Chapter Three
Objects and Interfaces

Object *noun* **1 synonym** THING, article; **related** doodad, gadget; **2 synonym** THING, being, entity, individual, material, matter, stuff, substance.[1]

"Objects solve everything," so you might have heard. If an object is a *thing*, then how does one *thing* solve other *things*? The answer is they don't. Things don't solve, people solve. The belief that "object virtues" solve all your programming problems is what some friends of mine classify as "objects on the brain." They suggest you attend meetings of your local OOPaholics Anonymous.

Using object-oriented languages to write applications and operating systems is only a matter of convenience if the ideas you wish to express in that code is best done in such a language. But any C++ programmer will be able to tell you great stories about how C++ solved many problems they encountered in C, but introduced a whole new class of unique problems. For one thing, your language of choice has never simplified design–it has only made the implementation of many designs faster and more robust. I wrote the code in this book in C++ for reasons of reusability and compactness.

So just what is an object? No doubt everyone reading this work will have a different idea about *objects* than myself. Objects are becoming so commonplace in just about every facet of computing that it has become difficult to understand what the word object means in a variety of contexts. Object models appear in places regardless of their relationships to any sort of object-oriented programming model. This chapter will attempt to clarify exactly what we mean by a Windows Object and what we mean by the interfaces that such objects support. The standardized specifications of both are part of OLE 2.0's Component Object Model.

Windows Objects are slightly different than what C++ programmers might be used to, for instance, our objects here do not allow direct access to data. Windows Objects can also be used and implemented in C code, that is, an object-oriented language is not necessary, only more convenient, to express object-oriented ideas. Real hacks can even implement these ideas in assembly–since C is just a nice way to express assembly–but I will certainly not offer advice on such efforts here.

We also have to distinguish between the object implementation and the object user, that this book will refer to as the "user" in programming contexts. Use of "user" here should not be confused with and "end user" who will only see the features you are implementing in your applications and will generally not be aware of our programming constructs.

This chapter will look at objects and interfaces in both C and C++ without having to get deeply into OLE 2.0 itself. The first sets of code we'll see don't even #include any of the OLE 2.0 include files, but still implement what we mean by a Windows Object, that is, something with interfaces. With a solid understanding of these fundamentals, we can move forward into seeing what is required for more complex Windows Objects with more useful capabilities. Note that much of the background presented in the section "IUnknown, the Root of All Evil" and beyond lead directly into Chapter 4.

I want to stress that an object as presented here is not a compound document object, that is, we're not yet talking about specific applications like Containers. Much of the information in this chapter and all those up to Chapter 9 deal with topics completely outside the realm of compound documents. So, as Yoda might suggest, "clear your mind of questions" and be prepared to learn just what we mean by object in the cosmos of OLE 2.0.

Do objects solve everything? No. Does OLE 2.0 and it's object model solve everything. No. OLE 2.0 intends to **simplify the expression** of object-oriented ideas under Windows. It does not intend to somehow make application or system design fall freely like manna from heaven. If it could we would not have to worry about the national debt.

1Webster's Collegiate Thesaurus, Merriam-Webster, Inc., 1976

The Ultimate Question to Life, the Universe, and Objects

I know a Windows Object exists that is capable of specific functions. How do I obtain a pointer to that object?

This question is a central theme to this book: this chapter and those that follow are concerned with specific types of objects, how you get a pointer to one, and what you can do with that object once you have the pointer. Each chapter generally deals with different object types (and how you identify that object), the interfaces they support, different techniques to obtain their pointers (for whatever code uses the object), and the specific functions you can call through those pointers. So the answer to our question, which is not forty-two, varies with each sub-technology in OLE 2.0. Realize as well that a "compound document object" is only one type of Windows Object and that Server applications are not the only object implementors. The fact is that almost all OLE 2.0 applications, regardless of what technologies they use, are both object users and object implementors.

To fully understand obtaining and using a Windows Object we must first go back to a few even more fundamental questions. First, what is an object? To answer that question we must first ask: What is an object class? To some an object class may seem some mighty spiritual force divinely manifested in your include files. In all reality, a class (and objects) can be described in terms that anthropologist Marvin Harris would call "practical and mundane"[1] for in one way or another, a programmer or compiler has to reduce the notion of a class into code.

A *class* in mundane terms is the *definition* a data structure (members) and functions that manipulate that structure (member functions). The concept can be expressed in any programming language; C++, SmallTalk, and other such languages have just formalized the notion. For example, C++ classes generally live in include files, such as the one shown in Chapter 2:

```
/*
 * SOMEFILE.H
 */

class __far CAppVars
   {
   public:
      HINSTANCE   m_hInst;
      HINSTANCE   m_hInstPrev;
      LPSTR       m_pszCmdLine;
      int         m_nCmdShow;
      HWND        m_hWnd;

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

A class is only definition and carries no implementation, although classes in some languages may define default implementations that are not realized until there is some instantiation of the data structure that contains a function table and the variables of the class. We call that instantiated structure an *object*. In C++, objects are manifested in memory as shown in the figure below:

The object has two components in memory: a function table that contains pointers to each member function defined in the object's class and a block containing the current values for each variable (or data member). The user of the object generally has some reference[2] to this chunk of memory which for the purposes of this book is always a pointer. The user obtains this reference using some type of function call (direct or implied) where that function allocates the block in memory, initializes the function table, and returns the reference to that memory to the user.

Once the user has the reference to the block of memory, it may call any of the functions in the object's function table and possibly access the object's variables depending on the language in use. The single most important benefit here is that in order to call any of the functions defined in an object class you must first

1From Cows, Pigs, Wars, and Witches, by Marvin Harris, Vintage Books, 1974
2My use of 'reference' here does *not* necessarily mean a C++ reference.

have some reference to an instantiated object so that some data exists on which those functions operate. Without having a reference to the object, there is simply no way to call one of the object's functions. Even with a pointer, the object can restrict your access to its variables or functions via language mechanisms like *public* and *private* members in C++. In contrast, a non-object-oriented languages like C allows you to call any function with any garbage you desire, and given a pointer to a data structure there is nothing to keep you from partying all over those variables.

The OLE 2.0 notion of class is even more strict than our general definition above because the only accessible members of a class are specific groups of functions called *interfaces*. As mentioned in Chapter 1 and as shown in Figure 3-1, an interface is a group of semantically related functions that are publicly accessible to the user of a Windows Object. An object's interface can really be viewed as just the function table part of an object in memory:

By themselves, interfaces in OLE 2.0 are not classes and thus they cannot be instantiated. In other words they provide a convenient structure to lay over the top of some function table to provide more readable and maintainable names for each function.

An interface, in obscenely mundane terms, is a block of memory containing an array of function pointers, that is, a function table; the interface definition itself just provided names for each pointer in that table. When a user of a Windows Object obtains a pointer to an interface that an object supports, we say it has a pointer to an interface "on" that object. Again, that pointer does not provide access to then entire object, instead it allows access to one interface on that object, that is, one set of functions. Through an indirection on that pointer the user calls a function of the object:

As mentioned in Chapter 1, the user of a Windows Object only has access to one interface through one pointer, even when the object itself actually supports more than one interface, that is, implements more than one set of related functions and provides multiple function tables. Note that when we graphically represent an object with interfaces, we use a circle to represent each interface as introduced in Chapter 1.

To use functions in a different interface, the user must obtain a second pointer to that other interface through the QueryInterface function contained in all interfaces. The section below called "IUnknown, the Root of All Evil" will explore QueryInterface in detail. IUnknown is a fundamental interface that all Windows Objects must support which is why it's always placed above the object in diagrams like that above instead of out to the side like other interfaces.

The function table itself is designed to have a layout identical to that generated by C++ compilers such that an indirection (->) on the pointer allows you to call an interface function. However, this does not force you to use C++ to program OLE 2.0; like I said in Chapter 2, C++ is simply more *convenient*. Any object implementation is only required to provide separate function tables for each supported interface: how you so choose to create each table will, of course, be different depending on your language of choice as the section "A Simple Object in C and C++" illustrates below.

Since neither use nor implementation of a Windows Object is not dependent on the programming language used, you can view OLE 2.0's object model, called the Component Object Model, as a *binary standard* which is a major advantage over other proposed object models. You may choose to implement an object in Visual Basic which is still usable from a C or C++ application as long as you can provide a pointer to your interface function tables. Microsoft has done us all a wonderful service by not limiting our choice of programming tools or languages.

So back to the Ultimate Question: I know there's a Windows Object, how do I obtain a pointer to an interface on that object? The answer greatly depends on how you identify the Windows Object which leads to four basic mechanisms in OLE 2.0 for getting that pointer:

1. Call an API that creates an object of only one type, that is, will only ever return a pointer to one specific interface.
2. Call an API that can create an object based on some class identifier and returns any interface pointer you request.
3. Call a member function of some interface that returns a specific interface pointer on another separate object.

4. Implement interface functions on your own objects to which other agents pass their interface pointers.

All of these mechansism are used both by OLE 2.0 applications and the OLE 2.0 libraries themselves. OLE 2.0 implements most of the APIs you'll use to obtain a pointer through methods 1 and 2, but you might implement your own private APIs to accomplish similar ends. You will use method 3 when you are the user of some object and have occasion to ask that object to create another object. You will use method 4 when you are an object implementor and your user needs to provide you with a pointer to it's own objects. This last method is how two applications, like a compound document container and server, initiate a two-way dialog: both applications implement specific (and different) objects and pass interface pointers to each other.

Windows Objects vs. C++ Objects

Since C++ is the most widely used object-oriented language for programming Windows, some readers will wonder why Windows Objects differ in many respects from C++ objects. This section attempts to explain exactly what those differences are and why they exist. The most pervading reason is that in C++ you may only use C++ objects that live and execute within your own application (EXE), possibly within DLLs (but at a price). On the other hand, you can use Windows Objects regardless of where they live and execute, be it in your own EXE, a DLL (including the operating system itself), or another EXE. In the future, Microsoft will enable Windows Objects to live and execute on another machine, a capability far out of reach of C++ objects.

Let's Go Traveling

Suppose I'm a C++ application that lives in Rugby, North Dakota (the geographic center of North America) and my application is bounded by the continental United States as illustrated in Figure 3-1. I can freely visit any of 48 states, no questions asked, by driving along an interstate. Access is fast and easy although I am subject to the laws of each individual state. I can also drive into Canada or Mexico to buy their goods and use their services, but I do have to stop at the border and answer a few questions; travel is a little slower but still quite easy. In programming terms, I can freely use any object class within the boundaries of my application as long as I obey the access rights of those individual objects. I can also use objects implemented in DLLs but there is more work in getting across the DLL boundary, even to my own DLL like Alaska.

I might live happily for a long time restricting myself to a single continent. But there are six other continents and many other countries on the planet that I might want to visit. Getting there is not easy–I have to transfer flights, go through Customs, and show passports. If I want to travel to a distant destination like Antananarivo, Madagascar, I would have to fly to Chicago, then to London, switch carriers to get to Nairobi, Kenya, then catch a final flight to Antananarivo. On each segment of my journey I will probably fly on a different airline in a different airplane (or I may only be able to travel by boat or train) and walk through customs offices in three different countries. If I step out of line anywhere I may find myself in a prison on the other side of the globe.

As a C++ application I experience the same difficulty using C++ objects implemented in other applications (countries) or code that is otherwise separated by a process boundary (that is, oceans) as

Figure 3-1: Travel within North America is fairly painless.

illustrated in Figure 3-2. The best I can hope for is becoming intimately familiar with the protocols and customs of each application along my way, knowledge that can only apply to those specific applications: when I want to visit use the services of different application I must learn another new interface. If time is not a luxury, I'll probably decide to only ever visit a few other countries.

Figure 3-2: Travel abroad involves much more time, effort, and knowledge.

OLE 2.0 offers you membership in the Windows Objects Club that makes travel or abroad much easier. Windows Objects standardizes the protocol for visiting any other country so you only have to learn one set of rules. Windows Objects offers non-stop flights to many countries (Windows Objects in DLLs) with a maximum of one stop to any other destination on the planet (Windows Objects in EXE applications). When

you are a member of the Windows Object Club, travel is as easy as showing your membership card and hopping on a plane for whatever destination you choose. No matter where you are, the Windows Object Club has a flight departing to every destination as depicted in Figure 3-3.

In programming terms you join the Club by using the various OLE 2.0 APIs to access specific objects without concern for where that object actually lives. Those APIs form the protocol you learn once; later in Chapters 9 and beyond we'll learn more about compound documents which provides you with the benefit of a personal interpreter in any country you visit or with whom you otherwise do business. When we talk about In-Place Activation, we'll see how the Windows Object Club can bring the country to you.

The Windows Object Club today offers easy travel between all countries and continents on our little blue planet. In the future, this club will provide the same benefits to interplanetary and interstellar travel without requiring you to even reapply. That is, Windows Objects will become network aware and allow you to use objects on other machines, either in your local-area or even a wide-area network. Perhaps someday we'll have the PLAN (planetary network) so you can use objects that live on the moon, either figuratively or physically.

Figure 3-3: The Windows Object Club simplifies travel, and someday will open more routes.

The purpose of this little exercise was to show that C++ objects are somewhat limited in scope because access to objects, being defined by the language, restricts you to objects that live in your own process space. Windows Objects, being defined by the system, opens access to any object anywhere on your machine, and eventually on other machines as well.

Other Windows Object and C++ Differences

Because the location of an object's implementation varies so widely between C++ objects and Windows Objects, there are a number of other key implementation differences that affect programming:

- Class definition
- Object instantiation
- Object references
- Object destruction

Class definition: C++ defines a class using the class keyword that generates some user-defined *type*. Members and member functions can be private, protected, and public. Furthermore, C++ classes can inherit from another class, thereby taking on all the characteristics (data and member function) of that base class with the ability to override or expand select pieces of that base class.

A Windows Object is defined in terms of what interfaces it supports. All objects support at least one interface called IUnknown that is discussed below in section 3.5; support of this one interface qualifies the object as a Windows Object. The object user learns about other interfaces the object supports through member functions of IUnknown.

Windows Objects are all therefore at least of type Unknown and can be treated as another type through a different interface. Because of this mechanism, there is no user-defined type associated with a Windows Object class as there is with a C++ class. A Windows Object class is identified only through a class ID (a structure called CLSID) which is stored in the registration database[1] along with information defines where the object lives and some characteristics that a potential user may wish to know without having to actually instantiate the object.

Object Instantiation: C++ objects are instantiated through various means, such as declaring a variable of the object's type on the stack, declaring a global variable, or using the new operator on that type. Regardless of the actual technique used, C++ eventually uses the new operator which in turn calls the object's constructor.

A Windows Object is generally instantiated through one of a number of different APIs depending on the exact type of object you wish to use–in some cases you are given an object pointer instead of instantiating it directly. In any case, one technique described in Chapter 4 is to use a thing called a *class object* to instantiate a Windows Object mush like the new operator works for C++ objects. A class object represents a specific class ID, is obtained by a specific OLE 2.0 API, and supports an interface called IClassFactory. The IClassFactory interface contains a member function called ::CreateInstance to which you pass the class ID of the object you want. IClassFactory::CreateInstance is the logical equivalent of new.

Object References: C++ objects can be referenced through an object variable, an object reference (a special type in C++), or a pointer to the object. Since objects are always local, their instantiations can live anywhere in your process space. Through any variable, the user has access to any public members of the object, or private and protected members if the user and object are friends.

As I hope I have been beating into your head by now, a Windows Object is *always* referenced through a pointer, not to the object itself but to an *interface*. This means that through a given interface pointer to the object the user can only access member functions in that interface. The user can never have a pointer to the whole object (since there is no definition of what the whole object contains) so there no access to data members and no concept of friend.

Through the interface known as IUnknown a user can get at other interfaces that the object also support, but that means obtaining a different pointer that refers to the same object. Each pointer to an interface points to a -function table in the object, and each table only contains members functions for a specific interface as shown in Figure 3-4. Since every interface defined in OLE 2.0 derives from IUnknown, it is not necessary to have an IUnknown pointer to query for other interfaces–you can use any other interface pointer as if it were IUnknown.

Figure 3-4: Multiple interface pointers to an object reference unique function tables in the object.

Once we have a pointer to an object's interface, we call the interface's member functions just as we would

1The registration database is stored in REG.DAT in your Windows directory. The Reg* APIs in SHELL.DLL allow you to programatically add or remove entries from this database. REGEDIT.EXE is an end-user tool allowing you to do the same manually.

call a member function of a C++ object through a pointer:

```
pObject->MemberFunction([parameters]);
```

Since a pointer to a Windows Object always points to a function table, such a pointer can also be used from C or assembly code not just C++, as described below in "A Simple Object in C and C++."

Object Destruction: In C++ one destroys an object using the delete operator on an object pointer. If the object was declared as a stack variable, then C++ automatically calls delete when the variable goes out of scope. If the user explicitly called new to obtain a pointer it must also explicitly call delete on that pointer.

The logical equivalent of delete on a Windows Object is a member function called ::Release that is part of IUnknown and therefore part of any other interface to which to have a pointer. Calling ::Release decrements a reference count that the object maintains and only when that count is to zero does the object actually free itself. Reference counting is a deeper topic discussed later in section 3.4.

A Simple Object in C and C++: RECTEnumerator

Windows Objects can be written in either C or C++. C++ is the natural and more convenient language to use as objects are more easily expressed in C++. However, with a little more overhead you can implement Windows Objects in C. The differences lie in how you create the function table for the object's interfaces and how to call the functions in those function tables. To illustrate the differences between the two languages let's implement type of object called an enumerator.

Enumerators are specific objects defined in OLE 2.0 that are used to communicate lists of information between another object and the user of that object. For example, you might be using an object that represents some source of data, call it a data object, and so you might ask that object what data formats it supports (which we'll see in Chapter 6). The data object would create another independent enumerator that allows the user to iterate through the list of formats supported by the data object. The user of the data object also becomes a user of the enumerator, albeit through a different interface pointer.

An enumerator supports one of a set of interfaces prefixed with IEnum. Since the elements of the enumerator's list varies by context, OLE 2.0 defines a number of IEnum<type> interfaces where <type> is the name of the specific data structure used for each element in the list. OLE 2.0 also provides marshaling support for each standard IEnum* interface. Each IEnum interface supports all member function of IUnknown (of course!) as well as four additional members to facilitate iteration over the list of elements:

::Next	Return the next n elements of the list starting at the current index.
::Skip	Skip past a n elements in the list.
::Reset	Set the current index to zero.
::Clone	Return a new enumerator object with the same state.

For this exercise let's define a custom interface called IEnumRECT and an object called RECTEnumerator that implements that interface. The interface is defined in IENUM0.H as shown in Listing 3-1 which can be found in the INC directory of the sample code. This file compiles differently for C and C++ depending on the __cplusplus symbol which is only defined when compiling for C++.[1] The C++ implementation of the RECTEnumerator object is shown in Listing 3-2 with the C implementation in Listing 3-3. Both samples are in the CHAP03 directory. Be sure not to confuse the name RECTEnumerator for this object with the name of anything we might use to implement the object: it's just a label.

IENUM0.H

```
/*
 * IENUM0.H
 *
 * Definition of an IEnumRECT interface as an example of the
 * interface notion introduced in OLE 2.0 with the Component Object
 * Model as well as the idea of enumerators.  This include file defines
 * the interface differently for C or C++.
 *
```

1The major C++ compilers, at least, define te __cplusplus symbol.

```
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */


#ifndef _IENUM0_H_
#define _IENUM0_H_


//C++ Definition of an interface.
#ifdef __cplusplus


//This is the interface:  a struct of pure virtual functions.
struct __far IEnumRECT
    {
    virtual DWORD AddRef(void)=0;
    virtual DWORD Release(void)=0;

    virtual BOOL  Next(DWORD, LPRECT, LPDWORD)=0;
    virtual BOOL  Skip(DWORD)=0;
    virtual void  Reset(void)=0;
    };

typedef IEnumRECT FAR * LPENUMRECT;


#else   //!__cplusplus

/*
 * A C interface is explicitly a structure containing a long
 * pointer to a virtual function table that we have to initialize
 * explicitly.
 */

typedef struct
    {
    struct IEnumRECTVtbl FAR *lpVtbl;
    } IEnumRECT;

typedef IEnumRECT FAR * LPENUMRECT;
```

Listing 3-1: The IENUM0.H include file found in the shared INC directory.

```
//This is just a convenient naming
typedef struct IEnumRECTVtbl IEnumRECTVtbl;
```

```
struct IEnumRECTVtbl
   {
   DWORD (* AddRef)(LPENUMRECT);
   DWORD (* Release)(LPENUMRECT);
   BOOL  (* Next)(LPENUMRECT, DWORD, LPRECT, LPDWORD);
   BOOL  (* Skip)(LPENUMRECT, DWORD);
   void  (* Reset)(LPENUMRECT);
   };

#endif  //!__cplusplus

#endif //_IENUM0_H_
```

ENUM.CPP

```
/*
 * ENUM.CPP
 *
 * Example enumerator interface in C++.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved

 */



#include <windows.h>
#include <malloc.h>
#include "enum.h"



int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hInstPrev
   , LPSTR pszCmdLine, int nCmdShow)
   {
   MSG         msg;
   LPAPPVARS   pAV;

   //Create and initialize the application.
   pAV=new CAppVars(hInst, hInstPrev, nCmdShow);

   if (NULL==pAV)
      return -1;
```

```
   if (pAV->FInit())
      {
      while (GetMessage(&msg, NULL, 0,0 ))
         {
         TranslateMessage(&msg);
         DispatchMessage(&msg);
         }
      }

   delete pAV;
   return msg.wParam;
   }




LRESULT FAR PASCAL __export EnumWndProc(HWND hWnd, UINT iMsg
   , WPARAM wParam, LPARAM lParam)
   {
   LPAPPVARS   pAV;
   RECT        rc;
   DWORD       cRect;

   COMMANDPARAMS(wID, wCode, hWndMsg);

   //This will be valid for all messages except WM_NCCREATE
   pAV=(LPAPPVARS)GetWindowLong(hWnd, ENUMWL_STRUCTURE);
```

Listing 3-2: The ENUM program implemented in C++.

```
   switch (iMsg)
      {
      case WM_NCCREATE:
         //CreateWindow passed pAV to us.
         pAV=(LPAPPVARS)((LONG)((LPCREATESTRUCT)lParam)->lpCreateParams);

         SetWindowLong(hWnd, ENUMWL_STRUCTURE, (LONG)pAV);
         return (DefWindowProc(hWnd, iMsg, wParam, lParam));


      case WM_DESTROY:
         PostQuitMessage(0);
         break;
```

```
    case WM_COMMAND:
        switch (wID)
            {
            case IDM_ENUMCREATE:
                if (NULL!=pAV->m_pIEnumRect)
                    pAV->m_pIEnumRect->Release();

                CreateRECTEnumerator(&pAV->m_pIEnumRect);
                break;


            case IDM_ENUMRELEASE:
                if (NULL==pAV->m_pIEnumRect)
                    break;

                if (0==pAV->m_pIEnumRect->Release())
                    pAV->m_pIEnumRect=NULL;

                break;


            case IDM_ENUMRUNTHROUGH:
                if (NULL==pAV->m_pIEnumRect)
                    break;

                while (pAV->m_pIEnumRect->Next(1, &rc, &cRect))
                    ;

                break;


            case IDM_ENUMEVERYTHIRD:
                if (NULL==pAV->m_pIEnumRect)
                    break;

                while (pAV->m_pIEnumRect->Next(1, &rc, &cRect))
                    {
                    if (!pAV->m_pIEnumRect->Skip(2))
                        break;
                    }

                break;


            case IDM_ENUMRESET:
                if (NULL==pAV->m_pIEnumRect)
```

```
                break;

            pAV->m_pIEnumRect->Reset();
            break;


        case IDM_ENUMEXIT:
            PostMessage(hWnd, WM_CLOSE, 0, 0L);
            break;
        }
      break;

    default:
      return (DefWindowProc(hWnd, iMsg, wParam, lParam));
    }

  return 0L;
  }




CAppVars::CAppVars(HINSTANCE hInst, HINSTANCE hInstPrev, UINT nCmdShow)
  {
  //Initialize WinMain parameter holders.
  m_hInst     =hInst;
  m_hInstPrev =hInstPrev;
  m_nCmdShow  =nCmdShow;

  m_hWnd=NULL;
  m_pIEnumRect=NULL;

  return;
  }


CAppVars::~CAppVars(void)
  {
  //Free the enumerator object if we have one.
  if (NULL!=m_pIEnumRect)
    m_pIEnumRect->Release();

  return;
  }
```

```
BOOL CAppVars::FInit(void)
   {
   WNDCLASS    wc;

   if (!m_hInstPrev)
      {
      wc.style         = CS_HREDRAW | CS_VREDRAW;
      wc.lpfnWndProc    = EnumWndProc;
      wc.cbClsExtra     = 0;
      wc.cbWndExtra     = CBWNDEXTRA;
      wc.hInstance      = m_hInst;
      wc.hIcon          = LoadIcon(m_hInst, "Icon");
      wc.hCursor        = LoadCursor(NULL, IDC_ARROW);
      wc.hbrBackground  = (HBRUSH)(COLOR_WINDOW + 1);
      wc.lpszMenuName   = MAKEINTRESOURCE(IDR_MENU);
      wc.lpszClassName  = "CPPEnum";

      if (!RegisterClass(&wc))
         return FALSE;
      }

   m_hWnd=CreateWindow("CPPEnum", "Enumerator in C++"
      , WS_MINIMIZEBOX | WS_OVERLAPPEDWINDOW
      ,35, 35, 350, 250, NULL, NULL, m_hInst, this);

   if (NULL==m_hWnd)
      return FALSE;

   ShowWindow(m_hWnd, m_nCmdShow);
   UpdateWindow(m_hWnd);

   return TRUE;
   }
```

IENUM.CPP

```
/*
 * IENUM.CPP
 *
 * Implements the CImpIEnumRECT class in C++
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
```

```
 */

#include <windows.h>
#include <malloc.h>
#include "enum.h"


/*
 * CreateRECTEnumerator
 *
 * Purpose:
 *  Given an array of rectangles, creates an enumerator interface
 *  on top of that array.
 *
 * Parameters:
 *  ppEnum          LPENUMRECT FAR * in which to return the interface
 *                  pointer on the created object.
 *
 * Return Value:
 *  BOOL            TRUE if the function is successful, FALSE otherwise.
 */

BOOL CreateRECTEnumerator(LPENUMRECT FAR *ppEnum)
   {
   if (NULL==ppEnum)
      return FALSE;

   //Create the object storing a pointer to the interface
   *ppEnum=(LPENUMRECT)new CImpIEnumRECT();

   if (NULL==*ppEnum)
      return FALSE;

   //If creation worked, AddRef the interface
   if (NULL!=*ppEnum)
      (*ppEnum)->AddRef();

   return (NULL!=*ppEnum);
   }


CImpIEnumRECT::CImpIEnumRECT(void)
   {
   UINT       i;

   //Initialize the array of rectangles
```

```
   for (i=0; i < 15; i++)
      SetRect(&m_rgrc[i], i, i*2, i*3, i*4);

   //Ref counts always start as zero
   m_cRef=0;

   //Current pointer is the first element.
   m_iCur=0;

   return;
   }


CImpIEnumRECT::~CImpIEnumRECT(void)
   {
   return;
   }



DWORD CImpIEnumRECT::AddRef(void)
   {
   return ++m_cRef;
   }


DWORD CImpIEnumRECT::Release(void)
   {
   DWORD       cRefT;

   cRefT=--m_cRef;

   if (0==m_cRef)
      delete this;

   return cRefT;
   }


BOOL CImpIEnumRECT::Next(DWORD cRect, LPRECT pRect, LPDWORD pdwRects)
   {
   DWORD          cRectReturn=0L;

   if (NULL==pdwRects)
      return FALSE;
```

```
   *pdwRects=0L;

   if (NULL==pRect || (m_iCur >= CRECTS))
      return FALSE;

   while (m_iCur < CRECTS && cRect > 0)
      {
      *pRect++=m_rgrc[m_iCur++];
      cRectReturn++;
      cRect--;
      }

   *pdwRects=(cRectReturn-cRect);
   return TRUE;
   }



BOOL CImpIEnumRECT::Skip(DWORD cSkip)
   {
   if ((m_iCur+cSkip) >= CRECTS)
      return FALSE;

   m_iCur+=cSkip;
   return TRUE;
   }



void CImpIEnumRECT::Reset(void)
   {
   m_iCur=0;
   return;
   }
```

ENUM.H

```
/*
 * ENUM.H
 *
 * Definitions, classes, and prototypes for Enumerator interface
 * example implemented in C++.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
```

```
 */


#ifndef _ENUM_H_
#define _ENUM_H_


#define INITGUIDS
#include <bookguid.h>    //Found in shared include directory.
#include <ienum0.h>      //Found in shared include directory.


//Menu Resource ID and Commands
#define IDR_MENU                1

#define IDM_ENUMCREATE          100
#define IDM_ENUMRELEASE         101
#define IDM_ENUMRUNTHROUGH      102
#define IDM_ENUMEVERYTHIRD      103
#define IDM_ENUMRESET           104
#define IDM_ENUMEXIT            105



//ENUM.CPP
LRESULT FAR PASCAL __export EnumWndProc(HWND, UINT, WPARAM,
LPARAM);

class __far CAppVars
   {
   friend LRESULT FAR PASCAL __export EnumWndProc(HWND, UINT, WPARAM,
LPARAM);

   protected:
      HINSTANCE      m_hInst;          //WinMain parameters
      HINSTANCE      m_hInstPrev;
      UINT           m_nCmdShow;

      HWND           m_hWnd;           //Main window handle
      LPENUMRECT     m_pIEnumRect;     //Enumerator interface we have.

   public:
      CAppVars(HINSTANCE, HINSTANCE, UINT);
      ~CAppVars(void);
      BOOL FInit(void);
   };
```

```
typedef CAppVars FAR * LPAPPVARS;

#define CBWNDEXTRA          sizeof(LONG)
#define ENUMWL_STRUCTURE    0


//IENUM.CPP

//Number of rectangles that objects with IEnumRECT support (for demo)
#define CRECTS      15

/*
 * A class definition, which OLE doesn't provide, then inherits from
 * whatever interfaces it supports.  Multiple inheritance works fine
 * in this scenario as well as single inheritance shown here.
 */
class __far CImpIEnumRECT : public IEnumRECT
    {
    private:
        DWORD          m_cRef;         //Interface reference count.
        DWORD          m_iCur;         //Current enumeration position
        RECT           m_rgrc[CRECTS]; //Rectangles we contain for enumeration

    public:
        CImpIEnumRECT(void);
        ~CImpIEnumRECT(void);

        virtual DWORD AddRef(void);
        virtual DWORD Release(void);
        virtual BOOL  Next(DWORD, LPRECT, LPDWORD);
        virtual BOOL  Skip(DWORD);
        virtual void  Reset(void);
    };


typedef CImpIEnumRECT FAR * LPIMPIENUMRECT;


//Function that creates one of these objects
BOOL CreateRECTEnumerator(LPENUMRECT FAR *);


#endif //_ENUM_H_
```

ENUM.C

```
/*
 * ENUM.C
 *
 * Example enumerator interface user in C
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 *
 */



#include <windows.h>
#include <malloc.h>
#include "enum.h"


int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hInstPrev
   , LPSTR pszCmdLine, int nCmdShow)
   {
   MSG        msg;
   LPAPPVARS   pAV;

   //Create and initialize the application.
   pAV=AppVarsConstructor(hInst, hInstPrev, nCmdShow);

   if (NULL==pAV)
      return -1;
```

Listing 3-3: The ENUM program implemented in C.

```
   if (AppVarsFInit(pAV))
      {

      while (GetMessage(&msg, NULL, 0,0 ))
         {
         TranslateMessage(&msg);
         DispatchMessage(&msg);
         }
      }

   AppVarsDestructor(pAV);
   return msg.wParam;
```

```
    }



LRESULT FAR PASCAL __export EnumWndProc(HWND hWnd, UINT iMsg
    , WPARAM wParam, LPARAM lParam)
    {
    LPAPPVARS   pAV;
    RECT        rc;
    DWORD       cRect;

    COMMANDPARAMS(wID, wCode, hWndMsg);

    //This will be valid for all messages except WM_NCCREATE
    pAV=(LPAPPVARS)GetWindowLong(hWnd, ENUMWL_STRUCTURE);

    switch (iMsg)
        {
        case WM_NCCREATE:
            //CreateWindow passed pAV to us.
            pAV=(LPAPPVARS)((LONG)((LPCREATESTRUCT)lParam)->lpCreateParams);

            SetWindowLong(hWnd, ENUMWL_STRUCTURE, (LONG)pAV);
            return (DefWindowProc(hWnd, iMsg, wParam, lParam));


        case WM_DESTROY:
            PostQuitMessage(0);
            break;


        case WM_COMMAND:
            switch (wID)
                {
                case IDM_ENUMCREATE:
                    if (NULL!=pAV->m_pIEnumRect)
                        pAV->m_pIEnumRect->lpVtbl->Release(pAV->m_pIEnumRect);

                    CreateRECTEnumerator(&pAV->m_pIEnumRect);
                    break;


                case IDM_ENUMRELEASE:
                    if (NULL==pAV->m_pIEnumRect)
                        break;
```

```
            if (0==pAV->m_pIEnumRect->lpVtbl->Release(pAV->m_pIEnumRect))
               pAV->m_pIEnumRect=NULL;

            break;


         case IDM_ENUMRUNTHROUGH:
            if (NULL==pAV->m_pIEnumRect)
               break;

            while (pAV->m_pIEnumRect->lpVtbl->Next(pAV->m_pIEnumRect
               , 1, &rc, &cRect))
               ;

            break;


         case IDM_ENUMEVERYTHIRD:
            if (NULL==pAV->m_pIEnumRect)
               break;

            while (pAV->m_pIEnumRect->lpVtbl->Next(pAV->m_pIEnumRect
               , 1, &rc, &cRect))
               {
               if (!pAV->m_pIEnumRect->lpVtbl->Skip(pAV->m_pIEnumRect, 2))
                  break;
               }
            break;


         case IDM_ENUMRESET:
            if (NULL==pAV->m_pIEnumRect)
               break;

            pAV->m_pIEnumRect->lpVtbl->Reset(pAV->m_pIEnumRect);
            break;


         case IDM_ENUMEXIT:
            PostMessage(hWnd, WM_CLOSE, 0, 0L);
            break;
         }
      break;

   default:
      return (DefWindowProc(hWnd, iMsg, wParam, lParam));
```

```
    }

  return 0L;
  }



LPAPPVARS AppVarsConstructor(HINSTANCE hInst, HINSTANCE hInstPrev
  , UINT nCmdShow)
  {
  LPAPPVARS      pAV;

  pAV=(LPAPPVARS)_fmalloc(sizeof(APPVARS));

  if (NULL==pAV)
    return NULL;

  pAV->m_hInst     =hInst;
  pAV->m_hInstPrev =hInstPrev;
  pAV->m_nCmdShow  =nCmdShow;

  pAV->m_hWnd=NULL;
  pAV->m_pIEnumRect=NULL;

  return pAV;
  }



void AppVarsDestructor(LPAPPVARS pAV)
  {
  //Free any object we still hold on to
  if (NULL!=pAV->m_pIEnumRect)
    pAV->m_pIEnumRect->lpVtbl->Release(pAV->m_pIEnumRect);

  if (IsWindow(pAV->m_hWnd))
    DestroyWindow(pAV->m_hWnd);

  _ffree((LPVOID)pAV);

  return;
  }



BOOL AppVarsFInit(LPAPPVARS pAV)
```

```
    {
    WNDCLASS    wc;

    if (!pAV->m_hInstPrev)
        {
        wc.style         = CS_HREDRAW | CS_VREDRAW;
        wc.lpfnWndProc    = (WNDPROC)EnumWndProc;
        wc.cbClsExtra     = 0;
        wc.cbWndExtra     = CBWNDEXTRA;
        wc.hInstance      = pAV->m_hInst;
        wc.hIcon          = LoadIcon(pAV->m_hInst, "Icon");
        wc.hCursor        = LoadCursor(NULL, IDC_ARROW);
        wc.hbrBackground  = (HBRUSH)(COLOR_WINDOW + 1);
        wc.lpszMenuName   = MAKEINTRESOURCE(IDR_MENU);
        wc.lpszClassName  = "CEnum";

        if (!RegisterClass(&wc))
            return FALSE;
        }

    pAV->m_hWnd=CreateWindow("CEnum", "Enumerator in C"
        , WS_MINIMIZEBOX | WS_OVERLAPPEDWINDOW
        ,35, 35, 350, 250, NULL, NULL, pAV->m_hInst, pAV);

    if (NULL==pAV->m_hWnd)
        return FALSE;

    ShowWindow(pAV->m_hWnd, pAV->m_nCmdShow);
    UpdateWindow(pAV->m_hWnd);

    return TRUE;
    }
```

IENUM.C

```
/*
 * IENUM.C
 *
 * Implements the IMPIENUMRECT structure and functions (that is,
 * an object) in C.
 */

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

```
#include "enum.h"


//We have to explicitly define the fucntion table for IEnumRECT in C
static IEnumRECTVtbl  vtEnumRect;
static BOOL           fVtblInitialized=FALSE;


/*
 * CreateRECTEnumerator
 *
 * Purpose:
 *  Given an array of rectangles, creates an enumerator interface
 *  on top of that array.
 *
 * Parameters:
 *  ppEnum         LPENUMRECT FAR * in which to return the interface
 *                 pointer on the created object.
 *
 * Return Value:
 *  BOOL           TRUE if the function is successful, FALSE otherwise.
 */

BOOL CreateRECTEnumerator(LPENUMRECT FAR *ppEnum)
   {
   if (NULL==ppEnum)
      return FALSE;

   //Create the object storing a pointer to the interface
   *ppEnum=(LPENUMRECT)IMPIEnumRect_Constructor();

   if (NULL==*ppEnum)
      return FALSE;

   //If creation worked, AddRef the interface
   if (NULL!=*ppEnum)
      (*ppEnum)->lpVtbl->AddRef(*ppEnum);

   return (NULL!=*ppEnum);
   }


LPIMPIENUMRECT IMPIEnumRect_Constructor(void)
   {
   LPIMPIENUMRECT      pER;
   UINT                i;
```

```
    /*
     * First time through initialize function table.  Such a table could be
     * defined as a constant instead of doing explicit initialization here.
     * However, this method shows exactly which pointers are going where and
     * does not depend on knowing the ordering of the functions in the table,
     * just the names.
     */
    if (!fVtblInitialized)
        {
        vtEnumRect.AddRef =IMPIEnumRect_AddRef;
        vtEnumRect.Release=IMPIEnumRect_Release;
        vtEnumRect.Next   =IMPIEnumRect_Next;
        vtEnumRect.Skip   =IMPIEnumRect_Skip;
        vtEnumRect.Reset  =IMPIEnumRect_Reset;

        fVtblInitialized=TRUE;
        }

    pER=(LPIMPIENUMRECT)_fmalloc(sizeof(IMPIENUMRECT));

    if (NULL==pER)
        return NULL;

    //Initialize function table pointer
    pER->lpVtbl=&vtEnumRect;

    //Initialize the array of rectangles
    for (i=0; i < 15; i++)
        SetRect(&pER->m_rgrc[i], i, i*2, i*3, i*4);

    //Ref counts always start as zero
    pER->m_cRef=0;

    //Current pointer is the first element.
    pER->m_iCur=0;

    return pER;
    }



void IMPIEnumRect_Destructor(LPIMPIENUMRECT pER)
    {
    if (NULL==pER)
        return;
```

```
   _ffree((LPVOID)pER);

   return;
   }


DWORD IMPIEnumRect_AddRef(LPENUMRECT pEnum)
   {
   LPIMPIENUMRECT      pER=(LPIMPIENUMRECT)pEnum;

   if (NULL==pER)
      return 0L;

   return ++pER->m_cRef;
   }



DWORD IMPIEnumRect_Release(LPENUMRECT pEnum)
   {
   LPIMPIENUMRECT      pER=(LPIMPIENUMRECT)pEnum;
   DWORD               cRefT;

   if (NULL==pER)
      return 0L;

   cRefT=--pER->m_cRef;

   if (0==pER->m_cRef)
      IMPIEnumRect_Destructor(pER);

   return cRefT;
   }




BOOL IMPIEnumRect_Next(LPENUMRECT pEnum, DWORD cRect, LPRECT pRect,
LPDWORD pdwRects)
   {
   LPIMPIENUMRECT  pER=(LPIMPIENUMRECT)pEnum;
   DWORD           cRectReturn=0L;

   if (NULL==pdwRects)
      return FALSE;
```

```
   *pdwRects=0L;

   if (NULL==pRect || (pER->m_iCur >= CRECTS))
      return FALSE;

   while (pER->m_iCur < CRECTS && cRect > 0)
      {
      *pRect++=pER->m_rgrc[pER->m_iCur++];
      cRectReturn++;
      cRect--;
      }

   *pdwRects=(cRectReturn-cRect);
   return TRUE;
   }



BOOL IMPIEnumRect_Skip(LPENUMRECT pEnum, DWORD cSkip)
   {
   LPIMPIENUMRECT      pER=(LPIMPIENUMRECT)pEnum;

   if (NULL==pER)
      return FALSE;

   if ((pER->m_iCur+cSkip) >= CRECTS)
      return FALSE;

   pER->m_iCur+=cSkip;
   return TRUE;
   }



void IMPIEnumRect_Reset(LPENUMRECT pEnum)
   {
   LPIMPIENUMRECT      pER=(LPIMPIENUMRECT)pEnum;

   if (NULL==pER)
      return;

   pER->m_iCur=0;
   return;
   }
```

ENUM.H

```
/*
 * ENUM.H
 *
 * Definitions, structures, and prototypes for Enumerator interface
 * example implemented in C.
 *
 * Copyright (c)1993 Microsoft Corporation, All Rights Reserved
 */


#ifndef _ENUM_H_
#define _ENUM_H_


#define INITGUIDS
#include <bookguid.h>
#include <ienum0.h>      //Found in shared include directory.


//Menu Resource ID and Commands
#define IDR_MENU                1

#define IDM_ENUMCREATE          100
#define IDM_ENUMRELEASE         101
#define IDM_ENUMRUNTHROUGH      102
#define IDM_ENUMEVERYTHIRD      103
#define IDM_ENUMRESET           104
#define IDM_ENUMEXIT            105



//ENUM.C
LRESULT FAR PASCAL __export EnumWndProc(HWND, UINT, WPARAM,
LPARAM);

typedef struct tagAPPVARS
   {
   HINSTANCE       m_hInst;          //WinMain parameters
   HINSTANCE       m_hInstPrev;
   UINT            m_nCmdShow;

   HWND            m_hWnd;           //Main window handle
   LPENUMRECT      m_pIEnumRect;     //Enumerator interface we have.
```

```
   } APPVARS, FAR * LPAPPVARS;


LPAPPVARS AppVarsConstructor(HINSTANCE, HINSTANCE, UINT);
void      AppVarsDestructor(LPAPPVARS);
BOOL      AppVarsFInit(LPAPPVARS);


#define CBWNDEXTRA          sizeof(LONG)
#define ENUMWL_STRUCTURE    0


//Number of rectangles that objects with IEnumRECT support (for demo)
#define CRECTS      15


/*
 * In C we make a class by reusing the elements of IEnumRECT
 * thereby inheriting from it, albeit manually.
 */
typedef struct tagIMPIENUMRECT
   {
   IEnumRECTVtbl FAR * lpVtbl;
   DWORD          m_cRef;        //Interface reference count.
   DWORD          m_iCur;        //Current enumeration position
   RECT           m_rgrc[CRECTS]; //Rectangles we contain for enumeration
   } IMPIENUMRECT, FAR * LPIMPIENUMRECT;


/*
 * And, of course, in C we have to separately declare member functions
 * with globally unique names, so prefixing with the class name should
 * remove any conflicts.
 */

LPIMPIENUMRECT  IMPIEnumRect_Constructor(void);
void           IMPIEnumRect_Destructor(LPIMPIENUMRECT);

DWORD          IMPIEnumRect_AddRef(LPENUMRECT);
DWORD          IMPIEnumRect_Release(LPENUMRECT);
BOOL           IMPIEnumRect_Next(LPENUMRECT, DWORD, LPRECT, LPDWORD);
BOOL           IMPIEnumRect_Skip(LPENUMRECT, DWORD);
void           IMPIEnumRect_Reset(LPENUMRECT);


//Function that creates one of these objects
```

```
BOOL CreateRECTEnumerator(LPENUMRECT FAR *);


#endif //_ENUM_H_
```

RECTEnumerator and the IEnumRECT Interface

The RECTEnumerator object supports one interface called IEnumRECT as shown in Listing 3-1 with the following member functions:

AddRef	Increment the reference count on the enumerator object.
Release	Decrement the reference count and free the enumerator object when the reference count is zero.
Next	Return the next *n* RECT structures starting at the current index.
Skip	Skip past a *n* RECTs in the list.
Reset	Set the current index to zero.

Since we have not yet examined the IUnknown, and to keep this example as simple as possible, IEnumRECT borrows the two IUnknown members ::AddRef and ::Release but does not include ::QueryInterface. For the same reason we also eliminate ::Clone that is part of standard IEnum interfaces.

When IENUM0.H is compiled for C++ it generates a C++ abstract base class, that is, one that defines a set of pure virtual functions (with "virtual" and "=0"). In addition, IENUM0.H define a far pointer type for this interface of the conventional form LP<INTERFACE> where <INTERFACE> is the interface name in all caps excluding the 'I' prefix. The C++ implementation of the RECTEnumerator object, CImpIEnumRECT in the ENUM.H file of Listing 3-2, inherits these function signatures from this interface and provides each implementation, that is, the C++ class will generate the function table for us.

Defining an interface in C is more work, primarily because we have to construct the function table manually. In Listing 3-3 the structure IEnumRECTVtbl[1] is a structure of function pointers that is exactly what C++ compilers create internally for C++ classes. The actual interface IEnumRECT is defined as a structure that contains a pointer to this function table. So when a C application has a pointer to an interface, it really has a pointer to a pointer to a function table. The C implementation of the RECTEnumerator object, a structure called IMPIENUMRECT in ENUM.H of listing 3-3, duplicates the *lpVtbl* member of IEnumRECT in its own structure thereby making a pointer to IMPIENUMRECT polymorphic with a pointer to IEnumRECT. This duplicates what happens automatically with C++ classes and is common in C-based OLE 2.0 code.

Creating the RECTEnumerator Object

When you choose the Enum/Create command from the menu of the ENUM program you will generate a call (see WM_COMMAND/IDM_ENUMCREATE) to CreateRECTEnumerator. This creation function creates the object and returns the IEnumRECT interface pointer in an **out-parameter**, that is, the caller passes the address in which CreateRECTEnumerator stores the interface pointer. This technique is used everywhere in OLE 2.0 to allow standardization of almost all return values into a type called HRESULT described in the section "HRESULT and SCODE" below. To not complicate your life now with HRESULT, we'll stick with a BOOL return type for now.

I must point out here that specific functions to create some type of object are rare in OLE 2.0: most often you use a class object and IClassFactory::CreateInstance which eliminate the need for most, but not all, APIs like CreateRECTEnumerator. As we'll see in Chapter 4, implementations of IClassFactory::CreateInstance look very much like CreateRECTEnumerator, in either language.

In this example the ENUM programs are both user and implementor of the same object. Both programs use an internal function, CreateRECTEnumerator to obtain an IEnumRECT pointer to the RECTEnumerator object. This demonstrates the typical fashion through which a user obtains an interface pointer by calling an API. Internally the CreateRECTEnumerator creates the function table for the IEnumRECT interface then allocates and initializes the object itself.

In C++ the new operator applied on the CImpIEnumRECT class automatically allocates memory for the

1The jargon name "Vtbl" means "virtual function table" which for this book is always refered to as just a function table.

object and creates the function table–all we have to do is initialize the object. Since the CImpIEnumRECT class inherits from IEnumRECT, we can typecast the pointer from new to an LPENUMRECT which turns it into a pointer to just the interface. So in C++ it's highly convenient to implement objects as C++ objects, although the pointers we return are always interface pointers.

In C we must manually fill the function table, manually allocate the object's memory, then initialize it as exactly as in C++. The function IMPIEnumRECT_Constructor handles all this for us which we use in place of the new operator in C++. This constructor function first creates the function table by storing function pointers in a global array of type IEnumRECTVtbl (this only needs to happen once for all instances of the object). Only the implementation of this object knows that the function table actually exists in a global variable like this–the object's user just sees the table but has no knowledge about where that table lives. In any case IMPIEnumRECT_Constructor then allocates the object's structure and stores a pointer to the function table in the *lpVtbl* member of the object. Finally it performs the same initialization as in C++.

Using an IEnumRECT Pointer

You will also notice that CreateRECTEnumerator calls the object's AddRef function before returning the pointer. This is one rule of reference counting as explained in "Reference Counting" below. However, this one call, along with calls to the other IEnumRECT functions in ENUM.C and ENUM.CPP, demonstrate the calling differences between C and C++. Given an interface pointer, a C++ user calls member functions through the pointer as with any other C++ object pointer:

```
//C++ call to interface member function
pIEnumRect->AddRef();
```

This will land in CImpIEnumRECT::AddRef just as any other C++ call would. The *this* pointer inside the member function is identical to pIEnumRECT through which the function was called.

In C we have a more complicated story. First, any member function call made through an interface pointer must indirect through the *lpVtbl* member first before getting at the function, and indirection done automatically in C++:

```
//C call to interface member function
pIEnumRect->lpVtbl->AddRef();
```

In order that the implementation of IEnumRECT::AddRef invoked here knows which object is being accessed, C users must pass the same interface pointer as the first parameter to the function. This mimics the behavior of the *this* pointer that is automatic in C++. Since this extra parameter is necessary the function prototypes in IENUM0.H for the C interface had to include the pointer type as the first parameter.

The two lines of code above illustrate why C++ is more **convenient**, but no more **functional**, than C in using objects. The C user will always need the extra indirection and the extra parameter which can add up fast to a lot of extra code. By no means, however, does that small fact render it impossible to write C code for OLE 2.0. An object implementor in C only needs to provide for creating the function table manually. But besides from these few differences, programming in OLE 2.0 is identical in either language.

Reference Counting

The implementation of the RECTEnumerator object is illustrative, but not useful. In order to build the bridge between illustrative objects and useful objects we need more information that applies to all remaining chapters of this book. One of the most important subjects is that of reference counting which is a set of rules to control an object's lifetime.

As an object, your reference counting is like living life where you are not allowed to rest eternally unless all your acquaintances have also passed on (obviously not everyone can be the same object or we'd all be immortal!) At birth you form an acquaintance with your mother and as you live life you meet new people and form more acquaintances. Whenever you form a new relationship you increment your reference count. Now when any acquaintance dies, you are released of that relationship and you decrement your reference count. Only when all relationships have ended are you allowed your personal journey to the afterlife. For an object, that reference count of zero means you are allowed to free your memory.

The rules governing reference counting can be distilled down to two fundamental principles:

1. Creation of a new interface pointer to an object must be accompanied by an ::AddRef call to the object through that new pointer.
2. Destruction of an interface pointer (that is, when it goes out of scope) must be accompanied by a ::Release call through that pointer before it is destroyed.

This means that whenever you assign one pointer to another in some piece of code you should ::AddRef the new copy (the left operand) of the pointer. Before that pointer is overwritten it must have ::Release called. All ::AddRef and ::Release calls made through interfaces affect the reference count of the entire object, not just the interface. Consider the following code:

```
LPSOMEINTERFACE   pISome1;
LPSOMEINTERFACE   pISome2
LPSOMEINTERFACE   pCopy;

//A function that creates the pointer AddRef's it.
CreateISomeObject(&pISome1);   //Some1 ref count=1
CreateISomeObject(&pISome2);   //Some2 ref count=1

pCopy=pISome1;            //Some1 count=1
pCopy->AddRef();          //AddRef new copy, Some1=2

[Do things]

pCopy->Release();         //Release before overwrite, Some1=1
pCopy=pISome2;            //Some2=1
pCopy->AddRef();          //Some2=2

[Things that make you go]
```

```
pCopy->Release();          //Release before overwrite, Some2=1
pCopy=NULL;

[Do more things]

pISome2->Release();        //Release when done, Some2=0, Some2 freed.
pISome1->Release();        //Release when done, Some1=0, Some1 freed.
```

An object's lifetime is controlled by all ::AddRef and ::Release calls on all its interfaces combined. Reference counting a specific interface is useful in debugging to verify that your user is calling counting you properly, but it is the object reference count that matters. Functions that create objects and return pointers are the functions that actually create the pointers. Such functions fill the out-parameters from which the caller receives the pointer. Therefore it is the creator, not the caller, that is responsible for the first ::AddRef on the object through the interface pointer it initially returns.

If some function receives an interface pointer through an *in-out parameter*, that function must be sure to call ::Release through that pointer before overwriting it and must ::AddRef the new pointer stored in its place.

My Kingdom for some Optimizations!

The stated rules and their affect on the code shown above probably seem rather fascist. Well, it is, but that doesn't mean there's no underground movement.

When you know the lifetimes of all pointers to the same object, you can bypass the majority of ::AddRef and ::Release calls. There are two manifestations of such knowledge: nested and overlapping lifetimes.

In the code above, every instance of pCopy is nested within the lifetimes of pISome1 and pISome2, that is, the copy lives and dies within the lifetime of the original. After CreateISomeObject is called, both objects have a reference count of one. The lifetimes of their pointers is bounded by these create calls and the final ::Release calls through those pointers. Since we know these lifetimes, we can eliminate any other ::AddRef and ::Release calls to copies of those pointers:

```
LPSOMEINTERFACE   pISome1;
LPSOMEINTERFACE   pISome2;
LPSOMEINTERFACE   pCopy;

CreateISomeObject(&pISome1);  //Some1 ref count=1
CreateISomeObject(&pISome2);  //Some2 ref count=1

pCopy=pISome1;             //Some1=1, pCopy nested in Some1's life

[Do things]

pCopy=pISome2;             //Some2=1, pCopy nested in Some2's life

[Do other things]

pICopy=NULL;               //No ::Release necessary

[Do anything, then cleanup]

pISome2->Release();        //Release when done, Some2=0, Some2 freed.
pISome1->Release();        //Release when done, Some1=0, Some1 freed.
```

Overlapping lifetimes are those when the original pointer dies after the copy is born but before the copies dies itself. If the copy is alive at the original's funeral, it can inherit ownership of the reference count on behalf of the original:

```
LPSOMEINTERFACE   pISome1;
LPSOMEINTERFACE   pCopy;

CreateISomeObject(&pISome1);  //Some1 ref count=1

pCopy=pISome1;      //Some1=1, pCopy nested in Some1's life
pISome1=NULL;       //Pointer destroyed, pCopy inherits count, Some1=1

pCopy->Release();   //Release inherited ref count, Some1=0, Some1 freed.
```

With these optimizations, reference counting can be reduced to four specific rules where an AddRef on a new copy of a pointer is necessary (and thus must be ::Released themselves when destroyed):

1. Functions that return a new interface pointer in an out-parameter, or as a return value must ::AddRef the object through that pointer before returning.
2 Functions that accept an in-out parameter must ::Release the in-parameter before overwriting it and ::AddRef the out-parameter. Callers of these functions must ::AddRef the passed pointer to maintain a separate copy since the called function ::Releases.
3. If two pointers to the same object have unrelated or lifetimes then ::AddRef must be called on each.
4. ::AddRef each local copy of a global pointer.

In all cases some piece of code must ::Release for every ::AddRef. In case #1 above the caller of a function that returns a new pointer (like CreateRECTEnumerator) becomes responsible for that new object. When the caller is done with the object, it must call ::Release. If the object's reference count is decreased to zero because of this, the object may be destroyed at the discretion of the implementor, but from the user's point of view the object is gone. If you fail to ::Release a reference count on your behalf, you generally doom the object to the boredom of useless immortality–memory may not be freed or the DLL or EXE supplying that object may not shut down. Be humane to your objects: be sure to release them.

Call-Use-Release

The first optimized reference counting rule above exposes a common pattern in OLE 2.0 programming. To use some object you will call some function that returns you a pointer to an interface. That function will call ::AddRef on behalf of this new pointer. You then use that pointer for however long you wish in whatever code. When you are done with it, call ::Release through that pointer to let it know you no longer need it.

The same object may, in fact, be in use through other pointers, even in another process. As far as you're concerned, you called the ::Release to free the reference count for which you were responsible, and after that time you know you cannot access that object again, for it may have been freed itself. If there, however, are other outstanding pointers to that object elsewhere the object is still in memory, but you don't care.

The final ::Release may do more than just free the object, since "free the object" may imply many other actions. For example, Compound File objects discussed in Chapter 5 may close a file; a memory manager object we'll see in Chapter 4 will free any allocations it has made; a compound document object we'll implement in Chapter 10 may close down an application. Since the ::Release member function can be overloaded in this manner, you will notice an absence of "close" APIs in OLE 2.0. There is an API to open a Compound File, but there is no API to close it–the API provides the initial ::AddRef, and closure is handled in the final ::Release.

IUnknown, The Root of All Evil

From our above discussion we can isolate two fundamental interface and object operations: reference counting and pointer creation. The interface called IUnknown that all objects support encapsulates these two ideas in three members functions:

::QueryInterface	Return a pointer to the requested interface on the same object. QueryInterface is considered a function that creates a pointer, so calls ::AddRef through any pointer it returns.
::AddRef	Increment the object's reference count.
::Release	Decrement the object's reference count and free the object when the reference count reaches zero.

Since ::AddRef and ::Release behave exactly as described in the previous section we won't examine them further here. Instead, we'll look closer at ::QueryInterface.

::QueryInterface is more than just the fundamental creator of interface pointers, but always returns a pointer to a different interface on the same object. That is, QueryInterface allows you to access eac separate function table supported by an individual object. How you obtain the first interface pointer on the object is one thing–QueryInterface allows you to get to those other interface pointers.

QueryInterface also allows an object user to discover an object's capabilities at run-time, instead of having incorporating specific knowledge about objects at compile-time. You learn capabilities by asking for additional interfaces that the object supports, a process called interface negotiation. When a you create an arbitrary object from some arbitrary class ID you should generally ask for an IUnknown pointer on that

object. If OLE can find the object then you will *always* be able to get an IUnknown pointer.

With this pointer you can now determine if the object supports a particular feature by calling QueryInterface. For example, to determine if the object supports data transfer call QueryInterface asking for an IDataObject interface (see Chapter 6). To determine if the object is a compound document object, meaning that it can be treated in a standard way for editing capabilities, QueryInterface for IOleObject (see Chapters 9, 10). IOleObject describes only an embedded object so if you want to determine if it support linking, QueryInterface for IOleLink (Chapter 9). To go even further you can ask the object if it supports in-place activation by calling QueryInterface for IOleInPlaceObject (see Chapters 15, 16, 17).

When you QueryInterface for a new pointer, you not only learn whether the object is capable of the set of functions implied by that interface, but you receive back the interface pointer through which you access those functions. This means that you cannot possibly attempt to use certain features on an object if it does not support those features because you can never get the right interface pointer from the object. In other words, if you speak a different language than I do, we cannot communicate: only when we establish a common language can we express our ideas, our functions, to one another as shown in Figure 3-5. Furthermore, it is impossible for me to offend you verbally unless I speak your language, that is, I am not able to pass the wrong object to a function that does not understand that object because I must use the language of the object to perform any function on it.

Preview Note: Permission has not been secured for the illustration that will go here (it's a Far Side cartoon).

Figure 3-5: IUnknown::QueryInterface lets one object determine how another object might communicate.

Applications also benefit from being able to dynamically make decisions about how to treat an object based on that object's capabilities, instead of rigidly compiling such behavior. Let's say I work at the United Nations building in New York City and I speak English and German. I walk into a room with 10 international delegates with whom I need to discuss a few issues. So I go up to one of the delegates and ask "Do you speak English?" which is met with an affirmative "Yes." Great, now we can talk. Well, partway through our conversation I find that I simply cannot express one of my ideas in English but I know I could express it in German–some languages have words that without equivalents in other languages. For example, English does not have a gender-neutral pronoun on the same lines as "his" or "her" and so it takes reasonable effort to eliminate sexist overtones from writing or speech. Other languages, on the other hand, do have such a pronoun, and so as I prefer to communicate without gender bias, I would prefer to use that language over English. So I ask "Sprechen Sie Deutsche?" to which the other responds "Ja." Since my partner also speaks German, I can now express my idea in that language. If German was not in my partner's repertoire then we would be limited to speaking English, and English only.

Note that my ability to communicate with anyone is limited not by all the languages I speak, but by the languages myself and the other person have in common. This means that my level of communication varies from person to person. With some people I can converse in two languages, with others I might only converse in one, or I may not be able to converse at all. The key point is that I learn this when I meet the person, and that my knowing many languages only allows me to speak with many more people, not only with people that speak exactly the same set of languages.

So how does this apply to objects? The QueryInterface mechanism allows an object, or a user of objects, to implement or be able to use as many interfaces as desired without any fear of excluding yourself from being able to converse with another piece of code. For example, a compound document can implement full in-place activation capabilities without restricting itself to be useful only to in-place container applications: a non-in-place container can still use that object as a non-in-place object and in such case the in-place activation interfaces are ignored entirely.

I've often been asked why there is not a function that returns a list of all the interfaces an object supports. The answer is that such a function is, for the most part, useless. What would you do programmatically with such information? While it might be useful in some very esoteric circumstances, you never really need to know whether or not an object supports an interface unless you intend to perform some function through that interface: so you ask for it via QueryInterface. Furthermore, an object may dynamically change what interfaces it supports and therefore having some list of interfaces obtained eons ago becomes obsolete and useless. For these, and I'm sure other reasons, QueryInterface is the only way to learn about an object's

capabilities.

QueryInterface vs. Inheritance

QueryInterface is an improvement over the use of base classes an inheritance for two reasons. The first major advantage is that given an arbitrary C++ object pointer to some base class object you really have no way to determine if that pointer is actually referring to some derived class object instead, that is, you have no way to examine the virtual function table to see just exactly what kind of object you have. Therefore you are *always and forever* restricted to dealing with that object on the base class's terms. QueryInterface, on the other hand, allows you to get at any virtual function table you want from the base IUnknown interface. Given any IUnknown you can find out how rich the object actually is. You can get from the base to more specific interfaces.

The second major advantage of QueryInterface is that unless an object supports an interface, you cannot call member functions that the object does not support. This is not true in C++ objects. Take, for example, the base object class CObject in Microsoft's Foundation Classes (MFC). A CObject may be capable of serializing itself to some storage device (like a file) and to ask the **question** "Can you serialize yourself" you call the member function ::IsSerializable. If the answer is positive, you can then call another function ::Serialize to actually perform the task.

That's nice, but a user of a CObject is in no way barred from calling ::Serialize at any time. In other words, the capability of serialization is not tightly coupled to the question of whether or not the object can actually serialize. The QueryInterface mechanism, on the other hand, does tightly couple the question and the capability. You must ask the object via QueryInterface whether it supports particular functionality (that is, for one of the IPersistStorage, IPersistFile, and IPersistStream interfaces) and *only if it does are you provided the interface through which to call such functions*. Given an arbitrary IUnknown object, you cannot possibly ask it to serialize itself without first asking for an interface that knows about serialization. If the object does not support the capability, you cannot get the interface. Therefore you cannot call unsupported functions.

Some Data Types and Calling Conventions

If you look in OLE 2.0's include file COMPOBJ.H you will find IUnknown declared as:

```
DECLARE_INTERFACE(IUnknown)
  {
  STDMETHOD(QueryInterface) (THIS_ REFIID riid, LPVOID FAR* ppvObj) PURE;
  STDMETHOD_(ULONG,AddRef) (THIS)  PURE;
  STDMETHOD_(ULONG,Release) (THIS) PURE;
  };
```

Offhand this will look very odd, and there are a number of components shown here that are used throughout the rest of OLE 2.0.

DECLARE_INTERFACE, STDMETHOD, STDMETHOD_, THIS, THIS_, and PURE are all macros that abstract the differences between C and C++ interface definitions as well as Win16, Win32, and MacIntosh implementations. When this interface declaration is compiled in C++ the result is similar to the definition of IEnumRECT in Listing 1-1; the same goes for a C compilation. For complete detail about how these macros expand, see the comments in the COMPOBJ.H file in your OLE 2.0 toolkit.

Note also that interfaces shown in Listing 3-3 are not exactly what is generated through these macros as real OLE 2.0 interfaces differ in calling convention and return type. The sections below look at these in more detail as well as the REFIID type

STDMETHOD and Associates

The STDMETHOD macro expands into HRESULT STDMETHODCALLTYPE. HRESULT is a special return value type discussed in section 3.6.2. STDMETHODCALLTYPE is defined under Windows 3.1 as __export __far __cdecl and under Windows NT as __export __cdecl. The cdecl type was necessary to support generation of the proper stack frame for member function calls, portability of C code from 16- to 32-bit, and interoperability between C and C++ implementations.[1] The OLE 2.0 architects would have preferred to use a more efficient calling convention (such as PASCAL) but were unable due to these constraints.

STDMETHOD_ allows a variation on the return type from HRESULT into any other type. ::AddRef and ::Release, for example, return the new reference count instead of an HRESULT. In this sense they are fairly unique: most other interface members in all of OLE 2.0 return HRESULTs, including ::QueryInterface.

Under C++ both STDMETHOD and STDMETHOD_ include the virtual keyword. The PURE macro also compiles under C++ as =0 to generate pure virtual members in the interface declarations. Of course, C

1Again, this convention is compatible with at least Microsoft C/C++ 7.0, Visual C++ 1.0, and Borland C++ 3.1.

compilations include neither.

Since the two STDMETHOD macros possible generate the virtual and =0 signatures, they are not used when implementing an interface, only in declaring one. Instead you use STDMETHODIMP or STDMETHODIMP_(<type>) which does nothing more than eliminate the virtual keyword in C++ compilations, but still generate either an HRESULT or <type> return value along with STDMETHODCALLTYPE.

HRESULT and SCODE

OLE 2.0 introduces a new return type used by ::QueryInterface and just about every other interface member function: HRESULT, or handle to a result. Conceptually an HRESULT is a status code, or SCODE, that describes what occurred, and a handle that can be used to obtain additional information about an error or how to recover from it. The intention is that over time, interfaces will return very detailed information that can describe a suggested course of action when failure occurs. Perhaps there might even be some other error-handling object that would figure out exactly how to handle any given HRESULT, perhaps using some sort of expert system internally.

SCODEs are 32-bit values containing a severity flag, a facility code, and an information code organized as shown in Figure 3-6. The Context field is reserved for future use when it will contain the handle to additional information. SCODEs are created or dissected using various macros in the OLE 2.0 include file SCODE.H

Figure 3-6: Structure of an SCODE

such as MAKE_SCODE. Some of the more commonly used SCODEs are shown in Table 3-1. An SCODE's facility field describes the source of the error, which might be in the marshaling of the function call, in the interface function itself, or elsewhere. Those SCODEs containing S_ carry information and mean 'success' whereas those containing E_ mean 'failure.' Many SCODEs are prefixed with other symbols, like "OLE_", that identify the specific technology generating the error. "OLE_" in this case means compound documents.

Table 3-1: Common SCODEs

S_OK	Function succeeded. Also used for functions that semantically return boolean information that succeed with a TRUE result.
S_FALSE	Function that semantically returns boolean information succeeded with a FALSE result.
E_NOINTERFACE	::QueryInterface could not return a pointer to the requested interface.
E_NOTIMPL	Member function contains no implementation.
E_FAIL	Unspecified failure
E_OUTOFMEMORY	Function failed to allocate necessary memory.

Since HRESULTs and SCODEs are not straight equivalents, OLE 2.0 provides a few functions (implemented in 2.0 as macros) that provide conversions between HRESULTs and SCODEs, both of which you'll use often in your own implementations. To create an HRESULT from an SCODE use the function ResultFromScode(SCODE) . To dig an SCODE out of an HRESULT use the function GetScode(HRESULT) (also a macro).

While this seems like a pain, especially on the receiving end of an HRESULT, the most common case allows you to bypass these functions altogether. If a function works completely it can return the pre-defined HRESULT called NOERROR, the equivalent of an HRESULT containing S_OK. The code receiving the HRESULT may use one of two macros to determine success or failure of the function (hr stands for some HRESULT):

SUCCEEDED(hr)	Tests the high bit of the HRESULT and returns TRUE if that bit is clear. This will return TRUE for any S_ SCODE and FALSE for any E_ SCODE.
FAILED(hr)	Tests the high bit of the HRESULT and returns TRUE if that bit is set. This will return TRUE for any E_ SCODE and FALSE for any S_ SCODE.

Use of SUCCEEDED and FAILED is preferred to comparing an HRESULT to NOERROR directly because some codes, such as S_FALSE or STG_S_CONVERTED (see Chapter 5) mean that the function actually succeeded and is returning more information than just that simple fact. A test like (NOERROR!=hr) will be FALSE with a HRESULT contains S_FALSE whereas the FAILED(hr) macro will be TRUE. The GetScode function is only really necessary when you want to find the exact reason for failure instead of just the fact that the function did fail.

Globally-Unique Identifiers: GUIDs, IIDs, CLSIDs

Every interface is defined by an interface identifier, or IID, which is a special case of a globally-unique identifier, or GUID (pronounced goo-id). GUIDs are 128-bit values created with a DEFINE_GUID macro (see GUID.H in the OLE 2.0 kit). Every interface and object class uses a GUID for identification. As described in the OLE 2.0 kit, Microsoft will either allocates one or more sets of 256 GUIDs for your exclusive use when you request them, or if you have a network card in your machine you can run a tool called UUIDGEN.EXE that will provide you with a set of 256 GUIDs based on the time of day, the date, and a unique number contained in your network card. The chance of duplicate GUID generated by this tool is about the same as two random atoms in the universe colliding to form a small avacado. In other words, don't worry about it.

NOTE: All the code shown in this book uses GUIDs prefixed with 000211 which are allocated for the author. **Do not use these GUIDs for your own products. Mine Mine Mine!**

OLE 2.0 defines IIDs for every standard interface along with a class IDs (CLSID) for every standard object class. When we call any function that asks for an IID or CLSID we pass a *reference* to an instance of the GUID structure that exists in our process space using the types REFIID or REFCLSID. When passing an IID or CLSID in C, you must use a pointer, that is, pass &IID_* or &CLSID_*; REFIID and REFCLSID are typed as const pointers to IID or CLSID. In C++, since a reference is a natural part of the language, you drop the &. We will see more specifics about the definition and use of GUIDs in Chapter 4 and beyond.

Finally, to compare two GUIDs for equality, use the IsEqualIID function that is contained in COMPOBJ.DLL, defined in COMPOBJ.H, and imported with COMPOBJ.LIB. You will use IsEqualIID frequently in implementations of QueryInterface.

OLE 2.0 Interfaces and APIs

OLE 2.0 defines no less than 62 interfaces, many of which it implements and uses internally. Those of importance to applications are shown in Figure 3-7, grouped together into the technology area to which they apply. Remember that higher technologies build on the lower technologies as discussed in Chapter 1.

This picture may look a little intimidating at first. There are many interfaces shown but only a handful that your application actually has to implement. For basic container applications that we'll see in Chapter 9, you only need to implement IOleClientSite and IAdviseSink. Where you only implement a few, you *use* many more, thereby contributing to the magnitude of Figure 3-7. In addition, some of the interfaces shown are only useful as base interfaces for others. For example, IPersist is the base class of IPersistFile and IPersistStorage. Rarely will you use or implement a simple base class by itself, if ever.

Figure 3-7: Interesting Interfaces in OLE 2.0 Technologies.

Custom Interfaces

While OLE 2.0 defines many standard interfaces, object can define and implement their own as long as their potential users are implemented to be aware of those interfaces. To reiterate a point, the interfaces that make up Compound Documents aim to eliminate the need for custom interfaces in particular scenarios. By eliminating custom interfaces from an object you greatly reduce the amount of code needed in a potential user of that object. The Polyline object we start developing in Chapter 4 begins its life with a custom interface, but throughout this book we'll convert it piece by piece into standard compound document interfaces.

The big restriction to custom interfaces is that they can only be used on objects that are implemented in DLLs, that is, objects implemented in other applications (that is, process spaces), can only expose standard interfaces to users. The reason is that all function parameters must be *marshaled* across the process boundary since the processes may differ with respect to 16- and 32-bit addressing, etc. OLE 2.0 internally implements

marshaling support for those standard interfaces that cross process boundaries, but at this time there no available mechanism to marshal parameters for custom interfaces. If you must call custom functions across process boundaries then you can use the Automation interface, IDispatch, to define and invoke those functions since IDispatch itself can be marshaled.

Preview Note: Seems that there will be a method to create custom marshaling for a custom interface. That will have to be included in Chapter 4.

Interfaces vs. APIs

In developing with OLE 2.0 you'll soon notice that you use relatively few APIs to achieve your goals as opposed to calling interface functions. Almost everything an OLE 2.0 application needs to do can be accomplished through obtaining a pointer to an interface and its member functions. Interfaces, in fact, make up more of the so-called API for a user of an object and define the implementation for an object itself instead of using more archaic mechanisms like explicit named exports, callback functions, and messages. There are only a few truly fundamental APIs in OLE 2.0, mostly concerned with creating objects or manipulating things like class IDs.

The majority of the hundred or so OLE 2.0 APIs are really 'wrappers' to sequences of commonly used interface calls. Some, for example, are the equivalent of calling ::QueryInterface for a specific interface, calling a member function with default parameters, and releasing that interface (Call-Use-Release again). Such wrappers are provided to simplify application development in most cases; their use is seldom required, but you can benefit from the convenience. Even then, typical compound document containers or objects, even with full in-place activation and drag-drop implemented, will generally use about 20 of these APIs total. Some you might use for very specific reasons; others, while they exist, you probably will never use.

If you are familiar with the OLE 1.0 API you'll find that many operations that were APIs, like OleSetHostNames, are replaced by an interface call, like IOleObject::SetHostNames. Many more should become apparent as we implement features using OLE 2.0.

A major advantage of defining new functions as interfaces means that people outside Microsoft can publish a new interface by providing an include file that defines the interface ID and the interface members, and possibly providing a marshaling DLL if you want that interface to be implementable in an EXE. This means that you require no update of the operating system to accomodate your functions and yourself and others can immediately start to use those interfaces without waiting for Microsoft to revise the system!

What is a Windows Object? (reprieve)

A Windows Object is any object, however it manifests itself, that supports at least one interface, IUnknown. A Windows Object must be able to provide a separate function table for each interface it supports. The implementations of IUnknown members in each supported interface must be aware of the entire object since it must be able to access all other interfaces in the object as well as be able to affect the object's reference count.

C++ multiple inheritance is a convenient way to provide multiple function tables for each interface as the compiler generates them automatically. Since each implementation of member functions are already part of your object class, they automatically have access to everything in the object.

However, since this book is targeted to help both C and C++ programmers alike, I will take a different approach. The object class itself will only inherit from IUnknown and implement these functions to control the object as a whole. Each interface supported by this object is implemented in a separate C++ class (what would be a C structure if you are taking that route) that singly inherits from the interface its implementing. These "interface implementations" are instantiated with the object and live as long as the object lives.

The IUnknown members of these "interface implementations" always delegate to some other IUnknown implementation, which in most cases is the overall object's IUnknown. Each interface implementation also holds a "back pointer" to the object in which they are contained in order that they might be able to get back to information stored in the object for the use of all interfaces. In C++ this generally requires that each interface implementation class be a friend of the object class. It is also highly useful to maintain an interface-level reference count for debugging.

You may still have one question in your mind: what about inheritace for Windows Objects? Can one Windows Object inherit from another? The truth is that there is no inheritance mechanism for inheritance is a way to achieve code re-use in C++. The Windows Object mechanism for re-use is called aggregation. But alas, we are beginning to discuss the finer details of implementation. So with that, we can close this chapter.

Summary

An object in any object-oriented presentation is a self-contained unit of data and functions to manipulate that

data. A Windows Object is a special manifestation of this definition that presents its functions as separate groups called interfaces. Windows Objects differ from C++ object in construction and use, but are more powerful than C++ objects as they can live anywhere on the system and still be as usable to an application as if they were incorporated into that application. The most fundamental question that forms a theme for this book is how to obtain interface pointers for a variety of objects and what you can do with that pointer once you obtain it.

The most basic functions of all interface pointers are concerned with reference counting and for obtaining other interface pointers on the same object. These functions are collected in an OLE 2.0 interface called IUnknown. Later chapters in this book deal with more specific types of objects, their special interfaces, how you obtain those interface pointers, and what you can do with them.

Implementations and users of objects written in C and C++ differ only slightly. Calling a member function through a C pointer requires an extra dereference through a pointer to an interface function table and passage of an extra parameter to simulate C++'s *this* pointer. C objects must manually construct the function tables for their interfaces. While C++ is more convenient, it is not the required language of OLE 2.0.

A type of object called an enumerator provides functions through IEnum interfaces to iterate over a list of elements. Since Windows Objects are portable across process boundaries, an enumerator object is used to pass lists of information across those same boundaries as well as the functions to iterate over that list.

OLE 2.0 interface members use a specific cdecl calling convention and the OLE 2.0 header files define a number of macros to isolate machine-specifics from the definitions of interfaces. Most interface members also return a type called an HRESULT that contains detailed error information.

OLE 2.0 defines a number of standard interfaces but allows those objects implemented in DLLs to define and implement custom interfaces. There is no current support for marshaling custom interfaces across a process boundary. While OLE 2.0 defines many interfaces, applications need only worry about a handful depending on the features those applications wish to implement.