Data Structures in Forth

by Joe and Marla Bartel, Hawthorne Technology

Is Forth Out of Date?

In the early days of computing, programming languages didn't provide for complex data structures. Languages like BASIC and FORTRAN only have arrays and single variables. Personal computers have greatly affected programming style. COBOL has always had complex data structures but only a few people use COBOL on a personal computer. It was only when Pascal and C became popular that many people started using a language that supported complex data structures. It was then that they started to think about using fields and records. Forth is one of those languages that has been around since the early days of computing. Its definition does not include data structures. Does this mean that Forth can not be the best language choice for an application that should be handled with records? This article shows why Forth shouldn't be neglected when dealing with data structures.

Are Pascal and C the Only Way to Go? (Data Structures Make the Difference.)

In Forth it is necessary to be explicit about pointer usage. Languages like C or Pascal do the same things with pointers but the means are hidden from the programmer. One result of this is that in C or Pascal, large, slow access routines can be generated without the programmer being aware of it. There is nothing that can be done in C that can't be done just as well in Forth when field words have been added. Another advantage to using Forth is that it doesn't take 500K of RAM and a hard disk to do useful program development.

Some programming styles and languages fit better with a particular programming problem than others. There are many cases where data structures can organize data and make a program easier to understand and cleaner to write. Pascal and C have data structures. Unfortunately there are many times, particularly in embedded systems for using small micros, where Pascal or C is not available, or costs too much, or is too inefficient. By using the same types of structures in Forth, you can continue to use familiar structures and algorithms. By doing so you can avoid the problems that might creep up on you when you are forced to use new methods.

Forth is Dynamic so Data Sstructures (Records / Field Words) Can be Added

Forth is a very flexible language that is easy to modify. A very useful way to take advantage of Forth's flexibility is by adding field words. With field words it is easy to create any kind of record or structure you need just as you can with Pascal or C.

HTForth (Hawthorne Technology) has field words included to simplify the use of abstract structures. The field words define an address that is an offset from a base address. The base address can be a fixed location in memory or it can be a pointer that is on the stack. The definitions for field words are provided in Forth so they can be used with any Forth. The words that are used for fields in HTForth are written in assembler and generate macros that are inserted in-line in the code when they are used. The result is the same as with threaded Forth but the speed is greatly improved. The examples presented here should work with any Forth but they have not been extensively tested.

HTForth field words:

```
BFIELD   defines a 1 byte field
WFIELD  defines a 2 byte field
LFIELD  defines a 4 byte field
$FIELD  defines a variable length field
```

You can achieve the same effect of the other field words by using a size of 1, 2, or 4 with the $FIELD.

These are defining words that create new words that have special actions when used. The compile time behavior is to create a new word that will add the value that was on the stack at compile time to the value that is on the stack at run time. The other compile time action is to increment the value on the parameter stack by the size of the field that is being defined.

The declaration of a record using the field words is like the declaration of a record type. No actual memory is allocated for any variable but the offsets and order of the fields is set up for when they are used. To actually allocate memory for a record that is defined some other method must be used.

The listing of the Forth definition for the field words can be used to define field words for any implementation of Forth that does not include them. Data structures can then be created with the field words that have been defined.

How to Define and Use Records and Pointers

The main example we will use will be a symbol table for an assembler. The symbol table for a conventional assembler has an entry for each symbol the user defines. Each entry consists of several fields that must be kept together for each table entry. This group of fields would be a record that you would define in HTForth like this:

```
    0                    (  offset of first field       )
    LFIELD   LINK        (  4 byte link to next symbol  )
    WFIELD   TAG         (  2 byte tag                  )
    LFIELD   SYMVAL      (  4 byte value                )
 12 $FIELD   SYMNAME     ( 12 byte symbol name          )
    CONSTANT SYMREC      (  size of record              )
```

The field words use the top of the stack as the offset into the structure for the field currently being defined. The size of the field being defined is then added to the top item on the stack. This way a field can be inserted into a structure and all the offsets used will be automatically adjusted.

After all the fields in a record have been defined, the value on the stack can be made into a constant that will give the size of the whole structure. A constant that has the size of the structure is very useful for defining arrays of structures, allocating new instances of a structure, and for reading or writing disk files. The constant can also be used to increment or decrement a pointer into an array of the structure. By using a named constant that is produced when the structure is defined you don't have to worry about the structures' exact size and future changes will be much easier to make.

Here are some examples of how to access the elements of records:

To access the tag of the symbol entry that variable SYMPNT points to, use:

```
SYMPNT @ TAG @W
```

To allocate a fixed array of 30 symbols use:

```
30 CREATE SYMTAB SYMREC 30 * ALLOT
```

To access the value of the 5th entry of the array use:

```
SYMREC 4 * SYMTAB + SYMVAL @
```

To allocate a new data item of the type SYMREC:

```
HERE SYMREC ALLOT
```

The result is the address of a new node that then can be linked into the rest of the table.

How to Define and Use Substructures

Substructures are records that are one element of another record.

A Pascal example of this would be:

```
friends = record
          name     : string [ 32 ] ;
          address = record
              street        : string [ 32 ] ;
              city          : string [ 16 ] ;
              state         : string [ 2 ] ;
              zip           : string [ 5 ] ;
              end ;
          phone             : string [ 15 ] ;
          notes = record
             birthday       : date ;
             favorite_color : string [ 10 ] ;
             hobbies        : string [ 20 ] ;
             end ;
        end ;
```

In this example, NOTES is a substructure of FRIENDS. You could go on to another layer of substructure by making HOBBIES a record.

To define substructures, push a zero on the stack and define the fields that make up the substructure. At the end of the definition of the fields use the variable field word $FIELD followed by a name (in this case 'NOTES'), to give the substructure a name. This will make the size of NOTES equal to the total size of the substructures' fields.

A substructure that will be common to many structures can be defined separately and its size saved in a named constant. The size value that is saved can be used with $FIELD to create a name and space for the substructure in the new structure being defined. This way a common substructure doesn't have to be re-defined for each of the structures it is to be used with.

```
   0
32 $FIELD NAME
   0
   32 $FIELD STREET
   16 $FIELD CITY
    2 $FIELD STATE
    5 $FIELD ZIP
$FIELD ADDRESS
32 $FIELD PHONE
   0
   WFIELD BIRTHDAY
   BFIELD SEX
   16 $FIELD HOBBIES
$FIELD NOTES
CONSTANT FRIENDS
```

To create a record of this type for BILL and one for ART we can write:

```
CREATE BILL FRIENDS ALLOT
CREATE ART  FRIENDS ALLOT
```

To access the ZIP code for BILL we would write:

```
BILL ADDRESS ZIP
```

And to do the same for ART we write:

```
ART ADDRESS ZIP
```

The parameter stack would then have the address of the first byte of his zip code.

How to Define and Use Variant Records

Variant records, or unions (as they are sometimes called), are when two or more fields are assigned to the same location in a record. A tag field is often used to determine which of the variant records is stored in the structure. For example, a record for DINNER might have a tag to indicate if you are eating out or cooking at home. If the TAG indicates EATING OUT, the RESTAURANT LIST will be using the WHERE TO GO field in your record. If the tag indicates COOKING AT HOME, the GROCERY LIST will be stored in that field. The programmer has to keep track of the tag indication so you don't end up at the local steak house and order a dozen eggs and gallon of milk.

Figure 1:Cross assembler look up routine in Pascal

```
{ ------------------------------------------------------------ }
{ Pascal linear search of alphabetical symbol table list -- }
type
  string12       = string[ 12 ];
  symbolpointer  = ^symrec;
  symrec         = record
                     link    : symbolpointer;
                     tag     : integer;
                     value   : integer;
                     symbol  : string12;
                     end;
{ Global Variables }
var
  counter             : integer ;
  firstsym, cursym    : symbolpointer ;
{ ------------------------------------------------------------ }
function newsym( labnam : string12 ): symbolpointer;
var
  tempsym  : symbolpointer;
begin
  new(tempsym);
  newsym := tempsym;
  newsym^.link   := nil;
  newsym^.tag    := 0;
  newsym^.value  := 0;
  newsym^.symbol := labnam;
  counter := counter + 1;
end;  { newsym }
{ ------------------------------------------------------------ }
procedure lookup( var labnam : string12; var whr : symbolpointer );
var
  nextsym                       : symbolpointer;
begin
  nextsym := firstsym;
  if firstsym <> nil then begin
    whr := nil;
    while whr = nil do begin
      cursym := nextsym;
      nextsym := nextsym^.link;
      if cursym^.symbol = labnam then whr := cursym
      else if labnam < cursym^.symbol then begin
        firstsym := newsym(labnam);
        whr := firstsym;
        firstsym^.link := cursym;
      end
      else if nextsym = nil then begin
        cursym^.link := newsym(labnam);
        whr := cursym^.link;
      end
      else if labnam < nextsym^.symbol then begin
        cursym^.link := newsym(labnam);
        whr := cursym^.link;
        cursym := cursym^.link;
        cursym^.link := nextsym;
      end;
    end;
  end
  else begin
    firstsym := newsym(labnam);
    whr := firstsym;
  end;
end;  { lookup }
{ ------------------------------------------------------------ }
```

Figure 2: Cross assembler look up routine in Forth

```
( ------------------------------------------------- )
( ---------- Field words for F-83 style Forth ---------- )
: bfield                ( n - n+1 )   ( byte field )
   dup 1+ swap
   create  ,   does> @ +
   ;
: wfield                ( n - n+2 )   ( word field )
   dup 2+ swap
   create  ,   does> @ +
   ;
: lfield                ( n - n+4 )   ( long field )
   dup 4+ swap
   create  ,   does> @ +
   ;
: $field               ( n s - n+s )   ( variable field )
   over + swap
   create  ,   does> @ +
   ;

( ------------------------------------------------- )
( -- FORTH linear search of alphabetical symbol table -- )
( ------ Define a symbol table record ------------------ )
   0
   lfield      link       ( pointer to next symbol )
   wfield      tag        ( 2 byte tag )
   lfield      value      ( 4 byte value )
   12 $field   symbol     ( 12 byte symbol name )
   constant symrec     ( size of record )

   0 constant nil
   variable counter   variable firstsym
( ------------------------------------------------- )
: $<12            ( ^stringA ^stringB -- tf )
  12  0  do
    over C@  over C@         ( -- ^A ^B a b )
    <> if  leave  then
    1+ swap  1+ swap
  loop
  C@ swap  C@ swap  <
  ;
( ------------------------------------------------- )
: $=12            ( ^stringA ^stringB -- tf )
  12  0  do
    over C@  over C@         ( -- ^A ^B a b )
    <> if  leave  then
    1+ swap  1+ swap
  loop
  C@ swap  C@ =
  ;
( ------------------------------------------------- )
: newsym     ( create a new symbol )
  here symrec allot
  nil over link !
  0  over tag !
  0  over value !
  over over symbol 12 cmove
  swap drop  counter 1+!
  ;
( ------------------------------------------------- )
variable nextsym  variable labnam  variable whr
variable cursym

: lookup  ( labnam -- whr )
  labnam !   nil whr !
  firstsym @ nextsym !
  firstsym @ nil <>
  if
    begin  whr @  nil =  while
      nextsym @  cursym !
      nextsym @ link @  nextsym !
      cursym @ symbol  labnam @  $=12
      if
        cursym @  whr !
      else labnam @  cursym @ symbol  $<12
        if
          labnam @  newsym  dup  firstsym ! whr !
          cursym @  firstsym @ link !
      else  nextsym @  nil =
        if
          labnam @ newsym  dup  cursym @ link  ! whr !
      else  labnam @  nextsym @ symbol  $<12
        if
          labnam @  newsym  dup  cursym @ link !
          dup  whr !  cursym !
          nextsym @  cursym @ link !
      then then then then
    repeat
  else
    labnam @  newsym  dup  firstsym ! whr !
  then
  whr
  ;  ( lookup )
( ------------------------------------------------- )
;S
```

To define variant records DUP the size constant value on the stack and then define the first variant of the record. The variant can have multiple fields defined and can have variants of its own. After defining the variant portion use MAX to get the larger size of the structure. This will insure that there is enough space allocated for the largest variant when variants are of different sizes. This method can also be used to create alternate definitions on the same level. If two different variants are wanted as substructures use MAX before + to add the largest variant substructure to the main structure.

Sample Forth Data Structure Code

The program examples in Figures 1 and 2 are from a lookup routine for a cross assembler Marla wrote this summer. To make it more understandable to those who are new to Forth, the same routine is presented in Turbo Pascal and then in F-83 style Forth. The cross assembler was written in Pascal. We carefully translated the Pascal into Forth but have not actually run the Forth code. The data structures allow for an almost line for line translation from Turbo Pascal to Forth but the Forth version results in more compact code.

When a program is being assembled, as each symbol is encountered, the lookup routine is called and the table is searched. If the symbol is not found, it is added to the table in alphabetical order. These symbols are being stored in a singly linked list. The link field of the defined record points to the next record in the list. The lookup routine steps through the list comparing the symbol from the assembly to those stored in the list. If it finds a match for the symbol, it returns a pointer to the record containing the matched symbol. The cross assembler can then use the information in the record or add things such as flagging multiple definitions of labels, undefined symbols, or recording the value of the symbol to be used in the rest of the program.

Each time the lookup routine compares a symbol that doesn't match, it checks to see if the entry in the list is still higher in the alphabet than the symbol being searched for. If it is not, you know the symbol is not in the list because you have passed the place where it would have been. If the symbol is not found, the NEWSYM routine is used to make a new entry in the table. The new entry is then linked in to the list just before the last item it was tested against.

List before new symbol is inserted:

```
Asymbol.link → Bsymbol.link → Csymbol.link → Esymbol.link
```

List after new symbol is inserted:

```
Asymbol.link → Bsymbol.link → Csymbol.link      Esymbol.link
                                          ↓     ↑
                                       Dsymbol.link
```

By linking things into the list in this manner, the symbols are kept in alphabetical order. This has several benefits. You don't have to

(Continued on page 26)

```
/* == execute printer mode instruction == */
set_mode(int flg)  /* From select() */
{ char pchr;

/* == the modes require ESC as initial input == */
   pchr = '\033'; prnt_code(pchr);   /* escape     */

/* == turn on Near Letter Quality print == */
   if(flg == 1) {
   pchr = 0x78;   prnt_code(pchr);   /* char 'x'   */
   pchr = 0x1;    prnt_code(pchr);   /* decimal 1  */
      return;      }
/* == turn off Near Letter Quality print == */
   if(flg == 2) {
   pchr = 0x78;   prnt_code(pchr);   /* char 'x'   */
   pchr = 0x0;    prnt_code(pchr);   /* decimal 0  */
      return;      }
```

Summary

This article has covered a considerable amount of material. We have learned that two levels of file input/output exist and how to program in either. Also that the command line arguments and redirection provide considerable flexibility in reading from and writing back to disk files. And then we saw that writing to the printer is simplified by the provision of interrupt 17H.

And that's it, mostly. Of course it doesn't just happen. A fair amount of time and effort must be invested as we descend into the depths of our machine's inner workings. The reward is a great deal of satisfaction in expanding our program activity beyond the keyboard and screen. ●

Reference Listing for PROGRAMMING INPUT/OUTPUT WITH C

1. "Advanced C Primer++" by Stephen Prata,
The Waite Group
Howard W. Sams & Co, Indianapolis, IN
2nd printing 1986

This text, chapter 3, "Binary and Text File I/O" provides a very thorough coverage of the low and high level file I/O functions.

2. "TURBO C Reference Guide"
Borland International, Inc.
4585 Scotts Valley Drive
Scotts Valley, CA 95066

3. "Programmer's Problem Solver for the IBM PC, XT & AT"
by Robert Jordain
A Brady Book
Published by Prentice Hall Press
New York, NY 10023
1986

Data Structures in Forth
(Continued from page 11)

search the whole list for each symbol, only till you reach the point where the symbol should have been. Another advantage is that the symbol table does not have to be sorted into alphabetical order to print it out at the end of the assembly. This can be useful in many lists. You have to search the list anyway and you can save the sort time and space that would be needed to change the order of things later.

In the Forth example, the word $<12 is used to test if a 12 character string is less than another. The word $=12 tests to see if two 12 character strings are equal. The word newsym allocates space for a new symbol node.

No matter what language you choose, you should try to write good, understandable code. Try to keep things simple. Don't be lazy or try to save a couple of bytes or a few lines of code at the expense of being understandable. Design your code so it can be changed without falling apart. Spend some time and effort on finding good ways of doing what you are trying to do. After you write something, look at it again in a few days. Most people will not send out the first draft of a letter without reading it and trying to make it better, so do the same with your code. A little effort in this area will be noticed by your boss or your peers as well as save you effort figuring out your own code. It can pay off in many indirect ways. ●