Manu Expósito Suárez presenta...

Los nuevos bucaneros. Episodio 2.

Lenguaje C para programadores Harbour

Abril 2021

v.7.00

Indice

Harbour y Lenguaje C.

1. Prólogo y propósito del curso.

Prólogo by Rafa Carmona

Un día charlando con un Manu Expósito me comentó cómo se titulaba el libro que escribí hace ya más de 19 años, **Los nuevos bucaneros del planeta, *Parte 1***.

Era un libro pensado y creado simplemente para mí, haciendo una introducción al inicio, entre otras cosas, sobre la magia que existía en la comunicación entre el lenguaje C y Harbour y liberado libre para cualquiera que le interesara el tema.

¿ Y por qué explico esto ? Porque me comentó que quería usar el mismo nombre.
Bien, le dije, que el libro era la *Parte 1*, y que si quería usar el mismo nombre o similar, entonces debería ponerle *Parte 2*, porque la *Parte 2* estaba todavía pendiente desde hace ya más de 19 años, y para mí sería todo un honor que la *Parte 2* sea esta.

Conozco a Manu Expósito hace muchos años, y conozco su cabezonería en poder optimizar el rendimiento al máximo en Harbour, y para ello C es el rey.

Ha llovido mucho desde entonces, y con ello el sistema entre C y Harbour ha sido ampliado y mejorado, y es necesario más que nunca esa Parte 2 de Manu Expósito.

He podido observar el temario que está preparando Manu Expósito sobre esta Parte 2 de los bucaneros, y sé que, no defraudará y no dejará a nadie indiferente.

Manu Expósito, es una de las personas que más sabe sobre la conexión entre C y Harbour, ha escrito su famosa librería HDO en puro C, si , todo está construido sobre el lenguaje C y conectado con Harbour, hasta las clases que usas en HDO han sido creadas desde C, puedes estar seguro que lo que tienes en la mano leyendo es oro puro y se te quitará ese miedo que tenemos todos cuando nos adentramos en un mundo desconocido y mágico como es el Lenguaje C, el rey indiscutible; ¡Viva C! ¡Viva Harbour!

Muchas gracias Rafa siempre te llevaré en el alma por los siglos de los siglos...

Y ahora el mío…

Desde hace bastantes años he tenido la intención de escribir unos apuntes para utilizarlos personalmente.
Hace mucho tiempo también, se lo referí a un compañero y me propuso la idea de que esos apuntes se convirtieran en un libro y publicarlo en alguna plataforma para que el que estuviera interesado pagara una pequeña cantidad y se pudiera beneficiar del mismo.
Eso sería también una recompensa para mí mismo.

En su día hice la prueba pero creo que no estaba yo muy entusiasmado.
Ahora se están dando las circunstancias necesarias para ponerme manos a la obra y empezar esta aventura.
Sencillamente tengo ganas…!!!

En su día mi amigo Rafa Carmona hizo un libro que se llamaba Bucaneros. En este libro se trataba todo lo referido a la programación en Harbour pero a alto nivel osea en PRG, haciendo una pequeña referencia al sistema extendido de Harbour. Por cierto, le he pedido a Rafa que si quiere nos escriba unas notas que con gusto sumaré a este prólogo.
Desde aquella fecha hasta ahora ha llovido mucho… Harbour se ha hecho mayor de edad y el sistema extendido ya esta muy maduro. Es la hora de tratar estos temas en profundidad.

Tengo que decir que Harbour integra “de Casa” casi todo lo que podamos necesitar y además de una manera muy óptima. Por lo que será casi innecesario que nos adentremos en los mares de C.
Pero este libro estará hecho para ese “casi”.
Por ejemplo si queremos integrar librerías de enlace dinámico (DLL) o desarrollar funciones que hagan más rápidas ciertas funciones críticas que son cuello de botella, será necesario echar mano del omnipresente Lenguaje C.
Al fin y al cabo el propio Harbour está construido en C.
Por suerte Harbour hereda del legendario Clipper el “Sistema Extendido y el Sistema ITEM API “ que es el método por el que podremos hacer la magia. Además a diferencia de lo que ocurría con Clipper, en Harbour no hay indocumentadas ya que tenemos a nuestra disposición todo su código fuente.
El sentido de este libro es de hacer de ***Cuaderno de bitácora*** en esta singladura que sabemos como empieza... pero no como acaba porque está por escribir. Este libro esta vivo se retroalimentará con lo que todos aportéis.
La idea es que iré escribiendo tema a tema y os lo entregaré en formato PDF y ePub para que lo uséis como un bloc de notas ampliando lo que sea necesario. Si luego queréis me lo podréis enviar para enriquecer lo que luego será el libro, que al finalizar recibiréis todos los inscritos.
El fin último es que cuando termine el viaje que recorrerá los Siete mares, todos seamos capaces de escribir nuestras funciones en C que mejoren la robustez y el rendimiento de nuestros programas y que podamos hacer nuestras propias librerías…

¡¡¡Así que… no perdamos ni un segundo!!!

Se abre el telón y comienza la aventura…

2. Herramientas necesarias para el curso

Antes de entrar en materia hay que saber que para poder realizar las prácticas del curso será necesario que tengáis al menos, las siguientes herramientas:

1) Editor de textos.

2) Compilador de lenguaje C.

3) Compilador Harbour con las libs preparadas para el lenguaje C elegido.

Podéis elegir los que gustéis ya que el curso será independiente de si queremos ejecutables de 32 o 64 bits o un copilador de C u otro y además para cualquier sistema operativo.

Como recomendación os diré que yo personalmente voy a utilizar:

1) VSCodium como editor

2) MinGW 9.30 de 64 bits

3) Harbour con las LIBs construidas con el compilador de C del punto 2

Pero ya os digo, podéis seguir usando lo que tengáis en este momento.

Como gestor de proyectos usaremos la herramienta presente en la instalación de Harbour ***HbMk2*** que es muy potente y nos facilitará enormemente el trabajo a la hora de construir nuestros ejecutable o librerías.

3. Introducción. Un poco de C.

En este apartado vamos a ver muy rápidamente la historia del Lenguaje C.

Hay muchísimas reseñas de como nació el Lenguaje C así que esto será muy breve y lo aprovecharemos para hablar de algunos conceptos básicos de la programación.

El Lenguaje C nació unos años antes que Clipper y por lo tanto que Harbour. Corría el año 1972 cuando un barbudo llamado Dennis Ritchie publicó la primera versión. Trabajaba en los legendarios laboratorios Bell de una compañía de telecomunicaciones norteamericana, la ATT.

Pero, porqué el nombre de lenguaje C? Pues sí, porque anteriormente hubo un Lenguaje B y posteriormente un lenguaje D. Creo que el bueno de Dennis no se estrujó mucho su coco para bautizarlo.

Inicialmente se usó para la implementación del sistema operativo UNIX padre de todos los Linux actuales.

Luego se ha usado para la construcción de otros sistemas operativos (SO) como los Linux, iOS o Windows.

Es un lenguaje tan potente, robusto, estructurado y que tiene muy pocas palabras clave.

Todo ello ha hecho que sea el lenguaje utilizado para desarrollar otros lenguajes como nuestro querido Harbour.

Como sabéis hay varios tipos de lenguaje:

* Lenguaje de bajo nivel:

Están totalmente acoplados a la máquina.

- Lenguaje Máquina que es el único que entiende nuestra computadora.

- Lenguaje ensamblador que es una evolución del Lenguaje Máquina que pone nombres abreviados a las instrucciones del lenguaje máquina.

* Lenguaje de alto nivel:

Son independientes de la máquina en los que se van a ejecutar y se parecen al lenguaje natural de los humanos (que hablan inglés)

En este bloque está el C, Harbour, Java, PHP, Basic, Pascal, C#, Fortran, COBOL etc

Dentro de estos lenguajes los hay de propósito general y lenguajes especializados en tareas concretas.

Por ejemplo C y Harbour son de propósito general pero Fortran está especializado en cálculos matemáticos y COBOL para hacer programas de gestión.

Para que una máquina entienda los lenguajes de alto nivel necesita que los conviertan en lenguaje máquina.

Hay varios tipos de ***traductores:***

- Ensambladores: generan código máquina a partir del lenguaje ensamblador

- Interpretes: que verifican y traducen a código máquina linea a linea cada instrucción y luego se ejecuta.

- Compiladores: Analizan y traducen a código máquina todo el programa para luego ser ejecutado.

Cada tipo de traductor tiene sus ventajas e inconvenientes, si queréis podemos hablar del tema en clase ;-).

Bien, el C pertenece a los compiladores ya cuando queremos construir un ejecutable C genera un OBJ libre de errores y luego la herramienta de enlace (LINK) obtiene lo necesarios de las LIBs o de otros obj para generar un EXE independiente. El lenguaje BASIC es el ejemplo típico de lenguaje interpretado, va verificando la instrucción según la encuentra, la traduce a lenguaje máquina y la ejecuta.

Harbour interpretado como el Basic sólo que para que el proceso sea más rápido y óptimo crea un código muy optimizado que no llega a ser lenguaje máquina pero que los mortales humanos no sabríamos descifrar. Ese código se llama pCode que luego interpreta la máquina virtual de Harbour.

El pCode de Harbour es muy rápido de ejecutar por la máquina virtual además de estar libre de errores léxicos.

Si compiláis un PRG con la opción -gh se genera un archivo *.hrb que contiene el pCode, en Harbour se llama binario portable y se puede ejecutar con la herramienta incluida en Harbour llamada HbRun.exe.

Pero Harbour suele generar un EXE igual que el Lenguaje C?, os preguntareis no?

El truco es que realmente dentro del EXE va la máquina virtual y el pCode, jajaja por eso son tan gordos esos EXE. Cosa que no debe asustar ya que esa obesidad no impide que a la vez sea súper veloz. Para el Mod Harbour esto viene de escándalo...

Dejemos de momento a Harbour y volvamos a C.

Destripando los programas de C

C es muy estructurado con esas funciones o procedimientos, aunque en C se le suele llamar función indistintamente, al fin y al cabo la única diferencia entre una función y un procedimiento es que la primera devuelve algún valor y los procedimientos no devuelven nada pero las dos pueden recibir o no parámetros.

En Harbour tenemos que indicarle al compilador si estamos tratando con una FUNCTION o con un PROCEDURE, de hecho el no respetar esto puede hacer que la compilación arroje una advertencia o un error.

Por cierto que sí hay mucha similitud entre ambos lenguajes en este tema.

Que no se olvide C a diferencia de Harbour si es “case sensitive” esto quiere decir que distingue entre mayúsculas y minúscula, para C no es igual “Valor” que “valor”. Aunque os cueste al principio eso nos vendrá muy bien, por ejemplo para las funciones que creemos para Harbour las pondremos todas en mayúsculas y para las de C yo voy a usar en el curso la llamada “Notación Húngara” por lo que no van a colisionar nunca. Podéis buscar en Internet ya que hay mucha literatura del tema de las diferentes notaciones. Seguimos pues...

Lo mismo que en Harbour si existe una función que se llame “main” será la primera en ejecutarse.

En C una función es el modo de agrupar instrucciones de una manera estructurada y coherente para hacer algo, que casualidad, lo mismo que en Harbour. Dicho de otra manera, una función no es más que un conjunto de instrucciones que realiza una determinada tarea.

Vamos a ver como es el esqueleto de una función típica en C

```
/* Esto es un comentario */
#include <stdio.h>

tipo_valor_devuelto nombre_funcion( tipo parámetro, … )
{
      declaración_de_variables_locales;
      …
      instrucciones; //  Esto es un comentario
      …
      return variable_del_tipo_definido_por_la_funcion;
}
```

<u>Los comentarios</u>

Los dos tipos de comentarios que hay en C existen en Harbour y funcionan de la misma manera:

/**/ Para comentarios que se pueden extender a más de una linea y

// para comentarios en linea

En ambos casos el compilador no los considera. Solo sirven para anotaciones entre programadores.

<u>Las directivas del preprocesador</u>

También en esto coinciden Harbour y C en la manera de definirlos pero cada uno tiene diferentes directivas y algunas coincidentes.

Solo veremos de entrada las dos principales:

#include → hace que el compilador incluya el archivo dentro del archivo fuente que estamos tratando. El archivo que vamos a incluir suelen ser con la extensión “*.h” aunque puede tener

cualquier extensión. Se suele poner entre <> cuando es un archivo del entorno de compilación o sea, los archivos "includes" del compilador. Y se suele poner entre "" cuando son archivos "include" locales. Es igual que en Harbour sólo que los archivos "include" en este caso tienen la típica extensión "*.ch" Como puedes comprobar funcionan exactamente igual en los dos compiladores.

En C antes de poder usar algo tiene que definido. Ese es el principal cometido de los archivos *.h que solemos incluir al principio de los programas

Ejemplos:

```
#include <stdio.h>
```

```
#include "hdo.api"
```

#define → sirve para definir nombres simbólicos que no van a cambiar a lo largo del programa o sea son *constantes.* En este caso, por suerte, también funcionan igual en los dos compiladores. El compilador se encargará de sustituir el símbolo por su autentico valor en tiempo de compilación.

Ejemplos:

```
#define MAXIMO 90
```

```
#define MINIMO 25
```

Nombre de la función → será el que usemos cada vez que necesitemos invocarla, por lo que es muy importante que sea auto descriptiva. Como comentamos más arriba debe estar definida antes de usarse. En la declaración hay que poner el tipo de valor que devuelve, el nombre y entre () los parámetros que recibe separados por "," y precediendo a cada uno de su tipo:

```
int doble( int ) ; // Esta es la definición
```

Ahora la implementación:

```
#include <stdio.h>
/* Función que devuelve el doble del numero entero pasado */
int doble( int iNum )
{
	int iRet;

	iRet = iNum * 2;
	return iRet;
}
```

En la definición no es necesario poner los nombres de las variables de los parámetros, sólo el tipo, en cambio, en la implementación es obligatorio.

Lo siguiente que nos encontramos es las {} entre ellas están todas las instrucciones que van a formar el cuerpo de la función. Ojo, cada instrucción termina con "**;**".

Otro ejemplo con más de un parámetro:

```
#include <stdio.h>
int multiplica( int iParam1, int iParam1 )
{
	int iRet;

	iRet =  iParam1 *  iParam1;

	return iRet;
}
```

4. Operadores

- Operadores aritméticos:

Operador	Significado
=	Asignación
*	Multiplicación
/	División
%	Resto de división entera (mod)
+	Suma
-	Resta
++	Incremento
--	Decremento

- Operadores relacionales o de comparación:

Operador	Significado
<	Menor que
<=	Menor o igual que
>	Mayor que
>=	Mayor o igual que
==	Igual a
!=	No igual que
&&	Y lógico → .and.
||	O lógico → .or.

En C hay más operadores pero vamos a dejarlo aquí. Solo decir que funcionan igual en Harbour.

Muchas veces va a aparecer la palabra ***expresión*** pero que esto?

Las expresiones son secuencias de funciones, operadores y variables o constantes que se utilizan para calcular un valor a partir de ella una vez evaluada.

5. Estructuras de control.

En este apartado vamos a ver las diferentes estructuras de programación en C con un esquema de cada una de ellas:

- Condicionales

`if` Funciona similar a como lo hace en Harbour

```
if( condición )
{
	sentencias
}
```

`if else`

```
if( condición )
{
	sentencias
}
else
{
	sentencias condición por defecto
}
```

`if else if`

```
if( condición 1 )
{
	sentencias;
}
else if( condición 2 )
{
	sentencias
}
// Puede haber más else if()
else
{
	sentencias condición por defecto
}
```

`? Operador condicional ternario:` con la siguiente sintaxis

```
expresión1 ? expersión2 : expresión3;
```

En Harbour existe una manera de hacerlo y que funciona exactamente igual se trata de

```
iif( expresión1, expersión2, expresión3 )
```

Si la expresión1 devuelve verdadero toma el valor de la expresión2 de lo contrario tomará el valor de expresión3.

Ejemplo de como calcular el máximo entre dos números:

```
iMax = a > b ? a : b;
```

Es lo mismo que hacer:

```
if( a > b )
{
	iMax = a;
}
else
{
	iMax = b;
}
```

`switch` Funciona similar a como lo hace en Harbour pero en C sólo se puede usar con expresiones de tipo entero o carácter. Equivaldría a **`if`** anidados.

```
Switch( expresión )
{
	case exprConst1:
		sentencias
		break;
	case exprConst2:
		sentencias
		break;
	case exprConstN:
		sentencias
		break;
	default:
		sentencias por defecto
		break;
}
```

- Iterativas

while

```
while( condición )
{
	sentencias;
}
```

do while esta no existe en Harbour. Garantiza que se van a ejecutar las sentencias del cuerpo al menos una vez

```
do
{
	sentencias
} while( condición )
```

for la sintaxis es muy diferente a la de Harbour

```
for( bloque_de_inicializar; expresión_condicional; bloque_de_incremento )
{
	sentencias
}
```

`bloque_de_inicializar:` si existe, se evalúa una vez y es donde se inicializan una o más variables.

`expresión_condicional:` si existe, se evalúa en cada iteración y aquí se comprueba la expresión de salida natural del `for.`

`bloque_de_incremento:` si existe, este bloque es donde se incrementan/decrementan las variables de control.

En cada bloque puede ir más de una expresión separadas por el operador coma *“,”*

Ejemplo, escribe la tabla de multiplicar del numero n pasado:

```
for( i = 0; i <= 10; i++ )
{
	printf( “n x %d = %d\n“, i, i * n );
}
```

Importante para este tipo de estructuras de programación las sentencias **`continue`** que fuerza una nueva iteración dentro de de una estructura repetitiva y **`break`** que fuerza la salida de una estructura selectiva o repetitiva.

6. Tipos de datos elementales de C

Esto se pone interesante… “Tú eres mi única constante en este mundo de variables!!” bonita frase no? Pero que es eso?

Todos los programas necesitan guardar la información que se va a utilizar en el mismo. Ese espacio de memoria se llaman variables y constantes. En Harbour hay un objeto más además de las variables y las constantes son los FIELD o campos que guardan los valores de los campos de las DBF.

Variable: Formalmente se dice que una variable es un objeto nombrado capaz de contener un dato que puede ser modificado durante la ejecución de programa. En C, las variables tienen un tipo, que es necesario especificar. Las variables se almacenan en la memoria, y el espacio que ocupan depende de su tipo. En Harbour no es necesario el tipo ya que dependerá del valor que se le asigne. Ya hablaremos de por qué Harbour es tan listo!!!.

Ejemplo:

```
char cAracter;

long lNum;

char *szCadena;
```

Constante: es lo mismo que una variable pero su valor no puede cambiar a lo largo de la ejecución del programa.

La mejor manera de declarar una constante es con #define pero también se puede usar el cualificador *const.*

Veamos unos ejemplos:

```
#define VERDAD 1

#define FALSO 0

#define MAX_EDAD 90

10, -5, “Manu” // también son constantes
```

En este caso se llaman constantes simbólicas y se usan para hacer el programa más entendible y fácil de manejar. El compilador se encarga de sustituir la constante por su valor. En el caso del ejemplo si la constante MAX_EDAD cambiara alguna vez a 100 solo bastaría modificarla en el define y no en cada aparición de la misma.

Este sistema de definir variables también se puede usar en Harbour.

En C está otra manera que es usando const, ejemplo:

```
const int iMaxEdad = 90;

const char *szNombre = “Manu”;
```

C estándar es fácil engañar con este último tipo de declarar constantes, así que recomiendo el primero.

Una vez visto el mundo de las variables y de las constantes hablemos de como se declaran y esto nos dará pie para tratar los tipos, los modificadores y los cualificadores.

Una variable se declara así: `tipo nombre;`

En C a diferencia de Harbour además de declarar las variables antes de usarlas hay que decirle el tipo.

Los tipos en C son:

- `int` *Números enteros.*

- `char` *Caracteres.*

- `float`, `double` *Números reales.*

Cada uno de ellos ocupa un espacio en memoria y hay que decirle al compilador el tipo para que reserve la necesaria para guardar dicho tipo. Por eso se dice que *C es altamente tipado*

Luego hay modificadores que hacen que el tamaño de la reserva de memoria varíe:

`short, long` se aplica a los enteros *`int`*.

Ejemplos:

```
char cLetra = 'A';

int iEdad;

long int lDistancia;

short int sAlto;
```

Estos dos últimos se podrían declarar sin poner la palabra `int`:

```
long lDistancia;

short sAlto;
```

Para declarar una cadena de caracteres se puede hacer así:

```
char *szCadena = "Esto es una cadena"; // o

char szCadena[] = "Esto es una cadena";
```

De esto tendremos que profundizar más adelante.

Lo que tenéis que sacar en claro es que es obligatorio en C indicar el tipo y que cada tipo ocupa un espacio en memoria.

El tipo carácter *char* C lo toma como un entero de un byte, lo digo porque se le puede aplicar el modificador *unsigned* o sea "sin signo" como a los *int.*

Ejemplos:

```
unsigned int uiEdad;

unsigned long ulDistancia;

unsigned char cLetra;
```

Tipos de datos en C en máquinas i686

Tipo	Tamaño (bytes)
char, unsigned char	1
short int, unsigned short int	2
int, unsigned int, long int, unsigned long int	4
float	4
double	8
long double	12

Por si alguien no se acuerda 1 byte == 8 bits.

Desgraciadamente los `int` pueden cambiar si la plataforma es de 32 o 64 bits, por lo que en la medida de lo posible es mejor no usarlos.

Hasta aquí los tipos elementales, simples o primitivos.

7. Tipos de datos estructurados: Tablas, arreglos o arrays vectores, matrices y tablas multidimensionales en C.

Primero, ¿qué es? Se llaman tipos estructurados o complejos porque están compuestos por los tipos elementales ya explicados, esta definición vale también para las estructuras que vamos a ver en el siguiente punto.

El concepto de tabla es exactamente el mismo que en Harbour, pero sólo el concepto. En general se podría decir que una tabla es una ***variable estructurada*** que contiene ***elementos del mismo tipo almacenados en memoria*** de una forma ***consecutiva***.

Tenemos que recordar que un array o tabla en Harbour es una variable de tipo 'A' y que puede contener elementos de cualquier otro tipo de los existentes en Harbour. Fijaros que en C los arrays sólo pueden contener elementos del mismo tipo de C y esto constituye una gran diferencia.

C necesita saber el tamaño del tipo que compone el array para poder hacer una composición en memoria que servirá, entre otras cosas, para recorrer los elementos del mismo con alguna sentencia repetitiva del tipo FOR o WHILE por ejemplo. Podríamos saber el numero de elementos de un array sabiendo la memoria que ocupa en su totalidad y dividiéndolo por el tamaño de uno de sus elementos. Para ello es muy útil el operado ***sizeof*** que devuelve el tamaño de un tipo o de una variable pasada, ejemplo

```
iAnchoInt = sizeof( int ); // Ojo int es el tipo no una variable
```

- Vector: es un array de una dimensión, o sea, el contenido no son otros arrays.

La declaración es: tipo nombre[dimensión o numero elementos]; // **dimensión = n. elementos**

Ejemplo:

```
long lDistacias[ 3 ] = { 2445456, 123785, 9842456 };
```

Importante, en C no existe el tipo cadena o string.

En C una cadena o string es un vector del tipo carácter.

Ejemplos, todos hacen es lo mismo:

```
char szCadena[ 5 ] = { 'h', 'o', 'l', 'a', '\0' };
```

```
char szCadena[] = { 'h', 'o', 'l', 'a', '\0' };
```

```
char szCadena[ 5 ] = "hola";
```

```
char szCadena[] = "hola";
```

o con la notación de puntero:

```
char *szCadena = "hola";
```

Esta última notación la veremos más adelante en el tema de punteros. De momento confórmate con mi palabra :)

Fíjate en la primera notación en el detalle del carácter especial *'\0'* que indica a C el final de cadena, de ahí que en mi notación húngara yo haya puesto el prefijo ***sz*** a Cadena (**sz**Cadena) con lo que quiero remarcar que es una cadena (s) terminada en 0 (z) (zero en inglés).

- Matriz: es un array de 2 dimensiones, o sea que contiene 2 vectores.

La declaración es: tipo nombre[dim1] [dim2];

Ejemplo:

```
Long lDistacias[ 2 ][ 3 ] = {
                              { 2445456, 123785, 9842456 },
                              { 3456454, 754332, 4656567 }
                             };
```

- Array multidimensional: es un array que contiene más de dos vectores.

Realmente una matriz es un array multidimensional de dos dimensiones, pero los informáticos somos así. En realidad los vectores y las matrices son los array que mas se usan.

Por cierto, los arrays multidimensionales casi nunca se inicializan, el ejemplo es para que os hagáis una representación mental.

La declaración es: tipo nombre[dim1] [dim2], ….;

Ejemplo:

```
Long lDistacias[ 2 ][ 3 ][ 3 ] = {{ { 2445456, 123785, 9842456 },
                                     { 3456454, 754332, 4656567 } },
                                   { { 2445456, 123785, 9842456 },
                                     { 3456454, 754332, 4656567 } },
                                   { { 2445456, 123785, 9842456 },
                                     { 3456454, 754332, 4656567 } }
                                  };
```

Pero realmente, a C le da igual la representación, C utiliza las dimensiones expresadas entre los corchetes para hacer los cortes. Sí amigos, podríamos haber hecho lo que pongo a continuación y el compilador de C no hubiera protestado y lo que es más importante lo trata perfectamente igual:

```
Long lDistacias[ 2 ][ 3 ][ 3 ] = { 2445456, 123785, 9842456,3456454, 754332,
4656567, 2445456, 123785, 9842456, 3456454, 754332, 4656567, 2445456, 123785,
9842456, 3456454, 754332, 4656567 };
```

Para acceder al contenido o asignación del mismo se usa los indices entre corchetes, por ejemplo:

```
lDistancia [ 2 ][ 1 ][ 1 ];
```

Importante: *en C el indice de los arrays empiezan en 0 y no en 1 como en Harbour* por ejemplo:

```
char szCadena[ 5 ] = “hola”;
// Si queremos saber el contenido de la segunda posición:
szCadena[ 1 ]; // el primero sería szCadena[ 0 ]
```

8. Tipos de datos estructurados: estructura, uniones y enumeraciones.

Estructuras:

Es un tipo de datos complejo porque al igual que en las tablas, puede tener uno o más *miembros*. Como decía mi profe, conjunto finito de elementos de cualquier tipo agrupados bajo el mismo nombre, para hacer más eficiente e intuitivo su manejo.

A cada elemento de la estructura se le denomina campo o miembro. Y al contrario de lo que sucede con las tablas, no se puede asegurar que los elementos de una estructura se organicen en memoria consecutivamente.

En otros lenguajes como Pascal las estructuras se llaman Registros, en Harbour no existen como tales pero se podría emular con clases que solo tengan datas o variables de instancia.

Para declarar una estructura se usa la clausula ***struct*** seguida del nombre y entre las llaves de apertura y cierre se declara cada uno de los miembros con su tipo.

Tal vez con un el ejemplo se vea más claro.

Ejemplos:

```
struct persona {
      char NIF[ 10 ];
      char nombre[30];
      int edad;
      int sexo;
};
```

¡¡¡Ojo!!! Es muy importante recordar que al declarar una estructura sólo estamos describiendo la composición de la misma, y *no estamos declarando ninguna variable*, por lo que no estamos reservando ninguna memoria. Para hacer un símil con Harbour es lo mismo que cuando se define una clase con la clausula CLASS … la variable se declara cuando se hace por ejemplo:

```
local oPersona := TPersona():new()
```

En C para declarar una variable de un tipo de estructura determinado, por ejemplo de la estructura propuesta en el ejemplo, se hace así:

```
struct persona alumno; // Mismamente
```

Hay más maneras de declarar variables de un tipo de estructura por ejemplo:

```
struct {
      char NIF[ 10 ];
      char nombre[30];
      int edad;
      int sexo;
} alumno;
```

Esto se hace cuando solo va a usar la definición de la estructura una sola vez.

Ya veremos más adelante que existe una manera más elegante e intuitiva que es usando *#typedef* para crear nuestros propios ”tipos” de datos.

La manera de inicializar una estructura a la vez que se declara es:

```
struct persona alumno = { “98879345J", “Pablo", 21, 0 };
```

Hay otra un poco desconocida y poco usada:

```
struct persona alumno = { .edad = 21, .nombre = "Pablo" };
```

La manera de acceder a un campo de la estructura es la siguiente:

```
alumno.edad = 19;
```

```
alumno.sexo = 1;
```

Ya veremos que hay otra manera de acceder cuando lo que tenemos es un puntero a la estructura:

```
alumno->edad = 19;
```

Esto lo veremos más adelante cuando tratemos los *punteros.*

Como colofón, sólo decir que la API de Harbour está llena de estructura lo cual indica lo útiles que son. Un ITEM desde el punto de vista de C es una estructura.

Uniones:

Las uniones son muy parecidas a las estructuras, con la particularidad de que todos los campos comienzan en la misma posición de memoria, o sea, que ocupan una misma área de memoria y por tanto puede verse como de diferentes tipos, dependiendo del momento. El tamaño en memoria no es la suma de los campos que la componen sino el tamaño del mayor de los campos.

Se declaran exactamente igual que las estructuras pero cambiando la palabra struct por union:

```
union miItem {
      int entero;
      float real;
      char cadena[ 30 ];
};
```

Y todo lo dicho para la estructuras es aplicable a las uniones, declaración y asignación. No me voy a extender más. Tan sólo decir que las variables de Harbour se basan en uniones, de hecho la representación de una variable de Harbour en C es una estructura que incluye entre sus miembros un campo que indica el tipo y una union con cada uno de los tipos disponibles en un PRG de Harbour. Esa estructura se llama HB_ITEM.

Enumeraciones:

Al igual que con las estructuras y las uniones con las enumeraciones podemos declarar nuevos tipos de datos para ello se utiliza la palabra ***enum.***

Las enumeraciones nos permiten definir un conjunto de valores que representarán algún parámetro cuyos valores sean discretos y finitos como por ejemplo los meses del año o los días de la semana.

Ejemplo de declaración:

```
enum meses { ene, feb, mar, abr, may, jun, jul, ago, sep, oct, nov, dic };
```

Luego podríamos crear una variable del tipo meses:

```
enum meses mesAnio;
```

Realmente lo que que utilizamos son identificadores, no cadenas de caracteres, los elementos no están entre “. Internamente, el C convierte cada uno de estos identificadores en números enteros, empezando por el 0 y de forma consecutiva. Aunque podemos asignar un valor diferente a los identificadores:

```
enum meses { ene = 1, feb, mar, abr, may, jun, jul, ago, sep, oct, nov, dic };
```

esto haría que empezara por 1 y no por 0, con lo que en este caso, todos se incrementarían en uno. Esta asignación se puede hacer a cada identificador si fuera necesario. Por ejemplo:

```
enum mascotas { perro = 101, gato = 202, erizo = 305, loro = 418 };
```

De hecho C convierte en enteros a cada identificador y por tanto se pueden usar como un entero más, Podríamos hacer esto:

```
perro + loro; // 101 + 418
```

Podríamos crear nuestro propio tipo bool en C de esta manera:

```
enum bool { F, T }; // F → 0 y T → 1
```

9. Creando nuestros propios tipos con typedef

C permite asignar sinónimos a cualquier tipo de dato de los explicados tan elementales como los agregados o estructurados. Esto hace más fácil definir variables de ese tipo, sobre todo de las estructuras, uniones y enumeraciones. Para ello se utiliza la palabra ***typedef***.

Por ejemplo dada la estructura

```
struct persona {
      char NIF[ 10 ];
      char nombre[30];
      int edad;
      int sexo;
};
// Creamos nuestro tipo
typedef struct persona TPersona; // Ya tenemos nuestro tipo  TPersona
// Declaración de una variable de tipo TPersona:
TPersona alumno;
```

Como puedes ver es más fácil que hacer:

```
struct persona alumno;
```

En los includes de Harbour hay muchos tipos definidos así, por ejemplo ***PHB_ITEM*** que veremos continuamente en este libro :)

En harbour también se definen ***HB_INT, HB_LONG, HB_BOOL***, etc que son los recomendables usar cuando hacemos nuestras funciones en C para Harbour en vez de **int, long**, por ejemplo (el bool, no existe en C). La combinación de ***typedef*** con las estructuras sobre todo da mucha facilidad de uso de esos tipos al Lenguaje C.

Importante como resumen y para terminar con el tema de tipos de datos:

- Declaración de una variable:

```
tipo variable;
```

Ejemplo:

```
int edad;
```

- Asignación de una variable ya declarada:

```
edad = 57;
```

ATENCIÓN IMPORTANTE:

- Visibilidad o alcance de una variable

Dependiendo de donde se declare una variable podrá ser:

Local: son las que se declaran dentro de una función. Son visibles sólo dentro de la función en la que se declaran.

Global: las que se declaran al principio del programa y fuera de cualquier función. Son visibles en todo el programa. Incluso en otros programas si se le antepone la palabra reservada ***extern.***

- Vida de una variable

Las variables **globales** estarán vigentes dentro de todo el programa donde se definen incluso fuera del mismo con el uso de ***extern***.

Las **locales** terminan su vida cuando se sale de la función en las que se declaran.

Pero cuando se declara una variable como `static` el valor de la misma permanece hasta que se sale de la ejecución del programa. Esto significa que funcionan como en Harbour.

- Conversión forzada o cast

Por ejemplo, si desea almacenar un valor '`int`' en un '`long`', puede hacer un cast '`int`' en '`long`'. Puede convertir los valores de un tipo a otro explícitamente utilizando el operador de conversión de la siguiente manera:

```
(type_name) expresión
```

Por ejemplo:

```
int suma = 17, contador = 5;
float media;

media = (float) suma / contador;
```

Dejamos aquí el tema de las variables.

10. Punteros en C.

Como dice mi amigo Antonio esto es ya para nota :)

Realmente puede parecer un poco complejo pero entendiendo la idea no lo es tanto.

Vamos al lío…

Antes de usar cualquier variable en C hay que declararla e incluso podríamos iniciarla.

Pues bien, todas las variable tienen un valor que ocupa un espacio de memoria.

Para que un programa conozca el valor de una variable necesita saber en qué parte de la memoria se encuentra, el tamaño y el tipo.

Se podría decir que un puntero es una variable cuyo contenido es la dirección de la memoria que ocupa otra variable.

Se podría representar la memoria como una calle con casas de igual tamaño y que cada una tiene un numero que la identifica. Sabiendo el numero podríamos enviar una carta a esa dirección… eso sería el puntero a esa casa, jajaja no sé si es muy buena la comparativa… :)

Declaración de punteros:

```
tipo_variable * nombre_variables; // Fijate el signo "*"
```

Como ves se declara como una variable más, con el tipo determinado que puede ser cualquiera de los vistos elementales o complejos y el nombre precedido por "*".

Ejemplos:

```
int *piEdad; // Puntero entero
```

```
float *plDistancia; // Puntero a float
```

Lo de la p delante yo lo uso para saber que es un puntero, no es obligatorio.

Hay un tipo especial de puntero, los ***`void`***, son punteros que solo se sabe eso, que son punteros, pero desconocemos su tipo, Cuando se sepa tendremos que hacer un `cast` así:

```
void *pDesconocido;
```

```
(int*) pDesconocido;
```

Luego en alguna parte cuando se le asigne el valor determinado habrá que hacer:

```
(tipo_forzado *) pDesconocido;
```

Esto lo veremos más adelante con la reserva de memoria dinámica, de momento lo dejo aquí.

Declarar un puntero a un determinado tipo no es lo mismo que declarar una variable de ese tipo.

Es importante para evitar problemas, inicializar los punteros en su declaración antes de trabajar con ellos. Si no se sabe el valor inicial se asigna el valor especial ***NULL***.

Por lo tanto los ejemplos que hemos puesto anteriormente deberían estar así:

```
int *piEdad = NULL; // Puntero entero
float *plDistancia = NULL; // Puntero a float
void *pDesconocido = NULL; // Puntero sin tipo inicial
```

Así evitaremos muchos problemas, ya que un puntero sin inicializar puede contener cualquier valor.

Operadores de punteros:

Para trabajar con los punteros, C pone a nuestra disposición dos operadores:

- De dirección &: se pone delante de cualquier variable y devuelve la dirección de dicha variable.

Ejemplo:

```
int *piEntero = NULL;
int iEntero = 100;

piEntero = &iEntero;  // piEntero contiene ahora un puntero a la variable  iEntero
```

- De contenido *: se pone delante de una variable de tipo puntero y devuelve el valor que contiene el puntero.

Ejemplo, DimeN es una función que devuelve por pantalla el numero pasado (creelo)

```
int *piEntero = NULL;
int iEntero = 100;

piEntero = &iEntero;  // piEntero contiene ahora un puntero a la variable iEntero

DimeN( iEntero ); // Muestra por pantalla 100
DimeN( *piEntero ) // También muestra por pantalla 100
```

Muy importante. El operador ***** se utiliza en la declaración como indicador de puntero, pero fuera de la declaración es el operador de contenido. Por lo tanto es un operador sobrecargado con dos funciones diferente en relación con los punteros. Con los puntero de tipo ***void*** no se puede usar ya que C necesita el tipo para saber el tamaño de memoria y poder devolver su contenido. El hecho de que dos punteros contengan el mismo valor no implica que los dos punteros sean iguales.

El operador **&** sólo se puede aplicar a variables en cambio el ***** se aplica a variables y expresiones de tipo puntero.

Esto es incorrecto: `&( iEntero + 10 );`

Aritmética de punteros: Hay la posibilidad de sumar o restar un entero a un puntero. No hay que confundir con la suma y resta de los números enteros o reales, no tiene nada que ver.

Las operaciones aritméticas admisibles para los punteros son: sumarle o restarle un entero, obteniendo como resultado un puntero del mismo tipo.

Los punteros se incrementan (+) y decrementan (-) en función del tipo de datos al que apuntan. Si se le suma uno a un puntero, el nuevo valor del puntero apuntaría al siguiente elemento en la memoria. *Este tema lo trataremos con más amplitud durante el resto del curso.*

Puntero a puntero: como cualquier otra variable se puede tener un puntero a otro puntero. Ya veremos la utilidad de esto más adelante.

Los puntero se pueden usar en C para pasar variables por referencia:

Ejemplo:

Sea la definición de esta función:

```
miFunc( int *piUno, int *piDos );
```

En alguna parte de nuestro programa podríamos hacer

```
int iUno = 5;
int iDos = 7;

miFun( &iUno, &iDos ); // Los cambios en iUno e iDos serán visibles desde aquí
```

Con lo que todo lo que hagamos en la función miFun con los dos parámetros sera visible en la función que llama a miFun. Funciona igual que en Harbour.

Ejemplo:

```
// Intercambia los valores de x e y
void cambio(int * x, int * y)
{
	int aux = *x;
	*x = *y;
	*y = aux;
}
```

11. Reserva y liberación de memoria dinámica.

Cuando declaramos una variable el compilador se encarga de reservar la memoria que va a usar esa variable. Pero hay ciertas ocasiones que esto no es posible, por ejemplo en los arrays que puede que aumenten su tamaño durante la ejecución del programa.

C pone a nuestra disposición básicamente 3 funciones:

`void *malloc( size_t tamaño )`: reserva la cantidad de memoria indicada por el parámetro "tamaño", y devuelve el puntero al comienzo de la zona de memoria asignada, o *NULL* si no se ha podido reservar.

`void *calloc( size_t numero, size_t tamaño )`: igual que *`malloc,`* esta función intenta reservar memoria pero si no lo consigue también devuelve *NULL*. Pero tiene dos diferencias, la primera es que devuelve la memoria del producto del número por tamaño y la segunda es que inicializa a cero toda la memoria reservada.

malloc se usa cuando queremos reservar memoria para un único elemento y ***calloc*** para un conjunto de ellos como por ejemplo un array.

`void free( void *puntero )`: es la función que libera la memoria reservada con las dos funciones anteriores.

En las tres funciones hay que utilizar un ***cast*** para forzar el cambio de tipo ya que devuelven *`void`* o recibe *`void`* en el caso de la función *`free.`*

Dicho esto…

IMPORTANTE: *No utilizar esas funciones en nuestras funciones de C para Harbour* ya que la librería estándar de Harbour nos provee de una funciones específicas con unas características propias y especiales.

Las funciones que vamos a utilizar en el libro para ***reserva de memoria en Harbour*** son las siguientes:

`void *hb_xalloc( HB_SIZE nSize );` Devuelve la memoria indicada por nSize y NULL si falla.

`void *hb_xgrab( HB_SIZE nSize );` Devuelve la memoria indicada por nSize y si falla se sale del programa.

`void *hb_xrealloc( void *pMem, HB_SIZE nSize );` Reasigna la memoria asociada al puntero pMem pasado a la indicada por nSize.

`void  hb_xfree( void *pMem );` Libera la memoria apuntada por pMem pasado, no devuelve nada.

Existen más funciones del API de Harbour relacionadas con la memoria pero no creo que las usemos en nuestras funciones.

Nota importante: *Usar siempre que sea posible las funciones de las API de Harbour* como por ejemplo las de tratamiento de cadenas, de archivos, etc, lo veremos más adelante.

```
char *szNombre = ( char *) hb_xgrab(  20 );
// Ya se puede usar sZNombre si no hubiera reservado la memoria saldría
// del programa con un error.
…
// Con hb_xalloc hay que controlar si se hizo la reserva ya que no
// devuelve error si fue imposible reservarla
char *szNombre = ( char *) hb_xalloc(  20 );
if( szNombre != NULL )
{
      // Se puede usar szNombre
}
else
{
      // Avisar del error
}
// Y ahora podríamos aumentar o disminuir el tamaño de la memoria así
szNombre = hb_xrealloc( szNombre, 50 );

// Por último cualquier reserva hecha con las funciones anteriores se libera con:
hb_xfree( szNombre );
```

Cualquier reserva que hagamos en nuestras funciones tendremos que liberarla ya que en C no hay un recolector de basura que lo haga por nosotros.

En los ejemplos del libro veremos mucho ***hb_xgrab*** y ***hb_xfree.***

Hasta aquí la parte de "C puro". Sé que se han quedado muchas cosas en el tintero, algunas de ellas las veremos a lo largo del libro en profundidad y otras se quedarán fuera. Pero espero que estas notas valgan como partida para entender la amplia bibliografía de C que hay en la red.

No he querido poner ejemplos de uso de lo explicado ya que no hay que confundir un programa de C puro a un programa con funciones para ser usadas desde PRG.

Podría haber hecho, por ejemplo, el típico "Hola Mundo", pero si queremos usar la salida por pantalla para que funcione desde Harbour no podríamos utilizar la función "printf" de C ya que Harbour usa las funciones del GT (General Terminal) o las que nos proporciona las librerías estándar de nuestro compilador Harbour.

En C puro:

```
void funHolaMundo( void )
{
      printf( “Hola mundo” );
}
```

En C para usar desde Harbour:

```
HB_FUNC( FUNHOLAMUNDO )
{
      hb_gtOutStd( “Hola mundo”, 10 );
}
```

Como se puede ver hay diferencias y unas normas que respetar para hacer funciones que se puedan usar desde PRG. Todo esto se explicará en su momento.

Lo que viene a partir de ahora ya si es plenamente Harbour y el uso de C para ampliar las posibilidades de nuestro lenguaje favorito…

12. Algunos conceptos básicos sobre la Máquina virtual (VM), pila (stack) y la Tabla de símbolos (symbol table) de Harbour.

Vamos con otro tema teórico pero, aún así, pienso que es necesario para entender como trabajan las funciones que vamos a hacer en C para ser usadas en PRG y porqué no, también saber como funciona por dentro los programas hechos con Harbour.

Para empezar las funciones de C que no cumplan unos requisitos que ya explicaremos más adelante no van a ser reconocidas por el sistema interno de Harbour.

Las variables de Harbour.

Como cualquier lenguaje, Harbour necesita guardar la información de los datos usados en variables.

Todas esa variables tienen un identificador que es el nombre de la misma y un valor que es el que le asignamos en cualquier momento de la ejecución del programa. La particularidad que tienen esas variables es que pueden ser de cualquier tipo simplemente igualándolas a un valor de otro tipo.

Por ejemplo:

```
procedure main
      local miVar
      miVar := “Hola”
      alert( “La variable miVar es de tipo: “ + valtype( miVar ) )
      miVar := 9
      alert( “La variable miVar es de tipo: “ + valtype( miVar ) )
      miVar := date()
      alert( “La variable miVar es de tipo: “ + valtype( miVar ) )
return
```

Esta característica no existe en C ya que es un lenguaje fuertemente tipado, esto es que en la declaración de las mismas hay que especificar el tipo que tendrá durante la ejecución del programa, esto es para que en tiempo de compilación C reserve una memoria determinada.

¿Entonces como consigue Harbour que una variable pueda contener cualquier tipo de dato?

Ya dijimos que en C existen unos tipos de datos agregados o complejos, las estructuras y las uniones. Con estos dos, Harbour hace la magia.

Todas las variables de Harbour son una estructura con, básicamente, dos miembros, el primero indica el tipo ***HB_TYPE*** que está definido internamente como un entero sin signo:

```
typedef HB_U32 HB_TYPE;
```

Y el segundo es una unión donde están representados todos los tipos que existen en Harbour. Recordad lo que es el tipo de C ***union.***

La estructura es la siguiente:

```
typedef struct _HB_ITEM
{
   HB_TYPE type;
   union
   {
      struct hb_struArray     asArray;
      struct hb_struBlock     asBlock;
      struct hb_struDateTime  asDateTime;
      struct hb_struDouble    asDouble;
      struct hb_struInteger   asInteger;
      struct hb_struLogical   asLogical;
      struct hb_struLong      asLong;
      struct hb_struPointer   asPointer;
      struct hb_struHash      asHash;
      struct hb_struMemvar    asMemvar;
      struct hb_struRefer     asRefer;
      struct hb_struEnum      asEnum;
      struct hb_struExtRef    asExtRef;
      struct hb_struString    asString;
      struct hb_struSymbol    asSymbol;
      struct hb_struRecover   asRecover;
   } item;
} HB_ITEM, *PHB_ITEM;
```

Como comentamos, en las uniones de C todos los miembros ocupan la misma área de memoria. Ahí está la magia…

Entonces podemos afirmar que una variable en PRG equivale en C a una estructura llamada ITEM… el tipo ITEM es la representación de las variables Harbour en C. Estas pueden ser creadas desde PRG o desde el código escrito en C como veremos más adelante.

No vamos a entrar en más profundidad de momento.

Los principales tipos que pueden tener las variables de Harbour son:

- Carácter y Memo (C y M): para tratar cadenas de caracteres.

- Numéricos (N): para tratar cualquier tipo de número con o sin signo y con o sin decimales.

- Lógicos (L): para tratar tipos booleanos o sea “verdad” o “mentira”.

- Fecha (D): para tratar las fechas.

- Array (A): para tratar tablas en memoria, los miembros del array pueden ser de cualquier tipo, recordad que los array de C sólo pueden ser de un tipo determinado.

- Bloque (B): para tratar los CodeBlocks que contienen código ejecutable que puede ser evaluado en cualquier momento.

- Objeto (O): para tratar objetos. Internamente Harbour lo trata como un array.

Esos son los principales, los heredados de Clipper. Pero en Harbour hay más como por ejemplo los **TimeStamp (T)**, los **Hash (H)**, los **Puntero (P)** y los **Símbolo (S).** Jajaja y aún hay más, pero con estos ya está bien.

Aquí tenéis la definición de todos los tipos datos que Harbour usa internamente y que recordad que es el primer miembro de la estructura ITEM:

```
#define HB_IT_NIL       0x00000
#define HB_IT_POINTER   0x00001
#define HB_IT_INTEGER   0x00002
#define HB_IT_HASH      0x00004
#define HB_IT_LONG      0x00008
#define HB_IT_DOUBLE    0x00010
#define HB_IT_DATE      0x00020
#define HB_IT_TIMESTAMP 0x00040
#define HB_IT_LOGICAL   0x00080
#define HB_IT_SYMBOL    0x00100
#define HB_IT_ALIAS     0x00200
#define HB_IT_STRING    0x00400
#define HB_IT_MEMOFLAG  0x00800
#define HB_IT_MEMO      ( HB_IT_MEMOFLAG | HB_IT_STRING )
#define HB_IT_BLOCK     0x01000
#define HB_IT_BYREF     0x02000
#define HB_IT_MEMVAR    0x04000
#define HB_IT_ARRAY     0x08000
#define HB_IT_ENUM      0x10000
#define HB_IT_EXTREF    0x20000
#define HB_IT_DEFAULT   0x40000
#define HB_IT_RECOVER   0x80000
#define HB_IT_OBJECT    HB_IT_ARRAY
#define HB_IT_NUMERIC   ( HB_IT_INTEGER | HB_IT_LONG | HB_IT_DOUBLE )
#define HB_IT_NUMINT    ( HB_IT_INTEGER | HB_IT_LONG )
#define HB_IT_DATETIME  ( HB_IT_DATE | HB_IT_TIMESTAMP )
#define HB_IT_ANY       0xFFFFFFFF
#define HB_IT_COMPLEX   ( HB_IT_BLOCK | HB_IT_ARRAY | HB_IT_HASH | HB_IT_POINTER
| /* HB_IT_MEMVAR | HB_IT_ENUM | HB_IT_EXTREF |*/ HB_IT_BYREF | HB_IT_STRING )
```

```
#define HB_IT_GCITEM    ( HB_IT_BLOCK | HB_IT_ARRAY | HB_IT_HASH | HB_IT_POINTER |
HB_IT_BYREF )
#define HB_IT_EVALITEM  ( HB_IT_BLOCK | HB_IT_SYMBOL )
#define HB_IT_HASHKEY   ( HB_IT_INTEGER | HB_IT_LONG | HB_IT_DOUBLE | HB_IT_DATE |
HB_IT_TIMESTAMP | HB_IT_STRING | HB_IT_POINTER )
```

El Campo o FIELD.

Al mismo nivel que las variables Harbour tiene el ***campo*** o ***FIELD*** con los que podremos manejar los campos de las DBF o mejor dicho de las RDD.

Seguro que os suena esto:

```
FIELD->CodPostal
```

Esta es la manera de decirle a Harbour que esa variables es especial y que la tiene que tratar como un campo, la otra manera es poniendo en vez de FIELD el alias o el área de trabajo.

Los símbolos.

Como diría mi amigo Cristóbal… ahí hay tela que cortar, jajaja, pues sí, hay mucha tela ya que a partir de aquí, vamos a hablar de una cosa tan esencial en Harbour como es la ***Tabla de Símbolos*** y de la ***Tabla Dinámica de Símbolos.***

Pero vamos a empezar por el comienzo, o sea, por la definición de símbolos. Un símbolo es el modo que utiliza la ***Máquina Virtual de Harbour*** para tratar las ***variables (ITEM)***, los ***campos (FIELD)*** y las ***funciones***, sí, sobre todo las funciones…

Cristóbal, se nos acumulan los términos, ahora aparece la **Máquina Virtual de Harbour** (***HVM***), pero bueno no nos vamos a precipitar, eso lo explicaremos después…

Ya hemos dicho que los valores puros de C no son compatibles a nivel PRG, para poder ser tratados desde Harbour tenemos que convertirlos a **ITEM** mediante las funciones del ***ITEM API*** o del S***istema Extendido***. Y las funciones no son tratadas de la misma manera que en C. Harbour necesita saber mas cosas de la función como por ejemplo el nombre como cadena, el ámbito y como no, la dirección donde se aloja en la memoria.

Para ello existe una estructura en C que es la que representa los símbolos.

```
typedef struct _HB_SYMB
{
   const char *szName;          /* the name of the symbol */
   union
   {
      HB_SYMBOLSCOPE value;      /* the scope of the symbol */
      void *pointer;             /* filler to force alignment */
```

```
   } scope;
   union
   {
      PHB_FUNC pFunPtr; /* machine code function address for function symbol table entries */
      PHB_PCODEFUNC  pCodeFunc;     /* PCODE function address */
      void *pStaticsBase;  /* base offset to array of statics */
   } value;
   PHB_DYNS pDynSym;          /* pointer to its dynamic symbol if defined */
} HB_SYMB, *PHB_SYMB;
```

La he puesto para que la veas pero no te asustes porque seguramente no la tendrás que tratar directamente nunca. Lo que sí quiero que entendáis es que es una estructura que contiene unos miembros. Los más importantes son el **Nombre** como cadena, el **Ámbito** y el **Puntero** a la función.

Sabiendo esto podemos decir que la ***Tabla de Símbolos*** es un espacio de memoria donde Harbour guarda los símbolos, o sea es una lista de símbolos. Para poder usar una función tiene que estar en la Tabla de Símbolos. Y por tanto la podemos buscar dentro la tabla de símbolos y sabremos muchas cosa de ella (nombre, ámbito y puntero a la función que representa).

Cuando hacemos referencia a una función desde PRG Harbour crea una entrada en la Tabla Dinámica de Símbolos.

Por otro lado en tiempo de compilación Harbour carga sus funciones internas con un programa en C llamado *initsymb.c*

Lo pongo aquí para que veáis como es una estructura de tipo símbolo, pero sólo para eso. No tener miedo ya que como digo, crear un símbolo manualmente es muy raro que tengamos que hacerlo.

Venga lo pongo ya sin más misterios:

```
#include "hbapi.h"
#include "hbvm.h"

HB_FUNC_EXTERN( AADD );
HB_FUNC_EXTERN( ABS );
HB_FUNC_EXTERN( ASC );
HB_FUNC_EXTERN( AT );
HB_FUNC_EXTERN( BOF );
HB_FUNC_EXTERN( BREAK );
HB_FUNC_EXTERN( CDOW );
HB_FUNC_EXTERN( CHR );
HB_FUNC_EXTERN( CMONTH );
HB_FUNC_EXTERN( COL );
HB_FUNC_EXTERN( CTOD );
HB_FUNC_EXTERN( DATE );
HB_FUNC_EXTERN( DAY );
HB_FUNC_EXTERN( DELETED );
HB_FUNC_EXTERN( DEVPOS );
```

```
HB_FUNC_EXTERN( DOW );
HB_FUNC_EXTERN( DTOC );
HB_FUNC_EXTERN( DTOS );
HB_FUNC_EXTERN( EMPTY );
HB_FUNC_EXTERN( EOF );
HB_FUNC_EXTERN( EXP );
HB_FUNC_EXTERN( FCOUNT );
HB_FUNC_EXTERN( FIELDNAME );
HB_FUNC_EXTERN( FLOCK );
HB_FUNC_EXTERN( FOUND );
HB_FUNC_EXTERN( INKEY );
HB_FUNC_EXTERN( INT );
HB_FUNC_EXTERN( LASTREC );
HB_FUNC_EXTERN( LEFT );
HB_FUNC_EXTERN( LEN );
HB_FUNC_EXTERN( LOCK );
HB_FUNC_EXTERN( LOG );
HB_FUNC_EXTERN( LOWER );
HB_FUNC_EXTERN( LTRIM );
HB_FUNC_EXTERN( MAX );
HB_FUNC_EXTERN( MIN );
HB_FUNC_EXTERN( MONTH );
HB_FUNC_EXTERN( PCOL );
HB_FUNC_EXTERN( PCOUNT );
HB_FUNC_EXTERN( PROW );
HB_FUNC_EXTERN( RECCOUNT );
HB_FUNC_EXTERN( RECNO );
HB_FUNC_EXTERN( REPLICATE );
HB_FUNC_EXTERN( RLOCK );
HB_FUNC_EXTERN( ROUND );
HB_FUNC_EXTERN( ROW );
HB_FUNC_EXTERN( RTRIM );
HB_FUNC_EXTERN( SECONDS );
HB_FUNC_EXTERN( SELECT );
HB_FUNC_EXTERN( SETPOS );
HB_FUNC_EXTERN( SETPOSBS );
HB_FUNC_EXTERN( SPACE );
HB_FUNC_EXTERN( SQRT );
HB_FUNC_EXTERN( STR );
HB_FUNC_EXTERN( SUBSTR );
HB_FUNC_EXTERN( TIME );
HB_FUNC_EXTERN( TRANSFORM );
HB_FUNC_EXTERN( TRIM );
HB_FUNC_EXTERN( TYPE );
HB_FUNC_EXTERN( UPPER );
HB_FUNC_EXTERN( VAL );
HB_FUNC_EXTERN( WORD );
HB_FUNC_EXTERN( YEAR );

static HB_SYMB symbols[] = {
   { "AADD",      { HB_FS_PUBLIC }, { HB_FUNCNAME( AADD )      }, NULL },
```

```
{ "ABS",       { HB_FS_PUBLIC }, { HB_FUNCNAME( ABS )       }, NULL },
{ "ASC",       { HB_FS_PUBLIC }, { HB_FUNCNAME( ASC )       }, NULL },
{ "AT",        { HB_FS_PUBLIC }, { HB_FUNCNAME( AT )        }, NULL },
{ "BOF",       { HB_FS_PUBLIC }, { HB_FUNCNAME( BOF )       }, NULL },
{ "BREAK",     { HB_FS_PUBLIC }, { HB_FUNCNAME( BREAK )     }, NULL },
{ "CDOW",      { HB_FS_PUBLIC }, { HB_FUNCNAME( CDOW )      }, NULL },
{ "CHR",       { HB_FS_PUBLIC }, { HB_FUNCNAME( CHR )       }, NULL },
{ "CMONTH",    { HB_FS_PUBLIC }, { HB_FUNCNAME( CMONTH )    }, NULL },
{ "COL",       { HB_FS_PUBLIC }, { HB_FUNCNAME( COL )       }, NULL },
{ "CTOD",      { HB_FS_PUBLIC }, { HB_FUNCNAME( CTOD )      }, NULL },
{ "DATE",      { HB_FS_PUBLIC }, { HB_FUNCNAME( DATE )      }, NULL },
{ "DAY",       { HB_FS_PUBLIC }, { HB_FUNCNAME( DAY )       }, NULL },
{ "DELETED",   { HB_FS_PUBLIC }, { HB_FUNCNAME( DELETED )   }, NULL },
{ "DEVPOS",    { HB_FS_PUBLIC }, { HB_FUNCNAME( DEVPOS )    }, NULL },
{ "DOW",       { HB_FS_PUBLIC }, { HB_FUNCNAME( DOW )       }, NULL },
{ "DTOC",      { HB_FS_PUBLIC }, { HB_FUNCNAME( DTOC )      }, NULL },
{ "DTOS",      { HB_FS_PUBLIC }, { HB_FUNCNAME( DTOS )      }, NULL },
{ "EMPTY",     { HB_FS_PUBLIC }, { HB_FUNCNAME( EMPTY )     }, NULL },
{ "EOF",       { HB_FS_PUBLIC }, { HB_FUNCNAME( EOF )       }, NULL },
{ "EXP",       { HB_FS_PUBLIC }, { HB_FUNCNAME( EXP )       }, NULL },
{ "FCOUNT",    { HB_FS_PUBLIC }, { HB_FUNCNAME( FCOUNT )    }, NULL },
{ "FIELDNAME", { HB_FS_PUBLIC }, { HB_FUNCNAME( FIELDNAME ) }, NULL },
{ "FLOCK",     { HB_FS_PUBLIC }, { HB_FUNCNAME( FLOCK )     }, NULL },
{ "FOUND",     { HB_FS_PUBLIC }, { HB_FUNCNAME( FOUND )     }, NULL },
{ "INKEY",     { HB_FS_PUBLIC }, { HB_FUNCNAME( INKEY )     }, NULL },
{ "INT",       { HB_FS_PUBLIC }, { HB_FUNCNAME( INT )       }, NULL },
{ "LASTREC",   { HB_FS_PUBLIC }, { HB_FUNCNAME( LASTREC )   }, NULL },
{ "LEFT",      { HB_FS_PUBLIC }, { HB_FUNCNAME( LEFT )      }, NULL },
{ "LEN",       { HB_FS_PUBLIC }, { HB_FUNCNAME( LEN )       }, NULL },
{ "LOCK",      { HB_FS_PUBLIC }, { HB_FUNCNAME( LOCK )      }, NULL },
{ "LOG",       { HB_FS_PUBLIC }, { HB_FUNCNAME( LOG )       }, NULL },
{ "LOWER",     { HB_FS_PUBLIC }, { HB_FUNCNAME( LOWER )     }, NULL },
{ "LTRIM",     { HB_FS_PUBLIC }, { HB_FUNCNAME( LTRIM )     }, NULL },
{ "MAX",       { HB_FS_PUBLIC }, { HB_FUNCNAME( MAX )       }, NULL },
{ "MIN",       { HB_FS_PUBLIC }, { HB_FUNCNAME( MIN )       }, NULL },
{ "MONTH",     { HB_FS_PUBLIC }, { HB_FUNCNAME( MONTH )     }, NULL },
{ "PCOL",      { HB_FS_PUBLIC }, { HB_FUNCNAME( PCOL )      }, NULL },
{ "PCOUNT",    { HB_FS_PUBLIC }, { HB_FUNCNAME( PCOUNT )    }, NULL },
{ "PROW",      { HB_FS_PUBLIC }, { HB_FUNCNAME( PROW )      }, NULL },
{ "RECCOUNT",  { HB_FS_PUBLIC }, { HB_FUNCNAME( RECCOUNT )  }, NULL },
{ "RECNO",     { HB_FS_PUBLIC }, { HB_FUNCNAME( RECNO )     }, NULL },
{ "REPLICATE", { HB_FS_PUBLIC }, { HB_FUNCNAME( REPLICATE ) }, NULL },
{ "RLOCK",     { HB_FS_PUBLIC }, { HB_FUNCNAME( RLOCK )     }, NULL },
{ "ROUND",     { HB_FS_PUBLIC }, { HB_FUNCNAME( ROUND )     }, NULL },
{ "ROW",       { HB_FS_PUBLIC }, { HB_FUNCNAME( ROW )       }, NULL },
{ "RTRIM",     { HB_FS_PUBLIC }, { HB_FUNCNAME( RTRIM )     }, NULL },
{ "SECONDS",   { HB_FS_PUBLIC }, { HB_FUNCNAME( SECONDS )   }, NULL },
{ "SELECT",    { HB_FS_PUBLIC }, { HB_FUNCNAME( SELECT )    }, NULL },
{ "SETPOS",    { HB_FS_PUBLIC }, { HB_FUNCNAME( SETPOS )    }, NULL },
{ "SETPOSBS",  { HB_FS_PUBLIC }, { HB_FUNCNAME( SETPOSBS )  }, NULL },
{ "SPACE",     { HB_FS_PUBLIC }, { HB_FUNCNAME( SPACE )     }, NULL },
```

```
   { "SQRT",      { HB_FS_PUBLIC }, { HB_FUNCNAME( SQRT )      }, NULL },
   { "STR",       { HB_FS_PUBLIC }, { HB_FUNCNAME( STR )       }, NULL },
   { "SUBSTR",    { HB_FS_PUBLIC }, { HB_FUNCNAME( SUBSTR )    }, NULL },
   { "TIME",      { HB_FS_PUBLIC }, { HB_FUNCNAME( TIME )      }, NULL },
   { "TRANSFORM", { HB_FS_PUBLIC }, { HB_FUNCNAME( TRANSFORM ) }, NULL },
   { "TRIM",      { HB_FS_PUBLIC }, { HB_FUNCNAME( TRIM )      }, NULL },
   { "TYPE",      { HB_FS_PUBLIC }, { HB_FUNCNAME( TYPE )      }, NULL },
   { "UPPER",     { HB_FS_PUBLIC }, { HB_FUNCNAME( UPPER )     }, NULL },
   { "VAL",       { HB_FS_PUBLIC }, { HB_FUNCNAME( VAL )       }, NULL },
   { "WORD",      { HB_FS_PUBLIC }, { HB_FUNCNAME( WORD )      }, NULL },
   { "YEAR",      { HB_FS_PUBLIC }, { HB_FUNCNAME( YEAR )      }, NULL }
};

/* NOTE: The system symbol table with runtime functions HAVE TO be called
         last */

void hb_vmSymbolInit_RT( void )
{
   HB_TRACE( HB_TR_DEBUG, ( "hb_vmSymbolInit_RT()" ) );
   hb_vmProcessSymbols( symbols, HB_SIZEOFARRAY( symbols ), "", 0, 0 );
}
```

Como veis la ***Tabla de Símbolos de Harbour*** es un array que contiene símbolos que son las funciones estándar de Harbour que usamos en nuestros programas. Y por tanto, y esto es lo importante, podemos buscar en el array por el nombre y así encontrar la posición o acceder directamente por la posición.

Un ejemplo de un símbolo es esto:

```
{ "YEAR", { HB_FS_PUBLIC }, { HB_FUNCNAME( YEAR ) }, NULL }
```

El **nombre** como cadena, el **ámbito** en este caso público y el **puntero** a la función.

¿Sabéis que tenemos a nuestra disposición a nivel PRG esos símbolos de la tabla?

¿Y que hay una clase para manejarlos?

Se pueden hacer cosas tan chulas como esta:

```
PROCEDURE Main()
   LOCAL oSym := Symbol():New( "QOUT" )

   ? "Now test the :Exec() method"

   oSym:Exec( "This string is being printed by QOUT" )
   oSym:Exec( "which is being invoked by the :Exec()" )
   oSym:Exec( "method in the Symbol class." )
   ?
   ? "symbol name: ", oSym:name
```

```
   ? "Comparing QOut symbol with xOut symbol"
   ? oSym:IsEqual( Symbol():New( "xOut" ) )
   ? "done!"
   ?
   RETURN
```

O podríamos usar esa idea pero sin Clase Symbol, directamente:

```
__dynsN2Sym( "Alert" ):exec( "Hola mundo" )
```

Con esta idea podríamos pasar a una función una cadena con la función que queramos ejecutar en la función de destino y así evitar el uso de codeblock, por ejemplo.

Esto tiene muchísimas utilidades para nosotros. Por ejemplo usar técnicas Data Driven en nuestros programas. Evitar muchas condiciones “IF”, etc. Como diría alguien “El límite lo pone tu mente”.

Pues ahora toca saber algo sobre la Máquina Virtual de Harbour (HVM), pero antes vamos a introducir otro concepto:

Qué es el pCode:

Ya vimos que había un único lenguaje que entendían las máquinas, adivinas cual? Sí, acertaste es el ***código máquina***.

El código máquina está estrechamente ligado a la máquina que lo va a ejecutar. Eso significa que tendremos que re-compilar nuestro código para cada máquina de tipo diferente. Para salvar este contratiempo se crearon los lenguajes interpretados como Java o Harbour.

Es cierto que Harbour no genera código máquina pero a cambio nos da la opción de generar un código binario portable que se podría ejecutar en cualquier máquina sin tener que volverlo a compilar. No lo solemos usar pero existe. Es el ***.hrb**. Si alguien tiene un sistema operativo Windows y otro Linux podéis hacer la prueba.

```
/* demo.prg */

procedure main
      cls
      ? “Hola mundo…”
      inkey( 0 )
return
```

Usando Hbmk2

```
hbmk2 -gh demo.prg
```

Esto genera un archivo "*demo.hrb*"

Para ejecutarlo en cualquier sistema operativo sin volverlo a compilar sólo tienes que hacer:

```
hbmk2 demo.hrb
```

Eso funciona en cualquier sitio para el que haya una Máquina Virtual Harbour. Funciona como Java pero en nuestro lenguaje. Además muy potente y listo para usar en Mod Harbour o sea en Apache… y hacer que nuestras aplicaciones funcionen en la web de una manera transparente…

Dicho lo cual…

Quieres ver pCode? Es muy fácil. Compila el ejemplo de arriba de esta manera:

```
harbour demo -gc2
```

Ahora mira en el directorio donde lo has hecho y verás que hay un archivo llamado demo.c con el siguiente contenido:

```
/*
 * Harbour 3.2.0dev (r2101261627)
 * Microsoft Visual C++ 19.28.29337 (64-bit)
 * Generated C source from "demo.prg"
 */

#include "hbvmpub.h"
#include "hbpcode.h"
#include "hbinit.h"

HB_FUNC( DEMO );
HB_FUNC( MAIN );
HB_FUNC_EXTERN( SCROLL );
HB_FUNC_EXTERN( SETPOS );
HB_FUNC_EXTERN( QOUT );
HB_FUNC_EXTERN( INKEY );

HB_INIT_SYMBOLS_BEGIN( hb_vm_SymbolInit_DEMO )
{ "DEMO", {HB_FS_PUBLIC | HB_FS_FIRST | HB_FS_LOCAL}, {HB_FUNCNAME( DEMO )},
NULL },
{ "MAIN", {HB_FS_PUBLIC | HB_FS_LOCAL}, {HB_FUNCNAME( MAIN )}, NULL },
{ "SCROLL", {HB_FS_PUBLIC}, {HB_FUNCNAME( SCROLL )}, NULL },
{ "SETPOS", {HB_FS_PUBLIC}, {HB_FUNCNAME( SETPOS )}, NULL },
{ "QOUT", {HB_FS_PUBLIC}, {HB_FUNCNAME( QOUT )}, NULL },
{ "INKEY", {HB_FS_PUBLIC}, {HB_FUNCNAME( INKEY )}, NULL }
HB_INIT_SYMBOLS_EX_END( hb_vm_SymbolInit_DEMO, "demo.prg", 0x0, 0x0003 )
```

```
#if defined( HB_PRAGMA_STARTUP )
   #pragma startup hb_vm_SymbolInit_DEMO
#elif defined( HB_DATASEG_STARTUP )
   #define HB_DATASEG_BODY    HB_DATASEG_FUNC( hb_vm_SymbolInit_DEMO )
   #include "hbiniseg.h"
#endif

HB_FUNC( DEMO )
{
   static const HB_BYTE pcode[] =
   {
      HB_P_ENDPROC
/* 00001 */
   };

   hb_vmExecute( pcode, symbols );
}

HB_FUNC( MAIN )
{
   static const HB_BYTE pcode[] =
   {
/* 00000 */ HB_P_LINE, 4, 0,   /* 4 */
      HB_P_PUSHFUNCSYM, 2, 0,  /* SCROLL */
      HB_P_DOSHORT, 0,
      HB_P_PUSHFUNCSYM, 3, 0,  /* SETPOS */
      HB_P_ZERO,
      HB_P_ZERO,
      HB_P_DOSHORT, 2,
/* 00015 */ HB_P_LINE, 6, 0,   /* 6 */
      HB_P_PUSHFUNCSYM, 4, 0,  /* QOUT */
      HB_P_PUSHSTRSHORT, 14,   /* 14 */
      'H', 'o', 'l', 'a', ' ', 'm', 'u', 'n', 'd', 'o', '.', '.', '.', 0,
      HB_P_DOSHORT, 1,
/* 00039 */ HB_P_LINE, 8, 0,   /* 8 */
      HB_P_PUSHFUNCSYM, 5, 0,  /* INKEY */
      HB_P_ZERO,
      HB_P_DOSHORT, 1,
/* 00048 */ HB_P_LINE, 10, 0,  /* 10 */
      HB_P_ENDPROC
/* 00052 */
   };
```

```
   hb_vmExecute( pcode, symbols );
}
```

Lo que hay en azul es el pCode, en rojo la ejecución del pCode y el principio en verde es la definición de los símbolos y su inserción en la tabla virtual de símbolos.

Qué es la Máquina Virtual de Harbour:

Ya está explicado con lo visto, pero podríamos decir que es la manera que tiene Harbour de hacer que la máquina donde se ejecuta nuestro programa entienda el pCode o lo que es lo mismo es un traductor de pCode a código máquina.

Hay que decir que la mayoría de nuestros programas son ejecutable que contienen el pCode y la máquina virtual.

Ahora tenemos que cerrar el círculo de este tema teórico.

Qué es la pila o stack:

En términos generales una pila es una lista ordenada donde se guardan y recuperan datos. Usa el método **LIFO** esto es *Last In, First Out*, en español: último en entrar, primero en salir. Y esto es aplicable a la pila o stack de Harbour. El elemento que ocupa lo más alto de la lista se llama ***cima***.

Hay tres funciones aplicables al stack:

- La creación crea la pila

- Apilar → ***push:*** añade un nuevo elemento a la pila que toma el foco como ***cima.***

-Des-apilar → ***pop:*** extrae el elemento que ocupa la ***cima*** disminuyendo en uno los elementos.

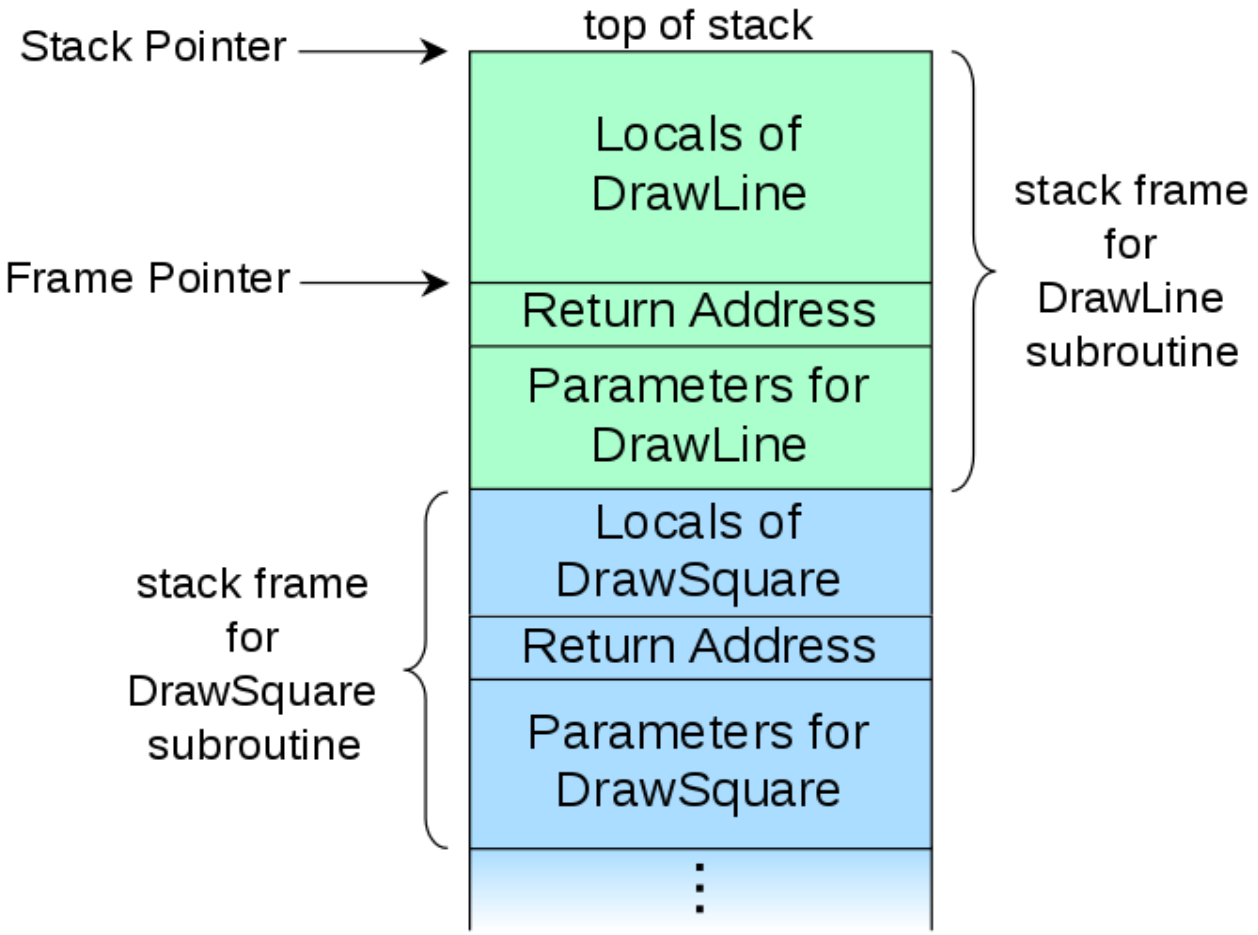
Un ejemplo real de una pila es un montón de platos apilados uno encima de otro.

Para este curso es muy importante el ***stack*** ya que en eso se basa el ***Sistema Extendido*** y el ***ITEM API.*** En siguiente tema veremos este tema más profundamente.

Podemos decir entonces que el stack es el instrumento que usa la ***HVM*** para poder ejecutar las funciones y los valores que tiene que procesar. La HVM crea una pila en la que introduce valores y funciones, aumenta la Pila. Cuando se ejecutan las funciones la pila disminuye.

Cuando aumenta la pila hasta tal punto que se sobrepasa su dimensión, la pila se desborda y el programa suelta un bonito **“Stack Overflow”**, gracias a que Harbour ha aumentado tanto la dimensión de la pila que es muy difícil ver este error, pero los viejos clipperos…

Esta imagen es de wikipedia y creo que ilustra gráficamente muy bien una pila:



Es importante la zona de la ***dirección de retorno.*** Hay una función interna del API de Harbour para acceder a ese **ITEM**, se llama `hb_stackReturnItem()`. *Quédate con la idea ya que será muy útil para usarla como variable de trabajo.*

Otra cosa importante es que la variables que cuyo ámbito de vida es durante toda la vida de la ejecución del programa como las estáticas o las pública se guardan en una zona permanente de memoria.

En resumen:

Harbour es un lenguaje interpretado que usa una máquina virtual para traducir el *pCode* a código máquina que es el lenguaje que entiende la máquina donde se ejecuta el programa.

La máquina virtual crea una pila en la que se van introduciendo los *símbolos* de la funciones que se van a ejecutar y los *ITEMs* que se van a procesar por esas funciones ya sean parámetros o variables.

En la pila se usa el símbolo de una función y no la propia función. Un símbolo es una estructura que contiene un miembro de tipo cadena con el nombre de la función, otro es el ámbito y otro el puntero a la función. Esos símbolos se guardan en una zona de memoria llamada *Tabla de símbolos* que hace de lista de funciones disponibles, principalmente.

13. Creación de funciones en C para ser usadas desde programas PRG de Harbour.

Como he dicho las funciones hechas en Lenguaje C para ser utilizadas desde los programas PRG de Harbour tiene que tener unas características. Vamos a explicarlas…

Todas las funciones que se ejecutan en Harbour no devuelven nada ni reciben nada, además son de tipo PASCAL. Ya hemos dicho que usan el stack para tratar los parámetros y el valor de retorno.

Este sería el esqueleto de una función prototipo:

```
void MIFUNCION( void )
{
      ...
      // Cuerpo de la función
      ...
}
```

Pero seguramente esto no os suena mucho… tal vez esto si:

```
HB_FUNC( MIFUNCION )
{
      ...
      // Cuerpo de la función
      ...
}
```

Esto:

```
HB_FUNC( MIFUNCION )
```

en vez de esto:

```
void MIFUNCION( void )
```

Es más fácil la primera ¿no?

Observa que el nombre está en mayúsculas y que a efectos prácticos no estamos limitados a 10 caracteres como en el legendario Clipper. Esto hace que podamos crear nombre de funciones autodefinidos.

Recuerda que C sólo trata con tipos de datos de C y que para poder procesar los datos de Harbour (ITEM) tendremos que extraer su valor interno expresado en un tipo de C. Una vez hecho esto podemos usar las funciones de C que tiene en su biblioteca estándar. Quiero recordar que en la

medida de lo posible deberíamos usar las funciones internas del API C de Harbour. Esto lo veremos en los modelos, pero por ejemplo:

hb_xstrcat() en vez de *strcat()*

hb_xstrcpy() en vez de strcpy()

Normalmente la función interna es más potente y se adapta a las necesidades de Harbour como por ejemplo los ***multihilos***. Existen funciones interna para casi todo y podremos usarlas sin problemas.

Una vez terminado el procesado hay que devolver esos datos de tipos de C a los ITEM para ser tratados desde PRG.

Para ello está el Sistema Extendido y el API ITEM que es el motivo de este libro y que explicaremos más adelante.

Lo dejamos aquí…

14. Cómo compilar código C en nuestros PRG

Bueno ya estamos a las puertas de lo que realmente nos gusta que es PROGRAMAR, pero antes hay que saber como se compilan esas funciones en C.

Realmente podríamos usar directamente el compilador de C con el que trabajemos. Pero cada compilador tiene sus propias opciones. ¿Cómo podemos evitar el tener que aprender estas opciones? ¿Cómo facilitar el proceso de construcción de EXE, LIB o simplemente un OBJ? Si tu respuesta ha sido: “Usar una herramienta MAKE” has acertado…

Y aquí aparece la súper herramienta make que nos brinda Harbour ***HbMk2***

Sintaxis:

hbmk2 [opciones] [<archivo[s]deordenes>] <fuente[s]
[.prg|.c|.obj|.o|.rc|.res|.def|.po|.pot|.hbl|@.clp|.d|.ch]>

Descripción:

hbmk2 es una herramienta integrada y portable de generación o automatización de código, haciendo posible la creación de varios tipos de ejecutables binarios (ejecutable, biblioteca dinámica, biblioteca estática, binario portátil de Harbour) desde múltiples tipos de archivos de código fuente (C, C++, Objective-C, Harbour, traducciones de 'gettext', recursos de Windows). 'Integrada' significa que un solo archivo de proyecto hbmk2 puede controlar todos, o casi todos, los aspectos del proceso de construcción. 'Portable' significa que un solo archivo de proyecto hbmk2 puede controlar la construcción del ejecutable binario en todas las plataformas de los sistemas operativos soportados y a través de todos los compiladores de C soportados. Ayuda también en la mayoría de los procesos de construcción por medio de pequeños y simples archivos de proyecto (opciones). hbmk2 soporta archivos de proyecto para C/C++/Objetive-C sin relación con Harbour. Para conseguir esos objetivos, hbmk2 detecta automáticamente a Harbour, al compilador de C y a las demás herramientas necesarias, las configura y luego las ejecuta convenientemente. hbmk2 permite extender los tipos de código fuente soportados por medio de complementos.
Además de construir ejecutables, hbmk2 puede ejecutar archivos de órdenes de Harbour (tanto en código fuente como precompilado) directamente, y también dispone de un intérprete de comandos interactivo.

Y ahora voy a poner mi propuesta de procesamiento por lotes genérico. Y que cada uno debe adaptar a su entorno. Vamos a asumir que los ejemplos de funciones en C las vamos a guardar en un archivo llamado “cursode.c” y el PRG tendrá cada uno su nombre diferente. Al final con “cursode.c” podríamos crear una LIB. Pero de momento la enlazaremos siempre como fuente, pero no te preocupes por los tiempos porque HbMk2 es muy inteligente y hace una compilación incremental que significa que si no ha habido variaciones no lo vuelve a compilar.

La estructura del directorio podría ser:

curso_c

ejemplos

fuentes_c

En ejemplos estarán metidos todos los PRG y en fuentes_c estará cursode.c

El **BAT** lo vamos a llamar ***do_xxxxx.bat*** siendo **xxxxx** el compilador que vamos a usar, por ejemplo ***do_mingw64.bat***.

```
@set comp=MiCompiladorC
@set DIR_HBBIN=MiDirDelBINdeHarbour
@set DIR_CCBIN=MiDirDelBINLengC
@rem esta dos no hay que cambiarlas
@set PATH=%DIR_HBBIN%;%DIR_CCBIN%;%PATH%
@hbmk2 -comp=%comp% %1 fuentes_c\cursode.c
```

Siendo:

MiCompiladorC →

- de 32 bits: mingw, msvc, clang, bcc, watcom, icc, pocc, xcc

- de 64 bits: bcc64, mingw64, msvc64, msvcia64, iccia64, pocc64

Para msvc y msvc64 hay que añadirle una línea que activa el entorno de dicho compilador.

MiDirDelBINdeHarbour → El directorio BIN de tu instalación Harbour.

MiDirDelBINLengC → El directorio BIN de tu instalación del compilador de Lenguaje C.

Lógicamente las LIBs de Harbour se tienen que corresponder con la del compilado de C que se use.

Vamos a poner el ejemplo para mingw de 64 bits y los dos de msvc el de 32 y 64 bits

do_mingw64.bat

```
@set comp=mingw64
```

```
@set DIR_HBBIN=u:\desarrollo\comp\xc\hb\bin

@set DIR_CCBIN=u:\desarrollo\comp\cc\mingw\64\9.30\bin

@set PATH=%DIR_HBBIN%;%DIR_CCBIN%;%PATH%
@hbmk2 -comp=%comp% %1 fuentes_c\cursode.c
```

do_msvc32.bat

```
@set comp=msvc

@set DIR_HBBIN=u:\desarrollo\comp\xc\hb\bin

@call "%ProgramFiles(x86)%/Microsoft Visual Studio/2019/Community/VC/Auxiliary/Build/vcvars32.bat"

@set PATH=%DIR_HBBIN%;%DIR_CCBIN%;%PATH%
@hbmk2 -comp=%comp% %1 fuentes_c\cursode.c
```

do_msvc64.bat

```
@set comp=msvc

@set DIR_HBBIN=u:\desarrollo\comp\xc\hb\bin

@call "%ProgramFiles(x86)%/Microsoft Visual Studio/2019/Community/VC/Auxiliary/Build/vcvars64.bat"

@set PATH=%DIR_HBBIN%;%DIR_CCBIN%;%PATH%
@hbmk2 -comp=%comp% %1 fuentes_c\cursode.c
```

Estos archivos de procesamiento por lotes son muy simples y lo iremos perfeccionando por ejemplo con control de errores, pero de momento lo dejaremos así…

15. El Sistema Extendido de Harbour

Una cosa muy importante que pueden tener los lenguajes de programación es la integración con el omnipotente Lenguaje C. Sin ir más lejos los programadores del Lenguaje Java cuenta con el Java Native Interface (JNI). Esto dota a esos lenguajes de una capacidad de expansión ilimitada…

Pues bien, Harbour no es menos y por falta de uno, nosotros tenemos a nuestra disposición dos maneras de integrar C en nuestras aplicaciones:

- El ***Sistema Extendido***

- El ***ITEM API***

Incluso podríamos mezclar ambas tecnologías. De momento en este tema vamos a tratar el primero.

El abuelo Clipper desde siempre ha implementado el *Sistema Extendido* y a partir de las versiones 5.xx también introdujo el *ITEM API*.

Como es bien conocido **Harbour** es una versión moderna y mejorada del viejo **Clipper** y por tanto lo ha heredado todo y lo ha mejorado en todos los aspectos. Por tanto el *Sistema Extendido* también está incluido en esa herencia. Se puede decir que es la manera tradicional de integrar funciones en C en nuestros PRG.

Como dijimos en temas anteriores, las funciones hechas en C para ser usadas en PRG tienen que cumplir unas condiciones, no pueden devolver nada y no pueden tener parámetros o sea la función sería así:

```
void MIFUNCION( void )
{
      // Cuerpo de la función
}
```

Pero Harbour nos simplifica un poco la definición y podemos usar esta otra:

```
HB_FUNC( MIFUNCION )
{
      // Cuerpo de la función
}
```

*S*iendo “`HB_FUNC( MIFUNCION )`” una manera más fácil de poner “`void MIFUNCION( void )`”

Entonces si, como vemos, estas funciones ni devuelve nada ni reciben parámetros, ¿como es que desde un PRG podemos pasar parámetros y recibir valores? Ahí está mi interés en explicar la ***VMH*** los ***símbolos*** y la ***pila*** o ***stack***.

Esas funciones usan la pila para pasar parámetros y la zona de devolución o de retorno (stackReturn) para devolver el valor.

Cada vez que desde un PRG ejecutamos una función, lo que hace la VMH es poner (push) en la pila todos los parámetros, este sería el esquema:

PUSH “El_Simbolo_de_la_Función_o_Método”

PUSCH “NIL para las funciones o el símbolo del objeto”

PUSCH “parametro1”

PUSCH “parametro2”

…

PUSCH “parametroN”

DO(Num_parametros) // Para ejecutar y hacer un pop de la pila.

DO puede ser una de estas tres:

- hb_vmDo(nPar) para las funciones procedimientos y métodos.

- hb_vmProc(nPar) para los procedimientos y funciones.

- hb_vmSend(nPar) para los métodos de los objetos.

Hay más, pero esos tres son los más importantes. De momento quédate con ese esquema, en el próximo tema donde se explicará cómo ejecutar funciones PRG en C lo veremos con más detenimiento.

Pues bien, las funciones del Sistema Extendido toman los parámetros que fueron introducidos en la pila por su numero de posición y ponen la variable de retorno en la zona retorno de la pila, así se hace la magia...

Por tanto, y a partir de esta afirmación podemos decir que hay dos tipos de funciones en el Sistema Extendido:

1.- Las que rescatan por su posición los parámetros introducidos en la pila.

2.- Las que actualizan el zona de retorno o de devolución de la pila.

Las primeras son las ***hb_par…()*** como por ejemplo, ***hb_parc()***, ***hb_parni()***, etc, las segundas son las ***hb_ret...()*** como por ejemplo, ***hb_retc()***, ***hb_retni()***, etc y además de estas dos hay las que tratan las variables pasada por referencia, son las ***hb_stor...()*** como por ejemplo ***hb_storc()*** o ***hb_storni()***.

Paso de parámetros.

Estas son las principales funciones encargadas de rescatar los parámetros pasados:

```
const char *hb_parc( int iParam );
const char *hb_parcx( int iParam );
HB_SIZE hb_parclen( int iParam );
HB_SIZE hb_parcsiz( int iParam );
const char *hb_pards( int iParam );
long hb_pardl( int iParam );
double hb_partd( int iParam );
int hb_parl( int iParam );
int hb_parldef( int iParam, int iDefValue );
double hb_parnd( int iParam );
int hb_parni( int iParam );
int hb_parnidef( int iParam, int iDefValue );
long hb_parnl( int iParam );
long hb_parnldef( int iParam, long lDefValue );
HB_MAXINT hb_parnint( int iParam );
HB_MAXINT hb_parnintdef( int iParam, HB_MAXINT nDefValue );
void *hb_parptr( int iParam );
```

Hay alguna más. Las que están en negrita son las que estaban en las versiones de Clipper 5.xx, el resto son extensiones introducidas por Harbour. Inicialmente vamos a explicar las de toda la vida y las más útiles de las nuevas.

Para no tener que explicarlas una a una voy a decir lo que significa el tipo de dato que hay delante de cada función (delante de *hb_par...()*) o lo que es lo mismo, lo que devuelve, que como se puede ver son tipos de datos en C:

***const char ** → una cadena constante

HB_SIZE → (long long) un entero largo largo

HB_ISIZ → (long long) un entero largo largo

long → (long) un entero largo

double → (double) un número real de doble precisión

HB_MAXINT → (long) o (long long) el máximo tamaño de entero largo disponible

***void ** → puntero sin tipo

Y la leyenda de los parámetros entre ( ):

int iParam → posición del parámetro como un int

DefValue → valor por defecto si no se le pasó el valor desde PRG

Muy importante que se entienda lo explicado.

Por ejemplo si usamos una función hecha en C en nuestro PRG

```
procedure main

      local nEdad, lMayorDeEdad
      nEdad := 57
       lMayorDeEdad := miMayorEdad( nEdad )
      alert( “Es “ + iif(  lMayorDeEdad, “mayor”, “menor” ) + “ de edad” )

return
```

En la función en C “`miMayorEdad`“ tenemos que rescatar el parámetro así:

```
int hb_parni( 1 ); // Recupera el parámetro 1 como un entero
```

Devolución de valores desde C.

La funciones para este cometido podríamos decir que son simétricas a la de los parámetros, quiero decir que pueden aceptar cualquier tipo de datos como los devueltos por las funciones del tipo *hb_par...()*, solo que a las funciones de tipo ***hb_ret...()*** hay que pasárselos como parámetro, estas son las principales:

```
void hb_ret( void );
void hb_retc( const char *szText );
void hb_retclen( const char *szText, HB_SIZE nLen );
void hb_retc_null( void );
void hb_retc_buffer( char *szText );
void hb_retclen_buffer( char *szText, HB_SIZE nLen );
void hb_retds( const char *szDate );
void hb_retd( int iYear, int iMonth, int iDay );
void hb_retl( int iTrueFalse );
void hb_retnd( double dNumber );
void hb_retni( int iNumber );
void hb_retnl( long lNumber );
void hb_retnint( HB_MAXINT nNumber );
void hb_reta( HB_SIZE nLen );
void hb_retptr( void * ptr );
```

Observa que he puesto en negrita las funciones que soporta Clipper, el resto es una extensión de Harbour.

Y estos son los parámetros que se les puede pasar para convertirlo en valores devueltos y que entienda Harbour desde PRG:

***const char ** →** una cadena constante

***char ** →** una cadena

***HB_SIZE* →** (long long) un entero largo largo

***HB_ISIZ* →** (long long) un entero largo largo

***int* →** (int) un entero

***long* →** (long) un entero largo

***double* →** (double) un real doble

***void* →** sin parámetros

***HB_MAXINT* →** (long) o (long long) el máximo tamaño de entero largo disponible.

Y es interesante observar en la terminación tanto de las funciones *hb_par...()* como *hb_ret...():*

***c* →** para cadenas *(hb_par**c**, hb_ret**c**)*

n* →** para los números seguido de tipo ***i de entero o ***l*** de largo *hb_parni, hb_parnl*

***l* →** para los tipos lógicos

***d* →** para fechas

***a* →** para arrays

En fin, como se puede observar son muy auto-descriptivas.

hb_retc_buffer() y ***hb_retclen_buffer()*** funcionan exactamente igual que ***hb_retc()*** y ***hb_retclen()*** respectivamente pero con la particularidad que liberan la memoria reservada previamente.

Si alguien no ha entendido algo que levante la mano…

Dicho esto ya estamos en disposición de escribir nuestras funciones en C para ser utilizadas en PRG.

Vamos a cambiar nuestro archivo por lotes para que controle si hay errores y nos avise de ello. Y si no los hay que ejecute el programa generado.

Pongo el del compilador de C MinGW de 64 bits (do_mingw64.bat).

Realmente es fácil de modificar, sólo hay que añadirle las ultimas lineas...

```
@set comp=mingw64
@set DIR_HBBIN=u:\desarrollo\comp\xc\hb\bin
@set DIR_CCBIN=u:\desarrollo\comp\cc\mingw\64\9.30\bin
@set PATH=%DIR_HBBIN%;%DIR_CCBIN%;%PATH%
@hbmk2 -comp=%comp% %1 fuentes_c\cursode.c
@if %errorlevel% neq 0 goto bld_error
@cls
%1
goto fin_exec
:bld_error
@echo ----------------------------------
@echo    Hay errores en la compilacion
@echo ----------------------------------
pause
:fin_exec
```

Vamos a empezar con los ejemplos.

Este es el primer PRG que llama a funciones hechas en C:

```
//-----------------------------------------------------------------------------
// Ejercicio uso de funciones C
// ej001.prg
//-----------------------------------------------------------------------------

procedure main()

    local getlist := {}
    local nYear := Year( Date() )
    local nCubo := 0
    local cNum, i, nDia := 0, cDia
    local cCadena := Space( 60 )
    local cCaracter := " "
    local n1 := 0, n2 := 0, nS, nR, nP, nD
    local nNota := 0, cEstado
    local aNum := { 10, 20, 30 }

    cls

        separa( .f. )

    @ 02, 01 SAY "Curso de C para Harbour"

    separa( .f. )

    @ 05, 01 SAY "Introduce un año para saber si es bisiesto........:" GET nYear PICTURE "@K 9999"
    @ 06, 01 SAY "Introduce un numero para calcular el cubo.........:" GET nCubo PICTURE "@K 9999"
    @ 07, 01 SAY "Introduce una cadena..............................:"
    @ 08, 10 GET cCadena PICTURE "@K"
    @ 09, 01 SAY "Introduce un caracter para contarlos en la cedena.:" GET cCaracter PICTURE "@K"
    @ 10, 01 SAY "Introduce un numero para tabla de multiplicar.....:" GET n1 PICTURE "@K 99"
    @ 11, 01 SAY "Introduce un numero para hacer calculos...........:" GET n2 PICTURE "@K 99"
    @ 12, 01 SAY "Introduce una nota de calificacion................:" GET nNota PICTURE "@K 99"
    @ 13, 01 SAY "Dame un numero del 1 - 7 para calcular dia semana.:" GET nDia PICTURE "@k 9"

        separa( .f. )

    READ
```

```
cls

    separa( .f. )

// Prueba de llamadas a funciones en C
? "El año " + hb_ntoc( nYear )
if mi_isLeap( nYear )
    ?? " es bisiesto"
else
    ?? " no es bisiesto"
endif

separa( .f. )

cNum := hb_ntoc( nCubo ) // El numero como cadena

// No devuelve nada sino que usa la misma variable para depositar el cubo
mi_cubo( @nCubo )

? "El cubo de " + cNum + " es " + hb_ntoc( nCubo )

separa( .f. )

cCadena := AllTrim( cCadena )
n := scanChar( cCadena, cCaracter )
if n > 0
    ? [En la cadena: "] + cCadena + ["]
    ? "La priemara aparicion del caracter '" + cCaracter + "' esta en la posicion " + HB_NToS( n )
    n := charCount( cCadena, cCaracter )
    ? "y se repite " + HB_NToS( n ) + " veces"
else
    ? "No hay '" + cCaracter + "' en la cadena: " + cCadena
endif

separa( .f. )

calcula( n1, n2, @nS, @nR, @nP, @nD ) // Ojo: los 4 ultimos pasado por referencia con "@"

// Ojo: Los parametros de calcula() se pasan por referencia
? "Suma.......: " + HB_NToS( n1 ) + " + " + HB_NToS( n2 ) + " = " + HB_NToS( nS )
? "Resta......: " + HB_NToS( n1 ) + " - " + HB_NToS( n2 ) + " = " + HB_NToS( nR )
? "Producto...: " + HB_NToS( n1 ) + " X " + HB_NToS( n2 ) + " = " + HB_NToS( nP )
? "Division...: " + HB_NToS( n1 ) + " / " + HB_NToS( n2 ) + " = " + HB_NToS( nD )

separa( .f. )

esActo( nNota, @cEstado )

? "Esta usted " + cEstado

separa( .t. )

// Le vamos a añadir 3 elementos al array para comprobar que sumaArray() solo sumas
// los numericos con o sin decimales
aAdd( aNum, "Hola" )
aAdd( aNum, 25.75 )
aAdd( aNum, 1000 )

cambiaValor( aNum )
for i := 1 to len( aNum )
    ? "Elemento", i, aNum[ i ]
next

? "La suma de los elementos del array es:", sumaArray( aNum )

separa( .f. )

? "Tabla de multiplicar del " + HB_NToS( n1 )

aTabla := tabla( n1 )

for i := 1 to len( aTabla )
```

```
        ? HB_NToS( n1 ) + " X " + HB_NToS( i ) + " = " + HB_NToS( aTabla[ i ] )
   next

    separa( .t. )

    cDia := diaSemana( nDia )

    if !Empty( cDia )
        ? "El dia es: " + cDia
    else
        ? "Error en dato introducido: solo numeros del 1 al 7..."
    endif

    separa( .t. )

return

// Escribe un separador en la pantalla

static procedure separa( lPara )

    ? Replicate( "-", 70 )

    if ValType( lPara ) == 'L' .and. lPara
        ? "Presiona una tecla para seguir..."
        Inkey( 100 )
    endif

return
```

Y ahora las funciones en C

```
/*
 * Devuelve si un año es bisiesto o no
 */
HB_FUNC( MI_ISLEAP )
{
        HB_UINT uiYear = hb_parni( 1 ); // Recupera un entero pasado a la funcion desde PRG
        HB_BOOL lRet = HB_FALSE;

        if( uiYear > 0 )
        {
                lRet = ( ( uiYear % 4 == 0 && uiYear % 100 != 0 ) || uiYear % 400 == 0 );
        }

        hb_retl( lRet ); // Devuelve un valor logico por medio de la pila de harbour
}

/*
 * El cubo de un numero pasado por referencia
 */
HB_FUNC( MI_CUBO )
{
        HB_MAXINT iCubo = hb_parnint( 1 );

        iCubo = iCubo * iCubo * iCubo;

        hb_stornint( iCubo, 1 );
}

/*
 * Cuenta las ocurrencias de un carácter dentro de una cadena
 */
HB_FUNC( CHARCOUNT )
{
        const char *szCadena = hb_parc( 1 ); // 1 parametro como cadena
        HB_SIZE uiLen = hb_parclen( 1 ); // Tamaño de la 1 cadena pasada
        HB_UINT uiContador = 0;

        if( uiLen > 0 )
```

```
        {
                const char *cCarcter = hb_parc( 2 ); // 2 parametro como cadena
                HB_SIZE i;

                for( i = 0; i < uiLen; i++ )
                {
                        if( szCadena[ i ] == cCarcter[ 0 ] )
                        {
                                ++uiContador;
                        }
                }
        }

        hb_retnint( uiContador );
}
```

Y ahora otra función que encuentra en una cadena pasada en primer lugar, la primera aparición del carácter pasado en segundo lugar:

```
/*
 * Busca la primera aparicion de un caracter dentro de una cadena
 */
HB_FUNC( SCANCHAR )
{
        const char *szCadena = hb_parc( 1 );
        HB_SIZE uiLen = hb_parclen( 1 );
        HB_SIZE uiPos = 0;

        if( uiLen > 0 )
        {
                const char *cCarcter = hb_parc( 2 );
                HB_SIZE i;

                for( i = 0; i < uiLen; i++ )
                {
                        if( szCadena[ i ] == cCarcter[ 0 ] )
                        {
                                uiPos = i + 1; // Se suma 1 ya que los arrays de C se basan en 0
                                break;
                        }
                }
        }

        hb_retnint( uiPos );
}
```

En los ejemplos anteriores se han usado las funciones para recibir parámetros de tipo cadena. De igualmente se haría con parámetros de tipo numérico:

```
/*
 * Devuelve si un año es bisiesto o no
 */
HB_FUNC( MI_ISLEAP )
{
    HB_UINT uiYear = hb_parni( 1 ); // Recupera un entero pasado a la funcion desde PRG
    HB_BOOL lRet = HB_FALSE;

    if( uiYear > 0 )
    {
        lRet = ( ( uiYear % 4 == 0 && uiYear % 100 != 0 ) || uiYear % 400 == 0 );
    }
```

```
    hb_retl( lRet ); // Devuelve un valor logico por medio de la pila de harbour
}
```

Más:

```
/*
 * El cubo de un numero pasado por referencia
 */
HB_FUNC( MI_CUBO )
{
    HB_MAXINT iCubo = hb_parnint( 1 );

    iCubo = iCubo * iCubo * iCubo;

    hb_stornint( iCubo, 1 );
}


/*
 * Comprueba si una nota está aprobada o no y devuelve un literal con el resultado
 */
HB_FUNC( ESACTO )
{
        unsigned int uiNota = hb_parnidef( 1, 3 ); // Si no se pasa la variable pone por defecto "3", nota
minima

        if( uiNota > 4 )
        {
                hb_storc( "APROBADO", 2 );
        }
        else
        {
                hb_storc( "SUSPENDIDO", 2 );
        }
}


/*
 * Dia de la semana
 * Uso de la funcion interna hb_xstrcpy() y de hb_retc_buffer()
 */

HB_FUNC( DIASEMANA )
{
        unsigned int uiDia = hb_parni( 1 ); // Recibe el numero

        if( uiDia >= 1 && uiDia <= 7 ) // Comprueba los topes
        {
                // Aquí se guarda el literal del dia de la semana
                // es una buena prctica iniciar los puntero a NULL
                char *szDia = NULL;

                // La funcion hb_xstrcpy reserva memoria si el primer parametro es NULL
                switch( uiDia )
                {
                        case 1 :
                                szDia = hb_xstrcpy( NULL, "Lunes", NULL );
                                break;

                        case 2 :
                                szDia = hb_xstrcpy( NULL, "Martes", NULL );
                                break;

                        case 3 :
                                szDia = hb_xstrcpy( NULL, "Miercoles", NULL );
                                break;

                        case 4 :
                                szDia = hb_xstrcpy( NULL, "Jueves", NULL );
```

```
                        break;

                case 5 :
                        szDia = hb_xstrcpy( NULL, "Viernes", NULL );
                        break;

                case 6 :
                        szDia = hb_xstrcpy( NULL, "Sabado", NULL );
                        break;

                case 7 :
                        szDia = hb_xstrcpy( NULL, "Domingo", NULL );
        }

        hb_retc_buffer( szDia ); // Devuelve szDia y libera la memoria
    }
    else
    {
        hb_ret(); // Devuelve nil
    }
}
```

Y otro:

```
/*
 * Da la vuenta a una cadena pasada
 */

HB_FUNC( REVERSE )
{
    const char *szInString = hb_parc( 1 );
    int iLen = hb_parclen( 1 );

    if( iLen )
    {
        char *szRetStr = hb_xgrab( iLen );
        int i;

        for( i = 0; i < iLen; i++ )
        {
            szRetStr[ i ] = szInString[ iLen - i - 1 ];
        }

        hb_retclen( szRetStr, iLen );

        hb_xfree( szRetStr );
    }
    else
    {
        hb_retc_null();
    }
}
```

Paso de parámetros por referencia.

En la mayoría de los lenguajes de programación a una función se le pueden pasar los parámetros de dos modos diferentes, por *valor* o por *referencia.*

Por ***valor*** significa que la función llamada va trabajar con una copia de las variables pasadas como parámetros, por lo que, cuando termina la ejecución de dicha función, las variables permanecen inalteradas.

Por ***referencia*** significa que se pasa como parámetro una referencia y no una copia, por lo tanto la función llamada puede cambiar el contenido de las variables y este cambio será visible desde el punto del programa en que se usó la función.

El sistema Extendido de Harbour también soporta el paso de parámetros por ***referencia*** por lo que los cambios que se hagan en dichos parámetros serán visibles a partir del punto en que se ejecutó la función. Esto es muy útil cuando una función tiene que devolver más de un valor, ya que de otra manera sería imposible.

La funciones que usaremos para esto son las que empiezam por ***hb_stor...()***, la explicación es exactamente la misma que la dicha en las funciones ***hb_par..()***, por lo que no vamos a repetirlo.

Las principales son:

```
int hb_stor( int iParam );
int hb_storc( const char *szText, int iParam );
int hb_storclen( const char *szText, HB_SIZE nLength, int iParam );
int hb_storclen_buffer( char *szText, HB_SIZE nLength, int iParam );
int hb_stords( const char *szDate, int iParam );
int hb_storl( int iLogical, int iParam );
int hb_storni( int iValue, int iParam );
int hb_stornl( long lValue, int iParam );
int hb_stornd( double dValue, int iParam );
int hb_stornint( HB_MAXINT nValue, int iParam );
int hb_storptr( void * pointer, int iParam );
```

Os pongo un ejemplo sencillo y poco útil pero creo que muy descriptivo:

```
/*
 * Uso de parámetros por referencia
 * Recibe 2 números enteros devuelve la suma, resta, producto y división de los números pasados
 */
HB_FUNC( CALCULA )
{
        int i1 = hb_parni( 1 );
        int i2 = hb_parni( 2 );

        // Paso por referencia:
        hb_storni( i1 + i2, 3 );
        hb_storni( i1 - i2, 4 );
        hb_storni( i1 * i2, 5 );
        hb_stornd( (double) i1 / i2, 6 ); // Ojo con el cast
}
```

Podríamos haber devuelto un vector (array unidimensional) con 4 elementos. Pero eso lo vamos a ver en el siguiente apartado. Además este método es la única forma de que una función devuelva, de alguna manera, más de un resultado, usando las variables pasadas por referencia “@”.

Tratamiento de arrays.

Es un poco complicado el tratamiento de arrays con el Sistema Extendido y tal vez fue por eso, a partir de las versiones 5.xx de Clipper se introdujera el ITEM API que veremos más adelante.

Los array de Harbour tienen muchas diferencias con los de C, para empezar los primeros aceptan cualquier tipo de dato mientras que los segundos solo datos del tipo declarado. Esto quiere decir que el tratamiento no es lo mismo. Harbour ha introducido funciones específicas en el Sistema Extendido, son las ***hb_parv...()***, *l*a diferencia con la funciones las ***hb_par...()***, sin la "v" es que las primeras pueden recibir además del primer parámetro del orden otros que indican la posición en el array. Por ejemplo:

Si tenemos el array

```
...
local aNum := { 10, 20, 30 }
...
// Se podría hacer

miFunc( aNum )
...
```

Y en C si queremos acceder al segundo elemento (20) se hace así:

```
int iSegundoElemto = hb_parvni( 1, 2 ); // En iSegundoElemto vale 20
// Se puede hacer ahora esto

iSegundoElemto = 100;
```

iSegundoElemto vale 100. Luego haremos un ejemplo demostrativo, pero antes vamos a ver las funciones:

Para los parámetros pasados:

```
const char *hb_parvc( int iParam, ... );
const char *hb_parvcx( int iParam, ... );
HB_SIZE hb_parvclen( int iParam, ... );
HB_SIZE hb_parvcsiz( int iParam, ... );
const char *hb_parvds( int iParam, ... );
int hb_parvl( int iParam, ... );
double hb_parvnd( int iParam, ... );
int hb_parvni( int iParam, ... );
long hb_parvnl( int iParam, ... );
HB_MAXINT hb_parvnint( int iParam, ... );
void *hb_parvptr( int iParam, ... );
```

Para la devolución del array modificado se usan las funciones similares a la devolución por referencia o sea las ***hb_stor...()*** *pero con la v también, o sea las****hb_storv...().*** Recuerda que los array en Harbour se pasan por referencia.

Para los devolución dentro del array:

```
int hb_storvclen( const char * szText, HB_SIZE nLength, int iParam, ... );
int hb_storvclen_buffer( char * szText, HB_SIZE nLength, int iParam, ... );
int hb_storvds( const char * szDate, int iParam, ... );
int hb_storvdl( long lJulian, int iParam, ... );
int hb_storvtdt( long lJulian, long lMilliSec, int iParam, ... );
int hb_storvl( int iLogical, int iParam, ... );
int hb_storvni( int iValue, int iParam, ... );
int hb_storvnl( long lValue, int iParam, ... );
int hb_storvnd( double dValue, int iParam, ... );
int hb_storvnint( HB_MAXINT nValue, int iParam, ... );
int hb_storvptr( void * pointer, int iParam, ... );
```

Los puntos suspensivos significa tanto en C como en Harbour que pueden recibir un número indeterminado de parámetros. Si sólo se le pasa el primer parámetro estas funciones se comportan exactamente igual que las ***hb_par...()*** o las ***hb_stor...()*** ya explicadas.

Veamos ejemplos:

```
/*
 * Incrementa en 100 el segundo elemento de un array pasado
 */
HB_FUNC( CAMBIAVALOR )
{
        int iVal = hb_parvni( 1, 2 );

        iVal = iVal + 100;

        hb_storvni( iVal, 1, 2 );
}
```

Hay unas funciones del sistema Extendido que no hemos visto aún y que sirven para informar de algunos aspectos de los parámetros pasados:

```
HB_ULONG hb_parinfo( int iParam );
HB_SIZE hb_parinfa( int iParamNum, HB_SIZE nArrayIndex );
int hb_pcount( void );
HB_BOOL hb_extIsArray( int iParam );
```

hb_parinfo(n) → Informa del tipo (interno) del parámetro pasado como un número, siendo este uno de los siguientes (sólo pongo los más usados):

```
HB_IT_NIL
HB_IT_POINTER
HB_IT_INTEGER
HB_IT_HASH
```

HB_IT_LONG
HB_IT_DOUBLE
HB_IT_DATE
HB_IT_LOGICAL
HB_IT_STRING
HB_IT_MEMO
HB_IT_BLOCK
HB_IT_BYREF
HB_IT_MEMVAR
HB_IT_ARRAY
HB_IT_OBJECT
HB_IT_NUMERIC
HB_IT_NUMINT
HB_IT_HASHKEY

hb_parinfa(n, i) → Informa sobre el array. Si solo se pasa "n" devuelve el tamaño del array (LEN), si se le pasa "n" e "i" devuelve el tipo del elemento del array ocupado por "i".

hb_pcount() → Devuelve el número de parámetros pasados.

hb_extIsArray(n) → *Devuelve un valor lógico que indica si el parámetro es o no un array.*

Un ejemplo mejorable pero didáctico:

```
/*
 * Devuelve la suma de los elementos numéricos de un array pasado
 */
HB_FUNC( SUMAARRAY )
{
        double uiTotal = 0.0;

        if( hb_extIsArray( 1 ) ) // Si es un array lo proceso
        {
                HB_SIZE i;
                HB_SIZE nLen = hb_parinfa( 1, 0 ); // Número de elementos

                for( i = 0; i < nLen; i++ )
                {
                        switch( hb_parinfa( 1, i + 1 ) ) // Sumo 1 C se basa en 0 para el primer elemento
                        {
                                case HB_IT_INTEGER :
                                case HB_IT_LONG :
                                        uiTotal += hb_parvnint( 1, i + 1 );
                                        break;
                                case HB_IT_DOUBLE :
                                        uiTotal += hb_parvnd( 1, i + 1 );
                                        break;
                                // El resto de tipos no numéricos no se consideran
                        }
                }
```

```
        }

        hb_retnd( uiTotal );
}
```

Ahora un ejemplo de una función que cree el array en C y lo rellene con la tabla de multiplicar del numero pasado. Del 1 al 10:

```
/*
 * Tabla de multiplicar del numero entero pasado
 */
HB_FUNC( TABLA )
{
        if( hb_parinfo( 1 ) == HB_IT_INTEGER ) // Para esto esta la macro HB_IS_NUMINT( p ) que hace esto
        {
                HB_MAXINT iMul = hb_parnint( 1 );
                HB_UINT n = 10;
                HB_UINT i;

                hb_reta( n ); // Crea el array vacio de 100 elementos y lo pone en la zona de devolución

                for( i = 1; i <= n; i++ )
                {
                        hb_storvnint( iMul * i, -1, i ); // -1 accede a la zona de devolución
                }
        }
        else
        {
                hb_ret(); // Devuelve NULL
        }
}
```

Mira esta línea: `hb_storvnint( iMul * i, -1, i );`

Atención, nota importante: el ***-1*** se usa para acceder al valor depositado en la zona de devolución de la pila o stackReturn.

La función ***hb_reta(nLen)*** crea un array de nLen elementos con nulos en la *zona de devolución* o *stackReturn* y con ***hb_storvnint(iMul * i, -1, i);*** lo rellenamos, ya que con ese -1 accedemos al ITEM del *stackReturn.*

Tratamiento de estructura de C

Como ya hemos comentado no hay estructuras de manera nativa en Harbour. Se puede decir que lo más cercano a una estructura de C en Harbour es una array con los miembros de la estructura, o una clase que implemente como datas cada uno de los miembros de la estructura.

Aprovechando este tema vamos a introducir la ***reserva de memoria dinámica*** usando las funciones que Harbour nos proporciona que como ya dijimos son:

→ ***void *hb_xalloc(HB_SIZE nSize);***

Reserva memoria y si no lo consigue devuelve NULL.

→ ***void *hb_xgrab(HB_SIZE nSize);***

Reserva memoria y si no lo consigue genera un error y sale del programa.

→ ***void hb_xfree(void *pMem);***

Libera la memoria reservada.

→ ***void *hb_xrealloc(void *pMem, HB_SIZE nSize);***

Reasigna memoria, puede ser en aumento o en disminución.

→ ***HB_SIZE hb_xsize(void *pMem);***

Devuelve el tamaño de un bloque de memoria asignado.

Otro aspecto que quería mencionar es la diferencia entre un array y un puntero en C:

*char *szCadena;*

char szCadena[20];

Parece que sea igual, pero hay una gran diferencia, la primera declara un puntero de tipo carácter pero no reserva memoria, por lo que si lo usamos directamente produciría errores importantes y difíciles de encontrar. El segundo reserva memoria para 20 caracteres. Ese detalle es muy importante. Con el primero habría que utilizar forzosamente alguna función que reserve memoria antes de poder usarlo…

Luego el tratamiento es exactamente el mismo.

Dicho esto, vamos a hacer una función que nos devuelva el contenido de una estructura de C a nuestro PRG usando un array y otra usando un objeto. Decir que Windows, MacOS y Linux están llenos de estructuras incluso el propio Harbour las tiene en sus APIs, por lo que más pronto que tarde, vamos a tener que enfrentarnos a esto.

```
//----------------------------------------------------------------------------
// Ejercicio sobre estructura con el sistema extendido
// En PRG es un array con los mismo elementos que la estructura
// ej002.prg
//----------------------------------------------------------------------------

procedure main

    local aPersona := damePersona( 1 );

    cls

    ? "Datos de la persona 1 con su tipo de dato:"
    for n := 1 to Len( aPersona )
        ? aPersona[ n ], " -> ", ValType( aPersona[ n ] )
    next
```

```
    ?
    ? "Presiona ENTER..."
    Inkey( 100 )

    aPersona := damePersona( 2 );

    ? "Datos de la persona 2:"
    for n := 1 to Len( aPersona )
        ? aPersona[ n ]
    next
    ?
    ? "Presiona ENTER..."
    Inkey( 100 )

return
```

Y esto es la parte de C:

```
/*
 * Definicion de Persona como estructura.
 */
typedef struct
{
        char szNIF[ 10 ];
        char szNombre[ 20 ];
        HB_UINT uiCodigo;
        float nSalrio;
        char szFecha[ 9 ];
        HB_BOOL bSoltero;
} TPersona;

/*
 * Funcion que devuelve una estructura como un array
 */
HB_FUNC( DAMEPERSONA )
{
        HB_UINT uiPersona = hb_parni( 1 );
        TPersona *persona = hb_xgrab( sizeof( TPersona ) ); //Reserva memoria para TPersona

        // Rellena todos los miembros de TPersona según su tipo
        if( uiPersona == 1 )
        {
                hb_xstrcpy( persona->szNIF, "53320105T", NULL );
                hb_xstrcpy( persona->szNombre, "Viruete", ", ", "Paco", NULL );
                persona->uiCodigo = 26212;
                persona->nSalrio = 3500.97;
                hb_xstrcpy( persona->szFecha, "19561225", NULL );
                persona->bSoltero = HB_FALSE;
        }
        else
        {
                hb_xstrcpy( persona->szNIF, "43310009H", NULL );
                hb_xstrcpy( persona->szNombre, "Grande", ", ", "Felix", NULL );
                persona->uiCodigo = 13101;
                persona->nSalrio = 2750.75;
                hb_xstrcpy( persona->szFecha, "19660213", NULL );
                persona->bSoltero = HB_TRUE;
        }
```

```
        hb_reta( 6 ); // Crea el array en el stackReturn

        // Rellena el array creado en el stackReturn con el "-1"
        hb_storvc( persona->szNIF, -1, 1 );             // Cadena
        hb_storvc( persona->szNombre, -1, 2 );  // Cadena
        hb_storvni( persona->uiCodigo, -1, 3 ); // Entero
        hb_storvnd( persona->nSalrio, -1, 4 );  // Real
        hb_storvds( persona->szFecha, -1, 5 );  // Fecha como cadena <---
        hb_storvl( persona->bSoltero, -1, 6 );  // Booleano <---

        // ATENCION: toda la memoria que hayamos reservado la tenemos que liberar
        hb_xfree( persona );
}
```

Y ahora usando un Objeto como receptor de la estructura:

```
//----------------------------------------------------------------------------
// Ejercicio sobre estructura con el sistema extendido
// Trucos...
// En PRG es un objeto con las mismas datas que miembros de la estructura
// ej003.prg
//----------------------------------------------------------------------------

#include "hbclass.ch"

procedure main

    local persona := TPersona():new( 1 )

    cls

    persona:muestra()
    ?
    ? "Presiona ENTER..."
    Inkey( 100 )

    cls

    persona := TPersona():new( 2 )
    persona:muestra()
    ?
    ? "Presiona ENTER..."
    Inkey( 100 )

    cls

    // En Harbour todas las clases se comportan como las clases estaticas de Java
    // por lo que podriamos hacer esto:
    TPersona():new( 1 ):muestra()
    ?
    ? "Presiona ENTER..."
    Inkey( 100 )

return

//----------------------------------------------------------------------------
// Clase TPersona contenedora de una estructura en C
//----------------------------------------------------------------------------
```

```
CREATE CLASS TPersona

        DATA cNIF
        DATA cNombre
        DATA nCodigo
        DATA nSalrio
        DATA dFecha
        DATA lSoltero

        METHOD new( nPersona ) CONSTRUCTOR
        METHOD muestra()

END CLASS

//-----------------------------------------------------------------------------
// Constructor

METHOD new( nPersona ) CLASS TPersona

    // Funcion en C que carga las datas desde la estructura de C
    // Deben coincidir las datas y los miembros de la estructura

    dameObjPersona( self, nPersona )

return self

//-----------------------------------------------------------------------------
// Muestra por pantalla el contenido del Objeto/Estructura

PROCEDURE muestra() CLASS TPersona

        ? "Datos de la persona del objeto:"
        ? ::cNIF
        ? ::cNombre
        ? ::nCodigo
        ? ::nSalrio
        ? ::dFecha
        ? ::lSoltero

return
```

Y la parte de C:

```
/*
 * Lo mismo que la anterior pero usando un objeto en vez de un array
 */

 #include "hbapiitm.h"

HB_FUNC( DAMEOBJPERSONA )
{
        PHB_ITEM obj = hb_param( 1, HB_IT_OBJECT );
        HB_UINT uiPersona = hb_parni( 2 );
        TPersona *persona = hb_xgrab( sizeof( TPersona ) ); //Reserva memoria para TPersona
```

```
        // Rellena todos los miembros de TPersona según su tipo
        if( uiPersona == 1 )
        {
                hb_xstrcpy( persona->szNIF, "53320105T", NULL );
                hb_xstrcpy( persona->szNombre, "Viruete", ", ", "Paco", NULL );
                persona->uiCodigo = 26212;
                persona->nSalrio = 3500.97;
                hb_xstrcpy( persona->szFecha, "19561225", NULL );
                persona->bSoltero = HB_FALSE;
        }
        else
        {
                hb_xstrcpy( persona->szNIF, "43310009H", NULL );
                hb_xstrcpy( persona->szNombre, "Grande", ", ", "Felix", NULL );
                persona->uiCodigo = 13101;
                persona->nSalrio = 2750.75;
                hb_xstrcpy( persona->szFecha, "19660213", NULL );
                persona->bSoltero = HB_TRUE;
        }

        hb_itemReturn( obj ); // Esto mete el objeto en el stackReturn

        // Rellena el objeto por posicion de la DATA creado en el stackReturn con el "-1"
        hb_storvc( persona->szNIF, -1, 1 );             // Cadena
        hb_storvc( persona->szNombre, -1, 2 );  // Cadena
        hb_storvni( persona->uiCodigo, -1, 3 ); // Entero
        hb_storvnd( persona->nSalrio, -1, 4 );  // Real
        hb_storvds( persona->szFecha, -1, 5 );  // Fecha como cadena <---
        hb_storvl( persona->bSoltero, -1, 6 );  // Booleano <---

        // ATENCION: toda la memoria que hayamos reservado la tenemos que liberar
        hb_xfree( persona );
}
```

Muy importante decir que somos nosotros los responsables de liberar la memoria reservada, en C no hay Recolector de Basura como en Harbour.

Harbour trata internamente los objetos y los arrays de la misma manera, si se hace un LEN() del objeto devolverá el numero de DATAs que contiene. Si pretendemos usar este truco hay que tener cuidado de no variar las DATAs y los miembros de la estructura…

He tenido que usar algo del ITEM API que veremos más adelante:

```
PHB_ITEM obj = hb_param( 1, HB_IT_OBJECT );
```

y

```
hb_itemReturn( obj ); // Esto mete el objeto en el stackReturn
```

Pero me ha parecido oportuno…

El sistema extendido es ideal para hacer ***interfaces*** a las funciones de la biblioteca estándar de C o a nuestras propias funciones escritas en C puro para ser usadas indistintamente en C y en PRG:

Esta es la función hecha en C para ser usada en C:

```
char *cstrtran( const char *cString, HB_SIZE nLenStr, const char *cFind, HB_SIZE nLenFind,
                const char *cReplace, HB_SIZE nLenRep )
{
        HB_SIZE i, n, w = 0;
        HB_BOOL fFind = HB_FALSE;
        char *cRet = ( char * ) hb_xgrab( nLenStr + 1 );

        for( i = 0; i < nLenStr; i++ )
        {
                for( n = 0; n < nLenFind; n++ )
                {
                        fFind = cFind[ n ] == cString[ i ];

                        if( fFind )
                        {
                                if( n < nLenRep )
                                {
                                        cRet[ w ] = cReplace[ n ];
                                        w++;
                                }

                                break;
                        }
                }

                if( !fFind )
                {
                        cRet[ w ] = cString[ i ];
                        w++;
                }
        }

        cRet[ w ] = '\0';

        return( cRet );
}
```

Y esta es el interfaz para usar desde PRG:

```
HB_FUNC( CSTRTRAN )
{
        hb_retc_buffer( cstrtran( hb_parc( 1 ), hb_parclen( 1 ), hb_parc( 2 ),
                                  hb_parclen( 2 ), hb_parc( 3 ), hb_parclen( 3 ) ) );
}
```

Es una idea muy chula la función cstrtran() se puede usar desde C y desde PRG.

Por otro lado esta función tiene mucha utilidad, mira un ejemplo.

La función *cStrTran(cCadena, cPatro1, cPatron2)*

cCadena → La cadena que se va a procesar.

cPatron1 → Caracteres que se van cambiar.

cPatron2 → Caracteres por que se van cambiar.

cPatron1 y cPatron2 no tienen porqué ser simétricos.

Aquí está el ejemplo:

```
//-----------------------------------------------------------------------------
//  Uso de Interface de la funcion de C hecho por nosostros mismos
// ej004.prg
//-----------------------------------------------------------------------------

REQUEST HB_CODEPAGE_ESISO

procedure main()

        local cString := "Manuel Expósito Suárez"

    hb_cdpSelect( "ESISO" )

        ? "Test con la cadena.....: " + cString
        ? "Pone en mayúscula las vocales.:", cStrTran( cString, "aeiouáéíóú", "AEIOUÁÉÍÓÚ" )
        ? "Quita las vocales.....:", cStrTran( cString, "aeiouáéíóúAEIOUÁÉÍÓÚ", "" )
        ? "Quita la tilde.....", cStrTran( cString, "áéíóú", "aeiou" )

      InKey( 100 )

return
```

Este ejemplo muestra una interfaz a un programa de usuario hecho en C y ahora vamos a ver como podemos usar cualquier función de la biblioteca de C. Voy a usar las funciones trigonométricas seno, coseno y tangente. Esta es la interfaz de esas funciones:

```
/*
 * Interfaces con funciones matematicas de C
 */
#include <math.h>
/* coseno */
HB_FUNC( C_COS )
{
        hb_retnd( cos( hb_parnd( 1 ) ) );
}
/* seno */
HB_FUNC( C_SIN )
{
        hb_retnd( sin( hb_parnd( 1 ) ) );
}
/* tangente */
HB_FUNC( C_TAN )
{
        hb_retnd( tan( hb_parnd( 1 ) ) );
}
```

Y esta otra es una interfaz a la función de token de C strtok() con algún control adicional:

```
/*
 * Usando interfaces a funciones de C
 */

HB_FUNC( STRTOK )
{
        int iParametros = hb_pcount();

        if( iParametros > 0 )
        {
                const char *p = hb_parc( 1 );
                const char *s = hb_parc( 2 );
```

```
                hb_retc( strtok( ( char * ) p, s ) );
        }
}
```

Y ahora el ejemplo de uso:

```
//-----------------------------------------------------------------------------
//  Uso de Interface de las funciones de la biblioteca de C
//  ej005.prg
//-----------------------------------------------------------------------------

procedure main

    local getlist := {}
    local cFrase := Space( 60 )
    local cSep := Space( 2 )
    local cToken, n := 0
    local nNum := 0.0

    set decimal to 6

    cls

    @ 01, 01 SAY "Interfaces..."
    @ 02, 01 SAY "Dame una frase:" GET cFrase PICTURE "@k!"
    @ 03, 01 SAY "Separador.....:" GET cSep   PICTURE "@k!"
    @ 04, 01 SAY "Entre numero..:" GET nNum   PICTURE "@k"
    read

    cFrase := AllTrim( cFrase ) // Quito espacios sobrantes
    cSep := AllTrim( cSep )

    if Empty( cSep )
        cSep := " " // Por defecto es un espacio
    endif

    cls
    ? "Frase....:", cFrase
    ? "Al reves.:", reverse( cFrase )
    ?

    /* El primer token */
    cToken := strtok( cFrase, cSep )

    /* Bucle mientras haya token */
    while( !Empty( cToken ) )
        ? ++n, "token ->", cToken
        cToken := strtok( nil, cSep )
    end

    ?
    ? "Funciones trigonometricas de", nNum, "experesada en radianes:"
    ? "Coseno.....:", c_Cos( nNum )
    ? "Seno.......:", c_Sin( nNum )
    ? "Tangente...:", c_Tan( nNum )

    ?
    ? "Presione ENTER para seguir..."
    Inkey( 100 )

return
```

Como se puede ver el Sistema Extendido abre un mundo de posibilidades. Todo lo que hay en C puede ser usado en nuestros PRGs.

Y a modo de resumen del sistema extendido podemos decir que hay tres tipos básicamente de funciones encargadas de convertir las variables de Harbour en variables de tipo primitivo o simples de C:

- Las funciones que tratan los parámetros pasados: ***hb_par..()***

hb_**par**...() → de parámetro

- Las funciones de devolución del resultado una vez procesado en C: ***hb_ret...()***

hb_**ret**...() → de return → stackReturn

- Las funciones que tratan los parámetros por referencia: ***hb_stor...()***

hb_**stor.**..() → stored to reference → Almacenado para referencia

Esquema de una función en C para ser usada desde PRG:

```
HB_FUNC( NOMBRE_FUNCION )
{
        // La recepción de parámetros
         … = hb_par...();
         … = hb_par...();
        …
        // En el cuerpo de la función se tratan y procesan los parámetros ya
        // convertidos a tipos de C por las funciones hb_par...();
        …
        // La devolución del dato resultado concreto
        hb_ret...( valor ); // Solo uno
}
```

Con el Sistema Extendido no se puede trabajar con las variables de Harbour directamente o sea, no se puede trabajar con los ***ITEM***, realmente son intermediarios, una capa. En el siguiente tema sí vamos a ver el ***ITEM API*** que es la nueva propuesta de Clipper, y que Harbour ha mejorado muchísimo, que hace que todo sea más flexible y robusto. Y con el que vamos a poder interactuar directamente con las variables de Harbour, que vistas desde C son la estructuras ***ITEM***. Incluidos los arrays, codeblock o tablas hash. Esto se pone bueno!!!

16. El Item Api. Ampliando el sistema Extendido.

Como ya he dicho en varias ocasiones anteriormente, esta fue la nueva apuesta que Clipper puso en marcha con la llegada de Clipper 5.00 y posteriores. Harbour lo ha ampliado y mejorado.

¿Pero qué es un API en Harbour?

Interfaz de Programación de Aplicaciones o en inglés A*pplication Programming Interface*. Un API es un conjunto de definiciones funciones, reglas y protocolos para poder programar en C código que podamos usar en nuestros programas PRG de Harbour. En la definición de las funciones se especifica el tipo de variable que devuelve y los tipos de las variables que se pueden usar como parámetros. El *Sistema Extendido* es un API y el *ITEM API* también lo es.

¿Y qué es un ITEM en Harbour?

Ya lo vimos anteriormente pero volvemos a hacerlo ahora. Un ITEM en Harbour es una estructura que representa a una variable de Harbour en C, está formada por dos miembros:

- ***HB_TYPE type*** que contiene el tipo de ITEM como un número entero. Todos los tipos de variables están representados.

- Y una ***union*** de C cuyos miembros son una estructuras con cada uno de los tipos de Harbour, esas estructura tienen un miembro llamado ***value*** que contiene el valor de la variable.

```
typedef struct _HB_ITEM
{
   HB_TYPE type;
   union
   {
      struct hb_struArray     asArray;
      struct hb_struBlock     asBlock;
      struct hb_struDateTime  asDateTime;
      struct hb_struDouble    asDouble;
      struct hb_struInteger   asInteger;
      struct hb_struLogical   asLogical;
      struct hb_struLong      asLong;
      struct hb_struPointer   asPointer;
      struct hb_struHash      asHash;
      struct hb_struMemvar    asMemvar;
      struct hb_struRefer     asRefer;
      struct hb_struEnum      asEnum;
      struct hb_struExtRef    asExtRef;
      struct hb_struString    asString;
      struct hb_struSymbol    asSymbol;
      struct hb_struRecover   asRecover;
   } item;
} HB_ITEM, * PHB_ITEM;
```

El miembro HB_TYPE type puede ser cualquiera de estos:

```
/* Tipos de elementos y macros de verificación de tipos */
#define HB_IT_NIL       0x00000
#define HB_IT_POINTER   0x00001
#define HB_IT_INTEGER   0x00002
#define HB_IT_HASH      0x00004
#define HB_IT_LONG      0x00008
#define HB_IT_DOUBLE    0x00010
#define HB_IT_DATE      0x00020
```

```
#define HB_IT_TIMESTAMP 0x00040
#define HB_IT_LOGICAL   0x00080
#define HB_IT_SYMBOL    0x00100
#define HB_IT_ALIAS     0x00200
#define HB_IT_STRING    0x00400
#define HB_IT_MEMOFLAG  0x00800
#define HB_IT_MEMO      ( HB_IT_MEMOFLAG | HB_IT_STRING )
#define HB_IT_BLOCK     0x01000
#define HB_IT_BYREF     0x02000
#define HB_IT_MEMVAR    0x04000
#define HB_IT_ARRAY     0x08000
#define HB_IT_ENUM      0x10000
#define HB_IT_EXTREF    0x20000
#define HB_IT_DEFAULT   0x40000
#define HB_IT_RECOVER   0x80000
#define HB_IT_OBJECT    HB_IT_ARRAY
#define HB_IT_NUMERIC   ( HB_IT_INTEGER | HB_IT_LONG | HB_IT_DOUBLE )
#define HB_IT_NUMINT    ( HB_IT_INTEGER | HB_IT_LONG )
#define HB_IT_DATETIME  ( HB_IT_DATE | HB_IT_TIMESTAMP )
#define HB_IT_ANY       0xFFFFFFFF
#define HB_IT_COMPLEX     ( HB_IT_BLOCK | HB_IT_ARRAY | HB_IT_HASH | HB_IT_POINTER | /* HB_IT_MEMVAR |
HB_IT_ENUM | HB_IT_EXTREF |*/ HB_IT_BYREF | HB_IT_STRING )
#define HB_IT_GCITEM    ( HB_IT_BLOCK | HB_IT_ARRAY | HB_IT_HASH | HB_IT_POINTER | HB_IT_BYREF )
#define HB_IT_EVALITEM  ( HB_IT_BLOCK | HB_IT_SYMBOL )
#define HB_IT_HASHKEY     ( HB_IT_INTEGER | HB_IT_LONG | HB_IT_DOUBLE | HB_IT_DATE | HB_IT_TIMESTAMP |
HB_IT_STRING | HB_IT_POINTER )
```

El miembro "***type***" de la estructura ***HB_ITEM*** determina que estructura de la *union* tiene el "***value***" apropiado. Esa "*union*" de C es la que hace que las variables de Harbour puedan contener diferentes tipos de datos simplemente con la asignación de un determinado valor.

Por ejemplo:

```
local xVar

ValType( xVar ) → U
xVar := 1000
ValType( xVar ) → N
xVar := "Hola mundo"
ValType( xVar ) → C
xVar := Date()
ValType( xVar ) → D
xVar := Array( 5 )
ValType( xVar ) → A
xVar := {||}
ValType( xVar ) → B

...
```

Esto en Lenguaje C no es posible ya que las variables tienen que estar definidas con su tipo antes de ser usadas. Y además no se puede cambiar ese tipo en la misma función o mejor dicho en el ámbito de esa variable.

Las funciones que componen el ***ITEM API*** están definidas en el archivo ***hbapiitm.h***, básicamente son las siguientes:

- Determina el número de parámetros pasados:

```
HB_USHORT hb_itemPCount( void );  // Igual que hb_pcount()
```

- Recoge el parámetro pasado desde PRG y lo asigna en C a un PHB_ITEM (puntero a ITEM):

```
PHB_ITEM hb_param( int iParam, long lMask );

PHB_ITEM hb_itemParam( HB_USHORT uiParam );

PHB_ITEM hb_itemParamPtr( HB_USHORT uiParam, long lMask ); // Igual que hb_param()
```

- Devuelve un ITEM de C a PRG

```
PHB_ITEM hb_itemReturn( PHB_ITEM pItem );

PHB_ITEM hb_itemReturnForward( PHB_ITEM pItem );

void hb_itemReturnRelease( PHB_ITEM pItem );
```

- Devuelve el tamaño de un ITEM del tipo Array, Hash o Cadena

```
HB_SIZE hb_itemSize( PHB_ITEM pItem );
```

- Devuelve el tipo actual del ITEM en forma de entero (ver tabla más abajo)

```
HB_TYPE hb_itemType( PHB_ITEM pItem );
```

- Devuelve el tipo actual del ITEM en forma de carácter, como ValType()

```
const char * hb_itemTypeStr( PHB_ITEM pItem );
```

- Crea un nuevo item

```
PHB_ITEM hb_itemNew( PHB_ITEM pNull );
```

- Libera la memoria ocupada por un item

```
HB_BOOL hb_itemRelease( PHB_ITEM pItem );
```

- Tratamiento básico de Arrays

```
PHB_ITEM hb_itemArrayNew( HB_SIZE nLen );

PHB_ITEM hb_itemArrayGet( PHB_ITEM pArray, HB_SIZE nIndex );
PHB_ITEM hb_itemArrayPut( PHB_ITEM pArray, HB_SIZE nIndex, PHB_ITEM pItem );
```

- Obtiene el valor actual (value) del item para ser usado en C:

```
char *hb_itemGetC( PHB_ITEM pItem );
```

```
HB_SIZE hb_itemGetCLen( PHB_ITEM pItem );
char *hb_itemGetDS( PHB_ITEM pItem, char *szDate );
char *hb_itemGetTS( PHB_ITEM pItem, char *szDateTime );
HB_BOOL hb_itemGetL( PHB_ITEM pItem );
double hb_itemGetND( PHB_ITEM pItem );
int hb_itemGetNI( PHB_ITEM pItem );
long hb_itemGetNL( PHB_ITEM pItem );
HB_MAXINT hb_itemGetNInt( PHB_ITEM pItem );
void *hb_itemGetPtr( PHB_ITEM pItem );
```

- Asigna a value (valor) al pItem, si pItem es NULL lo crea y asigna:

```
PHB_ITEM hb_itemPutC( PHB_ITEM pItem, const char * szText );
PHB_ITEM hb_itemPutCL( PHB_ITEM pItem, const char * szText, HB_SIZE nLen );
PHB_ITEM hb_itemPutCPtr( PHB_ITEM pItem, char * szText );
PHB_ITEM hb_itemPutD( PHB_ITEM pItem, int iYear, int iMonth, int iDay );
PHB_ITEM hb_itemPutDS( PHB_ITEM pItem, const char * szDate );
PHB_ITEM hb_itemPutTS( PHB_ITEM pItem, const char * szDateTime );
PHB_ITEM hb_itemPutDL( PHB_ITEM pItem, long lJulian );
PHB_ITEM hb_itemPutTD( PHB_ITEM pItem, double dTimeStamp );
PHB_ITEM hb_itemPutTDT( PHB_ITEM pItem, long lJulian, long lMilliSec );
PHB_ITEM hb_itemPutL( PHB_ITEM pItem, HB_BOOL bValue );
PHB_ITEM hb_itemPutND( PHB_ITEM pItem, double dNumber );
PHB_ITEM hb_itemPutNI( PHB_ITEM pItem, int iNumber );
PHB_ITEM hb_itemPutNL( PHB_ITEM pItem, long lNumber );
PHB_ITEM hb_itemPutNInt( PHB_ITEM pItem, HB_MAXINT nNumber );
PHB_ITEM hb_itemPutPtr( PHB_ITEM pItem, void * pValue );
```

- Asigna nil al item:

```
HB_ITEM hb_itemPutNil( PHB_ITEM pItem );
```

Hay más, pero nos quedamos con esas…

- ***<u>Entrada de parámetros de PRG a C:</u>***

A diferencia del *Sistema Extendido* que hay una función por cada tipo de dato de C, en el ***ITEM API*** se recogen los parámetros como un *ITEM*. Por lo que realmente sólo hay 3 funciones:

PHB_ITEM hb_param(int iParam, long lMask) *y*

PHB_ITEM hb_itemParamPtr(HB_USHORT uiParam, long lMask) son sinónimas, ambas reciben la variable pasada desde PRG como una estructura *ITEM* y una máscara del tipo (***HB_TYPE***) de *ITEM*. Si el parámetro no se corresponde con la máscara devuelve *NULL*.

PHB_ITEM hb_itemParam(HB_USHORT uiParam) funciona de la misma manera que las dos anteriores pero tiene un segundo parámetro con la mascara del tipo y además hacen una copia del *ITEM* pasado por lo que antes de salir de la función en la que estemos hay que liberar el *ITEM* copiado.

- ***Devolución de valores de C a PRG:***

Sólo se devuelve un ITEM sea del tipo que sea y no hay una función por cada tipo de dato como en el Sistema Extendido. Son sólo estas:

PHB_ITEM hb_itemReturn(PHB_ITEM pItem) devuelve el ITEM al PRG.

void hb_itemReturnRelease(PHB_ITEM pItem) devuelve el ITEM al PRG y lo libera de la memoria.

PHB_ITEM hb_itemReturnForward(PHB_ITEM pItem) devuelve el ITEM al PRG, este ITEM normalmente no se recoge con las funciones de recogida de parámetros del tipo "param" explicadas anteriormente ni son ITEM creados por nosotros sino sacados de la Pila (stack) y por tanto los ha creado el propio Harbour y será el recolector de basura el que lo libere.

Realmente casi sólo vamos a usar los dos primeros.

- ***Tratamiento básico de Arrays Harbour en C:***

Pues sólo hay 3 al igual que a alto nivel o sea en PRG:

Creación de un array con nLen elementos:

```
PHB_ITEM hb_itemArrayNew( HB_SIZE nLen )
```

Modificar el valor de un elementos:

```
PHB_ITEM hb_itemArrayPut( PHB_ITEM pArray, HB_SIZE nIndex, PHB_ITEM pItem )
```

Conseguir el valor de un elemento como ITEM:

```
PHB_ITEM hb_itemArrayGet( PHB_ITEM pArray, HB_SIZE nIndex )
```

Hay más funciones para el tratamiento de array de Harbour en C que veremos más adelante, Sólo adelantar que se tratan de funciones del tipo ***hb_itemArrayPut()*** y ***hb_itemArrayGet()*** pero en vez de tratar con el parámetro ***pItem*** o de devolver un ***ITEM***, tratan con valores de tipo C. Realmente es una capa para no tener que convertir los valores de tipo C a *ITEM*. Ya la veremos...

- ***Creación y liberación de un ITEM en C:***

Si en algún momento necesitamos crear un ITEM en una función de C tendremos que usar la función del ITEM API ***PHB_ITEM hb_itemNew(PHB_ITEM pNull)*** Si el parámetro que recibe "***pNull***" es otro *ITEM* existente crea una copia del mismo pero si es *NULL* crea un *ITEM* sin tipo "U" indefinido hasta que le asignemos algún valor a *value.* En cualquier caso tendremos que liberar el ITEM creado antes de abandonar la función con la función del ITEM API que tenemos para tal fin ***HB_BOOL hb_itemRelease(PHB_ITEM pItem)***. Le pasamos el ITEM creado y el Recolector de Basura se encarga de liberar la memoria utilizada.

- ***Obtención del valor actual (value) del ITEM para ser usado en C:***

Son las funciones del ***hb_itemGet...(PHB_ITEM pItem)***, devuelven el miembro "***value***" de la "***union***" de la estructura *ITEM*. Recordad, por tanto, que puede ser del tipo de C que sea ya que cualquiera de esos tipos están como miembros de la "***union***" (mira el código de la estructura *HB_ITEM* más arriba).
Importante el *ITEM* del que saquemos esa función tiene que existir previamente, ya sea porque lo hayamos obtenido como parámetro desde *PRG* o por alguna otra función interna de los *APIs* de *Harbour*. La mayoría de ellas están relacionadas más arriba.

- ***Asignación de un valor a value (valor) al ITEM:***

Son las funciones del *ITEM API* ***PHB_ITEM hb_itemPut...()***, normalmente reciben el *ITEM* al que le vamos a asignar el valor y el propio valor (de *tipo nativo de C*). Mira más arriba.
Muy importante si el primer parámetro es NULL crea un nuevo ITEM y le asigna el valor pasado como segundo parámetro.

Otras funciones importantes del API ITEM:

HB_BOOL hb_itemEqual(PHB_ITEM pItem1, PHB_ITEM pItem2) Compara dos *ITEM* pasados, devuelve *HB_TRUE* si son iguales o *HB_FALSE* si no lo son.

void hb_itemCopy(PHB_ITEM pDest, PHB_ITEM pSource) Copia un elemento de un lugar a otro respetando su contenido.

void hb_itemMove(PHB_ITEM pDest, PHB_ITEM pSource) Mueve el valor de un ITEM sin incrementar los contadores de referencia, el ITEM fuente se limpia (lo pone sin valor con tipo "U").

void hb_itemClear(PHB_ITEM pItem) Limpia el ITEM (lo pone sin valor con tipo "U").

PHB_ITEM hb_itemClone(PHB_ITEM pItem) Clona el ITEM dado.

char *hb_itemStr(PHB_ITEM pNumber, PHB_ITEM pWidth, PHB_ITEM pDec)
Convierte un ITEM numérico en cadena con el ancho y los decimales indicados.

char *hb_itemString(PHB_ITEM pItem, HB_SIZE *nLen, HB_BOOL *bFreeReq)
Convierte cualquier tipo de ITEM a cadena, en el puntero nLen obtendremos el ancho de la cadena y en bFreeReq un valor lógico indicando si tenemos que liberar la memoria ocupada por la cadena devuelta.

PHB_ITEM hb_itemValToStr(PHB_ITEM pItem) Es igual que la anterior pero en vez de una cadena devuelve un ITEM de tipo cadena con el valor convertido.

void hb_itemSwap(PHB_ITEM pItem1, PHB_ITEM pItem2) Intercambia los ITEM.

Hay más pero con estas son suficientes de momento.

<u>Notas sobre la nomenclaturas de las funciones del ITEM API :</u>

PHB_ITEM → se refiere a un puntero a una estructura tipo ITEM (o lo que es igual a una variable como la entendemos en un PRG).
PHB_ITEM pArray → Array de Harbour visto como estructura en C.
HB_SIZE nIndex → Indice de una array de Harbour (basado en 1 y no en 0 como los arrays de C).
HB_SIZE nLen → Tamaño de un array (numero de elementos) o un hash o de una cadena.

<u>Los sufijos de las funciones *hb_itemGet...()* y *hb_itemPut...()*</u>

Son los mismo que en el Sistema Extendido:
N → Números, el siguiente digito indica el tipo de número puede ser:
NI → Enteros
NL → Entero largo
ND → Real o doble
NInt → Máximo tamaño de un entero

D → Para fechas pasando int iYear, int iMonth, int iDay
DS → Fecha como cadena YYYYMMDD
DL → Como número entero largo

TS → DateTime como cadena

Ptr → Punteros

C → Cadena

L → lógico

<u>NOTA IMPORTANTE</u>: Cualquier *ITEM* que creemos directamente con ***hb_itemNew()*** o que hayamos recibido con la función ***hb_itemParam()*** o con las funciones de asignación ***hb_itemPut...()*** cuyo primer parámetro sea *NULL* ***hb_itemPut...(NULL, xxxx)*** tenemos que liberar la memoria con la función ***hb_itemRelease(),*** de lo contrario habrá perdidas de memoria.

Pasamos a los ejemplos. El primero es para explicar que una variables vista desde PRG es lo mismo que un ITEM visto desde C.

```
/*
 * ej000.prg
 * Uso de ITEM
 */

procedure main

    local cVar, nVar, dVar, lVar
    local aVar := Array( 4 )

    SET DATE FORMAT TO "dd/mm/yyyy"

    cls

    ? "Asignacion de variables desde C:"
    ? "--------------------------------"
    ?
    asignaVarC( @cVar, @nVar, @dVar, @lVar, aVar )

    ? "Cadena...:", cVar
    ? "Numero...:", nVar
    ? "Fecha....:", dVar
    ? "Logico...:", lVar
    ?
    ? "Array....:"
    for i := 1 to 4
        ? i, aVar[ i ]
    next

    ?
    ? "Presione ENTER para seguir..."
    Inkey( 100 )

return
```

La función ***asignaVarC()*** se encarga de inicializar con valores cada una de las variables pasadas por referencia:

```
/*
 * Asigna valores a parametros pasados por referencia
 */

HB_FUNC( ASIGNAVARC )
{
        PHB_ITEM cVar = hb_param( 1, HB_IT_BYREF );
        PHB_ITEM nVar = hb_param( 2, HB_IT_BYREF );
```

```
        PHB_ITEM dVar = hb_param( 3, HB_IT_BYREF );
        PHB_ITEM lVar = hb_param( 4, HB_IT_BYREF );
        PHB_ITEM aVar = hb_param( 5, HB_IT_ARRAY );

        // Cadena
        if( cVar )
        {
                hb_itemPutC( cVar, "Cadena asignada en C" );
        }

        // Numerico
        if( nVar )
        {
                hb_itemPutNI( nVar, 100 );
        }

        // Fecha
        if( dVar )
        {
                hb_itemPutDS( dVar, "20210517" );
        }

        // Logico
        if( lVar )
        {
                hb_itemPutL( lVar, HB_TRUE );
        }

        // Array
        if( aVar )
        {
                hb_itemArrayPut( aVar, 1, cVar );
                hb_itemArrayPut( aVar, 2, nVar );
                hb_itemArrayPut( aVar, 3, dVar );
                hb_itemArrayPut( aVar, 4, lVar );
        }
}
```

Ahora los mismos ejemplos del Sistema Extendido pero usando el ITEM API los PRG son exactamente los mismos por los que solo pongo las funciones en C

```
/*****************************************************************************
 ************************* Uso del ITEM API *********************************
 *****************************************************************************/

#include "hbapi.h"     // Siempre jay que incluir para funciones en C
#include "hbapiitm.h"  // Para uso del ITEM API

/*
 * Devuelve si un año es bisiesto o no
 */

HB_FUNC( MI_ISLEAP )
{
        // Recibe el parametro 1 como un entero (HB_IT_INTEGER).
```

```
        // Si no es del tipo especificado por la mascara nYear sera NULL
        PHB_ITEM nYear = hb_param( 1, HB_IT_INTEGER );
        // Crea el ITEM y asigna .f., Normalmente se usa hb_itemNew()
        //      pero si a las funciones hb_itemPut...() se les pasa en el parametro 1 un NULL
        //crean un nuevo ITEM y le asignan el valor del parametro 2
        PHB_ITEM lRet = hb_itemPutL( NULL, HB_FALSE );

        if( nYear ) // Si no es NULL
        {
                // Obtenemos el valor del ITEM como un entero
                HB_UINT uiYear = hb_itemGetNI( nYear );

                if( uiYear > 0 )
                {
                        // Asignamos en el ITEM lRet el valor logico devuelto por la expresion
                        hb_itemPutL( lRet, ( ( uiYear % 4 == 0 && uiYear % 100 != 0 ) || uiYear % 400 == 0 ) );
                }
        }

        // Devuelve el ITEM lRet y lo libera para no tener perdida de memoria
        hb_itemReturnRelease( lRet );
}

/*
 * El cubo de un numero pasado por referencia
 */

HB_FUNC( MI_CUBO )
{
        // Recuperamos el parametro pasado como un ITEM para entero de cualquier longitud
        PHB_ITEM nCubo = hb_param( 1, HB_IT_NUMINT );
        // Creamos un ITEM vacio
        PHB_ITEM nRes = hb_itemNew( NULL );

        if( nCubo ) // Si se paso un entero
        {
                // Recupera el entero del ITEM
                HB_MAXINT iCubo = hb_itemGetNInt( nCubo );

                // Asigna el value con el resultado del cubo al ITEM creado
                hb_itemPutNInt( nRes, iCubo * iCubo * iCubo );
        }

        // Devuelve el ITEM con el numero del resultado
        hb_itemReturnRelease( nRes );
}

/*
 * Cuenta las ocurrencias de un caracter dentro de una cadena
 */

HB_FUNC( CHARCOUNT )
{
        // Recuperamos el parametro pasado como ITEM de tipo cadena
        PHB_ITEM cCadena = hb_param( 1, HB_IT_STRING );
        // Crea un ITEM de tipo entero de cualquier longitud
        PHB_ITEM nRes = hb_itemPutNInt( NULL, 0 );

        if( cCadena ) // Si se paso un parametro valido o sea una cadena
        {
                // Recuperamos el segundo parametro como un ITEM cadena
                PHB_ITEM cCarcter = hb_param( 2, HB_IT_STRING );
```

```
                if( cCarcter ) // Si se paso un parametro valido
                {
                        // Ancho del ITEM cCadena
                        HB_SIZE uiLen = hb_itemGetCLen( cCadena );
                        HB_SIZE i; // Indice de incremento del for
                        HB_SIZE uiContador = 0; // Contador de ocurrencias
                        // Obtenemos la primera cadena del ITEM pasado
                        const char *szCadena = hb_itemGetC( cCadena );
                        // Obtenemos la segunda cadena del ITEM pasado
                        const char *szCarcter = hb_itemGetC( cCarcter );

                        // Proceso de calculo
                        for( i = 0; i < uiLen; i++ )
                        {
                                if( szCadena[ i ] == szCarcter[ 0 ] )
                                {
                                        ++uiContador;
                                }
                        }

                        // Asigna el contador al ITEM que vamos a devolver
                        hb_itemPutNInt( nRes, uiContador );
                }
        }

        // Devuelve el ITEM con el numero del resultado
        hb_itemReturnRelease( nRes );
}

/*
 * Busca la primera aparicion de un caracter dentro de una cadena
 */

HB_FUNC( SCANCHAR )
{
        // Recuperamos el parametro pasado como ITEM de tipo cadena
        PHB_ITEM cCadena = hb_param( 1, HB_IT_STRING );
        // Crea un ITEM de tipo entero de cualquier longitud
        PHB_ITEM nRes = hb_itemPutNInt( NULL, 0 );

        if( cCadena ) // Si se paso un parametro valido o sea una cadena
        {
                // Recuperamos el segundo parametro como un ITEM cadena
                PHB_ITEM cCarcter = hb_param( 2, HB_IT_STRING );

                if( cCarcter ) // Si se paso un parametro valido
                {
                        // Ancho del ITEM cCadena
                        HB_SIZE uiLen = hb_itemGetCLen( cCadena );
                        HB_SIZE i; // Indice de incremento del for
                        // Obtenemos la primera cadena del ITEM pasado
                        const char *szCadena = hb_itemGetC( cCadena );
                        // Obtenemos la segunda cadena del ITEM pasado
                        const char *szCarcter = hb_itemGetC( cCarcter );

                        // Proceso de calculo
                        for( i = 0; i < uiLen; i++ )
                        {
                                if( szCadena[ i ] == szCarcter[ 0 ] )
                                {
                                        // Asigna el contador al ITEM que vamos a devolver
                                        hb_itemPutNInt( nRes, i + 1 );
                                        break;
```

```
                                }
                        }
                }
        }

        // Devuelve el ITEM con el numero del resultado
        hb_itemReturnRelease( nRes );

}

/*
 * Uso de parametros por referencia
 * Recibe 2 numeros enteros devuelve la suma, resta, producto y division de los numeros pasados
 */

HB_FUNC( CALCULA )
{
        // Recupera los dos primeros parametros como numeros enteros
        PHB_ITEM nI1 = hb_param( 1, HB_IT_INTEGER );
        PHB_ITEM nI2 = hb_param( 2, HB_IT_INTEGER );

        // Comprueba los datos pasados
        if( nI1 && nI2 )
        {
                // Se recupera el valor de tipo C desde el ITEM
                int i1 = hb_itemGetNI( nI1 );
                int i2 = hb_itemGetNI( nI2 );
                // Recupera el resto de parametros por referencia
                PHB_ITEM nI3 = hb_param( 3, HB_IT_BYREF );
                PHB_ITEM nI4 = hb_param( 4, HB_IT_BYREF );
                PHB_ITEM nI5 = hb_param( 5, HB_IT_BYREF );
                PHB_ITEM nI6 = hb_param( 6, HB_IT_BYREF );
                // Asigna valores a los parametros
                hb_itemPutNInt( nI3, i1 + i2 );
                hb_itemPutNInt( nI4, i1 - i2 );
                hb_itemPutNInt( nI5, i1 * i2 );
                hb_itemPutND( nI6, ( double ) i1 / i2 );
        }
}

/*
 * Comprueba si una nota está aprobada o no y devuelve un literal con el resultado
 * por referencia
 */

HB_FUNC( ESACTO )
{
        PHB_ITEM nNota = hb_param( 1, HB_IT_NUMERIC );
        PHB_ITEM cRes = hb_param( 2, HB_IT_BYREF );

        if( nNota )
        {
                if( hb_itemGetNInt( nNota ) > 4 )
                {
                        hb_itemPutC( cRes, "APROBADO" );
                }
                else
                {
                        hb_itemPutC( cRes, "SUSPENDIDO" );
                }
        }
        else
        {
```

```
                hb_itemPutC( cRes, "NO SE HA PASADO UN DATO CORRECTO" );
        }
}

/*
 * Incrementa en 100 el segundo elemento de un array pasado
 */

HB_FUNC( CAMBIAVALOR )
{
        PHB_ITEM aArray = hb_param( 1, HB_IT_ARRAY );
        PHB_ITEM nVal = hb_itemPutND( NULL, 0.0 );

        if( aArray )
        {
                double dNum = hb_itemGetND( hb_itemArrayGet( aArray, 2 ) );

                hb_itemPutND( nVal, dNum + 100 );
        }

        hb_itemArrayPut( aArray, 2, nVal );
}

/*
 * Devuelve la suma de los elementos numericos de un array pasado
 */

HB_FUNC( SUMAARRAY )
{

        PHB_ITEM aArray = hb_param( 1, HB_IT_ARRAY );
        double dTotal = 0.0;

        if( aArray )
        {
                HB_SIZE i;
                HB_SIZE nLen = hb_itemSize( aArray ); // Número de elementos
                PHB_ITEM elemento = NULL;

                for( i = 1; i <= nLen; i++ )
                {
                        elemento = hb_itemArrayGet( aArray, i );

                        switch( hb_itemType( elemento ) )
                        {
                                case HB_IT_INTEGER :
                                case HB_IT_LONG :
                                        dTotal += hb_itemGetNInt( elemento );
                                        break;

                                case HB_IT_DOUBLE :
                                        dTotal += hb_itemGetND( elemento );
                                        break;
                                        // El resto de tipos no numericos no se consideran
                        }
                }
        }

        hb_itemReturnRelease( hb_itemPutND( NULL, dTotal ) );
}

/*
```

```
 * Tabla de multiplicar del numero entero pasado
 */

HB_FUNC( TABLA )
{
        PHB_ITEM nNum = hb_param( 1, HB_IT_INTEGER );

        if( nNum )
        {
                HB_UINT i;
                HB_UINT n = 10;
                HB_UINT iNum = hb_itemGetNI( nNum );
                PHB_ITEM aTabla = hb_itemArrayNew( 10 );
                PHB_ITEM nMul = hb_itemNew( NULL );

                for( i = 1; i <= n; i++ )
                {
                        hb_itemPutNInt( nMul, iNum * i );

                        hb_itemArrayPut( aTabla, i, nMul );
                }

                hb_itemReturnRelease( aTabla );
        }
        else
        {
                hb_ret();
        }
}

/*
 * Dia de la semana
 * Uso de la funcion interna hb_xstrcpy() y de hb_retc_buffer()
 */

HB_FUNC( DIASEMANA )
{
        PHB_ITEM nDia = hb_param( 1, HB_IT_INTEGER );
        PHB_ITEM cRes = hb_itemNew( NULL );

        if( nDia )
        {
                unsigned int uiDia = hb_itemGetNI( nDia ); // Obtiene el entero del ITEM

                if( uiDia >= 1 && uiDia <= 7 ) // Comprueba los topes
                {
                        // Aquí se guarda el literal del dia de la semana
                        // es una buena prctica iniciar los puntero a NULL
                        char *szDia = NULL;

                        // La funcion interna hb_xstrcpy reserva memoria si el primer parametro es NULL
                        switch( uiDia )
                        {
                                case 1 :
                                        szDia = hb_xstrcpy( NULL, "Lunes", NULL );
                                        break;

                                case 2 :
                                        szDia = hb_xstrcpy( NULL, "Martes", NULL );
                                        break;

                                case 3 :
```

```
                                        szDia = hb_xstrcpy( NULL, "Miercoles", NULL );
                                        break;

                                case 4 :
                                        szDia = hb_xstrcpy( NULL, "Jueves", NULL );
                                        break;

                                case 5 :
                                        szDia = hb_xstrcpy( NULL, "Viernes", NULL );
                                        break;

                                case 6 :
                                        szDia = hb_xstrcpy( NULL, "Sabado", NULL );
                                        break;

                                case 7 :
                                        szDia = hb_xstrcpy( NULL, "Domingo", NULL );
                        }

                        hb_itemPutC( cRes, szDia ); // Asigna literal al item para devolverlo
                }
        }

        // Devuelve y libera
        hb_itemReturnRelease( cRes );
}

/*
 * Definicion de Persona como estructura.
 */

typedef struct
{
        char szNIF[ 10 ];
        char szNombre[ 20 ];
        HB_UINT uiCodigo;
        float nSalrio;
        char szFecha[ 9 ];
        HB_BOOL bSoltero;
} TPersona;

/*
 * Funcion que devuelve una estructura como un array
 */

HB_FUNC( DAMEPERSONA )
{
        PHB_ITEM nPersona = hb_param( 1, HB_IT_INTEGER ); // Parametro pasado
        PHB_ITEM aPersona = hb_itemArrayNew( 6 ); // Array que se va a devolver
        PHB_ITEM wItem = hb_itemNew( NULL ); // Iten de trabajo
        TPersona *persona = hb_xgrab( sizeof( TPersona ) ); //Reserva memoria para TPersona

        // Rellena todos los miembros de TPersona según su tipo
        if( hb_itemGetNI( nPersona ) == 1 )
        {
                hb_xstrcpy( persona->szNIF, "53320105T", NULL );
                hb_xstrcpy( persona->szNombre, "Viruete", ", ", "Paco", NULL );
                persona->uiCodigo = 26212;
                persona->nSalrio = 3500.97;
                hb_xstrcpy( persona->szFecha, "19561225", NULL );
                persona->bSoltero = HB_FALSE;
        }
```

```
        else
        {
                hb_xstrcpy( persona->szNIF, "43310009H", NULL );
                hb_xstrcpy( persona->szNombre, "Grande", ", ", "Felix", NULL );
                persona->uiCodigo = 13101;
                persona->nSalrio = 2750.75;
                hb_xstrcpy( persona->szFecha, "19660213", NULL );
                persona->bSoltero = HB_TRUE;
        }

        // Rellena el array creado
        hb_itemPutC( wItem, persona->szNIF ); // Cadena
        hb_itemArrayPut( aPersona, 1, wItem );
        hb_itemPutC( wItem, persona->szNombre ); // Cadena
        hb_itemArrayPut( aPersona, 2, wItem );
        hb_itemPutNI( wItem, persona->uiCodigo ); // Entero
        hb_itemArrayPut( aPersona, 3, wItem );
        hb_itemPutND( wItem, persona->nSalrio ); // Real
        hb_itemArrayPut( aPersona, 4, wItem );
        hb_itemPutDS( wItem, persona->szFecha ); // Fecha como cadena
        hb_itemArrayPut( aPersona, 5, wItem );
        hb_itemPutL( wItem, persona->bSoltero ); // Booleano
        hb_itemArrayPut( aPersona, 6, wItem );

        // ATENCION: toda la memoria que hayamos reservado la tenemos que liberar
        hb_xfree( persona );
        hb_itemRelease( wItem );
        // Devuelve el array y lo libera
        hb_itemReturnRelease( aPersona );
}

/*
 * Lo mismo que la anterior pero usando un objeto en vez de un array
 */

HB_FUNC( DAMEOBJPERSONA )
{
        PHB_ITEM oPersona = hb_param( 1, HB_IT_OBJECT ); // Parametro pasado
        PHB_ITEM nPersona = hb_param( 2, HB_IT_INTEGER ); // Parametro pasado
        PHB_ITEM wItem = hb_itemNew( NULL ); // Iten de trabajo
        TPersona *persona = hb_xgrab( sizeof( TPersona ) ); //Reserva memoria para TPersona

        // Rellena todos los miembros de TPersona según su tipo
        if( hb_itemGetNI( nPersona ) == 1 )
        {
                hb_xstrcpy( persona->szNIF, "53320105T", NULL );
                hb_xstrcpy( persona->szNombre, "Viruete", ", ", "Paco", NULL );
                persona->uiCodigo = 26212;
                persona->nSalrio = 3500.97;
                hb_xstrcpy( persona->szFecha, "19561225", NULL );
                persona->bSoltero = HB_FALSE;
        }
        else
        {
                hb_xstrcpy( persona->szNIF, "43310009H", NULL );
                hb_xstrcpy( persona->szNombre, "Grande", ", ", "Felix", NULL );
                persona->uiCodigo = 13101;
                persona->nSalrio = 2750.75;
                hb_xstrcpy( persona->szFecha, "19660213", NULL );
                persona->bSoltero = HB_TRUE;
        }

        // Rellena el array creado
```

```
        hb_itemPutC( wItem, persona->szNIF ); // Cadena
        hb_itemArrayPut( oPersona, 1, wItem );
        hb_itemPutC( wItem, persona->szNombre ); // Cadena
        hb_itemArrayPut( oPersona, 2, wItem );
        hb_itemPutNI( wItem, persona->uiCodigo ); // Entero
        hb_itemArrayPut( oPersona, 3, wItem );
        hb_itemPutND( wItem, persona->nSalrio ); // Real
        hb_itemArrayPut( oPersona, 4, wItem );
        hb_itemPutDS( wItem, persona->szFecha ); // Fecha como cadena
        hb_itemArrayPut( oPersona, 5, wItem );
        hb_itemPutL( wItem, persona->bSoltero ); // Booleano
        hb_itemArrayPut( oPersona, 6, wItem );

        // ATENCION: toda la memoria que hayamos reservado la tenemos que liberar
        hb_xfree( persona );
        hb_itemRelease( wItem );
}


/*
 * Da la vuenta a una cadena pasada
 */

HB_FUNC( REVERSE )
{
        PHB_ITEM cInString = hb_param( 1, HB_IT_STRING );
        PHB_ITEM cRes = hb_itemNew( NULL );

        if( cInString )
        {
                const char *szInString = hb_itemGetC( cInString );
                int iLen = hb_itemSize( cInString );
                char *szRetStr = hb_xgrab( iLen );
                int i;

                for( i = 0; i < iLen; i++ )
                {
                        szRetStr[ i ] = szInString[ iLen - i - 1 ];
                }

                hb_itemPutCL( cRes, szRetStr, iLen );
        }

        hb_itemReturnRelease( cRes );

}

/*
 * Usando interfaces a funciones de C hechas por nosotros mismos
 */

char *cstrtran( const char *cString, HB_SIZE nLenStr, const char *cFind, HB_SIZE nLenFind,
                                const char *cReplace, HB_SIZE nLenRep )
{
        HB_SIZE i, n, w = 0;
        HB_BOOL fFind = HB_FALSE;
        char *cRet = ( char * ) hb_xgrab( nLenStr + 1 );

        for( i = 0; i < nLenStr; i++ )
        {
                for( n = 0; n < nLenFind; n++ )
                {
```

```
                        fFind = cFind[ n ] == cString[ i ];

                        if( fFind )
                        {
                                if( n < nLenRep )
                                {
                                        cRet[ w ] = cReplace[ n ];
                                        w++;
                                }

                                break;
                        }
                }

                if( !fFind )
                {
                        cRet[ w ] = cString[ i ];
                        w++;
                }
        }

        cRet[ w ] = '\0';

        return( cRet );
}

/*
 * Esta es la funcion interfaz entre cstrtran() en C puro y el PRG
 */

HB_FUNC( CSTRTRAN )
{
        PHB_ITEM cString = hb_param( 1, HB_IT_STRING );
        PHB_ITEM cFind = hb_param( 2, HB_IT_STRING );
        PHB_ITEM cReplace = hb_param( 3, HB_IT_STRING );

        PHB_ITEM cRes = hb_itemPutC( NULL, "" );

        if( cString && cFind && cReplace )
        {
                hb_itemPutC( cRes, cstrtran( hb_itemGetC( cString ),
                                             hb_itemGetCLen( cString ),
                                             hb_itemGetC( cFind ), hb_itemGetCLen( cFind ),
                                             hb_itemGetC( cReplace ), hb_itemGetCLen( cReplace ) ) );
        }

        hb_itemReturnRelease( cRes );
}

/*
 * Usando interfaces a funciones de C
 */

HB_FUNC( STRTOK )
{
        PHB_ITEM p = hb_param( 1, HB_IT_STRING );
        PHB_ITEM s = hb_param( 2, HB_IT_STRING );
        PHB_ITEM cRes = hb_itemNew( NULL );

        if( p && s )
        {
                hb_itemPutC( cRes, strtok( hb_itemGetC( p ), hb_itemGetC( s ) ) );
```

```
        }

        hb_itemReturnRelease( cRes );
}

/*
 * Interfaces con funciones matematicas de C
 */

#include <math.h>

/* coseno */
HB_FUNC( C_COS )
{
        PHB_ITEM nRes = hb_itemNew( NULL );
        PHB_ITEM nPar = hb_param( 1, HB_IT_NUMERIC );

        if( nPar )
        {
                hb_itemPutND( nRes, cos( hb_itemGetND( nPar ) ) );
        }

        hb_itemReturnRelease( nRes );
}
/* seno */
HB_FUNC( C_SIN )
{
        PHB_ITEM nRes = hb_itemNew( NULL );
        PHB_ITEM nPar = hb_param( 1, HB_IT_NUMERIC );

        if( nPar )
        {
                hb_itemPutND( nRes, sin( hb_itemGetND( nPar ) ) );
        }

        hb_itemReturnRelease( nRes );
}
/* tangente */
HB_FUNC( C_TAN )
{
        PHB_ITEM nRes = hb_itemNew( NULL );
        PHB_ITEM nPar = hb_param( 1, HB_IT_NUMERIC );

        if( nPar )
        {
                hb_itemPutND( nRes, tan( hb_itemGetND( nPar ) ) );
        }

        hb_itemReturnRelease( nRes );
}
```

Como se ha podido comprobar a nivel de PRG no cambia nada se use el *ITEM API* o el ***Sistema Extendido***.

A continuación vamos a ver algo completamente nuevo en *Harbour* son las extensiones del *ITEM API* para manejar *Arrays* y *Tablas Hash…*

Tratamiento de arrays.

Las funciones básicas para el tratamiento de arrays son las tres ya explicadas:

```
PHB_ITEM hb_itemArrayNew( HB_SIZE nLen );
PHB_ITEM hb_itemArrayGet( PHB_ITEM pArray, HB_SIZE nIndex );
PHB_ITEM hb_itemArrayPut( PHB_ITEM pArray, HB_SIZE nIndex, PHB_ITEM pItem );
```

El problema es que estas funciones solo trabajan con *ITEM*, o sea los parámetros desde C pasados y el resultado será *ITEM*. Realmente es como funcionan todas las funciones del ***ITEM API***.
Si queremos asignar un valor a un miembro de un array tenemos que hacerlo de esta manera:

```
…
hb_itemArrayPut( aMiArray, 2, cValor );
…
```

`aMiArray` → es el array al que le vamos a modificar el valor

2 → es la posición que vamos a modificarla

`cValor` → es el *ITEM* que vamos a asignar

O sea cValor es un *ITEM* y no un valor de tipo C como por ejemplo un *char* o una cadena (*char* *)
Si lo que tenemos es una cadena de C que queremos asignar a la segunda posición de nuestro array tendríamos que crear un *ITEM* asignarme la cadena y luego usar la función hb_itemArrayPut() para asignar el *ITEM*:

```
…
PHB_ITEM cValor = hb_itemNew( NULL );
…
hb_itemPutC( cValor, “Esto es una cadena” );
hb_itemArrayPut( aMiArray, 2, cValor );
…
```

Y claro, esto es un poco tedioso.
Debería haber algo que pudiera tratar directamente al array y que se encargue de crear el *ITEM* con el valor de la cadena y asignarlo de una sola vez…
La buena noticia es que existe y eso es lo que vamos a ver ahora.
Son funciones del ***ITEM API*** que se encarga de crear el ITEM que se va asignar a partir del valor tipo C. Como es lógico las hay para asigna y para extraer el valor en tipo C.

Estas son las genéricas de mantenimiento del array:

HB_BOOL hb_arrayNew(PHB_ITEM pItem, HB_SIZE nLen);
Convierte ***pItem*** en un *ITEM* de tipo array con ***nLen*** elementos, ***pItem*** tiene que ser un *ITEM* creado previamente.

HB_BOOL hb_arrayAdd(PHB_ITEM pArray, PHB_ITEM pItemValue);

Añade un nuevo *ITEM* al final del array incrementando el tamaño en uno. Funciona como la función PRG *AAdd()*.

HB_BOOL hb_arrayIns(PHB_ITEM pArray, HB_SIZE nIndex);

Inserta un *ITEM* vacío en la posición indicada por *nIndex*, ojo desplaza el resto de elementos del array y el último se trunca. Funciona como la función PRG *AIns()*.

HB_BOOL hb_arrayDel(PHB_ITEM pArray, HB_SIZE nIndex);

Elimina el elemento del array que ocupa *nIndex* y desplaza el resto de elemento una posición menos. Deja el último elemento como nulo que puede ser rellenado. Funciona como la función PRG *ADel()*.

HB_BOOL hb_arraySize(PHB_ITEM pArray, HB_SIZE nLen);

Reajusta el tamaño del array. Si nLen es mayor que el número de elementos crea la diferencia si nLen es menor elimina la diferencia. Funciona como la función PRG *ASize()*.

HB_BOOL hb_arrayLast(PHB_ITEM pArray, PHB_ITEM pResult);

Pone en *pResult* el ultimo elemento del array.

HB_BOOL hb_arrayGet(PHB_ITEM pArray, HB_SIZE nIndex, PHB_ITEM pItem);

Funciona como la función básica de manejo de arrays ***hb_itemArrayGet()***, la diferencia es que ***hb_itemArrayGet()*** crea y devuelve una copia del elemento de la posición nIndex y **hb_arrayGet()** copia el elemento en pItem que tiene que estar creado previamente.

HB_BOOL hb_arraySet(PHB_ITEM pArray, HB_SIZE nIndex, PHB_ITEM pItem);

Asigna pItem al elemento de pArray ubicado en la posición nIndex. Es identica a ***hb_itemArrayPut()*** con la diferencia que ***hb_itemArrayPut()*** devuelve el propio array pArray y hb_arraySet() devuelve un valor lógico indicando si se pudo hacer la asignación o no. Esta función es la equivalente a hacer en PRG:

```
pArray[ nIndex ] := pItem
```

Se puede ver que todas las funciones anteriores devuelven un valor lógico indicando si se pudo asignar el valor.

HB_TYPE hb_arrayGetType(PHB_ITEM pArray, HB_SIZE nIndex);

Devuelve un entero con el tipo de dato del elemento.

HB_SIZE hb_arrayLen(PHB_ITEM pArray);

Devuelve el numero de elementos del array. Funciona como Len(aArray).

Ahora vamos a analizar las funciones que obtienen el valor como tipo de datos C que contiene el ITEM que ocupa la posición indicada por nIndex:

char *hb_arrayGetC(PHB_ITEM pArray, HB_SIZE nIndex);
Devuelve cadena contenida en el elemento nIndex del array.

HB_SIZE hb_arrayGetCLen(PHB_ITEM pArray, HB_SIZE nIndex);
Devuelve el tamaño de la cadena contenida en el elemento nIndex del array..

HB_BOOL hb_arrayGetL(PHB_ITEM pArray, HB_SIZE nIndex);
Devuelve el valor lógico del elemento nIndex del array.

int hb_arrayGetNI(PHB_ITEM pArray, HB_SIZE nIndex);
Devuelve el valor entero del elemento nIndex del array.

long hb_arrayGetNL(PHB_ITEM pArray, HB_SIZE nIndex);
Devuelve el valor entero largo del elemento nIndex del array.

HB_MAXINT hb_arrayGetNInt(PHB_ITEM pArray, HB_SIZE nIndex);
Devuelve el valor entero largo con el mayor ancho disponible del elemento nIndex del array.

double hb_arrayGetND(PHB_ITEM pArray, HB_SIZE nIndex);
Devuelve el valor numero real del elemento nIndex del array.

char *hb_arrayGetDS(PHB_ITEM pArray, HB_SIZE nIndex, char * szDate);
Devuelve el valor tipo fecha como cadena del elemento nIndex del array.

long hb_arrayGetDL(PHB_ITEM pArray, HB_SIZE nIndex);
Devuelve el valor tipo fecha como entero largo del elemento nIndex del array.

Ahora vamos a analizar las funciones que asignan el valor como tipo de datos C en el ITEM que ocupa la posición indicada por nIndex, Estas funciones devuelven un valor lógico indicando si se consiguió o no la asignación.

HB_BOOL hb_arraySetDS(PHB_ITEM pArray, HB_SIZE nIndex, const char *szDate);
Asigna una fecha *szDate* como cadena (yyyymmdd) en la posición *nIndex*.

HB_BOOL hb_arraySetDL(PHB_ITEM pArray, HB_SIZE nIndex, long lDate);
Asigna una fecha *lDate* como un entero largo fecha juliana en la posición *nIndex*.

HB_BOOL hb_arraySetL(PHB_ITEM pArray, HB_SIZE nIndex, HB_BOOL fValue);
Asigna *fValue* como valor lógico en la posición *nIndex*

HB_BOOL hb_arraySetNI(PHB_ITEM pArray, HB_SIZE nIndex, int iNumber);
Asigna un número *iNumber* como entero en la posición *nIndex*

HB_BOOL hb_arraySetNL(PHB_ITEM pArray, HB_SIZE nIndex, long lNumber);
Asigna un número *lNumber* como entero largo en la posición *nIndex*

HB_BOOL hb_arraySetNInt(PHB_ITEM pArray, HB_SIZE nIndex, HB_MAXINT nNumber);
Asigna un número *nNumber* como entero de máxima capacidad en la posición *nIndex*

HB_BOOL hb_arraySetND(PHB_ITEM pArray, HB_SIZE nIndex, double dNumber);
Asigna un número *dNumber* como real en la posición *nIndex*

HB_BOOL hb_arraySetC(PHB_ITEM pArray, HB_SIZE nIndex, const char *szText);
Asigna una cadena *szText* en la posición *nIndex*

Este primer ejemplo modifica cada uno de los elementos de un vector con el valor del segundo parámetro. Esto además muestra como los arrays por defecto se pasan por referencia. Una segunda parte se ve como se crea y rellena un array con diferentes tipos de datos:

```
/*
 * Uso de funciones en de C para tratar arrays
 */

procedure main

    local aDesdeC
    local aArray := { 12.30, 11, 20, 3, 23, 89, 5, 15, 33.75, 1.98 }
    local nLen := Len( aArray )
    local nSuma := 100.55       // Prueba cambiar el numero

    set date format to "dd-mm-yyyy"

    cls

    // Esta funcion en C suma a lo elementos del array el valor del 2 parametro
    tramita( aArray, nSuma )

    ?
    ? "El valor de sumando:", nSuma
    ? "------------------------------------------------------------------------"
    ? "Valores anteriores", "Valores nuevos"
    ?

    for i := 1 to nLen
        ? aArray[ i ] - nSuma, aArray[ i ]
    next

    ?
    ? "Presiona una tecla para seguir..."
    Inkey( 100 )

    cls
```

```
    ?
    ? "------------------------------------------------------------------------"
    ? "Array creado desde C:"
    ?

    aDesdeC := creaArray()
    nLen := Len( aDesdeC )

    for i := 1 to nLen
        ? "Elemento " + HB_NToS( i ), aDesdeC[ i ]
    next

    Inkey( 100 )

return
```

Y esta es la funciones en C para el ejemplo:

```
/*
 * Funcion que recibe un array de numeros y los incrementa con
 * el valor numerico pasado como segundo parametro.
 */

 HB_FUNC( TRAMITA )
 {
         PHB_ITEM aDatos = hb_param( 1, HB_IT_ARRAY );

         if( aDatos )
         {
                PHB_ITEM nMod = hb_param( 2, HB_IT_NUMERIC );

                if( nMod )
                {
                        HB_SIZE nLen = hb_arrayLen( aDatos );
                        HB_SIZE i;
                        double dMod = hb_itemGetND( nMod );

                        for( i = 1; i <= nLen; i++ )
                        {
                                hb_arraySetND( aDatos, i, hb_arrayGetND( aDatos, i ) + dMod );
                        }
                }
         }
 }

/*
 * Crea un array
 */

HB_FUNC( CREAARRAY )
{
        PHB_ITEM aArray = hb_itemNew( NULL ); //Crea un ITEM vacio
        PHB_ITEM xItem = hb_itemNew( NULL );

        // Convertimo el ITEM creado como un array de 10 elementos. Se podia haber
        // usado aArray = hb_itemArrayNew( 10 ) que hace los dos pasos a la vez
        hb_arrayNew( aArray, 10 );

        hb_arraySetDS( aArray, 1, "20211231" );
        // La funcion hb_arraySetDL necesita la fecha juliana. Para convertir la como
        // cadena a juliana usamos la función interna hb_dateEncStr
        hb_arraySetDL( aArray, 2, hb_dateEncStr( "20211231" ) );
        hb_arraySetL( aArray, 3, HB_FALSE );
        hb_arraySetNI( aArray, 4, 35789 );
        hb_arraySetNL( aArray, 5, 15578952 );
        hb_arraySetNInt( aArray, 6, 95415545541 );
        hb_arraySetND( aArray, 7, 34954155455.41 );
        hb_arraySetC( aArray, 8, "Esto es una prueba de cadena" );
        hb_arraySet( aArray, 9, xItem ); // xItem sin asignar
        // Asignamos un valor al item creado xItem y lo metemos en el array
        hb_itemPutC( xItem, "Asignacion del item con valor tipo cadena");
```

```
        hb_arraySet( aArray, 10, xItem );

        hb_itemRelease( xItem ); // Destruye el item creado

        hb_itemReturnRelease( aArray ); // Devuelve el array y destruye el item
}
```

Otro ejemplo practico con un buffer para el registro de una DBF basado en array, aquellas famosas funciones Gather y Scatter:

```
/*
 * Uso de funciones en de C para tratar array
 */

procedure main

    local aBuf

    cls

    use test new

    test->( DBGoTo( 10 ) )
    aBuf := test->( aGather() ) // Lee el registro actual

    ? "------------------------------------------------------------------------"
    ? "Tipo: " + ValType( aBuf ), "  /  Num. elementos: " + HB_NToS( Len( aBuf ) )
    ? "------------------------------------------------------------------------"
    ? test->( RecNo() ), aBuf[ 2 ], aBuf[ 9 ]
    ? "------------------------------------------------------------------------"
    ? "Cambiamos la edad con un entero aleatorio"
    aBuf[ 9 ] := HB_RandomInt( 0, 99 )
    ? "El valor en buffer es:", aBuf[ 9 ]
    test->( aScatter( aBuf ) )  // Escribe en el registro
    ? "------------------------------------------------------------------------"
    ? "La edad en la DBF es ahora", test->( FieldGet( 9 ) )
    ? "------------------------------------------------------------------------"
    Inkey( 100 )

return
```

y ahora las dos funciones en C:

```
/*
 * Crea el array y lo devuelve relleno con el registro actual
 */
HB_FUNC( AGATHER )
{
        PHB_ITEM aBuffer = hb_itemNew( NULL );
        PHB_ITEM pValue = hb_itemNew( NULL );
        AREAP pArea = ( AREAP ) hb_rddGetCurrentWorkAreaPointer();
        HB_USHORT uiFields, i;

        // Cuenta el numero de campos de la dbf
        SELF_FIELDCOUNT( pArea, &uiFields );

        hb_arrayNew( aBuffer, uiFields );

        // Relleno el aBuffer con los datos del registro
        for( i = 1; i <= uiFields; i++ )
        {
                SELF_GETVALUE( pArea, i, pValue );
                hb_arraySet( aBuffer, i, pValue );
        }

        // Libero todo lo que he creado
```

```
        hb_itemRelease( pValue );

        // Devuelvo y libero el aBuffer relleno
        hb_itemReturnRelease( aBuffer );
}

/*
 * Salva el array en el registro actual
 */
HB_FUNC( ASCATTER )
{
        PHB_ITEM aBuffer = hb_param( 1, HB_IT_ARRAY );

        if( aBuffer )
        {
                AREAP pArea = ( AREAP ) hb_rddGetCurrentWorkAreaPointer();
                HB_USHORT uiFields, i;
                PHB_ITEM pValue = hb_itemNew( NULL );

                SELF_FIELDCOUNT( pArea, &uiFields );

                for( i = 1; i <= uiFields; i++ )
                {
                        hb_arrayGet( aBuffer, i, pValue );
                        SELF_PUTVALUE( pArea, i, pValue );
                }

                // Libero todo lo que he creado
                hb_itemRelease( pValue );
        }
}
```

Nota: El tema de tratamiento de fechas es un mundo. Sólo decir que como en la mayoría de los lenguajes de programación la manera de guardarla en archivos como las DBF es como un entero largo, o sea como una fecha juliana.

Tratamiento de Tablas Hash.

Realmente deberíamos hablar de ***Arrays Asociativos*** más que T***ablas Hash***, no obstante no voy a entrar en ninguna discusión al respecto.

¿Cual es el concepto de este tipo de dato de Harbour?

Es un contenedor de datos como los arrays pero que está constituido por la dupla ***(llave, valor)***. La principal características es la velocidad de búsqueda por esa clave.

Seguramente no será necesario usar los hash de Harbour a nivel de C pero vamos a ver las principales funciones. Estas funciones aceptaran un primer parámetro que será siempre el ITEM hash `pHash` y normalmente otros dos ITEM, la key pKey y el valor pValue. Repito *son ITEM los tres (o sea variables normales de Harbour a nivel de PRG) y no variables de tipo C*, por lo que tendremos siempre que asignar el valor de C a un ITEM previamente creado antes de meterlos en la Tablas Hash.

PHB_ITEM hb_hashNew(PHB_ITEM pItem);

Esta función o crea un hash nuevo si se le pasa NULL o cambia a tipo hash el item pasado. En ambos caso devuelve el ITEM como un hash limpio.

HB_SIZE hb_hashLen(PHB_ITEM pHash);
Devuelve la capacidad de la tabla hash o sea el numero de elementos disponibles usados o no.

HB_BOOL hb_hashDel(PHB_ITEM pHash, PHB_ITEM pKey);
Elimina el par cuya clave es pKey si existe.

HB_BOOL hb_hashAdd(PHB_ITEM pHash, PHB_ITEM pKey, PHB_ITEM pValue);
Si existe la clave pKey modifica el valor pValue y si no existe pKey añade el nuevo par.

HB_BOOL hb_hashAddNew(PHB_ITEM pHash, PHB_ITEM pKey, PHB_ITEM pValue);
Sólo añade un par si no existe la clave pKey.

HB_BOOL hb_hashRemove(PHB_ITEM pHash, PHB_ITEM pItem);
Función para eliminar pares como *hb_hashDel*, la difencia es que se le puede pasar una clave, un array de claves o un hash. En el caso de las dos últimas opciones hará un ***hb_hashDel()*** de cada elemento.

HB_BOOL hb_hashClear(PHB_ITEM pHash);

Elimina todos los pares del hash y pone la capacidad a 0.

void hb_hashSort(PHB_ITEM pHash);
Ordena el hash por la clave.

PHB_ITEM hb_hashClone(PHB_ITEM pHash);

Crea una tabla hash idéntica al a la pasada pHash.

PHB_ITEM hb_hashCloneTo(PHB_ITEM pDest, PHB_ITEM pHash);
Crea una tabla hash idéntica al a la pasada pHash en el item pasado pDest. Es como ***hb_hashClone()*** pero no crea el item si no que hay que pasarlo.

void hb_hashJoin(PHB_ITEM pDest, PHB_ITEM pSource, int iType);
Une dos hash en el primero pasado. Esa unión puede ser de varios tipo usando operadores lógicos:
HB_HASH_UNION: OR
HB_HASH_INTERSECT: AND
HB_HASH_DIFFERENCE: XOR
HB_HASH_REMOVE: NOT -> h1 AND (h1 XOR h2)

HB_BOOL hb_hashScan(PHB_ITEM pHash, PHB_ITEM pKey, HB_SIZE * pnPos);
Devuelve un valor lógico dependiendo si encontró la clave pKey en el hash. Además pone en pnPos (pasado por referencia, o sea, un puntero) la posición ocupada por pKey si ha sido encontrada o 0 si no la encontró.

HB_BOOL hb_hashScanSoft(PHB_ITEM pHash, PHB_ITEM pKey, HB_SIZE * pnPos);
Devuelve un valor lógico dependiendo si encontró la clave pKey en el hash. Además pone en pnPos (pasado por referencia, o sea, un puntero) la posición ocupada por pKey si ha sido encontrada o la posición anterior más cercana si no la encontró.

void hb_hashPreallocate(PHB_ITEM pHash, HB_SIZE nNewSize);
Modifica hash con la capacidad determinada por nNewSize.

PHB_ITEM hb_hashGetKeys(PHB_ITEM pHash);
Devuelve un array item con todas las claves del hash.

PHB_ITEM hb_hashGetValues(PHB_ITEM pHash);
Devuelve un array item con todos los valores del hash.

void hb_hashSetDefault(PHB_ITEM pHash, PHB_ITEM pValue);

Determina el valor por defecto de los valores, normalmente es nil.

PHB_ITEM hb_hashGetDefault(PHB_ITEM pHash);
Obtiene el actual valor por defecto de los valores.

void hb_hashSetFlags(PHB_ITEM pHash, int iFlags);
Determina el comportamiento del mantenimiento del hash por el iFlags pasado puede ser una combinación (con el operador lógico | or) de estos:

```
HB_HASH_AUTOADD_NEVER
HB_HASH_AUTOADD_ACCESS
HB_HASH_AUTOADD_ASSIGN
HB_HASH_AUTOADD_ALWAYS
HB_HASH_AUTOADD_REFERENCE
HB_HASH_AUTOADD_MASK
HB_HASH_RESORT
HB_HASH_IGNORECASE
HB_HASH_BINARY
HB_HASH_KEEPORDER
HB_HASH_FLAG_MASK
HB_HASH_FLAG_DEFAULT
```

void hb_hashClearFlags(PHB_ITEM pHash, int iFlags);
Elimina el tipo o tipos pasados (con el operador logico | or) en iFlags.

int hb_hashGetFlags(PHB_ITEM pHash);

Obtiene los tipos de flags activos.

void *hb_hashId(PHB_ITEM pHash);

Recupera el ID único del hash.

HB_COUNTER hb_hashRefs(PHB_ITEM pHash);

Recupera el número de referencias al hash.

Ejemplo básico de creación de un hash en C y tratarlo en PRG:

```
/*
 * Uso de funciones en de C para tratar hash
 */

procedure main

    local hTabla, xClave, xValor, xPar
    local nLen, i

    cls

    ?
    ? "-------------------------------------------------------------------------"
    ? "Hash creado desde C:"
    ? "-------------------------------------------------------------------------"
    ?

    hTabla := creaHash()
    nLen := Len( hTabla )

    hb_HCaseMatch( hTabla, .f. )  // Le da igual mayuscula o minusculas

    ?
    ? "El hash tiene " + HB_NToS( nLen ) + " pares ( clave, valor )"
    ?
    ? "-------------------------------------------------------------------------"
    ? "Lo mostramos usando un for next:"
    ? "-------------------------------------------------------------------------"
    for i := 1 to nLen
        xClave := hb_HKeyAt( hTabla, i )
        ? "Clave: ", xClave, " -> Valor: ", hb_HGet( hTabla, xClave )
    next
    ?
    ? "-------------------------------------------------------------------------"
    ? "Recomendado: usando un for each:"
    ? "-------------------------------------------------------------------------"

    for each xPar in hTabla
        ? "Clave: ", xPar:__enumKey(), " -> Valor: ", xPar:__enumValue()
    next

    ? "-------------------------------------------------------------------------"
    ?
    Inkey( 100 )

return
```

Y la función en C:

```
/*
 * Crea un hash en C
 */

HB_FUNC( CREAHASH )
{
        PHB_ITEM hHash = hb_hashNew( NULL ); //Crea un ITEM vacio como hash
        PHB_ITEM pKey = hb_itemNew( NULL ); // Como cargo para usarlo como key
        PHB_ITEM pValue = hb_itemNew( NULL ); // Como cargo para usarlo como value

        // Fechas
        hb_itemPutC( pKey, "fechaDS" ); // Fecha como cadena. Esta es la key
        hb_itemPutDS( pValue, "20211231" );
        hb_hashAdd( hHash, pKey, pValue ); // Añadimo el par al hash
        //---
        hb_itemPutC( pKey, "fechaDL" ); // Fecha como cadena. Esta es la key
        hb_itemPutDL( pValue, hb_dateEncStr( "20211231" ) );
        hb_hashAdd( hHash, pKey, pValue ); // Añadimo el par al hash
        // Logicos
        hb_itemPutC( pKey, "logico" );
        hb_itemPutL( pValue, HB_TRUE );
        hb_hashAdd( hHash, pKey, pValue );
        // Enteros
        hb_itemPutC( pKey, "entero" );
        hb_itemPutNI( pValue, 35789 );  // Entero
        hb_hashAdd( hHash, pKey, pValue );
        //---
        hb_itemPutC( pKey, "largo" );
        hb_itemPutNL( pValue, 15578952 );       // Entero largo
        hb_hashAdd( hHash, pKey, pValue );
        // Real
        hb_itemPutC( pKey, "real" );
        hb_itemPutND( pValue, 34954155455.41 ); // Doble o real
        hb_hashAdd( hHash, pKey, pValue );

        hb_itemRelease( pKey ); // Destruye el item creado para la key
        hb_itemRelease( pValue ); // Destruye el item creado para el value

        hb_itemReturnRelease( hHash ); // Devuelve el hash y destruye el item
}
```

Vamos a ser prácticos, en estos ejemplos, vamos a usar las funciones Gather y Scatter pero con un hash como buffer, es el mismo ejemplo de los arrays expuesto anteriormente, tal vez mejore el sistema porque se usa el nombre del campo como clave en vez de la posición que siempre cuesta un poco más memorizar:

```
/*
 * Uso de funciones en de C para tratar hash
 */

procedure main

    local hBuf

    cls

    use test new

    test->( DBGoTo( 10 ) )
    hBuf := test->( hGather() ) // Lee el registro actual

    ? "------------------------------------------------------------------------"
    ? "Tipo: " + ValType( hBuf ), "  /  Num. elementos: " + HB_NToS( Len( hBuf ) )
    ? "------------------------------------------------------------------------"
    ? test->( RecNo() ), hBuf[ "Last" ], hBuf[ "age" ]
    ? "------------------------------------------------------------------------"
    ? "Cambiamos la edad con un entero aleatorio"
```

```
    hBuf[ "age" ] := HB_RandomInt( 0, 99 )
    ? "El valor en buffer es:", hBuf[ "age" ]
    test->( hScatter( hBuf ) )  // Escribe en el registro
    ? "------------------------------------------------------------------------"
    ? "La edad en la DBF es ahora", test->( FieldGet( FieldPos( "age" ) ) )
    ? "------------------------------------------------------------------------"
    Inkey( 100 )

return
```

Y ahora veamos las funciones en C:

```
/*
 * Funciones Gather y Scatter para Hash
 */

/*
 * Crea el Hash y lo devuelve relleno con el registro actual
 */
HB_FUNC( HGATHER )
{
        PHB_ITEM hBuffer = hb_hashNew( NULL );
        PHB_ITEM pKey = hb_itemNew( NULL );
        PHB_ITEM pValue = hb_itemNew( NULL );
        AREAP pArea = ( AREAP ) hb_rddGetCurrentWorkAreaPointer();
        char *szFldName = ( char * ) hb_xgrab( pArea->uiMaxFieldNameLength + 1 );
        HB_USHORT uiFields, i;

        // Quita sensibilidad a mayusculas/minusculas
        hb_hashClearFlags( hBuffer, HB_HASH_BINARY );
        hb_hashSetFlags( hBuffer, HB_HASH_IGNORECASE | HB_HASH_RESORT );

        // Cuenta el numero de campos de la dbf
        SELF_FIELDCOUNT( pArea, &uiFields );

        // Reservo la capacidad del hBuffer
        hb_hashPreallocate( hBuffer, uiFields );

        // Relleno el hBuffer con los datos del registro
        for( i = 1; i <= uiFields; i++ )
        {
                SELF_FIELDNAME( pArea, i, szFldName );
                hb_itemPutC( pKey, szFldName );
                SELF_GETVALUE( pArea, i, pValue );
                hb_hashAdd( hBuffer, pKey, pValue );
        }

        // Libero todo lo que he creado
        hb_itemRelease( pKey );
        hb_itemRelease( pValue );
        hb_xfree( szFldName );

        // Devuelvo y libero el hBuffer relleno
        hb_itemReturnRelease( hBuffer );
}

/*
 * Salva el Hash en el registro actual
 */
HB_FUNC( HSCATTER )
{
        PHB_ITEM hBuffer = hb_param( 1, HB_IT_HASH );

        if( hBuffer )
        {
                AREAP pArea = ( AREAP ) hb_rddGetCurrentWorkAreaPointer();
                HB_USHORT uiFields, i;

                SELF_FIELDCOUNT( pArea, &uiFields );
```

```
                for( i = 1; i <= uiFields; i++ )
                {
                        SELF_PUTVALUE( pArea, i, hb_hashGetValueAt( hBuffer, i ) );
                }
        }
}
```

Ahora un ejemplo muy útil para ahorrar uso de variables en el manejo de campos de una DBF.
La clase *TWABuffer*, métodos:

* *new()p → para inicializar el objeto*
* *load() → para cargar el registro en el buffer*
* *save() → para salvar los valores del buffer al registro actuales*
* *blank() → pone el buffer con valores blancos*
* *getBuffer() → devuelve el hash buffer*
* *set(cFieldName, xValor) → para asignar un valor en el buffer*
* *get(cFieldName) → Para obtener un valor*
* *getAlias() → devuelve el alias asignado al objeto como la función Alias()*
* *getArea() → devuelve el area de trabajo (WA) asignado al objeto como la función Select()*
* *getLen() → devuelve el numero de pares del hash buffer*

Con esta clase veremos un poco como acceder a las funciones de manejo de las RDD desde C.
La clase se puede ampliar con más métodos. Pero he querido hacer sólo una clase que maneje un buffer para ahorrar variables y automatizar el proceso de salvar y recuperar la información de la DBF al buffer de la clase.

Ejemplo de uso, es un mantenimiento completo con altas, modificaciones y bajas de la DBF test.dbf que viene en la carpeta “test” de Harbour:

```
//-----------------------------------------------------------------------------
// Ejercicio uso de funciones C. Ejemplo de uso de la clase TWABuffer
// ej011.prg
//-----------------------------------------------------------------------------

#define B_BOX ( CHR( 218 ) + CHR( 196 ) + CHR( 191 ) + CHR( 179 ) + ;
       CHR( 217 ) + CHR( 196 ) + CHR( 192 ) + CHR( 179 ) + " " )

#define ID_MODIFICA 1
#define ID_ALTA     2

#include "InKey.ch"

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

procedure main()

    local oBuf

    set date format to "dd-mm-yyyy"
    set deleted on
    SetMode( 24, 80 )

    cls

    USE test NEW
```

```
    if test->( Used() )
        oBuf := TWABuffer():New()   // Crea el Work Area Buffer
        GestBrw( oBuf )
        test->( DBCloseArea() )
    endif

return

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

static procedure GestBrw( oBuf )

    local oBrw, oCol
    local nKey := 0
    local n, nFld

    oBrw := TBrowseDb( 1, 0, MaxRow() -1, MaxCol() )

    oBrw:colorSpec := "W+/B, N/BG"
    oBrw:ColSep    := " ³ "
    oBrw:HeadSep   := "ÄÅÄ"
    oBrw:FootSep   := "ÄÁÄ"

    nFld := oBuf:getLen()

    FOR n := 1 TO nFld
        oBrw:AddColumn( TBColumnNew( ( oBuf:getArea() )->( FieldName( n ) ), GenCB( oBuf, n ) ) )
    NEXT

    cls

    @ 0, 0 SAY PadC( "Ojeando la tabla: " + upper( oBuf:getAlias() ), MaxCol() + 1, " " );
           COLOR "W+/G+"

    @ MaxRow(),          0 SAY "INSERT"   COLOR "GR+/R+"
    @ MaxRow(), Col() + 1 SAY "Altas"    COLOR "W+/R+"
    @ MaxRow(), Col() + 1 SAY "ENTER"    COLOR "GR+/R+"
    @ MaxRow(), Col() + 1 SAY "Modifica" COLOR "W+/R+"
    @ MaxRow(), Col() + 1 SAY "SUPR"     COLOR "GR+/R+"
    @ MaxRow(), Col() + 1 SAY "Bajas"    COLOR "W+/R+"
    @ MaxRow(), Col() + 1 SAY "F4"       COLOR "GR+/R+"
    @ MaxRow(), Col() + 1 SAY "      "   COLOR "W+/R+"
    @ MaxRow(), Col() + 1 SAY "F5"       COLOR "GR+/R+"
    @ MaxRow(), Col() + 1 SAY "       "  COLOR "W+/R+"
    @ MaxRow(), Col() + 1 SAY "F6"       COLOR "GR+/R+"
    @ MaxRow(), Col() + 1 SAY "       "  COLOR "W+/R+"
    @ MaxRow(), Col() + 1 SAY "ESC"      COLOR "GR+/R+"
    @ MaxRow(), Col() + 1 SAY "Salir"    COLOR "W+/R+"

    while nKey != K_ESC

        oBrw:ForceStable()

        nKey = InKey( 0 )

        do case
            case nKey == K_DOWN;        oBrw:Down()             // Fila siguiente
            case nKey == K_UP;          oBrw:Up()               // Fila anterior
            case nKey == K_LEFT;        oBrw:Left()             // Va a la columna anterior
            case nKey == K_RIGHT;       oBrw:Right()            // Va a la columna siguiente
            case nKey == K_PGDN;        oBrw:pageDown()         // Va a la pagina siguiente
            case nKey == K_PGUP;        oBrw:pageUp()           // Va a la pagina anterior
            case nKey == K_CTRL_PGUP;   oBrw:goTop()            // Va al principio
            case nKey == K_CTRL_PGDN;   oBrw:goBottom()         // Va al final
            case nKey == K_HOME;        oBrw:home()            // Va a la primera columna visible
            case nKey == K_END;         oBrw:end()              // Va a la ultima columna visible
            case nKey == K_CTRL_LEFT;   oBrw:panLeft()          // Va a la primera columna
            case nKey == K_CTRL_RIGHT;  oBrw:panRight()         // Va a la ultima columna
            case nKey == K_CTRL_HOME;   oBrw:panHome()          // Va a la primera página
            case nKey == K_CTRL_END;    oBrw:panEnd()           // Va a la última página
            case nKey == K_DEL;         Borrar( oBuf, oBrw )    // Borra fila
            case nKey == K_INS;         Insertar( oBuf, oBrw )  // Inserta columna
```

```
            case nKey == K_ENTER;       Modificar( oBuf, oBrw ) // Modifica columna
        endcase

    end

return

//----------------------------------------------------------------------------//
// Crea los codeblock SETGET de las columnas del browse

static function GenCB( oBuf, n ); return( {|| ( oBuf:getArea() )->( FieldGet( n ) ) } )

//----------------------------------------------------------------------------//
// Pantalla de datos de la tabla

static function PantMuestra( oBuf, nTipo )

    local GetList := {}
    local cTipo, cId, hBuffer

    SET CURSOR ON

    if nTipo == ID_ALTA
        cTipo := "Insertando"
        cId := "nuevo"
        hBuffer := oBuf:Blank()
    else // nTipo == ID_MODIFICA
        cTipo := "Modificando"
        cId := HB_NToS( ( oBuf:getArea() )->( RecNo() ) )
        hBuffer := oBuf:Load()
    endif

    DispBox( 3, 2, 18, 74, B_BOX )

    @ 04, 03 SAY cTipo + " registro en tabla " + oBuf:getAlias() + " - Numero: " + cId

    @ 06, 03 SAY "First....:" GET hBuffer[ "First"    ] PICTURE "@K"
    @ 07, 03 SAY "Last.....:" GET hBuffer[ "Last"     ] PICTURE "@K"
    @ 08, 03 SAY "Street...:" GET hBuffer[ "Street"   ] PICTURE "@K"
    @ 09, 03 SAY "City.....:" GET hBuffer[ "City"     ] PICTURE "@K"
    @ 10, 03 SAY "State....:" GET hBuffer[ "State"    ] PICTURE "@K"
    @ 11, 03 SAY "Zip......:" GET hBuffer[ "Zip"      ] PICTURE "@K"
    @ 12, 03 SAY "Hiredate.:" GET hBuffer[ "Hiredate" ] PICTURE "@K"
    @ 13, 03 SAY "Married..:" GET hBuffer[ "Married"  ] PICTURE "@K"
    @ 14, 03 SAY "Age......:" GET hBuffer[ "Age"      ] PICTURE "@K"
    @ 15, 03 SAY "Salary...:" GET hBuffer[ "Salary"   ] PICTURE "@K"
    @ 16, 03 SAY "Notes:"
    @ 17, 03                  GET hBuffer[ "Notes"    ] PICTURE "@K"

return( GetList )

//----------------------------------------------------------------------------//
// Inserta una fila

static procedure Insertar( oBuf, oBrw )

    local cPant := SaveScreen( 3, 2, 18, 74 )
    local GetList := PantMuestra( oBuf, ID_ALTA )

    READ

    set cursor off

    RestScreen( 3, 2, 18, 74, cPant )

    if LastKey() != K_ESC .and. Updated()
        DBAppend()
        oBuf:save()
        Alert( "Tupla insertada" )
        oBrw:RefreshAll()
    endif
```

```
return

//----------------------------------------------------------------------------//
// Modifica la fila actual

static procedure Modificar( oBuf, oBrw )

    local cPant := SaveScreen( 3, 2, 18, 74 )
    local GetList :=  PantMuestra( oBuf, ID_MODIFICA )

    READ

    set cursor off

    RestScreen( 3, 2, 18, 74, cPant )

    if LastKey() != K_ESC .and. Updated()
        oBuf:save()
        Alert( "Tupla Modificada" )
        oBrw:RefreshCurrent()
    endif

return

//----------------------------------------------------------------------------//
// Borra la fila actual

static procedure Borrar( oBuf, oBrw )

    local nRec := ( oBuf:getArea() )->( RecNo() )

    if Alert( "Realmente quieres borrar el registro?", { "Si", "No" } ) == 1
        ( oBuf:getArea() )->( DBDelete() )
        if ( oBuf:getArea() )->( Deleted() )
            Alert( "Borrado..." )
            oBrw:RefreshAll()
        else
            Alert( "No se pudo borrar el registro;" + ;
               "El reistro está bloqueado por otro" )
        endif
    else
        Alert( "No se ha borrado..." )
    endif

return

//----------------------------------------------------------------------------//
// Incluye el fuente donde se define e implementa la clase, esto puede estar en
// una LIB que podemos enlazar con los programas que la usan

#include "thbuffer.prg"

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

Y ahora la clase, la definición está hecha en Harbour y la implementación de los métodos en C. El código está muy documentado:

```
//----------------------------------------------------------------------------
// Clase para el manejo de un buffer de tipo hash asociado a un WA de DBF
// Cusrso de C
//----------------------------------------------------------------------------

#include "HBClass.ch"

CREATE CLASS TWABuffer
```

```
    DATA iData PROTECTED  // Uso interno. Puntero a la estructura en C

    CONSTRUCTOR new()
    METHOD load()
    METHOD save()
    METHOD blank()
    METHOD getBuffer()
    METHOD set( cFieldName, xValue )
    METHOD get( cFieldName )
    METHOD getAlias()
    METHOD getArea()
    METHOD getLen()

   PROTECTED:
    DESTRUCTOR free() // Uso interno. El destructor es invocado automaticamente

END CLASS

// Implementacion de los metodos en C

#pragma BEGINDUMP

#define _IDATA      1   // Posicion de la data iData

#include "hbapi.h"
#include "hbapiitm.h"
#include "hbapirdd.h"
#include "hbstack.h"

/*
 * Estructura contenedora de variables de instancia internas
 */
typedef struct
{
    AREAP pArea;
    PHB_ITEM hBuffer;
    HB_USHORT uiFields;
} DATA, *PDATA;

/*
 * Constructor de la clase
 */
HB_FUNC_STATIC( TWABUFFER_NEW )
{
    PHB_ITEM pSelf = hb_stackSelfItem(); // Recupera el objeto desde el stack
    AREAP pArea = ( AREAP ) hb_rddGetCurrentWorkAreaPointer(); // Puntero al WA actual

    if( pArea ) // Hay WA?
    {
        PDATA pData = ( PDATA ) hb_xgrab( sizeof( DATA ) ); // Reserva memoria para la estructura DATA
        PHB_ITEM hBuffer = hb_hashNew( NULL ); // Crea el hBuffer de los campos
        PHB_ITEM pKey = hb_itemNew( NULL ); // Item para el relleno de las keys con el nombre de los campos
        HB_USHORT uiFields, i;
        // Reserva memoria para los nombres de los campos
        char *szFldName = ( char * ) hb_xgrab( pArea->uiMaxFieldNameLength + 1 );

        // Utilizamos la funcion del rdd para averiguar el numero de campos
        SELF_FIELDCOUNT( pArea, &uiFields );

        // Asignamos la capacidad del hBuffer con el numero de campos
        hb_hashPreallocate( hBuffer, uiFields );

        // Quita sensibilidad a mayusculas/minusculas
        hb_hashClearFlags( hBuffer, HB_HASH_BINARY );
        hb_hashSetFlags( hBuffer, HB_HASH_IGNORECASE | HB_HASH_RESORT );

        // Inicializa el hBuffer, crea pares con la key con el nombre del campo y el value a NULL
        for( i = 1; i <= uiFields; i++ )
        {
            // Obtiene el nombre del campo para asignarlo a la key
            SELF_FIELDNAME( pArea, i, szFldName );
            hb_itemPutC( pKey, szFldName );
```

```
            hb_hashAdd( hBuffer, pKey, NULL );
        }

        // Asigna los valores a los dos miembros de la estructura
        pData->pArea = pArea;
        pData->hBuffer = hBuffer;
        pData->uiFields = uiFields;

        // Asigna el puntero del PDATA a IDATA definido en la clase
        hb_arraySetPtr( pSelf, _IDATA, ( void * ) pData );
        // Libera todo lo creado en el método
        hb_itemRelease( pKey );
        hb_xfree( szFldName );
    }

    // El constructor devuelve Self siempre
    hb_itemReturnForward( pSelf );
}

/*
 * Carga el registro en el hBuffer
 */
HB_FUNC_STATIC( TWABUFFER_LOAD )
{
    PHB_ITEM pSelf = hb_stackSelfItem(); // Recupera el objeto desde el stack
    // Recupera la estructura interna desde el objeto
    PDATA pData = ( PDATA ) hb_arrayGetPtr( pSelf, _IDATA );
        PHB_ITEM pValue = hb_itemNew( NULL ); // Crea Item para el valor
        HB_USHORT i;

    // Carga el hBuffer a partir del registro actual
    for( i = 1; i <= pData->uiFields; i++ )
        {
        // Obtiene el valor del campo para asignarlo al value
                SELF_GETVALUE( pData->pArea, i, pValue );
                hb_hashAdd( pData->hBuffer, hb_hashGetKeyAt( pData->hBuffer, i ), pValue );
        }

        hb_itemRelease( pValue ); // Libera todo lo creado en el método

    // Devuelve el buffer
    hb_itemReturn( pData->hBuffer );
}

/*
 * Salva el hBuffer en el registro
 */
HB_FUNC_STATIC( TWABUFFER_SAVE )
{
    PHB_ITEM pSelf = hb_stackSelfItem(); // Recupera el objeto desde el stack
    // Recupera la estructura interna desde el objeto
    PDATA pData = ( PDATA ) hb_arrayGetPtr( pSelf, _IDATA );
    HB_USHORT i;

    // Salva el hBuffer al registro actual
    for( i = 1; i <= pData->uiFields; i++ )
        {
                SELF_PUTVALUE( pData->pArea, i, hb_hashGetValueAt( pData->hBuffer, i ) );
        }
}

/*
 * Limpia el buffer asignando valores vacios del tipo que sea el campo relacionado
 * Para hacer eso se usa el registro fantasma
 */
HB_FUNC_STATIC( TWABUFFER_BLANK )
{
    PHB_ITEM pSelf = hb_stackSelfItem(); // Recupera el objeto desde el stack
    // Recupera la estructura interna desde el objeto
    PDATA pData = ( PDATA ) hb_arrayGetPtr( pSelf, _IDATA );
    PHB_ITEM pRecNo = hb_itemNew( NULL ); // Para guardar el registro actual
    PHB_ITEM pRec0 = hb_itemPutNL( NULL, 0 ); // Para ir al registro fantasma. Recno 0
```

```
    SELF_RECID( pData->pArea, pRecNo ); // Guarda el registro actual
    SELF_GOTOID( pData->pArea, pRec0 ); // Va al registro fantasma
    HB_FUNC_EXEC( TWABUFFER_LOAD ); // Ejecuta el metodo load()
    SELF_GOTOID( pData->pArea, pRecNo ); // Vuelve al registro actual

    // Libera todo lo creado en el método
    hb_itemRelease( pRec0 );
    hb_itemRelease( pRecNo );

    // Devuelve el buffer
    hb_itemReturn( pData->hBuffer );
}

/*
 * Devuelve el hBuffer
 */
HB_FUNC_STATIC( TWABUFFER_GETBUFFER )
{
    PHB_ITEM pSelf = hb_stackSelfItem(); // Recupera el objeto desde el stack

    // Obtiene el hBuffer y lo devuelve al prg
    hb_itemReturn( ( ( PDATA ) hb_arrayGetPtr( pSelf, _IDATA ) )->hBuffer );
}

/*
 * Devuelve el valor actual del campo especificado en el hBuffer
 */
HB_FUNC_STATIC( TWABUFFER_GET )
{
    PHB_ITEM cKey = hb_param( 1, HB_IT_STRING );
    PHB_ITEM pRes = NULL;

    if( cKey )
    {
        PHB_ITEM pSelf = hb_stackSelfItem(); // Recupera el objeto desde el stack
        // Recupera la estructura interna desde el objeto
        PDATA pData = ( PDATA ) hb_arrayGetPtr( pSelf, _IDATA );

        // Asigna el value de la key pasada
        pRes = hb_hashGetItemPtr( pData->hBuffer, cKey, HB_HASH_AUTOADD_ACCESS );
    }

    hb_itemReturn( pRes );
}

/*
 * Asigna el valor del campo especificado en el buffer
 */
HB_FUNC_STATIC( TWABUFFER_SET )
{
    PHB_ITEM pKey = hb_param( 1, HB_IT_STRING ); // Recoge la key
    PHB_ITEM pValue = hb_param( 2, HB_IT_ANY ); // Recoge el value
    HB_BOOL fRes = HB_FALSE;

    if( pKey && pValue ) // Si se hanpasado los dos valores del par key / value
    {
        PHB_ITEM pSelf = hb_stackSelfItem(); // Recupera el objeto desde el stack
        // Recupera la estructura interna desde el objeto
        PDATA pData = ( PDATA ) hb_arrayGetPtr( pSelf, _IDATA );

        // Comprueba que la key es un nombre de campo de la dbf
        fRes = hb_hashScan( pData->hBuffer, pKey, NULL );

        if( fRes )
        {
            hb_hashAdd( pData->hBuffer, pKey, pValue ); // Asigna el valor a l value del para con esa key
        }
    }

    // Devuelve valor logico de fRes. Uso para ello el item creado por harbour para la devolución de valores.
    // Mirad el tema del Stack en el libro del alumno
```

```
    hb_itemPutL( hb_stackReturnItem(), fRes );
}

/*
 * Obtiene el numero de pares del hBuffer
 */
HB_FUNC_STATIC( TWABUFFER_GETLEN )
{
    PHB_ITEM pSelf = hb_stackSelfItem();  // Recupera el objeto desde el stack
    // Recupera la estructura interna desde el objeto
    PDATA pData = ( PDATA ) hb_arrayGetPtr( pSelf, _IDATA );

    // Devuelve WA. Uso para ello el item creado por harbour para la devolución de valores.
    // Mirad el tema del Stack en el libro del alumno
    hb_itemPutNI( hb_stackReturnItem(), pData->uiFields );
}

/*
 * Obtiene el alias asociado al hBuffer
 */
HB_FUNC_STATIC( TWABUFFER_GETALIAS )
{
    PHB_ITEM pSelf = hb_stackSelfItem();  // Recupera el objeto desde el stack
    // Recupera la estructura interna desde el objeto
    PDATA pData = ( PDATA ) hb_arrayGetPtr( pSelf, _IDATA );
    char *szAlias = ( char * ) hb_xgrab( HB_RDD_MAX_ALIAS_LEN + 1 );

    SELF_ALIAS( pData->pArea, szAlias ); // Obtiene el alias del area del Buffer

    // Devuelve el alias. Uso para ello el item creado por harbour para la devolución de valores.
    // Mirad el tema del Stack en el libro del alumno
    hb_itemPutC( hb_stackReturnItem(), szAlias );

    // Libera todo lo creado en el método
    hb_xfree( szAlias );
}

/*
 * Obtiene el wa asociado al hBuffer
 */
HB_FUNC_STATIC( TWABUFFER_GETAREA )
{
    PHB_ITEM pSelf = hb_stackSelfItem();  // Recupera el objeto desde el stack
    // Recupera la estructura interna desde el objeto
    PDATA pData = ( PDATA ) hb_arrayGetPtr( pSelf, _IDATA );

    // Devuelve WA. Uso para ello el item creado por harbour para la devolución de valores.
    // Mirad el tema del Stack en el libro del alumno
    hb_itemPutNI( hb_stackReturnItem(), pData->pArea->uiArea );
}

/*
 * Destructor de la clase, es llamado automaticamente
 */
HB_FUNC_STATIC( TWABUFFER_FREE )
{
    PHB_ITEM pSelf = hb_stackSelfItem();  // Recupera el objeto desde el stack
    // Recupera la estructura interna desde el objeto
    PDATA pData = ( PDATA ) hb_arrayGetPtr( pSelf, _IDATA );

    if( pData )
    {
        // Libera los miembros de la estructura interna creada
        hb_itemRelease( pData->hBuffer );
        hb_xfree( pData );
        hb_arraySetPtr( pSelf, _IDATA, NULL );
    }
}

#pragma ENDDUMP
```

17. Ejecutar funciones, métodos y codeBlock de Harbour en C.

Aveces es necesario ejecutar en C una función de Harbour (de las que se usan habitualmente en los PRG) o hecha en PRG por terceros o nosotros mismos. Incluso recuperar el valor que devuelve para procesarlo desde C. La buena noticia es que hacer esto en Harbour es muy fácil.

Para entender lo que vamos a tratar ahora sería necesario repasar el tema 12 ya que vamos a usar la pila (Stack), la tabla de símbolos (Symbol Table) y nosotros vamos a invocar la maquina virtual (VM).

Este es el listado de funciones.

Para la ejecución:
void hb_vmDo(HB_USHORT uiParams); → Invocar la máquina virtual.
void hb_vmProc(HB_USHORT uiParams); → Ejecuta una función o procedimiento.
void hb_vmFunction(HB_USHORT uiParams); → Ejecuta una función
void hb_vmSend(HB_USHORT uiParams); → Envía un mensaje a un objeto

Las 4 funciones anteriores reciben como parámetro un entero sin signo que representa el numero de parámetros que se ha pasado a la pila con las ***funciones de poner parámetros en la pila*** *que veremos más abajo.*

Para el tratamiento de codeblock:
PHB_ITEM hb_vmEvalBlock(PHB_ITEM pBlockItem); → Ejecuta el codeBlock pasado sin argumentos
Las siguientes ejecutan el codeBlock pasado con un número variable de argumentos:
PHB_ITEM hb_vmEvalBlockV(PHB_ITEM pBlockItem, HB_ULONG ulArgCount, ...);
PHB_ITEM hb_vmEvalBlockOrMacro(PHB_ITEM pItem); → Ejecuta el codeblock o macro apuntado por elemento dado
void hb_vmDestroyBlockOrMacro(PHB_ITEM pItem); → Destruye el bloque de código o la macro en un elemento dado

Para poner los parámetros en la la pila:
void hb_vmPush(PHB_ITEM pItem); → Pone un elemento genérico ***pItem.***
void hb_vmPushNil(void); → No pone nada
void hb_vmPushNumber(double dNumber, int iDec); → Pone un número y comprueba si es entero, largo o doble.
void hb_vmPushInteger(int iNumber); → Pone un número entero.
void hb_vmPushLong(long lNumber); → Pone un número entero largo.
void hb_vmPushDouble(double dNumber, int iDec); → Pone un número real.
void hb_vmPushSize(HB_ISIZ nNumber); → Pone un número HB_SIZE.
void hb_vmPushNumInt(HB_MAXINT nNumber); → Coloca un número y decide si es un número entero o HB_MAXINT.
void hb_vmPushLogical(HB_BOOL bValue); → Coloca un valor lógico.
void hb_vmPushString(const char * szText, HB_SIZE length); → Pone una cadena.

void hb_vmPushStringPcode(const char * szText, HB_SIZE length); → Pone una cadena de pcode.
void hb_vmPushDate(long lDate); → Pone una fecha como un entero largo.
void hb_vmPushTimeStamp(long lJulian, long lMilliSec); → Pone dos valores enteros largos como TimeStamp.
void 525223(PHB_SYMB pSym); → Pone un puntero de símbolo de función.
void hb_vmPushDynSym(PHB_DYNS pDynSym); → Pone un puntero de función/método.
void hb_vmPushEvalSym(void); → Pone un símbolo de evaluación de bloque de código.
void hb_vmPushPointer(void * pPointer); → Pone un elemento de tipo HB_IT_POINTER
void hb_vmPushPointerGC(void * pPointer); → Pone un elemento de tipo HB_IT_POINTER del Recolector de Basura.
void hb_vmPushItemRef(PHB_ITEM pItem); → Pone un item pasado por referencia.

Vamos a los ejemplos:

El PRG que llama a las funciones en C:

```
//-----------------------------------------------------------------------------
// Ejercicio uso de funciones C. Ejecutar funciones PRG desde C
// ej012.prg
//-----------------------------------------------------------------------------

procedure main

    local n

    cls

    // Muestra una cadena desde C en un Alert, no recibe parámetros
    ejAlertDesdeC00()

    // Muestra una cadena desde C en un Alert, recibe un parámetro de tipo cadena "C"
    ejAlertDesdeC01( "ALERT desde C pasando esta cadena" )
    // Mensaje de aviso al comprobar que no se le ha pasado una cadena
    ejAlertDesdeC01( Date() )

    // Muestra una cadena desde C en un Alert, recibe un parámetro de cualquier
    // tipo y lo pasa a cadena "C"
    ejAlertDesdeC02( "Hola soy una cadenita" ) // Cadena
    ejAlertDesdeC02( Time() ) // Cadena con formato de tiempo
    ejAlertDesdeC02( Date() ) // Fecha
    ejAlertDesdeC02( 1200 ) // Entero
    ejAlertDesdeC02( 390.25 ) // Numero real
    ejAlertDesdeC02( 1200 + 390.25 ) // Suma de entero y real
    ejAlertDesdeC02( .t. )  // Lógico
    ejAlertDesdeC02() // Sin parámetro, sale el mensaje con un aviso

    // Ejecuta una función hecha por nosotros mismos o de terceros
    n := ejSumaEnC()
    ? "Valor devuelto por ejSumaEnC", n
    n := ejSumaEnCPar( 10, 34.79)
    ? "Valor devuelto por ejSumaEnCPar -> 10 + 34.79 =", n
    n := ejSumaEnCParPro( 10, 34.79)
    ? "Valor devuelto por ejSumaEnCParPro -> ( 10 + 34.79 ) * 100 =", n
    ? "-----------------------------------------------------------------------"
    Inkey( 100 )

return

//-----------------------------------------------------------------------------
// Suma dos números
//-----------------------------------------------------------------------------
```

```
function suma( n1, n2 )

    if ValType( n1 ) != 'N'
        n1 := 0
    endif
    if ValType( n2 ) != 'N'
        n2 := 0
    endif

return n1 + n2

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

Y ahora las funciones en C:

```
/*
 * Ejecutar funciones PRG desde C
 */

#include "hbvm.h" // Necesario para invocar a la maquina virtual y usar sus funciones
#include "hbstack.h" // Para el manejo de la pila

/*
 * Llama a la función ALERT desde C.
 * Asignamos una cadena también desde C
 */
HB_FUNC( EJALERTDESDEC00 )
{
        // Busca el símbolo ALERT en la tabla dinámica de símbolos
        // La función hb_dynsymFind busca el símbolo a partir del nombre teniendo
        // en cuenta mayúsculas y minúsculas, si no lo en cuenta devuelve NULL por lo
        // que habría que controlar eso. En este caso no lo hago porque se trata de
        // una función de Harbour que siempre va a existir.
        PHB_DYNS pExecSym = hb_dynsymFind( "ALERT" );

        hb_vmPushDynSym( pExecSym ); // Pone el símbolo de la función en la pila
        // IMPORTANTE. La primera posición en la pila después del símbolo de la función
        // siempre será un NULL para las funciones y procedimientos o SELF para los
        // métodos de los clases
        hb_vmPushNil(); // Pone NULL
        // Pone una cadena como cadena de C. Hay que pasar el tamaño de la cadena
        hb_vmPushString( "Hola, esto es un ALERT desde C", 30 );
        // Esta función invoca a la Maquina Virtual de Harbour (VM) para ejecutar
        // la función. Hay que indicar el numero de parámetros.
        hb_vmDo( 1 );
}

/*
 * Llama a la función ALERT desde C.
 * Pasmos una cadena desde PRG y controla que sea una cadena
 */
HB_FUNC( EJALERTDESDEC01 )
{
        PHB_DYNS pExecSym = hb_dynsymFind( "ALERT" );
        PHB_ITEM cItem = hb_param( 1, HB_IT_STRING );

        hb_vmPushDynSym( pExecSym );
        hb_vmPushNil();

        // Controla que sea del tipo esperado en hb_param() o sea una cadena
        // HB_IT_STRING (Item Tipe String)
        if( cItem )
        {
                hb_vmPush( cItem ); // Pone el parámetro como un ITEM
        }
        else // Si no se ha pasado un Item de tipo "C"
        {
                // Pone una cadena como cadena de C. Hay que pasar el tamaño de la cadena
                hb_vmPushString( "ATENCION;No se ha pasado una cadena", 35 );
```

```
        }

        // Invoca a la MV para ejecutar lo que hay en la pila
        // Se puede usar una de las dos funciones
        //hb_vmDo( 1 );
        hb_vmProc( 1 );
}

/*
 * Llama a la funcion ALERT desde C.
 * Pasmos cualquier valor y tipo desde PRG y controla que se ha pasado algo
 */
HB_FUNC( EJALERTDESDEC02 )
{
        PHB_DYNS pExecSym = hb_dynsymFind( "ALERT" );
        PHB_ITEM xItem = hb_param( 1, HB_IT_ANY );

        hb_vmPushDynSym( pExecSym );
        hb_vmPushNil();

        if( xItem )
        {
                // La función hb_itemValToStr() crea un nuevo item de tipo carácter
                // a partir del item de cualquier tipo que le hayamos pasado, por lo
                // que hay que destruirlo una vez usado.
                PHB_ITEM cItem = hb_itemValToStr( xItem );

                hb_vmPush( cItem );
                hb_itemRelease( cItem );
        }
        else
        {
                hb_vmPushString( "ATENCION;No se ha pasado ningun valor", 37 );
        }
        // Se puede usar una de las dos funciones
        //hb_vmDo( 1 );
        hb_vmProc( 1 );
}

/*
 * Llama a la funcion Suma hecha en PRG desde C.
 * Los numero los metemos desde C
 */
HB_FUNC( EJSUMAENC )
{
        //  hb_dynsymFind() es case sensitive, Harbour pone todos los símbolos creados
        // desde PRG en mayúscula pero respeta mayúsculas y minúsculas, pero si el nombre
        // que le vamos a pasar parámetro no se sabe si está en mayúscula o minúscula se
        // puede usar hb_dynsymFindName() que lo pasa a mayúsculas antes de buscar.
        PHB_DYNS pExecSym = hb_dynsymFind( "SUMA" );

        // Cuando no hay una seguridad de que exista el símbolo hay que comprobarlo
        if( pExecSym )
        {
                hb_vmPushDynSym( pExecSym );
                hb_vmPushNil();
                hb_vmPushInteger( 1500 );
                hb_vmPushDouble( 250.35, 2 );
                // Se puede usar una de las dos funciones
                //hb_vmDo( 2 );
                hb_vmProc( 2 );
        }
}

/*
 * Llama a la función Suma hecha en PRG desde C.
 * Los numero los pasamos desde PRG
 */
HB_FUNC( EJSUMAENCPAR )
{
        PHB_DYNS pExecSym = hb_dynsymFind( "SUMA" );
```

```
        // Cuando no hay una seguridad de que exista el símbolo hay que comprobarlo
        if( pExecSym )
        {
                // Los dos parametros
                PHB_ITEM n1 = hb_param( 1, HB_IT_NUMERIC );
                PHB_ITEM n2 = hb_param( 2, HB_IT_NUMERIC );

                hb_vmPushDynSym( pExecSym );
                hb_vmPushNil();
                hb_vmPush( n1 );
                hb_vmPush( n2 );
                //hb_vmDo( 2 ); // Se puede usar una de las dos funciones
                hb_vmProc( 2 );
        }
}

/*
 * Llama a la función Suma hecha en PRG desde C y procesa el resultado en C. En
 * este caso multiplicamos por 100 y devuelve el resultado
 * Los 2 numero los pasamos desde PRG
 */
HB_FUNC( EJSUMAENCPARPRO )
{
        PHB_DYNS pExecSym = hb_dynsymFind( "SUMA" );

        // Cuando no hay una seguridad de que exista el símbolo hay que comprobarlo
        if( pExecSym )
        {
                double dNum, dRes;
                // Los dos parametros
                PHB_ITEM n1 = hb_param( 1, HB_IT_NUMERIC );
                PHB_ITEM n2 = hb_param( 2, HB_IT_NUMERIC );

                hb_vmPushDynSym( pExecSym );
                hb_vmPushNil();
                hb_vmPush( n1 );
                hb_vmPush( n2 );
                //hb_vmDo( 2 );
                hb_vmProc( 2 ); // El resultado lo pone en la pila en el item de devolución

                // El item de devolución se rescata con la función hb_stackReturnItem()
                // Saca el numero como tipo en C double para procesarlo
                dNum = hb_itemGetND( hb_stackReturnItem() ); // Saca el numero como tipo en C
                // Hace el procesamiento, en este caso multiplica por 100
                dRes = dNum * 100;
                // Resultado se pone en la pila en el item de devolución
                hb_itemPutND( hb_stackReturnItem(), dRes );
        }
}
```

Hasta ahora sólo hemos usado ***void hb_vmDo(HB_USHORT uiParams);*** para invocar a la Máquina Virtual (VM), pero realmente esta función se puede utilizar para ***funciones***, ***procedimientos*** y envío de ***métodos***. El resto de funciones está un poco más optimizadas para la ejecución de lo que indica el nombre y que ya está explicado al principio del tema.

void hb_vmProc(HB_USHORT uiParams); Debería usarse para procedimientos y funciones indistintamente**.**
void hb_vmFunction(HB_USHORT uiParams); Garantiza que el ***StackReturn*** va a estar limpio para aceptar esa devolución de valores.

void hb_vmSend(HB_USHORT uiParams); Esta debería ser la opción para el envío de mensajes a los objetos y que se ejecute el método correspondiente. ***hb_vmSend()*** sí sería conveniente usarla en vez de ***hb_vmDo()*** ya que está hecha sólo para este asunto y está más optimizada.

Dicho esto, yo *recomiendo utilizar **hb_vmProc()** para procedimientos y funciones indistintamente y **hb_vmSend()** sólo para métodos.*

Veamos un ejemplo de envío de mensajes con ***hb_vmSend()***.

El PRG con el ejemplo y la clase:

```
//----------------------------------------------------------------------------
// Ejercicio uso de metodos en C. Ejecutar metodos PRG desde C
// ej013.prg
//----------------------------------------------------------------------------

#include "HBClass.ch"

PROCEDURE main

   LOCAL oModel := TMiModelo():new( "Manu", "Exposito", 57, 1200 )

   oModel:verDatosModelo()

   cambiaDatosEnC( oModel )
   oModel:verDatosModelo()

   cambiaDatosEnCPar( oModel, "Gerogina", "Gamero", 33, 4523.65 )
   oModel:verDatosModelo()

RETURN

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

CREATE CLASS TMiModelo

   HIDDEN:
   DATA cNombre
   DATA cApellido
   DATA nEdad
   DATA nSueldo

   EXPORTED:
   CONSTRUCTOR new( cNombre, cApellido, nEdad, nSueldo )
   METHOD cambiaDatos( cNombre, cApellido, nEdad, nSueldo )
   METHOD isMayorEdad()
   METHOD guardaModelo()
   METHOD leeModelo()
   METHOD verDatosModelo()

   // Metodos SET / GET
   METHOD getNombre()
   METHOD setNombre( cNombre )
   METHOD getApellido()
   METHOD setApellido( cApellido )
   METHOD getEdad()
   METHOD setEdad( nEdad )
   METHOD getSueldo()
   METHOD setSueldo( nSueldo )

END CLASS

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

METHOD new( cNombre, cApellido, nEdad, nSueldo ) CLASS TMiModelo
```

```
   ::cambiaDatos( cNombre, cApellido, nEdad, nSueldo )

RETURN self

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

METHOD cambiaDatos( cNombre, cApellido, nEdad, nSueldo ) CLASS TMiModelo

   IF ValType( cNombre ) == 'C'
      ::cNombre := cNombre
   ENDIF

   IF ValType( cApellido ) == 'C'
      ::cApellido := cApellido
   ENDIF

   IF ValType( nEdad ) == 'N'
      ::nEdad := nEdad
   ENDIF

   IF ValType( nSueldo ) == 'N'
      ::nSueldo := nSueldo
   ENDIF

return self

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

METHOD isMayorEdad() CLASS TMiModelo
RETURN ::nEdad >= 18

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

METHOD guardaModelo() CLASS TMiModelo

   LOCAL lRet := .F.

   Alert( "Aqui se persiste el modelo" )

RETURN lRet

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

METHOD leeModelo() CLASS TMiModelo

   LOCAL lRet := .F.

   Alert( "Aqui se carga el modelo desde el DataSet" )

RETURN lRet

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

METHOD verDatosModelo() CLASS TMiModelo

   Alert( "INFORMACION;-----------;" + ";" + ;
          "Nombre...: " + ::getNombre() + ";" + ;
          "Apellido.: " + ::getApellido() + ";" + ;
          "Edad.....: " + hb_ntos( ::getEdad() ) + ";" + ;
          "Sueldo...: " + hb_ntos( ::getSueldo() ) )

RETURN self

//------------------------------------------------------------------------------
// Metodos SET / GET

METHOD getNombre() CLASS TMiModelo
RETURN ::cNombre

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

METHOD setNombre( cNombre ) CLASS TMiModelo
```

```
   IF ValType( cNombre ) == 'C' .AND. !Empty( cNombre )
      ::cNombre := cNombre
   ENDIF

RETURN self

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

METHOD getApellido() CLASS TMiModelo
RETURN ::cApellido

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

METHOD setApellido( cApellido ) CLASS TMiModelo

   IF ValType( cApellido ) == 'C' .AND. !Empty( cApellido )
      ::cApellido := cApellido
   ENDIF

RETURN self

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

METHOD getEdad() CLASS TMiModelo
RETURN ::nEdad

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

METHOD setEdad( nEdad ) CLASS TMiModelo

   IF ValType( nEdad ) == 'N' .AND. edad > 0
      ::nEdad := nEdad
   ENDIF

RETURN self

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

METHOD getSueldo() CLASS TMiModelo
RETURN ::nSueldo

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

METHOD setSueldo( nSueldo ) CLASS TMiModelo

   IF ValType( nSueldo ) == 'N' .AND. nSueldo > 0
      ::nSueldo := nSueldo
   ENDIF

RETURN self

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

Y ahora la parte de C:

```
/*
 * Ejemplo de ejacucion de metodos desde C
 * Toma los datos directamente desde C con variables de tipo C
 */

HB_FUNC( CAMBIADATOSENC )
{
        PHB_ITEM pObj = hb_param( 1, HB_IT_OBJECT );

        if( pObj )
        {
                PHB_DYNS pMsgSym = hb_dynsymFind( "CAMBIADATOS" );
```

```
                hb_vmPushDynSym( pMsgSym ); // Pone el metodo en la pila
                hb_vmPush( pObj ); // Esto es super importante, poner el objeto en primer lugar
                hb_vmPushString( "Isabel", 6 );
                hb_vmPushString( "Guerrero", 8 );
                hb_vmPushInteger( 52 );
                hb_vmPushDouble( 2500.75, 2 );
                hb_vmSend( 4 );
        }
}

/*
 * Ejemplo de ejacucion de metodos desde C
 * Toma los datos pasados desde PRG con Item
 */

HB_FUNC( CAMBIADATOSENCPAR )
{
        PHB_ITEM pObj = hb_param( 1, HB_IT_OBJECT );

        if( pObj )
        {
                PHB_ITEM cNombre = hb_param( 2, HB_IT_STRING );
                PHB_ITEM cApellido = hb_param( 3, HB_IT_STRING );
                PHB_ITEM nEdad = hb_param( 4, HB_IT_NUMINT );
                PHB_ITEM nSueldo = hb_param( 5, HB_IT_DOUBLE );
                PHB_DYNS pMsgSym = hb_dynsymFind( "CAMBIADATOS" );

                hb_vmPushDynSym( pMsgSym ); // Pone el metodo en la pila
                hb_vmPush( pObj ); // Esto es super importante, poner el objeto en primer lugar
                hb_vmPush( cNombre );
                hb_vmPush( cApellido );
                hb_vmPush( nEdad );
                hb_vmPush( nSueldo );
                hb_vmSend( 4 );
        }
}
```

Vamos con los **codeblocks…**

Al igual que con la ejecución de funciones, procedimientos y métodos, la evaluación de codeBlocks desde C es muy fácil y sigue las mismas maneras que lo ya descrito, o sea el uso de la pila y la llamada de la máquina virtual para procesar.

La propuesta inicial que hacía Clipper era el uso de las funciones siguientes:

HB_BOOL hb_evalNew(PHB_EVALINFO pEvalInfo, PHB_ITEM pItem); → *Prepara la estructura EVALINFO con el code block pasado y lista para recibir los parámetros.*

HB_BOOL hb_evalPutParam(PHB_EVALINFO pEvalInfo, PHB_ITEM pItem); → *Añade un parámetro a la estructura. Esta función se llamará tantas veces como parámetros tenga el codeblock.*

PHB_ITEM hb_evalLaunch(PHB_EVALINFO pEvalInfo); → *Evalúa el codeblock con los parámetros existentes en la estructura EVALINFO.*

HB_BOOL hb_evalRelease(PHB_EVALINFO pEvalInfo); → *Libera la memoria ocupada por la copia de los items usados como parámetros e inicializa la estructura para ser usada de nuevo.*
Ojo, si los items pasados como parámetros eran los originales recogidos con ***hb_param()*** *y no con* ***hb_itemParam()*** *(o sea copias) esta función producirá un error indeterminado al intentar liberar esos items.*

Un par de ejemplos:

```
/*
 * Evalúa un codeBlock pasado desde C.
 * Es la manera de hacerlo con la vieja propuesta de Clipper.
 * Sin parámetros
 */
HB_FUNC( EVALUA00 )
{
        PHB_ITEM pCB = hb_param( 1, HB_IT_BLOCK ); // El codeblock pasado

        if( pCB ) // Si se paso un CB
        {
                HB_EVALINFO EvalInfo; // Estructura HB_EVALINFO

                // Inicializa la estructura HB_EVALINFO (es un puntero (&))
                if( hb_evalNew( &EvalInfo, pCB ) )
                {
                        hb_evalLaunch( &EvalInfo ); // Evalua el CB
                }

                hb_evalRelease( &EvalInfo ); // Libera la memoria interna de la estructura
        }
}

/*
 * Evalúa un codeBlock pasado desde C.
 * Es la manera de hacerlo con la vieja propuesta de Clipper.
 * Con parámetros
 */
HB_FUNC( EVALUA01 )
{
        PHB_ITEM pCB = hb_param( 1, HB_IT_BLOCK ); // El codeblock pasado

        if( pCB ) // Si se paso un CB
        {
                HB_EVALINFO EvalInfo; // Estructura HB_EVALINFO

                // Inicializa la estructura HB_EVALINFO
                if( hb_evalNew( &EvalInfo, pCB ) )
                {
                        // Ojo usar hb_itemParam() que es una copia del parámetro pasado.
                        // No el propio parametro ya que hb_evalRelease libera esas copias de
                        // los item pasados.
                        hb_evalPutParam( &EvalInfo, hb_itemParam( 2 ) ); // Añade primer parámetro.
                        hb_evalPutParam( &EvalInfo, hb_itemParam( 3 ) ); // Añade segundo parámetro

                        hb_evalLaunch( &EvalInfo ); // Evalúa el CB con los parámetros pasados.
                }

                hb_evalRelease( &EvalInfo ); // Libera la memoria interna de la estructura
        }
}
```

Y las funciones indocumentadas de evaluación directa de codeblocks:

void hb_evalBlock0(PHB_ITEM pCodeBlock); → *Evalúa un codeblock sin parámetros.*

void hb_evalBlock1(PHB_ITEM pCodeBlock, PHB_ITEM pParam); → *Evalúa un codeblock con un parámetro.*

void hb_evalBlock(PHB_ITEM pCodeBlock, ...); → *Evalúa un codeblock con un número de parámetros indeterminado. Ojo el ultimo parámetro pasado debe ser NULL.*

Ejemplos:

```
/*
 * Evalúa un codeBlock pasado desde C.
 * Función para evaluar un CB sin parámetros directamente.
 */
HB_FUNC( EVALUA04 )
{
	PHB_ITEM pCB = hb_param( 1, HB_IT_BLOCK ); // El codeblock pasado

	if( pCB )
	{
		hb_evalBlock0( pCB );
	}
}

/*
 * Evalúa un codeBlock pasado desde C.
 * Función para evaluar un CB con un parámetro directamente.
 */
HB_FUNC( EVALUA05 )
{
	PHB_ITEM pCB = hb_param( 1, HB_IT_BLOCK ); // El codeblock pasado

	if( pCB )
	{
		hb_evalBlock1( pCB, hb_param( 2, HB_IT_ANY ) );
	}
}

/*
 * Evalúa un codeBlock pasado desde C.
 * Función para evaluar un CB con parámetros directamente.
 */
HB_FUNC( EVALUA06 )
{
	PHB_ITEM pCB = hb_param( 1, HB_IT_BLOCK ); // El codeblock pasado

	if( pCB )
	{
		hb_evalBlock( pCB, hb_param( 2, HB_IT_ANY ), hb_param( 3, HB_IT_ANY ), NULL );
	}
}
```

Como curiosidad hay que hacer constar que Harbour trata internamente los codeblocks como objetos. El objeto sería el nombre y el método para evaluar sería **`bCB:eval( p0, p1, …, pn )`** con los parámetros que se le pasen. Y precisamente esa es la nueva propuesta de Harbour para la evaluación interna de los codeblocks. Lamanera de hacerlo es como si se tratara de un metodo.

Lo veamos en los ejemplos:

```
/*
 * Evalúa un codeBlock pasado desde C.
 * Es la manera de hacerlo con la nueva propuesta de Harbour.
 * Funciona exactamente como hemos visto con los métodos de los objetos
 * Este es el mismo ejemplo que el de la función EVALUA00 visto anteriormente
 */
HB_FUNC( EVALUA02 )
{
	PHB_ITEM pCB = hb_param( 1, HB_IT_BLOCK ); // El codeblock pasado

	if( pCB )
	{
		hb_vmPushEvalSym();  // Pone el símbolo de la función eval() en la pila
```

```
                hb_vmPush( pCB );     // Pone el codeblock en la pila similar al objeto
                hb_vmSend( 0 );       // Ejecuta sin parámetros. Como si fuera un objeto
        }
}

/*
 * Evalúa un codeBlock pasado desde C.
 * Es la manera de hacerlo con la nueva propuesta de Harbour.
 * Funciona exactamente como hemos visto con los métodos de los objetos
 * Este es el mismo ejemplo que el de la función EVALUA00 visto anteriormente
 */
HB_FUNC( EVALUA03 )
{
        PHB_ITEM pCB = hb_param( 1, HB_IT_BLOCK ); // El codeblock pasado

        if( pCB )
        {
                hb_vmPushEvalSym();  // Pone el símbolo de la función eval() en la pila
                hb_vmPush( pCB );    // Pone el codeblock en la pila similar al objeto
                hb_vmPush( hb_param( 2, HB_IT_ANY ) );
                hb_vmPush( hb_param( 3, HB_IT_ANY ) );
                hb_vmSend( 2 );       // Ejecuta con dos parámetros. Como si fuera un objeto
        }
}
```

Y ahora el ejemplo de uso de las funciones descritas anteriormente desde PRG:

```
//-----------------------------------------------------------------------------
// Ejercicio uso de codeblock en C. Evalúa un codeBlock sin para metros desde C,
// al estilo viejo de Clipper y al nuevo de Harbour.
// ej014.prg
//-----------------------------------------------------------------------------

#include "HBClass.ch"

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

PROCEDURE main

    local bCB0 := { || Alert( "Hola mundo" ) }
    local bCB1 := { | p1, p2 | Alert( "Los parametros pasados: " + HB_ValToStr( p1 ) + ;
                                    " y " + HB_ValToStr( p2 ) ) }
    local bCB2 := { | p1 | Alert( "Directo con un parametro: " + HB_ValToStr( p1 ) ) }

    evalua00( bCB0 )
    evalua01( bCB1, "Primer parametro", Date() )
    // OJO: En Harbour se pueden tratar los CodeBlocks como objetos
    // El objeto es la variable de tipo codeBlock y con el metodo ::eval() se le
    // pasan los parametros y se evalua
    bCB1:eval( "Tratar codeblock como objeto", Time() )
    // Usando el nuevo metodo de Harbour
    evalua02( bCB0 )
    evalua03( bCB1, "Parametro 1", Seconds() )

    Alert( "Funciones directas..." )
    evalua04( bCB0 )
    evalua05( bCB2, "Con parametro directamente" )
    evalua06( bCB1, "Con un parametro...", "otro pametro" )

RETURN

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

18. El Error API. Gestión de errores Harbour desde C.

Uno de los tratamientos más importantes de un lenguaje de programación es la gestión de las excepciones, avisos y errores. Estos se pueden generar directamente por el propio programa en tiempo de ejecución o ser enviados por el programador para controlar excepciones que no serían detectadas por el lenguaje de programación, por ejemplo los errores al insertar en una base de datos un registro con una clave única duplicada, esto no es un error del lenguaje pero sí nos interesas capturar la excepción de la base de datos y gestionar como si fuera un error normal.

En este aspecto hay que decir que Harbour tiene un sistema muy fácil pero a la vez potente. La clase ***TError*** es la encargada de hacer ese trabajo. Esta clase no tiene métodos tan sólo cuenta con variables de instancia o datas que marcaran el estado del objeto error.

Normalmente es el propio sistema el que crea el objeto error y como mucho podemos tener un gestor de errores a nuestra medida para controlar la salida por pantalla o escribir en un archivo de LOG.

Pero aveces es necesario crear nuestro propio objeto error para gestionarlo manualmente. En PRG hay una función que se encarga de ello: ErrorNew(), una vez creado el objeto podríamos acceder a sus DATAs y asignarle valores directamente:

```
local oError := ErrorNew() // Crea el objeto

// Asigna nuevos valores a las DATAs del objeto recién creado:
oError:severity    := ES_ERROR
oError:genCode     := EG_UNSUPPORTED
oError:subSystem   := "MI_LIB"
oError:subCode     := 0
oError:description := “Esto es un error controlado por mi...”
oError:canRetry    := .F.
oError:canDefault  := .F.
oError:fileName    := ""
oError:osCode      := 0

// Lanza el error para que sea controlado por el gestor de errores por defecto o
// el nuestro propio
Eval( ErrorBlock(), oError )
```

Pues bien eso mismo lo podemos hacer desde C usando el Error API.

Para ello tenemos a nuestra disposición una serie de funciones que crean el objeto, asignan o recuperan los valores de las datas del mismo, lo lanzan una vez preparado y finalmente una función para liberar el objeto.

Estas son las datas o variables de instancia a las que podemos tener acceso con las funciones en C o directamente a las datas desde PRG:

Args: Contiene una matriz de argumentos proporcionados a un operador o función cuando se produce un error de argumento. Para otros tipos de errores, ***args*** contiene un valor NIL.

CanDefault: Contiene un valor lógico que indica si el subsistema puede realizar una recuperación de error predeterminada para la condición de error. Un valor de verdadero (.T.) Indica que la recuperación predeterminada está disponible. La disponibilidad del manejo predeterminado y la estrategia de recuperación predeterminada real depende del subsistema y de la condición del error. La acción mínima es simplemente ignorar la condición de error. La recuperación predeterminada se solicita devolviendo falso (.F.) Del bloque de error invocado para manejar el error. Tenga en cuenta que ***canDefault*** nunca es verdadero (.T.) si ***canSubstitute*** es verdadero (.T.).

CanRetry: Contiene un valor lógico que indica si el subsistema puede reintentar la operación que provocó la condición de error. Un valor de verdadero (.T.) Indica que es posible un reintento. El reintento puede estar disponible o no, según el subsistema y la condición de error. El reintento se solicita devolviendo verdadero (.T.) del bloque de error invocado para manejar el error. ***canRetry*** nunca contiene verdadero (.T.) Si ***canSubstitute*** contiene verdadero (.T.).

CanSubstitute: Contiene un valor lógico que indica si un nuevo resultado puede ser sustituido por la operación que produjo la condición de error. Los errores de argumento y algunos otros errores simples permiten que el gestor de errores sustituya la operación fallida por un nuevo valor de resultado. Un valor verdadero (.T.) significa que la sustitución es posible. La sustitución se realiza devolviendo el nuevo valor de resultado del bloque de código invocado para manejar el error. ***canSubstitute*** nunca es verdadero (.T.) si ***canDefault*** o ***canRetry*** es verdadero (.T.).

Cargo: Contiene un valor de cualquier tipo de datos no utilizado por el sistema de errores. Se proporciona como un valor definible por el usuario, lo que permite adjuntar información arbitraria a un objeto Error y recuperarla más tarde.

Description: Contiene una cadena de caracteres que describe la condición del error. Una cadena de longitud cero indica que el subsistema no proporciona una descripción imprimible del error. Si ***genCode*** no es cero, siempre hay disponible una descripción imprimible.

FileName: Contiene un valor de tipo cadena de carácteres que representa el nombre utilizado originalmente para abrir el archivo asociado con la condición de error. Una cadena de longitud cero indica que la condición de error no está asociada con un archivo en particular o que el subsistema no retiene la información del nombre del archivo.

GenCode: Contiene un valor numérico entero que representa un código de error genérico de Harbour. Los códigos de error genéricos permiten el manejo predeterminado de errores similares de diferentes subsistemas. Un valor cero indica que la condición de error es específica del subsistema y no corresponde a ninguno de los códigos de error genéricos. El archivo de encabezado, **error.ch**, proporciona un conjunto de constantes de manifiesto para códigos de error genéricos.

operation: Contiene una cadena de caracteres que describe la operación que se estaba intentando cuando ocurrió el error. Para operadores y funciones, ***operation*** contiene el nombre del operador o función. Para variables o funciones indefinidas, contiene el nombre de la variable o función. Una cadena de longitud cero indica que el subsistema no proporciona una descripción imprimible de la operación.

OsCode: Contiene un valor numérico entero que representa el código de error del sistema operativo asociado con la condición de error. Un valor de cero indica que la condición de error no fue causada por un error del sistema operativo. Cuando osCode se establece a un valor distinto de cero, **`DosError()`** se actualiza con el mismo valor. ***osCode*** refleja correctamente el código de error extendido de DOS para errores de archivo. Esto permite una distinción adecuada entre errores que resultan de compartir violaciones (por ejemplo, abrir EXCLUSIVO cuando otro proceso ya ha abierto el archivo) y violaciones de acceso (por ejemplo, abrir lectura / escritura cuando el archivo está marcado como solo lectura). Para obtener una lista de los códigos de error de DOS, consulte la Guía de apéndices y mensajes de error.

Severity: Contiene un valor numérico que indica la severidad de la condición del error. Se definen cuatro valores estándar en **error.ch**: **ES_WHOCARES** La condición no representa una falla; El error es informativo. **ES_WARNING** La condición no impide operaciones posteriores, pero puede resultar en un error más grave más adelante. **ES_ERROR** La condición evita operaciones posteriores sin una acción correctiva de algún tipo. **ES_CATASTROPHIC** La condición requiere la terminación inmediata de la aplicación. Tenga en cuenta que el código de soporte en tiempo de ejecución de Harbour solo genera errores con una gravedad de **ES_WARNING** o **ES_ERROR**.

SubCode: Contiene un valor numérico entero que representa un código de error específico del subsistema. Un valor de cero indica que el subsistema no asigna ningún número en particular a la condición de error.

Subsistem: Contiene una cadena de caracteres que representa el nombre del subsistema que genera el error. Para errores con los operadores y funciones básicos de Harbour, se proporciona el nombre de subsistema '**BASE**'. Para los errores generados por un controlador de base de datos, subsystem contiene el nombre del controlador de la base de datos.

tries: Contiene un valor numérico entero que representa el número de veces que se ha intentado la operación fallida. Cuando ***canRetry*** es verdadero (.T.), tries se puede usar para limitar el número de reintentos. Un valor de cero indica que el subsistema no rastrea el número de veces que se ha intentado la operación.

Funciones GET para obtener el valor de la DATA:

`PHB_ITEM hb_errGetCargo( PHB_ITEM pError );` → Recupera el valor de la data ***Cargo*** como ITEM ya que puede contener cualquier tipo de dato de Harbour.

`PHB_ITEM hb_errGetArgs( PHB_ITEM pError );` → Recupera el valor de la data ***Args*** como ITEM array con los parámetros de la función que ha provocado el error.

`const char *hb_errGetDescription( PHB_ITEM pError );` → Recupera el valor de la data ***Description*** como cadena de C.

const char *hb_errGetFileName(PHB_ITEM pError); → Recupera el valor de la data **fileName** como cadena de C.

HB_USHORT hb_errGetFlags(PHB_ITEM pError); → Recupera el valor de la data ***Flags*** como entero de C.

HB_ERRCODE hb_errGetGenCode(PHB_ITEM pError); → Recupera el valor de la data ***Code*** como ***HB_ERRCODE*** normalmente ***HB_SUCCESS → 0, HB_FAILURE → 1*** u otro entero diferente a 0 indicando el código de error. Si es 0 (***HB_SUCCESS)*** es que no ha habido errores.

const char *hb_errGetOperation(PHB_ITEM pError); → Recupera el valor de la data ***Operation*** como cadena de C.

HB_ERRCODE hb_errGetOsCode(PHB_ITEM pError); → Recupera el valor de la data ***osCode*** como ***HB_ERRCODE*** normalmente ***HB_SUCCESS → 0, HB_FAILURE → 1*** u otro entero diferente a 0 indicando el código de error. Si es 0 (***HB_SUCCESS)*** es que no ha habido errores.

HB_USHORT hb_errGetSeverity(PHB_ITEM pError); → Recupera el valor de la data ***Severity*** como entero de C.

HB_ERRCODE hb_errGetSubCode(PHB_ITEM pError); → Recupera el valor de la data ***subCode*** como ***HB_ERRCODE*** normalmente ***HB_SUCCESS → 0, HB_FAILURE → 1*** u otro entero diferente a 0 indicando el código de error. Si es 0 (***HB_SUCCESS)*** es que no ha habido errores.

const char *hb_errGetSubSystem(PHB_ITEM pError); → Recupera el valor de la data ***subSystem*** como cadena de C.

HB_USHORT hb_errGetTries(PHB_ITEM pError); → Recupera el valor de la data ***tries*** como entero de C.

Todas estas funciones reciben como único parámetro el objeto error y devuelven tipos de datos de C (no ITEM de Harbour salvo las dos primeras).

Funciones SET para asignar el valor de la DATA:

PHB_ITEM hb_errPutCargo(PHB_ITEM pError, PHB_ITEM pCargo); → Asigna el valor de la data ***Cargo*** como ITEM ya que puede contener cualquier tipo de dato de Harbour.

PHB_ITEM hb_errPutArgsArray(PHB_ITEM pError, PHB_ITEM pArgs); → Asigna el valor de la data ***Args*** como ITEM array con los parámetros de la función que ha provocado el error.

PHB_ITEM hb_errPutArgs(PHB_ITEM pError, HB_ULONG ulArgCount, ...); → Asigna el valor de la data ***Args*** es similar a la anterior con la diferencia que se le pasa después del objeto error el número de parámetros y una listas de esos parámetros.

PHB_ITEM hb_errPutDescription(PHB_ITEM pError, const char * szDescription); → Asigna el valor de la data ***Description*** como cadena de C.

PHB_ITEM hb_errPutFileName(PHB_ITEM pError, const char * szFileName); → Asigna el valor de la data **fileName** como cadena de C.

PHB_ITEM hb_errPutFlags(PHB_ITEM pError, HB_USHORT uiFlags); → Asigna el valor de la data ***Flags*** como entero de C.

PHB_ITEM hb_errPutGenCode(PHB_ITEM pError, HB_ERRCODE uiGenCode); → Asigna el valor de la data ***Code*** normalmente ***HB_SUCCESS → 0, HB_FAILURE → 1*** u otro entero diferente a 0 indicando el código de error. Si es 0 (***HB_SUCCESS)*** es que ha ido bien.

PHB_ITEM hb_errPutOperation(PHB_ITEM pError, const char * szOperation); → Asigna el valor de la data ***Operation*** como cadena de C.

PHB_ITEM hb_errPutOsCode(PHB_ITEM pError, HB_ERRCODE uiOsCode); → Asigna el valor de la data ***Code*** como ***HB_ERRCODE (HB_SUCCESS → 0 o HB_FAILURE → 1).***

PHB_ITEM hb_errPutSeverity(PHB_ITEM pError, HB_USHORT uiSeverity); → Asigna el valor de la data ***Severity*** como entero de C.

PHB_ITEM hb_errPutSubCode(PHB_ITEM pError, HB_ERRCODE uiSubCode); → Asigna el valor de la data **subCode** como ***HB_ERRCODE (HB_SUCCESS → 0 o HB_FAILURE → 1).***

PHB_ITEM hb_errPutSubSystem(PHB_ITEM pError, const char * szSubSystem); → Asigna el valor de la data ***subSystem*** como cadena de C.

PHB_ITEM hb_errPutTries(PHB_ITEM pError, HB_USHORT uiTries); → Asigna el valor de la data ***tries*** como entero de C.

PHB_ITEM hb_errNew(void); → Crea un objeto error vacío.

HB_USHORT hb_errLaunch(PHB_ITEM pError); → Lanza el error es igual que lo que hace en PRG → **`Eval( ErrorBlock(), oError )`**. Esta función puede devolver E_BREAK (interrumpe el proceso sin posibilidad de nuevos intentos), E_RETRY (se puede reintentar ejecutar denuevo) o E_DEFAULT (valor inicial).

void hb_errRelease(PHB_ITEM pError); → Libera la memoria ocupada por las datas y el propio objeto error.

Y ahora vamos a lo práctico con un ejemplo:

```
//------------------------------------------------------------------------------
// Ejercicio uso de gestion de errores en C.
// ej015.prg
//------------------------------------------------------------------------------

#include "error.ch"

PROCEDURE main

	cls

	? "Genera un error en el sistema desde C:"
	miThowError( ES_WARNING, 23, "Este es el warning 23 de mi sistema" )
	miThowError( ES_ERROR, 55, "Este es el error 55 de mi sistema" )
	miThowError( ES_CATASTROPHIC, 63, "Este es el error catastrofico 63 de mi sistema" )

RETURN

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

Otro PRG en el que los mensajes de error se guardan en un array el numero del error es el indice donde se guarda la descripción del error:

```
//------------------------------------------------------------------------------
// Ejercicio uso de gestion de errores en C.
// ej016.prg
//------------------------------------------------------------------------------

#include "error.ch"

PROCEDURE main

	cls

	? "Genera un error en el sistema desde C:"
	autoErr( ES_WARNING, 2 )
	autoErr( ES_ERROR, 1 )
	autoErr( ES_WHOCARES, 3 )

RETURN

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

procedure autoErr( nNivel, nCodErr )

	local aErr := { "Este es el error 1", ;
	                "Este es el error 2", ;
	                "Este es el error 3" }

	if nCodErr >= 1 .and. nCodErr <= 3
		miThowError( nNivel, nCodErr, aErr[ nCodErr ] )
	endif

return
```

Y esta es la función en C:

```
#include "hbapierr.h"

HB_FUNC( MITHOWERROR ) // ( HB_USHORT uiLevel, HDO_ERRCODE errCode, const char *szDesc )
{
	PHB_ITEM pError = hb_errNew();
	HB_USHORT uiLevel = hb_parnidef( 1, ES_WARNING );
	HB_ERRCODE errCode = hb_parnidef( 2, 0 );
	const char *szMsg = hb_parc( 3 );

	hb_errPutSubSystem( pError, "Curso de C (Subsistema)" ); // Nombre de subsistema
	hb_errPutSubCode( pError, errCode ); // Código del error en nuestro sistema
	hb_errPutDescription( pError, szMsg ); // Mensage
	hb_errPutSeverity( pError, uiLevel ); // Nivel del error

	hb_errPutFlags( pError, EF_CANDEFAULT ); // Pone el botón por defecto
	hb_errPutTries( pError, 5 );

	hb_errLaunch( pError ); // Se lanza el error relleno
	hb_errRelease( pError ); // Libera el objeto
}
```

19. El FileSys API. De archivos desde C.

En este tema vamos a tratar otro API de Harbour muy importante, el tratamiento de archivos. Como siempre desarrollando nuestros procesos en C podremos acelerar la creación, borrado y procesos con archivos.

La mayor parte de las funciones en C del ***FileSys API*** tienen su representación en PRG por lo que se puede intuir fácilmente su uso.

Estas son los principales funciones:

```
HB_BOOL hb_fsChDir( const char *pszDirName );
```
Cambia al directorio especificado.

```
void hb_fsClose( HB_FHANDLE hFileHandle );
```
Cierra el archivo.

```
void hb_fsCommit( HB_FHANDLE hFileHandle );
```
Actualiza los cambios en archivo.

```
HB_FHANDLE hb_fsCreate( const char *pszFileName, HB_FATTR ulAttr );
```
Crea un archivo y abre un archivo.

```
HB_FHANDLE hb_fsCreateEx( const char *pszFileName, HB_FATTR ulAttr,
                          HB_USHORT uiFlags );
```
Crear un archivo y lo abre con un modo apertura específico.

```
const char *hb_fsCurDir ( int iDrive );
```
Devuelve el nombre del directorio actual para la unidad especificada.

```
int hb_fsCurDrv( void );
```
Recuperar el número de unidad actual.

```
HB_BOOL hb_fsDelete( const char *pszFileName );
```
Borra un archivo.

```
HB_BOOL hb_fsEof( HB_FHANDLE hFileHandle );
```
Comprueba se se ha llegado al final del archivo.

HB_ERRCODE hb_fsError(void);
Recupera el error del sistema de archivos.

HB_BOOL hb_fsFile(const char *pszFileName);
Determina si existe el archivo.

HB_BOOL hb_fsIsDirectory(const char *pszFileName);
Determina si el nombre pasado es un directorio.

HB_FOFFSET hb_fsFSize(const char *pszFileName, HB_BOOL bUseDirEntry);
Determina el tamaño de un archivo.

HB_FHANDLE hb_fsExtOpen(const char *pszFileName, const char *pDefExt,
HB_FATTR nFlags, const char *pPaths, PHB_ITEM pError);
Abre un archivo con la extensión predeterminada en una lista de rutas.

HB_ERRCODE hb_fsIsDrv(int iDrive);
Determinar si un número de unidad es una unidad válida.

HB_BOOL hb_fsMkDir(const char *pszDirName);
Crea un directorio

HB_FHANDLE hb_fsOpen(const char *pszFileName, HB_USHORT uiFlags);
Abre o crea un archivo.

HB_FHANDLE hb_fsOpenEx(const char *pszFileName, HB_USHORT uiFlags, HB_FATTR nAttr);
Abre o crea un archivo con atributos dados.

HB_USHORT hb_fsRead(HB_FHANDLE hFileHandle, void *pBuff, HB_USHORT uiCount);
Lee el contenido desde la posición actual de un archivo en un búfer que puede ser hasta 64 KiB.

HB_SIZE hb_fsReadLarge(HB_FHANDLE hFileHandle, void *pBuff, HB_SIZE nCount);
Lee el contenido desde la posición actual de un archivo en un búfer que puede ser mayor de 64 KiB.

HB_SIZE hb_fsReadAt(HB_FHANDLE hFileHandle, void *pBuff, HB_SIZE nCount,
HB_FOFFSET nOffset);
Lee el contenido desde la posición dada de un archivo en un búfer que puede ser mayor de 64 KiB.

HB_BOOL hb_fsRmDir(const char *pszDirName);

Elimina un directorio.

HB_BOOL hb_fsRename(const char * pszOldName, const char *pszNewName);

Renombra un archivo.

HB_ULONG hb_fsSeek(HB_FHANDLE hFileHandle, HB_LONG lOffset, HB_USHORT uiMode);

Sitúa el puntero de lectura/escritura de un archivo en la posición indicada.

HB_FOFFSET hb_fsSeekLarge(HB_FHANDLE hFileHandle, HB_FOFFSET nOffset,
HB_USHORT uiFlags);

Sitúa el puntero de lectura/escritura de un archivo en la posición indicada usando la API de 64 bits

HB_FOFFSET hb_fsTell(HB_FHANDLE hFileHandle);

Recupera la posición actual del puntero de lectura/escritura del archivo.

HB_FOFFSET hb_fsGetSize(HB_FHANDLE hFileHandle);

Devuelve el tamaño de un archivo, ojo!!! puede cambiar la posición del puntero de lectura/escritura.

int hb_fsSetDevMode(HB_FHANDLE hFileHandle, int iDevMode);

Cambiar el modo de tratar un archivo o como texto o binario.

HB_USHORT hb_fsWrite(HB_FHANDLE hFileHandle, const void *pBuff,
HB_USHORT uiCount);

Escribe en un archivo abierto el contenido de un búfer que puede ser de hasta 64K.

HB_SIZE hb_fsWriteLarge(HB_FHANDLE hFileHandle, const void *pBuff,
HB_SIZE nCount);

Escribe en un archivo abierto el contenido de un búfer que puede ser mayor de 64K.

HB_SIZE hb_fsWriteAt(HB_FHANDLE hFileHandle, const void *pBuff, HB_SIZE nCount,
HB_FOFFSET nOffset);

Escribe en un archivo abierto desde la posición dada el contenido de un búfer que puede ser mayor de 64K.

HB_BOOL hb_fsNameExists(const char *pszFileName);

Comprueba si existe un nombre en el sistema, no se aceptan caracteres comodín.

HB_BOOL hb_fsFileExists(const char *pszFileName);

Comprueba si existe un archivo, no se aceptan caracteres comodín.

HB_BOOL hb_fsDirExists(const char *pszDirName);

Comprueba si existe un directorio, no acepta caracteres comodín.

HB_BOOL hb_fsCopy(const char *pszSource, const char *pszDest);

Hace una copia de un archivo con otro nombre.

No están todas las funciones del API pero sí la mayoría y las que con más se suelen usar.
Los parámetros de las funciones pueden ser:

pszDirName → Nombre del directorio

pszFileName → Nombre del archivo

hFileHandle → Identificador del archivo

ulAttr → Atributos

uiFlags → Opciones de apertura de archivo

iDrive → Numero de disco

pBuff → Búfer de entrada/salida

Y ahora los ejemplos.

Se propone un sistema log.

Esta es la implementación e Lenguaje C:

```
/*****************************************************************************
 ************************ Uso del FileSys API ******************************
 *****************************************************************************/

#include "hbapi.h"          // Siempre hay que incluir para funciones en C
#include "hbapifs.h"  // Para el Syetm API
#include "hbapiitm.h" // Para ITEM API

/***
 * Log manager desde C
 * IMPORTANTE: el ultimo argumento tiene que ser NULL
 * Ejemplo: writeLog( "miFic.log, HB_FALSE, "Prueba ", "de ", "LOGMANAGER", NULL );
 * Escribe las cadenas en el fichero miFic.log, si existe lo escribe al final y si no
 * lo crea y escribe.
 */
```

```
static const char *nfln;
static int iNflnLen;

/*
 * Para escribir el fichero LOG
 */

void writeLog( const char *szFileName, HB_BOOL bCreate, const char *zStr, ... )
{
	if( szFileName && zStr )	// Si se pasan los parametros
	{
		HB_FHANDLE hFile;
		HB_BOOL bExist = hb_fsFileExists( szFileName );	// Comprueba si existe el
archivo

		if( bCreate == HB_TRUE || bExist == HB_FALSE )
		{
			hFile = hb_fsCreate( szFileName, FC_NORMAL ); // Crea el archivo
		}
		else
		{
			hFile = hb_fsOpen( szFileName, FO_READWRITE ); // Abre el archivo
		}

		if( hFile != FS_ERROR )		// Si no hay errores
		{
			va_list vl;

			nfln = hb_conNewLine();
			iNflnLen = strlen( nfln );

			hb_fsSeekLarge( hFile, 0, FS_END );
			va_start( vl, zStr );

			do	// Recorre los parametros y los escribe en el LOG
			{
				hb_fsWriteLarge( hFile, zStr, strlen( zStr ) ); // Escribe uso esta
funcion por si es un bloque mayor de 64K
				hb_fsWrite( hFile, nfln, iNflnLen ); // Escribe fin de linea
			}
			while( ( zStr = va_arg( vl, char * ) ) != NULL );

			va_end( vl );

			hb_fsClose( hFile );  // Cierra archivo. El close hae el commit por lo que
no hace falta usar hb_fsCommit()
		}
	}
}

/*
 * writeLog para uso en PRG
 * Solo admite un parametro para escribir pero puede ser de cualquier tipo no solo
 * una cadena
 */

HB_FUNC( WRITELOG )
{
	const char *szFileName = hb_parc( 1 ); // Nombre del archivo
	HB_BOOL bCreate = hb_parldef( 2, HB_FALSE ); // Fuerza la creacion del archivo

	if( szFileName && hb_pcount() > 2 )
	{
		HB_SIZE nLen;
```

```
		HB_BOOL bFreeReq;
		char *szStr = hb_itemString( hb_param( 3, HB_IT_ANY ), &nLen, &bFreeReq ); // 
Convierte la variable pasada a cadena

		writeLog( szFileName, bCreate, szStr, NULL ); // Llama a la funcion en C de 
escribir al LOG

		if( bFreeReq )
		{
			hb_xfree( szStr );	// Si la funcion hb_itemString() que es necesario 
liberar memoria se hace
		}
	}
}

/*
 * Para cargar el fichero en una variable
 */

char *loadLog( const char *szFileName, HB_SIZE *nSize )
{
	HB_FHANDLE hFile = hb_fsOpen( szFileName, FO_READ ); // Abre el archivo solo para leer
	char *szValue = NULL;

	if( hFile != FS_ERROR )
	{
		*nSize = hb_fsGetSize( hFile );			// Devuelve el tamaño del 
archivo

		// Crea el bufer con el tamaño del archivo + el salto de linea
		szValue = ( char * ) hb_xgrab( *nSize + iNflnLen );

		hb_fsSeekLarge( hFile, 0, FS_SET );		// Se pone al principio del 
archivo y
		hb_fsReadLarge( hFile, szValue, *nSize );  // Lee todo y lo mete en el bufer
		hb_fsClose( hFile );					// Cierra el archivo LOG
	}

	return szValue; // Devuelve el contenido del LOG depositado en esta variable
}

/*
 * Para cargar el fichero en una variable desde PRG
 */

HB_FUNC( LOADLOG )
{
	const char *szFileName = hb_parc( 1 ); // Nombre del archivo LOG

	if( szFileName ) // Nombre del archivo LOG
	{
		HB_SIZE nSize = 0;
		char *szValue = loadLog( szFileName, &nSize );

		hb_retclen_buffer( szValue, nSize ); // Devuelve el contenido del archivo LOG
	}
	else
	{
		hb_retc_null(); // Si no se le pasa archivo LOG devuelve cadena nula
	}
}
```

Y este es su uso desde PRG, varios ejemplos:

```
//----------------------------------------------------------------------------
// Ejemplo de uso del sistema log manager
// Prgrama ej01.prg
//----------------------------------------------------------------------------

procedure main

    local n := 0
    local cFNane := "miArchivo.log"

    cls

    // Reiniciamos si existe el fichero
    // Simula errores
    writeLog( cFNane, .t., hb_TToC( hb_DateTime() ) + ;
              " - Se ha producido el error numero: " + HB_NToS( n ) )
    // Añade
    for n := 1 to 20
        writeLog( cFNane, .f., hb_TToC( hb_DateTime() )  + ;
                       " - Se ha producido el error numero: " + HB_NToS( n ) )
    next

    ? loadLog( cFNane )

    ? "Ahora creamos y leemos otro con el mismo nombre"
    ? "escribiendo diferentes tipos de datos:"

    Inkey( 100 )

    ?

    // Escribe en el fichero. Si existe lo destruye y lo crea denuevo (.t.)
    writeLog( cFNane, .t.,
"------------------------------------------------------------------------" )
    // Ecribe las siguientes lineas en el fichero existente
    writeLog( cFNane,, "Esta es la priemra linea..." )
    writeLog( cFNane,, Date() )
    writeLog( cFNane,, Time() )
    writeLog( cFNane,, 1367.89 )
    writeLog( cFNane,, .t. )
    writeLog( cFNane,, "" )
    writeLog( cFNane,, "Esto es todo..." )

    ? loadLog( cFNane )

    Inkey( 100 )

return
```

Otro con lectura de fichero externo y una funcion para leer el LOG:

```
//----------------------------------------------------------------------------
// Ejemplo de uso del sistema log manager
// Prgrama ej02.prg
//----------------------------------------------------------------------------

#include "Box.ch"
```

```
procedure main

    local n := 0
    local cFNane := "miArchivo.log"

    // Reiniciamos si existe el fichero
    // Simula errores
    writeLog( cFNane, .t., hb_TToC( hb_DateTime() ) + ;
              " - Se ha producido el error numero: " + HB_NToS( n ) )
    // Añade
    for n := 1 to 20
        writeLog( cFNane, .f., hb_TToC( hb_DateTime() )  + ;
                  " - Se ha producido el error numero: " + HB_NToS( n ) )
    next

    verLog( cFNane )

    cls

    ? "Ahora creamos y leemos otro con el mismo nombre"
    ? "escribiendo diferentes tipos de datos:"

    Inkey( 100 )

    // Escribe en el fichero. Si existe lo destruye y lo crea denuevo (.t.)
    writeLog( cFNane, .t., "------------------------------------------------------------" )
    // Ecribe las siguientes lineas en el fichero existente
    writeLog( cFNane,, "Esta es la priemra linea..." )
    writeLog( cFNane,, Date() )
    writeLog( cFNane,, Time() )
    writeLog( cFNane,, 1367.89 )
    writeLog( cFNane,, .t. )
    writeLog( cFNane,, "" )
    writeLog( cFNane,, "Esto es todo..." )

    verLog( cFNane )

    cls
    ? "Ahora vamos a leer cuaquier archivo, por ejemplo el código fuente ej01.prg"

    Inkey( 100 )

    verLog( "ej02.prg" )

return


//-----------------------------------------------------------------------------
// Muestra el contenido del archivo pasa usan el sistema log

static procedure verLog( cFileName )

    cls

    @ 01, 25 SAY "Contenido de: " + cFileName COLOR "GR+/R+"
    @ 02, 09, 21, 71 BOX B_DOUBLE_SINGLE + Space(1)

    MemoEdit( loadLog( cFileName ), 03, 10, 20, 70, .f. )

return

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

Y ahora una clase hecha en C que es un búfer para acelerar la escritura en un fichero:

```
//----------------------------------------------------------------------------
// Clase que gestiona un bufer de fichero
// Hecha casi toda en Lenguaje C

#include "hbclass.ch"

#define FB_MAX_SIZE                65535

CLASS TFBuffer

       DATA hFB  // Uso interno

       METHOD new( hFile, nSizeBuffer ) CONSTRUCTOR
       METHOD free() // Libera memoria
       METHOD creaBuffer( hFile, nSizeBuffer )  // Uso interno
       METHOD addString( cString ) // añade cadena al buffer, hace el ::flush() automaticamente
       METHOD addValue( value ) // convierte el valor en cadena y lo añade al buffer
       METHOD flush() // Por si se quiere salvar a disco manualmente

END CLASS

//----------------------------------------------------------------------------
// Constructor de la clase

METHOD new( xFile, nSizeBuffer ) CLASS TFBuffer

    local hFile
    local nType := ValType( xFile )

       if nType == 'N'
        hFile := xFile
    elseif nType == 'C'
        hFile := FCreate( xFile )
       else
              Alert( "Falta el nombre o el manejador del fichero" )
       endif

    if hFile > 0
        if ValType( nSizeBuffer ) != 'N' .or. nSizeBuffer < 0
            nSizeBuffer := FB_MAX_SIZE
        endif

        ::creaBuffer( hFile, nSizeBuffer )
    else
        Alert( "No se ha podido usar fichero " )
    endif

return self

//----------------------------------------------------------------------------
// Implementación de los métodos en Lenguaje C

#pragma BEGINDUMP

#include "hbapifs.h"
#include "hbapiitm.h"
#include "hbstack.h"

/* Estructura del bufer */
typedef struct
```

```
{
	unsigned char *pBuffer;
	unsigned int uiLen;
	unsigned int uiPos;
	HB_FHANDLE hFile;
} FBUFFER, *PFBUFFER;

static void __flushFB( PFBUFFER pFB );

//----------------------------------------------------------------------------
// Metodo para crear la estructura  buffer

HB_FUNC_STATIC( TFBUFFER_CREABUFFER )
{
	PHB_ITEM Self = hb_stackSelfItem();
	PFBUFFER pFB = ( PFBUFFER ) hb_xgrab( sizeof( FBUFFER ) );

	pFB->uiLen = hb_parnint( 2 );
	pFB->pBuffer = ( unsigned char * ) hb_xgrab( pFB->uiLen );
	pFB->uiPos = 0;
	pFB->hFile = ( HB_FHANDLE ) hb_parnint( 1 );

	hb_arraySetPtr( Self, 1, pFB );
}

//----------------------------------------------------------------------------
// Metodo para liberar el buffer

HB_FUNC_STATIC( TFBUFFER_FREE )
{
	PHB_ITEM Self = hb_stackSelfItem();
	PFBUFFER pFB = hb_arrayGetPtr( Self, 1 );

	if( pFB )
	{
		HB_BOOL fCloseFile = hb_parldef( 1, HB_FALSE );

		__flushFB( pFB );

		if( pFB->pBuffer )
		{
			hb_xfree( pFB->pBuffer );
		}

		if( fCloseFile && pFB->hFile )
		{
			hb_fsClose( pFB->hFile );
		}

		hb_xfree( pFB );
	}
}

//----------------------------------------------------------------------------
// Metodo para añadir cadenas al buffer

HB_FUNC_STATIC( TFBUFFER_ADDSTRING )
{
	PHB_ITEM Self = hb_stackSelfItem();
	PFBUFFER pFB = hb_arrayGetPtr( Self, 1 );
	PHB_ITEM pString = hb_param( 1, HB_IT_STRING  );

	if( pString )
	{
```

```
                unsigned int uiPos = 0;
                const char *szString = hb_itemGetCPtr( pString );
                unsigned int uiLen = hb_itemGetCLen( pString );

                while( uiPos < uiLen )
                {
                        if( pFB->uiPos == pFB->uiLen )
                        {
                                __flushFB( pFB );
                        }

                        pFB->pBuffer[ pFB->uiPos++ ] = ( unsigned char ) szString[ uiPos++ ];
                }
        }
}

//---------------------------------------------------------------------------
// Metodo para añadir cadenas al buffer

HB_FUNC_STATIC( TFBUFFER_ADDVALUE )
{
        PHB_ITEM Self = hb_stackSelfItem();
        PFBUFFER pFB = hb_arrayGetPtr( Self, 1 );
        PHB_ITEM pValue = hb_param( 1, HB_IT_ANY  );

        if( pValue )
        {
                unsigned int uiPos = 0;
                HB_SIZE uiLen;
                HB_BOOL fFree;
                char *szString = hb_itemString( pValue, &uiLen, &fFree );

                while( uiPos < uiLen )
                {
                        if( pFB->uiPos == pFB->uiLen )
                        {
                                __flushFB( pFB );
                        }

                        pFB->pBuffer[ pFB->uiPos++ ] = ( unsigned char ) szString[ uiPos++ ];
                }

                if( fFree )
                {
                        hb_xfree( szString );
                }
        }
}

//---------------------------------------------------------------------------
// Metodo para forzar escritura en disco del buffer e inicia la posicion

HB_FUNC_STATIC( TFBUFFER_FLUSH )
{
        PHB_ITEM Self = hb_stackSelfItem();
        PFBUFFER pFB = hb_arrayGetPtr( Self, 1 );

        __flushFB( pFB );
}

//---------------------------------------------------------------------------
// Funcion de uso interno. Escribe en disco el buffer e inicia la posicion

static void __flushFB( PFBUFFER pFB )
```

```
{
        if( pFB->uiPos > 0 )
        {
                hb_fsWriteLarge( pFB->hFile, pFB->pBuffer, pFB->uiPos );
                pFB->uiPos = 0;
        }
}

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

#pragma ENDDUMP
```

Un par de ejemplos de de uso de la clase TFBuffer:

```
//----------------------------------------------------------------------------
// Ejemplo de uso de TFBuffer
// Prgrama ej03.prg
//----------------------------------------------------------------------------

#include "tfbuffer.prg"

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

procedure main

        local hFile := FCreate( "prueba.log" )
        local o := TFBuffer():new( hFile )
        local i, t

        set date format to "yyyy-mm-dd"

        cls

    @ 09, 10 SAY "Se escribiran un millon de lineas."
    @ 10, 10 SAY "Espere, estoy procesando datos..."

        t := Seconds()

    // Para cadenas se puede usar el metodo ::addString() para lo demÃ¡s
    o:addString( "-------------------------------------------------------------" + hb_eol() )
        for i := 1 to 1000000
                o:addValue( i )                  // Numeric
                o:addValue( ( i % 2 ) == 0 )     // Logical
                o:addString( " <> " )            // String
                o:addValue( date() )             // Date
                o:addValue( " " )                // String
                o:addString( Time() + hb_eol())  // String
        next
    o:addString( "-------------------------------------------------------------" )

        t := Seconds() - t

        o:free()

        FClose( hFile )

        // Para calcular el tamaÃ±o en kB
        i := Hb_FSize( "prueba.log" ) / 1024

        Alert( "Ha tardado: " + AllTrim( Str( t ) ) + ;
```

```
              " segundos con un fichero de: " + AllTrim( Str( i ) ) + " kB")

return

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

Y otro ejemplo, la clase se encarga incluso de crear y abrir el archivo:

```
//----------------------------------------------------------------------------
// Ejemplo de uso de TFBuffer
// Prgrama ej04.prg
//----------------------------------------------------------------------------

#include "tfbuffer.prg"

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

procedure main

       local o := TFBuffer():new( "prueba.log" )  // Si se pasa el nombre lo crea y abre
       local i, t

       set date format to "yyyy-mm-dd"

       cls

    @ 09, 10 SAY "Se escribiran un millon de lineas."
    @ 10, 10 SAY "Espere, estoy procesando datos..."

       t := Seconds()

    // Para cadenas se puede usar el metodo ::addString() para lo demÃƒÂ¡s
    o:addString( "------------------------------------------------------------" + hb_eol() )
       for i := 1 to 1000000
              o:addValue( i )                 // Numeric
              o:addValue( ( i % 2 ) == 0 )    // Logical
              o:addValue( " <> " )            // String
              o:addValue( date() )            // Date
              o:addValue( " " )               // String
              o:addString( Time() )           // String
              o:addValue( hb_eol() )          // String
       next
    o:addString( "------------------------------------------------------------" )

       t := Seconds() - t

       o:free( .t. ) // true para que cierre el archivo

       // Para calcular el tamaño en Kb
       i := Hb_FSize( "prueba.log" ) / 1024

       Alert( "Ha tardado: " + AllTrim( Str( t ) ) + ;
              " segundos con un fichero de: " + AllTrim( Str( i ) ) + " kB")

return

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

20. Creando nuestras propias librerías o bibliotecas de funciones.

La mayoría de nosotros estamos familiarizados con el concepto de librería o biblioteca de funciones. De hecho la utilizamos a diario cada vez que construimos nuestros programas.

Pero **¿qué es una librería de funciones?**, podríamos decir que es un contenedor de funciones y procedimientos alojadas en un archivo, normalmente con una extensión ****.lib, *.a, *.dll*** o ****.so***. Ya las explicaremos más adelante que son cada una de ellas.

Ese archivo debería estar organizado con funciones y procedimientos con una misma funcionalidad, por ejemplo funcionalidades matemáticas, tratamiento de cadenas, salida por pantalla, listados etc.

Cuando tenemos una serie de funciones que utilizamos en más de un programa y su código está más que probado sería la hora de crear una biblioteca y con eso ganaremos en claridad y en facilidad.

Muchas de las librerías que usamos ya vienen con nuestro compilador favorito o nos la provee algún vendedor de software. Pero la buena noticia es que nosotros podemos crear las nuestras y así organizar mejor nuestro código y gestionar esas funciones y procedimientos que usamos en todos nuestros programas.

En este tema vamos a centrarnos en la creación de librerías para Harbour por lo que el código que introduzcamos en las mismas tiene que ser entendido por nuestro compilador. Eso quiere decir que tienen que estar hechas en PRG o en C (usando las diferentes API que Harbour proporciona).

Para poder usar las funciones y procedimientos que están incluidas en las librerías tenemos que saber los parámetros que reciben las funciones y el valor que devuelven. Esto es aplicable tanto para las librerías estáticas (*.lib o *.a) como para las dinámicas.

Básicamente hay dos ***tipos de librerías:***

1) Estáticas. Las funciones o procedimientos que la componen son incluidas en el ejecutable final una vez compilado y enlazado con ella por lo que ya no será necesaria su presencia para que nuestro programa se ejecute sin problemas. Se suelen identificar por su extensión ****.lib*** para los compiladores de Borland de 32 bits, MSVC, PellesC etc, todos ellos para entornos Windows y ****.a*** para todos los compiladores de entornos Unix, Linux, MacOs, etc y algunos compiladores portados a Windows como MinGW, cLang y los basados en cLang como por ejemplo las versiones 7.xx de 64 bits de Borland.

2) *Dinámicas*. Las funciones no quedan incluidas en el ejecutable que tan solo las referencia. Esto significa que la librería tiene que estar presente para que la ejecución del programa se pueda efectuar sin problemas. En entornos Windows la extensión abitual es ****.DLL*** y en entornos Unix, Linux, MacOs, etc la extensión suele ser ****.so***.

3) *De importación*. Son librerías estáticas que contienen la referencias a las funciones incluidas en las librerías dinámicas. Es una especie de tabla de símbolos. Sólo son necesarias para la compilación y enlace de nuestros programas y por tanto no tienen que estar presentes en tiempo de ejecución.

Como dije antes usamos continuamente las librerías proporcionadas por Harbour y los compiladores de C y otras de terceros como pueden ser las de FWH, Xailer o xHarbour.com. Pero lo importante es que también nosotros mismos podemos crear las nuestras propias. Y este es el motivo del presente tema…

Cada compilador de C tiene su propio método de construcción de librerías, por lo que habría que explicar como se hace en cada uno de ellos. La buena noticia es que la herramienta ***HbMk2*** que proporciona Harbour se encargara de interpretar los comandos necesarios para cada compilador de C quedando nosotros libres de tener que complicarnos.

Creo que la mejor manera de explicar como se hace es mediante un ejemplo que se podrá modificar para ser adaptada a cada necesidad de cada uno.

Para crear la librería usaremos la herramienta de Harbour HbMk2 así que vamos a necesitar saber alguna opción de la herramienta:

-o<outname> nombre del archivo de salida

-i<p>|-incpath=<p> rutas adicionales para buscar archivos de cabecera

-hblib crea una librería estática

-hbdyn crea una biblioteca dinámica (sin enlace a Harbour VM)

-hbdynvm crea una librería dinámica (con enlace a Harbour VM)

-implib=<output> crea una biblioteca de importación (en modo -hbdyn/-hbexe) denominada <output> (por defecto: igual que la salida)

Estas son las opciones básicas que vamos usar, HbMk2 tiene muchas más.

También deberíamos crear un archivo de proyecto donde incluiremos las opciones y los diferentes archivos en PRG y C que van a contener las funciones y procedimientos de nuestra librería. La extensión de estos archivos de proyecto de HbMk2 tienen la extensión *.hbp…

****.hbp*** archivo de proyecto. Puede contener cualquier número de opciones de la linea de comandos, que son los esperados para crear un objetivo final. La lineas que comienzan con el carácter "#" son ignoradas, por otra parte es opcional incluir caracteres de nueva linea y las opciones deben de estar separadas por espacios, como en la linea de comandos. Las opciones que contienen espacios deben encerrarse entre comillas dobles. Cada referencia a un archivo '.hbp' será ejecutado como un subproyecto.

Con todo esto ya estamos en disposición de crear nuestras librerías.

Normalmente se suele crear un archivo de proyecto donde se relacionan los ficheros fuente PRG y C y las opciones.

Este es el fuente código de un fichero de ejecución por lotes que hará la llamada a la herramienta ***HbMk2.***

Fuente ***buildlib.bat:***

```
@rem ----------------------------------------------------------------------
@rem Archivo por lotes para construir librerías
@rem Para adaptarlo a tu sistema cambiar las variables:
@rem COMP compilador de C para windows puedes usar mingw, mingw64, clang,
@rem      clang64, msvc, msvc64, clang-cl, clang-cl64, watcom, icc, icc64,
@rem      iccia64, msvcia64, bcc, bcc64, pocc, pocc64
@rem DIR_HBBIN  directorio bin de Harbour
@rem DIR_CCBIN  directorio bin del compilador de C
@rem ----------------------------------------------------------------------
@set COMP=mingw64
@set DIR_HBBIN=d:\mio\programacion\comp\xc\hb\bin
@set DIR_CCBIN=d:\mio\programacion\comp\cc\mingw\32\9.30\bin
@set PATHOLD=%PATH%
@set PATH=%DIR_HBBIN%;%DIR_CCBIN%;%PATH%
@hbmk2 -comp=%COMP% cursodec.hbp
@pause
@if %errorlevel% neq 0 goto bld_error
@goto fin_exec
:bld_error
@echo ----------------------------------
@echo    Hay errores en la compilación
@echo ----------------------------------
@pause
:fin_exec
@set PATH=%PATHOLD%
```

Y este es el primer modelo de archivo de proyecto de ***HbMk2****:*

```
#----------------------------------------------------------------------------
#  AUTOR.....: Manuel Expósito Suárez 2021
#  CLASE.....: cursodec.hbp
#  FECHA MOD.: 05/08/2021
#  VERSIÓN...: 1.00
#  PROPÓSITO.: Script de construcción de la librería para todos los compiladores
#              de C con Hbmk2 Harbour
#----------------------------------------------------------------------------

# Indica que vamos a construir una librería
-hblib

# Si hubiera ficheros includes para nuestra librería le indicaríamos su ubicación
# en nuestro caso no los hay
# -i./src/include

# Indicamos todos los PRG y el directorio que forman parte de la librería
./src/prg/tfbuffer.prg
./src/prg/thbuffer.prg

# Indicamos todos los C y el  directorio que forman parte de la librería
./src/c/cursode1.c
./src/c/cursode2.c

# Nombre de la librería y donde la tiene que dejar una vez construida
-o./lib/cursode

#----------------------------------------------------------------------------
```

Y este es un modelo que añade automáticamente todos los fuentes de una carpeta:

```
#-------------------------------------------------------------------------------
#  AUTOR.....: Manuel Expósito Suárez 2021
#  CLASE.....: cursodec.hbp
#  FECHA MOD.: 05/08/2021
#  VERSIÓN...: 1.00
#  PROPÓSITO.: Script de construcción de la librería para todos los compiladores
#              de C con Hbmk2 Harbour
#-------------------------------------------------------------------------------

# Indica que vamos a construir una librería
-hblib

# Si hubiera fichero includes para nuestra librería le indicamos su ubicación
# -i./src/include

# Indicamos que todos los PRG del directorio indicado forman parte de la librería
./src/prg/*.prg

# Indicamos que todos los C del directorio indicado forman parte de la librería
./src/c/*.c

# Nombre de la librería y donde la tiene que dejar una vez construida
-o./lib/cursode

#-------------------------------------------------------------------------------
```

Este último modelo tiene el inconveniente que si por error hay archivos que no forman parte de nuestra librería los incluirá también, por lo que recomiendo el primero.

La primera vez que se genera la librería suele tardar más que las siguientes ya que HbMk2 es inteligente y solo trata aquellos ficheros cuya fecha ha sido modificada por cambios en los mismos.