Using code blocks, again

Data Based Advisor
November 01, 1991 | Spence, Rick

One of the most difficult concepts to grasp in **Clipper** 5.01 is the application of code blocks. In this article, using one of Nantucket's sample programs as the key example, I'll show you how you can use code blocks to reduce the size of programs. Let's start, however, with a review of the **basics**.

One more time...

Code blocks are a new data type. They're stored in variables, just like other data types such as numerics and dates. You can pass code blocks as parameters, return them from functions, and copy them into other variables.

One of the reasons some programmers have problems understanding code blocks is the type of data they contain. Programmers are familiar with variables storing data such as names and addresses, telephone numbers, and account balances.

Code blocks, however, store pieces of program code. These pieces of code can be passed to subroutines, such as functions or procedures, and eventually that piece of code can be run. Code blocks are created using a rather bizarre syntax:

```
bVar := { || test ()}
```

Read the "{" character as meaning "start code block" and the "}" character as "end code block." The two "|" characters delimit the code block's formal parameter list. (I won't cover that this month.)

The bVar variable can be local, static, private, or public. It contains a piece of code that, when evaluated, calls the function test().

To evaluate, or run a code block, use the new `eval()` function:

```
result := eval( bVar )
```

This calls test(). Like all functions, `eval()` returns a result. It returns the result of the expression contained inside the code block. That's all there is to code blocks.

Code blocks for C programmers

For those who know the C language, **Clipper**'s code blocks resemble C's pointers to functions. Pointers to functions are a C data type, and they contain exactly what their name implies--a pointer to a function. The pointer contains the machine address of the function. When you "dereference," or point to, the pointer, the function is called. This is very similar to **Clipper**'s `eval()` function.

Compiled expressions

I've shown a code block that contained a function call, but what exactly can you store in a variable whose type is a code block? Code blocks can also contain compiled expressions, but what exactly is an expression?

Let's define an expression as a number of constants, variables, and functions calls, separated by operators, that's evaluate to one value. Perhaps it's easier to define an expression in X-Base terms. You could say an expression is anything you can put after the ? command:

```
? x ? 7 * 5 ? 3 - y ? test () / test1() * 4
```

In this example, `"x"`, `"7 * 5"`, `"3 - y"`, and `"test()/test1() * 4"` are all expressions. You can't write:

```
? SEEK "Spence"
```

Therefore `SEEK` isn't an expression. Following this logic, you can't put a `SEEK` command inside a code block. See how the analogy works? Note that `SEEK` is a, and commands aren't expressions.

Another one of **Clipper**'s syntactic entities is a statement. Statements aren't expressions, therefore you can't put them inside code blocks. Statements are things like `IF`, `DO WHILE`, `DO CASE`, and assignment, such as:

```
x = x + 1
```

This means you can't put an IF statement inside a code block, nor can you use the "=" assignment operator--or can you? For example, figure out what the following prints:

```
x = 1
b = { | |x = x + 1}
? eval(b)
? x
```

The first "?" prints ".F.", the second "?" prints "1". The confusing thing about the "=" operator is that it's overloaded. This means it does not of two things depending on how you use it. If you use it in the statement:

```
x = x + 1
```

it means "assign the sum of x and 1 to the variable x." It therefore performs an assignment. If you use the "=" operator in an expression:

```
x = 7
```

it means "compare two things for equality" (the comparison operator). The "=" is an assignment operator in a statement, and a comparison operator in an expression. When you use it inside a code block (which contains expressions), it means perform a comparison. Thus the code block:

```
{ || x = x + 1 }
```

when evaluated, means "compare x to x + 1," which returns false.

IF is a statement, therefore you can't place it inside a code block. However, there is a way to test from inside a code block. You use a conditional expression, iif, as in:

```
b = { || iif(x = 1, a(), b() )}
```

This means "if x is 1, call the function a; otherwise call the function b."

Using code blocks

In my article, "Code Block **Basics**" (Data Based Advisor Feb. 1991) I showed how to use code blocks to replace macros and parameterize logic. The first way is efficient; the second reduces the amount of code. Let's look at the second way.

Consider writing a routine to search an array. The built-in ascan() function does the basic job, but it compares the array elements with the search value according to the state of `SET EXACT`. Maybe you want to search an array ignoring case. You would write a function, such as ascani():

```
 1 FUNCTION ascani( ar, searchFor )
 2
 3 LOCAL i
 4
 5 DO WHILE i <= len(ar) .AND. ;
 6      upper( ar[i] ) != upper( searchFor )
 7
 8       i = i + 1
 9
10 ENDDO
11
12 RETURN iif(i > len(ar), 0, i)
```

Like the `ascan()` function, `ascani()` returns the element number where the search value is found, a zero if it isn't.

Assume you need a function to search an array, without regard for case, but now you want an exact comparison. (The previous example was a comparison according to the state of `SET EXACT`.) The routine, ascanie(), might look like:

```
 1 FUNCTION ascanie( ar, searchFor )
 2
 3 LOCAL i
 4
 5 DO WHILE i <= len(ar) .AND. ;
 6          !(upper(ar[i]) == upper(searchFor))
 7
 8     i = i + 1
 9
10 ENDDO
11
12 RETURN iif(i > len(ar), 0, i)
```

Compare the two routines for a moment. The only difference between `ascani()` and `ascanie()` is line 6. Here's line 6 in the first routine:

```
6 upper(ar[i]0 != upper(searchFor)
```

and line 6 in the second:

```
6 !(upper(ar[i]) == upper(searchFor))
```

The only difference is the way you compare the search value with the array elements. The point is, if you need to search an array 15 different ways, you'd write 15 different routines, each one different by just one line.
Take one more look. The only difference between these routines is the one line that actually does the comparison. It's wasteful to repeat the other code since it doesn't change. What you need to do is write one routine to search arrays, but pass a parameter indicating how to do the comparison. To do that you need to pass a piece of code to do the comparison, and what data type do you think you should use? A code block. The code block will do the comparison, and return a .T. if the element being compared is what you're looking for, .F. if it isn't. The code block will accept a parameter--the array element being compared. You can thus write the one `ascan()` function, as:

```
 1 FUNCTION ascan( ar, bComp )
 2
 3 LOCAL i 4
 5 DO WHILE i <= len(ar) .AND. ;
 6          !eval(bComp)
 7
 8    i = i + 1
 9
10 ENDDO
11
12 RETURN iif(i > len(ar), 0, i)
```

To perform the case-insensitive search, call ascan() with:

```
ascan(ar, { | el | upper( el ) = "SPENCE"} )
```

To do the exact, case-insensitive search, call it with:

```
ascan(ar, { | el | upper( el ) == "SPENCE"} )
```

Now you have one routine, with 15 different ways of calling it. Quite a saving! You may be aware that **Clipper** 5's built-in `ascan()` function lets you to pass exactly this sort of code block as the second parameter (it works just like I've shown).

A comprehensive example

For the comprehensive example, I've chosen one of Nantucket's source files, `LOCKS.PRG`, which you'll find on the release disk. LOCKS.PRG contains some routines to help you try file and record locks, to append blanks, and to open files on a network. The philosophy is, if at first your lock doesn't succeed, try, try, again! Actually, you don't try forever; you usually try for so long before reporting an error. Two functions, RecLock() and FilLock(), attempt record and file locks for the time specified by the caller. Their return result indicates their success or failure. Listing 1 shows these two routines.

These two routines are identical except that RecLock() attempts an rlock() and FilLock() attempts an flock().

```
Listing 1--Nantucket's RecLock and FilLock from LOCKS.PRG

/***
* RecLock( ) --> lSuccess
* Attempt to RLOCK() with optional retry */
FUNCTION RecLock( nSeconds )
   LOCAL lForever
   IF RLOCL()
      RETURN (.T.) // Locked
   ENDIF
   lForever = (nSeconds = 0)
   DO WHILE (lForever .OR. nSeconds > 0)
      IF RLOCK()
         RETURN (.T.) // Locked
      ENDIF
      INKEY(.5) // Wait 1/2 second
      nSeconds = nSeconds - .5
   ENDDO
RETURN (.F.) // Not locked
```

```
/*** * FilLock( ) --> lSuccess
* Attempt to FLOCK() with optional retry */
FUNCTION FilLock( nSeconds )
   LOCAL lForever
   IF FLOCK()
      RETURN (.T.) // Locked
   ENDIF
   lForever = (nSeconds = 0)
   DO WHILE ( lForever .OR. nSeconds > 0)
      IF FLOCK()
         RETURN (.T.) // Locked
      ENDIF
      INKEY(.5) // Wait 1/2 second
```

You can, of course, replace these with one routine, say NetTry(), and pass a code block to try either a record lock or a file lock (see Listing 2). You can call it with:

```
IF !NetTry( 5, { || rlock() } ) // Couldn't get record lock within 5 seconds
```

and:

```
IF !NetTry( 3, { || flock() } ) // Couldn't get file lock within 3 seconds
```

Without changing what NetTry() does, you can actually simplify it by writing the loop correctly. The version in Listing 2 evaluates the block outside the loop, and, if that succeeds, immediately exits. Otherwise it enters the loop which continually re-evaluates the block. Listing 3 shows a simpler version. It only contains one `RETURN` statement! (Note: you can perform the same simplification to the original RecLock and FilLock if you don't care to use NetTry().)

Listing 2--Replacing RecLock and FilLock

```
/***
* NetTry( , ) --> lSuccess
* Attempt bAction with optional retry */
FUNCTION NetTry( nSeconds , bAction)
   LOCAL lForever
   IF eval( bAction)
      RETURN (.T.) // Locked
   ENDIF
   lForever = ( nSeconds = 0 )
   DO WHILE ( lForever .OR. nSeconds > 0 )
      IF eval(baction)
         RETURN (.T.) // Locked
      ENDIF
      INKEY(.5) // Waite /2 second
      nSeconds = nSeconds - .5
   ENDDO
RETURN (.F.) // Not locked
```

So far, you've replaced RecLock() and FilLock() with one routine about half its size. You can take this one step further.

Listing 4 shows another routine, AddRec(), from LOCKS.PRG. Addrec() tries to append a record to a database using the `APPEND BLANK` command. If the command fails, it loops the specified number of seconds, just like the RecLock() and FilLock() functions.

Listing 3--Simplifying NetTry()

```
FUNCTION netTry( nSeconds, bAction )
   LOCAL lForever, lSuccess
   lForever := ( nSeconds == NIL .OR. nSeconds == 0)
   DO WHILE !( lSuccess := eval(bTry)) .AND. ;
             ( lForever .OR. nSeconds > 0)
      INKEY(.5) // Wait 1/2 second
      nSeconds = nSeconds - .5
   ENDDO
RETURN lSuccess
```

The code is once again very similar to RecLock() and FilLock(), thus a candiate for replacement with NetTry(). This time, there are a couple of complications. The first is that a failed `APPEND BLANK` is detected by checking the result of `neterr()` after the attempting the command.

In the previous case, you called functions (rlock() and flock()) which returned their success or failure. The second problem is that APPEND BLANK is a command. Since commands aren't expressions, they can't be placed directly inside code blocks.

You can overcome both problems, however. **Clipper** 5.01 introduced functional equivalents of all the built-in database processing commands. Dbseek(), for example, is the functional equivalent of the SEEK command. (In fact the standard header file, STD.CH, simply converts the SEEK command into a call to dbseek().) You can also call these functions directly, just like the functions you're used to calling, such as substr() and ctod().

Listing 4--AddRec from LOCKS.PRG

```
/***
* AddRec( WaitSeconds ) --> lSuccess
* Attempt to APPEND BLANK with optional retry */

FUNCTION AddRec( nWaitSeconds )
   LOCAL lForever
   APPEND BLANK
   IF .NOT. NETERR()
      RETURN (.T.)
   ENDIF
   lForever = (nWaitSeconds = 0)
   DO WHILE ( lForever .OR. nWaitSeconds > 0 )
      APPEND BLANK
      IF .NOT. NETERR()
          RETURN .T.
      ENDIF INKEY(.5) // Wait 1/2 second
      nWaitSeconds = nWaitSeconds - .5
    ENDDO
RETURN (.F.) // Not locked
```

The functional equivalent of APPEND BLANK is dbappend(), so you can call dbappend() in the code block you pass to NetTry(). Now, the block you pass to NetTry returns a .T. if the operation succeeds, otherwise a .F. You can check that by checking neterr(), so you replace Nantucket's AddRec() with a call to NetTry() with:

```
IF !NetTry( 4, { || dbAppend(), !neterr() }) // Couldn't APPEND record within 4 seconds
```

The final function to replace is Nantucket's NetUse(). It tries to open a file in the requested mode, then returns a logical indicating success or failure. An error is indicated by neterr() returning a .T.. I'll leave the call to NetTry() as an exercise. (Hint: the functional equivalent of USE is dbUse().

Summary

In this article, I show some practical ways to use code blocks. I showed how to use them to parameterize the logic of a routine. I used the example of searching arrays many different ways with one function, and another of replacing four almost identical Nantucket routines with one using four different calls.

A former member of the Nantucket development team, Rick Spence owns Software Design Consultants, a training and consulting company. He's also the author of **Clipper** Programming Guide, 2nd Edition, published by Data Based Solutions, Inc. and Slawson Communications, and technical editor of Compass For **Clipper**, a monthly journal from Island Publishing. You can contact Rick through Software Design Consultants at (818) 892-3398, on MCI (LSPENCE), or on CompuServe (71760,632).