$10 Volume XX, Number 5,6 January 1999 April

FORTH
DIMENSIONS

CONTENTS

DEPARTMENTS

EDITORIAL

The Practice of Forth

Forth always has been a language whose success was rooted not in theory but in practice. Despite a general lack of corporate or university sponsorship — with apologies to those companies and institutions of higher learning in which Forth has indeed been championed over the years — it has been in the trenches that Forth has proven its efficacy, efficiency, and vitality. A few publications have objectively documented Forth's strengths but most, relying by necessity on advertising dollars and appeal to mass interests in order to address their understandably bottom-line concerns, largely have ignored it. This is not to say that Forth is an unpublished language; the *Bibliography of Forth References* which was maintained for a number of years by The Institute for Forth Application and Research, documented a surprising depth and breadth of coverage, both academic and popular, of this language. (The *Bibliography,* when last I saw it, was sadly out of date; if updated, it probably would double its already impressive size.)

Despite the fondest wishes of many, Forth has never achieved mass appeal. Instead, it has suffered the fate of the long-distance runner, whose success lies in crossing the finish line, not in besting the pack.

But Forth mostly is a tool for toolbuilders and problem solvers, not the mass market. Its adaptability and flexibility have been of most value in situations calling for outstanding performance under unusual constraints. Fast development needed? Skillful Forth programmers regularly deliver full-featured programs in the time required by skilled users of other languages to deliver an initial prototype. Few resources available? Forth's model allows a degree of application functionality that can only be viewed as incredible in hardware that barely accommodates the run-time kernel of other languages.

Of course, true to its historical trend, this is swimming upstream. General practice these days — at least the tales that make news and drive up costs for consumers and small enterprises — is to throw more-expensive hardware at a problem, to deploy larger programming teams, to design solutions that ultimately will require expensive maintenance and administrative personnel until a bigger, costlier solution relegates the old one to the scrap heap.

But in the trenches, the troops carry on. Alone or in teams, proficient Forth programmers continue the daily work of finding appropriate niches, and of delivering good work on time. Forth's greatest asset is the integrity and diligence of its users who appreciate the benefits inherent in, or which can be coaxed from, what the mainstream might view as limitations.

Since its inception, Forth also has benefitted from the efforts of an even smaller minority of adherents, a few people whose public contributions have been not so much the programs they write or features they introduce to the language, but their ability to help this dispersed community of independent-minded users to cohere and communicate and cooperate in ways that benefit everyone. The loss of one of those people, as happened last spring, reminds us to be very grateful for each person who takes the time and thought necessary to share their experience, knowledge, and even wisdom, with the rest of us.

In Memoriam

With great regret, we must report that Robert Reiling passed away on Wednesday, May 5 of this year.

In the Forth community, Mr. Reiling was the director of the annual FORML Conference, and was a past President of the Forth Interest Group. His diplomacy and professional demeanor, as well as his personal commitment and friendliness, could always be relied upon, and he will be missed. His dedication and encouragement also extended to groups that included the seminal Homebrew Computer Club and local ham radio enthusiasts.

Bob had contracted cancer, and responded to treatment favorably enough to direct the 20th FORML Conference last November and, shortly thereafter, to resume his full-time work until the illness recurred.

We extend our condolences to Bob's friends and family and, like many others, are very grateful for his contributions and support.

Forth Dimensions
Volume XX, Number 5,6
January 1999 April

Published by the
Forth Interest Group

Editor
Marlin Ouverson

Circulation/Order Desk
Trace Carter

Forth Dimensions welcomes editorial material, letters to the editor, and comments from its readers. No responsibility is assumed for accuracy of submissions.

Subscription to *Forth Dimensions* is included with membership in the Forth Interest Group at $45 per year ($53 Canada/Mexico, $60 overseas air). For membership, change of address, and to submit items for publication, the address is:

Forth Interest Group
100 Dolores Street, suite 183
Carmel, California 93923
Administrative offices:
831.37.FORTH Fax: 831.373.2845

The Forth Interest Group
The Forth Interest Group is the association of programmers, managers, and engineers who create practical, Forth-based solutions to real-world needs. FIG provides a climate of intellectual exchange and benefits intended to assist each of its members. Publications, conferences, seminars, telecommunications, and area chapter meetings are among its activities.

FORTH DIMENSIONS (ISSN 0884-0822) is published bimonthly for $45/53/60 per year by Forth Interest Group at 1340 Munras Avenue, Suite 314, Monterey CA 93940. Periodicals postage rates paid at Monterey CA and at additional mailing offices.

POSTMASTER: Send address changes to FORTH DIMENSIONS, 100 Dolores Street, Suite 183, Carmel CA 93923-8665.

High Accuracy Table Lookup Using Cubic Interpolation

This algorithm basically trades speed for table size by assuming that the line joining points in a lookup table is really a curve. The value in question rests on the curve between the two middle points of a four-point segment. The curve is assumed to be a third-degree polynomial that passes through all four points.

Intended for use on small processors, this code uses only integer arithmetic. I originally wrote it to calculate various transcendental functions to 16-bit precision. There are more efficient ways to approximate such functions, but the general-purpose method presented here lends itself to arbitrary functions, too.

The theory behind the algorithm is as follows:

Given points y0, y1, y2, and y3, there is a point f(x) between y1 and y2 where the region of interest is $0 \le x < 1$.

```
f(x) = w0 + w1 * x + w2 * x^2 + w3 * x^3
```

For four equally spaced points (n = -1,0,1,2), f(n) gives four equations:

```
f(-1) = y0 = w0 - w1 + w2 - w3
f(0)  = y1 = w0
f(1)  = y2 = w0 + w1 + w2 + w3
f(2)  = y3 = w0 + 2w1 + 4w2 + 8w3
```

Simultaneously solving these equations yields the following coefficients upon which the algorithm is based:

```
w0 = y1
w1 = (-2y0 - 3y1 + 6y2 -  y3) / 6
w2 = ( 3y0 - 6y1 + 3y2      ) / 6
w3 = ( -y0 + 3y1 - 3y2 +  y3) / 6
```

The word `CUBIC4` does the approximation using four data points at an address. `CUBIC` does some indexing and scaling in order to be useful in using a lookup table.

The algorithm takes some shortcuts to keep the math simple, so a wildly varying lookup table could cause an overflow. In typical applications, you won't come close to this situation,. but it always pays to test.

The example given here represents the first quadrant of a sine function using 19 data points. This gives better than 16-bit precision. An 80 point table gives a maximum error of about .004 PPM.

```
\ Table Lookup Using Cubic Interpolation

: d2*           2dup d+ ;        : d3*          2dup d2* d+ ;
: d4*           d2* d2* ;        : d5*          2dup d4* d+ ;
: d6*           d2* d3* ;        : d16*         d4* d4* ;

8 cells         constant cellbits  \ bits/cell assuming byte addressing
                                   \ change if your address units aren't
bytes

1 cellbits 1- lshift 0  2constant wround  \ i.e. 0x00008000 for 16-bit
Forth


variable wptr   \ points to the input data

: @seq          ( -- d )
\ get next point for coefficients ( write in assembly for speed )
                wptr @ @ s>d
                [ 1 cells ] literal wptr +! ;
```

Brad Eckert • Mesa, Arizona
bradbits@hotmail.com

The author is an engineer who designs and programs microprocessor-based things.

```
: w1            ( a -- n )              \ 6 * w1
                wptr ! 0.
                @seq d2* d-     @seq d3* d-
                @seq d6* d+     @seq d-         drop ;


: w2            ( a -- n )              \ 6 * w2
                wptr ! 0.
                @seq d3* d+     @seq d6* d-
                @seq d3* d+                     drop ;


: w3            ( a -- n )              \ 6 * w3
                wptr ! 0.
                @seq d-         @seq d3* d+
                @seq d3* d-     @seq d+         drop ;


: cterm         ( frac n1 n2 -- n3 )    \ n3 = n1 * frac + n2
                >r m* d2* wround d+ nip \ trunc --> round
                r> + ;


: cubic4        ( frac a -- n )         \ frac = 0..maxint
\ perform cubic interpolation on 4-cell table at a
                >r dup dup r@ w3        \ w3
                r@ w2      cterm        \ w3*f + w2
                r@ w1      cterm 6 /    \ (w3*n*n + w2*n + w1) / 6
                r> cell+ @ cterm ;      \ *n + y1


: tcubic        ( n1 addr -- n2 )
\ perform cubic interpolation on table at addr
\ n1 = 0..2^cellsize-1
                dup cell+ >r  @         ( n1 tablesize | addr )
                um*   >r 1 rshift r>    ( frac offset | addr )
                cells r> + cubic4 ;


: CUBIC         ( n1 span addr -- n2 )
\ perform cubic interpolation on table at addr, n1 = 0..span-1
                >r >r 0 swap r> um/mod nip
                r> tcubic ;


create exampletable                     \ Sine table (1st quadrant)
   16 ,                                 ( 16 points plus 3 endpoints )
-3212 ,     0 ,  3212 ,  6393 ,  9512 , 12540 , 15447 , 18205 ,
20788 , 23170 , 25330 , 27246 , 28899 , 30274 , 31357 , 32138 ,
32610 , 32767 , 32610 ,                 \ clipped to maxint for 16-bit 4ths

.( 32768*sin(10degrees) is )  10 90 ExampleTable CUBIC .
```

User Stacks in ANS Forth

Forth programmers are, of course, familiar with the concept of the information stack, since the data stack and return stack are at the heart of Forth. Here I would like to remind readers of the concept of a stack as an *abstract data type*. In this view, a stack is defined in terms of the things you can do with it, regardless of the implementation details that make those things possible. In this view, a stack is characterized as follows:

- You can put things on a stack.
- You can take things off a stack.
- The thing taken off is always the last thing put on.

Here we present words to create and manipulate stacks implemented as a linked list.

Figure One illustrates the principle of the linked list. The rectangles represent *nodes*—a number of contiguous memory locations. These blocks of memory do not have to be next to each other, nor must they all be of the same size, nor do they have to be in order (although any of these conditions may be imposed by an implementor in the name of performance efficiency, depending on the application).

The key idea is the existence of a *link field* (shown in Figure One at the left end of each node) that points from one node to the next. There is a separate *pointer* to the head of the list, and the pointer of the last node is a *null pointer*, pointing to nothing. In Forth, it is convenient to use zero as a null pointer, since it is easy to test for and there are few systems that would allow memory location zero to be the valid starting address of a link node. Variations on this theme include having pointers to other locations on the list, circular lists (where the last item points to the first item) and doubly linked lists (with pointers going in both directions).

Linked lists are important in the computer world because:

- they can be traversed almost as rapidly as accessing contiguous memory locations,
- items can be added or removed "on the fly," therefore,
- they use memory efficiently.

We now have a pretty good problem specification. We need Forth words to:

- create a user stack
- push items onto the user stack from the Forth data stack
- pop items off the user stack onto the Forth data stack

It would also be handy if, following a *pop*, performing a *push* restored the items to the user stack in the same order they had been (making `push` and `pop` reciprocal operations).

The accompanying code shows one way to do this.

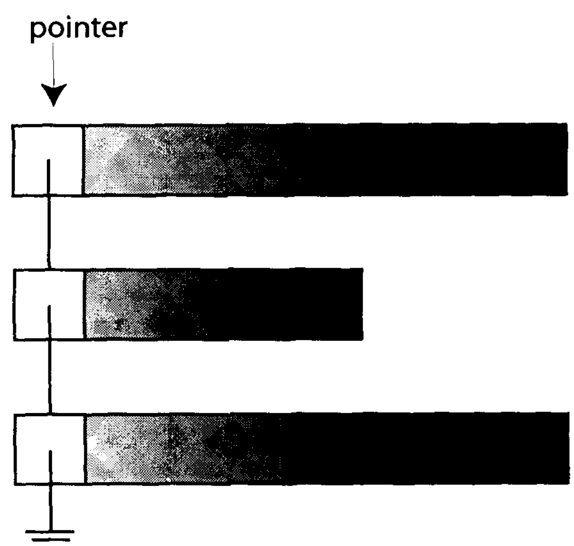
We have the defining word...

```
: stack  CREATE 0 , ;
```

Usage: `stack mystack` (creates a stack named `mystack`); then `mystack` puts the address of the pointer to the top of the user stack on the Forth data stack.

`CREATE` lays down the necessary header information for a new word in the Forth dictionary (itself often a rather complicated linked list or lists). `0 ,` gives the word a cell of data space and initializes it to the null pointer (since the stack is empty when created). When `mystack` is executed, its action will be to put the address of its cell of data space on the Forth data stack. Since that is all we need or want, there is no need for further action by a `DOES>` in this simple defining word.

Figure One



Now for push, which will create and populate a new node. We need a link field, which we will put first. This is a handy position, since the address of its cell will be the first node information available, and this way we can get at everything else with simple positive offsets. Since we want to be able to use variable-size nodes, the next cell will contain the node size, necessary overhead for this capability. The third cell will be the first of the cells containing the data of the node.

The size specification could be either the number of actual data cells or the actual node size, both data and overhead. My personal preference, implemented here, is to use the total node size. This means the programmer needs to remember to bump the size specification to the number of actual data cells plus two. Push will take items off the Forth data stack one by one and store them in the node in order, so

Leonard Zettel • Dearborn, Michigan
Zettel@acm.org

```
\ This is an ANS Forth Program requiring the Memory-Allocation word set
\ Words to handle a user-created stack as a linked list with nodes of arbitrary size.

: stack CREATE 0 , ;

: node_size ( node-addr -- node-size) CELL+ @ ;

: n! ( n1 .. nn addr n --) \ Store n1 to nn in consecutive cells
                           \ starting at addr.
  CELLS OVER + SWAP DO I ! 1 CELLS +LOOP ;

: n@ ( addr n -- n1 .. nn) \ Fetch n consecutive values starting at
                           \ addr + (wordsize)*(n-1) & leave them
                           \ on the stack.
  1- CELLS OVER + DO I @ -1 CELLS +LOOP ;

: node. ( addr —) \ Display the contents of the node at addr.
  DUP @ U. DUP CELL+ @ CELLS OVER + SWAP CELL+ DO I @ . 1 CELLS +LOOP ;

: list. ( ptr —) \ Display the contents of the stack pointed to by ptr.
  CR DUP @ 0= IF ." stack empty" DROP EXIT THEN
  CR BEGIN @ ?DUP WHILE DUP node. CR REPEAT ;

\ Thanks to Marcel Hendrix for noting that ALLOCATE works in address units.
: push ( nn .. n1 addr --) \ Push n1 .. nn onto the stack pointed to
                           \ by addr. nn is the node size in cells
  OVER >R R@ ( get_node)
  CELLS ALLOCATE           \ Get node space
  ABORT" push: ALLOCATE failed."
  >R DUP @                 \ Get address of node at the top of the node stack
  R@ ROT !                 \ Make new node top of stack
  R> R> n! ;               \ Store node contents.

: pop ( addr -- nn .. n1) \ Pop stack pointed to by addr, leaving
                          \ node values on the stack and freeing
                          \ the node space.
  DUP @ DUP 0= ABORT" Empty user stack."
  DUP @ ROT ! DUP >R
  CELL+ DUP @ 1- n@ R>
  FREE ABORT" pop: FREE failed" ;
```

the item deepest on the Forth data stack will be in the node's end position.

Looking at the code for push, OVER >R R@ parks a copy of the node size on the return stack. CELLS ALLOCATE gets a node-sized chunk of memory, ABORTing with an error message if for some reason it could not do so. >R parks the address of the new node on the return stack. DUP makes a copy of the address of the pointer to top-of-user-stack. @ puts the address of the current top-of-user-stack on the Forth data stack. R@ puts the address of the new node on the Forth data stack, ROT puts the address of the pointer-to-top-of-user-stack on top of it, and ! stores the new node address in pointer-to-top-of-user-stack. R> puts the address of the new node on the Forth data stack. The next R> puts the node size on top of it, and our word n! populates the node.

Since we mentioned it, let's take a look at n! (*n-store*). We need to store ***n*** items of information, each the size of a cell, in consecutive memory cells starting at *addr*. The obvious way to do this is with a DO ... LOOP. So let's see... we could set up the following:

```
0 DO SWAP OVER I CELLS + ! LOOP DROP ;
```

This would do it. 0 DO sets up the loop parameters. SWAP puts the next item to store on top of the stack. OVER puts a copy of the base address over it. I CELLS + gives the address the proper offset. ! stores the item. At the end of the loop, we are left with the base address on the stack, so we DROP it. Not too bad.

Can we do better? Suppose we could arrange it so that I furnished the storage address itself instead of a count. Then the business contents of the loop could simply be `I !`, but `I` would have to increase by the number of address units in a cell. We could do that with `1 CELLS +LOOP`. Okay, so far we have `DO I ! 1 CELLS +LOOP`. Then we notice we no longer need the `DROP`. All that's left is figuring out how to set up the proper `DO` range. The first value of `I` has to be *addr*. The last value of `I` used will be *addr* plus (*n*-1)*(address units in a cell). Given the rules governing `DO` loops, this means an upper limit of *addr* plus (*n*)*(address units in a cell), because with an increasing index, the iteration stops one pass short of the loop limit. We can get the required loop parameters with `CELLS OVER + SWAP`. `CELLS` switches us from number of cells to number of address units. `OVER +` gets the required upper limit. `SWAP` puts things in order for the following `DO`. At the cost of some preliminary setup work, we have reduced the number of words inside the loop (where most of the work will be done) from seven words to four, a fair savings.

Let's look at `pop`, the inverse operation of `push`. First we check that there is indeed something on the stack to pop, `ABORT"`ing if there isn't. Assuming we pass that test, the stack picture is now *(addr1 addr2)*, with *addr1* being the address of the pointer-to-top-of-stack, and *addr2* the address of the first cell of the top of stack. `DUP @` puts the pointer to the next item down (if any) on top of the data stack. `ROT !` makes the pointer-to-top-of stack point to that item, since that will be the new top of stack after the `pop` completes. `DUP >R` parks a copy of the address of the item to be popped on the return stack. `CELL+` bumps the address to the cell containing the size of the node. `DUP @ 1-` gives the parameters needed by `n@`, which puts the required information on the data stack. Finally, `R> FREE` gives the space occupied by the popped node back to the system, since the application no longer needs it. Doing this here means we don't have to worry further about garbage collection, which can be a headache. We'll let the system take care of that, since it should be more competent to do so.

`n@` follows the pattern of `n!` with some adjustments for circumstances. Here the index has to start at the high address and count down, thus the `-1 CELLS +LOOP`. The first address fetched will be at *addr* + (*n*-1)*(cell size in address units), so we have `1- CELLS OVER +`. Because this loop will be counting *down*, the final value of `I` in the loop will be the limit value, which we set to *addr*. So we see that what at first seems to be a Forth idiosyncrasy turns out to be nicely suited to the uses of zero-based addressing, where *n* items are indexed as *u*(0), *u*(1).. *u*(*n*-1) rather than *u*(1)... *u*(*n*).

At this point we have covered how to create a user stack, and how to push items on to it or pop them off. Another handy thing to do (perhaps while debugging an application) is list the contents of a stack. For this we have list. ("list-dot"). Given a pointer to a list, we look at the next pointer and, while it is non-null, we display the node contents with node. (which follows the same principles as n!) and then go on to the next item. node. has some complications that come from dealing with messy realities. Addresses in Forth can cover the full range of unsigned numbers, so the first cell is displayed using `U.` while the remaining values are diplayed with `.` (dot). This leads to some complications in setting up the `DO` parameters. We can still get the upper limit with **`DUP CELL+ @ CELLS OVER +`**, but since we have already displayed the contents of the first cell, we increment the **`DO`** starting value using **`CELL+`**.

Now that we have reviewed everything, let's try a simple example:

```
Stack mystack
```
...will create an empty user stack named `mystack`.

```
Mystack list
```
...will produce the message "stack empty," since we haven't put anything in it yet. So let's follow up with:

```
567 3 mystack push mystack list.
```
We should now see `0 3 567`. Now let's try:

```
mystack
pop mystack push
1009 885 234 5 mystack push
mystack list.
```

(On as many lines as you like, with as many uses of .S as you prefer). Using SwiftForth™ from FORTH, Inc. I saw:

```
22282240 5 234 885 1009
0 3 567
ok
```
...which is what I should have seen.

Forth-Gesellschaft eV (Germany's FIG) has changed the name and address of its web site. The new URL is:

http://www.forth-ev.de

For the benefit of those who do not read German, at press time, a translation of the whole site into English was in preparation.

The site's webmaster is Dr. Egmont Woitzel, member of the Board of Directors of Forth-Gesellschaft.

Up to now, the work put into the new site is entirely due to Dr. Egmont Woitzel and Professor Dr. Thomas Beierlein, both from the Directorial Board of Forth-Gesellschaft. Much additional work comes from Friederich Prinz, Editor of *Vierte Dimension* and Member of the Directorial Board.

Aspects of a Particular Three-Stack Machine Design

Abstract

H3sm ("Hohensee's 3-stack machine") is a demo implementation of a virtual computing machine with three distinctly featured stacks, plus a Size register controlling the data stack. The stacks are the Return Stack, the Pointer Stack, and the Data Stack. The Data Stack, the Size register and affiliated ALU and stack operators implement a fundamental type called a *pyte,* which is an integer at the current value of Size. Size varies from one to 256 eight-bit bytes. Pointers and return addresses and their respective stacks are address-bus cells, as usual. H3sm currently has only a vestigial interpreter and no interpretive threading (compiler) capability. The current H3sm does demonstrate pyte arithmetic.

GNU C source code for H3sm is at http://linux01.gwdg.de/~rhohen/H3sm.html, and is heavily commented.

H3sm and this essay are primarily the work of Rick (Richard Allen) Hohensee, with distinct improvements by Michael Somos (http://grail.cba.csuohio.edu/~somos/). Amongst other things, Somos generalized the code for either-endian hosts, which I did not intend to address myself.

Impetus

The idea of a three-stack "Forth" has been gnawing at me for several years. Around 1992, I attempted and failed to write a three-stacker on the Commodore 64. At the time, I thought a doubly linked dictionary was a good idea, and I ran out of steam trying to implement that in 6502 machine language. Jonah Thomas (JET) has since pointed out that I could do the things I wanted without double linking. The H3sm dictionary linking is fairly conventional in this regard, so, in true Forth style, JET must be credited for something that, thankfully, *isn't* in H3sm.

Several things about a conventional Forth bug me or just seem curious. The absence of microprocessor-style *conditional flags,* the plethora of size-typed operators, and the fact that I can never, to this moment, remember the order of operands to Forth `!` ("store"). I have hoped that three stacks can make a useful distinction between data and pointers, which will solve my little `!` problem, and will also provide some reduction in the namespace-explosion that is one of Forth's weak points.

Also curious is what I see as the missing coda to Phil Koopman's *Stack Machines: The New Wave.* This book describes the history of computing engines in terms of the number of stacks they have. Koopman points out that stacks like the typical CPU return stack and the Forth parameter stack are implicit to the instruction sets of their respective machines, and are not addressed, as registers are on machines with multiple similar registers. Koopman shows that computers have improved noticeably as they went from zero, to one, and then to two stacks. However, I don't recall much conjecture in the book on more than two stacks, or any compelling case for two being the absolute upper limit.

H3sm therefore begs to beg the question Koopman begs. Well then, many have said, why not 1024 stacks, or whatever? Because, if they're all the same, you wind up with wasteful addressing bits in the opcodes again. The key lies in the fact that with a small number like three, each "stack" can have properties distinct from the others. With two, you don't have much flexibility. With three, data items can be different in size than addresses. Variably sized, in fact. (Koopman's book is on the web in its entirety, by the way.)

Looking at machine design very subjectively, a Forth is a nice little assortment of data structures/mechanisms. Forth's openness and simplicity allows re-use of its component parts. H3sm adds a couple of distinct parts to the toolkit. An H3sm models a machine with an address bus of typical size, and may help abstract the size of the data bus over a wide range of possible sizes.

The name *pyte* originally was from "precision byte," and *Size* originally was called "Precision." My technical background is in the field operations of land surveying, where one develops a mindset in which numbers are *duals,* with a unit and a precision. I've wanted a computer that can change itself from a low-precision implement to a high-precision implement—such as from a surveyor's manual "Chinese Ninety" or an artist's outstretched thumb, to a first-order triangulation theodolite—with the change of one variable.

Design

Each of the three H3sm "stacks" has a behavior that is distinctly different from the other two.

return

The return stack is rather typical, containing address-size execution tokens. One day, we might do loop indices and such on the return stack, too.

pointer

The H3sm pointer stack is address-cell sized. The pointer stack is "sluggish"; it is not auto-pop/push. The pointer stack pointer is usually left pointing to the recently referenced cell.

data

The data stack operates on pytes, groups of 1, 2, 4, 8, 16...256 bytes. Boolean flags are the low byte of a pyte. False is zero. Non-zero is true. The H3sm `true` word asserts 255 in a flagpyte.

Size "register"

The current effective size of data stack operands is the Size state variable. There are user-visible accessors of this Size "reg-

Rick Hohensee • Adelphi, Maryland
humbubba@smarty.smart.net • rickh@capaccess.org
Download: http://linux01.gwdg.de/~rhohen/H3sm.html

Rick Hohensee promotes cLIeNUX Linux/GNU/etc., his complete Forth-influenced Unix environment. He is a semi-professional rock musician with a background in surveying and construction estimating.

ister." Operations on pytes are in terms of Size, except where a pyte is treated as a flag, *aka* a *flagpyte*.

So the three stacks are the data typing of H3sm; typing is enforced by the various operands. The data stack is where a datum can be treated arbitrarily. There are a few ops to move things from stack to stack, with some conversion and data loss in some cases, as may be necessary between pytes and addresses, and to and from Size.

The above data structures are defined by their interaction with the H3sm primitives in Table One [see following page]. (I kinda like the term *atoms* in lieu of the usual *primitives*, by the way.) It is messy, but not huge. I count 97 words. These atoms were more than sufficient to write the simple interpreter. The interpreter is about 20 non-atomic words, written as (C-compiled-in) threads of the atoms. Glaring omissions include `-`, `*`, `*/`, and `move`. Available flow-control is rudimentary. Conversely, there's about a dozen scaffolding constants and so on that could easily be done without. Note that, in exchange for things like `p+s` and so on, we don't have any of the likes of `2+`, `2DUP`, et al.

The functionality of the above atoms may be more than you think at first glance. The math and logic that does exist works at any `Size` from 1 to 256 bytes. Fairly rich pointer-twiddling is also available. I would describe this as "thicker" than a Forth. A quick session with some pyte arithmetic may illustrate some of this thickness. `r` is the register picture word. Numbers are hex.

Listine One. Sample session with pyte arithmetic.

```
(the next 2 lines are my florid shell prompt, with input of "H3sm")
$ cLIeNUX0 /dev/tty3 r 00:30:15   /mount/bl/H3sm
$H3sm
total Virtual Address Space including dictionary is 65536 bytes.
actual address of VAS is 0xbffe5d2c

gcc-compiled at 22:37:38 on Dec 28 1998

latest  bffe8674
r                                         (this is our H3sm input line, r )
    RETURN     POINTER      DATA   pyte Size = 4
       a34           0      msB, lower bytes --->
       a50           0      00 00 00 00   T.O.D.S.
         0           0      00 00 00 02
         0           0      00 00 00 00
         0           0      00 00 00 00
         0           0      00 00 00 00
       rsl= 2     psl= 0    dsl = 0 = lsB of TOS  ip = 2520

O-TAY!
44444444  66666666  10101010  r           (more input, 3 #'s and another r)
    RETURN     POINTER      DATA   pyte Size = 4
       a34           0      msB, lower bytes --->
       a50           0      10 10 10 10   T.O.D.S.
         0           0      66 66 66 66
         0           0      44 44 44 44
         0           0      00 00 00 00
         0           0      00 00 00 02
       rsl= 2     psl= 0    dsl = 12 = lsB of TOS  ip = 2520

O-TAY!
2222 + r                                  (etc.)
    RETURN     POINTER      DATA   pyte Size = 4
       a34           0      msB, lower bytes --->
       a50           0      10 10 32 32   T.O.D.S.
         0           0      66 66 66 66
         0           0      44 44 44 44
         0           0      00 00 00 00
         0           0      00 00 00 02
```

Table One.

(P; begins a pointer stack comment.
(R; is a Return Stack comment.
(is a Data Stack comment.
||| means "below (left of) here is required but not changed."
HNC is Head Name Cell of a dictionary word.

Atom name	Stacks effects	Size effects, comments
address	(P; --- ptr)	
AND	(pytea pyteb --- pyteaandb)	
bytemask	(--- 0xff)	
dualmask	(--- 0xffff)	might be handy for Unicode
		1 !SIZE
call	(R; --- xt)	
cells		4 !SIZE
aint	(flagp --- !flagp)	the NOT of a flag
bump	(--- junk)	
bye		return to caller of H3sm
		with flag byte of TOS
charsize		1 !SIZE (a cheat)
doHNC	(P; HNIC ---) (R; --- RET|null)	Forth EXECUTE
downsize		shift Size down, or to one !SIZE
drop	(pyte ---)	
dup	(pytea ||| --- pytea)	
ell		unconditional branch
!p	(P; p store |||)	store a ptr
extend		
emit	(pyte ---)	treated as a char
false	(--- 0flag)	
@	(P; ptr ||| ---) (--- pyte)	
@size	(P; noun --- noun)	!SIZE
flag	(pyte --- flagpyte)	bytewise OR a pyte into its low byte
four	(--- 4)	pyte constant
gap	(--- ptra-b) (P; ptra ptrb ||| ---)	
hostfn	(--- sh.ret.val) (P; epa bpa |||)	
!BUFFER0		
max	(a b --- maxab)	
NOT	(pyte --- !pyte)	
negate	(a --- 2's_complement_negative_a)	
last	(P; --- count.byte.addr)	
literal	(--- pyte)	!SIZE
-p	(pyte ---) (P; ptr --- ptr-intpartofpyte)	
no	(flagpyte ---)	conditional branch if true
nothing		NOP
nown	(P; --- nown_body)	
ones	(--- -1)	or fffff
one	(--- 1)	pyte constant
OR	(pytea pyteb --- pyteaORb)	
over	(a b --- a b a)	
pdrop	(P; ptr ---)	decr pointer stack lubber
pdup	(P; ptra --- ptra ptra)	
period	(--- 46)	ASCII . pyte constant
p@	(P; ptr1 --- ptr2)	ptr1 overwritten
+	(a b --- c)	
+p	(pyte ---) (P; ptr ---ptr+bytepartpyte)	

(Table continues on next page.)

Atom name	Stacks effects	Size effects, comments
p-s	(P; ptr --- ptr-Size)	
p+s	(P; ptr --- ptr+Size)	
p+b	(P; ptr --- ptr+1)	
p+c	(P; ptr --- ptr+4)	
p-c	(P; ptr --- ptr+4)	
p-c	(P; ptr --- ptr+4)	
pTOs	(P; Size |||)	!Size
sTOp	(P; --- Size)	
p!	(P; store p ||| --- store p)	
pswap	(P; a b --- b a)	
p>r	(P; ptr |||) (R; --- ptr)	
pUP	(P; --- oldptr)	
push	(R; --- ip)	
?=	(a b --- flagpyte)	
rdrop	(R; a ---)	
return	(R; xt ---)	
rpcopy	(R; a |||) (P; --- a)	dup r to p
r>p	(R; ptr ---) (P; --- ptr)	
r>s	(R; size ---)	!SIZE
saveDictionary		
sign	(pyte --- 1 or 254 or 0)	
sixteen	(--- 16)	pyte constant, decimal 16
sized	(P; --- ptr)	
size	(--- Size)	
s>r	(R; --- Size)	
space	(--- 32pyte)	pyte constant for a space
!	(P; ptr |||) (pyte ---)	
swap	(pytea pyteb --- pyteb pytea)	
ten	(--- 10)	pyte constant, decimal 10
three	(--- 1)	pyte constant
time	(--- utime.int)	4 !SIZE
TOcode	(P; HNC --- Code_Body_Cell)	
TOlast	(P; ptr ---)	update latest/last
TOlink	(P; HNC --- Link_Cell)	
>s	(size ---)	!SIZE
true	(--- true_flagpyte)	
two	(--- 2) pyte constant	
ushift	(shiftee amount --- shifted)	
upsize		!SIZE
vasbase	(P; --- addr.of.vas.x[0])	implementation requirement
wait	(P; bpa ||| --- epa)	blocks flow
0sl	(what ever ... ---)	
0psl	(P; what ever ... ---)	
0fsl	(R; what ever ... ---)	
XOR	(pytea pyteb --- pyteaXORb)	
yes	(flagpyte ---)	conditional branch if false.
zero	(--- 0)	0 as a pyte constant
ok		
r		machine language-monitor-style stack pic
tdump	(P; text ||| ---)	

```
      rsl= 2      psl= 0     dsl = 12 = lsB of TOS  ip = 2520

O-TAY!
2 TOsize + r
    RETURN      POINTER        DATA    pyte Size = 2
       a34            0        msB, lower bytes --->
       a50            0        42 42   T.O.D.S.
         0            0        66 66
         0            0        66 66
         0            0        44 44
         0            0        44 44
       rsl= 2      psl= 0      dsl = 12 = lsB of TOS  ip = 2520

O-TAY!
8 TOsize dup r
    RETURN      POINTER        DATA    pyte Size = 8
       a34            0        msB, lower bytes --->
       a50            0        42 42 66 66 66 66 44 44   T.O.D.S.
         0            0        42 42 66 66 66 66 44 44
         0            0        44 44 00 00 00 00 00 00
         0            0        00 02 00 00 00 00 00 00
         0            0        00 00 00 00 00 00 00 00
       rsl= 2      psl= 0      dsl = 14 = lsB of TOS  ip = 2520

O-TAY!
bye
$ cLIeNUX0 /dev/tty3 r 00:36:44   /mount/bl/H3sm
$
```

In the above, we did + at two and four bytes, and `dup` at eight bytes, to the pyte. This is *operator vectoring*, not overloading. There is no interpreting involved. I'm told that bigger adds are slow in silicon, without lots of extra silicon, but wide Booleans could be a big win in a relatively small amount of silicon if you have a use for them. More important may be the semantic freedom to design an algorithm for pytes, and to use it for whatever size data is appropriate at any particular moment. The H3sm interpreter is very nearly Unicode-transparent for this reason, although there are one or two charsize assumptions in the current code.

Implementation

For those who don't care to browse the source, H3sm is a rather nasty piece of C. H3sm is distinctly not what C likes to do. My interest in C in this context is simply as a portable assembler, and the code reflects that intent. As little as possible of C's sophistication is used. All of H3sm is in a single C function, *main()*. H3sm uses GNU C labels-as-values, in pure mimicry of Gforth. This is a GNU extension to C that is a form of computed GOTO, and is in H3sm's `NEXT` macro; i.e., it is essential to H3sm's threading scheme. I suspect H3sm's threading scheme, which I call Virtual Machine Subroutine Threading, has a unique aspect. It is similar to what has been called "call threading" (by Ertl or Paysan, I think, in comp.lang.forth). However, H3sm has no `W`, no "Working Register." An *atomic bit* is necessary in the headers of atoms (primitives) to distinguish them from threads. The resultant threading behavior is slightly less confusing to me, resembling normal subroutine calls a bit more than most other schemes.

One possible benefit of this cretinous C style is simple embeddability. It is trivial to rename *main()* and include H3sm in something else. This doesn't give any communication between H3sm and the host code, however. A bootable version of H3sm should not be terribly difficult, either, and perhaps would be more interesting than an embedded one.

The interpreter and pre-threader compiled-in threads in H3sm are very wasteful code, both in terms of code and memory used by the executable, which, at about 90K, is too big for such a simple program. Mercifully, that stuff only runs at startup. There was value in doing them the way I did, though, because the `cpp` macros served as a preview of the language and of what it would be like to program it from the interpreter.

In C, `Size` can be any integer between 1 and 256 inclusive. In silicon, `Size` may better work as 1, 2, 4, 8, etc. Maybe not. As it is, a `Size` that can match, e.g., Intel floating-point stack item sizes, is a happy accident of this implementation.

Loop indices are pytes. Pytes are relatively worse performance-wise in C than they would be in silicon, and will be the next thing I change in H3sm. I'm quite pleased that H3sm loops benchmark at about half the speed of Perl, for such a fragile demo, but with `int` loop indices on the return stack she should run distinctly more like a Forth.

Impressions

Well, *I* like it. I think it was worth doing. I see some possibility for pytes to reduce the "What is an int?" problems

Three-Stack Machine continues on page 67.

Polyalphabetic Encryption Cracker

The Polysub — well-known but not very secure

In this article, we will present the Polyalphabetic Substitution Cipher (the "PolySub"), which most readers should be familiar with. This is the one in which a key is repeatedly XOR'd with the plaintext to produce the ciphertext (or vice-versa). We will then present a program which will crack this cipher, working on the assumption that the plaintext characters have varying frequencies and that one plaintext character in particular is much more frequent than the others (we assume that this character is the blank). This article is oriented toward novices, so we provide a lot of implementation-level description of our encryption-cracking program.

On computers, the PolySub is usually implemented with XOR. This allows the same program to be used for both encryption and decryption, since XOR undoes itself. In pre-computer days, plus and minus were used. The PolySub was invented in 1568 by Leon Battista and was used extensively by the Union Army during the American Civil War.

Let's look at an example of the PolySub using plus and minus. We will use an alphabet of all capitals encoded 0 through 26 (a blank is a zero), and we will use a key of "DOOR" (see Figure One).

We are using modular addition to encrypt. The message would be decrypted by using modular subtraction. This method is an excellent system for anybody who doesn't have access to an electronic computer. The reason is that one can easily construct a "computer" out of cardboard.

The idea is to have a circular piece of cardboard riveted, through its center, to another piece of cardboard so that it can be spun freely. Both wheels have the alphabet written on them clockwise. To encrypt or decrypt, one locates the current letter from the key stream on the inner wheel and lines it up with the blank on the outer wheel.

If encrypting, one then locates the current message letter on the outer wheel and finds the corresponding encryption letter on the inner wheel. If decrypting, one would locate the current encrypted letter on the inner wheel and find the corresponding message letter on the outer wheel. Note that the famous Julius Caesar encryption scheme (adding three to every letter) is just a degenerative form of the Plus-Minus scheme. It has only a single character long key (the "C"). The Julius Caesar scheme is a Monoalphabetic Substitution cipher.

The PolySub's security can be enhanced a little bit by having a long key. This is best accomplished by repeatedly encrypting the message. For example, if your key is "DOORFENCE" the key length is nine. On the other hand, if the message is first encrypted with "DOOR" and then with "FENCE", the effect is the same as if it was encrypted once with a twenty-character (4*5) key of "JTCUIUTFGTUWRRTXICRW". The security is actually a little better, because the effective twenty-character key is a jumble of characters and can't be as easily guessed as "DOORFENCE" which is composed of recognizable English words. When doing multiple encryptions like this, one should be sure that none of the key lengths have common denominators. If they are the same lengths, for example "DOOR" and "GATE", it is still effectively a four-character key (although the characters at least are jumbled as "KPIW").

The first part of the CrakPoly program is the code to load and save files, and to encrypt and decrypt them. After that, we get into cracking ciphers for which we don't have a key. There are two phases to cracking the PolySub: the first is determination of the key length, and the second is determination of the key contents.

Preliminary Code — encrypting and decrypting files

Our PolySub cracking program is called "CrakPoly.scr" and is written in UR/Forth from Laboratory Microsystems, Inc. The source code is in Figure Two. CrakPoly should run under any Forth-83 compiler. It has been tested under both 32-bit and 16-bit UR/Forth. The reader is encouraged to put `QI` (provided on screen 5) inside various words as an aid to dissecting the program. Execution will stop and the user can examine the contents of variables before continuing with the program.

We have two data buffers, `CIPHERTEXT` and `PLAINTEXT`. These each have `FILE_SIZE` bytes of memory allocated to them. `FILE_SIZE` is defined in screen 1 and is currently set quite small, so readers with eight-bit computers can load and run the program. Readers with 32-bit computers should set `FILE_SIZE` larger.

The word `INPUT_FILE` in screen 17 is used to load a file

Figure One.

```
MEATLOAF FOR DINNER <-- the plaintext (unencrypted) message
DOORDOORDOORDOORDOO <-- the key stream
-------------------
QTPKPCPXDUCIDSXERTF <-- the ciphertext (encrypted) message

Below is the same thing in numerics.

13 05 01 20 12 15 01 06 00 06 15 18 00 04 09 14 14 05 18
04 15 15 18 04 15 15 18 04 15 15 18 04 15 15 18 04 15 15
---------------------------------------------------------
17 20 16 11 16 03 16 24 04 21 03 09 04 19 24 05 18 20 06
```

Hugh Aguilar • vaguilar@dancris.com

Figure Two.

```
Screen # 0
\ CRAKPOLY                                              19:36 05-29-99

Crack the polyalphabetic substitution cipher (XOR).
written by Hugh Aguilar
January/February/March/April 1999 Forth Dimensions
```

Screen # 1

```
\ word size arithmetic CHARS MOSTEST                    20:33 05-30-99

WSIZE CONSTANT W    \ less cumbersome to type

\ these depend upon having a 32-bit system
: W+     4 + ;
: W-     4 - ;
: W*     2* 2* ;
: W/     2/ 2/ ;

256 CONSTANT CHARS

CREATE MOSTEST 0 ,  BL MOSTEST C!
\ most frequent plain char

5000 CONSTANT FILE_SIZE     \ maximum file size
```

Screen # 2

```
\ LOW_ENCRYPT LOW_DECRYPT for Plus-Minus system  11:39 05-31-99

\\ Plus-Minus system

: LOW_ENCRYPT  \ plain_char key_char -- cipher_char
    +  DUP CHARS >= IF  CHARS -  THEN ;

: LOW_DECRYPT  \ cipher_char key_char -- plain_char
    -  DUP 0< IF  CHARS +  THEN ;
```

Screen # 3

```
\ LOW_ENCRYPT LOW_DECRYPT for Minus-Plus system  20:12 05-30-99

\\ Minus-Plus system

: LOW_ENCRYPT  \ plain_char key_char -- cipher_char
    -  DUP 0< IF  CHARS +  THEN ;

: LOW_DECRYPT  \ cipher_char key_char -- plain_char
    +  DUP CHARS >= IF  CHARS -  THEN ;
```

Screen # 4

```
\ LOW_ENCRYPT LOW_DECRYPT for XOR system         11:39 05-31-99

 \ XOR system

: LOW_ENCRYPT  \ plain_char key_char -- cipher_char
    XOR ;

: LOW_DECRYPT  \ cipher_char key_char -- plain_char
    XOR ;
```

into memory. It takes two parameters, the filename and the buffer pointer. The filename should be the address of a counted string containing the fully qualified filename. The buffer pointer should be either CIPHERTEXT or PLAINTEXT. OUTPUT_FILE is also in screen 17 and also takes a filename and a buffer pointer, but it outputs the contents of the buffer to the file.

If there is a document in PLAINTEXT, executing the word ENCRYPT in screen 14 will fill CIPHERTEXT with the encrypted version of the document. Executing the word DECRYPT, which is also in screen 14, will decrypt the document in CIPHERTEXT and fill PLAINTEXT with the unencrypted version.

Note that ENCRYPT and DECRYPT use the words LOW_ENCRYPT and LOW_DECRYPT which are defined in screen 4. These words in screen 4 are for the XOR PolySub. We also have versions of LOW_ENCRYPT and LOW_DECRYPT in screens 2 and 3. Both of these screens are commented out. Screen 2 is for the Plus-Minus PolySub, and screen 3 is for the Minus-Plus PolySub. If the reader is using either of these kinds of PolySub, he should comment out screen 4 and compile screen 2 or 3, instead.

Phase 1. — Determining key length

In order to determine the key's length, we need to assume that the characters in the plaintext are of varying frequencies. We don't care which characters are more frequent than the others or how they are distributed, so long as they aren't rectangularly distributed. We will repeatedly shift the ciphertext over and compare it against the original unshifted version of the ciphertext. We count how many of the characters being compared are equal to the character they line up against in the unshifted version.

We have an array called COINCIDENCES. The first index is the count of coincidences for ciphertext being shifted over by one character, the second index is the count of coincidences for ciphertext being shifted over by two characters, and so forth. COUNT_COINCIDENCES in screen 18 counts these coincidenses. COINCIDENCES actually contains percentages, rather than raw counts, because a different number of comparisons is done by each call to COUNT_COINCIDENCES. Our percentages have two decimal digits to the right of the decimal point.

FILL_COINCIDENCES in screen 19 calls COUNT_COINCIDENCES repeatedly and fills the COINCIDENCES array. Note that we have a word called SEARCH_SIZE which determines how many shifts we do. If our file is small, we only do a third of the total. This is because the more we shift, the less accuracy we have. If we did the entire file size, our numbers toward the end would be garbage and would only mess us up. Note that, the way the author originally had COUNT_COINCIDENCES written, it would rotate ciphertext around such that the characters at the end of the file would be compared to the characters at the beginning. In this way, we wouldn't get decreasing accuracy with increased shifts. This turned out to be a bad idea, because it caused coincidences to get counted more than once, which tended to smooth out the numbers.

Screen 21 contains the word SHOW_COINCIDENCES which uses these words to show what is in the COINCIDENCES array. If the reader uses SHOW_COINCIDENCES to look at COINCIDENCES, he should see there are spikes in the values. These spikes occur on multiples of the length of the key used to encrypt ciphertext. By eyeballing COINCIDENCES, it is fairly easy for the user to determine the key length.

We want our program to determine this automatically, however. There is some difficulty in this, because it is not clear what threshold a value must be over to be considered a spike. This threshold value varies with the data. Also, no matter how carefully the threshold value is set, some values which are spikes don't go over it, and some which aren't do go over it. There is a lot of variance in the data, especially when cracking small files.

We set our threshold to the midpoint of the data in COINCIDENCES. This is done by CALC_THRESHOLD in screen 22. The author originally tried using a constant value of 4%. This didn't work, because the threshold is at different heights, depending upon the length of the key. The author then tried using the average. This didn't work either; it was way too small, especially when the key length was large, and we got a lot of false spikes. The next attempt was to use the average plus the standard deviation multiplied by some empirically chosen

```
Screen # 5
\ miscellaneous words                                    12:03 05-30-99

: #?  \ d -- new_d  \ used in <# ... #> for the most sig digits
   2DUP D0= IF  BL HOLD  ELSE  #  THEN ;

: #_  \ d -- new_d  \ used in <# ... #> for the most sig digits
   2DUP D0= IF            ELSE  #  THEN ;

: QI  \ --
   QUERY INTERPRET ;

: ROVER  \ a b c -- a b c a  \ "rot over"
   2 PICK ;

: ZERO  \ adr --  \ zeros out the word at ADR
   0  SWAP ! ;


Screen # 6
\ miscellaneous words                                    13:23 05-31-99

: U>=  \ a b -- flag
   U< 0= ;

: INC  \ adr --  \ increments the value
   1  SWAP +! ;

: P_ALLOT  \ --  \ allots enough that HERE is paragraph aligned
   HERE 16 MOD  ?DUP IF  16 SWAP - ALLOT  THEN ;

: PCREATE  \ allotment --  \name  \ paragraph aligned CREATE
   P_ALLOT  HERE >R  ALLOT  R> CONSTANT ;

\ Don't use PCREATE in conjunction with DOES>.


Screen # 7
\ CARRAY WARRAY                                          19:39 05-30-99
\ Note that "base_adr" means the address provided by DOES>

: CARRAY  \ size --  \name       \ paragraph aligned char array
   CREATE  HERE >R  0 ,  P_ALLOT  HERE R> !  ALLOT
   DOES>  \ index base_adr -- address
       @ + ;

: WARRAY  \ size --  \name                   \ word array
   CREATE  W* ALLOT
   DOES>  \ index base_adr -- address
       SWAP W* + ;


Screen # 8
\ 2CARRAY WITHIN                                         19:39 05-30-99
\ Note that "base_adr" means the address provided by DOES>

: 2CARRAY  \ horz_size vert_size --  \name  \ 2D char array
   CREATE  OVER ,  DUP ,  * ALLOT
   DOES>  \ horz_index vert_index base_adr -- address
       DUP W+ W+ >R     \ return: data_adr --
       @                \ horz_index vert_index horz_size --
       *  +  R> + ;

: WITHIN  \ char lowest highest -- flag
```

```
   >R >R
   DUP R> >=  SWAP R> <=  AND ;


Screen # 9
\ PRINTABLE NUMERIC SPANISH                       13:23 05-31-99

: PRINTABLE  \ char -- flag
   32 127 WITHIN ;

: NUMERIC  \ char -- flag
   ASCII 0  ASCII 9  WITHIN ;

: SPANISH  \ char -- flag  \ accented chars and upside-down ? !
   >R             R@ 129 =        R@ 130 = OR
   R@ 144 = OR    R@ 160 = OR     R@ 161 = OR
   R@ 162 = OR    R@ 163 = OR     R@ 164 = OR
   R@ 165 = OR    R@ 168 = OR     R> 173 = OR ;


\ These are char_kind filter words.

Screen # 10
\ UPPERCASE ALPHA ALPHANUMERIC PUNCTUATION        13:23 05-31-99
: UPPERCASE  \ char -- flag
   ASCII A  ASCII Z  WITHIN ;

: LOWERCASE  \ char -- flag
   ASCII a  ASCII z  WITHIN ;

: ALPHA  \ char -- flag
   DUP UPPERCASE  SWAP LOWERCASE  OR ;

: ALPHANUMERIC  \ char -- flag
   DUP ALPHA  SWAP NUMERIC  OR ;

: PUNCTUATION  \ char -- flag  \ also includes the blank
   DUP ALPHANUMERIC 0=  SWAP PRINTABLE  AND ;
\ These are char_kind filter words.


Screen # 11
\ constants and variables                         20:36 05-30-99
100 CONSTANT KEY_SIZE
KEY_SIZE CARRAY KEY_STRING
KEY_SIZE WARRAY KEY_LENGTHS     VARIABLE BIG_KEY_LENGTHS
KEY_SIZE CHARS 2CARRAY KEY_CHAR
VARIABLE KEY_LENGTH            \ actual key size

FILE_SIZE PCREATE CIPHERTEXT
FILE_SIZE PCREATE PLAINTEXT
VARIABLE FILE_MORE             \ where we try to put more of file
VARIABLE FILE_LENGTH           \ actual file size
VARIABLE PAST_CIPHER           \ ptr past valid data in CIPHERTEXT

250 CONSTANT NON_CHAR          \ print this for nonprintable chars
16 CONSTANT DUMP_WIDTH         \ horizontal chars in DUMP display
18 CONSTANT SHOW_KEYS          \ keys shown by SHOW_KEY


Screen # 12
\ constants and variables DOSINT FILE1            12:06 05-30-99
300 CONSTANT MAX_SEARCH_SIZE
MAX_SEARCH_SIZE WARRAY COINCIDENCES
```

constant. For example, a constant of .68 will result in 75% of the values being under the threshold. This worked better, but it was overly complicated and still not good enough.

The midpoint worked best and was very simple. We have spikes clustered around some high value and non-spikes clustered around some low value. There are more non-spikes than spikes, especially when the key length is long, and this is what was messing us up when we were using the average. This disparity was what we were trying to compensate for with the standard deviation. By using the midpoint, we avoid concerning ourselves with how many spikes there are, relative to the number of non-spikes. The midpoint draws a line betwean the highest value and the lowest value, and this line pretty much separates the spikes from the non-spikes. CALC_THRESHOLD doesn't have to be perfect, because the KEY_LENGTHS array, described next, smooths over errors caused by values being seen as spikes when they are non-spikes, and vice-versa (as long as there aren't too many errors).

We have an array called KEY_LENGTHS as big as our maximum key size, and which we will fill with percentage probabilites of the key being any particular length. We have to do this because there is no way to be absolutely sure of the key length, due to the variance mentioned earlier. FILL_KEY_LENGTHS in screen 23 fills this array. This word calculates the distances betwean the spikes. If all these distances were the same, we would know for sure that this distance was the key length. They usually aren't, so we just count the times we see the different distances.

These counts go in KEY_LENGTHS. KEY_LENGTHS% in screen 24 converts these counts into percentages. This is mostly for aesthetic purposes when displaying them later; CALC_KEY_LENGTH doesn't need it done. We also have a variable called BIG_KEY_LENGTHS which counts any spike distances which are too big to fit in KEY_LENGTHS. Hopefully, this will be zero.

CALC_KEY_LENGTH calculates the actual key length. First it fills KEY_LENGTHS, then it searches through KEY_LENGTHS for the biggest value. The index to this value is our key length. If we have two or more values which are equal, we go with the smallest index. In almost all cases when this happens, the higher index is a multiple of the lower one. The smallest is the actual key length (otherwise, we would have a key which was some string repeated some number of times).

Screen 25 contains FILL_KEY_LENGTH, which does everything needed to determine the key length. This is the word the user will type at the keyboard in order to do phase one of the program. Note that, if the user disagrees with the program's idea of what the key length is, he can use KEY_LENGTH! to set it manually. FILL_KEY_LENGTH displays the front portion of COINCIDENCES at the top of the screen. This raw data is only marginally useful. FILL_KEY_LENGTH displays KEY_LENGTHS at the bottom of the screen. The user can see here what the probabilities of the various key lengths are. These are a guide for what to give KEY_LENGTH! if the user disagrees with what the program found to be the most likely. In practice, this is rarely needed; FILL_KEY_LENGTH is almost always correct.

Phase 2. — Determining key contents

We are ready for phase two, determination of what the contents of the key are. The individual characters of the key are solved for as if they were of distinct Mono-alphabetic ciphers. The second phase of the program the author found to be more straightforward than the first phase. It is all downhill from here!

In screen 1, we have a variable called MOSTEST which contains the plain character we think will be the most frequently occuring. This defaults to the blank. This value is not normally changed during the program's execution. It is made a variable rather than a constant, however, because the user may want to change it if he is decrypting some file which is not text. This change can be made without having to recompile the program. Note that, sometimes even in English text, the blank is not the most frequent character. Consider Figure Three, in which 'e' is the most frequent.

Figure Three.

Elephants are very eloquent.
Especially Penelope!

The program will still successfully crack ciphers like this. The text file for this article has 1.74 times as many blanks as 'e' characters. The ratio might

```
10000 CONSTANT UNITY  \ multiplier for percents
\ percents with two digits to right of decimal point

VARIABLE THRESHOLD  \ height to be considered a spike
CHARS WARRAY FREQS  \ count of encryption results

VARIABLE 'LOW_ENCRYPT   \ vector to LOW_ENCRYPT or LOW_DECRYPT
VARIABLE 'CHAR_KIND     \ vector to char kind checking word
DOSINT
0 CONSTANT READ_ONLY
1 CONSTANT WRITE_ONLY
2 CONSTANT READ_WRITE
HCB FILE1  \ handle control block
```

Screen # 13

```
\ <ENCRYPT>                                       11:05 05-27-99

VARIABLE SRC  \ either CIPHERTEXT or PLAINTEXT
VARIABLE DST  \ either CIPHERTEXT or PLAINTEXT

: ADVANCE_KEY_INDEX  \ key_index -- new_key_index
   1+  DUP KEY_LENGTH @ = IF  DROP 0  THEN ;

: <ENCRYPT>  \ source dest --  \ either CIPHERTEXT or PLAINTEXT
   DST !  SRC !  0  \ key_index --
   FILE_LENGTH @  0  DO
      SRC @ I + C@  OVER KEY_STRING C@  'LOW_ENCRYPT PERFORM
      DST @ I + C!
      ADVANCE_KEY_INDEX  LOOP
   DROP ;
```

Screen # 14

```
\ ENCRYPT DECRYPT GET_KEY                         20:14 05-30-99

: ENCRYPT  \ --
   [ '] LOW_ENCRYPT  'LOW_ENCRYPT  !
   CIPHERTEXT FILE_SIZE ERASE
   PLAINTEXT CIPHERTEXT <ENCRYPT> ;

: DECRYPT  \ --
   [ '] LOW_DECRYPT  'LOW_ENCRYPT  !
   PLAINTEXT FILE_SIZE ERASE
   CIPHERTEXT PLAINTEXT <ENCRYPT> ;

: GET_KEY  \ cipher_char plain_char -- key_char
   LOW_DECRYPT ;
```

Screen # 15

```
\ KEY_LENGTH! KEY_STRING! SHOW_KEY_STRING         20:14 05-30-99

: KEY_LENGTH!  \ key_length --
   DUP KEY_SIZE > ABORT" too long of a key"
   KEY_LENGTH ! ;

: KEY_STRING!  \ counted_string --
   COUNT  DUP KEY_LENGTH!
   0 DO
      DUP C@  I KEY_STRING C!
      1+ LOOP
   DROP ;
```

```
: SHOW_KEY_STRING   \ --
    0 KEY_STRING  KEY_LENGTH @  DUMP ;


Screen # 16
\ SHOW_PLAIN INIT_KEY_LENGTHS                        12:04 05-30-99

: <SHOW_PLAIN>  \ from --
    DECRYPT
    PLAINTEXT +  320  DUMP ;     \ a screenfull pretty much

: SHOW_PLAIN  \ --
    0 <SHOW_PLAIN> ;

: INIT_KEY_LENGTHS  \ --  \ sets BIG_KEY_LENGTHS as well
    KEY_SIZE 0 DO  I KEY_LENGTHS ZERO  LOOP
    BIG_KEY_LENGTHS ZERO ;


Screen # 17
\ INPUT_FILE OUTPUT_FILE                             20:40 05-30-99

: INPUT_FILE  \ filename buffer_ptr --
    >R  FILE1 NAME>HCB  R@ FILE_SIZE ERASE
    FILE1 READ_ONLY FOPEN  ABORT" can't open file for input."
    FILE1 R> FILE_SIZE FREAD  FILE_LENGTH !
    FILE1 FILE_MORE 1 FREAD ABORT" File is too big to load."
    FILE1 FCLOSE ABORT" can't close file for input." ;

: OUTPUT_FILE  \ filename buffer_ptr --
    >R  FILE1 NAME>HCB
    FILE1 WRITE_ONLY FMAKE  ABORT" Can't open file for output."
    FILE1 R> FILE_LENGTH @ FWRITE
    FILE_LENGTH @ < ABORT" Disk is full."
    FILE1 FCLOSE ABORT" Can't close file for output." ;


Screen # 18
\ COUNT_COINCIDENCES FILL_PAST_CIPHER                12:07 05-30-99
VARIABLE COIN_COUNT
VARIABLE COIN_SUM

: COUNT_COINCIDENCES  \ cipher_ptr1 cipher_ptr2 -- percentage
    COIN_COUNT ZERO  COIN_SUM ZERO
    BEGIN  DUP  PAST_CIPHER @  U< WHILE
        OVER C@  OVER C@  = IF  COIN_SUM INC  THEN
        SWAP 1+  SWAP 1+  COIN_COUNT INC  REPEAT
    2DROP
    COIN_SUM @  UNITY  COIN_COUNT @  */ ;
\ cipher_ptr1 is < cipher_ptr2

: FILL_PAST_CIPHER  \ --
    CIPHERTEXT  FILE_LENGTH @  +  PAST_CIPHER ! ;


Screen # 19
\ SEARCH_SIZE KEY_SEARCH_SIZE FILL_COINCIDENCES  12:08 05-30-99

: SEARCH_SIZE  \ -- search_size
    FILE_LENGTH @ 3 /  MAX_SEARCH_SIZE  MIN ;
\ We never shift less than one third of the file length.
\ This value is empirically determined.

: KEY_SEARCH_SIZE  \ -- key_search_size
```

be closer to 1.0 for languages other than English, or by happenstance in short files. The MOSTEST character doesn't have to strictly be the *most* frequent, as long as it is very frequent. The reason is that, in our KEY_CHAR array, we calculate the 256 best guesses for each character in the key. We have various ways of filtering out the "best" guesses, if they aren't likely to be characters the encrypter would have used in his key.

We have a two-dimensional array called KEY_CHAR which we are going to fill. Row 0 in the KEY_CHAR array will contain our best guess for what the key is. Row 1 is the second-best guess, and so forth. Let's first look at FILL_KEY in screen 28, and then work our way back through the lower-level routines.

FILL_KEY calls FILL_FREQS in screen 26 for each character of the key (column of KEY_CHAR). FILL_FREQS takes a pointer into CIPHERTEXT and increments through CIPHERTEXT by the key length. FILL_FREQS counts how many of each character is represented in CIPHERTEXT. FILL_FREQS is making this calculation as if for a Mono-alphabetic Substitution cipher whose characters just happen to be regularly spaced every KEY_LENGTH characters inside CIPHERTEXT.

FILL_KEY then calls COLUMN_FILL_KEY which will fill one column of KEY_CHAR. COLUMN_FILL_KEY calls SINGLE_FILL_KEY in screen 27 for each row. SINGLE_FILL_KEY takes the horizontal and vertical indices which it will be setting in KEY_CHAR. SINGLE_FILL_KEY finds the cipher character in FREQS which appears most often and assumes this must correspond to the MOSTEST plain character. SINGLE_FILL_KEY calculates what key character would have produced this cipher character, assuming that the plain character is the MOSTEST character. This character is stored in KEY_CHAR. SINGLE_FILL_KEY returns this most-frequent cipher character, the index into FREQS which pointed to the highest value. COLUMN_FILL_KEY stores a -1 value into this spot in FREQS before moving on to calculating the next most likely character. This is done so SINGLE_FILL_KEY doesn't find the same best value over and over.

Screen 30 has the TO_KEY_STRING routine. The author originally just copied row0 of KEY_CHAR over to KEY_STRING. This needed some enhancement. We were not taking into consideration that very few people are going to have a key with unprintable characters in it. We want to filter these out. We have several ways of filtering out unwanted characters. TO_KEY_STRING takes the cfa of a *char_kind* word (one of PRINTABLE, NUMERIC, SPANISH, UPPERCASE, LOWERCASE, ALPHA, ALPHA-NUMERIC, and PUNCTUATION). TO_KEY_STRING searches down each column in KEY_CHAR and finds the first character in the *char_kind* class which TO_KEY_STRING was given. Every column of KEY_CHAR will hold every possible character (each column has 256 entries), so we are bound to find something that satisfies our *char_kind* requirement. In this way, we get the best guesses which are of *some char_kind* class.

FILL_KEY_STRING does everything needed to determine the key contents. FILL_KEY_STRING uses ALPHA as its default *char_kind*. FILL_KEY_STRING is the word the user will type at the keyboard in order to do phase two of the program. In practice, especially when cracking short files, FILL_KEY_STRING will provide an incomplete answer (some key characters are right and some are wrong).

Interactive Guessing — Often needed on short files

There are two ways for the user to deal with an incorrect KEY_STRING content. One is to guess what the key string is, the other is to guess what the plaintext is. Often, by looking at the key string shown, the user can spot English words. If some characters seem wrong, look at the display of KEY_CHAR above for that character's column.

Scan down from the top to find a likely looking character. Use KEY_STRING! to set KEY_STRING. Use SHOW_PLAIN to see the resulting plaintext. The user can also use TO_KEY_STRING with some other *char_kind* routine (followed by SHOW_KEY_STRING) to try various filters. We have lots of *char_kind* routines. Note that encrypters sometimes are required to change their key every month. Often, people pick a key which is always used and then append the two-digit month number (01 of January, etc.) on the end of it. Look for patterns like this.

Back on screen 25, we had a word called TRY. After we have determined our KEY_STRING we normally run SHOW_PLAIN to see what we have achieved. We may find that the result is recognizeable text, but that some of the

```
    SEARCH_SIZE  KEY_SIZE  MIN ;

: FILL_COINCIDENCES  \ --  \ coincidences within CIPHERTEXT
    FILL_PAST_CIPHER
    SEARCH_SIZE 1 DO     \ minimum key length is 1
        CIPHERTEXT  DUP I +  COUNT_COINCIDENCES
        I COINCIDENCES  !  LOOP ;
```

```
Screen # 20
\ SHOW_INDEX SHOW_PERCENTAGE SHOW_TABLE_ENTRY    10:47 05-28-99

: SHOW_INDEX  \ index --
    0 <#  # #? #?  #> TYPE  ." )" ;

: SHOW_PERCENTAGE  \ percentage --  \ 2 digits right of decimal
    10 /  \ get rid of low digit
    0 <#  # ASCII . HOLD # #? #_  #> TYPE  ."   " ;

: SHOW_TABLE_ENTRY  \ percentage index --
    SHOW_INDEX  SHOW_PERCENTAGE ;

VARIABLE SHOW_FROM      \ starting index in PERCENTAGES
 48 CONSTANT SHOW_TOTAL  \ total percentages shown
  8 CONSTANT SHOW_ROW    \ should be a denominator of SHOW_TOTAL
```

```
Screen # 21
\ SHOW_COINCIDENCES SHOW_KEY_LENGTHS                 10:48 05-28-99

: SHOW_COINCIDENCES  \ from --  \ show SHOW_TOTAL at FROM
    SHOW_FROM !  CR
    SHOW_FROM @ SHOW_TOTAL + SEARCH_SIZE MIN  SHOW_FROM @  ?DO
        I COINCIDENCES @  I  SHOW_TABLE_ENTRY
        I 1+ SHOW_FROM @ -  SHOW_ROW MOD  0= IF  CR  THEN
        LOOP ;

: SHOW_KEY_LENGTHS  \ --  \ show them all
    CR  KEY_SIZE 1 DO
        I KEY_LENGTHS @  I  SHOW_TABLE_ENTRY
        I SHOW_ROW MOD  0= IF  CR  THEN
        LOOP
    CR ." too big = " BIG_KEY_LENGTHS @ SHOW_PERCENTAGE ;
```

```
Screen # 22
\ CALC_THRESHOLD                                     20:14 05-30-99

VARIABLE COIN_MIN  \ smallest value found in COINCIDENCES
VARIABLE COIN_MAX  \ largest value found in COINCIDENCES

: CALC_THRESHOLD  \ -- threshold  \ midpoint of COINCIDENCES
    100 COIN_MIN !  0 COIN_MAX !
    SEARCH_SIZE 1 DO  I COINCIDENCES @
        DUP COIN_MIN @ < IF  DUP COIN_MIN !  THEN
        DUP COIN_MAX @ > IF  DUP COIN_MAX !  THEN
        DROP LOOP
    COIN_MAX @ COIN_MIN @ - 2/  COIN_MIN @ + ;
```

```
Screen # 23
\ FILL_KEY_LENGTHS                                   12:21 05-29-99
: <FILL_KEY_LENGTHS>  \ distance_from_last_spike --
    DUP KEY_SIZE < IF  \ within key
```

```
        KEY_LENGTHS INC
    ELSE
        DROP  BIG_KEY_LENGTHS INC   THEN ;

: FILL_KEY_LENGTHS  \ -- spike_count
    0 0  \ spike_count last_spike --
    SEARCH_SIZE 1 DO
        I COINCIDENCES @  THRESHOLD @  U> IF  \ found a spike
            I SWAP -  <FILL_KEY_LENGTHS>
            1+  I  THEN  \ spike_count last_spike --
        LOOP
    0= ABORT" We found no spikes at all!" ;
```

Screen # 24

```
\ KEY_LENGTHS% CALC_KEY_LENGTH                          21:36 05-30-99

: KEY_LENGTHS%  \ spike_count --  \ change to percentages
    KEY_SIZE 1 DO
        I KEY_LENGTHS @  UNITY  ROVER  */  I KEY_LENGTHS !
        LOOP
    BIG_KEY_LENGTHS @  UNITY  ROT  */  BIG_KEY_LENGTHS ! ;

: CALC_KEY_LENGTH  \ -- length
    INIT_KEY_LENGTHS  FILL_KEY_LENGTHS KEY_LENGTHS%
    0  \ max_key_length --
    KEY_SIZE 1 DO
        I KEY_LENGTHS @  OVER KEY_LENGTHS @  > IF
            DROP  I  THEN
        LOOP ;
\ CALC_KEY_LENGTH uses the lower index if two have = values.
```

Screen # 25

```
\ FILL_KEY_LENGTH TRY                                   20:10 05-30-99

: FILL_KEY_LENGTH  \ --
    FILL_COINCIDENCES  1 SHOW_COINCIDENCES
    CALC_THRESHOLD  THRESHOLD !
    CR ." threshold = "  THRESHOLD @ SHOW_PERCENTAGE
    CALC_KEY_LENGTH KEY_LENGTH!  SHOW_KEY_LENGTHS
    CR ." Key length is: " KEY_LENGTH @ . ;

: TRY  \ plain_char horz_index vert_index --
```

(Figure Two — source code — continues on page 24.)

Figure Four.

```
AMENDMENT 4.

The right of the people to be secure in their persons,
houses, papers, and effects, against unreasonable searches
and seizures, shall not be violated, and no warrents shall
issue but upon probable cause, supported by oath or
affirmation, and particularly describing the place to be
searched, and the persons or things to be seized.
```

Figure Five.

```
" MESSAGE.TXT" PLAINTEXT INPUT_FILE
" Very-Personal" KEY_STRING!  ENCRYPT
```

characters are wrong. These wrong characters correspond to erroneous characters in KEY_STRING. Fixing this interactively is what TRY is for.

`TRY` takes a plain character, and a horizontal and vertical index into `PLAINTEXT`. We are hoping this plain character is what should go in that spot in `PLAINTEXT`. The reason we have a horizontal and vertical index into `PLAINTEXT` is that the `DUMP` in `SHOW_PLAIN` displays `PLAINTEXT` as a two-dimensional array. We are, presumably, using `TRY` after using `SHOW_PLAIN` while we are looking at `SHOW_PLAIN`'s output. `TRY` fixes the corresponding character in `KEY_STRING` and reruns `SHOW_PLAIN`. We can `TRY` another character, or we can stop if our plain text looks correct. This is quite similar to the Jeopardy game, in which a person looks at a plaintext message with some of the characters missing and tries to guess what those characters are. When the plaintext appears to be correct, execute `SHOW_KEY_STRING` to find out what key `TRY` has built.

An Example Run — The program from the user's persective

We are done with our study of the encryption-cracking program. Let's run through an example. The reader should enter the text in Figure Four exactly, and save it in a file called Message.txt. Be careful to put the end-of-lines at the same places so that the program will give the exact same results we will describe here. Message.txt should have a length of 354 characters.

Execute the code shown in Figure Five in order to fill `CIPHERTEXT` with encrypted data. Now pretend you don't know what the plaintext is or what the key is, and try to crack the cipher. First execute `FILL_KEY_LENGTH`. This will result in an output as shown in Figure Six. It seems clear that the key length is 13, since there is a 60% chance this is true. We have a 20% chance of it being 26, and a 20% chance of it being 52. Note that both 26 and 52 are multiples of 13. Take a glance over the `COINCIDENCES` data at the top and note that 13 has a value of 9.9%, which is considerably higher than the other values. This is definitely a spike.

Execute `FILL_KEY_STRING`. This will result in an output as shown in Figure Seven. The program has found "VeryePersonal". This looks good, except for that 'e' after "Very". Look at the

KEY_CHAR data shown at the top of the screen. Scan down the fifth column. The top character is 'e' and the second best one is '-'. The hyphen seems likely. Executing the code `" Very-Personal" KEY_STRING! SHOW_PLAIN` will show that this is the correct key. An alternative to scanning down columns in KEY_CHAR would be to use the *char_kind* filters. It seems clear there must be some punctuation character or a blank between "Very" and "Personal". Execute `' PUNCTUATION TO_KEY_STRING SHOW_KEY_STRING` which will set KEY_STRING to an all-punctuation guess. Look at what the fifth character is, and discover it is a hyphen. Scanning the columns in KEY_CHAR and using the *char_kind* filters are the two methods used for guessing the key.

Let's go back to our "VeryePersonal" key and try guessing the plaintext. Execute SHOW_PLAIN to see the plaintext. The result should be as shown in Figure Eight. This is clearly English plaintext with some characters wrong. For example, on the sixth row we see a word "pa8ers". We can guess that this is supposed to be the word "papers". Execute `ASCII p 2 5 TRY` to try out a 'p' in place of that '8'. Note that we are using a horizontal index of 2, since we start counting at zero. We are also using a vertical index of 5, since we count the rows from the top down, starting at zero. TRY automatically executes SHOW_PLAIN after fixing its KEY_STRING character so the user can see the result. Sometimes it is necessary to use TRY several times to fix several characters (or to fix one character, if you're not sure what it should be). When the plaintext looks correct, use SHOW_KEY_STRING to find what key you built with your various TRY executions.

Figure Six.

```
  1) 2.5    2) 3.4    3) 0.8    4) 1.7    5) 1.7    6) 0.8    7) 0.2    8) 1.7
  9) 1.7   10) 1.1   11) 1.1   12) 1.4   13) 9.9   14) 2.3   15) 2.0   16) 1.7
 17) 1.4   18) 2.6   19) 2.6   20) 1.1   21) 1.8   22) 0.6   23) 1.5   24) 1.2
 25) 1.2   26) 4.5   27) 1.2   28) 2.1   29) 1.2   30) 0.3   31) 2.1   32) 0.9
 33) 0.0   34) 3.4   35) 1.2   36) 1.2   37) 0.9   38) 1.5   39) 6.3   40) 0.9
 41) 0.3   42) 1.2   43) 0.3   44) 2.9   45) 0.6   46) 2.2   47) 2.9   48) 0.9

threshold =  4.9
  1) 0.0    2) 0.0    3) 0.0    4) 0.0    5) 0.0    6) 0.0    7) 0.0    8) 0.0
  9) 0.0   10) 0.0   11) 0.0   12) 0.0   13)60.0   14) 0.0   15) 0.0   16) 0.0
 17) 0.0   18) 0.0   19) 0.0   20) 0.0   21) 0.0   22) 0.0   23) 0.0   24) 0.0
 25) 0.0   26)20.0   27) 0.0   28) 0.0   29) 0.0   30) 0.0   31) 0.0   32) 0.0
 33) 0.0   34) 0.0   35) 0.0   36) 0.0   37) 0.0   38) 0.0   39) 0.0   40) 0.0
 41) 0.0   42) 0.0   43) 0.0   44) 0.0   45) 0.0   46) 0.0   47) 0.0   48) 0.0
 49) 0.0   50) 0.0   51) 0.0   52)20.0   53) 0.0   54) 0.0   55) 0.0   56) 0.0
 57) 0.0   58) 0.0   59) 0.0   60) 0.0   61) 0.0   62) 0.0   63) 0.0   64) 0.0
 65) 0.0   66) 0.0   67) 0.0   68) 0.0   69) 0.0   70) 0.0   71) 0.0   72) 0.0
 73) 0.0   74) 0.0   75) 0.0   76) 0.0   77) 0.0   78) 0.0   79) 0.0   80) 0.0
 81) 0.0   82) 0.0   83) 0.0   84) 0.0   85) 0.0   86) 0.0   87) 0.0   88) 0.0
 89) 0.0   90) 0.0   91) 0.0   92) 0.0   93) 0.0   94) 0.0   95) 0.0   96) 0.0
 97) 0.0   98) 0.0   99) 0.0
too big =  0.0
Key length is: 13
```

Figure Seven.

```
V e r y e · e   6 o   $ ?
·   ! 8 - ·   ! ! & / 3 1
· + < * o · ) 1 '   ; & )
· * = - } · $ 3 0 * &   "
· , " < y · , 7 : , * . (
· 1 6 , ~ · 7 ; s . + 2
· H X / ! · * =   ! = 5 *
Z #   4 b = + > # # > a -
7 $ 0 7 c · - " & ' ! % .
· ' 3 ; d · 1 % 1 ( ' - <
· - 4 = h · 6 & 2 5 4 / >
· 6 7 ? i · · ' ; : : m `
· · > · l } O : < = < o A
· i · · x P ! · = ? n ! !
{ O _ S     " r · c z " #
| ! #   I ! # # ^ B D # $
X " $ !   " % $   E " ' %
  % % " " # & ( " " # ( &

               0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
0017:015C10   56 65 72 79 65 50 65 72 73 6F 6E 61 6C           VeryePersonal
```

(Figure Two – source code – continued.)

```
    16 * + >R  R@ KEY_LENGTH @ MOD KEY_STRING
    R> CIPHERTEXT + C@  \ plain_char key_ptr cipher_char --
    ROT GET_KEY  SWAP C!
    SHOW_PLAIN ;
\ TRY assumes PLAINTEXT is paragraph aligned.
\ TRY acts like PLAINTEXT is a 16 wide 2d array (as DUMP shows).


Screen # 26
\ INIT_FREQS FILL_FREQS                          12:33 05-30-99

: INIT_FREQS  \ --
    CHARS 0 DO  I FREQS ZERO  LOOP ;

: FILL_FREQS  \ cipher_ptr --  \ steps by KEY_LENGTH
    INIT_FREQS
    PAST_CIPHER @  SWAP  DO
        I C@ FREQS  INC
        KEY_LENGTH @ +LOOP ;


Screen # 27
\ BEST_CIPHER_CHAR SINGLE_FILL_KEY               20:10 05-30-99

: BEST_CIPHER_CHAR  \ -- best_cipher_character
    -1 -1  \ best_cipher_char best_cipher_char_occurances --
    CHARS 0 DO  \ I is the test character
        I FREQS @  OVER  > IF  2DROP
            I  I FREQS @  THEN
        LOOP
    -1 = ABORT" FREQS was corrupt" ;

: SINGLE_FILL_KEY  \ horz_index vert_index -- best_cipher_char
```

(Figure Two – source code – continues on next page.)

Final Thoughts — PolySub encryption is a toy algorithm

Try the program using different key sizes. Try it with "SUPERCALIFRAGILISTIC" for a difficult exercise, and with "UNIQUE" for an easy exercise. Try it using a key containing mixed uppercase, lowercase, numbers, and so forth. It is kind of fun to crack ciphers with the program; it can be like solving a puzzle. Readers may also find it enjoyable to beef up CrakPoly in various ways. There are enhancements which would make CrakPoly better at cracking very short ciphers, though it is already quite good. Our Message.txt file was only 354 bytes, and CrakPoly cracked it with ease. The best enhancement would be to get rid of TRY's need for numeric coordinates into PLAINTEXT. Entering these is tedious and error-prone. We would want TRY to allow the user to move a cursor around in the plaintext with his arrow keys. When he got his cursor over an offending character, he would type the correct character and TRY would fix the key string and display a regenerated PLAINTEXT.

It is hoped that the reader has found our discussion of CrakPoly to be interesting. There might be a few readers who have a practical need for it. An example would be a company owner who could write a PolySub program and give it to his employees, saying, "Use this on all

Figure Eight.

```
               0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   0123456789ABCDEF
0017:096390   41 4D 45 4E 0C 4D 45 4E 54 20 34 2E 0D 0A 0D 0A   AMEN.MENT 4.....
0017:0963A0   54 20 65 20 72 69 67 68 74 20 6F 66 20 74 20 65   T e right of t e
0017:0963B0   20 70 65 6F 70 6C 65 20 74 6F 20 2A 65 20 73 65    people to *e se
0017:0963C0   63 75 72 65 20 69 6E 20 3C 68 65 69 72 20 70 65   cure in <heir pe
0017:0963D0   72 73 6F 6E 73 64 0D 0A 68 6F 75 73 65 73 2C 20   rsonsd..houses,
0017:0963E0   70 61 38 65 72 73 2C 20 61 6E 64 20 65 66 66 2D   pa8ers, and eff-
0017:0963F0   63 74 73 2C 20 61 67 61 69 6E 73 74 68 75 6E 72   cts, againsthunr
0017:096400   65 61 73 6F 6E 61 62 6C 65 68 73 65 61 72 63 68   easonablehsearch
0017:096410   65 73 0D 0A 61 6E 2C 20 73 65 69 7A 75 72 65 73   es..an, seizures
0017:096420   2C 20 73 20 61 6C 6C 20 6E 6F 74 20 62 65 20 76   , s all not be v
0017:096430   21 6F 6C 61 74 65 64 2C 20 61 6E 64 20 26 6F 20   !olated, and &o
0017:096440   77 61 72 72 65 6E 74 73 20 73 20 61 6C 6C 0D 0A   warrents s all..
0017:096450   69 73 73 75 65 20 62 3D 74 20 75 70 6F 6E 20 70   issue b=t upon p
0017:096460   72 6F 62 61 2A 6C 65 20 63 61 75 73 65 2C 20 73   roba*le cause, s
0017:096470   75 38 70 6F 72 74 65 64 20 62 79 20 6F 61 3C 68   u8ported by oa<h
0017:096480   20 6F 72 0D 0A 61 66 66 69 72 6D 29 74 69 6F 6E    or..affirm)tion
0017:096490   2C 20 61 6E 64 20 70 61 3A 74 69 63 75 6C 61 72   , and pa:ticular
0017:0964A0   6C 79 20 64 65 3B 63 72 69 62 69 6E 67 20 74 68   ly de;cribing th
0017:0964B0   65 20 38 6C 61 63 65 20 74 6F 20 62 65 0D 0A 3B   e 8lace to be..;
0017:0964C0   65 61 72 63 68 65 64 2C 20 61 6E 64 68 74 68 65   earched, andhthe
```

sensitive documents to prevent corporate espionage." Many of the employees will use it on personal documents they are storing on company computers. These, of course, are what the company owner was actually interested in. For the most part, however, CrakPoly is just a toy program without any commercial prospects.

CrakPoly could only be written in Forth, and it would never have been written in C++. The reason is that CrakPoly is necessarily interactive, with `TRY` and `TO_KEY_STRING` and `KEY_STRING!` and so forth. To write a GUI that would achieve this level of interactiveness would be more work than would be justified for a toy program. All commercial products these days have GUI interfaces, and C++ is oriented towards writing GUIs. C++ does not have any ready facility for executing commands from the keyboard. The author has used LEX/YACC under C++ to provide programs with a command-line interface. This is a powerful technique, but it also requires a lot of work. In Forth, the command-line interface is free. In general, a person who only knows C++ would have to decide that CrakPoly requires more work than it is worth, and would never start the project. This would be a shame, because CrakPoly does have some value.

The author found that writing CrakPoly was fun, and that using it is fun, too. Also, designing and writing fun programs is good practice for working on commercial products. C++, with its emphasis on GUIs and commercial development, requires too much work to be used in weekend projects. Because nobody programs as a leisure activity anymore, in so doing getting practice at programming, our professional programming is now described with terms like "death march project" and "anti-pattern." These apparently are the wages of professionalism.

In case any reader has been using the PolySub to encrypt anything of value, this article should dissuade him. Perhaps, in the future, we can delve into writing an encryption program which does provide good security. In the meantime, the reader is encouraged to use PGP, which provides good security and is a standard method of encryption. It is good to have a standard so that everybody can easily exchange encrypted files with one another. Standardizing on the PolySub because it is well-known, however, would be a mistake.

(Figure Two – source code – continued.)

```
    KEY_CHAR >R  BEST_CIPHER_CHAR
    DUP  MOSTEST C@  GET_KEY  R> C! ;


Screen # 28
\ COLUMN_FILL_KEY FILL_KEY                              12:34 05-30-99

: COLUMN_FILL_KEY  \ horz_index --
    CHARS 0 DO  \ I is the vert_index
        DUP I SINGLE_FILL_KEY  \ horz_index best_cipher_char --
        FREQS -1 SWAP !  \ won't be the best of next vert_index
        LOOP
    DROP ;

: FILL_KEY  \ --
    FILL_PAST_CIPHER
    KEY_LENGTH @ 0 DO  \ I is horz_index
        CIPHERTEXT I +  FILL_FREQS
        I COLUMN_FILL_KEY
        LOOP ;


Screen # 29
\ SHOW_KEY SHOW_KEY_HEX                                 19:18 05-29-99
: SHOW_KEY  \ --
    CR
    SHOW_KEYS 0 DO      \ J = vert_index
    KEY_LENGTH @ 0 DO   \ I = horz_index
        I J KEY_CHAR C@  DUP PRINTABLE IF
            EMIT  ELSE  DROP  NON_CHAR EMIT  THEN
        SPACE LOOP  CR LOOP ;

: SHOW_KEY_HEX  \ --
    CR  BASE @ >R HEX
    SHOW_KEYS 0 DO      \ J = vert_index
    KEY_LENGTH @ 0 DO   \ I = horz_index
        I J KEY_CHAR C@  0 <# # # BL HOLD #> TYPE
        LOOP  CR LOOP
    R> BASE ! ;


Screen # 30
\ TO_KEY_STRING FILL_KEY_STRING                         19:47 05-30-99
: <TO_KEY_STRING>  \ --
    KEY_LENGTH @ 0 DO   \ J is the horz_index
        0  I KEY_STRING  C!  \ default
        CHARS 0 DO      \ I is the vert_index
            J I KEY_CHAR C@  DUP 'CHAR_KIND PERFORM IF
                J KEY_STRING C!  LEAVE  ELSE  DROP  THEN
            LOOP
        LOOP ;

: TO_KEY_STRING  \ char_kind_cfa --
    'CHAR_KIND !  <TO_KEY_STRING> ;

: FILL_KEY_STRING  \ --
    FILL_KEY  SHOW_KEY
    ['] ALPHA TO_KEY_STRING  SHOW_KEY_STRING ;
```

PRESWOOP

Rick VanNorman took my Simple Object-Oriented Programming and extended it. It is much more powerful. Because of the extra power, it is no longer a simple implementation, but it is still easy to use and fast.

Rick implemented SWOOP for SwiftForth using general-purpose SwiftForth words. It is an easy task to define these general-purpose words in Standard Forth. With that prelude, SWOOP becomes available for Forths conforming to Standard Forth. I have been using Swoop in my work since the beginning of the year.

If you already have definitions for these words with the same meaning, you should comment out those definitions here—especially when your definitions are more efficient.

There are two problems not handled by Standard Forth.

1. In extending the set of classes, using `MARKER` may corrupt the list. In SwiftForth, PowerMacForth, and probably others, `CHAIN` *name* cooperates with `MARKER` to discard tokens that would cause trouble.
2. ANS Forth specifies word list identifiers as "implementation-dependent single-cell values that identify word lists," which is the weakest possible specification, meaning you know nothing about them. ANS Forth also ignores saving the system after compiling new definitions, and then reloading the system with a possible relocation of addresses.

Some systems, such as PowerMacForth, define a word list identifier (*wid*) so that it is valid only in the session in which it's defined. To provide maintenance and transition, `WORDLIST:` should provide, in such systems, named word list identifiers that can be used across sessions. The definition of `WORDLIST:` here is for implementations without a problem with word list identifiers.

```
0 [ IF] ******************************************************

ANS Prelude for SWOOP
====================

All these definitions are generally useful.

Comment out definitions with the same meaning that you
already have.

{

CELL   CELL-   /ALLOT   ?EXIT   -EXIT   !+   @+   STRING,

CHAIN   RELINK,   >LINK

-ORDER   +ORDER

CREATE-XT   WORDLIST:

****************************************************** [ THEN]

0 [ IF] ------------------------------------------------------

{  begins a comment that may extend over multiple lines
until a terminating right brace } is encountered.     ( -- )

This definition is first so it can be used henceforth.
```

Wil Baden • wilbaden@netcom.com
Costa Mesa, California

All code was checked in PowerMacForth and SwiftForth. Wil Baden, after years of profane language, retired to Standard Forth. For the source for this article, send e-mail requesting Standard Forth Tool Belt #8: "PRESWOOP."

```
--------------------------------------------------- [ THEN]

: NOT  0= ;

: {                        ( "ccc...}" -- )
    BEGIN  SOURCE +                   ( addr)
        [ CHAR] }  PARSE + > NOT WHILE   ( )
    REFILL 0= UNTIL THEN ; IMMEDIATE

{ ---------------------------------------------------------

CELL   CELL-   /ALLOT   ?EXIT   -EXIT   !+   @+   STRING,
=========================================================

CELL  and  CELL-  are convenient for address arithmetic.

/ALLOT  allots and clears dataspace.

?EXIT  is  IF EXIT THEN

-EXIT  is  0= IF EXIT THEN

@+  fetches the value x from addr, and increments the address
   by one cell.                     ( addr -- addr+4 x )

!+  writes the value x to addr, and increments the address by
   one cell.                        ( addr x -- addr+4 )

STRING, compiles the string at addr, whose length is u, in the
   dictionary starting at HERE, and allocates space for it.
                                             ( addr u -- )

These are all in SwiftForth, PowerMacForth, and others.

----------------------------------------------------------- }

1 CELLS CONSTANT CELL

: CELL-  CELL - ;

: /ALLOT ( n -- ) HERE SWAP  DUP ALLOT  ERASE ;

: ?EXIT  ( n -- )  \  IF EXIT THEN
   POSTPONE IF  POSTPONE EXIT  POSTPONE THEN ; IMMEDIATE

: -EXIT  ( n -- )  \  0= IF EXIT THEN
   POSTPONE 0=  POSTPONE IF  POSTPONE EXIT  POSTPONE THEN ;
   IMMEDIATE

: !+   ( addr n -- addr+CELL )  OVER ! CELL+ ;

: @+   ( addr -- addr+CELL n )  DUP CELL+ SWAP @ ;
```

```
: STRING,   ( str len -- )
    HERE  OVER 1+ CHARS ALLOT  2DUP C!  CHAR+  SWAP MOVE ;

{ -------------------------------------------------------------

CHAIN   RELINK,   >LINK
=======================

For relocation of machine addresses, they are referenced
self-relative.

CHAIN <name>  defines the head of a linked-list of addresses.
   The list must be pruned when elements are forgotten.
   In SwiftForth and PowerMacForth this will be done for you.
                                                   ( "name" -- )

RELINK,  takes a link from one list and installs it in the
   current list.                                     ( addr -- )

>LINK  adds a link starting at HERE to the top of the linked
   list whose head is at addr (normally a variable). The head
   is set to point to the new link, which, in turn, is set to
   point to the previous top link.                   ( addr -- )

------------------------------------------------------------- }

: CHAIN   ( "name" -- )  CREATE  0 , ;

: RELINK,  ( a -- ) DUP @ DUP IF  OVER +  HERE -  THEN , DROP ;

: >LINK  ( a -- )  ALIGN  HERE  OVER RELINK,  OVER - SWAP ! ;

{ -------------------------------------------------------------

-ORDER   +ORDER
===============

-ORDER removes a word list from the context, wherever it is.
                                                      ( wid -- )

+ORDER puts a word list in the context at the top.
                                                      ( wid -- )

------------------------------------------------------------- }

: -ORDER               ( wid -- )
   >R  GET-ORDER              ( widn ... wid1 n)( R: wid)
      DUP BEGIN  DUP WHILE    ( widn ... wid1 n i)
         DUP 1+ PICK          ( widn ... wid1 n i widi)
         R@ = IF              ( widn ... wid1 n i)
            DUP 1+ ROLL DROP
            >R 1- R>
```

Toolbelt #8 code continues on page 49.

SWOOP:
Object-Oriented Programming in SwiftForth

Wil Baden kindly introduced my object implementation in the preceding issue of *Forth Dimensions*. Here I will attempt to present the details of its operation.

1. Origins and motivations

Prior to embarking on this project, I evaluated several Forth OOP implementations: Baden[1], Ertl[4], McKewan[5], and Pountain[6]. None entirely met my requirements.

The first consideration I faced was the order of the object/message tuples. The two fundamental flavors of this syntax are *message-object* and *object-message*. Both have existing implementations, pros and cons, supporters and detractors. I decided on *object-message* because it more closely paralleled the Forth programming paradigm. It also has the benefit, in nested object definitions, of progressing from the general to the specific, or from the collection of data to the individual datum.

My second consideration was whether to have the components of a class parse or not. In most of the object-oriented Forths I studied, each entity parses its successor and determines what the phrase means. Ertl objected strongly to this as limiting the usefulness and extensibility of the messaging model—making it difficult to pass messages on the Forth stack—and as imposing an artificial dependency on the adjacency of operands. I agree with this analysis, and developed a syntax almost completely independent of parsing requirements.

The third consideration was that the class model had to provide for encapsulation and information hiding. This is apparently an absolute requirement if an object model is to be taken seriously. Some existing systems provide this, others do not.

All these features were implemented to one degree or another in the various systems I evaluated. But none addressed my fourth consideration: the need for the generated code to be target-compilable. This reduces to the need for the compile and interpret behaviors and structures to be fully separate from, and independent of, the run-time code.

2. Basic SWOOP Components

2.1. Defining a simple class

`POINT` (defined below) is a simple class I have been using as my primary building-block example for SWOOP. It demonstrates two of the four basic class member types: *data* and *colon.*

The word following `CLASS` is the name of the class; all definitions between `CLASS` and `END-CLASS` belong to it. These definitions are referred to as *members* of the class. When a class name is executed, it leaves its handle (*xt*) on the stack. The constructor words are the primary consumers of this handle.

```
CLASS POINT
      VARIABLE X
      VARIABLE Y
      : SHOW ( -- )   X @ . Y @ . ;
      : DOT ( -- )   ." Point at " SHOW ;
END-CLASS
```

The class definition itself does not allocate any instance storage; it only records how much storage is required for each instance of the class. `VARIABLE` reserves a cell of space and associates it with a member name.

The colon members `SHOW` and `DOT` are exactly like normal Forth colon definitions, except they are only valid in the execution context of an object of type `POINT`. `X` and `Y` also behave exactly like normal Forth `VARIABLE`s except for being valid only within the scope of a `POINT` object.

There are four kinds of members:

1. Data members, including all data definitions. Available data member defining words include `CREATE` (normally followed by data compiled with `,` or `C,`), `BUFFER:` (an array whose length is specified in address units), `VARIABLE`, `CVARIABLE` (single *char*), or `CONSTANT`s;
2. Colon members, normal colon definitions that may act on or use data members;
3. Deferred members, colon-like definitions with a default behavior that can be referenced while defining the class, but may have substitute behaviors defined by sub-classes defined later;
4. Other objects.

The deferred members allow for polymorphism and late binding, and will be discussed later.

2.2. Static instances of a class

Having defined a class, we can create an instance of it. `BUILDS` is the static instance constructor in SWOOP; it is a Forth defining word and requires the handle of a class on the stack when executed.

```
POINT BUILDS ORIGIN
```

This defines a static object of class `POINT` named `ORIGIN`. Now, any of the members of `POINT` may be referenced in the context of this object. For example:

```
5 ORIGIN X !
8 ORIGIN Y !
ORIGIN DOT
```

When the name of an object is executed, two things happen: first, the Forth interpreter's context is modified to include the namespace of the class that created it. Second, the

Rick Van Norman • rvn@forth.com
Manhattan Beach, California

Rick VanNorman, a Forth gypsy for almost 20 years, finally found a home at FORTH, Inc.

Some OOP Terminology

class A generalized specification for objects that will share common data structures and methods.

deferred member In SWOOP, a method that is subject to late binding. In C++, this is referred to as a *virtual method.*

early binding Resolving references to functions statically at compile time (the normal behavior of compilers). This gives the best performance, but is less flexible than late binding.

encapsulation Combining data and methods into a single package that responds to messages.

information hiding The ability of an object to possess data and methods that are not accessible outside its class.

inheritance The ability to define a new class based on a previously defined ("parent") class, and to have the new class automatically possess all members of the parent. It may add to or replace these members, or define actual behaviors for deferred members.

instance An object constructed according to a class specification. An instance is to its class as a building is to its blueprints.

instance data The data structures within an instance.

late binding The ability of an object to resolve references to functions dynamically at run time, based on the message sent to it. This is extremely flexible, but is inevitably slower than early binding.

member In C++, a data field in an object; in SWOOP, members include both data fields and methods.

message Data passed to an object for the purpose of requesting it to execute one of its methods.

method A function performed by an object in response to a message.

namespace In SWOOP, the names of members recognized in a particular object class, including its locally defined members in addition to those inherited from parent classes.

object A packaged combination of data and methods.

object-oriented programming (OOP) A programming system that features encapsulation, information hiding, polymorphism, and late binding.

polymorphism The ability of different sub-classes of a class to respond to the same message in different ways. For example, all vehicles can steer, but bicycles do it differently from automobiles.

address of the object is placed on the stack. The phrase

```
ORIGIN 2 CELLS DUMP
```

is perfectly valid (assuming you have a suitable `DUMP` function). Each of the members of the class act on this address. Members that represent data simply add an offset to it; members that are defer or colon definitions push the address into `'SELF` (which holds the current object address) before executing, and restore it afterwards.

2.3. Dynamic objects

We can also create a temporary context in which to reference the members of a class. `USING` parses the word following it and, assuming that it is the name of a class, makes its members available for use on data at a specified address. For instance, I can place data at `PAD` and use the members of `POINT` to act on it:

```
6 PAD !  9 PAD CELL+ !
PAD USING POINT DOT
```

This will print 6 and 9. It is a very simple, user-managed dynamic instance of a class. It is also, generally, *not* a good way to use dynamic objects.

A better idea is to let SWOOP manage dynamic instances for you. `NEW` is the dynamic constructor. It is not a defining word, but is a memory management word similar to `ALLOCATE`. It requires a class handle on the stack, and returns an address. When the object is no longer needed, it can be disposed of with `DESTROY`.

```
0 VALUE FOO
POINT NEW TO FOO
8 FOO USING POINT X !
99 FOO USING POINT Y !
FOO USING POINT DOT
FOO DESTROY  0 TO FOO
```

This example uses `FOO` to keep up with the address of an instance of `POINT`. After the instance is created, it may be manipulated (with a slight change in syntax) in the same way a static instance of `POINT` is. When it's no longer needed, the instance is destroyed and the address kept in `FOO` is invalidated.

Objects created by `NEW` do not exist in the Forth dictionary, and must be explicitly destroyed when no longer used.

Another form of dynamic object instantiation is *local objects*. These, like local variables, are available only inside a single colon definition, and are instantiated only while the definition is being executed. Here's an example:

```
: TEST ( -- )
   [OBJECTS  POINT MAKES JOE  OBJECTS]
   JOE DOT ;
```

You can define as many local objects as you need between `[OBJECTS` and `OBJECTS]`. They will all be instantiated when `TEST` is executed, and destroyed when it is completed. This is a particularly useful facility in Windows programming, because these objects can be used in Windows callback routines. Unfortunately, local objects cannot be implemented straightforwardly in ANS Forth, as that depends heavily on internal SwiftForth implementation features, so they are not included in the released code.

2.4. Embedded objects

Pre-defined classes may be used as members of other classes. The syntax for using one is the same as for defining static objects. These objects are not static; they will be constructed only when their container is instantiated.

```
CLASS RECTANGLE
   POINT BUILDS UL
   POINT BUILDS LR
   : SHOW ( -- )   UL DOT  LR DOT ;
   : DOT ( -- )   ." Rectangle, " SHOW ;
END-CLASS
```

In this example, the points giving the upper-left and lower-right corners of the rectangle are instantiated as POINT objects. The members of RECTANGLE may reference them by name, and may use any of the members of POINT to manipulate them. In this example, SHOW references the DOT member of POINT to print UL and LR; this member is *not* the same as the DOT member of RECTANGLE.

These embedded objects are exactly like data allocations in the class: they simply add an offset to the base address of the object's data when referenced. There is nothing special about creating an instance of such a class, but the created object has all public members of the embedded objects available as well.

2.5. Information hiding

Classes are composed of named members. Thus far, all the members have been visible in any reference to the class or an object of the class. Even though the member names are hidden from casual reference by the user, the information-hiding requirements of object-oriented programming are more stringent.

The accepted level of information hiding in OOP seems to be that classes must have at least the ability to hide members from any external access. SWOOP accomplishes this via three keywords:

- PUBLIC identifies members that can be accessed globally.
- PROTECTED identifies members that are available only within the class in which they are defined, and in its subclasses.
- PRIVATE identifies members that are available only within the defining class.

When a class definition is begun, all member names default to being PUBLIC, which is to say visible outside of the class definition. PRIVATE or PROTECTED changes the level of visibility of the members.

```
CLASS POINT
PRIVATE
  VARIABLE X
  VARIABLE Y
  : SHOW ( -- )   X @ .  Y @ . ;
PUBLIC
  : GET ( -- x y )   X @ Y @ ;
  : PUT ( x y -- )   Y ! X ! ;
  : DOT ( -- )  ." Point at " SHOW ;
END-CLASS
```

In this definition of POINT the members X, Y, and SHOW are now private, available to local use while defining POINT and hidden from view afterwards. Since a point is relatively useless unless its location can be set and read, members which can do this are provided in the public section. However, these definitions achieve the desired level of information hiding: the actual data storage is unavailable to the user and may only be accessed through the members provided for that purpose.

2.6. Inheritance and polymorphism

Inheritance is the ability to define a new class based on an existing class. The new *sub*-class, which initially has exactly the same members as its parent, can replace some of the inherited members or can add new ones. If the subclass redefines an existing member, all further use within the subclass will reference the new one; all prior references were already bound and continue to reference the previous member.

Polymorphism goes a step further than inheritance. In it, a new subclass inherits all the members of its parents, but may also redefine any DEFER: members of its parents.

For example, our previous example could be written this way:

```
CLASS POINT
  VARIABLE X
  VARIABLE Y
  DEFER: SHOW ( -- )   X @ . Y @ . ;
  : DOT ( -- )   ." Point at " SHOW ;
END-CLASS
```

Then you could make a sub-class like this:

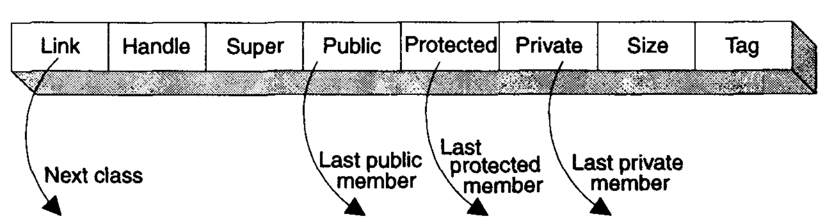
```
POINT SUBCLASS LABEL-POINT
  : SHOW ( -- )
    ." X" X @ .  ." Y" Y @ . ;
END-CLASS
LABEL-POINT BUILDS POO
POO DOT
```

The original definition DOT in the parent class POINT will still reference SHOW, but when it is executed for an instance of LABEL-POINT, the new behavior will automatically be substituted, so POO DOT will print the labeled coordinates.

3. Data Structures

This section will describe the basic data structures involved in classes and members, as a foundation for discussing more-detailed implementation strategies underlying SWOOP.

Figure One. Structure of a class



3.1. Classes

The data representation of a class is shown in Figure One. Each class is composed of a eight-cell structure. All classes are linked in a single list that originates in the list head `CLASSES`. This allows the system or user to see all created classes, and will be used in the future to facilitate the implementation of a class browser.

Each class has a unique handle. When executed, a class name will return this handle. The handle also happens to be the *xt* that is returned by ticking the class name. For example, if `POINT` is a class, then

```
' POINT .
```

prints the same value as

```
POINT .
```

Each class (except `SUPREME`) has a superclass. By default, it is `SUPREME`, but a class can be a child of any pre-existing class. The value in the Super field is the handle (*xt*) of the superclass.

Classes are composed of members, divided into three lists—public, protected, and private—which are identical except for their visibility to external references. Each list has a head in the class data structure. With inheritance, these lists may chain back into its superclass, and its superclass, etc., all the way back to `SUPREME`. The ordering within the chain is such that the head points to the last (most recently defined) member, which is linked to the next most recently defined, etc. This is the same ordering as within a Forth dictionary, and allows for redefinitions. These lists, in conjunction with the class handle and the wordlist `MEMBERS`, define the class namespace.

The size field represents the size (in bytes) required by a single instance of the class. This value is the sum of all explicitly referenced data in the class itself plus the size of its superclass.

The class tag is a simple constant used to identify the data structure as a valid class.

Figure Two. Basic structure of a member

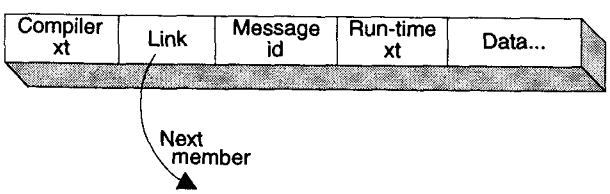
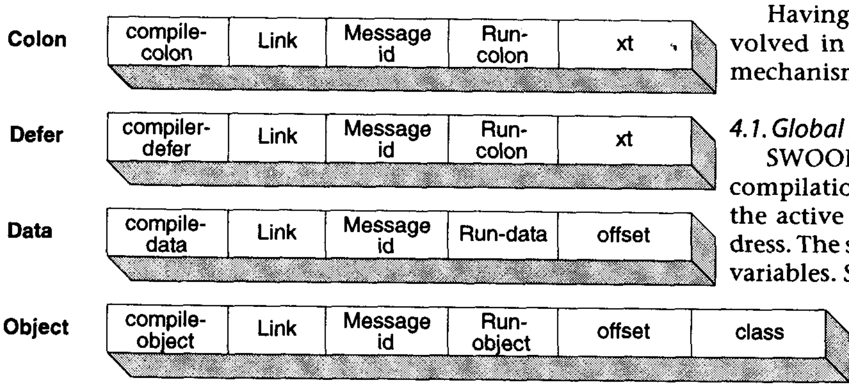


Figure Three. Data structures for various member types



A class definition is begun by `CLASS` or `SUBCLASS` and is ended by `END-CLASS`. While a class is being defined, the normal Forth interpreter/compiler is used; its behavior is modified by changing the search order to include the class namespace and the wordlist `CC-WORDS`.

All links in this system are relative, and all handles are execution tokens (*xt*). This is the only way I have found to generate a system I could guarantee to be portable across many different ANS Forth platforms. In the general case, this results in data structures that are relocatable. Specifically, in SwiftForth, this means that the objects created in the interactive system at a given address will work when saved as a DLLs, which are loaded arbitrary addresses by the operating system.

3.2. Members

Members are defined between `CLASS` and `END-CLASS`. They parallel the basic Forth constructs of variables, colon-definitions, and deferred words. The definition of a member has two parts. First is the member's name, which exists in the wordlist `MEMBERS`. The *xt* of this name is used as the *member id* when it is referenced. Second is the member's data structure. This contains information about how to compile and execute the member. Each member is of the general format shown in Figure Two; the specific format of some member types is shown in Figure Three.

The data structure associated with a member has five fields: member compiler, link, message id, member run time, and data. The data field is not of fixed length; its content depends on the programmatic expectations of the compiler and run-time routines.

The *compiler xt* is the early binding behavior for members, and the *run-time xt* is the late binding behavior. Each variety of member has a unique *compiler xt* and *run-time xt*; both expect the address of the member's data field on the stack when executed. The *message id* in each entry is the *xt* given by the member's name in the `MEMBERS` wordlist.

The data field contents vary depending on the type of member the structure represents. For data members, the data field contains the offset into the current object. For colon members, it contains the Forth *xt* which is executed to perform the actions defined for the member. In defer members, the data field also contains an *xt*, but it is only used if the defer is not extended beyond its default behavior. The data field of colon members contains the actual Forth *xt* to be compiled when the method is referenced. In object members, the data field contains both the offset in the current object of the member and the class handle of the member.

4. Implementation Strategies

Having discussed the basic syntax and data structures involved in SWOOP, we can now consider the underlying mechanisms in the system.

4.1. Global state information

SWOOP depends on two variables for its behavior during compilation and execution. `'THIS` contains the handle of the active class, and `'SELF` has the active object's data address. The system provides words to set, save, and restore these variables. See the section on system variables in Listing One.

In SwiftForth, these are implemented as user variables so that object code is re-entrant.

SWOOP maintains two wordlists associated with the compilation of classes and objects. MEMBERS contains the list of unique identifiers used to name the members of classes, and CC-WORDS contains the compiler words used to construct the definitions of the members of classes.

4.2. Classes and member identifiers

In other OOP implementations, classes are composed of instance data, methods that can act on the data, and messages corresponding to these methods that can be sent to objects derived from the class.

In SWOOP, instance data and methods are combined into a single orthogonal concept: members. Each member has a unique identifier which can be used as a message. Members exist as created *names* in the MEMBERS wordlist; each member's *xt* is its identifier. A given name will exist only once in MEMBERS; a member name always corresponds to the same identifier (i.e., *xt*), regardless of the class or context in which it is referenced.

Classes are composed of members organized in the public, protected, and private lists. The structure of a class is shown in Figure One. The member lists of a class are based on switches (VanNorman [7]) and use a member identifier as a key. A class doesn't know the names of its members, only their identifiers.

4.3. Compilation strategy

The two common models of object systems in Forth seem to be mutually exclusive: one parses and has encapsulation, the other doesn't parse but lacks information hiding.

The main strengths of the parsing model are encapsulation and information hiding. This is achieved by each word being immediate—it always executes, and it parses the next word instead of allowing the Forth interpreter to do so. This is how the context for the next word is enforced; it contains an implied search order change at each token of a multi-word phrase. An unpublished implementation by Charles Melice achieves information hiding via wordlists; each word parses and explicitly searches for its successor in a class-unique wordlist.

The main strength of the non-parsing model is its generality. Code simply pushes object addresses on the stack, modifies them, then eventually acts on these addresses. Each token is standalone, not knowing or caring what produced its input or what consumed its output. All names exist in the primary system wordlist.

The epiphany was my realization that the strengths of these models did not contradict each other. The SWOOP model is a synthesis of these two strengths. The result of this interplay of ideas is the *namespace*. A class's namespace is defined by all words in the MEMBERS wordlist whose handles match keys in the class's public, protected, or private member lists.

The executable definitions associated with entries in MEMBERS are immediate. When MEMBERS is part of the search order, a reference to a member may be found there, and it will be executed. When it is executed, it will search for a match on its handle in the list of keys in the member lists for the current class (identified by 'THIS). If a match is found, the compilation or execution *xt* associated with the matching member will be executed, depending on STATE. If there is no match in the current class, the name will be re-asserted in the input stream and the Forth interpreter will be invoked to search for it in other wordlists, handling it subsequently in normal fashion.

4.4. Compilation of classes and objects

One of my goals for SWOOP was to make the definition of classes and, in particular, the members of a class, map onto the common Forth paradigm, which meant being able to temporarily supercede the meaning of the Forth defining words. I achieved this by having a wordlist called CC-WORDS that contains all of the member-defining words, and which is only present in the search order while compiling a class.

The simplest way to discuss the compiler is to walk through its operation as a class is built. So, we define a simple class:

```
CLASS POINT
    VARIABLE X
    VARIABLE Y
    : DOT ( -- )    X @ .   Y @ . ;
END-CLASS
```

The phrase CLASS POINT creates a class data structure named POINT, links it into the CLASSES list, adds CC-WORDS and MEMBERS to the search order, and sets 'THIS and CSTATE to the handle of POINT. The variable CSTATE contains the handle of the current class being defined, and remains nonzero until END-CLASS is encountered. This is used by the various member compilers to decide what member references mean, and how to compile them.

VARIABLE X (and, likewise, Y) executes the class-defining word VARIABLE in CC-WORDS, which adds a member name to MEMBERS and to the chain of public members for POINT.

Although the colon definition DOT looks like a normal Forth definition, its critical components : and ; are highly specialized words in the CC-WORDS wordlist. This version of : searches for the name DOT in the MEMBERS wordlist; if there is one already, it uses its handle as the message ID for the member being defined. Otherwise, it constructs a name in MEMBERS (rather than with the class definitions being built), keeping its handle. Then it begins a :NONAME definition, which is terminated by the ;. This version of ; not only completes the definition, it uses its *xt* along with the message ID to construct the entry in the appropriate chain for DOT.

When a class member is referenced (such as in the reference to X in DOT), its compiler method is executed. This routine (such as COLON-METHOD and DATA-METHOD) compiles a reference to the member.

4.5. Self

Notice that, seemingly, we have inconsistent use of our members. While defining POINT, we simply reference X; while not defining POINT, we must reference an object prior to X. This problem is resolved in some systems by requiring SELF to appear as an object proxy during the definition of the class.

```
: DOT ( -- )    SELF X @ .   SELF Y @ . ;
```

This results in a more consistent syntax, but is wordy and repetitive. However, to the compiler, the reference to X is *not* ambiguous, so the explicit reference to SELF is unnecessary. While a class is being defined, SWOOP notices that X (or any other member) is indeed a reference to a member of the class being defined and *automatically* inserts SELF before the reference is compiled. This results in a simpler presentation of the routine, and makes the code inside a class look like it would if it were not part of a class definition at all.

4.6. Binding

The way a member is referenced may be decided at compile time or at run time.

If the decision is made at compile time, it is known as *early binding* and assumes that a specific, known member is being referenced. This provides for simple compilation and the best performance when executed.

If the decision is made at run time, it is known as *late binding*, which assumes that the member to be referenced is *not* known at compile time and must, therefore, be looked up at run time. This is slower than early binding because of the run-time lookup, but it is more general. Because of its interactive nature, this behavior parallels the use of the Forth interpreter to reference members.

SWOOP is primarily an early binding system, but late binding is available through two mechanisms. The first is *deferred members*, a technique that parallels the Forth concept of a deferred word. This implements the facet of late binding where the member name to be referenced is known, but the behavior is not yet determined when the reference is made. The second is the word SENDMSG, which sends an arbitrary message ID to an arbitrary object. This strategy makes it possible to, for example, send Windows message constants to a window object for processing.

Figure Four. Member data structure, showing embedded switch

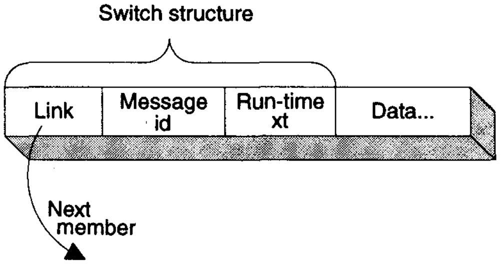


Figure Five

```
44B163    4 # EBP SUB                  83ED04
44B166    EBX 0 [ EBP] MOV             895D00
44B169    49030 [ EDI] EBX LEA         8D9F30900400
44B16F    0 [ EBX] EBX MOV             8B1B
44B171    RET                          C3
```

5. Optimization

Version 2.0 of SwiftForth (currently in beta release) will include both SWOOP and a powerful rule-based optimizing compiler. Many of its optimization strategies provide significant improvement on both the size and performance of code generated by SWOOP. For example, the sequence:

```
CLASS POINT
   VARIABLE X
   VARIABLE Y
   : DOT X ? Y ? ;
END-CLASS

CLASS RECT
   POINT BUILDS UL
   POINT BUILDS LR
END-CLASS

CLASS CUBE
   RECT BUILDS TOP
   RECT BUILDS BOT
END-CLASS

CUBE BUILDS FOO

: TEST1 ( -- )   FOO TOP UL X @ ;
```

...generates the code shown in Figure Five for TEST1, less than one machine instruction per Forth word.

6. Future enhancements

As noted, SWOOP was designed from the outset to be amenable to cross- or target-compiling. This is most obviously manifest in the separation of compile-time and run-time behaviors for members associated with a class. In a non-extensible, ROMable target, the compiler portion of the member data structure would reside in the host during compilation and interactive testing, and only the run-time support (shown in Figure Four) would reside in the target.

Note that the design of the member data structure incorporates a "switch," as described in my previous article [7]. These can be implemented extremely efficiently. Early-bound members will simply execute their *xts*; late-bound members will call the run-time switch.

7. Source code

The source code is broken into two basic parts: the preamble PRESWOOP, which Wil Baden presents elsewhere in this issue of *Forth Dimensions*, and the source code for SWOOP itself in Listing One. Listing Two provides some simple extensions to the object model, showing how to add new data types, etc.

References

1. Baden, Wil. "Simple Object-Oriented Programming," *Forth Dimensions* XX, No. 4 (1999), 14–17.
2. Entsminger, Gary. *The Tao of Objects*. Redwood City, California: M&T, 1990.
3. Ertl, Anton. "Standardizing OOF Extensions," *Forth Dimensions* XIX, No. 1 (1997), 24–25.
4. Ertl, Anton, "Yet Another Forth Objects Package," *Forth Dimensions* XIX, No. 2 (1997), 37–41.
5. McKewan, Andrew, "Object-Oriented Programming in ANS Forth," *Forth Dimensions* XVIII, No. 6 (1997), 14–29.
6. Pountain, Dick. *Object-Oriented Forth: Implementation of Data Structures*. London: Academic Press, 1987.
7. VanNorman, Rick. "A Forth Switchblade," *Forth Dimensions* XX, No. 3 (1998), 19–22.

Listing One

```
{ ==================================================================
(C) Copyright 1999 FORTH, Inc.   www.forth.com
FORTH, Inc. grants to members of the Forth Interest Group permission to use this code
providing the user clearly acknowledges FORTH, Inc. as author.  FORTH, Inc. assumes no
responsibility for the accuracy or completeness of this code.  We will greatly appre-
ciate being notified of any improvements users may make or recommend.
  ================================================================== }

{ ------------------------------------------------------------------
The following set of words have the most promise of performance
improvement if optimized with machine code. These inefficient versions
should be commented out if other versions already exist.

Classes return their xt when executed. A class's xt is considered
to be its handle. All class operations are based on this handle.

'THIS has the handle of the current class and
'SELF has the address of the current object.

THIS returns the handle of the current class and
SELF returns the address of the current data object, normally used
   only while defining a class.

>THIS writes a new value into 'THIS and
>SELF writes a new value into 'SELF.

>C C>  >S S>  are compiler macros which preserve the values of
   'THIS and 'SELF respectively.  They are used in pairs around
   code sequences.

>C C> save, set, and restore 'THIS.   "THIS >R >THIS ... R> >THIS"
>S S> save, set, and restore 'SELF.   "SELF >R >SELF ... R> >SELF"

>DATA returns a data address for the xt of an object
------------------------------------------------------------------ }

VARIABLE 'THIS
VARIABLE 'SELF

: THIS ( -- class )    'THIS @ ;
: SELF ( -- object )    'SELF @ ;

: >THIS ( class -- )    'THIS ! ;
: >SELF ( object -- )    'SELF ! ;

: >C ( class -- )
   POSTPONE THIS POSTPONE >R POSTPONE >THIS ; IMMEDIATE

: C> ( -- )
   POSTPONE R> POSTPONE >THIS ; IMMEDIATE

: >S ( object -- )
   POSTPONE SELF POSTPONE >R POSTPONE >SELF ; IMMEDIATE

: S> ( -- )
   POSTPONE R> POSTPONE >SELF ; IMMEDIATE

: >DATA ( xt -- object )    >BODY 3 CELLS + ;
```

```
{ -----------------------------------------------------------------
CSTATE has the class handle while we are defining a class.

"SELF" is a compiler tool to emplace a reference to SELF before
   each class-local item while compiling the class. This makes the
   code look nicer; instead of SELF X @ one can just say X @ .
   Pronounce this by wiggling two fingers on each hand in the air
   while saying the word SELF.

"THIS" emplaces a reference to the current class as necessary for
   resolving defer methods or simply executing a class member.
----------------------------------------------------------------- }

VARIABLE CSTATE

: "SELF" ( -- )
   CSTATE @ -EXIT  CSTATE @ THIS <> ?EXIT  POSTPONE SELF ;

: "THIS" ( -- )   CSTATE @ IF
      CSTATE @ THIS = IF  POSTPONE THIS  EXIT  THEN
   THEN  THIS POSTPONE LITERAL ;

{ -----------------------------------------------------------------
We manage our object system with two system wordlists.

CC-WORDS has the defining words used while building classes and
MEMBERS has the unique identifiers for class members.

+MEMBERS adds the MEMBERS wordlist to the search order and
-MEMBERS removes it from the search order.

+CC puts MEMBERS and CC-WORDS on the top of the search order and
-CC removes them from the search order.
----------------------------------------------------------------- }

WORDLIST: CC-WORDS
WORDLIST: MEMBERS

: +MEMBERS ( -- )   MEMBERS +ORDER ;
: -MEMBERS ( -- )   MEMBERS -ORDER ;

: +CC ( -- )   +MEMBERS CC-WORDS +ORDER ;
: -CC ( -- )   -MEMBERS CC-WORDS -ORDER ;

{ -----------------------------------------------------------------
Classes are:

   | link | xt | super | public | protected | private | size | tag |

>SUPER etc traverse this structure from the class handle.

SIZEOF returns the size of the specified class.

|CLASS| is how many cells are required to define a class.

CLASSTAG is a marker derrived from the xt of |CLASS|.
----------------------------------------------------------------- }

: BODY+ ( n "name" -- n+1 )
   CREATE DUP CELLS , 1+ DOES> @ SWAP >BODY + ;
```

```
0 BODY+ >CLINK
  BODY+ >CHANDLE
  BODY+ >SUPER
  BODY+ >PUBLIC
  BODY+ >PROTECTED
  BODY+ >PRIVATE
  BODY+ >SIZE
  BODY+ >CLASSTAG
CONSTANT |CLASS|

' |CLASS| CONSTANT CLASSTAG
' |CLASS| 1+ CONSTANT OBJTAG

: SIZEOF ( class -- n )   >SIZE @ ;

{ --------------------------------------------------------------------
Executing a named class returns its xt, which is its handle.

When a class is created, THIS will contain the handle of the class
until END-CLASS is executed.

CLASSES has the list of all known classes.
OPAQUE has 0 if new members are PUBLIC, 1 if new members are PROTECTED,
   and 2 if new members are PRIVATE.  This is an offset, translated into
   cells from >PUBLIC when used in NEW-MEMBER.

CLASS defines a new class.  With
SUBCLASS, we use
INHERITS to build a new class from an existing one.
RE-OPEN allows further refinements of a class.

SUPREME is the mother of all classes. Members may be added to
   it with extreme care.
------------------------------------------------------------------- }

CHAIN CLASSES

VARIABLE OPAQUE

: RE-OPEN   ( class -- )  DUP >THIS  CSTATE !  0 OPAQUE !  +CC ;

: (CLASS) ( -- )   CREATE-XT ( xt) DUP RE-OPEN
   CLASSES >LINK  ( xt) , |CLASS| 2 - CELLS /ALLOT CLASSTAG ,
   DOES> CELL+ @ ;

 (CLASS) SUPREME   -MEMBERS -CC

 : INHERITS ( class -- )
   HERE CELL- @ CLASSTAG <> ABORT" INHERITS must follow CLASS <name>"
   |CLASS| 1- CELLS NEGATE ALLOT   \ forget all except link.
   DUP ,                           \ point superclass field to new parent.
   DUP >PUBLIC RELINK,             \ inherit public
   DUP >PROTECTED RELINK,          \ and protected.
   0 ,                             \ never inherit private.
   DUP SIZEOF ,                    \ inherit size.
   CLASSTAG ,                      \ mark this as a class.
   DROP ;

 : CLASS ( -- )
    (CLASS) SUPREME INHERITS ;
```

```
: SUBCLASS ( class -- )
   (CLASS) INHERITS ;

{ -----------------------------------------------------------------------
COMPILE-AN-OBJECT compiles a reference that returns the object's
   address generated by the given xt and adds MEMBERS to the search order.

INTERPRET-AN-OBJECT returns an object's address.

(OBJECT) compiles or executes an object reference.

BUILDS creates a named object which looks like:
      | xt | class | data.... |

USING sets the class search order so that the MEMBERS wordlist is active.
   The net result is to allow the use of arbitrary class methods on an
   arbitrary address in memory.
---------------------------------------------------------------------- }

: COMPILE-AN-OBJECT ( addr xt -- )  >R
   @+ POSTPONE LITERAL R> COMPILE,  CELL+ @ >THIS +MEMBERS ;

: INTERPRET-AN-OBJECT ( addr xt -- addr )   >R
   @+ SWAP CELL+ @ >THIS +MEMBERS  R> EXECUTE ;

: (OBJECT) ( addr xt -- | addr )
   STATE @ IF COMPILE-AN-OBJECT ELSE INTERPRET-AN-OBJECT THEN ;

: BUILDS ( class -- )
   CREATE-XT IMMEDIATE ( xt) , OBJTAG , ( class) DUP ,  SIZEOF /ALLOT
   DOES> ['] >DATA  (OBJECT) ;

: USING ( -- )   ' DUP  >CLASSTAG @
   CLASSTAG <> ABORT" Class name must follow USING"
   >THIS +MEMBERS ; IMMEDIATE

{ -----------------------------------------------------------------------
NEW is the dynamic object constructor and
DESTROY is the corresponding destructor.
---------------------------------------------------------------------- }

: NEW ( class -- addr )
   DUP SIZEOF CELL+ CELL+ ALLOCATE THROW OBJTAG !+ SWAP !+ ;

: DESTROY ( addr -- )
   CELL- CELL- FREE THROW ;

{ -----------------------------------------------------------------------
A class has three member lists associated with it: public, protected, and
private These lists indicate which messages the class recognizes and how
to compile and/or execute the member when referenced. The format of these
lists is

   | compiler-xt | link | member handle | runtime-xt | data | ...

The data field varies from method to method. This is documented
below in the METHODS section.

The structure of the member list contains an embedded switch statement;
the link|member|xt pattern.
```

```
A member handle represents a valid member if it is in the MEMBERS
wordlist and either the public, protected, or private member list of the
current class. This represents the namespace of the class.

NEW-MEMBER builds a list entry for the current class associating the
  .member with compiler and runtime xts and a single data value.

BELONGS? returns the address of link if the member belongs to the
   current class.  BELONGS? should be coded for speed, as it is in the
   critical path for virtual methods.

PUBLIC? searches the public list,
PROTECTED? searches the protected list, and
PRIVATE? searches the private list of THIS .

CLASS-MEMBER? checks THIS class for the member. Used by RESOLVED, for
   virtual members (DEFER:) and so doesn't check PRIVATE.

VISIBLE-MEMBER? checks the member lists of THIS class for the member.
   Since this is the action of all members, it must function both
   during class compilaion and during method reference in normal
   compilation.

   If THIS is zero, it fails; no class is current to search.

   If CSTATE is non-zero, we are compiling a class.
   If CSTATE=THIS, the reference is to the current class; search
      public, protected, and private.
   If CSTATE<>THIS, the reference is to another class; search
      public and protected, but not private.

MEMBER? checks the specified class for the member id on the stack.
-------------------------------------------------------------------- }

: NEW-MEMBER ( member data runtime-xt compiler-xt -- )
   ALIGN
   ,  THIS >PUBLIC  OPAQUE @ CELLS + >LINK  ROT , , , ;

: BELONGS?   ( member list -- 'member true | member false )
   BEGIN
      DUP @ DUP WHILE +
      2DUP CELL+ @ =
   UNTIL NIP TRUE EXIT
   THEN  NIP ;

: PUBLIC? ( member -- 'member true | member 0 )
   THIS >PUBLIC BELONGS? ;

: PROTECTED? ( member -- 'member true | member 0 )
   THIS >PROTECTED BELONGS? ;

: PRIVATE? ( member -- 'member true | member 0 )
   THIS >PRIVATE BELONGS? ;

: CLASS-MEMBER? ( member -- 'member true | 0 )
   THIS IF
      PUBLIC?    DUP ?EXIT DROP
      PROTECTED? DUP ?EXIT DROP
   THEN DROP 0 ;
```

```
: VISIBLE-MEMBER? ( member -- 'member true | 0 )
   THIS IF                              \ class is selected
      PUBLIC? DUP ?EXIT DROP            \ exit if in public
      CSTATE @ IF                       \ compiling a class
         PROTECTED? DUP ?EXIT DROP      \ exit if in protected
         CSTATE @ THIS = IF             \ compiling this class
            PRIVATE? DUP ?EXIT DROP     \ exit if in private
         THEN                           \
      THEN                              \ else normal forth reference
   THEN DROP 0 ;                        \ failing

: MEMBER? ( member class -- 'member true | member 0 )
   >PUBLIC BELONGS? ;

{ ----------------------------------------------------------------------
EARLY-BINDING executes the compiler-xt of the given member, which
   compiles a reference to it according to the member type.

LATE-BINDING executes the runtime-xt of the given member.  All
   members require an object address on the stack when executing.
   This is used for runtime binding (i.e., true late binding) and
   for Forth interpreter access.

REFERENCE-MEMBER either compiles or executes a member.

?OBJECT throws if the entity whose address is on the stack is not
   an object.

SENDMSG executes the given member id in the context of the class to
   which the object belongs. This is considered to be sending a
   message.

RESOLVED looks up the member in the current class and executes it.
   This is used at runtime for late binding of virtual functions.
   We search from the class pointed to by THIS at runtime, and the
   first member match we find is executed. If no better behavior is
   defined than the initial DEFER:, we will find that and execute
   it by default.

---------------------------------------------------------------------- }

: EARLY-BINDING ( 'member -- )
   DUP 3 CELLS +  SWAP CELL - @ EXECUTE ;

: LATE-BINDING ( object 'member -- )
   OVER CELL- @ >THIS  2 CELLS + @+  EXECUTE  ;

: REFERENCE-MEMBER ( [object] 'member -- )
   STATE @ IF EARLY-BINDING ELSE
      CSTATE @ IF ( interpreting in a class definition)
         0 SWAP 2 CELLS + @+ EXECUTE
      ELSE
         LATE-BINDING  THIS 0= IF  -MEMBERS  THEN
      THEN
   THEN ;

: ?OBJECT ( object -- )
   2 CELLS - @ OBJTAG <> THROW ;

: RESOLVED ( member -- )
   CLASS-MEMBER? 0= THROW  3 CELLS + @ EXECUTE ;
```

FORTH INTEREST GROUP MAIL ORDER FORM

HOW TO ORDER: Complete form on back page and send with payment to the Forth Interest Group. All items have one price. Enter price on order form and calculate shipping & handling based on location and total.

FORTH DIMENSIONS BACK VOLUMES

A volume consists of the six issues from the volume year (May–April).

Volume 1 *Forth Dimensions* (1979–80) 101 – $85

Introduction to FIG, threaded code, TO variables, fig-Forth.

Volume 6 *Forth Dimensions* (1984–85) 106 – $85

Interactive editors, anonymous variables, list handling, integer solutions, control structures, debugging techniques, recursion, semaphores, simple I/O words, Quicksort, high-level packet communications, China FORML.

Volume 7 *Forth Dimensions* (1985–86) 107 – $65

Generic sort, Forth spreadsheet, control structures, pseudo-interrupts, number editing, Atari Forth, pretty printing, code modules, universal stack word, polynomial evaluation, F83 strings.

Volume 8 *Forth Dimensions* (1986–87) 108 – $65

Interrupt-driven serial input, database functions, TI 99/4A, XMODEM, on-line documentation, dual CFAs, random numbers, arrays, file query, Batcher's sort, screenless Forth, classes in Forth, Bresenham line-drawing algorithm, unsigned division, DOS file I/O.

Volume 9 *Forth Dimensions* (1987–88) 109 – $65

Fractal landscapes, stack error checking, perpetual date routines, headless compiler, execution security, ANS Forth meeting, computer-aided instruction, local variables, transcendental functions, education, relocatable Forth for 68000.

Volume 10 *Forth Dimensions* (1988–89) 110 – $65

dBase file access, string handling, local variables, data structures, object-oriented Forth, linear automata, standalone applications, 8250 drivers, serial data compression.

Volume 11 *Forth Dimensions* (1989–90) 111 – $45

Local variables, graphic filling algorithms, 80286 extended memory, expert systems, quaternion rotation calculation, multiprocessor Forth, double-entry bookkeeping, binary table search, phase-angle differential analyzer, sort contest.

Volume 12 *Forth Dimensions* (1990–91) 112 – $45

Floored division, stack variables, embedded control, Atari Forth, optimizing compiler, dynamic memory allocation, smart RAM, extended-precision math, interrupt handling, neural nets, Soviet Forth, arrays, metacompilation.

Volume 13 *Forth Dimensions* (1991–92) 113 – $45

Volume 14 *Forth Dimensions* (1992–93) 114 – $45

Volume 15 *Forth Dimensions* (1993–94) 115 – $45

Volume 16 *Forth Dimensions* (1994–95) 116 – $45

Volume 17 *Forth Dimensions* (1995–96) 117 – $45

NEW **Volume 18 *Forth Dimensions* (1996–97) 118 – $45**

Volume 19 *Forth Dimensions* (1997–98) 119 – $45

FORML CONFERENCE PROCEEDINGS

The annual FORML Conference is an educational forum for sharing and discussing new or unproven proposals intended to benefit Forth, and is for discussion of technical aspects of applications in Forth. Proceedings are a compilation of the papers and abstracts. FORML is an activity of the Forth Interest Group.

1981 FORML PROCEEDINGS 311 – $70
CODEless Forth machine, quadruple-precision arithmetic, overlays, executable vocabulary stack, data typing in Forth, vectored data structures, using Forth in a classroom, pyramid files, BASIC, LOGO, automatic cueing language for multimedia, NEXOS – a ROM-based multitasking operating system. *655 pp.*

1982 FORML PROCEEDINGS 312 – $65
Rockwell Forth processor, virtual execution, 32-bit Forth, ONLY for vocabularies, non-IMMEDIATE looping words, number-input wordset, I/O vectoring, recursive data structures, programmable-logic compiler. *295 pp.*

1983 FORML PROCEEDINGS 313 – $65
Non-Von Neuman machines, Forth instruction set, Chinese Forth, F83, compiler & interpreter co-routines, log & exponential function, rational arithmetic, transcendental functions in variable-precision Forth, portable file-system interface, Forth coding conventions, expert systems. *352 pp.*

1984 FORML PROCEEDINGS 314 – $65
Forth expert systems, consequent-reasoning inference engine, Zen floating point, portable graphics wordset, 32-bit Forth, HP71B Forth, NEON – object-oriented programming, decompiler design, arrays and stack variables. *378 pp.*

1986 FORML PROCEEDINGS 316 – $65
Threading techniques, Prolog, VLSI Forth microprocessor, natural-language interface, expert system shell, inference engine, multiple-inheritance system, automatic programming environment. *323 pp.*

1988 FORML PROCEEDINGS 318 – $65
Includes 1988 Australian FORML. Human interfaces, simple robotics kernel, MODUL Forth, parallel processing, programmable controllers, Prolog, simulations, language topics, hardware, Wil's workings & Ting's philosophy, Forth hardware applications, ANS Forth session, future of Forth in AI applications. *310 pp.*

1989 FORML PROCEEDINGS 319 – $65
Includes papers from '89 euroFORML. Pascal to Forth, extensible optimizer for compiling, 3D measurement with object-oriented Forth, CRC polynomials, F-PC, Harris C cross-compiler, modular approach to robotic control, RTX recompiler for on-line maintenance, modules, trainable neural nets. *433 pp.*

1992 FORML PROCEEDINGS 322 – $45
Object-oriented Forth based on classes rather than prototypes, color vision sizing processor, virtual file systems, transparent target development, signal-processing pattern classification, optimization in low-level Forth, local variables, embedded Forth, auto display of digital images, graphics package for F-PC, B-tree in Forth *200 pp.*

1993 FORML PROCEEDINGS 323 – $45
Includes papers from '92 euroForth and '93 euroForth Conferences. Forth in 32-bit protected mode, HDTV format converter, graphing functions, MIPS eForth, umbilical compilation, portable Forth engine, formal specifications of Forth, writing better Forth, Holon – a new way of Forth, FOSM – a Forth string matcher, Logo in Forth, programming productivity. *509 pp.*

1994–1995 FORML PROCEEDINGS (in one volume!) 325 – $55

Fast service by fax: 831.373.2845

BOOKS ABOUT FORTH

ALL ABOUT FORTH, 3rd ed., June 1990, Glen B. Haydon **201 – $90**

Annotated glossary of most Forth words in common use, including Forth-79, Forth-83, F-PC, MVP-Forth. Implementation examples in high-level Forth and/or 8086/88 assembler. Useful commentary given for each entry. *504 pp.*

eFORTH IMPLEMENTATION GUIDE, C.H. Ting **215 – $37**

eForth is a Forth model designed to be portable to many of the newer, more powerful processors available now and becoming available in the near future. *54 pp.* (w/disk)

Embedded Controller FORTH, 8051, William H. Payne **216 – $85**

Describes the implementation of an 8051 version of Forth. More than half the book is composed of source listings (w/ disks C050) 511 pp.

F83 SOURCE, Henry Laxen & Michael Perry **217 – $30**

A complete listing of F83, including source and shadow screens. Includes introduction on getting started. *208 pp.*

F-PC USERS MANUAL (2nd ed., V3.5) **350 – $30**

Users manual to the public-domain Forth system optimized for IBM PC/XT/AT computers. A fat, fast system with many tools. *143 pp.*

F-PC TECHNICAL REFERENCE MANUAL **351 – $45**

A must if you need to know F-PC's inner workings. *269 pp.*

THE FIRST COURSE, C.H. Ting **223 – $37**

This tutorial exposes you to the minimum set of Forth instructions needed to use Forth to solve practical problems in the shortest possible time. "...This tutorial was developed to complement *The Forth Course*, which skims too fast over elementary Forth instructions and dives too quickly into advanced topics in an upper-level college microcomputer laboratory..." A running F-PC Forth system would be very useful. *44 pp.*

THE FORTH COURSE, Richard E. Haskell **225 – $37**

This set of 11 lessons is designed to make it easy for you to learn Forth. The material was developed over several years of teaching Forth as part of a senior/graduate course in the design of embedded software computer systems at Oakland University in Rochester, Michigan. *156 pp.* (w/disk)

FORTH NOTEBOOK, Dr. C.H. Ting **232 – $37**

Good examples and applications — a great learning aid. polyFORTH is the dialect used, but some conversion advice is included. Code is well documented. 286 pp.

FORTH NOTEBOOK II, Dr. C.H. Ting **232a – $37**

Collection of research papers on various topics, such as image processing, parallel processing, and miscellaneous applications. *237 pp.*

FORTH PROGRAMMER'S HANDBOOK, **260 – $57**
Edward K. Conklin and Elizabeth D. Rather

EXCITING NEW TITLE

This reference book documents all ANS Forth wordsets (with details of more than 250 words), and describes the Forth virtual machine, implementation strategies, the impact of multitasking on program design, Forth assemblers, and coding style recommendations.

INSIDE F-83, Dr. C.H. Ting **235 – $37**

Invaluable for those using F-83. *226 pp.*

OBJECT-ORIENTED FORTH, Dick Pountain **242 – $50**

Implementation of data structures. First book to make object-oriented programming available to users of even very small home computers. *118 pp.*

STARTING FORTH (2nd ed.) Limited Reprint, Leo Brodie **245a – $50**

LIMITED TIME!

In this edition of *Starting Forth*—the most popular and complete introduction to Forth—syntax has been expanded to include the Forth-83 Standard. *(The original printing is now out of stock, but we are making available a special, limited-edition reprint with all the original content.) 346 pp.*

THINKING FORTH, Leo Brodie **255 – $35**

Back by popular demand! To program intelligently, you must first think intelligently. The bestselling author of *Starting Forth* is back, with the first guide to using Forth for applications. This book captures the philosophy of the language, showing users how to write more-readable, more-maintainable applications. Both beginning and experienced programmers will gain a better understanding and mastery of topics like decomposition, factoring, handling data, simplifying control structures, Forth style and conventions. To give you an idea of how these concepts can be applied, *Thinking Forth* contains revealing interviews with users and with Forth's creator, Charles H. Moore. Reprint of original. *272pp.*

WRITE YOUR OWN PROGRAMMING LANGUAGE USING C++, Norman Smith **270 – $35**

This book is about an application language. More specifically, it is about how to write your own custom application language. The book contains the tools necessary to begin the process, and a complete sample language implementation. (Guess what language!) Includes disk with complete source. *108 pp.*

WRITING FCODE PROGRAMS **252 – $60**

This manual is for designers of SBus interface cards and other devices that use the FCode interface language. It assumes familiarity with SBus card design requirements and Forth programming. Discusses SBus development for OpenBoot 1.0 and 2.0 systems. *414 pp.*

LEVELS OF MEMBERSHIP

Your standard membership in the Forth Interest Group brings *Forth Dimensions* and participation in FIG activities—like members-only sections of our web site, discounts, special interest groups, and more. But we hope you will consider joining the growing number of members who choose to show their increased support of FIG's mission and of Forth.

Ask about our *special incentives* for corporate and library members, or become an individual benefactor!

Company/Corporate – $125
Library – $125
Benefactor – $125
Standard – $45 (add $15 for non-U.S. delivery)

Forth Interest Group
See contact info on mail-order form, or send e-mail to:
office@forth.org