eBook Gratuit

APPRENEZ visual-foxpro

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#visual-foxpro

Table des matières

À propos 1

Chapitre 1: Démarrer avec visual-foxpro 2

Remarques 2

Versions 2

Examples 2

Installation ou configuration 3

Bonjour le monde 3

Ajouter un gestionnaire d'erreur global 3

Chapitre 2: Interopérabilité VFP avec .NET 5

Introduction 5

Examples 5

Utilisation de wwDotNetBridge pour exécuter du code .NET 5

Chapitre 3: Les opérateurs 7

Remarques 7

Examples 7

Opérateurs numériques 7

Opérateurs logiques 8

Opérateurs de caractères 9

Opérateurs de date et d'heure 12

Opérateurs relationnels 14

Crédits 19

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: visual-foxpro

It is an unofficial and free visual-foxpro ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official visual-foxpro.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec visual-foxpro

Remarques

Foxpro a été créé au début des années 80 (à l'origine sous le nom FoxBase - 1984?) Par le logiciel Fox et pris en charge sur les plates-formes Mac OS, Unix, DOS et Windows. Il était alors connu comme le moteur de base de données le plus rapide sur PC. Plus tard en 1992, *malheureusement* , il a été acquis par Microsoft. Après la reprise de Microsoft, Foxpro pour DOS (FPD) et Foxpro pour Windows (FPW) 2.6 ont été lancés en 1994. À la fin de 1995, Foxpro a reçu le nom de "Visual" et le support de la plate-forme était limité à Windows. C'était aussi la première version de Foxpro où le langage était orienté objet.

Le site officiel Visual Foxpro de Microsoft (communément appelé simplement VFP) le décrit comme:

> Le système de développement de bases de données Microsoft® Visual FoxPro® est un outil puissant pour créer rapidement des applications de base de données de bureau, client enrichi, client distribué, client / serveur et Web hautes performances.

Bien qu'il s'agisse d'un ancien langage, il est toujours considéré comme le langage le plus simple pour créer rapidement une application centrée sur les données pour le bureau Windows. **Si** vous avez besoin de créer ***une application basée sur les données pour Windows Desktop,*** puis de choisir VFP, vous le ferez vraiment facilement et rapidement.

Versions

Version	Libéré
FPW 2.6a	1994-10-28
Visual Foxpro 3.0	1995-12-16
Visual Foxpro 5.0	1997-01-24
Visual Foxpro 6.0	2000-08-18
Visual Foxpro 7.0	2002-01-04
Visual Foxpro 8.0	2003-10-25
Visual Foxpro 9.0	2004-12-13
Visual Foxpro 9.0 SP2	2007-10-21

Examples

Installation ou configuration

Instructions détaillées sur l'installation ou l'installation de visual-foxpro.

Bonjour le monde

Traditionnellement, dans toutes les langues, le premier exemple est "Hello World". Probablement faire cela est plus facile dans VFP:

```
? "Hello World"
```

Ajouter un gestionnaire d'erreur global

Une méthode simple pour détecter les erreurs non gérées (exceptions) dans une application VFP consiste à utiliser la commande ON ERROR au début de votre programme principal.

La commande ON ERROR suivante appelle une méthode du programme en cours appelée "errorHandler". Les valeurs renvoyées par ERROR (le numéro d'erreur VFP), MESSAGE (message d'erreur VFP), PROGRAM (nom du programme en cours d'exécution) et LINENO (numéro de ligne de l'erreur) sont transmises à la méthode errorHandler.

```
ON ERROR DO errorHandler WITH ERROR(), MESSAGE(), PROGRAM(), LINENO()
```

Une simple méthode errorHandler peut ressembler à ceci:

```
PROCEDURE errorHandler
    LPARAMETERS tnVFPErrorNumber, tcVFPErrorMessage, tcProcWithError, tnLineNumber

    STORE 'Error message: ' + tcVFPErrorMessage + CHR(13) + ;
        'Error number: ' + TRANSFORM(tnVFPErrorNumber) + CHR(13) + ;
        'Procedure with error: ' + tcProcWithError + CHR(13) + ;
        'Line number of error: ' + TRANSFORM(tnLineNumber) TO lcDetails

    MESSAGEBOX(lcDetails, 16, "Unhandled Exception")

    ON ERROR *
    ON SHUTDOWN
    CLEAR EVENTS

    QUIT
ENDPROC
```

Vous pouvez également modifier et restaurer le gestionnaire d'erreurs entre les deux. Par exemple, à un moment donné, vous voulez ouvrir toutes les tables d'un dossier exclusivement, et si vous ne le pouvez pas, vous ne voulez pas continuer:

```
procedure DoSomethingWithExclusiveLock(tcFolder)
local lcOldError, llInUse, ix && by default these variables have a value of .F.
lcError = on('error') && save current handler
on error llInUse = .T.  && new handler
local array laTables[1]
```

```
for ix=1 to adir(laTables, addbs(m.tcFolder) + '*.dbf'))
   use (addbs(m.tcFolder)+laTables[m.ix,1]) in 0 exclusive
endfor
on error &lcError && restore old handler
if m.llInUse && couldn't get exclusive lock on all tables
   close databases all
   return
endif
* do whatever
endproc
```

Astuce: Parfois, en particulier pendant le débogage, vous souhaitez restaurer le gestionnaire d'erreur par défaut, ce qui vous permet de pénétrer dans le code où l'erreur s'est produite, puis de l'ajouter à l'endroit, en ajoutant temporairement:

```
on error
```

ferait cela.

Lire Démarrer avec visual-foxpro en ligne: https://riptutorial.com/fr/visual-foxpro/topic/7391/demarrer-avec-visual-foxpro

Chapitre 2: Interopérabilité VFP avec .NET

Introduction

Cette rubrique couvrira l'interopérabilité entre VFP et .NET.

Examples

Utilisation de wwDotNetBridge pour exécuter du code .NET

Avec l'aide de wwDotNetBridge de West Wind , vous pouvez facilement accéder au code .NET dans un programme VFP.

Le livre blanc contient tous les détails, mais cet exemple concis aidera à illustrer les étapes de base de l'exécution d'une méthode dans un assembly .NET.

Notez que wwDotNetBridge peut accéder directement à des propriétés simples telles que les chaînes, les ints, etc. bas de cet exemple).

```
*!* Load WestWind .NET wrapper library (wwdotnetbridge.prg assumed to be in the search path)
IF (!wwDotNetBridge())
    RETURN .F.
ENDIF

lowwDotNetBridge = CREATEOBJECT("wwDotNetBridge","V4")

*!* Load .NET Assembly (include full or relative path if necessary)
IF !lowwDotNetBridge.LoadAssembly("SomeDotNetAssembly.dll")
    lcAssemblyLoadError = "LoadAssembly error: " + lowwDotNetBridge.cErrorMsg
    =MESSAGEBOX(lcAssemblyLoadError, MB_ICONSTOP, "Error")
    RETURN .F.
ENDIF

*!* Parameters to pass to class constructor
*!* You can pass up to 5 paramenters to the constructor
lcParameter1 = "StringParameter1"
lcParameter2 = "StringParameter2"
lnParameter3 = 3
lcParameter4 = .NULL.

*!* Get an instance of the assembly class
loAssemblyReference = lowwDotNetBridge.CreateInstance("MyDotNetProject.MyDotNetClass", ;
    lcParameter1, lcParameter2, lnParameter3, lcParameter4)
IF lowwDotNetBridge.lError
    lcAssemblyLoadError = "An error occurred loading the class: " + lowwDotNetBridge.cErrorMsg
    RETURN .F.
ENDIF

*!* Usage Example

*!* This example runs a method that return a boolean
*!* and populates a List<string> (SomeStringList).
*!*
```

```
*!* The assembly has a public property named "LastErrorMessage"
*!* with details about any handled exceptions/problems.

IF (!loAssemblyReference.SomePublicMethod())
    msg = "There was a problem executing the method:" + CRLF + ;
        loAssemblyReference.LastErrorMessage
    =MESSAGEBOX(msg, MB_ICONSTOP, "Error")
    RETURN .F.
ENDIF

*!* At this point the string list (SomeStringList) should be populated
*!* wwDotNetBridge can convert that list to a VFP COM array (0-based)

laVFPArrayOfStrings = lowwDotNetBridge.CreateArray()
laVFPArrayOfStrings.FromEnumerable(loAssemblyReference.SomeStringList)

FOR x = 0 TO laVFPArrayOfStrings.Count-1
    ? laVFPArrayOfStrings.Item(x)
ENDFOR
```

Lire Interopérabilité VFP avec .NET en ligne: https://riptutorial.com/fr/visual-foxpro/topic/9390/interoperabilite-vfp-avec--net

Chapitre 3: Les opérateurs

Remarques

Dans VFP, les opérateurs sont regroupés dans ceux-ci:

- Opérateurs numériques
- Opérateurs logiques
- Opérateurs de personnages
- Opérateurs de date et d'heure
- Opérateurs relationnels

Il y a aussi des opérateurs, implémentés en tant que fonctions (telles que les opérations binaires, la comparaison d'objets ...).

Nous allons examiner chacun par exemple.

Examples

Opérateurs numériques

Les opérateurs numériques sont les plus faciles et presque identiques aux autres langues.

- +, -, * et /. Opérateurs d'addition, de soustraction, de multiplication et de division (dans VFP, il n'y a pas de division entière, vous pouvez convertir un résultat en entier avec les fonctions INT (), CEILING () et FLOOR ()).
- Opérateur de module%.
- ^ et **. Puissance de l'opérateur (s). Ils font tous deux la même chose.
- (). Grouper les opérateurs.
- Les opérateurs ont priorité. La commande est:

```
( )
^ (or **)
/ and *
- and +
```

```
? 10 / 5 + 2 && Outputs 4
? 2 + 10 / 5 && Outputs 4 as well. Division has precedence.

* Both multiplication and division have same precedence
* They would be interpreted from left to right.
? 4 * 5 / 2 + 5 && Outputs 15
* Use parentheses whenever you are in doubt or want to be explicit
? ( (4 * 5) / 2 ) + 5 && Outputs 15. Explicit grouping of operations

? 4 * 5^2 && ^ has precedence, this is same as 4 * (5^2) = 100.
? (4 + 5)^2 && Using parentheses we say add 5 to 4 (9) and then square = 81.
```

Opérateurs logiques

Les opérateurs logiques dans VFP, dans leur ordre de priorité sont les suivants:

Opérateur	La description
()	Parenthèses, expressions de groupes
NE PAS, !	Refuser logiquement l'expression. PAS ou! n'a pas de différence.
ET	Logiquement ET les expressions
OU	Logiquement OU les expressions
<>,! =, #	Vérifiez les inégalités. Ainsi, même logique que OU exclusif - XOR

Historiquement, NOT, AND, OR sont écrits sous la forme .NOT., .AND., .OR. Vous pouvez toujours les utiliser si vous le souhaitez, mais AND, OR, NOT est plus simple et plus propre.

Pour faux et vrai, vous devez utiliser .F. et T. littéraux respectivement. Vous ne pouvez pas choisir d'utiliser F et T à la place.

```
* Some logical variables
local llOld, llEmail  && any variable declaration implicitly initializes the variable as .F. -
false
? m.llOld, m.llEmail && Prints .F. .F.

llOld   = .T.
llEmail = .F.

if ( m.llOld AND m.llEmail )
   ? 'Old AND should be emailed to'
endif
if ( m.llOld OR m.llEmail )
   ? 'Old OR should be emailed to'
endif
if ( m.llOld AND !m.llEmail ) && Same as (m.llOld AND NOT m.llEmail)
   ? 'Old BUT should NOT be emailed to'
endif

* Above code outputs
Old OR should be emailed to
Old BUT should NOT be emailed to
```

Dans VFP, les expressions logiques sont évaluées de manière raccourcie. C'est-à-dire que si la première partie de la vérification satisfait tout le résultat, le reste de l'expression n'est même **pas** interprété. Un échantillon suit:

```
? 1 = '2' && An obvious error. It would complain operator/operand type mismatch.

* However we could use such an expression in an if and get no error
* because it is not interpreted at all
```

```
* (VFP is dynamic and there is no compile time check)

local llProcess
llProcess = .T.

if (m.llProcess OR (1='2'))
   ? 'Should do processing'
endif

* Would output

Should do processing

* without any error because m.llProcess true means
* the whole expression would be true, thus the expression after OR
* is not interpreted at all.
```

Un piège qui attire les débutants est que, parfois, vous pouvez avoir besoin de plusieurs vérifications, par exemple dans une requête SQL, qui sont connectées avec des opérateurs AND ou OR. Quand il y en a beaucoup, on peut ignorer le fait que les opérateurs ont une priorité (dans l'ordre (), NOT, AND, OR) et pensent que l'interprétation serait faite de gauche à droite dans une chaîne. Considérons un échantillon:

```
select * from myTable where !isCustomer AND debit > 5000 OR discount > 5
```

quelle est l'intention de cette requête? Si nous le rendons explicite en regroupant les parenthèses, il est dit:

```
((NOT isCustomer) AND debit > 5000) OR discount > 5
```

simplifié, il ressemble à "firstExpression" OU (remise> 5). Quelle que soit l'intention, à cause de cela OU il sélectionnerait:

> toutes les lignes qui ont (discount> 5) - et aussi celles où il est client avec plus de 5000 débit.

L'intention était probablement de "me donner ceux où ce n'est PAS un client ET (le débit est supérieur à 5000 OU la remise est supérieure à 5)". Il serait clair dès le départ si nous utilisions des parenthèses:

```
select * from myTable where !isCustomer AND (debit > 5000 OR discount >
5)
```

Vous pouvez utiliser mais ne vaut pas la peine d'avoir des parenthèses pour l'opérateur NOT initial, lorsque son opérande est une expression unique et suffisamment lisible avec sa priorité -! IsCustomer est clairement lu comme (NOT isCustomer).

Opérateurs de caractères

Il n'y a que 4 opérateurs de caractères, dans leur ordre de priorité:

Opérateur	La description
()	Parenthèses pour le regroupement. Note: La documentation de VFP, que j'ai, manque celle-ci. Sans cela, l'opérateur est presque toujours inutile.
+	Concatène (joint) les chaînes côte à côte.
-	Concatène les chaînes en **déplaçant** les espaces de fin de chaîne de gauche à la fin de la chaîne de droite.
$	Vérifie si la première chaîne est contenue dans la seconde.

+ est le plus simple et permet également de concaténer des chaînes dans de nombreuses autres langues.

```
local firstName, lastName
firstName = "John"
lastName  = "Smith"

? m.firstName + " " + m.lastName
```

Résultats: John Smith

- est un peu difficile et peu connu. Il prend les espaces de fin de la chaîne de gauche, ajoute ces espaces à la chaîne de droite. Supposons que vous ayez une table avec le prénom et le nom de famille et que chacun d'eux comporte 20 caractères. Nous voulons concaténer les noms et prénoms pour créer un nom complet, **et** nous voulons également que la taille résultante soit fixe (dans ce cas 20 + 20 + 1 espace = 41). Faisons en sorte que vous ayez aussi une colonne de deuxième prénom et que vous souhaitiez que le nom complet ressemble à "lastName, firstName middleName_____". Il est plus facile de faire cela en utilisant - operator mais vous devriez noter l'astuce consistant à utiliser ici des parenthèses pour le regroupement afin d'obtenir exactement ce que nous voulons:

```
* Create a cursor for our sample and fill in a few names
Create Cursor Names (firstName c(20), midName c(20), lastName c(20))

Insert Into Names (firstName, midName, lastName) Values ('Cetin','', 'Basoz')
Insert Into Names (firstName, midName, lastName) Values ('John', 'M', 'Smith')
Insert Into Names (firstName, midName, lastName) Values ('John', 'F', 'Kennedy')
Insert Into Names (firstName, midName, lastName) Values ('Tom', '', 'Hanks')

* Select with tricky - operator
Select *, ;
    lastName - (', '+firstName-(' '+midName)) As FullName ;
    from Names ;
    INTO Cursor crsNames ;
    nofilter

Browse
```

Et la sortie est comme ça:

Prénom	midName	nom de famille	nom complet
Cetin		Basoz	Basoz, Cetin
John	M	Forgeron	Smith, John M
John	F	Kennedy	Kennedy, John F
À M		Hanks	Hanks, Tom

Dans la colonne fullName, tous les espaces à la fin sont bien poussés à la fin. Si vous vérifiez la structure, la colonne fullName a une largeur de 63 caractères (3 * 20 + 3 caractères ajoutés).

> Notez l'importance de regrouper les parenthèses (essayez de supprimer les parenthèses ou d'organiser différemment).

Bien que - l'opérateur puisse être tentant d'utiliser dans de tels cas, il y a un autre côté de la médaille. Cet opérateur est spécifique à VFP et le SQL n'est donc pas portable. Vous pouvez obtenir le même résultat avec ce SQL compatible ANSI:

```
Select *, ;
    CAST(RTRIM(lastName) +', '+ RTRIM(firstName) +' '+ midName as char(63)) As FullName ;
    from Names ;
    INTO Cursor crsNames ;
    nofilter
```

Le dernier opérateur est $. Il vérifie simplement si la chaîne de gauche fait partie de la chaîne de droite.

```
local upcased, digits, hexDigits
upcased = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
digits  = '0123456789'
hexDigits  = m.digits + 'ABCDEF'

? 'A' $ m.upcased && .T.
? 'a' $ m.upcased && .F.
? '1' $ m.digits && .T.
? 'F' $ m.digits && .F.
? 'F' $ m.hexDigits && .T.
```

> **Important:** Dans VFP, bien que vous puissiez écrire votre code dans tous les cas (supérieur, inférieur ou mixte), les chaînes sont toujours sensibles à la casse. Par exemple: "Smith" et "smith" sont deux valeurs distinctes. Ou dans votre tableau s'il y a une colonne de pays, vous ne trouverez pas «USA» si vous le recherchez avec «usa». Même chose avec l'opérateur $, "GE" $ "Germany" est faux.
>
> Note personnelle: Bien que vous puissiez aimer $ pour sa simplicité et que vous le trouverez souvent utilisé dans les codes sources de Microsoft, IMHO n’a que très peu de valeur. En pensant à beaucoup de milliers de lignes que j'ai écrites dans mon

opérateur, je pense que je trouverais très peu d'occasions dans mon propre code. Presque toujours, il y a une meilleure alternative (en particulier lorsque l'opérande gauche n'est pas un caractère unique et que la sensibilité à la casse est importante).

Opérateurs de date et d'heure

Il existe essentiellement deux opérateurs pour les valeurs de date et d'heure de date. + et - sont surchargés (probablement un terme C) pour faire du calcul date / heure:

Opérateur	La description
+	Ajoute des jours (date) ou des secondes (date / heure) à une valeur date / date / heure.
-	Obtient la différence de deux valeurs de date / date / heure. Soustrait les jours (date) ou les secondes (date / heure) des valeurs de date / heure.

+ est le plus facile. Il a deux opérandes, l'un est une valeur de date ou d'heure et l'autre est une valeur numérique (bien que vous puissiez utiliser n'importe quel chiffre, c'est un entier à toutes fins pratiques).

Quand l'un des opérandes est une date, alors l'opérande numérique est considéré comme "jour":

```
? Date() + 10 && Get the date 10 days later
* VFP is leap year aware when doing date math
? Date(2016, 2, 28) + 1 && Add 1 day to Feb 28, 2016. Returns Feb 29, 2016.
? Date(2017, 2, 28) + 1 && Add 1 day to Feb 28, 2017. Returns Mar 1, 2017.
```

Lorsqu'un des opérandes est un datetime, alors l'opérande numérique est considéré comme "second":

> Il y a 24 * 60 * 60 = 86400 secondes par jour

```
? Datetime() + 86400 && Add 1 day to current datetime.
```

Ajouter 4 heures et 15 minutes au 1er janvier 2016 14h20

```
? Datetime(2016, 1, 1, 14, 20, 0) + (4 * 3600 + 15 * 60)
```

Sorties vendredi 1 janvier 2016, 18h35.

> Avec une simple impression utilisant?, Ce que vous voyez sur votre écran dépend de vos paramètres de date. Par exemple, si vous n'avez rien changé, vos paramètres de date sont de style américain (MDY), le format 12 heures (AM / PM) et le siècle est affiché avec les 2 derniers chiffres uniquement.
>
> > Il existe un symbole spécial ^ pour les dates et les dates, qui oblige une chaîne à être interprétée «strictement» au format aaaa / MM / jj [HH: mm:

ss | hh: mm: ss tt]. Par conséquent, ^ peut également être considéré comme un opérateur date / datetime. Par exemple, considérez que certaines données proviennent d'une source dans un format comme 201610082230 (aaaaMMjjHHmm). Pour obtenir cette valeur en tant que date / heure valide:

```
Local cSample, tSample
cSample = '201610082230'
tSample = Ctot(Transform(m.cSample, '@R ^9999/99/99 99:99'))
? Transform(m.tSample, '@YL')
```

Sorties (en fonction du paramètre de date longue de votre système):

Samedi 8 octobre 2016 à 22h30

- est utilisé pour la soustraction. Ses opérandes sont soit des valeurs date / date / heure, soit l'une est une date / date et l'autre est une valeur numérique.

Commençons par les opérandes date / datetime et numeric simples (comme avec l'opérateur +):

Quand l'un des opérandes est une date, alors l'opérande numérique est considéré comme "jour":

```
? Date() - 10 && What was the date 10 days ago?
? Date(2016, 3, 1) - 1 && Returns Feb 29, 2016.
? Date(2017, 3, 1) - 1 && Returns Feb 28, 2017.
```

Lorsqu'un des opérandes est un datetime, alors l'opérande numérique est considéré comme "second":

```
? Datetime() - 86400 && Go back exactly one day
```

Il y a 1 heure et 30 minutes depuis "maintenant":

```
? Datetime() - (1 * 3600 + 30 * 60)
```

La deuxième forme consiste à obtenir la différence entre deux valeurs de date / date / heure. Les opérandes sont à la fois date et date, vous ne pouvez pas utiliser la date et l'heure au même moment (faites la conversion de type si nécessaire, VFP ne le fait pas pour vous). Les règles sont comme dans + et -, les opérandes sont la date, la différence est en **jours** , les opérandes sont datetime, alors la différence est en **secondes** .

Combien de jours à la veille du nouvel an (pour l'année 2016)?

```
? Date(2016, 12, 31) - Date()
```

Combien de secondes il reste à minuit?

```
? Dtot(Date()+1) - Datetime()
```

Dans le dernier exemple, nous avons utilisé une fonction Date / Datetime, DTOT - DateToTime pour obtenir la valeur de minuit du lendemain. Il existe de nombreuses fonctions de date / date / heure utiles, nous les avons toutes ignorées car elles ne sont pas considérées comme des opérateurs (bien qu'elles fonctionnent sur les dates / dates). Il en va de même pour les autres opérateurs.

La soustraction date / date / heure est **signée** . C'est-à-dire que si vous utilisez la date / date / heure la plus petite comme premier opérande, le résultat sera négatif. Vous pouvez utiliser la fonction abs () si vous avez besoin d'un résultat positif, quel que soit l'ordre des dates / dates.

Opérateurs relationnels

De tous les opérateurs, les opérateurs relationnels sont les plus complexes, c'est pourquoi nous les avons laissés jusqu'au bout.

Les opérateurs relationnels sont également appelés opérateurs de comparaison, ils sont utilisés pour comparer des choses.

Le résultat de la comparaison est booléen faux ou vrai.

Il est intéressant de noter que si vous le cochez dans VFP, vous ne verrez qu'une courte liste d'opérations et quelques lignes supplémentaires comme si c'était tout au sujet de ces opérateurs.

Eh bien, la complexité vient du fait qu'ils fonctionnent sur ***tous les types,*** que ce soit numérique, date, date-heure, logique ou chaîne, et même sur des objets. De plus, le comportement peut sembler bizarre, vous n'obtenez pas ce que vous attendez à *moins de* savoir ce qui affecte les résultats.

Commençons par une liste d'opérateurs relationnels:

Opérateur	La description	PLUS d'échantillon de base
>	Plus grand que	? 1> 2 && .F.
<	Moins que	? 1 <2 && .T.
> =	Plus grand ou égal à	? 1> = 2 && .F.
<=	Inférieur ou égal à	? 1 <= 2 && .T.
=	Égal à	? 1 = 1 && .T.
==	Est exactement égal à (fait sens pour les chaînes)	? '1' = '1' && .T.
! =, #, <>	Pas égal à (tous les 3 opérateurs agissent de la même manière, choisissez votre favori)	? 1! = 1 && .F.

Bien que vous puissiez les utiliser avec tous les types de données, il doit exister une compatibilité de type entre les opérandes. Par exemple, vous obtiendrez une erreur si vous essayez de comparer une date à un nombre entier.

La date et la date / heure peuvent être comparées, bien qu'il s'agisse de types différents, VFP effectue la conversion implicitement pour vous.

```
? Date() > DateTime() && .F.
? Date() <= DateTime() && .T.
? Date() < DateTime() && .T. if it is not midnight
```

Lorsque les opérandes sont numériques, tous ces opérateurs sont simples et directs, ils fonctionnent comme ils le feraient dans les expressions mathématiques.

Avec les opérandes logiques, .F. est considéré comme inférieur à .T.

Avec les objets, ce que nous comparons est la référence de l'objet en mémoire. La comparaison la plus utilisée consiste donc à déterminer si deux variables d'objet pointent vers le même objet. c'est à dire:

```
local o1, o2
o1 = createobject('Label')
o2 = createobject('Label')
? m.o1 = m.o2 && is o1 and o2 the same object?
? m.o1 > m.o2 && this would work too but likely you would never use

* remember we are comparing their references in memory
*
* They are different objects, but do they have any difference in their properties?
? CompObj(m.o1, m.o2) && .T. They are identical properties wise
```

La comparaison du type de données de caractère, c'est-à-dire la comparaison des chaînes de caractères, est la plus déroutante de VFP. Cela ne fonctionne pas comme dans d'autres langues et / ou bases de données et est unique à VFP (et peut-être à un autre langage xBase).

> Il y a de nombreuses années, j'ai même vu des membres très avancés de la communauté qui ne connaissaient pas encore le fonctionnement de ces opérateurs dans VFP. Il est donc tout à fait compréhensible que de légères nuances puissent facilement perturber les débutants.

La comparaison consiste essentiellement à être égal ou non. Si elles ne sont pas égales, alors nous pourrions penser aux opérateurs>, <,> =, <=, non? Avec des chaînes, la confusion est grande lorsque *deux chaînes sont considérées égales* .

> **Important: les** chaînes VFP sont sensibles à la casse. «A» et «a» sont deux chaînes distinctes. Ce n'est pas le cas avec de nombreuses bases de données où la valeur par défaut est d'utiliser un classement insensible à la casse. Par exemple, dans postgreSQL ou MS SQL Server sur une table créée avec un classement insensible à la casse (CI):

```
select * from myTable where Country = 'TURKEY'

select * from myTable where Country = 'Turkey'
```

donnerait le même résultat. Dans VFP, vous n'obtenez que ceux où correspond le casing. Cependant, VFP prend en charge certaines collations et effectue une comparaison insensible à la casse. (Ne faites pas confiance, voir ci-dessous)

- Si deux chaînes ne sont pas égales, tout va bien, à **condition de ne modifier aucune valeur par défaut,** puis de les comparer en *fonction de leurs valeurs ASCII* .

```
? 'Basoz' < 'Cetin' && is true.
? 'basoz' < 'Cetin' && is false.
? 'Cetin' < 'David' && is true.
? 'Çetin' < 'David' && is false.
```

La valeur par défaut pour le classement est "machine" et c'est ce que vous obtenez alors. Lorsque vous modifiez le classement en quelque chose d'autre, vous obtenez la comparaison basée sur l'ordre de tri de ce classement. Avec les paramètres de classement autres que ceux de la **machine** par défaut **,** vous indiquez également une insensibilité à la casse sur la comparaison (ne faites PAS cela pour l'égalité):

```
set collate to 'GENERAL'
? 'Basoz' < 'Cetin'
? 'basoz' < 'Cetin'
? 'Cetin' < 'David'
? 'Çetin' < 'David'
```

Toutes ces expressions sont maintenant TRUE.

Conseil **personnel** : Collations dans VFP n'a jamais été suffisamment fiable. Je vous suggère de ne pas utiliser les classements et de vous en tenir à la 'MACHINE' par défaut. Si vous utilisez des classements, gardez à l'esprit de vérifier d'abord si vous rencontrez quelque chose de très inattendu en ce qui concerne les données de caractère. J'ai vu et démontré que cela échouait dans de nombreux cas, mais alors j'ai arrêté d'essayer de l'utiliser beaucoup avant la version de VFP9, cela pourrait être cohérent maintenant, je ne sais vraiment pas.

Étant donné que nous avons couvert les cas d'inégalité avec des chaînes, le plus délicat est le cas de l'égalité. Dans VFP, deux paramètres affectent la comparaison:

1. SET EXACT (la valeur par défaut est OFF et permet d'effectuer des comparaisons régulières, sauf SQL)
2. SET ANSI (la valeur par défaut est OFF et permet des comparaisons uniquement en SQL. SET EXACT **n'a aucun effet** sur les comparaisons effectuées dans les requêtes SQL.

Avec SET EXACT OFF, lisez la comparaison comme "la chaîne à droite commence-t-elle par la chaîne à gauche"? Ils sont comparés à la longueur de la chaîne de droite.

```
? "Bobby" = "B" && Bobby starts with B, so TRUE
```

```
? "Bobby" = "Bob" && Bobby starts with Bob, so TRUE
? "Bobby" = "Bob " && Bobby starts with Bob but there is a trailing space there, FALSE
? "Bobby" = "bob" && would be true with collation set to GENERAL
```

> Notez qu'avec une comparaison régulière, "Bobby" = "B" est VRAI, mais "B" = "Bobby" est FAUX. En d'autres termes, la place des opérandes est importante.

Avec SET EXACT ON, les chaînes doivent correspondre parfaitement mais leurs espaces de fin sont ignorés (nous ignorons ici l'assemblage des ensembles, ce qui rendrait également insensible à la casse):

```
? "BOBBY" = "BOB" && FALSE
? "BOBBY" = "BOBBY" && TRUE
? "BOBBY" = "BOBBY     " && TRUE
? "BOBBY     " = "BOBBY" && TRUE
```

Maintenant, avec les commandes SQL, SET EXACT n'a aucun effet et se comporte comme le fait SET EXACT OFF.

```
Select * from Customers where Country = 'U'
```

Choisirait les clients des Etats-Unis, du Royaume-Uni, n'importe quel pays commençant par 'U'.

En SQL, cependant, par définition, modifier l'ordre des opérandes devrait donner le même résultat. Ainsi:

```
Select * from Customers where 'U' = Country
```

fonctionnerait également de la même manière (notez la différence avec les commandes non-SQL).

Lorsque vous souhaitez impliquer des correspondances exactes, une option consiste à activer ANSI:

```
SET ANSI ON
Select * from Customers where Country = 'USA'
```

renvoie tous les clients des États-Unis. Notez que les espaces de fin dans le champ de pays OR sur l'expression de droite sont ignorés. Peu importe le nombre de passagers de chaque côté. Vous obtenez la comparaison comme si cela avait été fait comme: RTRIM (Country) = RTRIM ('USA').

> Bien qu'il ne soit pas mentionné dans Opérateurs dans VFP, un *opérateur* SQL est LIKE. Lorsque vous utilisez LIKE, vous obtenez une comparaison de correspondance exacte, quel que soit le paramètre SET ANSI (en utilisant les forces LIKE et le cas ANSI ON implicite - après tout, il s'agit d'un opérateur ANSI). Cependant, attention, il y a une légère différence de comportement. Il n'ignorerait pas les espaces de fin, à **moins que** la taille totale des remorques ne soit égale ou inférieure à la taille du champ. Par exemple, si le champ Country est C (10), alors Country = 'USA' ou Country

= 'USA__' fonctionnerait, mais Country = 'USA_____' échouerait (les *caractères soulignés* indiquent un espace et le dernier contient plus de 7 espaces de fin).

Enfin, nous sommes au dernier opérateur, ==. Cela signifie exactement égal et fait avec utiliser des chaînes. Un avantage est que, en utilisant ==, vous voulez toujours dire que vous voulez une correspondance exacte, **indépendamment des** paramètres SET EXACT ou SET ANSI. Cependant, attention, son comportement est différent lorsqu'il s'agit d'une commande SQL ou d'une commande régulière non SQL.

Avec SQL:

```
Select * from Customers where Country == 'USA'
```

Quels que soient les paramètres ANSI et EXACT, nous voulons tous les clients des États-Unis uniquement. Les espaces de fin de chaque côté sont ignorés.

Avec non-SQL:

```
? m.lcString1 == m.lcString2
```

serait vrai ***que si*** elles sont exactement les mêmes, en ce qui concerne leur boîtier et la longueur (les espaces de fin ne sont pas ignorés). Il n'est pas effectué à partir des paramètres SET ANSI, EXACT ou COLLATE.

Lire Les opérateurs en ligne: https://riptutorial.com/fr/visual-foxpro/topic/7625/les-operateurs

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec visual-foxpro	Cetin Basoz, Community, Steve
2	Interopérabilité VFP avec .NET	Steve
3	Les opérateurs	Cetin Basoz