The How and Why of Recursion

Tamar E. Granor
Tomorrow's Solutions, LLC
Voice: 215-635-1958
Website: www.tomorrowssolutionsllc.com
Email: tamar@tomorrowssolutionsllc.com

Recursion is a programming technique in which code calls itself either directly or indirectly. In some cases, using recursion can make code much simpler and more readable. In others, it adds only complexity.

In this session, we'll look at the basics of recursion and how to implement it in Visual FoxPro. (It's easy.) Then, we'll look at VFP problems where recursion is the best choice and some where it's not.

Introduction

Back in the days when I needed to do proofs, I had a couple of favorite techniques to try first. One of them was proof by induction. That's the strategy where you first prove the conjecture for a base case (usually, 0 or 1) and then you prove that if your conjecture is true for *n*, then it's also true for *n+1*. (See Appendix: Proof by induction example at the end of this paper for a specific case.)

Given my liking for proof by induction, it's not surprising that when I learned to program, I was attracted to recursion, because it's also a way of getting something done by identifying base cases and then building up from those.

Many programming languages, including VFP, allow a routine (procedure, function, or method) to call itself, or one routine to call a second, which calls the first, and so on. This is *recursion*. Many people are taught that recursion is dangerous or to be avoided. In fact, when used appropriately, recursion is a powerful technique that can make your code better.

The term "recursion" comes from the verb "recur," defined as "occur again periodically or repeatedly." If you look at the program stack while running a recursive routine, you'll likely see it there repeatedly. Although many people in the software world use the term "recurse" for what a recursive routine does, I prefer "recur" and use it in this paper.

Recursion is ideal for code that needs to drill down in some way and perform the same operation on a series of items. In such cases, recursion often results in simpler code.

Working with recursion

The reason many people consider recursion dangerous is that if you don't set it up right, the routine keeps calling itself over and over and over until the program stack overflows or memory runs out. But it's not hard to get recursion right.

There are two main rules for a recursive routine. First, provide an exit case, that is, some code that tests whether you're reached the end and stops the series of calls. Second, make sure the calls do not step on each other. That generally means using parameters to communicate from one call to the next and making sure variables are declared local.

One of the classic examples of recursion is computing factorials. The factorial of a non-negative integer *n*, written n!, is the product of all the numbers from 1 to *n*. By definition, 0! is 1.

We can write a routine to compute factorials with a loop, as in **Listing 1**, included in the materials for this session as FactorialLoop.PRG.

Listing 1. You can compute factorials iteratively.

```
* Compute factorials iteratively
LPARAMETERS tnInput
```

```
* Reject negative numbers
IF m.tnInput < 0
   RETURN -1
ENDIF

LOCAL nResult, nCounter

nResult = 1

FOR nCounter = 1 TO m.tnInput
   nResult = nResult * m.nCounter
ENDFOR

RETURN m.nResult
```

But, by its nature, the factorial function is recursive. The definition is often written as in **Listing 2**. So, it's not surprising that it's quite easy to write recursive code to do the calculation, as in **Listing 3**, included in the materials for this session as FactorialRecursive.PRG.

Listing 2. The standard definition of the factorial function is recursive.

```
n! = 0, for n=1
n! = n * (n-1)!, for n>1
```

Listing 3. Given its recursive definition, it's not surprising that it's easy to write a recursive function for computing factorials.

```
* Compute factorials recursively
LPARAMETERS tnInput

LOCAL nResult

DO CASE
CASE m.tnInput > 0
   * Recursive case, moves us closer to base case
   nResult = m.tnInput * FactorialRecursive(m.tnInput-1)

CASE m.tnInput = 0
   * Base case, ensures ending
   nResult = 1

CASE m.tnInput < 0
   * Check for valid input
   nResult = -1

ENDCASE

RETURN m.nResult
```

The first case is the recursive case, where we call the function again. The definition of nResult as local ensures that each call of the function gets its own copy of nResult. The

second case is the exit case (or base case) that ensures that we'll eventually exit. The final case checks for valid input. (My first version of this function had the cases in the opposite order, but I realized that the recursive case is most likely and the invalid case least likely, so reversed the order to speed the function up.)

Suppose we call this function, passing 3 as the parameter, that is FactorialRecursive(3). We can easily see that the correct answer is 3 * 2 * 1 = 6. Here's what happens:

- The first time we come into the function, tnInput = 3. So, we enter the first case and start calculating:

  nResult = 3 * FactorialRecursive(3-1)

- The second time we come into the function, tnInput = 2 (3-1). Again, we go to the first case and start calculating:

  nResult = 2 * FactorialRecursive(2-1)

- The third time we come into the function, tnInput = 1 (2-1). This time, we go to the second case in the case statement and set nResult = 1. Then we return that value.

- We come back to the second instance of the function and finish our calculation:

  nResult = 2 * 1 = 2

  We return 2.

- Now we come back to the first instance of the function and finish that calculation:

  nResult = 3 * 2 = 6

  We return 6 as the final result.

This example demonstrates something that's generally true. Anything that can be done by recursion can also be done without it. In some cases, like this one, both versions are pretty simple, so from a readability perspective, either one is fine.

But what about speed? In this case, my testing finds the loop version somewhat faster than the recursive version, about 2.5 times faster when the initial value is 5 and about 8 times faster when the initial value gets near 500.

My speed tests raised another issue though. There's a limit to the program stack in VFP, that is, to how many routines you can call without returning from those calls. By default, that limit is 128. You can adjust that limit by setting STACKSIZE in the Config.FPW file; the maximum is 64000.

Stack size seems like it would be a major issue for using recursion, but in fact, until I decided to speed-test the factorial functions on values up to 500, I'd never run into the issue except in buggy code. The kinds of things I use recursion for in VFP never have hundreds or thousands of depth levels.

If you are using recursion in a case where you might run out of stack space, you can test in your code to prevent a crash. Calling PROGRAM(-1) returns the current stack level.

Before looking at how recursion makes things easier in VFP, let's detour into a case where it's clear you shouldn't use recursion.

When NOT to use recursion

Sometimes, even when recursion feels natural for a problem, it's not a good choice. As the previous section indicates, one reason recursion can be a bad choice is that it would overflow the program stack.

Another reason is that it makes you do much more work. The definition of Fibonacci numbers is naturally recursive. The *n*th Fibonacci number is the sum of the (*n*-1)th and the (*n*-2)th Fibonacci numbers. **Listing 4** lays out the definition mathematically.

Listing 4. The definition of Fibonacci numbers is recursive.

```
Fibonacci(1) = 1
Fibonacci(2) = 1
Fibonnaci(n) = Fibonacci(n-1) + Fibonacci(n-2), for n>2
```

But using recursion to compute Fibonacci numbers is a bad idea. **Listing 5** shows a recursive VFP function for Fibonacci numbers; it's a very simple function and it's easy to read and understand. (It's included in the materials for this session as FibRecur.PRG.)

Listing 5. It's easy to write a recursive function for Fibonacci numbers, but don't.

```
* Generate the nth Fibonacci # by recursion.
LPARAMETERS tnWhich

LOCAL nResult

DO CASE
CASE m.tnWhich > 2
   nResult = FibRecur(m.tnWhich-1) + FibRecur(m.tnWhich-2)

CASE INLIST(m.tnWhich, 1, 2)
   nResult = 1

OTHERWISE
   * 0 or negative
   nResult = 1

ENDCASE

RETURN m.nResult
```

Listing 6 shows the iterative version. The code is a little more complex and you may have to stop and think for a moment to see how it works. (This routine is included in the materials for this session as FibLoop.PRG.)

Listing 6. The iterative version of the Fibonacci function is a little less readable, but far more efficient.

```
* Generate the nth Fibonacci # by iteration.
```

```
LPARAMETERS tnWhich

LOCAL nResult, nPrev1, nPrev2, nI

DO CASE
CASE m.tnWhich < 1
   * Invalid
   nResult = -1

CASE INLIST(m.tnWhich, 1, 2)
   nResult = 1

OTHERWISE
   * Set initial values
   nPrev2 = 1
   nPrev1 = 1

   FOR nI = 3 TO m.tnWhich
      nResult = m.nPrev1 + m.nPrev2
      nPrev2 = m.nPrev1
      nPrev1 = m.nResult
   ENDFOR

ENDCASE

RETURN m.nResult
```

But in this case, the more readable recursive function is so much less efficient that there's simply no excuse for using it. In my testing, starting with the 3rd Fibonacci number, the iterative version starts out about 1.5 times faster. By the 10th Fibonacci number, the loop is 24 times faster, and by the 30th Fibonacci number, the loop is nearly 150,000 times faster.

Why? What's going on here? First, of course, even for the smallest values, the recursive version calls itself twice, but it's much worse than that. Take a look at what happens when you call the recursive Fibonacci function with an initial value of 6.

Initial call passes 6

Makes two calls 5 4

Each two calls 4 3 3 2

Etc. 3 2 2 1 2 1

Etc. 2 1

That's 15 calls to the function compute the 6th Fibonacci number. For the 7th you do all of those, plus the 9 calls it takes to compute 5th Fibonnaci number, plus 1 for the initial call, for a total of 25 calls.

Figure 1 shows the number of times the recursive version is called for each initial value from 3 to 20. A little effort indicates a simple formula for the number of calls, shown in **Listing 7**. In other words, the number of calls grows in almost the same way the Fibonacci numbers themselves grow. Although the number of calls doesn't quite double each time, this is exponential growth.

Listing 7. This formula specifies the number of calls to the recursive Fibonacci function for a given initial value.

```
* Let NumCalls(n) be the number of calls when computing the nth Fibonacci number.
NumCalls(n+1) = NumCalls(n) + NumCalls(n-1) + 1
```

Ivalue	Irecurcalls
3	3
4	5
5	9
6	15
7	25
8	41
9	67
10	109
11	177
12	287
13	465
14	753
15	1219
16	1973
17	3193
18	5167
19	8361
20	13529

Figure 1. The number of calls to the recursive Fibonacci function goes up fast.

The lesson here is that sometimes even something that is naturally recursive shouldn't be implemented with recursion. You need to think about how much work a recursive implementation must do.

Putting recursion to work

Now that we've looked at where you shouldn't use recursion, let's look at places where you should. A common theme of these cases is "drilling down," that is, situations where we need to walk through a hierarchy to get to the bottom.

Folders

Windows offers a hierarchical arrangement of folders (directories), where any folder can contain one or more additional folders, as well as individual files. There are many

situations where we need to drill down through the folder hierarchy, whether to get a list of all files, to create or delete certain files, or just to get a list of all the folders.

Drilling through a folder hierarchy without recursion means keeping track of the folders we still need to go through along with the results. Recursion makes the code simple. Because there is a limit on the number of characters in a fully-pathed filename (260), it's extremely unlikely you'd have stack size issues. In general, folder hierarchies are more likely to be broad than deep.

The termination case for drilling down through folders is reaching a folder that contains no additional folders.

When I first wrote this routine (many years ago), I stored the list of folders in an array. That version (DrillDownFolders.PRG in the materials for this session) is shown in **Listing 8**. After some initial set-up work, it uses ADIR() to get a list of folders in the specified folder, and then loops through that list. Each folder found is added as a new row in the array. Then, the routine is called recursively for that folder. This means the list of folders in the array is in what's called *depth-first order*, where we drill all the way to the bottom, then go back to the next item at the second-from-bottom level, drill to the bottom again, go up again and so on and so forth. (The alternative to depth-first order is *breadth-first order* where you show everything at one level before going on to the next level. Breadth-first is more difficult when using recursion; see "Recursion with multiple routines," later in this paper, for an example.) **Figure 2** shows partial results from calling DrillDownFolders, passing "D:\Writing" (the folder under which all my writing projects are stored) as the start folder.

Listing 8. Recursion makes it easy to drill down through the Windows folder hierarchy. This version stores the list of folders in an array.

```
* DrillDownFolders.PRG
* Fill an array with the list of all
* folders and subfolders in a particular
* folder. Recursive.

LPARAMETERS aResults, cFolder, lStartOver
   * aResults = array to hold results.
   * cFolder = start folder
   * lStartOver = should we empty aResults first.

LOCAL aCurFolder[1], nFolders, nFolder
LOCAL nResultCount, cOrigFolder, cFoundFolder

WAIT WINDOW LEFT("Processing folders inside " + m.cFolder,255) NOWAIT

SET ESCAPE ON

IF m.lStartOver
   DIMENSION aResults[1]
   nResultCount = 0
ELSE
```

```
   nResultCount = ALEN(m.aResults, 1)
ENDIF

* Hold current folder
cOrigFolder = SYS(5) + CURDIR()

* Switch to specified folder, if we can
LOCAL lProceed
TRY
   CD (m.cFolder)
   lProceed = .T.
CATCH
   lProceed = .F.
ENDTRY

IF m.lProceed
   * Get list of contained folders
   nFolders = ADIR(aCurFolder, "", "D")

   * Loop through list. We can start at position 3
   * because 1 and 2 are always "." and ".."
   FOR nFolder = 3 TO m.nFolders
      * First add it, adding the starting path
      nResultCount = m.nResultCount + 1
      DIMENSION aResults[m.nResultCount]
      cFoundFolder = ADDBS(m.cFolder) + aCurFolder[m.nFolder, 1]
      aResults[m.nResultCount] = m.cFoundFolder

      * Now, drill down
      DrillDownFolders(@aResults, m.cFoundFolder, .F.)

      * Reset folder count
      nResultCount = ALEN(m.aResults, 1)
   ENDFOR

   CD (m.cOrigFolder)
ENDIF

RETURN m.nResultCount
```

Figure 2. Partial results from calling DrillDownFolders on the folder that contains all my writing projects. Note that they are in depth-first order.

While it's not an issue in VFP 9, earlier versions of VFP limited arrays to 65,000 elements. If you start high enough in the folder hierarchy, it's not hard to imagine finding more than 65,000 subfolders at all levels. (Running the function against the root of my C: drive found more than 129,000 folders.)

For that reason, and because VFP was designed to work with tables, I created a modified version that puts the results into a cursor. It's shown in **Listing 9** and included in the materials for this session as DrillDownFoldersToCursor.PRG. The recursive logic is identical to the earlier version; the only difference is in saving the folders, and in the first parameter passed to the function, which is the name of the cursor.

Listing 9. This version of the code to collect a list of folders puts the results into a cursor.

```
* DrillDownFoldersToCursor.PRG
* Populate a cursor with the list of all
* folders and subfolders in a particular
* folder. Recursive.

LPARAMETERS cCursor, cFolder, lStartOver
```

```
   * cCursor = name of cursor to hold results.
   * cFolder = start folder
   * lStartOver = should we empty cursor first.

LOCAL aCurFolder[1], nFolders, nFolder
LOCAL nResultCount, cOrigFolder, cFoundFolder

WAIT WINDOW LEFT("Processing folders inside " + m.cFolder,255) NOWAIT

SET ESCAPE ON

IF NOT USED(m.cCursor)
   CREATE CURSOR (m.cCursor) (mFolder M)
ENDIF

IF m.lStartOver
   ZAP IN (m.cCursor)
   nResultCount = 0
ELSE
   nResultCount = RECCOUNT(m.cCursor)
ENDIF

* Hold current folder
cOrigFolder = SYS(5) + CURDIR()

* Switch to specified folder, if we can
LOCAL lProceed
TRY
   CD (m.cFolder)
   lProceed = .T.
CATCH
   lProceed = .F.
ENDTRY

IF m.lProceed

   * Get list of contained folders
   nFolders = ADIR(aCurFolder, "", "D")

   * Loop through list. We can start at position 3
   * because 1 and 2 are always "." and ".."
   FOR nFolder = 3 TO m.nFolders
      * First add it, adding the starting path
      cFoundFolder = ADDBS(m.cFolder) + aCurFolder[m.nFolder, 1]
      INSERT INTO (m.cCursor) VALUES (m.cFoundFolder)

      * Now, drill down
      DrillDownFoldersToCursor(m.cCursor, m.cFoundFolder, .F.)

      * Reset folder count
      nResultCount = RECCOUNT(m.cCursor)
   ENDFOR

   CD (m.cOrigFolder)
ENDIF
```

```
RETURN m.nResultCount
```

If what you really want to do is collect file names, it's easy to adapt this code. **Listing 10** (included in the materials for this session as DrillDownFilesToCursor.PRG) shows the code. There are two significant changes. The first is in the call to ADIR(). The version in the previous functions was designed to ignore files and return only folders. In this function, the call is set up to return both. The second change is that inside the loop, we determine whether a given item is a file or a folder. If it's a folder, we call the function recursively. If it's a file, we save it to the cursor.

Listing 10. Now that we have the basic structure, creating a function to collect all file names in a given folder hierarchy is easy.

```
* DrillDownFilesToCursor.PRG
* Populate a cursor with the list of all
* files found in a particular folder
* and its subfolders. Recursive.

LPARAMETERS cCursor, cFolder, lStartOver
   * cCursor = name of cursor to hold results.
   * cFolder = start folder
   * lStartOver = should we empty cursor first.

LOCAL aCurItems[1], nItems, nItem
LOCAL nResultCount, cOrigFolder, cFoundItem

WAIT WINDOW LEFT("Processing files inside " + m.cFolder,255) NOWAIT

SET ESCAPE ON

IF NOT USED(m.cCursor)
   CREATE CURSOR (m.cCursor) (mFolder M, mFile M)
ENDIF

IF m.lStartOver
   ZAP IN (m.cCursor)
   nResultCount = 0
ELSE
   nResultCount = RECCOUNT(m.cCursor)
ENDIF

* Hold current folder
cOrigFolder = SYS(5) + CURDIR()

* Switch to specified folder, if we can
LOCAL lProceed
TRY
   CD (m.cFolder)
   lProceed = .T.
CATCH
   lProceed = .F.
ENDTRY
```

```
IF m.lProceed

   * Get list of contained files and folders
   nItems = ADIR(aCurItems, "*", "D")

   * Loop through list. We can start at position 3
   * because 1 and 2 are always "." and ".."
   FOR nItem = 3 TO m.nItems
      * First add it, adding the starting path
      IF 'D' $ aCurItems[m.nItem, 5]
         * It's a folder, so drill down
         cFoundItem = ADDBS(m.cFolder) + aCurItems[m.nItem, 1]
         DrillDownFilesToCursor(m.cCursor, m.cFoundItem, .F.)
      ELSE
         * It's a file. Save it
         INSERT INTO (m.cCursor) VALUES (m.cFolder, aCurItems[m.nItem, 1])
      ENDIF

      * Reset folder count
      nResultCount = RECCOUNT(m.cCursor)
   ENDFOR

   CD (m.cOrigFolder)
ENDIF

RETURN m.nResultCount
```

Object hierarchies

VFP classes, forms, and other object hierarchies also lend themselves very naturally to recursion. I use recursion extensively in my base classes (included in the materials for this session as TSBase.VCX and TSBase.PRG) and in code that needs to walk through an object hierarchy. This section explores a few of those uses. (For some more examples, see https://tinyurl.com/2hwde7tn.)

VFP provides several language elements that make drilling into object hierarchies easy. First, every control capable of holding other objects (forms, containers, grids, etc.) has an Objects collection that holds an object reference to each member of the collection.

The FOR EACH loop makes it easy to go through collections. Such loops show up again and again in these examples. Sometimes, you're dealing with object hierarchies you've built, not the ones native to controls. In that case, you don't have the Objects collection, but assuming you're storing object references in properties and creating your own arrays or collections of references, you can still use FOR EACH (or a counted FOR loop) to go through and touch each object you're interested in.

The PEMStatus() function makes it easy to find out whether the object you're currently looking at has an Objects collection (that is, is a container), as well as determining whether it has whatever property or method you're interested in. For example, the call in **Listing 11** checks whether the object referenced by oObject has a Valid method.

Listing 11. The PEMStatus() function lets you check whether a given object has a specified PEM.

```
IF PEMStatus(m.oObject, 'Valid', 5)
   * Do something
ENDIF
```

The AMembers() function fills an array with a list of members (properties, events, and methods) of a specified object. You pass parameters to indicate what members you're interested in.

When using recursion to drill down through object hierarchies, the termination case (the one that ends the recursive calls) isn't as explicit as in the factorial and Fibonacci examples. What terminates the stack of calls is reaching a container that has no containers among its contents.

Setting up event binding

I often build container classes that represent a single thing on a form. In those cases, I want controls inside the container to refer a lot of events and other control to the container itself. For example, in many situations, I want a click on any control in the container to fire the container's Click method.

With BindEvent(), that's easy, except that I need to make sure the Click of every control inside the container is bound to the container, even if there are multiple levels of containers. So, my base container class has a method called BindClick, containing the code in **Listing 12**. The method receives a container object as parameter. It then loops through that container's Objects collection and does two things. First, if the contained object has a Click method, it binds the Click method of the contained object to Click method of the container. Second, if the contained object has an Objects collection of its own, it calls itself recursively, passing in that contained container object. In that way, the method drills down from whatever container is passed initially to every contained control.

Listing 12. This method of my base Container class, called BindClick, ensures that the Click method of every object inside the container is bound to the Click method of the container itself. That allows centralized handling of clicks anywhere on the container.

```
LPARAMETERS oContainer

FOR EACH oObject IN oContainer.Objects FOXOBJECT
   IF PEMSTATUS(oObject, "Click", 5)
      BINDEVENT(oObject, "Click", This, "Click")
   ENDIF

   IF PEMSTATUS(oObject, "Objects", 5)
      This.BindClick(m.oObject)
   ENDIF
ENDFOR
```

Of course, I may not want that behavior for every container, so the class also has a logical property, lBindClick. The container class's Init method includes the code in **Listing 13** to determine whether to bind controls within a particular container.

Listing 13. This code in my base container class's Init method determines whether clicks for contained controls are bound to the container's Click method.

```
IF This.lBindClick
   This.BindClick(This)
ENDIF
```

I have similar code for DblClick and MouseDown. Binding MouseDown for all controls in a container allows smoother drag-and-drop of the entire container.

I also have code in my base form class that helps me determine whether anything on the form has changed. Each of my base data entry controls has a custom property, lNoteChange, and a custom method called AnyChange. The InteractiveChange and ProgrammaticChange methods raise the AnyChange method (using the RaiseEvent() function).

The form class has a custom method called BindControlEvents that contains the code in **Listing 14**. BindControlEvents receives a container control as a parameter. It loops through the container's Objects collection, looking at each contained object. If the contained object has the lNoteChange property and that property is true, the AnyChange method of the object is bound to the form's custom AnyChange method. Then, if the contained object has an Objects collection of its own, BindControlEvents is called recursively, passing in the contained object, so we can drill down through all the containers.

Listing 14. This method, called BindControlEvents, is in my form base class. It allows me to have a form method fire when data changes anywhere on the form.

```
* Bind events of controls to events of the form as appropriate
LPARAMETERS toControl

LOCAL oControl

FOR EACH oControl IN toControl.Objects
   IF PEMSTATUS(oControl, "lNoteChange", 5) AND oControl.lNoteChange
      BINDEVENT(oControl, "AnyChange", This, "AnyChange")
   ENDIF

   IF PEMSTATUS(oControl, "Objects", 5)
      This.BindControlEvents(oControl)
   ENDIF

ENDFOR
```

The form class has a custom lNoteUserChanges property to determine whether we want to track changes on this form. (Often, I want to do so for data entry forms, but not for dialogs.)

The form class's Init method contains the code in **Listing 15** to start the binding process. The form itself is passed as a parameter.

Listing 15. This code in the Init method of my base form class starts the process of binding changes in all controls to a single form method.

```
IF This.lNoteUserChanges
   * This one handles changes to data.
   This.BindControlEvents(This)
ENDIF
```

With this setup, I can do things like modify a form's title bar to indicate that there's changed data in the form and prompt the user to save on exit only if data has changed.

Modifying font sizes

Many years ago, I figured out how to allow users to choose fonts and sizes and properly resize forms and controls, so they still look good. (My original article on the subject is at https://tinyurl.com/bmndzx33.) While none of my clients have wanted that exact capability in their applications, a couple of them have wanted to let users resize forms and change the font size to accommodate the new form size. The code to handle this (a class called cusFontSize, included in the materials for this session in FontSizer.VCX) uses recursion both in the initial set-up and when the form is resized. (This class assumes that all controls have Anchor set to 240, which ensures they resize and reposition appropriately when the form is resized.)

Resizing of a control's font is done proportionally to the control's original size and its original font size, so we need a way to store the original size. The class's Init method receives a container (presumably the form) as a parameter and contains the call in **Listing 16**.

Listing 16. The font resizing class's Init method calls SaveOriginal to store the original size of every object on the form.

```
This.SaveOriginal(oContainer)
```

The custom SaveOriginal method, shown in **Listing 17**, receives an object as a parameter, stores its size characteristics in properties it adds on the fly and then, if the object is a container, loops through the list of contained objects and calls itself recursively for each contained object.

Listing 17. The SaveOriginal method adds properties to the object passed in to hold its original height, width, and font size, and then calls itself recursively for any objects contained in this object.

```
LPARAMETERS oObject

IF PEMSTATUS(oObject, "Height", 5)
   AddProperty(oObject, "nOriginalHeight", oObject.Height)
ENDIF

IF PEMSTATUS(oObject, "Width", 5)
```

```
   AddProperty(oObject, "nOriginalWidth", oObject.Width)
ENDIF

IF PEMSTATUS(oObject, "FontSize", 5)
   AddProperty(oObject, "nOriginalFontSize", oObject.FontSize)
ENDIF

* Drill down
* Modified 7-August-2017 by TEG
* Although Control class has Objects property, it's not accessible.
IF PEMSTATUS(oObject, "Objects", 5) AND NOT UPPER(oObject.BaseClass) == "CONTROL"
   FOR EACH oContained IN oObject.Objects FOXOBJECT
      This.SaveOriginal(oContained)
   ENDFOR
ENDIF

RETURN
```

The class has a method CatchResize, which is bound to the form's Resize event. It contains the code in **Listing 18**. The method uses AEVENTS() to figure out what object was resized and then passes that object to the ChangeFontSize method. Because we want controls that used the same font size initially to use the same font size after a resize, this method computes a new font size based on the form's original size, its new size and its original font size.

Listing 18. When the form containing cusFontSize is resized, the CatchResize method fires to start the resizing process.

```
LOCAL nBound, aBoundEvent[1]

nBound = AEVENTS(aBoundEvent, 0)

IF nBound <> 0
   * Compute new font size for entire form here and save it.
   oTrigger = aBoundEvent[1]
   nNewSize = This.ComputeNewSize(oTrigger.nOriginalHeight * ;
                                  oTrigger.nOriginalWidth, ;
                                  oTrigger.Height * oTrigger.Width, ;
                                  oTrigger.nOriginalFontSize)
   This.nNewFontSize = m.nNewSize
   This.ChangeFontSize(aBoundEvent[1])
ENDIF

RETURN
```

ChangeFontSize, shown in **Listing 19**, recursively drills down to do the actual resizing. This method differs from the recursive drill-down routines we've looked at so far, because it makes the recursive call first and then does what it's supposed to in the current call. That's necessary in this case because we need to change the font size in contained objects before we change the font size in their containers. If the object passed in has the same original font size as the form itself, the routine uses the new font size computed for the form. If not, the routine computes a new font size for this control. Then, it makes sure that the actual text in

the control fits using that font size and, if necessary, reduces the computed font size. (Why wouldn't the text fit? Imagine a label with WordWrap set to .T. If one line is much longer than the others, the computed font size might be too big.)

Listing 19. The ChangeFontSize method drills down to the lowest-level controls and changes font sizes from innermost to outermost.

```
LPARAMETERS oObject

LOCAL nNewFontSize

* Should we change this one?
IF PEMSTATUS(oObject, "lChangeFontOnResize", 5)
   IF oObject.lChangeFontOnResize
      * Work from the bottom up, so changes to container don't affect contents
      IF PEMSTATUS(oObject, "Objects", 5)
         * Drill down
         FOR EACH oContained IN oObject.Objects FOXOBJECT
            This.ChangeFontSize(oContained)
         ENDFOR
      ENDIF

      * Change this object
      * Modified 9-January-2008 by TEG
      * Added test for added properties to ensure we don't error here.

      IF PEMSTATUS(oObject, "FontSize", 5) AND ;
         PEMSTATUS(oObject, "nOriginalHeight", 5)
         * Use previously computed size unless original
         * font size was different
         IF oObject.nOriginalFontSize = This.nOriginalFontSize
            nNewFontSize = This.nNewFontSize
         ELSE
            nNewFontSize = This.ComputeNewSize(oObject.nOriginalHeight * ;
                                               oObject.nOriginalWidth, ;
                                               oObject.Height * oObject.Width, ;
                                               oObject.nOriginalFontSize)
         ENDIF

         * Check that it fits in available space
         DO WHILE NOT This.CheckTextFits(oObject, m.nNewFontSize) AND ;
                  m.nNewFontSize >= This.nMinFontSize
            nNewFontSize = m.nNewFontSize - 1
         ENDDO

         oObject.FontSize = m.nNewFontSize

      ENDIF
   ENDIF
ENDIF

RETURN
```

The resizer is connected to the form by instantiating it in the form's Init method, as in **Listing 20**. The form's lChangeFontOnResize property lets you decide whether font sizes are changed when a given form is resized.

Listing 20. This code in the form's Init instantiates the font resizing object and passes the form as a parameter, so that initial sizes can be saved.

```
IF This.lChangeFontOnResize
   This.oFontResizer = NEWOBJECT("cusFontSize", "FontSizer", "", This)
ENDIF
```

Modify controls and code

About 15 years ago, I was working on a project with many forms where the original developer often hadn't renamed the controls from the defaults VFP assigns. Some of these forms contained dozens of controls. Making changes when everything had names like Text4 and Command22 was hard, but renaming the controls meant I had to be sure to fix all the code that mentioned them, too. These forms used pageframes and other containers, and the controls inside those mostly hadn't been renamed either, so I couldn't assume that Text4 was always the control I was working on.

I ended up creating a tool to let me rename the controls and fix the code automatically. The tool, called Control Renamer Builder, is available through VFPX (https://github.com/VFPX/ControlRenamerBuilder); my article about how it works and how to use it is at https://tinyurl.com/58m2hydt. (The code is available on the VFPX site, so it's not included in the downloads for this session.) You can see the Control Renamer Builder at work in **Figure 3**.

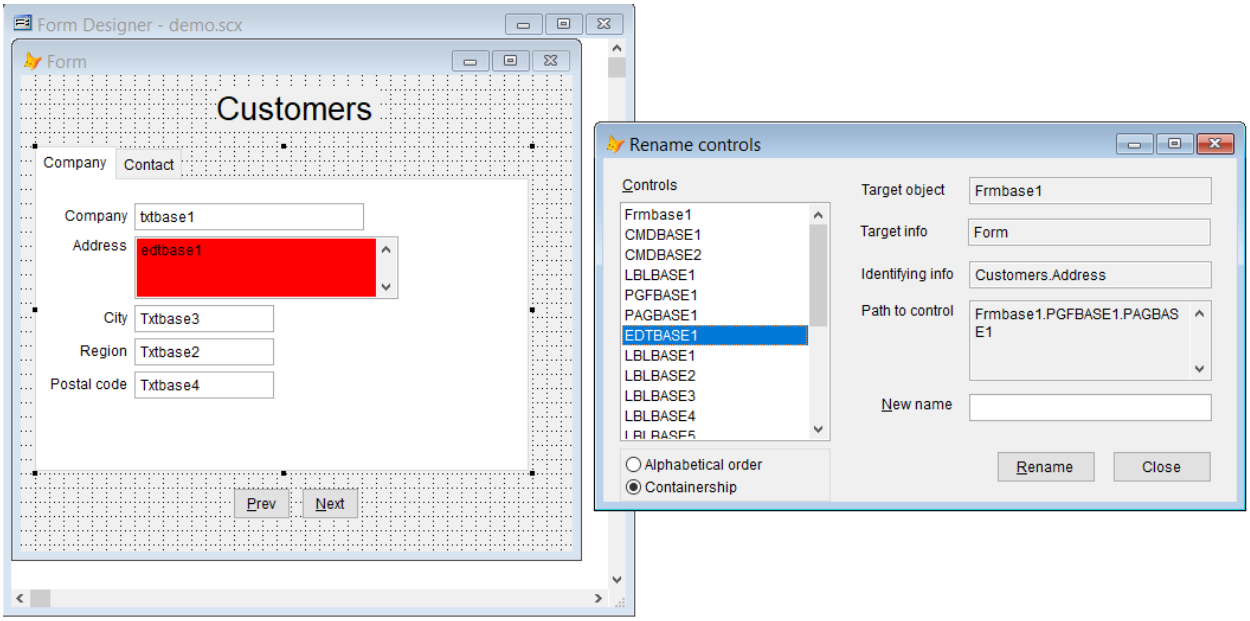

Figure 3. The Control Renamer Builder makes it easy to rename controls on a form without breaking code.

This tool uses recursion to drill down into the object hierarchy a couple of times. First, recursion is used to build a cursor containing information about every control on the form. The custom GrabControls method, shown in **Listing 21**, starts the process by creating a record for the "target object," the form or container whose controls we want to rename. It then calls the recursive DrillControls method to drill down.

Listing 21. This method collects information about the form or control whose controls we want to rename, puts that into a cursor, and calls the recursive DrillControls method to collect information about the contents.

```
PROCEDURE GrabControls
* Traverse the form/container and populate the cursor of controls
LOCAL cInfo

* Add the form itself, then drill down
cInfo = This.GetInfo(This.oObject)
INSERT INTO AllControls ;
      (cControlOrig, mFullPath, cInfo, lReadOnly) ;
      VALUES (This.oObject.Name, "", m.cInfo, .F.)

This.DrillControls(This.oObject, This.oObject.Name)

* Now get a list of unique names
SELECT DISTINCT cControlOrig ;
   FROM AllControls ;
   INTO CURSOR ControlNames

GO TOP IN AllControls

RETURN
```

DrillControls, shown in **Listing 22**, is a little more complicated, but not much. It uses AMEMBERS()to get a list of the controls in the current container. Then, it loops through that list, collecting information about each control, including whether the control's name can be changed. (Controls that are part of a container class cannot be renamed.) Then, it adds that control to the cursor. Finally, if the control is itself a container, DrillControls is called recursively.

Listing 22. This method drills down through an object hierarchy, putting information about each control into a cursor.

```
PROCEDURE DrillControls(oContainer, cHierarchy)
* Drill into a container and add all the controls
* inside to the cursor.

LOCAL nControls, aControls[1], nControl, oObject
LOCAL nPEMs, aPEMs[1], nNameRow, lReadOnly

nControls = AMEMBERS(aControls, oContainer, 2)

FOR nControl = 1 TO nControls
   * Figure out what info is available about
   * this control
```

```
   oObject = EVALUATE("oContainer." + aControls[nControl])
   cInfo = This.GetInfo(oObject)

   * Find out whether name can be changed.
   IF UPPER(oObject.BaseClass) <> "OLE"
      nPEMs = AMEMBERS(aPEMs, oObject, 3, "#+")
      nNameRow = ASCAN(aPEMs,"NAME",-1,-1,1,15)
      IF nNameRow <> 0
         lReadOnly = "R"$aPEMs[nNameRow,5]
      ELSE
         lReadOnly = .T.
      ENDIF
   ENDIF

   INSERT INTO AllControls ;
      (cControlOrig, mFullPath, cInfo, lReadOnly) ;
      VALUES (aControls[nControl], cHierarchy, m.cInfo, m.lReadOnly)

   * Drill down
   IF PEMSTATUS(oObject, "Objects", 5)
      * Drill down
      This.DrillControls(oObject, ;
                         cHierarchy + "." + aControls[nControl])
   ENDIF
ENDFOR

RETURN nControls
```

The second parameter to DrillControls is essential to the functioning of this tool. It contains the path through the object hierarchy to the control we're now working on. When GrabControls starts the process, it passes the name of the target object. Each recursive call then adds a period and the name of the control that's being passed in. So, for example, if a form named frmOrder has a container named cntCustomer, when DrillControls is called for that container, the second parameter would be "frmOrder.cntCustomer". This information helps us get access to the right object later in the process.

The Control Renamer Builder uses recursion a second time to find all the places in code where each control is referenced. The BuildCodeRefs method starts things off by calling the recursive DrillCode method, passing the target object as a parameter.

DrillCode, shown in **Listing 23**, uses AMEMBERS() to get a list of events and methods for the current object. For each event or method, it calls the AuditMethod method, which searches through the code of the specified method to find any references to any control that was identified by GrabControls, storing what it finds in another cursor. After all the methods and events have been audited, if the current object is a container, DrillCode is called recursively for each contained object. (As the comments note, I found a bug involving Grid.Objects. It's discussed in more detail in my article about this tool.)

Listing 23. The DrillCode method of the Control Renamer Builder's engine class, checks each method of the specified control for references to controls, and then calls itself recursively for any contained objects.

```
PROCEDURE DrillCode(oControl)

LOCAL aAllMem[1], nMembCount, nObject, nMember

* Make sure list of controls to search for exists
IF NOT USED("ControlNames")
   RETURN .F.
ENDIF

nMembCount = AMEMBERS(aAllMem, oControl, 1)
FOR nMember = 1 TO nMembCount
   IF INLIST(UPPER(aAllMem[nMember, 2]), "EVENT", "METHOD" )
      This.AuditMethod(oControl, aAllMem[nMember,1])
   ENDIF
ENDFOR

* Work around bug with Grid.Objects
DO CASE
CASE UPPER(oControl.BaseClass)="GRID"
   IF oControl.ColumnCount > 0
      FOR nObject = 1 TO oControl.ColumnCount
         This.DrillCode(oControl.Columns[ nObject ])
      ENDFOR
   ENDIF

OTHERWISE
   IF PEMSTATUS(oControl, "Objects", 5)
      FOR nObject =1 TO oControl.Objects.Count
         This.DrillCode(oControl.Objects[nObject])
      ENDFOR
   ENDIF
ENDCASE

RETURN
```

The tool contains a lot more code, but the rest isn't recursive. If you're interested, check out the linked article.

The Registry

The Windows Registry is another hierarchy and so recursion is a good way to navigate through it. The Registry has a set of root nodes (HKEY_CURRENT_USER, etc.), each of which contains a bunch of keys and values. A key can have subkeys, and those subkeys can have subkeys, and so on. Each key can have one or more values.

There's a class in the FoxPro Foundation Classes (FFC) that come with VFP that simplifies working with the Registry. Otherwise, you'd need to use a lot of API calls. (I wrote about the class many years ago: https://tinyurl.com/3kpdrued.)

The class's OpenKey method lets you focus on a particular key. (Think of it like selecting a work area in VFP.) Then the EnumKeyValues method puts the list of values for that key into an array. **Listing 24** uses this method to fill an array with the values of the Options key for VFP 9. **Figure 4** shows part of that section of the Registry on the computer where I'm writing this paper.

Listing 24. The FFC Registry class has a method to retrieve all the values for a given key.

```
#DEFINE HKEY_CURRENT_USER -2147483647
LOCAL aOptionsValues[1,2], oRegistry

oRegistry = NewObject( "Registry", HOME() + "FFC\Registry.VCX")

nError = oRegistry.OpenKey( ;
  "Software\Microsoft\VisualFoxPro\9.0\Options", ;
  HKEY_CURRENT_USER, .f.)

IF m.nError = 0
   nError = oRegistry.EnumKeyValues( @aOptionsValues )
ELSE
   MESSAGEBOX("OpenKey failed: " + TRANSFORM(m.nError))
   RETURN
ENDIF

IF m.nError = 0
   MESSAGEBOX("Success. Found " + TRANSFORM(ALEN(aOptionsValues, 1)) + " values.")
ELSE
   MESSAGEBOX("EnumKeyValues failed: "+ TRANSFORM(m.nError))
ENDIF
```

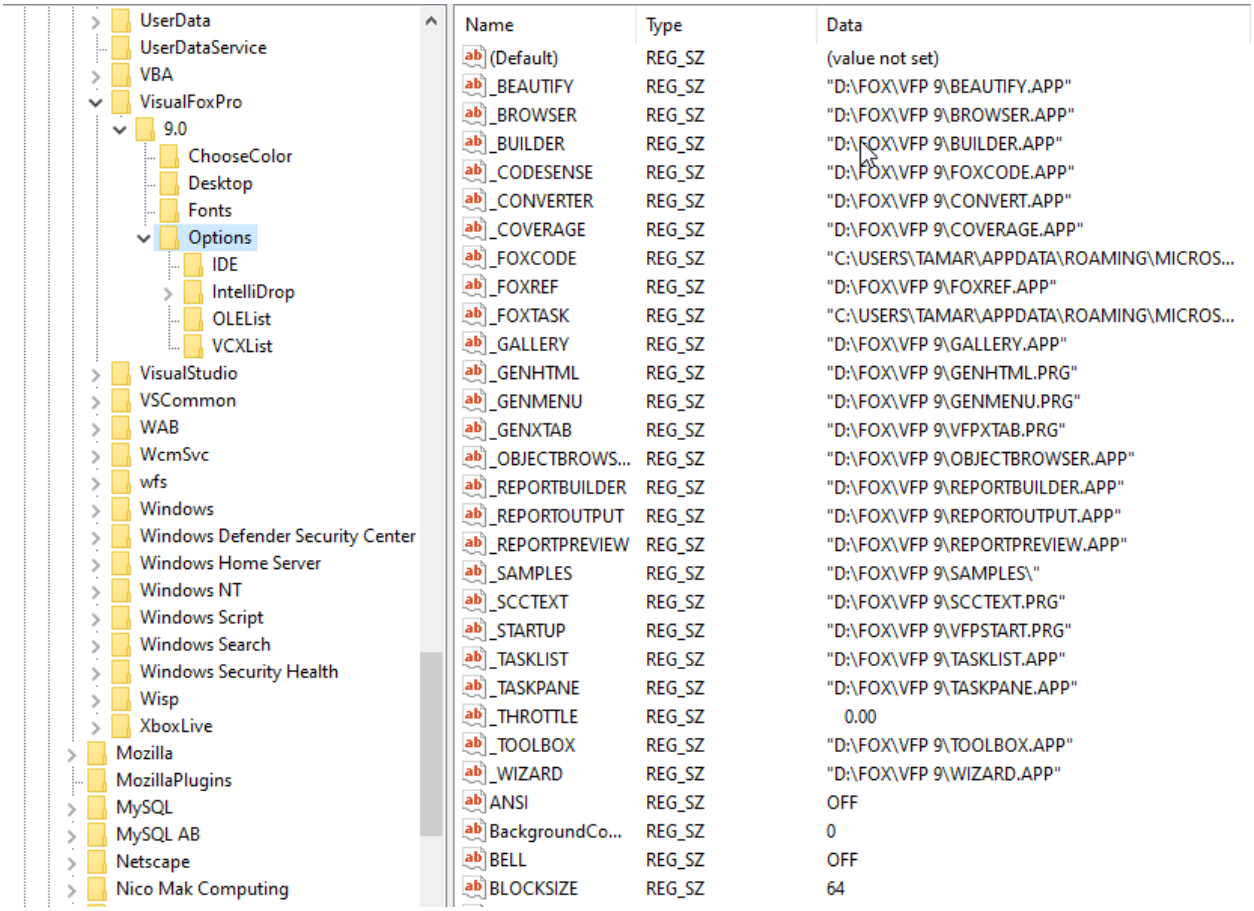

Figure 4. The Options key for VFP 9.0 in the Registry contains many values, and also has several subkeys.

However, as the figure shows, the Options key has several subkeys. What if you want to collect not just the values for that key, but also for all its subkeys and their subkeys and so on? That's a job for recursion.

Another method of the Registry class, EnumKeys, lets you fill an array with subkeys of the key you opened with OpenKey. I used all three methods (OpenKey, EnumKeyValues, and EnumKeys) to create a function that drills down and puts all the values for a specified key, including values of its subkeys, into an array. It's shown in Error! Not a valid bookmark self-reference. and included in the materials for this session as aGetKeys.PRG. Like VFP's native "a" functions, it expects an array as the first parameter, though you have to pass it by reference (which you don't for the native "a" functions).

After checking the parameters, the function instantiates the Registry class, if necessary. Then, it opens the specified key and uses EnumKeyValues to get the key's values. Those values are copied into the array passed into the function. Because the routine is recursive, the copying code first checks how many items are already in the array and expands the array from that point. Then, EnumKeys gets a list of subkeys and loops through that list, calling the function recursively for each, passing the same array in to hold the results. The termination case is having EnumKeys return an empty array.

Listing 25. This function accepts a Registry key and drills down to put all values of the key and its subkeys into an array.

```
* aGetKeys.PRG
* Get all keys and subkeys of a specified key
* with their values.
* This routine is recursive.
LPARAMETERS aAllKeys, cKey, nRoot, oRegistry
   * aAllKeys = the array to hold the keys and values
   * cKey = the key to start with
   * nRoot = the root node in the Registry
   * oRegistry = object reference to an instance of
   *             the Registry class

* Check parameters
ASSERT TYPE( "aAllKeys[1]") <> "U" ;
   MESSAGE "aGetKeys: First parameter must be an array"

IF TYPE( "aAllKeys[1]") = "U"
   ERROR 232,"aAllKeys"
   RETURN -1
ENDIF

ASSERT VARTYPE( cKey ) = "C" AND NOT EMPTY( cKey );
   MESSAGE "aGetKeys: Second parameter (cKey) must be " + ;
   "character and not empty"

IF VARTYPE( cKey ) <> "C" OR EMPTY( cKey )
   ERROR 11
   RETURN -1
ENDIF

ASSERT VARTYPE( nRoot ) = "N" AND NOT EMPTY( nRoot) ;
   MESSAGE "aGetKeys: Third parameter (nRoot) must be " + ;
   "numeric and not empty"

IF VARTYPE( nRoot ) <> "N" OR EMPTY( nRoot)
   ERROR 11
   RETURN -1
ENDIF

* Check for a registry object
IF VARTYPE( oRegistry ) <> "O"
   oRegistry = NEWOBJECT( "Registry", ;
                          HOME() + "FFC\Registry")
ENDIF

LOCAL aValues[1], aKeys[1]
LOCAL nCountSoFar, nValues, nItem, nKey

* Open the key and get started
WITH oRegistry
   IF .OpenKey( cKey, nRoot ) = 0
      * Get the values at this level and
      * copy them to the master array
```

```
      IF .EnumKeyValues( @aValues ) = 0
         nCountSoFar = ALEN( aAllKeys, 1)
         IF EMPTY(aAllKeys[1,1])
            nCountSoFar = 0
         ENDIF

         nValues = ALEN( aValues, 1)
         DIMENSION aAllKeys[ nCountSoFar + nValues, 3 ]

         FOR nItem = 1 TO nValues
            aAllKeys[ nCountSoFar + nItem, 1]= cKey
            aAllKeys[ nCountSoFar + nItem, 2] = aValues[ nItem, 1]
            aAllKeys[ nCountSoFar + nItem, 3] = aValues[ nItem, 2]
         ENDFOR
      ENDIF

      * Now get the keys at this level
      IF .EnumKeys( @aKeys ) = 0
         nKeyCount = ALEN( aKeys )
         FOR nKey = 1 TO nKeyCount
            aGetKeys( @aAllKeys, ;
                      ADDBS(cKey) + aKeys[ nKey ], nRoot, oRegistry )
         ENDFOR
      ENDIF
   ENDIF
ENDWITH

RETURN ALEN(aAllKeys, 1)
```

Call the function with code like that in **Listing 26**. The function returns the total number of values found in the key and its subkeys. **Figure 5** shows part of the resulting array.

Listing 26. To call aGetKeys, pass an array, the desired key, and the constant for the appropriate root.

```
DIMENSION aMyKeys[1]
* -2147483647 is HKEY_CURRENT_USER
?aGetKeys(@aMyKeys, "Software\Microsoft\VisualFoxPro\9.0\Options", -2147483647)
```

amykeys[190,3]	"2bb"
amykeys[191,1]	"Software\Microsoft\VisualFoxPro\9.0\Options"
amykeys[191,2]	"RptExprBuilderAlias"
amykeys[191,3]	"1"
amykeys[192,1]	"Software\Microsoft\VisualFoxPro\9.0\Options"
amykeys[192,2]	"RptBehavior"
amykeys[192,3]	"80"
amykeys[193,1]	"Software\Microsoft\VisualFoxPro\9.0\Options"
amykeys[193,2]	"RptFont"
amykeys[193,3]	"Calibri, 12, N, 1"
amykeys[194,1]	"Software\Microsoft\VisualFoxPro\9.0\Options\IDE"
amykeys[194,2]	"Program"
amykeys[194,3]	"011000100011000404"
amykeys[195,1]	"Software\Microsoft\VisualFoxPro\9.0\Options\IDE"
amykeys[195,2]	"ProgramExtension"
amykeys[195,3]	"PRG;MPR;QPR;SPR"
amykeys[196,1]	"Software\Microsoft\VisualFoxPro\9.0\Options\IDE"
amykeys[196,2]	"ProgramFont"
amykeys[196,3]	"Courier New, 14, N, 1"

Figure 5. The array you pass to aGetKeys contains values from multiple levels.

Removing Registry keys also calls for recursion. You cannot remove a key if it has subkeys, so to remove a key from the Registry, you need to first find and delete any subkeys. But of course, a subkey may have subkeys, so you have to find and delete them first, and so on, until you get to a key with no subkeys.

WARNING: Do not just remove stuff from the registry. If you want to test this deletion code, first add some dummy keys and values. You can do so manually using the Registry Editor, or you can use the SetRegKey method of the Registry class. See my paper linked earlier for the syntax and examples of SetRegKey.

To demonstrate, I added a key below HKEY_CURRENT_USER, gave it a couple of values and three levels of subkeys. This tree is shown in **Figure 6**.

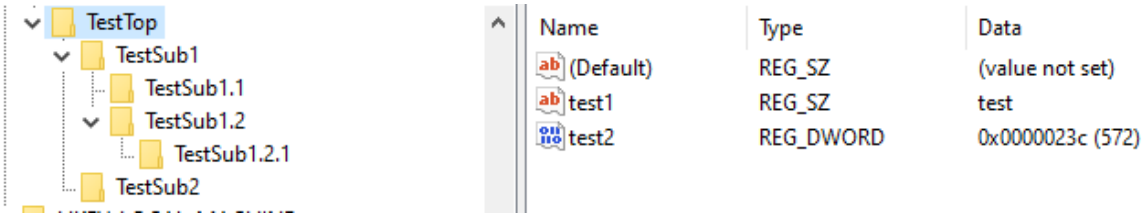

Figure 6. I added some keys and subkeys to my Registry to demonstrate deletion.

The Registry class has a DeleteKey method that accepts the root and the full key to delete. But as noted above, it only works if the key you specify has no subkeys. The function

DrillDownDelete, shown in **Listing 27** (and included in the materials for this session as DrillDownDelete.PRG), accepts a reference to the Registry object, the key you want to delete, and the root (called nRegKey in this code) and does the whole job. DrillDownDelete is another example where the recursion happens first and the work of the function afterward. After checking parameters, the function opens the specified key and retrieves a list of subkeys. If there are any, it calls itself for each of them. Once all the subkeys have been deleted, the function calls DeleteKey to delete the original key.

Listing 27. This function lets you delete a key and all its subkeys from the Registry.

```
* PROCEDURE DrillDownDelete
* Drill down into a key, deleting all subkeys,
* then deleting the key itself.
* This procedure is recursive.

LPARAMETERS oRegistry, cKey, nRegKey
   * oRegistry = object reference to Registry object
   * cKey = key to be deleted
   * nRegKey = hive from which key is to be deleted

* Check parameters
ASSERT VARTYPE(oRegistry) = "O" ;
   MESSAGE "DrillDownDelete: First parameter is required pointer to registry object"

IF VARTYPE(oRegistry) <> "O"
   ERROR 11
   RETURN .f.
ENDIF

ASSERT VARTYPE(cKey) = "C" AND NOT EMPTY(cKey) ;
   MESSAGE "DrillDownDelete: Second parameter is required registry key"

IF VARTYPE(cKey) <> "C" or EMPTY(cKey)
   ERROR 11
   RETURN .f.
ENDIF

ASSERT VARTYPE(nRegKey) = "N" AND NOT EMPTY(nRegKey) ;
   MESSAGE "DrillDownDelete: Third parameter is required registry node"

IF VARTYPE(nRegKey) <> "N" OR EMPTY(nRegKey)
   ERROR 11
   RETURN .F.
ENDIF

* set up an array to hold the list of subkeys
LOCAL aKeys[1], lSuccess, nSubkeyCount, nKey

lSuccess = .t.

WITH oRegistry
   IF .OpenKey(cKey, nRegKey) = 0
      IF .EnumKeys(@aKeys) = 0
```

```
         * Loop through subkeys
         nSubkeyCount = ALEN(aKeys, 1)
         FOR nKey = 1 TO nSubKeyCount
            lSuccess = DrillDownDelete( oRegistry, ;
                                        cKey + "\" + aKeys[nKey], ;
                                        nRegKey )
         ENDFOR
      ENDIF
      IF .DeleteKey(nRegKey, cKey) = 0
         lSuccess = lSuccess and .T.
      ELSE
         lSuccess = .F.
      ENDIF
   ELSE
      lSuccess = .F.
   ENDIF
ENDWITH

RETURN lSuccess
```

Listing 28 shows how to call DrillDownDelete for the Registry keys I added.

Listing 28. Deleting a key and all its subkeys is easy with the recursive DrillDownDelete function.

```
LOCAL oRegistry

oRegistry = NEWOBJECT( "Registry", HOME() + "FFC\Registry")
* -2147483647 is HKEY_CURRENT_USER
?DrillDownDelete(oRegistry, "TestTop", -2147483647)
```

Recursion with multiple routines

All the examples in this paper so far have involved a single recursive routine. Although many of them have a separate routine to kick off the process, once the recursive routine is called, all recursive calls come from that routine.

Some processes are more complicated and involve two (or more) routines that call each other and themselves recursively. Typically, in such cases, each routine contributes to the need to drill down into something.

The Object Inspector I created (available through Thor) has such a situation. You can read about the Object Inspector at https://tinyurl.com/3czbd85a, though it has been extended and added to Thor since that article was written. I wrote a separate article about how the tool works; it's at https://tinyurl.com/es47n2wy.

The Object Inspector was created because the VFP Debugger doesn't work very well with collections, and I was working on a project that had a large object hierarchy with many collections. You pass an object, which could be a collection, to the tool and it lets you see the object, its property values, and any objects it contains. For collections, it shows each

member. **Figure 7** shows the tool with information about a collection of objects representing countries.

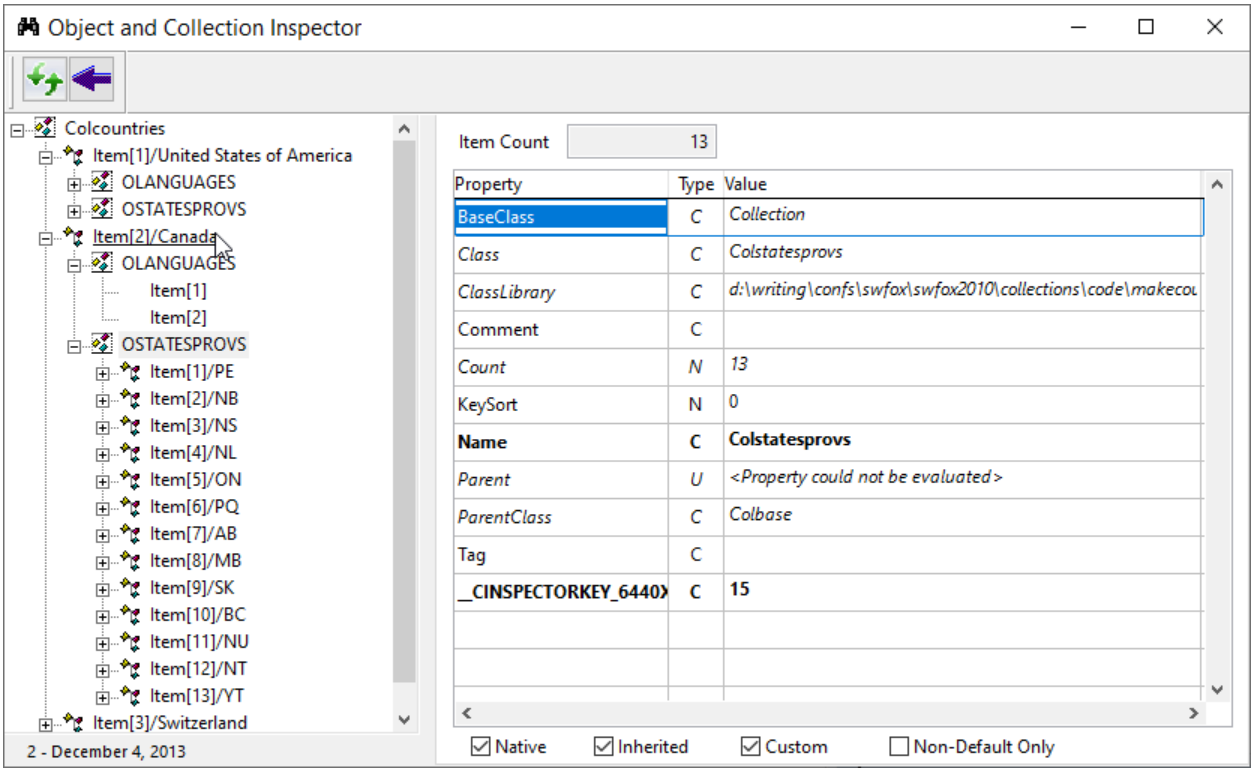

Figure 7. The Object Inspector uses recursion to collect information about collections and objects.

The Object Inspector uses Doug Hennig's Explorer classes. (His paper about this set of classes is at https://tinyurl.com/vc4b4hhf.) The key element is a treeview control; Doug created a wrapper class that contains the treeview and a couple of supporting classes. Most importantly, that wrapper class uses a cursor with certain columns to populate the treeview. The class has an abstract method called FillTreeViewCursor; you put code there to set the cursor up.

For the Object Inspector, we need to put a row in the cursor for each object in the hierarchy, including for each member of each collection. To add each member of a collection, we need to loop through the collection, and process each element. To add an object that's not a collection, we need to check every property of the object to see whether it's a reference to another object, and if so, process that object.

In the object hierarchies (controls on a form or class) we looked at earlier in this paper, we could be certain of reaching the bottom. But the Object Inspector needs to work with all kinds of object hierarchies, including those containing loops. The simplest case of a loop is two objects each containing references to each other, but loops can be much larger with a whole sequence of objects containing references to another object until eventually one points back to the first object.

In collecting the data for the Object Inspector, we need a way to track which objects we've already added to the cursor, so we don't end up with infinite recursion. We do that with a collection that contains one item for each object we've found. (The details are a little more complex than that—they're discussed in the "Inside the Object Inspector" article linked above.) The AddItemVisited method of that collection adds an item to the collection and assigns it a unique key.

Listing 29 shows the code in FillTreeViewCursor for the Object Inspector. (Code that logs what's going on has been omitted from all the Object Inspector methods shown here.) FillTreeViewCursor first sets up the collection of visited items. Then, it checks the "root item," the item that was passed into the tool and determines whether it's a collection or a scalar object, adds it to the collection of visited items, and calls the appropriate form method to process the collection or object. (The source for the Object Inspector is included when you add it to Thor, so is not in this session's downloads.)

Listing 29. Doug's treeview wrapper class has a method FillTreeViewCursor that's called to populate the treeview.

```
* Fill the cursor with info about the collection

* Set up the collection to track items we've seen,
* to avoid infinite recursion
ThisForm.oItemsVisited = NEWOBJECT("colItemsVisited", "ItemsVisited.PRG", ;
                                   "", ThisForm)
ThisForm.oItemsVisited.lAddToObject = ThisForm.lAddKeys

* First, the root item
IF PEMSTATUS(ThisForm.oRoot, "BaseClass", 5) AND ;
   UPPER(ThisForm.oRoot.BaseClass) = "COLLECTION"
   INSERT INTO (This.cCursorAlias) ;
      (ID, TYPE, TEXT, IMAGE, ;
       SORTED, PAGE, NODEKEY, REFID) ;
      VALUES ;
      ( "ROOT" , "TOP", ThisForm.cRootName, "Collection", ;
        .F., 1, This.GetNodeKey("TOP", "ROOT"), "")
   ThisForm.oItemsVisited.AddItemVisited(ThisForm.oRoot, "", "ROOT", "TOP")

   ThisForm.AddCollectionMembersToTreeViewCursor(ThisForm.oRoot, "ROOT", "TOP", ;
      This.cCursorAlias)
ELSE
   INSERT INTO (This.cCursorAlias) ;
      (ID, TYPE, TEXT, IMAGE, ;
       SORTED, PAGE, NODEKEY, REFID) ;
      VALUES ;
      ( "ROOT" , "TOP", ThisForm.cRootName, "Class", ;
        .F., 2, This.GetNodeKey("TOP", "ROOT"), "")
   ThisForm.oItemsVisited.AddItemVisited(ThisForm.oRoot, "", "ROOT", "TOP")

   ThisForm.FindObjectPropertiesForTreeViewCursor(ThisForm.oRoot, "ROOT", "TOP", ;
      This.cCursorAlias, 1)
ENDIF
```

```
RETURN
```

FillTreeViewCursor is the method that kicks things off. Two other methods do the actual work of drilling down and they call each other recursively.

For reasons I no longer remember (but I think are about which reference to an object is found first and thus displayed in full), it's best to do a breadth-first drilldown rather than depth-first. That is, I want to add all the children of a collection to the cursor before I drill down into those children and add all the object properties of an object before drilling down to the referenced objects. To do that, each of these methods uses an array to hold the list of recursive calls that need to be performed.

Listing 30 shows AddCollectionMembersToTreeviewCursor, which, as its name suggests, loops through a collection and adds each item to the cursor before drilling into those items (if they're objects).

From a recursion perspective, the key part of the code is the last part of the FOR loop. For each item in the collection, a row is added to the array laObjectInfo. It contains a flag to indicate whether this item is an object (vs. a scalar value), the type of item ("Class", "Collection", "Previous" (that is, an object we've seen before), "Nothing" (that is, scalar)), the unique ID we've assigned to the item, and the item's number (index) within the collection.

After that initial loop is finished, we loop through the array. If the item's type is "Collection", we call AddCollectionMembersToTreeviewCursor recursively. If the item's type is "Class" and we haven't seen it before, we call FindObjectPropertiesForTreeviewCursor (which is described below). If we've already seen it or it's scalar, we do nothing.

Listing 30. This method, AddCollectionMembersToTreeviewCursor, adds each member of a collection to the treeview cursor and then drills down into those items one by one.

```
* Add the members of a collection to the tree view cursor.
* This method can be called recursively.
LPARAMETERS oCollection, cParentKey, cParentType, cAlias, nLevel

IF PCOUNT() < 5
   nLevel = 1
ENDIF

* Make the compiler happy
EXTERNAL ARRAY oCollection

* Now, the members
LOCAL nItem, oObject, cKey, cLabel, lIsClass, cID, cItemType, nPage, ;
      cRefID, lIsCollection, laObjectInfo(1), nToDrillDown

nToDrillDown = 0

FOR nItem = 1 TO oCollection.Count
```

```
cLabel = "Item[" + TRANSFORM(m.nItem) + "]"
cKey = oCollection.GetKey(m.nItem)
IF NOT EMPTY(m.cKey)
   cLabel = m.cLabel +  "/" + m.cKey
ENDIF
cID = m.cParentKey + "/#" + TRANSFORM(m.nItem)

lIsClass = (VARTYPE(oCollection[m.nItem]) = "O")

* Modified 7-June-2010 by TEG
* If it's a class, check whether it's a collection
IF m.lIsClass
   lIsCollection = (PEMSTATUS(oCollection[m.nItem], "BaseClass", 5) AND ;
     UPPER(oCollection[m.nItem].BaseClass) = "COLLECTION")
ENDIF

* If this is an object, check whether it's already in the
* items we've visited. If so, categorize it differently
* and don't drill down.
cRefID = ""
IF m.lIsClass
   cRefID = This.oItemsVisited.GetItemKey(oCollection[m.nItem])
   IF NOT EMPTY(m.cRefID)
      cItemType = "Previous"
      nPage = 4
   ELSE
      * Modified 7-June-2010 by TEG
      * Handle collections correctly

      IF m.lIsCollection
         cItemType = "Collection"
         nPage = 1
      ELSE
         cItemType = "Class"
         nPage = 2
      ENDIF
   ENDIF
ELSE
   cItemType = "Nothing"
   nPage = 3
ENDIF

INSERT INTO (m.cAlias) ;
   (ID, PARENTID, PARENTTYPE, ;
    TYPE, TEXT, IMAGE, ;
    SORTED, PAGE, REFID, ;
    NODEKEY) ;
   VALUES ;
   (m.cID, m.cParentKey, m.cParentType, ;
    "ITEM", m.cLabel, m.cItemType, ;
    .F., m.nPage, m.cRefID, ;
    This.oTreeViewContainer.GetNodeKey("ITEM", m.cID))

* Modified 7-June-2010 by TEG
* There are now two values of cItemType we want to pursue
```

```
   IF m.lIsClass AND m.cItemType <> "Previous"
      This.oItemsVisited.AddItemVisited(oCollection[m.nItem], m.cParentKey, ;
         m.cID, "ITEM")
   ENDIF

   * Modified 17-July-2012 by TEG
   * Try breadth-first search
   nToDrillDown = m.nToDrillDown + 1
   DIMENSION laObjectInfo[m.nToDrillDown, 4]
   laObjectInfo[m.nToDrillDown, 1] = m.lIsClass
   laObjectInfo[m.nToDrillDown, 2] = m.cItemType
   laObjectInfo[m.nToDrillDown, 3] = m.cID
   laObjectInfo[m.nToDrillDown, 4] = m.nItem
ENDFOR

LOCAL nDrillDownItem

FOR nDrillDownItem = 1 TO m.nToDrillDown && oCollection.Count

   m.lIsClass   = laObjectInfo[m.nDrillDownItem, 1]
   m.cItemType  = laObjectInfo[m.nDrillDownItem, 2]
   m.cID        = laObjectInfo[m.nDrillDownItem, 3]
   nItem = laObjectInfo[m.nDrillDownItem, 4]

   * Is this member a collection?
   IF m.cItemType = "Collection"
      This.AddCollectionMembersToTreeViewCursor(oCollection[m.nItem], m.cID, ;
         "ITEM", m.cAlias, m.nLevel + 1)
   ENDIF

   IF m.cItemType = "Class"
      This.FindObjectPropertiesForTreeViewCursor(oCollection[m.nItem], m.cID, ;
         "ITEM", m.cAlias, m.nLevel + 1)
   ENDIF

ENDFOR
```

The other recursive method is FindObjectPropertiesForTreeViewCursor, shown in **Listing 31**. This method loops through the members of an object and adds any that are objects to the cursor. Like AddCollectionMembersToTreeViewCursor, it defers recursive calls until it has added all the object properties.

The main loop goes through all the PEMs of the object. For each property that references an object, it checks whether the referenced object is either a COM object or the main object of GDIPlusX, because we don't want to put either of those into the cursor. If not, it calls another method (AddObjectPropertyToTreeViewCursor) to add the object to the cursor and adds a row to the array aObjectInfo. The new row has three elements: the name of the method we need to call to process this object (either AddCollectionMembersToTreeViewCursor or FindObjectPropertiesForTreeViewCursor), a reference to the object, and the unique ID we've generated for the object.

After that loop finishes, we loop through the array. For each row, we construct the necessary call and then execute it.

It may seem strange that the array and the technique for making the call differ in the two routines. We don't have to store an object reference for collection members because the collection plus the item number in the collection gives us access to the object. For properties of objects, we don't have direct access like that. We could store the property name and then construct an object reference, but that's extra work.

Listing 31. The FindObjectPropertiesForTreeViewCursor method looks at each property of an object and adds those that reference objects to the cursor.

```
LPARAMETERS oObject, cParentKey, cParentType, cAlias, nLevel

LOCAL aProps[1], nPropCount, nProp, oChildObject, cName


IF PEMSTATUS(oObject, "Name", 5)
   cName = oObject.Name
ELSE
   cName = "UNNAMED"
ENDIF

* Modified 17-July-2012 by TEG
* Try breadth-first search
LOCAL aObjectInfo[1], nToDrillDown, nDrillDownItem, cID
nToDrillDown = 0

* Modified 17-July-2012 by TEG
* Pick up controls, too.

nPropCount = AMEMBERS(aProps, m.oObject, 1)
FOR nProp = 1 TO m.nPropCount
   IF (aProps[m.nProp, 2] = "Property" AND ;
      TYPE("oObject." + aProps[m.nProp, 1]) = "O") OR ;
      (aProps[m.nProp, 2] = "Object")
      oChildObject = EVALUATE("oObject." + aProps[m.nProp, 1])
      * Want to omit COM and null objects
      IF NOT ISNULL(m.oChildObject)
         DO CASE
         CASE PEMSTATUS(m.oChildObject, "BaseClass", 5) AND ;
              UPPER(oChildObject.BaseClass) = "COLLECTION"
            cID = This.AddCollectionPropertyToTreeViewCursor(m.oChildObject, ;
               aProps[m.nProp,1], m.cParentKey, m.cParentType, ;
               m.cAlias, m.nLevel + 1)

            * Modified 17-July-2012 by TEG
            * Put this item in the list for follow-up
            IF NOT EMPTY(m.cID)
               nToDrillDown = m.nToDrillDown + 1
               DIMENSION aObjectInfo[m.nToDrillDown, 3]

               aObjectInfo[m.nToDrillDown, 1] = ;
```

```
                  "AddCollectionMembersToTreeViewCursor"
               aObjectInfo[m.nToDrillDown, 2] = m.oChildObject
               aObjectInfo[m.nToDrillDown, 3] = m.cID
            ENDIF

         CASE ALLTRIM(COMCLASSINFO(m.oChildObject, 5)) = "1"
            * Modified 11-November-2012 by TEG
            * System object doesn't seem to have name property. Use class.
            * This is an imperfect test since, in theory, the object could have
            * been subclassed. But that's unlikely in practice
            IF NOT PEMSTATUS(m.oChildObject, "class", 5) OR ;
               NOT UPPER(m.oChildObject.Class) = "XFCSYSTEM"
               * Omit GDIPlusX stuff to avoid trouble
               cID = This.AddObjectPropertyToTreeViewCursor(m.oChildObject, ;
                  aProps[m.nProp,1], m.cParentKey, m.cParentType, ;
                  m.cAlias, m.nLevel + 1)

               * Modified 17-July-2012 by TEG
               * Put this item in the list for follow-up
               IF NOT EMPTY(m.cID)
                  nToDrillDown = m.nToDrillDown + 1
                  DIMENSION aObjectInfo[m.nToDrillDown, 3]

                  aObjectInfo[m.nToDrillDown, 1] = ;
                     "FindObjectPropertiesForTreeViewCursor"
                  aObjectInfo[m.nToDrillDown, 2] = m.oChildObject
                  aObjectInfo[m.nToDrillDown, 3] = m.cID
               ENDIF
            ENDIF
         OTHERWISE
            * COM object. Don't add it.
         ENDCASE
      ENDIF
   ENDIF
ENDFOR

* Modified 17-July-2012 by TEG
* Now drill down
LOCAL cMethod, cCommand
FOR nDrillDownItem = 1 TO m.nToDrillDown
   cMethod = aObjectInfo[m.nDrillDownItem, 1]
   oChildObject = aObjectInfo[m.nDrillDownItem, 2]
   cID = aObjectInfo[m.nDrillDownItem, 3]

   TEXT TO m.cCommand NOSHOW TEXTMERGE
   This.<<m.cMethod>>( m.oChildObject, m.cID, "PROPERTY", m.cAlias, m.nLevel + 1)
   ENDTEXT

   &cCommand
ENDFOR

RETURN
```

Technically, we could probably combine the two recursive methods into a single method, but each is complex enough that the resulting code would be hard to read. Having two methods collaborate makes much more sense.

Getting recursion wrong

In cases where recursion makes sense, there are basically two ways you can mess it up. The first is to fail to include a termination case. In the factorial and Fibonacci examples that open this paper, it's hard to see how you'd miss that since the termination cases are so obvious. But in many of the other examples, the termination case isn't quite as obvious because it's about not making the recursive call unless you find certain conditions.

For example, in the BindClick method used to have objects inside a container pass their clicks to the container (see "Setting up event binding," earlier in this paper), the termination condition is NOT PEMSTATUS(oObject, "Objects", 5). In the Registry examples, the termination condition is the array returned by the EnumKeys method being empty.

Of course, not only do you have to include a termination condition, but your recursive calls have to get you to the termination condition eventually, too. This is why both the factorial and Fibonacci routines check for negative inputs. If either was accidentally passed a negative value and we didn't check, recursion would continue until the program stack or memory was exhausted, because we'd never reach the termination condition (0 for factorial, 1 or 2 for Fibonacci).

The other easy way to mess up your recursive code is by not declaring variables properly. Far more than most VFP code, recursion relies on variables being local to the current call. (That is, you can actually use private variables, as long as they're explicitly declared. That means recursion was, in fact, possible in pre-VFP versions of FoxPro.) For example, the DrillControls method that's part of the Control Renamer (see "Modify controls and code," earlier in this paper) loops through all the controls in the specified container. There's a FOR loop, partially shown in **Listing 32**. If the counter variable nControl isn't declared local, the recursive calls could change it and when we return to the loop from the recursive call below it, we'd have lost our place in the array and might either reprocess items we've already processed (which could result in infinite recursion) or skip some items.

Listing 32. Declaring most variables local in recursive routines is essential.

```
nControls = AMEMBERS(aControls, oContainer, 2)

FOR nControl = 1 TO nControls
   * Figure out what info is available about
   * this control
   oObject = EVALUATE("oContainer." + aControls[nControl])
```

Here's another example. One of the naturally recursive algorithms is Merge Sort, a technique for sorting, generally applied to arrays. (In VFP, of course, we have the ASORT() function, so we don't actually need to write array sorting routines. But this routine provides an easy way to demonstrate the issues of variable scope in recursion.) Merge sort

divides the list to be sorted in two, calls itself to sort each half, and then merges them. Each time you call the recursive routine, you've cut the size it's dealing with in half. The recursive calls end when you get down to one item.

Listing 33 shows a VFP version of Merge Sort (included in the materials for this session as MergeSort.PRG). It receives an array as a parameter. If the array has more than one element, it figures out where the middle of the array is. Then, the left half of the array is copied into one array (aLow) and the right half into another (aHigh). The routine is called twice, passing the two different portions. When the routine returns, we loop through the two smaller arrays, copying items into the original array in the right order.

Listing 33. Merge Sort is a naturally recursive way to sort a list. Here, it's used to sort an array.

```
* Merge sort
* Sort an array by dividing it in half, sorting each half,
* and then merging the two. Recursive.
* For one-dimensional arrays
LPARAMETERS aArray

EXTERNAL ARRAY aArray

LOCAL nLen, nMiddle
LOCAL aLow[1], aHigh[1]

* Divide in half and call recursively.
* Then, merge results
nLen = ALEN(aArray, 1)
IF m.nLen > 1
   * Low end
   nMiddle = INT(m.nLen/2)
   DIMENSION aLow[m.nMiddle]
   ACOPY(aArray, aLow, 1, m.nMiddle)
   MergeSort(@aLow)

   * High end
   DIMENSION aHigh[m.nLen - m.nMiddle]
   ACOPY(aArray, aHigh, m.nMiddle+1, m.nLen-m.nMiddle)
   MergeSort(@aHigh)

   * Now merge
   LOCAL nLow, nHigh, nMain
   nLow = 1
   nHigh = 1
   nMain = 1

   DO WHILE (m.nLow <= m.nMiddle) AND (m.nHigh <= m.nLen-m.nMiddle)
      IF aLow[m.nLow] <= aHigh[m.nHigh]
         aArray[m.nMain] = aLow[m.nLow]
         nLow = m.nLow + 1
      ELSE
         aArray[m.nMain] = aHigh[m.nHigh]
         nHigh = m.nHigh + 1
      ENDIF
```

```
        nMain = m.nMain + 1
    ENDDO

    IF m.nLow <= m.nMiddle
        * Copy remaining items from aLow
        FOR m.nLow = m.nLow TO m.nMiddle
            aArray[m.nMain] = aLow[m.nLow]
            nMain = m.nMain + 1
        ENDFOR

    ELSE
        * Copy remaining items from aHigh
        FOR m.nHigh = m.nHigh TO m.nLen - m.nMiddle
            aArray[m.nMain] = aHigh[m.nHigh]
            nMain = m.nMain + 1
        ENDFOR
    ENDIF
ENDIF

RETURN
```

The materials for this session include TestMergeSort.PRG, which populates an array and then calls MergeSort to sort it.

The materials also include MergeSortBug.PRG, which is identical to MergeSort.PRG, except that the local declaration for aLow and aHigh has been commented out. This means that the two arrays are created as private in the first call to the routine and all calls share those two arrays (rather than each creating their own copies, as happens when they're declared in the routine). When you run this version of the sort (as from TestMergeSortBug.PRG, which is identical to TestMergeSort, except for calling this version and is included in the materials for this session), it fails with "Subscript is out of range" error in the merge portion of the process. That's because the last recursive call redimensions at least one of the arrays to have a single element, but when you return to an earlier call, that earlier call expects to find the array it created.

It's also worth noting that there are situations where using a private variable in a recursive routine is handy. It allows you to have some items that are calculated through all recursive calls. For example, I used a private variable to count the number of times the recursive Fibonacci routine was called. I declared lnCalls private in the calling code, and then incremented it at the top of the Fibonacci function. (A version of that function with the counter is included in the downloads for this session as FibRecurCount.PRG and a corresponding version of the calling routine is included as FibTimeCount.PRG) The code for generating combinations described in the next section of this paper uses the same approach to count how many combinations have been generated.

What else can you do with recursion?

The examples in this paper by no means exhaust the list of cases for recursion. In fact, they don't even exhaust the list of things I've previously written that use recursion.

I showed many years ago how to use recursion to generate combinations of a specified size from a list of items. (That is, given, say, the numbers from 1 to 10, generate all the different sets that contain exactly three of those numbers.) This classic math problem is easier to do with recursion than without, if the set size isn't known at the time you write the code. You'll find my article on that at http://www.tomorrowssolutionsllc.com/Articles/Generating%20combinations.PDF.

Another classic Computer Science algorithm other than Merge Sort can be implemented recursively, as well. Binary search, the technique for finding a particular item in a sorted list, is a naturally recursive algorithm. You start by checking the middle item in the list. If it matches the one you're looking for, you're done. If the middle element is larger than the search item, call the routine recursively with only the first half of the list. If the middle element is smaller than the search item, call the routine recursively with the second half of the list. Repeat until you find the item or run out of list to search. Because you cut the list in half on each call, you'd have to start with an extremely large list to overrun the stack. (For a list of 2^n-1 items, the search takes a maximum of n recursive calls. So even with the default STACKSIZE of 128, you can have a list of more 3.4×10^{38} items; that's way more items than you can store in a VFP table or put in a VFP array, even if you have tons of memory.) Although VFP's ASCAN() function means we don't need to write our own, my recursive version of binary search is included in the materials for this session as BinarySearch.PRG; a routine to randomly populate an array and then search for every number in the specified range is included as TestBinSearch.PRG.

Combinations, binary search, and merge sort are all divide and conquer algorithms. Recursion is great for anything you can describe that way. As we've already seen in this paper, recursion is also good for drilling down through hierarchies. Application development is full of hierarchies, so recursion is a great tool to have in your toolbox.

Appendix: Proof by induction example

I opened this paper by saying that I love proof by induction and that recursion is very similar. Here's a simple example of proof by induction. Prove that the sum of the integers from 1 to n is n(n+1)/2, or to write it algebraically:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

We start with the base case, where n=1. The sum of the integers from 1 to 1 is, of course, 1. The other side of the equation becomes:

$$\frac{1\,(1+1)}{2}$$

which reduces to

$$\frac{1(2)}{2} = \frac{2}{2} = 1$$

For the induction case, we assume the assertion is true for some value of *n*, say *k*. Can we then prove that it's true for $n = k+1$? That is, with the assumption, can we show:

$$\sum_{i=1}^{k+1} i = \frac{(\mathrm{k}+1)(\mathrm{k}+2)}{2}$$

The sum of the integers from 1 to $k+1$ is the sum of the integers from 1 to k plus $k+1$, that is:

$$\sum_{i=1}^{k+1} i = \sum_{i=1}^{k} i + (k+1)$$

We can substitute in the result we're assuming for the sum from 1 to k:

$$\sum_{i=1}^{k+1} i = \frac{k(k+1)}{2} + (k+1)$$

To simplify, we multiply the last term by $2/2$ (that is, 1):

$$\frac{k(k+1)}{2} + (k+1) = \frac{k(k+1)}{2} + \frac{2\,(k+1)}{2}$$

Which is:

$$\frac{k(k+1) + 2(k+1)}{2}$$

And simplifying terms:

$$\frac{k^2 + \mathrm{k} + 2k + 2}{2}$$

Which is:

$$\frac{k^2 + 3\mathrm{k} + 2}{2}$$

Which is the same as:

$$\frac{(k+1)(k+2)}{2}$$

Which is what we set out to prove.

