C++ y su conexión con Clipper (I)

Por Francisco Morero

Planteamiento de objetivos y definición de ámbito

No pretendo, y aunque lo pretendiese, no podría, detallar pormenorizadamente en el poco espacio disponible, todos los entresijos del C++. Pero sí deseo aclarar desde ya que a lo largo de este artículo intentaremos alcanzar tres objetivos principales:

- Clarificar las diferencias entre C y C++.

- Introducir al lector en la *Programación Orientada a Objetos* (OOP de ahora en adelante) en C++.

- Proporcionar algunas técnicas tan básicas como eficaces para poder manejar clases de C++ desde Clipper.

A quiénes va dirigido este artículo

Cuando comencé a escribir este artículo, me planteé como objetivo primordial hacerlo lo más asequible posible. Creo haber hecho una aproximación bastante buena, aunque de usted depende en última instancia decantarse en uno u otro sentido. En cualquier caso, me temo que son necesarios algunos conocimientos previos tanto sobre OOP y sobre C como para poder sacarle todo el partido que pueda tener. No vamos a abordar por tanto, ni los principios teóricos de la OOP, que se dan por sabidos, ni los del C por el mismo motivo.

De todos modos, aquellos que por una u otra razón no se encuentren muy satisfechos con sus conocimientos de C++ y/o OOP, ahora disponen de una buena oportunidad para ir adentrándose en lo que va a ser casi seguro, el terreno por el que nos seguiremos moviendo en los próximos años.

¿Es el C++ un sistema OOP?

No. Como no me gustan las medias tintas ni andarme por las ramas, de este modo zanjamos la cuestión de una vez por todas.

Evidentemente la respuesta no puede ser tan simple, ni la cuestión tan tajante, pero al menos, ya sabemos a dónde pretendo llegar.

La OOP fue concebida para ser interpretada, es decir para ser manejada por un intérprete, y el C++ es un compilador. Por lo tanto hay ciertos aspectos de la OOP de los puristas, que no pueden ser abordados de ninguna de las maneras desde C++.

Sin embargo, me parece importante en este punto destacar la encomiable labor realizada por el señor **Bjarne Stroustrup** para hacer del C un lenguaje OOP. Ante este trabajo no queda más remedio que quitarse el sombrero.

Cuando **Stroustrup** se planteó dotar al C de las ventajas de la OOP, es evidente que al menos una cosa tenía bien clara: no estaba dispuesto a perder ni un ápice de la velocidad del C para obtener una mejor implementación de la OOP.

El C fue concebido como un lenguaje de alto nivel que tenía que ser casi tan rápido y tan potente como el ensamblador, y esto es lo que lo ha hecho tan popular. Si eso se perdía posiblemente su trabajo hubiera quedado en agua de borrajas, y él lo sabía bien.

Es muy difícil hacer que un sistema que se ha diseñado para ser interpretado pueda funcionar en modo compilado. Hay que salvar no pocos obstáculos y de gran dificultad. Pero **Stroustrup** no sólo los ha salvado casi todos, sino que además lo ha hecho de forma sumamente eficiente. Tanto, que al día de hoy, cuando se piensa en OOP, se toma (a veces equivocadamente) el C++ como punto de referencia.

Bajo mi punto de vista, la OOP tiene que ser interpretada, por varias razones, y una de ellas es que para mí, una clase es algo que puede cambiar dinámicamente en tiempo de ejecución, y esto es algo absolutamente imposible en C++. Y cuando digo que una clase pueda cambiar dinámicamente me refiero a la propia estructura interna tanto de datos como de métodos de la clase.

En este sentido, el lenguaje C++ no proporciona en sí mismo los mecanismos para operar con las clases de esta forma. Lo que no quiere decir que no puedan almacenarse punteros a funciones en datos de una clase y que estos se cambien en tiempo de ejecución, o que un dato sea un array que crezca o disminuya su tamaño según el programa lo va necesitando. Me refiero la propia estructura intrínseca de la clase.

Hasta donde sé, esto no puede hacerse, y aunque se pudiese, seguiría sin serme válido, ya que de lo que hablo es de que el propio lenguaje proporcione estos medios.

De todos modos nunca he entendido, y cada vez entiendo menos que un lenguaje **no** sea interpretado, o pseudo-interpretado, al estilo de los lenguajes *p-code* (a este respecto, no olvidemos que las últimas versiones del C++ de Microsoft disponen de la posibilidad de generar bien código nativo o *p-code*).

Para mí, interpretar, en lugar de compilar, tiene la gran ventaja de la flexibilidad, y no quiero renunciar a ella por ir cinco picosegundos más rápido que el resto de los mortales.

Mi C++, tu C++, nuestro C++

De todos es sabido que recientemente han proliferado versiones distintas de C++, de tal modo que en cada vez más casos, el mismo código C++ no puede ser compilado por dos implementaciones diferentes de C++.

Esto es cuando menos una aberración. Puesto que el C++ es muy reciente, debiéramos ya haber aprendido la lección de la compatibilidad. Sin embargo cada casa productora de un compilador de este lenguaje, implementa ciertas funcionalidades, o ampliaciones al lenguaje con la intención, no se sabe bien de si hacerlo más potente y operativo, o bien de seguir construyendo la torre de *Babel*.

A todo lo largo de esta conferencia nos referiremos al C++ más estándar que existe, el que quedó recientemente definido y establecido por la comisión ANSI.

Puesto que no he podido hacerme con este documento, pido disculpas si en algún caso asumo que algo de lo que aquí se diga el ANSI C++ sin serlo.

¿Qué debo escribir en C, qué en C++ y qué en lenguaje de alto nivel?

No pretendo, líbrenme los hados de cometer tal confusión, convencer a nadie de que abandone su lenguaje de trabajo, ya sea Clipper, VO u otros, y que se ponga a codificarlo todo en C++.

Pero puesto que, en mayor o menor medida, todos recurrimos alguna vez al C para codificar algo; ya sea porque la velocidad de cierto proceso es crucial, ya sea porque deseamos escribir

una extensión -entiéndase este término en su sentido amplio- del lenguaje, ¿Por qué no hacerlo en C++ en lugar de en C?

Opino que, para escribir aplicaciones de gestión, no hay que abandonar un lenguaje de alto nivel con el que nos sentimos cómodos para ponerse a codificarlo todo en C, por mucho *más más* que ese C tenga; y aquel que lo haga, tiene la mayor de mis admiraciones y mis más profundas condolencias.

Bien es cierto, que cada vez más los lenguajes de gestión no sólo se remiten a abrir ficheros, leer datos del teclado y almacenarlos en los ficheros, para, de vez en cuando, hacer un listado. Hoy en día, cualquier proyecto de gestión, en cuanto que es un poco ambicioso requiere bien del manejo de tarjetas de adquisición de datos, bien del control de procesos de producción, control de detectores, complejos cálculos aritméticos, etc.

Lo que nos lleva irremisiblemente al C. Mi sugerencia es que, en lugar de codificar estos subsistemas en C, lo hagamos en C++, con lo que podremos aprovecharnos de toda la potencia que nos suministra la OOP.

Esto tampoco quiere decir que para codificar un conjunto de fórmulas matemáticas haya que crear una clase, no seamos más papistas que el Papa.

¿Qué pierdo y qué gano con escribir mi código en C++ en lugar de C?

He oído en más de una ocasión que no es abordable escribir código en C++ porque consume más memoria que el C, ya que lo objetos residen en memoria convencional, y no en memoria virtual.

No veo ninguna diferencia, desde el punto de vista del consumo de memoria, entre tener una docena de variables globales, o estas mismas variables agrupadas como una clase. Ambas están consumiendo la misma cantidad de memoria. Sin embargo, si las tengo como una clase tengo todas las ventajas de la OOP al alcance del operador de envío.

Con los métodos ocurre lo mismo, el código es tan máquina en un caso como en el otro, y en ambos, según el enlazador que utilicemos si estamos trabajando con Clipper, y por narices si es bajo Windows, puede ser *overlayado*.

Por otro lado, del mismo modo a como creamos variables locales a funciones, podemos crear objetos locales, que son destruidos automáticamente al finalizar el ámbito de la función, y la memoria liberada.

Bien es cierto que en algunos casos, es necesario reservar porciones más o menos grandes de memoria. Pongamos por caso un objeto que tiene que almacenar y manipular en memoria una zona de vídeo. En este caso, y aunque se estuviese trabajando en modo texto (si es gráfico mejor ni pensarlo), las cantidades de memoria que hay que reservar pueden llegar a ser considerables.

Esto tampoco es óbice para utilizar C++. En C, no se nos ocurre mantener variables de gran tamaño de forma permanente en memoria, lo que hacemos es reservar memoria en tiempo de ejecución. Guardamos un puntero en una variable para poder liberarla cuando ya no la necesitemos.

En C++ podemos hacer exactamente lo mismo, sólo que en lugar de guardar toda la memoria necesaria en un dato del objeto, lo que guardamos es el puntero. Con la ventaja añadida que no se nos va a olvidar nunca liberar esa memoria, ya que el objeto se encarga de ello al ser destruido. Aún más, esta memoria no hay ningún impedimento para que no sea virtual si se desea.

Por último, y por si acaso queda alguien por convencer, dos cosas más, por un lado, los objetos de C++ pueden crearse, accederse a ellos, y destruirse en tiempo de ejecución sin el más mínimo inconveniente. Y por el otro lado, los objetos, en tanto en cuanto no son más que un tipo de dato, tienen el mismo alcance (*scope*) que el resto de las variables del C. Es decir, un objeto creado dentro de una función, es destruido automáticamente cuando termina la función, y su destructor es llamado si estuviese redefinido o aquel por defecto que todos los objetos de C++ tienen.

Antes de terminar con esto quisiera hacer una puntualización para aquellos más puntillosos. Según parece, es cierto que el C++ requiere un poco más de consumo de memoria debido a ciertas estructuras que monta internamente para el manejo de las clases. Pero desde luego, creo que toda persona razonable está dispuesta a perder unos pocos *bytes* en el peor de los casos con tal de programar de una forma mucho más cómoda y eficaz.

Por otro lado están las ventajas que esto supone. A los programadores de Clipper, que aún utilizamos bastante código C en nuestras aplicaciones, no creo que haya que convencerles, sin embargo, los programadores de VO puede que no vean demasiadas ventajas en utilizar C++, ya que VO produce código nativo e incorpora en el lenguaje una excelente implementación OOP así como un buen montón de recursos para la manipulación de datos a bajo nivel.

A ellos les diría que piensen que al trabajar en Windows, todo su trabajo puede ser utilizado por todos los lenguajes disponibles para Windows vía DLLs. Si su trabajo lo hacen en C++, y a él le añaden una capa C puro de acceso a las clases, y lo guardan en una DLL, este trabajo puede ser útil para cualquier lenguaje que trabaje bajo Windows, desde VO hasta Visual Basic pasando por el propio C++. Tengan en cuenta, que al fin y al cabo, todos los lenguajes de Windows acceden a código máquina contenido en DLLs.

Las mejoras del C++ sobre el C

En este apartado nos limitaremos a comentar aquellas ventajas del C++ que más frecuentemente se utilizan, sin tan siquiera enumerarlas todas. No pretendemos hacer un estudio exhaustivo del C++, sino proporcionar aquellos recursos propios del C++ que posteriormente vamos a utilizar a lo largo de esta exposición.

Antes de comenzar con estas peculiaridades, quisiera aclarar que la finalidad de muchas de ellas es o bien hacer la programación más cómoda, o bien hacerla más legible o ambas a la vez. De todos es bien sabido que los programadores de C siempre nos hemos caracterizado por ser terriblemente vagos a la hora de escribir código, quizás porque en C hay que escribir gran cantidad del mismo.

Valores por defecto de los parámetros

En la declaración formal de una función, y al declarar los parámetros de la misma, se le pueden dar valores por defecto a estos parámetros, en caso de que estos parámetros no sean especificados, el compilador tomará los valores por defecto. Por ejemplo:

```
void Display(int i = 5, ;
             float f = 3.2 )
```

Posibles llamadas:

```
// Se asume i = 5 y f = 3.2
Display();
```

```
// Se asume         f = 3.2
Display( 7 );

// No se asume ningun valor
Display( 7, 0.7 )  ;
```

Obsérvese:

- Que los argumentos por defecto no tiene por qué ser del mismo tipo.
- Que pueden omitirse todos los argumentos si todos tienen un valor por defecto.

Sin embargo, una vez que se omite un argumento, tienen que omitirse todos los que le siguen, es decir, la especificación se realiza de izquierda a derecha. Lo cual es totalmente lógico si se tiene en cuenta que en C los tipos de datos son fijos y que los argumentos se pasan a las funciones a través de la pila.

Por ejemplo, la siguiente llamada a función es errónea (obsérvese la coma antes del parámetro):

```
// Error, falta el 1er argumento
Display( , 8.0 );
```

Esto, que a los programadores de Clipper nos parece una chorrada, supone una gran mejora en el C++, no tanto en cuanto a las posibilidades que abre, sino mas bien gracias a la comodidad que reporta a la hora de programar.

Creación y ámbito de las variables

En cada bloque (como es bien sabido por todos, definido por los operadores **{ }**) pueden definirse nuevas variables, las cuales son creadas al iniciarse el bloque, y destruidas al finalizar éste. Con ello no nos referimos solo a cada función (como ocurre tradicionalmente en C), sino también a cada estructura dentro de cada función. Por ejemplo, el código del fuente 1, que es una aberración en C, es perfectamente legítimo y válido en C++:

```
// Fuente 1

void MyFunc( int iNumber )
{
   if( iNumber > 0 )
   {
      int iSum = 0;
      for( int iCount = 0; iCount <= uNumber; iCount++ )
      {
         iSum += iCount;
      }
      printf( "%d", iSum );
   }
}
```

Mientras que la variable *iNumber*, por ser un parámetro de la función tiene ámbito para toda la función, la variable *iSum* tiene existencia sólo dentro del bloque abarcado por la condición *if***,** y la varible *iCount* dentro del bucle *for*.

Esto, por supuesto, no quiere decir que no puedan declararse todas las variables al comienzo de la función, al tradicional estilo del más puro C, pero si se desea, puede hacerse de este otro modo; lo que facilita la lectura del código, especialmente en funciones largas en las que la declaración de una variable, puede encontrarse lejos del lugar donde se utiliza.

Funciones *Inline*

Esta palabra reservada, es en C++ un cualificador de funciones. Del mismo modo a como los cualificadores de C *interrupt* o *static*, hacen que las funciones a las que anteceden tengan ciertas características especiales, el cualificador *Inline* hace que las funciones precedidas por la misma sean copiadas en lugar de ser llamadas en cada lugar del código donde son referenciadas.

Dicho de este modo, pudiera parecer que no hay ninguna diferencia entre un *Inline* y un #*define*, y sin embargo hay una diferencia tan fundamental como útil: La ventaja sobre un *#define* estriba en que el compilador puede hacer chequeo de tipo de los parámetros de este tipo de funciones.

Puesto que en cada referencia que se haga en nuestro código a estas funciones, su nombre va a ser sustituido por el contenido de la función, es muy importante que sólo utilicemos funciones *Inline* allí donde la velocidad de ejecución sea importante, y por el mismo motivo, es conveniente que sean lo más pequeñas posibles sino queremos engordar excesivamente nuestro ejecutable. Por ejemplo, véase el fuente 2:

```
// Fuente 2

inline int max( int a, int b )
{
   if( a > b )
   {
      return a;
   }
   return b;
}
```

Por otro lado, quisiera aclarar que este cualificador de funciones suele utilizarse para crear métodos de consulta de los datos de una clase. Como es bien sabido por todos, la OOP exige la encapsulación de datos y métodos de una clase, y para mantener la integridad del sistema, es necesario que a los datos de una clase se acceda solamente a través de métodos (en C++ a los métodos se les suele llamar *funciones miembro*) de la misma. Esto puede hacerse sin perder velocidad alguna gracias a las funciones *Inline.*

Nota: Los compiladores de C++, suelen permitir definir funciones *inline* sin la necesidad de especificar esta palabra reservada. Basta con situar el cuerpo de la función tras la declaración de la misma dentro de la declaración de la clase. Por ejemplo, véase el fuente 3:

```
// Fuente 3

class MyClass
      {
         public:
      MyClass( int iLength );
      int GetLen() { return iLength; }// funcion miembro inline
      };
```

Si se quiere utilizar este tipo de declaración de funciones *Inline* e incluir más de una línea, basta con separar estas por el separador de instrucciones del C, como es sabido, **;**. Por ejemplo:

```
int MyFunc() { puts( spName ); return 23; }
```

Obsérvese que la última instrucción dentro del bloque, termina con un punto y coma y que tras finalizar la declaración no se especifica el punto y coma, es decir, utilizamos la misma sintaxis que al codificar el resto de las funciones.

Sobrecarga de funciones

En C++, se permite escribir tantas funciones o métodos como se desee con el mismo nombre, siempre y cuando el número y/o el tipo de argumentos que recibe sea diferente para cada una de ellas. Dos funciones sobrecargadas pueden devolver un tipo distinto de dato cada una de ellas, pero no puede establecerse la sobrecarga en la diferencia entre los tipos de valores devueltos.

Esta es una de las cualidades principales del C++, pero a diferencia de lo que muchos pensamos cuando empezamos a estudiar C++, no se aplica sólo a los métodos de una clase, sino (y puesto que para el C++, los métodos, o funciones miembro son tratados como cualquier otra función) que puede aplicarse a cualquier función.

El compilador, decidirá a qué función se está llamando en cada caso y dará internamente un nombre único para cada una de ellas, de tal forma que las llamadas realizadas en tiempo de ejecución sean las apropiadas para cada caso.

Por ejemplo, en el fuente 4 podemos ver cómo emplearíamos la sobrecarga de funciones para elevar al cuadrado números enteros, long y float.

```
// Fuente 4

// Datos int
int Cuadrado( int iValue )
{
   return (iValue * iValue );
}

// Datos long
long Cuadrado( long lValue )
{
   return (lValue * lValue );
}

// Datos float
float Cuadrado( float fValue )
{
   return (fValue * fValue );
}
```

La sobrecarga, que en sí misma es una herramienta muy útil para poder elaborar un código más claro, puede convertirse en un arma de doble filo si abusamos de ella o la utilizamos incorrectamente. De este modo, no debemos dar nunca el mismo nombre a dos funciones que realizan tareas completamente distintas, salvo que sean métodos de clases donde quede bien claro lo que, en el contexto de la clase, significan.

Por ejemplo, si creamos dos funciones llamadas *Convertir()*, una de ellas convierte números decimales a binarios y otra convierte documentos de MS-Word a WordPerfect, **no** estaremos utilizando correctamente la sobrecarga, ya que esto nos puede producir más confusión que otra cosa.

Enlazado de funciones C y C++

Como ya hemos comentado anteriormente un poco de pasada, los compiladores de C++, cambian internamente los nombres de todas nuestras funciones, ya sean miembros de clases o no. Esto permite tanto la *sobrecarga* como el *polimorfismo*, necesarios en OOP. El problema que esto conlleva, aparece cuando hacemos referencia a una función externa compilada al estilo C puro o bien cuando deseamos mezclar funciones C y C++ en el mismo fichero. Ya que este nombre no puede ser cambiado internamente, porque ha sido compilado bajo las normas C y no las C++, es necesario informar de este hecho al compilador de C++ para que las referencias se puedan resolver adecuadamente.

Para poder acceder a las funciones escritas en C desde un código C++ es necesario identificarselas al compilador. Para ello se utiliza el especificador: *extern "C"* Por ejemplo:

```
extern "C"
{
#include "extend.h"
}
```

Esta declaración le dice al compilador de C++ que todo aquello que se encuentra entre las llaves ha sido compilado con un compilador de C. En caso de no utilizar las llaves, la especificación se aplica sólo a la instrucción que sigue a la declaración.

De esta forma, el compilador de C++ no generara nombres especiales para nuestras funciones C y el enlazador podrá resolver las referencias adecuadamente. Esto es lo que se suele hacer cuando utilizamos librerías de terceros.

Nota: Cuando se está llamando a funciones estándares de la librería de C, no es necesario incluir esta especificación.

Notación húngara en C++

En C++, a este respecto hay varias tendencias, pero quizás la más utilizada es componer el nombre de la clase con tres letras minúsculas y utilizarlo como sufijo en todos los datos que de esa clase se vayan creando.

Esta es la norma que yo personalmente utilizo, no quiero con ello ni imponerla ni decir que sea mejor que otras, pero es la que se ha utilizado en los ejemplos que acompañan a esta exposición.

Algunos autores, cuando crean objetos dinámicamente, anteponen el prefijo *p* para indicar que es un puntero al objeto más que el objeto en sí mismo. A mí personalmente este aspecto no me gusta demasiado porque puede confundir el nombre de la clase, y por lo tanto no lo utilizo. También es frecuente que los nombres de las clases comiencen con la letra *T* como prefijo. Esta es una costumbre que tomé desde que comencé a programar en OOP, y que últimamente estoy abandonando, por lo que se verá en algunos fragmentos de código y no en otros. La razón de que no la utilice demasiado últimamente es bien simple: pienso que si en OOP creamos nuevos

tipos de datos, y estos deben funcionar exactamente como aquello implementados por el lenguaje (al menos esa es la meta a la que se tiende), no hay ninguna necesidad de distinguirlos. Veamos algunos ejemplos:

- Objeto de la clase **Window**: *Window wndMain*;
- Objeto de la clase **Connect**: *Connect cntServer*;
- Objeto de la clase **Report**: *Report rptFactura*;