C++/CLI V: Propiedades

Las propiedades en C++/CLI tienen un tratamiento similar, por no decir idéntico, con su equivalente en C#, con la diferencia de que son algo más potentes, no porque lo hayan hecho así, sino por las características notacionales del C++.

El concepto de propiedad no nace con el .NET, sino que existe desde mucho antes. Ya Borland en su Delphi implementa la idea, luego extendida a C++Builder: la *__property*.

El concepto nace con la idea de tener una forma de acceder a variables internas de una clase sin romper el concepto de encapsulación, así como la de permitir realizar efectos laterales en dichos accesos.

Una propiedad aparentemente representa una variable miembro pública dentro de una clase, pero realmente oculta en su interior dos llamadas a métodos, una para la lectura y otra para la escritura. Por ejemplo, el código

```
int a=unObjeto.Valor;
```

podría parecer un acceso a una variable pública, pero no lo es cuando hablamos de propiedades. Y el siguiente código tampoco:

```
unObjeto.Valor=33;
```

Aunque a simple vista parezca una lectura y una escritura sobre una variable miembro pública, realmente se está ejecutando el siguiente código, que el compilador sustituye por nosotros:

```
int a=unObjeto.get_Valor();
unObjeto.set_Valor(33);
```

Pero vayamos por partes.

Propiedades triviales

Nunca el nombre estuvo mejor puesto, y aunque C# 3.0 lo traerá, en C++/CLI existe desde el origen del lenguaje. Una propiedad trivial se declara como:

```
ref class UnaClase
{
      public:
            property int Valor;
};
```

Y se escribe y se lee sobre ella tal y como hemos indicado más arriba. Pero el compilador nos está engañando (o más bien está haciendo tareas laboriosas en nuestro lugar, lo que siempre es de agradecer). Lo que realmente está creando es

```
ref class UnaClase
{
      int Valor;
public:
      int get_Valor(void){return Valor;}
```

```
        void set_Valor(int v){Valor=v;}
};
```

Aunque en nuestro código estemos trabajando como si dichos métodos no existieran, realmente están en el código compilado, y se llaman de esa misma forma. Es decir, que podríamos adelantarnos al compilador y crear dichos métodos nosotros mismos a mano, y entonces podrían ocurrir dos cosas:

- Si ponemos "property Valor;" en la zona pública de la clase, y quitamos la variable de la zona privada, el compilador entenderá que eso es una propiedad y podremos utilizar semántica de variable (es decir, como los ejemplos de código de más arriba).
- Si lo dejamos tal y como está, el compilador no entenderá que estamos definiendo una propiedad y tendremos que utilizar los métodos *get_Valor()* y *set_Valor()* como cualesquiera otros métodos normales.

Esto tiene una contrapartida, y es que si nuestra intención es no crear una propiedad con dicho nombre, podemos montar un lío de narices, y justo al revés, no querer una propiedad y montar una, por lo que lo recomendado es no implementar ningún método que empiece por *get_* ni por *set_*, pero a veces estos ya existen en bibliotecas antiguas, y no podemos evitar su uso.

Los chicos de Microsoft recomiendan el uso de este tipo de propiedades cuando no tengamos claro si definir una variable o un método, sobre todo en bibliotecas, ya que así podemos pasar a implementar "accesores" (por decirlo de alguna manera) de forma transparente sin tener que modificar el código existente (en general, sin tener que recompilarlo; tan sólo sería necesario cambiar el ensamblado viejo por el nuevo sin hacer nada más, y otros ensamblados que accedieran a él no notarían la diferencia, mientras que si seguimos la forma clásica, también sería necesario recompilar el código que accediera a dicho elemento y no solo el del propio elemento).

La explicación no tiene misterio. Si definimos una variable pública, el código que acceda a ella simplemente cargará la dirección de memoria en donde ésta se encuentre (aunque se trate de una doble indirección como en el caso de las referencias), mientras que si definimos una propiedad, el código hará una llamada a *get_<nombre>()* o *set_<nombre>()*, y si en la biblioteca hemos cambiado la variable por una propiedad, en el primer caso la aplicación fallará, en el segundo continuará ejecutándose sin problemas, y nosotros habremos podido cambiar el comportamiento de la biblioteca al vuelo y sin que nadie se entere.

Propiedades escalares

Estas son las propiedades 100% compatibles con .NET, y de forma equivalente a C#, se declaran de la misma forma:

```
ref class UnaClase
{
public:
      property int Valor
      {
            int get(void){return Valor;}
```

```
            void set(int v){Valor=v;}
        }
    };
```

El funcionamiento es idéntico a la forma anterior, con la diferencia de que ahora estamos declarando los dos métodos que leen y escriben la propiedad, exactamente como en C#. Y como en C# podemos tener propiedades de sólo lectura, de sólo escritura y de lectura/escritura dependiendo de qué métodos implementemos.

Una diferencia importante con C# es que el nombre de los métodos puede ser cualquiera; aquí nosotros los hemos llamado *get* y *set*, pero podríamos haberlos llamado *juan* y *pepe*. Lo que sí debemos tener en cuenta es la firma del método. El de lectura no ha de recibir ningún parámetro y devolver el mismo tipo que la propiedad y el de escritura justo al revés. Cualquier otra firma hará que el compilador proteste.

El C++/CLI tiene otra limitación (que creo que también tiene el C#, pero menos visible ya que parte de lo explicado queda completamente oculto por el compilador). Ya lo hemos dicho con anterioridad, si una propiedad se llama *Valor*, no podemos declarar métodos con los nombres *set_Valor()* ni *get_Valor()* ya que el compilador los está generando internamente para implementar las propiedades.

Lo que sí podemos hacer es declarar en un lado una propiedad e implentarla en otro. Considerando el código anterior, podríamos definir en el fichero cabecera:

```
ref class UnaClase
{
public:
      property int Valor
      {
      int get(void);
      void set(int v);
      }
};
```

y luego, en el fichero fuente (.cpp), implementar:

```
int UnaClase::Valor::get(void)
{
      return Valor;
}
void UnaClase::Valor::set(int v)
{
      Valor=v;
}
```

Con esta característica damos un paso más allá, permitiendo ver la declaración de la propiedad pero ocultando el código que la compone en, por ejemplo, una biblioteca. Aunque realmente esta característica tiene poco valor, ya que mediante reflexión y desensamblado es trivial conocer el código fuente, pero ahí está para quien quiera usarla.

Acceso, antecesores y otras variantes

Comentar brevemente, antes de entrar en las propiedades indexadas, que las propiedades pueden ser virtuales, se pueden heredar, ser estáticas, se pueden sobrescribir, pueden tener diferente tipo de acceso (público, privado, protegido…), e incluso la parte de escritura puede tener diferente tipo de acceso a la de lectura. Realmente no estamos limitados en nada respecto a los métodos normales, ya que de hecho las propiedades no son otra cosa que unos meros alias y forma sincopada de métodos, como hemos podido ver más arriba. Y para implementarlos sólo debemos poner las palabras reservadas adecuadas en los lugares adecuados.

De todos modos, el autor no es muy partidario de hacer "florituras" con las propiedades; por experiencia propia sabe que cuanto más complejo sea un código, más difícil es luego de depurar y extender. De todos modos podría tener su utilidad justificada. Imaginemos una biblioteca que expusiera una propiedad de sólo lectura para aquellos elementos que la usen directamente, y su escritura para aquellos que la heredaran para modificar en alguna manera su comportamiento. Deberíamos declarar entonces como público el acceso de lectura y como protegido el de escritura.

Finalmente, cuando surja la duda (que surgirá) sobre qué y cómo operar con las propiedades, simplemente tenemos que imaginarnos que son métodos sin el paréntesis final, y actuar en consecuencia.

Propiedades indexadas

Aquí volvemos a salirnos de las características de C#; las propiedades indexadas son aquellas que se pueden acceder como si fueran un array, es decir:

```
int a=unObjeto.Valor[índice];
```

o incluso:

```
int a=unObjeto[índice];
```

sin que *unObjeto* sea un array.

La potencia de todo esto no está en el propio sistema abreviado por corchetes, que es potente en sí, sino en que el *índice* puede ser **cualquier tipo de dato**; es decir, podemos expresar semánticas todo lo complejas que queramos, quizás sobrepasando el concepto de índice más allá incluso de la programación genérica, o más bien igualándolo sin toda la parafernalia que la acompaña.

Es decir, podríamos tener código como este:

```
Persona ^p=persona["RFOG"];

Casa ^c=casas->CasaDe[p];
```

En una metonimia para una consulta a una base de datos que obtenga en primer lugar la persona que viene representada por la cadena "RFOG" y en segundo la casa que pertenece a la

persona indicada. En el primer caso estamos utilizando lo que se conoce como "propiedad indexada por defecto" y en el segundo "propiedad indexada" a secas.

Evidentemente volvemos a encontrarnos con el tema de la complejidad dentro del código fuente, ya que dada la potencia del concepto, podemos llegar a realizar asignaciones un tanto absurdas, así que aquí, igual que antes, el autor recomienda que se utilice esta característica con cierto rigor y escepticismo.

Pero veamos cómo declarar este tipo de propiedades:

```
ref class Calendario
{
      static property bool EsFiesta[DateTime d]
      {
            bool ComprobarSiEsFiesta(DateTime d){...}
            void AsignarDiaDeFiesta(DateTime d,bool bFiesta){...}
      }
}
```

Debemos comentar que para la firma del método del método de escritura, el parámetro indexado viene antes que el valor de retorno (que en este caso es de asignación). El que hayamos indicado que la propiedad es estática no es más que una muestra para el lector; en este caso, ambos métodos solo podrán acceder a datos miembros estáticos de la clase *Calendario*, ya que la propiedad a la que están asociados es estática, por lo que en ningún momento se rompen las reglas.

El uso es sencillo:

```
      Calendario::EsFiesta[DateTime::Now]=true;
      if(Calendario::EsFiesta[DateTime::Now]) {...}
```

Pero podríamos haber escrito la clase de otra forma mucho más sucinta:

```
ref class Fiestas
{
      static property bool default[DateTime d]
      {
            bool ComprobarSiEsFiesta(DateTime d){...}
            void AsignarDiaDeFiesta(DateTime d,bool bFiesta){...}
      }
}
```

Y entonces podríamos usarla así:

```
      Fiestas[DateTime::Now]=true;
      if(Fiestas[DateTime::Now]) {...}
```

Esta última forma es la forma de indicar propiedades indexadas por defecto; simplemente tenemos que sustituir el nombre de la propiedad por la palabra reservada *default*.

Pero demos otra vuelta de tuerca más, demostrando la potencia del concepto. Para ello, examinemos el siguiente código fuente (que ya fue expuesto en mi tutorial sobre C++/CLI y que se puede descargar desde aquí):

```
class tito {…};
class ManejadorDeTitos
{
      …
      property tito ^Titos[String ^,int]
      {
            tito ^get(String ^ind1, int ind2) {…}
            void set(String înd1, int ind2, tito ^valor) {…}
      }
};
…
tito ^t=manejaTitos->Titos[“El mejor tito”, 5];
```

Y sigo copiando: "En el ejemplo vemos que si la definición de la clase podría complicarse un tanto, su manejo no, con lo que salimos ganando. Otra cosa que podemos observar es que esta forma de acceso es un tanto *difusa*. En nuestro ejemplo, ¿qué ventaja tiene acceder a los *titos* (por favor, no se rían), de esta forma? Pues depende de lo que sea un *tito*, pero imaginemos que tuviéramos una tabla de resultados organizados por categoría y puntuación. La obtención de quien obtuvo tal resultado y en tal categoría sería inmediata y natural, otra cosa sería qué se ejecuta, y cómo, dentro del método *get* o del *set*, pero eso queda bajo el criterio del programador."

Resumiendo: las propiedades indexadas no se quedan en un sólo índice, sino que se pueden especificar los que queramos y del tipo que queramos.

Más indexadores

Como el autor está algo cansado de escribir, va a copiar parte de su tutorial ya citado en donde explica el atajo para operar con propiedades indexadas cuyo índice sea numérico:

Propiedades array. Son propiedades que actúan como elementos de un array, o lo que es lo mismo, sirven para devolver valores de un array como si del propio array se tratara. Son iguales a las escalares, pero su tipo es un array:

```
property array<tipo>^Nombre
{
      array<tipo>^get(void) {…}
      vod set(array<tipo>^valor){…}
};
```

Su uso es idéntico a las escalares, pero en este caso debemos indicar un índice para poder acceder a cada elemento. Pongamos un ejemplo:

```
ref class Clase
{
public:
Clase(int tam) {arr=gcnew array<String ^>(tam);
property array<String ^> ^Cadenas
{
      array <String ^>^get(void) { return arr;}
```

```
        void set(array<String ^>^valor) {arr=valor;}
    }
    private:
        String^arr;
    };
    …
    Clase ^miClase=gcnew Clase(5);
    miClase->Cadenas[0]=”cadena1”:
    miClase->Cadenas[1]=”cadena2”:
    Console::WriteLine(miClase->Cadenas[1]);
```

Esta implementación tan sencilla para manejar propiedades que acceden a los elementos de un array se ha diseñado ex-profeso de esta forma para simplificar y llevar un paso más allá el acceso a los citados.