Go Above and Beyond ACHOICE with TBrowse

By John Armstrong

Using TBrowse to search arrays offers several advantages over ACHOICE(). John Armstrong demonstrates how to use TBrowse, ASCAN(), and RefreshAll() to perform complicated

ACHOICE() is a nifty function, but it's a closed system and much of its behavior is out of reach. For example, have you ever wanted to search an array based on the first two or three characters of each element rather than just the first character? Or how about something as simple as putting a leading blank in front of each element and still having the ability to search based on the first character of the element? To tackle these kinds of problems, you need an array-browser with more flexibility than ACHOICE(). While you could build such a beast from scratch using SCROLL(), TBrowse provides a much easier solution.

TBrowsing an array

TBrowsing an array is similar to TBrowsing a .DBF file and is fairly straightforward. (See the sidebar for how TBrowse retrieves data and displays it on the screen.) If the user's keystroke is a cursor control key, then the appropriate export method—UP(), DOWN(), and so forth—is invoked. The code associated with all but two of these methods is built into Clipper and is inaccessible.

The two exceptions are the BOTTOM() and TOP() methods. The behavior of these two methods is defined in the code blocks GoBottomBlock and GoTopBlock. When you're browsing an array in TBChoice, *go top* means go to the first element of the array and *go bottom* means go to the last element of the array.

Complicated browses

It gets more complicated, however, when users want to move around in the data using a method of their own, such as "seeking" an element based on an alpha key. It's simple to trap the user's keystroke based on a range of CHR() values, and it's also simple to perform an ASCAN() of the array and find the matching array element.

TBrowse Step-by-Step

Certain TBrowse functions work together closely to retrieve and display data on the screen. Here's how it works.

The process begins when the user invokes a method that moves the data pointer, making the TBrowse window unstable. To restabilize, work from the outer edge of the new dataset back toward the active record. If the invoked method moves the data forward, the "outer edge" is the record that will be displayed at the bottom of the about-to-be-displayed window. If the direction is backward through the data, the outer edge is the top line of the about-to-be-displayed window.

Stabilize() and SkipBlock() work together to identify the record that lies on this outer edge. Stabilize() first makes a theoretical calculation of where this record is and passes that number to SkipBlock(). SkipBlock() then checks the actual data and, if necessary, modifies the Stabilize calculation.

For example, assume a window is sized to display six records, and the currently highlighted record is on the fourth line of the window. When the user hits the PgDn key, the PgDn() method is invoked. It takes just a quick calculation to determine that the last line of the window about to be displayed is eight "counts" away from the current record (two to get to the bottom of the current window, and then six more to move forward a full screen of data).

So Stabilize() passes eight as a parameter to SkipBlock(). Logic in SkipBlock()calculates how many counts forward the data will actually allow, and returns the result of that calculation. (Add the count passed as a parameter to where the record pointer is and return that number (unless it's higher than the length of the array). The system uses this returned value to move the data pointer to the record to be displayed at the bottom of the new window.

At this point Stabilize() clears the window. If any data currently on the screen can be recycled, Stabilize() will write it back to the correct location on the screen. The rest of the data is unknown, however, and has to be retrieved. Stabilize() retrieves the data one record at a time. A call to SkipBlock moves the data pointer to the next record, and the code blocks for each column defined by TBColumnNew() get evaluated and return the actual data.

When Stabilize() makes calls to SkipBlock(), it passes a parameter, which is used by SkipBlock() to move the data pointer. The first time Stabilize() calls SkipBlock(), the parameter is 0. (Remember, the data pointer was moved after the previous call to Skipblock()). So TBColumnNew({||...}) is evaluated for the current record. Stabilize() moves backward (or forward) from there toward the current record by passing a parameter of -1 (or 1) to SkipBlock(). When it has retrieved and displayed all the new data it needs, it highlights the current record. Voila! We have a stable window.

Assuming a match is found, the index of the array element that needs to be made current is available (the index number is assigned to the variable "nPlace" in your sample code). But it's still completely outside the TBrowse system. You need to get it into TBrowse and then refresh the screen accordingly: Highlight the new element and display its neighboring elements above and below it on the screen. These tasks are the work of the code block assigned to the SkipBlock() method, which in the sample code is the user-defined function called MoveIt().

This process is kicked off by calling Refreshall(), which sends a message to the TBrowse object telling it that the system is unstable and the window needs to be redrawn. When RefreshAll() executes, it invokes one critical pass through SkipBlock(), which moves the TBrowse pointer, tracks the current active cell to its new position. When the standard TBrowse methods (PgUp(), PgDn, and so forth) are invoked, TBrowse takes care of this automatically, but you have to provide the code. The code has to go into the MoveIt() function because SkipBlock() is the method RefreshAll() invokes. But the code is different from the code that is used to redraw the array in the window, so use an IF statement to segregate the two sets of code with a flag to determine which part of the IF statement to execute.

When RefreshAll() is invoked, the index of the current array element is stored in nDex, and the index of the element you want to go to is stored in nPlace. Both variables get passed to MoveIt(), which calculates the number of places to move and returns that number. nPlace is also used as the flag to determine which part of MoveIt() to execute. Now that you have finished this part of the refresh, you'll want to set nPlace to zero. Returning it from the function won't work because SkipBlock() expects and can only use the number of places to move. However, by passing it into MoveIt() by reference, when you set its value to zero inside the function it's also set to zero in the calling code. You'll also want to set the value of the current index to its new value by using the same technique.

From here on it's clear sailing. The other half of the logic in MoveIt() gets called as many times as necessary to stabilize the system. It works just as if one of the internal TBrowse methods had been called. ▲

Please see README on the Companion Disk for information about the accompanying files.

John Armstrong is a programmer/analyst at Forum Capital Markets, an Old Greenwich, Connecticut, investment securities firm that uses Clipper to access its Sybase SQL SERVER. CompuServe 75170,373.