We Used to Do it That Way, but …

Tamar E. Granor, Ph.D.
Tomorrow's Solutions, LLC
8201 Cedar Road
Elkins Park, PA 19027
Voice: 215-635-1958
Fax: 215-635-2234
Email: tamar@tomorrowssolutionsllc.com

Each new version of Visual FoxPro has brought new ways to do things, but breaking old habits is hard. Now that the VFP language has stabilized, it's time to hone our skills. This session examines some of the changes in the language over the years and demonstrates that learning to use the newer constructs can result in better, more maintainable, faster code.

I started working with FoxBase+ in 1989. In the nearly two decades since, the already capable Fox language has been expanded and enhanced, so that there are often several ways to accomplish the same thing.

Now that new development on VFP has ended, we finally have a chance to catch up and figure out which are the best ways to do the things we do. Often (though not always), newer approaches run faster and are easier to maintain.

The topics in this paper are divided into three broad topics: string manipulation, working with data, and programming techniques.

Most of the examples in this document are included in the session materials. In some cases, the example appears as a PRG file. More often, there’s a form that demonstrates the same point as the example.

String Manipulation

The importance of working with text strings has increased significantly in the last decade. When I started writing database applications, the reasons to parse and build strings were pretty few; parsing names into their component parts, and building complete addresses were the main text manipulation I faced.

The Web introduced lots more opportunities to work with text and text files, building and parsing HTML and XML strings. At the same time, the VFP team gave us lots more language constructs to make these tasks easier.

In this section, we’ll look at parsing and building file names, reading and writing text files, parsing strings, and converting and formatting data.

Parsing and Building File and Path Names

Taking apart a filename and putting it back together, perhaps varying the path or extension, is a fairly common task. You may need to put a file in a particular place or create a back-up copy of a file, using the same name but a different extension, or any of a number of similar tasks.

Filenames consist of a path, a stem and an extension. The path ends with the rightmost backslash. The extension begins after the rightmost period. Everything in between is the stem.

Clearly, with such a strictly defined structure, it's not hard to parse or build filenames, using FoxPro's powerful string manipulation language. You can use functions like AT() (or better yet, RAT()), LEFT(), RIGHT() and SUBSTR(), together with easy concatenation using the + operator.

However, there's an easier, more readable, way. Starting in VFP 6, the language includes a set of functions with names like JustPath() and ForceExt() that make file and path manipulation a piece of cake. Not only that, but these functions were in the FoxTools library at least as far back as FoxPro 2.5 for Windows.

Let's look at specific tasks to see why you should learn and use these functions. Before we start, let me define some terminology a little more clearly. In particular, the term "filename" is ambiguous.

A *path* is a sequence of folders, possibly beginning with a drive designator such as D:\Fox\VFP9. A path may or may not include a terminating backslash.

An *extension* is the portion of a filename that normally indicates the file type. In the Windows world, it's usually three characters. For example, DOC, PRG or JPG. There are a few commonly used four-character extensions, such as JPEG and HTML. The extension follows a period in the filename.

A *stem* or *filestem* is the main part of the filename that indicates the name of a particular file. For example, in VFP.EXE, the stem is VFP.

Filename can refer to either the stem plus the extension, such as VFP.EXE, or the complete reference including the path, such as D:\Fox\VFP9\VFP.EXE. To avoid this ambiguity, I'll use filename to refer to only stem plus extension, and use *full name* or *fully-pathed filename* to refer to the complete reference.

Constructing fully-pathed filenames

There are several ways you might want put parts together into full names. Probably, the most common is taking a path and a filename to create a fully-pathed filename.

In the examples here, assume that cPath contains the path and cFile contains the filename.

When handling this task manually, the main issue is ensuring you have the final backslash between the path and the stem. Listing 1 shows one way to do that.

Listing 1. To combine a filename and path, you need to check for the backslash.

```
cFullName = cPath
IF RIGHT(cPath, 1) <> "\"
   cFullName = m.cFullName + "\"
ENDIF
cFullName = m.cFullName + m.cFile
```

Of course, this is a common enough operation that you might want to write it in a single line and you can do that, thanks to the IIF() function, as shown in Listing 2.

Listing 2. Using IIF(), you can combine the filename and path in one line.

```
cFullName = m.cPath + IIF(RIGHT(m.cPath,1) = "\", "", "\") + m.cFile
```

Both approaches give the same result. However, in both cases, you have to take a good look at the code to figure out what it does. A more maintainable approach is to use the ForcePath() function. It accepts two parameters, a filename and a path, and returns a fully-pathed filename. Listing 3 shows the same example, this time using ForcePath().

Listing 3. With the ForcePath() function, you don’t have to worry about the backslash.

```
cFullName = FORCEPATH(m.cFile, m.cPath)
```

This version is so much more readable that I'd pay a small speed penalty to have it in my code, and it turns out that's what it costs, sort of. Specifically, in my testing, the five-line version of the old way takes about twice as long as the one-liner. The ForcePath() version takes about 30% longer than the one-liner. However, all three are so fast that speed is essentially a non-issue. On my production machine, a loop with 100,000 passes took between 0.1 and 0.2 seconds, depending on the version. So, unless you're doing intensive file processing with a need to build millions of filenames, speed shouldn't be a consideration. For readability and maintainability, the winner is clear.

ForcePath() is actually much more powerful that these examples demonstrate. In Forcing paths and extensions below, I'll show you how you can use it to avoid a whole lot of parsing.

Adding the backslash

The tedious step in the old versions of the previous example is ensuring that the path ends with a backslash. This is actually another thing we can do better with a built-in function. As the earlier code demonstrates, the old way to do this is with IF or IIF(), as in Listing 4:

Listing 4. The code to add the backslash to a path isn't much different from the code used when combining filename and path. You can do it with IF or IIF().

```
* More readable old version
cPathWithBackSlash = m.cPath
IF RIGHT(m.cPath, 1) <> "\"
   cPathWithBackSlash = m. cPathWithBackSlash + "\"
ENDIF

* One-line old version
cPathWithBackSlash = m.cPath + IIF(RIGHT(m.cPath,1) = "\", "", "\")
```

But the AddBS() function provides a more readable alternative shown in Listing 5.

Listing 5. Like ForcePath(), the AddBS() function makes the code much easier to read.

```
cPathWithBackSlash = ADDBS(m.cPath)
```

The timing for this one is interesting. If the path already has the backslash, the three approaches all take essentially the same time. Over 10,000 passes, the differences were thousandths of a second on my machine. However, if the path needs the backslash added, my tests found the one-liner and the AddBS() versions essentially identical, but the more verbose version, using IF, took about twice as long.

Adding an extension

Just as you might combine a path and a filename to get a fully-pathed filename, you can combine a filestem and an extension to get a filename. Here, the key issue in the old version is ensuring you have the period between the stem and the extension.

For these examples, assume cStem contains the filestem and cExt contains the extension.

Here's the old way, using concatenation. Although it's unlikely that the filestem includes the trailing period, the code in Listing 6 checks for it anyway. (For completeness, you might even want to check the extension for a leading period, as well, so you don't end up with two.)

Listing 6. The code to combine the filestem with an extension is similar to the code for combining filename and path.

```
cFile = m.cStem
IF RIGHT(m.cFile,1) <> "."
   cFile = m.cFile + "."
```

```
ENDIF
cFile = m.cFile + m.cExt
```

As with files and paths, there is a less readable, one-line of this, shown in Listing 7.

Listing 7. You can combine filestem and extension in one line, using IIF().

```
cFile = m.cStem + IIF(RIGHT(m.cFile,1)=".","",".") + m.cExt
```

But the ForceExt() function supersedes both versions, providing a clear, readable line of code, shown in Listing 8

Listing 8. ForceExt() gives you a single line, and it’s easy to read.

```
cFile = FORCEEXT(m.cStem, m.cExt)
```

Once again, the one-liner and the new way take about the same time, while the longer version using IF takes about 2.5 times as long. Also, as with ForcePath(), ForceExt() is much more powerful than this example shows. I'll show you where it really shines in Forcing paths and extensions below.

All of the examples so far may seem a little cooked, since we're often handed fully-pathed filenames that we need to manipulate. That usually involves parsing them into their components and, frequently, putting them back together differently. We'll look next at the parsing tasks.

Separating the path and filename

Probably the most common parsing task is separating the path from the filename. That is, given a fully-pathed filename, extract the path and the filename into separate variables. Sometimes, you only need one or the other, of course. For these examples, we'll assume cFullName contains a fully-pathed filename.

Using VFP's string functions, you can separate the path and filename as in Listing 9.

Listing 9. Parsing out the path and filename isn’t hard, but it’s not immediately obvious what the code is doing.

```
* The old way
LOCAL nPathEnds
nPathEnds = RAT("\", m.cFullName)
cPath = LEFT(m.cFullName, m.nPathEnds-1)
cFile = SUBSTR(m.cFullName, m.nPathEnds + 1)
```

That's not too much code, but it certainly doesn't tell you what it's doing. The JustPath() and JustFName() functions let you understand the code immediately. Listing 10 shows the new way.

Listing 10. The JustPath() and JustFName() functions make the goal clear.

```
cPath = JUSTPATH(m.cFullName)
cFile = JUSTFNAME(m.cFullName)
```

The timing for this code surprised me. I expected the new way to be much faster. In fact, the comparison depends on the length of the path. With no path, just a filename, the new way is the same or faster. After that, the longer the path, the faster the old way is in comparison to the new. When I saw this result, I expected it to be a function of the number of levels (how many folders to traverse), but in fact, it seems to be based on the number of characters. Regardless, all of this is quite fast. The slowest test I saw took less than 0.4 seconds for 10,000 passes.

Finding the extension

Another common task is grabbing the extension from a filename, whether there's a path or not. This allows you to find out what kind of file you're dealing with. As with the path, sometimes you want to grab both filestem and extension, while at others, you need only the extension.

Listing 11 shows the old way to grab just the extension:

Listing 11. Extracting the extension also isn't difficult, but the code needs to be commented to make it clear.

```
LOCAL nPeriodAt

nPeriodAt = RAT(".", m.cFullName)
IF nPeriodAt = 0
   cExtension = ""
ELSE
   cExtension = RIGHT(m.cFullName, LEN(m.cFullName) - m.nPeriodAt)
ENDIF
```

The new way uses the JustExt() function, as in Listing 12.

Listing 12. With JustExt(), the code is short and sweet.

```
cExtension = JUSTEXT(m.cFullName)
```

Interestingly, in this case, the new way is faster, except in the case of a file with a long path and no extension. I suspect what's going on there is that JustExt() is walking backward from the end of the string looking for a period and not finding one. In all the other cases I tested, using JustExt() took from one-half to two-thirds as long as the old way. As with all the other tests, though, both versions were so fast as to make the question moot. In this case, my tests performed 10,000 passes in under 0.2 seconds.

Extracting just the stem

The last in the set of "pulling filenames apart" tasks is extracting just the filestem, the part in between the path and the extension. Like the other tasks, the old way involves parsing to find the final backslash and the final period, while the new way is a single function call, in this case, to JustStem().

The code for the old way is more complicated than the other cases because it has to deal with filenames with no path, filenames with no extension, filenames with neither and filenames with both. Listing 13 demonstrates.

Listing 13. Pulling out only the filestem is much more complicated than extracting any of the other parts.

```
LOCAL nPathEnds, nPeriodAt

nPathEnds = RAT("\", m.cFullName)
nPeriodAt = RAT(".", m.cFullName)
DO CASE
CASE nPathEnds = 0 AND nPeriodAt = 0
   * All we have is the stem
   cFileStem = m.cFullName

CASE nPathEnds = 0
   * No path, just filename
   cFileStem = LEFT(m.cFullName, m.nPeriodAt-1)

CASE nPeriodAt = 0
   * No extension
   cFileStem = SUBSTR(m.cFullName, m.nPathEnds + 1)

OTHERWISE
   * Have all parts, extract
   nFullLen = LEN(m.cFullName)
   nExtLen = nFullLen - m.nPeriodAt
   nStemLen = m.nFullLen - m.nPathEnds - m.nExtLen - 1
   cFileStem = SUBSTR(m.cFullName, m.nPathEnds + 1, m.nStemLen)

ENDCASE
```

From a code point of view, it's easy to see why the new way in Listing 14 is an improvement.

Listing 14. With JustStem(), getting the filestem is a one-liner.

```
cFileStem = JUSTSTEM(m.cFullName)
```

Not surprisingly, given how much code is involved in the old way, the new way was faster in every case I tested. The factor ranged from half the time of the old way down to a quarter of the time.

The full monty

Of course, what you sometimes want is to combine all of this and pull a filename into its component parts. Since getting the filestem is the most complex using the old way, my code for the complete parse in Listing 15 is based on that and simply adds the code to set the path and extension to each of the four cases.

Listing 15. Completely parsing a filename isn't much different than extracting the filestem.

```
LOCAL nPathEnds, nPeriodAt

nPathEnds = RAT("\", m.cFullName)
nPeriodAt = RAT(".", m.cFullName)
DO CASE
CASE nPathEnds = 0 AND nPeriodAt = 0
   * All we have is the stem
   cFileStem = m.cFullName
   cPath = ""
   cExtension = ""

CASE nPathEnds = 0
   * No path, just filename
   cFileStem = LEFT(m.cFullName, m.nPeriodAt-1)
   cPath = ""
   cExtension = RIGHT(m.cFullName, LEN(m.cFullName) - m.nPeriodAt)

CASE nPeriodAt = 0
   * No extension
   cFileStem = SUBSTR(m.cFullName, m.nPathEnds + 1)
   cPath = LEFT(m.cFullName, m.nPathEnds - 1 )
```

```
OTHERWISE
   * Have all parts, extract
   nFullLen = LEN(m.cFullName)
   nExtLen = nFullLen - m.nPeriodAt
   nStemLen = m.nFullLen - m.nPathEnds - m.nExtLen - 1
   cFileStem = SUBSTR(m.cFullName, m.nPathEnds + 1, m.nStemLen)
   cPath = LEFT(m.cFullName, m.nPathEnds - 1 )
   cExtension = RIGHT(m.cFullName, LEN(m.cFullName) - m.nPeriodAt)
ENDCASE
```

The new way in Listing 16 is just three lines.

Listing 16. The JustXXX() functions make short work of completely parsing a filename.

```
cFileStem = JUSTSTEM(m.cFullName)
cPath = JUSTPATH(m.cFullName)
cExtension = JUSTEXT(m.cFullName)
```

My tests show the new way to be faster, except when there's a long path and no extension. No doubt that's the same slowdown encountered when extracting the extension from such a filename.

Forcing paths and extensions

Finally, it's time to look at the real power of ForcePath() and ForceExt(). In earlier examples, we assumed that the filename didn't include a path or extension, respectively. In fact, these functions do as their names suggest, and remove an existing path or extension before adding the new one.

So, using ForcePath(), you can start with a fully-pathed filename and end up with a fully-pathed filename pointing to a different path. Similarly, ForceExt() lets you start with a filename (with or without path) including extension and returns a filename that's the same except that the extension has been changed.

Listing 17 shows the old way of forcing a path.

Listing 17. Replacing one path with the other requires both parsing and combining.

```
nPathEnds = RAT("\", m.cFullName)
cFile = SUBSTR(m.cFullName, m.nPathEnds + 1)

cNewName = cNewPath
IF RIGHT(cPath, 1) <> "\"
   cNewName = m.cNewName + "\"
ENDIF
cNewName = m.cNewName + m.cFile
```

Listing 18 shows the new way, once again, a one-liner.

Listing 18. ForcePath() lets you change the path with one function call.

```
cNewName = FORCEPATH(m.cFullName, m.cNewPath)
```

In my tests, the new way takes from one-third to one-half the time of the old way. The old way takes about the same time no matter what the original filename is, and whether or not it has a path. The new way varies with the length of the original path.

The old way of forcing an extension is shown in Listing 19.

Listing 19. Changing extensions is a lot like changing paths.

```
nPeriodAt = RAT(".", m.cFullName)
IF nPeriodAt = 0
   cNewName = m.cFullName
ELSE
   cNewName = LEFT(m.cFullName, m.nPeriodAt - 1)
ENDIF
cNewName = m.cNewName + "." + m.cNewExtension
```

Listing 20 shows the new way.

Listing 20. ForceExt() makes changing extensions simple.

```
cNewName = FORCEEXT(m.cFullName, m.cNewExtension)
```

Again, the new way is faster across the board. As in the other tests, the slowest case is with a long path and no extension, but even in that case, the new way took only about two-third the time of the new way.

Getting it right

As the examples demonstrate, using the newer functions is almost always faster, and is certainly more readable. It's also more likely to be right. When writing parsing code, it's easy to have subtle mistakes. One application I worked on did a lot of parsing and building file names, and seemed to work quite well. Then, my client handed me some new test data, involving folder names that included periods, and the application started bombing. The code for parsing file names and paths didn't consider the possibility of a period in the path. Replacing that code with appropriate use of the JUSTx() and FORCEx() functions fixed the problem.

The materials for this session include FileAndPath.SCX, which demonstrates the JUSTx() functions; BuildFileName.SCX, which demonstrates the FORCEx() functions, and TimingPathAndFileFunctions.prg, which performs a variety of timing tests related to this functions.

Reading and writing text files

When the low-level file functions (LLFFs) were introduced in FoxPro 1.0, I was intimidated by their name. "Low-level" sounded like something for the same people who used the LCK (Library Construction Kit) to build FLLs, not for mere mortals like me. Eventually, I needed to work with some external files and I learned that, in fact, the LLFFs were pretty much the same kind of mechanism I'd used to read data in other languages.

Even so, working with the LLFFs is tedious. You have to open the file with the right function, hang onto the handle that function returns, and make a series of function calls to actually read the data. Listing 21 shows code that reads the contents of a text file into a variable.

Listing 21. To read a text file with the low-level file functions, you open it and get a handle, then loop through until you run out of file.

```
* This code reads the file in blocks of
* 254 characters
nHandle = FOPEN(m.cFileName)
IF m.nHandle <> -1
   cContents = ""
   DO WHILE NOT FEOF(m.nHandle)
```

```
      cContents = m.cContents + FREAD(m.nHandle, 254)
   ENDDO
   FCLOSE(m.nHandle)
ENDIF
```

Writing a text file is a little simpler because the FWRITE() function accepts the length of the string as a parameter, and can handle arbitrarily large strings. Listing 22 shows one way to write a text file with the LLFFs.

Listing 22. Writing a text file with the low-level file functions is simpler than reading one.

```
nHandle = FCREATE(m.cFileName)
IF m.nHandle <> -1
   nResult = FWRITE(m.nHandle, m.cContents, LEN(m.cContents))
ENDIF
FCLOSE(m.nHandle)
```

While these approaches work, VFP 6 introduced a pair of functions that virtually eliminate the need to use the LLFFs: FileToString() and StrToFile(). As their names suggest, they read a file into a string and write a string to a file, respectively. They also convert the code blocks above into single lines of code. Listing 23 shows how to use FileToString(), while Listing 24 demonstrates StrToFile().

Listing 23. With FileToString(), reading in a text file takes just one line.

```
cContents = FILETOSTR(m.cFileName)
```

Listing 24. StrToFile() turns writing a text file into a one-liner.

```
nResult = STRTOFILE(m.cContents, m.cFileName, .F.)
```

Why switch?

If you need to read text files regularly, by now, you’ve probably created your own wrappers for the LLFFs, so that you can read and write text files with a single call, so why would you switch to the newer functions?

For reading files, the answer is simple: speed. I tested the loop in Listing 21 against the single line in Listing 23 on a file with 710,000 characters. My test read the file into memory 1000 times. For the LLFFs, I also tested with a variety of block sizes (the second parameter to FREAD()) from 254 bytes to 2540. While a larger block size made a difference (with the largest block size, 1000 passes took about 80% of the time as with the smallest block size), FileToStr() was more than 30 times faster than the fastest LLFF attempt.

The speed advantage of FileToStr() also varies with the size of the target file. For tiny files, FileToStr() has almost no advantage, but even in the vicinity of 40KB files, FileToStr() is about 30% faster than the LLFFs.

Listing 25 shows my code for testing read speed (included in the session materials as LLFFvsFileToStr.prg.

Listing 25. FileToStr() is just about always faster than reading with the LLFFs. For large files, it’s an order of magnitude or more faster.

```
* Compare LLFF with FileToStr()
```

```
#DEFINE PASSES 1000

* For LLFF, have to open file and read one
* line at a time.

LOCAL cFileName, nHandle, cContents
LOCAL nStart, nEnd, nPass, nBlockPass, nBlockSize

cFileName = GETFILE("TXT;LOG")

* Try different block lengths
FOR nBlockPass = 1 TO 10
   nBlockSize = m.nBlockPass * 254
   nStart = SECONDS()

   FOR nPass = 1 TO PASSES
      nHandle = FOPEN(m.cFileName)
      IF m.nHandle <> -1
         cContents = ""
         DO WHILE NOT FEOF(m.nHandle)
            cContents = m.cContents + FREAD(m.nHandle, m.nBlockSize)
         ENDDO
         FCLOSE(m.nHandle)
      ENDIF
   ENDFOR
   nEnd = SECONDS()

   ? " Using LLFF, result has ", LEN(m.cContents), " characters"
   ? " With block size = ", m.nBlockSize, ", total time = ", nEnd - nStart
ENDFOR
* For FileToStr(), one-liner
nStart = SECONDS()

FOR nPass = 1 TO PASSES
   cContents = FILETOSTR(m.cFileName)
ENDFOR
nEnd = SECONDS()

? " Using FILETOSTR(), result has ", LEN(m.cContents), " characters"
? " Total time = ", nEnd – nStart
```

For writing text files, the case is murkier. For small files, StrToFile() is about a third faster than the LLFFs. However, when the string to write is more than 327,680 (which is 320 * 1024) characters, StrToFile() has a significant slowdown, and the LLFFs are faster. According to VFP MVP Christof Wollenhaupt, the difference is in the way VFP translates the calls to API calls. He says that StrToFile uses:

> “a single call to the WriteFile() API function passing the string as a parameter.
>
> “Hence, I assume that this is an issue with the API or a driver. VFP makes synchronous API calls. What I guess is happening behind the scenes is that the driver stores data in a buffer and then performs an asynchronous file operation. When the content exceeds the size of the buffer, the driver would have to complete the first operation, before adding the second part to the async buffer. Alternatively, large blocks might be written with a lower priority as to not to slow down the system too much.

> "You don't see the same issue with FWRITE(), because FWRITE() splits the string into blocks of 0x20000 bytes and is therefore always below the limit. The single FWRITE() call results in three calls to the WriteFile API function."

Breaking the string up into pieces no more than 327,680 characters and issuing multiple calls to StrToFile() doesn't improve matters. (Given Christof's explanation, this makes sense.) In fact, in my tests, that approach was slower than the single call. The bottom line, therefore, is that if you might be writing large text files, you may want to stick with the LLFFs. If that's your choice, wrapping the write process up into a single function method is a good idea.

However, my tests show that once the file exists, adding text to it (with the optional append flag) is not significantly slower than appending text to a file below the limit.

Listing 26 shows my code for testing write speed (LLFFvsStrToFile.PRG in the session materials).

Listing 26. StrToFile() has only a small advantage over LLFFs for small to medium files. For large files, FWRITE() is a better choice.

```
* Compare LLFF with StrToFile()
#DEFINE PASSES 100

LOCAL cFileName, cContents, nHandle
LOCAL nPass, nSTart, nEnd

cFileName = FORCEPATH("TestOutput.TXT", SYS(2023))
SET SAFETY OFF

SET ALTERNATE TO StrToFileTiming.TXT
SET ALTERNATE ON

FOR nLength = 1 TO 10
   cContents = REPLICATE("Now is the time for all good men " + ;
                         "to come to the aid of their country." ;
                         + CHR(13) + CHR(10), 1000 * m.nLength)

   ? "Length of test string: ", LEN(m.cContents)

   * For LLFF, have to create file and then send a bit at a time
   nStart = SECONDS()
   FOR nPass = 1 TO PASSES
      nHandle = FCREATE(m.cFileName)
      IF m.nHandle <> -1
         nResult = FWRITE(m.nHandle, m.cContents, LEN(m.cContents))
      ENDIF
      FCLOSE(m.nHandle)
   ENDFOR
   nEnd = SECONDS()

   ? " With LLFF, characters written: ", m.nResult
   ? " Total time: ", m.nEnd - m.nStart

   * For StrToFile(), one line
   nStart = SECONDS()

   FOR nPass = 1 TO PASSES
      nResult = STRTOFILE(m.cContents, m.cFileName, .F.)
   ENDFOR
   nEnd = SECONDS()

   ? " With STRTOFILE(), characters written: ", m.nResult
   ? " Total time: ", m.nEnd - m.nStart

   * Try StrToFile() in loop, if past the magic number
   IF LEN(m.cContents) > 327680
      LOCAL nPart, cPart
      nStart = SECONDS()

      FOR nPass = 1 TO PASSES
         DELETE FILE (m.cFileName)
         FOR nPart = 1 TO CEILING(LEN(m.cContents)/327680)
            cPart = SUBSTR(m.cContents, (nPart-1) * 327680 + 1, 327680)
            nResult = STRTOFILE(m.cPart, m.cFileName, .T.)
         ENDFOR
      ENDFOR
      nEnd = SECONDS()

      ? " With STRTOFILE() in a loop, characters written: ", m.nResult
      ? " Total time: ", m.nEnd - m.nStart
   ENDIF
ENDFOR

SET ALTERNATE off
SET ALTERNATE TO
SET SAFETY ON
```

In addition to the two timing tests, the session materials include TextFileIO.SCX, which lets you test on files of your choosing.

Breaking strings up

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 was reduced to a single function call.

Parsing into lines

Perhaps the most common 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 27 does the job. However, this code is inflexible, since it looks only for a CHR(13) + CHR(10) combination to end the line. It's also slow; see later in this section for a discussion of timing.

Listing 27. 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 28 shows how to parse a string this way.

Listing 28. 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
```

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, 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 29. ALINES() is the preferred method for parsing strings, unless you could 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 the session materials 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 27, 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 28 was faster than the AT()/SUBSTR() approach to being, 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 27, except that we searched for the appropriate separate character or characters, not line break characters.

Today, there are several better ways to perform such a task. Which 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 30. (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 30. 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 31.

Listing 31. 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.

The one problem with ALINES() is that you have to know what the separators are. While that's no problem in many cases, if you want to break a string into words, the list can be quite long. (At a minimum, you'd need space, comma, colon, semi-colon, quotation mark, question mark, exclamation point, and dash.) A pair of functions added in VFP 5 is designed to address this need. GetWordCount() tells you how many words are in a string, while GetWordNum() returns a specified word. The example in Listing 32 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 32. The GetWordNum() and GetWordCount() functions are designed specifically to break a string into words. They know about all the normal separators between 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 .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 session materials 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 occurrence 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 33. Listing 34 shows the equivalent code using StrExtract(). There's no doubt that the second version is much easier to read and likely, to maintain, as well. StrExtract() also provides an easier path to extracting the strings between all occurrences of the delimiter pair, with its option nOccurrence parameter.

Listing 33. 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 34. 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 sped it up considerably. My form for testing these approaches is included in the session materials as ExtractString.SCX.

Building strings

Of course, not only do we need to break strings up into lines or find things within strings. We often want to put strings together, whether we're building HTML or XML strings, constructing SQL commands, or combining the parts of an address. We'll look at two issues: combining text and data, and converting between data types.

Combining text and data

FoxPro has had the ability to combine character strings, using the + operator, since its earliest days. But for some string construction, the textmerge facility, added in FoxPro 2.0 and enhanced several times since, is a better choice.

Textmerge lets you combine plain text with expressions to be evaluated to create a single character output. For example, say you have a table with three fields, cText, nAmount and dDate and you want to create a string with one field per line. Using VFP's string manipulation functions, you could do it with code like Listing 35.

Listing 35. FoxPro has had tools for combining strings from the early days. This line combines three fields into a single text block with one field per line.

```
cBlock = cText + CHR(13) + CHR(10) + ;
         TRANSFORM(nAmount) + CHR(13) + CHR(10)  + ;
         TRANSFORM(dDate)
```

Textmerge makes the code a lot more readable. Listing 36 shows code equivalent to Listing 35.

Listing 36. With textmerge, you don't have to add codes for line breaks or convert data to character.

```
TEXT TO m.cBlock TEXTMERGE NOSHOW
<<cText>>
<<nAmount>>
<<dDate>>
ENDTEXT
```

The example demonstrates two big advantages of textmerge. First, you don't have to use ASCII codes for line breaks; they're added automatically when you put a line break inside the TEXT block. Second, it handles most data types transparently, so you don't have to convert from numeric or date to character.

Textmerge was originally added to support the Screen and Menu Designer tools added at the same time and was used in the generator programs that came with those tools. In fact, you can still see it at work in GenMenu.PRG, the program that converts menu files (MNX) into menu programs (MPR).

In its early form, using textmerge was a little cumbersome. You had to SET TEXTMERGE TO a file to hold the result, and then SET TEXTMERGE ON to turn the process on. Then, any line of code preceded by \ or \\ was considered output to the textmerge file. (You used \ to start a new output line and \\ to continue on the current line.) Anything inside the textmerge delimiters (by default, << and >>) was evaluated, with the result converted to character and placed on the line. You could also put whole blocks of text into the output, using the TEXT/ENDTEXT pair; again, anything inside the textmerge delimiters was evaluated before outputting it. Finally, when you were done, you issued SET TEXTMERGE OFF and SET TEXTMERGE TO without a file name to stop the process and close the file.

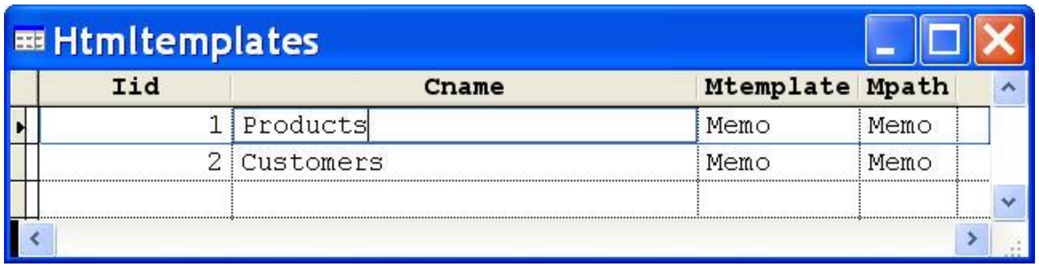
In VFP 7, with the need to build HTML and XML strings in mind, the textmerge system was enhanced to make it easier to use. The TEXT command gained a number of additional keywords. The most important additions were TEXTMERGE, which meant you could evaluate strings inside without issuing SET TEXTMERGE ON first, and TO, which let you send the results to a variable. (SET TEXTMERGE also gained the ability to send results to a variable in VFP 7.) The result of these changes is that there's little, if any, reason to use the SET TEXTMERGE commands anymore. Even if you want to send the results to a file, it may be easier to issue TEXT TO a variable, and then using StrToFile() to create the file.

A lot of people like to use textmerge to construct SQL commands, especially those being sent to a back-end server via SQL pass-through. Listing 37 shows a simple example; more often, the variables to be expanded would be based on choices made by a user.

Listing 37. Textmerge can make construction of SQL commands easier to read. The command is on a single line, though it wraps here.

```
* Normally, these would be constructed based on user inputs.
cWhere = [WHERE Country = "UK"]
cOrderBy = [ORDER BY City, CompanyName]
cFields = [CompanyName, ContactName, City, Phone]

* Now, use textmerge to build the query
TEXT TO cSQL TEXTMERGE NOSHOW
SELECT <<m.cFields>> FROM Customers <<m.cWhere>> <<m.cOrderBy>> INTO CURSOR
Result
ENDTEXT
```

VFP 7 also introduced a TextMerge() function that lets you perform textmerge on a string in a single line. This is most useful when you have a stored string containing expressions to evaluate, as you might when building web pages dynamically. For example, consider a table like the one in Figure 1 (included in the session materials as HTMLTemplates.DBF). The mTemplate memo field contains blocks like the one in Listing 38. Then, using code like Listing 39, you can generate an HTML file. Figure 2 shows the resulting HTML, displayed in the web browser control. That form is included in the session materials as BuildHTML.SCX.



Iid	Cname	Mtemplate	Mpath
1	Products	Memo	Memo
2	Customers	Memo	Memo

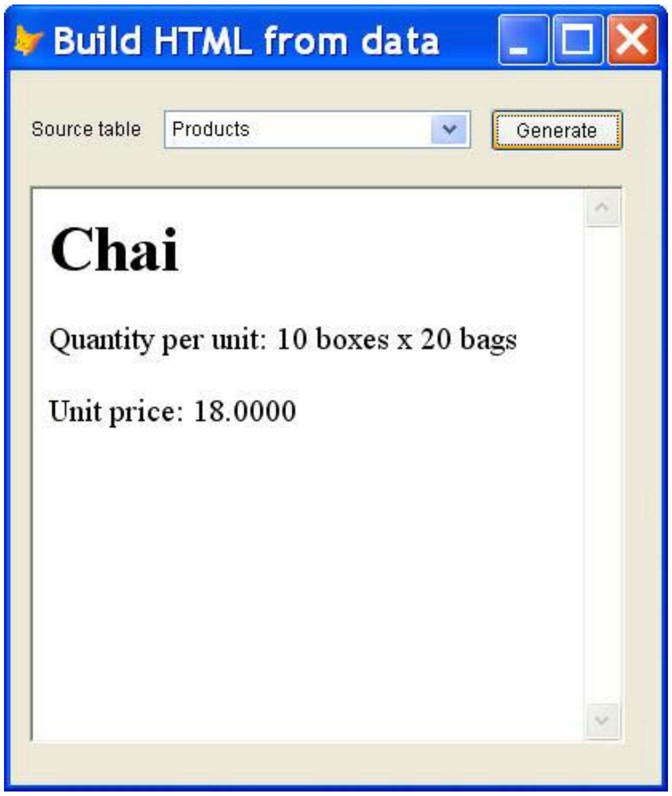
Figure 1. This table stores templates for creating HTML from a variety of tables.

Listing 38. You can incorporate VFP fields in HTML templates like this one, to be merged with the TextMerge() function.

```
<html>
<head>
</head>
<body>
<h1><<alltrim(productname)>></h1>
<p>Quantity per unit: <<quantityperunit>></p>
<p>Unit price: <<Unitprice>></p>
</body>
</html>
```

Listing 39. The TextMerge() function provides an easy way to add data to HTML.

```
IF SEEK(m.cTable, "HTMLTemplates", "cName")

   SELECT 0
   cTableWithPath = FORCEPATH(m.cTable,EVALUATE(HTMLTemplates.mPath))
   USE (m.cTableWithPath) ALIAS __Source

   cHTML = TEXTMERGE(HTMLTemplates.mTemplate)

   USE IN __Source

   This.cHTMLFile = FORCEPATH(m.cTable + "Record.HTM", SYS(2023))

   STRTOFILE(m.cHTML, This.cHTMLFile)
ENDIF
```



Chai

Quantity per unit: 10 boxes x 20 bags

Unit price: 18.0000

Figure 2. The HTML build via TextMerge() is like any other HTML and can be displayed in browser, or using the web browser control, as in this form.

Type conversions

Although textmerge, described in the preceding section, lets you avoid type conversions, some work involving strings does require you to change between character and other data types. VFP has provided facilities for these conversions since the early days, but as with so much else, the task has become easier over time.

In earlier versions, converting from other types to character involved a number of different functions, including DTOC(), TTOC(), PADL() and PADR(). Listing 40 shows a function created for this purpose by Andy Kramek and Marcia Akins. (The function and the corresponding Str2Exp later in this section are from “1001 Things You Wanted to Know about Visual FoxPro” and are included here with permission of the authors). This particular function is focused on creating strings that can be used in SQL pass-through commands, so it does things differently than a function aimed at generating output, but it demonstrates the key ideas.

Listing 40. To convert any expression to a string in earlier versions, you needed to figure out what type it was and apply the appropriate conversion.

```
**********************************************************************
* Program....: Exp2Str
* Compiler...: Visual FoxPro 06.00.8492.00 for Windows
* Abstract...: Returns the passed expression a either a quoted or
* ...........: unquoted string depending on its data type in order to
* ...........: build SQL where clauses on the fly
* From "1001 Things You Wanted to Know about Visual FoxPro
* Included courtesy of Marcia Akins
```

```
**********************************************************************
FUNCTION Exp2Str( tuExp, tcType )
*** Convert the passed expression to string
LOCAL lcRetVal, lcType
*** If no type passed -- map to expression type
lcType=IIF( TYPE( 'tcType' )='C', UPPER( ALLTRIM( tcType ) ), TYPE( 'tuExp' ))
*** Convert from type to char
DO CASE
  CASE INLIST( lcType, 'I', 'N' ) AND INT( tuExp ) = tuExp      && Integer
    lcRetVal = ALLTRIM( STR( tuExp, 16, 0 ) )
  CASE INLIST( lcType, 'N', 'Y' )                               && Numeric or Currency
    lcRetVal = ALLTRIM( PADL( tuExp, 32 ) )
  CASE lcType = 'C'                                             && Character
    lcRetVal = '"' + ALLTRIM( tuExp ) + '"'
  CASE lcType = 'L'                                             && Logical
    lcRetVal = IIF( !EMPTY( tuExp ), '.T.', '.F.')
  CASE lcType = 'D'                                             && Date
    lcRetVal = '"' + ALLTRIM( DTOC( tuExp ) ) + '"'
  CASE lcType = 'T'                                             && DateTime
    lcRetVal = '"' + ALLTRIM( TTOC( tuExp ) ) + '"'
  OTHERWISE
    *** There is no otherwise unless, of course, Visual FoxPro adds
    *** a new data type. In this case, the function must be modified
ENDCASE
*** Return value as character
RETURN lcRetVal
```

Before looking at an easier alternative to this function, let’s take a brief look at the PADL() function it uses for converting numeric data. PADL() is one of a set of three functions that pads strings to a specified length; the other two are PADR() and PADC(). The three differ in where they put the padding. PADL() pads on the left, PADR() pads on the right, and PADC() pads on both sides to center the original data.

The PADx() functions have two really useful features. First, they accept data of almost any time and do an implicit conversion to character. So, in Exp2Str, PADL() is used to convert numeric data types to character. Second, they have an optional third parameter that specifies the pad character. By default, the result is padded with spaces. However, these functions can pad with anything you want, so you can use them, for example, to build strings of digits with leading zeroes, or to create “trailers” of dots as you’d find in a table of contents or index. Listing 41 demonstrates by left-padding a number with zeroes.

Listing 41. The PADx() functions not only convert the first parameter to character, but can pad with any character.

```
? PADL(324, 10, "0")  && results in "0000000324”
```

For many purposes, the TRANSFORM() function makes code like Exp2Str() obsolete. It accepts a value of almost any type and returns the value as a string. The conversion provides the format you want in many cases. When it doesn’t, you can pass a format string (essentially, a combination of the Format and InputMasks properties) to indicate the desired output. Listing 42 shows how to use TRANSFORM() to convert a variety of data items.

Listing 42. Performing type conversions is easy with TRANSFORM.

```
? "TRANSFORM(123.45) = ", TRANSFORM(123.45)
? "TRANSFORM($123.45) = ", TRANSFORM($123.45)
? "TRANSFORM($123.45, '$9999999.9999') = ", ;
```

```
  TRANSFORM($123.45, "$9999999.9999") && to avoid rounding
? "TRANSFORM(.T.) = ", TRANSFORM(.T.)
? "TRANSFORM(DATE()) = ", TRANSFORM(DATE())
? "TRANSFORM(DATETIME()) = ", TRANSFORM(DATETIME())
```

TRANSFORM() without a format code has been available since VFP 6. It took a lot longer to get a generic way to go from character to the other types. Until VFP 9, to turn a string into a numeric or date or datetime value, you had to use the right function, such as VAL(), CTOD(), TTOD(). Listing 43 shows Andy and Marcia's version of such a function.

Listing 43. Prior to VFP 9, going from character to another type called for a function like this one.

```
**********************************************************************
* Program....: Str2Exp
* Compiler...: Visual FoxPro 06.00.8492.00 for Windows
* Abstract...: Passed a string and a data type, return the expression
* ..........: after conversion to the specified data type
* From "1001 Things You Wanted to Know about Visual FoxPro"
* Included courtesy of Marcia Akins
**********************************************************************
FUNCTION Str2Exp( tcExp, tcType )
*** Convert the passed string to the passed data type
LOCAL luRetVal, lcType

*** Remove double quotes (if any)
tcExp = STRTRAN( ALLTRIM( tcExp ), CHR( 34 ), "" )
*** If no type passed -- display error message
*** the procedure is not clairvoyant
IF TYPE( 'tcType' ) = 'C'
   lcType = UPPER( ALLTRIM( tcType ) )
ELSE
   *** Type is a required parameter. Let the developer know
   ERROR 'Missing Parameter: Expression type is a required parameter to
Str2Exp'
ENDIF
*** Convert from Character to type
DO CASE
  CASE INLIST( lcType, 'I', 'N' ) AND INT( VAL( tcExp ) ) == VAL( tcExp ) &&
Integer
    luRetVal = INT( VAL( tcExp ) )
  CASE INLIST( lcType, 'N', 'Y')                       && Numeric or Currency
    luRetVal = VAL( tcExp )
  CASE INLIST( lcType, 'C', 'M' )                      && Character or memo
    luRetVal = tcExp
  CASE lcType = 'L'                                    && Logical
    luRetVal = IIF( !EMPTY( tcExp ), .T., .F. )
  CASE lcType = 'D'                                    && Date
    luRetVal = CTOD( tcExp )
  CASE lcType = 'T'                                    && DateTime
    luRetVal = CTOT( tcExp )
  OTHERWISE
    *** There is no otherwise unless, of course, Visual FoxPro adds
    *** a new data type. In this case, the function must be modified
ENDCASE
*** Return value as Data Type
RETURN luRetVal
```

VFP 9 introduced the CAST() function, which provides a single way to convert between any compatible data types. CAST() accepts a single parameter with a rather unusual format and returns a value of a specified type. The syntax for CAST()'s parameter is:

```
uExpression AS cType [(nSize [, nDecimals] )]
```

In other works, you specify an expression, a type and, optionally, a size. The format for the type and size is the same as in the CREATE TABLE or CREATE CURSOR command. Listing 44 shows some examples. CAST() can handle a wide range of types, not just the obvious combinations. For example, trying to convert a numeric value to a logical results in 0 being .F. and all other values being .T.; that seems as good a choice as any. CAST() does fail on some conversions such as converting a date to logical or numeric.

Listing 44. With CAST(), you can do all your conversions with a single function; you just have to know the type and size you want.

```
? [CAST("123.45" as N(6,2)) = ], CAST("123.45" as N(6,2))
? [CAST("$123.45" as Y) = ], CAST("$123.45" as Y)
? [CAST("^2008-2-8" as D) = ], CAST("^2008-2-8" as D)
? [CAST("^2008-2-8 15:01" as T) = ], CAST("^2008-2-8 15:01" as T)
```

Both TRANSFORM() and CAST() can convert from other types to character, so which one should you use for that purpose? The answer depends on your needs. When you want a string of a particular size, use CAST(). When you need a string in a specific format, use TRANSFORM(). If you don’t know how big the result could be, also use TRANSFORM().

Working with data

VFP is all about data, so it’s not surprising that techniques for working with data have improved over time. The improvements include ways of dealing with work areas and aliases, temporary data, and primary keys, as well as a host of ways of working with dates.

Addressing work areas

In dBase II, you could open two tables simultaneously. Each occupied one work area. By the time I entered the Xbase world with FoxBase+, there were 10 work areas; most of the time, that was enough, but now and then, I found it limiting. FoxPro had 25 work areas in the standard version and 225 with the extended version; I'm not sure I ever found myself short of work areas, even in the standard version. When VFP shipped with 32,767 work areas per data session, I knew I'd never worry about available work areas again.

When there were only two work areas or even 10, keeping track of what table was open in which work area was no problem. But as soon as there were more work areas than I could keep track of in my head, I started making sure I didn't need to know where a given table was open.

The tools for letting us ignore work areas have gotten better and better over the years. There are three related issues. The first is to open a table without having to figure out what work area to use. The second is addressing an open table without knowing what work area it's in. The third is writing code without having to worry about what work area is current. Combining the three means you'll hardly ever need to worry about stepping on an open table or acting on the wrong table again.

Use aliases, not work area numbers or letters

Every time you open a table in VFP, the open table occupies a work area. As noted above, in VFP (all versions from 3 to 9), each data session offers 32,767 work areas. Work areas are numbered from 1 to 32,767. The first 10 work areas can also be referenced by the letters A through J.

When you USE a table, by default, it opens in the current work area. If there's already a table open in that work area, the open table is closed. Once a table is open in a given work area, you can refer to the work area by the alias of the open table. This is always a better approach than using the work area letter or number, since it lets your code express intent rather than structure. For example, if you know that the Customers table is open in work area 3, you can move to that work area with any of the lines of code in Listing 45.

Listing 45. There are three ways to switch to a specific work area: using the work area number, its letter, or the alias of the table open.

```
* Version 1
SELECT 3

* Version 2
SELECT C

* Version 3
SELECT Customers
```

The first two versions tell you very little; to understand them, you have to know what table is open in work area 3/C. The third version, though, tells you that you're moving to the work area for the Customer table. (In fact, there's rarely a reason to select a particular work area anymore; see “Use IN rather than SELECT” later in this paper.)

Open tables without thinking about work areas

While using the alias rather than the work area number in your code is handy, it wouldn't be all that powerful if you still had to remember which work areas are available every time you want to open a table. Fortunately, you can avoid that, as well.

VFP (and FoxPro before it) offers two ways to find an available work area. SELECT 0 moves to the lowest available work area, while the function call SELECT(1) returns the number of the highest available work area. So, you can be sure you're opening a table in an unused work area by issuing either of the sequences in Listing 46.

Listing 46. Open a table in an available work area by using either SELECT 0 or SELECT(1).

```
* Version 1
SELECT 0
USE YourTable

* Version 2
SELECT SELECT(1)
USE YourTable
```

In fact, you can consolidate either of those into one line of code, with one difference in behavior. The USE command, like many of VFP's Xbase commands, accepts the IN clause to specify the work area. When you specify IN 0, VFP opens the table in the lowest available work area;

specify IN SELECT(1), and VFP opens the table in the highest available work area. Listing 47 shows the two forms.

Listing 47. If you don’t need to land in the work area for the newly opened table, you can add the IN clause to USE.

```
* Version 3
USE YourTable IN 0

* Version 4
USE YourTable IN SELECT(1)
```

The difference between the one-line versions 3 and 4 and the two-line sequences of versions 1 and 2 is that after the one-liners, you haven't changed work areas. You're still in the same work area as before opening the table. Often, that doesn't matter, especially if you're opening a series of tables. Consider the code in Listing 48 and Listing 49. The two versions have the same effect, but not only is version 2 shorter and easier to read, but it expresses intent more clearly than version 1.

Listing 48. Selecting specific work areas makes this code longer and harder to read.

```
* Version 1
SELECT 1
USE Customers
SELECT 2
USE Orders
SELECT 3
USE OrderDetails
SELECT 1
* Do something with customer records
```

Listing 49. With USE IN, the code to open several tables is compact and clear.

```
* Version 2
USE Customers IN 0
USE Orders IN 0
USE OrderDetails IN 0

SELECT Customers
* Do something with customer records
```

Use IN rather than SELECT

Many of VFP's commands and functions operate in the current work area. If you forget to SELECT the right work area, or you do so, but then something changes the work area before your code runs, you can get nasty, unexpected results. Fortunately, many of those commands now accept an IN clause to indicate which work area to operate on. Similarly, most of the functions accept a parameter to indicate the work area.

When you use the IN clause or pass the parameter, you ensure that the command or function operates where you want it to; no event, ON KEY LABEL or user action (such as clicking on a grid) can switch work areas on you. In addition, you don’t need to save the current work area and switch to the desired work area, then restore the old work area after doing whatever you need.

Consider the code in Listing 50; four lines of code to do a simple calculation. (Full disclosure: I don’t think I’ve ever used the CALCULATE command in an application; it’s just convenient for this demonstration. One good place to use it though is instead of the COUNT, SUM and

AVERAGE commands, as those don’t support the IN clause.) You can do the same calculation in one line, using the IN clause, as in Listing 51.

Listing 50. Without the IN clause, you have to switch to the right work area and then switch back.

```
nOldSelect = SELECT()
SELECT Orders

CALCULATE SUM(Freight) TO nTotalFreight

SELECT (nOldSelect)
```

Listing 51. Using IN, the calculation is reduced to one line.

```
CALCULATE SUM(Freight) TO nTotalFreight IN Orders
```

Similarly, look at the difference when searching for a particular record with the SEEK() function. In Listing 52, you have to save and restore the work area, as well as the index tag. Using the optional alias and order parameters of the function, you can eliminate six lines of code and decrease the risk of error, as in Listing 53.

Listing 52. To find a particular record, you can make sure you’re in the right work area and using the right tag, then restore the tag and work area.

```
nOldSelect = SELECT()
SELECT Orders

cOldOrder = ORDER()
SET ORDER TO OrderID

IF SEEK("      10313")
   * Process this record
ENDIF

SET ORDER TO (cOldOrder)
SELECT (nOldSelect)
```

* Version 2

Listing 53. The optional parameters for SEEK() make the code shorter and less prone to error.

```
IF SEEK("      10313", "Orders", "OrderDate")
   * Process records for this date
ENDIF
```

Behind the scenes, of course, VFP does change work areas, but it handles all the details of making the change and restoring the original set-up. Keep this in mind, though, as you write code; commands that use IN and functions that pass the optional alias parameter must assume that they're executing in the specified work area.

The special case of REPLACE

While using the IN clause is convenient for most Xbase commands, for REPLACE, it's virtually required. First, REPLACE is destructive; changing data in the wrong area can have serious consequences. Second, REPLACE has a behavior that surprises many VFP developers; if you're at EOF in the current work area, nothing gets replaced.

When looking at other people's code, I often see lines like Listing 54. If the current work area is Orders, the code behaves as you'd expect, substituting the value of nFreightTotal into the Freight field of the current work area. In this situation, the alias Orders is simply unnecessary.

Listing 54. A REPLACE like this, that expects data to be replaced in the work area of the field listed, may not behave as you expect.

```
* DON’T DO THIS!
REPLACE Orders.Freight WITH m.nFreightTotal
```

But what happens if the current work area is something other than Orders? That depends on what's open in the current work area. If there's no table open in the current area, the user is prompted to open a table; if he cancels that dialog, error 52, "No table is open in the current work area" fires. If the user picks a table to open, and that table has any records, the replacement succeeds. If the user picks a table to open, and the table is empty, the replacement fails. Clearly, none of these results is desirable, since we certainly don't want the user prompted to open a table.

If a table is open in the current work area, the results depend on the position of the record pointer. If it's pointing to a valid record, the replacement succeeds. If the record pointer is at the end-of-file marker, the replacement silently fails to take place. It's easy to see that, even though the REPLACE command shown above might succeed, it's pretty risky as written.

Why does VFP behave this way? Because REPLACE, like most other Xbase commands, is scoped to the current work area. If you don't specify otherwise, REPLACE is understood as being REPLACE NEXT 1 in the current work area.

For comparison, consider the DELETE command, which also has a default scope of NEXT 1. You'd never issue DELETE with one work area selected, and expect it to delete a record in another work area (unless you added an IN clause). And, if the current work area is at EOF, you wouldn't expect DELETE to delete any records.

What makes REPLACE so confusing is its ability to replace fields in multiple tables at one time. For example, Listing 55 shows a valid (if dangerous) command. It affects the current record in both Person and Address. But what if one of those tables is at EOF? VFP's rules determine that it's the current work area that matters most. If you're at EOF there, the REPLACE doesn't take place at all; otherwise, it makes the attempt. However, if any other table is at EOF, nothing happens in that table.

Listing 55. Replacing data across several tables at once is another risky thing to do. You can’t be sure what the result will be.

```
REPLACE Person.cLast WITH m.cLast, ;
        Person.cFirst WITH m.cFirst, ;
        Address.cStreetAddr WITH m.cStreetAddr, ;
        Address.cCity WITH m.cCity, ;
        Address.cState WITH m.cState, ;
        Person.cEmail WITH m.cEmail
```

These risks mean that REPLACE should never be used to change records in more than one table at once and that it should always include the IN clause. Listing 56 shows a variation of the command in Listing 54. There's no ambiguity here. The Freight field in the Orders table will be updated as long as Orders is not at EOF. No errors, no dialogs appearing to the user (unless

Orders is not open, in which case this is a developer error), no failure based on the state of any other table.

Listing 56. Always add the IN clause to the REPLACE command.

```
REPLACE Freight WITH m.nFreightTotal IN Orders
```

For the more complex REPLACE in Listing 55, two commands would be better, as in Listing 57.

Listing 57. Use two REPLACE commands rather than changing data in two tables with a single command.

```
REPLACE cLast WITH m.cLast, ;
        cFirst WITH m.cFirst, ;
        cEmail WITH m.cEmail ;
   IN Person
REPLACE cStreetAddr WITH m.cStreetAddr, ;
        cCity WITH m.cCity, ;
        cState WITH m.cState ;
   IN Address
```

Where you can use IN

Not every Xbase command supports the IN clause. However, the ones that do include the ones you're most likely to use in applications. IN is supported for all the commands listed in Table 1.

Table 1. These commands support the IN clause. Use IN every time you use one of these commands in you code.

APPEND
BLANK
CALCULATE
DELETE
DISPLAY/LIST STRUCTURE
FLUSH
GO
PACK
RECALL
REPLACE
SEEK
SET FILTER
SET ORDER
SET RELATION
SKIP
UNLOCK

USE
ZAP

In addition, virtually all of the functions that operate on the current work area by default include an optional alias parameter. Among them are the ones listed in Table 2. For a complete list of functions that accept an alias parameter, search the VFP help using the string "nWorkArea | cTableAlias".

Table 2. These functions accept an optional alias parameter. Use it every time to make your code more reliable.

AFIELDS()
ALIAS()
ATagInfo()
BOF()
EOF()
FILTER()
FLOCK()
FOR()
FOUND()
LOCK()
ORDER()
RECCOUNT()
RECNO()
RLOCK()
SEEK()
TAG()
USED()

Unfortunately, some Xbase commands don't let you specify the IN clause. The two where this is the biggest issue for me are LOCATE and SCAN. When you use either of these, or other commands or functions that depend on the current work area, take precautions to ensure that you're where you think you are and that you leave things as you found them.

For all commands and functions that let you specify the alias, do so every single time. Your code will be more reliable and its purpose will be clearer.

Handling temporary data

There are often times in an application where you need to create a table to hold some data temporarily. There are a variety of reasons you need to do this from collecting data for reporting, to holding a list of user selections.

In early versions of Fox, there were only a couple of choices for handling such data. You could put it into an array or you could create a temporary table on disk and make sure to clean up afterward. Since FoxPro 2.0, though, the preferred approach is almost always to create a cursor.

A cursor is a temporary table (the name stands for "CURrent Set Of Records'). It can do almost anything a real VFP table can do, but it has two big advantages. First, when you close it, it disappears, leaving no traces behind on the disk. Second, its name is an alias, not a filename, so it needs to be unique only within the data session. This means you don't need to find a unique name to store it with.

When compared to creating and deleting an actual table from disk, a cursor has a third advantage: speed. As long as the cursor is small enough to fit, it will be kept entirely in memory, avoiding the drag of disk operations. I used the program in Listing 58 (included in the session materials as TableVsCursor.PRG) to test and found creating a populating a cursor more than 200 times faster than creating ad populating a table. (I also tested with 50,000 records and got the same results.)

Listing 58. As long as it fits in memory, creating a populating a cursor is two orders of magnitude faster than creating and populating a table.

```
* Compare table vs. cursor for speed of creation
#DEFINE PASSES 10

LOCAL nRecords, nStart, nEnd, nPass
LOCAL cFileName, cOldSafety
LOCAL cFldVal, nFldVal

cFileStem = "TempTable"
cFileName = FORCEEXT(FORCEPATH(m.cFileStem,SYS(2023)), "DBF")

nRecords = 5000
cFldVal = "abc"
nFldVal = 123

* Test 1: Creating from scratch
? "Create and add ", m.nRecords, " records"

* Make sure it doesn't already exist
ERASE (m.cFileName)

* First, a table
nStart = SECONDS()
FOR nPass = 1 TO PASSES

   CREATE TABLE (m.cFileName) (cFld C(3), nFld N(3))
   FOR nRecord = 1 TO m.nRecords
      INSERT INTO (cFileName) ;
         VALUES (m.cFldVal, m.nFldVal)
   ENDFOR
   USE
   ERASE (m.cFileName)
ENDFOR
```

```
nEnd = SECONDS()

? "  For table, total time: ", nEnd - nStart

*  Now, a cursor
nStart = SECONDS()
FOR nPass = 1 TO PASSES

   CREATE CURSOR (m.cFileStem) (cFld C(3), nFld N(3))
   FOR nRecord = 1 TO m.nRecords
      INSERT INTO (cFileStem) ;
         VALUES (m.cFldVal, m.nFldVal)
   ENDFOR
   USE

ENDFOR
nEnd = SECONDS()

? "  For cursor, total time: ", nEnd – nStart
```

What about arrays? Is a cursor a better choice than an array for storing a small to moderate amount of data? From a speed perspective, it depends what you want to do with the data. I tested with data sets of four columns and ranging from 10 rows to 10,010. The first test was creating and populating the array or cursor. In this test, arrays seems to be about a third faster. My second test looped through the array and the cursor. Looping through the cursor was faster, but not enough to matter. (I also did this test with a single column of data; in that case, looping the array was faster.) Finally, I tested searching for data in each case. Here, the cursor was a clear winner. For larger data sets, SEEKing in a cursor was an order of magnitude faster than searching in an array with ASCAN(). The cursor's advantage was even bigger when the search item wasn't in the data set. For 10,010 items, searching for a non-existent item in an array took 50 times as long as SEEKing the same non-existent item in a cursor.

Listing 59 shows my test code (included in the session materials as ArrayVsCursor.PRG). It stores the timing results to a text file because there's so much there.

Listing 59. Looping through an array isn't particularly faster or slower than looping through a cursor.

```
#DEFINE PASSES 100

LOCAL nDataSize, nSetSize
LOCAL aDataArray[1], nPass, nItem, nStart, nEnd

* Store output to file
SET ALTERNATE TO ArrayVsCursor.TXT
SET ALTERNATE ON

* Test for different data set sizes
FOR nSetSize = 10 TO 10010 STEP 1000

   ? "Testing set size of ", m.nSetSize

   * First, testing populating
   nStart = SECONDS()
   FOR nPass = 1 TO PASSES
      DIMENSION aDataArray[m.nSetSize, 4]
      FOR nItem = 1 TO m.nSetSize
         aDataArray[m.nItem, 1] = m.nItem
         aDataArray[m.nItem, 2] = "abc"
         aDataArray[m.nItem, 3] = DATE()
         aDataArray[m.nItem, 4] = m.nSetSize - m.nItem
      ENDFOR
   ENDFOR
   nEnd = SECONDS()

   ? " Array populated in ", m.nEnd - m.nStart

   nStart = SECONDS()
   FOR nPass = 1 TO PASSES
      CREATE CURSOR DataCursor (nFld N(5), cFld C(3), dFld D, nFld2 N(5))
      FOR nItem = 1 TO m.nSetSize
         INSERT INTO DataCursor VALUES (m.nItem, "abc", DATE(), ;
                                        m.nSetSize-m.nItem)
      ENDFOR
   ENDFOR
   nEnd = SECONDS()

   ? " Cursor populated in ", m.nEnd - m.nStart

   * Now test loop speed
   nStart = SECONDS()
   FOR nPass = 1 TO PASSES
      FOR nItem = 1 TO ALEN(aDataArray, 1)
         nValue = aDataArray[m.nItem, 1]
         cValue = aDataArray[m.nItem, 2]
         dValue = aDataArray[m.nItem, 3]
         nValue2 = aDataArray[m.nItem, 4]
      ENDFOR
   ENDFOR
   nEnd = SECONDS()

   ? " For array, looping, ", PASSES, " passes took ", m.nEnd - m.nStart

   nStart = SECONDS()
   FOR nPass = 1 TO PASSES
      SELECT DataCursor
      SCAN
         nValue = DataCursor.nFld
         cValue = DataCursor.cFld
         dValue = DataCursor.dFld
         nValue2 = DataCursor.nFld2
      ENDSCAN
   ENDFOR
   nEnd = SECONDS()

   ? " For cursor, looping, ", PASSES, " passes took ", m.nEnd - m.nStart

   * Now test search
   * First, for an item that's there.
   * Make a list of items to search, one for each pass
   RAND(-1)
   DIMENSION aSearchItems[PASSES]
   FOR nItem = 1 TO PASSES
      aSearchItems[m.nItem] = INT(RAND() * m.nSetSize) + 1
      ? " Search item ", m.nItem, " = ", aSearchItems[m.nItem]
   ENDFOR

   nStart = SECONDS()
   FOR nPass = 1 TO PASSES
```

```
      nSearchItem = aSearchItems[m.nPass]
      IF ASCAN(aDataArray, m.nSearchItem, 1) > 0
         lFound = .T.
      ENDIF
   ENDFOR
   nEnd = SECONDS()

   ? " For array, searching for an existing item, ", PASSES, ;
     " passes took ", m.nEnd - m.nStart

   * Need to index cursor before search
   SELECT DataCursor
   INDEX on nFld TAG nFld

   nStart = SECONDS()
   FOR nPass = 1 TO PASSES
      nSearchItem = aSearchItems[m.nPass]
      lFound = SEEK(m.nSearchItem, "DataCursor", "nFld")
   ENDFOR
   nEnd = SECONDS()

   ? " For cursor, searching for an existing item, ", PASSES, ;
     " passes took ", m.nEnd - m.nStart

   nStart = SECONDS()
   FOR nPass = 1 TO PASSES
      * Move record pointer to top to make test fair
      GO TOP
      nSearchItem = aSearchItems[m.nPass]
      lFound = SEEK(m.nSearchItem, "DataCursor", "nFld")
   ENDFOR
   nEnd = SECONDS()

   ? " For cursor, searching for an existing item, starting from top ", ;
     PASSES, " passes took ", m.nEnd - m.nStart

   * Now search for a non-existent item
   nSearchItem = 2 * m.nSetSize
   nStart = SECONDS()
   FOR nPass = 1 TO PASSES
      IF ASCAN(aDataArray, m.nSearchItem, 1) > 0
         lFound = .T.
      ENDIF
   ENDFOR
   nEnd = SECONDS()

   ? " For array, searching for a non-existent item, ", PASSES, ;
     " passes took ", m.nEnd - m.nStart

   nStart = SECONDS()
   FOR nPass = 1 TO PASSES
      lFound = SEEK(m.nSearchItem, "DataCursor", "nFld")
   ENDFOR
   nEnd = SECONDS()

   ? " For cursor, searching for a non-existend item, ", PASSES, ;
     " passes took ", m.nEnd - m.nStart

ENDFOR

SET ALTERNATE off
SET ALTERNATE TO

RETURN
```

Given the timing results, on the whole, for a data set of any size, it's probably best to use a cursor. For small data sets (tens of items), it doesn't really matter, so use whichever makes the code easier to write, read and maintain.

Establishing primary keys

FoxPro is a relational database, that is, a database where data is stored in multiple tables with fields that establish relationships between those tables. In order to establish those relationships, there needs to be a way to uniquely identify each record in a table. The field or fields that link one table to another are called keys.

When I started working with FoxBase+, it wasn't unusual to use keys that spanned several fields. For example, a customer might be linked to its orders using the company name and phone number, as in Listing 60.

Listing 60. In the early days of Xbase, it wasn't unusual to link two tables together using multiple fields.

```
USE Customers IN 0
USE Orders IN 0 ORDER Customer
SELECT Customers
SET RELATION TO UPPER(Company + PhoneNum) INTO Orders
```

However, working with this type of relationship is cumbersome, and even before VFP was introduced, using a single field to link two tables emerged as a best practice. When VFP 3 introduced the ability to designate a field as a primary key, with its uniqueness enforced by the database engine, the move to single field keys accelerated.

In the example above, rather than using company name plus phone number to identify a customer, you add a customer id field of some sort and propagate that to the child tables. Using this strategy, the code to open and relate the tables looks more like Listing 61.

Listing 61. Using a single field rather than multiple fields to relate two tables is a best practice.

```
USE Customers IN 0
USE Orders IN 0 ORDER CustomerID
SELECT Customers
SET RELATION TO CustomerID INTO Orders
```

Few would argue with the idea of using a single identifying field as a *primary key*, the principal identifier of a record, in a table like Customers or Products. We're accustomed to the idea of assigning an ID to real-world objects like customers and products. (Consider, for each, the UPC codes that adorn millions of products or the social security number that ostensibly uniquely identifies each American taxpayer.)

However, even tables that don't represent real-world objects with obvious identifiers should have primary keys as well. So, in the example above, not only should the Customers and Orders table have an ID field, so should the LineItems table that lists the individual items on each order.

You might wonder why you can't just identify each line item by its order number and line number, or by the order number plus the product number. The second example is easier to dispense with. If a detail line is identified by order number plus product number, each product can be included on only one line of an order. While that might be the norm, enforcing such a rule might impose a hardship on an application's users.

But why not identify a detail line by the order number and line number? This is a good lead-in to the question of surrogate keys.

Using surrogate keys

Although the topic is somewhat controversial, most VFP experts also recommend using what are known as *surrogate keys*. That is, give the table a field whose only purpose is to uniquely identify the record. The field has no other meaning, and normally, is never seen by users. VFP 8 made using surrogate keys easy, with its addition of an auto-incrementing integer data type.

There are many pros to working with surrogate keys as well as a few cons. The biggest strength of surrogate keys is that they never need to change. When you use a meaningful data item as a primary key (sometimes called an *intelligent key*), you run the risk that the data might change later. For example, consider a Customer table using the name of the customer (company) as a primary key. If the customer company changes its name, not only must the Customer table be updated, but all the records for that customer in other tables (such as Order) must be modified as well. With a surrogate primary key, only the actual customer record needs to change.

Another big win for surrogate keys come in ensuring uniqueness. Since surrogate keys are assigned internally and never seen by the user, it's easy to make sure to assign unique values. With intelligent keys, there's a chance that the data designated as the primary key won't be unique. Obviously, given names and company names aren't unique. Even supposedly unique identifiers like social security numbers turn out to duplicated occasionally, whether through error or fraud. Using a made-up "company code" type field is better, but then you're relying on the user to create unique identifiers. Even if you establish a rule for creating the identifiers (say, the first 10 characters of the company name plus the zip code), sooner or later, you're likely to run into a repeated value and have to find another way to generate the code.

Related to ensuring uniqueness is handling gaps. With surrogate keys, gaps in the sequence don't matter. No one ever sees them. If you use a meaningful field like order number, some users will want to make sure every value in the sequence is used. In multi-user applications, that turns out to be surprisingly difficult.

Because surrogate keys are always just one field, not multiple fields, writing joins that involve them is simple.

Finally on the positive side, surrogate keys are often smaller than intelligent keys. While storage isn't a big issue any more, anything that saves space without giving up something in return is still a plus.

There are two real downsides to using surrogate keys. The first is that they can actually make queries more complex. If a child table contains only a foreign key to the parent table, rather than actual data from the parent, every query that needs data from both must perform a join.

In addition, using surrogate keys makes it difficult to re-link records when data has been damaged in some way. Along these lines, it's also more difficult to simply look at a table and understand what it contains.

Overall, though, the positives of surrogate keys far outweigh the negatives, and using surrogate primary keys is a best practice for VFP. There are several ways to generate primary keys, but for VFP 8 and later, the auto-incrementing integer field type is the easiest way.

Manipulating dates

Calendar-related data is a big part of many applications. We store birth dates, dates hired, order dates, shipping dates, and many, many other dates. As a result, it's not unusual to need to manipulate dates in one way or another. Perhaps the most common tasks are constructing a date from character data and moving forward or backward by a week, month or year.

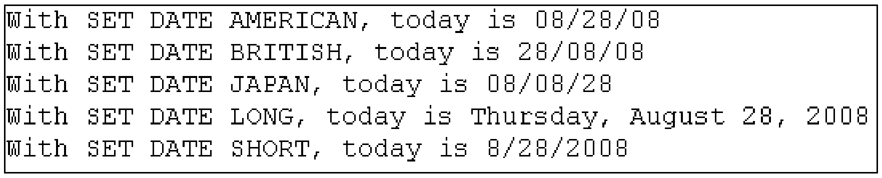
From date to character

There are various reasons why you need to change a date value into a character string. The two most common are for display and for use in a compound index (where the date is combined with other data such as a customer code).

DTOC() (Date TO Character), which has been in the language as long as I've been using it, takes a date and converts it to a character string. The format of the string is determined by the current SET DATE setting, which means DTOC() is fine for displaying dates in reports, but not for situations where you need to get the same results every time, given the same data. The code in Listing 62 shows how much the result can differ; the results are in Figure 3.

Listing 62. DTOC() converts dates to characters strings, based on the current setting of SET DATE.

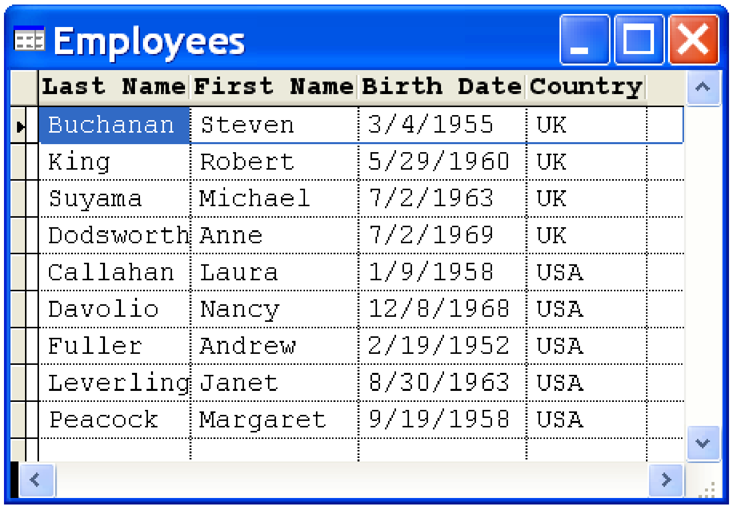
```
LOCAL dToday

dToday = DATE()

SET DATE AMERICAN
? "With SET DATE AMERICAN, today is", DTOC(m.dToday)

SET DATE BRITISH
? "With SET DATE BRITISH, today is", DTOC(m.dToday)

SET DATE JAPAN
? "With SET DATE JAPAN, today is", DTOC(m.dToday)

SET DATE LONG
? "With SET DATE LONG, today is", DTOC(m.dToday)

SET DATE SHORT
? "With SET DATE SHORT, today is", DTOC(m.dToday)
```



```
With SET DATE AMERICAN, today is 08/28/08
With SET DATE BRITISH, today is 28/08/08
With SET DATE JAPAN, today is 08/08/28
With SET DATE LONG, today is Thursday, August 28, 2008
With SET DATE SHORT, today is 8/28/2008
```

Figure 3. The results of DTOC() vary dramatically.

DTOC()'s dependence on SET DATE is fine when the goal is to display the result. In fact, that's the desired behavior. However, when creating indexes, that dependence is a problem. Consider the Northwind Employee table. Suppose you want to create an index based on the country and birth date of the employee. That is, you want to sort employees by country and birth date. Your first attempt for an index expression might be Country + DTOC(Birthdate). Figure 5 shows the table using such a tag. As you can see, the employees are in country order, but not properly sorted by birth date within the country.



Last Name	First Name	Birth Date	Country
Buchanan	Steven	3/4/1955	UK
King	Robert	5/29/1960	UK
Suyama	Michael	7/2/1963	UK
Dodsworth	Anne	7/2/1969	UK
Callahan	Laura	1/9/1958	USA
Davolio	Nancy	12/8/1968	USA
Fuller	Andrew	2/19/1952	USA
Leverling	Janet	8/30/1963	USA
Peacock	Margaret	9/19/1958	USA

Figure 4. Using DTOC() in an index doesn't usually give the results you expect.

The problem is that, with DATE set to AMERICAN (my default setting), the month comes first. So the employees are sorted by country, then by month, then by day within the month and finally by year.

There are two, functionally identical, alternatives to DTOC() for indexing; both have been in the language for a long time. The first option is to pass 1 as a second parameter to DTOC(). This tells the function to always put the date in YYYYMMDD format, regardless of the SET DATE setting. The DTOS() function (Date TO System or Date TO Standard) does the same thing. Either one can be safely used in an index expression. In the example, then, the desired index expression is Country + DTOC(Birthdate, 1) or Country + DTOS(Birthdate).

The session materials include DateToString.SCX, a form that lets you experiment with converting date to strings.

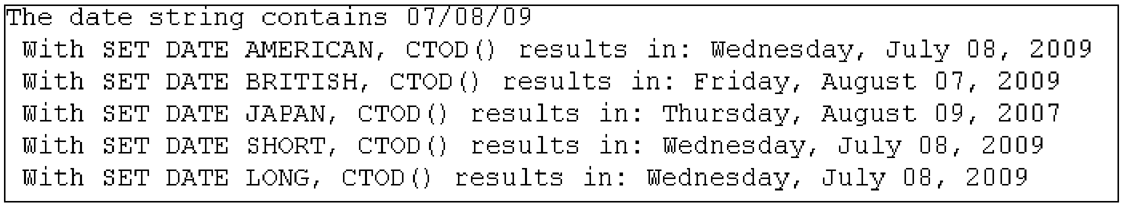
Creating dates

The inverse function to DTOC() is CTOD() (Character TO Date); it takes a character string that looks like a date and turns it into an actual date. Like DTOC(), it's dependent on the current SET DATE value to figure out what date a string represents. Listing 63 shows date conversion with a number of different settings. The results are shown in Figure 5. (In order to display the result, TRANSFORM() is used to convert back to a string, always in the current long date format.)

Listing 63. When converting from character to date, CTOD() relies on the SET DATE setting.

```
LOCAL cDate
```

```
cDate = "07/08/09"

? "The date string contains", m.cDate

SET DATE AMERICAN
? " With SET DATE AMERICAN, CTOD() results in:", TRANSFORM(CTOD(m.cDate),
"@YL")

SET DATE BRITISH
? " With SET DATE BRITISH, CTOD() results in:", TRANSFORM(CTOD(m.cDate),
"@YL")

SET DATE JAPAN
? " With SET DATE JAPAN, CTOD() results in:", TRANSFORM(CTOD(m.cDate), "@YL")

SET DATE SHORT
? " With SET DATE SHORT, CTOD() results in:", TRANSFORM(CTOD(m.cDate), "@YL")

SET DATE LONG
? " With SET DATE LONG, CTOD() results in:", TRANSFORM(CTOD(m.cDate), "@YL")
```



```
The date string contains 07/08/09
 With SET DATE AMERICAN, CTOD() results in: Wednesday, July 08, 2009
 With SET DATE BRITISH, CTOD() results in: Friday, August 07, 2009
 With SET DATE JAPAN, CTOD() results in: Thursday, August 09, 2007
 With SET DATE SHORT, CTOD() results in: Wednesday, July 08, 2009
 With SET DATE LONG, CTOD() results in: Wednesday, July 08, 2009
```

Figure 5. Converting from a character string to a date depends on the SET DATE setting.

The date in the example was chosen carefully to work with any date setting. With some dates and some SET DATE settings, CTOD() simply fails to do anything. For example, with SET DATE BRITISH, CTOD("9/28/58”) returns the empty string. SET DATE AMERICAN and the function call works.

There’s no analogue of DTOS() for converting from character to date because you simply need to know how to interpret a date string to turn it into a date value.

However, one traditional use of CTOD() is creating dates on the fly. It’s not unusual to see code like that in Listing 64 in older applications. This code, of course, depends on the SET DATE setting. There are two alternatives that don’t depend on that setting.

Listing 64. In FoxBase and FoxPro, it was common to use CTOD() to build date constants.

```
dStart = CTOD("09/01/2008")
```

The first one introduced was the curly brace ({}) notation for date constants. Rather than having to convert a string to a date, you can simply write the date between curly braces. Beginning in VFP 5, you can also use the strict date notation to make date constants independent of the SET DATE setting. Strict date notation begins with a carat (^) and then includes the date in YMD format. Listing 65 shows strict date format; Listing 66 shows the example of Listing 64 using a date constant in strict date format.

Listing 65. Curly braces enclose date constants. Use strict date format to avoid dependence on SET DATE.

```
{^ YYYY-MM-DD}
```

Listing 66. Use strict date format for date constants rather than converting character values.

```
dStart = {^ 2008-09-01}
```

While the ability to write unambiguous date constants is very helpful, in VFP 6, we were given the ability to unambiguously specify any date. The DATE() function accepts three optional numeric parameters: the year, the month and the day. When you pass those parameters, the function returns the corresponding date. Listing 67 shows the same example again, this time using DATE().

Listing 67. You can pass year, month and day to DATE() to construct a date value.

```
dStart = DATE(2008, 9, 1)
```

With these two facilities, as well as the date-parsing functions described in the next section, there's rarely, if ever, a reason to use CTOD() any more.

Parsing dates

In early versions of FoxPro, if you needed to extract part of a date (like the month), you had to convert it to character and then parse the character string, as in Listing 68. With DTOS(), you can avoid dependence on SET DATE (though most older code I've encountered doesn't do so), but you still have to go through two steps.

Listing 68. The old way to extract part of a date was to convert to character and parse the string.

```
* Assume dStart contains a date value
cStart = DTOS(m.dStart)
nMonth = SUBSTR(m.cStart, 5, 2)
```

Fortunately, you never have to do it that way. The YEAR(), MONTH() and DAY() functions let you extract the different parts of a date without the intermediate step of a character string. Listing 69 shows extraction of the month.

Listing 69. The new way to extract part of a date uses the functions provided for that purpose.

```
* Assume dStart contains a date value
nMonth = MONTH(m.dStart)
```

This set of functions actually goes beyond year, month and day. There are also functions to provide the day of the week (DOW()) , a character version of the day of the week (CDOW()), the name of the month (CMONTH()), the week of the year (WEEK()), and the quarter of the year (QUARTER()). DOW() and WEEK() are dependent on the current settings for first day of the week (SET FDOW) and first week of the year (SET FWEEK). The two character results depend on the VFP language resource you're using.

The session materials include ParseDate.SCX, a form that lets you try the new and old ways and compares their speed.

Date math

FoxPro has always supported date math. That is, you can subtract one date from another to find the number of days in between, and you can add a number to (or subtract a number from) a date to move to a new date. For example, the code in Listing 70 stores the date a week from today.

Listing 70. Date math means you can do arithmetic directly with dates.

```
dNextWeek = DATE() + 7
```

Sometimes, though, you need to set up a date that can’t be computed by simple date math, such as the first of the same month as a date variable. Before the date parsing functions described in the last section were added, you did that by converting from date to character, parsing the character string, making some changes, reassembling the string and then converting it back to date. Listing 71 shows code to find the first of the month from a given date. Because you have to convert back from string to character, you need to control the SET DATE setting.

Listing 71. At one time, you needed code like this to find the first of the month.

```
* Assume dDate contains the date of interest
cDate = DTOS(m.dDate)
cFirstOfMonth = LEFT(m.cDate, 4)+ "/" + SUBSTR(m.cDate, 5, 2) + "/01"
cOldDate = SET("DATE")
SET DATE YMD
dFirstOfMonth = CTOD(m.cFirstOfMonth)
SET DATE &cOldDate
```

The date parsing functions in the previous section make code like this much simpler. To find the first of the month, for example, you can just subtract the day of the month (which gives you the last day of the preceding month) and add 1, as in Listing 72.

Listing 72. The ability to parse dates directly adds to date math’s possibilities.

```
* Assume dDate contains the date of interest
dFirstOfMonth = m.dDate - DAY(m.dDate) + 1
```

One common date math task that was difficult at one time is moving to a new date a certain number of months or years before or after a given date. For example, you may want to schedule a follow-up one month from a particular date, or set a warranty to expire in six months from purchase date. The problem with date math here is that different months have different numbers of days, and even different years have different numbers of days. To find the date six months from today, you can’t just add 6*30 to the date. Even the date parsing functions don’t really solve this problem, since a day that exists in one month may not exist in another.

Prior to FoxPro 2.0, solving these problems required a lot of code. The easiest solution was to convert to character, do your work on strings, and then convert back. But even then, you had to make sure you didn’t land on a non-existent date. For year-wise calculations, you had to worry about leap years. Listing 73 shows a function that attempts to take all this into consideration. It accepts a date and a number of months; pass a positive number of months to go forward in time and a negative number to go backward.

Listing 73. Moving by months or years was difficult in early versions of VFP.

```
LPARAMETERS dDateIn, nMonths

LOCAL cDateIn, cDateOut, cMonth, cYear, cDay
LOCAL nNewMonth, nNewYear, nNewDay, nFebLast
LOCAL dOldDate, dDateOut

cDateIn = DTOS(m.dDateIN)
cYear = LEFT(m.cDateIn, 4)
cMonth = SUBSTR(m.cDateIn, 5, 2)
cDay = RIGHT(m.cDateIn, 2)
```

```
nNewMonth = VAL(m.cMonth) + m.nMonths
DO CASE
CASE m.nNewMonth > 12
   nNewYear = VAL(m.cYear) + INT(m.nNewMonth/12)
   nNewMonth = MOD(m.nNewMonth, 12)
CASE m.nNewMonth < 1
   nNewYear = VAL(m.cYear) + INT(m.nNewMonth/12) -1
   IF MOD(m.nNewMonth, 12) = 0
      nNewMonth = 12
   ELSE
      nNewMonth = MOD(m.nNewMonth, 12)
   ENDIF
OTHERWISE
   nNewYear = VAL(m.cYear)
ENDCASE

nNewDay = VAL(m.cDay)

DO CASE
CASE m.nNewMonth = 2
   * Is it a leap year?
   IF MOD(m.nNewYear, 4) = 0 AND ;
     (MOD(m.nNewYear, 100) <> 0 OR MOD(m.nNewYear, 400) = 0)
      nFebLast = 29
   ELSE
      nFebLast = 28
   ENDIF
   IF m.nNewDay > m.nFebLast
      m.nNewDay = m.nFebLast
   ENDIF
CASE INLIST(m.nNewMonth, 4, 6, 9, 11)
   IF m.nNewDay = 31
      m.nNewDay = 30
   ENDIF
OTHERWISE
   * Nothing to do
ENDCASE

cDateOut = STR(m.nNewYear) + "/" + STR(m.nNewMonth) + "/" + STR(m.nNewDay)

cOldDate = SET("Date")
SET DATE YMD
dDateOut = CTOD(m.cDateOut)
SET DATE &cOldDate

RETURN m.dDateOut
```

That’s a tremendous amount of code for a common situation. Of course, if you needed to do this kind of thing regularly, you’d create some helper functions, such as IsLeapYear (to determine whether a year is a leap year) and perhaps BuildDate (to take a year, month and day and convert it to a date—essentially, a homegrown version of DATE()).

Fortunately, you don’t need to do this. The GoMonth() function does the same thing as all this code. It’s also extremely fast. In my tests, the native GoMonth() function was an order of magnitude (that is, about 10 times) faster than the function in Listing 73.

GoMonth() accepts a date and a number of months, and returns the date that many months from the one specified. Listing 74 demonstrates.

Listing 74. GOMONTH() makes it easy to move by months or years.

```
* Move forward one month
dAddOneMonth = GOMONTH(m.dDate, 1)
* Move backward one month
dSubtractOneMonth = GOMONTH(m.dDate, -1)
* Move forward one year
dAddAYear = GOMONTH(m.dDate, 12)
* Move backward one year
dSubtractAYear = GOMONTH(m.dDate, -12)
```

The ability to construct and parse dates, along with moving forward and backward by whole months, makes working with dates a breeze. The session materials include DateMath.SCX, a form that demonstrates and speed tests the homegrown and built-in versions of GoMonth().

Programming techniques

The last area where we need to update our practices is in the basic code we write, including the constructs we use to structure our code.

Code that changes at run-time

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 four 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 use macros and don't use any of the other options. We'll look at each of them in order of their introduction into VFP.

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 75..

Listing 75. The macro operator lets you substitute the right keyword into the SET commands.

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

* Code that overwrites a file with equanimity

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 76 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 76. 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"]
   This.cWhereEnglish = "USA only"

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

CASE This.opgIncludeLocation.Value = 4 && One state
   cWhere = [cState = "] + This.cboStateProvince.Value + ["]
   This.cWhereEnglish = "One state: " + 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 without knowing their names. For example, code like Listing 77 was common.

Listing 77. In early versions of Fox, accessing a table or field without knowing its name required a macro.

```
USE &cTable
```

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 "Name expressions" later in this section 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, 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 78.

Listing 78. 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 79 (MacroVsEval.PRG in the session materials) to test; I found that the macro version took about a quarter of the time of the EVAL() version.

Listing 79. 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 80. 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 80. 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. However, 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 the session materials), 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.

Choosing the right loop command

When I started working with FoxBase+, it had only one way to write a loop. DO WHILE could handle whatever kind of loop you needed.

As time went by, though, looping choices in FoxPro improved. There are now no fewer than four different ways to construct a loop. So how do you know which one to use in a given situation? In fact, there are some fairly simple rules.

Looping through tables and cursors

Back in the early days of Xbase, we had to use DO WHILE to loop through a table (there was no such thing as a cursor then), as in Listing 81:

Listing 81. In early Xbase, DO WHILE was the only way to loop through a table.

```
GO TOP
DO WHILE NOT EOF()
   * Do whatever you need to this record
   SKIP
ENDDO
```

This structure worked, but you had to remember to put SKIP at the end of the loop and if you changed work areas within the loop, you had to make sure to change back before you reached SKIP or your code would fail.

The addition of the SCAN command made looping through tables much easier. The basic SCAN loop looks like Listing 82:

Listing 82. SCAN lets you loop through a table without having to issue SKIP or reset the work area.

```
SCAN
   * Do whatever you need to this record
ENDSCAN
```

SCAN has four advantages over DO WHILE NOT EOF(). First, unless you include the optional WHILE clause, it always starts at the top of the table.

Second, the SKIP is built in; you don't need to code it.

Third, at the end of each pass, it automatically returns to the controlling work area, whichever work area was selected when execution reached the SCAN command.

Fourth, in most cases, SCAN is faster than the equivalent DO WHILE loop. In my tests (DoWhileVsScan.SCX in the session materials), looping through an unordered table and doing nothing else, SCAN took about 70% of the time of DO WHILE.

Things change somewhat if you use an ordered table. In the same tests, if I SET ORDER TO an index tag, SCAN’s advantage changed to about 80% of DO WHILE’s time. In fact, the relative times depend on the tag you use. Over the years, I have seen a few cases where using a particular index order, DO WHILE was faster by as much as 20%, but usually SCAN has the advantage.

What if you only want to work with some of the records, that is, you need to filter the data? Both loop constructs can handle this, but there are same caveats.

DO WHILE continues as long as its condition is true. When you add a condition other than NOT EOF(), the loop continues only as long as records meet that condition. If you can't order the data so that all the records you want to process are together, then you need to use an IF statement inside the loop rather than adding the condition to the loop. For example, using the Northwind Customers table, suppose you want to work with all customers in the UK. The loop in Listing 83 won’t find them all.

Listing 83. When you add a condition other than NOT EOF() to a DO WHILE loop, you may not process all of the records that meet the condition.

```
GO TOP
DO WHILE NOT EOF() AND Country = "UK"
   * Do something with this record
   SKIP
ENDDO
```

Since there's no tag available for country, to make sure you find each UK customer with DO WHILE, you need code like Listing 84.

Listing 84. If you don't have an index on the field you want to filter by, with DO WHILE you need an IF inside the loop.

```
GO TOP
DO WHILE NOT EOF()
   IF Country = "UK"
      * Do something with this record
   ENDIF
   SKIP
ENDDO
```

SCAN offers a better alternative in this case; add the FOR clause, as in Listing 85.

Listing 85. SCAN FOR is an easier way to loop through all records matching a condition.

```
SCAN FOR Country = "UK"
   * Do something with this record
ENDSCAN
```

Sometimes, you can order the data so that all the records you want to process are together. In that case, stay away from SCAN FOR and use SCAN WHILE instead; in general, it will be much faster, since it only visits the matching records. The code in Listing 86 visits every record in the Nothwind OrderDetails table for a particular product.

Listing 86. When there's an index for the desired field, SCAN WHILE is generally much faster than SCAN FOR.

```
SELECT OrderDetails
SEEK m.nProductID
SCAN WHILE ProductID = m.nProductID
   * Do something with this record
ENDSCAN
```

In my tests, SEEK followed by SCAN WHILE is the fastest, but SEEK followed by DO WHILE is nearly as fast.

A final note on looping through tables. Often, there's actually no reason to do processing in a loop. Most of the Xbase commands in VFP accept FOR and WHILE clauses, so that all relevant records can be processed in a single command. Save loops for those times when you need to do more complex processing.

Counted Loops

More often than looping through a table, I need to write a loop that executes a fixed number of times. DO WHILE is up to the task, with a structure like the one in Listing 87.

Listing 87. You can use DO WHILE for a counted loop, but it's slow.

```
nCount = m.nStart
DO WHILE m.nCount <= m.nEnd
   * Do whatever you need to
   nCount = m.nCount + 1
ENDDO
```

Once again, though, VFP offers a better alternative, the FOR loop. Instead of the code above, use code like Listing 88.

Listing 88. A FOR loop is about 10 times faster than an equivalent DO WHILE loop.

```
FOR m.nCount = m.nStart TO m.nEnd
   * Do whatever you need to
ENDFOR
```

As with SCAN, FOR lets you omit the statement that keeps the loop moving; the loop variable is incremented automatically. Also, like SCAN, FOR is faster than DO WHILE. In this case, the difference is more than an order of magnitude. In my tests, using an otherwise empty loop, DO WHILE takes 10 to 13 times longer than FOR. (My tests are included in the session materials as DoWhileVsFor.SCX.)

There are a couple of things to be aware of with FOR. First, the end value is evaluated once when you enter the loop. That is, if you use a variable or expression to specify the last value, and do something in the loop that changes the value of the variable or expression, the number of passes doesn't change. For example, the loop in Listing 89 executes 200 times. You can't short-circuit it by changing the end variable; use EXIT instead.

Listing 89. The end condition of a FOR loop is evaluated only once. You can’t short-circuit it by changing the value of a variable or field.

```
nStart = 1
nEnd = 200
FOR m.nCount = m.nStart TO m.nEnd
   * Do whatever you need to
   nEnd = 100
ENDFOR
```

Second, FOR has an optional STEP clause that lets you count by something other than ones. You can specify any positive or negative number; it doesn't even have to be an integer. So you could write code like Listing 90.

Listing 90. The optional STEP clause of FOR lets you count by something other than ones.

```
FOR nValue = 0 TO 1 STEP .1
   * nValue will be 0, 0.1, 0.2, etc.
ENDFOR
```

VFP is smart enough that it tests whether you've passed the endpoint, rather than testing whether you've exactly matched it. So you can even write something like Listing 91.

Listing 91. A FOR loop stops when you pass the end value, even if you never hit it exactly.

```
FOR nValue = 0 TO 5 STEP .3
   * nValue will be 0, 0.3, 0.6, etc.
ENDFOR
```

This loop stops when nValue = 5.1; during the last pass through the loop, nValue = 4.8.

The ability to specify negative numbers means you can work backwards. Be sure, in that case, to make the start value larger than the end value. For example, you might write a loop like Listing 92.

Listing 92. You can even loop backwards with a FOR loop, using a negative STEP value. Be sure to set the start value to be higher than the end value.

```
FOR nValue = 500 TO 0 STEP -50
   * nValue will be 500, 450, 400, etc.
ENDFOR
```

Unless you need to test additional conditions, there's no reason ever to use DO WHILE for a counted loop. Even if you have additional conditions to test, you may be better off using IF and EXIT inside the loop.

What is DO WHILE good for?

If DO WHILE isn't the best choice for looping through tables and cursors or for counted loops, when is it appropriate? When you need to do something until a condition changes, that is, exactly for the cases its name implies.

Most of the DO WHILE loops I write look something like Listing 93.

Listing 93. DO WHILE is useful if you need to loop until a condition is met.

```
lFound = .F.
DO WHILE NOT m.lFound
   * Do something that sets lFound
ENDDO
```

For example, in a class that generates test data, I need to create a unique ID number that's random rather than ordered (to replicate the real world). I use the loop in Listing 94.

Listing 94. This loop generates random numbers between 100,000,00 and 999,999,999 until it finds one that hasn’t been generated previously.

```
lNewNum = .F.
DO WHILE NOT m.lNewNum
   nNumber = This.RandInt(10000000, 99999999)
   cNumber = TRANSFORM(m.nNumber)

   * Check whether it exists already
   IF NOT SEEK(m.cNumber, "__StudNums", "cNumber")
      lNewNum = .T.
      INSERT INTO __StudNums VALUES (m.cNumber)
   ENDIF
ENDDO
```

The RandInt method of this class returns a random integer between the parameters supplied. Then, I search the list of ID numbers already generated. If this one isn't there, I add it and set the lNewNum flag to .T. to end the loop.

Looping through collections

VFP has one additional looping construct that wasn't needed back in the old days. Both VFP itself and the many Automation servers it can talk to use collections to hold sets of similar items. For example, on a VFP form, there's a collection called Objects that contains an object reference to each control on the form. When automating Word, you can talk to its Documents collection or to an individual document's Paragraphs collection. VFP has built-in collections for forms.

Although you can traverse a collection using a FOR loop, VFP also supports the FOR EACH loop, designed specifically for walking through collections. To go through a collection with a FOR loop, you use code like Listing 95.

Listing 95. You can use a FOR loop to traverse a collection.

```
FOR nItem = 1 TO ThisForm.Objects.Count
   oObject = This.Objects[m.nItem]
   * Do what you need to with oObject
ENDFOR
```

The analogous FOR EACH loop looks like Listing 96.

Listing 96. FOR EACH was designed to loop through collections. On each pass, it gives you an object reference to the next object.

```
FOR EACH oObject IN ThisForm.Objects FOXOBJECT
   * Do what you need to with oObject
ENDFOR
```

There is one big difference. Although in my experience, the two loops process the items in the same order, you can't count on that. FOR EACH promises only to visit each item in the collection; it doesn't make any guarantees about the order in which they'll be processed.

In my tests, FOR EACH is about twice as fast as FOR. (My test code is included in the session materials as ForVsForEach.SCX.)

The FOXOBJECT keyword needs some explanation. Prior to VFP 8, there were only a few collections native to VFP, like the form's Controls collection and the grid's Columns collection. Most of the collections you needed to deal with, included some that appeared to be native (like the Projects and Files collections), were actually COM objects. As a result, FOR EACH was designed to work with COM objects. By default, the object it hands you each pass through the loop is a COM object.

In VFP 8, the Collection base class was added, giving us the ability to create our own native collections. Suddenly, having FOR EACH provide COM objects caused problems. Those objects didn't behave the way we expected. Not only that, but FOR EACH loops were slow.

So the FOXOBJECT keyword was added in VFP 9; when you add it to FOR EACH, the objects you're working with inside the loop are native VFP objects. Using FOXOBJECT, not only do the objects behave as expected, but FOR EACH without FOXOBJECT takes about 10 to 20 times as long as FOR EACH with FOXOBJECT. The bottom line is that when working with a native collection, you should always add FOXOBJECT to FOR EACH.

There is one situation where you must use FOR rather than FOR EACH. That's when you're removing items from the collection inside the loop. Assume oColl is a collection containing some items, where each item has a cKey property, indicating its key in the collection. Consider the code in Listing 97 to delete all the items from the collection, one by one:

Listing 97. This loop won't remove all the items from the collection.

```
FOR EACH oItem IN oColl FOXOBJECT
   oColl.Remove(m.oItem.cKey)
ENDFOR
```

In fact, only half the items get removed. Internally, VFP must use a pointer of some sort to keep track of its position in the collection. When you remove an item, you mess up the internal pointer.

A FOR loop, running backwards through the collection, solves the problem, as in Listing 98.

Listing 98. To remove items from a collection, you need to count backward.

```
FOR nItem = oColl.Count TO 1 STEP -1
   oItem = oColl[m.nItem]
   oColl.Remove(m.oItem.cKey)
ENDFOR
```

Of course, to remove all items from a collection, you can simply pass -1 to the Remove method, so this loop is unnecessary, as written. However, the same principle applies to a loop where you're doing some testing to determine whether to remove an item.

A final note. FOR EACH can be used with arrays as well as collections. However, I've never found a reason to do so. With an array, I generally like the guarantee of processing items in order.

To abbreviate or not to abbreviate

From its earliest days, FoxPro has supported four-character abbreviations for most keywords. So, veteran VFP developers know to type MODI COMM to open the program editor, or DISP STRU to see the list of fields in a table.

Using these abbreviations in the Command Window is no big deal. But, with a few exceptions (such as EVAL() rather than EVALUATE()), they have no place in code. On the flip side, the less you type, the fewer chances you have of typing the wrong thing.

Fortunately, the addition of IntelliSense in VFP 7 gives you the best of both worlds. You can type just enough to identify the command or function and IntelliSense fills in the rest. For almost every command and function, four characters (plus a space or left parenthesis) is enough to do the trick.

IntelliSense can actually save you much more typing. For example, type DOCASE followed by a space or Enter to insert the code block in Listing 99. Not only that, but the cursor is positioned after the CASE keyword on the second line, so you can start typing your first condition.

Listing 99. VFP's IntelliSense has lots of built-in smarts. Type DOCASE to insert this block.

```
DO CASE
CASE

OTHERWISE

ENDCASE
```

If IntelliSense doesn't have what you want, chances are you can add it. For example, I have BL defined to expand to BROWSE LAST, and DS set up for DISPLAY STRUCTURE. Adding this kind of shortcut is straightforward:

1. Open the IntelliSense Manager (Tools | IntelliSense Manager).
2. Click the Custom tab.

3. In the Replace textbox, type the abbreviation you want to use.
4. In the With textbox, type the expanded command.
5. Click the Add button.
6. Close the IntelliSense Manager.

Adding more complex items (like the DOCASE script) to IntelliSense is beyond the scope of this session. Check out this topic on the Visual FoxPro Wiki for a whole collection of scripts:

http://fox.wikis.com/wc.dll?Wiki~IntelliSenseCustomScripts~VFP

Use local variables

When I started using FoxBase+, there were two kinds of variables available: public and private. If you did nothing, any variables you used in code were private. Since that was the most restrictive scope available, it was easy to get into the habit of not declaring variables unless they were public.

VFP 3 changed the rules by introducing local scope for variables. Local is more restrictive than private, but has to be explicitly declared.

What do the three scopes mean and how do they work?

Public variables are available everywhere unless you define a private or local variable with the same name at a lower level. (In earlier versions of Fox, you couldn't redefine a public variable without releasing it first, but that's no longer true.) To define public variables, use the PUBLIC keyword. The variables are created and set to .F.

Private variables are available in the routine where they're declared or created and any routine called by that one (all the way down the call chain) unless something in the calling chain declares a local or private variable with the same name. To declare a private variable, use the PRIVATE keyword. However, unlike PUBLIC and LOCAL, doing so does *not* create the variable; it just reserves the name as private. You still have to actually assign a value to it, in order to create it. Listing 100 (included in the session materials as PrivateDoesNotCreate.PRG) demonstrates this. The variable cPrivate, declared private, but not created in the main program, is not the same cPrivate as in the subroutine. When you attempt to display the value of cPrivate after the call, an error is generated.

Listing 100. Declaring a variable PRIVATE doesn't create the variable.

```
* Declaring a variable private doesn't create the
* variable, just reserves the name as private.

PRIVATE cPrivate

DO SubProc

* The next line generates an error
ON ERROR WAIT WINDOW MESSAGE()
?m.cPrivate
ON ERROR

RETURN
```

```
PROCEDURE SubProc

* The variable here is private (because it's not declared)
* but it's not the same variable as in the main program

cPrivate="abc"
?m.cPrivate

RETURN
```

Local variables are available only in the routine where they're declared. To define a local variable, use the LOCAL keyword. The variable is created and set to .F.

Best practice is to use local variables for everything, unless you can identify a specific need for a broader scope. Most VFP experts recommend using only a single private variable in applications. That variable (often called goApp) holds an object reference to an application object, and is declared private in the application's main program. Defining the variable as private in the main program is sufficient to make it available throughout the application.

The problem with using public or private variables is that they make it too easy for one piece of code to accidentally break another. If one program (or class or form or report) depends on a particular variable that isn't passed as a parameter, and another program (or class or form) changes that variable, the first routine may no longer work.

By using only local variables and requiring all communication between routines to happen either through parameters or through properties of objects passed as parameters, code in one part of an application is protected from other parts of the application. (This is, in essence, what encapsulation is all about.)

The application object is typically the exception here, and is often used to communicate between different parts of the application. Even so, it's best if other parts of the application check for the application object's existence and work through its methods, rather than modifying application object properties directly. Limiting non-parameter communication to that provided by the application object also means that you know where to look when problems occurs.

Arrays and scope

Like other variables, arrays can be public, private or local. But unlike other variables, arrays have to be declared. To create a public or local array, you declare it like any other variable, just adding the dimensions, such as PUBLIC aPublic[2,4] or LOCAL aLocal[17]. In both cases, there's an optional ARRAY keyword.

To declare an array private, however, the PRIVATE keyword isn't sufficient. Instead, you use either the DIMENSION or the DECLARE keyword, as in Listing 101. Of course, you should choose one keyword or the other and use it every time.

Listing 101. To create a private array, use either DIMENSION or DECLARE.

```
DIMENSION aPrivate[10,3]
DECLARE aAlsoPrivate[100]
```

There's one confusing item. Once you create an array with whatever scope you choose, issuing DIMENSION or DECLARE simply reshapes it; these commands don't change the array's scope.

This actually makes sense when you consider what's really happening. If you use DIMENSION or DECLARE for an array that doesn't already exist, it's the same as using any other undeclared variable; it gets created as private. However, if you've already established the array's scope, it keeps that scope.

The same rule applies to the arrays created by VFP's "A" functions. There's a large set of functions (whose names all begin with the letter "A") that retrieves some information and puts it into an array. For example, APrinters() gets the list of available printers, while AFields() gets the list of fields in a table. For all these functions, if the array already exists, it keeps its original scope. If the function creates the array, it's declared private.

As with scalar variables, avoid public arrays entirely, and keep private arrays to a minimum. Declare array variables local.

The discussion here is just an overview of issues related to scope. For complete coverage of the topic, check out Barbara Peisch's session titled "Understanding Scope."

Passing parameters

There are two issues related to parameters where things have changed over the years: scope and figuring out how many parameters were passed.

In early Fox days, parameters received by a function or procedure were always private. That is, the command PARAMETERS x, y, z creates x, y and z as variables private to the routine. When local scope was added in VFP 3, we also got the LPARAMETERS command, which lets you indicate that the variables created to receive parameters should be scoped as local. Listing 102 shows both forms.

Listing 102. PARAMETERS defines private variables, while LPARAMETERS defines local variables. LPARAMETERS is a better choice.

```
PROCEDURE HasPrivateParams
PARAMETERS nFirst, cSecond

PROCEDURE HasLocalParams
LPARAMETERS   nFirst, cSecond
```

You can also define parameters without using either keyword by including them in the procedure, function or method header, as in Listing 103. Parameters declared this way are local.

Listing 103. Parameters listed in the header are local.

```
PROCEDURE ParamsInHeader(cSomething, nSomethingElse)
```

Making parameters local is always a better choice than making them private, for the same reasons that local variables are better than private variables.

Occasionally, a routine needs to know how many parameters were actually passed to it, so that it can behave appropriately. FoxPro has had the PARAMETERS() function since the early days to provide that information. In FoxPro 2.6, the PCOUNT() function was added "for dBase compatibility."

Unlike most of the functions so tagged, however, the dBase version is superior to the Fox version. The value returned by PARAMETERS() is reset every time another routine is called, including when ON KEY LABEL fires. So unless you grab the value as soon as you get into a routine and store it, the result can be wrong. In fact, because an ON KEY LABEL can fire at any time, even storing the return value of PARAMETERS() as the first executable line of a routine can occasionally fail. PCOUNT(), on the other hand, always returns the number of parameters passed to the currently executing routine.

Listing 104 demonstrates the difference between the two functions; this program is included in the session materials as CountingParameters.PRG. When you run this code, you get the output shown in Listing 105.

Listing 104. Use PCOUNT() rather than PARAMETERS() to determine how many parameters were passed.

```
* Demonstrate PARAMETERS() vs. PCOUNT()

Subproc("abc", 123)

RETURN

PROCEDURE Subproc(cParm1, nParm2)

? "Immediately on entry to Subproc"
? "  PARAMETERS() returns ", PARAMETERS()
? "  PCOUNT() returns ", PCOUNT()

Subsubproc()

? "After call to Subsubproc"
? "  PARAMETERS() returns ", PARAMETERS()
? "  PCOUNT() returns ", PCOUNT()

RETURN

PROCEDURE Subsubproc

RETURN
```

Listing 105. PCOUNT() returns the same value no matter where you call it in the routine. The result of PARAMETERS() depends on what else has happened.

```
Immediately on entry to Subproc
  PARAMETERS() returns    2
  PCOUNT() returns    2
After call to Subsubproc
  PARAMETERS() returns    0
  PCOUNT() returns    2
```

Keep on learning

As this document demonstrates, changing the way you handle a particular task can make your code faster, more reliable, and easier to maintain. “We’ve always done it that way” is not a good enough reason to keep doing something.

Implementing all of these changes at once is likely to be overwhelming. Instead, pick one or two to start with that are likely to have the most impact on your code. When you’ve mastered those, move on to a couple more.

In addition, while this session covers lots of ways to improve your VFP style, it’s not exhaustive. Spend some time with the help file or a good VFP book to see what else you’ve been missing.