Sample Chapter

C++/CLI IN ACTION

Nishant Sivakumar

MANNING

C++/CLI in Action

by Nishant Sivakumar

Sample Chapter 4

brief contents

4

Introduction to mixed-mode programming

In the previous three chapters of this book, we've gone through the syntax and semantics of the C++/CLI language. It's now time to step it up a gear and move into the world of *mixed-mode programming*.

When I say mixed-mode programming, I mean mixing managed and native code in the same assembly. You can't do that with any other CLI language, because except for C++/CLI, every other CLI compiler only understands managed code. Because C++ is intrinsically an unmanaged programming language, mixing managed and native code is absolutely within its capabilities. Of course, things don't work the same way in managed code as they do in native code. The types are different; memory allocation is manual in one and automatic in the other; the API is different; and so on. When you mix the two, you need to be able to accommodate the use of managed and native types together, to convert between types, and to smoothly interop between the managed and native code.

I'm going to show you the basic techniques that you can apply in your mixed-mode programming projects. By the end of this chapter, you'll be pretty comfortable with mixed-mode programming concepts. We'll begin with a look at the concept of CLI pointers and discuss both pinning pointers and interior pointers. We'll also discuss how they can be used to perform typical pointer operations. CLI pointers are frequently required when you're doing mixed-mode programming, especially when you're utilizing native APIs that accept pointer arguments. Thus it's important to understand how they work and how they can be used to interop between native and managed code.

We'll also briefly look at the various interop mechanisms available for mixing managed and native code, such as COM Callable Wrappers (CCW), P/Invoke, and C++ interop. Although we'll be exclusively using C++ interop in this book, it's important to be aware of the other mechanisms available and their pros and cons when compared with C++ interop.

We'll also examine how mixed types can be implemented using C++/CLI; in the course of the discussion, you'll learn how to develop a managed smart pointer class that will handle automatic resource deallocation for a native resource when used in a managed class. Mixed types will be a prominent feature in your mixed-mode programming adventures; thus, a good understanding of how to use them will be extremely beneficial to you.

We'll round off the chapter with a discussion of how to convert between CLI delegates and unmanaged function pointers using two new methods added in .NET 2.0 to the `Marshal` class. This knowledge will be useful to you when wrapping native API that uses unmanaged function pointers as callbacks.

4.1 Using interior and pinning pointers

You can't use native pointers with CLI objects on the managed heap. That is like trying to write Hindi text using the English alphabet—they're two different languages with entirely different alphabets. Native pointers are essentially variables that hold memory address locations. They point to a memory location rather than to a specific object. When we say a pointer points to an object, we essentially mean that a specific object is at that particular memory location.

This approach won't work with CLI objects because managed objects in the CLR heap don't remain at the same location for the entire period of their lifetime. Figure 4.1 shows a diagrammatic view of this problem. The Garbage Collector (GC) moves objects around during garbage-collection and heap-compaction cycles. A native pointer that points to a CLI object becomes garbage once the object has been relocated. By then, it's pointing to random memory. If an attempt is made to write to that memory, and that memory is now used by some other object, you end up corrupting the heap and possibly crashing your application.

C++/CLI provides two kinds of pointers that work around this problem. The first kind is called an *interior pointer*, which is updated by the runtime to reflect the new location of the object that's pointed to every time the object is relocated. The physical address pointed to by the interior pointer never remains the same, but it always points to the same object. The other kind is called a *pinning pointer*, which prevents the GC from relocating the object; in other words, it pins the object to a specific physical location in the CLR heap. With some restrictions, conversions are possible between interior, pinning, and native pointers.

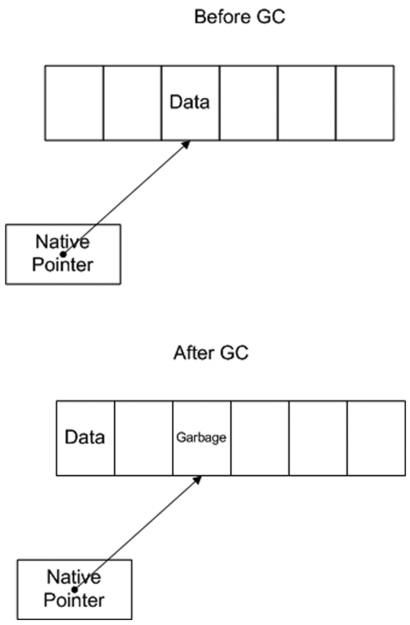


Figure 4.1 Problem using a native pointer with a managed object

Pointers by nature aren't safe, because they allow you to directly manipulate memory. For that reason, using pointers affects the type-safety and verifiability of your code. I strongly urge you to refrain from using CLI pointers in pure-managed applications (those compiled with `/clr:safe` or `/clr:pure`) and to use them strictly to make interop calls more convenient.

4.1.1 Interior pointers

An *interior pointer* is a pointer to a managed object or a member of a managed object that is updated automatically to accommodate for garbage-collection cycles that may result in the pointed-to object being relocated on the CLR heap. You may wonder how that's different from a managed handle or a tracking reference; the difference is that the interior pointer exhibits pointer semantics, and you can perform pointer operations such as pointer arithmetic on it. Although this isn't an exact analogy, think of it like a cell phone. People can call you on your cell phone (which is analogous to an interior pointer) wherever you are, because your number goes with you—the mobile network is constantly updated so that your location is always known. They wouldn't be able to do that with a landline (which is analogous to a native pointer), because a landline's physical location is fixed.

Interior pointer declarations use the same template-like syntax that is used for CLI arrays, as shown here:

```
interior_ptr< type > var = [address];
```

Listing 4.1 shows how an interior pointer gets updated when the object it points to is relocated.

Listing 4.1 Code that shows how an interior pointer is updated by the CLR

```
ref struct CData
{
   int age;
};

int main()
{
   for(int i=0; i<100000; i++)    ◁─❶
      gcnew CData();

   CData^ d = gcnew CData();
   d->age = 100;

   interior_ptr<int> pint = &d->age;    ◁─❷

printf("%p %d\r\n",pint,*pint);

   for(int i=0; i<100000; i++)    ◁─❸
      gcnew CData();

   printf("%p %d\r\n",pint,*pint);    ◁─❹
return 0;
}
```

In the sample code, you create 100,000 orphan `CData` objects ❶ so that you can fill up a good portion of the CLR heap. You then create a `CData` object that's stored in a variable and ❷ an interior pointer to the `int` member `age` of this `CData` object. You then print out the pointer address as well as the `int` value that is pointed to. Now, ❸ you create another 100,000 orphan `CData` objects; somewhere along the line, a garbage-collection cycle occurs (the orphan objects created earlier ❶ get collected because they aren't referenced anywhere). Note that you don't use a `GC::Collect` call because that's not guaranteed to force a garbage-collection cycle. As you've already seen in the discussion of the garbage-collection algorithm in the previous chapter, the GC frees up space by removing the orphan objects so that it can do further allocations. At the end of the code (by which time a garbage collection has occurred), you again ❹ print out the pointer address and the value of `age`. This is the output I got on my machine (note that the addresses will vary from machine to machine, so your output values won't be the same):

```
012CB4C8 100
012A13D0 100
```

As you can see, the address pointed to by the interior pointer has changed. Had this been a native pointer, it would have continued to point to the old address, which may now belong to some other data variable or may contain random data. Thus, using a native pointer to point to a managed object is a disastrous thing to attempt. The compiler won't let you do that: You can't assign the address of a CLI object to a native pointer, and you also can't convert from an interior pointer to a native pointer.

Passing by reference

Assume that you need to write a function that accepts an integer (by reference) and changes that integer using some predefined rule. Here's what such a function looks like when you use an interior pointer as the pass-by-reference argument:

```
void ChangeNumber(interior_ptr<int> num, int constant)
{
   *num += constant * *num;
}
```

And here's how you call the function:

```
CData^ d = gcnew CData();
d->age = 7;
interior_ptr<int> pint = &d->age;
ChangeNumber(pint, 3);
Console::WriteLine(d->age); // outputs 28
```

Because you pass an interior pointer, the original variable (the `age` member of the `CData` object) gets changed. Of course, for this specific scenario, you may as well have used a tracking reference as the first argument of the `ChangeNumber` function; but one advantage of using an interior pointer is that you can also pass a native pointer to the function, because a native pointer implicitly converts to an interior pointer (although the reverse isn't allowed). The following code work:

```
int number = 8;                       ❶ Pass native pointer to function
ChangeNumber(&number, 3);   <-┘
Console::WriteLine(number); // outputs 32
```

It's imperative that you remember this. You can pass a native pointer to function that expects an interior pointer as you do here ❶, because there is an implicit conversion from the interior pointer to the native pointer. But you can't pass an interior pointer to a native pointer; if you try that, you'll get a compiler error. Because native pointers convert to interior pointers, you should be aware that an interior pointer need not necessarily always point to the CLR heap: If it contains a converted native pointer, it's then pointing to the native C++ heap. Next, you'll see how interior pointers can be used in pointer arithmetic (something that can't be done with a tracking reference).

Pointer arithmetic

Interior pointers (like native pointers) support pointer arithmetic; thus, you may want to optimize a performance-sensitive piece of code by using direct pointer arithmetic on some data. Here's an example of a function that uses pointer arithmetic on an interior pointer to quickly sum the contents of an array of `int`s:

```
int SumArray(array<int>^% intarr)
{
    int sum = 0;                                  ❶ Get interior
    interior_ptr<int> p = &intarr[0];    <-┘   pointer to array
    while(p != &intarr[0]+ intarr->Length)   ❷ Iterate
        sum += *p++;                              through array
    return sum;
}
```

In this code, `p` is an interior pointer to the array ❶ (the address of the first element of the array is also the address of the array). You don't need to worry about the GC relocating the array in the CLR heap. You iterate through the array by using the `++` operator on the interior pointer ❷, and you add each element to the variable `sum` as you do so. This way, you avoid the overhead of going through the `System::Array` interface to access each array element.

It's not just arrays that can be manipulated using an interior pointer. Here's another example of using an interior pointer to manipulate the contents of a `System::String` object:

```
String^ str = "Nish wrote this book for Manning Publishing";
interior_ptr<Char> ptxt = const_cast< interior_ptr<Char> >(
    PtrToStringChars(str));       ◁─❶
interior_ptr<Char> ptxtorig = ptxt;   ◁─❷
while((*ptxt++)++);       ◁─❸
Console::WriteLine(str);       ◁─❹
while((*ptxtorig++)--);   ◁─❺
Console::WriteLine(str);       ◁─❻
```

You use the `PtrToStringChars` helper function ❶ to get an interior pointer to the underlying string buffer of a `System::String` object. The `PtrToStringChars` function is a helper function declared in `<vcclr.h>` that returns a `const` interior pointer to the first character of a `System::String`. Because it returns a `const` interior pointer, you have to use `const_cast` to convert it to a non-`const` pointer. You go through the string using a `while`-loop ❸ that increments the pointer as well as each character until a `nullptr` is encountered, because the underlying buffer of a `String` object is always `nullptr`-terminated. Next, when you use `Console::WriteLine` on the `String` object ❹, you can see that the string has changed to

```
Ojti!xspuf!uijt!cppl!gps!Nboojoh!Qvcmjtijoh
```

You've achieved encryption! (Just kidding.) Because you saved the original pointer in `ptxtorig` ❷, you can use it to convert the string back to its original form using another `while` loop. The second `while` loop ❺ increments the pointer but decrements each character until it reaches the end of the string (determined by the `nullptr`). Now, ❻ when you do a `Console::WriteLine`, you get the original string:

```
Nish wrote this book for Manning Publishing
```

Whenever you use an interior pointer, it's represented as a managed pointer in the generated MSIL. To distinguish it from a reference (which is also represented as a managed pointer in IL), a `modopt` of type `IsExplicitlyDereferenced` is emitted by the compiler. A `modopt` is an optional modifier that can be applied to a type's signature. Another interesting point in connection with interior pointers is that the `this` pointer of an instance of a value type is a non-`const` interior pointer to the type. Look at the value class shown here, which obtains an interior pointer to the class by assigning it to the `this` pointer:

```
value class V
{
    void Func()
    {
        interior_ptr<V> pV1 = this;
        //V* pV2 = this; <-- this won't compile
    }
};
```

As is obvious, in a value class, if you need to get a pointer to `this`, you should use an interior pointer, because the compiler won't allow you to use a native pointer. If you specifically need a native pointer to a value object that's on the managed heap, you have to pin the object using a pinning pointer and then assign it to the native pointer. We haven't discussed pinning pointers yet, but that's what we'll talk about in the next section.

A dangerous side-effect of using interior pointers to manipulate String objects

The CLR performs something called *string interning* on managed strings, so that multiple variables or literal occurrences of the same textual string always refer to a single instance of the `System::String` object. This is possible because `System::String` is immutable—the moment you change one of those variables, you change the reference, which now refers to a new `String` object (quite possibly another interned string). All this is fine as long as the strings are immutable. But when you use an interior or pinning pointer to directly access and change the underlying character array, you break the immutability of `String` objects. Here's some code that demonstrates what can go wrong:

```
String^ s1 = "Nishant Sivakumar";
String^ s2 = "Nishant Sivakumar";

interior_ptr<Char> p1 = const_cast<interior_ptr<Char> >(
    PtrToStringChars(s1));  // Get a pointer to s1
while(*p1)  // Change s1 through pointer p1
    (*p1++) = 'X';

Console::WriteLine("s1 = {0}\r\ns2 = {1}",s1,s2);
```

The output of is as follows:

```
s1 = XXXXXXXXXXXXXXXXX
s2 = XXXXXXXXXXXXXXXXX
```

You only changed one string, but both strings are changed. If you don't understand what's happening, this can be incredibly puzzling. You have two `String` handle variables, `s1` and `s2`, both containing the same string literal. You get an interior pointer `p1` to the string `s1` and change each character in `s1` to *X* (basically blanking out the string with the character *X*). Common logic would say that you have changed the string `s1`, and that's that. But because of string interning, `s1` and `s2` were both handles to the same `String` object on the CLR heap. When you change the underlying buffer of the string `s1` through the interior pointer, you change the interned string. This means any string handle to that `String` object now points to an entirely different string (the *X*-string in this case). The output of the `Console::WriteLine` should now make sense to you.

In this case, figuring out the problem was easy, because both string handles were in the same block of code, but the CLR performs string interning across application domains. This means changing an interned string can result in extremely hard-to-debug errors in totally disconnected parts of your application. My recommendation is to try to avoid directly changing a string through a pointer, except when you're sure you won't cause havoc in other parts of the code. Note that it's safe to read a string through a pointer; it's only dangerous when you change it, because you break the "strings are immutable" rule of the CLR. Alternatively, you can use the `String::IsInterned` function to determine if a specific string is interned, and change it only if it isn't an interned string.

4.1.2 Pinning pointers

As we discussed in the previous section, the GC moves CLI objects around the CLR heap during garbage-collection cycles and during heap-compaction operations. Native pointers don't work with CLI objects, for reasons previously mentioned. This is why we have interior pointers, which are self-adjusting pointers that update themselves to always refer to the same object, irrespective of where the object is located in the CLR heap. Although this is convenient when you need pointer access to CLI objects, it only works from managed code. If you need to pass a pointer to a CLI object to a native function (which runs outside the CLR), you can't pass an interior pointer, because the native function doesn't know what an interior pointer is, and an interior pointer can't convert to a native pointer. That's where pinning pointers come into play.

A pinning pointer pins a CLI object on the CLR heap; as long as the pinning pointer is *alive* (meaning it hasn't gone out of scope), the object remains pinned. The GC knows about pinned objects and won't relocate pinned objects. To continue the phone analogy, imagine a pinned pointer as being similar to your being forced to remain stationary (analogous to being pinned). Although you have a cell phone, your location is fixed; it's almost as if you had a fixed landline.

Because pinned objects don't move around, it's legal to convert a pinned pointer to a native pointer that can be passed to the native caller that's running outside the control of the CLR. The word *pinning* or *pinned* is a good choice; try to visualize an object that's pinned to a memory address, just like you pin a sticky note to your cubicle's side-board.

The syntax used for a pinning pointer is similar to that used for an interior pointer:

```
pin_ptr< type > var = [address];
```

The duration of pinning is the lifetime of the pinning pointer. As long as the pinning pointer is in scope and pointing to an object, that object remains pinned. If the pinning pointer is set to `nullptr`, then the object isn't pinned any longer; or if the pinning pointer is set to another object, the new object becomes pinned and the previous object isn't pinned any more.

Listing 4.2 demonstrates the difference between interior and pinning pointers. To simulate a real-world scenario within a short code snippet, I uses `for` loops to create a large number of objects to bring the GC into play.

Listing 4.2 Code that compares an interior pointer with a pinning pointer

```
for(int i=0; i<100000; i++)          | Fill portion
   gcnew CData();                    | of CLR heap

CData^ d1 = gcnew CData();         |
for(int i=0; i<1000; i++)          | 1
   gcnew CData();                  |
CData^ d2 = gcnew CData();         |

interior_ptr<int> intptr = &d1->age;    <- 2
pin_ptr<int> pinptr = &d2->age;    <- 3

printf("intptr=%p pinptr=%p\r\n",    <- Display pointer
   intptr,pinptr);                      addresses before GC

for(int i=0; i<100000; i++)          | 4
   gcnew CData();                    |

printf("intptr=%p pinptr=%p\r\n",    <- Display pointer
   intptr,pinptr);                      addresses after GC
```

In the code, you create two `CData` objects with a gap in between them ❶ and associate one of them with an interior pointer to the `age` member of the first object ❷.

The other is associated with a pinning pointer to the `age` member of the second object ❸. By creating a large number of orphan objects, you force a garbage-collection cycle ❹ (again, note that calling `GC::Collect` may not always force a garbage-collection cycle; you need to fill up a generation before a garbage-collection cycle will occur). The output I got was

```
intptr=012CB4C8 pinptr=012CE3B4
intptr=012A13D0 pinptr=012CE3B4
```

Your pointer addresses will be different, but after the garbage-collection cycle, you'll find that the address held by the pinned pointer (`pinptr`) has not changed, although the interior pointer (`intptr`) has changed. This is because the CLR and the GC see that the object is pinned and leave it alone (meaning it doesn't get relocated on the CLR heap). This is why you can pass a pinned pointer to native code (because you know that it won't be moved around).

Passing to native code

The fact that a pinning pointer always points to the same object (because the object is in a pinned state) allows the compiler to provide an implicit conversion from a pinning pointer to a native pointer. Thus, you can pass a pinning pointer to any native function that expects a native pointer, provided the pointers are of the same type. Obviously, you can't pass a pinning pointer to a `float` to a function expecting a native pointer to a `char`. Look at the following native function that accepts a `wchar_t*` and returns the number of vowels in the string pointed to by the `wchar_t*`:

```
#pragma unmanaged
int NativeCountVowels(wchar_t* pString)
{
   int count = 0;
   const wchar_t* vowarr = L"aeiouAEIOU";
   while(*pString)
      if(wcschr(vowarr,*pString++))
         count++;
   return count;
}
#pragma managed
```

Here's how you pass a pointer to a CLI object, after first pinning it, to the native function just defined:

```
String^ s = "Most people don't know that the CLR is written in C++";
pin_ptr<Char> p = const_cast< interior_ptr<Char> >(
   PtrToStringChars(s));
Console::WriteLine(NativeCountVowels(p));
```

#pragma managed/unmanaged

These are `#pragma` compiler directives that give you function-level control for compiling functions as managed or unmanaged. If you specify that a function is to be compiled as unmanaged, native code is generated, and the code is executed outside the CLR. If you specify a function as managed (which is the default), MSIL is generated, and the code executes within the CLR. Note that if you have an unmanaged function that you've marked as unmanaged, you should remember to re-enable managed compilation at the end of the function

`PtrToStringChars` returns a `const` interior pointer, which you cast to a non-`const` interior pointer; this is implicitly converted to a pinning pointer. You pass this pinning pointer, which implicitly converts to a native pointer, to the `NativeCountVowels` function. The ability to pass a pinning pointer to a function that expects a native pointer is extremely handy in mixed-mode programming, because it gives you an easy mechanism to pass pointers to objects on the CLR heap to native functions. Figure 4.2 illustrates the various pointer conversions that are available.

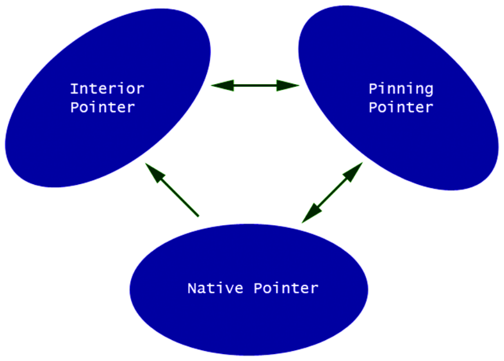


Figure 4.2 Pointer conversions between native, interior, and pinning pointers

As you can see in the figure, the only pointer conversion that is illegal is that from an interior pointer to a native pointer; every other conversion is allowed and implicitly done. You have seen how pinning pointers make it convenient for you to pass pointers to CLI objects to unmanaged code. I now have to warn you that pinning pointers should be used only when they're necessary, because tactless usage of pinning pointers results in what is called the *heap fragmentation problem*.

The heap fragmentation problem

Objects are always allocated sequentially in the CLR heap. Whenever a garbage collection occurs, orphan objects are removed, and the heap is compacted so it won't remain in a fragmented condition. (We covered this in the previous chapter when we discussed the multigenerational garbage-collection algorithm used by the CLR.) Let's assume that memory is allocated from a simple heap that looks like figures 4.3 through 4.6. Of course, this is a simplistic representation of the CLR's GC-based memory model, which involves a more complex algorithm. But the basic principle behind the heap fragmentation issue remains the same, and thus this simpler model will suffice for the present discussion. Figure 4.3 depicts the status of the heap before a garbage-collection cycle occurs.

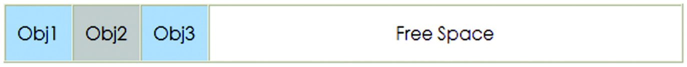


Figure 4.3 Before a garbage-collection cycle

There are presently three objects in the heap. Assume that `Obj2` (with the gray shaded background) is an orphan object, which means it will be cleaned up during the next garbage-collection cycle. Figure 4.4 shows what the heap looks like after the garbage-collection cycle.

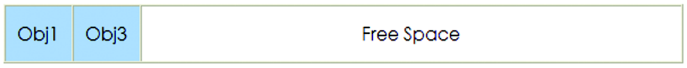


Figure 4.4 After a garbage-collection cycle (assuming no pinned objects)

The orphan object has been removed and a heap compaction has been performed, so `Obj1` and `Obj3` are now next to each other. The idea is to maximize the free space available in the heap and to put that free space in a single contiguous block of memory. Figure 4.5 shows what the heap would look like if there was a pinned object during the garbage-collection cycle.

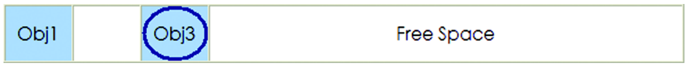


Figure 4.5 After a garbage-collection cycle (one pinned object)

Assume that `Obj3` is a pinned object (the circle represents the pinning). Because the GC won't move pinned objects, `Obj3` remains where it was. This results in fragmentation because the space between `Obj1` and `Obj2` cannot be added to the large continuous free block of memory. In this particular case, it's just a small gap that would have contained only a single object, and thus isn't a major issue. Now, assume that several pinned objects exist on the CLR heap when the garbage-collection cycle occurs. Figure 4.6 shows what happens in such a situation.

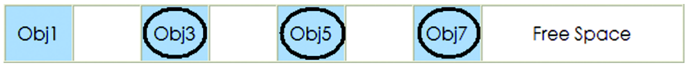


Figure 4.6 After a garbage-collection cycle (several pinned objects)

None of those pinned objects can be relocated. This means the compaction process can't be effectively implemented. When there are several such pinned objects, the heap is severely fragmented, resulting in slower and less-efficient memory allocation for new objects. This is the case because the GC has to try that much harder to find a block that's large enough to fit the requested object. Sometimes, although the total free space is bigger than the requested memory, the fact that there is no single continuous block of memory large enough to hold that object results in an unnecessary garbage-collection cycle or a memory exception. Obviously, this isn't an efficient scenario, and it's why you have to be extremely cautious when you use pinning pointers.

Recommendations for using pinning pointers

Now that you've seen where pinning pointers can be handy and where they can be a little dodgy, I'm going to give you some general tips on effectively using pinning pointers.

Unless you absolutely have to, don't use a pinning pointer! Whenever you think you need to use a pinning pointer, see if an interior pointer or a tracking reference may be a better option. If an interior pointer is acceptable as an alternative, chances are good that this is an improper place for using a pinning pointer.

If you need to pin multiple objects, try to allocate those objects together so that they're in an adjacent area in the CLR heap. That way, when you pin them, those pinned objects will be in a contiguous area of the heap. This reduces fragmentation compared to their being spread around the heap.

When making a call into native code, check to see if the CLR marshalling layer (or the target native code) does any pinning for you. If it does, you don't need to pin your object before passing it, because you'd be writing unnecessary (though harmless) code by adding an extra pinning pointer to the pinned object (which doesn't do anything to the pinned state of the object).

Newly-allocated objects are put into Generation-0 of the CLR heap. You know that garbage-collection cycles happen most frequently in the Generation-0 heap. Consequently, you should try to avoid pinning recently allocated objects; chances are that a garbage-collection cycle will occur while the object is still pinned.

Reduce the lifetime of a pinning pointer. The longer it stays in scope, the longer the object it points to remains pinned and the greater the chances of heap fragmentation. For instance, if you need a pinning pointer inside an `if` block, declare it inside the `if` block so the pinning ends when the `if` block exits.

Whenever you pass a pinning pointer to a native pointer, you have to ensure that the native pointer is used only if the pinning pointer is still alive. If the pinning pointer goes out of scope, the object becomes unpinned. Now it can be moved around by the GC. Once that happens, the native pointer is pointing to some random location on the CLR heap. I've heard the term *GC hole* used to refer to such a scenario, and it can be a tough debugging problem. Although it may sound like an unlikely contingency, think of what may happen if a native function that accepts a native pointer stores this pointer for later use. The caller code may have passed a pinning pointer to this function. Once the function has returned, the pinning will quickly stop, because the original pinning pointer won't be alive much longer. However, the saved pointer may be used later by some other function in the native code, which may result in some disastrous conditions (because the location the pointer points to may contain some other object now or even be free space). The best you can do is to know what the native code is going to do with a pointer before you pass a pinning pointer to it. That way, if you see that there is the risk of a GC hole, you avoid calling that function and try to find an alternate solution.

Note that these are general guidelines and not hard rules to be blindly followed at all times. It's good to have some basic strategies and to understand the exact consequences of what happens when you inappropriately use pinning pointers. Eventually, you have to evaluate your coding scenario and use your judgment to decide on the best course.

4.2 Working with interop mechanisms

The term *interop* (short for *interoperability*) is used to represent any situation where managed and unmanaged code have to interact with each other; it includes calling managed code from unmanaged code, as well as the reverse. C++/CLI provides the same mechanisms for interop that are available in other languages like C#, such as CCW and P/Invoke. In addition, C++/CLI also provides a mechanism called *C++ interop*, which allows you to use a mixed-mode executable or DLL to handle interop scenarios. For most purposes, C++ interop is a lot more convenient, flexible and performant than the other interop mechanisms. Throughout the rest of this book, we'll use C++ interop for the mixed-mode programming ventures.

In this section, we'll look at all three of the mechanisms available in VC++ 2005 to interop between managed and unmanaged code, and compare them in terms of ease of use and performance. When we discuss CCW and P/Invoke, you'll see why you may prefer C++ interop over them whenever that is an option. Of course, you should also be aware that both CCW and P/Invoke are powerful mechanisms that are handy for specific situations where C++ interop may not be the best option.

To simplify the discussion, I have sectioned it into the two potential interop situations: accessing a managed library from native code and accessing a native library from managed code. For most real-life applications that require some sort of interop, you'll encounter at least one of these situations, and quite possibly both.

4.2.1 Accessing a managed library from native code

You'll encounter scenarios where you need to access a managed library from native code, typically when you have an existing unmanaged application and, for whatever reasons, you need to interop with a new managed library as part of enhancing or revising your application. This is analogous to trying to play a DVD on an old VCR. You need to convert the DVD data (analogous to the managed library) to a format that can be played by the VCR (analogous to your native code).

In this section, you'll see how to do this using two different techniques: The first one will use a CCW, and the second one will use C++ interop. You'll use a simple managed library that lets you order a movie from a fictitious video store. Listing 4.3 shows the interface and skeletal class implementation for such a library.

Listing 4.3 The managed library implementation for the video store

```
namespace ManLib
{
    public interface class IVideoStore
    {
        bool IsMovieAvailable(String^);
        int GetMovieID(String^);
        int GetStockCount(int);
        bool OrderMovie(int, int);
        array<String^>^ GetMovieCast(int);

    };

    public ref class VideoStore : IVideoStore
    {
    public:
        virtual bool IsMovieAvailable(String^ strMovieName)
        . . .
        virtual int GetMovieID(String^ strMovieName)
        . . .
        virtual int GetStockCount(int movieID)
        . . .
        virtual bool OrderMovie(int movieID, int count)
        . . .
        virtual array<String^>^ GetMovieCast(int movieID)
        . . .
    };
}
```

The method names are self-explanatory, so I won't attempt to describe what each method does. Next, you're going to write two different applications. One uses a CCW and the other uses C++ interop, but both applications are functionally identical. They both check if a movie is available; after checking to see if the store has it, you'll order six copies of it. The applications also display a list of the cast of characters once the order is completed. All this is done through the managed library you just defined. Let's start with the CCW caller application.

Using a CCW to access a CLI library

CCW is a mechanism available in .NET that allows a COM client (such as a native C++ or VB 6 application) to access managed objects via proxy COM objects which wrap the managed objects. Figure 4.7 shows a diagrammatic representation of how the CCW connects a CLI library with a native caller.

The CCW is responsible for all managed/unmanaged marshalling. The managed object is virtually invisible to the COM client, which sees only the CCW. Using the DVD-VCR analogy, a CCW is like an external device that reads a DVD and passes converted data to a dummy video cassette disk that can be played on a VCR.

The CCW is reference-counted (like a normal COM object) and manages the lifetime of the wrapped managed object. When its reference count reaches zero, the managed object becomes a candidate for garbage collection. A deeper explanation of CCW or COM isn't within the scope or subject matter of this book, and unless you have some basic COM awareness, the code in this section may look a little strange. You should, however, still go through it, so that when you see the C++ interop version of the same app, you'll appreciate how much simpler and more convenient C++ interop is. In general, unless you've previously done a bit of COM programming, CCW isn't something I recommend as a suitable interop mechanism. With that in mind, let's get started on the CCW caller application.

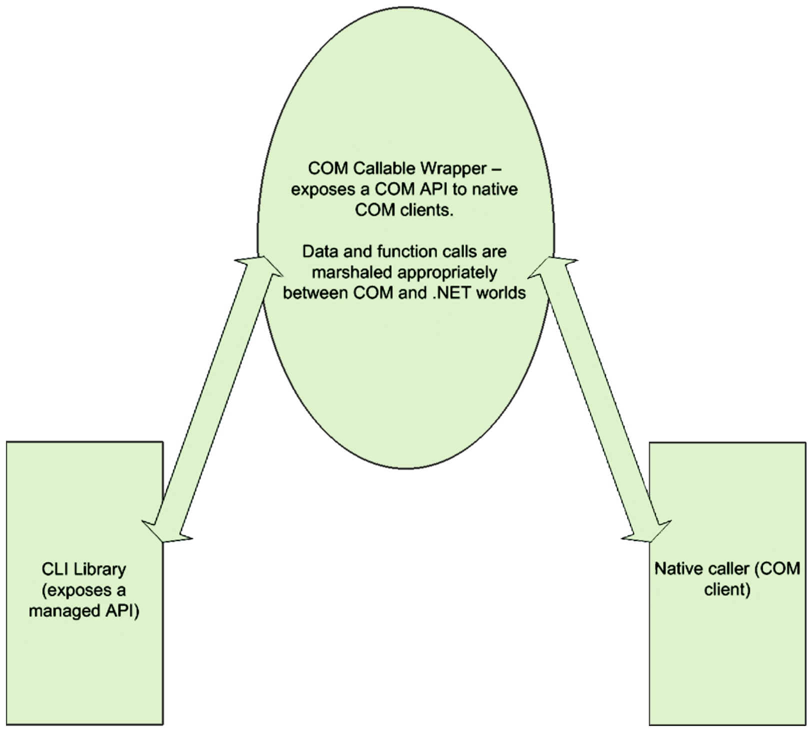

COM Callable Wrapper – exposes a COM API to native COM clients.
Data and function calls are marshaled appropriately between COM and .NET worlds
CLI Library (exposes a managed API)
Native caller (COM client)

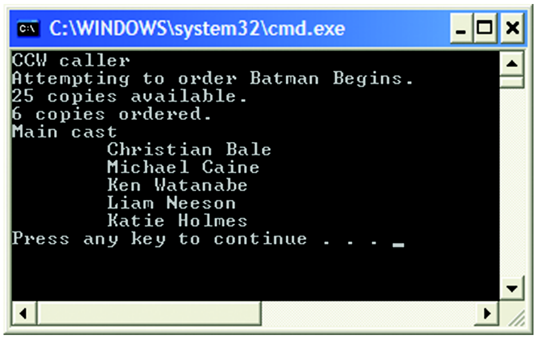
Figure 4.7 Native caller using a CCW to call into a CLI library

The managed assembly has to be registered for use by COM clients. For this, you use Regasm.exe, the Assembly Registration Tool provided with the .NET Framework. You can run the following command to register the assembly for COM access:

```
RegAsm.exe  ManLib.dll /tlb:ManLib.tlb
```

You use the `/tlb` option to output a type library file that will contain the accessible COM types within that library. You do that so you can import type information into the application by using `#import` on this tlb file. Note that only public members of public types are exposed via COM, and any other members or types will be invisible to the CCW. You can also control COM visibility of an assembly or a type at various levels of granularity using custom attributes such as `ComVisible`.

Before I show you the code for the application, look at the output you'll get when you run the final CCW application, shown in figure 4.8. You'll then know exactly what is being done as you examine the code.



```
CCW caller
Attempting to order Batman Begins.
25 copies available.
6 copies ordered.
Main cast
        Christian Bale
        Michael Caine
        Ken Watanabe
        Liam Neeson
        Katie Holmes
Press any key to continue . . . _
```

Figure 4.8
Output from the CCW caller application

Remember that the CCW caller is a native application and knows nothing of .NET or of managed types. Listing 4.4 shows the CCW caller application code.

Listing 4.4 Accessing the managed library via a CCW

```
#import "..\debug\ManLib.tlb" raw_interfaces_only      <-- Reference tlb using #import

using namespace ManLib;

void BuyMovie()
{
    cout << "CCW caller" << endl;
    ManLib::IVideoStorePtr pVidStore(__uuidof(VideoStore));   <-- Instantiates VideoStore object

    unsigned char ret = 0;
    CComBSTR strMovie = "Batman Begins";

    cout << "Attempting to order Batman Begins." << endl;
    if(SUCCEEDED(pVidStore->IsMovieAvailable(      <-- Check availability
        strMovie, &ret)) && ret)
    {
        long movieid = 0;
        if(SUCCEEDED(pVidStore->GetMovieID(strMovie, &movieid)))   <-- Retrieves movie ID
        {
            long count = 0;
            if(SUCCEEDED(pVidStore->GetStockCount(     <-- Returns number of copies
                movieid, &count)) && count > 5)
            {
                cout << count << " copies available." << endl;
                if(SUCCEEDED(pVidStore->OrderMovie(    <-- Order movie
                    movieid, 6, &ret)) && ret)
                {
                    cout << "6 copies ordered." << endl;
                    SAFEARRAY* pSA= NULL;
                    if(SUCCEEDED(pVidStore->GetMovieCast(   <-- Get cast list
                        movieid, &pSA)))
                        {
                            long lbound = 0, ubound = 0;

                            SafeArrayGetLBound (pSA, 1, &lbound);
                            SafeArrayGetUBound (pSA, 1, &ubound);

                            BSTR* s = NULL;
                            SafeArrayAccessData(pSA, (void**) &s);
                            cout << "Main cast" << endl;
                            for(long i=lbound; i<=ubound; i++)
                            {
                                wcout << "\t" << s[i] << endl;
                            }
                            SafeArrayUnaccessData(pSA);
                            SafeArrayDestroy (pSA);
                        }
                    }
                }
            }
        }
    }
```

Enumerates returned SAFEARRAY

The most noticeable thing about this code listing is that it's essentially COM-based in nature. There is no .NET presence syntactically in that code, and that code is only aware that it's accessing a COM object and its methods. The fact that the COM object is a CCW for an underlying managed object is imperceptible to the caller application. When you use `#import`, the compiler creates the required smart pointers to access the COM object, which saves you the trouble of manually having to call COM functions such as `AddRef`, `Release`, `QueryInterface`, and so on.

The .NET interface is now accessed via the CCW interface, which means the prototypes for the managed functions have been replaced with COM versions. As is typical in COM, all functions return an `HRESULT`. The return type in the managed code is converted to an `out` parameter of the function. For example, this is what the `IsMovieAvailable` function looks like in managed code:

```
bool IsMovieAvailable(String^);
```

And here's the COM version used in the CCW caller:

```
HRESULT IsMovieAvailable (/*[in]*/ BSTR,
    /*[out,retval]*/ unsigned char *);
```

The `String^` argument in managed code becomes a `BSTR` argument in COM code, and the `bool` return type, which is now an `out` parameter of the function, becomes an `unsigned char*`. Similarly, every other managed function has been given a COM version which uses COM types.

In the case of the GetMovieCast function, the COM version can be a little formidable for people without COM experience: It uses a SAFEARRAY, which isn't the simplest of the COM types. The original managed function, which looked like this

```
array<String^>^ GetMovieCast(int);
```

has been converted to the following COM version:

```
HRESULT GetMovieCast (/*[in]*/ long, /*[out,retval]*/ SAFEARRAY**);
```

Although the CLR marshalling layer takes care of type conversions between the managed and the COM world, the caller is responsible for freeing native resources once they're no longer required. You'll notice how you use SafeArrayDestroy to free the SAFEARRAY once you finish iterating through its contents. You can simplify the SAFEARRAY usage considerably by using the ATL CComSafeArray wrapper class, but I wanted to use straight COM to exemplify the differences between the CCW approach and the C++/CLI approach that will be discussed in the next section.

As I said, unless you're reasonably comfortable with COM and prepared to be responsible for native resource deallocation (or be familiar with using ATL wrapper classes), CCW can be intimidating. Of course, if your native application is COM-based and already makes extensive use of COM, CCW is the most natural choice and will merge smoothly with the rest of your native code. If not, then I strongly recommend that you don't use CCW; instead, you should use C++ interop. Moving on to the C++ interop version of the app you wrote previously, you'll appreciate how convenient it can be compared to CCW.

Preserving method signature for CCW

In the example, the return type of the managed method is converted to an out parameter in the COM method, and the return type is changed to an HRESULT. This can be inconvenient at times, because you need to check the HRESULT for success. You also need to pass a pointer to the original return type as a final argument. If you have control over the managed library and have the option of modifying the original managed code, you can use the MethodImplAttribute attribute with MethodImplOptions::PreserveSig on those methods where you want the signature to be preserved. For example, if you applied it to the IsMovieAvailable method, you'd get the following COM method: unsigned char IsMovieAvailable([in] BSTR), which is a little closer to the original managed method. You can also directly use the return type instead of having to pass a pointer to a variable.

Using C++ interop to access a CLI library

The C++ interop caller app is functionally identical to the CCW caller app. However, although the CCW caller was a native C++ application, the C++ interop caller has to be compiled with the `/clr` compilation switch—if not the whole application, then at least those source files that will access the managed library. You also need to add a reference to the managed library in the project settings. Figure 4.9 shows a diagrammatic view of a mixed-mode application accessing a CLI library via C++ interop. Using the DVD-VCR analogy, C++ interop is analogous to directly adding support for playing a DVD to your old VCR by adding a DVD player to the device (analogous to enabling `/clr` compilation).

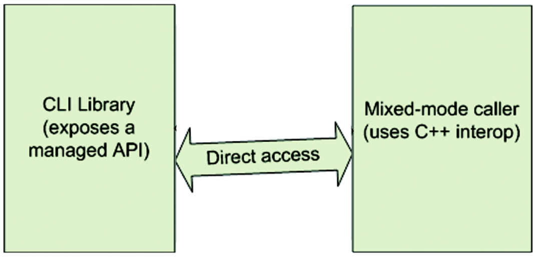
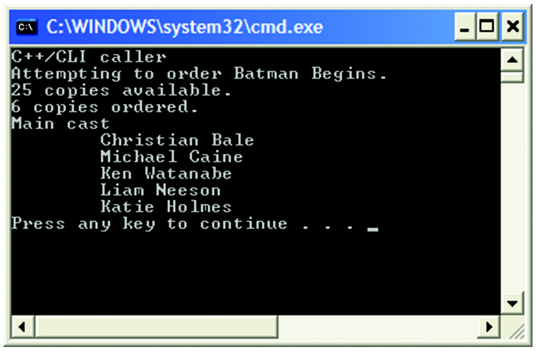

CLI Library (exposes a managed API)
Direct access
Mixed-mode caller (uses C++ interop)

Figure 4.9 Accessing a CLI library using C++ interop



```
C++/CLI caller
Attempting to order Batman Begins.
25 copies available.
6 copies ordered.
Main cast
        Christian Bale
        Michael Caine
        Ken Watanabe
        Liam Neeson
        Katie Holmes
Press any key to continue . . . _
```

Figure 4.10 Output from the C++ interop caller application

As you did with the CCW, look at the output you'll get when you run the final C++ interop application; see figure 4.10.

Listing 4.5 shows the code for the C++ interop version of the application. Not only is it shorter than the CCW version, but it directly uses CLI types to access the managed library.

Listing 4.5 Accessing the managed library via C++ interop

```
void BuyMovie()
{
    cout << "C++/CLI caller" << endl;
    VideoStore^ vidstore = gcnew VideoStore();
    cout << "Attempting to order Batman Begins." << endl;
    System::String^ strMovie = "Batman Begins";
    if(vidstore->IsMovieAvailable(strMovie))
    {
        int movieid = vidstore->GetMovieID(strMovie);
        int count = 0;
```

Instantiate VideoStore object

Invoke VideoStore methods

```
        if((count = vidstore->GetStockCount(movieid)) > 5)
        {
            cout << count << " copies available." << endl;
            if(vidstore->OrderMovie(movieid, 6))
            {
                cout << "6 copies ordered." << endl;
                cout << "Main cast" << endl;
                for each(System::String^ s in
                    vidstore->GetMovieCast(movieid))       <- 1 Enumerate String array
                {
                    System::Console::WriteLine("\t{0}", s);
                }
            }                                              Invoke VideoStore methods
        }
    }
}
```

As you can see, this is conventional C++/CLI code, except that it's part of a primarily unmanaged application that has had `/clr` enabled. In other words, you're seeing a mixed-mode application, albeit a simple one. You directly use CLI types like `String^` and CLI arrays, and thus there is no need to convert managed types to native types. In a real-life scenario, unless the original native application is COM-based, the mere fact that you didn't have to go through the exacting chore of enumerating a `SAFEARRAY` and can instead have the luxury of using `for each` (1) on the returned `String^` array should itself be reason enough to choose C++ interop over CCW in such a situation. Although I am firmly in favor of C++ interop over CCW, it is imprudent to take an uncompromising approach to all interop scenarios. There are situations where CCW is best and other situations (the majority, in my opinion) where C++ interop is more suitable. In the next section, we'll compare the two mechanisms to see where each is the better choice.

Comparison of the two techniques

You've seen two different techniques to access a managed library from a native application. One uses the CCW mechanism supported by the .NET Framework, and the other uses C++ interop supported by the C++/CLI compiler. The CCW caller is typically a native application (although you can do CCW from a mixed-mode app too), whereas the C++ interop caller has to enable `/clr` and is always a mixed-mode application. If you aren't prepared to use a mixed-mode caller application, CCW is your better choice, because C++ interop requires mixed-mode compilation.

If you aren't familiar with COM, or if your caller app isn't primarily COM-based, using CCW has the obvious disadvantage that you'll be forced to do quite a bit of COM programming. In most cases, you'll be responsible for freeing COM resources such as `SAFEARRAY`s and `BSTR`s. And unless you can apply the `MethodImplAttribute` with the `PreserveSig` option on your managed methods (which requires that you have source code access to the managed library), you'll also have to handle `out` parameters for your return types because the COM methods will return `HRESULT`s. If you use C++ interop, you'll directly use C++/CLI to access the managed library using .NET types.

Another issue with using CCW is that CCW lets you instantiate only those objects that have a default constructor. If an object has only nondefault constructors, you can't use that object through CCW. You can write another factory class that will instantiate and return an object of the type you want, but doing so adds unnecessary work and is possible only in scenarios where you can change the managed library. There are no such issues with C++ interop.

The last issue is that of performance. Although the performance improvement will vary depending on the scenario where interop is applied, typically C++ interop is much more performant than CCW. With CCW, the managed/unmanaged transitions are handled by the CCW marshalling layer. With C++ interop, you have more direct control over what you do, because you can choose to use managed types by default and convert to native types only where required.

Summarizing these points, my recommendation is to use CCW only when you have to, such as when the native app is COM-based and you don't want to bring in `/clr` compilation. For all other scenarios, C++ interop is your fastest and most convenient option. In later chapters, you'll see how you can take advantage of C++ interop when you mix managed technologies such as Windows Forms, Avalon, and so on, with native technologies like MFC. Now that you've seen how to access managed libraries from native code, let's look at the reverse process in the next section—accessing native libraries from managed code.

4.2.2 Accessing a native library from managed code

In this section, we'll look at the reverse scenario of what we covered in the previous section: accessing a native library from managed code. For example, you may want to use a native library that you have been using for years from your new managed applications, because you can't find a managed equivalent for it that's efficient enough for your purposes. Think of the DVD-VCR analogy, and swap the situation; you have a video tape, and you want to play it on your DVD player. You can approach this in two ways—one is to use the P/Invoke mechanism, and the other is to use C++ interop. We'll cover both mechanisms in this section, and we'll use the following library as the example native code:

```
class __declspec(dllexport) CStudentDB
{
public:
    CStudentDB(int id);                                  ❶ Exported
    int GetScore();                                         class
};

extern "C"  __declspec(dllexport) int GetStudentID(
    LPCTSTR strName);
extern "C" __declspec(dllexport) void GetStudentAddress(      ❷ Exported
    int id, LPTSTR szAddress);                                   functions
```

This code is part of a native DLL. There are two exported functions ❷, as well as an exported class ❶. You're going to write two sample programs, one using P/Invoke and the other using C++ interop, both of which will access this native DLL. Let's start with the P/Invoke sample.

Using P/Invoke to access a native DLL

P/Invoke (short for Platform Invoke) is a mechanism that allows you to call unmanaged functions declared in native DLLs from managed code by declaring those functions using the `DllImport` attribute. This is analogous to using a device that converts the video cassette data to a video stream that can be directly passed to the DVD player. The .NET P/Invoke layer does the type marshalling across the managed/unmanaged boundary. When you make a P/Invoke function call, the P/Invoke layer loads the containing DLL into the process memory (if it's not already loaded), gets the address of the unmanaged function to call, converts the managed types to native types as required, calls the function, converts the native types back to managed types as required, and returns control to the calling managed code. Figure 4.11 shows how a managed application accesses a native C-based API through P/Invoke.

All this is done transparently. As a user, it's convenient to call an unmanaged function as if it was a managed call, but you pay a price for all that convenience—performance. The managed/unmanaged transitions and data conversions slow down your application, although depending on your scenario, that may not be a big issue. Listing 4.6 shows the P/Invoke version of the program that calls into the native DLL you declared earlier.

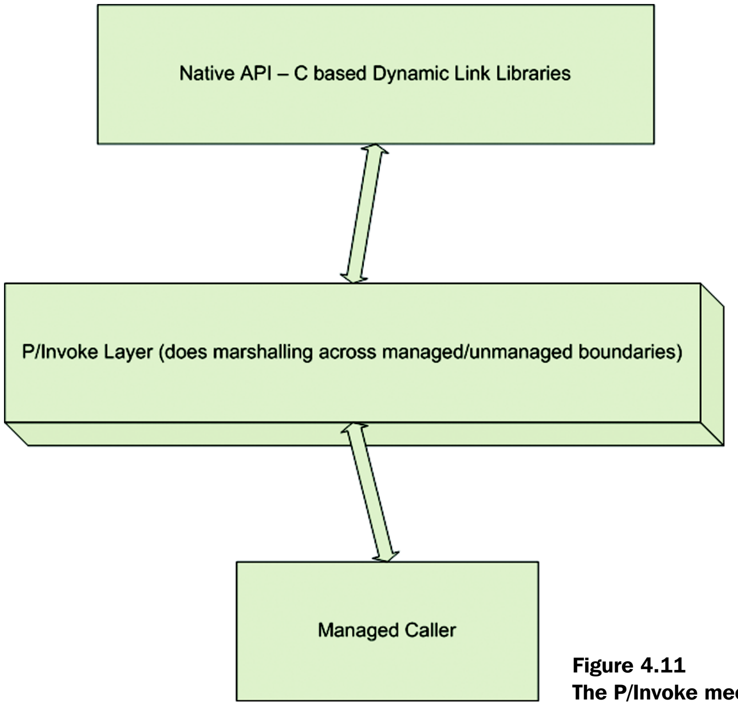

Native API – C based Dynamic Link Libraries
P/Invoke Layer (does marshalling across managed/unmanaged boundaries)
Managed Caller

Figure 4.11
The P/Invoke mechanism

Listing 4.6 Using P/Invoke to call functions in a native DLL

```
[DllImport("Natlib.dll",CharSet=CharSet::Unicode)]
extern "C" int GetStudentID(String^ strName);            ❶ P/Invoke declaration
                                                           for GetStudentID
[DllImport("Natlib.dll",CharSet=CharSet::Unicode)]
extern "C" void GetStudentAddress(int id,                 ❷ P/Invoke declaration
    StringBuilder^ szAddress);                             for GetStudentAddress

int main(array<System::String ^> ^args)
{
    int studid = GetStudentID("Howard Jones");
    StringBuilder^ addr = gcnew StringBuilder(100);        Directly call
    GetStudentAddress(studid, addr);                        methods
    Console::WriteLine("Student ID : {0}", studid);
    Console::WriteLine("Address : {0}", addr->ToString());
    return 0;
}
```

In the code, you declare each unmanaged function that you need to call. For the `GetStudentID` function ❶, notice how the `LPCTSTR` parameter in the DLL function has been changed to a `String^` parameter in the P/Invoked function. When the call is made, the P/Invoke layer converts the `String^` to an `LPCTSTR` before making the call. Similarly, for the `GetStudentAddress` function ❷, you use a `String-Builder^`, where the native function used an `LPTSTR`. You need to use managed types that closely match the corresponding unmanaged types, or else the calls will either fail with erroneous results or you may get a crash. For example, passing an `int` where a `char*` is expected results in unpredictable behavior depending on what the native function expects that `char*` to be. `DllImport` can take several arguments; for example, in the declaration of `GetStudentAddress` ❷, you specify that the `CharSet` is `Unicode`. Because the focus of this book is on using C++ interop, I don't want to discuss the various `DllImport` options in detail, but if you ever end up using P/Invoke, you should look them up in the MSDN library.

Notice that you don't make a call into the exported class, because you can only call exported functions using P/Invoke. If you want to use the exported class and you have access to the DLL source code, you have to export a function that would wrap the calls to the class for you. For instance, you'd have to export a function from the DLL such as the following, where the function declares an instance of the class, makes a call into a class method, and returns the value returned by the class method:

```
extern "C" __declspec(dllexport)_API int GetStudentScore(int id)
{
    CStudentDB sdb(id);
    return sdb.GetScore();
}
```

Obviously, if you have a DLL that exports several classes, and you don't have the option of writing wrapper functions for them, P/Invoke isn't applicable in calling those functions. The advantage of using P/Invoke is that you can directly use managed types and let the P/Invoke layer do your data marshalling for you. The disadvantages are poorer performance and having to re-declare every function you intend to call. That's why C++ interop is so much more convenient and, for most purposes, a far more performant option. With that in mind, let's move on to the C++ interop version of the previous program.

Using C++ interop to access a native DLL

To access the native library from C++ interop, all you need to do is to `#include` the required header files and link with the required lib files. This is like adding video cassette support to a DVD player—like one of those 2-in-1 players available from your nearest Wal-Mart. Figure 4.12 shows a diagrammatic view of how a

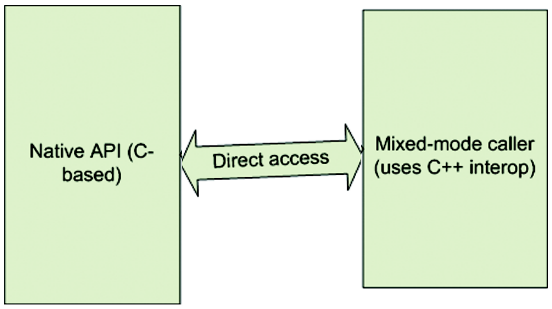


Figure 4.12 Accessing a native library using C++ interop

mixed-mode caller can directly access a native API via C++ interop; notice how there's no need for a middleman.

Once you do that, you can use the class directly, as shown in the listing 4.7.

Listing 4.7 Using C++ interop to call functions in a native DLL

```
#include "../NatLib.h" [#1]
#pragma comment(lib,"../Natlib")        <- Include required
                                           header and lib files
using namespace System;

int main(array<System::String ^> ^args)
{
    int studid = GetStudentID(_T("Howard Jones"));
    TCHAR addr[100];
    GetStudentAddress(studid, addr);                    Directly call
    Console::WriteLine("Student ID : {0}", studid);     exported native
    std::wcout << L"Address : " << addr << std::endl;   functions
    CStudentDB sdb(studid);
    Console::WriteLine("Score : {0}", sdb.GetScore());
    return 0;
}                        Instantiate exported class,
                               and invoke methods (1)
```

As is typical in mixed-mode applications, you use both native objects such as `cout` as well as managed objects such as `Console`. In addition, you can easily instantiate and use the exported class (1), which you couldn't do with P/Invoke without the help of an exported helper function that would do it for you. C++ interop is more convenient to use than P/Invoke because you don't have to declare the P/Invoke functions and can directly use the native types that are expected by the native DLL. Very often, you can avoid doing managed/unmanaged type conversions except where required. But P/Invoke does have its uses too; ideally, both mechanisms should be used where they're most suitable.

Comparison of the two techniques

Now that you have seen both techniques, let's summarize the pros and cons of each. P/Invoke is directly supported by the .NET Framework, and other languages like C# and VB.NET use it exclusively to call into unmanaged API. P/Invoke is extremely advantageous for other languages like C# and VB because it lets you access the unmanaged functions using managed types. Because C# and VB only understand managed types, this is a boon for them. With C++, this isn't as advantageous considering that C++/CLI supports both managed and unmanaged types. But if your scenario requires you to compile your C++/CLI code using `/clr:pure` (generate only MSIL—no native code), then P/Invoke is your only option. C++ interop isn't compatible with `/clr:pure` because it requires a mixed-mode assembly.

If that isn't a constraint, then C++ interop is far more convenient to use than P/Invoke. With C++ interop, you don't have to declare every API function and native structure using the `DllImport` attribute. You just include the header and lib files, and directly use the API as you would have done in an unmanaged application. You can access exported classes in a DLL, something that can't be directly done using P/Invoke. Another advantage is that you can minimize the managed/unmanaged type conversions that are required. For example, suppose your app needs a result that can be obtained from calling three native functions and then doing some processing on them. With C++ interop, you can make those API calls and perform native processing on their results to obtain the final result, which can at that point be converted to a managed type. With P/Invoke, even the subresults would use managed types, thus increasing the number of type conversions required.

In general, C++ interop is more performant than P/Invoke because of fewer type-marshalling requirements. Although you can tune up P/Invoke calls by using appropriate managed types in the declarations that reduce the type conversions required during P/Invoke, you eventually end up doing type marshalling on your own. With C++ interop, you have more control over the managed/unmanaged type conversions that need to be done. And you can speed up calls into native code by putting them in `#pragma unmanaged` blocks in your source code.

In this book, we'll be using C++ interop exclusively, because although P/Invoke may have its advantages, it's primarily intended for languages like C# and VB.NET. With C++/CLI, when you have the flexibility of using a powerful mechanism like C++ interop, you don't have to use P/Invoke. Whether you're calling into managed code from native code, calling native code from managed code, or mixing managed and native calls in a single block of code, we'll use C++ interop exclusively over other mechanisms like CCW and P/Invoke.

4.3 Using mixed types

A *mixed type* is one that requires object members to be allocated on both the CLI heap and in unmanaged memory, where these object members may be directly declared in the class or inherited from a parent. A more simplistic definition is to say that a mixed type is either a native type with a managed type member, or a managed type with a native type member. In this section, we'll examine both scenarios. An analogy is to think of a clock that shows time using hour, minute, and second hands as well as using a digital display—the time is represented both in analog and digital formats.

The VC++ 2005 compiler doesn't directly support mixed types! You may be thinking, if it doesn't, then why are you discussing mixed types in this chapter? I said "doesn't directly support"—the operative word is *directly*. There are library-provided workarounds to implement mixed types, and you can even write a smart pointer class to augment the possibilities.

The ability to support mixed types is important for C++/CLI as a language. The language is primarily being promoted as the number-one choice for doing any kind of managed/unmanaged interop programming. The moment you start doing that, the first thing you need, perhaps without your even realizing it, is support for mixed types. It's not surprising, when you think about it. Say you have an extensive MFC application with hundreds of MFC-derived classes. If you attempt to add a managed library to this application, you'll soon want to add managed type members to your MFC-derived classes (those would then be mixed types). The reverse situation for a managed application is equally possible, where your managed classes need native members when you try to add some native library to the application. In later chapters, when you see real-life applications of mixed-mode programming, just about every class you write will be a mixed-mode class.

4.3.1 Native types with managed members

In this section, we'll discuss how you can declare managed types in a native class. Obviously, you can't directly declare a managed type member in a native type. If you attempt to do so, you'll be greeted with a compiler error. The following code won't compile:

```
ref class Managed{};
class Native
{                        ❶ Throw compiler
    Managed^ m;        <-┘  error C3265
};
```

This code snippet throws compiler error C3265 ❶ because you attempt to declare a managed object as a native class member. To declare the managed member in the native type, you have to use the `gcroot` template class (which is declared in `<gcroot.h>`), which wraps the `GCHandle` structure (which is under the `System::Runtime::InteropServices` namespace). Here's how `gcroot` is used:

```
gcroot<MANTYPE^> pMan;
. . .                         ❶ Wrap managed
pMan = gcnew MANTYPE();    <-┘  type MANTYPE
pMan->ManagedMethod();     <-❷ Directly invoke methods of MANTYPE
```

Note how you can directly use `gcnew` to instantiate the `gcroot` variable ❶. This is possible because the `gcroot` template has an assignment `operator` that accepts a type `T`, which in this code snippet will be a `MANTYPE^`. There's also a conversion constructor for `gcroot` that takes a type `T`, which comes into play if you directly construct a `gcroot` object using `gcnew`. Similarly, you can also make `MANTYPE` method calls on the variable ❷. This is possible because `gcroot` has an `operator->` that returns a `T` (which is `MANTYPE^` in the example). In general, once you have a `gcroot<T>` variable, you can use it just as if it was a `T` variable, which is what `gcroot` is there for. Let's look at an example.

Imagine that you have a native `Grid` class that represents a grid control, which is internally bound to an XML data store. Next, assume that you have a managed `XmlDB` class, which is used to read and write data between the grid control and the XML data store. You're going to see how the native `Grid` class can have the `XmlDB` managed class as a member. The `XmlDB` class opens the Xml file in its constructor and closes the file in its destructor—thus, it's imperative that the destructor is called as soon as the grid control is closed. Here's the skeletal code listing of the managed `XmlDB` class:

```
ref class XmlDB
{
public:
    XmlDB(String^ XmlFile)
    {//Open XML data store
    }
    ~XmlDB()
    {//Close XML data store
    }
    void Read(int x, int y, String^ node)
    {//Read node data into (x,y)
    }
    void Write(int x, int y, String^ node)
    {//Write data in (x,y) into node
    }
};
```

I've kept the class simple, with both `Read` and `Write` methods, and nontrivial constructor and destructor. Listing 4.8 shows the skeletal native `Grid` class that uses `gcroot` to declare and use a member variable of type `XmlDB`.

Listing 4.8 Using `gcroot` to declare a managed member object

```
class Grid
{
    gcroot<XmlDB^> _xmldb;                          <- Managed member
public:                                                declared using gcroot
    Grid(char* XmlFile)
    {
        _xmldb = gcnew XmlDB(gcnew String(XmlFile));   <- Managed object
    }                                                     instantiated
    ~Grid()
    {                                               Managed object
        delete _xmldb;                          <-  deleted in destructor
    }
    const char* GetXmlStore()
    {//Return Xml file path
    }
    void PopulateGrid()
    {//Populate grid using the Xml file
        . . .
        for(int i=0; i<rows; i++)
            for(int j=0; j<cols; j++)
                _xmldb->Read(i, j, node[i,j]);
    }
    void OnGridUpdated()                             Calls made into
    {//Write updated cells back to the Xml file      managed object
        . . .
        for(int i=0; i<rows; i++)
            for(int j=0; j<cols; j++)
                _xmldb->Write(i, j, node[i,j]);
    }
    . . . //rest of the class goes here
};
```

You basically use the `gcroot<XmlDB^>` variable as if it was an `XmlDB^` variable. The only thing you have to watch out for is remembering to `delete` the managed object explicitly in the native destructor. The `XmlDB` object closes the file connection in its destructor, so it's important that you call `delete` as soon as the object isn't needed any longer (or else the file connection will remain open until the next garbage-collection cycle). This is such a common scenario that another template called `auto_gcroot` (declared in `<msclr\auto_gcroot.h>`) has been provided, which is almost the same as `gcroot`, except that it behaves like a smart pointer and has automatic resource management support. If you use `auto_gcroot`, you don't need to manually delete the managed object, because it's automatically done for you. If you re-wrote the class above using `auto_gcroot`, it would look like listing 4.9.

Listing 4.9 Using an `auto_gcroot` member

```
class Grid
{                                                   ❶ Use auto_gcroot
    msclr::auto_gcroot<XmlDB^> _xmldb;           <┘   instead of gcroot
public:
    Grid(char* XmlFile)
    {
        _xmldb = gcnew XmlDB(gcnew String(XmlFile));
    }
    const char* GetXmlStore()
    {//Return Xml file path
    }
    void PopulateGrid()
    {//Populate grid using the Xml file
     . . .
    }
    void OnGridUpdated()
    {//Write updated cells back to the Xml file
     . . .
    }
};
```

There are no changes except that you use `auto_gcroot` instead of `gcroot` ❶, and you don't have to call `delete` on the managed object, which is why you don't have a destructor. Of course, if the `Grid` class needs some resource cleanup, it will have a destructor, but it won't have to call `delete` on the `auto_gcroot` object. Now that you have seen how to have native types with managed members using `gcroot` and `auto_gcroot`, let's move ahead and see how to do the reverse—have managed types with native members.

4.3.2 *Managed types with native members*

As of VC++ 2005, the compiler won't permit you to put a whole native object member into a managed class. The following code won't compile:

```
class Native{};

ref class Managed
{
    Native m_n;        <-- ❶ Compiler error C4368
};
```

The attempt to declare a native member ❶ throws compiler error C4368: *cannot define 'm_n' as a member of managed 'Managed' : mixed types are not supported*. But it's permissible to have a pointer to a native object. The code in listing 4.10 will compile.

Listing 4.10 Managed class with native pointer member

```
ref class Managed
{
    Native* m_n;        <-- Declare pointer to native object
public:
    Managed()
    {
        m_n = new Native();        <-- Use new to create native object
    }
    ~Managed()
    {
        delete m_n;        <-- ❶ Manually delete native object
    }
};
```

The primary concern in this example is that you should `delete` the native object ❶ in the managed destructor. If you don't do so, there will be a memory leak, because native heap objects don't have the garbage collection and finalization support available in the CLR. In the previous section, you saw how the `auto_gcroot` template class acts as a smart pointer that automatically frees the managed resource when it goes out of scope. It would have been nice if such a class was provided for native objects too, but there isn't one provided in the library. Well, nothing stops you from rolling out your own smart pointer class that manages the native resource. Listing 4.11 shows a skeletal listing of what such a class can look like.

Listing 4.11 `CAutoNativePtr` skeletal listing

```
template<typename T> ref class CAutoNativePtr
{
private:
    T* _ptr;
public:
    CAutoNativePtr() : _ptr(nullptr)
    . . .
    CAutoNativePtr(T* t) : _ptr(t)
    . . .
    CAutoNativePtr(CAutoNativePtr<T>% an) : _ptr(an.Detach())
    . . .
    template<typename TDERIVED>
        CAutoNativePtr(CAutoNativePtr<TDERIVED>% an)
            : _ptr(an.Detach())
    . . .
    !CAutoNativePtr()
    . . .
    ~CAutoNativePtr()
    . . .
    CAutoNativePtr<T>% operator=(T* t)
    . . .
    CAutoNativePtr<T>% operator=(CAutoNativePtr<T>% an)
    . . .
    template<typename TDERIVED>
        CAutoNativePtr<T>% operator=(CAutoNativePtr<TDERIVED>% an)

    . . .
    static T* operator->(CAutoNativePtr<T>% an)
    . . .
    static operator T*(CAutoNativePtr<T>% an)
    . . .
    T* Detach()
    . . .
    void Attach(T* t)
    . . .
    void Destroy()
    . . .
};
```

T* holds native resource

Constructor overloads

Destructor and finalizer

Assignment operator overloads

Pointer to member operator overload

Cast operator to T*

Methods to associate and de-associate T*

Delete underlying T*

Now let's implement the various pieces of this class one by one. Doing so will also give you an example of how to implement a smart pointer class using managed code. Let's begin with the constructor overloads; see listing 4.12.

Listing 4.12 `CAutoNativePtr` constructor overloads

```
CAutoNativePtr() : _ptr(nullptr)          <- ❶ Default ctor
{
}
CAutoNativePtr(T* t) : _ptr(t)            <- Construct object
{                                            from existing T*
}
CAutoNativePtr(CAutoNativePtr<T>% an)
    : _ptr(an.Detach())                   <- ❷ Copy ctor
{
}
template<typename TDERIVED>
    CAutoNativePtr(CAutoNativePtr<TDERIVED>% an)
        : _ptr(an.Detach())               <- ❸ Copy ctor specialization
{                                            for derived object
}
```

The default constructor ❶ sets `_ptr` to a `nullptr`, whereas the constructor that takes an existing `T*` associates that `T*` with the class by assigning it to `_ptr`. Once a `T*` is associated with the class, the class is responsible for freeing that `T*` when it's no longer needed (which is when it goes out of scope). For the copy constructor ❷, you first detach the `T*` from the source object; otherwise, there will be two smart pointers associated with the same `T*`, which will result in double-deletion because both objects will attempt to delete the `T*` when it goes out of scope. By detaching the `T*` from the source object, you ensure that only one smart pointer is associated with the `T*`. You also need to have a specialization of the copy constructor ❸ to handle objects associated with a `TDERIVED*` where `TEDERIVED` is a type derived directly or indirectly from `T`.

Now, let's implement the destructor and the finalizer. You'll implement both so that even if somehow the user declares the smart pointer using handle semantics (instead of stack semantics) and then forgets to call `delete`, the `finalizer` will still free up the native resource:

```
!CAutoNativePtr()
{
    delete _ptr;
}
~CAutoNativePtr()
{
    this->!CAutoNativePtr();
}
```

Notice how you `delete` the native object in the finalizer and invoke the finalizer in the destructor. You do that to avoid code duplication. Let's implement the assignment operators now; you have three overloads for that, as shown in listing 4.13.

Listing 4.13 `CAutoNativePtr` assignment operators

```
CAutoNativePtr<T>% operator=(T* t)          ❶ Assignment from T*
{
    Attach(t);
    return *this;
}

CAutoNativePtr<T>% operator=(CAutoNativePtr<T>% an)      ❷ Assignment from another CAutoNativePtr
{
    if(this != %an) //check for self assignment
        Attach(an.Detach());
    return *this;
}

template<typename TDERIVED>
    CAutoNativePtr<T>% operator=(
        CAutoNativePtr<TDERIVED>% an)      ❸ Assignment from derived type CAutoNativePtr
{
    Attach(an.Detach());
    return *this;
}
```

The first overload ❶ takes a `T*` and is internally implemented using a call to `Attach`. You haven't seen `Attach` yet, but when you implement it, if the class is currently associated with a `T*`, it must be deleted before you take ownership of the new `T*`. The second overload ❷ is a copy assignment operator. You check for self-assignment and then use `Attach` on the `T*` that is obtained by calling `Detach` on the source object. The reason you call `Detach` is the same reason as for the copy constructor: You don't want two objects holding onto the same `T*`, because that would result in double deletion. The last overload ❸ is a specialization of the copy assignment operator that handles a `CAutoNativePtr` that owns a `T`-derived object.

Let's implement the pointer-to-member and the `T*`-cast operators, both of which return a `T*`:

```
static T* operator->(CAutoNativePtr<T>% an)
{
   return an._ptr;
}
```

```
static operator T*(CAutoNativePtr<T>% an)
{
   return an._ptr;
}
```

Both operators are implemented simply and return the `T*` owned by the object. The cast to `T*` allows `CAutoNativePtr` objects to be used anywhere a `T*` is expected. The `->` operator lets a user directly access `T` methods using the `CAutoNativePtr` object (essentially, this is what makes it a smart pointer). You still have to write the `Attach`, `Detach`, and `Destroy` methods; once you do that, you'll see an example of how to use the smart pointer class. Listing 4.14 shows the last of the methods.

Listing 4.14 The `Detach`, `Attach`, and `Destroy` methods

```
T* Detach()
{
    T* t = _ptr;
    _ptr = nullptr;          ❶ Detach method releases T*
    return t;
}

void Attach(T* t)
{
    if(t)
    {
        if(_ptr != t)
        {
            delete _ptr;
            _ptr = t;                          ❷ Attach method takes over ownership of T*
        }
    }
    else
    {
#ifdef _DEBUG
        throw gcnew Exception(
            "Attempting to Attach(...) a nullptr!");
#endif
    }
}

void Destroy()
{
    delete _ptr;             ❸ Delete underlying T* object
    _ptr = nullptr;
}
```

The `Detach` method ❶ releases ownership of the `T*` by setting its internal `T*` variable to a `nullptr` and returning the original `T*`. Once `Detach` is called, the smart pointer is no longer responsible for the native object, and the caller is responsible for freeing the resource when it's no longer needed. The `Attach` method ❷ takes over ownership of a `T*`; before it does that, it deletes the currently owned `T*`, if any. It also takes some safety precautions, such as checking to see that the calling code isn't erroneously trying to re-attach an already attached `T*`, in which case `Attach` does nothing; and checking to see if the caller is attempting to attach a `nullptr`, in which case it throws an exception (only in debug mode). The `Destroy` method ❸ deletes the underlying native object and sets the smart pointer's internal `T*` variable to a `nullptr`.

That's it; you've finished writing the smart pointer class. All you need to do now is write some code to see it in action: see listing 4.15.

Listing 4.15 Using the `CAutoNativePtr` class

```
class Native
{
public:
    void F(){}
};

class Derived : public Native{};

void SomeFunc(Native){}                      Functions that take
void SomeOtherFunc(Native*){}                Native object and Native*

ref class Ref
{
    CAutoNativePtr<Native> m_native;         <-- Declare smart pointer object
public:
    Ref()
    {}

    Ref(Native* pN) : m_native(pN)           <-- Constructor that takes T*
    {}

    Ref(CAutoNativePtr<Native> pN) : m_native(pN)   <-- Copy constructor comes into play
    {}

    void Change(Native* pNew)
    {
        m_native = pNew;                     <-- Assign from T*
    }

    void Change(CAutoNativePtr<Native> pNew)
    {
        m_native = pNew;                     <-- Assign from another smart pointer
    }
    void DoStuff()                              Logical NOT
    {                                           applied via T* cast
        if(!m_native)
        {
        }
        else
        {                                     ❶ -> operator
            m_native->F();                      at work
            SomeFunc(*m_native);                        ❷ T* cast
            SomeOtherFunc(m_native);                      at work
        }
    }

    bool DoComparisons(CAutoNativePtr<Native> a1,
        CAutoNativePtr<Native> a2, CAutoNativePtr<Native> a3)
    {
        return (a1 == a2) && (a1 != a3);            Operators == and !=
    }                                               applied via T* cast

    void Close()
    {
        m_native.Destroy();                     Free native
    }                                           resource
};

int main()
{
    CAutoNativePtr<Derived> d1(new Derived);
    CAutoNativePtr<Derived> d2(new Derived);

    CAutoNativePtr<Native> n1(d1);                  Call specialized ctor
                                                    for derived types
    n1 = d2;                        Specialized
                                    assignment operator
    return 0;                       for derived types
}
```

The class not only takes responsibility for freeing the native resource when it's no longer required, but it also provides a convenient interface to the caller. By using the `->` operator ❶, the calling code can directly access the `T*` methods. Because of the `T*` cast, passing the smart pointer to a function that expects a `T` object or a `T*` is also possible ❷. In general, the smart pointer object can be used and handled just as if it was a `T*` object, which is what you set out to do when you wrote the class. I hope that the step-by-step implementation of the earlier class gave you a good idea of how managed classes are written and how various operators are handled.

One last topic that we'll cover in this chapter is how to bridge the gap between native function pointers and CLI delegates.

4.4 Function pointers and delegates: bridging the gap

Function pointers are often used to implement a callback mechanism in native code. A *callback function* is one that's not explicitly invoked but is automatically invoked by another function when a certain event or state is triggered. Callback functions are similar to the CLI delegate/event mechanism, where the delegate associated with the event is invoked when that event is triggered. If you haven't used callbacks before, think of them as being similar to calling up a restaurant, placing an order, and asking them to call you back on your phone once the order is ready so you can pick it up.

When you start mixing native and managed code, you'll soon encounter a situation where you need to call a native function that expects a pointer to a callback function from managed code. At that point, you'll probably find it convenient if you can pass a delegate to that function rather than a function pointer. Delegates are an intrinsic part of the CLI, so using them is more natural from managed code, compared to passing a pointer to a function. (Note that this is one scenario where the P/Invoke mechanism may be useful, because it performs behind-the-scenes conversions between function pointers and delegates.) The reverse situation is also possible, where you have native code that is using an object that takes event handlers. You may find it convenient if you can pass a pointer to a native function as the event handler method.

In .NET 2.0, the framework has two new functions called `GetFunctionPointerForDelegate` and `GetDelegateForFunctionPointer` in the `Marshal` class. These functions convert between a function pointer and a delegate (see figure 4.13).

4.4.1 Using GetFunctionPointerForDelegate

Let's start by writing a managed class that enumerates the visible windows on your desktop; internally, you'll use the `EnumWindows` API function to do the window enumeration. The `EnumWindows` API is declared as follows:

```
BOOL EnumWindows(WNDENUMPROC lpEnumFunc,LPARAM lParam);
```

The first parameter to the function is a function pointer, `typedef`-ed as `WNDENUMPROC`, which is defined as follows:

```
typedef BOOL (CALLBACK* WNDENUMPROC)(HWND, LPARAM);
```

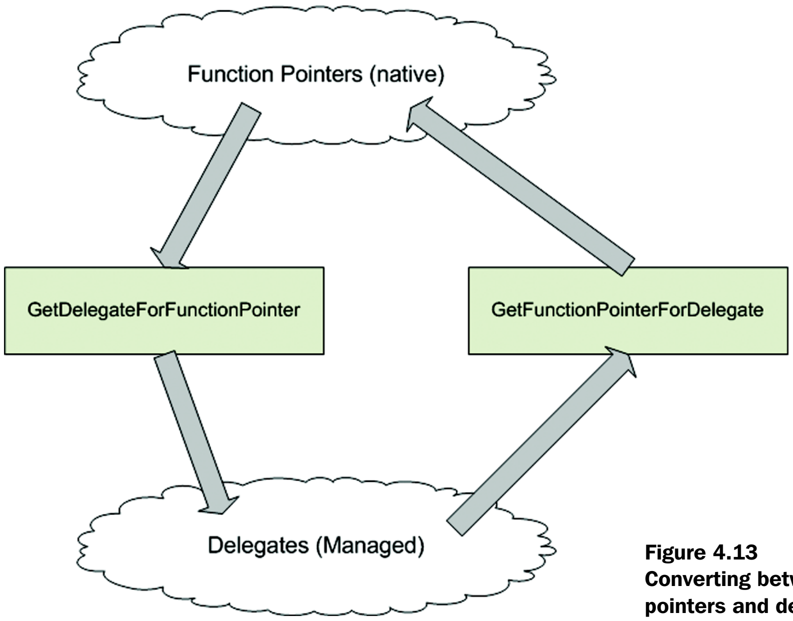

Function Pointers (native)
GetDelegateForFunctionPointer
GetFunctionPointerForDelegate
Delegates (Managed)

Figure 4.13 Converting between function pointers and delegates

For every window that is found, the API will call the function pointed to by `lpEnumFunc`. In the managed class, you'll declare a delegate that will serve as the managed version of this function pointer, and use that delegate to expose an event. You'll then use `Marshal::GetFunctionPointerForDelegate` to convert that delegate to a function pointer that can then be passed to the `EnumWindows` API function. Listing 4.16 shows the managed window enumeration class.

Listing 4.16 Class to enumerate windows

```
delegate bool EnumWindowsDelegateProc(
    IntPtr hwnd,IntPtr lParam);      <-- Delegate used in lieu of function pointer

ref class WindowEnumerator
{
private:
    EnumWindowsDelegateProc^ _WindowFound;      <-- Private delegate member
public:
    WindowEnumerator(EnumWindowsDelegateProc^ handler)
    {
        _WindowFound = handler;
    }
    void Init()
    {
        pin_ptr<EnumWindowsDelegateProc^> tmp =  &_WindowFound;   <-- Pin delegate before passing to native code (1)
        EnumWindows((WNDENUMPROC)
            Marshal::GetFunctionPointerForDelegate(
            _WindowFound).ToPointer(), 0);
    }
};
```

❷ **Call API with converted function pointer**

Note that you don't strictly need to pin the delegate ❶; you only need to ensure that the GC doesn't collect it. This is because the GC doesn't keep track of unmanaged pointers; but pinning solves that issue for you and doesn't cause any noticeable overhead in this scenario. When the call to `EnumWindows` is executed ❷, the callback that is passed to it is a native pointer to the delegate, which means this delegate is invoked for every window.

To use this class, all you need is a method that matches the `EnumWindows-DelegateProc` delegate signature. You're going to write such a method (as a static method of a class); it will display the title text of each visible window that's enumerated.

4.4.2 Using GetDelegateForFunctionPointer

You have only seen `GetFunctionPointerForDelegate` in action so far. To see the reverse function `GetDelegateForFunctionPointer`, you'll use a pointer to the native `printf` function to display each enumerated window. That way, in using a single example, you can see both conversions. Listing 4.17 shows the code for the class.

Listing 4.17 Code that performs delegate/function pointer conversions

```
ref class MyClass
{
    delegate int DispProc(String^, String^);
    static DispProc^ pDispProc = nullptr;
public:
    static MyClass()
    {
        HMODULE hLib = LoadLibrary(_T("msvcrt.dll"));
        if(hLib)
        {
            typedef int (*FUNC_PTR)(const char *, ...);
            FUNC_PTR pfn = reinterpret_cast<FUNC_PTR>(
                GetProcAddress(hLib, "printf"));
            if(pfn)
            {
                pDispProc = (DispProc^)
                    Marshal::GetDelegateForFunctionPointer(
                    (IntPtr)pfn,DispProc::typeid);
            }
            FreeLibrary(hLib);
```

❶ **Delegate to display each window**

❷ **Convert printf function to a delegate**

```
        }
    }
    static bool HandleFoundWindow(IntPtr hwnd,IntPtr lParam)
    {
        TCHAR buff[512];
        GetWindowText((HWND)hwnd.ToPointer(), buff, 511);
        if(IsWindowVisible((HWND)hwnd.ToPointer()) && _tcslen(buff))
            pDispProc("%s\r\n\r\n",gcnew String(buff));   <-
        return TRUE;                                         | Use pDispProc to
    }                                                        | indirectly invoke printf ❸
};
```

A delegate ❶ is used to display each enumerated window. The most interesting portion in this code snippet is the static constructor ❷, where you obtain a function pointer to the `printf` function (from msvcrt.dll), convert it to a delegate, and then save it in the `pDispProc` delegate variable. In the `HandleFoundWindow` method, when you invoke the `pDispProc` delegate ❸, `printf` gets executed, because the delegate is merely a proxy to the unmanaged function pointer (that points to `printf`). In the previous class, you passed a delegate to a method expecting a function pointer, and the delegate was invoked through that function pointer. However, in this class, you pass a function pointer to a delegate, and the function pointer is invoked through the delegate. Here's a code snippet that shows how these two classes can be put to use:

```
WindowEnumerator we(gcnew EnumWindowsDelegateProc(
    MyClass::HandleFoundWindow));
we.Init();
```

As you can see, the calling code has no idea that the class internally converts the delegate to a function pointer. This ability to convert between function pointers and delegates is particularly useful when you're wrapping a native library so as to expose it to the .NET world. Languages such as C# and VB.NET only understand delegates, whereas a lot of native libraries use function pointers to provide callbacks. By using these `Marshal` class methods, you can expose delegates to languages like C# and VB.NET and internally continue to use function pointers. In the same way, when you're attempting to use a managed class that has events from native code, you can take advantage of the ability to convert a function pointer to a delegate to directly pass pointers to native methods as the event handler functions.

A trick to use assembly code from managed code

You can't have inline assembly language code in managed blocks, which rules out using them in pure MSIL modules. Even in mixed-mode modules, you'd need to restrict them to unmanaged blocks of code. Here's a trick that uses `GetDelegateForFunctionPointer` and allows you to directly execute assembly code. You can declare an unsigned `char` array containing the assembly code for a function, convert that to a delegate using `GetDelegateForFunctionPointer`, and then execute that delegate. Here' s some sample code:

```
delegate Int32 DoubleNum(Int32 x);

unsigned char pNative[] =
{
    0x55, // push ebp
    0x8B, 0xEC, // mov ebp,esp
    0x8B, 0x45, 0x08, // mov eax,dword ptr [arg]
    0x03, 0x45, 0x08, // add eax,dword ptr [arg]
    0x5D, // pop ebp
    0xC3 // ret
};

DoubleNum^ pDoubleNum = (DoubleNum^)
    Marshal::GetDelegateForFunctionPointer(
        (IntPtr)pNative, DoubleNum::typeid);

Console::WriteLine(pDoubleNum(19));
```

You should do this only if you're familiar with assembly and are sure of what you're doing. It's more of a feel-good trick rather than something you'd want to put to use in production code.

4.5 Summary

In this chapter, we've discussed some seemingly disconnected topics, but they're all important concepts in mixed-mode programming. It was vital that you understand the workings of pinning and interior pointers and also have an idea of when and when not to use them.

We also covered various interop mechanisms, and although you saw that every mechanism has its own set of advantages and best-use situations, we won't be using either CCW or P/Invoke in the rest of this book. For the sort of mixed-mode programming we'll be covering, C++ interop is the fastest and most convenient option. We'll use that exclusively for our interop needs.

We discussed mixed types and how to use the `gcroot/auto_gcroot` library's provided classes. We also implemented a similar smart pointer class for automatically managing a native resource in a managed class. We'll be using mixed types considerably in the rest of this book, and the techniques you learned in this chapter will be repeatedly put into practice.

The last topic we discussed, which was how to convert between function pointers and delegates, is something that you'll see again when writing native wrappers for managed code or managed wrappers for native code.

In the next chapter, we'll use the information gained in this and the previous chapters to demonstrate how you can access and utilize native libraries from managed applications.

.NET

C++/CLI IN ACTION

Nishant Sivakumar

Developers initially welcomed Microsoft's Managed C++ for .NET, but the twisted syntax made it difficult to use. Its much-improved replacement, C++/CLI, now provides an effective bridge between the native and managed programming worlds. Using this technology, developers can combine existing C++ programs and .NET applications with little or no refactoring. Accessing .NET libraries like Windows Forms, WPF, and WCF from standard C++ is equally easy.

C++/CLI in Action is a practical guide that will help you breathe new life into your legacy C++ programs. The book begins with a concise C++/CLI tutorial. It then quickly moves to the key themes of native/managed code interop and mixed-mode programming. You'll learn to take advantage of GUI frameworks like Windows Forms and WPF while keeping your native C++ business logic. The book also covers methods for accessing C# or VB.NET components and libraries. Written for readers with a working knowledge of C++.

What's Inside

- Call C++ libraries from C# or VB.NET
- C++ for WPF and WCF
- Mixed-mode programming techniques
- Move from Managed C++ to C++/CLI

Nishant Sivakumar is a C++ developer living in Atlanta, GA. He has been a Microsoft Visual C++ MVP since 2002 and is active online at blog.voidnish.com.

For more information, code samples, and ebook visit
www.manning.com/Cplusplus/CLIinAction

"… a great resource, an outstanding job, a must-read …"

—Ayman B. Shoukry
VC++ Team
Microsoft Corporation

"… explains the C++/CLI language and shows how it can be used."

—James Johnson
www.misterdotnet.com/blog/

"… a great reference … a must-have in any C++ programmer's bookshelf."

—Michael L. Taylor, C# MVP

"… absolutely the only book you'll need to learn this powerful new .NET language … a lifesaver …."

—Christian Graus, Code Project

ISBN-10: 1-932394-81-8
ISBN-13: 978-1-932394-81-8

54999

9 781932 394818

MANNING $49.99 US/$64.99 Canada