C++/CLI (II): Sobre referencias y referencias de seguimiento (*tracking references*)

En C++, cuando queremos que una variable sea modificada dentro de un método tenemos dos opciones: pasar su dirección de memoria y operar dentro del método con el puntero, o utilizar referencias.

Aunque en un principio las referencias y los punteros pudieran parecer equivalentes, no lo son, ya que existen diferencias bastante notables. Pero vayamos por partes.

Veamos el siguiente código:

```
void CambiaA(int *b)
{
  (*b)++;
}

int main(void)
{
  int a=5;
  CambiaA(&a);
  printf("%i",a);
  return 0;
}
```

```
void CambiaA(int &b)
{
  b++;
}

int main(void)
{
  int a=5;
  CambiaA(a);
  printf("%i",a);
  return 0;
}
```

Observamos que el código es casi idéntico. El bloque de la derecha trabaja con referencias, el de la izquierda con punteros, y ambos realizan la misma acción: sacar por pantalla el número 6, que se corresponde de instanciar una variable que llamamos *a* (es decir, crear un hueco en la pila de suficiente tamaño para guardar un entero) y guardar en su interior el valor 5. Tras la llamada al método, el contenido de *a* se incrementa en uno, pero mediante diferentes vías (el hecho de que un compilador serio de C++ generara prácticamente el mismo código no nos interesa, lo que nos interesan son los conceptos subyacentes).

El código de la derecha, aparte de ser más claro, es más económico en memoria (aunque posiblemente no en tiempo de compilación). Cuando entramos en *CambiaA()*, el compilador no crea ninguna variable nueva, sino que anota dónde está *a* en la pila y, al ejecutar la instrucción b*++* **incrementará directamente la casilla en donde se ha guardado *a*, aunque ahora se llame *b*** (Otra vez queremos hacer notar que lo que realmente está ocurriendo es que el compilador cogerá de la pila el valor de *a*, lo meterá en un registro del micro, realizará el incremento y volverá a colocarlo en su lugar, pero repetimos que de momento sólo nos interesa el concepto).

En el código de la izquierda, la entrada en *CambiaA()* creará una nueva variable en la pila que es el puntero *b*, cuyo contenido será la dirección de memoria en donde esté guardado *a*. Y cuando incrementemos el valor de **b*, estamos realmente mirando dónde apunta *b* y cambiando dicho valor.

La diferencia es sutil pero muy importante. En este ejemplo puede no verse claro, pero si el lector utiliza sobrecarga de operadores podrá comprobar cuán potentes son las referencias a la hora de trabajar con estas construcciones, ya que mediante su uso permiten devolver una referencia a la misma variable, ahorrando así memoria y acelerando las cosas.

Un consejo práctico a la hora de elegir qué utilizar, si un puntero o una referencia, es hacernos la pregunta de si la llamada al método que va a modificar el parámetro trabaja con asignación de memoria dinámica o no; es decir, si lo que estamos pasando está instanciado en el montículo, lo suyo es pasar un puntero, y si está instanciado en la pila, usaremos referencias. Evidentemente no es una regla estricta, sino simplemente un mero juicio de valor ante la duda.

Un uso práctico para las referencias es la devolución de resultados. Supongamos que queremos realizar una operación matemática cuyo dominio en el resultado puede ser cualquier valor de la variable a considerar, por lo que no podemos determinar a priori si el resultado obtenido es ese porque la operación se llevó a cabo satisfactoriamente o porque el cálculo se detuvo ahí. Podríamos utilizar una firma de método como la siguiente:

```
bool OperacionMatematica(Elemento &e);
```

De modo que la devolución de *true* nos indica que lo que hay en *e* es válido, y en donde *Elemento* puede ser cualquier construcción matemática (Aquí entramos en otro tema que genera bastantes discusiones, y es la decisión de lanzar una excepción en caso de error o devolver un valor como acabamos de hacer, pero no vamos a discutir eso ahora).

Esta característica también está disponible en C++/CLI con el nombre de *referencia de seguimiento* (o *tracking reference* en inglés). El concepto es el mismo, aunque la nomenclatura y sintaxis sean diferentes.[1]

En C++/CLI una referencia se indica mediante el símbolo %, y su funcionamiento es básicamente el mismo. Realizando el código anterior en C++/CLI quedaría:

```
void CambiaA(int ^b)
{
  ^b++;
}

int main(array<String ^>^)
{
  int a=5;
  CambiaA(&a);
  Console::WriteLine(a.ToString());
  return 0;
}
```

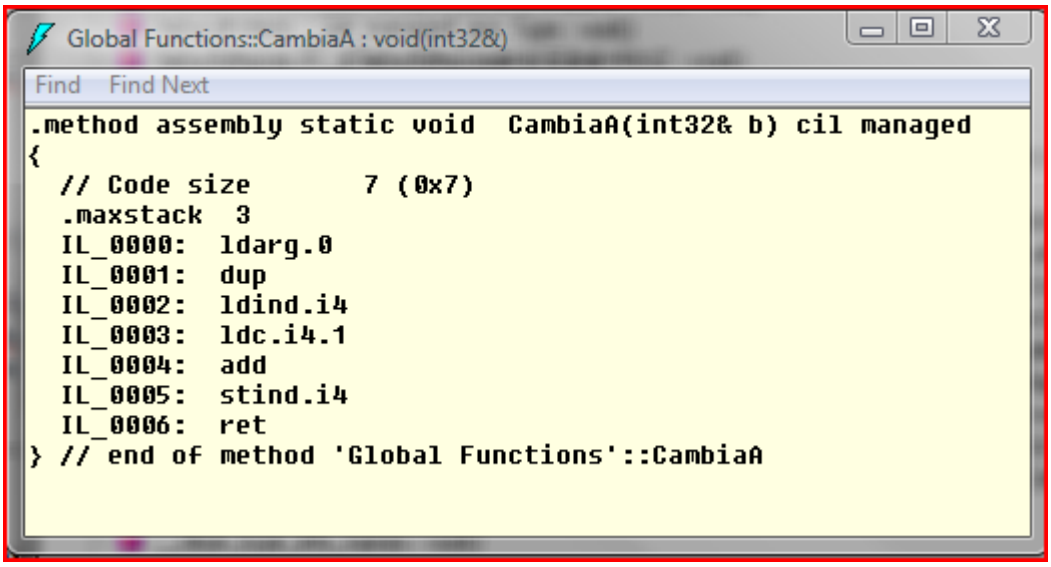
```
void CambiaA(int% b)
{
  b++;
}

int main(array<String ^>^)
{
  int a=5;
  CambiaA(a);
  Console::WriteLine(a.ToString());
  return 0;
}
```

Si observamos, los cambios son meramente cosméticos y estamos realizando las mismas acciones. Ahora disponemos en el código de la izquierda de referencias (en su equivalente al puntero nativo) y en el de la derecha de referencias de seguimiento (en su equivalente a la referencia nativa). Ya sé que la nomenclatura es un poco liosa al llamar *referencia* al manejador para las variables situadas en el montículo manejado y referencia de seguimiento al simple alias, pero es lo que hay.

[1] Esta característica está disponible en C# mediante los parámetros ***out*** y ***ref***, pero su potencia e integración en el lenguaje es mucho menor que en C++/CLI, que también tiene dichas construcciones, pero en su caso deben ir indicadas dentro de corchetes, ya que deben ser tratadas como atributos para respetar la idiosincrasia de C++.

Pese a lo que pudiera pensarse con el código de la izquierda, no hay ningún encajamiento ni desencajamiento, ya que al tener almacenada a en la pila, la llamada a *CambiaA()* simplemente modifica su valor. Vea si no el código MSIL generado para dicho método:



Pero vayamos un paso más en cuanto a C++/CLI. ¿Podemos pasar una referencia de seguimiento de una referencia? Pues claro que sí. Igual que en C++ podemos pasar la referencia de un puntero. Veámoslo.

```
using namespace System;

void CambiaSTR(String^ %str)
{
  str="Hola Nuevo";
}

int main(array<String ^>^)
{
  String ^cad="Hola Viejo";
  CambiaSTR(cad);
  Console::WriteLine(cad);
  return 0;
}
```

```
void CambiaA(int* &b)
{
  (*b)++;
}

int main(int argc, _TCHAR* [])
{
  int *a=new int();
  *a=5;
  CambiaA(a);
  printf("%i",*a);
  return 0;
}
```

Aquí el bloque de la derecha volverá a devolver 6 como respuesta y el de la izquierda “Hola Nuevo”, quedando la cadena “Hola Viejo” marcada para destrucción (“Hola viejo” es una *cadena estática* y está alojada dentro del ejecutable, en C++ simplemente no pasaría nada al perder la asignación de *cad*, pero en C++/CLI quizás la cadena se copie del segmento de datos al montículo manejado durante la carga del ensamblado. Realmente ignoro el comportamiento. En otro momento lo miraré).

Y cómo no, no es necesario llamar a un método o función para trabajar con las referencias:

```
using namespace System;

ref class Cosa
{
public:int a;
};
```

```
class Cosa
{
public:int a;
};

int _tmain(int argc, _TCHAR* argv[])
{
```

```
int main(array<System::String ^>
^args)
{
	Cosa ^c1=gcnew Cosa();
	c1->a=30;
	Cosa %c2=*c1;
	c2.a++;
	Cosa ^c3=%c2;
	c3->a++;
	Console::Write(c3-
>a.ToString());
	return 0;
}
```

```
	Cosa cosa;
	cosa.a=33;
	Cosa &rCosa=cosa;
	rCosa.a++;
	printf("%i",cosa.a);
	return 0;
}
```

Un tema curioso a observar en el bloque de C++/CLI (el de la izquierda) consiste en cómo desreferenciamos una referencia sobre una referencia de seguimiento. Me refiero a la línea

```
	Cosa %c2=*c1;
```

Si miramos con detalle, estamos haciendo exactamente lo mismo que con el código nativo cuando desreferenciamos un puntero a una referencia, lo que nos permitirá realizar plantillas que puedan trabajar indistintamente con objetos manejados o nativos.