PORT FE
PORT FE
PORT FE

Sorcerer Users' Group (Toronto) Newsletter
200 Balsam Ave., Toronto, Ont., M4E 3C3
Volume 1 Number 9
November 1980

EDITOR'S TURN

It gives me great pleasure to announce that we have obtained a meeting place for November and December. The next meeting will be held on Wednesday November 12 at 7:00 pm at the University Settlement, 23 Grange Rd., Toronto. This is located behind the Ontario College of Art. We also have the place for Wednesday December 10 so mark it down. The December meeting is very important and I hope that everybody will be able to come to that one.

Last month I stated that I was unaware of what David LeBlanc, our librarian, was intending to do next year. Well, he has informed me that he has a new job in Kitchener/Waterloo and is moving out there this week. He, therefore, will not be able to retain his esteemed position and I would like to take this opportunity to thank him for his work in helping us set up a data library.

It is becoming increasingly harder to write newsletters for this group since there is such a wide diversity of interests. I can easily write about the things that interest me and that I have researched extensively but I am not qualified to write about those things which I know nothing about, which is just about everything. When I compare the areas of my expertise to the many uses and interests that the people in this group put there Sorcerers to, it is barely a fractional part and almost insignificant. In the past, I have written mainly about the most basic features of the Sorcerer such as the I/O primitives, the Monitor Work Area and the Basic Work Area but these topics have pretty well been exhausted by now. Most of you in the group have had your computers for some time and are fairly well versed in these areas. I have also written many criticisms of computer users and vendors as well as software vendors. It is not a valuable or rewarding exercise to continue this line of thought. You all know how it is out there and how I feel about tactics and schemes used to rip off Joe Computeruser. I have also philosophised about the future of the group to the point where it starts to sound apocryphic on one hand and pessimistic on the other so I'll pass on that this month.

Since the interests are so varied and in many cases specialized, it is very difficult to find any published works let alone stand alone software programmes on your field. This means basically that you will have to do all your own experimentation, programming and discussing. What I have decided to do then, is list for you what I feel are the ten most important steps to writing good programmes, to programming efficiency and good organization. These are guidelines that I have developed for myself (painfully) from my experience in programming. I hope that all of you can benefit from them.

1) Throw away flow charts!-

Flow charts are not used very often any more because they are harder to design and more complex to understand than the programmes themselves. They are not useful in terms of a structured programming approach and actually hinder structured design. The results obtained from flow charts are not worth the effort that goes into making them. Instead, it is much more fruitful to simply write down on a piece of paper, in English or whatever language you speak, all the basic steps that a programme must go through. This better represents the logical flow that you have previously thought about and organizes these thoughts in a hardcopy form. It is easier to read and re-understand when you are doing the actual coding and yields better results for a less tedious effort. This also ties in with step 2.

2) Write modular segments!-

Never attempt to sit down and write a programme from beginning to end as one piece. It is impossible to debug, to understand and you will not be able to introduce new functions at a later date without a major, if not a complete rewrite. Instead, start at the goal end of the programme (on paper) and break it down into function modules. These function modules should be large, major functions. Then take each one of these modules individually and break them down into their component function modules. Keep breaking the programme down in this fashion until you reach the primitive function modules which cannot be easily broken down any further. Then write the code for each of the primitive modules as a stand alone subroutine by passing whatever parameters to the routines that are required and returning the necessary information (see step 4). These primitives should not be designed to work for only one larger routine, but should be capable of performing the required functions for any set of data passed to it. Then simply put these modules together (preferrably in the order of their breakdown with the largest, most global modules at the beginning and the most primitive modules at the end) by linking them with the proper subroutine calls and passing the correct parameters.

3) Use GOTO's sparingly (if at all)!-

Even in Basic, Assembler and other non-structured programming languages,you can almost eliminate GOTO's. The major drawback with them is that they transfer the control flow of the programme to unknown areas and interrupt the stepwise design of the programme. They obviously are used to jump out of a routine with no return information, that is, nobody knows where they came from and that makes it very difficult to debug. If used, they should only have a scope within the present subroutine and never bypass RETURNs or return information.

4) Special variable names!-

If you are using a structured language that keeps track of global and local variable names and separates them, such as 'C' does, then you don't have too much problem here. But if you are using a non-structured language, such as BASIC, then you should have specially coded names that are meaningful to you and will not interfere with or duplicate other variable names used in the programme. A separate set of names should be

kept for subroutines and for passing parameters to them. It is also advisable too always reset your variables in a subroutine to ensure that you are not working with improper data. Consider the following:

```
100 PARAM1=PRINC1
110 PARAM2=INT1
120 PARAM3=PMT1
130 PARAM4=BAL1
140 GOSUB 1000:REM MORTGAGE SUBROUTINE
150 BAL1=PARAM4:REM RETURNED VALUE
160 PARAM1=PRINC2
170 PARAM2=INT2
180 PARAM3=PMT2
190 PARAM4=BAL2
200 GOSUB 1000:REM MORTGAGE SUBROUTINE
210 BAL2=PARAM4:REM RETURNED VALUE

990 REM MORTGAGE SUBROUTINE
995 REM SET ALL VARIABLES TO ZERO
1000 MORTSUB1=0:MORTSUB2=0:MORTSUB3=0:MORTSUB4=0
1010 REM ASSIGN PARAMETERS
1020 MORTSUB1=PARAM1
1030 MORTSUB2=PARAM2
1040 MORTSUB3=PARAM3
1050 MORTSUB4=(MORTSUB1*MORTSUB2)/MORTSUB3
1060 PARAM4=MORTSUB4:REM RETURNED VALUE
1070 RETURN
```

This programme reserves the names PARAM# for passing parameters to subroutines. They can always be used to pass values to any routine in the programme without being stepped on. Furthermore, there is no conflict with the names used in the subroutine. It can take data from any call without being locked in to a particular calling routine. That is, a separate subroutine does not have to be set up to handle the data used in the second call to the subroutine because the variable names are independant.

5) Control the flow:-

This is probably the most important point. If the flow of the programme is not under your control all the time, then you will spend endless weeks of debugging trying to find out where in hell you are coming from and where you are going to. With a modular design, your subroutines may not compute the correct values (which is simple to correct) but at least you know where the programme has bombed and where the pointers are at all times. The following structure is suggested in order to simplify where in a programme a particular module is located:

1) Main controller- organizes the large modules and controls branching.
2) Main modules- the larger but functional modules eg. add, sort, delete, etc.
3) Main subroutines- these routines are specifically tailored to the main modules.
4) Service subroutines- these routines are common to many of the large modules and provide global services.

5) I/O subroutines- handles all special I/O formattins and filterins.

Do not allow fallthrough between subroutines-

```
100 REM SUBROUTINE A
110 REM VARIOUS STEPS FOLLOW
180 RETURN
190 STOP:REM DON'T ALLOW FALLTHROUGH
200 REM NEXT SUBROUTINE
```

6) Sinsle Entry/Sinsle Exit:-

Never enter a subroutine at any point other than the besinnins and never leave from any point other than the end. This keeps perfect control over what the subroutine does, what values it uses and what values it returns. It also means that if you need to make some chanses to the subroutine, you know that it always enters at this point and always leaves at that point. If a subroutine enters and leaves from different points then you will not be able to make chanses to the routine for one call without screwins everythins up for other calls. This may require a GOTO within the subroutine but it does not violate the scope rule.

```
100 REM VARIOUS STEPS
120
130
140 IF X=3 GOTO 180
150 REM MORE STEPS
160
170
180 REM RETURN VALUE ASSIGNMENTS
190 RETURN
195 STOP
```

This prosramme does not allow the subroutine to return from statement 140. If some chanses are required to be made to the return value assisnments or some new steps introduced, then you know that these can be picked up safely for all calls at statement 180. If the routine RETURN'ed from various points, then this could not be done without introducins traps at every RETURN statement.

7) Avoid complicated Boolean expressions:-

When you write a complicated Boolean expression, I suarantee that it will be the one and only time that you will ever understand it. Unless you want to write a two page documentation on it, avoid statements like-

```
150 IF((X=3 AND Y=2) OR (X=3 AND ((Y=4 AND Z=5) OR T=7) OR NOT(Y=5
AND S=9) OR G)) THEN X=X+Y+Z
```

Statements like the above will surely make you a much more miserable and anti-social person.

8) Use loop features liberally:-

The power of a computer is unleashed through the ability to perform repetitive steps at high speed. That's what the computer is all about and that's what loops are all about. In order to maintain maximum efficiency and speed, keep unnecessary steps out of the loops.

```
100 REM BAD LOOP
110 FOR I=1 TO Z
120 X=3
130 A(I)=2*X+I
140 NEXT I

200 REM BETTER LOOP
210 X=3
220 FOR I=1 TO Z
230 A(I)=2*X+I
240 NEXT I

300 REM EVEN BETTER LOOP
310 X=6
320 FOR I=1 TO Z
330 A(I)=X+I
340 NEXT I

400 REM BEST LOOP
410 FOR I=1 TO Z
420 A(I)=6+I
430 NEXT I
```

9) Get rid of old versions:-

The trend with most people I have programmed with is to keep 5 copies of every development version of a programme that they have ever worked on. The unfortunate thing, however, is that whenever they have tried to show me a programme they wrote, they can never find the latest version. I am sure they will eventually, but life is too short. (I have probably just lost all of my friends because they know who they are.) If you are afraid of losing your data or getting a poor copy (and you should because it happens) then you should make two copies of the programme on two different tapes or disks. However, you should get rid of the previous copies at the same time. The only copies of the programme you should have around should all be the same version - the latest one. Given the chance, you will confuse yourself beyond reclaimation. It's the nature of mankind. Don't be upset about it, just accept it and adjust your habits accordingly.

10) Make it all readable:-

If you would just take the time to pretty-print all your programmes and use a few REM statements and select (if possible) variable names that are meaningful, then you should never be faced with the situation where you can't understand your own logic when looking at an old programme. Recall the 'C' programme in the Sept. issue. This programme uses variable names that fully represent the intended functions. I could come back to

this programme four years down the road and be able to read the programme directly and understand it. Now, not all languages use names that are this convenient, but with a little bit of formatted printing and a few REM statements, this problem can be easily overcome. In extreme cases, it is not a crime to have a separate sheet of handwritten documentation to refresh your memory. In any case, a finished programme should be fully documented as to how to use the package, even if it is only intended for your own use. If you don't use the programme for a few months, you will surely forget the commands and functions that you built into it. I have written programmes that I forgot how to use and had to disassemble the thing to find out what functions and formats I built into it. If you can't remember your own programme, nobody else will either. The habit to write the code as quickly as possible in order to get it running with the least amount of typing is just laziness. If you have to, learn how to type faster. It's more productive than having to take a programme apart to find out how it runs.

These ten points work very well for me. They may not be perfectly suited for you but they can form the basis for developing your own system of organization. It may not seem like it, but I guarantee you that if you are organized both in your filing and your programming, you will be writing better programmes, faster and more efficiently than you would otherwise.

GENERAL NEWS

-Last month I told you that I recieved information on wargaming from Australia. That information is placed in the library folder that Duncan has and will be available to view at the next meeting. In the meantime, the writer is anxiously looking forward to communicating with anybody who is interested in this field and would like to recieve letters from you. If you are interested in writing, send your letters to

Adrian Pett
10 Burgundy Drive
Doncaster
Victoria, 3108
Australia

-Get your nominations in now if you are interested in being an officer for the group. We need a president, a vice-president, a secretary/treasurer and a librarian.