NOTACION HUNGARA

Artículo escrito por
Antonio Linares

1.- ESTABLEZCAMOS UNAS REGLAS: EL SISTEMA HUNGARO.

Aunque todos programemos usando el mismo lenguaje, me daréis la razón en que hay grandes diferencias en el estilo de escribir código de uno a otro programador. Y esto, desde luego, no es ninguna ventaja. De hecho es un gran problema.

A través de los artículos de esta revista, tendremos ocasión de revisar y estudiar el código de muchas funciones, utilidades, etc... Sería una gran ventaja, que pudiésemos tomar literalmente el código de una rutina que nos interesase y directamente importarla a las líneas de nuestro programa.

En muchas ocasiones, me ha ocurrido que alguien me ha traído unas rutinas y me ha pedido que les echase un vistazo, y muchas veces no lo he hecho porque era francamente difícil entender la organización-estructura-ideas de esas líneas de código. El problema es aún mayor si se trata de un programa profesional. ¿Cuantas veces os ha ocurrido que habéis retomado el trabajo de algún otro programador, y os ha costado un esfuerzo increíble entender cada línea de código? Muchas veces.

No se si sabréis quien es Charles Simonyi. Es un programador húngaro que ha trabajado (no se si aún está) en Microsoft. El inventó un sistema de escribir código para los programas, y debido a su nacionalidad, sus compañeros en plan de broma le llamaban el método Húngaro.

Hoy en día es el sistema ESTANDAR que utiliza el 'Microsoft Aplications Development Group'. Es decir, ES OBLIGATORIO para los programadores que trabajan en Microsoft. El que sus compañeros le llamasen el sistema húngaro, se debía a que a simple vista parecía que usaba 'palabras húngaras', por lo raro que se veían.

Pero Simonyi, es mucho Simonyi. El método Húngaro es realmente potente y creo que nos vendría a todos muy bien utilizarlo. De entrada, la primera ventaja que proporciona es que permite a los demás entender perfectamente lo que escribimos (o por lo menos lo facilita bastante).

Este sistema hemos empezado todos a verlo con el Clipper 5.0, cuando

delante de una variable encontramos una letra minúscula (c,n,l,d,a,...) que indica de que tipo es la variable en uso.

Pero el método húngaro es mucho más potente. Por eso os propongo que estudiéis las ideas que os apunto y que empecemos a usarlas todos. Yo, ya melo impuse cómo obligatorio. En poco tiempo se notan sus grandes beneficios...

1.- Prohibido TERMINANTEMENTE el uso de 'underscores', o sea, de E_S_T_O. Sólo pueden usarse en los defines pero NUNCA en el nombre de una variable o función. En vez de los under_scores hemos de usar NoUnderScores. Las partes de una palabra han de separarse con mayúsculas y minúsculas, NUNCA con UnderScores. OK???

2.- La primera letra de una variable debe indicar el tipo de la misma: Tipo esta representado por una letra que indica el tipo de variable.

c, n, l, d, o, a, etc...

cCadena, nNumero, lLogica, dFecha, oObjeto, aArray, etc...

3.- Los defines deben empezar por un prefijo (tres letras preferentemente) que indiquen que tipo de define es, o a que elemento se refieren. Sí se pueden usar UnderScores. También HEMOS DE INDICAR el tipo del define. Esto se hace así:

```
DBF_cNOMBRE
DBF_nCAMPOS
KEY_nENTER
BOX_lOPENED      ¿¿¿ Estupendo, no ???
```

4.- Las funciones del SISTEMA, es decir, aquellas que trae el lenguaje que usemos de SERIE, se escriben SIEMPRE en minúsculas. Sin usar NINGUNA mayúscula.

date(), time(), alltrim(), used(), etc...

5.- Las funciones que NOSOTROS HAYAMOS CREADO comienzan por MAYUSCULAS y pueden usar mayúsculas para separar una parte de otra.

Aread(), Awrite(), BoxMsg(), etc...

Veis que demasié? Si alguno de vosotros publica código, sabremos que funciones son de Clipper y cuales son nuestras. Que fácil !!! Si alguien lee código de otro sabrá rápidamente cuáles son creación propia del programador...

6.- Las funciones de conversión, es decir, aquellas que pasen de un valor a otro, usan el '2' cómo separador (se pronuncia two-'tu'-'to' -> 'a' en Español).

```
Str2Arr()
Hex2Dec()
Clr2Atr()
```

7.- Las palabras PROCEDURE, FUNCTION y RETURN deben ir SIEMPRE en MAYUSCULAS. Y deben ir al mismo nivel de indentación !!!

8.- Con los métodos de la programación orientada al objeto hay una excepción. La primera letra del mensaje debe ser en MINUSCULAS y las demás partes del nombre del mensaje pueden ir en Mayúsculas...

```
oBrowse:addColumn(oCol)
```

9.- OBLIGATORIO: Todos los comandos van en minúsculas !!! La razón es muy potente. Los nombres de los campos deben ir en MAYUSCULAS así al ver una línea de código es muy fácil saber sobre qué campos actúa un determinado comando. Los nombres de las DBFs y de los CAMPOS van siempre en MAYUSCULAS. Los alias también van en MAYUSCULAS.

10.- OBLIGATORIO: Dejar un espacio entre los paréntesis y su contenido. Os propongo a todos que lo practiquéis. Veréis que os gusta mucho, y que os va a gustar mucho más el que podáis entender más fácilmente el código de los demás. Me gustaría que lo probaseis y que deis vuestras opiniones al respecto. ¿OK? Se os ocurren más reglas que puedan mejorar la comprensión del código? Venga, seguro que sí! ¿Que os parece si le ponemos prefijo a las funciones, también, para saber que es lo que devuelven?

Os adjunto algunas funciones escritas así para que veáis la gran diferencia que supone el usar estas técnicas.

2.- LOS ARRAYS MULTIDIMENSIONALES EN PROFUNDIDAD.

Los Arrays Multidimensionales son la ESTRUCTURA DE DATOS más importante de Clipper 5.

Todos sabéis lo que es un Array. Los habéis utilizado en Summer 87. ¡ Ahora, en 5, proporcionan mil veces más potencia ! La primera pega que la gente me ha comentado acerca de esta nueva forma de arrays es entender lo de 'multidimensional'...

Si os digo: 'un array de una dimensión... ' lo entendéis perfectamente. :-).

Si os digo: 'un array de dos dimensiones' también me podéis seguir...

Si os digo: 'un array de tres dimensiones' más difícil, pero sí, sí sois capaces de entenderlo. (Muchos diréis, 'si, cómo alto, largo y ancho...tres dimensiones'... :-).

Vale. Si ahora digo, un array de 4 dimensiones (empiezan las caras raras :-S). Cómo imaginarse cuatro dimensiones?... Y si digo 'arrays de 100 dimensiones' >:-(Tranquilos, que no pasa nada... que no van por ahí los tiros.

La primera palabra que confunde es lo de 'dimensional'. Esa palabra no tiene que ver nada con lo que entendemos por 'dimensión' en el mundo real. Sencillamente la usamos por tradición, pero no significa nada de eso. No son dimensiones, como nosotros las entendemos.

Hablar de un array multidimensional quiere decir, sencillamente, que alguno de sus elementos es a su vez otro array, y puede seguir así, sucesivamente, mientras tengamos memoria suficiente. Pero olvidaros de lo de las dimensiones...

Y cómo se entiende que un elemento de un array pueda ser a su vez otro array? Bien. Para eso tendréis que armaros de valor :-) y empezar a usarla palabra 'PUNTERO'.

'PUNTERO' quiere decir 'LA DIRECCION DE...'. Si yo os pregunto a cualquiera de vosotros que donde esta vuestra casa, la dirección que me digáis es UN PUNTERO a vuestra CASA. ¿ok? Así, si yo quiero ir a buscaros, uso el PUNTERO que me dice en donde puedo encontraros (vuestra dirección). Me seguís? Venga...

Cuando Clipper monta un ARRAY en memoria, lo pone en algún sitio donde le quepa. Y para no gastar tiempo y energía en moverlo de un lado para otro, lo que nos da es su dirección, es decir, el lugar en donde el array 'vive'. ¿ok?

Cuando decimos que en un array multidimensional algún elemento puede ser a su vez un array, lo que estamos diciendo es que algún elemento contiene un PUNTERO (la dirección en donde hay otro array en memoria) a otro array. Así que todos los arrays estarán tranquilos en donde estén guardados (esto lo hace Clipper automáticamente), y manejaremos sus PUNTEROS (sus direcciones).

Todo esto lo hace Clipper por sí solo. Sin que nos demos cuenta. Nosotros sencillamente creemos que los estamos manipulando directamente, pero en realidad estamos usando sus direcciones (PUNTEROS a los ARRAYS). Más claro ahora ?

```
var1 = { 1, 2, 3 }
```

Clipper sitúa { 1, 2, 3 } en memoria y hace que 'var1' sea un puntero a dicho array. Si ahora creamos otra variable 'var2' y la hacemos igual a 'var1', lo que estamos es asignándole la misma dirección que contiene 'var1'.

```
var1 = { 1, 2, 3 }
var2 = var1
```

Sólo hay un array { 1, 2, 3 } y tanto 'var1' cómo 'var2' apuntan a él. Esto lo comprobamos si hacemos ahora:

```
var1 = NIL        // Estamos borrando la dirección
                  // que contenía

? var2[ 1 ]       // Devuelve 1. El primer elemento
                  // del array
```

Luego el array sigue existiendo aunque variemos 'var1'. ¿ Cuando Clipper elimina totalmente el array de memoria? Cuando no exista ningún PUNTERO a él.

```
var1 = NIL
var2 = NIL        // Ahora Clipper SI destruye { 1, 2, 3 }
```

En realidad Clipper NO TIENE arrays multidimensionales realmente. Lo que hace es que permite que un elemento de un array sea a su vez un PUNTERO a otro array que estará en otra parte de la memoria. Nosotros, como usuarios, no notamos nada de esto, el efecto que a nosotros nos produce es que sí existen.

Entendiendo esto, veamos ahora cual será el comportamiento de Clipper en la siguiente situación:

```
var1 = { 1, 2, { "A", "B", "C" } }
```

El tercer elemento de 'var1' en realidad es un puntero al array { "A","B", "C" } que estará en otra parte en la memoria. Si ahora hacemos,

```
var2 = var1[ 3 ]
var1 = NIL
? var2[ 1 ]          // Devolver  "A"
```

El array al que apuntaba el tercer elemento de 'var1', sigue existiendo y su dirección la tiene 'var2'.

Más adelante volveremos con todo esto otra vez, porque es fundamental si queremos sacarle todo la potencia a Clipper 5.

También os enseñaré algunas funciones muy interesantes. Algunas de

ellas nos permitirán SALVAR un array multidimensional a DISCO y otra función que nos permitirá leerlo desde DISCO y restaurarlo en memoria (no importa cuantas veces esté anidado). Mientras tanto, practicad todo lo que podáis, y recordad: ¡¡¡Equivocarse ayuda a APRENDER!!!

3.- LA UNION PERFECTA: ARRAYS MULTIDIMENSIONALES Y EL PREPROCESADOR.

Es fácil usar un array de dos o tres elementos. La mayoría de las veces nos acordamos de memoria de lo que guardamos en cada uno de sus elementos. Sin problema.

Cuando alguno de los elementos de un array es a su vez otro array, la cosa se va complicando. Tendremos que acudir frecuentemente a donde tengamos apuntado qué es qué. Nos equivocaremos fácilmente...

Cuando el array ya llega a dos o tres niveles de profundidad (es decir un elemento que es a su vez un array, en el que algún elemento es a su vez un array, y así sucesivamente...) dudo que muchos de nosotros seamos capaces de manejarlo de memoria sin que cometamos errores constantemente. La cabeza olvida fácilmente. Y cuando haya pasado algún tiempo desde que escribamos el código, peor que peor.

Pero, tranquilos. Tenemos el aliado perfecto: EL PREPROCESADOR. El preprocesador nos permitirá hacer verdaderas virguerías con los arrays multidimensionales y os aseguro que con la técnica que voy a explicaros NO COMETEREIS NI UN ERROR !!!

No penséis que es tan fácil cómo creéis. De hecho, yo pasé mucho tiempo probando y probando, hasta que di con el truco...

Existen unas reglas de ORO que tenéis que usar para que manipular ARRAYS MULTIDIMENSIONALES, AL NIVEL QUE SEA, sea un juego de niños. También vamos a ver COMO LEERLOS. No se trata sólo de escribirlos, se trata de entender luego lo que pongamos !!!

El preprocesador tiene la sentencia #define que nos permite sustituir una expresión por otra ANTES de que el compilador haga su trabajo. Imaginad que tenemos un array de 3 elementos:

```
aArray = { "A", "B", "C" }
```

Ahora supongamos que para evitarnos recordar lo que significa cada elemento usamos los siguientes #defines:

```
#define PRIMERO 1
#define SEGUNDO 2
#define TERCERO 3
```

Podemos sustituir los índices a usar por esos 'defines' con lo que todo será mucho más fácil:

```
? aArray[ PRIMERO ]
? aArray[ SEGUNDO ]
? aArray[ TERCERO ]
```

Vale. Pero... supongamos ahora que en el primer elemento de este array queremos crear otro array de tres elementos en el que vamos a guardar tres expresiones de color (en el formato corriente):

```
aArray[ PRIMERO ] = array( 3 )
```

Podemos pensar en volver a usar esos mismos defines, pero veréis que empezamos a liarnos:

```
aArray[ PRIMERO ][ PRIMERO ] = "B/W"
aArray[ PRIMERO ][ SEGUNDO ] = "GR+/B"
aArray[ PRIMERO ][ TERCERO ] = "BG+/R"
```

En vez de eso, probemos lo siguiente. Cómo sabemos que queremos guardar en el primer elemento un array de colores, en vez de usar #define PRIMERO 1, usaremos

```
#define aCOLORES 1
```

Ahora, crearemos otros defines para manipular los elementos de este array:

```
#define CLR_cNORMAL  1
#define CLR_cINVERSO 2
#define CLR_cBORDE   3
```

Fijaros en cómo queda ahora:

```
aArray[ aCOLORES ][ CLR_cNORMAL ] = "B/W"
aArray[ aCOLORES ][ CLR_cINVERSO ] = "GR+/B"
aArray[ aCOLORES ][ CLR_cBORDE ]  = "BG+/R"
```

Más difícil todavía. Supongamos que ahora queremos que el elemento aArray[aCOLORES][CLR_cBORDE] sea a su vez otro array que guarde los distintos tipos de color para bordes que vayamos a utilizar:

```
aArray[ aCOLORES ][ CLR_cNORMAL ] = "B/W"
aArray[ aCOLORES ][ CLR_cINVERSO ] = "GR+/B"
aArray[ aCOLORES ][ CLR_aBORDE ]  = "BG+/R"

aArray[ aCOLORES ][ CLR_aBORDE ][ BRD_cUNO ] = "BG+/R"
aArray[ aCOLORES ][ CLR_aBORDE ][ BRD_cDOS ] = "RB+/B"
aArray[ aCOLORES ][ CLR_aBORDE ][ BRD_cTRES ] = "RB+/B"
```

¿Os dais cuenta que es fácil? Ya nos estamos manejando a tres niveles y nos es difícil... podemos seguir así sin problema.

Os resumo las reglas a seguir:

- El nombre del DEFINE consta de un PREFIJO y de un SUFIJO.

- El PREFIJO constará de TRES LETRAS. Esas tres letras representan el nombre de lo que estemos usando. Si, por ejemplo, vamos a crear unos DEFINES para guardar los valores de determinadas pulsaciones, usaremos KEY_ de 'tecla' en Inglés, o si preferís TCL_ de 'tecla'. Por lo general se usan las CONSONANTES del nombre a escribir, suprimiendo las VOCALES TeCLa. ¿ OK ?

- Entre PREFIJO y SUFIJO situaremos un UNDERSCORE '_' (mucha palabra para tan poco :-) ¿no?). En Español: Un 'SUBRAYADO' o 'GUION ABAJO'.

- A continuación, y en minúsculas, escribiremos (siguiendo el METODO

HUNGARO) una letra en MINUSCULAS que represente el tipo que tiene el valor que vamos a substituir. Ojo, con esto. Tenemos DOS alternativas, o poner el tipo del elemento a SUSTITUIR (aCOLORES, Un array) o poner el tipo del elemento sustituido (nCOLORES, pues nCOLORES es 3). Según la situación usaremos una forma u otra. Yo prefiero la PRIMERA. Es mil veces más intuitiva, pero... lo mejor, SIEMPRE, será lo más práctico.

- A continuación va el SUFIJO. El SUFIJO representa la CARACTERISTICA o el DATO del PREFIJO que estemos utilizando. Así, en CLR_cNORMAL, NORMAL es de tipo 'c' y es el COLOR (CLR) 'NORMAL'.

- **IMPORTANTISIMO**: SUFIJO y PREFIJO en ELEMENTOS CONTIGUOS del array deben COINCIDIR (o será prácticamente la MISMA PALABRA, aunque abreviada). Esta es LA FORMA DE COMPROBAR QUE NO NOS HEMOS EQUIVOCADO AL USAR EL ARRAY MULTIDIMENSIONAL.

- Si un SUFIJO termina en S (una 'ese'), quiere decir que hay más de un elemento de ese tipo (por estar en plural) y entonces, el siguiente elemento debe indicar CUAL de los elementos es:

```
aArray[ aPANTALLA ][ PNT_aCOLORES ][ CLR_aNORMAL ][ NRM_cTINTA ]
```

Fijaros (es importante) que para saber si esta expresión está bien escrita o no, lo que hacemos es comparar los PREFIJOS CON LOS SUFIJOS anteriores:

```
aPANTALLA ... PNT   ( PaNTalla )
aCOLORES.....CLR   ( CoLoRes )
aNORMAL.....NRM   ( NoRMal )
etc...
```

Si existe esta correlación entre SUFIJOS y PREFIJOS, entonces es que está bien escrito. SIN ERRORES !!!

- Cómo se lee un array multidimensional escrito con estas normas:

El del ejemplo anterior:

```
aArray[ aPANTALLA ][ PNT_aCOLORES ][ CLR_aNORMAL ][ NRM_cTINTA ]
```

La TINTA del COLOR NORMAL de los COLORES de la PANTALLA. Más o menos, lo que se hace es leerlo de DERECHA a IZQUIERDA. Contra más practiquéis, más fácil se os hará. Si SUFIJOS y PREFIJOS coinciden, entonces está bien escrito.

Recordad que si un SUFIJO termina en 'S' es que representa a un elemento que consta de varios, así, en un bucle prestad atención:

```
FOR nCual := 1 TO 10

  aColor := aArray[ aPANTALLA ][ PNT_aCOLORES ][ nCual ]
  ...
NEXT
```

La clave: Práctica... mucha práctica. :-) . Espero que os haya gustado. Ya veréis cómo los usáis con toda confianza.

4.- BLOQUES DE CODIGO: QUE SON, COMO SE CONSTRUYEN, PARA QUE SIRVEN.

He leído bastantes definiciones de 'bloque de código'. Ninguna me ha gustado. :-). De por sí, el hablar de 'bloque de código' resulta lioso, pero si encima alguien viene y nos dice que es 'un nuevo tipo de dato', peor que peor... :-)

¿Que hay de nuevo y que hay de viejo en los 'bloques de código'? Antes que nada deciros que los 'bloques de código' NO son un 'invento' de Nantucket. Los 'bloques de código' existen desde hace mucho tiempo en otros lenguajes... lo que pasa es que nadie les llamaba así.

Hemos de armarnos de valor y volver a usar la palabra 'PUNTERO'. Hemos visto que un 'PUNTERO' es la dirección en memoria en donde hay algo determinado. Es la 'dirección' en donde ese 'algo' está.

Al aprender a programar en C, se suelen pasar malos ratos intentando 'hacerse' con los PUNTEROS. Pero entenderlos a fondo es muy importante. Fijaros si son importantes, que ahora Nantucket nos los pone cada vez más en Clipper.

Hemos visto que cuando usamos un ARRAY, Clipper lo sitúa en algún lugar de la memoria del ordenador y que para no estar moviéndolos de un lado a otro (pues pueden ser muy grandes y además resultaría muy lento), nos proporciona un PUNTERO (la dirección) a su posición en memoria.

Del mismo modo que una variable (ó un ARRAY) está situada en algún lugar de la memoria de nuestro ordenador, cuando el enlazador termina de construir el EXE, ha situado en determinadas partes del EXE a los diferentes 'trozos' de nuestro programa ('bloques de código').

Un bloque de código lo que hace es situar en algún lugar de la memoria una serie de instrucciones a ejecutarse más tarde, y lo que hace es DEVOLVER UN PUNTERO al lugar de la memoria en donde ha situado ese conjunto de instrucciones.

Cuando hacemos un EVAL(bVARIABLE), lo que hacemos es decirle al programa que 'salte' a esa dirección y que ejecute lo que hay allí. ¿ Lo entendéis mejor ahora? Espero que sí...

Esos PUNTEROS se guardan en variables que pasan a ser de tipo 'b' (bloque de código). En realidad esas variables 'valen' un número que indica una dirección en memoria. Estos valores son bastante delicados, por lo que Nantucket no nos deja que los manipulemos alegremente pues provocaríamos el CUELGUE DEL SISTEMA si le dijésemos a nuestro programa que 'saltase' a una parte de la memoria en donde no hubiese código sino cualquier otra cosa.

Esta es la razón por la que no podemos salvar el valor de un bloque de código a disco ni a un archivo. Estas direcciones variarán de un programa a otro, de un estado del sistema a otro.

Luego, ¿que es un bloque de código? Es un PUNTERO a un lugar de la memoria en donde se encuentran una serie de INSTRUCCIONES listas (ya están compiladas) para funcionar. ¿ok?

En C, por ejemplo, un 'bloque de código' se representa por el NOMBRE de la FUNCION sin usar los PARENTESIS. O sea, que si tenemos la función (en C) PRINTF(), el nombre 'PRINTF' es el PUNTERO a PRINTF(). Cuando le digamos al programa que salte a donde apunta 'PRINTF', saltaremos a donde está PRINTF(). Es cómo si hiciésemos en Clipper EVAL(PRINTF). Espero no liaros :-). ¡Seguro que no !

Bueno, vamos a lo siguiente.

¿Cómo se escribe un bloque de código?

Un bloque de código es una 'FUNCION SIN NOMBRE'. Lo vais a ver en este ejemplo:

```
FUNCTION PRUEBA( nValor1, nValor2 )

 tone( nValor1, nValor2 )

RETURN "Ya está"
```

Fijaros ahora en cómo escribiríamos esa misma función, pero creándola cómo un bloque de código:

```
{ | nValor1, nValor2 | tone( nValor1, nValor2 ), "Ya está" }
```

Es exactamente igual que la función que escribimos antes. La función PRUEBA(). Lo que ocurre es que ahora el nombre 'PRUEBA' no aparece por ninguna parte. ¿Comprendéis porque se les dice a los 'bloques de código' que son funciones sin nombre?

Los nombres que aparecen entre las barras verticales son los parámetros de esa función. Al igual que en una función tradicional, esos nombres pueden ser lo que nosotros queramos, en general usaremos los nombres que más nos recuerden lo que estamos haciendo. Mucha gente cuando ve bloques de código anda como loca buscando por el resto del programa los nombres que ahí aparecen entre barras verticales. Y pueden no estar en ninguna otra parte. Es exactamente igual que hacemos al definir los nombres de los parámetros a una función ¿ok?

Las líneas que tenía la función PRUEBA(), ahora se escriben separadas por comas. Así de fácil. Y... la última expresión que aparece en el bloque de código es lo que devolverá el bloque de código al ejecutarse, ¡¡¡ IGUAL QUE EL RETURN DE UNA FUNCION !!! ("Ya está" en nuestro ejemplo).

No pretendo con estás líneas que seáis unos maestros de los bloques de código. Pero leéroslas varias veces, consultad el manual y practicad. Es bien fácil (cuando se entiende :-)).

¿ Y para que sirve un bloque de código ?

Cómo os comenté antes, los programadores en C hace mucho tiempo que tienen 'bloques de código' y os diré que muchos de los programadores en C que conozco, NO LOS USAN para nada ni tienen mucha idea de para que sirven. Esa es una de las cosas que más me gusta de Clipper, que es la mejor 'academia' de programación que he conocido :-). Curiosamente... la

programación orientada al objeto tiene su fundamento en los punteros a funciones en casi un 90 % de su diseño...

Os voy a decir algo 'muy fuerte'. Agarraros...

Igual que los PARAMETROS de una FUNCION modifican el RESULTADO de esa FUNCION, un BLOQUE DE CODIGO modifica la LOGICA de UNA FUNCION. La FUNCION se COMPORTARA de un MODO DISTINTO si en su 'maquinaria' EVAL()UA determinados bloques de código que PUEDEN SER DISTINTOS de una vez para otra. Tiempo al tiempo... Esto no es llegar y topar...

Mucha gente que ya usa los bloques de código en Clipper, los usa más que nada para escribir menos líneas de código. Pero este no es su principal fin. Fundamentalmente sirven para hacer que algo (una función o un objeto) 'razonen' de otra manera. Que modifiquen su lógica. Eh ! Seguid leyendo! No os vayáis !Prometo no volver a decir más cosas así! :-)

Volveremos a verlos. Veremos muchos ejemplos. Es fácil, de verdad. Lo importante es que leáis esto algunas veces y que por lo menos 'os lo sepáis de memoria' :-).

5.- PREPARATE A CAMBIAR TU FORMA DE PENSAR: PROGRAMACION ORIENTADA AL OBJETO.

Tened por seguro que en el futuro más inmediato, nos VAMOS A HINCHAR de usar la PROGRAMACION ORIENTADA AL OBJETO. Dentro de poco, Nantucket sacará una nueva versión de Clipper (no me refiero a la 5.01, sino a la 5.02 o a la 5.05 o a la 6.0), y esa nueva versión va a ser un 'recital' de PROGRAMACION ORIENTADA AL OBJETO.

En serio, el OOPS (Object Oriented Programming System) es TREMENDAMENTE IMPORTANTE, IMPORTANTISIMO, así que cuanto antes empecemos a entenderlo bien, mucho mejor. Vamos a empezar a ver, paso a paso, cada secreto del OOPS. :-)

¿ Porqué aparece el OOPS? ¿De donde viene? ¿Para que sirve?

Supongo que todos los que estáis leyendo esto, sabéis perfectamente lo importante que es la PROGRAMACION ESTRUCTURADA. El lenguaje 'maestro' en programación ESTRUCTURADA es el C, y la estrella fundamental es 'LA FUNCION' (funciones estandar y UDFs). Todo esto lo tiene muy claro Clipper y por eso cada vez nos acercamos al C más y más.

La idea de la programación estructurada es REUTILIZAR NUESTRO ESFUERZO y ganar en SIMPLICIDAD. Cuando pasamos varias horas escribiendo una función, y al final conseguimos que haga lo que queremos, la guardamos en una librería y ya no nos volveremos a preocupar de cómo funciona, la utilizaremos y ya está. (por cierto, os supongo a todos 'maestros' en construir-manejar-estudiar-investigar-desarrollar librerías, sino es así, podemos dedicarle un estudio a fondo al tema, ¡decídmelo!).

En nuestra mente ganamos en SIMPLICIDAD. Ya no tendremos que volver a pensar en el mecanismo de esa FUNCION que hemos escrito. En el futuro, REUTILIZAREMOS NUESTRO ESFUERZO. (a esto se le llama 'reusabilidad del código').

La programación TRADICIONAL pensaba que esto era lo máximo en ESTRUCTURACION. Pero hay muchos genios sueltos por ahí... :-)

Fijaros en este detalle. Una FUNCION suele COMPORTARSE de distinta FORMA según los PARAMETROS (los DATOS) que le PROPORCIONEMOS. Y a la vez, la tarea habitual de una FUNCION suele ser MODIFICAR determinados DATOS... ¿Veis que fuertemente relacionados están los DATOS y las FUNCIONES?

Los estudiosos de la PROGRAMACION, se dieron cuenta de esta íntima relación entre DATOS y FUNCIONES. Y sencillamente se dijeron '¿PORQUE NO DESARROLLAMOS UNA ESTRUCTURA EN LA QUE DATOS Y FUNCIONES ESTEN JUNTOS?' ASI CONSEGUIREMOS UNA MAYOR ESTRUCTURACION EN NUESTROS PROGRAMAS, ES EVIDENTE. Y así, es como nacen los OBJETOS... (tachan !!!), las CLASES, el OOPS...

Luego, el primer OBJETIVO del OOPS es ganar en ESTRUCTURACION. Es decir, hacer que el desarrollo de nuestros programas sea MAS FACIL, que se REUTILICE mucho más nuestro ESFUERZO, y que tengamos que 'mantener' menos cosas en la cabeza. De hecho, el creador del C++ (C con CLASES),

afirmó que un programador que utilice OOPS puede escribir, él solo, un programa de 25.000 líneas de código sin gran esfuerzo !!! Cometer muy *pocos errores* !!! Y , *en todo momento*, mantendrá la ESTRUCTURA del programa en su cabeza... ¿¿¿A que os está interesando la programación orientada al objeto cada vez más??? :-)

Bien, sigamos...

Existen otras razones por las que resulta evidente que podemos ser MEJORESPROGRAMADORES usando OOPS. Veámoslas paso a paso:

Hemos visto que el objetivo que el desarrollo del OOPS se proponía era el AGRUPAR DATOS y FUNCIONES (veremos que a esto se le dice 'ENCAPSULACION'). Pero es que se dieron cuenta que lo que resultaba de agrupar DATOS y FUNCIONES se parecía a algo... ¿A que se parece? :-)

Fijaros en cualquier OBJETO de los que haya en la MESA más cercana a vosotros, ó en esa estantería, da igual... sí, un OBJETO cualquiera. ¿Lo habéis elegido ya? Yo, por ejemplo, tengo ahora aquí a mi lado un VASO de CRISTAL. Lo que os voy a contar de él sirve también (comprobadlo) con ese OBJETO que hayáis elegido.

¿ Que es un VASO DE CRISTAL ? Un VASO DE CRISTAL es 'algo' que tiene una serie de DATOS característicos y que se comporta de unas determinadas formas (METODOS) ante determinadas circunstancias (MENSAJES).

¿Cuales son los DATOS DE UN VASO DE CRISTAL? Pues, es de cristal, es transparente, mide unos 15 cms. de alto, pesará unos ciento y pico gramos, tiene una forma característica, un color determinado (aunque sea transparente puede tener un determinado color, no?), tiene algunos dibujos tallados, tiene una cavidad que es en donde puede contener líquido, etc...

Y ¿QUE SABE HACER UN VASO? Pues varias cosas... Sabe CONTENER EL LIQUIDO si DECIDIMOS LLENARLO. Rueda si lo empujamos, se rompe si lo hacemos caer desde alto, etc...

Fijaros que parecido es un OBJETO de la naturaleza a lo que nosotros nos proponíamos al ESTRUCTURAR CONJUNTAMENTE DATOS Y FUNCIONES. Que curioso...

Pero es que aún hay muchas más coincidencias y curiosidades...

¿Hace cuanto tiempo que sois programadores? ¿4, 5, 8, 10 años? ¿desde hace cuanto tiempo utilizáis al 100% las técnicas de programación estructurada? (No me mintáis, eh!), 3, 4, 5 años, ¿un poco más quizá?... :-)

Daros cuenta de esto. Desde que todos nosotros ERAMOS BEBES (si BEBES, con pañales, cunas, papillas, etc... :-)), hemos estado USANDO OBJETOS. Llevamos practicando TODA NUESTRA VIDA. Todos nosotros SOMOS MAESTROS en el uso y convivencia con los OBJETOS. Hay un MUNDO DE OBJETOS a nuestro alrededor desde SIEMPRE...!!!

De hecho, nuestro sistema nervioso (sí, ese que se altera cuando no aparece un bug :-)) sabe la importancia de todo esto, y nuestra mente,

generación tras generación, ha ido desarrollando extraordinarias capacitaciones en el uso de los objetos, para tener control en el mundo *que nos rodea. (Diréis que me he vuelto chalado, que tiene que ver todo esto con Clipper,... ya veréis que mucho más de lo que podáis pensar ahora...:-)).*

Una de las facultades del ordenador más increíble que existe (no, no me refiero al 486... sino al cerebro humano) es la capacidad de ABSTRACCION. De un mundo en que sólo existe lo PARTICULAR, nuestra mente crea lo GENERAL. Gracias a la ABSTRACCION, somos capaces de crear CLASES !!!

Cuando un niño le dice a su mamá, 'quiero ir al circo', el niño no especifica cual ni cómo ha de ser, el se refiere a la CLASE CIRCO. Esta usando lo GENERAL aunque su experiencia siempre ha sido con lo PARTICULAR. ¿Impresionante, no?

Bien, centrándome en los bytes... Todos estos factores tan importantes, pusieron de manifiesto la importancia del OOPS. Hoy, ninguna empresa importante de programación, duda de la extraordinaria POTENCIA de la PROGRAMACION ORIENTADA AL OBJETO.

Si un ordenador es capaz de simular una sesión de vuelo en avión (un simulador aéreo), ¿porque no iba a ser capaz de simular el universo de CLASES y OBJETOS que nos rodea? OOPS...

Se ha desarrollado toda una teoría del sistema OOPS, cuyas principales características paso a describiros:

- **OBJETO:** Es ALGO que tiene una serie de DATOS y que se comporta de unas formas determinadas ante determinados estímulos o mensajes. (Si yo le pego un martillazo a un vaso, mi mensaje es 'TE GOLPEO', el vaso saltará en trozos, ese es su METODO para ese MENSAJE).

- **CLASE:** Mientras que el OBJETO es algo REAL y TANGIBLE, la CLASE es ABSTRACTA. La CLASE indica las REGLAS por las que han de crearse y comportarse los OBJETOS de esa CLASE. ¿ Cómo sería la CLASE 'VASO'?

    DATOS: Tamaño, Peso, color, forma...
    METODOS: Llenarse, Rodar, romperse, vaciarse, destruirse, etc...

Existe una regla de ORO que no debáis olvidar: En OOPS todo lo que tenga una serie de DATOS y REALICE una serie de METODOS es susceptible de ser CONSTRUIDO COMO CLASE. Esto es muy importante. Cuando se trate de crear CLASES que tengan similitud con el mundo REAL, será muy fácil. Lo difícil es cuando vayamos a crear CLASES que no tengan nada que ver con el mundo REAL... Ahí la informática trasciende a la naturaleza, y nos permitirá crear CLASES totalmente inéditas!

Transformado a Word por :

JuanReyes@iname.com
Web de Manuales
http://members.xoom.com/manuales
http://members.xoom.com/jonysoft