How to Interface Between C and the Harbour Language

Author: Manu Expósito Suárez

From the book: The New Buccaneers — Episode 2

Translation From Spanish to English made with the assistance of ChatGPT (GPT-5)

Spanish Version: April 2021 - Version 7.00

English Version: November 2025

Special Thanks To Manu Expósito Suárez for creating such a wonderful book and allowing its translation and publication.

Index

10. **Pointers in C**
11. **Dynamic memory allocation and release**
12. **Some basic concepts about the Virtual Machine (VM), stack, and symbol table of Harbour**
13. **Creating C functions to be used from Harbour PRG programs**
14. **How to compile C code in our PRGs**
15. **The Harbour Extended System**
    - Passing parameters from PRG to C
    - Returning values from C
    - Passing variables by reference
    - Handling arrays
    - Handling C structures
16. **The Item API — Extending the Extended System**
    - Input and output of parameters
    - Handling arrays
    - Handling hash tables
17. **Executing Harbour functions and codeBlocks from C**
18. **The Error API — Managing Harbour errors from C**
19. **The FileSys API — File manipulation from C**
20. **Creating our own libraries or function libraries**
21. **Interfaces to DLL functions**

1. Prologue and purpose of the course

Foreword by Rafa Carmona

One day, chatting with Manu Expósito, he mentioned the title of a book I wrote more than 19 years ago: *The New Buccaneers of the Planet, Part 1*. It was a book conceived and created simply for myself, opening with an introduction—among other things—about the magic that existed in the communication between the C language and Harbour, and it was released free for anyone interested in the subject.

Why do I bring this up? Because he told me he wanted to use the same name. "Well," I told him, "my book was Part 1, and if you want to use the same or a similar name, then you should call it Part 2, because Part 2 has been pending for more than 19 years—and it would be an honour for me if this became that Part 2."

I have known Manu Expósito for many years, and I know his stubborn determination to push Harbour's performance to the limit—and for that, C is king.

Much has happened since then, and with it the system between C and Harbour has been expanded and improved; Manu Expósito's Part 2 is needed now more than ever.

I have seen the syllabus that Manu is preparing for this Part 2 of *Los nuevos bucaneros*, and I know it will not disappoint—no one will be left indifferent.

Manu Expósito is one of the people who knows the most about the connection between C and Harbour. He wrote his famous **HDO** library in pure C—yes, everything is built on the C language and connected with Harbour. Even the classes you use in HDO have been created from C. You can be sure that what you are holding in your hands is pure gold, and it will take away that fear we all feel when we enter an unknown and almost magical world like the C Language, the undisputed king. Long live C! Long live Harbour!

And now, my part... (by Manu Expósito)

For many years I've wanted to write some notes to use personally.

A long time ago, I mentioned this to a colleague, and he suggested that those notes could become a book and be published on some platform so that anyone interested could pay a small amount and benefit from it. That would also be a reward for me.

I tried it at the time, but I don't think I was very enthusiastic. Now the necessary circumstances are coming together to roll up my sleeves and start this adventure. I simply feel like doing it...!!!

Back in the day, my friend Rafa Carmona wrote a book called *Bucaneros*. That book covered everything related to programming in Harbour but at a high level in PRG, with a small reference to Harbour's extended system. By the way, I've asked Rafa if he wants to write a few notes for us, which I'll gladly add to this prologue.

A lot has happened since then… Harbour has come of age, and the extended system is now very mature. It's time to discuss these topics in depth.

I must say that Harbour includes "out of the box" almost everything we may need—and in a very optimal way. So it will be **almost** unnecessary for us to enter the seas of C. But this book is written for that *almost*.

For example, if we want to integrate dynamic link libraries (DLLs), or develop routines that make certain critical bottleneck functions faster, it will be necessary to reach for the omnipresent C language. After all, Harbour itself is built in C.

Luckily, Harbour inherits from the legendary Clipper the **Extended System** and the **ITEM API System**, which is the method by which we can do "magic." And unlike what happened with Clipper, in Harbour there are no undocumented black boxes—we have all its source code available to us.

The purpose of this book is to serve as a **logbook** for this voyage—a journey we know how it begins, but not how it ends, because it is yet to be written. This book is alive; it will be fed by what you all contribute.

The idea is that I will write it topic by topic and deliver it to you in **PDF** and **ePub** format so you can use it as a notebook, expanding what is necessary. If you want, you can later send your additions back to me to enrich what will eventually become the book—once finished, all registrants will receive it.

The ultimate goal is that when the voyage across the **Seven Seas** is over, we will all be capable of writing our own functions in C to improve the robustness and performance of our programs, and we'll be able to create our own libraries…

So… let's not waste a single second!!!

The curtain opens and the adventure begins…

2. Tools Needed for the Course

Before we dive in, you'll need at least the following tools to complete the course exercises:

1. A text editor.
2. A C language compiler.
3. Harbour, with libraries built for the chosen C compiler.
   You may use whichever ones you prefer, since the course is independent of target (32- or 64-bit), C compiler, and operating system.

Author's working setup (as a recommendation):

1. VSCodium as the editor
2. MinGW 9.30 (64-bit)
3. Harbour with LIBs built using the compiler in item 2
   But again, you can keep using what you already have.

As a project/build manager we'll use **HbMk2**, which comes with your Harbour installation.
It's powerful and greatly simplifies building executables and libraries.

3. Introduction — A Little Bit of C

In this section, we will very quickly review the history of the C language.
There are many accounts of how the C language was born, so this will be brief, and we will use the opportunity to discuss some basic programming concepts.

The C language was born a few years before Clipper and, therefore, before Harbour.
It was the year **1972** when a bearded man named **Dennis Ritchie** published the first version.
He worked in the legendary **Bell Labs** of an American telecommunications company, **AT&T**.

But why the name *C language*?
Well, yes — because previously there had been a *Language B* and later a *Language D*.
I think good old Dennis didn't strain his brain too much to baptize it.

Initially, C was used to implement the **UNIX operating system**, the father of all modern Linux systems.
It was later used to build other operating systems (OS) such as **Linux**, **iOS**, and **Windows**.

It is such a powerful, robust, and structured language — with very few keywords.
All of this has made it the language used to develop other languages, like our beloved **Harbour**.

3.1 Types of Languages

As you know, there are several types of programming languages:

• Low-Level Languages

They are completely tied to the machine.

- **Machine Language**, which is the only one our computer truly understands.
- **Assembly Language**, which is an evolution of machine language that assigns abbreviated names to machine instructions.

• High-Level Languages

They are independent of the machine on which they will run and resemble human natural language (which happens to be English).
In this category are **C**, **Harbour**, **Java**, **PHP**, **Basic**, **Pascal**, **C#**, **Fortran**, **COBOL**, etc.

Among these languages, some are general-purpose and others are specialized for specific tasks.
For example, **C** and **Harbour** are general-purpose, while **Fortran** specializes in mathematical computation and **COBOL** in business management programs.

For a computer to understand high-level languages, they must be converted into machine language.

3.2 Types of Translators

There are several types of translators:

- **Assemblers**: generate machine code from assembly language.
- **Interpreters**: verify and translate each instruction line by line into machine code and then execute it.
- **Compilers**: analyze and translate the entire program into machine code before execution.

Each type of translator has its advantages and disadvantages (we can talk about that in class 😉).

C belongs to the family of **compiled languages**.
When we build a C executable, it generates an error-free **OBJ** file, and then the **linker (LINK)** retrieves the necessary components from **LIBs** or other **OBJs** to generate a standalone **EXE**.

The **BASIC** language is the typical example of an interpreted language: it verifies each instruction as it encounters it, translates it into machine language, and executes it.

Harbour is interpreted like BASIC, except that — to make the process faster and more efficient — it creates a highly optimized code that is not pure machine language but which we mere mortals could not decipher.
That code is called **pCode**, and it is then interpreted by the **Harbour Virtual Machine (HVM)**.

Harbour's pCode executes very quickly under the virtual machine and is also free from lexical errors.

If you compile a PRG with the `-gh` option, a file with the extension `*.hrb` is generated containing the pCode.
In Harbour this is called a **portable binary**, and it can be executed using the included tool **HbRun.exe**.

But you may ask: does Harbour also generate an EXE like the C language?
The trick is that both the **virtual machine** and the **pCode** are actually embedded inside the EXE — that's why those EXEs are so large!
But don't be alarmed: that "obesity" doesn't stop them from being extremely fast at the same time.
For **Mod Harbour**, this behavior is fantastic!

Let's leave Harbour aside for a moment and return to C.

3.3 Dissecting C Programs

C is a very structured language, organized around functions or procedures.
In C, the term *function* is generally used for both — after all, the only difference is that a function returns

a value while a procedure does not.
Both can receive parameters or not.

In **Harbour**, we must indicate to the compiler whether we are defining a `FUNCTION` or a `PROCEDURE`; failing to do so may cause a warning or an error during compilation.

By the way, there is great similarity between both languages in this regard.

Unlike Harbour, **C is case-sensitive** — meaning it distinguishes between uppercase and lowercase.
For C, "Value" is not the same as "value."
Although this might be tricky at first, it can actually be useful.
For example, in this course, all functions created for Harbour will be written in **uppercase**, while those in C will use **Hungarian Notation**, ensuring that names never collide.
You can find a lot of material online about different naming conventions.

Just like in Harbour, if there is a function called **main**, it will be the first one executed.

In C, a function is a way to group instructions in a structured and coherent way to perform a task — coincidentally, the same as in Harbour.
In other words, a function is simply a set of instructions that perform a specific task.

3.4 The Skeleton of a Typical C Function

```c
/* This is a comment */
#include <stdio.h>

return_type function_name(type parameter, …)
{
    local_variable_declarations;
    …
    statements; // This is a comment
    …
    return variable_of_the_type_defined_by_the_function;
}
```

3.5 Comments

The two types of comments in C also exist in Harbour and work the same way:

- `/* ... */` for comments that can extend across multiple lines, and
- `//` for inline comments.

In both cases, the compiler ignores them.
They are used only for annotations among programmers.

3.6 Preprocessor Directives

In this, Harbour and C also coincide in how they are defined, though each has its own specific directives and some overlaps.
We will start by looking at the two main ones:

`#include`

This directive tells the compiler to include another file within the current source file.
The included file usually has the extension `.h`, although it may have any extension.
Angle brackets `< >` are used when including system headers (those that come with the compiler), and quotes `" "` are used for local include files.

This works exactly the same in Harbour, except that include files there typically have the `.ch` extension.
As you can see, both compilers handle this identically.

In C, before you can use something, it must be defined.
That is the main purpose of `.h` header files, which are usually included at the beginning of programs.

Examples:

```
#include <stdio.h>
#include "hdo.api"
```

`#define`

Used to define symbolic names that will not change throughout the program — that is, constants.
Fortunately, this works exactly the same in both compilers.
The compiler replaces the symbol with its actual value at compile time.

Examples:

```
#define MAXIMUM 90
#define MINIMUM 25
```

3.7 Function Names, Definitions, and Implementations

The **function name** is the identifier we use whenever we want to call it.
Therefore, it is very important that the name be self-descriptive.
As mentioned earlier, it must be defined before being used.
The declaration must specify the type of value it returns, its name, and — in parentheses — the parameters it receives, each preceded by its type and separated by commas.

Example — **function declaration**:

```
int double_value(int);
```

Now the implementation:

```
#include <stdio.h>
/* Function that returns twice the integer value passed */
int double_value(int iNum)
{
    int iRet;
    iRet = iNum * 2;
    return iRet;
}
```

In the declaration, it's not necessary to name the parameter variables — only their types — but in the implementation, it is mandatory.
Next come the curly braces `{}` — everything between them forms the **body** of the function.
Be careful: every instruction ends with a semicolon (`;`).

Another example with more than one parameter:

```
#include <stdio.h>

int multiply(int iParam1, int iParam2)
{
    int iRet;
    iRet = iParam1 * iParam2;
    return iRet;
}
```

4. Operators

4.1 Arithmetic Operators

Operator	Meaning
`=`	Assignment
`*`	Multiplication
`/`	Division
`%`	Remainder of integer division (*mod*)
`+`	Addition
`-`	Subtraction
`++`	Increment
`--`	Decrement

4.2 Relational or Comparison Operators

Operator	Meaning
`<`	Less than
`<=`	Less than or equal to
`>`	Greater than
`>=`	Greater than or equal to
`==`	Equal to
`!=`	Not equal to
`&&`	Logical AND → `.and.` in Harbour
||	Logical OR → `.or.` in Harbour

There are more operators in C, but we will stop here for now.
It's worth noting that all of these work the same way in **Harbour**.

You will often come across the word **expression**, but what does it mean?

Expressions are sequences of functions, operators, and variables or constants used to calculate a value once evaluated.

5. Control Structures

In this section, we will review the different programming structures in **C**, each with its corresponding syntax diagram.

5.1 Conditional Structures

`if`

Works similarly to how it does in **Harbour**:

```
if ( condition )
{
    statements;
}
```

`if ... else`

```
if ( condition )
{
    statements;
}
else
{
    default_condition_statements;
}
```

`if ... else if`

```
if ( condition1 )
{
    statements;
}
else if ( condition2 )
{
    statements;
}
// There can be more else if() blocks
else
{
    default_condition_statements;
}
```

Ternary Conditional Operator `?`

It has the following syntax:

```
expression1 ? expression2 : expression3;
```

In **Harbour**, there is an equivalent construct that works exactly the same way, using `iif()`:

```
iif( expression1, expression2, expression3 )
```

If `expression1` evaluates to true, the result takes the value of `expression2`; otherwise, it takes the value of `expression3`.

Example: Finding the maximum between two numbers:

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

Equivalent to:

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

`switch`

Works similarly to Harbour's `do case` structure, but in C it can only be used with integer or character expressions.
It behaves like nested `if` statements.

```
switch ( expression )
{
    case exprConst1:
        statements;
        break;

    case exprConst2:
        statements;
        break;

    case exprConstN:
        statements;
        break;

    default:
        default_statements;
        break;
}
```

5.2 Iterative Structures

`while`

```
while ( condition )
{
    statements;
}
```

`do ... while`

This construct **does not exist in Harbour**.
It guarantees that the body statements are executed **at least once**.

```
do
{
    statements;
}
while ( condition );
```

`for`

Its syntax is quite different from Harbour's:

```
for ( initialization_block; conditional_expression; increment_block )
{
    statements;
}
```

- **initialization_block**: evaluated once; initializes one or more variables.
- **conditional_expression**: evaluated on each iteration; checks the loop's exit condition.
- **increment_block**: executed after each iteration; usually increments or decrements control variables.
- Each block can contain more than one expression, separated by commas `,`.

Example: prints the multiplication table of a given number `n`:

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

5.3 Control Statements: `continue` and `break`

For these kinds of programming structures:

- `continue` forces the **next iteration** inside a loop.
- `break` forces an **exit** from a selective or repetitive structure.

6. Elementary Data Types in C

Now things are getting interesting...

> "You are my only constant in this world of variables!"
> Nice phrase, isn't it? But what does it really mean?

Every program needs to store the information it will use.
That memory space is used by **variables** and **constants**.

In **Harbour**, there is an additional concept beyond variables and constants — **FIELDs**, which hold the values of DBF table fields.

6.1 Variables

Formally, a variable is an *object with a name* that can contain a value which may be modified during the program's execution.

In **C**, variables have a *type* that must be specified.
They are stored in memory, and the amount of space they occupy depends on their type.

In **Harbour**, it is not necessary to declare the type since it depends on the value assigned to it.
(We'll talk later about why Harbour is so clever!)

Examples:

```
char cAracter;
long lNum;
char *szCadena;
```

6.2 Constants

A **constant** is like a variable, but its value cannot change during program execution.

The best way to declare a constant is using `#define`, but it is also possible to use the qualifier `const`.

Examples:

```
#define TRUE 1
#define FALSE 0
#define MAX_AGE 90
```

`10`, `-5`, `"Manu"` are also constants.

In this case, these are called **symbolic constants**, and they are used to make programs more understandable and easier to maintain.
The compiler automatically replaces the symbol with its actual value.

For example, if `MAX_AGE` later needed to change from 90 to 100, you would only have to modify it in the `#define` line, not everywhere in the code.

This method of defining constants can also be used in **Harbour**.

Another way to define constants in **C** is with the `const` keyword:

```
const int iMaxAge = 90;
const char *szName = "Manu";
```

However, standard C can sometimes be fooled by this second form, so the `#define` method is generally preferred.

6.3 Declaring Variables and Understanding Data Types

After discussing variables and constants, let's see how they are declared.
This will lead us to the concepts of **types**, **modifiers**, and **qualifiers**.

A variable is declared as follows:

```
type name;
```

Unlike in Harbour, in C you must **declare the variable before using it**, and you must also specify its type.

The main types in C are:

Type	Description
`int`	Integer numbers
`char`	Characters
`float`, `double`	Real numbers

Each occupies a certain amount of memory, and the compiler must know the type to reserve the necessary amount.
That's why C is said to be a **strongly typed language**.

6.4 Modifiers

There are modifiers that affect how much memory is reserved for a given type.

For integers (`int`), the most common are `short` and `long` .

Examples:

```
char cLetter = 'A';
int iAge;
long int lDistance;
short int sHeight;
```

The last two could also be declared without explicitly writing `int` :

```
long lDistance;
short sHeight;
```

6.5 Character Strings

A string of characters can be declared as follows:

```
char *szString = "This is a string"; // or
char szString[] = "This is a string";
```

We'll go deeper into this topic later.
For now, remember that in C it's **mandatory to specify the type**, and that **each type occupies a specific space in memory**.

6.6 Character Type

In C, the `char` type is treated as an **integer of one byte**, meaning it can also be modified with `unsigned` , just like integers.

Examples:

```
unsigned int uiAge;
unsigned long ulDistance;
unsigned char cLetter;
```

6.7 Data Type Sizes (on i686 machines)

Type	Size (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

Remember: **1 byte = 8 bits**.

Unfortunately, the size of `int` can vary depending on whether the platform is 32-bit or 64-bit.
For this reason, it's best to avoid relying too much on plain `int` and instead use explicit sizes when possible.

That concludes the discussion of **elementary, simple, or primitive types** in C.

7. Structured Data Types: Arrays, Vectors, Matrices, and Multidimensional Tables in C

First, what are they?

They are called *structured* or *complex types* because they are composed of the **elementary types** already discussed.
This same definition will also apply to the structures we'll examine in the next chapter.

The concept of a *table* is exactly the same as in **Harbour**, but only conceptually.
In general, we can say that a table is a **structured variable** that contains elements of the same type, stored consecutively in memory.

We must remember that an array (or table) in **Harbour** is a variable of type `'A'` that can contain elements of **any type** supported by Harbour.
In **C**, however, arrays can contain **only elements of the same type**, which is a major difference.

C needs to know the size of the type that makes up the array in order to lay it out properly in memory. Among other things, this allows the program to iterate through its elements using repetition statements such as `for` or `while`.

You can calculate the number of elements in an array by dividing the total memory occupied by the array by the size of one element.
The `sizeof` operator is very useful for this, as it returns the size of a type or variable.

Example:

```
iWidthInt = sizeof( int ); // Note: 'int' is the type, not a variable
```

7.1 Vector (One-Dimensional Array)

A **vector** is a one-dimensional array, meaning its contents are not other arrays.

Declaration:

```
type name[ dimension ]; // dimension = number of elements
```

Example:

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

Important: in C, there is **no native string type**.
In C, a *string* is a **character array**.

Examples (all equivalent):

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

Or using pointer notation:

```
char *szString = "hola";
```

This pointer notation will be covered later in the topic on **pointers**.
For now, just take my word for it. 😳

Notice in the first notation the special character `'\0'`, which indicates the end of the string.
That's why, in my Hungarian notation, I use the prefix **`sz`** in variable names like `szString` — to remind you that it's a **string (s)** terminated by **zero (z)**.

7.2 Matrix (Two-Dimensional Array)

A **matrix** is a two-dimensional array — that is, it contains two vectors.

Declaration:

```
type name[ dim1 ][ dim2 ];
```

Example:

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

7.3 Multidimensional Array

A **multidimensional array** is an array containing more than two vectors.
Technically, a matrix is a two-dimensional multidimensional array — but we programmers like to make that distinction.
In reality, vectors and matrices are the most commonly used forms.

Multidimensional arrays are rarely initialized this way, but the following example is given to help visualize them.

Declaration:

```
type name[ dim1 ][ dim2 ][ dim3 ]; // etc.
```

Example:

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

But really, C doesn't care about visual representation.
It simply uses the dimensions specified in the brackets to manage memory slicing.

So yes — you could write the same initialization *linearly*, and the C compiler would not complain.
Even better, it would treat it exactly the same:

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

7.4 Accessing Array Elements

To access or assign a value, use **indices in brackets**:

```
lDistance[2][1][1];
```

Important: in C, array indices start at **0**, not **1** as in Harbour.

Example:

```
char szString[5] = "hola";
// To get the second character:
szString[1]; // The first would be szString[0]
```

8. Structured Data Types: Structures, Unions, and Enumerations

8.1 Structures

A **structure** is a complex data type because, like arrays, it can contain one or more members.
As my professor used to say:

> "A structure is a finite set of elements of any type grouped under a single name to make their use more efficient and intuitive."

Each element of a structure is called a **field** or **member**.
Unlike arrays, however, the elements of a structure are **not guaranteed** to be stored consecutively in memory.

In other languages like **Pascal**, structures are known as *records*.
In **Harbour**, they don't exist as such, but can be emulated with **classes** that only contain *datas* or instance variables.

To declare a structure, we use the `struct` keyword followed by the name and a list of members enclosed in braces `{}`.

Example:

```
struct person {
    char NIF[10];
    char name[30];
    int age;
    int gender;
};
```

⚠ **Important:** Declaring a structure only *describes its composition* — it does **not** declare a variable or reserve memory.

To compare this with **Harbour**, it's like defining a class with the `CLASS` clause — the variable is only created when you instantiate it, e.g.:

```
local oPerson := TPerson():new()
```

In **C**, to declare a variable of a given structure type (using the above example):

```
struct person student;
```

You can also declare a structure and its variable at the same time:

```
struct {
    char NIF[10];
    char name[30];
    int age;
    int gender;
} student;
```

This form is typically used when the structure definition will only be used once.

Later, we'll see a more elegant and intuitive way to define our own "data types" using `typedef`.

Initializing Structures

A structure can be initialized at the same time it's declared:

```
struct person student = { "98879345J", "Pablo", 21, 0 };
```

There's also a lesser-known but perfectly valid method:

```
struct person student = { .age = 21, .name = "Pablo" };
```

Accessing Structure Fields

To access or modify structure fields:

```
student.age = 19;
student.gender = 1;
```

If you have a pointer to a structure, you use the **arrow operator (`->`)** instead:

```
student->age = 19;
```

We'll explore this in detail later when we cover **pointers**.

As a closing note, it's worth mentioning that the **Harbour API** is full of structures — a sign of how useful they are.
In fact, from the perspective of C, a Harbour **ITEM** is implemented as a structure.

8.2 Unions

Unions are very similar to structures, with one key difference:
all their fields start at the **same memory address** — they occupy the same area of memory.
Therefore, a union can be interpreted as **different types at different times**.

The size of a union in memory is **not the sum** of its members' sizes,
but rather the size of its **largest field**.

They are declared exactly like structures, except the keyword is `union` instead of `struct`.

Example:

```
union myItem {
    int integer;
    float real;
    char string[30];
};
```

Everything said about structures applies equally to unions — declaration, assignment, and access.
The main difference is how memory is shared among members.

Interestingly, **Harbour variables are based on unions**.
In fact, a Harbour variable's representation in C is a structure that includes a field indicating its type, plus a union containing each of the possible Harbour data representations.

That structure is known as `HB_ITEM`.

8.3 Enumerations

Just like structures and unions, **enumerations** allow us to define new data types, using the keyword `enum`.

Enumerations let us define a set of values that represent some discrete, finite parameter —
for example, months of the year or days of the week.

Example:

```
enum months { jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };
```

We can then declare a variable of this type:

```
enum months monthOfYear;
```

These identifiers are **not strings**, but symbolic constants.
Internally, C converts each identifier into an integer, starting from **0** and increasing consecutively.

We can also assign specific integer values manually:

```
enum months { jan = 1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };
```

This version starts from **1** instead of 0 — all subsequent identifiers automatically increase by 1.

You can even assign independent values:

```
enum pets { dog = 101, cat = 202, hedgehog = 305, parrot = 418 };
```

In C, each identifier in an enumeration becomes an integer constant, so you can use them in arithmetic expressions:

```
dog + parrot; // 101 + 418
```

Defining a Boolean Type

We can create our own Boolean type in C using an enumeration:

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

Summary

- **Structures** group multiple fields of possibly different types under one name.
- **Unions** allow different data types to share the same memory space.
- **Enumerations** define symbolic constants as integer values, improving readability and safety in code.

These constructs make C more expressive and organized, and they are heavily used throughout the **Harbour C API**.

9. Creating Our Own Types with `typedef`

C allows us to assign **synonyms** (aliases) to any data type — both primitive and structured.
This makes it much easier to define variables of those types, especially when dealing with **structures**, **unions**, and **enumerations**.
To achieve this, we use the keyword `typedef`.

For example, given the following structure:

```
struct person {
    char NIF[10];
    char name[30];
    int age;
    int gender;
};
```

We can create our own type as follows:

```
typedef struct person TPerson;  // Now we have our own type TPerson
```

And then declare a variable of this new type:

```
TPerson student;
```

As you can see, it's simpler than writing:

```
struct person student;
```

In Harbour's include files, there are many types defined in this way —
for instance, `PHB_ITEM`, which we'll encounter frequently throughout this book.

In Harbour, you'll also find typedef-based definitions such as `HB_INT`, `HB_LONG`, `HB_BOOL`, etc.
It's **recommended** to use these when writing C functions for Harbour, instead of the plain C types `int`, `long`, etc.
(Note that the native C language did not originally have a `bool` type.)

The combination of `typedef` and structures greatly simplifies the use of these types in C code.

Important Summary — Data Type Recap

Declaring a Variable

```
type variable;
```

Example:

```
int age;
```

Assigning a Value to an Existing Variable

```
age = 57;
```

⚠ Important Notice — Variable Scope and Lifetime

Scope (Visibility)

Depending on where a variable is declared, it can be:

- **Local:** declared inside a function; visible only within that function.
- **Global:** declared at the beginning of a program and outside any function; visible throughout the program —
  and even in other programs if preceded by the reserved word `extern`.

Lifetime (Duration)

- **Global variables** remain valid for the entire duration of the program (and can even persist across modules when declared with `extern`).
- **Local variables** cease to exist once the function they belong to finishes execution.
- When a variable is declared as `static`, its value persists until the program ends —
  this behavior is similar to **Harbour**'s static variables.

Type Casting (Forced Conversion)

If you wish to store a value of one type into another, you can **cast** it explicitly.
For example, to store an `int` into a `long`, you can cast using the following syntax:

```
(type_name) expression
```

Example:

```
int sum = 17, counter = 5;
float average;

average = (float) sum / counter;
```

Casting allows explicit conversion between compatible types.

That concludes the topic of **variables**.

10. Pointers in C

As my friend Antonio says — this topic is for those aiming for top grades! 😉
It may seem complex at first, but once you grasp the concept, it's not so bad.
Let's dive in...

Before using any variable in C, you must declare it — and optionally, you may also initialize it.
Every variable has a **value** that occupies a **memory space**.

For a program to access a variable's value, it must know:

1. where it is located in memory,
2. how much space it occupies, and
3. what type it is.

In simple terms, a **pointer** is a variable whose **content is the memory address** of another variable.

You can imagine memory as a street full of houses of equal size, each with a number (its address).
If you know the number, you can send a letter to that house — that's your pointer to that house.
(Hopefully, that analogy helps! 😄)

10.1 Declaring Pointers

```
type_variable * variable_name;  // Note the "*" symbol
```

As you can see, a pointer is declared like any other variable, specifying a type (which can be primitive or complex) and prefixing its name with an asterisk (`*`).

Examples:

```
int *piAge;        // Pointer to int
float *plDistance; // Pointer to float
```

The "p" prefix is just a naming convention I use to indicate that the variable is a pointer — it's not required.

There's a special type of pointer called a **void pointer**.
These are pointers that don't have a specific type yet. When we eventually know the type, we must **cast** it, like so:

```
void *pUnknown;
(int*) pUnknown;
```

Later, when assigning a specific value, we'll need to perform a **forced cast**:

```
(type_name *) pUnknown;
```

We'll revisit this when discussing **dynamic memory allocation**, but for now, just remember that `void *` pointers exist.

10.2 Initializing Pointers

Declaring a pointer to a certain type is **not** the same as declaring a variable of that type.
To avoid problems, always **initialize pointers** when you declare them.
If you don't yet have a valid address to assign, use the special value `NULL`.

Examples:

```
int *piAge = NULL;          // Pointer to int
float *plDistance = NULL;   // Pointer to float
void *pUnknown = NULL;      // Untyped pointer
```

This practice prevents many problems, because an uninitialized pointer can contain **any random value**, which could crash your program.

10.3 Pointer Operators

C provides two key operators for working with pointers:

1. **Address-of operator (`&`)**
    - Placed before a variable, it returns that variable's memory address.

    Example:

    ```
    int *piNumber = NULL;
    int iNumber = 100;

    piNumber = &iNumber;  // piNumber now contains the address of iNumber
    ```

2. **Dereference operator (`*`)**

- Placed before a pointer variable, it returns the value stored at that address.

Example:

```
int *piNumber = NULL;
int iNumber = 100;

piNumber = &iNumber;  // piNumber points to iNumber

DimeN( iNumber );   // Displays 100
DimeN( *piNumber ); // Also displays 100
```

⚠ **Important:**
The `*` operator is used in two distinct contexts:

- In a **declaration**, it indicates that a variable is a pointer.
- In an **expression**, it means "dereference this pointer."

So `*` is an **overloaded operator** in this sense.

You **cannot** use dereferencing with `void` pointers directly because C needs to know the data type and memory size to retrieve the value properly.

Also, just because two pointers contain the same value (i.e., they point to the same address) doesn't mean the pointers themselves are identical objects.

The `&` operator can only be applied to **variables**, while `*` applies to **pointers and pointer expressions**.

Invalid example:

```
&( iNumber + 10 ); // ❌ incorrect
```

10.4 Pointer Arithmetic

C allows **adding or subtracting integers** from pointers.
However, this is not the same as normal arithmetic with integers or floating-point numbers.

The allowed pointer operations are:

- Add or subtract an integer, resulting in another pointer of the same type.

Pointers increment (`+`) or decrement (`-`) according to the size of the data type they point to.
If you increment a pointer by one, it moves to the **next element** of that data type in memory.

This topic will be explored further when we discuss **memory management** later.

10.5 Pointer to Pointer

Like any variable, a pointer can also **point to another pointer**.
We'll see its usefulness later, especially when dealing with **strings** and **dynamic memory**.

10.6 Passing Variables by Reference

In C, pointers can be used to **pass variables by reference** — meaning changes made inside a function affect the variable in the calling function.

Example:

Function definition:

```
void myFunc( int *piOne, int *piTwo );
```

Usage:

```
int iOne = 5;
int iTwo = 7;

myFunc( &iOne, &iTwo ); // Changes to iOne and iTwo will be visible here
```

This works exactly the same way as passing **by reference** in **Harbour**.

10.7 Example: Swapping Values

Here's a practical example — a function that swaps the values of two variables:

```
// Swaps the values of x and y
void swap( int *x, int *y )
{
    int temp = *x;
    *x = *y;
    *y = temp;
}
```

Pointers are an essential concept in C, and once you understand them,
you'll start to see how much control they give you over memory and performance.

11. Dynamic Memory Allocation and Release

When we declare a variable, the compiler automatically allocates the memory that variable will use. However, there are situations where this is not possible — for example, arrays that may grow during the program's execution.

C provides three primary functions for **dynamic memory management**:

11.1 Standard C Memory Functions

```
void *malloc( size_t size );
```

Reserves the amount of memory specified by the `size` parameter and returns a pointer to the beginning of the allocated block.
If the allocation fails, it returns **NULL**.

```
void *calloc( size_t number, size_t size );
```

Similar to `malloc`, this function also attempts to allocate memory and returns **NULL** if unsuccessful.
It differs in two ways:

1. It allocates memory for `number × size` bytes.
2. It initializes all allocated bytes to **zero**.

Use **`malloc()`** when allocating memory for a single element, and **`calloc()`** when allocating space for multiple elements (such as an array).

```
void free( void *pointer );
```

Releases the memory previously allocated by either `malloc()` or `calloc()`.

All three functions require **type casting**, since they return or receive a `void *` type pointer.

11.2 Memory Allocation in Harbour C Extensions

⚠ **IMPORTANT:**
Do **not** use these standard C memory functions in your C functions intended for **Harbour**,
because Harbour's standard library provides its own specialized memory management API.

These Harbour API functions offer automatic integration with Harbour's internal memory tracking and are safer to use in Harbour extensions.

The functions we will use throughout this book are:

Function	Description
`void *hb_xalloc( HB_SIZE nSize );`	Allocates `nSize` bytes of memory and returns `NULL` if it fails.
`void *hb_xgrab( HB_SIZE nSize );`	Allocates `nSize` bytes of memory and **terminates the program** if it fails.
`void *hb_xrealloc( void *pMem, HB_SIZE nSize );`	Resizes the memory block referenced by `pMem` to the new size `nSize`.
`void hb_xfree( void *pMem );`	Frees the memory pointed to by `pMem`. Returns nothing.

There are other Harbour API functions related to memory management,
but these are the ones we'll use most frequently.

Important note: Always use the Harbour API functions whenever possible —
for memory, string manipulation, file I/O, etc.
We'll explore those later.

11.3 Examples

Allocating and Freeing Memory

```
char *szName = ( char * ) hb_xgrab( 20 );
// Now szName can be used. If the allocation fails, the program will terminate.
```

With `hb_xalloc()`, we must explicitly check whether the memory was allocated:

```
char *szName = ( char * ) hb_xalloc( 20 );
if( szName != NULL )
{
    // szName can be used
}
else
{
    // Handle the allocation error
}
```

To **resize** an existing memory block:

```
szName = hb_xrealloc( szName, 50 );
```

To **free** allocated memory:

```
hb_xfree( szName );
```

Remember: every block of memory allocated with these functions must be explicitly freed,
since **C does not have a garbage collector**.

Throughout this book, you'll often see `hb_xgrab()` and `hb_xfree()`.

11.4 A Note on Pure C vs. Harbour C

Up to this point, we've covered "pure C" concepts.
Many topics remain to be explored, and some will be discussed later in more depth.
Others will remain outside the scope of this book.
But hopefully, these notes will serve as a foundation for understanding the vast C literature available online.

I have intentionally not included traditional pure-C examples here to avoid confusion.
For example, a simple "Hello World" program in **pure C** might look like this:

```
void funHelloWorld( void )
{
    printf( "Hello world" );
}
```

However, if we want to create a function callable from **Harbour**,
we cannot use the C standard output functions such as `printf()`,
since Harbour handles screen output through its **GT (General Terminal)** system
or other standard Harbour libraries.

The equivalent Harbour-compatible function would be:

```
HB_FUNC( FUNHELLOWORLD )
{
    hb_gtOutStd( "Hello world", 10 );
}
```

As you can see, there are important differences and rules to follow
when writing C functions intended for use from Harbour code (`.prg` files).

From this point on, we'll focus entirely on **Harbour** —
and how to use C to expand the capabilities of our favorite language.

12. Some basic concepts about the Harbour Virtual Machine (VM), stack, and Symbol Table

We'll continue with another theoretical topic that, although a bit abstract, I believe is necessary to understand how the C functions we're going to write will work when used from PRG. And, why not, it also helps us understand how Harbour programs work internally.

To begin with, any C functions that do not meet certain requirements (which we will explain later) will **not** be recognized by Harbour's internal system.

Harbour variables

Like any language, Harbour needs to store the information it uses in **variables**.
Each of these variables has an identifier (its name) and a **value**, which we assign at any moment during the program's execution. A particular feature of Harbour is that these variables can be **of any type**, simply by assigning a value of a different type to them.

For example:

```
procedure main
    local miVar
    miVar := "Hello"
    alert( "The variable miVar is of type: " + valtype( miVar ) )
    miVar := 9
    alert( "The variable miVar is of type: " + valtype( miVar ) )
    miVar := date()
    alert( "The variable miVar is of type: " + valtype( miVar ) )
return
```

This feature doesn't exist in C, since C is a **strongly typed** language: when declaring variables, you must specify the type that they will have during the program's execution. This is so that at compile time C can reserve a specific amount of memory.

So how does Harbour make it possible for a variable to contain any kind of data?
As we already mentioned, in C there are **aggregate (complex) data types**: **structures** and **unions**. With these two, Harbour works its "magic."

All Harbour variables are a **structure** with basically two members. The first indicates the type, `HB_TYPE`, which is internally defined as an unsigned integer:

```
typedef HB_U32 HB_TYPE;
```

And the second is a **union** where all types that exist in Harbour are represented. (Remember what a C `union` is.)
The structure is as follows:

```
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;
```

As we said, in C **all members of a union occupy the same memory area**. That's where the trick lies...

So we can say that a variable in PRG is equivalent in C to a structure called `ITEM`... the `ITEM` type is the representation of Harbour variables in C. These can be created from PRG or from C code, as we will see later.

We won't go deeper for now.

The main types that Harbour variables can have are:

- **Character and Memo (C and M):** to handle character strings.
- **Numeric (N):** to handle any kind of number, signed or unsigned, integer or with decimals.
- **Logical (L):** to handle Boolean types (true/false).
- **Date (D):** to handle dates.
- **Array (A):** to handle in-memory tables. Array members can be of any type (remember that C arrays can only be of one specific type).
- **Block (B):** to handle **codeBlocks**, which contain executable code that can be evaluated at any time.
- **Object (O):** to handle objects. Internally, Harbour treats them as an array.

Those are the main ones inherited from Clipper. But Harbour has more, such as **TimeStamp (T)**, **Hash (H)**, **Pointer (P)** and **Symbol (S)**. (And there are even more, but these are enough for now.)

Here you have the definition of all the data types that Harbour uses internally — remember, this is the first member of the `ITEM` structure:

```
#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 )
```

The FIELD

At the same level as variables, Harbour also has the **FIELD**, with which we can handle DBF fields (or rather, **RDD** fields).

I'm sure this looks familiar:

```
FIELD->CodPostal
```

This is how we tell Harbour that the variable is special and that it must be treated as a **field**. The other way is to use the **alias** or **work area** instead of `FIELD`.

Symbols

As my friend Cristóbal would say… *there's a lot to unpack here!* And indeed, there is — because from here on we'll talk about something as essential in Harbour as the **Symbol Table** and the **Dynamic Symbol Table**.

But let's start at the beginning: the **definition of a symbol**.
A **symbol** is the way the Harbour Virtual Machine handles variables (`ITEM`), fields (`FIELD`), and functions — yes, **especially functions**.

Cristóbal, we're piling up terms… now the **Harbour Virtual Machine (HVM)** appears! But let's not get ahead of ourselves — we'll explain that right after this.

We've already said that bare C values are not compatible at PRG level. To be handled from Harbour, we have to convert them to `ITEM` s through the **ITEM API** or the **Extended System** functions. And functions are not handled in the same way as in C. Harbour needs to know more things about a function, such as the **name** (string), its **scope**, and of course the **memory address** where it resides.

For that, there is a C structure that represents **symbols**:

```
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;
```

I'm showing it so you can see it, but don't be alarmed — you'll very likely never have to handle it directly. What I want you to understand is that it's a structure that contains several members. The most important

ones are the **name** (string), the **scope**, and the **pointer** to the function.

Knowing this, we can say that the **Symbol Table** is a memory area where Harbour stores symbols — that is, a **list of symbols**. To be able to use a function, it must be in the Symbol Table. Therefore, we can **search** the symbol table for it and we'll know many things about it (its name, scope, and pointer to the function it represents).

When we reference a function from PRG, Harbour creates an entry in the **Dynamic Symbol Table**.

On the other hand, at compile time Harbour **loads its internal functions** with a C program called `initsymb.c`. I'm putting it here so you can see what a symbol structure looks like — but only for that. Don't be afraid, because as I said, it's very rare that we'd need to create a symbol manually.

Alright, here it is without further mystery:

```
#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 HAS 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 );
}
```

As you can see, the Harbour Symbol Table is an **array** that contains symbols that are the standard Harbour functions we use in our programs. And therefore — and this is what matters — we can **search** the array by name and thus find the position, or access it directly by position.

An example of a symbol is this:

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

The name as a string, the scope (in this case, public), and the pointer to the function.

Did you know that at PRG level we have these symbols from the table at our disposal? And that there's a **class** to handle them?
You can do things as cool as this:

```
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
```

Or we could use that idea **without** the `Symbol` class, directly:

```
__dynsN2Sym( "Alert" ):exec( "Hello world" )
```

With this idea, we could pass to a function a **string** with the name of the function we want to execute in the target function, and thus avoid using a codeblock, for example.
This has **many uses** for us. For example, using **data-driven** techniques in our programs, avoiding many `IF` conditions, etc. As someone would say, *"The only limit is your mind."*

Now it's time to learn something about the **Harbour Virtual Machine (HVM)** — but first, let's introduce another concept.

What is pCode?

We've already seen that there is only one language machines understand — can you guess which one? Yes, that's right: **machine code**.
Machine code is tightly tied to the machine that will execute it. That means we would have to **recompile** our code for each different kind of machine. To get around this, **interpreted languages** like **Java** or **Harbour** were created.

It's true that Harbour does not generate machine code, but instead gives us the option to generate a **portable binary code** that can be executed on any machine without having to recompile it. We don't usually use it, but it exists. It's the `.hrb` format. If someone has one Windows OS and another Linux OS, you can try it:

```
/* demo.prg */

procedure main
    cls
    ? "Hello world..."
    inkey( 0 )
return
```

Using `hbmk2`:

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

This generates a file **`demo.hrb`**.
To run it on any OS **without recompiling**, just do:

```
hbmk2 demo.hrb
```

That works **anywhere** there is a Harbour Virtual Machine. It works like Java, but in our language. It's very powerful and ready to use in **Mod Harbour** (on Apache), letting our applications work on the web **transparently**.

Having said that...

Do you want to see pCode? It's very easy. Compile the example above like this:

```
harbour demo -gc2
```

Now look in the directory — you'll see there is a file called `demo.c` with the following content:

```c
/*
 * 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 );
}
```

In this listing:

- The **blue** part is the pCode,
- The **red** part is the execution of the pCode, and
- The initial **green** part is the symbol definitions and their insertion into the virtual symbol table.

What is the Harbour Virtual Machine?

It's already been explained with what we've seen, but we could say that it's Harbour's way of making the machine where our program runs **understand the pCode** — in other words, it's a **translator** from pCode to machine code.

Note that most of our programs are executables that **contain the pCode and the virtual machine**.

Now we have to close the circle of this theoretical topic.

What is the stack?

In general terms, a **stack** is an ordered list where data is stored and retrieved. It uses the **LIFO** method — *Last In, First Out*. This applies to Harbour's stack as well. The element at the very top of the list is called the **top**.

There are three operations applicable to the stack:

- **Create** → creates the stack.
- **Push** → adds a new element to the stack, which becomes the top.

- **Pop** → removes the element at the top, decreasing the number of elements by one.

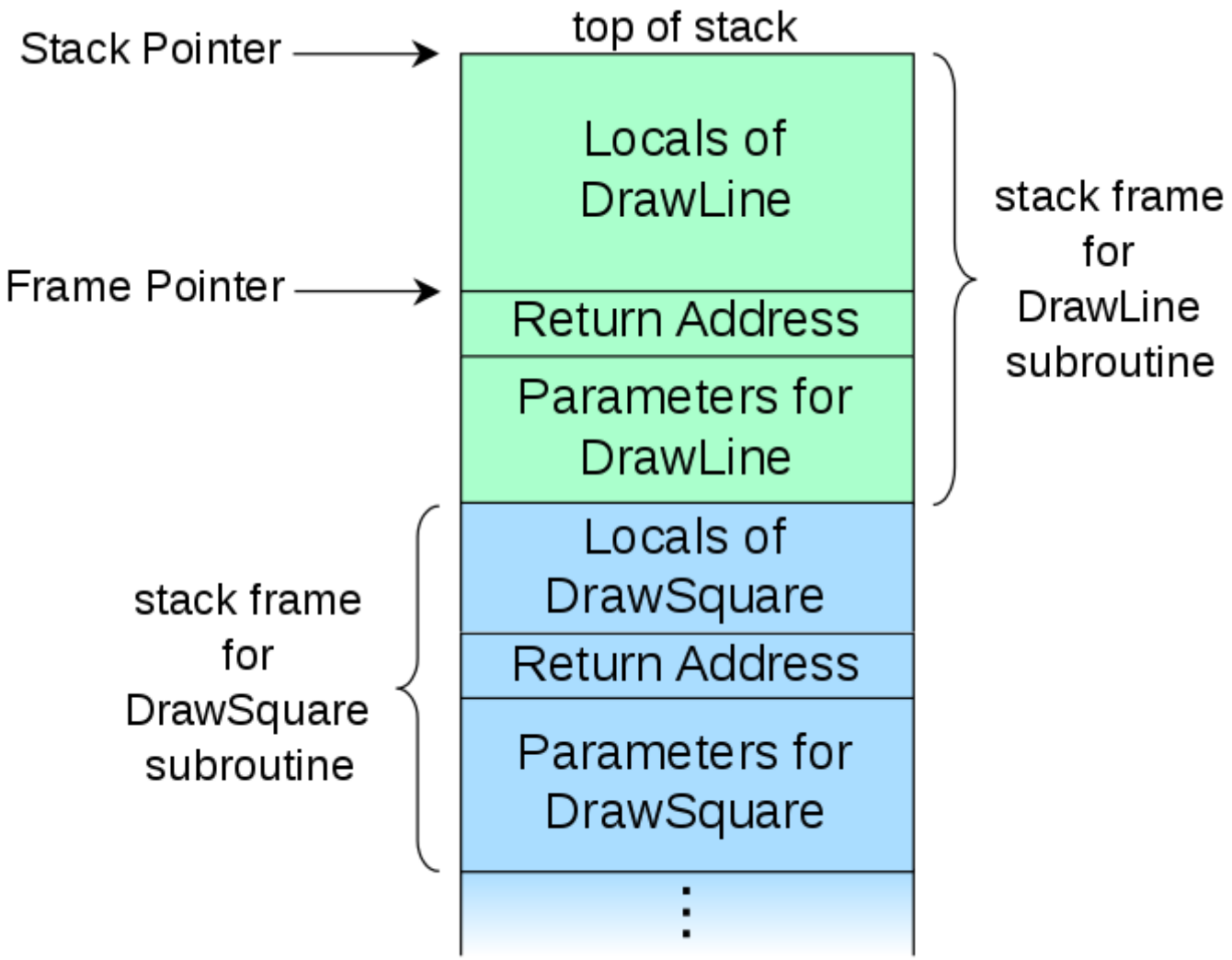
A real-world example of a stack is a stack of plates, one on top of the other.

For this course, the stack is very important since the **Extended System** and the **ITEM API** are based on it. In the next topic we will look at this more deeply.

We can then say that the stack is the **instrument that the HVM uses** to execute functions and the values it has to process. The HVM creates a stack into which it puts values and functions — the stack grows. When the functions are executed, the stack shrinks.

When the stack grows so much that it exceeds its size, the stack **overflows** and the program throws a nice "**Stack Overflow**." Fortunately, Harbour has so greatly increased the default stack size that it's very difficult to see this error — but old Clipper users…

This image is from Wikipedia and I think it graphically illustrates a stack very well:



It's important to note the **return address area**. There is an internal Harbour API function to access that ITEM: `hb_stackReturnItem()`. Keep this in mind, since it will be very useful to use it as a working variable.

Another important point is that variables whose **lifetime spans the entire execution** of the program, such as **static** or **public** variables, are stored in a **permanent** memory area.

In summary

Harbour is an **interpreted** language that uses a **virtual machine** to translate **pCode** into **machine code**, which is the language understood by the machine where the program runs.
The virtual machine creates a **stack** into which it pushes the **symbols** of the functions to be executed and the `ITEM`s that will be processed by those functions — whether parameters or variables.

In the stack, it's the **symbol of a function** that is used, not the function itself. A **symbol** is a structure that contains a member of type string with the function's name, another for its scope, and another for the pointer to the function. These symbols are stored in a memory area called the **Symbol Table**, which acts primarily as a **list of available functions**.

13. Creating C Functions to Be Used from Harbour PRG Programs

As we have already mentioned, functions written in the **C language** that are intended to be used from **Harbour PRG** programs must follow certain rules.
Let's go over them.

All functions that execute within Harbour **do not return or receive anything directly**, and, moreover, they are of **PASCAL calling convention**.
As stated earlier, Harbour uses the **stack** to handle both **parameters** and **return values**.

Here is the prototype skeleton of a C function:

```
void MIFUNCION( void )
{
    ...
    // Function body
    ...
}
```

But this probably doesn't look familiar to you... maybe this does:

```
HB_FUNC( MIFUNCION )
{
    ...
    // Function body
    ...
}
```

This:

```
HB_FUNC( MIFUNCION )
```

instead of:

```
void MIFUNCION( void )
```

It's much easier to use the first one, isn't it?

Notice that the name is in uppercase, and practically speaking, we are not limited to 10 characters as in the legendary **Clipper**.

This means we can now create **self-descriptive function names**.

13.1 Handling Data Between Harbour and C

Remember that **C** only works with **C data types**.
To process Harbour data (which are stored as `ITEM`s), we must extract their **internal values** expressed as C types.

Once we have them, we can use the functions provided by the **C standard library**.
However, as a best practice, we should use **Harbour's internal C API functions** whenever possible.

We'll see this in the examples later, but here's a simple comparison:

Harbour API Function	C Standard Equivalent
`hb_xstrcat()`	`strcat()`
`hb_xstrcpy()`	`strcpy()`

The internal Harbour versions are usually **more powerful**, adapted to Harbour's requirements — for instance, **multithreading** compatibility.
There are internal Harbour functions for almost everything, and you can safely use them.

13.2 Returning Values to Harbour

Once processing is complete, we must **return the results** — converting from C types back into Harbour **ITEMs** so they can be handled from the PRG level.

For this, Harbour provides the **Extended System and the ITEM API**, which is the main subject of this book and will be explained in detail later.

13.3 Summary

To recap:

- Use `HB_FUNC()` to declare C functions callable from Harbour.
- Always work through the **ITEM API** when passing or returning values between C and Harbour.
- Prefer Harbour API equivalents (`hb_x*()` , `hb_item*()` , etc.) over the C standard library for thread safety and integration.
- The Extended System and ITEM API will be your main tools for creating seamless C extensions.

That's all for now...

14. How to Compile C Code in Our PRG Programs

We are now finally reaching the part we all enjoy the most — **programming**.
But before that, we need to understand **how to compile those C functions**.

Technically, we could invoke the C compiler directly.
However, each compiler has its own options and syntax, which can be tedious to learn.
So, how can we avoid that?
How can we simplify the process of building **EXE**, **LIB**, or even just **OBJ** files?

If your answer was "**Use a MAKE tool**", you are absolutely right!

And that's where Harbour's amazing build tool comes in: `hbmk2`.

14.1 What Is `hbmk2`

Syntax:

```
hbmk2 [options] [<command files>] <sources>
[.prg|.c|.obj|.o|.rc|.res|.def|.po|.pot|.hbl|@.clp|.d|.ch]>
```

Description:

`hbmk2` is an **integrated and portable build automation tool**,
allowing you to create different types of binary executables
(executables, dynamic libraries, static libraries, portable Harbour binaries)
from multiple source file types (C, C++, Objective-C, Harbour, gettext translations, Windows resources).

- **Integrated** means a single `.hbmk2` project file can control nearly every aspect of the build process.
- **Portable** means the same `.hbmk2` project file works across all supported operating systems and compilers.

It also simplifies most build processes through small, straightforward **project option files**.

`hbmk2` supports C/C++/Objective-C projects, even those unrelated to Harbour.

To achieve this, it automatically detects:

- The Harbour installation,
- The C compiler,
- Other required tools,

then configures and invokes them as needed.

You can even **extend supported source types** via plug-ins.

In addition to building executables, `hbmk2` can also:

- Directly run Harbour source or precompiled code,
- Provide an **interactive command interpreter**.

14.2 Example: Generic Batch Compilation Setup

Below is a **generic batch script proposal** you can adapt to your environment.

We'll assume the C function examples are stored in a file named `cursode.c`,
and that each `.PRG` file will have its own name.
Eventually, you could compile all of them into a single **library (LIB)**,
but for now, we'll just compile the `.C` file each time as a source.

Don't worry about build times — `hbmk2` is smart enough to perform **incremental builds** (it recompiles only if changes are detected).

14.3 Directory Structure

```
curso_c
 ├── ejemplos
 │    ├── <your_prg_files.prg>
 └── fuentes_c
      └── cursode.c
```

Your batch file will be called:

```
do_<compiler>.bat
```

Example: `do_mingw64.bat`

14.4 Generic Batch Template

```
@set comp=MyCCompiler
@set DIR_HBBIN=MyHarbourBinDir
@set DIR_CCBIN=MyCBinDir
@rem The next two lines usually do not need to be changed
@set PATH=%DIR_HBBIN%;%DIR_CCBIN%;%PATH%
@hbmk2 -comp=%comp% %1 fuentes_c\cursode.c
```

14.5 Variable Description

- **MyCCompiler** →
  - 32-bit: `mingw`, `msvc`, `clang`, `bcc`, `watcom`, `icc`, `pocc`, `xcc`
  - 64-bit: `bcc64`, `mingw64`, `msvc64`, `msvcia64`, `iccia64`, `pocc64`

  For **MSVC** (both 32 and 64-bit), you must also call a script that initializes the environment.
- **MyHarbourBinDir** → Path to the **BIN** directory of your Harbour installation.
- **MyCBinDir** → Path to the **BIN** directory of your C compiler.

> ⚠ The Harbour **LIB** files must match the bitness (32/64) of the C compiler you are using.

14.6 Practical Examples

◇ MinGW 64-bit

```
@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
```

◇ MSVC 32-bit

```
@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
```

◇ MSVC 64-bit

```
@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
```

14.7 Notes

These batch files are deliberately simple — we'll refine them later by adding:

- Error handling,
- Build logging,
- Automatic library creation.

But for now, they're perfect for quick experiments and small projects.

☑ **Summary:**

- `hbmk2` is Harbour's universal build tool.
- Use simple batch scripts to manage builds.
- Match your Harbour installation and C compiler versions (32-bit or 64-bit).
- Incremental compilation saves time and keeps your workflow clean.

15. The Harbour Extended System

A very important capability programming languages can have is the integration with the omnipotent **C language**. For example, Java programmers rely on the **Java Native Interface (JNI)**. This gives those languages virtually unlimited expansion capacity.

Harbour is no different and—better yet—instead of just one mechanism we have **two** ways to integrate C into our applications:

- **The Extended System**
- **The ITEM API**

You can even mix both technologies. In this chapter we will focus on the first one.

Old **Clipper** has always implemented the Extended System and, starting from the 5.xx versions, it also introduced the **ITEM API**. As you know, Harbour is a modern, improved version of old Clipper; it inherits everything and improves it across the board. Therefore, the Extended System is included in that inheritance. You can think of it as the *traditional* way to integrate C functions into our `.prg` programs.

As we said earlier, C functions intended to be called from PRG must meet certain conditions: they **do not return anything directly** and **do not accept parameters**. In C that shape would be:

```
void MIFUNCION( void )
{
    // Function body
}
```

Harbour simplifies the declaration a bit, letting us write:

```
HB_FUNC( MIFUNCION )
{
    // Function body
}
```

In other words, `HB_FUNC( MIFUNCION )` is an easier way to write `void MIFUNCION( void )`.

So if, as we see, these functions neither return nor receive parameters, **how can we pass parameters from a PRG and receive values back?** That's where the **HVM**, **symbols**, and the **stack** come in.

These functions use the **stack** to receive parameters and the **return area** (*stackReturn*) to place the result.

Every time a PRG calls a function, the HVM **pushes** all parameters onto the stack. Conceptually:

```
PUSH  "Function_or_Method_Symbol"
PUSH  "NIL for functions or the object symbol for methods"
PUSH  "param1"
PUSH  "param2"
...
PUSH  "paramN"
DO( NumParams )  // executes and pops
```

`DO` can be one of these three:

- `hb_vmDo( nPar )` for functions, procedures, and methods
- `hb_vmProc( nPar )` for procedures and functions
- `hb_vmSend( nPar )` for object methods

(There are others, but these three are the most important. In the next chapter—where we'll see how to execute PRG functions from C—we'll revisit this in more detail.)

Extended System functions **retrieve parameters** that were pushed on the stack by their **positional index**, and they **place the return value** in the return area of the stack. That's the magic.

From this we can say there are **two** main kinds of Extended System helper functions:

1. Those that **retrieve** parameters from the stack by position.
2. Those that **write** the **return value** into the stack's return area.

The first group are the **`hb_par…()`** functions (e.g., `hb_parc()`, `hb_parni()`, etc.).
The second group are the **`hb_ret…()`** functions (e.g., `hb_retc()`, `hb_retni()`, etc.).
In addition, for parameters **passed by reference**, there are the **`hb_stor…()`** functions (e.g., `hb_storc()`, `hb_storni()`).

Parameter Passing

These are the main functions for **retrieving** passed parameters:

```
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 );
```

There are more. **Boldface** in the original listed the ones present in Clipper 5.xx; the rest are Harbour extensions. We'll first explain the classic ones and the most useful new ones.

Instead of going one by one, here is a **legend** of the **return types** (what each function returns), i.e., C types:

- `const char *` → a constant string
- `HB_SIZE` → a very large integer (like `long long`)
- `HB_ISIZ` → a very large integer (like `long long`)
- `long` → a long integer
- `double` → a double-precision real
- `HB_MAXINT` → the maximum available wide integer type (`long` / `long long`)
- `void *` → a typeless pointer

And for the **parameters**:

- `int iParam` → positional index of the parameter (1-based from PRG's perspective)
- `DefValue` → default value if the PRG did not pass that parameter

It's **very important** to understand this mapping.

Example (PRG calling a C function):

```
PROCEDURE Main()
   LOCAL nAge, lIsAdult
   nAge := 57
   lIsAdult := miMayorEdad( nAge )
   Alert( "He is " + IIF( lIsAdult, "an adult", "a minor" ) )
RETURN
```

In the C function `miMayorEdad` we retrieve the parameter like this:

```
int hb_parni( 1 );  // Retrieves parameter 1 as an integer
```

Returning Values from C

The "return" functions are basically **symmetric** to the parameter ones: they can accept the same kinds of data that the `hb_par…()` group returns—except in this case we **pass** those values **to** the `hb_ret…()` functions. Main ones:

```
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 );
```

(Again, the ones in the original **bold** were supported by Clipper; the rest are Harbour extensions.)

Accepted parameter types (what you pass **into** `hb_ret…()` so Harbour receives them as a PRG value):

- `const char *` → constant string
- `char *` → mutable string
- `HB_SIZE` / `HB_ISIZ` → very large integer
- `int` → integer
- `long` → long integer
- `double` → double-precision real

- `void` → no parameters
- `HB_MAXINT` → max-width integer

Naming hints used in both families (`hb_par…()` , `hb_ret…()`):

- `c` → strings (`hb_parc` , `hb_retc`)
- `n` → numbers; followed by `i` (integer) or `l` (long): `hb_parni` , `hb_parnl`
- `l` → logicals (booleans)
- `d` → dates
- `a` → arrays

Also note: `hb_retc_buffer()` and `hb_retclen_buffer()` work like their non- `_buffer` counterparts **but free** previously allocated memory.

Upgrading the Build Script (Batch)

Let's enhance the earlier batch file to stop on errors and run the program on success (MinGW 64-bit example, `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
@if %errorlevel% neq 0 goto bld_error
@cls
%1
goto fin_exec
:bld_error
@echo ----------------------------------
@echo    There were compilation errors
@echo ----------------------------------
pause
:fin_exec
```

Example 1 — Calling C Functions from PRG

`ej001.prg`

```
//----------------------------------------------------------------------------
// Exercise: using C functions
// ej001.prg
//----------------------------------------------------------------------------

PROCEDURE Main()

   LOCAL getlist := {}
   LOCAL nYear := Year( Date() )
   LOCAL nCube := 0
   LOCAL cNum, i, nDay := 0, cDay
   LOCAL cString := Space( 60 )
   LOCAL cChar := " "
   LOCAL n1 := 0, n2 := 0, nS, nR, nP, nD
   LOCAL nGrade := 0, cStatus
   LOCAL aNum := { 10, 20, 30 }

   CLS

   separa( .f. )

   @ 02, 01 SAY "C Course for Harbour"

   separa( .f. )

   @ 05, 01 SAY "Enter a year to check leap year..................:" GET nYear PICTURE "@K
9999"
   @ 06, 01 SAY "Enter a number to compute its cube...............:" GET nCube PICTURE "@K
9999"
   @ 07, 01 SAY "Enter a string...................................:"
   @ 08, 10 GET cString PICTURE "@K"
   @ 09, 01 SAY "Enter a character to count in the string.........:" GET cChar PICTURE "@K"
   @ 10, 01 SAY "Enter a number for multiplication table..........:" GET n1 PICTURE "@K 99"
   @ 11, 01 SAY "Enter a number to run calculations...............:" GET n2 PICTURE "@K 99"
   @ 12, 01 SAY "Enter a grade....................................:" GET nGrade PICTURE "@K
99"
   @ 13, 01 SAY "Give a number 1-7 to compute day of week.........:" GET nDay PICTURE "@k 9"

   separa( .f. )

   READ

   CLS

   separa( .f. )

   // Test calls to C functions
   ? "The year " + hb_ntoc( nYear )
   IF mi_isLeap( nYear )
      ?? " is leap"
   ELSE
      ?? " is not leap"
```

```
ENDIF

separa( .f. )

cNum := hb_ntoc( nCube ) // number as string

// Does not return a value; uses the same variable (by reference) to store the cube
mi_cubo( @nCube )

? "The cube of " + cNum + " is " + hb_ntoc( nCube )

separa( .f. )

cString := AllTrim( cString )
n := scanChar( cString, cChar )
IF n > 0
   ? [In the string: "] + cString + ["]
   ? "The first occurrence of character '" + cChar + "' is at position " + HB_NToS( n )
   n := charCount( cString, cChar )
   ? "and it repeats " + HB_NToS( n ) + " times"
ELSE
   ? "There is no '" + cChar + "' in the string: " + cString
ENDIF

separa( .f. )

calcula( n1, n2, @nS, @nR, @nP, @nD ) // Note: last 4 are by reference with '@'

// Note: parameters to calcula() are passed by reference
? "Sum.......: " + HB_NToS( n1 ) + " + " + HB_NToS( n2 ) + " = " + HB_NToS( nS )
? "Subtract..: " + HB_NToS( n1 ) + " - " + HB_NToS( n2 ) + " = " + HB_NToS( nR )
? "Product...: " + HB_NToS( n1 ) + " X " + HB_NToS( n2 ) + " = " + HB_NToS( nP )
? "Division..: " + HB_NToS( n1 ) + " / " + HB_NToS( n2 ) + " = " + HB_NToS( nD )

separa( .f. )

esActo( nGrade, @cStatus )

? "You are " + cStatus

separa( .t. )

// We'll add 3 elements to the array to verify that sumaArray() only sums numerics
AAdd( aNum, "Hello" )
AAdd( aNum, 25.75 )
AAdd( aNum, 1000 )

cambiaValor( aNum )
FOR i := 1 TO Len( aNum )
   ? "Element", i, aNum[ i ]
NEXT

? "The sum of array elements is:", sumaArray( aNum )
```

```
   separa( .f. )

   ? "Multiplication table of " + 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. )

   cDay := diaSemana( nDay )

   IF !Empty( cDay )
      ? "The day is: " + cDay
   ELSE
      ? "Input error: only numbers from 1 to 7..."
   ENDIF

   separa( .t. )

RETURN

// Draw a separator on screen
STATIC PROCEDURE separa( lPause )
   ? Replicate( "-", 70 )
   IF ValType( lPause ) == 'L' .AND. lPause
      ? "Press any key to continue..."
      Inkey( 100 )
   ENDIF
RETURN
```

```
/*
 * Returns whether a year is leap or not
 */
HB_FUNC( MI_ISLEAP )
{
   HB_UINT uiYear = hb_parni( 1 );   // Get integer parameter from PRG
   HB_BOOL lRet = HB_FALSE;

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

   hb_retl( lRet );                  // Return a logical value via Harbour's stack
}

/*
 * Cube of a number passed by reference
 */
HB_FUNC( MI_CUBO )
{
   HB_MAXINT iCubo = hb_parnint( 1 );
   iCubo = iCubo * iCubo * iCubo;
   hb_stornint( iCubo, 1 );
}

/*
 * Counts the occurrences of a character within a string
 */
HB_FUNC( CHARCOUNT )
{
   const char * szCadena = hb_parc( 1 );       // param 1 as string
   HB_SIZE uiLen = hb_parclen( 1 );            // length of param 1
   HB_UINT uiContador = 0;

   if( uiLen > 0 )
   {
      const char * cCaracter = hb_parc( 2 );   // param 2 as string
      HB_SIZE i;

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

   hb_retnint( uiContador );
}
```

```
/*
 * Finds the first occurrence of a character within a string
 */
HB_FUNC( SCANCHAR )
{
   const char * szCadena = hb_parc( 1 );
   HB_SIZE uiLen = hb_parclen( 1 );
   HB_SIZE uiPos = 0;

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

      for( i = 0; i < uiLen; i++ )
      {
         if( szCadena[ i ] == cCaracter[ 0 ] )
         {
            uiPos = i + 1;   // +1 because C arrays are 0-based
            break;
         }
      }
   }

   hb_retnint( uiPos );
}
```

```c
/*
 * Returns whether a year is leap or not
 */
HB_FUNC( MI_ISLEAP )
{
   HB_UINT uiYear = hb_parni( 1 );
   HB_BOOL lRet = HB_FALSE;

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

   hb_retl( lRet );
}

/*
 * Cube of a number passed by reference
 */
HB_FUNC( MI_CUBO )
{
   HB_MAXINT iCubo = hb_parnint( 1 );
   iCubo = iCubo * iCubo * iCubo;
   hb_stornint( iCubo, 1 );
}

/*
 * Checks whether a grade is passing and stores a literal with the result
 */
HB_FUNC( ESACTO )
{
   unsigned int uiNota = hb_parnidef( 1, 3 ); // Default to 3 (minimum) if not passed
   if( uiNota > 4 )
      hb_storc( "APPROVED", 2 );
   else
      hb_storc( "FAILED", 2 );
}

/*
 * Day of week
 * Uses hb_xstrcpy() and hb_retc_buffer()
 */
HB_FUNC( DIASEMANA )
{
   unsigned int uiDia = hb_parni( 1 ); // Receives the number

   if( uiDia >= 1 && uiDia <= 7 )
   {
      // Holds the day literal; good practice is to init pointers to NULL
      char * szDia = NULL;

      // hb_xstrcpy allocates memory if first parameter is NULL
```

```
      switch( uiDia )
      {
         case 1: szDia = hb_xstrcpy( NULL, "Monday",    NULL ); break;
         case 2: szDia = hb_xstrcpy( NULL, "Tuesday",   NULL ); break;
         case 3: szDia = hb_xstrcpy( NULL, "Wednesday", NULL ); break;
         case 4: szDia = hb_xstrcpy( NULL, "Thursday",  NULL ); break;
         case 5: szDia = hb_xstrcpy( NULL, "Friday",    NULL ); break;
         case 6: szDia = hb_xstrcpy( NULL, "Saturday",  NULL ); break;
         case 7: szDia = hb_xstrcpy( NULL, "Sunday",    NULL ); break;
      }

      hb_retc_buffer( szDia ); // Return and free memory
   }
   else
   {
      hb_ret(); // Return NIL
   }
}

/*
 * Reverses a string
 */
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();
   }
}
```

Passing Parameters by Reference

```
/*
 * By-reference parameters
 * Receives 2 integers and returns sum, difference, product, and division into refs
 */
HB_FUNC( CALCULA )
{
   int i1 = hb_parni( 1 );
   int i2 = hb_parni( 2 );

   // By reference:
   hb_storni( i1 + i2, 3 );
   hb_storni( i1 - i2, 4 );
   hb_storni( i1 * i2, 5 );
   hb_stornd( ( double ) i1 / i2, 6 ); // Take care with the cast
}
```

Working with Arrays

```
/*
 * Increases the second element of an array by 100
 */
HB_FUNC( CAMBIAVALOR )
{
   int iVal = hb_parvni( 1, 2 );
   iVal = iVal + 100;
   hb_storvni( iVal, 1, 2 );
}
```

```
/*
 * Sums numeric elements of a passed array
 */
HB_FUNC( SUMAARRAY )
{
   double uiTotal = 0.0;

   if( hb_extIsArray( 1 ) )
   {
      HB_SIZE i;
      HB_SIZE nLen = hb_parinfa( 1, 0 ); // number of elements

      for( i = 0; i < nLen; i++ )
      {
         switch( hb_parinfa( 1, i + 1 ) )
         {
            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;
         }
      }
   }

   hb_retnd( uiTotal );
}
```

```
/*
 * Multiplication table for the passed integer
 */
HB_FUNC( TABLA )
{
   if( hb_parinfo( 1 ) == HB_IT_INTEGER )
   {
      HB_MAXINT iMul = hb_parnint( 1 );
      HB_UINT n = 10, i;

      hb_reta( n );  // create array in stackReturn

      for( i = 1; i <= n; i++ )
         hb_storvnint( iMul * i, -1, i ); // -1 accesses the return area
   }
   else
   {
      hb_ret();  // NIL
   }
}
```

Working with C Structures

```
//-------------------------------------------------------------------------
// Exercise: structure via the Extended System
// In PRG: an array with the same elements as the C structure
// ej002.prg
//-------------------------------------------------------------------------
```

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

HB_FUNC( DAMEPERSONA )
{
   HB_UINT uiPersona = hb_parni( 1 );
   TPersona * persona = hb_xgrab( sizeof( TPersona ) );

   if( uiPersona == 1 )
   {
      hb_xstrcpy( persona->szNIF,    "53320105T", NULL );
      hb_xstrcpy( persona->szNombre, "Viruete", ", ", "Paco", NULL );
      persona->uiCodigo = 26212;
      persona->nSalrio  = 3500.97f;
      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  = 2750.75f;
      hb_xstrcpy( persona->szFecha,  "19660213", NULL );
      persona->bSoltero = HB_TRUE;
   }

   hb_reta( 6 );
   hb_storvc(  persona->szNIF,    -1, 1 );
   hb_storvc(  persona->szNombre,-1, 2 );
   hb_storvni( persona->uiCodigo,-1, 3 );
   hb_storvnd( persona->nSalrio, -1, 4 );
   hb_storvds( persona->szFecha, -1, 5 );
   hb_storvl(  persona->bSoltero,-1, 6 );
   hb_xfree( persona );
}
```

```
//----------------------------------------------------------------------------
// Exercise: structure via the Extended System — object variant
// ej003.prg
//----------------------------------------------------------------------------

#include "hbclass.ch"

PROCEDURE Main
   LOCAL persona := TPersona():new( 1 )
   CLS
   persona:muestra()
   ?
   ? "Press ENTER..."
   Inkey( 100 )
   CLS
   persona := TPersona():new( 2 )
   persona:muestra()
   ?
   ? "Press ENTER..."
   Inkey( 100 )
   CLS
   TPersona():new( 1 ):muestra()
   ?
   ? "Press ENTER..."
   Inkey( 100 )
RETURN

CREATE CLASS TPersona
   DATA cNIF
   DATA cNombre
   DATA nCodigo
   DATA nSalrio
   DATA dFecha
   DATA lSoltero
   METHOD new( nPersona ) CONSTRUCTOR
   METHOD muestra()
END CLASS

METHOD new( nPersona ) CLASS TPersona
   dameObjPersona( Self, nPersona )
RETURN Self

PROCEDURE muestra() CLASS TPersona
   ? "Object person data:"
   ? ::cNIF
   ? ::cNombre
   ? ::nCodigo
   ? ::nSalrio
   ? ::dFecha
   ? ::lSoltero
RETURN
```

```
#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 ) );

   if( uiPersona == 1 )
   {
      hb_xstrcpy( persona->szNIF,    "53320105T", NULL );
      hb_xstrcpy( persona->szNombre, "Viruete", ", ", "Paco", NULL );
      persona->uiCodigo = 26212;
      persona->nSalrio  = 3500.97f;
      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  = 2750.75f;
      hb_xstrcpy( persona->szFecha,  "19660213", NULL );
      persona->bSoltero = HB_TRUE;
   }

   hb_itemReturn( obj );
   hb_storvc(  persona->szNIF,    -1, 1 );
   hb_storvc(  persona->szNombre,-1, 2 );
   hb_storvni( persona->uiCodigo,-1, 3 );
   hb_storvnd( persona->nSalrio, -1, 4 );
   hb_storvds( persona->szFecha, -1, 5 );
   hb_storvl(  persona->bSoltero,-1, 6 );
   hb_xfree( persona );
}
```

Interfacing with C Library Functions (and Your Own 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;
}
```

```
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 ) ) );
}
```

```
//----------------------------------------------------------------------------
// Using interfaces to C standard library functions
// ej005.prg
//----------------------------------------------------------------------------

PROCEDURE Main

   LOCAL getlist := {}
   LOCAL cPhrase := 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 "Enter a phrase:"   GET cPhrase PICTURE "@k!"
   @ 03, 01 SAY "Separator.....:"   GET cSep    PICTURE "@k!"
   @ 04, 01 SAY "Enter number..:"   GET nNum    PICTURE "@k"
   READ

   cPhrase := AllTrim( cPhrase )
   cSep    := AllTrim( cSep )

   IF Empty( cSep )
      cSep := " " // default: space
   ENDIF

   CLS
   ? "Phrase....:", cPhrase
   ? "Reversed..:", reverse( cPhrase )
   ?

   /* first token */
   cToken := strtok( cPhrase, cSep )

   /* loop while there are tokens */
   WHILE !Empty( cToken )
      ? ++n, "token ->", cToken
      cToken := strtok( NIL, cSep )
   END

   ?
   ? "Trigonometric functions for", nNum, "in radians:"
   ? "Cosine......:", c_Cos( nNum )
   ? "Sine........:", c_Sin( nNum )
   ? "Tangent.....:", c_Tan( nNum )

   ?
   ? "Press ENTER to continue..."
```

```
   Inkey( 100 )

RETURN
```

Summary of the Extended System

There are basically **three** families that convert Harbour variables to/from primitive C types:

- **Parameter retrieval:** `hb_par…()`
- **Return value emission:** `hb_ret…()`
- **By-reference storage:** `hb_stor…()`

Skeleton of a C function called from PRG:

```
HB_FUNC( NOMBRE_FUNCION )
{
   // Receive parameters
   … = hb_par...();
   … = hb_par...();
   …

   // Process them (now C types)
   …

   // Return a single result
   hb_ret...( value );
}
```

With the **Extended System** you do **not** work directly with Harbour variables (ITEMs); it's an intermediary layer. In the next chapter we'll study the **ITEM API** (Clipper's newer proposal, much improved by Harbour), which makes things more flexible and robust and lets you interact directly with Harbour ITEMs —including arrays, codeblocks, and hashes.

16. The Item API. Extending the Extended System

As I've said several times before, this was the new approach Clipper introduced with Clipper 5.00 and later. Harbour has expanded and improved it.

What is an API in Harbour?

An **Application Programming Interface (API)** is a set of function definitions, rules, and protocols that let us write C code to be used from our Harbour PRG programs. Function signatures specify the return type and the parameter types. The **Extended System** is an API, and **the ITEM API** is also an API.

What is an ITEM in Harbour?

We saw this earlier, but let's revisit it now. An **ITEM** in Harbour is a C structure that represents a Harbour variable. It has two members:

- `HB_TYPE type`: holds the ITEM type as an integer (all variable types are represented).
- A C `union` whose members are structures for each Harbour type. Those structures each have a `value` member that contains the variable's value.

```
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;
```

The `HB_TYPE type` member may be any of the following:

```
/* Element types and type-checking macros */
#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 )
```

The `type` member determines which union structure contains the appropriate `value`. That union is what allows Harbour variables to hold different data types simply by assigning a given value.

PRG illustration

```
local xVar

ValType( xVar )         // -> U
xVar := 1000
ValType( xVar )         // -> N
xVar := "Hello world"
ValType( xVar )         // -> C
xVar := Date()
ValType( xVar )         // -> D
xVar := Array( 5 )
ValType( xVar )         // -> A
xVar := {||}
ValType( xVar )         // -> B
```

This is not possible in C, where variables must have a fixed type at declaration, and that type cannot change within the same scope.

The ITEM API functions are declared in `hbapiitm.h`. Broadly, they include:

Parameter count

```
HB_USHORT hb_itemPCount( void );  // Same as hb_pcount()
```

Fetch parameters passed from PRG (as ITEMs)

```
PHB_ITEM hb_param( int iParam, long lMask );
PHB_ITEM hb_itemParam( HB_USHORT uiParam );
PHB_ITEM hb_itemParamPtr( HB_USHORT uiParam, long lMask ); // Same as hb_param()
```

Return an ITEM from C to PRG

```
PHB_ITEM hb_itemReturn( PHB_ITEM pItem );
PHB_ITEM hb_itemReturnForward( PHB_ITEM pItem );
void     hb_itemReturnRelease( PHB_ITEM pItem );
```

Get size of an ITEM (Array, Hash, or String)

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

Get current ITEM type

```
HB_TYPE     hb_itemType( PHB_ITEM pItem );        // numeric type code
const char* hb_itemTypeStr( PHB_ITEM pItem );     // string type, like ValType()
```

Create and free an ITEM

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

Basic Array handling

```
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 );
```

Extract C values from an ITEM

```
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 );
```

Assign C values to an ITEM (create if NULL)

```
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 );
```

Assign NIL

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

There are more, but these are enough for now.

Passing parameters from PRG to C

Unlike the Extended System—where there's one function per C type—in the ITEM API parameters arrive as **ITEMs**. In practice:

- `PHB_ITEM hb_param( int iParam, long lMask )` and `PHB_ITEM hb_itemParamPtr( HB_USHORT uiParam, long lMask )` are synonyms. Both fetch a parameter as an ITEM and filter it with a type mask (`HB_TYPE`). If the parameter doesn't match the mask, they return `NULL`.
- `PHB_ITEM hb_itemParam( HB_USHORT uiParam )` works similarly but **makes a copy** of the ITEM, so you **must** release it before leaving your function.

Returning values from C to PRG

We always return an **ITEM** (no per-type return functions as in the Extended System). The main ones are:

- `PHB_ITEM hb_itemReturn( PHB_ITEM pItem )` — return the ITEM to PRG.
- `void hb_itemReturnRelease( PHB_ITEM pItem )` — return and free the ITEM.
- `PHB_ITEM hb_itemReturnForward( PHB_ITEM pItem )` — return an ITEM owned by Harbour (e.g., taken from the stack). The GC will free it.

Basic handling of Harbour arrays in C

As in PRG, there are essentially three basics:

- Create an array: `PHB_ITEM hb_itemArrayNew( HB_SIZE nLen )`
- Put an element: `PHB_ITEM hb_itemArrayPut( PHB_ITEM pArray, HB_SIZE nIndex, PHB_ITEM pItem )`
- Get an element: `PHB_ITEM hb_itemArrayGet( PHB_ITEM pArray, HB_SIZE nIndex )`

There are many more functions for arrays which we'll see later. Those variants avoid converting to/from ITEM by directly working with C values.

Creating and releasing an ITEM in C

If you need to create an ITEM inside a C function, use `PHB_ITEM hb_itemNew( PHB_ITEM pNull )`. If `pNull` is another ITEM, it clones it; if `NULL`, it creates an undefined-item (`ValType() == "U"`) until you assign a value. In all cases, release the ITEM before returning with `HB_BOOL hb_itemRelease( PHB_ITEM pItem )`.

Getting the current C value from an ITEM

Use the `hb_itemGet...()` functions listed above. The source ITEM must exist (e.g., passed from PRG or obtained from another API call).

Assigning a C value to an ITEM

Use `hb_itemPut...()` functions. If the first parameter is `NULL`, a new ITEM is created with that value.

Other useful ITEM API functions

- `HB_BOOL hb_itemEqual( PHB_ITEM pItem1, PHB_ITEM pItem2 )` — compare two ITEMs.
- `void hb_itemCopy( PHB_ITEM pDest, PHB_ITEM pSource )` — copy an ITEM preserving content.
- `void hb_itemMove( PHB_ITEM pDest, PHB_ITEM pSource )` — move value without increasing ref-count; source becomes `U`.
- `void hb_itemClear( PHB_ITEM pItem )` — clear to `U`.
- `PHB_ITEM hb_itemClone( PHB_ITEM pItem )` — clone an ITEM.
- `char * hb_itemStr( PHB_ITEM pNumber, PHB_ITEM pWidth, PHB_ITEM pDec )` — format numeric ITEM as string.
- `char * hb_itemString( PHB_ITEM pItem, HB_SIZE *nLen, HB_BOOL *bFreeReq )` — convert any ITEM to string; returns length and whether you must free.
- `PHB_ITEM hb_itemValToStr( PHB_ITEM pItem )` — like previous but returns a string ITEM.
- `void hb_itemSwap( PHB_ITEM pItem1, PHB_ITEM pItem2 )` — swap contents.

There are more, but this set suffices for now.

Naming notes

- `PHB_ITEM` — pointer to an ITEM (i.e., a PRG variable as seen from C).
- `PHB_ITEM pArray` — a Harbour array as an ITEM.
- `HB_SIZE nIndex` — array index (1-based for Harbour arrays).
- `HB_SIZE nLen` — length of an array, hash, or string.

Suffixes in `hb_itemGet...()` / `hb_itemPut...()` mirror the Extended System:

- `N` (numbers): `NI` (int), `NL` (long), `ND` (double), `NInt` (max int)
- `D` (date): `D` (Y,M,D), `DS` (YYYYMMDD), `DL` (Julian long)
- `TS` (datetime as string)
- `Ptr` (pointer)
- `C` (string)
- `L` (logical)

> **IMPORTANT:** Any ITEM created directly via `hb_itemNew()`, or received via `hb_itemParam()` or `hb_itemPut...()` with `NULL` as first arg (e.g., `hb_itemPut...( NULL, … )`), **must** be freed with `hb_itemRelease()` to avoid memory leaks.

Example: Variables in PRG are ITEMs in C

PRG

```
/*
 * ej000.prg
 * Using ITEM
 */

procedure main

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

    SET DATE FORMAT TO "dd/mm/yyyy"
    cls

    ? "Assignment of variables from C:"
    ? "--------------------------------"
    ?
    asignaVarC( @cVar, @nVar, @dVar, @lVar, aVar )

    ? "String...:", cVar
    ? "Number...:", nVar
    ? "Date.....:", dVar
    ? "Logical..:", lVar
    ?
    ? "Array....:"
    for i := 1 to 4
        ? i, aVar[ i ]
    next

    ?
    ? "Press ENTER to continue..."
    Inkey( 100 )

return
```

C

```
/*
 * Assign values to parameters passed by reference
 */

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 );

    // String
    if( cVar )
    {
        hb_itemPutC( cVar, "String assigned in C" );
    }

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

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

    // Logical
    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 );
    }
}
```

Further examples with the ITEM API

(Leap year test, cube, char count/scan, arithmetic by-ref, pass/fail literals, array element update, array sum, multiplication table, weekday name, string reverse, custom `cstrtran`, libc `strtok`, and math

wrappers.)

All code remains identical in PRG; only the C side uses the ITEM API variants.

Array helpers (ITEM API)

The basic array primitives are:

- `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 )`

These work only with `ITEM`s. If you want to set an element from a plain C value (like `char *` or `double`), you would normally create an `ITEM`, put the value into it with `hb_itemPut…()`, and then use `hb_itemArrayPut()`.

To avoid that boilerplate, Harbour adds **array helpers** that *get* and *set* C types directly, creating or reading the transient `ITEM` under the hood.

Generic maintenance helpers

- `HB_BOOL hb_arrayNew( PHB_ITEM pItem, HB_SIZE nLen )` — turns `pItem` into an array of `nLen` elements.
- `HB_BOOL hb_arrayAdd( PHB_ITEM pArray, PHB_ITEM pItemValue )` — append (like `AAdd()`).
- `HB_BOOL hb_arrayIns( PHB_ITEM pArray, HB_SIZE nIndex )` — insert empty item at `nIndex` (like `AIns()`).
- `HB_BOOL hb_arrayDel( PHB_ITEM pArray, HB_SIZE nIndex )` — delete at `nIndex` (like `ADel()`).
- `HB_BOOL hb_arraySize( PHB_ITEM pArray, HB_SIZE nLen )` — resize (like `ASize()`).
- `HB_BOOL hb_arrayLast( PHB_ITEM pArray, PHB_ITEM pResult )` — copy last item to `pResult`.
- `HB_BOOL hb_arrayGet( PHB_ITEM pArray, HB_SIZE nIndex, PHB_ITEM pItem )` — copy element into `pItem`.
- `HB_BOOL hb_arraySet( PHB_ITEM pArray, HB_SIZE nIndex, PHB_ITEM pItem )` — set element from `pItem`.
- `HB_TYPE hb_arrayGetType( PHB_ITEM pArray, HB_SIZE nIndex )` — get element type.
- `HB_SIZE hb_arrayLen( PHB_ITEM pArray )` — `Len( aArray )`.

Get element as C types

- `char *hb_arrayGetC( PHB_ITEM pArray, HB_SIZE nIndex )`
- `HB_SIZE hb_arrayGetCLen( PHB_ITEM pArray, HB_SIZE nIndex )`
- `HB_BOOL hb_arrayGetL( PHB_ITEM pArray, HB_SIZE nIndex )`

- `int hb_arrayGetNI( PHB_ITEM pArray, HB_SIZE nIndex )`
- `long hb_arrayGetNL( PHB_ITEM pArray, HB_SIZE nIndex )`
- `HB_MAXINT hb_arrayGetNInt( PHB_ITEM pArray, HB_SIZE nIndex )`
- `double hb_arrayGetND( PHB_ITEM pArray, HB_SIZE nIndex )`
- `char *hb_arrayGetDS( PHB_ITEM pArray, HB_SIZE nIndex, char * szDate )`
- `long hb_arrayGetDL( PHB_ITEM pArray, HB_SIZE nIndex )`

Set element from C types

- `HB_BOOL hb_arraySetDS( PHB_ITEM pArray, HB_SIZE nIndex, const char *szDate )`
- `HB_BOOL hb_arraySetDL( PHB_ITEM pArray, HB_SIZE nIndex, long lDate )`
- `HB_BOOL hb_arraySetL( PHB_ITEM pArray, HB_SIZE nIndex, HB_BOOL fValue )`
- `HB_BOOL hb_arraySetNI( PHB_ITEM pArray, HB_SIZE nIndex, int iNumber )`
- `HB_BOOL hb_arraySetNL( PHB_ITEM pArray, HB_SIZE nIndex, long lNumber )`
- `HB_BOOL hb_arraySetNInt( PHB_ITEM pArray, HB_SIZE nIndex, HB_MAXINT nNumber )`
- `HB_BOOL hb_arraySetND( PHB_ITEM pArray, HB_SIZE nIndex, double dNumber )`
- `HB_BOOL hb_arraySetC( PHB_ITEM pArray, HB_SIZE nIndex, const char *szText )`

Example 1 — process an array by reference and create a new array

PRG

```
/* Process an array and create another from C (ITEM API helpers) */

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    // Try changing this number

    SET DATE FORMAT TO "dd-mm-yyyy"
    CLS

    // Adds nSuma to each element (array passed by reference)
    tramita( aArray, nSuma )

    ?
    ? "Value added:", nSuma
    ? "-------------------------------------------------------------------------"
    ? "Previous values", "New values"
    ?

    FOR i := 1 TO nLen
        ? aArray[ i ] - nSuma, aArray[ i ]
    NEXT

    ?
    ? "Press a key to continue..."
    Inkey( 100 )
    CLS

    ?
    ? "-------------------------------------------------------------------------"
    ? "Array created in C:"
    ?

    aDesdeC := creaArray()
    nLen := Len( aDesdeC )

    FOR i := 1 TO nLen
        ? "Element " + HB_NToS( i ), aDesdeC[ i ]
    NEXT

    Inkey( 100 )

RETURN
```

C

```
/* Adds a numeric delta to every numeric element of an array (by reference) */
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 );
         }
      }
   }
}

/* Create and fill an array with mixed types */
HB_FUNC( CREAARRAY )
{
   PHB_ITEM aArray = hb_itemNew( NULL );  /* empty item */
   PHB_ITEM xItem  = hb_itemNew( NULL );

   /* Turn empty item into a 10-element array.
      (Could also do: aArray = hb_itemArrayNew( 10 ) ) */
   hb_arrayNew( aArray, 10 );

   hb_arraySetDS( aArray, 1, "20211231" );
   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" );

   /* Put an uninitialized item (nil) at pos 9 */
   hb_arraySet( aArray, 9, xItem );

   /* Now assign a string to xItem and store at pos 10 */
   hb_itemPutC( xItem, "Asignacion del item con valor tipo cadena" );
   hb_arraySet( aArray, 10, xItem );

   hb_itemRelease( xItem );
   hb_itemReturnRelease( aArray );
}
```

Gather/Scatter using arrays

A practical buffer pattern: read a DBF record into an array (`aGather()`), modify it, and write it back (`aScatter()`). Below, the PRG driver and the two C functions.

PRG

```
PROCEDURE Main()

    LOCAL aBuf

    CLS
    USE test NEW

    test->( DBGoTo( 10 ) )
    aBuf := test->( aGather() )   // read current record

    ? "------------------------------------------------------------------------"
    ? "Type: " + ValType( aBuf ), "  /  Elements: " + HB_NToS( Len( aBuf ) )
    ? "------------------------------------------------------------------------"
    ? test->( RecNo() ), aBuf[ 2 ], aBuf[ 9 ]
    ? "------------------------------------------------------------------------"
    ? "Change AGE with a random integer"
    aBuf[ 9 ] := HB_RandomInt( 0, 99 )
    ? "Buffer value:", aBuf[ 9 ]
    test->( aScatter( aBuf ) )    // write back
    ? "------------------------------------------------------------------------"
    ? "DBF AGE is now", test->( FieldGet( 9 ) )
    ? "------------------------------------------------------------------------"
    Inkey( 100 )

RETURN
```

C

```
/* Read current record into an array buffer */
HB_FUNC( AGATHER )
{
   PHB_ITEM aBuffer = hb_itemNew( NULL );
   PHB_ITEM pValue  = hb_itemNew( NULL );
   AREAP pArea = ( AREAP ) hb_rddGetCurrentWorkAreaPointer();
   HB_USHORT uiFields, i;

   SELF_FIELDCOUNT( pArea, &uiFields );
   hb_arrayNew( aBuffer, uiFields );

   for( i = 1; i <= uiFields; i++ )
   {
      SELF_GETVALUE( pArea, i, pValue );
      hb_arraySet( aBuffer, i, pValue );
   }

   hb_itemRelease( pValue );
   hb_itemReturnRelease( aBuffer );
}

/* Write an array buffer to the current record */
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 );
      }

      hb_itemRelease( pValue );
   }
}
```

> **Note on dates:** like most languages, DBF stores dates as a long integer (Julian.)

Hash tables (associative arrays)

Harbour hashes are (key,value) containers optimized for key lookups. In C with the ITEM API, the hash object and both key/value are all `ITEM`s. That means we still create/assign `ITEM`s and let the API

manage references.

Core hash functions

- `PHB_ITEM hb_hashNew( PHB_ITEM pItem )`
- `HB_SIZE hb_hashLen( PHB_ITEM pHash )`
- `HB_BOOL hb_hashDel( PHB_ITEM pHash, PHB_ITEM pKey )`
- `HB_BOOL hb_hashAdd( PHB_ITEM pHash, PHB_ITEM pKey, PHB_ITEM pValue )`
- `HB_BOOL hb_hashAddNew( PHB_ITEM pHash, PHB_ITEM pKey, PHB_ITEM pValue )`
- `HB_BOOL hb_hashRemove( PHB_ITEM pHash, PHB_ITEM pItem )`
- `HB_BOOL hb_hashClear( PHB_ITEM pHash )`
- `void hb_hashSort( PHB_ITEM pHash )`
- `PHB_ITEM hb_hashClone( PHB_ITEM pHash )`
- `PHB_ITEM hb_hashCloneTo( PHB_ITEM pDest, PHB_ITEM pHash )`
- `void hb_hashJoin( PHB_ITEM pDest, PHB_ITEM pSource, int iType )`
- `HB_BOOL hb_hashScan( PHB_ITEM pHash, PHB_ITEM pKey, HB_SIZE * pnPos )`
- `HB_BOOL hb_hashScanSoft( PHB_ITEM pHash, PHB_ITEM pKey, HB_SIZE * pnPos )`
- `void hb_hashPreallocate( PHB_ITEM pHash, HB_SIZE nNewSize )`
- `PHB_ITEM hb_hashGetKeys( PHB_ITEM pHash )`
- `PHB_ITEM hb_hashGetValues( PHB_ITEM pHash )`
- `void hb_hashSetDefault( PHB_ITEM pHash, PHB_ITEM pValue )`
- `PHB_ITEM hb_hashGetDefault( PHB_ITEM pHash )`
- `void hb_hashSetFlags( PHB_ITEM pHash, int iFlags )`
- `void hb_hashClearFlags( PHB_ITEM pHash, int iFlags )`
- `int hb_hashGetFlags( PHB_ITEM pHash )`
- `void * hb_hashId( PHB_ITEM pHash )`
- `HB_COUNTER hb_hashRefs( PHB_ITEM pHash )`

Example — build a hash in C, iterate in PRG

PRG

```
PROCEDURE Main()

    LOCAL hTabla, xClave, xPar
    LOCAL nLen, i

    CLS

    ?
    ? "------------------------------------------------------------------------"
    ? "Hash created in C:"
    ? "------------------------------------------------------------------------"
    ?

    hTabla := creaHash()
    nLen := Len( hTabla )

    hb_HCaseMatch( hTabla, .f. )  // case-insensitive

    ?
    ? "Hash has " + HB_NToS( nLen ) + " pairs (key,value)"
    ?
    ? "------------------------------------------------------------------------"
    ? "Plain FOR/NEXT:"
    ? "------------------------------------------------------------------------"
    FOR i := 1 TO nLen
        xClave := hb_HKeyAt( hTabla, i )
        ? "Key: ", xClave, " -> Value: ", hb_HGet( hTabla, xClave )
    NEXT

    ? "------------------------------------------------------------------------"
    ? "FOR EACH (recommended):"
    ? "------------------------------------------------------------------------"
    FOR EACH xPar IN hTabla
        ? "Key: ", xPar:__enumKey(), " -> Value: ", xPar:__enumValue()
    NEXT

    Inkey( 100 )

RETURN
```

C

```
/* Build a small sample hash with multiple types */
HB_FUNC( CREAHASH )
{
   PHB_ITEM hHash  = hb_hashNew( NULL );
   PHB_ITEM pKey   = hb_itemNew( NULL );
   PHB_ITEM pValue = hb_itemNew( NULL );

   /* Dates */
   hb_itemPutC(  pKey, "fechaDS" );
   hb_itemPutDS( pValue, "20211231" );
   hb_hashAdd( hHash, pKey, pValue );

   hb_itemPutC(  pKey, "fechaDL" );
   hb_itemPutDL( pValue, hb_dateEncStr( "20211231" ) );
   hb_hashAdd( hHash, pKey, pValue );

   /* Logical */
   hb_itemPutC(  pKey, "logico" );
   hb_itemPutL(  pValue, HB_TRUE );
   hb_hashAdd( hHash, pKey, pValue );

   /* Integers */
   hb_itemPutC(  pKey, "entero" );
   hb_itemPutNI( pValue, 35789 );
   hb_hashAdd( hHash, pKey, pValue );

   hb_itemPutC(  pKey, "largo" );
   hb_itemPutNL( pValue, 15578952 );
   hb_hashAdd( hHash, pKey, pValue );

   /* Double */
   hb_itemPutC(  pKey, "real" );
   hb_itemPutND( pValue, 34954155455.41 );
   hb_hashAdd( hHash, pKey, pValue );

   hb_itemRelease( pKey );
   hb_itemRelease( pValue );

   hb_itemReturnRelease( hHash );
}
```

Gather/Scatter using hashes

A more convenient buffer uses field *names* as keys instead of numeric positions.

PRG

```
PROCEDURE Main()

    LOCAL hBuf

    CLS
    USE test NEW

    test->( DBGoTo( 10 ) )
    hBuf := test->( hGather() )  // read current record

    ? "------------------------------------------------------------------------"
    ? "Type: " + ValType( hBuf ), "  /  Elements: " + HB_NToS( Len( hBuf ) )
    ? "------------------------------------------------------------------------"
    ? test->( RecNo() ), hBuf[ "Last" ], hBuf[ "age" ]
    ? "------------------------------------------------------------------------"
    ? "Change AGE with a random integer"
    hBuf[ "age" ] := HB_RandomInt( 0, 99 )
    ? "Buffer value:", hBuf[ "age" ]
    test->( hScatter( hBuf ) )     // write back
    ? "------------------------------------------------------------------------"
    ? "DBF AGE is now", test->( FieldGet( FieldPos( "age" ) ) )
    ? "------------------------------------------------------------------------"
    Inkey( 100 )

RETURN
```

C

```
/* Read current record into a hash buffer (key = field name) */
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;

   hb_hashClearFlags( hBuffer, HB_HASH_BINARY );
   hb_hashSetFlags( hBuffer, HB_HASH_IGNORECASE | HB_HASH_RESORT );

   SELF_FIELDCOUNT( pArea, &uiFields );
   hb_hashPreallocate( hBuffer, uiFields );

   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 );
   }

   hb_itemRelease( pKey );
   hb_itemRelease( pValue );
   hb_xfree( szFldName );

   hb_itemReturnRelease( hBuffer );
}

/* Write a hash buffer back to the current record */
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 ) );
      }
   }
}
```

`TWABuffer` — a tiny work-area buffer class

Finally, a class that encapsulates a DBF work-area buffer as a hash, minimizing loose variables and wiring.

Public methods

- `new()` – initialize (binds to the current work area)
- `load()` – read current record into buffer
- `save()` – write buffer back to current record
- `blank()` – clear buffer with blank values (via phantom record 0)
- `getBuffer()` – return the underlying hash
- `set( cFieldName, xValue )` – assign a value
- `get( cFieldName )` – fetch a value
- `getAlias()` – current alias (like `Alias()`)
- `getArea()` – current work area number (like `Select()`)
- `getLen()` – number of pairs in the buffer

Definition in Harbour; implementation in C

```
// thbuffer.prg (class definition)

#include "HBClass.ch"

CREATE CLASS TWABuffer

    DATA iData PROTECTED  // internal pointer to C-side state

    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()

END CLASS
```

C implementation (selected parts, fully translated & annotated)

```
#pragma BEGINDUMP

#define _IDATA 1

#include "hbapi.h"
#include "hbapiitm.h"
#include "hbapirdd.h"
#include "hbstack.h"

typedef struct
{
    AREAP    pArea;
    PHB_ITEM hBuffer;
    HB_USHORT uiFields;
} DATA, *PDATA;

/* Constructor */
HB_FUNC_STATIC( TWABUFFER_NEW )
{
    PHB_ITEM pSelf = hb_stackSelfItem();
    AREAP pArea = ( AREAP ) hb_rddGetCurrentWorkAreaPointer();

    if( pArea )
    {
        PDATA pData = ( PDATA ) hb_xgrab( sizeof( DATA ) );
        PHB_ITEM hBuffer = hb_hashNew( NULL );
        PHB_ITEM pKey = hb_itemNew( NULL );
        HB_USHORT uiFields, i;
        char *szFldName = ( char * ) hb_xgrab( pArea->uiMaxFieldNameLength + 1 );

        SELF_FIELDCOUNT( pArea, &uiFields );
        hb_hashPreallocate( hBuffer, uiFields );

        hb_hashClearFlags( hBuffer, HB_HASH_BINARY );
        hb_hashSetFlags( hBuffer, HB_HASH_IGNORECASE | HB_HASH_RESORT );

        for( i = 1; i <= uiFields; i++ )
        {
            SELF_FIELDNAME( pArea, i, szFldName );
            hb_itemPutC( pKey, szFldName );
            hb_hashAdd( hBuffer, pKey, NULL );
        }

        pData->pArea    = pArea;
        pData->hBuffer  = hBuffer;
        pData->uiFields = uiFields;

        hb_arraySetPtr( pSelf, _IDATA, ( void * ) pData );
        hb_itemRelease( pKey );
        hb_xfree( szFldName );
    }
```

```
    hb_itemReturnForward( pSelf );
}

/* Load current record into the buffer */
HB_FUNC_STATIC( TWABUFFER_LOAD )
{
    PHB_ITEM pSelf = hb_stackSelfItem();
    PDATA pData = ( PDATA ) hb_arrayGetPtr( pSelf, _IDATA );
    PHB_ITEM pValue = hb_itemNew( NULL );
    HB_USHORT i;

    for( i = 1; i <= pData->uiFields; i++ )
    {
        SELF_GETVALUE( pData->pArea, i, pValue );
        hb_hashAdd( pData->hBuffer, hb_hashGetKeyAt( pData->hBuffer, i ), pValue );
    }

    hb_itemRelease( pValue );
    hb_itemReturn( pData->hBuffer );
}

/* Save buffer to current record */
HB_FUNC_STATIC( TWABUFFER_SAVE )
{
    PHB_ITEM pSelf = hb_stackSelfItem();
    PDATA pData = ( PDATA ) hb_arrayGetPtr( pSelf, _IDATA );
    HB_USHORT i;

    for( i = 1; i <= pData->uiFields; i++ )
    {
        SELF_PUTVALUE( pData->pArea, i, hb_hashGetValueAt( pData->hBuffer, i ) );
    }
}

/* Blank buffer using phantom record 0 */
HB_FUNC_STATIC( TWABUFFER_BLANK )
{
    PHB_ITEM pSelf = hb_stackSelfItem();
    PDATA pData = ( PDATA ) hb_arrayGetPtr( pSelf, _IDATA );
    PHB_ITEM pRecNo = hb_itemNew( NULL );
    PHB_ITEM pRec0  = hb_itemPutNL( NULL, 0 );

    SELF_RECID(  pData->pArea, pRecNo );
    SELF_GOTOID( pData->pArea, pRec0 );
    HB_FUNC_EXEC( TWABUFFER_LOAD );
    SELF_GOTOID( pData->pArea, pRecNo );

    hb_itemRelease( pRec0 );
    hb_itemRelease( pRecNo );

    hb_itemReturn( pData->hBuffer );
}
```

```
/* Get underlying hash */
HB_FUNC_STATIC( TWABUFFER_GETBUFFER )
{
    PHB_ITEM pSelf = hb_stackSelfItem();
    hb_itemReturn( ( ( PDATA ) hb_arrayGetPtr( pSelf, _IDATA ) )->hBuffer );
}

/* Get value by field name */
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();
        PDATA pData = ( PDATA ) hb_arrayGetPtr( pSelf, _IDATA );
        pRes = hb_hashGetItemPtr( pData->hBuffer, cKey, HB_HASH_AUTOADD_ACCESS );
    }

    hb_itemReturn( pRes );
}

/* Set value by field name */
HB_FUNC_STATIC( TWABUFFER_SET )
{
    PHB_ITEM pKey = hb_param( 1, HB_IT_STRING );
    PHB_ITEM pValue = hb_param( 2, HB_IT_ANY );
    HB_BOOL fRes = HB_FALSE;

    if( pKey && pValue )
    {
        PHB_ITEM pSelf = hb_stackSelfItem();
        PDATA pData = ( PDATA ) hb_arrayGetPtr( pSelf, _IDATA );

        fRes = hb_hashScan( pData->hBuffer, pKey, NULL );
        if( fRes )
            hb_hashAdd( pData->hBuffer, pKey, pValue );
    }

    hb_itemPutL( hb_stackReturnItem(), fRes );
}

/* Count pairs */
HB_FUNC_STATIC( TWABUFFER_GETLEN )
{
    PHB_ITEM pSelf = hb_stackSelfItem();
    PDATA pData = ( PDATA ) hb_arrayGetPtr( pSelf, _IDATA );
    hb_itemPutNI( hb_stackReturnItem(), pData->uiFields );
}

/* Alias of bound work area */
HB_FUNC_STATIC( TWABUFFER_GETALIAS )
```

```
{
    PHB_ITEM pSelf = hb_stackSelfItem();
    PDATA pData = ( PDATA ) hb_arrayGetPtr( pSelf, _IDATA );
    char *szAlias = ( char * ) hb_xgrab( HB_RDD_MAX_ALIAS_LEN + 1 );

    SELF_ALIAS( pData->pArea, szAlias );
    hb_itemPutC( hb_stackReturnItem(), szAlias );
    hb_xfree( szAlias );
}

/* Work area number (Select()) */
HB_FUNC_STATIC( TWABUFFER_GETAREA )
{
    PHB_ITEM pSelf = hb_stackSelfItem();
    PDATA pData = ( PDATA ) hb_arrayGetPtr( pSelf, _IDATA );
    hb_itemPutNI( hb_stackReturnItem(), pData->pArea->uiArea );
}

/* Destructor */
HB_FUNC_STATIC( TWABUFFER_FREE )
{
    PHB_ITEM pSelf = hb_stackSelfItem();
    PDATA pData = ( PDATA ) hb_arrayGetPtr( pSelf, _IDATA );

    if( pData )
    {
        hb_itemRelease( pData->hBuffer );
        hb_xfree( pData );
        hb_arraySetPtr( pSelf, _IDATA, NULL );
    }
}

#pragma ENDDUMP
```

Takeaways

- ITEM API lets you work **directly with Harbour variables** (`ITEM` s) in C.
- Array/hash helpers remove a lot of glue code when you need C primitives in/out.
- Gather/Scatter patterns are straightforward and can be wrapped in classes for cleaner PRG code.

> Next chapter: the new ITEM API extensions for arrays and hashes in more advanced scenarios.

17. Execute Harbour functions, methods, and codeBlocks from C

Sometimes it's necessary in C to execute a Harbour function (those commonly used in PRGs) or one written in PRG by third parties or by ourselves — and even retrieve its return value in order to process it from C. The good news is that this is very easy to do in Harbour.
To understand what we're going to cover now, it would be useful to review **Topic 12**, since we're going to use the **stack**, the **symbol table**, and we'll invoke the **virtual machine (VM)** ourselves.

This is the list of functions.

For execution

```
void hb_vmDo( HB_USHORT uiParams );        // Invoke the virtual machine
void hb_vmProc( HB_USHORT uiParams );      // Execute a function or procedure
void hb_vmFunction( HB_USHORT uiParams );  // Execute a function
void hb_vmSend( HB_USHORT uiParams );      // Send a message to an object
```

The four functions above receive as a parameter an unsigned integer that represents the number of parameters that have been pushed onto the stack using the stack-push functions (listed below).

For handling codeBlocks

```
PHB_ITEM hb_vmEvalBlock( PHB_ITEM pBlockItem );                  // Evaluate the passed
codeBlock without arguments
PHB_ITEM hb_vmEvalBlockV( PHB_ITEM pBlockItem, HB_ULONG ulArgCount, ... ); // Evaluate the
passed codeBlock with a variable number of arguments
PHB_ITEM hb_vmEvalBlockOrMacro( PHB_ITEM pItem );                // Evaluate the codeBlock or
macro pointed to by the given item
void     hb_vmDestroyBlockOrMacro( PHB_ITEM pItem );             // Destroy the codeBlock or
macro in the given item
```

To push parameters onto the stack

```
void hb_vmPush( PHB_ITEM pItem );                           // Push a generic item
void hb_vmPushNil( void );                                  // Push NIL (nothing)
void hb_vmPushNumber( double dNumber, int iDec );           // Push a number; decides
integer/long/double
void hb_vmPushInteger( int iNumber );                       // Push an integer
void hb_vmPushLong( long lNumber );                         // Push a long integer
void hb_vmPushDouble( double dNumber, int iDec );           // Push a real number
void hb_vmPushSize( HB_ISIZ nNumber );                      // Push an HB_SIZE
void hb_vmPushNumInt( HB_MAXINT nNumber );                  // Push a number; decides integer
or HB_MAXINT
void hb_vmPushLogical( HB_BOOL bValue );                    // Push a logical value
void hb_vmPushString( const char * szText, HB_SIZE length );// Push a string
void hb_vmPushStringPcode( const char * szText, HB_SIZE length ); // Push a pcode string
void hb_vmPushDate( long lDate );                           // Push a date as a long integer
(Julian)
void hb_vmPushTimeStamp( long lJulian, long lMilliSec );    // Push two longs as a TimeStamp
void hb_vmPushSymbol( PHB_SYMB pSym );                      // Push a pointer to a function
symbol
void hb_vmPushDynSym( PHB_DYNS pDynSym );                   // Push a pointer to a
function/method symbol
void hb_vmPushEvalSym( void );                              // Push the evaluation symbol for
codeBlocks
void hb_vmPushPointer( void * pPointer );                   // Push an HB_IT_POINTER
void hb_vmPushPointerGC( void * pPointer );                 // Push a GC-managed HB_IT_POINTER
void hb_vmPushItemRef( PHB_ITEM pItem );                    // Push an item passed by
reference
```

Examples

The PRG that calls C functions

```
//-----------------------------------------------------------------------------
// Exercise: using C functions. Execute PRG functions from C
// ej012.prg
//-----------------------------------------------------------------------------

procedure main

    local n

    cls

    // Shows a string from C in an Alert; receives no parameters
    ejAlertDesdeC00()

    // Shows a string from C in an Alert; receives a string parameter "C"
    ejAlertDesdeC01( "ALERT from C passing this string" )
    // Warning message when checking that a string was not passed
    ejAlertDesdeC01( Date() )

    // Shows a string from C in an Alert; receives a parameter of any
    // type and converts it to "C" string
    ejAlertDesdeC02( "Hello, I'm a little string" ) // String
    ejAlertDesdeC02( Time() ) // String with time format
    ejAlertDesdeC02( Date() ) // Date
    ejAlertDesdeC02( 1200 ) // Integer
    ejAlertDesdeC02( 390.25 ) // Real number
    ejAlertDesdeC02( 1200 + 390.25 ) // Sum of integer and real
    ejAlertDesdeC02( .t. )  // Logical
    ejAlertDesdeC02() // Without parameter; shows message with a warning

    // Execute a function written by ourselves or by third parties
    n := ejSumaEnC()
    ? "Value returned by ejSumaEnC", n
    n := ejSumaEnCPar( 10, 34.79)
    ? "Value returned by ejSumaEnCPar -> 10 + 34.79 =", n
    n := ejSumaEnCParPro( 10, 34.79)
    ? "Value returned by ejSumaEnCParPro -> ( 10 + 34.79 ) * 100 =", n
    ? "-------------------------------------------------------------------------"
    Inkey( 100 )

return

//-----------------------------------------------------------------------------
// Add two numbers
//-----------------------------------------------------------------------------

function suma( n1, n2 )

    if ValType( n1 ) != 'N'
        n1 := 0
    endif
    if ValType( n2 ) != 'N'
```

```
        n2 := 0
    endif

return n1 + n2

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

And now the C functions

```c
/*
 * Execute PRG functions from C
 */

#include "hbvm.h"    // Required to invoke the virtual machine and use its functions
#include "hbstack.h" // For stack management

/*
 * Call the ALERT function from C.
 * We also assign a string from C
 */
HB_FUNC( EJALERTDESDEC00 )
{
    // Look up the ALERT symbol in the dynamic symbol table.
    // The hb_dynsymFind() function searches for the symbol by name taking
    // uppercase/lowercase into account; if it doesn't find it, it returns NULL,
    // which should be handled. In this case I don’t handle it because it’s a
    // Harbour function that will always exist.
    PHB_DYNS pExecSym = hb_dynsymFind( "ALERT" );

    hb_vmPushDynSym( pExecSym ); // Push the function symbol onto the stack
    // IMPORTANT: The first position on the stack after the function symbol
    // will always be NIL for functions and procedures, or SELF for class methods.
    hb_vmPushNil(); // Push NIL
    // Push a string as a C string. The string length must be passed.
    hb_vmPushString( "Hello, this is an ALERT from C", 30 );
    // This function invokes the Harbour Virtual Machine (VM) to execute
    // the function. You must indicate the number of parameters.
    hb_vmDo( 1 );
}

/*
 * Call the ALERT function from C.
 * We pass a string from PRG and check that it is a string
 */
HB_FUNC( EJALERTDESDEC01 )
{
    PHB_DYNS pExecSym = hb_dynsymFind( "ALERT" );
    PHB_ITEM cItem = hb_param( 1, HB_IT_STRING );

    hb_vmPushDynSym( pExecSym );
    hb_vmPushNil();

    // Ensure the type expected in hb_param() — i.e., a string (HB_IT_STRING)
    if( cItem )
    {
        hb_vmPush( cItem ); // Push the parameter as an ITEM
    }
    else // If an Item of type "C" (string) was not passed
    {
        // Push a string as a C string. The length must be passed.
        hb_vmPushString( "ATTENTION;No string was passed", 35 );
```

```
    }

    // Invoke the VM to execute what is on the stack.
    // Either function may be used:
    //hb_vmDo( 1 );
    hb_vmProc( 1 );
}

/*
 * Call the ALERT function from C.
 * We pass any value and type from PRG and check that something was passed
 */
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 )
    {
        // The hb_itemValToStr() function creates a new Item of character type
        // from an Item of any type passed to it; therefore, it must be released
        // once used.
        PHB_ITEM cItem = hb_itemValToStr( xItem );

        hb_vmPush( cItem );
        hb_itemRelease( cItem );
    }
    else
    {
        hb_vmPushString( "ATTENTION;No value was passed", 37 );
    }
    // Either function may be used:
    //hb_vmDo( 1 );
    hb_vmProc( 1 );
}

/*
 * Call the PRG function SUMA from C.
 * Numbers are provided from C
 */
HB_FUNC( EJSUMAENC )
{
    // hb_dynsymFind() is case sensitive; Harbour uppercases all symbols created
    // from PRG, but it respects case. If the name we pass as a parameter might be
    // in mixed case, hb_dynsymFindName() can be used as it uppercases before searching.
    PHB_DYNS pExecSym = hb_dynsymFind( "SUMA" );

    // When there is no certainty that the symbol exists, check it.
    if( pExecSym )
    {
```

```
        hb_vmPushDynSym( pExecSym );
        hb_vmPushNil();
        hb_vmPushInteger( 1500 );
        hb_vmPushDouble( 250.35, 2 );
        // Either function may be used:
        //hb_vmDo( 2 );
        hb_vmProc( 2 );
    }
}

/*
 * Call the PRG function SUMA from C.
 * Numbers are passed from PRG
 */
HB_FUNC( EJSUMAENCPAR )
{
    PHB_DYNS pExecSym = hb_dynsymFind( "SUMA" );

    // When there is no certainty that the symbol exists, check it.
    if( pExecSym )
    {
        // The two parameters
        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 ); // Either function may be used
        hb_vmProc( 2 );
    }
}

/*
 * Call the PRG function SUMA from C and process the result in C.
 * In this case we multiply by 100 and return the result
 * The two numbers are passed from PRG
 */
HB_FUNC( EJSUMAENCPARPRO )
{
    PHB_DYNS pExecSym = hb_dynsymFind( "SUMA" );

    // When there is no certainty that the symbol exists, check it.
    if( pExecSym )
    {
        double dNum, dRes;
        // The two parameters
        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 ); // The result is placed on the stack in the return item

        // The return item is retrieved with hb_stackReturnItem()
        // Extract the number as C type double to process it
        dNum = hb_itemGetND( hb_stackReturnItem() ); // Get the number as C type
        // Do the processing; in this case, multiply by 100
        dRes = dNum * 100;
        // Place the result onto the stack in the return item
        hb_itemPutND( hb_stackReturnItem(), dRes );
    }
}
```

Up to now, we've only used `void hb_vmDo( HB_USHORT uiParams );` to invoke the VM, but in fact this function can be used for functions, procedures, and method sends. The rest are slightly more optimized for the specific task indicated by their names (as explained at the beginning of the topic).

```
void hb_vmProc( HB_USHORT uiParams );     // Should be used for procedures and functions alike
void hb_vmFunction( HB_USHORT uiParams ); // Guarantees the return stack is clean for the
function's return value
void hb_vmSend( HB_USHORT uiParams );     // Preferred for sending messages to objects
(methods); more optimized than hb_vmDo() for this
```

That said, I recommend using **hb_vmProc()** for both procedures and functions, and **hb_vmSend()** only for methods.

Example of sending messages with hb_vmSend()

The PRG with the example and the class

```
//-----------------------------------------------------------------------------
// Exercise: using methods in C. Execute PRG methods from 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()

   // SET / GET methods
   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( "Here the model would be persisted" )

RETURN lRet

//-----------------------------------------------------------------------------

METHOD leeModelo() CLASS TMiModelo

   LOCAL lRet := .F.

   Alert( "Here the model would be loaded from the DataSet" )

RETURN lRet
```

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

METHOD verDatosModelo() CLASS TMiModelo

   Alert( "INFORMATION;-----------;" + ";" + ;
          "Name.....: " + ::getNombre() + ";" + ;
          "Surname..: " + ::getApellido() + ";" + ;
          "Age......: " + hb_ntos( ::getEdad() ) + ";" + ;
          "Salary...: " + hb_ntos( ::getSueldo() ) )

RETURN self

//----------------------------------------------------------------------------
// SET / GET Methods

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

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

And now the C part

```
/*
 * Example of executing methods from C
 * Takes the data directly from C with C-type variables
 */

HB_FUNC( CAMBIADATOSENC )
{
    PHB_ITEM pObj = hb_param( 1, HB_IT_OBJECT );

    if( pObj )
    {
        PHB_DYNS pMsgSym = hb_dynsymFind( "CAMBIADATOS" );

        hb_vmPushDynSym( pMsgSym ); // Push the method onto the stack
        hb_vmPush( pObj ); // This is super important: push the object first
        hb_vmPushString( "Isabel", 6 );
        hb_vmPushString( "Guerrero", 8 );
        hb_vmPushInteger( 52 );
        hb_vmPushDouble( 2500.75, 2 );
        hb_vmSend( 4 );
    }
}

/*
 * Example of executing methods from C
 * Takes the data passed from PRG as Items
 */

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 ); // Push the method onto the stack
        hb_vmPush( pObj );          // This is super important: push the object first
        hb_vmPush( cNombre );
        hb_vmPush( cApellido );
        hb_vmPush( nEdad );
        hb_vmPush( nSueldo );
        hb_vmSend( 4 );
    }
}
```

Now, codeBlocks...

Just like with executing functions, procedures, and methods, evaluating **codeBlocks** from C is very easy and follows the same approach already described — i.e., use of the **stack** and invoking the **virtual machine** to process them.

Clipper's initial proposal was to use the following functions:

```
HB_BOOL  hb_evalNew( PHB_EVALINFO pEvalInfo, PHB_ITEM pItem ); // Prepare EVALINFO with the
passed codeBlock, ready to receive parameters
HB_BOOL  hb_evalPutParam( PHB_EVALINFO pEvalInfo, PHB_ITEM pItem ); // Add a parameter to the
structure (call once per parameter)
PHB_ITEM hb_evalLaunch( PHB_EVALINFO pEvalInfo ); // Evaluate the codeBlock with the
parameters stored in EVALINFO
HB_BOOL  hb_evalRelease( PHB_EVALINFO pEvalInfo ); // Free memory occupied by copies of items
used as parameters and re-init the structure
// NOTE: If the items passed as parameters were the originals obtained with hb_param()
// (and not copies via hb_itemParam()), this function will cause an undefined error
// when attempting to free those original items.
```

A couple of examples

```
/*
 * Evaluate a codeBlock passed from C.
 * Clipper's legacy way.
 * Without parameters
 */
HB_FUNC( EVALUA00 )
{
    PHB_ITEM pCB = hb_param( 1, HB_IT_BLOCK ); // The passed codeBlock

    if( pCB ) // If a CB was passed
    {
        HB_EVALINFO EvalInfo; // Structure HB_EVALINFO

        // Initialize HB_EVALINFO (pass its address with &)
        if( hb_evalNew( &EvalInfo, pCB ) )
        {
            hb_evalLaunch( &EvalInfo ); // Evaluate the CB
        }

        hb_evalRelease( &EvalInfo ); // Free the structure's internal memory
    }
}

/*
 * Evaluate a codeBlock passed from C.
 * Clipper's legacy way.
 * With parameters
 */
HB_FUNC( EVALUA01 )
{
    PHB_ITEM pCB = hb_param( 1, HB_IT_BLOCK ); // The passed codeBlock

    if( pCB ) // If a CB was passed
    {
        HB_EVALINFO EvalInfo; // Structure HB_EVALINFO

        // Initialize HB_EVALINFO
        if( hb_evalNew( &EvalInfo, pCB ) )
        {
            // NOTE: Use hb_itemParam(), which creates a copy of the passed parameter.
            // Do not pass the original parameter, because hb_evalRelease frees those copies.
            hb_evalPutParam( &EvalInfo, hb_itemParam( 2 ) ); // Add first parameter
            hb_evalPutParam( &EvalInfo, hb_itemParam( 3 ) ); // Add second parameter

            hb_evalLaunch( &EvalInfo ); // Evaluate the CB with the passed parameters
        }

        hb_evalRelease( &EvalInfo ); // Free the structure's internal memory
    }
}
```

Undocumented direct codeBlock evaluation helpers

```
void hb_evalBlock0( PHB_ITEM pCodeBlock );                          // Evaluate a codeBlock
without parameters
void hb_evalBlock1( PHB_ITEM pCodeBlock, PHB_ITEM pParam );         // Evaluate a codeBlock
with one parameter
void hb_evalBlock( PHB_ITEM pCodeBlock, ... );                      // Evaluate a codeBlock
with an arbitrary number of parameters
// NOTE: The last parameter must be NULL.
```

Examples:

```
/*
 * Evaluate a codeBlock passed from C.
 * Helper to evaluate a CB without parameters directly.
 */
HB_FUNC( EVALUA04 )
{
    PHB_ITEM pCB = hb_param( 1, HB_IT_BLOCK ); // The passed codeBlock

    if( pCB )
    {
        hb_evalBlock0( pCB );
    }
}

/*
 * Evaluate a codeBlock passed from C.
 * Helper to evaluate a CB with one parameter directly.
 */
HB_FUNC( EVALUA05 )
{
    PHB_ITEM pCB = hb_param( 1, HB_IT_BLOCK ); // The passed codeBlock

    if( pCB )
    {
        hb_evalBlock1( pCB, hb_param( 2, HB_IT_ANY ) );
    }
}

/*
 * Evaluate a codeBlock passed from C.
 * Helper to evaluate a CB with parameters directly.
 */
HB_FUNC( EVALUA06 )
{
    PHB_ITEM pCB = hb_param( 1, HB_IT_BLOCK ); // The passed codeBlock

    if( pCB )
    {
        hb_evalBlock( pCB, hb_param( 2, HB_IT_ANY ), hb_param( 3, HB_IT_ANY ), NULL );
    }
}
```

As a curiosity, it's worth noting that Harbour internally treats codeBlocks as **objects**. The object would be the variable name, and the evaluation method would be `bCB:eval( p0, p1, …, pn )` with the parameters passed. This is precisely Harbour's newer approach to internal evaluation of codeBlocks — it's done *as if it were a method call*.

Let's see it in the examples:

```
/*
 * Evaluate a codeBlock passed from C.
 * Harbour's newer proposal.
 * Works exactly like object methods.
 * This is the same example as the EVALUA00 function above.
 */
HB_FUNC( EVALUA02 )
{
    PHB_ITEM pCB = hb_param( 1, HB_IT_BLOCK ); // The passed codeBlock

    if( pCB )
    {
        hb_vmPushEvalSym(); // Push the eval() symbol onto the stack
        hb_vmPush( pCB );   // Push the codeBlock onto the stack like an object
        hb_vmSend( 0 );     // Execute with no parameters, as if it were an object
    }
}

/*
 * Evaluate a codeBlock passed from C.
 * Harbour's newer proposal.
 * Works exactly like object methods.
 * This is the same example as the EVALUA01 function above.
 */
HB_FUNC( EVALUA03 )
{
    PHB_ITEM pCB = hb_param( 1, HB_IT_BLOCK ); // The passed codeBlock

    if( pCB )
    {
        hb_vmPushEvalSym(); // Push the eval() symbol onto the stack
        hb_vmPush( pCB );   // Push the codeBlock onto the stack like an object
        hb_vmPush( hb_param( 2, HB_IT_ANY ) );
        hb_vmPush( hb_param( 3, HB_IT_ANY ) );
        hb_vmSend( 2 );     // Execute with two parameters, as if it were an object
    }
}
```

Using the functions above from PRG

```
//-----------------------------------------------------------------------------
// Exercise: using codeBlocks in C. Evaluate a codeBlock without parameters from C,
// in Clipper’s old style and Harbour’s newer style.
// ej014.prg
//-----------------------------------------------------------------------------

#include "HBClass.ch"

//-----------------------------------------------------------------------------

PROCEDURE main

    local bCB0 := { || Alert( "Hello world" ) }
    local bCB1 := { | p1, p2 | Alert( "The parameters passed: " + HB_ValToStr( p1 ) + ;
                          " and " + HB_ValToStr( p2 ) ) }
    local bCB2 := { | p1 | Alert( "Direct with one parameter: " + HB_ValToStr( p1 ) ) }

    evalua00( bCB0 )
    evalua01( bCB1, "First parameter", Date() )
    // NOTE: In Harbour, codeBlocks can be treated as objects.
    // The object is the codeBlock variable and with the ::eval() method
    // you pass the parameters and evaluate it.
    bCB1:eval( "Treat codeBlock as object", Time() )
    // Using Harbour’s newer approach
    evalua02( bCB0 )
    evalua03( bCB1, "Parameter 1", Seconds() )

    Alert( "Direct functions..." )
    evalua04( bCB0 )
    evalua05( bCB2, "With parameter directly" )
    evalua06( bCB1, "With one parameter...", "another parameter" )

RETURN
```

18. The Error API — Handling Harbour Errors from C

One of the most important responsibilities in a programming language is handling exceptions, warnings, and errors. These can be raised directly by the program at runtime or intentionally triggered by the programmer to control exceptional conditions that the language itself wouldn't detect. For example, inserting a database record with a duplicate unique key is not a language-level error, but we *do* want to catch that database exception and handle it as a normal error.

Harbour provides a system that is both simple and powerful. The `TError` class is responsible for this work. This class has no methods; it only exposes instance variables (datas) that represent the state of the error object.

Usually, the runtime itself creates the error object; at most, you might have a custom error handler to control screen output or write to a log file. But sometimes you need to create your own error object and manage it manually. In PRG there is a function that does exactly that: `ErrorNew()`. Once the object is created, you can access its DATAs and assign values directly:

```
local oError := ErrorNew() // Create the object

// Assign new values to the DATAs of the freshly created object:
oError:severity    := ES_ERROR
oError:genCode     := EG_UNSUPPORTED
oError:subSystem   := "MI_LIB"
oError:subCode     := 0
oError:description := "This is an error I'm handling myself..."
oError:canRetry    := .F.
oError:canDefault  := .F.
oError:fileName    := ""
oError:osCode      := 0

// Raise the error so it's handled by the default (or your custom) error handler
Eval( ErrorBlock(), oError )
```

We can do the very same thing from C using the **Error API**. For that we have a set of functions to create the object, set/get the values of its datas, launch it once prepared, and finally free it. Below are the datas (instance variables) that you can access via C functions or directly from PRG:

- **Args**: Contains an array of arguments supplied to an operator or function when an argument error occurs. For other error types, `Args` contains `NIL`.
- **CanDefault**: Logical value indicating whether the subsystem can perform a default error recovery for the condition. `.T.` means a default recovery is available. Availability and actual strategy depend on

the subsystem and the error condition. The minimal action is to simply ignore the error. Default recovery is requested by returning `.F.` from the error block invoked to handle the error. Note: `canDefault` is never `.T.` if `canSubstitute` is `.T.`

- **CanRetry**: Logical value indicating whether the subsystem can retry the operation that caused the error. `.T.` means retry is possible. Whether it's actually possible depends on the subsystem and the condition. A retry is requested by returning `.T.` from the error block. `canRetry` is never `.T.` if `canSubstitute` is `.T.`
- **CanSubstitute**: Logical value indicating whether a new result can be substituted for the failed operation. Argument errors and some other simple errors allow the error handler to substitute a new result value. `.T.` means substitution is possible. Substitution is done by returning the new result value from the error block. `canSubstitute` is never `.T.` if `canDefault` or `canRetry` is `.T.`
- **Cargo**: Contains a value of any data type not used by the error system. This is a user-definable slot allowing you to attach arbitrary information to an error object and retrieve it later.
- **Description**: String describing the error condition. An empty string indicates that the subsystem did not provide a printable description. If `genCode` is non-zero, a printable description is always available.
- **FileName**: String representing the original name used to open the file associated with the error condition. An empty string indicates the error is not associated with a specific file, or the subsystem doesn't retain the filename.
- **GenCode**: Integer value representing a generic Harbour error code. Generic codes allow default handling of similar errors across subsystems. A value of `0` indicates the condition is subsystem-specific and does not correspond to any generic code. The header `error.ch` provides manifest constants for generic error codes.
- **Operation**: String describing the operation being attempted when the error occurred. For operators and functions, it contains the operator or function name. For undefined variables or functions, it contains their name. An empty string indicates the subsystem provided no printable description of the operation.
- **OsCode**: Integer value representing the OS error code associated with the condition. `0` indicates the condition was not caused by an OS error. When `osCode` is non-zero, `DosError()` is set to the same value. `osCode` correctly reflects the DOS extended error code for file errors, allowing proper distinction between sharing violations (e.g. opening `EXCLUSIVE` while another process has the file open) and access violations (e.g. opening read/write when the file is read-only). See the appendices/error messages guide for a list of DOS error codes.
- **Severity**: Integer indicating severity. Four standard values are defined in `error.ch`:
  - `ES_WHOCARES`: Informational; not a failure.

    - `ES_WARNING`: Does not prevent further operations but may lead to more serious errors later.
    - `ES_ERROR`: Prevents further operations without corrective action.
    - `ES_CATASTROPHIC`: Requires immediate termination of the application.

  Note: Harbour runtime support only generates errors with severities `ES_WARNING` or `ES_ERROR`.

- **SubCode**: Integer representing a subsystem-specific error code. `0` indicates the subsystem does not assign a specific number to the condition.
- **Subsystem**: String with the name of the subsystem generating the error. For core Harbour operators/functions, the subsystem name is `BASE`. For database driver errors, `subsystem` contains the driver name.
- **Tries**: Integer representing how many times the failed operation has been attempted. When `canRetry` is `.T.`, `tries` can be used to limit the number of retries. `0` indicates the subsystem does not track the count.

GET functions (read the value of a DATA)

```
PHB_ITEM   hb_errGetCargo( PHB_ITEM pError );        // Cargo as ITEM (can be any Harbour
type)
PHB_ITEM   hb_errGetArgs( PHB_ITEM pError );         // Args as ITEM array
const char *hb_errGetDescription( PHB_ITEM pError );
const char *hb_errGetFileName( PHB_ITEM pError );
HB_USHORT  hb_errGetFlags( PHB_ITEM pError );
HB_ERRCODE hb_errGetGenCode( PHB_ITEM pError );
const char *hb_errGetOperation( PHB_ITEM pError );
HB_ERRCODE hb_errGetOsCode( PHB_ITEM pError );
HB_USHORT  hb_errGetSeverity( PHB_ITEM pError );
HB_ERRCODE hb_errGetSubCode( PHB_ITEM pError );
const char *hb_errGetSubSystem( PHB_ITEM pError );
HB_USHORT  hb_errGetTries( PHB_ITEM pError );
```

All these functions take the error object as their only parameter and return C types (except the first two, which return Harbour `ITEM`s).

SET functions (assign the value of a DATA)

```
PHB_ITEM hb_errPutCargo( PHB_ITEM pError, PHB_ITEM pCargo );
PHB_ITEM hb_errPutArgsArray( PHB_ITEM pError, PHB_ITEM pArgs );
PHB_ITEM hb_errPutArgs( PHB_ITEM pError, HB_ULONG ulArgCount, ... );
PHB_ITEM hb_errPutDescription( PHB_ITEM pError, const char * szDescription );
PHB_ITEM hb_errPutFileName( PHB_ITEM pError, const char * szFileName );
PHB_ITEM hb_errPutFlags( PHB_ITEM pError, HB_USHORT uiFlags );
PHB_ITEM hb_errPutGenCode( PHB_ITEM pError, HB_ERRCODE uiGenCode );
PHB_ITEM hb_errPutOperation( PHB_ITEM pError, const char * szOperation );
PHB_ITEM hb_errPutOsCode( PHB_ITEM pError, HB_ERRCODE uiOsCode );
PHB_ITEM hb_errPutSeverity( PHB_ITEM pError, HB_USHORT uiSeverity );
PHB_ITEM hb_errPutSubCode( PHB_ITEM pError, HB_ERRCODE uiSubCode );
PHB_ITEM hb_errPutSubSystem( PHB_ITEM pError, const char * szSubSystem );
PHB_ITEM hb_errPutTries( PHB_ITEM pError, HB_USHORT uiTries );
PHB_ITEM hb_errNew( void );                              // Create an empty error object
HB_USHORT hb_errLaunch( PHB_ITEM pError );               // Launch the error (like: Eval(
ErrorBlock(), oError ))
void      hb_errRelease( PHB_ITEM pError );              // Free datas and the error object
itself
```

`hb_errLaunch()` can return `E_BREAK` (interrupts processing with no further retries), `E_RETRY` (the operation can be attempted again), or `E_DEFAULT` (initial/default value).

Practical example

```
//-----------------------------------------------------------------------------
// Exercise: using error handling from C.
// ej015.prg
//-----------------------------------------------------------------------------

#include "error.ch"

PROCEDURE main

   cls

   ? "Generate a system error from C:"
   miThowError( ES_WARNING, 23, "This is system warning 23" )
   miThowError( ES_ERROR, 55, "This is system error 55" )
   miThowError( ES_CATASTROPHIC, 63, "This is catastrophic error 63" )

RETURN
```

Another PRG where error messages are stored in an array; the error number is the index where the description is stored:

```
//----------------------------------------------------------------------------
// Exercise: using error handling from C.
// ej016.prg
//----------------------------------------------------------------------------

#include "error.ch"

PROCEDURE main

   cls

   ? "Generate a system error from C:"
   autoErr( ES_WARNING, 2 )
   autoErr( ES_ERROR, 1 )
   autoErr( ES_WHOCARES, 3 )

RETURN

//----------------------------------------------------------------------------

procedure autoErr( nNivel, nCodErr )

   local aErr := { "This is error 1", ;
                   "This is error 2", ;
                   "This is error 3" }

   if nCodErr >= 1 .and. nCodErr <= 3
      miThowError( nNivel, nCodErr, aErr[ nCodErr ] )
   endif

return
```

And here is the C function (verbatim):

```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)" ); // Subsystem name
   hb_errPutSubCode( pError, errCode );                     // Our system error code
   hb_errPutDescription( pError, szMsg );                   // Message
   hb_errPutSeverity( pError, uiLevel );                    // Error severity

   hb_errPutFlags( pError, EF_CANDEFAULT );                 // Enable default button
   hb_errPutTries( pError, 5 );

   hb_errLaunch( pError );                                  // Launch the prepared error
   hb_errRelease( pError );                                 // Free the object
}
```

19. The FileSys API: Working with Files from C

In this chapter we cover another very important Harbour API: file handling. As usual, implementing our processes in C can speed up creation, deletion, and general file operations.

Most C functions in the FileSys API have a PRG counterpart, which makes their usage easy to intuit.

Main functions

- `HB_BOOL hb_fsChDir( const char *pszDirName );`
  Change to the specified directory.
- `void hb_fsClose( HB_FHANDLE hFileHandle );`
  Close the file.
- `void hb_fsCommit( HB_FHANDLE hFileHandle );`
  Flush pending file changes to disk.
- `HB_FHANDLE hb_fsCreate( const char *pszFileName, HB_FATTR ulAttr );`
  Create and open a file.
- `HB_FHANDLE hb_fsCreateEx( const char *pszFileName, HB_FATTR ulAttr, HB_USHORT uiFlags );`
  Create and open a file with a specific open mode.
- `const char *hb_fsCurDir ( int iDrive );`
  Return the current directory for the specified drive.
- `int hb_fsCurDrv( void );`
  Get the current drive number.
- `HB_BOOL hb_fsDelete( const char *pszFileName );`
  Delete a file.
- `HB_BOOL hb_fsEof( HB_FHANDLE hFileHandle );`
  Check whether end-of-file has been reached.
- `HB_ERRCODE hb_fsError( void );`
  Retrieve the last filesystem error.
- `HB_BOOL hb_fsFile( const char *pszFileName );`
  Determine whether the file exists.
- `HB_BOOL hb_fsIsDirectory( const char *pszFileName );`
  Determine whether the given name is a directory.

- `HB_FOFFSET hb_fsFSize( const char *pszFileName, HB_BOOL bUseDirEntry );`
  Get a file's size.

- `HB_FHANDLE hb_fsExtOpen( const char *pszFileName, const char *pDefExt, HB_FATTR nFlags, const char *pPaths, PHB_ITEM pError );`
  Open a file with a default extension, searching a list of paths.

- `HB_ERRCODE hb_fsIsDrv( int iDrive );`
  Check whether a drive number is valid.

- `HB_BOOL hb_fsMkDir( const char *pszDirName );`
  Create a directory.

- `HB_FHANDLE hb_fsOpen( const char *pszFileName, HB_USHORT uiFlags );`
  Open or create a file.

- `HB_FHANDLE hb_fsOpenEx( const char *pszFileName, HB_USHORT uiFlags, HB_FATTR nAttr );`
  Open or create a file with the given attributes.

- `HB_USHORT hb_fsRead( HB_FHANDLE hFileHandle, void *pBuff, HB_USHORT uiCount );`
  Read up to 64 KiB from the file at the current position into a buffer.

- `HB_SIZE hb_fsReadLarge( HB_FHANDLE hFileHandle, void *pBuff, HB_SIZE nCount );`
  Read more than 64 KiB from the file at the current position into a buffer.

- `HB_SIZE hb_fsReadAt( HB_FHANDLE hFileHandle, void *pBuff, HB_SIZE nCount, HB_FOFFSET nOffset );`
  Read from a given file position into a buffer (can exceed 64 KiB).

- `HB_BOOL hb_fsRmDir( const char *pszDirName );`
  Remove a directory.

- `HB_BOOL hb_fsRename( const char * pszOldName, const char *pszNewName );`
  Rename a file.

- `HB_ULONG hb_fsSeek( HB_FHANDLE hFileHandle, HB_LONG lOffset, HB_USHORT uiMode );`
  Move the file R/W pointer to a given position.

- `HB_FOFFSET hb_fsSeekLarge( HB_FHANDLE hFileHandle, HB_FOFFSET nOffset, HB_USHORT uiFlags );`
  Move the file R/W pointer using the 64-bit API.

- `HB_FOFFSET hb_fsTell( HB_FHANDLE hFileHandle );`
  Return the current file position.

- `HB_FOFFSET hb_fsGetSize( HB_FHANDLE hFileHandle );`
  Return the file size (warning: may change the current file position).

- `int hb_fsSetDevMode( HB_FHANDLE hFileHandle, int iDevMode );`
  Set how a file is treated: text or binary.

- `HB_USHORT hb_fsWrite( HB_FHANDLE hFileHandle, const void *pBuff, HB_USHORT uiCount );`
  Write up to 64 KiB from a buffer to an open file.

- `HB_SIZE hb_fsWriteLarge( HB_FHANDLE hFileHandle, const void *pBuff, HB_SIZE nCount );`
  Write more than 64 KiB from a buffer to an open file.

- `HB_SIZE hb_fsWriteAt( HB_FHANDLE hFileHandle, const void *pBuff, HB_SIZE nCount, HB_FOFFSET nOffset );`
  Write from a buffer to an open file starting at a given offset (can exceed 64 KiB).

- `HB_BOOL hb_fsNameExists( const char *pszFileName );`
  Check whether a name exists in the system (no wildcards).

- `HB_BOOL hb_fsFileExists( const char *pszFileName );`
  Check whether a file exists (no wildcards).

- `HB_BOOL hb_fsDirExists( const char *pszDirName );`
  Check whether a directory exists (no wildcards).

- `HB_BOOL hb_fsCopy( const char *pszSource, const char *pszDest );`
  Copy a file to a new name.

This is not a complete list of the API, but it does include most of the common functions.

Parameter meanings:

- `pszDirName` → Directory name
- `pszFileName` → File name
- `hFileHandle` → File handle
- `ulAttr` → Attributes
- `uiFlags` → Open flags
- `iDrive` → Drive number
- `pBuff` → Input/output buffer

Example: A simple log system

C implementation

```c
/*****************************************************************************
 ************************* Using the FileSys API ****************************
 *****************************************************************************/

#include "hbapi.h"     /* Always include this for C functions */
#include "hbapifs.h"   /* System (filesystem) API */
#include "hbapiitm.h"  /* ITEM API */

/***
 * Log manager in C
 * IMPORTANT: the last argument must be NULL
 * Example: writeLog("miFic.log", HB_FALSE, "Prueba ", "de ", "LOGMANAGER", NULL);
 * Writes the strings to the file miFic.log; if it exists, append to the end;
 * if it does not, create it and write.
 */

static const char *nfln;
static int iNflnLen;

/*
 * Write to the LOG file
 */
void writeLog( const char *szFileName, HB_BOOL bCreate, const char *zStr, ... )
{
    if( szFileName && zStr )  /* If parameters are provided */
    {
        HB_FHANDLE hFile;
        HB_BOOL bExist = hb_fsFileExists( szFileName ); /* Check if file exists */

        if( bCreate == HB_TRUE || bExist == HB_FALSE )
        {
            hFile = hb_fsCreate( szFileName, FC_NORMAL ); /* Create the file */
        }
        else
        {
            hFile = hb_fsOpen( szFileName, FO_READWRITE ); /* Open the file */
        }

        if( hFile != FS_ERROR ) /* If no errors */
        {
            va_list vl;

            nfln = hb_conNewLine();
            iNflnLen = strlen( nfln );

            hb_fsSeekLarge( hFile, 0, FS_END );
            va_start( vl, zStr );

            do  /* Traverse the variadic arguments and write each to the LOG */
            {
                /* Use the Large variant in case a block exceeds 64K */
                hb_fsWriteLarge( hFile, zStr, strlen( zStr ) );
```

```
                hb_fsWrite( hFile, nfln, iNflnLen ); /* Write end-of-line */
            }
            while( ( zStr = va_arg( vl, char * ) ) != NULL );

            va_end( vl );

            /* Close also commits, so hb_fsCommit() is not required */
            hb_fsClose( hFile );
        }
    }
}

/*
 * writeLog for PRG usage
 * Accepts only one parameter to write, but it can be of any type (not just string)
 */
HB_FUNC( WRITELOG )
{
    const char *szFileName = hb_parc( 1 );            /* File name */
    HB_BOOL bCreate = hb_parldef( 2, HB_FALSE );      /* Force creation */

    if( szFileName && hb_pcount() > 2 )
    {
        HB_SIZE nLen;
        HB_BOOL bFreeReq;
        /* Convert the passed value to string */
        char *szStr = hb_itemString( hb_param( 3, HB_IT_ANY ), &nLen, &bFreeReq );

        /* Call the C function that writes to the LOG */
        writeLog( szFileName, bCreate, szStr, NULL );

        if( bFreeReq )
        {
            /* Free memory if hb_itemString() indicated that it's required */
            hb_xfree( szStr );
        }
    }
}

/*
 * Load the LOG file into a variable
 */
char *loadLog( const char *szFileName, HB_SIZE *nSize )
{
    HB_FHANDLE hFile = hb_fsOpen( szFileName, FO_READ ); /* Open file read-only */
    char *szValue = NULL;

    if( hFile != FS_ERROR )
    {
        *nSize = hb_fsGetSize( hFile );            /* Get file size */

        /* Allocate buffer: file size + newline */
        szValue = ( char * ) hb_xgrab( *nSize + iNflnLen );
```

```
        hb_fsSeekLarge( hFile, 0, FS_SET );        /* Seek to beginning */
        hb_fsReadLarge( hFile, szValue, *nSize );  /* Read all into buffer */
        hb_fsClose( hFile );                       /* Close LOG file */
    }

    return szValue; /* Return the LOG content in this variable */
}

/*
 * Load the LOG file into a variable from PRG
 */
HB_FUNC( LOADLOG )
{
    const char *szFileName = hb_parc( 1 ); /* LOG file name */

    if( szFileName )
    {
        HB_SIZE nSize = 0;
        char *szValue = loadLog( szFileName, &nSize );

        /* Return the LOG file content */
        hb_retclen_buffer( szValue, nSize );
    }
    else
    {
        /* If no file name passed, return a NULL string */
        hb_retc_null();
    }
}
```

PRG usage: several examples

```
//----------------------------------------------------------------------------
// Example usage of the log manager system
// Program ej01.prg
//----------------------------------------------------------------------------

PROCEDURE Main

    LOCAL n := 0
    LOCAL cFNane := "miArchivo.log"

    CLS

    // Reset if the file exists
    // Simulate errors
    writeLog( cFNane, .t., hb_TToC( hb_DateTime() ) + ;
        " - Se ha producido el error numero: " + HB_NToS( n ) )
    // Append
    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 )

    ?

    // Write to the file. If it exists, destroy and recreate it (.t.)
    writeLog( cFNane, .t., "----------------------------------------------------------------
------" )
    // Write the following lines to the existing file
    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
```

```
//----------------------------------------------------------------------------
// Example usage of the log manager system
// Program ej02.prg
//----------------------------------------------------------------------------

#include "Box.ch"

PROCEDURE Main

    LOCAL n := 0
    LOCAL cFNane := "miArchivo.log"

    // Reset if the file exists
    // Simulate errors
    writeLog( cFNane, .t., hb_TToC( hb_DateTime() ) + ;
        " - Se ha producido el error numero: " + HB_NToS( n ) )
    // Append
    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 )

    // Write to the file. If it exists, destroy and recreate it (.t.)
    writeLog( cFNane, .t., "---------------------------------------------------------------" )
    // Write the following lines to the existing file
    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
```

```
//----------------------------------------------------------------------------
// Show the file’s content using the log system
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

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

A class in C that buffers writes to a file

```
//----------------------------------------------------------------------------
// Class that manages a file buffer
// Implemented almost entirely in C

#include "hbclass.ch"

#define FB_MAX_SIZE  65535

CLASS TFBuffer

    DATA hFB  // Internal use

    METHOD new( hFile, nSizeBuffer ) CONSTRUCTOR
    METHOD free()              // Free memory
    METHOD creaBuffer( hFile, nSizeBuffer )  // Internal use
    METHOD addString( cString ) // Append a string to the buffer; auto ::flush() as needed
    METHOD addValue( value )    // Convert a value to string and append to the buffer
    METHOD flush()              // Optionally force a manual flush to disk

END CLASS

//----------------------------------------------------------------------------
// Class constructor

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
```

C implementation for the TFBuffer methods

```
//-----------------------------------------------------------------------------
// C implementation of the methods

#pragma BEGINDUMP

#include "hbapifs.h"
#include "hbapiitm.h"
#include "hbstack.h"

/* Buffer structure */
typedef struct
{
    unsigned char *pBuffer;
    unsigned int uiLen;
    unsigned int uiPos;
    HB_FHANDLE hFile;
} FBUFFER, *PFBUFFER;

static void __flushFB( PFBUFFER pFB );

//-----------------------------------------------------------------------------
// Method to create the buffer structure
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 );
}

//-----------------------------------------------------------------------------
// Method to free the 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 );
    }
}

//-----------------------------------------------------------------------------
// Method to append strings to the 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++ ];
        }
    }
}

//-----------------------------------------------------------------------------
// Method to append arbitrary values to the buffer (converted to string)
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 );
        }
    }
}

//-----------------------------------------------------------------------------
// Method to force a flush to disk and reset write position
HB_FUNC_STATIC( TFBUFFER_FLUSH )
{
    PHB_ITEM Self = hb_stackSelfItem();
    PFBUFFER pFB = hb_arrayGetPtr( Self, 1 );

    __flushFB( pFB );
}

//-----------------------------------------------------------------------------
// Internal helper: write buffer to disk and reset position
static void __flushFB( PFBUFFER pFB )
{
    if( pFB->uiPos > 0 )
    {
        hb_fsWriteLarge( pFB->hFile, pFB->pBuffer, pFB->uiPos );
        pFB->uiPos = 0;
    }
}

#pragma ENDDUMP
```

Usage examples of `TFBuffer`

```
//----------------------------------------------------------------------------
// Example usage of TFBuffer
// Program 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()

    // For strings you can use ::addString(); for everything else, ::addValue()
    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 )

    // Compute size in KiB
    i := Hb_FSize( "prueba.log" ) / 1024

    Alert( "Ha tardado: " + AllTrim( Str( t ) ) + ;
        " segundos con un fichero de: " + AllTrim( Str( i ) ) + " kB")

RETURN

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

```
//----------------------------------------------------------------------------
// Example usage of TFBuffer
// Program ej04.prg
//----------------------------------------------------------------------------

#include "tfbuffer.prg"

PROCEDURE Main

    LOCAL o := TFBuffer():new( "prueba.log" )  // If a name is passed, it creates and opens it
    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()

    // For strings you can use ::addString(); for everything else, ::addValue()
    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 → also close the file

    // Compute size in KiB
    i := Hb_FSize( "prueba.log" ) / 1024

    Alert( "Ha tardado: " + AllTrim( Str( t ) ) + ;
        " segundos con un fichero de: " + AllTrim( Str( i ) ) + " kB")

RETURN
```

> Notes
>
> - Comments inside C/PRG examples have been translated to English. The string literals remain as in the original examples, exactly as requested.
> - If you ever want the literals localized too, I can provide a separate variant.

20. Creating our own libraries of functions

Most of us are familiar with the concept of a **library** of functions. In fact, we use one every day whenever we build our programs.
But what *is* a function library? We could say it's a **container of functions and procedures** stored in a file, usually with an extension like `*.lib`, `*.a`, `*.dll`, or `*.so`. We'll explain each of these later.

That file should be organized around functions and procedures that share a common purpose — for example: math routines, string processing, screen output, reporting, etc.

When we have a set of functions used in more than one program and their code is well tested, it's time to create a **library**. Doing so gives us clarity and convenience.

Many of the libraries we use already come with our favorite compiler or are provided by a third-party vendor. The good news is that **we can create our own**, which lets us organize our code better and manage those functions and procedures we use across all our programs.

In this chapter, we'll focus on creating **libraries for Harbour**, so the code we put in them has to be understood by our compiler. That means they must be written in **PRG** or **C** (using the various APIs Harbour provides).

To use the functions and procedures included in libraries, we need to know the **parameters** they receive and the **values** they return. This applies to both **static** libraries (`*.lib` or `*.a`) and **dynamic** ones.

Basically, there are two kinds of libraries (plus one related case):

1. **Static.** The functions or procedures they contain are **included in the final executable** once it's compiled and linked, so the library file is no longer required for the program to run. On Windows, they're typically `*.lib` for Borland 32-bit, MSVC, Pelles C, etc. On Unix/Linux/macOS and similar, they're typically `*.a`. Some Windows-ported compilers also use `*.a`, such as MinGW, Clang, and Clang-based toolchains (e.g., Borland 64-bit 7.xx).
2. **Dynamic.** The functions are **not** included in the executable; the exe only **references** them. This means the library must be present at runtime for the program to execute. On Windows the usual extension is `*.dll`, and on Unix/Linux/macOS the extension is typically `*.so`.
3. **Import libraries.** These are **static** libraries that contain **references** to the functions provided by **dynamic** libraries — essentially a **symbol table**. They're needed only at **compile/link time**, and therefore do **not** need to be present at runtime.

As mentioned earlier, we continually use libraries provided by Harbour, the C compilers, and third parties such as FWH, Xailer, or xHarbour.com. But the important point is that **we can also create our own**, and that's the purpose of this chapter.

Each C compiler has its own way to build libraries, so we would need to explain the process for each of them. The good news is that Harbour's `hbmk2` tool takes care of invoking the necessary commands for each C compiler, freeing us from that complexity.

I think the best way to explain how to do this is with an example that you can adapt to your needs.

To create the library we'll use Harbour's `hbmk2` tool, so we need to know a few options from the tool:

- `-o<outname>` — output file name
- `-i<p>` or `-incpath=<p>` — additional include paths for header files
- `-hblib` — create a **static** library
- `-hbdyn` — create a **dynamic** library (**without** linking to the Harbour VM)
- `-hbdynvm` — create a **dynamic** library (**with** linking to the Harbour VM)
- `-implib=<output>` — create an **import** library (in `-hbdyn` / `-hbexe` mode) named `<output>` (default: same as output)
- `-hbimplib` — create an **import** library (**Windows only**)

These are the basic options we'll use; `hbmk2` has many more.

We should also create a **project file** where we include the options and the PRG/C source files that will contain the functions and procedures of our library. Harbour's `hbmk2` project files use the `*.hbp` extension.

> `*.hbp` — **project file.** May contain any number of command-line options (the same ones you'd pass to create a final target). Lines starting with `#` are ignored. Newlines are optional; options should be separated by spaces, as on the command line. Options containing spaces must be wrapped in double quotes. Each referenced `.hbp` file is executed as a **sub-project**.

With all this, we're ready to create our libraries. Typically, we create a project file listing the PRG and C sources and the options.

Below is the **batch** script that will call `hbmk2`:

Source `buildlib.bat`:

```
@rem ----------------------------------------------------------------------
@rem Batch file to build libraries
@rem To adapt to your system, change these variables:
@rem COMP  C compiler for Windows; valid values include: mingw, mingw64, clang,
@rem       clang64, msvc, msvc64, clang-cl, clang-cl64, watcom, icc, icc64,
@rem       iccia64, msvcia64, bcc, bcc64, pocc, pocc64
@rem DIR_HBBIN  Harbour bin directory
@rem DIR_CCBIN  C compiler bin directory
@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    There are compilation errors
@echo ----------------------------------
@pause
:fin_exec
@set PATH=%PATHOLD%
```

And here is the **first** `hbmk2` project file template:

```
#-----------------------------------------------------------------------------
#  AUTHOR....: Manuel Expósito Suárez 2021
#  CLASS.....: cursodec.hbp
#  MOD DATE..: 2021-08-05
#  VERSION...: 1.00
#  PURPOSE...: Build script for the library for all C compilers using Harbour hbmk2
#-----------------------------------------------------------------------------

# Indicate that we are going to build a library
-hblib

# If there were header (include) files for our library, we would indicate their location
# in our case there are none
# -i./src/include

# List all PRGs (and directories) that are part of the library
./src/prg/tfbuffer.prg
./src/prg/thbuffer.prg

# List all C files (and directories) that are part of the library
./src/c/cursode1.c
./src/c/cursode2.c

# Library name and where to place it once built
-o./lib/cursode

#-----------------------------------------------------------------------------
```

And here is a **variant** that automatically adds all sources from a folder:

```
#-------------------------------------------------------------------------------
#  AUTHOR....: Manuel Expósito Suárez 2021
#  CLASS.....: cursodec.hbp
#  MOD DATE..: 2021-08-05
#  VERSION...: 1.00
#  PURPOSE...: Build script for the library for all C compilers using Harbour hbmk2
#-------------------------------------------------------------------------------

# Indicate that we are going to build a library
-hblib

# If there were header (include) files for our library, indicate their location
# -i./src/include

# Indicate that all PRGs in the given directory are part of the library
./src/prg/*.prg

# Indicate that all C files in the given directory are part of the library
./src/c/*.c

# Library name and where to place it once built
-o./lib/cursode

#-------------------------------------------------------------------------------
```

This last template has the drawback that if there are files in the folder **by mistake** that should not be part of our library, they will be included as well — which is why I recommend the **first** template.

The first time you build the library it usually takes longer than subsequent builds because `hbmk2` is smart and only processes those files whose timestamp has changed since the last build.

21. Interfaces to DLL Functions

In this chapter, we'll learn how to use functions stored in a **DLL** from Harbour.
But first, we must be clear about a few ideas...

As we've seen in a previous topic, there are **two types of libraries** or **function collections**: *static* and *dynamic*.
What they have in common is that both store **functions** that can be used by our programs. In other words, a function library is a **container** where a set of reusable functions is kept so they can be included in our executable programs.

What determines whether a library is one type or the other is the **moment it links** with our program.

- **Static libraries** are linked during the **compile/link process**, so their functions are embedded inside the executable, and the library itself is no longer needed at runtime.
- **Dynamic libraries**, on the other hand, provide their functions only when the executable **requests** them dynamically — hence the name. Therefore, the file containing those functions must be **present while the program is running**.

For the executable to know about the public functions in a DLL, it needs a mechanism that provides that information.
This is done via **import libraries**, which are special **static libraries** that act as **symbol tables**, containing metadata about the DLL functions (their names and an ordinal number that will later be replaced by the memory address where the executable should look to locate each function).

Each type of library has its **advantages** and **drawbacks**:

- Using static libraries makes executables **larger**, but removes dependence on external programs and avoids compatibility problems that might arise with different versions of dynamic libraries (DLLs).
- Using dynamic libraries keeps executables **smaller**, and allows several programs to share the same DLL functions.

The way to build **static** and **dynamic** libraries differs.
Static libraries typically use extensions like `*.lib` or `*.a`, while dynamic ones are `*.dll` on Windows and `*.so` on Linux and other operating systems.

The term **DLL** stands for *Dynamic Link Library* in Windows, and the equivalent concept in Unix-like systems is **SO**, meaning *Shared Object*.

To use DLLs from a Harbour PRG we need:

1. the **import library** for the DLL, and
2. another library containing the **interface functions**.

How to create import libraries

Example: generating an import definition file (.DEF)

With GNU tools:

```
pexports libmariadb.dll > libmariadb.def
```

or

```
gendef libmariadb.dll
```

With Borland:

```
impdef libmariadb.def libmariadb.dll
```

Example: creating the import library

With GNU:

```
dlltool -d libmariadb.def -D libmariadb.dll -k -l libmariadb.a
```

With Borland:
(the next steps would follow the same logic but using Borland's import-library tools; details appear later in the book)