## (C++/CLI III): Los operadores *new* y *gcnew*

El concepto de clase lleva íntimamente asociado el concepto de constructor, tanto, que la primera no puede existir sin el segundo. Cuando obtenemos un objeto a partir de la definición de su clase correspondiente, se ejecuta el constructor adecuado que hayamos especificado, ya sea directa o indirectamente.

La relación constructor/clase es tan íntima que si el programador no especifica uno, el compilador creará uno *por defecto* que se encargará de inicializar todos los elementos de la clase con sus valores por defecto.

En C++ la instanciación (no me gusta nada la palabra, pero es la que consuetudinariamente se ha venido utilizando normalmente, así que es la que usaré) de una clase se realiza con el operador *new*, como se indica en el ejemplo:

```
Class Papel
{
    int x,y;
  public:
    void Escribe(char *,int tam){…}
};
…
Papel *miPapel=new Papel();
```

La línea que nos interesa es

```
Papel *miPapel=new Papel();
```

El operador *new* pide al sistema operativo un bloque de memoria capaz de contener un objeto del tipo *Papel* y posteriormente ejecuta el constructor de la clase. Como en nuestro caso no hemos indicado uno, ha sido el compilador el que lo ha implementado por nosotros, asignando el valor cero a *x* e *y* (o más bien llamando a los constructores por defecto de los tipos agregados; en C++ sólo se llaman para las clases, no para los tipos nativos; sin embargo, en C++/CLI el constructor se llama para todos los elementos, ya que todos son clases, incluidos los que pretenden ser nativos –que realmente no lo son).

En C++ y C++/CLI, si indicamos cualquier tipo de constructor, el compilador ya no nos creará uno por defecto, y si queremos uno de ese tipo, tendremos que indicarlo nosotros mismos.

Como todo operador que se precie, *new* puede ser sobrecargado, tanto a nivel de clase como globalmente, para poder así ajustar a nuestro gusto la asignación de memoria. Volviendo a nuestro primer artículo, podríamos querer que para una clase las asignaciones fueran realizadas sobre el montículo global. Pues nada, sobrecargamos el operador para esa clase y realizamos las asignaciones sobre *GlobalAlloc*. De momento no vamos a profundizar más en la sobrecarga de *new*, ya que explicar lo básico seguro que llega para un artículo o dos de estas dimensiones. Simplemente comentar que se puede hacer.

Pero C++/CLI no nos permite sobrecargar su equivalente, *gcnew*. ¿Y por qué?, se preguntarán. Pues es muy sencillo: en el .NET Framework la política de asignación de memoria está predefinida y marcada por el motor de tiempo de ejecución. No podemos sobrecargar *gcnew* porque sólo se nos permite una forma de asignar memoria. Podría existir una vía para soslayar

esto y es, utilizando las *APIs para hospedaje de CLR*, modificar dichos valores, pero en este caso ya no estamos operando con un *gcnew* sobrecargado, sino con las tripas del .NET (Quien quiera echar un vistazo rápido a dichas APIs, tiene un artículo de la MSDN traducido por mí y que puede leer aquí).

Conforme el lector va avanzando en estas explicaciones se va dando cuenta de que todas las decisiones de diseño que cambian el comportamiento de C++/CLI sobre C++ están perfectamente motivadas y justificadas. Quizás les gustaría echar un vistazo a los *Fundamentos lógicos del diseño de C++/CLI* de Herb Sutter(en donde se explican de forma razonada muchos detalles sobre la implementación de este lenguaje), pudiendo leerlo en castellano en otra traducción mía aquí.

***

Cuando trabajamos con variables situadas en el montículo, C++ trabaja casi igual que con variables alojadas en la pila, pero C++/CLI no. Y aquí debemos profundizar un poco en esto.

En C++/CLI existen dos tipos de clases. Los tipos-valor y los tipos-referencia. Los tipos valor se asignan en la pila. Los otros en el montículo. Los primeros heredan automáticamente de *System::ValueType* (y todos los tipos nativos lo hacen de esa clase). Los segundos heredan de *System::Object* (el que *System::ValueType* herede de *System::Object* es uno de los *misterios sin resolver* del CLI, ya trataremos eso en otro momento). Veamos un ejemplo.

```
//Tipo-valor
value class Complejo
{
  int real,int imag;
};
```

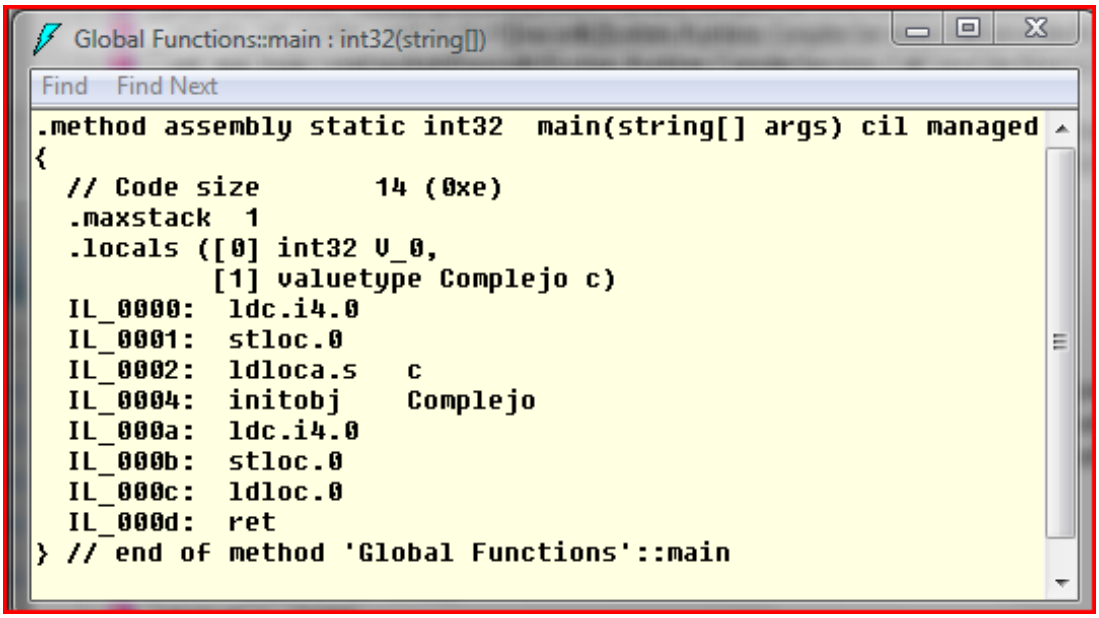
```
//Tipo-referencia
ref class Complejo
{
  int real,int imag;
};
```

El lector no observará ninguna diferencia salvo en la declaración, pero las hay, y no pocas. En un futuro artículo las trataremos. Ahora sólo nos interesa ver cómo trabaja el operador *gcnew*. Primero pongamos un ejemplo completo de cada tipo:

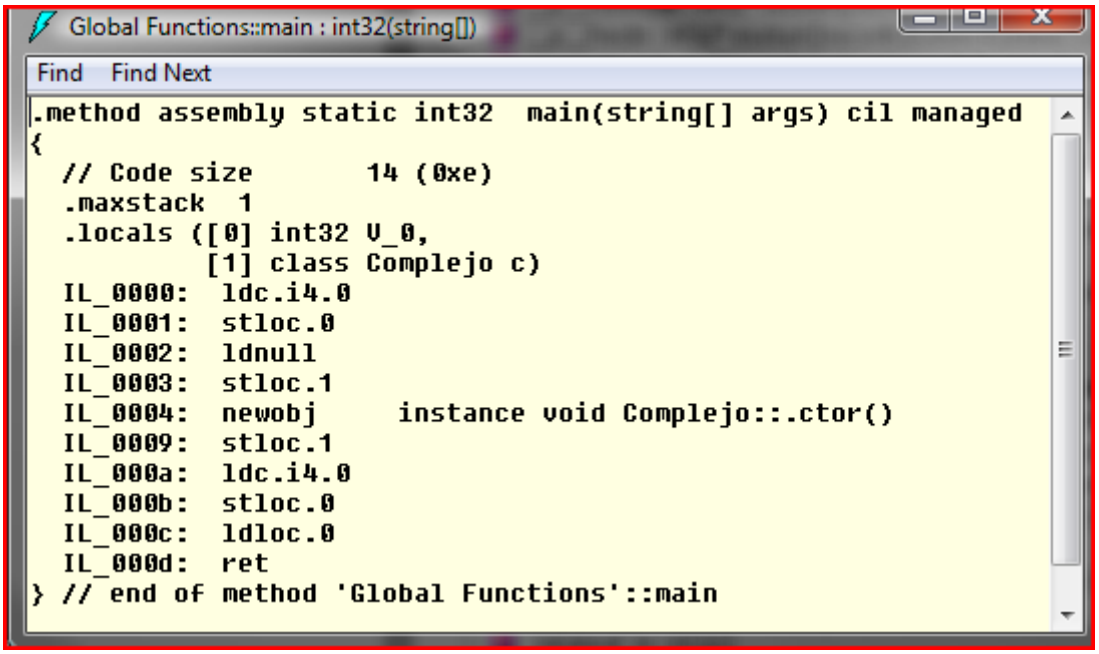
```
value class Complejo
{
public:
      int real,imag;
};

int main(array<System::String ^>
^args)
{
      Complejo c;
      return 0;
}
```

```
ref class Complejo
{
public:
      int real,imag;
};

int main(array<System::String ^>
^args)
{
      Complejo ^c=gcnew Complejo;
      return 0;
}
```

Veamos qué ha generado el compilador para el tipo-valor:




Y ahora para el tipo-referencia:



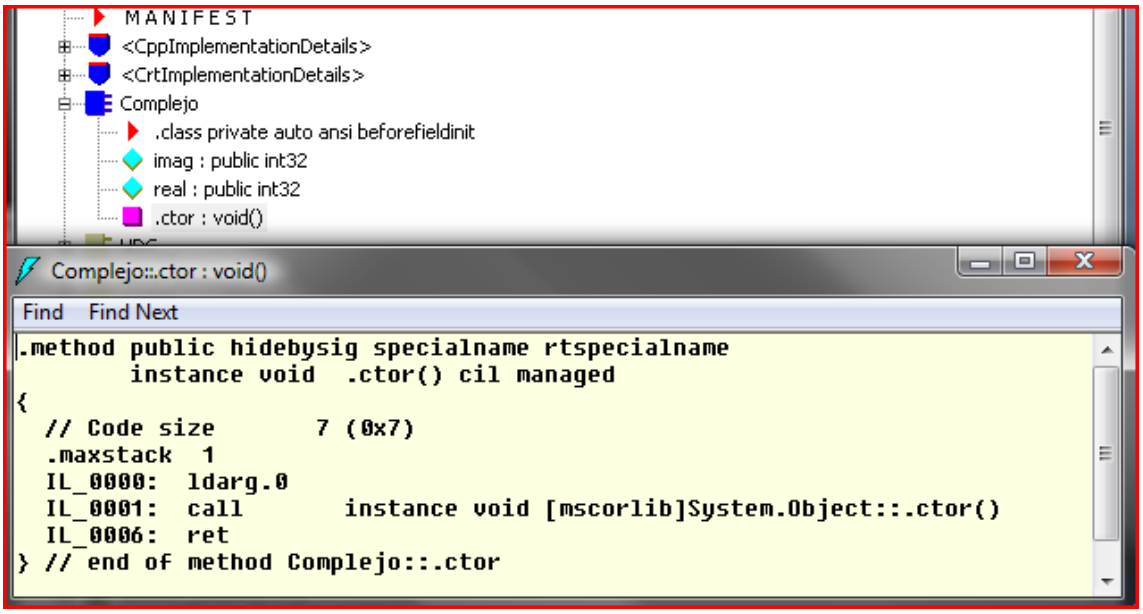

Curioso, ¿no?

Echemos un vistazo en primer lugar a la sección marcada como *.locals*. En ambos casos observamos que la segunda variable local es, en la primera imagen, un "*valuetype Complejo c"*, y en la segunda, un "*class Complejo c"*. Esos son los huecos que se van a reservar en la pila manejada. En el caso de la clase-valor, se trata del propio tipo, y en de la clase referencia, del manejador (o referencia, como queramos llamarlo).

También podemos observar cómo en el bloque de la instanciación de la clase por referencia cargamos el valor *nullptr* sobre el manejador de la misma, por lo que resulta redundante hacer nosotros mismos la asignación.

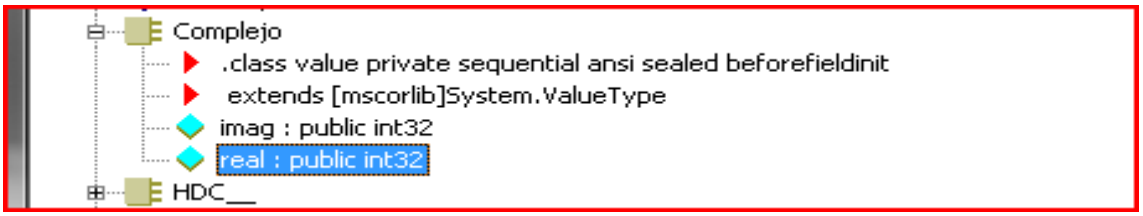
Y luego viene lo interesante de verdad. Un objeto-valor queda instanciado mediante la palabra reservada *initobj*. Y el objeto mediante *newobj*. Lo que nos indica que son cosas totalmente diferentes, aunque los pasos a seguir dentro de esas dos instrucciones sean más o menos iguales.

Primero se reserva memoria suficiente para contener el objeto. En el caso de la clase-referencia, el espacio ya está reservado en la pila. Luego se llama al constructor adecuado. Como no hemos especificado ninguno, el compilador ha creado uno por nosotros:




En la imagen podemos ver el tipo *Complejo* definido con su tipo, sus dos variables internas y el constructor citado. Más abajo vemos cómo se ha implementado dicho constructor, que al tratarse de un tipo trivial únicamente llama al de la clase padre.

En el caso del tipo por valor no hay constructor porque se trata de un tipo trivial que sólo implementa dos variables enteras:



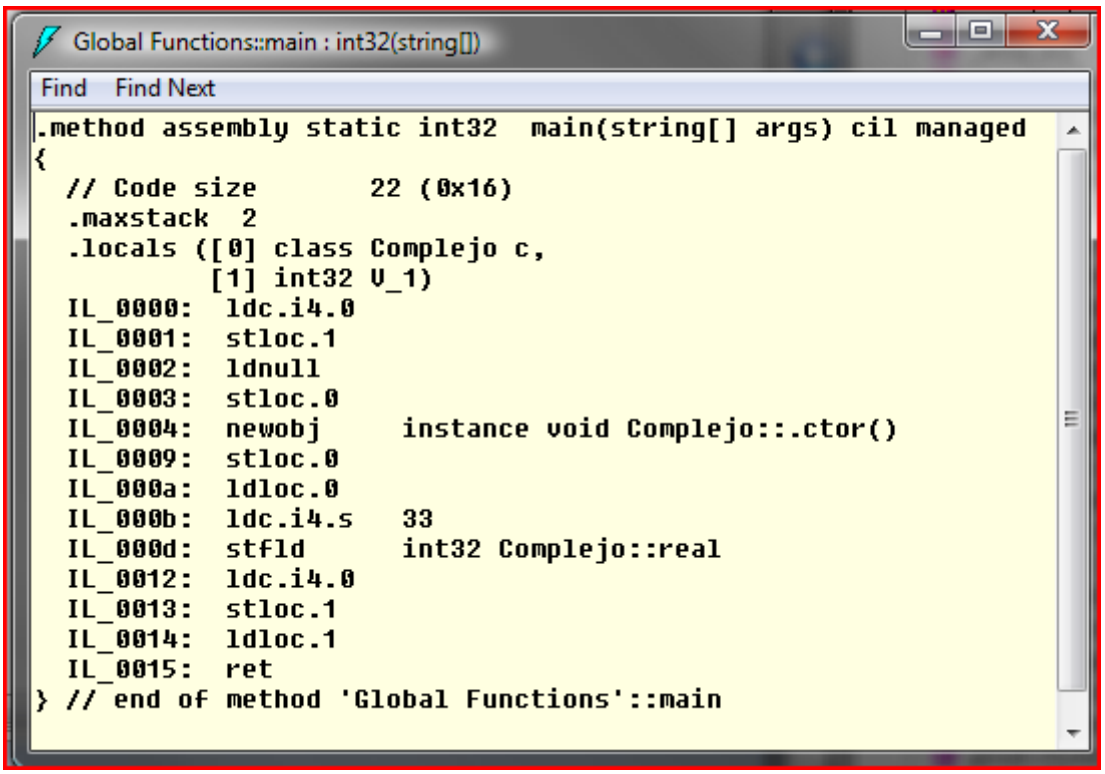

Después de ejecutar el constructor o de asignar los valores adecuados, si todo ha ido bien, entonces se asigna el manejador a la variable correspondiente. Si se ha producido algún problema, el valor de la referencia será *nullptr* siempre y cuando hayamos controlado la excepción dentro del constructor. En el ejemplo, si hubiera algún problema dentro del constructor, no volveríamos a *main()*, ya que la excepción no controlada sería elevada al manejador global, y entonces nos daría igual el valor de la referencia, puesto que estaría inalcanzable.

Y ahora demos otra vuelta de tuerca. Crucemos los ejemplos (Se recomienda a los programadores de C# no realizar este tipo de cosas con sus herramientas, podrían entrar en *La Dimensión Desconocida*. Avisados quedan. ☺ ).

```
value class Complejo
{
public:
      int real,imag;
};

int main(array<System::String ^>
^args)
{
      Complejo ^c=gcnew Complejo;
      c->real=33;
    return 0;
}
```

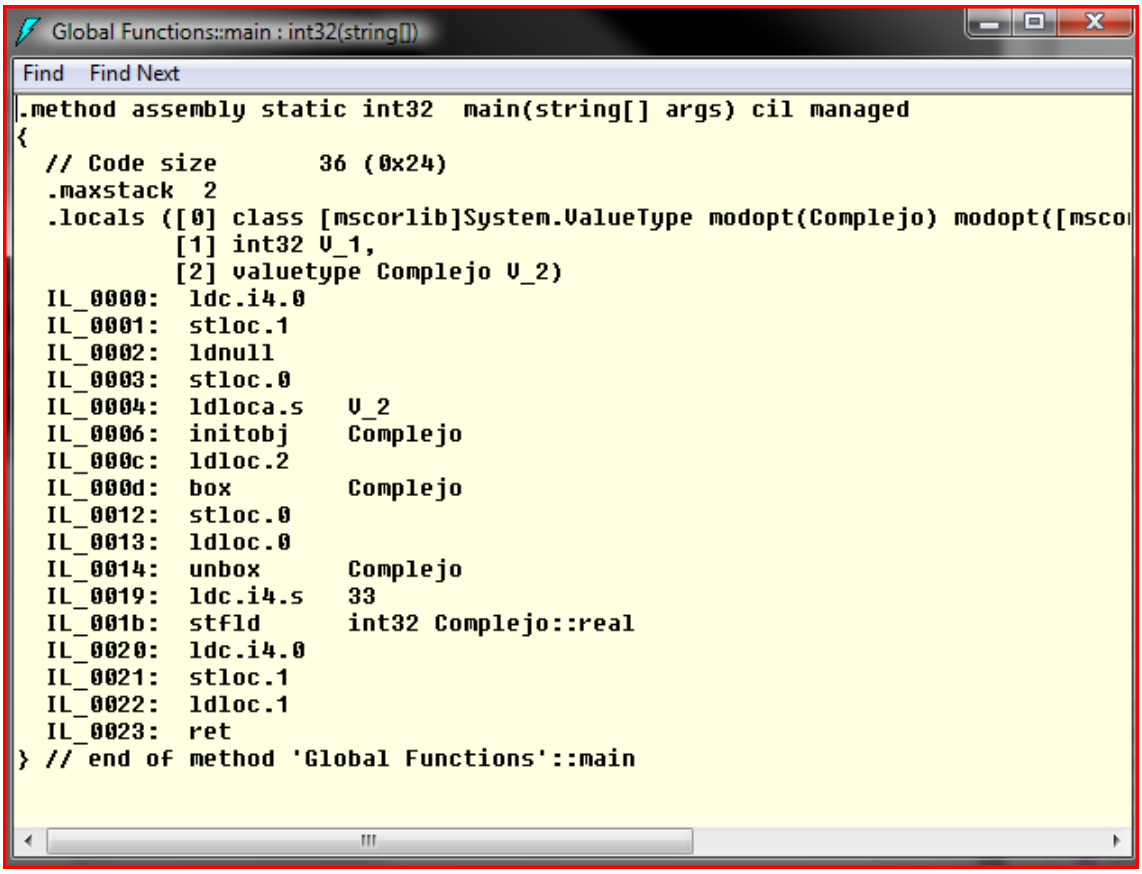
```
ref class Complejo
{
public:
      int real,imag;
};

int main(array<System::String ^>
^args)
{
      Complejo c;
      c.real=33;
    return 0;
}
```

Veamos qué ocurre si instanciamos un tipo-referencia en la pila, es decir, qué código genera la columna de la derecha:




¡No hay diferencia entre instanciarlo en la pila o el montículo! Efectivamente, el compilador de C++/CLI no engaña de malos modos. Por definición el .NET no permite que puedan existir objetos-referencia en la pila, pero como el C++ nativo no lo impide de forma alguna, los chicos de MS han hecho bien su trabajo y nos permiten trabajar con semántica de pila allí donde hay punteros. ¿No les esto suena de algo? Si, de hecho es casi equivalente a trabajar con las referencias de seguimiento que ya hemos visto. Cuando analicemos los destructores veremos qué significa todo esto para la destrucción determinista.

Ahora comprobemos el caso contrario (el bloque de código de la izquierda):




¡Buf, qué lío! Encajamientos, desencajamientos… esto merece un nuevo artículo, este también mostrando código en C#, ya que el tema es de un interés crucial para entender ciertos posibles cuellos de botella y caídas en el rendimiento general.