled) of toolbar buttons from a single bitmap image. The GizmoBar uses BttnCur to draw its buttons but is also capable of containing any other standard Windows control. The GizmoBar is not able to hold arbitrary custom controls, however. ***Preview Note: These DLLs may be updated from the ones in the OLE 2.0 Toolkit by the time this book is published.***

NOTE: The code for BttnCur is version 1.1, an improvement of BttnCur 1.0 that was included with *The*

Windows Interface book from Microsoft Press. Version 1.1 has two major feature enhancements: support different display resolutions and full color control to allow the standard black/white/gray buttons to change with the system button colors.

The StatStrip control provides a rudimentary message bar generally placed at the bottom of frame windows. The StatStrip is capable of managing a number of strings and displaying one of those strings on request. It also provides almost painless tracking of menu selections and displaying the appropriate message for each item. If you are interested in this mechanism, please study the sources in Appendix B.

All three of these DLLs are implemented in straight C, mostly because they were independent projects from this book. They just happen to work out nicely for us here. The StatStrip and GizmoBar are implemented as controls and thus support a number of APIs and equivalent messages as listed in Appendices B and C. To support their incorporation into the sample applications, CLASSLIB contains two wrapper classes CStatStrip and CGizmoBar for the controls.

Build Environment

The sample code depends on those files in the INC and LIB directories on the companions disk to be in your INCLUDE and LIB environment paths, respectively. The files in each directory and their use is listed in Table 2-4.

File	Purpose
INC Directory:	
BOOKGUID.H	Definitions of Globally Unique Identifiers (CLSIDs and IIDs) used in all samples in this book as well as anything else generally useful to all samples.
CLASSLIB.H	Include file for the class library.
CLASSRES.H	Resource constants for applications using the class library.
WIN1632.H	Macros that handle Win16 and Win32 differences.
DEBUG.H	Macros to facilitate simple debug output.
GIZMOBAR.H	Definitions for GIZMOBAR.DLL
STASTRIP.H	Definitions for STASTRIP.DLL
BTTNCUR.H	Definitions for BTTNCUR.DLL.
IENUM.H	Definition of an IEnumRECT interface for Chapter 3.
IPOLY*nn*.H	Definitions for the Polyline DLL object developed in Chapter *nn*.
Preview Note: There will be more here that we can list when the rest of the samples are complete.	
LIB Directory:	
Preview Note: This directory is a good repository for built libraries. which are not sources.	

Table 2-4: Contents of INC and LIB directories.

Finallt, for default MDI debug builds, run NMAKE the appropriate sample in its directory. Each sample has its own directory and its own MAKEFILE. Other build options are controlled through the environment variables shown in Table 2-5.

Variable	Purpose
SET RETAIL=1	Builds non-debug versions using optimizations and eliminating debugging symbols.
SET SDI=1	Builds SDI version instead of MDI version (default).

Table 2-5: Build options controlled through environment variables.