FoxRockX

Use MDots for speed, not just for correctness

Prefixing variable references with "m." doesn't just make your code unambiguous; it makes it faster.

Tamar E. Granor, Ph.D.

January 2016
Number 48

There may be no topic on which VFP developers as a group feel more strongly than whether or not to prefix all references to variables with "m." in order to prevent ambiguity. It may be time for that argument to end, though, because it turns out that using mdots makes your code run faster, too.

From its earliest days, FoxPro has given preference to field names in expressions. When an expression includes a name that's both a field of the table open in the current work area and a variable, unless you tell it otherwise, FoxPro uses the field. That is, when you have code like Listing 1, VFP first looks for fields named nHeight and nWidth. Only if it can't find them does it decide you must have meant variables.

Listing 1. When names are used in an expression, VFP gives preference to fields.

```
nArea = nHeight * nWidth
```

If you want to use the variable rather than a field of the same name, you can precede it with the letter "m" and a period. The combination is typically called "mdot" by VFP developers. Listing 2 shows the previous example with variables clearly indicated.

Listing 2. Mdots make it clear that you mean a variable.

```
nArea = m.nHeight * m.nWidth
```

In this example, mdot isn't needed for nArea because only variables can be assigned a new value using the equals sign.

Naming conventions as a solution

Because of this behavior, many VFP developers have adopted naming conventions that are meant to ensure that they never have variables and fields with the same name. The most common notation (recommended in the VFP Help file and generally referred to as "Hungarian") uses a scope letter ("l" for local, "p" for private, "g" for global/public) followed by a type letter ("c" for character, "n" for numeric, etc.) at the front of every variable name. In that notation, fields get a type letter, but no scope indicator. Using this notation, a field representing height would be nHeight, but a variable for height would be lnHeight.

The problem with relying on a naming convention is that VFP doesn't know about it and it doesn't prevent all conflicts. For example, it's not impossible to imagine having a field named lOrange and a variable named loRange. While these look different to a human reader, to the VFP engine, they're exactly the same and the field will be used any time there's ambiguity.

By now, you can probably tell that I'm in the "always use mdots" camp, and if you're firmly in the "no mdots" camp, no argument I can make about how VFP works or the potential for errors is likely to convince you.

MDots are faster

However, it also turns out that using mdots makes your code run faster. How much faster depends on the number of variable references and the number of fields in the table open in the current workarea.

I recently tested on two different computers, using two different programs, one with just a few variable references and one with many more. In each case, I also tested for different numbers of fields in the current workarea, starting with no table open, then with a table (actually, a cursor) with five fields, then one with 10 fields, and so on all way up to 200 fields in the table in the current work area.

Given VFP's preference for fields, I wasn't surprised to see that mdot was faster and that the more fields in the table in the current work area, the greater the advantage of mdot.

Listing 3 shows the first test program, the one with fewer variable references. The code uses height and width variables to compute perimeter and area. The computations are performed in a loop that runs for five seconds; there are a total of six references to variables in the loop and the computations.

Listing 3. This program compares use of variables with mdots to use of variables without mdots. The block being tested contains six variable references.

```
* Compare speed with and without mdot

#DEFINE SECONDSTORUN 5

LOCAL nCase1Start, nCase1LoopEnd,
nCase2LoopStart, nCase2LoopEnd
LOCAL nCase1Passes, nCase2Passes
LOCAL nLength, nWidth, nPerimeter, nArea

* Test multiple cases from no table open
* to table with many fields open.
* Store results in a cursor in a different
* workarea.

CREATE CURSOR csrMDotSpeeds
  (nFields N(3), nNoMDots I, nMDots I)
SELECT 0

LOCAL nFields, nField, cFieldList

* Initialize variables for calculations
nLength = 27.3
nWidth = 13.7

FOR nFields = 0 TO 200 STEP 5
  IF m.nFields <> 0
    cFieldList = ''
    FOR nField = 1 TO m.nFields
      cFieldList = m.cFieldList + "cField" + ;
        TRANSFORM(m.nField) + " C(5), "
    ENDFOR
    cFieldList = TRIM(m.cFieldList, ", ")

    CREATE CURSOR csrDummy (&cFieldList)
  ELSE
    SELECT 0
  ENDIF

  * Now do the test

  nCase1LoopStart = SECONDS()
  nCase1LoopEnd = m.nCase1LoopStart + ;
                  SECONDSTORUN
  nCase1Passes = 0

  DO WHILE nCase1LoopEnd > SECONDS()
    nCase1Passes = nCase1Passes + 1

    nPerimeter = 2*nLength + 2*nWidth
    nArea = nLength * nWidth
  ENDDO

  nCase2LoopStart = SECONDS()
  nCase2LoopEnd = m.nCase2LoopStart + ;
                  SECONDSTORUN
  nCase2Passes = 0

  DO WHILE m.nCase2LoopEnd > SECONDS()
    nCase2Passes = m.nCase2Passes + 1

    nPerimeter = 2*m.nLength + 2*m.nWidth
    nArea = m.nLength * m.nWidth

  ENDDO

  INSERT INTO csrMDotSpeeds
    VALUES (m.nFields, m.nCase1Passes, ;
            m.nCase2Passes)

  IF m.nFields <> 0
    USE IN csrDummy
  ENDIF
ENDFOR

RETURN
```

The results of this test on the two different machines were quite similar. With no table open in the current work area (the 0 case), the version without mdots was very slightly faster. After that, however, the mdots version was always faster. With 30 fields in the table, the mdots version completed more than 25% more iterations; with 50 fields, the mdots version completed 50% more iterations. By the top end of the test, 200 fields, the mdots version made 2.7 times as many passes.

The number of iterations completed by the code using mdots was remarkably stable. For a given machine, the difference between the maximum and the minimum was less than .02% of the maximum value.

On the other hand, the number of iterations completed by the code without mdots descended pretty steadily. With 200 fields, only about a third as many iterations were completed as with no open table.

It's important to note that we're talking about millions of iterations in five seconds, so the effect is small for any given variable reference. However, in an application, you likely have thousands or tens of thousands of variable references; in a typical application, it's likely that most of them occur with a table in the current area.

A larger test

I wanted to see the difference mdot makes in a program with many more variable references than the perimeter and area example. To do so, I adapted a piece of code from a client application. The core of the code is a function that determines whether a specified point is "near" a specified line. It accepts four parameters, a line, a point (in the form of row and column coordinates), and a tolerance. The tolerance indicates how far from the line something can be and still be considered "near." The actual code isn't important, but the function contains nearly 60 potentially ambiguous variable references.

The test, structured the same way as the previous test, is shown in Listing 4.

Listing 4. This code tests the speed of a program with more than 50 references to variables with and without mdots.

```
* Compare speed with and without mdot

#DEFINE SECONDSTORUN 5

LOCAL nCase1Start, nCase1LoopEnd, nCase2Loop-
Start, nCase2LoopEnd
LOCAL nCase1Passes, nCase2Passes

* Test multiple cases from no table open
* to table with many fields open.
* Store results in a cursor in a different
* workarea.

CREATE CURSOR csrMDotSpeedsLarge ;
  (nFields N(3), nNoMDots I, nMDots I)
SELECT 0
LOCAL nFields, nField, cFieldList
LOCAL oLine AS Line

oLine = CREATEOBJECT("Line")
oLine.Left = 27
oLine.Top = 13
oLine.Height = 152
oLine.Width = 53

FOR nFields = 0 TO 200 STEP 5
  IF m.nFields <> 0
    cFieldList = ''
    FOR nField = 1 TO m.nFields
      cFieldList = m.cFieldList + "cField" + ;
        TRANSFORM(m.nField) + " C(5), "
    ENDFOR
    cFieldList = TRIM(m.cFieldList, ", ")

    CREATE CURSOR csrDummy (&cFieldList)
  ELSE
    SELECT 0
  ENDIF

  * Now do the test

  nCase1LoopStart = SECONDS()
  nCase1LoopEnd = m.nCase1LoopStart + ;
                  SECONDSTORUN
  nCase1Passes = 0

  DO WHILE nCase1LoopEnd > SECONDS()
    nCase1Passes = nCase1Passes + 1

    IsPointNearLineNoMDot(oLine, 55, 45, 1)
    IsPointNearLineNoMDot(oLine, 100, 27, 2)
    IsPointNearLineNoMDot(oLine, 0, 0, 1)
    IsPointNearLineNoMDot(oLine, 500, 7, 3)
  ENDDO

  nCase2LoopStart = SECONDS()
  nCase2LoopEnd = m.nCase2LoopStart + ;
                  SECONDSTORUN
  nCase2Passes = 0

  DO WHILE m.nCase2LoopEnd > SECONDS()
    nCase2Passes = m.nCase2Passes + 1

    IsPointNearLineMDot(m.oLine, 55, 45, 1)
    IsPointNearLineMDot(m.oLine, 100, 27, 2)
    IsPointNearLineMDot(oLine, 0, 0, 1)
    IsPointNearLineMDot(oLine, 500, 7, 3)
  ENDDO

  INSERT INTO csrMDotSpeedsLarge ;
    VALUES (m.nFields, m.nCase1Passes, ;
            m.nCase2Passes)

  IF m.nFields <> 0
    USE IN csrDummy
  ENDIF
ENDFOR
```

By now, it should be no surprise that the more fields in the table open in the current work area, the greater the advantage of the version with mdots. In my tests, the mdot version ran about 25% more times at 70 fields in the work area and about 50% more times with 150 fields.

I suspect the reason the difference isn't as extreme as the earlier example is that there's a lot more code that isn't variable references in this example. That is, the overall code is more complex. (In fact, while the earlier example managed millions of passes in five seconds, the larger example completed only tens of thousands.)

To get a better sense of the difference between the two tests, I computed a rough "time per variable reference" for each. Specifically, I did the calculation in Listing 5, dividing the five seconds of the test by the product of the number of variable references and the number of passes completed. Of course, this is only an approximate time per variable reference because there's other code in each test. However, it let me do a comparison between the two tests.

Listing 5. This equation computes an approximate "time per variable reference."

```
Time = TestTime/((# of variables) * passes)
```

What I found was that the second, more complex, test took about an order of magnitude longer for each reference than the simpler test. Again, that's likely a reflection of the additional code in the more complex case.

What about arrays?

The code that determines whether a point is near a line uses a couple of arrays in its calculations. Since a reference to an array element can't be mistaken for a reference to a field, I wondered whether it makes a difference to use mdots on those references.

I tested by adding a third case to the larger test. It's structured the same way as the two tests in Listing 4, but calls a third version of IsPointNearLine that has mdots on references to scalar variables, but not on references to array elements.

I found only a tiny difference between this version and the one with mdots on all variable references including arrays. Most of the time (66 out of 82 cases), the one without mdots on array references was faster, but sometimes the one using mdots on array references was faster. That suggests that VFP is smart enough to not look (or to not look very hard) for a field when given an array reference.

The code for testing all three cases is included in the downloads for this session as UseMDotLarge.PRG, along with the three versions of the function to determine whether a point is near a line. They're included in this month's downloads as IsPointNearLineNoMDot.PRG, IsPointNearLineMDot.PRG and IsPointNearLineMDotNotArrays.PRG

A few words about timing tests

Testing in the Windows environment is inherently flawed. Between Windows itself and various services that are always running, any one test result might be inaccurate.

There are two things you can do to get better results. First, before testing, turn off anything you can that might interfere, such as an email client, on-demand virus scanning, and so forth. If you don't need a network for the test, consider disconnecting.

Second, perform more than one test for each case. That advice is also important because VFP caches data, so the first time you run a process that uses DBFs, it's likely to take longer than subsequent runs.

As I mentioned earlier, I did my testing on two different machines. In both cases, I made sure that Outlook and my Twitter client were closed. When a test was running, I didn't do anything else with that computer, not even touch the keyboard or move the mouse. In addition, over the course of writing, I ran each of my tests a number of times.

Even with these safeguards, test results should be seen more as an indicator than as a definitive answer. In this case, because the difference between the mdots and no mdots results are so large, it's safe to assert that mdots makes a difference. On the other hand, the difference between mdots on all variable references and mdots only on non-array element variable references is small enough to only hint at the answer. More testing in a more controlled environment is needed to confirm that result.

Just use mdots

As I said at the beginning, I'm already on the mdots bandwagon. I've been bitten too many times by code using a field when I meant a variable and I don't want to worry about that ever again. In addition, I work often on code originally written by others, so even if I adopted a strict naming convention, much of the code I touch likely wouldn't be using it.

But even if you truly believe your naming convention will protect you from that problem, the fact that omitting mdots makes your application slower should make you rethink your choice.

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. Tamar is author or co-author of a dozen books including the award winning Hacker's Guide to Visual FoxPro, Microsoft Office Automation with Visual FoxPro *and* Taming Visual FoxPro's SQL. *Her latest collaboration is* VFPX: Open Source Treasure for the VFP Developer, *available at www.foxrockx.com. Her other books are available from Hentzenwerke Publishing (www.hentzenwerke.com). Tamar was a Microsoft Support Most Valuable Professional from the program's inception in 1993 until 2011. She is one of the organizers of the annual Southwest Fox conference. In 2007, Tamar received the Visual FoxPro Community Lifetime Achievement Award. You can reach her at tamar@thegranors.com or through www.tomorrowssolutionsllc.com.*