

Pro Visual C++/CLI and the .NET 2.0 Platform



Stephen R. G. Fraser

Pro Visual C++/CLI and the .NET 2.0 Platform

Copyright © 2006 by Stephen R.G. Fraser

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN: 1-59059-640-4

Library of Congress Cataloging-in-Publication data is available upon request.

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editor: Ewan Buckingham

Technical Reviewer: Don Reamey

Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser

Project Managers: Laura Cheu, Richard Dal Porto

Copy Edit Manager: Nicole LeClerc

Copy Editors: Freelance Editorial Services, Ami Knox, Liz Welch

Assistant Production Director: Kari Brooks-Copony

Production Editor: Katie Stence

Compositor: Susan Gilnert

Proofreader: Elizabeth Berry

Indexer: John Collin

Artist: April Milne

Interior Designer: Van Winkle Design Group

Cover Designer: Kurt Krames

Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit <http://www.springeronline.com>.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit <http://www.apress.com>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com> in the Source Code section. You will need to answer questions pertaining to this book in order to successfully download the code.



Object-Oriented C++/CLI

In the previous chapter, I covered in detail the basics of C++/CLI, focusing on programming strictly in a procedural style. This chapter explores the real strength of C++/CLI: as an object-oriented language.

The chapter starts with a review of object-oriented programming (OOP) in general. You will then explore C++/CLI's OOP capabilities, focusing primarily on `ref` classes, which are the cornerstones of C++/CLI OOP. You will do this by breaking `ref` classes down into their parts and examining each part in detail. Finally, you will learn about interfaces.

Caution Don't skip this chapter, even if you know C++ very well, because several things are different between traditional C++ and C++/CLI. True, some of the changes may not be significant, but recognizing and understanding these changes now may make your life easier in the future.

OOP is more a way of thinking than a programming technique. For those making the transition from procedural programming, you must understand that OOP will involve a paradigm shift for you. But, once you realize this and make the shift, you will wonder why you programmed any other way.

OOP is just an abstraction taken from everyday life and applied to software development. The world is made up of objects. In front of you is a book. It is an object. You are sitting on a chair or a couch, or you might be lying on a bed—all objects. I could go on, but I'm sure you get the point. Almost every aspect of our lives revolves around interacting with, fixing, and improving objects. It should be second nature to do the same thing with software development.

Object-Oriented Concepts

All objects support three specific concepts: encapsulation, inheritance, and polymorphism. Think about the objects around you—no, scratch that; think about yourself. You are an object: You are made up of arms, legs, a torso, and a head, but how they work does not matter to you—this is encapsulation. You are a mammal, human, and male or female—this is inheritance. When greeted, you respond with “Good day,” “Bonjour,” “Guten Tag,” or “Buon giorno”—this is polymorphism.

As you shall see shortly, you can apply the object paradigm to software development as well. C++/CLI does it by using software objects called `ref` classes and `ref` structs. But before I get into software objects, let's examine the concepts of an object more generically.

Encapsulation

All objects are made up of a combination of different things or objects. Many of these things are not of any concern to the other objects that interact with them. Going back to you as an example of an

object, you are made up of things such as blood, muscles, and bone, but most objects that interact with you don't care about that level of things. Most objects that interact with you only care that you have hands, a mouth, ears, and other features at this level of abstraction.

Encapsulation basically means hiding the parts of an object that do things internally from other objects that interact with it. As you saw in the previous example, the internal workings of hands, a mouth, and ears are irrelevant to other objects that interact with you.

Encapsulation is generally used to simplify the model that other objects have to interact with. It allows other objects to only be concerned with using the right interface and passing the correct input to get the required response. For example, a car is a very complex object. But, to me, a car is simple: A steering wheel, an accelerator, and a brake represent the interface; and turning the steering wheel, stepping on the accelerator, and stepping on the brake represent input.

Encapsulation also allows an object to be fixed, updated, or replaced without having to change the other objects interacting with it. When I trade in my Mustang LX for a Mustang GT, I still only have to worry about turning the steering wheel, stepping on the accelerator, and stepping on the brake.

The most important thing about encapsulation is that because portions of the object are protected from external access, it is possible to maintain the internal integrity of the object. This is because it is possible to allow only indirect access, or no access at all, to private features of the object.

Inheritance

Inheritance is hardly a new concept. We all inherit many traits (good and bad) from both of our parents. We also inherit many traits from being a mammal, such as being born, being nursed, having four limbs, and so on. Being human, we inherit the traits of opposable thumbs, upright stature, capacity for language, and so forth. I'm sure you get the idea. Other objects also inherit from other more generic objects.

You can think of inheritance as a tree of objects starting with the most generic traits and expanding to the most specific. Each level of the tree expands on the definition of the previous level, until finally the object is fully defined. Inheritance allows for the reuse of previously defined objects. For example, when you say that a Mustang is a car, you know that it has four wheels and an engine. In this scenario, the base object definition came for free—you didn't have to define it again.

Notice, though, that a Mustang is always a car, but a car need not be a Mustang. The car could be a Ferrari. The link of inheritance is one way, toward the root.

Polymorphism

The hardest concept to grasp is polymorphism—not that it's difficult, but it's just taken so much for granted that it's almost completely overlooked. *Polymorphism* is simply the ability for different objects derived from a common base object to respond to the same stimuli in completely different ways.

For example, (well-trained) cats, dogs, and birds are all animals, but when asked to speak, they will all respond differently. (I added "well-trained" because normally a cat will look at you as if you are crazy, a dog will be to busy chasing his tail, and a bird will squawk even if you don't ask it to do anything.)

You can't have polymorphism without inheritance, as the stimuli that the object is expected to respond to must be to an interface that all objects have in common. In the preceding example, you are asking an animal to speak. Depending on the type of animal (inheritance), you will get a different response.

A key thing about polymorphism is that you know that you will get a response of a certain type, but the object responding—not the object requesting—determines what the actual response will be.

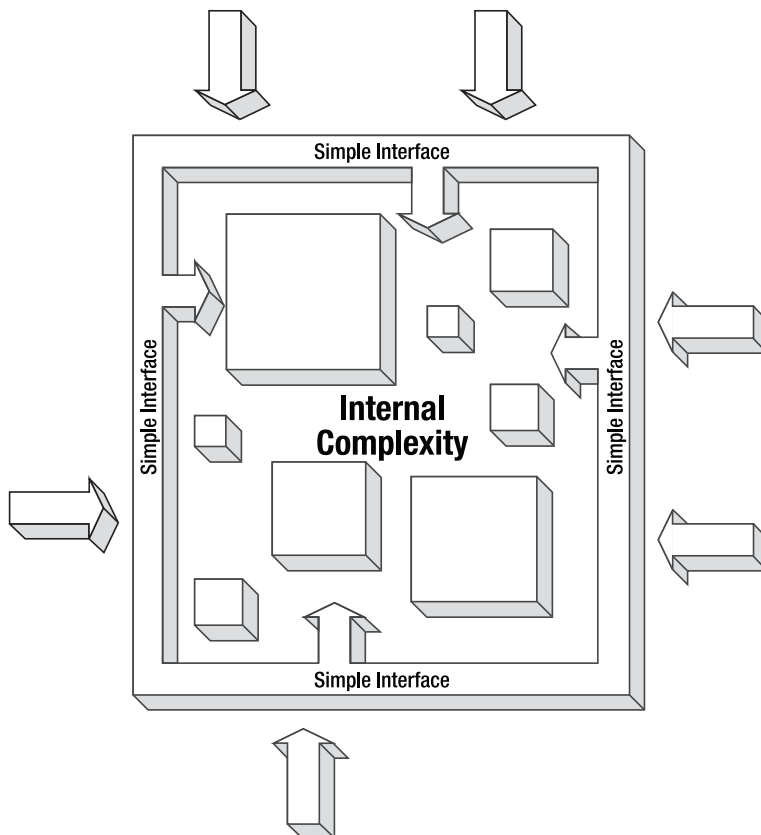
Applying Objects to Software Development

Okay, you know what objects and their concepts are and how to apply them to software development. With procedural programming, there is no concept of an object, just a continual stream of logic and data. Let me back up a bit on that. It could be argued that, even in procedural programming, objects exist, given that variables, literals, and constants could be considered objects (albeit simple ones). In procedural programming, breaking up the logic into smaller, more manageable pieces is done by way of functions. To group common data elements together, the structure or class is used depending on language.

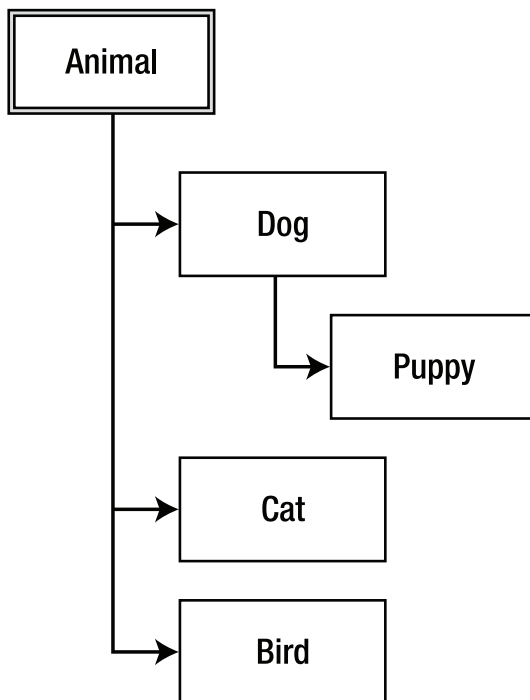
Before you jump on me, I would like to note that there were (obviously) other object-oriented languages before C++, but this book only covers C++/CLI's history. It wasn't until C++ that computer data and its associated logic was packaged together into the struct and a new construct known as the class. (If you are a purist, there was "C with Classes" first.) With the combination of data and logic associated with this data into a single construct, object-oriented concepts could be applied to programming.

Here, in a nutshell, is how object-oriented concepts are applied to C++/CLI development. Classes and structures are programming constructs that implement within the C++ language the three key object-oriented concepts: encapsulation, inheritance, and polymorphism.

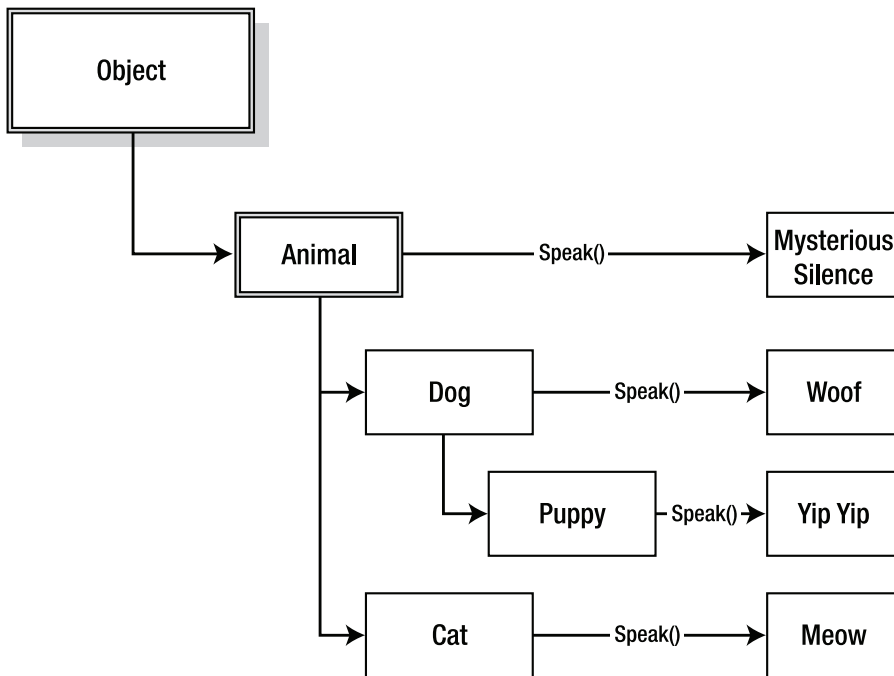
Encapsulation, or the hiding of complexity, is accomplished by not allowing access to all data and functionality found in a class. Instead, only a simpler and more restricted interface is provided to access the class.



Inheritance is the ability to innately reuse the functionality and data of one class within another derived class.



Polymorphism is the ability for different classes to respond to the same request in different ways. Classes provide something called the *virtual method*, or *function*, which allows any class derived from the same parent class to respond differently to the same request.



I expand on each of these concepts as the chapter progresses.

Now that you understand OOP conceptually, let's see how it is actually done with C++/CLI.

ref class/struct Basics

First off, there is nothing forcing you to program in an objected-oriented fashion with `ref` classes or `ref` structs, and using `ref` classes and `ref` structs does not mean you are doing OOP. For example, it is possible to break the code based on function areas instead of objects. It may appear to be OOP, but what you really are doing is just using a `ref` class as a namespace. Plus, you gain none of the benefits of OOP.

Note I am currently only dealing with the managed data types within C++/CLI (`ref` class, `ref` struct, `value` class, and `value` struct). In Chapters 20 and 21, I'll cover unmanaged and native data types (`class` and `struct`).

It is the organization of code and data into unique objects that distinguishes procedural coding from object-oriented coding.

For those totally new to object programming, you need to understand that each time you create (or instantiate) a `ref class` definition a new instance of the `ref class` object will be created. In other words, no matter how many instances you want, they will all be unique instances created from the same `ref class` definition.

But before you look at objects and OOP, you will look at the `ref class` and the `ref struct` and what makes up a `ref class` and a `ref struct` in general terms.

Declaring `ref classes` and `structs`

The `ref class` and `ref struct` are basically an extension of the traditional C++ `class` and `struct`. Like traditional C++ `classes` and `structs`, `ref classes` and `ref structs` are made up of variables and methods. Unlike traditional `classes` and `structs`, `ref classes` and `ref structs` are created and destroyed in a completely different manner. Also, `ref classes` and `ref structs` have an additional construct called the `property`.

Private, Public, and Protected Member Access Modifiers

There really is no real difference between a `ref class` and `ref struct` except for the default access to its members. A `ref class` defaults to private access to its members, whereas a `ref struct` default to public. Notice that I used the term “default.” It is possible to change the access level of either the `ref class` or the `ref struct`. So, truthfully, the usage of a `ref class` or a `ref struct` is just a matter of taste. Most people who code C++ use the keywords `ref class` when they create objects, and `ref struct` is very seldom if ever used.

Note Because `ref struct` is very seldom used, I'm going to use `ref class` from here on, but you can assume `ref struct` applies as well.

The way you declare `ref classes` is very similar to the way you declare traditional `classes`. Let's look at a `ref class` declaration. With what you learned in Chapter 2, much of a `ref class` definition should make sense. First, there is the declaration of the `ref class` itself and then the declaration of the `ref class`'s variables, properties, and methods.

The following example is the `Square` `ref class`, which is made up of a constructor, a method to calculate the square's area, and a dimension variable:

```
ref class Square
{
    // constructor
    Square ( int d)
    {
        Dims = d;
    }

    // method
    int Area()
    {
        return Dims * Dims;
    }

    // variable
    int Dims;
};
```


The first thing to note about this `ref class` is that because the access to `ref` classes defaults to `private`, the constructor, the method, and the variable are not accessible outside the `ref class`. This is probably not what you want. To make the `ref class`'s members accessible outside of the `ref class`, you need to add the access modifier `public`: to the definition:

```
ref class Square
{
public:
    // public constructor
    Square ( int d)
    {
        Dims = d;
    }

    // public method
    int Area()
    {
        return Dims * Dims;
    }

    // public variable
    int Dims;
};
```

With this addition, all the `ref class`'s members are accessible. What if you want some members to be `private` and some `public`? For example, what if you want the variable `Dims` only accessible through the constructor? To do this, you add the `private`: access modifier:

```
ref class Square
{
public:
    Square ( int d)
    {
        Dims = d;
    }

    int Area()
    {
        return Dims * Dims;
    }
private:
    int Dims;
};
```

Besides `public` and `private`, C++/CLI provides one additional member access modifier: `protected`. Protected access is sort of a combination of `public` and `private`; where a `protected ref class` member has `public` access when it's inherited but `private` access (i.e., can't be accessed) by methods that are members of a `ref class` that don't share inheritance.

Here is a quick recap of the access modifiers for members.

If the member has `public` access, it is

- Accessible by external functions and methods
- Accessible to derived ref classes

If the member has private access, it is

- Not accessible by external functions and methods
- Not accessible to derived ref classes

If the member has protected access, it is

- Not accessible by external functions and methods
- Accessible to derived ref classes

If you are visually oriented, as I am, maybe Figure 3-1 will help clear up member access modifiers.

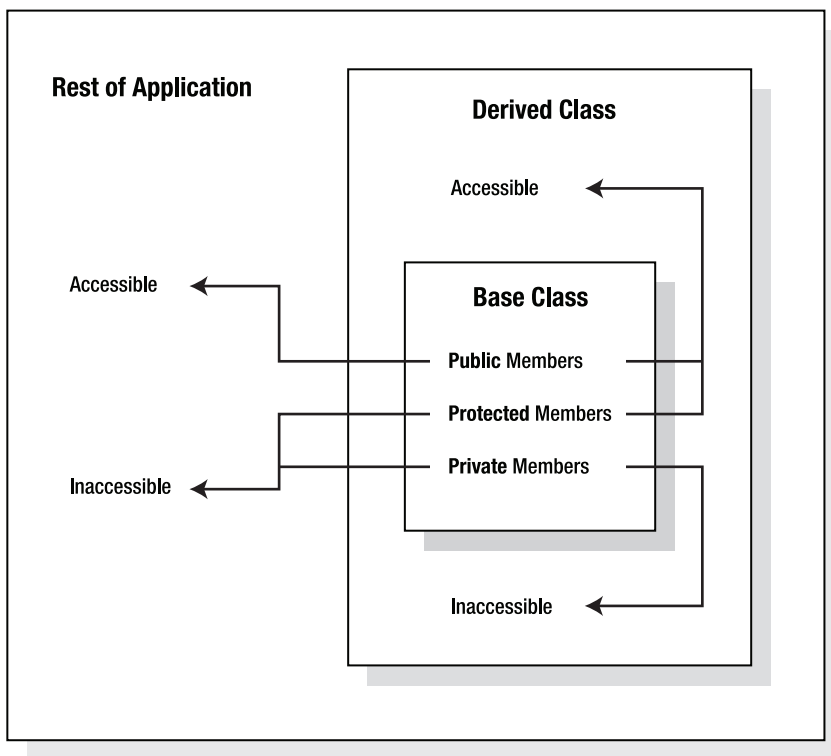


Figure 3-1. Summary of the three member access modifiers

The ref Keyword

If you have come from the traditional C++ world, you may have noticed the new keyword `ref` in front of the class's definition. This is one of the biggest and most important changes between traditional C++ and C++/CLI. (It is also a big change from C++/CLI and Managed Extensions for C++, as the keyword was `__gc`.) The use of the `ref` keyword tells the compiler, which in turn tells the common language runtime (CLR), that this class will be a reference object on the managed heap. C++/CLI, to be consistent with traditional C++, defaults to classes being placed on the CRT heap and not the

managed heap. It is up to developers to decide if they want the class to be managed and, if so, they must manually place the `ref` in front of the class.

For this minor inconvenience, you get several benefits:

- Garbage collection
- Inheritance from any .NET Framework base class that is not sealed (I cover sealed classes later in the chapter) or, if no base class is specified, automatic inheritance from the `System::Object` class
- Ability for the `ref` class to be used within .NET Framework collections and arrays
- Inheritance from any number of managed interfaces
- Ability to contain properties
- Ability to contain pointers to unmanaged classes

Unsafe Code Because pointers to unmanaged classes is unverifiable, placing these pointers within a `ref` class is unsafe.

With the good, there is the bad. Traditional C++ programmers might find these items drawbacks with `ref` classes:

- Only single class inheritance is allowed.
- Inheritance from unmanaged classes is not allowed.
- In addition, `ref` classes
 - Cannot be a parent class of an unmanaged type
 - Do not support friends
 - Cannot contain an overridden `gcnew` or `delete` operator
 - Must use public inheritance
 - Cannot be used with the `sizeof` or `offsetof` operator
- Pointer arithmetic on the `ref` class handles is not allowed.

On the other hand, these drawbacks may not be as bad as you might think. A `ref` class allows multiple interface inheritance, though, as I will show later, you are forced to implement all the methods for these interfaces within the new `ref` class. The .NET Framework is quite extensive, so inheritance of unmanaged classes may not be needed as frequently as you might expect. Overriding `gcnew` and `delete` seems to defeat the purpose of `ref` classes. Because pointer arithmetic is not allowed on handles, `sizeof` and `offsetof` are kind of useless, anyway, and pointer arithmetic is a very big contributor to memory leaks and programs aborting as a result of illegal memory access.

Inheriting `ref` classes

Even though writing a stand-alone `ref` class can provide quite a lot of functionality to an object, it is in the object-oriented nature of `ref` classes and their capability to inherit from other `ref` classes that their real strength lays.

As I mentioned earlier, `ref` classes have the ability to inherit from a single `ref` class and multiple interfaces. I focus on `ref` class inheritance now, and later in this chapter, I will look at interface inheritance.

Inheriting from a `ref` class allows an inheriting `ref` class (usually known as the *child*) to get access to all the public and protected members of the inherited `ref` class (usually known as the

parent or *base class*). You can think of inheritance in one of two ways: It allows the functionality of the base class to expand without the need to duplicate any of the code, or it allows the child class to fix or augment some feature of its parent class without having to know or understand how the parent functions (this is encapsulation, by the way). But, really, they both mean the same thing.

A restriction imposed by C++/CLI on *ref* classes is that *ref* classes can only use public inheritance. For example:

```
ref class childClass : public baseClass {};
```

is allowed, but the following will generate compile time errors:

```
ref class childClass : protected baseClass {}; // Error
ref class childClass : private baseClass {};   // Error
```

This means that with the public access to a base class, the child can access any public or protected member of the base class as if it were one of its own members. Private members of the base class, on the other hand, are not accessible by the child class, and trying to access them will generate a compilation error.

Unmanaged classes (also known as *native* and *unsafe* classes) can have public, protected, or private access to their base class. Notice there is no “*ref*” in front of these classes:

```
class childClass : public baseClass {}
class childClass : protected baseClass {}
class childClass : private baseClass {}
```

Unsafe Code *Unmanaged classes* are not verifiable and thus are an unsafe coding construct.

For private *ref* class access, all base class members are inherited as private and thus are not accessible. Protected *ref* class access allows access to public and protected base class members but changes the public access to protected. Personally, I’ve never used private or protected base class access, as I’ve simply never had a use for it, but it’s available if you ever need it.

Listing 3-1 shows the *Cube* *ref* class inheriting from a *Square* *ref* class. Notice that because both the member access and the *ref* class access of the *Square* *ref* class are public, the *Cube* *ref* class has complete access to the *Square* *ref* class and can use all the *Square* *ref* class’s members as if they were its own.

Listing 3-1. *Inheritance in Action*

```
using namespace System;

// Base class
ref class Square
{
public:
    int Area()
    {
        return Dims * Dims;
    }

    int Dims;
};
```

```
// Child class
ref class Cube : public Square
{
public:
    int Volume()
    {
        return Area() * Dims;
    }
};

// Inheritance in action
void main()
{
    Cube ^cube = gcnew Cube();
    cube->Dims = 3;

    Console::WriteLine(cube->Dims);
    Console::WriteLine(cube->Area());
    Console::WriteLine(cube->Volume());
}
```

Figure 3-2 shows the results of this little program.



Figure 3-2. Results of *Inherit.exe*

Sealed ref classes

A *sealed* ref class is one that cannot be inherited from. The sealed ref class enables a developer to stop all other developers from inheriting from the defined ref class. I have never had an opportunity to seal any of my ref classes. I have come across it a few times. Almost every time, I was forced to create my own similar ref class because I needed additional functionality that the sealed ref class lacked. Personally, I feel the sealed ref class goes against object-oriented development, because it stops one of the key OOP cornerstones: inheritance. But the tool is available for those who wish to use it.

The code to seal a ref class is simply the addition of the specific location identifier sealed after the ref class name in the ref class definition, like this:

```
ref class sealClass sealed
{
};
```

Using the ref class

Unlike procedural code, the declaration of a ref class is simply that: a declaration. The ref class's methods do nothing on their own and only have meaning within the confines of an object. An object is

an instantiated `ref` class. A neat thing about when a `ref` class is instantiated is that automatically all the `ref` classes it is derived from also get instantiated.

The code to instantiate or create an object from the `ref` class in the previous section is simply this:

```
Square ^sqr = gcnew Square(); // a handle
```

or this:

```
Square sqr; // a local or stack instance
```

Notice that you can create either a handle or a local or stack instance to the `Square` object. For those of you coming from a traditional C++ background, the syntax when working with handles is identical to pointers except for the initial declaration just shown. I personally have found the syntax when working with stack instances of an object a little easier but, as you will find out, in many cases you simply can't use them.

Handle to an Object

If you recall from the previous chapter, C++/CLI data types can fall into one of two types: value types and reference types. A `ref` class is a reference type. What this means is that the `ref` class, when created using the `gcnew` operator, allocates itself on the managed heap, and then a handle is placed on the stack indicating the location of the allocated object.

This is only half the story, though. The CLR places the `ref` class object on the managed heap. The CLR will maintain this `ref` class object on the heap so long as a handle is using it. When all handles to the object go out of scope or, in other words, no variables are accessing it, the CLR will delete it automatically.

Caution If the `ref` class object accesses certain unmanaged resources that hold system resources, the CLR will hold the `ref` class object for an indefinite (not necessarily infinite) period of time. Using COM-based ADO is a classic example of this. This was a major issue in prior versions of C++/CLI (Managed Extensions for C++), but the addition of deterministic destructors has helped alleviate this issue. I cover destructors later in this chapter.

Once you have created an instance of a `ref` class using the following:

```
Square ^sqr = gcnew Square(); // a handle
```

you now have access to its variables, properties, and methods. The code to access a reference object handle is simply the name of the object you created followed by the arrow `->` operator. For example:

```
sqr->Dims = 5;  
int area = sqr->Area();
```

You might be wondering why pointer arithmetic is not allowed on reference object handles. They seem harmless enough. Well, the problem comes from the fact that the location of the object in the managed heap memory can move. The garbage collection process not only deletes unused objects in heap memory, it also compacts it. Thus, it is possible that a `ref` class object can be relocated during the compacting process.

Local or Stack Objects

I assume that up until now you've been using the member access operator or dot `.` operator on faith that I would explain it later. There really isn't anything special about the dot operator; it's only used for accessing individual member variables, properties, or methods out of a `ref` class. Its syntax is simply this:

```

class-name . member-data-or-method
int intval;           // value type
String ^s = intval.ToString();
Square sqr;           // reference type
int i = sqr.Dims;

```

You have seen both the `->` and `.` operators used when accessing `ref` class members. What is the difference? The `->` operator is used to access data or methods from a handle or a pointer off a heap, whereas the dot `.` operator is used to access the object off the stack.

You have already seen one type of stack object the value type. It is also possible to create a stack object out of a reference type as was just shown.

Listing 3-2 is an example of using a `ref` class as both a stack and a heap reference.

Listing 3-2. Stack Reference Object in Action

```

using namespace System;

ref class Square
{
public:
    int Area()
    {
        return Dims * Dims;
    }
    int Dims;
};

void main()
{
    Square ^sqr1 = gcnew Square();    // Handle
    sqr1->Dims = 2;
    Console::WriteLine( sqr1->Area() );

    Square sqr2;                      // local stack instance
    sqr2.Dims = 3;
    Console::WriteLine( sqr2.Area() );
}

```

Figure 3-3 shows the results of this little program.

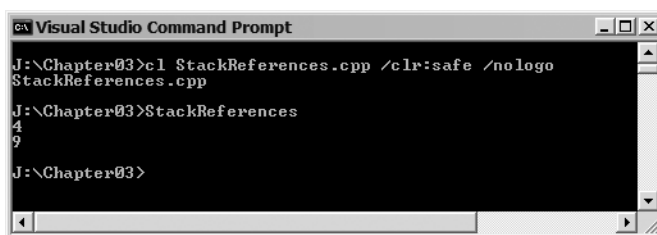


Figure 3-3. Results of *StackReferences.exe*

Member Variables

This fancy name is simply C++/CLI's way of reminding programmers that `ref` classes are objects. Member variables are simply variables defined within the definition of a `ref` class.

The syntax of defining member variables is identical to that of ordinary variables except for one important thing: You cannot initialize a variable in its definition. I explain how to initialize variables later in this chapter when I cover constructors. The definition of a variable is simply a data type and a variable name, or a comma-delimited list of variable names:

```
ref class varExample
{
    int x;
    String ^str1, ^str2;
};
```

In C++/CLI, member variables can be either managed data types or a pointer to an unmanaged data type.

Unsafe Code The pointer to an unmanaged data type causes the entire ref class to become unsafe.

Member variables can be public, protected, or private. With C++/CLI and ref classes, public member variables should be handled with care, especially if invalid values in these variables will cause problems in the program's execution. A better solution is to make them private (or protected, so that inherited access can still access them directly), and then make public properties to them for external methods to access. Properties can, if coded correctly, perform validation on the data entered into the variable. Otherwise, they work just like normal member variables. I cover properties later in this chapter.

Static Member Variables

Static member variables are variables that provide ref class-wide storage. In other words, the same variable is shared by all instances of a ref class. At first glance, you might wonder why you would want a shared member variable across all instances of a ref class type. A couple of reasons that you will frequently come across for using static member variable is as a counter of how many instances of the ref class have been created and how many of those instances are currently active. Though, I'm sure you will come up with several other reasons to use them.

To define a static member variable in a ref class, simply define it as static and assign a value to it in the ref class definitions, like this:

```
ref class staticVar
{
    static int staticVar = 3;
};
```

You might be wondering how initializing the variable within the ref class can work, as it would appear that the value would be reset for each instance of the ref class. Fortunately, this is not the case; only the first time that the ref class is instantiated is the variable created and initialized.

Member Methods

A *member method* is simply a fancy term that means that the function is declared within a ref class. Everything you learned about functions in the previous chapter is applicable to member methods. You might consider revisiting Chapter 2's section on functions if you are uncertain how they are defined or how they work.

Like all members of a `ref class`, member methods can be public, protected, or private. Public methods are accessible outside of the `ref class` and are the workhorse of interclass communication. It is via methods that `ref classes` pass messages, requesting and being requested to perform some type of functionality. Protected member methods are the same as private member methods except that inherited `ref classes` have access to them. Private `ref classes` encapsulate the functionality provided by the `ref class`, as they are not accessible from outside the `ref class` except via some public member method that uses it.

Just as a quick recap, Listing 3-3 is a public member method that calls a protected member method that calls a private member method.

Listing 3-3. *Member Methods in Action*

```
using namespace System;

ref class MethodEx
{
public:
    void printPublic(int num)
    {
        for (int i = 0; i < num; i++)
        {
            Console::WriteLine( "Public" );
        }
        printProtected(num/2);
    }
protected:
    void printProtected(int num)
    {
        for (int i = 0; i < num; i++)
        {
            Console::WriteLine( "Protected" );
        }
        printPrivate(num/2);
    }
private:
    void printPrivate(int num)
    {
        for (int i = 0; i < num; i++)
        {
            Console::WriteLine( "Private" );
        }
    }
};

int main()
{
    MethodEx ex;

    ex.printPublic(4);
    // ex.printProtected(4); // Error cannot access
    // ex.printPrivate(4);   // Error cannot access
}
```

Figure 3-4 shows the results of this little program.

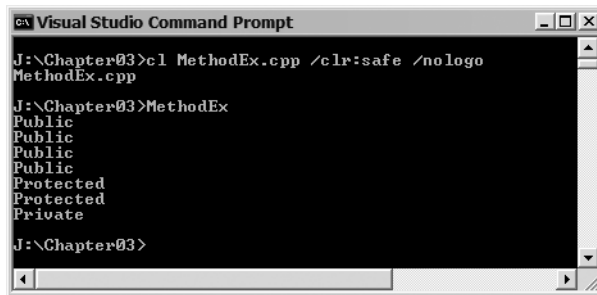


Figure 3-4. Results of *MethodEx.exe*

Static Member Methods

Static member methods are methods that have `ref class` scope. In other words, they exist without your having to create an instance of the `ref class`. Because they are not associated with any particular instance of a `ref class`, they can use only static member variables, which also are not associated with a particular instance. For the same reason, a static member method cannot be a virtual member method, as virtual member methods are also associated with `ref class` instances.

Coding static member methods is no different from coding normal member methods, except that the function declaration is prefixed with the `static` keyword.

Listing 3-4 uses a static member method to print out a static member variable. Oh, by the way, `WriteLine()` is also a static member method.

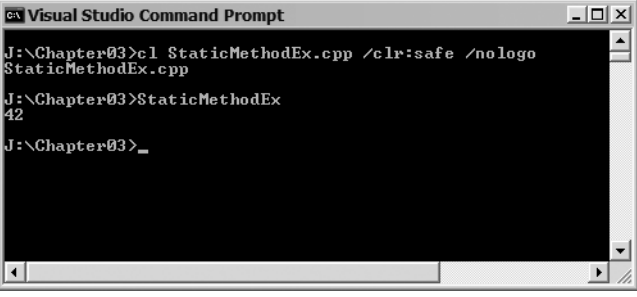
Listing 3-4. Static Member Methods and Variables in Action

```
using namespace System;

ref class StaticTest
{
private:
    static int x = 42;
public:
    static int get_x()
    {
        return x;
    }
};

void main()
{
    Console::WriteLine ( StaticTest::get_x() );
}
```

Figure 3-5 shows the results of this little program.



```
Visual Studio Command Prompt
J:\Chapter03>cl StaticMethodEx.cpp /clr:safe /nologo
StaticMethodEx.cpp
J:\Chapter03>StaticMethodEx
42
J:\Chapter03>_
```

Figure 3-5. Results of *StaticMethodEx.exe*

You might have noticed that to access the static member method, you use the `ref` class name and the `::` operator instead of the `.` or `->` operator. The reason is because you've not created an object, so you're effectively accessing the namespace tree.

ref class Constructors

The `ref` class *constructor* is a special `ref` class method that is different in many ways from the member method. In C++/CLI, a constructor is called whenever a new instance of a `ref` class is created. Instances of `ref` classes are created using the operator `gcnew`. Memory for the instance is allocated on the managed heap that is maintained by the CLR.

The purpose of the constructor is to get the object to an initialized state. There are two ways that the actual initialization process can take place, within the constructor or in a separate method that the constructor calls. Normally, you would use the first method if the `ref` class instance is only initialized once. The second method would be used if the `ref` class is reused and needs to be reinitialized at a later time. When this reinitialization takes place the initialization method is called, as you can't call the constructor method directly.

The `ref` class constructor process differs from the unmanaged class constructor process in that, for `ref` class constructors, all member variables are initialized to zero before the actual constructor is called. (Although this is helpful, initialization to zero is not always what you need. You might want to initialize to a specific value.) Thus, even if the constructor does nothing, all member variables will still have been initialized to zero or the data type's equivalent. For example, the `DateTime` data type initializes to 1/1/0001 12:00:00 a.m., which is this data type's equivalent of zero.

A `ref` class constructor method always has the same name as the `ref` class itself. A `ref` class constructor method does not return a value and must not be defined with the `void` return type. A constructor method can take any number of parameters. Note that a constructor method needs to have public accessibility to be accessible by the `gcnew` operator.

If no constructor method is defined for a `ref` class, then a default constructor method is generated. This constructor method does nothing of its own, except it calls the constructor method of its parent and sets all member variables to a zero value. If you define a constructor method, then a default constructor method will not be generated. Thus, if you create a constructor method with parameters and you expect the `ref` class to be able to be created without parameters, then you must manually create your own default zero-parameter constructor method.

A special construct of a constructor method is the *initializer list*. It's a list of variables that need to be initialized before the constructor method itself is called. You can use it to initialize the `ref` class's own variables as well; in fact, it's more efficient to do it this way, but it's also much harder to read in this format. The most common use of an initializer list is to initialize a parent `ref` classes by way of one of the parent's constructors. The syntax for an initializer list involves simply placing a

colon (:) and a comma-delimited list of functional notation variable declarations between the constructor method's declaration and the constructor method's implementation:

```
Constructor (int x, int y, int z) : var1(x, y), var2(z) { }
```

A new constructor type has been added in C++/CLI, the copy constructor. The copy constructor initializes a `ref class` object to be a copy of an existing `ref class` object of the same type. This type of constructor should not be anything new to developers who have already been coding in C++, except that the syntax has change a little. You now use the % operator instead of the & operator.

Listing 3-5 shows the constructors for a `ref class` called `ChildClass` inherited from `ParentClass`.

Listing 3-5. Constructors in Action

```
using namespace System;
```

```
// Parent Class
ref class ParentClass
{
public:
    // Default constructor that initializes ParentVal to a default value
    ParentClass() : PVal(10) { }

    // A constructor that initializes ParentVal to a passed value
    ParentClass(int inVal) : PVal(inVal) { }
    // Copy Constructor
    ParentClass(const ParentClass %p) : PVal(p.PVal) {}

    int PVal;
};

// Child class that inherits form ParentClass
ref class ChildClass : public ParentClass
{
public:
    // Default constructor that initializes ChildVal to a default value
    ChildClass () : CVal(20) {};    // default constructor

    // A constructor that initialized CVal to a passed value
    ChildClass(int inVal) : CVal(20) {};

    // A constructor that initialized the parent class with a passed value
    // and initializes ChildVal to a another passed value
    ChildClass (int inVal1, int inVal2) : ParentClass(inVal1), CVal(inVal2) { }

    // copy constructor
    ChildClass(const ChildClass %v) : ParentClass(v.PVal), CVal(v.CVal) { }

    int CVal;
};
```

```

void main()
{
    ParentClass p1(4);    // Constructor
    ParentClass p2 = p1;  // Copy Constructor

    p1.PVal = 2;          // Change original, new unchanged

    Console::WriteLine("p1.PVal=[{0}] p2.PVal=[{1}]", p1.PVal, p2.PVal);

    ChildClass ^c1 = gcnew ChildClass(5,6); // Constructor
    ChildClass c2 = *c1;                     // Copy Constructor

    c1->CVal = 12;          // Change original, new unchanged

    Console::WriteLine("c1=[{0}/{1}] c2=[{2}/{3}]",
        c1->PVal, c1->CVal, c2.PVal, c2.CVal);
}

```

Figure 3-6 shows the results of this little program.

```

J:\Chapter03>cl Constructors.cpp /clr:safe /nologo
Constructors.cpp
J:\Chapter03>Constructors
p1.PVal=[2] p2.PVal=[4]
c1=[5/12] c2=[5/6]
J:\Chapter03>

```

Figure 3-6. Results of *Constructors.exe*

Static ref class Constructors

In traditional C++, the syntax for initializing static member variables was rather cumbersome. It forced you to define it in the ref class and then initialize it outside the ref class before the `main()` function was called. You saw that with ref classes you could directly assign a value to a static member variable—but what happens if you need something more elaborate than a simple assignment? C++/CLI has provided a new construct for ref classes called the *static ref class constructor*.

The static ref class constructor's purpose is to initialize static member variables normally with something more complex than a simple assignment, but not necessarily. Any ref class can have a static ref class constructor, though it only really makes sense if the ref class has static member variables, because the static ref class constructor is not allowed to initialize any nonstatic member variables.

When the static ref class constructor is invoked it is undefined, but it is guaranteed to happen before any instances of the ref class are created or any references are made to any static members of the ref class.

If you recall, it is possible to initialize static member variables directly in the definition of the ref class. If you use the static ref class constructor, then these default values are overwritten by the value specified by the static ref class constructor.

The static `ref class` constructor syntax is identical to the default constructor syntax, except that the `static` keyword is placed in front. This means that a static `ref class` constructor cannot take any parameters.

In the following example, the `ref class` `Test` is made up of two static member variables initialized to 32 and a static `ref class` constructor that overwrites the first constant with the value of 42:

```
ref class Test
{
public:
    static Test()
    {
        value1 = 42;
    }
    static int value1 = 32;
    static int value2 = 32;
};
```

By the way, you can have both static and nonstatic (normal, I guess) constructor methods in the same `ref class`.

Destructors

Destructors serve two purposes in C++/CLI. The first is the deallocating of memory previously allocated to a heap by either the `new` or `gcnew` operator. The second purpose is the often overlooked releasing of system resources, either managed or unmanaged.

In versions prior to C++/CLI, Managed Extensions for C++ only handled the deallocating of managed memory and managed resources. To handle the releasing of unmanaged memory or unmanaged resources, it forced the developer to inherit from a .NET Framework interface `IDisposable`. If this means nothing to you, don't worry—things have gotten a whole lot easier.

Memory Management Destructors

All objects allocated on the managed heap using `gcnew` need to be deallocated. You have a choice about how to do this. You can call the `delete` operator on the handle of the `ref class` object, and the managed memory will be deallocated immediately in reverse order to which it was allocated. In other words, if the object allocated internal objects, then they would be deallocated first.

To start the deallocation process when a program is finished with the object, you simply call the `delete` operator on the object:

```
delete classname;
```

This is new to C++/CLI and is what is known as *deterministic cleanup*. The programmer now has complete control of when things are finally cleaned up.

The other choice is that the developer does not have to give up the garbage collection functionality of the CLR, if it doesn't matter when the memory is finally deallocated. In this case, the program is not required to call the `delete` operator; it simply has to do nothing. It will be the job of the CLR to detect when an object is no longer being accessed and then garbage collect it. In most cases, this choice is the best, and there is really no need to call the `delete` operator because CLR garbage collection works just fine.

The destructor method, which the `delete` operator calls, has the same syntax as the default constructor method except that a tilde (`~`) is placed before the destructor method's name:

```
~Test() {} // destructor
```

Note For the delete operator to be able to access the destructor, the destructor needs public access.

Within the destructor, you would call the delete operator for any objects that the `ref class` needs to clean up. If your `ref class` doesn't allocate anything while it was running, then there is no need to create a destructor as a default one will be generated for you.

Resource Management Destructors

Cleaning up managed resources like `String` or an `ArrayList` is handled just like managed memory resources.

Unmanaged resources, like open handles to files, kernel objects, or database objects, are a little trickier. The reason is if you don't specifically code the closing of these objects, then they don't get closed until the program ends or sometimes not until the machine is rebooted. Yikes!

Surprisingly, the CLR does not even have any explicit runtime support for cleaning up unmanaged resources. Instead, it is up to the programmer to implement a pattern for resource management based on a .NET Framework core interface `IDisposable`. The pattern is to place all resources that needed to be cleaned up within the `Dispose()` method exposed by the `IDisposable` interface and then call the `Dispose()` method when the resources are no longer needed. A little cumbersome, if you ask me.

With Managed Extensions for C++ version 1.1 and prior, that is exactly how you would have had to code unmanaged resource cleanup. In version C++/CLI, things have become a whole lot easier. With the addition of deterministic cleanup to C++/CLI, the delete operator now implements the `IDisposable` interface pattern automatically for you. Therefore, all you need to do to clean up your unmanaged resources is to add code to your `ref class`'s destructor and then call the delete operator when the object and the resources it's accessing are no longer needed. Simple, right? Well, there is a catch.

What if you forget to call the delete operator? The answer is, unfortunately, that the destructor and subsequently the `IDisposable` interface pattern are not called. This means the unmanaged resources are not cleaned up. Ouch! So much for .NET's great ability to clean up after itself, right?

Fortunately, this is not the end of the story. The CLR's garbage collection process has not yet occurred. This process will deallocate all managed objects whenever it gets around to that chore (you have no control of this). As an added bonus, C++/CLI has made things easier by providing an interface directly with the CLR garbage collection process (for when this finally does happen) called the `Finalize` destructor.

The `Finalize` destructor method is called by the CLR, when the CLR detects an object that needs to be cleaned up. An elegant solution, don't you think? Well, the elegance doesn't end there. The CLR, before it calls the `Finalize` destructor, checks to see if the delete operator has been already called on the object and, if so, does not even waste its time on calling the `Finalize` destructor.

What does this boil down to? You can clean up unmanaged resource yourself, or if you don't care when cleanup finally does occur (or you forget to do it), the CLR will do the cleanup for you. Nice, huh?

The `Finalize` destructor has the same syntax as the standard destructor, except an exclamation point (!) is used instead of a tilde (~), and it has to have protected access:

```
protected:
    !Test() {}    // Finalize destructor
```

Note The `Finalize` destructor must have protected access.

Here is how you code destructor logic, if you want all your bases covered for an object that has managed and unmanaged memory and resources to clean up:

```
ref class ChildClass : public ParentClass
{
public:
    ~Test()
    {
        // free all managed and unmanaged resources and memory
    }
protected:
    !Test()
    {
        // free all unmanaged resources and memory only
    }
}
```

The managed cleanup code is only found in the deterministic cleanup destructor, whereas unmanaged cleanup is found in both the deterministic cleanup and Finalize destructor. One thing you will find is that there is usually duplicate unmanaged memory and resource cleanup code in both of these destructors. Most likely, you will write an additional method, which these destructors call to eliminate this duplication.

Virtual Methods

Virtual methods are the cornerstone of polymorphism, as they allow different child ref classes derived from a common base ref class to respond to the same method call in a way specific to each child ref class. Polymorphism occurs when a virtual method is called through a base ref class handle. For example:

```
BaseClass ^BaseObject = gcnew ChildClass()
BaseObject->DoStuff() // will call the Child class version instead of the Base
                     // class version as long as DoStuff is declared a virtual.
```

This works because when the call is made, it is the type of the actual object pointed to that determines which copy of the virtual method is called.

Technically, when you declare a virtual method, you are telling the compiler that you want dynamic or runtime binding to be done on any method with an identical signature in a derived ref class. To make a method virtual, you simply need to place the keyword `virtual` in front of the method declaration.

```
virtual void Speak () {}
```

Any method that you declare as virtual will automatically be virtual for any directly or indirectly derived ref class.

Normally, in a standard virtual animal example, you would first declare a base ref class `Animal` with a virtual method of `Speak()`. You then create specific animal-type ref classes derived from `Animal` and override the virtual method `Speak()`. In the `main()` function, you would create an array of `Animal` objects and assign specific animal derived objects to it. Finally, you would loop through the `Animal` array. Because the `Speak()` method is virtual, the actual object type assigned to the `Animal` array determines which `Speak()` to execute.

There are two methods of overriding a virtual function: implicit and explicit (or named). You can also hide the virtual override sequence and start a new one, or you can simply stop the virtual sequence altogether.

Implicit Virtual Overriding

For implicit overriding, the method signature of the base `ref class` must be the same as the derived `ref class` including the prefix `virtual`. This means that the name of the method and the number of parameters and their types must be identical. The return type of the method need not be identical, but it must at least be derived from the same type as that of the base method's return type. Also, you need to append the new keyword `override` after the parameters:

```
virtual void Speak () override
{
}
```

Hiding Virtual Overriding

Usually, if a parent defines a method as `virtual`, there is usually a good reason. I haven't yet had a reason to do this, but if you really want to overrule a parent `ref class` and hide a method from propagating virtual overriding to its children, you can. To do so, add the keyword `new` after the method declaration:

```
void Speak() new
{
}
```

You can also hide virtual overriding propagation and start a new one from the current `ref class` by making the above member method `virtual`:

```
virtual void Speak() new
{
}
```

To me, both of these method declarations go against proper OOP, but they are available if there is good reason.

Explicit or Named Virtual Overriding

Explicit or named overriding allows you to assign a method with a different name to a virtual function. To do this, you need to declare the overriding method as `virtual` and then assign the name of the virtual method being overridden:

```
ref class Puppy : public Dog
{
public:
    virtual void Yip () = Dog::Speak
    {
    }
};
```

One handy feature of explicit overriding is that it allows you to overcome a virtual sequence that has been hidden by the `new` operator, as an explicit overriding does not need to override a direct parent. It can override an indirect parent, occurring before the virtual method sequence hide point, by specifying the `ref class` name of a grandparent (or great grandparent, or ...) along with the method being overridden:

```

ref class Tiger : public Cat
{
public:
    virtual void Growl () = Animal::Speak
    {
    }
};

```

An even cooler feature of explicit virtual overriding is that you can actually continue two different virtual method sequences from a single virtual method. You do this by explicitly overriding a sealed sequence, which has been declared virtual to start a new sequence, using the same method name as the new sequence and then adding an explicit virtual override. Explaining it is a bit confusing, but an example should make things clearer:

```

ref class Animal
{
public:
    virtual void Speak ()
    {
        Console::WriteLine("Animal is Mysteriously Silent");
    }
}

ref class Cat : public Animal
{
public:
    virtual void Speak() new // sequence hidden and a new one created
    {
        Console::WriteLine("Cat says Meow");
    }
};

ref class Tiger : public Cat
{
public:
    virtual void Speak() override = Animal::Speak //both sequences continue here
    {
        Console::WriteLine("Tiger says Growl");
    }
};

```

You can also do the same thing using a comma-delimited list of methods you want to explicitly override with this one method. However, in this case the new virtual method needs a different name, or the compiler will complain a little bit:

```

ref class Tiger : public Cat
{
public:
    virtual void Growl() = Animal::Speak, Cat::Speak
    {
        Console::WriteLine("Tiger says Growl");
    }
};

```

It probably is obvious, but like implicit overriding, explicit overriding requires that the signature of the overriding method must match the virtual method being overwritten.

Listing 3-6 is not your standard virtual animal example. It's a very contrived example, trying to show all the different statements associated with virtual methods.

Listing 3-6. *Virtual Methods in Action*

```
using namespace System;

ref class Animal
{
public:
    virtual void Speak ()
    {
        Console::WriteLine("Animal is Mysteriously Silent");
    }
};

ref class Dog : public Animal
{
public:
    // Standard explicit virtual override
    virtual void Speak() override
    {
        Console::WriteLine("Dog says Woof");
    }
};

ref class Puppy : public Dog
{
public:
    // Yip name overrides dog's virtual speak
    virtual void Yip() = Dog::Speak
    {
        Console::WriteLine("Puppy says Yip Yip");
    }
};

ref class Cat : public Animal
{
public:
    // Start a new speak virtual sequence so animal's virtual speak fails
    virtual void Speak() new
    {
        Console::WriteLine("Cat says Meow");
    }
};

ref class Tiger : public Cat
{
public:
    // Though inherited from cat, Tiger name overrides Animal's speak
    // thus, can speak though animal virtual sequence
    // also this method overrides Cat's virtual Speak method as well
```

```

        virtual void Growl() = Animal::Speak, Cat::Speak
        {
            Console::WriteLine("Tiger says Growl");
        }
    };

void main()
{
    // Array of Animal handles
    array<Animal^>^ animals = gcnew array<Animal^>
    {
        gcnew Animal(),
        gcnew Dog(),
        gcnew Puppy(),
        gcnew Cat(),
        gcnew Tiger()
    };

    for each ( Animal ^a in animals)
    {
        a->Speak();
    }

    Console::WriteLine();

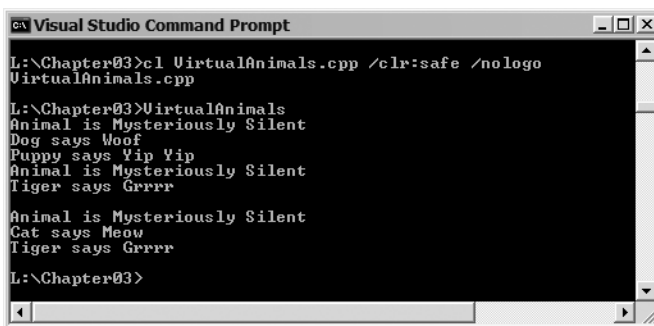
    Animal^ cat1 = gcnew Cat();
    Cat^ cat2 = gcnew Cat();
    Cat^ tiger = gcnew Tiger();

    // new cancels virtual sequence of Animal
    cat1->Speak();

    // new speak sequence established for cat
    cat2->Speak();
    tiger->Speak();
}

```

Figure 3-7 shows the results of this little program.



```

C:\ Visual Studio Command Prompt
L:\Chapter03>cl VirtualAnimals.cpp /clr:safe /nologo
VirtualAnimals.cpp
L:\Chapter03>VirtualAnimals
Animal is Mysteriously Silent
Dog says Woof
Puppy says Vip Vip
Animal is Mysteriously Silent
Tiger says Grrrr
Animal is Mysteriously Silent
Cat says Meow
Tiger says Grrrr
L:\Chapter03>

```

Figure 3-7. Results of *VirtualAnimals.exe*

Pure Virtual Method

When you look at the previous example, you may notice that the base `ref class` virtual method `Speak()` really has a nonsense implementation and shouldn't even be included in the `ref class`. A better way to implement this example and ensure that the virtual method is always overridden is to force the inheriting `ref classes` to override the virtual method and, if they don't, generate an error. You can do this with a *pure virtual method*.

The big difference between a pure virtual method and a virtual method is that a `ref class` that contains pure virtual methods cannot be instantiated. In other words, a `ref class` that has pure virtual methods must be inherited to be used. I cover this in more detail later in the chapter in the section about abstract `ref classes`.

A pure virtual method is simply a definition of a method without any implementation. When you use it, the compiler checks to make sure that the pure virtual method is overwritten. If it is not, then the compiler generates an error.

A pure virtual method has the same syntax as a regular virtual method, except that instead of a method implementation, `a = 0;` is appended:

```
virtual void PureVirtualFunction() = 0;
```

Caution You cannot hide a pure virtual method with the `new` operator.

Method Overriding

Method overriding is defining a method in a derived `ref class` that has an identical signature to the base `ref class`. How the derived `ref class` actually works depends on whether the method is virtual or not. If the method is virtual, it runs as I described previously.

On the other hand, if the method is not virtual, then it works in a completely different way, because polymorphism does not come into effect at all. First, no dynamic binding occurs, only standard static or compile-time binding. What this means is that whatever type the method is called with is executed. For example, in the `VirtualAnimal` example, if the `Speak()` method were not virtual, then the `Animal` `ref class`'s `Speak()` method would be called every time in the `for` each loop. This displays "Mysterious Silence" every time as opposed to the assorted messages generated by the virtual version of the example. The reason this happens is because the array is of type `Animal`.

To get each animal to speak now, you must create instances of each type of animal and call that animal's `speak()` method directly. Overriding a nonvirtual method simply has the effect of hiding the base `ref class`'s copy of the method.

Method Overloading

There is nothing special about coding overloaded methods, given that they are handled in exactly the same way as function overloading, which I covered in the previous chapter. The only real difference is that they are now methods inside a `ref class` and not functions out on their own. For example, here is the same `supersecret` method (this time) overloaded three times in a `Secret` `ref class`:

```
ref class Secret
{
    int Test () { /* do stuff */ }
    int Test (int x) { /* do stuff */ }
    int Test (int x, int y, double z) { /* do stuff */ }
};
```

Calling an overloaded method is nothing special either. Simply call the method you want with the correct parameters. For example, here is some code to call the second supersecret Test method from a handle called `secret` and the third method from a stack object:

```
secret->Test (0, 1, 2.0); // handle
secret.Test(5);          // local stack
```

For those of you coming from a traditional C++ or Visual Basic background, you might have used default arguments. Unfortunately, with C++/CLI, `ref` classes do not support default arguments in member methods. In fact, they generate an error.

A suggested solution to this change in syntax is to use overloaded methods. That is, define a method with fewer parameters and then initialize the variable in the method body. For example, here are four methods that when combined are equivalent to one method with three defaulted arguments:

```
ref class NoDefaultArgs
{
    // Invalid method with default values
    // int DefArgs (int x = 1, int y = 2, int z = 3) { /* do stuff */ }

    // Equivalent combination of overloaded methods
    int DefArgs ()
    {
        x = 1;
        y = 2;
        z = 3;
        /* do stuff */
    }
    int DefArgs ( int x )
    {
        y = 2;
        z = 3;
        /* do stuff */
    }
    int DefArgs ( int x, int y )
    {
        z = 3;
        /* do stuff */
    }
    int DefArgs ( int x, int y, int z )
    {
        /* do stuff */
    }
}
```

I'm sure there is a good reason why Microsoft eliminated default arguments, but personally, I hope it puts them back in, because using overloads can get quite cumbersome.

Managed Operator Overloading

Operator overloading is one important feature that most traditional C++ programmers learn to work with early in their careers. It is one of C++'s claims to fame. Operator overloading is the ability to use standard operators and give them meaning in a `ref` class—for example, adding two strings together to get a new concatenated string.

C++/CLI's `ref` classes support operator overloading as well, but in a slightly different syntax than traditional C++. The major difference in the syntax revolves round the aspect that to support the .NET Framework's feature of multiple (computer) language support, managed operator overloads must be declared as static. Also as a consequence of this, binary operators must pass both the left- and right-hand sides of the operator as parameters, and unary operators must pass the left-hand side of the operator as a parameter. This contrasts with the traditional operator overloading where the parameters are declared as member variables, and one fewer parameter is passed because the other parameter is an instance variable.

Therefore, traditional operator overloading syntax for the multiplication operator looks like this:

```
OpClass^ operator *(const OpClass ^rhs);
```

whereas managed operator overloading syntax for the multiplication operator looks like this:

```
static OpClass^ operator *(const OpClass ^lhs, const OpClass ^rhs);
```

One thing to keep in mind is that traditional operator overloading syntax is only supported within the C++/CLI language environment, because this syntax does not adhere to the requirements of being static and passing all operands.

A convenient feature of managed operator overloading for veteran C++ developers is that if you will never support multiple languages, then you can still use the traditional syntax that you are accustomed to. Or, if you do plan to support multiple languages, you can use the managed operator overloading syntax only with `ref` class for which you plan to support multiple languages and use traditional for the rest. Personally, I stick to the managed operator overloading syntax because, you never know, in the future, you might need multilanguage support.

Caution You cannot use both traditional and managed syntaxes for operator overloading for the same operator within the same `ref` class.

Not all operators can be overloaded. The most notable missing operators are the open and closed square brackets `[]`, open and closed round brackets `()`, `gcnew`, `new`, and `delete`. The Table 3-1 is a list of the operators available to be overloaded.

Table 3-1. *Supported Managed Operators*

Operators				
+	-	*	/	%
^	&		~	!
=	<	>	+=	--
*=	/=	%=	^=	&=
=	<<	>>	<<=	>>=
==	!=	<=	>=	&&
	++	--	,	

There are two types of operator overloads: unary and binary. You would have a good case, if you claimed that the increment and decrement operators are a third type of operator. As you will see,

the increment and decrement operator syntax is the same as that of a unary operator. It is only the implementation of the operators that makes them different.

Overloading Unary Operators

Unary operators, if you recall from the previous chapter, are operators that take only one operand. With built-in operators, except for the increment and decrement operators, the operand themselves are not changed by the operator. For example, when you place the negative operator in front of a number (operand), the number (operand) does not change and become negative, though the value returned is negative:

```
int i = 1;
int j = -i;    // j = -1 but i = 1
```

With managed operators, unlike their built-in arithmetic equivalent, you have a little more power over how unary operators actually work. You can make the operands mutable if you want, though, this could be dangerous, as most developers would not expect functionality. To ensure that the operand is not mutable, you should use the `const` operator in the argument of the operator overload method:

```
static OpClass^ operator -(const OpClass ^lhs)
{
    OpClass^ ret = gcnew OpClass();
    ret->i = -(lhs->i);
    return ret;
}
```

With the `const` operator, the compiler will fail if the `lhs` argument is changed within the body of the method. On the other hand, if you want the argument to change during the operation then you would leave off the `const` operator:

```
static OpClass^ operator -(OpClass ^lhs)
{
    lhs->i = -(lhs->i);
    return lhs;
}
```

The preceding is mutability and is probably not what you want but it is available if there is a need. In fact, this mutability was how I thought the increment and decrement was implemented but I found out instead that the compiler generates code specifically for the operators. Here is how you could implement an increment operator. Notice the `const` operator in the argument:

```
static OpClass^ operator ++(const OpClass ^lhs)
{
    OpClass^ ret = gcnew OpClass();
    ret->i = (lhs->i) + 1;
    return ret;
}
```

Overloading Binary Operators

A binary operator takes two operands; what you do with this operand when it comes to managed operator overloading is totally up to you. For example, you could return a `bool` for a logical operator, a result object for an arithmetic operator, or return a `void` and mutate the first argument for an assignment operator.

Here are each of these types of operation using the same operator.

Note You can't place all these overloads in one `ref class` as there would be method ambiguity, but each is perfectly valid if implemented uniquely in a `ref class`. On the other hand, the user of these operators would most likely be completely confused by the first two examples as they go against how they are normally used.

- Logical operator (unexpected implementation):

```
static bool operator *=(const OpClass ^lhs, const OpClass ^rhs)
{
    return lhs->i == rhs->i;
}
// ...
bool x = op1 *= op2;
```

- Arithmetic operator (unexpected implementation):

```
static OpClass^ operator *=(const OpClass ^lhs, const OpClass ^rhs)
{
    OpClass^ ret = gcnew OpClass();
    ret->i = lhs->i * rhs->i;
    return ret;
}
// ...
OpClass^ x = y *= z;
```

- Assignment operator (expected implementation):

```
static void operator *=(OpClass ^lhs, const OpClass ^rhs)
{
    lhs->i *= rhs->i;
}
// ...
x *= y;
```

By the way, you could even implement these operators a fourth way by having the operator overload return a different type than Boolean or the operator `ref class`. For example:

```
static int operator *=(const OpClass ^lhs, const OpClass ^rhs) {}
```

Listing 3-7 is an example of assorted managed operator overloads.

Listing 3-7. Operator Overload in Action

```
using namespace System;
```

```
ref class OpClass
{
public:
    OpClass() : i(0) {}
    OpClass(int x) : i(x) {}

    // x != y
    static bool operator !=(const OpClass ^lhs, const OpClass ^rhs)
    {
        return lhs->i != rhs->i;
    }
}
```

```

// x * y
static OpClass^ operator *(const OpClass ^lhs, const OpClass ^rhs)
{
    OpClass^ ret = gcnew OpClass();
    ret->i = lhs->i * rhs->i;
    return ret;
}

// x *= y
static void operator *=(OpClass ^lhs, const OpClass ^rhs)
{
    lhs->i *= rhs->i;
}

// -x
static OpClass^ operator -(const OpClass ^lhs)
{
    OpClass^ ret = gcnew OpClass();
    ret->i = -(lhs->i);
    return ret;
}

// ++x and x++
static OpClass^ operator ++(const OpClass ^lhs)
{
    OpClass^ ret = gcnew OpClass();
    ret->i = (lhs->i) + 1;
    return ret;
}

virtual String ^ ToString() override
{
    return i.ToString();
}
private:
    int i;
};

void main()
{
    OpClass ^op1 = gcnew OpClass(3);
    OpClass ^op2 = gcnew OpClass(5);
    OpClass ^op3 = gcnew OpClass(15);

    if ( op1 * op2 != op3)
        Console::WriteLine("Don't Equal");
    else
        Console::WriteLine("Equal");

    op1 *= op2;
    Console::WriteLine(op1);
}

```

```

Console::WriteLine(++op1); // prints 15 then increments to 16
Console::WriteLine(op1++); // increOpClassents to 17 then prints

Console::WriteLine(-op1); // Negation of OpClass1
Console::WriteLine(op1);   // prior Negation op left OpClass1 unchanged
}

```

Figure 3-8 shows the results of this little program.

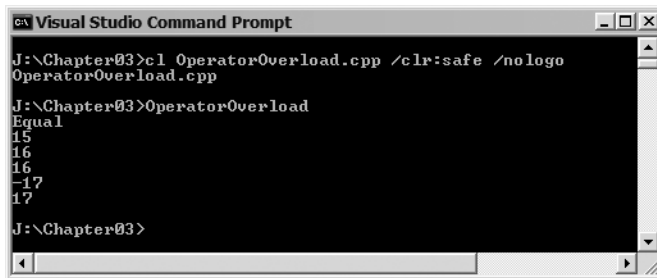


Figure 3-8. Results of *OperatorOverload.exe*

Both operands don't have to be of the same type as the defining ref class, but at least one of the managed operands must be of the same type as the defining ref class. Defining a managed operands with one argument, something other than the defining ref class, is not automatically associative. You must define the other combination as well. Listing 3-8 compares whether the ref class *Number* is greater than an *int* associatively.

Listing 3-8. *Operator Overload for Mixed Data Types in Action*

```
using namespace System;
```

```

ref class Number
{
public:
    Number(int x) : i(x) {}

    static bool operator >(Number^ n, int v) // maps to operator >
    {
        return n->i > v;
    }
    static bool operator >(int v, Number^ n) // maps to operator >
    {
        return v > n->i;
    }

    virtual String ^ ToString() override
    {
        return i.ToString();
    }
private:
    int i;
};

```

```

int main()
{
    Number^ n = gcnew Number(5);

    if ( n > 6 )
        Console::WriteLine("{0} Greater than 6", n);
    else
        Console::WriteLine("{0} Less than or Equal 6", n);

    if ( 6 > n )
        Console::WriteLine("6 Greater than {0}", n);
    else
        Console::WriteLine("6 Less than or Equal {0}", n);
}

```

Figure 3-9 shows the results of this little program.

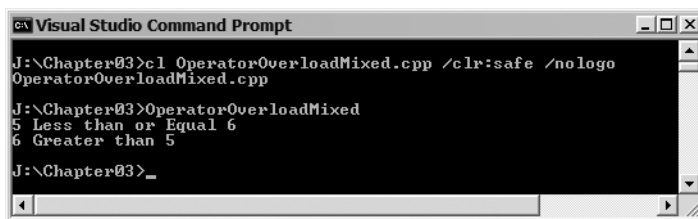


Figure 3-9. Results of *OperatorOverloadMixed.exe*

Member Properties

The purpose of properties is to enrich encapsulation for *ref* classes by hiding member variables, yet at the same time providing protected access to the values contained in these variables. Properties are successful in doing this, and as an added benefit, they provide an improved and simplified interface to a member variable.

A problem with traditional C++ classes is that there is no simple and standardized way of maintaining member variables. Frequently, programmers will simplify the syntax of interfacing with their class and allow public access to member variables even at the risk of having invalid data placed into them. Seeing the risk of exposing some of the more volatile variables, a programmer then might decide to write “getter and setter” methods. These methods protect the member variables but also complicate the necessary syntax for their access, because you always have to access the variables as a function call instead of using the more intuitive variable access format.

Properties solve this problem by providing direct member variable-like access to member variables, but with the security and flexibility of getter and setter methods. To the programmer accessing the *ref* class, properties act like member variables. Member properties resemble simple scalar variables, static variables, arrays, and indexes. To the developer of the *ref* class, properties are simply getter and setter methods with specific rules and syntax. The complexity of these methods is totally up to the *ref* class creator.

Trivial Properties

Trivial properties are the most common implementation of a property with getter and setter methods accessing a single member variable without any additional actions being done on that member variable. You might consider the trivial property as a placeholder for future enhancement to the *ref* classes

API as it enables a `ref class` to maintain binary compatibility when a more elaborate property evolves. If you were to initially code the API as directly accessing the member variable, then in the future, you would lose binary compatibility if you changed the API access to properties.

Coding trivial properties is, as the name suggests, trivial. Simply declare the member variable with a prefix of the keyword `property`.

```
property type PropertyName;
```

You would then access the property just as you would any other member variable but, under the covers, you are actually accessing the variable as a property.

Scalar Properties

One step up from trivial properties is the scalar property. This form of property allows the ability to provide read-only, write-only, or both read and write access to a member variable. It also allows doing things like validating the property before updating its underlying member variable or logging all changes made to the property.

To create a scalar property with read and write access, you need to extend the trivial property syntax by adding a `get()` and `set()` method:

```
property type PropertyName
{
    type get() {};
    void set (type value) {};
}
```

You can make a property write-only by excluding the `set` method in the property's declaration:

```
property type PropertyName
{
    type get() {};
}
```

Conversely, you can make the property read-only by excluding the `get` method:

```
property type PropertyName
{
    void set (type value) {};
}
```

The `get()` method gives you full access to the property to do as you please. The most common thing you will do is validate the parameter and then assign it to a private member variable.

The only real catch you might encounter is that the property name cannot be the same as a member variable. A conversion I use, which is by no means a standard, is to use a lowercase letter as the first letter of the member variable and an uppercase letter as the first letter of the property name.

With the addition of a `set()` method, you are now free to put any calculation you want within the method, but it must return the type specified. For this type of property, the most common body of the method is a simple return of the member variable storage of the property.

Listing 3-9 shows a trivial property, and scalar properties that are readable, writable, and both readable and writable.

Listing 3-9. *Scalar Properties in Action*

```
using namespace System;

ref class ScalarProp
{
```

```

public:
    // Constructor
    ScalarProp()
    {
        Cost        = 0.0;
        number       = 0;
        name         = "Blank Name";
        description   = "Scalar Property";
    }

    // trivial property
    property double Cost;

    // Read & write with validated parameter
    property int Number
    {
        void set(int value)
        {
            if (value < 1)
                value = 1;
            else if (value > 10)
                value = 10;

            number = value;
        }

        int get()
        {
            return number;
        }
    }

    // Write-only property
    property String^ Name
    {
        void set(String^ value)
        {
            name = value;
        }
    }

    // Ready-only property
    property String ^Description
    {
        String^ get()
        {
            return String::Concat(name, " ", description);
        }
    }
private:
    String ^name;
    String ^description;
    int    number;
};

```

```

void main()
{
    ScalarProp sp;

    sp.Name = "The Ref Class";

    Console::WriteLine(sp.Description);

    sp.Cost = 123.45;
    Console::WriteLine(sp.Cost);

    sp.Number = 20;    // Will be changed to 10
    Console::WriteLine(sp.Number);

    sp.Number = -5;    // Will be changed to 1
    Console::WriteLine(sp.Number);

    sp.Number = 6;    // Will not change
    Console::WriteLine(sp.Number);
}

```

Figure 3-10 shows the results of this program.

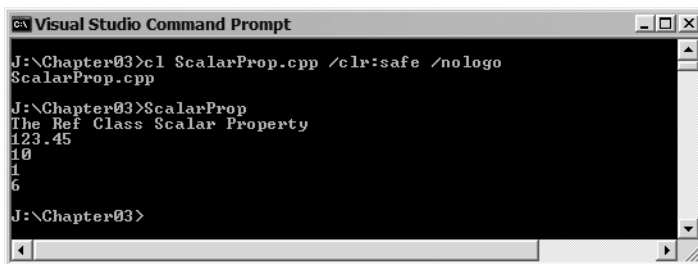


Figure 3-10. Results of *ScalarProp.exe*

Static Properties

As I mentioned previously, ref classes also contain static member variables. Likewise, C++/CLI provides property syntax to support *static properties*, or properties that have ref class-wide storage.

Static properties are nearly identical to scalar properties except that they contain the keyword `static` in their definition and they can only use static variables for storage. To create a readable and writable static property, simply use this syntax:

```

property static type PropertyName
{
    type get() {};
    void set (type value) {};
}

```

For example:

```

property static String^ Name
{
    void set(String^ value)
    {
        name = value;
    }
    String^ get()
    {
        return name;
    }
}

```

You can optionally place the keyword `static` in front of the `get()` and `set()` method, but I personally find this redundant.

Programmers can access a static property in the same way they would a static member variable, by using `ref class name` and the `::` operator:

```
class::PropertyName
```

For example:

```
StaticProp::Name = "Static Property";
Console::WriteLine(StaticProp::Name);
```

Listing 3-10 shows a simple readable and writable static `Name` property.

Listing 3-10. *Static Properties in Action*

```
using namespace System;
```

```

ref class StaticProp
{
public:
    property static String^ Name
    {
        void set(String^ value)
        {
            name = value;
        }
        String^ get()
        {
            return name;
        }
    }
private:
    static String^ name;
};

int main()
{
    StaticProp::Name = "Static Property";
    Console::WriteLine(StaticProp::Name);
}

```

Figure 3-11 shows the results of this little program.

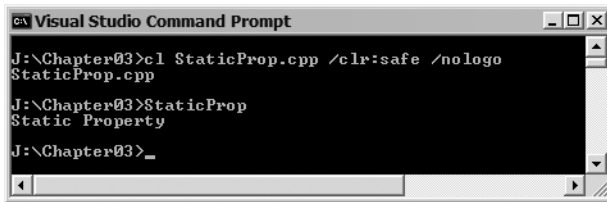


Figure 3-11. Results of *StaticProp.exe*

Array Properties

C++/CLI provides simple array syntax for properties. This is a big improvement over traditional C++, where getter and setter methods simply don't perform that elegantly.

The syntax for array properties is the same as that for the scalar property, except that the property's type is an array:

```
property array<type>^ NumArray
{
    array<type>^ get() {}
    void set ( array<type>^ value ) {}
}
```

For example:

```
property array<int>^ NumArray
{
    array<int>^ get() {}
    void set ( array<int>^ value ) {}
}
```

Once the `get()` and `set()` methods have been created, it is a simple matter to access an array property using normal array syntax. Listing 3-11 shows how to add a readable and writable array property to a `ref` class.

Listing 3-11. Array Properties in Action

```
using namespace System;

ref class ArrayProp
{
public:
    ArrayProp(int size)
    {
        numArray = gcnew array<int>(size);
    }

    property array<int>^ NumArray
    {
        array<int>^ get()
        {
            return numArray;
        }
    }
}
```

```

        void set ( array<int>^ value )
        {
            numArray = value;
        }
    }
private:
    array<int>^ numArray;
};

void main()
{
    ArrayProp aprop(5);

    for ( int i = 0 ; i < aprop.NumArray->Length ; ++i )
        aprop.NumArray[i] = i;

    for each (int i in aprop.NumArray)
        Console::WriteLine(i);
}

```

Figure 3-12 shows the results of this little program.

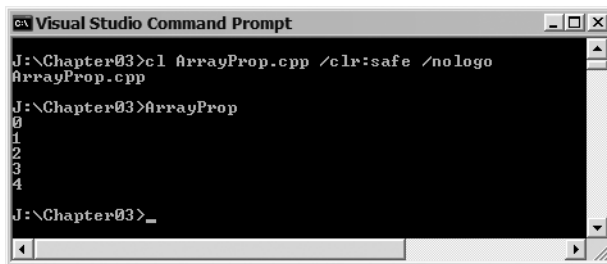


Figure 3-12. Results of *ArrayProp.exe*

Indexed Properties

At first glance, *indexed properties* may appear to provide the same functionality as array properties. They allow you to look up a property based on an index. The syntax to allow you to do this is more complex than that of the array property:

```

property type PropertyName [ indexType1, ..., indexTypeN ]
{
    type get(indexType1 index1, ..., indexTypeN indexN) {};
    void set(indexType1 index1, ..., indexTypeN indexN, type value) {};
}

```

Here is an example of two indices being used in an indexed property:

```

property AType^ PropertyName [ int, int ]
{
    AType^ get(String^ index1, int index2) {};
    void set(String^ index1, int index2, AType^ value) {};
}

```

So why would a programmer go through all of the problems of using indexed properties? It boils down to one thing: The index doesn't have to be numeric. In other words, when you use indexed properties, you get the ability to work with an array index of any type.

In the preceding sample, the index is of type `String^`. So, when programmers want to access an indexed property, they would access it like this:

```
PropertyName["StringValue", intValue]
```

If the indexed properties are still a little hazy, Listing 3-12 is a more complex example to show them in action. You start by defining a `Student` ref class with two trivial properties. You then create a `Course` ref class, which, using a nested ref class (covered next), stores a linked list of students and their grades for the course. You use an indexed property `ReportCard` to extract the grades from the linked list using the student's name.

Listing 3-12. Indexed Properties in Action

```
using namespace System;

ref class Student
{
public:
    Student(String^ s, int g)
    {
        Name = s;
        Grade = g;
    }

    property String^ Name;
    property int Grade;
};

ref class Course
{
    ref struct StuList
    {
        Student ^stu;
        StuList ^next;
    };
    StuList ^Stu;
    static StuList ^ReportCards = nullptr;

public:
    property Student^ ReportCard [String^]
    {
        Student^ get(String^ n)
        {
            for(Stu = ReportCards; Stu && (Stu->stu->Name != n); Stu = Stu->next)
                ;
            if (Stu != nullptr)
                return Stu->stu;
            else
                return gcnew Student("",0); // empty student
        }
    }
};
```

```

void set(String^ n, Student^ s)
{
    for(Stu = ReportCards; Stu && (Stu->stu->Name != n); Stu = Stu->next)
        ;
    if (Stu == nullptr)
    {
        StuList ^stuList = gcnew StuList;
        stuList->stu = s;
        stuList->next = ReportCards;
        ReportCards = stuList;
    }
}
};

void main()
{
    Course EnglishLit;
    Student Stephen("Stephen", 95);           // student as stack variable
    Student ^Sarah = gcnew Student("Sarah", 98); // student as heap variable

    EnglishLit.ReportCard[ "Stephen" ] = %Stephen;    // index as String literal
    EnglishLit.ReportCard[ Sarah->Name ] = Sarah;     // index as String^

    Console::WriteLine(EnglishLit.ReportCard[ Stephen.Name ]->Grade);
    Console::WriteLine(EnglishLit.ReportCard[ "Sarah" ]->Grade);
}

```

Figure 3-13 shows the results of this little program.

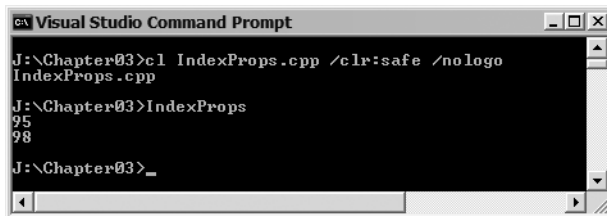


Figure 3-13. Results of *IndexProps.exe*

Default Indexed Property (Indexer)

Scalar properties provide fieldlike access on an instance of an object. A default indexed property, however, allows arraylike access directly on specific collection within an instance of an object. The default indexed property is a convenience, simplifying the access to a selected (default) collection within a `ref` class.

The syntax of a default indexed property is identical to an indexed property, except that the keyword `default` is used in place of the name of the property. That way, when you want to access the default collection within the `ref` class, you omit the property name and just reference the instance of the object as if it were the default collection itself.

Listing 3-13 is a simple example of a default index property where the default collection called `defaultArray` is coded to be the default index property.

Listing 3-13. *Indexed Properties in Action*

```

using namespace System;

ref class Numbers
{
public:
    Numbers()
    {
        defaultArray = gcnew array<String^>
        {
            "zero", "one", "two", "three", "four", "five"
        };
    }

    property String^ default [int]
    {
        String^ get(int index)
        {
            if (index < 0)
                index = 0;
            else if (index > defaultArray->Length)
                index = defaultArray->Length - 1;

            return defaultArray[index];
        }
    }
private:
    array<String^>^ defaultArray;
};

void main()
{
    Numbers numbers;

    Console::WriteLine(numbers[-1]);
    Console::WriteLine(numbers[3]);
    Console::WriteLine(numbers[10]);
}

```

Figure 3-14 shows the results of this little program.

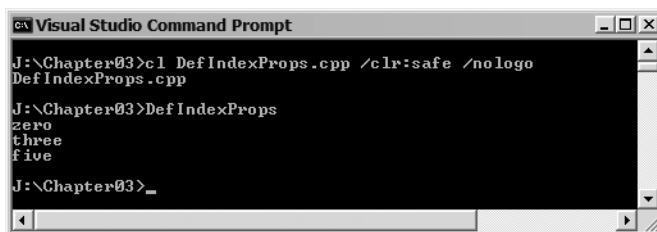


Figure 3-14. Results of *DefIndexProps.exe*

By the way, there is no restriction that the index be an integer. Just like an indexed property, a default indexed property can be any type and number of indexes.

Nested ref classes

As their name suggests, *nested* ref classes are ref classes defined inside another ref class. You might think of them as member ref classes.

Nested classes differ from inherited classes in that inherited classes have an “is a” relationship, whereas nested classes have a “contains a” relationship. In other words, for inheritance class A “is a” class B, and for nested classes, class C “contains a” class D. Of course, you can always use a separate class instead of a nested class to do this. What you gain by using nested classes is context, because a nested class only has context within the class containing it.

I very seldom use nested classes, but they do make sense if the nested class only really has meaning within its container class.

Like all members of a class, a nested class’s accessibility is determined by whether it is located within the public, protected, or private area of its class. Unlike member types, a nested class, though limited to the scope of the enclosing class, has its own members, and these members adhere to the accessibility of the nested class. For example, if the nested class has public accessibility, but the accessibility of the nested class’s member variable is private, then the member variable is private as far as the surrounding class is concerned, even though the nested class is accessible to external functions and methods.

In Listing 3-14, you can see a surrounding class with a nested class. The nested class has three members: a public, a protected, and a private member variable. The surrounding class has three member variable references to the nested class: public, protected, and private. The surrounding class also has an initializer list constructor for the member variables and a method to access all the nested class instances within the surrounding class. The listing shows an inheriting class to the surrounding class with a method showing how to access the nested class instances of its parent class. Finally, the listing shows a `main()` function that indicates how to reference the member variable found within the nested class within the surrounding class. The class has no output. Its purpose is to show you a method of accessing nested classes’ public members.

Listing 3-14. *Nested Classes in Action*

```
using namespace System;

ref class SurroundClass
{
public:
    ref class NestedClass        // Declaration of the nested class
    {
    public:
        int publicMember;
    protected:
        int protectedMember;
    private:
        int privateMember;
    };

    NestedClass^ protectedNC;    // protected variable reference to NestedClass

private:
    NestedClass^ privateNC;      // private variable reference to NestedClass
```

```

public:
    NestedClass^ publicNC;      // public variable reference to NestedClass

    // Constructor for SurroundClass
    // Notice the initializer list declaration of the reference member variable
    SurroundClass() : publicNC(gcnew NestedClass),
                     protectedNC(gcnew NestedClass),
                     privateNC(gcnew NestedClass)
    {}

    // A member showing how to access NestedClass within SurroundClass
    // Notice only public member variables of the nested class are accessed
    // The private and protected are hidden
    void method()
    {
        int x;

        NestedClass nc1;      // Declared another reference NestedClass

        x = nc1.publicMember;  // Accessing new NestedClass variable

        x = publicNC->publicMember; // Accessing public NestedClass variable
        x = protectedNC->publicMember; // Accessing protected NestedClass variable
        x = privateNC->publicMember; // Accessing private NestedClass variable
    }
};

// A inherited class showing how to access NestedClass within a member method
// Notice only public and protected NestedClass are accessed
// The private is hidden
ref class inheritSurroundClass : public SurroundClass
{
public:
    void method()
    {
        int x;

        NestedClass nc1;      // can access because NestedClass
                               // declaration protected

        x = nc1.publicMember;

        x = publicNC->publicMember;
        x = protectedNC->publicMember;
    }
};

// The main function shows how to access NestedClass from outside SurroundClass
// inheritance tree
// Notice only the public NestedClass reference is accessible
void main()
{
    SurroundClass sc;
    int x = sc.publicNC->publicMember;
}

```

There is a lot of code in Listing 3-14. Figure 3-15 should clear up any confusion.

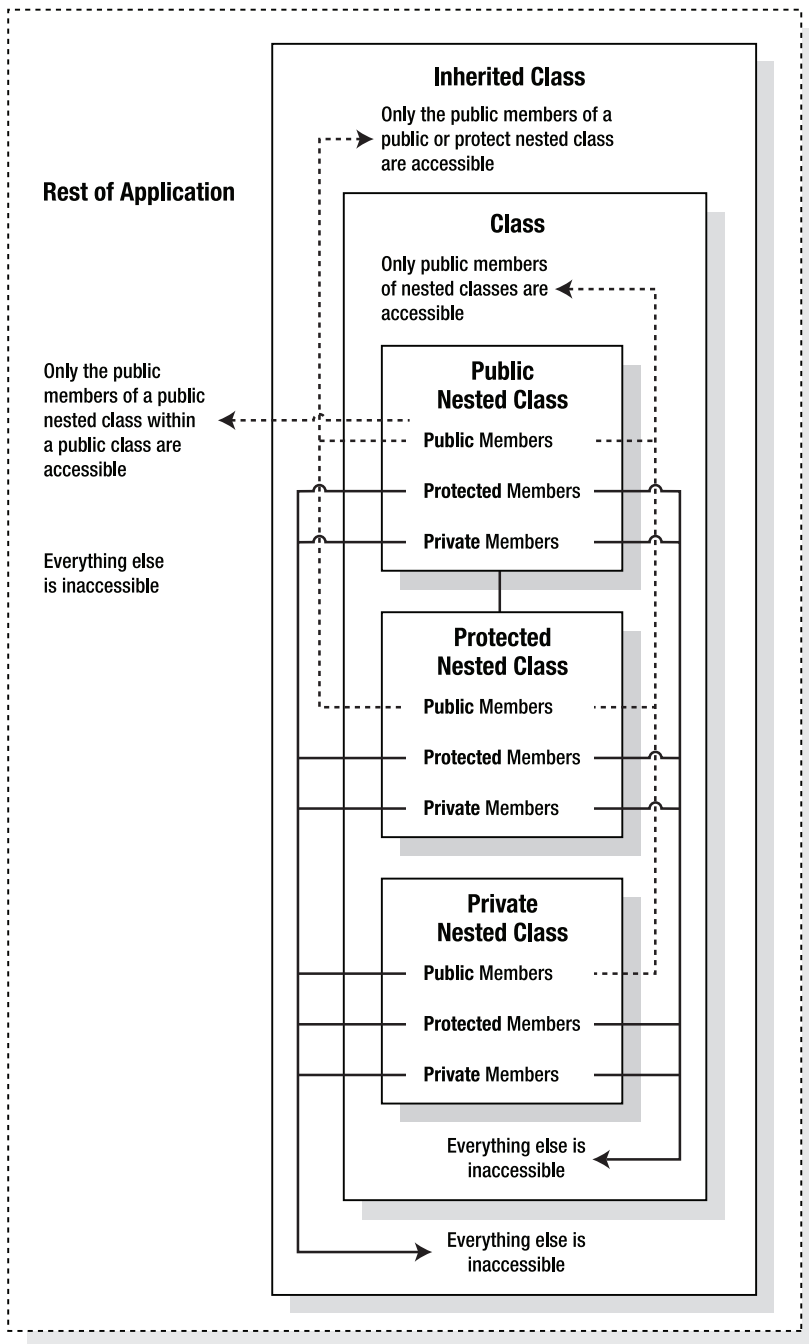


Figure 3-15. Accessing nested class members

Only public members are accessible outside of a nested class. For the surrounding class to access the public variable, the nested class can be public, protected, or private. For an inheriting class of the surrounding class, only public or protected access to the nested class will allow access to the nested class's public member variable. Finally, to access the nested class's public member variable outside of the inheritance tree of the surrounding class, both the nested class and the surrounding class must have public access.

Type Casting Between Classes

Type casting is the process of converting from one type to another. I covered type casting of the built-in types in Chapter 2. Now I expand on that discussion to include class and struct types.

C++/CLI provides three different operators for type casting between classes or structs: `static_cast`, `dynamic_cast`, and `safe_cast`. Each performs the process of trying to convert from one class type to another.

Notice that I wrote “trying to convert.” To legally convert a class to another, it needs to inherit from or be the class type to which it is being converted. For example, let's say class B inherits from class A, which in turn inherits from the `Object` class (all ref classes inherit from the `Object` class). This means that class B can safely be converted to a class A or the `Object` class. Class A, on the other hand, can safely convert to the `Object` class, but it would be an invalid conversion to class B, as class A is not inherited from class B.

The `static_cast` operator is the fastest of the three conversion operators, but it is also the most dangerous, as it assumes that the programmer knows what she is doing, and so it does no validity checks of its own. The syntax for the operator is simply this:

```
static_cast<target_type>(object_to_convert);
```

or

```
static_cast<int>(var);
static_cast<ClassA^>(ClassBvar);
```

Unsafe Code The `static_cast` operator cannot be verified and thus is classified as unsafe code.

The `dynamic_cast` operator is slower than the `static_cast` operator because it verifies that the type casting is valid. If the conversion is allowed, then the `dynamic_cast` operator completes the conversion. On the other hand, if it's not a valid conversion, then the `dynamic_cast` operator returns `nullptr`. The syntax of the `dynamic_cast` operator is identical to the `static_cast` operator except that `static` is replaced with `dynamic` in the following statement:

```
dynamic_cast<ClassA^>(ClassBvar);
```

A nifty little trick to check if a class is of a certain type can be done using the `dynamic_cast` operator. If you come from the C# world, then this is equivalent to the `is` operator:

```
if ( dynamic_cast<ClassA^>(ClassB) != 0)
{
    // ClassB is of type ClassA
}
```

The last conversion operator is the `safe_cast`. The `safe_cast` is the closest match the conversion behavior of all other .NET-supported languages and is designed so that you can rely exclusively on it for writing verifiable code. Because of this, C++/CLI uses `safe_cast` as the premier cast type for C-style casting. Thus, you normally won't even need to use the `safe_cast` operator at all, just the C-style

cast. The syntax of the `safe_cast` operator is also identical to the `static_cast` operator, except that `static` is replaced this time with `safe` in the following statement:

```
safe_cast<ClassA^>(ClassBvar);
```

The `safe_cast` operator is similar to the `static_cast` operator in that it can call user-defined conversions both implicitly and explicitly. It also can reverse standard conversion, like from a base class to an inherited class. The `safe_cast` is also like the `dynamic_cast` in that it checks to see if the cast was valid, except that instead of returning a `nullptr`, it throws an exception of type `System::InvalidCastException`. I cover exceptions in Chapter 4.

Listing 3-15 doesn't produce any output. I've provided comments on what the result of each statement is. If you want to prove to yourself that I'm right, you can run the code through a debugger and watch the results as you execute each statement.

Listing 3-15. *Type Casting in Action*

```
using namespace System;

ref class A {};
ref class B : public A {};
ref class C {};

void main()
{
    Object ^v1 = gcnew A();
    Object ^v2 = gcnew B();
    Object ^v3 = gcnew C();

    A ^a1 = gcnew A();
    A ^a2 = gcnew B();
    A ^a3 = dynamic_cast<A^>(v1); // downcast
    A ^a4 = dynamic_cast<A^>(v2); // downcast
    A ^a5 = static_cast<A^>(v3); // a5 has invalid value of type C class

    B ^b1 = gcnew B();
    B ^b2 = dynamic_cast<B^>(v2); // downcast
    B ^b3 = dynamic_cast<B^>(v3); // Fails b3 = null. Miss match classes
    B ^b4 = dynamic_cast<B^>(a2); // downcast

    C ^c1 = gcnew C();
    C ^c2 = dynamic_cast<C^>(v1); // Fails c2 = null. Miss match classes
    C ^c3 = static_cast<C^>(v2); // c3 has invalid value of type B class
    C ^c4 = safe_cast<C^>(v3); // downcast

    C ^c5 = (C^)(v3); // downcast

    // B ^e1 = safe_cast<B^>(c1); // does not compile as compiler knows these
    // are unrelated handles.
}
```

Abstract ref classes

An *abstract* ref class is an incomplete definition of a ref class, and it contains at least one pure virtual member method. It is a binding agreement between the ref class that derives from the abstract ref class and the ref class that calls the methods of that derived ref class.

In every other way, an abstract ref class is the same as a normal ref class. It can have variables, methods, properties, constructors, and destructors. The only thing it can't do is instantiate an object from itself. Though, it is possible to instantiate a derived ref class of the abstract ref class and then access the derived ref class using a handle to the abstract class.

```
AbstractClass ^ac = gcnew DerivedClass();
```

You might be wondering why you would need a constructor if you can't create an abstract class. The constructor of an abstract class serves the same purpose it does in a normal class: to initialize the member variables. There's one catch, though. The only place you can put an abstract class constructor is in the derived class's initializer list. Because the constructor only needs to be accessed by the deriving class, it's safest to declare the constructor as protected.

Any class that derives from an abstract class must implement the pure virtual function, or it will become an abstract class itself.

Any class that has pure virtual methods is abstract. In fact, even though C++/CLI has added the keyword `abstract` to declare a class as abstract, the keyword is optional and not needed. What it does is simply make the class notation explicit. It also makes your code more readable, as now you can see that a class is abstract from its initial declaration and you do not have to search the class for pure virtual methods.

However, if you do make the class abstract by including the keyword `abstract`, the class becomes abstract even if it normally would not be abstract. Thus, if a class is declared as abstract, you cannot create an instance of the class. Instead, only inherited classes from it can be instantiated. To make a class explicitly abstract, add the `abstract` keyword after the class declaration:

```
ref class AbstractExClass abstract
{
};
```

Because an abstract class has to be inherited, obviously a sealed class is not allowed, but it is legal to seal a virtual method, if the abstract class implements it.

To show abstract classes in action, Listing 3-16 shows an abstract class defined with a constructor and two methods, one of which is a pure virtual method. Another class inherits this class and seals `Method1`, but because it does not implement `Method2`, it too is abstract. Finally, this second abstract class is called by a third class, which implements the pure virtual function. Because the class now has all classes implemented, it can be instantiated. The example also shows how to pass an abstract class handle as a parameter.

Listing 3-16. Abstract Classes in Action

```
using namespace System;

ref class AbstractExClass abstract
{
protected:
    int AbstractVar;
    AbstractExClass(int val): AbstractVar(val) {}
```

```

public:
    virtual void Method1() = 0; // unimplemented method
    virtual void Method2() = 0; // unimplemented method
    void Method3()
    {
        Console::WriteLine(AbstractVar.ToString());
    }
};

ref class MidAbstractExClass abstract : public AbstractExClass
{
public:
    virtual void Method1() override sealed
    {
        Console::WriteLine((AbstractVar * 3).ToString());
    }
protected:
    MidAbstractExClass(int val) : AbstractExClass(val) {}
};

ref class DerivedExClass : public MidAbstractExClass
{
public:
    DerivedExClass(int val) : MidAbstractExClass(val) {}
    virtual void Method2() override
    {
        Console::WriteLine((AbstractVar * 2).ToString());
    }
};

void testMethod(AbstractExClass ^aec)
{
    aec->Method1();
    aec->Method2();
    aec->Method3();
}

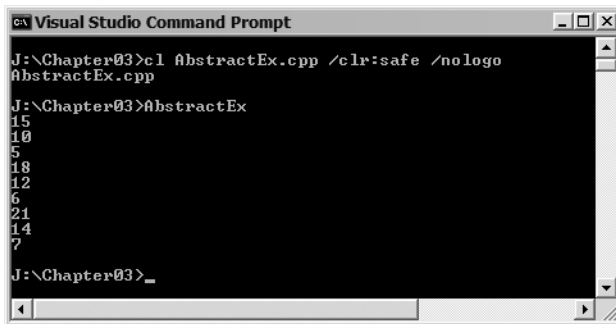
void main()
{
    AbstractExClass ^Ab1 = gcnew DerivedExClass(5);
    Ab1->Method1();
    Ab1->Method2();
    Ab1->Method3();

    AbstractExClass ^Ab2 = gcnew DerivedExClass(6);
    testMethod(Ab2);

    DerivedExClass ^dc = gcnew DerivedExClass(7);
    testMethod(dc);
}

```

Figure 3-16 shows the results of this little program.



```
Visual Studio Command Prompt
J:\Chapter03>cl AbstractEx.cpp /clr:safe /nologo
AbstractEx.cpp
J:\Chapter03>AbstractEx
15
10
5
18
12
6
21
14
7
J:\Chapter03>_
```

Figure 3-16. Results of *AbstractEx.exe*

Interfaces

An *interface* is similar to an abstract class in that it is a binding agreement between the class that derives from the abstract class and the class that calls the methods of that derived class. The key difference is that an interface only contains public, pure virtual methods. As the name suggests, it defines an interface to a class. But defining is all it does, as it does not contain variables or implementations for any methods.

Though classes can only inherit one class, they are able to inherit as many interfaces as needed to define the interface to the class. It is up to the class to implement all interfaces.

Like an abstract class, you can't instantiate an object from an interface. But, like abstract ref classes, it is possible to instantiate a ref class that implements the interface and then access the implementing ref class using a handle to the interface.

```
AnInterface ^iface = gcnew AnInterfaceImplementer();
```

This allows a developer to write a “generic” class that operates only on the interface. When a developer implements the interface, the base class can use the derived object in place of the interface.

Traditionally, C++ programmers have defined an interface as a ref class that contains only pure virtual methods. With C++/CLI, it has been formalized with the keywords `interface` and `class`. To create an interface, preface the keyword `class` with the keyword `interface` (instead of `ref`) in the definition and then place in the body of the interface a set of public, pure virtual methods.

Note You can also just place within the interface class method prototypes without the keyword `virtual` or the “= 0” suffix, because they are assumed.

Because only public access is allowed within an interface, the default logically for interface access is public. This means there is no need to include the public access modifier, as you would if it were a class. Oh, by the way, if you try to use access modifiers in your interface, the compiler slaps your hand and tells you to remove them.

Obviously, because an interface is only made up of pure virtual methods, the sealed keyword has no relevance to interfaces and will generate an error.

One additional note about interfaces: Even though they cannot contain method variables, it is perfectly legal to define properties within an interface. The definition of the properties cannot have an implementation—like other methods in the interface, the properties need to be implemented in the interface's inheriting class.

Listing 3-17 shows how to create a couple of interfaces, one with pure virtual methods only and another with a combination of methods and property definitions. It then shows how to do multiple inheritances in a ref class (one base class and two interfaces).

Listing 3-17. *Interfaces in Action*

```
using namespace System;

interface class Interface1
{
    void Method1();
    void Method2();
};

interface class Interface2
{
    void Method3();
    property String^ X;
};

ref class Base
{
public:
    void MethodBase()
    {
        Console::WriteLine("MethodBase()");
    }
};

ref class DerivedClass : public Base, public Interface1, public Interface2
{
public:
    virtual property String^ X
    {
        String^ get()
        {
            return x;
        }

        void set(String^ value)
        {
            x = value;
        }
    }

    virtual void Method1()
    {
        Console::WriteLine("Method1()");
    }
}
```

```

    virtual void Method2()
    {
        Console::WriteLine("Method2()");
    }

    virtual void Method3()
    {
        Console::WriteLine("Method3()");
    }

    virtual void Print()
    {
        MethodBase();
        Method1();
        Method2();
        Method3();
    }

private:
    String^ x;
};

void main()
{
    DerivedClass dc;

    dc.X = "Start'n Up";
    Console::WriteLine(dc.X);

    dc.Print();
}

```

Figure 3-17 shows the results of this little program. One thing you should note about the code is that the class that implements the interface requires each interface method to be prefixed with the keyword `virtual`. If you forget, you'll get a sequence of pretty self-explanatory compile time errors telling you to add the keyword.

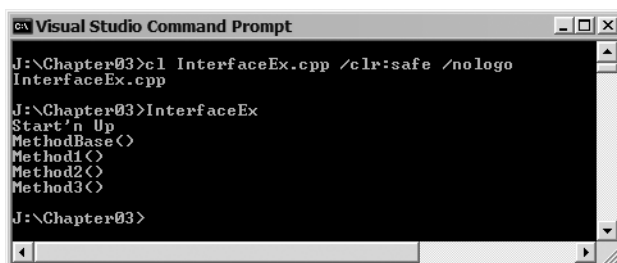


Figure 3-17. Results of *InterfaceEx.exe*

Summary

This chapter covered the basics of objected-oriented development using C++/CLI. You started with a quick refresher on what objects are and their fundamental concepts. From there, you saw how these concepts fit into the world of C++/CLI. You looked at `ref` classes in general, and then you broke down a `ref` class into its parts: member variables, member methods, and member properties. You finished the chapter by looking at abstract `ref` classes and interfaces.

Unlike the basics, C++/CLI has implemented many changes to traditional C++. Though none of the changes are complex—in fact, many simplify things—this chapter should be read carefully by experienced C++ programmers.

You will continue to examine C++/CLI in the next chapter, but now that you have covered the basics, you can move onto a few more complex and, dare I say, fun topics.