Handling Code that Changes at Runtime

What's the best way to do things if you don't know exactly what code to run until you need to run it?

Tamar E. Granor, Ph.D.

There are lots of places in VFP code where you don't know until the code is running exactly what you want it to do. Perhaps you want to let the user choose a file to operate on, or you want to run a report based on criteria specified by a user. VFP offers a number of different ways to handle code that isn't known until run-time. Knowing which one to use when affects the efficiency, accuracy, and reliability of your code, but many people stick with macros and don't use any of the other options. In this article, I'll look at each of the options and discuss where it's appropriate.

Macros

The macro substitution operator, &, was already available when I started using FoxBase+. As its name suggests, it substitutes the contents of the specified variable into the code. Macros are useful when you need to substitute a keyword into your code or when you want to build up all or part of a command before executing it.

The prototypical use for macros (and, I suspect, the reason we have them at all) is in handling the various SET commands that let us control the way VFP operates. Many of them expect the literal string ON or OFF as their operand. When you need to save and restore one of these values, a macro makes the code quite compact, as in Listing 1.

Listing 1. The macro operator lets you substitute the right keyword into the SET commands.

```
cOldSafety = SET("SAFETY")
SET SAFETY OFF

* Code that overwrites a file
* without prompting

SET SAFETY &cOldSafety
```

The other principal use for macros in VFP 9 is to build a command or a portion of a command based on user input or other factors. For example, my applications have a form class for reporting that includes an abstract method called GetReportData. Forms based on that class usually contain a number of controls where the user indicates which records to include in the report. The code in GetReportData typically populates a variable called cWhere based on the user's selections, and then macro-expands cWhere in a SQL SELECT command to collect the desired data. The code in Listing 2 is drawn from such a form. It stores a condition to the variable cWhere, based on whether the user wants to list only people with US addresses, only those with non-US addresses or only those from a particular state or province. That variable is then used in the WHERE clause of the query that grabs the report data.

Listing 2. Macros are handy when you want to construct all or part of a command in code and then execute it.

```
cWhere = ".T."

DO CASE
CASE This.opgIncludeLocation.Value = 2
   * US only
   cWhere = [cCountry = "USA"]

CASE This.opgIncludeLocation.Value = 3
   * Foreign only
   cWhere = [cCountry <> "USA"]

CASE This.opgIncludeLocation.Value = 4
   * One state
   cWhere = [cState = "] + ;
     This.cboStateProvince.Value + ["]

OTHERWISE
ENDCASE

* Some other things happen in here,
* including cOrder getting set to
* the list of sort fields chosen.
SELECT iID, cFirst, cLast, cPublic, ;
       mStreetAddr + CRLF + ;
       IIF(EMPTY(cCity), "", ;
           ALLTRIM(cCity) + ", " + ;
           ALLTRIM(cState) + " " + ;
           cPostCode + CRLF) + cCountry ;
         AS cFullAddress, ;
       mEmail, curPerson.mNotes, ;
       lForum, dJoined, ;
       cCountry, cState, cPostCode ;
  FROM curPerson ;
  WHERE &cWhere ;
  ORDER BY &cOrder ;
  INTO CURSOR csrPerson
```

In early versions of Fox, macros were also useful for letting you work with tables and fields with-

out knowing their names. For example, code like Listing 3 was common.

Listing 3. In early versions of Fox, accessing a table or field without knowing its name required a macro.

```
USE &cTable
```

Today, you should avoid using macros with variables that contain filenames. Since Windows 95 eliminated the 8.3 naming convention for files, file names may contain spaces. When you macro-expand a filename with embedded spaces, you'll either get an error or unexpected behavior. For example, say the variable cTable contains "C:\DOCUMENTS AND SETTINGS\TAMAR GRANOR\APPLICATION DATA\MICROSOFT\VISUAL FOXPRO 9\FOXUSER.DBF", which is the default location for the VFP Resource file. Issuing USE &cTable results in error 36, "Command contains unrecognized phrase/keyword." That's because everything after the first space is seen as additional clauses for the USE command; since there's no AND clause, the command fails. See the section on Name expressions later in this article for a better alternative.

The EVAL() function

Eventually, the Fox team noticed that while macros were useful, sometimes you knew that what you wanted to evaluate on the fly was an expression. Apparently, knowing that you're evaluating an expression rather than a command or a part of a command allows the FoxPro engine to do things faster. So, in FoxPro 2.0, they added the EVALUATE() function (fairly universally abbreviated EVAL()) that lets you store an expression in a string and then evaluate it.

EVAL() is useful in reports, where macros don't work. For example, if you want to put the third field of the table in a report expression and you don't know its name, you can use EVAL(FIELD(3)).

EVAL() is also handy for turning the names of objects into object references. For example, if you have a variable cControl that contains the name of a control on a form, you can get a reference to the control itself with code like Listing 4.

Listing 4. Use EVAL() to turn the names of objects into object references.

```
oControl = EVAL("ThisForm." + m.cControl)
```

Don't use EVAL() in FOR clauses; use a macro instead. When you EVAL(), the string is re-evaluated for each record, while a macro is expanded once. (The condition is evaluated for each record in either case, but the preparation varies.) I used the program in Listing 5 (MacroVsEval.PRG in this month's downloads) to test; I found that the macro version took about a quarter of the time of the EVAL() version.

Listing 5. While EVAL() is usually a better choice than a macro, that's not true in a FOR clause.

```
#DEFINE PASSES 1000

OPEN DATABASE HOME(2) + "Northwind\Northwind"

USE Northwind!OrderDetails

LOCAL nStart, nEnd, nPass, cCondition

cCondition = "Discount > 0"

nStart = SECONDS()
FOR nPass = 1 TO PASSES
   COUNT FOR &cCondition
ENDFOR
nEnd = SECONDS()

?"With macro, ", PASSES, " passes take ", ;
 nEnd-nStart

nStart = SECONDS()
FOR nPass = 1 TO PASSES
   COUNT FOR EVALUATE(m.cCondition)
ENDFOR
nEnd = SECONDS()

?"With EVALUATE(), ",PASSES, ;
 " passes take ", nEnd-nStart

RETURN
```

Name expressions

At the same time that EVAL() was added, the Fox team gave us name expressions, also known as indirect references. These let you store the name of a thing in a variable and operate on that variable. It's useful anywhere that VFP expects a name, whether it's a table name, a field name, a file name, an alias, or some other name. For example, if you have the name of a table you want to open in the variable cTable, you can open the table as in the first line of Listing 6. You can go farther with this, though. Say, you have not just the table name, but the order you want and the alias you want to assign stored in variables. Each can use a name expression, as in the second line.

Listing 6. Use a name expression to open a table when the table name is stored in a variable.

```
USE (m.cTable) IN 0

USE (m.cTable) IN 0 ORDER (m.cOrder) ;
   ALIAS (m.cAlias)
```

The biggest reason for using name expressions is macros' inability to handle names that include embedded spaces. While you may choose not to use spaces in file names, you rarely can control the complete path to your application. So addressing files with a macro is an invitation to crash your code.

It's been an article of faith for many years that name expressions are faster than macros. In my tests, that's true, but the differences I see are minimal, on the order of 1% or less. However, it appears that the slow part in my tests is opening

the table and that that's overwhelming the difference between the two approaches. If I test with the table already open in another work area (MacroVsNameExp.PRG in this month's downloads), I find that using the name expression takes about 75% of the time of the macro.

Regardless of speed issues, the protection name expressions give you for embedded spaces means that you should always use a name expression rather than a macro when the command expects a name.

Compile at runtime

For many years, these three techniques (macros, EVAL() and name expressions) were all the native choices you had for specifying code at runtime. Then, in Service Pack 3 for VFP 6, the Fox team offered another option: compiling code at runtime.

Until that version, the COMPILE command worked only at design-time, so you had to write and compile all code ahead of time, though you could use code or data to decide which code to actually run.

Combined with the text manipulation commands I covered in earlier articles in this series (such as StrToFile()), this change made it possible to generate code dynamically, then compile it and execute. But the need for COMPILE at runtime didn't last long.

EXECSCRIPT()

In VFP 7, the Fox team added the EXECSCRIPT() function, which compiles and executes the string you pass it. The big advantage EXECSCRIPT() and runtime compilation have over earlier techniques is that they can handle multi-line sequences of code, where a macro allows you to substitute for at most one full line of code. This means that you can even use structures like loops in code executed this way.

EXECSCRIPT() has one required parameter, a string, which it executes. If an error occurs while executing the code, the prevailing error handler deals with it.

EXECSCRIPT() can also accept parameters to pass on to the code it executes. The syntax is:

```
EXECSCRIPT(cCode [, uParam1 [, uParam2, … ]])
```

Where would you use it EXECSCRIPT()? One place is for data-driving processes. For example, in one application I'm working on, I have to check Canadian health IDs to make sure they meet the rules for the appropriate province. There are some common techniques, but each province has slightly different rules.

I have a class that contains methods for common functionality. The process is driven by a table with one record for each province. That record has a memo field containing the exact code I need to execute to do the check for that province. It may or may not use some of the common methods. Each of those code blocks accepts two parameters: the health ID to check and an object reference to the class.

My code to do the actual check pulls the memo field for the right province into a variable, cCode, and then has this line:

```
lReturn = EXECSCRIPT(cCode, ;
            ALLTRIM(cID), This)
```

By passing a reference to the ID checking class into the stored code, it can make calls to the common methods in the class.

The ability to call code this way avoids having a complex CASE statement in this code, and means that if (when) a province changes its rules, only a single table needs to change, not the whole class.

Making the right choice

With so many ways to specify code at runtime, how can you decide which one to use? The rules are actually pretty simple:

When you want to use a variable or expression to substitute for a name, always use a name expression.

When you want to substitute for an expression, use EVAL() except when you're in a FOR clause.

When you want to substitute for all or part of a command, but not a name or expression, use a macro.

Finally, when you need to substitute for a block of lines, use EXECSCRIPT().

Help me write this column

If you've changed the way you write VFP code, and you're using new functions, commands, classes, or other language elements that I haven't covered in this column, please drop me a note at the address below.

Author Profile

Tamar E. Granor, Ph.D. is the owner of Tomorrow's Solutions, LLC. She has developed and enhanced numerous Visual FoxPro applications for businesses and other organizations. She currently focuses on working with other developers through consulting and subcontracting. Tamar is author or co-author of ten books including the award winning Hacker's Guide to Visual FoxPro, Microsoft Office Automation with VisualFoxPro and Taming Visual FoxPro's SQL . Her latest collaboration is Making Sense of Sedna and SP2, coming out this year. Her books are available from Hentzenwerke Publishing (www.hentzenwerke.com). Tamar is a Microsoft Support Most Valuable Professional. In 2007, Tamar received the Visual FoxPro Community Lifetime Achievement Award. You can reach her at tamar@thegranors.com or through www.tomorrowssolutionsllc.com.