Hi out there, all you bugs in Rhug. I hope you are having more time with your computers than most of the members of the group I've talked with lately. It seems like there is a lot coming up with the new news from Zenith/Heath on the Z line of computers and also on the local front. At the next meeting we are planning a discussion on standard language use among the group and Pascal (HT+ and Lucidata) has the upper hand. If you have thoughts on this show up at the July meeting or forever hold your disk because we are planning a possible group buy of a language and also planning possible group programming and instruction if the interest is here. Other users' groups interested in a possible buy of this type please contact us! Please note the newsletter is late and will be until other arrangements can be made for editor time. For now the newsletter will be bi-monthly. How about some articles from you guys (I thought 300 Baud was slow).

**LANGUAGE DIFFERENCES (Part 2)**

by Jim Scott

INTRODUCTION

This is the fourth in a series of articles on the principles of programming, and the second on the subject of "Language Differences". Last month I discussed the distinction between a language and the various implementations of a language, in particular the compiler versus the interpreter. I explained that interpreters generally provide for interactive testing, which is especially advantageous to the beginning programmer, whereas compilers generally produce programs which run much faster and often require less memory than those that run under an interpreter. Finally, I discussed the idea of high-level versus low-level languages.

STRUCTURED VERSUS UNSTRUCTURED LANGUAGES

A program which is structured is likely to be easy to understand when another programmer reads it. Some languages make it very easy to write structured programs, and other languages make it very awkward. I will use the term "structured language" to mean a language that makes it easy and natural to write structured programs.

A structured program is one which can be broken down into pieces that take only three forms:

1. Sequence: a simple sequence of statements, to be executed one after the other; for example,

```
statement1
statement2
statement3
statement4
```

2. Alternative: a test for a certain condition, followed by a separate sequence of statements to be executed for each of the possible results of the test; for example (in an arbitrary Pascal-like language),
3. Loop: a sequence of statements to be repeatedly executed until a certain condition is met; for example,

```
REPEAT
  statement1
  statement2
  statement3
UNTIL condition
```

It should be understood that these forms can be used recursively; for example, the following is structured:

```
statement1
REPEAT
  IF condition1
    THEN statement2
    REPEAT
      statement3
    UNTIL condition2
  ELSE IF condition3
    THEN statement4
    ELSE statement5
UNTIL condition4
statement6
```

Also, the Alternative and Loop forms given have different representations in different languages, and may have multiple representations in some languages. The closest thing to the loop in Basic is the FOR statement. Pascal can loop with the REPEAT, WHILE, and FOR statements.

It is possible to do structured programming in just about any language. But it is difficult and unnatural in many languages. A good rule of thumb is this: a structured language is one in which it is possible to implement the three forms given above without resorting to the GOTO statement or its equivalent (e.g., a jump or branch instruction).

Consider the Alternative structure, with multiple statements in each of the two parts (THEN and ELSE). A structured language will have a way of grouping a simple sequence to show that it is to be treated as a unit; an unstructured language, lacking a way of grouping the statements, will typically require one or more GOTO's to direct control to the appropriate set of statements based on the result of the condition test.

Here is an example, first in Pascal:

```
IF I = IMAX
  THEN BEGIN
    X := Y + Z;
    COUNT := 0
  END
ELSE BEGIN
  X := 1;
  COUNT := COUNT + 1
END;
```
Note that Pascal uses BEGIN and END to group the statements that belong to the THEN and ELSE parts of the IF statement. Now let’s look at the same thing in Basic:

```
100 IF I = IMAX THEN 300
200 GOTO 600
300 X = Y + 1
400 COUNT = 0
500 GOTO 800
600 X = 1
700 COUNT = COUNT + 1
800 . . .
```

Since the THEN part of the Basic IF statement can take only one statement, it has to be a GOTO to transfer to the group of statements we really want to execute. Likewise for ELSE. Note that some definitions of Basic have ELSE and some don’t, but it makes little difference; this example assumes there is no ELSE keyword.

Third, here is the same thing in 8080 assembler language:

```
LDA 1 Load I and compare it to IMAX.
LXI H,IMAX
CMP M
JZ XISI Jump if I = IMAX.
JMP XNOT1 Jump if I < IMAX.
XISI LDA Y Compute X = Y + 1.
LXI H,Y
ADD M
STA X
LXI H,COUNT Set COUNT to 0.
MVI M,0
JMP NEXT
XNOT1 LXI H,X Set X to 1.
MVI M,1
LDA COUNT Compute COUNT = COUNT + 1
ADI 1
STA COUNT
NEXT . . .
```

Pascal, C, PL/I, and Algol are structured languages. Basic, Fortran, APL, and assembler are not structured. Cobol and Forth are sort of structured; they allow grouping of statements in the THEN and ELSE parts of IF statements, but not in an unlimited way. Some nested IF structures can’t be expressed in a straightforward manner in these two languages. Furthermore, Cobol’s version of the Loop, its PERFORM statement, doesn’t surround the statement or statements to be repeated; they must be located in a separate part of the program, and given a name which the PERFORM refers to.

(I should mention at this point that I have been reading about Forth and C for this article, but I lack experience with them. Therefore, my characterizations of them are sitting ducks for more knowledgeable readers.)

As the examples indicate, structured programming can be done in any language, if the proper techniques are used. I will give this topic more coverage in a future article.
WRITABILITY AND READABILITY

I will use the term "writability" to refer collectively to those aspects of a language which make it easier to write programs. Once you have a concept in your head for a limited part of a program, say an algebraic formula to be computed, it may be quite straightforward to translate that concept into the language you are working with. If so, you are working with a writable language. On the other hand, if you have to get out sheets of scrap paper and diagram the flow of data, or if you have to keep trying variations of programs statements and hand-checking them before you find one that seems to work, then you are not working with a writable language. (Or possibly you have not really finished the program design yet, and you are trying to code too big a lump.)

Likewise, I will use the term "readability" to indicate the ease with which well-written programs in a given language can be read and understood by a programmer other than the author of the program, or by the original programmer several weeks or months after writing the programs.

Of course, there are a number of factors involved here beyond those which are inherent in a given language, particularly when it comes to readability. A person who understands and is used to the language itself has a clear advantage. And a program which is badly written may be unreadable regardless of the language. The proper use of comments, indentation, blank lines, and variable names can go a long way toward making a program readable in any language.

There is also a controversial aspect. Is a language better if it uses more "natural" forms and notations? Or are some "unnatural" forms worth getting used to because they help the programming process in the end? What determines what is natural, anyway?

Let's look at an example. Forth uses reverse Polish notation and a stack. E.g., the expression

\[ \text{1 2 + 3 * 4 /} \]

is the Forth equivalent of the algebraic expression

\[ ((1 + 2) * 3) / 4 \]

In most other high-level languages, this would be written

\[ \text{((1 + 2) + 3) / 4} \]

For me, the latter is the much more natural notation.

(In fact, I would like to add a highly personal comment, and hope that Forth-lovers will settle for dismissing me as a harmless buffoon: I suspect that there are only two advantages to the reverse Polish notation and the stack as used by Forth: (1) They made it easier on the guy who had to write the software to implement the language, and (2) they appeal to the type of programmer who feels that the puzzle-solving aspect of programming is more important than productivity. Remember, as I mentioned above, I have no experience with Forth, so my remarks should be consumed a la sodium chloride.)

Perhaps for the experienced Forth programmer, Forth notation is quite natural. Perhaps it all depends on which programming languages you learned first.

In C, the statement

\[ ++\text{NUM} \]
means that the variable NUM is to be incremented; that is, 1 is to be added to it. In most other languages, this statement would be written something like the following:

\[ NUM = NUM + 1 \]

In fact, this statement is also valid in C. Now which is more natural? Having been brought up on Fortran and PL/I, I am more comfortable with the second form. Those who are disturbed by the fact that this form looks like an algebraic equation equivalent to the ridiculous question "what number is equal to itself plus one?" may prefer the Pascal version

\[ NUM := NUM + 1 \]

or the APL notation

\[ NUM \leftarrow NUM + 1 \]

(where "<-" is the APL left-arrow symbol).

Having given sufficient notice that determination of writability and readability is a personal judgment, I offer the following opinions:

Pascal, PL/I, C, Algol, Cobol, and APL are all writable languages. The first five are very similar languages in many ways; there are important differences among them, to be sure, but if you have seen one of these languages you would not be totally lost when you see the others.

But APL is a special case. (Remember that in the previous article I classified it as a "very high-level" language.) It has a lot of special symbols; many of them are very powerful, and may take the place of a large number of statements in an ordinary high-level language. But learning APL's set of operators takes time; until their use comes naturally, you may be tempted to limit yourself to that subset of APL that corresponds to other languages. Once you have APL in your head, it is writable because it is very powerful (in spite of being unstructured).

Unwritable languages are not writable, because they are just the opposite of being powerful. Forth (for me) is not writable, because it is unnatural.

Basic and Fortran are in between. The fact that they are unstructured makes them harder to program in than those I listed as writable. But they are more nearly writable than assembler or Forth.

Those that are writable are not necessarily readable. In some languages, it is easy enough to keep the logic straight during the programming process; but if the language makes the program obscure, reading and understanding the program may require almost as much effort as writing it did.

In my opinion, Pascal, Cobol, Algol, and PL/I are readable languages. Again, this means that programs written in these languages may be readable if the programmer used those style techniques that enhance readability, such as indentation, comments, blank lines, and meaningful variable names. Unreadable programs can be written in any language.

I am not sure how to classify C, having no experience with it, but from what I have seen it is semi-readable. It may be quite readable to those who are used to it.

I classify Basic, Fortran, Forth, APL, and assembler as not readable. But good programming style can help a lot. Assembler programmers seem to recognize the need to strive for readability more often than others. Most programs I have seen in Basic and APL were written with little apparent concern for readability. Both need heavy use of comments, but most programmers seem to prefer filling out each line with multiple statements, thereby compounding the readability problem.
The following table summarizes my judgments of the languages that are generally available for microcomputers:

<table>
<thead>
<tr>
<th>Compiled</th>
<th>Interpreted</th>
<th>Level</th>
<th>Structured</th>
<th>Writable</th>
<th>Readable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algol</td>
<td>yes</td>
<td>high</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>APL</td>
<td>yes</td>
<td>very high</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Assembler</td>
<td>yes</td>
<td>low</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Basic</td>
<td>yes</td>
<td>high</td>
<td>no</td>
<td>seal</td>
<td>no</td>
</tr>
<tr>
<td>C</td>
<td>yes</td>
<td>high</td>
<td>yes</td>
<td>yes</td>
<td>seal</td>
</tr>
<tr>
<td>Cobol</td>
<td>yes</td>
<td>high</td>
<td>seal</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Forth</td>
<td>yes</td>
<td>high</td>
<td>seal</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Fortran</td>
<td>yes</td>
<td>high</td>
<td>no</td>
<td>seal</td>
<td>no</td>
</tr>
<tr>
<td>Pascal</td>
<td>yes</td>
<td>seal*</td>
<td>high</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>PL/I</td>
<td>yes</td>
<td>high</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

*UCSD Pascal is compiled to produce F-code, which is then executed under an interpreter.

If you were to choose a language to use for your programming, which one should it be? Well, to start with, there always seem to be a few applications that absolutely require assembler language. But assembler has nothing to offer in the other cases. So for general programming, use a high-level (or very high-level) language.

Since Basic is much more widely available than the other high-level languages for microcomputers, there will be cases where it is the only language available. But if it is a question of Basic being free versus other languages costing money, consider how much it might be worth to you to avoid the many disadvantages of Basic.

If you are a beginning programmer and need the ability to do interactive testing (see previous article), you must choose among APL, Basic, and Forth. (As far as I have been able to determine, UCSD Pascal's F-code interpreter isn't any help in this area.) Assuming that all three are available to you, I would recommend APL. The main problem with APL for beginning programmers is that it provides little preparation for learning other, more orthodox, high-level languages. Then again, APL might be easier to learn if you haven't already gotten used to another language.

If you are beyond the beginner stage, so that you don't need an interpreter holding your hand, I would recommend choosing among Algol (which is not widely available), APL, C, Cobol, Pascal, or PL/I. Don't overlook languages not included here, such as Lisp, which I don't know enough about to classify. If several of these are available to you, you should weigh the following factors:

Cost - How much must you pay for the compiler or interpreter, plus any other software or hardware required?

Memory - How big is the compiler or interpreter, and how big will the programs you write be? (In the case of a compiler, this means the size of the executable program; in the case of an interpreter, it means the size of the source program, since it will
have to be in memory whenever it is executed.)

Execution speed - How fast will your programs run? APL and UCSD Pascal programs will probably run noticeably slower, since they interpreted. The others, including compiler implementations of Pascal, will probably run as fast as you want them to.

Popularity - How much software is available in source program form for each language?

Vendor - What is the reputation of the software company providing the language implementation? Will you be able to get updates and bug fixes? Will there be any problems if you want to sell programs you write using their implementation of the language?

and finally...

Nature of the language - Do you want a very high-level language (APL)? Or do you want a structured, writable, readable language?

HDOS DIRECTORY PRIMER - by Carlos Chafin

The past couple of months have proven to be a real obstacle course for me as far as finding my usual time at the computer. At first it was the wedding, honeymoon and then the big crash, nothing to do with my new wife but rather the disk drive and H-8 disk control in my computer. To make a long story short, my computer wiped a disk directory off a disk which contained my only updated version of a new finance program I have been converting to assembler for the past two months. My first feeling was "Hey, No Prob, I can fix this", I can now say "No Way Hosier". There are however many repairable directory problems in HDOS if you own a good disk dump utility program and hold a reasonal understanding of the HDOS disk directory structure. I can just wish you more luck than I had this time around.

Let's cover a little background of DOS operation in general. First point is both CP/M and HDOS offer dynamic file allocation. This simply means that these systems are capable of putting a multi-record file on the disk media where ever it will fit and thus not necessarily in a consecutive manner. In short the disk directory must not only know file names but where every record in that file might be on the media where it lives.

I am sure the users of HDOS have noticed at one time or another the presents of the three disk files labeled DIRECT.SYS, RG1.SYS and GRT.SYS on every disk they initialize. If DIRECT.SYS is the disk directory what are these other two files? RG1.SYS acts as a bad sector (CLUSTER) lock out information table and GRT.SYS acts as the location table for HDOS to find consecutive locations or non-consecutive file position allocation on the disk. To examine this a little closer we must be aware of a couple of the ways and means by which HDOS does the job it does on the various disk systems it will support. One point is HDOS was designed to operate in a system with limited disk space and the RG1 and GRT files both handle disk storage in clusters of data which always divide down to 200 clusters regardless of the disk size. After all, 200 bytes will fit into one sector (it is less than 256) and as we all know every sector counts. On the H-17 disk system with the siemens drives (40 trk - 400 sector) each cluster of data is 2 sectors (512 bytes) in size. Where as the TANDON 80 trk dual side drives receive a allocation of 8 sectors and a resulting 2048 byte cluster size.

With this knowledge under our hat what does the RG1 file look like? Starting on track one (40 trk format) byte one in the RG1 file states the status of the sector pair (0,1) and whether that cluster has been lockout during an INIT on the disk involved. An FF indicates locked. The RG1 file lives on one sector located at track 1 sector 0 on the 40 trk system. The first 200 bytes of the file are used and by the way don't be surprised to see FF's (hex) on clusters 2,3, and 4 as they are read directly by the boot process before HDOS signs on. RG1 stands for RESERVED TABLE GROUP.

The GRT Group Relocation Table is set up in a similar fashion (first byte representing files starting location) as pointed two from the directory information file DIRECT.SYS. This table is a linked list lookup table and thus can only look forward. For example, the directory knows the file name and extension and the beginning address of the file in question. It also knows the end address of the end of the file, how long the file is and date and other data on the file. So let's say the directory says go to the GRT and see where this file lives after we get past its first 512 bytes. If you increment to the next byte in GRT's table and go read that sector your following the vary steps HDOS does as it reads in a multi sector file. At this point go on to the next byte in the table (from the position of the files last cluster read) until you get to the last cluster (remember the directory knows where that is). The following is the layout of the HDOS DIRECT.SYS.
The DIRECT.SYS file is set up as follows:

Bytes 1-8 = File name
4-11 = Extension
12 = Project (not supported at this time)
13 = Version
14 = Cluster Factor (size)
15 = Flags
16 = Nothing
17 = First Cluster location
18 = Last Cluster location
19 = Last sector index
20,21 = File Creation Date (augustin)
22,23 = File Update (augustin)

In closing I would like to say, the utility program UDUMP as sold by the Keyboard Studio Inc. is my choice of a top notch dump utility and is a worth while addition to you're software tool box for HDOS. It uses the screen edit approach and reverse video accents changes before you write them back to disk. It also keeps track of where you are in a file (how many sectors in) and where the file begins and ends on the disk by using the BRT and RBT tables if they are intact. If you haven't guessed by now, mine weren't and 10 sectors of directory are still in the ether.

* * * * * * * * * * * * * * * * *

MEETING NOTICE - The July meeting of RHUG will be at ALPHA AUDIO and should be a good meeting to attend because of several items in the horizon. Time is 7:30 the date Monday the 12th and the door code is 4302. After you enter it come to the third floor conference room. Hope to see you there. CARLOS CHAFIN (This month's editor).

(c) 1982 Richmond Heath User's Group
Contents may only be reproduced with proper credit to both the individual author and the Richmond Heath User's Group, but are not to be reproduced for commercial use without the express consent of RHUG.