Implementing a Property in C++

Posted by **Emad Barsoum** on **April 1st, 2003**
Environment: Visual C++ 7.0, Windows XP sp1, Windows 2000 sp3

Abstract

This project tries to simulate the property behavior that exists in C# (and other languages...) in C++, without using any extensions. Most libraries or compilers that implement properties in C++ use extensions, as in Managed C++ or C++ Builder, or they use the set and get methods that look like normal methods, not properties.

Details

Let's first see what a property is. A property acts like a field or member variable, in that is uses the library, but it accesses the underlying variable through the read or write methods or the set or get methods.

For example, if I have class A and property Count, I can write the following code:

```
1.   A foo;
2.
3.   cout  << foo.Count;
```

The Count actualy calls the get function that returns the value of the required variable. One of the main advantages of using a property instead of directly using the variable is that you can set the property as read-only (you are allowed only to read the variable without changing it), write-only, or both read and write. So, let's begin implementing it:

We need to be able to do the following:

```
1.     int i = foo.Count;    //-- Will call the get function to
2.                           //   get the underlying value --
3.
4.     foo.Count = i;        //-- Will call the set function to set
5.                           //   the value --
```

So, it obvious that we need to overload the '=' sign to set the variable, and also the return type (which we will now have later).

We will implement a class called property, which will act exactly like a property, as in the following:

```
template<typename Container, typename ValueType, int nPropType>

class property {}
```

This Template class will represent our property. The Container is the type of class that will contain the variable, the set & get methods, and the property. ValueType is the type of the internal variable itself, and nPropType is the access type of the property read-only, write-only, or read/write.

Now we need to set a pointer to the set and get methods from the container class to the property and also override the '=' operator so that the property will act like the variable. So, let's see the full listing of the property class:

```
#define READ_ONLY 1
#define WRITE_ONLY 2
#define READ_WRITE 3

template <typename Container, typename ValueType, int nPropType>
class property
{
public:
property()
{
  m_cObject = NULL;
  Set = NULL;
  Get = NULL;
}
//-- This to set a pointer to the class that contain the
//   property --
void setContainer(Container* cObject)
{
  m_cObject = cObject;
}
//-- Set the set member function that will change the value --
void setter(void (Container::*pSet)(ValueType value))
{
  if((nPropType == WRITE_ONLY) || (nPropType == READ_WRITE))
    Set = pSet;
  else
    Set = NULL;
}
//-- Set the get member function that will retrieve the value --
void getter(ValueType (Container::*pGet)())
{
  if((nPropType == READ_ONLY) || (nPropType == READ_WRITE))
    Get = pGet;
  else
    Get = NULL;
}
```

```
37. //-- Overload the '=' sign to set the value using the set
38. //   member --
39. ValueType operator =(const ValueType& value)
40. {
41.   assert(m_cObject != NULL);
42.   assert(Set != NULL);
43.   (m_cObject->*Set)(value);
44.   return value;
45. }
46. //-- To make possible to cast the property class to the
47. //   internal type --
48. operator ValueType()
49. {
50.   assert(m_cObject != NULL);
51.   assert(Get != NULL);
52.   return (m_cObject->*Get)();
53. }
54. private:
55.   Container* m_cObject;  //-- Pointer to the module that
56.                          //   contains the property --
57.   void (Container::*Set)(ValueType value);
58.                          //-- Pointer to set member function --
59.   ValueType (Container::*Get)();
60.                          //-- Pointer to get member function --
61. };
```

So, now let's see it piece by piece:

In the following code, just set the Container pointer to a valid instance that will have the property:

```
1. void setContainer(Container* cObject)
2. {
3.   m_cObject = cObject;
4. }
```

In the following code, set a pointer to the get and set member functions of the container class. The only restriction is that the set must have a single parameter and return void and the get no paramter, but return the value type:

```
1. //-- Set the set member function that will change the value --
2. void setter(void (Container::*pSet)(ValueType value))
3. {
4.   if((nPropType == WRITE_ONLY) || (nPropType == READ_WRITE))
5.     Set = pSet;
6.   else
7.     Set = NULL;
8. }
9. //-- Set the get member function that will retrieve the value --
10. void getter(ValueType (Container::*pGet)())
11. {
12.   if((nPropType == READ_ONLY) || (nPropType == READ_WRITE))
13.     Get = pGet;
```

```
14.   else
15.     Get = NULL;
16. }
```

In the following code, the first operator is the '='; it calls the set member of the container and gives it the value. The second part is to make the entire property class act as the ValueType so it returns the return of the get function:

```
 1. //-- Overload the '=' sign to set the value using the set member --
 2. ValueType operator =(const ValueType& value)
 3. {
 4.   assert(m_cObject != NULL);
 5.   assert(Set != NULL);
 6.   (m_cObject->*Set)(value);
 7.   return value;
 8. }
 9. //-- To make possible to cast the property class to the
10. //   internal type --
11. operator ValueType()
12. {
13.   assert(m_cObject != NULL);
14.   assert(Get != NULL);
15.   return (m_cObject->*Get)();
16. }
```

So, let's see how to use it now:

As shown below, the PropTest class implements a simple property called Count. The actual value will be stored and retrieved from the private member variable "m_nCount", through the get and set methods. The get and set methods can have any name as long as their address is passed to the property class as shown in the constructor of the PropTest, in the line "property<PropTest,int,READ_WRITE> Count; " that we have a read & write property of type integer in class PropTest called Count. Now you can call Count as a normal member variable, but in fact you call the set and get methods indirectly.

The initialization shown in the constructor of the PropTest class is necessary for the property class to work.

```
 1. class PropTest
 2. {
 3. public:
 4.   PropTest()
 5.   {
 6.     Count.setContainer(this);
 7.     Count.setter(&PropTest::setCount);
 8.     Count.getter(&PropTest::getCount);
 9.   }
10.   int getCount()
11.   {
12.     return m_nCount;
```

```
13.   }
14.   void setCount(int nCount)
15.   {
16.     m_nCount = nCount;
17.   }
18.   property<PropTest,int,READ_WRITE> Count;
19.
20.
21. private:
22.   int m_nCount;
23. };
```

As shown below, you use the Count property as if it were a normal variable.

```
1. int i = 5,j;
2. PropTest test;
3. test.Count = i;    //-- call the set method --
4. j= test.Count;     //-- call the get method --
```

For read-only, you create an instance for the property as follows:

```
1. property<PropTest,int,READ_ONLY > Count;
```

For write-only, you create an instance for the property as follows:

```
1. property<PropTest,int,WRITE_ONLY > Count;
```

> **Note:** If you set the property to read-only and you try to write to it, it will cause an assertion. The same will occur if you set the property to write-only and you try to read it.

Conclusion

This article showed you how to implement a property in C++, using standard C++ without any extension. Of course, directly using the set and get methods is more efficient because with this method you have a new instance of the property class per each property.

Downloads

Download demo project - 100 Kb