Harbour (programming language)

Harbour is a computer programming language, primarily used to create database/business programs. It is a modernized, open source and cross-platform version of the older Clipper system, which in turn developed from the dBase database market of the 1980s and 1990s.

Harbour code using the same databases can be compiled under a wide variety of platforms, including Microsoft Windows, Linux, Unix variants, several BSD descendants, Mac OS X, MINIX 3, Windows CE, Pocket PC, Symbian, iOS, Android, QNX, VxWorks, OS/2 (including eComStation and ArcaOS),[1] BeOS/Haiku, AIX and MS-DOS.

History

The idea of a free software Clipper compiler had been floating around for a long time and the subject has often cropped up in discussion on comp.lang.clipper. Antonio Linares founded the Harbour project and the implementation was started in March 1999. The name "Harbour" was proposed by Linares, it is a play on a Clipper as a type of ship. Harbour is a synonym for port (where ships dock), and Harbour is a port of the Clipper language.

In 2009 Harbour was substantially redesigned, mainly by Viktor Szakáts and Przemyslaw Czerpak.

Database support

Harbour extends the Clipper Replaceable Database Drivers (RDD) approach. It offers multiple RDDs such as DBF, DBFNTX, DBFCDX, DBFDBT and DBFFPT. In Harbour multiple RDDs can be used in a single application, and new logical RDDs can be defined by combining other RDDs. The RDD architecture allows for inheritance, so that a given RDD may extend the functionality of other existing RDD(s). Third-party RDDs, like RDDSQL, RDDSIX, RMDBFCDX, Advantage Database Server, and Mediator exemplify some of the RDD architecture features. DBFNTX implementation has almost the same functionality of DBFCDX and RDDSIX. NETIO and LetoDB[2] provide remote access over TCP protocol.

Harbour Project

Paradigm	multi-paradigm: imperative, functional, object-oriented, reflective
Designed by	Antonio Linares
Developer	Viktor Szakáts and community
First appeared	1999
Stable release	3.0.0 / 17 July 2011
Preview release	3.2.0dev (https://sourceforge.net/projects/harbour-project/files/binaries-windows/nightly/)
Typing discipline	Optionally duck, dynamic, safe, partially strong
OS	Cross-platform
License	Open-source GPL-compatible
Filename extensions	.prg, .ch, .hb, .hbp
Website	harbour.github.io (https://harbour.github.io/)
Dialects	
Clipper, Xbase++, FlagShip, FoxPro, xHarbour	
Influenced by	
dBase, Clipper	
Influenced	

Harbour also offers ODBC support by means of an OOP syntax, and ADO support by means of OLE. MySQL, PostgreSQL, SQLite, Firebird, Oracle are examples of databases which Harbour can connect to.

xHarbour

xBase technologies often are confused with RDBMS software. Although this is true, xBase is more than a simple database system as at the same time xBase languages using purely DBF can not provide the full concept of a real RDBMS.

Programming philosophy

Harbour aims to be written once, compiled anywhere. As the same compiler is available for all of the above operating systems, there is no need for recoding to produce identical products for different platforms, except when operating system dependent features are used. Cross-compiling is supported with MinGW. Under Microsoft Windows, Harbour is more stable but less well-documented than Clipper, but has multi-platform capability and is more transparent, customizable and can run from a USB flash drive.

Under Linux and Windows Mobile, Clipper source code can be compiled with Harbour with very little adaptation. Most software originally written to run on Xbase++, FlagShip, FoxPro, xHarbour and others dialects can be compiled with Harbor with some adaptation. As of 2010 many efforts have been made to make the transition from other xBase dialects easier.

Harbour can use the following C compilers, among others: GCC, MinGW, Clang, ICC, Microsoft Visual C++ (6.0+), Borland C++, Watcom C, Pelles C and Sun Studio.

Harbour can make use of multiple Graphical Terminal emulation, including console drivers, and Hybrid Console/GUIs, such as GTWvt, and GTWvg.

Harbour supports external GUI's, free (e.g. HBQt, HWGui, Mini-GUI (latest version based on Qt and QtContribs[3]) and commercial (e.g. FiveWin, Xailer). HBQt is a library providing bindings to Qt. HBIDE application is a sample of HBQt potential.

Harbour is 100% Clipper-compatible[4] and supports many language syntax extensions including greatly extended run-time libraries such as OLE, Blat, OpenSSL, Free Image, GD, hbtip, hbtpathy, PCRE, hbmzip (zlib), hbbz2 (bzip2), cURL, Cairo, its own implementation of CA-Tools, updated NanFor libraries and many others. Harbour has an active development community and extensive third party support.

Any xBase language provides a very productive way to build business and data intensive applications. Harbour is not an exception.

Macro Operator (runtime compiler)

One of the most powerful features of xBase languages is the Macro Operator '&'. Harbour's implementation of the Macro Operator allows for runtime compilation of any valid Harbour expression. Such a compiled expression may be used as a VALUE, i.e. the right side of an assignment (rvalue), but such a compiled expression may be used to resolve the left side (lvalue) of an assignment, i.e. private, or public variables, or a database field.

Additionally, the Macro Operator may compile and execute function calls, complete assignments, or even list of arguments, and the result of the macro may be used to resolve any of the above contexts in the compiled application. In other words, any Harbour application may be extended and modified at runtime to compile and execute additional code on-demand.

The latest Macro compiler can compile any valid Harbour code including code to per-process before compile.

Syntax:

```
&( ... )
```

The text value of the expression '...' will be compiled, and the value resulting from the execution of the compiled code is the result.

```
&SomeId
```

is the short form for &(SomeId).

```
&SomeId.postfix
```

is the short form of &(SomeId + "postfix").

Object Oriented Programming

Programming in an OOP style is a broader issue than a specific library or a specific interface, but OOP programming is something many Clipper programmers have come to expect. CA-Clipper 5.2 and especially 5.3 added a number of base classes, and a matching OOP syntax. Libraries such as Class(y) (https://web.archive.org/web/20010501165630/http://appsolutions.com/Classy/), FieWin, Clip4Win, and Top Class provide additional OOP functionality.

Harbour has OOP extensions with full support for classes including inheritance, based on Class(y) syntax. OOP syntax in Harbour is very similar to that of earlier Clipper class libraries so it should be possible to maintain legacy Clipper code with minimal changes.

Syntax and semantics

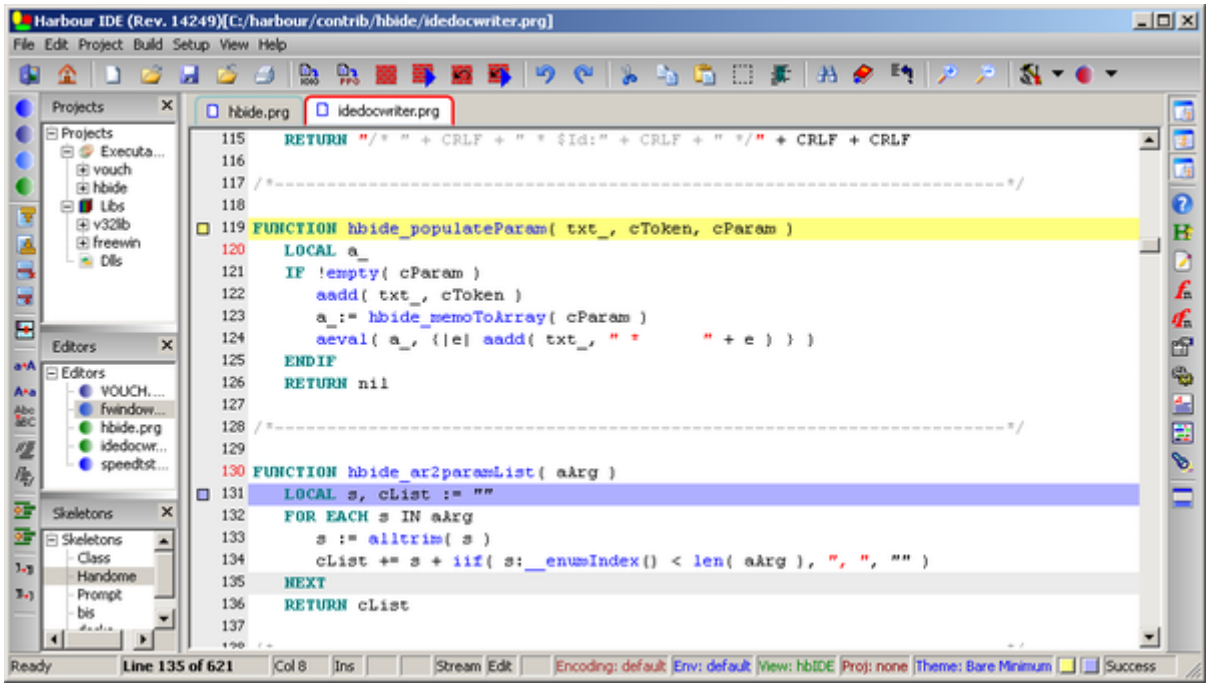
Harbour as every xBase language is case insensitive and can optionally accept keywords written just by their first four characters.

Built-in data types

Harbour has six scalar types : Nil, String, Date, Logical, Numeric, Pointer, and four complex types: Array, Object, CodeBlock, and Hash. A scalar holds a single value, such as a string, numeric, or reference to any other type. Arrays are ordered lists of scalars or complex types, indexed by number, starting at 1. Hashes, or associative arrays, are unordered collections of any type values indexed by their associated key, which may be of any scalar or complex type.

Literal (static) representation of scalar types:

- Nil: `NIL`
- String: `"hello", 'hello', [hello]`
- Date: `0d20100405`



Harbour code on HBIDE

- Logical: `.T., .F.`
- Numeric: `1, 1.1, -1, 0xFF`

Complex Types may also be represent as literal values:

- Array: `{ "String", 1, { "Nested Array" }, .T., FunctionCall(), @FunctionPointer() }`
- CodeBlock: `{ |Arg1, ArgN| Arg1 := ArgN + OuterVar + FunctionCall() }`
- Hash: `{ "Name" => "John", 1 => "Numeric key", "Name2" => { "Nested" => "Hash" } }`

Hashes may use *any* type including other Hashes as the *Key* for any element. Hashes and Arrays may contain *any* type as the *Value* of any member, including nesting arrays, and Hashes.

Codeblocks may have references to Variables of the Procedure/Function>method in which it was defined. Such Codeblocks may be returned as a value, or by means of an argument passed `BY REFERENCE`, in such case the Codeblock will "outlive" the routine in which it was defined, and any variables it references, will be a `DETACHED` variable.

Detached variables will maintain their value for as long as a Codeblock referencing them still exists. Such values will be shared with any other Codeblock which may have access to those same variables. If the Codeblock did not outlive its containing routine, and will be evaluated within the lifetime of the routine in which it is defined, changes to its *Detached Variables*(s) by means of its evaluation, will be reflected back at its parent routine.

Codeblocks can be evaluated any number of times, by means of the `Eval( BlockExp )` function.

Variables

All types can be assigned to named variables. Named variable identifiers are 1 to 63 ASCII characters long, start with `[A-Z|_]` and further consist of the characters `[A-Z|0-9|_]` up to a maximum of 63 characters. Named variables are not case sensitive.

Variables have one of the following scopes:

- `LOCAL`: Visible only within the routine which declared it. Value is lost upon exit of the routine.
- `STATIC`: Visible only within the routine which declared it. Value is preserved for subsequent invocations of the routine. If a STATIC variable is declared before any Procedure/Function/Method is defined, it has a MODULE scope, and is visible within any routine defined within that same source file, it will maintain its life for the duration of the application lifetime.
- `PRIVATE`: Visible within the routine which declared it, and all routines `called` by that routine.
- `PUBLIC`: Visible by `all` routines in the same application.

`LOCAL` and `STATIC` are resolved at compile time, and thus are much faster than `PRIVATE` and `PUBLIC` variables which are dynamic entities accessed by means of a runtime Symbol table. For this same reason, `LOCAL` and `STATIC` variables are `not` exposed to the Macro compiler, and any macro code which attempts to reference them will generate a runtime error.

Due to the dynamic nature of `PRIVATE` and `PUBLIC` variables, they can be created and destroyed at runtime, can be accessed and modified by means of runtime macros, and can be accessed and modified by Codeblocks created on the fly.

Control Structures

The basic control structures include all of the standard dBase, and Clipper control structures as well as additional ones inspired by the C or Java programming languages:

Loops

```
[DO] WHILE ConditionExp
   ...
   [LOOP]
   [EXIT]
END[DO]
```

```
FOR Var := InitExp TO EndExp [STEP StepExp]
   ...
   [LOOP]
   [EXIT]
NEXT
```

```
FOR EACH Var IN CollectionExp
   ...
   [Var:__enumIndex()]
   [LOOP]
   [EXIT]
NEXT
```

- The *...* is a sequence of one or more Harbour statements, and square brackets [ ] denote optional syntax.
- The *Var*:__enumIndex() may be optionally used to retrieve the current iteration index (1 based).
- The *LOOP* statement restarts the current iteration of the enclosing loop structure, and if the enclosing loop is a *FOR* or *FOR EACH* loop, it increases the iterator, moving to the next iteration of the loop.
- The *EXIT* statement immediately terminates execution of the enclosing loop structure.
- The *NEXT* statement closes the control structure and moves to the next iteration of loop structure.

In the *FOR* statement, the *assignment* expression is evaluated prior to the first loop iteration. The *TO* expression is evaluated and compared against the value of the control variable, prior to each iteration, and the loop is terminated if it evaluates to a numeric value greater than the numeric value of the control variable. The optional *STEP* expression is evaluated after each iteration, prior to deciding whether to perform the next iteration.

In *FOR EACH*, the *Var* variable will have the value (scalar, or complex) of the respective element in the collection value. The collection expression may be an Array (of any type or combinations of types), a Hash Table, or an Object type.

IF statements

```
IF CondExp
   ...
[ELSEIF] CondExp
   ...
[ELSE]
   ...
END[IF]
```

... represents 0 or more *statement(s)*.

The condition expression(s) has to evaluate to a *LOGICAL* value.

SWITCH statements

Harbour supports a SWITCH construct inspired by the C implementation of switch().

```
SWITCH SwitchExp
CASE LiteralExp
   ...
   [EXIT]
[CASE LiteralExp]
   ...
   [EXIT]
[OTHERWISE]
   ...
END[SWITCH]
```

- The *LiteralExp* must be a compiled time resolvable numeric expression, and may involve operators, as long as such operators involve compile time static value.
- The *EXIT* optional statement is the equivalent of the C statement *break*, and if present, execution of the SWITCH structure will end when the EXIT statement is reached, otherwise

it will continue with the first statement below the next CASE statement (fall through).

BEGIN SEQUENCE statements

```
BEGIN SEQUENCE
   ...
   [BREAK]
   [Break( [Exp] )]
RECOVER [USING Var]
   ...
END[SEQUENCE]
```

or:

```
BEGIN SEQUENCE
   ...
   [BREAK]
   [Break()]
END[SEQUENCE]
```

The BEGIN SEQUENCE structure allows for a well behaved abortion of any sequence, even when crossing nested procedures/functions. This means that a called procedure/function, may issue a BREAK statement, or a Break() expression, to force unfolding of any nested procedure/functions, all the way back to the first outer BEGIN SEQUENCE structure, either after its respective END statement, or a RECOVER clause if present. The Break statement may optionally pass any type of expression, which may be accepted by the RECOVER statement to allow further recovery handling.

Additionally the Harbour *Error Object* supports *canDefault*, *canRetry* and *canSubstitute* properties, which allows error handlers to perform some preparations, and then request a *Retry Operation*, a *Resume*, or return a Value to replace the expression triggering the error condition.

Alternatively TRY [CATCH] [FINALLY] statements are available on *xhb* library working like the SEQUENCE construct.

Procedures and functions

```
[STATIC] PROCEDURE SomeProcedureName
[STATIC] PROCEDURE SomeProcedureName()
[STATIC] PROCEDURE SomeProcedureName( Param1 [, ParamsN] )
```

```
INIT PROCEDURE SomeProcedureName
EXIT PROCEDURE SomeProcedureName
```

```
[STATIC] FUNCTION SomeProcedureName
[STATIC] FUNCTION SomeProcedureName()
[STATIC] FUNCTION SomeProcedureName( Param1 [, ParamsN] )
```

Procedures and Functions in Harbour can be specified with the keywords `PROCEDURE`, or `FUNCTION`. Naming rules are the same as those for *Variables* (up to 63 characters non-case sensitive). Both Procedures and Functions may be qualified by the scope qualifier *STATIC* to restrict their usage to the scope of the module where defined.

The *INIT* or *EXIT* optional qualifiers, will flag the procedure to be automatically invoked just before calling the application startup procedure, or just after quitting the application, respectively. Parameters passed to a procedure/function appear in the subroutine as local variables, and may accept any type, including references.

Changes to argument variables are not reflected in respective variables passed by the calling procedure/function/method unless explicitly passed BY REFERENCE using the @ prefix.

PROCEDURE has no return value, and if used in an Expression context will produce a *NIL* value.

FUNCTION may return any type by means of the RETURN statement, anywhere in the body of its definition.

An example procedure definition and a function call follows:

```
x := Cube( 2 )

FUNCTION Cube( n )
   RETURN n ** 3
```

Sample code

The typical "hello world" program would be:

```
? "Hello, world!"
```

Or:

```
QOut( "Hello, world!" )
```

Or:

```
Alert( "Hello, world!" )
```

Or, enclosed in an explicit procedure:

```
PROCEDURE Main()

   ? "Hello, world!"

   RETURN
```

OOP examples

Main procedure:

```
#include "hbclass.ch"

PROCEDURE Main()

   LOCAL oPerson
```

```
    CLS

    oPerson := Person():New( "Dave" )
    oPerson:Eyes := "Invalid"
    oPerson:Eyes := "Blue"

    Alert( oPerson:Describe() )

    RETURN
```

Class definition:

```
 CREATE CLASS Person

    VAR Name INIT ""

    METHOD New( cName )
    METHOD Describe()

    ACCESS Eyes INLINE ::pvtEyes
    ASSIGN Eyes( x ) INLINE iif( HB_ISSTRING( x ) .AND. x $ "Blue,Brown,Green", ::pvtEyes :=
x, Alert( "Invalid value" ) )

    PROTECTED:

    VAR pvtEyes

 ENDCLASS

 // Sample of normal Method definition
 METHOD New( cName ) CLASS Person

    ::Name := cName

    RETURN Self

 METHOD Describe() CLASS Person

    LOCAL cDescription

    IF Empty( ::Name )
       cDescription := "I have no name yet."
    ELSE
       cDescription := "My name is: " + ::Name + ";"
    ENDIF

    IF ! Empty( ::Eyes )
       cDescription += "my eyes' color is: " + ::Eyes
    ENDIF

    RETURN cDescription
```

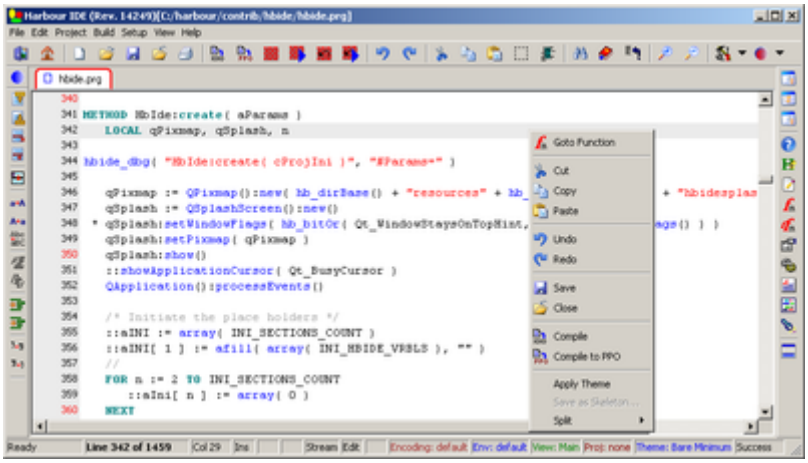
Tools

- hbmk2 – Build tool like make
- hbrun – Shell interpreter for Harbour. Macro compiling allows to run any valid Harbour code as it's being compiled
- hbformat – Formats source code written on Harbour or another dialect according to defined rules
- hbpp – Pre-processor, a powerful tool which avoids typical problems found on C language per-processor
- hbi18n – Tools to localizing text on applications
- hbdoc – Creates documentation for Harbour

All tools are multi-platform.

Development

Today Harbour development is led by Viktor Szakáts in collaboration with Przemysław Czerpak who also contributes many components of the core language and supplementary components. HBIDE and some other components, especially HBQt, are developed by Pritpal Bedi. Other members of the development community send changes to the GitHub source repository.[5] As of 2015 Harbour development is active and vibrant.



HBIDE look.

xHarbour comparison

xHarbour is a fork[6] of the earlier Harbour project. xHarbour takes a more aggressive approach to implementing new features in the language, while Harbour is more conservative in its approach, aiming first of all for an exact replication of Clipper behaviour and then implementing new features and extensions as a secondary consideration. It should also be noted that Harbour is supported on a wide variety of operating systems while xHarbour only really supports MS Windows and Linux 32-bit.

The Harbour developers have attempted to document all hidden behaviour in the Clipper language and test Harbour-compiled code alongside the same code compiled with Clipper to maintain compatibility.

The Harbour developers explicitly reject extensions to the language where those extensions would break Clipper compatibility. These rejections were softened recently since the new Harbour architecture allows extensions out of the core compiler.

A detailed comparison between extensions implemented in Harbour and xHarbour can be found in the source repository of the project on GitHub.[7]

GUI libraries and tools

- **hbide (https://sourceforge.net/projects/qtcontribs/)** – Integrated Development Environment to help Harbour development and various xBase dialects
- **PTSource IDE (https://ptsource.github.io/Developer-Platform/)** – Integrated Development Environment includes Harbour
- **HwGui (https://sourceforge.net/projects/hwgui/)** – Open Source cross-platform GUI library for Harbour
- **HMG (https://hmg.ruano.org/index.php)** – Free / Open Source xBase Win32 / GUI Development System for Harbour
- **MiniGUI (http://hmgextended.com/)**[8] – Free / Open Source xBase Win32 / GUI Development System (a Fork (software development) of both HMG and Harbour)
- **ooHG (https://oohg.github.io/)** – Object Oriented Harbour GUI – a fork "class based and oop programming" of HMG

- **Marinas-GUI (http://marinas-gui.org/)** – Multi-Platform QT Based GUI Development Package for Harbour. Marinas-GUI downloads as a complete installation package for the chosen target platform (IDE, Version Control, Harbour/C Compiler, Libraries etc.) – Basically install and start coding and compiling

See also

- Visual FoxPro
- Visual Objects
- Xbase++
- PWCT free open source visual programming language support Harbour through HarbourPWCT

References

1. "Harbour" (https://ecsoft2.org/harbour). Retrieved 3 September 2020.
2. "LetoDB" (https://sourceforge.net/projects/letodb). Sourceforge.net. Retrieved 9 December 2013.
3. "QtContribs - Harbour Qt Projects" (https://sourceforge.net/projects/qtcontribs/). *SourceForge*.
4. "Official Harbour page" (https://harbour.github.io/). The Harbour Project. Retrieved 9 December 2013.
5. "harbour 路 GitHub" (https://github.com/harbour). Github.com. Retrieved 9 December 2013.
6. "About xHarbour" (http://www.xharbour.org/index.asp?page=about/index). Xharbour.org. Retrieved 9 December 2013.
7. "xhb-diff.txt" (https://github.com/harbour/core/raw/master/doc/xhb-diff.txt). *GitHub*. Retrieved 9 December 2013.
8. vailtom (17 August 2009). "Harbour MiniGUI Extended Edition. | Free Communications software downloads at" (https://sourceforge.net/projects/hmgs-minigui/). Sourceforge.net. Retrieved 9 December 2013.

External links

- Official website (https://harbour.github.io)
- The Oasis (https://harbour.github.io/the-oasis/) Clipper, FoxPro and Xbase++ community repository
- HBIDE (http://hbide.vouch.info/)
- Harbour Developers Mailing List (https://groups.google.com/group/harbour-devel/)
- Harbour Users Mailing List (https://groups.google.com/group/harbour-users/)
- Extensive Harbour documentation, libraries, tools site (http://www.kresin.ru/en/harbour.html)
- Harbour Wiki (Harbour Functions Dictionary) (https://github.com/Petewg/harbour-core/wiki)