Laying the foundation

(using Clipper 5.0's arrays and code blocks)

Part -1 : Arrays

Data Based Advisor
April 01, 1991 de Lisle, Phil

In last month's article, I used **Clipper** concepts to describe some of the technical jargon surrounding object-oriented programming or OOP. This month I'll examine the new features of **Clipper** 5.0 that you'll use to create the user-defined objects (UDOs) I cover later in this series. To that end, this article will only deal with the arrays, code bocks, and objects we'll need later. I'll also offer some undocumented tips you may find useful.

Arrays in 5.0

Arrays in the new **Clipper** are radically different from their counterparts in Summer '87. The most important difference is that they're now a true data type. This means they can be `RETURN`ed from a user-defined function (UDF).

Unlike arrays in other programming languages, a 5.0 **array** is simply a reference to a location in memory where the data is stored. This has an important ramification--you can now assign the same **array** to two different variables. In the code below, we create two local variables, aArray and aNew, which are assigned NIL by **Clipper** automatically. We then assign an **array** with two elements to aArray. When we assign aArray to aNew, both varibles contain a reference to the same area of memory.

```
local aArray, aNew
aArray := {"Example", "Array"}
aNew := aArray
```

However when two or more arrays reference the same area of memory in this way, any change made to an element in one **array** will be reflected in the others. So, for example, if aArray[2] is changed to "**ARRAY**", aNew[2] also contains "**ARRAY**"

Here's a quick tip: If you need a copy of an **array** (so changes to one aren't reflected in the other), use the `ACLONE()` function.

Summer '87 arrays are always passed to UDFs by reference--we aren't required to supply the reference symbol "@" in front of the **array** parameter as we are with other variable types.

In 5.0, though, arrays aren't passed by reference. However, this change has no effect on our code because an **array** is just a reference to an area memory. Any changes made to the **array** within the UDF are still "visible" from the calling procedure.

Suppose we want our UDF to assign a totally different **array** to our **array** parameter. In Summer '87, this was messy and involved laboriously copying each element of the new **array** to the **array** parameter. In 5.0 we just pass the **array** parameter by reference, prefixing it with "@". Within the UDF, we assign the new **array** to the **array** parameter, so that it now holds a reference to a different area of memory. When our UDF returns, the **array** variable retains this new reference.

Multi-dimensional arrays--do they exist?

5.0 does not have multi-dimensional arrays! As in Summer '87, they have a single dimension. However, in 5.0, it's easy to create multi-dimensional arrays.

How does this work? Well, we now know that arrays are a true data type, and from Summer '87, we also know that **Clipper**'s arrays can store any data type to any element and that each **array** element can store a different data type. Therefore, we can store the reference to another **array** to an **array** element. We call this a "nested" **array**.

We can use this nested approach to create multi-dimensional arrays. However, a word of caution: traditional programming languages require each element within the same column of a multi-dimensional **array** to contain the same data type. **Clipper** imposes no such restriction.

The life of an array

An **array** only "exists" as long as there's at least one active reference to it. Consider the following code fragment, which creates a local variable, assigns an **array** to it, then displays it before reassigning it a numeric:

```
local aArray := {"Array ", "Lives"}
? aArray[1], aArray[2]
aArray := 25
```

The moment we reassign 25 to aArray, the reference to {"**Array**", "Lives"} is destroyed and the memory occupied by the **array** is returned to the pool--there's no longer a reference to it.

To prevent an element of a nested **array** from being "de-referenced," assign it to another variable before reassigning the **array** variable, as in the code below:

```
local aArray, aSave
aArray := {12, {"Kee", "this"}, .f.}
aSave := aArray[2]
aArray := NIL
```

"But arrays look like..."

Many **Clipper** developers with C or Pascal/Modula2 experience comment that **Clipper**'s new arrays look suspiciously like linked lists of structures/records, as indeed they do. With the new preprocessor, you can use an **array** to create a structure and a linked list where the **array** element number acts as the pointer into the list. See `DIRECTORY.CH` supplied on the distribution disks and the documentation for `DIRECTORY()` for ideas on how to do this.

Conclusion

This month we've looked at some of the aspects of 5.0's new features, including arrays and code blocks, which are the foundation for UDOs. We've also examined 5.0's objects, which we'll need to understand before we can create templates for our UDOs.

Next time I'll cover the alias technique in detail. This is a pure OOP technique that works with both Summer '87 and 5.0.

> Despite his French name, Phil de Lisle is English. He is the CEO of Lamaura Development and has spoken at many user groups and developers conferences in the U.S. and around the world on Clipper and OOP. You can contact Phil through Data Based Advisor or on CompuServe (100016,1254).

http://www.accessmylibrary.com/coms2/summary_0286-9230360_ITM