8 VARIABLES, CONSTANTS, AND ARRAYS

As we have seen throughout the previous seven chapters, FORTH programmers use the stack to store numbers temporarily while they perform calculations or to pass arguments from one word to another. When programmers need to store numbers more permanently, they use variables and constants.

In this chapter, we'll learn how FORTH treats variables and constants, and in the process we'll see how to directly access locations in memory.

Variables

Let's start with an example of a situation in which you'd want to use a variable--to store the day's date.[†] First we'll create a variable called DATE. We do this by saying

```
VARIABLE DATE
```

If today is the twelfth, we now say

```
12 DATE !
```

that is, we put a twelve on the stack, then give the name of the variable, then finally execute the word $\boxed{!}$, which is pronounced *store*. This phrase stores the number twelve into the variable *DATE*.

Conversely, we can say

[†] For Beginners

Suppose your computer generates bank statements all day, and every statement must show the date. You don't want to keep the date on the stack all the time, and you don't want the date to be part of a definition that you'd have to redefine every day. You want to use a variable.

```
DATE @
```

that is, we can name the variable, then execute the word @, which is pronounced fetch. This phrase fetches the twelve and puts it on the stack. Thus the phrase

```
DATE @ . 12 ok
```

prints the date.

To make matters even easier, there is a FORTH word whose definition is this:

```
: ?   @ . ;
```

So instead of "DATE-fetch-dot," we could simply type

```
DATE ? 12 ok
```

The value of DATE will be twelve until we change it. To change it, we simply store a new number:

```
13 DATE ! ok
DATE ? 13 ok
```

Conceivably we could define additional variables for the month and year:

```
VARIABLE DATE  VARIABLE MONTH  VARIABLE YEAR
```

then define a word called !DATE (for "store-the-date") like this:

```
: !DATE   YEAR !  DATE !  MONTH ! ;
```

to be used like this:

```
7 31 80 !DATE ok
```

then define a word called .DATE (for "print-the-date") like this:

```
: .DATE   MONTH ?  DATE ?  YEAR ? ;
```

Your FORTH system already has a number of variables defined; one is called BASE. BASE contains the number base that you're currently working in. In fact, the definitions of HEX and DECIMAL (and OCTAL, if your system has it) are simply

```
: DECIMAL   10 BASE ! ;
: HEX   16 BASE ! ;
: OCTAL   8 BASE ! ;
```

You can work in any number base by simply storing it into BASE.[†]

Somewhere in the definitions of the system words which perform input and output number conversions, you will find the phrase

```
BASE @
```

because the current value of BASE is used in the conversion process. Thus a single routine can convert numbers in any base. This leads us to make a formal statement about the use of variables:

> In FORTH, variables are appropriate for any value that is used inside a definition which may need to change at any time after the definition has already been compiled.

A Closer Look at Variables

When you create a variable such as DATE by using the phrase

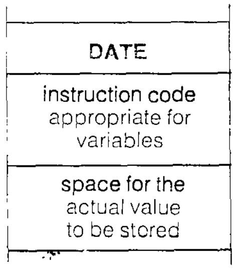
```
VARIABLE DATE
```

you are really compiling a new word, called DATE, into the dictionary. A simplified view would look like this:

[†] For Experts

A three-letter code such as an airport terminal name, can be stored as a single-length unsigned number in base 36. For example:

```
: ALPHA   36 BASE ! ; ok
ALPHA ok
ZAP U. ZAP ok
```

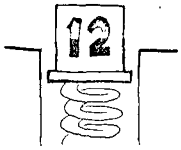


†

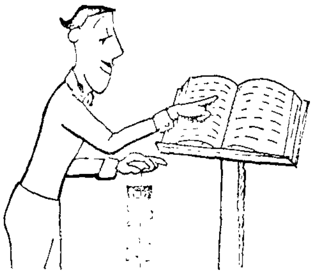
DATE is like any other word in your dictionary except that you defined it with the word VARIABLE instead of the word :. As a result, you didn't have to define what your definition would do; the word VARIABLE itself spells out what is supposed to happen. And here is what happens:

When you say

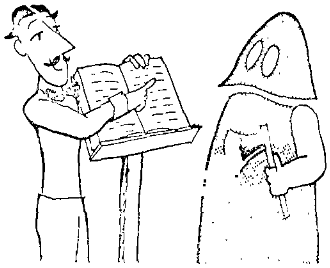
```
12 DATE !
```



Twelve goes onto the stack,



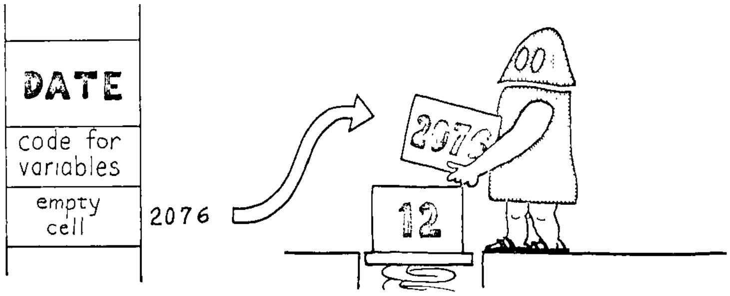
then the text interpreter looks up DATE in the dictionary



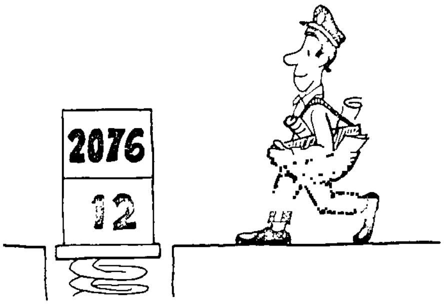
and, finding it, points it out to EXECUTE.

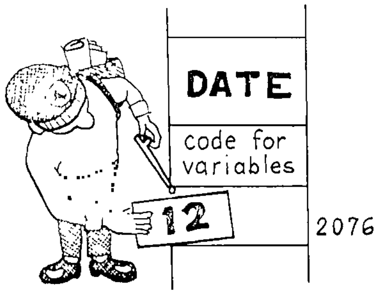
[†] For Experts

In the next chapter we'll show you what a dictionary entry really looks like in memory.



EXECUTE executes a variable by copying the address of the variable's "empty" cell (where the value will go) onto the stack.†





The word ! takes the address (on top) and the value (underneath), and stores the value into that location. Whatever number used to be at that address is replaced by the new number.

(To remember what order the arguments belong in, think of setting down your parcel, then sticking the address label on top.)

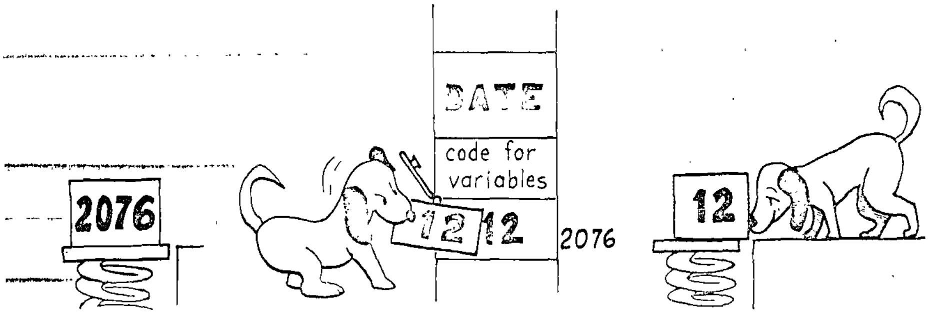
---

†For Beginners

In computer terminology, an address is a number which identifies a location in computer memory. For example, at address 2076 (addresses are usually expressed as hexadecimal, unsigned numbers), we can have a 16-bit representation of the value 12. Here 2076 is the "address"; 12 is the "contents."

The word @ expects one argument only: an address, which in this case is supplied by the name of the variable, as in

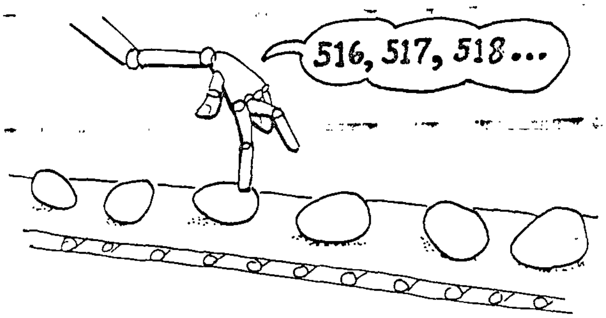
```
DATE @
```



Using the value on the stack as an address, the word @ pushes the contents of that location onto the stack, "dropping" the address. (The contents of the location remain intact.)

Using a Variable as a Counter

In FORTH, a variable is ideal for keeping a count of something. To reuse our egg-packer example, we might keep track of how many eggs go down the conveyor belt in a single day. (This example will work at your terminal, so enter it as we go.)



First we can define

```
VARIABLE EGGS
```

to keep the count in. To start with a clean slate every morning, we would store a zero into EGGS by executing a word whose definition looks like this:

```
: RESET   0 EGGS ! ;
```

Then somewhere in our egg-packing application, we would define a word which executes the following phrase every time an egg passes an electric eye on the conveyor:

```
1 EGGS +!
```

The word +! adds the given value to the contents of the given address.[†] (It doesn't bother to tell you what the contents are.) Thus the phrase

```
1 EGGS +!
```

increments the count of eggs by one. For purposes of illustration, let's put this phrase inside a definition like this:

```
: EGG   1 EGGS +! ;
```

At the end of the day, we would say

```
EGGS ?
```

to find out how many eggs went by since morning.

Let's try it:

```
RESET ok
EGG ok
EGG ok
EGG ok
EGGS ? 3 ok
```

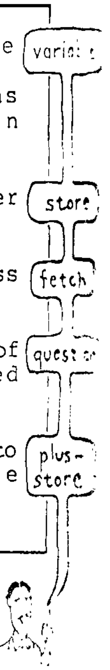
Here's a review of the words we've covered in the chapter so far:

[†] For the Curious

+! is usually defined in assembly language, but an equivalent high-level definition is

```
: +!   DUP @ ROT  + SWAP ! ;
```

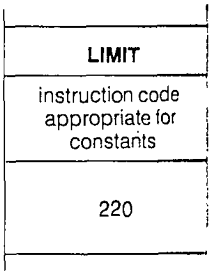
VARIABLE xxx	(--)	Creates a variable named xxx;
	xxx: (-- adr)	the word xxx returns its address when executed.
!	(n adr --)	Stores a 16-bit number into the address.
@	(adr -- n)	Replaces the address with its contents.
?	(adr --)	Prints the contents of the address, followed by one space.
+!	(n adr --)	Adds a 16-bit number to the contents of the address.



Constants

While variables are normally used for values that may change, constants are used for values that won't change. In FORTH, we create a constant and set its value at the same time, like this:

```
220 CONSTANT LIMIT
```



Here we have defined a constant named LIMIT, and given it the value 220. Now we can use the word LIMIT in place of the value, like this:

```
: ?TOO.HOT   LIMIT > IF ." DANGER -- REDUCE HEAT " THEN ;
```

If the number on the stack is greater than 220, then the warning message will be printed.

Notice that when we say

```
LIMIT
```

we get the value, not the address. We don't need the "fetch."

This is an important difference between variables and constants.[†] The reason for the difference is that with variables, we need the address to have the option of fetching or storing. With constants, we always want the value; we almost never store.

One use for constants is to name a hardware address. For example, a microprocessor-controlled camera application might contain this definition:

```
: PHOTOGRAPH   SHUTTER OPEN  TIME EXPOSE  SHUTTER CLOSE ;
```

Here the word SHUTTER has been defined as a constant so that execution of SHUTTER returns the hardware address of the camera's shutter. It might, for example, be defined:

```
HEX
3E27 CONSTANT SHUTTER
DECIMAL
```

The words OPEN and CLOSE might be defined simply as

```
: OPEN    1 SWAP ! ;
: CLOSE   0 SWAP ! ;
```

so that the phrase

```
SHUTTER OPEN
```

writes a "1" to the shutter address, causing the shutter to open.

Here are some situations when it's good to define numbers as constants:

1. When it's important that you make your application more readable. One of the elements of FORTH style is that definitions should be self-documenting, as is the definition of PHOTOGRAPH above.

[†] For People Who Intend to Use polyFORTH's Target Compiler T.M.

In your case the difference is more profound. A constant's <u>value</u> will be compiled into PROM; a variable compiles into PROM a reference to a location in RAM.

2. When it's more convenient to use a name instead of the number. For example, if you think you may have to change the value (because, for instance, the hardware might get changed) you will only have to change the value once--in the block where the constant is defined--then recompile your application.

3. When you are using the same value many times in your application. In the compiled form of a definition, reference to a constant requires less memory space.[†]

CONSTANT xxx	(n --) xxx: (-- n)	Creates a constant named xxx with the value n; the word xxx returns n when executed.

[†] For polyFORTH Users

Because of reason 3, polyFORTH includes constant-definitions of two often-used numbers:

```
0 CONSTANT 0
1 CONSTANT 1
```

Double-length Variables and Constants[†]

You can define a double-length variable by using the word 2VARIABLE. For example,

```
2VARIABLE DATE
```

Now you can use the FORTH words 2! (pronounced two-store) and 2@ (two-fetch) to access this double-length variable. You can store a double-length number into it by simply saying

```
800,000 DATE 2!
```

and fetch it back with

```
DATE 2@ D. 800000 ok
```

Or you can store the full month/date/year into it, like this:

```
7/16/81 DATE 2!
```

and fetch it back with

```
DATE 2@ .DATE 7/16/81 ok
```

assuming that you've loaded the version of .DATE we gave in the last chapter.[‡]

You can define a double-length constant by using the FORTH word [illegible] like this:

```
200,000 2CONSTANT APPLES
```

Now the word APPLES will place the double-length number on the stack.

```
APPLES D. 200000 ok
```

[†] FORTH-79 Standard

The words described in this section are not required except in the Double Number Word Set.

[‡] For polyFORTH Users

polyFORTH uses an even-more-clever arrangement to store the date as one single-length integer.

Use of 2CONSTANT becomes necessary when you need to include a double-length value inside a definition. In FORTH the only way to do this is by first defining the double-length value as a 2CONSTANT. For example, to define a word which adds 400,000 to a double-length value on the stack, we must define

```
400,000 2CONSTANT MUCH
: MUCH-MORE   MUCH D+ ;
```

in order to be able to say

```
APPLES MUCH-MORE D. 600000 ok
```
†

As the prefix "2" reminds us, we can also use 2CONSTANT to define a pair of single-length numbers. The reason for putting two numbers under the same name is a matter of convenience and of saving space in the dictionary.

As an example, recall (from Chap. 5) that we can use the phrase

```
355 113 */
```

to multiply a number by an approximation of pi. We could store these two integers as a 2CONSTANT as follows:

```
355 113 2CONSTANT PI
```

then simply use the phrase

```
PI */
```

as in

```
10000 PI */ . 31415 ok
```

Here is a review of the double-length data-structure words:

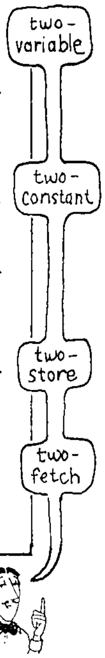
---

† For polyFORTH Users

polyFORTH includes the following definition for a double-length zero for convenient use inside a colon definition:

```
0. 2CONSTANT 0.
```

2VARIABLE xxx	(--) xxx: (-- adr)	Creates a double-length variable named xxx; the word xxx returns its address when executed.
2CONSTANT xxx	(d --) xxx: (-- d)	Creates a double-length constant named xxx with the value d; the word xxx returns the value d when executed.
2!	(d adr --)	Stores a double-length number into the address.
2@	(adr -- d)	Returns the double-length contents of the address.

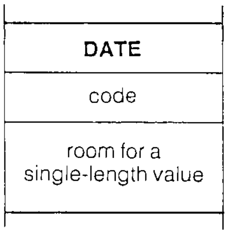


Arrays

As you know, the phrase

    VARIABLE DATE

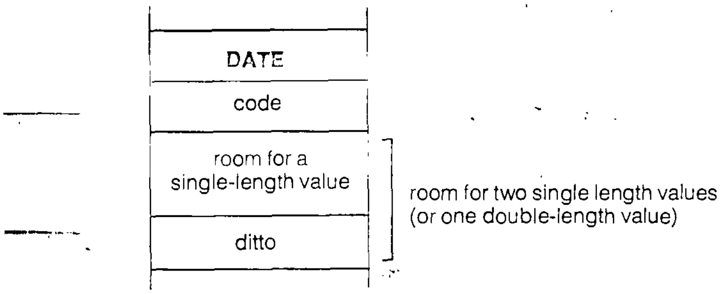
creates a definition which conceptually looks like this:



Now if you say

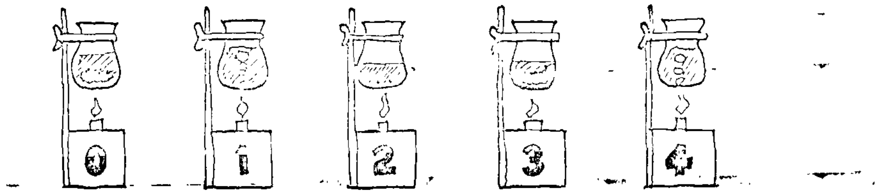
    2 ALLOT

an additional two bytes are allotted in the definition, like this:



The result is the same as if you had used 2VARIABLE. By changing the argument to ALLOT, however, you can define any number of variables under the same name. Such a group of variables is called an "array."

For example, let's say that in our laboratory, we have not just one, but five burners that heat various kinds of liquids.



We can make our word ?TOO-HOT check that all five burners have not exceeded their individual limit if we define LIMIT using an array rather than a constant.

Let's give the array the name LIMITS, like this:

```
    VARIABLE LIMITS  8 ALLOT
```

The phrase "8 ALLOT" gives the array an extra eight bytes or four cells (five cells in all).

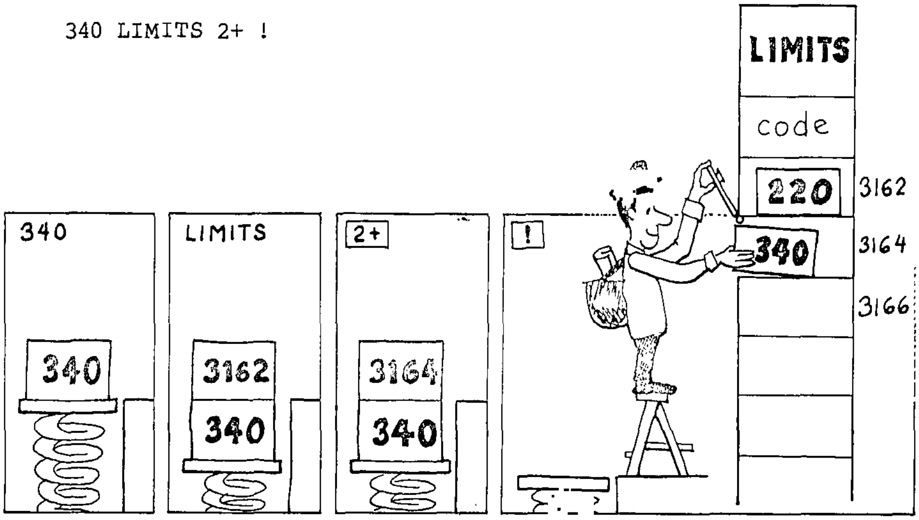
LIMITS	addresses
code	↓
room for burner-0's limit	3162
room for burner-1's limit	3164
room for burner-2's limit	3166
room for burner-3's limit	3168
room for burner-4's limit	316A

Suppose we want the limit for burner 0 to be 220. We can store this value by simply saying

```
220 LIMITS !
```

because LIMITS returns the address of the first cell in the array. Suppose we want the limit for burner 1 to be 340. We can store this value by adding 2 bytes to the address of the original cell, like this:

```
340 LIMITS 2+ !
```



We can store limits for burners 2, 3, and 4 by adding the "offsets" 4, 6, and 8, respectively, to the original address. Since the offset is always double the burner number, we can define the convenient word

```
: LIMIT   2* LIMITS + ;
```

to take a burner number on the stack and compute an address that reflects the appropriate offset.†

Now if we want the value 170 to be the limit for burner 2, we simply say

```
170 2 LIMIT !
```

or similarly, we can fetch the limit for burner 2 with the phrase

```
2 LIMIT ? 170 ok
```

This technique increases the usefulness of the word LIMIT, so that we can redefine ?TOO.HOT as follows:

```
: ?TOO.HOT   ( burner# temp -- )
   LIMIT @ >  IF ." DANGER -- REDUCE HEAT " THEN ;
```

which works like this:

```
210 0 ?TOO.HOT ok
230 0 ?TOO.HOT DANGER -- REDUCE HEAT ok
300 1 ?TOO.HOT ok
350 1 ?TOO.HOT DANGER -- REDUCE HEAT ok
```

etc.

---

†For Beginners

a) Some people call the "offset" an "index," and some people say that one uses an offset to "index into" an array.

b) The reason we number our burners 0 through 4 instead of 1 through 5 is so that we can use the burner number itself (doubled for byte addressing) as the offset.

A thing which most people would call the "first" in a series, programmers think of as the "zeroth." Still, if you need to call the burner on the left "burner 1," you can simply change LIMIT to say

```
: LIMIT   1- 2* LIMITS + ;
```

Another Example — Using an Array for Counting

Meanwhile, back at the egg ranch:

Here's another example of an array. In this example, each element of the array is used as a separate counter. Thus we can keep track of how many cartons of "extra large" eggs the machine has packed, how many "large," and so forth.

Recall from our previous definition of EGGSIZE (in Chap. 4) that we used four categories of acceptable eggs, plus two categories of "bad eggs."

```
0 REJECT
1 SMALL
2 MEDIUM
3 LARGE
4 EXTRA LARGE
5 ERROR
```

So let's create an array that is six cells long:

```
VARIABLE COUNTS  10 ALLOT
```

The counts will be incremented using the word +!, so we must be able to set all the elements in the array to zero before we begin counting. The phrase

```
COUNTS 12 0 FILL
```

will fill twelve bytes, starting at the address of COUNTS, with zeros. If your FORTH system includes the word ERASE,† it's better to use it in this situation. ERASE fills the given number of bytes with zeroes. Use it like this:

```
COUNTS 12 ERASE
```

FILL	(adr n b --)	Fills n bytes of memory, beginning at the address, with value b.
ERASE	(adr n --)	Fills n bytes of memory, beginning at the address, with zeroes.

---

† FORTH-79 Standard

ERASE is included in the optional Reference Word Set.

For convenience, we can put the phrase inside a definition, like this:

```
: RESET   COUNTS 12 ERASE ;
```

Now let's define a word which will give us the address of one of the counters, depending on the category number it is given (0 through 5), like this:

```
: COUNTER   2*  COUNTS + ;
```

and another word which will add one to the counter whose number is given, like this:

```
: TALLY   COUNTER 1 SWAP +! ;
```

The "1" serves as the increment for +!, and SWAP puts the arguments for +! in the order they belong, i.e., (n adr --).

Now, for instance, the phrase

```
3 TALLY
```

will increment the counter that corresponds to large eggs.

Now let's define a word which converts the weight per dozen into a category number:

```
: CATEGORY   DUP 18 < IF 0 ELSE
             DUP 21 < IF 1 ELSE
             DUP 24 < IF 2 ELSE
             DUP 27 < IF 3 ELSE
             DUP 30 < IF 4 ELSE
                         5
           THEN THEN THEN THEN THEN   SWAP DROP ;†
```

(By the time we get to the phrase "SWAP DROP," we will have two values on the stack: the weight which we have been DUPing and the category number, which will be on top. We want only the category number; "SWAP DROP" eliminates the weight.)

---

† For Experts

We'll see a simpler definition at the end of this chapter.

For instance, the phrase

```
25 CATEGORY
```

will leave the number 3 on the stack. The above definition of CATEGORY resembles our old definition of EGGSIZE, but, in the true FORTH style of keeping words as short as possible, we have removed the output messages from the definition. Instead, we'll define an additional word which expects a category number and prints an output message, like this:

```
: LABEL   DUP  0= IF ." REJECT "       ELSE
          DUP 1 = IF ." SMALL "        ELSE
          DUP 2 = IF ." MEDIUM "       ELSE
          DUP 3 = IF ." LARGE "        ELSE
          DUP 4 = IF ." EXTRA LARGE "  ELSE
                     ." ERROR "
          THEN THEN THEN THEN THEN DROP ; †
```

For example:

```
1 LABEL SMALL ok
```

Now we can define EGGSIZE using three of our own words:

```
: EGGSIZE   CATEGORY  DUP LABEL  TALLY ;
```

Thus the phrase

```
23 EGGSIZE
```

will print

```
MEDIUM ok
```

at your terminal and update the counter for medium eggs.

How will we read the counters at the end of the day? We could check each cell in the array separately with a phrase such as

```
3 COUNTER ?
```

(which would tell us how many "large" cartons were packed). But let's get a little fancier and define our own word to print a table of the day's results in this format:

---

†For Experts

We'll see a more elegant version of this definition in the next chapter.

```
QUANTITY      SIZE

    1         REJECT
  112         SMALL
  132         MEDIUM
  143         LARGE
  159         EXTRA LARGE
    0         ERROR
```

Since we have already devised category numbers, we can simply use a DO loop and index on the category number, like this:

```
: REPORT   PAGE   ." QUANTITY     SIZE"  CR CR
     6 0 DO  I COUNTER @  5 U.R
             7 SPACES  I LABEL CR        LOOP ;
```

(The phrase

```
I COUNTER @  5 U.R
```

takes the category number given by I, indexes into the array, and prints the contents of the proper element in a five-column field.)

Factoring Definitions

This is a good time to talk about factoring as it applies to FORTH definitions. We've just seen an example in which factoring simplified our problem.

Our first definition of EGGSIZE, from Chap. 4, categorized eggs by weight and printed the name of the categories at the terminal. In our present version we factored out the "categorizing" and the "printing" into two separate words. We can use the word CATEGORY to provide the argument either for the printing word or the counter-tallying word (or both). And we can use the printing word, LABEL, in both EGGSIZE and REPORT.

As Charles Moore, the inventor of FORTH, has written:

> A good FORTH vocabulary contains a large number of small words. It is not enough to break a problem into small pieces. The object is to isolate words that can be reused.

For example, in the recipe:

Get can of tomato sauce.
Open can of tomato sauce.
Pour tomato sauce into pan.
Get can of mushrooms.
Open can of mushrooms.
Pour mushrooms into pan.

you can "factor out" the getting, opening, and pouring, since they are common to both cans. Then you can give the factored-out process a name and simply write:

```
TOMATOES ADD
MUSHROOMS ADD
```

and any chef who's graduated from the Postfix School of Cookery will know exactly what you mean.

Not only does factoring make a program easier to write (and fix!), it saves memory space, too. A reusable word such as ADD gets defined only once. The more complicated the application, the greater the savings.

Here's another thought about FORTH style before we leave the egg ranch. Recall our definition of EGGSIZE

```
: EGGSIZE   CATEGORY DUP LABEL  TALLY ;
```

CATEGORY gave us a value which we wanted to pass on to both LABEL and TALLY, so we include the DUP. To make the definition "cleaner," we might have been tempted to take the DUP out and put it inside the definition of LABEL, at the beginning. Thus we might have written

```
: EGGSIZE   CATEGORY LABEL  TALLY ;
```

where CATEGORY passes the value to LABEL, and LABEL passes it on to TALLY. Certainly this approach would have worked. But then, when we defined REPORT, we would have had to say

```
I LABEL DROP
```

instead of simply

```
I LABEL
```

FORTH programmers tend to follow this convention: when possible, words should destroy their own parameters. In general, it's better to put the DUP inside the "calling definition" (EGGSIZE, here) than in the "called" definition (LABEL, here).

Another Example — "Looping" through an Array

We'd like to introduce a little technique that is relevant to arrays. We can best illustrate this technique by writing our own definition of a FORTH word called DUMP.[†] DUMP is used to print out the contents of a series of memory addresses. The usage is

```
adr count DUMP
```

For instance, we could enter

```
COUNTS 12 DUMP
```

to print out the contents of our egg-counting array called COUNTS. Since DUMP is primarily designed as a programming tool to print out the contents of memory locations, it prints either byte-by-byte or cell-by-cell, depending on the type of addressing the computer uses. Our version of DUMP will print cell-by-cell.

Obviously our DUMP will involve a DO loop. The question is: what should we use for an index? Although we might use the count itself (0 - 6) as the loop index, it's better to use the address as the index.

The address of COUNTS will be the starting index for the loop, while the address plus the count will serve as the limit, like this:

```
: DUMP   OVER + SWAP  DO CR I @  5 U.R  2 /LOOP ; ‡
```

The key phrase here is

```
OVER + SWAP
```

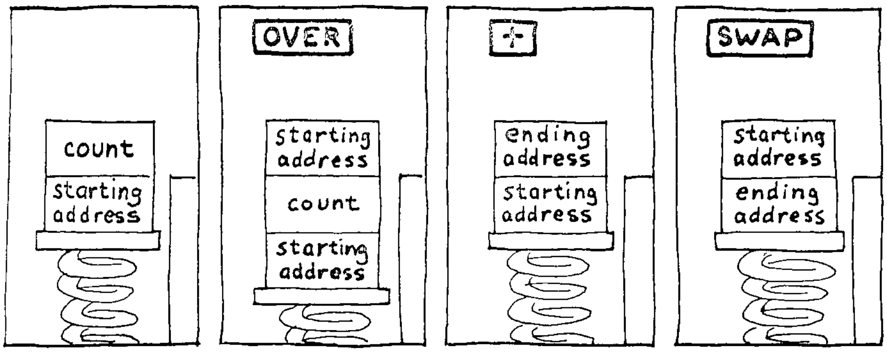
which immediately precedes the DO.

[†] FORTH-79 Standard

The Standard does not require DUMP.

[‡] For Those Whose Systems Do Not Have /LOOP

Substitute +LOOP.



The ending and starting addresses are now on the stack, ready to serve as the limit and index for the DO loop. Since we are "indexing on the addresses," once we are inside the loop we merely have to say

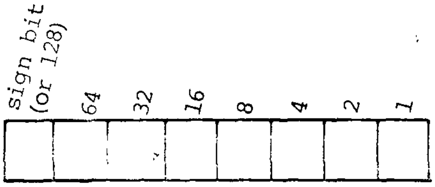
```
I @  5 U.R
```

to print the contents of each element in the array. Since we are examining bytes in pairs (because @ fetches a 16-bit value), we increment the index by <u>two</u> each time, by using

```
2 /LOOP
```

Byte Arrays

FORTH lets you create an array in which each element consists of a single byte rather than a full cell. This is useful any time you are storing a series of numbers whose range fits into that which can be expressed within eight bits.



The range of an unsigned 8-bit number is 0 to 255. Byte arrays are also used to store ASCII character strings. The benefit of using a byte array instead of a cell array is that you can get the same amount of data in half the memory space.

The mechanics of using a byte array are the same as using a cell array except that

1. you don't have to double the offset, since each element corresponds to one address, and
2. you must use the words C! and C@ instead of ! and @. These words, which operate on byte values only, have been given the prefix "C" because their typical use is accessing ASCII characters.

C!	(b adr --)	Stores an 8-bit value into the address.
C@	(adr -- b)	Fetches an 8-bit value from the address.

Initializing an Array

Many situations call for an array whose values never change during the operation of the application and which may as well be stored into the array at the same time that the array is created, just as CONSTANTs are. FORTH provides the means to accomplish this through the two words CREATE and , (pronounced *create* and *comma*).

Suppose we want permanent values in our LIMITS array. Instead of saying

```
    VARIABLE LIMITS  8 ALLOT
```

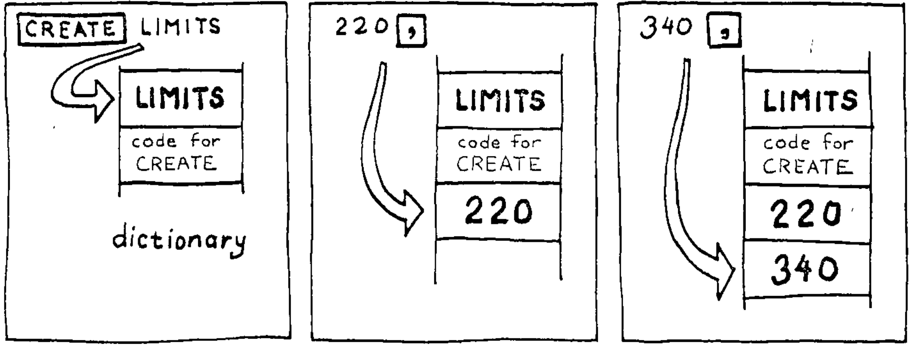
we can say

```
    CREATE LIMITS 220 ,  340 ,  170 ,  100 ,  190 ,
```

Usually the above line would be loaded from a disk block, but it also works interactively.

Like the word VARIABLE, CREATE puts a new name in the dictionary at compile time and returns the address of that definition when it is executed. But it does not "allot" any bytes for a value.

The word , takes a number off the stack and stores it into the array. So each time you express a number and follow it with ,, you add one cell to the array.†



---

† For Newcomers

Ingrained habits, learned from English writing, lead some newcomers to forget to type the final , in the line. Remember that , does not separate the numbers, it *compiles* them.

You can access the elements in a CREATE array just as you would the elements in a VARIABLE array. For example:

```
LIMITS 2+ @ 340 ok
```

You can even store new values into the array, just as you would into a VARIABLE array, as long as you don't do this in an application that you someday hope to target compile.†

To initialize a byte-array that has been defined with CREATE, you can use the word C, (c-comma).‡ For instance, we could store each of the values used in our egg-sorting definition CATEGORY as follows:

```
CREATE SIZES  18 C,  21 C,  24 C,  27 C,  30 C,  255 C,
```

This would allow us to redefine CATEGORY using a DO loop rather than a series of nested IF...THEN statements, as follows‡

```
: CATEGORY   6 0 DO  DUP  SIZES I + C@
     < IF  DROP  I LEAVE  THEN  LOOP ;
```

Note that we have added a maximum (255) to the array to simplify our definition regarding category 5.

Including the initialization of the SIZES array, this version takes only three lines of source text as opposed to six and takes less space in the dictionary, too.

---

†For People Who Intend to Use polyFORTH's Target Compiler

In a target-compiled application, VARIABLE arrays will reside in RAM; tables defined by CREATE and initialized by , or C, will reside, fixed, in PROM.

‡FORTH-79 Standard

C, is included in the optional Reference Word Set.

‡For People Who Don't Like Guessing How It Works

The idea here is this: since there are five possible categories, we can use the category numbers as our loop index. Each time around, we compare the number on the stack against the element in SIZES, offset by the current loop index. As soon as the weight on the stack is greater than one of the elements in the array, we leave the loop and use I to tell us how many times we had looped before we "left." Since this number is our offset into the array, it will also be our category number.

Here is a list of the FORTH words we've covered in this chapter:

CONSTANT xxx	(n --) xxx: (-- n)	Creates a constant named xxx with the value n; the word xxx returns n when executed.
VARIABLE xxx	(--) xxx: (-- adr)	Creates a variable named xxx; the word xxx returns its address when executed.
CREATE xxx	(--) xxx: (-- adr)	Creates a dictionary entry (head and code pointer only) named xxx; the word xxx returns its address when executed.
!	(n adr --)	Stores a 16-bit number into the address.
@	(adr -- n)	Replaces the address with its contents.
?	(adr --)	Prints the contents of the address, followed by one space.
+!	(n adr --)	Adds a 16-bit number to the contents of the address.
ALLOT	(n --)	Adds n bytes to the parameter field of the most recently defined word.
,	(n --)	Compiles n into the next available cell in the dictionary.
C!	(b adr --)	Stores an 8-bit value into the address.
C@	(adr -- b)	Fetches an 8-bit value from the address.
FILL	(adr n b --)	Fills n bytes of memory, beginning at the address, with value b.
BASE	(n --)	A variable which contains the value of the number base being used by the system.

Double-length Operators (Optional in FORTH-79 Standard)

2VARIABLE xxx	(--) xxx: (-- adr)	Creates a double-length variable named xxx; the word xxx returns its address when executed.
2CONSTANT xxx	(d --) xxx: (-- d)	Creates a double-length constant named xxx with the value d; the word xxx returns the value d when executed.
2!	(d adr --)	Stores a double-length number into the address.
2@	(adr -- d)	Returns the double-length contents of the address.

Words Included in the FORTH-79 Standard Reference Word Set

C,	(b --)	Compiles b into the next available byte in the dictionary.
DUMP	(adr u --)	Displays u bytes of memory, starting at the address.
ERASE	(adr n --)	Stores zeroes into n bytes of memory, beginning at adr.

Additional Words Available in Some Systems

0	(-- 0)	Returns the constant zero.
1	(-- 1)	Returns the constant one.
0.	(-- 0 0)	Returns the double-length constant zero.

KEY

n, n1 ...	16-bit signed numbers	b	8-bit byte
d, d1, ...	32-bit signed numbers	f	Boolean flag
u, u1, ...	16-bit unsigned numbers	c	ASCII character value
ud, ud1, ...	32-bit unsigned numbers	adr	address

Review of Terms

Array	a series of memory locations with a single name. Values can be stored and fetched into the individual locations by giving the name of the array and adding an offset to its address.
Constant	a value which has a name. The value is stored in memory and usually never changes.
Factoring	as it applies to programming in FORTH, simplifying a large job by extracting those elements which might be reused and defining those elements as operations.
Fetch	to retrieve a value from a given memory location.
Initialize	to give a variable (or array) its initial value(s) before the rest of the program begins.
Offset	a number which can be added to the address of the beginning of an array to produce the address of the desired location within the array.
Store	to place a value in a given memory location.
Variable	a location in memory which has a name and in which values are frequently stored and fetched.

Problems — Chapter 8

1. a) Write two words called BAKE-PIE and EAT-PIE. The first word increases the number of available PIES by one. The second decreases the number by one and thanks you for the pie. But if there are no pies, it types "What pie?" (Make sure you start out with no pies.)

```
EAT-PIE WHAT PIE?
BAKE-PIE ok
EAT-PIE THANK YOU! ok
```

   b) Write a word called FREEZE-PIES which takes all the available pies and adds them to the number of pies in the freezer. Remember that frozen pies cannot be eaten.

```
BAKE-PIE BAKE-PIE FREEZE-PIES ok
PIES ? 0
FROZEN-PIES ? 2 ok
```

2. Define a word called .BASE which prints the current value of the variable BASE in decimal. Test it by first changing BASE to some value other than ten. (This one's trickier than it may seem.)

```
DECIMAL .BASE 10 ok
HEX .BASE 16 ok
```

3. Define a number-formatting word called M. which prints a double-length number with a decimal point. The position of the decimal point within the number is movable and depends on the value of a variable that you will define as PLACES. For example, if you store a "1" into PLACES, you will get

```
200,000 M. 20000.0 ok
```

that is, with the decimal point one place from the right. A zero in PLACES should produce no decimal point at all.

4. In order to keep track of the inventory of colored pencils in your office, create an array, each cell of which contains the count of a different colored pencil. Define a set of words so that, for example, the phrase

   RED PENCILS

   returns the address of the cell that contains the count of red pencils, etc. Then set these variables to indicate the following counts:

   23 red pencils
   15 blue pencils
   12 green pencils
    0 orange pencils

5. A histogram is a graphic representation of a series of values. Each value is shown by the height or length of a bar. In this exercise you will create an array of values and print a histogram which displays a line of "*"s for each value. First create an array with about ten cells. Initialize each element of the array with a value in the range of zero to seventy. Then define a word PLOT which will print a line for each value. On each line print the number of the cell followed by a number of "*"s equal to the contents of that cell.

   For example, if the array has four cells and contains the values 1, 2, 3, and 4, then PLOT would produce:

```
1 *
2 **
3 ***
4 ****
```

6. Create an application that displays a tic-tac-toe board, so that two human players can make their moves by entering them from the keyboard. For example, the phrase

```
4 X!
```

puts an "X" in box 4 (counting starts with 1) and produces this display:

```
    |   |
---------
  X |   |
---------
    |   |
```

Then the phrase

```
3 O!
```

puts an "O" in box 3 and prints the display:

```
    |   | O
---------
  X |   |
---------
    |   |
```

Use a byte array to remember the contents of the board, with the value 1 to signify an "X," a -1 to signify a "O," and a 0 to signify an empty box.

(NOTE: until we explain more about vocabularies, avoid naming anything "X," since this may conflict with the editor's X.)