⇐ DEVELOPMENT-INDEX

FORTH is not C

getting used with data stack

(c) 2017 Andreas Klimas (w3group, German)

At http://prog21.dadgum.com/33.html recently I found a nice essay **"Understanding What It's Like to Program in Forth"**, from James Hague. Because I'm continously on my way to become a better FORTH programmer I read it. As an exercise I think I had to refactor it and I think this example is programming non-FORTH in FORTH, because of the complex interface. Three arguments on data stack cries for refactoring and a different kind of usage.

The task is: Write a Forth word to add together two integer vectors (a.k.a. arrays) of three elements each.

One has to know that the usage of data stack in FORTH is very different than the Stack in C (or other languages). The data stack in FORTH is shallow and fast. Like processor registers. Too many arguments on data stack is the source of troubles. The return stack should be kept in CPU as well and hence is also limited, but extraordinarily fast.

If we need more items to manage locally, we have to introduce other techniques. An own stack comes to mind. There is no problem with it, because instead of writing *>r ... r>* we can say (for example) *+local ... -local* and then we can maintain a stack of any size we need. With this technique you can open a new *stack* and it must not be private for only one word. So you can open a new stack at prompt and easy debug your words. This is not possible if the return stack is used.

As a small rule of thumb for using the data stack in FORTH:
Use only one or two stack manipulation words in a definition (dup, swap, over, drop).
The data stack is the communication path between words and has to be designed properly. Stack gymnastic is a sign of bad design.

Green zero or one argument. It is best that all words are written this way

Orange two arguments. Sometimes there is no other way, try to avoid it if possible.

Red three arguments. Use it very carefully. Try to reduce it to a number of two. It is a sign that troubles might be near.

```
The C example
void vadd(int *v1, int *v2, int *v3)
{
       v3[0] = v1[0] + v2[0];
       v3[1] = v1[1] + v2[1];
       v3[2] = v1[2] + v2[2];
}
```

```
The FORTH solution from James
: 1st ;
: 2nd cell+ ;
: 3rd 2 cells + ;
: vadd ( v1 v2 v3 -- )
       >r
          over 1st @  over 1st @  + r@ 1st !
          over 2nd @  over 2nd @  + r@ 2nd !
          over 3rd @  over 3rd @  + r@ 3rd !
       rdrop drop drop ;
```

I will show a **different approach** with two streaming address register. One on data stack and one on return stack. First, we define a vector constructor. We are not using structs.

```
: vector ( --) create [ 3 cells ] literal allot ;
```

We create some vectors

```
vector v1
vector v2
vector v3
```

next we are going to define an initializing word

```
: 0vector ( addr--) [ 3 cells ] literal 0 fill ;

v1 0vector
v2 0vector
v3 0vector
```

if your implementation doesn't provide these words ...

```
: @r postpone r@ postpone @ ; immediate
: !r postpone r@ postpone ! ; immediate
: @r+ postpone r> postpone dup postpone cell+ postpone >r postpone @ ; immediate
: !r+ postpone r> postpone dup postpone cell+ postpone >r postpone ! ; immediate
: rdrop postpone r> postpone drop ; immediate
```

next we are going to define some debugging tools

```
: .vector ( v-addr--) @+ . @+ . @ . ;
```

copy tools as well

```
: vcopy ( src dst--) [ 3 cells ] literal move ;
: vector! ( z y x dst--) >r !r+ !r+ r> ! ;
: vector@ ( src--) [ 3 cells ] literal move ;
```

and the addition

```
: vector+ ( src dst-- ; src+dst=>dst)
 >r ( src ; destination address to R register)
 \ now we are in streaming mode with src pointer on data stack
 \ and destination pointer on return stack (as register R)
 @+ @r + !r+       ( v1.x + v2.x => v3.x)
 @+ @r + !r+       ( v1.y + v2.y => v3.y)
 @  @r + !r  rdrop ( v1.z + v2.z => v3.z)
;
```

Now we are testing our new vector addition tool

```
ok> 4703 4702 4701 v1 vector!
ok> 7000 6000 5000 v2 vector!
ok> v1 .vector
4701 4702 4703
ok> v2 .vector
5000 6000 7000
ok> v1 v2 vector+
ok> v2 .vector
9701 10702 11703
ok> v1 .vector
4701 4702 4703
```

finally implementing the requirement

```
: vector++ ( v1 v2 v3-- ; v1+v2=>v3)
 dup >r vcopy r> ( v1 v3 ; v3 initialized with v2)
 vector+
;

ok> 4703 4702 4701 v1 vector!
```

```
ok> 7000 6000 5000 v2 vector!
ok> v1 v2 v3 vector++
ok> v3 .vector
9701 10702 11703
```

Conclusion
I want to comment James conclusion. On his page he wrote:
"And that's it--three element vector addition in Forth. One solution at least; I can think of several completely different approaches, and I don't claim that this is the most concise of them. It has some interesting properties, not the least of which is that there aren't any named variables. On the other hand, all of this puzzling, all this revision...(1)to solve a problem which takes no thought at all in most languages. And while the (2)C version can be switched from integers to floating point values just by changing the parameter types, that change would require completely rewriting the Forth code, because there's a separate floating point stack."

(1) I don't think that there is any difficulty for an experienced FORTH programmer, as it is for an experienced C programmer.
(2) Yes, agreed. The FORTH way is minimalistic. It is unlikely that a FORTH application will need both integer and float vector arithemtic. And if so, well, it might took some keystrokes more.
(3) I wouldn't implement vector++, it seems not useful to me. vector+ should be enough.

DONE