Essential Guide to Managed Extensions for C++

SIVA CHALLA AND ARTUR LAKSBERG

Apress™

Essential Guide to Managed Extensions for C++

ISBN (pbk): 1-893115-28-3

Printed and bound in the United States of America 12345678910

Editorial Directors: Dan Appleman, Peter Blackburn, Gary Cornell,
Jason Gilmore, Karen Watterson

Managing Editor: Grace Wong

Project Manager: Tracy Brown

Copy Editors: Ami Knox, Nicole LeClerc

Production Editor: Kari Brooks

Compositor: Impressions Book and Journal Services, Inc.

Indexer: Ann Rogers

Cover Designer: Tom Debolski

Marketing Manager: Stephanie Rodriguez

Distributed to the book trade in the United States by Springer-Verlag New York, Inc.,175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States, fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 901 Grayson Street, Suite 204, Berkeley, CA 94710. Phone 510-549-5938, fax: 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

CHAPTER 9

Properties

You have heard about data encapsulation. Roughly speaking, what it means is that all data members should be hidden behind public interfaces. Quite often you might find yourself writing code like this:

```
class Student
{
    int age_;
    // . . .
public:
    // . . .
    int GetAge() { return age_; }
    void SetAge(int age) { age_=age; }
};
```

And then you might use class `Student` in the following manner:

```
int CompareByAge(Student* student1, Student* student2 )
{
    return student1->GetAge() - student2->GetAge();
}
```

What's wrong with this code? The data encapsulation principle is obeyed—but at the price of code readability.

And this is exactly when *properties* are useful. A convenient syntactic notation long enjoyed by Visual Basic programmers, properties have actually been around in Visual C++ for quite some time supported by `__declspec(property)`. However, the `__declspec(property)` syntax wasn't very popular with C++ programmers.

Fortunately, properties have made their way into the .NET family of languages, including MC++. This chapter shows how to use scalar and indexed properties, explains what you can and cannot do with properties, and demonstrates how properties work.

Managed Properties: First Steps

Properties supported in MC++ are called *managed properties*. A property is declared using the `__property` keyword followed by a property method declaration. Note that the name of property methods must start with the prefix `get_` or `set_`. The method with the `get_` prefix is called a *getter* and the method with the `set_` prefix is called a *setter*. The name of the getter must be the same as the name of the setter except for the prefix. Here is how you would rewrite the preceding example using managed properties:

```
#using<mscorlib.dll>
 __gc class Student
{
    int age_;
    // . . .
public:
    // . . .
    __property int get_Age() { return age_; }
    __property void set_Age(int age) { age_=age; }
};
```

Using properties is now just as simple as this:

```
int CompareByAge(Student* student1, Student* student2)
{
    return student1->Age - student2->Age;
}
```

As you can see, a property can be used like a data member. In this example, the compiler replaces `Age` with a call to the appropriate method, which in this context is `get_Age`.

Note the keyword `__gc` in front of the class declaration. Managed properties can only be declared in managed classes—that is, either gc classes or value types.

Scalar and Indexed Properties

The common language runtime supports two types of properties: *scalar* and *indexed*. The MC++ compiler also supports both of them.

A scalar property is defined by a getter that does not take any parameters and a setter that takes exactly one parameter. A property is indexed if the getter takes one or more parameters and the setter takes more than one parameter.

To illustrate managed properties, let's consider the following scenario. Imagine that you need to design a database of students. Given a student's name or ID, this simple database should be able to return the student's address. Here is how you can do it:

```
#using <mscorlib.dll>
using namespace System;
__gc class Student
{
    String* name_;
    String* address_;
    /* . . . */
public:
    __property String* get_Address() { return address_; }
    __property void set_Address( String* address )
    { address_ = address; }
    /* . . . */
};
__gc class Database
{
    Student* students_[];
    int MapNameToId( String* name );
public:
    __property String* get_Address( int id )
    {
        return students_[id]->Address;
    }
    __property String* get_Address( String* name )
    {
        int id = MapNameToId( name );
        return students_[id]->Address;
    }
protected:
    __property void set_Address( int id, String *address )
    {
        students_[id]->Address = address;
    }
    __property void set_Address( String* name, String *address )
    {
        int id = MapNameToId( name );
        students_[id]->Address = address;
    }
};
Database* OpenDatabase();
```

Access to a student's address(es) is now achieved as follows:

```
int main()
{
    Database* pDatabase = OpenDatabase();

    String* address1 = pDatabase->Address["John"];
    String* address2 = pDatabase->Address[89640];
}
```

In the preceding sample, method `Student::get_Address` takes no arguments and returns a `String*`. This is an example of a scalar property. In contrast, `Database::get_Address` takes an argument, an index. That's why this kind of property is called indexed.

Implementing Property Access Methods

As you may have noticed, there are actually two `get_Address` methods defined in the class `Database`. One of them takes a student ID (`int`) as a parameter, whereas the other one takes `String*`, which demonstrates that property methods can be overloaded.

Overall, getters and setters are just regular functions. Not only can they be overloaded, they can also be declared virtual, pure virtual, or static. Property methods don't have to have the same access level. As in the example in the preceding section, `Database::get_Address` is public but `Database::set_Address` is protected. Furthermore, a property does not have to have both a getter and a setter; having either one is enough to define a property.

If property methods are declared as `static`, no object is needed to access the property. This is similar to how you would access a static data member, as demonstrated here:

```
#using<mscorlib.dll>
using namespace System;
void main()
{
    // Print current directory
    Console::WriteLine( Environment::CurrentDirectory );
}
```

This example uses the BCL class `Environment`, with a property called `CurrentDirectory`. Note that because the property is static, no instance of the class `Environment` is required. The same result could be achieved by calling the getter `get_CurrentDirectory`, which is a static member function:

```
void PrintDir()
{
    // Print current directory
    Console::WriteLine( Environment::get_CurrentDirectory() );
}
```

If you are concerned about performance overhead associated with a function call when using a property—don't worry. Property methods can be inlined, the same way as regular C++ member functions.

Parameters of Property Access Methods

Let's go back to our class `Student` example presented earlier and look closer at the getter and the setter. As you can see, `get_Age` takes no arguments and returns an `int` (one could argue that `unsigned char`, for instance, would be more appropriate, but let's ignore that for now).

The setter, `set_Age`, is a method that takes `int` as an argument. Note that the type of the setter parameter is the same as the return type of the getter—`int`. The MC++ compiler requires these two types to be identical. That understood, we could now say that the property `Age` has type `int`.

When it comes to indexed properties, keep in mind that to assign a value to the property you must pass it in the *last* argument of the setter. Let's look at the class `Database` in the example in the section "Scalar and Indexed Properties" earlier. What if you mistakenly defined the `set_Address` methods with the wrong order of parameters? For example:

```
#using <mscorlib.dll>
using namespace System;
 __gc class Database
{
    // . . .
protected:
    __property void set_Address( String *address, int id )
    {
        students_[id]->Address = address;
    }
    __property void set_Address( String *address, String* name )
    {
        int id = MapNameToId( name );
        students_[id]->Address = address;
    }
};
```

In case of the first `set_Address`, you will get an error because the type of the setter's last argument (`int`) doesn't match the getter's return type (`String*`).

The second case is far more dangerous: the code will compile but produce wrong results. We will get back to this issue in the next section. For now, just remember that the last argument of the setter is used for passing a value to the property.

A well-designed property behaves exactly as if it were a public data member. Consider an example of using our class `Student`, defined earlier in this chapter:

```
void ResetAges(Student* student1, Student* student2)
{
    return student1->Age = student2->Age = 0;
}
```

This would certainly work if `Age` were a public data member of class `Student`. However, this code gives an error:

```
'Student::set_Age' : cannot convert parameter 1 from 'void' to 'int'
```

The problem lies in the return type of `Student::set_Age`, which is `void`. When properties are expanded into function calls, the compiler comes up with this:

```
void ResetAges(Student* student1, Student* student2)
{
    return student1->set_Age( student2->set_Age(0) );
}
```

This code doesn't work because `student2->set_Age(0)` returns `void` and there is no conversion from `void` to `int`.

The solution? Define your setter as a method returning the property type or a reference to the property type:

```
__property int set_Age(int age) { age_= age; return age_; }
```

Bear in mind, however, that this approach has certain performance implications—returning a variable doesn't come free. In some cases, the compiler can "optimize away" such overhead, but this is not always possible.

How Properties Work

When you define a property by declaring a getter or a setter, the MC++ compiler "pretends" a data member is defined. This data member is called a *pseudo* member because it doesn't actually exist. The compiler replaces the pseudo member in your code by a call to the appropriate method which, depending on the context, is either the getter or the setter:

```
#using <mscorlib.dll>
__gc class MyArray
{
    // . . .
public:
    // . . .
    __property int get_Length();
    __property void set_Length(int);
};

void IncrementLength( MyArray* pArray )
{
    int nLen = pArray->Length; // calls pArray->get_Length();
    pArray->Length = nLen + 1; // calls pArray->set_Length(nLen + 1);
}
```

Remember, we warned you in the last section that the value of the property must be passed in the last argument of the setter. Now it's time to shed some light on this situation. Here is how the compiler generates the function call for the setter: given an indexed property `Address`, the compiler will convert the expression

```
pDB->Address[S"John Smith"] = S"Seattle";
```

into

```
pDB->set_Address(S"John Smith", S"Seattle");
```

passing the value from the right side of the assignment in a last argument of the setter. That's why if this last argument expects anything else other than the new value of the property, you will not get the result you want.

What if you want to increment a property using operator++? You can do that as follows:

```
void IncrementLength2( MyArray* pArray )
{
    ++pArray->Length;
    // compiler generates:
    // pArray->set_Length( pArray->get_Length() + 1);
    pArray->Length++;
    // compiler generates:
    // int tmp; tmp = pArray->get_Length(),
    //      pArray->set_Length( tmp + 1), tmp;
}
```

As in unmanaged C++, the post-increment operator is less efficient than the pre-increment operator—it requires a temporary variable to hold the value of the property before the increment. Hence this advice: where you can, consider using a pre-increment operator instead of a post-increment operator.

What You Cannot Do with Properties

As we said earlier, a good property behaves like a data member. Is there anything you can do with a data member but not with a property? Unfortunately, yes. Let's recall the earlier example with class Student. Consider the following:

```
#using <mscorlib.dll>
__gc class Student
{
    int age_;
    // . . .
public:
    // . . .
    __property int get_Age() { return age_; }
    __property void set_Age(int age) { age_=age; }
    void Birthday();
};
void IncrementAge( int* pAge )
{
    (*pAge)++;
}
void Student::Birthday()
```

```
{
    IncrementAge( &Age ); // error!
}
```

With what you now know about properties, it should come as no surprise that this code won't work. Still wondering why? Look at the call to `IncrementAge`—the function expects a parameter of type `int*`, but the property, `Age`, is provided instead. What can the compiler do? The pseudo member `Age` can be replaced with either `get_Age` or `set_Age`, neither of which would yield the desired result. That's why *taking address of a property is illegal*—and results in a compile-time error.

There is also one restriction to overloading property methods. Examine the following code:

```
#using <mscorlib.dll>
using namespace System;
__gc class Product
{
    // . . .
};
__gc class Inventory
{
    // . . .
public:
    // . . .
    __property int get_ItemsSold(int ProductID);
    __property void set_ItemsSold(int ProductID, int value);
    __property int get_ItemsSold() __gc[];*/ // error!
};
void SellProduct( Inventory* pInventory, int ProductID )
{
    pInventory->ItemsSold[ProductID]++;
}
```

As you can see, the method `get_ItemsSold` is overloaded: the first function takes one argument (`int`), and the other one takes no arguments but returns a managed array. Now we have an ambiguity problem in the function `SellProduct`: how do you convert the property `ItemsSold` into a getter/setter function call? It is impossible to determine from the context whether either of the methods `int get_ItemsSold(int)` or `int get_ItemsSold()__gc []` should be called. So, what we want you to take away from this is *an array property declaration shall not overload an indexed property.*

Summary

As this chapter has shown, managed properties are easy to use because the syntax for defining them is much simpler than that of regular C++ properties. Being syntactic sugar for function calls, properties behave like data members, making code that uses them cleaner and easier to understand.

In the next chapter, we will dive into another advanced topic of MC++—operators. Like properties, operators provide a more natural way of coding by hiding function calls "under the hood." You will see how to define and use operators and user-defined conversions for value types as well as gc classes.