

Scientific Data Structures

Contents

§1 Typed data structures	92
§§1 Type descriptors	94
§§2 Typed scalars	94
§§3 Defining several scalars at once	95
§§4 Generic access	97
§§5 The intelligent floating point stack (ifstack)	99
§§6 Unary and binary generic operators	100
§2 Arrays of typed data	104
§§1 Improved (FORTRAN-like) array notation	105
§§2 Large matrices	106
§§3 Using high memory	108
§§4 A general typed-array definition	110
§§5 2ARRAY and }}	113
§3 Tuning for speed	114

Data structures are the soul of any computer program in any language. Some languages, most notably FORTRAN and BASIC, predefine some data structures but require extensive contortions to define others. This straitjacket approach has virtues as well as defects:

- The pre-defined structures are what most users need to solve standard problems, so meet 80-90% of the cases in practice. That is, they are not terribly restrictive.
- Because the most-needed structures are predefined and have a standard format, they do not have to be invented each time a program is written. Standardization facilitates the exchange and portability of programs.
- Standardized data structures aid program development in discrete modules, permitting sections written by different persons or teams to interface properly with minimal tuning.

FORTH pre-defines a minimal set of data structures but allows unlimited definition of new structures. How is this different from

Pascal, Ada or even C? FORTH not only permits extension of the set of data structures, it permits definition of new operators on them. Thus, e.g., FORTH permits simple implementation of complex arithmetic whereas the aforementioned do not.

This chapter¹ suggests protocols for arrays and typed data² that will increase the portability of code and encourage the exchange of scientific programs. The keys to this are generic operations that recognize the data type of a scalar or array variable at run-time and act appropriately.

§1 Typed data structures

One of the virtues of FORTRAN or BASIC is that the programmer does not have to keep track of what type of data he is fetching and storing from memory. In fact, the user does not even program such operations explicitly — the compiler takes care of everything including the bookkeeping. Mixed-arithmetic expressions like

$$Z = -37.2E-17 * CEXP(CMPLX(R**2, W)/32) / DSIN(W)$$

place great demands on a compiler. The compiler first tabulates the types of the variables and literals in the expression, and then decide which run-time routines to insert. With two types of integers and four types of floating-point numbers (REAL*4, REAL*8, COMPLEX*8 and COMPLEX*16) a typical binary operator such as exponentiation (**) offers 36 possibilities. No wonder FORTRAN compilers are slow.

FORTH sacrifices automation, opting for a small, fast, flexible compiler. The traditional FORTH style gives each type of data its own operators. However, if a program demands all the standard REAL*4, REAL*8, COMPLEX*8 and COMPLEX*16 data types (not to mention INTEGER*2 and *4), having to remember them all and use them appropriately is a chore. This problem has

-
1. Much of the material in this chapter has appeared previously in J.V. Noble, *J. FORTH Ap. and Res.* 6 (1990) 47.
 2. Most languages classify data by type: in FORTRAN, e.g., we have INTEGER, INTEGER*4, REAL, REAL*8, COMPLEX and COMPLEX*16, requiring 2, 4, 4, 8, 8 and 16 bytes of memory, respectively.

led me to experiment with generic access operators, **G@** and **G!**. These let FORTH keep track of which words to use in fetching and storing the "scientific" data types to the fstack (which may partly reside on a co-processor like the 80x87 or MC68881 chips). Corresponding generic unary and binary floating point operators **GDUP**, **G***, etc. allow programs themselves to be generic.

I have lately further modified the scheme to permit more complete automation. The kernel of the method is an "intelligent" fstack, or ifstack, that records the type of each number on it. The generic arithmetic operators and library functions decide from the information on the ifstack how to treat their operands.

An ifstack-based protocol for floating point and complex arithmetic has drawbacks and advantages. A major drawback is the run-time overhead in maintaining the ifstack, and in choosing the appropriate operator for a given situation. In other words we trade convenience for a non-negligible execution speed penalty. To some extent this can be mitigated by *computing* decisions and by vectoring rather than branching (i.e. no Eaker **CASE** statements or **IF ... ELSE ... THENs**). Moreover, although the definitions are coded in high-level FORTH for portability, the key words should be hand-assembled for the target machine. Finally, my high-level ifstack manager has plenty of error checking that could be dispensed with when speed is an issue.

The chief advantages of the ifstack are:

- Unlike FORTRAN, this scheme permits generic routines that will accept several types of input. Hence, e.g., a matrix inversion routine will happily invert **REAL*4**, **REAL*8**, **COMPLEX** and **DCOMPLEX** matrices.
- A FORTRAN → FORTH translator³ becomes simple with generic operators.
- The ifstack permits recursive programming *a la* LISP.

3. See J.V. Noble, *J. FORTH Ap. and Res.* 6 (1990) 131. See also Chapter 11, where we describe a simple FORmula TRANslator.

§§1 Type descriptors


To decide at run-time which @ or ! to use for a particular datum, FORTH needs to know what type of datum it is. The scheme described here wastes a little memory by attaching to each variable a label that tells G@ and G! how to get hold of it.

Here is how we label types:

\ Data type identifiers	
0 CONSTANT REAL*4	\ 4 bytes long
1 CONSTANT REAL*8	\ 8 bytes long
2 CONSTANT COMPLEX	\ 8 bytes long
3 CONSTANT DCOMPLEX	\ 16 bytes long

```
\ a simple version of #BYTES
CREATE #bytes 4 C, 8 C, 8 C, 16 C,
: #BYTES ( type - #bytes) #bytes + C@ ;
```

§§2 Typed scalars

We want the machine to remember for us the data-specific fetches and stores to the co-processor. To accomplish this, the typed variable has to place its address and type on the stack. Thus we need a data structure that we might visualize diagrammatically in Fig. 5-1 below (a cell  represents 2 bytes):

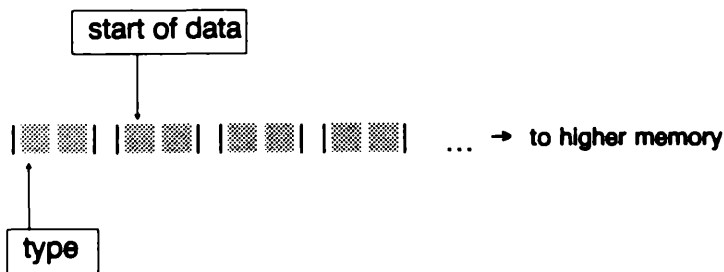


Fig. 5-1 Memory structure of a typed scalar

We implement a scalar through the defining word

```

: SCALAR ( type - - )
  CREATE DUP , #BYTES ALLOT
  DOES> DUP@ SWAP 2+ SWAP ; ( - - adr t)

```

The word **SCALAR** is used as

```

REAL*4 SCALAR X
REAL*8 SCALAR XX
COMPLEX SCALAR Z
DCOMPLEX SCALAR ZZ
... etc. ...

```

Defining several scalars at once

One aspect of the FORTH method of handling variables, that seems strange to programmers familiar with Pascal, BASIC or FORTRAN, is that **VARIABLE**, **CONSTANT** or a new defining word like **SCALAR** need to be repeated for each one defined, as above. That is, such defining words generally do not accept name-lists.

This idiosyncrasy can be traced to FORTH's abhorrence of variables:

- Easily read (and maintained) FORTH code consists of short definitions with few (generally ≤ 4) numbers on the stack. Such programs have small use for variables, especially since the top of the return stack can serve as a local variable.
- In FORTH as in BASIC, variables tend to be global and hence corruptible. The variables in a large program can have unmnemonic names or names that do not express their meaning simply because we run out of names.
- Experienced FORTH programmers tend to reserve named variables for such special purposes as vectoring execution.
- The standard FORTH kernel therefore discourages named variables by making them as tedious as possible.

Most objections to variables can be resolved by making them local. Local variables are relatively easy to define in FORTH: a

straightforward but cumbersome method for making “headerless” words is given in Kelly’s and Spies’s book⁴.

HS/FORTH⁵ provides beheading in a particularly simple form: **BEHEAD’ NAME**, or **BEHEAD” NAME1 NAME2**.

Used after **NAME** has been invoked in the words that need to reference it, **BEHEAD’** removes **NAME**’s dictionary entry leaving pointers and code fields intact and recovering the unused dictionary space. The more powerful word **BEHEAD”** does the same for the range of dictionary entries **NAME1 ... NAME2**, inclusive.

Beheading variable or constant names makes them local to the definitions that use them; they cannot be further accessed – or corrupted – by later definitions. (Pountain⁶ has given yet another method for making variables local, using a syntax derived from “object-oriented” languages such as SMALLTALK.)

Variables are essential for scientific programming. Since we must often have more than two variables, it is silly to repeat **SCALAR**. A simple way to allow **SCALAR** to use a list is

```
: SCALARS ( n - - )
  SWAP 0 DO DUP SCALAR LOOP DROP ;
\ Examples:
\ 2 REAL*4 SCALARS A B
\ 5 COMPLEX SCALARS XA XB XC XD XE
```

I find the use of **SCALARS** with modifiers and lists more convenient and readable than many repetitions of **SCALAR**. Its resemblance to FORTRAN (thereby helping me live with my FORTRAN-inspired habits) is pure coincidence. Although possible to use a terminator (”, e.g.) rather than a count (to define the variable list) I feel it is desirable for the programmer to know

4. M.G. Kelly and N. Spies, *Forth, a Text and Reference* (Prentice-Hall, New Jersey, 1986), p. 324 ff.
5. ©Harvard Softworks, P.O. Box 69, Springboro, Ohio 45066 Tel: (513) 748-0390.
6. Dick Pountain, “Object-oriented FORTH”, *Byte Magazine*, 8/86; *Object-oriented FORTH* (Academic Press, Inc., Orlando, 1987).

how many variable names he has supplied, hence the counted version.

§§4 Generic access

A major theme of FORTH is to replace decisions by calculation whenever possible⁷. This philosophy usually pays dividends in execution speed and brevity of code.

But there is an even more important reason to avoid **IF ... THEN** decisions, especially when working with modern microprocessors. CPUs like 80x86 and MC680x0 achieve their speed in part by pre-fetching instructions and storing them in a queue in high speed on-chip cache memory. A conditional-branch machine instruction (the crux of **IF ... THEN**) empties the queue whenever the branch is taken. Branches should be avoided because they slow execution far more than one might expect based on their clock-counts alone.

To replace decisions, we use the standard FORTH technique of the execution array (analogous to the familiar assembly language jump table). This lets us compute from the type descriptor which fetch or store to use.

We now define **G@** and **G!** as execution arrays using⁸ an execution-array–defining word **G**:

```
: G: CREATE ]
    DOES> OVER + + @ EXECUTE ; (t -)
G: G@ R32@ R64@ X@ DX@ ;
G: G! R32! R64! X! DX! ;
```

assembled from components of the FORTH compiler. That is, the ordinary colon **:** might have the high-level definition (shorn of error detection)

-
7. Leo Brodie, *Thinking FORTH* (Prentice-Hall, Inc., Englewood Cliffs, N.J., 1984), p. 118ff. See also J.V. Noble, "Avoid Decisions", *Computers in Physics* 5,4 (1991) 386.
 8. HS/FORTH uses a word pair **CASE: ... ;CASE** that performs the same task as **G: ... ;** below. **G:** was inspired by Michael Ham (*Dr. Dobbs's Journal*, October 1986).

```

: : CREATE ] DOES> @ EXECUTE ;

```

CREATE makes the new dictionary entry, and **]** switches to compile mode. **DOES>** specifies the run-time action (recall any word created by **CREATE** leaves its parameter field address –pfa– on the stack at run-time, *prior* to the actions following **DOES>**). In the case of **:** the run-time action is to fetch the pfa of the new word and execute it. At run-time, words defined using **G:** add twice the type descriptor to the pfa (to get the offset into the array) then fetch the desired address and **EXECUTE** it.

Microprocessors like the MC680x0 and 80386 that can address large, level memories require no further elaboration for **G@** and **G!**. However, if large arrays are to be addressed within the segmented memory addressing protocol of the 8086/80286 chips, we would have to define **G@** and **G!** to use the “far” forms of addressing words⁹. For example, in HS/FORTH such words as **R32@L** expect a segment paragraph number and offset (32 bits total) as the complete address of the variable being fetched to the 87stack. In that case we modify the definition of **SCALAR** to include the segment paragraph number (seg) in the definition (**LISTS** is nonstandard – it is HS/FORTH’s name for the portion of the dictionary containing the word headers)

```

: SCALAR ( type - )
  CREATE DUP ,           \ make header , type
  #BYTES ALLOT           \ reserve space
  DOES> >R
  [ LISTS @ ] LITERAL    ( - - seg)
  R@ 2+                  ( - - seg off)
  R> @ ;                 ( - - seg off type)
\ Ex: REAL*4 SCALAR X

```

9. Consult, e.g., L.J. Scanlon, *op. cit.*; or R. Lafore, *op. cit.* HS/FORTH defines “far” access operators, **@L** and **!L** of all types, that expect a “long” address on the stack. For example, **CODE R32@L DS POP. FWAIT. DS: [BX] DWORD-PTR. FLD. END-CODE**

§§§ The intelligent floating point stack (ifstack)

The ifstack is a more complex data structure than either a simple fstack or the parameter/return stacks. When a typed datum is placed on the ifstack its type must be placed there also.

But the typed data have varying lengths, from 4 to 16 bytes. We can deal with this two different ways: either **ALLOT** enough memory to hold a stack of the longest type, making each position on the ifstack 18 bytes wide (to hold datum plus type); or manage the ifstack as a modified heap, with the address of a given datum being computable from the ifstack-pointer and the data type.

The 18-byte wide ifstack wastes memory, but is easy to program. (In retrospect, this is exactly the method I used to program adaptive numerical quadrature¹⁰.) After several false attempts I settled on the fixed-width ifstack. High level FORTH code for this variant is given below.

<pre> \ TYPED DATA STACK MANAGER TASK FSTACKS FIND CP@ 0 = ?(FLOAD COMPLEX.FTH) \ define data-type tokens 0 CONSTANT REAL*4 1 CONSTANT REAL*8 2 CONSTANT COMPLEX 3 CONSTANT DCOMPLEX CREATE #bytes 4 C, 8 C, 8 C, 16 C, : #BYTES #bytes + C@ ; (type -- length in bytes) \ define scalar and scalars : SCALAR (type --) CREATE DUP , #BYTES ALLOT DOES> DUP@ SWAP 2+ SWAP ; </pre>	<pre> (-- seg off type) \ say: REAL*4 SCALAR X : SCALARS (n type --) SWAP 0 DO DUP SCALAR LOOP DROP ; \ say: 4 DCOMPLEX SCALARS XA XB XC XD \ definitions for the parallel stack of types and data \ Brodie, TF (Brady, NY, 1984) p. 207. CREATE FSTACK 20 18 * 2+ ALLOT \ 2 tos-pointer, 20 18-byte cells : FS.INIT FSTACK 0! ; : >EMPTY (-- seg off) [LISTS @] LITERAL FSTACK DUP@ 18 * 2+ + ; : >FS (seg off type --) \ say: X >FS >R >EMPTY (-- seg off seg' off) R@ OVER ! \ store type on ifstack </pre>
---	--

10. J.V. Noble, "Scientific Computation in FORTH", *Computers in Physics* 3 (1989) 31; and Ch. 8§1§§5 of this book.

```

FS> ( seg off type -- ) \ say: X FS>
FSTACK 1-! \ dec ifstack ptr
>R >EMPTY ( -- seg off seg' off )
DUP@ R@ = \ src.type = dest.type ?
IF 2+ DSWAP ( -- seg' off' seg off )
R> #BYTES ( -- seg' off' seg off n)
CMOVEL \ move data from ifstack
ELSE RDROP CR
." ATTEMPT TO STORE TO
WRONG DATA TYPE" ABORT
THEN ;

\ execution-array defining word
\ HS/FORTH has the faster
\ CASE: ... ;CASE pair for the same job

: G: CREATE ] DOES> OVER + +

```

```

@ EXECUTE ; ( t -- )

G: G@ R32@L R64@L X@L DX@L ;
G: G! R32!L R64!L X!L DX!L ;
\ move data from ifstack to/from FPU
: FS>F ( -- t 87: -- x )
FSTACK 1-! \ dec ifstack ptr
>EMPTY ( -- seg off )
DUP@ >R 2+
R@ ( -- seg off type)
G@ R> ;
\ move data from ifstack to 87stack, leave type

: F>FS ( t -- 87: x -- )
>R >EMPTY ( -- seg off )
R@ OVER !
2+ R> ( -- seg off type)

```

The stack comments and comments should make the preceding code self-explanatory.

§§6 Unary and binary generic operators

We want to define generic unary and binary operators whose run-time action selects the desired operation using information contained in the ifstack. A unary operator such as **FNEGATE** or **FEXP** expects one argument and leaves one result. With a floating-point coprocessor (FPU) the only distinction is between real or complex. This distinction is contained in the second bit of the type descriptor, which we exhibit in Table 5-1 on page 101, in binary notation.

Real and complex can then be distinguished *via* the code fragment

```
2 AND ( type -- 0 = real | 2 = complex)
```

Table 5-1 Bit-patterns of data type descriptors

Type	BINARY representation
REAL*4	00000000 00000000
REAL*8	00000000 00000001
COMPLEX	00000000 00000010
DCOMPLEX	00000000 00000011

Since most unary operators produce results of the same type as their argument, we write a defining word for *generic* unary operators:

```

: GU:  CREATE ] DOES>  ( - - pfa)
      FS>F  ( - - pfa t) \ get data
      UNDER 2 AND  +    ( - - t adr )
      @ EXECUTE         \ do it
      F>FS ;            \ return ans.

```

When we use **GU:** in the form

```
GU:  GNEGATE  FNEGATE  XNEGATE  ;
```

CREATE produces a dictionary entry for **GNEGATE**; **]** turns on the compiler so the previously defined words **FNEGATE** and **XNEGATE** have their addresses compiled into **GNEGATE**'s parameter field; and **DOES>** attaches the run-time code. The run-time code converts the real/complex bit into an offset, 0 or 2 which is added to the address of the daughter word to get the address where the pointer to the actual code is stored. This pointer is fetched and **EXECUTEd**.

A few unary operators like **XABS** (complex absolute value) return real values from complex arguments. If we want to use **GU:** to define, say, **GABS**, we must remember to redefine **XABS** so it zeros the second bit of the type descriptor left on

the stack, before returning its result to the ifstack. This is just a 1 **AND** so is fast.

A binary operator (one that takes two arguments) expects its arguments *and* their types on the ifstack. There is no distinction between single- and double-precision arithmetic on most numeric coprocessors. However, the result must leave the proper type-label on the stack. Here is what we want to happen, illustrated in Fig. 5-2 as a matrix TYPE(arga, argb)

TYPE_{ab}					
	<i>a \ b</i>	R	D	X	DX
R		R	R	X	X
D		R	D	X	DX
X		X	X	X	X
DX		X	DX	X	DX

Fig. 5-2 Types resulting from 2-argument operators

Note: this protocol avoids misleading precision for the results of computations. It seems more scientific than FORTRAN's "convert intermediate results to the precision of the highest-precision operand" protocol.

If we think of the indices and entries in Fig. 5-2 as numbers 0, 1, 2, 3 (so we can use them as indices into a table) rather than as letters, a simple algorithm emerges: the first bit of the result is the logical-AND of the first bits of the two operands, and the second bit of the result is the logical-OR of their second bits. Although we would program this in assembler for speed, the high-level definition is

```

: NEW.TYPE      ( a b -- a2 + b2 + a1b1)
  DDUP          ( -- a b a b)
  AND           ( -- a b ab)
  1 AND         ( -- a b [ab]1)
  -ROT OR       ( -- [ab]1 a + b)
  2 AND         ( -- [ab]1 [a + b]2)
  + ;           ( -- a2 + b2 + a1b1)

```

Since only logical operations are used, **NEW.TYPE** is faster than table lookup or branching. Note that in programming this key word we have obeyed the central FORTH precept: "Keep it simple!" by choosing a data structure (the numeric type tokens 0-3) that is easily manipulated.

We will also need a way to select the appropriate operator from a jump table of addresses. Given that the precision (internal) is irrelevant, again all that matters is whether the number is real or complex, *i.e.* the second bits of the numbers. The first operation must then be to divide by 2 (right-shift by one bit). We then have the matrix of Fig. 5-3 below

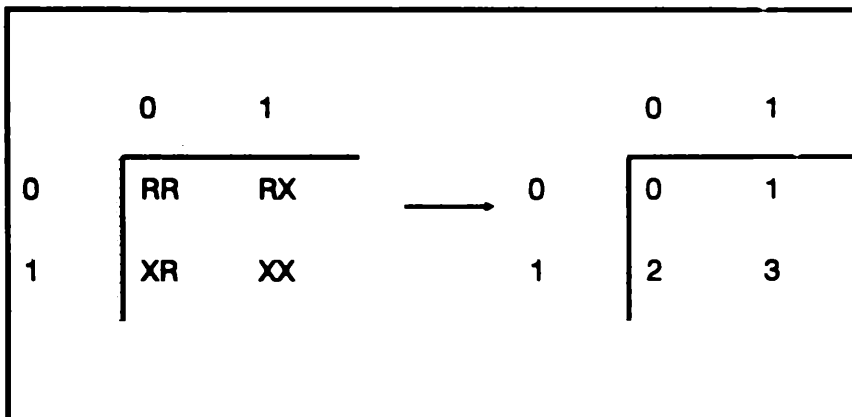


Fig. 5-3 Operator selection matrix

where RR stands for real-real, *etc.* The numerical elements are generated as $2^J + I$. This leads to the word

```

: WHICH.OP      ( a b -- c)
  2/ SWAP 2 AND + ;

```

Thus we come to the binary generic-operator defining word

```

: GB:  CREATE ] DOES>      ( - - pfa)
      FS>F FS>F             ( - - pfa t0 t1)
      NEW.TYPE UNDER      ( - - t' pfa t')
      WHICH.OP 2* +        \ make result-type
      @ EXECUTE            \ select binop
      F>FS ;               \ save result
\say: GB:  G*      F*  F*X  X*F  X*  ;

```

The generic multiply **G***, *e.g.*, picks out, at run-time, which of four routines to use. By using only logical or shift operations we have made even the high-level definitions fairly quick in comparison with the times of floating point operations.

The only instance where one might forego the overhead penalty paid for the convenience of generic coding would be in nested inner loops, such as occur in matrix operations. Here it might pay to code four inner loops, one for each type, and then access them generically, *e.g.*

```

: RLOOP      ... real words ...      ;
: DRLOOP     ... dreal words ...     ;
: XLOOP      ... complex words ...   ;
: DXLOOP     ... dcomplex words ...  ;
G: GLOOP RLOOP DRLOOP XLOOP DXLOOP ;

```

§2 Arrays of typed data

Numerical arrays represent a frequently encountered characteristic feature of scientific programming. Arrays *per se* are hardly foreign to FORTH. Arrays of typed data are novel, however, and therefore worth elaborating. Following Brodie's advice (TF, p. 48ff) we first specify the "user interface" (matrix notation) and then proceed to implementation.

§§1 Improved (FORTRAN-like) array notation

Something like $V(15)$ – the 15'th element of V – is the commonest notation for array elements in high-level languages because lineprinters and terminals do not easily recognize subscripts. In FORTH, the most natural notation would be postfix (RPN), $15\ V$ – but this is both hard to read and unintuitive¹¹. That is, $15\ V$ does not say, immediately and unambiguously, “I am the 15th element of the array V !”

FORTH's idiosyncrasies forbid saying $V(15)$ because the parser recognizes $V(15)$ as a single word¹². Since we want the 15 to be parsed, we would have to modify the FORTRAN-ish notation to $V(\bullet 15)$ or $V(\bullet 15\bullet)$, where \bullet stands for a blank space (ASCII 32). Unfortunately, “(” is a reserved word. While we might place the matrix definitions in a separate vocabulary – which would let us redefine anything we want – “(” is too useful as a comment delineator to dispense with.

This leaves the second possibility, where “(” becomes part of the array name, $V($. To make $V(\bullet 15\bullet)$ work, “)” must become an operator – unless we want to leave postfix notation entirely, with all the complication *that* would entail¹³. Since “)” is not a reserved word, nothing in principle prevents defining it as an operator. However, such usage would conflict with comments.

The square braces, $[]$, are commonly used in matrix notation; however both are reserved FORTH words, *i.e.* forbidden. This leaves the curly braces $\{ \}$, which are unused by FORTH.

Of the two possible forms, $V\bullet\{ \bullet 15 \}$ or $V\bullet\{ \bullet 15\bullet \}$, the latter has the advantage that the opening brace, $\{$, is only part of the name, but reminds us that the name $V\{$ is an array, exactly as names ending with $\$$ are strings, *etc.* The notation suggests a further

-
1. Other authors have noted this and proposed more readable matrix notations. See, *e.g.*, Joe Barnhart, “FORTH and the Fast Fourier Transform” *Dr. Dobbs's Journal*, September, 1984, p. 34. Also Dick Pountains's book *Object-oriented FORTH* (*op cit.*) uses an array naming convention with brackets.
 2. Recall that the standard FORTH delimiter is the ASCII blank (20h = 32d).
 3. See, *e.g.*, L. Brodie, *Thinking FORTH* (*op cit.*) p. 113ff.

mnemonic refinement, namely to place $\{\{$ and $\}\}$ at the ends of 2-dimensional arrays, as in $M\{\{ \bullet 3 \bullet 5 \bullet \}\}$.

How will this notation operate? Clearly, to place the (generalized) address of the n 'th element (of a 1-dimensional array) on the stack we would say

$$V\{\bullet n \bullet\},$$

whereas

$$M\{\{ \bullet m \bullet n \bullet \}\}$$

should analogously place the address of the m,n 'th element of a 2-dimensional array on the stack.

§§2 Large matrices

The defining word **SCALAR** given in §2 above allots space in the dictionary – for most FORTHS, code + data must fit here – or in the **LISTS** segment of HS/FORTH (part of the dictionary). This is OK for variables, but not for arrays, since even a modest matrix would exhaust the (≤ 64 Kbyte) **LISTS** segment.

A **REAL*4** matrix uses 4 bytes per element. The largest such array that can be stored in a 65,536 (*i.e.*, 2^{16})-byte segment is 128×128 . This is the largest array that can be addressed with unsigned 16-bit numbers. On the other hand, a filled IBM PC/XT clone has 640 Kbytes of memory under MS-DOS. Even a generous FORTH kernel (plus DOS) takes up less than 150 K; hence 450 K is available to hold large arrays. Up to 8 Mbytes can be added as EMS storage, assuming a suitable memory management scheme¹⁴. That is, in principle one could tackle matrix problems of order 350×350 . What about speed? The dominant term in solving linear equations by –say– Gaussian elimination with partial pivoting is

14. See, *e.g.*, Ray Duncan, "FORTH support for Intel/Lotus expanded memory", *Dr. Dobb's Journal*, August 1986; also, John A. Lefor and Karen Lund, "Reaching into expanded memory", *PC Text Journal*, May 1987.

$$T = \frac{1}{3}mn^3$$

where m is the time for 1 multiply and 1 add, and n is the order of the matrix. We should also include the fetch + store time, since the bus bandwidth is as much a limiting factor as the FPU arithmetic speed. For the 8086/8087 the time m is of order 400 clock cycles. Thus the asymptotic execution time on a 10 MHz machine should be of order 10 minutes for $n = 350$.

On the 80386/80387 combination running at 25 MHz, the execution time for the same problem should be only 2.3 minutes or so. Thus it would be practical (i.e., execution time \approx 1 hour) on such machines, even without special equipment such as the IIT 80c387, or an array co-processor, or a faster procedure such as Strassen's algorithm (see Ch. 4 §8), to tackle $10^3 \times 10^3$ dense matrix problems.

The crucial question therefore, is memory. The Intel machines were designed around a segmented memory architecture. That is, to avoid having to use (expensive) 32-bit address registers, the 8086/80286 chips were designed to use 16-bit registers. However, these chips have more than 16 external address lines — 20 for the 8086, 24 for the 80286. Thus the absolute address is compounded of two numbers: a **segment descriptor** and an **offset**, which must be present in appropriate registers. The segment descriptor is the **absolute address** of a 16-byte **paragraph**, divided by 16. The offset is any (unsigned) integer from 0 to $2^{16} - 1 = 65,535$ that can fit in a 16-bit register.

The chips contain 4 segment registers: SS (stack segment), CS (code segment), DS (data segment) and ES (extra segment). Five registers, BP, SP, SI, DI and BX, can be used for offsets, although they are not entirely interchangeable (some have specific functions in some of the more complex machine instructions, such as string operations). Manifestly, since the 8086 can address

$$2^{20} = 1,048,576 \text{ bytes ("1 megabyte"),}$$

the largest segment number is $2^{14} - 1 = 16,383$.

A typical (segmented) address is expressed in Intel assembly code as

CS: [BX + SI + 0008]

which translates in words to “add the offset in BX to that in SI and then add 8 to get the total offset; take the segment descriptor in CS, multiply by 16 and add to produce the absolute address.

§§3 Using high memory

HS/FORTH permits accessing all the memory in a PC/AT (up to 1 megabyte) in the following manner:

- Define a named segment of length 1 byte: this marks the beginning of available memory.
- Then tell both FORTH and DOS how much memory you want.

As might be expected, HS/FORTH defines non-standard words (coded as DOS function calls) to use the various DOS service routines that allocate memory, *etc.*¹⁵

```
MEMORY 4+ @ S->D
DCONSTANT MEM.START \ beg. of free memory
40.960 DCONSTANT MAX.PARS
\ 40960 = 655360 /16
: TOTAL.PARS MAX.PARS MEM.START D- ;
\ # pars of memory available
1 SEGMENT SUPERSEG
\ define named segment 1 byte long
TOTAL.PARS DROP FREE-SIZE
\ tell DOS and HS/FORTH about it
```

Having allocated the memory, how can we address it efficiently? We would like the simplicity of double-length integer arithmetic for computing an (absolute) array address, as in

$$\text{abs.adr}(A_{ij}) = \text{abs.adr}(A_{00}) + (\text{row.length} * i + j) * \# \text{BYTES}$$

However, although the absolute address referenced by a segment and offset is unique, *i.e.* the absolute address in bytes is

15. See, *e.g.*, D.N. Jump, *Programmer's guide to MS-DOS*, rev. ed. (Brady Books, New York, 1987).

$$\text{abs.adr} = 16 * \text{segment} + \text{offset} ,$$

the reverse translation, of an absolute address (in bytes) to the segment + offset notation expected by 80x86 processors is *not* unique. This naturally poses a problem when the processor tries to prevent segments from overlapping (protected mode). In such cases, the only answer is a memory management scheme that computes segments and offsets (by brute force) in a non-overlapping fashion. For example, we might define large arrays such that each row has its own segment paragraph.

The 80386 CPU has a third mode that permits direct 32-bit addressing of 4 gigabytes (albeit few computer users have quite this much fast memory available). A scheme for addressing large amounts of RAM in 80386 machines (without leaving MS-DOS) has been discussed in *Dr. Dobb's Journal*¹⁶.

For 8086 PC's and/or real-mode programming on 80286 + machines, we can merely ignore whether segments overlap. Oddly, standard assembly programming books¹⁷ omit this way of addressing segmented memory.

The 8086 permits 32-bit addressing as long as we translate 32-bit addresses to the segment + offset notation expected by the 80x86 processors in real mode. A word that performs this conversion is **SEG.OFF**, defined as

```

: > SEG.OFF          ( d - - seg off)
  OVER 15 AND        ( - - d off)
    -ROT D16/ DROP    ( - - off seg)
  SWAP ;

```

The 32-bit address is placed on the stack as a double-length integer, with the low-order (*i.e.* offset part) above the segment part. The phrase **OVER 15 AND** saves bits 0-3 (of the 32-bit

5. See, e.g., Al Williams, "DOS + 386 = 4 Gigabytes", *Dr. Dobb's Journal*, July 1990, p. 62.

7. C. Morgan and M. Waite, *8086/8088 16-bit microprocessor primer* (Byte/McGraw-Hill, Peterborough, 1982); L. Scanlon, *IBM PC & XT assembly language: a guide for programmers* (Brady/Prentice-Hall, Bowie, Md., 1983); R. Lafore, *Assembly language primer for the IBM PC & XT* (Plume/Waite, New York, 1984).

address); **-ROT D/16 DROP** then shifts the (32-bit) address right 4 bits and drops the least-significant part, to produce the offset. This conversion method produces offsets in the range 00-0F (hex), that clearly have nothing to do with the original offset (that led to the 32-bit absolute address *via* $16 * \text{seg} + \text{off}$).

§§4 A general typed-array definition

For the new syntax to work the word **}** must compute the address of the n'th element of **V{** from the information on the stack, and **}}** must do the same for **M{**. In order to encompass matrices of typed data we specify that the results of the phrases **V{ n }** and **M{ { m n } }** be to leave the generalized address on the parameter stack, i.e. to leave the stack picture (-- seg off type), exactly as with **SCALARS**.

Before we can define **}**, however, we must specify the data structure it operates on, i.e. the array header.

Once again we begin with the user interface. We can opt for maximum generality or maximum simplicity. My first attempt fell into the first category, permitting the user to define a named segment of given length and to define an array in that segment. Lately I realized this generality accomplishes little, so have abandoned it. All arrays will be defined in the heap, named **SUPERSEG** as above. To define a length- 50 **1ARRAY** of 4-byte numbers we will say

```
50 LONG REAL*4 1ARRAY V{
```

Now, before we work out the mechanics of **1ARRAY**, we imagine that an array will be stored as in Fig. 5-4 below:

The proposed data structure consists of an 8-byte header (the **array descriptor**) in the dictionary (**LISTS** in HS/FORTH), with the **body** of the array stored elsewhere. The array descriptor points to the absolute address of the array data (body).

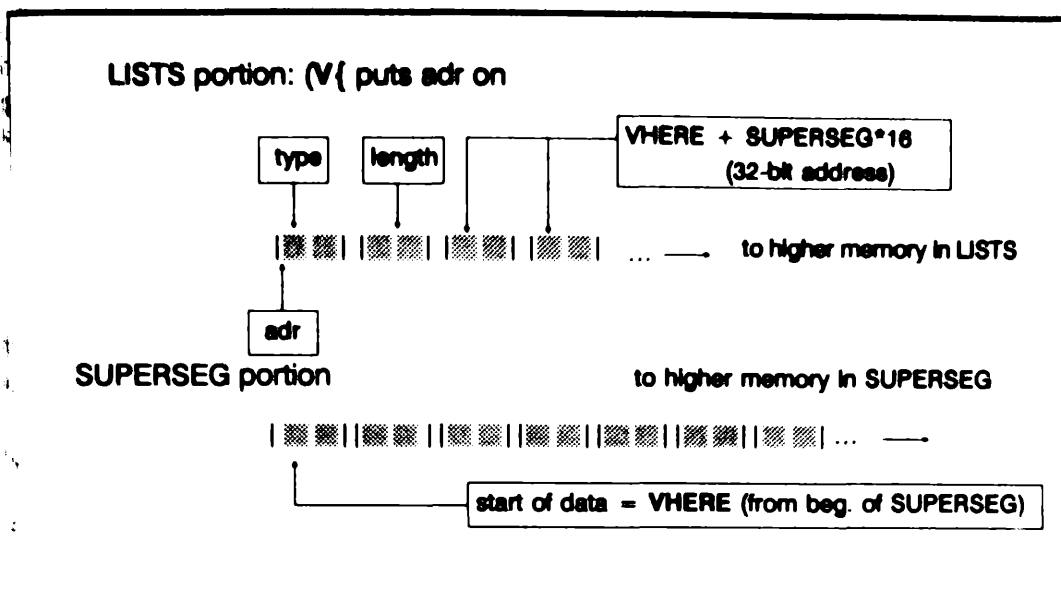


Fig. 5-4 Structure of a 1-dimensional array in SUPERSEG

The array-defining word **1ARRAY** must perform the following tasks:

- place the length and type of the data in the first two cells (4 bytes) of the array descriptor.
- place the 32-bit address (start of data) in the next two cells (4 bytes) of the array descriptor.
- allot the necessary storage in **SUPERSEG**.
- at run-time, place the generalized address, length and type on the parameter stack.

The start of data is handled by **VHERE**, a word that puts the next vacant (32-bit) address in **SUPERSEG** on TOS.

We define **VALLOT** to keep track of the storage used by arrays. **VALLOT** increments the pointer in **VHERE** (and aborts with a warning if the segment length is exceeded).

We first define some auxiliary words:

```
MEMORY 4 + @
S->D DCONSTANT MEM.START
\ beg. of free memory
```

40.960 DCONSTANT MAX.PARS \40960 = 655360/16

: TOTAL.PARS MAX.PARS MEM.START D- ;
 \ # pars avail. mem.

1 SEGMENT SUPERSEG \ named seg. 1 byte long
 TOTAL.PARS DROP
 FREE-SIZE \ tell DOS and HS/FORTH

DVARIABLE VHERE>
 : INIT.VHERE> 0.0 VHERE> DI ;
 INIT.VHERE>
 : VHERE (- - d.offset) VHERE> D@ ;
 : D4/ D2/ D2/ ; : D16/ D4/ D4/ ;

: TOO-BIG? VHERE >SEG.OFF
 0 AND TOTAL.PARS D>
 ABORT" INSUFFICIENT ROOM IN SUPERSEG" ;
 \ check whether new value of VHERE> passes end
 \ of SUPERSEG

: VALLOT (d.#bytes - -)
 VHERE D+ DDUP TOO.BIG?
 VHERE> DI ;

\ Array-defining words
 \ Ex: 50 LONG REAL*4 1ARRAY V{
 \ V{ (- - adr)
 \ V{ 17 } >FS (:: - - V[17])

: LONG DUP ;
 FIND D, 0= ?(: D, SWAP , , ;)
 \ conditionally compile D,
 : 1ARRAY (||t - -)
 CREATE UNDER D, \ t,l into 1st 4 bytes (- - |t)
 SUPERSEG @ 16 M* \ start of SUPERSEG
 VHERE D+ D, \ abs. address → next 4 bytes
 #BYTES M* (|t - - #bytes to allot)
 VALLOT ; \ allot space in the segment
 \ run-time action: (- - adr)

We also need some words to go with **1ARRAY**:

```
: }      ( adr n -- seg.off[n] t )
  SWAP  DUP@  R>      \ type → rstack
  4+  D@
  ROT  R@  #BYTES  M*
  D+  >SEG.OFF  R>  ; ( -- seg.off[n] t )
```

Finally, here is a useful diagnostic word

```
: ?TYPE      ( t -- )      \ it's ok for this to be slow!
  DUP 0 = IF DROP ." REAL*4"  EXIT THEN
  DUP 1 = IF DROP ." REAL*8"  EXIT  THEN
  DUP 2 = IF DROP ." COMPLEX"  EXIT  THEN
  DUP 3 = IF DROP ." DCOMPLEX" EXIT  THEN
  ." NOT A DEFINED DATA TYPE" ABORT ;
```

§§5 2ARRAY and }}

We now want to define arrays of higher dimensionality. For example, to define a 2-dimensional array we might say

```
90 LONG BY 90 WIDE COMPLEX 2ARRAY XA{{
```

This leads to the definitions

```
: BY ;      \ a do-nothing word for style
: WIDE * ;      ( l w -- l*w )
: 2ARRAY      ( l*w t -- ) 1ARRAY ;
```

Now let us define **}}** to fetch the double-indexed address:

```
: }}      ( adr m n -- a[m*l+n] t )
  >R OVER 2+ @      ( -- adr m l*w )
  * R> + } ;
```

By correct factoring (putting some of the work into **WIDE**) we achieved an easy definition of **2ARRAY**. Careful factoring also let us define **}}** in terms of **}**.

§3 Tuning for speed

Some of the words in our typed-data/matrix lexicons should be optimized or redefined in machine code. Accessing matrix elements imposes a non-trivial overhead on matrix operations. We can reduce the execution time with inline code, either in the traditional FORTH manner *via* selected assembler definitions, or with a recursive-descent optimizer such as HS/FORTH's¹⁸.

Experience teaches that optimization is most fruitful (most bang for the buck) applied to entire inner loops and other selected areas of code, rather than to access words *per se*. By hand-coding the innermost loop in matrix inversion and FFT routines, one achieves programs that run in (asymptotically) minimum time on the 8086/8087 chip set.

Significant speed increases in data access could perhaps be obtained with multiple code field (MCF) words, as described by Shaw¹⁹, and as implemented by HS/FORTH in the words **VAR**, **AT**, **IS**, and variants thereof. The disadvantage of MCF style is that compile-time binding, while faster in execution, loses the flexibility of run-time binding. That is, data types would — as with FORTRAN — be specified at compile-time, and lexicons would be recompiled to run with specific types. Run-time binding as described in this Chapter produces *generic* words that can handle all four standard scientific data types, a major advantage over MCF.

FORTH data structures, especially as defined in this Chapter, do little— or no bounds checking, hence do not prevent accidentally overwriting key parts of the operating system.

The new fetch and store words were defined in high-level FORTH for safety. Adding bounds-checking to arrays, at least during the debug cycle, is strongly recommended to avoid crashing, or even damaging, the system.

-
18. J.S. Callahan, *Proc. 1988 Rochester FORTH Conference* (Inst. for Applied FORTH Research, Inc., 1988), p. 39.
 19. G. Shaw, "Forth Shifts Gears, I", *Computer Language* (May 1988) p. 67; "Forth Shifts Gears, II", *Computer Language* (June 1988) p. 61; *Proc. 9th Asilomar FORML Conference (JFAR 5 (1988) 347.)*