Note: This file is also saved in Rich Text Format as APPA.RTF. We recommend that you use APPA.RTF if you have a word processor that can read Rich Text Format files.

Appendix A

A C++ Briefing

This appendix is intended to be a briefing about C++ for C programmers. It explains the C++ language from a C perspective so that you can understand the code in this book. This appendix does not describe any details about OLE itself but covers the aspects of the C++ language that I use in the book's samples to implement OLE features. When I use the word *object* in this appendix, I mean a C++ object, not an OLE object. I do not claim to be a C++ expert, so please refer to any of the plethora of C++ books available in order to understand this language more fully.

User-Defined Types: C++ Classes

Many a C application is built on top of a number of data structures. One of these might be a typical user-defined structure of application variables such as the following:

```
typedef struct tagAPP
   {
   HINSTANCE   hInst;              //WinMain parameters
   HINSTANCE   hInstPrev;
   LPSTR       pszCmdLine;
   int         nCmdShow;
   HWND        hWnd;               //Main window handle
   } APP;

typedef APP *PAPP;
```

To manage this structure, an application implements a function to allocate one of these structures, a function to initialize it, and a function to free it:

```
PAPP AppPAllocate(HINSTANCE, HINSTANCE, LPSTR, int);
BOOL  AppInit(PAPP);
PAPP AppPFree(PAPP);
```

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

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

```
class CApp
   {
   public:
       HINSTANCE   m_hInst;              //WinMain parameters
       HINSTANCE   m_hInstPrev;
       LPSTR       m_pszCmdLine;
       int         m_nCmdShow;
       HWND        m_hWnd;               //Main window handle
   public:
       CApp(HINSTANCE, HINSTANCE, LPSTR, int);
       ~CApp(void);        BOOL Init(void);
   };

typedef CApp *PCApp;
```

The name after *class* can be whatever name you want.

Although we could have used APP, paralleling the C structure, *CApp* 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 a member of a class.

To use this class, another piece of code must instantiate a C++ object of the class. In C terms, *CApp* is a structure. To use the structure, you still have to allocate it. 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, as follows:

```
PCApp    pApp;
pApp=new CApp(hInst, hInstPrev, pszCmdLine, nCmdShow);
```

In a 32-bit memory model, *new* allocates far memory and returns a far pointer. (In 16-bit Windows, this requires the keyword *__far* before *CApp* in the class declaration with Microsoft compilers or *__huge* for Borland compilers.) If the allocation fails, *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 the funny-looking entry in the following class declaration:

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

To implement a constructor, you supply a piece of code in which the function name is *<class>::<class> (<argument list>)*, where *::* means "member function of," as in the following:

```
CApp::CApp(HINSTANCE hInst, HINSTANCE hInstPrev
   , LPSTR pszCmdLine, int nCmdShow)
   {    //Initialize members of the object.
   m_hInst=hInst;
   m_hInstPrev=hInstPrev;
   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 to the compiler internally 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 arguments. Unlike a C function, however, it has no return value because the *new* operator will return regardless of whether the allocation succeeded. Because the constructor cannot return a value, C++ programmers typically avoid placing code that might fail in the constructor, opting instead for a second function to initialize the object after it has been positively instantiated.

Inside the constructor, as well as inside any other member function of the class, you can directly access the data members in this object instantiation. Again, the *m_* prefix on data members is the common convention used to distinguish their names from other variables, especially because the names of data members often conflict with argument names.

Implicitly, all the 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 a member such as *m_hInst* directly is equivalent to writing *this->m_hInst*; the latter is more verbose, so it is not often used.

The code that calls *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 preceding class declaration, you'll notice that the functions we defined separately from a structure are pulled into the class itself. The caller does not have to call a function and pass a structure pointer, as is illustrated in the following:

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

Instead, the caller can dereference a member function through the following pointer:

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

The *Init* function is implemented with the same *::* notation that the constructor uses:

```
BOOL CApp::Init(void)
   {
   //Code to register window class might go here.

   m_hWnd=CreateWindow(...);   //Create main application window.

   if (NULL!=m_hWnd)
```

```
        {
        ShowWindow(m_hWnd, m_nCmdShow);
        UpdateWindow(m_hWnd);
        }

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

Again, because a constructor cannot indicate failure through a return value, C++ programmers typically supply a second initialization function, such as *Init*, to perform operations that might be 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, one great advantage of using member functions is that you can call member functions in a class only through a pointer to an object of that class. This prevents problems that occur when you accidentally pass the wrong pointer to the wrong function, an act that usually brings about some very wrong events.

Finally, when you are finished with this object, you'll want to clean up the object and free the memory it occupies. Instead of calling a specific function for this purpose, you use C++'s *delete* operator:

```
delete pApp;
```

The *delete* operator frees the memory allocated by *new*, but before doing so it calls the object's *destructor,* which is that even funnier-looking function in the class declaration (with the tilde, ~) but which comes with an implementation like any other member function:

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

.
.
.

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

   return;
   }
```

The destructor has no parameters and no return value because after this function returns, the object is simply gone. Therefore, telling anyone that something in here worked or failed has no point because there is no longer an object to which such information would apply. The destructor is a great place—your only chance, in fact—to perform final cleanup of any allocations made in the course of this object's lifetime.

Of course, you can define classes and use constructors, destructors, and member functions in many other ways than I've shown here. However, this reflects the way I've implemented all the sample code in this book.

Access Rights

You probably noticed those *public* labels in the class definitions and might by now be wondering what they're for. In addition to *public*, two variations of *public* can appear anywhere in the class definition: *protected* and *private*.

When a data member or a member function is declared under a *public* label, any other piece of code with a pointer to an object of this class can directly access those members by means of dereferencing, as follows:

```
PCApp          pApp;
HINSTANCE      hInst2;

pApp=new CApp(hInst, hPrevInst, pszCmdLine, nCmdShow);

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

if (!pApp->Init())      //Public member function access
   {
   [Other code here]
   }
```

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

```
pApp->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. To prevent such arbitrary access to an object's data members, you can mark such data members as *private* in the class, as in the following:

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

       HWND        m_hWnd;               //Main window handle

   public:
       CApp(HINSTANCE, HINSTANCE, LPSTR, int);
       ~CApp(void);
       BOOL Init(void);
   };
```

Now code such as *pApp->hInst=NULL* will fail with a compiler error because 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 be called only from within the implementation of any other member function. In the absence of any label, *private* is used by default.

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 has. For example, a window procedure for a window created inside an object's initializer is a good case for a friend:

```
class CApp
   {
   friend LRESULT APIENTRY AppWndProc([WndProc parameters]);

   private:
       [Private members accessible in AppWndProc]

   .
   .
   .

   };
```

Any member declared after a *protected* label is 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 brings us to the subject of inheritance.

Single Inheritance

A key feature of the C++ language is code reusability through a mechanism called *inheritance*—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 used to concentrate code common to a number of other classes in one base class—that is, to place the code where other classes can reuse it. Applications for Windows written in C++ typically have some sort of base class to manage a window, as in the following *CWindow* class:

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

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

       HWND Window(void);
   };
```

The *CWindow* member function *Window* simply returns *m_hWnd*, allowing read-only access to that member.

If you now want to make a more specific type of window, such as a frame window, you 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, as follows:

```
class CFrame : public CWindow
   {
   //CFrame gets all CWindow’s variables.
   protected:
       //We can now add more members specific to our class.
       HMENU    m_hMenu;

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

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

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*.

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 *CWindow::Window* through that pointer. The code that executes will be the implementation of *CWindow*. The implementation of *CFrame* can, if it wants, redeclare *Window* in its class and provide a separate implementation that might perform other operations, as follows:

```
class 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 follows:

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

In programming, it is often convenient to typecast pointers of various types to a single type that contains the common elements. In C++, you can legally typecast a *CFrame* pointer to a *CWindow* pointer because *CFrame* looks like *CWindow*. However, calling a member function through that pointer might not do what you expect, as in the following:

```
CWindow *pWindow;
HWND     hWnd;

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

Whose *Window* is called? Because it is calling through a pointer of type *CWindow* *, this code calls *CWindow::Window*, not *CFrame::Window*.

Programmers would like to be able to write a piece of code that knows about only the *CWindow* class but that is also capable of calling the *Window* member functions of the derived class. For example, 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 typecasting problem described in the previous section, we have to redefine the *CWindow* class to make *Window* a virtual function using the keyword *virtual*, as follows:

```
class CWindow
    {

    .
    .
    .

    virtual HWND Window(void);
   };
```

The *virtual* keyword does not appear in the implementation of *CWindow::Window*.

If *CFrame* wants to override *CWindow::Window*, it declares the same function in its own class and provides an implementation of *Window*, as shown in the following:

```
class CFrame : public CWindow
   {

   .
   .
   .

   virtual HWND Window(void);
   };

CFrame::Window(void)
   {
   [Code that overrides default behavior of 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 *CWindow::Window* declared as *virtual*, the piece of code we saw earlier has a different behavior:

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

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

C++ compilers implement this mechanism by means of a *virtual function table* (sometimes referred to as a *vtable* or *vtbl*) that lives with each object. The function table of a *CWindow* object will contain one pointer to *CWindow::Window*. If *CFrame* overrides the virtual functions in *CWindow*, its table will contain a pointer to *CFrame::Window*. If, however, *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++ (at least Visual C++ and Borland C++) is really a pointer to a pointer to the object's function table. 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 A-1. So if the virtual *Window* of the *CWindow* class and of all derived classes always occupies the first position in the table, calls such as *pWindow->Window* are actually calls to whatever address is in that position.

Kraig Brockschmidt, ***Inside OLE, 2nd edition.*** **Appendix A, EG3, dC**

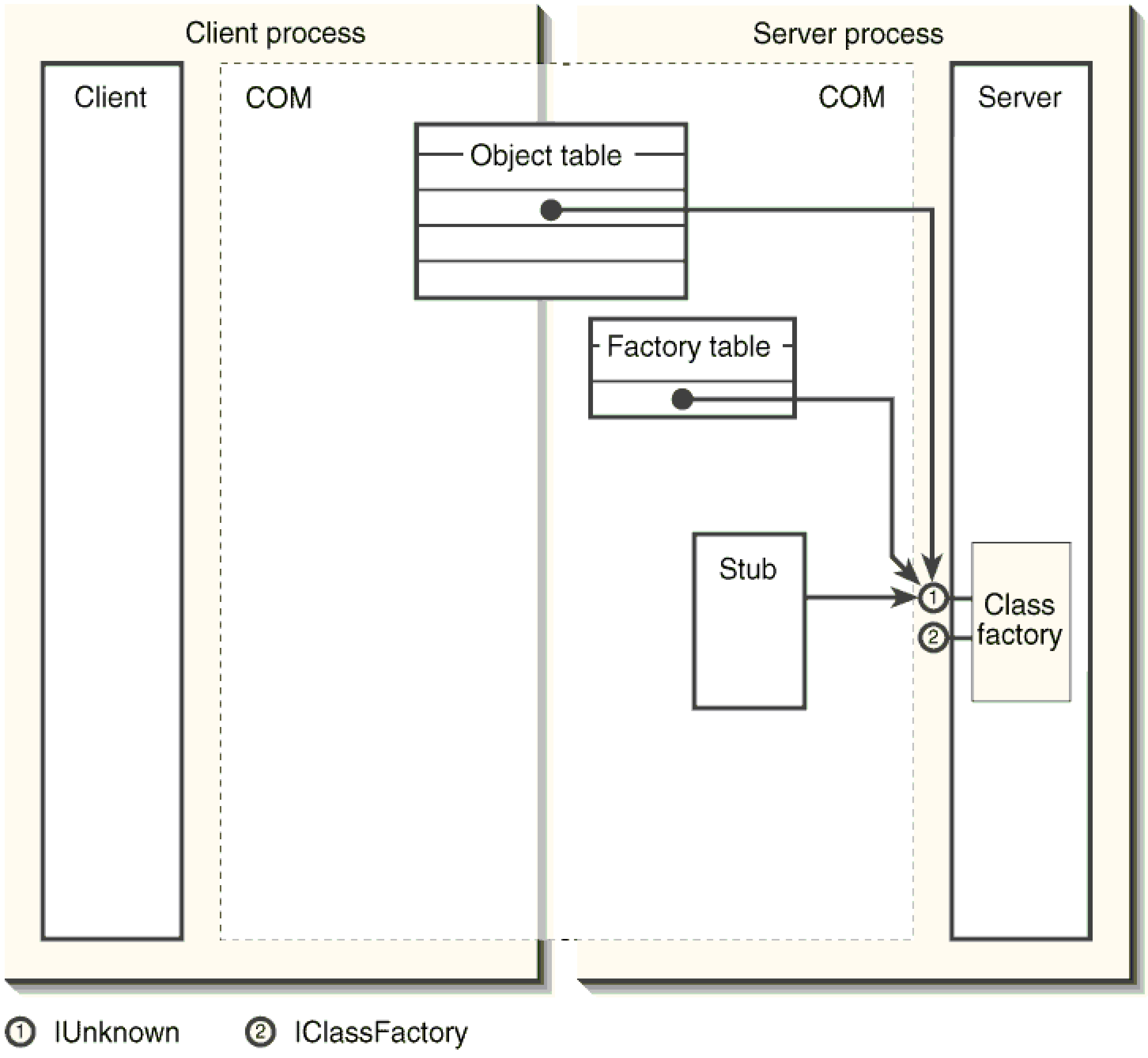

Figure A-1.

C++ compilers call ***virtual funct****ions of an object by means of a function table.*

Virtual functions can also be declared as *pure virtual* by appending *=0* to the function in the class declaration, as follows:

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

Pure virtual means "no implementation defined," which renders *CWindow* into an *abstract base class*—that is, you cannot instantiate a *CWindow* by itself. In other words, pure virtual functions do not create entries in an object's function table, 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 to."

You might have noticed by now that an OLE interface is exactly like a C++

function table, and this is intentional. OLE's interfaces are defined as abstract base classes, 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 because interfaces themselves cannot create a table, you must provide the implementations that will. OLE, however, does not require that you use C++ to generate the function table; although C++ compilers naturally create function tables, you can just as easily write explicit C code to do the same.

Multiple Inheritance

The preceding section described single inheritance—that is, inheritance from a single base class. C++ allows a derived class to inherit from multiple base classes and thus to inherit implementations and members from multiple sources. The samples in this book do not use multiple inheritance, although no technical reasons prevent them from doing so. They use single inheritance only to remain comprehensible to C programmers who are just beginning to understand the concept. In any case, multiple inheritance is evident in the following class declaration:

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

```
        virtual FunctionC(void);
    };

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

//Note comma delineating multiple base classes.
class 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 A-2. A pointer to an object of the derived class points to a table that contains all the member 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 dutifully calls the function in whatever table the pointer referenced.

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 that is shared between all function tables, just as each function in Figure A-2 is shared between the base class table and the derived class table.

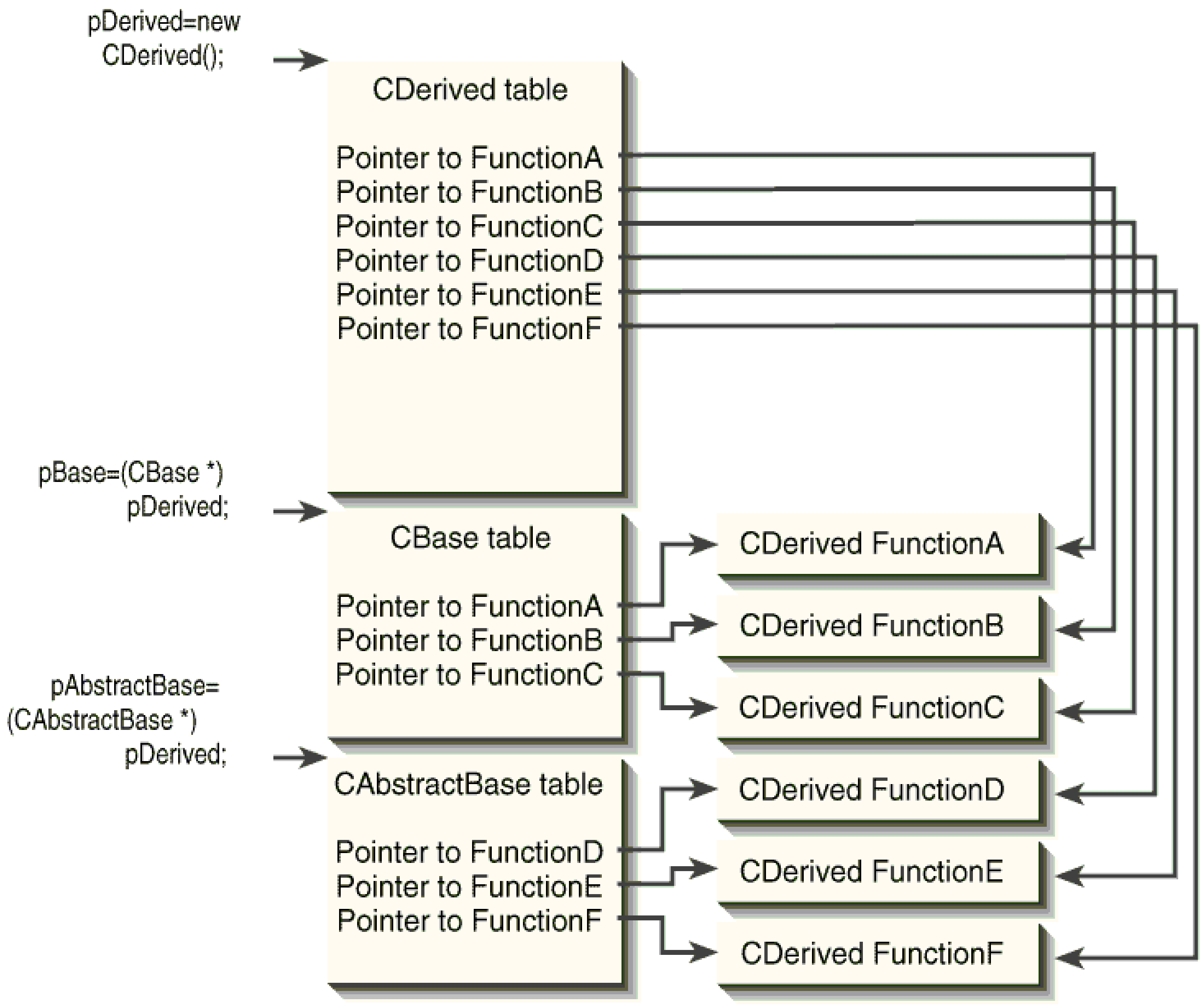
pDerived=new CDerived();
CDerived table
Pointer to FunctionA
Pointer to FunctionB
Pointer to FunctionC
Pointer to FunctionD
Pointer to FunctionE
Pointer to FunctionF
pBase=(CBase *) pDerived;
CBase table
Pointer to FunctionA
Pointer to FunctionB
Pointer to FunctionC
pAbstractBase= (CAbstractBase *) pDerived;
CAbstractBase table
Pointer to FunctionD
Pointer to FunctionE
Pointer to FunctionF
CDerived FunctionA
CDerived FunctionB
CDerived FunctionC
CDerived FunctionD
CDerived FunctionE
CDerived FunctionF

Figure A-2.

Objects of classes using multiple inheritance contain multiple tables.