C++/CLI (I): Punteros y Referencias

De montículos, pilas y asignaciones

De todos es conocida la existencia de dos almacenes de memoria para el uso de nuestros programas. Por un lado tenemos la pila y por el otro el montículo. En la pila se colocan las variables automáticas (sí, lector, no te extrañes de ese nombre: una variable automática es aquella que definimos y declaramos de forma directa, es decir *int a* declara una variable automática con el nombre de *a*). Y en el montículo las variables indirectas o de asignación dinámica.

Tradicionalmente Windows pone a disposición para cada programa una pila y tres montículos, a saber, el montículo local, el global y el virtual (no vamos a entrar en detalles, pero el lector inquieto podría echar un vistazo a las funciones del API de Win32 *VirtualAlloc()*, *GlobalAlloc()* y *LocalAlloc()* para comprobarlo).

Una aplicación .NET añade dos estructuras de memoria más, es decir, tanto una pila manejada como un montículo manejado. Ambos están situados y controlados dentro de la máquina virtual .NET, y generalmente son los que el programa va a utilizar siempre que no ejecute código nativo mediante Interop de atributos o de IJW.

Quizás al programador de C# le llame la atención la separación entre pila manejada y pila nativa, pero así es en la realidad. En C++/CLI, utilizando el Interop IJW, podemos colocar conjuntamente ambos tipos de variables sin ningún problema, pero cada una irá a su pila, es decir, en el fragmento de código

```
void fnc(void) {
char *pCad;
StringBuilder ^s;
…
```

la variable *pCad* irá almacenada en la pila nativa y *s* lo irá en la manejada. Y cuando hagamos

```
pCad=new char[20];
s=gcnew StringBuilder();
```

el hueco para contener a *pCad* irá al montículo nativo y el correspondiente hueco para *s* se reservará en el montículo manejado, y se asignarán sus direcciones respectivas al contenido de ambas variables.

Describir ambas pilas no tiene mucho sentido ya que ambas son casi equivalentes salvo que una la controla el motor .NET y la otra el sistema operativo, y su funcionamiento es idéntico: guardar los puntos de retorno de los métodos y las variables automáticas. La única ventaja de la nativa es que tiene un área especial para almacenar variables globales, que no está presente en su contrapartida manejada.

Los montículos sí que son enormemente diferentes. El nativo no es más que un saco de memoria física asignada al espacio de direcciones de la aplicación, mientras que el manejado es otra cosa mucho más delicada y potente: puede controlar desbordamientos, olvidos en la liberación de bloques, y se compacta para hacer sitio a nuevos bloques.

Punteros y referencias

pCad es un puntero, *s* es una referencia. ¿Cuál es la diferencia? El espacio reservado al que apunta el primero va en el montículo nativo, y el del segundo en el manejado. Dicho así, y habiendo leído el párrafo anterior casi lo tenemos todo.

Pero veamos algunos detalles. Como *pCad* contiene una dirección de memoria física, podemos cambiar su valor sin problemas. No me refiero a cambiar *a lo que apunta pCad*, sino a cambiar el contenido del propio *pCad*. Ni el compilador ni el entorno de ejecución nos dirán nada. En el segundo ejemplo hemos asignado en el montículo 20 bytes, por lo que es perfectamente válido hacer

```
*(pCad+10)='a';
```

Simplemente estamos cogiendo lo que hay en *pCad*, le sumamos 10 y escribimos en la dirección de memoria resultante la letra 'a'.

Eso, en .NET, no se puede hacer porque, siguiendo con el ejemplo, *s* no es un puntero, sino una referencia. No contiene una dirección de memoria física, sino un índice dentro de una tabla que es interna al motor de tiempo de ejecución del .NET y que describe los bloques de memoria asignados dentro del montículo manejado. Como poco hay una doble indirección, es decir, para acceder al contenido de *s* tenemos que, primero buscar su valor dentro de dicha tabla, y luego mirar a dónde realmente está apuntando. Pero eso no lo hacemos nosotros, lo hace el propio .NET.

Las ventajas de este sistema son evidentes. No podemos salirnos, si intentamos escribir fuera de rango el entorno de ejecución nos avisará. Si nos olvidamos de liberar el bloque, cuando *s* salga de ámbito (es decir, el valor almacenado dentro de *s* deje de existir porque haya sido eliminado de la pila), el recolector automático de basura entrará en marcha y liberará el bloque que ahora ha quedado huérfano.

Estas dos cosas no se pueden realizar con un puntero. Si el contenido de *pCad* se pierde, hemos eliminado la opción de recuperar el bloque, ya que el valor perdido es el que le dice al sistema qué tiene que liberar. Y escribir fuera es completamente trivial (y potencialmente catastrófico).

¿Qué es más interesante? Desde mi punto de vista, las referencias. Añadimos una indirección extra y posiblemente algo más de sobrecarga, pero eliminamos los dos errores más temidos de un programador de C++: olvidarnos de liberar recursos y sobreescribir fuera de rango.

Proxyes

La mayor limitación en esto es que no tenemos elección sobre qué usar. Es decir, si trabajamos con objetos .NET (C++/CLI) tenemos que forzosamente utilizar referencias. Y si trabajamos con elementos nativos (C++) sólo podemos usar punteros.

Hasta que los chicos de Microsoft no se decidan a implementar en el lenguaje referencias nativas y punteros manejados (cosa que Herb Sutter ha citado alguna vez como posible), tenemos que trabajar así.

Pero existe una forma para poder hacer lo del párrafo anterior y, aunque es ineficiente, es posible. Y es la utilización de Interop mediante objetos proxy. En otro momento hablaremos de ello (más o menos cuando salga Orcas, que trae algo sobre el tema).

Punteros interiores

Pero no todo dentro del mundo manejado son limitaciones a la hora de manejar referencias. En cierta medida las podemos controlar como punteros… siempre y cuando no nos salgamos del objeto.

Es decir, podemos tener un *puntero interior* al interior de un objeto, y podremos trabajar con él de forma idéntica a los punteros siempre que no nos salgamos del interior del mismo.

Quien quiera ver algo por su cuenta, aquí y aquí tiene referencias sobre cómo operan los punteros interiores. También puede esperar a que ponga alguna entrada sobre ellos aquí.