Por Francisco Morero

# Creación y destrucción de objetos

## El Constructor

Aunque el *constructor*, como ya hemos comentado anteriormente, es una función miembro, debido a su caracter especial hemos preferido tratarlo por separado. Al igual que haremos más adelante con el *destructor*.

Esta función miembro, es llamada automáticamente cada vez que se crea un nuevo objeto, su misión es asegurarse de que los datos de la clase serán correctamente inicializados; aunque a veces, suelen incluirse dentro del constructor otras tareas como la reserva dinámica de memoria (en caso de que la clase así lo requiera). Algunas peculiariedades del constructor son:

- El consturctor es invocado cada vez que el objeto entra en ámbito.

- Al declararlo como función miembro de la clase, **tiene** que tener el mismo nombre que la clase. Por jemplo:

```
Date::Date()
```

- El constructor puede recibir una lista de valores como parámetros. Por ejemplo:

```
Date OtraFecha( 12, 5, 1995 );
```

- Al declararlo, no se puede especificar el tipo de valor devuelto por el constructor, ya que el constructor no puede devolver ningún tipo de valor, ni siquiera *void*.

- El constructor también puede ser sobrecargado. De esta forma se pueden construir los objetos de tantas formas como constructores existan.

- En caso de que no se especifique un constructor al declarar una clase, C++ proporciona un por defecto consistente en reservar una zona de memoria con el tamaño necesario para albergar la información de la clase.

## El Destructor

Este método es la contrapartida del constructor. Es invocado automáticamente cada vez que el objeto sale de ámbito, y se encarga de realizar la tarea de *dejar todo como estaba* antes de que el objeto sea destruido, en caso de que fuese necesario.

Por ejemplo, si el objeto reservó memoria dinámicamente al ser creado, esta debe ser liberada antes de que el objeto sea destruido. Del mismo modo a como el constructor se encarga de la reserva de memoria el destructor se encarga de liberarla. Algunas peculiariedades del destructor son:

- Tiene el mismo nombre de la clase, pero con el signo **~** como prefijo. Por ejemplo: *Date::~Date()*

- Sólo puede haber un destructor por cada clase.

- No puede ser sobrecargado.

- No recibe parámetros ni devuelve valor alguno.

- Si no se especifica destructor alguno al declarar una clase (lo que es bastante habitual para clases simples), C++ proporciona un destructor por defecto que lo que hace es liberar la memoria que se utilizó para albergar el objeto cuando fue creado al entrar en ámbito.

## Sobrecarga del Constructor

En C++ es muy habitual sobrecargar el constructor de las clases para así disponer de más de un modo de creación de objetos para cada clase. El constructor se sobrecarga igual que cualquier otra función de C++.

Veamos un ejemplo. Para clases sencillas, es muy habitual disponer de un constructor que asume ciertos valores por defecto para todos sus parámetros, es decir, no recibe ningún parámetro; como es muy habitual, a este constructor se le llama en C++ *constructor por defecto*. Siempre que se pueda, es conveniente disponer de un constructor por defecto para todas las clases.

En nuestra clase, podemos crear un constructor que toma la fecha del sistema como valores de inicialización de sus datos.

```
// ---- Clase Date con sobrecarga de constructores --------- //
class Date
{
  public:
      Date();                  // Constructor sin parámetros

// Constructor con parámetros
      Date( int dia, int mes, int ano );
     ............
     ............
     ............
};

// ------------------------------------------------------//
// Constructor por defecto

Date::Date()
{
  GetSysDate( nDia, nMes, nAno );      // Se asume que existe
                                       // esta función
}                                 // en las librerías de nuestro
                                   // C++.

// Constructor sobrecargado

Date::Date( int Dia, int Mes, int Ano )
{
   SetDia( Dia );
   SetMes( Mes );
   SetAno( Ano );
}

// ------------------------------------------------------//
```

Como se puede ver, este segundo constructor llama a ciertos métodos de la clase *Date* para inicializar valores. Esto está totalmente permitido en C++, de hecho, los constructores pueden invocar cualquier método de la clase siempre y cuando estos métodos no tengan que consultar datos que aún no han sido inicializados.

Al primero de los dos constructores, se le conoce como el *constructor por defecto*, ya que no recibe ningún parámetro. Este constructor tiene un significado especial en C++, y como veremos más adelante, en caso de existir, es invocado automáticamente por el compilador en ciertas situaciones.

## Creación y destrucción dinámica de objetos

### El *Free Store*

En C++, el *Free Store* (almacenamiento disponible) es el equivalente al *heap* de C, es decir, la porción de memoria disponible durante la ejecución del programa (en *run time*).
La diferencia entre ámbas estriba en las funciones que se utilizan para acceder a esta zona de memoria.

Mientras que en C se utiliza *malloc()*, que devuelve un puntero *void* a la zona de memoria reservada (si la operación tuvo éxito, en caso contrario devuelve *NULL*), en C++, se utiliza el operador *new*.

La razón de no utilizar *malloc* es bien simple, y es que en C++, no sólo hay que reservar una zona de memoria, sino que además es necesario invocar al constructor de la clase que se está instanciando para que el objeto sea inicializado correctamente, y *malloc* no sabe nada acerca del tipo de varible que va a ser creada.

*malloc()*, no hace nada con la memoria, salvo reservarla, dejando en ella la basura que ésta contuviese. Bien es cierto que existen funciones en casi todos los compiladores de C, como es el caso de *calloc()*, que además limpia esa zona de memoria rellenándola a ceros. Sin embrago, esto sigue siendo insuficiente, ya que esta zona de memoria, en C++ tiene que contener información válida para el objeto creado, y de ésto es de lo que se encarga el(los) constructor(es) de la clase. Por ello es imprescindible que el constructor de la clase sea llamado cada vez que se crea un nuevo objeto.

Obviamente, aquellos datos creados dinámicamente en memoria tienen un alcance (*scope*) más allá de la función donde fueron creados, y existen, al igual que ocurre en C, hasta que son destruidos. Es decir, esta memoria no es liberada automáticamente al salir de ámbito.

### El Operador new

Al conocer el operador *new* la clase de la que se está instanciando un objeto, puede llamar al constructor de la clase. De hecho, dependiendo del número y tipo de parámetros especificados, *new* llamará a uno u otro constructor en caso de haber más de uno. Veamos un ejemplo de cómo se crean datos dinámicamente en C y C++.

```
// ---------- Creación de datos dinámicamente en C ----------//
   ........................................
   ........................................

  struct MyStruct *msData

  msData=(struct MyStruct *) malloc(sizeof ( struct MyStruct ) );
```

```
//  ------ Creación de datos dinámicamente en C++ ----------//

   ..............................................
   ..............................................

  Date *MyBirthday, *YourBirthday;

  MyBirthday   = new Date;                  // Se llama al constructor por
defecto
  YourBirthday = new Date( 3, 2, 67 );    // Otro constructor es llamado
```

Observe algunas de las diferencias entre *malloc* y *new*:

- *new*, a diferencia de *malloc()*, no es una función, por ello no lleva ni paréntesis ni argumentos. *new* es un operador que puede aplicarse a cualquier tipo de dato.

- *new* no necesita que se le informe del tamaño de memoria que deseamos reservar, porque *new* lo sabe. Es decir, no hay que usar el *sizeof()* del objeto para reservar su memoria.

- *new* devuelve un puntero, pero no es necesario moldear el tipo (*type casting*) del puntero al asignarlo a una variable. El compilador se encarga de comprobar que el tipo de puntero se corresponde con el del objeto que está siendo creado, generando un error en caso de que no se correspondan.

- Si *new* no puede reservar la memoria necesaria, devuleve 0, en lugar de NULL, como hace *malloc*. De echo, en C++ todo puntero nulo vale 0, en lugar de NULL.

**El operador delete**

*delete* es a *free()* lo que *new* es a *malloc()*. Para destruir los objetos que creamos en el ejemplo anterior y liberar su memoria, tan sólo hay que hacer lo siguiente:

```
        delete MyBirthday;
        delete YourBirthday;
```

Del mismo modo que *new* llama al constructor de la clase, *delete* llama al destructor antes de liberar la memoria.

Algunas precauciones al utilizar *delete* son las siguientes:

- *delete* sólo puede utilizarse para liberar memoria reservada con *new*.

- Dicha memoria puede liberarse una sola vez. Utilizar este operador para liberar memoria no reservada con *new*, o intentar borrar un puntero dos veces, provocará resultados inesperados y posiblemente el cuelgue del programa.

- Borrar (o liberar si se prefiere) un puntero nulo (léase nulo como 0, no como NULL) es perfectamente lícito en C++, y por lo tanto, no provoca ningún resultado colateral o imprevisto.

- Es responsabilidad del programador evitar estos errores, ya que el compilador no es capaz de detectarlos. Esto que parece muy simple, no lo es tanto, y con frecuencia se suelen comenter estos errores al principio de la programación en C++.

**Nota**: Los operadores *new* y *delete* pueden utilizarse tanto con los nuevos tipos de datos creados por nosotros como con los predeterminados (*built-in*). Lo que resulta especialmente interesante a la hora de crear arrays.

### El puntero this

Este es un putero especial que sólo es accesible desde las funciones miembro de una clase. *this* apunta al objeto para el cual ha sido invocada la función miembro que se está ejecutando. Cuando se llama a una función miembro para un objeto determinado, el compilador asigna la dirección del objeto al puntero *this* y después llama al método. *this* es como se le llama en C++ a *Self***.**

Este puntero puede utilizarse, entre otras muchas cosas para identificar objetos. Supongamos que tenemos una clase en la que alguno de sus métodos recibe como parámetro una objeto de esa misma clase, en tal caso sería más que conveniente poder determinar si el objeto pasado como parámetro al método es o no el propio objeto.

### Reserva dinámica de memoria virtual

En muchas ocasiones, cuando creamos porciones de código en C, simplemente deseamos acelerar procesos que trabajan con copias locales de datos y devuelven un resultado a CA-Clipper o CA-VO, pero en otras ocasiones, creamos datos que deseamos que permanezcan en memoria duarante largo periodo de tiempo, o en algunos otros casos, durante toda la ejecución de la aplicación.

En el primer caso, no tenemos que preocuparnos por la memoria, ya que la memoria que reservemos para realizar los procesos deseados, aun cuando sea tomada de la memoria base disponible para la ejecución de nuestra aplicación (los primeros 640 Kb), esta va a ser liberada automáticamente cuando cada función finalice.

Un problema bien distinto se nos plantea cuando deseamos reservar o bien grandes cantidades de memoria, o bien memoria durante un largo período de tiempo. En tal caso, no debemos (a veces ni tan siquiera podemos) tirar de la memoria base, y lo que hacemos es recurrir al API de CA-Clipper o CA-VO para reservar memoria virtual, el famoso VMM.

En ámbos casos, como ya comentamos anteriormente, no hay ningún problema en utilizar memoria virtual desde nuestras rutinas C++, siempre que lo deseemos, haremos exáctamente lo mismo que hacíamos en C tradicional, pero ahora haremos la reserva de memoria en el constructor y la liberación de la memoria en el destructor.

Un problema especial aparece cuando deseamos que el propio objeto (además de la memoria que este maneja) resida en memoria virtual. Podemos crear objetos dinámicamente en memoria virtual en CA-VO, ya que en este caso es MS-Windows el que está soportando el manejo de memoria, para ello, es necesario que hayamos construido nuestro ejecutable para funcionar bajo MS-Windows.

Con CA-Clipper el problema es distinto ya que el operador *new* no sabe nada del VMM de Clipper. Pero tampoco debe preocuparnos esto si recordamos lo que ya se dijo al comienzo de esta exposición a propósito de la memoria y los objetos y sobre todo, si diseñamos apropiadamente nuestro código C++. De todos modos, y para aquellos que áun permanezcan excepticos, zanjaré este tema diciendo que los operadores *new* y *delete*, pueden ser redefinidos, para que realicen otras tareas (llamar a *_xgrab()* por ejemplo), y estos operadores, aun después de ser redefinidos, seguirán llamando al constructor y al destructor del objeto.

# Algunos ejemplos de uso común

## CA-Clipper y CA-VO

De la frecuente necesidad de utilizar código C en nuestras aplicaciones CA-Clipper, creo que no es necesario comentar nada, basta con echar un vistazo a las ofertas disponibles de productos de terceros, sin embargo, el caso de CA-VO, es diferente.

CA prometió desde el principio que su nuevo compilador sería tan rápido como lo es el C, siempre y cuando se tomasen ciertas precauciones. De hecho, es cierto que todo aquel código que es compilado por VO a leguaje máquina -lo que solemos llamar *código nativo*-, es muy rápido y, sino tan rápido como C, casi. Por otra parte, VO proporciona acceso a todo el API de MS-Windows además de un buen número de nuevos operadores (entre los que se hayan los de manipulación de bits) además de un buen número de otras filigranas. Por lo que no es casi nunca necesario incluir código C en nuestras aplicaciones CA-VO.

Un aparente problema surje cuando creamos clases, ya que la OOP en este lenguaje aparentemente es siempre traducida a *p-code*. Y digo que este problema no lo es tal, porque, como ya comentamos al principio de esta exposición, la OOP adquiere su verdadera potencia (que no rapidez) cuando es interpretada o pseudo-interpretada.

Otro problema, y éste algo más peliagudo, surje del hecho de que VO no dispone, a diferencia de C++, de herencia mútiple. Ante esto caben dos posturas:

- Por un lado están los que afirman -no sin falta de fundamento- que todo sistema de herencia múltiple es reducible a un sistema de herencia simple más eficiente, lo que no quiere decir que sea más simple, ya que suelen complejificarse al reducirlos. Les pasa algo parecido a lo que ocurre con las bases de datos al llevarlas a sus formas normales.

- Por el otro lado, estamos los que opinamos que las herramientas tinen que proporcionarnos medios, y que somos lo que las sufrimos, los que debemos decidir si utilizamos o no estos medios. A este respecto tengo un ejemplo muy simple. Hace algún tiempo, estuve dando un curso de C++ a un grupo de profesores de informática, y uno de ellos, había estado estudiando una teoría linguística bastante compleja que resultaba relativamente fácil implementable desde la OOP con herencia múltiple. El no disponía de tiempo como para conocerla igual que su creador y después además reducirla a un modelo de herencia simple, aunque quería probar su validez implementándola someramente. Si no hubiese ninguna implementación de OOP con herencia múltiple, este hombre se hubiera quedado con las ganas de probar aquella teoría.

En cualquier caso, y como quiera que fuere, ya que en CA-VO no es habitual incluir código C en la mayoría de los casos, sólo daremos un ejemplo muy simple de conexión con C++. Quede claro que el resto de los ejemplos son fácilmente transcribibles para que funcionen bajo CA-VO.
En los ejemplos que siguen, comenzaremos desde lo más simple e iremos avanzando hasta cubrir todo lo que hasta ahora hemos visto casi exclusivamente desde un punto de vista teórico.

## Algo básico: La clase Date

Puesto que esta es la clase que hemos utilizado como ejemplo a lo largo de toda esta exposición, nos remitiremos al código que se acompaña en el disquete adjunto. Esta clase, la vamos primero a compilar y enlazar desde C++, con la intención de hacerlo lo más simple posible. En el siguiente ejemplo, la utilizaremos desde Clipper. **Nota**: En este ejemplo no se ha seguido la notación húngara intencionadamente.

## Objetos como miembros

Del mismo modo en que utilizamos cualquier tipo de variable (*int*, *char*, *struct*, ... ) para crear los datos de una clase, podemos utilizar objetos de otras clases como objetos miembros. Al hecho de crear nuevas clases utilizando otras clases como componentes, se le llama *composición*, y a la clase *compuesta*. También se les suele llamar a las clases compuestas *contenedoras* y a las albergadas por las primeras *contenidas*.

El objeto de la clase contenida no será inicializado (construido) hasta que no se construya el objeto de la clase contenedora.

Para llamar al constructor de un objeto miembro es necesario especificar un *inicializador miembro*, lo que se hace poniendo un dos puntos (**::**) después de la lista de parámetros del constructor de la clase contenedora.

A este constructor se le llama como a una función, cuyo nombre es el nombre del objeto miembro, y dependiendo del tipo y número de argumentos que esta función reciba, uno u otro constructor de la clase contenida será llamado.

Sin embargo, y como se dijo antes, es obligatorio especificar un inicializador miembro, no asumiendose el constructor por defecto de la clase contenida.

Todo esto que parece críptico y enrevesado, es bastante simple, y quedará mucho más claro con el siguiente ejemplo (sólo hemos incluido la declaración la implementación de la clase, de su utilización, hay un fichero separado de ejemplo):

```
// ------------------------------------------------------- //
#include <stdio.h>
#include <string.h>

// ------------------------------------------------------- //
// Clase Contenida
// ------------------------------------------------------- //

class Edad
{
   public:
// El constructor puede ser una función inline
      Edad() { nEdad = 0; }
// Constructor sobrecargado
      Edad( int nInit ) { nEdad = nInit; }

      int  GetEdad() { return nEdad; }
      void Display() { printf( "%d\n", nEdad ); }

   private:
      int nEdad;
};

// ------------------------------------------------------- //
// Clase Contenedora
// ------------------------------------------------------- //

class Sujeto
{
  public:
      Sujeto();                 // Constructor por defecto
      Sujeto( char * spInitName, int nInitAge );
      void Display( char * spComent );
```

```
  private:
      char spName[ 25 ];
      Edad nAge;
};

// ------------------------------------------------------- //
// Constructor por defecto

Sujeto::Sujeto() : nAge( 0 )
{
   strncpy( spName, "Juan Nadie", 25 );
}

// ------------------------------------------------------- //
// Observe cómo en el constructor de esta clase se llama
// al constructor de la clase contenida.

Sujeto::Sujeto(char*spInitName, int nInitAge) : nAge(nInitAge)
{
   strncpy( spName, spInitName, 25 );
}

// ------------------------------------------------------- //

void Sujeto::Display( char * spComent )
{
   printf("%s\nNombre:%s,Edad:%d\n",spComent,spName,nAge.GetEdad());
}

// ------------------------------------------------------- //
```

### Acceso desde Clipper a Objetos de C++ y Reserva dinámica de memoria

Este ejemplo se divide en dos partes:

- En la primera se implementa un manejador de ratón muy simple, y se muestra cómo utilizar éste desde Clipper. Para ello se sigue el siguiente proceso (desde el más bajo al más alto nivel):
    - Una clase C++ accede a todas las funciones de bajo nivel del driver de ratón mediante la interrupción 33Hex. Esta clase facilita el acceso a estas funciones y almacena el estado del ratón.
    - Un grupo de funciones de C llamables desde Clipper crea un objeto ratón único y accede a él. Le solicita acciones y/o devuleve el estado del ratón a la aplicación Clipper.
    - Una clase creada con HighClass hace de envoltorio de los dos módulos anteriores, resultando muy cómodo y manejable el ratón desde aplicaciones Clipper.
- En la segunda parte de este ejemplo, se muestra cómo crear y destruir objetos de C++ dinámicamente (en tiempo de ejecución) desde Clipper. Además se muestra cómo almacenar objetos en un array de C++ y algunas de las técnicas utilizadas para manejar datos estáticos. Para ello se crean objetos que no son más que asteriscos que son mostrados en pantalla tras ser creados, destruyéndose en orden inverso al que se crearon.

## Despedida y cierre

Cuando ya llevaba escritas casi treinta páginas y no faltaban muchos días para que este escrito entrase en imprenta, me ocurrió algo que todo aquel que alguna vez haya tenido que publicar su trabajo conoce perfectamente: me sobrevino un ataque de pánico.

Entonces, como casi siempre que algo se me viene encima, llamé a mi amigo Marino y le descargué mis penas como una ametralladora de fácil gatillo expulsa su munición: El tema de mi conferencia es excesivamente amplio, no puedo reducirlo más de lo que ya lo he hecho, llevo escrito temario para hablar cuatro horas y sólo tengo una y media para exponerlo, aún me quedan un montón de cosas que decir y no se qué hacer.

Marino, tras escucharme pacientemente, como siempre, me dio ánimos y me sugirió que si no me quedaba más remedio que eliminar algo, pulsase [**Ctrl-Inicio]**, leyese el título de mi escrito y quitase aquello que no se ajustaba al propósito de la exposición.

Como siempre, él tenía razón, me di cuenta que estaba intentando hacer un curso de C++ en lugar de intentar hacer una introducción al C++ encaminada a explicar cómo se conectan éste y Clipper. Así que quité todo aquello que no fuese, a mi juicio, absolutamente imprescindible para conseguir este objetivo, y añadí (de todo lo que me faltaba) sólo aquello que se ceñía a tal propósito.

Por supuesto que se han quedado muchas cosas fuera, demasiadas para mi gusto, y algunas de ellas muy importantes, como son la sobrecarga de operadores, la herencia, en todos sus modos, métodos virtuales, y un sin fin de cosas más. Pero el objetivo inicial consistía en proporcionar, a aquellos que lo pudieran necesitar, los medios para escribir ese código C que casi todas nuestras aplicaciones Clipper contienen, en C++ en lugar de C.

Espero haber eliminado con acierto los aspectos de la OOP en C++ que son menos necesarios para este tipo de trabajo y en cualquier caso, siempre pido disculpas si no tuve buen tino en la misma medida en la que animo a todos aquellos interesados a que sigan profundizando en este tema. Os aseguro que muchas veces, en la soledad de mi habitación, he disfrutado aprendiendo C++ tanto como se supone que los entendidos lo hacen contemplando un cuadro o escuchando una pieza musical, es decir con verdadera pasión.