Try these new 5.01 language constructs.

(extensions to the database programming language in Nantucket Corp's Clipper 5.01 database development system)

Data Based Advisor
June 01, 1992 | Gutierrez, Dan D.

Clipper 5.01 introduced basic language constructs that can make programming more efficient and effective:

```
* TYPE() and VALTYPE()
* ":=" assignment operator
* Increment (++) and decrement (--) operators
* "==" or "exactly equal" operator
* Code blocks and extended expressions as replacements for macros
* // and /* */ for commenting
```

To put **Clipper** 5.01 through its paces, you have to understand basic language constructs, such as data types, operators, extended expressions, alias functions, and revised methods of commenting source code. In this article, we look at each of these issues to lay a framework for more comprehensive programming.

Are you my type?

Clipper 5.01 offers all of the standard xBase data types plus a few new ones. For database fields, only the standard xBase character, numeric, date, logical, and memo types are available. However, a memory variable (memvar) can hold those, plus array, code block, object, and `NIL` data.

A memvar's type is determined by "dynamic typing," the data type of the value it's first assigned. This is considered a characteristic of a weakly-typed language, since the type can be changed dynamically by simply assigning it a value of a different type. While language purists have often pointed to this as a dangerous shortcoming in xBase, xBase developers have viewed dynamic typing as a programming plus, since it lets you change types at will. Regardless of which camp you're in, the use of Hungarian notation along with a strict adherence to maintaining the initial type will help you avoid confusion. (Remember, Hungarian notation aids type consistency by prefixing a single lower-case character designator such as "c" for character, which permanently ties the memvar to a specific data type.)

The following declaration statement assigns a value of each available type to a memory variable:

```
LOCAL cZIP      := '90024-4559',;
      nSalary   := 45675,;
      dHireDate := CTOD ('10/27/60'),;
      lFullTime :=.T.,;
      aTaxRate  := {8.25, 7.75, 6.5},;
      bMax      := {|| a },;
      maxwidth  := MAX( maxwidth, a) },;
      oBrowser  := TBrowseNEW (10, 10, 20, 70),;
      vTemp
```

This example demonstrates the Hungarian naming conventions, how to construct constant values for each data type, and how to use the in-line assignment operator, **:=**, which we'll learn more about later. Notice that the last memvar defined, vTemp, has a prefix of "v," which indicates that this memvar can have a variable data type during the execution of the program. In other words, it's expected to undergo dynamic retyping.

It can be convenient to determine the type of a memvar during execution. For example, a procedure or user-defined function (UDF) might require the type of a parameter passed to it for proper processing. For this purpose, **Clipper** has two built-in functions, `TYPE()` and `VALTYPE()`.

The `VALTYPE()` function is generally considered more convenient since `TYPE()` uses macro expansion to do its work--though both functions return a one- or two-character code indicating the parameter type.

The `TYPE()` function requires a single-character expression parameter enclosed in quotes. For example, `TYPE(` "aMonth") returns an "A" indicating an array type, whereas `TYPE(` "aMonth[10]") might return a "C" since the 10th element of the array holds a character string.

The following code returns the types of the memvars assigned above to produce the output shown next to each line:

```
?? VALTYPE( cZIP ), VALTYPE( nSalary )        // C N
?? VALTYPE( dHireDate ), VALTYPE( lFullTime ) // D L
?? VALTYPE( aTaxRate ), VALTYPE( bMax )       // A B
?? VALTYPE( oBrowser ), VALTYPE( vTemp )      // O U
```

The type returned for vTemp is U, because **Clipper** assigns the value `NIL` to all declared but uninitialized variables; so a type check of a variable whose value is `NIL` is "**U**" or undefined.

The assignment operator

As you'll notice from my code example above, the new **:=** assignment operator is used to place a value in a memvar or database field. Contrast this with the use of **=** in xBase. Depending on context, sometimes **=** handles assignment, and sometimes it's a test for equality. In **Clipper** 5.01, the **=** can still be used for assignment, but it's discouraged in favor of **:=**. There are other reasons for using **:=**, namely because it can be more liberally placed in a program. Consider these examples:

```
nSalary := 25000 // Typical assignment statement
```

Here, we see that **:=** can be used as a substitute for **=** when assigning values to a memvar. There's no advantage to this beyond consistency of using **:=** for assignment purposes.

```
IF ( dDate := (date()-365)) = CTOD('10/27/60')
```

In the IF statement, the logical expression has two components. The right side shows a date constant, the left side has an in-line assignment. Here, the value (date()-365) is used for the comparison, but it'll also be assigned to the memvar dDate. (Of course, the same could be done in other languages, but two lines are needed. **Clipper** 5, as will become evident, has many such shortcuts.)

```
? sqrt(nValue := (nValue**2))
```

In this example, the built-in sqrt() function is called, but its argument contains an in-line assignment. The numeric expression nValue**2 is evaluated, passed to sqrt() and assigned to nValue.

```
cCustNo := cSortSeq := custfile->custno+dtoc(date())
```

Here's a multiple assignment. (In xBase this is done with a `STORE` command.)

```
Custfile->custno := txn->txnno := cCustNo+dtoc(date())
```

Finally, **:=** can be used instead of the `REPLACE` command to assign values to database fields.

Other operators

Clipper 5 has many other operators to make programming more streamlined. First, we'll look at the increment "**++**" and decrement "**--**" operators. In **Clipper** we no longer need to write:

```
nValue = nValue+1
```

or:

```
nValue = nValue-1
```

since the increment and decrement operators serve this purpose:

```
nValue++
nValue–
```

There are actually two forms to these operators. The illustrations above show the post-increment and post-decrement form. This means that the increment is done **after** the original value is used in an expression. This contrasts with the pre-increment and pre-decrement form in which the operation is carried out **before** the value is used. Let's go through some examples:

```
LOCAL nValue := 0, nNewValue, nValue1, nValue2
*
* Prefix-increment operator
nValue := 0
nNewValue := ++nValue // Change nValue
                      // BEFORE assignment
? nValue // 1
? nNewValue // 1
*
* Postfix-decrement operator
nValue := 1
nNewValue := nValue-- // Change nValue
                      // AFTER assignment
? nValue // 0
? nNewValue // 1
*
* Postfix-increment operator in an expression
*
* (Nantucket invented this word, rather than
*
* simply using "suffix.")
nValue := 10
? nValue++ * nValue // 110: increment; use 10;
```

```
                    // multiply by 11
? nValue // 11
*
* Prefix-decrement operator in an expression
nValue := 10
? --nValue * nValue // 81: decrement, use 9;
                    // multiply by 9
? nValue // 9
*
* Combined prefix and postfix operators
nValue := 10
? --nValue * nValue++ // 81: decrement, use 9;
                      // multiply by 9; increment
? nValue // 10
*
* Combined prefix and postfix operators
nValue1 := 10
nValue2 := 10
? --nValue1 * nValue2++ // 90: decrement, use 9;
                        // multiply by 10; increment
? nValue1 // 9
? nValue2 // 11
```

The comments associated with each use detail how the expression is evaluated. If the answers don't make sense, try to remember the difference between when a memvar is used in an expression vs. when it's updated with an increment or decrement operator.

There's also the following shorthand notation:

```
nValue += 365 // Equivalent to nValue:=nValue+365
             // also: -= *= /= %= ^=
```

which makes your code more concise.

Relational operators

As I've mentioned, the = operator is overloaded, so **Clipper** 5 offers another operator to combat this situation. The "exactly equal" (==) operator is used in logical expressions to test equivalence. Although it's commonly used with any data type, **==** has special significance for character string comparisons. The basic difference between **=** and **==** lies in the way strings of unequal length are handled. Moreover, **=** is affected by the current setting of `SET EXACT`, whereas **==** is not. The examples below illustrate how **=** and **==** differ:

```
SET EXACT OFF
? '123' = ''          // Always .T. if right string is null.
? '' = '123'          // Always .F. if right is  longer.
? '123' = '123456'    // Same as above.
? '123456' = '123'    // Do comparison for each right string character.
                      // If all equal, then .T.
```

With `SET EXACT ON`, all of the above logical expressions evaluate to **.F.**, since strings with unequal lengths are automatically unequal. In addition, comparisons with **=** and `SET EXACT OFF` begin by removing all trailing spaces (not physically, just for the comparison). Under these conditions the following expression is evaluated as **.T.**:

```
SET EXACT ON ? '123' = '123' // .T. since equal in length and characters.
```

With the **==** operator, the problem is much simpler. Consider the following:

```
? 'Fredi' == 'Fred' // .F. since unequal in length.
? 'DDG' == 'DDG'    // .T. the only time equal is possible.
```

The setting of `SET EXACT` doesn't influence the outcome; in fact, it's a good idea to leave `EXACT OFF` for the duration of the program.

One more thing to note is the "**!=**" operator. For comparison, dBase III PLUS requires the **<>** or **#** operators when testing for inequality.

Now, with **Clipper** 5, we can use the more recognizable exclamation point with the equal sign. **!** also makes its way into logical functions such as eof() where testing for "not end-of-file" becomes !eof().

The fall of macros

Most xBase people, including developers moving from **Clipper** Summer '87, have been educated to incorporate the "**&**" macro operator. Indeed, in those dialects, **&** was a necessary evil that most people convinced themselves was both necessary and good. **Clipper** 5 has taken the opposite position. Their reasoning: Programs that use the macro operator heavily yield poorer performance than those that don't.

You can now avoid the **&** in **Clipper** 5. One area is solved by extended expressions, the other by code blocks.

Extended expressions

A useful **Clipper** feature is extended expressions. These represent a path away from macro use:

dbf_name = 'orders' **&&** Variable DBF name
&& becomes part of USE
USE **&**dbf_name

can now be replaced with:

USE (dbf_name) // A clean alternative with no **&** thus eliminating a macro expansion.

Another common application of the macro is in variable database field names:

code_field = "CUST_NO" **&&** Variable field name
&& becomes part of
&& the REPLACE statement

```
REPLACE &code_field WITH mTemp
```

This can be replaced (notice the change in coding style too) in **Clipper** 5 with:

```
LOCAL cField : = 'SUPP_NO', bField
USE supplier NEW
bField : = fieldblock(cField) // Create retrieval / assignment
                                      // code block for SUPP_NO
eval (bField, 'DDG001')
```

In this example, the built-in fieldblock() function builds a code block (a topic deserving its own discussion) which, when evaluated, either gets or sets a database field's value. The example above sets the field. If just the field's value is to be retrieved, EVAL(bField) would suffice. In this example, you can see that macros are no longer needed for variable field name situations.

Alias functions

Clipper offers a big improvement over the dBASE III PLUS standard by enabling database-oriented functions to operate on a non-selected work area. The example below shows how this works:

```
? customer-> (eof()) // Test end-of-file in CUSTOMER work area
? movie-> (recno())  // Get current record number in MOVIE
? txn-> (bof())      // Test beginning-of-file in TXN work area
? txn-> (deleted())  // Return deleted flag of current record
                     // in TXN work area
? txn-> (fcount())   // Get number of fields in TXN work area
? txn-> (lastrec())  // Return last record number in TXN
? txn-> (reccount()) // Display number of records in TXN
? txn-> (recsize())  // Record size in TXN work area
```

In dBase III PLUS, you have to use a `SELECT` statement to bring the desired work area into view and then perform the function.

With **Clipper** 5, you can even define a `SEEK()` UDF and use it as:

```
txn-> (seek (movie->itemno))
```

Extended expressions are extremely flexible, but you may prefer to have `ALIAS` clauses available for all related commands as in **Clipper**'s `SKIP` command:

```
SKIP 1 ALIAS customer
```

SKIP is the only **Clipper** 5 command with an ALIAS clause, but by using a feature of **Clipper**'s preprocessor (user-defined commands or UDCs) you can have:

```
SEEK movie->itemno ALIAS txn
```

even though this isn't part of the **Clipper** 5 language definition. (s warrant future discussion.)

Comments, comments, everywhere

Comments should be everywhere in an application program for all the obvious reasons. In fact, you may have noticed an unfamiliar comment syntax introduced throughout the examples of this article. With **Clipper** 5, commenting is made rather attractive by allowing for the more "C-like" notation using "**//**" and "**/* */**". The **//** comment syntax operates in the same manner and replaces dBase's **&&.** **/**/** lets you bracket off a group of characters considered comments. The most common usage of these types of comments are in surrounding major comment blocks such as those preceding subprogram code.

```
/*
DispMsg ( cMsg, nTone, nDuration ) - - NIL
This function displays a message cMsg, sounds a tone described by nTone,
for a duration of nDuration.
*/
```

The above example could have been written with an * as the first non-blank character of each line, but it's more easily done with **/* */**.

That's a wrap

Now that we've looked at basic ways to become productive with **Clipper** 5, it's time to investigate one of **Clipper**'s most dynamic features, the preprocessor, our subject for next time.