Breaking Up is Not Hard to Do

VFP provides lots of tools for breaking strings up into their component parts.

Tamar E. Granor, Ph.D.

In my last article, I looked at techniques for reading and writing text files. Once you've read a text file into memory, or perhaps created a long string in some other way, it's not unusual to need to break it up into lines, or words, or based on some other criteria. Prior to VFP 6, you had to use different approaches depending on the criteria for parsing. With the introduction of the ALINES() function, though, most simple parsing has been reduced to a single function call.

Parsing into lines

Perhaps the most common parsing task is taking a long string and dividing it into lines. The oldest way to do this kind of parsing is to use AT() to find the next end-of-line, then use LEFT() and/or SUBSTR() to pull it out. Code like Listing 1 does the job. However, this code is inflexible since it looks only for a CHR(13) + CHR(10) combination to end the line. A line might end with just CHR(13) or just CHR(10). It's also slow; I'll discuss timing a little later on.

Listing 1. You can parse a string into lines with AT(), LEFT() and SUBSTR(), but it's slow and not very flexible.

```
nBreak = AT(CHR(13) + CHR(10), m.cString)
nLine = 0
DO WHILE m.nBreak > 0
   nLine = m.nLine + 1
   DIMENSION aContents3[m.nLine]
   aContents3[m.nLine] = LEFT(m.cString, ;
                              m.nBreak-1)
   cString = SUBSTR(m.cString, m.nBreak + 2)
   nBreak = AT(CHR(13) + CHR(10), m.cString)
ENDDO
IF NOT EMPTY(m.cString)
   nLine = m.nLine + 1
   DIMENSION aContents3[m.nLine]
   aContents3[m.nLine] = m.cString
ENDIF
```

Looking a little bit outside the box provides a second approach. The MLINE() function is advertised as meant for retrieving one line from a memo field. However, it actually works on any string. MLINE() accepts three parameters: the string or memo field to look at, the line number to return, and optionally, a starting point in the string. Using that optional third parameter improves the function's performance significantly. Specifically, the system variable _MLINE keeps track of where you were in parsing a string and can be used to start where you stopped. So rather than calling MLINE() first for line 1, then for line 2, and so forth, by passing _MLINE as the third parameter and always asking for line 1 (after the current _MLINE position), MLINE() can work much faster. Listing 2 shows how to parse a string this way.

Listing 2. MLINE() lets you parse any string, not just a memo field. Combine it with the _MLINE system variable to speed things up.

```
LOCAL nOldMemoWidth
nOldMemoWidth = SET("Memowidth")
SET MEMOWIDTH TO 1024

nLines = MEMLINES(m.cOriginal)
_MLINE=0
DIMENSION aContents1[m.nLines]
FOR nLine = 1 TO m.nLines
   aContents1[m.nLine] = ;
      MLINE(m.cOriginal, 1, _MLINE)
ENDFOR

SET MEMOWIDTH TO &nOldMemoWidth
```

As the code indicates, MLINE() is sensitive to the current SET MEMOWIDTH value. In the example, I've set it to a very large value to ensure that the lines break only on line-break characters. While this version is more flexible and faster than the first, it's still quite slow, especially as the original string gets longer.

VFP 6 introduced the ALINES() function. It accepts an array and a string and breaks the string up into lines, putting one line in each array element. Not only does it reduce the code above to a single line (Listing 3), but it's blazingly fast. The only issue is that, in VFP 8 and earlier, arrays are limited to 65,000 elements, so you can't use ALINES() if the string could have more lines than that. That limit is gone in VFP 9.

Listing 3. ALINES() is the preferred method for parsing strings, unless you hit the array size limit.

```
nLines = ALINES(aContents2, ;
                m.cOriginal)
```

To compare the speed of the three approaches, I tested on 10 different string lengths from 8,500 characters to 85,000 (my test code is included in this month's downloads as BreakStringsIntoLines.

PRG). In each case, I ran 1,000 passes. The time for ALINES() grew linearly, that is, in proportion to the string length. Breaking an 85,000-character string into lines (2000 of them) 1000 times took under 2 seconds on my production machine.

The other approaches, using AT(), LEFT() and SUBSTR(), or using MLINE(), grew much faster than linearly. For the code in Listing 1, 1000 passes for 8,500 characters took only about 2.5 seconds, but for 85,000 characters, 1000 passes took almost 210 seconds. That is, as the string grew 10 times longer, parsing it took about 100 times as long.

The MLINE() approach in Listing 2 was faster than the AT()/SUBSTR() approach to begin with, and maintained that advantage, but still grew much faster than linearly. For an 8,500-character string, 1000 passes took about 1.3 seconds. For the 85,000-character string, it grew to more than 82 seconds.

Given these comparisons, it's clear that ALINES() is the way to go. But what if you want to parse into something other than lines?

Parsing based on contents

Historically, to break a string into words, or divide it up based on a separator character, we used AT() and SUBSTR(). The code was pretty much like Listing 1, except that we searched for the appropriate separator character or characters, not line break characters.

Today, there are several better ways to perform such a task. The approach to use depends on the exact type of parsing you need. First, despite its name, ALINES() is quite good for general parsing. In VFP 6, it could only handle lines, so to parse based on anything else you had to convert the separators into line breaks, as in Listing 4. You can, of course, break this code up into two lines, one to transform the commas into CHR(13)'s, and the second to call ALINES().

Listing 4. In VFP 6, to use ALINES() for anything other than lines, you had to use STRTRAN() first.

```
* Imagine that you want to break up a
* comma-separated string like
* "Red,Orange,Yellow,Green,Blue,Purple"
* contained in cString.
nItems = ALINES(aColors, ;
           STRTRAN(m.cString,",",CHR(13)))
```

In VFP 7, ALINES() got an additional parameter, the parse character. In fact, you can pass a whole list of parse characters, separated by commas. They're applied in the order they appear in the function call. So, the previous example can now be written as in Listing 5.

Listing 5. In VFP 7 and later, ALINES() can parse based on any characters you specify.

```
nItems = ALINES(aColors, m.cString, ",")
```

As with parsing into lines, the longer the original string, the more of an advantage ALINES() has over a loop. In my tests, with the 6-color string shown here, the loop took about 4 times as long as ALINES(). With a string containing around 6,000 items, the loop took 50 times as long.

A pair of functions added in VFP 7 are designed specifically for breaking strings into words. GetWordCount() tells you how many words are in a string, while GetWordNum() extracts a specific word. By default, they define words as separated by spaces, but you can pass a list of separators to indicate all the common ways to end words. GetWordCount() tells you how many words are in a string, while GetWordNum() returns a specified word. The example in Listing 6 shows how to extract each word in turn from a string. Normally, you'd have additional code inside the loop to do something with that word.

Listing 6. The GetWordNum() and GetWordCount() functions are designed specifically to break a string into words.

```
nWordCount = GetWordCount(cInputString)
FOR nWordNum = 1 TO nWordCount
   cCurWord =  GetWordNum(cInputString, ;
                          nWordNum)
ENDFOR
```

These two functions are really designed more for extracting specific words than for parsing whole strings, though. They're much slower than ALINES() and even slower than manually parsing with AT() and SUBSTR(). Like manual parsing, the slowdown is much more than linear. In my tests, parsing a 160-character string into words 10 times with these functions took less than .01 seconds, but for a string of more than 80,000 characters, 10 passes took more than 100 seconds. By comparison, using ALINES() with the optional parse characters, the time went from about .001 seconds for 160 characters to .08 seconds for the 80,000+-character string.

So it's best to reserve these functions for situations where you need to pull out only particular words from a string. The downloads include WordsOut.SCX, which demonstrates four approaches, and tests their speed.

Extracting portions of a string

Sometimes, rather than breaking a string up into its component lines or words, you need to retrieve some portion of the string based on either position or contents. The technique for extracting part of a string based on position hasn't changed over the years; use SUBSTR().

But extracting part of a string based on the string's content became much easier with the addition of the StrExtract() function in VFP 7. This function lets you specify delimiters that mark the ends of the substring you're interested in. It also accepts an optional parameter to indicate which occur-

rence of the delimiters you want. StrExtract() is ideally suited for retrieving information from XML or HTML strings, but can be used any time you have data in a structured format.

Like the other parsing examples, prior to VFP 7, this kind of task was done with AT() and SUBSTR(), like the code in Listing 7. Listing 8 shows the equivalent code using StrExtract(). There's no doubt that the second version is much easier to read and probably to maintain, as well. StrExtract() also provides an easier path to extracting the strings between all occurrences of the delimiter pair, with its optional nOccurrence parameter.

Listing 7. You can find a string between a pair of delimiters using AT() (or, in this case, ATC() to make the search case-insensitive) and SUBSTR().

```
cResult = ""
nStartPos = ATC(m.cStart, m.cInputString)
IF nStartPos > 0
   nEndPos = ATC(m.cStop, ;
      SUBSTR(m.cInputString, ;
      m.nStartPos + LEN(m.cStart)))
   IF nEndPos > 0
      cResult = SUBSTR(m.cInputString, ;
         m.nStartPos + LEN(m.cStart), ;
         m.nEndPos - 1 )
   ELSE
      * Grab rest of string
      cResult = SUBSTR(m.cInputString, ;
         m.nStartPos + LEN(m.cStart))
   ENDIF
ENDIF
```

Listing 8. StrExtract() makes the process much easier to read. The price is slower execution.

```
cResult = STREXTRACT(m.cInputString, ;
                     m.cSTart, m.cStop, 1, 1)
```

Unfortunately, StrExtract() is also a lot slower than manual parsing. In my tests, when finding all occurrences of a delimiter pair in a given string and extracting the text between each pair, StrExtract() was 1 to 2 orders of magnitude slower. That's enough of a difference to seriously consider dealing with the less readable version, particularly if you're going to use it repeatedly (obviously, test in your situation first and if you decide to go with the less readable code, do yourself a favor and write a wrapper function for it). One surprise in my testing was that passing the flag that makes StrExtract() case-insensitive speeds it up considerably. My form for testing these approaches is included in the downloads as ExtractString.SCX.

The bottom line

For parsing any string that uses a regular separator, ALINES() is always the way to go. It's fast and clear. For more complicated kinds of parsing, such as finding words or finding strings between delimiters, you need to test your particular situation and make a choice between more readable code and faster code.

Sidebar: Delimiters and Separators

Devlopers tend to toss the words "delimiter" and "separator" around interchangeably, and the VFP documentation doesn't help because it misuses them. However, delimiters and separators are actually two different things, and knowing the difference can be very helpful.

As its name suggests, a "separator" separates things. It comes between two items of some type. Probably the most common separator is the comma. It is used in between the items of many lists. ALINES() parses based on one or more separators.

"Delimiters" come in pairs and mark the ends of things. For example, we use pairs of quotation marks to delimit character strings. In HTML, we use angle bracket pairs to delimit tags. StrExtract() parses based on delimiters.

The worst confusion in VFP over the difference between delimiters and separators is in the various forms of APPEND FROM and COPY TO. Many of the text formats (like CSV and DELIMITED) actually use both delimiters and separators. For example, in CSV format, strings are delimited with quotes and all fields are separated by commas. Both the commands themselves and the Help file (while better than it used to be) get it wrong at least some of the time. The DELIMITED WITH CHARACTER clause actually specifies the separator. The DELIMITED WITH clause sometimes specifies a delimiter and sometimes specifies a separator.

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 nine books including the award winning Hacker's Guide to Visual FoxPro and Microsoft Office Automation with VisualFoxPro. Her most recent books are Taming Visual FoxPro's SQL and What's New in Nine: Visual FoxPro's Latest Hits. Her books are available from Hentzenwerke Publishing (www.hentzenwerke.com). Tamar is a Microsoft Certified Professional and a Microsoft Support Most Valuable Professional. In 2007, Tamar received the Visual FoxPro Community Lifetime Achievement Award. Tamar speaks frequently about Visual FoxPro at conferences and user groups in North America and Europe, including every FoxPro DevCon since 1993. You can reach her at tamar@thegranors.com or through www.tomorrowssolutionsllc.com.