CONSTRUCTORES Y DESTRUCTORES

CONSTRUCTORES

Los constructores son funciones miembro especiales que se llaman de modo automático al crear los objetos, y pueden ser usadas para dar valor inicial a cada una de las variables miembro. El ***nombre del constructor*** es siempre el mismo que el ***nombre de la clase***. Así el constructor de la clase ***C_Cuenta*** se llamará, a su vez, ***C_Cuenta***. Además, los constructores se caracterizan porque *se declaran y definen sin valor de retorno*, ni siquiera ***void***. Utilizando las capacidades de sobrecarga de funciones de C++, para una clase se pueden definir ***varios constructores***.

El uso del ***constructor*** es tan importante que, en el caso de que el programador no defina ningún constructor para una clase, el compilador de C++ proporciona un ***constructor de oficio*** (también llamado a veces ***por defecto***).

Si la clase ***C_Cuenta*** se completa con su ***constructor***, su declaración quedará de la siguiente forma:

```
class C_Cuenta {
    // Variables miembro

    private:
        double Saldo;      // Saldo Actual de la cuenta
        double Interes;    // Interés aplicado

    public:
        // Constructor
        C_Cuenta((double unSaldo, double unInteres) {
          SetSaldo(unSaldo);
        SetInteres(unInteres);
        }


         // Acciones básicas
        double GetSaldo()
        { return Saldo; }

        double GetInteres()
            { return Interes; }

        void SetInteres(double unInteres)
            { Interes = unInteres; }

        void Ingreso(double unaCantidad)
            { SetSaldo( GetSaldo() + unaCantidad ); }

};
```

Conviene insistir en que el ***constructor*** se declara y define sin valor de retorno, y que en una clase puede haber ***varios constructores***, pues como el resto de las funciones puede estar sobrecargado. La definición del ***constructor*** de la clase ***C_Cuenta*** pudiera ser la que a continuación se presenta:

```
C_Cuenta::C_Cuenta(double unSaldo, double unInteres)
{
    // La clase puede hacer llamadas a los métodos
    //    previamente definidos
    SetSaldo(unSaldo);
    SetInteres(unInteres);
}
```

Teniendo en cuenta que el ***constructor*** es una ***función miembro***, y que como tal tiene ***acceso directo a las variables miembro privadas*** de la clase, el ***constructor*** también podría definirse del siguiente modo:

```
C_Cuenta::C_Cuenta(double unSaldo, double unInteres)
{
    // El constructor accede a las variables miembro y les asigna el
    //    valor de los parámetros
    Saldo   = unSaldo;
    Interes = unInteres;
}
```

CONSTRUCTOR POR DEFECTO Y CONSTRUCTOR CON PARÁMETROS CON VALOR POR DEFECTO

Se llama ***constructor por defecto*** a un constructor que no necesita que se le pasen parámetros o argumentos para inicializar las variables miembro de la clase. Un ***constructor por defecto*** es pues un constructor que no tiene argumentos o que, si los tiene, todos sus argumentos tienen asignados un valor por defecto en la declaración del constructor. En cualquier caso, puede ser llamado sin tenerle que pasar ningún argumento.

El ***constructor por defecto es necesario*** si se quiere hacer una declaración en la forma:

```
C_Cuenta c1;
```

y también cuando se quiere crear un ***vector de objetos***, por ejemplo en la forma:

```
C_Cuenta cuentas[100];
```

ya que en este caso se crean e inicializan múltiples objetos sin poderles pasar argumentos personalizados o propios para cada uno de ellos.

Al igual que todas las demás funciones de C++, el ***constructor*** puede tener definidos unos ***valores por defecto*** para los parámetros, que se asignen a las variables miembro de la clase. Esto es especialmente útil en el caso de que una variable miembro repita su valor para todos o casi todos los objetos de esa clase que se creen. Considérese el ejemplo siguiente:

```
class C_Cuenta {
    // Variables miembro
    private:
        double Saldo;      // Saldo Actual de la cuenta
        double Interes;    // Interés aplicado
    public:
        // Constructor
        C_Cuenta(double unSaldo=0.0, double unInteres=0.0)
        {
            SetSaldo(unSaldo);
            SetInteres(unInteres);
        }
        // Métodos
        double GetSaldo()
            { return Saldo; }
        double GetInteres()
            { return Interes; }
        void SetSaldo(double unSaldo)
            { Saldo = unSaldo; }
        void SetInteres(double unInteres)
            { Interes = unInteres; }
        void Ingreso(double unaCantidad)
            { SetSaldo( GetSaldo() + unaCantidad ); }
};
```

```
void main() {
    // Ya es válida la construcción sin parámetros
    C_Cuenta C0;                // unSaldo=0.0 y unInteres=0.0
    // También es válida con un parámetro
    C_Cuenta C1(10.0);          // unSaldo=10.0 y unInteres=0.0
    // y con dos parámetros
    C_Cuenta C2(20.0, 1.0);     // unSaldo=20.0 y unInteres=1.0
    ...
}
```

En el ejemplo anterior se observa la utilización de un mismo ***constructor*** para crear objetos de la clase ***C_Cuenta*** de tres maneras distintas. La primera llamada al ***constructor*** se hace sin argumentos, por lo que las variables miembro tomaran los valores por defecto dados en la definición de la clase. En este caso ***Saldo*** valdrá 0 e ***Interes*** también valdrá 0.

En la segunda llamada se pasa un único argumento, que se asignará a la primera variable de la definición del ***constructor***, es decir a ***Saldo***. La otra variable, para la que no se asigna ningún valor en la llamada, tomará el valor asignado por defecto. Hay que recordar aquí que *no es posible en la llamada asignar un valor al segundo argumento si no ha sido asignado antes otro valor a todos los argumentos anteriores* (en este caso sólo al primero). En la tercera llamada al ***constructor*** se pasan dos argumentos por la ventana de la función, por lo que las variables miembro tomarán esos valores.

CONSTRUCTOR DE OFICIO

¿Qué hubiera pasado si en la clase ***C_Cuenta*** no se hubiera definido ningún ***constructor***? Pues en este caso concreto, no hubiera pasado nada, o al menos nada catastrófico. El compilador de C++ habría creado un ***constructor de oficio***, sin argumentos. ¿Qué puede hacer un ***constructor*** sin argumentos? Pues lo más razonable que puede hacer es inicializar todas las variables miembro a cero. Quizás esto es muy razonable para el ***Saldo***, aunque quizás no tanto para la variable ***Interes***.

Así pues, se llamará en estos apuntes ***constructor por defecto*** a un constructor que no tiene argumentos o que si los tiene, se han definido con valores por defecto. Se llamará ***constructor de oficio*** al ***constructor por defecto*** que define automáticamente el compilador si el usuario no define ningún constructor. Ambos conceptos no son equivalentes, pues si bien todo ***constructor de oficio*** es ***constructor por defecto*** (ya que no tiene argumentos), lo contrario no es cierto, pues el programador puede definir ***constructores por defecto*** que obviamente no son ***de oficio***.

Un punto importante es que el compilador sólo crea un ***constructor de oficio*** en el caso de que el programador no haya definido ***ningún constructor***. En el caso de que el usuario sólo haya definido un ***constructor con argumentos*** y se necesite un ***constructor por defecto*** para crear por ejemplo un ***vector de objetos***, el compilador no crea este ***constructor por defecto*** sino que da un mensaje de error.

Los ***constructores de oficio*** son cómodos para el programador (no tiene que programarlos) y en muchos casos también correctos y suficientes. Sin embargo, ya se verá en un próximo apartado que en ocasiones conducen a resultados incorrectos e incluso a errores fatales.

CONSTRUCTOR DE COPIA

Ya se ha comentado que C++ ***obliga a inicializar*** las variables miembro de una clase llamando a un ***constructor***, cada vez que se crea un objeto de dicha clase. Se ha comentado también que el constructor puede recibir como parámetros los valores que tiene que asignar a las variables miembro, o puede asignar valores por defecto.

Existe un caso particular de gran interés no comprendido en lo explicado hasta ahora y que se produce cuando se ***crea un objeto*** inicializándolo a partir de ***otro objeto de la misma clase***. Por ejemplo, C++ permite crear tres objetos ***c1, c2*** y ***c3*** de la siguiente forma:

```
C_Cuenta c1(1000.0, 8.5);
C_Cuenta c2 = c1;
C_Cuenta c3(c1);
```

En la primera sentencia se crea un objeto ***c1*** con un saldo de 1000 y un interés del 8.5%. En la segunda se crea un objeto ***c2*** a cuyas variables miembro se les asignan los mismos valores que tienen en ***c1***. La tercera sentencia es una forma sintáctica equivalente a la segunda: también ***c3*** se inicializa con los valores de ***c1***.

En las sentencias anteriores se han creado tres objetos y por definición se ha tenido que llamar tres veces a un constructor. Realmente así ha sido: en la primera sentencia se ha llamado al ***constructor con argumentos*** definido en la clase, pero en la segunda y en la tercera se ha llamado a un constructor especial llamado ***constructor de copia*** (***copy constructor***). Por definición, el ***constructor de copia*** tiene ***un único argumento*** que es una ***referencia constante a un objeto de la clase***. Su declaración sería pues como sigue:

```
C_Cuenta(const C_Cuenta&);
```

Las sentencias anteriores de declaración de los objetos ***c2*** y ***c3*** funcionarían correctamente aunque no se haya declarado y definido en la clase ***C_Cuenta*** ningún constructor de copia. Esto es así porque el compilador de C++ proporciona también un ***constructor de copia de oficio***, cuando el programador no lo define. El ***constructor de copia de oficio*** se limita a realizar una ***copia bit a bit*** de las variables miembro del objeto original al objeto copia. En este caso, eso es perfectamente correcto y es todo lo que se necesita. Pronto se verá algún ejemplo en el que esta copia bit a bit no da los resultados esperados. En este caso el programador debe preparar su propio ***constructor de copia*** e incluirlo en la clase como un constructor sobrecargado más.

Además del ejemplo visto de declaración de un objeto iniciándolo a partir de otro objeto de la misma clase, hay otros dos casos muy importantes en los que se utiliza el constructor de copia:

1. Cuando a una función se le pasan objetos como ***argumentos por valor***, y
2. Cuando una función tiene un ***objeto como valor de retorno***.

En ambos casos hay que crear copias del objeto y para ello se utiliza el ***constructor de copia***.

NECESIDAD DE ESCRIBIR UN CONSTRUCTOR DE COPIA

Ha llegado ya el momento de explicar cómo surge la necesidad de escribir un ***constructor de copia*** distinto del que proporciona el compilador. Considérese una clase ***Alumno*** con dos variables miembro: un puntero a ***char*** llamado ***nombre*** y un ***long*** llamado ***nmat*** que representa el número de matrícula..

```
class Alumno {
    char* nombre;
    long  nmat;
    ...
};
```

En realidad, *esta clase no incluye el nombre del alumno*, sino sólo un puntero a carácter que permitirá almacenar la dirección de memoria donde está realmente almacenado el nombre. Esta memoria se reservará dinámicamente cuando el objeto vaya a ser inicializado. Lo importante es darse cuenta de que *el nombre no es realmente una variable miembro de la clase*: la variable miembro es un puntero a la zona de memoria donde está almacenado. Esta situación se puede ver gráficamente en la figura 1, en la que se muestra un objeto ***a*** de la clase ***Alumno***.

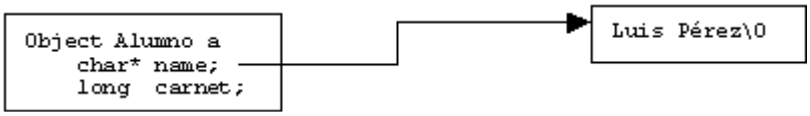

Figura 1. Objeto con reserva dinámica de memoria.

Supóngase ahora que con el ***constructor de copia*** suministrado por el compilador se crea un nuevo objeto ***b*** a partir de ***a***. Las variables miembro de ***b*** van a ser una ***copia bit a bit*** de las del objeto ***a***. Esto quiere decir que la variable miembro ***b.nombre*** contiene la misma dirección de memoria que ***a.nombre***. Esta situación se representa gráficamente en la figura 2: ambos objetos apuntan a la misma dirección de memoria.

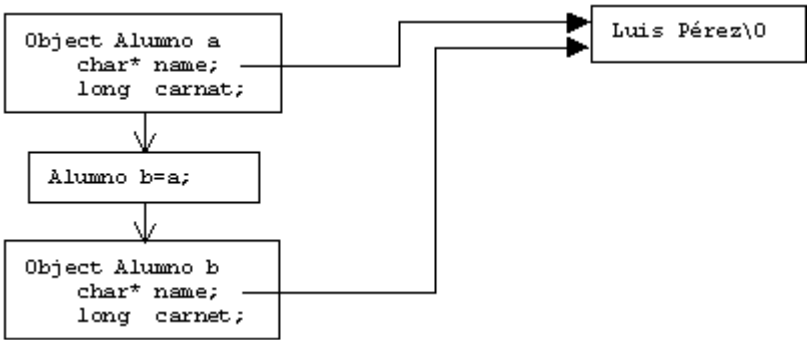

Figura 2. Copia bit a bit del objeto de la figura 1.

La situación mostrada en la figura 2 puede tener consecuencias no deseadas. Por ejemplo, si se quiere cambiar el nombre del Alumno ***a***, lo primero que se hace es liberar la memoria a la que apunta ***a.nombre***, reservar memoria para el nuevo nombre haciendo que ***a.nombre*** apunte al comienzo de dicha memoria, y almacenar allí el nuevo nombre de ***a***. Como el objeto ***b*** no se ha tocado, su variable miembro ***b.nombre*** se ha quedado apuntado a una posición de memoria que ha sido liberada en el proceso de cambio de nombre de ***a***. La consecuencia es que ***b*** ha perdido información y los más probable es que el programa falle.

Se llega a una situación parecida cuando se destruye uno de los dos objetos ***a*** o ***b***. Al destruir uno de los objetos se libera la memoria que comparten, con el consiguiente perjuicio para el objeto que queda, puesto que su puntero contiene la dirección de una zona de memoria liberada, disponible para almacenar otra información.

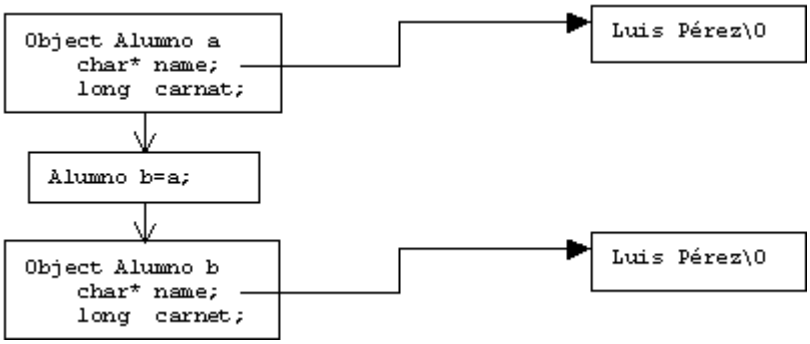

Figura 3. Copia correcta del objeto de la figura 1.

Finalmente, la figura 3 muestra la situación a la que se llega con un ***constructor de copia*** correctamente programado por el usuario. En este caso, el constructor ***no copia bit a bit*** la dirección contenida en ***a.nombre***, sino que reserva memoria, copia a esa memoria el contenido apuntado por ***a.nombre***, y guarda en ***b.nombre*** la dirección de esa nueva memoria reservada. Ninguno de los problemas anteriores sucede ahora.

LOS CONSTRUCTORES Y EL OPERADOR DE ASIGNACIÓN (=)

En un apartado anterior se ha visto una declaración de un objeto en la forma:

```
C_Cuenta c2 = c1;
```

Esta sentencia es completamente diferente de una simple ***sentencia de asignación*** entre dos objetos previamente creados, tal como la siguiente:

```
c2 = c1;
```

En este último caso no se llama a ningún ***constructor***, porque se supone que ***c1*** y ***c2*** existían previamente. En este segundo caso se utiliza el ***operador de asignación*** (=) estándar de C y C++, que permite realizar asignaciones entre objetos de estructuras o clases. Este operador realiza una ***asignación bit a bit*** de los valores de las variables miembro de ***c1*** en ***c2***. En este sentido es similar al ***constructor de copia*** y, por las mismas razones que éste, da lugar también a errores o resultados incorrectos.

La solución en este caso es ***redefinir*** o ***sobrecargar*** el ***operador de asignación*** (=) de modo que vaya más allá de la ***copia bit a bit*** y se comporte adecuadamente. En el apartado anterior se ha explicado a fondo el origen y la solución de este problema, común al ***constructor de copia*** y al ***operador de asignación*** (=). Básicamente, el ***operador de asignación sobrecargado*** (=) debe de llegar a una situación como la que se muestra en la figura 3.

DESTRUCTORES

El complemento a los constructores de una clase es el ***destructor***. Así como el constructor se llama al declarar o crear un objeto, el ***destructor*** es llamado cuando el objeto va a dejar de existir por haber llegado al final de su vida. En el caso de que un objeto (***local*** o ***auto***) haya sido definido dentro de un bloque {…}, el ***destructor*** es llamado cuando el programa llega al final de ese bloque. Si el objeto es ***global*** o ***static*** su duración es la misma que la del programa, y por tanto el ***destructor*** es llamado al terminar la ejecución del programa. Los objetos creados con ***reserva dinámica de memoria*** (en general, los creados con el operador ***new***) no están sometidos a las reglas de duración habituales, y existen hasta que el programa termina o hasta que son explícitamente destruidos con el operador ***delete***. En este caso la responsabilidad es del programador, y no del compilador o del sistema operativo.

A diferencia del ***constructor***, el ***destructor*** es siempre único (no puede estar sobrecargado) y ***no tiene argumentos*** en ningún caso. Tampoco tiene ***valor de retorno***. Su nombre es el mismo que el de la clase precedido por el carácter tilde (~), carácter que se consigue con ***Alt+126*** en el teclado del PC. En el caso de que el programador no defina un ***destructor***, el compilador de C++ proporciona un ***destructor de oficio***, que es casi siempre plenamente adecuado (excepto para liberar memoria de vectores y matrices).

En el caso de que la clase ***C_Cuenta*** necesitase un ***destructor***, la declaración sería así:

```
~C_Cuenta();
```

y la definición de la clase, añadiendo en este caso como variable miembro una cadena de caracteres que contenga el nombre del titular, podría ser como sigue:

```
        class C_Cuenta {

        // Variables miembro private:
        char   *Nombre;    // Nombre de la persona
        double Saldo;      // Saldo Actual de la cuenta
        double Interes;    // Interés aplicado

        public:
        //Constructor
        C_Cuenta(const char *unNombre, double unSaldo=0.0, double unInteres=0.0)
        {
        Nombre = new char[strlen(unNombre)+1];
        strcpy(Nombre, unNombre);
        Saldo = unSaldo;
        Interes = unInteres;
        }

        // Destructor
        ~C_Cuenta(){
           delete [] Nombre; }   // Libera la memoria apuntada por puntero

        // Métodos
        char *GetNombre()
            { return Nombre; }
        double GetSaldo()
            { return Saldo; }
        double GetInteres()
            { return Interes; }
        void SetSaldo(double unSaldo)
            { Saldo = unSaldo; }
        void SetInteres(double unInteres)
            { Interes = unInteres; }
        void Ingreso(double unaCantidad)
            { SetSaldo( GetSaldo() + unaCantidad ); }
};


void main(void)
 {
    C_Cuenta C0("Igor");
    // Tambien es valida con dos argumentos

    C_Cuenta C1("Juan", 10.0);
    //   y con los tres argumentos

    C_Cuenta C2("Itxaso", 20.0, 1.0);
    ...
}
```