Por Francisco Morero

Primeros aspectos de las clases en C++

Creación de nuevos tipos en C y C++

La primera implementación de C++ que pude ver no era sino una herramienta, que transformaba código C++ en código C puro. Con esto sólo quiero decir una vez más que la OOP no es un lenguaje en sí mismo, y que en C pueden simularse las clases mediante otros procedimientos.

De hecho, en muchas ocasiones, he oído decir a programadores de C que no hay ninguna o prácticamente ninguna diferencia entre una clase y una estructura (*struct*). La verdad es que, aunque sean bastante parecidas, hay muchas, pero sobre todo una fundamental, la *herencia*. De una *struct*, no se puede heredar, de una clase sí, y este "pequeño" detalle, no tiene precio.
Sin embargo, y puesto que tienen bastantes similitudes, podemos decir que mientras que en C se utilizan las estructuras para crear nuevos tipos de datos, en C++ se utilizan las clases.
En C, declaramos una estructura que contiene datos, y un grupo de funciones que permite manipular dichas estructuras. En C++ este concepto se mantiene, pero es llevado mucho más lejos. De hecho, en C++, los datos y sus funciones asociadas son la misma cosa, forman una misma entidad, a ésta la llamamos clase, y cada una de las instanciaciones que realizamos de esa clase (cada una de las variables que creamos del nuevo tipo de dato), les llamamos objetos.

Básicamente son la misma cosa, pero C++ proporciona una serie de mecanismos para la creación y el mantenimiento de estas "nuevas estructuras" que hacen la programación mucho más cómoda y sencilla.

Creación de clases

Y como hemos comentado que una clase es un nuevo tipo de dato, en el ejemplo que vamos a utilizar para estudiar la implementación OOPS del C++, crearemos un tipo de dato no existente en C, el dato tipo *fecha*.

Declaración de la clase

```
// Fuente 1 ---- Declaración de la clase Date ----------------------- //
class Date
{
   public:
      Date( int dia, int mes, int ano );     // Constructor
      void Display();           // Función miembro o Método
      ~Date();                               // Destructor

   private:
      int nDia, nMes, nAno;            // Datos miembro o Datos
};                 // OJO, la definición de clase termina con ";"
// ------------------------------------------------------- //
```

Lo que hemos hecho en el fuente 1 es simplemente declarar las variables de las que consta la clase y la declaración formal de las *funciones miembro* que la van a manejar (en este caso tres:

Date(), *Display()* y *~Date()*). Ahora podemos codificar estas funciones miembro. (Se hará de forma muy sucinta, dado por supuesto que se conoce C lo suficiente, y vaya por delante que cualquiera puede mejorar esta implementación).

Implementación de la clase

```
// Fuente 2 ------- Implementación de la clase Fecha --------------- //
// El constructor
Date::Date( int dia, int mes, int ano )
{
   nDia = Min(   31, Max( dia, 1 ) );
   nMes = Min(   12, Max( mes, 1 ) );
   nAno = Min( 9999, Max( ano, 0 ) );
}

// Método
Date::Display()
{
  printf( "Dia %d del %d de %d\n", nDia, nMes, nAno );
}

// El destructor
Date::~Date()
{
  // no hacemos nada por ahora
}
// ------------------------------------------------------ //
```

Llegados a este punto, me gustaría hacer notar los siguientes hechos:

- Existen porciones en esta clase que son *public*, y otras *private*. Más adelante veremos lo que son.

- Dentro de la clase se definen funciones. Lo que enfatiza la fuerte unión existente entre los datos y las funciones que los manejan.

- Para saber que una función pertenece a una clase, se antepone como sufijo el nombre de la clase, seguido del operador **::**.

- El constructor se encarga de que se inicialicen correctamente los datos de la clase a partir de los valores de inicialización que le llegan mediante sus parámetros (definidos formalmente en la clase). Esta es la misión del constructor: asegurarse de que se inicialicen correctamente los datos de la clase.

- El destructor se encarga de realizar aquellas operaciones que consideremos oportunas antes de que el objeto sea destruido. Un objeto es destruido cuando termina su ámbito (*scope*), a menos que haya sido creado dinámicamente en memoria (lo que veremos más adelante). Normalmente, el destructor se encarga de liberar aquella memoria que a veces algunas clases recogen en tiempo de ejecución. Es corriente que los constructores reserven memoria dinámicamente (llamada a *malloc()* o similar) y el destructor libere la memoria reservada (llamada a *free()* o similar).

- El destructor no es necesario declararlo sino se va a reescribir, ya que C++ proporciona un destructor por defecto.

Utilización de las clases

Una vez que una clase ha sido definida y codificada, se puede utilizar de forma muy similar a como se utiliza cualquier otro tipo de dato en C o C++.

```
      Date fecha1( 12, 03, 1995 );
      Date fecha2( 19, 06, 1961 );
```

incluso se pueden declarar objetos con fechas erróneas, ya que el constructor se encargará de corregir estos desajustes según el modo en como haya sido codificado:

```
Date fecha_erronea( 33, 14, -5 );
```

Esta instanciación de la clase *Date*, no daría ningún problema, ya que el constructor se asegura de que el objeto sea creado adecuadamente. Obsérvese también que la lista de parámetros pasados al constructor se corresponde con la definición formal del mismo hecha en la clase.
Como puede observarse, hay una gran similitud entre:

```
      int y = 5;
```

y

```
date f( 1,1,50 );
```

Sin embargo, me gustaría hacer una aclaración conceptual importante. No es lo mismo *instanciar* que *asignar*. Los datos de un objeto pueden asignarse siempre que se desee y la clase lo permita, pero un objeto sólo puede instanciarse (inicializarse o crearse si se prefiere), una sola vez.

Veamos ahora cómo se mostraría una fecha:

```
      fecha1.Display();
```

Como se puede observar, incluso la nomenclatura de C++ enfatiza el fuerte vínculo existente entre los datos y los métodos, es decir, entre los datos y las funciones que los manejan.

Visibilidad de los miembros de la clase

Las directivas *private*, *public* y *protected* indican la visibilidad de los miembros de la clase. La visibilidad establecida por una directiva se determina a partir de la misma hacia abajo para todas la declaraciones realizadas de todos los miembros de la clase (es decir, tanto datos como métodos), hasta que se encuentra otra directiva distinta.

Miembros *private*:

Estos sólo pueden ser accedidos por funciones miembro de la clase, es decir, por los métodos de la clase. (Para ser exactos, hay que decir que también pueden ser accedidos por un tipo especial de clases y funciones llamados *friend*).

Los miembros privados de una clase, gestionan el funcionamiento interno de la clase, es decir, la *implementación* de la clase.

Miembros *public*

A estos pueden acceder tanto las funciones miembro (métodos) de la clase como cualquier otra función del programa, siempre y cuando, claro está, el objeto exista en el ámbito (*scope*).
Los miembros públicos de una clase determinan la forma en la que el programa interactua con la clase y la gestiona, es decir, el *interfaz* de la clase.

Miembros *protected*

Estos son un tipo especial de miembros que se utilizan en la herencia. Estos miembros son visibles sólo dentro de la clase donde han sido declarados y en todas las clases heredadas de ella. De este modo, podemos considerarlos como un tipo especial de *private*. En cualquier caso, y debido al alcance de esta exposición, no los consideraremos.

Aunque se pueden utilizar tantas etiquetas *public*, *private* y *protected* como se desee, normalmente se utilizan sólo una *public* y una *private* (y una *protected* si fuese necesario), lo que permite, agrupar los miembros, dando mayor claridad al código.

Si al principio de una clase no se especifica ninguna etiqueta, se asume por defecto *private*, aunque es muy conveniente olvidarse de esta asunción y especificar claramente qué miembros son privados y cuáles públicos.

Es muy importante recalcar que aunque se puede hacer que algunos, o incluso todos los datos de una clase sean *public*, existe la recomendación con grado casi de norma de que todos los datos de las clases sean *private* y sólo puedan ser *public* los métodos (todos o algunos).

Existe un cualificador especial para los datos miembro en una clase, que puede aplicar a cualquiera de los tipos mencionados anteriormente, y es denotado por la palabra reservada *static*. Cuando a un dato es de tipo *static*, sólo se crea una copia de él al instanciar el primer objeto de la clase, no creándose ninguna otra copia de él por muchos objetos de la clase que se instancien.

Un dato de este tipo, puede servir, por ejemplo, para llevar un recuento de cuántos objetos se han instanciado de una clase determinada. Incrementando en el constructor de la clase el contador. Por ejemplo, véase el fuente 3:

```
// Fuente 3 ------Declaración de la clase TEST ----------------------//
class Test
{
   public:
       Test();
   ........................................
   ........................................

   private:
       static unsigned long nCount;
   ........................................
   ........................................
};

// ------------------------------------------------------ //
// Codificación de la clase

unsigned long Test::nCount = 0;
```

```
Test::Test()
{
   nCount++;
}
// ------------------------------------------------------//
```

Obsérvense en el ejemplo del fuente 3 los siguientes hechos:

- El dato *static* es declarado dentro del bloque *private*, y por lo tanto sólo es accesible desde dentro de la clase. De igual modo podría haberse declarado *public*.

- El constructor no inicializa este dato, ya que mientras que de este dato existe sólo una copia independientemente del número de instanciaciones que existan de la clase, el constructor es llamado para cada nueva instanciación de la clase.

- Los datos *private static* se inicializan del mismo modo a como se hace con los *public*, y esta inicialización se realiza en el fichero de implementación de la clase.

- La inicialización no se realiza en el fichero cabecera, ya que éste puede ser incluido más de una vez en un mismo programa.

Funciones *Miembro*

Aunque en C++ a los métodos se les suele llamar *funciones miembro*, a lo largo de esta exposición, como ya hemos venido haciendo, utilizaremos ambas expresiones indistintamente.

Básicamente, una función miembro es muy similar a cualquier otra función de C. Veamos primero las diferencias y después las similitudes.

- El prototipo del método aparece dentro de la definición de la clase.

- En la declaración se utiliza el operador resolución (**::**), lo que hace que el *scope* del método se circunscriba a su clase. Esto permite que puedan existir funciones con el mismo nombre que métodos, e incluso que distintas clases tengan métodos con el mismo nombre sin que haya conflicto entre ellos.

- Sólo las funciones miembro declaradas como *public* pueden ser invocadas desde fuera de la clase.

- Los métodos *private*, sólo pueden ser llamados desde otros métodos de la propia clase.

- Un método siempre es invocado asociado a un objeto existente. Para ello se utiliza el operador de envío (**.**). Por ejemplo:

```
  // Muestra la fecha del objeto fecha1
  fecha1.Display()

  // Muestra la fecha del objeto fecha2
  fecha2.Display()
```

Por lo demás, los métodos y las funciones de C++, se comportan de igual manera. Sin embargo, hay unos cuantos puntos de especial interés sobre los que me gustaría llamar la atención:

- Los métodos pueden sobrecargarse (*overloading*) como si se tratase de cualquier otra función C++.

- Es lícito llamar a un método, o consultar un dato, a través de un puntero al objeto. Obsérvese en este punto cómo se mantiene la similitud sintáctica con el C tradicional en cuanto al acceso a elementos de una *struct*. Por ejemplo:

```
Date fecha1( 12, 03, 1995 );

Date *fecPtr = &fecha1;

fecPtr->Display();
```

Acceso a los Datos

Los principios teóricos del modelo OOP implican que ningún dato pueda ser manipulado sino a través de un método y por lo tanto sólo son manipulables dentro de la propia clase. De este modo, se mantiene la encapsulación de las clases y por lo tanto se hace posible su posterior mantenimiento.

Para ello, debemos articular algún modo indirecto de consulta y modificación de los datos de la clase. La solución pasa por construir métodos de acceso (consulta) y métodos de modificación de los datos de la clase.

Me gustaría antes de seguir hablando de esto, recalcar la importancia que tiene que a los datos de una clase se acceda a través de un método, ya que los programadores de Clipper, solemos (a veces por pereza, a veces por exceso de celo en la optimización del código para que sea lo más rápido posible), pasarnos esta norma por alto. De hecho, Clipper, internamente accede a los datos de las clases, ya sea para consultarlos, o para modificarlos, a través de funciones C. Por otro lado, aquellos que conozcan *High Class*, sabrán que este motor OOP provee de unos especificadores especiales para declarar aquellos métodos de acceso y de modificación de los datos de una clase.

Evidentemente, siempre es más lento acceder a un dato a través de un método, que hacerlo directamente, pero si nos saltamos esta norma a la torera, nos estamos saltando parte del paradigma OOP, y quizás nos encontremos un día con que no podemos hacer las modificaciones necesarias en una clase porque no la encapsulamos debidamente cuando la creamos.

Esto, cuando se codifican clases de Clipper no es tan crucial como cuando se hace con C++, ya que en Clipper no existe una tipificación rígida de datos, pero recomiendo efusivamente que se haga con C++, especialmente si se piensa que es muy poco más el código necesario y que la pérdida de velocidad es absolutamente despreciable.

Para codificar los métodos de *consulta* de los datos de las clases, suele seguirse el siguiente patrón (nada de esto es obligatorio, pero es recomendable):

- Para que resulte un código más rápido de ejecución, y debido a que estas funciones requieren muy poco código, se escriben como funciones *inline*.

- Para que el código sea más claro, se codifican al declararlas dentro de la propia declaración de la clase.

- Se suele utilizar el prefijo *get* para identificarlas, seguido del nombre del dato al que se accede.

Veamos en el fuente 4 cómo quedaría nuestra declaración de la clase según esto:

```
// Fuente 4 ----Nueva declaración de la clase Fecha ----------------- //
class Date
{
 public:
     Date( int dia, int mes, int ano );      // Constructor

// Funciones miembro (métodos)
     int  GetDia() { return nDia; }
     int  GetMes() { return nMes; }
     int  GetAno() { return nAno; }
     void Display();

     ~Date();                  // Destructor

 private:
    int nDia, nMes, nAno;       // Datos miembro o Datos
    };
// ---------------------------------------------------------- //
```

De este modo, podemos cambiar toda la codificación interna de la clase sin que se vea afectado el resto del código que la utiliza.

Del mismo modo, para codificar los métodos de *modificación* de los datos de una clase, suelen seguir las siguientes normas:

- Si requieren poco código se escriben como los métodos *get*, es decir, con la directiva *inline* y dentro de la declaración de la clase. En caso contrario, se declaran dentro de la declaración de la clase (como es obligatorio), no utilizando la directiva *inline* y se codifican fuera.

- Se suele utilizar el prefijo *set* para identificarlos, seguido del nombre del dato al que se accede.

Veamos en el fuente 5 cómo quedaría tanto la declaración como la codificación de nuestra clase según esto:

```
// Fuente 5 ------ Nueva declaración de la clase Fecha ------------ //
// Declaración de la clase

class Date
{
  public:
      Date( int dia, int mes, int ano );     // Constructor

       // Métodos de consulta
      int  GetDia() { return nDia; }
      int  GetMes() { return nMes; }
      int  GetAno() { return nAno; }

       // Métodos de modificación
      void SetDia( int Dia );
      void SetMes( int Mes );
      void SetAno( int Ano );

      void Display();                 // Método de visualización

      ~Date();                        // Destructor
```

```
   private:
       int nDia, nMes, nAno;          // Datos miembro o Datos
};

// ------------------------------------------------------- //
// Codificación de la clase

Date::Date( int dia, int mes, int ano )
{
   SetDia( nDia );
   SetMes( nMes );
   SetAno( nAno );
}

void Date::SetDia( int Dia )
{
  nDia = Min( 31, Max( Dia, 1 ) );
}

void Date::SetMes( int Mes )
{
  nMes = Min( 12, Max( Mes, 1 ) );
}

void Date::SetAno( int Ano )
{
  nAno = Min( 9999, Max( Ano, 0 ) );
}

Date::~Date()
{
   // No hacemos nada, lo ponemos sólo para ver cómo se haría en caso de
necesitarlo.
}

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

Como se puede observar en el código aquí expuesto, mientras que los métodos *get* sólo devuelven el valor del dato al que acceden, los métodos *set*, comprueban la validez del parámetro recibido para asegurar la consistencia interna de los datos de la clase.

En este ejemplo, aunque los métodos *set*, debido a su tamaño, podían haberse declarado *inline*, no se ha hecho así intencionadamente, ya que no suelen serlo.

Ambito de los objetos

Los objetos, en cuanto a su ámbito (*scope*), podemos dividirlos en las siguientes categorías:

Locales

El constructor es invocado cuando el programa alcanza la sección de código de la función donde el objeto es declarado.

El destructor es llamado cuando el programa sale de dicha sección, es decir, sale de ámbito.

Estáticos

El constructor es invocado cuando el programa alcanza por primera vez la sección de código de la función donde el objeto es declarado.

El destructor es llamado cuando el programa finaliza.

Globales

El constructor es invocado cuando al comenzar el programa.

El destructor es llamado cuando el programa finaliza.

Como puede observarse, existen las misma categorías para ámbito de objetos que para el resto de los tipos de datos de C++. Es importante no confundir el ámbito de un objeto con el ámbito de los miembros de un objeto.

Las cabeceras y los ficheros de código fuente

Como es bien sabido, en C los ficheros cabecera suelen utilizarse para especificar las declaraciones de tipos definidos por el usuario, constantes, estructuras, etc. En C++, se utilizan con la misma finalidad, pero además, existe una práctica muy extendida que proviene del C. Como ya comentamos, el equivalente a las estructuras del C son las clases en C++, y del mismo modo a como en C la declaración de estos tipos se realiza en ficheros cabecera, en C++ la declaración de las clases se realiza en ficheros cabecera.

Mientras que en C, los ficheros cabecera suelen tener extensión .H, en C++, aquellos ficheros que contienen declaración de clases suelen notarse con la extensión .HPP

De este modo, mientras que en los ficheros .HPP se declaran las clases, en los ficheros .CPP se implementa la codificación de dichas clases.

Mientras que los ficheros cabecera quedan a disposición del programador que va a utilizar la clase, la codificación de la misma suele estar en ficheros .OBJ o .LIB, que impiden la manipulación inapropiada de la misma por parte del usuario de la clase.

De este modo, cada vez que se va a utilizar una clase al principio de fichero se hace un *#include <clase.hpp>*, cargando así todas las definiciones pertinentes a la clase.

Para evitar que por error se incluyan dos veces las definiciones contenidas en un fichero de cabecera, suele utilizarse la directiva:

```
#ifndef
      ...........
      ...........
      ...........
#endif
```

colocando entre ambas las declaraciones pertinentes. Por ejemplo, para la declaración de la clase *Date* en un fichero cabecera haríamos algo así como lo que vemos en el fuente 6:

```
// Fuente 6 -------- Fichero DATE.HPP ------------------------------- //
```

```
#ifndef _DATE_H_

#define _DATE_H_
//Lo que impide que vuelva a ser definida la clase

class Date
{
  public:
      Date( int dia, int mes, int ano );     // Constructor

       // Métodos de consulta
      int  GetDia() { return nDia; }
      int  GetMes() { return nMes; }
      int  GetAno() { return nAno; }

       // Métodos de modificación
      void SetDia( int Dia );
      void SetMes( int Mes );
      void SetAno( int Ano );

      void Display();                 // Método de visualización

      ~Date();                        // Destructor

  private:
      int nDia, nMes, nAno;           // Datos miembro o Datos
};

#endif
// ------------------------------------------------------- //


Tal y como hemos comentado, la codificación de la clase quedaría en un
fichero .CPP separado, con una apariencia similar a la del fuente 7.


// Fuente 7 -------- Fichero DATE.CPP ------------------------------ //
#include "date.hpp"

Date::Date()
{
  ......................
}

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

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

Por este número nada más. En el siguiente ofreceremos la última entrega de este extenso artículo, con la creación y destrucción de objetos y algunos ejemplos de uso común para Clipper y Visual objects.