Hungarian Notation Revisited.

# Introduction

A little about me (Sylvain Larche)

Programming has always been part of my career and business I started with dBase III the switch to Clipper Summer 87 the all the other version except 5.3

I still maintain about 30 programs written in Clipper 5.2e using pure DOS or running on windows 98, XP and Windows 7. Some of them got converted in FiveWin for Clipper then FiveWin Harbour

Now I am starting to use MiniGui extended and I like it. It's IDE is nice and easy to use finally I deserve a break from manually coding windows screen.!!!

In 198? at the Ottawa Clipper User Group, David Morgon from Nantucket made a presentation of Clipper 5 what a revelation. At this moment in time and with the help of 3 others friend we started the Montreal Clipper User Group it was very successful and we had close to 300 active members (small Company, Government, Corporate and individual )

During this time we started to code and showed other the Clipper way and found a document about Hungarian notation by “Robert A. DiFalco”.

We talk about it in our monthly meeting and saw that it was a good way to write code and make us better code writer so we could share among our user base our “nice function”. We felt the need to make modification to better adapt to the power of the clipper language and now for the [x]Harbour language. We introduced a notion of SCOPING that makes thing even simpler to understand.

As the acting President/Vice-President of the User Group I assisted to almost every DEVCON by Nantucket and Computer Associated. I have learned a lot about Clipper and I was able to share this knowledge with all our member. So as a thank you for all these opportunities I am contributing to help as much as I can.

Here are the guideline we created and showed all our member via our BBS, newsletter, monthly meeting, seminar as well as onsite training.

Enjoy.

Here is the Chart:

Hungarian Notation Version – GUCM (Montreal Clipper User Group)

|  |  |
| --- | --- |
| Variable Scope | Variable Type |
| P = Parameter | A = Array |
| L = Local | B = Codeblock |
| S = Static | C = Character / String |
| V = private | D = Date |
| B = Public | H = Handle |
| G = Global | L = Logic |
| D = Define | M = Memo |
|  | N = Numeric |
|  | O = Object |
|  | P = Pointers |
|  | X = Undefined, Variable type(USUAL) |
|  | T = date Time |
|  | U = Undefined, Variable Type (USUAL) |

Someone said: If it was hard to write then it should be hard to read. (Probably a C programmer!!)

**That's was not our idea about coding.**

Coding is a bit like an art form and is a "living thing" in that it grows in very unpredictable ways. So that said the GUCM.LIB was developed by a few members and used ABSOLUTE guideline in order for all to be able to read/learned and bug fix each function as needed. These rules are no ABSOLUTE but we think it’s the best we can do.

# Rule #1

ALWAYS use Hungarian Notation for all function and variable naming  
 (exception of looping variable you can use x, y, z or i, j,k )

# Rule #2

Declare 1 variable per line, aligned them and document it

LOCAL lnAge AS NUMERIC := 10 // Patient Age

LOCAL lcFName AS STRING := “Sylvain” // Patient First Name

# Rule #3

CODE should always be easy to read with lots of comment to explain what, why and how

# Rule #4

Never use plural for a variable name this way you don't have to remember if it was declared with an "s" or not.

# Rule #5

Create variable and function name that are meaningful

# Rule #6

Always write command in UPPERCASE i.e. : DO WHILE, IF, ENDIF, USE…NEW ALIAS etc..

# Rule #7

All function/procedure in Upper Lower with some exception for function grouping

- All array function i.e.: aDel, aSort, aScan, aIns, aTail, aEval etc...

- All file function i.e.: fRead, fWrite, fOpen, fClose fSeek etc..

- All Database function i.e.: dbSeek, dbCloseArea, dbCommit, dbGoto, dbAppend etc..

- All String/Date function i.e.: cDoW, cMonth, cTod, dToS

# Rule #7

Use standard abbreviation i.e. : Scr = Screen, Crs = Cursor, Prn = Printer etc..

# Rule #8

Always assign type to your variable its part of the EXPLICIT model.

Updated in 1992 base on Steve Straley seminar at our Montreal DevCon to now write variable typing (even if it's not operational)

Note: Even if today [x]Harbour is not checking for Strong typing your code will be ready for it when it does.

## Example

LOCAL lcName AS STRING := "Montreal"

LOCAL lnCount AS NUMERIC := 100

PRIVATE vcName AS STRING := "Another string"

# Rule #9

Use Typing in Database fieldname so this way it's easy to remember

ie: lcName := Database->cName

lnAge := Database->nAge

These conventions take in the same factors as outlined by Charles Simonyi for creating names in a program. The factors listed below are directly quoted from his monograph.

Begin quote:

1. Mnemonic value - so that the programmer can remember the name.

2. Suggestive value - so that others can read the code easily.

3. "Consistency" - this is often viewed as an aesthetic idea, yet it also has to do with the information efficiency of the program text. Roughly speaking, we want similar names for similar quantities.

4. Speed of decision- we cannot spend too much time pondering the name of a single quantity, nor is there time for typing and editing extremely long variable names.

End quote:

# Rule #10

Always use EXPLICIT coding instead of implicit coding which is based the knowledge of the programmer and after a while it can fail.

# Rule #11

1. Using alias with everything i.e.: Customer->( dbAppend() ) or Customer->( MyFunction() )

2. Use NIL to pass unneeded parameter i.e.: MemoLine( cString, NIL, nLine ) // explicitly skip Param #2

# Rule #12

Always try to have 1 entry point and 1 EXIT point in a Function/Procedure

FUNCTION Test()

...

...

RETURN( \_Value\_ ) <-- only exit point

# Rule #13

Use BEGIN SEQUENCE END SEQUENCE for easier coding

## Example:

FUNCTION MyTest()

LOCAL llValue := .T.

BEGIN SEQUENCE

IF Condition1...

Break( 1 ) // Using Break() so you can pass a value to RECOVER USING

ENDIF // Otherwise use BREAK command if no value need to be passed

IF Condition2…

Break( 2 )

ENDIF

RECOVER USING x

llValue := .F.

DO CASE

CASE x == 1 ; Alert( "Missing ..." )

CASE x == 2 ; Alert( "Invalid ..." )

etc..

ENDCASE

END SEQUENCE

RETURN( llValue )

# Rule #14

Always use := to assign a value to a variable it's easier to read

## Example

IF ( npos = Val( cChar ) ) = 65 <-- Hard to read

IF ( nPos := Val( cChar ) ) == 65 <-- Easier to read

# Rule #15

Always use == instead of = since it can be used as an assign operator this will make code more EXPLICIT also = is not the same as == check SET EXACT command to understand full behavior

# Rule #16

NEVER USE short form of command or Function it's harder to read because it's base on IMPLICIT information we need to know that. It’s a bad idea imported from the dBase iii era down to Clipper and now [x]Harbour. I wished it would be taken away forever.

## Example

FUNC = FUNCTION

RETU = RETURN

SUBS = SubStr()

REPL = ( Replicate() or REPLACE ) ? that why no shortcut should be used

# Rule #17

Align your :=

Align tour REPLACE ... WITH ... for easier reading (1 replace … with …, … with .. is not faster)

Align your #DEFINE

Align all other code that would be clearer when aligned in a meaningful way

# Rule #18

Comment your function and document them using a STANDARD Function header.

## Example

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Function....: MySpecialGuizmo

\* Description.: Long description of the function and it's usage

\*

\* Parameter...: pcLine AS STRING : Description ...

\* pcDelim AS STRING : Description ...

\* Author........: Sylvain Larche

\* Date............: 8/12/2011 10:45:45 PM

\* Revision …...: 1.00

\* 1.01 Fixed bug in pcDelim processing etc..

\* Todo...........: Description of things to do...

\* Status.........: Working, Failing, Bugged, etc..

\* Copyright….: (c) Sylvain Larche 2011

\* Usage..........: Example of code using the function

\* 🡪 Please follow clipper standard for this (read your norton guide !!! )

\*--------------------------------------------------------------

FUNCTION MySpecialGuizmo( ; // Short Description of function

pcLine AS STRING, ; // Desc. of pcLine

pcDelim AS STRING ; // Desc. of pcDelim

) AS STRING // Desc. of return value

\*-------------------------------------------

RETURN( lcValue )

NOTE : You can also use the nanforum (LIBNF) FT\_DOC template which can create doc for your program or library

# Rule #19

Always do parameter checking in your function and set them to default value if they are optional

## Example

DEFAULT pcDelim TO "," // <--- This COMMAND is declared in Common.ch

DEFAULT pnLen TO 10 // 🡨 use the following xCommand and insert it in your project .CH file

// Checking 1st param...

IF ...

ENDIF

**NOTE 1 : I personnaly use this version of the DEFAULT TO command**

#xcommand DEFAULT <v1> TO <x1> [, <vn> TO <xn> ] ;

=> ;

IF <v1> == NIL .OR. ValType(<v1>) # ValType(<x1>); <v1> := <x1> ; END ;

[; IF <vn> == NIL .OR. ValType(<vn>) # ValType(<xn>) ; <vn> := <xn> ; END ]

**NOTE 2 : like Variable declaration write one DEFAULT TO per line**

# Rule #20

Use special paragraph line to separate part of your code it create area well define and easy to follow.

## Example

\*--------------------------------------------

FUNCTION Test( ; // Short Desc of function

pcParam1 AS STRING, ; // Desc

)

\*--------------------------------------------

LOCAL lnValue AS NUMERIC := 10

\*- - - - - - - - - - - - - - - - - - - - - -

// Parameter Checking

...

\*- - - - - - - - - - - - - - - - - - - - - -

// Load value from database

\*- - - - - - - - - - - - - - - - - - - - - -

// Process info

RETURN( lnValue )

# Rule #21

Use "" to define string instead of '' (single quote) or [] for consistency

# Rule #22

Always put an OTHERWISE clause in your DO CASE... you never know...

# Rule #23

Use short DO CASE if possible, it’s a must for the eyes

## Example

DO CASE

CASE lcVar == "C" ; lnReturn := 1

CASE lcVar == "D" ; lnReturn := 2

CASE lcVar == "E" ; lnReturn := 3

OTHERWISE ; lnReturn := -1

ENDCASE

# Rule #24

Use negative -1 to return Error state from your function call if possible

## Example

IF MyFunc( 1, 2, 3 ) <> -1 // An Error was detected ?

// The function returned no error

ELSE

// An ERROR was return, so take care of it

ENDIF

# Rule #25

The compiler is your friend use it and get warning for all undeclared variables “/W[n]”

# Rule #26

Use Include file for MEMVAR declaration for Public and Private variable

#INCLUDE "AppsVar.ch" // List of public and Private variable

FUNCTION Abc()

LOCAL...

code...

bcPrnConfig := ... // <-- This variable might be declared in another

// program so to avoid error using /W compiler switch

// include a .CH file.

RETURN( ... )

-> File AppsVar.ch

MEMVAR bcPrnConfig // Printer Config File (default to hp.dat)

MEMVAR bnPrnOrientation // Printer Orientation (1=Portrait 2= Landscape)

MEMVAR bnPrnCopie // Printer Page Copy (default to 1)

etc..

# Rule #27

When creating a function try to make them as "generic" as possible and independent from the other code try not to use public/private variable if possible.

# Rule #28

Used specific Tone/sound for specific error within your application

i.e.: Tone(300,3) for not found error

Tone(500,2) for invalid value etc..

User will understand your CUI/GUI better

# Rule #29

Use standard interface in your code, use Alert() Msgbox(), MsgError() . Make interface similar for each form so "S" is search in the entire program.

# Rule #30

Use Array to store common information or CLASS\*(Clipper with Class(y) or [x]Harbour)

## Example

PRIVATE vaApps := Array( 5 )

where vaApps[ 1 ] == "Application Name"

vaApps[ 2 ] == "c:\AppFolder"

vaApps[ 3 ] == etc...

vaApps[ 4 ] == etc...

vaApps[ 5 ] == etc...

\*Class definition can vary please read about it.

CLASS Apps

DATA cAppName

DATA cAppFolder

END CLASS

? Apps:cAppName + " " + Apps:cAppFolder

NOTE: Use Define to access Array

#DEFINE dnAppName 1

#DEFINE dnAppFolder 2

etc ...

Then use vaApps[ dnAppName ] to access it, it's easier/safer and if you add a lot of information to this array you can easily change the order without affecting the program. Just be sure to recompile all source if those define are in a .CH file or at least be part of your dependent list for that .PRG file

# Rule #31

Try to access multidimensional array using aTest[ i, j ] instead of aTest[i][j] it's easier to read but that's your call

# Rule #32

When doing loop (DO WHILE or FOR NEXT ) try to use a variable in the evaluation if possible, reevaluating a LastRec() can slow down your program when processing large record count.

## Example

FOR i := 1 TO Customer->( LastRec() )

NEXT

use :

lnNbCust := Customer->( LastRec() )

FOR i := 1 TO lnNbCust

NEXT

# Rule #33

DO NOT OPEN/CLOSE database file each time your call a module

Use something like OpenFiles() when starting the program and CloseFiles() when exiting the program. This has also the benefit of grouping the Open/Close together. If there is too many database then Open/Close by type

## Example : OpenModuleGL() / CloseModuleGL()

# Rule #34

USE DBFCDX rdd it's easier to maintain code for index when you have more than 1 program accessing the same database

## Example :

Program1.EXE -> Need Cust.dbf and Index Tag CustByName and CustByCode

Program2.EXE -> Need Cust.dbf and Index Tag CustByName and CustByNumber

Since all Tag are store in the CDX file you don't have to update both program to open and synchronize all INDEXES at the same time

NOTE: Use CDX with same Name as DBF if possible so it open automatically ( Cust.dbf and Cust.cdx )

# Rule #35

USE ".FPT" memo file if you need them in your apps it's much safer when it grows bigger -> 10 Meg+

# Rule #36

Always write NIL as parameter to a function if skipping an optional parameter this is the EXPLICIT coding, this way it's easier to see skipped parameter compare this.

## Example

ORIGINAL --> MyFunc( pcParam1, pnParam2, pcParam3,, pcParam5 ) <-- pcParam4 is optional

EXPLICIT --> MyFunc( pcParam1, pnParam2, pcParam3, NIL, pcParam5 ) <-- pcParam4 is optional

# Rule #37

Always use iif() since if() can be easily confuse in cases like this

IF ( laTable[ 1 ] == "Test" .AND. laIndex[ 1 ] == "Test01" )

...

ENDIF

If ( CheckThis(), SetProperty(…), SetProperty(…) ) 🡨- This is a iif() call

# Rule #38

Always use EXPLICIT ALIAS in your program it’s safer if you have multiple field with the same name and as a bonus you don’t need all those “Select( xxx)” statement.

## Example

FUNCTION SaveRec( pnCodeID )

// Check if we have

IF ! Cust->( dbSeek( pnCodeID ) )

// Customer ID Found

Cust->( dbAppend( .T. ) ) // Append new record

ELSE

Cust->( RLock() ) // Lock Record

ENDIF

REPLACE Cust->nCustID WITH pnCodeID

REPLACE Cust->cName WITH vcName

REPLACE Cust->cAddress WITH vcAddress

...

Cust->( dbCommit() )

Cust->( dbUnLock() )

RETURN( NIL )

# Rule #39

Avoid & "Macro" if possible, there is a lot of case where they are avoidable

## Example 1

FUNCTION CheckIfSomethingExist( pcAlias, pxValue )

LOCAL llValid AS LOGICAL := .F.

llValid := ( pcAlias )->( dbSeek( pxValue ) )

RETURN( llValid )

## Example 2

SELECT &MyAlias --> SELECT (MyAlias)

## Example 3

MyField\_&n := 10 --> ( "MyField\_" + AllTrim( Str( n ) ) ) := 10

**NOTE : take a look at function Field\*()**

# Rule #40

It's not because it's a test program that is should not be documented many time such test become program utilities.

# Rule #41

When documenting your code think! "What do I need to know about this" in 2 years

# Rule #42

Document Library and batch file

# Rule #43

Codeblock are your friend use them wisely, they are fast and very versatile

FUNCTION Test()

LOCAL lbCond := { | cID | Cust->nCustID == cID }

IF Eval( lbCond )

// Condition matched

ELSE

// Did not fit in...

ENDIF

RETURN( NIL )

# Rule #44

Use Comment marker to help in optimizing/documenting etc..

## Example

Here are some we used in our dev. team

//? ... <-- Unfinished and need answer

//OPT ... <-- Need Optimization

//TBC ... <-- Too Be Completed "function under development"

# Rule #45

Always use EXPLICIT K\_\* not their numeric value (Add #Include “inkey.ch” to your .prg)

## Example

IF nKey == 27 --> IF nKey == K\_ESC

# Rule #46

Try to always use the ELSE / ELSEIF to deal with unexpected value

# Rule #47

Always use ERROR CHECKING function so your program can be ROCK SOLID this is the key to successful software. (Anything to make the customer happy!!)

# Rule #48

Use Logging function to Log pertinent information about program status, process and state this help when debugging. I like it when a client calls and say to me your program crash but it's not my fault, and then looking at the log file I prove him wrong!!!

## Example

TLog( "Stating Report : Print Client", "From: " + lcFrom, "To: " + lcTo )

TLog( "Processing Error 1" )

etc..

**NOTE: Creating an Apps.Log file with all these information’s that are quite useful for recreating the bug or to help finding it.**

# Rule #49

Use Database Stamping information it's useful for many thing like database replication, transaction surveillance etc..

## Example

Simple Stamping:

Add 3 fields to any database you want stamped!

dStamp AS DATE // Date()

tStamp AS STRING // Time()

cStamp AS STRING // UserID()

Complex Stamping:

Add needed fields.

-> For when Adding a record

dStampAdd

tStampAdd

cStampAdd

-> For when Updating a record

dStampUpd

tStampUpd

cStampUpd

-> For when viewing a record (when dealing with confidential records)

dStampView

tStampView

cStampView

(Deleting, Printing, And Closing (Claim) etc... it up to your need and imagination

**NOTE 1: HBSIX offer some SX\_SetTrigger() that can be useful.**

**NOTE 2: Add another database for keeping multiple Stamping info (log of all access)**

# Rule #50

When creating report use some type of PREVIEW or print to file option. Useful to customer to save paper printout when all they needed was to consult the report.

**NOTE: Now in windows it makes sense even more.**

# Rule #51

Maintain a Changlog file for all new/fixed informations (if using a CVS program this can be skipped or complement it)

**NOTE : see Point #7 in second section (project Management)**

# Rule #52

In the Montreal Clipper User Group we created what we call:

UDFC or "U"ser "D"efine "F"unction "C"ommand to help in parameter parsing in complex function that we did not want to be a COMMAND they add the advantage to be ALIASable

## Example

Cust->( PicTb( ... ) )

\*-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

#xTRANSLATE PickTB( AT <nTop> , <nLeft> ;

TO <nBottom>, <nRight> ;

USING <aCol> ;

[ TITLE <cTitle> ] ;

[ WHILE [ TOP <cWhileTop> ] ;

[ BOTTOM <cWhileBot> ] ;

] ;

[ KEY HANDLER <cKeyHandler> ] ;

[ DISPLAY <cDisplay> ] ;

[ <lIncr:INCREMENTAL> ] ;

) ;

=> ;

( ;

PickTB( <nTop>, <nLeft>, <nBottom>, <nRight>, ;

<aCol>, ;

<cTitle>, ;

{ | poTB, pnKey | <cKeyHandler>( poTB, pnKey ) }, ;

{ | poTB, pnKey | <cDisplay>( poTB, pnKey ) }, ;

<cWhileTop>, ;

<cWhileBot>, ;

<.lIncr.> ;

) ;

)

# Rule #53

Try avoiding to Name a Function, Procedure or Variable to an existing name within the [x]Harbour domain to avoid confusion and perhaps an “unspecified” side effect.

## Example :

LOCAL Color := “BG+” // Variable Color is using a know keyword

Use cColor or lcColor instead

FUNCTION Alert() …

Use FUNCTION MyAlert() instead

# Rule #54

Try avoiding the underscore “\_” in Variable Name and function/procedure use UpperLower instead it’s much easier to read and it shoud be used for INTERNAL struff only.

## Example :

FUNCTION Add\_Record() should be FUNCTION AddRecord()

Project management 101

Always put a readme.\* file in the program folder with project info containing detail like:

1. Version of compiler used: Clipper 5.2e

2. Version of linker used: Blinker 3.1

3. Commercial Library used (Name, version)

4. Internal Library used (Name, version )

5. List of program file and a brief description for each of them

6. Create a Project structure

--+ SoftWare1

--+ Backup

--+ Bin

--+ Doc

--+ LibBackup ( Contain a ZIP file of the Library and all its files )

--+ Database

--+ Source

--+ Include

--+ Lib

--+ Contract (Contain scan: contract, and document exchanged with customer)

When dealing with many program for a specific client adapt the structure accordingly.

--+ ClientName

--+ Software 1

--+ Backup

--+ Bin

--+ Doc

--+ Database

--+ Lib (specific to Software1 )

--+ Source

--+ Source

--+ Include

--+ Contract (Contain scan: contract, and document exchanged with customer)

--+ Software 2

etc...

--+ LibBackup (Contain a ZIP file of the Library and all its files)

--+ Lib (Contain Common library)

- SourceLib1

- SourceLib2

7. Use a Version Control Software like Microsoft SourceSafe to maintain revision to your software. Other CVS program includes Tortoise CVS / Tortoise SVN.

8. Use a good editor like MultiEdit or Brief/dBrief it’s specially configured for Clipper/Harbour usage.

**NOTE: Used HB-IDE, xMate or HMGS-IDE**

9. FINALY: BACKUP BACKUP BACKUP and BACKUP...

Code safety is important, loosing code is frustrating and you know like me that doing it again you always forget something you did before.

Exit / Quit

In my programming years I have followed these guidelines and I can truly say that it made my life easier especially when I had to work on an old project for a client. Getting back and fixing or adding new functionality was much easier this way.

When writing a program for a customer I know the pressure of doing it faster and faster but when I take the time to explain my way of doing things they understand that later in the life of the program that work will pay itself off by loosing less time getting back in the "project".

Another thing is that doing it like this help me do LESS coding error and allows me to find stupid bug faster. My programs are less prone to bug/crash since coding them like that.

This Hungarian Notation doc was base of work of:

Robert A. Difalco

Fresh Technologies

Special Thanks to Tom Leylan and Matt Maier

Updated by Sylvain Larche for the (Montreal Clipper User Group)

Updated by Sylvain Larche for the [x]Harbour version (work in progress)

## EXTRA NOTE:

If you want to see Hungarian at is best stay tune and look at the new upcoming version of HMGS-IDE

Hope this will inspire you to construct quality code and to share the joy of programming.

Sylvain Larche

Iosys Technologie Inc.