# Putting Parameters in Perspective

*Doug Hennig*

**Concentrating on communication between routines can reduce up to one-third of the errors in your applications. This month's column examines some "Best Practices" regarding the use of parameters.**

Welcome to "Best Practices", the monthly column that discusses solid programming practices, with emphasis on techniques and ideas specific to FoxPro. Last month we discussed the concept of assertions, which are pieces of code that test assumptions programmers make to ensure they're true and the code doesn't break. This month, we'll discuss parameters. This has a logical tie-in to last month's column since assertions are often used for parameter testing.

Parameters seems like one of those topics that should be a no-brainer, but good system design requires looking at how the different pieces of a system communicate with each other. Studies cited by Steve McConnell in his book *Code Complete* found that 39% of all errors are due to internal interface problems; in other words, modules not communicating properly with each other. Spending time investigating this topic can help reduce more than one-third of the errors you encounter.

The first topic to get out of the way is the use of parameters at all. I can't tell you how many projects I've taken over from other consultants that used no parameters at all. These systems used public or private variables to communicate between programs. You can get away with that in the xBASE languages because a variable is scoped to the routine that created it and all of its subroutines. The problem with this approach is that it tightly couples a subroutine to the program that called it. Tight coupling is normally a bad practice.

Just in case you're fuzzy on this whole "good-bad" thing, let's discuss coupling for a moment. "Coupling" refers to how closely two programs are intertwined. One of the ways to couple two programs is to make one of them use a lot of assumptions about the other, such as that certain variables exist and contain certain values. Routines that are tightly coupled are like Siamese twins: if they always live together, the problems are tough enough, but taking them apart requires very complex surgery.

There are probably a hundred reasons why loose coupling is better than tight coupling; here are a few:

- If you edit one of the routines, you must look at the other routine to ensure it isn't affected. That effectively doubles your maintenance load.
- Tighter coupling reduces portability, since the subroutine makes assumptions about the calling program that might not be true in a different system.
- A subroutine can inadvertently change the value of variables used by the calling routine. Visual FoxPro (VFP) introduced LOCAL variables to help reduce coupling, since you'll get an error message if you reference a variable local to a routine in one of its subroutines.

Now that we have that out of the way, let's examine some "best practices" for parameters.

## LPARAMETERS Vs. PARAMETERS

The PARAMETERS statement accepts parameters passed to a function and implicitly makes them PRIVATE. This means other routines called by the function can see and possibly change the value of these parameters. VFP has a new LPARAMETERS statement that makes the parameters LOCAL, thus hiding them from any routine but the current one. Since there's no downside to using LPARAMETERS instead of PARAMETERS, always use LPARAMETERS. What if a routine called by the function needs access to one of the parameters? Simple: explicitly pass it to the routine. This reduces coupling between the two routines, so it's always a good idea.

## Eliminating Errors Related to Parameters

How many times have you called a little-used FoxPro function only to get an error because the number or type of parameters was incorrect? The error message you receive alerts you to the fact that the parameters are incorrect, which is a better solution than to return invalid results or hang the computer. Your own functions should do the same: test the number, type, and range of acceptable values for parameters. FoxPro has built-in functions to help with this task, including PARAMETERS() and TYPE().

One thing to watch out for: according to the FoxPro Help, "The value returned by PARAMETERS( ) is reset every time a program, procedure, or user-defined function is called or when ON KEY LABEL is executed." This means you must test the value of PARAMETERS() in a function before calling any other subroutine. I frequently save the value returned by PARAMETERS() in a variable such as lnParameters to ensure I can test for the number of parameters passed to this function, not the last function called. You can also use the undocumented PCOUNT() function, which was added in FoxPro 2.6 for dBASE compatibility and works the way you might expect PARAMETERS() should work.

Although it's riskier, you could reduce the overhead of parameter testing in an internal function (such as one called by only one routine or a protected method in a VFP class) by inserting assertion testing code during development and testing of the function, and then removing it from the production version (see last month's "Best Practices" column for ideas on assertion testing). Since such a function will only ever get called from a known routine, you just need to ensure that the calling routine passes the correct parameters to the function.

What should you do when you encounter bad parameters passed to a function? The most obvious action is to display a message using WAIT WINDOW or MESSAGEBOX. You could also trigger an error in VFP using the ERROR command if you have an error handler to log the error and gracefully exit the application. The ERROR command can accept an error number if the error you wish to trigger matches a built-in VFP error (such as 1229, which is "too few arguments") or you can use a custom error message. Instead of triggering an error, the function could return a value indicating an error occurred (such as .F. or -1), could set a global flag such as glERROR_OCCURRED, or could exit the application using CANCEL or calling an application exit routine. The action I use most frequently is to display a message and return a value indicating an error occurred. This puts the onus on the calling routine to test if an error occurred and take appropriate action, but this is generally more flexible than having the function simply exit the application.

## Don't Monkey With Input-Only Parameters

"Input-only" parameters are those whose values are passed into the routine and aren't intended to be changed upon return. Changing the value of input-only parameters can have unexpected effects. Here's an example: the following routine is used to determine if a name is a duplicate or not:

```
parameters tcName
local llFound
tcName = upper(alltrim(tcName))
seek tcName
llFound = found()
return llFound
```

This code is misleading: it gives the impression that tcName's value isn't an input-only parameter when it really should be, since changing the contents of tcName isn't the intention of the routine. In addition, this practice can actually cause the contents of a variable passed to the routine to be changed under certain conditions. Parameters are passed to procedures by reference, which means any changes made to the parameters in the procedure are reflected back in the calling routine. By default, parameters are passed to functions by value, but that's controlled with the SET UDFPARMS command, and you can't predict how that might be set on someone else's machine. If it's set to "REFERENCE", upon return from this routine, the variable passed as the name to check has had its contents trimmed and upper-cased, which probably isn't desirable.

If you have to alter the contents of input-only parameters, copy them to local variables and alter the local variables. Here's the above routine rewritten following this idea:

```
parameters tcName
local lcName, llFound
lcName = upper(alltrim(tcName))
seek lcName
llFound = found()
return llFound
```

### Number of Parameters

In *Code Complete*, Steve McConnell suggests that the number of parameters for a function should be kept to seven or less. Whenever I break this rule, I pay for it. One of the routines in *Stonefield Data Dictionary*, called LOOK_UP, has nine parameters, while another called BROWSER has eight. Although most of these parameters are well-documented both in the header of each routine and in the user manual, guess which functions I get the most support calls about?

There are several ways to reduce the number of parameters needed by a routine. Although it's rarely a good idea, using global variables might be useful under certain circumstances. For example, in a FoxPro 2.x security function, instead of passing the name of the user in a parameter, you could expect that a global variable (such as gcUSER_NAME) contains it. Since using global variables has a number of pitfalls, a big one being increasing the coupling between a function and the routines that call it, consider this option very carefully before using it.

Another possibility is to look up the needed information in a table rather than passing it as a parameter. In the case of the LOOK_UP and BROWSER functions mentioned earlier, I couldn't reduce the number of parameters because of backward compatibility. However, I could eliminate how many needed to be passed when called by new applications by looking up the value to use for unpassed parameters in a table. We'll discuss this idea of unpassed or "optional" parameters more later.

VFP provides other solutions as well. For example, if the function is a method in a class, you can use properties of the class instead of passing parameters. Another useful technique is to pass an object to a function. Properties of the object are then available to the function without explicitly passing them as parameters. This can also help solve another problem, that of how to return more than one value from a function or how to return any value at all from a form instantiated from a class.

Let's say we have a class based on the Form base class. The form displays a list of tables to reindex (passed into the class in an array), a checkbox to indicate if the tables should also be packed, and OK and Cancel buttons so the user can decide whether to do any reindexing or not. We want the form to be released when the user chooses OK or Cancel and to return three sets of values: an array containing those tables chosen by the user, a flag set to .T. if the tables should also be packed, and a flag set to .T. if OK was chosen. The problem: the Show() method used to display the form doesn't return anything but .T., and once the form has been released, there's no way to examine any of its properties.

To solve this, create a Parameter class:

```
define class Parameter as custom
  dimension aParameters[1], aArray[1]
enddefine
```

The sole purpose of this class is to act as a container of values passed into or out of a function. The aParameters property will contain one row for each individual value needed, while aArray is used if an array must be passed.

Before calling a function, instantiate an object from this class and initialize the aParameters and aArray properties appropriately. Then pass the object to the function. The function can update the aParameters and aArray properties of the object before returning, and the calling routine can examine these properties upon return from the function.

In the following example, we need two single-valued parameters (one which is .T. if the tables should be packed and one indicating if OK was chosen) so the aParameters property of the Parameter object is dimensioned to 2. The SelectTablesForm class, which displays the form when its Show() method is called, accepts the Parameter object in its Init() method.

```
oParms = createobject('Parameter')
dimension oParms.aParameters[2]
oForm = createobject('SelectTablesForm', oParms)
oForm.Show()
```

When the user clicks on the OK button, the form puts .T. in oParms.aParameters[1] if the pack checkbox was checked, .T. in oParms.aParameters[2], and copies the names of selected tables into the oParms.aArray property. If the user clicks on Cancel, it puts .F. in oParms.aParameters[2]. Upon return, the calling routine can examine the properties of oParms to determine what actions to take.

The source code disk contains these classes and a program called REINDEX.PRG that calls them.

The downside to passing an object to a function is that both the function and the calling routine must have knowledge of the properties of the object and have agreed upon the structure of the returned values. This increases the coupling between the function and the calling program.

### Optional Parameters

The simplest function to call is one that needs no parameters. However, as I mentioned earlier, it's not a good idea to minimize the number of parameters by increasing the coupling of the function. Instead, look for opportunities to use optional parameters. An "optional" parameter is one that can be passed or not. If the parameter is passed, the function uses it; if not, the function uses a default value instead or performs its operation differently. An example of a built-in function that accepts optional parameters is SEEK(). The first parameter, which is required, is the value to look for. The second parameter, the table to look in, is optional: if it isn't passed, the table in the current work area is used. The VFP version of SEEK() supports a third optional parameter, the tag to use. If it isn't specified, SEEK() uses the current order for the table.

The advantage of using optional parameters is that they make calling a function easier (fewer parameters to type or even remember) most of the time yet provide additional functionality when required.

There are a couple of ways you can detect if an optional parameter was passed or not. You can use PARAMETERS() to determine how many parameters were passed, and assume if the number is smaller than expected that one or more optional parameters were omitted. My preferred approach, however, is to test the type of optional parameters; unpassed parameters are logical with a value of .F., so TYPE() returns "L" for unpassed parameters. If that parameter should be logical even if it's passed, no harm is done.

Here's an example of a routine that handles optional parameters. It's a wrapper program for the VFP MESSAGEBOX() function, one that handles Yes or No questions only and returns .T. if the user chose Yes or .F. if they chose No (this program is called YN.PRG on the source code disk). The first parameter, which is required, is the message to display. The second, which is optional, is the title for the message box. If it isn't passed, the title of the main VFP window is used. The third parameter, which is also optional, is the default value for the function. If .T. is passed, the Yes button is the default. If .F. is passed or it isn't passed at all (we don't care which), the No button is the default.

```
function YN
lparameters tcMessage, tcTitle, tlDefault
```

```
#include '\VFP\FOXPRO.H'
local lcTitle, llDefault
lcTitle = iif(type('tcTitle') = 'L', ;
  _screen.Caption, tcTitle)
return messagebox(tcMessage, ;
  MB_YESNO + MB_ICONQUESTION + ;
  iif(tlDefault, MB_DEFBUTTON1, MB_DEFBUTTON2), ;
  lcTitle) = IDYES
```

Here are several examples of calling this function (this code is in TESTYN.PRG on the source code disk):

```
YN('Delete this record?')
YN('A serious error occurred. Terminate the ' + ;
  'application?', 'Application Error', .T.)
YN('Exiting system. Save current record?',, .T.)
```

Notice the use of a double comma in the last example. In FoxPro 2.x, you couldn't leave out a parameter in the middle of a list of parameters; you had to provide a "dummy" parameter (such as a blank string or .F.) in its place. In VFP, you can now simply omit the unpassed parameter by entering nothing between its comma and the one following.

Although YN handled an unpassed tcTitle by assigning a hard-coded default value to it (the screen's caption), if the function is a method in a class, a better mechanism is to have a property of the class contain the default to use instead. For example, assume YN is a method in a general message handling class. The class has a property called cTitle that's used as the default title for all message boxes if a title isn't passed to a method. In this case, the line that assigns lcTitle would be:

```
lcTitle = iif(type('tcTitle') = 'L', ;
  This.cTitle, tcTitle)
```

The source code disk includes a class called MessageMgr with a YN method. TESTYN.PRG shows the same examples calling this method as it did calling YN.PRG.

The advantage of this approach is flexibility: the developer can specify the title to use once and then not have to pass the title parameter unless it must be different for a particular message box.

### Order of Parameters

While the order in which parameters are passed may not seem important, it can make a difference in how easy it is to use a function. For example, I was tired of constantly using ASCAN() followed by ASUBSCRIPT() to find the row for a value in a multi-dimensional array. Since laziness, not necessity, is the motherhood of invention, I wrote a function called ArrayScan() that, when passed an array and a value, returns the row where the value is found in the array. Although the order of the parameters didn't matter, I opted to pass the array first and value second because that's the order of parameters for ASCAN(). Now I don't have to remember two different parameter schemes for similar functions; both ASCAN() and ArrayScan() accept the same parameters, but ArrayScan() works a bit differently.

To minimize the use of "dummy" parameters or double commas when calling a function that supports optional parameters, order the parameters in the order of likelihood of use, with mandatory ones first. For example, if you expect to specify the default button more often than the title for the message box, a better choice for the order of the parameters in the YN function examined earlier would be:

```
lparameters tcMessage, tlDefault, tcTitle
```

### Conclusion

Hopefully, this article has given you some ideas to consider on a topic that you might have thought is straight-forward. Properly concentrating on communication between the routines in your applications can go a long way to eliminating errors.

Speaking of errors, next month we're going to examine error handling in VFP. This can be quite a complex subject, so we'll spend a couple of columns investigating the issues, and look at a general error handling class.

*Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Sask., Canada. He is the author of Stonefield's add-on tools for FoxPro developers, including Stonefield Data Dictionary for FoxPro 2.x and Stonefield Database Toolkit for Visual FoxPro. He is also the author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's "The Pros Talk Visual FoxPro" series. Doug has spoken at user groups and regional conferences all over North America. CompuServe 75156,2326.*