Create the truly reusable routine (design of Clipper Winter '84 and Clipper 5.0)

Data Based Advisor
March 01, 1991 | Straley, Stephen J.; Tamburrino, Jim

Create the Truly Reusable Routine

Let's define a few terms and ideas. To start with, turn to the beginning of Clipper. In most of our X-Base experience, we've been programming procedures and procedure files. This came down to us from the DO command in dBASE II. Whenever we want something to happen, we write code in a PRG file, then DO it. Everything we write "DOes" something, and when an operation is complete, it "RETURNs" program flow back to the calling routine.

What made Clipper stand above the rest six years ago was a twist in this approach. What Brian Russell (builder of Clipper Winter '84) did to the X-Base approach was to give a value to these operations that DO things.

In other words, everything still DOes something, except in some cases the operation will end with a value that's returned to the calling task. At that point in history, the concept of a user-defined function (UDF) was born. (What's amazing is that no one got worked up over a user-defined procedure but everyone was excited over a UDF.) A UDF is nothing more than a procedural set of operations with a return value. That value becomes the value of the function and is stored, evaluated, displayed, or used in other programming expressions.

To add to this twist, Brian did two more things. First, he treated procedures and functions the same and gave a default return value to a procedure. In every version prior to Clipper 5.0, the value of a procedure has been a logical false (.F.). These values have never been used because whenever we call a procedure, it's in a DO command. For example, consider these two programming statements:

```
DO Eof
x = EOF ()
```

In the first statement, the Eof subroutine is called. We don't know anything about it other than that whatever operation is inside of it is needed. We can say the same thing about the second statement with one addition: the value returned from the EOF() subroutine is stored to the variable X. The first statement may perform the same way, except that the value is simply ignored, since there's no assignment statement associated with its call. In this lies the key: if all procedures return a default value that's ignored, then procedures and UDFs are identical. This explains why a procedure and a function with the same name can't exist together--they'll generate a duplicate symbol compilation error.

From this came Brian's second concept: He allowed the call to a procedure to be made without a DO command. Since a procedure or a function really is a set of operations with a return value, (a default of NIL in Clipper 5.0 and logical false; in prior versions or UDFs) they can be considered an expression with a formal name. This may sound confusing at first, but stick with us as we explore the intent of the language. The reason we mention this now is to introduce the notion that an expression may be evaluated anywhere in a command line. For example, consider these two expressions:

```
IF (1 = 1)
IF EOF()
```

In the second example, EOF() is a logical value. In the first example, the value of the expression is determined by the use of parentheses and has the value of the comparison of 1 against 1. Both have a logical data type as a value but only the EOF() function has a formal name.

Expressions are the key to understanding Clipper 5.0. An expression may be a procedure, function, or series of procedures, functions, or operations. The value of the expression may be used in a programming statement or the value of the expression may be ignored, like a procedure's value in a DO command. This approach allows us to use the language in a variety of problem-solving environments and a myriad of styles and techniques.

In last month's article, we referred to the STD.CH (Standard Clipper Header file) to help us understand Clipper 5.0. Now we should look at the compiler switch,/P, which allows the preprocessor output (PPO) file. Whenever we write a piece of Clipper code that has its origins in dBASE, we can simply use this switch to dump the preprocessor output to file on disk to help us learn how Clipper now works. For example, consider this programming extract:

```
CLEAR SCREEN
DO WHILE !EOF ()
    ? RECNO()
    SKIP
ENDDO
```

When Clipper is called like this:

```
Clipper filename /p
```

the output file will be the same root name as the filename with a PPO file extension. This file name can be assigned when a formal file name follows the /p switch without a space (/pDump.$$$). Looking at this file, we see the previous Clipper lines translated into the following:

```
Clear()
while !EOF()
    QOut( RECNO() )
    dbSkip(1)
ENDDO
```

Don't be too concerned with some of the functions that are called. You can use this file to get a better understanding of how Clipper treats the X-Base language and how it translates your code, using the preprocessor, into the internal calls it needs to make an EXE file. For example, the ? command is nothing more than a call to the QOUT() function, which doesn't need the DO command to work it simply stans alone on a command line. What we've known as a command in the past is really nothing more than a function or series of functions known as an expression.

Clipper 5.0 advantages

Each feature in Clipper 5.0 serves a specific purpose, but collectively they make up the complete Clipper picture:

* *Pre-processor*--The translator that matches patterns and text and converts them to internal Clipper functions and statements.

* *New data types*--Formal recognition of code blocks, arrays, and NIL are added to numerics, dates, strings, and logical data variables.

* *Objects*--A series of pre-defined structures that can be programmed to do many new types of database operations.

* *Linker*--A more powerful linker with various new options to handle different programming situations using features such as pre-linked libraries, incremental linking, and dynamic overlays.

* *Data storage classes*--Two new methods in which Clipper will treat the existence and visibility of a variable.

* *Parameter passing*--Along with private parameters, LOCAL parameters have been added, as well as the ability to pass parameters by reference and/or value. Up to 128 parameters can be passed to a subroutine (procedure or function).

* *Database manipulation*--Character fields can be up to 64K -1 bytes in size 1,024 fields can exist in a database. You can have multiple relations with 15 indexes per database and eight child relations per parent.

* *Programming lenghts*--While 250 bytes is the limit of the character length for an index expression, there's virtually no limit on the number of characters processed by the compiler on one line of code.

* Array support--Clipper now supports "multi- dimensional" arrays. In addition, many old array functions have been beefed up and new ones introduced.

* *Clipper switches*--Additional compiling swithes have added more power to the programming language, including /N (no implied start-up routine), /W (enable warnings), /P (generate PPO file), /D (define manifest constant), and /V (automatic variable declaration).

* *Menu systems*--Clipper has always offered the programmer the ability to create greater "video-puffery" for the client. The @..PROMPT/MENU TO commands are one example.

* *Data-entry system*--The entire data-entry system has been opened up to allow us the ability to customize how a basic GET works. The potential uses range from resizing the cursor in insert or overstrike mode, to adding error sounds following invalid keystrokes, or creating GETs that go from right to left (rather than left to right)

* *Error system*--As in the past, the error system along with the new error object is programmable. In addition, new functions such as ALERT() give programmrs direct control of video, printer, and disk drive if an error occurs.

How we now work

To get a better feel of how each of these components really works and how we can apply the concept of programming data and not subroutines, let's look at how we work today.

In most programming environments, we write a routine--either a function or a procedure. Within minutes, we'll want to use that generic routine in several places in the application. To do this, we need to either accept more parameters into the subroutine or rewrite the routine to be more generic. As a result, we re-program the same routine over and over again.

The thrust with Clipper 5.0 is to write specific routines to tell generic routines what to do. In essence, the specific functions rewrite the generic functions for us. This means that we'll be programming for change, and we'll be more modular and granular. With dynamic overlays in Clipper's new linker (RTLink) and the Virtual Memory Manager (VMM), which allows up to 64M of the virtual memory, the possibilities are awesome. This may sound confusing, but once you've grappled with and conquered it, you'll see the advantages. To start, let's look at system design and development.

System design

Let's assume you've selected Clipper on a DOS platform to satisfy your client's needs. The first step in designing an application in Clipper is writing out what is known as the "require definition." Some call this a "systems requirement specifications," and others call it a "spec sheet."

Whatever you call it, the point is to write down what the system needs. The requirements study should be a clear statement of the project's environment and objectives. Databases can't be defined without knowing what needs to be reported menus, colors, and keystrokes can't be selected without due consideration of the needs of the users. Both topics lend themselves to solution quite easily with Clipper 5.0. However, without a good plan of attack, the most powerful program in the most powerful language will crumble. End users must always be considered.

When developing a system in a group environment, concentrate on programming standards and styles. This includes, perhaps, a data dictionary of generic and corporate functions and subroutines, standard header and banner styles, default colors and keystrokes escape routines, and standard header files.

Programming in the beginning

Try to program in as much pseudo-code as possible. Clipper is no longer limited to 256 characters to a command line. In addition, new notational possibilities have been added to the language, and you should take advantage of them. For instance, use them at the top of every routine to spell out the purpose of the code including the definition of the parameters. This is unnecessary if the function's name or parameter names are clear however, in the beginning, especially for a large group-developed application, these conventions should be the rule rather than the exception. Here are some of the new notational methods:

* **NOTE** :--This is a standard X-Base command for a complete line of notes.

* ***** --This is another standard X-Base command for a complete line of notes.

* **&&**--This allows notes to be given in-line with the program, for example:

```
DO WHILE !EOF() && Scanning the clients work area
```

* **//** --This is the same as the && notational command and is preferred over the && symbol since they may not be supported in future versions. An example of this is:

```
DO WHILE !EOF() // Scanning the clients work area
```

Double forward slashes, as well as &&, may be used on a command line by themselves. For example:

```
// Scanning the clients work area DO WHILE !EOF()
```

* **/* ... */**--This is a new way to make notations. Once a note is established with the /* symbols, all characters thereafter will be considered part of the note until the terminating characters */ are found. An example of this is:

```
/* This will scan the data base in the clients work area until the end of file marker is encountered */ DO WHILE !EOF()
```

We prefer this method for in-line notes since it parallels the C language and allows for a more free-form style of writing. The drawbacks are that if the end of note characters are not used (*/), then code will become part of a continuing note. In addition, you have to be sure not to over-use this form of notation and write enough notes to fill a novel.

Next time, we'll look at database design relational models, indexes, filters, and aliases in Clipper 5.0.

Stephen J. Straley, formerly at Nantucket, is the author of Programming with Clipper 5.0 published by Bantam, and From The Desk of Steve Straley, published by Four Seasons Publishing Co., Inc. Jim Tamburrino, formerly with Bell-South Services, is President of LMT Computer Services, Inc. Steve and Jim conduct the Steve Straley Clipper 5.0 Seminars, produced by Four Seasons Publishing Co., and may be contacted at (212) 599-2141 or on CompuServe (74105,756).